

Mybatis 任务二：复杂映射&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 ResultMap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
- * `ResultMap`
如果实体的属性名与表中字段名不一致，可以使用`ResultMap`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="password"></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(1, "子慕");
    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(1, "子慕");
    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(1);  
    param.setUsername("子慕");  
    List<User> list = userMapper.findByIdAndUsername3(param);  
    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

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

    <select id="findByUsername1" 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.findByUsername1("%王%");
    for (User user : list) {
        System.out.println(user);
    }
}
```

2) 方式二

UserMapper接口

```
public interface UserMapper {

    public List<User> findByUsername2(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="findByUsername2" 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.findByUsername2("%王%");
    for (User user : list) {
        System.out.println(user);
    }
}
```

3) `${}` 与 `#{}` 区别【笔试题】

`#{}` :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，`#{}` 括号中名称随便写。

`${}` :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<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(1);
    user.setUsername("小二王");
    user.setSex("女");

    userMapper.updateIf(user);
}
```

2.2.3 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);
```

UserMapper.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);

```

UserMapper.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);
}
```

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

```
<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

```
<!-- 分页助手的插件 -->
<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

案例环境准备

```
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 `sys_role` (`id`),
  CONSTRAINT `sys_user_role_ibfk_2` FOREIGN KEY (`roleid`) REFERENCES `user` (`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');
```

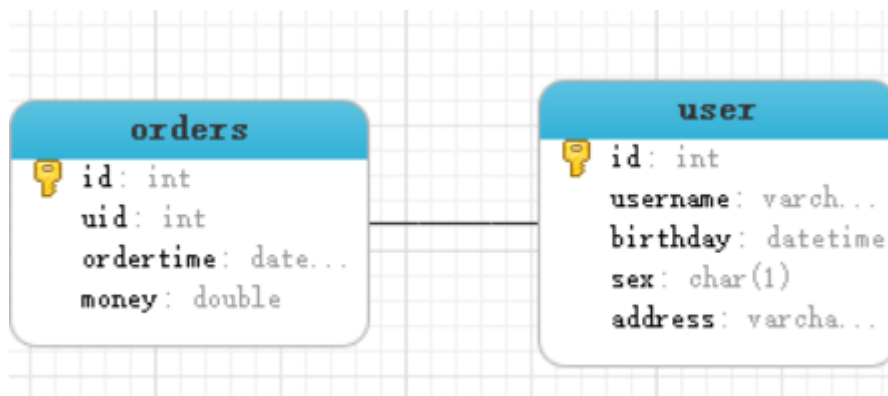
4.2 一对一（多对一）

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：查询所有订单，与此同时查询出每个订单所属的用户



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>

    <!--
        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);
    }
}

```

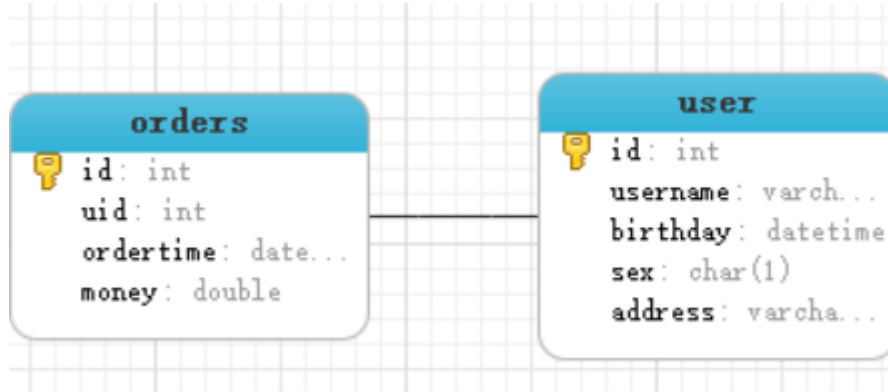
4.3 一对多

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：查询所有用户，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);
    }
}

```

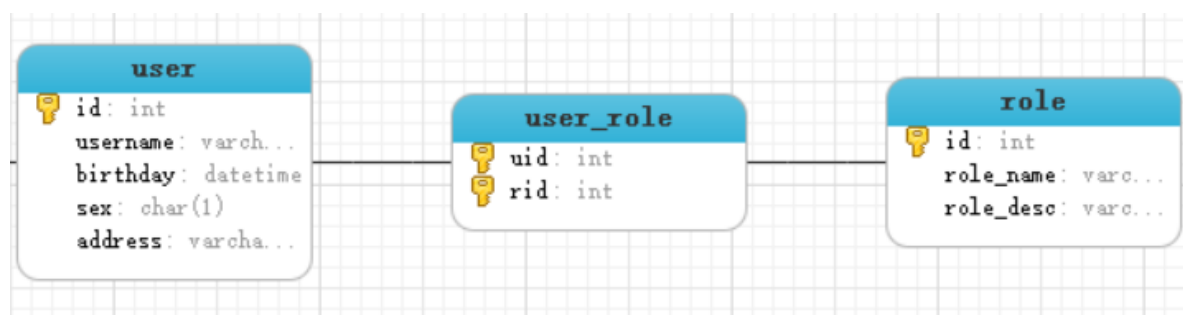
4.4 多对多

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：查询所有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

- * 多对一（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

```
-- 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

```
public interface OrderMapper {

    public List<Order> findAllWithUser();

}
```

2) OrderMapper.xml映射

```
<!--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

```
public interface UserMapper {  
  
    public User findById(Integer id);  
}
```

4) UserMapper.xml映射

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

5) 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);  
    }  
}
```

5.3 一对多嵌套查询

5.3.1 介绍

需求：查询所有用户，与此同时查询出该用户具有的订单

一对多查询语句

```
-- 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

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

b) UserMapper.xml映射

```
<!--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.findById"></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

```
@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