

MyBatis-Plus v3.3.2 使用教程



MyBatis-Plus，Mybatis 增强工具包 - 只做增强不做改变，简化CRUD操作。



下载手机APP
畅享精彩阅读

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

AbstractWrapper

QueryWrapper

UpdateWrapper

使用 Wrapper 自定义SQL

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

数据安全保护

多数据源

多租户 SQL 解析器

动态表名 SQL 解析器

MybatisX 快速开发插件

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 | |
| FAQ | |
| 常见问题 | |
| 捐赠支持 | |
| 更新日志 | |

致谢

当前文档《MyBatis-Plus v3.3.2 使用教程》由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2020-05-26。

书栈网仅提供文档编写、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，书栈网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，欢迎分享到书栈网，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，请到书栈网获取最新的文档，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-3.3.2>

书栈官网：<https://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

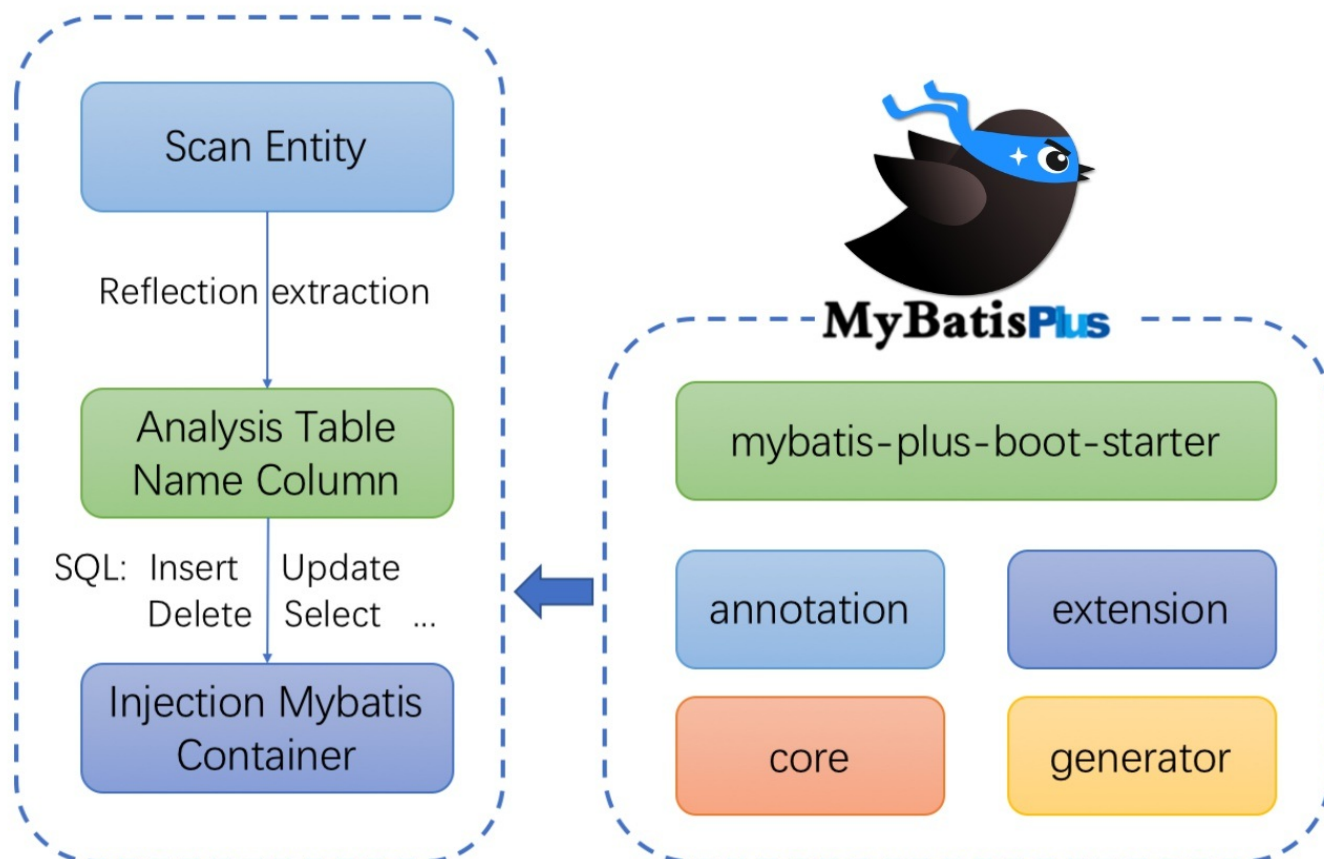
- 无侵入: 只做增强不做改变, 引入它不会对现有工程产生影响, 如丝般顺滑
- 损耗小: 启动即会自动注入基本 CURD, 性能基本无损耗, 直接面向对象操作

- 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



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

- [renren-security](#)  : 人人权限系统
- [freeter-admin](#)  : 飞特后台管理系统
- [vblog](#)  : VBlog 博客系统
- [jiiiiiin-security](#)  : jiiiiiin权限系统
- [hdw-dubbo](#)  : HDW快速开发平台
- [pybbs](#)  : 更好用的Java语言社区(论坛)
- [SmallBun](#)  : SmallBun企业级开发脚手架
- [webplus](#)  : 综合开发平台
- [x-boot](#)  : VUE 前后端分离开发平台
- [nice-blog-sys](#)  : 基于SpirngBoot开发, 好看的个人博客
- [Diboot](#)  : 轻代码开发平台
- [tyboot](#)  : 基于SpringBoot的快速开发脚手架
- [ac-blog](#)  : ac博客网站
- [spider-flow](#)  : 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
- [goodskill](#)  : 基于Dubbo + SpringBoot搭建的秒杀系统
- [SpringBoot_MyBatisPlus](#)  : SpringBoot集成MyBatisPlus 
- [bootplus](#)  : 基于 `SpringBoot + Shiro + MyBatisPlus` 的权限管理框架
- [Dice](#)  : 一个Vue 2.x 和 SpringBoot 全家桶开发的前后端分离的个人内容管理系统: 「博客」、「权限管理」、「代码段」、「媒体库」等。
- [thyme-boot](#)  : 基于SpringBoot+Layui+Vue的快速后台开发框架
- [zuihou-admin-cloud](#)  : 基于SpringCloud的SaaS微服务脚手架

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
- 上海乔融金融信息服务有限公司
- 苏州墨焱网络科技有限公司
- 深兰科技(上海)有限公司
- 济南悦码信息科技有限公司
- 北京魔力耳朵科技有限公司
- 宋城独木桥网络有限公司
- 重庆景辉昱阳科技有限公司
- 厦门亲禾教育科技有限公司
- 济南申宝网络科技有限公司
- 奇安信科技集团股份有限公司
- 苏州时新集成技术有限公司
- 中科传媒科技有限责任公司

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=Jones, 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](#)[↗] | [Spring MVC 快速启动示例](#)[↗]

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：

```
1. compile group: 'com.baomidou', name: 'mybatis-plus-boot-starter', version: '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(Application.class, args);
7.     }
8.
9. }
```

- Spring MVC 工程：
 - 配置 MapperScan

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

- 调整 SqlSessionFactory 为 MyBatis-Plus 的 SqlSessionFactory

```
<bean id="sqlSessionFactory"
1. class="com.baomidou.mybatisplus.extension.spring.MybatisSqlSessionFactoryBean">
2.     <property name="dataSource" ref="dataSource"/>
3. </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](#)[↗]、[Spring MVC 快速启动示例](#)[↗]。

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 |
| 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) |
| 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) |

关于`autoResultMap`的说明：

mp会自动构建一个 `ResultMap` 并注入到mybatis里(一般用不上).下面讲两句：因为mp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到mybatis里 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。

@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ID自增 |
| NONE | 无状态, 该类型为未设置主键类型(注解里等于跟随全局, 全局里约等于 INPUT) |
| INPUT | insert前自行set主键值 |
| ASSIGN_ID | 分配ID(主键类型为Number(Long和Integer)或String)(since 3.3.0), 使用接口 <code>IdentifierGenerator</code> 的方法 <code>nextId</code> (默认实现类为 <code>DefaultIdentifierGenerator</code> 雪花算法) |
| ASSIGN_UUID | 分配UUID, 主键类型为String(since 3.3.0), 使用接口 <code>IdentifierGenerator</code> 的方法 <code>nextUUID</code> (默认default方法) |
| ID_WORKER | 分布式全局唯一ID 长整型类型(please use <code>ASSIGN_ID</code>) |
| UUID | 32位UUID字符串(please use <code>ASSIGN_UUID</code>) |
| ID_WORKER_STR | 分布式全局唯一ID 字符串类型(please use <code>ASSIGN_ID</code>) |

@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 | "" | 数据库字段名 |
| el | String | 否 | "" | 映射为原生 <code>#{ ... }</code> 逻辑, 相当于写在 xml 里的 <code><el></code> 部分 |
| exist | boolean | 否 | true | 是否为数据库表字段 |
| condition | String | 否 | "" | 字段 <code>where</code> 实体查询条件, 有值设置则按设置的为准, 没有则为默认全局的 <code>%s=#{%s}</code> , 参考 这里 |
| update | String | 否 | "" | 字段 <code>update set</code> 部分插入, 例如: <code>update="%s+1"</code> : 表示插入时会set <code>version=version+1</code> (优先级高于 <code>el</code> 属性) |
| insertStrategy | Enum | N | DEFAULT | 举例: NOT NULL: <pre>insert into table a(<if test="columnProperty != null">column</if> value<if test="columnProperty != null">#{columnProperty}</if>)</pre> |
| updateStrategy | Enum | N | DEFAULT | 举例: IGNORED: <pre>update table a set column=#{columnProperty}</pre> |
| | | | | 举例: NOT EMPTY: <pre><if test="columnProperty != null"></pre> |

| | | | | |
|------------------|---|---|---------------------------------------|---|
| whereStrategy | Enum | N | DEFAULT | <code>null and columnProperty!=''">columnProperty}</if></code> |
| 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 fo 进行处理 |
| jdbcType | JdbcType | 否 | JdbcType.UNDEFINED | JDBC类型（该默认值不会按照该值生效） |
| typeHandler | <code>Class<? extends TypeHandler></code> | 否 | <code>UnknownTypeHandler.class</code> | 类型处理器（该默认值不会按照该值生效） |
| numericScale | String | 否 | "" | 指定小数点后保留的位数 |

关于`jdbcType`和`typeHandler`以及`numericScale`的说明：

`numericScale` 只生效于 update 的sql. `jdbcType` 和 `typeHandler` 如果不配合 `@TableName#autoResultMap = true` 一起使用,也只生效于 update 的sql. 对于 `typeHandler` 如果你的字段类型和set进去的类型为 `equals` 关系,则只需要让你的 `typeHandler` 让Mybatis加载到即可,不需要使用注解

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 `@Verison` 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, 支持method上以及mapper接口上

| 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](#)

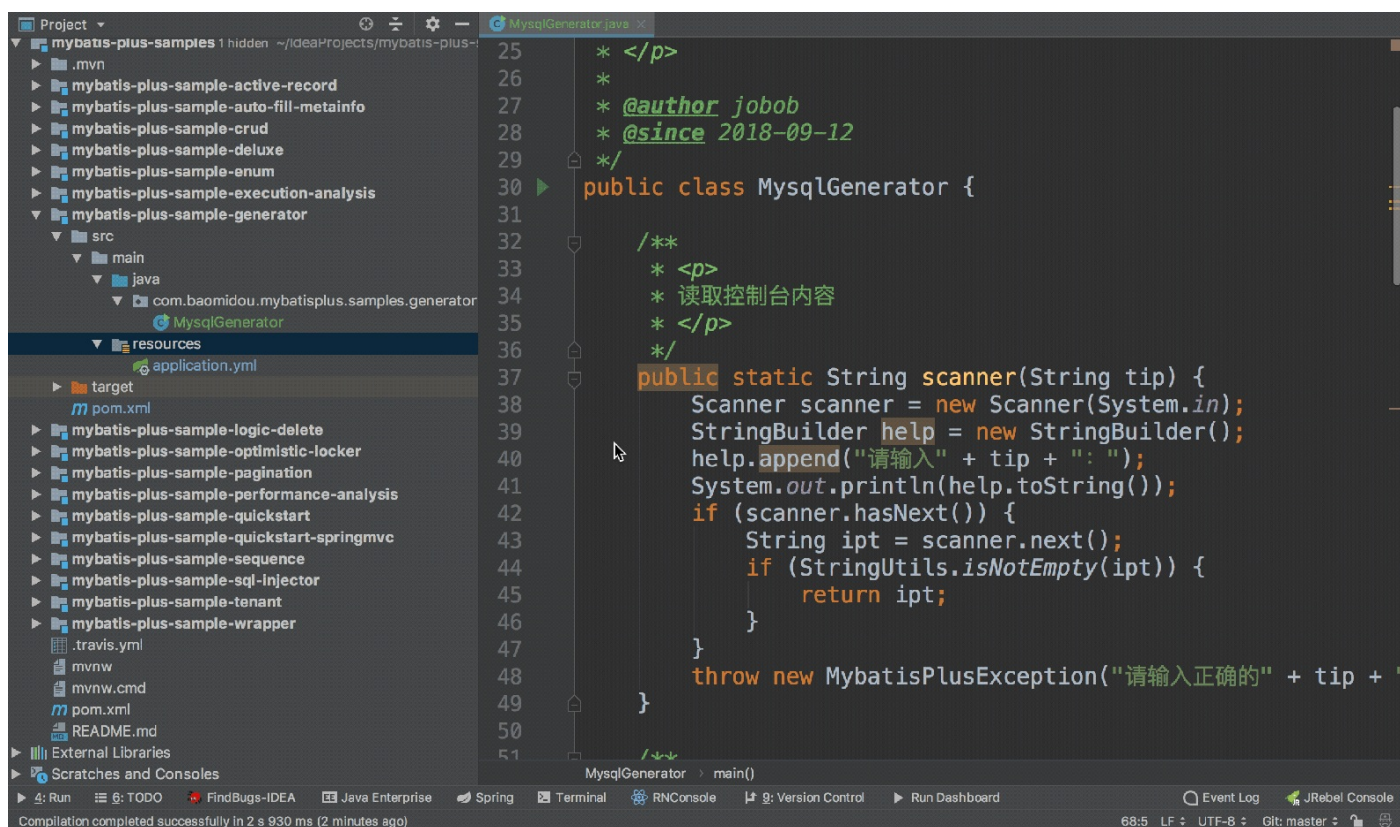
代码生成器

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.isEmpty(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?useUnicode=true&useSSL=false&characterEncoding=utf8");
39.     // dsc.setSchemaName("public");
40.     dsc.setDriverName("com.mysql.jdbc.Driver");
41.     dsc.setUsername("root");
42.     dsc.setPassword("密码");
43.     mpg.setDataSource(dsc);
44.
45.     // 包配置
46.     PackageConfig pc = new PackageConfig();
47.     pc.setModuleName(scanner("模块名"));
48.     pc.setParent("com.baomidou.ant");
49.     mpg.setPackageInfo(pc);
50.
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 的名称会跟着发生变化!!
71.             return projectPath + "/src/main/resources/mapper/" + pc.getModuleName()
72.                 + "/" + tableInfo.getEntityName() + "Mapper" + StringPool.DOT_XML;
73.         }
74.     });
75.     /*
76.     cfg.setFileCreate(new IFileCreate() {
77.         @Override
78.         public boolean isCreate(ConfigBuilder configBuilder, FileType fileType, String filePath) {
79.             // 判断自定义文件夹是否需要创建
80.             checkDir("调用默认方法创建的目录, 自定义目录用");
81.             if (fileType == FileType.MAPPER) {
82.                 // 已经生成 mapper 文件判断存在, 不想重新生成返回 false
83.                 return !new File(filePath).exists();
84.             }
85.             // 允许生成模板文件
86.             return true;
87.         }
88.     });
89.     */
90.     cfg.setFileOutConfigList(focList);
91.     mpg.setCfg(cfg);
92.
93.     // 配置模板
94.     TemplateConfig templateConfig = new TemplateConfig();
95.
96.     // 配置自定义输出模板
97.     // 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
98.     // templateConfig.setEntity("templates/entity2.java");
99.     // templateConfig.setService();
100.    // templateConfig.setController();
101.
102.    templateConfig.setXml(null);
103.    mpg.setTemplate(templateConfig);
104.
105.    // 策略配置
106.    StrategyConfig strategy = new StrategyConfig();
107.    strategy.setNaming(NamingStrategy.underline_to_camel);
108.    strategy.setColumnNaming(NamingStrategy.underline_to_camel);
109.    strategy.setSuperEntityClass("你自己的父类实体,没有就不用设置!");
110.    strategy.setEntityLombokModel(true);
111.    strategy.setRestControllerStyle(true);
112.    // 公共父类
113.    strategy.setSuperControllerClass("你自己的父类控制器,没有就不用设置!");
114.    // 写于父类中的公共字段
115.    strategy.setSuperEntityColumns("id");
116.    strategy.setInclude(scanner("表名, 多个英文逗号分割").split(","));
117.    strategy.setControllerMappingHyphenStyle(true);
118.    strategy.setTablePrefix(pc.getModuleName() + "_");
119.    mpg.setStrategy(strategy);
120.    mpg.setTemplateEngine(new FreemarkerTemplateEngine());
121.    mpg.execute();

```

```
122.     }  
123.  
124. }
```

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
  f convert = false
  f keyFlag = true
  f keyIdentityFlag = true
  f name = "id"
  f type = "int(11)"
  f propertyName = "id"
  f columnType = {DbColumnType@2135} "INTEGER"
  f comment = ""
  f fill = null
  f 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<br>
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 PRIVILEGES
8.     */
9.    @Override
10.   public String[] fieldCustom() {
11.       return new String[]{"NULL", "PRIVILEGES"};
12.   }
13. })

```


CRUD 接口

Service CRUD 接口

说明：

- 通用 Service CRUD 封装 `IService` 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. // 插入一条记录（选择字段，策略插入）
2. boolean save(T entity);
3. // 插入（批量）
4. boolean saveBatch(Collection<T> entityList);
5. // 插入（批量）
6. boolean saveBatch(Collection<T> entityList, int batchSize);
```

参数说明

| 类型 | 参数名 | 描述 |
|---------------|------------|--------|
| T | entity | 实体对象 |
| Collection<T> | entityList | 实体对象集合 |
| int | batchSize | 插入批次数量 |

SaveOrUpdate

```
1. // TableId 注解存在更新记录，否插入一条记录
2. boolean saveOrUpdate(T entity);
3. // 根据updateWrapper尝试更新，否继续执行saveOrUpdate(T)方法
4. boolean saveOrUpdate(T entity, Wrapper<T> updateWrapper);
5. // 批量修改插入
6. boolean saveOrUpdateBatch(Collection<T> entityList);
7. // 批量修改插入
8. boolean saveOrUpdateBatch(Collection<T> entityList, int batchSize);
```

参数说明

| 类型 | 参数名 | 描述 |
|----|-----|----|
| | | |

| | | |
|---------------|---------------|-------------------------|
| T | entity | 实体对象 |
| Wrapper<T> | updateWrapper | 实体对象封装操作类 UpdateWrapper |
| Collection<T> | entityList | 实体对象集合 |
| int | batchSize | 插入批次数量 |

Remove

```
1. // 根据 entity 条件, 删除记录
2. boolean remove(Wrapper<T> queryWrapper);
3. // 根据 ID 删除
4. boolean removeById(Serializable id);
5. // 根据 columnMap 条件, 删除记录
6. boolean removeByMap(Map<String, Object> columnMap);
7. // 删除 (根据ID 批量删除)
8. boolean removeByIds(Collection<? extends Serializable> idList);
```

参数说明

| 类型 | 参数名 | 描述 |
|------------------------------------|--------------|--------------------|
| Wrapper<T> | queryWrapper | 实体包装类 QueryWrapper |
| Serializable | id | 主键ID |
| Map<String, Object> | columnMap | 表字段 map 对象 |
| Collection<? extends Serializable> | idList | 主键ID列表 |

Update

```
1. // 根据 UpdateWrapper 条件, 更新记录 需要设置sqlset
2. boolean update(Wrapper<T> updateWrapper);
3. // 根据 whereEntity 条件, 更新记录
4. boolean update(T entity, Wrapper<T> updateWrapper);
5. // 根据 ID 选择修改
6. boolean updateById(T entity);
7. // 根据ID 批量更新
8. boolean updateBatchById(Collection<T> entityList);
9. // 根据ID 批量更新
10. boolean updateBatchById(Collection<T> entityList, int batchSize);
```

参数说明

| 类型 | 参数名 | 描述 |
|---------------|---------------|-------------------------|
| Wrapper<T> | updateWrapper | 实体对象封装操作类 UpdateWrapper |
| T | entity | 实体对象 |
| Collection<T> | entityList | 实体对象集合 |
| int | batchSize | 更新批次数量 |

Get

```
1. // 根据 ID 查询
2. T getById(Serializable id);
3. // 根据 Wrapper, 查询一条记录。结果集, 如果是多个会抛出异常, 随机取一条加上限制条件 wrapper.last("LIMIT 1")
4. T getOne(Wrapper<T> queryWrapper);
5. // 根据 Wrapper, 查询一条记录
6. T getOne(Wrapper<T> queryWrapper, boolean throwEx);
7. // 根据 Wrapper, 查询一条记录
8. Map<String, Object> getMap(Wrapper<T> queryWrapper);
9. // 根据 Wrapper, 查询一条记录
10. <V> V getObj(Wrapper<T> queryWrapper, Function<? super Object, V> mapper);
```

参数说明

| 类型 | 参数名 | 描述 |
|-----------------------------|--------------|------------------------|
| Serializable | id | 主键ID |
| Wrapper<T> | queryWrapper | 实体对象封装操作类 QueryWrapper |
| boolean | throwEx | 有多个 result 是否抛出异常 |
| T | entity | 实体对象 |
| Function<? super Object, V> | mapper | 转换函数 |

List

```
1. // 查询所有
2. List<T> list();
3. // 查询列表
4. List<T> list(Wrapper<T> queryWrapper);
5. // 查询 (根据ID 批量查询)
6. Collection<T> listByIds(Collection<? extends Serializable> idList);
7. // 查询 (根据 columnMap 条件)
8. Collection<T> listByMap(Map<String, Object> columnMap);
9. // 查询所有列表
10. List<Map<String, Object>> listMaps();
11. // 查询列表
12. List<Map<String, Object>> listMaps(Wrapper<T> queryWrapper);
13. // 查询全部记录
14. List<Object> listObjs();
15. // 查询全部记录
16. <V> List<V> listObjs(Function<? super Object, V> mapper);
17. // 根据 Wrapper 条件, 查询全部记录
18. List<Object> listObjs(Wrapper<T> queryWrapper);
19. // 根据 Wrapper 条件, 查询全部记录
20. <V> List<V> listObjs(Wrapper<T> queryWrapper, Function<? super Object, V> mapper);
```

参数说明

| 类型 | 参数名 | 描述 |
|----|-----|----|
|----|-----|----|

| | | |
|------------------------------------|--------------|------------------------|
| Wrapper<T> | queryWrapper | 实体对象封装操作类 QueryWrapper |
| Collection<? extends Serializable> | idList | 主键ID列表 |
| Map<?String, Object> | columnMap | 表字段 map 对象 |
| Function<? super Object, V> | mapper | 转换函数 |

Page

```
1. // 无条件翻页查询
2. IPage<T> page(IPage<T> page);
3. // 翻页查询
4. IPage<T> page(IPage<T> page, Wrapper<T> queryWrapper);
5. // 无条件翻页查询
6. IPage<Map<String, Object>> pageMaps(IPage<T> page);
7. // 翻页查询
8. IPage<Map<String, Object>> pageMaps(IPage<T> page, Wrapper<T> queryWrapper);
```

参数说明

| 类型 | 参数名 | 描述 |
|------------|--------------|------------------------|
| IPage<T> | page | 翻页对象 |
| Wrapper<T> | queryWrapper | 实体对象封装操作类 QueryWrapper |

Count

```
1. // 查询总记录数
2. int count();
3. // 根据 wrapper 条件, 查询总记录数
4. int count(Wrapper<T> queryWrapper);
```

参数说明

| 类型 | 参数名 | 描述 |
|------------|--------------|------------------------|
| Wrapper<T> | queryWrapper | 实体对象封装操作类 QueryWrapper |

Chain

query

```
1. // 链式查询 普通
2. QueryChainWrapper<T> query();
3. // 链式查询 lambda 式。注意：不支持 Kotlin
4. LambdaQueryChainWrapper<T> lambdaQuery();
5.
6. // 示例：
7. query().eq("column", value).one();
```

```
8. lambdaQuery().eq(Entity::getId, value).list();
```

update

```
1. // 链式更改 普通
2. UpdateChainWrapper<T> update();
3. // 链式更改 lambda 式。注意：不支持 Kotlin
4. LambdaUpdateChainWrapper<T> lambdaUpdate();
5.
6. // 示例：
7. update().eq("column", value).remove();
8. lambdaUpdate().eq(Entity::getId, value).update(entity);
```

Mapper CRUD 接口

说明：

- 通用 CRUD 封装 `BaseMapper`  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. // 插入一条记录
2. int insert(T entity);
```

参数说明

| 类型 | 参数名 | 描述 |
|----|--------|------|
| T | entity | 实体对象 |

Delete

```
1. // 根据 entity 条件，删除记录
2. int delete(@Param(Constants.WRAPPER) Wrapper<T> wrapper);
3. // 删除（根据ID 批量删除）
4. int deleteBatchIds(@Param(Constants.COLLECTION) Collection<? extends Serializable> idList);
5. // 根据 ID 删除
6. int deleteById(Serializable id);
7. // 根据 columnMap 条件，删除记录
8. int deleteByMap(@Param(Constants.COLUMN_MAP) Map<String, Object> columnMap);
```

参数说明

| 类型 | 参数名 | 描述 |
|------------------------------------|-----------|---------------------------|
| Wrapper<T> | wrapper | 实体对象封装操作类（可以为 null） |
| Collection<? extends Serializable> | idList | 主键ID列表(不能为 null 以及 empty) |
| Serializable | id | 主键ID |
| Map<String, Object> | columnMap | 表字段 map 对象 |

Update

```

1. // 根据 whereEntity 条件, 更新记录
2. int update(@Param(Constants.ENTITY) T entity, @Param(Constants.WRAPPER) Wrapper<T> updateWrapper);
3. // 根据 ID 修改
4. int updateById(@Param(Constants.ENTITY) T entity);

```

参数说明

| 类型 | 参数名 | 描述 |
|------------|---------------|---|
| T | entity | 实体对象（set 条件值, 可为 null） |
| Wrapper<T> | updateWrapper | 实体对象封装操作类（可以为 null, 里面的 entity 用于生成 where 语句） |

Select

```

1. // 根据 ID 查询
2. T selectById(Serializable id);
3. // 根据 entity 条件, 查询一条记录
4. T selectOne(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
5.
6. // 查询（根据ID 批量查询）
7. List<T> selectBatchIds(@Param(Constants.COLLECTION) Collection<? extends Serializable> idList);
8. // 根据 entity 条件, 查询全部记录
9. List<T> selectList(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
10. // 查询（根据 columnMap 条件）
11. List<T> selectByMap(@Param(Constants.COLUMN_MAP) Map<String, Object> columnMap);
12. // 根据 wrapper 条件, 查询全部记录
13. List<Map<String, Object>> selectMaps(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
14. // 根据 wrapper 条件, 查询全部记录。注意：只返回第一个字段的值
15. List<Object> selectObjs(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
16.
17. // 根据 entity 条件, 查询全部记录（并翻页）
18. IPage<T> selectPage(IPage<T> page, @Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
19. // 根据 wrapper 条件, 查询全部记录（并翻页）
20. IPage<Map<String, Object>> selectMapsPage(IPage<T> page, @Param(Constants.WRAPPER) Wrapper<T> queryWrapper);
21. // 根据 wrapper 条件, 查询总记录数
22. Integer selectCount(@Param(Constants.WRAPPER) Wrapper<T> queryWrapper);

```

参数说明

| 类型 | 参数名 | 描述 |
|------------------------------------|--------------|-------------------------------|
| Serializable | id | 主键ID |
| Wrapper<T> | queryWrapper | 实体对象封装操作类（可以为 null） |
| Collection<? extends Serializable> | idList | 主键ID列表（不能为 null 以及 empty） |
| Map<String, Object> | columnMap | 表字段 map 对象 |
| IPage<T> | page | 分页查询条件（可以为 RowBounds.DEFAULT） |

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](#)

使用详细见[源码注释](#)

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

1. wrapper 很重
2. 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)
3. 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
4. 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)
3. allEq(boolean condition, BiPredicate<R, V> filter, Map<R, V> params, boolean 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 NOT 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") --> id 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(Consumer<Param> consumer)
2. or(boolean condition, Consumer<Param> consumer)
```

- 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(Consumer<Param> consumer)
2. and(boolean condition, Consumer<Param> consumer)
```

- 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(Consumer<Param> consumer)
2. nested(boolean condition, Consumer<Param> consumer)
```

- 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` ，以下两种方案取其其一即可

Service.java

```
1. mySqlMapper.getAll(Wrappers.<MySqlData>lambdaQuery().eq(MySqlData::getGroup, 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>
```

kotlin使用wrapper

kotlin 可以使用 `QueryWrapper` 和 `UpdateWrapper` 但无法使用 `LambdaQueryWrapper` 和 `LambdaUpdateWrapper` 如果想使用 lambda 方式的 wrapper 请使用 `KtQueryWrapper` 和 `KtUpdateWrapper`

分页插件

示例工程：

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

```

1. <!-- spring xml 方式 -->
2. <property name="plugins">
3.     <array>
4.         <bean class="com.baomidou.mybatisplus.extension.plugins.PaginationInterceptor">
5.             <property name="sqlParser" ref="自定义解析类、可以没有"/>
6.             <property name="dialectClazz" value="自定义方言类、可以没有"/>
7.             <!-- COUNT SQL 解析.可以没有 -->
8.             <property name="countSqlParser" ref="countSqlParser"/>
9.         </bean>
10.    </array>
11. </property>
12.
13. <bean id="countSqlParser"
14.     class="com.baomidou.mybatisplus.extension.plugins.pagination.optimize.JsqlParserCountOptimize">
15.     <!-- 设置为 true 可以优化部分 left join 的sql -->
16.     <property name="optimizeJoin" value="true"/>
17. </bean>

```

```

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 PaginationInterceptor();
10.        // 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
11.        // paginationInterceptor.setOverflow(false);
12.        // 设置最大单页限制数量, 默认 500 条, -1 不受限制
13.        // paginationInterceptor.setLimit(500);
14.        // 开启 count 的 join 优化,只针对部分 left join
15.        paginationInterceptor.setCountSqlParser(new JsqlParserCountOptimize(true));
16.        return paginationInterceptor;
17.    }
18. }

```

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.      * </p>
6.      *
7.      * @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,必须放在第一位(你可以继承Page实现自己的分页对象)
8.      * @param state 状态
9.      * @return 分页对象
10.     */
11.     IPage<User> selectPageVo(Page<?> page, Integer state);
12. }

```

- 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

```

1. public IPage<User> selectUserPage(Page<User> page, Integer state) {
2.     // 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
3.     // page.setOptimizeCountSql(false);
4.     // 当 total 为小于 0 或者设置 setSearchCount(false) 分页插件不会进行 count 查询
5.     // 要点!! 分页返回的对象与传入的对象是同一个
6.     return userMapper.selectPageVo(page, state);
7. }

```

Sequence主键

主键生成策略必须使用**INPUT**

支持父类定义@KeySequence子类继承使用

支持主键类型指定(3.3.0开始自动识别主键类型)

内置支持：

- DB2KeyGenerator
- H2KeyGenerator
- KingbaseKeyGenerator
- OracleKeyGenerator
- PostgreKeyGenerator

如果内置支持不满足你的需求，可实现IKeyGenerator接口来进行扩展。

举个栗子

```
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. }
```

Spring-Boot

方式一：使用配置类

```
1. @Bean
2. public IKeyGenerator keyGenerator() {
3.     return new H2KeyGenerator();
4. }
```

方式二：通过MybatisPlusPropertiesCustomizer自定义

```
1. @Bean
2. public MybatisPlusPropertiesCustomizer plusPropertiesCustomizer() {
3.     return plusProperties -> plusProperties.getGlobalConfig().getDbConfig().setKeyGenerator(new
4. H2KeyGenerator());
5. }
```

Spring

方式一：XML配置

```
1. <bean id="globalConfig" class="com.baomidou.mybatisplus.core.config.GlobalConfig">
2.     <property name="dbConfig" ref="dbConfig"/>
3. </bean>
4.
5. <bean id="dbConfig" class="com.baomidou.mybatisplus.core.config.GlobalConfig.DbConfig">
6.     <property name="keyGenerator" ref="keyGenerator"/>
7. </bean>
8.
9. <bean id="keyGenerator" class="com.baomidou.mybatisplus.extension.incrementer.H2KeyGenerator"/>
```

方式二：注解配置

```
1. @Bean
2. public GlobalConfig globalConfig() {
3.     GlobalConfig conf = new GlobalConfig();
4.     conf.setDbConfig(new GlobalConfig.DbConfig().setKeyGenerator(new H2KeyGenerator()));
5.     return conf;
6. }
```

自定义ID生成器

自3.3.0开始, 默认使用雪花算法+UUID(不含中划线)

自定义示例工程:

- spring-boot示例 : [传送门](#)

| 方法 | 主键生成策略 | 主键类型 | 说明 |
|----------|-------------------------------------|---------------------|--|
| nextId | ASSIGN_ID, ID_WORKER, ID_WORKER_STR | Long,Integer,String | 支持自动转换为String类型, 但数值类型不支持自动转换, 需精准匹配, 例如返回Long, 实体主键就不支持定义为Integer |
| nextUUID | ASSIGN_UUID, UUID | String | 默认不含中划线的UUID生成 |

Spring-Boot

方式一：声明为Bean供Spring扫描注入

```
1. @Component
2. public class CustomIdGenerator implements IdentifierGenerator {
3.     @Override
4.     public Long nextId(Object entity) {
5.         //可以将当前传入的class全类名来作为bizKey,或者提取参数来生成bizKey进行分布式Id调用生成.
6.         String bizKey = entity.getClass().getName();
7.         //根据bizKey调用分布式ID生成
8.         long id = ....;
9.         //返回生成的id值即可.
10.        return id;
11.    }
12. }
```

方式二：使用配置类

```
1. @Bean
2. public IdentifierGenerator idGenerator() {
3.     return new CustomIdGenerator();
4. }
```

方式三：通过MybatisPlusPropertiesCustomizer自定义

```
1. @Bean
2. public MybatisPlusPropertiesCustomizer plusPropertiesCustomizer() {
3.     return plusProperties -> plusProperties.getGlobalConfig().setIdentifierGenerator(new CustomIdGenerator());
4. }
```

Spring

方式一：XML配置

```
1. <bean name="customIdGenerator" class="com.baomidou.samples.incrementer.CustomIdGenerator"/>
2.
3. <bean id="globalConfig" class="com.baomidou.mybatisplus.core.config.GlobalConfig">
4.     <property name="identifierGenerator" ref="customIdGenerator"/>
5. </bean>
```

方式二：注解配置

```
1. @Bean
2. public GlobalConfig globalConfig() {
3.     GlobalConfig conf = new GlobalConfig();
4.     conf.setIdentifierGenerator(new CustomIdGenerator());
5.     return conf;
6. }
```


- [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](#)
- [数据安全保护](#)
- [多数据源](#)
- [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" value="classpath*:mybatis/mappers/**/*.xml"/>
13.        <constructor-arg name="enabled" value="true"/>
14.    </bean>
15.
16. 构造2:
17.    <bean class="com.baomidou.mybatisplus.spring.MybatisMapperRefresh">
18.        <constructor-arg name="sqlSessionFactory" ref="sqlSessionFactory"/>
19.        <constructor-arg name="mapperLocations" value="classpath*:mybatis/mappers/**/*.xml"/>
20.        <constructor-arg name="delaySeconds" value="10"/>
21.        <constructor-arg name="sleepSeconds" value="20"/>
22.        <constructor-arg name="enabled" value="true"/>
23.    </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-field: flag #全局逻辑删除字段值 3.3.0开始支持, 详情看下面。
5.       logic-delete-value: 1 # 逻辑已删除值(默认为 1)
6.       logic-not-delete-value: 0 # 逻辑未删除值(默认为 0)
```

- 实体类字段上加上 @TableLogic 注解

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

说明：

- 字段支持所有数据类型(推荐使用 Integer , Boolean , LocalDateTime)
- 如果使用 LocalDateTime , 建议逻辑未删除值设置为字符串 null , 逻辑删除值只支持数据库函数例如 now()
- 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 update user set deleted=1 where id =1 and deleted=0
3. 查找 select * from user where deleted=0
```

- 全局逻辑删除：begin 3.3.0

如果公司代码比较规范，比如统一了全局都是flag为逻辑删除字段。

使用此配置则不需要在实体类上添加 @TableLogic。

但如果实体类上有 @TableLogic 则以实体上的为准，忽略全局。 即先查找注解再查找全局，都没有则此表没有逻辑删除。

```
1. mybatis-plus:
2.   global-config:
3.     db-config:
4.       logic-delete-field: flag #全局逻辑删除字段值
```

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，如果你无需使用原生枚举，可配置默认枚举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

1、声明通用枚举属性

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.     ....

```

如何序列化枚举值为数据库存储值?

Jackson

一、重写toString方法

springboot

```

1.     @Bean
2.     public Jackson2ObjectMapperBuilderCustomizer customizer(){
3.         return builder -> builder.featuresToEnable(SerializationFeature.WRITE_ENUMS_USING_TO_STRING);
4.     }

```

jackson

```

1.     ObjectMapper objectMapper = new ObjectMapper();
2.     objectMapper.configure(SerializationFeature.WRITE_ENUMS_USING_TO_STRING, true);

```

以上两种方式任选其一,然后在枚举中复写toString方法即可。

二、注解处理

```

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
11.    @JsonValue    //标记响应json值
12.    private final int code;
13. }

```

Fastjson

一、重写toString方法

全局处理方式

```

1.     FastJsonConfig config = new FastJsonConfig();
2.     config.setSerializerFeatures(SerializerFeature.WriteEnumUsingToString);

```

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。

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" />
```

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. @Slf4j
2. @Component
3. public class MyMetaObjectHandler implements MetaObjectHandler {
4.
5.     @Override
6.     public void insertFill(MetaObject metaObject) {
7.         log.info("start insert fill ....");
8.         this.strictInsertFill(metaObject, "createTime", LocalDateTime.class, LocalDateTime.now()); // 起始版本
9.         3.3.0(推荐使用)
10.        this.fillStrategy(metaObject, "createTime", LocalDateTime.now()); // 也可以使用(3.3.0 该方法有bug请升级到之
11.        后的版本如`3.3.1.8-SNAPSHOT`)
12.        /* 上面选其一使用,下面的已过时(注意 strictInsertFill 有多个方法,详细查看源码) */
13.        //this.setFieldValByName("operator", "Jerry", metaObject);
14.        //this.setInsertFieldValByName("operator", "Jerry", metaObject);
15.    }
16.
17.    @Override
18.    public void updateFill(MetaObject metaObject) {
19.        log.info("start update fill ....");
20.        this.strictUpdateFill(metaObject, "updateTime", LocalDateTime.class, LocalDateTime.now()); // 起始版本
21.        3.3.0(推荐使用)
22.        this.fillStrategy(metaObject, "updateTime", LocalDateTime.now()); // 也可以使用(3.3.0 该方法有bug请升级到之
23.        后的版本如`3.3.1.8-SNAPSHOT`)
24.        /* 上面选其一使用,下面的已过时(注意 strictUpdateFill 有多个方法,详细查看源码) */
25.        //this.setFieldValByName("operator", "Tom", metaObject);
26.        //this.setUpdateFieldValByName("operator", "Tom", metaObject);
27.    }
28. }
```

注意事项：

- 字段必须声明 `TableField` 注解, 属性 `fill` 选择对应策略, 该声明告知 `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` 或 `@Bean` 注入
- 要想根据注解 `FieldFill.xxx` 和 字段名 以及 字段类型 来区分必须使用父类的 `strictInsertFill` 或者 `strictUpdateFill` 方法
- 不需要根据任何来区分可以使用父类的 `fillStrategy` 方法

```
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<?> mapperClass);
12. }
```

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 interceptor="com.baomidou.mybatisplus.extension.plugins.PerformanceInterceptor">
6.         <property name="maxTime" value="100" />
7.         <!--SQL是否格式化 默认false-->
8.         <property name="format" value="true" />
9.     </plugin>
10. </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. #3.2.1以上使用
2. modulelist=com.baomidou.mybatisplus.extension.p6spy.MybatisPlusLogFactory,com.p6spy.engine.outage.P6OutageFactory
3. #3.2.1以下使用或者不配置
4. #modulelist=com.p6spy.engine.logging.P6LogFactory,com.p6spy.engine.outage.P6OutageFactory
5. # 自定义日志打印
6. logMessageFormat=com.baomidou.mybatisplus.extension.p6spy.P6SpyLogger
7. #日志输出到控制台
8. appender=com.baomidou.mybatisplus.extension.p6spy.StdoutLogger
9. # 使用日志系统记录 sql
10. #appender=com.p6spy.engine.spy.appender.Slf4JLogger
11. # 设置 p6spy driver 代理
12. deregisterdrivers=true
13. # 取消JDBC URL前缀
14. useprefix=true
15. # 配置记录 Log 例外,可去掉的结果集有error,info,batch,debug,statement,commit,rollback,result,resultset.
```

```
16. excludecategories=info,debug,result,commit,resultset
17. # 日期格式
18. dateformat=yyyy-MM-dd HH:mm:ss
19. # 实际驱动可多个
20. #driverlist=org.h2.Driver
21. # 是否开启慢SQL记录
22. outagedetection=true
23. # 慢SQL记录标准 2 秒
24. 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
- 打印出sql为null,在excludecategories增加commit
- 批量操作不打印sql,去除excludecategories中的batch
- 批量操作打印重复的问题请使用MybatisPlusLogFactory (3.2.1新增)
- 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:

```
1. <bean 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
```

数据安全保护

该功能为了保护数据库配置及数据安全，在一定的程度上控制开发人员流动导致敏感信息泄露。

- 3.3.2 开始支持
- 配置安全

YML 配置：

```
1. // 加密配置 mpw: 开头紧接加密内容（非数据库配置专用 YML 中其它配置也是可以使用的）
2. spring:
3.   datasource:
4.     url: mpw:qRhvCwF4G0QjessEB3G+a5okP+uXXr96wcucn2Pev6BfaoEMZ1gVpPPhdDmjQqoM
5.     password: mpw:Hzy5iliJbwDHhJls1L0j6w==
6.     username: mpw:Xb+EgsyuYRXw7U7sBJjBpA==
```

密钥加密：

```
1. // 生成 16 位随机 AES 密钥
2. String randomKey = AES.generateRandomKey();
3.
4. // 随机密钥加密
5. String result = AES.encrypt(data, randomKey);
```

如何使用：

```
1. // Jar 启动参数（idea 设置 Program arguments）
2. --mpw.key=d1104d7c2b606f0b
```

- 数据安全：

待完善

注意！

- 加密配置必须以 mpw: 字符串开头
- 随机密钥请负责人妥善保管，当然越少人知道越好。

多数据源

一个基于springboot的快速集成多数据源的启动器

build passing

maven-central v3.1.0

license apache

JDK 1.7+

springBoot 1.4+ 1.5+ 2.0+

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**。

示例项目 可参考项目下的samples目录。

特性

1. 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。
2. 内置敏感参数加密和启动初始化表结构schema数据库database。
3. 提供对Druid, Mybatis-Plus, P6sy, Jndi的快速集成。
4. 简化Druid和HikariCp配置, 提供全局参数配置。
5. 提供自定义数据源来源接口(默认使用yml或properties配置)。
6. 提供项目启动后增减数据源方案。
7. 提供Mybatis环境下的 纯读写分离 方案。
8. 使用spel动态参数解析数据源, 如从session, header或参数中获取数据源。(多租户架构神器)
9. 提供多层数据源嵌套切换。(ServiceA >>> ServiceB >>> ServiceC, 每个Service都是不同的数据源)
10. 提供 不使用注解 而 使用 正则 或 **spel** 来切换数据源方案(实验性功能)。
11. 基于seata的分布式事务支持。

约定

1. 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。
2. 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。
3. 切换数据源可以是组名, 也可以是具体数据源名称。组名则切换时采用负载均衡算法切换。
4. 默认的数据源名称为 **master**, 你可以通过 `spring.datasource.dynamic.primary` 修改。
5. 方法上的注解优先于类上注解。

使用方法

1. 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>

```

1. 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.       strict: false #设置严格模式,默认false不启动. 启动后在未匹配到指定数据源时候回抛出异常,不启动会使用默认数据源.
6.       datasource:
7.         master:
8.           url: jdbc:mysql://xx.xx.xx.xx:3306/dynamic
9.           username: root
10.          password: 123456
11.          driver-class-name: com.mysql.jdbc.Driver
12.         slave_1:
13.           url: jdbc:mysql://xx.xx.xx.xx:3307/dynamic
14.           username: root
15.           password: 123456
16.           driver-class-name: com.mysql.jdbc.Driver
17.         slave_2:
18.           url: ENC(xxxxx) # 内置加密,使用请查看详细文档
19.           username: ENC(xxxxx)
20.           password: ENC(xxxxx)
21.           driver-class-name: com.mysql.jdbc.Driver
22.           schema: db/schema.sql # 配置则生效,自动初始化表结构
23.           data: db/data.sql # 配置则生效,自动初始化数据
24.           continue-on-error: true # 默认true,初始化失败是否继续
25.           separator: ";" # sql默认分号分隔符
26.
27.       #.....省略
28.       #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

```

| 1. # 多主多从 | 纯粹多库 (记得设置primary) | 混合配置 |
|----------------------|--------------------|-------------|
| 2. spring: | spring: | spring: |
| 3. datasource: | datasource: | datasource: |
| 4. dynamic: | dynamic: | dynamic: |
| 5. datasource: | datasource: | datasource: |
| 6. master_1: | mysql: | master: |
| 7. master_2: | oracle: | slave_1: |
| 8. slave_1: | sqlserver: | slave_2: |
| 9. slave_2: | postgresql: | oracle_1: |
| 10. slave_3: | h2: | oracle_2: |

1. 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。

强烈建议只注解在service实现上。

| 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. }
```

赶紧集成体验一下吧！ 如果需要更多功能请点击[下面链接查看详细文档！](#)

分布式事务，加密, Druid集成，MybatisPlus集成，动态增减数据源，自定义切换规则, 纯读写分离插件等等更多更
[细致的文档在这里](#) 

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.      * 这里固定写成租户 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（ 注意观察 SQL ）<br>
7.      */
8.     List<ISqlParser> sqlParserList = new ArrayList<>();
9.     TenantSqlParser tenantSqlParser = new TenantSqlParser();
10.    tenantSqlParser.setTenantHandler(new TenantHandler() {
11.        @Override
12.        public Expression getTenantId(boolean where) {
13.            // 该 where 条件 3.2.0 版本开始添加的，用于分区是否是在 where 条件中使用
14.            // 如果是in/between之类的多个tenantId的情况，参考下方示例
15.            return new LongValue(1L);
16.        }
17.
18.        @Override
19.        public String getTenantIdColumn() {
20.            return "tenant_id";
21.        }
22.
23.        @Override
24.        public boolean doTableFilter(String tableName) {
25.            // 这里可以判断是否过滤表
26.            /*
27.            if ("user".equals(tableName)) {
28.                return true;
29.            }*/
30.            return false;
31.        }
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 = SqlParserHelper.getMappedStatement(metaObject);
39.            // 过滤自定义查询此时无租户信息约束【 麻花藤 】出现

```

```

40.         if ("com.baomidou.springboot.mapper.UserMapper.selectListBySQL".equals(ms.getId())) {
41.             return true;
42.         }
43.         return false;
44.     }
45. });
46. return paginationInterceptor;
47. }

```

- 关于多租户实现条件tenant_id in (1,2,3)的解决方案

核心代码： MybatisPlusConfig

```

1.  /**
2.   * 2019-8-1
3.   *
4.   * https://gitee.com/baomidou/mybatis-plus/issues/IZZ3M
5.   *
6.   * 参考示例：
7.   * https://gitee.com/baomidou/mybatis-plus-samples/tree/master/mybatis-plus-sample-tenant
8.   *
9.   * tenant_id in (1,2)
10.  *
11.  * @return
12.  */
13. @Override
14. public Expression getTenantId(boolean where) {
15.     //如果是where, 可以追加多租户多个条件in, 不是where的情况：比如当insert时, 不能insert into user(name, tenant_id)
16.     values('test', tenant_id IN (1, 2));
17.     final boolean multipleTenantIds = true;//自己判断是单个tenantId还是需要多个id in(1,2,3)
18.     if (where && multipleTenantIds) {
19.         //演示如何实现tenant_id in (1,2)
20.         return multipleTenantIdCondition();
21.     } else {
22.         //演示：tenant_id=1
23.         return singleTenantIdCondition();
24.     }
25. }
26. private Expression singleTenantIdCondition() {
27.     return new LongValue(1);//ID自己想办法获取到
28. }
29.
30. private Expression multipleTenantIdCondition() {
31.     final InExpression inExpression = new InExpression();
32.     inExpression.setLeftExpression(new Column(getTenantIdColumn()));
33.     final ExpressionList itemsList = new ExpressionList();
34.     final List<Expression> inValues = new ArrayList<>(2);
35.     inValues.add(new LongValue(1));//ID自己想办法获取到
36.     inValues.add(new LongValue(2));
37.     itemsList.setExpressions(inValues);
38.     inExpression.setRightItemsList(itemsList);
39.     return inExpression;

```



```
40.     }
41.
42.
43.     public class MyTenantParser extends TenantSqlParser {
44.
45.         //目前这种情况比较小众，自己定制可以参考
46.         //参考 https://gitee.com/baomidou/mybatis-plus-samples/blob/master/mybatis-plus-sample-tenant/src/main/java/com/baomidou/mybatisplus/samples/tenant/config/MyTenantParser.java
47.     }
```

- 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.
2.
3.
4. # 开启 SQL 解析缓存注解生效
5. mybatis-plus:
6.     global-config:
7.         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

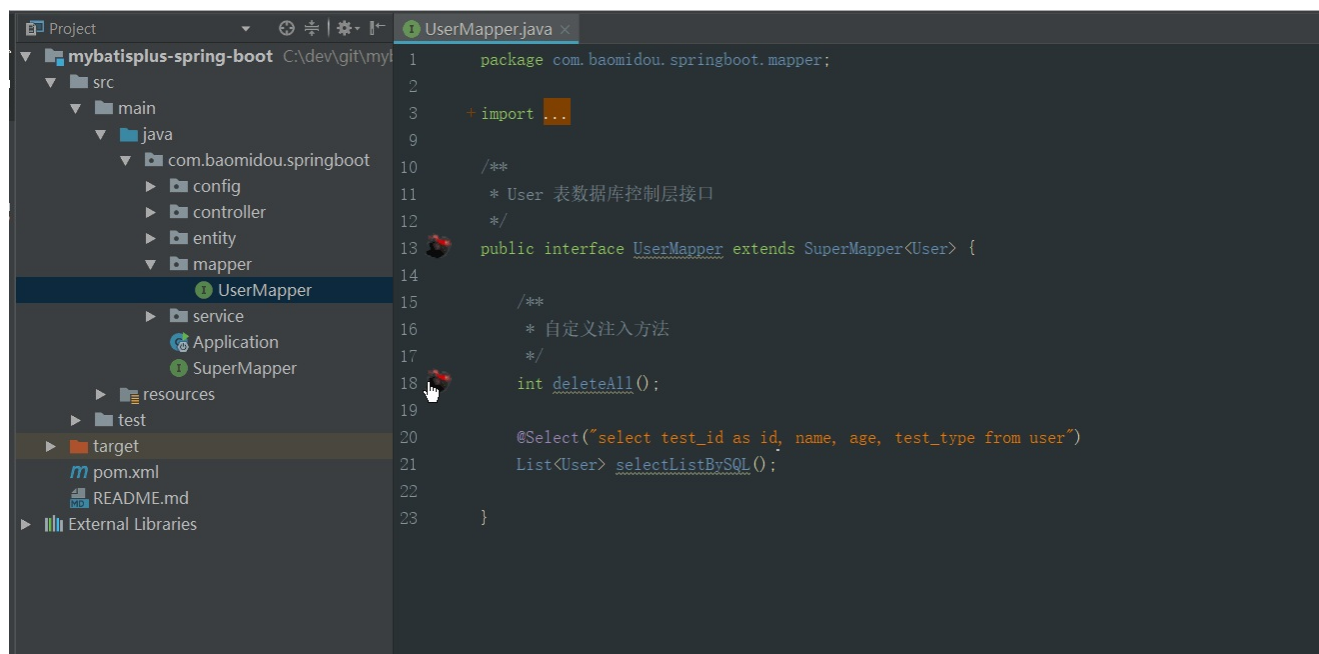
MybatisX 是一款基于 IDEA 的快速开发插件，为效率而生。

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

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:

```
1. <bean id="sqlSessionFactory" 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.
7. <bean id="configuration" class="com.baomidou.mybatisplus.core.MybatisConfiguration">
8.     .....
9. </bean>
10.
11. <bean id="globalConfig" class="com.baomidou.mybatisplus.core.config.GlobalConfig">
12.     <property name="dbConfig" ref="dbConfig"/> <!-- 非必须 -->
13.     .....
14. </bean>
15.
16. <bean id="dbConfig" 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

localCacheScope

- 类型： `String`
- 默认值： `SESSION`

Mybatis一级缓存，默认为 `SESSION`。

- `SESSION` session级别缓存，同一个session相同查询语句不会再次查询数据库
- `STATEMENT` 关闭一级缓存

单服务架构中（有且仅有只有一个程序提供相同服务），一级缓存开启不会影响业务，只会提高性能。微服务架构中需要关闭一级缓存，原因：`Service1`先查询数据，若之后`Service2`修改了数据，之后`Service1`又再次以同样的查询条件查询数据，因走缓存会出现查处的数据不是最新数据

cacheEnabled

- 类型： `boolean`
- 默认值： `true`

开启Mybatis二级缓存，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

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 3.3.0)

- 类型: `com.baomidou.mybatisplus.core.incrementer.IdentifierGenerator`
- 默认值: `com.baomidou.mybatisplus.core.incrementer.DefaultIdentifierGenerator`

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

idType

- 类型: `com.baomidou.mybatisplus.annotation.IdType`
- 默认值: `ASSIGN_ID`

全局默认主键类型

tablePrefix

- 类型: `String`
- 默认值: `null`

表名前缀

schema

- 类型: `String`
- 默认值: `null`

schema

columnFormat

- 类型: `String`
- 默认值: `null`

字段 `format`, 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

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)

insertStrategy

- 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

- 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

enableSqlFilter (since 3.3.1)

默认激活进行sql模糊表名匹配

关闭之后likeTable与notLikeTable将失效，include和exclude将使用内存过滤

如果有sql语法兼容性问题的话，请手动设置为false

已知无法使用：微软系，达梦，MyCat中间件， [支持情况传送门](#) 

include

需要包含的表名，当enableSqlFilter为false时，允许正则表达式（与exclude二选一配置）

likeTable

自3.3.0起，模糊匹配表名（与notLikeTable二选一配置）

exclude

需要排除的表名，当enableSqlFilter为false时，允许正则表达式

notLikeTable

自3.3.0起，模糊排除表名

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），自3.3.2开始更名为 [chainModel](#)

chainModel (since 3.3.2)

【实体】是否为链式模型（默认 false）

entityLombokModel

【实体】是否为lombok模型（默认 false）

3.3.2以下版本默认生成了链式模型，3.3.2以后，默认不生成，如有需要，请开启 [chainModel](#)

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)

- [常见问题](#)
- [捐赠支持](#)

常见问题

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](#)）

```

1. <bean id="sqlSessionFactory" class="com.baomidou.mybatisplus.extension.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
1. 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：

```

1. import com.baomidou.mybatisplus.core.mapper.BaseMapper;
2.
3. public interface SuperMapper<T> extends BaseMapper<T> {
4.     // your methods
5. }

```

- 异常二：

```

1. 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()
1. Log/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

```
1. @Override
2. public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
3.     FastJsonHttpMessageConverter fastJsonConverter = new FastJsonHttpMessageConverter();
4.     FastJsonConfig fjc = new FastJsonConfig();
5.     // 配置序列化策略
6.     fjc.setSerializerFeatures(SerializerFeature.BrowserCompatible);
7.     fastJsonConverter.setFastJsonConfig(fjc);
8.     converters.add(fastJsonConverter);
9. }
```

- JackJson 处理方式

- 方式一

```
1. // 注解处理，这里可以配置公共 BaseEntity 处理
2. @JsonSerialize(using=ToStringSerializer.class)
3. public long getId() {
4.     return id;
5. }
```

- 方式二

```
1. // 全局配置序列化返回 JSON 处理
2. final ObjectMapper objectMapper = new ObjectMapper();
3. SimpleModule simpleModule = new SimpleModule();
4. simpleModule.addSerializer(Long.class, ToStringSerializer.instance);
5. 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：

```
1. @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：

```
1. //updateAllColumnById(entity) // 全部字段更新：3.0已经移除
2. mapper.update(
3.     new User().setName("mp").setAge(3),
4.     Wrappers.<User>lambdaUpdate()
5.         .set(User::getEmail, null) //把email设置成null
6.         .eq(User::getId, 2)
7. );
8. //也可以参考下面这种写法
9. mapper.update(
10.     null,
11.     Wrappers.<User>lambdaUpdate()
12.         .set(User::getAge, 3)
13.         .set(User::getName, "mp")
14.         .set(User::getEmail, null) //把email设置成null
15.         .eq(User::getId, 2)
16. );
```

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：

```
1. 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

```

1. //2.x
2. EntityWrapper<H2User> ew = new EntityWrapper<>();
3. ew.setSqlSelect("test_id as id, name, age");//只查询3个字段
4. List<H2User> list = userService.selectList(ew);
5. for(H2User u:list){
6.     Assert.assertNotNull(u.getId());
7.     Assert.assertNotNull(u.getName());
8.     Assert.assertNull(u.getPrice()); // 这个字段没有查询出来
9. }
10.
11. //3.x
12. mapper.selectList(
13.     wrappers.<User>lambdaQuery()
14.     .select(User::getId, User::getName)
15. );
16. //或者使用QueryWrapper
17. mapper.selectList(
18.     new QueryWrapper<User>()
19.     .select("id","name")
20. );

```

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

```

1. @CacheNamespace(implementation=MybatisRedisCache.class,eviction=MybatisRedisCache.class)
2. 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

```

1. <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=NULL Spring Bean 配置方式:

```
1. MybatisConfiguration configuration = new MybatisConfiguration();
2. configuration.setDefaultScriptingLanguage(MybatisXMLLanguageDriver.class);
3. configuration.setJdbcTypeForNull(JdbcType.NULL);
4. configuration.setMapUnderscoreToCamelCase(true); //开启下划线转驼峰
5. sqlSessionSessionFactory.setConfiguration(configuration);
```

yaml 配置

```
1. mybatis-plus:
2.   configuration:
3.     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

```
1. @Bean
2. public ConfigurationCustomizer configurationCustomizer() {
3.     return i -> i.setObjectWrapperFactory(new MybatisMapWrapperFactory());
4. }
```

在 wrapper 中如何使用 limit 限制 SQL

```
1. // 取 1 条数据
2. 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

i. 配置自定义的 SqlInjector

```

1. @Bean
2. public LogicSqlInjector logicSqlInjector(){
3.     return new LogicSqlInjector() {
4.         /**
5.          * 注入自定义全局方法
6.          */
7.         @Override
8.         public List<AbstractMethod> getMethodList() {
9.             List<AbstractMethod> methodList = super.getMethodList();
10.            methodList.add(new LogicDeleteByIdWithFill());
11.            return methodList;
12.        }
13.    };
14. }
```

i. 配置自己的全局 baseMapper 并使用

```

1. public interface MyBaseMapper<T> extends BaseMapper<T> {
2.
3.     /**
4.      * 自定义全局方法
5.      */
6.     int deleteByIdWithFill(T entity);
7. }
```

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。

1. 不同的数据库对关键字的处理不同, 很难维护。

2. 在数据库设计时候本身不建议使用关键字。
3. 交由用户去处理关键字。

```
1. @TableField(value = "`status`")
2. 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.SQLException

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.SQLException

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在1.1.21版本之前不支持, [参考issue](#)

解决方案: 1. 升级druid到1.1.21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

mp版本从3.1.0及以下版本升级到高版本, JDK8日期新类型LocalDateTime等无法映射 (报错)

MP_3.1.0及之前的版本, 依赖的是mybatis 3.5.0,

MP_3.1.1升级了mybatis的依赖到3.5.1, 而mybatis 3.5.1 对于新日期类型, 需要JDBC driver支持JDBC 4.2 API.

如果你的jdbc驱动版本不支持, 那么就会出现新日期类型报错。

参考 blog.mybatis.org

1. `There is` one backward incompatible changes since `3.5.0`.
Because of the fix for `#1478`, `LocalDateTypeHandler`, `LocalTimeTypeHandler` and `LocalDateTimeTypeHandler` now
2. require a JDBC driver that supports JDBC 4.2 API.
3. `[EDIT]` These type handlers no longer work with Druid. Please see `#1516`.

Failed to bind properties under 'mybatis-plus.configuration.incomplete-result-maps[0].assistant.configuration.mapped-statements[0].parameter-map.parameter-mappings[0]' to org.apache.ibatis.mapping.ParameterMapping

springboot 2.2.0 之前无此问题，springboot 2.2.0 出现此问题

现象：1.本地启动无问题，打成war包部署到服务器报此问题

原因：springboot 2.2.0 构造器注入的问题，mybatis 私有构造器不能绑定属性，造成依赖mybatis的框架比如MP报错 [参考issue] (<https://github.com/spring-projects/spring-boot/issues/18670>) 此问题已在springboot2.2.1中修复

解决方案：1.将springboot降级到2.1.x或升级到2.2.1起（建议springboot2.2.2）

分离打包部署出现ClassNotFoundException

针对3.3.2以下版本：

现象：开发工具运行没问题，打包部署服务器执行lambda表达式出现ClassNotFoundException

原因：执行反序列化时类加载器错误

解决方案：去除spring-boot-maven-plugin插件进行打包或升级至3.3.2，参考示例 ([分离打包](#))

启动 mybatis 本身的 log 日志

1. `# 方式一`
2. mybatis-plus:
3. configuration:
4. log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
- 5.
6. `# 方式二` application.yml 中增加配置，指定 mapper 文件所在的包
7. logging:
8. level:
9. com.baomidou.example.mapper: debug

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

CHANGELOG

[v3.3.2] 2020.5.26

- 分页参数提取, 单元测试用例修复
- 达梦数据库代码生成器表过滤支持
- 微软数据库代码生成器表过滤支持
- 修复代码生成器属性字段规则错误
- SelectById 支持自定义方法名
- 修复分页插件获取数据库类型问题
- Json转换器空值处理
- bugfix(mybatis-plus-generator):SQL类型返回错误问题
- 调整未知方言异常, 自动识别url转换小写匹配.
- fix: 初始化 TableInfo 中遇到多个字段有 @TableId 注解时未能抛出异常的问题
- SuperController有Class参数的set方法
- 增加方法StrategyConfig.setSuperServiceImplClass(java.lang.Class<?>).
- 代码生成器命名策略调整.
- 扩展分页缓存key值计算.
- 去除方法推测, 直接访问属性字段.
- 修正枚举处理器类型不匹配比较.
- 修改表前缀匹配方式
- 修改在Mybatis全局配置文件中设置分页插件参数不生效问题
- 修改在Mybatis全局配置文件中设置分页插件参数不生效问
- 修复PR未指定解析器的时候引发空指针
- 增加分页插件limit参数配置
- 修复指定superEntityClass重复生成父类字段问题
- 无主键的情况无需导入IdType与TableId包
- 调整生成BaseResultMap格式
- 支持lombok模式下选择是否进行链式set生成
- 修复解析器for update错误
- 过滤PG约束列(只留下主键约束)
- 增加生成器禁用模板生成
- fix(kotlin): 修复动态表名 BUG, 最大努力替换表名
- 修复PG约束生成重复属性字段问题
- fix(kotlin): 将 LambdaUtils 中缓存的 key 改为 String
- 代码生成器增加数据库关键字处理接口
- fix github/issues/2454 支持注解可继承
- 新增 AES 加密数据库用户名密码
- 优化方法入参泛型, 支持更多类型
- 修复代码生成器开启移除is前缀生成实体缺少包导入
- fixed github issues/2470

[v3.3.1] 2020.1.17

- 新增 `TableName` 注解属性 `excludeProperty` 支持排除字段
- 新增 `ServiceImpl#entityClass` 属性, 减少泛型提取
- 新增 phoenix 支持
- 新增支持 hbase 的选装件 `Upsert`
- 新增生成器策略配置 `enableSqlFilter` 属性来控制是否启用 SQL 过滤表支持
- 新增批量执行方法, 方便用户自定义批量执行操作
- `Wrapper` 支持 `clear` 清空
- `Wrapper` 子类新增 `func` 方法, 主要为了支持在 `if else` 情况下使用 `Wrapper` 的不同 method 不会导致断链 (链式调用不能一链到底)
- `BaseMapper` 部分入参为 `Wrapper` 的 `select` 方法支持 `wrapper.first` 来设置 RDS 的 hint
- `KtUpdateWrapper#set` 支持 value 为 null
- 支持泛型主键支持
- 优化分页拦截器数据类型与方言实现类配置
- 二级缓存复用 count 查询缓存
- `IService` 部分 method 调整为 default 方法
- 二级缓存兼容 json 序列化情况 (主要出现默认缓存 count 出现 long 反序列化回来为 int 的情况)
- 处理批量操作嵌套事物问题 (二级缓存更新问题)
- 修复启用乐观锁下 `updateById` 时自动填充不生效的问题
- 修复自动填充接口的 default 方法 (`setFieldValByName` 和 `getFieldValByName`) 某些情况下会发生异常的问题
- 修复 `KtWrapper` 嵌套函数问题
- 修复 Freemarker 生成 Kotlin 类的常量错误
- 修复 `StringUtils#guessGetterName` 错误
- 修复 `SerializationUtils` 资源未释放问题

[v3.3.0] 2019.12.06

- `BaseMapper` 接口两个 `page` 方法优化
- `IService` 以及 `ServiceImpl` 对应 `page` 方法优化, 个别返回 `collection` 的方法修改为返回 `list`
- 逻辑删除字段的两个表示已删除和未删除的定义支持字符串 `"null"`
- 修复批量操作未清空缓存
- 批量操作异常转换为 `DataAccessException`
- mybatis up 3.5.3, mybatis-spring up 2.0.3, jsqlparser up 3.1
- mapper 选装件包调整, chainWrapper 包调整
- 新增 `ChainWrappers` 工具类
- 新增 `IdentifierGenerator` 接口, 支持自定义 Id 生成
- 代码生成工具废弃正则表名匹配, 新增 `likeTable` 与 `notLikeTable`
- 分页插件支持自定义处理页数限制与溢出总页数处理
- 修复 `SqlExplainInterceptor` 导致的 Oracle 序列自增两次
- 分页二级缓存支持
- 扩展 p6spy 日志打印
- `DbConfig` 加入新属性 `propertyFormat`, `TableFieldInfo` 移除属性 `related`
- 优化序列生成器, 过时 `KeySequence` 的 `clazz` 属性
- 修复 `Ognl` 表达式关键字导致的 null 值判断失效
- 修复更新填充开关失效
- 优化填充逻辑
- `ISqlRunner` 支持 `selectPage`

- 支持全局逻辑删除字段
- BaseMapper的方法可自定义
- 添加【虚谷】【Oracle12c】【Kingbase】数据库支持
- 解决数据库字段与实体字段名称不同时出现 `null as xxx` 的情况
- 过时ID_WORKER_STR, 自动识别主键类型
- 配置开启注解, TableName也强制生成

[v3.2.0] 2019.08.26

- 代码生成器添加达梦数据库支持
- 修复多主键查询表字段SQL的Bug
- 新增 updateWrapper 尝试更新, 否继续执行saveOrUpdate(T)方法
- 代码生成器 pg 增加 numeric instant 类型支持
- 修复InjectionConfig不存在时无法生成代码的问题
- fix: #1386(github) 逻辑删除字段为Date类型并且非删除数据日期为null
- 升级依赖 mybatis 版本为 3.5.2
- 升级依赖 jsqlparser 版本为 2.1
- 应 EasyScheduler 计划提交 Apache 孵化请求移除 996NPL 协议限制
- 调整 SQL 移除 SET 部分 Github/1460
- 移除 SqlMethod 枚举 UPDATE_ALL_COLUMN_BY_ID 属性, 推荐使用 AlwaysUpdateSomeColumnById 套
- fix: #1412(github) github:mybatis-plus-generator can't support oracle
- fix: github 1380
- 移除全局配置的 dbType 和 columnLike
- 移除 fieldStrategy, 使用上个版本新增的三个替代
- 移除 PerformanceInterceptor 相关, 建议使用 p6spy
- 移除 el 拆分为 jdbcType typeHandler 等具体属性
- 升级 gradle-5.5.1,lombok-1.18.4
- 当selectStatement.getSelectBody()的类型为SetOperationList
- 移除 GlobalConfig#sqlParserCache 属性, 移除 LogicSqlInjector, OrderItem 新增2个快捷生成的method, page 新增一个入参是 List 的 addOrder method
- Nested 接口个别入参是 `Function<Param, Param> func` 的method, 入参更改为 `Consumer<Param> consumer`, 不影响规范的使用
- fixed gitee/I10XWC 允许根据 TableField 信息判断自定义类型
- Merge pull request #1445 from kana112233/3.0
- 支持过滤父类属性功能
- 添加批量异常捕获测试
- 多租户ID 值表达式, 支持多个 ID 条件查询
- 扩展新增 json 类型处理器 jackson fastjson 两种实现

[v3.1.2] 2019.06.26

- EnumTypeHandler 更名为 MybatisEnumTypeHandler, 移除 EnumAnnotationTypeHandler
- 新增自动构建 resultMap 功能, 去除转义符
- 注解增加变量控制是否自动生成resultmap

- 修改分页缓存Key值错误
- TableField.el 属性标记过时
- 取消 MybatisMapWrapperFactory 的自动注册
- starter 增加默认xml路径扫描
- 新增 MybatisPlusPropertiesCustomizer 及配置使用
- ConfigurationCustomizer 内部方法入参更新为 MybatisConfiguration
- 原有 fieldStrategy 标记过时,新增 3 种 fieldStrategy 进行区分
- 获取注入方法时传递当前mapperClass
- 增加sqlite代码自动生成测试代码及测试用的数据库文件
- JsqlParserCountOptimize 对 left join 的 sql 优化 count 更精确
- fix(AbstractWrapper.java): 修复 lambda 表达式在 order、groupBy 只有条件一个时引起的类型推断错误
- apply plugin: 'kotlin'
- refactor(order): 修复排序字段优先级问题(#IX1Q0)
- 启动就缓存 lambdacache
- Merge pull request #1213 from sandynz/feature/sqlComment 支持SQL注释
- 去除 wrapper 的一些变量,wrapper 内部 string 传递优化
- fix: #1160(github) 分页组件orderBy: 同时存在group by 和order by, 且IPage 参数中存在排序属性时, 拼接
- Merge pull request #1253 from ShammgodYoung/patch-1 代码生成器输入表名忽略大小写
- 新增渲染对象 MAP 信息预处理注入
- 修改 dts rabbitAdmin bean 判断方式
- Merge pull request #1255 from ShammgodYoung/patch-2 对serialVersionUID属性进行缩进
- JsqlParserCountOptimize 加入 boolean 字段,判断是否优化 join
- Merge pull request #1256 from baomidou/master Master
- freemarker entity 模板缩进调整
- 增加jdbcType,typeHandler属性, 合并el属性

[v3.1.1] 2019.04.25

- 新增 996icu license 协议
- 新增 mybatis-plus-dts 分布式事务 rabbit 可靠消息机制
- 新增 DynamicTableNameParser 解析器、支持动态表名
- 优化 getOne 日志打印
- sql 优化跳过存储过程
- 优化分页查询(count为0不继续查询)
- 修复分页一级缓存无法继续翻页问题
- MybatisMapWrapperFactory 自动注入
- 支持纯注解下使用 IPage 的子类作为返回值
- 逻辑删除不再需要 LogicInject
- GlobalConfig 加入 enableSqlRunner 属性控制是否注入 SqlRunner ,默认 false
- SqlParser注解不再需要全局设置参数才会缓存,以及支持注解在 mapper 上
- GlobalConfig 的 sqlParserCache 设置为过时
- mybatis 升级到 3.5.1 , mybatis-spring 升级到 2.0.1 , jsqlparser 降级到 1.2
- ISqlInjector 接口 移除 injectSqlRunner 方法
- SqlFormatter 类设置为过时

- 解决自动注入的 `method` 的 `SqlCommandType` 在逻辑删除下混乱问题
- 新增 `AlwaysUpdateSomeColumnById` 选装件
- `SFunction` 继承 `Function`
- `DbConfig` 的 `columnLike` 和 `dbType` 属性设置为过时
- `DbConfig` 新增 `schema` 和 `columnFormat` 属性
- `TableField` 注解增加 `keepGlobalFormat` 属性
- `TableName` 注解增加 `schema` 和 `keepGlobalPrefix` 属性
- fixed bug tmp文件格式错乱 github #1048
- 处理表/字段名称抽象 `INameConvert` 接口策略 github #1038
- DB2支持动态 `schema` 配置 github #1035
- 把字段缓存的key从className替换成了.class, 如果使用dev-tools会导致: `MybatisPlusException: Your property named "xxxx" cannot find the corresponding database column name!`(解决方案: 去掉dev-tools)

[v3.1.0] 2019.02.24

- 升级 `mybatis` 到 `3.5.0` 版本
- 升级 `mybatis-spring` 到 `2.0.0` 版本
- 升级 `jsqlparser` 到 `1.4` 版本
- 新增 p6spy 日志打印支持
- 变更 `IService` 的 `getOne(wrapper<T> queryWrapper)` 方法如果获取到多条数据将会抛出 `TooManyResultsException` 异常
- 修复 自定义分页功能不支持注解 `@select` 问题
- 修复 生成器的配置 `kotlin` 模式下 `swagger` 模式无效问题
- 修复 生成器 `is` 开头字段无法自动注解问题
- 修复 生成器 `Serializable` `Active` 模式继承父类包自动导入异常问题
- 修复 生成器 支持公共字段自动读取父类 `class` 属性问题
- 修复 枚举(注解方式)转换器在存储过程中转换失败
- 修复 `beetl` 模板逻辑删除注解错误问题
- 修复 通过 `mybatis-config.xml` 方式构建的 `Configuration` 的 `mapUnderscoreToCamelCase` 默认值非 `true` 的问题
- 修复 `sql`解析器动态代理引发的bug
- 修复 `mapper` 使用纯注解下可能触发的重试机制在个别情况下启动报错的问题
- 优化 支持指定 `defaultEnumTypeHandler` 来进行通用枚举处理
- 优化 从 `hibernate copy` 最新代码到 `SqlFormatter`
- 移除 `wrapper` 的 `in` 以及 `notIn` 方法内部对入参 `coll` 及 `动态数组` 的非empty判断(注意: 如果以前有直接使用以上的方法的入参可能为 `empty` 的现在会产出如下sql: `in ()` 或 `not in ()` 导致报错)
- 移除 `wrapper` 的 `notInOrThrow` 和 `inOrThrow` 方法(使用新版的 `in` 以及 `notIn` 效果一样, 异常则为 `sql`异常)
- 移除 `IService` 的 `query` 链式调用的 `delete` 操作
- 移除 `xml` 热加载相关配置项, 只保留 `MybatisMapperRefresh` 该类并打上过时标志
- 日常优化

[v3.0.7.1] 2019.01.02

- 修复 `lambdaWrapper` 的获取不到主键缓存的问题
- 优化 `IService` 新增的 `update` 链式调用支持 `remove` 操作
- 过时 `IService` 新增的 `query` 链式调用的 `delete` 打上过时标识
- 日常优化

[v3.0.7] 2019.01.01

- 优化 `generator` 的 `postgresSql` 数据库支持生成 `java8` 时间类型
- 优化 `generator` 的 `sqlServer` 数据库支持生成 `java8` 时间类型
- 优化 `LambdaWrapper` 反射获取字段信息支持首字母大写的字段
- 优化 仅 `LambdaWrapper` 的 `select` 优化(支持字段对不上数据库时自动 `as`)
- 优化 重复扫描 `BaseMapper` 子类时, `TableInfo` 缓存的 `Configuration` 只保留最后一个
- 优化 `MergeSegments` 获取 `getSqlSegment` 方式
- 优化 SQL 自动注入器的初始化 `modelClass` 过程, 提高初始化速度
- 优化 `BaseMapper` 的 `update` 方法的第一个入参支持为 `null`
- 新增 `IService` 增加4个链式调用方法
- 新增 代码生成器增加 `beetl` 模板
- 新增 `IdWorker` 增加毫秒时间 ID 可用于订单 ID
- 新增 `wrapper` 新增 `inOrThrow` 方法, 入参为 `empty` 则抛出 `MybatisPlusException` 异常
- 新增 `MetaObjectHandler` 新提供几个能根据注解才插入值的 `default` 方法
- 新增 `kotlin` 下 `lambda` 的支持, `KtQueryWrapper` 和 `KtUpdateWrapper` 类
- 新增 简化MP自定义SQL使用方法, 现在可以使用 `自定义sql` + `${ew.customSqlSegment}` 方式
- 新增 提供新的 `InsertBatchSomeColumn` 选装件
- 修复 `Page` 的 `setTotal(Long total) -> setTotal(long total)`
- 修复 `Page` 的 `setSearchCount` 为 `public`
- 修复 `TenantSqlParser` 如果 `where` 条件的开头是一个 `orExpression`, 直接在左边用`and`拼接租户信息会造成逻辑不符合预期的问题
- 修复 `wrapper` 的 `lambda` 方法会向下传递 `sqlSelect`
- 修复 `ServiceImpl` 个别 `batch` 操作 `flushStatements` 问题
- 修复 `selectObjs` 泛型错误问题
- 移除 `InsertBatchAllColumn` 选装件
- 移除 `ServiceImpl` 的 `batch` 操作之外的事务注解
- 移除 `Model` 的事务注解
- 移除 `AbstractSqlInjector` 的 `isInjectSqlRunner` 方法(`SqlRunner`初始化较早, 目前 `isInjectSqlRunner`无法控制)
- 移除 `MybatisSessionFactoryBuilder`
- 移除 对 `mybatis-plus-generator` 包的依赖, 自己按需引入
- 还原 `xml` 热加载, 打上过时标识
- 升级 `jsqlparser` 依赖到 1.3
- 日常优化

[v3.0.6] 2018.11.18

- 修复entity中2个以上条件并且拼接ORDER BY 或 GROUP BY 产生的 WHERE X1 =? AND X2
- `refactor(SerializedLambda.java)`: 重构方法增加反序列化安全性, 优化命名

- 基础Mapper优化支持自定义父类Mapper构造自己需要的注入方法
- 使用代替
- 部分优化：直到抛出异常时才进行字符串 format
- 优化 IdWorker 生成UUID使用并发性能
- feat：动态分页模型、优化分页方言重新修正db2分页语句
- Assert 支持 i18n 多语言错误提示
- 支持 total 控制是否 count sql 新增 isSearchCount 方法
- feat: move spring dependency from core module to extension
- fix: Junit.assertTrue
- 强制使用自定义ParameterHandler, 去除byId类型限制.
- 新增选装件的 InsertBatch 通用方法, 以及相应测试, 以及代码和性能的优化
- IPage 新增功能, 泛型转换
- 自动填充判断填充值是否为空, 为空时跳过填充逻辑
- batchsize 阈值设 30 修改为 1000 提升效率
- 修复在极端情况下saveOrUpdate执行错误
- 移除 MybatisSqlSessionTemplate
- 移除 xml 热加载
- 其他优化

[v3.0.5] 2018.10.11

- 移除 ApiAssert 改为 Assert
- 移除 ApiResult 改为 R
- SQL 注入器优化
- 移除 excludeColumns 方法
- 修复 last 方法的 condition 入参不生效的问题
- 修复去除1=1 BUG
- 移除对 spring-devtools 的支持
- 修复实体属性都为null时Sql拼接出错问题
- 缓存Class反射信息, 提升效率
- 继承Model类的实体中, 现在无需重写pkVal()方法
- 解决在设置了config-location的情况下报mpe的bug, 以及优化初始化逻辑
- 修复存在 mapper.xml 情况下逻辑删除失效
- 调整 关于ServiceImpl中的事务问题 gitee issue/IN8T8
- 修复 DB2分页方言 github issues/526

[v3.0.4] 2018.09.28

- 修正全局配置 FieldStrategy 为非默认值
- 修正批量事务异常问题
- Api 层 R 类自动处理逻辑失败
- 修改h2脚本初始化加载, 去除测试用例注入.
- 新增注释其它

[v3.0.3] 2018.09.17

- 新增筛选查询字段方法
- fixed orderBy多入参的bug
- 新增 LogicDeleteByIdWithFill 组件
- fixed github issues/476 issues/473
- fixed github issues/360 gitee issues/IMIHN IM6GM
- 改进 allEq入参的value改用泛型
- fixed saveOrUpdateBatch使用BatchExecutor
- fixed 修正getOne获取多条数据为抛出异常
- 修正service 的getOne 方法
- 修正service 的个别方法为default方法
- 修复了page在set了desc下,sql有bug的问题
- 去除不再需要的方法
- 解决 generator 的 optional 的俩 jar 问题
- 重载 select(Predicate predicate)
- 其他优化

[v3.0.2] 2018.09.11

- 新增 wrapper 条件辅助类
- 新增 banner 属性控制是否打印
- 修复 gitee #IMMF4:批量插入(AR)事务无效
- fix: entity 无主键,生成 ew 的 where 条件的 bug
- 处理SqlRunner的sqlSession获取与释放
- 去除全局缓存sqlSession,增加Model,通用service层sqlSession释放
- ext: 抽象原生枚举处理类注册,方便扩展
- 优化扩展性其他

[v3.0.1] 2018.08.31

- 修复代码生成器设置表前缀异常
- 新增 EnumValue 注解方式扫描通用枚举处理
- 修复逻辑删除混用失败
- DB2 方言改进何鹏举优化
- 新增测试用例及其他

[v3.0-RELEASE] 2018.08.28 代号：超级棒棒糖

- 乐观锁 update(et,ew)方法 et带上 version 注解字段回写
- 优化改进优化代码生成器
- 包扫描为空时不抛出异常(枚举,别名)
- 去除 SqlSession
- 修改 issue 模板,完善注释
- 优化初始化过程,添加逻辑删除注解次数检测

- SQL检查允许跳过检查
- 支持达梦数据库
- 修改 code 为数值型严谨限制简化 api 层命名及初始值规则
- 初始化 SQL 解析移至 SqlInjector
- 其他代码优化

[v3.0-RC3] 2018.08.19 代号：超级棒棒糖 RC3

- 支持 TableField select 属性 false 排除默认注入大字段查询
- 解决 page 反序列化 pages 属性报错
- 合并2.x dataSource被代理处理
- 去除DbConfig.columnUnderline属性
- 过滤掉selectObjs查询结果集为空的情况
- baseMapper 的 insert 和 update 返回值不再使用包装类
- fixed Gitee issues/IM3NW
- 优化代码完善注释等

[v3.0-RC2] 2018.08.10 代号：超级棒棒糖 RC2

- 生成器加回 MODULE_NAME 开放配置 config
- 修复setting - defaultEnumTypeHandler属性配置无效
- 兼容 Spring boot 1.x 启动.
- 日常优化 , 测试用例 , 优化抛出异常的过程
- 新增 Gitee Github issue,pull_request模板
- 移除数据库关键字转义, 只支持注解模式转义
- 优化掉抛异常为使用 assert 或者 exceptionUtils
- 设置下划线转驼峰到 configuration 优化 ColumnUnderline
- 解决 page 序列化 asc desc 多态序列化异常
- 默认的 dbType 改为 other, 如果用户没有配置才会自动获取 dbType
- 优化,ColumnUnderline与MapUnderscoreToCamelCase意义相同
- fixed ILY8C 生成器指定 IdType 场景导入包
- 补充注释新增大量测试用例

[v3.0-RC1] 2018.08.01 代号：超级棒棒糖 RC1

- 优化工具类部分代码, 并修复一个在多线程环境下可能会引发死锁的BUG
- 新增断言类, 顺便修改几处地方的判断抛异常为使用断言
- 去掉多余的 “implements Serializable”
- 魔法值都改为全局常量模式
- 咩咩说了 MP 3.0 分页已经飘飘欲仙了, 不在需要迁就使用 PageHelper 模式
- issue #384 QueryWrapper 支持排除指定字段模式
- 全新 banner, 全新感觉
- 再优化一下抛异常的过程
- 修改 class 实例化对象的方式, 现在可以实例化私有 class

- 支持无配置可启动使用 Gitee issues/ILJQA
- 释放sqlSession,待优化 ActiveRecord单元测试
- 解决只调用 last 产生的 sql 会出的问题
- 修复Lambda首位属性为基类属性时错误.
- 增加泛型限制, 格式化下代码.
- 优化一下 AbstractWrapper 使用的 ISqlSegment
- 其他

[v3.0-RC] 2018.07.23 代号：超级棒棒糖 RC

- 优化 page 当 size 小于 0 自动调整为 list 模式
- 新增 攻击 SQL 阻断解析器
- 优化解析核心方法名, 新增 querywrapper lambda 转换参数测试
- 调整通用 service 层方法命名为阿里规范 (小白鼠, 对不起, 请唾弃我们吧! 然后修改下您的项目。)
- 代码生成器允许正则表达式匹配表名
- 乐观锁 回写更新后的version到实体
- Github #385:查询动态表名能利用Wrapper
- 修复 Gitee issues/ILEYD
- Page 的序列化接口挪到 IPage 接口
- 解决了 gamma 不能自动赋值 ID
- 代码改个常量引用优化

[v3.0-gamma] 2018.07.15 代号：超级棒棒糖 伽玛

- IPage 新增 listMode 集合模式
- fixd gitee issues/IL7W4
- fixed gitee issues/IL7W4
- 优化生成器包导入
- 解决 Page asc, desc 异常
- 逻辑删除无法 set where entity 一个参数并存逻辑
- 合并 PR 修改typeAliasesPackage扫描多维度
- 完善 3.0 测试用例
- 代码性能优化及其他

[v3.0-beta] 2018.07.07 代号：超级棒棒糖 贝塔

- 新增字段 LIKE 查询注入全局配置, 默认 true 开启
- 修改 dbtype 的 oracle db2 修改 CONCAT 方式
- 修正无论 update 的入参 updateWrapper 如何变化, 逻辑删除下依然存在限制条件
- 注释加上告警, 完善注释
- 修复 github issues/377 378 389
- 解决逻辑删除同时存在非逻辑删除逻辑
- 逻辑删除支持 delete set 其他字段, update 排除逻辑删除字段
- 支持 typeAliasesPackage 多项每项都有通配符 com.a.b..po, com.c..po

- 修复 gitee issues/IKJ48 IL0B2
- 其他完善

[v3.0-alpha] 2018.07.01 代号：超级棒棒糖

- 升级 JDK 8 + 优化性能 Wrapper 支持 lambda 语法
- 模块化 MP 合理的分配各个包结构
- 重构注入方法，支持任意方法精简注入模式
- 全局配置下划线转换消灭注入 AS 语句
- 改造 Wrapper 更改为 QueryWrapper UpdateWrapper
- 重构 分页插件 消灭固定分页模型，支持 Mapper 直接返回 IPage 接口
- 新增 Rest Api 通过 Controller 层
- 实体 String 类型字段默认使用 LIKE 查询 SelectOne 默认 LIMIT 1
- 辅助支持 selectMaps 新增 bean map 互转工具类
- 增加 db2 支持 starter 改为 Spring boot 2+ 支持
- 重构生成器提供自定义 DB 多种模板引擎支持
- 相关 BUG 修复

[v2.1.9] 2018.01.28 代号：怀念（纪念 2017 baomidou 组织小伙伴 MP 共同成长之路，奔向 2018 旺旺旺）

- page 分页新增控制是否优化 Count Sql 设置

```
1. // 不进行 count sql 优化
2. page.setOptimizeCountSql(false);
```

- 注入定义填充，支持sql注入器,主键生成器.
- fixed github issues/231
- fixed github issues/234
- 修改逻辑删除 selectByIds coll 问题
- fixed gitee issues/IHF7N
- fixed gitee issues/IHH83
- 兼容配置方式, 优先使用自定义注入.
- 其他优化

[v2.1.9-SNAPSHOT] 2018.01.16

- 调整 Gradle 依赖模式
- IdType 可选 ID_WORKER_STR `字符串类型` IdWorker.getIdStr() 字符串类型
- TableField 注解新增属性 `update` 预处理 set 字段自定义注入 fixed gitee IHART

```
1. 例如: @TableField(.., update="%s+1") 其中 %s 会填充为字段
2. 输出 SQL 为: update 表 set 字段=字段+1 where ...
```

1. 例如：`@TableField(..., update="now()")` 使用数据库时间
2. 输出 SQL 为：`update 表 set 字段=now() where ...`

- `TableField` 注解新增属性 `condition` 预处理 WHERE 实体条件自定义运算规则

1. `@TableField(condition = SqlCondition.LIKE)`
2. `private String name;`
3. 输出 SQL 为：`select 表 where name LIKE CONCAT('%',值,'%')`

- 添加 `spring-boot-starter` 模块内置 `jdbc mp` 包不需要单独引入 更舒服的使用 `boot`
- 添加对 SQL Server 视图生成的支持
- 允许字段策略独立设置，默认为 `naming` 策略

1. `strategy.setNaming(NamingStrategy.underline_to_camel);` // 表名生成策略
2. `strategy.setColumnNaming(NamingStrategy.underline_to_camel);` // 允许字段策略独立设置，默认为 `naming` 策略

- 代码生成器抽象 `AbstractTemplateEngine` 模板引擎抽象类，可自定义模板引擎，新增内置 `freemarker` 可选

1. // 选择 `freemarker` 引擎
2. `mpg.setTemplateEngine(new FreemarkerTemplateEngine());`

- 相关 SQL 解析如多租户可通过 `@SqlParser(filter=true)` 排除 SQL 解析

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

- 解决xml加载顺序问题，可随意引入其他 xml sql 片段
- 修复 `author` 带123的bug
- fix `#IGQE:Wrapper`为空,但是`page.getCondition()`不为空的情况, `Condition`无法传递问题
- fix `#IH6ED:Pagination` dubbo 排序等属性序列化不支持
- 判断`Wrapper`是否为空,使用`==`,避免被`equals`方法重载的影响
- 避免注入自定义基类
- 剥离 `sql` 单独提出至 `SqlUtils`
- 统一缩进编码风格
- 优化生成代码执行性能 [github issues/219](#)
- 优化 `sql` 解析过程
- fixed [gitee issues/IHCQB](#)
- `springboot-configuration-processor` 修改 `compileOnly`为`optional`
- 其他

[v2.1.8] 2018.01.02 代号：囍

- 修复代码生成器>字段前缀导致的bug
- 使用类全名替代手写的全名
- build修改
- 脚本警告,忽略目录
- 其他优化

[v2.1.8-SNAPSHOT] 2017.12.28 代号：翻车鱼（秋秋赐名）

- 返回Map自动下划线转驼峰
- kotlin entity 静态常量支持
- 优化 pagination 构造模式
- Merge pull request #201
- fix: selectByMap @alexqdjay
- 增加sqlRunner测试用例, 修复selectObjs只获取一个字段的bug
- 新增 BlobTypeHandler
- 去掉参数map的初始大小配置
- 增加.editorconfig, 模板空格问题修复.
- Hikaricp连接池无法打印sql
- 全局中去掉了路径, mapperLocations不可缺少了.
- k 神 全部覆盖测试用例

[v2.1.7] 2017.12.11 代号：清风徐来 ， 该版本号存在bug 请改为 2.1.8-SNAPSHOT +

- 枚举处理：基本类型，Number类型，String类型
- IGDRW:源码注释错误，容易给人误导 注释错误问题
- 炮灰 PR !42:添加分页构造方法重载 添加分页构造方法重载
- 代码生成 > oracle > 解决超出最大游标的问题
- fixed gitee IGNL9
- k 神 一大波 testcase 来袭
- 使用transient关键字去除Page中部分字段参与序列化
- 去除无效日志
- fix #IGI3H:selectBatchIds 参数改为Collection类型
- bugfix for logic delete sql injector
- 添加多个排序字段支持
- fixed github #185:2.0.2版本 自增主键 批量插入问题 pr
- 其他优化`

[v2.1.6] 2017.11.22 代号：小秋秋之吻

- 模块拆分为 support core generate 代码生成分离可选择依赖
- 解决 gitee issue IFX30 拆分 mybatis-plus-support 包支持
- 解决 gitee issue IGAPX 通用枚举 bigdecimal 类型映射

- druid补充, 填充字段修改
- 修复 kotlin 代码生成部分逻辑 Bug
- 合并 gitee pr 40 updateAllColumn****等方法排除fill = FieldFill.INSERT注释的字段 感谢 Elsf
- 构造模式设置 kotlin 修改
- Sql 工具类反射实例优化
- 其他优化

[v2.1.5] 2017.11.11 代号：离神

- 通用枚举 spring boot 兼容调整
- PostgreSQL 支持关键词非关键词转换问题
- Cat73 PR 稍微调整下自动生成的代码
- 支持 kotlin 代码生成
- bugfix for metaObj handler set val which not included in ...
- alibaba 规范调整
- 其他

[v2.1.3 - 2.1.4] 2017.10.15

- 新增通用枚举处理器, 参考 spring boot demno
- 优化 SQL 解析器
- 新增 schema 租户解析器待完善
- 其他优化

[v2.1.2] 2017.09.17 代号： X

- 修复代码生成器 Bug
- fixed gitee issues/IF2DY
- 修改 page 可链式操作
- 去掉转义 oracle
- fixed github issues/119
- fixed gitee issues/IF20I

[v2.1.1] 2017.09.12 代号：小锅盖

- 修改分页超过总记录数自动设置第一页 bug @wujing 感谢 pr
- fixed IEID6
- 升级 mybatis 3.4.5
- 升级生成器模板引擎 velocity 2.0
- 升级 jsqlparser 1.1
- 新增 SQL 解析链可动态扩展自定义 SQL 解析
- 新增 多租户 SQL 解析逻辑, 具体查看 spring boot 演示 demo
- jasonlong10 PR 性能分析拦截器 支持OraclePreparedStatementWrapper的情况打印 SQL

- fixed github issues/145
- fixed gitee issue/IF10F
- add sqlSelect("distinct test_type") test case
- 添加填充生成器遗漏 TableField 导入类
- fixed github issues/MYSQL表名含有保留字代码生成时报错 #124:字段全为 大写 下划线命名支持
- fixed github issues/134
- PostgreSQL 代码生成支持指定 schema 表字段按照默认排序
- 其他优化调整

[v2.1.0] 2017.08.01 代号：小秋秋

主体功能

- 批量sqlSession没有关闭问题修复
- 处理sql格式化报错问题, 添加填充信息

. 91:关于insertBatch在大数据量的时候优化 github

- 新增 uuid 主键测试用例
- 修复BUG自动填充会覆盖之前的值
- 升级pom依赖, spring-test作用域test
- 更改sqlServer驱动, 去掉乐观锁不需要的string类型测试

. 86:关于plus的底层映射设计问题 github issue

- SqlHelper处理Wrapper为空, 但是page.getCondition()不为空的情况
- Merge pull request !33:生成实体增加字段排序 from 老千/master
- 解决使用代理对象导致无法获取实例缓存信息
- 解决布尔类型is开头生成sql错误问题
- DBType设置错误
- fix #351:DB2Dialect返回NULL
- fix #356:自动代码生成的Boolean类型的get方法不对
- fix #353:代码生成@TableLogic问题
- 新增 PostgreSqlInjector 自动注入器, 处理字段大小写敏感, 自动双引号转义。
- 仓库地址与用户信息使用自定义传入。
- fix #357:代码生成@TableLogic引入包Bug
- Sequence 新增 mac 判断, 分页 pageHelper 模式新增 freeTotal() 方法

. 95:分页插件两个建议 Github, selectItems contains #{ } \${{}},

- 添加 `Wrapper#setSqlSelect(String... columns)` 方法, 方便通过自动生成的实体...
- fixed github 116 issue
- fixed osgit IE436 IDVPZ IDTZH

代码生成

- 修改实体生成模板
- 修复自动填充代码生成错误
- 新增 postgresql schemaname 生成器支持
- 调整序列化导入问题
- 其他

[v2.1-gamma] 2017.06.29

主体功能

- 修正之前sqlserver自动获取类型错误问题
- 修复用户无法自定义分页数据库方言问题

代码生成

- 完善了自动填充代码生成
- 修复postgresql生成重复字段问题

上个版本 (2.0.9) 升级导致的问题

- 修复实体主键不在第一位无法读取的问题
- 修复在自定义insert操作时报 `Insert not found et` 异常, 见#331
- 修复Sql生成错误问题(普通注入Group, Having, Order)
- 修复逻辑删除生成Sql顺序错误
- 感谢各路小伙伴及时反馈的问题, 上个版本给大家带来的问题深感抱歉

Mybatis-Plus-Boot-Start [1.0.4]

主体变动

- 去除Mybatis-plus直接依赖
- 去除SpringBoot jdbc-starter直接依赖

[v2.0.9] 2017.06.26 代号: K 神

Mybaitis-Plus #####主体功能

- 修正乐观锁和逻辑删除冲突问题
- 处理在生成注入SQL时之前没有考虑到存在且打开下划线配置情况

- 修复EntityWrapper继承关系问题
- Wrapper添加条件判断
- 性能分析插件支持记录日志提示
- Wrapper重写了toString方式, 解决之前Debug时显示为null给用户造成错觉
- 处理Sequence非毫秒内并发偶数居多问题
- 忽略策略优化处理更改了注解的属性
- 注入Sql的方式优化, 去除之前XML注入方式
- 处理逻辑删除出现2个Where的问题
- 添加其他数据库序列的实现方式, 并开放出接口给用户自行扩展
- 乐观锁优化调整
- 优化Wrapper中Where AND OR 去除之前基于反射方式实现, 提高代码运行效率
- 处理不添加mybatis-config.xml主键无法填充问题
- MybatisPlus添加支持gradle构建方式
- Wrapper 添加 `and()` `or()` 方法
- 优化GlobalConfiguration, 抽离出GlobalConfigUtils减少耦合
- 修复SqlServer2008与SqlServer2005分页问题
- 新增自动识别数据库, 减少用户显式配置
- 优化分页插件减少用户显示配置属性
- 自动填充字段问题解决
- 新增PageHelper, 获取当前线程来管理分页(之前老用户最好不要使用, 改方式只用户适用MybatisPageHelper用户习惯)
- 大幅度的添加测试用例(感谢K神支持)
- 代码的其他优化
- 添加了JSqlparser的依赖以后不用手动去添加该Jar包

代码生成

- 支持逻辑删除方式生成
- 支持乐观锁方式生成
- 修复生成器不能识别sqlServer的自增主键代码生成器不能识别SqlServer自增主键的问题
- 支持Lombok方式生成
- 支持构建模式方式生成
- 添加Clob和Blob类型转换
- 修复Oracle的Number类型字段转换错误问题

Mybatis-Plus-Boot-Start [1.0.2] 代号：清风 ####主体功能

- 处理AR模式devtool替换数据源失效问题
- 添加逻辑删除支持
- 添加序列支持

[v2.0.8] 2017.05.15

- Wrapper添加设置对象sqlSelect
- 兼容无注解情况
- 乐观锁去除默认short实现, 优化绑定注册器在扫描阶段绑定. 测试改为h2环境.

- 优化热加载, 去除mapper路径配置.
- 减少刷新Mapper配置
- 修复tableFiled value 为空情况, 开启下划线命名
- sequence 升级提示
- 开放表信息、预留子类重写
- 修改Idwork测试
- 支持 devtools
- fixed 259 支持 xml resultMap 公共字段生成
- fixed pulls 28 支持属性重载

[v2.0.6 2.0.7] 2017.04.20

- 新增 逻辑删除
- 新增 Oracle Sequence
- 新增 jdk1.8 时间类型
- 完善支持乐观锁
- 完善字段填充器, 支持更新填充
- 升级 mybatis 依赖为 3.4.4
- 代码调整优化, 支持 wrapper limit 等逻辑
- 修复 Id 策略 auto bug , 生成器 bug 其他

[v2.0.5] 2017.03.25

- 修复分页连接池没有关闭的bug
- issues fixed 217
- IMetaObjectHandler当主键类型是AUTO或者INPUT的时候不起效的bug
- 修复 like 占位符问题
- 生成代码的时候如果目录不存在则新建

[v2.0.3 - v2.0.4] 2017.03.22

- 优化Wrapper代码结构
- 优化原有数据库连接获取
- 解决Page初始化问题(之前只能通过构造方法生效, 现在可以通过setget也可以生效)
- 支持乐观锁插件
- 改造Wrapper让JDBC底层来处理参数, 更好的与PreparedStatement结合
- 修复相关错误日志提示级别
- Wrapper开放isWhere方法, 现在可以自定义是否拼接"WHERE"
- JDK版本向下兼容, 之前相关代码用到了1.7新特性, 当前版本解除
- sqlserver生成bug修复以及代码优化
- 优化MybatisPlus, SqlSession获取
- 解决未配置切点的情况下获取的sqlSession提交不属于当前事务的问题以及多个sqlSession造成的事务问题
- 增强执行sql类, sqlRunner
- Model添加序列化ID, 避免以后在修改Model后没有设置序列号ID时序列号ID可以会变动的情况

- 添加重写默认BaseMapper测试用例
- 感谢各路小伙伴提问的好的建议以及贡献代码,就不一一点名了

[v2.0.2] 2017.02.13

- 修复全局配置不起作用 2.0.1 逻辑
- 去除byId强制配置类型
- Wrapper Page 等程序优化
- 优化AR模式自动关闭数据库连接(之前需要手动设置事务)
- 优化代码生成器,下划线名称注解不处理驼峰,支持自定义更多的模板例如 jsp html 等
- 新增 service 层测试
- sql日志记录整合至性能分析插件.
- 处理多数据源分页插件支持多重数据库

[v2.0.1] 2017.01.15

- 解决EntityWrapper对布尔类型构造sql语句错误
- 全局配置初始化日志提示调整
- Mybatis依赖升级至3.4.2,Mybatis-Spring依赖升级至1.3.1
- Service中补充方法(selectObjs,selectMaps)
- 解决selectCount数据库返回null报错问题
- 支持PostgreSql代码生成
- 拓展支持外部提供转义字符以及关键字列表
- 开放数据库表无主键依然注入MP的CRUD(无主键不能使用MP的xxById方法)
- 解决EntityWrapper拼接SQL时,首次调用OR方法不起作用的问题
- sqlServer代码生成(基于2008版本)
- 解决生成代码时未导入BigDecimal问题.
- 释放自动读取数据库时的数据库连接
- 优化全局校验机制(机制为EMPTY增加忽略Date类型)
- 优化注入,避免扫描到BaseMapper
- 优化注入,去除多余注入方法
- SQLlikeType改名为SqlLike
- 解决热加载关联查询错误问题
- SqlQuery改名为SqlRunner
- 优化完善代码生成器
- 修复代码生成器未导入@tableName
- 全局配置需要手动添加MP的默认注入类,更改为自动注入简化配置
- Wrapper增加ne方法
- 修复Mybatis动态参数无法生成totalCount问题
- 代码结构优化,生成器模板优化
- 解决issues[138,140,142,148,151,152,153,156,157],具体请查看里程碑[mybatis-plus 2.0.1 计划](#)中所有issues

[v2.0.0] 2016.12.11

- 支持全局大写命名策略
- 自动分页Count语句优化
- 优化现有全局配置策略
- 优化全局验证策略
- 优化代码生成器(之前硬编码,现使用模板形式)
- 优化注入通用方法ByMap逻辑
- 添加自动选择数据库类型
- 改善SqlExplainInterceptor(自行判断MySQL版本不支持该拦截器则直接放行(版本过低小于5.6.3))
- 修复部分特殊字符多次转义的问题
- 优化现有EntityWrapper添加Wrapper父类以及Condition链式查询
- Wrapper类使LIKE方法兼容多种数据库
- 优化日志使用原生Mybatis自带的日志输出提示信息
- 修复使用缓存导致使用分页无法计算Count值
- 修复PerformanceInterceptor替换 `?` 导致打印SQL不准确问题,并添加格式化SQL选项
- 添加多种数据库支持,请查看DBType
- 添加字符串类型字段非空校验策略(字符串类型自动判断非空以及非空字符串)
- Wrapper添加类似QBC查询(eq、gt、lt等等)
- 支持AR模式(需继承Model)
- 合并所有Selective通用方法(例如:去除之前的insert方法并把之前的insetSelective改名为insert)
- 解决sql剥离器会去除 `--` 的情况
- 支持MySQL关键词,自动转义
- 精简底层Service、Mapper继承结构
- 不喜欢在XML中写SQL的福音,新增执行SQL方式,具体请查看SqlQuery
- 优化代码结构
- 解决
issues[95, 96, 98, 100, 103, 104, 108, 114, 119, 121, 123, 124, 125, 126, 127, 128, 131, 133, 134, 135], 具体请查看里程碑[mybatis-plus 2.0 计划](#)中所有issues

[v1.4.9] 2016.10.28

- ServiceImpl去除@Transactional注解、去除Slf4j依赖
- 解决使用EntityWrapper查询时,参数为特殊字符时,存在sql注入问题
- 调整Mybatis驼峰配置顺序 MybatisPlus > Mybatis
- 优化分页插件并修复分页溢出设置不起作用问题
- 去除DBKeywordsProcessor,添加MySQL自动转义关键词
- 代码生成器新增支持TEXT、TIME、TIMESTAMP类型生成
- 新增批量插入方法
- 代码生成器新增Controller层代码生成
- 调整EntityWrapper类部分List入参为Collection
- 代码生成器优化支持 resultMap

[v1.4.8] 2016.10.12

- insertOrUpdate增加主键空字符串判断
- 支持Mybatis原生驼峰配置 mapUnderscoreToCamelCase 开关设置

- 支持 TableField FieldStrategy 注解全局配置
- SelectOne、SelectCount方法支持EntityWrapper方式
- oracle 代码生成器支持 Integer Long Double 类型区分
- 修复INPUT主键策略InsertOrUpdate方法Bug
- EntityWrapper IN 添加可变数组支持
- 基础Mapper、Service通用方法PK参数类型更改至Serializable
- 当selectOne结果集不唯一时, 添加警告提示(需开启日志warn模式)
- BaseService添加logger, 子类直接调用logger不用重新定义(需slf4j依赖)

[v1.4.7] 2016.09.27

- 主键注解 I 改为 PK 方便理解, 去掉 mapper 注解
- 性能分析插件, 特殊处理 \$ 符内容
- 添加自动提交事务说明, 新增事务测试
- 支持 resultMap 实体结果集映射
- 增加#TableField(e1 = "")表达式, 当该Field为对象时, 可使用#{对象.属性}来映射到数据表、及测试
- 新增 typeHandler 级联查询支持
- 新增验证字段策略枚举类
- 代码生成器支持实体构建者模型设置
- 代码生成器新增实体常量生成支持
- CRUD 新增 insertOrUpdate 方法
- 解决MessageFormat.format格式化数字类型sql错误
- EntityWrapper添加 EXISTS、IN、BETWEEN AND(感谢D.Yang提出)方法支持
- 支持 mysql5.7+ json enum 类型, 代码生成
- 支持无XML依然注入CRUD方法
- 修改Mybatis原生配置文件加载顺序

[v1.4.6] 2016.09.05

- 新增无 @TableId 注解跳过注入SQL
- 支持非表映射对象插入不执行填充
- xxxByMap 支持 null 查询

[v1.4.5] 2016.08.28

- 新增 XML 修改自动热加载功能
- 添加自动处理EntityWrapper方法中的MessageFormat Params类型为字符串的参数
- 新增表公共字段自动填充功能

[v1.4.4] 2016.08.25

- entitywrapper所有条件类方法支持传入null参数, 该条件不会附加到SQL语句中
- TSQlPlus更名为TSqlPlus与整体命名保持一致。
- 修复mysql关键字bug—将关键字映射转换加上``符号, 增加xml文件生成时可自定义文件后缀名

- 关闭资源前增加非空判断, 避免错误sql引起的空指针错误, 增加选择 `current>pages` 判断
- TSQL 相关类实现序列化支持 `dubbo`
- 增加 `mybatis` 自动热加载插件
- 支持数据库 `order key` 等关键词转义 `curd` 操作

[v1.4.3] 2016.08.23

- 优化 `Sequence` 兼容无法获取 `mac` 情况
- 兼容用户设置 `ID` 空字符串, 自动填充
- 纯大写命名, 转为小写属性
- 修改`EntityWrapper`符合T-SQL语法标准的条件进行方法封装定义
- 升级 1.4.3 测试传递依赖

[v1.4.0] 2016.08.17

- 增加自定义 `select` 结果集, 优化 `page` 分页
- 未考虑 函数, 去掉 `field` 优化
- 新增 `delete update` 全表操作禁止执行拦截器

[v1.3.9] 2016.08.09

- 修复 `bug`
- 解决插入 `map` 异常
- 插入 `map` 不处理, 原样返回
- 优化 `IdWorker` 生成器
- 支持自定义 `LanguageDriver`
- 支持代码生成自定义类名
- 升级 `mybatis 3.4.1` 依赖

[v1.3.6] 2016.07.28

- 支持全局表字段下划线命名设置
- 增加自定义 注入 `sql` 方法
- 优化分页总记录数为0不执行列表查询逻辑
- 自动生成 `xml` 基础字段增加 `AS` 处理
- 支持字段子查询

[v1.3.5] 2016.07.24

- 升级 1.3.5 支持全局表字段下划线命名设置
- 添加发现设置多个主键注解抛出异常
- 添加无主键主键启动异常
- 去掉重置 `getDefaultScriptingLanguageInstance`

- 修改歧义重载方法

[v1.3.3] 2016.07.15

- 处理 SimpleDateFormat 非现场安全问题
- 修改 oracle 分页 bug 修复
- oracle TIMESTAMP 生成支持 bug 修复

[v1.3.2] 2016.07.12

- service 暴露 sqlSegment 的方法调用
- 新增 sql 执行性能分析 plugins
- 新增 deleteByMap , selectByMap

[v1.3.0] 2016.07.07

- 支持 like 比较等查询 sqlSegment 实现
- 支持 typeAliasesPackage 通配符扫描, 无 count 分页查询
- mybatis mapper 方法调用执行原理测试
- 添加 IOC 演示用例

[v1.2.17] 2016.06.15

- 优化 代码生成器 感谢 yanghu pull request
- 调整 sql 加载顺序 xmlSql > curdSql
- 支持 CURD 二级缓存
- 增加缓存测试, 及特殊字符测试

[v1.2.15] 2016.04.27

- 新增 支持oracle 自动代码生成, 测试 功能
- 新增 UUID 策略
- 演示demo 点击 spring-wind
- 新增支持单表 count 查询

[v1.2.12] 2016.04.22

- 添加 service 层支持泛型 id 支持, 自动生成代码优化
- 升级 mybatis 为 3.4.0 , mybatis-spring 为 1.3.0

[v1.2.11] 2016.04.18

- 新增批量更新，支持 oracle 批量操作
- 去掉，移植至 spring-wind 的文档
- 支持 jdk1.5 修改 param 描述
- 添加数据库类型

[v1.2.9] 2016.04.10

- EntityWrapper 新增无 order by 构造方法
- MailHelper 重载 sendMail 方法
- 新增 String 主键ID 支持 CommonMapper
- 原来方法 selectList 分离为 selectList , selectPage 两个方法
- 优化代码生成器，添加文档说明、其他

[v1.2.8] 2016.04.02

- 优化生成代码处理大写字段，支持自动生成 entity mapper service 文件
- 优化分页 index 超出逻辑，新增 5 个 CRUD 操作方法
- 开放模板引擎 getHtmltext 方法
- 优化邮件发送配置添加说明文档
- 添加文档说明、其他

[v1.2.6] 2016.03.29

- 优化代码 service 层封装，抽离 list 、 page 方法
- 优化分页 count sql 语句
- 改进 mail 工具类
- 完善 framework 对 spring 框架的支持
- 添加文档说明、其他

[v1.2.5] 2016.03.25

- 独立支持id泛型的 baseMapper
- 更完善的自动生成工具
- 支持实体封装排序
- 分页插件完善
- 抽离 service 主键泛型支持

[v1.2.2] 2016.03.14

- 注解 ID 区分 AUTO 数据库自增，ID_WORKER 自动填充自定义自增ID ， INPUT 手动输入 。
- 优化代码及自动生成器功能。
- 其他