**Mybatis第四天**

上海千锋 文威

* Mybatis延时加载策略(理解)
* Mybatis缓存(理解)
* Mybatis注解开发**(掌握)**

## 1.1 Mybatis延迟加载策略

问题: 在一对多中，当我们一个用户，它有100个账户。

在查询用户的时候，要不要把关联的账户查询出来?

在查询账户的时候，要不要把关联的用户查询出来?

答案：

在查询用户时，用户下的账户信息应该是什么时候用，什么时候查询。

在查询账户时，账户所属的用户信息应该是随着账户信息一起查询出来的。

什么是延迟加载:

在真正使用数据的时候才发起查询，不用的时候不查询。按需加载(延迟加载)

什么是立即加载:

不管用不用，只要一调用方法，立马查询出来。

总结:

一对多，多对多关系的情况下，通常我们都是采用按需加载(延迟加载)。

多对一，一对一关系的情况下，通常我们采用的都是立即加载。

### 1.1.1 使用 assocation 实现延迟加载

需求:

查询账户时，延时加载用户信息。

* 引入依赖、引入核心配置文件(省略)
* 创建实体

User.java

|  |
| --- |
| **public class** User {  **private** Integer **id**;  **private** String **username**;  **private** Date **birthday**;  **private** String **sex**;  **private** String **address**;  **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;  }   @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **", username='"** + **username** + **'\''** +  **", birthday="** + **birthday** +  **", sex='"** + **sex** + **'\''** +  **", address='"** + **address** + **'\''** +  **'}'**;  } } |

Account.java

|  |
| --- |
| **public class** Account **implements** Serializable {  **private** Integer **id**;  **private** Integer **uid**;  **private** Double **money**;  private 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;  }   **public** User getUser() {  **return user**;  }   **public void** setUser(User user) {  **this**.**user** = user;  }   @Override  **public** String toString() {  **return "Account{"** +  **"id="** + **id** +  **", uid="** + **uid** +  **", money="** + **money** +  **", user="** + **user** +  **'}'**;  } } |

* 创建AccountDao接口

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

* 创建AccountDao.xml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.AccountDao"**>  <**resultMap id="accountMap" type="account"**>  <**id column="id" property="id"**></**id**>  <**result column="uid" property="uid"**></**result**>  <**result column="money" property="money"**></**result**>  *<!--  select： 填写我们要调用的 select 映射的 id   column ： 填写我们要传递给 select 映射的参数（必须要写） 就是查询出来的account的uid属性*  *fetchType="lazy" 配置懒加载，只在当前方法里面有效，如果想要所有的方法都进行懒加载，需要进行全局懒加载的配置(后面有说到)  -->* <**association property="user"** select**="com.qf.dao.UserDao.findById" column="uid"**></**association**>  </**resultMap**>  <**select id="findAccount" resultMap="accountMap"**>  select *\** from account  </**select**>  </**mapper**> |

我们现在只是查询出来了account记录，但是对应的user信息并没有查询出来，如何查询user?我们需要根据查询出来的account里面的uid查询对应的user信息。

所以我们需要在UserDao里面定义一个根据uid(就是user表的主键)查询用户信息。

* 定义UserDao接口里面的方法

|  |
| --- |
| **public interface** UserDao {  *//根据主键id查询用户信息* **public** User findById(Integer id); } |

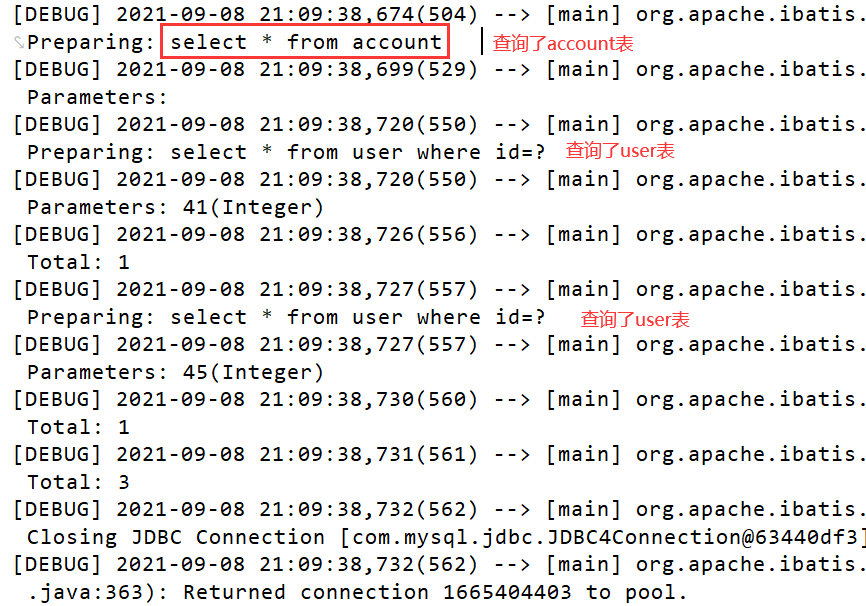
* 定义UserDao.xml配置

|  |
| --- |
| *<?***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="findById" resultType="user"**>  select *\** from user where id=#{id}  </**select**> </**mapper**> |

* 测试

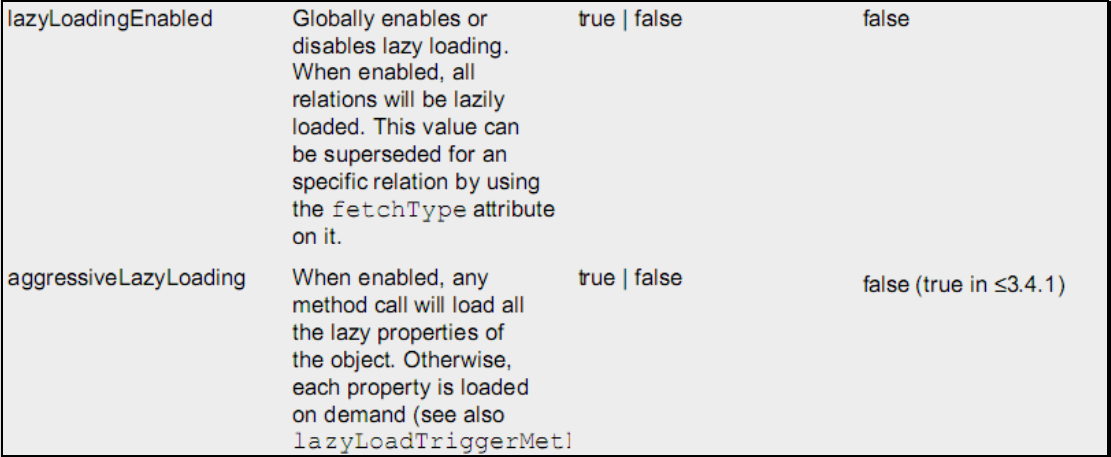
|  |
| --- |
| **public class** TestAccount {  SqlSession **sqlSession**;  AccountDao **accountDao**;  @Before  **public void** init() **throws** Exception{  InputStream in = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  SqlSessionFactoryBuilder builder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sessionFactory = builder.build(in);  **sqlSession** = sessionFactory.openSession(**true**);  **accountDao** = **sqlSession**.getMapper(AccountDao.**class**);  }   @After  **public void** after(){  **sqlSession**.close();  }   @Test  **public void** test(){  List<Account> accountList = **accountDao**.findAccount();  */\*for(Account account : accountList){  System.out.println(account);  }\*/* } } |

测试我们直接调用方法测试(把循环遍历的代码注释掉)。



通过以上观察我们发现，调用findAccount方法的时候，把account表和user表都查询出来了，并不是我们想象中的懒加载。究其原因是我们还没有配置Mybatis的延迟加载而已。

如何配置?  
进入Mybaits的官方文档，找到settings的说明信息：

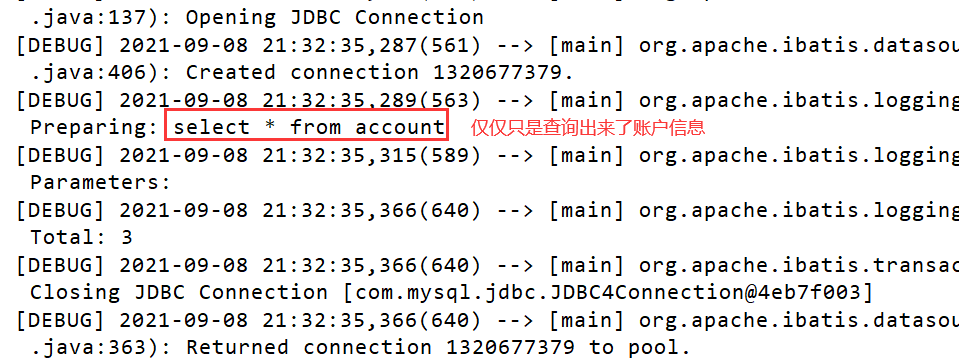


我们需要在Mybatis的配置文件SqlMapConfig.xml文件中添加延迟加载的配置。

|  |
| --- |
| <**settings**>  <**setting name="lazyLoadingEnabled" value="true"**/>  <**setting name="aggressiveLazyLoading" value="false"**/>//mybatis3.4.1版本后可以不写，默认为false。 </**settings**> |



此时再进行测试:



这样我们就实现了一对一(多对一)形式的懒加载。

### 1.1.2 使用Collection实现延迟加载

同样我们也可以在一对多关系配置的结点中配置延迟加载策略。<Collection> 节点中也有select属性，column属性。

需求:

完成加载用户对象时，查询该用户所拥有的账户信息。

* 定义实体类(User Account)

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

|  |
| --- |
| **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** +  **'}'**;  } } |

* 定义UserDao接口

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

* 定义UserDao.xml配置文件

|  |
| --- |
| *<?***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"**>   <**resultMap id="userMap" type="user"**>  <**id property="id" column="id"**></**id**>  <**result property="username" column="username"**></**result**>  <**result property="birthday" column="birthday"**></**result**>  <**result property="sex" column="sex"**></**result**>  <**result property="address" column="address"**></**result**>  <**collection property="accounts" select="com.qf.dao.AccountDao.findAccountById" column="id"**></**collection**>  </**resultMap**>  <**select id="findAllUser" resultMap="userMap"**>  select *\** from user  </**select**> </**mapper**> |

**Collection标签：主要用于加载关联的集合对象 。**

**select属性：**

**用于指定查询account列表的sql语句，所以填写的是该sql映射的id。**

**column 属性：**

**用于指定select 属性的 sql 语句的参数来源，上面的参数来自于user的id列，所以就写成id这一个字段名了。**

查询出用户后，我们还需要将对应的account查询出来。此时可以获取account的id值，那么我们可以根据account的id查询用户表user。

* 定义AccountDao接口

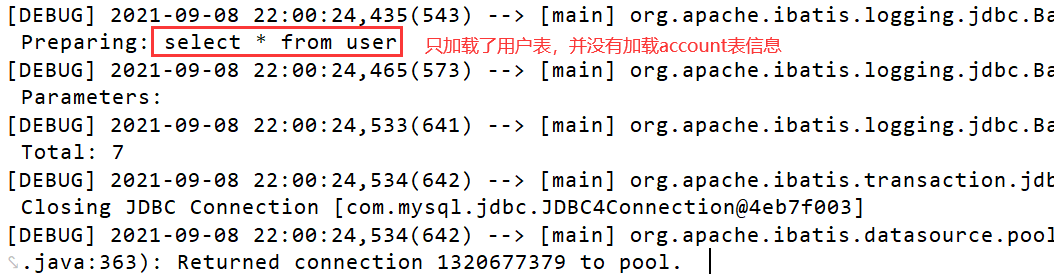
|  |
| --- |
| **public interface** AccountDao {  *//根据用户id查询账户信息* **public** Account findAccountById(Integer id); } |

* AccountDao.xml配置文件

|  |
| --- |
| *<?***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.AccountDao"**>   <**select id="findAccountById" resultType="account"**>  select *\** from account where uid = #{id}  </**select**> </**mapper**> |

* 配置延迟加载(和一对一延时加载的配置一样)
* 测试

|  |
| --- |
| @Test **public void** test(){  List<User> accountList = **userDao**.findAllUser();  */\* for(User user : accountList){  System.out.println(user);  }\*/* } |



## 1.2 Mybatis缓存

* 什么是缓存?

缓存就是存在内存中的数据。

* 为什么使用缓存?

减少和数据库交互的次数，提高执行效率。

* 什么样的数据能使用缓存，什么样的数据不适合使用缓存?
* 适用于缓存:

经常查询，且不经常改变的数据。

数据的正确与否对最终结果影响不大的。

* 不适用与缓存的:

经常改变的，数据的正确与否对结果影响很大的。

比如:商品的库存、银行的汇率、股市的股价等。

* Mybatis中的一级缓存和二级缓存:
* 一级缓存:

它指的是mybatis中的SqlSession对象的缓存。当我们执行完查询之后，查询的结果会同时存在在SqlSession为我们提供的一块区域中。该区域的结构是一个Map。

当我们再次查询同样的数据，mybatis会先去SqlSession中查询是否有，有的话直接拿出来使用。当SqlSession对象消失时，Mybatis的一级缓存也就消失了。

* 二级缓存：

它指的是Mybatis中SqlSessionFactory对象的缓存，由同一个SqlSessioFactory对象创建的，SqlSession共享其缓存。

### 1.2.1 证明一级缓存存在

一级缓存是SqlSession级别的缓存，只要SqlSession没有flush或close，它就存在。

我们以AccountDao为例:

#### 1.2.1.1 编写持久层接口

|  |
| --- |
| *//根据id查询用户信息* **public** User findUserById(Integer id); |

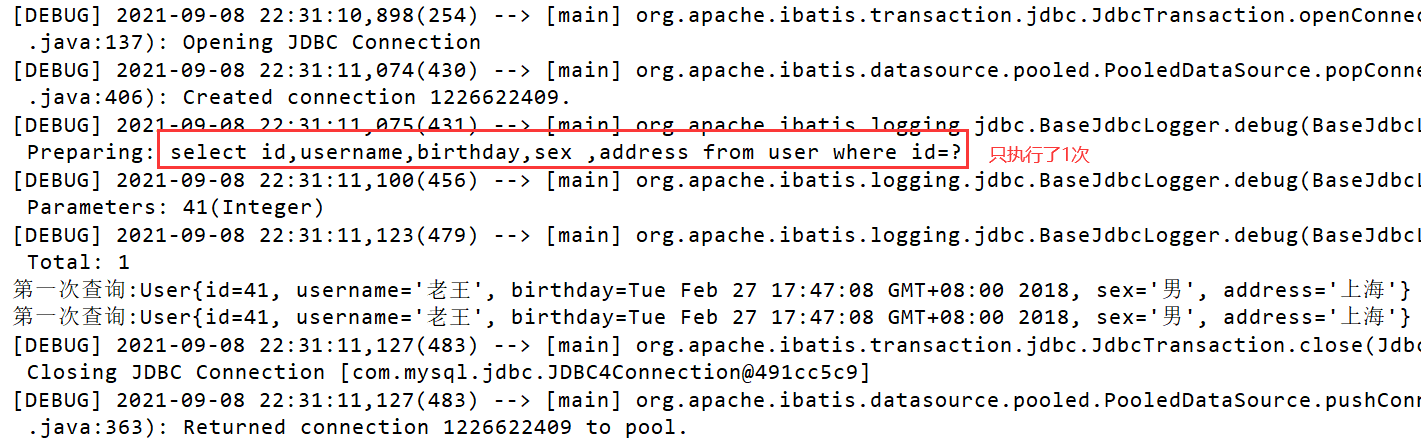
#### 1.2.1.2 编写映射文件

|  |
| --- |
| <**select id="findUserById" resultType="user" parameterType="int"**>  select id,username,birthday,sex ,address from user where id=#{id} </**select**> |

#### 1.2.1.3 编写测试方法

|  |
| --- |
| @Test **public void** testFindUserById() **throws** Exception{  User user1 = **userDao**.findUserById(41);  System.***out***.println(**"第一次查询:"** + user1);  User user2 = **userDao**.findUserById(41);  System.***out***.println(**"第一次查询:"** + user2);  **sqlSession**.close(); } |

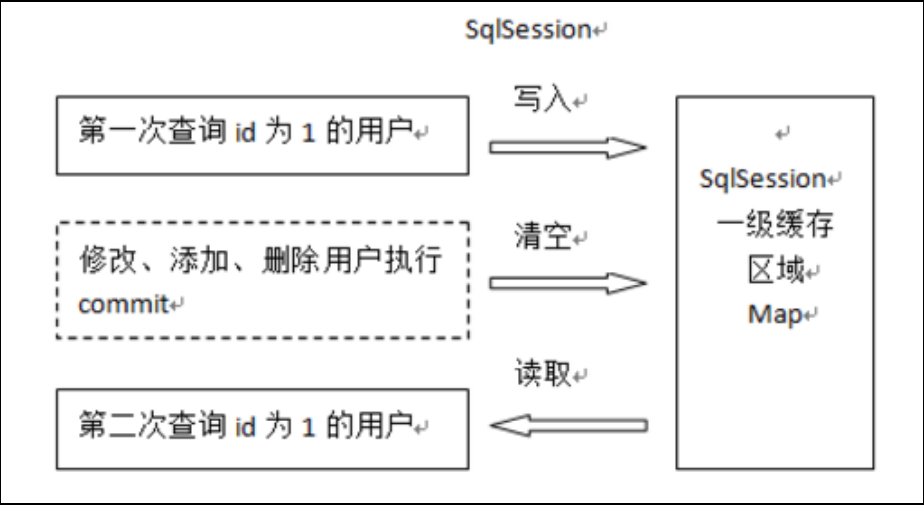
接下来，我们观察控制台输出信息：



我们可以发现，虽然在上面的代码中我们查询了两次，但最后只执行了一次数据库操作，这就是Mybatis提供给我们的一级缓存在起作用了。因为一级缓存的存在，导致第二次查询id为41的记录时，并没有发出sql语句从数据库中查询数据，而是从一级缓存中查询。

### 1.2.2 一级缓存的分析

一级缓存是SqlSession范围的缓存，当调用SqlSession的修改，添加，删除，commit()，close()等方法时，就会清空一级缓存。



第一次发起查询用户id为 1 的用户信息，先去找缓存中是否有id为 1 的用户信息，如果没有，从数据库查询用户信息。

得到用户信息，将用户信息存储到一级缓存中。

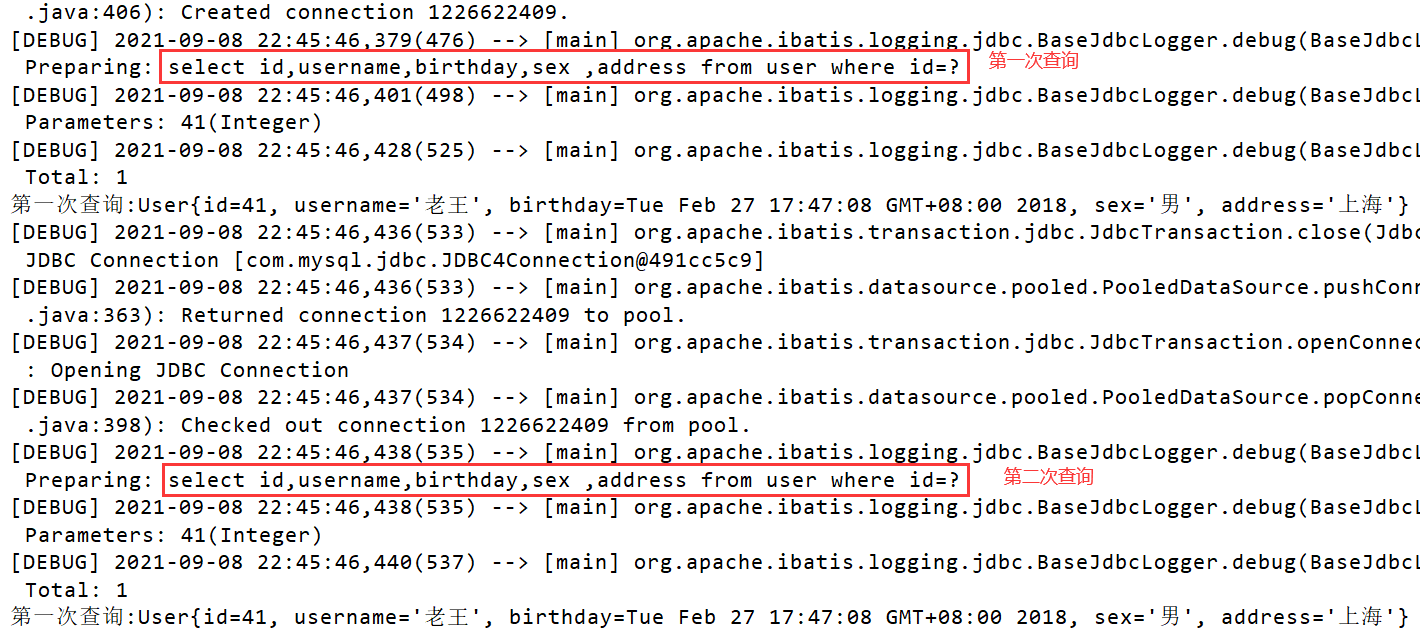
如果sqlSession去执行 commit操作（执行插入、更新、删除），清空 SqlSession 中的一级缓存，这样做的目的为了让缓存中存储的是最新的信息，**避免脏读**。

第二次发起查询用户id为1的用户信息，先去找缓存中是否有id为1的用户信息，缓存中有，直接从缓存中获取用户信息。

#### 1.2.2.1 测试清空缓存

|  |
| --- |
| @Test **public void** testFindUserById() **throws** Exception{  User user1 = **userDao**.findUserById(41);  System.***out***.println(**"第一次查询:"** + user1);  **sqlSession**.close();  *//重新获取SqlSession对象* SqlSession sqlSession = **sessionFactory**.openSession(**true**);  **userDao** = sqlSession.getMapper(UserDao.**class**);  User user2 = **userDao**.findUserById(41);  System.***out***.println(**"第一次查询:"** + user2);  **this**.**sqlSession**.close(); } |

我们注意观察控制台信息:



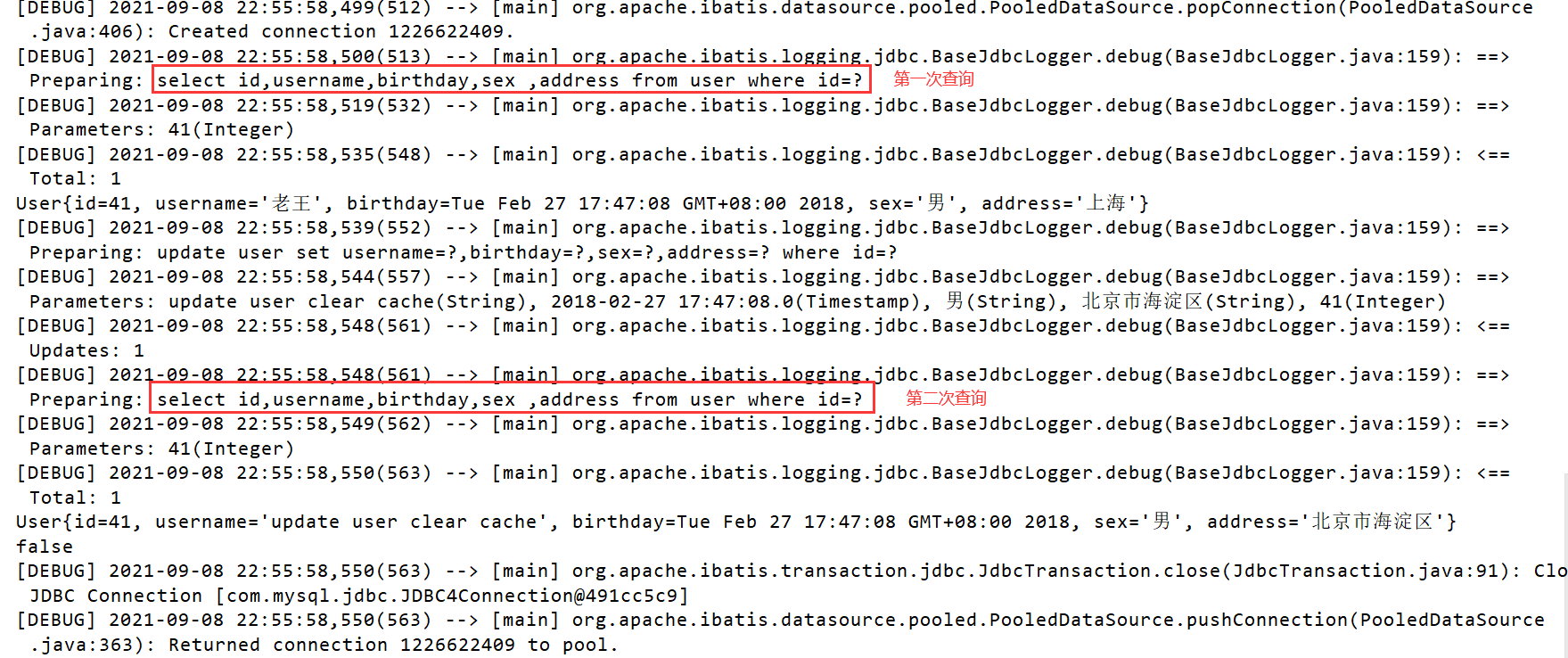
问题:如果数据库里面的数据发生变化，缓存里面的数据会同步更新吗？

我们可以测试缓存能不能同步。

#### 1.2.2.2 测试缓存同步

|  |
| --- |
| *//测试缓存同步* @Test **public void** testClearlCache(){  *//1.根据 id 查询用户* User user1 = **userDao**.findUserById(41);  System.***out***.println(user1);  *//2.更新用户信息* user1.setUsername(**"update user clear cache"**);  user1.setAddress(**"北京市海淀区"**);  **userDao**.updateUser(user1);  *//3.再次查询 id 为 41 的用户* User user2 = **userDao**.findUserById(41);  System.***out***.println(user2);  System.***out***.println(user1 == user2); } |

然后观察控制台



也就是说我们第二次获取数据，**并没有从缓存里面获取，而是直接查询了数据库**。

为什么?

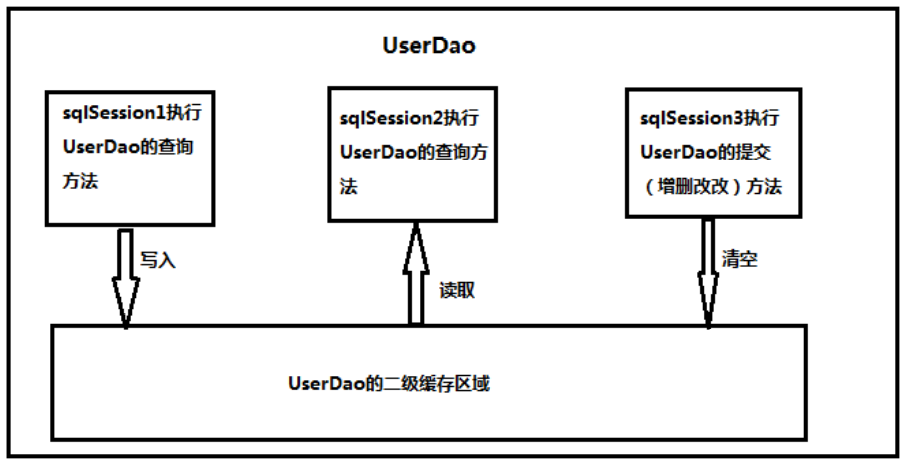
我们前面说过:

**当调用SqlSession的修改，添加，删除，commit()，close()等方法时，就会清空一级缓存。**

### 1.2.3 二级缓存

二级缓存是mapper映射级别的缓存，多个SqlSession去操作同一个Mapper映射的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

二级缓存结构图:



#### 1.2.3.1 测试二级缓存

* 第一步 在SqlMapConfig.xml 文件开启二级缓存

|  |
| --- |
| <**settings**>  *<!-- 开启二级缓存的支持 -->* <**setting name="cacheEnabled" value="true"**/> </**settings**> |

因为cacheEnabled的取值默认就为true，所以这一步可以省略不配置。为true代表开启二级缓存；为false代表不开启二级缓存。

* 第二步 配置相关的Mapper映射文件

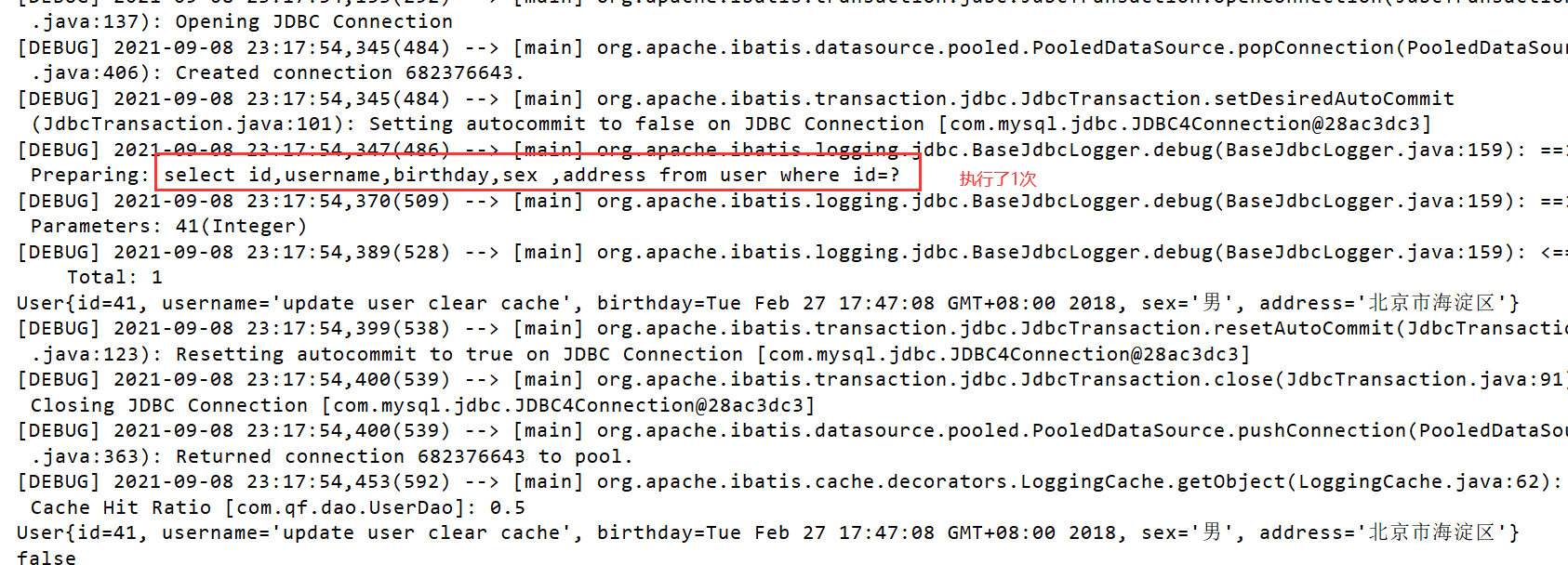
|  |
| --- |
| <**mapper namespace="com.qf.dao.UserDao"**>  *<!-- 开启二级缓存的支持 -->* <**cache**></**cache**> |

* 第三步 配置statement 上面的useCache属性

|  |
| --- |
| <**select id="findUserById" resultType="user" parameterType="int" useCache="true"**>  select id,username,birthday,sex ,address from user where id=#{id} </**select**> |

* 测试二级缓存

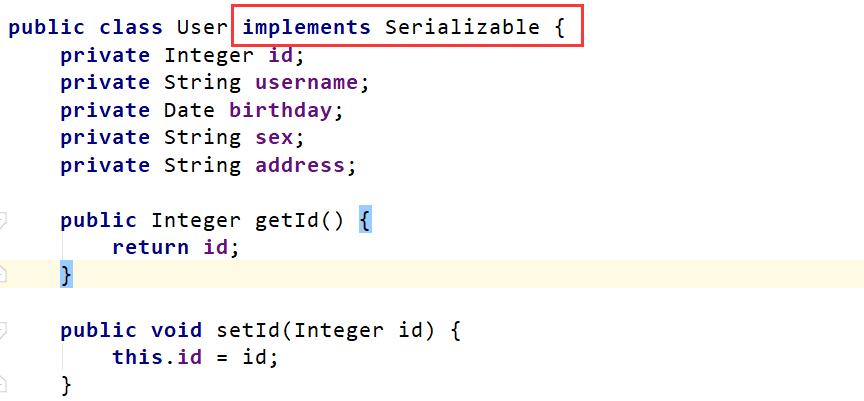
|  |
| --- |
| */\*\*  \* 测试二级缓存  \*/* @Test **public void** testFirstLevelCache(){  SqlSession sqlSession1 = **sessionFactory**.openSession();  UserDao dao1 = sqlSession1.getMapper(UserDao.**class**);  User user1 = dao1.findUserById(41);  System.***out***.println(user1);  sqlSession1.close();*//一级缓存消失* SqlSession sqlSession2 = **sessionFactory**.openSession();  UserDao dao2 = sqlSession2.getMapper(UserDao.**class**);  User user2 = dao2.findUserById(41);  System.***out***.println(user2);  sqlSession2.close();*//一级缓存消失 猜测会再次查询数据库* System.***out***.println(user1 == user2); } |



经过上面的测试，我们发现执行了两次查询，并且在执行第一次查询后，我们关闭了一级缓存，再去执行第二次查询时，我们发现并没有对数据库发出sql语句，所以此时的数据就只能是来自于我们所说的二级缓存。

注意:

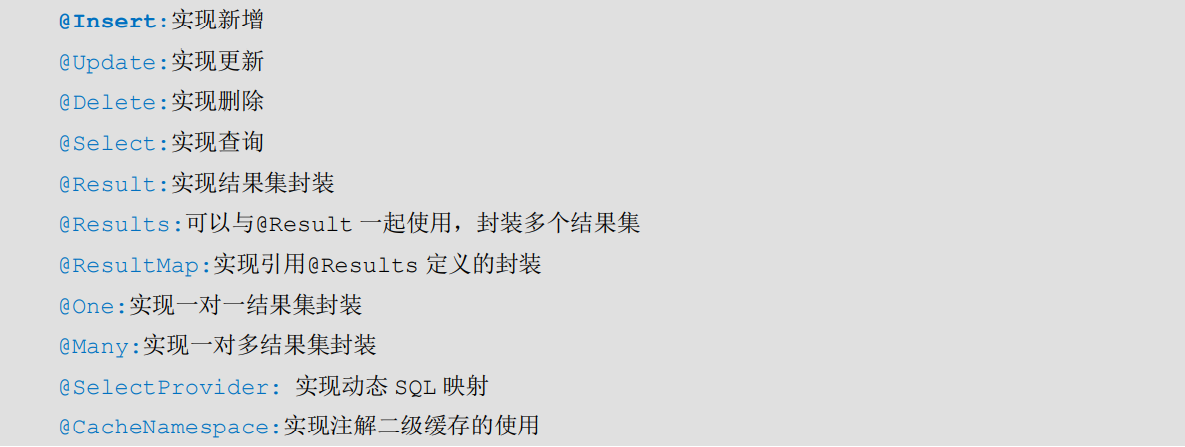
当我们在使用二级缓存时，**所缓存的类一定要实现java.io.Serializable接口**，这种就可以使用序列化方式来保存对象。



## 1.3 Mybatis 注解开发

这几年来注解开发越来越流行，Mybatis也可以使用注解开发方式，这样我们就可以减少编写Mapper映射文件了。本次我们先围绕一些基本的CRUD来学习，再学习复杂映射关系。

mybatis的常用注解说明:



### 1.3.1使用Mybatis注解实现基本CRUD

单表CRUD操作是最基本的操作，前面我们的学习都是基于Mybaits的映射文件来实现的。

* 引入依赖，创建核心配置文件(省略)
* 创建实体类

|  |
| --- |
| **public class** User **implements** Serializable {  **private** Integer **id**;  **private** String **username**;  **private** Date **birthday**;  **private** String **sex**;  **private** String **address**;   **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;  }   @Override  **public** String toString() {  **return "User{"** +  **"id="** + **id** +  **", username='"** + **username** + **'\''** +  **", birthday="** + **birthday** +  **", sex='"** + **sex** + **'\''** +  **", address='"** + **address** + **'\''** +  **'}'**;  } } |

* 定义接口

|  |
| --- |
| **public interface** UserDao {   @Select(**"select \* from user"**)  **public** List<User> findAll();   */\*\*  \* 保存操作  \** ***@param user*** *\** ***@return*** *\*/* @Insert(**"insert into user(username,sex,birthday,address)values(#{username},#{sex},#{birthday},#{address})"**)  @SelectKey(keyColumn=**"id"**,keyProperty=**"id"**,resultType=Integer.**class**,before =  **false**,statement = { **"select last\_insert\_id()"** })  **int** saveUser(User user);  */\*\*  \* 更新操作  \** ***@param user*** *\** ***@return*** *\*/* @Update(**"update user set username=#{username},address=#{address},sex=#{sex},birthday=#{birthday} where id =#{id} "**)  **void** updateUser(User user);  */\*\*  \* 删除用户  \** ***@param userId*** *\** ***@return*** *\*/* @Delete(**"delete from user where id = #{uid} "**)  **void** deleteUser(Integer userId);  */\*\*  \* 查询使用聚合函数  \** ***@return*** *\*/* @Select(**"select count(\*) from user "**)  **int** findTotal();  } |

注意: 如果数据表字段和实体类字段不一致怎么办?

User.java

|  |
| --- |
| **public class** User **implements** Serializable {  **private** Integer **userId**;  **private** String **userName**;  **private** Date **userBirthday**;  **private** String **userSex**;  **private** String **userAddress**;  //get set toString方法省略 } |

UserDao.java

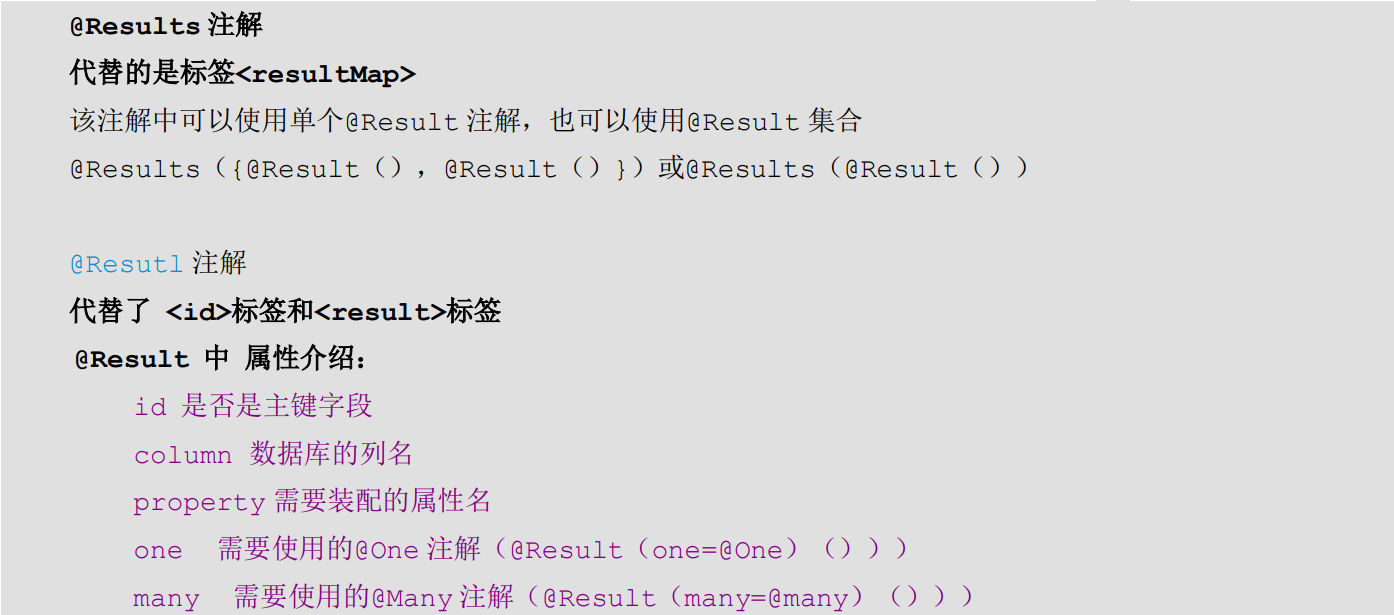
|  |
| --- |
| @Results(id=**"userMap"**,  value= {  @Result(id=**true**,column=**"id"**,property=**"userId"**),  @Result(column=**"username"**,property=**"userName"**),  @Result(column=**"sex"**,property=**"userSex"**),  @Result(column=**"address"**,property=**"userAddress"**),  @Result(column=**"birthday"**,property=**"userBirthday"**)  }) @Select(**"select \* from user"**) **public** List<User> findAll(); |

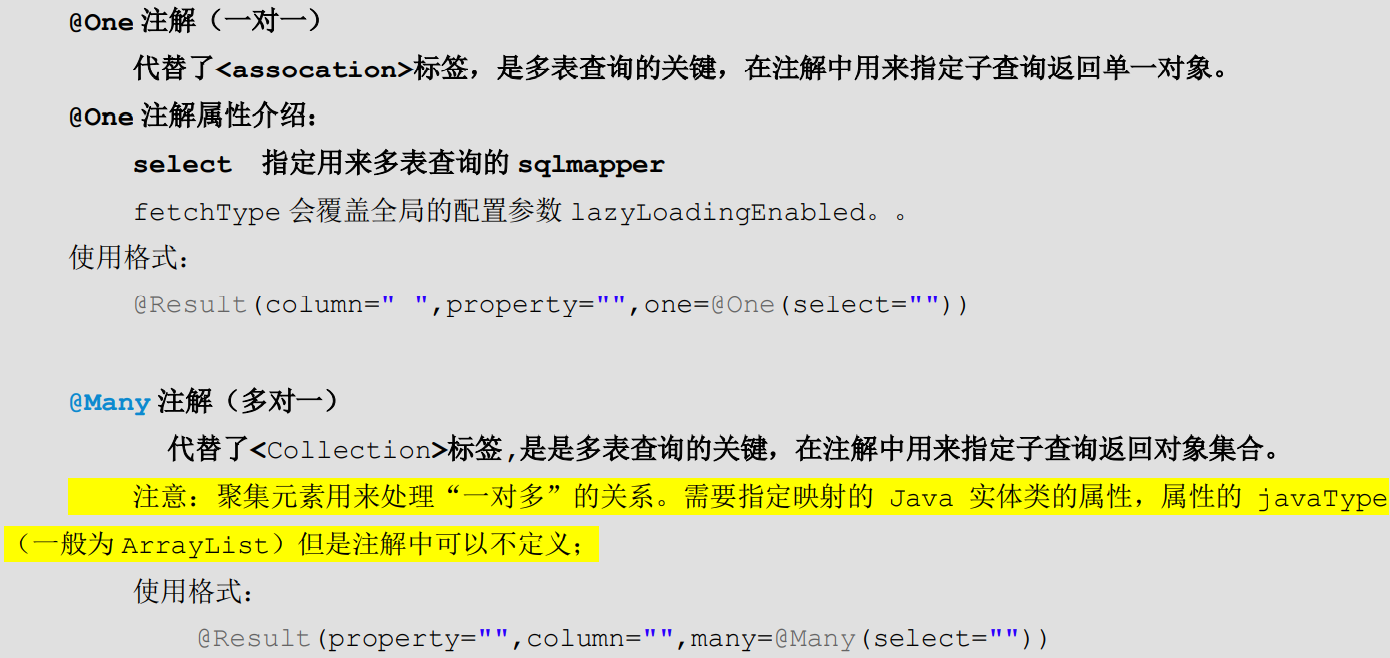
测试省略。

### 1.3.2使用注解实现复杂关系映射开发(1对1)

实现复杂关系映射之前我们可以在映射文件中通过配置来实现，在使用注解开发时我们需要借助@Results注解，@Result 注解，@One 注解，@Many注解。

复杂关系映射的注解说明：





需求：加载账户信息时并且加载该账户的用户信息，根据情况可实现延迟加载。（注解方式实现）

#### 1.3.2.1添加User实体类及Account实体类

User.java

|  |
| --- |
| **public class** User **implements** Serializable {  **private** Integer **userId**;  **private** String **userName**;  **private** Date **userBirthday**;  **private** String **userSex**;  **private** String **userAddress**;  // get set toString方法省略  } |

Account.java

|  |
| --- |
| **public class** Account **implements** Serializable {  **private** Integer **id**;  **private** Integer **uid**;  **private** Double **money**;  *//多对一关系映射：从表方应该包含一个主表方的对象引用* **private** User **user**;  // get set toString方法省略 } |

#### 1.3.2.2 添加账户、用户的持久层接口并使用注解配置

AccountDao.java

|  |
| --- |
| **public interface** AccountDao {  @Results(id=**"accountMap"**,  value= {  @Result(id=**true**,column=**"id"**,property=**"id"**),  @Result(column=**"uid"**,property=**"uid"**),  @Result(column=**"money"**,property=**"money"**),  @Result(column=**"uid"**,  property=**"user"**,  one=@One(select=**"com.qf.dao.UserDao.findById"**,  fetchType= FetchType.***LAZY***)  )  })  @Select(**"select \* from account"**)  List<Account> findAll(); } |

UserDao.java

|  |
| --- |
| @Results(id=**"userMap"**,  value= {  @Result(id=**true**,column=**"id"**,property=**"userId"**),  @Result(column=**"username"**,property=**"userName"**),  @Result(column=**"sex"**,property=**"userSex"**),  @Result(column=**"address"**,property=**"userAddress"**),  @Result(column=**"birthday"**,property=**"userBirthday"**)  }) @Select(**"select \* from user"**) **public** List<User> findAll();  **@ResultMap("userMap") @Select("select \* from user where id = #{uid} ") public User findById();** |

#### 1.3.2.3 测试一对一关联及延迟加载

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

注释遍历的代码，观察控制台，是否存在延迟加载的情况。

### 1.3.3 使用注解实现一对多复杂关系映射

需求：

查询用户信息时，也要查询他的账户列表。使用注解方式实现。

分析：

一个用户具有多个账户信息，所以形成了用户(User)与账户(Account)之间的一对多关系。

#### 1.3.3.1 添加实体类

User.java

|  |
| --- |
| **public class** User **implements** Serializable {  **private** Integer **userId**;  **private** String **userName**;  **private** Date **userBirthday**;  **private** String **userSex**;  **private** String **userAddress**;  *//一对多关系映射：主表方法应该包含一个从表方的集合引用* **private** List<Account> **accounts**;  // get set toString方法省略  } |

Account.java

|  |
| --- |
| **public class** Account **implements** Serializable {  **private** Integer **id**;  **private** Integer **uid**;  **private** Double **money**;  // get set toString方法省略  } |

#### 1.3.3.2添加账户、用户的持久层接口并使用注解配置

UserDao.java

|  |
| --- |
| **public interface** UserDao {  @Results(id=**"userMap"**,  value= {  @Result(id=**true**,column=**"id"**,property=**"userId"**),  @Result(column=**"username"**,property=**"userName"**),  @Result(column=**"sex"**,property=**"userSex"**),  @Result(column=**"address"**,property=**"userAddress"**),  @Result(column=**"birthday"**,property=**"userBirthday"**),  @Result(column=**"id"**,property=**"accounts"**,  many=@Many(  select=**"com.qf.dao.AccountDao.findByuId"**,  fetchType= FetchType.***LAZY*** )  )  })  @Select(**"select \* from user"**)  **public** List<User> findAll(); |

AccountDao.java

|  |
| --- |
| **public interface** AccountDao {  @Select(**"select \* from account where uid = #{uid} "**)  List<Account> findByuId(Integer userId); } |

#### 1.3.3.3 测试

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