**Mybatis框架课程第二天**

# Mybatis实现CRUD操作

## Mybatis实现CRUD操作

### 功能需求

基于已有数据表user，使用MyBatis实现以下功能：

* 根据用户id查询一个用户
* 根据用户名称模糊查询用户列表
* 添加一个用户
* 根据用户id修改用户名
* 根据用户id删除用户

### 工程准备

参照第一天Mybatis快速入门部分

### 实现根据用户id查询一个用户

使用的Sql：SELECT \* FROM USER WHERE ID=1

#### Pojo

|  |
| --- |
| **public class** User {  **private int** id;  **private** String username;// 用户名称  **private** String sex; // 性别  **private** Date birthday; // 生日  **private** String address; // 地址  get/set……  } |

#### 映射文件

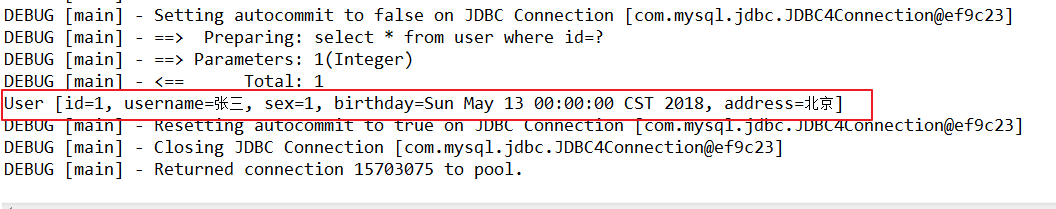
在resources目录下新建sqlmap子目录,创建UserMapper.xml,添加如下内容

|  |
| --- |
| <?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=*"test"*>  <select id=*"queryUserById"* parameterType=*"Integer"* resultType=*"com.itheima.mybatis.pojo.User">*  select \* from user where id=#{id}  </select>  </mapper> |

#### 测试程序

|  |
| --- |
| **public** **class** MybatisTest {  @Test  **public** **void** testQueryUserById() **throws** Exception { InputStream inputStream = Resources.*getResourceAsStream*("SqlMapConfig.xml"); SqlSessionFactoryBuilder sqlSessionFactoryBuilder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  SqlSession sqlSession = sqlSessionFactory.openSession();  User user = sqlSession.selectOne("test.queryUserById", 1);  System.***out***.println(user);  sqlSession.close();  }  } |

#### 效果



### 实现根据用户名模糊查询用户列表

使用的Sql：SELECT \* FROM USER WHERE USERNAME LIKE '%王%'

#### 方法一

##### （1）映射文件

在UserMapper.xml配置文件中添加如下内容：

|  |
| --- |
| <!-- 如果返回多个结果，mybatis会自动把返回的结果放在list容器中 -->  <!-- resultType的配置和返回一个结果的配置一样 -->  <select id=*"queryUserByUsername1"* parameterType=*"string"*  resultType=*"com.itheima.mybatis.pojo.User"*>  SELECT \* FROM USER WHERE USERNAME LIKE #{username}  </select> |

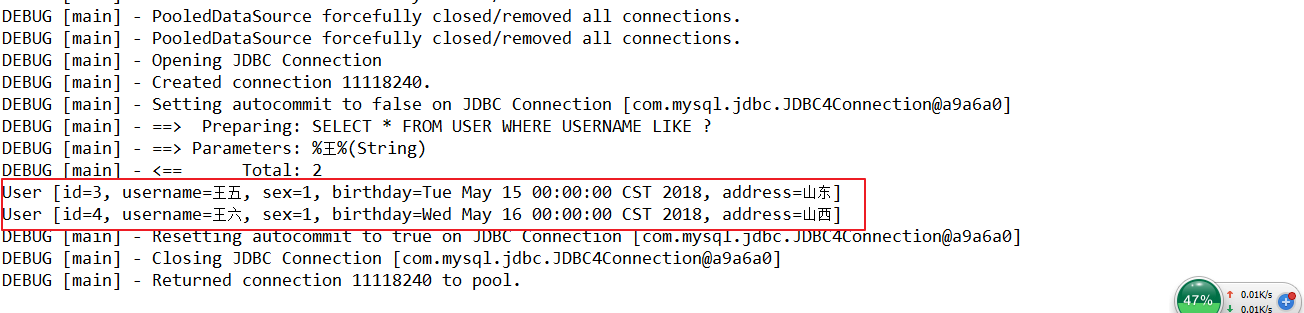
##### （2）测试程序

MybatisTest中添加测试方法如下：

|  |
| --- |
| @Test  **public** **void** testQueryUserByUsername1() **throws** Exception {  // 1. 加载SqlMapConfig.xml配置文件  InputStream inputStream = Resources.*getResourceAsStream*("SqlMapConfig.xml");  // 2. 创建SqlSessionFactoryBuilder对象  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = **new** SqlSessionFactoryBuilder();  // 3. 创建SqlSessionFactory对象  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  // 4. 创建SqlSession对象  SqlSession sqlSession = sqlSessionFactory.openSession();  // 5. 执行SqlSession对象执行查询，获取结果User  // 查询多条数据使用selectList方法  List<User> list = sqlSession.selectList("test.queryUserByUsername1", "%王%");  // 6. 打印结果  **for** (User user : list) {  System.***out***.println(user);  }  // 7. 释放资源  sqlSession.close();  } |

##### （3）效果

测试效果如下图：

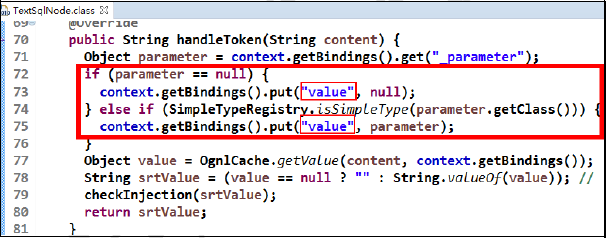


#### 方法二

##### （1）映射文件

在UserMapper.xml配置文件中添加如下内容：

|  |
| --- |
| <!-- 如果传入的参数是简单数据类型，${}里面必须写value -->  <select id=*"queryUserByUsername2"* parameterType=*"string"*  resultType=*"com.itheima.mybatis.pojo.User"*>  SELECT \* FROM USER WHERE USERNAME LIKE '%${value}%'  </select> |

注意：如果传入的参数是简单数据类型，${}里面必须写value字符串，源码如下：

这就说明了源码中指定了读取的key的名字就是”value”，所以我们在绑定参数时就只能叫value的名字了。

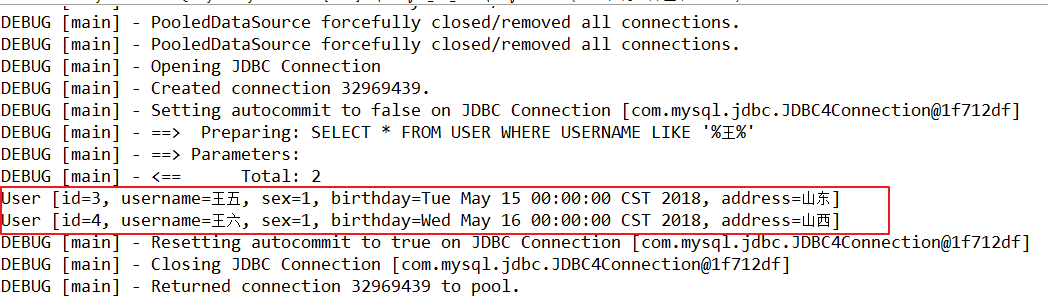
##### （2）测试程序

MybatisTest中添加测试方法如下：

|  |
| --- |
| @Test  **public** **void** testQueryUserByUsername2() **throws** Exception {  // 1. 加载SqlMapConfig.xml配置文件  InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");  // 2. 创建SqlSessionFactoryBuilder对象  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  // 3. 创建SqlSessionFactory对象  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  // 4. 创建SqlSession对象  SqlSession sqlSession = sqlSessionFactory.openSession();  // 5. 执行SqlSession对象执行查询，获取结果User  // 查询多条数据使用selectList方法  List<User> list = sqlSession.selectList("test.queryUserByUsername2", "王");  // 6. 打印结果  **for** (User user : list) {  System.***out***.println(user);  }  // 7. 释放资源  sqlSession.close();  } |

##### （3）效果

测试结果如下图：



### 小结

#### #{}和${}

* 关于#{}：

1、#{}等同于PreparedStatement中的占位符?，会自动对传入的字符串数据加一对单引号，可以避免Sql注入。

比如  select \* from user where username = #{username} ,传入的username为小张，那么最后打印出来的就是

 select \* from user where username = ‘小张’

2、#{}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，#{}括号中可以是任意名称。

#### 关于${}:

1、${}将传入的数据直接显示生成在Sql中，只是简单的拼接。如：order by ${id}，如果传入的值是id，则解析成的Sql为order by id。

如果上面的例子使用${}，则成了select \* from user where username = 小张

2、${}可以接收简单类型值或Pojo属性值，如果parameterType传输单个简单类型值，${}括号中只能是“value”这个字符串

* 总结：

1、$方式一般用于传入数据库对象，例如传入表名、order by 的字段

2、一般能用#的就别用$.

#### 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

#### 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。

### 实现添加用户

使用的Sql：

INSERT INTO USER (username,birthday,sex,address) VALUES

('黄忠','2016-07-26','1','三国')

#### 映射文件

在UserMapper.xml配置文件中添加如下内容：

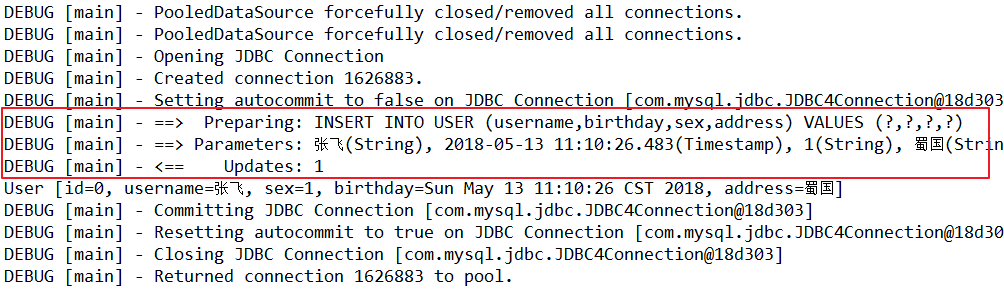
|  |
| --- |
| <!-- 保存用户 -->  <insert id=*"saveUser"* parameterType=*"com.itheima.mybatis.pojo.User"*>  INSERT INTO USER  (username,birthday,sex,address) VALUES  (#{username},#{birthday},#{sex},#{address})  </insert> |

#### 测试程序

MybatisTest中添加测试方法如下：

|  |
| --- |
| @Test  **public** **void** testSaveUser() **throws** Exception{  // 1. 加载SqlMapConfig.xml配置文件  InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");  // 2. 创建SqlSessionFactoryBuilder对象  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  // 3. 创建SqlSessionFactory对象  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  // 4. 创建SqlSession对象  SqlSession sqlSession = sqlSessionFactory.openSession();  // 5. 执行SqlSession对象执行保存  // 创建需要保存的User  User user = **new** User();  user.setUsername("张飞");  user.setSex("1");  user.setBirthday(**new** Date());  user.setAddress("蜀国");  sqlSession.insert("test.saveUser", user);  System.***out***.println(user);  // 需要进行事务提交  sqlSession.commit();  // 7. 释放资源  sqlSession.close();  } |

#### 效果



如上所示，保存成功，但是id=0，需要解决id返回不正常的问题。

#### Mysql自增主键返回

查询Mysql自增id的Sql：SELECT LAST\_INSERT\_ID()

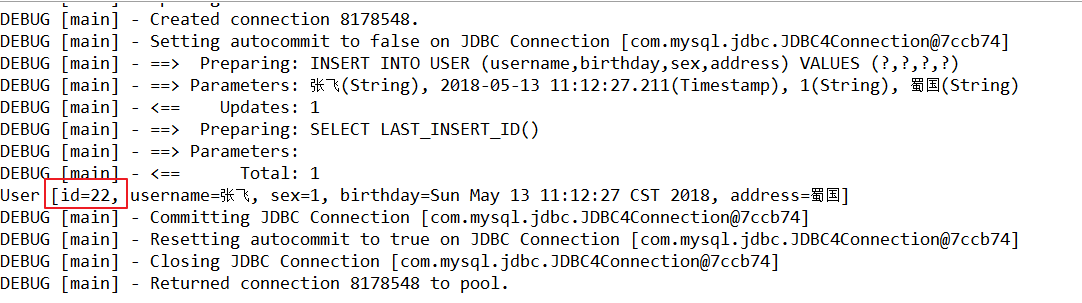
通过修改UserMapper.xml映射文件，可以将Mysql自增主键返回：

如下添加selectKey 标签

|  |
| --- |
| <!-- 保存用户 -->  <insert id=*"saveUser"* parameterType=*"com.itheima.mybatis.pojo.User"*>  <!-- selectKey 标签实现主键返回 -->  <!-- keyProperty：主键对应的pojo中的哪一个属性 -->  <!-- order：设置在执行insert语句前执行查询id的sql，孩纸在执行insert语句之后执行查询id的sql -->  <!-- resultType：设置返回的id的类型 -->  <selectKey keyProperty=*"id"* order=*"AFTER"*  resultType=*"Integer"*>  SELECT LAST\_INSERT\_ID()  </selectKey>  INSERT INTO USER  (username,birthday,sex,address) VALUES  (#{username},#{birthday},#{sex},#{address})  </insert> |

LAST\_INSERT\_ID():是Mysql的函数，返回auto\_increment自增列新记录id值。

效果如下图所示：



返回的id为22，能够正确的返回id了。

### 修改用户

根据用户id修改用户名

使用的Sql：UPDATE USER SET USERNAME = '赵云' WHERE ID= 1

#### 映射文件

在UserMapper.xml配置文件中添加如下内容：

|  |
| --- |
| <!-- 更新用户 -->  <update id=*"updateUserById"* parameterType=*"com.itheima.mybatis.pojo.User"*>  UPDATE USER SET  USERNAME = #{username} WHERE ID = #{id}  </update> |

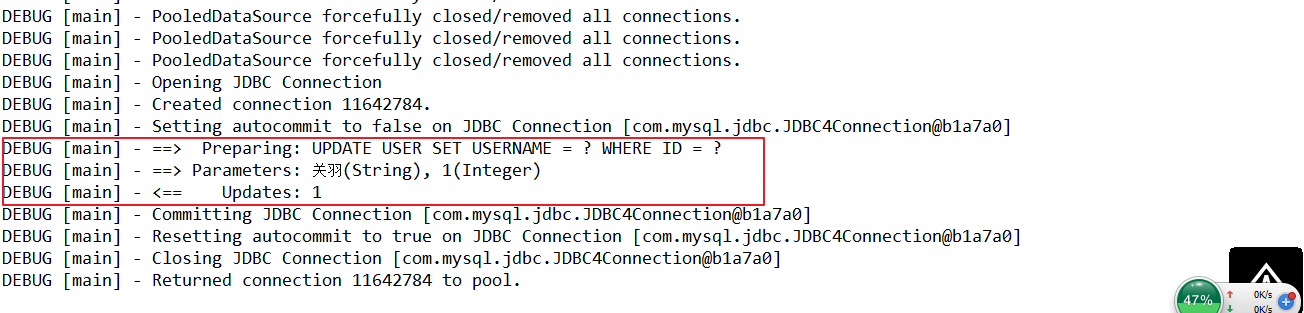
#### 测试程序

MybatisTest中添加测试方法如下：

|  |
| --- |
| @Test  **public** **void** testUpdateUserById() **throws** Exception{  // 1. 加载SqlMapConfig.xml配置文件  InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");  // 2. 创建SqlSessionFactoryBuilder对象  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  // 3. 创建SqlSessionFactory对象  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  // 4. 创建SqlSession对象  SqlSession sqlSession = sqlSessionFactory.openSession();  // 5. 执行SqlSession对象执行更新  // 创建需要更新的User  User user = **new** User();  user.setId(1);  user.setUsername("关羽");    sqlSession.update("test.updateUserById", user);  // 需要进行事务提交  sqlSession.commit();  // 7. 释放资源  sqlSession.close();  } |

#### 效果

测试效果如下图：



### 删除用户

根据用户id删除用户

使用的Sql：DELETE FROM USER WHERE ID = 1

#### 映射文件

在UserMapper.xml配置文件中添加如下内容：

|  |
| --- |
| <!-- 删除用户 -->  <delete id=*"deleteUserById"* parameterType=*"Integer"*>  delete from user where  id=#{id}  </delete> |

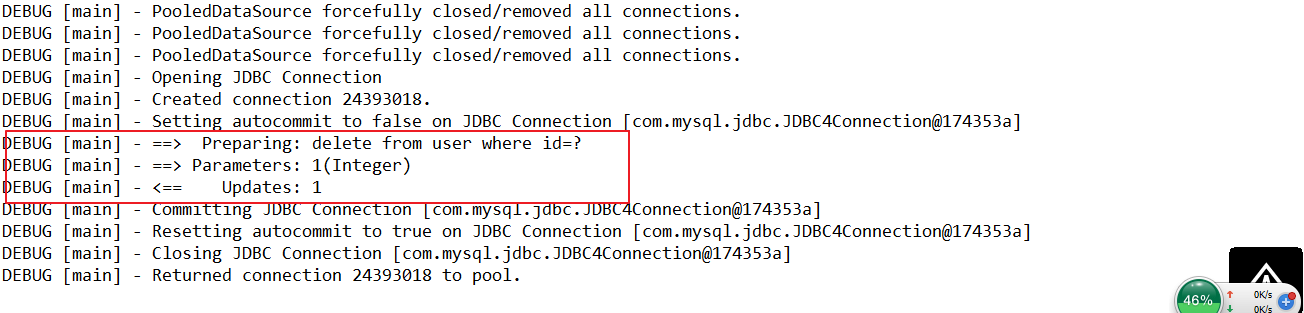
#### 测试程序

MybatisTest中添加测试方法如下：

|  |
| --- |
| @Test  **public** **void** testDeleteUserById() **throws** Exception{  // 1. 加载SqlMapConfig.xml配置文件  InputStream inputStream = Resources.getResourceAsStream("SqlMapConfig.xml");  // 2. 创建SqlSessionFactoryBuilder对象  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  // 3. 创建SqlSessionFactory对象  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  // 4. 创建SqlSession对象  SqlSession sqlSession = sqlSessionFactory.openSession();  // 5. 执行SqlSession对象执行删除  sqlSession.delete("test.deleteUserById", 1);  // 需要进行事务提交  sqlSession.commit();  // 7. 释放资源  sqlSession.close();  } |

#### 效果

测试效果如下图：



## Mybatis解决原生jdbc编程的问题

1、频繁创建、释放数据库连接造成系统资源浪费，影响系统性能。使用数据库连接池技术可以解决此问题。

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

2、Sql语句写在代码中造成代码不易维护，实际应用中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。

3、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（硬编码）。

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。

4、对结果集解析麻烦（查询列硬编码），Sql变化会导致解析代码变化，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。

# 第2章Mybatis的两种Dao开发方式

使用MyBatis开发Dao，通常有两个方法，即原始Dao开发方法和Mapper动态代理开发方法。

## 功能需求

使用MyBatis开发DAO实现以下的功能：

* 根据用户id查询一个用户信息

## SqlSession使用分析

SqlSession中封装了对数据库的操作，如：查询、插入、更新、删除等。

通过SqlSessionFactory创建SqlSession，而SqlSessionFactory是通过SqlSessionFactoryBuilder进行创建。

SqlSessionFactoryBuilder用于创建SqlSessionFacoty，SqlSessionFacoty一旦创建完成就不需要SqlSessionFactoryBuilder了，因为SqlSession是通过SqlSessionFactory生产，所以可以将SqlSessionFactoryBuilder当成一个工具类使用，最佳使用范围是方法范围即方法体内局部变量。

SqlSessionFactory是一个接口，接口中定义了openSession的不同重载方法，SqlSessionFactory的最佳使用范围是整个应用运行期间，一旦创建后可以重复使用，通常以单例模式管理SqlSessionFactory。

SqlSession是一个面向用户的接口， sqlSession中定义了数据库操作方法。

每个线程都应该有它自己的SqlSession实例。SqlSession的实例不能共享使用，它也是线程不安全的（因为SqlSession底层对应的是Connection连接）。因此最佳的范围是请求或方法范围。

## 原始Dao开发方式

原始Dao开发方式需要程序员编写Dao接口和Dao实现类。

### 映射文件

编写映射文件如下：（也可以使用入门程序完成的映射文件）

|  |
| --- |
| <?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">  <!-- namespace：命名空间，用于隔离sql，还有一个很重要的作用，后面会讲 -->  <mapper namespace=*"test"*>  <!-- 根据id查询用户 -->  <select id=*"queryUserById"* parameterType=*"Integer"*  resultType=*"com.itheima.mybatis.pojo.User"*>  select \* from user where id = #{id}  </select>  </mapper> |

### Dao接口

先进行DAO的接口开发，编码如下：

|  |
| --- |
| **public** **interface** UserDao {  /\*\*  \* 根据id查询用户  \*  \* **@param** id  \* **@return**  \*/  User queryUserById(**int** id);  } |

### Dao实现类

编写的Dao实现类如下

|  |
| --- |
| **public** **class** UserDaoImpl **implements** UserDao {  **private** SqlSessionFactory sqlSessionFactory;  **public** UserDaoImpl(SqlSessionFactory sqlSessionFactory) {  **super**();  **this**.sqlSessionFactory = sqlSessionFactory;  }  @Override  **public** User queryUserById(**int** id) {  // 创建SqlSession  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();  // 执行查询逻辑  User user = sqlSession.selectOne("test.queryUserById", id);  // 释放资源  sqlSession.close();  **return** user;  }  } |

### Dao测试

创建一个JUnit的测试类，对UserDao进行测试，测试代码如下：

|  |
| --- |
| **public** **class** UserDaoTest {  **private** SqlSessionFactory sqlSessionFactory;  @Before  **public** **void** init() **throws** Exception {  // 创建SqlSessionFactoryBuilder  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = **new** SqlSessionFactoryBuilder();  // 加载SqlMapConfig.xml配置文件  InputStream inputStream = Resources.*getResourceAsStream*("SqlMapConfig.xml");  // 创建SqlsessionFactory  **this**.sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  }  @Test  **public** **void** testQueryUserById() {  // 创建DAO  UserDao userDao = **new** UserDaoImpl(**this**.sqlSessionFactory);  // 执行查询  User user = userDao.queryUserById(2);  System.***out***.println(user);  }  } |

### 原始Dao开发方式源码分析SQL语句执行过程

现在我们可以通过调用SqlSession接口中的方法实现CRUD操作了，那么SqlSession接口中的方法是如何实现的。

此处一起来分析SqlSession接口中的insert()方法的执行过程（也可断点进入其它接口，原理一样），我们进行源码跟踪发现了Mybatis只是对于Jdbc的封装。

|  |
| --- |
| insert()方法的执行过程分析  \* DefaultSqlSession----> return update(statement, parameter);185行  \* 进入DefaultSqlSession---> public int update(String statement, Object parameter) {方法定义  \* 进入198行--- return executor.update(ms, wrapCollection(parameter));  \* 找到executor对象的update方法------------查看update()方法-----------------BaseExecutor类的update()方法  \* 进入BaseExecutor类的update()第117行return doUpdate(ms, parameter);  \* 查看doUpdate()方法------------------查看SimpleExecutor类的doUpdate()方法  \* 查看doUpdate()方法------------------stmt = prepareStatement(handler, ms.getStatementLog()); 49行  \* 查看prepareStatement()方法----------进入这个方法的84,85行  \* 得到代码： Connection connection = getConnection(statementLog);  stmt = handler.prepare(connection, transaction.getTimeout());  查看代码：  stmt = handler.prepare(connection, transaction.getTimeout()); //StatementHandler的子类BaseStatementHandler类  查看BaseStatementHandler类的prepare（）方法，该类的84行  再进入这个类的88行,找到代码:  statement = instantiateStatement(connection);  进入SimpleStatementHandler类的instantiateStatement(connection)  protected Statement instantiateStatement(Connection connection) throws SQLException {  if (mappedStatement.getResultSetType() != null) {  return connection.createStatement(mappedStatement.getResultSetType().getValue(), ResultSet.CONCUR\_READ\_ONLY);  } else {  return connection.createStatement();  }  } |

### 思考

原始方式开发Dao层时，如果接口中定义了很多方法，那么实现类中就得写很多的实现代码，但Dao层的逻辑总结起来就是增、删、改、查，是否可以不写实现类？

## Mapper动态代理方式

### 开发规范

Mapper动态代理开发方式只需要程序员开发Mapper接口（相当于Dao接口），Mybatis框架会根据接口定义创建接口的动态代理对象，代理对象的方法同Dao接口实现类中的方法。

Mapper接口开发需要遵循以下4个规范：

* Mapper映射文件中的namespace与mapper接口的类路径相同。
* Mapper接口方法名和Mapper映射文件中定义的每个Sql的id相同
* Mapper接口方法的输入参数类型和Mapper映射文件中定义的每个Sql的ParameterType的类型相同
* Mapper接口方法的输出参数类型和Mapper映射文件中定义的每个Sql的resultType的类型相同

### Mapper.xml(映射文件)

定义Mapper映射文件UserMapper.xml，放在resources的mapper目录下，

UserMapper.xml配置文件内容：

|  |
| --- |
| <?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，使用动态代理开发DAO，1. namespace必须和Mapper接口类路径一致 -->  <mapper namespace=*"com.itheima.mybatis.mapper.UserMapper"*>  <!-- 根据用户id查询用户 -->  <!-- 2. id必须和Mapper接口方法名一致 -->  <!-- 3. parameterType必须和接口方法参数类型一致 -->  <!-- 4. resultType必须和接口方法返回值类型一致 -->  <select id=*"queryUserById"* parameterType=*"Integer"*  resultType=*"com.itheima.mybatis.pojo.User"*>  select \* from user where id = #{id}  </select>  </mapper> |

### UserMapper(接口文件)

创建UserMapper接口代码如下：

|  |
| --- |
| **public** **interface** UserMapper {  /\*\*  \* 根据id查询  \*  \* **@param** id  \* **@return**  \*/  User queryUserById(**int** id);  } |

### 加载UserMapper.xml文件

修改SqlMapConfig.xml文件，添加以下所示的内容：

|  |
| --- |
| <!-- 加载映射文件 -->  <mappers>  <mapper resource=*"sqlmap/UserMapper.xml"* />  <mapper resource=*"mapper/UserMapper.xml"* />  </mappers> |

### 测试

编写的测试方法如下：

|  |
| --- |
| **public** **class** UserMapperTest {  **private** SqlSessionFactory sqlSessionFactory;  @Before  **public** **void** init() **throws** Exception {  // 创建SqlSessionFactoryBuilder  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = **new** SqlSessionFactoryBuilder();  // 加载SqlMapConfig.xml配置文件  InputStream inputStream = Resources.*getResourceAsStream*("SqlMapConfig.xml");  // 创建SqlsessionFactory  **this**.sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  }  @Test  **public** **void** testQueryUserById() {  // 获取sqlSession  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();  // 从sqlSession中获取Mapper接口的代理对象  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  // 执行查询方法  User user = userMapper.queryUserById(2);  System.***out***.println(user);  sqlSession.close();  }  } |

### Mapper动态代理方式实现DAO源码分析

我们总结一下会发现，采用代理方式开发DAO是非常简单的，只需要定义DAO接口并开发出它所对应的sql映射文件，就可以实现DAO。我们一起思考一下，这个接口的实现类是没有的，那么DAO接口的实现是如何产生的？

通过对源代码进行跟踪，我们会发现这个过程如下：

|  |
| --- |
| 在调用SqlSession接口的getMapper()方法时  getMapper(Class<T> type, SqlSession sqlSession)  它的底层进一步调用了JDK动态代理方式来生成代理子类对象的  protected T newInstance(MapperProxy<T> mapperProxy) {  return (T) Proxy.newProxyInstance(mapperInterface.getClassLoader(),  new Class[] { mapperInterface }, mapperProxy);  }  当调用具体的方法时，会去调用MapperProxy的invoke()方法  这个invoke()的最后一行： return mapperMethod.execute(sqlSession, args);  继续跟踪这个方法：execute(sqlSession, args);  下面是它的方法体  Object result;  switch (command.getType()) {  case INSERT: {  Object param = method.convertArgsToSqlCommandParam(args);  result = rowCountResult(sqlSession.insert(command.getName(), param));  break;  }  case UPDATE: {  Object param = method.convertArgsToSqlCommandParam(args);  result = rowCountResult(sqlSession.update(command.getName(), param));  break;  }  case DELETE: {  Object param = method.convertArgsToSqlCommandParam(args);  result = rowCountResult(sqlSession.delete(command.getName(), param));  break;  }  case SELECT:  if (method.returnsVoid() && method.hasResultHandler()) {  executeWithResultHandler(sqlSession, args);  result = null;  } else if (method.returnsMany()) {  result = executeForMany(sqlSession, args);  } else if (method.returnsMap()) {  result = executeForMap(sqlSession, args);  } else if (method.returnsCursor()) {  result = executeForCursor(sqlSession, args);  } else {  Object param = method.convertArgsToSqlCommandParam(args);  result = sqlSession.selectOne(command.getName(), param);  }  break;  case FLUSH:  result = sqlSession.flushStatements();  break;  default:  throw new BindingException("Unknown execution method for: " + command.getName());  } |

## 小结

Mybatis官方推荐使用Mapper接口的方式开发Dao，这样程序员就不用去开发实现类了，这种方式是我们后面进入企业的主流方式。但是有一些企业还是用第一种方式开发Dao，因为在介绍Mybatis时，我们知道Mybatis的前身是ibatis，而ibatis是没有提供这种方式开发Dao的，所有很多公司还没有习惯使用第二种方式开发Dao。

# 全局配置文件SqlMapConfig.xml

## 配置内容和顺序

SqlMapConfig.xml中配置的内容和顺序如下：

**properties**（属性配置）

settings（全局配置参数）

**typeAliases**（类型别名）

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）

注意：标签是有顺序的，如上。

## properties（属性）

使用的properties的好处：配置数据共享

db.properties配置文件内容如下：

|  |
| --- |
| 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：

|  |
| --- |
| <?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>  <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/UserMapper.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。

## typeAliases（类型别名）

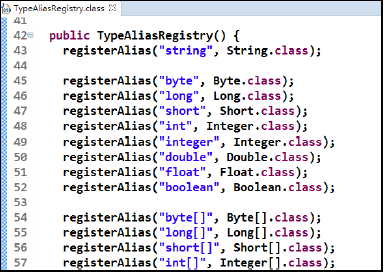
Mybatis默认情况下支持一些参数写别名，如下，也可以参考官方文档说明（第19页）

### Mybatis默认支持的别名

|  |  |
| --- | --- |
| 别名 | 映射的类型 |
| \_byte | byte |
| \_long | long |
| \_short | short |
| \_int | int |
| \_integer | int |
| \_double | double |
| \_float | float |
| \_boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |
| map | Map |

这些都是支持的默认别名。我们也可以从源码角度来看它们分别都是如何定义出来的。

可以参考TypeAliasRegistery.class的源码。

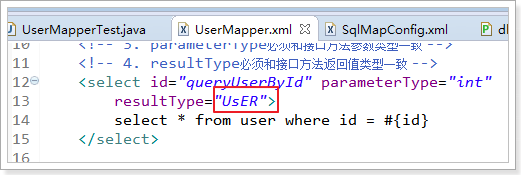


### 自定义别名

在SqlMapConfig.xml中配置如下：

|  |
| --- |
| <?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 -->  <property name=*"jdbc.username"* value=*"root123"* />  <property name=*"jdbc.password"* value=*"root123"* />  </properties>  <typeAliases>  <!-- 单个别名定义 -->  <typeAlias alias=*"user"* type=*"com.itheima.mybatis.pojo.User"* />  <!-- 批量别名定义，扫描整个包下的类，别名为类名（大小写不敏感） -->  <package name=*"com.itheima.mybatis.pojo"* />  <package name=*"其它包"* />  </typeAliases>  <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/UserMapper.xml"* />  <mapper resource=*"mapper/UserMapper.xml"* />  </mappers>  </configuration> |

在mapper.xml配置文件中，就可以使用设置的别名了



**注意：**别名大小写不敏感，推荐使用<package/>批量定义别名

## mappers（映射器）

Mapper配置的几种方法：

### 通过resource加载单个映射文件

使用相对于类路径的资源（现在的使用方式）

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

### 另外两种加载方式（仅支持Mapper动态代理开发Dao）

### 通过mapper接口加载单个映射文件

* **通过Mapper接口类加载单个映射文件**

使用mapper接口类路径，如：

<mapper class="com.itheima.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。

* **批量加载映射文件**

加载指定包下的所有Mapper接口类，如：

<package name="com.itheima.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。

* **注意：**使用Mapper动态代理方式开发Dao层时，推荐使用<package/>批量加载的方式，这种方式也用的最多。

# 第四章 Mybatis输入类型和结果类型

## parameterType(输入类型)

### 传递简单类型

参考第一天内容。

使用#{}占位符，或者${}进行Sql拼接。

### 传递Pojo对象

参考第一天的内容。

#{}或者${}括号中的值为Pojo属性名称。

### 传递Pojo包装对象

包装对象：Pojo类中的一个属性是另外一个Pojo。

为什么使用包装对象？

在平时的开发中，不一定就是单纯的对一个实体进行增删改查，例如完成用户信息的综合查询，有时需要传入查询条件很复杂，可能包括用户信息、关联表的其它信息等。针对这种需求，在Mybatis中我们可以使用自定义的包装类型的Pojo，在包装类型的Pojo中将复杂的查询条件包装进去。

**功能需求：**根据用户名模糊查询用户信息，查询条件放到QueryVo的user属性中。

#### 编写QueryVo

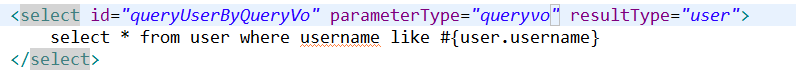
|  |
| --- |
| **public** **class** QueryVo {  // 包含其他的pojo  **private** User user;  **public** User getUser() {  **return** user;  }  **public** **void** setUser(User user) {  **this**.user = user;  }  } |

#### Sql语句

SELECT \* FROM USER WHERE USERNAME LIKE '%王%'

#### Mapper.xml文件

在UserMapper.xml中配置Sql，如下图。



#### Mapper接口

在UserMapper接口中添加方法，如下图：



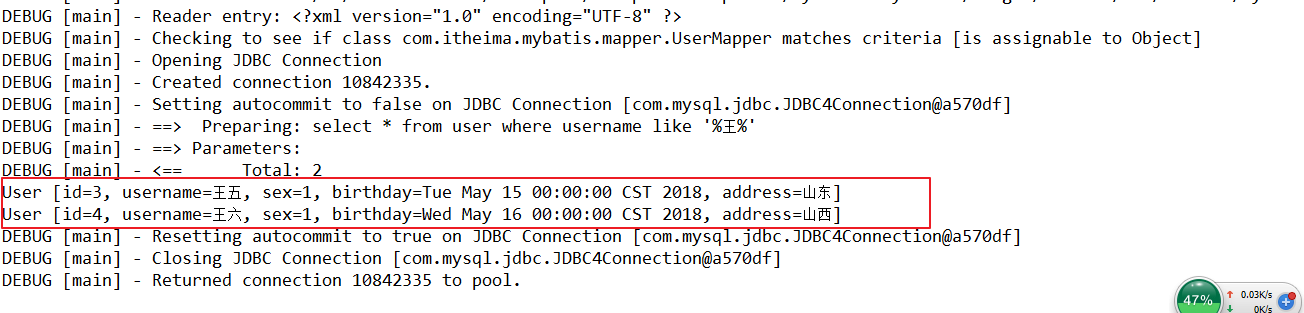
#### 测试方法

在UserMapeprTest增加测试方法，如下：

|  |
| --- |
| @Test  **public** **void** testQueryUserByQueryVo() {  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  // 使用userMapper执行查询，使用包装对象  QueryVo queryVo = **new** QueryVo();  // 设置user条件  User user = **new** User();  user.setUsername("%王%");  // 设置到包装对象中  queryVo.setUser(user);  // 执行查询  List<User> list = userMapper.queryUserByQueryVo(queryVo);  **for** (User u : list) {  System.***out***.println(u);  }  sqlSession.close();  } |

#### 效果

测试结果如下图：



## resultType(输出结果类型)

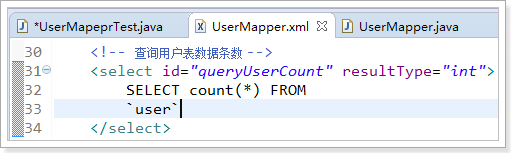
### 输出简单类型

功能需求：查询用户表数据条数

Sql语句：SELECT count(\*) FROM user

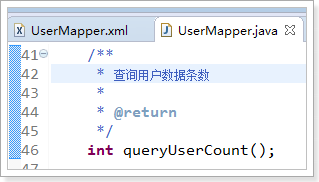
#### Mapper.xml文件

在UserMapper.xml中配置Sql，如下图：



#### Mapper接口

在UserMapper添加方法，如下图：



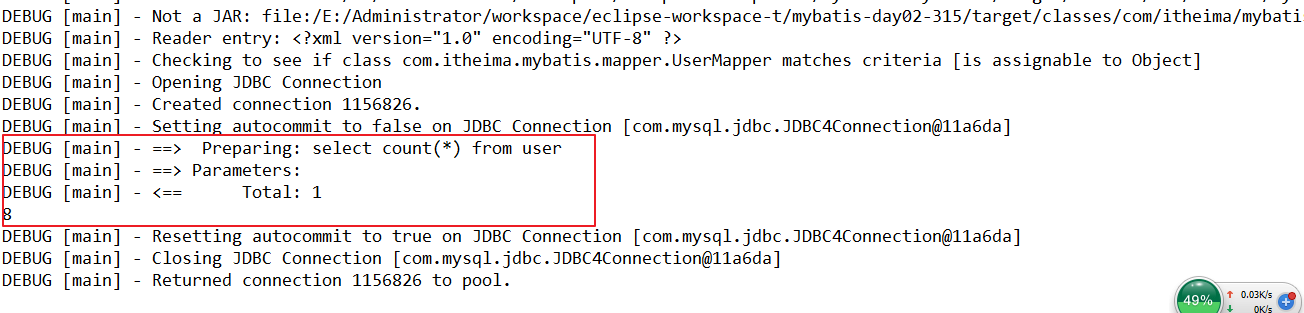
#### 测试方法

在UserMapeprTest增加测试方法，如下：

|  |
| --- |
| @Test  **public** **void** testQueryUserCount() {  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  **int** count = userMapper.queryUserCount();  System.***out***.println(count);  sqlSession.close();  } |

#### 效果

测试结果如下图：



注意：输出简单类型必须查询出来的结果集有一条记录，最终将第一个字段的值转换为输出类型

### 输出Pojo对象

参考第一天内容

### 输出Pojo列表

参考第一天内容

## 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。

**功能需求：**查询订单表orders的所有数据

Sql语句：SELECT id, user\_id, number, createtime, note FROM orders

### 创建数据表、初始化测试数据、声明Pojo对象

创建订单表并查询订单表的全部数据，Sql语句如下：

|  |
| --- |
| -- ----------------------------  -- Table structure for `orders`  -- ----------------------------  DROP TABLE IF EXISTS `orders`;  CREATE TABLE `orders` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `user\_id` int(11) NOT NULL COMMENT '下单用户id',  `number` varchar(32) NOT NULL COMMENT '订单号',  `createtime` datetime NOT NULL COMMENT '创建订单时间',  `note` varchar(100) DEFAULT NULL COMMENT '备注',  PRIMARY KEY (`id`),  KEY `FK\_orders\_1` (`user\_id`),  CONSTRAINT `FK\_orders\_id` FOREIGN KEY (`user\_id`) REFERENCES `user` (`id`) ON DELETE NO ACTION ON UPDATE NO ACTION  ) ENGINE=InnoDB AUTO\_INCREMENT=6 DEFAULT CHARSET=utf8;  -- ----------------------------  -- Records of orders  -- ----------------------------  INSERT INTO `orders` VALUES ('3', '3', '1000010', '2015-02-04 13:22:35', null);  INSERT INTO `orders` VALUES ('4', '3', '1000011', '2015-02-03 13:22:41', null);  INSERT INTO `orders` VALUES ('5', '4', '1000012', '2015-02-12 16:13:23', null); |

Orders对象：

|  |
| --- |
| **public** **class** Orders {  // 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OrdersMapper.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=*"com.itheima.mybatis.mapper.OrdersMapper"*>  <!-- 查询所有的订单数据 -->  <select id=*"queryOrdersAll"* resultType=*"orders"*>  SELECT id, user\_id,  number,  createtime, note FROM orders  </select>  </mapper> |

### Mapper接口

编写接口如下：

|  |
| --- |
| **public** **interface** OrdersMapper {  /\*\*  \* 查询所有订单  \*  \* **@return**  \*/  List<Orders> queryOrdersAll();  } |

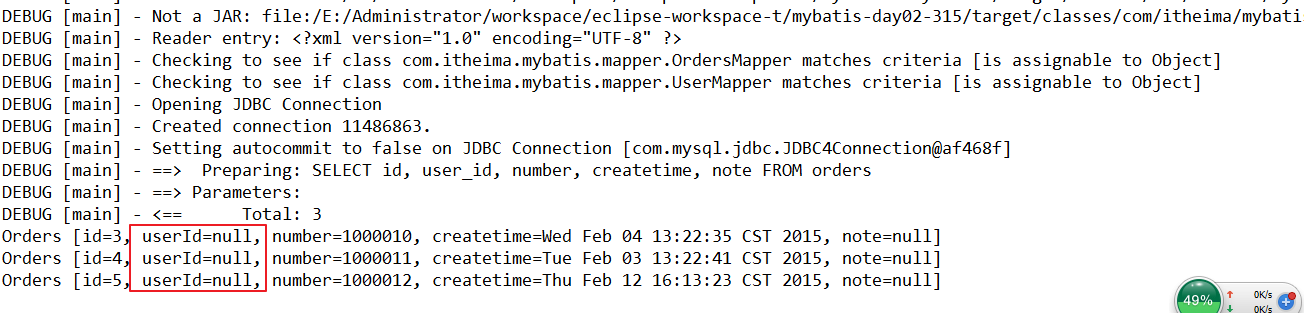
### 测试方法

编写测试方法OrdersMapperTest如下：

|  |
| --- |
| **public** **class** OrdersMapperTest {  **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  OrdersMapper orderMapper = sqlSession.getMapper(OrdersMapper.**class**);  // 执行查询  List<Orders> list = orderMapper.queryOrdersAll();  **for** (Orders order : list) {  System.***out***.println(order);  }  sqlSession.close();  }  } |

### 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=*"com.itheima.mybatis.mapper.OrdersMapper"*>  <!-- 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=*"orders"* 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=*"queryOrdersAll"* resultMap=*"orderResultMap"*>  SELECT id, user\_id,  number,  createtime, note FROM orders  </select>  </mapper> |

### 效果

只需要修改Mapper.xml就可以了，再次测试结果如下：

