MyBatis

一、MyBatis简介

1、MyBatis历史

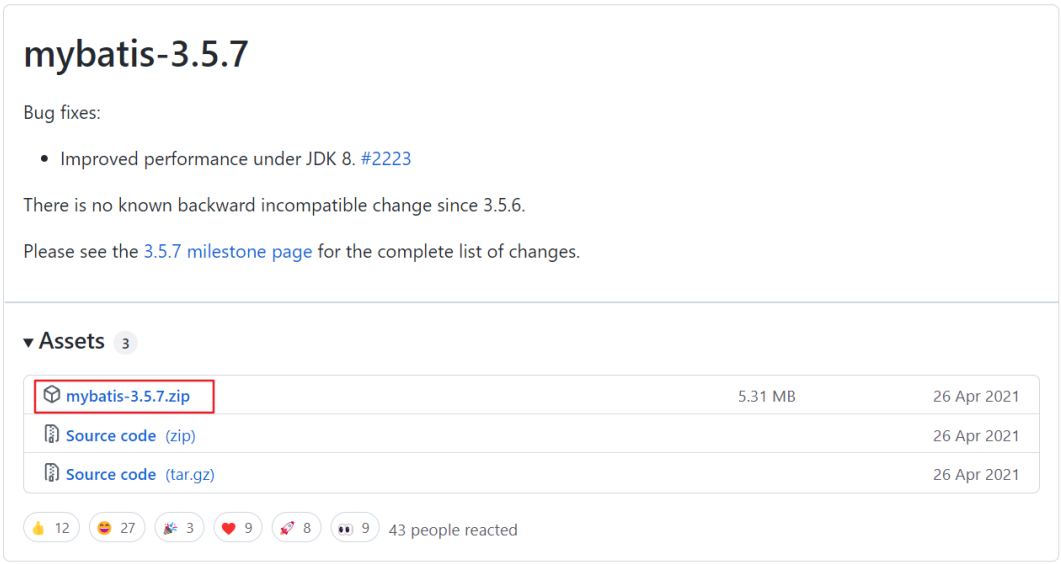
- MyBatis最初是Apache的一个开源项目iBatis, 2010年6月这个项目由Apache Software Foundation迁移到了Google Code。随着开发团队转投Google Code旗下，iBatis3.x正式更名为MyBatis。代码于2013年11月迁移到Github
- iBatis一词来源于“internet”和“abatis”的组合，是一个基于Java的持久层框架。iBatis提供的持久层框架包括SQL Maps和Data Access Objects（DAO）

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（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框架**

3、MyBatis下载

1. 官网：https://github.com/mybatis/mybatis-3
2. 下载最新版本



4、和其他持久层技术的对比

- **JDBC**
    - SQL夹杂着在Java代码中耦合度高，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，开发效率低
- **Hibernate和JPA**
    - 操作简便，开发效率高
    - 程序中的长度复杂SQL需要绕过框架
    - 内部自动产生的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进行部分映射时比较困难
    - 反射操作太多，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
    - 开发效率稍逊于Hibernate，但是完全能够接受

二、MyBatis-HelloWorld

1、开发环境

- IDE：idea
- 构建工具：Maven
- Mysql
- MyBatis

2、创建Maven工程

①打包方式：jar

```
<packaging>jar</packaging>
```

②引入依赖

```
<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>8.0.25</version>
    </dependency>
</dependencies>
```

3、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，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" ?>
<!--MyBatis约束-->
<!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd">
```

```
<configuration>
    <!--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="com.mysql.cj.jdbc.Driver"/>
                <property name="url"
value="jdbc:mysql://localhost:3306/MyBatis"/>
                <property name="username" value="root"/>
                <property name="password" value="123456"/>
            </dataSource>
        </environment>
    </environments>
    <!--引入映射文件-->
    <mappers>
        <mapper resource="mappers/UserMapper.xml"/>
    </mappers>
</configuration>
```

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，不需要提供实现类**

```
public interface UserMapper{
    //添加用户信息
    int insertUser();
}
```

5、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	记录/列（列名）
对象	记录/行（行值）

- 映射文件的命名规则
- 表所对应的**实体类的类名+Mapper.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
- 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

```
<?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">

<!--namespace的值与mapper接口对应-->
<mapper namespace="com.atguigu.mybatis.mapper.UserMapper">
    <!--int insertUser();也就是mapper接口中的方法-->
    <insert id="insertUser">
        insert into t_user values(null,'张三','123',23,'女')
    </insert>
</mapper>
```

- 对于查询功能
    - 需要在select标签后面添加**resultType**【设置默认映射关系，实体类属性和列名对应】或者**resultMap**【设置自定义映射】属性
    - 值为实体类全类名

6、通过junit测试

- 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");

        //读取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
        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();

        //提交事务才可以加入到数据库中
        //sqlSession.commit();
        System.out.println("result:" + result);
    }
}
```

1. 总结

    1. 在获取SqlSession对象之后，如果没有设置openSession为true，在执行对应的方法之后需要**提交事务**才会对数据库进行修改
    2. 也可以将openSession设置为true，可以自动提价事务

7、加入log4j日志功能

①引入依赖

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

②加入log4j的配置文件

- log4j的配置文件命名为log4j.xml，存放在src/main/resources目录下
- 日志的级别：FATAL(致命)>ERROR(错误)>WARN(警告)>INFO(信息)>DEBUG(调试) 从左到右打印的内容越来越详细

```
<?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>
```

三、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，但是相对顺序不能乱）

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">
                <!--
                    数据源信息可以使用配置文件properties，使用properties标
签实现
                -->
                <!--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（一样
的包名）----resource中创建包使用斜线/，也即是文件名写为
com/atguigu/mapper/UserMapper.xml
            2. mapper接口要和mapper映射文件的名字一致
        -->
        <package name="com.atguigu.mybatis.mapper"/>
    </mappers>
</configuration>
```

默认的类型别名

Alias	Mapped Type
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BidDecimal
object	Object
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

四、MyBatis的增删改查

1. 添加

```
<!--int insertUser();-->
<insert id="insertUser">
    insert into t_user
values(null,'admin','123456',23,'男','12345@qq.com')
</insert>
```

2. 删除

```
<!--int deleteUser();-->
 <delete id="deleteUser">
     delete from t_user where id = 6
 </delete>
```

3. 修改

```
<!--int updateUser();-->
 <update id="updateUser">
     update t_user set username = '张三' where id = 5
 </update>
```

4. 查询一个实体类对象

```
<!--User getUserById();-->
<select id="getUserById"
resultType="com.atguigu.mybatis.bean.User">
    select * from t_user where id = 2
</select>
```

5. 查询集合

```
<!--List<User> getUserList();-->
<select id="getUserList"
resultType="com.atguigu.mybatis.bean.User">
    select * from t_user
</select>
```

注意：

- 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
- 当查询的数据为多条时，不能使用实体类作业返回值，只能使用集合，否则会抛出异常TooManyResultsException
- 若查询的数据只有一条，可以用实体类或集合作为返回值

五、MyBatis获取参数值的两种方式（重点）

- 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
- #{}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
- **重点学会使用 #{}**

1、单个字面量类型的参数

- 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${} 和 #{} 以任意的名称获取参数的值，**注意${}需要手动添加单引号**

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

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

2、多个字面量类型的参数

- 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，arg2……为键；①以param1，param2，param3……为键
- 因此只需要通过 ${} 和 #{} 访问集合的键就可以获取对应的值，注意 ${} 需要手动添加单引号
- 使用arg或者param都可以，arg从arg0开始，param从param1开始

```
<!--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>
```

3、map集合类型的参数

- 若mapper接口中需要的参数为多个时，此时可以手动创建map集合，将这些参数放在map中只需要通过${} 和 #{} 访问map集合就可以获取相应的值，注意 ${} 需要手动添加单引号

```
<!--User checkLogingByMap(Map<String,Object> map);-->
<select id="checkLogingByMap" resultType="User">
    select * from t_user where username = #{username} and
password = #{password}
</select>
```

```
@Test
public void checkLogingByMap(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    ParameterMapper mapper =
sqlSession.getMapper(ParameterMapper.class);
    Map<String,Object> map = new HashMap<>();
    map.put("usermane","admin");
    map.put("password","123456");
    User user = mapper.checkLoginByMap(map);
    System.out.println(user);
}
```

4、实体类类型的参数【建议】

- 若mapper接口中方法参数为实体类对象此时可以使用 ${} 和 #{}，通过访问实体类对象中的**属性名**获取属性值，注意 ${} 需要手动添加单引号
- **属性名：get和set 方法的方法名去掉get和set之后首字母变为小写**

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

```
@Test
public void insertUser() {
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    ParameterMapper mapper =
sqlSession.getMapper(ParameterMapper.class);
    User user = new
User(null,"Tom","123456",12,"男","123@321.com");
    mapper.insertUser(user);
}
```

5、使用@Param标识参数【建议】

- 可以通过@Param注解表示mapper接口中的方法参数，此时会将这些参数放在mapper集合中
    - 以@Param注解的value属性值为键，以参数为值
    - 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 ${} 需要手动添加单引号

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

```
@Test
public void checkLoginByParam(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    ParameterMapper mapper =
sqlSession.getMapper(ParameterMapper.class);
    mapper.CheckLogingByParam("admin","123456");
}
```

六、MyBatis的各种查询功能

1. 查询结果的数据**只有一条**
    1. 实体类对象接收
    2. List集合接收
    3. Map集合接收
2. 查询结果的数据**有多条**
    1. 实体类类型的List集合接收
    2. Map类型的List集合接收
    3. 在mapper接口方法上添加**@MapKey注解**

1、查询一个实体类对象

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

2、查询一个List集合

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

3、查询单个数据

类型别名参考第三大点中的表格，也可以用默认的类型别名如Integer int 等

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

4、查询一条数据为map集合

```
<!--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}-->
```

5、查询多条数据为map集合

①方法一

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

②方法二

```
public interface SelectMapper{
    //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}
    }
-->
```

七、特殊SQL的执行

1、模糊查询

```
<!--List<User> getUserByLike(@Param("username") String username);-->
<select id="getUserByLike" resultType="User">
    <!--三种方式编写SQL语句-->
    <!--select * from t_user where username like '%${username}%'-->
    <!--select * from t_user where username like concat('%',#
{username},'%')-->
    select * from t_user where username like "%"#{username}"%"
</select>
```

- **其中like "%"#{username}"%" 最常用**

2、批量删除

- **只能使用 ${}**，如果使用#{}，则解析后的sql语句为delete from t_user where id in ('1,2,3')，这样1,2,3就是一个整体，不符合要求

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

```
@Test
public void deleteMore(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    SqlMapper mapper = sqlSession.getMapper(SqlMapper.class);
    int result = mapper.deleteMore("1,2,3,4");
    System.out.println(result);
}
```

3、动态设置表名

- 只能使用 ${}，因为表名不能加单引号

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

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>
```

```
//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
}
```

八、自定义映射resultMap

1、解决映射关系

- 在SQL语句中为字段取别名
- 设置MyBatis的全局配置，然后实体类中按照设置的关系定义属性名

```
<!--在mybats-config.xml全局配置文件中-->
<settings>
    <!--将下换线_自动映射为驼峰，如emp_name ==> empName-->
    <setting name="mapUnderscoreToCamelCase" value="true"/>
</settings>
```

2、resultMap处理字段和属性的映射关系

- 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：设置普通字段的映射关系（实体类属性名）**
        - **column：设置映射关系中表中的字段名（列名）**
- 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**

```
<!--在当前的mybatis-mapper.xml中设置-->
<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>
```

3、多对一映射处理

> 查询员工信息以及员工所对应的部门信息

- 在**一的那方**的实体类中添加多的那方的实体类对象，以及构造器，get和set方法

```
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
}
```

①级联关系处理多对一映射关系

```
<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>

    <!--注意需要指定是哪个类的did-->
    <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>
```

②使用association处理多对一映射关系

- association：处理多对一的映射关系
- propert：需要处理多对一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">
        <id property="did" column="did"></id>
        <result property="deptName" column="dept_name"></result>
    </association>

</resultMap>
<!--Emp getEmpAndDept(@Param("eid")Integer eid);-->
<select id="getEmpAndDept" resultMap="empAndDeptResultMapTwo">
    select * from t_emp left join t_dept on t_emp.eid = t_dept.did
where t_emp.eid = #{eid}
</select>
```

③分步查询处理多对一映射关系【常用】

1、查询员工信息

1. select：设置分布查询的sql唯一标识（namespace.SQLId 或 mapper接口的全类名）
2. column：设置分布查询的条件

```
<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>

    <!--select属性指定了dept值的数据来源于哪个SQL查询出来的，并且通过did去查询-->
    <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

```
<!--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>
```

4、一对多映射处理

- 在**多的一方的实体类**定义一个**一的那方的集合**

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

①collection标签

- collection：用来处理一对多的映射关系
- ofType：表示该属性对应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>
```

②分步查询

1. 查询部门信息

```
<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.getDeptAndEmpBySt
epTwo"
                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

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

5、延迟加载

- 分布查询的优点：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>
    <setting name="lazyLoadingEnabled" value="true"/>
</settings>
```

- 关闭延迟加载，两条SQL语句都执行
- 开启延迟加载，只执行获取emp的SQL语句，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.getEmpAndDeptBySt
epTwo"
                 column="did"
                 fetchType="lazy"></association>
</resultMap>
```

九、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

①if

- if标签可以通过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，则标签中的内容被执行，反之不执行。
- **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用来拼接and语句，例如当empName为null时
        - 如果不加上1=1，则SQL语句为select * from t_emp where and age = ? and sex = ? ，此时where会与and连用，SQL**语句报错**
        - 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<Emp> getEmpByCondition(Emp emp);-->
<select id="getEmpByCondition" resultType="Emp">
    select * from t_emp where 1=1
    <!--直接写属性名-->
    <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>
```

②where

- where和if一般结合使用
- 若where标签中的if标签**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【注意：where标签不能去掉条件后多余的and/or】

```
<!--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>
```

③trim

- 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

```
<!--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>
```

④choose、when、otherwise

- **choose、when、otherwise相当于if...else if..else**
- **when至少要有一个，otherwise至多只有一个**

```
<!--相当于if a else (if b else(if c d)) 只会执行其中一个-->
<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>
```

⑤foreach

- 属性
    - collection：设置要循环的数据或集合
    - 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**【数组为逗号】，分隔符前后默认有一个空格
    - open：设置foreach标签中的内容开始符
    - close：设置foreach标签中的内容结束符
- 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>
```

```
@Test
public void deleteMoreByArray() {
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    DynamicSQLMapper mapper =
sqlSession.getMapper(DynamicSQLMapper.class);
    int result = mapper.deleteMoreByArray(new Integer[]{6, 7, 8, 9});
    System.out.println(result);
}
```

- 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);
}
```

⑥SQL片段

- SQL片段，可以记录一些公共的sql片段，在使用的地方通过include标签引入
- 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>
```

十、MyBatis的缓存

1、MyBatis的一级缓存【默认开启】

1. 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，就会从缓存中直接取出，不会从数据库重新访问
2. 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
    2. 同一个SqlSession但是查询条件不同
    3. 同一个SqlSession两次查询期间执行了任何一次**增删改**操作
    4. 同一个SqlSession两次查询期间手动清空了缓存
       【sqlSession.clearCache();】

2、MyBatis的二级缓存

1. 二级缓存是**SQLSessionFactory**级别的，通过同一个SQLSessionFactory创建的SQLSession查询的结果会被缓存，此后拖再次执行相同的查询语句，结果就会从缓存中取出
2. 二级缓存**开启**的条件【同时满足】
    1. 在核心配置文件中，设置全局配置属性cacheEnabled="true"，默认为true，可以不需要设置
    2. 在映射文件中设置标签<cache /> 【直接写在mapper配置文件中即可】
    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
3. 二级缓存**失效**的情况
    1. 两次查询之间执行了**任意的增删改**，会使一级缓存和二级缓存同时失效

3、二级缓存的相关配置

- 在mapper配置文件的cache标签可以设置一些属性
- eviction：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 Fist 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：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。【提高性能】
    - 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【保证安全】

4、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
- 如果一级缓存没有命中，则查询数据库
- **SQLSession关闭之后，一级缓存的数据会引入二级缓存**

5、整合第三方缓存EHCache【了解】

①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>
```

②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

③创建EHCache的配置文件ehcache.xml

注意：配置文件名字必须为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>
```

④设置二级缓存的类型

- 在xxxMapper.xml文件中设置二级缓存类型

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

⑤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>
```

⑥EHCache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

十一、MyBatis的逆向工程

逆向工程的本质是代码生成器

- 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，反向生成一下资源
    - Java实体类
    - Mapper接口
    - Mapper映射文件（配置文件）

1、创建逆向工程的步骤

①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>
```

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

- 注意：配置文件名字必须为 generatorConfig.xml

```
<?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maven中的插件

1. 在IDEA自带的maven工具中，找到对应的插件**直接双击运行**【双击mybatis-generator插件】
2. 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
    1. dependency中的数据库驱动和插件中的数据库驱动**应该相同**
3. 执行结果：**创建Mapper接口、实体类和映射文件**

2、QBC

①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

```
@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

- `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));`

十二、分页插件

1、分页插件使用步骤

①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>
```

②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>
```

2、分页插件的使用

①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);
}
```

②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);
}
```

结果

```
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);
}
```

结果

```
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**中直接输出的数据

③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]

附录：模板文件

MyBatis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>
    <!--数据库信息配置文件-->
    <properties resource="jdbc.properties"/>
    <!--类型别名，别名为类名且不区分大小写-->
    <typeAliases>
```

```
<!--        <typeAlias type="com.atguigu.mybatis.pojo.User"
alias="User"/>-->
        <package name="com.atguigu.mybatis.pojo"/>
    </typeAliases>
    <!--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}"/>
            </dataSource>
        </environment>
    </environments>
    <!--引入映射文件-->
    <mappers>
<!--        如果导入包要求mapper包名和mapper接口包名一致且文件名一致-->
<!--        <mapper resource="mappers/ParameterMapper.xml"/>-->
        <package name="com.atguigu.mybatis.mapper"/>
    </mappers>
</configuration>
```

MyBatis的Mapper配置文件

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

</mapper>
```

SQLSession工具类

```
public class SqlSessionUtils {
    public static SqlSession getSqlSession() {
        SqlSession sqlSession = null;
        try {
            InputStream inputStream =
Resources.getResourceAsStream("mybatis-config.xml");
            SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
            sqlSession = sqlSessionFactory.openSession(true);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return sqlSession;
    }
}
```