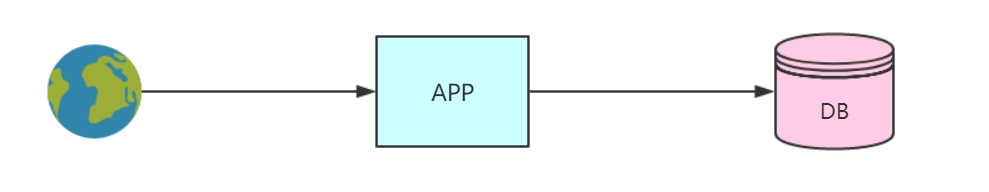
### 架构演变过程

我们最先接触的单体架构，整个系统就只有一个工程，打包往往是打成了war包，然后部署到单一tomcat上面，这种就是单体架构，如图：



假如系统按照功能划分了，商品模块，购物车模块，订单模块，物流模块等等模块。那么所有模块都会在一个工程里面，这就是单体架构。

单体架构优点

1、结构简单，部署简单

2、所需的硬件资源少

3、节省成本

缺点

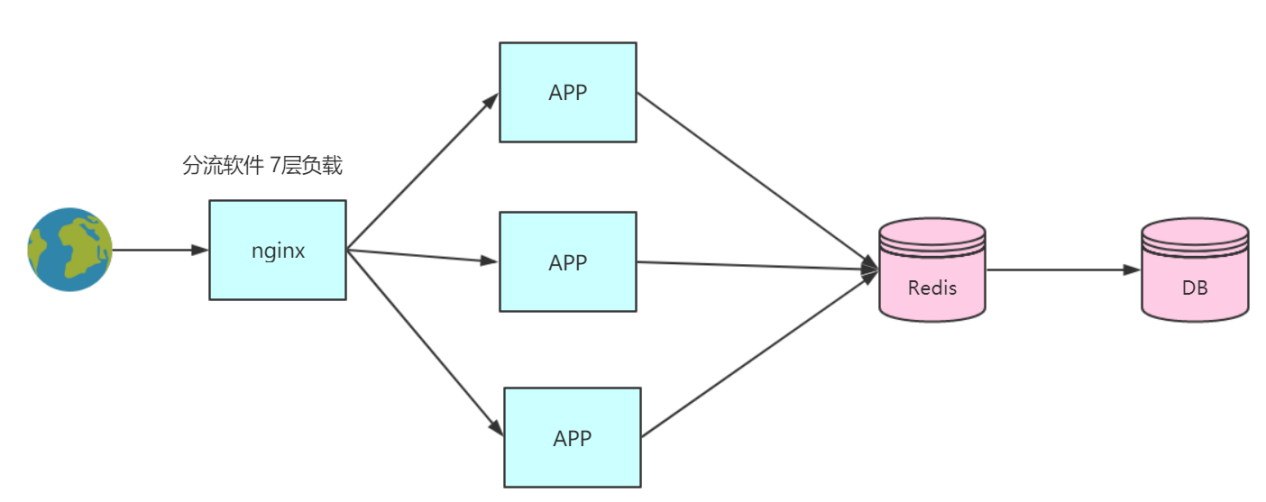
1、版本迭代慢，往往改动一个代码会影响全局

2、不能满足一定并发的访问

3、代码维护困难，所有代码在一个工程里面，存在被其他人修改的风险

随着业务的拓展，公司的发展，单体架构慢慢的不能满足我们的需求，我们需要对架构进行变动，我们能够想到的最简单的办法就是加机器，对应用横向扩展。

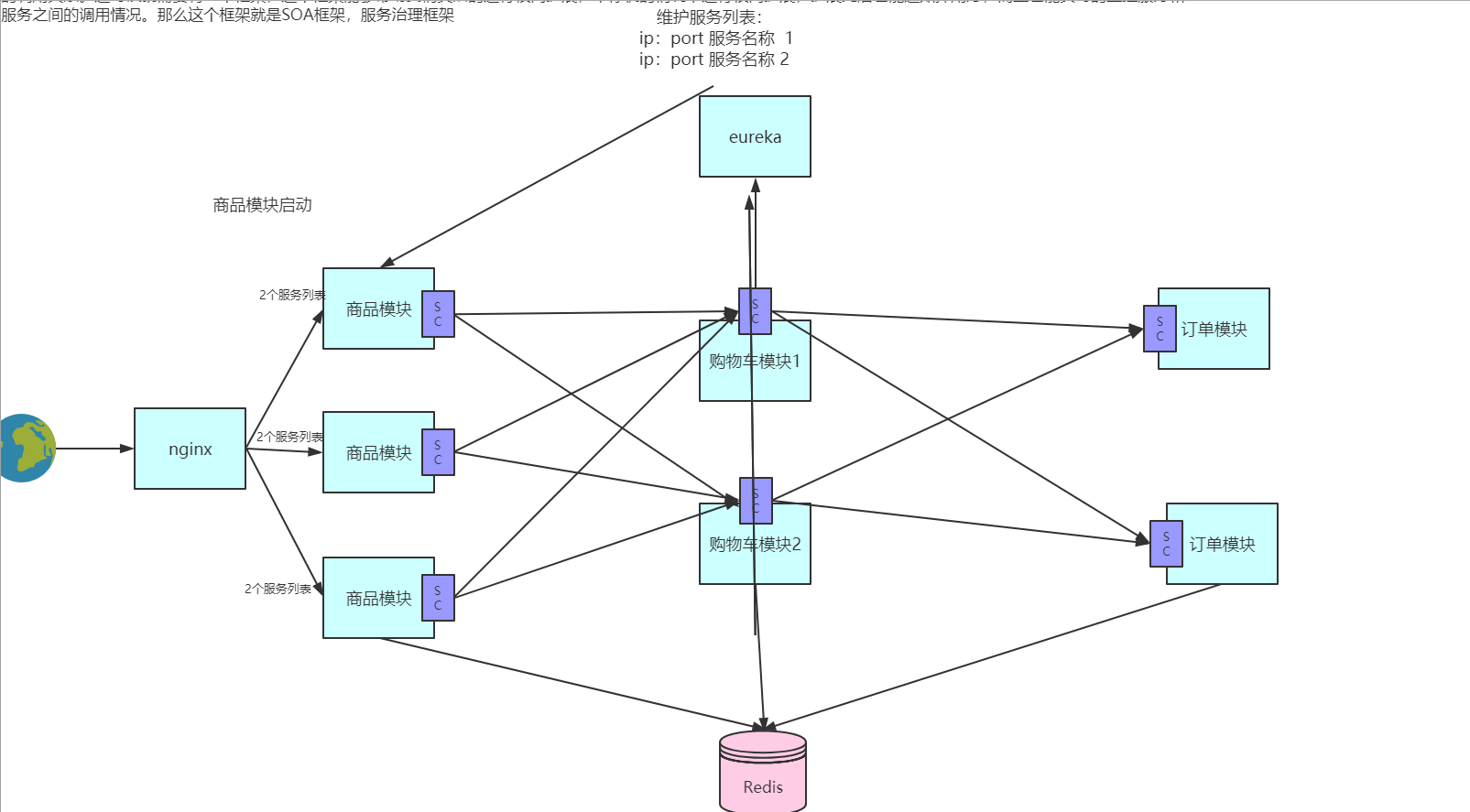
如图：



这种架构貌似暂时解决了我们的问题，但是用户量慢慢增加后，我们只能通过横向加机器来解决，还是会存在版本迭代慢，代码维护困难的问题。而且用户请求往往是读多写少的情况，所以可能真正需要扩容的只是商品模块而已，而现在是整个工程都扩容了，这无形中是一种资源的浪费，因为其他模块可能根本不需要扩容就可以满足需求。所以我们有必要对整个工程按照模块进行拆分，拆分后的架构图如下：



模块拆分后，模块和模块之间是需要通过接口调用的方式进行通信，模块和模块之间通过分流软件进行负载均衡。这个架构解决前面的资源浪费问题和代码管理问题，因为我们是对系统拆分了，各个模块都有单独的工程，比如我修改商品模块，就不需要担心会不会影响购物车模块。但是这种架构扩展非常麻烦，一旦需要横向加机器，或者减机器都需要修改nginx配置，一旦机器变多了以后，nginx的配置量就是一个不能完成的工作。OK，这时候SOA服务治理框架就应运而生，架构图如下：



基于注册中心的SOA框架，扩展是非常方便的，因为不需要维护分流工具，但我们启动应用的时候就会把服务通过http的方式注册到注册中心。

在SOA框架中一般会有三种角色：1、注册中心 2、服务提供方 3、服务消费方

1. 注册中心

在注册中心维护了服务列表

1. 服务提供方

服务提供方启动的时候会把自己注册到注册中心

1. 服务消费方

服务消费方启动的时候，把获取注册中心的服务列表，然后调用的时候从这个服务列表中选择某一个去调用。

微服务工程的特点：

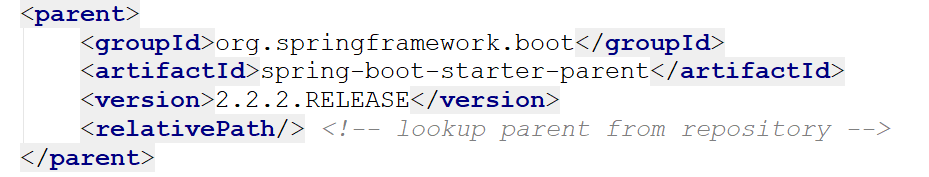
1. 扩展灵活
2. 每个应用都规模不大
3. 服务边界清晰，各司其职
4. 打包应用变多，往往需要借助CI持续集成工具

### 搭建简单的微服务工程

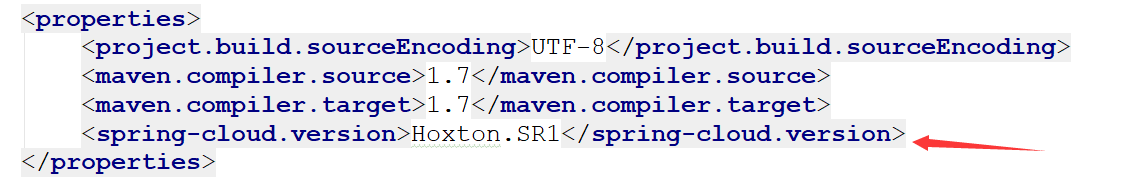
1. 注册中心搭建

Springcloud中，我们选择eureka作为注册中心，springcloud工程是基于springboot工程的。

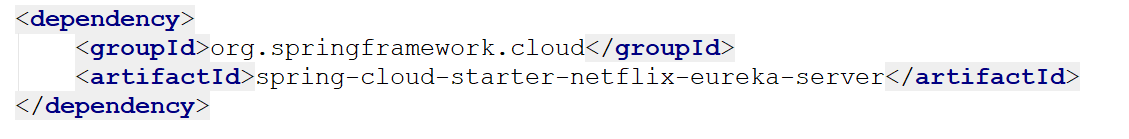
pom.xml中jar包依赖：



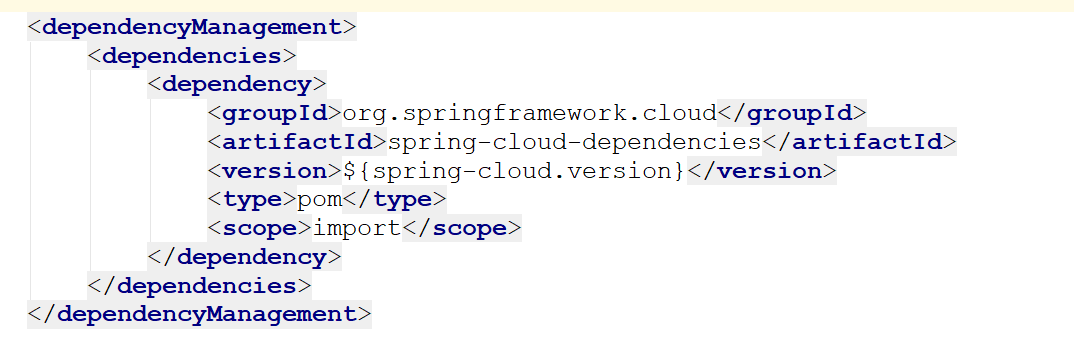
Springcloud的版本



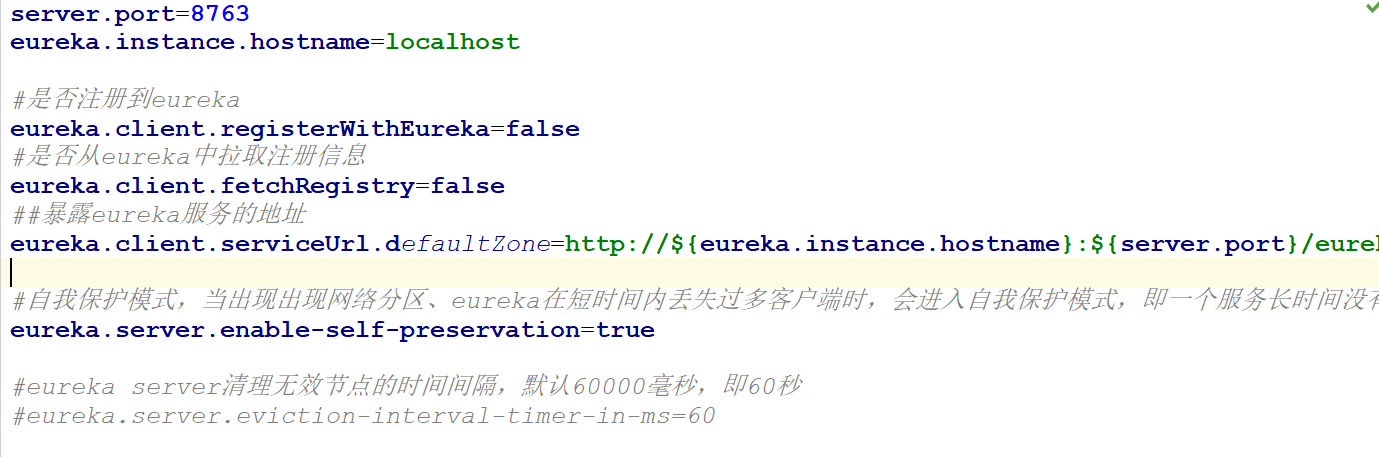
Eureka服务端启动器导入



Springcloud的依赖仓库导入

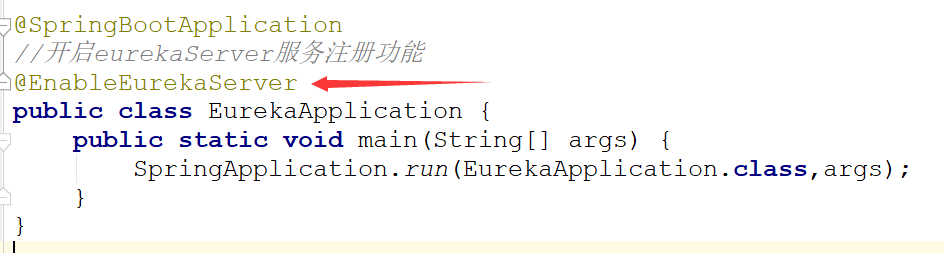


Application.properties配置文件



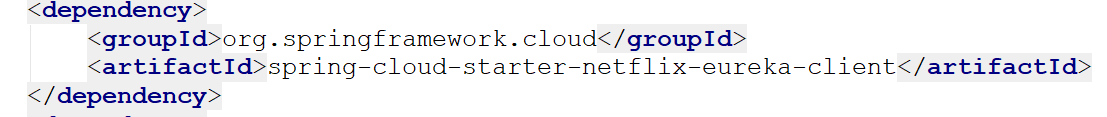
配置文件属性后续会详解

启动类

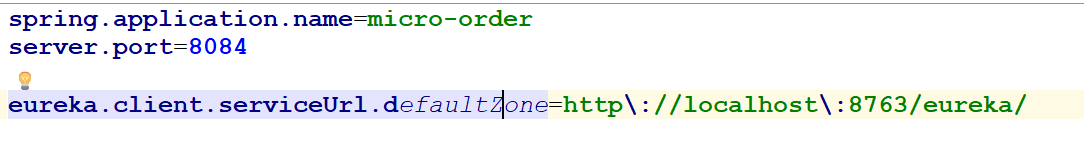


1. 服务提供方

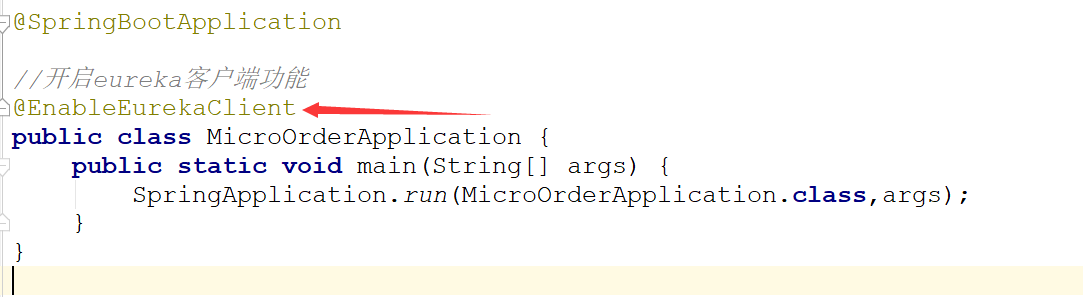
Pom的jar包依赖，其他都跟eureka服务端是一样的，只是服务提供方要把服务注册到eureka服务端，所以服务提供方就是eureka的客户端，所以需要导入eureka客户端的启动器。



bootstrap.properties

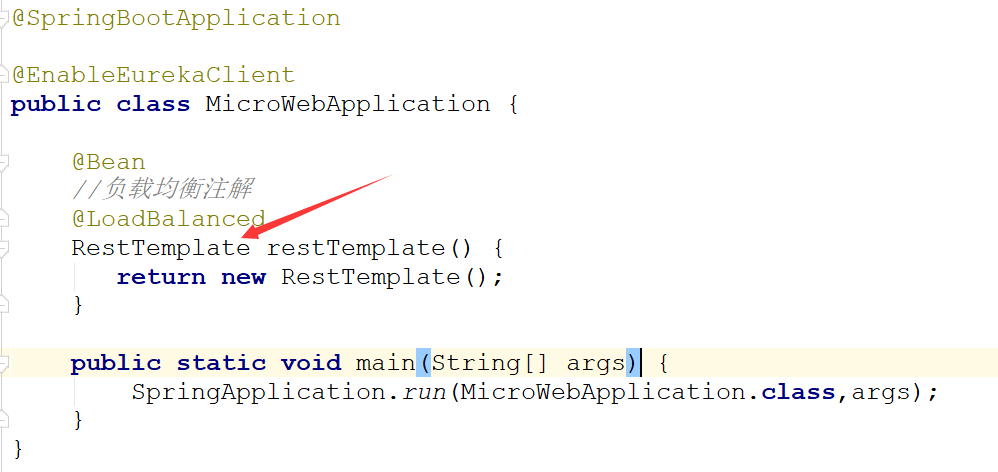


启动类

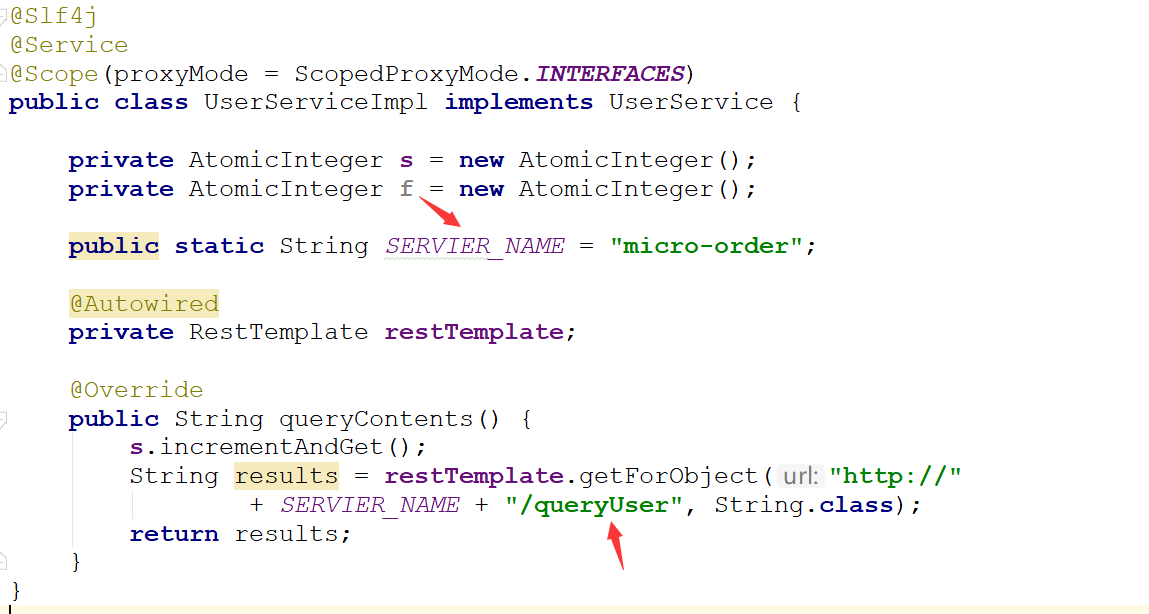


1. 服务消费方

pom和属性配置文件基本上差不多，消费要负责调用服务提供方，所以需要调用客户端

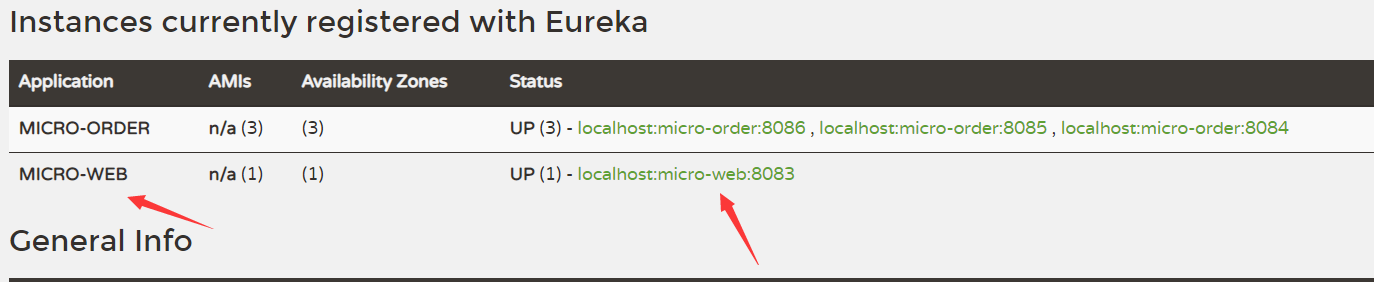


服务调用的时候就根据服务提供方的服务名称来调用的



服务提供方的名称，再加上服务提供方的接口名就可以完成调用了。

服务提供方和服务消费启动的时候都会往服务注册中心注册服务，eureka服务端也可以通过界面查看到服务注册情况：

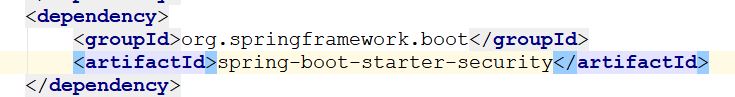


### Eureka用户认证

连接到eureka的时候需要带上连接的用户名和密码

eureka服务端改造

添加启动器



代码配置，关闭csrf验证



Application.properties配置

开启basic校验,设置登录用户名密码

**security.basic.enabled**=**true  
spring.security.user.name**=**admin  
spring.security.user.password**=**admin**

Eureka客户端改造

跟eureka连接的时候要带上用户名密码

**eureka.client.serviceUrl.defaultZone**=**http://admin:admin@localhost:8763/eureka/**

### 服务续约保活

当客户端启动想eureka注册了本身服务列表后，需要隔段时间发送一次心跳给eureka服务端来证明自己还活着，当eureka收到这个心跳请求后才会知道客户端还活着，才会维护该客户端的服务列表信息。一旦因为某些原因导致客户端没有按时发送心跳给eureka服务端，这时候eureka可能会认为你这个客户端已经挂了，它就有可能把该服务从服务列表中删除掉。

有关续约保活的配置

客户端配置

*#服务续约，心跳的时间间隔***eureka.instance.lease-renewal-interval-in-seconds**=**30***#如果从前一次发送心跳时间起，90秒没接受到新的心跳，讲剔除服务***eureka.instance.lease-expiration-duration-in-seconds**=**90**

*#表示eureka client间隔多久去拉取服务注册信息，默认为30秒***eureka.client.registry-fetch-interval-seconds**=**30**

服务端配置

*#自我保护模式，当出现出现网络分区、eureka在短时间内丢失过多客户端时，会进入自我保护模式，即一个服务长时间没有发送心跳，eureka也不会将其删除，默认为true***eureka.server.enable-self-preservation**=**true**

*#Eureka Server 在运行期间会去统计心跳失败比例在 15 分钟之内是否低于 85%，如果低于 85%，Eureka Server 会将这些实例保护起来***eureka.server.renewal-percent-threshold**=**0.85**

*#eureka server清理无效节点的时间间隔，默认60000毫秒，即60秒***eureka.server.eviction-interval-timer-in-ms**=**60000**

### Eureka健康检测

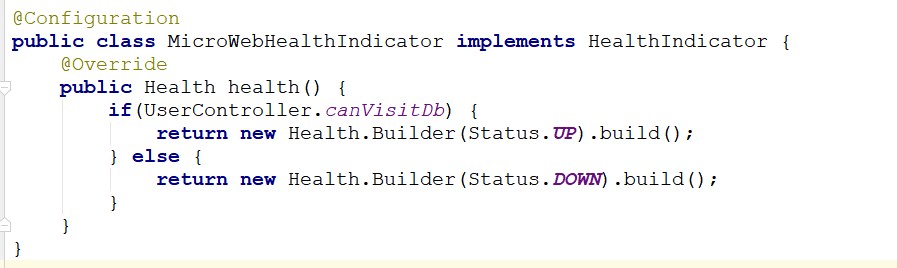
Eureka默认的健康检测只是你校验服务连接是否是UP还是DOWN的，然后客户端只会调用状态为UP状态的服务，但是有的情况下，虽然服务连接是好的，但是有可能这个服务的某些接口不是正常的，可能由于需要连接Redis，mongodb或者DB有问题导致接口调用失败，所以理论上服务虽然能够正常调用，但是它不是一个健康的服务。所以我们就有必要对这种情况做自定义健康检测。

Application.properties配置

开启健康检测

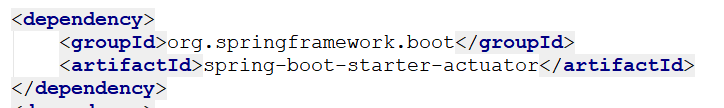
*#健康检测***eureka.client.healthcheck.enabled**=**true**

自定义健康检测代码



我们可以在health方法里面去连接数据库，如果连接异常了则返回DOWN，如果没异常则方法UP，这个health方法是线程去掉的，隔一段时间掉一次

检测检测依赖的jar包



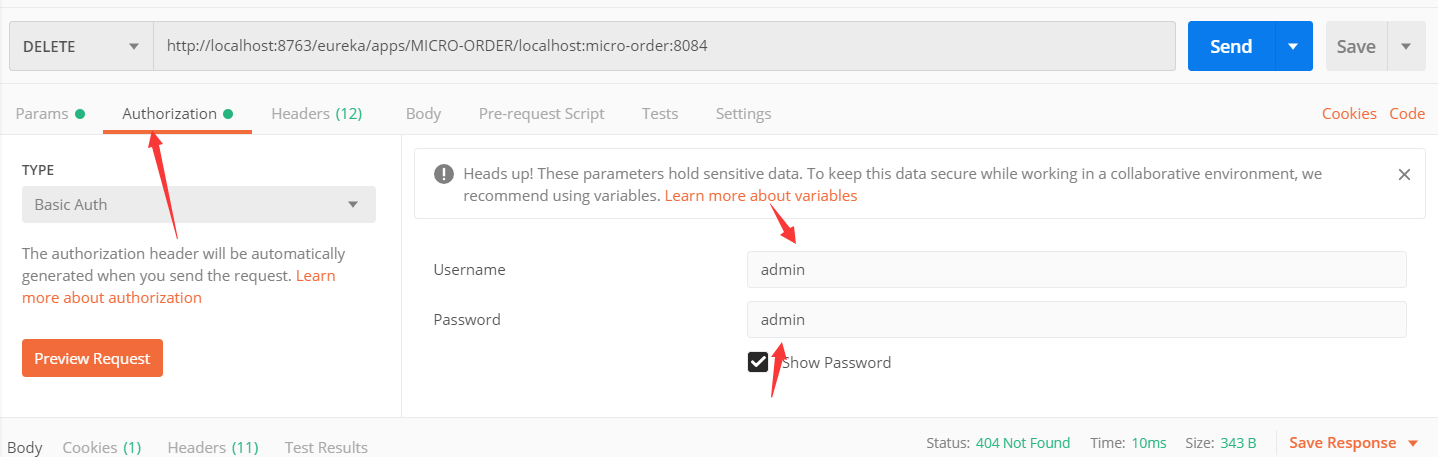
### 服务下线

比如有些情况是服务主机意外宕机了，也就意味着服务没办法给eureka心跳信息了，但是eureka在没有接受到心跳的情况下依赖维护该服务90s，在这90s之内可能会有客户端调用到该服务，这就可能会导致调用失败。所以我们必须要有一个机制能手动的立马把宕机的服务从eureka服务列表中清除掉，避免被服务调用方调用到。

调用服务下线的接口：

这个接口是调用eureka服务端的接口

http://localhost:8763/eureka/apps/MICRO-ORDER/localhost:micro-order:8084



### Eureka高可用

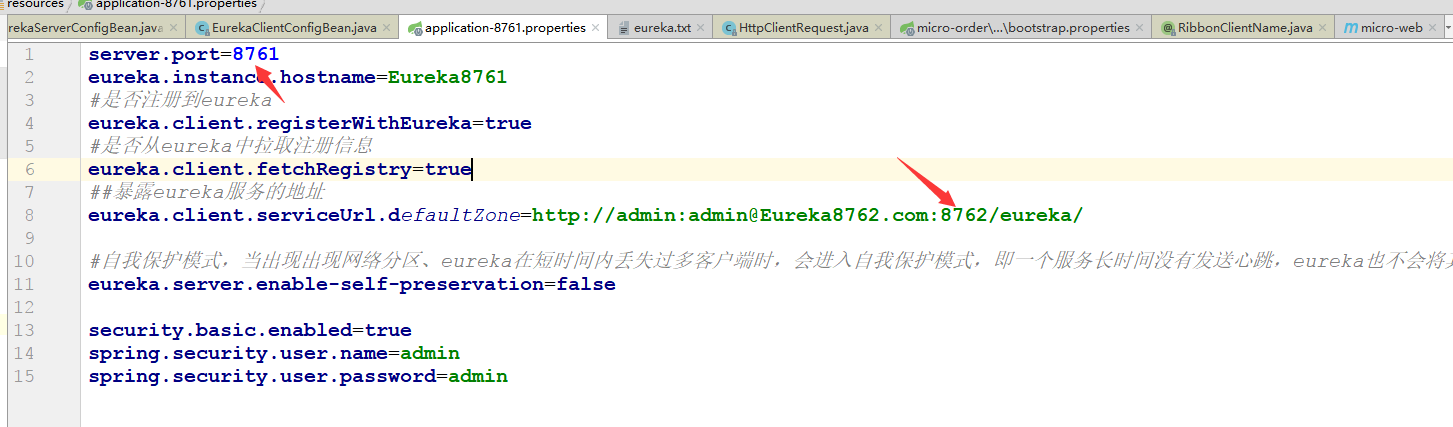
Eureka热备份的架构图如下：



整个微服务中存在多个eureka服务，每个eureka服务都是相互复制的，会把客户端注册进来的服务复制到eureka集群中的其他节点里面来。其实简单来说就是 eureka每个节点相互复制。

具体配置如下：

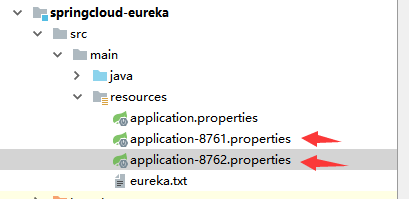
端口为8761的eureka服务端把自己注册到8762的eureka服务端



同样的道理，8762注册到8761

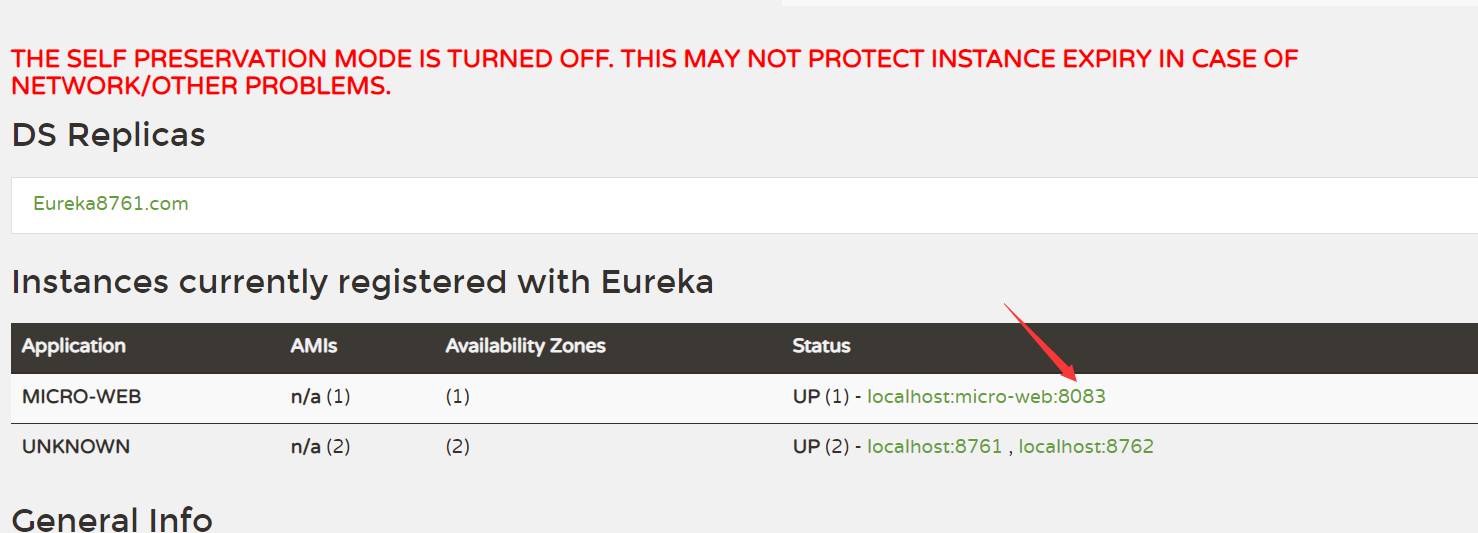


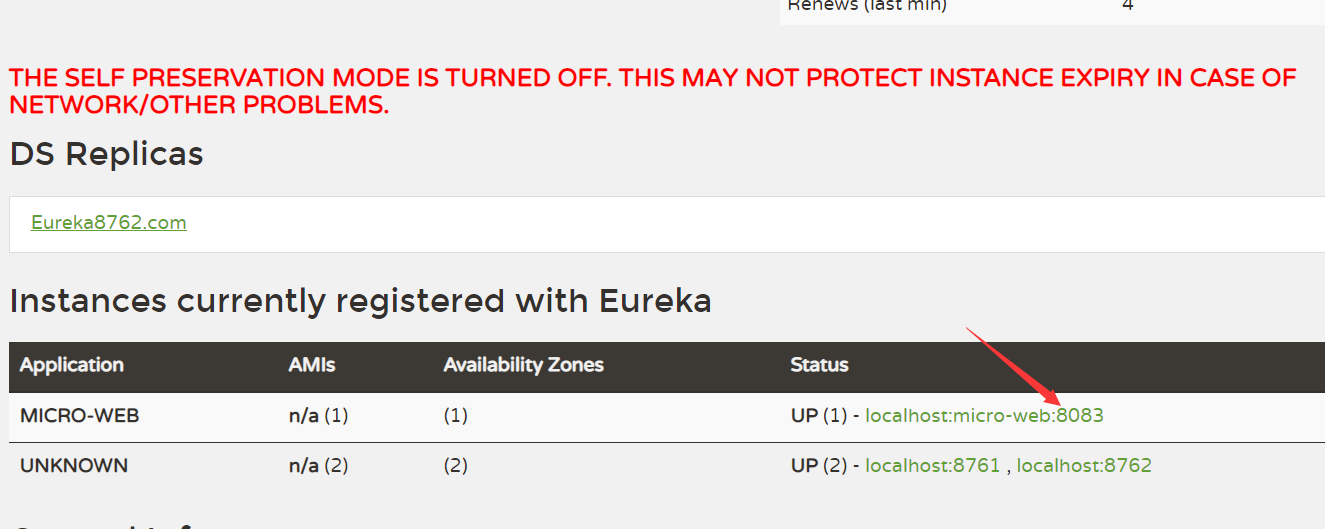
配置存在在两个配置文件中



启动的时候按照指定配置文件启动

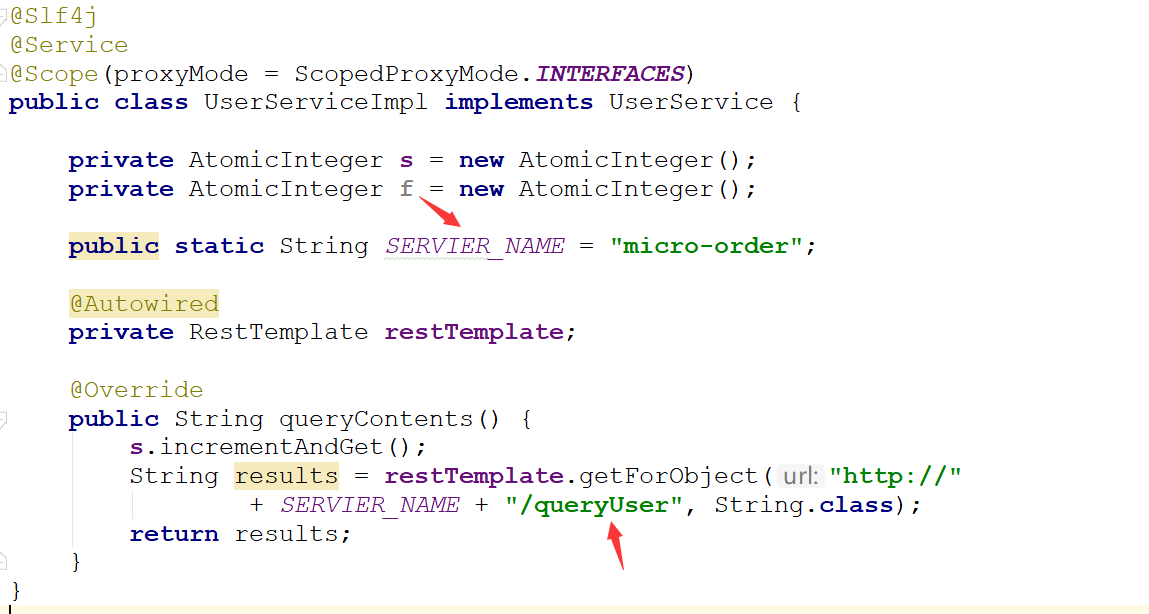
java -jar springcloud-eureka.jar --spring.profiles.active=8761





### Ribbon API

Ribbon是一个独立的组件，是用来进行远程接口调用的，代码如下



通过getForObject方法可以掉到用 micro-order服务的，queryUser接口。然后在调用期间会存在负载均衡，micro-order服务对应有几个服务实例就会根据负载均衡算法选择某一个去调用。

Get请求

getForEntity：此方法有三种重载形式，分别为：

getForEntity(String url, Class<T> responseType)

getForEntity(String url, Class<T> responseType, Object... uriVariables)

getForEntity(String url, Class<T> responseType, Map<String, ?> uriVariables)

getForEntity(URI url, Class<T> responseType)

注意：此方法返回的是一个包装对象ResponseEntity<T>其中T为responseType传入类型，想拿到返回类型需要使用这个包装类对象的getBody()方法

getForObject:此方法也有三种重载形式，这点与getForEntity方法相同：

getForObject(String url, Class<T> responseType)

getForObject(String url, Class<T> responseType, Object... uriVariables)

getForObject(String url, Class<T> responseType, Map<String, ?> uriVariables)

getForObject(URI url, Class<T> responseType)

注意：此方法返回的对象类型为responseType传入类型

Post请求

post请求和get请求都有\*ForEntity和\*ForObject方法，其中参数列表有些不同，除了这两个方法外，还有一个postForLocation方法，其中postForLocation以post请求提交资源，并返回新资源的URI

postForEntity：此方法有三种重载形式，分别为：

postForEntity(String url, Object request, Class<T> responseType, Object... uriVariables)

postForEntity(String url, Object request, Class<T> responseType, Map<String, ?> uriVariables)

postForEntity(URI url, Object request, Class<T> responseType)

注意：此方法返回的是一个包装对象ResponseEntity<T>其中T为responseType传入类型，想拿到返回类型需要使用这个包装类对象的getBody()方法

postForObject:此方法也有三种重载形式，这点与postForEntity方法相同：

postForObject(String url, Object request, Class<T> responseType, Object... uriVariables)

postForObject(String url, Object request, Class<T> responseType, Map<String, ?> uriVariables)

postForObject(URI url, Object request, Class<T> responseType)

注意：此方法返回的对象类型为responseType传入类型

postForLocation:此方法中同样有三种重载形式，分别为：

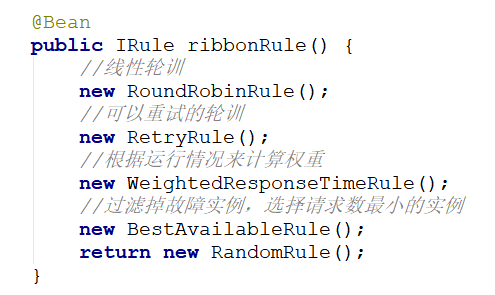
postForLocation(String url, Object request, Object... uriVariables)

postForLocation(String url, Object request, Map<String, ?> uriVariables)

postForLocation(URI url, Object request)

注意：此方法返回的是新资源的URI，相比getForEntity、getForObject、postForEntity、postForObject方法不同的是这个方法中无需指定返回类型，因为返回类型就是URI，通过Object... uriVariables、Map<String, ?> uriVariables进行传参依旧需要占位符，参看postForEntity部分代码

### 负载均衡算法



*//线性轮训***new** RoundRobinRule();  
*//可以重试的轮训***new** RetryRule();  
*//根据运行情况来计算权重***new** WeightedResponseTimeRule();  
*//过滤掉故障实例，选择请求数最小的实例***new** BestAvailableRule();

//随机

**new** RandomRule();

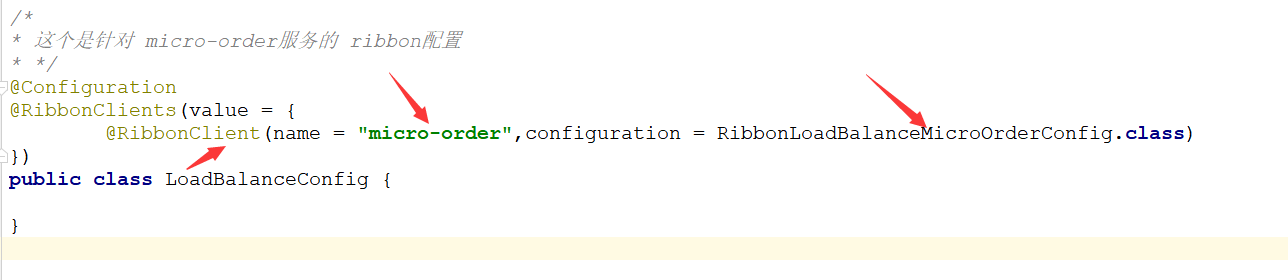
### Ribbon配置

Application.properties配置

*#点对点直连测试配置  
# 关闭ribbon访问注册中心Eureka Server发现服务，但是服务依旧会注册。  
#true使用eureka false不使用***ribbon.eureka.enabled**=**true  
spring.cloud.loadbalancer.retry.enabled**=**true***#指定调用的节点***micro-order.ribbon.listOfServers=localhost:8001** *#单位ms ,请求连接超时时间***micro-order.ribbon.ConnectTimeout=1000** *#单位ms ,请求处理的超时时间***micro-order.ribbon.ReadTimeout=2000  
  
micro-order.ribbon.OkToRetryOnAllOperations=true** *#切换实例的重试次数***micro-order.ribbon.MaxAutoRetriesNextServer=2***#对当前实例的重试次数 当Eureka中可以找到服务，但是服务连不上时将会重试***micro-order.ribbon.MaxAutoRetries=2  
  
micro-order.ribbon.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RandomRule  
micro-order.ribbon.NFLoadBalancerPingClassName=com.netflix.loadbalancer.PingUrl**

代码配置

使用@RibbonClients加载配置



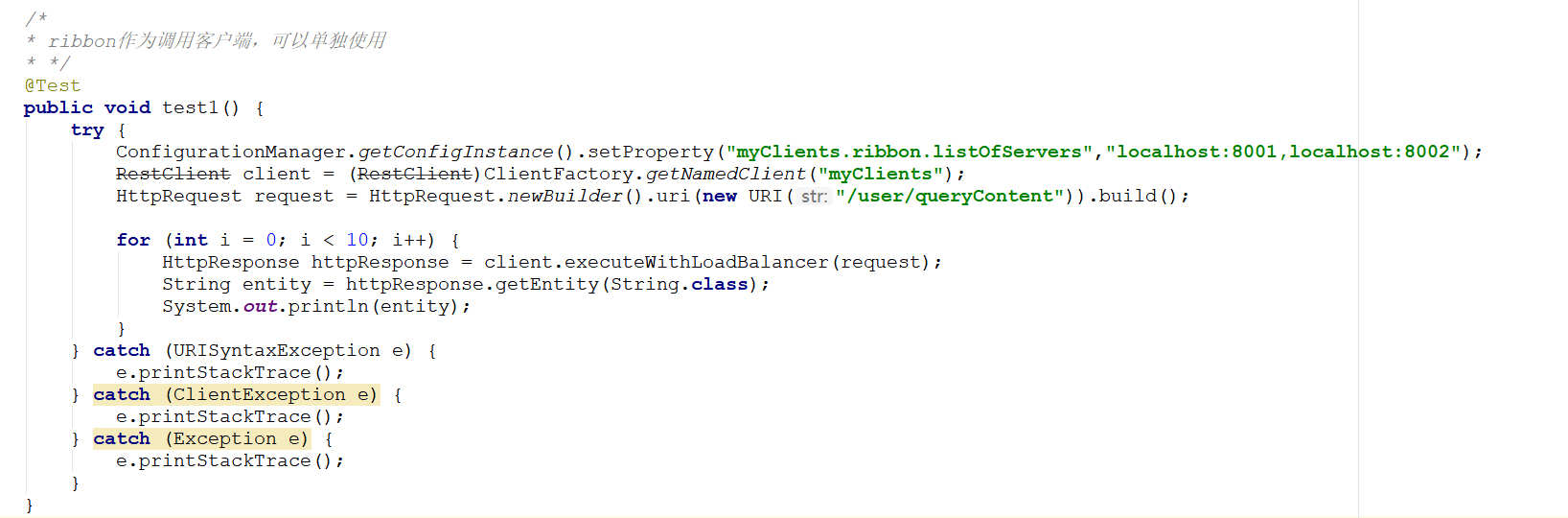
这个配置类只针对micro-order服务，微服务系统里面有很多服务，这就可以区别化配置。

配置configuration配置类的时候，一定要注意，配置类不能陪@ComponentScan注解扫描到，如果被扫描到了则该配置类就是所有服务共用的配置了。



### Ribbon单独使用

Ribbon是一个独立组件，可以脱离springcloud使用的

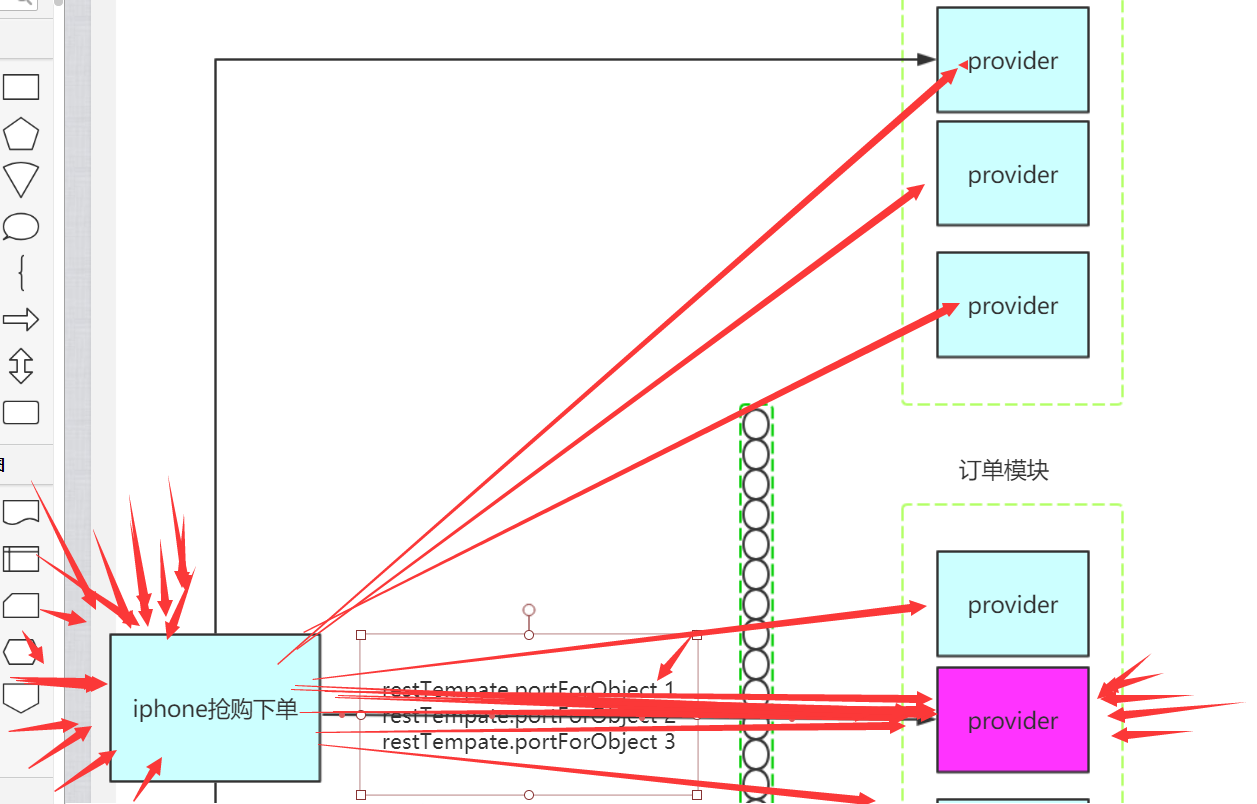


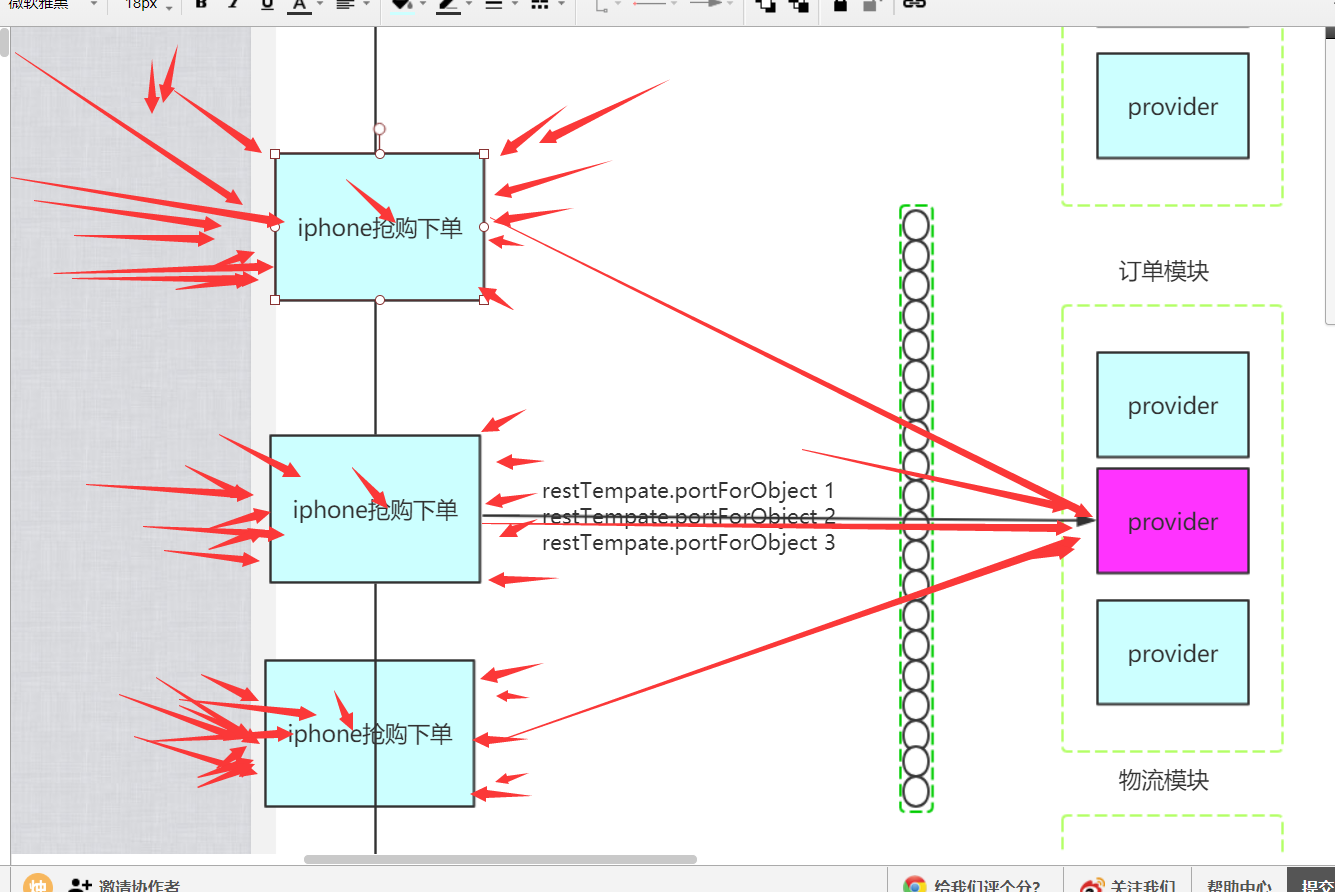
需要依赖两个jar



### 服务雪崩

雪崩是系统中的蝴蝶效应导致其发生的原因多种多样，有不合理的容量设计，或者是高并发下某一个方法响应变慢，亦或是某台机器的资源耗尽。从源头上我们无法完全杜绝雪崩源头的发生，但是雪崩的根本原因来源于服务之间的强依赖，所以我们可以提前评估。当整个微服务系统中，有一个节点出现异常情况，就有可能在高并发的情况下出现雪崩，导致调用它的上游系统出现响应延迟，响应延迟就会导致tomcat连接本耗尽，导致该服务节点不能正常的接收到正常的情况，这就是服务雪崩行为。



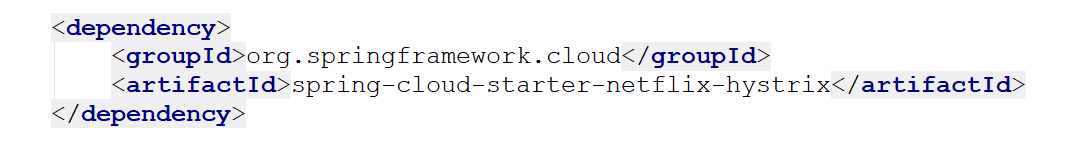


### 服务隔离

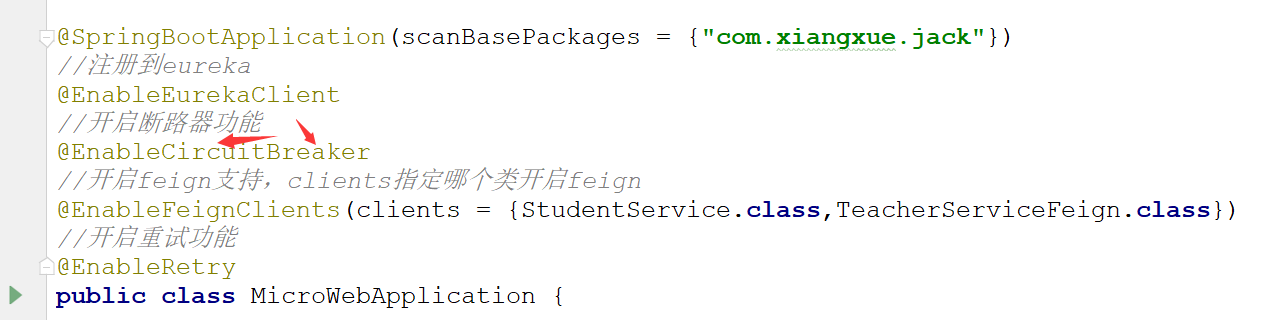
如果整个系统雪崩是由于一个接口导致的，由于这一个接口响应不及时导致问题，那么我们就有必要对这个接口进行隔离，就是只允许这个接口最多能接受多少的并发，做了这样的限制后，该接口的主机就会空余线程出来接收其他的情况，不会被哪个坏了的接口占用满。

Hystrix就是一个不错的服务隔离框架。

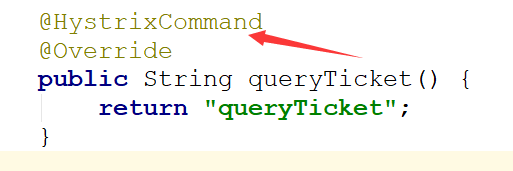
Hystrix的导入



启动类开启hystrix功能



代码使用



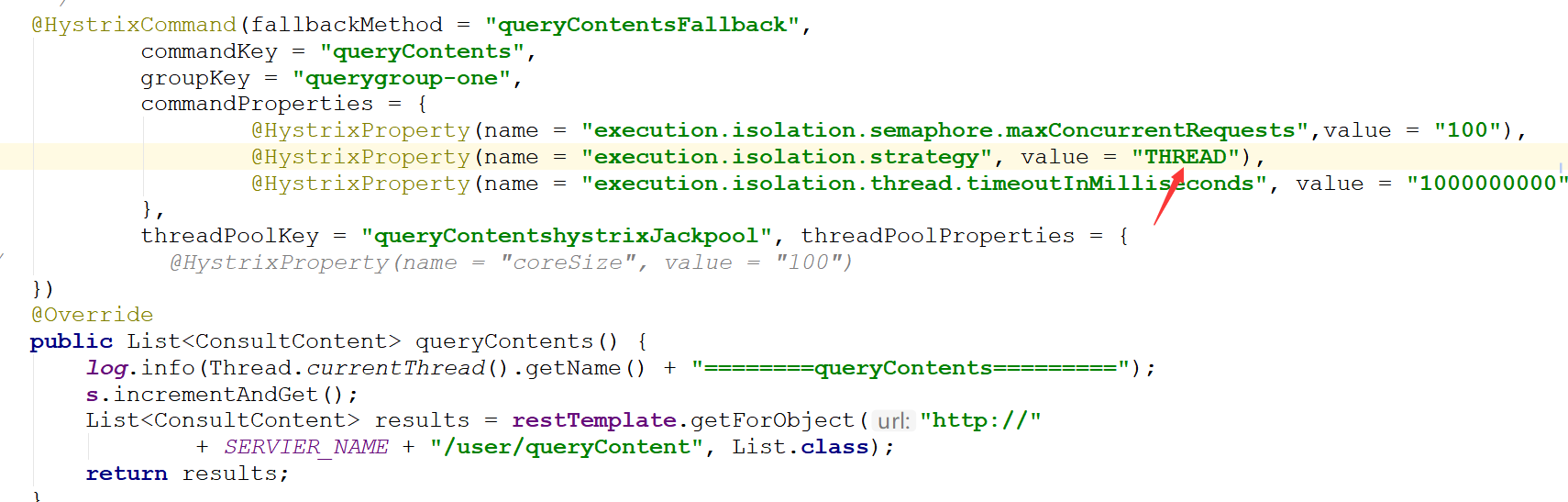
### Hystrix服务隔离策略

1. 线程池隔离

THREAD 线程池隔离策略 独立线程接收请求 默认的

默认采用的就是线程池隔离

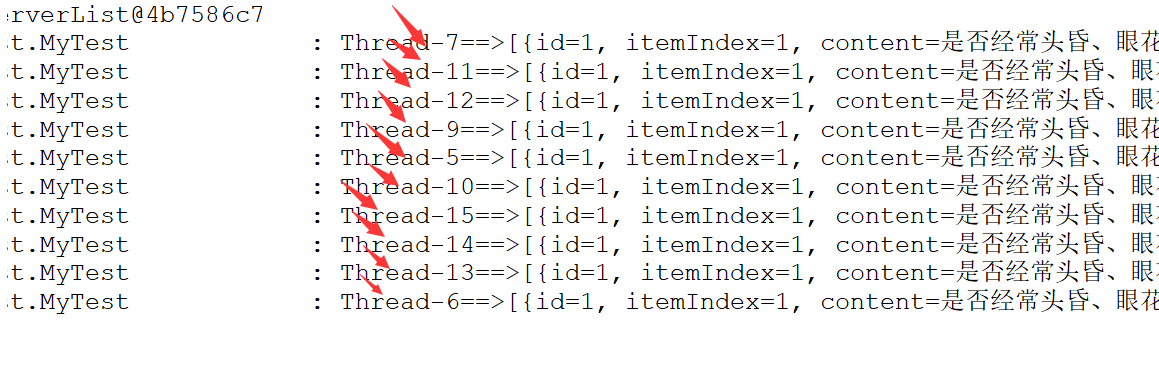
代码配置

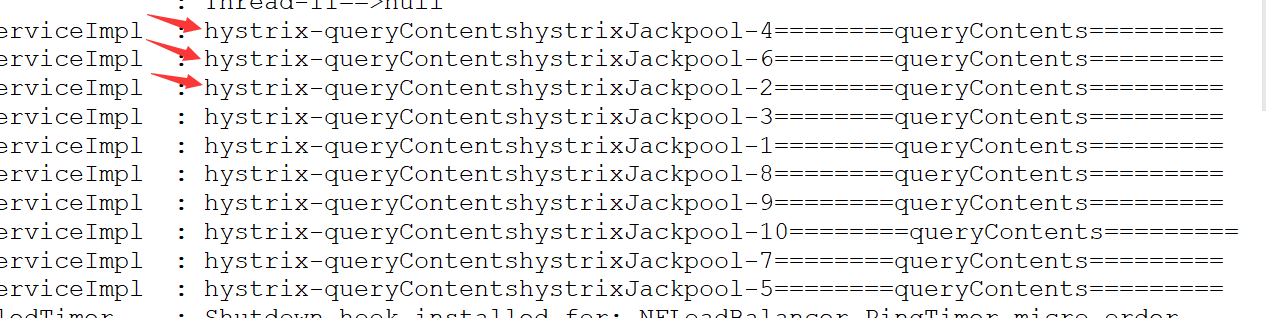


默认线程池中有10个线程，可以配置线程池中线程大小



线程池隔离策略，hystrix是会单独创建线程的，单元测试如下：





可以看到，用户线程和业务类中的线程是不一样的

1. 信号量隔离

信号量隔离是采用一个全局变量来控制并发量，一个请求过来全局变量加1，单加到跟配置中的大小相等是就不再接受用户请求了。

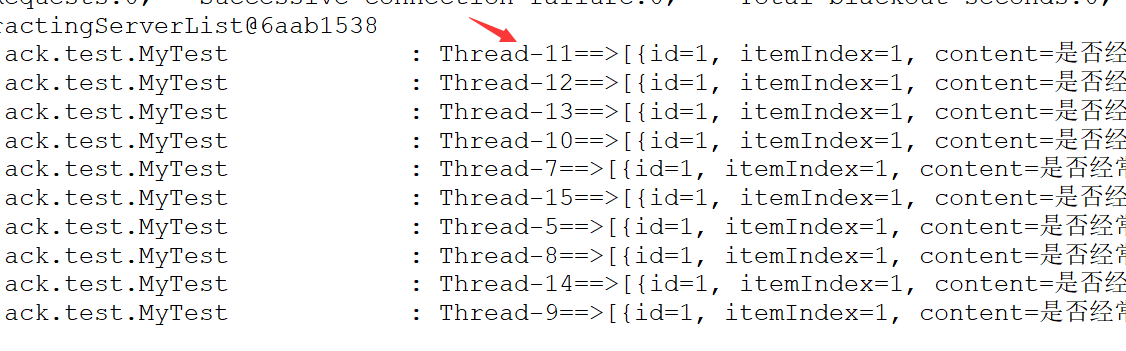
代码配置

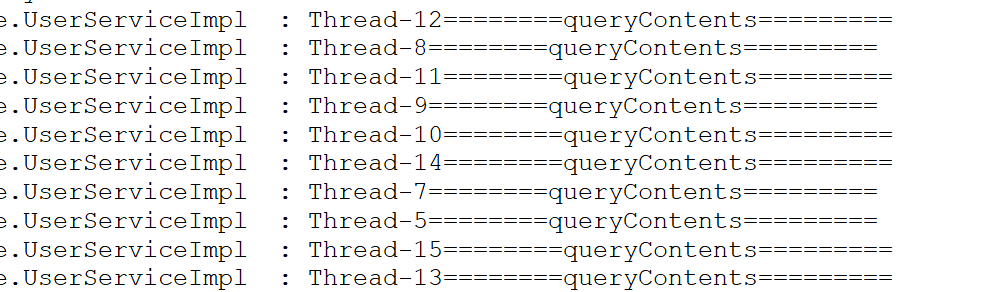


**execution.isolation.semaphore.maxConcurrentRequests**

这参数是用来控制信号量隔离级别的并发大小的。

单元测试





可以看到，单元测试中的线程和业务类中的线程是一样的，没有单独开启线程。

### Hystrix服务降级

服务降级是对服务调用过程的出现的异常的友好封装，当出现异常时，我们不希望直接把异常原样返回，所以当出现异常时我们需要对异常信息进行包装，抛一个友好的信息给前端。

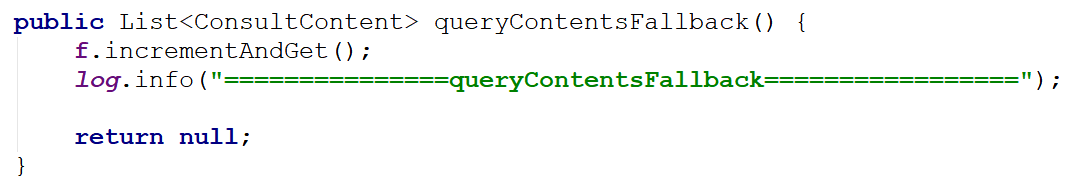
Hystrix降级的使用比较简单

代码示例



指定降级方法

定义降级方法，降级方法的返回值和业务方法的方法值要一样

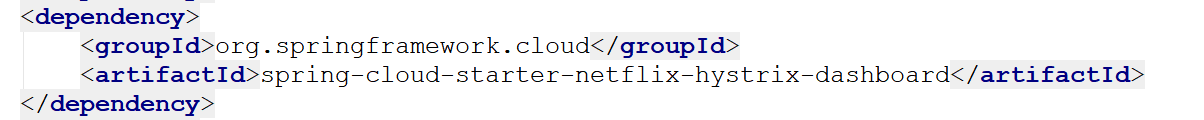


### Hystrix数据监控

Hystrix进行服务熔断时会对调用结果进行统计，比如超时数、bad请求数、降级数、异常数等等都会有统计，那么统计的数据就需要有一个界面来展示，hystrix-dashboard就是这么一个展示hystrix统计结果的服务。

Dashboard工程搭建

pom.xml



启动类

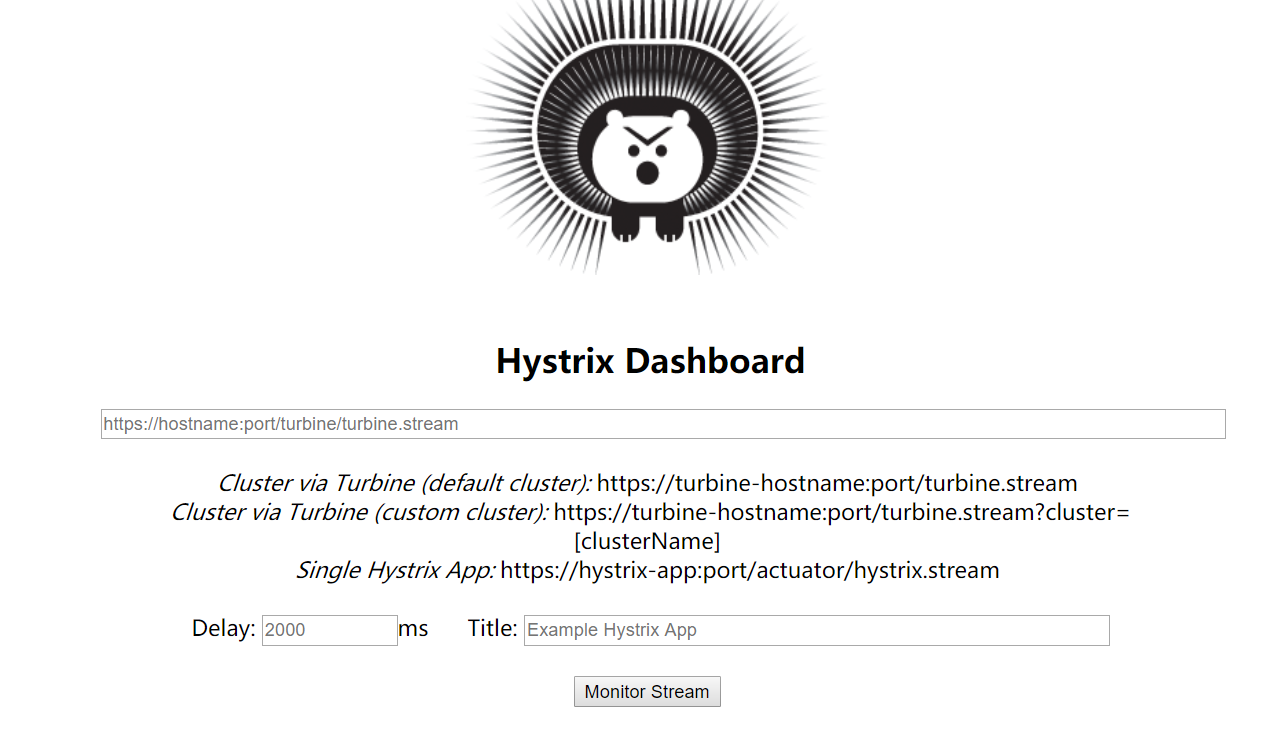


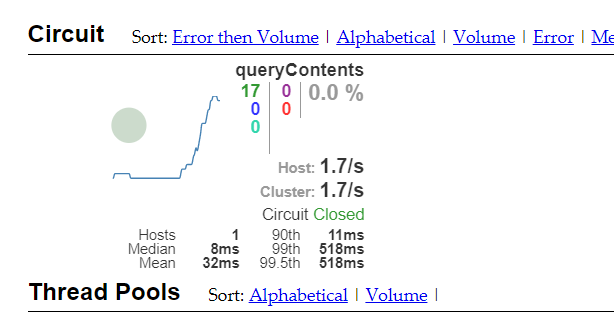
Dashboadr界面

*http://localhost:9990/hystrix*

然后在界面中输入需要监控的端点url：

*http://localhost:8083/actuator/hystrix.stream*





### Hystrix熔断

熔断就像家里的保险丝一样，家里的保险丝一旦断了，家里就没点了，家里用电器功率高了就会导致保险丝端掉。在我们springcloud领域也可以这样理解，如果并发高了就可能触发hystrix的熔断。

熔断发生的三个必要条件：

1. 有一个统计的时间周期，滚动窗口

相应的配置属性

*metrics.rollingStats.timeInMilliseconds*

默认10000毫秒

1. 请求次数必须达到一定数量

相应的配置属性

*circuitBreaker.requestVolumeThreshold*

默认20次

1. 失败率达到默认失败率

相应的配置属性

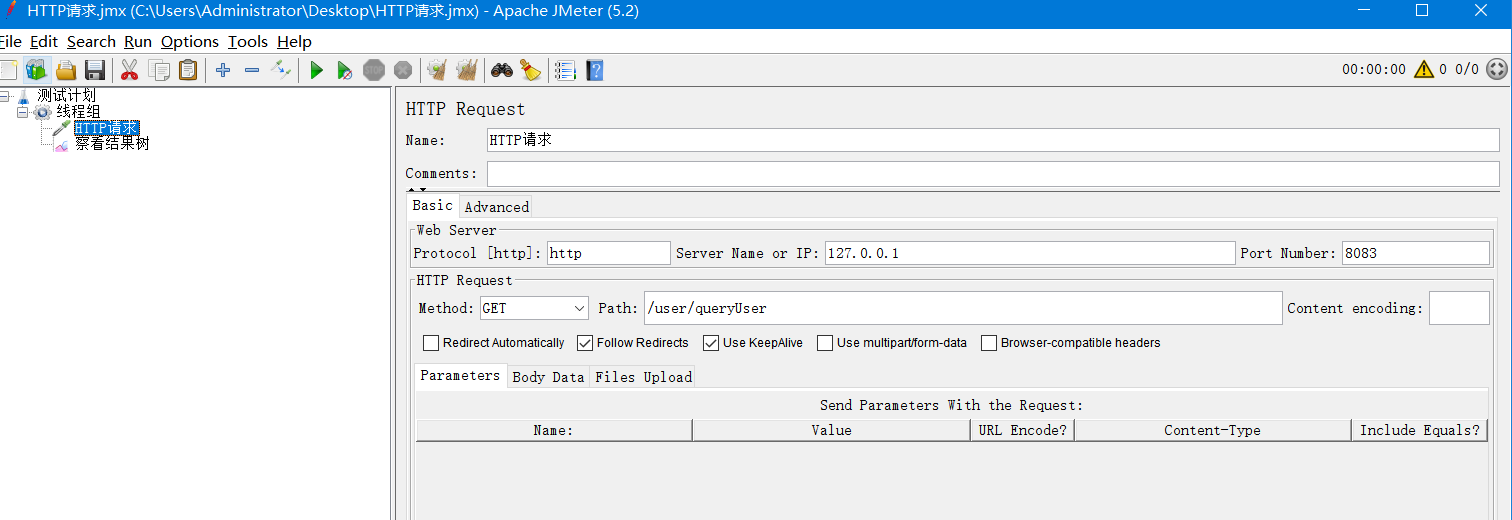
*circuitBreaker.errorThresholdPercentage*

默认50%

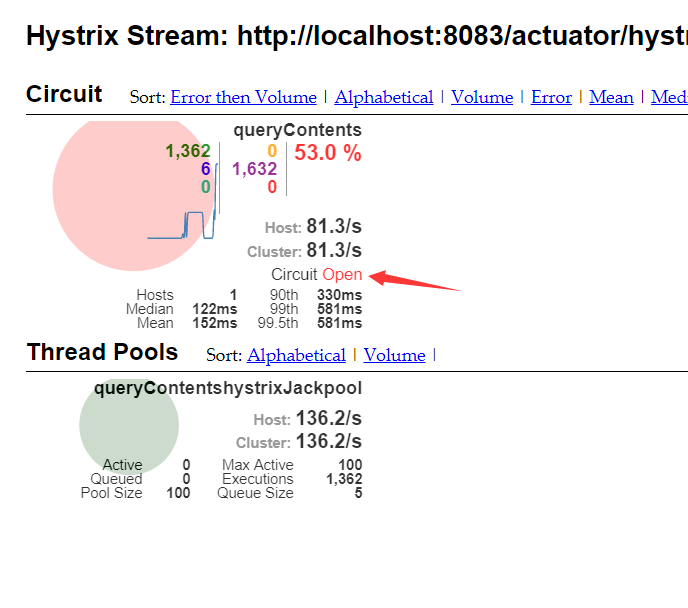
上述3个条件缺一不可，必须全部满足才能开启hystrix的熔断功能。

当我们的对一个线程池大小是100的方法压测时看看hystrix的熔断效果：

Jmeter压测：



Hystrix dashboard界面



可以看到失败率超过50%时，circuit的状态是open的。

熔断器的三个状态：

1、关闭状态

关闭状态时用户请求是可以到达服务提供方的

2、开启状态

开启状态时用户请求是不能到达服务提供方的，直接会走降级方法

3、半开状态

当hystrix熔断器开启时，过一段时间后，熔断器就会由开启状态变成半开状态。

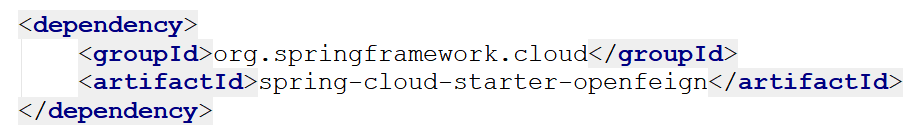
半开状态的熔断器是可以接受用户请求并把请求传递给服务提供方的，这时候如果远程调用返回成功，那么熔断器就会有半开状态变成关闭状态，反之，如果调用失败，熔断器就会有半开状态变成开启状态。

Hystrix功能建议在并发比较高的方法上使用，并不是所有方法都得使用的。

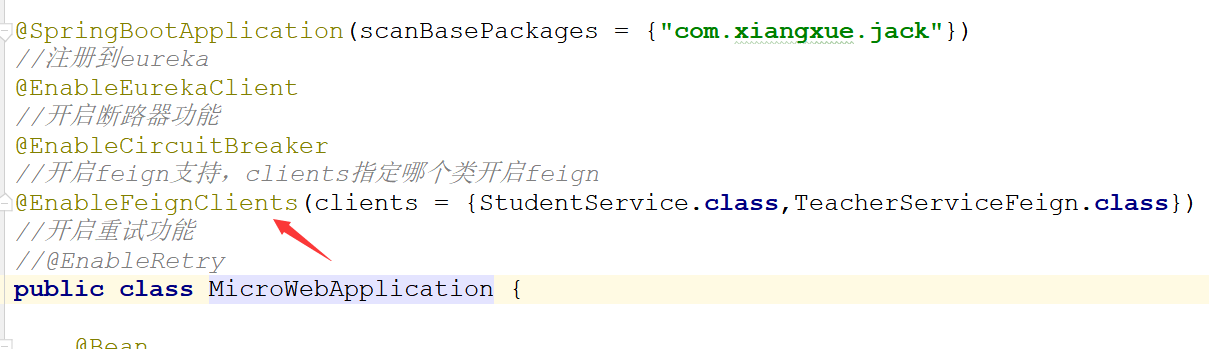
### Feign的使用

Feign是对服务端和客户端通用接口的封装，让代码可以复用做到统一管理。

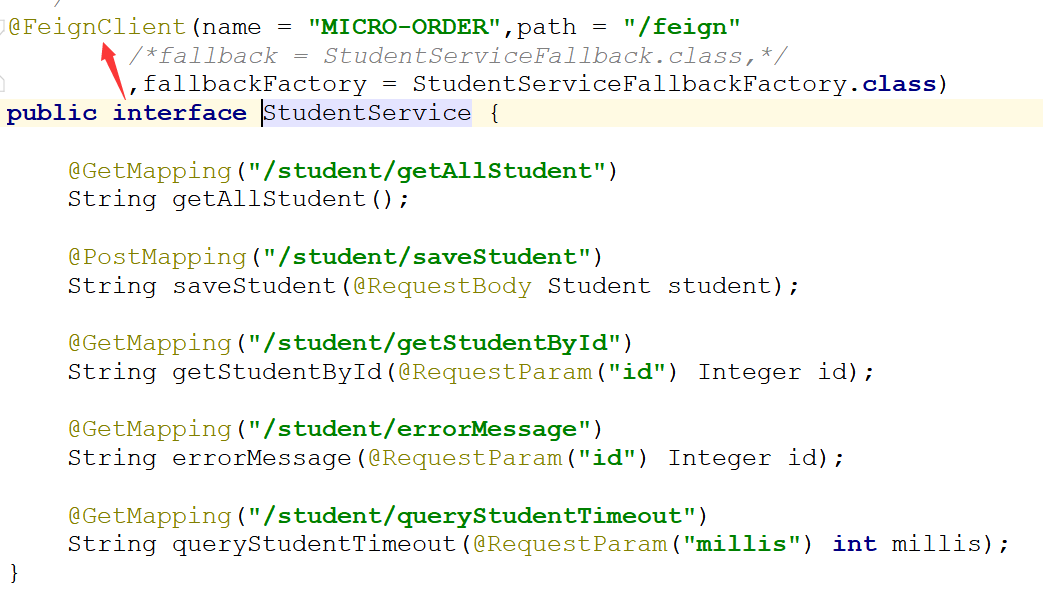
1. jar包导入



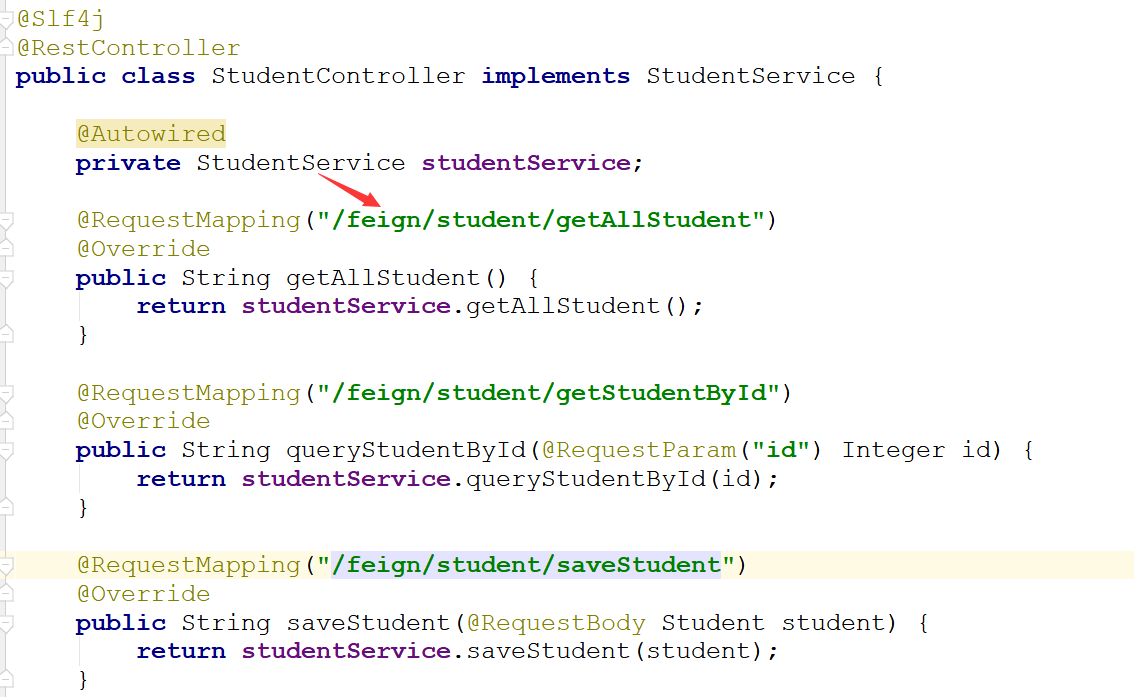
1. 启动类导入feign客户端



1. feign客户端



1. 服务端接口

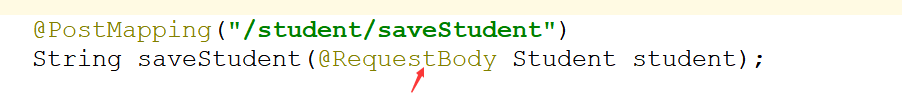


服务端接口必须定义跟feign客户端相同的url。

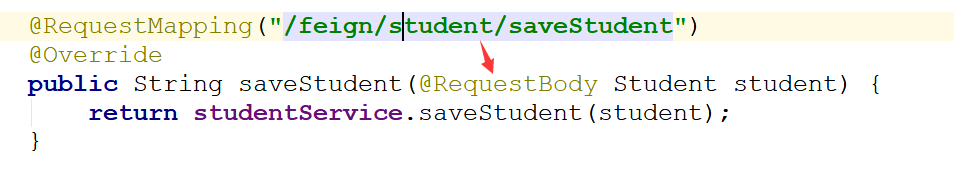
1. 参数传递

对象类型参数

调用方



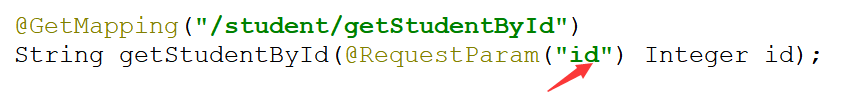
服务方



实际上传的是字符串

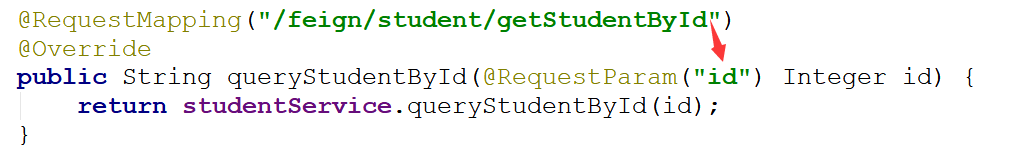
其他参数

调用方



这里必须指定参数名称

服务方



1. Feign的服务降级



这里在调用对应feign客户端方法出现异常了，就会回调到create方法中，最终会回调到对应的客户端方法中。

1. Feign的异常过滤器

这个过滤器是对异常信息的再封装，把feign的异常信息封装成我们系统的通用异常对象



过滤器把异常返回后，feign前面定义的降级方法就会调到create方法。

Feign 我不建议大家使用，流程简单并发不高的方法可以用一用。

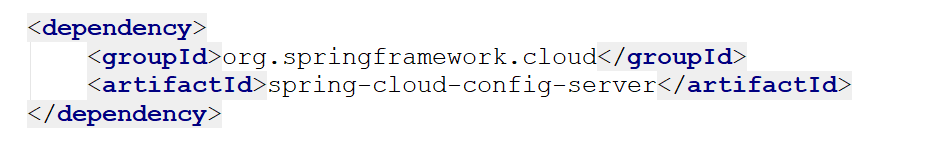
### 分布式配置中心

分布式配置中心解决了什么问题

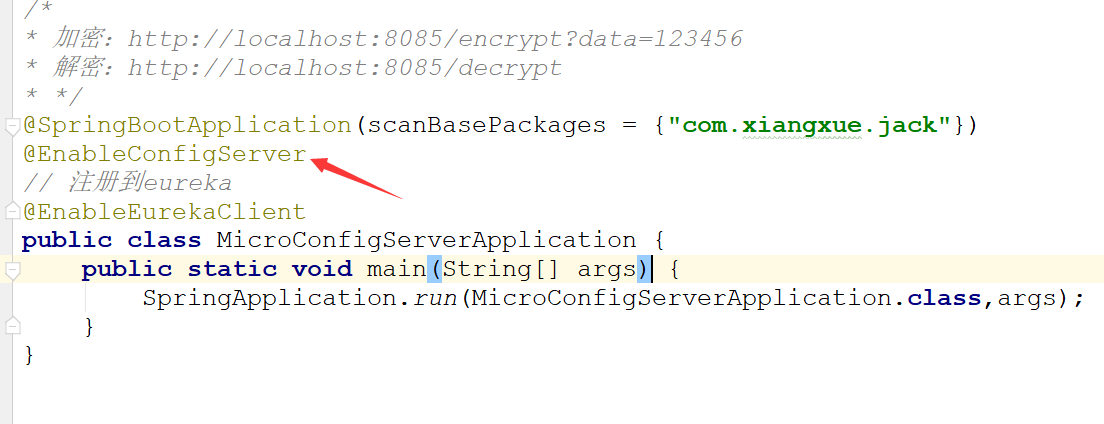
1. 抽取出各模块公共的部分，做到一处修改各处生效的目标
2. 做到系统的高可用，修改了配置文件后可用在个模块动态刷新，不需要重启服务器

Springcloud配置中心服务端搭建

1. jar包导入



2、启动类



3、分布式配置中心配置规则

分布式配置中心服务端是需要远程连接代码仓库的，比如GitHub，gitlab等，把需要管理的配置文件放到代码仓库中管理，所以服务端就需要有连接代码仓库的配置：

**spring.cloud.config.server.git.uri**=**https://github.com/zg-jack/zg-config-repo  
spring.cloud.config.server.git.search-paths**=**config-repo  
spring.cloud.config.server.git.username**=**zg-jack  
spring.cloud.config.server.git.password**=**zg0001jack**

4、客户端使用配置中心

客户端只需要指定连接的服务端就行了，从服务端拉取配置信息

1. jar包依赖



1. Properties配置

**spring.cloud.config.profile**=**dev  
spring.cloud.config.label**=**master***#这种配置是configserver还单机情况，直接连接这个单机服务就行***spring.cloud.config.uri**=**http://localhost:8085/**

5、客户端快速失败和重试

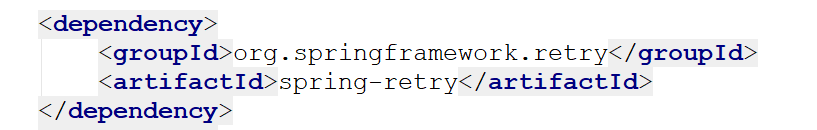
当客户端连服务端失败时，客户端就快速失败，不进行加载其他的spring容器

快速失败

*#如果连接不上获取配置有问题，快速响应失败***spring.cloud.config.fail-fast**=**true**

客户端也有重试功能，连不上服务端是有重试机制

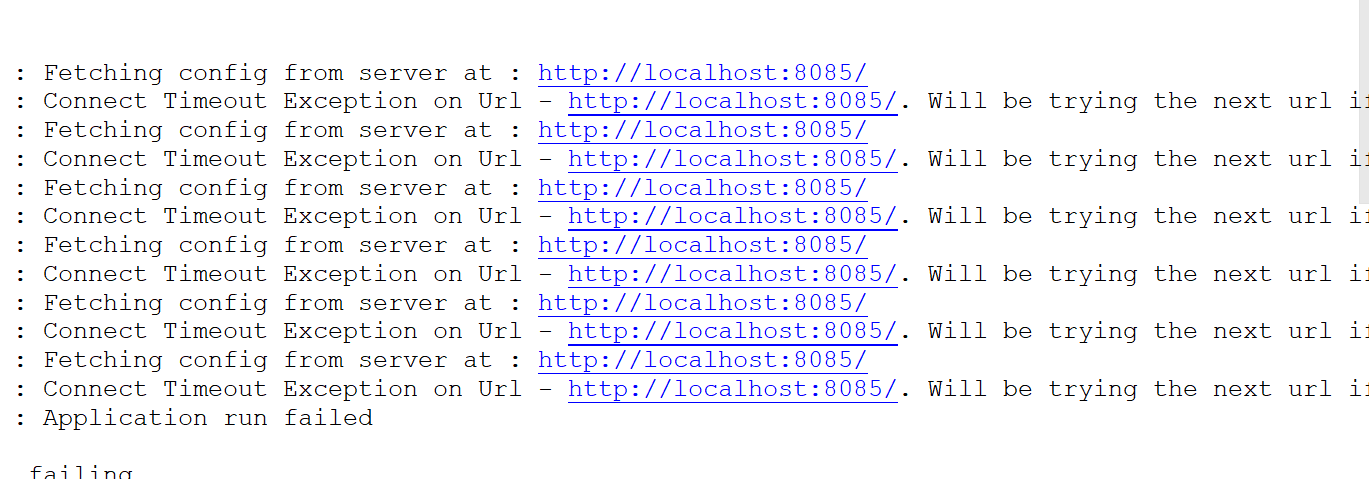
重试功能jar包导入



重试配置

*#默认重试的间隔时间，默认1000ms***spring.cloud.config.retry.multiplier**=**1000***#下一间隔时间的乘数，默认是1.1  
#spring.cloud.config.retry.initial-interval=1.1  
#最大间隔时间，最大2000ms***spring.cloud.config.retry.max-interval**=**2000***#最大重试次数，默认6次***spring.cloud.config.retry.max-attempts**=**6**

重试效果



6、配置信息的加密

在配置中心中，有些信息是比较敏感的，比如密码信息，在配置密码信息的时候有必要对密码信息加密以免密码信息泄露，springcloud配置中心也支持配置信息加密的，这里一RSA非对称加密举例。

1. 本地生成秘钥对

cd到jdk的keytool目录:D:\Program Files\Java\jdk1.8.0\_92\jre\bin

里面有一个keytool.exe可执行文件

1. 执行指令生成秘钥文件

keytool -genkeypair -alias config-server -keyalg RSA -keystore config-server.keystore -validity 365

指令执行成功后会在bin目录生成一个config-server.keystore文件，把该文件copy到配置中心服务工程中resources目录下。

1. 服务端工程配置

Properties配置文件

添加秘钥配置

*#加密配置***encrypt.key-store.location**=**config-server.keystore  
encrypt.key-store.alias**=**config-server  
encrypt.key-store.password**=**123456  
encrypt.key-store.secret**=**123456**

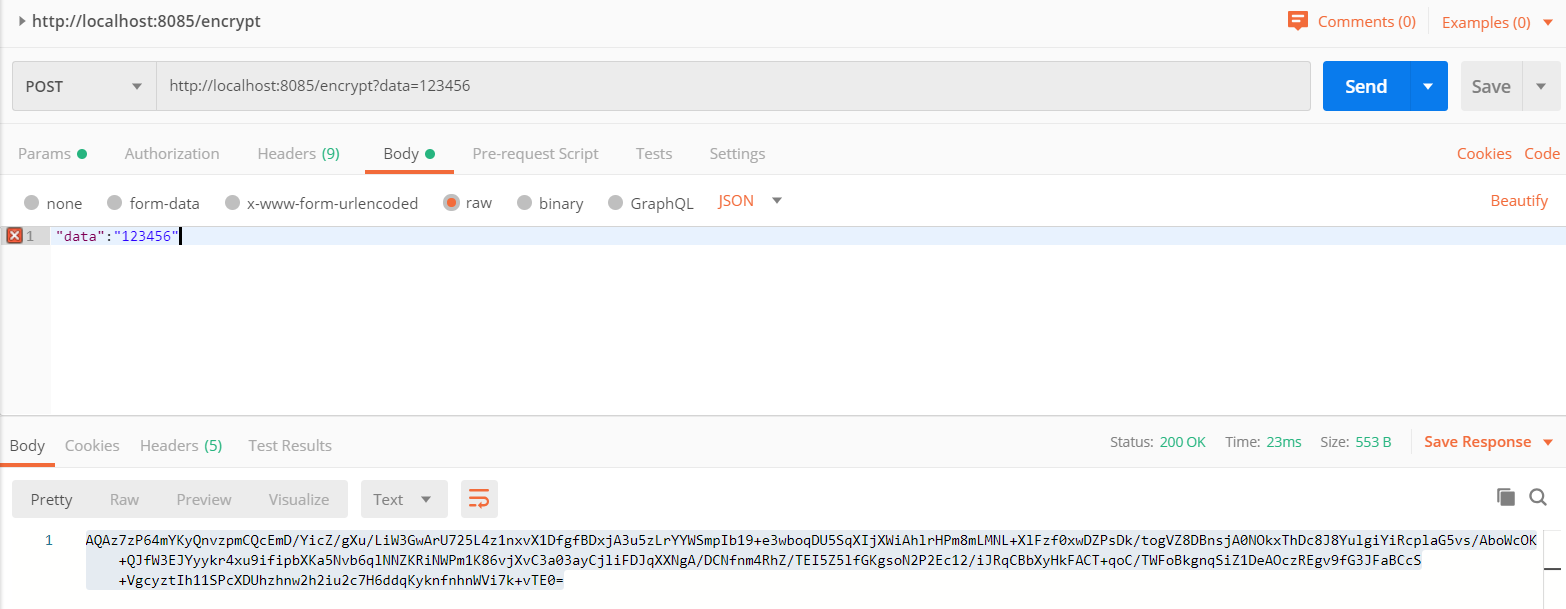
pom中添加静态文件扫描，让能够扫描到.keystore文件



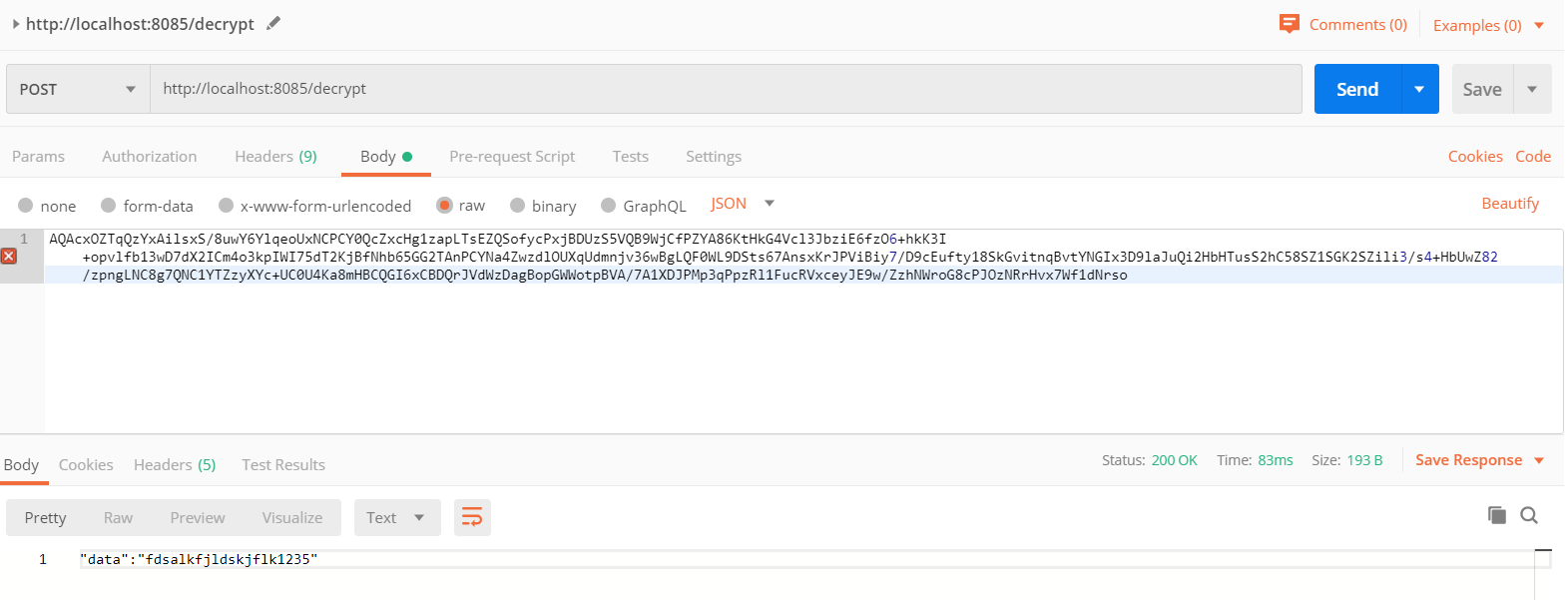
1. 密码加密和解密接口

在服务端中有提供对信息加密和解密接口的

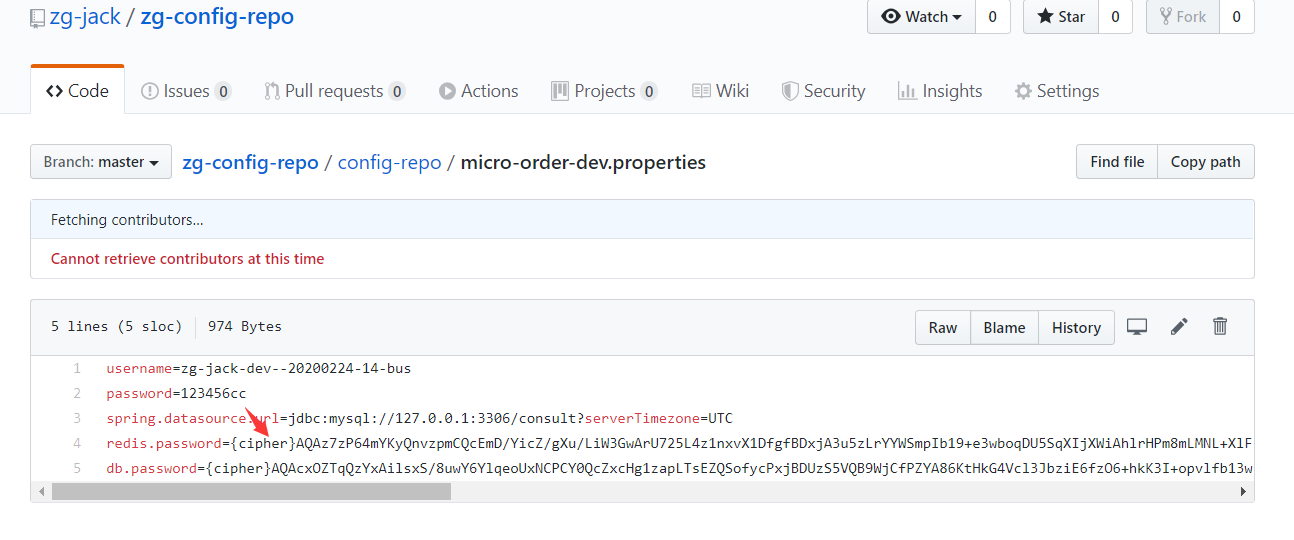
加密接口：<http://localhost:8085/encrypt?data=123456，post请求>



解密接口：http://localhost:8085/decrypt，post请求



1. 代码仓库密文配置



密文前面一定要加上{cipher}标识这个是密文配置，需要服务端来解密的。

1. 配置动态加载刷新

这个是一个革命性的功能，在运行期修改配置文件后，我们通过这个动态刷新功能可以不重启服务器，这样我们系统理论上可以保证7\*24小时的工作

1. Environment的动态刷新

动态刷新其实要有一个契机，其实这个契机就是手动调用刷新接口，如果你想刷新哪台主机的配置，就调用哪台注解的刷新接口

刷新接口为：http://localhost:8088/actuator/refresh

2、@Value注入的属性动态刷新

其实前面在调用刷新接口后，@Value注入的属性是没有刷新的还是老的配置，这个也好理解，@Value注入的属性是项目启动就已经定了的。如果要使@Value属性也刷新，就必须要在类上面加上：@RefreshScope注解。

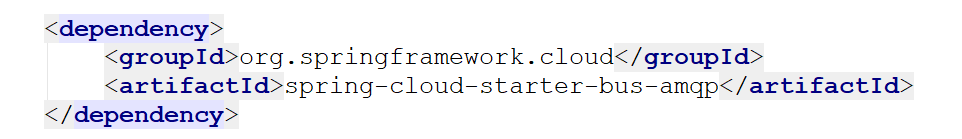
但是调用每台主机的刷新接口显然太麻烦了，如果需要刷新的集群机器有几百台，是不是就需要手动调用几百次呢，这几乎是一个不能完成的工作量。

Springcloud中也提供了消息总线的东西，借助mq来完成消息的广播，当需要刷新时我们就只要调用一次刷新接口即可。

1. 消息总线

消息总线其实很简单，就是为了解决一点刷新的功能，在一个点调用请求刷新接口，然后所有的在消息总线中的端点都能接到刷新的消息，所有我们必须把每一个端点都拉入到消息总线中来。

1. jar导入



1. properties配置

其实就是连接mq的配置

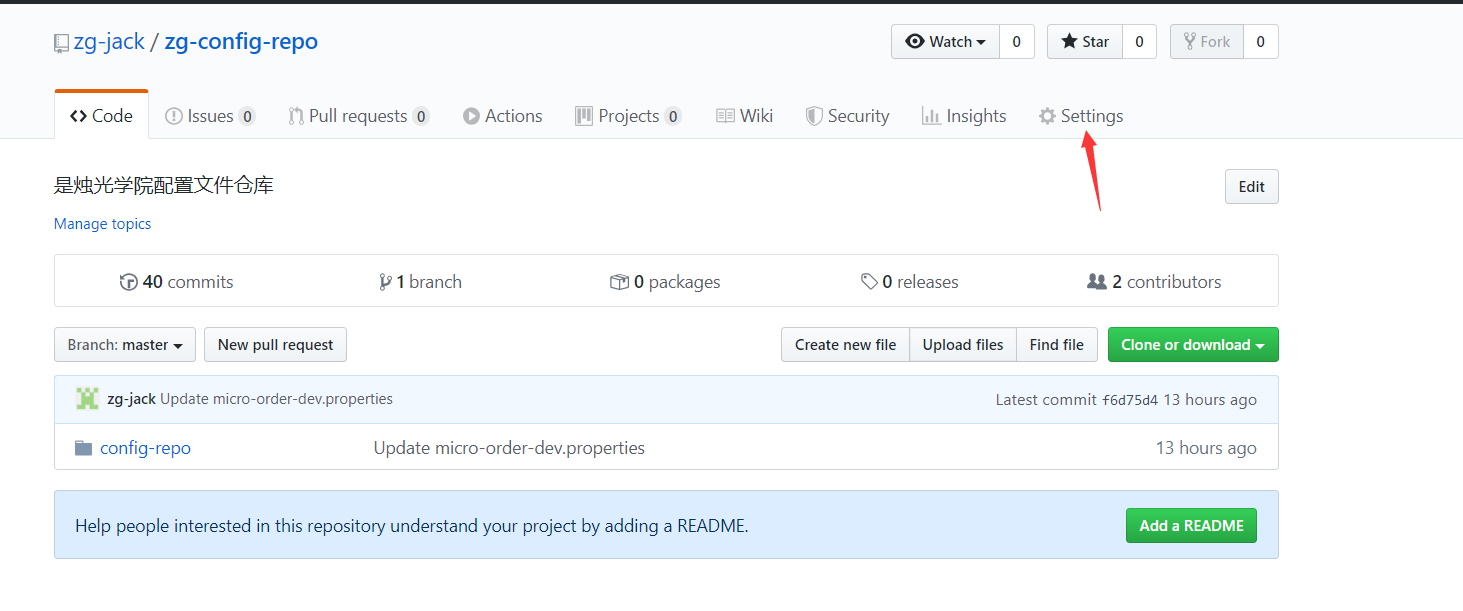
**spring.rabbitmq.host**=**192.168.67.139  
spring.rabbitmq.port**=**5672  
spring.rabbitmq.username**=**admin  
spring.rabbitmq.password**=**admin***# 刷新配置url http://localhost:8081/actuator/bus-refresh***spring.cloud.bus.refresh.enabled**=**true  
spring.cloud.bus.trace.enabled**=**true**

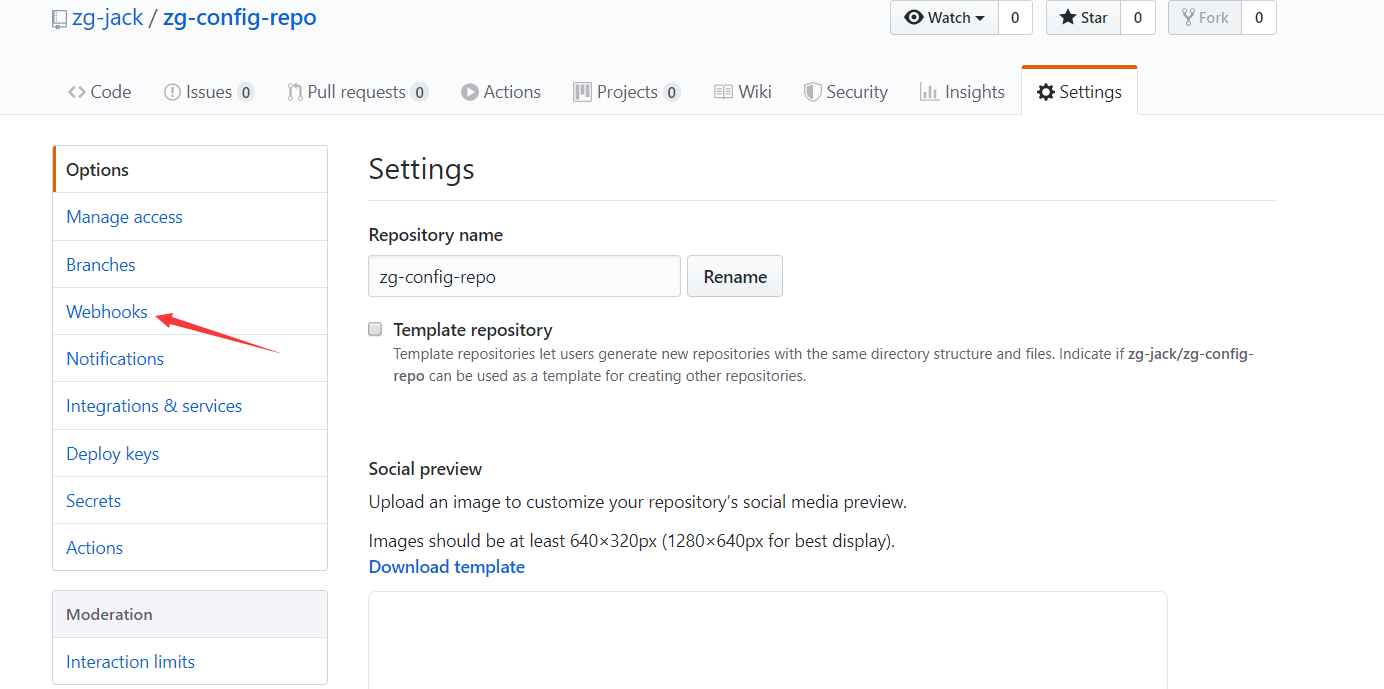
通过这两步就已经完成了拉入消息总线的工作了。

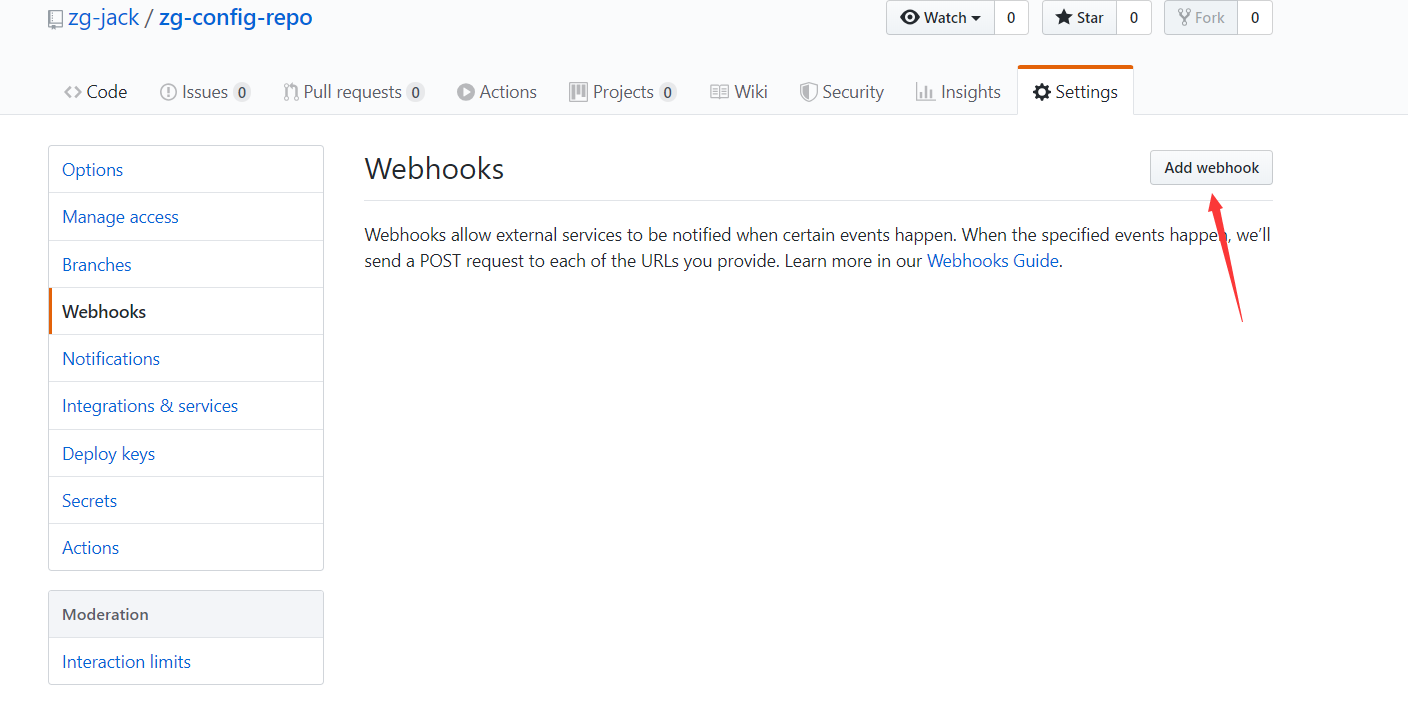
如果要刷新配置，就只要调用任意一个消息总线端点调用刷新接口即可，其他的端点就会收到刷新配置的消息。

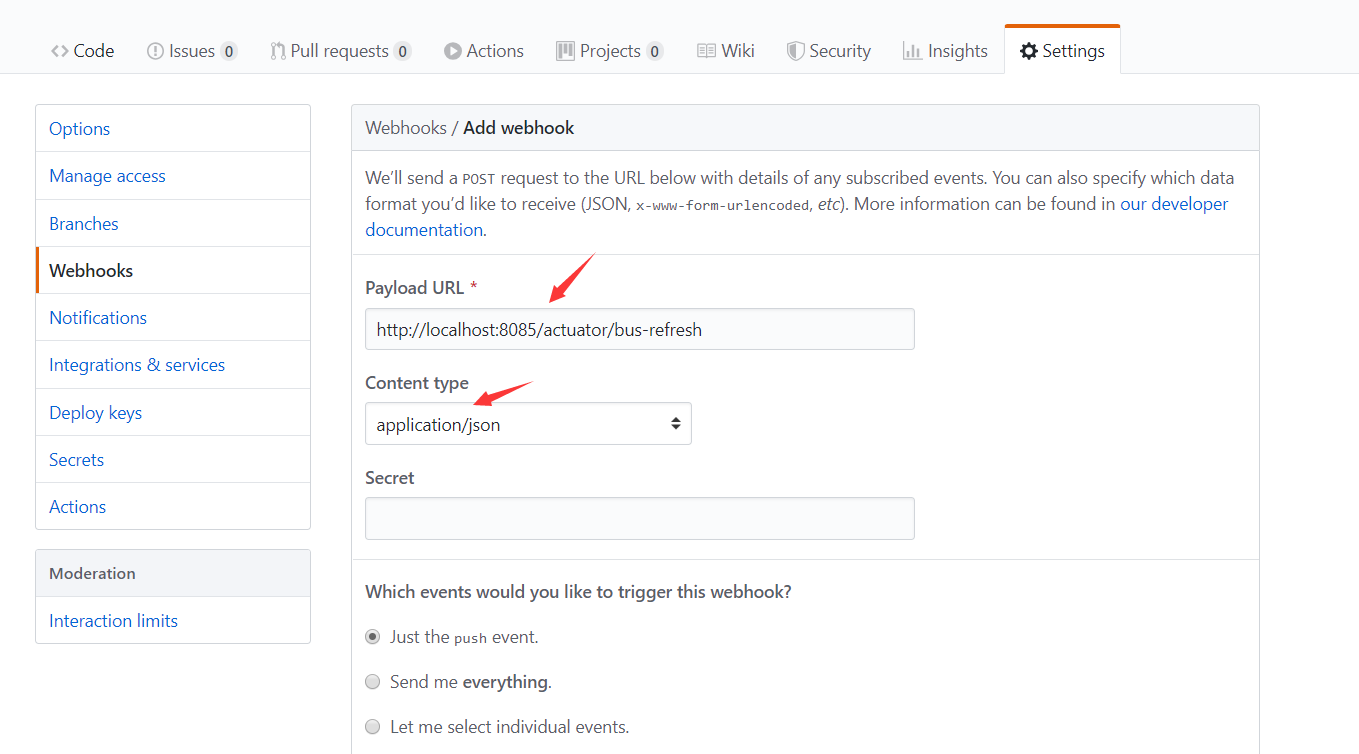
刷新接口：<http://localhost:8085/actuator/bus-refresh>

这个接口也可以配置到GitHub中，只要GitHub有代码变更，就会调用这个接口







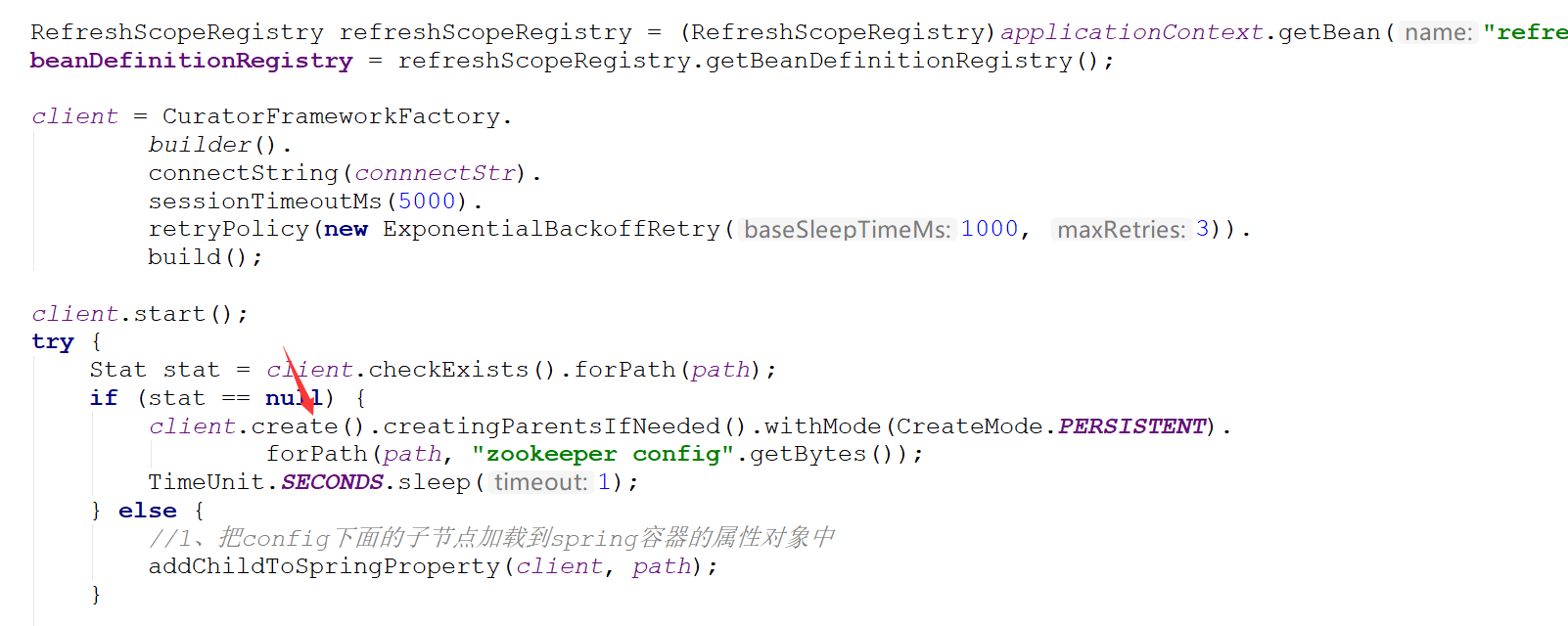


当然这里配置的URL是你生产环境的外网地址，这样当你GitHub中修改了配置后，就会主动调用消息总线的刷新接口。

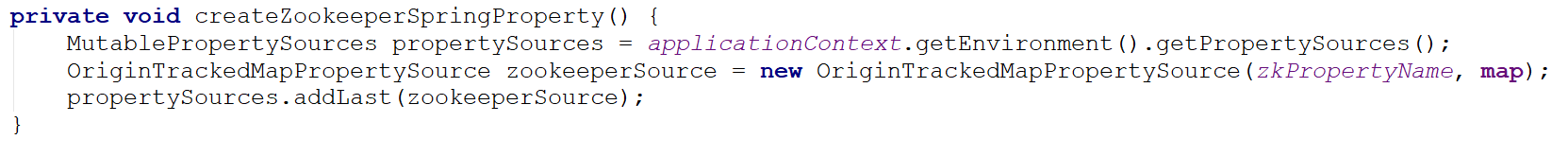
消息总线弊端就是太重了，一个集群通知刷新配置功能，还得用一个rabbitmq来做，如果项目中根本用不到rabbitmq呢？就加大了项目负担，加大了维护成本。差评。

### 自定义分布式配置中心

1. zookeeper创建节点及事件监听

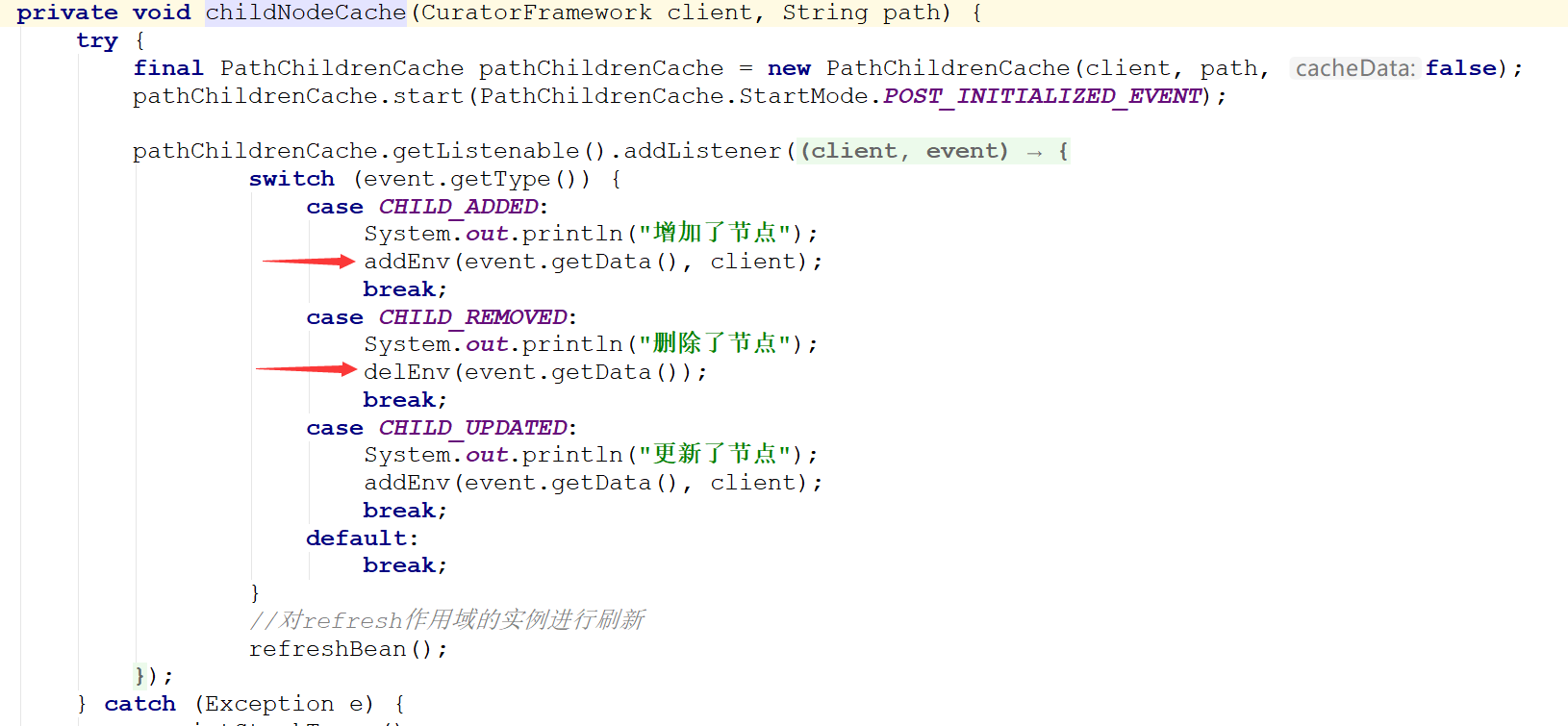


1. 如果zookeeper里面有配置了把配置加载到spring中

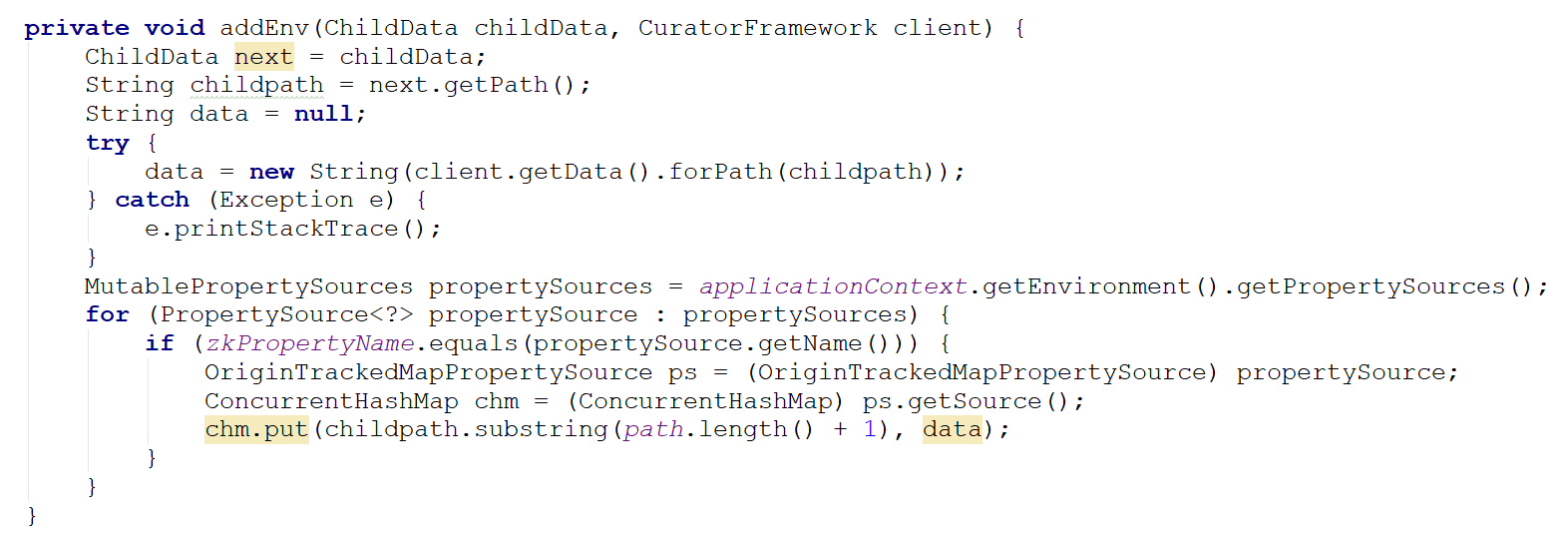


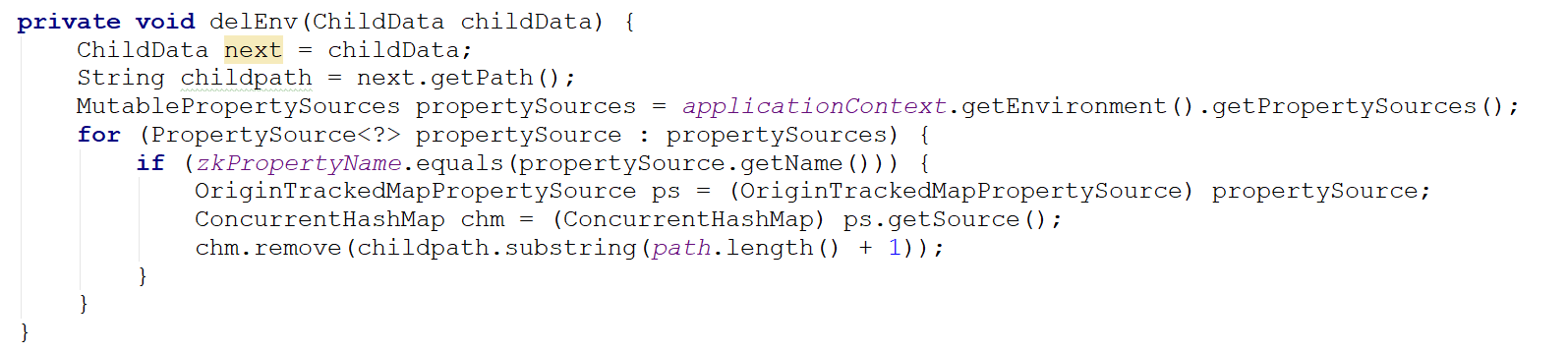


1. 为配置节点添加事件

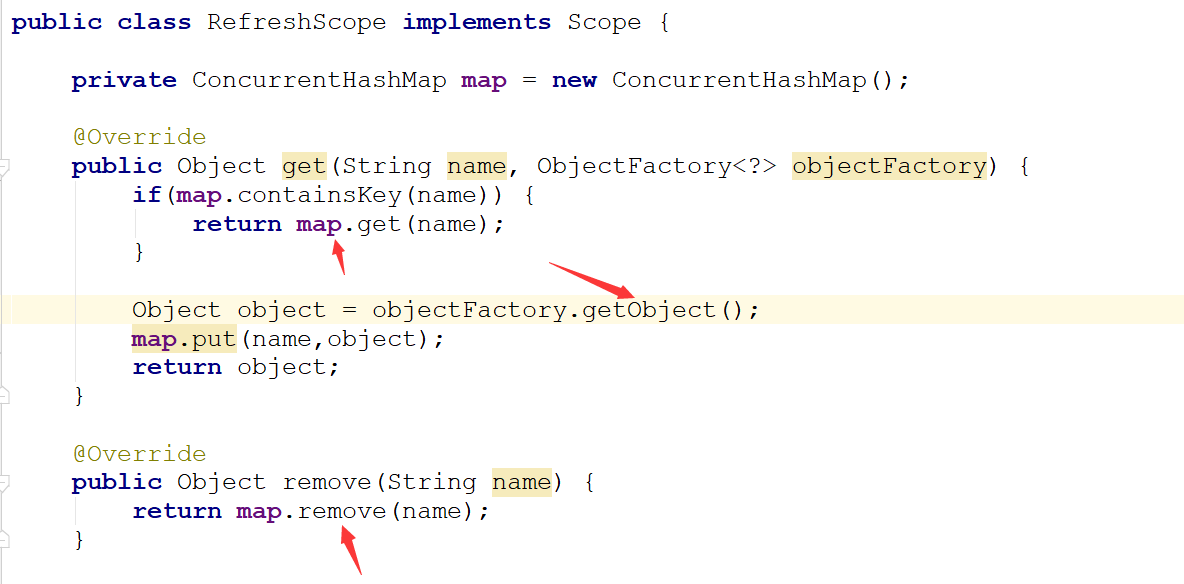


当有节点新增、修改、删除时触发对应的事件，事件方法：



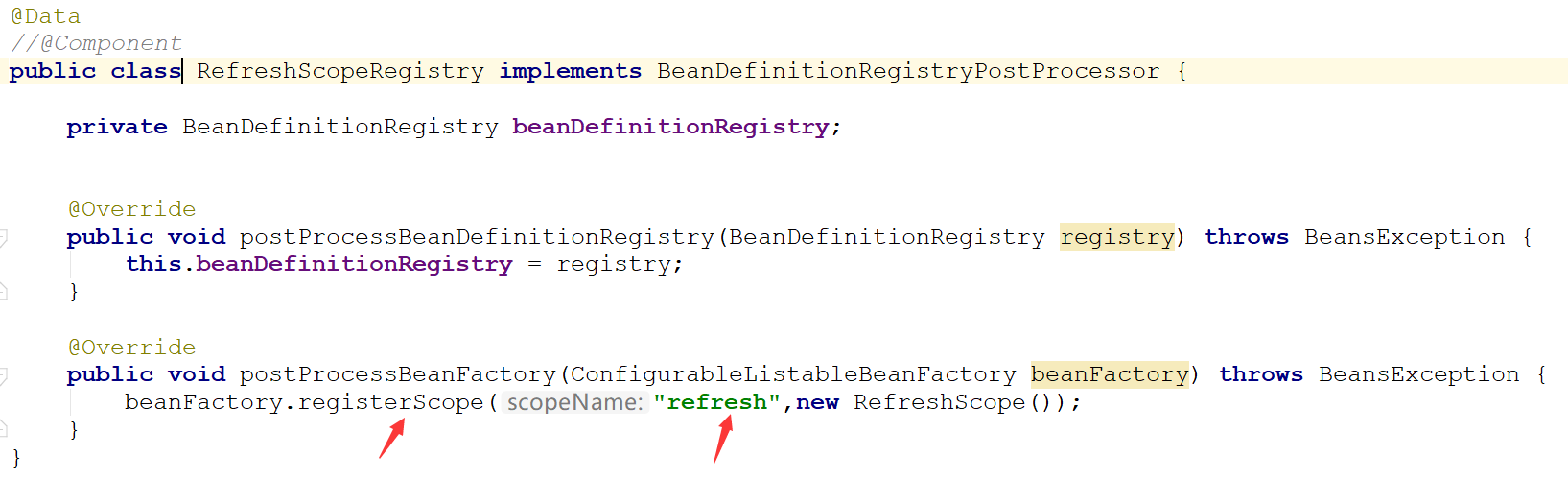


1. 对象重新创建触发@Value的依赖注入
2. 自定义Scope，自己来管理bean

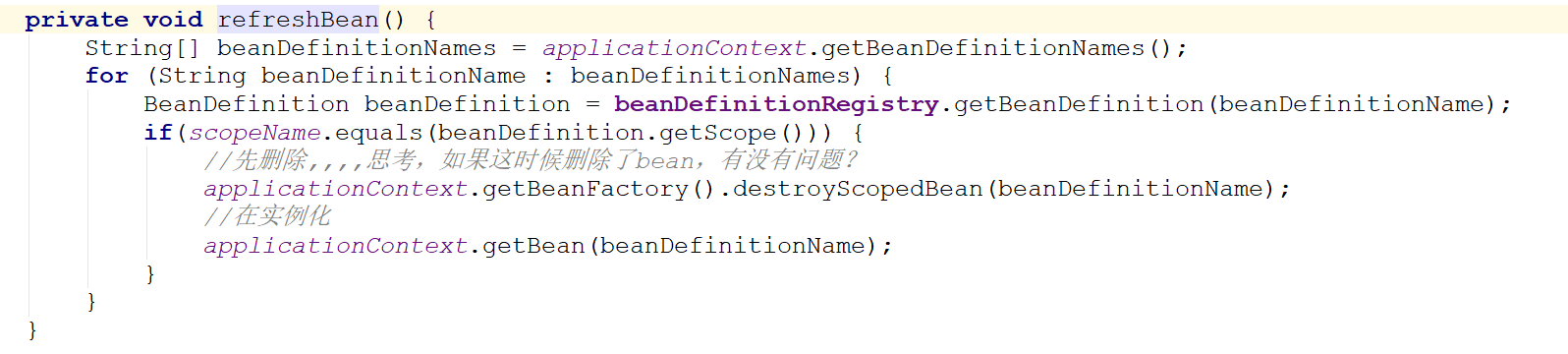


如果缓存中有则直接返回，如果缓存没用则调用getObject方法创建对象

1. 把自定义作用域注册到spring中



1. 在有节点更新修改时重新创建节点



这里就会把自己管理的bean从缓存中删除，并且调用getBean创建新的对象

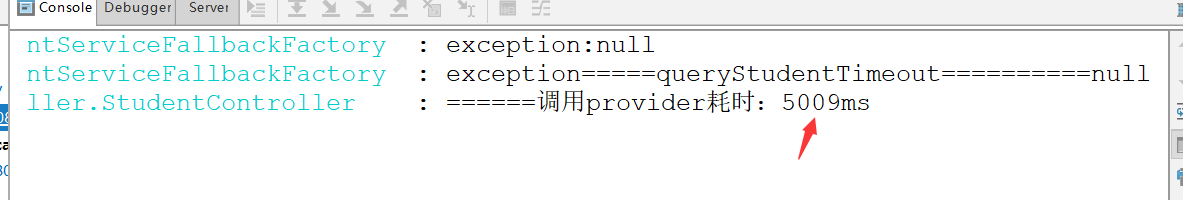
### 超时配置



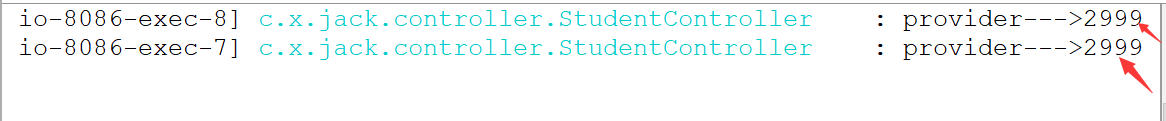
超时时间配置，如果hystrix超时时间大于ribbon超时时间，那么就以hystrix超时时间为准。

http://localhost:8083/student/timeOut?millis=2999

总共耗时：



Ribbon重试



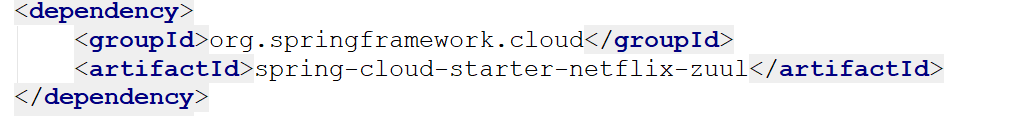
这种请求是hystrix超时了，触发了ribbon的重试，ribbon调用还没结束，但是超过了hystrix的超时，所以hystrix拿不到异常。

### Zuul服务网关

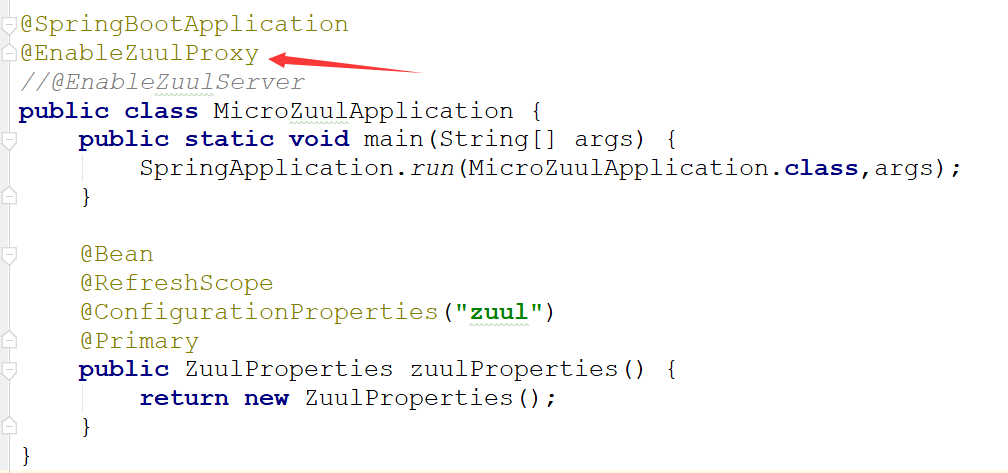
Zuul是分布式springcloud项目的流量入口，理论上所有进入到微服务系统的请求都要经过zuul来过滤和路由。

1. zuul服务网关的搭建

Jar包导入



启动类加注解



1. 配置详解

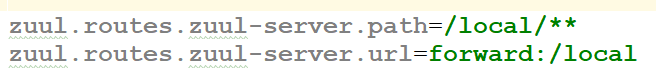
配置拦截的url和该url路由的服务



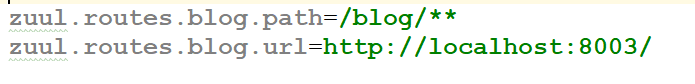
配置敏感请求头过滤



Zuul本地跳转



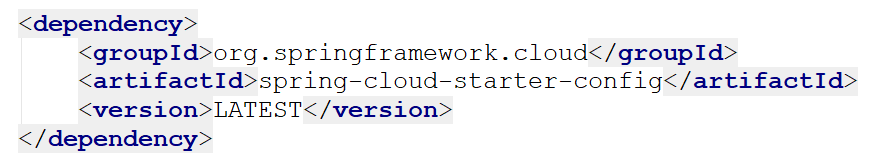
Zuul跳转到指定地址



1. 路由动态刷新

路由动态刷新其实就是分布式配置中心的思路，就是路由配置规则我们可以在项目启动后，通过修改远程GitHub上的配置，让配置在运行时生效，不需要重启机器。所以就需要加入配置中心的配置。

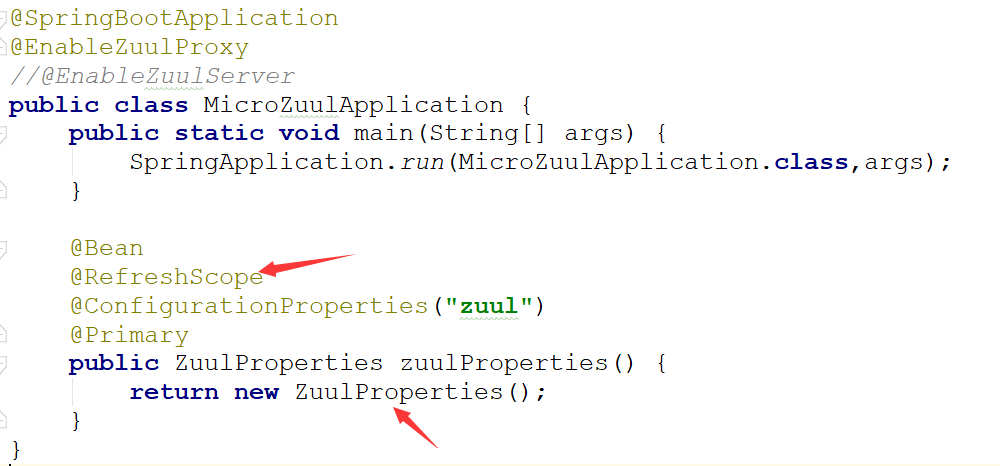
配置中心客户端jar



配置信息

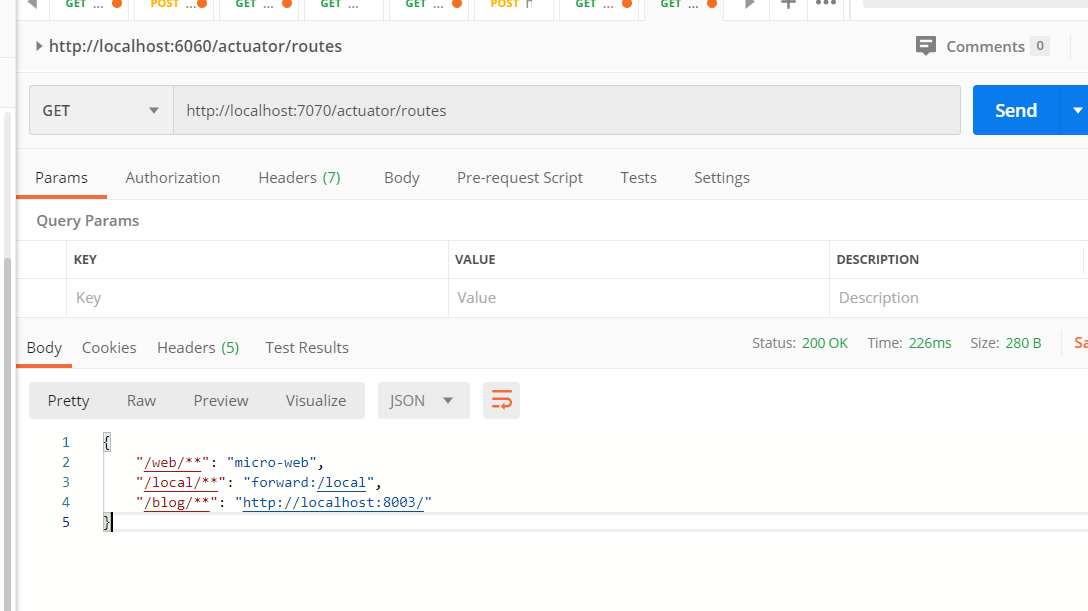


配置动态刷新类



获取路由规则的接口

<http://localhost:7070/actuator/routes>



1. zuul过滤器

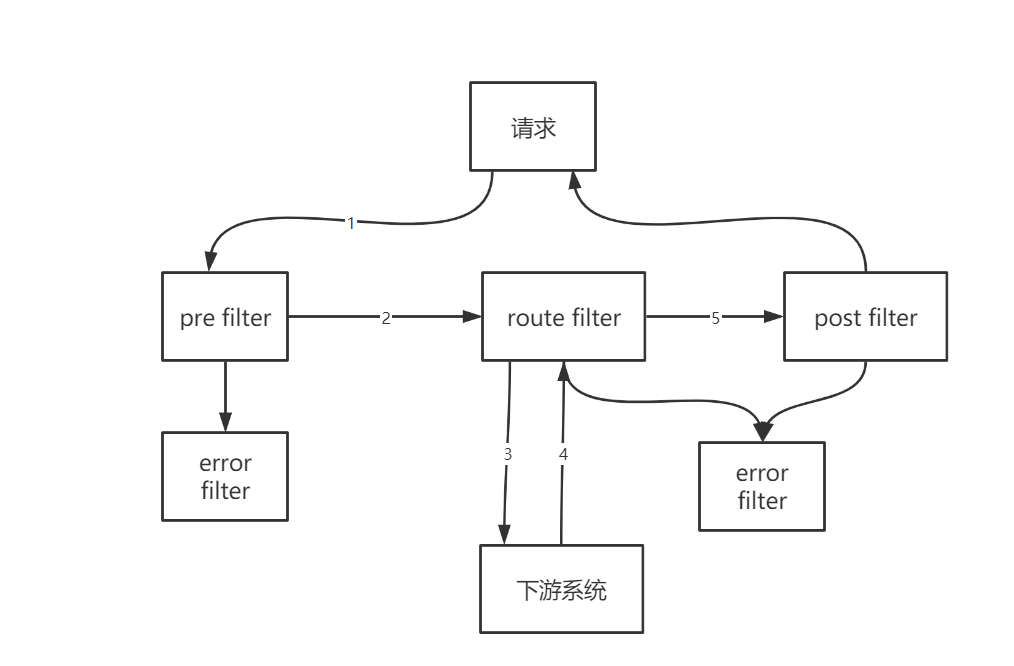
Zuul大部分功能都是通过过滤器来实现的，Zuul定义了4种标准的过滤器类型，这些过滤器类型对应于请求的典型生命周期。  
a、pre: 这种过滤器在请求被路由之前调用。可利用这种过滤器实现身份验证、在集群中选择请求的微服务，记录调试信息等。

b、routing: 这种过滤器将请求路由到微服务。这种过滤器用于构建发送给微服务的请求，并使用apache httpclient或netflix ribbon请求微服务。

c、post: 这种过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准的http header、收集统计信息和指标、将响应从微服务发送给客户端等。

e、error: 在其他阶段发送错误时执行该过滤器。

执行流程



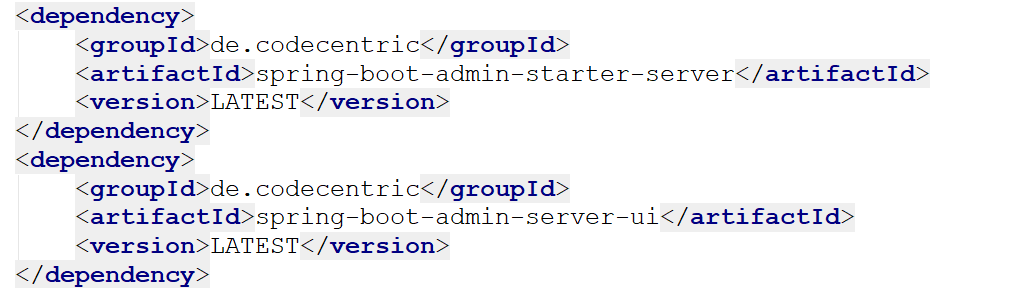
### Springcloud admin

Springcloud admin是基于

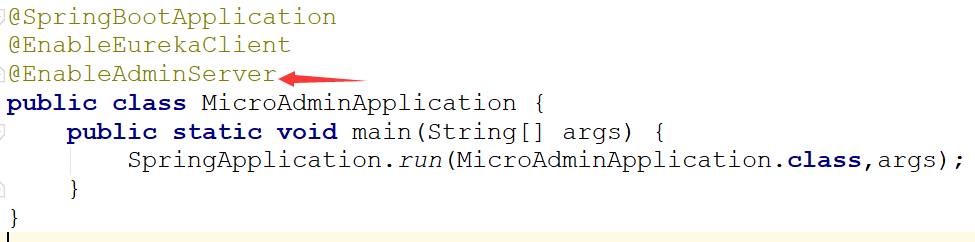
<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-actuator</**artifactId**>  
</**dependency**>

把actuator负责统计数据，admin是根据统计出来的数据来进行展示的，可以很好的监控整个微服务系统中的实例运行情况信息。

1. jar导入



1. 启动类加注解



1. 安全配置

