**每特教育互联网微服务电商项目课程安排**

# 课程内容

会员登陆模块优化与门户平台整合

1.使用注解方式实现参数验证

2.Redis删除Token与数据库状态Token如何保持一致

3.创建门户Portal-Web整合Ftl

4.Portal-Web调用会员服务接口实现注册

# 相关说明

**该项目“基于SpringCloud2.x构建微服务电商项目”由每特教育|蚂蚁课堂版权所有，未经过允许的情况下，私自分享视频和源码属于违法行为。**

**该项目作者:97后互联网架构师-余胜军**

**相关网址:www.mayikt.com**

**联系方式:微信yushengjun644、QQ644064779**

**该想项目是采用目前比较流行的SpringBoot/SpringCloud构建微服务电商项目，实现一套串联的微服务电商项目，能完全掌握该知识，可以在一线城市拿到月薪25+k薪资。**

**完全符合一线城市微服务电商的需求，对中国程序猿学习微服务电商架构，有非常大的帮助，该项目涵盖从微服务电商需求讨论、数据库设计、技术选型、互联网安全架构、整合SpringCloud各自组件、分布式基础设施、使Docker+k8s+jenkins实现微服务自动化部署、项目上线域名配置、SEO优化、大数据统计分析等。**

**注意：该项目会在2019年4月份左右上线GitHub和码云等开源网站平台。**

**如需参加培训，可以加余老师QQ644064779**

## 学习时间

**学习周期大概四个月时间，每周 2 4 6 晚上20:30-22:30，一个半小时编码课、半个小时面试题辅导。**

**2019年01月03日本周四，开始干我们微服务电商结业项目啦！！希望同学们，坚持哦！**

# 需求讨论与技术架构选型

## 技术选型

**A. 项目采用SpringBoot2.x+SpringCloud2.x构建微服务电商项目**

**1.使用SpringCloudEureka作为注册中心，实现服务治理**

**2.使用Zuul网关框架管理服务请求入口**

**3.使用Ribbon实现本地负载均衡器和FeginHttp客户端调用工具**

**4.使用Hystrix服务保护框架(服务降级、隔离、熔断、限流)**

**5.使用消息总线Stream RabbitMQ和Kafka**

**6.微服务API接口安全控制与单点登陆系统CAS+JWT+Oauth2.0**

**B. 分布式基础设施环境构建**

**1.分布式任务调度平台XXL-Job**

**2.分布式日志采集系统ELK +kafka**

**3.分布式事务解决方案LCN （MQ）**

**4.分布式锁解决方案Zookeeper、Redis**

**5.分布式配置中心携程阿波罗**

**6.高并发分布式全局ID生成雪花算法**

**7.分布式Session框架Spring-Session**

**8.分布式服务追踪与调用链ZipKin**

**C.项目运营与部署环境**

**1.分布式设施环境，统一采用docker安装**

**2.使用jenkins+docker+k8s实现自动部署**

**3.微服务API管理ApiSwagger**

**4.使用GitLab代码管理**

**5.统一采用第三方云数据库**

**6.使用七牛云服务器对静态资源实现加速**

**7.构建企业级Maven私服**

## 课程安排

**第一节 需求讨论与技术架构选型**

**第二节 微服务架构环境搭建Maven私服仓库、GitLiab代码仓库**

**第三节 构建微服务电商项目与数据库实战设计**

**第四节 分布式基础设施之分布式配置中心环境搭建**

**第五节 微服务电商项目-会员服务-联合登陆**

**第六节 微服务电商项目-会员服务-唯一登陆**

**第七节 微服务电商项目-会员服务-SSO联合登陆**

**第八节 微服务电商项目-微服务公众号-构建微信公众号**

**第九节 微服务电商项目-消息服务平台-消息服务平台**

**第十节 微服务电商项目-前端技术-VUE**

**第十一节 微服务电商项目-商品服务-数据设计**

**第十二节 微服务电商项目-商品服务-搜索服务**

**第十三节 微服务电商项目-商品服务-购物车设计**

**第十四节 微服务电商项目-订单服务-订单服务**

**第十五节 微服务电商项目-聚合支付-数据设计**

**第十六节 微服务电商项目-聚合支付-银联支付**

**第十七节 微服务电商项目-聚合支付-支付宝支付**

**第十八节 微服务电商项目-秒杀服务-秒杀抢购**

**第十九节 微服务电商项目-后台运营平台设计**

**第二十节 微服务电商项目ELK日志分析**

**第二十一节 微服务电商项目安全架构**

**第二十二节 微服务电商项目静态资源CDN加速**

**第二十三节 使用jenkins+docker+k8s实现自动部署**

**第二十四节 微服务电商项目上线部署环境**

**第二十五节 微服务电商项目大数据用户行为分析**

**最终以实际课程表格为准。**

## 环境要求

**为了能够更好的学习互联网微服务架构，该项目对环境要求非常高，建议电脑配置CPU在I5、32GB内存或者电脑采用集群化部署。**

**1.JDK统一要求:JDK1.8K**

**2.Maven 统一管理Jar**

**3.统一采用Docker安装软件**

**4.编码统一采用为UTF-8**

**5.开发工具IDE或者Eclipse**

## 系统架构

**每特教育独创电商项目完全采用微服务架构模式，如果对微服务架构概念还是不够理解的话，可以参考SpringCloud2.x课程。**

**详细参考架构文档:**

****

## 构建项目

meite-shop-parent-----公共Pranet接口

-----meite-shop-basics----分布式基础设施

---------meite-shop-basics-springcloud-eureka—注册中心 8080

---------meite-shop-basics-apollo-config-server—阿波罗分布式配置中心

---------meite-shop-basics-springcloud-zuul—统一请求入口 80

---------meite-shop-basics-xuxueli-xxljob—分布式任务调度平台

---------meite-shop-basics-codingapi-lcn—分布式事务解决框架

---------meite-shop-basics-codingapi- ZipKin —分布式调用链系统

-----meite-shop-service-api提供公共接口

------------ meite-shop-service-api-weixin 微信服务接口

------------ meite-shop-service-api-member会员服务接口

------------ meite-shop-service-api-sso sso服务接口

------------ meite-shop-service-api-item商品服务接口

------------ meite-shop-service-api-search 搜索服务接口

------------ meite-shop-service-api-pay聚合支付平台

------------ meite-shop-service-api-order订单服务接口

------------ meite-shop-service-api-spike 秒杀服务接口

服务接口中包含内存内容: 实体类层、接口层

-----meite-shop-service-impl公共接口的实现

------------ meite-shop-service-weixin 微信服务接口实现

------------ meite-shop-service-member会员服务接口实现

------------ meite-shop-service-api-sso sso服务接口实现

------------ meite-shop-service-tem商品服务接口实现

------------ meite-shop-service-search 搜索服务接口实现

------------ meite-shop-service-pay聚合支付平台接口实现

------------ meite-shop-service-order订单服务接口实现

------------ meite-shop-service-api-spike 秒杀服务接口

-----meite-shop-common 工具类

---------meite-shop-common-core—核心工具类

-----meite-shop-portal 门户平台

--------meite-shop-portal-web 门户网站 支持PC端和H5端访问

--------meite-shop-portal-sso 单点登陆系统

--------meite-shop-portal-search 搜索系统

--------meite-shop-portal-spike 秒杀系统

--------meite-shop-portal-cms 系统

**后期有新的功能在加。**

**预计40个项目**

**如果项目需要被其他项目继承的话，那么该项目选择maven类型为pom类型**

## 版本说明

**SpringCloud2.x以上版本**

## 注意事项

1. [统一采用@GetMapping](mailto:1.统一采用@GetMapping)**或者@PostMapping实现接口映射**

# 环境构建

步骤①创建以下项目，maven类型选择为Pom类型

1. 创建meite-shop-parent
2. 创建meite-shop-basics
3. 创建meite-shop-portal
4. 创建meite-shop-service
5. 创建meite-shop-service-api

步骤②分布式基础设施eureka

1. 创建meite-shop-basics-springcloud-eureka

步骤③创建微信、会员服务接口

1. 创建meite-shop-service-api-member
2. 创建meite-shop-service-api-weixin

步骤④创建微信、会员服务接口实现maven类型选择为Jar类型

1.创建meite-shop-service-member

1.创建meite-shop-service-weixin

## 分布式基础设施环境环境准备

### 基于Docker部署GitLab环境搭建

**建议虚拟机内存2G以上**

**1.下载镜像文件**

|  |
| --- |
| docker pull beginor/gitlab-ce:11.0.1-ce.0 |

**注意：一定要配置阿里云的加速镜像**

1. **创建GitLab 的配置 (etc) 、 日志 (log) 、数据 (data) 放到容器之外， 便于日后升级， 因此请先准备这三个目录。**

|  |
| --- |
| **mkdir -p /mnt/gitlab/etc**  **mkdir -p /mnt/gitlab/log**  **mkdir -p /mnt/gitlab/data** |

1. **运行GitLab容器**

|  |
| --- |
| **docker run \**  **--detach \**  **--publish 8443:443 \**  **--publish 8090:80 \**  **--name gitlab \**  **--restart unless-stopped \**  **-v /mnt/gitlab/etc:/etc/gitlab \**  **-v /mnt/gitlab/log:/var/log/gitlab \**  **-v /mnt/gitlab/data:/var/opt/gitlab \**  **beginor/gitlab-ce:11.0.1-ce.0** |

停止docker容器，并且删除

Docker stop 容器id

Docker rm 容器id

systemctl stop firewalld

4.修改/mnt/gitlab/etc/gitlab.rb

**把external\_url改成部署机器的域名或者IP地址**

|  |
| --- |
| **vi /mnt/gitlab/etc/gitlab.rb** |

**external\_url 'http://192.168.212.227'**

1. **修改/mnt/gitlab/data/gitlab-rails/etc/gitlab.yml**

**vi /mnt/gitlab/data/gitlab-rails/etc/gitlab.yml**

**找到关键字 \* ## Web server settings \***

**将host的值改成映射的外部主机ip地址和端口，这里会显示在gitlab克隆地址**



到此为止，gitlab的web管理页面就可以正常访问



**修改默认root账户密码**

**Yushengjun644**

#### GitLab创建项目





#### Eclipse提交项目

1. 项目初始化



选择Git

创建存储库





提交代码





#### 新增用户权限

1. 注册用户mayikt 密码yushengjun644 授权Git代码地址
2. 设置中找到成员，添加项目权限即可

### SwaggerApi管理

#### meite-shop-service-api

##### 引入Maven依赖

|  |
| --- |
| <!-- swagger-spring-boot -->  <dependency>  <groupId>com.spring4all</groupId>  <artifactId>swagger-spring-boot-starter</artifactId>  <version>1.7.0.RELEASE</version>  </dependency> |

#### meite-shop-service-member

配置文件新增

|  |
| --- |
| ####swagger相关配置  swagger:  base-package: com.mayikt.member.service  title: SpringCloud2.x构建微服务电商项目-微信服务接口  description: 该项目“基于SpringCloud2.x构建微服务电商项目”由每特教育|蚂蚁课堂版权所有，未经过允许的情况下，私自分享视频和源码属于违法行为。  version: 1.1  terms-of-service-url: www.mayikt.com  contact:  name: 97后互联网架构师-余胜军  email: 644064779@qq.com |

@EnableSwagger2Doc 开启Swagger生成文件

#### meite-shop-service-api-weixin

配置文件新增

|  |
| --- |
| ####swagger相关配置  swagger:  base-package: com.mayikt.weixin.service  title: SpringCloud2.x构建微服务电商项目-微信服务接口  description: 该项目“基于SpringCloud2.x构建微服务电商项目”由每特教育|蚂蚁课堂版权所有，未经过允许的情况下，私自分享视频和源码属于违法行为。  version: 1.1  terms-of-service-url: www.mayikt.com  contact:  name: 97后互联网架构师-余胜军  email: 644064779@qq.com |

@EnableSwagger2Doc 开启Swagger生成文件

http://127.0.0.1:8300/swagger-ui.html

http://127.0.0.1:8200/swagger-ui.html

#### 使用微服务网关统一管理Api

创建项目meite-shop-basics-springcloud-zuul

##### Maven依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-zuul</artifactId>  </dependency>  <!-- swagger-spring-boot -->  <dependency>  <groupId>com.spring4all</groupId>  <artifactId>swagger-spring-boot-starter</artifactId>  <version>1.7.0.RELEASE</version>  </dependency>  </dependencies> |

##### 统一管理API代码

|  |
| --- |
| @SpringBootApplication  @EnableEurekaClient  @EnableZuulProxy  @EnableSwagger2Doc  **public** **class** AppGateWay {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(AppGateWay.**class**, args);  }  // 添加文档来源  @Component  @Primary  **class** DocumentationConfig **implements** SwaggerResourcesProvider {  @Override  **public** List<SwaggerResource> get() {  List resources = **new** ArrayList<>();  // app-itmayiedu-order  resources.add(swaggerResource("app-mayikt-member", "/app-mayikt-member/v2/api-docs", "2.0"));  resources.add(swaggerResource("app-mayikt-weixin", "/app-mayikt-weixin/v2/api-docs", "2.0"));  **return** resources;  }  **private** SwaggerResource swaggerResource(String name, String location, String version) {  SwaggerResource swaggerResource = **new** SwaggerResource();  swaggerResource.setName(name);  swaggerResource.setLocation(location);  swaggerResource.setSwaggerVersion(version);  **return** swaggerResource;  }  }  } |

##### 相关配置

|  |
| --- |
| ###服务启动端口号  server:  port: 80  ###服务名称(服务注册到eureka名称)  spring:  application:  name: app-mayikt-zuul  ###服务注册到eureka地址  eureka:  client:  service-url:  defaultZone: http://localhost:8100/eureka  ### 配置网关反向代理  zuul:  routes:  api-a:  ### 以 /api-weixin/访问转发到会员服务  path: /api-weixin/\*\*  serviceId: app-mayikt-weixin  api-b:  ### 以 /api-member/访问转发到订单服务  path: /api-member/\*\*  serviceId: app-mayikt-member |

### Maven私服环境搭建

1.一些无法从外部仓库下载的构件，例如内部的项目还能部署到私服上，以便供其他依赖项目使用。

2. 为了节省带宽和时间，在局域网内架设一个私有的仓库服务器，用其代理所有外部的远程仓库。当本地Maven项目需要下载构件时，先去私服请求，如果私服没有，则再去远程仓库请求，从远程仓库下载构件后，把构件缓存在私服上。这样，及时暂时没有Internet链接，由于私服已经缓存了大量构件，整个项目还是可以正常使用的。同时，也降低了中央仓库的负荷。



#### 什么场景用Maven私服呢？

在实际开发中，项目中可能会用到第三方的jar、内部通讯的服务接口都会打入到公司的私服中。

#### 基于Docker搭建Maven私服

|  |
| --- |
| 1. 下载一个nexus3的镜像   docker pull sonatype/nexus3   1. 将容器内部/var/nexus-data挂载到主机/root/nexus-data目录。   docker run -d -p 8081:8081 --name nexus -v /root/nexus-data:/var/nexus-data --restart=always sonatype/nexus3 |

关闭防火墙,访问<http://ip:8081>

Maven私服启动容器稍微比较慢，等待1分钟即可。

默认登陆账号 admin admin123

#### 创建Maven私服仓库

##### 创建私服仓库

创建仓库，点击Create repository,然后选择maven2(hosted)然后输入仓库名称（test-release）。在version policy中选择这个仓库的类型，这里选择release,在Deployment policy中选择Allow redeploy（这个很重要）.

##### 创建私服账号

点击左侧菜单栏的Users菜单，然后点击Create local user.我这里创建了一个用户，账号密码都是：mayikt

Mayikt-release

##### 本地settings.xml

|  |
| --- |
| <servers>  <server>  <id>mayikt</id>  <username>mayikt</username>  <password>mayikt</password>  </server>    </servers> |

##### 创建一个Maven工程

创建一个maven工程，并且打包到maven私服。

相关配置

|  |
| --- |
| <!--注意限定版本一定为RELEASE,因为上传的对应仓库的存储类型为RELEASE -->  <!--指定仓库地址 -->  <distributionManagement>  <repository>  <!--此名称要和.m2/settings.xml中设置的ID一致 -->  <id>mayikt</id>  <url>http://192.168.212.230:8081/repository/mayikt-release/</url>  </repository>  </distributionManagement>  <build>  <plugins>  <!--发布代码Jar插件 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-deploy-plugin</artifactId>  <version>2.7</version>  </plugin>  <!--发布源码插件 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-source-plugin</artifactId>  <version>2.2.1</version>  <executions>  <execution>  <phase>package</phase>  <goals>  <goal>jar</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build> |

mvn deploy

##### 测试依赖信息

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.mayikt</groupId>  <artifactId>mayikt\_springboot</artifactId>  <version>0.0.1-RELEASE</version>  </dependency>  </dependencies>  <repositories>  <repository>  <id>mayikt</id>  <url>http://192.168.212.233:8081/repository/mayikt-release/</url>  </repository>  </repositories> |

#### 如何判断文件是否发生改变

如何知道一个文件是否改变了呢？当然是用比较文件hash值的方法，文件hash又叫文件签名，文件中哪怕一个bit位被改变了，文件hash就会不同。比较常用的文件hash算法有MD5和SHA-1。

#### 发布微服务接口

Non-resolvable parent POM: Could not find artifact com.mayikt:meite-shop-service-api:po

m:0.0.1-RELEASE in central (https://repo.maven.apache.org/maven2) and 'parent.relativePath' points

|  |
| --- |
| <parent>  <groupId>com.mayikt</groupId>  <artifactId>meite-shop-service-api</artifactId>  <version>0.0.1-SNAPSHOT</version>  </parent> |

### 构建分布式配置中心阿波罗

#### 为什么要使用配置文件中心？

为什么要使用分布式配置中心？ 统一管理微服务配置文件，可以实现动态化刷新配置文件。

阿波罗 SpringCloudConfig

为什么我们要使用阿波罗 不使用SpringCloudConfig

阿波罗配置文件存放在数据库中，SpringCloudConfig存放在Git里面。

#### 搭建分布式配置中心阿波罗

1. 下载aploll配置中心 https://github.com/nobodyiam/apollo-build-scripts
2. 上传apollo-build-scripts-master文件到服务器中
3. unzip apollo-build-scripts-master.zip 解压配置文件

如果没有unzip命令的话，安装zip插件 yum -y install zip unzip

1. 配置数据策略

修改demo.sh账号:cdb-8y8qmojr.gz.tencentcdb.com:10010 root Ww861642669+

1. 启动阿波罗 ./demo.sh start

<https://github.com/ctripcorp/apollo/wiki/Apollo%E9%85%8D%E7%BD%AE%E4%B8%AD%E5%BF%83%E4%BB%8B%E7%BB%8D>

systemctl stop firewalld.service

默认账号密码 Apollo admin

#### 服务客户端集成配置文件

1.将本地配置存入到阿波罗平台中。

转换工具

<http://www.toyaml.com/index.html>

2.引入Maven依赖

|  |
| --- |
| <dependency>  <groupId>com.ctrip.framework.apollo</groupId>  <artifactId>apollo-client</artifactId>  <version>1.0.0</version>  </dependency>  <dependency>  <groupId>com.ctrip.framework.apollo</groupId>  <artifactId>apollo-core</artifactId>  <version>1.0.0</version>  </dependency> |

3.创建 application.properties

|  |
| --- |
| app.id=mayikt644064779  apollo.meta=http://192.168.212.236:8080 |

4.项目启动开启阿波罗配置文件

@EnableApolloConfig

5.修改/opt/settings/server.properties（Mac/Linux）或C:\opt\settings\server.properties（Windows）文件，设置env为DEV：

env=DEV

#### 网关服务集成阿波罗

|  |
| --- |
| @SpringBootApplication  @EnableEurekaClient  @EnableZuulProxy  @EnableSwagger2Doc  @EnableApolloConfig  **public** **class** AppGateWay {  // 获取ApolloConfig  @ApolloConfig  **private** Config appConfig;  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(AppGateWay.**class**, args);  }  // 添加文档来源  @Component  @Primary  **class** DocumentationConfig **implements** SwaggerResourcesProvider {  @Override  **public** List<SwaggerResource> get() {  // 开启监听，配置文件发生改变需要更改  appConfig.addChangeListener(**new** ConfigChangeListener() {  @Override  **public** **void** onChange(ConfigChangeEvent changeEvent) {  get();  }  });  **return** resources();  }  /\*\*  \* 从阿波罗服务器中获取resources  \*  \* **@return**  \*/  **private** List<SwaggerResource> resources() {  List resources = **new** ArrayList<>();  // app-itmayiedu-order  // 网关使用服务别名获取远程服务的SwaggerApi  String swaggerDocJson = swaggerDocument();  JSONArray jsonArray = JSONArray.*parseArray*(swaggerDocJson);  **for** (Object object : jsonArray) {  JSONObject jsonObject = (JSONObject) object;  String name = jsonObject.getString("name");  String location = jsonObject.getString("location");  String version = jsonObject.getString("version");  resources.add(swaggerResource(name, location, version));  }  **return** resources;  }  /\*\*  \* 获取swaggerDocument配置  \*  \* **@return**  \*/  **private** String swaggerDocument() {  String property = appConfig.getProperty("mayikt.zuul.swaggerDocument", "");  **return** property;  }  **private** SwaggerResource swaggerResource(String name, String location, String version) {  SwaggerResource swaggerResource = **new** SwaggerResource();  swaggerResource.setName(name);  swaggerResource.setLocation(location);  swaggerResource.setSwaggerVersion(version);  **return** swaggerResource;  }  }  } |

自定义Swagger文档配置mayikt.zuul.swaggerDocument

[

{

"name": "app-mayikt-member",

"location": "/app-mayikt-member/v2/api-docs",

"version": "2.0"

},

{

"name": "app-mayikt-weixin",

"location": "/app-mayikt-weixin/v2/api-docs",

"version": "2.0"

}

]

项目启动监听

|  |
| --- |
| @Component  @Slf4j  **public** **class** MyCommandLineRunner **implements** CommandLineRunner {  @ApolloConfig  **private** Config config;  @Override  **public** **void** run(String... args) **throws** Exception {  config.addChangeListener(**new** ConfigChangeListener() {  @Override  **public** **void** onChange(ConfigChangeEvent changeEvent) {  ***log***.debug("####分布式配置中心监听#####" + changeEvent.changedKeys().toString());  }  });  }  } |

### 搭建企业级微信公众号

微信公众平台：

<https://mp.weixin.qq.com/cgi-bin/home?t=home/index&lang=zh_CN&token=795093844>

微信公众号测试平台

https://mp.weixin.qq.com/debug/cgi-bin/sandbox?t=sandbox/login

微信公众平台密码 [mayikt\_2019@163.com](mailto:mayikt_2019@163.com) 15527339672w.

#### 微信公众平台环境搭建

<https://mp.weixin.qq.com/wiki?t=resource/res_main&id=mp1421135319>

画图演示原理

#### 外网映射工具

Natapp网址ngrok

<https://natapp.cn/>

windows环境运行

运行 natapp -authtoken=a021a8fe913ea048

http://mtmayikt.natapp1.cc/wx/portal/wx5c43fde3c9733d9e

#### WxJava框架快速开发微信公众号

WxJava 微信公众号框架 <https://github.com/Wechat-Group/WxJava>

#### 微服务电商项目引入WxJava框架

错误方式:

|  |
| --- |
| <dependency>  <groupId>com.github.binarywang</groupId>  <artifactId>weixin-java-mp</artifactId>  <version>3.3.0</version>  </ dependency> |

java.lang.NoSuchMethodError: com.thoughtworks.xstream.io.xml.XppDriver.<init>(Lcom/thoughtworks/xstream/io/naming/NameCoder;)

原因是微信框架引入了xstream的版本为1.4.1 而springCloud中eureka-client也引入了xstream为了1.4.9从而版本有冲突。

正确方式:

|  |
| --- |
| <dependency>  <groupId>com.github.binarywang</groupId>  <artifactId>weixin-java-mp</artifactId>  <version>3.3.0</version>  <exclusions>  <exclusion>  <artifactId>xstream</artifactId>  <groupId>com.thoughtworks.xstream</groupId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>  spring-cloud-starter-netflix-eureka-client  </artifactId>  <exclusions>  <exclusion>  <artifactId>xstream</artifactId>  <groupId>com.thoughtworks.xstream</groupId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <artifactId>xstream</artifactId>  <groupId>com.thoughtworks.xstream</groupId>  <version>1.4.10</version>  </dependency> |

##### yml配置

|  |
| --- |
| logging:  level:  org.springframework.web: INFO  com.github.binarywang.demo.wx.mp: DEBUG  me.chanjar.weixin: DEBUG  wx:  mp:  configs:  - appId: wx6f8ce0ccadf3afff #（一个公众号的appid）  secret: cf2b0b4908a84d92b38b939a6fbabe09#（公众号的appsecret）  token: mayikt #（接口配置里的Token值）  mayikt:  weixin:  registration:  code:  ###微信注册码消息  message: 您的注册码为:%s，请关注<a href="https://ke.qq.com/course/273548">腾讯课堂免费公开课</a>，欢迎观看97后架构师余老师的精品课程讲解。官方QQ群：<a href='https://jq.qq.com/?\_wv=1027&k=5TVfAMF'>193086273</a>，期待你的加入，感谢！  ###默认提示消息  default:  registration:  code:  message: 您的消息，我们已经收到,会及时回复给您的! |

#### 微信公众号开发案例

##### 案例1

关注微信公众号，在公众号输入手机号码，对应返回验证码。

|  |
| --- |
| @Component  **public** **class** MsgHandler **extends** AbstractHandler {  // 用户发送手机验证码提示  @Value("${mayikt.weixin.registration.code.message}")  **private** String registrationCodeMessage;  // 默认用户发送验证码提示  @Value("${mayikt.weixin.default.registration.code.message}")  **private** String defaultRegistrationCodeMessage;  @Override  **public** WxMpXmlOutMessage handle(WxMpXmlMessage wxMessage, Map<String, Object> context, WxMpService weixinService,  WxSessionManager sessionManager) {  **if** (!wxMessage.getMsgType().equals(XmlMsgType.***EVENT***)) {  // **TODO** 可以选择将消息保存到本地  }  // 当用户输入关键词如“你好”，“客服”等，并且有客服在线时，把消息转发给在线客服  **try** {  **if** (StringUtils.*startsWithAny*(wxMessage.getContent(), "你好", "客服")  && weixinService.getKefuService().kfOnlineList().getKfOnlineList().size() > 0) {  **return** WxMpXmlOutMessage.*TRANSFER\_CUSTOMER\_SERVICE*().fromUser(wxMessage.getToUser())  .toUser(wxMessage.getFromUser()).build();  }  } **catch** (WxErrorException e) {  e.printStackTrace();  }  // **TODO** 组装回复消息  // 1.验证关键字是否为手机号码类型  String fromMsg = wxMessage.getContent();  **if** (RegexUtils.*checkMobile*(fromMsg)) {  // 如果发送消息为手机号码类型,则发送短信验证码  **int** registCode = registCode();  String retContext = registrationCodeMessage.replaceAll("registrationCodeMessage", registCode + "");  **return** **new** TextBuilder().build(retContext, wxMessage, weixinService);  }  **return** **new** TextBuilder().build(defaultRegistrationCodeMessage, wxMessage, weixinService);  }  // 获取注册码  **private** **int** registCode() {  **int** registCode = (**int**) (Math.*random*() \* 9000 + 1000);  **return** registCode;  }  } |

注意：后期会将验证码存放在redis中。

### 微服务项目整合WxJava框架提供注册码接口

#### 将实体类层抽取出来单独服务应用

创建meite-shop-api-entity模块

---- meite-shop-api-weixin-entity---实体类

#### 规定统一微服务接口状态码

创建meite-shop-common-core工程

{"rtnCode":500,"msg":"系统错误!","data":null}

{"rtnCode":200,"msg":"success","data":{"appId":"yushengjun","appName":"mayikt"}}

##### BaseApiService

|  |
| --- |
| @Data  @Component  **public** **class** BaseApiService<T> {  **public** BaseResponse<T> setResultError(Integer code, String msg) {  **return** setResult(code, msg, **null**);  }  // 返回错误，可以传msg  **public** BaseResponse<T> setResultError(String msg) {  **return** setResult(Constants.***HTTP\_RES\_CODE\_500***, msg, **null**);  }  // 返回成功，可以传data值  **public** BaseResponse<T> setResultSuccess(Object data) {  **return** setResult(Constants.***HTTP\_RES\_CODE\_200***, Constants.***HTTP\_RES\_CODE\_200\_VALUE***, data);  }  // 返回成功，沒有data值  **public** BaseResponse<T> setResultSuccess() {  **return** setResult(Constants.***HTTP\_RES\_CODE\_200***, Constants.***HTTP\_RES\_CODE\_200\_VALUE***, **null**);  }  // 返回成功，沒有data值  **public** BaseResponse<T> setResultSuccess(String msg) {  **return** setResult(Constants.***HTTP\_RES\_CODE\_200***, msg, **null**);  }  // 通用封装  **public** BaseResponse<T> setResult(Integer code, String msg, Object data) {  **return** **new** BaseResponse(code, msg, data);  }  } |

|  |
| --- |
| @Data  **public** **class** BaseResponse<T> {  **private** Integer rtnCode;  **private** String msg;  **private** T data;  **public** BaseResponse() {  }  **public** BaseResponse(Integer rtnCode, String msg, T data) {  **super**();  **this**.rtnCode = rtnCode;  **this**.msg = msg;  **this**.data = data;  }  } |

|  |
| --- |
| **public** **interface** Constants {  // 响应请求成功  String ***HTTP\_RES\_CODE\_200\_VALUE*** = "success";  // 系统错误  String ***HTTP\_RES\_CODE\_500\_VALUE*** = "fial";  // 响应请求成功code  Integer ***HTTP\_RES\_CODE\_200*** = 200;  // 系统错误  Integer ***HTTP\_RES\_CODE\_500*** = 500;  // 未关联QQ账号  Integer ***HTTP\_RES\_CODE\_201*** = 201;  // 发送邮件  String ***MSG\_EMAIL*** = "email";  // 会员token  String ***TOKEN\_MEMBER*** = "TOKEN\_MEMBER";  // 用户有效期 90天  Long ***TOKEN\_MEMBER\_TIME*** = (**long**) (60 \* 60 \* 24 \* 90);  **int** ***COOKIE\_TOKEN\_MEMBER\_TIME*** = (60 \* 60 \* 24 \* 90);  // cookie 会员 totoken 名称  String ***COOKIE\_MEMBER\_TOKEN*** = "cookie\_member\_token";  // 微信code  String ***WEIXINCODE\_KEY*** = "weixin.code";  // 微信注册码有效期30分钟  Long ***WEIXINCODE\_TIMEOUT*** = 1800l;  } |

|  |
| --- |
| @RestController  **public** **class** WeiXinAppServiceImpl **extends** BaseApiService<AppEntity> **implements** WeiXinAppService {  @Value("${mayikt.weixin.name}")  **private** String name;  **public** BaseResponse<AppEntity> getApp() {  // return setResultSuccess(new AppEntity("yushengjun", "mayikt"));  **return** setResultError("系统错误!");  }  } |

#### 微信服务提供注册验证码接口

##### 微信服务引入 Maven依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>com.mayikt</groupId>  <artifactId>meite-shop-service-api-weixin</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <dependency>  <groupId>com.github.binarywang</groupId>  <artifactId>weixin-java-mp</artifactId>  <version>3.3.0</version>  <exclusions>  <exclusion>  <artifactId>xstream</artifactId>  <groupId>com.thoughtworks.xstream</groupId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>  spring-cloud-starter-netflix-eureka-client  </artifactId>  <exclusions>  <exclusion>  <artifactId>xstream</artifactId>  <groupId>com.thoughtworks.xstream</groupId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <artifactId>xstream</artifactId>  <groupId>com.thoughtworks.xstream</groupId>  <version>1.4.10</version>  </dependency>  </dependencies> |

##### MsgHandler

|  |
| --- |
| /\*\*  \* 发送验证码消息  \*/  @Value("${mayikt.weixin.registration.code.message}")  **private** String registrationCodeMessage;  /\*\*  \* 默认回复消息  \*/  @Value("${mayikt.weixin.default.registration.code.message}")  **private** String defaultRegistrationCodeMessage;  @Autowired  **private** RedisUtil redisUtil;  @Override  **public** WxMpXmlOutMessage handle(WxMpXmlMessage wxMessage, Map<String, Object> context, WxMpService weixinService,  WxSessionManager sessionManager) {  **if** (!wxMessage.getMsgType().equals(XmlMsgType.***EVENT***)) {  // **TODO** 可以选择将消息保存到本地  }  // 当用户输入关键词如“你好”，“客服”等，并且有客服在线时，把消息转发给在线客服  **try** {  **if** (StringUtils.*startsWithAny*(wxMessage.getContent(), "你好", "客服")  && weixinService.getKefuService().kfOnlineList().getKfOnlineList().size() > 0) {  **return** WxMpXmlOutMessage.*TRANSFER\_CUSTOMER\_SERVICE*().fromUser(wxMessage.getToUser())  .toUser(wxMessage.getFromUser()).build();  }  } **catch** (WxErrorException e) {  e.printStackTrace();  }  // 1.获取客户端发送的消息  String fromContent = wxMessage.getContent();  // 2.如果客户端发送消息为手机号码，则发送验证码  **if** (RegexUtils.*checkMobile*(fromContent)) {  // 3.生成随机四位注册码  **int** registCode = registCode();  String content = String.*format*(registrationCodeMessage, registCode);  // 4.将验证码存放在Redis中  redisUtil.setString(Constants.***WEIXINCODE\_KEY*** + fromContent, registCode + "", Constants.***WEIXINCODE\_TIMEOUT***);  **return** **new** TextBuilder().build(content, wxMessage, weixinService);  }  **return** **new** TextBuilder().build(defaultRegistrationCodeMessage, wxMessage, weixinService);  }  // 获取注册码  **private** **int** registCode() {  **int** registCode = (**int**) (Math.*random*() \* 9000 + 1000);  **return** registCode;  } |

##### RedisUtil

|  |
| --- |
| @Component  **public** **class** RedisUtil {  @Autowired  **private** StringRedisTemplate stringRedisTemplate;  /\*\*  \* 存放string类型  \*  \* **@param** key  \* key  \* **@param** data  \* 数据  \* **@param** timeout  \* 超时间  \*/  **public** **void** setString(String key, String data, Long timeout) {  stringRedisTemplate.opsForValue().set(key, data);  **if** (timeout != **null**) {  stringRedisTemplate.expire(key, timeout, TimeUnit.***SECONDS***);  }  }  /\*\*  \* 存放string类型  \*  \* **@param** key  \* key  \* **@param** data  \* 数据  \*/  **public** **void** setString(String key, String data) {  setString(key, data, **null**);  }  /\*\*  \* 根据key查询string类型  \*  \* **@param** key  \* **@return**  \*/  **public** String getString(String key) {  String value = stringRedisTemplate.opsForValue().get(key);  **return** value;  }  /\*\*  \* 根据对应的key删除key  \*  \* **@param** key  \*/  **public** **void** delKey(String key) {  stringRedisTemplate.delete(key);  }  } |

##### RegexUtils

|  |
| --- |
| **public** **class** RegexUtils {  /\*\*  \* 验证Email  \*  \* **@param** email  \* email地址，格式：zhangsan@zuidaima.com，zhangsan@xxx.com.cn，  \* xxx代表邮件服务商  \* **@return** 验证成功返回true，验证失败返回false  \*/  **public** **static** **boolean** checkEmail(String email) {  String regex = "\\w+@\\w+\\.[a-z]+(\\.[a-z]+)?";  **return** Pattern.*matches*(regex, email);  }  /\*\*  \* 验证身份证号码  \*  \* **@param** idCard  \* 居民身份证号码15位或18位，最后一位可能是数字或字母  \* **@return** 验证成功返回true，验证失败返回false  \*/  **public** **static** **boolean** checkIdCard(String idCard) {  String regex = "[1-9]\\d{13,16}[a-zA-Z0-9]{1}";  **return** Pattern.*matches*(regex, idCard);  }  /\*\*  \* 验证手机号码（支持国际格式，+86135xxxx...（中国内地），+00852137xxxx...（中国香港））  \*  \* **@param** mobile  \* 移动、联通、电信运营商的号码段  \* <p>  \* 移动的号段：134(0-8)、135、136、137、138、139、147（预计用于TD上网卡）  \* 、150、151、152、157（TD专用）、158、159、187（未启用）、188（TD专用） 177 170 166  \* 开头  \* </p>  \* <p>  \* 联通的号段：130、131、132、155、156（世界风专用）、185（未启用）、186（3g）  \* </p>  \* <p>  \* 电信的号段：133、153、180（未启用）、189  \* </p>  \* **@return** 验证成功返回true，验证失败返回false  \*/  **public** **static** **boolean** checkMobile(String mobile) {  String regex = "^(13[0-9]|14[579]|15[0-3,5-9]|16[6]|17[0135678]|18[0-9]|19[89])\\d{8}$";  **return** Pattern.*matches*(regex, mobile);  }  /\*\*  \* 验证固定电话号码  \*  \* **@param** phone  \* 电话号码，格式：国家（地区）电话代码 + 区号（城市代码） + 电话号码，如：+8602085588447  \* <p>  \* <b>国家（地区） 代码 ：</b>标识电话号码的国家（地区）的标准国家（地区）代码。它包含从 0 到 9  \* 的一位或多位数字， 数字之后是空格分隔的国家（地区）代码。  \* </p>  \* <p>  \* <b>区号（城市代码）：</b>这可能包含一个或多个从 0 到 9 的数字，地区或城市代码放在圆括号——  \* 对不使用地区或城市代码的国家（地区），则省略该组件。  \* </p>  \* <p>  \* <b>电话号码：</b>这包含从 0 到 9 的一个或多个数字  \* </p>  \* **@return** 验证成功返回true，验证失败返回false  \*/  **public** **static** **boolean** checkPhone(String phone) {  String regex = "(\\+\\d+)?(\\d{3,4}\\-?)?\\d{7,8}$";  **return** Pattern.*matches*(regex, phone);  }  /\*\*  \* 验证整数（正整数和负整数）  \*  \* **@param** digit  \* 一位或多位0-9之间的整数  \* **@return** 验证成功返回true，验证失败返回false  \*/  **public** **static** **boolean** checkDigit(String digit) {  String regex = "\\-?[1-9]\\d+";  **return** Pattern.*matches*(regex, digit);  }  /\*\*  \* 验证整数和浮点数（正负整数和正负浮点数）  \*  \* **@param** decimals  \* 一位或多位0-9之间的浮点数，如：1.23，233.30  \* **@return** 验证成功返回true，验证失败返回false  \*/  **public** **static** **boolean** checkDecimals(String decimals) {  String regex = "\\-?[1-9]\\d+(\\.\\d+)?";  **return** Pattern.*matches*(regex, decimals);  }  /\*\*  \* 验证空白字符  \*  \* **@param** blankSpace  \* 空白字符，包括：空格、\t、\n、\r、\f、\x0B  \* **@return** 验证成功返回true，验证失败返回false  \*/  **public** **static** **boolean** checkBlankSpace(String blankSpace) {  String regex = "\\s+";  **return** Pattern.*matches*(regex, blankSpace);  }  /\*\*  \* 验证中文  \*  \* **@param** chinese  \* 中文字符  \* **@return** 验证成功返回true，验证失败返回false  \*/  **public** **static** **boolean** checkChinese(String chinese) {  String regex = "^[\u4E00-\u9FA5]+$";  **return** Pattern.*matches*(regex, chinese);  }  /\*\*  \* 验证日期（年月日）  \*  \* **@param** birthday  \* 日期，格式：1992-09-03，或1992.09.03  \* **@return** 验证成功返回true，验证失败返回false  \*/  **public** **static** **boolean** checkBirthday(String birthday) {  String regex = "[1-9]{4}([-./])\\d{1,2}\\1\\d{1,2}";  **return** Pattern.*matches*(regex, birthday);  }  /\*\*  \* 验证URL地址  \*  \* **@param** url  \* 格式：http://blog.csdn.net:80/xyang81/article/details/7705960? 或  \* http://www.csdn.net:80  \* **@return** 验证成功返回true，验证失败返回false  \*/  **public** **static** **boolean** checkURL(String url) {  String regex = "(https?://(w{3}\\.)?)?\\w+\\.\\w+(\\.[a-zA-Z]+)\*(:\\d{1,5})?(/\\w\*)\*(\\??(.+=.\*)?(&.+=.\*)?)?";  **return** Pattern.*matches*(regex, url);  }  /\*\*  \* <pre>  \* 获取网址 URL 的一级域  \* </pre>  \*  \* **@param** url  \* **@return**  \*/  **public** **static** String getDomain(String url) {  Pattern p = Pattern.*compile*("(?<=http://|\\.)[^.]\*?\\.(com|cn|net|org|biz|info|cc|tv)",  Pattern.***CASE\_INSENSITIVE***);  // 获取完整的域名  // Pattern  // p=Pattern.compile("[^//]\*?\\.(com|cn|net|org|biz|info|cc|tv)",  // Pattern.CASE\_INSENSITIVE);  Matcher matcher = p.matcher(url);  matcher.find();  **return** matcher.group();  }  /\*\*  \* 匹配中国邮政编码  \*  \* **@param** postcode  \* 邮政编码  \* **@return** 验证成功返回true，验证失败返回false  \*/  **public** **static** **boolean** checkPostcode(String postcode) {  String regex = "[1-9]\\d{5}";  **return** Pattern.*matches*(regex, postcode);  }  /\*\*  \* 匹配IP地址(简单匹配，格式，如：192.168.1.1，127.0.0.1，没有匹配IP段的大小)  \*  \* **@param** ipAddress  \* IPv4标准地址  \* **@return** 验证成功返回true，验证失败返回false  \*/  **public** **static** **boolean** checkIpAddress(String ipAddress) {  String regex = "[1-9](\\d{1,2})?\\.(0|([1-9](\\d{1,2})?))\\.(0|([1-9](\\d{1,2})?))\\.(0|([1-9](\\d{1,2})?))";  **return** Pattern.*matches*(regex, ipAddress);  }  } |

##### YML配置

|  |
| --- |
| spring:  # application:  # name: app-mayikt-weixin  redis:  host: 192.168.212.240  port: 6379  pool:  max-idle: 100  min-idle: 1  max-active: 1000  max-wait: -1  logging:  level:  org.springframework.web: INFO  com.github.binarywang.demo.wx.mp: DEBUG  me.chanjar.weixin: DEBUG  wx:  mp:  configs:  - appId: wx5c43fde3c9733d9e #（一个公众号的appid）  secret: b8b217126c33a5fb7074927d5e72a81a #（公众号的appsecret）  token: mayikt #（接口配置里的Token值）  server:  port: 8200    mayikt:  weixin:  registration:  code:  ###微信注册码消息  message: 您的注册码为:%s，请关注<a href="https://ke.qq.com/course/273548">腾讯课堂免费公开课</a>，欢迎观看97后架构师余老师的精品课程讲解。官方QQ群：<a href='https://jq.qq.com/?\_wv=1027&k=5TVfAMF'>193086273</a>，期待你的加入，感谢！  ###默认提示消息  default:  registration:  code:  message: 您的消息，我们已经收到,会及时回复给您的! |

##### 相关依赖

RedisMaven依赖

|  |
| --- |
| <!-- 集成redis -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis</artifactId>  </dependency> |

commons-lang3Maven依赖

|  |
| --- |
| <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-lang3</artifactId>  </dependency> |

##### 将微信配置文件上传到阿波罗

1. 阿波罗平台创建Namespace
2. application.properties

|  |
| --- |
| app.id=new\_app-mayikt-weixin  apollo.meta=http://192.168.212.240:8080  apollo.bootstrap.enabled = true  apollo.bootstrap.namespaces = application,mayikt.weixin |

<https://github.com/ctripcorp/apollo/wiki/Apollo%E6%A0%B8%E5%BF%83%E6%A6%82%E5%BF%B5%E4%B9%8B%E2%80%9CNamespace%E2%80%9D>

##### 提供给会员服务接口注册码验证接口

|  |
| --- |
| @Api(tags = "微信注册码验证码接口")  **public** **interface** VerificaCodeService {  /\*\*  \* 功能说明:根据手机号码验证码token是否正确  \*  \* **@return**  \*/  @ApiOperation(value = "根据手机号码验证码token是否正确")  @GetMapping("/verificaWeixinCode")  @ApiImplicitParams({  // @ApiImplicitParam(paramType="header",name="name",dataType="String",required=true,value="用户的姓名",defaultValue="zhaojigang"),  @ApiImplicitParam(paramType = "query", name = "phone", dataType = "String", required = **true**, value = "用户手机号码"),  @ApiImplicitParam(paramType = "query", name = "weixinCode", dataType = "String", required = **true**, value = "微信注册码") })  **public** BaseResponse<JSONObject> verificaWeixinCode(String phone, String weixinCode);  } |

|  |
| --- |
| @RestController  **public** **class** VerificaCodeServiceImpl **extends** BaseApiService<JSONObject> **implements** VerificaCodeService {  @Autowired  **private** RedisUtil redisUtil;  @Override  **public** BaseResponse<JSONObject> verificaWeixinCode(String phone, String weixinCode) {  **if** (StringUtils.*isEmpty*(phone)) {  **return** setResultError("手机号码不能为空!");  }  **if** (StringUtils.*isEmpty*(weixinCode)) {  **return** setResultError("注册码不能为空!");  }  String code = redisUtil.getString(Constants.***WEIXINCODE\_KEY*** + phone);  **if** (StringUtils.*isEmpty*(code)) {  **return** setResultError("注册码已经过期,请重新发送验证码");  }  **if** (!code.equals(weixinCode)) {  **return** setResultError("注册码不正确");  }  **return** setResultSuccess("注册码验证码正确");  }  } |

会员注册需求分析：

1. 微信专注订阅号输入手机号码获取注册码

流程：使用微信事件通知直接返回注册码（将手机号码和对应的注册码存入在redis中）

手机号码作为rediskey,value表示注册码

1. 用户注册调用会员服务接口，会员服务接口调用微信注册码验证接口传递：手机号码和注册码

4.微信服务接口需要提供根据手机号+注册码验证是否正确

微信框架中官方Demo例子中xstream 1.4.10

微信服务项目xstream1.4.9

如何解决Mmaven依赖Jar冲突问题

1.排除依赖jar，在强制引入需要的版本

## 构建会员服务接口

VO（View Object）：视图对象，用于展示层，它的作用是把某个指定页面（或组件）的所有数据封装起来。

DTO（Data Transfer Object）：数据传输对象，这个概念来源于J2EE的设计模式，原来的目的是为了EJB的分布式应用提供粗粒度的数据实体，以减少分布式调用的次数，从而提高分布式调用的性能和降低网络负载，但在这里，我泛指用于展示层与服务层之间的数据传输对象。

DO（Domain Object）：领域对象，就是从现实世界中抽象出来的有形或无形的业务实体。

PO（Persistent Object）：持久化对象，它跟持久层（通常是关系型数据库）的数据结构形成一一对应的映射关系，如果持久层是关系型数据库，那么，数据表中的每个字段（或若干个）就对应PO的一个（或若干个）属性。

### 会员基本用户设计

|  |
| --- |
| DROP TABLE IF EXISTS `meite\_user`;  CREATE TABLE `meite\_user` (  `USER\_ID` int(12) NOT NULL AUTO\_INCREMENT COMMENT 'user\_id',  `MOBILE` varchar(11) NOT NULL COMMENT '手机号',  `EMAIL` varchar(50) NOT NULL COMMENT '邮箱号',  `PASSWORD` varchar(64) NOT NULL COMMENT '密码',  `USER\_NAME` varchar(50) DEFAULT NULL COMMENT '用户名',  `SEX` tinyint(1) DEFAULT '0' COMMENT '性别 1男 2女',  `AGE` tinyint(3) DEFAULT '0' COMMENT '年龄',  `CREATE\_TIME` timestamp NULL DEFAULT NULL COMMENT '注册时间',  `IS\_AVALIBLE` tinyint(1) DEFAULT '1' COMMENT '是否可用 1正常 2冻结',  `PIC\_IMG` varchar(255) DEFAULT NULL COMMENT '用户头像',  `QQ\_OPENID` varchar(50) DEFAULT NULL COMMENT 'QQ联合登陆id',  `WX\_OPENID` varchar(50) DEFAULT NULL COMMENT '微信公众号关注id',  PRIMARY KEY (`USER\_ID`),  UNIQUE KEY `MOBILE\_UNIQUE` (`MOBILE`),  UNIQUE KEY `EMAIL\_UNIQUE` (`EMAIL`)  ) ENGINE=InnoDB AUTO\_INCREMENT=25 DEFAULT CHARSET=utf8 COMMENT='用户会员表'; |

### 会员服务提供用户注册接口

|  |
| --- |
| @Api(tags = "会员注册接口")  **public** **interface** MemberRegisterService {  /\*\*  \* 用户注册接口  \*  \* **@param** userEntity  \* **@return**  \*/  @PostMapping("/register")  @ApiOperation(value = "会员用户注册信息接口")  BaseResponse<JSONObject> register(@RequestBody UserEntity userEntity,  @RequestParam("registCode") String registCode);  }  @Data  @ApiModel(value = "用户中注册")  **public** **class** UserEntity {  /\*\*  \* userid  \*/  @ApiModelProperty(value = "用户id")  **private** Long userid;  /\*\*  \* 手机号码  \*/  @ApiModelProperty(value = "手机号码")  **private** String mobile;  /\*\*  \* 邮箱  \*/  @ApiModelProperty(value = "邮箱")  **private** String email;  /\*\*  \* 密码  \*/  @ApiModelProperty(value = "密码")  **private** String password;  /\*\*  \* 用户名称  \*/  @ApiModelProperty(value = "用户名称")  **private** String userName;  /\*\*  \* 性别 0 男 1女  \*/  @ApiModelProperty(value = "用户性别")  **private** **char** sex;  /\*\*  \* 年龄  \*/  @ApiModelProperty(value = "用户年龄")  **private** Long age;  /\*\*  \* 注册时间  \*/  @ApiModelProperty(value = "创建时间")  **private** Date createTime;  /\*\*  \* 修改时间  \*  \*/  @ApiModelProperty(value = "修改时间")  **private** Date updateTime;  /\*\*  \* 账号是否可以用 1 正常 0冻结  \*/  @ApiModelProperty(value = "账号是否可以用 1 正常 0冻结")  **private** **char** is\_avalible;  /\*\*  \* 用户头像  \*/  @ApiModelProperty(value = " 用户头像")  **private** String pic\_img;  /\*\*  \* 用户关联 QQ 开放ID  \*/  @ApiModelProperty(value = "用户关联 QQ 开放ID")  **private** Date qq\_openid;  /\*\*  \* 用户关联 微信 开放ID  \*/  @ApiModelProperty(value = "用户关联 微信 开放ID")  **private** Date WX\_OPENID;  } |

#### 服务接口实现

|  |
| --- |
| @RestController  **public** **class** MemberRegisterServiceImpl **extends** BaseApiService<JSONObject> **implements** MemberRegisterService {  @Autowired  **private** UserMapper userMapper;  @Autowired  **private** VerificaCodeServiceFeign verificaCodeServiceFeign;  @Override  **public** BaseResponse<JSONObject> register(@RequestBody UserEntity userEntity, String registCode) {  // 1.验证参数  String userName = userEntity.getUserName();  **if** (StringUtils.*isEmpty*(userName)) {  **return** setResultError("用户名称不能为空!");  }  String mobile = userEntity.getMobile();  **if** (StringUtils.*isEmpty*(mobile)) {  **return** setResultError("手机号码不能为空!");  }  String password = userEntity.getPassword();  **if** (StringUtils.*isEmpty*(password)) {  **return** setResultError("密码不能为空!");  }  String newPassWord = MD5Util.*MD5*(password);  // 将密码采用MD5加密  userEntity.setPassword(newPassWord);  // 调用微信接口,验证注册码是否正确  BaseResponse<JSONObject> resultVerificaWeixinCode = verificaCodeServiceFeign.verificaWeixinCode(mobile,  registCode);  **if** (!resultVerificaWeixinCode.getCode().equals(Constants.***HTTP\_RES\_CODE\_200***)) {  **return** setResultError(resultVerificaWeixinCode.getMsg());  }  **int** registerResult = userMapper.register(userEntity);  **return** registerResult > 0 ? setResultSuccess("注册成功") : setResultSuccess("注册失败");  }  } |

### 会员服务根据手机查询用户信息

|  |
| --- |
| @Api(tags = "会员服务接口")  **public** **interface** MemberService {  /\*\*  \* 根据手机号码查询是否已经存在,如果存在返回当前用户信息  \*  \* **@param** mobile  \* **@return**  \*/  @ApiOperation(value = "根据手机号码查询是否已经存在")  @ApiImplicitParams({  @ApiImplicitParam(paramType = "query", name = "mobile", dataType = "String", required = **true**, value = "用户手机号码"), })  @PostMapping("/existMobile")  BaseResponse<UserEntity> existMobile(@RequestParam("mobile") String mobile);  } |

#### 服务接口实现

|  |
| --- |
| @RestController  **public** **class** MemberServiceImpl **extends** BaseApiService<UserEntity> **implements** MemberService {  @Autowired  **private** UserMapper userMapper;  @Override  **public** BaseResponse<UserEntity> existMobile(String mobile) {  // 1.验证参数  **if** (StringUtils.*isEmpty*(mobile)) {  **return** setResultError("手机号码不能为空!");  }  UserEntity userEntity = userMapper.existMobile(mobile);  **if** (userEntity == **null**) {  **return** setResultError(Constants.***HTTP\_RES\_CODE\_EXISTMOBILE\_202***, "用户不存在");  }  // 注意需要将敏感数据进行脱敏  userEntity.setPassword(**null**);  **return** setResultSuccess(userEntity);  }  } |

### UserMapper

|  |
| --- |
| **public** **interface** UserMapper {  @Insert("INSERT INTO `meite\_user` VALUES (null,#{mobile}, #{email}, #{password}, #{userName}, null, null, null, '1', null, null, null);")  **int** register(UserEntity userEntity);  @Select("SELECT \* FROM meite\_user WHERE MOBILE=#{mobile};")  UserEntity existMobile(@Param("mobile") String mobile);  } |

### Maven依赖

|  |
| --- |
| <!-- mybatis相关依赖 -->  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.1.1</version>  </dependency>  <!-- mysql 依赖 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency> |

### MD5工具类

|  |
| --- |
| **public** **class** MD5Util {  **public** **final** **static** String MD5(String s) {  **char** hexDigits[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };  **try** {  **byte**[] btInput = s.getBytes();  // 获得MD5摘要算法的 MessageDigest 对象  MessageDigest mdInst = MessageDigest.*getInstance*("MD5");  // 使用指定的字节更新摘要  mdInst.update(btInput);  // 获得密文  **byte**[] md = mdInst.digest();  // 把密文转换成十六进制的字符串形式  **int** j = md.length;  **char** str[] = **new** **char**[j \* 2];  **int** k = 0;  **for** (**int** i = 0; i < j; i++) {  **byte** byte0 = md[i];  str[k++] = hexDigits[byte0 >>> 4 & 0xf];  str[k++] = hexDigits[byte0 & 0xf];  }  **return** **new** String(str);  } **catch** (Exception e) {  e.printStackTrace();  **return** **null**;  }  }  } |

### MD5工具类

|  |
| --- |
| spring:  # application:  # name: app-mayikt-weixin  redis:  host: 188.131.155.46  password: 123456  port: 6379  pool:  max-idle: 100  min-idle: 1  max-active: 1000  max-wait: -1  ###数据库相关连接  datasource:  username: root  password: root  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://127.0.0.1:3306/meite\_weixin |

### 常见问题

1.@RequestBody 必须要写在实现接口中

2.Feign 客户端调用的时候如果有参数的话，默认是发送post请求

3.服务接口中的请求参数必须要加上 @RequestParam("registCode")注解

思考： 开放接口实体类、数据库操作的实体类 、视图参数实体类

## 微服务接口细分安全领域DTO与DO层架构

### DTO与DO关系

在日常的项目开发中，VO对应于页面上需要显示的数据（表单），DO对应于数据库中存储的数据（数据表），DTO对应于除二者之外需要进行接口形式传递的数据。

传统方式

1、项目太小，对于一种业务实体，封装成一个DO就够了。

2、并不熟悉DTO、VO，更不知道他们之间的区别。

3、了解DO\DTO\VO之间的区别，但是懒得用。

### 构建meite-shop-api-dto

----meite-shop-api-dto

------ meite-shop-api-member-dto

------ meite-shop-api-weixin-dto

项目请求参数与响应参数

com.mayikt.member.output.dto ---输出参数

com.mayikt.member.input.dto---输入参数

### DTO与DO转换工具类

BeanUtils.copyProperties(dtoEntity, newInstance);

封装

|  |
| --- |
| **public** **class** MiteBeanUtils<Dto, Do> {  /\*\*  \* dot 转换为Do 工具类  \*  \* **@param** dtoEntity  \* **@param** doEntity  \* **@return**  \*/  **public** **static** <Do> Do dtoToDo(Object dtoEntity, Class<Do> doClass) {  // 判断dto是否为空!  **if** (dtoEntity == **null**) {  **return** **null**;  }  // 判断DoClass 是否为空  **if** (doClass == **null**) {  **return** **null**;  }  **try** {  Do newInstance = doClass.newInstance();  BeanUtils.*copyProperties*(dtoEntity, newInstance);  // Dto转换Do  **return** newInstance;  } **catch** (Exception e) {  **return** **null**;  }  }  /\*\*  \* do 转换为Dto 工具类  \*  \* **@param** dtoEntity  \* **@param** doEntity  \* **@return**  \*/  **public** **static** <Dto> Dto doToDto(Object doEntity, Class<Dto> dtoClass) {  // 判断dto是否为空!  **if** (doEntity == **null**) {  **return** **null**;  }  // 判断DoClass 是否为空  **if** (dtoClass == **null**) {  **return** **null**;  }  **try** {  Dto newInstance = dtoClass.newInstance();  BeanUtils.*copyProperties*(doEntity, newInstance);  // Dto转换Do  **return** newInstance;  } **catch** (Exception e) {  **return** **null**;  }  }  // 后面集合类型带封装  } |

### 整合全局捕获异常

|  |
| --- |
| @ControllerAdvice  @Slf4j  **public** **class** GlobalExceptionHandler **extends** BaseApiService<JSONObject> {  @ExceptionHandler(RuntimeException.**class**)  @ResponseBody  **public** BaseResponse<JSONObject> exceptionHandler(Exception e) {  ***log***.info("###全局捕获异常###,error:{}", e);  **return** setResultError("系统错误!");  }  } |

## 会员服务唯一登陆

### 登陆唯一表设计

|  |
| --- |
| CREATE TABLE `meite\_user\_token` (  `id` int(11) NOT NULL AUTO\_INCREMENT,  `token` varchar(255) DEFAULT NULL,  `login\_type` varchar(255) CHARACTER SET utf8 DEFAULT NULL,  `device\_infor` varchar(255) DEFAULT NULL,  `is\_availability` int(2) DEFAULT NULL,  `user\_id` int(11) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB AUTO\_INCREMENT=2 |

|  |
| --- |
| **public** **interface** UserMapper {  @Insert("INSERT INTO `meite\_user` VALUES (null,#{mobile}, #{email}, #{password}, #{userName}, null, null, null, '1', null, null, null);")  **int** register(UserDo userDo);  @Select("SELECT \* FROM meite\_user WHERE MOBILE=#{mobile};")  UserDo existMobile(@Param("mobile") String mobile);  @Select("SELECT USER\_ID AS USERID ,MOBILE AS MOBILE,EMAIL AS EMAIL,PASSWORD AS PASSWORD, USER\_NAME AS USER\_NAME ,SEX AS SEX ,AGE AS AGE ,CREATE\_TIME AS CREATETIME,IS\_AVALIBLE AS ISAVALIBLE,PIC\_IMG AS PICIMG,QQ\_OPENID AS QQOPENID,WX\_OPENID AS WXOPENID "  + " FROM meite\_user WHERE MOBILE=#{0} and password=#{1};")  UserDo login(@Param("mobile") String mobile, @Param("password") String password);  @Select("SELECT USER\_ID AS USERID ,MOBILE AS MOBILE,EMAIL AS EMAIL,PASSWORD AS PASSWORD, USER\_NAME AS USER\_NAME ,SEX AS SEX ,AGE AS AGE ,CREATE\_TIME AS CREATETIME,IS\_AVALIBLE AS ISAVALIBLE,PIC\_IMG AS PICIMG,QQ\_OPENID AS QQOPENID,WX\_OPENID AS WXOPENID"  + " FROM meite\_user WHERE user\_Id=#{userId}")  UserDo findByUserId(@Param("userId") Long userId);  } |

|  |
| --- |
| **public** **interface** UserTokenMapper {  @Select("SELECT id as id ,token as token ,login\_type as LoginType, device\_infor as deviceInfor ,is\_availability as isAvailability,user\_id as userId"  + "" + ""  + " , create\_time as createTime,update\_time as updateTime FROM meite\_user\_token WHERE user\_id=#{userId} AND login\_type=#{loginType} and is\_availability ='0'; ")  UserTokenDo selectByUserIdAndLoginType(@Param("userId") Long userId, @Param("loginType") String loginType);  @Update(" update meite\_user\_token set is\_availability ='1',update\_time=now() where user\_id=#{userId} and login\_type =#{loginType} ")  **int** updateTokenAvailability(@Param("userId") Long userId, @Param("loginType") String loginType);  // INSERT INTO `meite\_user\_token` VALUES ('2', '1', 'PC', '苹果7p', '1', '1');  @Insert(" INSERT INTO `meite\_user\_token` VALUES (null, #{token},#{loginType}, #{deviceInfor}, 0, #{userId} ,now(),null ); ")  **int** insertUserToken(UserTokenDo userTokenDo);  } |

|  |
| --- |
| @Data  **public** **class** UserTokenDo **extends** BaseDo {  /\*\*  \* id  \*/  **private** Long id;  /\*\*  \* 用户token  \*/  **private** String token;  /\*\*  \* 登陆类型  \*/  **private** String loginType;  /\*\*  \* 设备信息  \*/  **private** String deviceInfor;  /\*\*  \* 用户userId  \*/  **private** Long userId;  /\*\*  \* 注册时间  \*/  **private** Date createTime;  /\*\*  \* 修改时间  \*  \*/  **private** Date updateTime;  }  @Data  **public** **class** BaseDo {  /\*\*  \* 注册时间  \*/  **private** Date createTime;  /\*\*  \* 修改时间  \*  \*/  **private** Date updateTime;  /\*\*  \* id  \*/  **private** Long id;  /\*\*  \* 是否可用 0可用 1不可用  \*/  **private** Long isAvailability;  } |

### GenerateToken

|  |
| --- |
| @Component  **public** **class** GenerateToken {  @Autowired  **private** RedisUtil redisUtil;  /\*\*  \* 生成令牌  \*  \* **@param** prefix  \* 令牌key前缀  \* **@param** redisValue  \* redis存放的值  \* **@return** 返回token  \*/  **public** String createToken(String keyPrefix, String redisValue) {  **return** createToken(keyPrefix, redisValue, **null**);  }  /\*\*  \* 生成令牌  \*  \* **@param** prefix  \* 令牌key前缀  \* **@param** redisValue  \* redis存放的值  \* **@param** time  \* 有效期  \* **@return** 返回token  \*/  **public** String createToken(String keyPrefix, String redisValue, Long time) {  **if** (StringUtils.*isEmpty*(redisValue)) {  **new** Exception("redisValue Not nul");  }  String token = keyPrefix + UUID.*randomUUID*().toString().replace("-", "");  redisUtil.setString(token, redisValue, time);  **return** token;  }  /\*\*  \* 根据token获取redis中的value值  \*  \* **@param** token  \* **@return**  \*/  **public** String getToken(String token) {  **if** (StringUtils.*isEmpty*(token)) {  **return** **null**;  }  String value = redisUtil.getString(token);  **return** value;  }  /\*\*  \* 移除token  \*  \* **@param** token  \* **@return**  \*/  **public** Boolean removeToken(String token) {  **if** (StringUtils.*isEmpty*(token)) {  **return** **null**;  }  **return** redisUtil.delKey(token);  }  } |

### 新增常量信息

|  |
| --- |
| // token  String ***MEMBER\_TOKEN\_KEYPREFIX*** = "mayikt.member.login";  // 安卓的登陆类型  String ***MEMBER\_LOGIN\_TYPE\_ANDROID*** = "Android";  // IOS的登陆类型  String ***MEMBER\_LOGIN\_TYPE\_IOS*** = "IOS";  // PC的登陆类型  String ***MEMBER\_LOGIN\_TYPE\_PC*** = "PC";  // 登陆超时时间 有效期 90天  Long ***MEMBRE\_LOGIN\_TOKEN\_TIME*** = 77776000L; |

### 唯一登陆接口实现

|  |
| --- |
| @Data  @ApiModel(value = "用户登陆参数")  **public** **class** UserLoginInpDTO {  /\*\*  \* 手机号码  \*/  @ApiModelProperty(value = "手机号码")  **private** String mobile;  /\*\*  \* 密码  \*/  @ApiModelProperty(value = "密码")  **private** String password;  /\*\*  \* 登陆类型 PC、Android 、IOS  \*/  @ApiModelProperty(value = "登陆类型")  **private** String loginType;  /\*\*  \* 设备信息  \*/  @ApiModelProperty(value = "设备信息")  **private** String deviceInfor;  } |

|  |
| --- |
| @Api(tags = "用户登陆服务接口")  **public** **interface** MemberLoginService {  /\*\*  \* 用户登陆接口  \*  \* **@param** userEntity  \* **@return**  \*/  @PostMapping("/login")  @ApiOperation(value = "会员用户登陆信息接口")  BaseResponse<JSONObject> login(@RequestBody UserLoginInpDTO userLoginInpDTO);  }    @Autowired  **private** UserMapper userMapper;  @Autowired  **private** UserTokenMapper userTokenMapper;  @Autowired  **private** GenerateToken generateToken;  @Override  **public** BaseResponse<JSONObject> login(@RequestBody UserLoginInpDTO userLoginInpDTO) {  // 1.验证参数  String mobile = userLoginInpDTO.getMobile();  **if** (StringUtils.*isEmpty*(mobile)) {  **return** setResultError("手机号码不能为空!");  }  String password = userLoginInpDTO.getPassword();  **if** (StringUtils.*isEmpty*(password)) {  **return** setResultError("密码不能为空!");  }  String loginType = userLoginInpDTO.getLoginType();  **if** (StringUtils.*isEmpty*(loginType)) {  **return** setResultError("登陆类型不能为空!");  }  **if** (!(loginType.equals(Constants.***MEMBER\_LOGIN\_TYPE\_ANDROID***) || loginType.equals(Constants.***MEMBER\_LOGIN\_TYPE\_IOS***)  || loginType.equals(Constants.***MEMBER\_LOGIN\_TYPE\_PC***))) {  **return** setResultError("登陆类型出现错误!");  }  // 设备信息  String deviceInfor = userLoginInpDTO.getDeviceInfor();  **if** (StringUtils.*isEmpty*(deviceInfor)) {  **return** setResultError("设备信息不能为空!");  }  String newPassWord = MD5Util.*MD5*(password);  // 2.用户名称与密码登陆  UserDo userDo = userMapper.login(mobile, newPassWord);  **if** (userDo == **null**) {  **return** setResultError("用户名称与密码错误!");  }  // 3.查询之前是否有过登陆  Long userId = userDo.getUserId();  UserTokenDo userTokenDo = userTokenMapper.selectByUserIdAndLoginType(userId, loginType);  **if** (userTokenDo != **null**) {  // 4.清除之前的token  String token = userTokenDo.getToken();  Boolean removeToken = generateToken.removeToken(token);  **if** (removeToken) {  userTokenMapper.updateTokenAvailability(userId, loginType);  }  }  // 5. 生成新的token  String token = generateToken.createToken(Constants.***MEMBER\_TOKEN\_KEYPREFIX***, userId + "",  Constants.***MEMBRE\_LOGIN\_TOKEN\_TIME***);  JSONObject tokenData = **new** JSONObject();  tokenData.put("token", token);  // 6.存入在数据库中  UserTokenDo userToken = **new** UserTokenDo();  userToken.setUserId(userId);  userToken.setLoginType(userLoginInpDTO.getLoginType());  userToken.setToken(token);  userToken.setDeviceInfor(deviceInfor);  userTokenMapper.insertUserToken(userToken);  **return** setResultSuccess(tokenData);  } |

### 根据Token查询用户信息

|  |
| --- |
| /\*\*  \* 根据token查询用户信息  \*  \* **@param** userEntity  \* **@return**  \*/  @GetMapping("/getUserInfo")  @ApiOperation(value = "/getUserInfo")  BaseResponse<UserOutDTO> getInfo(@RequestParam("token") String token);  @Override  **public** BaseResponse<UserOutDTO> getInfo(String token) {  // 1.参数验证  **if** (StringUtils.*isEmpty*(token)) {  **return** setResultError("token不能为空!");  }  // 2.使用token向redis中查询userId  String redisValue = generateToken.getToken(token);  **if** (StringUtils.*isEmpty*(redisValue)) {  **return** setResultError("token已经失效或者不正确");  }  Long userId = TypeCastHelper.*toLong*(redisValue);  // 3.根据userId查询用户信息  UserDo userDo = userMapper.findByUserId(userId);  **if** (userDo == **null**) {  **return** setResultError("用户信息不存在!");  }  // 4.将Do转换为Dto  UserOutDTO doToDto = MeiteBeanUtils.*doToDto*(userDo, UserOutDTO.**class**);  **return** setResultSuccess(doToDto);  } |

### 转换类型工具类

|  |
| --- |
| **public** **class** TypeCastHelper {  **private** **static** **final** String ***DEFAULT\_STRING*** = "";  **private** **static** **final** **int** ***DEFAULT\_INT*** = 0;  **private** **static** **final** **long** ***DEFAULT\_LONG*** = 0L;  **private** **static** **final** **double** ***DEFAULT\_DOUBLE*** = 0.0D;  **private** **static** **final** **float** ***DEFAULT\_FLOAT*** = 0.0F;  **private** **static** **final** **boolean** ***DEFAULT\_BOOLEAN*** = Boolean.***FALSE***;  **public** **static** String toString(Object obj, String defaultValue) {  **return** obj != **null** ? obj.toString() : defaultValue;  }  **public** **static** String toString(Object obj) {  **return** *toString*(obj, ***DEFAULT\_STRING***);  }  **public** **static** String toString(Integer intValue, String defaultValue) {  **return** intValue != **null** ? String.*valueOf*(intValue) : defaultValue;  }  **public** **static** String toString(Integer intValue) {  **return** *toString*(intValue, ***DEFAULT\_STRING***);  }  **public** **static** String toString(Long longValue, String defaultValue) {  **return** longValue != **null** ? String.*valueOf*(longValue) : defaultValue;  }  **public** **static** String toString(Long longValue) {  **return** *toString*(longValue, ***DEFAULT\_STRING***);  }  **public** **static** String toString(Boolean booleanValue, String defaultValue) {  **return** booleanValue != **null** ? String.*valueOf*(booleanValue) : defaultValue;  }  **public** **static** String toString(Boolean booleanValue) {  **return** *toString*(booleanValue, ***DEFAULT\_STRING***);  }  **public** **static** String toString(Double doubleValue, String defaultValue) {  **return** **null** != doubleValue ? String.*valueOf*(doubleValue) : defaultValue;  }  **public** **static** String toString(Double doubleValue) {  **return** *toString*(doubleValue, ***DEFAULT\_STRING***);  }  **public** **static** String toString(Float floatValue, String defaultValue) {  **return** **null** != floatValue ? String.*valueOf*(floatValue) : defaultValue;  }  **public** **static** String toString(Float floatValue) {  **return** *toString*(floatValue, ***DEFAULT\_STRING***);  }  **public** **static** **int** toInt(Object objectValue, **int** defaultValue) {  **int** intValue = defaultValue;  **if** (**null** != objectValue) {  **try** {  intValue = Integer.*parseInt*(*toString*(objectValue, *toString*(defaultValue)));  } **catch** (NumberFormatException numberFormatException) {  System.***err***.println(numberFormatException.getMessage());  }  }  **return** intValue;  }  **public** **static** **int** toInt(Object objectValue) {  **return** *toInt*(objectValue, ***DEFAULT\_INT***);  }  **public** **static** **int** toInt(String stringValue, **int** defaultValue) {  **int** intValue = defaultValue;  **if** (**null** != stringValue && stringValue.length() > 0) {  **try** {  intValue = Integer.*parseInt*(stringValue);  } **catch** (NumberFormatException numberFormatException) {  System.***err***.println(numberFormatException.getMessage());  }  }  **return** intValue;  }  **public** **static** **int** toInt(Long longValue, **int** defaultValue) {  **int** intValue = defaultValue;  **if** (**null** != longValue) {  intValue = longValue.intValue();  }  **return** intValue;  }  **public** **static** **int** toInt(Long longValue) {  **return** *toInt*(longValue, ***DEFAULT\_INT***);  }  **public** **static** **int** toInt(Boolean booleanValue, **int** defaultValue) {  **int** intValue = defaultValue;  **if** (**null** != booleanValue) {  intValue = booleanValue ? 1 : 0;  }  **return** intValue;  }  **public** **static** **int** toInt(Boolean booleanValue) {  **return** *toInt*(booleanValue, ***DEFAULT\_INT***);  }  **public** **static** **int** toInt(Double doubleValue, **int** defaultValue) {  **int** intValue = defaultValue;  **if** (**null** != doubleValue) {  intValue = doubleValue.intValue();  }  **return** intValue;  }  **public** **static** **int** toInt(Double doubleValue) {  **return** *toInt*(doubleValue, ***DEFAULT\_INT***);  }  **public** **static** **int** toInt(Float floatValue, **int** defaultValue) {  **int** intValue = defaultValue;  **if** (**null** != floatValue) {  intValue = floatValue.intValue();  }  **return** intValue;  }  **public** **static** **int** toInt(Float floatValue) {  **return** *toInt*(floatValue, ***DEFAULT\_INT***);  }  **public** **static** **long** toLong(Object objectValue, **long** defaultValue) {  **long** longValue = defaultValue;  **if** (**null** != objectValue) {  **try** {  longValue = Long.*parseLong*(*toString*(objectValue, *toString*(defaultValue)));  } **catch** (NumberFormatException numberFormatException) {  System.***err***.println(numberFormatException.getMessage());  }  }  **return** longValue;  }  **public** **static** **long** toLong(Object objectValue) {  **return** *toLong*(objectValue, ***DEFAULT\_LONG***);  }  **public** **static** **long** toLong(String stringValue, **long** defaultValue) {  **long** longValue = defaultValue;  **if** (**null** != stringValue && stringValue.length() > 0) {  longValue = Long.*parseLong*(stringValue);  }  **return** longValue;  }  **public** **static** **long** toLong(String stringValue) {  **return** *toLong*(stringValue, ***DEFAULT\_LONG***);  }  **public** **static** **long** toLong(Integer intValue, **long** defaultValue) {  **long** longValue = defaultValue;  **if** (**null** != intValue) {  longValue = intValue.longValue();  }  **return** longValue;  }  **public** **static** **long** toLong(Integer intValue) {  **return** *toLong*(intValue, ***DEFAULT\_LONG***);  }  **public** **static** **long** toLong(Boolean booleanValue, **long** defaultValue) {  **long** longValue = defaultValue;  **if** (**null** != booleanValue) {  longValue = booleanValue ? 1L : 0L;  }  **return** longValue;  }  **public** **static** **long** toLong(Boolean booleanValue) {  **return** *toLong*(booleanValue, ***DEFAULT\_LONG***);  }  **public** **static** **long** toLong(Double doubleValue, **long** defaultValue) {  **long** longValue = defaultValue;  **if** (**null** != doubleValue) {  longValue = doubleValue.longValue();  }  **return** longValue;  }  **public** **static** **long** toLong(Double doubleValue) {  **return** *toLong*(doubleValue, ***DEFAULT\_LONG***);  }  **public** **static** **long** toLong(Float floatValue, **long** defaultValue) {  **long** longValue = defaultValue;  **if** (**null** != floatValue) {  longValue = floatValue.longValue();  }  **return** longValue;  }  **public** **static** **long** toLong(Float floatValue) {  **return** *toLong*(floatValue, ***DEFAULT\_LONG***);  }  **public** **static** **boolean** toBoolean(Object objectValue, **boolean** defaultValue) {  **boolean** booleanValue = defaultValue;  **if** (**null** != objectValue) {  booleanValue = Boolean.*valueOf*(*toString*(objectValue, *toString*(defaultValue)));  }  **return** booleanValue;  }  **public** **static** **boolean** toBoolean(Object objectValue) {  **return** *toBoolean*(objectValue, ***DEFAULT\_BOOLEAN***);  }  **public** **static** **boolean** toBoolean(String stringValue, **boolean** defaultValue) {  **boolean** booleanValue = defaultValue;  **if** (**null** != stringValue && stringValue.length() > 0) {  booleanValue = Boolean.*valueOf*(stringValue);  }  **return** booleanValue;  }  **public** **static** **boolean** toBoolean(String stringValue) {  **return** *toBoolean*(stringValue, ***DEFAULT\_BOOLEAN***);  }  **public** **static** **boolean** toBoolean(Integer intValue, **boolean** defaultValue) {  **boolean** booleanValue = defaultValue;  **if** (**null** != intValue) {  booleanValue = intValue == 1;  }  **return** booleanValue;  }  **public** **static** **boolean** toBoolean(Integer intValue) {  **return** *toBoolean*(intValue, ***DEFAULT\_BOOLEAN***);  }  **public** **static** **boolean** toBoolean(Long longValue, **boolean** defaultValue) {  **boolean** booleanValue = defaultValue;  **if** (**null** != longValue) {  booleanValue = longValue.longValue() == 1L;  }  **return** booleanValue;  }  **public** **static** **boolean** toBoolean(Long longValue) {  **return** *toBoolean*(longValue, ***DEFAULT\_BOOLEAN***);  }  **public** **static** **float** toFloat(Object objectValue, **float** defaultValue) {  **float** floatValue = defaultValue;  **if** (**null** != objectValue) {  **try** {  floatValue = Float.*valueOf*(*toString*(objectValue, *toString*(defaultValue)));  } **catch** (NumberFormatException numberFormatException) {  System.***err***.println(numberFormatException);  }  }  **return** floatValue;  }  **public** **static** **float** toFloat(Object objectValue) {  **return** *toFloat*(objectValue, ***DEFAULT\_FLOAT***);  }  **public** **static** **float** toFloat(String stringValue, **float** defaultValue) {  **float** floatValue = defaultValue;  **if** (**null** != stringValue && stringValue.length() > 0) {  **try** {  floatValue = Float.*valueOf*(stringValue);  } **catch** (NumberFormatException numberFormatException) {  System.***err***.println(numberFormatException.getMessage());  }  }  **return** floatValue;  }  **public** **static** **float** toFloat(String stringValue) {  **return** *toFloat*(stringValue, ***DEFAULT\_FLOAT***);  }  **public** **static** **float** toFloat(Integer intValue, **float** defaultValue) {  **float** floatValue = defaultValue;  **if** (**null** != intValue) {  floatValue = intValue.floatValue();  }  **return** floatValue;  }  **public** **static** **float** toFloat(Integer intValue) {  **return** *toFloat*(intValue, ***DEFAULT\_FLOAT***);  }  **public** **static** **float** toFloat(Long longValue, **float** defaultValue) {  **float** floatValue = defaultValue;  **if** (**null** != longValue) {  floatValue = longValue.floatValue();  }  **return** floatValue;  }  **public** **static** **float** toFloat(Long longValue) {  **return** *toFloat*(longValue, ***DEFAULT\_FLOAT***);  }  **public** **static** **float** toFloat(Double doubleValue, **float** defaultValue) {  **float** floatValue = defaultValue;  **if** (**null** != doubleValue) {  floatValue = doubleValue.floatValue();  }  **return** floatValue;  }  **public** **static** **float** toFloat(Double doubleValue) {  **return** *toFloat*(doubleValue, ***DEFAULT\_FLOAT***);  }  **public** **static** **double** toDouble(Object objectValue, **double** defaultValue) {  **double** doubleValue = defaultValue;  **if** (**null** != objectValue) {  doubleValue = Double.*parseDouble*(*toString*(objectValue, *toString*(defaultValue)));  }  **return** doubleValue;  }  **public** **static** **double** toDouble(Object objectValue) {  **return** *toDouble*(objectValue, ***DEFAULT\_DOUBLE***);  }  **public** **static** **double** toDouble(String stringValue, **double** defaultValue) {  **double** doubleValue = defaultValue;  **if** (**null** != stringValue && stringValue.length() > 0) {  doubleValue = Double.*parseDouble*(stringValue);  }  **return** doubleValue;  }  **public** **static** **double** toDouble(String stringValue) {  **return** *toDouble*(stringValue, ***DEFAULT\_DOUBLE***);  }  **public** **static** **double** toDouble(Integer intValue, **double** defaultValue) {  **double** doubleValue = defaultValue;  **if** (**null** != intValue) {  doubleValue = intValue.doubleValue();  }  **return** doubleValue;  }  **public** **static** **double** toDouble(Integer intValue) {  **return** *toDouble*(intValue, ***DEFAULT\_DOUBLE***);  }  **public** **static** **double** toDouble(Long longValue, **double** defaultValue) {  **double** doubleValue = defaultValue;  **if** (**null** != longValue) {  doubleValue = longValue.doubleValue();  }  **return** doubleValue;  }  **public** **static** **double** toDouble(Long longValue) {  **return** *toDouble*(longValue, ***DEFAULT\_DOUBLE***);  }  **public** **static** **double** toDouble(Float floatValue, **double** defaultValue) {  **double** doubleValue = defaultValue;  **if** (**null** != floatValue) {  doubleValue = floatValue.doubleValue();  }  **return** doubleValue;  }  **public** **static** **double** toDouble(Float floatValue) {  **return** *toDouble*(floatValue, ***DEFAULT\_DOUBLE***);  }  } |

### 笔记

假设一个用户登陆可以同时在PC、IOS、Android 最终生成三个Token

mayikt.member.login8faa6de4327e4086a666ef1221284dfb –PC登陆

mayikt.member.login8faa6de4327e4086a666ef1221284dfb –redis存放PCtoken

## 会员登陆模块优化与门户平台整合

### BaseApiService新增

|  |
| --- |
| // 调用数据库层判断  **public** Boolean toDaoResult(**int** result) {  **return** result > 0 ? **true** : **false**;  } |

### 修改RedisUtil

|  |
| --- |
| @Component  **public** **class** RedisUtil {  @Autowired  **private** StringRedisTemplate stringRedisTemplate;  /\*\*  \* 存放string类型  \*  \* **@param** key  \* key  \* **@param** data  \* 数据  \* **@param** timeout  \* 超时间  \*/  **public** **void** setString(String key, String data, Long timeout) {  **try** {  stringRedisTemplate.opsForValue().set(key, data);  **if** (timeout != **null**) {  stringRedisTemplate.expire(key, timeout, TimeUnit.***SECONDS***);  }  } **catch** (Exception e) {  }  }  /\*\*  \* 开启Redis 事务  \*  \* **@param** isTransaction  \*/  **public** **void** begin() {  // 开启Redis 事务权限  stringRedisTemplate.setEnableTransactionSupport(**true**);  // 开启事务  stringRedisTemplate.multi();  }  /\*\*  \* 提交事务  \*  \* **@param** isTransaction  \*/  **public** **void** exec() {  // 成功提交事务  stringRedisTemplate.exec();  }  /\*\*  \* 回滚Redis 事务  \*/  **public** **void** discard() {  stringRedisTemplate.discard();  }  /\*\*  \* 存放string类型  \*  \* **@param** key  \* key  \* **@param** data  \* 数据  \*/  **public** **void** setString(String key, String data) {  setString(key, data, **null**);  }  /\*\*  \* 根据key查询string类型  \*  \* **@param** key  \* **@return**  \*/  **public** String getString(String key) {  String value = stringRedisTemplate.opsForValue().get(key);  **return** value;  }  /\*\*  \* 根据对应的key删除key  \*  \* **@param** key  \*/  **public** Boolean delKey(String key) {  **return** stringRedisTemplate.delete(key);  }  } |

### RedisDataSoureceTransaction

|  |
| --- |
| @Component  @Scope(ConfigurableListableBeanFactory.***SCOPE\_PROTOTYPE***)  **public** **class** RedisDataSoureceTransaction {  @Autowired  **private** RedisUtil redisUtil;  /\*\*  \* 数据源事务管理器  \*/  @Autowired  **private** DataSourceTransactionManager dataSourceTransactionManager;  /\*\*  \* 开始事务 采用默认传播行为  \*  \* **@return**  \*/  **public** TransactionStatus begin() {  // 手动begin数据库事务  // 1.开启数据库的事务 事务传播行为  TransactionStatus transaction = dataSourceTransactionManager.getTransaction(**new** DefaultTransactionAttribute());  // 2.开启redis事务  redisUtil.begin();  **return** transaction;  }  /\*\*  \* 提交事务  \*  \* **@param** transactionStatus  \* 事务传播行为  \* **@throws** Exception  \*/  **public** **void** commit(TransactionStatus transactionStatus) **throws** Exception {  **if** (transactionStatus == **null**) {  **throw** **new** Exception("transactionStatus is null");  }  // 支持Redis与数据库事务同时提交  dataSourceTransactionManager.commit(transactionStatus);  }  /\*\*  \* 回滚事务  \*  \* **@param** transactionStatus  \* **@throws** Exception  \*/  **public** **void** rollback(TransactionStatus transactionStatus) **throws** Exception {  **if** (transactionStatus == **null**) {  **throw** **new** Exception("transactionStatus is null");  }  // 1.回滚数据库事务 redis事务和数据库的事务同时回滚  dataSourceTransactionManager.rollback(transactionStatus);  // // 2.回滚redis事务  // redisUtil.discard();  }  // 如果redis的值与数据库的值保持不一致话  } |

### 登陆接口业务逻辑

|  |
| --- |
| @RestController  **public** **class** MemberLoginServiceImpl **extends** BaseApiService<JSONObject> **implements** MemberLoginService {  @Autowired  **private** UserMapper userMapper;  @Autowired  **private** GenerateToken generateToken;  @Autowired  **private** UserTokenMapper userTokenMapper;  /\*\*  \* 手动事务工具类  \*/  @Autowired  **private** RedisDataSoureceTransaction manualTransaction;  /\*\*  \* redis 工具类  \*/  @Autowired  **private** RedisUtil redisUtil;  @Override  **public** BaseResponse<JSONObject> login(@RequestBody UserLoginInpDTO userLoginInpDTO) {  // 1.验证参数  String mobile = userLoginInpDTO.getMobile();  **if** (StringUtils.*isEmpty*(mobile)) {  **return** setResultError("手机号码不能为空!");  }  String password = userLoginInpDTO.getPassword();  **if** (StringUtils.*isEmpty*(password)) {  **return** setResultError("密码不能为空!");  }  // 判断登陆类型  String loginType = userLoginInpDTO.getLoginType();  **if** (StringUtils.*isEmpty*(loginType)) {  **return** setResultError("登陆类型不能为空!");  }  // 目的是限制范围  **if** (!(loginType.equals(Constants.***MEMBER\_LOGIN\_TYPE\_ANDROID***) || loginType.equals(Constants.***MEMBER\_LOGIN\_TYPE\_IOS***)  || loginType.equals(Constants.***MEMBER\_LOGIN\_TYPE\_PC***))) {  **return** setResultError("登陆类型出现错误!");  }  // 设备信息  String deviceInfor = userLoginInpDTO.getDeviceInfor();  **if** (StringUtils.*isEmpty*(deviceInfor)) {  **return** setResultError("设备信息不能为空!");  }  // 2.对登陆密码实现加密  String newPassWord = MD5Util.*MD5*(password);  // 3.使用手机号码+密码查询数据库 ，判断用户是否存在  UserDo userDo = userMapper.login(mobile, newPassWord);  **if** (userDo == **null**) {  **return** setResultError("用户名称或者密码错误!");  }  TransactionStatus transactionStatus = **null**;  **try** {  // 1.获取用户UserId  Long userId = userDo.getUserId();  // 2.生成用户令牌Key  String keyPrefix = Constants.***MEMBER\_TOKEN\_KEYPREFIX*** + loginType;  // 5.根据userId+loginType 查询当前登陆类型账号之前是否有登陆过，如果登陆过 清除之前redistoken  UserTokenDo userTokenDo = userTokenMapper.selectByUserIdAndLoginType(userId, loginType);  transactionStatus = manualTransaction.begin();  // // ####开启手动事务  **if** (userTokenDo != **null**) {  // 如果登陆过 清除之前redistoken  String oriToken = userTokenDo.getToken();  // 移除Token  generateToken.removeToken(oriToken);  **int** updateTokenAvailability = userTokenMapper.updateTokenAvailability(oriToken);  **if** (updateTokenAvailability < 0) {  manualTransaction.rollback(transactionStatus);  **return** setResultError("系统错误");  }  }  // 4.将用户生成的令牌插入到Token记录表中  UserTokenDo userToken = **new** UserTokenDo();  userToken.setUserId(userId);  userToken.setLoginType(userLoginInpDTO.getLoginType());  String newToken = generateToken.createToken(keyPrefix, userId + "");  userToken.setToken(newToken);  userToken.setDeviceInfor(deviceInfor);  **int** result = userTokenMapper.insertUserToken(userToken);  **if** (!toDaoResult(result)) {  manualTransaction.rollback(transactionStatus);  **return** setResultError("系统错误!");  }  // #######提交事务  JSONObject data = **new** JSONObject();  data.put("token", newToken);  manualTransaction.commit(transactionStatus);  **return** setResultSuccess(data);  } **catch** (Exception e) {  **try** {  // 回滚事务  manualTransaction.rollback(transactionStatus);  } **catch** (Exception e1) {  }  **return** setResultError("系统错误!");  }  }  } |

### 创建meite-shop-portal-web

#### 首页控制层

|  |
| --- |
| @Controller  **public** **class** IndexController {  /\*\*  \* 跳转到首页  \*  \* **@return**  \*/  @RequestMapping("/")  **public** String index() {  **return** "index";  }  /\*\*  \* 跳转到首页  \*  \* **@return**  \*/  @RequestMapping("/index.html")  **public** String indexHtml() {  **return** index();  }  } |

#### 登陆控制层

|  |
| --- |
| @Controller  **public** **class** LoginController {  **private** **static** **final** String ***MEMBER\_LOGIN\_PAGE*** = "member/login";  /\*\*  \* 跳转到注册页面  \*  \* **@return**  \*/  @GetMapping("/login.html")  **public** String getLogin() {  **return** ***MEMBER\_LOGIN\_PAGE***;  }  /\*\*  \* 跳转到注册页面  \*  \* **@return**  \*/  @PostMapping("/login.html")  **public** String postLogin() {  **return** **null**;  }  } |

#### 注册控制层

|  |
| --- |
| @Controller  **public** **class** RegisterController {  **private** **static** **final** String ***MEMBER\_REGISTER\_PAGE*** = "member/register";  /\*\*  \* 跳转到注册页面  \*  \* **@return**  \*/  @GetMapping("/register.html")  **public** String getRegister() {  **return** ***MEMBER\_REGISTER\_PAGE***;  }  /\*\*  \* 跳转到注册页面  \*  \* **@return**  \*/  @PostMapping("/register.html")  **public** String postRegister() {  **return** **null**;  }  } |

#### Maven依赖

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

#### application.yml

|  |
| --- |
| #### 整合freemarker  spring:  freemarker:  cache: **false**  charset: UTF-8  check-template-location: **true**  content-type: text/html  expose-request-attributes: **true**  expose-session-attributes: **true**  request-context-attribute: request  suffix: .ftl  template-loader-path:  - classpath:/templates  application:  name: app-mayikt-portal-web  ###服务注册到eureka地址  eureka:  client:  service-url:  defaultZone: http://localhost:8100/eureka |

#### 页面详见资料

笔记 redis 目前存在的token mt.mb.loginAndroidd116885178a14f4ca8046e2afb214d5

新token

mt.mb.loginAndroid41176d288b76415794b831fa0c43e53b

数据库中的token mt.mb.loginAndroiddccc3c55724c4bb3b060cec5df3a87bc

Redis 中的token mt.mb.loginAndroiddccc3c55724c4bb3b060cec5df3a87bc

新的redis中的token mt.mb.loginAndroid8735ce40754d41e5b4f93d52b338f178

数据库中的token mt.mb.loginAndroid8735ce40754d41e5b4f93d52b338f178

## 实现门户网站会员登陆和注册

### 创建图形验证码工具类

|  |
| --- |
| @Controller  **public** **class** VerifyController {  /\*\*  \* 生成验证码  \*/  @RequestMapping(value = "/getVerify")  **public** **void** getVerify(HttpServletRequest request, HttpServletResponse response) {  **try** {  response.setContentType("image/jpeg");// 设置相应类型,告诉浏览器输出的内容为图片  response.setHeader("Pragma", "No-cache");// 设置响应头信息，告诉浏览器不要缓存此内容  response.setHeader("Cache-Control", "no-cache");  response.setDateHeader("Expire", 0);  RandomValidateCodeUtil randomValidateCode = **new** RandomValidateCodeUtil();  randomValidateCode.getRandcode(request, response);// 输出验证码图片方法  } **catch** (Exception e) {  }  }  }  **public** **class** RandomValidateCodeUtil {  **public** **static** **final** String ***RANDOMCODEKEY*** = "RANDOMVALIDATECODEKEY";// 放到session中的key  **private** String randString = "0123456789";// 随机产生只有数字的字符串 private String  // private String randString = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";//随机产生只有字母的字符串  // private String randString =  // "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";//随机产生数字与字母组合的字符串  **private** **int** width = 95;// 图片宽  **private** **int** height = 25;// 图片高  **private** **int** lineSize = 40;// 干扰线数量  **private** **int** stringNum = 4;// 随机产生字符数量  **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(RandomValidateCodeUtil.**class**);  **private** Random random = **new** Random();  /\*\*  \* 获得字体  \*/  **private** Font getFont() {  **return** **new** Font("Fixedsys", Font.***CENTER\_BASELINE***, 18);  }  /\*\*  \* 获得颜色  \*/  **private** Color getRandColor(**int** fc, **int** bc) {  **if** (fc > 255)  fc = 255;  **if** (bc > 255)  bc = 255;  **int** r = fc + random.nextInt(bc - fc - 16);  **int** g = fc + random.nextInt(bc - fc - 14);  **int** b = fc + random.nextInt(bc - fc - 18);  **return** **new** Color(r, g, b);  }  /\*\*  \* 生成随机图片  \*/  **public** **void** getRandcode(HttpServletRequest request, HttpServletResponse response) {  HttpSession session = request.getSession();  // BufferedImage类是具有缓冲区的Image类,Image类是用于描述图像信息的类  BufferedImage image = **new** BufferedImage(width, height, BufferedImage.***TYPE\_INT\_BGR***);  Graphics g = image.getGraphics();// 产生Image对象的Graphics对象,改对象可以在图像上进行各种绘制操作  g.fillRect(0, 0, width, height);// 图片大小  g.setFont(**new** Font("Times New Roman", Font.***ROMAN\_BASELINE***, 18));// 字体大小  g.setColor(getRandColor(110, 133));// 字体颜色  // 绘制干扰线  **for** (**int** i = 0; i <= lineSize; i++) {  drowLine(g);  }  // 绘制随机字符  String randomString = "";  **for** (**int** i = 1; i <= stringNum; i++) {  randomString = drowString(g, randomString, i);  }  ***logger***.info(randomString);  // 将生成的随机字符串保存到session中  session.removeAttribute(***RANDOMCODEKEY***);  session.setAttribute(***RANDOMCODEKEY***, randomString);  g.dispose();  **try** {  // 将内存中的图片通过流动形式输出到客户端  ImageIO.*write*(image, "JPEG", response.getOutputStream());  } **catch** (Exception e) {  ***logger***.error("将内存中的图片通过流动形式输出到客户端失败>>>> ", e);  }  }  /\*\*  \* 绘制字符串  \*/  **private** String drowString(Graphics g, String randomString, **int** i) {  g.setFont(getFont());  g.setColor(**new** Color(random.nextInt(101), random.nextInt(111), random.nextInt(121)));  String rand = String.*valueOf*(getRandomString(random.nextInt(randString.length())));  randomString += rand;  g.translate(random.nextInt(3), random.nextInt(3));  g.drawString(rand, 13 \* i, 16);  **return** randomString;  }  /\*\*  \* 绘制干扰线  \*/  **private** **void** drowLine(Graphics g) {  **int** x = random.nextInt(width);  **int** y = random.nextInt(height);  **int** xl = random.nextInt(13);  **int** yl = random.nextInt(15);  g.drawLine(x, y, x + xl, y + yl);  }  /\*\*  \* 获取随机的字符  \*/  **public** String getRandomString(**int** num) {  **return** String.*valueOf*(randString.charAt(num));  }  **public** **static** Boolean checkVerify(String verifiCode, HttpSession httpSession) {  **if** (StringUtils.*isEmpty*(verifiCode)) {  **return** **false**;  }  String random = (String) httpSession.getAttribute("RANDOMVALIDATECODEKEY");  **if** (StringUtils.*isEmpty*(random)) {  **return** **false**;  }  **if** (!random.equals(verifiCode)) {  **return** **false**;  }  **return** **true**;  }  } |

### 注册功能实现

#### 注册控制类

|  |
| --- |
| @Controller  @Slf4j  **public** **class** RegisterController **extends** BaseWebController {  **private** **static** **final** String ***MB\_REGISTER\_FTL*** = "member/register";  /\*\*  \* 跳转到登陆页面页面  \*/  **private** **static** **final** String ***MB\_LOGIN\_FTL*** = "member/login";  @Autowired  **private** MemberRegisterServiceFeign memberRegisterServiceFeign;  /\*\*  \* 跳转到注册页面  \*  \* **@return**  \*/  @GetMapping("/register")  **public** String getRegister() {  **return** ***MB\_REGISTER\_FTL***;  }  /\*\*  \* 跳转到注册页面  \*  \* **@return**  \*/  @PostMapping("/register")  **public** String postRegister(@ModelAttribute("registerVo") @Validated RegisterVo registerVo,  BindingResult bindingResult, HttpSession httpSession, Model model) {  // 1.参数验证  **if** (bindingResult.hasErrors()) {  // 获取第一个错误!  String errorMsg = bindingResult.getFieldError().getDefaultMessage();  setErrorMsg(model, errorMsg);  **return** ***MB\_REGISTER\_FTL***;  }  // 将VO转换DTO  UserInpDTO voToDto = MeiteBeanUtils.*voToDto*(registerVo, UserInpDTO.**class**);  **try** {  String registCode = registerVo.getRegistCode();  BaseResponse<JSONObject> register = memberRegisterServiceFeign.register(voToDto, registCode);  **if** (!register.getCode().equals(Constants.***HTTP\_RES\_CODE\_200***)) {  model.addAttribute("error", register.getMsg());  **return** ***MB\_REGISTER\_FTL***;  }  } **catch** (Exception e) {  ***log***.error(">>>>>", e);  model.addAttribute("error", "系统出现错误!");  **return** ***MB\_REGISTER\_FTL***;  }  // 注册成功,跳转到登陆页面  **return** ***MB\_LOGIN\_FTL***;  }  } |

#### 使用注解验证参数

|  |
| --- |
| @Data  @Valid  **public** **class** RegisterVo {  /\*\*  \* 手机号码  \*/  @NotBlank(message = "手机号码不能为空")  @Size(min = 11, max = 11, message = "手机号码长度不正确")  @Pattern(regexp = "^(((13[0-9])|(14[579])|(15([0-3]|[5-9]))|(16[6])|(17[0135678])|(18[0-9])|(19[89]))\\d{8})$", message = "手机号格式错误")  **private** String mobile;  /\*\*  \* 密码  \*/  @NotNull(message = "密码不能为空!")  **private** String password;  /\*\*  \* 注册码  \*/  @NotNull(message = "注册码不能为空!")  **private** String registCode;  /\*\*  \* 图形验证码  \*/  @NotBlank(message = "图形验证码不能为空!")  **private** String graphicCode;  } |

#### Vo转Dto工具类

|  |
| --- |
| **public** **class** MeiteBeanUtils<Vo, Dto> {  /\*\*  \* dot 转换为Do 工具类  \*  \* **@param** dtoEntity  \* **@param** doEntity  \* **@return**  \*/  **public** **static** <Dto> Dto voToDto(Object voEntity, Class<Dto> dtoClass) {  // 判断VoSF是否为空!  **if** (voEntity == **null**) {  **return** **null**;  }  // 判断DtoClass 是否为空  **if** (dtoClass == **null**) {  **return** **null**;  }  **try** {  Dto newInstance = dtoClass.newInstance();  BeanUtils.*copyProperties*(voEntity, newInstance);  // Dto转换Do  **return** newInstance;  } **catch** (Exception e) {  **return** **null**;  }  }  // 后面集合类型带封装  } |

#### BaseWebController

|  |
| --- |
| **public** **class** BaseWebController {  /\*\*  \* 500页面  \*/  **protected** **static** **final** String ***ERROR\_500\_FTL*** = "500.ftl";  **public** Boolean isSuccess(BaseResponse<?> baseResp) {  **if** (baseResp == **null**) {  **return** **false**;  }  **if** (!baseResp.getCode().equals(Constants.***HTTP\_RES\_CODE\_200***)) {  **return** **false**;  }  **return** **true**;  }  **public** **void** setErrorMsg(Model model, String errorMsg) {  model.addAttribute("error", errorMsg);  }  } |

#### LoginController

|  |
| --- |
| @Controller  **public** **class** LoginController **extends** BaseWebController {  /\*\*  \* 跳转到登陆页面页面  \*/  **private** **static** **final** String ***MB\_LOGIN\_FTL*** = "member/login";  /\*\*  \* 重定向到首页  \*/  **private** **static** **final** String ***REDIRECT\_INDEX*** = "redirect:/";  @Autowired  **private** MemberLoginServiceFeign memberLoginServiceFeign;  /\*\*  \* 跳转页面  \*  \* **@return**  \*/  @GetMapping("/login")  **public** String getLogin() {  **return** ***MB\_LOGIN\_FTL***;  }  /\*\*  \* 接受请求参数  \*  \* **@return**  \*/  @PostMapping("/login")  **public** String postLogin(@ModelAttribute("loginVo") LoginVo loginVo, Model model, HttpServletRequest request,  HttpServletResponse response, HttpSession httpSession) {  // 1.图形验证码判断  String graphicCode = loginVo.getGraphicCode();  **if** (!RandomValidateCodeUtil.*checkVerify*(graphicCode, httpSession)) {  setErrorMsg(model, "图形验证码不正确!");  **return** ***MB\_LOGIN\_FTL***;  }  // 2.将vo转换为dto  UserLoginInpDTO voToDto = MeiteBeanUtils.*voToDto*(loginVo, UserLoginInpDTO.**class**);  voToDto.setLoginType(Constants.***MEMBER\_LOGIN\_TYPE\_PC***);  String info = webBrowserInfo(request);  voToDto.setDeviceInfor(info);  BaseResponse<JSONObject> login = memberLoginServiceFeign.login(voToDto);  **if** (!isSuccess(login)) {  setErrorMsg(model, login.getMsg());  **return** ***MB\_LOGIN\_FTL***;  }  // 2.将token存入到cookie中  JSONObject data = login.getData();  String token = data.getString("token");  CookieUtils.*setCookie*(request, response, "meite\_token", token);  // 登陆成功后，跳转到首页  **return** ***REDIRECT\_INDEX***;  }  @RequestMapping("/exit")  **public** String exit(HttpServletRequest request) {  // 1. 从cookie中获取token  String token = CookieUtils.*getCookieValue*(request, "meite\_token");  **if** (!StringUtils.*isEmpty*(token)) {  BaseResponse<JSONObject> delToken = memberLoginServiceFeign.delToken(token);  **if** (isSuccess(delToken)) {  **return** ***REDIRECT\_INDEX***;  }  }  **return** ***ERROR\_500\_FTL***;  }  /\*\*  \* 获取浏览器信息  \*  \* **@return**  \*/  **public** String webBrowserInfo(HttpServletRequest request) {  // 获取浏览器信息  Browser browser = UserAgent.*parseUserAgentString*(request.getHeader("User-Agent")).getBrowser();  // 获取浏览器版本号  Version version = browser.getVersion(request.getHeader("User-Agent"));  String info = browser.getName() + "/" + version.getVersion();  **return** info;  }  } |

#### CookieUtils

|  |
| --- |
| **public** **final** **class** CookieUtils {  /\*\*  \* 得到Cookie的值, 不编码  \*  \* **@param** request  \* **@param** cookieName  \* **@return**  \*/  **public** **static** String getCookieValue(HttpServletRequest request, String cookieName) {  **return** *getCookieValue*(request, cookieName, **false**);  }  /\*\*  \* 得到Cookie的值,  \*  \* **@param** request  \* **@param** cookieName  \* **@return**  \*/  **public** **static** String getCookieValue(HttpServletRequest request, String cookieName, **boolean** isDecoder) {  Cookie[] cookieList = request.getCookies();  **if** (cookieList == **null** || cookieName == **null**) {  **return** **null**;  }  String retValue = **null**;  **try** {  **for** (**int** i = 0; i < cookieList.length; i++) {  **if** (cookieList[i].getName().equals(cookieName)) {  **if** (isDecoder) {  retValue = URLDecoder.*decode*(cookieList[i].getValue(), "UTF-8");  } **else** {  retValue = cookieList[i].getValue();  }  **break**;  }  }  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  **return** retValue;  }  /\*\*  \* 得到Cookie的值,  \*  \* **@param** request  \* **@param** cookieName  \* **@return**  \*/  **public** **static** String getCookieValue(HttpServletRequest request, String cookieName, String encodeString) {  Cookie[] cookieList = request.getCookies();  **if** (cookieList == **null** || cookieName == **null**) {  **return** **null**;  }  String retValue = **null**;  **try** {  **for** (**int** i = 0; i < cookieList.length; i++) {  **if** (cookieList[i].getName().equals(cookieName)) {  retValue = URLDecoder.*decode*(cookieList[i].getValue(), encodeString);  **break**;  }  }  } **catch** (UnsupportedEncodingException e) {  e.printStackTrace();  }  **return** retValue;  }  /\*\*  \* 设置Cookie的值 不设置生效时间默认浏览器关闭即失效,也不编码  \*/  **public** **static** **void** setCookie(HttpServletRequest request, HttpServletResponse response, String cookieName,  String cookieValue) {  *setCookie*(request, response, cookieName, cookieValue, -1);  }  /\*\*  \* 设置Cookie的值 在指定时间内生效,但不编码  \*/  **public** **static** **void** setCookie(HttpServletRequest request, HttpServletResponse response, String cookieName,  String cookieValue, **int** cookieMaxage) {  *setCookie*(request, response, cookieName, cookieValue, cookieMaxage, **false**);  }  /\*\*  \* 设置Cookie的值 不设置生效时间,但编码  \*/  **public** **static** **void** setCookie(HttpServletRequest request, HttpServletResponse response, String cookieName,  String cookieValue, **boolean** isEncode) {  *setCookie*(request, response, cookieName, cookieValue, -1, isEncode);  }  /\*\*  \* 设置Cookie的值 在指定时间内生效, 编码参数  \*/  **public** **static** **void** setCookie(HttpServletRequest request, HttpServletResponse response, String cookieName,  String cookieValue, **int** cookieMaxage, **boolean** isEncode) {  *doSetCookie*(request, response, cookieName, cookieValue, cookieMaxage, isEncode);  }  /\*\*  \* 设置Cookie的值 在指定时间内生效, 编码参数(指定编码)  \*/  **public** **static** **void** setCookie(HttpServletRequest request, HttpServletResponse response, String cookieName,  String cookieValue, **int** cookieMaxage, String encodeString) {  *doSetCookie*(request, response, cookieName, cookieValue, cookieMaxage, encodeString);  }  /\*\*  \* 删除Cookie带cookie域名  \*/  **public** **static** **void** deleteCookie(HttpServletRequest request, HttpServletResponse response, String cookieName) {  *doSetCookie*(request, response, cookieName, "", -1, **false**);  }  /\*\*  \* 设置Cookie的值，并使其在指定时间内生效  \*  \* **@param** cookieMaxage  \* cookie生效的最大秒数  \*/  **private** **static** **final** **void** doSetCookie(HttpServletRequest request, HttpServletResponse response, String cookieName,  String cookieValue, **int** cookieMaxage, **boolean** isEncode) {  **try** {  **if** (cookieValue == **null**) {  cookieValue = "";  } **else** **if** (isEncode) {  cookieValue = URLEncoder.*encode*(cookieValue, "utf-8");  }  Cookie cookie = **new** Cookie(cookieName, cookieValue);  **if** (cookieMaxage > 0)  cookie.setMaxAge(cookieMaxage);  **if** (**null** != request) {// 设置域名的cookie  String domainName = *getDomainName*(request);  System.***out***.println(domainName);  **if** (!"localhost".equals(domainName)) {  // cookie.setDomain(domainName);  }  }  cookie.setPath("/");  response.addCookie(cookie);  } **catch** (Exception e) {  e.printStackTrace();  }  }  /\*\*  \* 设置Cookie的值，并使其在指定时间内生效  \*  \* **@param** cookieMaxage  \* cookie生效的最大秒数  \*/  **private** **static** **final** **void** doSetCookie(HttpServletRequest request, HttpServletResponse response, String cookieName,  String cookieValue, **int** cookieMaxage, String encodeString) {  **try** {  **if** (cookieValue == **null**) {  cookieValue = "";  } **else** {  cookieValue = URLEncoder.*encode*(cookieValue, encodeString);  }  Cookie cookie = **new** Cookie(cookieName, cookieValue);  **if** (cookieMaxage > 0)  cookie.setMaxAge(cookieMaxage);  **if** (**null** != request) {// 设置域名的cookie  String domainName = *getDomainName*(request);  System.***out***.println(domainName);  **if** (!"localhost".equals(domainName)) {  // 本地测试的时候不要写.实际发布时在打开  // cookie.setDomain(domainName);  }  }  cookie.setPath("/");  response.addCookie(cookie);  } **catch** (Exception e) {  e.printStackTrace();  }  }  /\*\*  \* 得到cookie的域名  \*/  **private** **static** **final** String getDomainName(HttpServletRequest request) {  String domainName = **null**;  String serverName = request.getRequestURL().toString();  **if** (serverName == **null** || serverName.equals("")) {  domainName = "";  } **else** {  **final** **int** end = serverName.lastIndexOf("/");  serverName = serverName.substring(0, end);  **final** String[] domains = serverName.split("\\.");  **int** len = domains.length;  **if** (len > 3) {  // www.xxx.com.cn  domainName = "." + domains[len - 3] + "." + domains[len - 2] + "." + domains[len - 1];  } **else** **if** (len <= 3 && len > 1) {  // xxx.com or xxx.cn  domainName = "." + domains[len - 2] + "." + domains[len - 1];  } **else** {  domainName = serverName;  }  }  **if** (domainName != **null** && domainName.indexOf(":") > 0) {  String[] ary = domainName.split("\\:");  domainName = ary[0];  }  **return** domainName;  }  } |

图形验证码作用： 防止机器模拟实现接口调用

## 第三方联合登陆

### QQ联合登陆

腾讯开放平台网址: <https://connect.qq.com/index.html>

腾讯开放平台文档: http://wiki.connect.qq.com/

授权链接:

https://graph.qq.com/oauth2.0/authorize?response\_type=code&client\_id=101462456&redirect\_uri=http://shop.mayikt.com/qqLoginBack&state=1

&

将腾讯开放平台Sdk4J放入本地Maven私服仓库中

|  |
| --- |
| mvn install:install-file -Dfile=jar包的位置(参数一) -DgroupId=groupId(参数二) -DartifactId=artifactId(参数三) -Dversion=version(参数四) -Dpackaging=jar  mvn install:install-file -Dfile="E:\Sdk4J.jar" -DgroupId=com.tengxun -DartifactId=sdk4j -Dversion=1.0 -Dpackaging=jar |

#### 生成授权链接

|  |
| --- |
| @Controller  **public** **class** QQAuthoriController **extends** BaseWebController {  @Autowired  **private** QQAuthoriFeign qqAuthoriFeign;  **private** **static** **final** String ***MB\_QQ\_QQLOGIN*** = "member/qqlogin";  @Autowired  **private** MemberLoginServiceFeign memberLoginServiceFeign;  /\*\*  \* 重定向到首页  \*/  **private** **static** **final** String ***REDIRECT\_INDEX*** = "redirect:/";  /\*\*  \* 生成QQ授权回调地址  \*  \* **@return**  \*/  @RequestMapping("/qqAuth")  **public** String qqAuth(HttpServletRequest request) {  **try** {  String authorizeURL = **new** Oauth().getAuthorizeURL(request);  **return** "redirect:" + authorizeURL;  } **catch** (Exception e) {  **return** ***ERROR\_500\_FTL***;  }  } |

#### QQ授权回调

|  |
| --- |
| @RequestMapping("/qqLoginBack")  **public** String qqLoginBack(HttpServletRequest request, HttpServletResponse response, HttpSession httpSession) {  **try** {  AccessToken accessTokenObj = (**new** Oauth()).getAccessTokenByRequest(request);  **if** (accessTokenObj == **null**) {  **return** ***ERROR\_500\_FTL***;  }  String accessToken = accessTokenObj.getAccessToken();  **if** (StringUtils.*isEmpty*(accessToken)) {  **return** ***ERROR\_500\_FTL***;  }  // 获取用户openid  OpenID openIDObj = **new** OpenID(accessToken);  String openId = openIDObj.getUserOpenID();  **if** (StringUtils.*isEmpty*(openId)) {  **return** ***ERROR\_500\_FTL***;  }  BaseResponse<JSONObject> findByOpenId = qqAuthoriFeign.findByOpenId(openId);  **if** (!isSuccess(findByOpenId)) {  **return** ***ERROR\_500\_FTL***;  }  Integer resultCode = findByOpenId.getCode();  // 如果使用openid没有查询到用户信息，则跳转到绑定用户信息页面  **if** (resultCode.equals(Constants.***HTTP\_RES\_CODE\_NOTUSER\_203***)) {  // 使用openid获取用户信息  UserInfo qzoneUserInfo = **new** UserInfo(accessToken, openId);  UserInfoBean userInfoBean = qzoneUserInfo.getUserInfo();  **if** (userInfoBean == **null**) {  **return** ***ERROR\_500\_FTL***;  }  String avatarURL100 = userInfoBean.getAvatar().getAvatarURL100();  // 返回用户头像页面展示  request.setAttribute("avatarURL100", avatarURL100);  httpSession.setAttribute(WebConstants.***LOGIN\_QQ\_OPENID***, openId);  **return** ***MB\_QQ\_QQLOGIN***;  }  // 自动实现登陆  JSONObject data = findByOpenId.getData();  String token = data.getString("token");  CookieUtils.*setCookie*(request, response, WebConstants.***LOGIN\_TOKEN\_COOKIENAME***, token);  **return** ***REDIRECT\_INDEX***;  } **catch** (Exception e) {  **return** ***ERROR\_500\_FTL***;  }  } |

#### QQ绑定请求

|  |
| --- |
| @RequestMapping("/qqJointLogin")  **public** String qqJointLogin(@ModelAttribute("loginVo") LoginVo loginVo, Model model, HttpServletRequest request,  HttpServletResponse response, HttpSession httpSession) {  // 1.获取用户openid  String qqOpenId = (String) httpSession.getAttribute(WebConstants.***LOGIN\_QQ\_OPENID***);  **if** (StringUtils.*isEmpty*(qqOpenId)) {  **return** ***ERROR\_500\_FTL***;  }  // 2.将vo转换dto调用会员登陆接口  UserLoginInpDTO userLoginInpDTO = MeiteBeanUtils.*voToDto*(loginVo, UserLoginInpDTO.**class**);  userLoginInpDTO.setQqOpenId(qqOpenId);  userLoginInpDTO.setLoginType(Constants.***MEMBER\_LOGIN\_TYPE\_PC***);  String info = webBrowserInfo(request);  userLoginInpDTO.setDeviceInfor(info);  BaseResponse<JSONObject> login = memberLoginServiceFeign.login(userLoginInpDTO);  **if** (!isSuccess(login)) {  setErrorMsg(model, login.getMsg());  **return** ***MB\_QQ\_QQLOGIN***;  }  // 3.登陆成功之后如何处理 保持会话信息 将token存入到cookie 里面 首页读取cookietoken 查询用户信息返回到页面展示  JSONObject data = login.getData();  String token = data.getString("token");  CookieUtils.*setCookie*(request, response, WebConstants.***LOGIN\_TOKEN\_COOKIENAME***, token);  **return** ***REDIRECT\_INDEX***;  } |

#### 会员服务提供接口

根据 openid查询是否已经绑定,如果已经绑定，则直接实现自动登陆

|  |
| --- |
| **public** **interface** QQAuthoriService {  /\*\*  \* 根据 openid查询是否已经绑定,如果已经绑定，则直接实现自动登陆  \*  \* **@param** openid  \* **@return**  \*/  @RequestMapping("/findByOpenId")  BaseResponse<JSONObject> findByOpenId(@RequestParam("qqOpenId") String qqOpenId);  }  @RestController  **public** **class** QQAuthoriServiceImpl **extends** BaseApiService<JSONObject> **implements** QQAuthoriService {  @Autowired  **private** UserMapper userMapper;  @Autowired  **private** GenerateToken generateToken;  @Override  **public** BaseResponse<JSONObject> findByOpenId(String qqOpenId) {  **if** (StringUtils.*isEmpty*(qqOpenId)) {  **return** setResultError("qqOpenId不能为空!");  }  // 1.根据openid查询用户信息  UserDo userDo = userMapper.findByOpenId(qqOpenId);  **if** (userDo == **null**) {  **return** setResultError(Constants.***HTTP\_RES\_CODE\_NOTUSER\_203***, "根据qqOpenId没有查询到用户信息");  }  // 2.如果能够查询到用户信息,则直接生成对应的用户令牌  String keyPrefix = Constants.***MEMBER\_TOKEN\_KEYPREFIX*** + Constants.***HTTP\_RES\_CODE\_QQ\_LOGINTYPE***;  Long userId = userDo.getUserId();  String userToken = generateToken.createToken(keyPrefix, userId + "");  JSONObject data = **new** JSONObject();  data.put("token", userToken);  **return** setResultSuccess(data);  }  } |

#### 登陆接口改造

##### UserLoginInpDTO新增开放ID

|  |
| --- |
| /\*\*  \* 腾讯开放ID  \*/  @ApiModelProperty(value = "腾讯开放ID")  **private** String qqOpenId; |

|  |
| --- |
| // openid关联用户账号信息  String qqOpenId = userLoginInpDTO.getQqOpenId();  **if** (!StringUtils.*isEmpty*(qqOpenId)) {  userMapper.updateUserOpenId(qqOpenId, userId);  } |

##### UserLoginInpDTO新增开放ID

#### 数据库访问层

|  |
| --- |
| @Select("SELECT USER\_ID AS USERID ,MOBILE AS MOBILE,EMAIL AS EMAIL,PASSWORD AS PASSWORD, USER\_NAME AS USER\_NAME ,SEX AS SEX ,AGE AS AGE ,CREATE\_TIME AS CREATETIME,IS\_AVALIBLE AS ISAVALIBLE,PIC\_IMG AS PICIMG,QQ\_OPENID AS QQOPENID,WX\_OPENID AS WXOPENID"  + " FROM meite\_user WHERE qq\_openid=#{qqOpenId}")  UserDo findByOpenId(@Param("qqOpenId") String qqOpenId);  @Update("update meite\_user set QQ\_OPENID =#{0} WHERE USER\_ID=#{1}")  **int** updateUserOpenId(@Param("qqOpenId") String qqOpenId, @Param("userId") Long userId); |

#### 新增数据常量

|  |
| --- |
| String LOGIN\_QQ\_OPENID = "qq\_openid";  // 用户信息不存在  Integer HTTP\_RES\_CODE\_NOTUSER\_203 = 203; |

QQ联合登陆实现思路： 必须掌握 OAuth2.0 协议

1.生成QQ联合登陆授权链接

<https://graph.qq.com/oauth2.0/authorize?response_type=code&client_id=101462456&>

redirect\_uri=http://shop.mayikt.com/qqLoginBack

2.(腾讯)用户选择账号后，使用重定向方式跳转回调地址

<http://shop.mayikt.com/qqLoginBack> 传递一个code参数

1. code作用授权码

使用code参数获取accessToken，使用accessToken获取用户信息(openid、头像、QQ年龄之类)

1. 生成QQ授权链接
2. 获取授权码，使用授权码获取用户的openid
3. 使用openid可以查询到用户信息

Openid作用：开放id 开放userid

代码实现思路：

1. 编写授权链接接口
2. 编写授权回调接口

### 获取到授权码

### 使用授权码获取accessToken

### 使用accessToken获取用户openid

1. 使用openid查询数据库user信息表中是否有关联

SELECT \* FROM meite\_user where QQ\_OPENID='4130A96EDE4187C8FD6BB055CC542B40';

### 如果使用openid能够查询用户信息，说明用户已经绑定成功，自动实现登陆

### 如果使用openid没有查询到用户信息的话，说明用户没有绑定账信息，跳转到关联账号页面

#### 关联成功账号之后，将openid修改为对应的账号信息。

update meite\_user set QQ\_OPENID ='4130A96EDE4187C8FD6BB055CC542B41' where USER\_ID='111';

127.0.0.1域名下存入这个状态值

Shop.mayikt.com 域名下获取状态值

### 微信联合登陆

微信公众号 获取用户信息接口

https://mp.weixin.qq.com/wiki?t=resource/res\_main&id=mp1421140842

### 动静分离架构

为什么项目中需要采用动静分离架构？

动静分离架构与前后端分离架构区别

七牛云 官网:

https://www.qiniu.com/?hmsr=baidu&hmpl=SEM&hmcu=pinzhuan\_title&hmkw=&hmci=

1MBPS=0.125Mb=128Kb,Mbps=Mbit/s即兆比特每秒

## 分布式Session问题

|  |
| --- |
| <!--spring session 与redis应用基本环境配置,需要开启redis后才可以使用，不然启动Spring boot会报错 -->  <dependency>  <groupId>org.springframework.session</groupId>  <artifactId>spring-session-data-redis</artifactId>  </dependency>  <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-pool2</artifactId>  </dependency>  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  </dependency> |

|  |
| --- |
| redis:  hostname: 188.131.155.46  port: 6379  password: 123456 |

|  |
| --- |
| //这个类用配置redis服务器的连接  //maxInactiveIntervalInSeconds为SpringSession的过期时间（单位：秒）  @EnableRedisHttpSession(maxInactiveIntervalInSeconds = 1800)  **public** **class** SessionConfig {  // 冒号后的值为没有配置文件时，制动装载的默认值  @Value("${redis.hostname:localhost}")  String hostName;  @Value("${redis.port:6379}")  **int** port;  @Value("${redis.password:123456}")  String passWord;  @Bean  **public** JedisConnectionFactory connectionFactory() {  JedisConnectionFactory connection = **new** JedisConnectionFactory();  connection.~~setPort~~(port);  connection.~~setHostName~~(hostName);  connection.~~setPassword~~(passWord);  **return** connection;  }  } |

负载均衡相关配置

|  |
| --- |
| #user nobody;  worker\_processes 1;  #error\_log logs/error.log;  #error\_log logs/error.log notice;  #error\_log logs/error.log info;  #pid logs/nginx.pid;  events {  worker\_connections 1024;  }  http {  include mime.types;  default\_type application/octet-stream;  sendfile on;  keepalive\_timeout 65;    upstream backServer{  server 192.168.18.220:8080;  server 192.168.18.220:8081;  }  server {  listen 80;  server\_name shop.mayikt.com;  location / {  proxy\_pass http://backServer;  index index.html index.htm;  }  }  } |

使用Spring-session框架 或者是使用token替代session

动静分离与前后分离区别？

动静分离： 动态资源（jsp、ftl）与静态资源（js、img、css）分开部署

前后分离： 接口与视图（前端进行开发）分开独立 vue

微服务电商项目动静分离架构模式

公司静态资源如何处理的？nginx

早期蚂蚁课堂 2017 2018

在一个网站中，那么请求是比较占宽带资源。（静态资源） css、js、img

服务器带宽1兆带宽=128kb/s

512/kb 4s 动态资源占带宽很小.

带宽的价格是非常昂贵

静态资源服务器平台（对象文件存储） 比如七牛云、阿里云（oss）、腾讯云 内置CDN内容分发

为什么域名最终解析的IP地址都不一样 CDN作用 就近原则、

CDN内容分发 就是将静态资源服务器会部署全国各个服务器节点。

用户访问的时候，遵循就近原则

如何将静态资源存放在七牛云

testmeite.itmayiedu.com 关联七牛云CDN

testmeite.itmayiedu.com/index.js csdn

动静分离架构模式有什么缺点：

[www.mayikt.com](http://www.mayikt.com)

static.mayikt.com

跨域问题

如何解决跨域问题 nginx

[www.mayiti.com/static](http://www.mayiti.com/static) 转发到static.mayikt.com

[www.mayiti.com](http://www.mayiti.com) 转发到真实ip地址

## SSO单点登陆系统

### 为什么需要做单点登陆系统

大型互联网公司中，公司旗下可能会有多个子系统，每个登陆实现统一管理

多个账户信息统一管理 SSO单点登陆认证授权系统

月底 蚂蚁宝典 蚂蚁课堂 介入SSO

### 单点登陆系统实现思路



### 单点登陆系统框架

[**https://github.com/xuxueli/xxl-sso**](https://github.com/xuxueli/xxl-sso) **xxl-sso**

<http://www.xuxueli.com/xxl-sso/#/> xxl-sso官方文档

### 思考需要修改XXL-SSO框架的

1. 修改XXL-SSO认证服务授权页面样式
2. 修改XXL-SSO认证服务数据库账号密码

http://xxlssoclient1.com:8084/xxl-sso-web-sample-springboot/

<http://xxlssoclient2.com:8085/xxl-sso-web-sample-springboot/>

客户端访问<http://xxlssoclient1.com:8084>重定向<http://xxlssoserver.com:8080（SSO>认证授权系统）

登陆成功之后跳转原来地址：

<http://xxlssoclient1.com:8084/xxl-sso-web-sample-springboot/?xxl_sso_sessionid=1000_1db53a90152140c6a8421c83546f75ab>

疑问：为什么在认证授权中心登陆成功之后，会带一个xxl\_sso\_sessionid参数。

疑问：为什么ssoClient需要集成redis

作业题：认证授权中心如何实现集群化

注意：删除热部署jar

源码分析：

思考问题：

访问客户端的时候，如何自动重定向到认证授权中心的?

过滤器请求,如果没有获取到用户的会话信息的话，自动重定向跳转到认证授权中心

重点断点调试 核心依赖Jar中 XxlSsoWebFilter类

Client原理分析：

1. 先从Cookie中获取当前CookieId
2. 如果用户没有登陆的情况下，重定向到认证授权中心

http://xxlssoserver.com:8080/xxl-sso-server/login?redirect\_url=http://xxlssoclient1.com:8084/xxl-sso-web-sample-springboot/

为什么redirect\_url地址

在认证授权中心登陆成功之后，返回原来地址 （回调地址）

1. 认证授权登陆成功之后，在认证授权域名下存放对应的cookie信息

4. 认证授权系统回调到子系统中传递xxl\_sso\_sessionid

<http://xxlssoclient1.com:8084/xxl-sso-web-sample-springboot/?xxl_sso_sessionid=1000_4529db8531d54e23856e44002ace6cbb>

1. 回调到子系统的时候，进XxlSsoWebFilter拦截 获取xxl\_sso\_sessionid 信息
2. 子系统使用xxl\_sso\_sessionid从redis查询认证授权系统登陆的用户信息，将用户信息在子系统域名下存放对应的用户Cookie信息

这样的 可以保证 认证授权系统和子系统双方Cookie信息同步

思考 其他其他子系统如何实现面密登陆的呢？

第二SSOClient系统登陆的话

### 项目如何集成XXL-SSO项目

1. 认证授权中心xxl-sso-server
2. 修改登陆数据库验证
3. 修改页面静态资源
4. SSOClient端整合

#### 创建认证授权中心meite-shop-basics-xxl-sso-server

##### Maven依赖信息

|  |
| --- |
| <!-- freemarker -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-freemarker</artifactId>  </dependency>  <!-- sso core -->  <dependency>  <groupId>com.xuxueli</groupId>  <artifactId>xxl-sso-core</artifactId>  <version>1.1.1-SNAPSHOT</version>  </dependency>  <!-- springcloud feign组件 -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-openfeign</artifactId>  </dependency>  <dependency>  <groupId>com.mayikt</groupId>  <artifactId>meite-shop-service-api-member</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency>  <dependency>  <groupId>com.mayikt</groupId>  <artifactId>meite-shop-common-web-core</artifactId>  <version>0.0.1-SNAPSHOT</version>  </dependency> |

##### 改造登陆代码

|  |
| --- |
| userLoginInpDTO.setMobile(username);  userLoginInpDTO.setPassword(password);  userLoginInpDTO.setLoginType(Constants.***MEMBER\_LOGIN\_TYPE\_PC***);  String info = webBrowserInfo(request);  userLoginInpDTO.setDeviceInfor(info);  BaseResponse<UserOutDTO> ssoLogin = memberServiceFeign.ssoLogin(userLoginInpDTO);  **if** (!isSuccess(ssoLogin)) {  redirectAttributes.addAttribute("errorMsg", ssoLogin.getMsg());  redirectAttributes.addAttribute(Conf.***REDIRECT\_URL***, request.getParameter(Conf.***REDIRECT\_URL***));  **return** "redirect:/login";  }  UserOutDTO data = ssoLogin.getData();  **if** (data == **null**) {  redirectAttributes.addAttribute("errorMsg", "用户信息数据为空!");  redirectAttributes.addAttribute(Conf.***REDIRECT\_URL***, request.getParameter(Conf.***REDIRECT\_URL***));  **return** "redirect:/login";  } |

##### 登陆接口

|  |
| --- |
| @Override  **public** BaseResponse<UserOutDTO> ssoLogin(@RequestBody UserLoginInpDTO userLoginInpDTO) {  // 1.验证参数  String mobile = userLoginInpDTO.getMobile();  **if** (StringUtils.*isEmpty*(mobile)) {  **return** setResultError("手机号码不能为空!");  }  String password = userLoginInpDTO.getPassword();  **if** (StringUtils.*isEmpty*(password)) {  **return** setResultError("密码不能为空!");  }  // 判断登陆类型  String loginType = userLoginInpDTO.getLoginType();  **if** (StringUtils.*isEmpty*(loginType)) {  **return** setResultError("登陆类型不能为空!");  }  // 目的是限制范围  **if** (!(loginType.equals(Constants.***MEMBER\_LOGIN\_TYPE\_ANDROID***) || loginType.equals(Constants.***MEMBER\_LOGIN\_TYPE\_IOS***)  || loginType.equals(Constants.***MEMBER\_LOGIN\_TYPE\_PC***))) {  **return** setResultError("登陆类型出现错误!");  }  // 设备信息  String deviceInfor = userLoginInpDTO.getDeviceInfor();  **if** (StringUtils.*isEmpty*(deviceInfor)) {  **return** setResultError("设备信息不能为空!");  }  // 2.对登陆密码实现加密  String newPassWord = MD5Util.*MD5*(password);  // 3.使用手机号码+密码查询数据库 ，判断用户是否存在  UserDo userDo = userMapper.login(mobile, newPassWord);  **if** (userDo == **null**) {  **return** setResultError("用户名称或者密码错误!");  }  **return** setResultSuccess(MeiteBeanUtils.*doToDto*(userDo, UserOutDTO.**class**));  } |

|  |
| --- |
| /\*\*  \* SSO认证系统登陆接口  \*  \* **@param** userLoginInpDTO  \* **@return**  \*/  @PostMapping("/ssoLogin")  **public** BaseResponse<UserOutDTO> ssoLogin(@RequestBody UserLoginInpDTO userLoginInpDTO); |

### meite-shop-portal项目集成

#### application.yml

|  |
| --- |
| #### 整合freemarker  spring:  freemarker:  cache: **false**  charset: UTF-8  check-template-location: **true**  content-type: text/html  expose-request-attributes: **true**  expose-session-attributes: **true**  request-context-attribute: request  suffix: .ftl  template-loader-path:  - classpath:/templates  application:  name: app-mayikt-portal-pay  ###服务注册到eureka地址  eureka:  client:  service-url:  defaultZone: http://localhost:8100/eureka  server:  port: 8050  redis:  hostname: 127.0.0.1  port: 6379  password: 123456  xxl-sso:  excluded:  paths:  xxl:  sso:  server: http://xxlssoserver.com:8080/xxl-sso-server  logout:  path: /logout  redis:  address: redis://127.0.0.1:6379 |

#### XxlSsoConfig

|  |
| --- |
| @Configuration  **public** **class** XxlSsoConfig **implements** DisposableBean {  @Value("${xxl.sso.server}")  **private** String xxlSsoServer;  @Value("${xxl.sso.logout.path}")  **private** String xxlSsoLogoutPath;  @Value("${xxl-sso.excluded.paths}")  **private** String xxlSsoExcludedPaths;  @Value("${xxl.sso.redis.address}")  **private** String xxlSsoRedisAddress;  @Bean  **public** FilterRegistrationBean xxlSsoFilterRegistration() {  // xxl-sso, redis init  JedisUtil.*init*(xxlSsoRedisAddress);  // xxl-sso, filter init  FilterRegistrationBean registration = **new** FilterRegistrationBean();  registration.setName("XxlSsoWebFilter");  registration.setOrder(1);  registration.addUrlPatterns("/\*");  registration.setFilter(**new** XxlSsoWebFilter());  registration.addInitParameter(Conf.***SSO\_SERVER***, xxlSsoServer);  registration.addInitParameter(Conf.***SSO\_LOGOUT\_PATH***, xxlSsoLogoutPath);  registration.addInitParameter(Conf.***SSO\_EXCLUDED\_PATHS***, xxlSsoExcludedPaths);  **return** registration;  }  @Override  **public** **void** destroy() **throws** Exception {  // xxl-sso, redis close  JedisUtil.*close*();  }  } |

在线properties转yml方式

https://www.bejson.com/devtools/properties2yaml/

### host文件配置域名

127.0.0.1 pay.mayikt.com:8050 支付项目

127.0.0.1 [www.mayikt.com](http://www.mayikt.com) 门户网站

member.mayikt.com

order.mayikt.com

单点登陆系统设计思想都是一样

联合登陆都会遵循oatuh2.0协议

思考：我们在项目中，如何集成xxl-sso框架？

肯定是修改xxl-sso框架？应该修改那些地方？

1. 认证授权中心

需要介入微服务基础设施

修改登陆验证接口（调用会员数据库验证）

需要修改登陆界面

1. SSOClient集成

集成XxlSsoConfig使用XxlSsoWebFilter（注意排除请求）

portal-web 会员登陆、注册 商品搜索、下单

pay-web 单独 聚合支付 银联 支付宝 微信支付

portal-web pay-web 需要实现SSO单点登陆

对比分析法

### XXL-SSO退出源码分析

<http://www.xuxueli.com/xxl-sso/#/?id=%E9%85%8D%E7%BD%AE%E8%AF%B4%E6%98%8E-1>

### XXL-SSO认证授权中心如何实现高可用

使用Nginx实现xxl-sso-server认集群和故障转移

|  |
| --- |
| upstream backServer{  server 127.0.0.1:8080;  server 127.0.0.1:8081;  }  server {  listen 80;  server\_name xxlssoserver.com;  location / {  proxy\_pass http://backServer;  ###nginx与上游服务器(真实访问的服务器)超时时间 后端服务器连接的超时时间\_发起握手等候响应超时时间  proxy\_connect\_timeout 1s;  ###nginx发送给上游服务器(真实访问的服务器)超时时间  proxy\_send\_timeout 1s;  ### nginx接受上游服务器(真实访问的服务器)超时时间  proxy\_read\_timeout 1s;  index index.html index.htm;  }  } |

### XXL-SSOToken认证

如何保证SSO-Server高可用的问题

可以考虑SSO-Servet集群问题

思考：SSOClient实现注销，URL地址如何拼接

SSO基于Cookie 形式 、Token形式

SSO基于Token形式的实现 普通登陆一模一样的路程

基于令牌形式实现SSO

1. 调用认证授权中心接口登陆返回用户令牌信息
2. SSOClient项目需要实现单点登陆的话，在调用SSOClient接口的时候请求头中传递Token信息
3. sessionId 相当于Token

# 构建商品服务平台

## 数据库表结构分析

## 商品服务基础组件安装

使用Elasticsearch构建搜索服务

<https://hub.docker.com/> dockerHub

回顾基础知识：Elasticsearch为什么查询速度非常快？

Elasticsearch底层采用倒排索引对文档的内容进行关键词分词，每个关键词对应多个文档出现具体位置信息，后期查询的时候直接通过分词定位到相关的文档信息，默认情况下Elasticsearch对中文分词不是很友好，可以采用第三方Ik分词器或者自定义热词。

## docker安装ES & Kibana

|  |
| --- |
| 1. 下载ES镜像问题   docker pull elasticsearch   1. 运行ES   docker run -it --name elasticsearch -d -p 9200:9200 -p 9300:9300 -p 5601:5601 elasticsearch   1. 测试运行结果   http://192.168.212.245:9200/ |

注意：-p 5601:5601 是kibana的端口地址 (我这里kibana的container共用elasticsearch的网络，所以这样设置

Docker在安装镜像文件，报错的时候如何定位错误原因

为什么docker后台启动容器成功了，为什么访问不了？

安装Kibana

|  |
| --- |
| docker run -it -d -e ELASTICSEARCH\_URL=http://127.0.0.1:9200 --name kibana --network=container:elasticsearch kibana  测试运行结果  http://192.168.212.245:5601/app/kibana |

docker run -it --name elasticsearch -d -p 9200:9200 -p 9300:9300 -p 5601:5601 elasticsearch -v /usr/local/es/es.yml:/usr/share/elasticsearch/config/elasticsearch.yml

## 商品服务搜索如何实现

商品服务搜索采用ElasticSearch实现

需要考虑问题？如何实现ES与mysql数据库同步

## logstash mysql 同步到 elasticsearch

Logstash是一个开源数据收集引擎，具有实时管道功能。Logstash可以动态地将来自不同数据源的数据统一起来，并将数据标准化到你所选择的目的地

|  |
| --- |
| 1.上传logstash-6.4.3.tar.gz到服务中  2.tar –zxvf logstash-6.4.3.tar.gz  3.cd logstash-6.4.3  4. bin/logstash-plugin install logstash-input-jdbc  5. bin/logstash-plugin install logstash-output-elasticsearch |

## 相关配置文件说明

|  |
| --- |
| jdbc\_driver\_library: jdbc mysql 驱动的路径，在上一步中已经下载  jdbc\_driver\_class: 驱动类的名字，mysql 填 com.mysql.jdbc.Driver 就好了  jdbc\_connection\_string: mysql 地址  jdbc\_user: mysql 用户  jdbc\_password: mysql 密码  schedule: 执行 sql 时机，类似 crontab 的调度  statement: 要执行的 sql，以 “:” 开头是定义的变量，可以通过 parameters 来设置变量，这里的 sql\_last\_value 是内置的变量，表示上一次 sql 执行中 update\_time 的值，这里 update\_time 条件是 >= 因为时间有可能相等，没有等号可能会漏掉一些增量  use\_column\_value: 使用递增列的值  tracking\_column\_type: 递增字段的类型，numeric 表示数值类型, timestamp 表示时间戳类型  tracking\_column: 递增字段的名称，这里使用 update\_time 这一列，这列的类型是 timestamp  last\_run\_metadata\_path: 同步点文件，这个文件记录了上次的同步点，重启时会读取这个文件，这个文件可以手动修改 |

相关配置参考附件

Crontab <https://tool.lu/crontab/> 注意：Crontab表达式以分为单位

上传mysql jar /usr/local/sql/ mysql-connector-java-5.1.46.jar

./bin/logstash -f mysql.conf 启动

## 多文件方式同步ES数据

一个 logstash 实例可以借助 pipelines 机制同步多个表，只需要写多个配置文件就可以了，假设我们有两个表 table1 和 table2，对应两个配置文件 sync\_table1.cfg 和 sync\_table2.cfg

在 config/pipelines.yml 中配置

|  |
| --- |
| - pipeline.id: table1  path.config: "config/sync\_table1.cfg"  - pipeline.id: table2  path.config: "config/sync\_table2.cfg" |

./bin/logstash

## logstash-input-jdbc原理

使用 logstash-input-jdbc 插件读取 mysql 的数据，这个插件的工作原理比较简单，就是定时执行一个 sql，然后将 sql 执行的结果写入到流中，增量获取的方式没有通过 binlog 方式同步，而是用一个递增字段作为条件去查询，每次都记录当前查询的位置，由于递增的特性，只需要查询比当前大的记录即可获取这段时间内的全部增量，一般的递增字段有两种，AUTO\_INCREMENT 的主键 id 和 ON UPDATE CURRENT\_TIMESTAMP 的 update\_time 字段，id 字段只适用于那种只有插入没有更新的表，update\_time 更加通用一些，建议在 mysql 表设计的时候都增加一个 update\_time 字段

## 基于Docker方式实现Elasticsearch集群

注意：spring-boot-starter-data-elasticsearch必须为集群方式连接，否则情况下会报一下错误

None of the configured nodes are available

|  |
| --- |
| 1.mkdir -p es/config  2.cd es  3.mkdir data1  4.mkdir data2  5.mkdir data3  firewall-cmd --add-port=9300/tcp  firewall-cmd --add-port=9301/tcp   1. mkdir plugins1 2. mkdir plugins2 |

|  |
| --- |
| 在es/config分别放入es1.yml、es2.yml  Es1:  cluster.name: elasticsearch-cluster  node.name: es-node1  network.bind\_host: 0.0.0.0  network.publish\_host: 192.168.212.252  http.port: 9200  transport.tcp.port: 9300  http.cors.enabled: true  http.cors.allow-origin: "\*"  node.master: true  node.data: true  discovery.zen.ping.unicast.hosts: ["192.168.212.252:9300","192.168.212.252:9301"]  discovery.zen.minimum\_master\_nodes: 1  Es2:  cluster.name: elasticsearch-cluster  node.name: es-node2  network.bind\_host: 0.0.0.0  network.publish\_host: 192.168.212.252  http.port: 9201  transport.tcp.port: 9301  http.cors.enabled: true  http.cors.allow-origin: "\*"  node.master: true  node.data: true  discovery.zen.ping.unicast.hosts: ["192.168.212.252:9300","192.168.212.252:9301"]  discovery.zen.minimum\_master\_nodes: 1 |

启动容器1

|  |
| --- |
| docker run -e ES\_JAVA\_OPTS="-Xms256m -Xmx256m" -d -p 9200:9200 -p 9300:9300 -p 5601:5601 -v /usr/local/es/config/es1.yml:/usr/share/elasticsearch/config/elasticsearch.yml -v /usr/local/es/plugins1:/usr/share/elasticsearch/plugins -v /usr/local/es/data1:/usr/share/elasticsearch/data --name ES01 elasticsearch |

启动容器2

|  |
| --- |
| docker run -e ES\_JAVA\_OPTS="-Xms256m -Xmx256m" -d -p 9201:9201 -p 9301:9301 -v /usr/local/es/config/es2.yml:/usr/share/elasticsearch/config/elasticsearch.yml -v /usr/local/es/plugins2:/usr/share/elasticsearch/plugins -v /usr/local/es/data2:/usr/share/elasticsearch/data --name ES02 elasticsearch |

测试集群效果

<http://192.168.212.185:9200/_cat/nodes?pretty>

## SpringBoot整合Elasticsearch

1. 创建服务项目meite-shop-service-goods服务接口
2. 创建商品搜索服务接口

|  |
| --- |
| **public** **interface** ProductSearchService {  @GetMapping("/search")  **public** BaseResponse<List<ProductDto>> search(String name);  } |

3.dto实体类

|  |
| --- |
| @Data  **public** **class** ProductDto {  /\*\* 主键ID \*/  **private** Integer id;  /\*\* 类型ID \*/  **private** Integer categoryId;  /\*\* 名称 \*/  **private** String name;  /\*\* 小标题 \*/  **private** String subtitle;  /\*\* 主图像 \*/  **private** String mainImage;  /\*\* 小标题图像 \*/  **private** String subImages;  /\*\* 描述 \*/  **private** String detail;  /\*\* 商品规格 \*/  **private** String attributeList;  /\*\* 价格 \*/  **private** Double price;  /\*\* 库存 \*/  **private** Integer stock;  /\*\* 状态 \*/  **private** Integer status;  /\*\* 乐观锁 \*/  **private** Integer revision;  /\*\* 创建人 \*/  **private** String createdBy;  /\*\* 创建时间 \*/  **private** Date createdTime;  /\*\* 更新人 \*/  **private** String updatedBy;  /\*\* 更新时间 \*/  **private** Timestamp updatedTime;  } |

4.maven依赖

|  |
| --- |
| <!-- springboot 整合ES -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-elasticsearch</artifactId>  </dependency>  <dependency>  <groupId>com.querydsl</groupId>  <artifactId>querydsl-apt</artifactId>  </dependency>  <dependency>  <groupId>com.querydsl</groupId>  <artifactId>querydsl-jpa</artifactId>  </dependency>  <dependency>  <groupId>ma.glasnost.orika</groupId>  <artifactId>orika-core</artifactId>  <version>1.5.2</version>  </dependency> |

|  |
| --- |
| @RestController  **public** **class** ProductSearchServiceImpl **extends** BaseApiService<List<ProductDto>> **implements** ProductSearchService {  @Autowired  **private** ProductReposiory productReposiory;  @Override  **public** BaseResponse<List<ProductDto>> search(String name) {  BoolQueryBuilder builder = QueryBuilders.*boolQuery*();  // 模拟查询  builder.must(QueryBuilders.*fuzzyQuery*("name", name));  Pageable pageable = **new** QPageRequest(0, 5);  Page<ProductEntity> page = productReposiory.search(builder, pageable);  List<ProductEntity> content = page.getContent();  MapperFactory mapperFactory = **new** DefaultMapperFactory.Builder().build();  List<ProductDto> mapAsList = mapperFactory.getMapperFacade().mapAsList(content, ProductDto.**class**);  **return** setResultSuccess(mapAsList);  }  } |

|  |
| --- |
| **public** **interface** ProductReposiory **extends** ElasticsearchRepository<ProductEntity, Long> {  }  @Document(indexName = "goods", type = "goods")  @Data  **public** **class** ProductEntity {  /\*\* 主键ID \*/  **private** Integer id;  /\*\* 类型ID \*/  **private** Integer categoryId;  /\*\* 名称 \*/  **private** String name;  /\*\* 小标题 \*/  **private** String subtitle;  /\*\* 主图像 \*/  **private** String mainImage;  /\*\* 小标题图像 \*/  **private** String subImages;  /\*\* 描述 \*/  **private** String detail;  /\*\* 商品规格 \*/  **private** String attributeList;  /\*\* 价格 \*/  **private** Double price;  /\*\* 库存 \*/  **private** Integer stock;  /\*\* 状态 \*/  **private** Integer status;  /\*\* 创建人 \*/  **private** String createdBy;  /\*\* 创建时间 \*/  **private** Date createdTime;  /\*\* 更新时间 \*/  **private** Timestamp updatedTime;  } |

|  |
| --- |
| ###服务启动端口号  server:  port: 8500  ###服务名称(服务注册到eureka名称)  eureka:  client:  service-url:  defaultZone: http://localhost:8100/eureka    spring:  application:  name: app-mayikt-goods  redis:  host: 188.131.155.46  port: 6379  password: 123456  pool:  max-idle: 100  min-idle: 1  max-active: 1000  max-wait: -1  ###数据库相关连接  datasource:  username: root  password: root  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://127.0.0.1:3306/goods?useUnicode=true&characterEncoding=UTF-  data:  elasticsearch:  ####集群名称  cluster-name: elasticsearch-cluster  ####地址  cluster-nodes: 192.168.212.252:9300 |

## Elasticsearch集成IK分词器

注意：docker下载ES默认版本号码5.6.12，IK分词器对应的也应该是5.6.12，否则情况下会报错版本号码不一致。

## ES文档类型映射

|  |
| --- |
| GET /goods/\_mapping  DELETE /goods  PUT /goods  POST /goods/\_mapping/goods  {  "goods": {  "properties": {  "@timestamp": {  "type": "date"  },  "@version": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "attribute\_list": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "category\_id": {  "type": "long"  },  "created\_time": {  "type": "date"  },  "detail": {  "type": "text",  "analyzer":"ik\_smart",  "search\_analyzer":"ik\_smart"  },  "id": {  "type": "long"  },  "main\_image": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "name": {  "type": "text",  "analyzer":"ik\_smart",  "search\_analyzer":"ik\_smart"  },  "revision": {  "type": "long"  },  "status": {  "type": "long"  },  "sub\_images": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "subtitle": {  "type": "text",  "analyzer":"ik\_smart",  "search\_analyzer":"ik\_smart"  },  "updated\_time": {  "type": "date"  }  }  }  } |

<http://192.168.212.252:9200/_analyze>

|  |
| --- |
| {  "analyzer": "ik\_smart",  "text": "苹果"  }  {  "analyzer": "standard",  "text": "奥迪a4l"  } |

1. MQ与logstash实现ES与数据库同步区别

Logstash实现ES与数据库同步：

使用定时器方式 、实现简单

MQ实现ES与数据库同步：

实时性、复杂性更高、一致性强

如果没有整合IK分词，es无法支持模糊查询

## 拼音分词器

elasticsearch-analysis-pinyin 是 ElasticSearch的拼音插件，强大的功能支持拼音等的搜索。

在plugins目录上传pinyin插件，测试效果。

http://192.168.212.253:9200/\_analyze

|  |
| --- |
| {  "analyzer": "pinyin",  "text": "苹果"  } |

## ES如何支持拼音和中文分词

1. 自定义分词器 支持拼音和中文分词

|  |
| --- |
| DELETE /goods  PUT /goods  {  "settings": {  "analysis": {  "analyzer": {  "ik\_smart\_pinyin": {  "type": "custom",  "tokenizer": "ik\_smart",  "filter": ["my\_pinyin", "word\_delimiter"]  },  "ik\_max\_word\_pinyin": {  "type": "custom",  "tokenizer": "ik\_max\_word",  "filter": ["my\_pinyin", "word\_delimiter"]  }  },  "filter": {  "my\_pinyin": {  "type" : "pinyin",  "keep\_separate\_first\_letter" : true,  "keep\_full\_pinyin" : true,  "keep\_original" : true,  "limit\_first\_letter\_length" : 16,  "lowercase" : true,  "remove\_duplicated\_term" : true  }  }  }  }    } |

### 重新指定文档类型映射拼音分词类型

|  |
| --- |
| POST /goods/\_mapping/goods  {      "goods": {  "properties": {  "@timestamp": {  "type": "date"  },  "@version": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "attribute\_list": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "category\_id": {  "type": "long"  },  "created\_time": {  "type": "date"  },  "detail": {  "type": "text",  "analyzer":"ik\_smart\_pinyin",  "search\_analyzer":"ik\_smart\_pinyin"  },  "id": {  "type": "long"  },  "main\_image": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "name": {  "type": "text",  "analyzer":"ik\_smart\_pinyin",  "search\_analyzer":"ik\_smart\_pinyin"  },  "revision": {  "type": "long"  },  "status": {  "type": "long"  },  "sub\_images": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "subtitle": {  "type": "text",  "analyzer":"ik\_smart",  "search\_analyzer":"ik\_smart"  },  "updated\_time": {  "type": "date"  }  }  }  } |

# 分布式日志收集系统

## 为什么要搭建分布式日志收集系统

在传统项目中，如果在生产环境中，有多台不同的服务器集群，如果生产环境需要通过日志定位项目的Bug的话，需要在每台节点上使用传统的命令方式查询，这样效率非常底下。

通常，日志被分散在储存不同的设备上。如果你管理数十上百台服务器，你还在使用依次登录每台机器的传统方法查阅日志。这样是不是感觉很繁琐和效率低下。当务之急我们使用集中化的日志管理，例如：开源的syslog，将所有服务器上的日志收集汇总。

集中化管理日志后，日志的统计和检索又成为一件比较麻烦的事情，一般我们使用grep、awk和wc等Linux命令能实现检索和统计，但是对于要求更高的查询、排序和统计等要求和庞大的机器数量依然使用这样的方法难免有点力不从心。

命令方式:

tail -n 300 tomcat.log | grep '8050'

tail -100f myes.log

画图演示

## ELK+Kafka实现分布式日志收集原理

使用SpringAop进行日志收集，然后通过kafka将日志发送给logstash，logstash再将日志写入elasticsearch，这样elasticsearch就有了日志数据了，最后，则使用kibana将存放在elasticsearch中的日志数据显示出来，并且可以做实时的数据图表分析等等。

## ELK+Kafka实现分布式日志收集职责

logstash做日志对接，接受应用系统的log，然后将其写入到elasticsearch中，logstash可以支持N种log渠道，kafka渠道写进来的、和log目录对接的方式、也可以对reids中的log数据进行监控读取，等等。

elasticsearch存储日志数据，方便的扩展特效，可以存储足够多的日志数据。

kibana则是对存放在elasticsearch中的log数据进行：数据展现、报表展现，并且是实时的。

为什么在分布式情况下，需要实现分布式日志收集。

在生产环境下，服务节点采用集群的话，查询日志是否方便。

缺点：查询日志非常繁琐、效率非常低。

疑问：为什么需要elk+kafka实现分布式日志收集。

画图演示。

单纯使用ElK实现分布式日志收集缺点：

1. logstash太多了，扩展不好。
2. 读取IO文件，可能会产生日志丢失
3. 不是实时性

中间需要引入到Kafka

elk+kafka+搭建后台系统

面试题：

Logstash输入来源有那些？

本地文件、kafka、数据库、mongdb、redis

ELK elasticsearch （存放日志）logstash （采集） kinbana（展示数据） kafka

Logstash输入 输出 过滤器

AOP如何拦截日志

Logstash怎么订阅kafka呢

logstash需不需要集群

周四 elk+kafka讲完

周二课程目标先搭建 ELK

思考问题：

1.那些日志信息需要输入logstash （error级别）

2.AOP 异常通知 服务与服务之间如何区分日志索引文件（服务名称）

3.在分布式日志收集中，相同的服务集群的话是不需要区分日志索引文件。

4.目的为了 统一管理相同节点日志我信息。

5.相同的服务集群的话，是是不需要区分日志索引文件 搜索日志的时候，如何定位服务器节点信息呢？

## ELK+Kafka环境搭建

### Kafka环境搭建

ELK+Kafka环境搭建

ELK elasticsearch （存放日志）logstash （采集） kinbana（展示数据）

Kafka、Zookeeper（集群）

下载kafka和Zookeeper镜像文件

|  |
| --- |
| docker pull wurstmeister/kafka  docker pull wurstmeister/zookeeper |

运行Zookeeper环境

|  |
| --- |
| docker run -d --name zookeeper -p 2181:2181 -t wurstmeister/zookeeper |

运行Kafka环境

|  |
| --- |
| docker run --name kafka01 \  -p 9092:9092 \  -e KAFKA\_BROKER\_ID=0 \  -e KAFKA\_ZOOKEEPER\_CONNECT=192.168.212.252:2181 \  -e KAFKA\_ADVERTISED\_LISTENERS=PLAINTEXT://192.168.212.252:9092 \  -e KAFKA\_LISTENERS=PLAINTEXT://0.0.0.0:9092 \  -d wurstmeister/kafka |

kafka-manager有图形化UI

|  |
| --- |
| docker run -itd \  --restart=always \  --name=kafka-manager \  -p 9000:9000 \  -e ZK\_HOSTS="192.168.212.137:2181" \  sheepkiller/kafka-manager |

192.168.212.137:9000

进入Kafka容器类

docker exec -it kafka01 /bin/bash

创建my\_log topic

|  |
| --- |
| /opt/kafka/bin/kafka-topics.sh --create --zookeeper 192.168.212.252:2181 --replication-factor 1 --partitions 1 --topic my\_log |

查询创建的主题

|  |
| --- |
| /opt/kafka/bin/kafka-topics.sh --list --zookeeper 192.168.212.252:2181 |

### AOP异常通知管理日志

创建meite-shop-basics-elk-kafka项目

为什么采用AOP技术拦截日志信息

步骤流程：

1. 项目使用Aop技术拦截日志信息，将日志信息投递 my\_log 主题

meite-shop-basics-elk-kafka

1. Json的格式如何定义的呢？ 统一AOP日志输出格式、
2. Logstash 订阅my\_log主题 获取日志信息，输出到ES中。
3. 定时系统，调用ES查询Error异常以邮件方式通知给测试专员。

# 聚合支付平台设计

## 支付平台

在线支付平台分为：支付宝、平安支付、微信支付、小米支付、银联支付等。

聚合支付平台: ping++、mustPay 等。

聚合支付与第三方支付的区别

## 支付项目中常见难点

1. 支付回调接口中，产生延迟通知？如何解决支付状态不一致问题
2. 重试支付回调接口时，如何保证回调接口通知幂等性问题
3. 支付金额与商品金额如果不一致时，如何处理
4. 第三方支付平台如何与对接系统保证分布式事务问题
5. 支付服务如何与其他系统保证分布式事务问题

## 支付系统实现原理

同步回调：

异步回调：

## 银联支付接口

<https://open.unionpay.com/tjweb/acproduct/dictionary>

<https://open.unionpay.com/tjweb/acproduct/list?apiservId=448>

在线html格式化 http://tool.oschina.net/codeformat/html

实现步骤：

1. 下载对应的银联demo
2. 创建D:\certs目录
3. 下载对应测试证书，放入D:\certs

测试账号信息:

卡号：6216261000000000018

姓名：全渠道

身份证号码: 341126197709218366

## 外网映射工具

NatApp

natapp -authtoken=2fb5f28e255be404

## 支付系统表结构分析

查看附件SQL

## 构建支付系统

meite-shop-service-pay

### 创建支付令牌接口

|  |
| --- |
| **public** **interface** PayMentTransacService {  /\*\*  \* 创建支付令牌  \*  \* **@return**  \*/  @GetMapping("/cratePayToken")  **public** BaseResponse<JSONObject> cratePayToken(@Validated PayCratePayTokenDto payCratePayTokenDto);  }  @RestController  **public** **class** PayMentTransacTokenServiceImpl **extends** BaseApiService<JSONObject> **implements** PayMentTransacTokenService {  @Autowired  **private** PaymentTransactionMapper paymentTransactionMapper;  @Autowired  **private** GenerateToken generateToken;  @Override  **public** BaseResponse<JSONObject> cratePayToken(PayCratePayTokenDto payCratePayTokenDto) {  String orderId = payCratePayTokenDto.getOrderId();  **if** (StringUtils.*isEmpty*(orderId)) {  **return** setResultError("订单号码不能为空!");  }  Long payAmount = payCratePayTokenDto.getPayAmount();  **if** (payAmount == **null**) {  **return** setResultError("金额不能为空!");  }  Long userId = payCratePayTokenDto.getUserId();  **if** (userId == **null**) {  **return** setResultError("userId不能为空!");  }  // 2.将输入插入数据库中  PaymentTransactionEntity paymentTransactionEntity = **new** PaymentTransactionEntity();  paymentTransactionEntity.setOrderId(orderId);  paymentTransactionEntity.setPayAmount(payAmount);  paymentTransactionEntity.setUserId(userId);  // 使用雪花算法 生成全局id  paymentTransactionEntity.setPaymentId(SnowflakeIdUtils.*nextId*() + "");  **int** result = paymentTransactionMapper.insertPaymentTransaction(paymentTransactionEntity);  **if** (!toDaoResult(result)) {  **return** setResultError("系统错误!");  }  Long payId = paymentTransactionEntity.getId();  **if** (payId == **null**) {  **return** setResultError("系统错误!");  }  // 3.生成对应支付令牌  String keyPrefix = "pay\_";  String token = generateToken.createToken(keyPrefix, payId + "");  JSONObject dataResult = **new** JSONObject();  dataResult.put("token", token);  **return** setResultSuccess(dataResult);  }  } |

# 基于策略+工厂模式实现支付系统

## 策略模式概要

策略模式是对算法的包装，是把使用算法的责任和算法本身分割开来，委派给不同的对象管理。策略模式通常把一个系列的算法包装到一系列的策略类里面，作为一个抽象策略类的子类。

## 角色的划分

这个模式涉及到三个角色：

● 环境(Context)角色：持有一个Strategy的引用。

● 抽象策略(Strategy)角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。

● 具体策略(ConcreteStrategy)角色：包装了相关的算法或行为。

meite-shop-pay-plugin

---

## 项目构建

### PayStrategy

|  |
| --- |
| **public** **interface** PayStrategy {  /\*\*  \*  \* **@param** paymentChannel  \* 渠道<br>  \* **@param** payToken  \* 支付令牌<br>  \* **@return**  \*/  String toPayHtml(PaymentChannelEntity paymentChannel, PayMentTransacDTO payMentTransacDTO);  } |

### UnionPayStrategy

|  |
| --- |
| @Slf4j  @Component  **public** **class** UnionPayStrategy **implements** PayStrategy {  @Override  **public** String toPayHtml(PaymentChannelEntity paymentChannel, PayMentTransacDTO payMentTransacDTO) {  ***log***.info(">>>>>>>>银联支付组装参数开始<<<<<<<<<<<<");  Map<String, String> requestData = **new** HashMap<String, String>();  /\*\*\* 银联全渠道系统，产品参数，除了encoding自行选择外其他不需修改 \*\*\*/  requestData.put("version", UnionPayBase.*version*); // 版本号，全渠道默认值  requestData.put("encoding", UnionPayBase.*encoding*); // 字符集编码，可以使用UTF-8,GBK两种方式  requestData.put("signMethod", SDKConfig.*getConfig*().getSignMethod()); // 签名方法  requestData.put("txnType", "01"); // 交易类型 ，01：消费  requestData.put("txnSubType", "01"); // 交易子类型， 01：自助消费  requestData.put("bizType", "000201"); // 业务类型，B2C网关支付，手机wap支付  requestData.put("channelType", "07"); // 渠道类型，这个字段区分B2C网关支付和手机wap支付；07：PC,平板  // 08：手机  /\*\*\* 商户接入参数 \*\*\*/  String merchantId = paymentChannel.getMerchantId();  requestData.put("merId", merchantId); // 商户号码，请改成自己申请的正式商户号或者open上注册得来的777测试商户号  requestData.put("accessType", "0"); // 接入类型，0：直连商户  String paymentId = payMentTransacDTO.getPaymentId();  requestData.put("orderId", paymentId); // 商户订单号，8-40位数字字母，不能含“-”或“\_”，可以自行定制规则  requestData.put("txnTime", format(payMentTransacDTO.getCreatedTime())); // 订单发送时间，取系统时间，格式为YYYYMMDDhhmmss，必须取当前时间，否则会报txnTime无效  requestData.put("currencyCode", "156"); // 交易币种（境内商户一般是156 人民币）  Long payAmount = payMentTransacDTO.getPayAmount();  requestData.put("txnAmt", payAmount + ""); // 交易金额，单位分，不要带小数点  // requestData.put("reqReserved", "透传字段");  // //请求方保留域，如需使用请启用即可；透传字段（可以实现商户自定义参数的追踪）本交易的后台通知,对本交易的交易状态查询交易、对账文件中均会原样返回，商户可以按需上传，长度为1-1024个字节。出现&={}[]符号时可能导致查询接口应答报文解析失败，建议尽量只传字母数字并使用|分割，或者可以最外层做一次base64编码(base64编码之后出现的等号不会导致解析失败可以不用管)。  requestData.put("riskRateInfo", "{commodityName=测试商品名称}");  // 前台通知地址 （需设置为外网能访问 http https均可），支付成功后的页面 点击“返回商户”按钮的时候将异步通知报文post到该地址  // 如果想要实现过几秒中自动跳转回商户页面权限，需联系银联业务申请开通自动返回商户权限  // 异步通知参数详见open.unionpay.com帮助中心 下载 产品接口规范 网关支付产品接口规范 消费交易 商户通知\  String syncUrl = paymentChannel.getSyncUrl();  requestData.put("frontUrl", syncUrl);  // 后台通知地址（需设置为【外网】能访问 http  // https均可），支付成功后银联会自动将异步通知报文post到商户上送的该地址，失败的交易银联不会发送后台通知  // 后台通知参数详见open.unionpay.com帮助中心 下载 产品接口规范 网关支付产品接口规范 消费交易 商户通知  // 注意:1.需设置为外网能访问，否则收不到通知 2.http https均可 3.收单后台通知后需要10秒内返回http200或302状态码  // 4.如果银联通知服务器发送通知后10秒内未收到返回状态码或者应答码非http200，那么银联会间隔一段时间再次发送。总共发送5次，每次的间隔时间为0,1,2,4分钟。  // 5.后台通知地址如果上送了带有？的参数，例如：http://abc/web?a=b&c=d  // 在后台通知处理程序验证签名之前需要编写逻辑将这些字段去掉再验签，否则将会验签失败  String asynUrl = paymentChannel.getAsynUrl();  requestData.put("backUrl", asynUrl);  // 订单超时时间。  // 超过此时间后，除网银交易外，其他交易银联系统会拒绝受理，提示超时。  // 跳转银行网银交易如果超时后交易成功，会自动退款，大约5个工作日金额返还到持卡人账户。  // 此时间建议取支付时的北京时间加15分钟。  // 超过超时时间调查询接口应答origRespCode不是A6或者00的就可以判断为失败。  requestData.put("payTimeout",  **new** SimpleDateFormat("yyyyMMddHHmmss").format(**new** Date().getTime() + 15 \* 60 \* 1000));  //////////////////////////////////////////////////  //  // 报文中特殊用法请查看 PCwap网关跳转支付特殊用法.txt  //  //////////////////////////////////////////////////  /\*\* 请求参数设置完毕，以下对请求参数进行签名并生成html表单，将表单写入浏览器跳转打开银联页面 \*\*/  Map<String, String> submitFromData = AcpService.*sign*(requestData, UnionPayBase.*encoding*); // 报文中certId,signature的值是在signData方法中获取并自动赋值的，只要证书配置正确即可。  String requestFrontUrl = SDKConfig.*getConfig*().getFrontRequestUrl(); // 获取请求银联的前台地址：对应属性文件acp\_sdk.properties文件中的acpsdk.frontTransUrl  String html = AcpService.*createAutoFormHtml*(requestFrontUrl, submitFromData, UnionPayBase.*encoding*); // 生成自动跳转的Html表单  LogUtil.*writeLog*("打印请求HTML，此为请求报文，为联调排查问题的依据：" + html);  // 将生成的html写到浏览器中完成自动跳转打开银联支付页面；这里调用signData之后，将html写到浏览器跳转到银联页面之前均不能对html中的表单项的名称和值进行修改，如果修改会导致验签不通过  **return** html;  }  **private** String format(Date timeDate) {  String date = **new** java.text.SimpleDateFormat("yyyyMMddHHmmss").format(timeDate);  **return** date;  }  } |

### PayContextService

|  |
| --- |
| **public** **interface** PayContextService {  @GetMapping("/toPayHtml")  **public** BaseResponse<JSONObject> toPayHtml(String channelId, String payToken);  } |

### PayContextServiceImpl

|  |
| --- |
| @RestController  **public** **class** PayContextServiceImpl **extends** BaseApiService<JSONObject> **implements** PayContextService {  @Autowired  **private** PaymentChannelMapper paymentChannelMapper;  @Autowired  **private** PayMentTransacInfoService payMentTransacInfoService;  **public** BaseResponse<JSONObject> toPayHtml(String channelId, String payToken) {  // 1.使用渠道id查询渠道信息  PaymentChannelEntity pymentChannel = paymentChannelMapper.selectBychannelId(channelId);  **if** (pymentChannel == **null**) {  **return** setResultError("没有查询到该渠道信息");  }  // 2.使用payToken查询待支付信息  BaseResponse<PayMentTransacDTO> tokenByPayMentTransac = payMentTransacInfoService  .tokenByPayMentTransac(payToken);  **if** (!isSuccess(tokenByPayMentTransac)) {  **return** setResultSuccess(tokenByPayMentTransac.getMsg());  }  // 3.使用Java反射机制初始化子类  String classAddres = pymentChannel.getClassAddres();  PayStrategy payStrategy = StrategyFactory.*getPayStrategy*(classAddres);  **if** (payStrategy == **null**) {  **return** setResultError("支付系统网关错误!");  }  PayMentTransacDTO payMentTransacDTO = tokenByPayMentTransac.getData();  // 4.直接执行子类实现方法  String payHtml = payStrategy.toPayHtml(pymentChannel, payMentTransacDTO);  JSONObject data = **new** JSONObject();  data.put("payHtml", payHtml);  **return** setResultSuccess(data);  }  } |

### StrategyFactory

|  |
| --- |
| **public** **class** StrategyFactory {  **private** **static** Map<String, PayStrategy> *strategyBean* = **new** ConcurrentHashMap<String, PayStrategy>();  /\*\*  \* 使用Java反射机制初始化类  \*  \* **@param** classAddres  \* **@return**  \*/  **public** **static** PayStrategy getPayStrategy(String classAddres) {  **try** {  PayStrategy beanPayStrategy = *strategyBean*.get(classAddres);  **if** (beanPayStrategy != **null**) {  **return** beanPayStrategy;  }  Class<?> forName = Class.*forName*(classAddres);  PayStrategy payStrategy = (PayStrategy) forName.newInstance();  *strategyBean*.put(classAddres, payStrategy);  **return** payStrategy;  } **catch** (Exception e) {  **return** **null**;  }  }  } |

# 聚合支付平台对接支付系统

<https://docs.open.alipay.com/270/106291/> 支付宝电商文档

<https://openhome.alipay.com/platform/appDaily.htm> 支付宝沙箱环境

# 基于模版方法+异步回调实现

　抽象模板(Abstract Template)角色有如下责任：

定义了一个或多个抽象操作，以便让子类实现。这些抽象操作叫做基本操作，它们是一个顶级逻辑的组成步骤。

定义并实现了一个模板方法。这个模板方法一般是一个具体方法，它给出了一个顶级逻辑的骨架，而逻辑的组成步骤在相应的抽象操作中，推迟到子类实现。顶级逻辑也有可能调用一些具体方法。

具体模板(Concrete Template)角色又如下责任：

实现父类所定义的一个或多个抽象方法，它们是一个顶级逻辑的组成步骤。

每一个抽象模板角色都可以有任意多个具体模板角色与之对应，而每一个具体模板角色都可以给出这些抽象方法（也就是顶级逻辑的组成步骤）的不同实现，从而使得顶级逻辑的实现各不相同。

## 创建抽象模版

|  |
| --- |
| **public** **abstract** **class** AbstractPayCallbackTemplate {  /\*\*  \* 异步回调验证签名参数  \*  \* **@param** req  \* **@param** resp  \* **@return**  \*/  **public** **abstract** Map<String, String> verifySignature(HttpServletRequest req, HttpServletResponse resp);  /\*\*  \* 异步回调执行业务逻辑  \*  \* **@return**  \*/  **public** **abstract** String asyncService(Map<String, String> verifySignature);  /\*\*  \* 失败结果  \*  \* **@return**  \*/  **public** **abstract** String failResult();  /\*\*  \* 失败结果  \*  \* **@return**  \*/  **public** **abstract** String successResult();  **public** String asyncCallBack(HttpServletRequest req, HttpServletResponse resp) {  // 1.获取第三方支付回调参数信息  Map<String, String> verifySignature = verifySignature(req, resp);  **if** (verifySignature.get(PayConstant.***RESULT\_NAME***).equals(PayConstant.***RESULT\_PAYCODE\_201***)) {  **return** failResult();  }  // 2.将支付参数信息根据支付id插入到数据库中  payLog(verifySignature);  // 3.处理异步回调相关日志信息  **return** asyncService(verifySignature);  }  /\*\*  \* 将log写入到日志中  \*  \* **@param** paymentParameter  \*/  **private** **void** payLog(Map<String, String> paymentParameter) {  String paymentId = paymentParameter.get("paymentId");  ***log***.info(">>>>>paymentId:{},paymentParameter:{}", paymentId, paymentParameter);  }  } |

## 银联模版具体实现

|  |
| --- |
| @Component  **public** **class** UnionPayCallbackTemplate **extends** AbstractPayCallbackTemplate {  @Autowired  **private** PaymentTransactionMapper paymentTransactionMapper;  @Override  **public** Map<String, String> verifySignature(HttpServletRequest req, HttpServletResponse resp) {  LogUtil.*writeLog*("BackRcvResponse接收后台通知开始");  String encoding = req.getParameter(SDKConstants.***param\_encoding***);  // 获取银联通知服务器发送的后台通知参数  Map<String, String> reqParam = *getAllRequestParam*(req);  LogUtil.*printRequestLog*(reqParam);  // 重要！验证签名前不要修改reqParam中的键值对的内容，否则会验签不过  **if** (!AcpService.*validate*(reqParam, encoding)) {  LogUtil.*writeLog*("验证签名结果[失败].");  // 验签失败，需解决验签问题  reqParam.put(PayConstant.***RESULT\_NAME***, PayConstant.***RESULT\_PAYCODE\_201***);  } **else** {  LogUtil.*writeLog*("验证签名结果[成功].");  reqParam.put(PayConstant.***RESULT\_NAME***, PayConstant.***RESULT\_PAYCODE\_200***);  }  reqParam.put("paymentId", reqParam.get("orderId"));  **return** reqParam;  }  @Override  **public** String asyncService(Map<String, String> verifySignature) {  String orderId = verifySignature.get("orderId"); // 获取后台通知的数据，其他字段也可用类似方式获取  **if** (StringUtils.*isEmpty*(orderId)) {  **return** failResult();  }  // 1.根据orderId查询该支付信息  PaymentTransactionEntity paymentTransaction = paymentTransactionMapper.selectByPaymentId(orderId);  **if** (paymentTransaction == **null**) {  **return** failResult();  }  // 2.判断之前是否已经支付过,已经支付过则返回ok  Integer paymentStatus = paymentTransaction.getPaymentStatus();  **if** (paymentStatus.equals(PayConstant.***PAY\_STATUS\_SUCCESS***)) {  **return** successResult();  }  String respCode = verifySignature.get("respCode");  // 3.判断银联状态码不是00或者A6状态则状态码修改为已支付失败  **if** (!(respCode.equals("00") || respCode.equals("A6"))) {  **return** failResult();  }  // 4.如果银联状态码返回是00和A6的话  paymentTransactionMapper.updatePaymentStatus(1, orderId);  **return** successResult();  }  @Override  **public** String failResult() {  **return** PayConstant.***YINLIAN\_RESULT\_FAIL***;  }  @Override  **public** String successResult() {  // **TODO** Auto-generated method stub  **return** PayConstant.***PAY\_STATUS\_SUCCESS***;  }  /\*\*  \* 获取请求参数中所有的信息 当商户上送frontUrl或backUrl地址中带有参数信息的时候，  \* 这种方式会将url地址中的参数读到map中，会导多出来这些信息从而致验签失败，  \* 这个时候可以自行修改过滤掉url中的参数或者使用getAllRequestParamStream方法。  \*  \* **@param** request  \* **@return**  \*/  **public** **static** Map<String, String> getAllRequestParam(**final** HttpServletRequest request) {  Map<String, String> res = **new** HashMap<String, String>();  Enumeration<?> temp = request.getParameterNames();  **if** (**null** != temp) {  **while** (temp.hasMoreElements()) {  String en = (String) temp.nextElement();  String value = request.getParameter(en);  res.put(en, value);  // 在报文上送时，如果字段的值为空，则不上送<下面的处理为在获取所有参数数据时，判断若值为空，则删除这个字段>  **if** (res.get(en) == **null** || "".equals(res.get(en))) {  // System.out.println("======为空的字段名===="+en);  res.remove(en);  }  }  }  **return** res;  }  /\*\*  \* 获取请求参数中所有的信息。  \* 非struts可以改用此方法获取，好处是可以过滤掉request.getParameter方法过滤不掉的url中的参数。  \* struts可能对某些content-type会提前读取参数导致从inputstream读不到信息，所以可能用不了这个方法。  \* 理论应该可以调整struts配置使不影响，但请自己去研究。  \* 调用本方法之前不能调用req.getParameter("key");这种方法，否则会导致request取不到输入流。  \*  \* **@param** request  \* **@return**  \*/  **public** **static** Map<String, String> getAllRequestParamStream(**final** HttpServletRequest request) {  Map<String, String> res = **new** HashMap<String, String>();  **try** {  String notifyStr = **new** String(IOUtils.*toByteArray*(request.getInputStream()), UnionPayBase.*encoding*);  LogUtil.*writeLog*("收到通知报文：" + notifyStr);  String[] kvs = notifyStr.split("&");  **for** (String kv : kvs) {  String[] tmp = kv.split("=");  **if** (tmp.length >= 2) {  String key = tmp[0];  String value = URLDecoder.*decode*(tmp[1], UnionPayBase.*encoding*);  res.put(key, value);  }  }  } **catch** (UnsupportedEncodingException e) {  LogUtil.*writeLog*("getAllRequestParamStream.UnsupportedEncodingException error: " + e.getClass() + ":"  + e.getMessage());  } **catch** (IOException e) {  LogUtil.*writeLog*("getAllRequestParamStream.IOException error: " + e.getClass() + ":" + e.getMessage());  }  **return** res;  }  } |

## 模版工厂获取具体模版

|  |
| --- |
| **public** **class** TemplateFactory {  /\*\*  \* 获取具体模版方法类  \*  \* **@param** payCallbackName  \* bean实例名称  \* **@return**  \*/  **public** **static** AbstractPayCallbackTemplate getPayCallbackTemplate(String payCallbackName) {  **return** (AbstractPayCallbackTemplate) SpringContextUtil.*getBean*(payCallbackName);  }  } |

|  |
| --- |
| @RestController  **public** **class** PayAsynCallbackService {  /\*\*  \* 获取银联回调模版  \*/  **private** **static** **final** String ***UNIONPAYCALLBACKT\_EMPLATENAME*** = "unionPayCallbackTemplate";  /\*\*  \* 银联异步回调通知  \*  \* **@return**  \*/  @RequestMapping("/unionPayAsynCallback")  **public** String unionPayCallback(HttpServletRequest req, HttpServletResponse resp) {  AbstractPayCallbackTemplate payCallbackTemplate = TemplateFactory  .*getPayCallbackTemplate*(***UNIONPAYCALLBACKT\_EMPLATENAME***);  **return** payCallbackTemplate.asyncCallBack(req, resp);  }  } |

|  |
| --- |
| @Component  **public** **class** SpringContextUtil **implements** ApplicationContextAware {  /\*\*  \* 上下文对象实例  \*/  **private** **static** ApplicationContext *applicationContext*;  @Autowired  **public** **void** setApplicationContext(ApplicationContext applicationContext) **throws** BeansException {  **this**.*applicationContext* = applicationContext;  }  /\*\*  \* 获取applicationContext  \*  \* **@return**  \*/  **public** **static** ApplicationContext getApplicationContext() {  **return** *applicationContext*;  }  /\*\*  \* 通过name获取 Bean.  \*  \* **@param** name  \* **@return**  \*/  **public** **static** Object getBean(String name) {  **return** *getApplicationContext*().getBean(name);  }  /\*\*  \* 通过class获取Bean.  \*  \* **@param** clazz  \* **@param** <T>  \* **@return**  \*/  **public** **static** <T> T getBean(Class<T> clazz) {  **return** *getApplicationContext*().getBean(clazz);  }  /\*\*  \* 通过name,以及Clazz返回指定的Bean  \*  \* **@param** name  \* **@param** clazz  \* **@param** <T>  \* **@return**  \*/  **public** **static** <T> T getBean(String name, Class<T> clazz) {  **return** *getApplicationContext*().getBean(name, clazz);  }  } |

聚合支付平台优缺点：

优点：对接接口简单、后期维护成本低。

缺点：跑路或者资金链锻炼，安全系数低。

适合于中小型公司 月流水几万左右。

传统在线支付平台

回调设计模式：模版方法 外观设计模式

策略+ 模版+单例

对接的第三方支付接口整体流程都是相同的。

思考：

银联支付接口对接实现步骤：

1. 银联开放平台申请测试账号

<https://open.unionpay.com/tjweb/acproduct/list?apiservId=448>

1. 下载对应的商户的证书 放入到d:/certs/
2. 启动银联支付demo 的时候修改acp\_test\_sign.p12 证书名称

银联支付请求分析：/form\_6\_2\_FrontConsume 跳转银联支付接口。

银联支付以分为单位 支付宝接口元 为单位

表的 设计中如何存放金额呢? 用什么类型合适呢 long 存放分单位

1.24元等于多少分 24/100 0.24

前台通知:

后台通知：

验证签名作用：防止数据被篡改

银联支付提交原理：

1. 封装支付请求参数
2. 将参数验证签名，以html 表单形式提交给银联支付接口。

以html表单形式提交有什么好处、方便提交参数。

思考问题

在第三方支付平台（银联、支付宝、微信支付）支付结果是如何通知给商户端？

支付系统分布式事务问题存在原因：

用户在银联系统中，支付完成之后，如何把结果通知给商户端。

跨系统中如何解决分布式事务

采用最终一致性问题 双方可以短暂一致性，但是最终一定实现一致性问题，柔性事务

采用通知补偿性

同步回调（前台通知）：第三方支付系统以浏览器重定向形式将支付结果给商户端

作用：提示给用户在银联支付系统中，已经支付成功。

异步回调（后台通知）：第三方支付系统使用类似于HttpClient技术调用商户接口进行通知。

作用：将银联支付安全的形式发送给商户端，进行修改对应的订单状态。

注意：异步通知修改订单状态

本地开发中使用内网穿透技术 natapp

http://localhost:8080/ACPSample\_B2C/frontRcvResponse

验证签名原理：

MD5 加密加盐 不可逆

商户端 银联支付接口

10 MD5 ==aaabb 10 MD5=aaabb

验证签名防止数据被篡改

加密是防止抓包获取明文的数据 可逆。

互联网安全架构平台设计，具体底层如何实现。

适配器 模版方法 策略。

支付流程：

1. 订单服务调用支付服务 生成支付令牌（userID、订单号码、支付金额）
2. 订单服务获取支付令牌，在使用订单web传递Token形式跳转到支付Web。

预提交待支付记录 保证参数的安全问题。

设计模式：单例 工厂 代理 模版 外观、策略

策略+工厂+反射机制 实现解决if else 多重判断问题。

聚合支付平台对接多个不同的支付接口，但是支付接口大体实现思路都是相同的，唯一实现接口不同。

重构一下聚合支付平台提交参数 根据不同渠道（支付接口）返回不同的from表单提交参数。

分析设计模式

传统多个 if判断 维护信、不易扩展。

使用策略模式解决 多重if 判断问题。

在渠道表中CLASS\_ADDRES 字段 具体策略实现的子类 最终使用java反射机制执行。

大体实现思路：使用渠道id查询数据获取CLASS\_ADDRES，在使用java的反射机制执行toPayHtml。

策略模式

回调代码中，如何实现基于设计模式的重构？

支付回调（大体流程肯定是相同的），实现不同具体对接回调的代码不同。

相同点：使用异步通知，返回对应的结果。

模版方法设计模式（定义抽象类 （相同的代码）定义大体的骨架），不同的方法使用具体子类进行实现

相同点：

1. 将报文数据存放到es
2. 验证报文参数
3. 将日志根据支付id存放到数据库中
4. 执行的异步回调业务逻辑

# 基于XXL-Job实现异步对账

|  |
| --- |
| **public** **interface** PaymentCompensationService {  /\*\*  \* 根据payMentId查询支付信息  \*  \*  \* **@param** payMentId  \* **@return**  \*/  @GetMapping("/payMentCompensation")  **public** BaseResponse<JSONObject> payMentCompensation(String payMentId);  } |

|  |
| --- |
| @RestController  **public** **class** PaymentCompensationServiceImpl **extends** BaseApiService<JSONObject> **implements** PaymentCompensationService {  @Autowired  **private** PaymentTransactionMapper paymentTransactionMapper;  @Autowired  **private** PaymentChannelMapper paymentChannelMapper;  @Override  **public** BaseResponse<JSONObject> payMentCompensation(String payMentId) {  **if** (StringUtils.*isEmpty*(payMentId)) {  **return** setResultError("payMentId不能为空");  }  PaymentTransactionEntity paymentTransaction = paymentTransactionMapper.selectByPaymentId(payMentId);  **if** (paymentTransaction == **null**) {  **return** setResultError("paymentTransaction为空!");  }  // 2.获取所有的渠道重试id  List<PaymentChannelEntity> paymentChannelList = paymentChannelMapper.selectAll();  **for** (PaymentChannelEntity pcl : paymentChannelList) {  **if** (pcl != **null**) {  **return** compensationStrategy(paymentTransaction, pcl);  }  }  **return** setResultError("没有执行重试任务");  }  **private** BaseResponse<JSONObject> compensationStrategy(PaymentTransactionEntity paymentTransaction,  PaymentChannelEntity paymentChannelEntity) {  String retryBeanId = paymentChannelEntity.getRetryBeanId();  PaymentCompensationStrategy paymentCompensationStrategy = CompensationStrategyFactory  .*getPaymentCompensationStrategy*(retryBeanId);  // 3.实现子类重试  Boolean payMentCompensation = paymentCompensationStrategy.payMentCompensation(paymentTransaction,  paymentChannelEntity);  **return** payMentCompensation ? setResultSuccess("重试成功!") : setResultError("重试失败!");  }  } |

|  |
| --- |
| **public** **interface** PaymentCompensationStrategy {  /\*\*  \* 渠道名称  \*  \* **@param** paymentChannel  \* **@return**  \*/  **public** Boolean payMentCompensation(PaymentTransactionEntity paymentTransaction, PaymentChannelEntity paymentChanne);  } |

|  |
| --- |
| @Component  **public** **class** UnionPayCompensationStrategy **extends** BaseApiService<JSONObject> **implements** PaymentCompensationStrategy {  @Autowired  **private** PaymentTransactionMapper paymentTransactionMapper;  @Override  **public** Boolean payMentCompensation(PaymentTransactionEntity paymentTransaction,  PaymentChannelEntity paymentChanne) {  // 1.商户号码  String merchantId = paymentChanne.getMerchantId();  // 2.下单时间  Date createdTime = paymentTransaction.getCreatedTime();  // 3.支付id  String paymentId = paymentTransaction.getPaymentId();  // 4.调用银联支付接口  Boolean unionPayCompensation = unionPayCompensation(paymentId, format(createdTime), merchantId);  **return** unionPayCompensation;  }  **private** Boolean unionPayCompensation(String orderId, String txnTime, String merchantId) {  Map<String, String> data = **new** HashMap<String, String>();  /\*\*\* 银联全渠道系统，产品参数，除了encoding自行选择外其他不需修改 \*\*\*/  data.put("version", UnionPayBase.*version*); // 版本号  data.put("encoding", UnionPayBase.*encoding*); // 字符集编码 可以使用UTF-8,GBK两种方式  data.put("signMethod", SDKConfig.*getConfig*().getSignMethod()); // 签名方法  data.put("txnType", "00"); // 交易类型 00-默认  data.put("txnSubType", "00"); // 交易子类型 默认00  data.put("bizType", "000201"); // 业务类型 B2C网关支付，手机wap支付  /\*\*\* 商户接入参数 \*\*\*/  data.put("merId", merchantId); // 商户号码，请改成自己申请的商户号或者open上注册得来的777商户号测试  data.put("accessType", "0"); // 接入类型，商户接入固定填0，不需修改  /\*\*\* 要调通交易以下字段必须修改 \*\*\*/  data.put("orderId", orderId); // \*\*\*\*商户订单号，每次发交易测试需修改为被查询的交易的订单号  data.put("txnTime", txnTime); // \*\*\*\*订单发送时间，每次发交易测试需修改为被查询的交易的订单发送时间  /\*\* 请求参数设置完毕，以下对请求参数进行签名并发送http post请求，接收同步应答报文-------------> \*\*/  Map<String, String> reqData = AcpService.*sign*(data, UnionPayBase.*encoding*);// 报文中certId,signature的值是在signData方法中获取并自动赋值的，只要证书配置正确即可。  String url = SDKConfig.*getConfig*().getSingleQueryUrl();// 交易请求url从配置文件读取对应属性文件acp\_sdk.properties中的  // acpsdk.singleQueryUrl  // 这里调用signData之后，调用submitUrl之前不能对submitFromData中的键值对做任何修改，如果修改会导致验签不通过  Map<String, String> rspData = AcpService.*post*(reqData, url, UnionPayBase.*encoding*);  /\*\* 对应答码的处理，请根据您的业务逻辑来编写程序,以下应答码处理逻辑仅供参考-------------> \*\*/  // 应答码规范参考open.unionpay.com帮助中心 下载 产品接口规范 《平台接入接口规范-第5部分-附录》  **if** (!rspData.isEmpty()) {  **if** (AcpService.*validate*(rspData, UnionPayBase.*encoding*)) {  LogUtil.*writeLog*("验证签名成功");  **if** ("00".equals(rspData.get("respCode"))) {// 如果查询交易成功  // 处理被查询交易的应答码逻辑  String origRespCode = rspData.get("origRespCode");  **if** ("00".equals(origRespCode)) {  // 交易成功，更新商户订单状态  // 2.将状态改为已经支付成功  paymentTransactionMapper.updatePaymentStatus(PayConstant.***PAY\_STATUS\_SUCCESS*** + "", orderId,  "yinlian\_pay");  // 3.调用积分服务接口增加积分(处理幂等性问题)  **return** **true**;  } **else** **if** ("03".equals(origRespCode) || "04".equals(origRespCode) || "05".equals(origRespCode)) {  // 需再次发起交易状态查询交易  // **TODO**  } **else** {  // 其他应答码为失败请排查原因  // **TODO**  }  } **else** {// 查询交易本身失败，或者未查到原交易，检查查询交易报文要素  // **TODO**  }  } **else** {  LogUtil.*writeErrorLog*("验证签名失败");  // **TODO** 检查验证签名失败的原因  }  } **else** {  // 未返回正确的http状态  LogUtil.*writeErrorLog*("未获取到返回报文或返回http状态码非200");  }  **return** **false**;  }  **private** String format(Date timeDate) {  String date = **new** java.text.SimpleDateFormat("yyyyMMddHHmmss").format(timeDate);  **return** date;  }  } |

假设前提：第三方支付系统无法将支付结果通知商户端。

存在问题？如果需要对账的话？那么这笔第三方到底是那个渠道已经支付完成的呢、

# 异步回调整合多线程

## 多线程相关配置

|  |
| --- |
| ###多线程配置  threadPool:  ###核心线程数  corePoolSize: 10  ###最大线程数  maxPoolSize: 20  ## 队列容量  queueCapacity: 16 |

|  |
| --- |
| @Configuration  @EnableAsync  @Slf4j  **public** **class** AsyncTaskConfig **implements** AsyncConfigurer {  /\*\*  \* 最小线程数(核心线程数)  \*/  @Value("${threadPool.corePoolSize}")  **private** **int** corePoolSize;  /\*\*  \* 最大线程数  \*/  @Value("${threadPool.maxPoolSize}")  **private** **int** maxPoolSize;  /\*\*  \* 等待队列(队列最大长度)  \*/  @Value("${threadPool.queueCapacity}")  **private** **int** queueCapacity;  /\*\*  \* ThredPoolTaskExcutor的处理流程 当池子大小小于corePoolSize，就新建线程，并处理请求  \* 当池子大小等于corePoolSize，把请求放入workQueue中，池子里的空闲线程就去workQueue中取任务并处理  \* 当workQueue放不下任务时，就新建线程入池，并处理请求，如果池子大小撑到了maximumPoolSize，  \* 就用RejectedExecutionHandler来做拒绝处理  \* 当池子的线程数大于corePoolSize时，多余的线程会等待keepAliveTime长时间，如果无请求可处理就自行销毁  \*/  @Override  @Bean(name = "taskExecutor")  **public** Executor getAsyncExecutor() {  ThreadPoolTaskExecutor taskExecutor = **new** ThreadPoolTaskExecutor();  // 最小线程数(核心线程数)  taskExecutor.setCorePoolSize(corePoolSize);  // 最大线程数  taskExecutor.setMaxPoolSize(maxPoolSize);  // 等待队列(队列最大长度)  taskExecutor.setQueueCapacity(queueCapacity);  taskExecutor.initialize();  **return** taskExecutor;  }  /\*\*  \* 异步异常处理  \*  \* **@return**  \*/  @Override  **public** AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {  **return** **new** SpringAsyncExceptionHandler();  }  **class** SpringAsyncExceptionHandler **implements** AsyncUncaughtExceptionHandler {  @Override  **public** **void** handleUncaughtException(Throwable throwable, Method method, Object... obj) {  ***log***.error("Exception occurs in async method", throwable.getMessage());  }  }  } |

## 使用多线程异步回收日志

|  |
| --- |
| @Slf4j  @Component  **public** **abstract** **class** AbstractPayCallbackTemplate {  @Autowired  **private** ThreadPoolTaskExecutor threadPoolTaskExecutor;  @Autowired  **private** PaymentTransactionLogMapper paymentTransactionLogMapper;  /\*\*  \* 获取所有请求的参数，封装成Map集合 并且验证是否被篡改  \*  \* **@param** req  \* **@param** resp  \* **@return**  \*/  **public** **abstract** Map<String, String> verifySignature(HttpServletRequest req, HttpServletResponse resp);  /\*\*  \* 异步回调执行业务逻辑  \*  \* **@param** verifySignature  \*/  **public** **abstract** String asyncService(Map<String, String> verifySignature);  **public** **abstract** String failResult();  **public** **abstract** String successResult();  /\*\*  \* \*1. 将报文数据存放到es <br>  \* 1. 验证报文参数<br>  \* 2. 将日志根据支付id存放到数据库中<br>  \* 3. 执行的异步回调业务逻辑<br>  \*  \*/  **public** String asyncCallBack(HttpServletRequest req, HttpServletResponse resp) {  // 1. 验证报文参数 相同点 获取所有的请求参数封装成为map集合 并且进行参数验证  Map<String, String> verifySignature = verifySignature(req, resp);  // 2.将日志根据支付id存放到数据库中  String paymentId = verifySignature.get("paymentId");  **if** (StringUtils.*isEmpty*(paymentId)) {  **return** failResult();  }  ***log***.info(">>>service-1");  // 3.采用异步形式写入日志到数据库中  threadPoolTaskExecutor.execute(**new** LogThread(paymentId, verifySignature));  ***log***.info(">>>service-4");  String result = verifySignature.get(PayConstant.***RESULT\_NAME***);  // 4.201报文验证签名失败  **if** (result.equals(PayConstant.***RESULT\_PAYCODE\_201***)) {  **return** failResult();  }  // 5.执行的异步回调业务逻辑  **return** asyncService(verifySignature);  }  /\*\*  \*  \*  \* 采用多线程技术或者MQ形式进行存放到数据库中  \*  \* **@param** paymentId  \* **@param** verifySignature  \*/  **private** **void** payLog(String paymentId, Map<String, String> verifySignature) {  ***log***.info(">>paymentId:{paymentId},verifySignature:{}", verifySignature);  PaymentTransactionLogEntity paymentTransactionLogEntity = **new** PaymentTransactionLogEntity();  paymentTransactionLogEntity.setAsyncLog(verifySignature.toString());  paymentTransactionLogEntity.setTransactionId(paymentId);  paymentTransactionLogMapper.insertTransactionLog(paymentTransactionLogEntity);  }  **class** LogThread **implements** Runnable {  **private** String paymentId;  **private** Map<String, String> verifySignature;  **public** LogThread(String paymentId, Map<String, String> verifySignature) {  **this**.paymentId = paymentId;  **this**.verifySignature = verifySignature;  }  @Override  **public** **void** run() {  ***log***.info(">>>>异步线程开始...service-2");  payLog(paymentId, verifySignature);  ***log***.info(">>>>异步线程结束...service-3");  }  }  } |

## 配置线程池时线程数设置多少好

CPU密集型时，任务可以少配置线程数，大概和机器的cpu核数相当，这样可以使得每个线程都在执行任务

IO密集型时，大部分线程都阻塞，故需要多配置线程数，2\*cpu核数

# 构建分布式分布式任务调度平台

|  |
| --- |
| @JobHandler(value = "payCompensateHandler")  @Component  @Slf4j  **public** **class** PayCompensateHandler **extends** IJobHandler {  @Override  **public** ReturnT<String> execute(String param) **throws** Exception {  ***log***.info(">>>>>>>>>PayCompensateHandler");  **return** ***SUCCESS***;  }  // PAYMENT\_ID INTEGRAL  } |

1. 基于RabbitMQ实现分布式事务 1节课
2. 微服务秒杀抢购设计思想（redis、MQ） 四节课
3. 大型互联网安全平台架构(1节课)
4. 基于docker+Jekens实现自动化部署（1节课）

# 基于RabbitMQ实现分布式事务

## 支付服务(生产者)

### Maven相关依赖

|  |
| --- |
| <!-- 添加springboot对amqp的支持 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-amqp</artifactId>  </dependency> |

### 生产者代码

|  |
| --- |
| @Component  **public** **class** IntegralProducer **implements** RabbitTemplate.ConfirmCallback {  @Autowired  **private** RabbitTemplate rabbitTemplate;  @Transactional  **public** **void** send(JSONObject jsonObject) {  String jsonString = jsonObject.toJSONString();  System.***out***.println("jsonString:" + jsonString);  String paymentId = jsonObject.getString("paymentId");  // 封装消息  Message message = MessageBuilder.*withBody*(jsonString.getBytes())  .setContentType(MessageProperties.***CONTENT\_TYPE\_JSON***).setContentEncoding("utf-8").setMessageId(paymentId)  .build();  // 构建回调返回的数据（消息id）  **this**.rabbitTemplate.setMandatory(**true**);  **this**.rabbitTemplate.setConfirmCallback(**this**);  CorrelationData correlationData = **new** CorrelationData(jsonString);  rabbitTemplate.convertAndSend("integral\_exchange\_name", "integralRoutingKey", message, correlationData);  }  // 生产消息确认机制 生产者往服务器端发送消息的时候，采用应答机制  @Override  **public** **void** confirm(CorrelationData correlationData, **boolean** ack, String cause) {  String jsonString = correlationData.getId();  System.***out***.println("消息id:" + correlationData.getId());  **if** (ack) {  System.***out***.println("消息发送确认成功");  } **else** {  JSONObject jsonObject = JSONObject.*parseObject*(jsonString);  send(jsonObject);  System.***out***.println("消息发送确认失败:" + cause);  }  }  } |

|  |
| --- |
| **package** com.mayikt.pay.callback.template;  **import** java.util.Map;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** org.apache.commons.lang.StringUtils;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.scheduling.annotation.Async;  **import** org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;  **import** org.springframework.stereotype.Component;  **import** org.springframework.transaction.annotation.Transactional;  **import** com.alibaba.fastjson.JSONObject;  **import** com.mayikt.pay.constant.PayConstant;  **import** com.mayikt.pay.mapper.PaymentTransactionLogMapper;  **import** com.mayikt.pay.mapper.entity.PaymentTransactionEntity;  **import** com.mayikt.pay.mapper.entity.PaymentTransactionLogEntity;  **import** com.mayikt.pay.producer.IntegralProducer;  **import** com.sun.swing.internal.plaf.synth.resources.synth;  **import** lombok.Synchronized;  **import** lombok.extern.slf4j.Slf4j;  /\*\*  \*  \*  \*  \*  \* **@description**: 使用模版方法重构异步回调代码  \* **@author**: 97后互联网架构师-余胜军  \* **@contact**: QQ644064779、微信yushengjun644 www.mayikt.com  \* **@date**: 2019年1月3日 下午3:03:17  \* **@version** V1.0  \* **@Copyright** 该项目“基于SpringCloud2.x构建微服务电商项目”由每特教育|蚂蚁课堂版权所有，未经过允许的情况下，  \* 私自分享视频和源码属于违法行为。  \*/  @Slf4j  @Component  **public** **abstract** **class** AbstractPayCallbackTemplate {  @Autowired  **private** PaymentTransactionLogMapper paymentTransactionLogMapper;  @Autowired  **private** ThreadPoolTaskExecutor threadPoolTaskExecutor;  @Autowired  **private** IntegralProducer integralProducer;  /\*\*  \* 获取所有请求的参数，封装成Map集合 并且验证是否被篡改  \*  \* **@param** req  \* **@param** resp  \* **@return**  \*/  **public** **abstract** Map<String, String> verifySignature(HttpServletRequest req, HttpServletResponse resp);  /\*\*  \* 异步回调执行业务逻辑  \*  \* **@param** verifySignature  \*/  @Transactional  **public** **abstract** String asyncService(Map<String, String> verifySignature);  **public** **abstract** String failResult();  **public** **abstract** String successResult();  /\*\*  \* \*1. 将报文数据存放到es <br>  \* 1. 验证报文参数<br>  \* 2. 将日志根据支付id存放到数据库中<br>  \* 3. 执行的异步回调业务逻辑<br>  \*  \*/  @Transactional  **public** String asyncCallBack(HttpServletRequest req, HttpServletResponse resp) {  // 1. 验证报文参数 相同点 获取所有的请求参数封装成为map集合 并且进行参数验证  Map<String, String> verifySignature = verifySignature(req, resp);  // 2.将日志根据支付id存放到数据库中  String paymentId = verifySignature.get("paymentId");  **if** (StringUtils.*isEmpty*(paymentId)) {  **return** failResult();  }  ***log***.info(">>>>>asyncCallBack service 01");  // 3.采用异步形式写入日志到数据库中  threadPoolTaskExecutor.execute(**new** PayLogThread(paymentId, verifySignature));  ***log***.info(">>>>>asyncCallBack service 04");  String result = verifySignature.get(PayConstant.***RESULT\_NAME***);  // 4.201报文验证签名失败  **if** (result.equals(PayConstant.***RESULT\_PAYCODE\_201***)) {  **return** failResult();  }  // 5.执行的异步回调业务逻辑  **return** asyncService(verifySignature);  }  /\*\*  \*  \*  \* 采用多线程技术或者MQ形式进行存放到数据库中  \*  \* **@param** paymentId  \* **@param** verifySignature  \*/  **private** **void** payLog(String paymentId, Map<String, String> verifySignature) {  ***log***.info(">>paymentId:{paymentId},verifySignature:{}", verifySignature);  PaymentTransactionLogEntity paymentTransactionLog = **new** PaymentTransactionLogEntity();  paymentTransactionLog.setTransactionId(paymentId);  paymentTransactionLog.setAsyncLog(verifySignature.toString());  paymentTransactionLogMapper.insertTransactionLog(paymentTransactionLog);  }  // A 1423 B 1234  /\*\*  \* 使用多线程写入日志目的：加快响应 提高程序效率 使用线程池维护线程  \*/  **class** PayLogThread **implements** Runnable {  **private** String paymentId;  **private** Map<String, String> verifySignature;  **public** PayLogThread(String paymentId, Map<String, String> verifySignature) {  **this**.paymentId = paymentId;  **this**.verifySignature = verifySignature;  }  @Override  **public** **void** run() {  ***log***.info(">>>>>asyncCallBack service 02");  payLog(paymentId, verifySignature);  ***log***.info(">>>>>asyncCallBack service 03");  }  }  /\*\*  \* 增加积分  \*  \* **@param** paymentTransaction  \*/  @Async  **protected** **void** addIntegral(PaymentTransactionEntity paymentTransaction) {  JSONObject jsonObject = **new** JSONObject();  jsonObject.put("paymentId", paymentTransaction.getPaymentId());  jsonObject.put("userId", paymentTransaction.getUserId());  jsonObject.put("integral", 100);  jsonObject.put("paymentChannel", paymentTransaction.getPaymentChannel());  integralProducer.send(jsonObject);  }  } |

|  |
| --- |
| @Component  **public** **class** UnionPayCallbackTemplate **extends** AbstractPayCallbackTemplate {  @Autowired  **private** PaymentTransactionMapper paymentTransactionMapper;  @Autowired  **private** IntegralProducer integralProducer;  @Override  **public** Map<String, String> verifySignature(HttpServletRequest req, HttpServletResponse resp) {  LogUtil.*writeLog*("BackRcvResponse接收后台通知开始");  String encoding = req.getParameter(SDKConstants.***param\_encoding***);  // 获取银联通知服务器发送的后台通知参数  Map<String, String> reqParam = *getAllRequestParam*(req);  LogUtil.*printRequestLog*(reqParam);  // 重要！验证签名前不要修改reqParam中的键值对的内容，否则会验签不过  **if** (!AcpService.*validate*(reqParam, encoding)) {  LogUtil.*writeLog*("验证签名结果[失败].");  reqParam.put(PayConstant.***RESULT\_NAME***, PayConstant.***RESULT\_PAYCODE\_201***);  } **else** {  LogUtil.*writeLog*("验证签名结果[成功].");  // 【注：为了安全验签成功才应该写商户的成功处理逻辑】交易成功，更新商户订单状态  String orderId = reqParam.get("orderId"); // 获取后台通知的数据，其他字段也可用类似方式获取  reqParam.put("paymentId", orderId);  reqParam.put(PayConstant.***RESULT\_NAME***, PayConstant.***RESULT\_PAYCODE\_200***);  }  LogUtil.*writeLog*("BackRcvResponse接收后台通知结束");  **return** reqParam;  }  // 异步回调中网络尝试延迟，导致异步回调重复执行 可能存在幂等性问题  //  @Override  @Transactional  **public** String asyncService(Map<String, String> verifySignature) {  String orderId = verifySignature.get("orderId"); // 获取后台通知的数据，其他字段也可用类似方式获取  String respCode = verifySignature.get("respCode");  // 判断respCode=00、A6后，对涉及资金类的交易，请再发起查询接口查询，确定交易成功后更新数据库。  System.***out***.println("orderId:" + orderId + ",respCode:" + respCode);  // 1.判断respCode是否为已经支付成功断respCode=00、A6后，  **if** (!(respCode.equals("00") || respCode.equals("A6"))) {  **return** failResult();  }  // 根据日志 手动补偿 使用支付id调用第三方支付接口查询  PaymentTransactionEntity paymentTransaction = paymentTransactionMapper.selectByPaymentId(orderId);  **if** (paymentTransaction.getPaymentStatus().equals(PayConstant.***PAY\_STATUS\_SUCCESS***)) {  // 网络重试中，之前已经支付过  **return** successResult();  }  // 2.将状态改为已经支付成功  paymentTransactionMapper.updatePaymentStatus(PayConstant.***PAY\_STATUS\_SUCCESS*** + "", orderId, "yinlian\_pay");  // 3.调用积分服务接口增加积分(处理幂等性问题)  paymentTransaction.setPaymentChannel("yinlian\_pay");  addIntegral(paymentTransaction);  **return** successResult();  }  @Override  **public** String failResult() {  **return** PayConstant.***YINLIAN\_RESULT\_FAIL***;  }  @Override  **public** String successResult() {  **return** PayConstant.***YINLIAN\_RESULT\_SUCCESS***;  }  /\*\*  \* 获取请求参数中所有的信息 当商户上送frontUrl或backUrl地址中带有参数信息的时候，  \* 这种方式会将url地址中的参数读到map中，会导多出来这些信息从而致验签失败，  \* 这个时候可以自行修改过滤掉url中的参数或者使用getAllRequestParamStream方法。  \*  \* **@param** request  \* **@return**  \*/  **public** **static** Map<String, String> getAllRequestParam(**final** HttpServletRequest request) {  Map<String, String> res = **new** HashMap<String, String>();  Enumeration<?> temp = request.getParameterNames();  **if** (**null** != temp) {  **while** (temp.hasMoreElements()) {  String en = (String) temp.nextElement();  String value = request.getParameter(en);  res.put(en, value);  // 在报文上送时，如果字段的值为空，则不上送<下面的处理为在获取所有参数数据时，判断若值为空，则删除这个字段>  **if** (res.get(en) == **null** || "".equals(res.get(en))) {  // System.out.println("======为空的字段名===="+en);  res.remove(en);  }  }  }  **return** res;  }  /\*\*  \* 获取请求参数中所有的信息。  \* 非struts可以改用此方法获取，好处是可以过滤掉request.getParameter方法过滤不掉的url中的参数。  \* struts可能对某些content-type会提前读取参数导致从inputstream读不到信息，所以可能用不了这个方法。  \* 理论应该可以调整struts配置使不影响，但请自己去研究。  \* 调用本方法之前不能调用req.getParameter("key");这种方法，否则会导致request取不到输入流。  \*  \* **@param** request  \* **@return**  \*/  **public** **static** Map<String, String> getAllRequestParamStream(**final** HttpServletRequest request) {  Map<String, String> res = **new** HashMap<String, String>();  **try** {  String notifyStr = **new** String(IOUtils.*toByteArray*(request.getInputStream()), UnionPayBase.*encoding*);  LogUtil.*writeLog*("收到通知报文：" + notifyStr);  String[] kvs = notifyStr.split("&");  **for** (String kv : kvs) {  String[] tmp = kv.split("=");  **if** (tmp.length >= 2) {  String key = tmp[0];  String value = URLDecoder.*decode*(tmp[1], UnionPayBase.*encoding*);  res.put(key, value);  }  }  } **catch** (UnsupportedEncodingException e) {  LogUtil.*writeLog*("getAllRequestParamStream.UnsupportedEncodingException error: " + e.getClass() + ":"  + e.getMessage());  } **catch** (IOException e) {  LogUtil.*writeLog*("getAllRequestParamStream.IOException error: " + e.getClass() + ":" + e.getMessage());  }  **return** res;  }  /\*\*  \* 回调机制 必须遵循规范 重试机制都是采用间隔新 错开的话 必须  \*/  } |

### 配置文件

|  |
| --- |
| ###服务启动端口号  server:  port: 8600  ###服务名称(服务注册到eureka名称)  #spring:  # application:  # name: app-mayikt-member  ###服务注册到eureka地址  eureka:  client:  service-url:  defaultZone: http://localhost:8100/eureka    spring:  application:  name: app-mayikt-pay  zipkin:  base-url: http://127.0.0.1:9411/  ###全部采集  sleuth:  sampler:  probability: 1.0  redis:  host: 111.231.55.110  port: 6379  # password: 123456  pool:  max-idle: 100  min-idle: 1  max-active: 1000  max-wait: -1  ###数据库相关连接  datasource:  username: root  password: root  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://127.0.0.1:3306/meite\_pay?useUnicode=true&characterEncoding=UTF-8  rabbitmq:  ####连接地址  host: 127.0.0.1  ####端口号  port: 5672  ####账号  username: guest  ####密码  password: guest  ### 地址  virtual-host: /admin\_host  ###开启消息确认机制 confirms  publisher-confirms: **true**  publisher-returns: **true**  ####打印MyBatias日志  logging:  level:  ### 开发环境使用DEBUG 生产环境info或者error  com.mayikt.member.mapper: DEBUG    ###多线程配置  threadPool:  ###核心线程数  corePoolSize: 10  ###最大线程数  maxPoolSize: 20  ## 队列容量  queueCapacity: 16 |

### Rabbitmq配置

|  |
| --- |
| @Component  **public** **class** RabbitmqConfig {  // 添加积分队列  **public** **static** **final** String ***INTEGRAL\_DIC\_QUEUE*** = "integral\_queue";  // 补单队列，  **public** **static** **final** String ***INTEGRAL\_CREATE\_QUEUE*** = "integral\_create\_queue";  // 积分交换机  **private** **static** **final** String ***INTEGRAL\_EXCHANGE\_NAME*** = "integral\_exchange\_name";  // 1.定义订单队列  @Bean  **public** Queue directIntegralDicQueue() {  **return** **new** Queue(***INTEGRAL\_DIC\_QUEUE***);  }  // 2.定义补订单队列  @Bean  **public** Queue directCreateintegralQueue() {  **return** **new** Queue(***INTEGRAL\_CREATE\_QUEUE***);  }  // 2.定义交换机  @Bean  DirectExchange directintegralExchange() {  **return** **new** DirectExchange(***INTEGRAL\_EXCHANGE\_NAME***);  }  // 3.积分队列与交换机绑定  @Bean  Binding bindingExchangeintegralDicQueue() {  **return** BindingBuilder.*bind*(directIntegralDicQueue()).to(directintegralExchange()).with("integralRoutingKey");  }  // 3.补单队列与交换机绑定  @Bean  Binding bindingExchangeCreateintegral() {  **return** BindingBuilder.*bind*(directCreateintegralQueue()).to(directintegralExchange()).with("integralRoutingKey");  }  } |

## 积分服务(消费者)

|  |
| --- |
| @Component  @Slf4j  **public** **class** IntegralConsumer {  @Autowired  **private** IntegralMapper integralMapper;  @RabbitListener(queues = "integral\_queue")  **public** **void** process(Message message, @Headers Map<String, Object> headers, Channel channel) **throws** IOException {  **try** {  String messageId = message.getMessageProperties().getMessageId();  String msg = **new** String(message.getBody(), "UTF-8");  ***log***.info(">>>messageId:{},msg:{}", messageId, msg);  JSONObject jsonObject = JSONObject.*parseObject*(msg);  String paymentId = jsonObject.getString("paymentId");  **if** (StringUtils.*isEmpty*(paymentId)) {  ***log***.error(">>>>支付id不能为空 paymentId:{}", paymentId);  basicNack(message, channel);  **return**;  }  // 使用paymentId查询是否已经增加过积分  IntegralEntity resultIntegralEntity = integralMapper.findIntegral(paymentId);  **if** (resultIntegralEntity != **null**) {  ***log***.error(">>>>paymentId:{}已经增加过积分", paymentId);  basicNack(message, channel);  **return**;  }  Integer userId = jsonObject.getInteger("userId");  **if** (userId == **null**) {  ***log***.error(">>>>paymentId:{},对应的用户userId参数为空", paymentId);  basicNack(message, channel);  **return**;  }  Long integral = jsonObject.getLong("integral");  **if** (integral == **null**) {  ***log***.error(">>>>paymentId:{},对应的用户integral参数为空", integral);  **return**;  }  IntegralEntity integralEntity = **new** IntegralEntity();  integralEntity.setPaymentId(paymentId);  integralEntity.setIntegral(integral);  integralEntity.setUserId(userId);  integralEntity.setAvailability(1);  **int** insertIntegral = integralMapper.insertIntegral(integralEntity);  **if** (insertIntegral > 0) {  // 手动签收消息,通知mq服务器端删除该消息  basicNack(message, channel);  }  } **catch** (Exception e) {  ***log***.error(">>>>ERROR MSG:", e.getMessage());  basicNack(message, channel);  }  }  **private** **void** basicNack(Message message, Channel channel) **throws** IOException {  channel.basicNack(message.getMessageProperties().getDeliveryTag(), **false**, **false**);  }  } |

|  |
| --- |
| ###服务启动端口号  server:  port: 8800  ###服务名称(服务注册到eureka名称)  #spring:  # application:  # name: app-mayikt-member  ###服务注册到eureka地址  eureka:  client:  service-url:  defaultZone: http://localhost:8100/eureka    spring:  application:  name: app-mayikt-integral  zipkin:  base-url: http://127.0.0.1:9411/  ###全部采集  sleuth:  sampler:  probability: 1.0  redis:  host: 111.231.55.110  port: 6379  # password: 123456  pool:  max-idle: 100  min-idle: 1  max-active: 1000  max-wait: -1  ###数据库相关连接  datasource:  username: root  password: root  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://127.0.0.1:3306/meite\_integral?useUnicode=true&characterEncoding=UTF-8  rabbitmq:  ####连接地址  host: 127.0.0.1  ####端口号  port: 5672  ####账号  username: guest  ####密码  password: guest  ### 地址  virtual-host: /admin\_host  listener:  simple:  retry:  ####开启消费者（程序出现异常的情况下会）进行重试  enabled: **true**  ####最大重试次数  max-attempts: 5  ####重试间隔时间  initial-interval: 1000  ####开启手动ack  acknowledge-mode: manual  default-requeue-rejected: **false**  ####打印MyBatias日志  logging:  level:  ### 开发环境使用DEBUG 生产环境info或者error  com.mayikt.member.mapper: DEBUG    ###多线程配置  threadPool:  ###核心线程数  corePoolSize: 10  ###最大线程数  maxPoolSize: 20  ## 队列容量  queueCapacity: 16 |

1. 支付服务投递增加积分消息到 积分队列中， 积分服务监听积分队列，一旦有新的积分消息插入到积分表中。

基于MQ解决分布式事务原理：

1. 确保生产者（支付服务）一定将增加积分的消息投递到MQ中，使用确认机制.
2. 确保消费者消费消息一定成功 采用手动ack应答形式，如果没有出现异常情况下通知MQ删除该消息，否则情况下不断重试消费。

注意：重试过程中，需要注意消息幂等性问题。

1. 如果MQ异步代码执行之后，代码出现异常的情况下，如何保证分布式事务问题。

采用补偿队列

# 基于网关搭建微服务安全架构平台

## 网关相关配置

|  |
| --- |
| @Component  @Slf4j  **public** **class** GatewayFilter **extends** ZuulFilter {  @Autowired  **private** BlacklistMapper blacklistMapper;  @Override  **public** Object run() **throws** ZuulException {  RequestContext ctx = RequestContext.getCurrentContext();  HttpServletRequest request = ctx.getRequest();  HttpServletResponse response = ctx.getResponse();  response.setContentType("UTF-8");  // 1.获取ip地址  String ipAddres = getIpAddr(request);  **if** (StringUtils.isEmpty(ipAddres)) {  resultError(ctx, "未能够获取到ip地址");  }  // 2.查询数据库黑名单  MeiteBlacklist meiteBlacklist = blacklistMapper.findBlacklist(ipAddres);  **if** (meiteBlacklist != **null**) {  resultError(ctx, "ip:" + ipAddres + ",Insufficient access rights");  }  log.info(">>>>>>ip:{},验证通过>>>>>>>", ipAddres);  // 3.将ip地址传递到转发服务中  response.addHeader("ipAddres", ipAddres);  // 4.外网传递参数验证  Map<String, String> verifyMap = SignUtil.toVerifyMap(request.getParameterMap(), **false**);  **if** (!SignUtil.verify(verifyMap)) {  resultError(ctx, "ip:" + ipAddres + ",Sign fail");  }  **return** **null**;  }  **private** **void** resultError(RequestContext ctx, String errorMsg) {  ctx.setResponseStatusCode(401);  ctx.setSendZuulResponse(**false**);  ctx.setResponseBody(errorMsg);  }  @Override  **public** **boolean** shouldFilter() {  **return** **true**;  }  @Override  **public** **int** filterOrder() {  **return** 0;  }  /\*\*  \* 在方法之前拦截  \*  \* **@return**  \*/  @Override  **public** String filterType() {  **return** "pre";  }  /\*\*  \* 获取Ip地址  \*  \* **@param** request  \* **@return**  \*/  **public** String getIpAddr(HttpServletRequest request) {  String ip = request.getHeader("X-Forwarded-For");  **if** (ip == **null** || ip.length() == 0 || "unknown".equalsIgnoreCase(ip)) {  ip = request.getHeader("Proxy-Client-IP");  }  **if** (ip == **null** || ip.length() == 0 || "unknown".equalsIgnoreCase(ip)) {  ip = request.getHeader("WL-Proxy-Client-IP");  }  **if** (ip == **null** || ip.length() == 0 || "unknown".equalsIgnoreCase(ip)) {  ip = request.getHeader("HTTP\_CLIENT\_IP");  }  **if** (ip == **null** || ip.length() == 0 || "unknown".equalsIgnoreCase(ip)) {  ip = request.getHeader("HTTP\_X\_FORWARDED\_FOR");  }  **if** (ip == **null** || ip.length() == 0 || "unknown".equalsIgnoreCase(ip)) {  ip = request.getRemoteAddr();  }  **return** ip;  }  } |

## 基于建造者模式实现重构

建造者模式（Builder Pattern）使用多个简单的对象一步一步构建成一个复杂的对象。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

一个 Builder 类会一步一步构造最终的对象。该 Builder 类是独立于其他对象的

**将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表**

### 定义一个创建接口

|  |
| --- |
| **public** **interface** GatewayBuild {  /\*\*  \* 黑名单拦截  \*/  Boolean blackBlock(RequestContext ctx, String ipAddres, HttpServletResponse response);  /\*\*  \* 参数验证  \*/  Boolean toVerifyMap(RequestContext ctx, String ipAddres, HttpServletRequest request);  } |

### 参数创建者

|  |
| --- |
| @Slf4j  @Component  **public** **class** VerificationBuild **implements** GatewayBuild {  @Autowired  **private** BlacklistMapper blacklistMapper;  @Override  **public** Boolean blackBlock(RequestContext ctx, String ipAddres, HttpServletResponse response) {  // 2.查询数据库黑名单  MeiteBlacklist meiteBlacklist = blacklistMapper.findBlacklist(ipAddres);  **if** (meiteBlacklist != **null**) {  resultError(ctx, "ip:" + ipAddres + ",Insufficient access rights");  **return** **false**;  }  ***log***.info(">>>>>>ip:{},验证通过>>>>>>>", ipAddres);  // 3.将ip地址传递到转发服务中  response.addHeader("ipAddres", ipAddres);  ***log***.info(">>>>>>ip:{},验证通过>>>>>>>", ipAddres);  **return** **true**;  }  @Override  **public** Boolean toVerifyMap(RequestContext ctx, String ipAddres, HttpServletRequest request) {  // 4.外网传递参数验证  Map<String, String> verifyMap = SignUtil.*toVerifyMap*(request.getParameterMap(), **false**);  **if** (!SignUtil.*verify*(verifyMap)) {  resultError(ctx, "ip:" + ipAddres + ",Sign fail");  **return** **false**;  }  **return** **true**;  }  **private** **void** resultError(RequestContext ctx, String errorMsg) {  ctx.setResponseStatusCode(401);  ctx.setSendZuulResponse(**false**);  ctx.setResponseBody(errorMsg);  }  } |

### 定义建造者组装类

|  |
| --- |
| @Component  **public** **class** GatewayDirector {  @Resource(name = "verificationBuild")  **private** GatewayBuild gatewayBuild;  **public** **void** direcot(RequestContext ctx, String ipAddres, HttpServletResponse response, HttpServletRequest request) {  /\*\*  \* 黑名单拦截  \*/  Boolean blackBlock = gatewayBuild.blackBlock(ctx, ipAddres, response);  **if** (!blackBlock) {  **return**;  }  /\*\*  \* 参数验证  \*/  Boolean verifyMap = gatewayBuild.toVerifyMap(ctx, ipAddres, request);  **if** (!verifyMap) {  **return**;  }  }  } |

### 防止XSS攻击

|  |
| --- |
| /\*\*  \* 过滤参数  \*/  **private** Map<String, List<String>> filterParameters(HttpServletRequest request, RequestContext ctx) {  Map<String, List<String>> requestQueryParams = ctx.getRequestQueryParams();  **if** (requestQueryParams == **null**) {  requestQueryParams = **new** HashMap<>();  }  Enumeration em = request.getParameterNames();  **while** (em.hasMoreElements()) {  String name = (String) em.nextElement();  String value = request.getParameter(name);  ArrayList<String> arrayList = **new** ArrayList<>();  // 将参数转化为html参数 防止xss攻击  arrayList.add(StringEscapeUtils.*escapeHtml*(value));  requestQueryParams.put(name, arrayList);  }  **return** requestQueryParams;  } |

ctx.setRequestQueryParams(filterParameters(request, ctx));

## 基于令牌实现API接口调用

|  |
| --- |
| **public** **interface** AuthorizationService {  /\*\*  \* 机构申请 获取appid 和appsecret  \*  \* **@return**  \*/  @GetMapping("/applyAppInfo")  **public** BaseResponse<JSONObject> applyAppInfo(@RequestParam("appName") String appName);  /\*  \* 使用appid 和appsecret密钥获取AccessToken  \*/  @GetMapping("/getAccessToken")  **public** BaseResponse<JSONObject> getAccessToken(@RequestParam("appId") String appId,  @RequestParam("appSecret") String appSecret);  /\*  \* 验证Token是否失效  \*/  @GetMapping("/getAppInfo")  **public** BaseResponse<JSONObject> getAppInfo(@RequestParam("accessToken") String accessToken);  } |

|  |
| --- |
| @RestController  **public** **class** AuthorizationServiceImpl **extends** BaseApiService<JSONObject> **implements** AuthorizationService {  @Autowired  **private** AppInfoMapper appInfoMapper;  @Autowired  **private** GenerateToken generateToken;  @Override  **public** BaseResponse<JSONObject> applyAppInfo(String appName) {  **if** (StringUtils.*isEmpty*(appName)) {  **return** setResultError("机构名称不能为空!");  }  // 1.生成appid  Guid guid = **new** Guid();  String appId = guid.getAppId();  // 2.生成密钥  String appScrect = guid.getAppScrect();  // 3.插入到数据库中  MeiteAppInfo meiteAppInfo = **new** MeiteAppInfo(appName, appId, appScrect);  **int** insertAppInfo = appInfoMapper.insertAppInfo(meiteAppInfo);  **if** (!toDaoResult(insertAppInfo)) {  **return** setResultError("系统出现错误!");  }  JSONObject data = **new** JSONObject();  data.put("appId", appId);  data.put("appScrect", appScrect);  **return** setResultSuccess(data);  }  @Override  **public** BaseResponse<JSONObject> getAccessToken(String appId, String appSecret) {  **if** (StringUtils.*isEmpty*(appId)) {  **return** setResultError("appId不能为空!");  }  **if** (StringUtils.*isEmpty*(appSecret)) {  **return** setResultError("appSecret不能为空!");  }  // 1.使用appId+appSecret查询数据库  MeiteAppInfo meiteAppInfo = appInfoMapper.selectByAppInfo(appId, appSecret);  **if** (meiteAppInfo == **null**) {  **return** setResultError("appId或者是appSecret错误!");  }  // 2.生成accessToken  String dbAppId = meiteAppInfo.getAppId();  String accessToken = generateToken.createToken("auth", dbAppId);  JSONObject data = **new** JSONObject();  data.put("accessToken", accessToken);  **return** setResultSuccess(data);  }  @Override  **public** BaseResponse<JSONObject> getAppInfo(String accessToken) {  **if** (StringUtils.*isEmpty*(accessToken)) {  **return** setResultError("AccessToken cannot be empty ");  }  String appId = generateToken.getToken(accessToken);  **if** (StringUtils.*isEmpty*(appId)) {  **return** setResultError("accessToken invalid");  }  MeiteAppInfo meiteAppInfo = appInfoMapper.findByAppInfo(appId);  **if** (meiteAppInfo == **null**) {  **return** setResultError("AccessToken invalid");  }  JSONObject data = **new** JSONObject();  data.put("appInfo", meiteAppInfo);  **return** setResultSuccess(data);  }  } |

|  |
| --- |
| **public** **class** Guid {  **public** String appKey;  /\*\*  \* **@description**:随机获取key值  \* **@return**  \*/  **public** String guid() {  UUID uuid = UUID.*randomUUID*();  String key = uuid.toString();  **return** key;  }  /\*\*  \* 这是其中一个url的参数，是GUID的，全球唯一标志符  \*  \* **@param** product  \* **@return**  \*/  **public** String getAppId() {  Guid g = **new** Guid();  String guid = g.guid();  appKey = guid;  **return** appKey;  }  /\*\*  \* 根据md5加密  \*  \* **@param** product  \* **@return**  \*/  **public** String getAppScrect() {  String mw = "key" + appKey;  String app\_sign = MD5Util.*MD5*(mw).toUpperCase();// 得到以后还要用MD5加密。  **return** app\_sign;  }  } |

## 基于责任链模式实现网关权限控制

关键：将用户的请求分发给多个对象，这些对象组织成一个责任链，即每个对象都含有后继对象的引用，并要求责任链上的每个对象，若能处理用户请求，就做处理，不再将用户的请求传递给该链上的下一个对象；若不能处理请求，就必须将请求传递给链上的下一个对象。

包含两种角色：

        1、处理者(Handler)：接口，负责规定具体处理者处理用户请求的方法以及具体处理者，设置后继对象的方法。

        2、具体处理者(ConcreteHandler)：接口类的实例，通过调用处理者接口规定的方法处理用户请求，即在接到用户的请求后，处理者将调用接口规定的方法，在执行该方法的过程中若发现能处理用户的请求，就处理有关数据，否则就反馈无法处理的信息给用户，然后将用户的请求传递给自己的后继对象。

优点：

            1、责任链中的对象只和自己的后继是低耦合关系，和替他对象毫无关联；

            2、当在处理者中分配职责时，责任链给应用程序更多的灵活性；

            3、应用程序可以动态的add、delete处理者或重新指派处理者的职责；

            4、应用程序可以动态地改变处理者之间的先后顺序；

            5、使用责任链的用户不必知道处理者的信息，用户不会知道到底哪个对象处理了它的请求；

网关Handler

|  |
| --- |
| **public** **interface** GatewayHandler {  /\*\*  \* 网关拦截处理请求  \*/  Boolean service(RequestContext ctx, String ipAddres, HttpServletRequest request, HttpServletResponse response);  /\*\*  \* 设置下一个  \*/  **void** setNextHandler(GatewayHandler gatewayHandler);  /\*\*  \* 获取下一个Handler  \*  \* **@return**  \*/  **public** GatewayHandler getNextHandler();  } |

|  |
| --- |
| @Component  @Slf4j  **public** **class** BlacklistHandler **extends** BaseHandler **implements** GatewayHandler {  @Autowired  **private** BlacklistMapper blacklistMapper;  @Override  **public** Boolean service(RequestContext ctx, String ipAddres, HttpServletRequest request,  HttpServletResponse response) {  // >>>>>>>>>>>>>黑名单拦截操作<<<<<<<<<<<<<<<<<<<  ***log***.info(">>>>>>>>>拦截1 黑名单拦截 ipAddres:{}<<<<<<<<<<<<<<<<<<<<<<<<<<", ipAddres);  MeiteBlacklist meiteBlacklist = blacklistMapper.findBlacklist(ipAddres);  **if** (meiteBlacklist != **null**) {  resultError(ctx, "ip:" + ipAddres + ",Insufficient access rights");  **return** Boolean.***FALSE***;  }  // 传递给下一个  gatewayHandler.service(ctx, ipAddres, request, response);  **return** Boolean.***TRUE***;  }  } |

BaseHandler

|  |
| --- |
| **public** **class** BaseHandler {  **protected** GatewayHandler gatewayHandler;  **public** **void** setNextHandler(GatewayHandler gatewayHandler) {  **this**.gatewayHandler = gatewayHandler;  }  **public** GatewayHandler getNextHandler() {  **return** gatewayHandler;  }  **protected** **void** resultError(RequestContext ctx, String errorMsg) {  ctx.setResponseStatusCode(401);  // 网关响应为false 不会转发服务  ctx.setSendZuulResponse(**false**);  ctx.setResponseBody(errorMsg);  }  // 接口直接返回true 或者false  **public** Boolean isSuccess(BaseResponse<?> baseResp) {  **if** (baseResp == **null**) {  **return** **false**;  }  **if** (!baseResp.getCode().equals(Constants.***HTTP\_RES\_CODE\_200***)) {  **return** **false**;  }  **return** **true**;  }  } |

### 工厂存放责任链类

|  |
| --- |
| **public** **class** FactoryHandler {  **public** **static** GatewayHandler getHandler() {  // 1.黑名单拦截  GatewayHandler handler1 = (GatewayHandler) SpringContextUtil.*getBean*("blacklistHandler");  // 2.验证accessToken  GatewayHandler handler2 = (GatewayHandler) SpringContextUtil.*getBean*("apiAuthorityHandler");  handler1.setNextHandler(handler2);  // 3.API接口参数接口验签  GatewayHandler handler3 = (GatewayHandler) SpringContextUtil.*getBean*("toVerifyMapHandler");  handler2.setNextHandler(handler3);  **return** handler1;  }  } |

### 责任链执行

|  |
| --- |
| @Component  **public** **class** ResponsibilityClient {  **public** **void** responsibility(RequestContext ctx, String ipAddres, HttpServletRequest request,  HttpServletResponse response) {  GatewayHandler handler = FactoryHandler.*getHandler*();  handler.service(ctx, ipAddres, request, response);  }  } |

**问题：如果有其他合作伙伴（其他公司）需要调用我们公司接口，如何保证安全性问题。  
注意：接口实现限制 、黑名单与白名单、接口权限开关、合作伙伴管理等。**

**搭建一套oath2.0开放授权平台 银联、支付宝、微信支付、微信公众号**

**申请appid、app\_secret**

**思考问题：**

**第一个点appid（商户号 永久不会发生变化的）与app\_secret（密钥 可以重置）区别是什么？**

**第二点为什么获取accesToken的时候需要appid+app\_secret才能获取**

**防止黑客知道你的appid+app\_secret模拟请求，如果一旦泄漏的话，直接重试app\_secret密钥。**

**QQ账号密码 QQ账号是不变的，QQ密码可以改变的。**

1. **提供申请接口，获取appid和app\_secret**
2. **提供一个使用appid+appid获取accesToken**
3. **提供验证accesToken接口**

## 秒杀系统设计

**Java实现微服务秒杀抢购课程安排**

1. 秒杀抢购前端优化方案
2. 秒杀抢购如何防止超卖问题
3. 基于MQ和Redis实现秒杀抢购
4. 秒杀抢购如何防止伪造

### 使用Nginx实现页面缓存

|  |
| --- |
| events {  #的最大连接数（包含所有连接数）1024  worker\_connections 1024; ## Default: 1024  }  http{  # 代理缓存配置  proxy\_cache\_path "./meite\_cachedata" levels=1:2 keys\_zone=meitecache:256m inactive=1d max\_size=1000g;  server {  listen 80;  location /{  #使用缓存名称  proxy\_cache meitecache;  #对以下状态码实现缓存  proxy\_cache\_valid 200 206 304 301 302 1d;  #缓存的key  proxy\_cache\_key $request\_uri;  add\_header X-Cache-Status $upstream\_cache\_status;  #反向代理地址  proxy\_pass http://127.0.0.1:8080;  }    }  } |

### 使用数据库乐观锁实现防止超卖问题

#### 数据库表接口

|  |
| --- |
| CREATE TABLE `meite\_order` (  `seckill\_id` bigint(20) NOT NULL COMMENT '秒杀商品id',  `user\_phone` bigint(20) NOT NULL COMMENT '用户手机号',  `state` tinyint(4) NOT NULL DEFAULT '-1' COMMENT '状态标示:-1:无效 0:成功 1:已付款 2:已发货',  `create\_time` datetime NOT NULL COMMENT '创建时间',  KEY `idx\_create\_time` (`create\_time`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='秒杀成功明细表';  CREATE TABLE `meite\_seckill` (  `seckill\_id` bigint(20) NOT NULL COMMENT '商品库存id',  `name` varchar(120) CHARACTER SET utf8 NOT NULL COMMENT '商品名称',  `inventory` int(11) NOT NULL COMMENT '库存数量',  `start\_time` datetime NOT NULL COMMENT '秒杀开启时间',  `end\_time` datetime NOT NULL COMMENT '秒杀结束时间',  `create\_time` datetime NOT NULL COMMENT '创建时间',  `version` bigint(20) NOT NULL DEFAULT '0',  PRIMARY KEY (`seckill\_id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COMMENT='秒杀库存表'; |

#### 定义秒杀服务接口

|  |
| --- |
| /\*\*  \* 用户秒杀接口  \*  \* **@phone** 手机号码<br>  \* **@seckillId** 库存id  \* **@return**  \*/  @RequestMapping("/spike")  **public** BaseResponse<JSONObject> spike(String phone, Long seckillId);  @RestController  @Slf4j  **public** **class** SpikeCommodityServiceImpl **extends** BaseApiService<JSONObject> **implements** SpikeCommodityService {  @Autowired  **private** SeckillMapper seckillMapper;  @Autowired  **private** OrderMapper orderMapper;  @Autowired  **private** RedisUtil redisUtil;  @Override  @Transactional  **public** BaseResponse<JSONObject> spike(String phone, Long seckillId) {  // 1.验证参数  **if** (StringUtils.*isEmpty*(phone)) {  **return** setResultError("手机号码不能为空!");  }  **if** (seckillId == **null**) {  **return** setResultError("库存id不能为空!");  }  // >>>限制用户访问频率 比如10秒中只能访问一次  Boolean resultNx = redisUtil.setNx(phone, seckillId + "", 10l);  **if** (!resultNx) {  **return** setResultError("该用户操作过于频繁,请稍后重试!");  }  // 2.根据库存id查询商品是否存在  SeckillEntity seckillEntity = seckillMapper.findBySeckillId(seckillId);  **if** (seckillEntity == **null**) {  **return** setResultError("该商品信息不存在!");  }  // 3.对库存的数量实现减去1  Long version = seckillEntity.getVersion();  **int** inventoryDeduction = seckillMapper.optimisticVersionSeckill(seckillId, version);  **if** (!toDaoResult(inventoryDeduction)) {  ***log***.info(">>>>phone:{},seckillId:{},秒杀失败!", phone, seckillId);  **return** setResultError("系统错误!");  }  ***log***.info(">>>>phone:{},seckillId:{},扣库存成功!", phone, seckillId);  // 4.添加秒杀成功订单  OrderEntity orderEntity = **new** OrderEntity();  orderEntity.setSeckillId(seckillId);  orderEntity.setUserPhone(phone);  **int** insertOrder = orderMapper.insertOrder(orderEntity);  **if** (!toDaoResult(insertOrder)) {  **return** setResultError("系统错误!");  }  ***log***.info(">>>>phone:{},seckillId:{},秒杀成功", phone, seckillId);  **return** setResultSuccess("恭喜你,秒杀成功!");  }  } |

#### 数据库访问层

修改库存

|  |
| --- |
| **public** **interface** SeckillMapper {  /\*\*  \* 使用乐观锁修改库存信息 and inventory>0方式  \*  \* **@param** seckillId  \* **@return**  \*/  @Update("update meite\_seckill set inventory=inventory-1 where seckill\_id='10001' and inventory>0")  **int** optimisticLockSeckill(Long seckillId);  /\*\*  \* 基于版本号形式实现乐观锁  \*  \* **@param** seckillId  \* **@return**  \*/  @Update("update meite\_seckill set inventory=inventory-1 ,version=version+1 where seckill\_id=#{seckillId} and version=#{version} and inventory>0;")  **int** optimisticVersionSeckill(@Param("seckillId") Long seckillId, @Param("version") Long version);  @Update("update meite\_seckill set inventory=inventory-1 where seckill\_id='10001';")  **int** inventoryDeduction(Long seckillId);  @Select("SELECT seckill\_id AS seckillId,name as name,inventory as inventory,start\_time as startTime,end\_time as endTime,create\_time as createTime,version as version from meite\_seckill where seckill\_id=#{seckillId}")  SeckillEntity findBySeckillId(Long seckillId);  } |

#### 秒杀成功记录

|  |
| --- |
| **public** **interface** OrderMapper {  @Insert("INSERT INTO `meite\_order` VALUES (#{seckillId},#{userPhone}, '1', now());")  **int** insertOrder(OrderEntity orderEntity);  } |

### 秒杀服务基于库存令牌桶实现修改商品库存

#### 生产者

##### MQ相关配置

|  |
| --- |
| @Component  **public** **class** RabbitmqConfig {  // 添加修改库存队列  **public** **static** **final** String ***MODIFY\_INVENTORY\_QUEUE*** = "modify\_inventory\_queue";  // 交换机名称  **private** **static** **final** String ***MODIFY\_EXCHANGE\_NAME*** = "modify\_exchange\_name";  // 1.添加交换机队列  @Bean  **public** Queue directModifyInventoryQueue() {  **return** **new** Queue(***MODIFY\_INVENTORY\_QUEUE***);  }  // 2.定义交换机  @Bean  DirectExchange directModifyExchange() {  **return** **new** DirectExchange(***MODIFY\_EXCHANGE\_NAME***);  }  // 3.修改库存队列绑定交换机  @Bean  Binding bindingExchangeintegralDicQueue() {  **return** BindingBuilder.*bind*(directModifyInventoryQueue()).to(directModifyExchange()).with("modifyRoutingKey");  }  } |

##### 生产者发送消息

|  |
| --- |
| @Component  @Slf4j  **public** **class** SpikeCommodityProducer **implements** RabbitTemplate.ConfirmCallback {  @Autowired  **private** RabbitTemplate rabbitTemplate;  @Transactional  **public** **void** send(JSONObject jsonObject) {  String jsonString = jsonObject.toJSONString();  System.***out***.println("jsonString:" + jsonString);  String messAgeId = UUID.*randomUUID*().toString().replace("-", "");  // 封装消息  Message message = MessageBuilder.*withBody*(jsonString.getBytes())  .setContentType(MessageProperties.***CONTENT\_TYPE\_JSON***).setContentEncoding("utf-8").setMessageId(messAgeId)  .build();  // 构建回调返回的数据（消息id）  **this**.rabbitTemplate.setMandatory(**true**);  **this**.rabbitTemplate.setConfirmCallback(**this**);  CorrelationData correlationData = **new** CorrelationData(jsonString);  rabbitTemplate.convertAndSend("modify\_exchange\_name", "modifyRoutingKey", message, correlationData);  }  // 生产消息确认机制 生产者往服务器端发送消息的时候，采用应答机制  @Override  **public** **void** confirm(CorrelationData correlationData, **boolean** ack, String cause) {  String jsonString = correlationData.getId();  System.***out***.println("消息id:" + correlationData.getId());  **if** (ack) {  ***log***.info(">>>使用MQ消息确认机制确保消息一定要投递到MQ中成功");  **return**;  }  JSONObject jsonObject = JSONObject.*parseObject*(jsonString);  // 生产者消息投递失败的话，采用递归重试机制  send(jsonObject);  ***log***.info(">>>使用MQ消息确认机制投递到MQ中失败");  }  } |

#### 消费者

|  |
| --- |
| @Component  @Slf4j  **public** **class** StockConsumer {  @Autowired  **private** SeckillMapper seckillMapper;  @Autowired  **private** OrderMapper orderMapper;  @RabbitListener(queues = "modify\_inventory\_queue")  @Transactional  **public** **void** process(Message message, @Headers Map<String, Object> headers, Channel channel) **throws** IOException {  String messageId = message.getMessageProperties().getMessageId();  String msg = **new** String(message.getBody(), "UTF-8");  ***log***.info(">>>messageId:{},msg:{}", messageId, msg);  JSONObject jsonObject = JSONObject.*parseObject*(msg);  // 1.获取秒杀id  Long seckillId = jsonObject.getLong("seckillId");  SeckillEntity seckillEntity = seckillMapper.findBySeckillId(seckillId);  **if** (seckillEntity == **null**) {  ***log***.warn("seckillId:{},商品信息不存在!", seckillId);  **return**;  }  Long version = seckillEntity.getVersion();  **int** inventoryDeduction = seckillMapper.inventoryDeduction(seckillId, version);  **if** (!toDaoResult(inventoryDeduction)) {  ***log***.info(">>>seckillId:{}修改库存失败>>>>inventoryDeduction返回为{} 秒杀失败！", seckillId, inventoryDeduction);  **return**;  }  // 2.添加秒杀订单  OrderEntity orderEntity = **new** OrderEntity();  String phone = jsonObject.getString("phone");  orderEntity.setUserPhone(phone);  orderEntity.setSeckillId(seckillId);  orderEntity.setState(1l);  **int** insertOrder = orderMapper.insertOrder(orderEntity);  **if** (!toDaoResult(insertOrder)) {  **return**;  }  ***log***.info(">>>修改库存成功seckillId:{}>>>>inventoryDeduction返回为{} 秒杀成功", seckillId, inventoryDeduction);  }  // 调用数据库层判断  **public** Boolean toDaoResult(**int** result) {  **return** result > 0 ? **true** : **false**;  }  } |

#### 新增对应商品令牌桶

|  |
| --- |
| /\*\*  \* 新增对应商品库存令牌桶  \*  \* **@seckillId** 商品库存id  \*/  @RequestMapping("/addSpikeToken")  **public** BaseResponse<JSONObject> addSpikeToken(Long seckillId, Long tokenQuantity);  @Async  **private** **void** createSeckillToken(Long seckillId, Long tokenQuantity) {  generateToken.createListToken("seckill\_", seckillId + "", tokenQuantity);  } |

#### 根据手机号码和商品库存id查询秒杀记录

|  |
| --- |
| **public** **interface** OrderSeckillService {  @RequestMapping("/getOrder")  **public** BaseResponse<JSONObject> getOrder(String phone, Long seckillId);  }  @RestController  **public** **class** OrderSeckillServiceImpl **extends** BaseApiService<JSONObject> **implements** OrderSeckillService {  @Autowired  **private** OrderMapper orderMapper;  @Override  **public** BaseResponse<JSONObject> getOrder(String phone, Long seckillId) {  **if** (StringUtils.*isEmpty*(phone)) {  **return** setResultError("手机号码不能为空!");  }  **if** (seckillId == **null**) {  **return** setResultError("商品库存id不能为空!");  }  OrderEntity orderEntity = orderMapper.findByOrder(phone, seckillId);  **if** (orderEntity == **null**) {  **return** setResultError("正在排队中.....");  }  **return** setResultSuccess("恭喜你秒杀成功!");  }  } |

# 秒杀接口实现高并发限流方案

## Java高并发实现限流算法

常见限流算法常用的限流算法有：令牌桶，漏桶。

### 令牌桶算法

在秒杀活动中，用户的请求速率是不固定的，这里我们假定为10r/s，令牌按照5个每秒的速率放入令牌桶，桶中最多存放20个令牌。仔细想想，是不是总有那么一部分请求被丢弃。

### 漏桶算法

漏桶算法的主要目的是控制数据注入到网络的速率，平滑网络上的突发流量。漏桶算法提供了一种机制，通过它，突发流量可以被整形以便为网络提供一个稳定的流量

### 市面上常用实现限流框架

Nginx+Lua、Guava、hystrix等

### 基于谷歌RateLimiter实现限流

Google的Guava工具包中就提供了一个限流工具类——RateLimiter，本文也是通过使用该工具类来实现限流功能。RateLimiter是基于“令牌通算法”来实现限流的。

#### 项目整合方式

|  |
| --- |
| <dependency>  <groupId>com.google.guava</groupId>  <artifactId>guava</artifactId>  </dependency> |
| **private** RateLimiter rateLimiter = RateLimiter.*create*(1);  @Autowired  **private** GenerateToken generateToken;  @Override  **public** **void** service(RequestContext ctx, HttpServletRequest req, HttpServletResponse response) {  // 1.用户限流频率设置 每秒中限制1个请求  **boolean** tryAcquire = rateLimiter.tryAcquire(0, TimeUnit.***SECONDS***);  **if** (!tryAcquire) {  resultError(500, ctx, "现在抢购的人数过多，请稍等一下下哦！");  **return**;  }  // 2.使用redis限制用户访问频率  String seckillId = req.getParameter("seckillId");  String seckillToken = generateToken.getListKeyToken(seckillId + "");  **if** (StringUtils.*isEmpty*(seckillToken)) {  ***log***.info(">>>seckillId:{}, 亲，该秒杀已经售空，请下次再来!", seckillId);  resultError(500, ctx, "亲，该秒杀已经售空，请下次再来!");  **return**;  }  // 3.执行修改库存操作  nextGatewayHandler.service(ctx, req, response);  } |

### 基于网关实现高并发限流

|  |
| --- |
| @Component  @Slf4j  **public** **class** CurrentLimitHandler **extends** BaseHandler **implements** GatewayHandler {  **private** RateLimiter rateLimiter = RateLimiter.*create*(1);  @Autowired  **private** GenerateToken generateToken;  @Override  **public** **void** service(RequestContext ctx, HttpServletRequest req, HttpServletResponse response) {  // 1.用户限流频率设置 每秒中限制1个请求  **boolean** tryAcquire = rateLimiter.tryAcquire(0, TimeUnit.***SECONDS***);  **if** (!tryAcquire) {  resultError(500, ctx, "现在抢购的人数过多，请稍等一下下哦！");  **return**;  }  // 2.使用redis限制用户访问频率  String seckillId = req.getParameter("seckillId");  String seckillToken = generateToken.getListKeyToken(seckillId + "");  **if** (StringUtils.*isEmpty*(seckillToken)) {  ***log***.info(">>>seckillId:{}, 亲，该秒杀已经售空，请下次再来!", seckillId);  resultError(500, ctx, "亲，该秒杀已经售空，请下次再来!");  **return**;  }  // 3.执行修改库存操作  nextGatewayHandler.service(ctx, req, response);  }  } |

###>>>>>秒杀接口线程池名称:hystrix-SpikeCommodityServiceImpl-1

### 秒杀接口使用hystrix实现线程隔离

## 服务降级、熔断、限流概念

### 服务雪崩效应

服务雪崩效应产生与服务堆积在同一个线程池中，因为所有的请求都是同一个线程池进行处理，这时候如果在高并发情况下，所有的请求全部访问同一个接口，

这时候可能会导致其他服务没有线程进行接受请求，这就是服务雪崩效应效应。

### 服务降级

在高并发情况下，防止用户一直等待，使用服务降级方式(直接返回一个友好的提示给客户端，调用fallBack方法)

### 服务熔断

熔断机制目的为了保护服务，在高并发的情况下，如果请求达到一定极限(可以自己设置阔值)如果流量超出了设置阈值，让后直接拒绝访问，保护当前服务。使用服务降级方式返回一个友好提示，服务熔断和服务降级一起使用

### 服务隔离

因为默认情况下，只有一个线程池会维护所有的服务接口，如果大量的请求访问同一个接口，达到tomcat 线程池默认极限，可能会导致其他服务无法访问。

解决服务雪崩效应:使用服务隔离机制(线程池方式和信号量)，使用线程池方式实現隔离的原理: 相当于每个接口(服务)都有自己独立的线程池，因为每个线程池互不影响，这样的话就可以解决服务雪崩效应。

线程池隔离:

每个服务接口，都有自己独立的线程池，每个线程池互不影响。

信号量隔离:

使用一个原子计数器（或信号量）来记录当前有多少个线程在运行，当请求进来时先判断计数器的数值，若超过设置的最大线程个数则拒绝该请求，若不超过则通行，这时候计数器+1，请求返 回成功后计数器-1。

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>  </dependency> |

@EnableHystrix 开启hystrix

|  |
| --- |
| @Override  @Transactional  @HystrixCommand(fallbackMethod = "spikeFallback")  **public** BaseResponse<JSONObject> spike(String phone, Long seckillId) {  ***log***.info(">>>>>>秒杀服务接口:spike()线程名称:" + Thread.*currentThread*().getName());  // 1.秒杀服务实现限流  // boolean tryAcquire = rateLimiter.tryAcquire(0, TimeUnit.SECONDS);  // if (!tryAcquire) {  // return setResultError("服务忙，请稍后重试!");  // }  // 1.参数验证  **if** (StringUtils.*isEmpty*(phone)) {  **return** setResultError("手机号码不能为空!");  }  **if** (seckillId == **null**) {  **return** setResultError("商品库存id不能为空!");  }  // 3.获取到秒杀token之后，异步放入mq中实现修改商品的库存  sendSeckillMsg(seckillId, phone);  **return** setResultSuccess("正在排队中.......");  }  **private** BaseResponse<JSONObject> spikeFallback(String phone, Long seckillId) {  **return** setResultError("服务器忙,请稍后重试!");  } |

###>>>>>秒杀接口线程池名称:http-nio-9800-exec-1

>>>>>查询秒杀结果线程名称:http-nio-9800-exec-2

**前端：**

1. **使用动静分离、将静态资源存放到第三方文件服务器中实现cdn加速，目的减轻秒杀抢购带宽**
2. **当用户点击秒杀按钮的时候，应该将按钮disabled 防止重复提交**
3. **使用复杂的图形验证码防止机器模拟**
4. **秒杀详情页面，使用定时器根据用户信息查询秒杀结果**
5. **商品的详情页面使用nginx+lua+openresty 实现静态化页面**

**网关：**

1. **ratelimter、nginx、hystrix、redis实现限流 令牌痛+漏铜算法 对用户秒杀请求实现限流和服务保护。**
2. **用户黑名单和白名单拦截**

**秒杀接口:**

1. **服务降级级、隔离、熔断**
2. **从redis中获取秒杀的令牌（能够获取到令牌就能够秒杀成功,否则就秒杀失败!）**
3. **异步使用MQ执行修改库存操作**
4. **提供一个根据用户信息查询秒杀结果接口**

**项目部署点：**

**Nginx+lvs 实现服务高可用和集群**

**其他点：**

**分时段抢购**

**具体例子：现在有100个商品同时秒杀抢购，每个商品库存为100个？**

**基于mq+库存令牌桶 100\*100=10000 数据库执行修改库存操作压力还是非常大？**

**最靠谱的访问：12306 分时段秒杀 中午 下午**