MyBatis笔记

> MyBatis介绍

MyBatis 是一款优秀的持久层框架。

MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。

官网：https://mybatis.org/mybatis-3/zh/#

1.1、Maven引入MyBatis流程

1. **首先配置maven的版本以及配置文件，也可以使用默认配置。**
2. **点击project structure，创建new module，选择maven，配置groupId和artifactId**
3. **编写测试代码**

①数据准备

```
CREATE DATABASE `mybatis_db`;
USE `mybatis_db`;
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(50) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `address` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT
CHARSET=utf8;
insert  into `user`(`id`,`username`,`age`,`address`)
values (1,'UZI',19,'上海'),(2,'PDD',25,'上海');
```

②导入依赖pom.xml

```
```

```
<dependencies>

    <!--mybatis依赖-->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.5.4</version>
    </dependency>

    <!-- junit测试 -->
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>
    <!-- junit测试,与8版本JDK对应 -->
    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>6.14.3</version>
    </dependency>

    <!-- log4j日志 -->
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
    </dependency>

    <!--mysql驱动-->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-
java</artifactId>
        <version>5.1.47</version>
    </dependency>
```

```
</dependencies>
```

③编写核心配置

在资源目录下(resource)创建: mybatis-config.xml 内容如下:

```
<?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>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
                <property name="driver"
value="com.mysql.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3306/mybatis_db"/>
                <property name="username"
value="root"/>
                <property name="password"
value="302305"/>
            </dataSource>
        </environment>
    </environments>
    <mappers>
        <mapper resource="com/hq/dao/UserDao.xml"/>
    </mappers>
</configuration>
```

④定义接口及对应的xml映射文件

com.hq.pojo.User

```
public class User {

    int id;
    String username;
    int age;
    String address;
}
```

com.hq.dao.UserDao:

```
public interface UserDao {
    List<User> findAll();
}
```

资源目录(resource)下: com/hq/dao/UserDao.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.hq.dao.UserDao">

    <select id="findAll"
resultType="com.hq.pojo.User">
        select * from user
    </select>
</mapper>
```

⑤编写测试类

获取SqlSession,通过SqlSession获取UserDao调用对应的方法

```
@Test
public void test_Mybatis() throws IOException {
    //定义mybatis配置文件的路径
    String resource = "mybatis-config.xml";
```

```
    InputStream inputStream =
Resources.getResourceAsStream(resource);
    SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
    //获取Sqlsession对象
    SqlSession sqlSession =
sqlSessionFactory.openSession();
    //获取UserDao实现类对象
    UserDao userDao =
sqlSession.getMapper(UserDao.class);
    //调用方法测试
    List<User> userList = userDao.findAll();
    System.out.println(userList);
    //释放资源
    sqlSession.close();
}
```

1.2、参数获取

1.2.1 一个参数

1.2.1.1 基本类型参数

我们可以使用#{}直接来取值，写任意名字都可以获取到参数。但是**一般用方法的参数名来取**。

例如：接口中方法定义如下

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

xml中内容如下：

```
<select id="findById" resultType="com.hq.pojo.User">

    select * from user where id = #{id}
</select>
```

测试代码内容如下：

```
User user = userDao.findById(1);
System.out.println(user);
```

1.2.1.2 POJO的实体属性名

我们可以**使用POJO中的实体属性名**来获取对应的值。

例如：接口中方法定义如下

```
User findByUser(User user);
```

xml中内容如下：

```
<select id="findByUser"
resultType="com.hq.pojo.User">
    select * from user where where id = #{id}
</select>
```

测试代码内容如下：

```
User user = new User();
user.setId(1);
User user_1 = userDao.findByUser(user);
System.out.println(user_1);
```

1.2.2 多个参数

1.2.1.1 Map中的key

我们可以使用**map中的key**来获取对应的值。

例如：接口中方法定义如下

```
User findByMap(Map map);
```

xml中内容如下：

```
<select id="findByMap"
resultType="com.hq.pojo.User">
    select * from user where id = #{id} and username
= #{username}
</select>
```

方法调用：

```
Map map = new HashMap();
map.put("id",1);
map.put("username","UZI");
User user_2 = userDao.findByMap(map);
System.out.println(user_2);
```

1.2.2.2 @Param

Mybatis会把多个参数放入一个Map集合中，默认的key是argx和paramx这种格式。

例如：接口中方法定义如下

```
User findByCondition(Integer id,String username);
```

最终map中的键值对如下：

```
{arg1=PDD, arg0=2, param1=2, param2=PDD}
```

我们虽然可以使用对应的默认key来获取值，但是这种方式可读性不好。我们一般在方法**参数前使用@Param来设置参数名。**

例如：接口中方法定义

```
User findByCondition(@Param("id") Integer
id,@Param("username") String username);
```

最终map中的键值对如下：

```
{id=2, param1=2, username=PDD, param2=PDD}
```

所以我们就可以使用如下方式来获取参数

```
<select id="findByCondition"
resultType="com.hq.pojo.User">
    select * from user where id = #{id} and username
= #{username}
</select>
```

测试代码

```
User user = new User();
user.setId(1);
user.setUsername("UZI");
User user_3 = userDao.findByCondition(user.getId(),
user.getUsername());
System.out.println(user_3);
```

建议如果只有一个参数的时候不用做什么特殊处理。**如果是有多个参数的情况下一定要加上@Param来设置参数名，或者使用Map。**

2、Mybatis实现增删改查

2.1 新增

①接口中增加相关方法

```
void insertUser(User user);
```

②映射文件UserDao.xml增加响应的标签

```
# 自动增长的id
<insert id="insertUser">
    insert into user values(null,#{username},#
{age},#{address})
</insert>
```

注意：要记得提交事务。

```
//调用方法测试
userDao.insertUser(new User("若风",18,"上海"));
sqlSession.commit();
```

2.2 删除

①接口中增加相关方法

```
void deleteById(Integer id);
```

②映射文件UserDao.xml增加响应的标签

```
<delete id="deleteById">
    delete from user where id = #{id}
</delete>
```

注意：要记得提交事务。

```
//调用方法测试
userDao.deleteById(1);
sqlSession.commit();
```

2.3 修改

①接口中增加相关方法

```
void updateUser(User user);
```

②映射文件UserDao.xml增加响应的标签

```
<!—更新用户—>
<update id="updateUser">
    update user set username = #{username},age = #
{age},address = #{address} where id = #{id}
</update>
```

注意：要记得提交事务。

```
User userDaoById = userDao.findById(2);
userDaoById.setUsername("大司马");
userDao.updateUser(userDaoById);
sqlSession.commit();
```

2.4 查询

①接口中增加相关方法

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

②映射文件UserDao.xml增加响应的标签

```
<select id="findById" resultType="com.hq.pojo.User">
    select * from user where id = #{id}
</select>
```

①接口中增加相关方法

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

②映射文件UserDao.xml增加响应的标签

```
<select id="findAll" resultType="com.hq.pojo.User">
    select * from user
</select>
```

3、配置文件详解

3.1、properties

可以使用properties，读取properties配置文件。使用其中的resource属性来设置配置文件的路径。

然后使用${key}来获取配置文件中的值。

例如：在**resources目录下有jdbc.properties文件**，内容如下：

```
jdbc.driver=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mybatis_db
jdbc.username=root
jdbc.password=302305
```

在mybatis-config.xml中：配置文件地址

```
<?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>
    <!—设置配置文件所在的路径—>
    <properties resource="jdbc.properties">
</properties>
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>
            <dataSource type="POOLED">
```

```
                <!--获取配置文件中配置的对应的值来设置连接
相关参数-->
                <property name="driver"
value="${jdbc.driver}"/>
                <property name="url"
value="${jdbc.url}"/>
                <property name="username"
value="${jdbc.username}"/>
                <property name="password"
value="${jdbc.password}"/>
            </dataSource>
        </environment>
    </environments>
</configuration>
```

3.2 settings

可以使用该标签来设置进行一些设置

例如：

```
<settings>
    <!--开启自动驼峰命名映射-->
    <setting name="mapUnderscoreToCamelCase"
value="true"/>
</settings>
```

具体的设置参考：https://mybatis.org/mybatis-3/zh/configuration.html#settings

3.3 typeAliases

可以用来设置**给实体类包下的所有文件设置别名。默认别名是类名首字母小写。**例如：com.sangeng.pojo.User**别名为user**

```
<typeAliases>
    <package name="com.hq.pojo"></package>
</typeAliases>
```

3.4 environments

配置数据库相关的环境，例如事物管理器，连接池相关参数等。

```
<!--设置默认环境-->
<environments default="development">

    <!--设置该环境的唯一标识-->
    <environment id="development">
        <transactionManager type="JDBC"/>
        <dataSource type="POOLED">
            <!--获取配置文件中配置的对应的值来设置连接相关
参数-->
            <property name="driver"
value="${jdbc.driver}"/>
            <property name="url"
value="${jdbc.url}"/>
            <property name="username"
value="${jdbc.username}"/>
            <property name="password"
value="${jdbc.password}"/>
        </dataSource>
    </environment>
</environments>
```

3.5 mappers

该标签的作用是加载映射的，加载方式有如下几种（**主要使用第四种**）：

①使用相对于类路径的资源引用，例如：

```
<!-- 使用相对于类路径的资源引用 -->
<mappers>
  <mapper
resource="org/mybatis/builder/AuthorMapper.xml" />
  <mapper
resource="org/mybatis/builder/BlogMapper.xml" />
  <mapper
resource="org/mybatis/builder/PostMapper.xml" />
</mappers>
```

②使用完全限定资源定位符（URL），例如：

```
<!-- 使用完全限定资源定位符（URL） -->
<mappers>
  <mapper
url="file:///var/mappers/AuthorMapper.xml" />
  <mapper url="file:///var/mappers/BlogMapper.xml" />
  <mapper url="file:///var/mappers/PostMapper.xml" />
</mappers>
```

③使用映射器接口实现类的完全限定类名，例如：

```
<!-- 使用映射器接口实现类的完全限定类名 -->
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper" />
  <mapper class="org.mybatis.builder.BlogMapper" />
  <mapper class="org.mybatis.builder.PostMapper" />
</mappers>
```

④将包内的映射器接口实现全部注册为映射器，例如：

```
<!-- 定义dao接口所在的包。要求xml文件存放的路径和dao接口的包
名要对应 -->
<mappers>
  <package name="com.hq.dao" />
</mappers>
```

4、配置log4j

在resources目录下创建log4j.properties文件，内容如下

```
### direct log messages to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppend
er
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.Patter
nLayout
log4j.appender.stdout.layout.ConversionPattern=%d{AB
SOLUTE} %5p %c{1}:%L - %m%n

### direct messages to file mylog.log ###
log4j.appender.file=org.apache.log4j.FileAppender
log4j.appender.file.File=c:/mylog.log
log4j.appender.file.layout=org.apache.log4j.PatternL
ayout
log4j.appender.file.layout.ConversionPattern=%d{ABSO
LUTE} %5p %c{1}:%L - %m%n

### set log levels - for more verbose logging change
'info' to 'debug' ###

log4j.rootLogger=debug, stdout
```

引入依赖

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

5、动态SQL

在实际开发中的SQL语句没有之前的这么简单，很多时候需要根据传入的参数情况动态的生成SQL语句。Mybatis提供了动态SQL相关的标签让我们使用。

5.1 if

可以**使用if标签进行条件判断，条件成立才会把if标签中的内容拼接进sql语句。**

定义接口方法：

```
User findByUserName(@Param("id") int id,
@Param("username") String username);
```

定义sql语句：

```
<select id="findByCondition"
resultType="com.hq.pojo.User">
    select * from user
    where  id = #{id}
    <if test="username≠null">
        and username = #{username}
    </if>
</select>
```

测试方法：

```
User userDaoByUserName = userDao.findByUserName(2,
"pdd");
System.out.println(userDaoByUserName);
```

如果参数username为null则执行的sql为：**select * from user where id = ?**

如果参数username不为null则执行的sql为：**select * from user where id = ? and username = ?**

注意：在test属性中表示参数的时候不需要写#{}，写了会出问题。

5.2 trim

可以使用该标签**动态的添加前缀或后缀，也可以使用该标签动态的消除前缀。**

5.2.1 prefixOverrides属性

用来**设置需要被清除的前缀，多个值可以用|分隔，注意|前后不要有空格。**
例如： and|or

5.2.2 suffixOverrides属性

用来设置需要被**清除的后缀**，多个值可以用|分隔，注意|前后不要有空格。
例如： and|or

最终执行的sql为： select * from user where id = ? 去掉了前缀and和后缀like

```
<select id="findByUserName" resultType="user">
    select * from user
    where
    <trim prefixOverrides="and|or"
suffixOverrides="like">
        and id = #{id} like
    </trim>
    <if test="username≠null">
        and username = #{username}
    </if>
</select>
```

5.2.3 prefix属性

用来设置**动态添加的前缀**，如果**标签中有内容就会添加上设置的前缀**

例如：

```
<select id="findByCondition"
resultType="com.hq.pojo.User">
    select * from user
    <trim prefix="where" >
        1=1
    </trim>
</select>
```

最终执行的sql为：`select * from user where 1=1`　动态增加了前缀where

5.2.4 suffix属性

用来设置**动态添加的后缀**，如果标签中有内容就会添加上设置的后缀

```
<select id="findByCondition"
resultType="com.sangeng.pojo.User">
    select * from user
    <trim suffix="1=1" >
        where
    </trim>
</select>
```

最终执行的sql为：`select * from user where 1=1`　动态增加了后缀1=1

5.2.5 动态添加前缀where 并且消除前缀and或者or

```
User findByCondition(@Param("id") Integer
id,@Param("username") String username);
```

```
<select id="findByCondition"
resultType="com.sangeng.pojo.User">
    select * from user
    <trim prefix="where" prefixOverrides="and|or" >
        <if test="id≠null">
            id = #{id}
        </if>
        <if test="username≠null">
            and username = #{username}
        </if>
    </trim>
</select>
```

- 调用方法时如果传入的id和username为null则执行的SQL为：select * from user
- 调用方法时如果传入的id不为空，username为null，则执行的SQL为：select * from user where id= ?
- **调用方法时如果传入的id为null，username不为空，则执行的SQL为：select * from user where username = ?**

5.3 where

where标签等价于：

```
<trim prefix="where" prefixOverrides="and|or" >
</trim>
```

可以使用where标签动态的拼接where并且去除前缀的and或者or。

例如：

```
<select id="testWhere"
resultType="com.hq.pojo.User">
    select * from user
    <where>
        <if test="id≠null">
            id = #{id}
        </if>
        <if test="username≠null">
            and username = #{username}
        </if>
    </where>
</select>
```

如果id和username都为null，则执行的sql为：**select * from user**

如果id为null，username不为null，则执行的sql为：**select * from user where username = ?**

5.4 set

set标签等价于,**suffixOverrides清除末尾','**

```
<trim prefix="set" suffixOverrides="," ></trim>
```

可以使用set标签动态的拼接set并且去除后缀的逗号。

例如：**如果传入的值都为空，最后的sql语句执行不了**

```
<update id="updateUser">
    UPDATE USER
    <set>
        <if test="username≠null">
            username = #{username},
        </if>
        <if test="age≠null">
            age = #{age},
```

```
        </if>
        <if test="address≠null">
            address = #{address},
        </if>
    </set>
    where id = #{id}
</update>
```

如果调用方法时传入的User对象的id为2，username不为null，其他属性都为null则最终执行的sql为：UPDATE USER SET username = ? where id = 2

5.5 foreach

可以使用**foreach标签**遍历集合或者数组类型的参数，获取其中的元素拿来动态的拼接SQL语句。

例如：方法定义如下

```
List<User> findByIds(@Param("ids") Integer[] ids);
```

如果期望动态的根据实际传入的数组的长度拼接SQL语句。例如传入长度为4个数组最终执行的SQL为：

```
select * from User WHERE id in( ? , ? , ? , ?, ? )
```

则在xml映射文件中可以使用以下写法

```
<select id="findByIds"
resultType="com.hq.pojo.User">
    select * from User
    <where>
        <foreach collection="ids" open="id in("
close=")" item="id" separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

```
List<User> userDaoByIds = userDao.findByIds(new
Integer[]{2, 3});
System.out.println(userDaoByIds);
```

collection：表示要遍历的参数，与方法入参名称相同。

open：表示遍历开始时拼接的语句

item：表示给当前遍历到的元素的取的名字，与#{}内部参数名称相同

separator：表示每遍历完一次拼接的分隔符

close：表示最后一次遍历完拼接的语句

注意：如果方法参数是数组类型，默认的参数名是array，如果方法参数是list集合默认的参数名是list。建议遇到数组或者集合类型的参数统一使用@Param注解进行命名。

5.6 choose、when、otherwise

当我们不想使用所有的条件，而只是**想从多个条件中只选择一个使用，并且有优先级时。可以使用choose系列标签。**类似于java中的switch。

例如：接口中方法定义如下

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

- 如果user对象的id不为空时就通过id查询。

- 如果id为null,username不为null就通过username查询。
- 如果id和username都会null就查询id为3的用户

xml映射文件如下

```
<select id="selectChose"
resultType="com.hq.pojo.User">
    select * from user
    <where>
        <choose>
            <when test="id≠0">
                    id = #{id}
            </when>
            <when test="username≠null">
                username = #{username}
            </when>
            <otherwise>
                id = 3
            </otherwise>
        </choose>
    </where>
</select>
```

- **choose类似于java中的switch**
- **when类似于java中的case**
- **otherwise类似于java中的default**

一个choose标签中最多只会有一个when中的判断成立。从上到下去进行判断。如果成立了就把标签体的内容拼接到sql中，并且不会进行其它when的判断和拼接。如果所有的when都不成立则拼接otherwise中的语句。

6、开发实例

6.1、案例环境

6.1.1 案例数据初始化sql

```
CREATE DATABASE /*!32312 IF NOT EXISTS*/`mybatis_db`
/*!40100 DEFAULT CHARACTER SET utf8 */;

USE `mybatis_db`;
DROP TABLE IF EXISTS `orders`;
CREATE TABLE `orders` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `createtime` timestamp NOT NULL DEFAULT
CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
COMMENT '创建时间',
  `price` int(11) DEFAULT NULL COMMENT '价格',
  `remark` varchar(100) DEFAULT NULL COMMENT '备注',
  `user_id` int(11) DEFAULT NULL COMMENT '用户id',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT
CHARSET=utf8;
insert  into
`orders`(`id`,`createtime`,`price`,`remark`,`user_id
`) values (1,'2014-06-26 16:55:43',2000,'无',2),
(2,'2021-02-23 16:55:57',3000,'无',3),(3,'2021-02-23
16:56:21',4000,'无',2);
DROP TABLE IF EXISTS `role`;

CREATE TABLE `role` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(100) DEFAULT NULL COMMENT '角色名',
  `desc` varchar(100) DEFAULT NULL COMMENT '角色描
述',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT
CHARSET=utf8;
```

```sql
/*Data for the table `role` */

insert  into `role`(`id`,`name`,`desc`) values
(1,'总经理','一人之下'),(2,'CFO',NULL);

/*Table structure for table `user` */

DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(50) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `address` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=33 DEFAULT
CHARSET=utf8;

/*Data for the table `user` */

insert  into `user`(`id`,`username`,`age`,`address`)
values (2,'pdd',26,NULL),(3,'UZI',19,'上海11'),
(4,'RF',19,NULL);

/*Table structure for table `user_role` */

DROP TABLE IF EXISTS `user_role`;

CREATE TABLE `user_role` (
  `user_id` int(11) DEFAULT NULL,
  `role_id` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

/*Data for the table `user_role` */

insert  into `user_role`(`user_id`,`role_id`) values
(2,2),(2,1),(3,1);
```

6.1.2 实体类

User.java

```
public class User {

    private Integer id;

    private String username;

    private Integer age;

    private String address;


    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", age=" + age +
                ", address='" + address + '\'' +
                '}';
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }
```

```
    public void setUsername(String username) {
        this.username = username;
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public User() {
    }

    public User(Integer id, String username, Integer
age, String address) {
        this.id = id;
        this.username = username;
        this.age = age;
        this.address = address;
    }

}
```

Order.java

```
```

```java
public class Order {
    private Integer id;
    private Date createtime;
    private Integer price;
    private String remark;
    private Integer userId;

    @Override
    public String toString() {
        return "Order{" +
                "id=" + id +
                ", createtime=" + createtime +
                ", price=" + price +
                ", remark='" + remark + '\'' +
                ", userId=" + userId +
                '}';
    }
    public Order() {
    }
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public Date getCreatetime() {
        return createtime;
    }

    public void setCreatetime(Date createtime) {
        this.createtime = createtime;
    }

    public Integer getPrice() {
        return price;
```

```
    }

    public void setPrice(Integer price) {
        this.price = price;
    }

    public String getRemark() {
        return remark;
    }

    public void setRemark(String remark) {
        this.remark = remark;
    }

    public Integer getUserId() {
        return userId;
    }

    public void setUserId(Integer userId) {
        this.userId = userId;
    }

    public Order(Integer id, Date createtime,
Integer price, String remark, Integer userId) {
        this.id = id;
        this.createtime = createtime;
        this.price = price;
        this.remark = remark;
        this.userId = userId;
    }
}
```

Role.java

```
public class Role {
```

```
    private Integer id;
    private String name;
    private String desc;

    @Override
    public String toString() {
        return "Role{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", desc='" + desc + '\'' +
                '}';
    }
    public Role() {
    }
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDesc() {
        return desc;
    }

    public void setDesc(String desc) {
        this.desc = desc;
    }
```

```
    public Role(Integer id, String name, String
desc) {
        this.id = id;
        this.name = name;
        this.desc = desc;
    }
}
```

6.2、SQL片段抽取

我们在xml映射文件中编写SQL语句的时候可能会遇到**重复的SQL片段**。这种SQL片段我们可以使用sql标签来进行抽取。然后在需要使用的时候使用include标签进行使用。

例如:

```
<sql id="baseSelect" >id,username,age,address</sql>
<select id="findAll" resultType="com.hq.pojo.User">
    select <include refid="baseSelect"/>  from user
</select>
```

最终执行的sql为: **select id,username,age,address from user**

6.3、ResultMap

6.3.1 基本使用

我们可以**使用resultMap标签自定义结果集和实体类属性的映射规则。**

resultMap 用来自定义结果集和实体类的映射
属性:
    id 相当于这个resultMap的唯一标识
    type 用来指定映射到哪个实体类
id标签 用来指定主键列的映射规则

属性:
    property 要映射的属性名
    column 对应的列名
result标签 用来指定普通列的映射规则
属性:
    property 要映射的属性名
    column 对应的列名

```
<resultMap id="orderMap" type="com.hq.pojo.Order" >
    <id column="id" property="id"></id>
    <result column="createtime"
property="createtime"></result>
    <result column="price" property="price">
</result>
    <result column="remark" property="remark">
</result>
    <result column="user_id" property="userId">
</result>
</resultMap>

<!—使用我们自定义的映射规则—>
<select id="findAll" resultMap="orderMap">
    SELECT id,createtime,price,remark,user_id  FROM
ORDERS
</select>
```

接口方法

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

测试方法

```
@BeforeTest
public void init() throws IOException {
    //定义mybatis配置文件的路径
    String resource = "mybatis-config.xml";
```

```
    InputStream inputStream =
Resources.getResourceAsStream(resource);
    SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
    //获取Sqlsession对象
    sqlSession = sqlSessionFactory.openSession();
}

@Test
public void test_Mybatis() throws IOException {
    OrderDao orderDao =
sqlSession.getMapper(OrderDao.class);
    //调用方法测试
    List<Order> orderDaoAll = orderDao.findAll();
    System.out.println(orderDaoAll);
}

@AfterTest
public void destroy() throws IOException {
    //提交session
    sqlSession.commit();
    //释放资源
    sqlSession.close();
}
```

6.3.2 自动映射

我们定义resultMap时默认情况下**自动映射是开启状态**的。也就是**如果结果集的列名和我们的属性名相同是会自动映射的我们只需要写特殊情况的映射关系**即可。

例如：

下面这种写法和上面的写法会有相同的效果，因为其他属性的属性名和结果集的列名都是相同的会自动映射。

```
<resultMap id="orderMapAuto"
type="com.hq.pojo.Order" >
    <result column="user_id" property="userId">
</result>
</resultMap>
<!--使用我们自定义的映射规则-->
<select id="findAll" resultMap="orderMapAuto">
    SELECT id,createtime,price,remark,user_id  FROM
ORDERS
</select>
```

如有需要**可以选择关闭自动映射**可以**把resultMap的autoMapping属性设置为false。**

例如：

```
<resultMap id="orderMap"
type="com.sangeng.pojo.Order" autoMapping="false">
    <id column="id" property="id"></id>
    <result column="createtime"
property="createtime"></result>
    <result column="price" property="price">
</result>
    <result column="remark" property="remark">
</result>
    <result column="user_id" property="userId">
</result>
</resultMap>
```

6.3.3 继承映射关系

我们可以使用resultMap 的extends属性来指定一个resultMap，从而**复用重复的映射关系配置。通常在项目中关闭自动映射。**

例如：

```
<!--定义个父映射，供其他resultMap继承-->
```

```
<resultMap id="baseOrderMap"
type="com.hq.pojo.Order" autoMapping="false">
    <id column="id" property="id"></id>
    <result column="createtime"
property="createtime"></result>
    <result column="price" property="price">
</result>
    <result column="remark" property="remark">
</result>
</resultMap>
<!--继承baseOrderMap，然后只需要写自己特有的映射关系即可-->
<resultMap id="orderMapExtends"
type="com.hq.pojo.Order" autoMapping="false"
extends="baseOrderMap">
    <result column="user_id" property="userId">
</result>
</resultMap>

<select id="findAll" resultMap="orderMapExtends">
    SELECT id,createtime,price,remark,user_id  FROM
ORDERS
</select>
```

6.4. 多表查询

有的时候我们需要查询多张表的数据才可以得到我们要的结果。

我们可以直接写一个多表关联的SQL进行查询。也可以分步进行多次的查询来拿到我们需要的结果。

Mybatis就提供了对应的配置，可以让我们去更方便的进行相应的查询和对应的结果集处理。

6.4.1 多表关联查询

> 一对一关系

两个实体之间是一对一的关系。（例如我们需要查询订单，要求还需要下单用户的数据。这里的订单相对于用户是一对一。

例如：方法定义如下

```
//根据订单id查询订单，要求把下单用户的信息也查询出来
List<Order> findById(Integer id);
```

因为期望Order中还能包含下单用户的数据，所以可以再Order中增加一个属性

```
private User user;
```

SQL语句如下

```
SELECT

o.id,o.`createtime`,o.`price`,o.`remark`,o.`user_id`
,u.`id` uid,u.`username`,u.`age`,u.`address`
FROM
    orders o inner join USER u
WHERE
    o.`user_id` = u.`id`
    AND o.id = 2
```

6.4.1.1 使用ResultMap对所有字段进行映射（一对一）

可以使用ResultMap设置user对象的属性的映射规则。

①resultMap定义，**主要是对user对象的属性设置映射规则，** **user.属性**

```
<!--定义个父映射，供其他resultMap继承-->
<resultMap id="baseOrderMap"
type="com.hq.pojo.Order" autoMapping="false">
```

```
    <id column="id" property="id"></id>
    <result column="createtime"
property="createtime"></result>
    <result column="price" property="price">
</result>
    <result column="remark" property="remark">
</result>
</resultMap>
<!--继承baseOrderMap，然后只需要写自己特有的映射关系即可-->
<resultMap id="orderMapExtends"
type="com.hq.pojo.Order" autoMapping="false"
extends="baseOrderMap">
    <result column="user_id" property="userId">
</result>
</resultMap>

<!--Order和User关联的映射-->
<resultMap id="orderUserMap"
type="com.hq.pojo.Order" autoMapping="false"
extends="orderMapExtends">
    <result property="user.id" column="uid">
</result>
    <result property="user.username"
column="username"></result>
    <result property="user.age" column="age">
</result>
    <result property="user.address"
column="address"></result>
</resultMap>
```

②使用定义好的resultMap

```
<!--根据订单id查询订单，要求把下单用户的信息也查询出来-->
<select id="findById" resultMap="orderUserMap">
    SELECT

o.`id`,o.`createtime`,o.`price`,o.`remark`,o.`user_i
d`,u.`id` uid,u.`username`,u.`age`,u.`address`
    FROM
    orders o,`user` u
    WHERE
    o.id = #{id} AND
    o.`user_id`=u.`id`
</select>
```

6.4.1.2 使用ResultMap中的association（一对一）

可以使用ResultMap中的**子标签association,property="user"实体类中的实体属性** 来设置关联实体类的映射规则.

①定义resultMap

```
<association property="user"
javaType="com.hq.pojo.User">
```

```
<!--定义个父映射，供其他resultMap继承-->
<resultMap id="baseOrderMap"
type="com.hq.pojo.Order" autoMapping="false">
    <id column="id" property="id"></id>
    <result column="createtime"
property="createtime"></result>
    <result column="price" property="price">
</result>
    <result column="remark" property="remark">
</result>
</resultMap>
<!--继承baseOrderMap，然后只需要写自己特有的映射关系即可-->
```

```
<resultMap id="orderMapExtends"
type="com.hq.pojo.Order" autoMapping="false"
extends="baseOrderMap">
    <result column="user_id" property="userId">
</result>
</resultMap>

<!--Order和User关联的映射（使用association）-->
<resultMap id="orderUserMapUseAssociation"
type="com.hq.pojo.Order" autoMapping="false"
extends="orderMapExtends">
    <association property="user"
javaType="com.hq.pojo.User">
        <id property="id" column="uid"></id>
        <result property="username"
column="username"></result>
        <result property="age" column="age">
</result>
        <result property="address" column="address">
</result>
    </association>
</resultMap>
```

②使用resultMap

```
<!--根据订单id查询订单，要求把下单用户的信息也查询出来-->
<select id="findByIdAssociation"
resultMap="orderUserMapUseAssociation">
    SELECT

o.`id`,o.`createtime`,o.`price`,o.`remark`,o.`user_i
d`,u.`id` uid,u.`username`,u.`age`,u.`address`
    FROM
    orders o,`user` u
    WHERE
    o.id = #{id} AND
    o.`user_id`=u.`id`
</select>
```

测试方法

```
OrderDao orderDao =
sqlSession.getMapper(OrderDao.class);
//调用方法测试
List<Order> orderDaoAll =
orderDao.findByIdAssociation(2);
```

6.4.1.3 使用ResultMap中的collection（一对多和多对多）

> 一对多关系

两个实体之间是一对多的关系。（例如我们需要查询用户，要求还需要该用户所具有的角色信息。这里的用户相对于角色是一对多的。）

例如：方法定义如下

```
//根据id查询用户，并且要求把该用户所具有的角色信息也查询出来
User findByIdCollection(Integer id);
```

因为期望User中还能包含该用户所具有的角色信息，所以可以在User中增加一个属性

```
// 该用户所具有的角色
private List<Role> roles;
```

期望SQL语句如下

```
SELECT
    u.`id`,u.`username`,u.`age`,u.`address`,r.id
rid,r.name,r.desc
FROM
    USER u,user_role ur,role r
WHERE
    u.id=ur.user_id AND ur.role_id = r.id
    AND u.id = 2
```

结果集

id	username	age	address	rid	name	desc
2	pdd	25	上海	2	CFO	(NULL)
2	pdd	25	上海	1	总经理	一人之下

我们可以使用如下的方式封装结果集。

可以使用ResultMap中的**子标签collection来设置关联实体类的映射规则.**

①定义ResultMap

```
<collection property="roles" ofType="com.hq.pojo.Role"
>
```

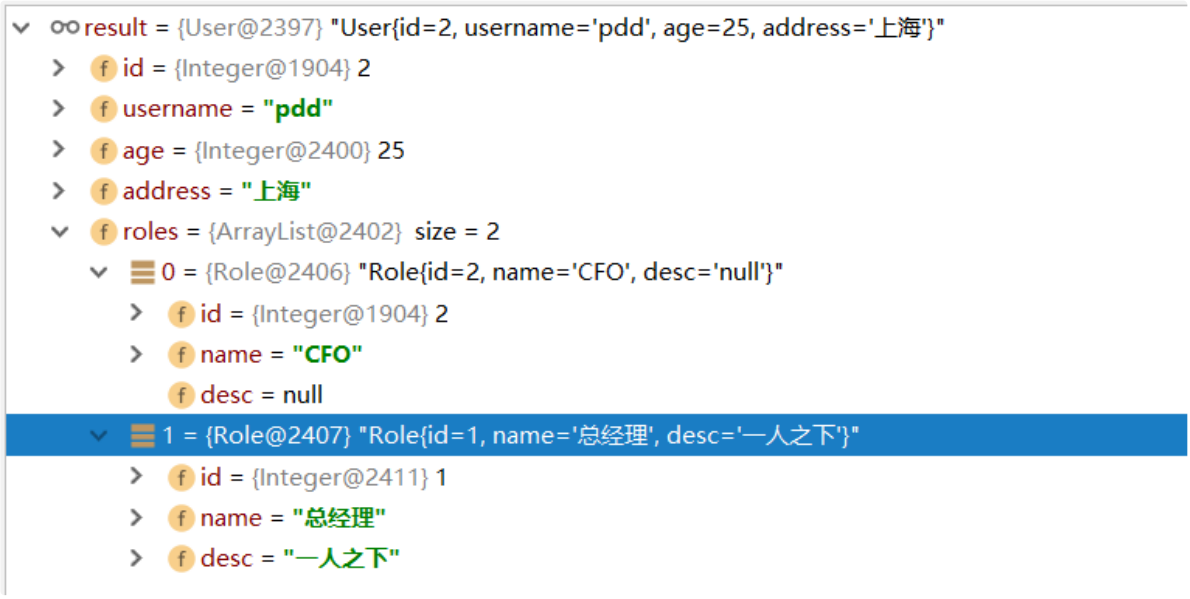
```
<resultMap id="userMap" type="com.hq.pojo.User"
autoMapping="false">
    <id property="id" column="id"></id>
    <result property="username" column="username">
</result>
    <result property="age" column="age"></result>
    <result property="address" column="address">
</result>
</resultMap>
```

```
<resultMap id="userRoleMap" type="com.hq.pojo.User"
 autoMapping="false" extends="userMap">
    <collection property="roles"
ofType="com.hq.pojo.Role" >
        <id property="id" column="rid"></id>
        <result property="name" column="name">
</result>
        <result property="desc" column="desc">
</result>
    </collection>
</resultMap>
```

②使用ResultMap

```
<select id="findById" resultMap="userRoleMap" >
    SELECT
    u.`id`,u.`username`,u.`age`,u.`address`,r.id
rid,r.name,r.desc
    FROM
    USER u,user_role ur,role r
    WHERE
    u.id=ur.user_id AND ur.role_id = r.id
    AND u.id = #{id}
</select>
```

最终封装完的结果如下：



测试方法

```
UserDao userDao =
sqlSession.getMapper(UserDao.class);
//调用方法测试
User user = userDao.findByIdCollection(2);
```

6.4.2 分步查询

如果有需要多表查询的需求我们也可以选择用多次查询的方式来查询出我们想要的数据。Mybatis也提供了对应的配置。**例如我们需要查询用户，要求还需要查询出该用户所具有的角色信息。我们可以选择先查询User表查询用户信息。然后在去查询关联的角色信息。**

6.4.2.1 实现步骤

具体步骤如下：

> 定义相应分步骤的查询方法

因为我们要分两步查询： 1.查询User 2.根据用户的id查询Role 所以我们需要定义下面两个方法，并且把对应的标签也先写好

1.查询User

```
//根据用户名查询用户，并且要求把该用户所具有的角色信息也查询出来
User findByUsername(String username);
```

```
<!--根据用户名查询用户-->
<select id="findByUsername"
resultType="com.hq.pojo.User">
    select id,username,age,address from user where
username = #{username}
</select>
```

2.根据user_id查询Role

```
public interface RoleDao {
    //根据userId查询所具有的角色
    List<Role> findRoleByUserId(Integer userId);
}
```

```
<!--根据userId查询所具有的角色-->
<select id="findRoleByUserId"
resultType="com.hq.pojo.Role">
    select
    r.id,r.name,r.desc
    from
    role r,user_role ur
    where
    ur.role_id = r.id
    and ur.user_id = #{userId}
</select>
```

> 配置分步查询

我们期望的效果是调用findByUsername方法查询出来的结果中就包含角色的信息。所以我们需要修改userDao.xml中的findByUsername方法的RestltMap，指定分步查询。

select属性：指定用哪个查询来查询当前属性的数据 写法：包名.接口名.方法名
column属性：设置当前结果集中哪列的数据作为select属性指定的查询方法需要参数

```
<resultMap id="userMap" type="com.hq.pojo.User">
    <id property="id" column="id"></id>
    <result property="username" column="username">
</result>
    <result property="age" column="age"></result>
```

```
    <result property="address" column="address">
</result>
</resultMap>
<!--
            select属性：指定用哪个查询来查询当前属性的数据
写法：包名.接口名.方法名
            column属性：设置当前结果集中哪列的数据作为
select属性指定的查询方法需要参数
        -->
<resultMap id="userRoleMapBySelect"
type="com.hq.pojo.User" extends="userMap">
    <collection property="roles"
                ofType="com.hq.pojo.Role"

 select="com.hq.dao.RoleDao.findRoleByUserId"
                column="id">
    </collection>
</resultMap>
```

改变findByUsername使用我们刚刚创建的resultMap

```
<!--根据用户名查询用户-->
<select id="findByUsername"
resultMap="userRoleMapBySelect">
    select id,username,age,address from user where
username = #{username}
</select>
```

> 假设有两个参数`userId`和`roleId`，你可以考虑以下方式：

```
<resultMap id="userRoleMapBySelect"
type="com.hq.pojo.User">
    <collection property="roles"
                ofType="com.hq.pojo.Role"

 select="com.hq.dao.RoleDao.findRoleByUserIdAndRoleI
d"
                column="{userId=id,
roleId=anotherColumn}">
    </collection>
</resultMap>
```

然后，在RoleDao接口中，你的方法可能看起来像这样：

```
List<Role>
findRoleByUserIdAndRoleId(@Param("userId") Long
userId, @Param("roleId") Long roleId);
```

6.4.2.2 设置按需加载

我们可以设置按需加载，这样在我们代码中需要用到关联数据的时候才会去查询关联数据。（fetchType属性为lazy，实现延迟加载）

有两种方式可以配置分别是**全局配置和局部配置**

1. 局部配置

   设置fetchType属性为lazy

   ```
   <resultMap id="userRoleMapBySelect"
   type="com.hq.pojo.User" extends="userMap">
       <collection property="roles"
                   ofType="com.hq.pojo.Role"

    select="com.hq.dao.RoleDao.findRoleByUserId"
                   column="id" fetchType="lazy">
       </collection>
   </resultMap>
   ```

2. 全局配置

设置lazyLoadingEnabled为true

```
<settings>
    <setting name="lazyLoadingEnabled"
value="true"/>
</settings>
```

7.分页查询-PageHelper

我们可以使用PageHelper非常方便的帮我们实现分页查询的需求。不需要自己在SQL中拼接SQL相关参数，并且能非常方便的获取的总页数总条数等分页相关数据。

7.1 实现步骤

> 定义方法查询方法以及生成对应标签

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

```
<select id="findAll" resultType="com.hq.pojo.User">
    select id,username,age,address from user
</select>
```

> 引入依赖

```
<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper</artifactId>
    <version>4.0.0</version>
</dependency>
```

> 配置Mybatis核心配置文件mybatis-config使用分页插件,放在environments之前

```
<plugins>
    <!-- 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>
</plugins>
```

> 开始分页查询

我们只需要在使用查询方法前设置分页参数即可。**pageNum（从1开始，分页的页数） pageSize（每一页的数据量大小）**

```
public void test_Mybatis() throws IOException {
    //设置分页参数
    UserDao userDao =
sqlSession.getMapper(UserDao.class);
    //设置分页查询参数
    PageHelper.startPage(1,2); // pageNum pageSize
    List<User> users = userDao.findAll(); // 查出来的
当前页的数据集合
    System.out.println(users.get(0));
}
```

如果需要获取总页数总条数等分页相关数据，只需要创建一个PageInfo对象，把刚刚查询出的返回值做为构造方法参数传入。然后使用pageInfo对象获取即可。

```
PageInfo<User> pageInfo = new PageInfo<User>(users);
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());

总条数: 3
总页数: 2
当前页: 1
每页显示长度: 2
```

7.2 一对多多表查询分页问题

我们在进行一对多的多表查询时，如果使用了PageHelper进行分页。**会出现关联数据不全的情况，原因是关联的属性（例如角色）还没有变成集合就被limit了。**我们可以使用分步查询的方式解决该问题（**先查用户，再查用户关联的角色**）。