**Mybatis框架课程第三天**

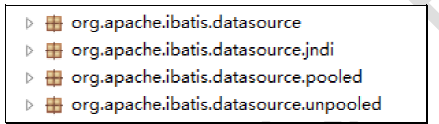
# Mybatis连接池与事务深入

## Mybatis的连接池技术

我们在前面的WEB课程中也学习过类似的连接池技术，而在Mybatis中也有连接池技术，但是它采用的是自己的连接池技术。在Mybatis的SqlMapConfig.xml配置文件中，通过<dataSource type=” POOLED”>来实现Mybatis中连接池的配置。

### Mybatis连接池的分类

在Mybatis中我们将它的数据源dataSource分为以下几类：



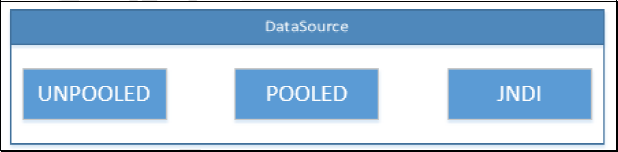
可以看出Mybatis将它自己的数据源分为三类：

UNPOOLED 不使用连接池的数据源

POOLED 使用连接池的数据源

JNDI 使用JNDI实现的数据源

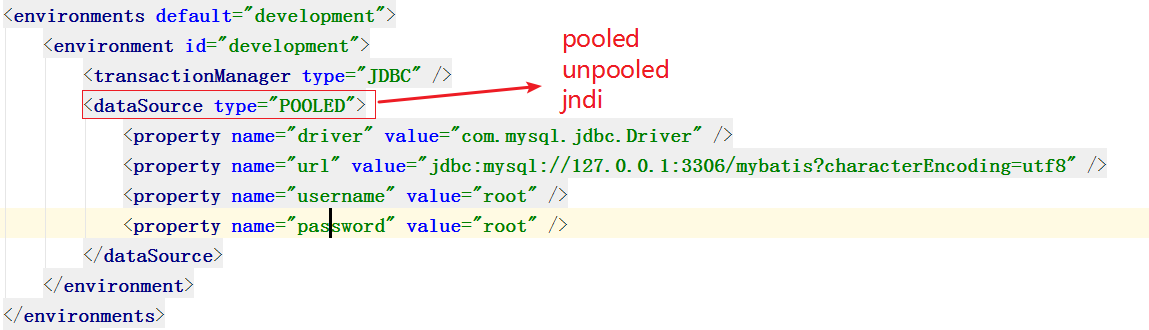
具体结构如下：



相应地，MyBatis内部分别定义了实现了java.sql.DataSource接口的UnpooledDataSource，PooledDataSource类来表示UNPOOLED、POOLED类型的数据源。

在这三种数据源中，我们一般采用的是POOLED数据源（很多时候我们所说的数据源就是为了更好的管理数据库连接，也就是我们所说的连接池技术）。

### Mybatis中的数据源配置



MyBatis在初始化时，解析此文件，根据<dataSource>的type属性来创建相应类型的的数据源DataSource，即：

type=”POOLED”：MyBatis会创建PooledDataSource实例

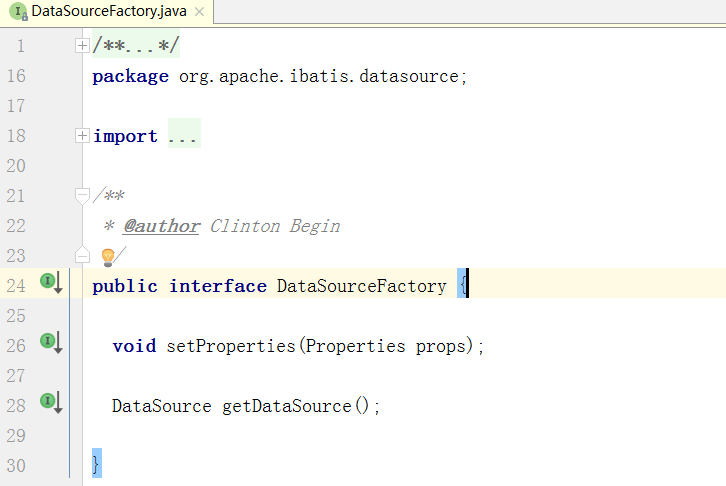
type=”UNPOOLED” ： MyBatis会创建UnpooledDataSource实例

type=”JNDI”：MyBatis会从JNDI服务上查找DataSource实例，然后返回使用

### Mybatis中DataSource的存取

MyBatis是通过工厂模式来创建数据源DataSource对象的，MyBatis定义了抽象的工厂接口*:****org.apache.ibatis.datasource.DataSourceFactory***,通过其getDataSource()方法返回数据源DataSource。

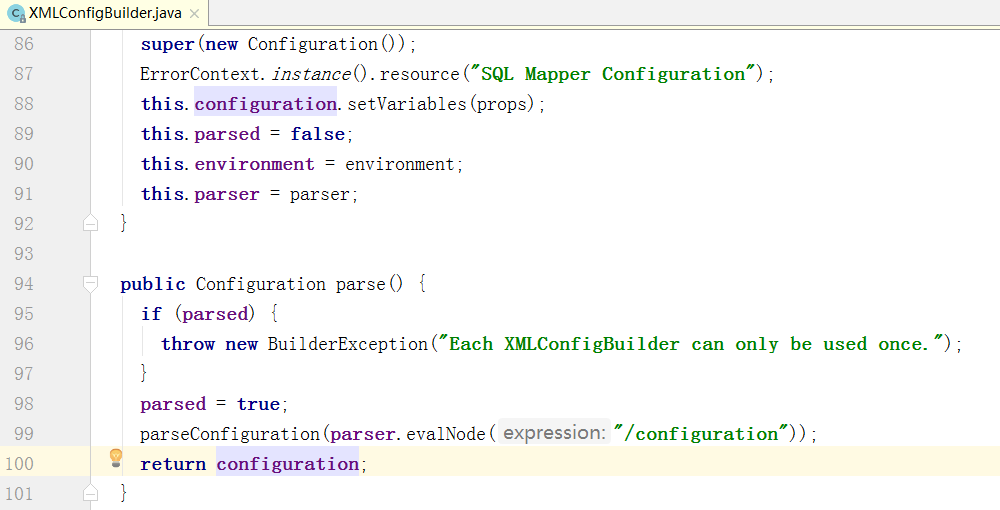
下面是DataSourceFactory源码，具体如下：



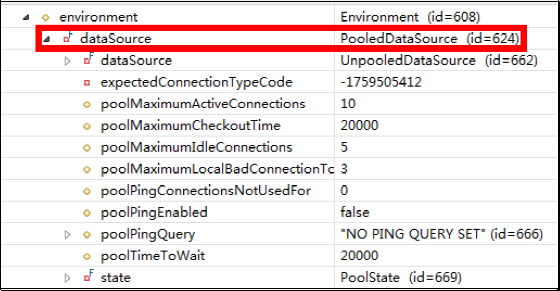
MyBatis创建了DataSource实例后，会将其放到Configuration对象内的Environment对象中， 供以后使用。

具体分析过程如下：

1. 先进入XMLConfigBuilder类中，可以找到如下代码：

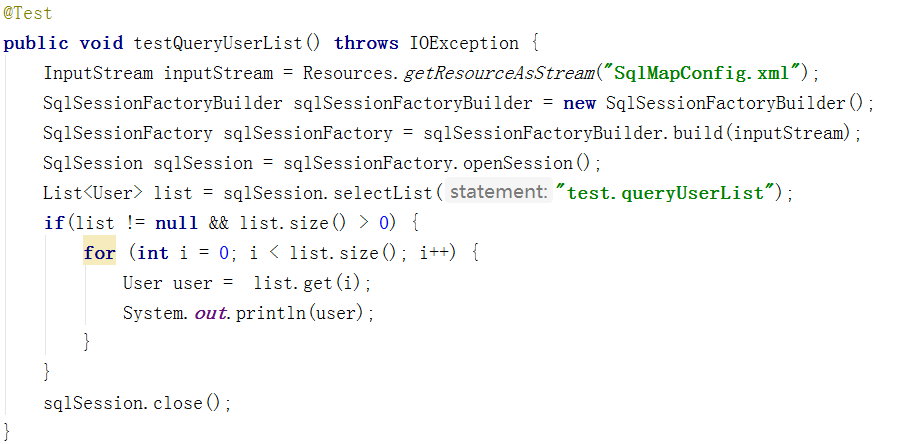


2.分析configuration对象的environment属性，结果如下：

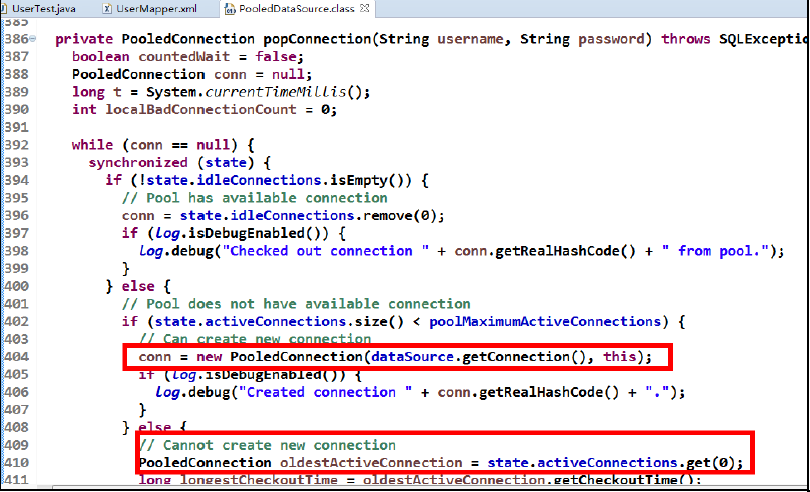


### Mybatis中连接的获取过程分析

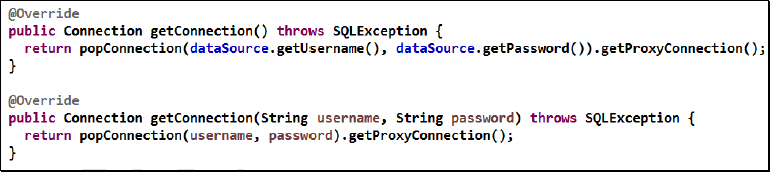
当我们需要创建􀀁SqlSession 对象并需要执行SQL 语句时，这时候MyBatis 才会去调用dataSource 对象来创建java.sql.Connection 对象。也就是说，java.sql.Connection 对象的创建一直延迟到执行SQL 语句的时候。



前4句都不会导致java.sql.Connection对象的创建，只有当第5 句sqlSession.selectList("test.queryUserList")，才会触发MyBatis 在底层执行下面这个方法来创建java.sql.Connection 对象。 如何证明它的加载过程呢？ 我们可以通过断点调试，在PooledDataSource 中找到如下popConnection()方法，如下所示：



下面是连接获取的源代码：



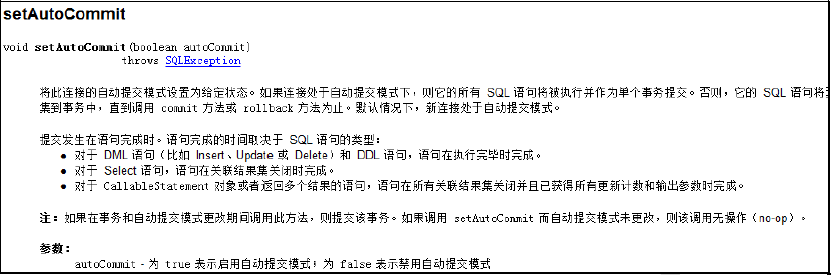
最后我们可以发现，真正连接打开的时间点，只是在我们执行SQL语句时，才会进行。其实这样做我们也可以进一步发现，数据库连接是我们最为宝贵的资源，只有在要用到的时候，才去获取并打开连接，当我们用完了就再立即将数据库连接归还到连接池中。

## Mybatis的事务控制

### JDBC中事务的回顾

在JDBC中我们可以通过手动方式将事务的提交改为手动方式，通过setAutoCommit()方法就可以调整。

通过JDK文档，我们找到该方法如下：

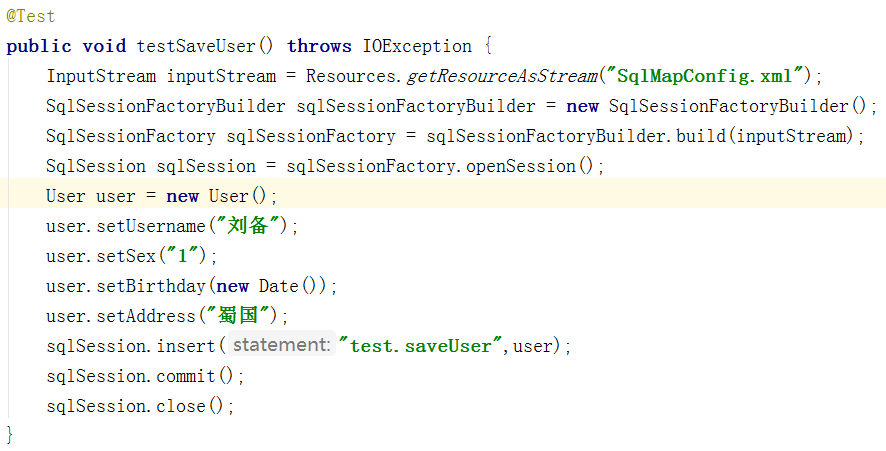


那么我们的Mybatis框架因为是对JDBC的封装，所以Mybatis框架的事务控制方式，本身也是用JDBC的setAutoCommit()方法来设置事务提交方式的。

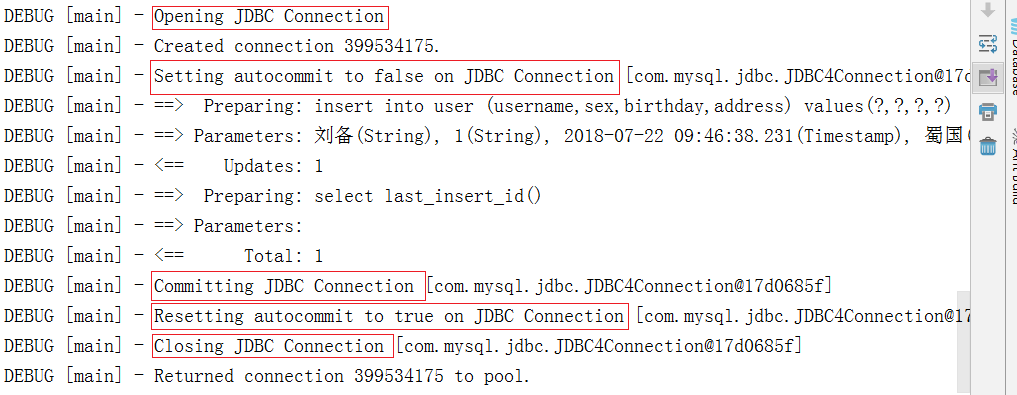
### Mybatis中事务提交方式

Mybatis中事务的提交方式，本质上就是调用JDBC的setAutoCommit()来实现事务控制。

我们运行之前所写的代码：



观察在它在控制台输出的结果：

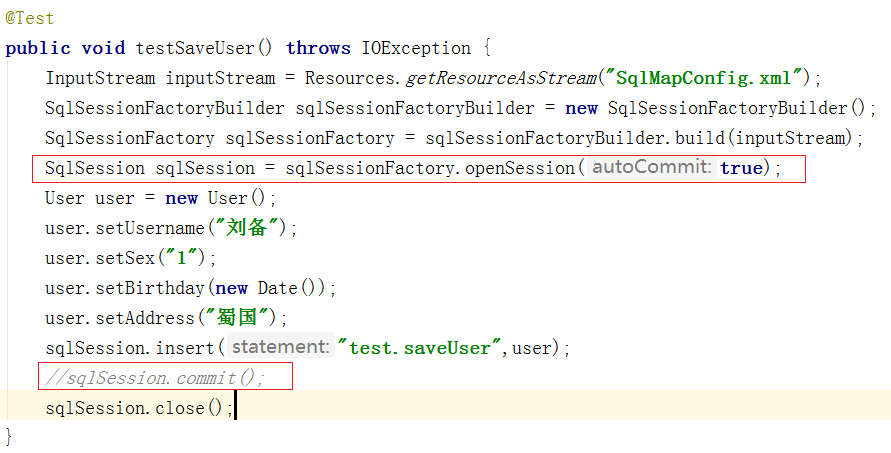


这是我们的Connection的整个变化过程，通过分析我们能够发现之前的CUD操作过程中，我们都要手动进行事务的提交，原因是setAutoCommit()方法，在执行时它的值被设置为false了，所以我们在CUD操作中，必须通过sqlSession.commit()方法来执行提交操作。

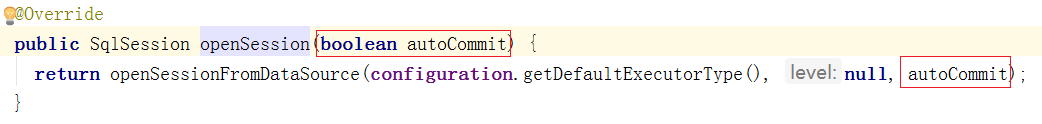
### Mybatis自动提交事务的设置

通过上面的研究和分析，现在我们一起思考，为什么CUD过程中必须使用sqlSession.commit()提交事务？主要原因就是在连接池中取出的连接，都会将调用connection.setAutoCommit(false)方法，这样我们就必须使用sqlSession.commit()方法，相当于使用了JDBC中的connection.commit()方法实现事务提交。

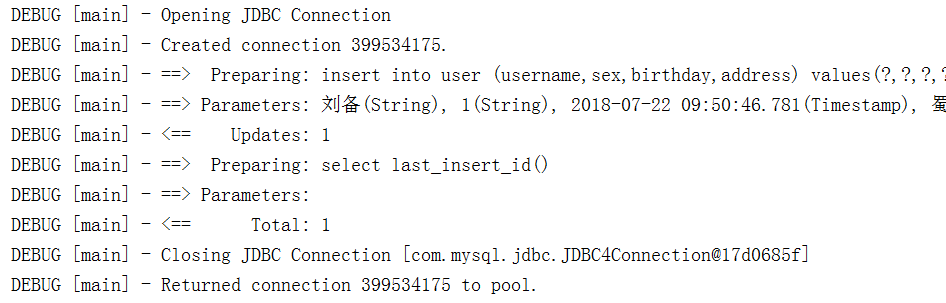
明白这一点后，我们现在一起尝试不进行手动提交，一样实现CUD操作。



所对应的DefaultSqlSessionFactory类的源代码：



运行的结果如下：

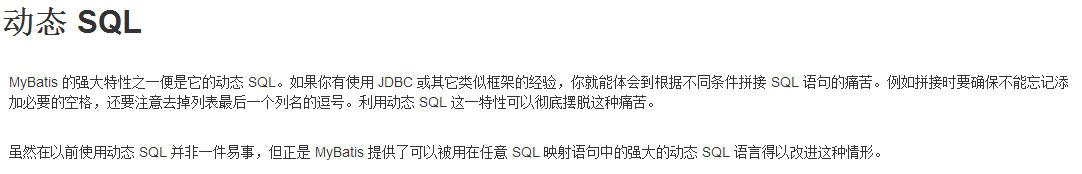


我们发现，此时事务就设置为自动提交了，同样可以实现CUD操作时记录的保存。虽然这也是一种方式，但就编程而言，设置为自动提交方式为false再根据情况决定是否进行提交，这种方式更常用。因为我们可以根据业务情况来决定提交是否进行提交。

# Mybatis映射文件的SQL深入

Mybatis的映射文件中，前面我们的SQL都是比较简单的，有些时候业务逻辑复杂时，我们的SQL是动态变化的，此时在前面的学习中我们的SQL就不能满足要求了。

参考的官方文档，描述如下：



通过Mybatis提供的各种标签方法实现动态拼接Sql。

**功能需求：**根据性别和名字查询用户

Sql语句：

SELECT id, username, birthday, sex, address FROM `user` WHERE sex = 1 AND username LIKE '%张%'

## If标签

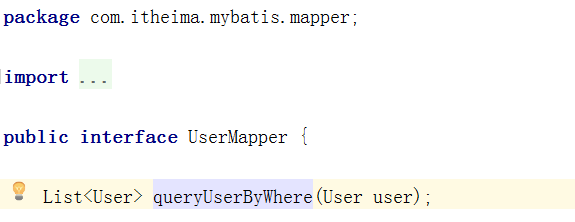
### Mapper.xml文件

UserMapper.xml配置Sql，如下：

|  |
| --- |
| <!-- 根据条件查询用户 -->  <select id=*"queryUserByWhere"* parameterType=*"user"* resultType=*"user"*>  SELECT id, username, birthday, sex, address FROM `user`  WHERE sex = #{sex} AND username LIKE #{username}  </select> |

### Mapper接口

编写Mapper接口，如下图：



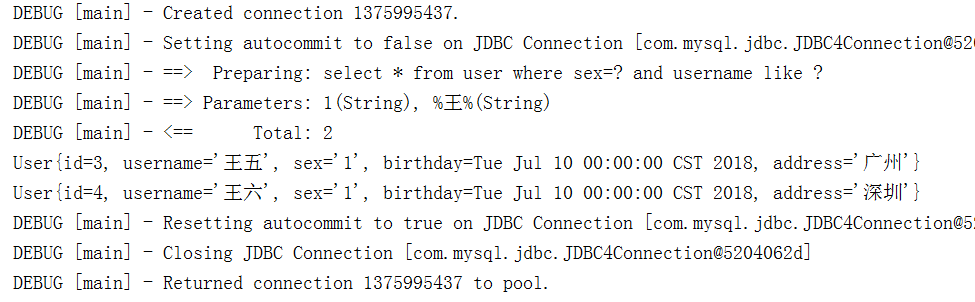
### 测试方法

在UserMapperTest添加测试方法，如下：

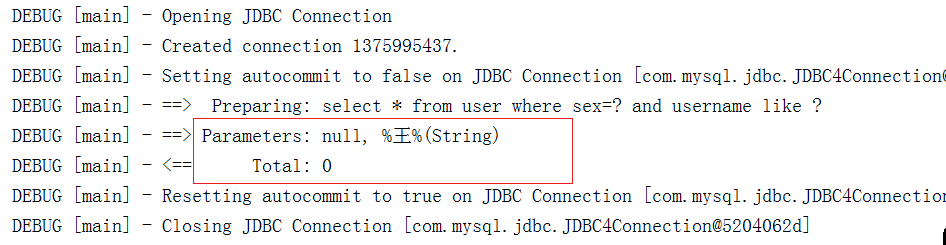
|  |
| --- |
| @Test  **public** **void** testQueryUserByWhere() {  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  // 使用userMapper执行根据条件查询用户  User user = **new** User();  user.setSex("1");  user.setUsername("%王%");  List<User> list = userMapper.queryUserByWhere(user);  **for** (User u : list) {  System.***out***.println(u);  }  sqlSession.close();  } |

### 效果

测试效果如下图：



如果注释掉 user.setSex("1")，测试结果如下图：



测试结果二很显然不合理。

按照之前所学的，要解决这个问题，需要编写多个Sql，查询条件越多，需要编写的Sql就更多了，显然这样是不靠谱的。

解决方案，使用动态Sql的if标签

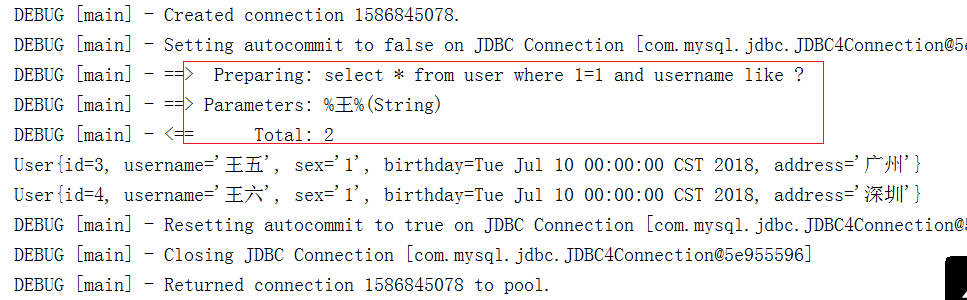
### 使用if标签

改造UserMapper.xml，如下：

|  |
| --- |
| <!-- 根据条件查询用户 -->  <select id=*"queryUserByWhere"* parameterType=*"user"* resultType=*"user"*>  SELECT id, username, birthday, sex, address FROM `user`  WHERE 1=1  <if test=*"sex != null and sex != ''"*>  AND sex = #{sex}  </if>  <if test=*"username != null and username != ''"*>  AND username LIKE  #{username}  </if>  </select> |

注意字符串类型的数据需要做不等于空字符串校验。

### 效果



如上图所示，测试OK

## Where标签

上面的Sql还有where 1=1 这样的语句，很麻烦

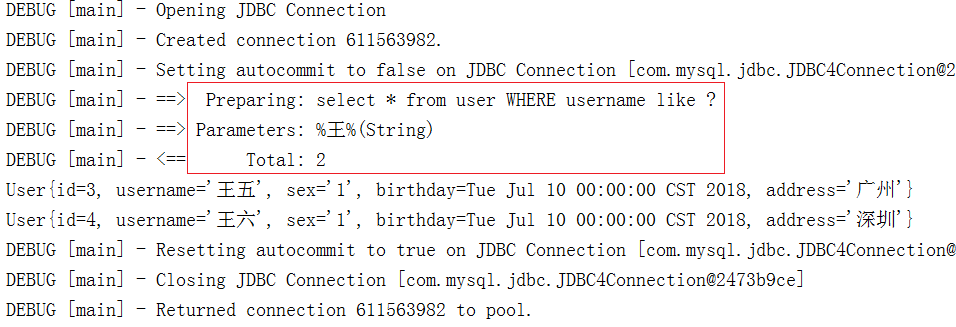
可以使用where标签进行改造

改造UserMapper.xml，如下

|  |
| --- |
| <!-- 根据条件查询用户 -->  <select id=*"queryUserByWhere"* parameterType=*"user"* resultType=*"user"*>  SELECT id, username, birthday, sex, address FROM `user`  <!-- where标签可以自动添加where，同时处理sql语句中第一个and关键字 -->  <where>  <if test=*"sex != null"*>  AND sex = #{sex}  </if>  <if test=*"username != null and username != ''"*>  AND username LIKE #{username}  </if>  </where>  </select> |

### 效果

测试效果如下图：



## Sql片段

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

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

|  |
| --- |
| <!-- 根据条件查询用户 -->  <select id=*"queryUserByWhere"* parameterType=*"user"* resultType=*"user"*>  <!-- SELECT id, username, birthday, sex, address FROM `user` -->  <!-- 使用include标签加载sql片段；refid是sql片段id -->  SELECT <include refid=*"userFields"* /> FROM `user`  <!-- where标签可以自动添加where关键字，同时处理sql语句中第一个and关键字 -->  <where>  <if test=*"sex != null"*>  AND sex = #{sex}  </if>  <if test=*"username != null and username != ''"*>  AND username LIKE #{username}  </if>  </where>  </select>  <!-- 声明sql片段 -->  <sql id=*"userFields"*>  id, username, birthday, sex, address  </sql> |

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

|  |
| --- |
| SELECT <include refid=*"com.itheima.mybatis.mapper.OrdersMapper.userFields"* /> FROM `user` |

## foreach标签

向Sql传递List，Mybatis使用foreach解析。

**功能需求：**根据多个id查询用户信息

查询Sql语句：

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

### Mapper.xml文件

UserMapper.xml添加Sql，如下：

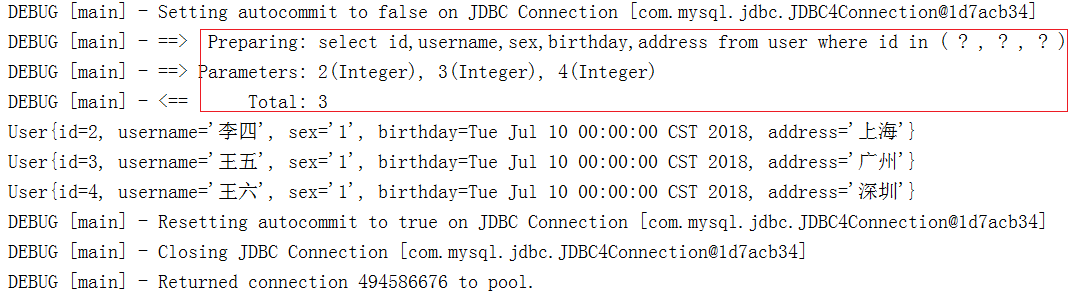
|  |
| --- |
| <!-- 根据ids查询用户 -->  <select id=*"queryUserByIds"* parameterType=*"list"* resultType=*"user"*>  SELECT \* FROM `user`  <where>  <!-- foreach标签，进行遍历 -->  <!-- collection：遍历的集合，如果传入List集合则必须为“list”，  如果传入普通数组，则必须为“array” -->  <!-- item：遍历的项目，可以随便写，，但是和后面的#{}里面要一致 -->  <!-- open：在前面添加的sql片段 -->  <!-- close：在结尾处添加的sql片段 -->  <!-- separator：指定遍历的元素之间使用的分隔符 -->  <foreach collection=*"list"* item=*"item"* open=*"id IN ("* close=*")"*  separator=*","*>  #{item}  </foreach>  </where>  </select> |

测试方法如下图：

|  |
| --- |
| @Test  **public** **void** testQueryUserByIds() {  SqlSession sqlSession = **this**.sqlSessionFactory.openSession();  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  // 使用userMapper执行根据条件查询用户  List<Integer> ids = **new** ArrayList<Integer>();  ids.add(2);  ids.add(3);  ids.add(4);  List<User> list = userMapper.queryUserByIds(ids);  **for** (User u : list) {  System.***out***.println(u);  }  sqlSession.close();  } |

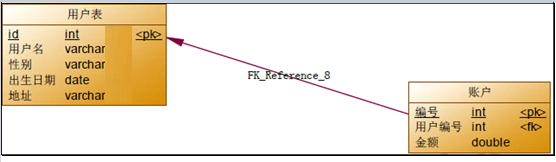
### 效果

测试效果如下图：



# 第3章Mybatis的多表关联查询

本次案例主要以最为简单的用户和账户的模型来分析Mybatis 多表关系。用户为User 表，账户为Account 表。一个用户（User）可以有多个账户（Account）。具体关系如下：



## 一对一查询

案例：查询所有账户信息，关联查询出各个账户所属的用户信息。

注意：因为一个账户只能属于某一个用户，所以从查询账户信息出发关联查询用户信息为一对一查询。如果从用户信息出发查询用户下的账户信息则为一对多查询，因为一个用户可以有多个账户。

### 方式一

#### 定义账户信息的POJO类

使用resultType，定义账户信息pojo 类，此pojo 类中包括了账户信息。具体实现如下：

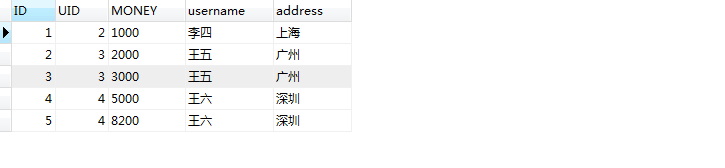
|  |
| --- |
| **package** com.itheima.mybatis.pojo;  **public class** Account {  **private** Integer **id**;  **private** Integer **uid**;  **private** Double **money**;  **private** User **user**;   **public** User getUser() {  **return user**;  }  **public void** setUser(User user) {  **this**.**user** = user;  }   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** Integer getUid() {  **return uid**;  }   **public void** setUid(Integer uid) {  **this**.**uid** = uid;  }   **public** Double getMoney() {  **return money**;  }   **public void** setMoney(Double money) {  **this**.**money** = money;  }   @Override  **public** String toString() {  **return "Account{"** +  **"id="** + **id** +  **", uid="** + **uid** +  **", money="** + **money** + **'\''** +  **'}'**;  } } |

#### 编写SQL语句

实现查询账户信息时，也要查询账户所对应的用户信息。

|  |
| --- |
| SELECT  a.\*, u.username,  u.address  FROM  account a  LEFT JOIN USER u ON a.uid = u.id; |

在MySQL中测试的查询结果如下：



#### 定义AccountUser类

为了能够封装上面SQL语句的查询结果，定义 AccountUser类中要包含账户信息同时还要包含用户信息，所以我们在定义AccountUser类时可以继承Account类。

|  |
| --- |
| **package** com.itheima.mybatis.pojo;  **public class** AccountUser **extends** Account {  **private** String **username**;  **private** String **address**;  **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** String getAddress() {  **return address**;  }   **public void** setAddress(String address) {  **this**.**address** = address;  }   @Override  **public** String toString() {  **return "AccountUser{"** +  **"username='"** + **username** + **'\''** +  **", address='"** + **address** + **'\''** +  **"} "** + **super**.toString();  } } |

#### 定义AccountMapper接口

定义AccountMapper接口，用于查询account相关的账户信息。为了查询账户信息同时还能关联查询出他的用户信息，所以我们的返回结果采用的是AccountUser类型

|  |
| --- |
| **package** com.itheima.mybatis.mapper;  **import** com.itheima.mybatis.pojo.Account; **import** com.itheima.mybatis.pojo.AccountUser;  **import** java.util.List;  **public interface** AccountMapper {   */\*\*  \* 查询所有账户信息，并关联查出用户信息  \** ***@return*** *\*/* List<AccountUser> queryAccountUserList(); } |

#### 定义AccountMapper.xml文件中的查询配置信息

|  |
| --- |
| <**select id="queryAccountUserList" resultType="accountuser"**>  SELECT  a.*\**, u.username,  u.address  FROM  account a  LEFT JOIN USER u ON a.uid = u.id; </**select**> |

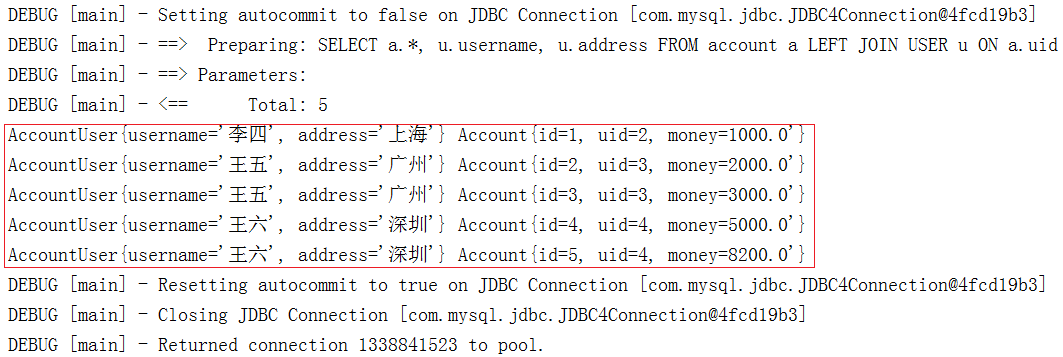
注意：因为上面查询的结果中包含了账户信息同时还包含了用户信息，所以我们的返回值类型returnType的值设置为AccountUser类型，这样就可以接收账户信息和用户信息了

#### 测试方法

编写测试方法，用于测试查询结果。

|  |
| --- |
| @Test **public void** testQueryAccountUserList() **throws** IOException {  InputStream inputStream = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  SqlSession sqlSession = sqlSessionFactory.openSession();  AccountMapper accountMapper = sqlSession.getMapper(AccountMapper.**class**);  List<AccountUser> list = accountMapper.queryAccountUserList();  **for** (**int** i = 0; i < list.size(); i++) {  AccountUser accountUser = list.get(i);  System.***out***.println(accountUser);  }  sqlSession.close(); } |

#### 效果



#### 小结

定义专门的Pojo类作为输出类型，其中定义了sql查询结果集所有的字段。此方法较为简单，企业中使用普遍，但缺乏设计，Pojo过多会造成维护不便。

### 方式二

使用resultMap，定义专门的resultMap用于映射一对一查询结果。

通过面向对象的(has a)关系可以得知，我们可以在Account类中加入一个User类的对象来代表这个账户是哪个用户的。

#### 修改Account类

在Account类中加入User类的对象作为Account类的一个属性。

|  |
| --- |
| **package** com.itheima.mybatis.pojo;  **public class** Account {  **private** Integer **id**;  **private** Integer **uid**;  **private** Double **money**;  **private** User **user**;   **public** User getUser() {  **return user**;  }  **public void** setUser(User user) {  **this**.**user** = user;  }   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** Integer getUid() {  **return uid**;  }   **public void** setUid(Integer uid) {  **this**.**uid** = uid;  }   **public** Double getMoney() {  **return money**;  }   **public void** setMoney(Double money) {  **this**.**money** = money;  }   @Override  **public** String toString() {  **return "Account{"** +  **"id="** + **id** +  **", uid="** + **uid** +  **", money="** + **money** +  **", user="** + **user** +  **'}'**;  } } |

#### 修改AccountMapper接口中的方法

|  |
| --- |
| List<Account> queryAccountList(); |

注意：此种方式，将返回值改 为了Account类型。因为Account类中包含了一个User类的对象，它可以封装账户所对应的用户信息。

#### 重新定义AccountMapper.xml文件

|  |
| --- |
| <**select id="queryAccountList" resultMap="accountResultMap"**>  select a.*\**,u.username,u.address from account a left join user u on a.uid=u.id; </**select**>  <**resultMap id="accountResultMap" type="account"**>  *<!--账户信息-->* <**id column="id" property="id"**/>  <**result column="uid" property="uid"**/>  <**result column="money" property="money"**/>  *<!--用户信息-->* <**association property="user" javaType="User"**>  <**id column="uid" property="id"**/>  <**result column="username" property="username"**/>  <**result column="address" property="address"**/>  </**association**> </**resultMap**> |

association部分定义了账户关联的用户信息。表示关联查询结果

property=*"user"*：关联查询的结果存储在Account对象的哪个属性。

javaType=*"User"*：指定关联查询的结果对象类型。此处可以使用别名，也可以使用全限定名。

<id/>用于定义主键的映射关系，如果是多主键可以配置多个<id/>

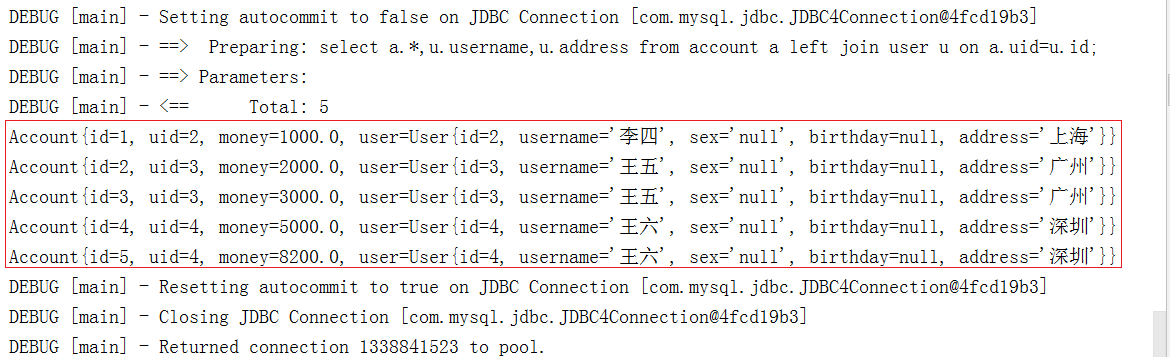
<result/>用于定义普通字段的映射关系

column属性指定查询列名，property指定映射的属性名

#### 测试方法

|  |
| --- |
| @Test **public void** testQueryAccountList() **throws** IOException {  InputStream inputStream = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  SqlSession sqlSession = sqlSessionFactory.openSession();  AccountMapper accountMapper = sqlSession.getMapper(AccountMapper.**class**);  List<Account> list = accountMapper.queryAccountList();  **for** (**int** i = 0; i < list.size(); i++) {  Account account = list.get(i);  System.***out***.println(account);  }  sqlSession.close(); } |

#### 效果



## 一对多查询

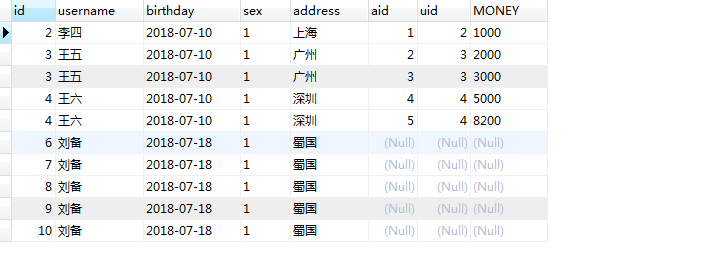
需求：查询所有用户信息及用户关联的账户信息。

分析：用户信息和他的账户信息为一对多关系，并且查询过程中如果用户没有账户信息，此时也要将用户信息查询出来。

### 编写SQL语句

|  |
| --- |
| SELECT  u.\*, a.id aid,  a.uid,  a.MONEY  FROM  USER u  LEFT JOIN account a ON u.id = a.UID |

测试该SQL语句在MySQL客户端工具的查询结果如下：



#### 改写User类加入List<Account>

为了能够让查询的User信息中，带有他的个人多个账户信息，我们就需要在User类中添加一个集合，用于存放他的多个账户信息，这样他们之间的关联关系就保存了。

|  |
| --- |
| **package** com.itheima.mybatis.pojo;  **import** java.util.Date; **import** java.util.List;  **public class** User {  **private** Integer **id**;  **private** String **username**;  **private** String **sex**;  **private** Date **birthday**;  **private** String **address**;   **private** List<Account> **accountList**;   **public** List<Account> getAccountList() {  **return accountList**;  }   **public void** setAccountList(List<Account> accountList) {  **this**.**accountList** = accountList;  }   **public** Integer getId() {  **return id**;  }   **public void** setId(Integer id) {  **this**.**id** = id;  }   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** String getSex() {  **return sex**;  }   **public void** setSex(String sex) {  **this**.**sex** = sex;  }   **public** Date getBirthday() {  **return birthday**;  }   **public void** setBirthday(Date birthday) {  **this**.**birthday** = birthday;  }   **public** String getAddress() {  **return address**;  }   **public void** setAddress(String address) {  **this**.**address** = address;  }   @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **", username='"** + **username** + **'\''** +  **", sex='"** + **sex** + **'\''** +  **", birthday="** + **birthday** +  **", address='"** + **address** + **'\''** +  **", accountList="** + **accountList** +  **'}'**;  } } |

### UserMapper接口中加入查询方法

在UserMapper接口中加入查询方法

|  |
| --- |
| List<User> queryUserAccounts(); |

### 修改UserMapper.xml映射文件

|  |
| --- |
| <**select id="queryUserAccounts" resultMap="userAccountsResultMap"**>  SELECT  u.*\**, a.id aid,  a.uid,  a.MONEY  FROM  USER u  LEFT JOIN account a ON u.id = a.UID </**select**>  <**resultMap id="userAccountsResultMap" type="user"**>  *<!--用户信息-->* <**id column="id" property="id"**/>  <**result column="username" property="username"**/>  <**result column="sex" property="sex"**/>  <**result column="birthday" property="birthday"**/>  <**result column="address" property="address"**/>  *<!--账户信息-->* <**collection property="accountList" ofType="account"**>  <**id column="aid" property="id"**/>  <**result column="uid" property="uid"**/>  <**result column="money" property="money"**/>  </**collection**> </**resultMap**> |

collection部分定义了用户关联的账户信息。表示关联查询结果集

property=*"accountList"*：关联查询的结果集存储在User对象的上哪个属性。

ofType=*"account"*：指定关联查询的结果集中的对象类型即List中的对象类型。此处可以使用别名，也可以使用全限定名。

<id/>及<result/>的意义同一对一查询。

### 测试方法

|  |
| --- |
| @Test **public void** testQueryUserAccounts() **throws** IOException {  InputStream inputStream = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  SqlSession sqlSession = sqlSessionFactory.openSession();  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  List<User> list = userMapper.queryUserAccounts();  **for** (**int** i = 0; i < list.size(); i++) {  User user1 = list.get(i);  System.***out***.println(user1);  }  sqlSession.close(); } |

### 效果

## Mybatis维护多对多关系

### 多对多关系分析

通过前面的学习，我们使用Mybatis实现一对多关系的维护。多对多关系其实我们看成是双向的一对多关系。

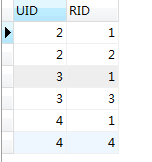
用户与角色的多对多关系模型如下：



在MySQL 数据库中添加角色表，用户角色的中间表。

角色表

  
用户角色中间表



### 实现Role到User的一对多

需求：实现查询所有角色并且加载它所分配的用户信息。

分析：查询角色我们需要用到Role 表，但角色分配的用户的信息我们并不能直接找到用户信息，而是要通过中间表(USER\_ROLE 表)才能关联到用户信息。

下面是实现的SQL 语句：

|  |
| --- |
| SELECT  \*  FROM  role r  LEFT JOIN user\_role ur ON r.R\_ID = ur.RID  LEFT JOIN USER u ON ur.UID = u.id  ORDER BY  username |

#### 编写角色实体类

我们加载的信息中不仅有角色信息，同时还要加载关联加载具有该角色的用户信息。因为一个角色可以分配给多个用户，所以我们可以考虑在Role类中加入一个List<User>的属性，用于存放这个角色分配给了哪些用户对象。

具体的角色实体类（Role类）如下：

|  |
| --- |
| **package** com.itheima.mybatis.pojo;  **import** java.util.List;  **public class** Role {  **private** Integer **roleId**;  **private** String **roleName**;  **private** String **roleDesc**;   **private** List<User> **userList**;   **public** List<User> getUserList() {  **return userList**;  }   **public void** setUserList(List<User> userList) {  **this**.**userList** = userList;  }   **public** Integer getRoleId() {  **return roleId**;  }   **public void** setRoleId(Integer roleId) {  **this**.**roleId** = roleId;  }   **public** String getRoleName() {  **return roleName**;  }   **public void** setRoleName(String roleName) {  **this**.**roleName** = roleName;  }   **public** String getRoleDesc() {  **return roleDesc**;  }   **public void** setRoleDesc(String roleDesc) {  **this**.**roleDesc** = roleDesc;  }   @Override  **public** String toString() {  **return "Role{"** +  **"roleId="** + **roleId** +  **", roleName='"** + **roleName** + **'\''** +  **", roleDesc='"** + **roleDesc** + **'\''** +  **", userList="** + **userList** +  **'}'**;  } } |

其中Role类中userList集合就是用于存在该角色分配给的用户列表。

#### 编写RoleMapper接口

在RoleDao接口中添加一个用于查询所有角色并关联查询出它所分配的用户列表的方法。

|  |
| --- |
| **package** com.itheima.mybatis.mapper;  **import** com.itheima.mybatis.pojo.Role;  **import** java.util.List;  **public interface** RoleMapper {  List<Role> queryRoleUsers(); } |

#### 编写RoleMapper.xml映射文件

添加RoleMapper.xml文件，并在SqlMapConfig.xml文件中加载该映射文件。

在RoleMapper.xml文件中添加一个resultMap用于加载角色对象及它所关联的用户信息。

|  |
| --- |
| *<?***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"*>*** *<!--根标签：映射的意思，sql语句写在标签内-->* <**mapper namespace="com.itheima.mybatis.mapper.RoleMapper"**>  <**select id="queryRoleUsers" resultMap="roleResultMap"**>  select *\** from role r left join user\_role ur on r.R\_ID=ur.RID left join user u on ur.UID=u.id order by username  </**select**>    <**resultMap id="roleResultMap" type="role"**>  *<!--role-->* <**id column="r\_id" property="roleId"**/>  <**result column="r\_name" property="roleName"**/>  <**result column="r\_desc" property="roleDesc"**/>  *<!--user-->* <**collection property="userList" ofType="user"**>  <**id column="id" property="id"**/>  <**result column="username" property="username"**/>  <**result column="sex" property="sex"**/>  <**result column="birthday" property="birthday"**/>  <**result column="address" property="address"**/>  </**collection**>  </**resultMap**> </**mapper**> |

#### 测试方法

添加TestRoleMapper类，并加入测试方法，测试方法如下：

|  |
| --- |
| @Test **public void** testQueryRoleList() **throws** IOException {  InputStream inputStream = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  SqlSession sqlSession = sqlSessionFactory.openSession();  RoleMapper roleMapper = sqlSession.getMapper(RoleMapper.**class**);  List<Role> list = roleMapper.queryRoleUsers();  **for** (**int** i = 0; i < list.size(); i++) {  Role role = list.get(i);  System.***out***.println(role);  }  sqlSession.close(); } |

#### 效果

### 实现User到Role的一对多

从User出发，我们也可以发现一个用户可以具有多个角色，这样用户到角色的关系也还是一对多关系。这样我们就可以认为User与Role的多对多关系，可以被拆解成两个一对多关系来实现。

#### 作业：实现User到Role的一对多查询

需求：实现查询所有用户信息并关联查询出每个用户的角色列表。

# 总结

本次课程主要是不断深入学习Mybatis的一些知识点，比如我们通过学习能够更好的掌握Mybatis连接池技术、事务、动态Sql、复杂结果集的封装，多表的关联查询。