# 项目构造：



spring

# Spring分模块配置

就是在一个配置文件中引入另一个配置文件，就叫分模块配置

## EG：



# Spring

## 什么是Spring

Spring是一个开源框架，Spring是于2003 年兴起的一个轻量级的Java 开发框架，由Rod Johnson在其著作Expert One-On-One J2EE Development and Design中阐述的部分理念和原型衍生而来。它是为了解决企业应用开发的复杂性而创建的。框架的主要优势之一就是其分层架构，分层架构允许使用者选择使用哪一个组件，同时为J2EE应用程序开发提供集成的框架。Spring使用基本的JavaBean来完成以前只可能由EJB完成的事情。然而，Spring的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何Java应用都可以从Spring中受益。Spring的核心是控制反转（IoC）和面向切面（AOP）。简单来说，**Spring**是一个分层的**JavaSE/EEfull-stack(**一站式**)** 轻量级开源框架。

## Bean元素

### 概念：

用来描述需要交给spring容器进行管理的对象

Eg:（这个是在xml配置文件中的）



### 属性：

**Class：**是对应的类

**Id：**很久以前的属性，是不可以重复的

**Name：**是为了获取对象的时候根据该名字来获取对象（可以重复）

（尽可能的用Name属性）

**Scope：**明确Bean的作用范围（singleton单例的）（prototype多例的）

是嵌套在beans标签里面的

Scope属性还有request、session，表示的是分别与相应的生命周期一致。一旦生命周期结束就 从spring中移出（不常用）

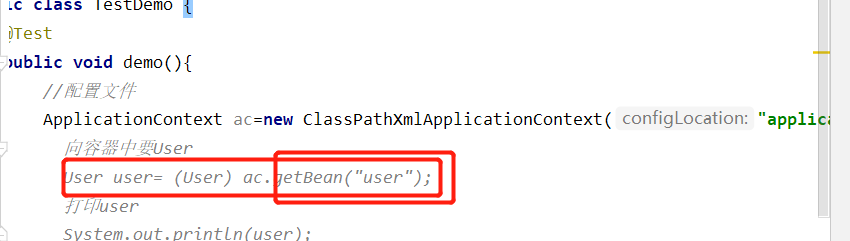
#### 注意：如果是 单例的话，A==B，返回的将是true

单例对象：表示在spring容器中只会存在一个实例**（默认）**

多例对象：被标识为多例的对象，每次创建都是新的对象

（action就必须要声明成多例的）

### 获取spring容器管理的对象：

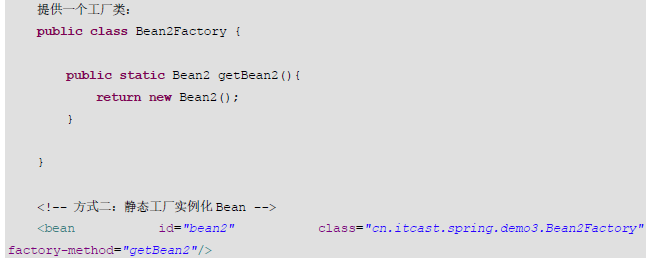


### Spring生成Bean的时候3种方式：

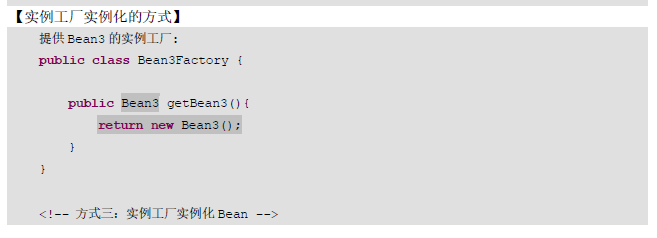
**空参构造生成：**

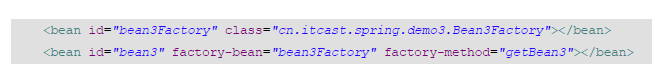
就是构造方法中没有参数

**静态工厂实例化：**



**实例化工厂实例化：**





### 生命周期方法属性：（了解）

初始化的时候的方法：

Init-method

Destory-method

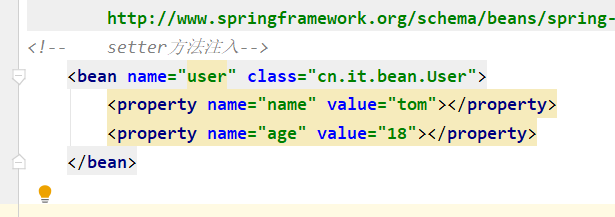
这个在user对象中定义相应的方法

要看到效果，就在代码处把容器关了就好

applicationContext.close();

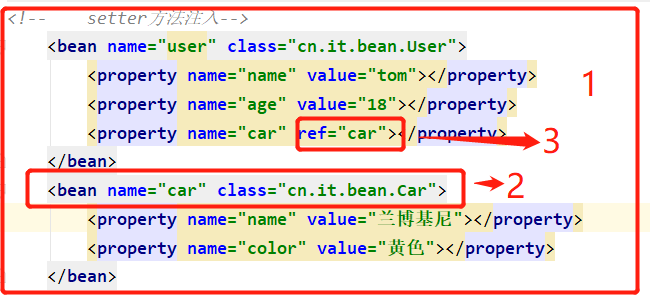
# Spring属性注入：

## Set方法注入



就是利用setter、getter方法

## 对象属性的注入：



注入一个对象

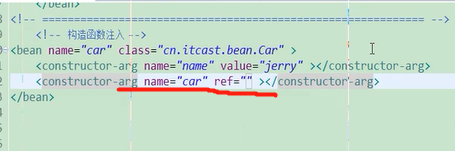
## 构造函数注入

Name：表示构造函数的参数名

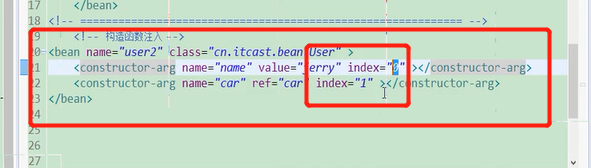
Index表示构造函数的参数索引

Type:构造函数的参数类型

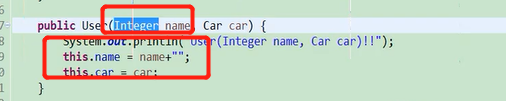
（反正就是最好是一一对应的，这样就不会出错）

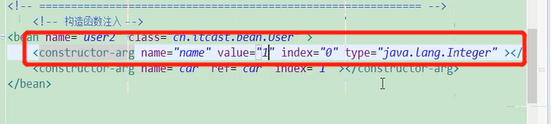


但是要按构造函数形参的顺序（严格）



引入了index，这样在配置中写错顺序也不会有事了





这样后面的那个type指定类型就不会报错了

表示走第一个参数，名为name，类型为Integer类型



## p名称空间注入（了解）

以后自己在看看

## Spel注入（了解）

以后自己在看看

## 复杂类型属性的注入：

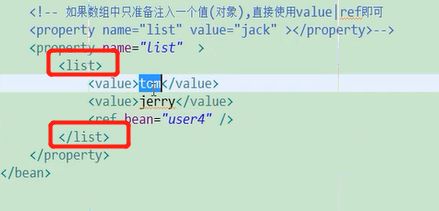
### 数组类型：



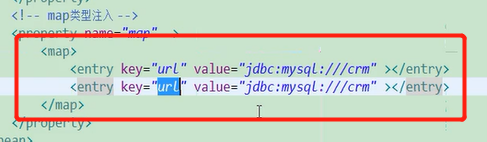
不管你怎么玩，都是在一个bean里面的，直接用name找到相应的属性

### List类型：

在List当中，就用list外括号，其他类型同



### Map类型：



但记住，还是在bean类所指定的类里面括起来的。

Map是键值对，key-value

就像写jdbc驱动一样，但是我们不这样用，以后我们的jdbc直接交给spring进行管理就好了

### property类型：

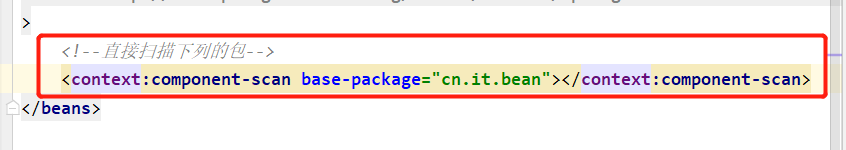
也是键值对，



## 注解注入：（也蛮常用的）

代码一多就很难看，快是快，看看项目催不催上线

### 配置文件：



直接去扫描指定包（该包下的子包也会跟着一起扫面）,并将类自动生成**bean**,其中的name自己在类中自己定义

这样可以省去大量的配置文件，开始默认是Compontent,后面加了另外三个，就容易区分各层，定义的什么层就是什么层的bean。



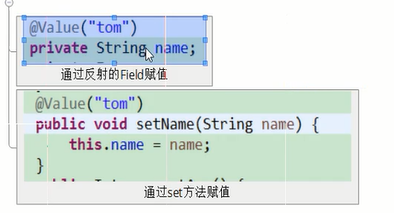
直接@Scope

Scope也在上面直接指定了，默认是singleton单例的。

（singleton单例的）（prototype多例的）

单例与多例的区分见上面的代码

### 单属性注入：



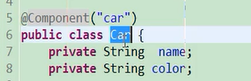
**不好的一点就是破坏了对象的封装性**

第一个通过成员变量赋值（常用）

第二个直接放在setter方法上赋值（了解）

### 给对象中的对象赋值：

要先把对象中的对象注册到component当中去还要给上name

**1：** **2：**

直接autowired自动封装

若要进行值注入的话，直接在对象的对象中用value进行值注入

### 注意：

如果匹配多个类型一致的对象，将无法选择具体注入哪一个对象，

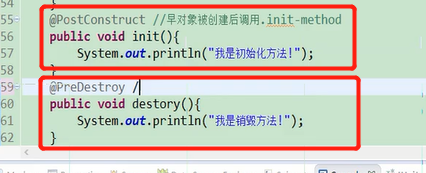
比如你注解中value给了，又去xml文件中给一下，那么他就会迷茫，在随机给一个，这时候我们可以



利用注解@Qualifier来指定中间的是name 属性

自动装配并去找car2的值

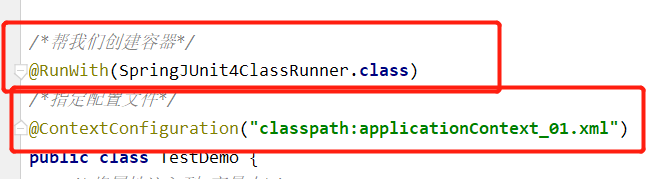
### 注解配置初始化和销毁方法：



这个是在init和destory方法是在pojo类当中定义的。

# Spring整合Junit测试：

## 导入xml配置文件

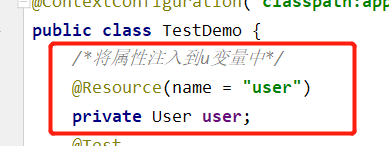


@RunWith就是默认创建容器

然后来指定配置文件@ContextConfiguration

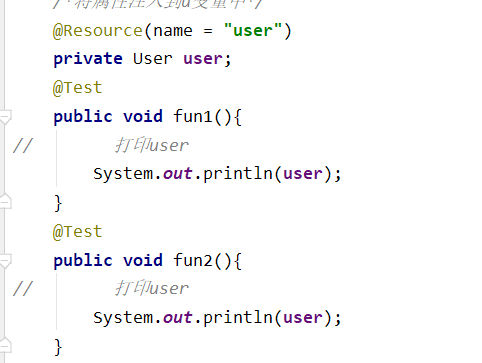
然后是

### 属性注入



表示以bean中的User来注入

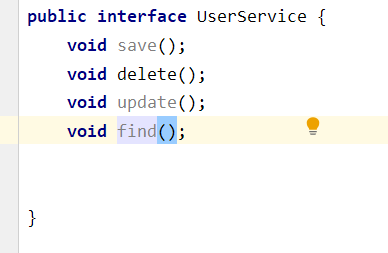
还可以一个多用



# Spring实现动态代理

## 详解：

一个接口



实现服务接口



然后定义一个**UserService**的代理工厂**UserServiceProxyFactory**

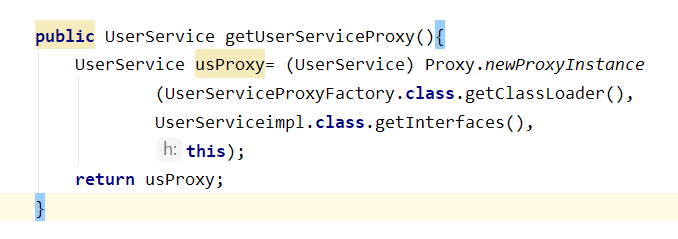
用于生成代理对象

这个工厂必须实现**InvocationHandler**接口

参数1：类加载器（用本类的就行）UserServiceProxyFactory.**class**.getClassLoader()

参数2：要实现代理对象的接口UserServiceimpl.**class**.getInterfaces()，直接反射获取

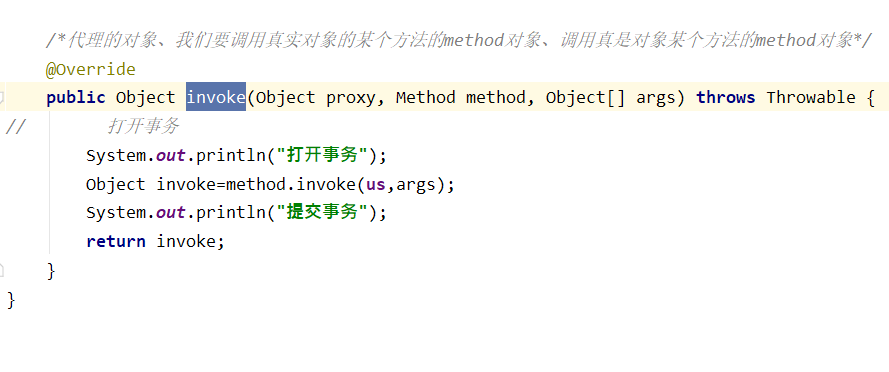
参数3：3.实现增强的代码(对原方法增强的一个方法)，因为是本函数实现了**InvocationHandler接口**，就直接this就行了



让他返回生成的代理对象

由于**UserServiceProxyFactory**实现了**InvocationHandler**接口，就要复写**invoke**方法，

invoke是干嘛用的呢，



上面的invoke方法中的形参代表

***代理的对象****、*

***我们要调用真实对象的某个方法的method对象****、*

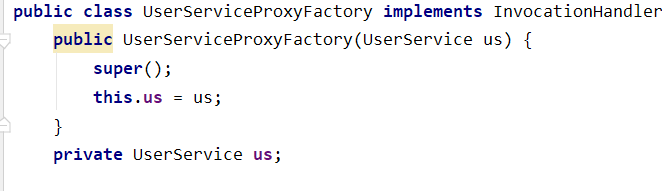
***调用真是对象某个方法的method对象***

Invoke方法就是你在代理对象中要扩充的方法，就在里面写，见上面的例子

然后由于我们是在工厂中定义私有变量，并写了一个构造将其传入，以后也是常用这种方法

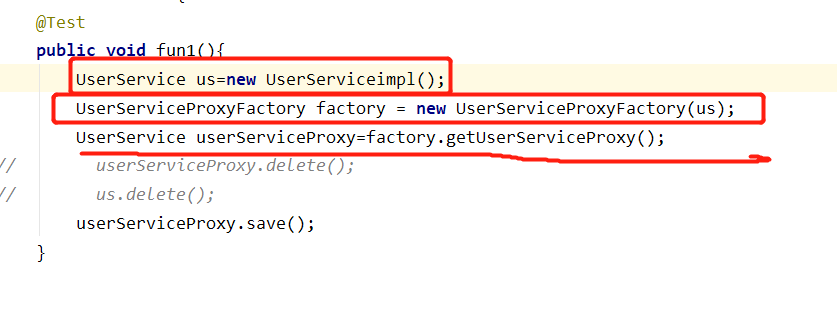
## 注意：

**UserServiceProxyFactory**里面一定要传入被代理对象



直接用构造传入就好

看测试类：



先生成一个工厂，然后在生成工厂类对象**factory**,factory在调用自己定义的getUserServiceProxy方法来获得，代理对象。

# cglib代理：

下次再说

cglib可以代理没有接口的对象，一般多用动态代理

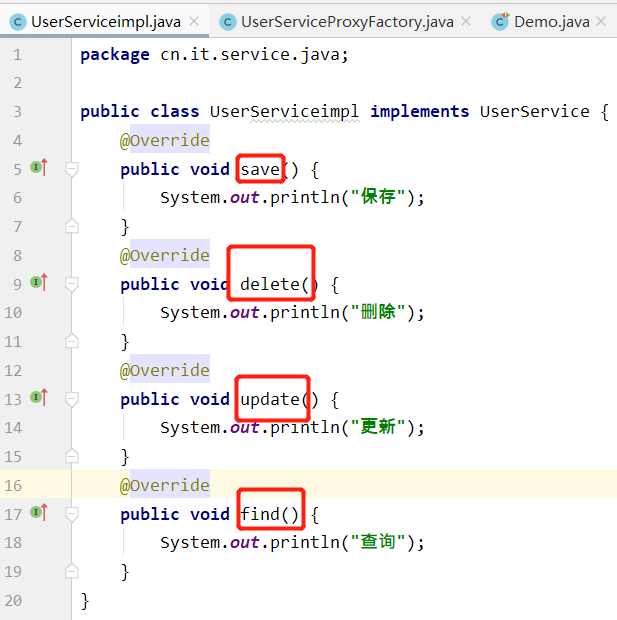
# Spring AOP（面向切面的编程）

横向重复代码，纵向抽取

## Joinpoint连接点：

目标对象（被代理对象）当中，所有可以增强的方法

可被应用通知的方法就是连接点



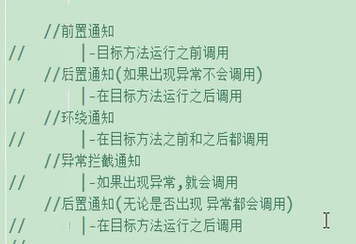
## Pointcut切入点：目标对象已经增强的方法

目标方法真正的被代理了，他就变成了切入点

## Advice(通知、增强)：增强的代码

写在Aspect中



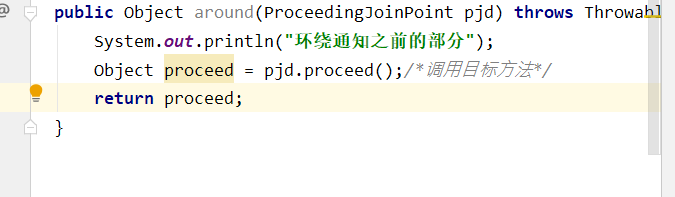


### 前置通知

### 后置通知

如果出现异常就不调用

### 环绕通知

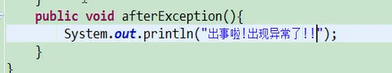


必须要接收ProceedingJoinPoint

#### 包：



### 异常拦截通知



如果出现异常就会调用

## Target(目标对象)：被代理对象

## Weaving(织入):将通知织入切入点，形成代理的过程，就叫织入

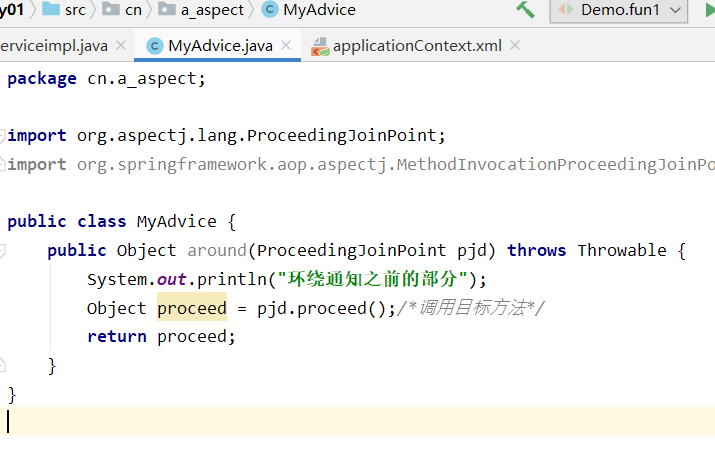
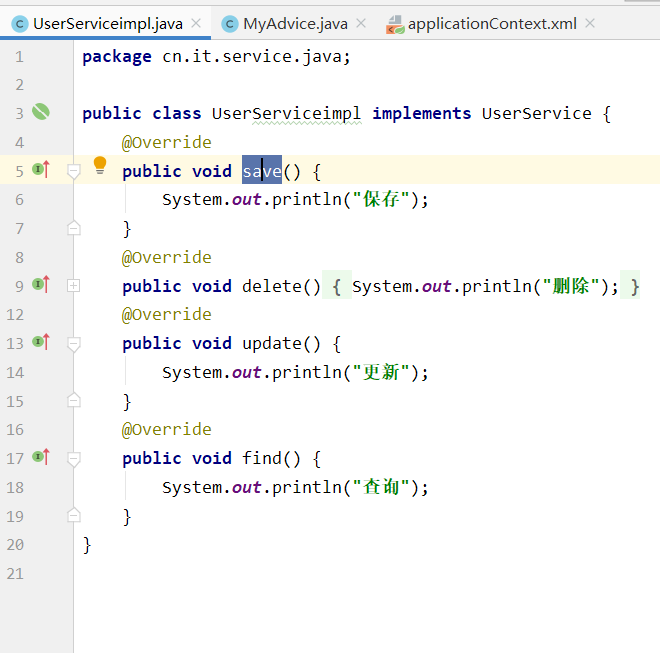
见下方

## Proxy(代理)：将通知织入到目标对象之后，形成代理对象

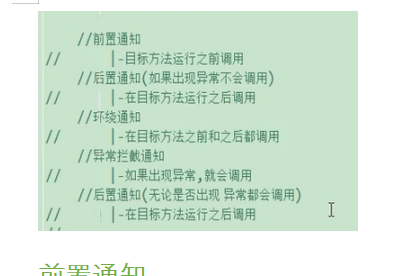
实际配置文件：(applicationContext.xml文件)也是将目标织

入切点

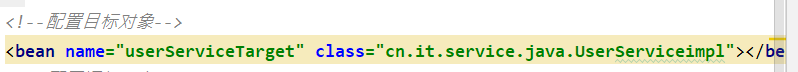
前提：



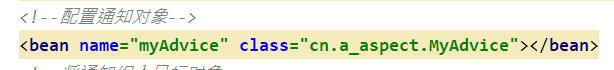
我的Advice里面还有

这个，有必要就配入

第一步



第二步



第三步

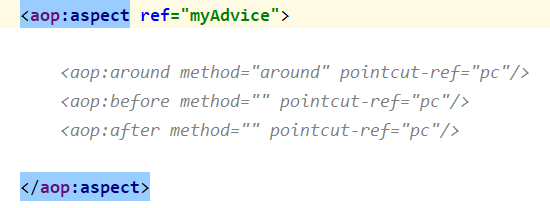
配置切入点



***excution表达式*** *版本1：void cn.it.service.java.UserServiceimpl.save()  
正常版本的  
版本2：\* cn.it.service.java.UserServiceimpl.save()  
表示对该路径下的不管什么返回值方法的，save方法进行增强  
版本3：\* cn.it.service.java.UserServiceimpl.\*()  
表示对该路径下的不管什么返回值方法的，所有方法进行增强  
注意：上面的方法表示空参的方法  
  
4-5最常用  
版本4：\* cn.it.service.java.\*Serviceimpl.\*(..)  
表示对该路径下的不管什么返回值方法的、以Serviceimpl，所有方法进行增强,且不限任何参数  
版本5：\* cn.it.service.java..\*Serviceimpl.\*(..)  
不仅会找该目录下的，还会找子目录的子包中，以Serviceimpl结尾的方法*

第四步

配置通知

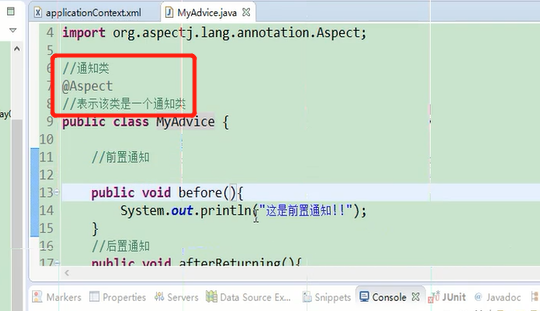


配置之前都是要将他们注册进去的

思想也是代理，动态代理可以体现AOP思想

## 注解配置完成织入

在通知类前面加一个@Aspect,表示他是一个通知类

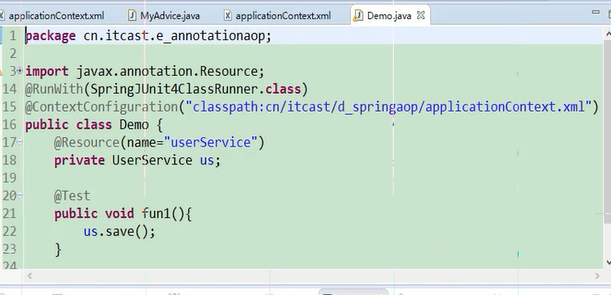


@Before表示前置通知后面加的是execution表达式

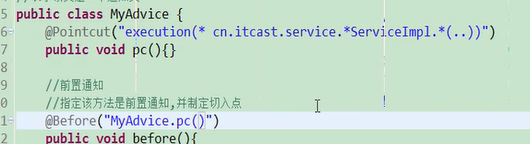


其余也是相同的，这里不做说明。

测试类也是相同的方法



### 把这个切点拿出来



方便管理，就不用在去改代码了

在方法名前面加一个@Pointcut，然后在

通知前面EG：@Before(“MyAdvice.pc()”)

Mybatis

# Mybatis架构



1. mybatis配置

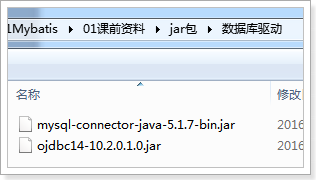
SqlMapConfig.xml，此文件作为mybatis的全局配置文件，配置了mybatis的运行环境等信息。

mapper.xml文件即sql映射文件，文件中配置了操作数据库的sql语句。此文件需要在SqlMapConfig.xml中加载。

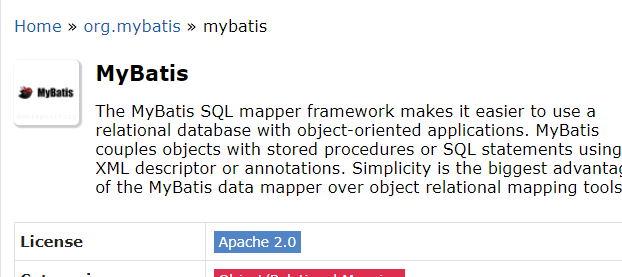
1. 通过mybatis环境等配置信息构造SqlSessionFactory即会话工厂
2. 由会话工厂创建sqlSession即会话，操作数据库需要通过sqlSession进行。
3. mybatis底层自定义了Executor执行器接口操作数据库，Executor接口有两个实现，一个是基本执行器、一个是缓存执行器。
4. Mapped Statement也是mybatis一个底层封装对象，它包装了mybatis配置信息及sql映射信息等。mapper.xml文件中一个sql对应一个Mapped Statement对象，sql的id即是Mapped statement的id。
5. Mapped Statement对sql执行输入参数进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql前将输入的java对象映射至sql中，输入参数映射就是jdbc编程中对preparedStatement设置参数。
6. Mapped Statement对sql执行输出结果进行定义，包括HashMap、基本类型、pojo，Executor通过Mapped Statement在执行sql后将输出结果映射至java对象中，输出结果映射过程相当于jdbc编程中对结果的解析处理过程。

# Mybaitis环境配置

## JAR包

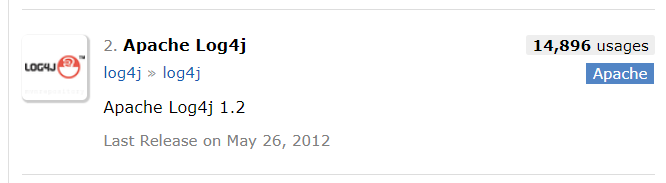


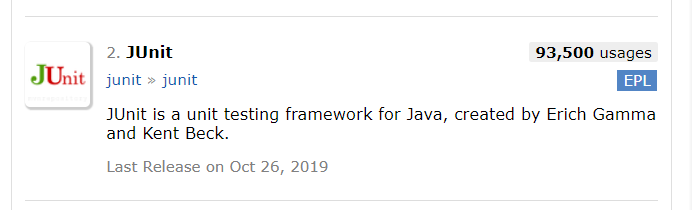
是mysql，或者ojdbc的驱动



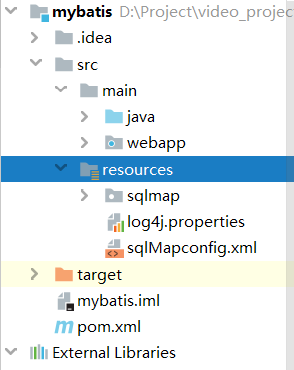
直接maven导入

日志文件来的

  
junit测试



# 搭建例子



## log4j.properties

把日志交给log4j进行管理

*# Global logging configuration***log4j.rootLogger**=**DEBUG, stdout***# Console output...***log4j.appender.stdout**=**org.apache.log4j.ConsoleAppender  
log4j.appender.stdout.layout**=**org.apache.log4j.PatternLayout  
log4j.appender.stdout.layout.ConversionPattern**=**%5p [%t] - %m%n**

属于资源文件，放在resources文件里面就好

## SqlMapConfig.xml配置文件

*<?***xml version="1.0" encoding="UTF-8"** *?>***<!DOCTYPE configuration  
 PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-config.dtd"*>***<**configuration**>  
 *<!-- 和spring整合后 environments配置将废除 -->* <**environments default="development"**>  
 <**environment id="development"**>  
 *<!-- 使用jdbc事务管理 -->* <**transactionManager type="JDBC"** />  
 *<!-- 数据库连接池 -->* <**dataSource type="POOLED"**>  
 <**property name="driver" value="com.mysql.jdbc.Driver"** />  
 <**property name="url"  
 value="jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8"** />  
 <**property name="username" value="root"** />  
 <**property name="password" value="rose"** />  
 </**dataSource**>  
 </**environment**>  
 </**environments**>  
  
<**mappers**>  
<**mapper resource="sqlmap/User.xml"**></**mapper**>  
</**mappers**>  
 </**configuration**>

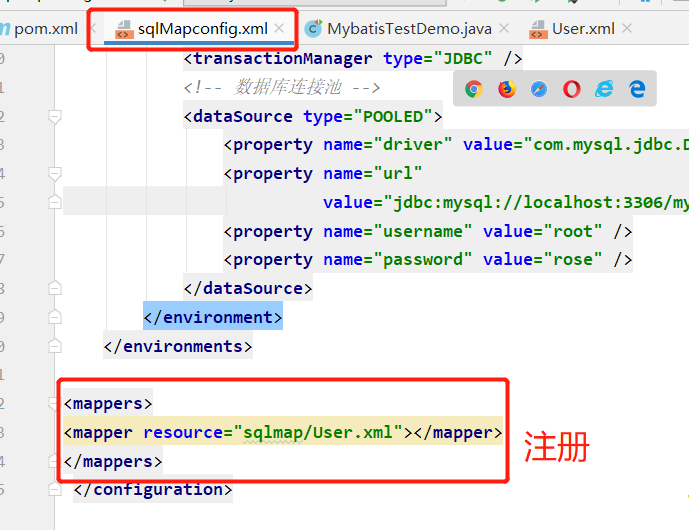
# 各个XX(mapper).xml

必须要的头约束

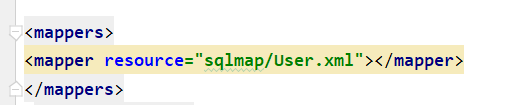
*<?***xml version="1.0" encoding="UTF-8"** *?>***<!DOCTYPE mapper  
 PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  
 "http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>***

都要在**configuration里面进行mapper注册**

**EG:**



**在SqlMapConfig.xml**

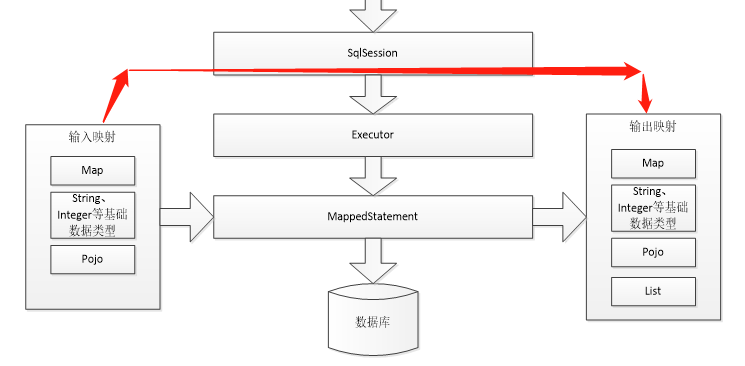


注意：

这里面放的是相对的路径

各个XX(mapper).xml映射的是pojo类中的对象

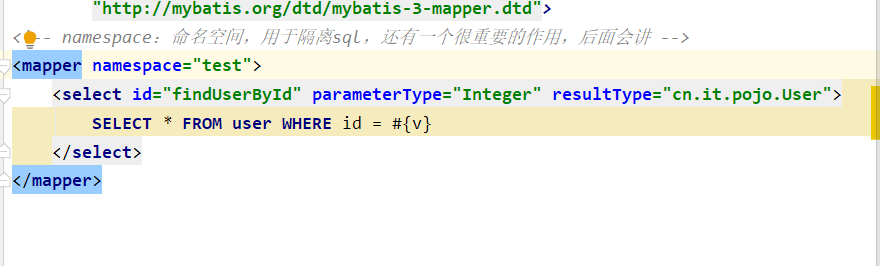
pojo类作为mybatis进行sql映射使用，pojo类通常与数据库表对应



输入的东西，然后输出处理过后的东西。

就是进行数据库操作的东西

### EG:



### parameterType和resultType

parameterType：指定输入参数类型

resultType：指定输出结果类型，mybatis将sql查询结果的一行记录数据映射为resultType指定类型的对象。如果有多条数据，则分别进行映射，并把对象放到容器List中

如果是返回的是list，填的也是List的泛型

### #{}和${}

#{}表示一个占位符号，通过#{}可以实现preparedStatement向占位符中设置值，自动进行java类型和jdbc类型转换。#{}可以有效防止sql注入。 #{}可以接收简单类型值或pojo属性值。 如果parameterType传输单个简单类型值，#{}括号中可以是value或其它名称。

${}表示拼接sql串，通过${}可以将parameterType 传入的内容拼接在sql中且不进行jdbc类型转换， ${}可以接收简单类型值或pojo属性值，如果parameterType传输单个简单类型值，${}括号中只能是value。

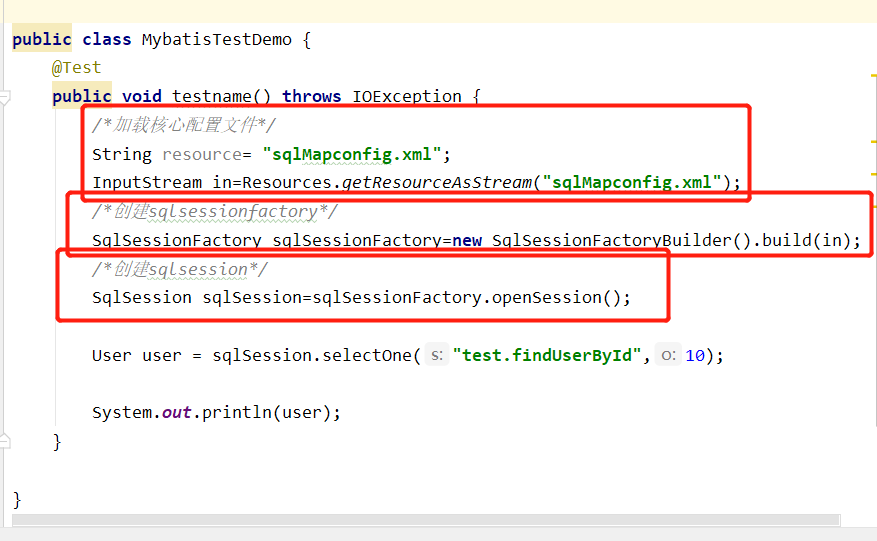
# 数据库操作：

记得要commit才能生效

Rollback回滚，让sqlsession来进行操作就好了

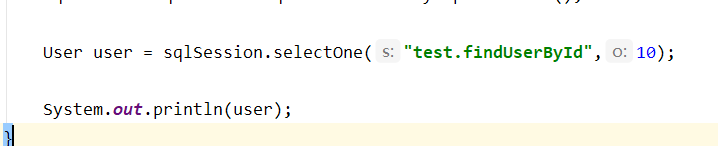
## 简单查询：

junit测试

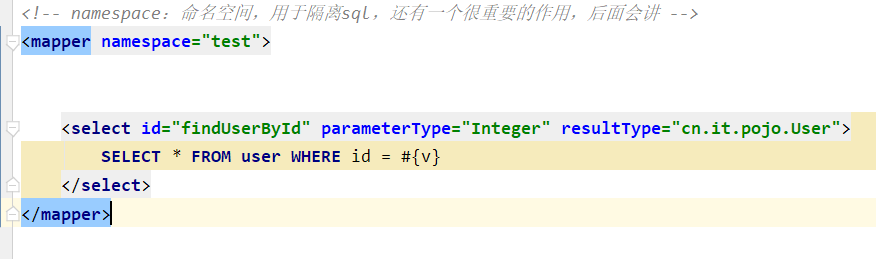


跟架构一样，先有sqlMapconfig.xml文件，然后才有SqlSessionFactory

，后又产生SqlSession，sqlsession就可以进行数据库操作了



 test是命名空间，是对应的mapper.xml文件中配置好，进行数据库操作的id



### 注意：

Integer,本来是java.lang.Integer,但是我们的mybatis帮我们取完别名了。

这里的形参不是jdbc中的？

而是#{v} 里面v随意写的，但也有说法，

只有占位符的时候，才可以乱写，但是也要有内容，

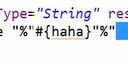
当表示拼接符${xxxxx}$的时候，里面必须要是规定value的字段

如果传入的参数是简单数据类型，${}里面必须写value

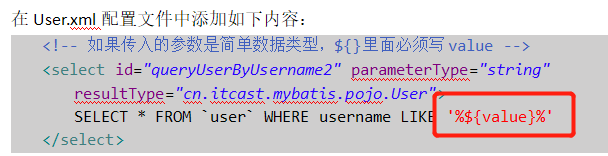
万一有sql注入咋办

最终版本可以写成这样子"%"{hhh}"%"

这样不仅防注入，还可以里面随便填写



#### 例如：



### selectOne和selectList

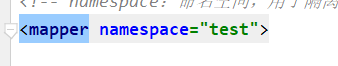
selectOne查询一条记录，如果使用selectOne查询多条记录则抛出异常：

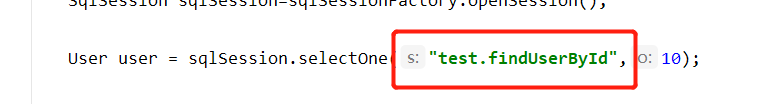
org.apache.ibatis.exceptions.TooManyResultsException: Expected one result (or null) to be returned by selectOne(), but found: 3

at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:70)

selectList可以查询一条或多条记录。

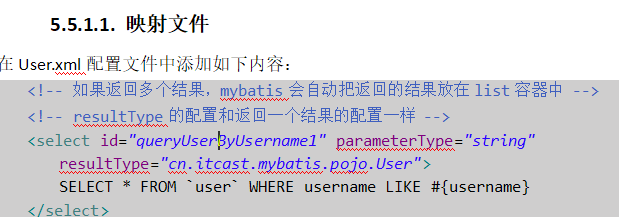
### 命名空间

加入命名空间，是为了以后调用的时候，比如说是pojo中User的mapper，就给个给User,这样就不会搞混了，调用的时候

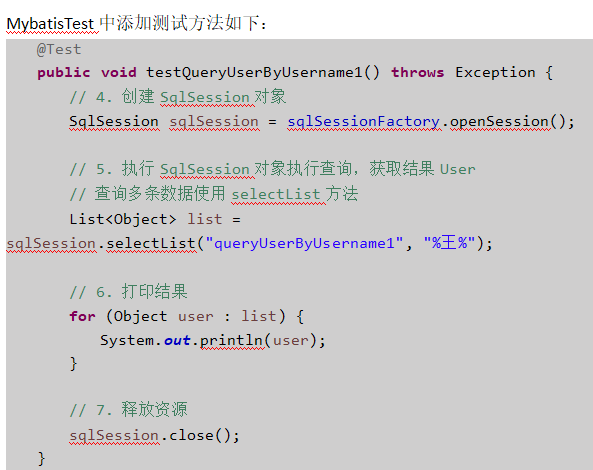


用user接收，以后基本是传入一个user然后在返回一个user回来，mybatis自动帮我们封装好

## 模糊查询



### 测试程序：



#### 注意：

上面占位哪里，是有加双引号的

因为返回的很多，就直接用list集合接收了。

### selectOne和selectList

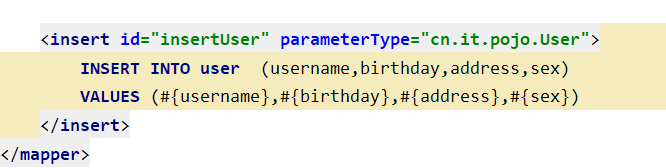
selectOne查询一条记录，如果使用selectOne查询多条记录则抛出异常：

org.apache.ibatis.exceptions.TooManyResultsException: Expected one result (or null) to be returned by selectOne(), but found: 3

at org.apache.ibatis.session.defaults.DefaultSqlSession.selectOne(DefaultSqlSession.java:70)

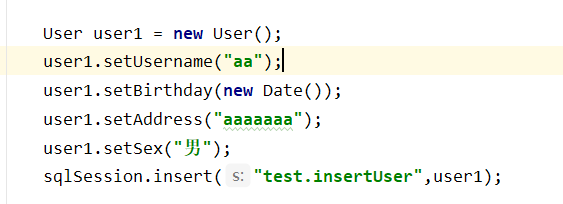
selectList可以查询一条或多条记录。

## 数据插入：



注意要和pojo中的数据一一对应

调用



这样想想交给spring容器更好

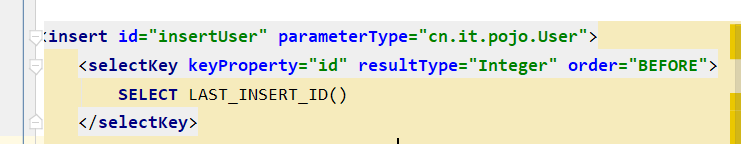
注意上面用的是#{}而不是"%"{hhh}"%"或者${value}

**记得要commit；**

**才能最终提交**

有时候要刚插入新用户，他就要下单了，然后我们就要,当即返回新用户的id，我们就

先insert，然后这样



### Order是

其实就是id在保存数据之前生成还是在保存数据之后生成的。

1. order AFTER 表示 SELECT LAST\_INSERT\_ID() 在insert执行之后执行,多用与自增主键，

自增长类型的就是AFTER，就是先把数据保存下来，然后在生成id，就是after

1. BEFORE 表示 SELECT LAST\_INSERT\_ID() 在insert执行之前执行，这样的话就拿不到主键了，

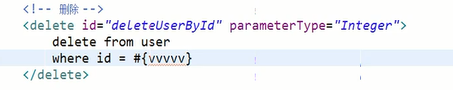
如果是在保存数据之后生成的，生成了id然后一起和数据一起插进去，就是before

用oracle就是之前。

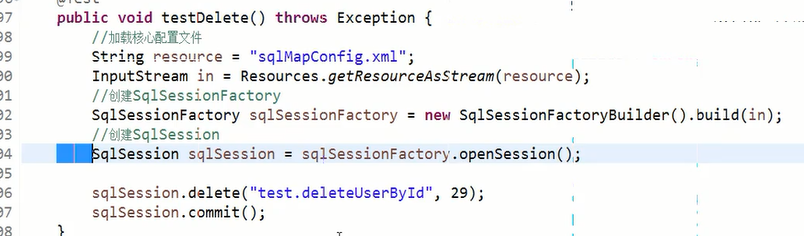
用integer类型的，然后是mysqsl，就是after

## 数据删除

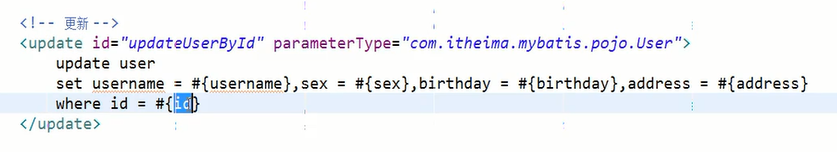
记得要commit才能生效



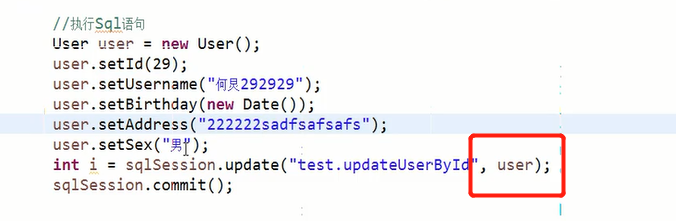
占位符写这个也行



## 数据更新：



记得传过来的是一个user对象



# 面试题

## Mybatis解决jdbc编程的问题

1. 数据库连接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库连接池可解决此问题。

解决：在SqlMapConfig.xml中配置数据连接池，使用连接池管理数据库链接。

1. Sql语句写在代码中造成代码不易维护，实际应用sql变化的可能较大，sql变动需要改变java代码。

解决：将Sql语句配置在XXXXmapper.xml文件中与java代码分离。

1. 向sql语句传参数麻烦，因为sql语句的where条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决：Mybatis自动将java对象映射至sql语句，通过statement中的parameterType定义输入参数的类型。

1. 对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。

解决：Mybatis自动将sql执行结果映射至java对象，通过statement中的resultType定义输出结果的类型。

## mybatis与hibernate不同

## mybatis与hibernate不同

Mybatis和hibernate不同，它不完全是一个ORM框架，因为MyBatis需要程序员自己编写Sql语句。mybatis可以通过XML或注解方式灵活配置要运行的sql语句，并将java对象和sql语句映射生成最终执行的sql，最后将sql执行的结果再映射生成java对象。

Mybatis学习门槛低，简单易学，程序员直接编写原生态sql，可严格控制sql执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，例如互联网软件、企业运营类软件等，因为这类软件需求变化频繁，一但需求变化要求成果输出迅速。但是灵活的前提是mybatis无法做到数据库无关性，如果需要实现支持多种数据库的软件则需要自定义多套sql映射文件，工作量大。

Hibernate对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件（例如需求固定的定制化软件）如果用hibernate开发可以节省很多代码，提高效率。但是Hibernate的学习门槛高，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡，以及怎样用好Hibernate需要具有很强的经验和能力才行。

总之，按照用户的需求在有限的资源环境下只要能做出维护性、扩展性良好的软件架构都是好架构，所以框架只有适合才是最好。

# Mapper动态代理开发

## 四个原则：

//接口 方法名 == User.xml中id名

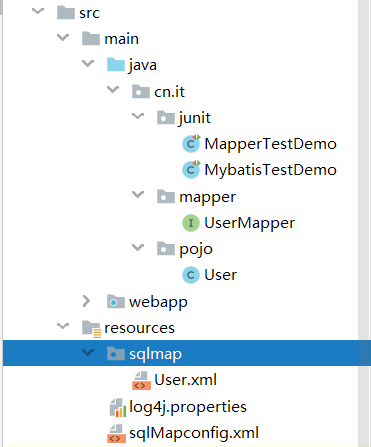
//返回值类型 与mapper.xml文件中返回值类型要一致

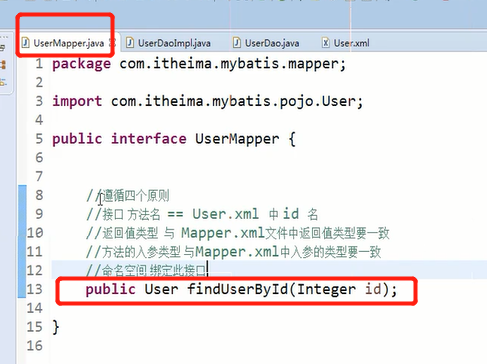
//方法的入参类型与Mapper.xml中入参的类型要一致

//命名空间 绑定此接口

### Eg:

目录结构

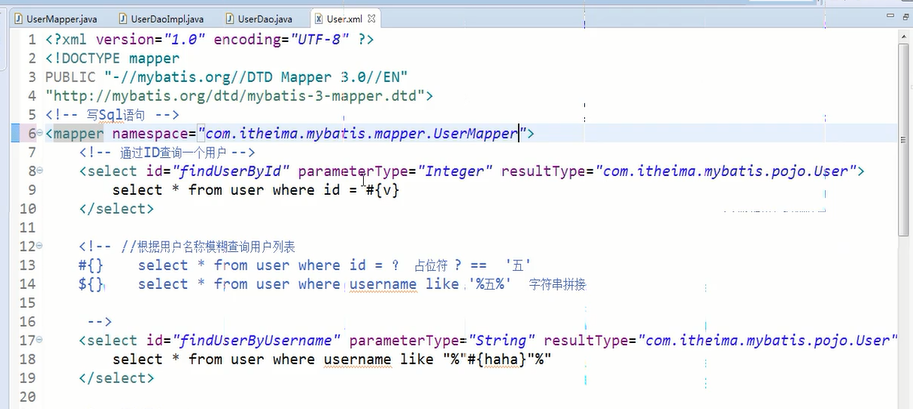




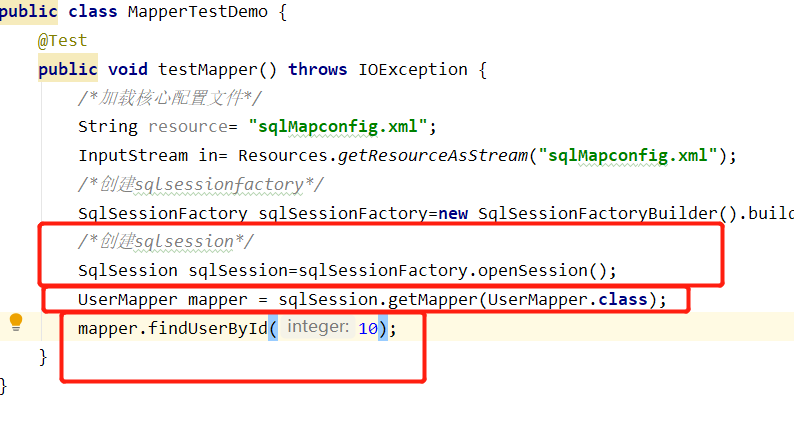
与User.xml文件对应

是能从userMapper中找到对应的接口userMapper

正规的命名空间：



测试：



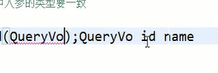
还是要先opensSession()

## selectOne和selectList

动态代理对象调用sqlSession.selectOne()和sqlSession.selectList()是根据mapper接口方法的返回值决定，如果返回list则调用selectList方法，如果返回单个对象则调用selectOne方法。

## namespace

mybatis官方推荐使用mapper代理方法开发mapper接口，程序员不用编写mapper接口实现类，使用mapper代理方法时，输入参数可以使用pojo包装对象或map对象，保证dao的通用性。

往这里一放，就不管你怎么变，都不用在改了

# SqlMapConfig.xml配置文件

## 配置内容

严格按照

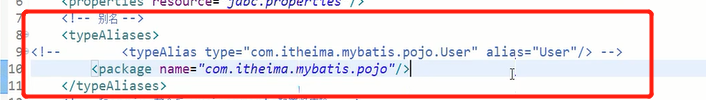
**SqlMapConfig.xml中配置的内容和顺序如下：**

### **properties**（属性）

### settings（全局配置参数）

### **typeAliases**（类型别名）

（这个类别名其实就是给包取个别名）



会自动扫描父包及其子包

### typeHandlers（类型处理器）

### objectFactory（对象工厂）

### plugins（插件）

### environments（环境集合属性对象）

### environment（环境子属性对象）

### transactionManager（事务管理）

### dataSource（数据源）

### **mappers**（映射器）



这三个，resource,class,url智能出现一个

#### <mapper resource=" " />

使用相对于类路径的资源（现在的使用方式）

如：<mapper resource="sqlmap/User.xml" />

#### <mapper class=" " />

使用mapper接口类路径

如：<mapper class="cn.itcast.mybatis.mapper.UserMapper"/>

注意：此种方法要求mapper接口名称和mapper映射文件名称相同，且放在同一个目录中。

**最长用的就是package**

#### 指定包：<package name=""/>

注册指定包下的所有mapper接口

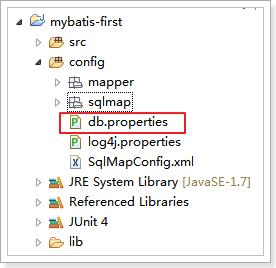
如：<package name="c n.itcast.mybatis.mapper"/>

注意：此种方法要求mapper接口名称和mapper映射文件名称相同，且放在同一个目录中。

## properties（属性）

SqlMapConfig.xml可以引用java属性文件中的配置信息如下：

在config下定义db.properties文件，如下所示：



**db.properties配置文件**内容如下：

加个jdbc就是不要搞混了，跟命名空间的意思一样

jdbc.driver=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mybatis?characterEncoding=utf-8

jdbc.username=root

jdbc.password=root

SqlMapConfig.xml引用如下：直接在xml文件下配置了

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE configuration

PUBLIC "-//mybatis.org//DTD Config 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>

<!-- 是用resource属性加载外部配置文件 -->

<properties resource=*"db.properties"*>

<!-- 在properties内部用property定义属性 -->

<!-- 如果外部配置文件有该属性，则内部定义属性被外部属性覆盖 -->

<property name=*"jdbc.username"* value=*"root123"* />

<property name=*"jdbc.password"* value=*"root123"* />

</properties>

<!-- 和spring整合后 environments配置将废除 -->

<environments default=*"development"*>

<environment id=*"development"*>

<!-- 使用jdbc事务管理 -->

<transactionManager type=*"JDBC"* />

<!-- 数据库连接池 -->

<dataSource type=*"POOLED"*>

<property name=*"driver"* value=*"${jdbc.driver}"* />

<property name=*"url"* value=*"${jdbc.url}"* />

<property name=*"username"* value=*"${jdbc.username}"* />

<property name=*"password"* value=*"${jdbc.password}"* />

</dataSource>

</environment>

</environments>

<!-- 加载映射文件 -->

<mappers>

<mapper resource=*"sqlmap/User.xml"* />

<mapper resource=*"mapper/UserMapper.xml"* />

</mappers>

</configuration>

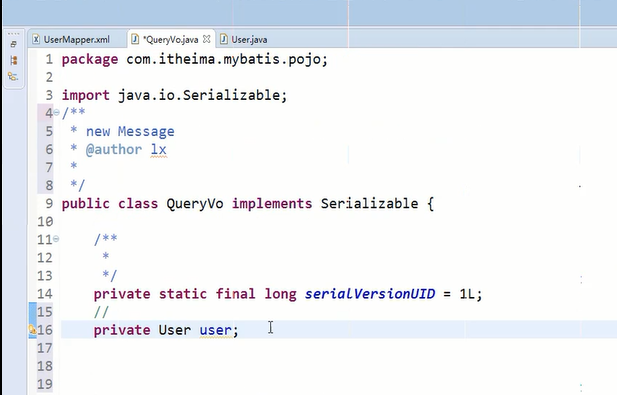
注意： MyBatis 将按照下面的顺序来加载属性：

* 在 properties 元素体内定义的属性首先被读取。
* 然后会读取properties 元素中resource或 url 加载的属性，它会覆盖已读取的同名属性。

# Vo类：

VO，这是用来转换从entity到dto，或者从dto到entity的中间的东西。

要实现



包装一个对象

## resultMap

resultType可以指定将查询结果映射为pojo，但需要pojo的属性名和sql查询的列名一致方可映射成功。

如果sql查询字段名和pojo的属性名不一致，可以通过resultMap将字段名和属性名作一个对应关系 ，resultMap实质上还需要将查询结果映射到pojo对象中。

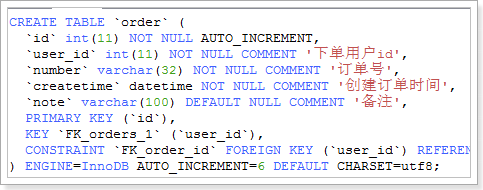
resultMap可以实现将查询结果映射为复杂类型的pojo，比如在查询结果映射对象中包括pojo和list实现一对一查询和一对多查询。

需求：查询订单表order的所有数据

sql：SELECT id, user\_id, number, createtime, note FROM `order`

### 声明pojo对象

数据库表如下图：



Order对象：

**public** **class** Order {

// 订单id

**private** **int** id;

// 用户id

**private** Integer userId;

// 订单号

**private** String number;

// 订单创建时间

**private** Date createtime;

// 备注

**private** String note;

get/set。。。

}

### Mapper.xml文件

创建OrderMapper.xml配置文件，如下：

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- namespace：命名空间，用于隔离sql，还有一个很重要的作用，Mapper动态代理开发的时候使用，需要指定Mapper的类路径 -->

<mapper namespace=*"cn.itcast.mybatis.mapper.OrderMapper"*>

<!-- 查询所有的订单数据 -->

<select id=*"queryOrderAll"* resultType=*"order"*>

SELECT id, user\_id,

number,

createtime, note FROM `order`

</select>

</mapper>

### Mapper接口

编写接口如下：

**public** **interface** OrderMapper {

/\*\*

\* 查询所有订单

\*

\* **@return**

\*/

List<Order> queryOrderAll();

}

### 测试方法

编写测试方法OrderMapperTest如下：

**public** **class** OrderMapperTest {

**private** SqlSessionFactory sqlSessionFactory;

@Before

**public** **void** init() **throws** Exception {

InputStream inputStream = Resources.*getResourceAsStream*("SqlMapConfig.xml");

**this**.sqlSessionFactory = **new** SqlSessionFactoryBuilder().build(inputStream);

}

@Test

**public** **void** testQueryAll() {

// 获取sqlSession

SqlSession sqlSession = **this**.sqlSessionFactory.openSession();

// 获取OrderMapper

OrderMapper orderMapper = sqlSession.getMapper(OrderMapper.**class**);

// 执行查询

List<Order> list = orderMapper.queryOrderAll();

**for** (Order order : list) {

System.***out***.println(order);

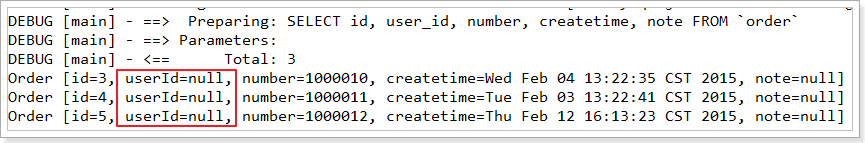
}

}

}

### 效果

测试效果如下图：



发现userId为null

解决方案：使用resultMap

### 使用resultMap

由于上边的mapper.xml中sql查询列(user\_id)和Order类属性(userId)不一致，所以查询结果不能映射到pojo中。

需要定义resultMap，把orderResultMap将sql查询列(user\_id)和Order类属性(userId)对应起来

改造OrderMapper.xml，如下：

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!-- namespace：命名空间，用于隔离sql，还有一个很重要的作用，Mapper动态代理开发的时候使用，需要指定Mapper的类路径 -->

<mapper namespace=*"cn.itcast.mybatis.mapper.OrderMapper"*>

<!-- resultMap最终还是要将结果映射到pojo上，type就是指定映射到哪一个pojo -->

<!-- id：设置ResultMap的id -->

<resultMap type=*"order"* id=*"orderResultMap"*>

<!-- 定义主键 ,非常重要。如果是多个字段,则定义多个id -->

<!-- property：主键在pojo中的属性名 -->

<!-- column：主键在数据库中的列名 -->

<id property=*"id"* column=*"id"* />

<!-- 定义普通属性 -->

<result property=*"userId"* column=*"user\_id"* />

<result property=*"number"* column=*"number"* />

<result property=*"createtime"* column=*"createtime"* />

<result property=*"note"* column=*"note"* />

</resultMap>

<!-- 查询所有的订单数据 -->

<select id=*"queryOrderAll"* resultMap=*"orderResultMap"*>

SELECT id, user\_id,

number,

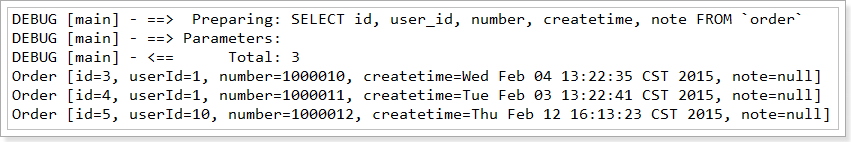
createtime, note FROM `order`

</select>

</mapper>

### 效果

只需要修改Mapper.xml就可以了，再次测试结果如下：



### 使用if标签

改造UserMapper.xml，如下：

<!-- 根据条件查询用户 -->

<select id=*"queryUserByWhere"* parameterType=*"user"* resultType=*"user"*>

SELECT id, username, birthday, sex, address FROM `user`

WHERE 1=1

<if test=*"sex != null and sex != ''"*>

AND sex = #{sex}

</if>

<if test=*"username != null and username != ''"*>

AND username LIKE

'%${username}%'

</if>

</select>

注意字符串类型的数据需要要做不等于空字符串校验。

上面的sql还有where 1=1 这样的语句，很麻烦

可以使用where标签进行改造

改造UserMapper.xml，如下

<!-- 根据条件查询用户 -->

<select id=*"queryUserByWhere"* parameterType=*"user"* resultType=*"user"*>

SELECT id, username, birthday, sex, address FROM `user`

<!-- where标签可以自动添加where，同时处理sql语句中第一个and关键字 -->

<where>

<if test=*"sex != null"*>

AND sex = #{sex}

</if>

<if test=*"username != null and username != ''"*>

AND username LIKE

'%${username}%'

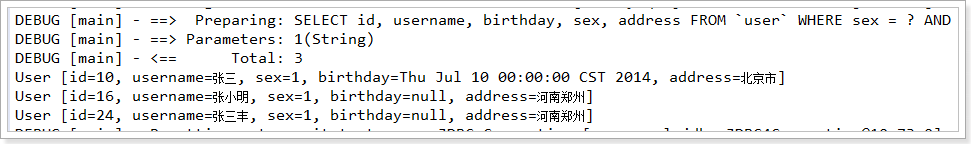
</if>

</where>

</select>

### 效果

测试效果如下图：



## Where标签

上面的sql还有where 1=1 这样的语句，很麻烦

可以使用where标签进行改造

改造UserMapper.xml，如下

<!-- 根据条件查询用户 -->

<select id=*"queryUserByWhere"* parameterType=*"user"* resultType=*"user"*>

SELECT id, username, birthday, sex, address FROM `user`

<!-- where标签可以自动添加where，同时处理sql语句中第一个and关键字 -->

<where>

<if test=*"sex != null"*>

AND sex = #{sex}

</if>

<if test=*"username != null and username != ''"*>

AND username LIKE

'%${username}%'

</if>

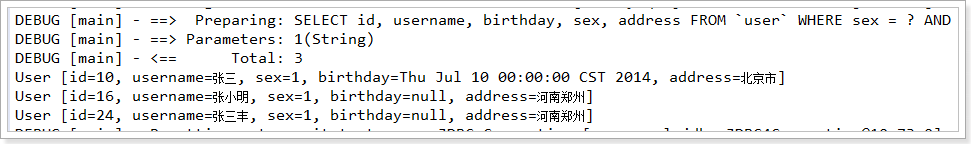
</where>

</select>

Where可以去掉第一个前AND

### 效果

测试效果如下图：



## Sql片段

Sql中可将重复的sql提取出来，使用时用include引用即可，最终达到sql重用的目的。

把上面例子中的id, username, birthday, sex, address提取出来，作为sql片段，如下：

<!-- 根据条件查询用户 -->

<select id=*"queryUserByWhere"* parameterType=*"user"* resultType=*"user"*>

<!-- SELECT id, username, birthday, sex, address FROM `user` -->

<!-- 使用include标签加载sql片段；refid是sql片段id -->

SELECT <include refid=*"userFields"* /> FROM `user`

<!-- where标签可以自动添加where关键字，同时处理sql语句中第一个and关键字 -->

<where>

<if test=*"sex != null"*>

AND sex = #{sex}

</if>

<if test=*"username != null and username != ''"*>

AND username LIKE

'%${username}%'

</if>

</where>

</select>

<!-- 声明sql片段 -->

<sql id=*"userFields"*>

id, username, birthday, sex, address

</sql>

如果要使用别的Mapper.xml配置的sql片段，可以在refid前面加上对应的Mapper.xml的namespace

例如下图



## foreach标签

向sql传递数组或List，mybatis使用foreach解析，如下：

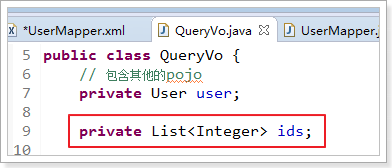
根据多个id查询用户信息

查询sql：

SELECT \* FROM user WHERE id IN (1,10,24)

### 改造QueryVo

如下图在pojo中定义list属性ids存储多个用户id，并添加getter/setter方法



### Mapper.xml文件

UserMapper.xml添加sql，如下：

<!-- 根据ids查询用户 -->

<select id=*"queryUserByIds"* parameterType=*"queryVo"* resultType=*"user"*>

SELECT \* FROM `user`

<where>

<!-- foreach标签，进行遍历 -->

<!-- collection：遍历的集合，这里是QueryVo的ids属性 -->

<!-- item：遍历的项目，可以随便写，，但是和后面的#{}里面要一致 -->

<!-- open：在前面添加的sql片段 -->

<!-- close：在结尾处添加的sql片段 -->

<!-- separator：指定遍历的元素之间使用的分隔符 -->

<foreach collection=*"ids"* item=*"item"* open=*"id IN ("* close=*")"*

separator=*","*>

#{item}

</foreach>

</where>

</select>

测试方法如下图：

@Test

**public** **void** testQueryUserByIds() {

// mybatis和spring整合，整合之后，交给spring管理

SqlSession sqlSession = **this**.sqlSessionFactory.openSession();

// 创建Mapper接口的动态代理对象，整合之后，交给spring管理

UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);

// 使用userMapper执行根据条件查询用户

QueryVo queryVo = **new** QueryVo();

List<Integer> ids = **new** ArrayList<>();

ids.add(1);

ids.add(10);

ids.add(24);

queryVo.setIds(ids);

List<User> list = userMapper.queryUserByIds(queryVo);

**for** (User u : list) {

System.***out***.println(u);

}

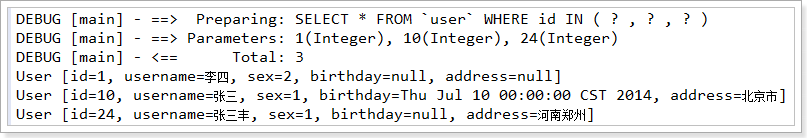
// mybatis和spring整合，整合之后，交给spring管理

sqlSession.close();

}

### 效果

测试效果如下图：



### Mapper代理形式开发dao

#### 实现Mapper.xml

编写UserMapper.xml配置文件，如下：

<?xml version=*"1.0"* encoding=*"UTF-8"* ?>

<!DOCTYPE mapper

PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"

"http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace=*"cn.itcast.mybatis.mapper.UserMapper"*>

<!-- 根据用户id查询 -->

<select id=*"queryUserById"* parameterType=*"int"* resultType=*"user"*>

select \* from user where id = #{id}

</select>

<!-- 根据用户名模糊查询用户 -->

<select id=*"queryUserByUsername"* parameterType=*"string"*

resultType=*"user"*>

select \* from user where username like '%${value}%'

</select>

<!-- 添加用户 -->

<insert id=*"saveUser"* parameterType=*"user"*>

<selectKey keyProperty=*"id"* keyColumn=*"id"* order=*"AFTER"*

resultType=*"int"*>

select last\_insert\_id()

</selectKey>

insert into user

(username,birthday,sex,address) values

(#{username},#{birthday},#{sex},#{address})

</insert>

</mapper>

#### 实现UserMapper接口

**public** **interface** UserMapper {

/\*\*

\* 根据用户id查询

\*

\* **@param** id

\* **@return**

\*/

User queryUserById(**int** id);

/\*\*

\* 根据用户名模糊查询用户

\*

\* **@param** username

\* **@return**

\*/

List<User> queryUserByUsername(String username);

/\*\*

\* 添加用户

\*

\* **@param** user

\*/

**void** saveUser(User user);

}

#### 方式一：配置mapper代理

在applicationContext.xml添加配置

MapperFactoryBean也是属于mybatis-spring整合包

<!-- Mapper代理的方式开发方式一，配置Mapper代理对象 -->

<bean id=*"userMapper"* class=*"org.mybatis.spring.mapper.MapperFactoryBean"*>

<!-- 配置Mapper接口 -->

<property name=*"mapperInterface"* value=*"cn.itcast.mybatis.mapper.UserMapper"* />

<!-- 配置sqlSessionFactory -->

<property name=*"sqlSessionFactory"* ref=*"sqlSessionFactory"* />

</bean>

#### 测试方法

**public** **class** UserMapperTest {

**private** ApplicationContext context;

@Before

**public** **void** setUp() **throws** Exception {

**this**.context = **new** ClassPathXmlApplicationContext("classpath:applicationContext.xml");

}

@Test

**public** **void** testQueryUserById() {

// 获取Mapper

UserMapper userMapper = **this**.context.getBean(UserMapper.**class**);

User user = userMapper.queryUserById(1);

System.***out***.println(user);

}

@Test

**public** **void** testQueryUserByUsername() {

// 获取Mapper

UserMapper userMapper = **this**.context.getBean(UserMapper.**class**);

List<User> list = userMapper.queryUserByUsername("张");

**for** (User user : list) {

System.***out***.println(user);

}

}

@Test

**public** **void** testSaveUser() {

// 获取Mapper

UserMapper userMapper = **this**.context.getBean(UserMapper.**class**);

User user = **new** User();

user.setUsername("曹操");

user.setSex("1");

user.setBirthday(**new** Date());

user.setAddress("三国");

userMapper.saveUser(user);

System.***out***.println(user);

}

}

#### 方式二：扫描包形式配置mapper

<!-- Mapper代理的方式开发方式二，扫描包方式配置代理 -->

<bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>

<!-- 配置Mapper接口 -->

<property name=*"basePackage"* value=*"cn.itcast.mybatis.mapper"* />

</bean>

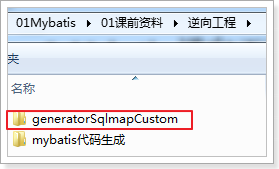
每个mapper代理对象的id就是类名，首字母小写

# Mybatis逆向工程

使用官方网站的Mapper自动生成工具mybatis-generator-core-1.3.2来生成po类和Mapper映射文件

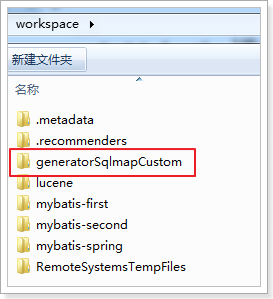
## 导入逆向工程

使用课前资料已有逆向工程，如下图：



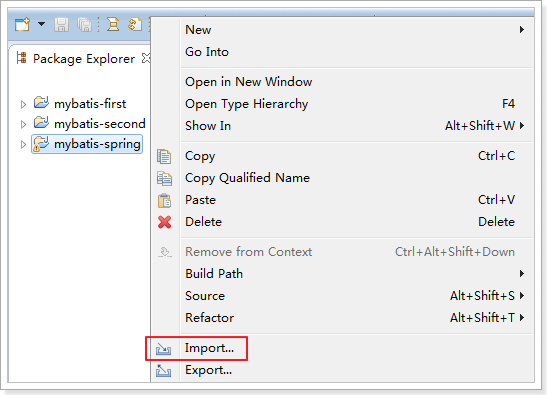
### 复制逆向工程到工作空间中

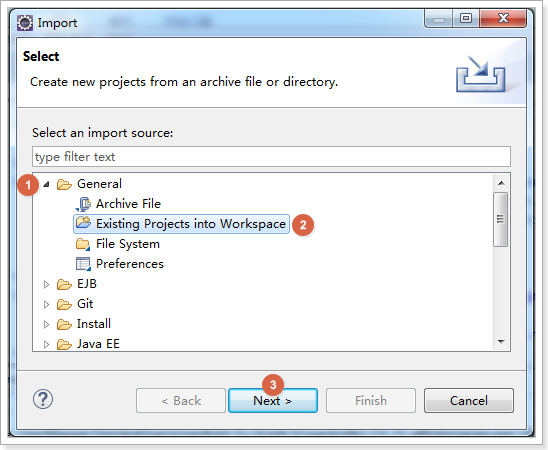
复制的效果如下图：



### 导入逆向工程到eclipse中

如下图方式进行导入：

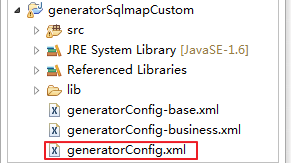






## 修改配置文件

在generatorConfig.xml中配置Mapper生成的详细信息，如下图：



注意修改以下几点:

1. 修改要生成的数据库表
2. pojo文件所在包路径
3. Mapper所在的包路径

配置文件如下:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<!DOCTYPE generatorConfiguration

PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"

"http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">

<generatorConfiguration>

<context id=*"testTables"* targetRuntime=*"MyBatis3"*>

<commentGenerator>

<!-- 是否去除自动生成的注释 true：是 ： false:否 -->

<property name=*"suppressAllComments"* value=*"true"* />

</commentGenerator>

<!--数据库连接的信息：驱动类、连接地址、用户名、密码 -->

<jdbcConnection driverClass=*"com.mysql.jdbc.Driver"*

connectionURL=*"jdbc:mysql://localhost:3306/mybatis" userId="root" password="root"*>

</jdbcConnection>

<!-- <jdbcConnection driverClass="oracle.jdbc.OracleDriver" connectionURL="jdbc:oracle:thin:@127.0.0.1:1521:yycg"

userId="yycg" password="yycg"> </jdbcConnection> -->

<!-- 默认false，把JDBC DECIMAL 和 NUMERIC 类型解析为 Integer，为 true时把JDBC DECIMAL

和 NUMERIC 类型解析为java.math.BigDecimal -->

<javaTypeResolver>

<property name=*"forceBigDecimals"* value=*"false"* />

</javaTypeResolver>

<!-- targetProject:生成PO类的位置 -->

<javaModelGenerator targetPackage=*"cn.itcast.ssm.po"*

targetProject=*".\src"*>

<!-- enableSubPackages:是否让schema作为包的后缀 -->

<property name=*"enableSubPackages"* value=*"false"* />

<!-- 从数据库返回的值被清理前后的空格 -->

<property name=*"trimStrings"* value=*"true"* />

</javaModelGenerator>

<!-- targetProject:mapper映射文件生成的位置 -->

<sqlMapGenerator targetPackage=*"cn.itcast.ssm.mapper"*

targetProject=*".\src"*>

<!-- enableSubPackages:是否让schema作为包的后缀 -->

<property name=*"enableSubPackages"* value=*"false"* />

</sqlMapGenerator>

<!-- targetPackage：mapper接口生成的位置 -->

<javaClientGenerator type=*"XMLMAPPER"*

targetPackage=*"cn.itcast.ssm.mapper"* targetProject=*".\src"*>

<!-- enableSubPackages:是否让schema作为包的后缀 -->

<property name=*"enableSubPackages"* value=*"false"* />

</javaClientGenerator>

<!-- 指定数据库表 -->

<table schema=*""* tableName=*"user"*></table>

<table schema=*""* tableName=*"order"*></table>

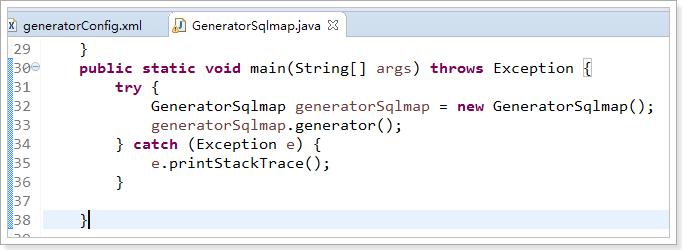
</context>

</generatorConfiguration>

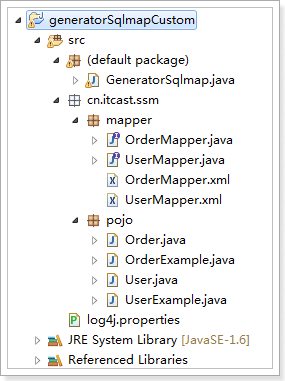
## 生成逆向工程代码

找到下图所示的java文件，执行工程main主函数,



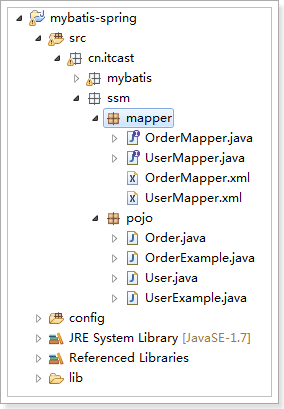


刷新工程，发现代码生成，如下图：



## 测试逆向工程代码

1. 复制生成的代码到mybatis-spring工程，如下图



2. 修改spring配置文件

在applicationContext.xml修改

<!-- Mapper代理的方式开发，扫描包方式配置代理 -->

<bean class=*"org.mybatis.spring.mapper.MapperScannerConfigurer"*>

<!-- 配置Mapper接口，如果需要加载多个包，直接写进来，中间用，分隔 -->

<!-- <property name="basePackage" value="cn.itcast.mybatis.mapper" /> -->

<property name=*"basePackage"* value=*"cn.itcast.ssm.mapper"* />

</bean>

3. 编写测试方法：

**public** **class** UserMapperTest {

**private** ApplicationContext context;

@Before

**public** **void** setUp() **throws** Exception {

**this**.context = **new** ClassPathXmlApplicationContext("classpath:applicationContext.xml");

}

@Test

**public** **void** testInsert() {

// 获取Mapper

UserMapper userMapper = **this**.context.getBean(UserMapper.**class**);

User user = **new** User();

user.setUsername("曹操");

user.setSex("1");

user.setBirthday(**new** Date());

user.setAddress("三国");

userMapper.insert(user);

}

@Test

**public** **void** testSelectByExample() {

// 获取Mapper

UserMapper userMapper = **this**.context.getBean(UserMapper.**class**);

// 创建User对象扩展类，用户设置查询条件

UserExample example = **new** UserExample();

example.createCriteria().andUsernameLike("%张%");

// 查询数据

List<User> list = userMapper.selectByExample(example);

System.***out***.println(list.size());

}

@Test

**public** **void** testSelectByPrimaryKey() {

// 获取Mapper

UserMapper userMapper = **this**.context.getBean(UserMapper.**class**);

User user = userMapper.selectByPrimaryKey(1);

System.***out***.println(user);

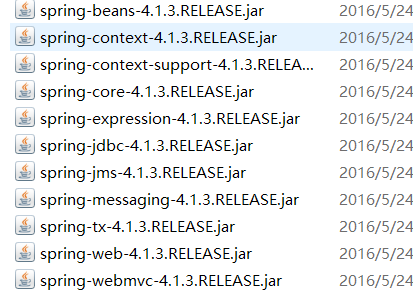
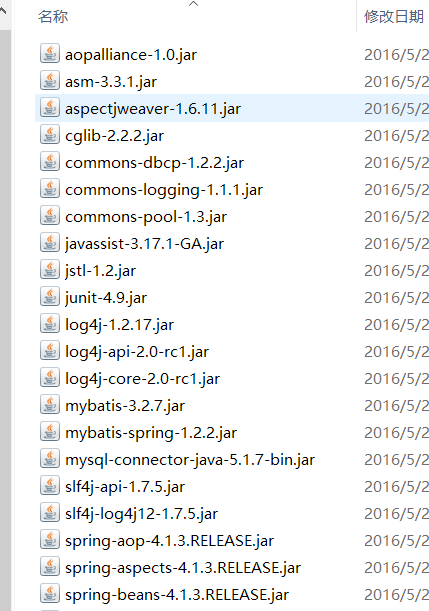
}

}

注意：

1. 逆向工程生成的代码只能做单表查询
2. 不能在生成的代码上进行扩展，因为如果数据库变更，需要重新使用逆向工程生成代码，原来编写的代码就被覆盖了。
3. 一张表会生成4个文件

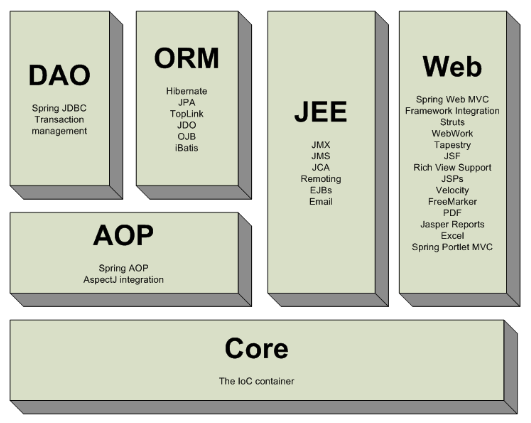
# Spring-mybatis所有包：



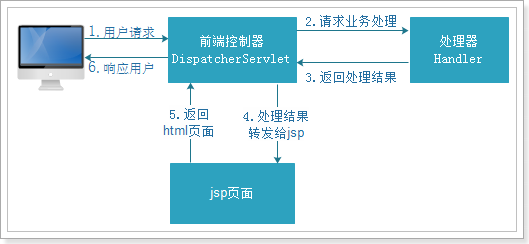
SpringMVC

# Springmvc架构

Spring web mvc和Struts2都属于表现层的框架,它是Spring框架的一部分,我们可以从Spring的整体结构中看得出来,如下图：



Springmvc处理流程



**中心就是前端控制器DispatcherServle**t

**Servlet的话就是放在controller层的**

**并且使用前要在web.xml文件下注册**

<servlet>

<servlet-name>Hello</servlet-name>这里是servlet的名字

<servlet-class>com.nwu.lch.Hello</servlet-class>这里写servlet类在的包路径

</servlet>

<servlet-mapping>这里是地址映射

<servlet-name>Hello</servlet-name>这个和上面的同名

<url-pattern>/Hello</url-pattern>这里写servlet映射地址

</servlet-mapping>

处理器就是controller层



第三个对静态资源放行

拦截器。

怎么使用拦截器

Springmvc的头和spring的头是一摸一样的

# Springmvc尝试：

## 配置前端控制器：

在web.xml文件中

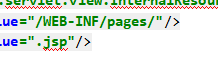
**<!DOCTYPE web-app PUBLIC  
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"  
 "http://java.sun.com/dtd/web-app\_2\_3.dtd" *>***<**web-app**>  
 <**display-name**>Archetype Created Web Application</**display-name**>  
 <**servlet**>  
 <**servlet-name**>springmvc</**servlet-name**>  
 <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  
//这里的时候记得去看看以前配置servlet的区别，通常都这样子配置  
 <**init-param**>  
 <**param-name**>contextConfigLocation</**param-name**>  
 <**param-value**>classpath:springmvc.xml</**param-value**>  
 </**init-param**>  
 </**servlet**>  
 <**servlet-mapping**>  
 <**servlet-name**>springmvc</**servlet-name**>  
 <**url-pattern**>\*.action</**url-pattern**>  
 </**servlet-mapping**>  
   
</**web-app**>

### Springmvc.xml文件

头

*<?***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:p="http://www.springframework.org/schema/p"  
 xmlns:context="http://www.springframework.org/schema/context"  
 xmlns:mvc="http://www.springframework.org/schema/mvc"  
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-4.0.xsd  
 http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd  
 http://www.springframework.org/schema/context [http://www.springframework.org/schema/context/spring-context-4.0.xsd"](http://www.springframework.org/schema/context/spring-context-4.0.xsd\">)**[>](http://www.springframework.org/schema/context/spring-context-4.0.xsd\">)

//以下的三个东西是要我们自己配置的  
  
 *<* *<!--开启注解扫描-->* <**context:component-scan base-package="cn.itcast.controller"**/>  
  
 *<!--视图解析器对象-->  
 <!--视图解析器对象-->*<**bean id="internalResourceViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver"**>  
 <**property name="prefix" value="/WEB-INF/pages/"**/>  
 <**property name="suffix" value=".jsp"**/>  
</**bean**>



如果web-inf前面少了一个/，在requestmapping中，配置多路径就会报错  
  
 *<!--开启SpringMVC框架的支持-->* <**mvc:annotation-driven**/>  
</**beans**>

</**beans**>

## Controller层

**将拦截的请求重新分配，由视图解析器解析视图**

package cn.itcast.controller;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

/\*\*

\* 控制器

\* @author rt

\*/

@Controller

public class HelloController {

/\*\*

\* 接收请求

\* @return

\*/

@RequestMapping(path="/hello")

public String sayHello() {

System.out.println("Hello SpringMVC!!");

return "success";

}

}

## 配置顺序

入门案例的执行过程分析

1. 入门案例的执行流程

1. 当启动Tomcat服务器的时候，因为配置了load-on-startup标签，所以会创建DispatcherServlet对象，

就会加载springmvc.xml配置文件

2. 开启了注解扫描，那么HelloController对象就会被创建

3. 从index.jsp发送请求，请求会先到达DispatcherServlet核心控制器，根据配置@RequestMapping注解

找到执行的具体方法

4. 根据执行方法的返回值，再根据配置的视图解析器，去指定的目录下查找指定名称的JSP文件

5. Tomcat服务器渲染页面，做出响应

SpringMVC官方提供图形

入门案例中的组件分析

1. 前端控制器（DispatcherServlet）

2. 处理器映射器（HandlerMapping）

3. 处理器（Handler）

4. 处理器适配器（HandlAdapter）

5. 视图解析器（View Resolver）

6. 视图（View）

## RequestMapping注解

1. RequestMapping注解的作用是建立请求URL和处理方法之间的对应关系

2. RequestMapping注解可以作用在方法和类上

1. 作用在类上：第一级的访问目录

2. 作用在方法上：第二级的访问目录

3. 细节：路径可以不编写 / 表示应用的根目录开始

4. 细节：${ pageContext.request.contextPath }也可以省略不写，但是路径上不能写 /

3. RequestMapping的属性

1. path 指定请求路径的url

2. value value属性和path属性是一样的

3. mthod 指定该方法的请求方式

4. params 指定限制请求参数的条件

5. headers 发送的请求中必须包含的请求头

注意：如果要用path的话，就不能多个参数，如果要多个参数的话，就用value



### 请求参数的绑定

1. 请求参数的绑定说明

1. 绑定机制

1. 表单提交的数据都是k=v格式的 username=haha&password=123

2. SpringMVC的参数绑定过程是把表单提交的请求参数，作为控制器中方法的参数进行绑定的

3. 要求：提交表单的name和参数的名称是相同的

2. 支持的数据类型

1. 基本数据类型和字符串类型

2. 实体类型（JavaBean）

3. 集合数据类型（List、map集合等）

2. 基本数据类型和字符串类型

1. 提交表单的name和参数的名称是相同的

2. 区分大小写

3. 实体类型（JavaBean）

1. 提交表单的name和JavaBean中的属性名称需要一致

2. 如果一个JavaBean类中包含其他的引用类型，那么表单的name属性需要编写成：对象.属性 例如：

address.name

4. 给集合属性数据封装

1. JSP页面编写方式：list[0].属性

5. 请求参数中文乱码的解决

1. 在web.xml中配置Spring提供的过滤器类

6. 自定义类型转换器

1. 表单提交的任何数据类型全部都是字符串类型，但是后台定义Integer类型，数据也可以封装上，说明

Spring框架内部会默认进行数据类型转换。

2. 如果想自定义数据类型转换，可以实现Converter的接口

1. 自定义类型转换器

<!-- 配置过滤器，解决中文乱码的问题 -->

<filter>

<filter-name>characterEncodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filterclass>

<!-- 指定字符集 -->

<init-param>

<param-name>encoding</param-name>

<param-value>UTF-8</param-value>

</init-param>

</filter>

<filter-mapping>

<filter-name>characterEncodingFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

package cn.itcast.utils;

import java.text.DateFormat;

import java.text.SimpleDateFormat;

import java.util.Date;

import org.springframework.core.convert.converter.Converter;

/\*\*

\* 把字符串转换成日期的转换器

\* @author rt

\*/

public class StringToDateConverter implements Converter<String, Date>{

/\*\*

\* 进行类型转换的方法

\*/

public Date convert(String source) {

// 判断

if(source == null) {

throw new RuntimeException("参数不能为空");

}

### params

如果是params的话，就是涉及参数传递，然后就要传递参数，还是必须传入

controller层



这个是jsp页面

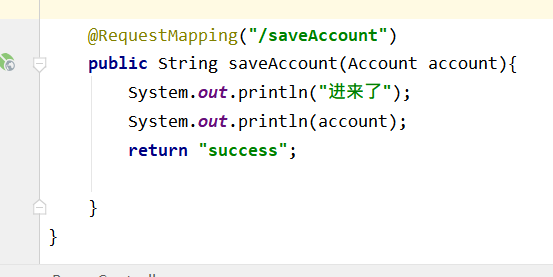


### headers属性

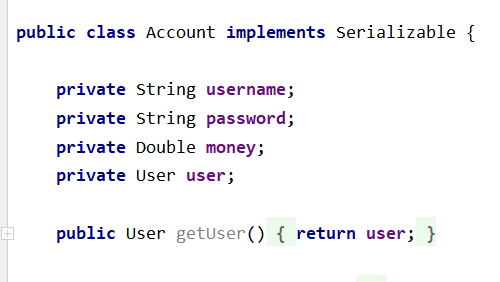
请求头

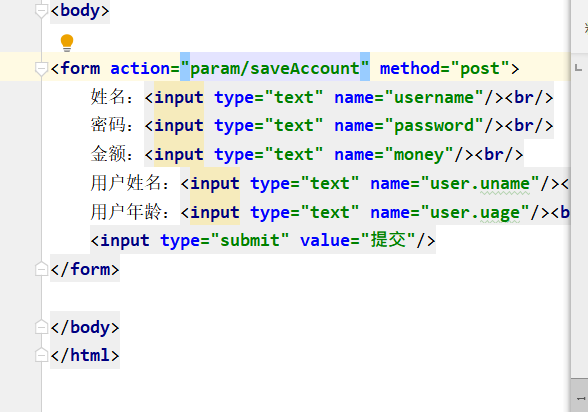
## 请求参数的绑定（简单的绑定）：

请求参数的绑定



请求的参数过来了，他会自动封装进去的

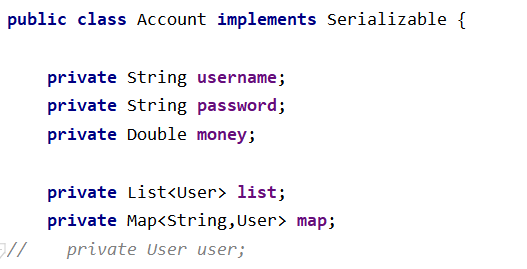




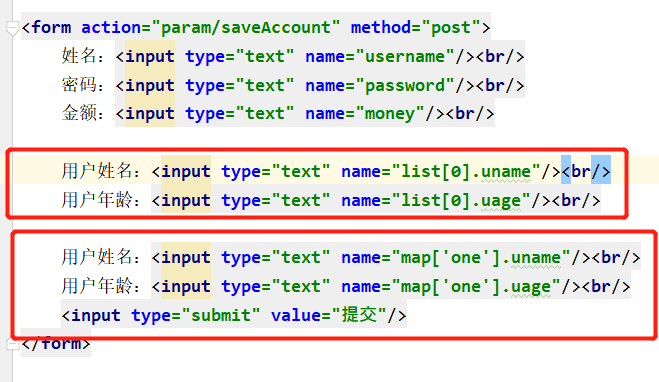
如果是内置对象，就用上面的方式，他是会自动封装进去的

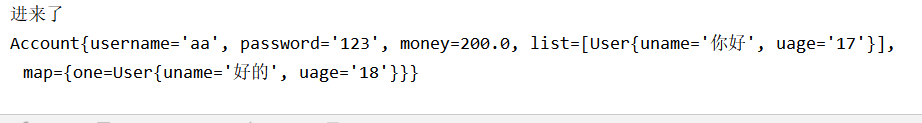
## 请求参数绑定（复杂的）：

pojo类



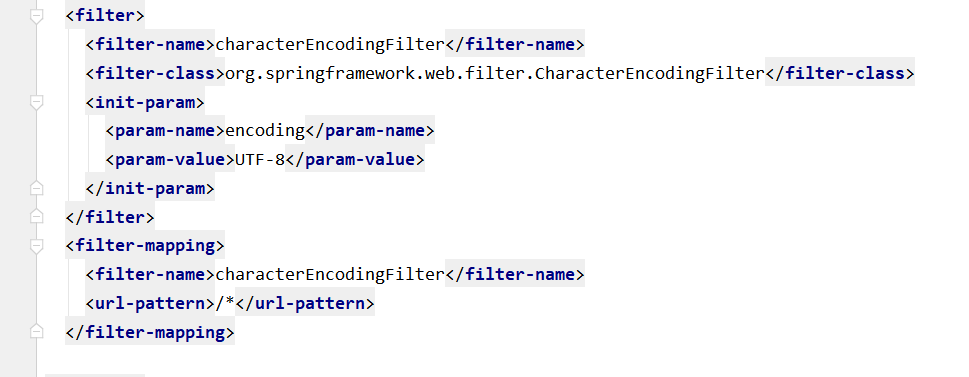
Jsp页面中





## 配置乱码问题：

拦截器，在web.xml文件中配置



记得要初始化参数，告诉他你要配的是什么的编码，就是解决哪里的问题

这里红光是没有问题的。



注意：springmvc会自动帮你进行数据转换，你提交的是age是String类型的话，他会自动帮你转为Integer

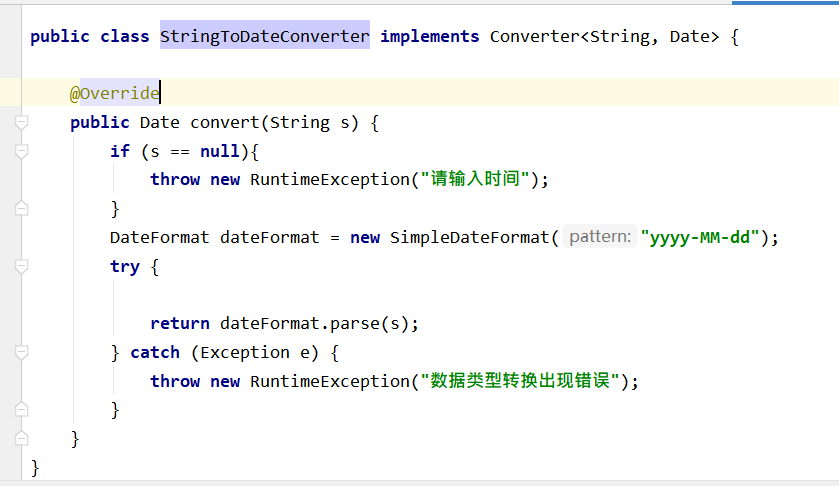
## 自定义类型转换器：

用于数据类型封装不进去

可以自己写一个类，作为类型转换器

前端控制器会自动找你的类型转换器

必须要实现一个Converter<S,T>接口，里面是两个泛型



将转换器配置到springmvc.xml文件中

就是将类型转换器，让他能够被前端控制器识别



## 获取Servlet原生的API

想要获得内置对象就直接在方法上加

## 常用的注解：

### RequestParam注解

1. 作用：把请求中的指定名称的参数传递给控制器中的形参赋值

2. 属性

1. value：请求参数中的名称

2. required：请求参数中是否必须提供此参数，默认值是true，必须提供

#### 代码如下

/\*\*

\* 接收请求

\* @return

\*/

@RequestMapping(path="/hello")

public String sayHello(@RequestParam(value="username",required=false)String name) {

System.out.println("aaaa");

System.out.println(name);

return "success";

}

### RequestBody注解

1. 作用：用于获取请求体的内容（注意：get方法不可以）

2. 属性

1. required：是否必须有请求体，默认值是true

#### 代码如下

/\*\*

\* 接收请求

\* @return

\*/

@RequestMapping(path="/hello")

public String sayHello(@RequestBody String body) {

System.out.println("aaaa");

System.out.println(body);

return "success";

}

### PathVariable注解

1. 作用：拥有绑定url中的占位符的。例如：url中有/delete/{id}，{id}就是占位符

2. 属性

1. value：指定url中的占位符名称

3. Restful风格的URL

1. 请求路径一样，可以根据不同的请求方式去执行后台的不同方法

2. restful风格的URL优点

1. 结构清晰

2. 符合标准

3. 易于理解

4. 扩展方便

### **代码如下**

<a href="user/hello/1">入门案例</a>

/\*\*

\* 接收请求

\* @return

\*/

@RequestMapping(path="/hello/{id}")

public String sayHello(@PathVariable(value="id") String id) {

System.out.println(id);

return "success";

}

是直接斜杠然后传id

### RequestHeader注解

1. 作用：获取指定请求头的值

2. 属性

1. value：请求头的名称

3. 代码如下

@RequestMapping(path="/hello")

public String sayHello(@RequestHeader(value="Accept") String header) {

System.out.println(header);

return "success";

}

### CookieValue注解

1. 作用：用于获取指定cookie的名称的值

2. 属性

1. value：cookie的名称

3. 代码

@RequestMapping(path="/hello")

public String sayHello(@CookieValue(value="JSESSIONID") String cookieValue) {

System.out.println(cookieValue);

return "success";

}

### ModelAttribute注解

1. 作用

1. 出现在方法上：表示当前方法会在控制器方法执行前线执行。

2. 出现在参数上：获取指定的数据给参数赋值。

2. 应用场景

1. 当提交表单数据不是完整的实体数据时，保证没有提交的字段使用数据库原来的数据。

3. 具体的代码

1. 修饰的方法有返回值

2. 修饰的方法没有返回值

/\*\*

\* 作用在方法，先执行

\* @param name

\* @return

\*/

@ModelAttribute

public User showUser(String name) {

System.out.println("showUser执行了...");

// 模拟从数据库中查询对象

User user = new User();

user.setName("哈哈");

user.setPassword("123");

user.setMoney(100d);

return user;

}

/\*\*

\* 修改用户的方法

\* @param cookieValue

\* @return

\*/

@RequestMapping(path="/updateUser")

public String updateUser(User user) {

System.out.println(user);

return "success";

}

/\*\*

\* 作用在方法，先执行

\* @param name

\* @return

\*/

@ModelAttribute

public void showUser(String name,Map<String, User> map) {

System.out.println("showUser执行了...");

// 模拟从数据库中查询对象

User user = new User();

user.setName("哈哈");

user.setPassword("123");

user.setMoney(100d);

map.put("abc", user);

}

/\*\*

\* 修改用户的方法

\* @param cookieValue

\* @return

\*/

RequestMapping(path="/updateUser")

public String updateUser(@ModelAttribute(value="abc") User user) {

System.out.println(user);

return "success";

}

@Controller

@RequestMapping(path="/user")

@SessionAttributes(value= {"username","password","age"},types=

{String.class,Integer.class}) // 把数据存入到session域对象中

public class HelloController {

/\*\*

\* 向session中存入值

\* @return

\*/

@RequestMapping(path="/save")

public String save(Model model) {

System.out.println("向session域中保存数据");

model.addAttribute("username", "root");

model.addAttribute("password", "123");

model.addAttribute("age", 20);

return "success";

}

/\*\*

\* 从session中获取值

\* @return

\*/

@RequestMapping(path="/find")

public String find(ModelMap modelMap) {

String username = (String) modelMap.get("username");

String password = (String) modelMap.get("password");

Integer age = (Integer) modelMap.get("age");

System.out.println(username + " : "+password +" : "+age);

return "success";

}

/\*\*

\* 清除值

\* @return

\*/

@RequestMapping(path="/delete")

public String delete(SessionStatus status) {

status.setComplete();

return "success";

}

}