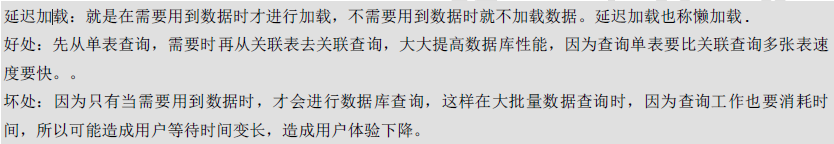
**Mybatis框架课程第四天**

# Mybatis延迟加载策略

通过前面的学习，我们已经掌握了Mybatis中一对一，一对多，多对多关系的配置及实现，可以实现对象的关联查询。实际开发过程中很多时候我们并不需要总是在加载用户信息时就一定要加载他的账户信息。此时就是我们所说的延迟加载。

### 何为延迟加载



前面实现多表操作时，我们使用了resultMap来实现一对一，一对多，多对多关系的操作。主要是通过association、collection实现一对一及一对多映射。association、collection具备延迟加载功能。

### 实现需求

需求：查询账户(Account)信息并且关联查询用户(User)信息。如果先查询账户(Account)信息即可满足要求，当我们需要查询用户(User)信息时再查询用户(User)信息。把对用户(User)信息的按需去查询就是延迟加载。

### 使用Assocation实现延迟加载

需求：查询账户信息同时查询用户信息。

Account实体类中加入一个User类的对象

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

#### 第一步：只查询账户信息的Mapper接口

SQL：select \* from account;

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

#### 第二步：AccountMapper.xml映射文件

|  |
| --- |
| <**select id="queryAccountList" resultMap="accountLazyLoadUserResultMap"**>  select *\** from account </**select**> |

其中上面resultMap属性的值accountLazyLoadUserResultMap，它是我们自定义的resultMap，具体如下：

|  |
| --- |
| <**resultMap id="accountLazyLoadUserResultMap" 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-->  <!--  select:指定延迟加载要执行的statement的id（是根据uid查询用户信息的statement）  要使用UserMapper.xml中getUserById完成根据用户id（uid）对用户信息的查询，如果  getUserById不在本Mapper中，需要前面加namespace  column：延迟加载查询传入的参数（用户所关联账户信息的外键字段名uid）  -->* <**association property="user" javaType="User" select="com.itheima.mybatis.mapper.UserMapper .getUserById**  **" column="uid"**>  </**association**> </**resultMap**> |

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

#### 第三步：UserMapper接口及UserMapper.xml映射文件

修改项目中的UserMapper接口，添加一个根据用户id查询用户对象的方法。

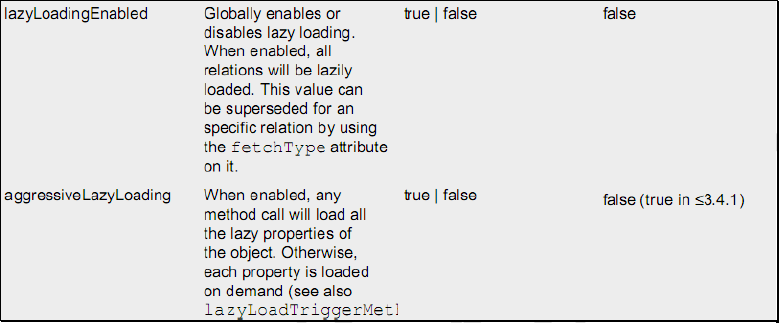
|  |
| --- |
| User getUserById(**int** id); |

在UserMapper.xml映射文件中添加映射。

|  |
| --- |
| <**select id="getUserById" parameterType="int" resultType="user"**>  select *\** from user where id=#{id} </**select**> |

#### 第四步：开启Mybatis的延迟加载策略

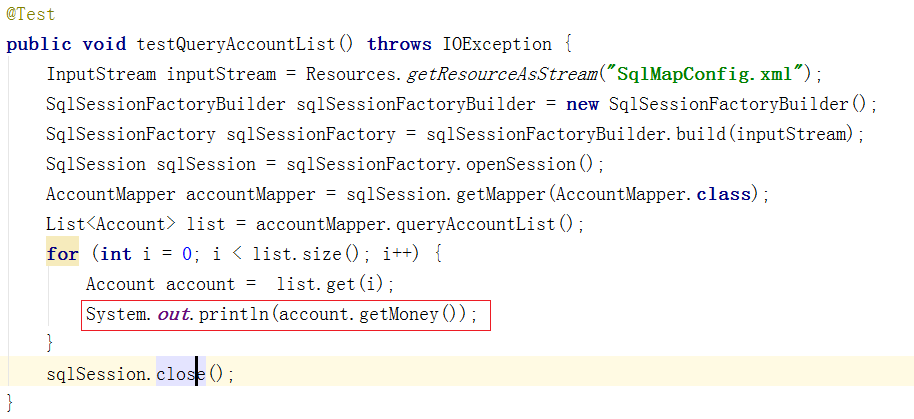
进入Mybaits的官方文档，找到settings的说明信息：



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。

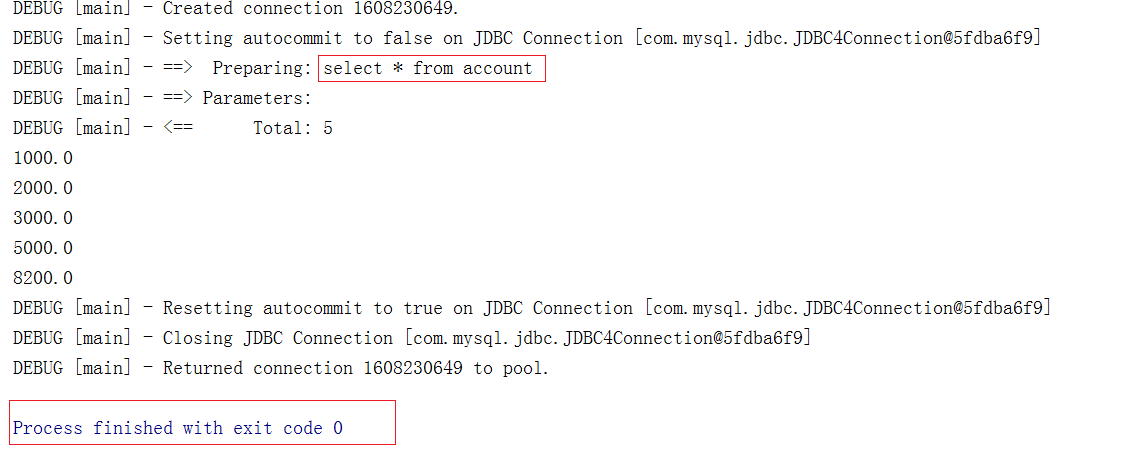


#### 第四步：编写测试只查账户信息不查用户信息



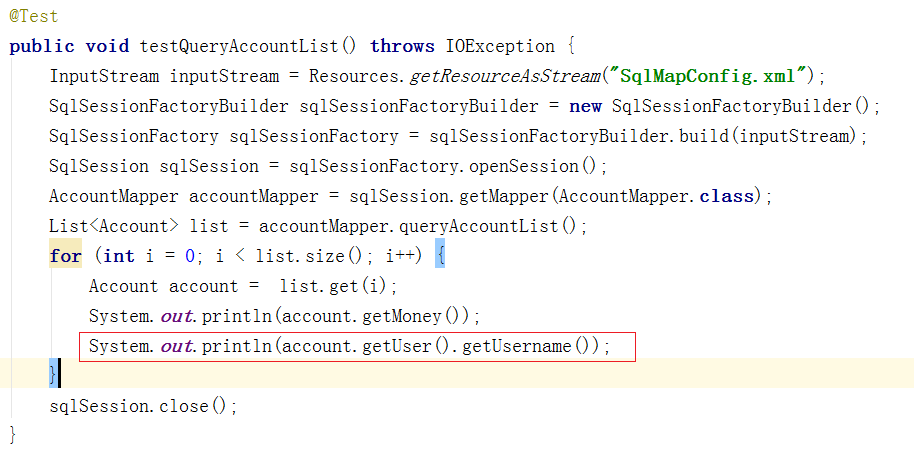
注意：只打印account账户的money字段信息

#### 效果



我们发现，因为本次只是将Account对象查询出来放入List集合中，并没有涉及和使用到User对象，所以就没有发出SQL语句查询账户所关联的User对象的查询。

#### 第五步：测试加载账户信息同时加载用户信息

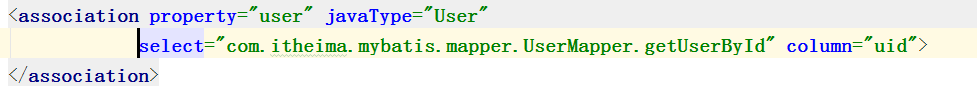


#### 效果

#### 小结

通过本示例，我们可以发现Mybatis的延迟加载还要有很明显效果，对于提升软件性能这是一个不错的手段。

实现的关键：association的配置



### 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<collection>结点中配置延迟加载策略。

<collection>结点中也有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。

#### 第一步：在User实体类中加入List<Account>属性

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

#### 第二步：UserMapper接口及AccountMapper接口

在UserMapper接口中添加查询所有用户信息的方法

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

在AccountMapper接口中添加一个根据用户id查询账户列表的方法

|  |
| --- |
| List<Account> getAccountsByUid (int uid); |

#### 第三步：UserMapper.xml 配置文件

|  |
| --- |
| <**select id="queryUserList" resultMap="userLazyloadAccountsResultMap"**>  SELECT *\** FROM USER </**select**> <**resultMap id="userLazyloadAccountsResultMap" 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" select="com.itheima.mybatis.mapper.UserMapper.getAccountsByUid" column="id"**>  <**id column="id" property="id"**/>  <**result column="uid" property="uid"**/>  <**result column="money" property="money"**/>  </**collection**> </**resultMap**> |

<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

#### 第四步：AccountMapper.xml映射文件

UserMapper.xml映射文件中的<collection>标签的select属性的值就是来自这个文件的<select>的id的值。

|  |
| --- |
| <**select id="getAccountsByUid" parameterType="int" resultType="account"**  >  select *\** from account where uid=#{uid} </**select**> |

#### 第五步：开启Mybatis的延迟加载

在Mybatis的配置文件SqlMapConfig.xml中添加延迟加载的配置

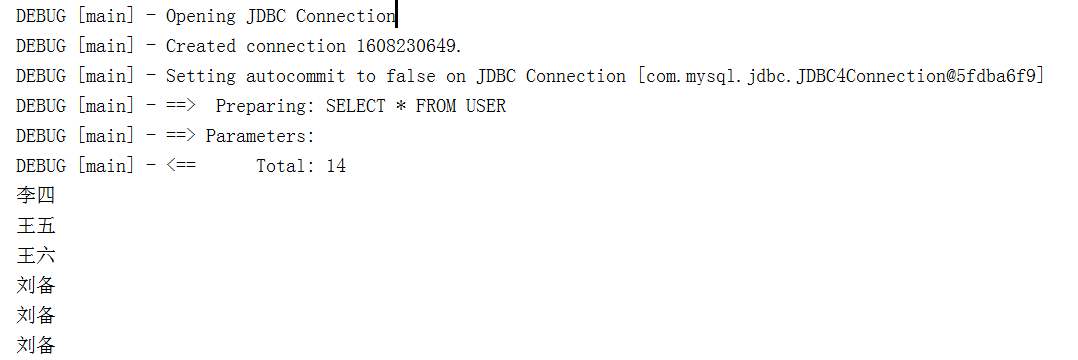


#### 第六步：测试只加载用户信息

|  |
| --- |
| @Test **public void** testQueryUserList() **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.queryUserList();  **for** (**int** i = 0; i < list.size(); i++) {  User user1 = list.get(i);  System.***out***.println(user1.getUsername());  }  sqlSession.close(); } |

注意：只打印User的username字段信息

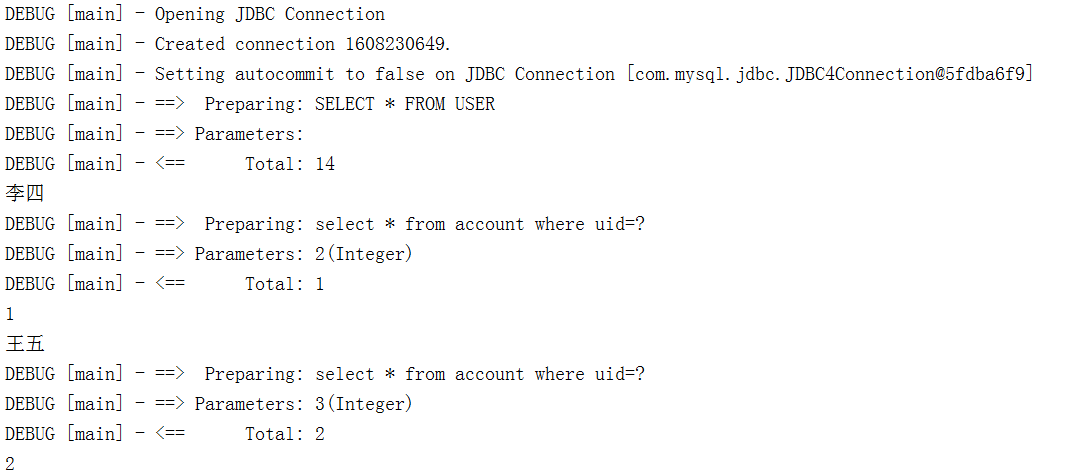
#### 效果



#### 第七步：测试加载用户信息同时还加载账户列表

|  |
| --- |
| @Test **public void** testQueryUserList() **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.queryUserList();  **for** (**int** i = 0; i < list.size(); i++) {  User user1 = list.get(i);  System.***out***.println(user1.getUsername());  System.***out***.println(  (user1.getAccountList()!=**null**?  user1.getAccountList().size():0));  }  sqlSession.close(); } |

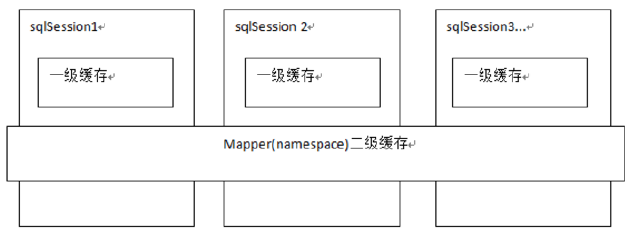
#### 效果



# Mybatis缓存

像大多数的持久化框架一样，Mybatis也提供了缓存策略，通过缓存策略来减少数据库的查询次数，从而提高性能。

Mybatis中缓存分为一级缓存，二级缓存。



## Mybatis一级缓存

### 证明一级缓存的存在

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

#### 第一步：编写UserMapper接口

|  |
| --- |
| User queryUserById(Integer id); |

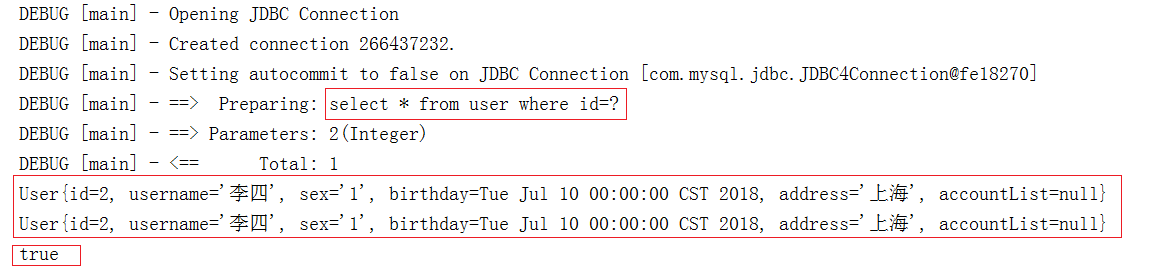
#### 第二步：编写UserMapper.xml映射文件

|  |
| --- |
| <**select id="queryUserById" parameterType="\_int" resultType="user"**>  select *\** from user where id=#{id} </**select**> |

#### 第三步：编写测试方法

|  |
| --- |
| @Test **public void** testQueryUserById() **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**);  User user = userMapper.queryUserById(2);  User user1 = userMapper.queryUserById(2);  System.***out***.println(user);  System.***out***.println(user1);  System.***out***.println(user == user1);  sqlSession.close(); } |

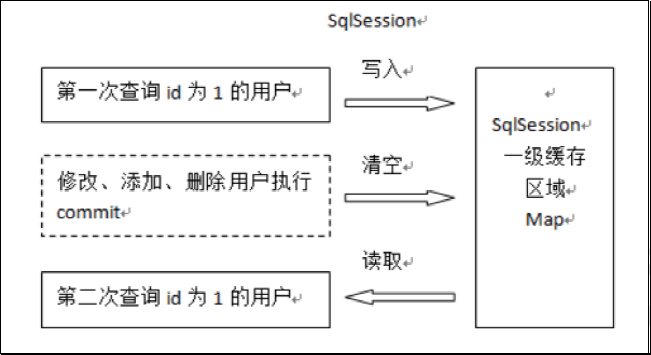
#### 效果



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2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。

### 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。

### 测试一级缓存的清空

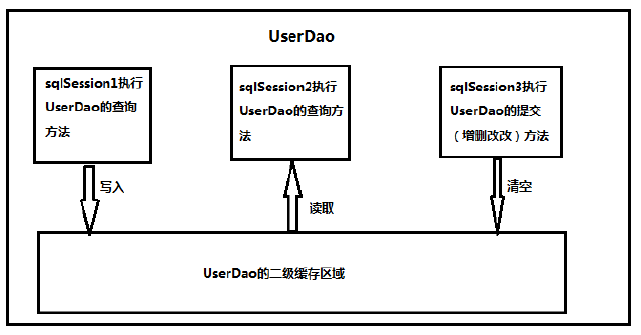
|  |
| --- |
| @Test **public void** testQueryUserById() **throws** IOException {  InputStream inputStream = Resources.*getResourceAsStream*(**"SqlMapConfig.xml"**);  SqlSessionFactoryBuilder sqlSessionFactoryBuilder = **new** SqlSessionFactoryBuilder();  SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(inputStream);  SqlSession sqlSession = sqlSessionFactory.openSession();  UserMapper userMapper = sqlSession.getMapper(UserMapper.**class**);  User user = userMapper.queryUserById(2);  System.***out***.println(user);  sqlSession.close();   SqlSession sqlSession1 = sqlSessionFactory.openSession();  UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.**class**);  User user1 = userMapper1.queryUserById(2);  System.***out***.println(user1);  System.***out***.println(user == user1);  sqlSession.close(); } |

当执行sqlSession.close()后，再次获取sqlSession并查询id=2的User对象时，又重新执行了Sql 语句，从数据库进行了查询操作。

## Mybatis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



首先开启mybatis的二级缓存。

sqlSession1去查询用户信息，查询到用户信息会将查询数据存储到二级缓存中。

如果SqlSession3去执行相同 mapper映射下sql，执行commit提交，将会清空该 mapper映射下的二级缓存区域的数据。

sqlSession2去查询与sqlSession1相同的用户信息，首先会去缓存中找是否存在数据，如果存在直接从缓存中取出数据。

### 二级缓存的开启与关闭

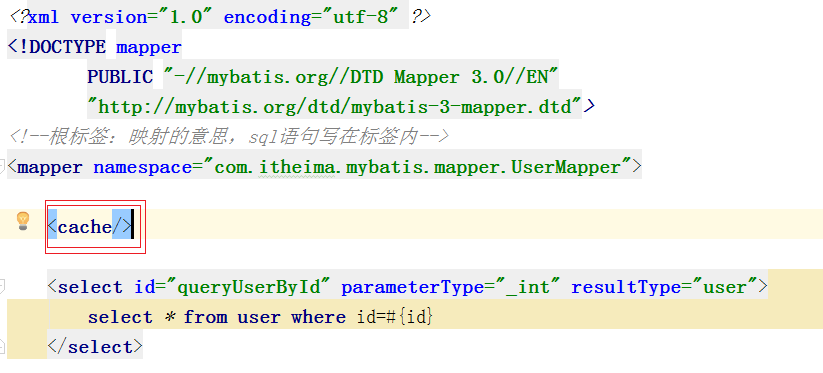
#### 第一步：在SqlMapConfig.xml文件开启二级缓存

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

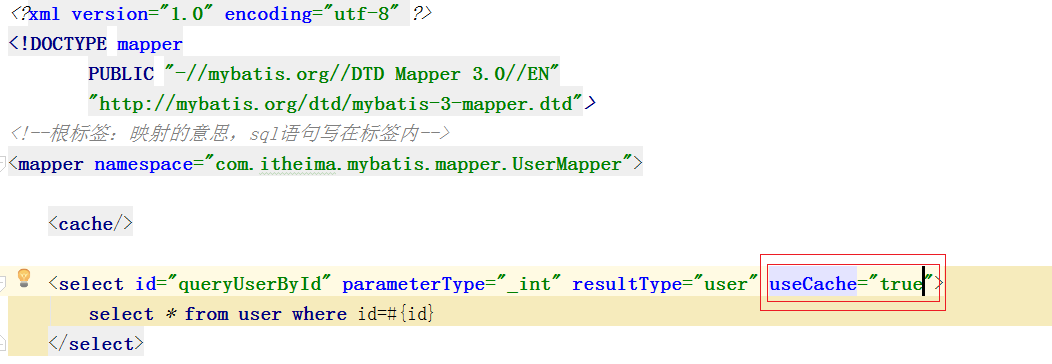
因为cacheEnabled的取值默认就为true，所以这一步可以省略不配置。为true代表开启二级缓存；为false代表不开启二级缓存。

#### 第二步：配置相关的Mapper映射文件

<cache>标签表示当前这个mapper映射将使用二级缓存，区分的标准就看mapper的namespace值。



#### 第三步：配置statement上面的useCache属性



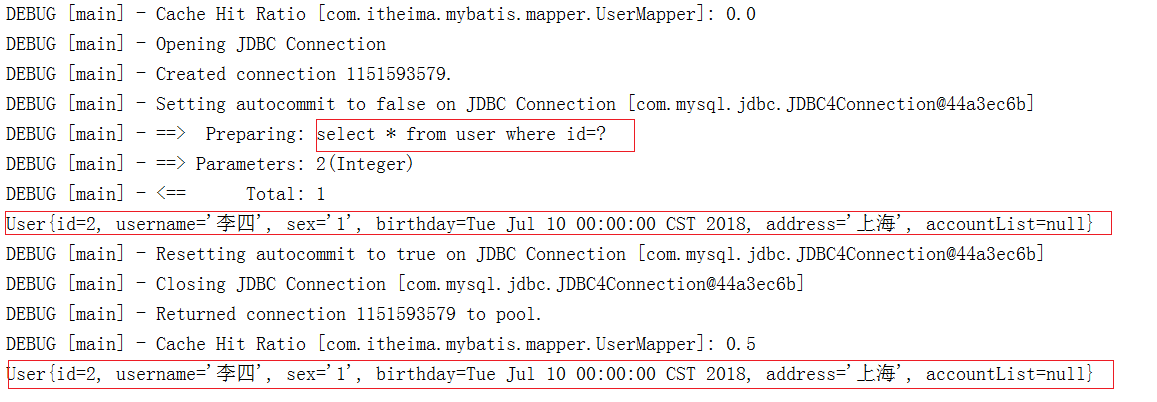
将UserMapper.xml映射文件中的<select>标签中设置useCache=”true”代表当前这个statement要使用二级缓存，如果不使用二级缓存可以设置为false。

注意：针对每次查询都需要最新的数据sql，要设置成useCache=false，禁用二级缓存。

### 二级缓存测试

|  |
| --- |
| @Test **public void** testQueryUserById() **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**);  User user = userMapper.queryUserById(2);  System.***out***.println(user);  sqlSession.close();   SqlSession sqlSession1 = sqlSessionFactory.openSession();  UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.**class**);  User user1 = userMapper1.queryUserById(2);  System.***out***.println(user1);  sqlSession.close(); } |

#### 效果

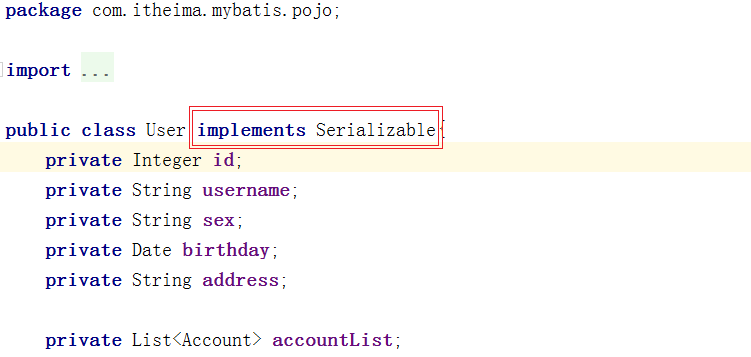


经过上面的测试，我们发现执行了两次查询，并且在执行第一次查询后，我们关闭了一级缓存，再去执行第二次查询时，我们发现并没有对数据库发出sql语句，所以此时的数据就只能是来自于我们所说的二级缓存。

### 二级缓存注意事项

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。

如下图：



# Mybatis 注解开发

Mybatis最大的特点是使用XML存储SQL语句，但是Mybatis也可以使用注解开发方式，好处是不用编写Mapper映射文件，坏处是SQL语句又耦合在Java代码中了。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及延迟加载。

## 使用Mybatis注解实现基本CRUD

单表的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。

### Mybatis的注解说明

@Insert:实现新增

@Update:实现更新

@Delete:实现删除

@Select:实现查询

@Result:实现结果集封装

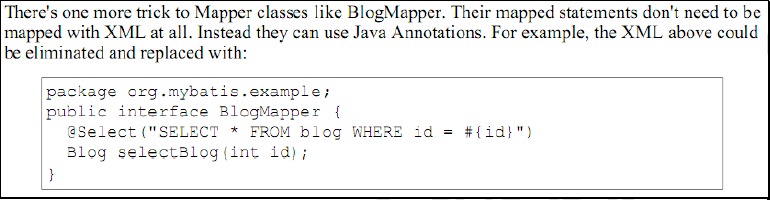
@Results:可以与@Result一起使用，封装多个结果集

@One:实现一对一结果集封装

@Many:实现一对多结果集封装

@SelectProvider: 实现动态SQL映射

我们也通过查看Mybatis官方文档来学习Mybatis注解开发



### 使用注解方式开发UserMapper接口

在原有的项目中，把UserMapper接口中添加CRUD方法，并带上基本的注解。

|  |
| --- |
| **package** com.itheima.mybatis.mapper;  **import** com.itheima.mybatis.pojo.User; **import** org.apache.ibatis.annotations.Delete; **import** org.apache.ibatis.annotations.Insert; **import** org.apache.ibatis.annotations.Select; **import** org.apache.ibatis.annotations.Update;  **import** java.util.List;  **public interface** UserMapper {   @Select(value = **"select \* from user where id=#{id}"**)  User queryUserById(Integer id);   @Insert(value = **"insert into user(username,sex,birthday,address) "** +  **"values (#{username},#{sex},#{birthday},#{address})"**)  **void** saveUser(User user);   @Update(value=**"update user set username=#{username} where id=#{id}"**)  **void** updateUsernameById(User user);   @Delete(value=**"delete from user where id=#{id}"**)  **void** deleteUserById(**int** id); } |

通过注解方式，我们就不需要再去编写UserMapper.xml 映射文件了。

### 修改SqlMapConfig 配置文件

因为不存在UserMapper.xml文件了，这样我们就不需要在Mybatis配置文件中加载UserMapper.xml映射文件了。此时我们只需要Mybatis 的配置文件能够加载我们的UserMapper接口就可以了。

|  |
| --- |
| <**mappers**>  <**package name="com.itheima.mybatis.mapper"**/> </**mappers**> |

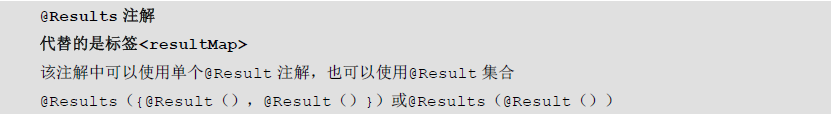
### 编写测试类

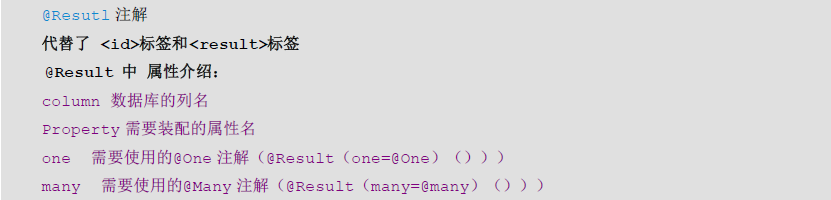
|  |
| --- |
| **package** com.itheima.mybatis.test;  **import** com.itheima.mybatis.mapper.UserMapper; **import** com.itheima.mybatis.pojo.User; **import** org.apache.ibatis.io.Resources; **import** org.apache.ibatis.session.SqlSession; **import** org.apache.ibatis.session.SqlSessionFactory; **import** org.apache.ibatis.session.SqlSessionFactoryBuilder; **import** org.junit.Test;  **import** java.io.IOException; **import** java.io.InputStream; **import** java.util.Date; **import** java.util.List;  **public class** TestUserMapper {   @Test  **public void** testQueryUserById() **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**);  User user = userMapper.queryUserById(2);  System.***out***.println(user);  sqlSession.close();  }    @Test  **public void** testSaveUser() **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**);  User user = **new** User();  user.setUsername(**"燕子李三"**);  user.setSex(**"1"**);  user.setBirthday(**new** Date());  user.setAddress(**"中华民国"**);  userMapper.saveUser(user);  System.***out***.println(user);  sqlSession.commit();  sqlSession.close();  }    @Test  **public void** testUpdateUsernameById() **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**);  User user = **new** User();  user.setId(12);  user.setUsername(**"小燕子"**);  userMapper.updateUsernameById(user);  sqlSession.commit();  sqlSession.close();  }    @Test  **public void** testDeleteUserById() **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**);  userMapper.deleteUserById(12);  sqlSession.commit();  sqlSession.close();  } } |

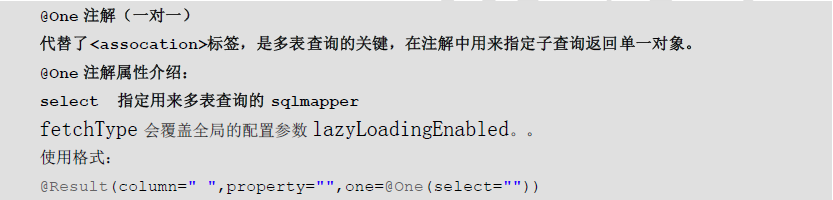
## 使用注解实现复杂关系映射开发

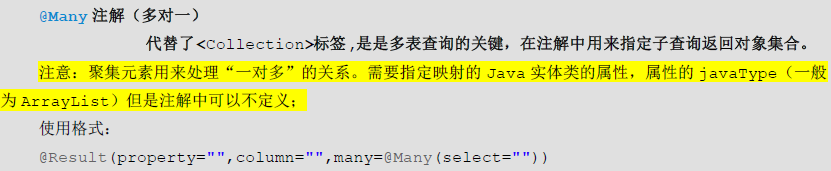
实现复杂关系映射之前我们可以在映射文件中通过配置<resultMap>来实现，但通过后我们发现并没有@ResultMap这个注解。下面我们一起来学习@Results注解，@Result注解，@One注解，@Many注解。

### 复杂关系映射的注解说明









### 使用注解实现一对一复杂关系映射及延迟加载

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）

#### 添加User实体类及Account实体类

Account实体类

|  |
| --- |
| **package** com.itheima.mybatis.pojo;  **import** java.io.Serializable;  **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实体类

|  |
| --- |
| **package** com.itheima.mybatis.pojo;  **import** java.io.Serializable; **import** java.util.Date; **import** java.util.List;  **public class** User **implements** Serializable{  **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** + **'\''** +  **'}'**;  } } |

#### 添加UserMapper接口及AccountMapper接口

AccountMapper接口

|  |
| --- |
| @Select(value=**"select \* from account"**)  @Results({  @Result(id = **true**,column = **"id"**,property = **"id"**),  @Result(column = **"uid"**,property = **"uid"**),  @Result(column = **"money"**,property = **"money"**),  @Result(property = **"user"**,column = **"uid"**,javaType = User.**class**,  one = @One(select = **"com.itheima.mybatis.mapper.UserMapper.getUserById"**,fetchType = FetchType.***LAZY***))  })  List<Account> queryAccountList(); |

UserMapper接口

|  |
| --- |
| @Select(value=**"select \* from user where id=#{id}"**) User getUserById(**int** id); |

#### 测试

参考本次课程的一对一关联延迟加载测试方法

### 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。

#### User实体类及Account实体类

参见注解实现一对一映射

#### UserMapper接口和AccountMapper接口

AccountMapper接口中根据uid查询账户信息

|  |
| --- |
| @Select(value=**"select \* from account where uid=#{uid}"**) List<Account> getAccountByUid(**int** uid); |

UserMapper接口注解配置

|  |
| --- |
| @Select(value=**"select \* from user"**) @Results({  @Result(id=**true**,column = **"id"**,property = **"id"**),  @Result(column = **"username"**,property = **"username"**),  @Result(column = **"sex"**,property = **"sex"**),  @Result(column = **"birthday"**,property = **"birthday"**),  @Result(column = **"address"**,property = **"address"**),  @Result(property = **"accountList"**,column = **"id"**,javaType = List.**class**,  many = @Many(select = **"com.itheima.mybatis.mapper.AccountMapper.getAccountByUid"**,fetchType = FetchType.***LAZY***)) }) List<User> queryUserAccounts(); |

@Many:相当于<collection>的配置

select属性：代表将要执行的sql语句

fetchType属性：代表加载方式，一般如果要延迟加载都设置为LAZY的值

#### 测试

参考本次课程的一对多关联延迟加载测试方法

# 总结

本次课程结束了，通过Mybatis课程的学习，相信大家的水平都得到了一个质的飞跃，通过框架课程的学习， 我们会发现现在的自己变得更强大了。