

# MyBatis-Plus 3.x

## 文档手册

书栈(BookStack.CN)

# 目 录

致谢

快速入门

简介

快速开始

安装

配置

注解

核心功能

代码生成器

CRUD 接口

条件构造器

分页插件

Sequence主键

自定义ID生成器

插件扩展

热加载

逻辑删除

通用枚举

字段类型处理器

自动填充功能

Sql 注入器

攻击 SQL 阻断解析器

性能分析插件

执行 SQL 分析打印

乐观锁插件

动态数据源

分布式事务

多租户 SQL 解析器

动态表名 SQL 解析器

MybatisX 快速开发插件

FAQ

常见问题

捐赠支持

配置

使用配置

基本配置

使用方式

Configuration

GlobalConfig

DbConfig

代码生成器配置

基本配置

数据源 dataSourceConfig 配置

数据库表配置

包名配置

模板配置

全局策略 globalConfig 配置

注入 injectionConfig 配置

## 致谢

当前文档《MyBatis-Plus 3.x 文档手册》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-11-06。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN)，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

内容来源：MyBatis-Plus <https://mybatis.plus/>

文档地址：<http://www.bookstack.cn/books/mybatis-plus-3.x>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

- [简介](#)
- [快速开始](#)
- [安装](#)
- [配置](#)
- [注解](#)

## 简介

[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。



## 愿景

我们的愿景是成为 MyBatis 最好的搭档，就像 魂斗罗 中的 1P、2P，基友搭配，效率翻倍。



**TO BE THE BEST PARTNER OF MYBATIS**

## 特性

- 无侵入：只做增强不做改变，引入它不会对现有工程产生影响，如丝般顺滑

- 损耗小：启动即会自动注入基本 CURD，性能基本无损耗，直接面向对象操作
- 强大的 **CRUD** 操作：内置通用 Mapper、通用 Service，仅仅通过少量配置即可实现单表大部分 CRUD 操作，更有强大的条件构造器，满足各类使用需求
- 支持 **Lambda** 形式调用：通过 Lambda 表达式，方便的编写各类查询条件，无需再担心字段写错
- 支持主键自动生成：支持多达 4 种主键策略（内含分布式唯一 ID 生成器 - Sequence），可自由配置，完美解决主键问题
- 支持 **ActiveRecord** 模式：支持 ActiveRecord 形式调用，实体类只需继承 Model 类即可进行强大的 CRUD 操作
- 支持自定义全局通用操作：支持全局通用方法注入（Write once, use anywhere）
- 内置代码生成器：采用代码或者 Maven 插件可快速生成 Mapper、Model、Service、Controller 层代码，支持模板引擎，更有超多自定义配置等您来使用
- 内置分页插件：基于 MyBatis 物理分页，开发者无需关心具体操作，配置好插件之后，写分页等同于普通 List 查询
- 分页插件支持多种数据库：支持 MySQL、MariaDB、Oracle、DB2、H2、HSQL、SQLite、Postgre、SQLServer 等多种数据库
- 内置性能分析插件：可输出 Sql 语句以及其执行时间，建议开发测试时启用该功能，能快速揪出慢查询
- 内置全局拦截插件：提供全表 delete、update 操作智能分析阻断，也可自定义拦截规则，预防误操作

## 支持数据库

---

- mysql、mariadb、oracle、db2、h2、hsql、sqlite、postgresql、sqlserver
- 达梦数据库、虚谷数据库、人大金仓数据库

## 框架结构

---

framework

## 代码托管

[Gitee](#) | [Github](#)

## 参与贡献

欢迎各路好汉一起来参与完善 MyBatis-Plus，我们期待你的 PR！

- 贡献代码：代码地址 [MyBatis-Plus](#)，欢迎提交 Issue 或者 Pull Requests
- 维护文档：文档地址 [MyBatis-Plus-Doc](#)，欢迎参与翻译和修订

## 优秀视频

第三方录制的优秀视频教程，加入该列表必须是免费教程。

- [MyBatis-Plus 入门](#) - [视频教程](#) - [慕课网](#)
- [MyBatis-Plus 进阶](#) - [视频教程](#) - [慕课网](#)



# 优秀案例

名称登记按照时间先后，需加入列表的同学可以告诉我们。

- [SpringWind](#)<sup>↗</sup>: Java EE ( J2EE ) 快速开发框架
- [Crown](#)<sup>↗</sup>: Mybatisplus 3.0 教学版
- [Crab](#)<sup>↗</sup>: WEB 极速开发框架
- [KangarooAdmin](#)<sup>↗</sup>: 轻量级权限管理框架
- [iBase4J](#)<sup>↗</sup>: Java 分布式快速开发基础平台
- [framework](#)<sup>↗</sup>: 后台管理框架
- [BMS](#)<sup>↗</sup>: 基础权限开发框架
- [spring-shiro-training](#)<sup>↗</sup>: 简单实用的权限脚手架
- [center](#)<sup>↗</sup>: 系统管理中心系统
- [skeleton](#)<sup>↗</sup>: Springboot-Shiro 脚手架
- [springboot\\_mybatisplus](#)<sup>↗</sup>: 基于 SpringBoot 的美女图片爬虫系统
- [guns](#)<sup>↗</sup>: guns 后台管理系统
- [maple](#)<sup>↗</sup>: maple 企业信息化的开发基础平台
- [jeeweb-mybatis](#)<sup>↗</sup>: JeeWeb 敏捷开发平台
- [youngcms](#)<sup>↗</sup>: CMS 平台
- [king-admin](#)<sup>↗</sup>: 前后端分离的基础权限管理后台
- [jeefast](#)<sup>↗</sup>: 前后端分离 Vue 快速开发平台
- [bing-upms](#)<sup>↗</sup>: SpringBoot + Shiro +FreeMarker 制作的通用权限管理
- [slife](#)<sup>↗</sup>: SpringBoot 企业级快速开发脚手架
- [pig](#)<sup>↗</sup>: 微服务 Spring Cloud 架构
- [mysiteforme](#)<sup>↗</sup>: 系统后台
- [watchdog-framework](#)<sup>↗</sup>: 基础权限框架
- [iartisan-admin-template](#)<sup>↗</sup>: Java 快速开发平台

- [ifast](#)<sup>↗</sup>: ifast 快速开发平台
- [roses](#)<sup>↗</sup>: 基于 Spring Cloud 的分布式框架
- [renren-security](#)<sup>↗</sup>: 人人权限系统
- [freeter-admin](#)<sup>↗</sup>: 飞特后台管理系统
- [vblog](#)<sup>↗</sup>: VBlog 博客系统
- [jiiiiiin-security](#)<sup>↗</sup>: jiiiiiin权限系统
- [hdw-dubbo](#)<sup>↗</sup>: HDW快速开发平台
- [pybbs](#)<sup>↗</sup>: 更好用的Java语言社区(论坛)
- [SmallBun](#)<sup>↗</sup>: SmallBun企业级开发脚手架
- [webplus](#)<sup>↗</sup>: 综合开发平台
- [x-boot](#)<sup>↗</sup>: VUE 前后端分离开发平台
- [nice-blog-sys](#)<sup>↗</sup>: 基于SpirngBoot开发, 好看的个人博客
- [Diboot](#)<sup>↗</sup>: 轻代码开发平台
- [tyboot](#)<sup>↗</sup>: 基于SpringBoot的快速开发脚手架
- [ac-blog](#)<sup>↗</sup> ac博客网站
- [spider-flow](#)<sup>↗</sup> 新一代爬虫平台, 以图形化方式定义爬虫流程, 不写代码即可完成爬虫

## 接入企业

---

名称按照登记先后, 希望出现您公司名称的小伙伴可以告诉我们!

- 正保远程教育集团
- 苏州罗想软件股份有限公司
- 上海箱讯网络科技有限公司
- 青岛帕特智能科技有限公司
- 成都泰尔数据服务有限公司
- 北京环球万合信息技术有限公司
- 北京万学教育科技有限公司
- 重庆声光电智联电子科技有限公司
- 锦途停车服务(天津)有限公司
- 浙江左中右电动汽车服务有限公司

- 迪斯马森科技有限公司
- 成都好玩123科技有限公司
- 深圳华云声信息技术有限公司
- 昆明万德科技有限公司
- 浙江华坤道威
- 南京昆虫软件有限公司
- 上海营联信息技术有限公司
- 上海绚奕网络技术有限公司
- 四川淘金你我信息技术有限公司
- 合肥迈思泰合信息科技有限公司
- 深圳前海蚂蚁芯城科技有限公司
- 广州金鹏集团有限公司
- 安徽自由纪信息科技有限公司
- 杭州目光科技有限公司
- 迈普拉斯科技有限公司
- 贵州红小牛数据有限公司
- 天津市神州商龙科技股份有限公司
- 安徽银通物联有限公司
- 南宁九一在线信息科技有限公司
- 青海智软网络科技有限公司
- 安徽银基信息安全技术有限责任公司
- 上海融宇信息技术有限公司
- 北京奥维云网科技股份有限公司
- 深圳市雁联移动科技有限公司
- 广东睿医大数据有限公司
- 武汉追忆那年网络科技有限公司
- 成都艺尔特科技有限公司
- 深圳市易帮云科技有限公司
- 上海中科软科技股份有限公司
- 北京熊小猫英语科技有限公司
- 武汉桑梓信息科技有限公司
- 腾讯科技（深圳）有限公司
- 苏州环境云信息科技有限公司
- 杭州阿启视科技有限公司
- 杭州杰竞科技有限公司
- 北京云图征信有限公司
- 上海科匠信息科技有限公司
- 深圳小鲨智能科技有限公司
- 深圳市优加互联科技有限公司

- 北京天赋通教育科技有限公司
- 上海(壹美分)胤新信息科技有限公司
- 厦门栗子科技有限公司
- 山东畅想云教育科技有限公司
- 成都云堆移动信息技术有限公司
- 杭州一修鸽科技有限公司
- 北京乾元大通教育科技有限公司
- 苏州帝博信息技术有限公司
- 深圳来电科技有限公司
- 上海银基信息安全有限公司
- 济南果壳科技信息有限公司
- 云鹊医疗科技(上海)有限公司
- 昆明有数科技有限公司
- 大手云(上海)金融信息服务有限公司
- 深圳未来云集

# 快速开始

我们将通过一个简单的 Demo 来阐述 MyBatis-Plus 的强大功能，在此之前，我们假设您已经：

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

现有一张 `User` 表，其表结构如下：

id	name	age	email
1	Jone	18	test1@baomidou.com
2	Jack	20	test2@baomidou.com
3	Tom	28	test3@baomidou.com
4	Sandy	21	test4@baomidou.com
5	Billie	24	test5@baomidou.com

其对应的数据库 Schema 脚本如下：

```
1. DROP TABLE IF EXISTS user;
2.
3. CREATE TABLE user
4. (
5.     id BIGINT(20) NOT NULL COMMENT '主键ID',
6.     name VARCHAR(30) NULL DEFAULT NULL COMMENT '姓名',
7.     age INT(11) NULL DEFAULT NULL COMMENT '年龄',
8.     email VARCHAR(50) NULL DEFAULT NULL COMMENT '邮箱',
9.     PRIMARY KEY (id)
10. );
```

其对应的数据库 Data 脚本如下：

```
1. DELETE FROM user;
2.
3. INSERT INTO user (id, name, age, email) VALUES
4. (1, 'Jone', 18, 'test1@baomidou.com'),
5. (2, 'Jack', 20, 'test2@baomidou.com'),
6. (3, 'Tom', 28, 'test3@baomidou.com'),
7. (4, 'Sandy', 21, 'test4@baomidou.com'),
8. (5, 'Billie', 24, 'test5@baomidou.com');
```

## Question

如果从零开始用 MyBatis-Plus 来实现该表的增删改查我们需要做什么呢？

## 初始化工程

创建一个空的 Spring Boot 工程（工程将以 H2 作为默认数据库进行演示）

可以使用 [Spring Initializer](#)  快速初始化一个 Spring Boot 工程

## 添加依赖

引入 Spring Boot Starter 父工程：

```
1. <parent>
2.     <groupId>org.springframework.boot</groupId>
3.     <artifactId>spring-boot-starter-parent</artifactId>
4.     <version>spring-latest-version</version>
5.     <relativePath/>
6. </parent>
```

引入 `spring-boot-starter`、`spring-boot-starter-test`、`mybatis-plus-boot-starter`、`lombok`、`h2` 依赖：

```
1. <dependencies>
2.     <dependency>
3.         <groupId>org.springframework.boot</groupId>
4.         <artifactId>spring-boot-starter</artifactId>
5.     </dependency>
6.     <dependency>
7.         <groupId>org.springframework.boot</groupId>
8.         <artifactId>spring-boot-starter-test</artifactId>
9.         <scope>test</scope>
10.    </dependency>
11.    <dependency>
12.        <groupId>org.projectlombok</groupId>
13.        <artifactId>lombok</artifactId>
14.        <optional>true</optional>
15.    </dependency>
```

```

16.     <dependency>
17.         <groupId>com.baomidou</groupId>
18.         <artifactId>mybatis-plus-boot-starter</artifactId>
19.         <version>starter-latest-version</version>
20.     </dependency>
21.     <dependency>
22.         <groupId>com.h2database</groupId>
23.         <artifactId>h2</artifactId>
24.         <scope>runtime</scope>
25.     </dependency>
26. </dependencies>

```

## 配置

在 `application.yml` 配置文件中添加 H2 数据库的相关配置：

```

1. # DataSource Config
2. spring:
3.     datasource:
4.         driver-class-name: org.h2.Driver
5.         schema: classpath:db/schema-h2.sql
6.         data: classpath:db/data-h2.sql
7.         url: jdbc:h2:mem:test
8.         username: root
9.         password: test

```


在 Spring Boot 启动类中添加 `@MapperScan` 注解，扫描 Mapper 文件夹：

```

1. @SpringBootApplication
2. @MapperScan("com.baomidou.mybatisplus.samples.quickstart.mapper")
3. public class Application {
4.
5.     public static void main(String[] args) {
6.         SpringApplication.run(QuickStartApplication.class, args);
7.     }
8.
9. }

```

## 编码

编写实体类 `User.java` （此处使用了 [Lombok](#)  简化代码）

```

1. @Data
2. public class User {
3.     private Long id;
4.     private String name;
5.     private Integer age;
6.     private String email;
7. }

```

编写Mapper类 `UserMapper.java`

```

1. public interface UserMapper extends BaseMapper<User> {
2.
3. }

```

## 开始使用

添加测试类，进行功能测试：

```

1. @RunWith(SpringRunner.class)
2. @SpringBootTest
3. public class SampleTest {
4.
5.     @Autowired
6.     private UserMapper userMapper;
7.
8.     @Test
9.     public void testSelect() {
10.         System.out.println("----- selectAll method test -----");
11.         List<User> userList = userMapper.selectList(null);
12.         Assert.assertEquals(5, userList.size());
13.         userList.forEach(System.out::println);
14.     }
15.
16. }

```

UserMapper 中的 `selectList()` 方法的参数为 MP 内置的条件封装器 `Wrapper`，所以不填写就是无任何条件



控制台输出：

```
1. User(id=1, name=Jone, age=18, email=test1@baomidou.com)
2. User(id=2, name=Jack, age=20, email=test2@baomidou.com)
3. User(id=3, name=Tom, age=28, email=test3@baomidou.com)
4. User(id=4, name=Sandy, age=21, email=test4@baomidou.com)
5. User(id=5, name=Billie, age=24, email=test5@baomidou.com)
```

完整的代码示例请移步：[Spring Boot 快速启动示例](#)<sup>↗</sup> | [Spring MVC 快速启动示例](#)<sup>↗</sup>

## 小结

通过以上几个简单的步骤，我们就实现了 User 表的 CRUD 功能，甚至连 XML 文件都不用编写！


从以上步骤中，我们可以看到集成 **MyBatis-Plus** 非常的简单，只需要引入 starter 工程，并配置 mapper 扫描路径即可。

但 MyBatis-Plus 的强大远不止这些功能，想要详细了解 MyBatis-Plus 的强大功能？那就继续往下看吧！

# 安装

全新的 **MyBatis-Plus** 3.0 版本基于 JDK8，提供了 **lambda** 形式的调用，所以安装集成 MP3.0 要求如下：

- JDK 8+
- Maven or Gradle

JDK7 以及下的请参考 MP2.0 版本，地址：[2.0 文档](#) 

## Release

### Spring Boot

Maven:

```
1. <dependency>
2.     <groupId>com.baomidou</groupId>
3.     <artifactId>mybatis-plus-boot-starter</artifactId>
4.     <version>starter-latest-version</version>
5. </dependency>
```

Gradle:

```
compile group: 'com.baomidou', name: 'mybatis-plus-boot-starter', version:
1. 'starter-latest-version'
```

### Spring MVC

Maven:


```
1. <dependency>
2.     <groupId>com.baomidou</groupId>
3.     <artifactId>mybatis-plus</artifactId>
4.     <version>latest-version</version>
5. </dependency>
```

Gradle:

```
1. compile group: 'com.baomidou', name: 'mybatis-plus', version: 'latest-version'
```

引入 `MyBatis-Plus` 之后请不要再次引入 `MyBatis` 以及 `MyBatis-Spring`，以避免因版本差异导致的问题。

## Snapshot

快照 SNAPSHOT 版本需要添加仓库，且版本号为快照版本 [点击查看最新快照版本号](#) 。

Maven:

```
1. <repository>
2.     <id>snapshots</id>
3.     <url>https://oss.sonatype.org/content/repositories/snapshots/</url>
4. </repository>
```

Gradle:

```
1. repositories {
2.     maven { url 'https://oss.sonatype.org/content/repositories/snapshots/' }
3. }
```

## 配置

MyBatis-Plus 的配置异常的简单，我们仅需要一些简单的配置即可使用 MyBatis-Plus 的强大功能！

在讲解配置之前，请确保您已经安装了 MyBatis-Plus，如果您尚未安装，请查看 [安装](#) 一章。

- Spring Boot 工程：
  - 配置 MapperScan 注解

```
1. @SpringBootApplication
2. @MapperScan("com.baomidou.mybatisplus.samples.quickstart.mapper")
3. public class Application {
4.
5.     public static void main(String[] args) {
6.         SpringApplication.run(QuickStartApplication.class, args);
7.     }
8.
9. }
```

- Spring MVC 工程：
  - 配置 MapperScan

```
1. <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
2.     <property name="basePackage"
3.     value="com.baomidou.mybatisplus.samples.quickstart.mapper"/>
4. </bean>
```

- 调整 SqlSessionFactory 为 MyBatis-Plus 的 SqlSessionFactory

```
1. <bean id="sqlSessionFactory"
2.     class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
3.     <property name="dataSource" ref="dataSource"/>
4. </bean>
```

通常来说，一般的简单工程，通过以上配置即可正常使用 MyBatis-Plus，具体可参考以下项目：

[Spring Boot 快速启动示例](#)<sup>🔗</sup>、[Spring MVC 快速启动示例](#)<sup>🔗</sup>。

同时 MyBatis-Plus 提供了大量的个性化配置来满足不同复杂度的工程，大家可根据自己的项目按需

配置

取用，详细配置请参考[配置一文](#)

# 注解

介绍

MybatisPlus

注解包相关类详解(更多详细描述可点击查看源码注释)

注解类包：

mybatis-plus-annotation

## @TableName

- 描述：表名注解

属性	类型	必须指定	默认值	描述
value	String	否	""	表名
schema	String	否	""	schema(@since 3.1.1)
keepGlobalPrefix	boolean	否	false	是否保持使用全局的 tablePrefix 的值(如果设置了全局 tablePrefix 且自行设置了 value 的值)(@since 3.1.1)
resultMap	String	否	""	xml 中 resultMap 的 id
autoResultMap	boolean	否	false	是否自动构建 resultMap 并使用(如果设置 resultMap 则不会进行 resultMap 的自动构建并注入)(@since 3.1.2)

关于 autoResultMap 的说明：

从mp的原理上讲,因为底层是mybatis,所以一些mybatis的常识你要知道,mp只是帮你注入了常用crud注入之前可以说是动态的(根据你entity的字段以及注解变化而变化),但是注入之后是静态的(等于你写在xml的东西)而对于直接指定 typeHandler ,mybatis只支持你写在2个地方：

- 定义在resultMap里,只作用于select查询的返回结果封装
- 定义在 insert 和 update sql的 #{property} 里的 property 后面(例：#{property,typehandler=xxx.xxx.xxx} ),只作用于 设置值 而除了这两种直接指定 typeHandler ,mybatis有一个全局的扫描你自己的 typeHandler 包的配置,这是根据你的 property 的类型去找 typeHandler 并使用这个属性的作用就是:如果你的 property 类型... todo

## @TableId

- 描述：主键注解

--	--	--	--	--

属性	类型	必须指定	默认值	描述
value	String	否	""	主键字段名
type	Enum	否	IdType.NONE	主键类型

## IdType

值	描述
AUTO	数据库自增
INPUT	自行输入
ID_WORKER	分布式全局唯一ID 长整型类型
UUID	32位UUID字符串
NONE	无状态
ID_WORKER_STR	分布式全局唯一ID 字符串类型

## @TableField

- 描述：字段注解(非主键)

属性	类型	必须指定	默认值	描述
value	String	否	""	字段名
el	String	否	""	映射为原生 <code>#{ ... }</code> 逻辑, 当于写在 xml 里的 <code>#{ ... }</code> 部分
exist	boolean	否	true	是否为数据库表字段
condition	String	否	""	字段 <code>where</code> 实体查询比较件, 有值设置则按设置的值为; 没有则为默认全局的 <code>%s=#{ %s }</code> , <a href="#">参考</a>
update	String	否	""	字段 <code>update set</code> 部分注, 例如: <code>update="%s+1"</code> : 表更新时会set <code>version=version+1</code> (该属性优先级高于 <code>el</code> 属性)
strategy	Enum	否	FieldStrategy.DEFAULT	字段验证策略 3.1.2+使用下3个替代
insertStrategy	Enum	N	DEFAULT	举例: NOT NULL: <code>insert into table a(&lt;if test="columnProperty != null"&gt;column&lt;/if&gt;) values (&lt;if test="columnProperty null"&gt;#{columnProperty}&lt;/if&gt;)</code> (since v_3.1.2
updateStrategy	Enum	N	DEFAULT	举例: IGNORED: <code>update table_a set column=#{columnProperty}</code> (since

				v_3.1.2)
whereStrategy	Enum	N	DEFAULT	举例：NOT EMPTY： <code>where&lt;if test="columnProperty null and columnProperty!=''"&gt;column{columnProperty}&lt;/if&gt;</code> (since v_3.1.2)
fill	Enum	否	FieldFill.DEFAULT	字段自动填充策略
select	boolean	否	true	是否进行 select 查询
keepGlobalFormat	boolean	否	false	是否保持使用全局的 format 进行处理(@since 3.1.1)

FieldStrategy

值	描述
IGNORED	忽略判断
NOT_NULL	非NULL判断
NOT_EMPTY	非空判断 (只对字符串类型字段, 其他类型字段依然为非NULL判断)
DEFAULT	追随全局配置

FieldFill

值	描述
DEFAULT	默认不处理
INSERT	插入时填充字段
UPDATE	更新时填充字段
INSERT_UPDATE	插入和更新时填充字段

@Version

- 描述：乐观锁注解、标记 `@Version` 在字段上

@EnumValue

- 描述：通枚举类注解 (注解在枚举字段上)

@TableLogic

- 描述：表字段逻辑处理注解 (逻辑删除)

属性	类型	必须指定	默认值	描述
value	String	否	""	逻辑未删除值
delval	String	否	""	逻辑删除值



## @SqlParser

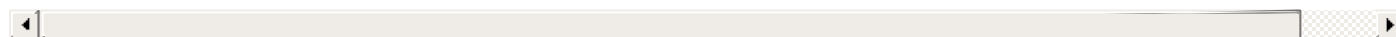
- 描述：租户注解 目前只支持注解在 mapper 的方法上(3.1.1开始支持注解在mapper上)

属性	类型	必须指定	默认值	描述
filter	boolean	否	false	true: 表示过滤SQL解析, 即不会进入ISqlParser解析链, 否则会进解析链并追加例如tenant_id等条件

## @KeySequence

- 描述：序列主键策略 `oracle`
- 属性：value、resultMap

属性	类型	必须指定	默认值	描述
value	String	否	""	序列名
clazz	Class	否	Long.class	id的类型, 可以指定String.class, 这样返回的Sequence值是字符串"1"



## 核心功能

- 代码生成器
- CRUD 接口
- 条件构造器
- 分页插件
- Sequence主键
- 自定义ID生成器

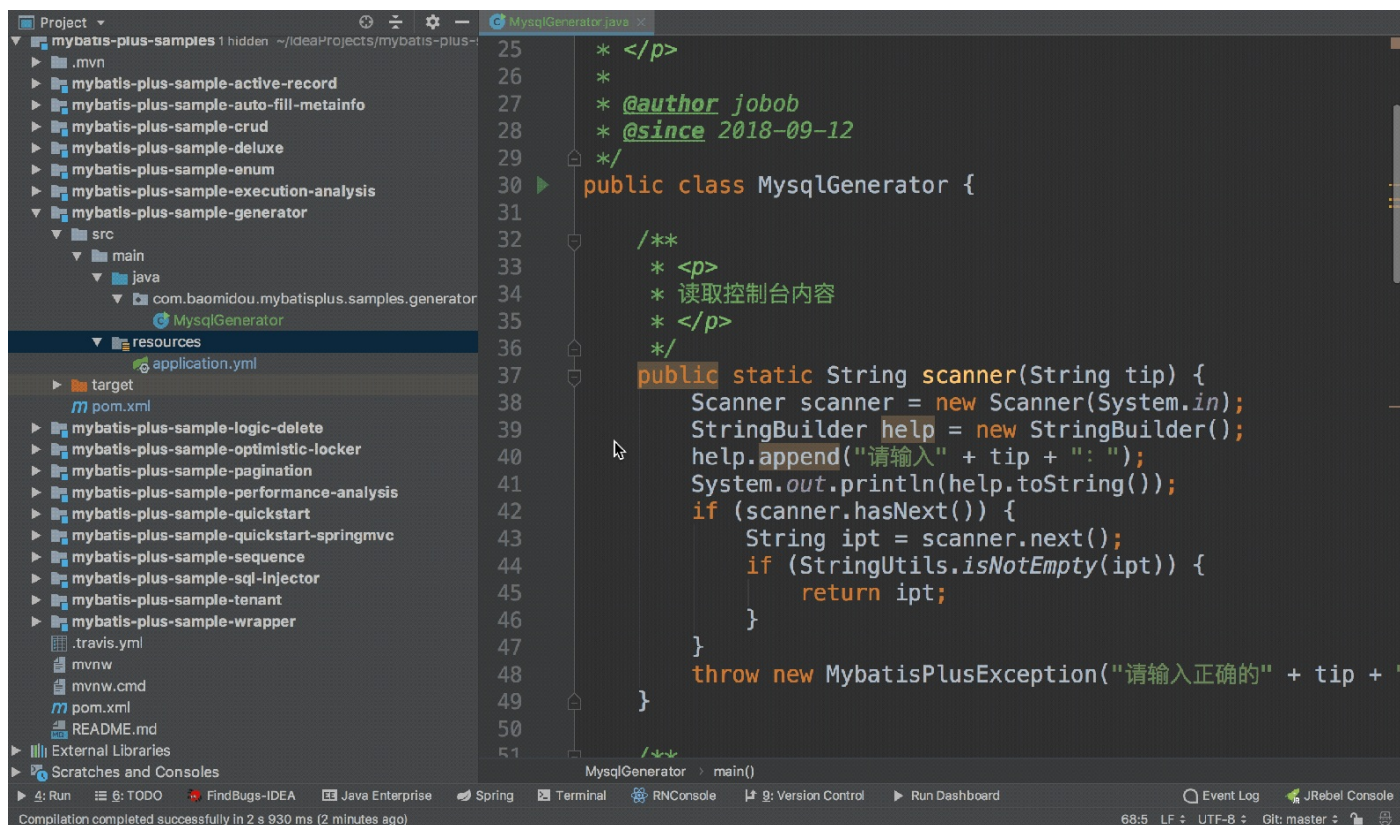
# 代码生成器

AutoGenerator 是 MyBatis-Plus 的代码生成器，通过 AutoGenerator 可以快速生成 Entity、Mapper、Mapper XML、Service、Controller 等各个模块的代码，极大的提升了开发效率。

特别说明：

自定义模板有哪些可用参数？[Github](#) [Gitee](#) AbstractTemplateEngine 类中方法 getObjectMap 返回 objectMap 的所有值都可用。

演示效果图：



1. // 演示例子，执行 main 方法控制台输入模块表名回车自动生成对应项目目录中
2. public class CodeGenerator {
- 3.
4. /\*\*
5. \* <p>
6. \* 读取控制台内容
7. \* </p>
8. \*/
9. public static String scanner(String tip) {

```
10.         Scanner scanner = new Scanner(System.in);
11.         StringBuilder help = new StringBuilder();
12.         help.append("请输入" + tip + ":");
13.         System.out.println(help.toString());
14.         if (scanner.hasNext()) {
15.             String ipt = scanner.next();
16.             if (StringUtils.isNotEmpty(ipt)) {
17.                 return ipt;
18.             }
19.         }
20.         throw new MybatisPlusException("请输入正确的" + tip + "!");
21.     }
22.
23.     public static void main(String[] args) {
24.         // 代码生成器
25.         AutoGenerator mpg = new AutoGenerator();
26.
27.         // 全局配置
28.         GlobalConfig gc = new GlobalConfig();
29.         String projectPath = System.getProperty("user.dir");
30.         gc.setOutputDir(projectPath + "/src/main/java");
31.         gc.setAuthor("jobob");
32.         gc.setOpen(false);
33.         // gc.setSwagger2(true); 实体属性 Swagger2 注解
34.         mpg.setGlobalConfig(gc);
35.
36.         // 数据源配置
37.         DataSourceConfig dsc = new DataSourceConfig();
38.         dsc.setUrl("jdbc:mysql://localhost:3306/ant?
39. useUnicode=true&useSSL=false&characterEncoding=utf8");
40.         // dsc.setSchemaName("public");
41.         dsc.setDriverName("com.mysql.jdbc.Driver");
42.         dsc.setUsername("root");
43.         dsc.setPassword("密码");
44.         mpg.setDataSource(dsc);
45.
46.         // 包配置
47.         PackageConfig pc = new PackageConfig();
48.         pc.setModuleName(scanner("模块名"));
49.         pc.setParent("com.baomidou.ant");
50.         mpg.setPackageInfo(pc);
```

```

51.         // 自定义配置
52.         InjectionConfig cfg = new InjectionConfig() {
53.             @Override
54.             public void initMap() {
55.                 // to do nothing
56.             }
57.         };
58.
59.         // 如果模板引擎是 freemarker
60.         String templatePath = "/templates/mapper.xml.ftl";
61.         // 如果模板引擎是 velocity
62.         // String templatePath = "/templates/mapper.xml.vm";
63.
64.         // 自定义输出配置
65.         List<FileOutConfig> focList = new ArrayList<>();
66.         // 自定义配置会被优先输出
67.         focList.add(new FileOutConfig(templatePath) {
68.             @Override
69.             public String outputFile(TableInfo tableInfo) {
70.                 // 自定义输出文件名 , 如果你 Entity 设置了前后缀、此处注意 xml 的名称会
71.                 // 跟着发生变化!!
72.                 return projectPath + "/src/main/resources/mapper/" +
73.                     pc.getModuleName()
74.                     + "/" + tableInfo.getEntityName() + "Mapper" +
75.                     StringPool.DOT_XML;
76.             }
77.         });
78.         /*
79.         cfg.setFileCreate(new IFileCreate() {
80.             @Override
81.             public boolean isCreate(ConfigBuilder configBuilder, FileType
82.             fileType, String filePath) {
83.                 // 判断自定义文件夹是否需要创建
84.                 checkDir("调用默认方法创建的目录");
85.                 return false;
86.             }
87.         });
88.         */
89.         cfg.setFileOutConfigList(focList);
90.         mpg.setCfg(cfg);
91.
92.         // 配置模板
93.         TemplateConfig templateConfig = new TemplateConfig();

```

```

90.
91.      // 配置自定义输出模板
92.      //指定自定义模板路径, 注意不要带上.ftl/.vm, 会根据使用的模板引擎自动识别
93.      // templateConfig.setEntity("templates/entity2.java");
94.      // templateConfig.setService();
95.      // templateConfig.setController();
96.
97.      templateConfig.setXml(null);
98.      mpg.setTemplate(templateConfig);
99.
100.     // 策略配置
101.     StrategyConfig strategy = new StrategyConfig();
102.     strategy.setNaming(NamingStrategy.underline_to_camel);
103.     strategy.setColumnNaming(NamingStrategy.underline_to_camel);
104.     strategy.setSuperEntityClass("com.baomidou.ant.common.BaseEntity");
105.     strategy.setEntityLombokModel(true);
106.     strategy.setRestControllerStyle(true);
107.     // 公共父类
108.     strategy.setSuperControllerClass("com.baomidou.ant.common.BaseController");
109.     // 写于父类中的公共字段
110.     strategy.setSuperEntityColumns("id");
111.     strategy.setInclude(scanner("表名, 多个英文逗号分割").split(","));
112.     strategy.setControllerMappingHyphenStyle(true);
113.     strategy.setTablePrefix(pc.getModuleName() + "_");
114.     mpg.setStrategy(strategy);
115.     mpg.setTemplateEngine(new FreemarkerTemplateEngine());
116.     mpg.execute();
117. }
118.
119. }
```

更多详细配置, 请参考[代码生成器配置](#)一文。

## 使用教程

### 添加依赖

MyBatis-Plus 从 **3.0.3** 之后移除了代码生成器与模板引擎的默认依赖, 需要手动添加相关依赖:

- 添加 代码生成器 依赖

```

1. <dependency>
2.     <groupId>com.baomidou</groupId>
3.     <artifactId>mybatis-plus-generator</artifactId>
4.     <version>latest-version</version>
5. </dependency>

```

- 添加 模板引擎 依赖，MyBatis-Plus 支持 Velocity (默认)、Freemarker、Beetl，用户可以选择自己熟悉的模板引擎，如果都不满足您的要求，可以采用自定义模板引擎。

Velocity (默认)：

```

1. <dependency>
2.     <groupId>org.apache.velocity</groupId>
3.     <artifactId>velocity-engine-core</artifactId>
4.     <version>latest-velocity-version</version>
5. </dependency>

```

Freemarker：

```

1. <dependency>
2.     <groupId>org.freemarker</groupId>
3.     <artifactId>freemarker</artifactId>
4.     <version>latest-freemarker-version</version>
5. </dependency>

```

Beetl：

```

1. <dependency>
2.     <groupId>com.beetl</groupId>
3.     <artifactId>beetl</artifactId>
4.     <version>latest-beetl-version</version>
5. </dependency>

```

注意！如果您选择了非默认引擎，需要在 AutoGenerator 中 设置模板引擎。

```

1. AutoGenerator generator = new AutoGenerator();
2.
3. // set freemarker engine
4. generator.setTemplateEngine(new FreemarkerTemplateEngine());
5.
6. // set beetl engine

```

```

7. generator.setTemplateEngine(new BeetlTemplateEngine());
8.
9. // set custom engine (reference class is your custom engine class)
10. generator.setTemplateEngine(new CustomTemplateEngine());
11.
12. // other config
13. ...

```

## 编写配置

MyBatis-Plus 的代码生成器提供了大量的自定义参数供用户选择，能够满足绝大部分人的使用需求。

- 配置 GlobalConfig

```

1. GlobalConfig globalConfig = new GlobalConfig();
2. globalConfig.setOutputDir(System.getProperty("user.dir") + "/src/main/java");
3. globalConfig.setAuthor("jobob");
4. globalConfig.setOpen(false);

```

- 配置 DataSourceConfig

```

1. DataSourceConfig dataSourceConfig = new DataSourceConfig();
   dataSourceConfig.setUrl("jdbc:mysql://localhost:3306/ant?
2. useUnicode=true&useSSL=false&characterEncoding=utf8");
3. dataSourceConfig.setDriverName("com.mysql.jdbc.Driver");
4. dataSourceConfig.setUsername("root");
5. dataSourceConfig.setPassword("password");

```

## 自定义模板引擎

请继承类 `com.baomidou.mybatisplus.generator.engine.AbstractTemplateEngine`

## 自定义代码模板

```

1. //指定自定义模板路径，位置：/resources/templates/entity2.java.ftl(或者是.vm)
2. //注意不要带上.ftl(或者是.vm)，会根据使用的模板引擎自动识别
3. TemplateConfig templateConfig = new TemplateConfig()
4.     .setEntity("templates/entity2.java");
5.

```



```
6. AutoGenerator mpg = new AutoGenerator();
7. //配置自定义模板
8. mpg.setTemplate(templateConfig);
```

## 自定义属性注入

```
1. InjectionConfig injectionConfig = new InjectionConfig() {
2.     //自定义属性注入:abc
3.     //在.ftl(或者是.vm)模板中, 通过${cfg.abc}获取属性
4.     @Override
5.     public void initMap() {
6.         Map<String, Object> map = new HashMap<>();
7.         map.put("abc", this.getConfig().getGlobalConfig().getAuthor() + "-mp");
8.         this.setMap(map);
9.     }
10. };
11. AutoGenerator mpg = new AutoGenerator();
12. //配置自定义属性注入
13. mpg.setCfg(injectionConfig);
```

```
1. entity2.java.ftl
2. 自定义属性注入abc=${cfg.abc}
3.
4. entity2.java.vm
5. 自定义属性注入abc=${!cfg.abc}
```

## 字段其他信息查询注入

field

```

field = {TableField@1953} "TableField(convert=false, keyFlag=true, keyIdentityFlag=true, name=id, type=int(11), propertyType=... View
  convert = false
  keyFlag = true
  keyIdentityFlag = true
  name = "id"
  type = "int(11)"
  propertyName = "id"
  columnType = {DbColumnType@2135} "INTEGER"
  comment = ""
  fill = null
  customMap = {HashMap@1956} size = 2
    "NULL" -> "NO"
    "PRIVILEGES" -> "select,insert,update,references"

```

对象 **TableInfo** 字段列表 **fields**  
属性 **TableField** 自定义查询字段 **MAP** 结果

```

1 show full fields from jobs_info

```

演示为 **MySQLQuery** 数据信息类方法 **tablesSql** 输出 SQL 查询结果，自定义 Null 及 Privileges 字段

Field	Type	Collation	Null	Key	Default	Extra	Privileges	Comment
id	int(11)	(NULL)	NO	PRI	(NULL)	auto_increment	select,insert,update,references	
job_group	int(11)	(NULL)	NO		(NULL)		select,insert,update,references	执行器主键ID
job_cron	varchar(128)	utf8_general_ci	NO		(NULL)		select,insert,update,references	任务执行CRON

```

1. new DataSourceConfig().setDbQuery(new MySQLQuery() {
2.
3.     /**
4.      * 重写父类预留查询自定义字段<br>
5.      * 这里查询的 SQL 对应父类 tableFieldsSql 的查询字段，默认不能满足你的需求请重写它
6.      * 模板中调用： table.fields 获取所有字段信息，
7.      * 然后循环字段获取 field.customMap 从 MAP 中获取注入字段如下 NULL 或者
8.      PRIVILEGES
9.      */
10.     @Override
11.     public String[] fieldCustom() {
12.         return new String[]{"NULL", "PRIVILEGES"};
13.     }
14. })

```

## CRUD 接口

### Mapper CRUD 接口

说明：

- 通用 CRUD 封装 `BaseMapper` <sup>↗</sup> 接口，为 `Mybatis-Plus` 启动时自动解析实体表关系映射转换为 `Mybatis` 内部对象注入容器
- 泛型 `T` 为任意实体对象
- 参数 `Serializable` 为任意类型主键 `Mybatis-Plus` 不推荐使用复合主键约定每一张表都有自己的唯一 `id` 主键
- 对象 `wrapper` 为 条件构造器

#### insert

```

1.  /**
2.   * <p>
3.   * 插入一条记录
4.   * </p>
5.   *
6.   * @param entity 实体对象
7.   * @return 插入成功记录数
8.   */
9.  int insert(T entity);

```

#### deleteById

```

1.  /**
2.   * <p>
3.   * 根据 ID 删除
4.   * </p>
5.   *
6.   * @param id 主键ID
7.   * @return 删除成功记录数
8.   */
9.  int deleteById(Serializable id);

```

#### deleteByMap

```

1.  /**
2.   * <p>
3.   * 根据 columnMap 条件，删除记录
4.   * </p>
5.   *
6.   * @param columnMap 表字段 map 对象
7.   * @return 删除成功记录数
8.   */
9.  int deleteByMap(@Param(Constants.COLUMN_MAP) Map<String, Object> columnMap);

```

## delete

```

1.  /**
2.   * <p>
3.   * 根据 entity 条件，删除记录
4.   * </p>
5.   *
6.   * @param wrapper 实体对象封装操作类（可以为 null）
7.   * @return 删除成功记录数
8.   */
9.  int delete(@Param(Constants.WRAPPER) Wrapper<T> wrapper);

```

## deleteBatchIds

```

1.  /**
2.   * <p>
3.   * 删除（根据ID 批量删除）
4.   * </p>
5.   *
6.   * @param idList 主键ID列表(不能为 null 以及 empty)
7.   * @return 删除成功记录数
8.   */
9.  int deleteBatchIds(@Param(Constants.COLLECTION) Collection<? extends
    Serializable> idList);

```

## updateById

```

1.  /**
2.   * <p>
3.   * 根据 ID 修改

```

```

4.  * </p>
5.  *
6.  * @param entity 实体对象
7.  * @return 修改成功记录数
8.  */
9.  int updateById(@Param(Constants.ENTITY) T entity);

```

## update

```

1.  /**
2.  * <p>
3.  * 根据 whereEntity 条件, 更新记录
4.  * </p>
5.  *
6.  * @param entity          实体对象 (set 条件值, 可为 null)
7.  * @param updateWrapper  实体对象封装操作类 (可以为 null, 里面的 entity 用于生成 where
8.  * 语句)
9.  * @return 修改成功记录数
10. */
    int update(@Param(Constants.ENTITY) T entity, @Param(Constants.WRAPPER)
    Wrapper<T> updateWrapper);

```

## selectById

```

1.  /**
2.  * <p>
3.  * 根据 ID 查询
4.  * </p>
5.  *
6.  * @param id 主键ID
7.  * @return 实体
8.  */
9.  T selectById(Serializable id);

```

## selectBatchIds

```

1.  /**
2.  * <p>
3.  * 查询 (根据ID 批量查询)
4.  * </p>

```

```

5.  *
6.  * @param idList 主键ID列表(不能为 null 以及 empty)
7.  * @return 实体集合
8.  */
   List<T> selectBatchIds(@Param(Constants.COLLECTION) Collection<? extends
9.  Serializable> idList);

```

## selectByMap

```

1.  /**
2.  * <p>
3.  * 查询 (根据 columnMap 条件)
4.  * </p>
5.  *
6.  * @param columnMap 表字段 map 对象
7.  * @return 实体集合
8.  */
   List<T> selectByMap(@Param(Constants.COLUMN_MAP) Map<String, Object>
9.  columnMap);
10.

```

## selectOne

- 如果逻辑非唯一该方法不会自动替您 `limit 1` 你需要 `wrapper.last("limit 1")` 设置唯一性。

```

1.  /**
2.  * <p>
3.  * 根据 entity 条件, 查询一条记录
4.  * </p>
5.  *
6.  * @param queryWrapper 实体对象
7.  * @return 实体
8.  */
9.  T selectOne(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

```

## selectCount

```

1.  /**
2.  * <p>
3.  * 根据 wrapper 条件, 查询总记录数

```

```

4.  * </p>
5.  *
6.  * @param queryWrapper 实体对象
7.  * @return 满足条件记录数
8.  */
9. Integer selectCount(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

```

## selectList

```

1. /**
2.  * <p>
3.  * 根据 entity 条件，查询全部记录
4.  * </p>
5.  *
6.  * @param queryWrapper 实体对象封装操作类（可以为 null）
7.  * @return 实体集合
8.  */
9. List<T> selectList(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

```

## selectMaps

```

1. /**
2.  * <p>
3.  * 根据 wrapper 条件，查询全部记录
4.  * </p>
5.  *
6.  * @param queryWrapper 实体对象封装操作类（可以为 null）
7.  * @return 字段映射对象 Map 集合
8.  */
9. List<Map<String, Object>> selectMaps(@Param(Constants.WRAPPER) Wrapper<T>
    queryWrapper);

```

## selectObjs

```

1. /**
2.  * <p>
3.  * 根据 wrapper 条件，查询全部记录
4.  * 注意：只返回第一个字段的值
5.  * </p>
6.  *

```

```

7.  * @param queryWrapper 实体对象封装操作类（可以为 null）
8.  * @return 字段映射对象集合
9.  */
10. List<Object> selectObjs(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

```

## selectPage

```

1.  /**
2.  * <p>
3.  * 根据 entity 条件，查询全部记录（并翻页）
4.  * </p>
5.  *
6.  * @param page          分页查询条件（可以为 RowBounds.DEFAULT）
7.  * @param queryWrapper 实体对象封装操作类（可以为 null）
8.  * @return 实体分页对象
9.  */
    IPage<T> selectPage(IPage<T> page, @Param(Constants.WRAPPER) Wrapper<T>
10. queryWrapper);

```

## selectMapsPage

```

1.  /**
2.  * <p>
3.  * 根据 wrapper 条件，查询全部记录（并翻页）
4.  * </p>
5.  *
6.  * @param page          分页查询条件
7.  * @param queryWrapper 实体对象封装操作类
8.  * @return 字段映射对象 Map 分页对象
9.  */
    IPage<Map<String, Object>> selectMapsPage(IPage<T> page,
10. @Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

```

## Service CRUD 接口

说明：

- 通用 Service CRUD 封装 `IService` <sup>🔗</sup> 接口，进一步封装 CRUD 采用 `get` 查询单行 `remove` 删除 `list` 查询集合 `page` 分页 前缀命名方式区分 `Mapper` 层避免混淆，
- 泛型 `T` 为任意实体对象



- 建议如果存在自定义通用 Service 方法的可能，请创建自己的 `IBaseService` 继承 `Mybatis-Plus` 提供的基类
- 对象 `Wrapper` 为 条件构造器

## save

```

1.  /**
2.   * <p>
3.   * 插入一条记录（选择字段，策略插入）
4.   * </p>
5.   *
6.   * @param entity 实体对象
7.   */
8.  boolean save(T entity);

```

## saveBatch

```

1.  /**
2.   * 插入（批量）
3.   *
4.   * @param entityList 实体对象集合
5.   * @param batchSize 插入批次数量
6.   */
7.  boolean saveBatch(Collection<T> entityList);

```

## saveBatch

```

1.  /**
2.   * 插入（批量）
3.   *
4.   * @param entityList 实体对象集合
5.   * @param batchSize 插入批次数量
6.   */
7.  boolean saveBatch(Collection<T> entityList, int batchSize);

```

## saveOrUpdateBatch

```

1.  /**
2.   * <p>
3.   * 批量修改插入

```

```

4.  * </p>
5.  *
6.  * @param entityList 实体对象集合
7.  */
8.  boolean saveOrUpdateBatch(Collection<T> entityList);

```

## saveOrUpdateBatch

```

1.  /**
2.  * <p>
3.  * 批量修改插入
4.  * </p>
5.  *
6.  * @param entityList 实体对象集合
7.  * @param batchSize 每次的数量
8.  */
9.  boolean saveOrUpdateBatch(Collection<T> entityList, int batchSize);

```

## removeById

```

1.  /**
2.  * <p>
3.  * 根据 ID 删除
4.  * </p>
5.  *
6.  * @param id 主键ID
7.  */
8.  boolean removeById(Serializable id);

```

## removeByMap

```

1.  /**
2.  * <p>
3.  * 根据 columnMap 条件, 删除记录
4.  * </p>
5.  *
6.  * @param columnMap 表字段 map 对象
7.  */
8.  boolean removeByMap(Map<String, Object> columnMap);

```

## remove

```

1.  /**
2.   * <p>
3.   * 根据 entity 条件, 删除记录
4.   * </p>
5.   *
6.   * @param queryWrapper 实体包装类 {@link
7.   *   com.baomidou.mybatisplus.core.conditions.query.QueryWrapper}
8.   */
9.  boolean remove(Wrapper<T> queryWrapper);

```

## removeByIds

```

1.  /**
2.   * <p>
3.   * 删除 (根据ID 批量删除)
4.   * </p>
5.   *
6.   * @param idList 主键ID列表
7.   */
8.  boolean removeByIds(Collection<? extends Serializable> idList);

```

## updateById

```

1.  /**
2.   * <p>
3.   * 根据 ID 选择修改
4.   * </p>
5.   *
6.   * @param entity 实体对象
7.   */
8.  boolean updateById(T entity);

```

## update

```

1.  /**
2.   * <p>
3.   * 根据 whereEntity 条件, 更新记录
4.   * </p>

```

```

5.  *
6.  * @param entity      实体对象
   * @param updateWrapper 实体对象封装操作类 {@link
7.  com.baomidou.mybatisplus.core.conditions.update.UpdateWrapper}
8.  */
9.  boolean update(T entity, Wrapper<T> updateWrapper);

```

## updateBatchById

```

1.  /**
2.  * <p>
3.  * 根据ID 批量更新
4.  * </p>
5.  *
6.  * @param entityList 实体对象集合
7.  * @param batchSize 更新批次数量
8.  */
9.  boolean updateBatchById(Collection<T> entityList, int batchSize);

```

## saveOrUpdate

```

1.  /**
2.  * <p>
3.  * TableId 注解存在更新记录，否插入一条记录
4.  * </p>
5.  *
6.  * @param entity 实体对象
7.  */
8.  boolean saveOrUpdate(T entity);

```

## getById

```

1.  /**
2.  * <p>
3.  * 根据 ID 查询
4.  * </p>
5.  *
6.  * @param id 主键ID
7.  */
8.  T getById(Serializable id);

```

## listByIds

```

1.  /**
2.   * <p>
3.   * 查询（根据ID 批量查询）
4.   * </p>
5.   *
6.   * @param idList 主键ID列表
7.   */
8.  Collection<T> listByIds(Collection<? extends Serializable> idList);

```

## listByMap

```

1.  /**
2.   * <p>
3.   * 查询（根据 columnMap 条件）
4.   * </p>
5.   *
6.   * @param columnMap 表字段 map 对象
7.   */
8.  Collection<T> listByMap(Map<String, Object> columnMap);

```

## getOne

```

1.  /**
2.   * <p>
3.   * 根据 wrapper, 查询一条记录
4.   * </p>
5.   *
6.   * @param queryWrapper 实体对象封装操作类 {@link
7.   *   com.baomidou.mybatisplus.core.conditions.query.QueryWrapper}
8.   * @param throwEx      有多个 result 是否抛出异常
9.   */
9.  T getOne(Wrapper<T> queryWrapper, boolean throwEx);

```

## getMap

```

1.  /**
2.   * <p>
3.   * 根据 wrapper, 查询一条记录

```

```

4.  * </p>
5.  *
   * @param queryWrapper 实体对象封装操作类 {@link
6.  com.baomidou.mybatisplus.core.conditions.query.QueryWrapper}
7.  */
8.  Map<String, Object> getMap(Wrapper<T> queryWrapper);

```

## getObj

```

1.  /**
2.  * <p>
3.  * 根据 Wrapper, 查询一条记录
4.  * </p>
5.  *
   * @param queryWrapper 实体对象封装操作类 {@link
6.  com.baomidou.mybatisplus.core.conditions.query.QueryWrapper}
7.  */
8.  Object getObj(Wrapper<T> queryWrapper);

```

## count

```

1.  /**
2.  * <p>
3.  * 根据 Wrapper 条件, 查询总记录数
4.  * </p>
5.  *
   * @param queryWrapper 实体对象封装操作类 {@link
6.  com.baomidou.mybatisplus.core.conditions.query.QueryWrapper}
7.  */
8.  int count(Wrapper<T> queryWrapper);

```

## list

```

1.  /**
2.  * <p>
3.  * 查询列表
4.  * </p>
5.  *
   * @param queryWrapper 实体对象封装操作类 {@link
6.  com.baomidou.mybatisplus.core.conditions.query.QueryWrapper}
7.  */

```

```
8. List<T> list(Wrapper<T> queryWrapper);
```

## page

```
1. /**
2.  * <p>
3.  * 翻页查询
4.  * </p>
5.  *
6.  * @param page          翻页对象
7.  * @param queryWrapper 实体对象封装操作类 {@link
8.  *                        com.baomidou.mybatisplus.core.conditions.query.QueryWrapper}
9.  */
9. IPage<T> page(IPage<T> page, Wrapper<T> queryWrapper);
```

## listMaps

```
1. /**
2.  * <p>
3.  * 查询列表
4.  * </p>
5.  *
6.  * @param queryWrapper 实体对象封装操作类 {@link
7.  *                        com.baomidou.mybatisplus.core.conditions.query.QueryWrapper}
8.  */
8. List<Map<String, Object>> listMaps(Wrapper<T> queryWrapper);
```

## listObjs

```
1. /**
2.  * <p>
3.  * 根据 Wrapper 条件，查询全部记录
4.  * </p>
5.  *
6.  * @param queryWrapper 实体对象封装操作类 {@link
7.  *                        com.baomidou.mybatisplus.core.conditions.query.QueryWrapper}
8.  */
8. List<Object> listObjs(Wrapper<T> queryWrapper);
```

## pageMaps

```

1.  /**
2.   * <p>
3.   * 翻页查询
4.   * </p>
5.   *
6.   * @param page          翻页对象
7.   * @param queryWrapper 实体对象封装操作类 {@link
8.   * com.baomidou.mybatisplus.core.conditions.query.QueryWrapper}
9.   */
9.  IPage<Map<String, Object>> pageMaps(IPage<T> page, Wrapper<T> queryWrapper);

```

## mapper 层 选装件

说明：

选装件位于 `com.baomidou.mybatisplus.extension.injector.methods.additional` 包下需要

配合Sql 注入器使用, [案例](#)<sup>↗</sup>使用详细见[源码注释](#)<sup>↗</sup>

### AlwaysUpdateSomeColumnById

```

1.  int alwaysUpdateSomeColumnById(T entity);

```

### insertBatchSomeColumn

```

1.  int insertBatchSomeColumn(List<T> entityList);

```

### deleteByIdWithFill

```


1.  int deleteByIdWithFill(T entity);

```



# 条件构造器

说明:

- 以下出现的第一个入参 `boolean condition` 表示该条件是否加入最后生成的sql中
- 以下代码块内的多个方法均为从上往下补全个别 `boolean` 类型的入参,默认为 `true`
- 以下出现的泛型 `Param` 均为 `Wrapper` 的子类实例(均具有 `AbstractWrapper` 的所有方法)
- 以下方法在入参中出现的 `R` 为泛型,在普通wrapper中是 `String`,在LambdaWrapper中是函数(例: `Entity::getId`, `Entity` 为实体类, `getId` 为字段 `id` 的`getMethod`)
- 以下方法入参中的 `R column` 均表示数据库字段,当 `R` 具体类型为 `String` 时则为数据库字段名(字段名是数据库关键字的自己用转义符包裹!)!而不是实体类数据字段名!!!,另当 `R` 具体类型为 `SFunction` 时项目runtime不支持eclipse自家的编译器!!!
- 以下举例均为使用普通wrapper,入参为 `Map` 和 `List` 的均以 `json` 形式表现!
- 使用中如果入参的 `Map` 或者 `List` 为空,则不会加入最后生成的sql中!!!
- 有任何疑问就点开源码看,看不懂函数的[点击我学习新知识](#) 

警告:

不支持以及不赞成在 RPC 调用中把 wrapper 进行传输

- wrapper 很重
- 传输 wrapper 可以类比为你的 controller 用 map 接收值(开发一时爽,维护火葬场)
- 正确的 RPC 调用姿势是写一个 DTO 进行传输,被调用方再根据 DTO 执行相应的操作
- 我们拒绝接受任何关于 RPC 传输 wrapper 报错相关的 issue 甚至 pr

## AbstractWrapper

说明:

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 条件注意: entity 生成的 where 条件与 使用各个 api 生成的 where 条件没有任何关联行为

### allEq

1. `allEq(Map<R, V> params)`
2. `allEq(Map<R, V> params, boolean null2IsNull)`
3. `allEq(boolean condition, Map<R, V> params, boolean null2IsNull)`

- 全部eq(或个别isNull)

个别参数说明：

`params` : `key` 为数据库字段名, `value` 为字段值 `null2IsNull` : 为 `true` 则在 `map` 的 `value` 为 `null` 时调用 `isNull` 方法, 为 `false` 时则忽略 `value` 为 `null` 的

- 例1: `allEq({id:1,name:"老王",age:null}) --> id = 1 and name = '老王' and age is null`
- 例2: `allEq({id:1,name:"老王",age:null}, false) --> id = 1 and name = '老王'`

```
1. allEq(BiPredicate<R, V> filter, Map<R, V> params)
2. allEq(BiPredicate<R, V> filter, Map<R, V> params, boolean null2IsNull)
   allEq(boolean condition, BiPredicate<R, V> filter, Map<R, V> params, boolean
3. null2IsNull)
```

个别参数说明：

`filter` : 过滤函数, 是否允许字段传入比对条件中 `params` 与 `null2IsNull` : 同上

- 例1: `allEq((k,v) -> k.indexOf("a") >= 0, {id:1,name:"老王",age:null}) --> name = '老王' and age is null`
- 例2: `allEq((k,v) -> k.indexOf("a") >= 0, {id:1,name:"老王",age:null}, false) --> name = '老王'`

## eq

```
1. eq(R column, Object val)
2. eq(boolean condition, R column, Object val)
```

- 等于 =
- 例: `eq("name", "老王") --> name = '老王'`

## ne

```
1. ne(R column, Object val)
2. ne(boolean condition, R column, Object val)
```

- 不等于 <>
- 例: `ne("name", "老王") --> name <> '老王'`

## gt

1. `gt(R column, Object val)`
2. `gt(boolean condition, R column, Object val)`

- 大于 >
- 例: `gt("age", 18) --> age > 18`

## ge

1. `ge(R column, Object val)`
2. `ge(boolean condition, R column, Object val)`

- 大于等于 >=
- 例: `ge("age", 18) --> age >= 18`

## lt

1. `lt(R column, Object val)`
2. `lt(boolean condition, R column, Object val)`

- 小于 <
- 例: `lt("age", 18) --> age < 18`

## le

1. `le(R column, Object val)`
2. `le(boolean condition, R column, Object val)`

- 小于等于 <=
- 例: `le("age", 18) --> age <= 18`

## between

1. `between(R column, Object val1, Object val2)`
2. `between(boolean condition, R column, Object val1, Object val2)`

- BETWEEN 值1 AND 值2
- 例: `between("age", 18, 30) --> age between 18 and 30`

## notBetween

1. `notBetween(R column, Object val1, Object val2)`
2. `notBetween(boolean condition, R column, Object val1, Object val2)`

- NOT BETWEEN 值1 AND 值2
- 例: `notBetween("age", 18, 30)` --> `age not between 18 and 30`

## like

1. `like(R column, Object val)`
2. `like(boolean condition, R column, Object val)`

- LIKE '%值%'
- 例: `like("name", "王")` --> `name like '王%'`

## notLike

1. `notLike(R column, Object val)`
2. `notLike(boolean condition, R column, Object val)`

- NOT LIKE '%值%'
- 例: `notLike("name", "王")` --> `name not like '王%'`

## likeLeft

1. `likeLeft(R column, Object val)`
2. `likeLeft(boolean condition, R column, Object val)`

- LIKE '值%'
- 例: `likeLeft("name", "王")` --> `name like '王'`

## likeRight

1. `likeRight(R column, Object val)`
2. `likeRight(boolean condition, R column, Object val)`

- LIKE '值%'
- 例: `likeRight("name", "王")` --> `name like '王'`

## isNull

1. `isNull(R column)`
2. `isNull(boolean condition, R column)`

- 字段 IS NULL
- 例: `isNull("name") --> name is null`

## isNotNull

1. `isNotNull(R column)`
2. `isNotNull(boolean condition, R column)`

- 字段 IS NOT NULL
- 例: `isNotNull("name") --> name is not null`

## in

1. `in(R column, Collection<?> value)`
2. `in(boolean condition, R column, Collection<?> value)`

- 字段 IN (value.get(0), value.get(1), ...)
- 例: `in("age", {1,2,3}) --> age in (1,2,3)`

1. `in(R column, Object... values)`
2. `in(boolean condition, R column, Object... values)`

- 字段 IN (v0, v1, ...)
- 例: `in("age", 1, 2, 3) --> age in (1,2,3)`

## notIn

1. `notIn(R column, Collection<?> value)`
2. `notIn(boolean condition, R column, Collection<?> value)`

- 字段 IN (value.get(0), value.get(1), ...)
- 例: `notIn("age", {1,2,3}) --> age not in (1,2,3)`

1. `notIn(R column, Object... values)`
2. `notIn(boolean condition, R column, Object... values)`

- 字段 NOT IN (v0, v1, ...)

- 例: `notIn("age", 1, 2, 3) --> age not in (1,2,3)`

## inSql

1. `inSql(R column, String inValue)`
2. `inSql(boolean condition, R column, String inValue)`

- 字段 IN ( sql语句 )
- 例: `inSql("age", "1,2,3,4,5,6") --> 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

1. `notInSql(R column, String inValue)`
2. `notInSql(boolean condition, R column, String inValue)`

- 字段 NOT IN ( sql语句 )
- 例: `notInSql("age", "1,2,3,4,5,6") --> age not in (1,2,3,4,5,6)`
- 例: `notInSql("id", "select id from table where id < 3") --> age not in (select id from table where id < 3)`

## groupBy

1. `groupBy(R... columns)`
2. `groupBy(boolean condition, R... columns)`

- 分组: GROUP BY 字段, ...
- 例: `groupBy("id", "name") --> group by id,name`

## orderByAsc

1. `orderByAsc(R... columns)`
2. `orderByAsc(boolean condition, R... columns)`

- 排序: ORDER BY 字段, ... ASC
- 例: `orderByAsc("id", "name") --> order by id ASC,name ASC`

## orderByDesc

1. `orderByDesc(R... columns)`
2. `orderByDesc(boolean condition, R... columns)`

- 排序: ORDER BY 字段, ... DESC
- 例: `orderByDesc("id", "name") --> order by id DESC,name DESC`

## orderBy

1. `orderBy(boolean condition, boolean isAsc, R... columns)`

- 排序: ORDER BY 字段, ...
- 例: `orderBy(true, true, "id", "name") --> order by id ASC,name ASC`

## having

1. `having(String sqlHaving, Object... params)`
2. `having(boolean condition, String sqlHaving, Object... params)`

- HAVING ( sql语句 )
- 例: `having("sum(age) > 10") --> having sum(age) > 10`
- 例: `having("sum(age) > {0}", 11) --> having sum(age) > 11`

## or

1. `or()`
2. `or(boolean condition)`

- 拼接 OR

注意事项:

主动调用 `or` 表示紧接着下一个方法不是用 `and` 连接!(不调用 `or` 则默认为使用 `and` 连接)

- 例: `eq("id",1).or().eq("name","老王") --> id = 1 or name = '老王'`

1. `or(Function<Param, Param> func)`
2. `or(boolean condition, Function<Param, Param> func)`

- OR 嵌套
- 例: `or(i -> i.eq("name", "李白").ne("status", "活着")) --> or (name = '李白' and status <> '活着')`

## and

1. `and(Function<Param, Param> func)`
2. `and(boolean condition, Function<Param, Param> func)`

- AND 嵌套

- 例: `and(i -> i.eq("name", "李白").ne("status", "活着")) --> and (name = '李白' and status <> '活着')`

## nested

1. `nested(Function<Param, Param> func)`
2. `nested(boolean condition, Function<Param, Param> func)`

- 正常嵌套 不带 AND 或者 OR

- 例: `nested(i -> i.eq("name", "李白").ne("status", "活着")) --> (name = '李白' and status <> '活着')`

## apply

1. `apply(String applySql, Object... params)`
2. `apply(boolean condition, String applySql, Object... params)`

- 拼接 sql

注意事项:

该方法可用于数据库函数动态入参的 `params` 对应前面 `applySql` 内部的 `{index}` 部分. 这样是不会有sql注入风险的, 反之会有!

- 例: `apply("id = 1") --> id = 1`
- 例: `apply("date_format(dateColumn, '%Y-%m-%d') = '2008-08-08'") --> date_format(dateColumn, '%Y-%m-%d') = '2008-08-08')`
- 例: `apply("date_format(dateColumn, '%Y-%m-%d') = {0}", "2008-08-08") --> date_format(dateColumn, '%Y-%m-%d') = '2008-08-08')`

## last

1. `last(String lastSql)`
2. `last(boolean condition, String lastSql)`



- 无视优化规则直接拼接到 sql 的最后

注意事项：

只能调用一次, 多次调用以最后一次为准有sql注入的风险, 请谨慎使用

- 例: `last("limit 1")`

## exists

1. `exists(String existsSql)`
2. `exists(boolean condition, String existsSql)`

- 拼接 EXISTS ( sql语句 )
- 例: `exists("select id from table where age = 1") --> exists (select id from table where age = 1)`

## notExists

1. `notExists(String notExistsSql)`
2. `notExists(boolean condition, String notExistsSql)`

- 拼接 NOT EXISTS ( sql语句 )
- 例: `notExists("select id from table where age = 1") --> not exists (select id from table where age = 1)`

## QueryWrapper

说明：

继承自 AbstractWrapper , 自身的内部属性 entity 也用于生成 where 条件及 LambdaQueryWrapper, 可以通过 `new QueryWrapper().lambda()` 方法获取

## select

1. `select(String... sqlSelect)`
2. `select(Predicate<TableFieldInfo> predicate)`
3. `select(Class<T> entityClass, Predicate<TableFieldInfo> predicate)`

- 设置查询字段

说明:

以上方分法为两类. 第二类方法为: 过滤查询字段(主键除外), 入参不包含 class 的调用前需要 wrapper 内的 entity 属性有值! 这两类方法重复调用以最后一次为准

- 例: `select("id", "name", "age")`
- 例: `select(i -> i.getProperty().startsWith("test"))`

## excludeColumns @Deprecated

- 排除查询字段

已从 3.0.5 版本上移除此方法!

## UpdateWrapper

说明:

继承自 `AbstractWrapper`, 自身的内部属性 `entity` 也用于生成 where 条件及 `LambdaUpdateWrapper`, 可以通过 `new UpdateWrapper().lambda()` 方法获取!

## set

1. `set(String column, Object val)`
2. `set(boolean condition, String column, Object val)`

- SQL SET 字段
- 例: `set("name", "老李头")`
- 例: `set("name", "")` --> 数据库字段值变为空字符串
- 例: `set("name", null)` --> 数据库字段值变为 `null`

## setSql

1. `setSql(String sql)`

- 设置 SET 部分 SQL
- 例: `setSql("name = '老李头'")`

## lambda

- 获取 `LambdaWrapper` 在 `QueryWrapper` 中是获

取 `LambdaQueryWrapper` 在 `UpdateWrapper` 中是获取 `LambdaUpdateWrapper`

## 使用 Wrapper 自定义SQL

需求来源：

在使用了 `mybatis-plus` 之后，自定义SQL的同时也想使用 `Wrapper` 的便利应该怎么办？

在 `mybatis-plus` 版本 `3.0.7` 得到了完美解决版本需要大于或等于 `3.0.7`，以下两种方案取其一即可

### Service.java

```
mysqlMapper.getAll(Wrappers.<MysqlData>lambdaQuery().eq(MysqlData::getGroup,
1. 1));
```

### 方案一 注解方式 Mapper.java

```
1. @Select("select * from mysql_data ${ew.customSqlSegment}")
2. List<MysqlData> getAll(@Param(Constants.WRAPPER) Wrapper wrapper);
```

### 方案二 XML形式 Mapper.xml

```
1. <select id="getAll" resultType="MysqlData">
2.     SELECT * FROM mysql_data ${ew.customSqlSegment}
3. </select>
4.
```

# 分页插件

示例工程：

[mybatis-plus-sample-pagination](#) 

```

1. <!-- spring xml 方式 -->
2. <plugins>
    <plugin
3. interceptor="com.baomidou.mybatisplus.extension.plugins.PaginationInterceptor">
4.     <property name="sqlParser" ref="自定义解析类、可以没有" />
5.     <property name="dialectClazz" value="自定义方言类、可以没有" />
6.   </plugin>
7. </plugins>

```

```

1. //Spring boot方式
2. @EnableTransactionManagement
3. @Configuration
4. @MapperScan("com.baomidou.cloud.service.*.mapper*")
5. public class MybatisPlusConfig {
6.
7.     @Bean
8.     public PaginationInterceptor paginationInterceptor() {
9.         PaginationInterceptor paginationInterceptor = new
10.         PaginationInterceptor();
11.         // 设置请求的页面大于最大页后操作， true调回到首页， false 继续请求 默认false
12.         // paginationInterceptor.setOverflow(false);
13.         // 设置最大单页限制数量，默认 500 条， -1 不受限制
14.         // paginationInterceptor.setLimit(500);
15.         return paginationInterceptor;
16.     }
17. }

```

## XML 自定义分页

- UserMapper.java 方法内容

```

1. public interface UserMapper{ //可以继承或者不继承BaseMapper

```

```

2.      /**
3.      * <p>
4.      * 查询 : 根据state状态查询用户列表, 分页显示
5.      * 注意!! : 如果入参是有多个, 需要加注解指定参数名才能在xml中取值
6.      * </p>
7.      *
8.      * @param page 分页对象, xml中可以从里面进行取值, 传递参数 Page 即自动分页, 必须放在第
9.      * 一位(你可以继承Page实现自己的分页对象)
10.     * @param state 状态
11.     * @return 分页对象
12.     */
13.     IPage<User> selectPageVo(Page page, @Param("state") Integer state);

```

- UserMapper.xml 等同于编写一个普通 list 查询, mybatis-plus 自动替你分页

```

1. <select id="selectPageVo" resultType="com.baomidou.cloud.entity.UserVo">
2.     SELECT id,name FROM user WHERE state=#{state}
3. </select>

```

- UserServiceImpl.java 调用分页方法

```

public IPage<User> selectUserPage(Page<User> page, Integer state) {
    // 不进行 count sql 优化, 解决 MP 无法自动优化 SQL 问题, 这时候你需要自己查询
    count 部分
    // page.setOptimizeCountSql(false);
    // 当 total 为小于 0 或者设置 setSearchCount(false) 分页插件不会进行 count
    查询
    // 要点!! 分页返回的对象与传入的对象是同一个
    return userMapper.selectPageVo(page, state);
}

```

# Sequence主键

## 实体主键支持Sequence

- oracle等数据库主键策略配置Sequence
- GlobalConfig配置KeyGenerator

```
1. GlobalConfig conf = new GlobalConfig();
   conf.setDbConfig(new GlobalConfig.DbConfig().setKeyGenerator(new
2. OracleKeyGenerator()));
```

- mybatis-plus-boot-starter[配置参考](#)

```
1.
2. @Bean
3. public OracleKeyGenerator oracleKeyGenerator(){
4.     return new OracleKeyGenerator();
5. }
6.
```

- 实体类配置主键Sequence, 指定主键@TableId(type=IdType.INPUT)//不能使用AUTO

```
1. @TableName("TEST_SEQUSER")
2. @KeySequence("SEQ_TEST")//类注解
3. public class TestSequser{
4.     @TableId(value = "ID", type = IdType.INPUT)
5.     private Long id;
6.
7. }
```

- 支持父类定义@KeySequence, 子类使用, 这样就可以几个表共用一个Sequence

```
1. @KeySequence("SEQ_TEST")
2. public abstract class Parent{
3.
4. }
5.
6. public class Child extends Parent{
7.
8. }
```

以上步骤就可以使用Sequence当主键了。

Spring MVC: xml配置, 请参考【[安装集成](#)】

## 如何使用Sequence作为主键，但是实体主键类型是String

自版本3.2.1开始, 此属性废除, 无需指定

也就是说, 表的主键是varchar2, 但是需要从sequence中取值

- 1. 实体定义@KeySequence 注解clazz指定类型String.class
- 2. 实体定义主键的类型String

```
1. @KeySequence(value = "SEQ_ORACLE_STRING_KEY", clazz = String.class)
2. public class YourEntity{
3.
4.     @TableId(value = "ID_STR", type = IdType.INPUT)
5.     private String idStr;
6.     ...
7. }
```

- 3. 正常配置GlobalConfig配置keyGenerator

```
1. @Bean
2. public GlobalConfig globalConfiguration() {
3.     GlobalConfig conf = new GlobalConfig();
4.     conf.setDbConfig(new GlobalConfig.DbConfig().setKeyGenerator(new
5. OracleKeyGenerator()));
6.     return conf;
7. }
```

# 自定义ID生成器

自3.2.1开始，配合ID\_WORKER，ID\_WORKER\_STR一起使用。

默认使用雪花算法

(com.baomidou.mybatisplus.core.incrementer.SnowflakeIdGenerator)。

```
1.  //方式一
2.  @Component
3.  public class CustomIdGenerator implements IdGenerator {
4.      @Override
5.      public long nextId(Object entity) {
6.          //实现自定义ID生成...
7.          return System.currentTimeMillis();
8.      }
9.  }
10.
11. //方式二
12. @Bean
13. public IdGenerator idGenerator() {
14.     return new CustomIdGenerator();
15. }
16.
```



- [热加载](#)
- [逻辑删除](#)
- [通用枚举](#)
- [字段类型处理器](#)
- [自动填充功能](#)
- [Sql 注入器](#)
- [攻击 SQL 阻断解析器](#)
- [性能分析插件](#)
- [执行 SQL 分析打印](#)
- [乐观锁插件](#)
- [动态数据源](#)
- [分布式事务](#)
- [多租户 SQL 解析器](#)
- [动态表名 SQL 解析器](#)
- [MybatisX 快速开发插件](#)

# 热加载

3.0.6 版本上移除了该功能,不过最新快照版已加回来并打上废弃标识, 3.1.0 版本上已完全移除

开启动态加载 mapper.xml

- 多数据源配置多个 MybatisMapperRefresh 启动 bean
- 默认情况下,eclipse保存会自动编译,idea需自己手动编译一次

## 1. 参数说明：

2. sqlSessionFactory:session工厂
3. mapperLocations:mapper匹配路径
4. enabled:是否开启动态加载 默认:false
5. delaySeconds:项目启动延迟加载时间 单位:秒 默认:10s
6. sleepSeconds:刷新时间间隔 单位:秒 默认:20s
7. 提供了两个构造,挑选一个配置进入spring配置文件即可：
- 8.

## 9. 构造1:

```
10. <bean class="com.baomidou.mybatisplus.spring.MybatisMapperRefresh">
11.     <constructor-arg name="sqlSessionFactory" ref="sqlSessionFactory"/>
12.     <constructor-arg name="mapperLocations"
13. value="classpath*:mybatis/mappers/**/*.xml"/>
14.     <constructor-arg name="enabled" value="true"/>
15. </bean>
```

## 16. 构造2:

```
17. <bean class="com.baomidou.mybatisplus.spring.MybatisMapperRefresh">
18.     <constructor-arg name="sqlSessionFactory" ref="sqlSessionFactory"/>
19.     <constructor-arg name="mapperLocations"
20. value="classpath*:mybatis/mappers/**/*.xml"/>
21.     <constructor-arg name="delaySeconds" value="10"/>
22.     <constructor-arg name="sleepSeconds" value="20"/>
23.     <constructor-arg name="enabled" value="true"/>
24. </bean>
```

# 逻辑删除

SpringBoot 配置方式:

- application.yml 加入配置(如果你的默认值和mp默认的一样,该配置可无):

```
1. mybatis-plus:
2.   global-config:
3.     db-config:
4.       logic-delete-value: 1 # 逻辑已删除值(默认为 1)
5.       logic-not-delete-value: 0 # 逻辑未删除值(默认为 0)
```

- 注册 Bean(3.1.1开始不再需要这一步):

```
1. import com.baomidou.mybatisplus.core.injector.ISqlInjector;
2. import com.baomidou.mybatisplus.extension.injector.LogicSqlInjector;
3. import org.springframework.context.annotation.Bean;
4. import org.springframework.context.annotation.Configuration;
5.
6. @Configuration
7. public class MyBatisPlusConfiguration {
8.
9.     @Bean
10.    public ISqlInjector sqlInjector() {
11.        return new LogicSqlInjector();
12.    }
13. }
```

- 实体类字段上加上 `@TableLogic` 注解

```
1. @TableLogic
2. private Integer deleted;
```

- 效果: 使用mp自带方法删除和查找都会附带逻辑删除功能 (自己写的xml不会)

```
1. example
2. 删除时 update user set deleted=1 where id =1 and deleted=0
3. 查找时 select * from user where deleted=0
```

附件说明

- 逻辑删除是为了方便数据恢复和保护数据本身价值等等的一种方案，但实际就是删除。
- 如果你需要再查出来就不应使用逻辑删除，而是以一个状态去表示。如： 员工离职，账号被锁定等都应该是一个状态字段，此种场景不应使用逻辑删除。
- 若确需查找删除数据，如老板需要查看历史所有数据的统计汇总信息，请单独手写sql。

# 通用枚举

解决了繁琐的配置，让 mybatis 优雅的使用枚举属性！

自 **3.1.0** 开始，可配置默认枚举处理类来省略扫描通用枚举配置 [默认枚举配置](#)

- 升级说明：

**3.1.0** 以下版本改变了原生默认行为，升级时请将默认枚举设置为 `EnumOrdinalTypeHandler`

- 影响用户：

实体中使用原生枚举

- 其他说明：

配置枚举包扫描的时候能提前注册使用注解枚举的缓存

- 推荐配置：

- 使用实现 `IEnum` 接口

- 推荐配置 `defaultEnumTypeHandler`

- 使用注解枚举处理

- 推荐配置 `typeEnumsPackage`

- 注解枚举处理与 `IEnum` 接口

- 推荐配置 `typeEnumsPackage`

- 与原生枚举混用

- 需配置 `defaultEnumTypeHandler` 与 `typeEnumsPackage`

## 1、申明通用枚举属性

方式一： 使用 `@EnumValue` 注解枚举属性 [完整示例](#) 

```

1. public enum GradeEnum {
2.
3.     PRIMARY(1, "小学"), SECONDORY(2, "中学"), HIGH(3, "高中");
4.
5.     GradeEnum(int code, String desc) {
6.         this.code = code;
7.         this.desc = desc;
8.     }
9.
10.     @EnumValue//标记数据库存的值是code

```

```

11.     private final int code;
12.     //。。。
13. }

```

方式二： 枚举属性，实现 IEnum 接口如下：

```

1.  public enum AgeEnum implements IEnum<Integer> {
2.      ONE(1, "一岁"),
3.      TWO(2, "二岁"),
4.      THREE(3, "三岁");
5.
6.      private int value;
7.      private String desc;
8.
9.      @Override
10.     public Integer getValue() {
11.         return this.value;
12.     }
13. }

```

实体属性使用枚举类型

```

1.  public class User{
2.      /**
3.       * 名字
4.       * 数据库字段：name varchar(20)
5.       */
6.      private String name;
7.
8.      /**
9.       * 年龄， IEnum接口的枚举处理
10.      * 数据库字段：age INT(3)
11.      */
12.      private AgeEnum age;
13.
14.
15.      /**
16.       * 年级， 原生枚举（带{@link com.baomidou.mybatisplus.annotation.EnumValue}）：
17.       * 数据库字段：grade INT(2)
18.       */
19.      private GradeEnum grade;
20. }

```

## 2、配置扫描通用枚举

- 注意!! spring mvc 配置参考, 安装集成 MybatisSqlSessionFactoryBean 枚举包扫描, spring boot 例子配置如下: 示例工程:

[mybatisplus-spring-boot](#) 

配置文件 resources/application.yml

```
1. mybatis-plus:
2.     # 支持统配符 * 或者 ; 分割
3.     typeEnumsPackage: com.baomidou.springboot.entity.enums
4.     ....
```

## 3、JSON序列化处理

### 一、Jackson

1. 在需要响应描述字段的get方法上添加@JsonValue注解即可

### 二、Fastjson

#### 1. 全局处理方式

```
1. FastJsonConfig config = new FastJsonConfig();
2. //设置WriteEnumUsingToString
3. config.setSerializerFeatures(SerializerFeature.WriteEnumUsingToString);
4. converter.setFastJsonConfig(config);
```

#### 2. 局部处理方式

```
1. @JSONField(serializeFeatures= SerializerFeature.WriteEnumUsingToString)
2. private UserStatus status;
```

以上两种方式任选其一, 然后在枚举中复写toString方法即可.

#### 3. JavaBean方式序列化枚举, 无需重写toString方法

```
1.      @JSONType(serializeEnumAsJavaBean = true)
2.      public enum GradeEnum {
```



# 字段类型处理器

类型处理器，用于 `JavaType` 与 `JdbcType` 之间的转换，用于 `PreparedStatement` 设置参数值和从 `ResultSet` 或 `CallableStatement` 中取出一个值，本文讲解 `mybatis-plus` 内置常用类型处理器如何通过 `TableField` 注解快速注入到 `mybatis` 容器中。

示例工程：

[mybatis-plus-sample-typehandler](#) 

- JSON 字段类型

```

1. @Data
2. @Accessors(chain = true)
3. @TableName(autoResultMap = true)
4. public class User {
5.     private Long id;
6.
7.     ...
8.
9.
10.    /**
11.     * 注意！！ 必须开启映射注解
12.     *
13.     * @TableName(autoResultMap = true)
14.     *
15.     * 以下两种类型处理器，二选一 也可以同时存在
16.     *
17.     * 注意！！选择对应的 JSON 处理器也必须存在对应 JSON 解析依赖包
18.     */
19.    @TableField(typeHandler = JacksonTypeHandler.class)
20.    // @TableField(typeHandler = FastjsonTypeHandler.class)
21.    private OtherInfo otherInfo;
22.
23. }
```

该注解对应了 XML 中写法为

```

<result column="other_info" jdbcType="VARCHAR" property="otherInfo"
1. typeHandler="com.baomidou.mybatisplus.extension.handlers.JacksonTypeHandler" />
2.
```



# 自动填充功能

示例工程：

[mybatis-plus-sample-auto-fill-metainfo](#) 

- 实现元对象处理器接口：  
com.baomidou.mybatisplus.core.handlers.MetaObjectHandler
- 注解填充字段 `@TableField(.. fill = FieldFill.INSERT)` 生成器策略部分也可以配置！

```
1. public class User {
2.
3.     // 注意！这里需要标记为填充字段
4.     @TableField(.. fill = FieldFill.INSERT)
5.     private String fillField;
6.
7.     ....
8. }
```

- 自定义实现类 MyMetaObjectHandler

```
1. @Component
2. public class MyMetaObjectHandler implements MetaObjectHandler {
3.
4.     private static final Logger LOGGER =
5.     LoggerFactory.getLogger(MyMetaObjectHandler.class);
6.
7.     @Override
8.     public void insertFill(MetaObject metaObject) {
9.         LOGGER.info("start insert fill ....");
10.         this.setFieldValByName("operator", "Jerry", metaObject); //版本号3.0.6以
11.         及之前的版本
12.         //this.setInsertFieldValByName("operator", "Jerry",
13.         metaObject); // @since 快照：3.0.7.2-SNAPSHOT, @since 正式版暂未发布3.0.7
14.     }
15.
16.     @Override
17.     public void updateFill(MetaObject metaObject) {
18.         LOGGER.info("start update fill ....");
19.         this.setFieldValByName("operator", "Tom", metaObject);
20.     }
21. }
```

```

        //this.setUpdateFieldValByName("operator", "Tom", metaObject);//@since
17. 快照 : 3.0.7.2-SNAPSHOT, @since 正式版暂未发布3.0.7
18.     }
19. }

```

#### 注意事项:

- 字段必须声明 `TableField` 注解, 属性 `fill` 选择对应策略, 该申明告知 `Mybatis-Plus` 需要预留注入 `SQL` 字段
- 填充处理器 `MyMetaObjectHandler` 在 Spring Boot 中需要声明 `@Component` 注入
- 必须使用父类的 `setFieldValByName()` 或者 `setInsertFieldValByName/setUpdateFieldValByName` 方法, 否则不会根据注解 `FieldFill.xxx` 来区分

```

1. public enum FieldFill {
2.     /**
3.      * 默认不处理
4.      */
5.     DEFAULT,
6.     /**
7.      * 插入填充字段
8.      */
9.     INSERT,
10.    /**
11.     * 更新填充字段
12.     */
13.    UPDATE,
14.    /**
15.     * 插入和更新填充字段
16.     */
17.    INSERT_UPDATE
18. }

```

# Sql 注入器

## 注入器配置

全局配置 `sqlInjector` 用于注入 `ISqlInjector` 接口的子类，实现自定义方法注入。

参考默认注入器 `DefaultSqlInjector` [↗](#)

- SQL 自动注入器接口 `ISqlInjector`

```

1. public interface ISqlInjector {
2.
3.     /**
4.      * <p>
5.      * 检查SQL是否注入(已经注入过不再注入)
6.      * </p>
7.      *
8.      * @param builderAssistant mapper 信息
9.      * @param mapperClass      mapper 接口的 class 对象
10.     */
11.     void inspectInject(MapperBuilderAssistant builderAssistant, Class<?>
12. mapperClass);
13. }
```

自定义自己的通用方法可以实现接口 `ISqlInjector` 也可以继承抽象类 `AbstractSqlInjector` 注入通用方法 `SQL 语句` 然后继承 `BaseMapper` 添加自定义方法，全局配置 `sqlInjector` 注入 MP 会自动将类所有方法注入到 `mybatis` 容器中。

参考[自定义BaseMapper示例](#) [↗](#))

# 攻击 SQL 阻断解析器

作用！阻止恶意的全表更新删除

```
1. @Bean
2. public PaginationInterceptor paginationInterceptor() {
3.     PaginationInterceptor paginationInterceptor = new PaginationInterceptor();
4.
5.     ...
6.
7.     List<ISqlParser> sqlParserList = new ArrayList<>();
8.     // 攻击 SQL 阻断解析器、加入解析链
9.     sqlParserList.add(new BlockAttackSqlParser() {
10.         @Override
11.         public void processDelete(Delete delete) {
12.             // 如果你想自定义做点什么，可以重写父类方法像这样子
13.             if ("user".equals(delete.getTable().getName())) {
14.                 // 自定义跳过某个表，其他关联表可以调用 delete.getTables() 判断
15.                 return ;
16.             }
17.             super.processDelete(delete);
18.         }
19.     });
20.     paginationInterceptor.setSqlParserList(sqlParserList);
21.
22.     ...
23.
24.     return paginationInterceptor;
25. }
```

# 性能分析插件

性能分析拦截器，用于输出每条 SQL 语句及其执行时间

该插件 **3.2.0** 以上版本移除推荐使用第三方扩展 **执行 SQL 分析打印** 功能

- 使用如下：

```
1. <plugins>
2.     ....
3.
4.     <!-- SQL 执行性能分析，开发环境使用，线上不推荐。 maxTime 指的是 sql 最大执行时长 -->
5.     <plugin
6.         interceptor="com.baomidou.mybatisplus.extension.plugins.PerformanceInterceptor">
7.             <property name="maxTime" value="100" />
8.             <!--SQL是否格式化 默认false-->
9.             <property name="format" value="true" />
10.        </plugin>
11. </plugins>
```

```
1. //Spring boot方式
2. @EnableTransactionManagement
3. @Configuration
4. @MapperScan("com.baomidou.cloud.service.*.mapper*")
5. public class MybatisPlusConfig {
6.
7.     /**
8.      * SQL执行效率插件
9.      */
10.    @Bean
11.    @Profile({"dev", "test"})// 设置 dev test 环境开启
12.    public PerformanceInterceptor performanceInterceptor() {
13.        return new PerformanceInterceptor();
14.    }
15. }
```

注意！参数说明：

- 参数：maxTime SQL 执行最大时长，超过自动停止运行，有助于发现问题。
- 参数：format SQL 是否格式化，默认false。

- 该插件只用于开发环境，不建议生产环境使用。



# 执行 SQL 分析打印

该功能依赖 **p6spy** 组件，完美的输出打印 SQL 及执行时长 **3.1.0** 以上版本

示例工程：

[mybatis-plus-sample-crud](#) 

- p6spy 依赖引入Maven：

```
1. <dependency>
2.   <groupId>p6spy</groupId>
3.   <artifactId>p6spy</artifactId>
4.   <version>最新版本</version>
5. </dependency>
```

Gradle：

```
1. compile group: 'p6spy', name: 'p6spy', version: '最新版本'
```

- application.yml 配置：

```
1. spring:
2.   datasource:
3.     driver-class-name: com.p6spy.engine.spy.P6SpyDriver
4.     url: jdbc:p6spy:h2:mem:test
5.     ...
```

- spy.properties 配置：

```
1. module.log=com.p6spy.engine.logging.P6LogFactory,com.p6spy.engine.outage.P6Outage
2. # 自定义日志打印
3. logMessageFormat=com.baomidou.mybatisplus.extension.p6spy.P6SpyLogger
4. #日志输出到控制台
5. appender=com.baomidou.mybatisplus.extension.p6spy.StdoutLogger
6. # 使用日志系统记录 sql
7. #appender=com.p6spy.engine.spy.appender.Slf4JLogger
8. # 设置 p6spy driver 代理
9. deregisterdrivers=true
10. # 取消JDBC URL前缀
```

```
11. useprefix=true
    # 配置记录 Log 例外,可去掉的结果集有
12. error,info,batch,debug,statement,commit,rollback,result,resultset.
13. excludecategories=info,debug,result,batch,resultset
14. # 日期格式
15. dateformat=yyyy-MM-dd HH:mm:ss
16. # 实际驱动可多个
17. #driverlist=org.h2.Driver
18. # 是否开启慢SQL记录
19. outagedetection=true
20. # 慢SQL记录标准 2 秒
21. outagedetectioninterval=2
```

注意！

- driver-class-name 为 p6spy 提供的驱动类
- url 前缀为 jdbc:p6spy 跟着冒号为对应数据库连接地址
- 该插件有性能损耗，不建议生产环境使用。

# 乐观锁插件

## 主要适用场景

意图：

当要更新一条记录的时候，希望这条记录没有被别人更新

乐观锁实现方式：

- 取出记录时，获取当前version
- 更新时，带上这个version
- 执行更新时， `set version = newVersion where version = oldVersion`
- 如果version不对，就更新失败乐观锁配置需要2步 记得两步

## 1. 插件配置

spring xml:

```
<bean
1. class="com.baomidou.mybatisplus.extension.plugins.OptimisticLockerInterceptor"/>
```

spring boot:

```
1. @Bean
2. public OptimisticLockerInterceptor optimisticLockerInterceptor() {
3.     return new OptimisticLockerInterceptor();
4. }
```

## 2. 注解实体字段 @Version 必须要！

```
1. @Version
2. private Integer version;
```

特别说明：

- 支持的数据类型只有：`int, Integer, long, Long, Date, Timestamp, LocalDateTime`
- 整数类型下 `newVersion = oldVersion + 1`

- `newVersion` 会回写到 `entity` 中
- 仅支持 `updateById(id)` 与 `update(entity, wrapper)` 方法
- 在 `update(entity, wrapper)` 方法下, `wrapper` 不能复用!!!

## 示例

示例Java代码 ( 参考[test case](#) 代码 )

```
1. int id = 100;
2. int version = 2;
3.
4. User u = new User();
5. u.setId(id);
6. u.setVersion(version);
7. u.setXXX(xxx);
8.
9. if(userService.updateById(u)){
10.     System.out.println("Update successfully");
11. }else{
12.     System.out.println("Update failed due to modified by others");
13. }
```

示例SQL原理

```
1. update tbl_user set name = 'update',version = 3 where id = 100 and version = 2
```

## 动态数据源

# dynamic datasource

一个基于springboot的快速集成多数据源的启动器

build error maven central 2.5.7 license apache JDK 1.7+ springBoot 1.4+ 1.5+ 2.0+

Github | 码云Gitee

## 简介

dynamic-datasource-spring-boot-starter 是一个基于springboot的快速集成多数据源的启动器。

其支持 Jdk 1.7+, SpringBoot 1.4.x 1.5.x 2.0.x。最新版为

maven central 2.5.7

示例项目 可参考项目下的samples目录。

示例项目 可参考项目下的samples目录。

示例项目 可参考项目下的samples目录。

## 优势

网上关于动态数据源的切换的文档有很多，核心只有两种。

- 构建多套环境，优势是方便控制也容易集成一些简单的分布式事务，缺点是非动态同时代码量较多，配置难度大。
- 基于spring提供原生的 `AbstractRoutingDataSource`，参考一些文档自己实现切换。如果你的数据源较少，场景不复杂，选择以上任意一种都可以。如果你需要更多特性，请尝试本动态数据源。
- 数据源分组，适用于多种场景 纯粹多库 读写分离 一主多从 混合模式。
- 简单集成Druid数据源监控多数据源，简单集成Mybatis-Plus简化单表，简单集成P6sy格式化

sql，简单集成Jndi数据源。

- 简化Druid和HikariCp配置，提供全局参数配置。
- 提供自定义数据源来源(默认使用yaml或properties配置)。
- 项目启动后能动态增减数据源。
- 使用spel动态参数解析数据源，如从session，header和参数中获取数据源。（多租户架构神器）
- 多层数据源嵌套切换。（一个业务ServiceA调用ServiceB，ServiceB调用ServiceC，每个Service都是不同的数据源）
- 使用正则匹配或spel表达式来切换数据源（实验性功能）。

## 劣势

不能使用多数据源事务（同一个数据源下能使用事务），网上其他方案也都不能提供。

如果你需要使用到分布式事务，那么你的架构应该到了微服务化的时候了。

如果呼声强烈，项目达到800 star，作者考虑集成分布式事务。

PS：如果您只是几个数据库但是有强烈的需求分布式事务，建议还是使用传统方式自己构建多套环境集成atomic这类，网上百度很多。

## 约定

- 本框架只做 切换数据源 这件核心的事情，并不限制你的具体操作，切换了数据源可以做任何CRUD。
- 配置文件所有以下划线 `_` 分割的数据源 首部 即为组的名称，相同组名称的数据源会放在一个组下。
- 切换数据源即可是组名，也可是具体数据源名称，切换时默认采用负载均衡机制切换。
- 默认的数据源名称为 **master**，你可以通过spring.datasource.dynamic.primary修改。
- 方法上的注解优先于类上注解。

## 建议

强烈建议在 主从模式 下遵循普遍的规则，以便他人能更轻易理解你的代码。

主数据库 建议 只执行 `INSERT` `UPDATE` `DELETE` 操作。

从数据库 建议 只执行 `SELECT` 操作。

## 使用方法

- 引入dynamic-datasource-spring-boot-starter。

```

1. <dependency>
2.   <groupId>com.baomidou</groupId>
3.   <artifactId>dynamic-datasource-spring-boot-starter</artifactId>
4.   <version>${version}</version>
5. </dependency>

```

- 配置数据源。

```

1. spring:
2.   datasource:
3.     dynamic:
4.       primary: master #设置默认的数据源或者数据源组, 默认值即为master
5.       datasource:
6.         master:
7.           username: root
8.           password: 123456
9.           driver-class-name: com.mysql.jdbc.Driver
10.          url: jdbc:mysql://xx.xx.xx.xx:3306/dynamic
11.        slave_1:
12.          username: root
13.          password: 123456
14.          driver-class-name: com.mysql.jdbc.Driver
15.          url: jdbc:mysql://xx.xx.xx.xx:3307/dynamic
16.        slave_2:
17.          username: root
18.          password: 123456
19.          driver-class-name: com.mysql.jdbc.Driver
20.          url: jdbc:mysql://xx.xx.xx.xx:3308/dynamic
21.        #.....省略
22.        #以上会配置一个默认库master, 一个组slave下有两个子库slave_1, slave_2

```

	# 多主多从	纯粹多库 (记得设置primary)	混合配
1. 置			
2. spring:	spring:	spring:	
3.   datasource:	datasource:	datasource:	
4.     dynamic:	dynamic:	dynamic:	

datasource:	datasource:
5. datasource:	
master_1:	mysql:
6. master:	
master_2:	oracle:
7. slave_1:	
slave_1:	sqlserver:
8. slave_2:	
slave_2:	postgresql:
9. oracle_1:	
slave_3:	h2:
10. oracle_2:	

- 使用 `@DS` 切换数据源。`@DS` 可以注解在方法上和类上，同时存在方法注解优先于类上注解。

注解在service实现或mapper接口方法上，但强烈不建议同时在service和mapper注解。（可能会有问题）

注解	结果
没有@DS	默认数据源
@DS("dsName")	dsName可以为组名也可以为具体某个库的名称

```

1. @Service
2. @DS("slave")
3. public class UserServiceImpl implements UserService {
4.
5.     @Autowired
6.     private JdbcTemplate jdbcTemplate;
7.
8.     public List<Map<String, Object>> selectAll() {
9.         return jdbcTemplate.queryForList("select * from user");
10.    }
11.
12.    @Override
13.    @DS("slave_1")
14.    public List<Map<String, Object>> selectByCondition() {
15.        return jdbcTemplate.queryForList("select * from user where age >10");
16.    }
17. }

```

在mybatis环境下也可注解在mapper接口层。

```

1. @DS("slave")

```



```
2. public interface UserMapper {
3.
4.     @Insert("INSERT INTO user (name,age) values (#{name},#{age})")
5.     boolean addUser(@Param("name") String name, @Param("age") Integer age);
6.
7.     @Update("UPDATE user set name=#{name}, age=#{age} where id =#{id}")
8.     boolean updateUser(@Param("id") Integer id, @Param("name") String name,
9.         @Param("age") Integer age);
10.
11.    @Delete("DELETE from user where id =#{id}")
12.    boolean deleteUser(@Param("id") Integer id);
13.
14.    @Select("SELECT * FROM user")
15.    @DS("slave_1")
16.    List<User> selectAll();
17. }
```

---

赶紧集成体验一下吧！ 如果需要更多功能请继续往下看！

---

- Druid集成, MybatisPlus集成, 动态增减数据源等等更多更细致的文档在 [这里 点击查看](#)
- 项目Javadoc一览 [点击查看](#)

# 分布式事务

暂时支持 rabbit 实现可靠消息分布式事务 3.1.1 以上版本


示例工程：

[mybatis-plus-sample-dts-rabbit](#) 

- 更多待完善

# 多租户 SQL 解析器

- 这里配合 分页拦截器 使用， spring boot 例子配置如下：示例工程：

[mybatis-plus-sample-tenant](#) 

[mybatisplus-spring-boot](#) 

```

1. @Bean
2. public PaginationInterceptor paginationInterceptor() {
3.     PaginationInterceptor paginationInterceptor = new PaginationInterceptor();
4.     /*
5.      * 【测试多租户】 SQL 解析处理拦截器<br>
6.      * 这里固定写成住户 1 实际情况你可以从cookie读取，因此数据看不到 【 麻花藤 】 这条记录
7.      * （ 注意观察 SQL ）<br>
8.      */
9.     List<ISqlParser> sqlParserList = new ArrayList<>();
10.    TenantSqlParser tenantSqlParser = new TenantSqlParser();
11.    tenantSqlParser.setTenantHandler(new TenantHandler() {
12.        @Override
13.        public Expression getTenantId(boolean where) {
14.            // 该 where 条件 3.2.0 版本开始添加的，用于分区是否为在 where 条件中使用
15.            // 此判断用于支持返回多个租户 ID 场景，具体使用查看示例工程
16.            return new LongValue(1L);
17.        }
18.
19.        @Override
20.        public String getTenantIdColumn() {
21.            return "tenant_id";
22.        }
23.
24.        @Override
25.        public boolean doTableFilter(String tableName) {
26.            // 这里可以判断是否过滤表
27.            /*
28.             * if ("user".equals(tableName)) {
29.                 return true;
30.             }*/
31.            return false;

```

```

32.     });
33.     sqlParserList.add(tenantSqlParser);
34.     paginationInterceptor.setSqlParserList(sqlParserList);
35.     paginationInterceptor.setSqlParserFilter(new ISqlParserFilter() {
36.         @Override
37.         public boolean doFilter(MetaObject metaObject) {
38.             MappedStatement ms =
39.                 SqlParserHelper.getMappedStatement(metaObject);
40.             // 过滤自定义查询此时无租户信息约束【麻花藤】出现
41.             if
42.                 ("com.baomidou.springboot.mapper.UserMapper.selectListBySQL".equals(ms.getId()))
43.             {
44.                 return true;
45.             }
46.             return false;
47.         }
48.     });
49.     return paginationInterceptor;
50. }

```

- 相关 SQL 解析如多租户可通过 `@SqlParser(filter=true)` 排除 SQL 解析，注意！！全局配置 `sqlParserCache` 设置为 `true` 才生效。（3.1.1开始不再需要这一步）

```

1. # 开启 SQL 解析缓存注解生效
2. mybatis-plus:
3.     global-config:
4.         sql-parser-cache: true

```

# 动态表名 SQL 解析器

该功能解决动态表名支持 **3.1.1** 以上版本

简单示例：

[mybatis-plus-sample-dynamic-tablename](#) 

源码文件：

[DynamicTableNameParser](#) 

- 具体使用参考多租户实现 `ITableNameHandler` 接口注入到 `DynamicTableNameParser` 处理器链中，将动态表名解析器注入到 MP 解析链。

注意事项：

- 原理为解析替换设定表名为处理器的返回表名，表名建议可以定义复杂一些避免误替换
- 例如：真实表名为 `user` 设定为 `mp_dt_user` 处理器替换为 `user_2019` 等

# MybatisX 快速开发插件

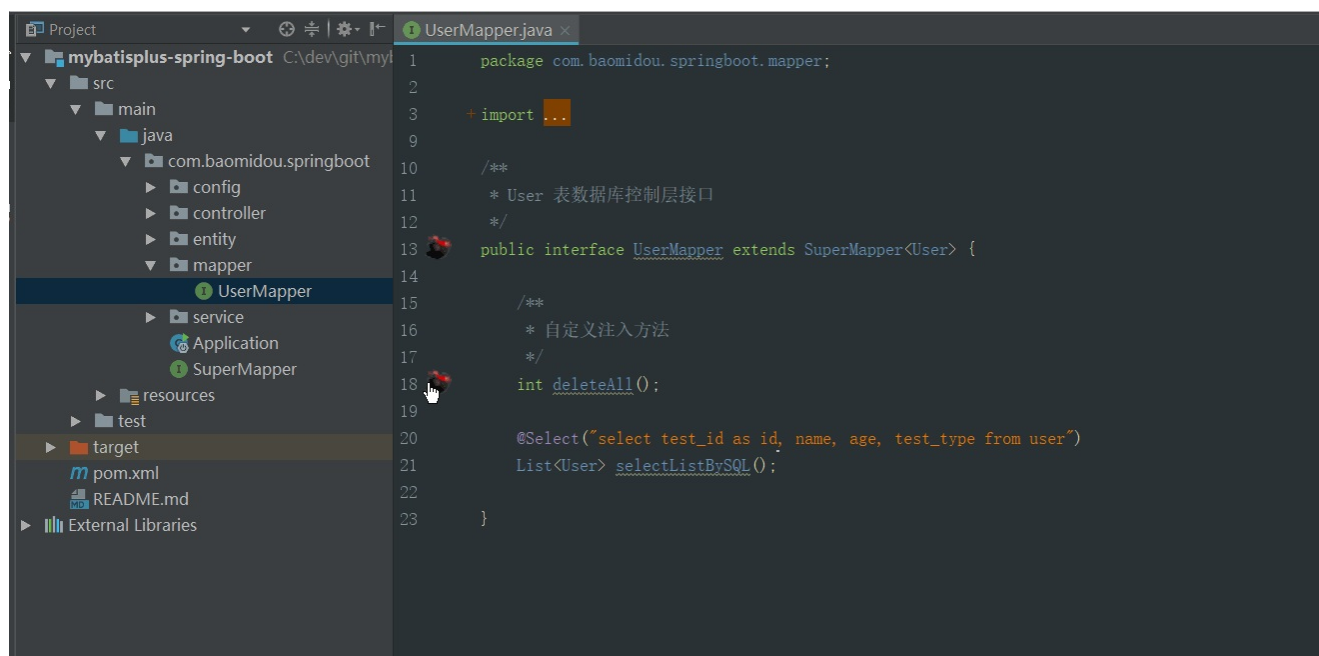
MybatisX 是一款基于 IDEA 的快速开发插件，为效率而生。

安装方法：打开 IDEA，进入 File -> Settings -> Plugins -> Browse Repositories，输入 `mybatisx` 搜索并安装。

如果各位觉得好用，请为该插件打一个[五分好评](#) 哦！源码地址：[MybatisX 源码](#)

## 功能

- Java 与 XML 调回跳转
- Mapper 方法自动生成 XML



## 计划支持

- 连接数据源之后 xml 里自动提示字段
- sql 增删改查
- 集成 MP 代码生成
- 其他

- [常见问题](#)
- [捐赠支持](#)

## 常见问题

---

- 如何排除非表中字段？
- 排除实体父类属性
- 出现 Invalid bound statement (not found) 异常
- 自定义 SQL 无法执行
- 启动时异常
- 关于 Long 型主键填充不生效的问题
- ID\_WORKER 生成主键太长导致 js 精度丢失
- 插入或更新的字段有 空字符串 或者 null
- 字段类型为 bit、tinyint(1) 时映射为 boolean 类型
- 出现 2 个 limit 语句
- insert 后如何返回主键
- MP 如何查指定的几个字段
- mapper 层二级缓存问题
- mapper 层二级缓存刷新问题
- Cause: org.apache.ibatis.type.TypeException:Error setting null for parameter #1 with JdbcType OTHER
- 自定义 sql 里使用 Page 对象传参无法获取
- 如何使用：【Map下划线自动转驼峰】
- 在 wrapper 中如何使用 limit 限制 SQL
- 通用 insertBatch 为什么放在 service 层处理
- 逻辑删除下 自动填充 功能没有效果
- 3.x数据库关键字如何处理？
- MybatisPlusException: Your property named "xxx" cannot find the corresponding database column name!
- Error attempting to get column 'create\_time' from result set. Cause: java.sql.SQLFeatureNotSupportedException

## 如何排除非表中字段？

---

以下三种方式选择一种即可：

- 使用 `transient` 修饰

```
1. private transient String noColumn;
```

- 使用 `static` 修饰



```
1. private static String noColumn;
```

- 使用 `TableField` 注解

```
1. @TableField(exist=false)
2. private String noColumn;
```

## 排除实体父类属性

```
1. /**
2.  * 忽略父类 createTime 字段映射
3.  */
4. private transient String createTime;
```

## 出现 Invalid bound statement (not found) 异常

不要怀疑，正视自己，这个异常肯定是你插入的姿势不对.....

- 检查是不是引入 jar 冲突
- 检查 Mapper.java 的扫描路径
  - 方法一：在 `Configuration` 类上使用注解 `MapperScan`

```
1. @Configuration
2. @MapperScan("com.yourpackage.*.mapper")
3. public class YourConfigClass{
4.     ...
5. }
```

- 方法二：在 `Configuration` 类里面，配置 `MapperScannerConfigurer` ([查看示例](#))

```
1. @Bean
2. public MapperScannerConfigurer mapperScannerConfigurer(){
3.     MapperScannerConfigurer scannerConfigurer = new MapperScannerConfigurer();
4.     //可以通过环境变量获取你的mapper路径, 这样mapper扫描可以通过配置文件配置了
5.     scannerConfigurer.setBasePackage("com.yourpackage.*.mapper");
6.     return scannerConfigurer;
```

```
7. }
```

- 检查是否指定了主键？如未指定，则会导致 `selectById` 相关 ID 无法操作，请用注解 `@TableId` 注解表 ID 主键。当然 `@TableId` 注解可以没有！但是你的主键必须叫 `id`（忽略大小写）
- `SqlSessionFactory` 不要使用原生的，请使用 `MybatisSqlSessionFactory`
- 检查是否自定义了 `SqlInjector`，是否复写了 `getMethodList()` 方法，该方法里是否注入了你需要的方法（可参考 `DefaultSqlInjector`）

## 自定义 SQL 无法执行

问题描述：指在 XML 中里面自定义 SQL，却无法调用。本功能同 `MyBatis` 一样需要配置 XML 扫描路径：

- Spring MVC 配置（参考 [mybatisplus-spring-mvc](#) ）

```
<bean id="sqlSessionFactory"
1. class="com.baomidou.mybatisplus.spring.MybatisSqlSessionFactoryBean">
2.   <property name="dataSource" ref="dataSource" />
3.   <property name="typeAliasesPackage" value="xxx.entity" />
4.   <property name="mapperLocations" value="classpath*:mybatis/**/*.xml"/>
5.   ...
6. </bean>
```

- Spring Boot 配置（参考 [mybatisplus-spring-boot](#) ）

```
1. mybatis-plus:
2.   mapper-locations: classpath*:mapper/**/*.xml
```

- 对于 `IDEA` 系列编辑器，XML 文件是不能放在 `java` 文件夹中的，IDEA 默认不会编译源码文件夹中的 XML 文件，可以参照以下方式解决：
  - 将配置文件放在 `resource` 文件夹中
  - 对于 Maven 项目，可指定 POM 文件的 `resource`

```
1. <build>
2.   <resources>
3.     <resource>
4.       <!-- xml放在java目录下-->
```

```

5.         <directory>src/main/java</directory>
6.         <includes>
7.             <include>**/*.xml</include>
8.         </includes>
9.     </resource>
10.    <!-- 指定资源的位置（xml放在resources下，可以不用指定） -->
11.    <resource>
12.        <directory>src/main/resources</directory>
13.    </resource>
14. </resources>
15. </build>

```

注意！Maven 多模块项目的扫描路径需以 `classpath*` 开头（即加载多个 jar 包下的 XML 文件）

## 启动时异常

- 异常一：

```

java.lang.ClassCastException:
sun.reflect.generics.reflectiveObjects.TypeVariableImpl cannot be cast to
java.lang.Class

```

MapperScan 需要排除 `com.baomidou.mybatisplus.mapper.BaseMapper` 类 及其 子类（自定义公共 Mapper），比如：

```

import com.baomidou.mybatisplus.core.mapper.BaseMapper;

public interface SuperMapper<T> extends BaseMapper<T> {
    // your methods
}

```

- 异常二：

Injection of autowired

原因：低版本不支持泛型注入，请升级 Spring 版本到 4+ 以上。

- 异常三：

```
java.lang.NoSuchMethodError:
org.apache.ibatis.session.Configuration.getDefaultScriptingLanguageInstance
Lorg/apache/ibatis/scripting/LanguageDriver
```

版本引入问题：3.4.1 版本里没有，3.4.2 里面才有！

## 关于 Long 型主键填充不生效的问题

检查是不是用了 `long` 而不是 `Long` ！

`long` 类型默认值为 0，而 MP 只会判断是否为 `null`

## ID\_WORKER 生成主键太长导致 js 精度丢失

JavaScript 无法处理 Java 的长整型 Long 导致精度丢失，具体表现为主键最后两位永远为 0，  
解决思路：Long 转为 String 返回

- FastJson 处理方式

```
@Override
public void configureMessageConverters(List<HttpMessageConverter<?>>
converters) {
    FastJsonHttpMessageConverter fastJsonConverter = new
FastJsonHttpMessageConverter();
    FastJsonConfig fjc = new FastJsonConfig();
    // 配置序列化策略
    fjc.setSerializerFeatures(SerializerFeature.BrowserCompatible);
    fastJsonConverter.setFastJsonConfig(fjc);
    converters.add(fastJsonConverter);
}
```

- Jackson 处理方式

- 方式一

```
// 注解处理，这里可以配置公共 BaseEntity 处理
@JsonSerialize(using=ToStringSerializer.class)
public long getId() {
    return id;
}
```

- 方式二

```
// 全局配置序列化返回 JSON 处理
final ObjectMapper objectMapper = new ObjectMapper();
SimpleModule simpleModule = new SimpleModule();
simpleModule.addSerializer(Long.class, ToStringSerializer.instance);
objectMapper.registerModule(simpleModule);
```

- 比较一般的处理方式：增加一个 `public String getIdStr()` 方法，前台获取 `idStr`

## 插入或更新的字段有 空字符串 或者 null

FieldStrategy 有三种策略：

- IGNORED：忽略
- NOT\_NULL：非 NULL，默认策略
- NOT\_EMPTY：非空当用户有更新字段为 空字符串 或者 `null` 的需求时，需要对 `FieldStrategy` 策略进行调整：
- 方式一：调整全局的验证策略

注入配置 GlobalConfiguration 属性 fieldStrategy

- 方式二：调整字段验证注解

根据具体情况，在需要更新的字段中调整验证注解，如验证非空：

```
@TableField(strategy=FieldStrategy.NOT_EMPTY)
```

- 方式三：使用 `UpdateWrapper` (3.x)

使用以下方法来进行更新或插入操作：

```
//updateAllColumnById(entity) // 全部字段更新：3.0已经移除
mapper.update(
    new User().setName("mp").setAge(3),
    Wrappers.<User>lambdaUpdate()
        .set(User::getEmail, null) //把email设置成null
        .eq(User::getId, 2)
);
//也可以参考下面这种写法
mapper.update(
    null,
    Wrappers.<User>lambdaUpdate()
        .set(User::getAge, 3)
        .set(User::getName, "mp")
        .set(User::getEmail, null) //把email设置成null
        .eq(User::getId, 2)
);
```

## 字段类型为 bit、tinyint(1) 时映射为 boolean 类型

默认mysql驱动会把tinyint(1)字段映射为boolean：0=false，非0=true

MyBatis 是不会自动处理该映射，如果不想把tinyint(1)映射为boolean类型：

- 修改类型tinyint(1)为tinyint(2)或者int
- 需要修改请求连接添加参数 `tinyInt1isBit=false` ，如下：

```
jdbc:mysql://127.0.0.1:3306/mp?tinyInt1isBit=false
```

## 出现 2 个 limit 语句

原因：配了 2 个分页拦截器！检查配置文件或者代码，只留一个！

## insert 后如何返回主键

insert 后主键会自动 set 到实体的 ID 字段，所以你只需要 getId() 就好

## MP 如何查指定的几个字段

## EntityWrapper.sqlSelect 配置你想要查询的字段

```
//2.x
EntityWrapper<H2User> ew = new EntityWrapper<>();
ew.setSqlSelect("test_id as id, name, age");//只查询3个字段
List<H2User> list = userService.selectList(ew);
for(H2User u:list){
    Assert.assertNotNull(u.getId());
    Assert.assertNotNull(u.getName());
    Assert.assertNull(u.getPrice()); // 这个字段没有查询出来
}

//3.x
mapper.selectList(
    Wrappers.<User>lambdaQuery()
        .select(User::getId, User::getName)
);
//或者使用QueryWrapper
mapper.selectList(
    new QueryWrapper<User>()
        .select("id","name")
);
```

## mapper 层二级缓存问题

我们建议缓存放到 service 层，你可以自定义自己的 BaseServiceImpl 重写注解父类方法，继承自己的实现。

当然如果你是一个极端分子，请使用 CachePaginationInterceptor 替换默认分页，这样支持分页缓存。

## mapper 层二级缓存刷新问题

如果你按照 mybatis 的方式配置第三方二级缓存，并且使用 2.0.9 以上的版本，则会发现自带的方法无法更新缓存内容，那么请按如下方式解决（二选一）：

1.在代码中 mybatis 的 mapper 层添加缓存注释，声明 implementation 或 eviction 的值为 cache 接口的实现类

```
@CacheNamespace(implementation=MybatisRedisCache.class,eviction=MybatisRedisCache.class)
public interface DataResourceMapper extends BaseMapper<DataResource>{}
```

2. 在对应的 mapper.xml 中将原有注释修改为链接式声明, 以保证 xml 文件里的缓存能够正常

```
<cache-ref namespace="com.mst.cms.dao.DataResourceMapper"></cache-ref>
```

## Cause:

```
org.apache.ibatis.type.TypeException:Error
setting null for parameter #1 with JdbcType
OTHER
```

配置 jdbcTypeForNull=NULLSpring Bean 配置方式:

```
MybatisConfiguration configuration = new MybatisConfiguration();
configuration.setDefaultScriptingLanguage(MybatisXMLLanguageDriver.class);
configuration.setJdbcTypeForNull(JdbcType.NULL);
configuration.setMapUnderscoreToCamelCase(true); //开启下划线转驼峰
sqlSessionFactory.setConfiguration(configuration);
```

yaml 配置

```
mybatis-plus:
  configuration:
    jdbc-type-for-null: 'null'
```

## 自定义 sql 里使用 Page 对象传参无法获取

Page 对象是继承 RowBounds, 是 Mybatis 内置对象, 无法在 mapper 里获取请使用自定义 Map/对象, 或者通过@Param("page") int page,size 来传参

## 如何使用:【Map下划线自动转驼峰】

指的是: `resultType="java.util.Map"`

- spring boot



```
@Bean
public ConfigurationCustomizer configurationCustomizer() {
    return i -> i.setObjectWrapperFactory(new
    MybatisMapWrapperFactory());
}
```

## 在 wrapper 中如何使用 limit 限制 SQL

```
// 取 1 条数据
wrapper.last("limit 1");
```

## 通用 insertBatch 为什么放在 service 层处理

- SQL 长度有限制海量数据量单条 SQL 无法执行，就算可执行也容易引起内存泄露 JDBC 连接超时等
- 不同数据库对于单条 SQL 批量语法不一样不利于通用
- 目前的解决方案：循环预处理批量提交，虽然性能比单 SQL 慢但是可以解决以上问题。

## 逻辑删除下 自动填充 功能没有效果

- 自动填充的实现方式是填充到入参的 `entity` 内, 由于 `baseMapper` 提供的删除接口入参不是 `entity` 所以逻辑删除无效
- 如果你想要使用自动填充有效：
  - 方式一：使用update方法： `UpdateWrapper.set("logicDeleteColumn", "deleteValue")`
  - 方式二：配合Sql注入器并使用我们提供的 `com.baomidou.mybatisplus.extension.injector.methods.LogicDeleteByIdWithFill` 类注意该类只能填充指定了自动填充的字段, 其他字段无效
- 方式2下：Java Config Bean 配置
  - 配置自定义的 `SqlInjector`

```

@Bean
public LogicSqlInjector logicSqlInjector(){
    return new LogicSqlInjector() {
        /**
         * 注入自定义全局方法
         */
        @Override
        public List<AbstractMethod> getMethodList() {
            List<AbstractMethod> methodList = super.getMethodList();
            methodList.add(new LogicDeleteByIdWithFill());
            return methodList;
        }
    };
}

```

- 配置自己的全局 baseMapper 并使用

```

public interface MyBaseMapper<T> extends BaseMapper<T> {

    /**
     * 自定义全局方法
     */
    int deleteByIdWithFill(T entity);
}

```

### 3.x数据库关键字如何处理？

在以前的版本是自动识别关键字进行处理的，但是3.x移除了这个功能。

- 不同的数据库对关键字的处理不同, 很难维护。
- 在数据库设计时候本身不建议使用关键字。
- 交由用户去处理关键字。

```

@TableField(value = "`status`")
private Boolean status;

```

MybatisPlusException: Your property named "xxx" cannot find the corresponding

## database column name!

---

针对3.1.1以及后面的版本：

现象： 单元测试没问题，启动服务器进行调试就出现这个问题

原因： dev-tools, 3.1.1+针对字段缓存，使用.class来作为key替换了原来的className，而使用dev-tools会把.class使用不同的classLoader加载，导致可能出现找不到的情况


解决方案： 去掉dev-tools插件

## Error attempting to get column 'create\_time' from result set. Cause: java.sql.SQLFeatureNotSupportedException

---

3.1.0之前版本没问题，针对3.1.1以及后续版本出现上述问题

现象： 集成druid数据源，使用3.1.0之前版本没问题，升级mp到3.1.1+后，运行时报错：java.sql.SQLFeatureNotSupportedException

原因： mp3.1.1+使用了新版jdbc，LocalDateTime等新日期类型处理方式升级，但druid不支持，[参考issue](#) 

解决方案： 1. 坐等druid升级解决这个问题；2.保持mp版本3.1.0；3.紧跟mp版本，换掉druid数据源



## 捐赠支持

您的支持是鼓励我们前行的动力，无论金额多少都足够表达您这份心意。



项目的发展离不开你的支持，  
请作者喝杯咖啡吧！



支付宝扫码捐赠



微信扫码捐赠

## 配置

- [使用配置](#)
- [代码生成器配置](#)

# 使用配置

---

本文讲解了 `MyBatis-Plus` 在使用过程中的配置选项，其中，部分配置继承自 `MyBatis` 原生所支持的配置

- [基本配置](#)
- [使用方式](#)
- [Configuration](#)
- [GlobalConfig](#)
- [DbConfig](#)

## 基本配置

---

本部分配置包含了大部分用户的常用配置，其中一部分为 MyBatis 原生所支持的配置

# 使用方式

Spring Boot:

```

1. mybatis-plus:
2.     .....
3.     configuration:
4.     .....
5.     global-config:
6.     .....
7.     db-config:
8.     .....
```

Spring MVC:

```

    <bean id="sqlSessionFactory"
1.   class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
2.       <property name="configuration" ref="configuration"/> <!-- 非必须 -->
3.       <property name="globalConfig" ref="globalConfig"/> <!-- 非必须 -->
4.       .....
5.   </bean>
6.
    <bean id="configuration"
7.   class="com.baomidou.mybatisplus.core.MybatisConfiguration">
8.       .....
9.   </bean>
10.
    <bean id="globalConfig"
11.  class="com.baomidou.mybatisplus.core.config.GlobalConfig">
12.      <property name="dbConfig" ref="dbConfig"/> <!-- 非必须 -->
13.      .....
14.  </bean>
15.
    <bean id="dbConfig"
16.  class="com.baomidou.mybatisplus.core.config.GlobalConfig.DbConfig">
17.      .....
18.  </bean>
```

## configLocation

- 类型: `String`



- 默认值: `null` MyBatis 配置文件位置, 如果您有单独的 MyBatis 配置, 请将其路径配置到 `configLocation` 中. MyBatis Configuration 的具体内容请参考[MyBatis 官方文档](#)

## mapperLocations

- 类型: `String[]`
- 默认值: `[]` MyBatis Mapper 所对应的 XML 文件位置, 如果您在 Mapper 中有自定义方法(XML 中有自定义实现), 需要进行该配置, 告诉 Mapper 所对应的 XML 文件位置

Maven 多模块项目的扫描路径需以 `classpath*:` 开头 (即加载多个 jar 包下的 XML 文件)

## typeAliasesPackage

- 类型: `String`
- 默认值: `null` MyBatis 别名包扫描路径, 通过该属性可以给包中的类注册别名, 注册后在 Mapper 对应的 XML 文件中可以直接使用类名, 而不用使用全限定的类名(即 XML 中调用的时候不用包含包名)

## typeAliasesSuperType

- 类型: `Class<?>`
- 默认值: `null` 该配置请和 `typeAliasesPackage` 一起使用, 如果配置了该属性, 则仅仅会扫描路径下以该类作为父类的域对象

## typeHandlersPackage

- 类型: `String`
- 默认值: `null` TypeHandler 扫描路径, 如果配置了该属性, `SqlSessionFactoryBean` 会把该包下面的类注册为对应的 TypeHandler

TypeHandler 通常用于自定义类型转换。

## typeEnumsPackage

- 类型: `String`
- 默认值: `null` 枚举类 扫描路径, 如果配置了该属性, 会将路径下的枚举类进行注入, 让实体类字段能够简单快捷的使用枚举属性

## checkConfigLocation Spring Boot Only

- 类型: `boolean`
- 默认值: `false` 启动时是否检查 MyBatis XML 文件的存在, 默认不检查

## executorType Spring Boot Only

- 类型: `ExecutorType`
- 默认值: `simple` 通过该属性可指定 MyBatis 的执行器, MyBatis 的执行器总共有三种:
- `ExecutorType.SIMPLE`: 该执行器类型不做特殊的事情, 为每个语句的执行创建一个新的预处理语句 ( `PreparedStatement` )
- `ExecutorType.REUSE`: 该执行器类型会复用预处理语句 ( `PreparedStatement` )
- `ExecutorType.BATCH`: 该执行器类型会批量执行所有的更新语句

## configurationProperties

- 类型: `Properties`
- 默认值: `null` 指定外部化 MyBatis Properties 配置, 通过该配置可以抽离配置, 实现不同环境的配置部署

## configuration

- 类型: `Configuration`
- 默认值: `null` 原生 MyBatis 所支持的配置, 具体请查看 [Configuration](#)

## globalConfig

- 类型: `GlobalConfig`
- 默认值: `null` MyBatis-Plus 全局策略配置, 具体请查看 [GlobalConfig](#)

# Configuration

本部分 (Configuration) 的配置大都为 MyBatis 原生支持的配置, 这意味着您可以通过 MyBatis XML 配置文件的形式进行配置。

## mapUnderscoreToCamelCase

- 类型: `boolean`
- 默认值: `true` 是否开启自动驼峰命名规则 (camel case) 映射, 即从经典数据库列名 `A_COLUMN` (下划线命名) 到经典 Java 属性名 `aColumn` (驼峰命名) 的类似映射。

注意

此属性在 MyBatis 中原默认值为 `false`, 在 MyBatis-Plus 中, 此属性也将用于生成最终的 SQL 的 `select body`

如果您的数据库命名符合规则无需使用 `@TableField` 注解指定数据库字段名

## defaultEnumTypeHandler

- 类型: `Class<? extends TypeHandler>`
- 默认值: `org.apache.ibatis.type.EnumTypeHandler` 默认枚举处理类, 如果配置了该属性, 枚举将统一使用指定处理器进行处理
- `org.apache.ibatis.type.EnumTypeHandler` : 存储枚举的名称
- `org.apache.ibatis.type.EnumOrdinalTypeHandler` : 存储枚举的索引
- `com.baomidou.mybatisplus.extension.handlers.MybatisEnumTypeHandler` : 枚举类需要实现 `IEnum` 接口或字段标记 `@EnumValue` 注解. (3.1.2以下版本为 `EnumTypeHandler`)
- `com.baomidou.mybatisplus.extension.handlers.EnumAnnotationTypeHandler` : 枚举类字段需要标记 `@EnumValue` 注解

## aggressiveLazyLoading

- 类型: `boolean`
- 默认值: `true` 当设置为 `true` 的时候, 懒加载的对象可能被任何懒属性全部加载, 否则, 每个属性都按需加载。需要和 `lazyLoadingEnabled` 一起使用。

## autoMappingBehavior

- 类型: `AutoMappingBehavior`

- 默认值: `partial` MyBatis 自动映射策略, 通过该配置可指定 MyBatis 是否并且如何来自自动映射数据表字段与对象的属性, 总共有 3 种可选值:
- `AutoMappingBehavior.NONE`: 不启用自动映射
- `AutoMappingBehavior.PARTIAL`: 只对非嵌套的 `resultMap` 进行自动映射
- `AutoMappingBehavior.FULL`: 对所有的 `resultMap` 都进行自动映射

## autoMappingUnknownColumnBehavior

- 类型: `AutoMappingUnknownColumnBehavior`
- 默认值: `NONE` MyBatis 自动映射时未知列或未知属性处理策略, 通过该配置可指定 MyBatis 在自动映射过程中遇到未知列或者未知属性时如何处理, 总共有 3 种可选值:
- `AutoMappingUnknownColumnBehavior.NONE`: 不做任何处理 (默认值)
- `AutoMappingUnknownColumnBehavior.WARNING`: 以日志的形式打印相关警告信息
- `AutoMappingUnknownColumnBehavior.FAILING`: 当作映射失败处理, 并抛出异常和详细信息

## cacheEnabled

- 类型: `boolean`
- 默认值: `true` 全局地开启或关闭配置文件中的所有映射器已经配置的任何缓存, 默认为 `true`。

## callSettersOnNulls

- 类型: `boolean`
- 默认值: `false` 指定当结果集中值为 `null` 的时候是否调用映射对象的 `Setter` (`Map` 对象时为 `put`) 方法, 通常运用于有 `Map.keySet()` 依赖或 `null` 值初始化的情况。

通俗的讲, 即 MyBatis 在使用 `resultMap` 来映射查询结果中的列, 如果查询结果中包含空值的列, 则 MyBatis 在映射的时候, 不会映射这个字段, 这就导致在调用到该字段的时候由于没有映射, 取不到而报空指针异常。

当您遇到类似的情况, 请针对该属性进行相关配置以解决以上问题。

基本类型 (`int`、`boolean` 等) 是不能设置成 `null` 的。

## configurationFactory

- 类型: `Class<?>`

- 默认值：`null` 指定一个提供 `Configuration` 实例的工厂类。该工厂生产的实例将用来加载已经被反序列化对象的懒加载属性值，其必须包含一个签名方法 `static Configuration getConfiguration()`。（从 3.2.3 版本开始）

# GlobalConfig

---

## banner

- 类型: `boolean`
- 默认值: `true` 是否控制台 print mybatis-plus 的 LOGO

## sqlParserCache(Deprecated 3.1.1, 直接开启缓存)

- 类型: `boolean`
- 默认值: `false` 是否缓存 Sql 解析, 默认不缓存

## workerId

- 类型: `Long`
- 默认值: `null` 机器 ID 部分(影响雪花ID)

## datacenterId

- 类型: `Long`
- 默认值: `null` 数据标识 ID 部分(影响雪花ID)(workerId 和 datacenterId 一起配置才能重新初始化 Sequence)

## enableSqlRunner

- 类型: `boolean`
- 默认值: `false` 是否初始化  
SqlRunner(`com.baomidou.mybatisplus.extension.toolkit.SqlRunner`)

## sqlInjector

- 类型: `com.baomidou.mybatisplus.core.injector.ISqlInjector`
- 默认值: `com.baomidou.mybatisplus.core.injector.DefaultSqlInjector` SQL注入器  
(starter 下支持 `@bean` 注入)

## superMapperClass

- 类型: `Class`
- 默认值: `com.baomidou.mybatisplus.core.mapper.Mapper.class` 通用Mapper父类(影响 sqlInjector, 只有这个的子类的 mapper 才会注入 sqlInjector 内的 method)

## metaObjectHandler

- 类型: `com.baomidou.mybatisplus.core.handlers.MetaObjectHandler`
- 默认值: `null` 元对象字段填充控制器(starter 下支持 `@bean` 注入)

## idGenerator(since 3.2.1)

- 类型: `com.baomidou.mybatisplus.core.incrementer.IdGenerator`
- 默认值: `null` Id生成器(starter 下支持 `@bean` 注入)

## dbConfig

- 类型: `com.baomidou.mybatisplus.annotation.DbConfig`
- 默认值: `null` MyBatis-Plus 全局策略中的 DB 策略配置, 具体请查看 [DbConfig](#)

# DbConfig

## dbType(Deprecated 3.1.1, 这个属性没什么用)

- 类型: `com.baomidou.mybatisplus.annotation.DbType`
- 默认值: `OTHER` 数据库类型, 默认值为 `未知的数据库类型` 如果值为 `OTHER`, 启动时会根据数据库连接 url 获取数据库类型; 如果不是 `OTHER` 则不会自动获取数据库类型

## idType

- 类型: `com.baomidou.mybatisplus.annotation.IdType`
- 默认值: `ID_WORKER` 全局默认主键类型

## tablePrefix

- 类型: `String`
- 默认值: `null` 表名前缀

## schema(since 3.1.1)

- 类型: `String`
- 默认值: `null` schema

## columnFormat(since 3.1.1)

- 类型: `String`
- 默认值: `null` 字段 format(since 3.1.1), 例: `%s`, (对主键无效)

## tableUnderline

- 类型: `boolean`
- 默认值: `true` 表名、是否使用下划线命名, 默认数据库表使用下划线命名

## columnLike(Deprecated 3.1.1)

- 类型: `boolean`
- 默认值: `false` 是否开启 LIKE 查询, 即根据 entity 自动生成的 where 条件中 String 类型字段 是否使用 LIKE, 默认不开启

## capitalMode



- 类型: `boolean`
- 默认值: `false` 是否开启大写命名, 默认不开启

## keyGenerator

- 类型: `com.baomidou.mybatisplus.core.incrementer.IKeyGenerator`
- 默认值: `null` 表主键生成器(starter 下支持 `@bean` 注入)

## logicDeleteValue

- 类型: `String`
- 默认值: `1` 逻辑已删除值, (`逻辑删除` 下有效)

## logicNotDeleteValue

- 类型: `String`
- 默认值: `0` 逻辑未删除值, (`逻辑删除` 下有效)

## fieldStrategy(Deprecated 3.1.2, 将用下面三个新的取代)

- 类型: `com.baomidou.mybatisplus.annotation.FieldStrategy`
- 默认值: `NOT_NULL` 字段验证策略

说明:

该策略约定了如何产出注入的sql, 涉及 `insert` , `update` 以及 `wrapper` 内部的 `entity` 属性生成的 `where` 条件

## insertStrategy(since 3.1.2)

- 类型: `com.baomidou.mybatisplus.annotation.FieldStrategy`
- 默认值: `NOT_NULL` 字段验证策略之 `insert`

说明:

在 `insert` 的时候的字段验证策略目前没有默认值, 等 `{@link #fieldStrategy}` 完全去除掉, 会给个默认值 `NOT_NULL` 没配则按 `{@link #fieldStrategy}` 为准

## updateStrategy(since 3.1.2)

- 类型: `com.baomidou.mybatisplus.annotation.FieldStrategy`
- 默认值: `NOT_NULL` 字段验证策略之 `update`

说明：

在 update 的时候的字段验证策略目前没有默认值,等 {@link #fieldStrategy} 完全去除掉,会给个默认值 NOT\_NULL没配则按 {@link #fieldStrategy} 为准

## selectStrategy(since 3.1.2)

- 类型: `com.baomidou.mybatisplus.annotation.FieldStrategy`
- 默认值: `NOT_NULL` 字段验证策略之 select

说明：

在 select 的时候的字段验证策略: wrapper 根据内部 entity 生成的 where 条件目前没有默认值,等 {@link #fieldStrategy} 完全去除掉,会给个默认值 NOT\_NULL没配则按 {@link #fieldStrategy} 为准

# 代码生成器配置

---

- [基本配置](#)
- [数据源 dataSourceConfig 配置](#)
- [数据库表配置](#)
- [包名配置](#)
- [模板配置](#)
- [全局策略 globalConfig 配置](#)
- [注入 injectionConfig 配置](#)

# 基本配置

---

## dataSource

- 类型: `DataSourceConfig`
- 默认值: `null` 数据源配置, 通过该配置, 指定需要生成代码的具体数据库, 具体请查看 [数据源配置](#)

## strategy

- 类型: `StrategyConfig`
- 默认值: `null` 数据库表配置, 通过该配置, 可指定需要生成哪些表或者排除哪些表, 具体请查看 [数据库表配置](#)

## packageInfo

- 类型: `PackageConfig`
- 默认值: `null` 包名配置, 通过该配置, 指定生成代码的包路径, 具体请查看 [包名配置](#)

## template

- 类型: `TemplateConfig`
- 默认值: `null` 模板配置, 可自定义代码生成的模板, 实现个性化操作, 具体请查看 [模板配置](#)

## globalConfig

- 类型: `GlobalConfig`
- 默认值: `null` 全局策略配置, 具体请查看 [全局策略配置](#)

## injectionConfig

- 类型: `InjectionConfig`
- 默认值: `null` 注入配置, 通过该配置, 可注入自定义参数等操作以实现个性化操作, 具体请查看 [注入配置](#)

# 数据源 dataSourceConfig 配置

---

## dbQuery

- 数据库信息查询类
- 默认由 `dbType` 类型决定选择对应数据库内置实现实现 `IDbQuery` 接口自定义数据库查询  
`SQL 语句` 定制化返回自己需要的内容

## dbType

- 数据库类型
- 该类内置了常用的数据库类型【必须】

## schemaName

- 数据库 schema name
- 例如 `PostgreSQL` 可指定为 `public`

## typeConvert

- 类型转换
- 默认由 `dbType` 类型决定选择对应数据库内置实现实现 `ITypeConvert` 接口自定义数据库  
`字段类型` 转换为自己需要的 `java` 类型，内置转换类型无法满足可实现 `IColumnType`  
接口自定义

## url

- 驱动连接的URL

## driverName

- 驱动名称

## username

- 数据库连接用户名

## password

- 数据库连接密码

## 数据库表配置

---

### isCapitalMode

是否大写命名

### skipView

是否跳过视图

### naming

数据库表映射到实体的命名策略

### columnNaming

数据库表字段映射到实体的命名策略，未指定按照 `naming` 执行

### tablePrefix

表前缀

### fieldPrefix

字段前缀

### superEntityClass

自定义继承的Entity类全称，带包名

### superEntityColumns

自定义基础的Entity类，公共字段

### superMapperClass

自定义继承的Mapper类全称，带包名

### superServiceClass

自定义继承的Service类全称，带包名

## superServiceImplClass

自定义继承的ServiceImpl类全称，带包名

## superControllerClass

自定义继承的Controller类全称，带包名

## include

需要包含的表名，允许正则表达式（与exclude二选一配置）

## exclude

需要排除的表名，允许正则表达式

## entityColumnConstant

【实体】是否生成字段常量（默认 false）

## entityBuilderModel

【实体】是否为构建者模型（默认 false）

## entityLombokModel

【实体】是否为lombok模型（默认 false）

## entityBooleanColumnRemoveIsPrefix

Boolean类型字段是否移除is前缀（默认 false）

## restControllerStyle

生成 `@RestController` 控制器

## controllerMappingHyphenStyle

驼峰转连字符

## entityTableFieldAnnotationEnable

是否生成实体时，生成字段注解

versionFieldName

乐观锁属性名称

logicDeleteFieldName

逻辑删除属性名称

tableFillList

表填充字段



## 包名配置

---

### parent

父包名。如果为空，将下面子包名必须写全部， 否则就只需写子包名

### moduleName

父包模块名

### entity

Entity包名

### service

Service包名

### serviceImpl

Service Impl包名

### mapper

Mapper包名

### xml

Mapper XML包名

### controller

Controller包名

### pathInfo

路径配置信息

## 模板配置

---

### entity

Java 实体类模板

### entityKt

Kotlin 实体类模板

### service

Service 类模板

### serviceImpl

Service impl 实现类模板

### mapper

mapper 模板

### xml

mapper xml 模板

### controller

controller 控制器模板

## 全局策略 globalConfig 配置

---

### outputDir

- 生成文件的输出目录
- 默认值: `D 盘根目录`

### fileOverride

- 是否覆盖已有文件
- 默认值: `false`

### open

- 是否打开输出目录
- 默认值: `true`

### enableCache

- 是否在xml中添加二级缓存配置
- 默认值: `false`

### author

- 开发人员
- 默认值: `null`

### kotlin

- 开启 Kotlin 模式
- 默认值: `false`

### swagger2

- 开启 swagger2 模式
- 默认值: `false`

### activeRecord

- 开启 ActiveRecord 模式

- 默认值: `false`

## baseResultMap

- 开启 BaseResultMap
- 默认值: `false`

## baseColumnList

- 开启 baseColumnList
- 默认值: `false`

## dateType

- 时间类型对应策略
- 默认值: `TIME_PACK`

注意事项:

如下配置 `%s` 为占位符

## entityName

- 实体命名方式
- 默认值: `null` 例如: `%sEntity` 生成 `UserEntity`

## mapperName

- mapper 命名方式
- 默认值: `null` 例如: `%sDao` 生成 `UserDao`

## xmlName

- Mapper xml 命名方式
- 默认值: `null` 例如: `%sDao` 生成 `UserDao.xml`

## serviceName

- service 命名方式
- 默认值: `null` 例如: `%sBusiness` 生成 `UserBusiness`

## serviceImplName

- service impl 命名方式
- 默认值: `null` 例如: `%sBusinessImpl` 生成 `UserBusinessImpl`

## controllerName

- controller 命名方式
- 默认值: `null` 例如: `%sAction` 生成 `UserAction`

## idType

- 指定生成的主键的ID类型
- 默认值: `null`

## 注入 injectionConfig 配置

---

### map

- 自定义返回配置 Map 对象
- 该对象可以传递到模板引擎通过 `cfg.xxx` 引用

### fileOutConfigList

- 自定义输出文件
- 配置 `FileOutConfig` 指定模板文件、输出文件达到自定义文件生成目的

### fileCreate

- 自定义判断是否创建文件
- 实现 `IFileCreate` 接口该配置用于判断某个类是否需要覆盖创建，当然你可以自己实现差异算法 `merge` 文件

### initMap

- 注入自定义 Map 对象(注意需要setMap放进去)