第三讲 动态 sql 与多表连接查询

一、简化编写的 SQL 片段

Sql 中可将重复的 sql 提取出来，使用时用 include 引用即可，最终达到 sql 重用的目的。

- 定义代码片段

```
<!-- 抽取重复的语句代码片段 -->
<sql id="defaultSql">
    select * from user
</sql>
```

- 引用代码片段

```
<!-- 配置查询所有操作 -->
 <select id="findAll" resultType="user">
  <include refid="defaultSql"></include>
 </select>
<!-- 根据 id 查询 -->
 <select id="findById" resultType="UsEr" parameterType="int">
 <include refid="defaultSql"></include>
    where id = #{uid}
 </select>
```

二、动态 sql

1. <if>标签

需求：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 不同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。

- 持久层 Dao 接口

```
/**
 * 根据用户信息，查询用户列表
  * @param user
 * @return
  */
 List<User> findByUser(User user);
```

- 持久层 Dao 映射配置

```
<select id="findByUser" resultType="user" parameterType="user">
   select * from user where 1=1
 <if test="username!=null and username != '' ">
    and username like #{username}
  </if>
 <if test="address != null">
    and address like #{address}
</if>
</select>
```

注意:<if>标签的 test 属性中写的是对象的属性名,如果是包装类的对象要使用 OGNL 表达式的写法。 另外要注意 where 1=1 的作用~!

- 测试

```
 @Test
  public void testFindByUser()  {
    User u = new User();
    u.setUsername("%王%");
    u.setAddress("%顺义%");
    //6.执行操作
    List<User> users = userDao.findByUser(u);
    for(User user : users) {
     System.out.println(user);
    }
     }
```

2. <where>标签

为了简化上面 where 1=1 的条件拼装，我们可以采用<where>标签来简化开发。

持久层 Dao 映射配置

```
<!-- 根据用户信息查询 -->
 <select id="findByUser" resultType="user" parameterType="user">
  <include refid="defaultSql"></include>
  <where>
  <if test="username!=null and username != '' ">
    and username like #{username}
   </if>
  <if test="address != null">
    and address like #{address}
   </if>
  </where>
 </select>
```

3. <foreach>标签

需求：传入多个 id 查询用户信息，用下边两个 sql 实现：

SELECT * FROM USERS WHERE username LIKE '%张%' AND (id =10 OR id =89 OR id=16)

SELECT * FROM USERS WHERE username LIKE '%张%' AND id IN (10,89,16)

这样我们在进行范围查询时，就要将一个集合中的值，作为参数动态添加进来。 这样我们将如何进行参数的传递

- 在 QueryVo 中加入一个 List 集合用于封装参数

```
public class QueryVo implements Serializable {

 private List<Integer> ids;
 public List<Integer> getIds() {
   return ids;
 }

 public void setIds(List<Integer> ids) {
   this.ids = ids;
 }
     }
```

- 持久层 Dao 接口

```
List<User> findInIds(QueryVo vo);
```

- 持久层 Dao 映射配置

```
<!-- 查询所有用户在 id 的集合之中 -->
  <select id="findInIds" resultType="user" parameterType="queryvo">
  <!--   select * from user where id in (1,2,3,4,5); -->
  <include refid="defaultSql"></include>
    <where>
    <if test="ids != null and ids.size() > 0">
      <foreach collection="ids" open="id in ( " close=")" item="uid"   separator=",">
       #{uid}
      </foreach>
    </if>
    </where>
  </select>
```

SQL 语句：

select 字段 from user where id in (?)

<foreach>标签用于遍历集合，它的属性：

collection:代表要遍历的集合元素，注意编写时不要写#{}

open:代表语句的开始部分

close:代表结束部分

item:代表遍历集合的每个元素，生成的变量名

sperator:代表分隔符

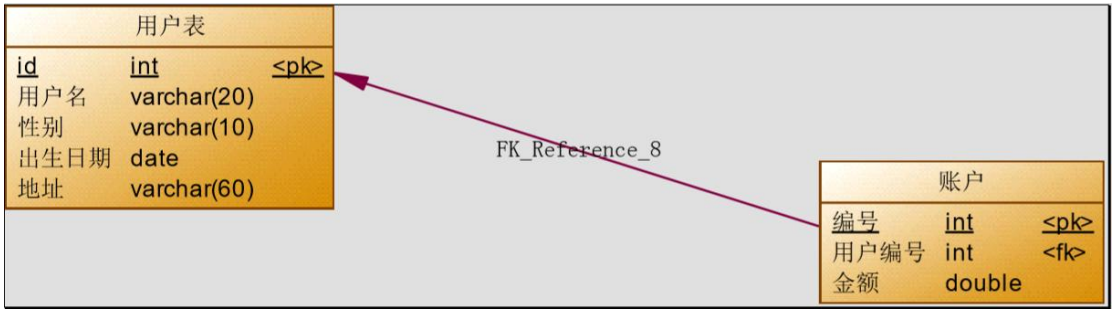
- 编写测试方法

```
@Test
public void testFindInIds()  {
      QueryVo vo = new QueryVo();
    List<Integer> ids = new ArrayList<Integer>();
    ids.add(41);
    ids.add(42);
    ids.add(43);
    ids.add(46);
    ids.add(57);
    vo.setIds(ids);
  //6.执行操作
    List<User> users = userDao.findInIds(vo);
```

```
    for(User user : users) {
      System.out.println(user);
    }
    }
```

三、 Mybatis 多表查询

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. 一对一查询(多对一)

需求　查询所有账户信息，关联查询下单用户信息。

注意：　因为一个账户信息只能供某个用户使用，所以从查询账户信息出发关联查询用户信息为一对一查询。如果从用户信息出发查询用户下的账户信息则为一对多查询，因为一个用户可以有多个账户。

① 方式一

- 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

实现查询账户信息时，也要查询账户所对应的用户信息。

```
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 super.toString() + "       AccountUser [username=" + username + ",
address=" + address + "]";
  }
}
```

- 定义账户的持久层 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
    */
  List<AccountUser> findAll();
 }
```

- 定义 AccountDao.xml 文件中的查询配置信息

```
<?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="club.banyuan.dao.IAccountDao">
  <!-- 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

```
public class AccountTest {
      private InputStream in ;
   private SqlSessionFactory factory;
   private SqlSession session;
   private IAccountDao accountDao;
      @Test
   public void testFindAll()   {
      //6.执行操作
     List<AccountUser> accountusers = accountDao.findAll();
     for(AccountUser au : accountusers) {
      System.out.println(au);
     }
         }
 @Before
//在测试方法执行之前执行
   public void init()throws Exception {
     //1.读取配置文件
     in = Resources.getResourceAsStream("SqlMapConfig.xml");
     //2.创建构建者对象
     SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
     //3.创建 SqlSession 工厂对象
     factory = builder.build(in);
   //4.创建 SqlSession 对象
     session = factory.openSession();
   //5.创建 Dao 的代理对象
     accountDao = session.getMapper(IAccountDao.class);
   }
 @After//在测试方法执行完成之后执行
   public void destroy() throws Exception{
     session.commit();
   //7.释放资源
     session.close();
     in.close();
 }
 }
```

定义专门的 po 类作为输出类型，其中定义了 sql 查询结果集所有的字段。此方法较为简

单，企业中使用普遍。

② 方式二

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。

- 修改 Account 类

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 + "]";
  }
 }
```

- 修改 AccountDao 接口中的方法

```
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
   */
  List<Account> findAll();
}
```

注意：第二种方式，将返回值改 为了 Account 类型。 因为 Account 类中包含了一个 User 类的对象，它可以封装账户所对应的用户信息。

- 重新定义 AccountDao.xml 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="club.banyuan.dao.IAccountDao">
 <!-- 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>
```

- 在 AccountTest 类中加入测试方法

```
 @Test
  public void testFindAll()  {
    List<Account> accounts = accountDao.findAll();
   for(Account au : accounts) {
    System.out.println(au);
    System.out.println(au.getUser());
   }
}
```

2. 一对多查询

需求:

查询所有用户信息及用户关联的账户信息。 分析: 用户信息和他的账户信息为一对多关系，并且查询过程中如果用户没有账户信息，此时也要将用户信息 查询出来，我们想到了左外连接查询比较合适。

- 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
```

id	username	birthday	sex	address	id	uid	money
41	老王	2018-02-27 17:47:08	男	北京	1	41	1000
41	老王	2018-02-27 17:47:08	男	北京	3	41	2000
42	小二王	2018-03-02 15:09:37	女	北京金燕龙	(NULL)	(NULL)	(NULL)
43	小二王	2018-03-04 11:34:34	女	北京金燕龙	(NULL)	(NULL)	(NULL)
45	传智播客	2018-03-04 12:04:06	男	北京金燕龙	2	45	1000

- User 类加入 List<Account>

```
public class User implements Serializable {


 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 + "]";
  }
}
```

- 用户持久层 Dao 接口中加入查询方法

```
List<User> findAll();
```

- 用户持久层 Dao 映射文件配置

```
<?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="club.banyuan.dao.IUserDao">
 <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>
```

collection 部分定义了用户关联的账户信息。表示关联查询结果集 property="accList" ：

关联查询的结果集存储在 User 对象的上哪个属性。 ofType="account" ：

指定关联查询的结果集中的对象类型即 List 中的对象类型。此处可以使用别名，也可以使用全限定名。

- 测试方法

```
public class UserTest {
    private InputStream in ;
  private SqlSessionFactory factory;
  private SqlSession session;
  private IUserDao userDao;
    @Test
  public void testFindAll()  {
    //6.执行操作
   List<User> users = userDao.findAll();
   for(User user : users) {
    System.out.println("-------每个用户的内容---------");
    System.out.println(user);
    System.out.println(user.getAccounts());
   }
  }
 @Before
//在测试方法执行之前执行
 public void init()throws Exception {
   //1.读取配置文件
   in = Resources.getResourceAsStream("SqlMapConfig.xml");
   //2.创建构建者对象
   SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
   //3.创建 SqlSession 工厂对象
   factory = builder.build(in);
  //4.创建 SqlSession 对象
   session = factory.openSession();
  //5.创建 Dao 的代理对象
```

```
    userDao = session.getMapper(IUserDao.class);   }
 @After//在测试方法执行完成之后执行
  public void destroy() throws Exception{
    session.commit();
  //7.释放资源
    session.close();
    in.close();
  }
 }
```

3. 多对多

① 实现 Role 到 User 多对多

- 用户与角色的多对多关系模型

在 MySQL 数据库中添加角色表，用户角色的中间表。

角色表

ID	ROLE_NAME	ROLE_DESC
1	院长	管理整个学院
2	总裁	管理整个公司
3	校长	管理整个学校

用户角色中间表

UID	RID
41	1
45	1
41	2

- 业务要求及实现 SQL

需求： 实现查询所有对象并且加载它所分配的用户信息。

分析： 查询角色我们需要用到 Role 表，但角色分配的用户的信息我们并不能直接找到用户信息，而是要通过中 间表(USER_ROLE 表)才能关联到用户信息。

下面是实现的 SQL 语句

```
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 implements Serializable {

    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 List<User> getUsers() {
          return users;
```

```
    }

    public void setUsers(List<User> users) {
        this.users = 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;
    }

    @Override
    public String toString() {
        return "Role{" +
                "roleId=" + roleId +
                ", roleName='" + roleName + '\'' +
                ", roleDesc='" + roleDesc + '\'' +
                '}';
    }
```

```
}
```

- 编写 Role 持久层接口

```
public interface IRoleDao {

    /**
     * 查询所有角色
      * @return
      */
     List<Role> findAll();
}
```

- 编写映射文件

```
<?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="club.banyuan.dao.IRoleDao">

    <!--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>
```

- 编写测试类

```
public class RoleTest {

    private InputStream in;
     private SqlSession sqlSession;
     private IRoleDao roleDao;

    @Before//用于在测试方法执行之前执行
     public void init()throws Exception{
         //1.读取配置文件，生成字节输入流
         in = Resources.getResourceAsStream("SqlMapConfig.xml");
         //2.获取
 SqlSessionFactory SqlSessionFactory factory = new
SqlSessionFactoryBuilder().build(in);
         //3.获取 SqlSession 对象
         sqlSession = factory.openSession(true);
         //4.获取 dao 的代理对象
         roleDao = sqlSession.getMapper(IRoleDao.class);
     }

    @After//用于在测试方法执行之后执行
     public void destroy()throws Exception{
         //提交事务
         // sqlSession.commit();
        //6.释放资源
         sqlSession.close();
         in.close();
     }

    /**
     * 测试查询所有
      */
     @Test
```

```
    public void testFindAll(){
        List<Role> roles = roleDao.findAll();
        for(Role role : roles){
            System.out.println("---每个角色的信息----");
            System.out.println(role);
            System.out.println(role.getUsers());
        }
    }
}
```

② 实现 User 到 Role 的多对多

从 User 出发，我们也可以发现一个用户可以具有多个角色，这样用户到角色的关系也还是一对多关系。这样 我们就可以认为 User 与 Role 的多对多关系，可以被拆解成两个一对多关系来实现。

四、 Mybatis 延迟加载策略

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。

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。

2. 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。

- 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
   */
  List<Account> findAll();
 }
```

- 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">

<mapper namespace="com.itheima.dao.IAccountDao">
  <!-- 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"
select="com.itheima.dao.IUserDao.findById" column="uid">
   </association>
  </resultMap>
 <select id="findAll" resultMap="accountMap">
   select * from account
  </select>
 </mapper>
```

- 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);
   }
```

```
<?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.IUserDao">
 <!-- 根据 id 查询 -->
   <select id="findById" resultType="user" parameterType="int" >
    select * from user where id = #{uid}
   </select>
 </mapper>
```

- 开启 Mybatis 的延迟加载策略

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"/>
</settings>
```

- 编写测试只查账户信息不查用户信息

```
public class AccountTest {
      private InputStream in ;
   private SqlSessionFactory factory;
   private SqlSession session;
   private IAccountDao accountDao;
      @Test
   public void testFindAll()  {
      //6.执行操作
    List<Account> accounts = accountDao.findAll();
   }
 @Before//在测试方法执行之前执行
   public void init()throws Exception {
    //1.读取配置文件
```

```
    in = Resources.getResourceAsStream("SqlMapConfig.xml");
    //2.创建构建者对象
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    //3.创建 SqlSession 工厂对象
    factory = builder.build(in);
  //4.创建 SqlSession 对象
    session = factory.openSession();
  //5.创建 Dao 的代理对象
    accountDao = session.getMapper(IAccountDao.class);
  }
 @After//在测试方法执行完成之后执行
  public void destroy() throws Exception{
    //7.释放资源
    session.close();
  in.close();
  }
 }
```

```
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]
```

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。

3. 使用 Collection 实现延迟加载

需求： 完成加载用户对象时，查询该用户所拥有的账户信息。

- 在 User 实体类中加入 List<Account>属性

```
public class User implements Serializable {


 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 + "]";
  }
 }
```

- 编写用户和账户持久层接口的方法

```
 /**
  * 查询所有用户，同时获取出每个用户下的所有账户信
  * @return
   */
  List<User> findAll();


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

- 编写用户持久层映射配置

```
 <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 property="accounts" ofType="account"
        select="com.itheima.dao.IAccountDao.findByUid"        column="id">
    </collection>
  </resultMap>
 <!-- 配置查询所有操作 -->
  <select id="findAll" resultMap="userMap">
```

```
    select * from user
  </select>
```

- 编写账户持久层映射配置

```
<!-- 根据用户 id 查询账户信息 -->
 <select id="findByUid" resultType="account" parameterType="int">
  select * from account where uid = #{uid}
 </select>
```

- 测试只加载用户信息

```
public class UserTest {
      private InputStream in ;
   private SqlSessionFactory factory;
   private SqlSession session;
   private IUserDao userDao;
      @Test
 public void testFindAll()   {
      //6.执行操作
    List<User> users = userDao.findAll();
   }

 @Before//在测试方法执行之前执行
   public void init()throws Exception {
    //1.读取配置文件
    in = Resources.getResourceAsStream("SqlMapConfig.xml");
    //2.创建构建者对象
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    //3.创建 SqlSession 工厂对象
    factory = builder.build(in);
  //4.创建 SqlSession 对象
    session = factory.openSession();
  //5.创建 Dao 的代理对象
    userDao = session.getMapper(IUserDao.class);
   }
 @After//在测试方法执行完成之后执行
 public void destroy() throws Exception{
    session.commit();
```

```
    //7.释放资源
    session.close();
    in.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
```

发现没有加载 Account 账户信息

五、 Mybatis 缓存

1. Mybatis 一级缓存

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

- 编写用户持久层 Dao 接口

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

- 编写用户持久层映射文件

```
<?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">
<mapper namespace="com.itheima.dao.IUserDao">
   <!-- 根据 id 查询 -->
   <select id="findById" resultType="UsEr" parameterType="int" useCache="true">
    select * from user where id = #{uid}
   </select>
 </mapper>
```

- 编写测试方法

```
public class UserTest {
      private InputStream in ;
   private SqlSessionFactory factory;
   private SqlSession session;
   private IUserDao userDao;

 @Test
   public void testFindById()  {
    //6.执行操作
   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);
   }
 @Before//在测试方法执行之前执行
   public void init()throws Exception {
    //1.读取配置文件
    in = Resources.getResourceAsStream("SqlMapConfig.xml");
    //2.创建构建者对象
    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    //3.创建 SqlSession 工厂对象
    factory = builder.build(in);
   //4.创建 SqlSession 对象
    session = factory.openSession();
   //5.创建 Dao 的代理对象
    userDao = session.getMapper(IUserDao.class);  }
 @After//在测试方法执行完成之后执行
   public void destroy() throws Exception{
    //7.释放资源
    session.close();
   in.close();
   }
 }
```

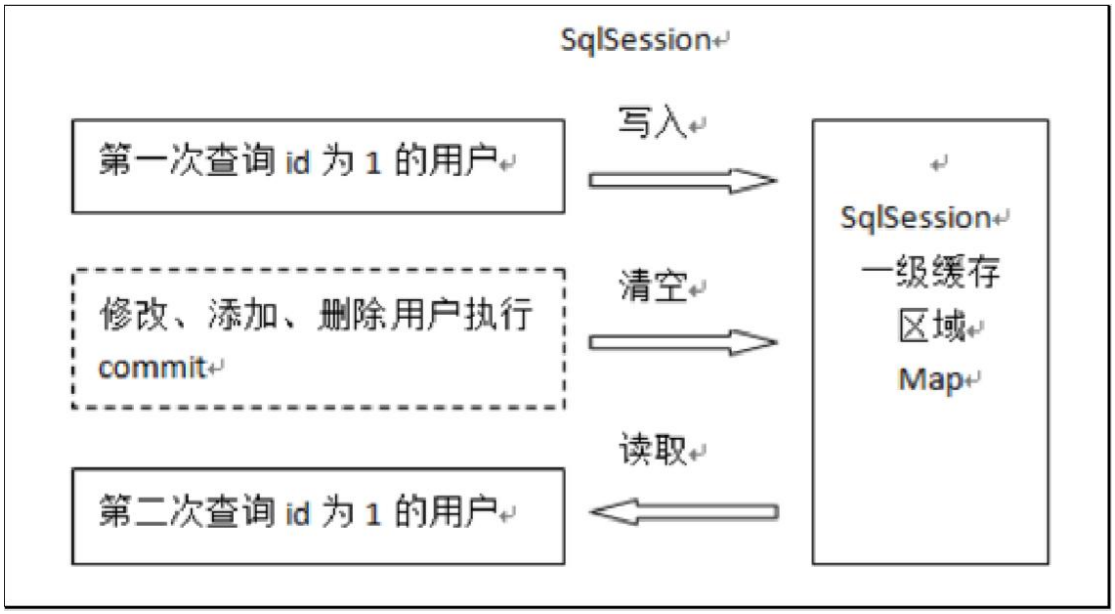
- 测试结果如下：

```
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
```

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。

- 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

```
/**
     * 测试一级缓存
      */
     @Test
     public void testFirstLevelCache(){          User user1 = userDao.findById(41);
```

```
        System.out.println(user1);
//            sqlSession.close();
          //再次获取 SqlSession 对象
//            sqlSession = factory.openSession();

          sqlSession.clearCache();
//此方法也可以清空缓存
          userDao = sqlSession.getMapper(IUserDao.class);

          User user2 = userDao.findById(41);
           System.out.println(user2);
           System.out.println(user1 == user2);
    }

    /**
     * 测试缓存的同步
      */
     @Test
    public void testClearlCache(){
          //1.根据 id 查询用户
          User user1 = userDao.findById(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.findById(41);
          System.out.println(user2);
          System.out.println(user1 == user2);
     }

     /**
     * 测试缓存的同步
```

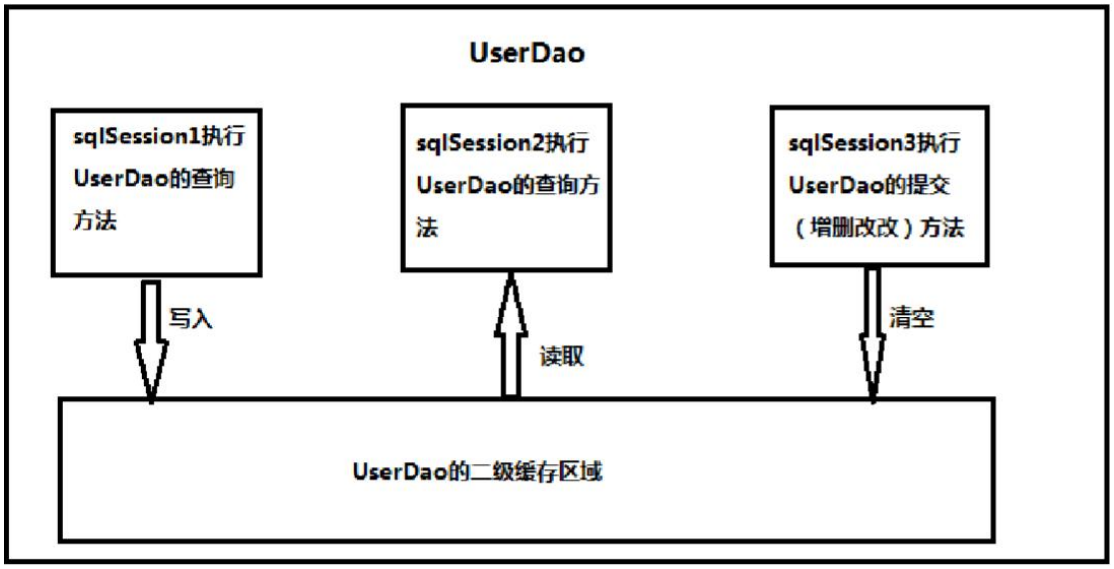
```
     */
    @Test
    public void testClearlCache(){
        //1.根据 id 查询用户
        User user1 = userDao.findById(41);
        System.out.println(user1);

        //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.findById(41);
        System.out.println(user2);
        System.out.println(user1 == user2);
    }
```

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. 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。

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 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

<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。

```
<?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.IUserDao">
  <!-- 开启二级缓存的支持 -->
  <cache></cache>
 </mapper>
```

- 第三步：配置 statement 上面的 useCache 属性

```
<!-- 根据 id 查询 -->
 <select id="findById" resultType="user" parameterType="int" useCache="true">
  select * from user where id = #{uid}
</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。

- 二级缓存测试

```
public class SecondLevelCacheTest {

    private InputStream in;
     private   SqlSessionFactory factory;

    @Before//用于在测试方法执行之前执行
     public void init()throws Exception{
         //1.读取配置文件，生成字节输入流
          in = Resources.getResourceAsStream("SqlMapConfig.xml");
         //2.获取 SqlSessionFactory
          factory = new SqlSessionFactoryBuilder().build(in);
     }

    @After//用于在测试方法执行之后执行
     public void destroy()throws Exception{
          in.close();
     }

    /**
     * 测试一级缓存
      */
```

```
    @Test
    public void testFirstLevelCache(){
        SqlSession sqlSession1 = factory.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 sqlSession2 = factory.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。

③ 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。

```
public class User implements Serializable {

 private Integer id;
  private String username;
  private Date birthday;
  private String sex;
  private String address;
}
```