**Mybatis第三天**

上海千锋 文威

课程目标：

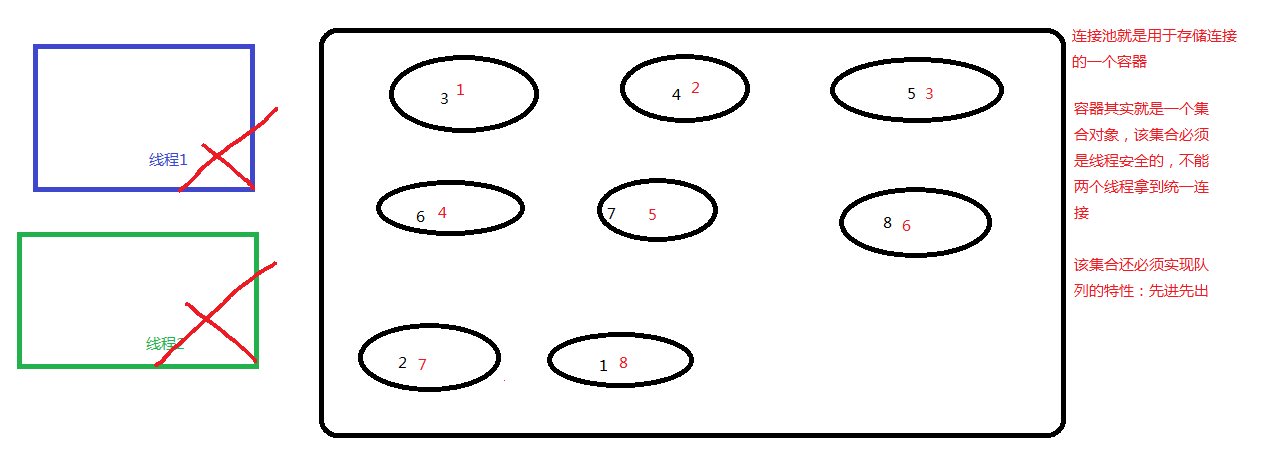
* 理解Mybatis连接池与事务控制
* Mybatis动态sql**(重点)**
* Mybatis多表查询之一对多**(重点)**
* Mybatis多表查询之多对多**(重点)**

## 1.1 Mybatis连接池与事务深入

### 1.1.1 什么是连接池

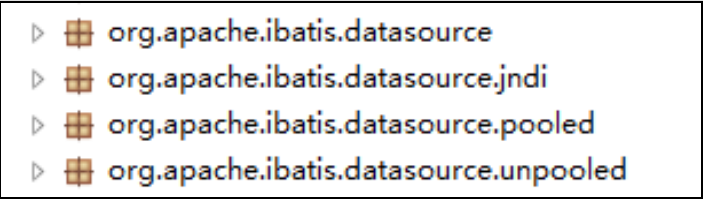
我们在前面的WEB课程中也学习过类似的连接池技术，而在 Mybatis 中也有连接池技术，但是它采用的是自己的连接池技术。在Mybatis的SqlMapConfig.xml 配置文件中，通过来实现 Mybatis中连接池的配置。**用连接池的目的主要是为了减少我们获取连接所消耗的时间。**

那么什么是连接池呢?我们通过下面一幅图来描述什么是连接池。



### 1.1.2 Mybatis中的连接池

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：

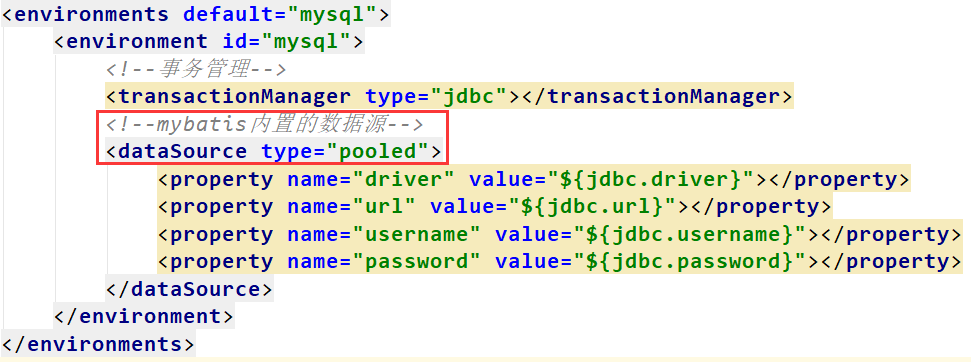
* UNPOOLED 不使用连接池的数据源 每次连接数据库都要创建新的数据库连接对象。
* POOLED 使用连接池的数据源

采用传统的javax.sql.DataSource规范中的连接池，mybatis有针对规范的实现。

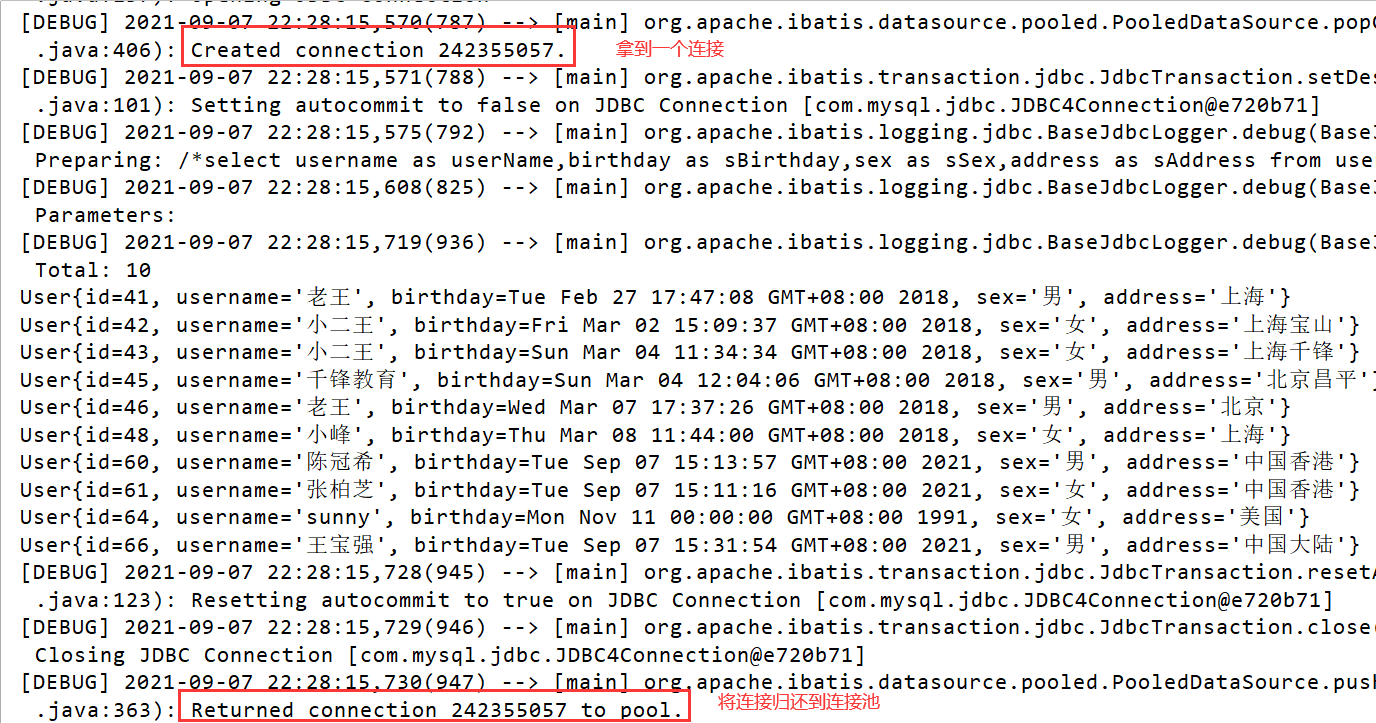
* JNDI 使用 JNDI 实现的数据源

采用服务器的JNDI技术实现，来获取DataSource对象。如果工程不是web工程是不能使用的。我们课程使用的是tomcat服务器，采用的是dbcp连接池。

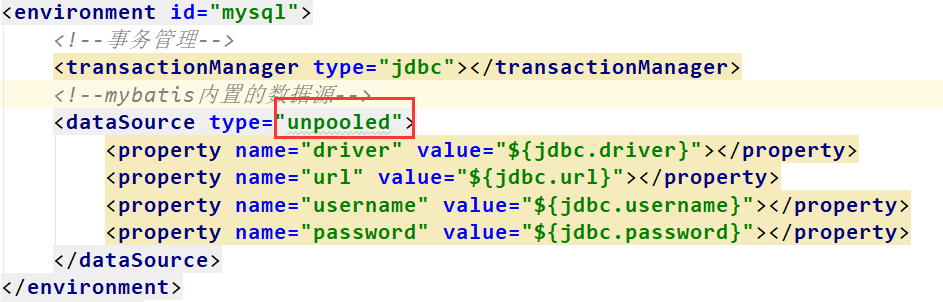
下面我们来看看，我们在mybatis案例中配置数据源的位置:



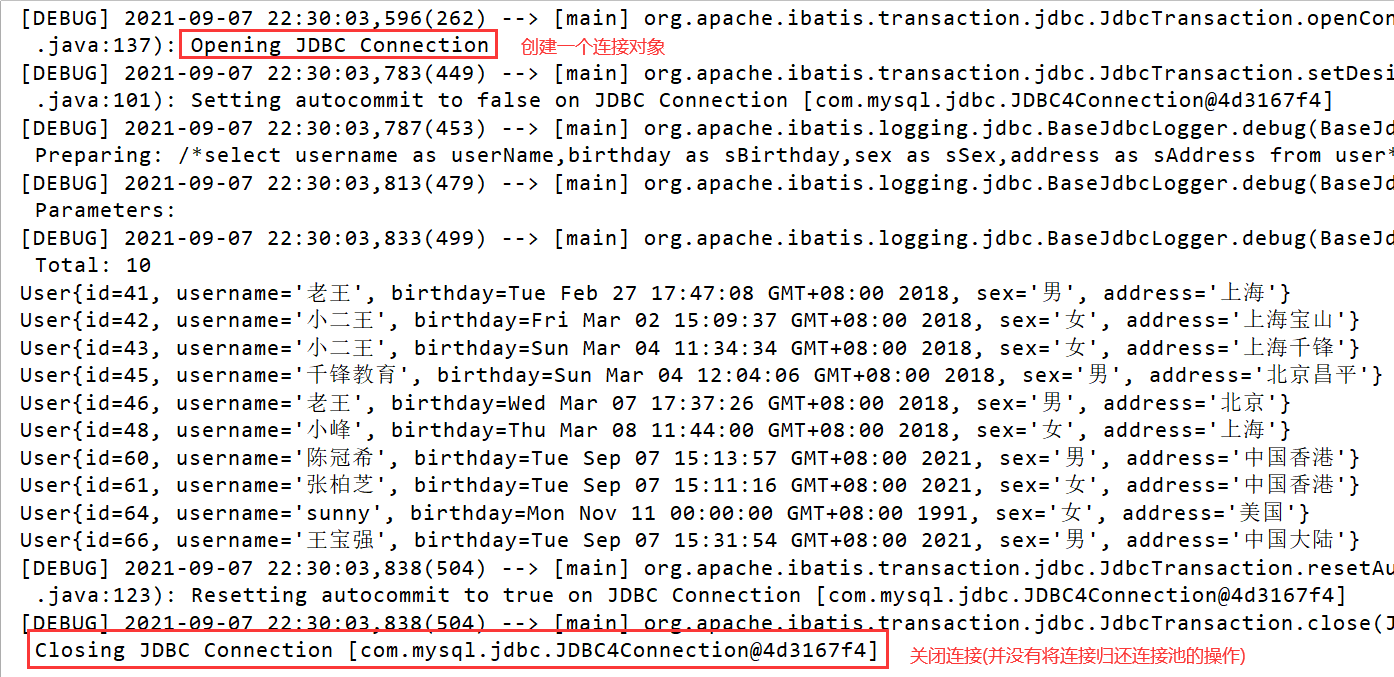
我们现在使用数据源的情况下，执行findAll方法,然后查看控制台信息:



我们再使用unpooled类型的连接:



再看控制台效果:



那么他们的底层源码到底是怎么描述的呢?

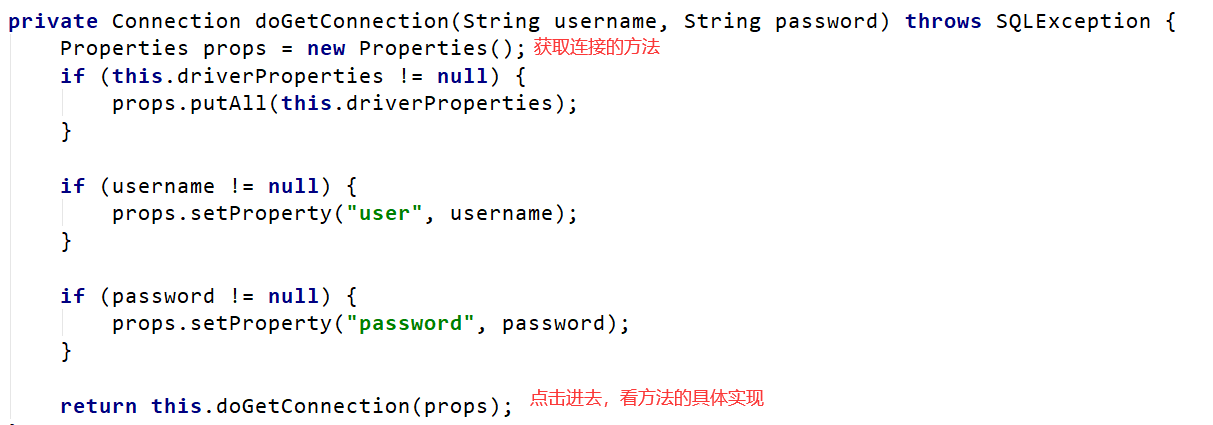
**其实描述POOLED连接类池类型的类是PooledDataSource。**

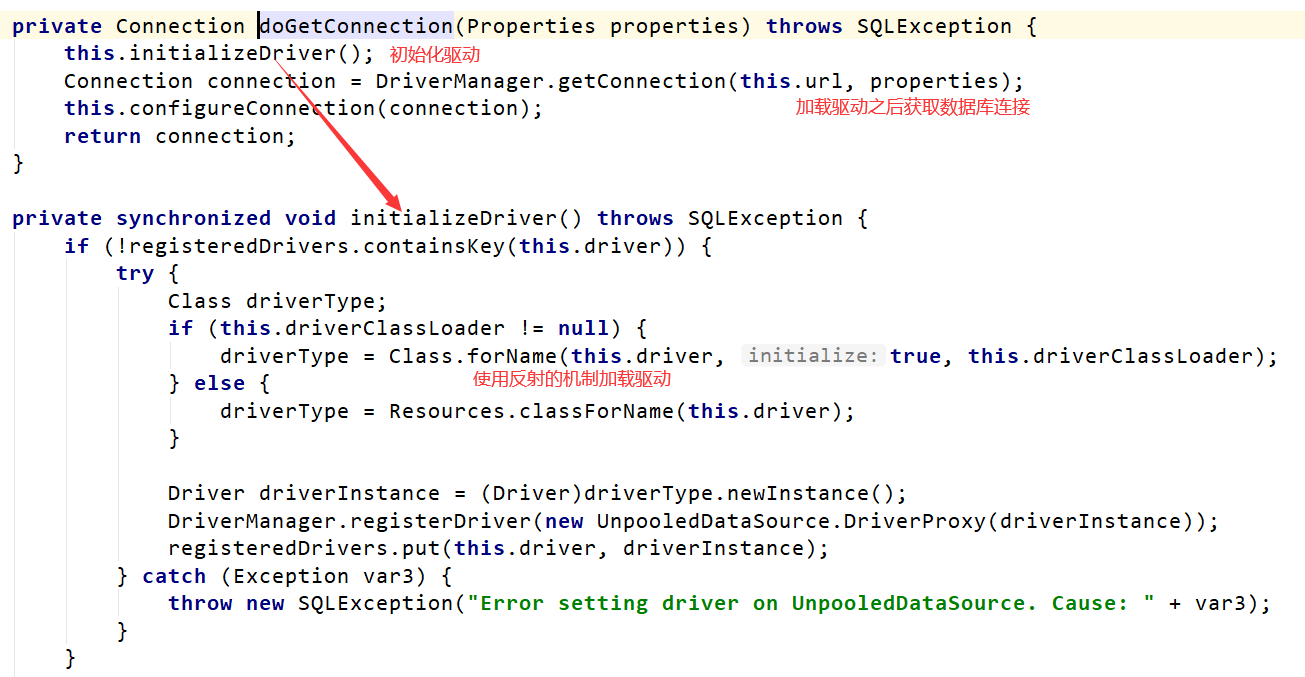
**描述UNPOOLED连接类型的类是UnpooledDataSource。**

**这两个类都实现了DataSource接口**

我们打开IDEA，使用Ctrl+N的快捷键打开这两个类:

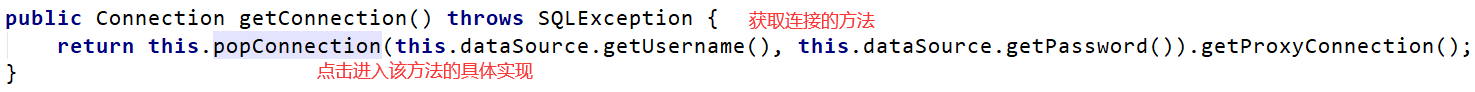
* 我们先查看UnpooledDataSource的获取连接的源码。

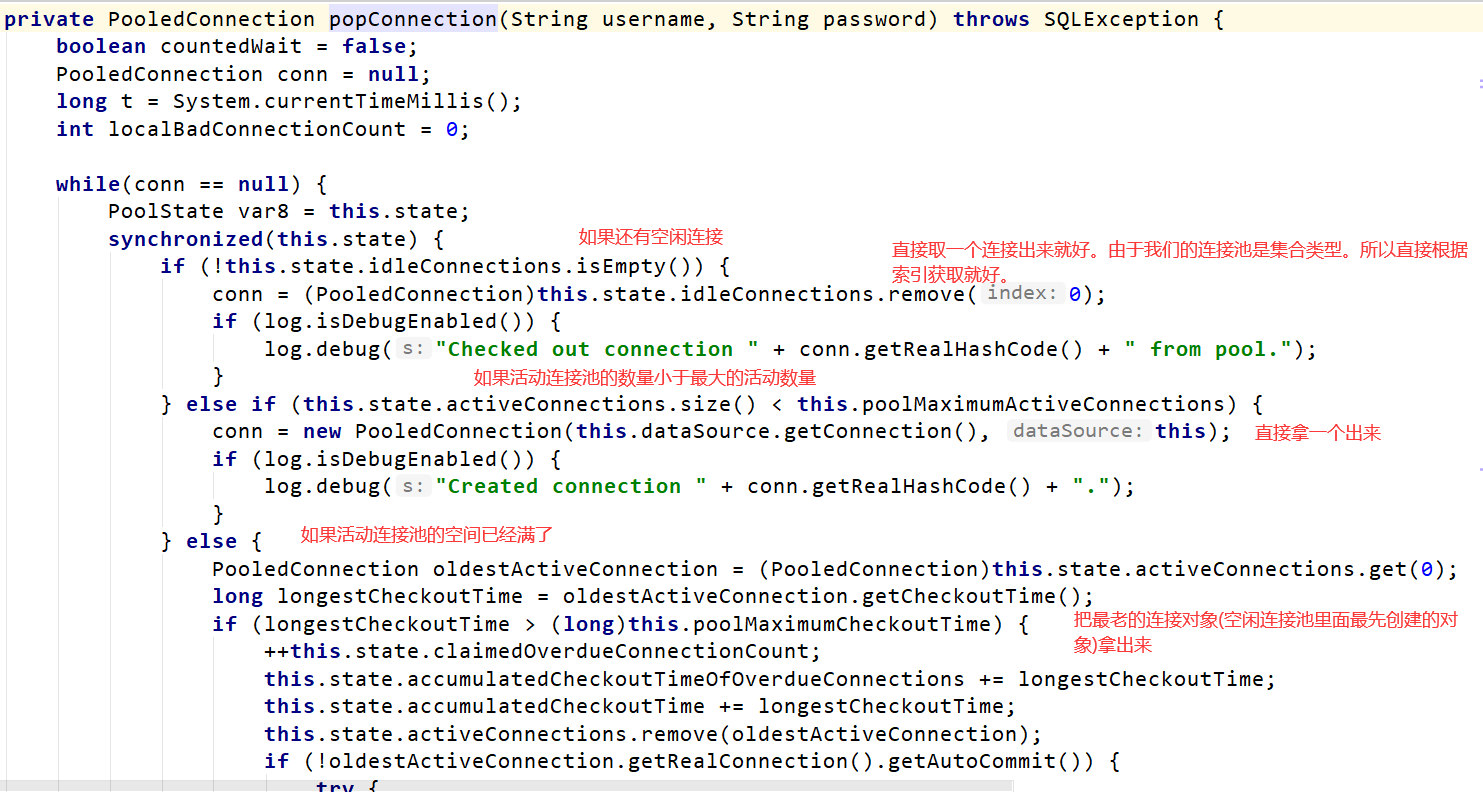




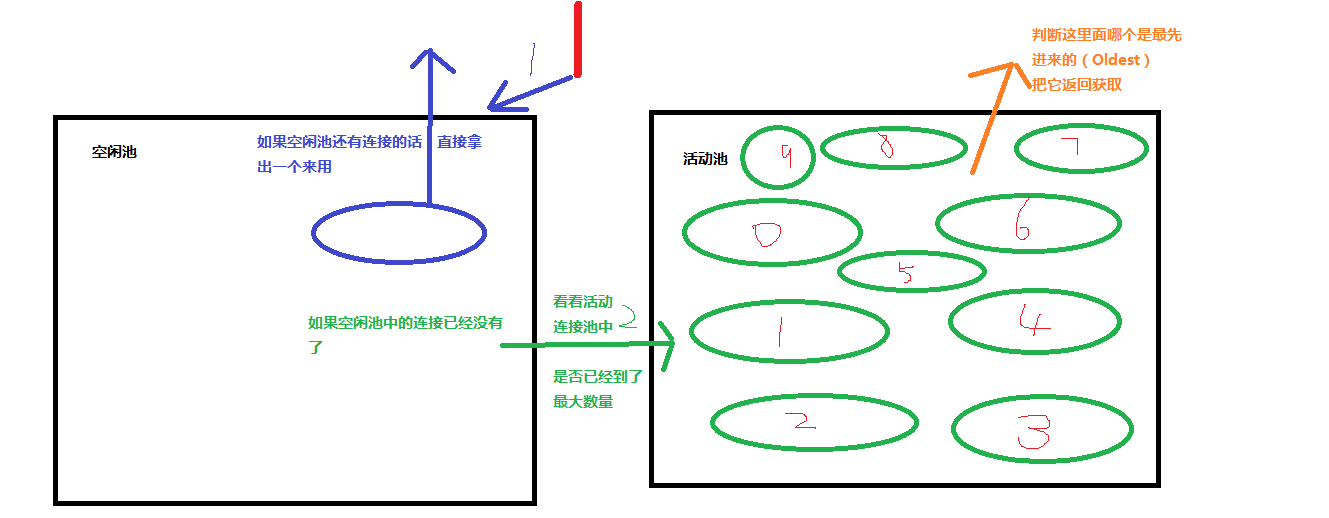
通过这部分源码，我们发现UnpooledDataSource获取连接是传统的jdbc获取连接的方式。

* 我们再查看PooledDataSource





下面我们通过一个图来描述mybatis的连接池



### 1.1.3 Mybatis的事务控制

讲到事务，首先想到的就是:

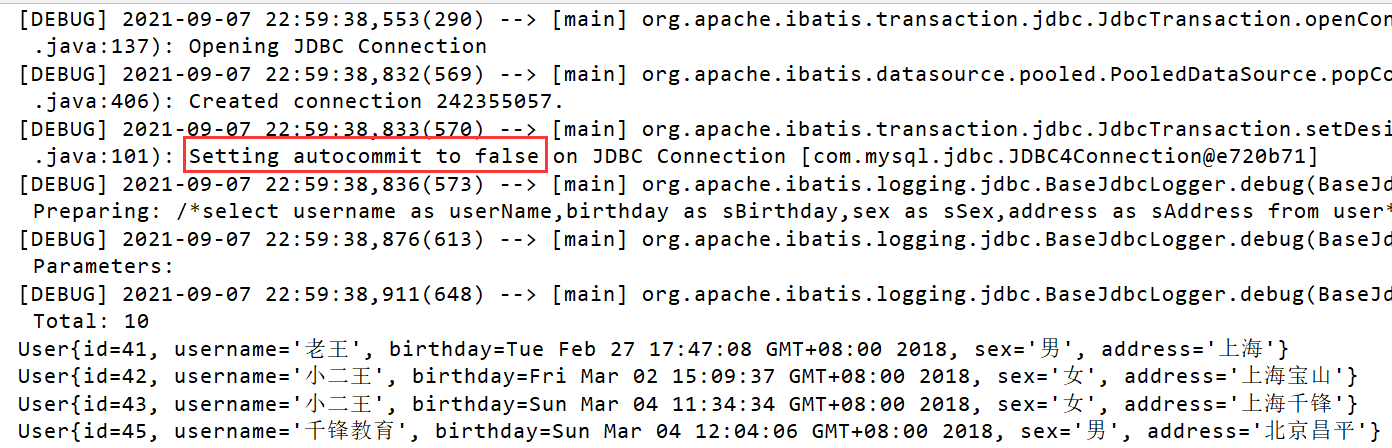
什么是事务?

事务的几大特性?

不考虑隔离性会造成的问题?

事务的隔离级别?

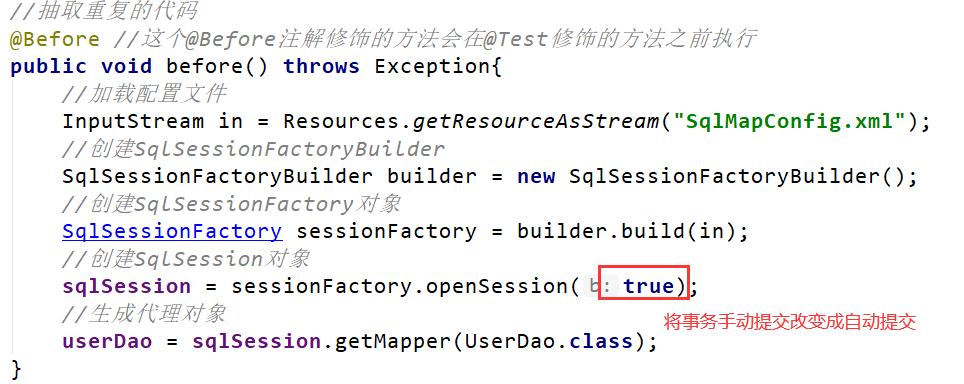
我们还是运行findAll方法，来观察控制台输出的日志信息。

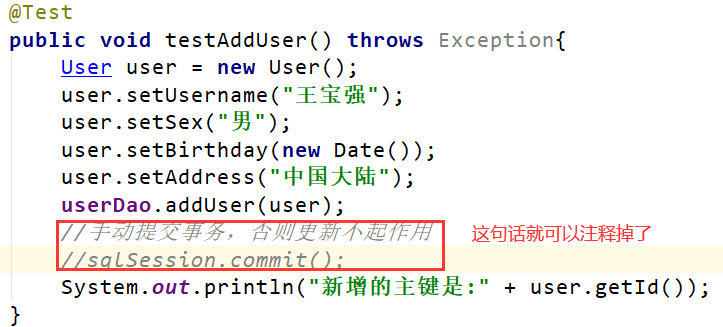


在JDBC 中我们可以通过手动方式将事务的提交改为手动方式，通过 setAutoCommit()方法就可以调整。我们的 Mybatis 框架因为是对 JDBC 的封装，所以 Mybatis 框架的事务控制方式，本身也是用 JDBC 的setAutoCommit()方法来设置事务提交方式的。

我们观察红色部分的信息，加上我们之前的 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**

****

****

## 1.2 Mybatis的动态sql

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。

### 1.2.1 动态sql之<if>标签

我们根据实体类的不同取值，使用不同的SQL语句来进行查询。比如在id如果不为空时可以根据 id查询，如果username不空时还要加入用户名作为条件。这种情况在我们的多条件组合查询中经常会碰到。

* 创建接口

|  |
| --- |
| *//复杂条件查询* **public** List<User> findByUser(User user); |

* 增加接口方法的配置

|  |
| --- |
| <**select id="findByUser" resultType="user" parameterType="user"**>  select *\** from user where 1=1  <**if test="username!=null and username != ''"**>  and username=#{username}  </**if**>  <**if test="birthday!=null"**>  and birthday=#{birthday}  </**if**>  <**if test="sex!=null"**>  and sex=#{sex}  </**if**>  <**if test="address!=null"**>  and address=#{address}  </**if**> </**select**> |

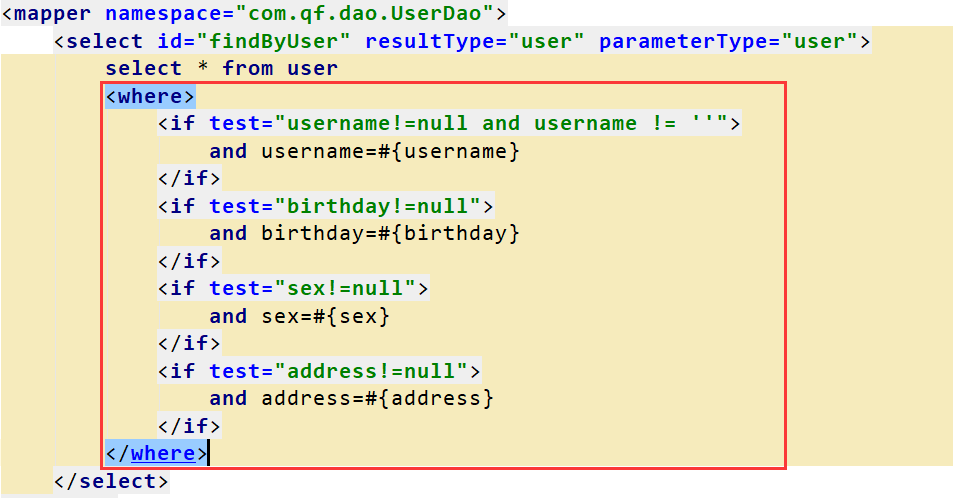
* 测试

|  |
| --- |
| **public class** TestUser {   UserDao **userDao**;  SqlSession **sqlSession**;   *//抽取重复的代码* @Before *//这个@Before注解修饰的方法会在@Test修饰的方法之前执行* **public void** before() **throws** Exception{  *//加载配置文件* InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  *//创建SqlSessionFactoryBuilder* SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  *//创建SqlSessionFactory对象* SqlSessionFactory sessionFactory = builder.build(in);  *//创建SqlSession对象* **sqlSession** = sessionFactory.openSession(**true**);  *//生成代理对象* **userDao** = **sqlSession**.getMapper(UserDao.**class**);  }   @After *//@After注解修饰的方法在@Test注解修饰的方法之后执行* **public void** after(){  **sqlSession**.close();  }   **@Test  public void testFindByUser(){  User user = new User();  user.setSex("男");  user.setAddress("上海");  List<User> userList = userDao.findByUser(user);  for(User u : userList){  System.*out*.println(u);  }  }** |

### 1.2.2 动态sql之<where>标签

为了简化上面where 1=1的条件拼装，我们可以采用标签来简化开发。

**使用where标签将if标签代码块包起来，将1=1条件去掉。**



测试结果和上面的那种一样的。

### 1.2.3 动态sql之<foreach>标签

现在有一个需求，就是根据id进行批量删除，怎么实现?其实实现思路和我们上一个章节的批量查询是一样的。

* 定义接口

|  |
| --- |
| *//批量删除* **public void** deleteUserByIds(QueryVo queryVo); |

* 增加接口的配置文件

|  |
| --- |
| <**delete id="deleteUserByIds" parameterType="com.qf.pojo.QueryVo"**>  delete from user where id in  <**foreach collection="ids" item="id" open="(" close=")" separator=","**>  #{id}  </**foreach**> </**delete**> |

* 测试

|  |
| --- |
| @Test **public void** testDeleteUserByIds(){  List<Integer> list = **new** ArrayList();  list.add(61);  list.add(64);  list.add(67);  QueryVo queryVo = **new** QueryVo();  queryVo.setIds(list);  **userDao**.deleteUserByIds(queryVo); } |

### 1.2.4 动态sql之<set>标签

* 定义接口

|  |
| --- |
| **public void** updateUser(User user) |

* 增加配置

|  |
| --- |
| <**update id="updateByUser" parameterType="user"**>  update user  <**set**>  <**if test="username!=null and username != '' "**>  username=#{username},  </**if**>  <**if test="birthday!=null"**>  birthday=#{birthday},  </**if**>  <**if test="sex!=null and sex != '' "**>  sex=#{sex},  </**if**>  <**if test="address!=null and address != '' "**>  address=#{address},  </**if**>  </**set**>  where id=#{id} </**update**> |

* 测试

|  |
| --- |
| @Test **public void** testUpdateByUser(){  User user = **new** User();  user.setId(60);  user.setBirthday(**new** Date());  user.setAddress(**"加拿大"**);  **userDao**.updateByUser(user); } |

### 1.2.5 动态sql之动态添加

* 定义接口

|  |
| --- |
| **public void** addUser(User user); |

* 定义配置文件

|  |
| --- |
| <**sql id="key"**>  <**trim suffixOverrides=","**>  <**if test="username!=null and username != '' "**>  username,  </**if**>  <**if test="birthday!=null and username != '' "**>  birthday,  </**if**>  <**if test="sex!=null and username != '' "**>  sex,  </**if**>  <**if test="address!=null and username != '' "**>  address,  </**if**>  </**trim**> </**sql**>  <**sql id="value"**>  <**trim suffixOverrides=","**>  <**if test="username!=null and username != '' "**>  #{username},  </**if**>  <**if test="birthday!=null and username != '' "**>  #{birthday},  </**if**>  <**if test="sex!=null and username != '' "**>  #{sex},  </**if**>  <**if test="address!=null and username != '' "**>  #{address},  </**if**>  </**trim**> </**sql**> <**insert id="addUser" parameterType="user"**>  insert into user(<**include refid="key"**/>) values(<**include refid="value"**/>) </**insert**> |

* 定义测试类

|  |
| --- |
| @Test **public void** testAddUser(){  User user = **new** User();  user.setUsername(**"张卫健"**);  user.setAddress(**"香港"**);  **userDao**.addUser(user); } |

### 1.2.6 Mybatis的批处理

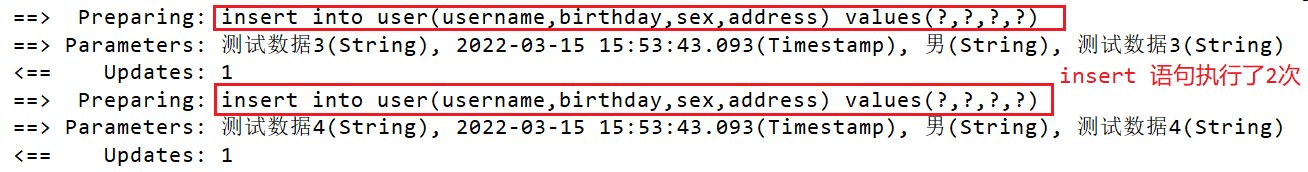
需求：批量新增一批数据。

解决问题的方案有两种:

第一种：将批量新增的数据 放在List集合里面，遍历集合循环新增数据即可。

|  |
| --- |
| @Test **public void** test04() **throws** Exception{  UserDao userDao = **sqlSession**.getMapper(UserDao.**class**);  List<User> userList = **new** ArrayList<User>();  User user1 = **new** User();  user1.setUsername(**"测试数据3"**);  user1.setBirthday(**new** Date());  user1.setSex(**"男"**);  user1.setAddress(**"测试数据3"**);  User user2 = **new** User();  user2.setUsername(**"测试数据4"**);  user2.setBirthday(**new** Date());  user2.setSex(**"男"**);  user2.setAddress(**"测试数据4"**);  userList.add(user1);  userList.add(user2);  *//批量新增* **for**(User user : userList){  userDao.addUser(user);  }  **sqlSession**.commit(); } |

控制台执行结果如下:



由于insert语句多次执行，造成执行效率很低

第二种：开启批处理指令，使用批处理的方式进行数据的新增。

如何开启批处理?

①：在mybatis的核心配置文件上开启

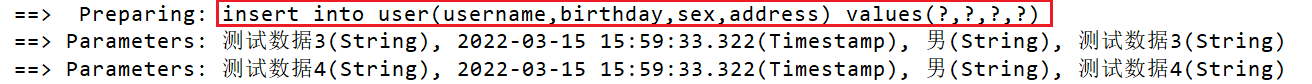
|  |
| --- |
| <**settings**>  <**setting name="defaultExecutorType" value="BATCH"**/> </**settings**> |

②：在创建SqlSession对象的时候，指定开启批处理

|  |
| --- |
| **sqlSession** = sessionFactory.openSession(**ExecutorType.*BATCH***,**false**); |

注意：使用批处理的话，一定要将事务的自动提交关闭。

通用是上面的test4方法执行，控制台输出日志信息不一样:

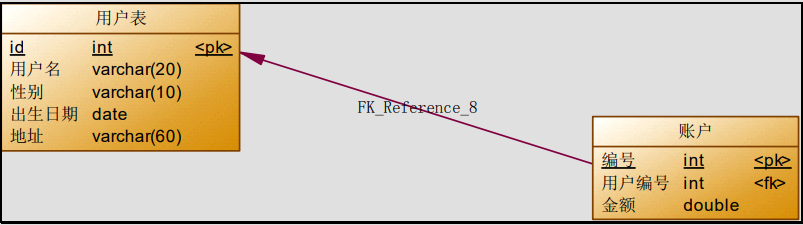


由此可见 insert语句只执行了一次。底层执行的语句是:

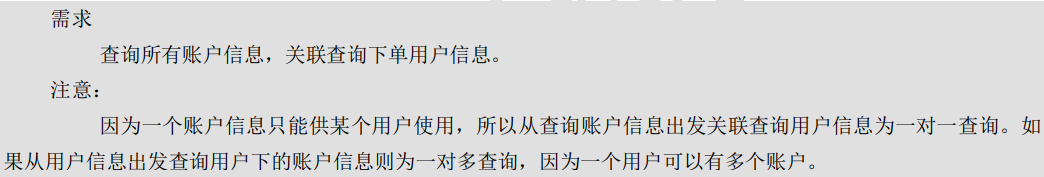
INSERT INTO USER(username,birthday,sex,address) VALUES(?,?,?,?),(?,?,?,?)

## 1.3 Mybatis查询之多表查询

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：



### 1.3.1 一对一查询(多对一)



**方式1：**

* 定义账户信息的实体类

|  |
| --- |
| **public class** Account **implements** Serializable {   **private** Integer **id**;  **private** Integer **uid**;  **private** Double **money**;   **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

|  |
| --- |
| SELECT  account.\*,  user.username,  user.address  FROM  account,  USER  WHERE account.uid = user.id |

* 定义AccountUser类

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

|  |
| --- |
| **public class** AccountUser **extends** Account **implements** Serializable {  **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** + **'\''** +  **'}'**;  } } |

* 定义账户的持久层Dao接口

|  |
| --- |
| **public interface** AccountDao {  List<AccountUser> findAll(); } |

* 定义AccountDao.xml文件中的查询配置信息

|  |
| --- |
| <**mapper namespace="com.qf.dao.AccountDao"**>  *<!-- 配置查询所有操作-->* <**select id="findAll" resultType="accountuser"**>  select a.*\**,u.username,u.address from account a,user u where a.uid =u.id;  </**select**> </**mapper**> |

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。

* 创建AccountTest测试类

|  |
| --- |
| @Test **public void** testFindAll(){  List<AccountUser> accountList = **accountDao**.findAll();  **for**(AccountUser ac : accountList){  System.***out***.println(ac);  } } |

**方式2:**

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。

* 在Account类中加入User类的对象作为Account类的一个属性。

|  |
| --- |
| **public class** Account **implements** Serializable {   **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** +  **'}'**;  } } |

* 定义接口

|  |
| --- |
| **public interface** AccountDao {  List<Account> findAll(); } |

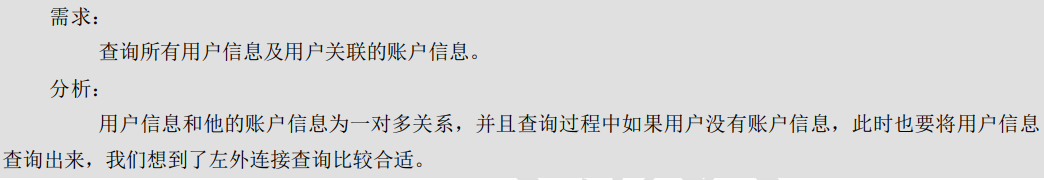
* 定义配置文件

|  |
| --- |
| <**mapper namespace="com.qf.dao.AccountDao"**>  *<!-- 建立对应关系 -->* <**resultMap type="account" id="accountMap"**>  <**id column="aid" property="id"**/>  <**result column="uid" property="uid"**/>  <**result column="money" property="money"**/>  *<!-- 它是用于指定从表方的引用实体属性的 -->* <**association property="user" javaType="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"**/>  </**association**>  </**resultMap**>  <**select id="findAll" resultMap="accountMap"**>  select u.*\**,a.id as aid,a.uid,a.money from account a,user u where a.uid =u.id;  </**select**> </**mapper**> |

* 测试

|  |
| --- |
| @Test **public void** testFindAll(){  List<Account> accountList = **accountDao**.findAll();  **for**(Account ac : accountList){  System.***out***.println(ac);  } } |

### 1.3.2 一对多查询



* 编写sql语句

|  |
| --- |
| SELECT  u.\*, acc.id id,  acc.uid,  acc.money  FROM  USER u  LEFT JOIN account acc ON u.id = acc.uid |

查询出来的结果:



* User类加入List<Account>

|  |
| --- |
| **public class** User {  **private** Integer **id**;  **private** String **username**;  **private** Date **birthday**;  **private** String **sex**;  **private** String **address**;  **private** List<Account> **accounts**;   **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** Date getBirthday() {  **return birthday**;  }   **public void** setBirthday(Date birthday) {  **this**.**birthday** = birthday;  }   **public** String getSex() {  **return sex**;  }   **public void** setSex(String sex) {  **this**.**sex** = sex;  }   **public** String getAddress() {  **return address**;  }   **public void** setAddress(String address) {  **this**.**address** = address;  }   **public** List<Account> getAccounts() {  **return accounts**;  }   **public void** setAccounts(List<Account> accounts) {  **this**.**accounts** = accounts;  }   @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **", username='"** + **username** + **'\''** +  **", birthday="** + **birthday** +  **", sex='"** + **sex** + **'\''** +  **", address='"** + **address** + **'\''** +  **", accounts="** + **accounts** +  **'}'**;  } } |

* 用户持久层UserDao接口中加入查询方法

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

* 定义配置文件

|  |
| --- |
| <**mapper namespace="com.qf.dao.UserDao"**>  <**resultMap type="user" id="userMap"**>  <**id column="id" property="id"**></**id**>  <**result column="username" property="username"**/>  <**result column="address" property="address"**/>  <**result column="sex" property="sex"**/>  <**result column="birthday" property="birthday"**/>  *<!-- collection 是用于建立一对多中集合属性的对应关系  ofType 用于指定集合元素的数据类型  -->* <**collection property="accounts" ofType="account"**>  <**id column="aid" property="id"**/>  <**result column="uid" property="uid"**/>  <**result column="money" property="money"**/>  </**collection**>  </**resultMap**>  *<!-- 配置查询所有操作 -->* <**select id="findAll" resultMap="userMap"**>  select u.*\**,a.id as aid ,a.uid,a.money from user u left outer join account  a on u.id =a.uid  </**select**> </**mapper**> |

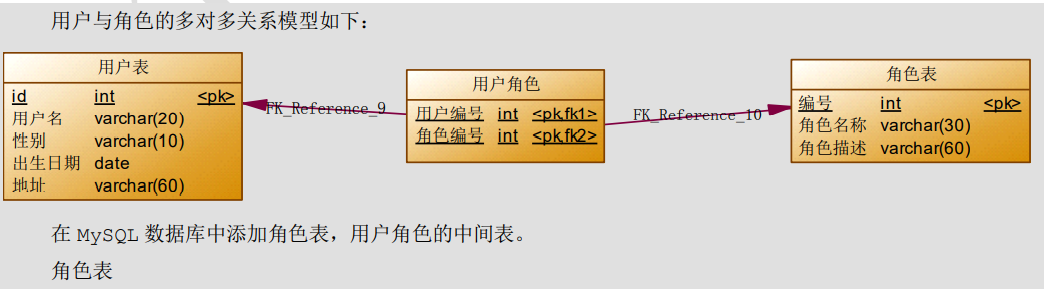
* 测试

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

### 1.3.3 多对多查询

需求：实现Role到User多对多

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。



* Sql编写

|  |
| --- |
| SELECT  r.\*,u.id uid,  u.username username,  u.birthday birthday,  u.sex sex,  u.address address  FROM  role r  INNER JOIN  user\_role ur  ON ( r.id = ur.rid)  INNER JOIN  `user` u  ON (ur.uid = u.id); |

* 编写角色实体类

|  |
| --- |
| **public class** Role {  **private** Integer **roleId**;  **private** String **roleName**;  **private** String **roleDesc**;  *//多对多的关系映射：一个角色可以赋予多个用户* **private** List<User> **users**;   **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;  }   **public** List<User> getUsers() {  **return users**;  }   **public void** setUsers(List<User> users) {  **this**.**users** = users;  }   @Override  **public** String toString() {  **return "Role{"** +  **"roleId="** + **roleId** +  **", roleName='"** + **roleName** + **'\''** +  **", roleDesc='"** + **roleDesc** + **'\''** +  **", users="** + **users** +  **'}'**;  } } |

* 编写接口

|  |
| --- |
| **public interface** RoleDao {  List<Role> findAll(); } |

* 编写配置文件

|  |
| --- |
| <**mapper namespace="com.qf.dao.RoleDao"**>  *<!--定义 role 表的 ResultMap-->* <**resultMap id="roleMap" type="role"**>  <**id property="roleId" column="rid"**></**id**>  <**result property="roleName" column="role\_name"**></**result**>  <**result property="roleDesc" column="role\_desc"**></**result**>  <**collection property="users" ofType="user"**>  <**id column="id" property="id"**></**id**>  <**result column="username" property="username"**></**result**>  <**result column="address" property="address"**></**result**>  <**result column="sex" property="sex"**></**result**>  <**result column="birthday" property="birthday"**></**result**>  </**collection**>  </**resultMap**>  *<!--查询所有-->* <**select id="findAll" resultMap="roleMap"**>  select u.*\**,r.id as rid,r.role\_name,r.role\_desc from role r  left outer join user\_role ur on r.id = ur.rid  left outer join user u on u.id = ur.uid  </**select**> </**mapper**> |

* 测试

|  |
| --- |
| **public class** TestRole {   RoleDao **roleDao**;  SqlSession **sqlSession**;   *//抽取重复的代码* @Before *//这个@Before注解修饰的方法会在@Test修饰的方法之前执行* **public void** before() **throws** Exception{  *//加载配置文件* InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  *//创建SqlSessionFactoryBuilder* SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  *//创建SqlSessionFactory对象* SqlSessionFactory sessionFactory = builder.build(in);  *//创建SqlSession对象* **sqlSession** = sessionFactory.openSession(**true**);  *//生成代理对象* **roleDao** = **sqlSession**.getMapper(RoleDao.**class**);  }   @After *//@After注解修饰的方法在@Test注解修饰的方法之后执行* **public void** after(){  **sqlSession**.close();  }   @Test  **public void** test(){  List<Role> roleList = **roleDao**.findAll();  **for**(Role role : roleList){  System.***out***.println(role);  }  } } |