MyBatis学习

1.MyBatis简介

1. MyBatis历史

   - MyBatis最初是Apache的一个开源项目iBatis, 2010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 Code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 java 的持久层框架。iBatis 提供的持久层框架包括 SQL Maps（数据库数据与 Java 数据的映射关系）和 Data Access Object（DAO）

2. 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**持久层框架**
   2. MyBatis 避免了几乎所有的 JDBC 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 Object，普通的 java 对象）映射成数据库中的记录
   4. MyBatis 是一个 半自动的 ORM（Object Relation Mapping，对象关系映射）框架

3. 和其他持久层技术的比较

   - JDBC
     - SQL 夹杂在 Java 语句中耦合度高，导致硬编码内伤
     - 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
   - Hibernate 和 JPA
     - 操作简洁，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，不容易做特殊优化
     - 基于全映射的全自动框架，大量字段的 POJO 进行部分映射比较困难

        - 反射操作过多，导致数据库性能下降
    - 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
        - 开发效率稍逊 HIbernate，但是完全能接受

2.搭建 MyBatis

1. 引入依赖

```
    <?xml version="1.0" encoding="UTF-8"?>
    <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/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>

        <groupId>org.example</groupId>
        <artifactId>MyBatis_test</artifactId>
        <version>1.0-SNAPSHOT</version>
        <packaging>pom</packaging>
        <modules>
            <module>MyBatis_demo1</module>
        </modules>
        <dependencies>
            <!-- mybatis驱动 -->
            <dependency>
                <groupId>org.mybatis</groupId>
                <artifactId>mybatis</artifactId>
                <version>3.5.11</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>8.0.30</version>
            </dependency>
            <dependency>
                <groupId>org.apache.logging.log4j</groupId>
                <artifactId>log4j-core</artifactId>
                <version>2.19.0</version>
            </dependency>
            <dependency>
                <groupId>org.springframework</groupId>
```

```
                <artifactId>spring-test</artifactId>
                <version>6.0.3</version>
            </dependency>
            <dependency>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>1.18.24</version>
                <scope>provided</scope>
            </dependency>
        </dependencies>

        <properties>
            <maven.compiler.source>8</maven.compiler.source>
            <maven.compiler.target>8</maven.compiler.target>
            <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
        </properties>


    </project>
```

2. 配置核心配置文件

> 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，所以大家操作时可以直接复制、粘贴。 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

```
    <?xml version="1.0" encoding="UTF-8" ?>
    <!DOCTYPE configuration
            PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
            "http://mybatis.org/dtd/mybatis-3-config.dtd">
    <configuration>
        <environments default="development">
            <environment id="development">
                <transactionManager type="JDBC"/>
                <dataSource type="POOLED">
                    <property name="driver" value="com.mysql.jdbc.Driver"/>
                    <property name="url"
value="jdbc:mysql://localhost:3306/MyBatis"/>
                    <property name="username" value="root"/>
                    <property name="password" value="123456"/>
                </dataSource>
            </environment>
        </environments>
        <mappers>
            <mapper resource="org/mybatis/example/BlogMapper.xml"/>
        </mappers>
    </configuration>
```

3. 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

```
public interface UserMapper {
    /*
    * MyBatis面向接口编程的两个一致
    * 1. 映射文件的namespace要和mapper接口的全类名保持一致
    * 2. 映射文件中SQL语句的id要和mapper接口中的方法名一致
    * */
    int insertUser();
}
```

4. 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

   > 类对应表，属性对应字段/列
   > 属性对应字段/列
   > 对象对应记录/行

> 1. 映射文件的命名规则: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
> 2. MyBatis中可以面向接口操作数据，要保证两个一致： a>mapper接口的全类名和映射文件的命名空间(namespace)保持一致,b>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

5. 通过 junit 测试功能

   - SqlSession：代表 Java 程序和数据库之间的会话。
   - 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，那么我们可以把创建的这个对象的相关代码封装到一个工厂类，以后都可以使用这个工厂类来生产我们需要的对象

```
public class MyBatisTest {
    @Test
    public void testMyBatis() throws Exception {
        //加载核心配置文件
        InputStream is = Resources.getResourceAsStream("mybatis-
config.xml");
        //获取SqlSessionFactoryBuilder
```

```
        SqlSessionFactoryBuilder sqlSessionFactoryBuilder = new
SqlSessionFactoryBuilder();
        //获取SqlSessionFactory
        SqlSessionFactory sqlSessionFactory =
sqlSessionFactoryBuilder.build(is);
        //获取SqlSession
        SqlSession sqlSession = sqlSessionFactory.openSession();
        //获取mapper接口对象
        UserMapper mapper = sqlSession.getMapper(UserMapper.class);
        //测试功能
        int result = mapper.insertUser();
        //提交事务
        sqlSession.commit();
        System.out.println("result:" + result);
    }
}
```

- 注意：此时需要手动提交，如果自动提交事务，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

6. 加入 log4j 日志功能

    1. 加入依赖

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

    2. 加入log4j的配置文件

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <appender name="STDOUT" class="org.apache.log4j.ConsoleAppender">
        <param name="Encoding" value="UTF-8" />
        <layout class="org.apache.log4j.PatternLayout">
            <param name="ConversionPattern" value="%-5p %d{MM-dd
HH:mm:ss,SSS} %m (%F:%L) \n" />
        </layout>
    </appender>
    <logger name="java.sql">
        <level value="debug" />
    </logger>
    <logger name="org.apache.ibatis">
        <level value="info" />
```

```
    </logger>
    <root>
        <level value="debug" />
        <appender-ref ref="STDOUT" />
    </root>
</log4j:configuration>
```

> 日志的级别
> FATAL:致命;ERROR(错误);WARN(警告);INFO(信息);DEBUG(调试)(从左到右打印的信息越来越详细)

3.核心配置文件详解

> 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

```
<?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>
        <!--
        typeAlias：设置某个具体的类型的别名
        属性：
        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>-->
        <!--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：设置多个连接数据库的环境
    属性：
        default：设置默认使用的环境的id
    -->
    <environments default="mysql_test">
        <!--
```

```
            environment：设置具体的连接数据库的环境信息
            属性：
                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">
                <!--
                transactionManager：设置事务管理方式
                属性：
                    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"/>
                <!--
                dataSource：设置数据源
                属性：
                    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">
                    <!--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>
```

> 注意：jdbc:mysql://localhost:3306/mybatis?serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=utf8在properties文件中写为:url=jdbc:mysql://localhost:3306/mybatis ?serverTimezone=UTC&useSSL=false&useUnicode=true&characterEncoding=utf8

4.用MyBatis进行CRUD测试

```
import com.mapper.UserMapper;
import com.pojo.User;
import com.utils.SqlSessionFactoryUtil;
import org.apache.ibatis.session.SqlSession;
import org.junit.Test;

import java.util.List;

public class UserTest {
    @Test
    public void testInsert() {
        //从工具方法中获取 SqlSession
        SqlSession sqlSession = SqlSessionFactoryUtil.getSqlSession();
        //获取接口对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        //测试功能
        int result = userMapper.insertUser();
        sqlSession.commit();
        System.out.println("result:"+result);
        sqlSession.close();
    }
    @Test
    public void testUpdate(){
        //从工具方法中获取 SqlSession
        SqlSession sqlSession = SqlSessionFactoryUtil.getSqlSession();
        //获取接口对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        //测试功能
        userMapper.updateUser();
        sqlSession.commit();
        sqlSession.close();
    }
    @Test
    public void testDelete(){
        //从工具方法中获取 SqlSession
        SqlSession sqlSession = SqlSessionFactoryUtil.getSqlSession();
        //获取接口对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        //测试功能
        userMapper.deleteUser();
        sqlSession.commit();
        sqlSession.close();
    }
    @Test
    public void testSelect(){
        //从工具方法中获取 SqlSession
        SqlSession sqlSession = SqlSessionFactoryUtil.getSqlSession();
        //获取接口对象
        UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
        //测试功能
```

```
            User user = userMapper.selectUserById();
            System.out.println(user);
            sqlSession.commit();
            sqlSession.close();
        }
        @Test
        public void testSelectAll(){
            //从工具方法中获取 SqlSession
            SqlSession sqlSession = SqlSessionFactoryUtil.getSqlSession();
            //获取接口对象
            UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
            //测试功能
            List<User> users = userMapper.selectUserAll();
            for (User user:users){
                System.out.println(user);
            }
            sqlSession.commit();
            sqlSession.close();
        }
    }
```

```
    <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
    <mapper namespace="com.mapper.UserMapper">
        <insert id="insertUser">
            insert into t_user values (3,'admin3','123456',23,'男','12345@qq.com')
        </insert>
        <update id="updateUser">
            update t_user set name = '张三' where id = '2'
        </update>
        <delete id="deleteUser">
            delete from t_user where id = '3'
        </delete>
        <select id="selectUserById" resultType="User">
            select * from t_user where id = '2'
        </select>
        <select id="selectUserAll" resultType="User">
            select * from t_user
        </select>
    </mapper>
```

5.MyBatis获取参数值的两种方式(重点)

> https://blog.csdn.net/baiqi123456/article/details/123750259

- MyBatis获取参数值的两种方式：${}和#{}
  ${}本质是字符串拼接
  #{}本质是占位符赋值

- MyBatis获取参数值的各种情况:

1. mapper接口方法的参数为单个的字面量案例 可以通过${}和#{}以任意的字符串获取参数值，但是需要注意${}的单引号问题

```
    <!--User getUserByUsername(String username);-->
    <select id="getUserByUsername" resultType="User">
    <!--两种方式-->
    <!--select * from t_user where username = #{username}-->
        select * from t_user where username = '${username}'
    </select>
```

2. mapper接口方法的参数为多个时 此时MyBatis会将这些参数放在一个map集合中，以两种方式进行存储
   a>以agr0，arg1...为键，以参数为值
   b>以param1，param2...为键，以参数为值
   c>两者混用
   因此只需要通过#{}和${}以键的方式访问值即可，但是需要注意${}的单引号问题

```
    <!--User checkLogin(String username,String password)-->
    <select id="checkLogin" resultType="User">
    <!--select * from t_user where username = #{arg0} and password = #
{arg1}-->
        select * from t_user where username = '${param0}' and password
= '${param1}'
    </select>
```

3. 若mapper接口的方法的参数有多个时,可以手动将这些参数放在一个map中存储

```
    <!--User checkLogin(String username,String password)-->
    <select id="checkLogin" resultType="User">
    <!--select * from t_user where username = #{arg0} and password = #
{arg1}-->
        select * from t_user where username = '${param0}' and password
= '${param1}'
    </select>
```

```
    @Test
    public void testCheckLoginByMap(){
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        ParameterMapper mapper =
sqlSession.getMapper(ParameterMapper.class);
        HashMap<String, Object> map = new HashMap<>();
        map.put("username","hhh");
        map.put("password","123");
```

```
        User user = mapper.checkLoginByMap(map);
        System.out.println(user);
    }
```

4. 若mapper接口的方法的参数是实体类类型的参数时

```
    <!--int insertUser(User user)-->
    <insert id="insertUser">
        insert into t_user values(null,#{username},#{password},#{age},#
{sex},#{email})
    </insert>
```

```
    @Test
    public void testInsertUser(){
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        ParameterMapper mapper =
sqlSession.getMapper(ParameterMapper.class);
        int result = mapper.insertUser(new User(null, "777", "123", 23,
"男", "1224231137@qq.com"));
        System.out.println(result);
    }
```

5. 使用@Param注解命名参数 此时MyBatis会将这些参数放在一个map集合中，以两种方式进行存储
   a>以@Param为键,以参数为值
   b>以param1，param2...为键，以参数为值

```
    <!--User checkLoginByParam(@Param("username")String
username,@Param("password") String password)-->
    <select id="checkLoginByParam" resultType="User">
        select * from t_user where username = #{username} and password
= #{password};
    </select>
```

```
    @Test
    public void testCheckLoginByParam(){
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        ParameterMapper mapper =
sqlSession.getMapper(ParameterMapper.class);
        User user = mapper.checkLoginByParam("777","123");
        System.out.println(user);
    }
```

6.MyBatis的各种查询功能

- MyBatis 的各种查询功能
  1. 若查询出的数据只有一条
     a>可以通过实体类对象接收
     b>可以通过list集合接收
     c>可以通过mao集合接收
     结果:{password=123456, gender=男, name=admin3, id=3, age=23, email=12345@qq.com}
  2. 若当前查询出的数据有多条
     a>可以通过实体类类型的list集合接收
     b>可以通过map类型的list集合接收
     c>可以在mapper接口的方法上添加@MapKey注解， 此时可以将每条数据转换的map集合作为值，以某个字段的值作为键，放在同一个map集合中
     注意：一定不能通过实体类对象接收，抛出异常TooManyResultException
     MyBatis中设置了默认的类型别名

1. 查询一个实体类对象

```
//根据id查询用户信息
List<User> getUserById(@Param("id") Integer id);
```

```
<!--User getUserById(@Param("id") Integer id)-->
<select id="getUserById" resultType="User">
    select * from t_user where id = #{id};
</select>
```

2. 查询一个 List 集合

```
//查询所有的用户信息
List<User> getAllUser();
```

```
<!--List<User> getAllUser();-->
<select id="getAllUser" resultType="User">
    select * from t_user;
</select>
```

3. 查询单个数据

```
//查询用户信息的总记录数
Integer getCount();
```

```
    <!--Integer getCount();-->
    <select id="getCount" resultType="Integer">
        select count(*) from t_user;
    </select>
```

4. 查询一条数据为 Map 的集合

```
    //根据id查询用户信息为一个map集合
    Map<String,Object> getUserByIdToMap(@Param("id") Integer id);
```

```
    <!--Map<String,Object> getUserByIdToMap(@Param("id") Integer id);-->
    <select id="getUserByIdToMap" resultType="map">
        select * from t_user where id = #{id};
    </select>
```

5. 查询多条数据为 Map 的集合

```
    //查询所有用户信息为map集合
    //List<Map<String,Object>> getAllUserToMap();
    @MapKey("id")//找一个唯一字段做键
    Map<String,Object> getAllUserToMap();
```

```
    <!--Map<String,Object> getAllUserToMap();-->
    <select id="getAllUserToMap" resultType="map">
        select * from t_user;
    </select>
```

7.特殊SQL的执行(使用${}的情况)

1. 模糊查询

```
    <select id="getUserByLike" resultType="User">
    /*select * from t_user where name like '%${name}%'*/
    /*select * from t_user where name like concat('%',#{name},'%')*/
    select * from t_user where name like "%"${name}"%"
</select>
```

2. 批量删除

```
    <!--int deleteMore(@Param("ids") String ids);-->
    <delete id="deleteMore">
        delete from t_user where id in (${ids})
    </delete>
```

```
    @Test
    public void testDeleteMore(){
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        SQLMapper mapper = sqlSession.getMapper(SQLMapper.class);
        int deleteMore = mapper.deleteMore("4,8");
    }
```

3. 动态设置表名

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

```
    @Test
    public void testGetUserByTableName(){
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        SQLMapper mapper = sqlSession.getMapper(SQLMapper.class);
        List<User> list = mapper.getUserByTableName("t_user");
        System.out.println(list);
    }
```

4. 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)

    - 添加班级信息
    - 获取新添加的班级的 id
    - 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

```

```

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

```
    @Test
    public void testInsertUser(){
        SqlSession sqlSession = SqlSessionUtils.getSqlSession();
        SQLMapper mapper = sqlSession.getMapper(SQLMapper.class);
        User user = new User(null, "王五", "123", 23, "男", "1224231137");
        mapper.insertUser(user);
        System.out.println(user);
    }
```

8.自定义映射resultMap

> 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. 在 sql 语句中给字段名取别名，别名为属性名，如 emp_name -> empName

```
    <select id="getAllEmp" resultType="Emp">
        select eid,emp_name empName,age,sex,email from t_emp
    </select>
```

2. 在 mybatis-config 中修改配置文件，设置 setting，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。

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

3. 通过 resultMap 设置自定义的映射关系

```
    <!--自定义映射关系-->
    <!--
        resultMap：设置自定义映射关系
        id：唯一标识，不能重复
        type： 设置映射关系的实体类类型
        属性：
        property：设置映射关系中的属性名，必须是type属性设置的实体类类型中的属性名
```

```
            column：设置映射关系中的字段名，必须是sql语句查询出的字段名
    -->
    <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>
```

4. 多对一映射处理

   - 级联方式处理映射关系

```
     </select>
     <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.did = 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

```
    <resultMap id="empAndDeptResultMapTwo" 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" javaType="Dept">
        <result property="did" column="did"></result>
        <result property="deptName" column="dept_name"></result>
```

```
    </association>
</resultMap>
```

- 分步查询处理映射关系

1. 查询员工信息

   - select：设置分步查询的 sql 的唯一标识（namespace.SQLId 或 mapper 接口的全类名。方法名)
   - column: 设置分布查询的条件

```
    //通过分步查询查询员工以及员工所对应的部门信息
    //分步查询第一步：查询员工信息
    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.llt.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

```
    //通过分步查询查询员工以及员工所对应的部门信息
    //分布查询第二部：通过did查询员工所对应的部门
    Dept getEmpAndDeptByStepTwo(@Param("did") Integer did);
```

```
    <!--mapUnderscoreToCamelCase-->
    <select id="getEmpAndDeptByStepTwo" resultType="Dept">
        select * from t_dept where did = #{did}
    </select>
```

分布查询的优点:可以实现延迟加载（解释：需要什么值，就加载什么值 ， 节省资源），但是必须在核心配置文件中设置全局配置信息：

- lazyLoadingEnabled：延迟加载全局开关，当开启时，所有关联对象都会延迟加载（默认关闭，如需延迟加载，手动开启）
- aggressiveLazyLoading：当开启时，任何方法的调用都会加载该对象的所有属性。否则每个属性都会按需加载（默认关闭）

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

- fetchType: 当开启了全局的延迟加载之后，可通过此属性手动控制延迟加载的效果
- fetchType：“lazy|eager”：lazy 表示延迟加载，eager 表示立即加载

```
    <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.llt.mybatis.mapper.DeptMapper.getEmpAndDeptByStepTwo"
                column="did"
                fetchType="eager"></association>
    </resultMap>
```

9.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

1. if:根据标签中test属性所对应的表达式决定标签中的内容是否需要拼接到SQL中

```
    <!--List<Emp> getEmpByCondition(Emp emp);-->
    <select id="getEmpByCondition" resultType="Emp">
        select * from t_emp where 1=1
        <if test="empName != null and empName !=''">
            and emp_name = #{empName}
        </if>
        <if test="age != null and age !=''">
            and age = #{age}
        </if>
```

```
        <if test="sex != null and sex !=''">
            and sex = #{sex}
        </if>
        <if test="email != null and email !=''">
            and email = #{email}
        </if>
    </select>
```

2. where: 当where标签中有内容时,会自动生成where关键字,并且将内容前多余的and或or去掉 当where标签中没有where没有内容时,此时where标签没有任何效果 注意:where标签不能将其中内容后面多余的and或or去掉

```
    <!--List<Emp> getEmpByCondition(Emp emp);-->
    <select id="getEmpByCondition" resultType="Emp">
        select * from t_emp
        <where>
            <if test="empName != null and empName !=''">
                emp_name = #{empName}
            </if>
            <if test="age != null and age !=''">
                and age = #{age}
            </if>
            <if test="sex != null and sex !=''">
                and sex = #{sex}
            </if>
            <if test="email != null and email !=''">
                and email = #{email}
            </if>
        </where>
    </select>
```

注意：where标签不能去掉条件后多余的and/or

3. trim: prefix/suffix:将trim标签内容前或后面添加指定内容 suffixOverrides/prefixOverrides:将trim标签内容前或后面去掉指定内容 若标签中没有内容,trim标签也没有任何效果

```
    <!--List<Emp> getEmpByCondition(Emp emp);-->
    <select id="getEmpByCondition" resultType="Emp">
        select * from t_emp
        <trim prefix="where" suffixOverrides="and|or">
            <if test="empName != null and empName !=''">
                emp_name = #{empName} and
            </if>
            <if test="age != null and age !=''">
                age = #{age} and
            </if>
            <if test="sex != null and sex !=''">
                sex = #{sex} or
            </if>
```

```
            <if test="email != null and email !=''">
                email = #{email}
            </if>
        </trim>
    </select>
```

4. choose,when,otherwise,相当于if...else if...else when至少要有一个，otherwise最多只能有一个

```
    <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>
```

5. foreach collection:设置需要循环的数组或集合 item:表示数组或集合中的每一个数据 separator:循环体之间的分隔符 open:foreach标签所循环的所有内容的开始符 close:foreach标签所训话的所有内容的结束符

   批量删除

```
    <!--int deleteMoreByArray(Integer[] eids);-->
    <delete id="deleteMoreByArray">
        delete from t_emp where eid in
        <foreach collection="eids" item="eid" separator="," open="("
close=")">
            #{eid}
        </foreach>
    </delete>
```

   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>
```

6. sql标签 设置SQL片段:<sql id="empColumns">eid,emp_name,sex,email 引用SQL片段:<include refid="empColumns">

```
<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>
```

10.Mybatis的缓存

1. 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
    - 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
        2. 在映射文件中设置标签<cache />
        3. 二级缓存必须在SqlSession关闭或提交之后有效
        4. 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
2. MyBatis的二级缓存
    - 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

3. 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

4. 整合第三方缓存EHCache（了解）

    1. 添加依赖
    2. 创建EHCache的配置文件ehcache.xml
    3. 设置二级缓存的类型
    4. 加入logback日志

11.Mybatis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 Java实体类 Mapper接口 Mapper映射文件

1. 创建逆向工程的步骤 1.添加依赖和插件

```
    <?xml version="1.0" encoding="UTF-8"?>
    <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/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>

        <groupId>org.example</groupId>
        <artifactId>Mybatis_MBG</artifactId>
        <version>1.0-SNAPSHOT</version>
        <packaging>jar</packaging>

        <properties>
            <maven.compiler.source>19</maven.compiler.source>
            <maven.compiler.target>19</maven.compiler.target>
            <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
        </properties>

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

    </project>
```

2. 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>
```

3. 创建逆向工程的配置文件

```
    <?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="root"
                            password="123456">
            </jdbcConnection>
            <!-- javaBean的生成策略-->
            <javaModelGenerator targetPackage="com.atguigu.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.atguigu.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.atguigu.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="t_emp" domainObjectName="Emp"/>
            <table tableName="t_dept" domainObjectName="Dept"/>
        </context>
    </generatorConfiguration>
```

4. 执行MBG插件的generate目标 选中Maven->Plugins->mybatis-generator->mybatis-generator:generate，双击

12.QBC

1. 查询

```
    @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);
    }
```

2. 增改

> mapper.updateByPrimaryKey(new Emp(1,"admin",22,null,"456@qq.com",3));
> mapper.updateByPrimaryKeySelective(new Emp(2,"admin2",22,null,"456@qq.com",3));

13.分页插件

1. 配置分页插件

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

2. 实现分页功能 limit, index, pageSize index:当前页的起始索引 pageSize:每页显示的条数 pageNum:当前页的页码 index=(pageNum-1)*pageSize 使用Mybatis的分页插件实现分页功能

    1. 需要在查询功能之前开启分页 PageHelper.startPage(int pageNum,int pageSize)
    2. 在查询功能之后获取分页相关信息 PageInfo<Emp> page = new PageInfo<>(list,5) list表示分页数据 5表示当前当行分页的数量

```
    @Test
    public void testPageHelper(){
        SqlSession sqlSession = SqlSessionFactoryUtil.getSqlSession();
        EmpMapper mapper = sqlSession.getMapper(EmpMapper.class);
        Page<Object> page = PageHelper.startPage(2,4);
        System.out.println(page);
        List<Emp> list = mapper.selectByExample(null);
        PageInfo<Emp> pageInfo = new PageInfo<>(list,4);
        System.out.println(pageInfo);
        list.forEach(emp -> System.out.println(emp));
    }
```