Mybatis 框架课程第四天

第1章 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。

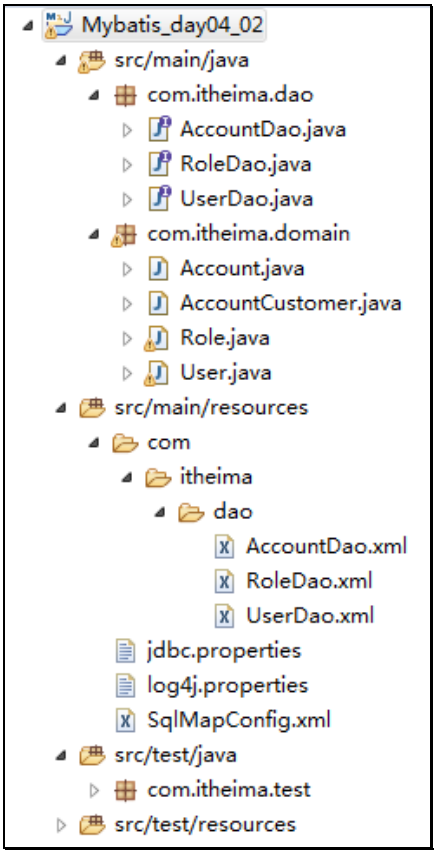
1.1.1 何为延迟加载?

延迟加载：就是在需要用到数据时才进行加载，不需要用到数据时就不加载数据。延迟加载也称懒加载.
好处：先从单表查询，需要时再从关联表去关联查询，大大提高数据库性能，因为查询单表要比关联查询多张表速度要快。。
坏处：因为只有当需要用到数据时，才会进行数据库查询，这样在大批量数据查询时，因为查询工作也要消耗时间，所以可能造成用户等待时间变长，造成用户体验下降。

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。

1.1.2 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。
工程目录结构如下：



1.1.3 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

```
public class Account {
    private int id;
    private int uid;
    private double money;
    //加入一个用户对象的信息  has a 关系
    private User user;
    //省略 getter 与 setter
}
```

1.1.3.1 第一步：只查询账户信息 的 DAO 接口

```
SQL:select * from account;
```

AccountDao 类中添加查询账户信息的方法:

AccountDao.java

```java
package com.itheima.dao;

import java.util.List;

public interface AccountDao {
    //查询账户信息
    public List<Account> findAccounts();
}
```

1.1.3.2 第二步：AccountDao.xml 映射文件

AccountDao.xml

```xml
<?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.itheima.dao.AccountDao">
    <select id="findAccounts" resultMap="accountLazyLoadUser">
        select * from account;
    </select>
```

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

```xml
<resultMap type="account" id="accountLazyLoadUser">
    <id property="id" column="id"/>
    <result property="uid" column="uid"/>
    <result property="money" column="money"/>
    <!--  select：指定延迟加载要执行的statement的id（是根据uid查询用户信息是statement）
          要使用UserDao.xml中findUserById完成根据用户id（uid）对用户信息的查询，
          如果findUserById不在本mapper中， 需要前边加namespace
          column：用户所关联账户信息的外键字段名uid -->
    <association property="user" javaType="com.itheima.domain.User"
        select="com.itheima.dao.UserDao.findUserById" 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

1.1.3.3 第三步：UserDao 接口及 UserDao.xml 映射文件

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

```java
public interface UserDao {
    public List<User> findUserById(Integer id);
}
```

在 UserDao.xml 映射文件中添加映射。

```
<mapper namespace="com.itheima.dao.UserDao">

    <select id="findUserById" parameterType="int" resultType="user">
       select * from user where id=#{id}
    </select>
```

1.1.3.4 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：

lazyLoadingEnabled	Globally enables or disables lazy loading. When enabled, all relations will be lazily loaded. This value can be superseded for an specific relation by using the fetchType attribute on it.	true | false	false
aggressiveLazyLoading	When enabled, any method call will load all the lazy properties of the object. Otherwise, each property is loaded on demand (see also lazyLoadTriggerMet	true | false	false (true in ≤3.4.1)

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。

SqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration
  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
   <properties resource="jdbc.properties"></properties>
   <!-- 配置延迟加载策略 -->
   <settings>
       <!-- 打开延迟加载的开关 -->
        <setting name="lazyLoadingEnabled" value="true" />
        <!-- 将积极加载改为消息加载即按需加载 -->
        <setting name="aggressiveLazyLoading" value="false"/>
   </settings>
```

1.1.3.5 第四步：编写测试只查账户信息不查用户信息。

```
@Test
public void testAccount() {
    SqlSession sqlSession = sqlSessionFactory.openSession();
    AccountDao accountDao = sqlSession.getMapper(AccountDao.class);
    List<Account> accList = accountDao.findAccounts();
    sqlSession.close();
}
```

测试结果如下：

```
Opening JDBC Connection
Created connection 815992954.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@30a3107a]
==>  Preparing: select * from account;
==> Parameters:
<==      Total: 3
Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@30a3107a]
Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@30a3107a]
```

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。

1.1.3.6 第五步：测试加载账户信息同时加载用户信息

重新修改测试方法：

```
@Test
public void testAccount() {
    SqlSession sqlSession = sqlSessionFactory.openSession();
    AccountDao accountDao = sqlSession.getMapper(AccountDao.class);
    List<Account> accList = accountDao.findAccounts();
    for(Account acc :accList) {
        System.out.println(acc.getId()+acc.getMoney());
        System.out.println(acc.getUser().getId());
    }
    sqlSession.close();
}
```

这一句代码，导致了加载用户信息，所以就会发出sql语句，去加载用户信息

测试效果如下：

```
Created connection 815992954.
Setting autocommit to false on JDBC Connection
==>  Preparing: select * from account;
==> Parameters:
<==      Total: 3

==>  Preparing: select * from user where id=?
==> Parameters: 41(Integer)
<==      Total: 1


==>  Preparing: select * from user where id=?
==> Parameters: 45(Integer)
<==      Total: 1
```

1.1.3.7 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

```
    <association property="user" javaType="com.itheima.domain.User"
            select="com.itheima.dao.UserDao.findUserById"
column="uid"></association>
```

1.1.4 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。

1.1.4.1 第一步：在 User 实体类中加入 List<Account>属性

User.java
```
package com.itheima.domain;

import java.io.Serializable;

public class User implements Serializable {
    private int id;
    private String username;// 用户姓名
    private String sex;// 性别
    private Date birthday;// 生日
    private String address;// 地址

    private List<Account> accList;
```

1.1.4.2 第二步：UserDao 接口及 AccountDao 接口

在 UserDao 接口中添加查询所有用户信息的方法

UserDao.java
```
package com.itheima.dao;

import java.util.List;

public interface UserDao {
    public List<User> findUserList();
}
```

在 AccountDao 接口中添加一个根据用户 id 查询账户列表的方法

AccountDao.java
```
package com.itheima.dao;

import java.util.List;

public interface AccountDao {
    //根据uid,查询账户信息
    public List<Account> findAccountsByUid(Integer uid);
}
```

1.1.4.3 第三步：UserDao.xml 配置文件

UserDao.xml

```
<?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.itheima.dao.UserDao">

    <select id="findUserList" resultMap="userLazyLoadAccountListResultMap">
       select * from user;
    </select>

    <resultMap type="user" id="userLazyLoadAccountListResultMap">
       <id property="id" column="id"/>
       <result property="username" column="username"/>
       <result property="sex" column="sex"/>
       <result property="birthday" column="birthday"/>
       <result property="address" column="address"/>
       <collection property="accList" ofType="account"
           select="com.itheima.dao.AccountDao.findAccountsByUid" column="id">
          <id property="id" column="id" />
          <result property="uid" column="uid"/>
          <result property="money" column="money"/>
       </collection>
    </resultMap>
</mapper>
```

select属性的值是来自于AccountDao.xml文件的<select>标签的id属性。
column属性来自于user的主属性id

<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

1.1.4.4 第四步：AccountDao.xml 映射文件

UserDao.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.

AccountDao.xml

```
<?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.itheima.dao.AccountDao">
    <select id="findAccountsByUid" parameterType="int" resultType="Account">
       select * from account where uid=#{uid} ;
    </select>
</mapper>
```

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。

```
<settings>
    <!-- 打开延迟加载的开关 -->
     <setting name="lazyLoadingEnabled" value="true" />
     <!-- 将积极加载改为消息加载即按需加载 -->
     <setting name="aggressiveLazyLoading" value="false"/>
</settings>
```

1.1.4.5 第六步：测试只加载用户信息

在 UserTest 类中加入测试方法，如下：

```
@Test
public void testUser() {
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession.getMapper(UserDao.class);

    List<User> userList = userDao.findUserList();
    for(User user :userList) {
        System.out.println(user.getId()+","+user.getUsername());
    }
    sqlSession.close();
}
```

测试结果如下：

```
Created connection 885851948.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@34cd072c]
==>  Preparing: select * from user;
==> Parameters:
<==      Total: 4
```

我们发现并没有加载 Account 账户信息。

1.1.4.6 第七步：测试加载用户信息同时还加载账户列表

```
@Test
public void testUser() {
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession.getMapper(UserDao.class);

    List<User> userList = userDao.findUserList();
    for(User user :userList) {
        System.out.println(user.getId()+","+user.getUsername());
        List<Account> accList = user.getAccList();
        for(Account acc :accList) {
            System.out.println(acc.getMoney());
        }
    }
    sqlSession.close();
}
```

加载账户信息,它会导致再次发出SQL语句加载Account账户信息

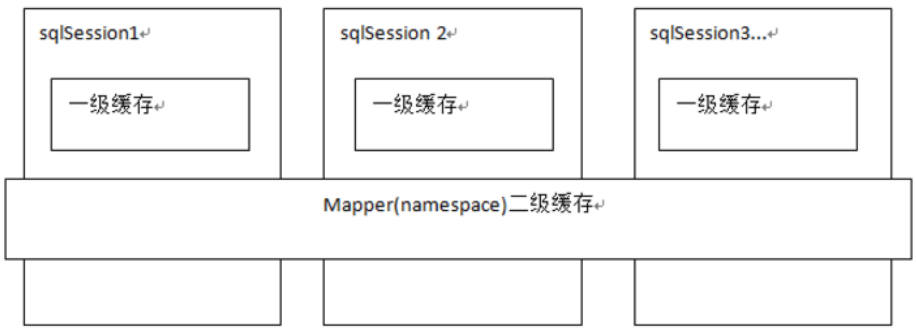
测试结果如下：

```
Created connection 885851948.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.
==>  Preparing: select * from user;
==> Parameters:
<==      Total: 4

==>  Preparing: select * from account where uid=? ;
==> Parameters: 41(Integer)
<==      Total: 2



==>  Preparing: select * from account where uid=? ;
==> Parameters: 42(Integer)
<==      Total: 0

==>  Preparing: select * from account where uid=? ;
==> Parameters: 43(Integer)
<==      Total: 0

==>  Preparing: select * from account where uid=? ;
==> Parameters: 45(Integer)
```

第2章 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。



2.1 Mybatis 一级缓存

2.1.1 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。

2.1.1.1 第一步：编写 UserDao 接口

```
public interface UserDao {
    public User findUserById(Integer id);
}
```

2.1.1.2 第二步：编写 UserDao.xml 映射文件

```
<mapper namespace="com.itheima.dao.UserDao">
    <select id="findUserById" parameterType="int" resultType="user">
       select * from user where id=#{id};
    </select>
</mapper>
```

2.1.1.3 第三步：编写测试方法

在 UserTest 类中编写测试方法

```
@Test
public void testUser() {
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User user1 = userDao.findUserById(41);
    User user2 = userDao.findUserById(41);
    System.out.println(user1);
    //不需要再次从数据库中查询，发现user1和user2的地址相同，是同一个对象
    System.out.println(user2);
}
```

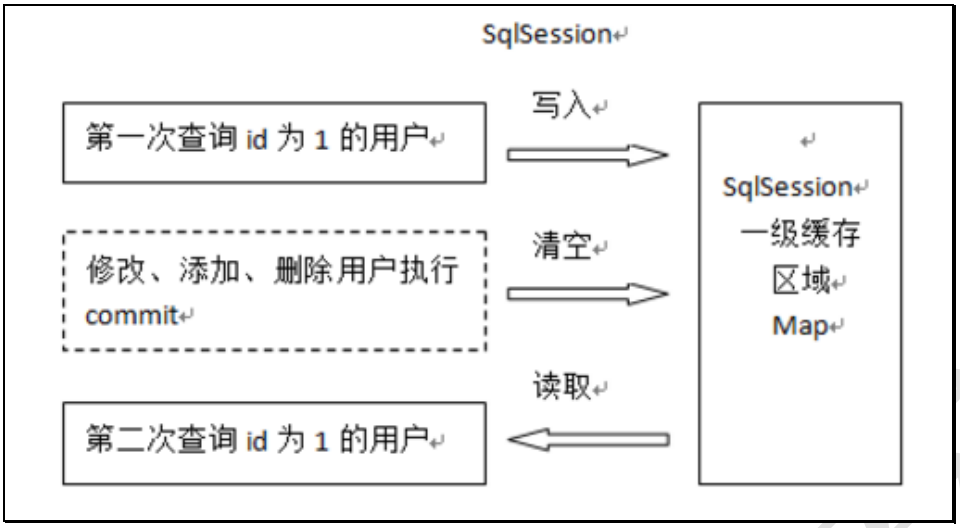
测试结果如下：

```
Opening JDBC Connection
Created connection 1150538133.
Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@4493d195]
==>  Preparing: select * from user where id=?;
==> Parameters: 41(Integer)
<==      Total: 1
com.itheima.domain.User@42f93a98
com.itheima.domain.User@42f93a98
```

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。

2.1.2 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。

2.1.3 测试一级缓存的清空

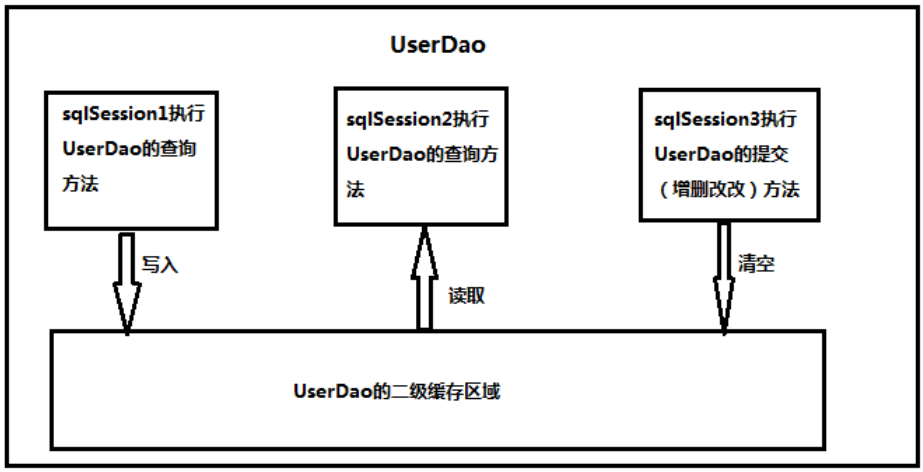
```
@Test
public void testUser() {
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User user1 = userDao.findUserById(41);
    User user2 = userDao.findUserById(41);
    System.out.println(user1);
    //不需要再次从数据库中查询，发现user1和user2的地址相同，是同一个对象
    System.out.println(user2);

    sqlSession.close();//sqlSession关闭后，一级缓存清空了
    SqlSession sqlSession2 = sqlSessionFactory.openSession();
    UserDao userDao2 = sqlSession2.getMapper(UserDao.class);//再次从数据库中读取数据
    User user3 = userDao2.findUserById(41);
    System.out.println(user3);
}
```

当执行 sqlSession.close()后，再次获取 sqlSession 并查询 id=41 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。

2.2 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。

2.2.1 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。

2.2.2 二级缓存的开启与关闭

2.2.2.1 第一步：在 SqlMapConfig.xml 文件开启二级缓存

```
<settings>
    <!-- 二级缓存开关 -->
    <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。

2.2.2.2 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。

UserDao.xml

```xml
<?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.itheima.dao.UserDao">
    <cache />
    <select id="findUserById" parameterType="int" resultType="user">
        select * from user where id=#{id};
    </select>
</mapper>
```

2.2.2.3 第三步：配置 statement 上面的 useCache 属性

```xml
<mapper namespace="com.itheima.dao.UserDao">
    <cache />
    <select id="findUserById" parameterType="int" resultType="user" useCache="true">
        select * from user where id=#{id};
    </select>
</mapper>
```

将 UserDao.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。

2.2.3 二级缓存测试

```java
@Test
public void testUser() {
    SqlSession sqlSession1 = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession1.getMapper(UserDao.class);
    User user1 = userDao.findUserById(41);

    sqlSession1.commit();//一级缓存清空了

    SqlSession sqlSession2 = sqlSessionFactory.openSession();
    userDao = sqlSession2.getMapper(UserDao.class);
    User user2 = userDao.findUserById(41);//没有发出sql语句,数据只能来自二级缓存
    sqlSession2.close();
}
```

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。

2.2.4 二级缓存注意事项

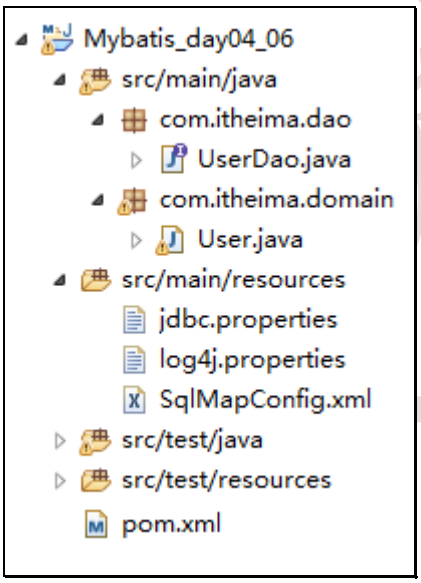
当我们在使用二级缓存时，所缓存的类一定要实现 java.io.Serializable 接口，这种就可以使用序列化方式来保存对象。
如下图：

User.java

```
2
3 import java.io.Serializable;
6
7 public class User implements Serializable {
8     private int id;
9     private String username;// 用户姓名
10    private String sex;// 性别
11    private Date birthday;// 生日
12    private String address;// 地址
```

第3章 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及延迟加载。
工程目录结构如下：



3.1 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。

3.1.1 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

There's one more trick to Mapper classes like BlogMapper. Their mapped statements don't need to be mapped with XML at all. Instead they can use Java Annotations. For example, the XML above could be eliminated and replaced with:

```
package org.mybatis.example;
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}
```

3.1.2 使用注解方式开发 UserDao 接口

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

UserDao.java

```
import java.util.List;
/**
 * 基于注解的映射方式，实现对数据的增删改查，将sql语句直接写在注解的括号中
 * 这是一个接口，其不需要类去实现它
 * 下边分别是插入，删除，修改，查询一个记录，查询所有的记录
 *
 */
public interface UserDao {
    @Insert("insert into user(username,sex,birthday,address) "
            + "values(#{username},#{sex},#{birthday},#{address})")
    public void saveUser(User user);

    @Delete("delete from user where id=#{id}")
    public void deleteById(Integer id);

    @Update("update user set username=#{username},birthday=#{birthday},"
            + "sex=#{sex},address=#{address} where id=#{id}")
    public void updateUser(User user);

    @Select("select * from user where id=#{id}")
    public User findUserById(Integer id);

    @Select("select * from user")
    public List<User> findAllUsers();
```

通过注解方式，我们就不需要再去编写 UserDao.xml 映射文件了。

3.1.3 修改 SqlMapConfig 配置文件

因为不存在 UserDao.xml 文件了，这样我们就不需要在 Mybatis 配置文件中加载 UserDao.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 UserDao 接口就可以了。

```
<!-- 加载映射接口 -->
<mappers>
  <mapper class="com.itheima.dao.UserDao"/>
</mappers>
```

3.1.4 编写测试方法

实现新增的注解开发测试

```
@Test
public void testInsertUser() {
    SqlSession sqlSession = sqlSessionFactory.openSession(true);
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User user = new User();
    user.setUsername("黑马 程序员");
    user.setSex("男");
    user.setBirthday(new Date());
    user.setAddress("北京顺义");
    userDao.saveUser(user);
    sqlSession.close();
}
```

实现删除的注解开发测试

```
@Test
public void testDeletetUser() {
    SqlSession sqlSession = sqlSessionFactory.openSession(true);
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    userDao.deleteById(47);//删除刚才新增的记录
    sqlSession.close();
}
```

实现更新的注解开发测试

```
@Test
public void testUpdateUser() {
    SqlSession sqlSession = sqlSessionFactory.openSession(true);
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User user = new User();
    user.setId(48);
    user.setUsername("小马宝莉");
    user.setSex("女");
    user.setBirthday(new Date());
    user.setAddress("北京修正");
    userDao.updateUser(user);
    sqlSession.close();
}
```

实现根据 id 查询用户的测试

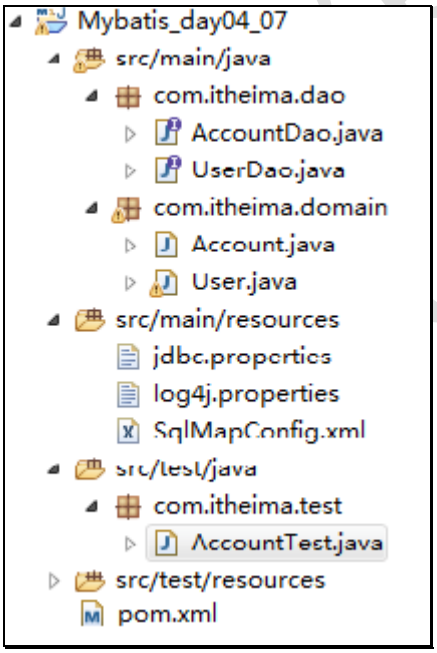
```
@Test
public void testFindUserById() {
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User user = userDao.findUserById(48);
    System.out.println(user);
    sqlSession.close();
}
```

实现查询所有用户信息

```
@Test
public void testFindAll() {
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    List<User> userList = userDao.findAllUsers();
    for(User user:userList) {
        System.out.println(user);
    }
    sqlSession.close();
}
```

3.2 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。实现后的工程结构如下：



3.2.1 复杂关系映射的注解说明

@Results 注解

代替的是标签<resultMap>

该注解中可以使用单个@Result 注解，也可以使用@Result 集合

@Results（{@Result（），@Result（）}）或@Results（@Result（））

@Resutl 注解

代替了 <id>标签和<result>标签

@Result 中 属性介绍：

column 数据库的列名

Property 需要装配的属性名

one 需要使用的@One 注解（@Result（one=@One）（））

many 需要使用的@Many 注解（@Result（many=@many）（））

@One 注解（一对一）

代替了<assocation>标签，是多表查询的关键，在注解中用来指定子查询返回单一对象。

@One 注解属性介绍：

select 指定用来多表查询的 sqlmapper

fetchType 会覆盖全局的配置参数 lazyLoadingEnabled。。

使用格式：

```
@Result(column=" ",property="",one=@One(select=""))
```

@Many 注解（多对一）

代替了<Collection>标签,是是多表查询的关键，在注解中用来指定子查询返回对象集合。

注意：聚集元素用来处理“一对多”的关系。需要指定映射的 Java 实体类的属性，属性的 javaType（一般为 ArrayList）但是注解中可以不定义；

使用格式：

```
@Result(property="",column="",many=@Many(select=""))
```

3.2.2 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）

3.2.2.1 添加 User 实体类及 Account 实体类

Account 实体类

Account.java | User.java

```
1 package com.itheima.domain;
2
3 import java.io.Serializable;
4
5 public class Account implements Serializable {
6     private int id;
7     private int uid;
8     private double money;
9     private User user ;
```

关联该账户的用户信息

User 实体类

```
import java.io.Serializable;
public class User implements Serializable {
    private int id;
    private String username;// 用户姓名
    private String sex;// 性别
    private Date birthday;// 生日
    private String address;// 地址
```

3.2.2.2 添加 UserDao 接口及 AccountDao 接口

UserDao 接口

```
public interface UserDao {
    @Select("select * from user where id = #{id}")
    public User findUserById(Integer id);
}
```

AccountDao 接口

```
public interface AccountDao {
    //查询账户信息，还要加载该账户所分配的用户信息
    public List<Account> findAccountList();
}
```

3.2.2.3 在 AccountDao 接口中添加注解实现复杂查询

AccountDao.java

```
1  package com.itheima.dao;
2
3  import java.util.List;
13
14 public interface AccountDao {
15     //查询账户信息，还要加载该账户所分配的用户信息
16     @Select("select * from account")
17     @Results({
18         @Result(id=true,property="id",column="id"),
19         @Result(property="uid",column="uid"),
20         @Result(property="money",column="money"),
21         @Result(property="user",column="uid",javaType=User.class,
22               one=@One(select="com.itheima.dao.UserDao.findUserById",fetchType=FetchType.LAZY))
23     })
24     public List<Account> findAccountList();
25 }
```

实现了mapper映射文件中的resultMap效果

这是来自于UserDao接口中的findUserById()方法的调用，用于查询指定账户下面的User信息

实现懒加载

3.2.2.4 测试一对一关联及延迟加载

```
@Test
public void testInsertUser() {
    SqlSession sqlSession = sqlSessionFactory.openSession(true);
    AccountDao accountDao = sqlSession.getMapper(AccountDao.class);
    List<Account> accList = accountDao.findAccountList();
    for(Account acc :accList) {
        System.out.println(acc.getId()+","+acc.getMoney());
        //采用了延迟加载用户对象，如此将此行和后面一行注掉，就不会有sql语句执行查询用户
        User user = acc.getUser();
        System.out.println(user.getAddress());
    }
    sqlSession.close();
}
```

3.2.3 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。

3.2.3.1 User 实体类及 Account 实体类

User 实体类

```
public class User implements Serializable {
    private int id;
    private String username;// 用户姓名
    private String sex;// 性别
    private Date birthday;// 生日
    private String address;// 地址
    private List<Account> accList;//保存多个账户信息
```

Account 实体类

```
public class Account implements Serializable {
    private int id;
    private int uid;
    private double money;
```

3.2.3.2 UserDao 接口及 AccountDao 接口

根据 uid 查询账户信息

```
public interface AccountDao {
    @Select("select * from account where uid=#{uid}")
    public List<Account> findAccountListByUid(Integer uid);
```

实现查询用户信息时，关联加载他的账户列表，并且要求使用延迟加载

```
public interface UserDao {
    @Select("select * from user")
    @Results({
        @Result(id=true,property="id",column="id"),
        @Result(property="username",column="username"),
        @Result(property="birthday",column="birthday"),
        @Result(property="sex",column="sex"),
        @Result(property="address",column="address"),
        @Result(property="accList",column="id",javaType=List.class,
                many=@Many(select="com.itheima.dao.AccountDao.findAccountListByUid",fetchType=FetchType.LAZY))
    })
    public List<User> findUserList();
```

它是User类中的List集合名称，用于存放多个Account账户信息
用户的id,需要根据这个id去加载他的账户信息
代表accList属性的类型,List是集合类型
延迟加载
加载账户Account使用该包下的查询语句
findAccountListByUid来招待查询

@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

3.2.3.3 添加测试方法

```
@Test
public void testInsertUser() {
    SqlSession sqlSession = sqlSessionFactory.openSession(true);
    AccountDao accountDao = sqlSession.getMapper(AccountDao.class);
    List<Account> accList = accountDao.findAccountList();
    for(Account acc :accList) {
        System.out.println(acc.getId()+","+acc.getMoney());
        //采用了延迟加载用户对象，如此将此行和后面一行注掉，就不会有sql语句执行查询用户
        User user = acc.getUser();
        System.out.println(user.getAddress());
    }
    sqlSession.close();
}
```

第4章 Mybatis 课程总结

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。做一个综合案例！

传智播客