01. 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 什么是延迟加载?

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。

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。
>
> mybatis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。

1.3 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。

1.3.1 账户的持久层DAO接口

```
public interface IAccountDao {
    /**
     * 查询所有账户，同时获取账户的所属用户名称以及它的地址信息
     * @return
     */
    public List<Account> findAll();
}
```

1.3.2 账户的持久层映射文件

```
<?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">
<!--namespace命名空间写当前你想映射的接口-->
<mapper namespace="com.dgut.dao.IAccountDao">
    <!-- 建立对应关系 -->
    <resultMap id="findAllMap" type="account">
        <id property="id" column="accountId"/>
        <result property="money" column="money"/>
        <!-- 它是用于指定从表方的引用实体属性的
              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
              -->
        <association property="user" javaType="user"
select="com.dgut.dao.IUserDao.findById" column="uid">
        </association>
    </resultMap>

    <select id="findAll" resultMap="findAllMap">
        SELECT  * FROM account
    </select>

</mapper>
```

1.3.3 用户的持久层接口和映射文件

```
public interface IUserDao {
/**
  * 根据id查询
  * @param userId
  * @return */
  User findById(Integer userId);
}

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

1.4 开启Mybatis的延迟加载策略

```
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。
<!-- 开启延迟加载的支持 -->
<settings>
    <setting name="lazyLoadingEnabled" value="true"/>
    <setting name="aggressiveLazyLoading" value="false"/>
  <!-- 指定哪个对象的方法触发一次延迟加载。  用逗号分隔的方法列表。
equals,clone,hashCode,toString -->
        <setting name="lazyLoadTriggerMethods" value=""/>
 </settings>
```

1.5 编写测试只查账户信息不查用户信息。

```
  @Test
    public void findAll(){
        List<Account> all = accountDao.findAll();
    }
```

```
2019-09-05 23:08:07,739 56    [          main] DEBUG .apache.ibatis.io.ResolverUtil  - Checking to see if class com.dgut.pojo.QueryUserVo matches criteria
2019-09-05 23:08:07,740 57    [          main] DEBUG .apache.ibatis.io.ResolverUtil  - Checking to see if class com.dgut.pojo.QueryVO matches criteria [is
2019-09-05 23:08:07,768 85    [          main] DEBUG source.pooled.PooledDataSource  - PooledDataSource forcefully closed/removed all connections.
2019-09-05 23:08:07,768 85    [          main] DEBUG source.pooled.PooledDataSource  - PooledDataSource forcefully closed/removed all connections.
2019-09-05 23:08:07,768 85    [          main] DEBUG source.pooled.PooledDataSource  - PooledDataSource forcefully closed/removed all connections.
2019-09-05 23:08:07,769 86    [          main] DEBUG source.pooled.PooledDataSource  - PooledDataSource forcefully closed/removed all connections.
2019-09-05 23:08:07,890 207   [          main] DEBUG ansaction.jdbc.JdbcTransaction  - Opening JDBC Connection
2019-09-05 23:08:08,107 424   [          main] DEBUG source.pooled.PooledDataSource  - Created connection 796667727.
2019-09-05 23:08:08,112 429   [          main] DEBUG m.dgut.dao.IAccountDao.findAll  - ==>  Preparing: SELECT * FROM account
2019-09-05 23:08:08,156 473   [          main] DEBUG m.dgut.dao.IAccountDao.findAll  - ==> Parameters:
2019-09-05 23:08:08,244 561   [          main] DEBUG m.dgut.dao.IAccountDao.findAll  - <==      Total: 3
2019-09-05 23:08:08,245 562   [          main] DEBUG ansaction.jdbc.JdbcTransaction  - Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@2f7c2f4f]
2019-09-05 23:08:08,245 562   [          main] DEBUG source.pooled.PooledDataSource  - Returned connection 796667727 to pool.
```

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。

1.4 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结点中配置延迟加载策略。
>
> 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。

1.4.1 在User实体类中加入List<Account>属性

```
public class User {
    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;
    private List<Account> accounts;
    ...
  }
```

1.4.2 编写用户和账户持久层接口的方法

```
IUserDao.java

/**
 * 查询所有用户，同时获取出每个用户下的所有账户信息(延迟加载)
 */
public List<User> findUsersWithAccounts2();

IAccountDao.java
/**
 * 根据用户id查询账户信息
 * @param uid
 * @return
 */
List<Account> findByUid(Integer uid);
```

1.4.3 编写用户持久层映射配置

```
IUserDao.xml
 <resultMap id="findUsersWithAccountsMap2" type="user">
        <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
  select是用于指定查询账户的唯一标识（账户的dao全限定类名加上方法名称）
  column是用于指定使用哪个字段的值作为条件查询 -->
        <collection property="accounts" ofType="com.dgut.domain.Account"
select="com.dgut.dao.IAccountDao.findByUid" column="id">
        </collection>
    </resultMap>

    <select id="findUsersWithAccounts2" resultMap="findUsersWithAccountsMap2">
```

```
        select  * from user
    </select>


IAccountDao.xml
<select id="findByUid" resultType="account">
      select * from account where uid = #{uid}
  </select>
```

> **<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

1.4.4 测试

```
@Test
public void findAllUsers(){
    List<User> usersWithAccounts2 = userDao.findUsersWithAccounts2();
//        System.out.println(usersWithAccounts2.get(0).getAccounts());
}
```

02. 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 编写用户持久层Dao接口

```
     * 根据id查询相应的用户
     *
     * @param id
     * @return 用户
     */
    public User findById(Integer id);
```

2.1.1.2 编写用户持久层映射文件

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

2.1.1.3 编写测试方法

```java
@Test
    public void testFindById() {
        User user = userDao.findById(41);
        System.out.println("第一次查询的用户：" + user);
        User user2 = userDao.findById(41);
        System.out.println("第二次查询用户：" + user2);
        System.out.println(user == user2);
    }
```

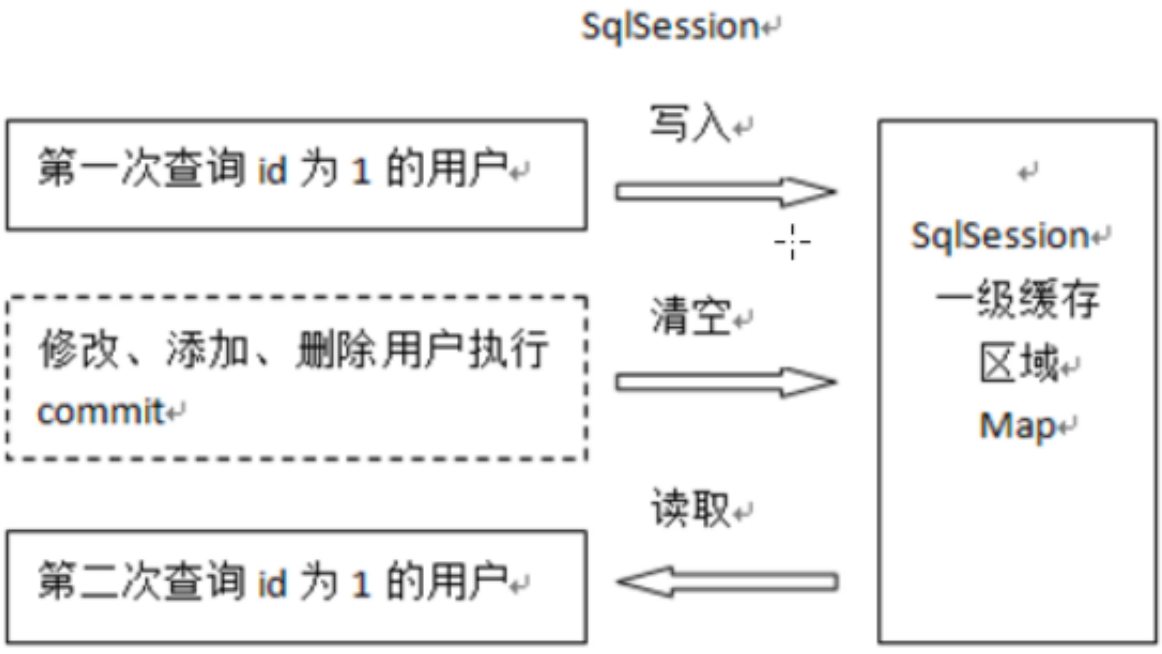
2.1.1.4 测试结果

```
2019-09-06 10:58:40,878 40     [           main] DEBUG .apache.ibatis.io.ResolverUtil  - Checking to see if class com.dgut.pojo.QueryVO matches criteria [is a
2019-09-06 10:58:40,898 60     [           main] DEBUG source.pooled.PooledDataSource  - PooledDataSource forcefully closed/removed all connections.
2019-09-06 10:58:40,898 60     [           main] DEBUG source.pooled.PooledDataSource  - PooledDataSource forcefully closed/removed all connections.
2019-09-06 10:58:40,898 60     [           main] DEBUG source.pooled.PooledDataSource  - PooledDataSource forcefully closed/removed all connections.
2019-09-06 10:58:40,899 61     [           main] DEBUG source.pooled.PooledDataSource  - PooledDataSource forcefully closed/removed all connections.
2019-09-06 10:58:40,987 149    [           main] DEBUG ansaction.jdbc.JdbcTransaction  - Opening JDBC Connection
2019-09-06 10:58:41,143 305    [           main] DEBUG source.pooled.PooledDataSource  - Created connection 479397964.
2019-09-06 10:58:41,147 309    [           main] DEBUG com.dgut.dao.IUserDao.findById  - ==>  Preparing: select * from user where id = ?
2019-09-06 10:58:41,177 339    [           main] DEBUG com.dgut.dao.IUserDao.findById  - ==> Parameters: 41(Integer)
2019-09-06 10:58:41,198 360    [           main] DEBUG com.dgut.dao.IUserDao.findById  <==      Total: 1
第一次查询的用户：User{id=41, username='老王', birthday=Wed Feb 28 07:47:08 CST 2018, sex='男', address='东莞南城', accounts=null, roles=null}
第二次查询用户：User{id=41, username='老王', birthday=Wed Feb 28 07:47:08 CST 2018, sex='男', address='东莞南城', accounts=null, roles=null}
true
2019-09-06 10:58:41,201 363    [           main] DEBUG ansaction.jdbc.JdbcTransaction  - Closing JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@1c93084c]
2019-09-06 10:58:41,201 363    [           main] DEBUG source.pooled.PooledDataSource  - Returned connection 479397964 to pool.
```

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

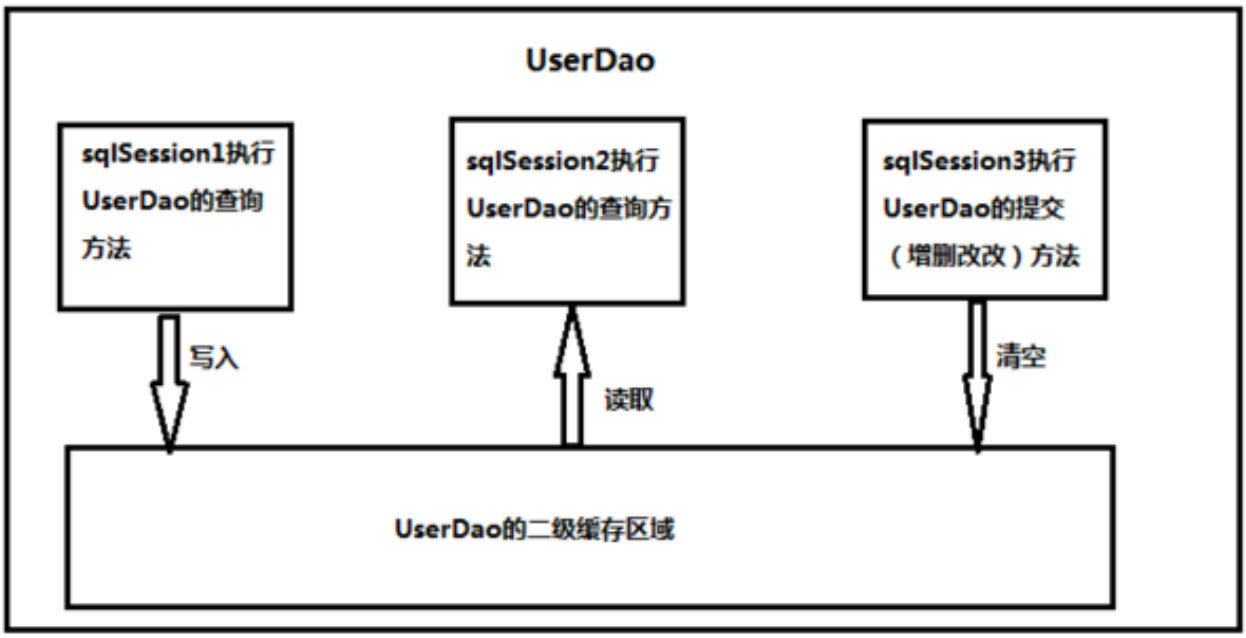
```
    @Test
    public void testFindById() {
        User user = userDao.findById(41);
        System.out.println("第一次查询的用户: " + user);
        //清空缓存
        sqlSession.clearCache();
        User user2 = userDao.findById(41);
        System.out.println("第二次查询用户: " + user2);
        System.out.println(user == user2);
    }


   @Test
    public void testFindById() {
        User user = userDao.findById(41);
        System.out.println("第一次查询的用户: " + user);
        user.setUsername("yyy");
      //更新用户信息
        userDao.updateUser(user);
        User user2 = userDao.findById(41);
        System.out.println("第二次查询用户: " + user2);
        System.out.println(user == user2);
    }
```

2.2Mybatis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>
    <!-- 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。

2.2.2.2 第二步：配置相关的**Mapper**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">
<!--namespace命名空间写当前你想映射的接口-->
<mapper namespace="com.dgut.dao.IUserDao">
    <!-- 开启二级缓存的支持 -->
    <cache/>
</mapper>
```

2.2.2.3 第三步：配置**statement**上面的**useCache**属性

```
 <select id="findById" parameterType="Integer" resultType="user"
useCache="true">
        select * from user where id = #{id}
  </select>
```

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

```
@Test public void testSecondLevelCache(){
      SqlSession sqlSession1 = sessionFactory.openSession();
      IUserDao dao1 = sqlSession1.getMapper(IUserDao.class);
      User user1 = dao1.findById(41);
      System.out.println(user1);
      sqlSession1.close();
 //一级缓存消失 SqlSession
      SqlSession sqlSession2 = sessionFactory.openSession();
      IUserDao dao2 = sqlSession2.getMapper(IUserDao.class);
      User user2 = dao2.findById(41);
      System.out.println(user2);
      sqlSession2.close();
      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。

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。

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。

01.mybatis的常用注解说明

```
@Insert:实现新增
@Options:主键映射配置
@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
@ResultMap:实现引用@Results定义的封装
@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
```

02.使用Mybatis注解实现基本CRUD

2.1 编写实体类

```
public class User2 {
    private Integer uid;
    private String userName;
    private Date u_birthday;
    private String u_sex;
    private String u_address;

    public Integer getUid() {
        return uid;
    }

    public void setUid(Integer uid) {
        this.uid = uid;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public Date getU_birthday() {
        return u_birthday;
    }

    public void setU_birthday(Date u_birthday) {
        this.u_birthday = u_birthday;
    }
```

```
    public String getU_sex() {
        return u_sex;
    }

    public void setU_sex(String u_sex) {
        this.u_sex = u_sex;
    }

    public String getU_address() {
        return u_address;
    }

    public void setU_address(String u_address) {
        this.u_address = u_address;
    }

    @Override
    public String toString() {
        return "User2{" +
                "uid=" + uid +
                ", userName='" + userName + '\'' +
                ", u_birthday=" + u_birthday +
                ", u_sex='" + u_sex + '\'' +
                ", u_address='" + u_address + '\'' +
                '}';
    }
}
```

注意：

此处我们故意和数据库表的列名不一致。

2.2 使用注解方式开发持久层接口

```
public interface IAnnoUserDao {

    /**
     * 查询所有用户
     * * @return
     */
    @Select(value = {"select * from user"})
    @Results(id = "bcd", value = {
            @Result(column = "id", property = "uid", id = true),
            @Result(column = "username", property = "userName"),
            @Result(column = "sex", property = "u_sex"),
            @Result(column = "address", property = "u_address"),
            @Result(column = "birthday", property = "u_birthday")
    })
```

```
    public List<User2> findAll();


    /**
     * 保存用户
     * @param user
     * @return 影响行数
     */
    @Insert(value = "insert into user(username,sex,birthday,address)values(#
{username},#{sex},#{birthday},#{address})")
    @Options(keyColumn = "id",keyProperty = "id",useGeneratedKeys = true)
    public int saveUser(User user);

  /**
     * 删除用户
     *
     * @param userId * @return
     */
    @Delete("delete from user where id = #{uid} ")
    int deleteUser(Integer userId);

   /**
     * 查询使用聚合函数
     * * @return
     */
    @Select("select count(*) from user ")
    int findTotal();

    /**
     * 模糊查询
     * @param name
     * @return
     */
    @Select("select * from user where username like #{username} ")
    List<User> findByName(String name);


  /**
     * 查询用户时，可以同时得到用户下所包含的账户信息(注解)
     * @return 用户
     */
    @Select("select * from user")
    @Results(value = {
            @Result(property = "id",column = "id", id = true),
            @Result(property = "username",column = "username"),
            @Result(property = "address",column = "address"),
            @Result(property = "sex",column = "sex"),
```

```
            @Result(property = "accounts",many = @Many(select =
"com.dgut.dao.IAccountDao.findByUid",fetchType = FetchType.LAZY),column =
"id")
    })
    public List<User> findUsersWithAccountsAnno();
}
```

试一试：

- 用注解的方式实现
  - 新增用户返回用户的id
  - 查找指定的用户，测试注解的二级缓存如何实现?
  - 查找所有的用户及其关联的用户信息（采用嵌套结果方式）
  - 查询所有的用户及其关联的角色信息（采用嵌套查询的方式）
  - 基于注解动态sql的方式查询用户的（username，address）信息，其中username，address可为空