**Mybatis第二天**

上海千锋 文威

1、回顾mybatis环境搭建和自定义环境搭建

2、Mybatis基于代理Dao的crud操作**(重点)**

3、crud中可能遇到的问题:参数的传递和返回值结果的封装

4、介绍Mybatis基于传统dao方式的使用(了解)

5、Mybatis主配置文件中的常用配置**(重点)**

Properties标签

TypeAliases标签

Mapper标签的子标签：package

## 1.1 Mybatis基于代理dao的crud操作

### 1.1.1 搭建mybatis工程环境

* 引入依赖

|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.mybatis</**groupId**>  <**artifactId**>mybatis</**artifactId**>  <**version**>3.4.5</**version**>  </**dependency**>  <**dependency**>  <**groupId**>junit</**groupId**>  <**artifactId**>junit</**artifactId**>  <**version**>4.10</**version**>  <**scope**>test</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.6</**version**>  <**scope**>runtime</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>log4j</**groupId**>  <**artifactId**>log4j</**artifactId**>  <**version**>1.2.12</**version**>  </**dependency**> </**dependencies**> |

* 定义核心配置文件

|  |
| --- |
| *<?***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**>  <**environments default="mybatis"**>  <**environment id="mybatis"**>  <**transactionManager type="jdbc"**></**transactionManager**>  <**dataSource type="pooled"**>  <**property name="driver" value="com.mysql.jdbc.Driver"**></**property**>  <**property name="url" value="jdbc:mysql://192.168.10.137:3306/mybatis"**></**property**>  <**property name="username" value="root"**></**property**>  <**property name="password" value="Admin123!"**></**property**>  </**dataSource**>  </**environment**>  </**environments**>  <**mappers**>  <**mapper resource="com/qf/dao/UserDao.xml"**></**mapper**>  </**mappers**> </**configuration**> |

* 定义实体类

|  |
| --- |
| **public class** User {  **private** Integer **id**;  **private** String **username**;  **private** Date **birthday**;  **private** String **sex**;  **private** String **address**;  // get set方法省略 } |

* 定义接口

|  |
| --- |
| **public interface** UserDao {  **public** List<User> findAll(); } |

* 定义mapper映射文件

|  |
| --- |
| *<?***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="com.qf.dao.UserDao"**>   <**select id="findAll" resultType="com.qf.pojo.User"**>  select *\** from user  </**select**> </**mapper**> |

* 测试

|  |
| --- |
| **public class** TestUser {  @Test  **public void** test() **throws** Exception{  InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = builder.build(in);  SqlSession sqlSession = sqlSessionFactory.openSession();  UserDao userDao = sqlSession.getMapper(UserDao.**class**);  List<User> userList = userDao.findAll();  **for**(User user : userList){  System.***out***.println(user);  }  sqlSession.close();  } } |

### 1.1.2 根据id查询

* 在接口中添加根据id查询的方法

|  |
| --- |
| *//根据id查询User* **public** User findUserById(); |

* 在映射文件中添加配置

|  |
| --- |
| *<!--  根据id查询用户信息  parameterType:指定输入参数的类型  resultType：指定数据结果封装的数据类型 -->* <**select id="findUserById" parameterType="int" resultType="com.qf.pojo.User"** >  select *\** from user where id=#{id} </**select**> |

注意：

**sql 语句中使用#{}字符：**

**它代表占位符，相当于原来 jdbc 部分所学的?，都是用于执行语句时替换实际的数据。**

**具体的数据是由#{}里面的内容决定的。**

**#{}中内容的写法：**

**由于数据类型是基本类型，所以此处可以随意写。**

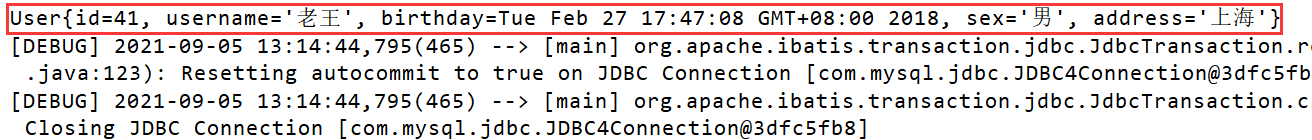
* 在测试类中添加测试

|  |
| --- |
| @Test **public void** testFindUserById() **throws** Exception{  InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = builder.build(in);  SqlSession sqlSession = sqlSessionFactory.openSession();  UserDao userDao = sqlSession.getMapper(UserDao.**class**);  User user = userDao.findUserById(41);  System.***out***.println(user);  sqlSession.close(); } |

我们测试结果没有问题，但是这种写法比较麻烦。每次测试的时候，都需要写大量的重复代码。我们可以不可以把这些重复的代码抽取出来呢?答案是可以的。

|  |
| --- |
| **public class** TestUser {  SqlSession **sqlSession**;  UserDao **userDao**;   @Before *//在@Test注解修饰的方法之前执行* **public void** init() **throws** Exception{  InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = builder.build(in);  **sqlSession** = sqlSessionFactory.openSession();  **userDao** = **sqlSession**.getMapper(UserDao.**class**);  }   @After *//在@Test注解修饰的方法之后执行* **public void** after(){  **sqlSession**.close();  }   @Test  **public void** testFindUserById() **throws** Exception{  User user = userDao.findUserById(41);  System.***out***.println(user);  } } |

改造之后测试没有问题



### 1.1.3 保存的操作

* 在接口中添加保存的方法

|  |
| --- |
| *//保存用户信息* **public void** saveUser(User user); |

* 在映射文件里面添加保存的配置信息

|  |
| --- |
| *<!--  保存用户信息  #{} 占位符 -->* <**insert id="saveUser" parameterType="com.qf.pojo.User"**>  insert into user(username,birthday,sex,address) values(#{username},#{birthday},#{sex},#{address}); </**insert**> |

细节：

parameterType 属性：代表参数的类型，因为我们要传入的是一个类的对象，所以类型就写类的全名称。

sql 语句中使用#{}字符：

它代表占位符，相当于原来 jdbc 部分所学的?，都是用于执行语句时替换实际的数据。具体的数据是由#{}里面的内容决定的。

#{}中内容的写法： 由于我们保存方法的参数是一个 User 对象，此处要写 User 对象中的属性名称。

* 测试

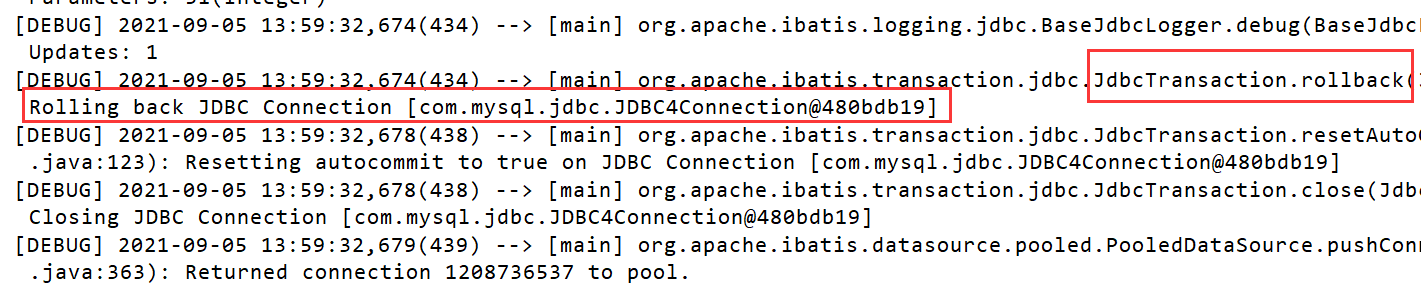
|  |
| --- |
| @Test **public void** testSaveUser(){  User user = **new** User();  user.setUsername(**"张宇"**);  user.setBirthday(**new** Date());  user.setSex(**"男"**);  user.setAddress(**"CHN"**);  **userDao**.saveUser(user);} |

此时执行结果，发现数据库里面并没有新增记录。这是为什么?

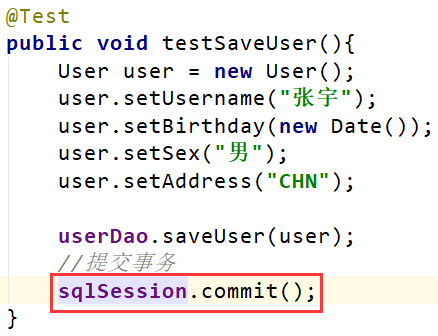
我们查看IDEA控制台日志信息:

发现竟然执行回滚的操作了！！！

如图：



这一点和 jdbc 是一样的，我们在实现增删改时一定要去控制事务的提交，那么在 mybatis 中如何控制事务 提交呢？ 可以使用:session.commit();来实现事务提交。



再次测试，发现数据新增成功。

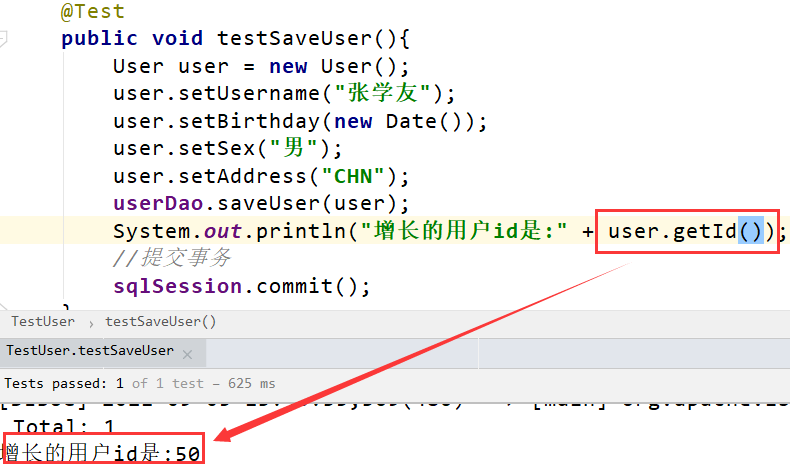
**问题扩展：新增用户 id 的返回值问题。**

新增用户后，同时还要返回当前新增用户的 id 值，因为 id 是由数据库的自动增长来实现的，所以就相当于我们要在新增后将自动增长 auto\_increment 的值返回。

**第一种方式:**

|  |
| --- |
| <**insert id="saveUser" parameterType="com.qf.pojo.User"**>  /\*  配置保存之后，获取新增记录的id值  keyColumn="id" 数据库主键   keyProperty="id" 主键对应实体类的属性  resultType="int" 主键的数据类型  \*/  <**selectKey keyColumn="id" keyProperty="id" resultType="int"**>  select last\_insert\_id();  </**selectKey**>  insert into user(username,birthday,sex,address) values(#{username},#{birthday},#{sex},#{address}); </**insert**> |

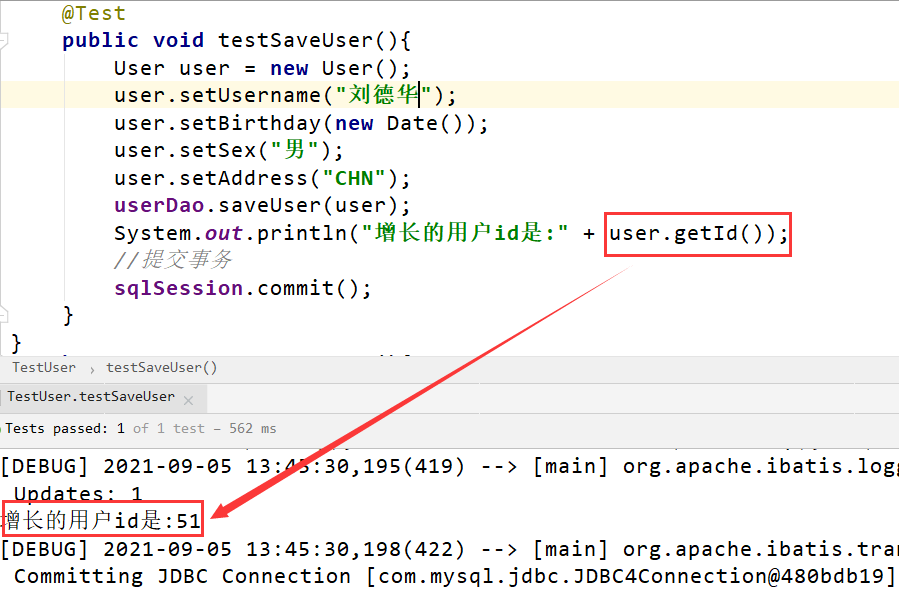
测试：



**第二种方式：**

|  |
| --- |
| *<!--  保存用户信息  #{} 占位符 -->* <**insert id="saveUser" parameterType="com.qf.pojo.User" useGeneratedKeys="true" keyProperty="id" keyColumn="id"**>  insert into user(username,birthday,sex,address) values(#{username},#{birthday},#{sex},#{address}); </**insert**> |

**测试：**

****

### 1.1.4 修改的操作

* 在持久层接口中添加更新方法

|  |
| --- |
| *//修改用户* **public void** updateUser(User user); |

* 在用户的映射配置文件中配置

|  |
| --- |
| *<!--  修改用户信息 -->* <**update id="updateUser" parameterType="com.qf.pojo.User"**>  update user set username=#{username},birthday=#{birthday},sex=#{sex},address=#{address} where id=#{id} </**update**> |

* 测试

|  |
| --- |
| @Test **public void** testUpdate(){  User user = **new** User();  user.setUsername(**"马德华"**);  user.setBirthday(**new** Date());  user.setSex(**"男"**);  user.setAddress(**"USA"**);  user.setId(51);  **userDao**.updateUser(user);  **sqlSession**.commit(); } |

### 1.1.5 删除的操作

* 在接口中添加删除的方法

|  |
| --- |
| *//删除用户* **public void** deleteUserById(Integer id) |

* 在映射文件添加删除的操作

|  |
| --- |
| <**delete id="deleteUserById" parameterType="int"**>  delete from user where id=#{id} </**delete**> |

* 测试

|  |
| --- |
| @Test **public void** testDeleteUserById(){  **userDao**.deleteUserById(51);  **sqlSession**.commit(); } |

### 1.1.6 模糊查询

* 在接口中添加模糊查询的方法

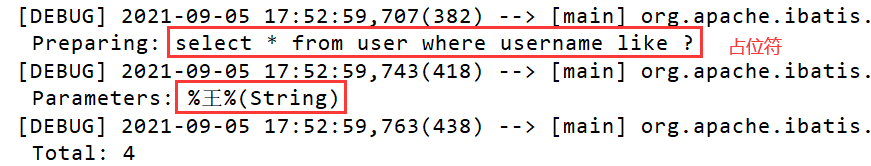
|  |
| --- |
| *//模糊查询* **public** List<User> findUserByName(String name); |

* 在映射文件中添加模糊查询的配置

|  |
| --- |
| *<!--模糊查询-->* <**select id="findUserByName" parameterType="string" resultType="com.qf.pojo.User"**>  select *\** from user where username like #{name} </**select**> |

* 测试模糊查询

|  |
| --- |
| @Test **public void** testFindUserByName(){  List<User> list = **userDao**.findUserByName(**"%王%"**);  **for**(User user : list){  System.***out***.println(user);  } } |



我们在配置文件中没有加入%来作为模糊查询的条件，所以在传入字符串实参时，就需要给定模糊查询的标识%。配置文件中的#{username}也只是一个占位符，所以 SQL 语句显示为“？”。

模糊查询的另一种配置方式：

* 定义一个新的模糊查询接口

|  |
| --- |
| *//模糊查询2* **public** List<User> findUserByName2(String name); |

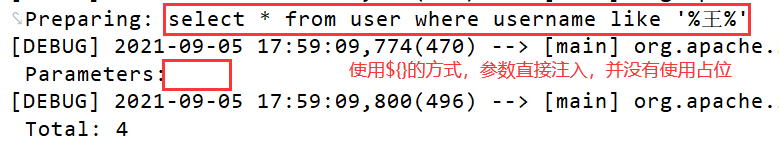
* 添加对应接口的映射配置文件信息

|  |
| --- |
| *<!--模糊查询2-->* <**select id="findUserByName2" parameterType="string" resultType="com.qf.pojo.User"**>  select *\** from user where username like '%${value}%' </**select**> |

* 测试

|  |
| --- |
| @Test **public void** testFindUserByName2(){  List<User> list = **userDao**.findUserByName2(**"王"**);  **for**(User user : list){  System.***out***.println(user);  } } |

我们查看控制台日志信息



可以发现，我们在程序代码中就不需要加入模糊查询的匹配符%了，这两种方式的实现效果是一样的，但执行的语句是不一样的。

**总结:#{} 和${}的区别**

#{}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。**

### 1.1.7 聚合函数查询

* 添加查询接口

|  |
| --- |
| *//聚合函数查询* **public** Integer getTotal(); |

* 添加聚合函数查询的mapper映射配置

|  |
| --- |
| *<!--聚合函数查询-->* <**select id="getTotal" resultType="int"**>  select *count*(id) from user; </**select**> |

* 测试

|  |
| --- |
| @Test **public void** testGetTotal(){  Integer total = **userDao**.getTotal();  System.***out***.println(total); } |

### 1.1.8 在接口中传入多个参数的问题

如果我们在接口中传入多个参数，对参数的位置不加以指定的话，那么执行结果一定会报错。所以在传递多个参数的时候，一定要指明参数对应的位置。

**第一种方式:**

* 定义接口

|  |
| --- |
| *//传递多个参数的问题 根据性别和地址一起查询* **public** List<User> findUserBySexAndAddress(String sex, String address); |

* 增加接口的配置文件

|  |
| --- |
| <**select id="findUserBySexAndAddress" resultType="com.qf.pojo.User"**>  select *\** from user where sex='**${param1}**' and address='**${param2}**' </**select**> |

注意:

在sql语句中，使用${param1} 或者${param2} 指定参数传递的位置 也可以使用${arg0} ${arg1}。

**第二种方式：**

* 定义接口

|  |
| --- |
| *//传递多个参数的问题 根据性别和地址一起查询* **public** List<User> findUserBySexAndAddress(@Param(**"sex"**) String sex,@Param("address") String address); |

* 增加接口的配置文件

|  |
| --- |
| <**select id="findUserBySexAndAddress" resultType="com.qf.pojo.User"**>  select *\** from user where **sex='${sex}' and address='${address}'** </**select**> |

* 测试

|  |
| --- |
| @Test **public void** testFindUserBySexAndAddress(){  List<User> userList = **userDao**.findUserBySexAndAddress(**"男"**,**"上海"**);  **for**(User user : userList){  System.***out***.println(user);  } } |

## 1.2 Mybatis的参数深入

### 1.2.1 输入参数--parameterType

我们在上一章节中已经介绍了 SQL 语句传参，使用标签的 parameterType 属性来设定。该属性的取值可以 是基本类型，引用类型（例如:String 类型），还可以是实体类类型（POJO 类）。**同时也可以使用实体类的包装类**，本章节将介绍如何使用实体类的包装类作为参数传递。

但是需要注意的是：

基本类型和 String 我们可以直接写类型名称，也可以使用包名。类名的方式，例如：java.lang.String。

比如我们的案例中，也可以这么写

|  |
| --- |
| *<!--模糊查询1-->* <**select id="findUserByName" parameterType="java.lang.String" resultType="com.qf.pojo.User"**>  select *\** from user where username like #{username} </**select**> |

但是我们是直接写的类名string

**实体类类型，目前我们只能使用全限定类名(接下来我们就可以讲解到)**。

究其原因，**是mybaits 在加载时已经把常用的数据类型注册了别名**，从而我们在使用时可以不写包名，而我们的是实体类并没有注册别名，所以必须写全限定类名。在今天课程的最后一个章节中将讲解如何注册实体类的别名。

我们查看mybatis源码可以看到：



### 1.2.2 传递pojo对象

需求：根据id的集合批量查询用户信息

* 编写接口

|  |
| --- |
| *//根据id的集合查询用户列表* **public** List<User> findUserByIds(QueryVo queryVo); |

* 编写实体类

|  |
| --- |
| **public class** QueryVo {   *//保存id的集合* List<Integer> **ids**;   **public** List<Integer> getIds() {  **return ids**;  }   **public void** setIds(List<Integer> ids) {  **this**.**ids** = ids;  } } |

* 编写映射文件的接口配置

|  |
| --- |
| *<!--  根据用户id的集合进行查询  foreach 遍历集合的标签  collection:list集合的名称  item:遍历的集合元素的别名  separator：select \* from user where id in(1,2,3); in后面括号里面参数用逗号分隔开来。  open:括号的开始部分 (  close:括号的结束部分 ) -->* <**select id="findUserByIds" parameterType="com.qf.pojo.QueryVo" resultType="com.qf.pojo.User"**>  select *\** from user where id in  <**foreach collection="ids" item="id" separator="," open="(" close=")"**>  #{id}  </**foreach**> </**select**> |

* 编写测试类

|  |
| --- |
| @Test **public void** testFindUserByIds(){  QueryVo queryVo = **new** QueryVo();  List<Integer> list = **new** ArrayList<Integer>();  list.add(41);  list.add(42);  list.add(43);  queryVo.setIds(list);  List<User> userList = **userDao**.findUserByIds(queryVo);  **for**(User user : userList){  System.***out***.println(user);  } } |

### 1.2.3 输出参数--ResultType和ResultMap

resultType 属性可以指定结果集的类型，它支持基本类型和实体类类型。

我们在前面的 CRUD 案例中已经对此属性进行过应用了。

需要注意的是，它和 parameterType 一样，如果注册过类型别名的，可以直接使用别名。没有注册过的必须 使用全限定类名。例如：我们的实体类此时必须是全限定类名（今天最后一个章节会讲解如何配置实体类的别名）。

同时，当是实体类名称是，还有一个要求，实体类中的属性名称必须和查询语句中的列名保持一致，否则无法 实现封装。

基本的实例我们就不演示了，因为在前面的章节我们已经演示过。

现在我们演示两种比较特殊的情况：

#### 1.2.3.1 数据表字段和实体字段不一致(1)

* 修改实体类

这里实体类字段我们都使用驼峰命名规则来定义。

|  |
| --- |
| **public class** User {  **private** Integer **userId**;  **private** String **userName**;  **private** Date **userBirthday**;  **private** String **userSex**;  **private** String **userAddress**;   **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** Date getUserBirthday() {  **return userBirthday**;  }   **public void** setUserBirthday(Date userBirthday) {  **this**.**userBirthday** = userBirthday;  }   **public** String getUserSex() {  **return userSex**;  }   **public void** setUserSex(String userSex) {  **this**.**userSex** = userSex;  }   **public** String getUserAddress() {  **return userAddress**;  }   **public void** setUserAddress(String userAddress) {  **this**.**userAddress** = userAddress;  }  @Override **public** String toString() {  **return "User{"** +  **"userId="** + **userId** +  **", userName='"** + **userName** + **'\''** +  **", userBirthday="** + **userBirthday** +  **", userSex='"** + **userSex** + **'\''** +  **", userAddress='"** + **userAddress** + **'\''** +  **'}'**; }  } |

* 查询接口

|  |
| --- |
| *//查询所有User信息* **public** List<User> findAll(); |

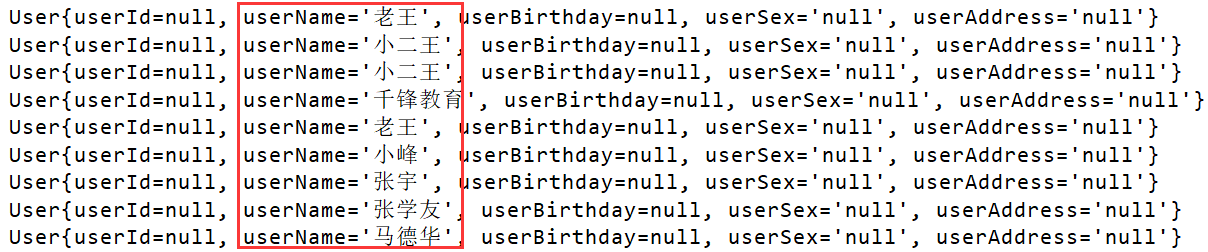
* 配置文件

|  |
| --- |
| <**select id="findAll" resultType="com.qf.pojo.User"**>  select *\** from user </**select**> |

* 测试

|  |
| --- |
| @Test **public void** testFindAll() **throws** Exception{  List<User> userList = **userDao**.findAll();  **for**(User user : userList){  System.***out***.println(user);  } } |

观察测试结果：



思考：

为什么userName会有值，其他字段没有值呢?

解答：

**因为在mysql和windows中，忽略了大小写**。只要数据表字段和实体类字段名字一致就可以。但是其他字段和数据表字段名字却不一样。所以没法映射。

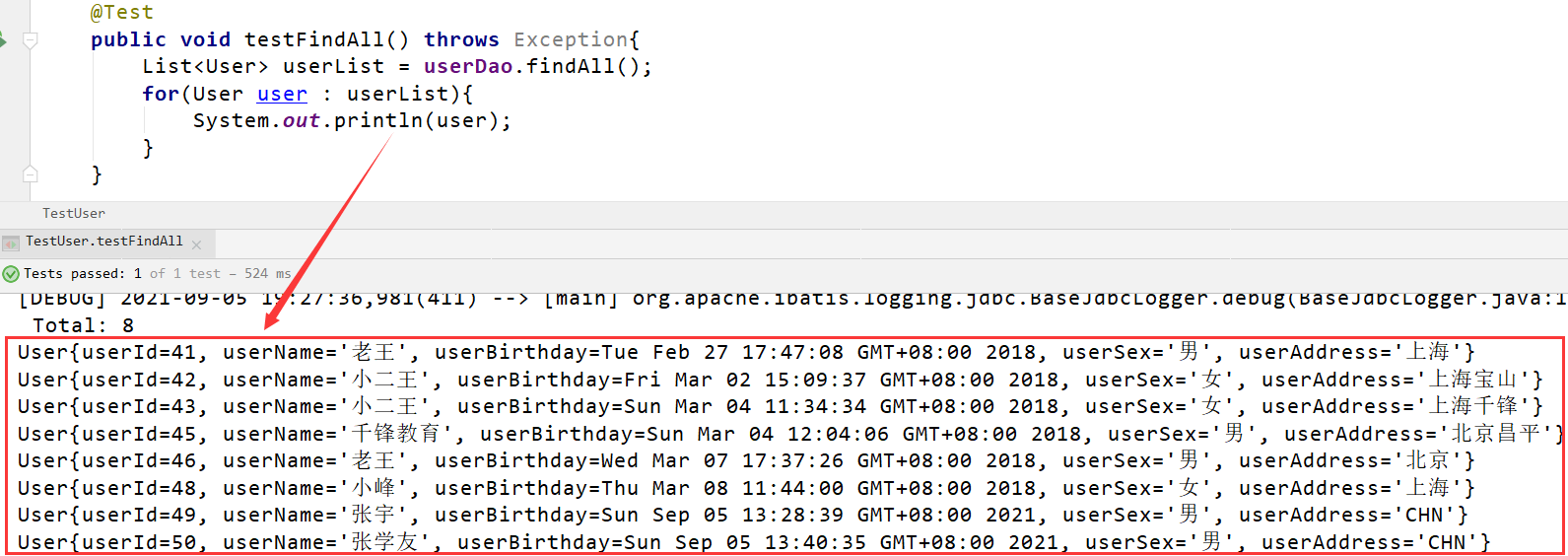
解决方案：

**使用别名查询**

* 修改映射配置

|  |
| --- |
| <**select id="findAll" resultType="com.qf.pojo.User"**>  select id as userId,username as userName,birthday as userBirthday,sex as userSex,address as userAddress from user </**select**> |

* 测试查看效果



**思考： 如果我们的查询很多，都使用别名的话写起来岂不是很麻烦，有没有别的解决办法呢？**

使用ResultMap!

#### 1.2.3.2 数据表字段和实体字段不一致(2)

* 实体(省略)，和上面一样。
* 配置文件

|  |
| --- |
| *<!--使用ResultMap标签手动映射，解决实体字段和数据表字段不一致的问题-->* <**resultMap id="userMap" type="com.qf.pojo.User"**>  <**id property="userId" column="id"**></**id**>  <**result property="userName" column="username"**></**result**>  <**result property="userBirthday" column="birthday"**></**result**>  <**result property="userSex" column="sex"**></**result**>  <**result property="userAddress" column="address"**></**result**> </**resultMap**> <**select id="findAll" resultMap="userMap"**>  select *\** from user </**select**> |

解读：

resultMap标签：

id:自定义，select查询标签的返回值属性引用。

type: 映射实体的全限定名(如果配置别名，也可以写别名，我们后面讲到别名再使用。)

id标签：

描述主键字段的映射关系。

property：实体类的属性。

column：数据表字段名称。

result：

描述非主键字段的映射关系。

property：实体类的属性。

column：数据表字段名称。

select标签里面的resultMap属性:

描述查询结果的返回值类型，**这里不能再使用resultType属性了！！！**需要引用resultMap标签里面的id值。

最后总结：

**resultType和resultMap的区别?**

## 1.3 Mybatis 传统 DAO 层开发[了解]

使用Mybatis开发Dao，通常有两个方法，即原始Dao开发方式和Mapper接口代理开发方式。而现在主流的开发方式是接口代理开发方式，这种方式总体上更加简便。我们的课程讲解也主要以接口代理开发方式为主。在第二章节已经给大家介绍了基于代理方式的dao开发，现在给大家介绍一下基于传统编写Dao实现类的开发方式。

在这里我们给大家演示单表的增删该查操作即可。

### 1.3.1 创建实体类

实体类保持数据表字段和实体字段名称一致即可。这里省略

### 1.3.2 创建接口

|  |
| --- |
| **public interface** UserDao {  *//查询所有User信息* **public** List<User> findAll();   *//根据id查询User* **public** User findUserById(Integer id);   *//保存用户信息* **public void** saveUser(User user);   *//修改用户* **public void** updateUser(User user);   *//删除用户* **public void** deleteUserById(Integer id); } |

### 1.3.3 定义映射文件

|  |
| --- |
| *<?***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="com.qf.dao.UserDao"**>   *<!--查询所有用户信息-->* <**select id="findAll" resultType="com.qf.pojo.User"**>  select *\** from user  </**select**>   *<!--根据id查询用户信息-->* <**select id="findUserById" parameterType="int" resultType="com.qf.pojo.User"**>  select *\** from user where id=#{id}  </**select**>   *<!--保存用户信息-->* <**insert id="saveUser" parameterType="com.qf.pojo.User"**>  insert into user(username,birthday,sex,address) values(#{username},#{birthday},#{sex},#{address});  </**insert**>   *<!--  修改用户信息  -->* <**update id="updateUser" parameterType="com.qf.pojo.User"**>  update user set username=#{username},birthday=#{birthday},sex=#{sex},address=#{address} where id=#{id}  </**update**>   *<!--删除用户-->* <**delete id="deleteUserById" parameterType="int"**>  delete from user where id=#{id}  </**delete**> </**mapper**> |

### 1.3.4 定义接口实现类

|  |
| --- |
| **public class** UserDaoImpl **implements** UserDao {    **private** SqlSessionFactory **sqlSessionFactory**;   **public** UserDaoImpl(SqlSessionFactory sqlSessionFactory) {  **this**.**sqlSessionFactory** = sqlSessionFactory;  }   **public** List<User> findAll() {  SqlSession sqlSession = **sqlSessionFactory**.openSession();  List<User> userList = sqlSession.selectList(**"com.qf.dao.UserDao.findAll"**);  sqlSession.close();  **return** userList;  }   **public** User findUserById(Integer id) {  SqlSession sqlSession = **sqlSessionFactory**.openSession();  User user = sqlSession.selectOne(**"com.qf.dao.UserDao.findUserById"**,id);  sqlSession.close();  **return** user;  }   **public void** saveUser(User user) {  SqlSession sqlSession = **sqlSessionFactory**.openSession();  sqlSession.insert(**"com.qf.dao.UserDao.saveUser"**,user);  sqlSession.commit();  sqlSession.close();  }   **public void** updateUser(User user) {  SqlSession sqlSession = **sqlSessionFactory**.openSession();  sqlSession.update(**"com.qf.dao.UserDao.updateUser"**,user);  sqlSession.commit();  sqlSession.close();  }   **public void** deleteUserById(Integer id) {  SqlSession sqlSession = **sqlSessionFactory**.openSession();  sqlSession.delete(**"com.qf.dao.UserDao.deleteUserById"**,id);  sqlSession.commit();  sqlSession.close();  } |

### 1.3.5 测试

|  |
| --- |
| **public class** TestUser {  SqlSession **sqlSession**;  SqlSessionFactory **sqlSessionFactory**;   @Before *//在@Test注解修饰的方法之前执行* **public void** init() **throws** Exception{  InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  **sqlSessionFactory** = builder.build(in);  }   @Test  **public void** testFindAll() **throws** Exception{  UserDao userDao = **new** UserDaoImpl(**sqlSessionFactory**);  List<User> userList = userDao.findAll();  **for**(User user : userList){  System.***out***.println(user);  }  }   @Test  **public void** testFindUserById(){  UserDao userDao = **new** UserDaoImpl(**sqlSessionFactory**);  User user = userDao.findUserById(41);  System.***out***.println(user);  }   @Test  **public void** testSaveUser(){  UserDao userDao = **new** UserDaoImpl(**sqlSessionFactory**);  User user = **new** User();  user.setUsername(**"张宇"**);  user.setBirthday(**new** Date());  user.setSex(**"男"**);  user.setAddress(**"CHN"**);  userDao.saveUser(user);   }   @Test  **public void** testUpdateUser(){  UserDao userDao = **new** UserDaoImpl(**sqlSessionFactory**);  User user = **new** User();  user.setUsername(**"张小宇"**);  user.setBirthday(**new** Date());  user.setSex(**"男"**);  user.setAddress(**"CHN"**);  userDao.updateUser(user);  }   @Test  **public void** testDeleteUser(){  UserDao userDao = **new** UserDaoImpl(**sqlSessionFactory**);  userDao.deleteUserById(56);  } } |

## 1.4 SqlMapConfig.xml的一些细节

### 1.4.1 SqlMapConfig.xml 中配置的内容和顺序

-properties（属性）

--property

-settings（全局配置参数）

--setting

-typeAliases（类型别名）

--typeAliase

--package

-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）

--mapper

--package

### 1.4.2 properties标签

我们一般会把数据库配置信息定义在一个独立的配置文件里面，比如db.properties。那么我们如何在mybatis的核心配置文件里面加载外部的数据库配置信息呢?

* 定义db.properties配置文件

|  |
| --- |
| **jdbc.driver**=**com.mysql.jdbc.Driver jdbc.url**=**jdbc:mysql://192.168.10.137:3306/mybatis jdbc.username**=**root jdbc.password**=**Admin123!** |

* 在SqlMapConfig.xml引入数据库配置信息



### 1.4.3 typeAliases标签

在前面我们讲的 Mybatis 支持的默认别名，我们也可以采用自定义别名方式来开发。

配置别名的方式有两种:

* 第一种

第一步:修改核心配置文件:



第二步:修改映射文件，使用别名替换



* 第二种



映射文件的修改和第一种一样，这里不再演示。

### 1.4.4 Mappers标签

Mappers标签里面定义mapper标签。作用是用来在核心配置文件里面引入映射文件。

* <mapper resurce=””></mapper>

|  |
| --- |
| <**mappers**>  <**mapper resource="com/qf/dao/UserDao.xml"**></**mapper**> </**mappers**> |

* <mapper class=””></mapper>

使用 mapper 接口类路径。如果我们使用注解开发的时候，就需要使用这个路径

|  |
| --- |
| <**mappers**>  <**mapper class="com.qf.dao.UserDao"**></**mapper**> </**mappers**> |

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。

* <package name=””>

|  |
| --- |
| <**mappers**>  <**package name="com.qf.dao"**></**package**> </**mappers**> |

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。