MyBatis

Not Given

2022 年 7 月 16 日

目录

第一章：MyBatis 简介

一、原始 JDBC 操作

1. 查询代码

```
//注册驱动
Class.forName("com.mysql.jdbc.Driver");
//获得连接
Connection connection = DriverManager.getConnection( url: "jdbc:mysql:///test",  user: "root",  password: "root");
//获得statement
PreparedStatement statement = connection.prepareStatement( sql: "select id,username,password from user");
//执行查询
ResultSet resultSet = statement.executeQuery();
//遍历结果集
while(resultSet.next()){
    //封装实体
    User user = new User();
    user.setId(resultSet.getInt( columnLabel: "id"));
    user.setUsername(resultSet.getString( columnLabel: "username"));
    user.setPassword(resultSet.getString( columnLabel: "password"));
    //user实体封装完毕
    System.out.println(user);
}
//释放资源
resultSet.close();
statement.close();
connection.close();
```

2. 插入代码

```
//模拟实体对象
User user = new User();
user.setId(2);
user.setUsername("tom");
user.setPassword("lucy");

//注册驱动
Class.forName("com.mysql.jdbc.Driver");
//获得连接
Connection connection = DriverManager.getConnection( url: "jdbc:mysql:///test",  user: "root",  password: "root");
//获得statement
PreparedStatement statement = connection.prepareStatement( sql: "insert into user(id,username,password) values(?,?,?)");
//设置占位符参数
statement.setInt( parameterIndex: 1,user.getId());
statement.setString( parameterIndex: 2,user.getUsername());
statement.setString( parameterIndex: 3,user.getPassword());
//执行更新操作
statement.executeUpdate();
//释放资源
statement.close();
connection.close();
```

3. 旧方法存在的问题：重复代码过多；频繁开启和关闭，过于消耗资源；sql 语句和 java 代码耦合度高

二、MyBatis 的概述

1. 解决现有 JDBC 问题的方案

应对上述问题给出的解决方案:

① 使用数据库连接池初始化连接资源

② 将sql语句抽取到xml配置文件中

③ 使用反射、内省等底层技术，自动将实体与表进行属性与字段的自动映射

2. MyBatis 概述

- mybatis是一个优秀的基于java的持久层框架，它内部封装了jdbc，使开发者只需要关注sql语句本身，而不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。
- mybatis通过xml或注解的方式将要执行的各种statement配置起来，并通过java对象和statement中sql的动态参数进行映射生成最终执行的sql语句。
- 最后mybatis框架执行sql并将结果映射为java对象并返回。采用ORM思想解决了实体和数据库映射的问题，对jdbc 进行了封装，屏蔽了jdbc api 底层访问细节，使我们不用与jdbc api 打交道，就可以完成对数据库的持久化操作。

第二章：MyBatis 的基本使用方式

一、基本开发步骤

1. 添加 MyBatis 坐标

```xml
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.32</version>
    </dependency>
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.4.6</version>
    </dependency>
```

2. 创建 User 表

id	username	password
1	zhangsan	123
2	lisi	123
3	wangwu	123
4	zhaoliu	123
5	tianqi	123
(Auto)	(NULL)	(NULL)

3. 创建 User 实体类

```java
public class User {

    private int id;
    private String username;
    private String password;

    public int getId() {
        return id;
    }

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

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
```

4. 编写映射文件 UserMapper.xml

```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.d
<mapper namespace="userMapper">

    <select id="findAll" resultType="com.itheima.domain.User">
        select * from user
    </select>

</mapper>
```

5. 编写核心文件 SqlMapConfig.xml

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

    <!--数据源环境-->
    <environments default="developement">
        <environment id="developement">
            <transactionManager type="JDBC"></transactionManager>
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://localhost:3306/test"/>
                <property name="username" value="root"/>
                <property name="password" value="root"/>
            </dataSource>
        </environment>
    </environments>

</configuration>
```

6. 测试

```java
//加载核心配置文件
InputStream resourceAsStream = Resources.getResourceAsStream("SqlMapConfig.xml");
//获得sqlSession 工厂对象
SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
//获得sqlSession对象
SqlSession sqlSession = sqlSessionFactory.openSession();
//执行sql语句
List<User> userList = sqlSession.selectList("userMapper.findAll");
//打印结果
System.out.println(userList);
//释放资源
sqlSession.close();
```

二、映射文件

1. 概述

2. 插入操作：要注意标注属性值。还需要注意的是 MyBatis 默认事务不提交，所以要手动编写代码提交

```xml
<!--插入操作-->
<insert id="save" parameterType="com.itheima.domain.User">
    insert into user values(#{id},#{username},#{password})
</insert>
```

基本代码

- 插入语句使用insert标签
- 在映射文件中使用parameterType属性指定要插入的数据类型
- Sql语句中使用#{实体属性名}方式引用实体中的属性值
- 插入操作使用的API是sqlSession.insert(“命名空间.id”,实体对象);
- 插入操作涉及数据库数据变化，所以要使用sqlSession对象显示的提交事务，即sqlSession.commit()

注意事项

3. 修改操作

```xml
<!--修改操作-->
<update id="update" parameterType="com.itheima.domain.User">
    update user set username=#{username},password=#{password} where id=#{id}
</update>
```

基本代码

- 修改语句使用update标签
- 修改操作使用的API是sqlSession.update(“命名空间.id”,实体对象);

注意事项

4. 删除操作

```xml
<!--删除操作-->
<delete id="delete" parameterType="java.lang.Integer">
    delete from user where id=#{id}
</delete>
```

基本代码

- 删除语句使用delete标签
- Sql语句中使用#{任意字符串}方式引用传递的单个参数
- 删除操作使用的API是sqlSession.delete(“命名空间.id”,Object);

注意事项

三、核心配置文件

1. 基本层级关系

- configuration 配置
  - properties 属性
  - settings 设置
  - typeAliases 类型别名
  - typeHandlers 类型处理器
  - objectFactory 对象工厂
  - plugins 插件
  - environments 环境
    - environment 环境变量
      - transactionManager 事务管理器
      - dataSource 数据源
  - databaseIdProvider 数据库厂商标识
  - mappers 映射器

2. 环境标签

数据库环境的配置，支持多环境配置

```xml
<environments default="development">
    <environment id="development">
        <transactionManager type="JDBC"/>
        <dataSource type="POOLED">
            <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>
```


基本结构

其中，事务管理器（transactionManager）类型有两种：

- JDBC：这个配置就是直接使用了JDBC的提交和回滚设置，它依赖于从数据源得到的连接来管理事务作用域。
- MANAGED：这个配置几乎没做什么。它从来不提交或回滚一个连接，而是让容器来管理事务的整个生命周期（比如 JEE 应用服务器的上下文）。默认情况下它会关闭连接，然而一些容器并不希望这样，因此需要将 closeConnection 属性设置为 false 来阻止它默认的关闭行为。

其中，数据源（dataSource）类型有三种：

- UNPOOLED：这个数据源的实现只是每次被请求时打开和关闭连接。
- POOLED：这种数据源的实现利用“池”的概念将 JDBC 连接对象组织起来。
- JNDI：这个数据源的实现是为了能在如 EJB 或应用服务器这类容器中使用，容器可以集中或在外部配置数据源，然后放置一个 JNDI 上下文的引用。

配置详解

3. 映射标签：主要配置加载映射文件的方式

该标签的作用是加载映射的，加载方式有如下几种：

- 使用相对于类路径的资源引用，例如：<mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
- 使用完全限定资源定位符（URL），例如：<mapper url="file:///var/mappers/AuthorMapper.xml"/>
- 使用映射器接口实现类的完全限定类名，例如：<mapper class="org.mybatis.builder.AuthorMapper"/>
- 将包内的映射器接口实现全部注册为映射器，例如：<package name="org.mybatis.builder"/>

4. 属性标签：有时会将数据源的配置信息分离到 properties 文件中，再在数据库配置文件中引入

基本功能

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

jdbc.properties

```
<!--数据源环境-->
<environments default="developement">
    <environment id="developement">
        <transactionManager type="JDBC"></transactionManager>
        <dataSource type="POOLED">
            <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>
```

sqlMapConfig.xml

5. 别名标签：可以用一个短名称来代替全类名

四、MyBatis 的 API

1. SqlSessionFactoryBuilder：SqlSession 工厂构建器

常用API：SqlSessionFactory build(InputStream inputStream)

通过加载mybatis的核心文件的输入流的形式构建一个SqlSessionFactory对象

```
String resource = "org/mybatis/builder/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(inputStream);
```

其中，Resources 工具类，这个类在 org.apache.ibatis.io 包中。Resources 类帮助你从类路径下、文件系统或一个 web URL 中加载资源文件。

2. SqlSessionFactory：创建 Session

方法	解释
openSession()	会默认开启一个事务，但事务不会自动提交，也就意味着需要手动提交该事务，更新操作数据才会持久化到数据库中
openSession(boolean autoCommit)	参数为是否自动提交，如果设置为true，那么不需要手动提交事务

3. SqlSession：可以进行执行语句、提交或回滚事务等操作

SqlSession 实例在 MyBatis 中是非常强大的一个类。在这里你会看到所有执行语句、提交或回滚事务和获取映射器实例的方法。

执行语句的方法主要有：

```
<T> T selectOne(String statement, Object parameter)
<E> List<E> selectList(String statement, Object parameter)
int insert(String statement, Object parameter)
int update(String statement, Object parameter)
int delete(String statement, Object parameter)
```

操作事务的方法主要有：

```
void commit()
void rollback()
```

第三章：MyBatis 的 DAO 层实现

一、传统开发方式

1. 定义接口

```
public class UserMapperImpl implements UserMapper {


}
```

2. 实现接口

```
public class UserMapperImpl implements UserMapper {

    public List<User> findAll() throws IOException {
        InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapconfig.xml");
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
        SqlSession sqlSession = sqlSessionFactory.openSession();
        List<User> userList = sqlSession.selectList( s: "userMapper.findAll");
        return userList;
    }
}
```

3. 在业务层调用方法

```
public class ServiceDemo {

    public static void main(String[] args) throws IOException {

        //创建dao层对象
        UserMapper userMapper = new UserMapperImpl();
        List<User> all = userMapper.findAll();

        System.out.println(all);

    }

}
```

二、接口代理方式

1. 基本规范

采用 Mybatis 的代理开发方式实现 DAO 层的开发，这种方式是我们后面进入企业的主流。

Mapper 接口开发方法只需要程序员编写 Mapper 接口（相当于 Dao 接口），由 Mybatis 框架根据接口定义创建接口的动态代理对象，代理对象的方法体同上边 Dao 接口实现类方法。

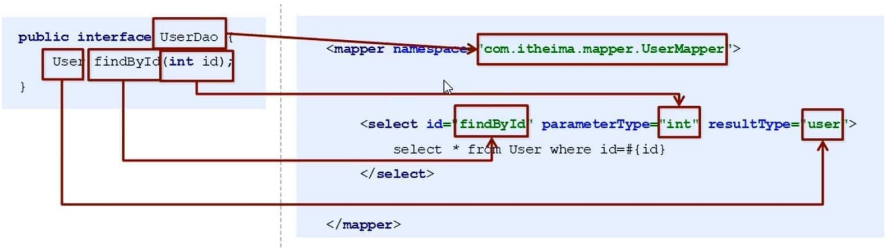
Mapper 接口开发需要遵循以下规范:

1、 Mapper.xml 文件中的 namespace 与 mapper 接口的全限定名相同

2、 Mapper 接口方法名和 Mapper.xml 中定义的每个 statement 的 id 相同

3、 Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的类型相同

4、 Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultType 的类型相同



2. 步骤

(1) 实现规范，保证各名称一致

(2) 使用 getMapper 方法创建动态代理对象

```
public class ServiceDemo {

    public static void main(String[] args) throws IOException {

        InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
        SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
        SqlSession sqlSession = sqlSessionFactory.openSession();

        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        List<User> all = mapper.findAll();
        System.out.println(all);


    }
```

第四章：映射文件深入

一、概述

1. 主要内容：动态 sql 的使用和 sql 抽取
2. 功能：根据传递参数的不同，使用不同的 sql 语句；sql 语句的复用

二、动态 sql

(一)if

1. 功能：进行逻辑判断
2. 使用方式

我们根据实体类的不同取值，使用不同的SQL语句来进行查询。比如在 id如果不为空时可以根据id查询，如果username 不同空时还要加入用户名作为条件。这种情况在我们的多条件组合查询中经常会碰到。

```
<select id="findByCondition" parameterType="user" resultType="user">
  select * from User
  <where>
    <if test="id!=0">
      and id=#{id}
    </if>
    <if test="username!=null">
      and username=#{username}
    </if>
  </where>
</select>
```

(二)foreach

1. 功能：进行 sql 语句的循环拼接
2. 使用方式

循环执行sql的拼接操作，例如：SELECT * FROM USER WHERE id IN (1,2,5)。

```
<select id="findByIds" parameterType="list" resultType="user">
    select * from User
    <where>
        <foreach collection="array" open="id in(" close=")" item="id" separator=",">
            #{id}
        </foreach>
    </where>
</select>
```

三、sql 语句的抽取

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

```
<!--抽取sql片段简化编写-->
<sql id="selectUser" select * from User</sql>
<select id="findById" parameterType="int" resultType="user">
  <include refid="selectUser"></include> where id=#{id}
</select>
<select id="findByIds" parameterType="list" resultType="user">
  <include refid="selectUser"></include>
  <where>
    <foreach collection="array" open="id in(" close=")" item="id" separator=",">
      #{id}
    </foreach>
  </where>
</select>
```

第五章：核心配置文件深入

一、TypeHandler

(一) 简介

1. 功能：进行类型处理器定义

2. 基本转换

无论是 MyBatis 在预处理语句（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时，都会用类型处理器将获取的值以合适的方式转换成 Java 类型。下表描述了一些默认的类型处理器（截取部分）。

类型处理器	Java 类型	JDBC 类型
BooleanTypeHandler	java.lang.Boolean, boolean	数据库兼容的 BOOLEAN
ByteTypeHandler	java.lang.Byte, byte	数据库兼容的 NUMERIC 或 BYTE
ShortTypeHandler	java.lang.Short, short	数据库兼容的 NUMERIC 或 SHORT INTEGER
IntegerTypeHandler	java.lang.Integer, int	数据库兼容的 NUMERIC 或 INTEGER
LongTypeHandler	java.lang.Long, long	数据库兼容的 NUMERIC 或 LONG INTEGER

3. 自定义转换

你可以重写类型处理器或创建你自己的类型处理器来处理不支持的或非标准的类型。具体做法为：实现 org.apache.ibatis.type.TypeHandler 接口，或继承一个很便利的类 org.apache.ibatis.type.BaseTypeHandler，然后可以选择性地将它映射到一个JDBC类型。例如需求：一个Java中的Date数据类型，我想将之存到数据库的时候存成一个1970年至今的毫秒数，取出来时转换成java的Date，即java的Date与数据库的varchar毫秒值之间转换。

开发步骤：

① 定义转换类继承类BaseTypeHandler<T>

② 覆盖4个未实现的方法，其中setNonNullParameter为java程序设置数据到数据库的回调方法，getNullableResult为查询时 mysql的字符串类型转换成java的Type类型的方法

③ 在MyBatis核心配置文件中进行注册

④ 测试转换是否正确

(二) 自定义转换的具体步骤

1. 定义转换类，集成类 BaseTypeHandler<T>，并覆盖方法

```java
public class DateTypeHandler extends BaseTypeHandler<Date> {
    //将java类型 转换成 数据库需要的类型
    public void setNonNullParameter(PreparedStatement preparedStatement, int i, Date date, JdbcType jdbcType) t
        long time = date.getTime();
        preparedStatement.setLong(i,time);
    }

    //将数据库中类型 转换成java类型
    public Date getNullableResult(ResultSet resultSet, String s) throws SQLException {
        return null;
    }

    //将数据库中类型 转换成java类型
    public Date getNullableResult(ResultSet resultSet, int i) throws SQLException {
        return null;
    }
}
```

2. 在 MyBatis 核心配置文件中进行注册

```xml
<!--注册类型处理器-->
<typeHandlers>
    <typeHandler handler="com.itheima.handler.DateTypeHandler"></typeHandler>
</typeHandlers>
```

3. 测试

二、Plugin

主要用来配置分页助手，基本方式如下：

1. 导入 PageHelper 的坐标

① 导入通用PageHelper坐标

```xml
<!-- 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. 配置插件

② 在mybatis核心配置文件中配置PageHelper插件

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

3. 设置分页相关参数

```java
@Test
public void test3() throws IOException {
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = sqlSessionFactory.openSession();
    UserMapper mapper = sqlSession.getMapper(UserMapper.class);

    //设置分页相关参数    当前页+每页显示的条数
    PageHelper.startPage( pageNum: 1, pageSize: 3);

    List<User> userList = mapper.findAll();
    for (User user : userList) {
        System.out.println(user);
    }

    sqlSession.close();
}
```

4. 获得分页相关参数

```java
//获得与分页相关参数
PageInfo<User> pageInfo = new PageInfo<User>(userList);
System.out.println("当前页："+pageInfo.getPageNum());
System.out.println("每页显示条数："+pageInfo.getPageSize());
System.out.println("总条数："+pageInfo.getTotal());
System.out.println("总页数："+pageInfo.getPages());
System.out.println("上一页："+pageInfo.getPrePage());
System.out.println("下一页："+pageInfo.getNextPage());
System.out.println("是否是第一个："+pageInfo.isIsFirstPage());
System.out.println("是否是最后一个："+pageInfo.isIsLastPage());
```

第六章：多表操作

一、多表关系概述

1. 一对一：共用同一个主键，或者一方有外键
2. 一对多：多的一方有一个外键与一的一方的主键相关联
3. 多对多：利用中间表进行管理，中间表中维护两个外键

二、一对一配置实现

1. 定义封装类

```
public class Order {

    private int id;
    private Date ordertime;
    private double total;

    //当前订单属于哪一个用户
    private User user;

    public int getId() {
        return id;
    }

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

2. 指定查询字段与实体属性的映射关系。可以直接使用 Map 映射，也可以添加匹配

```
<resultMap id="orderMap" type="order">
    <!--手动指定字段与实体属性的映射关系
        column: 数据表的字段名称
        property: 实体的属性名称
    -->
    <id column="oid" property="id"></id>
    <result column="ordertime" property="ordertime"></result>
    <result column="total" property="total"></result>
    <result column="uid" property="user.id"></result>
    <result column="username" property="user.username"></result>
    <result column="password" property="user.password"></result>
    <result column="birthday" property="user.birthday"></result>
</resultMap>
```

```
<!--
    property: 当前实体(order)中的属性名称(private User user)
    javaType: 当前实体(order)中的属性的类型(User)
-->
<association property="user" javaType="user">
    <id column="uid" property="id"></id>
    <result column="username" property="username"></result>
    <result column="password" property="password"></result>
    <result column="birthday" property="birthday"></result>
</association>
```

3. 编写查询 sql 语句

```
<select id="findAll" resultMap="orderMap">
    SELECT *,o.id oid FROM orders o,USER u WHERE o.uid=u.id
</select>
```

三、一对多配置实现

1. 定义封装类

```
public class User {

    private int id;
    private String username;
    private String password;
    private Date birthday;

    public List<Order> getOrderList() {
        return orderList;
    }

    public void setOrderList(List<Order> orderList) {
        this.orderList = orderList;
    }

    //描述的是当前用户存在哪些订单
    private List<Order> orderList;
```

2. 指定映射关系

```
<resultMap id="userMap" type="user">
    <id column="uid" property="id"></id>
    <result column="username" property="username"></result>
    <result column="password" property="password"></result>
    <result column="birthday" property="birthday"></result>
    <!--配置集合信息
        property:集合名称
        ofType：当前集合中的数据类型
    -->
    <collection property="orderList" ofType="order">
        <!--封装order的数据-->
        <id column="oid" property="id"></id>
        <result column="ordertime" property="ordertime"></result>
        <result column="total" property="total"></result>
    </collection>
</resultMap>
```

3. 编写查询 sql 语句

```
<select id="findAll" resultMap="userMap">
    SELECT *,o.id oid FROM USER u,orders o WHERE u.id=o.uid
</select>
```

四、多对多配置实现

示例的需求为查询用户的同时，查询出该用户所有的角色，基本步骤如下：

1. 定义封装类

```
public class Role {

    private int id;
    private String roleName;
    private String roleDesc;

    public int getId() {
        return id;
    }

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

    public String getRoleName() {
        return roleName;
    }

    public void setRoleName(String roleName) {
```

Role.java

```
public class User {

    private int id;
    private String username;
    private String password;
    private Date birthday;

    //描述的是当前用户存在哪些订单
    private List<Order> orderList;

    //描述的是当前用户具备哪些角色
    private List<Role> roleList;


    public List<Order> getOrderList() { return orderList; }
```

User.java

2. 指定映射关系

```
<resultMap id="userRoleMap" type="user">
    <!--user的信息-->
    <id column="userId" property="id"></id>
    <result column="username" property="username"></result>
    <result column="password" property="password"></result>
    <result column="birthday" property="birthday"></result>
    <!--user内部的roleList信息-->
    <collection property="roleList" ofType="role">
        <id column="roleId" property="id"></id>
        <result column="" property="roleName"></result>
        <result column="" property="roleDesc"></result>
    </collection>
</resultMap>
```

3. 编写查询 sql 语句

```
<select id="findUserAndRoleAll" resultMap="userRoleMap">
    SELECT * FROM USER u,sys_user_role ur,sys_role r WHERE u.id=ur.userId AND ur.roleId=r.id
</select>
```

第七章：注解开发

一、常用注解介绍

1. 注解的作用：简化开发

2. 常用注解

@Insert：实现新增
@Update：实现更新
@Delete：实现删除
@Select：实现查询
@Result：实现结果集封装
@Results：可以与@Result一起使用，封装多个结果集
@One：实现一对一结果集封装
@Many：实现一对多结果集封装

二、注解配置增删改查的操作

1. 配置映射关系

```java
public interface UserMapper {

    @Insert("insert into user values(#{id},#{username},#{password},#{birthday})")
    public void save(User user);

    @Update("update user set username=#{username},password=#{password} where id=#{id}")
    public void update(User user);

    @Delete("delete from user where id=#{id}")
    public void delete(int id);

    @Select("select * from user where id=#{id}")
    public User findById(int id);

    @Select("select * from user")
    public List<User> findAll();
```

2. 加载映射关系。类似于组件扫描，扫描存在注解的包

```xml
<!--加载映射关系-->
<mappers>
    <!--指定接口所在的包-->
    <package name="com.itheima.mapper"></package>
</mappers>
```

三、注解配置多表查询

(一) 基本注解介绍

实现复杂关系映射之前我们可以在映射文件中通过配置<resultMap>来实现，使用注解开发后，我们可以使用@Results注解，@Result注解，@One注解，@Many注解组合完成复杂关系的配置

注解	说明
@Results	代替的是标签<resultMap>该注解中可以使用单个@Result注解，也可以使用@Result集合。使用格式：@Results（{@Result（），@Result（）}）或@Results（@Result（））
@Resut	代替了<id>标签和<result>标签 @Result中属性介绍： column：数据库的列名 property：需要装配的属性名 one：需要使用的@One 注解（@Result（one=@One）（）） many：需要使用的@Many 注解（@Result（many=@many）（））

(二) 一对一的注解开发

1. 创建查询接口

2. 添加注解，配置查询语句，以及数据库参数与 java 类属性的映射关系

```java
public interface OrderMapper {

    @Select("select *,o.id oid from orders o,user u where o.uid=u.id")
    @Results({
            @Result(column = "oid",property = "id"),
            @Result(column = "ordertime",property = "ordertime"),
            @Result(column = "total",property = "total"),
            @Result(column = "uid",property = "user.id"),
            @Result(column = "username",property = "user.username"),
            @Result(column = "password",property = "user.password")
    })
    public List<Order> findAll();

}
```

属性对应

```java
public interface OrderMapper {

    @Select("select * from orders")
    @Results({
            @Result(column = "id",property = "id"),
            @Result(column = "ordertime",property = "ordertime"),
            @Result(column = "total",property = "total"),
            @Result(
                    property = "user", //要封装的属性名称
                    column = "uid", //根据那个字段去查询user表的数据
                    javaType = User.class, //要封装的实体类型
                    //select属性 代表查询那个接口的方法获得数据
                    one = @One(select = "com.itheima.mapper.UserMapper.findById")
            )
    })
    public List<Order> findAll();
```

类 Association 方式

3. 获取接口代理对象，进行查询测试

```java
    InputStream resourceAsStream = Resources.getResourceAsStream("sqlMapConfig.xml");
    SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(resourceAsStream);
    SqlSession sqlSession = sqlSessionFactory.openSession( b: true);
    mapper = sqlSession.getMapper(OrderMapper.class);
}


@Test
public void testSave(){
    List<Order> all = mapper.findAll();
    for (Order order : all) {
        System.out.println(order);
    }
}
```

(三) 一对多的注解开发

1. 创建查询接口

2. 添加注解，配置查询语句和映射关系。这时需要调用其他类的查询方法

```java
    @Select("select * from user")
    @Results({
            @Result(id=true ,column = "id",property = "id"),
            @Result(column = "username",property = "username"),
            @Result(column = "pasword",property = "password"),
            @Result(
                    property = "orderList",
                    column = "id",
                    javaType = List.class,
                    many = @Many(select = "com.itheima.mapper.OrderMapper.findByUid")
            )
    })
    public List<User> findUserAndOrderAll();

}
```

UserMapper

```java
@Select("select * from orders where uid=#{uid}")
public List<Order> findByUid(int uid);
```

OrderMapper

3. 查询测试。具体方式可以查看接口代理部分

(四) 多对多的注解开发

1. 创建查询接口

2. 添加注解，配置查询语句和映射关系。这时需要调用其他类的查询方法

```
@Select("SELECT * FROM USER")
@Results({
        @Result(id = true, column = "id", property = "id"),
        @Result(id = true, column = "username", property = "username"),
        @Result(id = true, column = "password", property = "password"),
        @Result(
                property = "roleList",
                column = "id",
                javaType = List.class,
                many = @Many(select = "com.itheima.mapper.RoleMapper.findByUid")
        )
})
public List<User> findUserAndRoleAll();
```

UserMapper

```
public interface RoleMapper {

    @Select("SELECT * FROM sys_user_role ur,sys_role r WHERE ur.roleId=r.id AND ur.userId=#{uid}")
    public List<Role> findByUid(int uid);
}
```

OrderMapper

3. 查询测试。具体方式可以查看接口代理部分