技术架构

## 八种商业模式

### 1、C2C模式（Consumer To Consumer 平台模式 ）

用户到用户，这种模式本质是将自己的流量或者用户转卖给视频或者直播的内容提供者，通过出售内容分成获利。

平台模式避开了非常沉重的内容和服务，扩张迅速，但实际这种模式也有缺陷，在线教育这两年的发展使内容迅速贬值，比较难带来更免费用户和流量。

**代表网站：**

51cto <http://edu.51cto.com/>

腾讯课堂 <https://ke.qq.com/>

### 2、B2C模式（Business To Customer 会员模式）

商家到用户，这种模式是自己制作大量自有版权的视频，放在自有平台上，让用户按月付费或者按年付费。 这种模式简单，快速，只要专心录制大量视频即可快速发展，其曾因为 lynda 的天价融资而 大热。但在中国由于版权保护意识不强，教育内容易于复制，有海量的免费资源的竞争对手众多等原因，难以取得像样的现金流。

**代表网站：**

lynda <https://www.lynda.com/>

慕课网 <https://www.imooc.com/>

谷粒学院 <http://www.gulixueyuan.com/>

### 3、B2B2C（商家到商家到用户）

平台链接第三方教育机构和用户，平台一般不直接提供课程内容，而是更多承担教育的互联网载体角色，为教学过程各个环节提供全方位支持和服务。

**代表网站：**

51cto <http://edu.51cto.com/>

腾讯课堂 <https://ke.qq.com/>

### 4、垂直领域

这种模式需要糅合录播，直播，帮助服务等多种手段，对学生学习某一项内容负责。这种模式收费高，有较强的壁垒。这种产品一旦形成口碑，会有稳定的用户群和收入，但产品非常复杂，难度大，门槛高，即使单独一个项目都会耗费大量的人力物力，因此发展速度较慢。

**代表网站：**

51cto的微职位 <http://edu.51cto.com/>

网易云课堂的微专业 <https://study.163.com/>

### 5、直播、互动

这种模式将传统课堂上的反馈，交互，答疑搬到线上。让用户容易接受，只要服务贴心，用 户就愿意买单，因此有丰富现金流。但缺陷是只能通过平台吸引用户，造成了竞争门槛过低， 模式雷同，对手众多，收益的永远是拥有流量或者用户的大平台。

**代表网站：**

腾讯课堂： <https://ke.qq.com/>

学而思 <https://www.xueersi.com>

### 6、1 对 1

让一个讲师在一定时间内对一个学员进行辅导，学生按照时间支付费用。这种模式收费容易， 现金流好，产品难度不大，市场空间大，但是人力资源的获取消耗却是巨大的，如果师资上控制不好，比如优秀的讲师留不住，或者整体成本太大，都会导致 1 对 1 模式难以发展。

**代表网站：**

VIPKID <https://www.vipkid.com.cn/>

学而思 <https://www.xueersi.com>

### 7、O2O 模式（Online To Offline 线上到线下）

就是通过免费内容或者运营，让线上平台获取用户和流量，将用户吸引到线下开课，或 者让学员到加盟的线下机构上课。这种模式形式简单，收益高，只要把控用户需求，吸引到用户，收费不成问题，而且符合传统的消费习惯。

**代表网站：**

启德教育 <https://www.eic.org.cn/>

### 8、freemium（免费增值）

Freemium最早由AVC的Fred Wilson在2006年提出, 指的是用免费服务吸引用户，然后通过增值服务，将部分免费用户转化为收费用户，实现变现。Freemium模式中有“二八定律”的因素，即一小部分对价格不敏感的高端用户，愿意为一些额 外的功能付费，为服务提供者带来大部分收入。

**代表网站：**

中国大学慕课 <http://www.icourse163.org>

通过免费的名校课程和高校建立合作，吸引用户。提供考研专栏和学校云增值服务

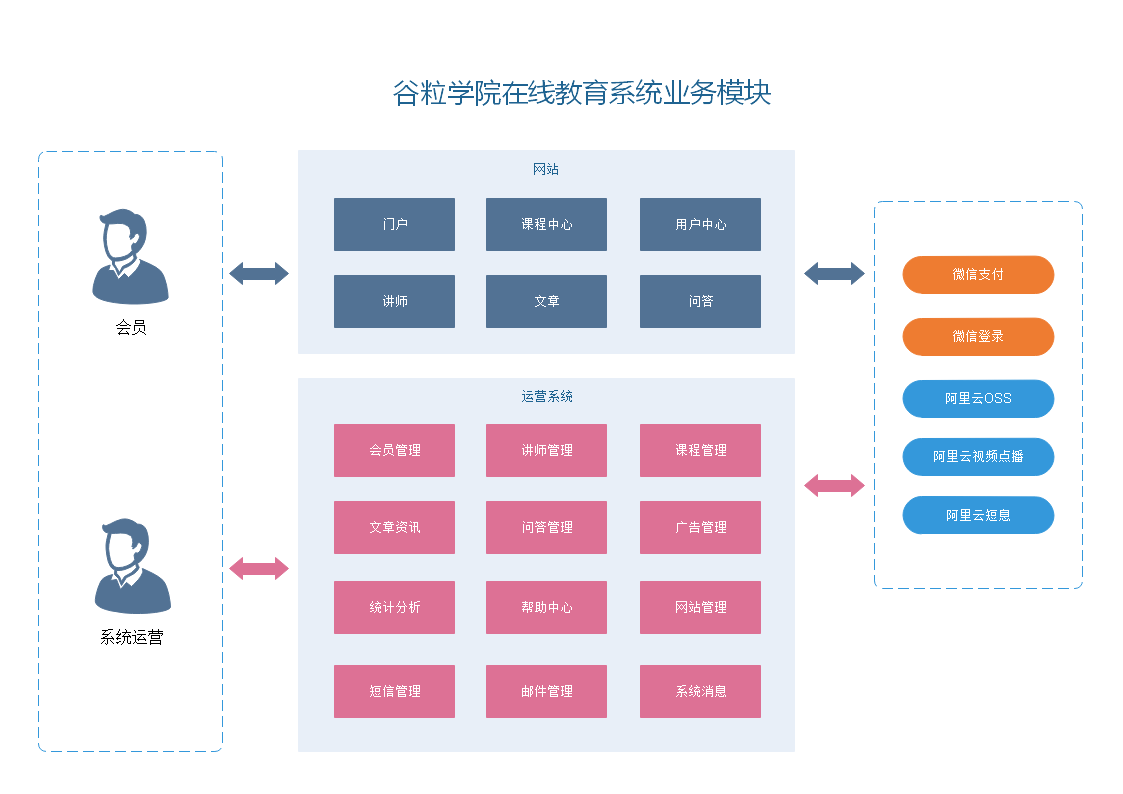
学堂在线 <http://www.xuetangx.com/>

课程免费，如果希望得到课程的认证证书则要缴纳相应的费用

## 一、功能简介

谷粒学院，是一个B2C模式的职业技能在线教育系统，分为前台用户系统和后台运营平台。

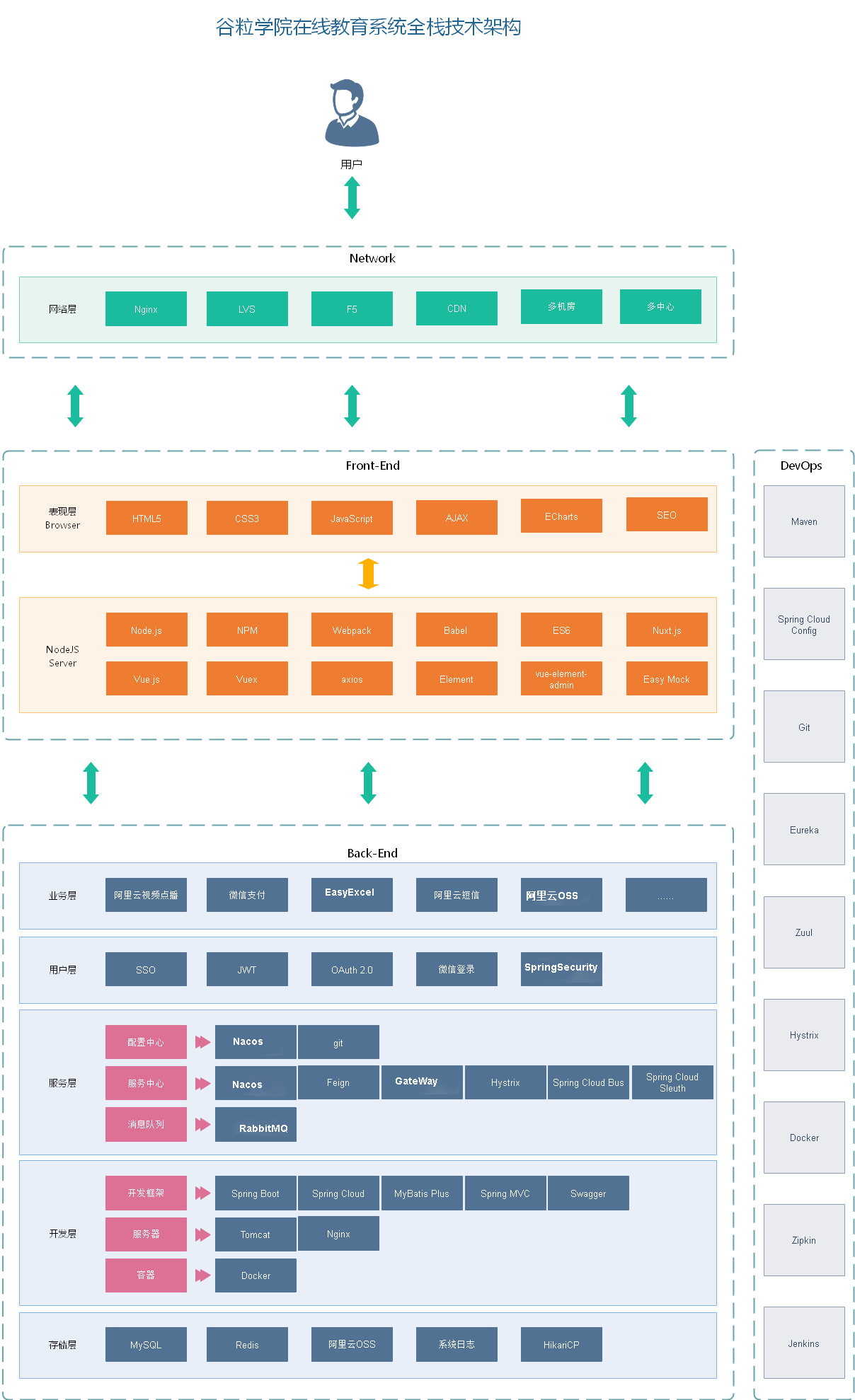
## 二、系统模块



## 三、系统架构

**架构设计需要考虑的几个方面：**

* **性能：**主要考虑访问频率，每个用户每天的访问次数。项目初始阶段用户的访问量并不大，如果考虑做运营推广，可能会迎来服务器访问量骤增，因此要考虑**分布式部署，引入缓存**
* **可扩展性：**系统功能会随着用户量的增加以及多变的互联网用户需求不断地扩展，因此考虑到系统的可扩展性的要求需要**使用微服务架构，引入消息中间件**
* **高可用：**系统一旦宕机，将会带来不可挽回的损失，因此必须做负载均衡，甚至是异地多活这类复杂的方案。如果数据丢失，修复将会非常麻烦，只能靠人工逐条修复，这个很难接受，因此需要考虑存储高可靠。我们需要考虑多种异常情况：机器故障、机房故障，针对机器故障，我们需要设计 MySQL 同机房主备方案；针对机房故障，我们需要设计 MySQL 跨机房同步方案。
* **安全性：**系统的信息有一定的隐私性，例如用户的个人身份信息，不包含强隐私（例如玉照、情感）的信息，因此使用账号密码管理、数据库访问权限控制即可。
* **成本：**视频类网站的主要成本在于服务器成本、流量成本、存储成本、流媒体研发成本，中小型公司可以考虑使用云服务器和云服务。



## 技术栈



MyBatisPlus

**01-MyBatisPlus简介**

### 一、简介

官网：<http://mp.baomidou.com/>

参考教程：<http://mp.baomidou.com/guide/>

[MyBatis-Plus](https://github.com/baomidou/mybatis-plus)（简称 MP）是一个 [MyBatis](http://www.mybatis.org/mybatis-3/) 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。

### 二、特性

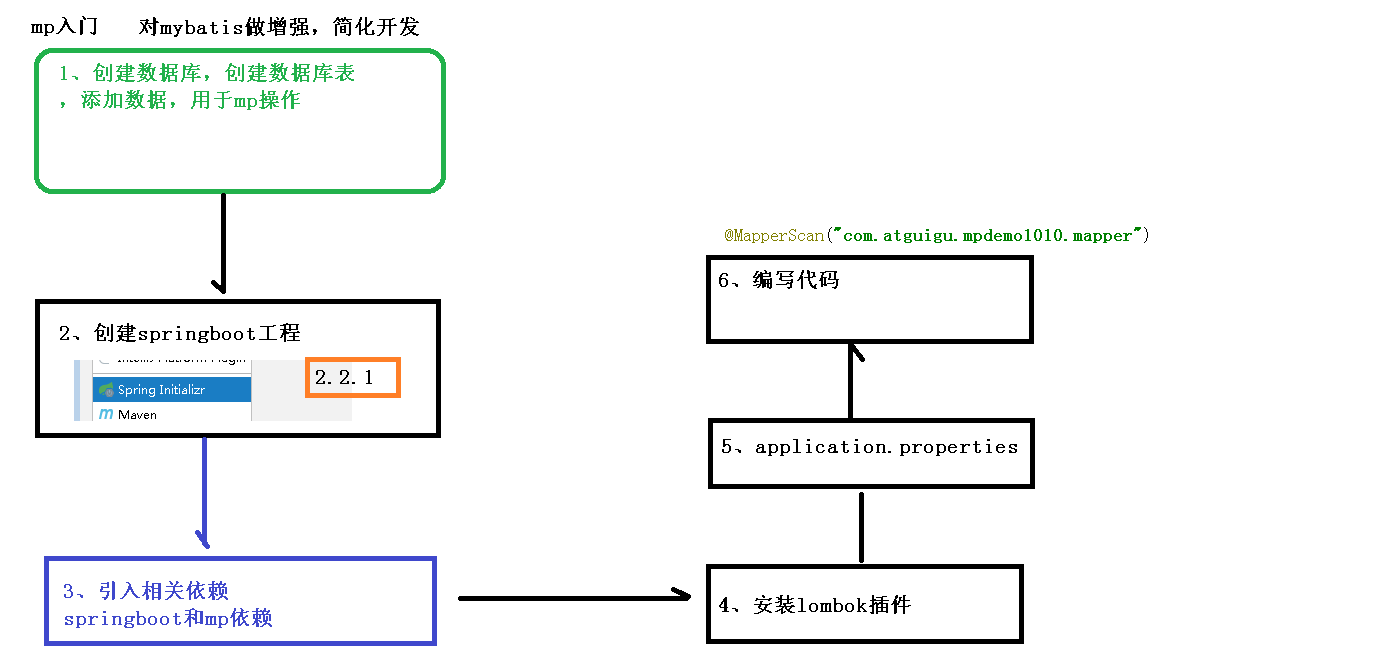
* **无侵入**：只做增强不做改变，引入它不会对现有工程产生影响，如丝般顺滑
* **损耗小**：启动即会自动注入基本 CURD，性能基本无损耗，直接面向对象操作
* **强大的 CRUD 操作**：内置通用 Mapper、通用 Service，仅仅通过少量配置即可实现单表大部分 CRUD 操作，更有强大的条件构造器，满足各类使用需求
* **支持 Lambda 形式调用**：通过 Lambda 表达式，方便的编写各类查询条件，无需再担心字段写错
* **支持多种数据库**：支持 MySQL、MariaDB、Oracle、DB2、H2、HSQL、SQLite、Postgre、SQLServer2005、SQLServer 等多种数据库
* **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 XML 热加载**：Mapper 对应的 XML 支持热加载，对于简单的 CRUD 操作，甚至可以无 XML 启动
* **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 ）
* **支持关键词自动转义**：支持数据库关键词（order、key......）自动转义，还可自定义关键词
* **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
* **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
* **内置 Sql 注入剥离器**：支持 Sql 注入剥离，有效预防 Sql 注入攻击

## 02-MyBatisPlus入门

快速开始参考：<http://mp.baomidou.com/guide/quick-start.html>

测试项目： mybatis\_plus

数据库：mybatis\_plus



### 一、创建并初始化数据库

#### 1、创建数据库：

CREATE DATABASE mybatis\_plus;

#### 2、创建 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：

USE mybatis\_plus;

DROP TABLE IF EXISTS user;  
CREATE TABLE user  
*(* id BIGINT*(*20*)* NOT NULL COMMENT '主键ID' PRIMARY KEY AUTO\_INCREMENT,  
 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'  
  
*)*;

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'*)*;

### 二、初始化工程

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

Group：com.atguigu  
Artifact：mybatis-plus  
版本：2.2.1.RELEASE

### 三、添加依赖

#### 1、引入依赖

spring-boot-starter、spring-boot-starter-test

添加：mybatis-plus-boot-starter、MySQL、lombok、

在项目中使用Lombok可以减少很多重复代码的书写。比如说getter/setter/toString等方法的编写

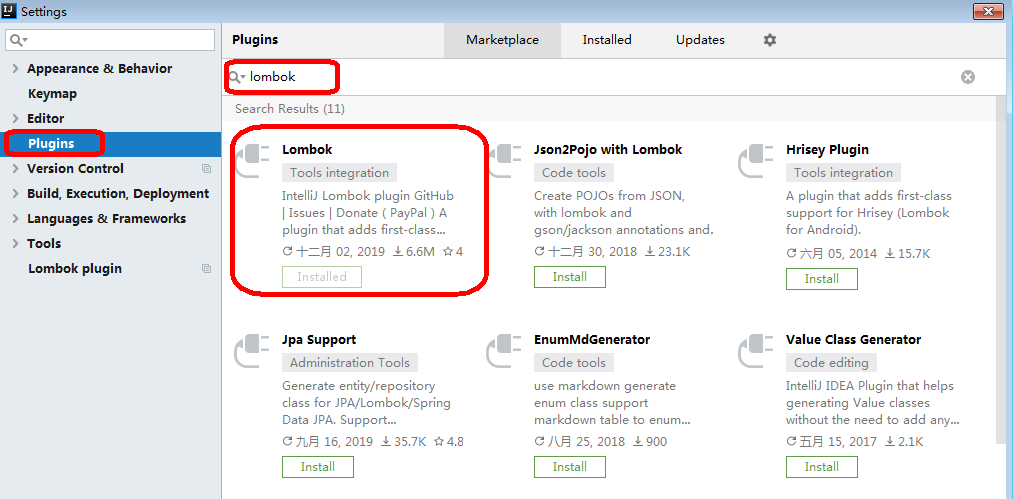
< *<*dependencies*>*

*<*dependency*>  
 <*groupId*>*org.springframework.boot*</*groupId*>  
 <*artifactId*>*spring-boot-starter*</*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*>* <!--mybatis-plus-->  
 *<*dependency*>  
 <*groupId*>*com.baomidou*</*groupId*>  
 <*artifactId*>*mybatis-plus-boot-starter*</*artifactId*>  
 <*version*>*3.0.5*</*version*>  
 </*dependency*>* <!--mysql-->  
 *<*dependency*>  
 <*groupId*>*mysql*</*groupId*>  
 <*artifactId*>*mysql-connector-java*</*artifactId*>  
 </*dependency*>* <!--lombok用来简化实体类-->  
 *<*dependency*>  
 <*groupId*>*org.projectlombok*</*groupId*>  
 <*artifactId*>*lombok*</*artifactId*>  
 </*dependency*>  
</*dependencies*>*

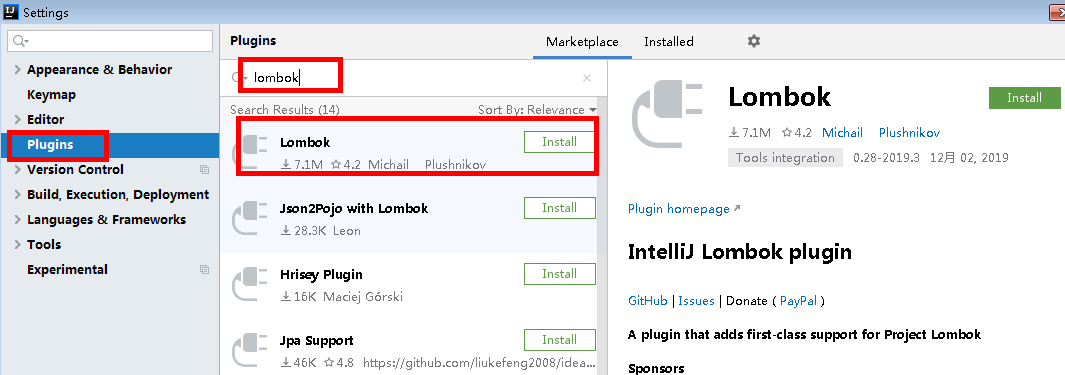
**注意：**引入 MyBatis-Plus 之后请不要再次引入 MyBatis 以及 MyBatis-Spring，以避免因版本差异导致的问题。

#### 2、idea中安装lombok插件

**（1）idea2018版本**



**（2）idea2019版本**



### 四、配置

在 application.properties 配置文件中添加 MySQL 数据库的相关配置：

mysql5

#mysql数据库连接

spring.datasource.driver-class-name=com.mysql.jdbc.Driver  
spring.datasource.url=jdbc:mysql://localhost:3306/mybatis\_plus  
spring.datasource.username=root  
spring.datasource.password=123456

mysql8以上（spring boot 2.1）

**注意：**driver和url的变化

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/mybatis\_plus?serverTimezone=GMT%2B8  
spring.datasource.username=root  
spring.datasource.password=123456

**注意：**

1、这里的 url 使用了 ?serverTimezone=GMT%2B8 后缀，因为Spring Boot 2.1 集成了 8.0版本的jdbc驱动，这个版本的 jdbc 驱动需要添加这个后缀，否则运行测试用例报告如下错误：

java.sql.SQLException: The server time zone value 'ÖÐ¹ú±ê×¼Ê±¼ä' is unrecognized or represents more

2、这里的 driver-class-name 使用了  com.mysql.cj.jdbc.Driver ，在 jdbc 8 中 建议使用这个驱动，之前的 com.mysql.jdbc.Driver 已经被废弃，否则运行测试用例的时候会有 WARN 信息

#### 汇总配置

mybatis-plus:

*#mapper-locations:classpath:mybatis/\*\*/\*Mapper.xml  
 # 在classpath前添加星号可以使项目热加载成功* mapper-locations:classpath\*:mybatis/\*\*/\*Mapper.xml  
 *#实体扫描，多个package用逗号或者分号分隔* typeAliasesPackage:com.nis.project  
 global-config:  
 *#主键类型 0:"数据库ID自增",1:"用户输入ID",2:"全局唯一ID (数字类型唯一ID)",3:"全局唯一ID UUID";* id-type:3  
 *#机器 ID 部分(影响雪花ID)* workerId:1  
 *#数据标识 ID 部分(影响雪花ID)(workerId 和 datacenterId 一起配置才能重新初始化 Sequence)* datacenterId:18  
 *#字段策略 0:"忽略判断",1:"非 NULL 判断"),2:"非空判断"* field-strategy:2  
 *#驼峰下划线转换* db-column-underline:true  
 *#刷新mapper 调试神器* refresh-mapper:true  
 *#数据库大写下划线转换  
 #capital-mode:true  
 #序列接口实现类配置  
 #key-generator:com.baomidou.springboot.xxx  
 #逻辑删除配置（下面3个配置）* logic-delete-value:0  
 logic-not-delete-value:1  
 *#自定义SQL注入器  
 #sql-injector:com.baomidou.mybatisplus.mapper.LogicSqlInjector  
 #自定义填充策略接口实现  
 #meta-object-handler:com.baomidou.springboot.xxx* configuration:  
 map-underscore-to-camel-case:true  
 cache-enabled:false  
 *# 这个配置会将执行的sql打印出来，在开发或测试的时候可以用* log-impl:org.apache.ibatis.logging.stdout.StdOutImpl

### 五、编写代码

#### 1、主类

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

**注意：**扫描的包名根据实际情况修改

@SpringBootApplication

@MapperScan*(*"com.atguigu.mybatisplus.mapper"*)*public class MybatisPlusApplication *{* ......  
*}*

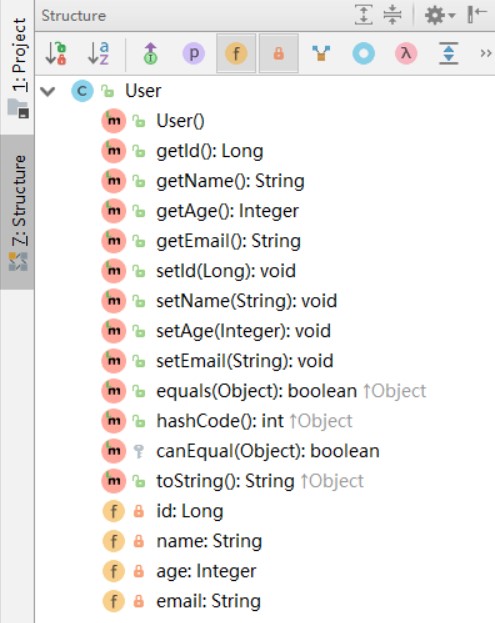
#### 2、实体

创建包 entity 编写实体类 User.java（此处使用了 [Lombok](https://www.projectlombok.org/) 简化代码）

@Data

public class User *{* private Long id;  
 private String name;  
 private Integer age;  
 private String email;  
*}*

查看编译结果



Lombok使用参考：

<https://blog.csdn.net/motui/article/details/79012846>

#### 3、mapper

创建包 mapper 编写Mapper 接口： UserMapper.java

public interface UserMapper extends BaseMapper*<*User*> {*

*}*

### 六、开始使用

添加测试类，进行功能测试：

@RunWith*(*SpringRunner.class*)*

@SpringBootTest  
public class MybatisPlusApplicationTests *{* @Autowired  
 private UserMapper userMapper;  
  
 @Test  
 public void testSelectList*() {* System.*out*.println*((*"----- selectAll method test ------"*))*;  
 //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  
 List*<*User*>* users = userMapper.selectList*(*null*)*;  
 users.forEach*(*System.*out*::println*)*;  
 *}  
}*

**注意：**

IDEA在 userMapper 处报错，因为找不到注入的对象，因为类是动态创建的，但是程序可以正确的执行。

为了避免报错，可以在 dao 层 的接口上添加 @Repository 注

控制台输出：

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)

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！

### 七、配置日志

查看sql输出日志

#mybatis日志

mybatis-plus.configuration.log-impl=org.apache.ibatis.logging.stdout.StdOutImpl

## 03-MyBatisPlus的CRUD 接口

### insert

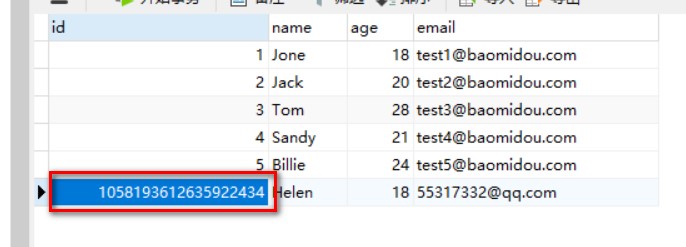


#### 1、插入操作

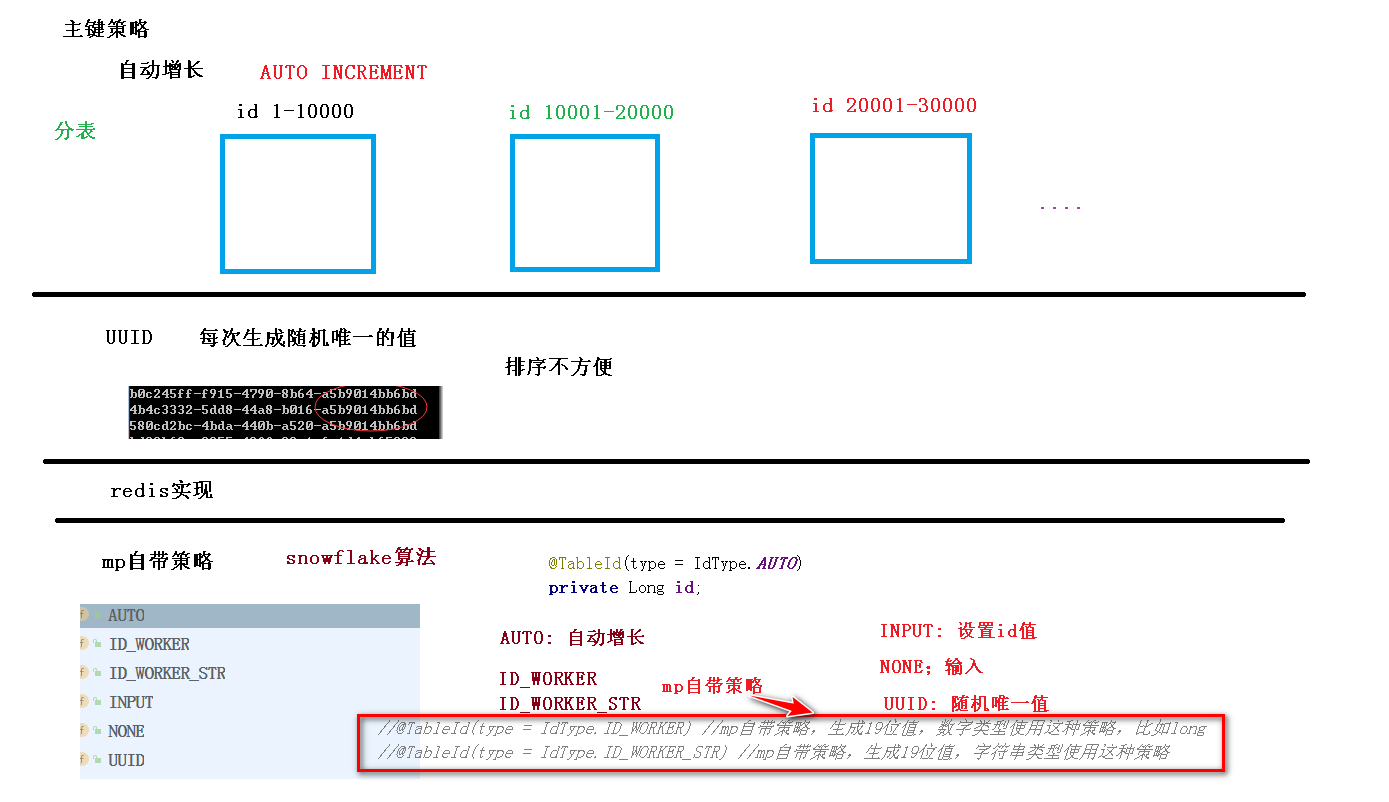
@SpringBootTest

public class CRUDTests *{* @Autowired  
 private UserMapper userMapper;  
  
 @Test  
 public void testInsert*() {* User user = new User*()*;  
 user.setName*(*"Helen"*)*;  
 user.setAge*(*18*)*;  
 user.setEmail*(*"55317332@qq.com"*)*;  
 int result = userMapper.insert*(*user*)*;  
 System.*out*.println*(*result*)*; //影响的行数  
 System.*out*.println*(*user*)*; //id自动回填  
 *}  
}*

**注意：**数据库插入id值默认为：全局唯一id



#### 2、主键策略



**（1）ID\_WORKER**

MyBatis-Plus默认的主键策略是：ID\_WORKER  *全局唯一ID 【雪花算法】*

ID\_WORKER\_STR是ID\_WORKER的字符串形式。

**参考资料：分布式系统唯一ID生成方案汇总：**<https://www.cnblogs.com/haoxinyue/p/5208136.html>

**（2）自增策略**

* 要想主键自增需要配置如下主键策略
  + 需要在创建数据表的时候设置主键自增
  + 实体字段中配置 @TableId(type = IdType.AUTO)

@TableId*(*type = IdType.*AUTO)*

private Long id;

要想影响所有实体的配置，可以设置全局主键配置

#全局设置主键生成策略

mybatis-plus.global-config.db-config.id-type=*auto*

其它主键策略：分析 IdType 源码可知

@Getter

public enum IdType *{  
 /\*\*  
 \* 数据库ID自增  
 \*/  
 AUTO(*0*)*,  
 */\*\*  
 \* 该类型为未设置主键类型  
 \*/  
 NONE(*1*)*,  
 */\*\*  
 \* 用户输入ID  
 \* 该类型可以通过自己注册自动填充插件进行填充  
 \*/  
 INPUT(*2*)*,  
 /\* 以下3种类型、只有当插入对象ID 为空，才自动填充。 \*/  
 */\*\*  
 \* 全局唯一ID (idWorker)  
 \*/  
 ID\_WORKER(*3*)*,  
 */\*\*  
 \* 全局唯一ID (UUID)  
 \*/  
 UUID(*4*)*,  
 */\*\*  
 \* 字符串全局唯一ID (idWorker 的字符串表示)  
 \*/  
 ID\_WORKER\_STR(*5*)*;  
 private int key;  
  
 IdType*(*int key*) {* this.key = key;  
 *}  
}*

相关配置参考 汇总配置 【点击跳转】

#### 拓展：[分布**式系统唯一ID生成方案汇总**](https://www.cnblogs.com/haoxinyue/p/5208136.html)

系统唯一ID是我们在设计一个系统的时候常常会遇见的问题，也常常为这个问题而纠结。生成ID的方法有很多，适应不同的场景、需求以及性能要求。所以有些比较复杂的系统会有多个ID生成的策略。下面就介绍一些常见的ID生成策略。

**1. 数据库自增长序列或字段**

最常见的方式。利用数据库，全数据库唯一。

优点：

1）简单，代码方便，性能可以接受。

2）数字ID天然排序，对分页或者需要排序的结果很有帮助。

缺点：

1）不同数据库语法和实现不同，数据库迁移的时候或多数据库版本支持的时候需要处理。

2）在单个数据库或读写分离或一主多从的情况下，只有一个主库可以生成。有单点故障的风险。

3）在性能达不到要求的情况下，比较难于扩展。（不适用于海量高并发）

4）如果遇见多个系统需要合并或者涉及到数据迁移会相当痛苦。

5）分表分库的时候会有麻烦。

6）并非一定连续，类似MySQL，当生成新ID的事务回滚，那么后续的事务也不会再用这个ID了。这个在性能和连续性的折中。如果为了保证连续，必须要在事务结束后才能生成ID，那性能就会出现问题。

7）在分布式数据库中，如果采用了自增主键的话，有可能会带来尾部热点。分布式数据库常常使用range的分区方式，在大量新增记录的时候，IO会集中在一个分区上，造成热点数据。

优化方案：

1）针对主库单点，如果有多个Master库，则每个Master库设置的起始数字不一样，步长一样，可以是Master的个数。比如：Master1 生成的是 1，4，7，10，Master2生成的是2,5,8,11 Master3生成的是 3,6,9,12。这样就可以有效生成集群中的唯一ID，也可以大大降低ID生成数据库操作的负载。

**2. UUID**

常见的方式。可以利用数据库也可以利用程序生成，一般来说全球唯一。UUID是由32个的16进制数字组成，所以每个UUID的长度是128位（16^32 = 2^128）。UUID作为一种广泛使用标准，有多个实现版本，影响它的因素包括时间、网卡MAC地址、自定义Namesapce等等。

优点：

1）简单，代码方便。

2）生成ID性能非常好，基本不会有性能问题。

3）全球唯一，在遇见数据迁移，系统数据合并，或者数据库变更等情况下，可以从容应对。

缺点：

1）没有排序，无法保证趋势递增。

2）UUID往往是使用字符串存储，查询的效率比较低。

3）存储空间比较大，如果是海量数据库，就需要考虑存储量的问题。

4）传输数据量大

5）不可读。

**3. UUID的变种**

1）为了解决UUID不可读，可以使用UUID to Int64的方法。及

/// <summary>

/// 根据GUID获取唯一数字序列

/// </summary>

public static long GuidToInt64()

{

byte[] bytes = Guid.NewGuid().ToByteArray();

return BitConverter.ToInt64(bytes, 0);

}

2）为了解决UUID无序的问题，NHibernate在其主键生成方式中提供了Comb算法（combined guid/timestamp）。保留GUID的10个字节，用另6个字节表示GUID生成的时间（DateTime）。

/// <summary>

/// Generate a new <see cref="Guid"/> using the comb algorithm.

/// </summary>

private Guid GenerateComb()

{

byte[] guidArray = Guid.NewGuid().ToByteArray();

DateTime baseDate = new DateTime(1900, 1, 1);

DateTime now = DateTime.Now;

// Get the days and milliseconds which will be used to build

//the byte string

TimeSpan days = new TimeSpan(now.Ticks - baseDate.Ticks);

TimeSpan msecs = now.TimeOfDay;

// Convert to a byte array

// Note that SQL Server is accurate to 1/300th of a

// millisecond so we divide by 3.333333

byte[] daysArray = BitConverter.GetBytes(days.Days);

byte[] msecsArray = BitConverter.GetBytes((long)

(msecs.TotalMilliseconds / 3.333333));

// Reverse the bytes to match SQL Servers ordering

Array.Reverse(daysArray);

Array.Reverse(msecsArray);

// Copy the bytes into the guid

Array.Copy(daysArray, daysArray.Length - 2, guidArray,

guidArray.Length - 6, 2);

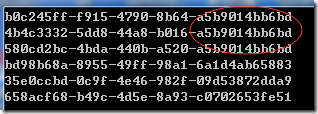
Array.Copy(msecsArray, msecsArray.Length - 4, guidArray,

guidArray.Length - 4, 4);

return new Guid(guidArray);

}

用上面的算法测试一下，得到如下的结果：作为比较，前面3个是使用COMB算法得出的结果，最后12个字符串是时间序（统一毫秒生成的3个UUID），过段时间如果再次生成，则12个字符串会比图示的要大。后面3个是直接生成的GUID。

[](http://images2015.cnblogs.com/blog/15700/201602/15700-20160227213048174-1443183768.png)

如果想把时间序放在前面，可以生成后改变12个字符串的位置，也可以修改算法类的最后两个Array.Copy。

**4. Redis生成ID**

当使用数据库来生成ID性能不够要求的时候，我们可以尝试使用Redis来生成ID。这主要依赖于Redis是单线程的，所以也可以用生成全局唯一的ID。可以用Redis的原子操作 INCR和INCRBY来实现。

可以使用Redis集群来获取更高的吞吐量。假如一个集群中有5台Redis。可以初始化每台Redis的值分别是1,2,3,4,5，然后步长都是5。各个Redis生成的ID为：

A：1,6,11,16,21

B：2,7,12,17,22

C：3,8,13,18,23

D：4,9,14,19,24

E：5,10,15,20,25

这个，随便负载到哪个机确定好，未来很难做修改。但是3-5台服务器基本能够满足器上，都可以获得不同的ID。但是步长和初始值一定需要事先需要了。使用Redis集群也可以方式单点故障的问题。

另外，比较适合使用Redis来生成每天从0开始的流水号。比如订单号=日期+当日自增长号。可以每天在Redis中生成一个Key，使用INCR进行累加。

优点：

1）不依赖于数据库，灵活方便，且性能优于数据库。

2）数字ID天然排序，对分页或者需要排序的结果很有帮助。

缺点：

1）如果系统中没有Redis，还需要引入新的组件，增加系统复杂度。

2）需要编码和配置的工作量比较大。

**5. Twitter的snowflake算法**

snowflake是Twitter开源的分布式ID生成算法，结果是一个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。具体实现的代码可以参看<https://github.com/twitter/snowflake>。雪花算法支持的TPS可以达到419万左右（2^22\*1000）。

雪花算法在工程实现上有单机版本和分布式版本。单机版本如下，分布式版本可以参看美团leaf算法：https://github.com/Meituan-Dianping/Leaf

C#代码如下：

/// <summary>

/// From: https://github.com/twitter/snowflake

/// An object that generates IDs.

/// This is broken into a separate class in case

/// we ever want to support multiple worker threads

/// per process

/// </summary>

public class IdWorker

{

private long workerId;

private long datacenterId;

private long sequence = 0L;

private static long twepoch = 1288834974657L;

private static long workerIdBits = 5L;

private static long datacenterIdBits = 5L;

private static long maxWorkerId = -1L ^ (-1L << (int)workerIdBits);

private static long maxDatacenterId = -1L ^ (-1L << (int)datacenterIdBits);

private static long sequenceBits = 12L;

private long workerIdShift = sequenceBits;

private long datacenterIdShift = sequenceBits + workerIdBits;

private long timestampLeftShift = sequenceBits + workerIdBits + datacenterIdBits;

private long sequenceMask = -1L ^ (-1L << (int)sequenceBits);

private long lastTimestamp = -1L;

private static object syncRoot = new object();

public IdWorker(long workerId, long datacenterId)

{

// sanity check for workerId

if (workerId > maxWorkerId || workerId < 0)

{

throw new ArgumentException(string.Format("worker Id can't be greater than %d or less than 0", maxWorkerId));

}

if (datacenterId > maxDatacenterId || datacenterId < 0)

{

throw new ArgumentException(string.Format("datacenter Id can't be greater than %d or less than 0", maxDatacenterId));

}

this.workerId = workerId;

this.datacenterId = datacenterId;

}

public long nextId()

{

lock (syncRoot)

{

long timestamp = timeGen();

if (timestamp < lastTimestamp)

{

throw new ApplicationException(string.Format("Clock moved backwards. Refusing to generate id for %d milliseconds", lastTimestamp - timestamp));

}

if (lastTimestamp == timestamp)

{

sequence = (sequence + 1) & sequenceMask;

if (sequence == 0)

{

timestamp = tilNextMillis(lastTimestamp);

}

}

else

{

sequence = 0L;

}

lastTimestamp = timestamp;

return ((timestamp - twepoch) << (int)timestampLeftShift) | (datacenterId << (int)datacenterIdShift) | (workerId << (int)workerIdShift) | sequence;

}

}

protected long tilNextMillis(long lastTimestamp)

{

long timestamp = timeGen();

while (timestamp <= lastTimestamp)

{

timestamp = timeGen();

}

return timestamp;

}

protected long timeGen()

{

return (long)(DateTime.UtcNow - new DateTime(1970, 1, 1, 0, 0, 0, DateTimeKind.Utc)).TotalMilliseconds;

}

}

测试代码如下：

private static void TestIdWorker()

{

HashSet<long> set = new HashSet<long>();

IdWorker idWorker1 = new IdWorker(0, 0);

IdWorker idWorker2 = new IdWorker(1, 0);

Thread t1 = new Thread(() => DoTestIdWoker(idWorker1, set));

Thread t2 = new Thread(() => DoTestIdWoker(idWorker2, set));

t1.IsBackground = true;

t2.IsBackground = true;

t1.Start();

t2.Start();

try

{

Thread.Sleep(30000);

t1.Abort();

t2.Abort();

}

catch (Exception e)

{

}

Console.WriteLine("done");

}

private static void DoTestIdWoker(IdWorker idWorker, HashSet<long> set)

{

while (true)

{

long id = idWorker.nextId();

if (!set.Add(id))

{

Console.WriteLine("duplicate:" + id);

}

Thread.Sleep(1);

}

}

snowflake算法可以根据自身项目的需要进行一定的修改。比如估算未来的数据中心个数，每个数据中心的机器数以及统一毫秒可以能的并发数来调整在算法中所需要的bit数。

优点：

1）不依赖于数据库，灵活方便，且性能优于数据库。

2）ID按照时间在单机上是递增的。

缺点：

1）在单机上是递增的，但是由于涉及到分布式环境，每台机器上的时钟不可能完全同步，在算法上要解决时间回拨的问题。

**6. 利用zookeeper生成唯一ID**

zookeeper主要通过其znode数据版本来生成序列号，可以生成32位和64位的数据版本号，客户端可以使用这个版本号来作为唯一的序列号。

很少会使用zookeeper来生成唯一ID。主要是由于需要依赖zookeeper，并且是多步调用API，如果在竞争较大的情况下，需要考虑使用分布式锁。因此，性能在高并发的分布式环境下，也不甚理想。

7. MongoDB的ObjectId

MongoDB的ObjectId和snowflake算法类似。它设计成轻量型的，不同的机器都能用全局唯一的同种方法方便地生成它。MongoDB 从一开始就设计用来作为分布式数据库，处理多个节点是一个核心要求。使其在分片环境中要容易生成得多。

其格式如下：



前4 个字节是从标准纪元开始的时间戳，单位为秒。时间戳，与随后的5 个字节组合起来，提供了秒级别的唯一性。由于时间戳在前，这意味着ObjectId 大致会按照插入的顺序排列。这对于某些方面很有用，如将其作为索引提高效率。这4 个字节也隐含了文档创建的时间。绝大多数客户端类库都会公开一个方法从ObjectId 获取这个信息。  
接下来的3 字节是所在主机的唯一标识符。通常是机器主机名的散列值。这样就可以确保不同主机生成不同的ObjectId，不产生冲突。  
为了确保在同一台机器上并发的多个进程产生的ObjectId 是唯一的，接下来的两字节来自产生ObjectId 的进程标识符（PID）。  
前9 字节保证了同一秒钟不同机器不同进程产生的ObjectId 是唯一的。后3 字节就是一个自动增加的计数器，确保相同进程同一秒产生的ObjectId 也是不一样的。同一秒钟最多允许每个进程拥有2563（16 777 216）个不同的ObjectId。

实现的源码可以到MongoDB官方网站下载。

8. TiDB的主键

TiDB默认是支持自增主键的，对未声明主键的表，会提供了一个隐式主键\_tidb\_rowid，因为这个主键大体上是单调递增的，所以也会出现我们前面说的“尾部热点”问题。

TiDB也提供了UUID函数，而且在4.0版本中还提供了另一种解决方案AutoRandom。TiDB 模仿MySQL的 AutoIncrement，提供了AutoRandom关键字用于生成一个随机ID填充指定列。

### 二、update

#### 1、根据Id更新操作

**注意：**update时生成的sql自动是动态sql：UPDATE user SET age=? WHERE id=?

@Test

public void testUpdateById*() {* User user = new User*()*;  
 user.setId*(*1L*)*;  
 user.setAge*(*28*)*;  
 int result = userMapper.updateById*(*user*)*;  
 System.*out*.println*(*result*)*;  
*}*

#### 2、自动填充

项目中经常会遇到一些数据，每次都使用相同的方式填充，例如记录的创建时间，更新时间等。

我们可以使用MyBatis Plus的自动填充功能，完成这些字段的赋值工作：



**（1）数据库表中添加自动填充字段**

在User表中添加datetime类型的新的字段 create\_time、update\_time

**（2）实体上添加注解**

@Data

public class User *{*...

@TableField*(*fill = FieldFill.*INSERT)* private Date createTime;//对应create\_time  
 //@TableField(fill = FieldFill.UPDATE)  
 @TableField*(*fill = FieldFill.*INSERT\_UPDATE)* private Date updateTime;

...

*}*

**（3）实现元对象处理器接口**

**注意：不要忘记添加 @Component 注解**

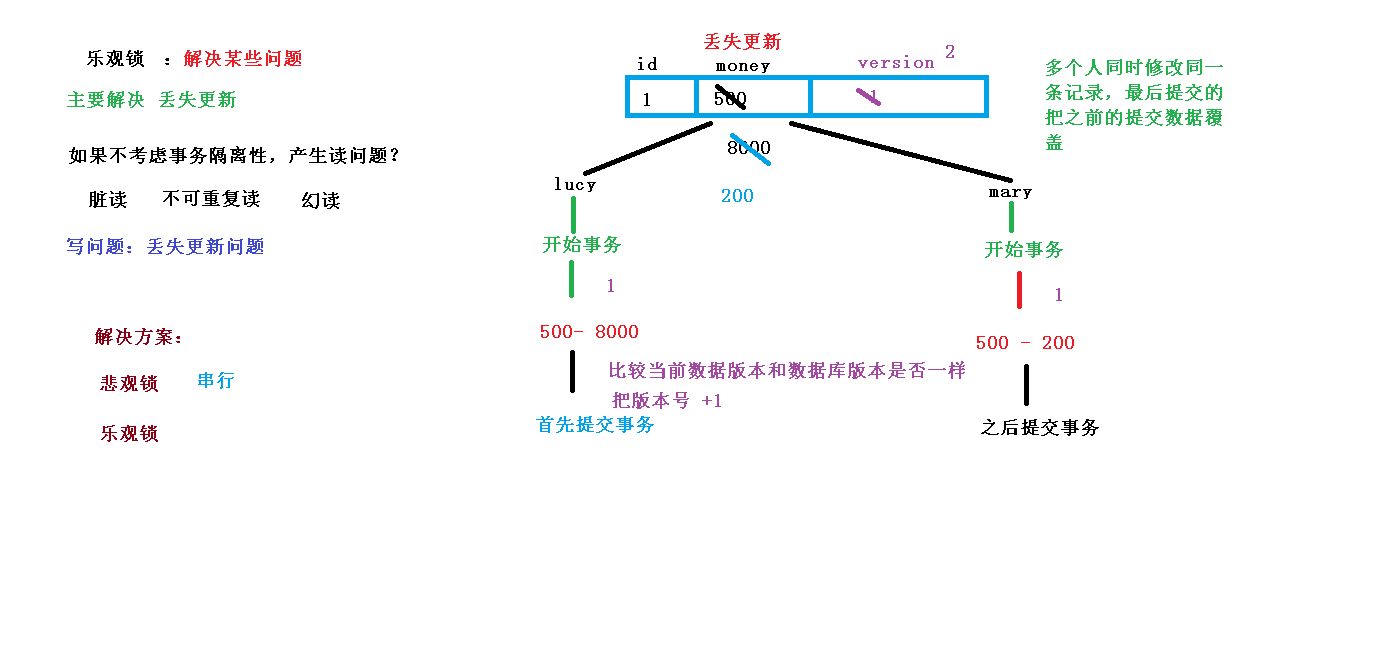
package com.atguigu.mybatisplus.handler;

import com.baomidou.mybatisplus.core.handlers.MetaObjectHandler;  
import org.apache.ibatis.reflection.MetaObject;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.stereotype.Component;  
  
import java.util.Date;  
  
@Component  
public class MyMetaObjectHandler implements MetaObjectHandler *{* private static final Logger *LOGGER* = LoggerFactory.*getLogger(*MyMetaObjectHandler.class*)*;  
  
 @Override  
 public void insertFill*(*MetaObject metaObject*) {  
 LOGGER*.info*(*"start insert fill ...."*)*;  
 this.setFieldValByName*(*"createTime", new Date*()*, metaObject*)*;  
 this.setFieldValByName*(*"updateTime", new Date*()*, metaObject*)*;  
 *}* @Override  
 public void updateFill*(*MetaObject metaObject*) {  
 LOGGER*.info*(*"start update fill ...."*)*;  
 this.setFieldValByName*(*"updateTime", new Date*()*, metaObject*)*;  
 *}  
}*

**（4）测试**

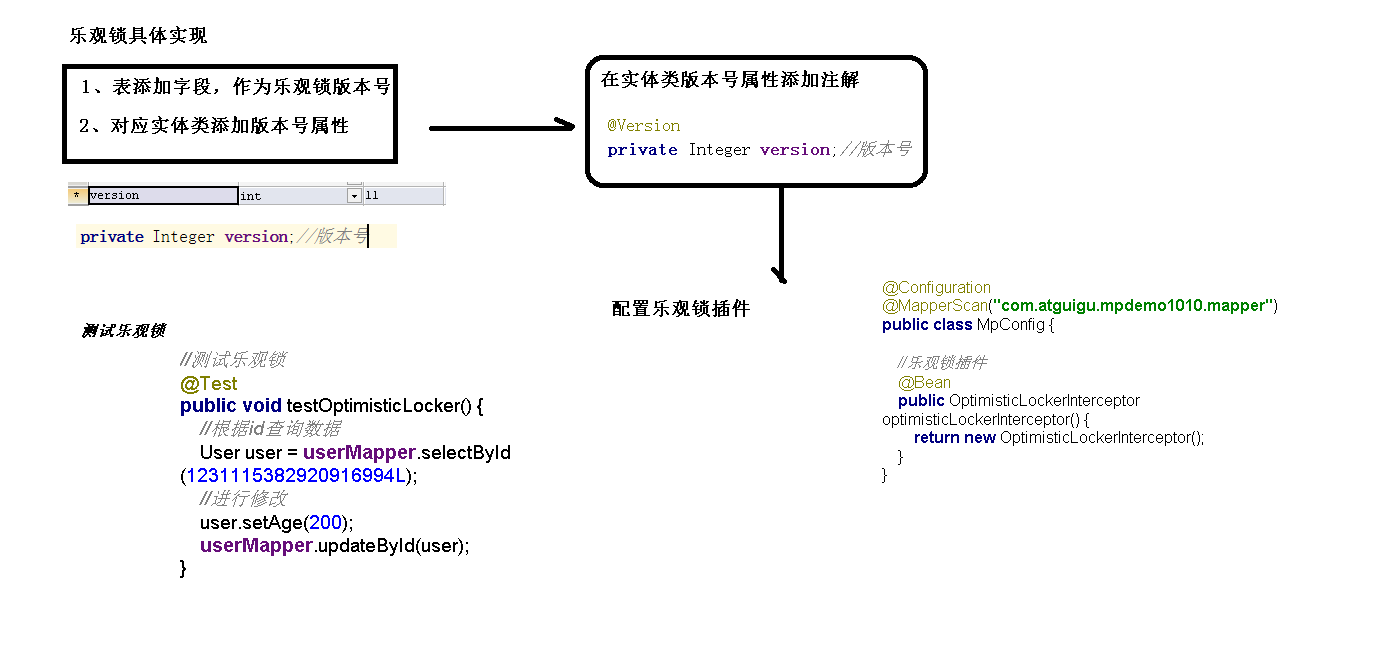
#### 3、乐观锁

**主要适用场景：**当要更新一条记录的时候，希望这条记录没有被别人更新，也就是说实现线程安全的数据更新



**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



**（1）数据库中添加version字段**

ALTER TABLE `user` ADD COLUMN `version` INT

C:\Users\gouwenyong0000\Documents\My Knowledge\temp\4582cfd6-5fb7-4336-b5b2-cfb54f21d292\128\index_files\7bf260f8-d483-49fe-b448-b2fbea3dddaf.png

**（2）实体类添加version字段**

并添加 @Version 注解

@Version

@TableField*(*fill = FieldFill.*INSERT)//为了第一次填充默认值*private Integer version;

**（3）元对象处理器接口添加version的insert默认值**

@Override

public void insertFill*(*MetaObject metaObject*) {*......

this.setFieldValByName*(*"version",1,metaObject*)*;//第一插入式添加默认版本1  
*}*

**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!!!

**（4）在 MybatisPlusConfig 中注册 Bean**

创建配置类

package com.atguigu.mybatisplus.config;

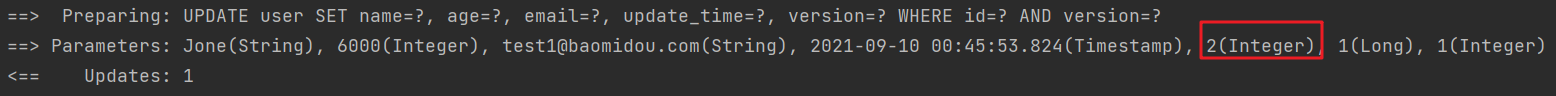
import com.baomidou.mybatisplus.extension.plugins.OptimisticLockerInterceptor;  
import com.baomidou.mybatisplus.extension.plugins.PaginationInterceptor;  
import org.mybatis.spring.annotation.MapperScan;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.transaction.annotation.EnableTransactionManagement;  
  
@EnableTransactionManagement  
@Configuration  
@MapperScan*(*"com.atguigu.mybatis\_plus.mapper"*)*public class MybatisPlusConfig *{  
 /\*\*  
 \* 乐观锁插件  
 \*/* @Bean  
 public OptimisticLockerInterceptor optimisticLockerInterceptor*() {* return new OptimisticLockerInterceptor*()*;  
 *}  
}*

**（5）测试乐观锁可以修改成功**

测试后分析打印的sql语句，将version的数值进行了加1操作

*/\*\**

*\* 测试 乐观锁插件  
 \*/*@Test  
public void testOptimisticLocker*() {* //查询  
 User user = userMapper.selectById*(*1L*)*;  
 //修改数据  
 user.setName*(*"Helen Yao"*)*;  
 user.setEmail*(*"helen@qq.com"*)*;  
 //执行更新  
 userMapper.updateById*(*user*)*;  
*}*



**（5）测试乐观锁修改失败**

*/\*\**

*\* 测试乐观锁插件 失败  
 \*/*@Test  
public void testOptimisticLockerFail*() {* //查询  
 User user = userMapper.selectById*(*1L*)*;  
 //修改数据  
 user.setName*(*"Helen Yao1"*)*;  
 user.setEmail*(*"helen@qq.com1"*)*;  
 //模拟取出数据后，数据库中version实际数据比取出的值大，即已被其它线程修改并更新了version  
 user.setVersion*(*user.getVersion*()* - 1*)*;  
 //执行更新  
 userMapper.updateById*(*user*)*;  
*}*

### 三、select



#### 1、根据id查询记录

@Test

public void testSelectById*() {* User user = userMapper.selectById*(*1L*)*;  
 System.*out*.println*(*user*)*;  
*}*

#### 2、通过多个id批量查询

完成了动态sql的foreach的功能

@Test

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

#### 3、简单的条件查询

通过map封装查询条件

@Test

public void testSelectByMap*() {* HashMap*<*String, Object*>* map = new HashMap*<>()*;  
 map.put*(*"name", "Helen"*)*;  
 map.put*(*"age", 18*)*;  
 List*<*User*>* users = userMapper.selectByMap*(*map*)*;  
 users.forEach*(*System.*out*::println*)*;  
*}*

**注意：**map中的key对应的是数据库中的列名。例如数据库user\_id，实体类是userId，这时map的key需要填写user\_id

#### 4、分页

MyBatis Plus自带分页插件，只要简单的配置即可实现分页功能

**（1）创建配置类**

此时可以删除主类中的 *@MapperScan*扫描注解

*/\*\**

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

**（2）测试selectPage分页**

**测试：**最终通过page对象获取相关数据

@Test

public void testSelectPage*() {* Page*<*User*>* page = new Page*<>(*1, 5*)*;  
 userMapper.selectPage*(*page, null*)*;  
 page.getRecords*()*.forEach*(*System.*out*::println*)*;// page封装所有数据  
 System.*out*.println*(*page.getCurrent*())*;  
 System.*out*.println*(*page.getPages*())*;  
 System.*out*.println*(*page.getSize*())*;  
 System.*out*.println*(*page.getTotal*())*;  
 System.*out*.println*(*page.hasNext*())*;  
 System.*out*.println*(*page.hasPrevious*())*;  
*}*

控制台sql语句打印：SELECT id,name,age,email,create\_time,update\_time FROM user LIMIT 0,5

**（3）测试selectMapsPage分页：结果集是Map**

@Test

public void testSelectMapsPage*() {* Page*<*User*>* page = new Page*<>(*1, 5*)*;  
 IPage*<*Map*<*String, Object*>>* mapIPage = userMapper.selectMapsPage*(*page, null*)*;  
 //注意：此行必须使用 mapIPage 获取记录列表，否则会有数据类型转换错误  
 mapIPage.getRecords*()*.forEach*(*System.*out*::println*)*;  
 System.*out*.println*(*page.getCurrent*())*;  
 System.*out*.println*(*page.getPages*())*;  
 System.*out*.println*(*page.getSize*())*;  
 System.*out*.println*(*page.getTotal*())*;  
 System.*out*.println*(*page.hasNext*())*;  
 System.*out*.println*(*page.hasPrevious*())*;  
*}*

### 四、delete



#### 1、根据id删除记录

@Test

public void testDeleteById*() {* int result = userMapper.deleteById*(*8L*)*;  
 System.*out*.println*(*result*)*;  
*}*

#### 2、批量删除

@Test

public void testDeleteBatchIds*() {* int result = userMapper.deleteBatchIds*(*Arrays.*asList(*8, 9, 10*))*;  
 System.*out*.println*(*result*)*;  
*}*

#### 3、简单的条件查询删除

@Test

public void testDeleteByMap*() {* HashMap*<*String, Object*>* map = new HashMap*<>()*;  
 map.put*(*"name", "Helen"*)*;  
 map.put*(*"age", 18*)*;  
 int result = userMapper.deleteByMap*(*map*)*;  
 System.*out*.println*(*result*)*;  
*}*

#### 4、逻辑删除

* 物理删除：真实删除，将对应数据从数据库中删除，之后查询不到此条被删除数据
* 逻辑删除：假删除，将对应数据中代表是否被删除字段状态修改为“被删除状态”，之后在数据库中仍旧能看到此条数据记录

**（1）数据库中添加 deleted字段**

ALTER TABLE `user` ADD COLUMN `deleted` boolean

C:\Users\gouwenyong0000\Documents\My Knowledge\temp\4582cfd6-5fb7-4336-b5b2-cfb54f21d292\128\index_files\bc4cbff4-c2b8-45d5-ae8d-53439dd2330c.png

**（2）实体类添加deleted 字段**

并加上 @TableLogic 注解 和 @TableField(fill = FieldFill.INSERT) 注解

@TableLogic

@TableField*(*fill = FieldFill.*INSERT)*//为了第一次填充默认值 或者数据库设置默认值  
private Integer deleted;

**（3）元对象处理器接口添加deleted的insert默认填充值**

@Override

public void insertFill*(*MetaObject metaObject*) {*......  
 this.setFieldValByName*(*"deleted", 0, metaObject*)*;  
*}*

**（4）application.properties 加入配置**

此为默认值，如果你的默认值和mp默认的一样,该配置可无

mybatis-plus.global-config.db-config.logic-delete-value=1  
mybatis-plus.global-config.db-config.logic-not-delete-value=0

**（5）在 MybatisPlusConfig 中注册 Bean**

*/\*\**

*\* 逻辑删除插件  
 \*/*

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

**（6）测试逻辑删除**

* 测试后发现，数据并没有被删除，deleted字段的值由0变成了1
* 测试后分析打印的sql语句，是一条update
* **注意：**被删除数据的deleted 字段的值必须是 0，才能被选取出来执行逻辑删除的操作

*/\*\**

*\* 测试 逻辑删除  
 \*/*@Test  
public void testLogicDelete*() {* int result = userMapper.deleteById*(*1L*)*;  
 System.*out*.println*(*result*)*;  
*}*

**（7）测试逻辑删除后的查询**

MyBatis Plus中查询操作也会自动添加逻辑删除字段的判断

*/\*\**

*\* 测试 逻辑删除后的查询：  
 \* 不包括被逻辑删除的记录  
 \*/*@Test  
public void testLogicDeleteSelect*() {* User user = new User*()*;  
 List*<*User*>* users = userMapper.selectList*(*null*)*;  
 users.forEach*(*System.*out*::println*)*;  
*}*

测试后分析打印的sql语句，包含 WHERE deleted=0

SELECT id,name,age,email,create\_time,update\_time,deleted FROM user WHERE deleted=0

注意：查询被删除的数据【delete=1】，使用Mybatis手写SQL完成

### 五、性能分析

性能分析拦截器，用于输出每条 SQL 语句及其执行时间

SQL 性能执行分析,开发环境使用，超过指定时间，停止运行。有助于发现问题

#### 1、配置插件

**（1）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。

**（2）在 MybatisPlusConfig 中配置**

*/\*\**

*\* SQL 执行性能分析插件  
 \* 开发环境使用，线上不推荐。 maxTime 指的是 sql 最大执行时长  
 \*/*@Bean  
@Profile*({*"dev", "test"*})*// 设置 dev test 环境开启  
public PerformanceInterceptor performanceInterceptor*() {* PerformanceInterceptor performanceInterceptor = new PerformanceInterceptor*()*;  
 performanceInterceptor.setMaxTime*(*100*)*;//ms，超过此处设置的ms则sql不执行  
 performanceInterceptor.setFormat*(*true*)*;  
 return performanceInterceptor;  
*}*

**（3）Spring Boot 中设置dev环境**

#环境设置：dev、test、prod  
spring.profiles.active=dev

可以针对各环境新建不同的配置文件application-dev.properties、application-test.properties、application-prod.properties

也可以自定义环境名称：如test1、test2

#### 2、测试

**（1）常规测试**

*/\*\**

*\* 测试 性能分析插件  
 \*/*@Test  
public void testPerformance*() {* User user = new User*()*;  
 user.setName*(*"我是Helen"*)*;  
 user.setEmail*(*"helen@sina.com"*)*;  
 user.setAge*(*18*)*;  
 userMapper.insert*(*user*)*;  
*}*

**输出：**

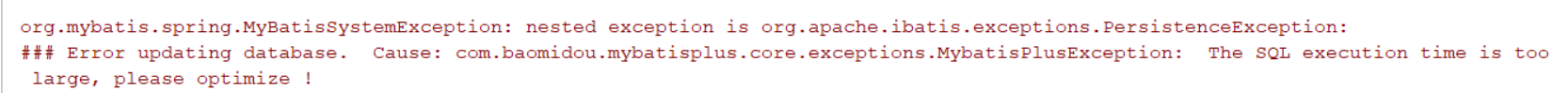


**（2）将maxTime 改小之后再次进行测试**

performanceInterceptor.setMaxTime(5);//ms，超过此处设置的ms不执行

如果执行时间过长，则抛出异常：The SQL execution time is too large,

**输出：**



### 六、其它

如果想进行复杂条件查询，那么需要使用条件构造器 Wapper，涉及到如下方法

**1、delete**

**2、selectOne**

**3、selectCount**

**4、selectList**

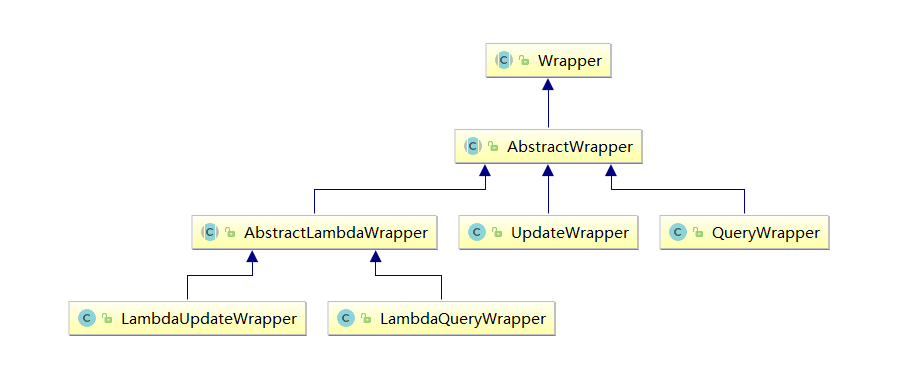
**5、selectMaps**

**6、selectObjs**

**7、update**

## 04-MyBatisPlus条件构造器

### 一、wapper介绍



Wrapper ： 条件构造抽象类，最顶端父类

    AbstractWrapper ： 用于查询条件封装，生成 sql 的 where 条件

        QueryWrapper ： Entity 对象封装操作类，不是用lambda语法

        UpdateWrapper ： Update 条件封装，用于Entity对象更新操作

    AbstractLambdaWrapper ： Lambda 语法使用 Wrapper统一处理解析 lambda 获取 column。

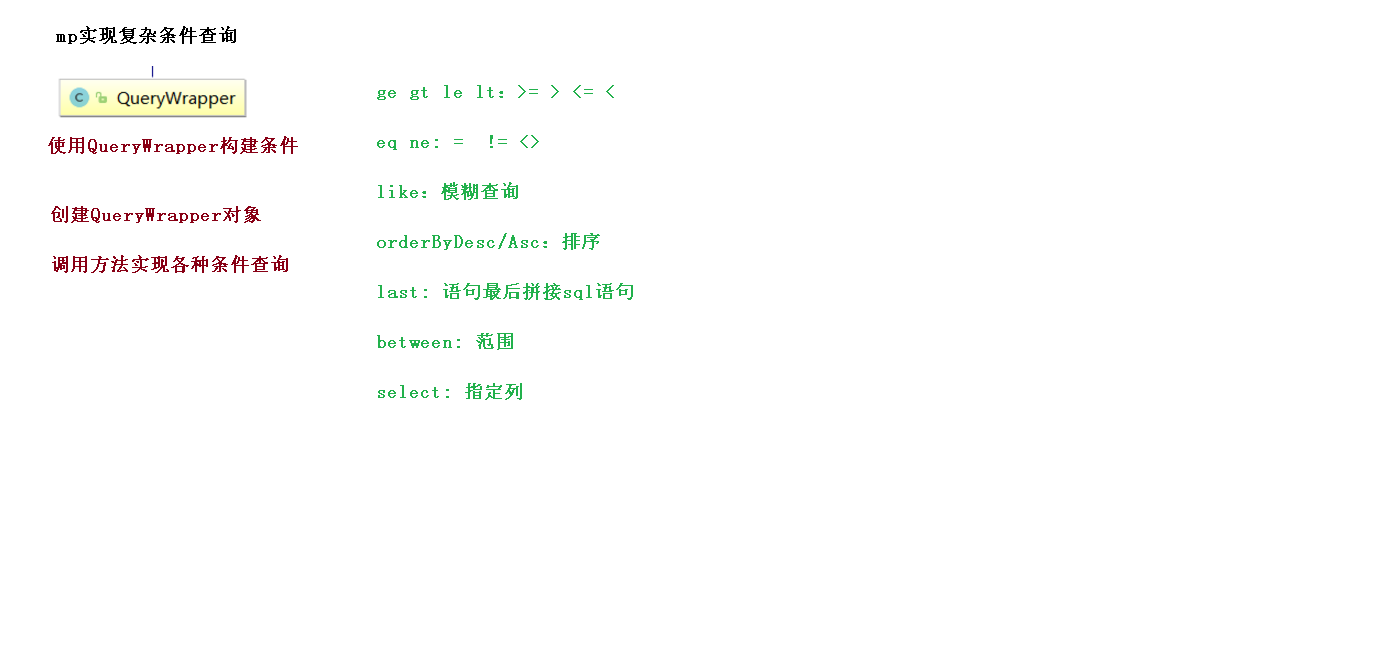
        LambdaQueryWrapper ：看名称也能明白就是用于Lambda语法使用的查询Wrapper

        LambdaUpdateWrapper ： Lambda 更新封装Wrapper

@RunWith*(*SpringRunner.class*)*

@SpringBootTest  
public class QueryWrapperTests *{* @Autowired  
 private UserMapper userMapper;  
*}*

### 二、AbstractWrapper

 **注意：**以下条件构造器的方法入参中的 column 均表示数据库字段

#### 1、ge、gt、le、lt、isNull、isNotNull

@Test

public void testDelete*() {* QueryWrapper*<*User*>* queryWrapper = new QueryWrapper*<>()*;  
 queryWrapper  
 .isNull*(*"name"*)* .ge*(*"age", 12*)* .isNotNull*(*"email"*)*;  
 int result = userMapper.delete*(*queryWrapper*)*;  
 System.*out*.println*(*"delete return count = " + result*)*;  
*}*

SQL：UPDATE user SET deleted=1 WHERE deleted=0 AND name IS NULL AND age >= ? AND email IS NOT NULL

#### 2、eq、ne

**注意：**seletOne返回的是一条实体记录，当出现多条时会报错

@Test

public void testSelectOne*() {* QueryWrapper*<*User*>* queryWrapper = new QueryWrapper*<>()*;  
 queryWrapper.eq*(*"name", "Tom"*)*;  
 User user = userMapper.selectOne*(*queryWrapper*)*;  
 System.*out*.println*(*user*)*;  
*}*

SELECT id,name,age,email,create\_time,update\_time,deleted,version FROM user WHERE deleted=0 AND name = ?

#### 3、between、notBetween

包含大小边界

@Test

public void testSelectCount*() {* QueryWrapper*<*User*>* queryWrapper = new QueryWrapper*<>()*;  
 queryWrapper.between*(*"age", 20, 30*)*;  
 Integer count = userMapper.selectCount*(*queryWrapper*)*;  
 System.*out*.println*(*count*)*;  
*}*

SELECT COUNT(1) FROM user WHERE deleted=0 AND age BETWEEN ? AND ?

#### 4、allEq

@Test

public void testSelectList*() {* QueryWrapper*<*User*>* queryWrapper = new QueryWrapper*<>()*;  
 Map*<*String, Object*>* map = new HashMap*<>()*;  
 map.put*(*"id", 2*)*;  
 map.put*(*"name", "Jack"*)*;  
 map.put*(*"age", 20*)*;  
 queryWrapper.allEq*(*map*)*;  
 List*<*User*>* users = userMapper.selectList*(*queryWrapper*)*;  
 users.forEach*(*System.*out*::println*)*;  
*}*

SELECT id,name,age,email,create\_time,update\_time,deleted,version

FROM user WHERE deleted=0 AND name = ? AND id = ? AND age = ?

#### 5、like、notLike、likeLeft、likeRight

selectMaps返回Map集合列表

@Test

public void testSelectMaps*() {* QueryWrapper*<*User*>* queryWrapper = new QueryWrapper*<>()*;  
 queryWrapper  
 .notLike*(*"name", "e"*)* .likeRight*(*"email", "t"*)*;  
 List*<*Map*<*String, Object*>>* maps = userMapper.selectMaps*(*queryWrapper*)*;//返回值是Map列表  
 maps.forEach*(*System.*out*::println*)*;  
*}*

SELECT id,name,age,email,create\_time,update\_time,deleted,version

FROM user WHERE deleted=0 AND name NOT LIKE ? AND email LIKE ?

#### 6、in、notIn、inSql、notinSql、exists、notExists

in、notIn：

* notIn("age",{1,2,3})--->age not in (1,2,3)
* notIn("age", 1, 2, 3)--->age not in (1,2,3)

inSql、notinSql：可以实现子查询

* 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)

@Test

public void testSelectObjs*() {* QueryWrapper*<*User*>* queryWrapper = new QueryWrapper*<>()*;  
 //queryWrapper.in("id", 1, 2, 3);  
 queryWrapper.inSql*(*"id", "select id from user where id < 3"*)*;  
 List*<*Object*>* objects = userMapper.selectObjs*(*queryWrapper*)*;//返回值是Object列表  
 objects.forEach*(*System.*out*::println*)*;  
*}*

SELECT id,name,age,email,create\_time,update\_time,deleted,version

FROM user WHERE deleted=0 AND id IN (select id from user where id < 3)

#### 7、or、and

**注意：**这里使用的是 UpdateWrapper

不调用or则默认为使用 and 连

@Test

public void testUpdate1*() {* //修改值  
 User user = new User*()*;  
 user.setAge*(*99*)*;  
 user.setName*(*"Andy"*)*;  
 //修改条件  
 UpdateWrapper*<*User*>* userUpdateWrapper = new UpdateWrapper*<>()*;  
 userUpdateWrapper  
 .like*(*"name", "h"*)* .or*()* .between*(*"age", 20, 30*)*;  
 int result = userMapper.update*(*user, userUpdateWrapper*)*;  
 System.*out*.println*(*result*)*;  
*}*

UPDATE user SET name=?, age=?, update\_time=? WHERE deleted=0 AND name LIKE ? OR age BETWEEN ? AND ?

#### 8、嵌套or、嵌套and

这里使用了lambda表达式，or中的表达式最后翻译成sql时会被加上圆括号

@Test

public void testUpdate2*() {* //修改值  
 User user = new User*()*;  
 user.setAge*(*99*)*;  
 user.setName*(*"Andy"*)*;  
 //修改条件  
 UpdateWrapper*<*User*>* userUpdateWrapper = new UpdateWrapper*<>()*;  
 userUpdateWrapper  
 .like*(*"name", "h"*)* .or*(*i -> i.eq*(*"name", "李白"*)*.ne*(*"age", 20*))*;  
 int result = userMapper.update*(*user, userUpdateWrapper*)*;  
 System.*out*.println*(*result*)*;  
*}*

UPDATE user SET name=?, age=?, update\_time=?

WHERE deleted=0 AND name LIKE ?

OR ( name = ? AND age <> ? )

#### 9、orderBy、orderByDesc、orderByAsc

@Test

public void testSelectListOrderBy*() {* QueryWrapper*<*User*>* queryWrapper = new QueryWrapper*<>()*;  
 queryWrapper.orderByDesc*(*"id"*)*;  
 List*<*User*>* users = userMapper.selectList*(*queryWrapper*)*;  
 users.forEach*(*System.*out*::println*)*;  
*}*

SELECT id,name,age,email,create\_time,update\_time,deleted,version

FROM user WHERE deleted=0 ORDER BY id DESC

#### 10、last

直接拼接到 sql 的最后

**注意：**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

@Test

public void testSelectListLast*() {* QueryWrapper*<*User*>* queryWrapper = new QueryWrapper*<>()*;  
 queryWrapper.last*(*"limit 1"*)*;  
 List*<*User*>* users = userMapper.selectList*(*queryWrapper*)*;  
 users.forEach*(*System.*out*::println*)*;  
*}*

SELECT id,name,age,email,create\_time,update\_time,deleted,version

FROM user WHERE deleted=0 limit 1

#### 11、指定要查询的列

@Test

public void testSelectListColumn*() {* QueryWrapper*<*User*>* queryWrapper = new QueryWrapper*<>()*;  
 queryWrapper.select*(*"id", "name", "age"*)*;  
 List*<*User*>* users = userMapper.selectList*(*queryWrapper*)*;  
 users.forEach*(*System.*out*::println*)*;  
*}*

SELECT id,name,age FROM user WHERE deleted=0

#### 12、set、setSql

最终的sql会合并 user.setAge()，以及 userUpdateWrapper.set()  和 setSql() 中 的字段

@Test、

public void testUpdateSet*() {* //修改值  
 User user = new User*()*;  
 user.setAge*(*99*)*;  
 //修改条件  
 UpdateWrapper*<*User*>* userUpdateWrapper = new UpdateWrapper*<>()*;  
 userUpdateWrapper  
 .like*(*"name", "h"*)* .set*(*"name", "老李头"*)*//除了可以查询还可以使用set设置修改的字段  
 .setSql*(*" email = '123@qq.com'"*)*;//可以有子查询  
 int result = userMapper.update*(*user, userUpdateWrapper*)*;  
*}*

UPDATE user SET age=?, update\_time=?, name=?, email = '123@qq.com' WHERE deleted=0 AND name LIKE ?