Mybatis 任务二：配置文件深入

课程任务主要内容：

```
* Mybatis 高级查询
* 映射配置文件深入
* 核心配置文件深入
* Mybatis 多表查询
* Mybatis 嵌套查询
```

一 Mybatis 高级查询

1.1 ResutlMap 属性

建立对象关系映射

```
* resultType
        如果实体的属性名与表中字段名一致，将查询结果自动封装到实体类中

* ResutlMap
        如果实体的属性名与表中字段名不一致，可以使用 ResutlMap 实现手动封装到实体类中
```

1) 编写 UserMapper 接口

```
public interface UserMapper {
    public List<User> findAllResultMap();
}
```

2) 编写 UserMapper.xml

```
<!--
        实现手动映射封装
            resultMap
                id="userResultMap" 此标签唯一标识
                type="user" 封装后的实体类型
            <id column="uid" property="id"></id>   表中主键字段封装
                column="uid"   表中的字段名
                property="id" user 实体的属性名
            <result column="NAME" property="username"></result>   表中普通字段封装
                column="NAME"   表中的字段名
                property="username" user 实体的属性名

                补充：如果有查询结果有 字段与属性是对应的，可以省略手动封装 【了解】
    -->
    <resultMap id="userResultMap" type="user">
        <id column="uid" property="id"></id>
        <result column="NAME" property="username"></result>
```

```
        <result column="PASSWORD" property="username"></result>
    </resultMap>

    <select id="findAllResultMap" resultMap="userResultMap">
       SELECT id AS uid,username AS NAME,password AS PASSWORD FROM USER
    </select>
```

3）代码测试

```
@Test
public void testFindAllResultMap() throws Exception {
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    List<User> list = userMapper.findAllResultMap();
    for (User user : list) {
            System.out.println(user); }
}
```

1.2 多条件查询（三种）

需求

根据 id 和 username 查询 user 表

1）方式一

使用 `#{arg0}-#{argn}` 或者 `#{param1}-#{paramn}` 获取参数

UserMapper 接口

```
public interface UserMapper {

    public List<User> findByIdAndUsername1(Integer id, String username);

}
```

UserMapper.xml

```
<mapper namespace="com.lagou.mapper.UserMapper">

    <select id="findByIdAndUsername1" resultType="user">
       <!--  select * from user where id = #{arg0} and username = #{arg1} -->
        select * from user where id = #{param1} and username = #{param2}
    </select>

</mapper>
```

测试

```
@Test
public void testFindByIdAndUsername() throws Exception {

    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    List<User> list = userMapper.findByIdAndUsername1(41, "老王");
    System.out.println(list);
}
```

2）方式二

使用注解，引入 `@Param()` 注解获取参数

UserMapper 接口

```
public interface UserMapper {

    public List<User> findByIdAndUsername2(@Param("id") Integer
id,@Param("username") String username);

}
```

UserMapper.xml

```
<mapper namespace="com.lagou.mapper.UserMapper">

    <select id="findByIdAndUsername2" resultType="user">
        select * from user where id = #{id} and username = #{username}
    </select>

</mapper>
```

测试

```
@Test
public void testFindByIdAndUsername() throws Exception {

    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    List<User> list = userMapper.findByIdAndUsername2(41, "老王");
    System.out.println(list);
}
```

3）方式三（推荐）

使用 pojo 对象传递参数

UserMapper 接口

```
public interface UserMapper {

    public List<User> findByIdAndUsername3(User user);

}
```

UserMapper.xml

```
<mapper namespace="com.lagou.mapper.UserMapper">

    <select id="findByIdAndUsername3" parameterType="com.lagou.domain.User"
resultType="com.lagou.domain.User">
        select * from user where id = #{id} and username = #{username}
    </select>

</mapper>
```

测试

```
@Test
public void testFindByIdAndUsername() throws Exception {

    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    User param = new User();
    param.setId(41);
    param.setUsername("老王");
    List<User> list = userMapper.findByIdAndUsername3(41, "老王");
    System.out.println(list);

}
```

1.3 模糊查询

需求

根据 username 模糊查询 user 表

1）方式一

UserMapper 接口

```
public interface UserMapper {

    public List<User> findByUsername1(String username);
}
```

UserMapper.xml

```xml
<mapper namespace="com.lagou.mapper.UserMapper">

    <select id="findByUsername1" parameterType="string" resultType="user">
        select * from user where username like #{username}
    </select>

</mapper>
```

测试

```java
@Test
public void testFindByUsername() throws Exception {

    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    List<User> list = userMapper.findByUsername1("%王%");
    for (User user : list) {
        System.out.println(user);
    }
}
```

2）方式二

UserMapper 接口

```java
public interface UserMapper {

    public List<User> findByUsername2(String username);
}
```

UserMapper.xml

```xml
<mapper namespace="com.lagou.mapper.UserMapper">

    <!-- 不推荐使用，因为Oracle数据库 除了设置别名其余位置不能使用 双引号 -->
    <select id="findByUsername2" parameterType="string" resultType="user">
        select * from user where username like "%"#{username}"%"
    </select>

</mapper>
```

测试

```
@Test
public void testFindByUsername() throws Exception {

    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    List<User> list = userMapper.findByUsername2("王");
    for (User user : list) {
        System.out.println(user);
    }
}
```

3）方式三

UserMapper 接口

```
public interface UserMapper {

    public List<User> findByUsername3(String username);
}
```

UserMapper.xml

```
<mapper namespace="com.lagou.mapper.UserMapper">

    <!--不推荐使用，因为会出现 sql 注入问题-->
    <select id="findByUsername3" parameterType="string" resultType="user">
        select * from user where username like '%${value}%'
    </select>

</mapper>
```

测试

```
@Test
public void testFindByUsername() throws Exception {

    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    List<User> list = userMapper.findByUsername3("王");
    for (User user : list) {
        System.out.println(user);
    }
}
```

4）方式四（推荐）

UserMapper 接口

```
public interface UserMapper {

    public List<User> findByUsername4(String username);
}
```

UserMapper.xml

```
<mapper namespace="com.lagou.mapper.UserMapper">

     <!--
            推荐使用，concat() 字符串拼接函数
            注意：在 Oracle 中，concat() 函数只能传递二次参数，我们解决方案是嵌套拼接
     -->
    <select id="findByUsername4" parameterType="string" resultType="user">
        select * from user where username like concat(concat('%',#
{username}),'%');
    </select>

</mapper>
```

测试

```
@Test
public void testFindByUsername() throws Exception {

    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    List<User> list = userMapper.findByUsername4("王");
    for (User user : list) {
        System.out.println(user);
    }
}
```

5) ${} 与 #{} 区别【面试题】

#{}:表示一个占位符号

- 通过#{}可以实现 preparedStatement 向占位符中设置值，自动进行 java 类型和 jdbc 类型转换，#{}可以有效防止 sql 注入。
- #{}可以接收简单类型值或 pojo 属性值。
- 如果 parameterType 传输单个简单类型值，#{}括号中可以是 value 或其它名称。

${}:表示拼接 sql 串

- 通过${}可以将 parameterType 传入的内容拼接在 sql 中且不进行 jdbc 类型转换，会出现 sql 注入问题。
- ${}可以接收简单类型值或 pojo 属性值。
- 如果 parameterType 传输单个简单类型值，${}括号中只能是 value。

- 补充：TextSqlNode.java 源码可以证明

二 Mybatis 映射文件深入

2.1 返回主键

应用场景

我们很多时候有这种需求，向数据库插入一条记录后，希望能立即拿到这条记录在数据库中的主键值。

2.1.1 useGeneratedKeys

```
public interface UserMapper {

    // 返回主键
    public void save(User user);

}
```

```
<!--
        useGeneratedKeys="true" 声明返回主键
        keyProperty="id" 把返回主键的值，封装到实体的 id 属性中

        注意：只适用于主键自增的数据库，mysql 和 sqlserver 支持，oracle 不支持
 -->
<insert id="save" parameterType="user" useGeneratedKeys="true" keyProperty="id">
    INSERT INTO `user`(username,birthday,sex,address)
        values(#{username},#{birthday},#{sex},#{address})
</insert>
```

注意：只适用于主键自增的数据库，mysql 和 sqlserver 支持，oracle 不行。

2.1.2 selectKey

```
public interface UserMapper {

    // 返回主键
    public void save(User user);

}
```

```
<!--
        selectKey 适用范围广，支持所有类型数据库
            keyColumn="id"   指定主键列名
            keyProperty="id" 指定主键封装到实体的 id 属性中
            resultType="int" 指定主键类型
```

```
            order="AFTER"    设置在 sql 语句执行前（后），执行此语句
-->
<insert id="save" parameterType="user">
    <selectKey keyColumn="id" keyProperty="id" resultType="int" order="AFTER">
        SELECT LAST_INSERT_ID();
    </selectKey>
    INSERT INTO `user`(username,birthday,sex,address)
        values(#{username},#{birthday},#{sex},#{address})
</insert>
```

11.3 测试代码

```
@Test
public void testSave() throws Exception {
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    User user = new User();
    user.setUsername("子慕");
    user.setAddress("北京");
    user.setBirthday(new Date());
    user.setSex("男");

    userMapper.save(user);
    System.out.println("返回主键:" + user.getId());
}
```

2.2 动态 SQL

应用场景

当我们要根据不同的条件，来执行不同的 sql 语句的时候，需要用到动态 sql。

2.2.1 动态 SQL 之<if>

需求

根据 id 和 username 查询，但是不确定两个都有值。

a) UserMapper 接口

```
public List<User> findByIdAndUsernameIf(User user);
```

b) UserMapper.xml 映射

```
<!--
     where 标签相当于 where 1=1，但是如果没有条件，就不会拼接 where 关键字
-->
<select id="findByIdAndUsernameIf" parameterType="user" resultType="user">
    SELECT * FROM `user`
```

```
    <where>
        <if test="id != null">
            AND id = #{id}
        </if>
        <if test="username != null">
            AND username = #{username}
        </if>
    </where>
</select>
```

c）测试代码

```
// if 标签 where 标签
@Test
public void testFindByIdAndUsernameIf() throws Exception {

    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    User param = new User();
    // param.setId(42);
    // param.setUsername("王小二");

    List<User> list = userMapper.findByIdAndUsernameIf(param);
    System.out.println(list);
}
```

2.2.2 动态 SQL 之<choose>

需求

如果有 id 只使用 id 做查询，没有 id 的话看是否有 username，有 username 就根据 username 做查询，如果都没有，就不带条件。

a）UserMapper 接口

```
public List<User> findByIdAndUsernameChoose(User user);
```

b）UserMapper.xml 映射

```
<!--
        choose 标签相当于 swtich 语句
            when 标签相当于 case 语句
            otherwise 标签相当于 default 语句
-->
<select id="findByIdAndUsernameChoose" parameterType="user" resultType="user">
    SELECT * FROM `user`
    <where>
        <choose>
            <when test="id != null">
```

```
                AND id = #{id}
            </when>
            <when test="username != null">
                AND username = #{username}
            </when>
            <otherwise>
                AND 1=1
            </otherwise>
        </choose>
    </where>
</select>
```

c) 测试代码

```
// choose 标签
@Test
public void testFindByIdAndUsernameChoose() throws Exception {

    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    User param = new User();
    // param.setId(42);
    // param.setUsername("王小二");

    List<User> list = userMapper.findByIdAndUsernameChoose(param);
    System.out.println(list);
}
```

2.2.3 动态 SQL 之<set>

需求

动态更新 user 表数据，如果该属性有值就更新，没有值不做处理。

a) UserMapper 接口

```
public void updateIf(User user);
```

b) UserMapper.xml 映射

```
<!--
        set 标签在更新的时候，自动加上 set 关键字，然后去掉最后一个条件的逗号
-->
<update id="updateIf" parameterType="user">
    UPDATE `user`
    <set>
        <if test="username != null">
            username = #{username},
        </if>
        <if test="birthday != null">
            birthday = #{birthday},
```

```
        </if>
        <if test="sex !=null">
            sex = #{sex},
        </if>
        <if test="address !=null">
            address = #{address},
        </if>
    </set>
    WHERE  id = #{id}
</update>
```

c）测试代码

```
// set 标签
@Test
public void testUpdateIf()throws Exception{
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
    User user = new User();
    user.setId(51);
    user.setUsername("小二王");
    user.setSex("女");

    userMapper.updateIf(user);

}
```

2.2.5 动态 SQL 之<foreach>

foreach 主要是用来做数据的循环遍历

例如：`select * from user where id in (1,2,3)` 在这样的语句中，传入的参数部分必须依靠 foreach 遍历才能实现。

```
* <foreach>标签用于遍历集合，它的属性：

    • collection：代表要遍历的集合元素

    • 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
```

a）集合

UserMapper 接口

```
 public List<User> findByList(List<Integer> ids);
```

UserMaper.xml 映射

```
<!--
        如果查询条件为普通类型 List 集合，collection 属性值为：collection 或者 list
-->
<select id="findByList" parameterType="list" resultType="user" >
   SELECT * FROM `user`
    <where>
        <foreach collection="collection" open="id in(" close=")" item="id"
separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

测试代码

```
// foreach 标签 list
@Test
public void testFindByList() throws Exception {
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    List<Integer> ids = new ArrayList<>();
    ids.add(46);
    ids.add(48);
    ids.add(51);

    List<User> list = userMapper.findByList(ids);
    System.out.println(list);
}
```

b) 数组

UserMapper 接口

```
public List<User> findByArray(Integer[] ids);
```

UserMaper.xml 映射

```
<!--
        如果查询条件为普通类型 Array 数组，collection 属性值为：array
-->
<select id="findByArray" parameterType="int" resultType="user">
    SELECT * FROM `user`
    <where>
        <foreach collection="array" open="id in(" close=")" item="id"
separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

测试代码

```
// foreach 标签 array
@Test
public void testFindByArray() throws Exception {
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    Integer[] ids = {46, 48, 51};
    List<User> list = userMapper.findByArray(ids);

    System.out.println(list);
}
```

c) pojo

QueryVo

```
public class QueryVo {

    private List<Integer> ids;

}
```

核心配置文件

```
<!--设置实体别名-->
<typeAliases>
    <typeAlias type="com.lagou.domain.User" alias="user"></typeAlias>
    <typeAlias type="com.lagou.domain.QueryVo" alias="queryVo"></typeAlias>
</typeAliases>
```

UserMapper 接口

```
public List<User> findByPojo(QueryVo queryVo);
```

UserMaper.xml 映射

```
<!--
        如果查询条件为复杂类型 pojo 对象，collection 属性值为：集合或数组的属性名
-->
<select id="findByPojo" parameterType="queryVo" resultType="user">
    SELECT * FROM `user`
    <where>
        <foreach collection="ids" open="id in(" close=")" item="id"
separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

测试代码

```
// foreach 标签 pojo
@Test
public void testFindByPojo() throws Exception {
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    List<Integer> ids = new ArrayList<>();
    ids.add(46);
    ids.add(48);
    ids.add(51);
    QueryVo queryVo = new QueryVo();
    queryVo.setIds(ids);

    List<User> list = userMapper.findByPojo(queryVo);
    System.out.println(list);
}
```

2.3 SQL 片段

应用场景

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

```
<!--抽取的 sql 片段-->
<sql id="selectUser">
    SELECT * FROM `user`
</sql>


<select id="findByList" parameterType="list" resultType="user" >
    <!--引入 sql 片段-->
    <include refid="selectUser"></include>
    <where>
        <foreach collection="collection" open="id in(" close=")" item="id"
separator=",">
```

```
            #{id}
        </foreach>
    </where>
</select>

<select id="findByArray" parameterType="integer[]" resultType="user">
    <!--引入 sql 片段-->
    <include refid="selectUser"></include>
    <where>
        <foreach collection="array" open="id in(" close=")" item="id"
separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

2.4 知识小结

MyBatis 映射文件配置

```
<select>：查询

<insert>：插入

<update>：修改

<delete>：删除

<selectKey>：返回主键

<where>：where 条件

<if>：if 判断

<foreach>：for 循环

<set>：set 设置

<sql>：sql 片段抽取
```

三 Mybatis 核心配置文件深入

3.1 plugins 标签

MyBatis 可以使用第三方的插件来对功能进行扩展，分页助手 PageHelper 是将分页的复杂操作进行封装，使用简单的方式即可获得分页的相关数据

开发步骤:

①导入通用 PageHelper 的坐标

②在 mybatis 核心配置文件中配置 PageHelper 插件

③测试分页数据获取

①导入通用 PageHelper 坐标

```
<!-- 分页助手 -->
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>3.7.5</version>
</dependency>
<dependency>
    <groupId>com.github.jsqlparser</groupId>
    <artifactId>jsqlparser</artifactId>
    <version>0.9.1</version>
</dependency>
```

②在 mybatis 核心配置文件中配置 PageHelper 插件

```
<!-- 注意：分页助手的插件  配置在通用馆 mapper 之前 -->
<plugin interceptor="com.github.pagehelper.PageHelper">
    <!-- 指定方言 -->
    <property name="dialect" value="mysql"/>
</plugin>
```

③测试分页代码实现

```
@Test
public void testPageHelper(){
    //设置分页参数
    PageHelper.startPage(1,2);

    List<User> select = userMapper2.select(null);
    for(User user : select){
        System.out.println(user);
    }
}
```

获得分页相关的其他参数

```
//其他分页的数据
PageInfo<User> pageInfo = new PageInfo<User>(select);
System.out.println("总条数："+pageInfo.getTotal());
System.out.println("总页数："+pageInfo.getPages());
System.out.println("当前页："+pageInfo.getPageNum());
System.out.println("每页显示长度："+pageInfo.getPageSize());
System.out.println("是否第一页："+pageInfo.isIsFirstPage());
System.out.println("是否最后一页："+pageInfo.isIsLastPage());
```

3.2 知识小结

MyBatis 核心配置文件常用标签:

1、properties 标签：该标签可以加载外部的 properties 文件

2、typeAliases 标签：设置类型别名

3、environments 标签：数据源环境配置标签

4、plugins 标签：配置 MyBatis 的插件

四 Mybatis 多表查询

4.1 数据库表关系介绍

关系型数据库表关系分为

```
* 一对一

* 一对多

* 多对多
```

举例

```
* 人和身份证号就是一对一
    一个人只能有一个身份证号
    一个身份证号只能属于一个人

* 用户和订单就是一对多，订单和用户就是多对一
    一个用户可以下多个订单
    多个订单属于同一个用户

* 学生和课程就是多对多
    一个学生可以选修多门课程
    一个课程可以被多个学生选修

* 特例
    一个订单只从属于一个用户，所以 mybatis 将多对一看成了一对一
```

```
DROP TABLE IF EXISTS `orders`;
CREATE TABLE `orders` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `ordertime` varchar(255) DEFAULT NULL,
  `total` DOUBLE DEFAULT NULL,
  `uid` INT(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `uid` (`uid`),
  CONSTRAINT `orders_ibfk_1` FOREIGN KEY (`uid`) REFERENCES `user` (`id`)
) ENGINE=INNODB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of orders
-- ----------------------------
INSERT INTO `orders` VALUES ('1', '2020-12-12', '3000', '1');
INSERT INTO `orders` VALUES ('2', '2020-12-12', '4000', '1');
INSERT INTO `orders` VALUES ('3', '2020-12-12', '5000', '2');

-- ----------------------------
-- Table structure for sys_role
-- ----------------------------
```

```
DROP TABLE IF EXISTS `sys_role`;
CREATE TABLE `sys_role` (
  `id` INT(11) NOT NULL AUTO_INCREMENT,
  `rolename` VARCHAR(255) DEFAULT NULL,
  `roleDesc` VARCHAR(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of sys_role
-- ----------------------------
INSERT INTO `sys_role` VALUES ('1', 'CTO', 'CTO');
INSERT INTO `sys_role` VALUES ('2', 'CEO', 'CEO');

-- ----------------------------
-- Table structure for sys_user_role
-- ----------------------------
DROP TABLE IF EXISTS `sys_user_role`;
CREATE TABLE `sys_user_role` (
  `userid` INT(11) NOT NULL,
  `roleid` INT(11) NOT NULL,
  PRIMARY KEY (`userid`,`roleid`),
  KEY `roleid` (`roleid`),
  CONSTRAINT `sys_user_role_ibfk_1` FOREIGN KEY (`userid`) REFERENCES `user` (`id`),
  CONSTRAINT `sys_user_role_ibfk_2` FOREIGN KEY (`roleid`) REFERENCES `sys_role` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

-- ----------------------------
-- Records of sys_user_role
-- ----------------------------
INSERT INTO `sys_user_role` VALUES ('1', '1');
INSERT INTO `sys_user_role` VALUES ('2', '1');
INSERT INTO `sys_user_role` VALUES ('1', '2');
INSERT INTO `sys_user_role` VALUES ('2', '2');
```

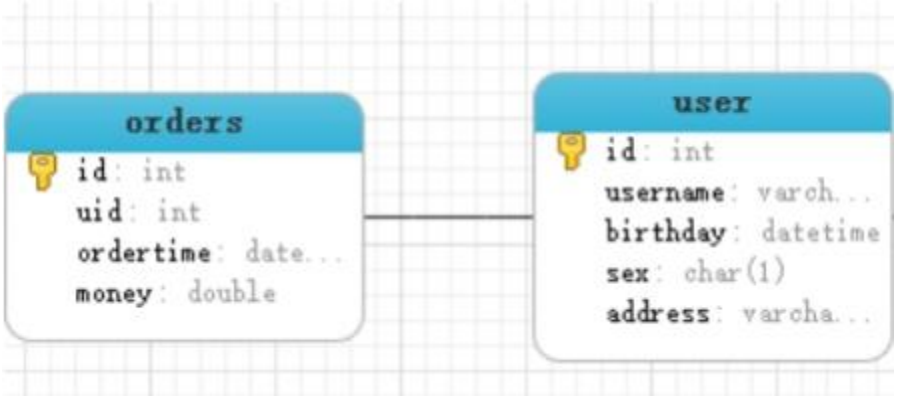
4.2 一对一（多对一）multitable

4.2.1 介绍

一对一查询模型

用户表和订单表的关系为，一个用户有多个订单，一个订单只从属于一个用户

一对一查询的需求：查询一个订单，与此同时查询出该订单所属的用户



一对一查询语句

```
SELECT * FROM orders o LEFT JOIN USER u ON o.`uid`=u.`id`;
```

4.2.2 代码实现

1）Order 实体

```
public class Order {

    private Integer id;
    private Date ordertime;
    private double money;

    // 表示当前订单属于哪个用户
    private User user;
}
```

2）OrderMapper 接口

```
public interface OrderMapper {

    public List<Order> findAllWithUser();
}
```

3）OrderMapper.xml 映射

```
<resultMap id="orderMap" type="com.lagou.domain.Order">
    <id column="id" property="id"></id>
    <result column="ordertime" property="ordertime"></result>
    <result column="money" property="money"></result>
    <result column="uid" property="user.id"></result>
    <result column="username" property="user.username"></result>
    <result column="birthday" property="user.birthday"></result>
    <result column="sex" property="user.sex"></result>
    <result column="address" property="user.address"></result>
</resultMap>
<select id="findAllWithUser" resultMap="orderMap">
    SELECT * FROM orders o LEFT JOIN USER u ON o.`uid`=u.`id`;
</select>
```

```
<resultMap id="orderMap" type="com.lagou.domain.Order">
    <id column="id" property="id"></id>
    <result column="ordertime" property="ordertime"></result>
    <result column="money" property="money"></result>

        <!--
            一对一（多对一）使用 association 标签关联
                property="user" 封装实体的属性名
                javaType="user" 封装实体的属性类型
        -->
    <association property="user" javaType="com.lagou.domain.User">
        <id column="uid" property="id"></id>
        <result column="username" property="username"></result>
        <result column="birthday" property="birthday"></result>
        <result column="sex" property="sex"></result>
        <result column="address" property="address"></result>
    </association>
</resultMap>
```

4）测试代码

```
@Test
public void testOrderWithUser() throws Exception {
    OrderMapper orderMapper = sqlSession.getMapper(OrderMapper.class);

    List<Order> list = orderMapper.findAllWithUser();

    for (Order order : list) {
        System.out.println(order);
    }
}
```

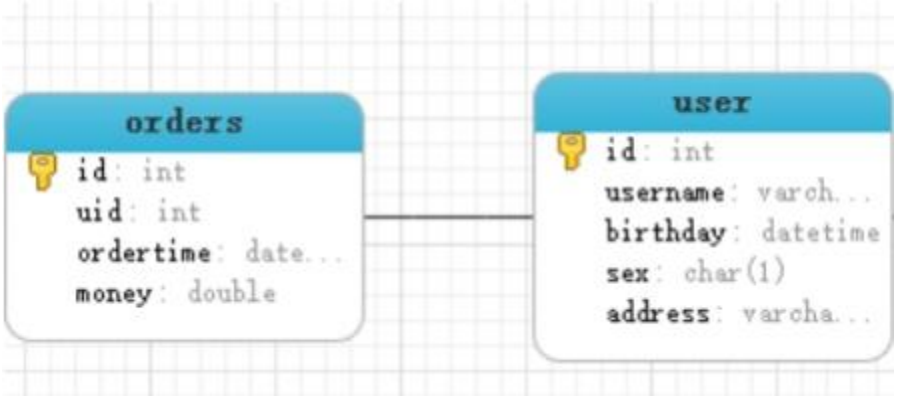
4.3 一对多

4.3.1 介绍

一对多查询模型

用户表和订单表的关系为，一个用户有多个订单，一个订单只从属于一个用户

一对多查询的需求：查询一个用户，与此同时查询出该用户具有的订单



一对多查询语句

```
SELECT *,o.id oid FROM USER u LEFT JOIN orders o ON u.`id` = o.`uid`;
```

4.3.2 代码实现

1) User 实体

```
public class User {
    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;

    // 代表当前用户具备的订单列表
    private List<Order> orderList;
}
```

2) UserMapper 接口

```
public interface UserMapper {

    public List<User> findAllWithOrder();
}
```

3) UserMapper.xml 映射

```
<resultMap id="userMap" type="com.lagou.domain.User">
    <id column="id" property="id"></id>
    <result column="username" property="username"></result>
    <result column="birthday" property="birthday"></result>
    <result column="sex" property="sex"></result>
    <result column="address" property="address"></result>
        <!--
            一对多使用 collection 标签关联
                property="orderList"    封装到集合的属性名
                ofType="order"          封装集合的泛型类型
        -->
    <collection property="orderList" ofType="com.lagou.domain.Order">
        <id column="oid" property="id"></id>
        <result column="ordertime" property="ordertime"></result>
        <result column="money" property="money"></result>
    </collection>
</resultMap>

<select id="findAllWithOrder" resultMap="userMap">
    SELECT *,o.id oid FROM USER u LEFT JOIN orders o ON u.`id`=o.`uid`;
</select>
```

4）测试代码

```
@Test
public void testUserWithOrder() throws Exception {
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    List<User> list = userMapper.findAllWithOrder();

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

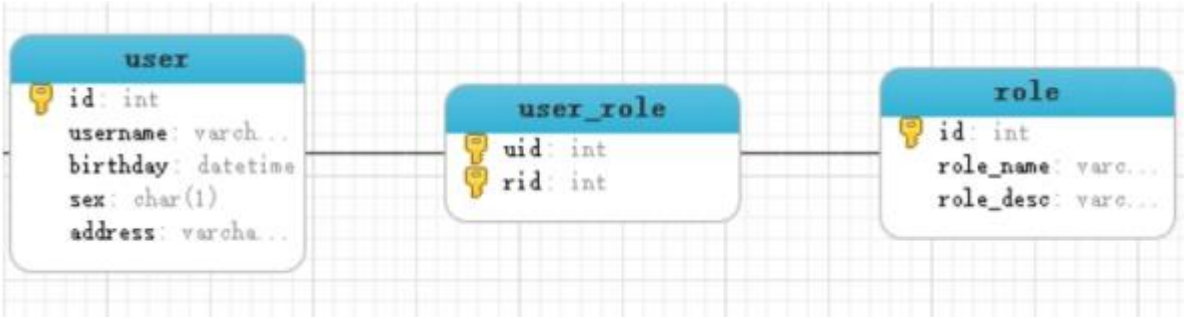
4.4 多对多

4.4.1 介绍

多对多查询的模型

用户表和角色表的关系为，一个用户有多个角色，一个角色被多个用户使用

多对多查询的需求：查询用户同时查询出该用户的所有角色



多对多查询语句

```
SELECT
  *
FROM
  USER u -- 用户表
  LEFT JOIN user_role ur  -- 左外连接中间表
    ON u.`id` = ur.`uid`
  LEFT JOIN role r -- 左外连接中间表
    ON ur.`rid` = r.`id` ;
```

4.4.2 代码实现

1）User 和 Role 实体

```
public class User {
```

```
    private Integer id;
    private String username;
    private Date birthday;
    private String sex;
    private String address;
    // 代表当前用户关联的角色列表
    private List<Role> roleList;
}

public class Role {

    private Integer id;
    private String roleName;
    private String roleDesc;
}
```

2）UserMapper 接口

```
public interface UserMapper {

    public List<User> findAllWithRole();
}
```

3）UserMapper.xml 映射

```
<resultMap id="userAndRoleMap" type="com.lagou.domain.User">
    <id column="id" property="id"></id>
    <result column="username" property="username"></result>
    <result column="birthday" property="birthday"></result>
    <result column="sex" property="sex"></result>
    <result column="address" property="address"></result>
    <collection property="orderList" ofType="com.lagou.domain.Role">
        <id column="rid" property="id"></id>
        <result column="role_name" property="roleName"></result>
        <result column="role_desc" property="roleDesc"></result>
    </collection>
</resultMap>

<select id="findAllWithRole" resultMap="userAndRoleMap">
    SELECT * FROM USER u LEFT JOIN user_role ur ON u.`id`=ur.`uid` INNER JOIN
role r ON ur.`rid` = r.`id`;
</select>
```

4）测试代码

```
@Test
public void testUserWithRole() throws Exception {
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    List<User> list = userMapper.findAllWithRole();

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

4.5 小结

MyBatis 多表配置方式

```
* 一对一配置：使用<resultMap>+<association>做配置

* 一对多配置：使用<resultMap>+<collection>做配置

* 多对多配置：使用<resultMap>+<collection>做配置

* 多对多的配置跟一对多很相似，难度在于 SQL 语句的编写。
```

五 MyBatis 嵌套查询

5.1 什么是嵌套查询

嵌套查询就是将原来多表查询中的联合查询语句拆成单个表的查询，再使用 mybatis 的语法嵌套在一起。

举个栗子

```
* 需求：查询一个订单，与此同时查询出该订单所属的用户

1. 联合查询
        SELECT * FROM orders o LEFT JOIN USER u ON o.`uid`=u.`id`;
2. 嵌套查询
    2.1 先查询订单
        SELECT * FROM orders
    2.2 再根据订单 uid 外键，查询用户
        SELECT * FROM `user` WHERE id = #{根据订单查询的 uid}
    2.3 最后使用 mybatis，将以上二步嵌套起来
            ...
```

5.2 一对一嵌套查询

5.2.1 介绍

需求：查询一个订单，与此同时查询出该订单所属的用户

一对一查询语句

```sql
-- 先查询订单
SELECT * FROM orders;

-- 再根据订单 uid 外键，查询用户
SELECT * FROM `user` WHERE id = #{订单的uid};
```

5.2.2 代码实现

1) OrderMapper 接口

```java
public interface OrderMapper {

    public List<Order> findAllWithUser();
}
```

2) OrderMapper.xml 映射

```xml
<!--一对一嵌套查询-->
<resultMap id="orderMap" type="order">
    <id column="id" property="id"></id>
    <result column="ordertime" property="ordertime"></result>
    <result column="money" property="money"></result>
    <!--根据订单中 uid 外键，查询用户表-->
    <association property="user" javaType="user" column="uid"
select="com.lagou.mapper.UserMapper.findById"></association>
</resultMap>

<select id="findAllWithUser" resultMap="orderMap" >
    SELECT * FROM orders
</select>
```

3) UserMapper 接口

```java
public interface UserMapper {

    public User findById(Integer id);
}
```

4) UserMapper.xml 映射

```xml
<select id="findById" parameterType="int" resultType="user">
    SELECT * FROM `user` where id = #{uid}
</select>
```

5) 测试代码

```java
@Test
public void testOrderWithUser() throws Exception {
    OrderMapper orderMapper = sqlSession.getMapper(OrderMapper.class);

    List<Order> list = orderMapper.findAllWithUser();

    for (Order order : list) {
        System.out.println(order);
    }
}
```

5.3 一对多嵌套查询

5.3.1 介绍

需求：查询一个用户，与此同时查询出该用户具有的订单

一对多查询语句

```sql
-- 先查询用户
SELECT * FROM `user`;

-- 再根据用户 id 主键，查询订单列表
SELECT * FROM orders where uid = #{用户 id};
```

5.3.2 代码实现

a) UserMapper **接口**

```java
public interface UserMapper {

    public List<User> findAllWithOrder();
}
```

b) UserMapper.xml **映射**

```xml
<!--一对多嵌套查询-->
<resultMap id="userMap" type="user">
    <id column="id" property="id"></id>
    <result column="username" property="username"></result>
    <result column="birthday" property="birthday"></result>
    <result column="sex" property="sex"></result>
    <result column="address" property="address"></result>
    <!--根据用户 id，查询订单表-->
    <collection property="orderList" column="id" ofType="order"
select="com.lagou.mapper.OrderMapper.findByUid"></collection>
</resultMap>

<select id="findAllWithOrder" resultMap="userMap">
    SELECT * FROM `user`
```

```
</select>
```

c) OrderMapper 接口

```
public interface OrderMapper {

    public List<Order> findByUid(Integer uid);
}
```

d) OrderMapper.xml 映射

```
<select id="findByUid" parameterType="int" resultType="order">
    SELECT * FROM orders where uid = #{uid}
</select>
```

e) 测试代码

```
@Test
public void testUserWithOrder() throws Exception {
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    List<User> list = userMapper.findAllWithOrder();

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

5.4 多对多嵌套查询

5.4.1 介绍

需求：查询用户同时查询出该用户的所有角色

多对多查询语句

```
-- 先查询用户
SELECT * FROM `user`;

-- 再根据用户 id 主键，查询角色列表
SELECT * FROM role r INNER JOIN user_role ur ON r.`id` = ur.`rid`
    WHERE ur.`uid` = #{用户 id};
```

5.4.2 代码实现

a) UserMapper 接口

```
public interface UserMapper {

    public List<User> findAllWithRole();
}
```

b) UserMapper.xml 映射

```
<!--多对多嵌套查询-->
<resultMap id="userAndRoleMap" type="user">
    <id column="id" property="id"></id>
    <result column="username" property="username"></result>
    <result column="birthday" property="birthday"></result>
    <result column="sex" property="sex"></result>
    <result column="adress" property="address"></result>
    <!--根据用户 id，查询角色列表-->
    <collection property="roleList" column="id" ofType="role"
select="com.lagou.mapper.RoleMapper.findByUid"></collection>
</resultMap>

<select id="findAllWithRole" resultMap="userAndRoleMap">
    SELECT * FROM `user`
</select>
```

c) RoleMapper 接口

```
public interface RoleMapper {

    public List<Role> findByUid(Integer uid);
}
```

d) RoleMapper.xml 映射

```
<select id="findByUid" parameterType="int" resultType="role">
    SELECT r.id,r.`role_name` roleName,r.`role_desc` roleDesc FROM role r
        INNER JOIN user_role ur ON r.`id` = ur.`rid` WHERE ur.`uid` = #{uid}
</select>
```

e) 测试代码

```java
@Test
public void testUserWithRole() throws Exception {
    UserMapper userMapper = sqlSession.getMapper(UserMapper.class);

    List<User> list = userMapper.findAllWithRole();

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

5.5 小结

一对一配置：使用<resultMap>+<association>做配置，通过 column 条件，执行 select 查询

一对多配置：使用<resultMap>+<collection>做配置，通过 column 条件，执行 select 查询

多对多配置：使用<resultMap>+<collection>做配置，通过 column 条件，执行 select 查询

优点：简化多表查询操作

缺点：执行多次 sql 语句，浪费数据库性能