**MyBatis**

第二天

# 课程计划

1. 输入映射
2. 输出映射
3. 动态SQL
4. 关联映射
5. Spring集成MyBatis
6. 反向工程
7. JDK动态代理

# 输入映射

## 入参POJO

User对象包含username属性，比如我们把User对象的实例作为入参，想根据User.username模糊查询用户信息。

### 新建工程

新建工程mybatis-03(可以直接复制mybatis-02,稍微修改即可，这里不做详细介绍了)

### 创建UserMapper接口

|  |
| --- |
| **package** cn.itcast.mybatis.mapper; **import** cn.itcast.mybatis.model.User; **public interface** UserMapper {   */\*\*  \* 根据用户名字模糊查询  \*  \** ***@param user*** *\** ***@return*** *\*/* User getUserByName(User user); } |

### 创建MyBatis.xml

|  |
| --- |
| *<?***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**>  *<!--properties-->* <**properties resource="jdbc.properties"** />   *<!--别名-->* <**typeAliases**>  *<!--给cn.itcast.mybatis.model包下的所有实体Bean取别名，别名为类名-->* <**package name="cn.itcast.mybatis.model"** />  </**typeAliases**>    *<!--数据源配置-->* <**environments default="development"**>  <**environment id="development"**>  <**transactionManager type="JDBC"**/>  <**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.pwd}"**/>  </**dataSource**>  </**environment**>  </**environments**>   *<!--映射文件指定-->* <**mappers**>  <**package name="cn.itcast.mybatis.mapper"** />  </**mappers**> </**configuration**> |

### 创建UserMapper.xml

入参为User对象时，对应的SQL语句对应占位符取值方法#{入参属性名} 其中#{}为固定写法，如下SELECT *\** FROM user WHERE userid=#{userid} 其中#{userid}表示取入参User的userid属性。

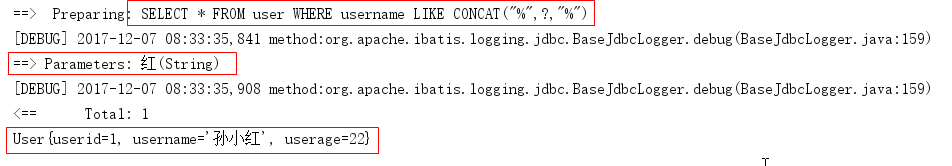
|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"** *?>* **<!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd"*>*** <**mapper namespace="cn.itcast.mybatis.mapper.UserMapper"**>  *<!--getUserByName-->* <**select id="getUserByName" parameterType="User" resultType="User"**>  SELECT *\** FROM user WHERE username LIKE CONCAT("%",#{username},"%")  </**select**> </**mapper**> |

### 编写测试

创建MyBatisTest.java

|  |
| --- |
| **public class** MyBatisTest {   *//SqlSession会话* **private** SqlSession **session**;   **private** UserMapper **userMapper**;   @Before  **public void** init() {  *//获得SqlSession* **session** = MyBatisUtils.*getSession*();   *//获取UserDao的代理对象* **userMapper** = **session**.getMapper(UserMapper.**class**);  }   */\*\*  \* 入参为JavaBean  \* 根据用户名模糊查询用户信息  \** ***@return*** *\*/* @Test  **public void** testGetUserByName(){  User user = **new** User();  user.setUsername(**"红"**);  User userinfo = **userMapper**.getUserByName(user);  System.***out***.println(userinfo);  } } |

测试结果



## 入参Map

### 修改UserMapper.xml

添加一个select节点

|  |
| --- |
| *<!--getUserByMap-->* <**select id="getUserByMap" parameterType="java.util.Map" resultType="User"**>  SELECT *\** FROM USER WHERE username=#{user\_name} AND userage=#{user\_age} </**select**> |

### 修改UserMapper接口

接口中增加如下方法

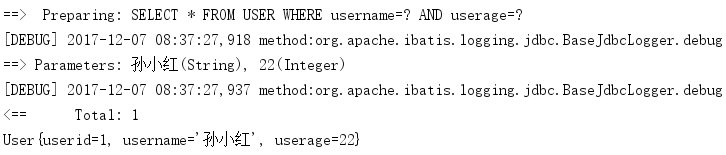
|  |
| --- |
| */\*\*\*  \* 根据Map查询数据  \** ***@param dataMap*** *\** ***@return*** *\*/* List<User> getUserByMap(Map<String, Object> dataMap); |

### 编写测试

在MyBatisTest.java中加入如下测试代码

|  |
| --- |
| */\*\*\*  \* 基于Map入参查询  \*/* @Test **public void** testGetByMap(){  *//创建Map，设置查询条件* Map<String,Object> dataMap = **new** HashMap<String,Object>();  dataMap.put(**"user\_name"**,**"孙小红"**);  dataMap.put(**"user\_age"**,22);   *//执行查询* List<User> users = **userMapper**.getUserByMap(dataMap);  **for** (User user : users) {  System.***out***.println(user);  } } |

测试结果



## 数据类型别名参照表

上面我们说了java.util.Map，每次都需要写Map的全限定名，MyBatis给java的大部分数据类型注册了别名，每次只用输入别名即可，而不需要输入全限定名。

别名参考类：org.apache.ibatis.type.TypeAliasRegistry

代码：

|  |
| --- |
| **public** TypeAliasRegistry() {  **this**.registerAlias(**"string"**, String.**class**);  **this**.registerAlias(**"byte"**, Byte.**class**);  **this**.registerAlias(**"long"**, Long.**class**);  **this**.registerAlias(**"short"**, Short.**class**);  **this**.registerAlias(**"int"**, Integer.**class**);  **this**.registerAlias(**"integer"**, Integer.**class**);  **this**.registerAlias(**"double"**, Double.**class**);  **this**.registerAlias(**"float"**, Float.**class**);  **this**.registerAlias(**"boolean"**, Boolean.**class**);  **this**.registerAlias(**"byte[]"**, Byte[].**class**);  **this**.registerAlias(**"long[]"**, Long[].**class**);  **this**.registerAlias(**"short[]"**, Short[].**class**);  **this**.registerAlias(**"int[]"**, Integer[].**class**);  **this**.registerAlias(**"integer[]"**, Integer[].**class**);  **this**.registerAlias(**"double[]"**, Double[].**class**);  **this**.registerAlias(**"float[]"**, Float[].**class**);  **this**.registerAlias(**"boolean[]"**, Boolean[].**class**);  **this**.registerAlias(**"\_byte"**, Byte.TYPE);  **this**.registerAlias(**"\_long"**, Long.TYPE);  **this**.registerAlias(**"\_short"**, Short.TYPE);  **this**.registerAlias(**"\_int"**, Integer.TYPE);  **this**.registerAlias(**"\_integer"**, Integer.TYPE);  **this**.registerAlias(**"\_double"**, Double.TYPE);  **this**.registerAlias(**"\_float"**, Float.TYPE);  **this**.registerAlias(**"\_boolean"**, Boolean.TYPE);  **this**.registerAlias(**"\_byte[]"**, **byte**[].**class**);  **this**.registerAlias(**"\_long[]"**, **long**[].**class**);  **this**.registerAlias(**"\_short[]"**, **short**[].**class**);  **this**.registerAlias(**"\_int[]"**, **int**[].**class**);  **this**.registerAlias(**"\_integer[]"**, **int**[].**class**);  **this**.registerAlias(**"\_double[]"**, **double**[].**class**);  **this**.registerAlias(**"\_float[]"**, **float**[].**class**);  **this**.registerAlias(**"\_boolean[]"**, **boolean**[].**class**);  **this**.registerAlias(**"date"**, Date.**class**);  **this**.registerAlias(**"decimal"**, BigDecimal.**class**);  **this**.registerAlias(**"bigdecimal"**, BigDecimal.**class**);  **this**.registerAlias(**"biginteger"**, BigInteger.**class**);  **this**.registerAlias(**"object"**, Object.**class**);  **this**.registerAlias(**"date[]"**, Date[].**class**);  **this**.registerAlias(**"decimal[]"**, BigDecimal[].**class**);  **this**.registerAlias(**"bigdecimal[]"**, BigDecimal[].**class**);  **this**.registerAlias(**"biginteger[]"**, BigInteger[].**class**);  **this**.registerAlias(**"object[]"**, Object[].**class**);  **this**.registerAlias(**"map"**, Map.**class**);  **this**.registerAlias(**"hashmap"**, HashMap.**class**);  **this**.registerAlias(**"list"**, List.**class**);  **this**.registerAlias(**"arraylist"**, ArrayList.**class**);  **this**.registerAlias(**"collection"**, Collection.**class**);  **this**.registerAlias(**"iterator"**, Iterator.**class**);  **this**.registerAlias(**"ResultSet"**, ResultSet.**class**); } |

### 改造UserMapper.xml

改造前：

|  |
| --- |
| *<!--getUserByMap-->* <**select id="getUserByMap" parameterType="java.util.Map" resultType="User"**>  SELECT *\** FROM USER WHERE username=#{user\_name} AND userage=#{user\_age} </**select**> |

改造后：

|  |
| --- |
| *<!--getUserByMap-->* <**select id="getUserByMap" parameterType="map" resultType="User"**>  SELECT *\** FROM USER WHERE username=#{user\_name} AND userage=#{user\_age} </**select**> |

改造后效果和改造前一样。

# 输出映射

## int类型输出

统计当前有多少用户，需要返回int数据类型。

### 修改UserMapper.xml

在UserMapper.xml增加如下代码

|  |
| --- |
| *<!--getUserCount-->* <**select id="getUserCount" resultType="java.lang.Integer"**>  SELECT *COUNT*(*\**) FROM user </**select**> |

### 修改UserMapper接口

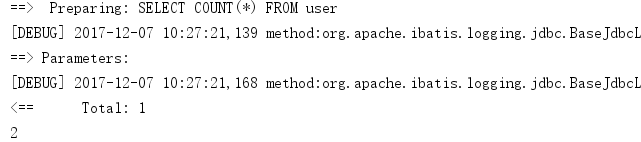
增加如下代码

|  |
| --- |
| */\*\*\*  \* 查询用户总数  \** ***@return*** *\*/* **public** Integer getUserCount(); |

### 编写测试

|  |
| --- |
| */\*\*\*  \* 查询用户总数  \*/* @Test **public void** testGetUserCount(){  Integer count = **userMapper**.getUserCount();  System.***out***.println(count); } |

测试结果



## 其他数据类型别名参照表

其他类型参照下面注册代码，如果没有注册，直接写类的全限定名。

|  |
| --- |
| **public** TypeAliasRegistry() {  **this**.registerAlias(**"string"**, String.**class**);  **this**.registerAlias(**"byte"**, Byte.**class**);  **this**.registerAlias(**"long"**, Long.**class**);  **this**.registerAlias(**"short"**, Short.**class**);  **this**.registerAlias(**"int"**, Integer.**class**);  **this**.registerAlias(**"integer"**, Integer.**class**);  **this**.registerAlias(**"double"**, Double.**class**);  **this**.registerAlias(**"float"**, Float.**class**);  **this**.registerAlias(**"boolean"**, Boolean.**class**);  **this**.registerAlias(**"byte[]"**, Byte[].**class**);  **this**.registerAlias(**"long[]"**, Long[].**class**);  **this**.registerAlias(**"short[]"**, Short[].**class**);  **this**.registerAlias(**"int[]"**, Integer[].**class**);  **this**.registerAlias(**"integer[]"**, Integer[].**class**);  **this**.registerAlias(**"double[]"**, Double[].**class**);  **this**.registerAlias(**"float[]"**, Float[].**class**);  **this**.registerAlias(**"boolean[]"**, Boolean[].**class**);  **this**.registerAlias(**"\_byte"**, Byte.TYPE);  **this**.registerAlias(**"\_long"**, Long.TYPE);  **this**.registerAlias(**"\_short"**, Short.TYPE);  **this**.registerAlias(**"\_int"**, Integer.TYPE);  **this**.registerAlias(**"\_integer"**, Integer.TYPE);  **this**.registerAlias(**"\_double"**, Double.TYPE);  **this**.registerAlias(**"\_float"**, Float.TYPE);  **this**.registerAlias(**"\_boolean"**, Boolean.TYPE);  **this**.registerAlias(**"\_byte[]"**, **byte**[].**class**);  **this**.registerAlias(**"\_long[]"**, **long**[].**class**);  **this**.registerAlias(**"\_short[]"**, **short**[].**class**);  **this**.registerAlias(**"\_int[]"**, **int**[].**class**);  **this**.registerAlias(**"\_integer[]"**, **int**[].**class**);  **this**.registerAlias(**"\_double[]"**, **double**[].**class**);  **this**.registerAlias(**"\_float[]"**, **float**[].**class**);  **this**.registerAlias(**"\_boolean[]"**, **boolean**[].**class**);  **this**.registerAlias(**"date"**, Date.**class**);  **this**.registerAlias(**"decimal"**, BigDecimal.**class**);  **this**.registerAlias(**"bigdecimal"**, BigDecimal.**class**);  **this**.registerAlias(**"biginteger"**, BigInteger.**class**);  **this**.registerAlias(**"object"**, Object.**class**);  **this**.registerAlias(**"date[]"**, Date[].**class**);  **this**.registerAlias(**"decimal[]"**, BigDecimal[].**class**);  **this**.registerAlias(**"bigdecimal[]"**, BigDecimal[].**class**);  **this**.registerAlias(**"biginteger[]"**, BigInteger[].**class**);  **this**.registerAlias(**"object[]"**, Object[].**class**);  **this**.registerAlias(**"map"**, Map.**class**);  **this**.registerAlias(**"hashmap"**, HashMap.**class**);  **this**.registerAlias(**"list"**, List.**class**);  **this**.registerAlias(**"arraylist"**, ArrayList.**class**);  **this**.registerAlias(**"collection"**, Collection.**class**);  **this**.registerAlias(**"iterator"**, Iterator.**class**);  **this**.registerAlias(**"ResultSet"**, ResultSet.**class**); } |

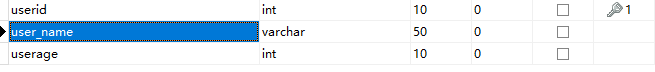
## resultType

resultType主要作用用于告诉MyBatis查询的结果集需要转换成什么数据类型，如果数据库表的列和javaBean的属性名字一一对应，则会通过反射机制自动映射并转换。

假如现在有这么一个需求，数据库username字段变成了user\_name，我们按照原来的查询条件进行查询

### 修改数据库表结构

修改user表，将username改成user\_name



### 修改UserMapper.xml

增加如下代码

|  |
| --- |
| *<!--getUserById-->* <**select id="getUserById" parameterType="int" resultType="User"**>  SELECT *\** FROM USER WHERE userid=#{id} </**select**> |

### 修改UserMapper接口

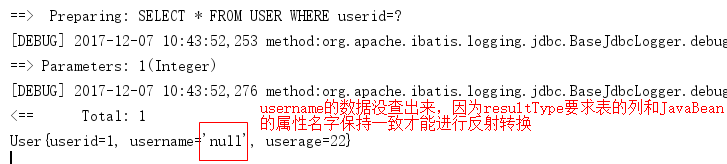
增加如下代码

|  |
| --- |
| */\*\*  \* 根据ID查询用户  \** ***@param userid*** *\** ***@return*** *\*/* **public** User getUserById(**int** userid); |

### 编写测试

|  |
| --- |
| */\*\*\*  \* 根据ID查询用户信息  \*/* @Test **public void** testGetUserById(){  **int** userid = 1;  User user = **userMapper**.getUserById(userid);  System.***out***.println(user); } |

测试结果



### 解决列和属性不一致映射问题

像这个问题怎么解决呢？SQL可以用别名代替是不是就能解决这个问题！

#### 修改UserMapper.xml

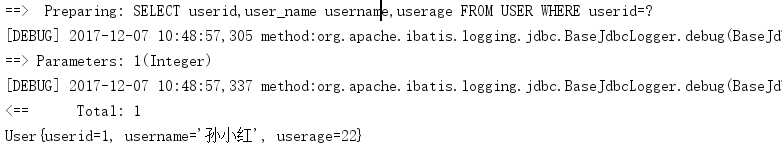
改造前

|  |
| --- |
| *<!--getUserById-->* <**select id="getUserById" parameterType="int" resultType="User"**>  SELECT \* FROM USER WHERE userid=#{id} </**select**> |

改造后

|  |
| --- |
| *<!--getUserById-->* <**select id="getUserById" parameterType="int" resultType="User"**>  SELECT userid,user\_name username,userage FROM USER WHERE userid=#{id} </**select**> |

测试结果



## resultMap映射配置

针对上述映射关系问题，这里还提供另外一种解决方案，resultMap，resultMap更像Hibernate里面的hbm.xml文件的映射功能，提供了数据库表字段和实体Bean之间属性的一种映射关系，然后根据映射关系进行转换。

### 添加resultMap节点

在UserMapper.xml中添加resultMap节点，代码如下：

|  |
| --- |
| *<!--UserResultMap-->* <**resultMap id="UserResultMap" type="User"**>  <**id column="userid" property="userid"**/>  <**result column="user\_name" property="username"** />  <**result column="userage" property="userage"** /> </**resultMap**> |

这里id=”UserResultMap”标示给当前resultMap取一个唯一标识，其他地方调用它可以通过该id来查找，type=”User”表示该resultMap最终映射转换成什么数据类型。

column主要对应SQL语句中查询出的列名，property主要对应JavaBean的属性名。

id主要用来映射主键列和属性对应关系，有去除重复的功能。

result主要用来映射普通列和属性对应关系。

### resultType换成resultMap

改造前

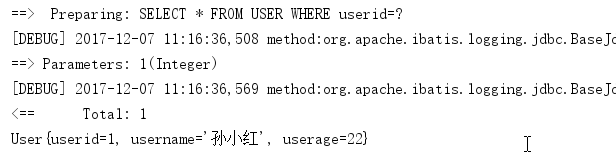
|  |
| --- |
| *<!--getUserById-->* <**select id="getUserById" parameterType="int" resultType="User"**>  SELECT userid,user\_name username,userage FROM USER WHERE userid=#{id} </**select**> |

改造后

|  |
| --- |
| <**select id="getUserById" parameterType="int" resultMap="UserResultMap"**>  SELECT *\** FROM USER WHERE userid=#{id} </**select**> |

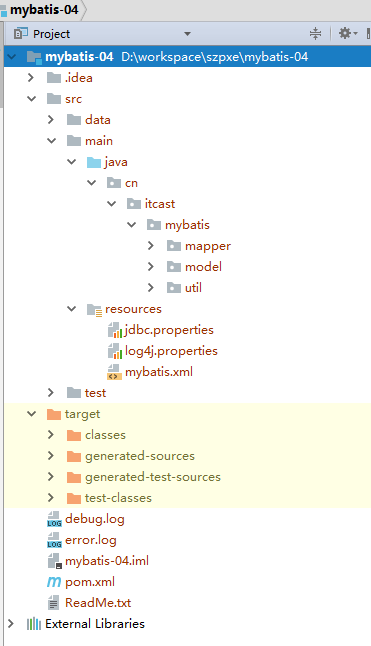
resultType换成了resultMap，resultMap的值换成了resultMap节点的id值,查询语句也没有用别名了。

测试结果



# 动态SQL

将项目复制一份，改名mybatis-04，结构如下图：



## If条件判断

在平时开发中，往往会有条件查询，例如：用户名字,用户年龄范围进行多条件查询。当用户名不为空时，根据用户名模糊查找，用户年不为空时候，根据年龄查找。

Mybatis动态SQL提供了if条件判断。接着我们基于MyBatis动态SQL提供的if条件判断完成上面的功能。

根据用户名字和年龄进行多条件查询

用户名字不为空：模糊查询

用户年龄不为空：根据用户年龄查询

用户名字不为空、用户年龄为空

用户ID：

### 修改UserMapper.xml

|  |
| --- |
| *<!--getUserByMap-->* <**select id="getUserByMap" parameterType="map" resultMap="UserResultMap"**>  SELECT *\** FROM USER  WHERE 1=1  <**if test="user\_name!=null"**>  AND user\_name like CONCAT("%",#{user\_name},"%")  </**if**>  <**if test="user\_age!=null"**>  AND userage=#{user\_age}  </**if**> </**select**> |

注意：

上面WHERE后面加了个1=1是为了防止user\_name或者user\_age为空时候，造成SQL语句为SELECT \* FROM USER WHERE AND.... 的情况出现。

### 修改UserMapper接口

|  |
| --- |
| **public interface** UserMapper {  */\*\*\*  \* 根据Map查询数据  \** ***@param dataMap*** *\** ***@return*** *\*/* List<User> getUserByMap(Map<String, Object> dataMap); } |

### 编写测试

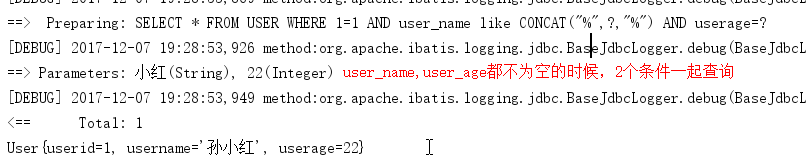
|  |
| --- |
| **public class** MyBatisTest {   *//SqlSession会话* **private** SqlSession **session**;   **private** UserMapper **userMapper**;   @Before  **public void** init() {  *//获得SqlSession* **session** = MyBatisUtils.*getSession*();   *//获取UserDao的代理对象* **userMapper** = **session**.getMapper(UserMapper.**class**);  }   */\*\*\*  \* 基于Map入参查询  \*/* @Test  **public void** testGetByMap(){  *//创建Map，设置查询条件* Map<String,Object> dataMap = **new** HashMap<String,Object>();  dataMap.put(**"user\_name"**,**"孙小红"**);  dataMap.put(**"user\_age"**,22);   *//执行查询* List<User> users = **userMapper**.getUserByMap(dataMap);  **for** (User user : users) {  System.***out***.println(user);  }  } } |

### 测试结果分析

user\_name,user\_age都不为空：

SQL语句：

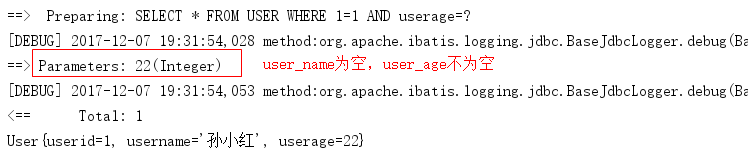
SELECT \* FROM USER WHERE 1=1 AND user\_name like CONCAT("%",?,"%") AND userage=?



user\_name为空，user\_age不为空:

SQL语句：

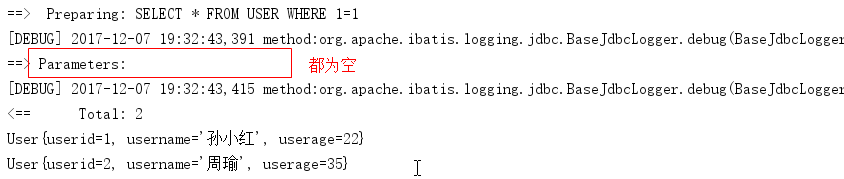
SELECT \* FROM USER WHERE 1=1 AND userage=?



user\_name,user\_age都为空：

SQL语句：

SELECT \* FROM USER WHERE 1=1



## choose, when, otherwise

在平时开发中，往往会有条件查询，例如：用户ID，用户名字,用户年龄范围进行多条件查询。当用户名不为空时，根据用户名模糊查找，用户年不为空时候，根据年龄查找。当用户ID不为空时，只根据ID查询。

Mybatis动态SQL提供了choose条件判断，它有2个子节点when和otherwise，这2个都是用于控制流程的。看下面一段代码：

|  |
| --- |
| **int** i =1; **if**(i==1){  System.***out***.println(**"张三"**); }**else if**(i==2){  System.***out***.println(**"小红"**); }**else if**(i==3){  System.***out***.println(**"小白"**); }**else**{  System.***out***.println(**"出了鬼。。。。"**); } |

when就相当于上面流程控制代码的if(){}else if(){}的条件控制流程

otherwise相当于上面流程控制代码的else{}的控制流程

接着我们基于MyBatis动态SQL提供的choose条件判断完成上面的功能。

### 修改UserMapper.xml

在UserMapper.xml里面增加如下代码

|  |
| --- |
| *<!--getUserByCondition-->* <**select id="getUserByCondition" parameterType="map" resultMap="UserResultMap"**>  SELECT *\** FROM USER  WHERE 1=1  <**choose**>  <**when test="user\_id!=null"**>  AND userid=#{user\_id}  </**when**>  <**otherwise**>  <**if test="user\_name!=null"**>  AND user\_name like CONCAT("%",#{user\_name},"%")  </**if**>  <**if test="user\_age!=null"**>  AND userage=#{user\_age}  </**if**>  </**otherwise**>  </**choose**> </**select**> |

### 修改UserMapper接口

增加如下代码

|  |
| --- |
| */\*\*\*  \* 条件查询用户  \** ***@param dataMap*** *\** ***@return*** *\*/* List<User> getUserByCondition(Map<String, Object> dataMap); |

### 编写测试

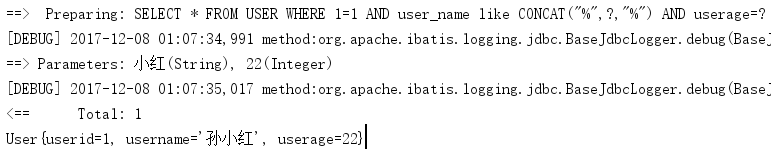
|  |
| --- |
| */\*\*\*  \* Choose测试  \*/* @Test **public void** testGetUserByCondition(){  *//创建Map，设置查询条件* Map<String,Object> dataMap = **new** HashMap<String,Object>();  dataMap.put(**"user\_id"**,1);  dataMap.put(**"user\_name"**,**"小红"**);  dataMap.put(**"user\_age"**,22);   *//执行查询* List<User> users = **userMapper**.getUserByCondition(dataMap);  **for** (User user : users) {  System.***out***.println(user);  } } |

### 测试结果分析

user\_id为空

SQL语句：

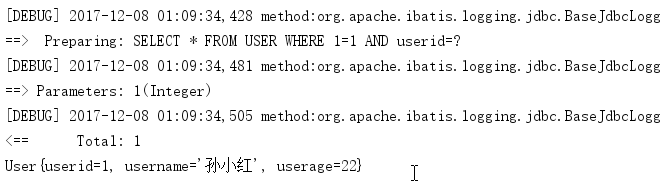
SELECT \* FROM USER WHERE 1=1 AND user\_name like CONCAT("%",?,"%") AND userage=?



user\_id不为空

SQL语句：

SELECT \* FROM USER WHERE 1=1 AND userid=?



## where

前面我们每次要在where条件后面加上1=1才能保证SQL语句正确，MyBatis提供了一个where标签，用于控制条件语句的。

被where元素包含的标记返回任意内容,会在返回的内容前插入“WHERE” 。,如果以“AND”或“OR”开头的内容,那么会过去掉AND或者OR。

### 修改UserMapper.xml

改造getUserByCondition在UserMapper.xml中的代码

改造前：

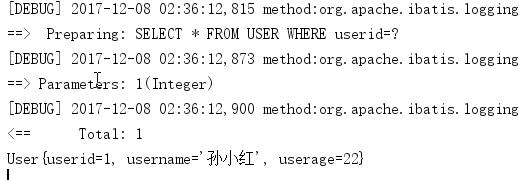
|  |
| --- |
| *<!--getUserByCondition-->* <**select id="getUserByCondition" parameterType="map" resultMap="UserResultMap"**>  SELECT *\** FROM USER  WHERE 1=1  <**choose**>  <**when test="user\_id!=null"**>  AND userid=#{user\_id}  </**when**>  <**otherwise**>  <**if test="user\_name!=null"**>  AND user\_name like CONCAT("%",#{user\_name},"%")  </**if**>  <**if test="user\_age!=null"**>  AND userage=#{user\_age}  </**if**>  </**otherwise**>  </**choose**> </**select**> |

改造后：

|  |
| --- |
| <**select id="getUserByCondition" parameterType="map" resultMap="UserResultMap"**>  SELECT *\** FROM USER  <**where**>  <**choose**>  <**when test="user\_id!=null"**>  AND userid=#{user\_id}  </**when**>  <**otherwise**>  <**if test="user\_name!=null"**>  AND user\_name like CONCAT("%",#{user\_name},"%")  </**if**>  <**if test="user\_age!=null"**>  AND userage=#{user\_age}  </**if**>  </**otherwise**>  </**choose**>  </**where**> </**select**> |

改造后，将1=1去掉了，增加了where节点。

测试结果：



## trim

上面的实现方案也可以用trim来实现，它比where标签更灵活。我们把上面代码进行改造。

### 修改UserMapper.xml

修改前：

|  |
| --- |
| <**select id="getUserByCondition" parameterType="map" resultMap="UserResultMap"**>  SELECT *\** FROM USER  <**where**>  <**choose**>  <**when test="user\_id!=null"**>  AND userid=#{user\_id}  </**when**>  <**otherwise**>  <**if test="user\_name!=null"**>  AND user\_name like CONCAT("%",#{user\_name},"%")  </**if**>  <**if test="user\_age!=null"**>  AND userage=#{user\_age}  </**if**>  </**otherwise**>  </**choose**>  </**where**> </**select**> |

修改后：

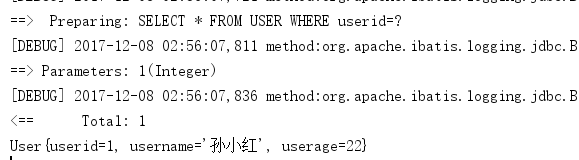
|  |
| --- |
| <**select id="getUserByCondition" parameterType="map" resultMap="UserResultMap"**>  SELECT *\** FROM USER  <**trim prefixOverrides="OR|AND" prefix="WHERE"**>  <**choose**>  <**when test="user\_id!=null"**>  AND userid=#{user\_id}  </**when**>  <**otherwise**>  <**if test="user\_name!=null"**>  AND user\_name like CONCAT("%",#{user\_name},"%")  </**if**>  <**if test="user\_age!=null"**>  AND userage=#{user\_age}  </**if**>  </**otherwise**>  </**choose**>  </**trim**> </**select**> |

说明：

prefixOverrides="OR|AND" 表示在返回内容是否为OR或者AND开始，如果是，则将他们去掉。

prefix="WHERE" 标示当trim里面包含的节点返回内容时，在返回的内容前面增加一个前缀，前缀为where。

测试结果:



## set

在开发中，往往需要动态修改数据，比如修改用户表信息，传入User实体Bean，每次需要根据User判断，哪些属性有值则修改，没值则不改。MyBatis提供了set标签很好的解决了这个问题。

### 修改UserMapper.xml

增加如下代码

|  |
| --- |
| *<!--updateUser-->* <**update id="updateUserById" parameterType="User"**>  UPDATE USER  <**set**>  <**if test="username!=null"**>  user\_name=#{username},  </**if**>  <**if test="userage!=null"**>  userage=#{userage},  </**if**>  </**set**>  WHERE  userid=#{userid} </**update**> |

### 修改UserMapper接口

增加如下代码

|  |
| --- |
| */\*\*\*  \* 修改用户  \** ***@param user*** *\*/* **public void** updateUserById(User user); |

### 编写测试

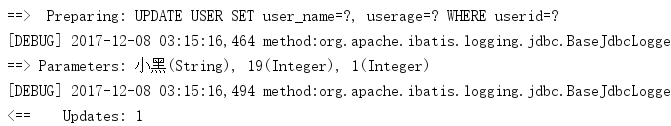
|  |
| --- |
| */\*\*\*  \* set测试  \*/* @Test **public void** testUpdateUserById(){  *//要修改的用户信息* User user = **new** User();  user.setUserid(1);  user.setUsername(**"小黑"**);  user.setUserage(19);   *//执行查询* **userMapper**.updateUserById(user);  **session**.commit(); } |

测试结果：

username、usserage全不为空

SQL语句：

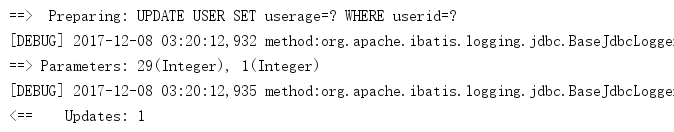
UPDATE USER SET user\_name=?, userage=? WHERE userid=?



username为空、usserage不为空

SQL语句：

UPDATE USER SET userage=? WHERE userid=?



## foreach

工作中常常需要根据多个ID查询多个用户信息，如下SQL：

select \* from user where userid in(1,2,3,4,5,6)

这个SQL语句查询要比一条一条查询效率高很多。MyBatis动态SQL提供了foreach，可以动态循环拼接where后面的条件。

### 修改UserMapper.xml

增加如下代码

|  |
| --- |
| *<!--getUserByIds-->* <**select id="getUserByIds" parameterType="int" resultMap="UserResultMap"**>  SELECT *\** FROM USER  <**foreach collection="array" item="uid" open="WHERE userid IN(" separator="," close=")"**>  #{uid}  </**foreach**> </**select**> |

????

### 修改UserMapper接口

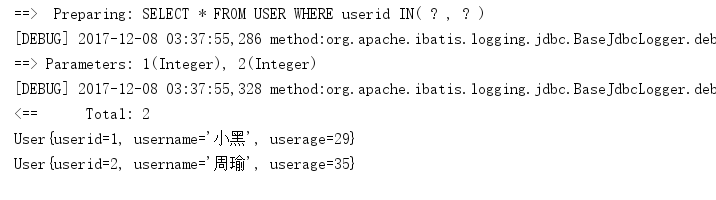
增加如下代码

|  |
| --- |
| */\*\*\*  \* 根据ID集合查询用户  \** ***@param ids*** *\** ***@return*** *\*/* **public** List<User> getUserByIds(**int**[] ids); |

### 编写测试

|  |
| --- |
| */\*\*\*  \* foreach测试  \*/* @Test **public void** testGetUserByIds(){  **int**[] ids = **new int**[2];  ids[0] = 1;  ids[1] = 2;   *//根据ID集合查询用户信息* List<User> users = **userMapper**.getUserByIds(ids);  **for** (User user : users) {  System.***out***.println(user);  } } |

测试结果



### 总结

|  |
| --- |
| *<!--getUserByIds-->* <**select id="getUserByIds" parameterType="int" resultMap="UserResultMap"**>  SELECT *\** FROM USER  <**foreach collection="array" item="uid" open="WHERE userid IN(" separator="," close=")"**>  #{uid}  </**foreach**> </**select**> |

对上面的代码我们进行一下解析和总结。

我们最终是为了实现SELECT *\** FROM USER WHERE userid IN(1,2)

foreach节点表示循环某一集合数据，collection属性表示循环要取的数据，因为入参是数组，所以这里直接写array即可，item表示定义一个变量接收当前循环的对象，也就是集合中的一个对象。

我们要实现上面的SQL语句，前面一部分已经[SELECT *\** FROM USER]已经完成，后面一部分就在foreach里完成，我们把[WHERE userid IN(1,2)]分为3部分，一部分是WHERE userid IN( ,第二部分是 1,2 ，第三部分是) 。我们可以把第一部分当做循环迭代拼接数据后要拼接的前缀，也就是open=”WHERE userid IN(”，第二部分是通过foreach循环从集合中取出的数据进行拼接，这里从循环的item=”uid”里取，每次拼接按照道理应该要追加一个逗号,而这个要拼接的逗号通过separator指定，第三部分也就是SQL语句结束的）即close=”)”。

## sql片段

我们经常会编写SELECT \* FROM USER WHERE xxx 而前面这一部分在大量重复去写，给我们增加了很多工作量，MyBaits提供了sql片段功能，可以将一部分重复代码提取出来，哪里需要调用，直接引用即可，充分重用了重复代码。

### 修改UserMapper.xml

#### 定义SQL片段

|  |
| --- |
| *<!--UserColumnSelect-->* <**sql id="UserColumnSelect"**>  SELECT *\** FROM USER </**sql**> |

#### 引用SQL片段

修改**getUserByIds**方法

修改前：

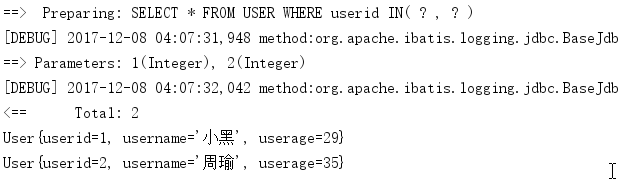
|  |
| --- |
| *<!--getUserByIds-->* <**select id="getUserByIds" parameterType="int" resultMap="UserResultMap"**>  SELECT *\** FROM USER  <**foreach collection="array" item="uid" open="WHERE userid IN(" separator="," close=")"**>  #{uid}  </**foreach**> </**select**> |

修改后：

|  |
| --- |
| <**select id="getUserByIds" parameterType="int" resultMap="UserResultMap"**>  <**include refid="UserColumnSelect"** />  <**foreach collection="array" item="uid" open="WHERE userid IN(" separator="," close=")"**>  #{uid}  </**foreach**> </**select**> |

通过include进行引用，refid=”sql片段的ID”

测试结果



# 映射关系

为了不和前面项目混响，我们这里再创建一个新的项目，将项目mybatis-04复制一份改名mybatis-05即可，这里不详细介绍了。

## 一对一关联映射

一个用户只有一张身份证信息，一张身份证信息也只属于一个人，他们的关系叫一对一关联映射。以前Hibernate通过配置映射关系来实现，MyBatis也可以通过配置文件来实现映射关系。

### 表结构

user表

|  |
| --- |
| CREATE TABLE `user` (  `userid` int(10) NOT NULL AUTO\_INCREMENT,  `user\_name` varchar(50) NOT NULL,  `userage` int(10) NOT NULL,  PRIMARY KEY (`userid`)  ) ENGINE=InnoDB AUTO\_INCREMENT=3 DEFAULT CHARSET=utf8; |

idcard表

|  |
| --- |
| CREATE TABLE `idcard` (  `cardId` varchar(40) NOT NULL,  `province` varchar(60) NOT NULL,  `city` varchar(60) NOT NULL,  `usserid` int(11) NOT NULL,  PRIMARY KEY (`cardId`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8; |

idcard表的userid和user表建立外键关系。

### 实现方案

要实现一对一关联映射，这里提出2种方案

1)一次关联查询

|  |
| --- |
| SELECT card.cardId,card.province,card.city,u.\* FROM idcard card INNER JOIN `user` u ON card.userid=u.userid |

1. 分两次查询
2. 查询出用户身份证信息,得到用户userid=1

|  |
| --- |
| SELECT \* FROM idcard WHERE cardId='429004199302554418'; |

1. 再查询出对应身份证信息的用户，将上面得到的userid=1的参数作为条件使用

|  |
| --- |
| SELECT \* FROM `user` WHERE userid=1 |

### 创建实体Bean

#### User

|  |
| --- |
| **package** cn.itcast.mybatis.model; **import** java.io.Serializable; **public class** User **implements** Serializable{  */\*\*\*  \* 用户ID  \*/* **private** Integer **userid**;   */\*\*\*  \* 用户名字  \*/* **private** String **username**;  */\*\*\*  \* 用户年龄  \*/* **private** Integer **userage**;  **public** Integer getUserid() {  **return userid**;  }  **public void** setUserid(Integer userid) {  **this**.**userid** = userid;  }  **public** String getUsername() {  **return username**;  }  **public void** setUsername(String username) {  **this**.**username** = username;  }  **public** Integer getUserage() {  **return userage**;  }  **public void** setUserage(Integer userage) {  **this**.**userage** = userage;  }   @Override  **public** String toString() {  **return "User{"** +  **"userid="** + **userid** +  **", username='"** + **username** + **'\''** +  **", userage="** + **userage** +  **'}'**;  } } |

#### IdCard

|  |
| --- |
| **package** cn.itcast.mybatis.model; **import** java.io.Serializable; **public class** IdCard **implements** Serializable {   **private** String **cardId**;  **private** String **province**;  **private** String **city**;  *//和User实现一对一关联映射* **private** User **user**;   **public** String getCardId() {  **return cardId**;  }   **public void** setCardId(String cardId) {  **this**.**cardId** = cardId;  }   **public** String getProvince() {  **return province**;  }   **public void** setProvince(String province) {  **this**.**province** = province;  }   **public** String getCity() {  **return city**;  }   **public void** setCity(String city) {  **this**.**city** = city;  }   **public** User getUser() {  **return user**;  }   **public void** setUser(User user) {  **this**.**user** = user;  }   @Override  **public** String toString() {  **return "IdCard{"** +  **"cardId='"** + **cardId** + **'\''** +  **", province='"** + **province** + **'\''** +  **", city='"** + **city** + **'\''** +  **", user="** + **user** +  **'}'**;  } } |

### 方案一：一次查询

1)一次关联查询

|  |
| --- |
| SELECT card.cardId,card.province,card.city,u.\* FROM idcard card INNER JOIN `user` u ON card.userid=u.userid |

#### 创建IdCardMapper.xml

增加如下代码

|  |
| --- |
| *<?***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.IdCardMapper"**>  *<!--UserResultMap-->* <**resultMap id="IdCardResultMap" type="IdCard"**>  <**id column="cardId" property="cardId"**/>  <**result column="province" property="province"** />  <**result column="city" property="city"** />   *<!--  一对一映射配置  property:IdCard的属性  association:配置对应属性的映射关系，和resultMap配置一样  javaType:表示association的数据最终要转换的类型  -->* <**association property="user" javaType="User"**>  <**id column="userid" property="userid"**/>  <**result column="user\_name" property="username"**/>  <**result column="userage" property="userage"**/>  </**association**>  </**resultMap**>    *<!--getCardInfo-->* <**select id="getCardInfo" parameterType="string" resultMap="IdCardResultMap"**>  SELECT  card.cardId,card.province,card.city,u.*\** FROM  idcard card  INNER JOIN  `user` u  ON  card.userid=u.userid  WHERE card.cardId=#{cardId}  </**select**> </**mapper**> |

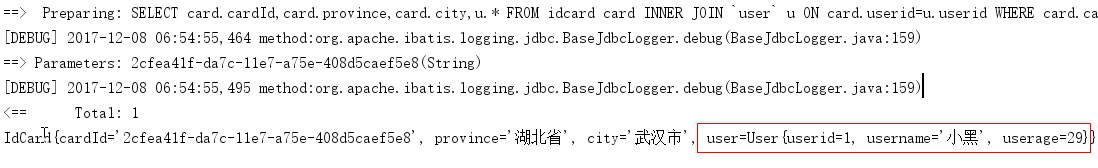
#### 创建IdCardMapper接口

|  |
| --- |
| **package** cn.itcast.mybatis.mapper; **import** cn.itcast.mybatis.model.IdCard; **public interface** IdCardMapper {   */\*\*\*  \* 根据身份证号码查询身份证信息和用户信息  \** ***@param cardId*** *\** ***@return*** *\*/* **public** IdCard getCardInfo(String cardId); } |

#### 编写测试

|  |
| --- |
| **package** cn.itcast; **import** cn.itcast.mybatis.mapper.IdCardMapper; **import** cn.itcast.mybatis.mapper.UserMapper; **import** cn.itcast.mybatis.model.IdCard; **import** cn.itcast.mybatis.util.MyBatisUtils; **import** org.apache.ibatis.session.SqlSession; **import** org.junit.Before; **import** org.junit.Test;  **public class** MyBatisTest {   *//SqlSession会话* **private** SqlSession **session**;   **private** UserMapper **userMapper**;   **private** IdCardMapper **idCardMapper**;   @Before  **public void** init() {  *//获得SqlSession* **session** = MyBatisUtils.*getSession*();   *//获取UserDao的代理对象* **userMapper** = **session**.getMapper(UserMapper.**class**);  *//获取IdCardDao的代理对象* **idCardMapper** = **session**.getMapper(IdCardMapper.**class**);  }    */\*\*\*  \* 一对一关联映射  \* 一次查询  \*/* @Test  **public void** testGetCardInfo(){  String cardId=**"2cfea41f-da7c-11e7-a75e-408d5caef5e8"**;  IdCard cardInfo = **idCardMapper**.getCardInfo(cardId);  System.***out***.println(cardInfo);  } } |

运行结果



### 方案二：二次查询

#### 修改IdCardMapper.xml

加入如下代码

|  |
| --- |
| *<!--IdCardResultMapOfSelect-->* <**resultMap id="IdCardResultMapOfSelect" type="IdCard"**>  <**id column="cardId" property="cardId"**/>  <**result column="province" property="province"** />  <**result column="city" property="city"** />   *<!--  一对一映射配置  property:IdCard的属性  association:配置对应属性的映射关系  select:调用id=getUserById的select节点  column：调用外部查询的时候将当前查询出的userid列的值传入过去  注：association这里不需要设置转换类型，因为id=getUserById的select节点返回的resultType指明了User类型  -->* <**association property="user" select="getUserById" column="userid"** /> </**resultMap**>  *<!--getIdCardByIdOfSelect-->* <**select id="getIdCardByIdOfSelect" parameterType="string" resultMap="IdCardResultMapOfSelect"**>  SELECT *\** FROM idcard WHERE cardId=#{cardId} </**select**>  *<!--getUserById-->* <**select id="getUserById" parameterType="int" resultType="User"**>  SELECT userid,user\_name username,userage FROM `user` WHERE userid=#{userid} </**select**> |

#### 修改IdCardMapper接口

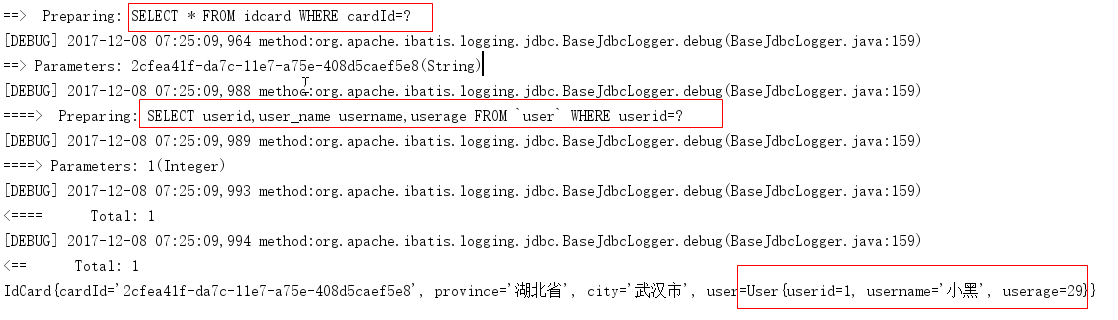
增加如下代码

|  |
| --- |
| */\*\*\*  \* 根据身份证号码查询身份证信息和用户信息  \** ***@param cardId*** *\** ***@return*** *\*/* IdCard getIdCardByIdOfSelect(String cardId); |

#### 编写测试

|  |
| --- |
| */\*\*\*  \* 一对一关联映射  \* 二次查询  \*/* @Test **public void** testGetIdCardByIdOfSelect(){  String cardId=**"2cfea41f-da7c-11e7-a75e-408d5caef5e8"**;  IdCard cardInfo = **idCardMapper**.getIdCardByIdOfSelect(cardId);  System.***out***.println(cardInfo); } |

测试结果



## 一对多

刚才我们说了一对一映射关系有2种结局方案，那么一对多呢？也有2种结局方案，而且方案相同，都是一次查询和多次查询。

模拟这么一个场景，一个用户拥有多个手机，查询用户的同时，查询出用户的所有手机信息。例如：小红有1部华为手机，1部小米手机，现在要查询出小红个人信息同时还要查出她拥有的手机信息。

### 表结构

mobile表

|  |
| --- |
| CREATE TABLE `mobile` (  `mnumber` varchar(40) NOT NULL,  `mname` varchar(20) NOT NULL,  `userid` int(10) NOT NULL,  PRIMARY KEY (`mnumber`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8; |

模拟数据



### 实现方案

1. 一次关联查询

|  |
| --- |
| SELECT u.userid,u.user\_name,u.userage FROM `user` u LEFT JOIN mobile m ON u.userid=m.userid |

1. 分二次查询
2. 先根据用户ID查询出用户信息，再根据用户ID查询用户手机信息

|  |
| --- |
| SELECT \* FROM `user` WHERE userid=1 |

1. 根据用户ID查询手机信息

|  |
| --- |
| SELECT \* FROM mobile WHERE userid =1 |

### 创建实体Bean

|  |
| --- |
| **package** cn.itcast.mybatis.model; **public class** Mobile {  **private** String **mnumber**;  **private** String **mname**;  **private** Integer **userid**;   **public** String getMnumber() {  **return mnumber**;  }   **public void** setMnumber(String mnumber) {  **this**.**mnumber** = mnumber;  }   **public** String getMname() {  **return mname**;  }   **public void** setMname(String mname) {  **this**.**mname** = mname;  }   **public** Integer getUserid() {  **return userid**;  }   **public void** setUserid(Integer userid) {  **this**.**userid** = userid;  }   @Override  **public** String toString() {  **return "Mobile{"** +  **"mnumber='"** + **mnumber** + **'\''** +  **", mname='"** + **mname** + **'\''** +  **", userid="** + **userid** +  **'}'**;  } } |

### 修改User实体Bean

因为用户User和手机信息Mobile之间的关系属于一对多的关系，所以我们需要在User对象里添加一个List<Mobile> mobiles属性,代码如下：

|  |
| --- |
| *//一对多映射关系* **private** List<Mobile> **mobiles**;  **public** List<Mobile> getMobiles() {  **return mobiles**; }  **public void** setMobiles(List<Mobile> mobiles) {  **this**.**mobiles** = mobiles; } |

### 方案一：一次查询

1)一次关联查询

|  |
| --- |
| SELECT u.userid,u.user\_name,u.userage,m.\* FROM `user` u LEFT JOIN mobile m ON u.userid=m.userid |

#### 修改UserMapper.xml

加入如下代码

|  |
| --- |
| <**resultMap id="UserResultMap" type="User"**>  <**id column="userid" property="userid"**/>  <**result column="user\_name" property="username"**/>  <**result column="userage" property="userage"**/>   *<!--  一对多关系映射  collection:实现一对多集合映射关系  property:User对应的属性  ofType:当前映射所要转的类型  -->* <**collection property="mobiles" ofType="Mobile"**>  <**id column="mnumber" property="mnumber"**/>  <**result column="mname" property="mname"**/>  </**collection**> </**resultMap**>  *<!--getUserById-->* <**select id="getUserById" parameterType="int" resultMap="UserResultMap"**>  SELECT u.userid,u.user\_name,u.userage,m.*\** FROM `user` u LEFT JOIN mobile m ON u.userid=m.userid WHERE u.userid=#{userid} </**select**> |

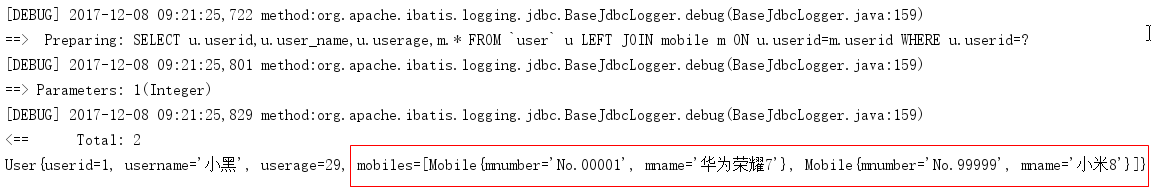
#### 修改UserMapper接口

|  |
| --- |
| **package** cn.itcast.mybatis.mapper; **import** cn.itcast.mybatis.model.User; **public interface** UserMapper {   */\*\*  \* 根据用户ID查询用户信息  \** ***@param userid*** *\** ***@return*** *\*/* User getUserById(**int** userid); } |

#### 编写测试

|  |
| --- |
| */\*\*\*\*  \* 实现一对多查询  \* 一次查询  \*/* @Test **public void** testGetUserById(){  **int** userid =1;  User user = **userMapper**.getUserById(userid);  System.***out***.println(user); } |

测试结果



### 方案二：二次查询

2)分二次查询

1. 先根据用户ID查询出用户信息，再根据用户ID查询用户手机信息

|  |
| --- |
| SELECT \* FROM `user` WHERE userid=1 |

1. 根据用户ID查询手机信息

|  |
| --- |
| SELECT \* FROM mobile WHERE userid =1 |

#### 修改UserMapper.xml

加入如下代码

|  |
| --- |
| <**resultMap id="UserResultMapOfSelect" type="User"**>  <**id column="userid" property="userid"**/>  <**result column="user\_name" property="username"**/>  <**result column="userage" property="userage"**/>   *<!--  一对多关系映射  collection:实现一对多集合映射关系  property:User对应的属性mobiles  column:将查询出来的列userid的值作为参数传给id=getMobilesByUserId的查询节点  -->* <**collection property="mobiles" select="getMobilesByUserId" column="userid"** /> </**resultMap**>  *<!--getUserByIdOfSelect-->* <**select id="getUserByIdOfSelect" parameterType="int" resultMap="UserResultMapOfSelect"**>  SELECT *\** FROM `user` WHERE userid=#{userid} </**select**>  *<!--getMobilesByUserId-->* <**select id="getMobilesByUserId" parameterType="int" resultType="Mobile"**>  SELECT *\** FROM mobile WHERE userid =#{userid} </**select**> |

#### 修改UserMapper接口

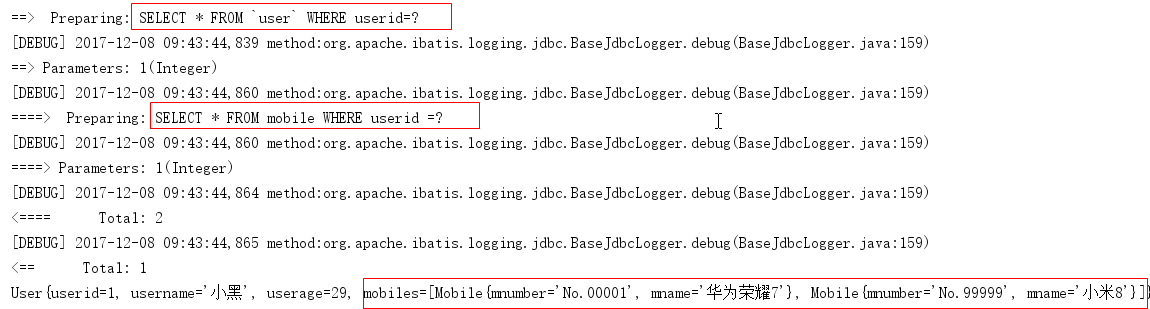
加入如下代码

|  |
| --- |
| */\*\*  \* 根据用户ID查询用户信息  \** ***@param userid*** *\** ***@return*** *\*/* User getUserByIdOfSelect(**int** userid); |

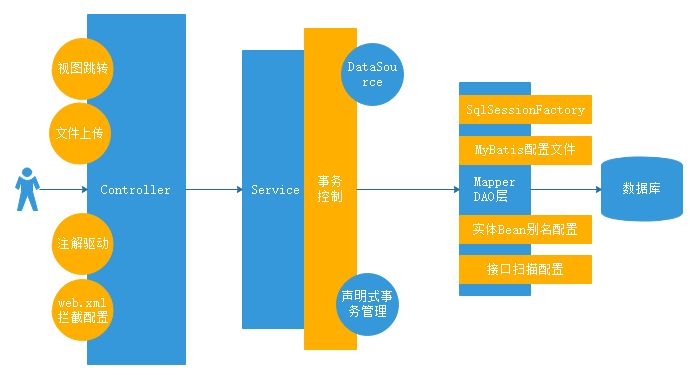
#### 编写测试

|  |
| --- |
| */\*\*\*\*  \* 实现一对多查询  \* 二次查询  \*/* @Test **public void** testGetUserByIdOfSelect(){  **int** userid =1;  User user = **userMapper**.getUserByIdOfSelect(userid);  System.***out***.println(user); } |

测试结果



# MyBatis+Spring集成

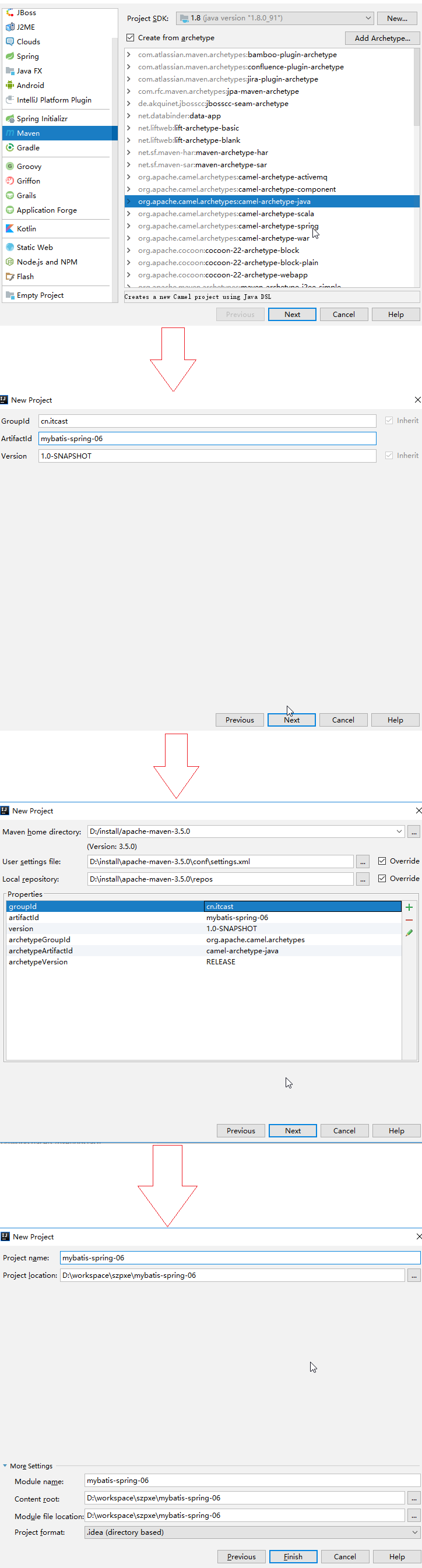


## 整合步骤

1. 创建一个java工程。
2. 导入jar包。（这里使用maven）
3. mybatis的配置文件mybatis.xml
4. 编写Spring的配置文件
5. 数据库连接及连接池
6. sqlsessionFactory对象，配置到spring容器中
7. 编写Spring的配置文件
8. 复制jdbc.properties配置文件到新工程
9. 复制log4j.properties配置文件到新工程

## 创建工程

工程名字:mybatis-spring-06,创建步骤如下图：



## 导入pom依赖

|  |
| --- |
| <**properties**>  <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  <**project.reporting.outputEncoding**>UTF-8</**project.reporting.outputEncoding**>  <**commons.beanutils**>1.9.3</**commons.beanutils**>  <**spring.version**>4.3.2.RELEASE</**spring.version**>  <**aspectj.version**>1.8.6</**aspectj.version**>  <**aspectj.weaver**>1.8.6</**aspectj.weaver**>  <**mybatis.spring.version**>1.3.0</**mybatis.spring.version**>  <**mybatis.version**>3.4.5</**mybatis.version**>  <**mysql.version**>5.1.32</**mysql.version**>  <**jolbox.version**>0.8.0.RELEASE</**jolbox.version**>  <**slf4j.version**>1.7.25</**slf4j.version**>  <**slf4j.log4j12.version**>1.7.25</**slf4j.log4j12.version**>  <**log4j.version**>1.2.17</**log4j.version**>  <**log4j.core.version**>2.3</**log4j.core.version**>  <**commons.logging.version**>1.2</**commons.logging.version**>  <**junit.version**>4.12</**junit.version**> </**properties**>   <**dependencies**>  <**dependency**>  <**groupId**>commons-beanutils</**groupId**>  <**artifactId**>commons-beanutils</**artifactId**>  <**version**>${commons.beanutils}</**version**>  </**dependency**>   *<!-- spring-core -->* <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-core</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>   *<!-- spring-context -->* <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-context</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>   *<!-- Spring AOP -->* <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-aop</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>   *<!-- AspectJ Runtime -->* <**dependency**>  <**groupId**>org.aspectj</**groupId**>  <**artifactId**>aspectjrt</**artifactId**>  <**version**>${aspectj.version}</**version**>  </**dependency**>   *<!-- AspectJ Weaver -->* <**dependency**>  <**groupId**>org.aspectj</**groupId**>  <**artifactId**>aspectjweaver</**artifactId**>  <**version**>${aspectj.weaver}</**version**>  </**dependency**>   *<!-- Spring Jdbc 的支持 -->* <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-jdbc</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>   *<!-- SpringMVC -->* <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-webmvc</**artifactId**>  <**version**>${spring.version}</**version**>  </**dependency**>    *<!-- mybatis-spring 整合 -->* <**dependency**>  <**groupId**>org.mybatis</**groupId**>  <**artifactId**>mybatis-spring</**artifactId**>  <**version**>${mybatis.spring.version}</**version**>  </**dependency**>     *<!-- mybatis -->* <**dependency**>  <**groupId**>org.mybatis</**groupId**>  <**artifactId**>mybatis</**artifactId**>  <**version**>${mybatis.version}</**version**>  </**dependency**>    *<!-- MySql -->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>${mysql.version}</**version**>  </**dependency**>   *<!-- 连接池 -->* <**dependency**>  <**groupId**>com.jolbox</**groupId**>  <**artifactId**>bonecp-spring</**artifactId**>  <**version**>${jolbox.version}</**version**>  </**dependency**>   *<!--日志包-->* <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-api</**artifactId**>  <**version**>${slf4j.version}</**version**>  <**optional**>true</**optional**>  </**dependency**>  <**dependency**>  <**groupId**>org.slf4j</**groupId**>  <**artifactId**>slf4j-log4j12</**artifactId**>  <**version**>${slf4j.log4j12.version}</**version**>  <**optional**>true</**optional**>  </**dependency**>  <**dependency**>  <**groupId**>log4j</**groupId**>  <**artifactId**>log4j</**artifactId**>  <**version**>${log4j.version}</**version**>  <**optional**>true</**optional**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.logging.log4j</**groupId**>  <**artifactId**>log4j-core</**artifactId**>  <**version**>${log4j.core.version}</**version**>  <**optional**>true</**optional**>  </**dependency**>  <**dependency**>  <**groupId**>commons-logging</**groupId**>  <**artifactId**>commons-logging</**artifactId**>  <**version**>${commons.logging.version}</**version**>  <**optional**>true</**optional**>  </**dependency**>   *<!-- Test dependencies -->* <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>${junit.version}</**version**>  <**scope**>test</**scope**>  </**dependency**> </**dependencies**>  <**build**>  <**defaultGoal**>install</**defaultGoal**>   <**plugins**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>3.6.1</**version**>  <**configuration**>  <**source**>1.8</**source**>  <**target**>1.8</**target**>  </**configuration**>  </**plugin**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-resources-plugin</**artifactId**>  <**version**>3.0.2</**version**>  <**configuration**>  <**encoding**>UTF-8</**encoding**>  </**configuration**>  </**plugin**>  </**plugins**>    *<!--IDEA是不会编译src的java目录的xml文件，需要手动指定哪些配置文件需要读取-->* <**resources**>  <**resource**>  <**directory**>src/main/java</**directory**>  <**includes**>  <**include**>\*\*/\*.xml</**include**>  </**includes**>  </**resource**>  </**resources**> </**build**> |

## 导入log4j.properties

|  |
| --- |
| **log4j.rootLogger** = **debug,stdout,D,E  log4j.appender.stdout** = **org.apache.log4j.ConsoleAppender log4j.appender.stdout.Target** = **System.out log4j.appender.stdout.layout** = **org.apache.log4j.PatternLayout log4j.appender.stdout.layout.ConversionPattern** = **[%-5p] %d{yyyy-MM-dd HH:mm:ss,SSS} method:%l%n%m%n  log4j.appender.D** = **org.apache.log4j.DailyRollingFileAppender log4j.appender.D.File** = **debug.log log4j.appender.D.Append** = **true log4j.appender.D.Threshold** = **DEBUG  log4j.appender.D.layout** = **org.apache.log4j.PatternLayout log4j.appender.D.layout.ConversionPattern** = **%-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n  log4j.appender.E** = **org.apache.log4j.DailyRollingFileAppender log4j.appender.E.File** = **error.log log4j.appender.E.Append** = **true log4j.appender.E.Threshold** = **ERROR  log4j.appender.E.layout** = **org.apache.log4j.PatternLayout log4j.appender.E.layout.ConversionPattern** = **%-d{yyyy-MM-dd HH:mm:ss} [ %t:%r ] - [ %p ] %m%n** |

## 导入jdbc.properties

|  |
| --- |
| **jdbc.driver**=**com.mysql.jdbc.Driver jdbc.url**=**jdbc:mysql://localhost:3306/demo?characterEncoding=utf-8 jdbc.username**=**root jdbc.pwd**=**123456** |

## 创建MyBatis核心配置文件

|  |
| --- |
| *<?***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**></**configuration**> |

## 创建Spring配置文件

|  |
| --- |
| *<?***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" xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:aop="http://www.springframework.org/schema/aop"  xsi:schemaLocation="  http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/context  http://www.springframework.org/schema/context/spring-context.xsd  http://www.springframework.org/schema/mvc  http://www.springframework.org/schema/mvc/spring-mvc.xsd  http://www.springframework.org/schema/tx  http://www.springframework.org/schema/tx/spring-tx.xsd  http://www.springframework.org/schema/aop  http://www.springframework.org/schema/aop/spring-aop.xsd"**>   *<!-- Service层扫描 -->* <**context:component-scan base-package="cn.itcast.springmybatis.service.impl"** />   *<!-- 解析jdbc配置文件 -->* <**context:property-placeholder location="classpath:jdbc.properties"** />   *<!-- the transactional advice (what 'happens'; see the <aop:advisor/> bean  below) 事务传播特性配置 -->* <**tx:advice id="txAdvice" transaction-manager="txManager"**>  *<!-- the transactional semantics... -->* <**tx:attributes**>  <**tx:method name="add\*" propagation="REQUIRED" isolation="DEFAULT"  rollback-for="java.lang.Exception"** />  <**tx:method name="save\*" propagation="REQUIRED" isolation="DEFAULT"  rollback-for="java.lang.Exception"** />  <**tx:method name="insert\*" propagation="REQUIRED" isolation="DEFAULT"  rollback-for="java.lang.Exception"** />  <**tx:method name="update\*" propagation="REQUIRED" isolation="DEFAULT"  rollback-for="java.lang.Exception"** />  <**tx:method name="modify\*" propagation="REQUIRED" isolation="DEFAULT"  rollback-for="java.lang.Exception"** />  <**tx:method name="delete\*" propagation="REQUIRED" isolation="DEFAULT"  rollback-for="java.lang.Exception"** />   *<!-- 查询方法 -->* <**tx:method name="query\*" read-only="true"** />  <**tx:method name="select\*" read-only="true"** />  <**tx:method name="find\*" read-only="true"** />  </**tx:attributes**>  </**tx:advice**>    *<!-- 配置事务管理器 -->* <**bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  <**property name="dataSource" ref="dataSource"** />  </**bean**>   *<!-- 配置数据源 -->* <**bean id="dataSource" class="com.jolbox.bonecp.BoneCPDataSource" destroy-method="close"**>  <**property name="driverClass" value="${jdbc.driver}"** />  <**property name="jdbcUrl" value="${jdbc.url}"** />  <**property name="username" value="${jdbc.username}"** />  <**property name="password" value="${jdbc.pwd}"** />  </**bean**>    *<!-- 声明式事务AOP配置 -->* <**aop:config**>  <**aop:pointcut expression="execution(\* cn.itcast.springmybatis.service.impl.\*.\*(..))" id="tranpointcut"** />  <**aop:advisor advice-ref="txAdvice" pointcut-ref="tranpointcut"** />  </**aop:config**>   *<!-- SqlSessionFactoryBean -->* <**bean id="sqlSessionFactoryBean" class="org.mybatis.spring.SqlSessionFactoryBean"**>  <**property name="configLocation" value="classpath:mybatis.xml"** />  <**property name="typeAliasesPackage" value="cn.itcast.springmybatis.model"** />  <**property name="mapperLocations"**>  <**list**>  <**value**>classpath:cn/itcast/springmybatis/mapper/\*Mapper.xml</**value**>  </**list**>  </**property**>  <**property name="dataSource" ref="dataSource"** />  </**bean**>   *<!-- 配置接口扫描包 -->* <**bean class="org.mybatis.spring.mapper.MapperScannerConfigurer"  p:basePackage="cn.itcast.springmybatis.mapper"  p:sqlSessionFactoryBeanName="sqlSessionFactoryBean"** />  </**beans**> |

## 编写测试案例

### 创建User实体Bean

在cn.itcast.springmybatis.model包下创建User

|  |
| --- |
| **package** cn.itcast.springmybatis.model; **import** java.io.Serializable; **public class** User **implements** Serializable{  */\*\*\*  \* 用户ID  \*/* **private** Integer **userid**;   */\*\*\*  \* 用户名字  \*/* **private** String **username**;  */\*\*\*  \* 用户年龄  \*/* **private** Integer **userage**;  **public** Integer getUserid() {  **return userid**;  }  **public void** setUserid(Integer userid) {  **this**.**userid** = userid;  }  **public** String getUsername() {  **return username**;  }  **public void** setUsername(String username) {  **this**.**username** = username;  }  **public** Integer getUserage() {  **return userage**;  }  **public void** setUserage(Integer userage) {  **this**.**userage** = userage;  }   @Override  **public** String toString() {  **return "User{"** +  **"userid="** + **userid** +  **", username='"** + **username** + **'\''** +  **", userage="** + **userage** +  **'}'**;  } } |

### 创建UserMapper.xml

在cn.itcast.springmybatis.mapper包下创建UserMapper.xml

|  |
| --- |
| *<?***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.springmybatis.mapper.UserMapper"**>  *<!-- addUser -->* <**insert id="addUser" parameterType="User"**>  INSERT INTO user(userid,user\_name,userage) VALUES(#{userid},#{username},#{userage})  </**insert**>  </**mapper**> |

### 创建UserMapper接口

在cn.itcast.springmybatis.mapper包下创建UserMapper接口

|  |
| --- |
| **package** cn.itcast.springmybatis.mapper; **import** cn.itcast.springmybatis.model.User; **public interface** UserMapper {   */\*\*  \* 增加用户  \** ***@param user*** *\** ***@return*** *\*/* **int** addUser(User user); } |

### 创建UserService接口

在包cn.itcast.springmybatis.service下创建UserService接口

|  |
| --- |
| **package** cn.itcast.springmybatis.service; **import** cn.itcast.springmybatis.model.User; **public interface** UserService {   */\*\*  \* 增加用户  \** ***@param user*** *\** ***@return*** *\*/* **int** addUser(User user); } |

### 创建UserServiceImpl

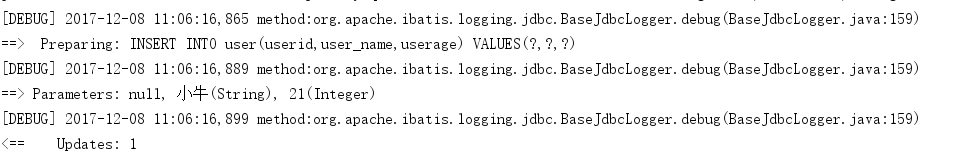
在包cn.itcast.springmybatis.service.impl下创建UserServiceImpl类实现UserService接口

|  |
| --- |
| **package** cn.itcast.springmybatis.service.impl; **import** cn.itcast.springmybatis.mapper.UserMapper; **import** cn.itcast.springmybatis.model.User; **import** cn.itcast.springmybatis.service.UserService; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.stereotype.Service; @Service **public class** UserServiceImpl **implements** UserService {   @Autowired  **private** UserMapper **userMapper**;   @Override  **public int** addUser(User user) {  **return userMapper**.addUser(user);  } } |

### 编写测试

|  |
| --- |
| **package** cn.itcast; **import** cn.itcast.springmybatis.model.User; **import** cn.itcast.springmybatis.service.UserService; **import** org.junit.Before; **import** org.junit.Test; **import** org.springframework.context.ApplicationContext; **import** org.springframework.context.support.ClassPathXmlApplicationContext; **public class** MyBatisTest {  *//UserService接口* **private** UserService **userService**;  **private** ApplicationContext **act**;   @Before  **public void** init() {  **act** = **new** ClassPathXmlApplicationContext(**"classpath:spring.xml"**);  **userService** = **act**.getBean(UserService.**class**);  }   */\*\*  \* 入参为JavaBean  \* 根据用户名模糊查询用户信息  \** ***@return*** *\*/* @Test  **public void** testAddUser(){  User user = **new** User();  user.setUsername(**"小牛"**);  user.setUserage(21);  **userService**.addUser(user);  } } |

测试结果

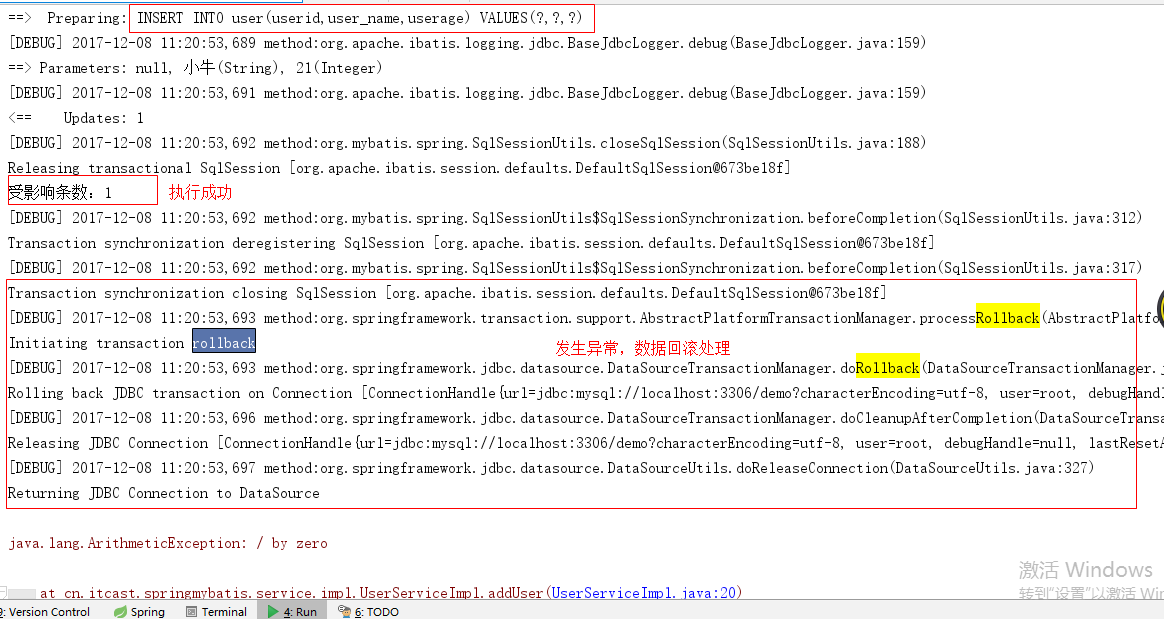


## 事务测试

修改UserServiceImpl中的addUser方法

|  |
| --- |
| @Override **public int** addUser(User user) {  **int** acount = **userMapper**.addUser(user);  System.***out***.println(**"受影响条数："**+acount);   *//制造算术异常，看数据是否回滚* **int** q = 0;  **int** i = 10/q;  **return** acount; } |

测试结果



数据库中没有小牛，事务成功

