# Mybatis

## 学习目标

```
1. 能够使用动态sql完成sql拼接
2. 能够使用resultMap完成多表查询
3. 能够使用一对一嵌套查询
4. 能够使用一对多嵌套查询
5. 能够完成多表连接查询
6. 了解加载策略的作用和配置
7. 了解缓存的作用和配置
```

## 一、动态SQL

我们已经学过了Mybatis的SQL写法，下面看几个需求：

- findbyCondtion(User user):根据传入的user对象进行查询，将不为空的属性作为查询条件
- update(User user):根据传入的user对象进行更新，将不为空的属性更新到数据库
- inerst(User user):根据传入的user对象进行新增，将不为空的属性插入到数据库

像上面的场景，**程序运行时的不同条件产生不同的SQL**，这就用到了动态SQL。

动态SQL是Mybatis的强大特性之一。它的主要作用是通过Mybatis提供的标签实现sql的动态拼装。

Mybatis3之后，需要了解的动态SQL标签仅仅只有下面几个:

- if 用于条件判断
- (where, set) 用于去除分隔符
- foreach 用于循环遍历
- sql片段

### 1.1 条件判断

查询findByUser1(User user) 根据user对象(name email)中不为空的属性进行查询

#### 1.1.1 接口文件

```
//条件判断
List<User> findByUser(User user);
```

#### 1.1.2 映射文件

**方式一**

```
<!--if 单分支条件判断
    where 1=1 只是为了格式正确
-->
<select id="findByUser" resultType="com.itheima.domain.User">
    select * from user where 1=1
    <if test="name != null and name != ''">
        and name = #{name}
    </if>
    <if test="email != null and email != ''">
        and email = #{email}
    </if>
</select>
```

**方式二**

```
<!--
    where关键字作用:
    1> 当where代码块中所有条件都不成立的时候, 整个where代码块不生效
    2> 当where代码块中至少有一个条件成立的时候, 它会
        在where代码块之前添加一个 where关键字
        如果你的where代码块是以and|or开头, 它会帮你删掉第一个and|or
-->
<select id="findByUser" resultType="com.itheima.domain.User">
    select * from user
    <where>
        <if test="name != null and name != ''">
            and name = #{name}
        </if>
        <if test="email != null and email != ''">
            and email = #{email}
        </if>
    </where>
</select>
```

### 1.1.3 测试

```
@Test
public void testFindByUser() {
    UserDao userDao = sqlSession.getMapper(UserDao.class);
    User userParam = new User();
    userParam.setName("传智播客");
    userParam.setEmail("admin@itcast.cn");
    List<User> users = userDao.findByUser(userParam);
    for (User user : users) {
        System.out.println(user);
    }
}
```

测试结果如下:

```
==>  Preparing: select * from user where 1=1 and name = ? and email = ?
[DEBUG] 2019-12-14 19:36:49,151 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debu
==> Parameters: 传智播客(String), admin@itcast.cn(String)
[DEBUG] 2019-12-14 19:36:49,173 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debu
<==      Total: 2
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sat Dec 1
User{uid=5, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sat Dec 1
[DEBUG] 2019-12-14 19:36:49,175 method:org.apache.ibatis.transaction.jdbc.JdbcTransaction
```

```
==>  Preparing: select * from user WHERE name = ? and email = ?
[DEBUG] 2019-12-14 19:37:41,220 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.jav
==> Parameters: 传智播客(String), admin@itcast.cn(String)
[DEBUG] 2019-12-14 19:37:41,240 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.jav
<==      Total: 2
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sat Dec 14 00:00:00 CST 2019}
User{uid=5, name='传智播客', password='admin', email='admin@itcast.cn', birthday=Sat Dec 14 00:00:00 CST 2019}
```

## 1.2 set 用于update语句

根据uid进行更新一个user对象中不为空的属性

### 1.2.1 接口文件

```
//修改
void update(User user);
```

### 1.2.2 映射文件

```
    <!--按照id 修改其他的-->
    <!--
        set作用:
            1 在set代码块之前加一个set关键字
            2 去掉set代码块中的最后一个,
        注意: set代码块中要至少有一个条件是成立的
    -->
    <update id="update" parameterType="com.itheima.domain.User">
        update user
        <set>
            <if test="name != null and name != ''">
                name = #{name},
            </if>
            <if test="password != null and password != ''">
                password = #{password},
            </if>
            <if test="email != null and email != ''">
                email = #{email},
            </if>
            <if test="birthday != null">
                birthday = #{birthday},
            </if>
        </set>
        where uid = #{uid}
    </update>
```

### 1.2.3 测试

```
    //修改
    @Test
    public void testUpdate() {
```

```
        //创建user对象
        User user = new User();
        //user.setName("传智播客8");
        //user.setPassword("admin8");
        user.setEmail("admin@itcast1.cn");
        //user.setBirthday(new Date());
        user.setUid(2);

        //执行操作
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        userDao.update(user);
    }
```

测试结果如下:

```
==>  Preparing: update user SET email = ? where uid = ?
[DEBUG] 2019-12-14 19:39:39,872 method:org.apache.ibatis.logg
==> Parameters: admin@itcast1.cn(String), 2(Integer)
[DEBUG] 2019-12-14 19:39:39,874 method:org.apache.ibatis.logg
<==    Updates: 1
[DEBUG] 2019-12-14 19:39:39,874 method:org.apache.ibatis.tran
```

## 1.3 foreach

foreach主要是用来做遍历。

典型的应用场景是SQL中的in语法中，select * from user where uid 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。我们传入的参数，一般有下面几个形式:

- 集合(List Set)
- 数组
- pojo

```
foreach的选项
    collection：数据源【重点关注这一项，它的值会根据出入的参数类型不同而不同】
    open:开始遍历之前的拼接字符串
    close:结束遍历之后的拼接字符串
    separator:每次遍历之间的分隔符
    item:每次遍历出的数据
    index:遍历的次数，从0开始
```

### 方式一: 使用集合做参数

#### 接口文件

```
List<User> findByUids1(List<Integer> uids);
```

#### 映射文件

```
    <!--传入的参数是一个集合 3  5  7, 想要的结果是(3,5,7)-->
    <select id="findByUids1" resultType="com.itheima.domain.User">
```

```
        select * from user where uid in
        <!--
            collection  数据源，如果传入的参数类型是Collection，那么这里的值是collection
            item  每次循环的临时变量
            separator    每次循环的分隔符
            open  开始符号
            close 结束符号
            index 遍历的索引
        -->
        <foreach collection="collection" item="it" separator="," open="(" close=")">
            #{it}
        </foreach>
    </select>
```

**测试**

```
    @Test
    public void testFindByUids1() {
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<Integer> uids = new ArrayList<>();
        uids.add(3);
        uids.add(5);
        uids.add(7);
        List<User> users = userDao.findByUids1(uids);
        for (User user : users) {
            System.out.println(user);
        }
    }
```

测试结果如下:

```
==>  Preparing: select * from user where uid in ( ? , ? , ? )
[DEBUG] 2019-12-14 19:42:08,165 method:org.apache.ibatis.logging.jdbc.Base
==> Parameters: 3(Integer), 5(Integer), 7(Integer)
[DEBUG] 2019-12-14 19:42:08,186 method:org.apache.ibatis.logging.jdbc.Base
<==      Total: 2
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn', birt
User{uid=5, name='传智播客', password='admin', email='admin@itcast.cn', birt
```

## 方式二: 使用数组做参数

**接口文件**

```
List<User> findByUids2(Integer[] uids);
```

**映射文件**

```
    <select id="findByUids2" resultType="com.itheima.domain.User">
        select * from user where uid in
        <!--collection  数据源，如果传入的参数类型是数组，那么这里的值是array-->
        <foreach collection="array" item="it" separator="," open="(" close=")">
            #{it}
        </foreach>
    </select>
```

**测试**

```
    @Test
    public void testFindByUids2() {
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findByUids2(new Integer[]{3,5,7});
        for (User user : users) {
            System.out.println(user);
        }
    }
```

测试结果如下:

```
==>  Preparing: select * from user where uid in ( ? , ? , ? )
[DEBUG] 2019-12-14 19:42:47,307 method:org.apache.ibatis.logging.jdk
==> Parameters: 3(Integer), 5(Integer), 7(Integer)
[DEBUG] 2019-12-14 19:42:47,327 method:org.apache.ibatis.logging.jdk
<==      Total: 2
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn
User{uid=5, name='传智播客', password='admin', email='admin@itcast.cn
```

## 方式三: 使用实体做参数

**接口文件**

```
List<User> findByUids3(User user);
```

**映射文件**

```
    <select id="findByUids3" resultType="com.itheima.domain.User">
        select * from user where uid in
         <!--collection  数据源，如果传入的参数类型是实体，那么这里的值是实体类中属性名称-
->
        <foreach collection="uids" item="it" separator="," open="(" close=")">
            #{it}
        </foreach>
    </select>
```

**测试**

```
    @Test
    public void testFindByUids3() {
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<Integer> uids = new ArrayList<>();
```

```
        uids.add(3);
        uids.add(5);
        uids.add(7);

        User userParam = new User();
        userParam.setUids(uids);

        List<User> users = userDao.findByUids3(userParam);
        for (User user : users) {
            System.out.println(user);
        }
    }
```

测试结果如下:

```
==>  Preparing: select * from user where uid in ( ? , ? , ? )
[DEBUG] 2019-12-14 19:43:07,714 method:org.apache.ibatis.logging.jdbc.Bas
==> Parameters: 3(Integer), 5(Integer), 7(Integer)
[DEBUG] 2019-12-14 19:43:07,737 method:org.apache.ibatis.logging.jdbc.Bas
<==      Total: 2
User{uid=3, name='传智播客', password='admin', email='admin@itcast.cn', bir
User{uid=5, name='传智播客', password='admin', email='admin@itcast.cn', bir
```

## 1.4 sql片段和include

sql片段的作用是将代码中重复的部分,提取出来达到复用的目的

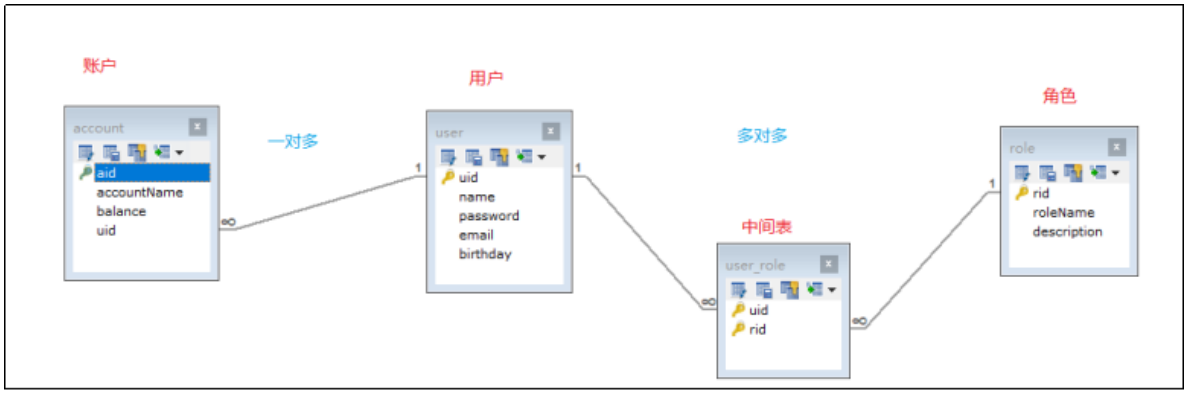
> 定义SQL片段:
>
> 引用SQL片段:

```
<select id="findByUids2" resultType="com.itheima.domain.User">
    <include refid="sql1"></include>
    <foreach collection="array" item="it" separator="," open="(" close=")">
        #{it}
    </foreach>
</select>

<select id="findByUids3" resultType="com.itheima.domain.User">
    <include refid="sql1"></include>
    <foreach collection="uids" item="it" separator="," open="(" close=")">
        #{it}
    </foreach>
</select>


<!--提取了一段公共sql-->
<sql id="sql1">
    select * from user where uid in
</sql>
```

# 二、多表关系

## 2.1 准备多表环境

### 2.1.1 创建数据表




```
DROP TABLE IF EXISTS `user_role`;
DROP TABLE IF EXISTS `role`;
DROP TABLE IF EXISTS `account`;
DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (
  `uid` INT(11) NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(100) NOT NULL,
  `password` VARCHAR(50) NOT NULL,
  `email` VARCHAR(50) DEFAULT NULL,
  `birthday` DATE DEFAULT NULL,
  PRIMARY KEY  (`uid`)
) ENGINE=INNODB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
INSERT  INTO `user`(`uid`,`name`,`password`,`email`,`birthday`) VALUES (1,'传智播客1','admin1','admin1@itcast.cn','2019-05-30'),(2,'传智播客2','admin2','admin2@itcast.cn','2019-06-01');


CREATE TABLE `account` (
  `aid` INT(11) NOT NULL AUTO_INCREMENT,
  `accountName` VARCHAR(100) DEFAULT NULL,
  `balance` FLOAT(10,2) DEFAULT NULL,
  `uid` INT(11) DEFAULT NULL,
  PRIMARY KEY  (`aid`),
  KEY `fk_uid` (`uid`),
  CONSTRAINT `fk_uid` FOREIGN KEY (`uid`) REFERENCES `user` (`uid`)
) ENGINE=INNODB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8;
INSERT  INTO `account`(`aid`,`accountName`,`balance`,`uid`) VALUES (1,'B01',100.00,1),(2,'B02',200.00,1),(3,'B03',300.00,1),(4,'B04',400.00,2);


CREATE TABLE `role` (
  `rid` INT(11) NOT NULL AUTO_INCREMENT,
  `roleName` VARCHAR(100) NOT NULL,
  `description` VARCHAR(300) DEFAULT NULL,
  PRIMARY KEY  (`rid`)
) ENGINE=INNODB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
INSERT  INTO `role`(`rid`,`roleName`,`description`) VALUES (1,'出库管理','只能对商品进行出库操作'),(2,'入库管理','只能对商品进行入库操作');


CREATE TABLE `user_role` (
  `uid` INT(11) NOT NULL,
  `rid` INT(11) NOT NULL,
  PRIMARY KEY  (`uid`,`rid`),
  KEY `rid_fk_1` (`rid`),
```

```
  CONSTRAINT `uid_fk_1` FOREIGN KEY (`uid`) REFERENCES `user` (`uid`),
  CONSTRAINT `rid_fk_1` FOREIGN KEY (`rid`) REFERENCES `role` (`rid`)
) ENGINE=INNODB DEFAULT CHARSET=utf8;
INSERT  INTO `user_role`(`uid`,`rid`) VALUES (1,1),(2,1),(1,2);
```

### 2.1.2 创建新工程,导入昨天的jar包(略)

### 2.1.3 创建实体类

```
public class User {
    private Integer uid;
    private String name;
    private String password;
    private String email;
    private Date birthday;
}

public class Account {
    private Integer aid;
    private String accountName;
    private Float balance;
}

public class Role {
    private Integer rid;
    private String roleName;
    private String description;
}
```

### 2.1.4 创建dao的接口(略)

### 2.1.5 创建dao的映射文件(略)

### 2.1.6 加入mybatis的主配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>

    <!--环境配置-->
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <!--数据库连接四要素-->
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql:///mybatis_118"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
```

```
    </environments>

    <!--引入映射文件(sql语句)-->
    <mappers>
        <mapper resource="mappers/UserMapper.xml"/>
        <mapper resource="mappers/RoleMapper.xml"/>
        <mapper resource="mappers/AccountMapper.xml"/>
    </mappers>
</configuration>
```

### 2.1.7 加入日志配置文件(略)

### 2.1.8 建立类间关系

**建立User和Account之间的关系**

```
User.java
public class User {
    private Integer uid;
    private String name;
    private String password;
    private String email;
    private Date birthday;

    //建立从用户到账户的关系
    private List<Account> accounts = new ArrayList<>();
```

```
Account.java
public class Account {
    private Integer aid;
    private String accountName;
    private Float balance;


    //建立从账户到用户的关系
    private User user;
```

**建立User和Role之间的关系**

```
User.java
public class User {
    private Integer uid;
    private String name;
    private String password;
    private String email;
    private Date birthday;

    //建立从用户到角色的关系
    private List<Role> roles = new ArrayList<>();
```

```
Role.java
public class Role {
    private Integer rid;
    private String roleName;
    private String description;


    //建立从角色到用户的关系
    private List<User> users = new ArrayList<>();
```

## 2.2 账户到用户的一对一关系

查询所有账户, 并且查出账户所属的用户信息

### 2.2.1 接口文件

```
public interface AccountDao {
    //查询所有账户, 并且查出账户所属的用户信息
    List<Account> findAllWithUser();
}
```

### 2.2.2 映射文件

```
    <resultMap id="accountMap" type="com.itheima.domain.Account">
        <result column="aid" property="aid"/>
        <result column="accountName" property="accountName"/>
        <result column="balance" property="balance"/>

        <!--
            association : 表示一对一配置
                property : 表示封装到实体对象中哪个属性
                javaType : 封装到属性的类型
```

```
        -->
        <association property="user" javaType="com.itheima.domain.User">
            <result column="uid" property="uid"/>
            <result column="name" property="name"/>
            <result column="password" property="password"/>
            <result column="email" property="email"/>
            <result column="birthday" property="birthday"/>
        </association>
    </resultMap>
    <select id="findAllWithUser" resultMap="accountMap">
        SELECT * FROM account a LEFT JOIN USER u ON a.uid = u.uid
    </select>
```

### 2.2.3 测试

```
public class AccountDaoTest extends BaseUtil {
    @Test
    public void testFindAllWithUser(){
        AccountDao accountDao = sqlSession.getMapper(AccountDao.class);
        List<Account> accounts = accountDao.findAllWithUser();

        for (Account account : accounts) {
            System.out.println(account);
            System.out.println(account.getUser());
            System.out.println("=======================");
        }
    }
}
```

测试结果如下:

```
==>  Preparing: SELECT * FROM account a LEFT JOIN USER u ON a.uid = u.uid
[DEBUG] 2019-12-14 19:44:39,868 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.java:143
==> Parameters:
[DEBUG] 2019-12-14 19:44:39,944 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.java:143
<==      Total: 4
Account{aid=1, accountName='B01', balance=100.0}
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sat Dec 14 00:00:00 CST 2019}
=======================
Account{aid=2, accountName='B02', balance=200.0}
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sat Dec 14 00:00:00 CST 2019}
=======================
Account{aid=3, accountName='B03', balance=300.0}
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sat Dec 14 00:00:00 CST 2019}
=======================
Account{aid=4, accountName='B04', balance=400.0}
User{uid=2, name='传智播客2', password='admin2', email='admin@itcast1.cn', birthday=Sat Jun 01 00:00:00 CST 2019}
=======================
```

---

## 2.3 用户到账户的一对多关系

查询所有用户, 并且查出用户的账户信息

### 2.2.1 接口文件

```
public interface UserDao {
    //查询所有用户, 并且查出用户的账户信息
    List<User> findAllWithAccounts();
}
```

### 2.2.2 映射文件

```
    <resultMap id="userMap" type="com.itheima.domain.User">
        <result column="uid" property="uid"/>
        <result column="name" property="name"/>
        <result column="password" property="password"/>
        <result column="email" property="email"/>
        <result column="birthday" property="birthday"/>

        <!--
            collection : 表示一对多配置
                property : 表示封装到实体对象中哪个属性
                ofType : 表示泛型中的实体类型
        -->
        <collection property="accounts" ofType="com.itheima.domain.Account">
            <result column="aid" property="aid"/>
            <result column="accountName" property="accountName"/>
            <result column="balance" property="balance"/>
        </collection>
    </resultMap>

    <select id="findAllWithAccounts" resultMap="userMap">
        SELECT * FROM USER u LEFT JOIN account a ON a.uid = u.uid
    </select>
```

### 2.2.3 测试

```
public class UserDaoTest extends BaseUtil {
    @Test
    public void testFindAllWithAccounts(){
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findAllWithAccounts();
        for (User user : users) {
            System.out.println(user);
            for (Account account : user.getAccounts()) {
                System.out.println(account);
            }

            System.out.println("====================");
        }
    }
}
```

测试结果如下:

```
==>  Preparing: SELECT * FROM USER u LEFT JOIN account a ON a.uid = u.uid
[DEBUG] 2019-12-14 19:45:22,083 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger
==> Parameters:
[DEBUG] 2019-12-14 19:45:22,113 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger
<==      Total: 6
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sa
Account{aid=1, accountName='B01', balance=100.0}
Account{aid=2, accountName='B02', balance=200.0}
Account{aid=3, accountName='B03', balance=300.0}
====================
User{uid=2, name='传智播客2', password='admin2', email='admin@itcast1.cn', birthday=Sa
Account{aid=4, accountName='B04', balance=400.0}
```

## 2.4 用户到角色的一对多关系

查询所有用户, 并且查出用户的角色信息

### 2.2.1 接口文件

```
public interface UserDao {
    //查询所有用户, 并且查出用户的角色信息
    List<User> findAllWithRoles();
}
```

### 2.2.2 映射文件

```
    <resultMap id="userMap2" type="com.itheima.domain.User">
        <result column="uid" property="uid"/>
        <result column="name" property="name"/>
        <result column="password" property="password"/>
        <result column="email" property="email"/>
        <result column="birthday" property="birthday"/>

        <!--
            collection : 表示一对多配置
                property : 表示封装到实体对象中哪个属性
                ofType : 表示泛型中的实体类型
        -->
        <collection property="roles" ofType="com.itheima.domain.Role">
            <result column="rid" property="rid"/>
            <result column="roleName" property="roleName"/>
            <result column="description" property="description"/>
        </collection>
    </resultMap>

    <select id="findAllWithRoles" resultMap="userMap2">
        SELECT * FROM USER u
            LEFT JOIN user_role ur ON u.uid = ur.uid
            LEFT JOIN role r ON ur.rid = r.rid
    </select>
```

### 2.2.3 测试

```
public class UserDaoTest extends BaseUtil {
    @Test
    public void testFindAllWithRoles(){
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findAllWithRoles();
        for (User user : users) {
            System.out.println(user);
            for (Role role : user.getRoles()) {
                System.out.println(role);
            }
            System.out.println("====================");
        }
    }
}
```

测试结果如下:

```
==> Preparing: SELECT * FROM USER u LEFT JOIN user_role ur ON u.uid = ur.uid LEFT JOIN role r ON ur.rid = r.rid
[DEBUG] 2019-12-14 19:46:18,539 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.java:14
==> Parameters:
[DEBUG] 2019-12-14 19:46:18,569 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLogger.java:14
<==      Total: 5
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sat Dec 14 00:00:00 CST 2019}
Role{rid=1, roleName='出库管理', description='只能对商品进行出库操作'}
Role{rid=2, roleName='入库管理', description='只能对商品进行入库操作'}
====================
User{uid=2, name='传智播客2', password='admin2', email='admin@itcast1.cn', birthday=Sat Jun 01 00:00:00 CST 2019}
Role{rid=1, roleName='出库管理', description='只能对商品进行出库操作'}
====================
```

# 三、嵌套查询

## 3.1 什么是嵌套查询

将一次多表联合查询**尽量**使用多次单表查询来替代( 分步查询 ), 最后将多次查询的结果嵌套组装起来

优点: 每次都是简单的单表

缺点: 使用更加麻烦

## 3.2 从账户到用户的一对一

### 3.2.1 思路分析

```
-- 查询所有账户，并且查出账户所属的用户信息(左外连接)
SELECT * FROM account a LEFT JOIN USER u ON a.uid = u.uid

-- 1 先查询账户信息
SELECT * FROM account;
-- 2 再根据上面得到的uid查询用户信息
SELECT * FROM USER WHERE uid = 1;
SELECT * FROM USER WHERE uid = 2;
```

### 3.2.2 AccountDao接口文件

```
public interface AccountDao {
    //查询所有账户，并且查出账户所属的用户信息
    List<Account> findAllWithUser();
}
```

### 3.2.3 AccountDao映射文件

```
    <resultMap id="accountMap" type="com.itheima.domain.Account">
        <result column="aid" property="aid"/>
        <result column="accountName" property="accountName"/>
        <result column="balance" property="balance"/>
        <!--
        association : 一对一配置
         property="user"  User实体类中的属性字段
         javaType="User" 返回的数据字段类型
         后两个配置表示数据的来源
         select="com.itheima.dao.UserDao.findByUid"
            表示调用另一个配置文件(UserDao映射文件)中的根据id查询方法
         column="uid"  表示根据用户的id传入到findByUid方法中查询
```

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

### 3.2.4 UserDao接口文件

```
public interface UserDao {
    //根据uid查询user
    User findByUid(Integer uid);
}
```

### 3.2.5 UserDao映射文件

```
    <!--根据uid查询user-->
    <select id="findByUid" resultType="com.itheima.domain.User">
        SELECT * FROM USER WHERE uid = #{uid};
    </select>
```

### 3.2.6 测试

```
public class AccountDaoTest extends BaseUtil {
    @Test
    public void testFindAllWithUser(){
        AccountDao accountDao = sqlSession.getMapper(AccountDao.class);
        List<Account> accounts = accountDao.findAllWithUser();
        for (Account account : accounts) {
            System.out.println(account.getUser());
            System.out.println("=======================");
        }
    }
}
```

测试结果如下:

```
==>  Preparing: SELECT * FROM account   1
[DEBUG] 2019-12-14 19:55:01,985 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.
==> Parameters:
[DEBUG] 2019-12-14 19:55:02,086 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.
<==      Total: 4
[DEBUG] 2019-12-14 19:55:02,087 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.
==>  Preparing: SELECT * FROM USER WHERE uid = ?;   2
[DEBUG] 2019-12-14 19:55:02,087 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.
==> Parameters: 1(Integer)
[DEBUG] 2019-12-14 19:55:02,092 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.
<==      Total: 1
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sat
=======================
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sat
=======================
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sat
=======================
[DEBUG] 2019-12-14 19:55:02,093 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.
==>  Preparing: SELECT * FROM USER WHERE uid = ?;   3
[DEBUG] 2019-12-14 19:55:02,096 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.
==> Parameters: 2(Integer)
[DEBUG] 2019-12-14 19:55:02,097 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.
```

## 3.3 从用户到账户的一对多

### 3.3.1 思路分析

```sql
-- 查询所有用户，并且查出用户的账户信息(左外连接)
SELECT * FROM USER u LEFT JOIN account a ON a.uid = u.uid

-- 1 查询所有用户
SELECT * FROM USER;
-- 2 根据上面得到的uid查询账户
SELECT * FROM account WHERE uid = 1;
SELECT * FROM account WHERE uid = 2;
```

### 3.3.2 UserDao接口文件

```java
public interface UserDao {
    //查询所有用户，并且查出用户的账户信息
    List<User> findAllWithAccounts();
}
```

### 3.3.3 UserDao映射文件

```xml
    <resultMap id="userMap" type="com.itheima.domain.User">
        <result column="uid" property="uid"/>
        <result column="name" property="name"/>
        <result column="password" property="password"/>
        <result column="email" property="email"/>
        <result column="birthday" property="birthday"/>

        <!--
                collection : 表示一对多配置
                property="accounts"  User实体类中的属性字段
                ofType="com.itheima.domain.Account" 返回的List集合中的属性类型
                后两个配置表示数据的来源
                select="com.itheima.dao.AccountDao.findAccountByUid"
                    表示调用另一个配置文件(AccountDao映射文件)中的根据id查询方法
                column="id"  表示根据用户的id传入到findByUid方法中 查询
        -->
        <collection property="accounts" ofType="com.itheima.domain.Account"
            select="com.itheima.dao.AccountDao.findByUid" column="uid"
        >
        </collection>
    </resultMap>
```

### 3.3.4 AccountDao接口文件

```java
public interface AccountDao {
    //根据uid查询到所有的账户
    List<Account> findByUid(Integer uid);
}
```

### 3.3.5 AccountDao映射文件

```
    <select id="findByUid" resultType="com.itheima.domain.Account"
            useCache="true"  flushCache="true">
        SELECT * FROM account WHERE uid = #{uid}
    </select>
```

### 3.3.6 测试

```
public class UserDaoTest extends BaseUtil {
    @Test
    public void testFindAllWithAccounts(){
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findAllWithAccounts();
        for (User user : users) {
            System.out.println(user);
            for (Account account : user.getAccounts()) {
                System.out.println(account);
            }

            System.out.println("====================");
        }
    }
}
```

测试结果如下:

```
==> Preparing: SELECT * FROM USER   1
[DEBUG] 2019-12-14 20:00:20,965 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJ
==> Parameters:
[DEBUG] 2019-12-14 20:00:20,991 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJ
====>  Preparing: SELECT * FROM account WHERE uid = ?   2
[DEBUG] 2019-12-14 20:00:20,992 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJ
====> Parameters: 1(Integer)
[DEBUG] 2019-12-14 20:00:20,997 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJ
<====      Total: 3
[DEBUG] 2019-12-14 20:00:21,000 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJ
====>  Preparing: SELECT * FROM account WHERE uid = ?   3
[DEBUG] 2019-12-14 20:00:21,001 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJ
====> Parameters: 2(Integer)
[DEBUG] 2019-12-14 20:00:21,003 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJ
<====      Total: 1
[DEBUG] 2019-12-14 20:00:21,005 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJ
<==      Total: 2
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sat Dec 14 00:
Account{aid=1, accountName='B01', balance=100.0}
Account{aid=2, accountName='B02', balance=200.0}
Account{aid=3, accountName='B03', balance=300.0}
====================
User{uid=2, name='传智播客2', password='admin2', email='admin@itcast1.cn', birthday=Sat Jun 01 00:
Account{aid=4, accountName='B04', balance=400.0}
====================
```

## 3.4 从用户到角色的一对多

### 3.3.1 思路分析

```sql
-- 查询所有用户，并且查出用户的角色信息
SELECT * FROM USER u
        LEFT JOIN user_role ur ON u.uid = ur.uid
        LEFT JOIN role r ON ur.rid = r.rid


-- 查询所有用户
SELECT * FROM USER
-- 根据uid查询角色信息
SELECT * FROM user_role ur LEFT JOIN role r ON ur.rid = r.rid WHERE ur.uid = 1;
SELECT * FROM user_role ur LEFT JOIN role r ON ur.rid = r.rid WHERE ur.uid = 2;
```

### 3.3.2 UserDao接口文件

```java
public interface UserDao {
    //查询所有用户，并且查出用户的角色信息
    List<User> findAllWithRoles();
}
```

### 3.3.3 UserDao映射文件

```xml
    <resultMap id="userMap2" type="com.itheima.domain.User">
        <result column="uid" property="uid"/>
        <result column="name" property="name"/>
        <result column="password" property="password"/>
        <result column="email" property="email"/>
        <result column="birthday" property="birthday"/>

        <!--
            当select要调用的sql语句就在当前文件的时候,可以直接使用statementId表示,省略
namespace
        -->
        <collection property="roles" ofType="com.itheima.domain.Role"
            select="findRolesByUid" column="uid">
        </collection>
    </resultMap>

    <select id="findAllWithRoles" resultMap="userMap2">
        SELECT * FROM USER
    </select>

    <select id="findRolesByUid" resultType="com.itheima.domain.Role">
        SELECT * FROM user_role ur LEFT JOIN role r ON ur.rid = r.rid
        WHERE ur.uid = #{uid};
    </select>
```

### 3.3.4 测试

```java
public class UserDaoTest extends BaseUtil {
    @Test
    public void testFindAllWithRoles(){
        UserDao userDao = sqlSession.getMapper(UserDao.class);
        List<User> users = userDao.findAllWithRoles();
        for (User user : users) {
            System.out.println(user);
            for (Role role : user.getRoles()) {
                System.out.println(role);
```

```
            }
            System.out.println("====================");
        }
    }
}
```

测试结果如下:

```
==> Preparing: SELECT * FROM USER
[DEBUG] 2019-12-14 20:02:33,672 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLog
==> Parameters:
[DEBUG] 2019-12-14 20:02:33,697 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLog
====>  Preparing: SELECT * FROM user_role ur LEFT JOIN role r ON ur.rid = r.rid WHERE ur.uid = ?;  2
[DEBUG] 2019-12-14 20:02:33,698 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLog
====> Parameters: 1(Integer)
[DEBUG] 2019-12-14 20:02:33,702 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLog
<====      Total: 2
[DEBUG] 2019-12-14 20:02:33,704 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLog
====>  Preparing: SELECT * FROM user_role ur LEFT JOIN role r ON ur.rid = r.rid WHERE ur.uid = ?;  3
[DEBUG] 2019-12-14 20:02:33,705 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLog
====> Parameters: 2(Integer)
[DEBUG] 2019-12-14 20:02:33,707 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLog
<====      Total: 1
[DEBUG] 2019-12-14 20:02:33,708 method:org.apache.ibatis.logging.jdbc.BaseJdbcLogger.debug(BaseJdbcLog
<==      Total: 2
User{uid=1, name='传智播客1', password='admin1', email='admin@itcast1.cn', birthday=Sat Dec 14 00:00:00
Role{rid=1, roleName='出库管理', description='只能对商品进行出库操作'}
Role{rid=2, roleName='入库管理', description='只能对商品进行入库操作'}
====================
User{uid=2, name='传智播客2', password='admin2', email='admin@itcast1.cn', birthday=Sat Jun 01 00:00:00
Role{rid=1, roleName='出库管理', description='只能对商品进行出库操作'}
```

---

# 四、加载策略

## 4.1 什么是加载策略

当多个模型之间存在关联关系时, 加载一个模型的同时, 是否要立即加载其关联的模型, 我们把这种决策成为加载策略.

如果加载一个模型的时候, 需要立即加载出其关联的所有模型, 这种策略称为立即加载

如果加载一个模型的时候, 不需要立即加载出其关联的所有模型, 等到真正需要的时候再加载, 这种策略称为延迟加载(懒加载).

## 4.2 Mybatis的加载策略

Mybatis中的加载策略有两种:立即加载和延迟加载， 默认加载策略是**立即加载**，也就是在加载一个对象的时候会立即联合加载到其关联的对象。当然，Mybatis也提供了修改加载策略的方法。

**全局修改**

在Mybatis 的配置文件中可以使用setting修改全局的加载策略。

```
<setting name="lazyLoadingEnabled" value="true|false（默认值）"/>
```

**局部修改**

在和元素中都有一个fetchType属性，该值会覆盖掉全局参数的配置。

- fetchType="lazy" 懒加载策略
- fetchType="eager" 立即加载策略

```
<association fetchType="eager|lazy"></association>
<collection fetchType="eager|lazy"></collection>
```

**注意**

- 在配置了延迟加载策略后，即使没有调用关联对象的任何方法，当你调用当前对象的equals、clone、hashCode、toString方法时也会触发关联对象的查询。
- 在配置文件中可以使用lazyLoadTriggerMethods配置项覆盖掉mybatis的默认行为。

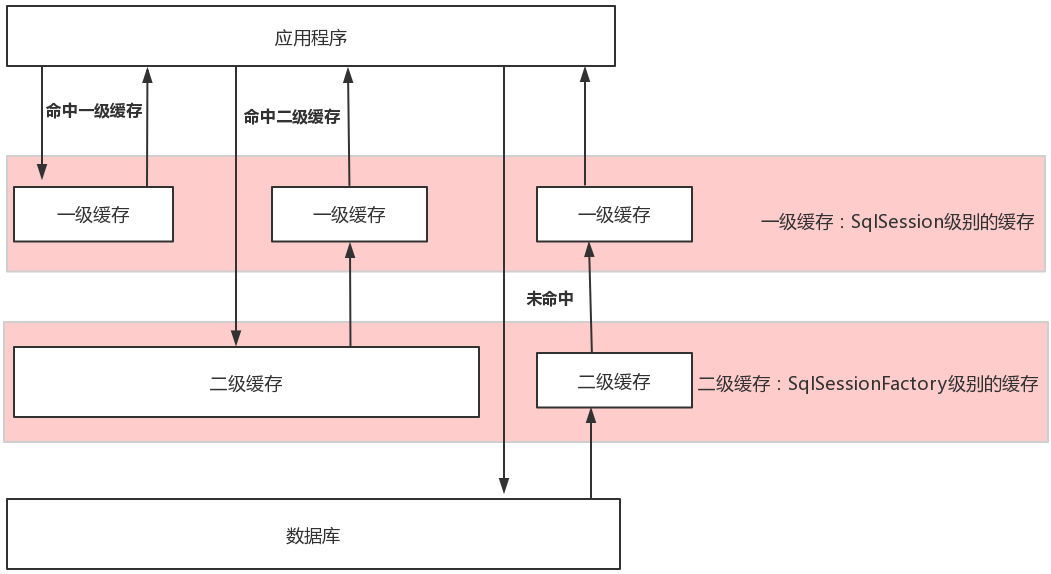
```
<setting name="lazyLoadTriggerMethods" value="getUid,toString"/>
```

---

# 五、缓存机制

缓存是用来提高查询效率的，所有的持久层框架基本上都有缓存机制。

Mybatis有两级缓存，一级缓存是SqlSession级别的，二级缓存是映射器级别的。




## 3.1 一级缓存

一级缓存是SqlSession级别的缓存，是默认开启且无法关闭的。

```
public void testFindByUid(){
            InputStream stream =
Resources.getResourceAsStream("sqlMapConfig.xml");
        SqlSessionFactory factory = new
SqlSessionFactoryBuilder().build(stream);
        SqlSession sqlSession = factory.openSession();

        AccountDao accountDao1 = sqlSession.getMapper(AccountDao.class);
        List<Account> a1 = accountDao1.findByUid(1);//发送SQL

        System.out.println("==================================================");
```

```
        AccountDao accountDao2 = sqlSession.getMapper(AccountDao.class);
        List<Account> a2 = accountDao2.findByUid(1);//没有发送SQL

        System.out.println(a1 == a2);//内存地址相等
        sqlSession.commit();
        sqlSession.close();
}
```

- 同一个sqlSession中两次执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。
- 当一个sqlSession结束后该sqlSession中的一级缓存也就不存在了。
- 不同的sqlSession之间的缓存数据区域（HashMap）是互相不影响的。

注意:

- 调用SqlSession的clearCache()，或者执行C（增加）U（更新）D（删除）操作，都会清空一级缓存。
- 查询语句中这样的配置< select flushCache="true"/>也会清除一级缓存。

## 3.2 二级缓存

二级缓存是映射器级别的缓存，是默认开启，但是可以关闭的。

二级缓存是多个SqlSession共享的，其作用域是mapper的同一个namespace，不同的sqlSession两次执行相同namespace下的sql语句且向sql中传递参数也相同即最终执行相同的sql语句，第一次执行完毕会将数据库中查询的数据写到缓存（内存），第二次会从缓存中获取数据将不再从数据库查询，从而提高查询效率。

**二级缓存的使用条件**

1 在主配置文件中开启二级缓存(默认)

```
sqlMapConfig.xml
<!--全局配置-->
<settings>
    <!--开启二级缓存  默认值是true开启的-->
    <setting name="cacheEnabled" value="true"/>
```

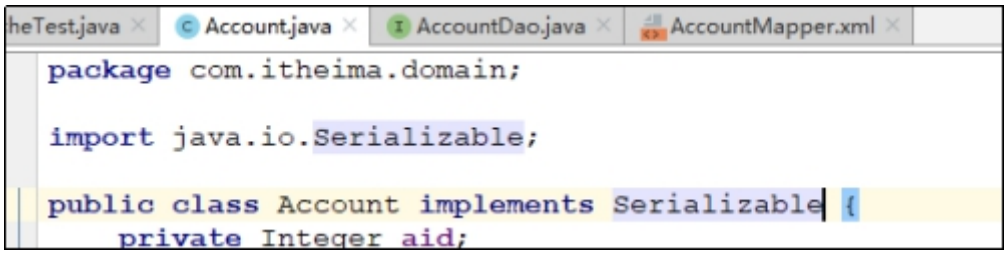
2 在Mapper.xml中开启二级缓存

```
CacheTest.java   AccountDao.java   AccountMapper.xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.itheima.dao.AccountDao">
    <!--开启二级缓存-->
    <cache/>
```

3 在Select标签中开启二级缓存(默认)

```
CacheTest.java   AccountDao.java   AccountMapper.xml
<mapper namespace="com.itheima.dao.AccountDao">
    <!--开启二级缓存-->
    <cache/>

    <!--根据uid查询到所有的账户
        useCache="true"  使用缓存  默认值true
    -->
    <select id="findByUid" resultType="com.itheima.domain.Account" useCache="true">
        SELECT * FROM account WHERE uid = #{uid}
    </select>
```

4 开启实体类的序列化




验证：

```
    //测试二级缓存
     @Test
     public void testTwoCache() throws IOException {
         InputStream stream = Resources.getResourceAsStream("sqlMapConfig.xml");
         SqlSessionFactory factory = new
SqlSessionFactoryBuilder().build(stream);

         SqlSession sqlSession1 = factory.openSession();
         AccountDao accountDao1 = sqlSession1.getMapper(AccountDao.class);
         List<Account> a1 = accountDao1.findByUid(1);//发送SQL
         sqlSession1.commit();
         sqlSession1.close();

         System.out.println("==============================================");

         SqlSession sqlSession2 = factory.openSession();
         AccountDao accountDao2 = sqlSession2.getMapper(AccountDao.class);
         List<Account> a2 = accountDao2.findByUid(1);//没有发送
         sqlSession2.commit();
         sqlSession2.close();

         //注意这里得到的对象是从缓存中拷贝出来的
         System.out.println(a1 == a2);//false
     }
```