

## MyBatis-Plus

### 前言：

在持久层框架中 mybatis 应用比较多，而且比重在逐渐的上升。通常项目的组合是 SSM。mybatis 之所以火，是因为他的灵活，使用方便，优化比较容易。

mybatis 的直接执行 sql 语句，sql 语句是写在 xml 文件中，使用 mybatis 需要多个 xml 配置文件，在一定程度上比较繁琐。一般数据库的操作都要涉及到 CURD。

mybatis-plus 是在 mybatis 上的增强，减少了 xml 的配置，几乎不用编写 xml 就可以做到单表的 CURD,很是方便，极大提供了开发的效率。我们写程序目的就是让生活更加简单。

### 一、什么是 mybatis-plus

MyBatis-Plus (简称 MP) 是一个 MyBatis 的增强工具，在 MyBatis 的基础上只做增强不做改变，为简化开发、提高效率而生。

MyBatis-Plus 在 MyBatis 之上套了一层外衣，单表 CURD 的操作几乎都可以由 MyBatis-Plus 代替执行。而且提供了各种查询方式，分页行为。作为使用者无需编写 xml,直接调用 MyBatis-Plus 提供的 API 就可以了。

官网：<http://mp.baomidou.com/>

## 二、快速开始

几分钟就可以上手 MP,前提是需要熟悉 mybatis, spring 或 spring boot, maven, 掌握 lambda 表达式更能提升效率。

准备环境:

- 拥有 Java 开发环境以及相应 IDE
- 熟悉 Spring Boot
- 熟悉 Maven

课堂的开发环境: IntelliJ IDEA 2018 Ultimate , MySQL 5.7 , Maven 3 , Spring Boot 2.x。

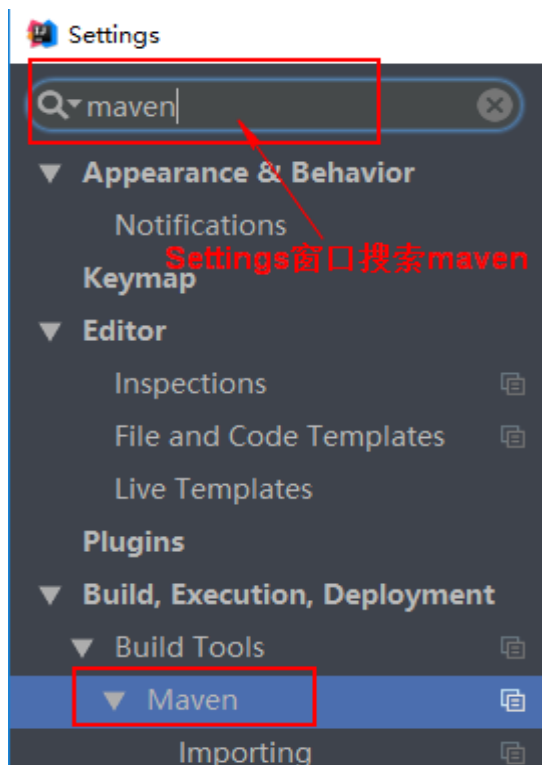
### 1、准备数据表

字段	索引	外键	触发器	选项	注释	SQL 预览				
名					类型	长度	小数点	不是 null		
id					int	11	0	<input checked="" type="checkbox"/>	🔑 1	
name					varchar	50	0	<input type="checkbox"/>		
email					varchar	80	0	<input type="checkbox"/>		
▶ age					int	11	0	<input type="checkbox"/>		

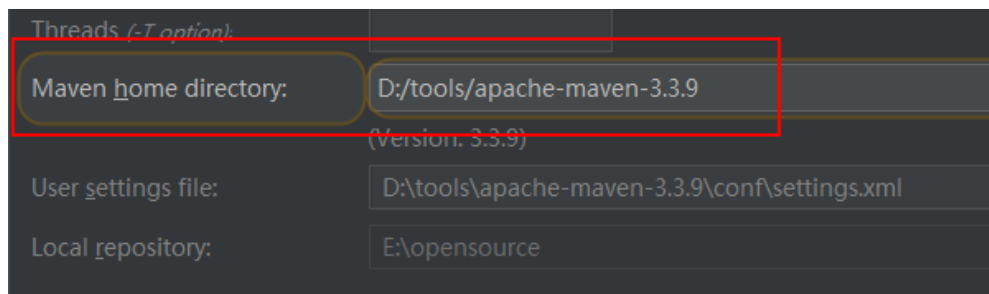
建表 sql:

```
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(50) DEFAULT NULL,
  `email` varchar(80) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

## 2、idea 中配置 maven



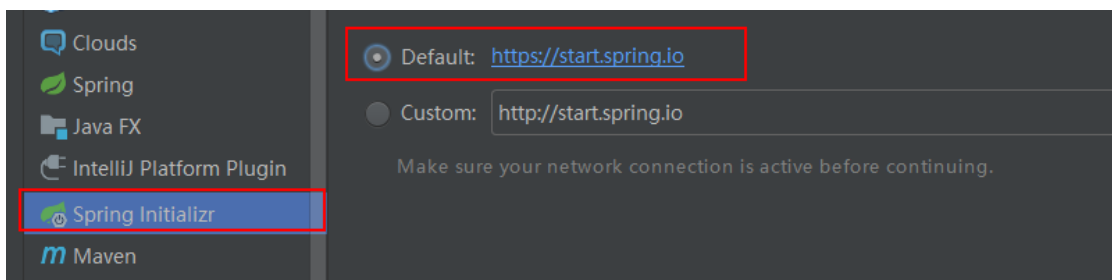
设置 maven 本地仓库地址



## 3、创建 spring boot 应用

使用 Spring Initializr 快速初始化一个 Spring Boot 工程

idea 中 File – New Project 选择 Spring Initializr 后填写项目信息，创建工程。



项目信息：

**Project Metadata**

Group:

Artifact:

Type:

Language:

Packaging:

Java Version:

Version:

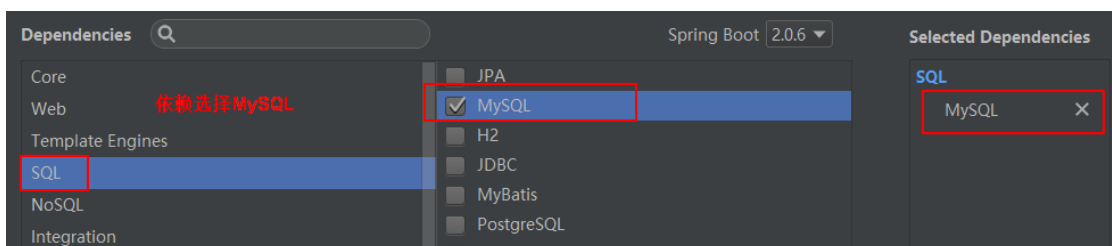
Name:

Description:

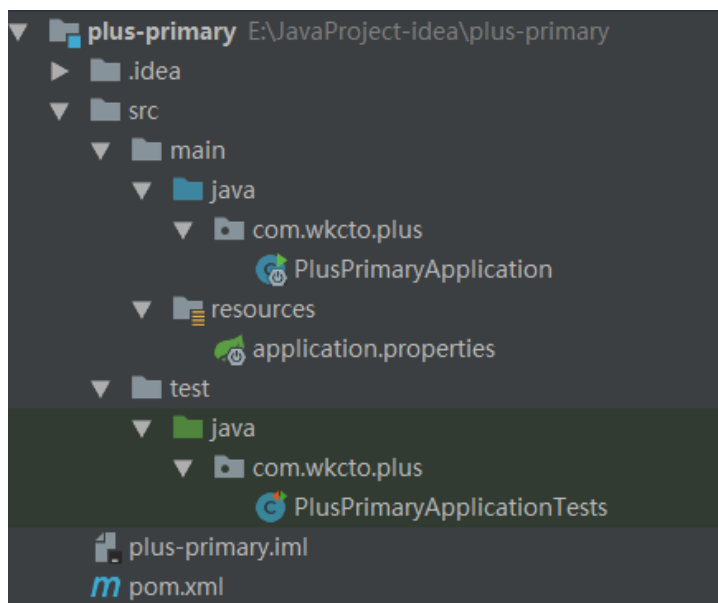
Package:

包名我手工改了 plus

选择依赖项目 MySQL



创建好的项目结构：



## 5、pom.xml

```
<dependency>
  <groupId>com.baomidou</groupId>
  <artifactId>mybatis-plus-boot-starter</artifactId>
  <version>3.0.5</version>
</dependency>
```

## 6、application.yml 添加数据库配置

```
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://127.0.0.1:3306/springdb?useUnicode=true&characterEncoding=UTF-8
    username: root
    password: 123456
```

## 7、实体 entity

```
// 实体类：属性和列名一样
public class User {
    //指定主键
    @TableId(value = "id",type = IdType.AUTO)
    private Integer id;
    private String name;
    private String email;
    private int age;
    // set | get | toString()
    public User() {
    }
    public User(String name, String email, int age) {
        this.name = name;
        this.email = email;
        this.age = age;
    }
}
```

@TableId 设置主键， IdType.AUTO 使用自动增长产生主键

## 8、mapper

```
package com.wkcto.plus.dao;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.wkcto.plus.entity.User;

public interface UserMapper extends BaseMapper<User> {
}
```

继承 MyBatis Plus 中的 BaseMapper ， 在 UserMapper 中使用 MP 中的方法，实现 CURD。

## 9、添加@MapperScan 扫描 Mapper 文件夹

```
@SpringBootApplication
@MapperScan(
    value = "com.wkcto.plus.dao"
)
public class PlusPrimaryApplication {

    public static void main(String[] args) {

        SpringApplication.run(PlusPrimaryApplication.class, args);

    }

}
```

## 10、测试

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class PlusPrimaryApplicationTests {

    @Autowired
    private UserMapper userDao;

    @Test
    public void testInsertUser(){

        User user = new User("张三","zhangsan@126.com",20);
        int rows = userDao.insert(user);
        System.out.println("insert user result:"+rows);

    }

}
```

## 三、配置 mybatis 日志

application.yml

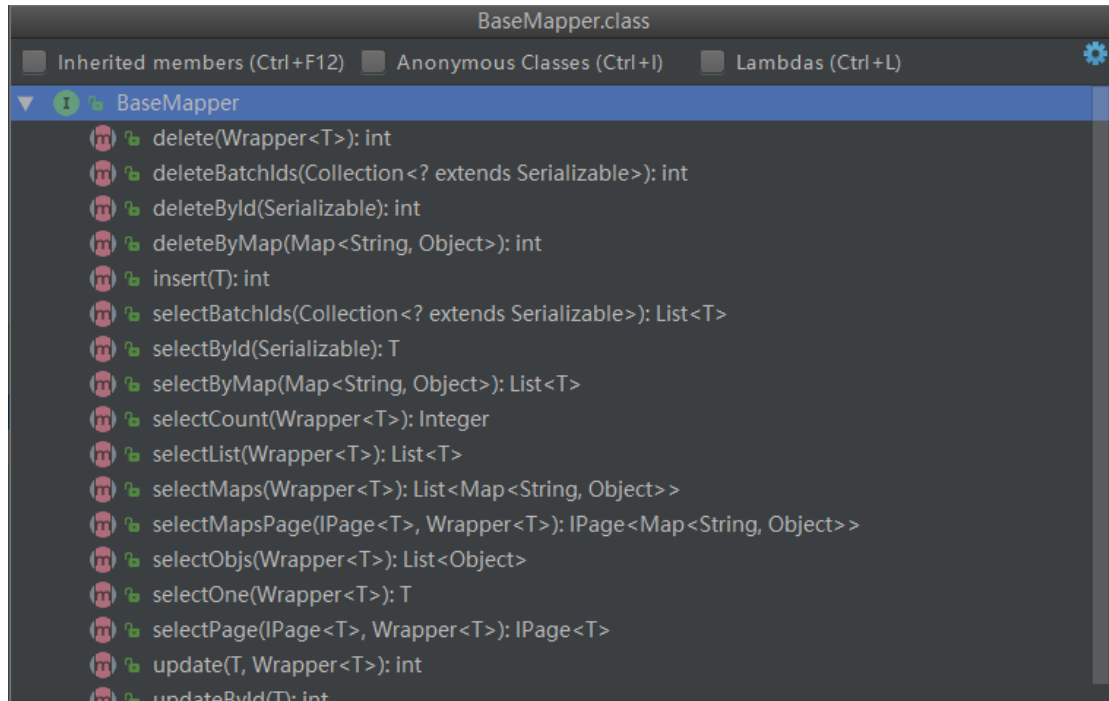
```
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdoutImpl
```

## 四、CRUD 基本用法

CRUD 的操作是来自 BaseMapper 中的方法。BaseMapper 中共有 17 个方法，

CRUD 操作都有多个不同参数的方法。继承 BaseMapper 可以其中的方法。

BaseMapper 方法列表：



## 1、insert 操作

```
@Test
public void testInsertUser(){
    User user = new User("张三","zhangsan@126.com",20);
    int rows = userDao.insert(user);
    System.out.println("insert user result:"+rows);
}
```

注：insert()返回值 int，数据插入成功的行数，成功的记录数。getId()获取主键值



## 2、update 操作

```
/**
 * 根据id更新记录
 */
@Test
public void testUpdate(){
    User user = new User("zs","zs@163.com",20);
    user.setId(1);
    int rows = userDao.updateById(user);
    System.out.println("update user rows:"+rows);
}
```

注意：null 的字段不更新

```
/**根据id更新记录 */
@Test
public void testUpdate(){
    User user = new User();
    System.out.println("user:"+user);
    user.setId(1);
    user.setName("lisi");
    int rows = userDao.updateById(user);
    System.out.println("update user rows:"+rows);
}
```

日志：

```
user:User{id=null, name='null', email='null', age=0}
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSe
JDBC Connection [HikariProxyConnection@1862900975 wrapping
==> Preparing: UPDATE user SET name=?, age=? WHERE id=?
==> Parameters: lisi(String), 0(Integer), 1(Integer)
<==      Updates: 1
```

email 没有赋值，是 null，所有没有出现在 set 语句中；age 有默认 0，被更新了。

## 3、delete 操作

删除有多个方法：

```

}
/** 根据id删除 */
@Test
public void testDelete() {
    User user = new User();
    user.setId(1);
    int rows = userDao.deleteById(1);
    System.out.println("update user rows:" + rows);
}

```

delete(Wrapper<User> wrapper) int  
deleteBatchIds(Collection<? extends Serializable> ids) int  
deleteById(Serializable serializable) int  
deleteByMap(Map<String, Object> map) int  
hashCode() int

(1) deleteById:按主键删除，其他方法后面介绍

```

/** 根据id删除 */
@Test
public void testDelete() {
    int rows = userDao.deleteById(1);
    System.out.println("update user rows:" + rows);
}

```

(2) 根据 Map 中条件删除

```

/** 根据Map删除 */
@Test
public void testDeleteByMap() {
    Map<String, Object> param = new HashMap<>();
    param.put("age", 24);
    param.put("name", "张三3");
    int rows = userDao.deleteByMap(param);
    System.out.println("update user rows:" + rows);
}

```

注：删除条件封装在 Map 中，key 是列名，value 是值，多个 key 之间 and 联接。

日志：

```

==> Preparing: DELETE FROM user WHERE name = ? AND age = ?
==> Parameters: 张三3(String), 24(Integer)
<== Updates: 0

```

(3) 批量删除

```

/** 批量删除 */
@Test
public void testDeleteBatch() {
    List<Integer> idList = Stream.of(1, 2, 3, 4, 5)
        .collect(Collectors.toList());
    int rows = userDao.deleteBatchIds(idList);
    System.out.println("update user rows:" + rows);
}

```

注：list 集合的创建可以使用 lambda 表达式，也可以使用 add()。例如：

idlist.add(1); idlist.add(2);idlist.add(3)...

把要删除的 id 放入 List ，传给 deleteBatchIds()。 批量操作使用 in(...)

```
==> Preparing: DELETE FROM user WHERE id IN ( ? , ? , ? , ? , ? )
==> Parameters: 1(Integer), 2(Integer), 3(Integer), 4(Integer), 5(Integer)
<== Updates: 2
```

## 4、select 操作

### (1) 根据 id 主键查询

```
/**主键查询*/
@Test
public void testSelectById(){
    User user = userDao.selectById(6);
    System.out.println("selectById:"+user);
}
```

注：没有查询结果，不会报错。

日志：

```
==> Preparing: SELECT id,name,email,age FROM user WHERE id=?
==> Parameters: 6(Integer)
<== Columns: id, name, email, age
<== Row: 6, 张三2, zhangsan@126.com, 22
<== Total: 1
```

### (2) 批量查询记录

```
/**根据id批量查询*/
@Test
public void testSelectBatchById(){
    List<Integer> idlist = new ArrayList<>();
    idlist.add(1);
    idlist.add(2);
    List<User> users = userDao.selectBatchIds(idlist);
    System.out.println("selectBatchIds:"+users.size());
}
```

注：根据 id 查询记录，把需要查询的多个 id 存入到 List，调用 selectBatchIds(),

传入 List，返回值也是 List。 查询条件是 from user where in id (1,2)

### (3) 使用 Map 的条件查询

```
/**根据Map作为条件查询*/
@Test
public void testSelectByMap(){
    Map<String,Object> param = new HashMap<>();
    param.put("age", 20);
    param.put("email", "zhangsan@163.com");
    List<User> users = userDao.selectByMap(param);
    System.out.println("selectBatchIds:"+users.size());
}
```

把要查询的条件字段 put 到 Map, key 是字段, value 是条件值。多个条件是 and 联接。调用 selectByMap(),传入 Map 作为参数, 返回值是 List 集合。

```
==> Preparing: SELECT id,name,email,age FROM user WHERE age = ? AND email = ?
==> Parameters: 20(Integer), zhangsan@163.com(String)
<==      Total: 0
```

更多的查询方式, 在后面章节作为专题讲解。包括条件对象 Wrapper, lambda 表达式, 分页查询等等。


## 五、ActiveRecord (AR)

ActiveRecord 是什么:

- 每一个数据库表对应创建一个类, 类的每一个对象实例对应于数据库中表的一行记录; 通常表的每个字段在类中都有相应的 Field;
- ActiveRecord 负责把自己持久化. 在 ActiveRecord 中封装了对数据库的访问, 通过对象自己实现 CRUD, 实现优雅的数据库操作。
- ActiveRecord 也封装了部分业务逻辑。可以作为业务对象使用。

## 1、AR 之 insert

### (1) dept 表设计

字段	索引	外键	触发器	选项	注释	SQL 预览
名			类型	长度	小数点	不是 null
id			int	11	0	<input checked="" type="checkbox"/>  1
name			varchar	50	0	<input type="checkbox"/>
mobile			varchar	50	0	<input type="checkbox"/>
▶ manager			int	11	0	<input type="checkbox"/>

sql:

```
DROP TABLE IF EXISTS `dept`;
CREATE TABLE `dept` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(50) DEFAULT NULL,
  `mobile` varchar(50) DEFAULT NULL,
  `manager` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

### (2) entity 实体类

```
public class Dept extends Model<Dept> {
    //指定主键
    @TableId(value = "id",type = IdType.AUTO)
    private Integer id;
    private String name;
    private String mobile;
    private Integer manager;
    // set | get | toString
}
```

必须继承 Model,Model 定义了表的 CRUD 方法, Dept 属性名和列名是一样的。

### (3) mapper

```
package com.wkcto.plus.dao;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.wkcto.plus.entity.Dept;

public interface DeptMapper extends BaseMapper<Dept> {
}
```

不使用 mapper, 也需要定义这个类, MP 通过 mapper 获取到表的结构; 不定义

时，MP 报错无法获取表的结构信息。

#### (4)测试 AR

```
@Test
public void testARInsert(){
    Dept dept = new Dept();
    dept.setName("销售部");
    dept.setMobile("010-45667789");
    dept.setManager(1);

    boolean result = dept.insert();
    System.out.println("AR Insert : "+result);
}
```

## 2、AR 之 update

```
/**按主键更新*/
@Test
public void testARUpdate(){
    Dept dept = new Dept();
    dept.setId(5);
    dept.setName("销售部改为市场部1111");
    //没有赋值mobile , null的属性值不更新
    //dept.setMobile("010-22222222");
    dept.setManager(2);

    boolean result = dept.updateById();
    System.out.println("AR Update : "+result);
}
```

创建实体对象，对要更新的属性赋值，null 的属性不更新，根据主键更新记录。

返回值是 boolean，true 更新成功。没有更新记录是 false。

日志：

```
=> Preparing: UPDATE dept SET name=?, manager=? WHERE id=?
=> Parameters: 销售部改为市场部(String), 2(Integer), 1(Integer)
<= Updates: 1
```

### 3、AR 之 delete

```
/**根据主键查询*/
@Test
public void testARDelete(){
    Dept dept = new Dept();
    boolean result = dept.deleteById(1);
    System.out.println("AR delete :" + result);
}
```

使用主键作为删除条件, deleteById()参数是主键值, sql 语句条件是 where id=1。

返回值始终是 true。通过源码查看：

```
public static boolean delBool(Integer result) {
    return null != result && result >= 0;
}
```

删除返回值判断条件是 result >=0 , 只有 sql 语法是正确的, 返回就是 true。和删除记录的数量无关。

日志：

```
==> Preparing: DELETE FROM dept WHERE id=?
==> Parameters: 1(Integer)
<== Updates: 1
Closing non transactional SqlSession [org.apa
AR delete :true
```

### 4、AR 之 select

(1)对象调用 selectById()

```
/**根据对象自己的id属性查询*/
@Test
public void testARSelect(){
    Dept dept = new Dept();
    dept.setId(2);
    Dept selectDept = dept.selectById();
    System.out.println("AR select :"+selectDept);
}
```

对象提供主键值, 调用 selectById()无参数, 使用 id=2 作为查询条件, 返回值是

查询的结果对象；没有查询到对象，返回是 null；不提供主键 id 值，报错如下：

com.baomidou.mybatisplus.core.exceptions.MybatisPlusException: selectById  
primaryKey is null.

## (2)selectById(主键)

```
/**根据参数id值查询*/  
@Test  
public void testARSelectById(){  
    Dept dept = new Dept();  
    //dept.setId(2); 不设置id值  
    Dept selectDept = dept.selectById(1); //参数值作为查询条件  
    System.out.println("AR selectById :"+selectDept);  
}
```

创建对象，不设值 id 主键值， selectById 的参数是查询条件，和对象的属性值无关。返回值是结果对象，id 不存在返回 null。

## (3)其它方法

查询操作是最多的，其它方法的使用在介绍 Wrapper 对象后讲解。

```
/**根据  
@Test  
public  
    Dept  
    Dept  
    List<Dept>  
    int  
    List<Dept>  
    Dept  
    IPage<Dept>  
    dept.sel  
}
```

## 六、表和列

主键 ,TableName, TableId

### 1、主键类型

IdType 枚举类，主键定义如下：



```
public enum IdType {
    AUTO(0),
    NONE(1),
    INPUT(2),
    ID_WORKER(3),
    UUID(4),
    ID_WORKER_STR(5);
}
```

- 0. none 没有主键
- 1. auto 自动增长(mysql, sql server)
- 2. input 手工输入
- 3. id\_worker: 实体类用 Long id , 表的列用 bigint , int 类型大小不够
- 4. id\_worker\_str 实体类使用 String id, 表的列使用 varchar 50
- 5. uuid 实体类使用 String id, 列使用 varchar 50

id\_worker: Twitter 雪花算法-分布式 ID

## 2、指定表名

定义实体类，默认的表名和实体类同名；如果不一致，在实体类定义上面使用

@TableName 说明表名称。

例如：@TableName(value="数据库表名")

步骤：

(1) 表

字段	索引	外键	触发器	选项	注释	SQL 预览			
名			类型	长度	小数点	不是 null			
▶ id			int	11	0	<input checked="" type="checkbox"/>	🔑 1		
city			varchar	50	0	<input type="checkbox"/>			
street			varchar	255	0	<input type="checkbox"/>			
zipcode			varchar	255	0	<input type="checkbox"/>			

(2) entity

```
//value:表示数据库表名
@TableName(value = "user_address")
public class Address {

    @TableId(value = "id",type = IdType.AUTO)
    private Integer id;
    private String city;
    private String street;
    private String zipcode;
    //set | get
}
```

(3) mapper

```
public interface AddressMapper extends BaseMapper<Address>{
}
```

(4) 测试

注入 mapper 对象

```
@Autowired
private AddressMapper addressDao;
```

insert 记录:

```
@Test
public void testTableName(){
    Address address = new Address();
    address.setCity("北京");
    address.setStreet("长安街");
    address.setZipcode("010");
    int rows = addressDao.insert(address);
    System.out.println("insert rows:"+rows);
}
```

日志:

```
==> Preparing: INSERT INTO user_address ( city, street, zipcode )
==> Parameters: 北京(String), 长安街(String), 010(String)
<== Updates: 1
```

### 3、指定列名

#### (1) 表

字段	索引	外键	触发器	选项	注释	SQL 预览	
名			类型	长度	小数点	不是 null	
id			int	11	0	<input checked="" type="checkbox"/>	 1
empid			int	11	0	<input checked="" type="checkbox"/>	
empsal			float	10	2	<input checked="" type="checkbox"/>	

#### (2) entity

```
@TableName("Salary")
public class Salary {
    @TableId(value = "id", type = IdType.AUTO)
    private Integer id;
    // @TableField 注解value是字段名称
    @TableField(value = "empid")
    private Integer employid;
    @TableField(value = "empsal")
    private float employsal;
    // set | get
}
```

#### (3) mapper

```
public interface SalaryMapper extends BaseMapper<Salary> {
}
```

#### (4) 测试

```
/**测试属性和字段不同*/
@Test
public void testTableField(){
    Salary sal = new Salary();
    sal.setEmployid(1);
    sal.setEmploysal(5000);
    int rows = salDao.insert(sal);
    System.out.println("Table Field rows:"+rows);
}
```

日志：

```
Preparing: INSERT INTO Salary ( empid, empsal ) VALUES ( ?, ? )
Parameters: 1(Integer), 5000.0(Float)
Updates: 1
```

#### 4、驼峰命名

列名使用下划线,属性名是驼峰命名方式。MyBatis 默认支持这种规则。

##### (1) 表定义

字段	索引	外键	触发器	选项	注释	SQL 预览			
名					类型	长度	小数点	不是 null	
▶ id					int	11	0	<input checked="" type="checkbox"/>	 1
cust_name					varchar	50	0	<input type="checkbox"/>	
cust_age					int	11	0	<input type="checkbox"/>	
cust_email					varchar	100	0	<input type="checkbox"/>	

##### (2) entity

```
public class Customer {
    @TableId(value = "id",type = IdType.AUTO)
    private Integer id;
    private String custName;
    private String custEmail;
    private Integer custAge;
    // set | get
}
```

##### (3) mapper

```
public interface CustomerMapper extends BaseMapper<Customer> {
}
```

##### (4)测试

```

@Autowired
private CustomerMapper custDao;

@Test
public void testInsertCust(){
    Customer cust = new Customer();
    cust.setCustAge(20);
    cust.setCustEmail("zhangsan@sina.com");
    cust.setCustName("zhangsan");
    int rows = custDao.insert(cust);
    System.out.println("insert cust rows:"+rows);
}

```

日志:

```

Preparing: INSERT INTO customer ( cust_name, cust_email, cust_age ) VALUES ( ?, ?, ? )
Parameters: zhangsan(String), zhangsan@sina.com(String), 20(Integer)
Updates: 1

```

## 七、自定义 sql

### 1、表定义

字段	索引	外键	触发器	选项	注释	SQL 预览	
名						长度	小数点
id						11	0
name						80	0
age						11	0
email						80	0
status						11	0

### 2、创建实体

```

public class Student {
    @TableId(value = "id",type = IdType.AUTO)
    private Integer id;
    private String name;
    private Integer age;
    private String email;
    private Integer status;
    // set | get
}

```

### 3、创建 Mapper

```
public interface StudentMapper extends BaseMapper<Student> {  
    List<Student> selectByName();  
}
```

### 4、新建 sql 映射 xml 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="com.wkcto.plus.dao.StudentMapper">  
  
    <select id="selectByName" resultType="com.wkcto.plus.entity.Student">  
        SELECT id, name, age, email, status FROM student  
        ORDER BY id LIMIT 10  
    </select>  
</mapper>
```

### 5、配置 xml 文件位置

application.yml

```
mybatis-plus:  
  configuration:  
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl  
  mapper-locations: classpath*:xml/*Mapper.xml
```

### 6、测试

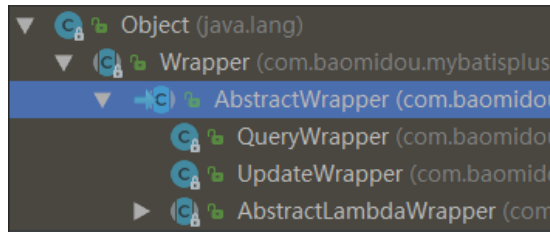
```
@Test  
public void testSelectByName(){  
    List<Student> studentList = studentMapper.selectByName();  
    studentList.forEach(stu -> {  
        System.out.println(stu);  
    });  
}
```

日志:

```
=> Preparing: SELECT id, name, age, email, status FROM student ORDER BY id LIMIT 10  
=> Parameters:  
<== Columns: id, name, age, email, status  
<== Row: 1, 张三, 22, zs@sina.com, 1  
<== Total: 1
```

## 八、查询和分页

### 1、查询构造器：Wrapper



QueryWrapper(LambdaQueryWrapper) 和 UpdateWrapper(LambdaUpdateWrapper) 的父类用于生成 sql 的 where 条件, entity 属性也用于生成 sql 的 where 条件. MP3.x开始支持 lambda 表达式, LambdaQueryWrapper, LambdaUpdateWrapper 支持 lambda 表达式的构造查询条件。

条件：

条件	说明
allEq	基于 map 的相等
eq	等于 =
ne	不等于 <>
gt	大于 >
ge	大于等于 >=
lt	小于 <
le	小于等于 <=
between	BETWEEN 值 1 AND 值 2
notBetween	NOT BETWEEN 值 1 AND 值 2

like	LIKE '%值%'
notLike	NOT LIKE '%值%'
likeLeft	LIKE '值'
likeRight	LIKE '值%'
isNull	字段 IS NULL
isNotNull	字段 IS NOT NULL
in	字段 IN (value1, value2, ...)
notIn	字段 NOT IN (value1, value2, ...)
inSql	字段 IN ( sql 语句 )  例: inSql("age", "1,2,3")--->age in (1,2,3,4,5,6) 例: inSql("id", "select id from table where id < 3")--->id in (select id from table where id < 3)
notInSql	字段 NOT IN ( sql 语句 )
groupBy	GROUP BY 字段
orderByAsc	升序 ORDER BY 字段, ... ASC
orderByDesc	降序 ORDER BY 字段, ... DESC
orderBy	自定义字段排序  orderBy(true, true, "id", "name")--->order by id ASC,name ASC
having	条件分组
or	OR 语句, 拼接 + OR 字段=值
and	AND 语句, 拼接 + AND 字段=值
apply	拼接 sql



last	在 sql 语句后拼接自定义条件
exists	拼接 EXISTS ( sql 语句 ) 例：exists("select id from table where age = 1")--->exists (select id from table where age = 1)
notExists	拼接 NOT EXISTS ( sql 语句 )
nested	正常嵌套 不带 AND 或者 OR

### 1)、QueryWrapper: 查询条件封装类

方法	说明
select	设置查询字段 select 后面的内容

### 2)、UpdateWrapper:更新条件封装类

方法	说明
set	设置要更新的字段, MP 拼接 sql 语句
setSql	参数是 sql 语句, MP 不在处理语句

## 2、查询

Student 表：初始数据

id	name	age	email	status
1	张三	22	zs@sina.com	1
2	李四	29	lisi@163.com	1
3	李雷	31	liei@163.com	2
4	周丽	36	zhouli@sina.com	2
5	周名明	24	(Null)	2
6	冯亮	32	(Null)	2

### 1)、allEq

以 Map 为参数条件

a) 条件：name 是张三，age 是 22

```
@Test
public void testAllEq(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    Map<String,Object> tj = new HashMap<>();
    tj.put("name", "张三");
    tj.put("age", 22);
    qw.allEq(tj);
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}
```

日志：where name="张三" and age = 22

```
Preparing: SELECT id,name,age,email,status FROM student WHERE name = ? AND age = ?
Parameters: 张三(String), 22(Integer)
Columns: id, name, age, email, status
Row: 1, 张三, 22, zs@sina.com, 1
Total: 1
```

b) 查询条件有 null

```
@Test
public void testAllEq2(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    Map<String,Object> tj = new HashMap<>();
    tj.put("name", "张三");
    //age 是null
    tj.put("age", null);
    //构建 qw 第二个参数为 true
    qw.allEq(tj,true);
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}
```

true 日志:

```
FROM student WHERE name = ? AND age IS NULL
```

false 日志:

```
FROM student WHERE name = ?
```

## 2)、eq

等于 =

name 等于李四

```
@Test
public void testEq(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    // eq(列名, 值)
    qw.eq("name", "李四");
    //qw.eq("age", 22); 多个eq是 and 连接条件
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}
```

日志:

```
FROM student WHERE name = ?
```

## 3)、ne

ne 不等于

```
@Test
public void testNe(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    // ne(列名, 值)
    qw.ne("name", "李四");
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}
```

日志:

```
FROM student WHERE name <> ?
```

#### 4)、gt

gt 大于

```
@Test
public void testGt(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.gt("age", 20);
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}
```

日志:

```
FROM student WHERE age > ?
```

#### 5)、ge

ge 大于等于

```
@Test
public void testGe(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.ge("age", 22);
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}
```

日志:

```
FROM student WHERE age >= ?
```

#### 6)、lt

lt 小于

```

@Test
public void testLt(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.lt("age", 20);
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

日志:

```
FROM student WHERE age < ?
```

### 7)、le

le 小于等于 <=

```

@Test
public void testLe(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.le("age", 20);
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

日志:

```
FROM student WHERE age <= ?
```

### 8)、between

between 在两个值范围之间

```

@Test
public void testBetween(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.between("age", 18, 28);
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

日志

```
FROM student WHERE age BETWEEN ? AND ?
```

## 9)、notBetween

notBetween 不在两个值范围之间

```
@Test
public void testNotBetween(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.notBetween("age", 18, 28);
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}
```

日志:

```
FROM student WHERE age NOT BETWEEN ? AND ?
```

## 10)、like , notLike

like 匹配值 “%值%”

notLike 不匹配 “%值%”

```
@Test
public void testLike(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.like("name", "张");
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}
```

日志:

```
Preparing: SELECT id,name,age,email,status FROM student WHERE name LIKE ?
Parameters: %张%(String)
```

## 11)、likeLeft, likeRight

likeLeft 匹配 like “%值”

likeRight 匹配 like “值%”

```

@Test
public void testLikeLeft(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.likeLeft("name", "张");
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

日志:

```

==> Preparing: SELECT id,name,age,email,status FROM student WHERE name LIKE ?
==> Parameters: %张(String)

```

## 12)、isNull , isNotNull

isNull 判断字段值为 null

isNotNull 字段值不为 null

```

@Test
public void testIsNull(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.isNull("age");
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

日志:

```

FROM student WHERE age IS NULL

```

## 13)、in , notIn

in 后面值列表， 在列表中都是符合条件的。

notIn 不在列表中的

```

@Test
public void testIn(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.in("name", "张三", "李四");
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

```

@Test
public void testIn2(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    List<Integer> list = new ArrayList<>();
    list.add(1);
    list.add(2);
    qw.in("status", list);
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

日志:

```

Preparing: SELECT id,name,age,email,status FROM student WHERE name IN (?,?)
Parameters: 张三(String), 李四(String)

```

#### 14)、inSql, notInSql

inSql 常用来做子查询 类似 in()

notInSql 类似 notIn()

```

@Test
public void testInSql(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.inSql("age", "select age from student where id=1");
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

日志:

```

Preparing: SELECT id,name,age,email,status FROM student WHERE age IN (select age from student where id=1)

```

#### 15)、groupBy

groupBy 基于多个字段分组



```

@Test
public void testGroupBy(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.select("name,count(*) personNumbers ");
    qw.groupBy("name");
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

日志:

```

Preparing: SELECT name,count(*) personNumbers FROM student GROUP BY name
Parameters:
Columns: name, personNumbers
Row: 冯亮, 1
Row: 周丽, 1

```

## 16)、orderByAsc ,orderyByDesc , orderBy

orderByAsc 按字段升序

orderByDesc 按字段降序

orderBy 每个字段指定排序方向

```

@Test
public void testOrderBy(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    //可以使用多列排序
    qw.orderByAsc("name","age");
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

日志:

```

FROM student ORDER BY name ASC , age ASC

```

## 17)、 or , and

or 连接条件用 or, 默认是 and

and 连接条件用 and

```

@Test
public void testOr(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.eq("name", "张三")
        .or()
        .eq("age", 22);
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

日志:

```
FROM student WHERE name = ? OR age = ?
```

#### 18)、last

last 拼接 sql 语句

```

@Test
public void testLast(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.eq("name", "张三")
        .or()
        .eq("age", 22)
        .last("limit 1");
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

日志:

```
FROM student WHERE name = ? OR age = ? limit 1
```

#### 19)、exists ,notExists

exists 拼接 EXISTS ( sql 语句 )

notExists 是 exists 的相反操作

```

@Test
public void testExists(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.exists("select id from student where age > 20");
    List<Student> students = studentMapper.selectList(qw);
    toJson(students);
}

```

日志:

```

EXISTS (select id from student where age > 20)

```

### 3、分页

前提： 配置分页插件，实现物理分页。默认是内存分页

```

@Configuration
public class Config {

    //分页插件
    @Bean
    public PaginationInterceptor paginationInterceptor(){
        return new PaginationInterceptor();
    }
}

```

分页查询:

```

@Test
public void testPage(){
    QueryWrapper<Student> qw = new QueryWrapper<>();
    qw.gt("age", 20);
    qw.orderByAsc("id");
    IPage<Student> page = new Page<>();
    page.setCurrent(1);
    page.setSize(3);
    IPage<Student> pageResult = studentMapper.selectPage(page,qw);

    List<Student> students = pageResult.getRecords();
    System.out.println(students);
    System.out.println("总页数:"+pageResult.getPages());
    System.out.println("总记录数:" + pageResult.getTotal());
    System.out.println("当前页码:"+pageResult.getCurrent());
    System.out.println("每页大小:"+pageResult.getSize());
}

```

日志:

```

FROM student WHERE age > ? ORDER BY id ASC LIMIT 0,3

```

输出：

```
总页数:2  
总记录数:6  
当前页码:1  
每页大小:3
```

## 九、MP 生成器

准备条件：

```
<!-- 模板引擎 -->
```

```
<dependency>
```

```
    <groupId>org.apache.velocity</groupId>
```

```
    <artifactId>velocity-engine-core</artifactId>
```

```
    <version>2.0</version>
```

```
</dependency>
```

创建生成类：

```
public class AtuoGenerate {  
    public static void main(String[] args) {  
        AutoGenerator mg = new AutoGenerator();
```

```
//全局配置  
GlobalConfig gc = new GlobalConfig();  
String projectPath = System.getProperty("user.dir");  
  
gc.setOutputDir(projectPath+"/src/main/java");  
gc.setAuthor("changming");  
gc.setServiceName("%sService");  
gc.setServiceImplName("%sServiceImpl");  
mg.setGlobalConfig(gc);
```

```
//数据源配置
DataSourceConfig db = new DataSourceConfig();
db.setDriverName("com.mysql.jdbc.Driver");
db.setUrl("jdbc:mysql://localhost:3306/springdb");
db.setUsername("root");
db.setPassword("root");
mg.setDataSource(db);
```

```
//package配置
PackageConfig pc = new PackageConfig();
pc.setModuleName("order");
pc.setParent("com.wkcto");
mg.setPackageInfo(pc);

//策略配置
StrategyConfig sc = new StrategyConfig();
sc.setColumnNaming(NamingStrategy.underline_to_camel);
mg.setStrategy(sc);

mg.execute();
```