# 介绍

挑战一个晚上学完Mybatis

## MyBatis历史

MyBatis最初是Apachel的一个开源项目iBatis,201O年6月这个项目由Apache Software Foundation迁移到了

Google Code。随着开发团队转投Google Codei旗下，iBatis3.X正式更名为MyBatis。代码于2013年11月迁移到Github。

iBatis-一词来源于"internet"和'abatis'"的组合，是一个基于ava的持久层框架。iBatis提供的持久层框架包括**SQL Maps（封装JDBC）**和Data Access Objects(DAO，数据访问对象)。

## MyBatis特性

1)**MyBatis是**支持定制化SQL、存储过程以及高级映射的优秀的***持久层框架***

2)MyBatis避免了几乎所有的DBC代码和手动设置参数以及获取结果集

3)MyBatis**可以使用简单的XML或注解用于配置和原始映射，将接口和Java的POjO(Plain Old Java Objects,普通的Java对象)映射成数据库中的记录，也就是mybatis操作数据库的过程。**

4)MyBatis是一个半自动的ORM(Object Relation Mapping，对象关系映射。)框架

‍

ORM ：将Java实体类与数据库字段进行映射，这样我们操作Java对象的时候就算是再操作数据库字段。

‍

映射：再数据库字段与Java变量名一致时就可以映射成java的一个实体类对象。

高级映射：通过一对多，多对多查询时，本地类也要与表一样才能查询成功，通过高级映射实现，实体类之间的关系。

**MyBatis就是封装了JDBC**

‍

## 和其它持久化层技术对北比

1. JDBC

* SQL夹杂在ava代码中耦合度高，导致硬编码内伤(代码写死，后续只能改源码)
* 维护不易且实际开发需求中SQL有变化，频繁修改的情况多见
* 代码冗长，开发效率低

1. Hibernate和JPA

* 操作简便，开发效率高
* 程序中的长难复杂SQL需要绕过框架
* 内部自动生产的SQL,不容易故特殊优化
* 基于全映射的全自动框架，大量字段的POO进行部分映射时比较困难。
* 反射操作太多，导致数据库性能下降

1. MyBatis

* 轻量级，性能出色
* SQL和Java编码分开，功能边界清晰。Java代码专注业务、SQL语句专注数据
* o开发效率稍逊于Hibernate,但是完全能够接受

‍

# 搭建MyBatis

## 1.创建maven工程

* 打包方式：jar

* 引入依赖
* pom.xml配置文件：
* <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">  
   <modelVersion>4.0.0</modelVersion>  
   <groupId>org.example</groupId>  
   <artifactId>mybatis01</artifactId>  
   <version>1.0-SNAPSHOT</version>  
    
   <packaging>jar</packaging>  
    
    
   <name>Archetype - mybatis01</name>  
   <url>http://maven.apache.org</url>  
    
    
   <dependencies>  
   <!-- Mybatis核心 -->  
   <dependency>  
   <groupId>org.mybatis</groupId>  
   <artifactId>mybatis</artifactId>  
   <version>3.5.7</version>  
   </dependency>  
   <!-- junit测试 -->  
   <dependency>  
   <groupId>junit</groupId>  
   <artifactId>junit</artifactId>  
   <version>4.12</version>  
   <scope>test</scope>  
   </dependency>  
   <!-- MySQL驱动 -->  
   <dependency>  
   <groupId>mysql</groupId>  
   <artifactId>mysql-connector-java</artifactId>  
   <version>5.1.3</version>  
   </dependency>  
   </dependencies>  
    
  </project>

‍

‍

## 2.创建MyBatis的核心配置文件

习惯上命名为mybatis-config.xml​，这个文件名仅仅只是建议，并非强制要求。将来整合Spring之后，这个配置文件可以省略。

* 核心配置文件主要用于配置连接数据库的环境以及MyBatis的全局配置信息

* 核心配置文件存放的位置是src/main/resources目录下

核心配置文件中的标签必须按照固定的顺序(有的标签可以不写，但顺序一定不能乱)：  
properties、settings、typeAliases、typeHandlers、objectFactory、objectWrapperFactory、reflectorFactory、plugins、environments、databaseIdProvider、mappers

‍

1. typeAlias：设置某个具体的类型的别名，在后续使用就输入别名即可（一般不用，而是使用package，因为这个用一次设置一次）

* type：需要设置别名的类型的全类名
* alias：设置此类型的别名，且别名不区分大小写。若不设置此属性，该类型拥有默认的别名，即类名
* <typeAlias type="com.atguigu.mybatis.bean.User"></typeAlias>  
  <typeAlias type="com.atguigu.mybatis.bean.User" alias="user"></typeAlias>

1. environments：设置多个连接数据库的环境

* default：设置默认使用的环境的id
* ​<environments default="mysql\_test"> ​

1. environment：设置具体的连接数据库的环境信息  
   id：设置环境的唯一标识，可通过environments标签中的default设置某一个环境的id，表示默认使用的环境

* <environment id="mysql\_test">​

1. transactionManager：设置事务管理方式  
   type：设置事务管理方式，type="JDBC|MANAGED"  
   type="JDBC"：设置当前环境的事务管理都必须手动处理  
   type="MANAGED"：设置事务被管理，例如spring中的AOP  
   <transactionManager type="JDBC"/>​

1. dataSource：设置数据源

* + type：设置数据源的类型，type="POOLED|UNPOOLED|JNDI"

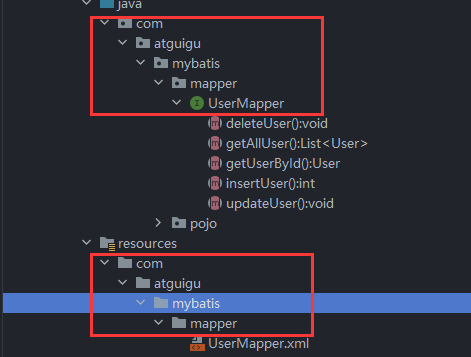
* + type="POOLED"：使用数据库连接池，即会将创建的连接进行缓存，下次使用可以从缓存中直接获取，不需要重新创建

* + type="UNPOOLED"：不使用数据库连接池，即每次使用连接都需要重新创建 type="JNDI"：调用上下文中的数据源
* ​<dataSource type="POOLED"> ​

<?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>  
 <!--引入properties文件，此时就可以${属性名}的方式访问属性值-->  
 <properties resource="jdbc.properties"></properties>  
 <settings>  
 <!--将表中字段的下划线自动转换为驼峰-->  
 <setting name="mapUnderscoreToCamelCase" value="true"/>  
 <!--开启延迟加载-->  
 <setting name="lazyLoadingEnabled" value="true"/>  
 </settings>  
 <typeAliases>  
 <!--以包为单位，设置改包下所有的类型都拥有默认的别名，即类名且不区分大小写-->  
 <package name="com.atguigu.mybatis.bean"/>  
 </typeAliases>  
 <environments default="mysql\_test">  
 <environment id="mysql\_test">  
 <transactionManager type="JDBC"/>  
 <dataSource type="POOLED">  
 <!--设置驱动类的全类名-->  
 <property name="driver" value="${jdbc.driver}"/>  
 <!--设置连接数据库的连接地址-->  
 <property name="url" value="${jdbc.url}"/>  
 <!--设置连接数据库的用户名-->  
 <property name="username" value="${jdbc.username}"/>  
 <!--设置连接数据库的密码-->  
 <property name="password" value="${jdbc.password}"/>  
 </dataSource>  
 </environment>  
 </environments>  
 <!--引入映射文件-->  
 <mappers>  
 <!-- <mapper resource="UserMapper.xml"/> -->  
 <!--  
 以包为单位，将包下所有的映射文件引入核心配置文件  
 注意：  
 1. 此方式必须保证mapper接口和mapper映射文件必须在相同的包下  
 2. mapper接口要和mapper映射文件的名字一致  
 -->  
 <package name="com.atguigu.mybatis.mapper"/>  
 </mappers>  
</configuration>

‍

‍

* ​​

‍

‍

## 3.创建实体类与数据表

SQL文件：

CREATE TABLE Student (  
 id INT AUTO\_INCREMENT PRIMARY KEY,  
 name VARCHAR(50),  
 age INT,  
 gender VARCHAR(10)  
);  
  
INSERT INTO Student (name, age, gender) VALUES ('Alice', 20, 'Female');  
INSERT INTO Student (name, age, gender) VALUES ('Bob', 22, 'Male');  
INSERT INTO Student (name, age, gender) VALUES ('Cathy', 21, 'Female');  
INSERT INTO Student (name, age, gender) VALUES ('David', 23, 'Male');  
INSERT INTO Student (name, age, gender) VALUES ('Emily', 19, 'Female');  
INSERT INTO Student (name, age, gender) VALUES ('Frank', 20, 'Male');  
INSERT INTO Student (name, age, gender) VALUES ('Grace', 22, 'Female');  
INSERT INTO Student (name, age, gender) VALUES ('Henry', 21, 'Male');  
INSERT INTO Student (name, age, gender) VALUES ('Ivy', 23, 'Female');  
INSERT INTO Student (name, age, gender) VALUES ('Jack', 20, 'Male');  
INSERT INTO Student (name, age, gender) VALUES ('Kelly', 22, 'Female');  
INSERT INTO Student (name, age, gender) VALUES ('Leo', 21, 'Male');  
INSERT INTO Student (name, age, gender) VALUES ('Mia', 19, 'Female');  
INSERT INTO Student (name, age, gender) VALUES ('Nick', 20, 'Male');  
INSERT INTO Student (name, age, gender) VALUES ('Olivia', 22, 'Female');  
INSERT INTO Student (name, age, gender) VALUES ('Peter', 21, 'Male');  
INSERT INTO Student (name, age, gender) VALUES ('Queenie', 23, 'Female');  
INSERT INTO Student (name, age, gender) VALUES ('Ryan', 20, 'Male');  
INSERT INTO Student (name, age, gender) VALUES ('Samantha', 22, 'Female');  
INSERT INTO Student (name, age, gender) VALUES ('Tom', 21, 'Male');

pojo实体类

package com.hui7.pojo;  
  
public class Student {  
 private int id;  
 private String name;  
 private int age;  
 private String gender;  
  
 public String getGender() {  
 return gender;  
 }  
  
 public void setGender(String gender) {  
 this.gender = gender;  
 }  
  
 public Student(int id, String name, int age) {  
 this.id = id;  
 this.name = name;  
 this.age = age;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 ", age=" + age +  
 ", gender='" + gender + '\'' +  
 '}';  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

‍

## 4.创建mapper接口

MyBatis中的mapper接口相当于以前的dao。但是区别在于，mapper仅仅是接口，我们不需要提供实现类

package com.hui7.mapper;  
  
public interface StudentMapper {  
   
 int InsertStudent();  
}

‍

‍

## 创建MyBatis的映射文件

* 相关概念：ORM（Object Relationship Mapping）对象关系映射。

* + 对象：Java的实体类对象

* + 关系：关系型数据库

* + 映射：二者之间的对应关系

| Java概念 | 数据库概念 |
| --- | --- |
| 类 | 表 |
| 属性 | 字段/列 |
| 对象 | 记录/行 |

‍

**POJO（Plain Old Java Object）/实体类：** 这是表示数据库表中记录的 Java 对象。

* 映射文件

* + 表所对应的实体类的类名+Mapper.xml

* + 例如：表t\_user，映射的实体类为User，所对应的映射文件为UserMapper.xml

* + 因此一个映射文件对应一个实体类，对应一张表的操作

* + MyBatis映射文件用于编写SQL，访问以及操作表中的数据

* + MyBatis映射文件存放的位置是src/main/resources/mappers目录下

‍

StudentMapper接口对应的是操作数据的方法。（解耦SQL语句到xml中）

StudentMapper.xml 是具体的Sql语句。

package com.hui7.mapper;  
  
public interface StudentMapper {  
  
 int InsertStudent();  
}

‍

* MyBatis中可以面向接口操作数据，要保证两个一致

* + mapper接口的全类名和映射文件的命名空间（namespace）保持一致

* + mapper接口中方法的方法名和映射文件中编写SQL的标签的id属性保持一致

在查询中必须要设置resultType。

StudentMapper.xml：mappe标签命名空间绑定的就是StudentMapper接口

四个增删改查固定的标签的ID绑定的是StudentMapper接口的方法

<?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为映射的根节点，用来管理DAO接口  
 namespace指定DAO接口的完整类名，表示mapper配置文件管理哪个DAO接口(包.接口名)  
 mybatis会依据这个接口动态创建一个实现类去实现这个接口，而这个实现类是一个Mapper对象  
 -->  
<mapper namespace="com.hui7.mapper.StudentMapper">  
 <!--  
 id = "接口中的方法名"  
 parameterType = "接口中传入方法的参数类型"  
 resultType = "返回实体类对象：包.类名" 处理结果集 自动封装  
 注意:sql语句后不要出现";"号  
 查询：select标签  
 增加：insert标签  
 修改：update标签  
 删除：delete标签  
 -->  
  
 <insert id="InsertStudent">  
 insert into Student values ('王梅',22,'女')  
 </insert>  
  
</mapper>

‍

‍

‍

‍

## 5.通过Test单元测试功能

* SqlSession：代表Java程序和数据库之间的会话。（HttpSession是Java程序和浏览器之间的会话）

* SqlSessionFactory：是“生产”SqlSession的“工厂”

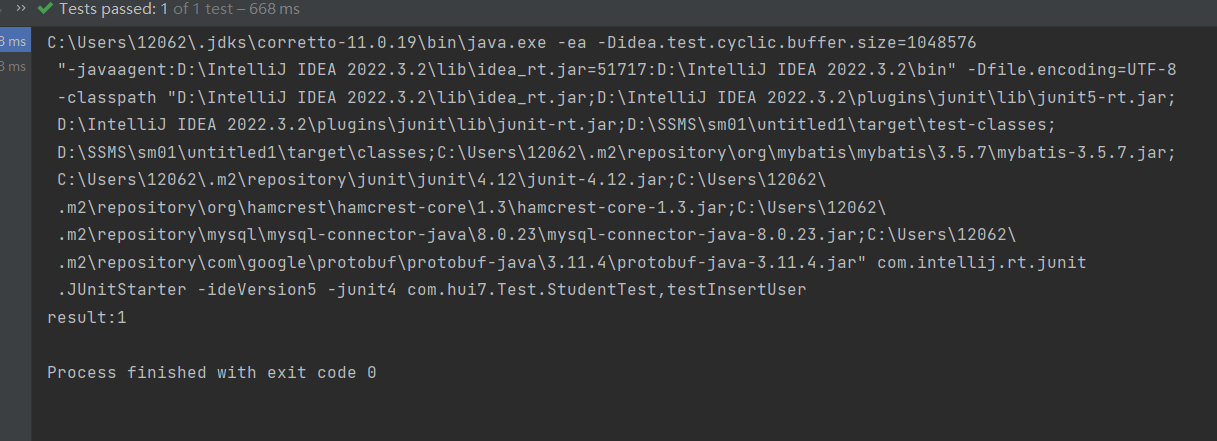
* 工厂模式：如果创建某一个对象，使用的过程基本固定，那么我们就可以把创建这个对象的相关代码封装到一个“工厂类”中，以后都使用这个工厂类来“生产”我们需要的对象

public class UserMapperTest {  
 @Test  
 public void testInsertUser() throws IOException {  
 //读取MyBatis的核心配置文件  
 InputStream is = Resources.getResourceAsStream("mybatis-config.xml");  
 //获取SqlSessionFactoryBuilder对象  
 SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  
 //通过核心配置文件所对应的字节输入流创建工厂类SqlSessionFactory，生产SqlSession对象  
 SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(is);  
 //获取sqlSession，此时通过SqlSession对象所操作的sql都必须手动提交或回滚事务  
 //SqlSession sqlSession = sqlSessionFactory.openSession();  
 //创建SqlSession对象，此时通过SqlSession对象所操作的sql都会自动提交   
 SqlSession sqlSession = sqlSessionFactory.openSession(true);  
 //通过代理模式创建UserMapper接口的代理实现类对象（就是帮我们创建接口的实例化对象）  
 UserMapper userMapper = sqlSession.getMapper(UserMapper.class);  
 //调用UserMapper接口中的方法，就可以根据UserMapper的全类名匹配元素文件，通过调用的方法名匹配映射文件中的SQL标签，并执行标签中的SQL语句  
 int result = userMapper.insertUser();  
 //提交事务  
 //sqlSession.commit();  
 System.out.println("result:" + result);  
 }  
}

‍

* 此时需要手动提交事务，如果要自动提交事务，则在获取sqlSession对象时，使用SqlSession sqlSession = sqlSessionFactory.openSession(true);​​，传入一个Boolean类型的参数，值为true，这样就可以自动提交

运行结果：

​​

‍

## 加入log4j日志功能（非必要）

### 1. 在[mybatis](https://so.csdn.net/so/search?q=mybatis&spm=1001.2101.3001.7020)的配置文件中配置

要注意的点: mybatis的配置文件里面的标签有存放位置严格要求, 这个settings必须放在合适的位置

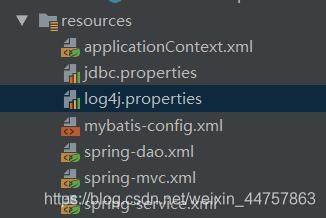
<setting name="logImpl" value="LOG4J"/></settings>

### 2. 导入jar包

在pom.xml中添加坐标:

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

### 3. 在resources目录下创建[log4j](https://so.csdn.net/so/search?q=log4j&spm=1001.2101.3001.7020).properties

如图所示:  
​​

### log4j.properties的简单配置

1. 加入log4j的配置文件

* + log4j的配置文件名为log4j.xml，在src/main/resources目录下创建

* + 日志的级别：FATAL(致命)>ERROR(错误)>WARN(警告)>INFO(息)>DEBUG(调试) 从左到右打印的内容越来越详细

######################## 将等级为DEBUG的日志信息输出到consoleh和file两个目的地, console和file的定义在下面的代码  
log4j.rootLogger=DEBUG,console,file  
  
########################控制台输出的相关设置  
log4j.appender.console = org.apache.log4j.ConsoleAppender  
#在控制台输出  
log4j.appender.console.Target = System.out  
#在DEBUG级别输出  
log4j.appender.console.Threshold = DEBUG  
log4j.appender.console.layout = org.apache.log4j.PatternLayout  
#日志格式  
log4j.appender.console.layout.ConversionPattern=[%c]-%m%n  
  
######################日志输出级别  
log4j.logger.org.mybatis=DEBUG  
log4j.logger.java.sql=DEBUG  
log4j.logger.java.sql.Statement=DEBUG  
log4j.logger.java.sql.ResultSet=DEBUG  
log4j.logger.java.sql.PreparedStatement=DEBUG

‍

‍

‍

## 6.外部数据源：properties（优化）

在MyBatis中，外部数据源的properties文件通常用于存储数据库连接信息，例如数据库的URL、用户名、密码等。这样做的好处是可以将数据库连接信息与代码解耦，使得代码更加清晰和易于维护。这样的设计使得我们可以轻松地在不同的环境中切换数据库，而不需要修改代码。

‍

### 创建properties

在resources中创建jdbc.properties​文件，用与存储数据库的配置信息

文件名的规范：这个properties文件是什么相关的就取什么名字

变量名的规范：当前是jdbc.properties​，那么变量名就要以，jdbc.xxxx​来命名。

jdbc.driver=com.mysql.cj.jdbc.Driver  
jdbc.url=jdbc:mysql://localhost:3306/mybatis  
jdbc.username=beidao23  
jdbc.password=beidaomitu233

‍

### 引入properties

1. 引入

目前需要把mybatis中的数据连接配置放到外部资源中

在 mybatis-config.xml​中的<configuration>​标签内添加

<properties resource="jdbc.properties"></properties>

1. 替换数据：

<configuration>  
 <properties resource="jdbc.properties"></properties>  
 <!--设置连接数据库的环境-->  
 <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}"/>  
 .....

‍

# 使用MyBatis

‍

* 注意：

* 1. 查询的标签select必须设置属性resultType或resultMap，用于设置实体类和数据库表的映射关系

* + - resultType：自动映射，用于属性名和表中字段名一致的情况

* + - resultMap：自定义映射，用于一对多或多对一或字段名和属性名不一致的情况

* 1. 当查询的数据为多条时，不能使用实体类作为返回值，只能使用集合，否则会抛出异常TooManyResultsException；但是若查询的数据只有一条，可以使用实体类或集合作为返回值

‍

## MyBatis传参进行增删改查（重点）

这里获取的参数是从前端传输来的数据，例如用户名，文章id等等，后端基于这个参数拼接到sql语句中进行查询。

* MyBatis获取参数值的两种方式：${}和#{}

* ${}的本质就是字符串拼接，#{}的本质就是占位符赋值

${}使用字符串拼接的方式拼接sql，若为字符串类型或日期类型的字段进行赋值时，需要手动加单引号，且会导致sql注入。

#{}使用占位符赋值的方式拼接sql，此时为字符串类型或日期类型的字段进行赋值时，可以自动添加单引号

‍

前端->service->实体类->mybatis->mysql

所以mybatis获取参数值的上一个地方是实体类（xxxmapper.java）

‍

‍

‍

用参数来传递值进行CURD操作的流程：

使用添加数据来演示

1. 在实体类中添加方法和参数（五种传参方式）

* //@注解  
   void InsertStudent(@Param("StudentName") String StudentName,@Param("age") int age ,@Param("gender") String gender);

1. 在映射中添加获取参数值的方式以及要获取的参数

* <insert id="InsertStudent">  
   INSERT INTO student(StudentName, age,gender) VALUES(#{StudentName}, #{age},#{gender})  
   </insert>

1. 在Test测试类中写实参，传输进去。

* @Test  
   public void testInsertUser() throws IOException {  
   //读取MyBatis的核心配置文件  
   InputStream is = Resources.getResourceAsStream("mybatis-config.xml");  
   //获取SqlSessionFactoryBuilder对象  
   SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  
   //通过核心配置文件所对应的字节输入流创建工厂类SqlSessionFactory，生产SqlSession对象  
   SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(is);  
   //获取sqlSession，此时通过SqlSession对象所操作的sql都必须手动提交或回滚事务  
   //SqlSession sqlSession = sqlSessionFactory.openSession();  
   //创建SqlSession对象，此时通过SqlSession对象所操作的sql都会自动提交  
   SqlSession sqlSession = sqlSessionFactory.openSession(true);  
   //通过代理模式创建UserMapper接口的代理实现类对象  
   StudentMapper userMapper = sqlSession.getMapper(StudentMapper.class);  
   //调用UserMapper接口中的方法，就可以根据UserMapper的全类名匹配元素文件，通过调用的方法名匹配映射文件中的SQL标签，并执行标签中的SQL语句  
   userMapper.InsertStudent("牛马z",22,"男");  
   System.out.println("插入成功");  
    
   }

‍

演示代码并不全,主要关注实现的逻辑，只有重要突出逻辑部分。

在本章 结尾有完整的代码。

每种传参方法代码了一种数据库操作（增删改查）

‍

### 传输：单个字面量（删除）

* 若mapper接口中的方法参数为单个的字面量类型，此时可以使用${}和#{}以任意的名称获取参数的值，注意${}需要手动加单引号

//StudentMapper.java  
 // 根据id删除学生记录 单字面量  
 void deleteStudentById(int id);  
  
//StudentTest.java  
 @Test  
 public void deleteStudentById(){  
 int studentIdToDelete = 1;   
 studentMapper.deleteStudentById(studentIdToDelete);  
 System.out.println("删除成功");  
 }

StudentMapper.xml

<!--根据id来删除-->  
 <delete id="deleteStudentById" parameterType="int">  
 DELETE FROM student  
 WHERE id = #{id}  
 </delete>

‍

### 传输：多个字面量（）

* 若mapper接口中的方法参数为多个时，此时MyBatis会自动将这些参数放在一个map集合中

* 1. 以arg0,arg1...为键，以参数为值；

* 1. 以param1,param2...为键，以参数为值；

* 因此只需要通过${}和#{}访问map集合的键就可以获取相对应的值，注意${}需要手动加单引号。

* 使用arg或者param都行，要注意的是，arg是从arg0开始的，param是从param1开始的

<!--User checkLogin(String username,String password);-->  
<select id="checkLogin" resultType="User">   
 select \* from t\_user where username = #{arg0} and password = #{arg1}   
</select>

<!--User checkLogin(String username,String password);-->  
<select id="checkLogin" resultType="User">  
 select \* from t\_user where username = '${param1}' and password = '${param2}'  
</select>

‍

‍

### 传输：map集合类型（查询）

* 若mapper接口中的方法需要的参数为多个时，此时可以手动创建map集合，将这些数据放在map中只需要通过${}和#{}访问map集合的键就可以获取相对应的值，注意${}需要手动加单引号

* 查询时一定要写 resultType="com.hui7.pojo.Student"​​返回值类型，这个是别名，可以直接写student，不区分大小写
* ​Map<String, Object>​ 是一个 Java 中的集合类型，它表示一个键值对（key-value）的映射。在这个特定的声明中，String​ 表示键的类型，而 Object​ 表示值的类型。这种泛型声明的 Map​ 允许你使用任意类型的值，因为 Object​ 是所有 Java 类型的超类。

//StudentMapper.java  
 // 根据id查询学生记录 多个字面量  
 Student selectStudentByMap(Map<String, Object> params);  
  
//StudentTest.java  
 @Test  
// 查询单个Student数据  
 public void selectStudentByMap() {  
 // 创建 Map 参数  
 Map<String, Object> params = new HashMap<>();  
 params.put("id", 10);  
 params.put("studentName", "Sophia Garcia");  
  
 // 测试查询方法  
 Student student = studentMapper.selectStudentByMap(params);  
  
 if (student != null) {  
 System.out.println("查询结果：" + student.toString());  
 } else {  
 System.out.println("未找到对应的学生记录");  
 }  
 }

<!-- 根据id和名字查询学生记录 -->  
 <select id="selectStudentByMap" parameterType="java.util.Map" resultType="com.hui7.pojo.Student">  
 SELECT \* FROM student WHERE id = #{id} AND studentName = #{studentName}  
 </select>

‍

​**resultType**​**：**\* resultType​ 用于指定查询结果的类型，即每一行数据将被映射为一个指定类型的对象。

* 对于简单的查询，可以直接指定 Java 对象的类型，MyBatis 将会尝试通过列名和属性名的匹配来进行映射。

‍

​**resultMap**​**：**\* resultMap​ 是一种更加灵活和详细的映射方式，它允许你手动配置查询结果和 Java 对象之间的映射关系。

* 通过使用 resultMap​，你可以指定每个列如何映射到 Java 对象的属性，包括复杂类型、关联关系等。

* 与 resultType​ 不同，resultMap​ 提供更多的控制权，适用于复杂的查询场景。

‍

### 传输：实体类类型的参数（改）

* 若mapper接口中的方法参数为实体类对象时此时可以使用${}和#{}，通过访问实体类对象中的属性名获取属性值，注意${}需要手动加单引号

//StudentMapper.java  
 // 更新学生记录 实体类参数  
 void updateStudent(Student student);  
  
//StudentTest.java  
 @Test  
 //更新Studen数据  
 public void updateStudent(){  
 // 创建学生类并添加信息  
 Student updatedStudent = new Student();  
 updatedStudent.setId(2); // Assuming you want to update the student with ID 1  
 updatedStudent.setStudentName("被改名的牛马z");  
 updatedStudent.setAge(25);  
 updatedStudent.setGender("Female");  
  
 // 将学生类作为参数传给更新的方法  
 studentMapper.updateStudent(updatedStudent);  
  
 System.out.println("更新成功");  
 }

<!-- 通过id来查找要修改的人并修改 -->  
 <update id="updateStudent">  
 UPDATE student  
 SET studentName = #{studentName}, age = #{age}, gender = #{gender}  
 WHERE id = #{id}  
 </update>

‍

‍

### 传输：@Param注解参数（添加——这个是重点）

* 可以通过@Param注解标识mapper接口中的方法参数，此时，会将这些参数放在map集合中

* 1. 以@Param注解的value属性值为键，以参数为值；

* 1. 以param1,param2...为键，以参数为值；

* 只需要通过${}和#{}访问map集合的键就可以获取相对应的值，注意${}需要手动加单引号

‍

//StudentMapper.java  
 //添加学生数据 @注解 的形式传参,  
 void InsertStudent(@Param("StudentName") String StudentName,@Param("age") int age ,@Param("gender") String gender);  
  
  
//StudentTest.java  
 @Test  
// 插入学生数据  
 public void testInsertStudent() {  
 studentMapper.InsertStudent("牛马z", 32, "男");  
 System.out.println("插入成功");  
 }

<insert id="InsertStudent">  
 INSERT INTO student(StudentName, age,gender) VALUES(#{StudentName}, #{age},#{gender})  
 </insert>

‍

‍

### 总结

* 把物种方式分成两种情况进行处理就足够了

* 1. 实体类类型的参数

* 1. 使用@Param标识参数

‍

#### 全部的代码

StudentTest测试类

package com.hui7.Test;  
  
import com.hui7.mapper.StudentMapper;  
import com.hui7.pojo.Student;  
import org.apache.ibatis.io.Resources;  
import org.apache.ibatis.session.SqlSession;  
import org.apache.ibatis.session.SqlSessionFactory;  
import org.apache.ibatis.session.SqlSessionFactoryBuilder;  
import org.junit.After;  
import org.junit.Before;  
import org.junit.Test;  
  
import java.io.IOException;  
import java.io.InputStream;  
import java.util.HashMap;  
import java.util.Map;  
  
public class StudentTest {  
  
 private SqlSession sqlSession;  
 private StudentMapper studentMapper;  
  
 @Before  
 public void setUp() throws IOException {  
 // 读取 MyBatis 的核心配置文件  
 InputStream is = Resources.getResourceAsStream("mybatis-config.xml");  
 // 获取 SqlSessionFactoryBuilder 对象  
 SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  
 // 通过核心配置文件所对应的字节输入流创建工厂类 SqlSessionFactory，生产 SqlSession 对象  
 SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(is);  
 // 创建 SqlSession 对象，此时通过 SqlSession 对象所操作的 SQL 都会自动提交  
 sqlSession = sqlSessionFactory.openSession(true);  
 // 通过代理模式创建 StudentMapper 接口的代理实现类对象  
 studentMapper = sqlSession.getMapper(StudentMapper.class);  
 }  
  
 @Test  
// 插入学生数据  
 public void testInsertStudent() {  
 studentMapper.InsertStudent("牛马z", 32, "男");  
 System.out.println("插入成功");  
 }  
  
 @Test  
// 查询单个Student数据  
 public void selectStudentByMap() {  
 // 创建 Map 参数  
 Map<String, Object> params = new HashMap<>();  
 params.put("id", 10);  
 params.put("studentName", "Sophia Garcia");  
  
 // 测试查询方法  
 Student student = studentMapper.selectStudentByMap(params);  
  
 if (student != null) {  
 System.out.println("查询结果：" + student.toString());  
 } else {  
 System.out.println("未找到对应的学生记录");  
 }  
 }  
  
 @Test  
 //更新Studen数据  
 public void updateStudent(){  
 // 创建学生类并添加信息  
 Student updatedStudent = new Student();  
 updatedStudent.setId(2); // Assuming you want to update the student with ID 1  
 updatedStudent.setStudentName("被改名的牛马z");  
 updatedStudent.setAge(25);  
 updatedStudent.setGender("Female");  
  
 // 将学生类作为参数传给更新的方法  
 studentMapper.updateStudent(updatedStudent);  
  
 System.out.println("更新成功");  
  
  
 }  
  
 @Test  
 public void deleteStudentById(){  
 // Specify the ID of the student you want to delete  
 int studentIdToDelete = 1; // Assuming you want to delete the student with ID 1  
 // Call the delete method in StudentMapper  
 studentMapper.deleteStudentById(studentIdToDelete);  
 System.out.println("删除成功");  
 }  
  
 @After  
 public void tearDown() {  
 // 关闭 SqlSession  
 sqlSession.close();  
 }  
}

‍

StudentMapper 方法接口

package com.hui7.mapper;  
  
import org.apache.ibatis.annotations.Param;  
import com.hui7.pojo.Student;  
  
import java.util.Map;  
  
public interface StudentMapper {  
  
  
 // 根据id删除学生记录 单字面量  
 void deleteStudentById(int id);  
  
 // 根据id查询学生记录 多个字面量  
 Student selectStudentByMap(Map<String, Object> params);  
  
 // 更新学生记录 实体类参数  
 void updateStudent(Student student);  
  
 //添加学生数据 @注解 的形式传参,  
 void InsertStudent(@Param("StudentName") String StudentName,@Param("age") int age ,@Param("gender") String gender);  
  
  
}

‍

Student学生类

package com.hui7.pojo;  
  
public class Student {  
 private int id;  
 private String StudentName;  
 private int age;  
 private String gender;  
  
 public String getGender() {  
 return gender;  
 }  
  
 public Student() {  
 }  
  
 public void setGender(String gender) {  
 this.gender = gender;  
 }  
  
 public Student(int id, String StudentName, int age) {  
 this.id = id;  
 this.StudentName = StudentName;  
 this.age = age;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 @Override  
 public String toString() {  
 return "Student{" +  
 "id=" + id +  
 ", StudentName='" + StudentName + '\'' +  
 ", age=" + age +  
 ", gender='" + gender + '\'' +  
 '}';  
 }  
  
 public String getStudentName() {  
 return StudentName;  
 }  
  
 public void setStudentName(String studentName) {  
 StudentName = studentName;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

‍

StudentMapper.xml sql语句

<?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为映射的根节点，用来管理DAO接口  
 namespace指定DAO接口的完整类名，表示mapper配置文件管理哪个DAO接口(包.接口名)  
 mybatis会依据这个接口动态创建一个实现类去实现这个接口，而这个实现类是一个Mapper对象  
 -->  
  
<mapper namespace="com.hui7.mapper.StudentMapper">  
 <!--  
 id = "接口中的方法名"  
 parameterType = "接口中传入方法的参数类型"  
 resultType = "返回实体类对象：包.类名" 处理结果集 自动封装  
 注意:sql语句后不要出现";"号  
 查询：select标签  
 增加：insert标签  
 修改：update标签  
 删除：delete标签  
 -->  
  
 <insert id="InsertStudent">  
 INSERT INTO student(StudentName, age,gender) VALUES(#{StudentName}, #{age},#{gender})  
 </insert>  
  
  
 <!-- 根据id和名字查询学生记录 -->  
 <select id="selectStudentByMap" parameterType="java.util.Map" resultType="com.hui7.pojo.Student">  
 SELECT \* FROM student WHERE id = #{id} AND studentName = #{studentName}  
 </select>  
  
 <!-- 通过id来查找要修改的人并修改 -->  
 <update id="updateStudent">  
 UPDATE student  
 SET studentName = #{studentName}, age = #{age}, gender = #{gender}  
 WHERE id = #{id}  
 </update>  
  
  
 <!--根据id来删除-->  
 <delete id="deleteStudentById" parameterType="int">  
 DELETE FROM student  
 WHERE id = #{id}  
 </delete>  
  
  
</mapper>

‍

sql代码

-- 创建 Student 表  
CREATE TABLE Student (  
 id INT PRIMARY KEY AUTO\_INCREMENT,  
 studentName VARCHAR(255) NOT NULL,  
 age INT,  
 gender VARCHAR(10)  
);  
  
-- 插入测试数据  
INSERT INTO Student (studentName, age, gender) VALUES  
 ('John Doe', 20, 'Male'),  
 ('Jane Smith', 22, 'Female'),  
 ('Bob Johnson', 25, 'Male'),  
 ('Alice Williams', 21, 'Female'),  
 ('Tom Brown', 23, 'Male'),  
 ('Emily Davis', 24, 'Female'),  
 ('Mike Wilson', 22, 'Male'),  
 ('Sara Miller', 23, 'Female'),  
 ('Alex Turner', 24, 'Male'),  
 ('Sophia Garcia', 21, 'Female');

‍

pom.xml

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <groupId>org.example</groupId>  
 <artifactId>mybatis01</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <packaging>jar</packaging>  
  
  
 <name>Archetype - mybatis01</name>  
 <url>http://maven.apache.org</url>  
  
 <dependencies>  
 <!-- Mybatis核心 -->  
 <dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>3.5.7</version>  
 </dependency>  
 <!-- junit测试 -->  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.12</version>  
 <scope>test</scope>  
 </dependency>  
 <!-- MySQL驱动 -->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>8.0.23</version>  
 </dependency>  
 </dependencies>  
  
</project>

‍

‍

‍

## MyBatis返回值的接收

1. 如果查询出的数据只有一条，可以通过

* 1. 实体类对象接收

* 1. List集合接收

* 1. Map集合接收，结果{password=123456, sex=男, id=1, age=23, username=admin}​​

1. 如果查询出的数据有多条，一定不能用实体类对象接收，会抛异常TooManyResultsException，可以通过

* 1. 实体类类型的LIst集合接收

* 1. Map类型的LIst集合接收

* 1. 在mapper接口的方法上添加@MapKey注解

‍

### 返回单条

#### 查询单个数据

查询用户的总记录数

@return

在MyBatis中，对于Java中常用的类型都设置了类型别名

* 例如：java.lang.Integer-->int|integer

* 例如：int-->\_int|\_integer

* 例如：Map-->map,List-->list

int getCount();  
  
 @Test  
 //以整型返回一个计算之后的数据:count()...等函数的结果  
 public void getCount(){  
 int StudentNumber= studentMapper.getCount();  
 System.out.println("学生总数为:"+StudentNumber);  
  
 }

<!-- 获取数据总数 -->  
 <select id="getCount" resultType="\_integer">  
 select count(id) from student  
 </select>

‍

#### 以map集合形式返回单条数据

/\*\*   
 \* 根据用户id查询用户信息为map集合   
 \* @param id   
 \* @return   
 \*/   
Map<String, Object> getUserToMap(@Param("id") int id);

<!--Map<String, Object> getUserToMap(@Param("id") int id);-->  
<select id="getUserToMap" resultType="map">  
 select \* from t\_user where id = #{id}  
</select>  
<!--结果：{password=123456, sex=男, id=1, age=23, username=admin}-->

‍

‍

### 返回多条

#### 将数据以List集合返回

/\*\*  
 \* 查询所有用户信息  
 \* @return  
 \*/  
  
// 获取多条数据时，要将返回值设置为list集合或者map集合，或者两个组合  
 List<Student> getAllStudent();  
  
 @Test  
 // 以list集合返回多条数据  
 public void getAllStudent(){  
 List<Student> StudentList= studentMapper.getAllStudent();  
 System.out.println("获取了全部学生");  
 System.out.println(StudentList);  
 }

0 <!-- 以List<Map<String,Object>>返回数据 -->  
 <select id="GetSomeStudent" resultType="com.hui7.pojo.Student">  
 SELECT \* FROM student WHERE age = #{age}  
 </select>

‍

#### 以Map集合返回多条数据

‍

/\*\*  
 \* 查询所有用户信息为map集合  
 \* @return  
 \* 将表中的数据以map集合的方式查询，一条数据对应一个map；若有多条数据，就会产生多个map集合，并且最终要以一个map的方式返回数据，此时需要通过@MapKey注解设置map集合的键，值是每条数据所对应的map集合  
 \*/  
@MapKey("id")  
Map<String, Object> getAllUserToMap();

<!--Map<String, Object> getAllUserToMap();-->  
<select id="getAllUserToMap" resultType="map">  
 select \* from t\_user  
</select>  
<!--  
 结果：  
 {  
 1={password=123456, sex=男, id=1, age=23, username=admin},  
 2={password=123456, sex=男, id=2, age=23, username=张三},  
 3={password=123456, sex=男, id=3, age=23, username=张三}  
 }  
-->

‍

‍

#### 以List<Map<String, Object>>返回多条map集合

/\*\*   
 \* 查询所有用户信息为map集合   
 \* @return   
 \* 将表中的数据以map集合的方式查询，一条数据对应一个map；若有多条数据，就会产生多个map集合，此时可以将这些map放在一个list集合中获取   
 \*/   
 // 获取多条数据时，要将返回值设置为list集合或者map集合，或者两个组合  
 List<Map<String,Object>> GetSomeStudent(@Param("age") int age);  
 // 以map形式返回数据,一个map实际上只是一条数据,而用List集合装map就可以返回多个  
 @Test  
 public void GetSomeStudent(){  
// 返回一个包含多个map的List集合  
 List<Map<String,Object>> SomeStudent=studentMapper.GetSomeStudent(32);  
 System.out.println("32岁的学生有:"+SomeStudent);  
 }

‍

<!--Map<String, Object> getAllUserToMap();-->   
 <!-- 以List<Map<String,Object>>返回数据 -->  
 <select id="GetSomeStudent" resultType="com.hui7.pojo.Student">  
 SELECT \* FROM student WHERE age = #{age}  
 </select>  
<!--  
 结果：  
32岁的学生有:[Student{id=15, StudentName='牛马z', age=32, gender='男'}, Student{id=14, StudentName='牛马z', age=32, gender='男'}, Student{id=16, StudentName='牛马z', age=32, gender='男'}, Student{id=17, StudentName='牛马z', age=32, gender='男'}, Student{id=18, StudentName='牛马z', age=32, gender='男'}, Student{id=19, StudentName='牛马z', age=32, gender='男'}]  
  
-->

‍

## 其他SQL操作

‍

### 模糊查询

/\*\*  
 \* 根据用户名进行模糊查询  
 \*/  
 List<Student> selectStudentByLikeName(@Param("StudentName") String StudentName);  
  
 @Test  
 public void selectStudentByLikeName(){  
 List<Student> SelectByLikeName= studentMapper.selectStudentByLikeName("牛马");  
 System.out.println("查找到的人:"+SelectByLikeName);  
 }

<!-- 以List<Map<String,Object>>返回数据 -->  
 <select id="selectStudentByLikeName" resultType="com.hui7.pojo.Student">  
 SELECT \* FROM student WHERE studentName LIKE "%"#{StudentName}"%"  
 </select>

* 其中select \* from t\_user where username like "%"#{StudentName}"%"​​是最常用的
* 主要有以下三种：

* + select \* from student where username like concat('%',#{StudentName},'%')

* + select \* from student where username like '%${StudentName}%'

* + **select \* from student where username like concat('%',#{StudentName},'%')**

‍

### 批量删除

批量删除只能使用${}。如果使用#{}，会解析成一个整体

#{}传参：delete from t\_user where id in ('1,2,3')​，这样是将1,2,3​看做是一个整体，只有id为1,2,3​的数据会被删除。

‍

正确的SQL的语句：delete from t\_user where id in (1,2,3)​，或者delete from t\_user where id in ('1','2','3')​

‍

/\*\*  
 \* 根据id批量删除  
 \* 使用多个id组合成的字符串  
 \*/  
 int deleteStudents(@Param("id") String ids);

<!-- 以List<Map<String,Object>>返回数据 -->  
 <select id="deleteStudents" resultType="com.hui7.pojo.Student">  
 DELETE FROM student where id in(${ids})  
 </select>

删除没有返回值。

// 批量删除  
 @Test  
 public void deleteStudents(){  
 studentMapper.deleteStudents("12,13,14");  
 System.out.println("删除成功");  
 }

‍

### 动态设置表名

* 只能使用${}，因为表名不能加单引号

#### 动态设置表名的场景：

* 1. **多租户系统：**

* + - **示例场景：** 在一个多租户系统中，不同的租户可能共享相同的数据库结构，但需要将数据存储在不同的表中，以实现数据隔离。

* + - **应用：** 在一个通用的数据操作方法中，动态设置表名可以根据不同的租户动态选择对应的表，而不是在代码中硬编码表名。

* 1. **动态数据源切换：**

* + - **示例场景：** 在某些情况下，需要切换数据库连接，操作不同的数据库表。

* + - **应用：** 可以根据业务需求，动态设置表名，以适应不同的数据源，实现数据库的切换。
* 总体而言，这两个功能在处理复杂的数据库操作、多租户系统和需要动态切换数据源的情况下，提供了更高的灵活性和可定制性。然而，对于简单的场景，可能并不是必需的功能。
* ‍

/\*\*  
 \* 查询指定表中的数据  
 \* @param tableName   
 \* @return java.util.List<com.atguigu.mybatis.pojo.User>  
 \* @date 2022/2/27 14:41  
 \*/  
List<User> getUserByTable(@Param("tableName") String tableName);

<!--List<User> getUserByTable(@Param("tableName") String tableName);-->  
<select id="getUserByTable" resultType="User">  
 select \* from ${tableName}  
</select>

### 添加功能获取自增的主键

* 使用场景

* + t\_clazz(clazz\_id,clazz\_name)

* + t\_student(student\_id,student\_name,clazz\_id)

* 1. 添加班级信息

* 1. 获取新添加的班级的id

* 1. 为班级分配学生，即将某学的班级id修改为新添加的班级的id

* 在mapper.xml中设置两个属性

* + useGeneratedKeys：设置使用自增的主键

* + keyProperty：因为增删改有统一的返回值是受影响的行数，因此只能将获取的自增的主键放在传输的参数user对象的某个属性中
  + ‍

#### 获取自增主键的场景：

* 1. **插入数据后需要获得自动生成的主键值：**

* + - **示例场景：** 在插入新的记录后，需要获取自动生成的主键值，以便在后续操作中使用。

* + - **应用：** 在一对多关系中，当你需要在一个表中插入记录，并将生成的主键作为外键插入到另一个表中时，获取自增主键就显得尤为重要。

* 1. **需要将生成的主键值作为外键插入到关联表中：**

* + - **示例场景：** 在插入数据后，需要将生成的主键值作为外键插入到另一张关联表中。

* + - **应用：** 在上述的例子中，添加班级信息后，需要将新添加的班级的主键作为学生表中的外键，从而实现关联。

/\*\*  
 \* 添加用户信息  
 \* @param user   
 \* @date 2022/2/27 15:04  
 \*/  
void insertUser(User user);

<!--void insertUser(User user);-->  
<insert id="insertUser" useGeneratedKeys="true" keyProperty="id">  
 insert into t\_user values (null,#{username},#{password},#{age},#{sex},#{email})  
</insert>

//测试类  
@Test  
public void insertUser() {  
 SqlSession sqlSession = SqlSessionUtils.getSqlSession();  
 SQLMapper mapper = sqlSession.getMapper(SQLMapper.class);  
 User user = new User(null, "ton", "123", 23, "男", "123@321.com");  
 mapper.insertUser(user);  
 System.out.println(user);  
 //输出：user{id=10, username='ton', password='123', age=23, sex='男', email='123@321.com'}，自增主键存放到了user的id属性中  
}

‍

‍

‍

# resultMap自定义映射

‍

自定义映射的使用场景

1. 数据表字段名和变量名的不同时。（命名规则是不一样的,数据库推荐使用 \_ 来命名，而java是驼峰命名）

1. 多个表对一个表的映射（多表联查）

1. 一个表对多个表的映射

‍

‍

## resultMap处理字段和属性的映射关系

‍

### 字段与属性名的转换

* **若字段名和实体类中的属性名不一致**，则可以通过resultMap设置自定义映射，即使字段名和属性名一致的属性也要映射，也就是全部属性都要列出来

<resultMap id="empResultMap" type="Emp">  
 <id property="eid" column="eid"></id>  
 <result property="empName" column="emp\_name"></result>  
 <result property="age" column="age"></result>  
 <result property="sex" column="sex"></result>  
 <result property="email" column="email"></result>  
</resultMap>  
<!--List<Emp> getAllEmp();-->  
<select id="getAllEmp" resultMap="empResultMap">  
 select \* from t\_emp  
</select>

* **若字段名和实体类中的属性名不一致，但是字段名符合数据库的规则（使用\_）**，实体类中的属性名符合Java的规则（使用驼峰）。此时也可通过以下**两种**方式处理字段名和实体类中的属性的映射关系

* 1. 可以通过**为字段起别名**的方式，保证和实体类中的属性名保持一致
  + <!--List<Emp> getAllEmp();-->  
    <select id="getAllEmp" resultType="Emp">  
     select eid,emp\_name empName,age,sex,email from t\_emp  
    </select>

* 1. 可以在MyBatis的核心配置文件中的setting​​标签中，设置一个全局配置信息mapUnderscoreToCamelCase，可以在查询表中数据时，自动将\_类型的字段名转换为驼峰，例如：字段名user\_name，设置了mapUnderscoreToCamelCase，此时字段名就会转换为userName
  + <configuration>  
     <!-- 配置mybatis自动转换为驼峰式命名 -->  
     <settings>  
     <setting name="mapUnderscoreToCamelCase" value="true"/>  
     </settings>  
      
    </configuration>
* ‍

‍

## 多对一映射处理

‍

多对一映射是数据库中一种常见的关系映射方式，通常用于描述两个表之间的关系，其中多个记录在一个表中关联到另一个表中的一个记录。具体而言，多对一关系表示多个记录在一个表（"多" 的一端）关联到另一个表的一个记录（"一" 的一端）。

以下是多对一映射的一些特征：

1. **关联方向：** 多对一映射表示的是一对多的关系，其中多的一端关联到另一端的一个记录。关系通常在数据表中通过外键来表示。

1. **示例场景：** 考虑一个学校管理系统，有两个表：Student​ 表和 Class​ 表。一个班级中可以有多个学生，但每个学生只属于一个班级。在这种情况下，学生和班级之间的关系就是多对一。

1. **表现形式：** 多对一关系通常在数据库表中通过在 "多" 的一端的表中添加外键来表示。外键指向 "一" 的一端的表的主键。

1. **实体类关系映射：** 在面向对象的编程中，多对一关系通常通过在对象模型中使用引用关系来表示。在上述学校管理系统的例子中，Student​ 类中可能包含一个指向 Class​ 对象的引用，表示学生所属的班级。

1. **查询：** 查询多对一关系时，可以通过关联表的外键和主键进行连接查询，以获取相关联的信息。在上述例子中，可以通过学生表中的班级外键和班级表的主键进行连接查询，获取学生所属的班级信息。

‍

场景：

1. **多个学生属于同一个班级：**

* + **主实体：** 班级

* + **从实体：** 学生

* + 一个班级可以包含多个学生，但每个学生只能属于一个班级。

1. **多个订单属于同一个客户：**

* + **主实体：** 客户

* + **从实体：** 订单

* + 一个客户可以有多个订单，但每个订单只能属于一个客户。

在数据库中，多对一关系通常通过在从实体中添加外键，指向主实体的主键来建立。外键在从实体中表示关联，指向主实体中的唯一标识。这种映射关系有助于在数据库中维护实体之间的关系，并支持更复杂的查询和分析。

实际案例：

在数据库查询中，多对一关系通常在需要获取从实体（子实体）相关的主实体（主实体）信息时才会用到。这种查询场景通常涉及使用 JOIN 操作，将从实体与主实体关联起来，以便获取更完整的信息。以下是一些使用多对一关系的查询场景：

1. **获取学生所属班级信息：**

* + **场景：** 在查询学生信息时，需要同时获取学生所属班级的信息。

* + **查询语句：**
  + SELECT students.\*, classes.class\_name  
    FROM students  
    JOIN classes ON students.class\_id = classes.class\_id;

1. **获取订单所属客户信息：**

* + **场景：** 在查询订单信息时，需要同时获取订单所属客户的信息。

* + **查询语句：**
  + SELECT orders.\*, customers.customer\_name  
    FROM orders  
    JOIN customers ON orders.customer\_id = customers.customer\_id;

1. **获取评论所属文章信息：**

* + **场景：** 在查询评论信息时，需要同时获取评论所属文章的信息。

* + **查询语句：**
  + SELECT comments.\*, articles.title  
    FROM comments  
    JOIN articles ON comments.article\_id = articles.article\_id;

在这些查询中，JOIN​ 操作用于将从实体表与主实体表关联起来，通过外键关系将两者连接在一起。这样可以一次性获取主实体的信息，避免了多次查询的开销，并使结果更具有可读性。

在关系数据库中，多对一关系的使用场景通常涉及到需要获取主实体信息的情况，而不仅仅是从实体的信息。这种查询方式有助于避免数据冗余，提高查询效率，同时也更符合关系数据库的设计原则。

‍

‍

‍

查询员工信息以及员工所对应的部门信息

public class Emp {   
 private Integer eid;   
 private String empName;   
 private Integer age;   
 private String sex;   
 private String email;   
 private Dept dept;  
 //...构造器、get、set方法等  
}

‍

* resultMap：设置自定义映射

* + 属性：

* + - id：表示自定义映射的唯一标识，不能重复

* + - type：查询的数据要映射的实体类的类型

* + 子标签：

* + - id：设置主键的映射关系

* + - result：设置普通字段的映射关系

* + - 子标签属性：

* + - * property：设置映射关系中实体类中的属性名

* + - * column：设置映射关系中表中的字段名

‍

‍

### 级联方式处理映射关系（用的少）

<resultMap id="empAndDeptResultMapOne" type="Emp">  
 <id property="eid" column="eid"></id>  
 <result property="empName" column="emp\_name"></result>  
 <result property="age" column="age"></result>  
 <result property="sex" column="sex"></result>  
 <result property="email" column="email"></result>  
 <result property="dept.did" column="did"></result>  
 <result property="dept.deptName" column="dept\_name"></result>  
</resultMap>  
<!--Emp getEmpAndDept(@Param("eid")Integer eid);-->  
<select id="getEmpAndDept" resultMap="empAndDeptResultMapOne">  
 select \* from t\_emp left join t\_dept on t\_emp.eid = t\_dept.did where t\_emp.eid = #{eid}  
</select>

‍

### 使用association处理映射关系

* association：处理多对一的映射关系

* property：需要处理多对的映射关系的属性名

* javaType：该属性的类型

**这里的映射处理实际上就是把表连接起来。通过association标签来连接另一个表，并且设置另一个关联表的信息。**

查询时两张表就是连起来的。

<!-- 定义 Company 的 ResultMap -->  
 <resultMap id="companyResultMap" type="com.hui7.pojo.Company">  
 <id property="companyId" column="company\_id"/>  
 <result property="companyName" column="company\_name"/>  
 </resultMap>  
  
 <!-- 定义 Employee 的 ResultMap，包含关联关系 -->  
 <resultMap id="employeeResultMap" type="com.hui7.pojo.Employee">  
 <id property="employeeId" column="employee\_id"/>  
 <result property="employeeName" column="employee\_name"/>  
 <result property="employeeAge" column="employee\_age"/>  
 <result property="employeeGender" column="employee\_gender"/>  
 <result property="companyId" column="company\_id"/>   
 <result property="companyName" column="company\_name"/>  
  
  
 <!-- 使用 <association> 处理关联关系 -->  
 <association property="company" javaType="com.hui7.pojo.Company" resultMap="companyResultMap">  
 <id property="companyId" column="company\_id"/>  
 <result property="companyName" column="company\_name"/>  
 </association>  
  
  
 </resultMap>  
  
 <!-- 查询语句，使用上述 ResultMap -->  
 <select id="GetEmployeeInfoById" parameterType="java.lang.Integer" resultMap="employeeResultMap">  
 SELECT e.\*, c.\*  
 FROM employee e  
 LEFT JOIN company c ON e.company\_id = c.company\_id  
 WHERE e.employee\_id = #{CompanyId}  
 </select>

‍

// 模糊查询  
 @Test  
 public void GetEmployeeInfoById(){  
 Employee employee= employeeMapper.GetEmployeeInfoById(1);  
 System.out.println("查找到的人:"+employee);  
 }

‍

‍

#### 多表查询

必须要在关联表里放上想要查询另一个表的属性映射到变量里才可以，否则返回的查询数据不会显示关联查询到的信息。

在多的信息表里使用association关联少的信息表

public class Employee {  
 private int employeeId;  
 private String employeeName;  
 private int employeeAge;  
 private String employeeGender;  
 private int companyId;  
  
 private String companyName;  
  
 private Company company; // 添加关联的 Company 对象

‍

‍

### 分步查询

#### 1. 查询员工信息

* select：设置分布查询的sql的唯一标识（namespace.SQLId或mapper接口的全类名.方法名）

* column：设置分步查询的条件

//EmpMapper里的方法  
/\*\*  
 \* 通过分步查询，员工及所对应的部门信息  
 \* 分步查询第一步：查询员工信息  
 \* @param   
 \* @return com.atguigu.mybatis.pojo.Emp  
 \* @date 2022/2/27 20:17  
 \*/  
Emp getEmpAndDeptByStepOne(@Param("eid") Integer eid);

<resultMap id="empAndDeptByStepResultMap" type="Emp">  
 <id property="eid" column="eid"></id>  
 <result property="empName" column="emp\_name"></result>  
 <result property="age" column="age"></result>  
 <result property="sex" column="sex"></result>  
 <result property="email" column="email"></result>  
 <association property="dept"  
 select="com.atguigu.mybatis.mapper.DeptMapper.getEmpAndDeptByStepTwo"  
 column="did"></association>  
</resultMap>  
<!--Emp getEmpAndDeptByStepOne(@Param("eid") Integer eid);-->  
<select id="getEmpAndDeptByStepOne" resultMap="empAndDeptByStepResultMap">  
 select \* from t\_emp where eid = #{eid}  
</select>

#### 2. 查询部门信息

//DeptMapper里的方法  
/\*\*  
 \* 通过分步查询，员工及所对应的部门信息  
 \* 分步查询第二步：通过did查询员工对应的部门信息  
 \* @param  
 \* @return com.atguigu.mybatis.pojo.Emp  
 \* @date 2022/2/27 20:23  
 \*/  
Dept getEmpAndDeptByStepTwo(@Param("did") Integer did);

<!--此处的resultMap仅是处理字段和属性的映射关系-->  
<resultMap id="EmpAndDeptByStepTwoResultMap" type="Dept">  
 <id property="did" column="did"></id>  
 <result property="deptName" column="dept\_name"></result>  
</resultMap>  
<!--Dept getEmpAndDeptByStepTwo(@Param("did") Integer did);-->  
<select id="getEmpAndDeptByStepTwo" resultMap="EmpAndDeptByStepTwoResultMap">  
 select \* from t\_dept where did = #{did}  
</select>

‍

‍

‍

## 一对多映射处理

public class Dept {  
 private Integer did;  
 private String deptName;  
 private List<Emp> emps;  
 //...构造器、get、set方法等  
}

### collection

* collection：用来处理一对多的映射关系

* ofType：表示该属性对饮的集合中存储的数据的类型

<resultMap id="DeptAndEmpResultMap" type="Dept">  
 <id property="did" column="did"></id>  
 <result property="deptName" column="dept\_name"></result>  
 <collection property="emps" ofType="Emp">  
 <id property="eid" column="eid"></id>  
 <result property="empName" column="emp\_name"></result>  
 <result property="age" column="age"></result>  
 <result property="sex" column="sex"></result>  
 <result property="email" column="email"></result>  
 </collection>  
</resultMap>  
<!--Dept getDeptAndEmp(@Param("did") Integer did);-->  
<select id="getDeptAndEmp" resultMap="DeptAndEmpResultMap">  
 select \* from t\_dept left join t\_emp on t\_dept.did = t\_emp.did where t\_dept.did = #{did}  
</select>

### 分步查询

#### 1. 查询部门信息

/\*\*  
 \* 通过分步查询，查询部门及对应的所有员工信息  
 \* 分步查询第一步：查询部门信息  
 \* @param did   
 \* @return com.atguigu.mybatis.pojo.Dept  
 \* @date 2022/2/27 22:04  
 \*/  
Dept getDeptAndEmpByStepOne(@Param("did") Integer did);

<resultMap id="DeptAndEmpByStepOneResultMap" type="Dept">  
 <id property="did" column="did"></id>  
 <result property="deptName" column="dept\_name"></result>  
 <collection property="emps"  
 select="com.atguigu.mybatis.mapper.EmpMapper.getDeptAndEmpByStepTwo"  
 column="did"></collection>  
</resultMap>  
<!--Dept getDeptAndEmpByStepOne(@Param("did") Integer did);-->  
<select id="getDeptAndEmpByStepOne" resultMap="DeptAndEmpByStepOneResultMap">  
 select \* from t\_dept where did = #{did}  
</select>

#### 2. 根据部门id查询部门中的所有员工

/\*\*  
 \* 通过分步查询，查询部门及对应的所有员工信息  
 \* 分步查询第二步：根据部门id查询部门中的所有员工  
 \* @param did  
 \* @return java.util.List<com.atguigu.mybatis.pojo.Emp>  
 \* @date 2022/2/27 22:10  
 \*/  
List<Emp> getDeptAndEmpByStepTwo(@Param("did") Integer did);

<!--List<Emp> getDeptAndEmpByStepTwo(@Param("did") Integer did);-->  
<select id="getDeptAndEmpByStepTwo" resultType="Emp">  
 select \* from t\_emp where did = #{did}  
</select>

## 延迟加载

* **分步查询的优点：可以实现延迟加载，但是必须在核心配置文件中设置全局配置信息**：

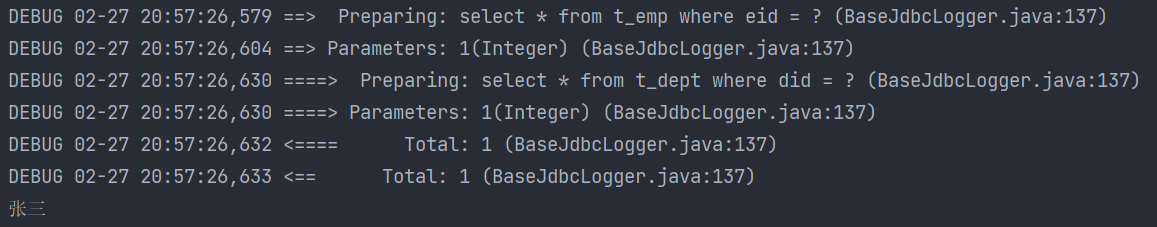
* + lazyLoadingEnabled：延迟加载的全局开关。当开启时，所有关联对象都会延迟加载

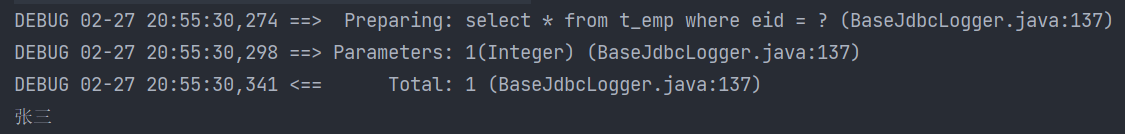
* + aggressiveLazyLoading：当开启时，任何方法的调用都会加载该对象的所有属性。 否则，每个属性会按需加载

* 此时就可以实现按需加载，获取的数据是什么，就只会执行相应的sql。此时可通过association和collection中的fetchType属性设置当前的分步查询是否使用延迟加载，fetchType="lazy(延迟加载)|eager(立即加载)"

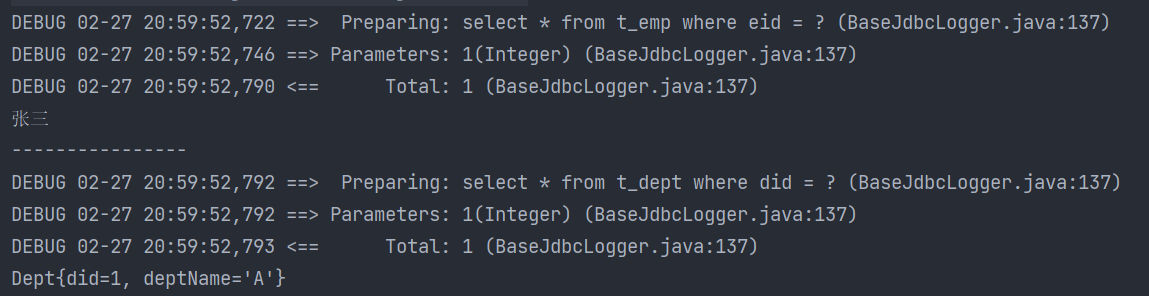
<settings>  
 <!--开启延迟加载-->  
 <setting name="lazyLoadingEnabled" value="true"/>  
</settings>

@Test  
public void getEmpAndDeptByStepOne() {  
 SqlSession sqlSession = SqlSessionUtils.getSqlSession();  
 EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);  
 Emp emp = mapper.getEmpAndDeptByStepOne(1);  
 System.out.println(emp.getEmpName());  
}

* 关闭延迟加载，两条SQL语句都运行了​

* 开启延迟加载，只运行获取emp的SQL语句  
  ​​

@Test  
public void getEmpAndDeptByStepOne() {  
 SqlSession sqlSession = SqlSessionUtils.getSqlSession();  
 EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);  
 Emp emp = mapper.getEmpAndDeptByStepOne(1);  
 System.out.println(emp.getEmpName());  
 System.out.println("----------------");  
 System.out.println(emp.getDept());  
}

* 开启后，需要用到查询dept的时候才会调用相应的SQL语句​

* fetchType：当开启了全局的延迟加载之后，可以通过该属性手动控制延迟加载的效果，fetchType="lazy(延迟加载)|eager(立即加载)"
* <resultMap id="empAndDeptByStepResultMap" type="Emp">  
   <id property="eid" column="eid"></id>  
   <result property="empName" column="emp\_name"></result>  
   <result property="age" column="age"></result>  
   <result property="sex" column="sex"></result>  
   <result property="email" column="email"></result>  
   <association property="dept"  
   select="com.atguigu.mybatis.mapper.DeptMapper.getEmpAndDeptByStepTwo"  
   column="did"  
   fetchType="lazy"></association>  
  </resultMap>

‍

‍

# 动态SQL

* Mybatis框架的动态SQL技术是一种根据特定条件动态拼装SQL语句的功能，它存在的意义是为了解决拼接SQL语句字符串时的痛点问题

* 要求传输过来的参数存在时就要拼接，不存在时就不拼接。

* 拼接条件查询，就是拼接多where查询起来会更精确，例如筛选功能

* 前端传输过来的数据如果没有填写就是null或者""空字符串，当判断是空字符串就不拼接这个查询的参数。

* 实体类传输参数给SQl语句时创建的实体类对象信息如果就是null和"",但参数个数不能少。

‍

## 判断条件动态添加

每个写法之间只是有一点小小的改进而已。

### if 标签

**在动态sql中不转自动将变量驼峰命名转化为数据库下划线命名！！！**

* if标签可通过test属性（即传递过来的数据）的表达式进行判断，若表达式的结果为true，则标签中的内容会执行；反之标签中的内容不会执行

* 在where后面添加一个恒成立条件1=1​​

* + 这个恒成立条件并不会影响查询的结果

* + 这个1=1​​可以用来拼接and​​语句，例如：当empName为null时

* + - 如果不加上恒成立条件，则SQL语句为select \* from t\_emp where and age = ? and sex = ? and email = ?​​，此时where​​会与and​​连用，SQL语句会报错

* + - 如果加上一个恒成立条件，则SQL语句为select \* from t\_emp where 1= 1 and age = ? and sex = ? and email = ?​​，此时不报错

<!-- List<Employee> DynamicSelectEmployee(@Param("employee") Employee employee);-->  
 <select id="DynamicSelectEmployee" parameterType="com.hui7.pojo.Employee" resultType="com.hui7.pojo.Employee">  
 SELECT \* FROM employee WHERE 1=1  
 <if test="employee.employeeName != null and employee.employeeName != ''">  
 AND employee\_name = #{employee.employeeName}  
 </if>  
 <if test="employee.employeeAge != null and employee.employeeAge != ''">  
 AND employee\_age = #{employee.employeeAge}  
 </if>  
 <if test="employee.employeeGender != null and employee.employeeGender != ''">  
 AND employee\_gender = #{employee.employeeGender}  
 </if>  
 <if test="employee.companyId != null and employee.companyId != ''">  
 AND company\_id = #{employee.companyId}  
 </if>  
 </select>

‍

‍

### where 标签

* where和if一般结合使用：

* + 若where标签中的if条件都不满足，则where标签没有任何功能，即不会添加where关键字

* + 若where标签中的if条件满足，则where标签会自动添加where关键字，并将条件最前方多余的and/or去掉

<select id="DynamicSelectEmployee" parameterType="com.hui7.pojo.Employee" resultType="com.hui7.pojo.Employee">  
 SELECT \* FROM employee  
 <where>  
 <if test="employee.employeeName != null and employee.employeeName != ''">  
 AND employee\_name = #{employee.employeeName}  
 </if>  
 <if test="employee.employeeAge != null and employee.employeeAge != ''">  
 AND employee\_age = #{employee.employeeAge}  
 </if>  
 <if test="employee.employeeGender != null and employee.employeeGender != ''">  
 AND employee\_gender = #{employee.employeeGender}  
 </if>  
 <if test="employee.companyId != null and employee.companyId != ''">  
 AND company\_id = #{employee.companyId}  
 </if>  
 </where>  
 </select>

* 注意：where标签不能去掉条件后多余的and/or
* <!--这种用法是错误的，只能去掉条件前面的and/or，条件后面的不行-->  
  <if test="empName != null and empName !=''">  
  emp\_name = #{empName} and  
  </if>  
  <if test="age != null and age !=''">  
   age = #{age}  
  </if>

‍

‍

‍

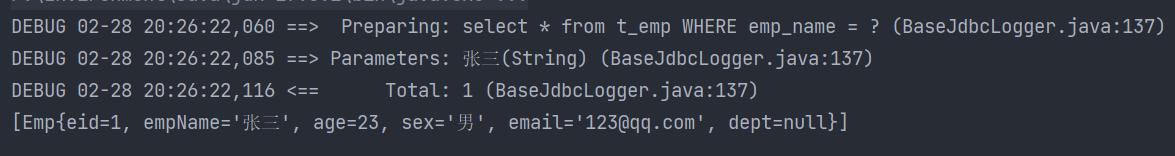
### choose、when、otherwise 标签

* ​choose、when、otherwise​相当于switch case default​

* when至少要有一个，otherwise至多只有一个

<select id="getEmpByChoose" resultType="Emp">  
 select \* from t\_emp  
 <where>  
 <choose>  
 <when test="empName != null and empName != ''">  
 emp\_name = #{empName}  
 </when>  
 <when test="age != null and age != ''">  
 age = #{age}  
 </when>  
 <when test="sex != null and sex != ''">  
 sex = #{sex}  
 </when>  
 <when test="email != null and email != ''">  
 email = #{email}  
 </when>  
 <otherwise>  
 did = 1  
 </otherwise>  
 </choose>  
 </where>  
</select>

@Test  
public void getEmpByChoose() {  
 SqlSession sqlSession = SqlSessionUtils.getSqlSession();  
 DynamicSQLMapper mapper = sqlSession.getMapper(DynamicSQLMapper.class);  
 List<Emp> emps = mapper.getEmpByChoose(new Emp(null, "张三", 23, "男", "123@qq.com", null));  
 System.out.println(emps);  
}

​​

‍

‍

‍

## trim 标签

* trim用于去掉或添加标签中的内容

* 常用属性

* + prefix：在trim标签中的内容的前面添加某些内容

* + suffix：在trim标签中的内容的后面添加某些内容

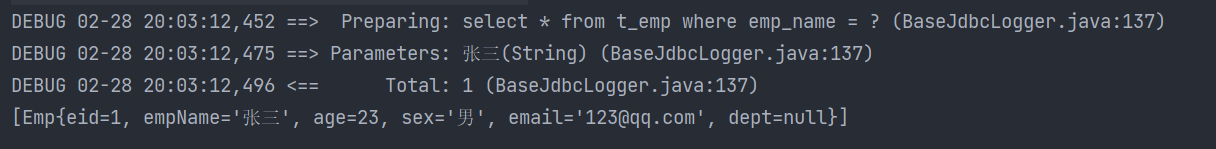
* + prefixOverrides：在trim标签中的内容的前面去掉某些内容

* + suffixOverrides：在trim标签中的内容的后面去掉某些内容

* 若trim中的标签都不满足条件，则trim标签没有任何效果，也就是只剩下select \* from t\_emp​

<select id="DynamicSelectEmployee" parameterType="com.hui7.pojo.Employee" resultType="com.hui7.pojo.Employee">  
 SELECT \* FROM employee  
 <trim prefix="where" prefixOverrides="and|or">  
 <if test="employee.employeeName != null and employee.employeeName != ''">  
 AND employee\_name = #{employee.employeeName}  
 </if>  
 <if test="employee.employeeAge != null and employee.employeeAge != ''">  
 AND employee\_age = #{employee.employeeAge}  
 </if>  
 <if test="employee.employeeGender != null and employee.employeeGender != ''">  
 AND employee\_gender = #{employee.employeeGender}  
 </if>  
 <if test="employee.companyId != null and employee.companyId != ''">  
 AND company\_id = #{employee.companyId}  
 </if>  
 </trim>  
 </select>

//测试类  
@Test  
public void getEmpByCondition() {  
 SqlSession sqlSession = SqlSessionUtils.getSqlSession();  
 DynamicSQLMapper mapper = sqlSession.getMapper(DynamicSQLMapper.class);  
 List<Emp> emps= mapper.getEmpByCondition(new Emp(null, "张三", null, null, null, null));  
 System.out.println(emps);  
}

​​

‍

## foreach 循环标签

应用场景：批量添加和批量删除

* 属性：

* + collection：设置要循环的数组或集合

* + item：表示集合或数组中的每一个数据

* + separator：设置循环体之间的分隔符，分隔符前后默认有一个空格，如,​ 也可以设置成 or and ，设计的是条件之间的拼接也可以
* 在每次循环的最外部添加

* + open：设置foreach标签中的内容的开始符

* + close：设置foreach标签中的内容的结束符

‍

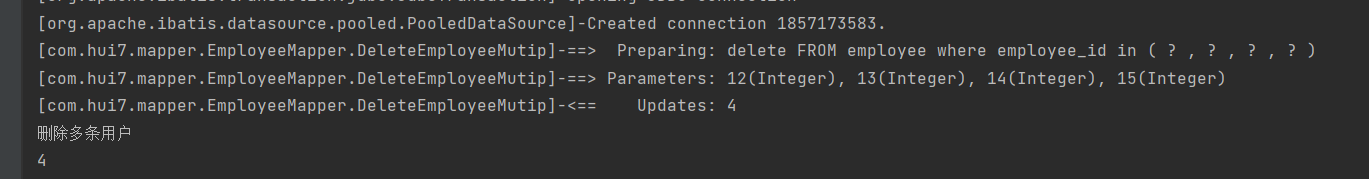
‍

### 批量删除

设置open和close之后回拼接上括号。  
​delete FROM employee where employee\_id in ( ? , ? , ? , ? )​

// int DeleteEmployeeMutip(@Param("ids") Integer[] ids);   
  
<delete id="DeleteEmployeeMutip">  
 delete FROM employee where employee\_id in  
 <foreach collection="ids" item="ids" separator="," open="(" close=")">  
 #{ids}  
 </foreach>  
 </delete>

//批量删除  
 @Test  
 public void DeleteEmployeeMutip(){  
 int employee = employeeMapper.DeleteEmployeeMutip(new Integer[]{12, 13, 14, 15});  
 System.out.println("删除多条用户");  
 System.out.println(employee);  
 }

​​

‍

‍

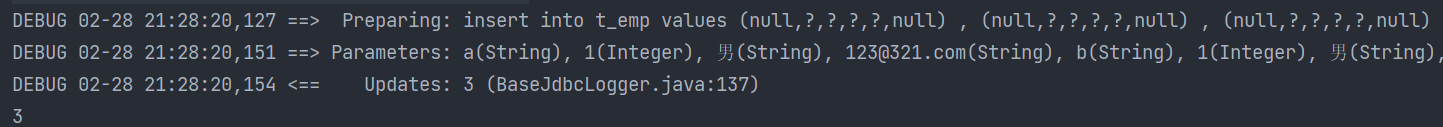
‍

‍

### 批量添加

<!--int insertMoreByList(@Param("emps") List<Emp> emps);-->  
<insert id="insertMoreByList">  
 insert into t\_emp values  
 <foreach collection="emps" item="emp" separator=",">  
 (null,#{emp.empName},#{emp.age},#{emp.sex},#{emp.email},null)  
 </foreach>  
</insert>

@Test  
public void insertMoreByList() {  
 SqlSession sqlSession = SqlSessionUtils.getSqlSession();  
 DynamicSQLMapper mapper = sqlSession.getMapper(DynamicSQLMapper.class);  
 Emp emp1 = new Emp(null,"a",1,"男","123@321.com",null);  
 Emp emp2 = new Emp(null,"b",1,"男","123@321.com",null);  
 Emp emp3 = new Emp(null,"c",1,"男","123@321.com",null);  
 List<Emp> emps = Arrays.asList(emp1, emp2, emp3);  
 int result = mapper.insertMoreByList(emps);  
 System.out.println(result);  
}

​​

‍

‍

‍

## SQL片段

* sql片段，可以记录一段公共sql片段，在使用的地方通过include标签进行引入

* 声明sql片段：<sql>​标签

<sql id="empColumns">eid,emp\_name,age,sex,email</sql>

* 引用sql片段：<include>​标签

<!--List<Emp> getEmpByCondition(Emp emp);-->  
<select id="getEmpByCondition" resultType="Emp">  
 select <include refid="empColumns"></include> from t\_emp  
</select>

‍

‍

‍

# MyBatis的缓存

## MyBatis的一级缓存

* 一级缓存是SqlSession级别的，通过同一个SqlSession查询的数据会被缓存，下次查询相同的数据，就会从缓存中直接获取，不会从数据库重新访问。

* 使一级缓存失效的四种情况：

* 1. 不同的SqlSession对应不同的一级缓存

* 1. 同一个SqlSession但是查询条件不同

* 1. 同一个SqlSession两次查询期间执行了任何一次增删改操作

* 1. 同一个SqlSession两次查询期间手动清空了缓存

## MyBatis的二级缓存

* 二级缓存是SqlSessionFactory级别，通过同一个SqlSessionFactory创建的SqlSession查询的结果会被缓存；此后若再次执行相同的查询语句，结果就会从缓存中获取

* 二级缓存开启的条件

* 1. 在核心配置文件中，设置全局配置属性cacheEnabled="true"，默认为true，不需要设置

* 1. 在映射文件中设置标签<cache />

* 1. 二级缓存必须在SqlSession关闭或提交之后有效

* 1. 查询的数据所转换的实体类类型必须实现序列化的接口

* 使二级缓存失效的情况：两次查询之间执行了任意的增删改，会使一级和二级缓存同时失效

‍

手动清楚缓存（只能一级的缓存）

sqlSession1.clearcache();

‍

## 二级缓存的相关配置

* 在mapper配置文件中添加的cache标签可以设置一些属性

* eviction属性：缓存回收策略

* + LRU（Least Recently Used） – 最近最少使用的：移除最长时间不被使用的对象。

* + FIFO（First in First out） – 先进先出：按对象进入缓存的顺序来移除它们。

* + SOFT – 软引用：移除基于垃圾回收器状态和软引用规则的对象。

* + WEAK – 弱引用：更积极地移除基于垃圾收集器状态和弱引用规则的对象。

* + 默认的是 LRU

* flushInterval属性：刷新间隔，单位毫秒

* + 默认情况是不设置，也就是没有刷新间隔，缓存仅仅调用语句（增删改）时刷新

* size属性：引用数目，正整数

* + 代表缓存最多可以存储多少个对象，太大容易导致内存溢出

* readOnly属性：只读，true/false

* + true：只读缓存；会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这提供了很重要的性能优势。

* + false：读写缓存；会返回缓存对象的拷贝（通过序列化）。这会慢一些，但是安全，因此默认是false

## MyBatis缓存查询的顺序

* 先查询二级缓存，因为二级缓存中可能会有其他程序已经查出来的数据，可以拿来直接使用

* 如果二级缓存没有命中，再查询一级缓存

* 如果一级缓存也没有命中，则查询数据库

* SqlSession关闭之后，一级缓存中的数据会写入二级缓存

## 整合第三方缓存EHCache（了解）

### 添加依赖

<!-- Mybatis EHCache整合包 -->  
<dependency>  
 <groupId>org.mybatis.caches</groupId>  
 <artifactId>mybatis-ehcache</artifactId>  
 <version>1.2.1</version>  
</dependency>  
<!-- slf4j日志门面的一个具体实现 -->  
<dependency>  
 <groupId>ch.qos.logback</groupId>  
 <artifactId>logback-classic</artifactId>  
 <version>1.2.3</version>  
</dependency>

### 各个jar包的功能

| jar包名称 | 作用 |
| --- | --- |
| mybatis-ehcache | Mybatis和EHCache的整合包 |
| ehcache | EHCache核心包 |
| slf4j-api | SLF4J日志门面包 |
| logback-classic | 支持SLF4J门面接口的一个具体实现 |

### 创建EHCache的配置文件ehcache.xml

* 名字必须叫ehcache.xml​

<?xml version="1.0" encoding="utf-8" ?>  
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">  
 <!-- 磁盘保存路径 -->  
 <diskStore path="D:\atguigu\ehcache"/>  
 <defaultCache  
 maxElementsInMemory="1000"  
 maxElementsOnDisk="10000000"  
 eternal="false"  
 overflowToDisk="true"  
 timeToIdleSeconds="120"  
 timeToLiveSeconds="120"  
 diskExpiryThreadIntervalSeconds="120"  
 memoryStoreEvictionPolicy="LRU">  
 </defaultCache>  
</ehcache>

### 设置二级缓存的类型

* 在xxxMapper.xml文件中设置二级缓存类型

<cache type="org.mybatis.caches.ehcache.EhcacheCache"/>

### 加入logback日志

* 存在SLF4J时，作为简易日志的log4j将失效，此时我们需要借助SLF4J的具体实现logback来打印日志。创建logback的配置文件logback.xml​，名字固定，不可改变

<?xml version="1.0" encoding="UTF-8"?>  
<configuration debug="true">  
 <!-- 指定日志输出的位置 -->  
 <appender name="STDOUT"  
 class="ch.qos.logback.core.ConsoleAppender">  
 <encoder>  
 <!-- 日志输出的格式 -->  
 <!-- 按照顺序分别是：时间、日志级别、线程名称、打印日志的类、日志主体内容、换行 -->  
 <pattern>[%d{HH:mm:ss.SSS}] [%-5level] [%thread] [%logger] [%msg]%n</pattern>  
 </encoder>  
 </appender>  
 <!-- 设置全局日志级别。日志级别按顺序分别是：DEBUG、INFO、WARN、ERROR -->  
 <!-- 指定任何一个日志级别都只打印当前级别和后面级别的日志。 -->  
 <root level="DEBUG">  
 <!-- 指定打印日志的appender，这里通过“STDOUT”引用了前面配置的appender -->  
 <appender-ref ref="STDOUT" />  
 </root>  
 <!-- 根据特殊需求指定局部日志级别 -->  
 <logger name="com.atguigu.crowd.mapper" level="DEBUG"/>  
</configuration>

### EHCache配置文件说明

| 属性名 | 是否必须 | 作用 |
| --- | --- | --- |
| maxElementsInMemory | 是 | 在内存中缓存的element的最大数目 |
| maxElementsOnDisk | 是 | 在磁盘上缓存的element的最大数目，若是0表示无穷大 |
| eternal | 是 | 设定缓存的elements是否永远不过期。 如果为true，则缓存的数据始终有效， 如果为false那么还要根据timeToIdleSeconds、timeToLiveSeconds判断 |
| overflowToDisk | 是 | 设定当内存缓存溢出的时候是否将过期的element缓存到磁盘上 |
| timeToIdleSeconds | 否 | 当缓存在EhCache中的数据前后两次访问的时间超过timeToIdleSeconds的属性取值时， 这些数据便会删除，默认值是0,也就是可闲置时间无穷大 |
| timeToLiveSeconds | 否 | 缓存element的有效生命期，默认是0.,也就是element存活时间无穷大 |
| diskSpoolBufferSizeMB | 否 | DiskStore(磁盘缓存)的缓存区大小。默认是30MB。每个Cache都应该有自己的一个缓冲区 |
| diskPersistent | 否 | 在VM重启的时候是否启用磁盘保存EhCache中的数据，默认是false |
| diskExpiryThreadIntervalSeconds | 否 | 磁盘缓存的清理线程运行间隔，默认是120秒。每个120s， 相应的线程会进行一次EhCache中数据的清理工作 |
| memoryStoreEvictionPolicy | 否 | 当内存缓存达到最大，有新的element加入的时候， 移除缓存中element的策略。 默认是LRU（最近最少使用），可选的有LFU（最不常使用）和FIFO（先进先出 |

‍

‍

‍

# MyBatis的逆向工程

* 正向工程：先创建Java实体类，由框架负责根据实体类生成数据库表。Hibernate是支持正向工程的

* 逆向工程：先创建数据库表，由框架负责根据数据库表，反向生成如下资源：

* + Java实体类

* + Mapper接口

* + Mapper映射文件

## 创建逆向工程的步骤

### 添加依赖和插件

<dependencies>  
 <!-- MyBatis核心依赖包 -->  
 <dependency>  
 <groupId>org.mybatis</groupId>  
 <artifactId>mybatis</artifactId>  
 <version>3.5.9</version>  
 </dependency>  
 <!-- junit测试 -->  
 <dependency>  
 <groupId>junit</groupId>  
 <artifactId>junit</artifactId>  
 <version>4.13.2</version>  
 <scope>test</scope>  
 </dependency>  
 <!-- MySQL驱动 -->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>8.0.27</version>  
 </dependency>  
 <!-- log4j日志 -->  
 <dependency>  
 <groupId>log4j</groupId>  
 <artifactId>log4j</artifactId>  
 <version>1.2.17</version>  
 </dependency>  
</dependencies>  
<!-- 控制Maven在构建过程中相关配置 -->  
<build>  
 <!-- 构建过程中用到的插件 -->  
 <plugins>  
 <!-- 具体插件，逆向工程的操作是以构建过程中插件形式出现的 -->  
 <plugin>  
 <groupId>org.mybatis.generator</groupId>  
 <artifactId>mybatis-generator-maven-plugin</artifactId>  
 <version>1.3.0</version>  
 <!-- 插件的依赖 -->  
 <dependencies>  
 <!-- 逆向工程的核心依赖 -->  
 <dependency>  
 <groupId>org.mybatis.generator</groupId>  
 <artifactId>mybatis-generator-core</artifactId>  
 <version>1.3.2</version>  
 </dependency>  
 <!-- 数据库连接池 -->  
 <dependency>  
 <groupId>com.mchange</groupId>  
 <artifactId>c3p0</artifactId>  
 <version>0.9.2</version>  
 </dependency>  
 <!-- MySQL驱动 -->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>8.0.27</version>  
 </dependency>  
 </dependencies>  
 </plugin>  
 </plugins>  
</build>

### 创建MyBatis的核心配置文件

<?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>  
 <properties resource="jdbc.properties"/>  
 <typeAliases>  
 <package name=""/>  
 </typeAliases>  
 <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>  
 <mappers>  
 <package name=""/>  
 </mappers>  
</configuration>

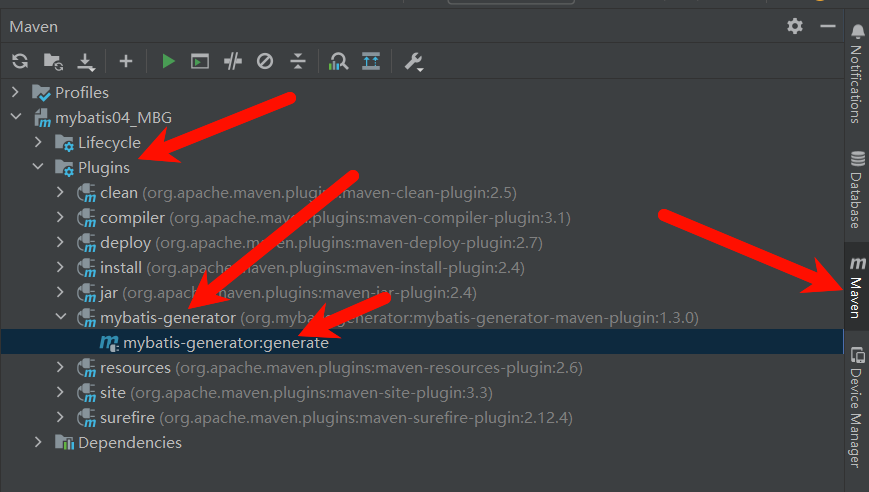
### 创建逆向工程的配置文件

* 文件名必须是：generatorConfig.xml​​

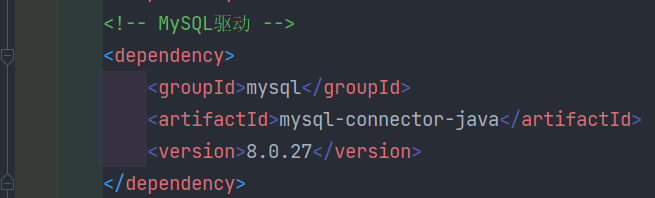
* 根据自己的数据库来配置，以及要创建带本地类的名字包名等等
* ​targetRuntime="MyBatis3Simple"​**配置的是生成文件中查询语句以及实体类的丰富度。一般使用MyBatis3（添加了很多普安段文件，容错率大大提高。）**

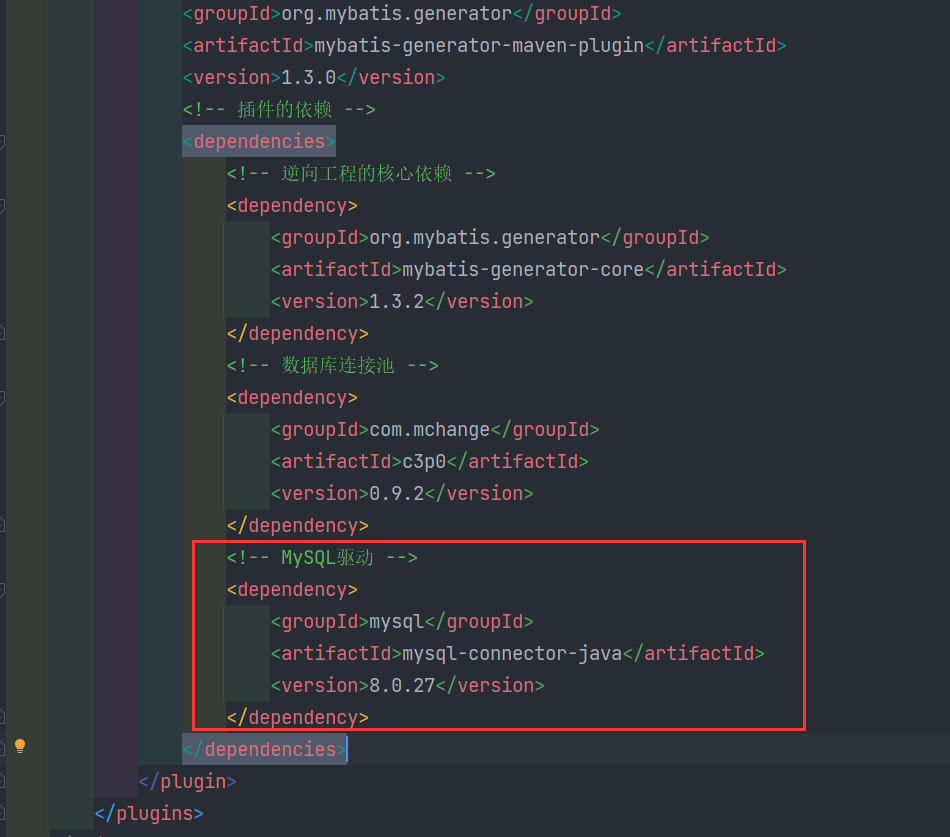
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE generatorConfiguration  
 PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"  
 "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd">  
<generatorConfiguration>  
 <!--  
 targetRuntime: 执行生成的逆向工程的版本  
 MyBatis3Simple: 生成基本的CRUD（清新简洁版）  
 MyBatis3: 生成带条件的CRUD（奢华尊享版）  
 -->  
 <context id="DB2Tables" targetRuntime="MyBatis3Simple">  
 <!-- 数据库的连接信息 -->  
 <jdbcConnection driverClass="com.mysql.cj.jdbc.Driver"  
 connectionURL="jdbc:mysql://localhost:3306/mybatis"  
 userId="beidao23"  
 password="beidaomitu233">  
 </jdbcConnection>  
 <!-- javaBean的生成策略-->  
 <javaModelGenerator targetPackage="com.hui7.mybatis.pojo" targetProject=".\src\main\java">  
 <property name="enableSubPackages" value="true" />  
 <property name="trimStrings" value="true" />  
 </javaModelGenerator>  
 <!-- SQL映射文件的生成策略 -->  
 <sqlMapGenerator targetPackage="com.hui7.mybatis.mapper"  
 targetProject=".\src\main\resources">  
 <property name="enableSubPackages" value="true" />  
 </sqlMapGenerator>  
 <!-- Mapper接口的生成策略 -->  
 <javaClientGenerator type="XMLMAPPER"  
 targetPackage="com.hui7.mybatis.mapper" targetProject=".\src\main\java">  
 <property name="enableSubPackages" value="true" />  
 </javaClientGenerator>  
 <!-- 逆向分析的表 -->  
 <!-- tableName设置为\*号，可以对应所有表，此时不写domainObjectName -->  
 <!-- domainObjectName属性指定生成出来的实体类的类名 -->  
 <table tableName="employee" domainObjectName="Employee"/>  
 <table tableName="company" domainObjectName="Company"/>  
 </context>  
</generatorConfiguration>

### 执行MBG插件的generate目标

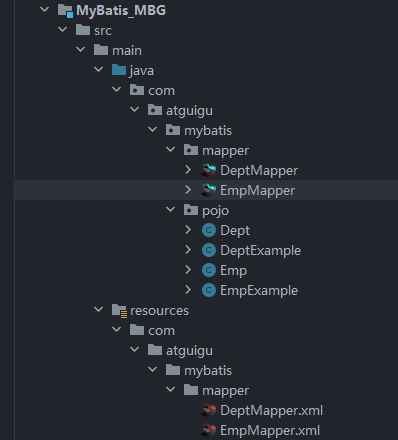
* ​​​​

* 如果出现报错：Exception getting JDBC Driver​，可能是pom.xml中，数据库驱动配置错误

* + dependency中的驱动​

* + mybatis-generator-maven-plugin插件中的驱动​

* + 两者的驱动版本应该相同

* 执行结果​

‍

‍

‍

‍

## QBC语法

### 查询

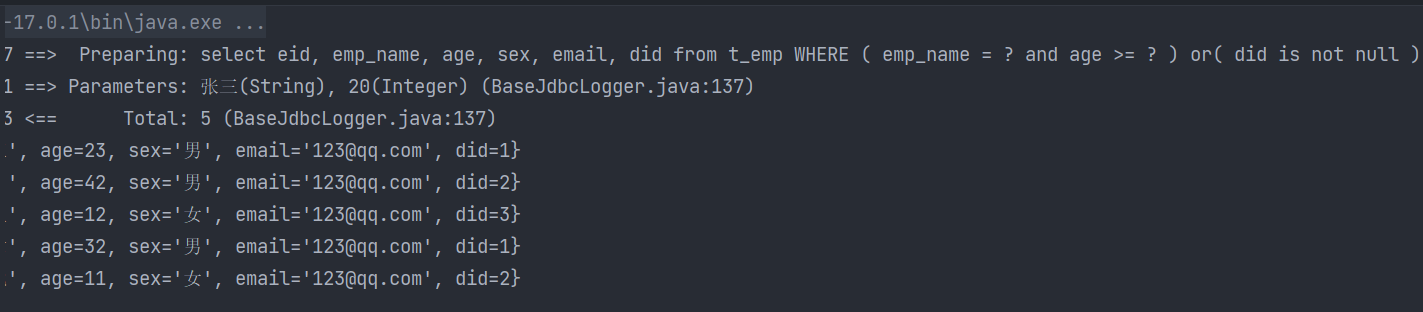
* ​selectByExample​​：按条件查询，需要传入一个example对象或者null；如果传入一个null，则表示没有条件，也就是查询所有数据

* ​example.createCriteria().xxx​​：创建条件对象，通过andXXX方法为SQL添加查询添加，每个条件之间是and关系

* ​example.or().xxx​​：将之前添加的条件通过or拼接其他条件

* **支持链式调用**  
  ​​
* **支持链式调用**

@Test public void testMBG() throws IOException {  
 InputStream is = Resources.getResourceAsStream("mybatis-config.xml");  
 SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  
 SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(is);  
 SqlSession sqlSession = sqlSessionFactory.openSession(true);  
 EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);  
 EmpExample example = new EmpExample();  
 //名字为张三，且年龄大于等于20  
 example.createCriteria().andEmpNameEqualTo("张三").andAgeGreaterThanOrEqualTo(20);  
 //或者did不为空  
 example.or().andDidIsNotNull();  
 List<Emp> emps = mapper.selectByExample(example);  
 emps.forEach(System.out::println);  
}

​​

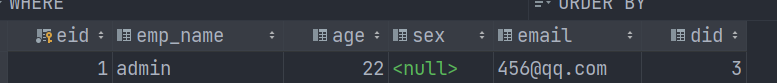
‍

‍

### 增改

* ​updateByPrimaryKey​：通过主键进行数据修改，如果某一个值为null，也会将对应的字段改为null

* + ​mapper.updateByPrimaryKey(new Emp(1,"admin",22,null,"456@qq.com",3));​

* + ​​

* ​updateByPrimaryKeySelective()​：通过主键进行选择性数据修改，如果某个值为null，则不修改这个字段

* + ​mapper.updateByPrimaryKeySelective(new Emp(2,"admin2",22,null,"456@qq.com",3));​

* + ​​

‍

‍

‍

# 分页插件

## 分页插件使用步骤

### 添加依赖

<!-- https://mvnrepository.com/artifact/com.github.pagehelper/pagehelper -->  
<dependency>  
 <groupId>com.github.pagehelper</groupId>  
 <artifactId>pagehelper</artifactId>  
 <version>5.2.0</version>  
</dependency>

### 配置分页插件

* 在MyBatis的核心配置文件（mybatis-config.xml）中配置插件

* ​​

<plugins>  
 <!--设置分页插件-->  
 <plugin interceptor="com.github.pagehelper.PageInterceptor"></plugin>  
</plugins>

## 分页插件的使用

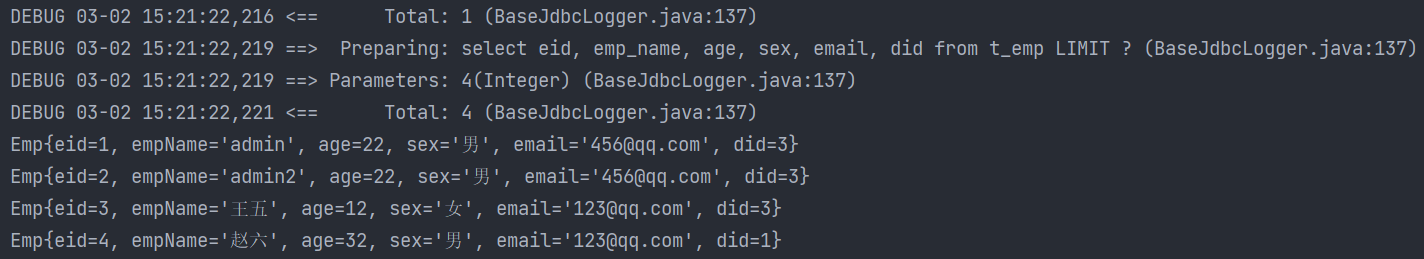
### 开启分页功能

* 在查询功能之前使用PageHelper.startPage(int pageNum, int pageSize)​开启分页功能

* + pageNum：当前页的页码

* + pageSize：每页显示的条数

@Test  
public void testPageHelper() throws IOException {  
 InputStream is = Resources.getResourceAsStream("mybatis-config.xml");  
 SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  
 SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(is);  
 SqlSession sqlSession = sqlSessionFactory.openSession(true);  
 EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);  
 //访问第一页，每页四条数据  
 PageHelper.startPage(1,4);  
 List<Emp> emps = mapper.selectByExample(null);  
 emps.forEach(System.out::println);  
}

​​

### 分页相关数据

#### 方法一：直接输出

@Test  
public void testPageHelper() throws IOException {  
 InputStream is = Resources.getResourceAsStream("mybatis-config.xml");  
 SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  
 SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(is);  
 SqlSession sqlSession = sqlSessionFactory.openSession(true);  
 EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);  
 //访问第一页，每页四条数据  
 Page<Object> page = PageHelper.startPage(1, 4);  
 List<Emp> emps = mapper.selectByExample(null);  
 //在查询到List集合后，打印分页数据  
 System.out.println(page);  
}

* 分页相关数据：
* Page{count=true, pageNum=1, pageSize=4, startRow=0, endRow=4, total=8, pages=2, reasonable=false, pageSizeZero=false}[Emp{eid=1, empName='admin', age=22, sex='男', email='456@qq.com', did=3}, Emp{eid=2, empName='admin2', age=22, sex='男', email='456@qq.com', did=3}, Emp{eid=3, empName='王五', age=12, sex='女', email='123@qq.com', did=3}, Emp{eid=4, empName='赵六', age=32, sex='男', email='123@qq.com', did=1}]

#### 方法二使用PageInfo

* 在查询获取list集合之后，使用PageInfo<T> pageInfo = new PageInfo<>(List<T> list, intnavigatePages)​获取分页相关数据

* + list：分页之后的数据

* + navigatePages：导航分页的页码数

@Test  
public void testPageHelper() throws IOException {  
 InputStream is = Resources.getResourceAsStream("mybatis-config.xml");  
 SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new SqlSessionFactoryBuilder();  
 SqlSessionFactory sqlSessionFactory = sqlSessionFactoryBuilder.build(is);  
 SqlSession sqlSession = sqlSessionFactory.openSession(true);  
 EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);  
 PageHelper.startPage(1, 4);  
 List<Emp> emps = mapper.selectByExample(null);  
 PageInfo<Emp> page = new PageInfo<>(emps,5);  
 System.out.println(page);  
}

* 分页相关数据：
* PageInfo{  
  pageNum=1, pageSize=4, size=4, startRow=1, endRow=4, total=8, pages=2,   
  list=Page{count=true, pageNum=1, pageSize=4, startRow=0, endRow=4, total=8, pages=2, reasonable=false, pageSizeZero=false}[Emp{eid=1, empName='admin', age=22, sex='男', email='456@qq.com', did=3}, Emp{eid=2, empName='admin2', age=22, sex='男', email='456@qq.com', did=3}, Emp{eid=3, empName='王五', age=12, sex='女', email='123@qq.com', did=3}, Emp{eid=4, empName='赵六', age=32, sex='男', email='123@qq.com', did=1}],   
  prePage=0, nextPage=2, isFirstPage=true, isLastPage=false, hasPreviousPage=false, hasNextPage=true, navigatePages=5, navigateFirstPage=1, navigateLastPage=2, navigatepageNums=[1, 2]}

* 其中list中的数据等同于方法一中直接输出的page数据

#### 常用数据：

* pageNum：当前页的页码

* pageSize：每页显示的条数

* size：当前页显示的真实条数

* total：总记录数

* pages：总页数

* prePage：上一页的页码

* nextPage：下一页的页码

* isFirstPage/isLastPage：是否为第一页/最后一页

* hasPreviousPage/hasNextPage：是否存在上一页/下一页

* navigatePages：导航分页的页码数

* navigatepageNums：导航分页的页码，[1,2,3,4,5]

‍