

MyBatis

[01_MyBatis课程介绍哔哩哔哩bilibili](#)

MyBatis简介

MyBatis历史与特性

1. 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。

2. iBatis

iBatis提供的持久层框架包括SQL Maps和Data Access Objects (DAO)。

SQL Maps指数据库中的数据和java数据之间的映射关系，即MyBatis封装JDBC的过程。

3. 特性

- 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。
- 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。
- 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。
- MyBatis 是一个半自动的ORM (Object Relation Mapping, 对象关系映射) 框架，即将实体类对象和关系型数据库中的数据创建映射关系。

对比其他持久化层技术

1. JDBC

- SQL 夹杂在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。

2. 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。

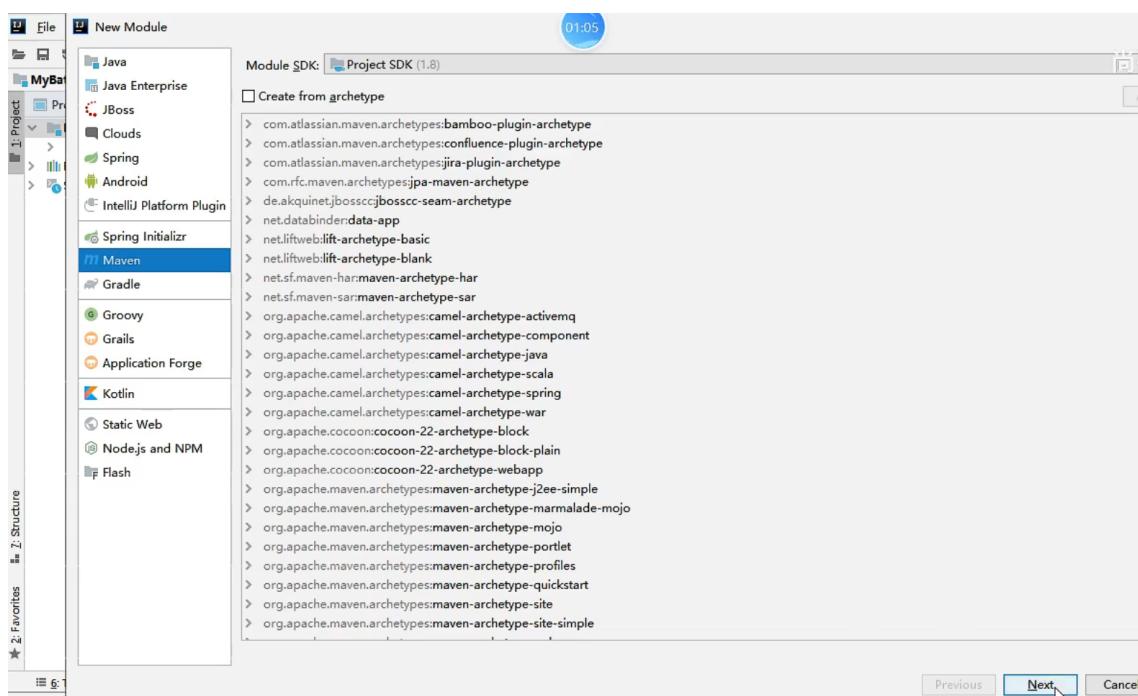
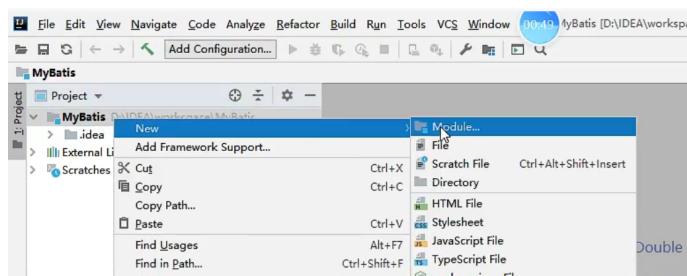
3. MyBatis

- 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。

MyBatis搭建步骤

创建Maven工程

1. 创建module



2. 导入依赖

```
<dependencies>
    <!-- Mybatis核心 -->
    <dependency>
        <groupId>org.mybatis</groupId>
```

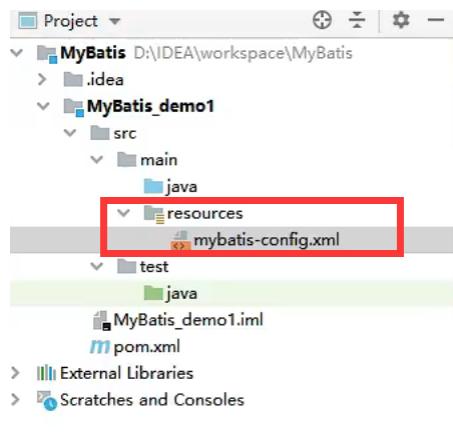
```

<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>5.1.3</version>
</dependency>
</dependencies>

```

创建MyBatis核心配置文件

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，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>
    <!--设置连接数据库的环境-->
    <environments default="development">
        <environment id="development">
            <transactionManager type="JDBC"/>    <!--事务管理器类型-->
            <dataSource type="POOLED">
                <property name="driver"
value="com.mysql.cj.jdbc.Driver"/>

```

```

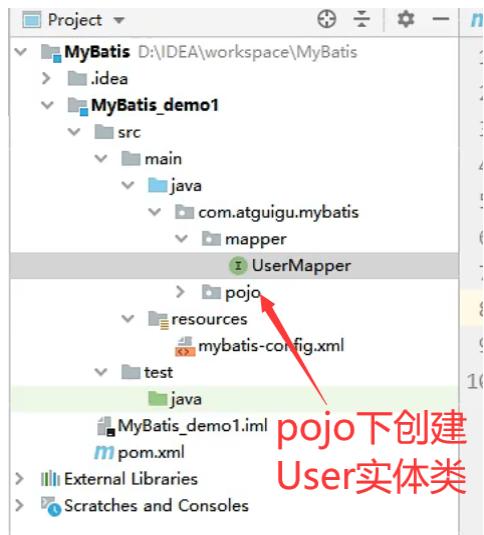
<property name="url"
value="jdbc:mysql://localhost:3306/MyBatis"/> <!--3306/后面是数据库
名称-->
    <property name="username" value="root"/>
    <property name="password" value="123456"/>
</dataSource>
</environment>
</environments>
<!--引入映射文件-->
<mappers>
    <mapper resource="mappers/UserMapper.xml"/>
</mappers>
</configuration>

```

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。

当通过MyBatis中的方式创建了Mapper接口对象， 调用接口中的方法时， 就能自动对应某个sql语句并且执行。



```

package com.atguigu.mybatisplus;

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

```

创建MyBatis的映射文件(重要)

1. ORM

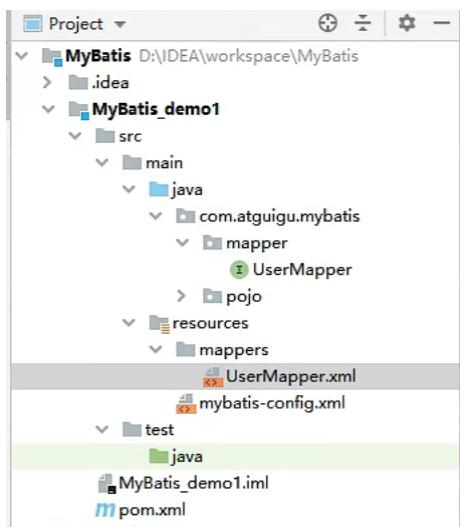
对象指的是Java的实体类对象，关系指的是关系型数据库，映射指的是二者之间的对应关系。

Java概念	数据库概念
类	表
属性	字段/列
对象	记录/行

2. 映射文件

MyBatis映射文件用于编写SQL，访问以及操作表中的数据，存放的位置是src/main/resources/mappers目录下。

映射文件的名字与Mapper接口名一致即可。



3. MyBatis使用中的一致性 (重点)

使用MyBatis后，总的映射关系为：表——实体类——Mapper接口——映射文件。维护好这个映射关系就已经能正确使用MyBatis了。因此写语句时，先在接口中写方法，然后去映射文件进行配置。

MyBatis面向接口编程，要注意两个一致：

- (1) mapper接口的全类名和映射文件的命名空间 (namespace) 保持一致。
- (2) 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。

```
package com.atguigu.mybatisplus.mapper;

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

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.atguigu.mybatisplus.UserMapper"> <!--与Mapper接口的全类名一致-->
    <!--int insertUser();-->
    <insert id="insertUser"> <!--与接口方法名一致-->
        insert into t_user values(null,'张三','123',23,'女')
    </insert>
</mapper>
```

写完配置文件后，需要将映射文件路径写到MyBatis核心配置文件中。

搭建过程测试

```
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);

        //创建SqlSession对象，当设置为true，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
//sqlSession.commit();
System.out.println("result:" + result);
}

}

//也可以手动开启和提交事务
//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(); //此时默认
不自动提交
//手动提交事务
sqlSession.commit();

```

说明：

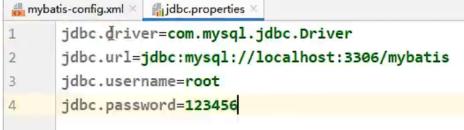
- SqlSession：代表Java程序和数据库之间的会话。（ HttpSession是Java程序和浏览器之间的会话）
- SqlSessionFactory：是“生产”SqlSession的“工厂”。
- 工厂模式：如果创建某一个对象，使用的过程基本固定，那么我们就可以把创建这个对象的相关代码封装到一个“工厂类”中，以后都使用这个工厂类来“生产”我们需要的对象。
- 如果要自动提交事务，则在获取sqlSession对象时，使用 `sqlSession = sqlSessionFactory.openSession(true);`，传入一个 Boolean类型的参数，值为true，这样就可以自动提交。

核心配置文件

了解即可，使用时直接复制。在SSM整合环境中，可以没有核心配置文件。

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。



```
mybatis-config.xml
1 jdbc.driver=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/mybatis
3 jdbc.username=root
4 jdbc.password=123456
```

```
<?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>

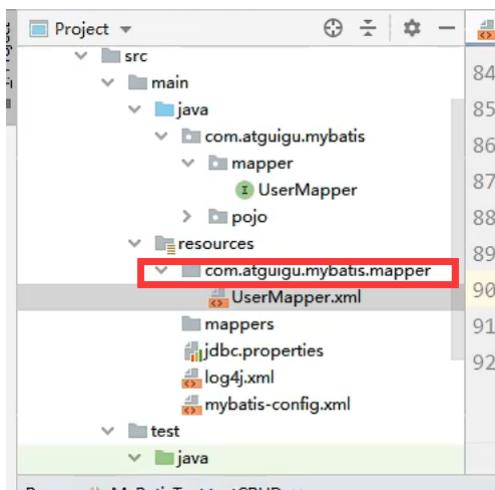
    <typeAliases>
        <!--
            typeAlias: 设置某个具体的类型的别名，设置别名有利于查询语句的
            resultType属性设置
            属性：
            type: 需要设置别名的类型的全类名
            alias: 设置此类型的别名，且别名不区分大小写。若不设置此属性，该类型拥
            有默认的别名，即类名
        -->
        <!--<typeAlias type="com.atguigu.mybatisplus.bean.User">
        </typeAlias>-->
        <!--<typeAlias type="com.atguigu.mybatisplus.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.mybatisplus.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.mybatisplus.mapper"/>
</mappers>
</configuration>

```



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

5. 查询集合

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

- 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
2. 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获取参数值方式

获取参数值的两种方式

1. 两种方式比较

`${}的本质是字符串拼接, #{}的本质是占位符赋值。`

`${}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; 但是#{}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, 会自动添加单引号。`

2. 能用\${}情况尽量用\${}。

字面量类型的参数

1. 单个字面量类型

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（最好见名识意）获取参数的值。注意\${}是字符串拼接，传入字符串需要手动加单引号。

```
<!--User getUserByUsername(String username); mapper接口的方法，传入参数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

在MyBatis的底层，若检测到Mapper接口中的方法传入多个参数，会自动把这些参数放在一个map集合中。这个map集合存储数据的方式有两种：

方式1：以arg0,arg1...为键，以参数为值；
方式2：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集合里相对应的值，注意\${}需要手动加单引号。

```
<!--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>
```

map集合&实体类类型的参数

1. map集合类型

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。

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

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

2. 实体类类型

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。

属性：在实体类中有getset方法的就是属性。

```
@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);  
}
```

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

@Param标识参数

与多个字面量参数相似，MyBatis会将这些参数放在一个map集合中，以两种方式进行存储：

方式1：以@Param(value = "值")注解中的值为键，传入的参数为值。

方式2：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。

```
<!--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>
```

或者

```
<!--User CheckLoginByParam(@Param("username") String username,
@Param("password") String password);-->
<select id="CheckLoginByParam" resultType="User">
    select * from t_user where username = #{param1} and
password = #{param2}
</select>
```

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

建议：用@Param进行字面量类型的传参。

MyBatis查询功能

查询实体类&List&单个数

1. 查询一个实体类对象

```
/**  
 * 根据用户id查询用户信息  
 * @param id  
 * @return  
 */  
User getUserById(@Param("id") int id);
```

```
<!--User getUserById(@Param("id") int id);-->  
<select id="getUserById" resultType="User">  
    select * from t_user where id = #{id}  
</select>  
  
<!--查询结果:  
User{id=3,username='admin',password='12346',age=23,sex='男',email='12@qq.com'}-->
```

2. 查询结果为多个实体类，用List接收

```
/**  
 * 查询所有用户信息  
 * @return  
 */  
List<User> getUserList();
```

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

3. 查询结果为单个数据

```
/**  
 * 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();
```

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

查询结果存在map里

1. 查询单条数据

有时查询的数据没有对应的实体类可以接收，此时将查询出来的数据存放在map里。

```
/**  
 * 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}-->
```

2. 查询多条数据

(1) 方法一：将map内嵌在list集合里

```
/**  
 * 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  
 */  
List<Map<String, Object>> getAllUserToMap();
```

```
<!--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：加@MapKey注解，注解里的value会作为map的键。

```
/**  
 * 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}  
    }  
-->
```

特殊SQL的执行

模糊查询&批量删除&动态表名

1. 模糊查询

由于#{}占位，解析时会自动加上单引号，因此'%"#{}}%'的形式会报错。

```
/**  
 * 根据用户名进行模糊查询  
 * @param username  
 * @return java.util.List<com.atguigu.mybatisplus.pojo.User>  
 * @date 2022/2/26 21:56  
 */  
List<User> getUserByLike(@Param("username") String username);
```

```
<!--List<User> getUserByLike(@Param("username") String  
username);-->  
<select id="getUserByLike" resultType="User">  
    <!--方法一： select * from t_user where username like  
    '%${mohu}%'-->  
    <!--方法二： select * from t_user where username like  
    concat('%',#{mohu},'%')-->  
    select * from t_user where username like "%"#{mohu}"%"  
</select>
```

其中`select * from t_user where username like "%"#{mohu}"%"`是最常用的。

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。正确的语句应该是`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  
 * @param ids  
 * @return int  
 * @date 2022/2/26 22:06  
 */  
int deleteMore(@Param("ids") String ids);
```

```
<delete id="deleteMore">
    delete from t_user where id in (${ids})
</delete>
```

```
//测试类
@Test
public void deleteMore() {
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    SQLMapper mapper = sqlSession.getMapper(SQLMapper.class);
    int result = mapper.deleteMore("1,2,3,8");
    System.out.println(result);
}
```

3. 动态表名

只能使用\${}，因为表名不能加单引号。

```
/**
 * 查询指定表中的数据
 * @param tableName
 * @return java.util.List<com.atguigu.mybatisplus.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>
```

设置主键自增

当插入数据时，可以设置主键自增。

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。

```
/**  
 * 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  
}
```

自定义映射

字段名和属性名不一致的映射

当在mapper.xml文件里写语句时，resultType仅支持字段名与属性名完全相同。由于数据库表字段常使用_，而java属性常使用驼峰，两者可能不完全相同，此时会有问题。解决方法有三种：

(1) 方法一：为字段起别名，保持和属性名的一致。

```
<!--List<Emp> getAllEmp();-->  
<select id="getAllEmp" resultType="Emp">  
    select eid,emp_name empName,age,sex,email from t_emp <!--起别名  
后，查询出来的字段将变成empName-->  
</select>
```

(2) 方法二：设置全局配置，将_自动映射为驼峰

在MyBatis的核心配置文件mybatis-config.xml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`。

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

(3) 方法三：通过resultMap设置自定义的映射关系

使用`resultMap`时，需要将实体类的所有属性都列出来，即使字段名和属性名一致也需要映射。

```
<!--id表示自定义映射的唯一标识，不能重复-->
<!--type是查询的数据要映射的实体类类型-->
<!--子标签中
    id设置主键的映射，result设置普通字段的映射
    property是实体类的属性名，column是查询的表字段名-->

<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>
```

多对一映射处理

1. 问题描述

假设有部门类`Dept`和员工类`Emp`，由于员工和部门的关系是多对一，我们的员工类`Emp`类定义如下。

```

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
}

```

在员工表里，除了员工信息，还有部门id和部门名两个字段。此时若用两表联查查询员工信息，查出来的字段did、dept_name分别是整型和字符串类型，但对应的实体类Emp中只有Dept类型的属性。

该问题的解决方法有如下三种。

2. 级联属性赋值

在resultMap中对级联属性进行赋值。

```

<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">
        <!--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>
    </result>
</resultMap>

```

查询结果：

```
Emp{eid=1, empName='张三', age=23, sex='男', email='123@qq.com', dept=Dept{did=1, deptName='A'}}
```

3. 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>

```

4. 分步查询

顾名思义，采用多步完成。在该问题中，先查询Emp，再查询Dept。

(1) 第一步：查询员工信息

```

//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.mybatisplus.Emp
 */
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>
    <!--
        select: 设置分步查询的sql语句的唯一标识。书写规则：mapper接口的
        全类名.方法名
        column: 设置分步查询的条件。即第一步查询结束后，用什么字段作为条件
        去进行第二步查询
    -->

```

```

<association property="dept"
    select="com.atguigu.mybatisplus.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) 第二步：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.mybatisplus.pojo.Emp
 */
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>

```

延迟加载

1. 含义

分步查询的好处就是延迟加载。比如上面的问题，我们先查询了员工表，后查询了部门表。如果设置了延迟加载，查询不涉及部门表的信息时，我们可以只查询员工表。只有当需要查询部门表时再去查询部门表，提高了性能。

2. 延迟加载的实现

实现延迟加载，需要在核心配置文件进行配置。

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。若要实现延迟加载，这个属性需要关闭（默认也是为false）。

```
<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());
}
```

现在我们只想要获得员工姓名。若关闭延迟加载，两条sql都会执行：

```
DEBUG 02-27 20:57:26,579 ==> Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 02-27 20:57:26,604 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 02-27 20:57:26,630 ====> Preparing: select * from t_dept where did = ? (BaseJdbcLogger.java:137)
DEBUG 02-27 20:57:26,630 ====> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 02-27 20:57:26,632 <====      Total: 1 (BaseJdbcLogger.java:137)
DEBUG 02-27 20:57:26,633 <==      Total: 1 (BaseJdbcLogger.java:137)
张三
```

若开启延迟加载，仅执行第一条sql：

```
DEBUG 02-27 20:55:30,274 ==> Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 02-27 20:55:30,298 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 02-27 20:55:30,341 <==      Total: 1 (BaseJdbcLogger.java:137)
张三
```

开启后，需要用到查询dept的时候才会调用相应的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());
}
```

```
DEBUG 02-27 20:59:52,722 ==> Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 02-27 20:59:52,746 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 02-27 20:59:52,790 <==      Total: 1 (BaseJdbcLogger.java:137)
张三
-----
DEBUG 02-27 20:59:52,792 ==> Preparing: select * from t_dept where did = ? (BaseJdbcLogger.java:137)
DEBUG 02-27 20:59:52,792 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 02-27 20:59:52,793 <==      Total: 1 (BaseJdbcLogger.java:137)
Dept{did=1, deptName='A'}
```

3. fetchType属性

lazyLoadingEnabled属性开启的是延迟加载的**全局开关**，即每个分步查询都会延迟加载。若某些mapper不想要延迟加载，可以通过fetchType属性手动控制。需要注意的是，fetchType仅在lazyLoadingEnabled开启的情况下才会生效。

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.mybatisplus.mapper.DeptMapper.getEmpAndDeptByStepTwo"
            column="did"
            fetchType="lazy"></association>
    </resultMap>
```

一对多映射处理

1. 问题描述

和多对一相同，假设有部门类Dept和员工类Emp，由于部门和员工的关系是一对多，我们的部门类Dept定义如下。

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

当以Dept为主表进行查询时，查询的Dept表里有列表属性，此时需要进行处理。

2. 使用collection处理

```

<resultMap id="DeptAndEmpResultMap" type="Dept">
    <id property="did" column="did"></id>
    <result property="deptName" column="dept_name"></result>
    <!--
        collection: 处理一对多的映射关系。
        property: 需要处理映射关系的对象（列表名）。
        ofType: 表示该属性所对应的集合中存储数据的类型。
    -->
    <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>

```

3. 分步查询

(1) 第一步：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.mybatisplus.Dept
 */
Dept getDeptAndEmpByStepOne(@Param("did") Integer did);

```

```

<resultMap id="DeptAndEmpByStepOneResultMap" type="Dept">
    <id property="did" column="did"></id>
    <result property="deptName" column="dept_name"></result>
    <!--
        select: 设置分步查询的sql语句的唯一标识。书写规则：mapper接口的
        全类名.方法名
        column: 设置分步查询的条件。即第一步查询结束后，用什么字段作为条件
        去进行第二步查询
    -->
    <collection property="emps"

```

```

select="com.atguigu.mybatisplus.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) 第二步：查询员工信息

```

/**
 * 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.mybatisplus.Emp>
 */
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>

```

动态SQL

简介&if&where&trim

1. 简介

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。

例如有语句 `select * from emp where emp_name = #{name} and age = #{age} and sex = #{sex}`。此时若name没有传进来，第一项查询不会进行，语句变成了 `select * from emp where and age = #{age} and sex = #{sex}`，是个错误的语句。

2. if标签

作用：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。

用法：

```
<!--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>
```

说明：

在where后面需要添加一个恒成立条件1=1，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。

3. where标签

作用：where标签一般和if标签一起使用。这样就不需要添加恒成立条件。

- 当where标签中间有内容时，会自动生成where关键字，并且将内容前多余的and或or去掉（内容后的无法去掉）；
- 当where标签中间没有内容时，此时where标签没有任何效果，即不会生成where语句。

用法：

```
<!--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>

```

4. trim

作用：trim用于去掉或添加标签中的内容。

常用属性：

- prefix：在trim包裹的sql前添加指定内容，可以理解为在前面添加内容。
- suffix：在trim包裹的sql后添加指定内容，可以理解为在后面添加内容。
- prefixOverride：去掉trim包裹的sql的指定首部，可以理解为对内部语句去除特定首部。
- suffixOverride：去掉trim包裹的sql的指定尾部，可以理解为对内部语句去除特定尾部。

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`。

用法：

```

<!--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>
```

choose&when&otherwise

1. 作用

choose、when、otherwise标签是一起使用的，相当于`if.....else if.....else`，其中choose用于开启这个判断，when相当于`if.....else if`，而otherwise相当于最后的`else`。

因此，when至少要有一个，otherwise至多只有一个。

2. 用法

```
<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);
}

```

```

DEBUG 11-30 16:29:43,931 ==> Preparing: select * from t_emp WHERE emp_name = ? (BaseJdbcLogger.java:137)
DEBUG 11-30 16:29:43,959 ==> Parameters: 张三(String) (BaseJdbcLogger.java:137)
DEBUG 11-30 16:29:43,978 <== Total: 1 (BaseJdbcLogger.java:137)
[Emp{eid=1, empName='张三', age=23, sex='男', email='123@qq.com', dept=null}]

Process finished with exit code 0

```

for each

1. 作用及属性

for each标签用于循环，其主要的属性为：

- 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标签内循环内容前的开始符（一般仅在where id in使用）
- close：foreach标签内循环内容后的结束符（一般仅在where id in使用）

2. 批量删除

(1) 方式1：where id in ()

```

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

```

(2) 方式2：where id = or id =

```

<!--int deleteMoreByArray(@Param("eids") Integer[] eids);-->
<delete id="deleteMoreByArray">
    delete from t_emp where
    <foreach collection="eids" item="eid" separator="or">
        eid = #{eid}
    </foreach>
</delete>

```

3. 批量添加

```
insert into 表名 values (),(),()
```

```

<!--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);
}

```

```

DEBUG 11-30 17:05:04,646 ==> Preparing: insert into t_emp values (null,?, ?, ?, ?, null), (null, ?, ?, ?, ?, null), (null, ?, ?, ?, ?, null)
DEBUG 11-30 17:05:04,683 ==> Parameters: a1(String), 23(Integer), 男(String), 123@qq.com(String), a2(String),
DEBUG 11-30 17:05:04,688 <==    Updates: 3  (BaseJdbcLogger.java:137)
3

```

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">id,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缓存

一级缓存

1. 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。一级缓存默认开启。

```
@Test
public void testCache(){
    SqlSession sqlSession = SqlSessionUtils.getSqlSession();
    CacheMapper mapper =
    sqlSession.getMapper(CacheMapper.class);
    Emp emp1 = mapper.getEmpById(1);
    System.out.println(emp1);
    Emp emp2 = mapper.getEmpById(1);
    System.out.println(emp2);
}
```

```
DEBUG 11-30 17:24:57,397 ==> Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 11-30 17:24:57,424 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 11-30 17:24:57,444 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=23, sex='男', email='123@qq.com', dept=null}
Emp{eid=1, empName='张三', age=23, sex='男', email='123@qq.com', dept=null} I
```

sql只执行了一次

```

//测试同一个sqlSession, 不同Mapper对象
@Test
public void testCache(){
    sqlSession sqlSession = sqlSessionUtils.getSqlSession();
    CacheMapper mapper1 =
        sqlSession.getMapper(CacheMapper.class);
    Emp emp1 = mapper1.getEmpById(1);
    System.out.println(emp1);

    CacheMapper mapper2 =
        sqlSession.getMapper(CacheMapper.class);
    Emp emp2 = mapper2.getEmpById(1);
    System.out.println(emp2);
}

```

```

DEBUG 11-30 17:26:43,742 ==> Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 11-30 17:26:43,773 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 11-30 17:26:43,794 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=23, sex='男', email='123@qq.com', dept=null}
Emp{eid=1, empName='张三', age=23, sex='男', email='123@qq.com', dept=null}

```

```

//测试不同sqlSession
@Test
public void testCache(){
    sqlSession sqlSession1 = sqlSessionUtils.getSqlSession();
    CacheMapper mapper1 =
        sqlSession1.getMapper(CacheMapper.class);
    Emp emp1 = mapper1.getEmpById(1);
    System.out.println(emp1);

    sqlSession sqlSession2 = sqlSessionUtils.getSqlSession();
    CacheMapper mapper2 =
        sqlSession2.getMapper(CacheMapper.class);
    Emp emp2 = mapper2.getEmpById(1);
    System.out.println(emp2);
}

```

```

DEBUG 11-30 17:27:32,124 ==> Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 11-30 17:27:32,160 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 11-30 17:27:32,181 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=23, sex='男', email='123@qq.com', dept=null}
DEBUG 11-30 17:27:32,278 ==> Preparing: select * from t_emp where eid = ? (BaseJdbcLogger.java:137)
DEBUG 11-30 17:27:32,279 ==> Parameters: 1(Integer) (BaseJdbcLogger.java:137)
DEBUG 11-30 17:27:32,280 <==      Total: 1 (BaseJdbcLogger.java:137)
Emp{eid=1, empName='张三', age=23, sex='男', email='123@qq.com', dept=null}

```

可以看到，没有从缓存中取数据。

2. 一级缓存失效的四种情况

- 不同的SqlSession对应不同的一级缓存

- 同一个SqlSession但是查询条件不同
- 同一个SqlSession两次查询期间执行了任何一次增删改操作
- 同一个SqlSession两次查询期间手动清空了缓存,
`sqlSession.clearCache()`

二级缓存

1. 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。

2. 二级缓存开启条件

- (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（写sql语句的xml文件）设置标签。
- (3) 二级缓存必须在SqlSession关闭或提交之后有效。

```
sqlSession.commit();
sqlSession.close();
```

- (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。

```
public class Emp implements Serializable {
}
```

3. 二级缓存失效情况

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。

4. 二级缓存相关配置

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

缓存查询顺序

- 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。

MyBatis逆向工程

1. 逆向工程简介

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Java实体类、Mapper接口、Mapper映射文件等资源。

2. 创建步骤

(1) 第一步：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>mysql-connector-
java</artifactId>
    <version>8.0.27</version>
</dependency>
</dependencies>
</plugin>
</plugins>
</build>

```

(2) 第二步：创建MyBatis核心配置文件

常规配置，因为这步和逆向工程无关

(3) 第三步：创建逆向工程的配置文件

文件名必须是：`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.mybatisplus"
targetProject=".\\src\\main\\java">
        <property name="enableSubPackages" value="true" />
    <!--是否可以使用子包-->
        <property name="trimStrings" value="true" />
    <!--去掉字段前后的空格来生成属性-->
    </javaModelGenerator>
    <!-- SQL映射文件的生成策略 -->

```

```

<sqlMapGenerator
    targetPackage="com.atguigu.mybatisplus.mapper"
        targetProject=".\\src\\main\\resources">
    <property name="enableSubPackages" value="true" />
</sqlMapGenerator>
<!-- Mapper接口的生成策略 -->
<javaClientGenerator type="XMLMAPPER"

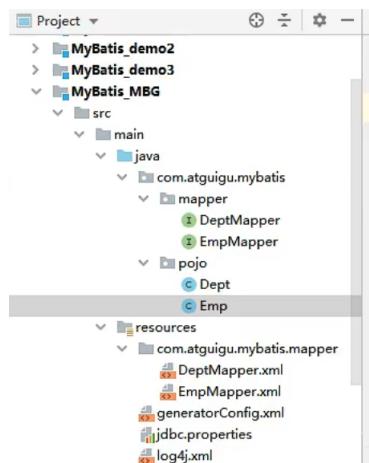
targetPackage="com.atguigu.mybatisplus.mapper"
targetProject=".\\src\\main\\java">
    <property name="enableSubPackages" value="true" />
</javaClientGenerator>
<!-- 逆向分析的表 -->
<!-- tableName设置为*号，可以对应所有表，此时不写
domainObjectName -->
    <!-- domainObjectName属性指定生成出来的实体类的类名 -->
    <table tableName="t_emp" domainObjectName="Emp"/>
<!--映射文件和接口名无需指定-->
    <table tableName="t_dept" domainObjectName="Dept"/>
</context>
</generatorConfiguration>

```

(4) 第四步：启动maven中的插件并执行



双击启动后会出现对应的实体类，映射文件和Mapper接口：



分页插件

使用步骤

1. 步骤1：添加依赖

在pom.xml中添加依赖

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

2. 步骤2：配置分页插件

在MyBatis核心配置文件中配置插件。

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



分页插件的使用

1. 开启分页

在查询之前使用 `PageHelper.startPage(int pageNum, int pageSize)` 开启分页功能。

```

@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2页，每页四条数据
    PageHelper.startPage(2, 4);

    List<Emp> emps = mapper.selectByExample(null);
    emps.forEach(System.out::println);
}

```

底层执行的语句和查询结果如下：

```

DEBUG 11-30 19:22:51,751 Cache Hit Ratio [SQL_CACHE]: 0.0 (LoggingCache.java:60)
DEBUG 11-30 19:22:51,812 ==> Preparing: SELECT count(0) FROM t_emp (BaseJdbcLogger.java:137)
DEBUG 11-30 19:22:51,835 ==> Parameters: (BaseJdbcLogger.java:137)
DEBUG 11-30 19:22:51,850 <==      Total: 1 (BaseJdbcLogger.java:137)
DEBUG 11-30 19:22:51,853 ==> Preparing: select eid, emp_name, age, sex, email, did from t_emp LIMIT ?, ? (BaseJdbcLogger.java:137)
DEBUG 11-30 19:22:51,854 ==> Parameters: 4(Long), 4(Integer) (BaseJdbcLogger.java:137)
DEBUG 11-30 19:22:51,856 <==      Total: 4 (BaseJdbcLogger.java:137)
Emp{eid=5, empName='田七', age=28, sex='男', email='123@qq.com', did=2}
Emp{eid=9, empName='a', age=null, sex='null', email='null', did=null}
Emp{eid=10, empName='a', age=null, sex='null', email='null', did=null}
Emp{eid=11, empName='a', age=null, sex='null', email='null', did=null}

```

2. 分页相关数据查看

(1) 方式一：用Page对象接收分页结果，直接查看

```

@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2页，每页四条数据
    PageHelper.startPage(2, 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);  
}
```

可以得到分页结果：

```
Page{count=true, pageNum=2, pageSize=4, startRow=4, endRow=8,  
total=12, pages=3, reasonable=false, pageSizeZero=false}  
[Emp{eid=5, empName='admin', age=22, sex='男',  
email='456@qq.com', did=3}, Emp{eid=9, empName='admin2',  
age=22, sex='男', email='456@qq.com', did=3}, Emp{eid=10,  
empName='王五', age=12, sex='女', email='123@qq.com', did=3},  
Emp{eid=11, empName='赵六', age=32, sex='男',  
email='123@qq.com', did=1}]
```

(2) 方式二： PageInfo 获取分页相关所有数据

方式一只能获取分页相关的部分数据，方式二可以获得所有数据。

语法： `PageInfo<T> pageInfo = new PageInfo<>(List<T> list,
intnavigatePages)`

- list: 分页之后的数据
- navigatePages: 导航分页的页码数（比如网页中分页查询时能一次性看到34567 5个页码，那么页码数就是5）

```
@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(2, 4);  
    List<Emp> emps = mapper.selectByExample(null);  
    PageInfo<Emp> page = new PageInfo<>(emps, 5);  
    System.out.println(page);  
}
```

获得的数据如下，其中list就是方式一获得的Page对象：

```
PageInfo{  
    pageNum=2, pageSize=4, size=4, startRow=5, endRow=8, total=12, pages=3,  
    list=Page{count=true, pageNum=2, pageSize=4, startRow=4, endRow=8, total=12, pages=3,  
    reasonable=false, pageSizeZero=false}[Emp{eid=5, empName='田七', age=28, sex='男',  
    email='123@qq.com', did=2}, Emp{eid=9, empName='a', age=null, sex='null', email='null', did=null},  
    Emp{eid=10, empName='a', age=null, sex='null', email='null', did=null}, Emp{eid=11, empName='a',  
    age=null, sex='null', email='null', did=null}],  
    prePage=1, nextPage=3, isFirstPage=false, isLastPage=false, hasPreviousPage=true, hasNextPage=true,  
    navigatePages=5, navigateFirstPage=1, navigateLastPage=3, navigatepageNums=[1, 2, 3]}
```

3. 分页相关数据解释

- 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]