MyBatis-Plus

看官方文档是很重要的，下面的学习也是基于官方文档。 https://mp.baomidou.com/ 。

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

快速入门

1. 导入对应的依赖
2. 研究以来如何配置
3. 代码如何编写
4. 怎么扩展

步骤

1. 创建数据库 mybatis_plus
2. 创建user表

```sql
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)
);
```

```sql
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');
```

3. 编写项目，初始化项目，这里使用SpringBoot
4. 导入依赖

```xml
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.3.1.tmp</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

5. 连接数据库

```
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.url=jdbc:mysql://localhost:3306/mybatis_plus?
useSSL=true&useUnicode=true&characterEncoding=utf-8
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

6. 编写代码

- pojo

```
package com.cc.pojo;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private Long id;
    private String name;
    private String password;
    private String email;
}
```

- mapper接口

```
package com.cc.mapper;

import com.baomidou.mybatisplus.core.mapper.BaseMapper;
import com.cc.pojo.User;
import org.springframework.stereotype.Repository;

@Repository//代表持久层
public interface UserMapper extends BaseMapper<User> {

}
```

- 注意：这里编写完我们要在主启动类上去扫描我们的mapper包下的所有接口

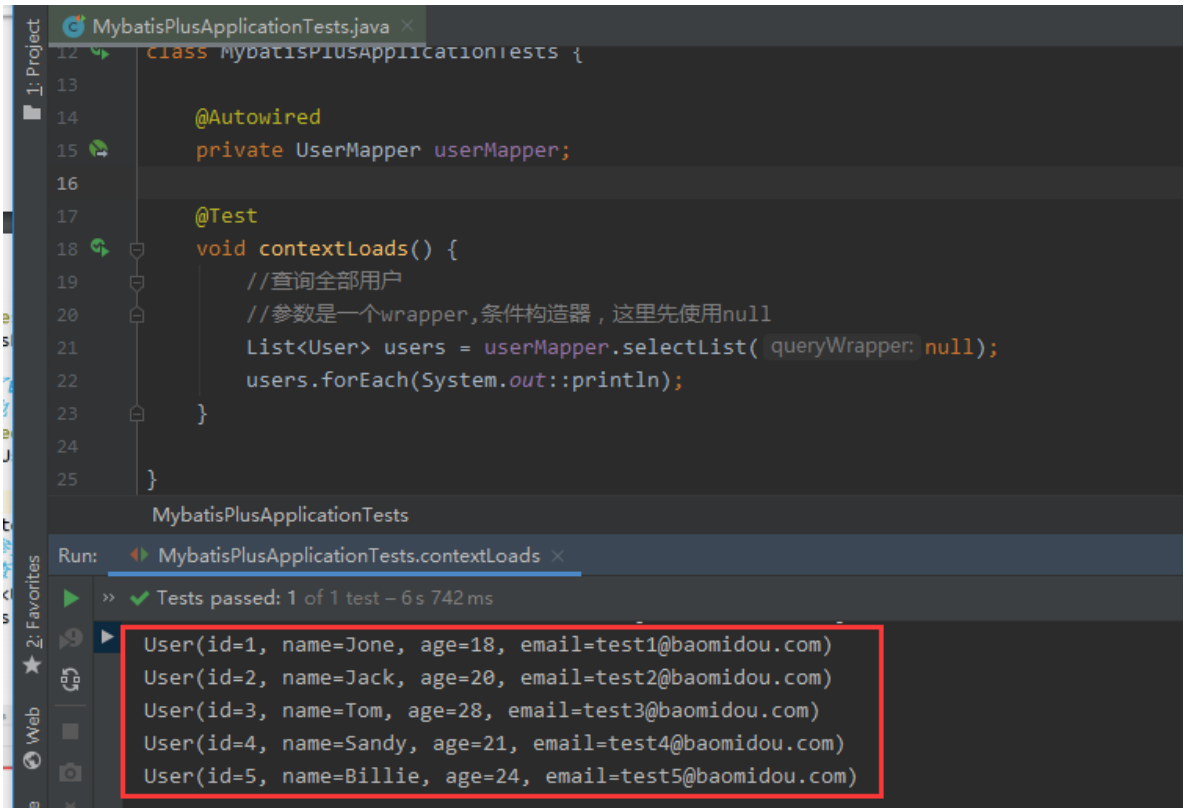
```
@MapperScan("com.cc.mapper")
```

- 测试类中进行测试

```
package com.cc;

import com.cc.mapper.UserMapper;
import com.cc.pojo.User;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.List;
```

```java
@SpringBootTest
class MybatisPlusApplicationTests {

    @Autowired
    private UserMapper userMapper;

    @Test
    void contextLoads() {
        //查询全部用户
        //参数是一个wrapper,条件构造器，这里先使用null
        List<User> users = userMapper.selectList(null);
        users.forEach(System.out::println);
    }

}
```

- 结果



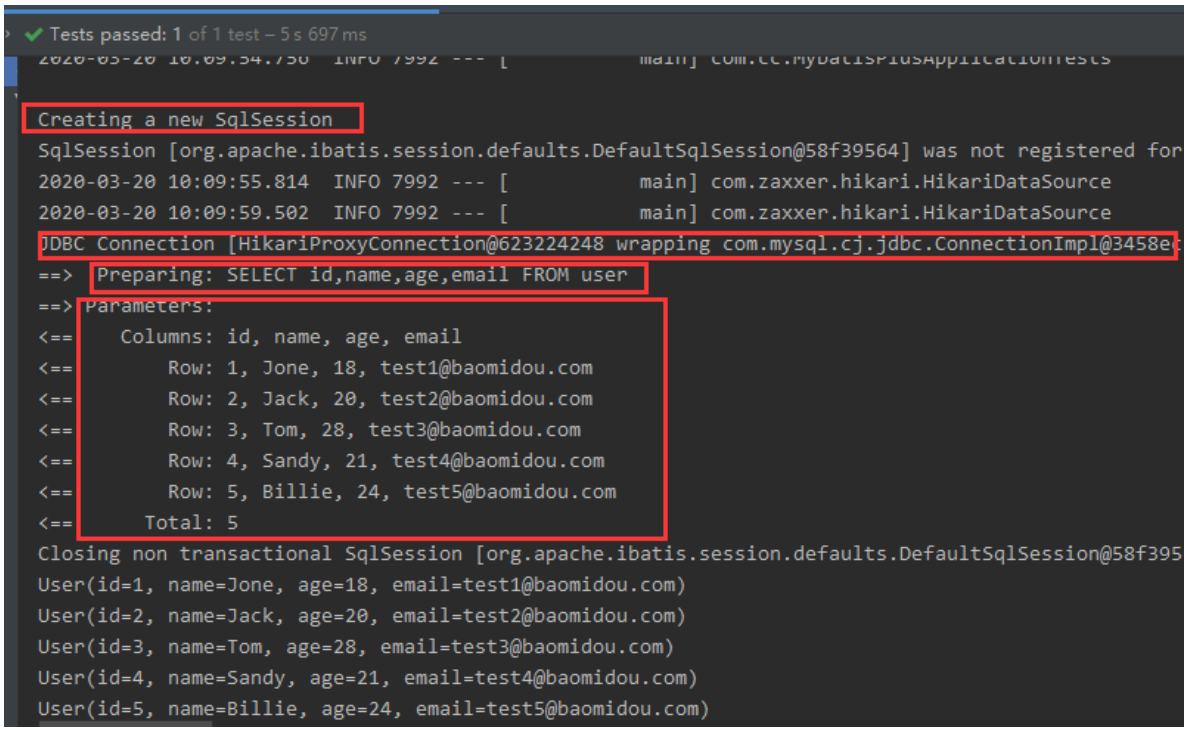
结果全部查询完毕！

可以看到我们只写了一个mapper接口，方法和sql语句一点都没写，那么是谁帮我们写的呢？不言而喻，都是Mybatis-Plus帮我们写好了。

配置日志

在学习mybatis的时候，我们想要看sql语句，就配置了日志，这里也同样我们可以配置一下。

```
#配置日志
mybatis-plus.configuration.log-impl=org.apache.ibatis.logging.stdout.StdOutImpl
```

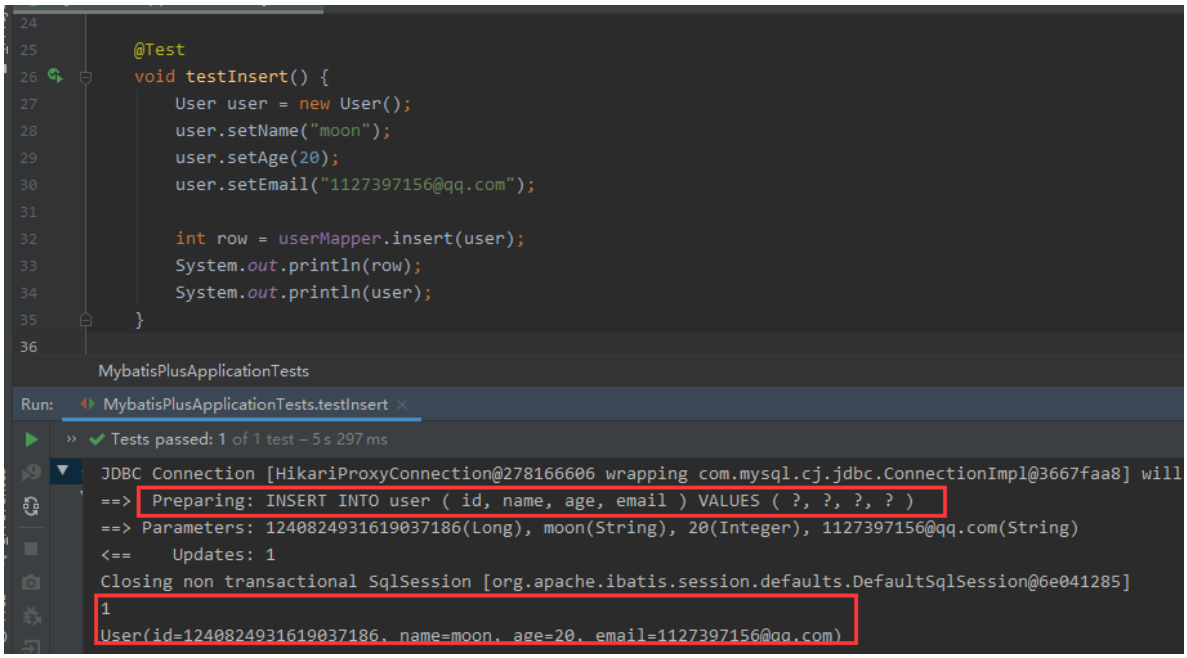


配置完日志，再次测试，可以看到自动生成的sql语句。是不是很简便！！！

CRUD扩展

插入操作

insert插入



这里我们没有设置id，但是从插入结果上看，自动帮我们生成了一个id.

主键生成策略

雪花算法：

snowflake是Twitter开源的分布式ID生成算法，结果是一不个long型的ID。其核心思想是：使用41bit作为毫秒数，10bit作为机器的ID(5个bit是数据中心，5个bit的机器ID)，12bit作为毫秒内的流水号（意味着每个节点在每毫秒可以产生4096个ID），最后还有一个符号位，永远是0。可以保证几乎全球唯一。

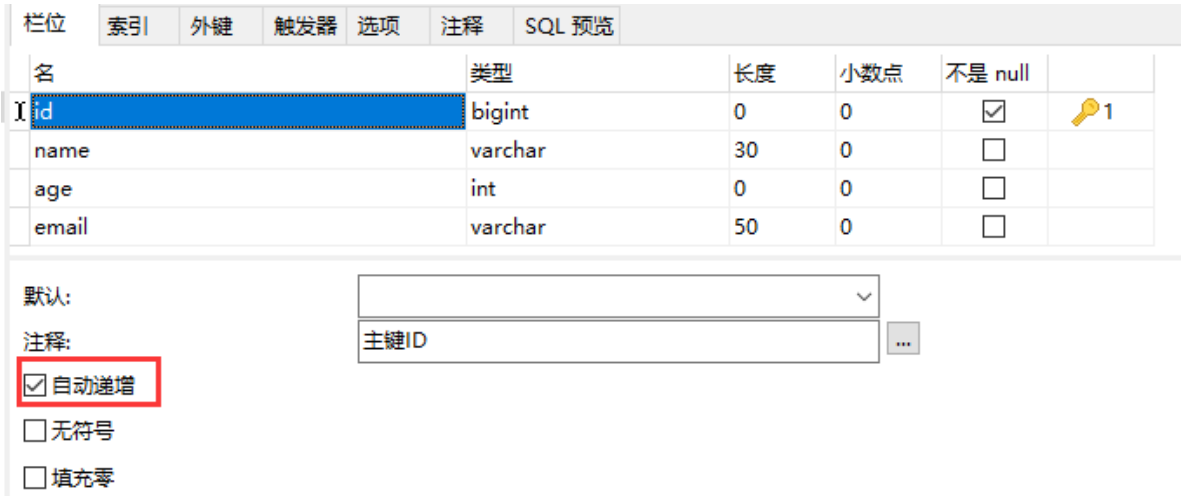
主键自增

我么需要配置逐渐自增：

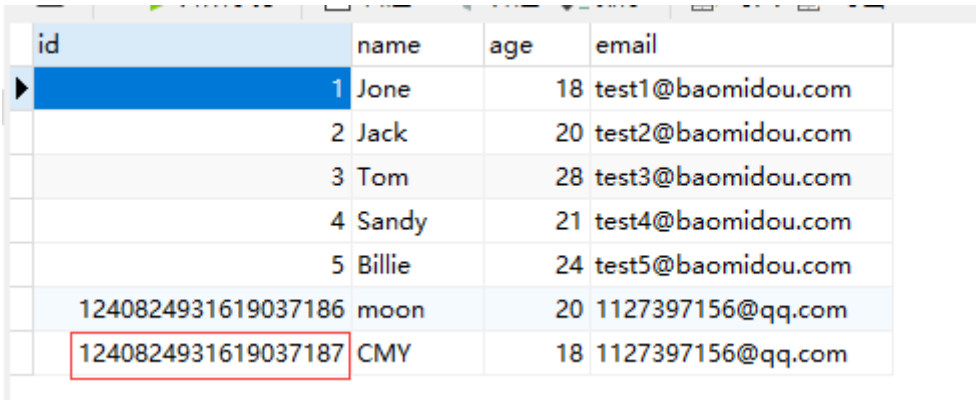
1. 实体字段上

```
@TableId(type = IdType.AUTO)
private Long id;
```

2. 数据库字段一定设置是自增



3. 再次测试插入即可。



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
1240824931619037186	moon	20	1127397156@qq.com
1240824931619037187	CMY	18	1127397156@qq.com

其余的源码解释

```
public enum IdType {
    AUTO(0),//数据库id自增
    NONE(1),//未设置主键
    INPUT(2),//手动输入
    ID_WORKER(3),//默认的全局唯一id
    ID_WORKER_STR(4),//ID_WORKER 字符串表示法
    UUID(5);//全局唯一id uuid
}
```

更新操作

```
@Test
    void testUpdate() {
        User user = new User();
        user.setId(1L);
        user.setName("MOON");
        user.setAge(6);

        int i = userMapper.updateById(user);
        System.out.println(i);
    }
```

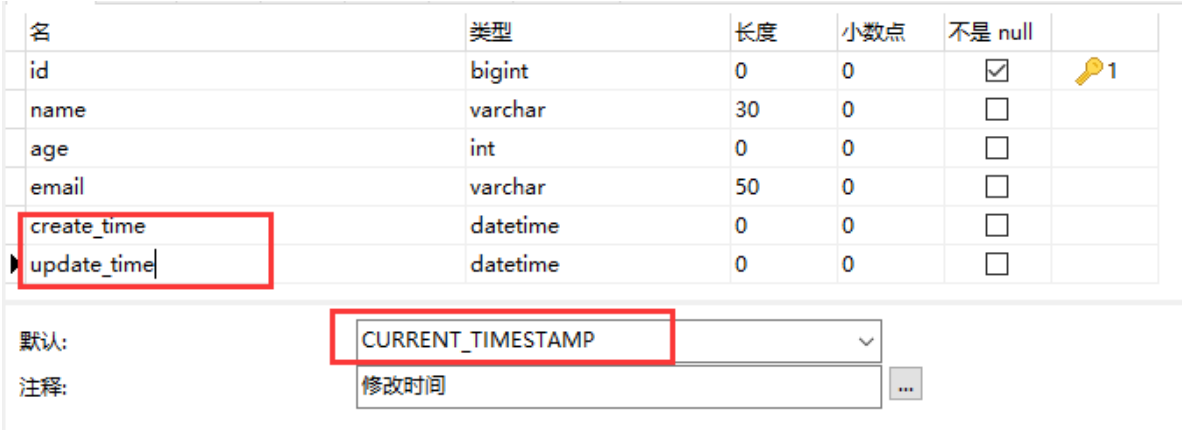
```
Creating a new SqlSession
SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@3d96fa9e] was not registered for
2020-03-20 10:50:51.488  INFO 6584 --- [           main] com.zaxxer.hikari.HikariDataSource
2020-03-20 10:51:02.238  INFO 6584 --- [           main] com.zaxxer.hikari.HikariDataSource
JDBC Connection [HikariProxyConnection@1567857145 wrapping com.mysql.cj.jdbc.ConnectionImpl@49a6f4
==>  Preparing: UPDATE user SET name=?, age=? WHERE id=?
==> Parameters: MOON(String), 6(Integer), 1(Long)
<==    Updates: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@3d96fa9
1
```

自动填充

在设计数据库的时候，我们很多时候都有创建时间，修改时间！那我么这里怎么让他自动完成呢！

方式一：**数据库级别**

1. 在表中新增字段 create_time， update_time



2. 实体类中加上这两个字段

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

3.再次测试插入方法

id	name	age	email	create_time	update_time
1	MOON	6	test1@baomidou.com	2020-03-20 10:58:16	2020-03-20 10:58:16
2	Jack	20	test2@baomidou.com	2020-03-20 10:58:16	2020-03-20 10:58:16
3	Tom	28	test3@baomidou.com	2020-03-20 10:58:16	2020-03-20 10:58:16
4	Sandy	21	test4@baomidou.com	2020-03-20 10:58:16	2020-03-20 10:58:16
5	Billie	24	test5@baomidou.com	2020-03-20 10:58:16	2020-03-20 10:58:16
1240824931619037186	moon	20	1127397156@qq.com	2020-03-20 10:58:16	2020-03-20 10:58:16
1240824931619037187	CMY	18	1127397156@qq.com	2020-03-20 10:58:16	2020-03-20 10:58:16
1240824931619037188	CMY	18	1127397156@qq.com	2020-03-20 11:02:39	2020-03-20 11:02:39

4. 再次测试更新，查看时间是否变化

方式二：代码级别

1. 删除数据库的默认值, 更新操作。
2. 实体类字段属性上需要增加注解

```
//字段上添加填充内容
@TableField(fill = FieldFill.INSERT)
private Date createTime;

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

注意：Date一定不要导错包，确保导入的包是

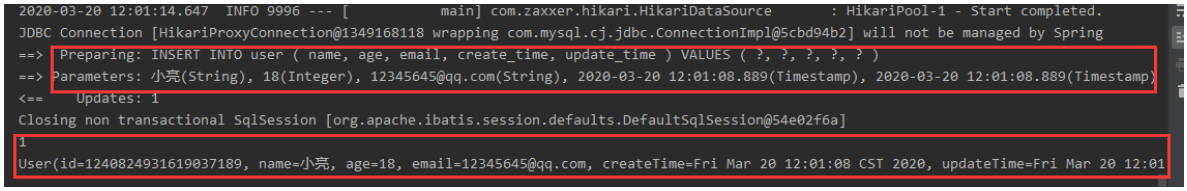
```
import java.util.Date;
//不然会出现下述错误：
Could not set property 'createTime' of 'class com.cc.pojo.User' with value 'Fri
Mar 20 11:54:32 CST 2020' Cause: java.lang.IllegalArgumentException: argument
type mismatch
```

3. 编写处理球来处理这个注解即可。

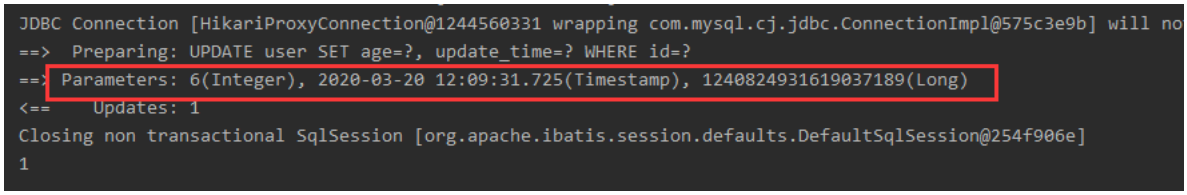
```
package com.cc.handler;

import com.baomidou.mybatisplus.core.handlers.MetaObjectHandler;
import lombok.extern.slf4j.Slf4j;
import org.apache.ibatis.reflection.MetaObject;
import org.springframework.stereotype.Component;

import java.util.Date;

@Slf4j
@Component
public class MyMetaObjectHandler implements MetaObjectHandler {
    //插入时的填充策略
    @Override
    public void insertFill(MetaObject metaObject) {
        log.info("start insert fill....");
        this.setFieldValByName("createTime",new Date(),metaObject);
        this.setFieldValByName("updateTime",new Date(),metaObject);
    }

    //更新时的填充策略
    @Override
    public void updateFill(MetaObject metaObject) {
```

```
            log.info("start update  fill....");
            this.setFieldValByName("updateTime",new Date(),metaObject);
        }
    }
```

4. 测试插入方法，查看时间。



5. 测试更新，观察时间即可。



乐观锁

> 乐观锁 : 故名思意十分乐观，它总是认为不会出现问题，无论干什么不去上锁！如果出现了问题， 再次更新值测试
>
> 悲观锁：故名思意十分悲观，它总是认为总是出现问题，无论干什么都会上锁！再去操作！

这里说一下乐观锁

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：1、先查询，获得版本号 version = 1
-- A
update user set name = "moon", version = version + 1
where id = 2 and version = 1
-- B 线程抢先完成，这个时候 version = 2，会导致 A 修改失败!
update user set name = "moon", version = version + 1
where id = 2 and version = 1
```

测试一下MP的乐观锁插件

1.给数据库中增加version字段

id	name	age	email	version	create_time	update_time
1	MOON--	6	test1@baomidou.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16
2	Jack	20	test2@baomidou.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16
3	Tom	28	test3@baomidou.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16
4	Sandy	21	test4@baomidou.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16
5	Billie	24	test5@baomidou.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16
1240824931619037186	moon	20	1127397156@qq.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16
1240824931619037187	CMY	18	1127397156@qq.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16
1240824931619037188	CMY	6	1127397156@qq.com	1	2020-03-20 11:02:39	2020-03-20 11:02:39
1240824931619037189	小亮	6	12345645@qq.com	1	2020-03-20 12:01:09	2020-03-20 12:09:32

2. 给实体类中加对应的字段

```
@Version
private Integer version;
```

3. 注册组件

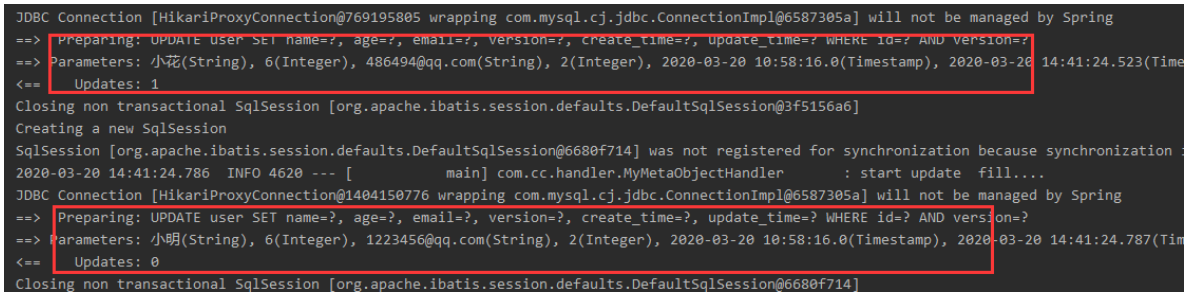
```
package com.cc.config;

import com.baomidou.mybatisplus.extension.plugins.OptimisticLockerInterceptor;
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@MapperScan("com.cc.mapper")
@EnableTransactionManagement
@Configuration//配置类
public class MyBatisPlusConfig {
    //注册乐观锁插件
    @Bean
    public OptimisticLockerInterceptor optimisticLockerInterceptor() {
        return new OptimisticLockerInterceptor();
    }

}
```

4. 测试 ！

```
//测试乐观锁成功
@Test
void testOptimisticLocker() {
    // 1. 查询用户信息
    User user = userMapper.selectById(1L);
    // 2. 修改用户信息
    user.setName("小月");
    user.setEmail("123456@qq.com");
    // 3. 执行更新操作
    userMapper.updateById(user);
}
```

```
// 测试乐观锁失败！多线程下
@Test
public void testOptimisticLocker2(){
```

```
    // 线程 1
    User user = userMapper.selectById(1L);
    user.setName("小明");
    user.setEmail("1223456@qq.com");

    // 模拟另外一个线程执行了插队操作
    User user2 = userMapper.selectById(1L);
    user2.setName("小花");
    user2.setEmail("486494@qq.com");
    userMapper.updateById(user2);

    // 自旋锁来多次尝试提交!
    userMapper.updateById(user); // 如果没有乐观锁就会覆盖插队线程的值!
}
```



id	name	age	email	version	create_time	update_time
1	小花	6	486494@qq.com	2	2020-03-20 10:58:16	2020-03-20 14:41:25

可以看到数据库中只更新了插队更新的那条数据。

查询操作

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

// 测试批量查询!
@Test
public void testSelectByBatchId(){
    List<User> users = userMapper.selectBatchIds(Arrays.asList(1, 2, 3));
    users.forEach(System.out::println);
}

// 按条件查询之一使用map操作
@Test
public void testSelectByBatchIds(){
    HashMap<String, Object> map = new HashMap<>();
    // 自定义要查询
    map.put("name","狂神说Java");
    map.put("age",3);
    List<User> users = userMapper.selectByMap(map);
    users.forEach(System.out::println);
}
```

分页查询

基本每个网站中都使用了分页插件。

1. 原始的limit进行分页
2. pageHelper第三方插件
3. MP也内置了分页插件。

使用

1. 配置拦截器组件即可

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

2. 直接使用Page对象即可。

```
@Test
void testPage() {
    //参数一：  当前页
    //参数二：  页面大小
    //使用了分页插件之后，所有的分页操作都非常简单！
    Page<User> page = new Page<>(2, 5);
    userMapper.selectPage(page, null);

    page.getRecords().forEach(System.out::println);
    System.out.println(page.getTotal());
}
```

结果

删除操作

1、根据 id 删除记录

```
// 测试删除
@Test
public void testDeleteById(){
    userMapper.deleteById(1240824931619037186L);
}

// 通过id批量删除
```

```
@Test
public void testDeleteBatchId(){

 userMapper.deleteBatchIds(Arrays.asList(1240824931619037187L,124082493161903718
8L));
}

// 通过map删除
@Test
public void testDeleteMap() {
    HashMap<String, Object> map = new HashMap<>();
    map.put("name", "小花");
    userMapper.deleteByMap(map);

}
```

有时候我们会遇到逻辑删除！

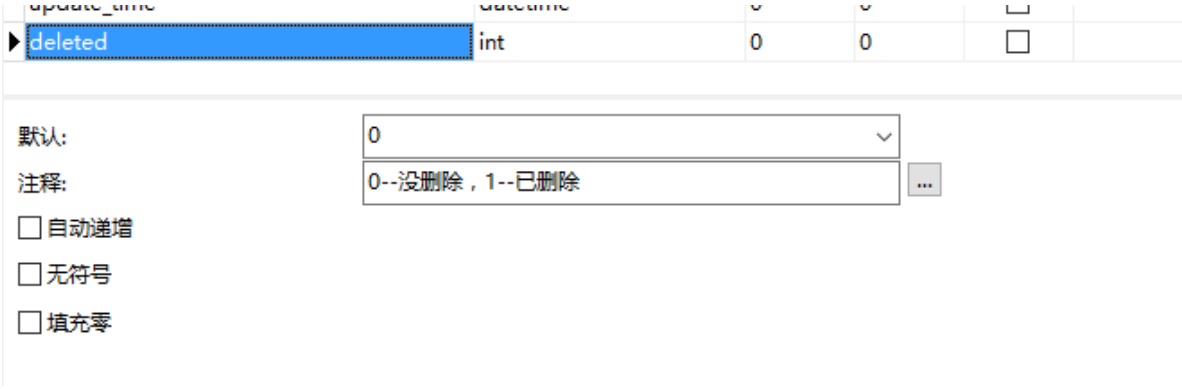
逻辑删除

> 物理删除：从数据库中直接删除
> 逻辑删除：在数据库中没有被移除，而是通过一个变量来让他失效 deleted = 0 => deleted = 1

管理员可以查看被删除的记录，防止数据的丢失，类似与回收站。

测试一下：

1.在表中增加一个deleted字段



2.在实体类中增加属性

```
@TableLogic //逻辑删除
private Integer deleted;
```

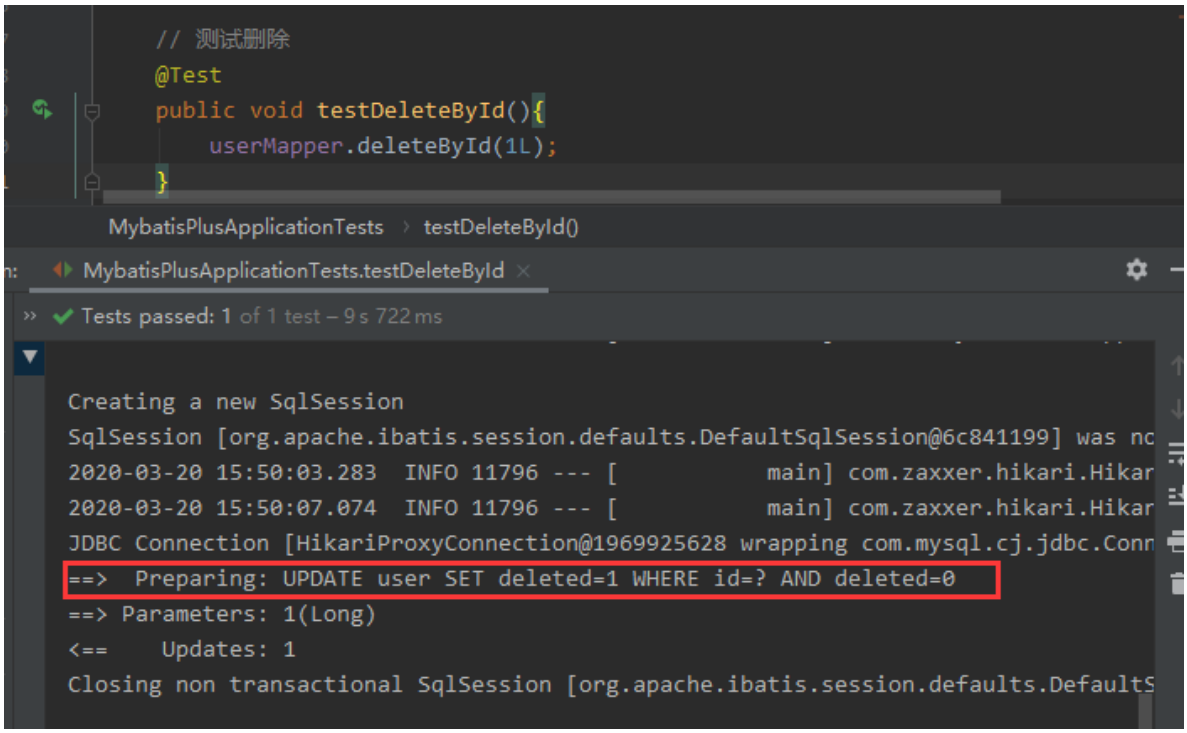
3. 配置

```
#配置逻辑删除
mybatis-plus.global-config.db-config.logic-delete-value=1
mybatis-plus.global-config.db-config.logic-not-delete-value=0
```

如果你引入的MP版本是3.1.1之前的，那需要注册 Bean(3.1.1开始不再需要这一步)：

```java
import com.baomidou.mybatisplus.core.injector.ISqlInjector;
import com.baomidou.mybatisplus.extension.injector.LogicSqlInjector;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class MyBatisPlusConfiguration {

    @Bean
    public ISqlInjector sqlInjector() {
        return new LogicSqlInjector();
    }
}
```

4. 测试一下



数据依然在数据库中，但是值发生了变化。

id	name	age	email	version	create_time	update_time	deleted
1	小花	6	486494@qq.com	2	2020-03-20 10:58:16	2020-03-20 14:41:25	1
2	Jack	20	test2@baomidou.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16	0
3	Tom	28	test3@baomidou.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16	0
4	Sandy	21	test4@baomidou.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16	0
5	Billie	24	test5@baomidou.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16	0
1240824931619037186	moon	20	1127397156@qq.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16	0
1240824931619037187	CMY	18	1127397156@qq.com	1	2020-03-20 10:58:16	2020-03-20 10:58:16	0
1240824931619037188	CMY	6	1127397156@qq.com	1	2020-03-20 11:02:39	2020-03-20 11:02:39	0
1240824931619037189	小亮	6	12345645@qq.com	1	2020-03-20 12:01:09	2020-03-20 12:09:32	0

再测试一下查询

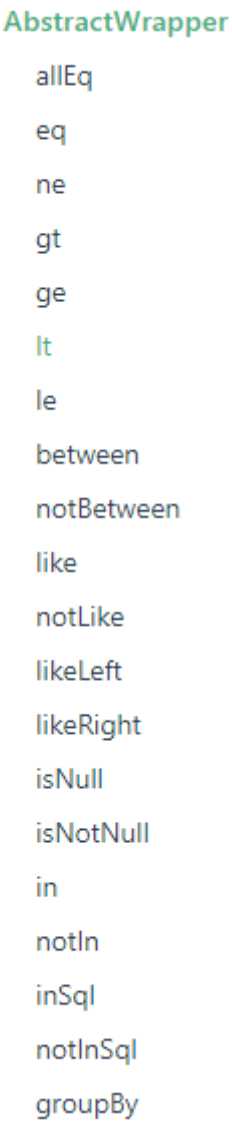
可以看到在查询的时候会自动过滤被逻辑删除的字段。

##

条件构造器

Wrapper

一般写一些复杂的sql就可以使用他来代替。



1. 测试，看输出的SQL进行分析

```
@Test
void test1() {
    // 查询name不为空的用户，并且邮箱不为空的用户，年龄大于等于12
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper
            .isNotNull("name")
            .isNotNull("email")
            .ge("age", 12);
    userMapper.selectList(wrapper).forEach(System.out::println);
}
```

```
ring: SELECT id,name,age,email,version,deleted,create_time,update_time FROM user WHERE deleted=0 AND (name IS NOT NULL AND email IS NOT NULL AND age >= ?)
ters: 12(Integer)
umns: id, name, age, email, version, deleted, create_time, update_time
 Row: 2, Jack, 20, test2@baomidou.com, 1, 0, 2020-03-20 10:58:16, 2020-03-20 10:58:16
 Row: 3, Tom, 28, test3@baomidou.com, 1, 0, 2020-03-20 10:58:16, 2020-03-20 10:58:16
 Row: 4, Sandy, 21, test4@baomidou.com, 1, 0, 2020-03-20 10:58:16, 2020-03-20 10:58:16
 Row: 5, Billie, 24, test5@baomidou.com, 1, 0, 2020-03-20 10:58:16, 2020-03-20 10:58:16
 Row: 1240824931619037186, moon, 20, 1127397156@qq.com, 1, 0, 2020-03-20 10:58:16, 2020-03-20 10:58:16
 Row: 1240824931619037187, CMY, 18, 1127397156@qq.com, 1, 0, 2020-03-20 10:58:16, 2020-03-20 10:58:16
otal: 6
```

2. 测试，看输出的SQL进行分析

```
@Test
void test2() {
    // 查询名字moon
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.eq("name", "moon");
    User user = userMapper.selectOne(wrapper); // 查询一个数据，出现多个结果使用List 或者 Map
    System.out.println(user);
}
```

```
==>  Preparing: SELECT id,name,age,email,version,deleted,create_time,update_time FROM user WHERE deleted=0 AND (name = ?)
==> Parameters: moon(String)
<==    Columns: id, name, age, email, version, deleted, create_time, update_time
<==        Row: 1240824931619037186, moon, 20, 1127397156@qq.com, 1, 0, 2020-03-20 10:58:16, 2020-03-20 10:58:16
<==      Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@7e446d92]
User(id=1240824931619037186, name=moon, age=20, email=1127397156@qq.com, version=1, deleted=0, createTime=Fri Mar 20 10:58:16 CST 2020)
```

3. 测试，看输出的SQL进行分析

```
@Test
void test3() {
    // 查询年龄在 20 ~ 30 岁之间的用户
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    wrapper.between("age", 20, 30); // 区间
    Integer count = userMapper.selectCount(wrapper);// 查询结果数
    System.out.println(count);
}
```

```
  Preparing: SELECT COUNT( 1 ) FROM user WHERE deleted=0 AND (age BETWEEN ? AND ?)
 Parameters: 20(Integer), 30(Integer)
    Columns: COUNT( 1 )
        Row: 5
      Total: 1
osing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@10c72a6f]
```

4. 测试，看输出的SQL进行分析

```
// 模糊查询
 @Test
 void test4(){
     QueryWrapper<User> wrapper = new QueryWrapper<>();
     // 左和右 t%
     wrapper
             .notLike("name","e")
             .likeRight("email","t");
     List<Map<String, Object>> maps = userMapper.selectMaps(wrapper);
     maps.forEach(System.out::println);
 }
```

```
==>  Preparing: SELECT id,name,age,email,version,deleted,create_time,update_time FROM user WHERE deleted=0 AND (name NOT LIKE ? AND email LIKE ?)
==> Parameters: %e%(String), t%(String)
<==    Columns: id, name, age, email, version, deleted, create_time, update_time
<==        Row: 2, Jack, 20, test2@baomidou.com, 1, 0, 2020-03-20 10:58:16, 2020-03-20 10:58:16
<==        Row: 3, Tom, 28, test3@baomidou.com, 1, 0, 2020-03-20 10:58:16, 2020-03-20 10:58:16
<==        Row: 4, Sandy, 21, test4@baomidou.com, 1, 0, 2020-03-20 10:58:16, 2020-03-20 10:58:16
<==      Total: 3
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@3078cac]
{update_time=2020-03-20 10:58:16.0, deleted=0, create_time=2020-03-20 10:58:16.0, name=Jack, id=2, version=1, age=20, email=test2@baomidou.com}
{update_time=2020-03-20 10:58:16.0, deleted=0, create_time=2020-03-20 10:58:16.0, name=Tom, id=3, version=1, age=28, email=test3@baomidou.com}
{update_time=2020-03-20 10:58:16.0, deleted=0, create_time=2020-03-20 10:58:16.0, name=Sandy, id=4, version=1, age=21, email=test4@baomidou.com}
```

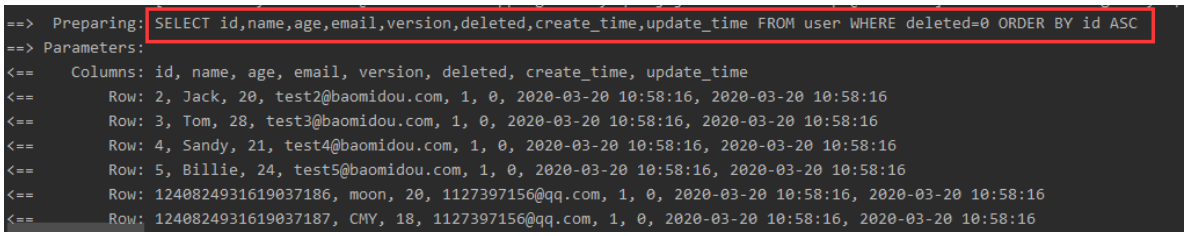

5. 测试，看输出的SQL进行分析

```
// 模糊查询
@Test
void test5(){
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    // id 在子查询中查出来
    wrapper.inSql("id","select id from user where id<3");
    List<Object> objects = userMapper.selectObjs(wrapper);
    objects.forEach(System.out::println);
}
```

```
==>  Preparing: SELECT id,name,age,email,version,deleted,create_time,update_time FROM user WHERE deleted=0 AND (id IN (select id from user where id<3))
==> Parameters:
<==    Columns: id, name, age, email, version, deleted, create_time, update_time
<==        Row: 2, Jack, 20, test2@baomidou.com, 1, 0, 2020-03-20 10:58:16, 2020-03-20 10:58:16
<==      Total: 1
Closing non transactional SqlSession [org.apache.ibatis.session.defaults.DefaultSqlSession@6d5c2745]
2
```


6. 测试，看输出的SQL进行分析

```java
@Test
void test6(){
    QueryWrapper<User> wrapper = new QueryWrapper<>();
    // 通过id进行排序
    wrapper.orderByAsc("id");
    List<User> users = userMapper.selectList(wrapper);
    users.forEach(System.out::println);
}
```



代码自动生成器

dao、pojo、service、controller都自己去编写完成！ 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。

测试：

添加依赖

有的依赖不是必须的，比如lombok,可以根据下面的配置文件来。

```xml
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-generator</artifactId>
    <version>3.3.1.tmp</version>
</dependency>
<dependency>
    <groupId>org.apache.velocity</groupId>
    <artifactId>velocity-engine-core</artifactId>
    <version>2.2</version>
</dependency>

<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.3.1.tmp</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>org.projectlombok</groupId>
```

```
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>

<dependency>
<groupId>org.mybatis.spring.boot</groupId>
<artifactId>mybatis-spring-boot-starter</artifactId>
<version>1.3.1</version>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

下面具体的配置解释可以看官方文档：

```
package com.cc;

import com.baomidou.mybatisplus.annotation.DbType;
import com.baomidou.mybatisplus.annotation.FieldFill;
import com.baomidou.mybatisplus.annotation.IdType;
import com.baomidou.mybatisplus.generator.AutoGenerator;
import com.baomidou.mybatisplus.generator.config.DataSourceConfig;
import com.baomidou.mybatisplus.generator.config.GlobalConfig;
import com.baomidou.mybatisplus.generator.config.PackageConfig;
import com.baomidou.mybatisplus.generator.config.StrategyConfig;
import com.baomidou.mybatisplus.generator.config.po.TableFill;
import com.baomidou.mybatisplus.generator.config.rules.DateType;
import com.baomidou.mybatisplus.generator.config.rules.NamingStrategy;

import java.util.ArrayList;

// 代码自动生成器
public class moonCode {
public static void main(String[] args) {
        // 需要构建一个 代码自动生成器 对象
        AutoGenerator mpg = new AutoGenerator();
        // 配置策略
        // 1、全局配置
        GlobalConfig gc = new GlobalConfig();
        String projectPath = System.getProperty("user.dir");
        gc.setOutputDir(projectPath+"/src/main/java");
        gc.setAuthor("moon");
        gc.setOpen(false);
```

```
        gc.setFileOverride(false); // 是否覆盖
        gc.setServiceName("%sService"); // 去Service的I前缀
        gc.setIdType(IdType.ID_WORKER);
        gc.setDateType(DateType.ONLY_DATE);
        //gc.setSwagger2(true);
        mpg.setGlobalConfig(gc);
        //2、设置数据源
        DataSourceConfig dsc = new DataSourceConfig();
        dsc.setUrl("jdbc:mysql://localhost:3306/mis?
useSSL=true&useUnicode=true&characterEncoding=utf-
8&serverTimezone=Asia/Shanghai");
        dsc.setDriverName("com.mysql.cj.jdbc.Driver");
        dsc.setUsername("root");
        dsc.setPassword("123456");
        dsc.setDbType(DbType.MYSQL);
        mpg.setDataSource(dsc);
        //3、包的配置
        PackageConfig pc = new PackageConfig();
        //pc.setModuleName("user");
        pc.setParent("com.cc");
        pc.setEntity("entity");
        pc.setMapper("mapper");
        pc.setService("service");
        pc.setController("controller");
        mpg.setPackageInfo(pc);
        //4、策略配置
        StrategyConfig strategy = new StrategyConfig();
        strategy.setInclude("student"); // 设置要映射的表名
        strategy.setNaming(NamingStrategy.underline_to_camel);
        strategy.setColumnNaming(NamingStrategy.underline_to_camel);
        strategy.setEntityLombokModel(true); // 自动lombok;
        strategy.setLogicDeleteFieldName("deleted");
        // 自动填充配置
        TableFill gmtCreate = new TableFill("gmt_create", FieldFill.INSERT);
        TableFill gmtModified = new TableFill("gmt_modified",
FieldFill.INSERT_UPDATE);
        ArrayList<TableFill> tableFills = new ArrayList<>();
        tableFills.add(gmtCreate);
        tableFills.add(gmtModified);
        strategy.setTableFillList(tableFills);
        // 乐观锁
        strategy.setVersionFieldName("version");
        strategy.setRestControllerStyle(true);
        strategy.setControllerMappingHyphenStyle(true);
//localhost:8080/hello_id_2
        mpg.setStrategy(strategy);
        mpg.execute(); //执行
    }
}
```