

快速使用

注意: 文章使用版本 - [官网地址](#)

```
<dependency>
  <groupId>com.baomidou</groupId>
  <artifactId>mybatis-plus-boot-starter</artifactId>
  <version>3.4.0</version>
</dependency>
```

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。

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

支持数据库支持数据库

- mysql、mariadb、oracle、db2、h2、hsql、sqlite、postgresql、sqlserver、presto、Gauss、Firebird
- Phoenix、clickhouse、Sybase ASE、OceanBase、达梦数据库、虚谷数据库、人大金仓数据库、南大通用数据库、

快速入门

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：

```
DROP TABLE IF EXISTS user;

CREATE TABLE user
(
    id BIGINT(20) NOT NULL COMMENT '主键ID',
    name VARCHAR(30) NULL DEFAULT NULL COMMENT '姓名',
    age INT(11) NULL DEFAULT NULL COMMENT '年龄',
    email VARCHAR(50) NULL DEFAULT NULL COMMENT '邮箱',
    PRIMARY KEY (id)
);
```

其对应的数据库 Data 脚本如下：

```
DELETE FROM user;

INSERT INTO user (id, name, age, email) VALUES
(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');
```

添加依赖

引入 Spring Boot Starter 父工程：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.4.RELEASE</version>
    <relativePath/>
</parent>
```

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
```

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
  <groupId>com.baomidou</groupId>
  <artifactId>mybatis-plus-boot-starter</artifactId>
  <version>3.4.0</version>
</dependency>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
</dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>

```

配置

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

```

# DataSource Config
spring:
  datasource
    username: root
    password: 123456
    # 我是用的 mysql 8版本 向下支持 mysql 5 所以如果是 mysql 5 也可以使用一样的 url \
    driver
      url: jdbc:mysql://localhost:3306/mybatis_plus?
      useSSL=false&useUnicode=true&characterEncoding=utf-8&serverTimezone=GMT%2B8
      driver-class-name: com.mysql.cj.jdbc.Driver

```

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

```

// com.lomonkey.mapper 为 mapper文件位置
@MapperScan("com.lomonkey.mapper")
@SpringBootApplication
public class LeanApplication {
    public static void main(String[] args) {
        SpringApplication.run(LeanApplication.class);
    }
}

```

编码

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

默认配置：

表名称: 实体类名称 小写,这里默认查询 `user` 表

主键: 默认是字段为 `id` 的属性, 生成策略为 `@TableName` 的 `@IdType` 里面的 `ASSIGN_ID` 可以为 `Number` 或者 `String` 类型

```
@Data
public class User {
    private Long id;
    private String name;
    private Integer age;
    private String email;
}
```

扩展: 也可以使用 `table-prefix` 来指定定表的前缀

```
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
  global-config:
    db-config:
      table-prefix: tb_
```

编写Mapper类 `UserMapper.java`

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

开始使用

添加测试类, 进行功能测试:

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MapperTest {
    @Resource
    private UserMapper userMapper;

    @Test
    public void test() {
        // selectList 参数为 wrapper 是一个复杂查询的构造器,传入null.默认查询所有
        List<TbUser> tbUsers = userMapper.selectList(null);
        tbUsers.forEach(System.out::println);
    }
}
```

```
2020-09-28 09:54:57.617 INFO 14080 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
TbUser(id=1, name=Jone, age=18, email=test1@baomidou.com)
TbUser(id=2, name=Jack, age=20, email=test2@baomidou.com)
TbUser(id=3, name=Tom, age=28, email=test3@baomidou.com)
TbUser(id=4, name=Sandy, age=21, email=test4@baomidou.com)
TbUser(id=5, name=Billie, age=24, email=test5@baomidou.com)
2020-09-28 09:54:57.656 INFO 14080 --- [extShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Shutdown initiated...
2020-09-28 09:54:57.662 INFO 14080 --- [extShutdownHook] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Shutdown completed.
```

配置日志

```
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl # 这里使用控制台输出，其他
    需引入依赖
```

```
JDBC Connection [HikariProxyConnection@936261188 wrapping com.mysql.cj.jdbc.ConnectionImpl@4b869331] will not be managed by Spring
==> Preparing: SELECT id,name,age,email FROM user
==> Parameters:
<== Columns: id, name, age, email
<== Row: 1, Jone, 18, test1@baomidou.com
<== Row: 2, Jack, 20, test2@baomidou.com
<== Row: 3, Tom, 28, test3@baomidou.com
<== Row: 4, Sandy, 21, test4@baomidou.com
<== Row: 5, Billie, 24, test5@baomidou.com
<== Total: 5
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@46d8f407]
User(id=1, name=Jone, age=18, email=test1@baomidou.com)
User(id=2, name=Jack, age=20, email=test2@baomidou.com)
User(id=3, name=Tom, age=28, email=test3@baomidou.com)
User(id=4, name=Sandy, age=21, email=test4@baomidou.com)
User(id=5, name=Billie, age=24, email=test5@baomidou.com)
```

CRUD操作

增加操作

```
@Test
public void add() {
    User user = new User();
    user.setAge(20);
    user.setEmail("lomonkey@aliyun.com");
    user.setName("Lomonkey");
    int result = userMapper.insert(user);
    System.out.println(user);
    System.out.println("result = " + result);
}
```

```
==> Preparing: INSERT INTO user ( id, name, age, email ) VALUES ( ?, ?, ?, ? )
==> Parameters: 1310419023559942145(Long), Lomonkey(String), 20(Integer), lomonkey@aliyun.com(String)
<== Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@2b8bd14b]
User(id=1310419023559942145, name=Lomonkey, age=20, email=lomonkey@aliyun.com)
result = 1
```

注意:

1. 默认添加的 id为Long类型的数字`1310402418784423937(Long)`默认使用的[@TableId] (#table-name)中[IdType] (#id-type)的`ASSIGN_ID`
2. 在添加完成之后看到 user 会回显增加的用户id
3. 成功返回 1

主键生成策略

主键附录-官网地址

默认使用 - ASSIGN_ID - 雪花算法

值	描述
ASSIGN_ID	分配ID(主键类型为 Number(Long和Integer) 或 String)(since 3.3.0),使用接口 IdentifierGenerator 的方法 nextId (默认实现类为 DefaultIdentifierGenerator 雪花算法)

主键自增 - AUTO

值	描述
AUTO	数据库ID自增

1. 注意：数据库要支持自增

2. 名

	类型	长度	小数点	不是 null	虚拟	键	注释
id	bigint	20	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	 1	主键ID
name	varchar	30	0	<input type="checkbox"/>	<input type="checkbox"/>		姓名
age	int	11	0	<input type="checkbox"/>	<input type="checkbox"/>		年龄
email	varchar	50	0	<input type="checkbox"/>	<input type="checkbox"/>		邮箱

默认:

☒ 自动递增

☐ 无符号

☐ 填充零

2. 在主键上添加注解 @TableId(type=IdType.AUTO)

3. 再次插入即可

其他策略请看[官网](#)或者[附录注解](#)

更新操作

```
@Test
public void updateTest() {
    User user = new User();
    user.setId(1L);
    user.setName("updateName");
    int update = userMapper.updateById(user);
    System.out.println(user);
    System.out.println("update = " + update);
}
```

```
=> Preparing: UPDATE user SET name=? WHERE id=?
=> Parameters: updateName(String), 1(Long)
<== Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@29852487]
User(id=1, name=updateName, age=null, email=null)
update = 1
```

注意:

1. 根据 `set方法` 指定的属性进行`修改`, 其他属性并不会重置为`null`

2. 修改成功返回 1

删除操作

```
@Test
public void deleteTest() {
    int delete = userMapper.deleteById(1310414914635251715L);
    System.out.println("delete = " + delete);
}
```

```
==> Preparing: DELETE FROM user WHERE id=?
==> Parameters: 1310414914635251715(Long)
<== Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@2b8bd14b]
delete = 1
```

注意:

1. 删除成功返回 `1`

```
/**
 * 根据 map 中条件删除
 */
@Test
public void deleteByMap() {
    HashMap<String, Object> map = new HashMap<>();
    map.put("name", "Lomonkey");
    int result = userMapper.deleteByMap(map);
    System.out.println(result);
}
```

```
==> Preparing: DELETE FROM user WHERE name = ?
==> Parameters: Lomonkey(String)
<== Updates: 3
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@5f13be1]
result = 3
```

注意:

1. 删除多少返回 `result` 为 多少。这里删除了 3 条数据，故返回3

```
@Test
public void deleteByWrapper() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    // 姓名 %Lomonkey% 的
    wrapper.like("name", "Lomonkey");
    // 邮箱 %lv 的
    wrapper.likeLeft("email", "lv");
    // id >= 30
    wrapper.ge("id", 30);
    // age 在 30 到 50
    wrapper.between("age", 30, 50);
    userMapper.delete(wrapper);
}
```

// 链式编程

```

@Test
public void deleteBywrapper() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    // 姓名 %Lomonkey% 的
    wrapper
        .like("name", "Lomonkey")
        // 邮箱 %lv 的
        .likeLeft("email", "lv")
        // id >= 30
        .ge("id", 30)
        // age 在 30-50 之间
        .between("age", 30, 50);
    userMapper.delete(wrapper);
}

```

```

JDBC Connection [HikariProxyConnection@302059473 wrapping com.mysql.cj.jdbc.ConnectionImpl@7c84195] will not be managed by Spring
==> Preparing: DELETE FROM user WHERE (name LIKE ? AND email LIKE ? AND id >= ? AND age BETWEEN ? AND ?)
==> Parameters: %Lomonkey%(String), %lv(String), 30(Integer), 30(Integer), 50(Integer)
<== Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@55f8669d]

```

其他删除 API

`wrapper` 表示复杂查询,可以构建复杂的

```

delete(wrapper<User> wrapper) int
deleteBatchIds(Collection<? extends Serializable> idList) int
deleteByMap(Map<String, Object> columnMap) int
deleteById(Serializable id) int

```

Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards [Next Tip](#)

查询操作

```

selectList(wrapper<User> queryWrapper) List<User>
selectBatchIds(Collection<? extends Serializable> idList) List<User>
selectById(Serializable id) User
selectByMap(Map<String, Object> columnMap) List<User>
selectCount(wrapper<User> queryWrapper) Integer
selectMaps(wrapper<User> queryWrapper) List<Map<String, Object>>
selectMapsPage(E page, wrapper<User> queryWrapper) E
selectObjs(wrapper<User> queryWrapper) List<Object>
selectOne(wrapper<User> queryWrapper) User
selectPage(E page, wrapper<User> queryWrapper) E

```

Press Enter to insert. Tab to replace [Next Tip](#)

SelectList


```
// selectList
@Test
public void selectList() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.notLike("name", "Jack")
        .in("id", Arrays.asList(1, 2, 3, 4, 5))
        .orderByAsc("id")
        // 结果只显示 name 和 email
        .select("name", "email");
    List<User> users = userMapper.selectList(wrapper);
    users.forEach(System.out::println);
}
```

```
JDBC Connection [HikariProxyConnection@1069531012 wrapping com.mysql.cj.jdbc.ConnectionImpl@23f72d88] will not be managed by Spring
==> Preparing: SELECT name,email FROM user WHERE (name NOT LIKE ? AND id IN (?, ?, ?, ?, ?)) ORDER BY id ASC
==> Parameters: %Jack%(String), 1(Integer), 2(Integer), 3(Integer), 4(Integer), 5(Integer)
<== Columns: name, email
<== Row: Lomonkey, test1@baomidou.com
<== Row: Tom, test3@baomidou.com
<== Row: Sandy, test4@baomidou.com
<== Row: Billie, test5@baomidou.com
<== Total: 4
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@19ccca5]
User(id=null, name=Lomonkey, age=null, email=test1@baomidou.com)
User(id=null, name=Tom, age=null, email=test3@baomidou.com)
User(id=null, name=Sandy, age=null, email=test4@baomidou.com)
User(id=null, name=Billie, age=null, email=test5@baomidou.com)
```

SelectBatchIds

```
@Test
public void selectBatchIds() {
    List<User> users = userMapper.selectBatchIds(Arrays.asList(1, 2, 3));
    users.forEach(System.out::println);
}
```

```
==> Preparing: SELECT id,name,age,email FROM user WHERE id IN ( ?, ?, ? )
==> Parameters: 1(Integer), 2(Integer), 3(Integer)
<== Columns: id, name, age, email
<== Row: 1, Lomonkey, 55, test1@baomidou.com
<== Row: 2, Jack, 20, test2@baomidou.com
<== Row: 3, Tom, 30, test3@baomidou.com
<== Total: 3
```

SelectById

```
@Test
public void selectById() {
    User user = userMapper.selectById(1L);
    System.out.println(user);
}
```

```
==> Preparing: SELECT id,name,age,email FROM user WHERE id=?
==> Parameters: 1(Long)
<== Columns: id, name, age, email
<== Row: 1, Lomonkey, 55, test1@baomidou.com
<== Total: 1
```

SelectByMap

```
@Test
public void selectByMap() {
    HashMap<String, Object> map = new HashMap<>();
    map.put("name", "Jack");
    List<User> users = userMapper.selectByMap(map);
    users.forEach(System.out::println);
}
```

```
=> Preparing: SELECT id,name,age,email FROM user WHERE name = ?
=> Parameters: Jack(String)
<== Columns: id, name, age, email
<== Row: 2, Jack, 20, test2@baomidou.com
<== Total: 1
```

SelectCount

```
@Test
public void selectCount() {
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.in("id", Arrays.asList(1, 2, 3, 4, 5));
    Integer integer = userMapper.selectCount(wrapper);
    System.out.println("integer = " + integer);
}
```

```
=> Preparing: SELECT COUNT( 1 ) FROM user WHERE (id IN (?, ?, ?, ?, ?))
=> Parameters: 1(Integer), 2(Integer), 3(Integer), 4(Integer), 5(Integer)
<== Columns: COUNT( 1 )
<== Row: 5
<== Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@54f66455]
integer = 5
```

SelectPage(分页)

分页查询

步骤:

1. 创建配置类, 注入Bean
2. 使用 IPage的子类 Page

官网中:

```
// 配置类
// 如果这里使用 @MapperScan 则 在启动类 LeanApplication上的@MapperScan可以不用写
@MapperScan("com.lomoney.mapper")
@Configuration
public class MybatisPlusConfig {

    /**
     * 新的分页插件,一缓和二缓遵循mybatis的规则,需要设置
     * MybatisConfiguration#useDeprecatedExecutor = false 避免缓存出现问题(该属性会在旧插件移除后一同移除)
     */
}
```

```

@Bean
public MybatisPlusInterceptor mybatisPlusInterceptor() {
    MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
    interceptor.addInnerInterceptor(new
PaginationInnerInterceptor(DbType.H2));
    return interceptor;
}

@Bean
public ConfigurationCustomizer configurationCustomizer() {
    return configuration -> configuration.setUseDeprecatedExecutor(false);
}
}

```

```

@Test
public void selectPage() {
    // 复杂查询
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.like("name", "Lomonkey");

    // 配置分页
    Page<User> page = new Page<>(1, 3);

    // 查询
    userMapper.selectPage(page, wrapper);
    System.out.println(page);
}

```

```

JDBC Connection [HikariProxyConnection@926905424 wrapping com.mysql.cj.jdbc.ConnectionImpl@d74bac4] will not be managed by Spring
==> Preparing: SELECT COUNT(1) FROM user WHERE (name LIKE ?)
==> Parameters: %Lomonkey%(String)
<== Columns: COUNT(1)
<== Row: 10
<== Total: 1
==> Preparing: SELECT id,name,age,email FROM user WHERE (name LIKE ?) LIMIT ?
==> Parameters: %Lomonkey%(String), 3(Long)
<== Columns: id, name, age, email
<== Row: 1, Lomonkey, 55, test1@baomidou.com
<== Row: 1310419023559942145, Lomonkey2, 20, lomonkey@aliyun.com
<== Row: 1310419023559942149, Lomonkey, 50, null
<== Total: 3
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@6a2eea2a]
com.baomidou.mybatisplus.extension.plugins.pagination.Page@7ff7af971

```

自动填充

创建时间\修改时间 一般是自动完。

[阿里巴巴开发\(嵩山\)手册](#)规定：必备三个字段 `create_time` 和 `update_time` 以及 `id`

9. 【强制】表必备三字段：id, create_time, update_time。

说明：其中 id 必为主键，类型为 bigint unsigned、单表时自增、步长为 1。create_time, update_time 的类型均为 datetime 类型，前者现在时表示主动式创建，后者过去分词表示被动式更新。

1. 修改表结构 增加两个字段 `create_time` 和 `update_time`

名	类型	长度	小数点	不是 null	虚拟	键	注释
id	bigint	20	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1	主键ID
name	varchar	30	0	<input type="checkbox"/>	<input type="checkbox"/>		姓名
age	int	11	0	<input type="checkbox"/>	<input type="checkbox"/>		年龄
email	varchar	50	0	<input type="checkbox"/>	<input type="checkbox"/>		邮箱
create_time	datetime	0	0	<input type="checkbox"/>	<input type="checkbox"/>		
update_time	datetime	0	0	<input type="checkbox"/>	<input type="checkbox"/>		

2. 修改实体类增加 `createTime` 和 `updateTime` 两个字段

```
@Data
public class User {
    private Long id;
    private String name;
    private Integer age;
    private String email;

    private Date createTime;
    private Date updateTime;
}
```

数据库级别

工作中一般不允许修改数据库, 顾该方法不推荐使用,但是也是一种处理方式

!

字段	索引	外键	触发器	选项	注释	SQL 预览					
名					类型	长度	小数点	不是 null	虚拟	键	注释
id					bigint	20	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	1	主键ID
name					varchar	30	0	<input type="checkbox"/>	<input type="checkbox"/>		姓名
age					int	11	0	<input type="checkbox"/>	<input type="checkbox"/>		年龄
email					varchar	50	0	<input type="checkbox"/>	<input type="checkbox"/>		邮箱
create_time					datetime	0	0	<input type="checkbox"/>	<input type="checkbox"/>		创建时间
update_time					datetime	0	0	<input type="checkbox"/>	<input type="checkbox"/>		修改时间

默认:

▼

☐ 根据当前时间戳更新

```
@Test
public void insertUser() {
    User user = new User();
    user.setName("Monkey");
    user.setAge(20);
    userMapper.insert(user);
}
```


id	name	age	email	create_time	update_time
226882	Monkey	20	(Null)	2020-09-28 15:50:35	2020-09-28 15:50:35

```
@Test
public void updateById() {
    User user = new User();
    user.setId(1310487052847226882L);
    user.setName("UpdateMonkey");
    user.setAge(40);
    userMapper.updateById(user);
}
```

id	name	age	email	create_time	update_time
01310487052847226882	UpdateMonkey	40	(Null)	2020-09-28 15:50:35	2020-09-28 15:54:54

代码级别

1. 先设置数据库为之前状态

字段	索引	外键	触发器	选项	注释	SQL 预览					
名					类型	长度	小数点	不是 null	虚拟	键	注释
id					bigint	20	0	<input checked="" type="checkbox"/>	<input type="checkbox"/>	 1	主键ID
name					varchar	30	0	<input type="checkbox"/>	<input type="checkbox"/>		姓名
age					int	11	0	<input type="checkbox"/>	<input type="checkbox"/>		年龄
email					varchar	50	0	<input type="checkbox"/>	<input type="checkbox"/>		邮箱
▶ create_time					datetime	<div>▼</div> 0	0	<input type="checkbox"/>	<input type="checkbox"/>		创建时间
update_time					datetime	0	0	<input type="checkbox"/>	<input type="checkbox"/>		修改时间

默认:

NULL

▼

☐ 根据当前时间戳更新

2. 自定义实现类 MyMetaObjectHandler

```
@Slf4j
@Component
public class MyMetaObjectHandler implements MetaObjectHandler {

    @Override
    public void insertFill(MetaObject metaObject) {
        log.info("start insert fill ....");
        Date date = new Date();
        this.strictInsertFill(metaObject, "createTime", Date.class, date); // 起始版本 3.3.0(推荐使用)
        this.strictInsertFill(metaObject, "updateTime", Date.class, date); // 起始版本 3.3.0(推荐使用)
    }

    @Override
    public void updateFill(MetaObject metaObject) {
        log.info("start update fill ....");
        this.strictInsertFill(metaObject, "updateTime", Date.class, new Date());
        // 起始版本 3.3.0(推荐使用)
    }
}
```

```
/**
 * 添加用户
 */
@Test
public void insertUser() {
    User user = new User();
    user.setName("Monkey");
    user.setAge(20);
    userMapper.insert(user);
}

/**
 * 修改用户
 */
@Test
public void updateUser() {
    User user = new User();
}
```

```

user.setId(1310487052847226882L);
user.setName("UpdateMonkey3333");
user.setAge(40);
userMapper.updateById(user);
}

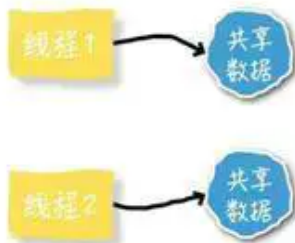
```

乐观锁插件

乐观锁：

乐观锁假设数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则返回给用户错误的信息，让用户决定如何去做。乐观锁适用于读操作多的场景，这样可以提高程序的吞吐量。

线程1获取共享数据直接开始操作



线程2获取共享数据直接开始操作

线程1在提交前，根据版本进行冲突检测



线程2在提交前，根据版本进行冲突检测

乐观锁机制：

1. 取出记录时，获取当前version
2. 更新时，带上这个version
3. 执行更新时， set version = newVersion where version = oldVersion
4. 如果version不对，就更新失败

使用场景

- 当要更新一条记录的时候，希望这条记录没有被别人更新

测试乐观锁插件

1. 修改表结构 增加字段 `version` 默认值为 1

字段	索引	外键	触发器	选项	注释	SQL 预览
名					类型	长度
id					bigint	20
name					varchar	30
age					int	11
email					varchar	50
create_time					datetime	0
update_time					datetime	0
version					int	20

默认:	1
<input type="checkbox"/> 自动递增	
<input type="checkbox"/> 无符号	
<input type="checkbox"/> 填充零	

2. 实体类添加字段 `version` 并且添加注解 `@version`

```

@Data
public class User {
    private Long id;
    private String name;
    private Integer age;
    private String email;

    @TableField(fill = FieldFill.INSERT)
    private Date createTime;

    @TableField(fill = FieldFill.INSERT_UPDATE)
    private Date updateTime;

    // 乐观锁
    @Version
    private Integer version;
}

```

3. 在 MybatisPlusConfig 中添加 乐观锁配置

```

@MapperScan("com.lomonkey.mapper")
@Configuration
public class MybatisPlusConfig {

    @Bean
    public MybatisPlusInterceptor mybatisPlusInterceptor() {
        MybatisPlusInterceptor interceptor = new MybatisPlusInterceptor();
        // 乐观锁配置
        interceptor.addInnerInterceptor(new
        OptimisticLockerInnerInterceptor());
        return interceptor;
    }
}

```

4. 测试

```

/**
 * 乐观锁测试，乐观锁 测试必须携带 version 否则无效
 *
 * 结果 线程二修改成功，线程一 修改失败
 */
@Test
public void OptimisticLocker() {

    // 线程一
    User user1 = userMapper.selectById(1L);
    // 获取 version
    Integer version = user1.getVersion();
    user1.setName("我是线程一修改的name");

    // 线程二
    User user2 = userMapper.selectById(1L);
    // 获取 version
    Integer version2 = user2.getVersion();
    user2.setName("我是线程二修改的name");
}

```

```
userMapper.updateById(user2);

userMapper.updateById(user1);

}
```

逻辑删除

说明:

只对自动注入的sql起效:

- 插入: 不作限制
- 查找: 追加where条件过滤掉已删除数据,且使用 wrapper.entity 生成的where条件会忽略该字段
- 更新: 追加where条件防止更新到已删除数据,且使用 wrapper.entity 生成的where条件会忽略该字段
- 删除: 转变为 更新

例如:

- 删除: `update user set deleted=1 where id = 1 and deleted=0`
- 查找: `select id,name,deleted from user where deleted=0`

字段类型支持说明:

- 支持所有数据类型(推荐使用 `Integer`, `Boolean`, `LocalDateTime`)
- 如果数据库字段使用 `datetime`,逻辑未删除值和已删除值支持配置为字符串 `null`,另一个值支持配置为函数来获取值如 `now()`

附录:

- 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。
- 如果你需要频繁查出来看就不应使用逻辑删除, 而是以一个状态去表示。

测试:

1. 修改表结构 添加 `id_delete` 字段, 1表示删除, 0表示未删除
2. 修改实体类添加属性 `isDeleted`

```
@Data
public class User {
    @TableId(type = IdType.ASSIGN_ID)
    private Long id;
    private String name;
    private Integer age;
    private String email;

    @TableField(fill = FieldFill.INSERT)
    private Date createTime;

    @TableField(fill = FieldFill.INSERT_UPDATE)
    private Date updateTime;

    @Version
    private Integer version;

    /** 逻辑删除: 1表示删除 0 表示未删除*/
}
```



```
@TableLogic
private Integer isDeleted;
}
```

3. 在 application.yml 中配置逻辑删除配置

```
mybatis-plus:
  configuration:
    log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
  global-config:
    db-config:
      # 删除用 1
      logic-delete-field: 1
      # 未删除用 0
      logic-not-delete-value: 0
```

4. 测试删除

```
@Test
public void deleteTest() {
    int delete = userMapper.deleteById(1310419023559942154L);
    System.out.println("delete = " + delete);
}
```

```
JDBC Connection [HikariProxyConnection@346847161 wrapping com.mysql.cj.jdbc.ConnectionImpl@3f4cd155] will not be managed by Spring
==> Preparing: UPDATE user SET is_deleted=1 WHERE id=? AND is_deleted=0
==> Parameters: 1310419023559942154 (Long)
<== Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@39109136]
delete = 1
```

性能分析插件

```
// 该插件 3.1.2 后版本废弃，推荐使用
// @Bean
// public PerformanceInterceptor performanceInterceptor(){
//     //启用性能分析插件
//     return new PerformanceInterceptor();
// }
```

在 classpath 下添加配置 spy.properties

```
modulelist=com.baomidou.mybatisplus.extension.p6spy.MybatisPlusLogFactory,com.p6spy.engine.outage.P6OutageFactory
# 自定义日志打印
logMessageFormat=com.baomidou.mybatisplus.extension.p6spy.P6SpyLogger
#日志输出到控制台
appender=com.baomidou.mybatisplus.extension.p6spy.StdoutLogger
# 使用日志系统记录 sql
#appender=com.p6spy.engine.spy.appender.Slf4JLogger
# 设置 p6spy driver 代理
deregisterdrivers=true
# 取消JDBC URL前缀
useprefix=true
# 配置记录 Log 例外,可去掉的结果集有
error,info,batch,debug,statement,commit,rollback,result,resultset.
excludecategories=info,debug,result,commit,resultset
# 日期格式
dateformat=yyyy-MM-dd HH:mm:ss
```

```
# 实际驱动可多个
#driverlist=org.h2.Driver
# 是否开启慢SQL记录
outagedetection=true
# 慢SQL记录标准 2 秒
outagedetectioninterval=2
```

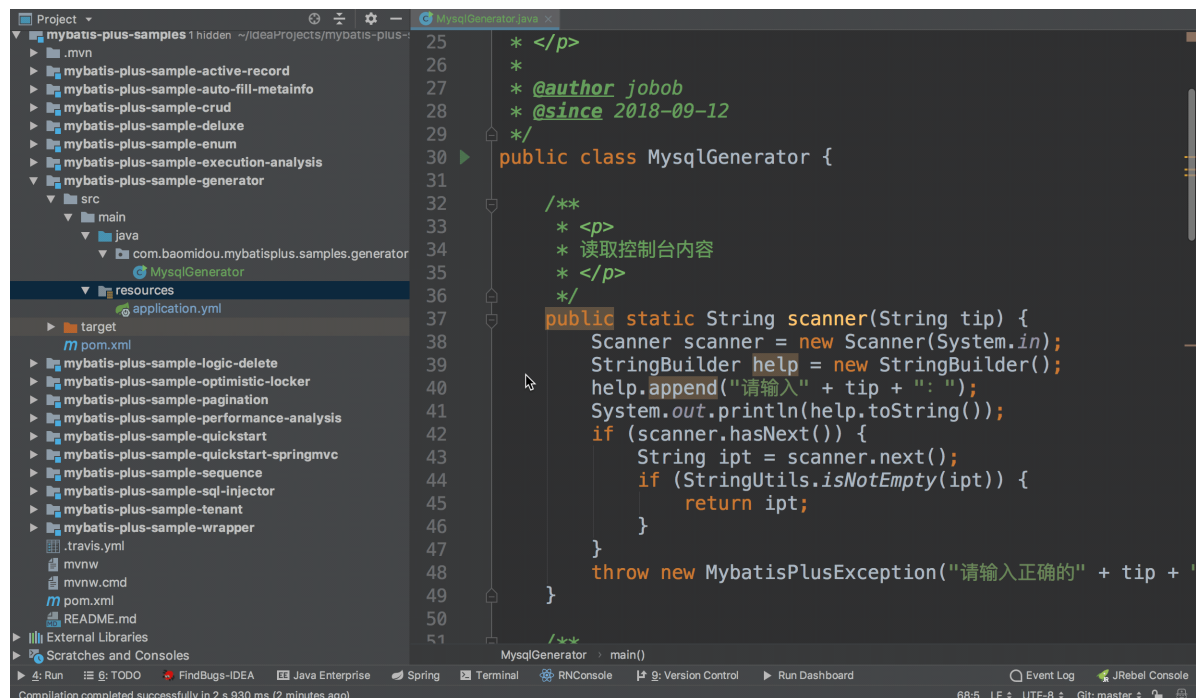
代码生成器

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：



// 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

```
public class CodeGenerator {

    /**
     * <p>
     * 读取控制台内容
     * </p>
     */
    public static String scanner(String tip) {
        Scanner scanner = new Scanner(System.in);
        StringBuilder help = new StringBuilder();
        help.append("请输入" + tip + "：");
        System.out.println(help.toString());
        if (scanner.hasNext()) {
            String ipt = scanner.next();
            if (StringUtils.isNotBlank(ipt)) {
                return ipt;
            }
        }
    }
}
```

```

    }
    throw new MybatisPlusException("请输入正确的" + tip + "! ");
}

public static void main(String[] args) {
    // 代码生成器
    AutoGenerator mpg = new AutoGenerator();

    // 全局配置
    GlobalConfig gc = new GlobalConfig();
    String projectPath = System.getProperty("user.dir");
    gc.setOutputDir(projectPath + "/src/main/java");
    gc.setAuthor("jobob");
    gc.setOpen(false);
    // gc.setSwagger2(true); 实体属性 Swagger2 注解
    mpg.setGlobalConfig(gc);

    // 数据源配置
    DataSourceConfig dsc = new DataSourceConfig();
    dsc.setUrl("jdbc:mysql://localhost:3306/ant?
useUnicode=true&useSSL=false&characterEncoding=utf8");
    // dsc.setSchemaName("public");
    dsc.setDriverName("com.mysql.jdbc.Driver");
    dsc.setUsername("root");
    dsc.setPassword("密码");
    mpg.setDataSource(dsc);

    // 包配置
    PackageConfig pc = new PackageConfig();
    pc.setModuleName(scanner("模块名"));
    pc.setParent("com.baomidou.ant");
    mpg.setPackageInfo(pc);

    // 自定义配置
    InjectionConfig cfg = new InjectionConfig() {
        @Override
        public void initMap() {
            // to do nothing
        }
    };

    // 如果模板引擎是 freemarker
    String templatePath = "/templates/mapper.xml.ftl";
    // 如果模板引擎是 velocity
    // String templatePath = "/templates/mapper.xml.vm";

    // 自定义输出配置
    List<FileOutConfig> focList = new ArrayList<>();
    // 自定义配置会被优先输出
    focList.add(new FileOutConfig(templatePath) {
        @Override
        public String outputFile(TableInfo tableInfo) {
            // 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！！
            return projectPath + "/src/main/resources/mapper/" +
pc.getModuleName()
                + "/" + tableInfo.getEntityName() + "Mapper" +
StringPool.DOT_XML;

```

```

    }
});
/*
cfg.setFileCreate(new IFileCreate() {
    @Override
    public boolean isCreate(ConfigBuilder configBuilder, FileType
fileType, String filePath) {
        // 判断自定义文件夹是否需要创建
        checkDir("调用默认方法创建的目录，自定义目录用");
        if (fileType == FileType.MAPPER) {
            // 已经生成 mapper 文件判断存在，不想重新生成返回 false
            return !new File(filePath).exists();
        }
        // 允许生成模板文件
        return true;
    }
});
*/
cfg.setFileOutConfigList(focList);
mpg.setCfg(cfg);

// 配置模板
TemplateConfig templateConfig = new TemplateConfig();

// 配置自定义输出模板
//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
// templateConfig.setEntity("templates/entity2.java");
// templateConfig.setService();
// templateConfig.setController();

templateConfig.setXml(null);
mpg.setTemplate(templateConfig);

// 策略配置
StrategyConfig strategy = new StrategyConfig();
strategy.setNaming(NamingStrategy.underline_to_camel);
strategy.setColumnNaming(NamingStrategy.underline_to_camel);
strategy.setSuperEntityClass("你自己的父类实体,没有就不用设置!");
strategy.setEntityLombokModel(true);
strategy.setRestControllerStyle(true);
// 公共父类
strategy.setSuperControllerClass("你自己的父类控制器,没有就不用设置!");
// 写于父类中的公共字段
strategy.setSuperEntityColumns("id");
strategy.setInclude(scanner("表名，多个英文逗号分割").split(","));
strategy.setControllerMappingHyphenStyle(true);
strategy.setTablePrefix(pc.getModuleName() + "_");
mpg.setStrategy(strategy);
mpg.setTemplateEngine(new FreemarkerTemplateEngine());
mpg.execute();
}
}

```

更多详细配置，请参考[代码生成器配置](#)一文。

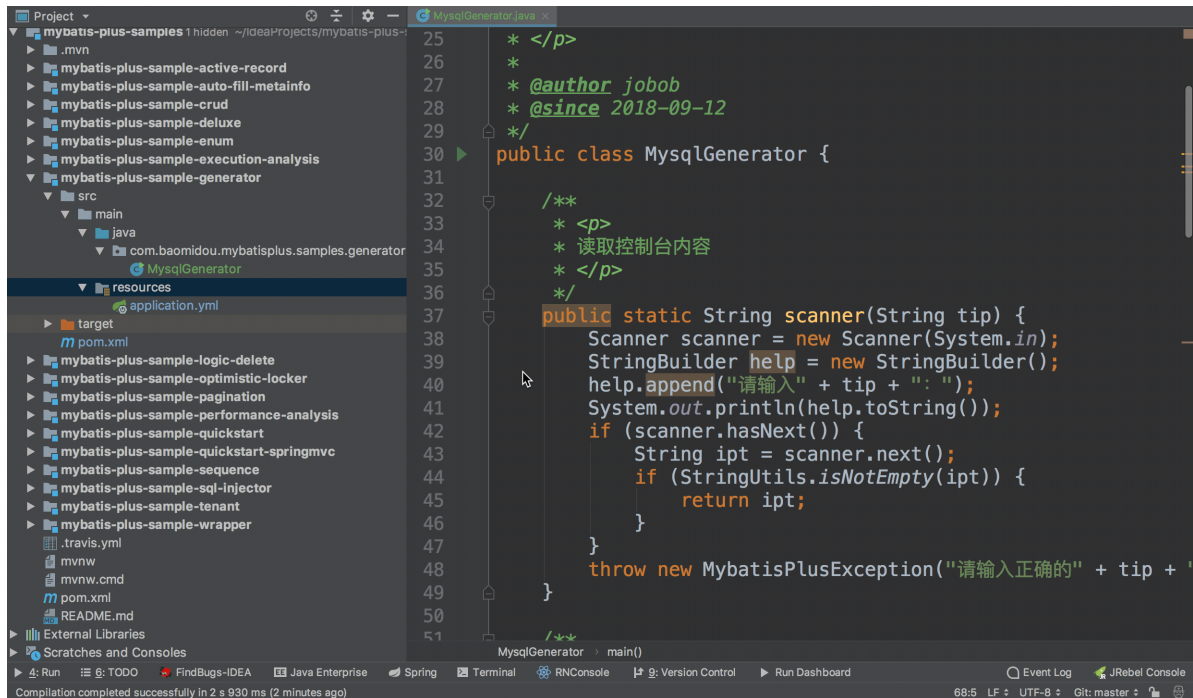
代码生成器

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:



// 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

```
public class CodeGenerator {

    /**
     * <p>
     * 读取控制台内容
     * </p>
     */
    public static String scanner(String tip) {
        Scanner scanner = new Scanner(System.in);
        StringBuilder help = new StringBuilder();
        help.append("请输入" + tip + ": ");
        System.out.println(help.toString());
        if (scanner.hasNext()) {
            String ipt = scanner.next();
            if (StringUtils.isNotBlank(ipt)) {
                return ipt;
            }
        }
        throw new MybatisPlusException("请输入正确的" + tip + "!!");
    }

    public static void main(String[] args) {
        // 代码生成器
        AutoGenerator mpg = new AutoGenerator();
```

```

// 全局配置
GlobalConfig gc = new GlobalConfig();
String projectPath = System.getProperty("user.dir");
gc.setOutputDir(projectPath + "/src/main/java");
gc.setAuthor("jobob");
gc.setOpen(false);
// gc.setSwagger2(true); 实体属性 Swagger2 注解
mpg.setGlobalConfig(gc);

// 数据源配置
DataSourceConfig dsc = new DataSourceConfig();
dsc.setUrl("jdbc:mysql://localhost:3306/ant?
useUnicode=true&useSSL=false&characterEncoding=utf8");
// dsc.setSchemaName("public");
dsc.setDriverName("com.mysql.jdbc.Driver");
dsc.setUsername("root");
dsc.setPassword("密码");
mpg.setDataSource(dsc);

// 包配置
PackageConfig pc = new PackageConfig();
pc.setModuleName(scanner("模块名"));
pc.setParent("com.baomidou.ant");
mpg.setPackageInfo(pc);

// 自定义配置
InjectionConfig cfg = new InjectionConfig() {
    @Override
    public void initMap() {
        // to do nothing
    }
};

// 如果模板引擎是 freemarker
String templatePath = "/templates/mapper.xml.ftl";
// 如果模板引擎是 velocity
// String templatePath = "/templates/mapper.xml.vm";

// 自定义输出配置
List<FileOutConfig> focList = new ArrayList<>();
// 自定义配置会被优先输出
focList.add(new FileOutConfig(templatePath) {
    @Override
    public String outputFile(TableInfo tableInfo) {
        // 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！！
        return projectPath + "/src/main/resources/mapper/" +
pc.getModuleName()
        + "/" + tableInfo.getEntityName() + "Mapper" +
StringPool.DOT_XML;
    }
});
/*
cfg.setFileCreate(new IFileCreate() {
    @Override
    public boolean isCreate(ConfigBuilder configBuilder, FileType
fileType, String filePath) {

```

```

        // 判断自定义文件夹是否需要创建
        checkDir("调用默认方法创建的目录，自定义目录用");
        if (fileType == FileType.MAPPER) {
            // 已经生成 mapper 文件判断存在，不想重新生成返回 false
            return !new File(filePath).exists();
        }
        // 允许生成模板文件
        return true;
    }
});
*/
cfg.setFileOutConfigList(focList);
mpg.setCfg(cfg);

// 配置模板
TemplateConfig templateConfig = new TemplateConfig();

// 配置自定义输出模板
//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
// templateConfig.setEntity("templates/entity2.java");
// templateConfig.setService();
// templateConfig.setController();

templateConfig.setXml(null);
mpg.setTemplate(templateConfig);

// 策略配置
StrategyConfig strategy = new StrategyConfig();
strategy.setNaming(NamingStrategy.underline_to_camel);
strategy.setColumnNaming(NamingStrategy.underline_to_camel);
strategy.setSuperEntityClass("你自己的父类实体,没有就不用设置!");
strategy.setEntityLombokModel(true);
strategy.setRestControllerStyle(true);
// 公共父类
strategy.setSuperControllerClass("你自己的父类控制器,没有就不用设置!");
// 写于父类中的公共字段
strategy.setSuperEntityColumns("id");
strategy.setInclude(scanner("表名，多个英文逗号分割").split(","));
strategy.setControllerMappingHyphenStyle(true);
strategy.setTablePrefix(pc.getModuleName() + "_");
mpg.setStrategy(strategy);
mpg.setTemplateEngine(new FreemarkerTemplateEngine());
mpg.execute();
}

}

```

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

```
<dependency>
  <groupId>com.baomidou</groupId>
  <artifactId>mybatis-plus-generator</artifactId>
  <version>3.4.0</version>
</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）：

```
<dependency>
  <groupId>org.apache.velocity</groupId>
  <artifactId>velocity-engine-core</artifactId>
  <version>2.2</version>
</dependency>
```

Freemarker：

```
<dependency>
  <groupId>org.freemarker</groupId>
  <artifactId>freemarker</artifactId>
  <version>2.3.30</version>
</dependency>
```

Beetl：

```
<dependency>
  <groupId>com.ibeetl</groupId>
  <artifactId>beetl</artifactId>
  <version>3.2.1.RELEASE</version>
</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。

```
AutoGenerator generator = new AutoGenerator();

// set freemarker engine
generator.setTemplateEngine(new FreemarkerTemplateEngine());

// set beetl engine
generator.setTemplateEngine(new BeetlTemplateEngine());

// set custom engine (reference class is your custom engine class)
generator.setTemplateEngine(new CustomTemplateEngine());

// other config
...
```


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

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

- 配置 DataSourceConfig

```
DataSourceConfig dataSourceConfig = new DataSourceConfig();
dataSourceConfig.setUrl("jdbc:mysql://localhost:3306/ant?
useUnicode=true&useSSL=false&characterEncoding=utf8");
dataSourceConfig.setDriverName("com.mysql.jdbc.Driver");
dataSourceConfig.setUsername("root");
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

```
//指定自定义模板路径，位置：/resources/templates/entity2.java.ftl(或者是.vm)
//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
TemplateConfig templateConfig = new TemplateConfig()
    .setEntity("templates/entity2.java");

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

自定义属性注入

```
InjectionConfig injectionConfig = new InjectionConfig() {
    //自定义属性注入:abc
    //在.ftl(或者是.vm)模板中，通过${cfg.abc}获取属性
    @Override
    public void initMap() {
        Map<String, Object> map = new HashMap<>();
        map.put("abc", this.getConfig().getGlobalConfig().getAuthor() + "-mp");
        this.setMap(map);
    }
};
AutoGenerator mpg = new AutoGenerator();
//配置自定义属性注入
mpg.setCfg(injectionConfig);
entity2.java.ftl
自定义属性注入abc=${cfg.abc}
```

```
entity2.java.vm
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), propertyName=id, columnType={DbColumnType@2135} "INTEGER", comment="", fill=null, customMap={HashMap@1956} size=2)"
  convert = false
  keyFlag = true
  keyIdentityFlag = true
  name = "id"
  type = "int(11)"
  propertyName = "id"
  columnType = {DbColumnType@2135} "INTEGER"
  comment = ""
  fill = null
  customMap = {HashMap@1956} size = 2
    "NULL" -> "NO"
    "PRIVILEGES" -> "select,insert,update,references"
```

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

1 show full fields from jobs_info

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

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

```
new DataSourceConfig().setDbQuery(new MySqlQuery() {

    /**
     * 重写父类预留查询自定义字段<br>
     * 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>
     * 模板中调用： table.fields 获取所有字段信息，
     * 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
     */
    @Override
    public String[] fieldCustom() {
        return new String[]{"NULL", "PRIVILEGES"};
    }
})
```

附录

注解-官网地址

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(根据你的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:

1. 定义在resultMap里,只作用于select查询的返回结果封装
2. 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>#{ ... }</code> 部分
exist	boolean	否	true	是否为数据库表字段
condition	String	否	""	字段 where 实体查询比较条件,有值设置则按设置的值为准,没有则为默认全局的 <code>%s=#{%s}</code> , 参考
update	String	否	""	字段 update set 部分注入,例如: <code>update="%s+1"</code> : 表示更新时会 set <code>version=version+1</code> (该属性优先级高于 <code>el</code> 属性)
insertStrategy	Enum	N	DEFAULT	举例: NOT_NULL: <code>insert into table_a(<if test="columnProperty != null">column</if>) values (<if test="columnProperty != null">#{columnProperty}</if>)</code>
updateStrategy	Enum	N	DEFAULT	举例: IGNORED: <code>update table_a set column=#{columnProperty}</code>
whereStrategy	Enum	N	DEFAULT	举例: NOT_EMPTY: <code>where <if test="columnProperty != null and columnProperty!=''">column=#{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 format 进行处理
jdbcType	JdbcType	否	JdbcType.UNDEFINED	JDBC类型 (该默认值不代表会按照该值生效)
typeHandler	Class<? extends TypeHandler>	否	UnknownTypeHandler.class	类型处理器 (该默认值不代表会按照该值生效)
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 `@version` 在字段上

@EnumValue

- 描述：通枚举类注解(注解在枚举字段上)

@TableLogic

- 描述：表字段逻辑处理注解（逻辑删除）

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

@SqlParser

- 描述：租户注解,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"