注:

JDK:1.8

IDE:IDEA

参考书籍:Spring+Cloud微服务实战

# 一.Spring Cloud 整体简介

Spring Cloud 是基于Spring Boot 实现的微服务架构开发工具,它为微服务架构中涉及的配置管理、服务治理、断路器、智能路由、微代理、控制总线、全局锁、决策精选、分布式会话和集群状态管理等操作提供了一种简单的开发方式。

Spring Cloud包含了多个子项目：

1. Spring Cloud Config：配置管理工具，支持使用Git存储配置内容，可以使用它实现应用配置的外部化存储，并支持客户端信息刷新、加密/解密配置内容.
2. Spring Cloud Netflix: 核心组件,对多个Netflix OSS开源套件进行整合.
   1. Eureka:服务治理组件,包含服务注册中心,服务注册与发现机制的实现.
   2. Hystrix:容错管理组件,实现断路器模式,帮助服务依赖中出现的延迟,为故障提供强大的容错能力.
   3. Ribbon:客户端负载均衡的服务调用组件.
   4. Feign:基于Ribbon和Hystrix的声明式服务调用组件.
   5. Zuul:网关组件,提供智能路由、访问过滤等功能。
   6. Archaius:外部化配置组件
3. Spring Cloud Bus : 事件,消息总线,用于传播集群中的状态变化或事件,以触发后续的处理,比如用来动态刷新配置等.
4. Spring Cloud Ribbon:实现客户端负载均衡.
5. Spring Cloud Hystrix:实现了断路器、线程隔离、服务降级、请求缓存、请求合并已经服务监控等功能。
6. Spring Cloud Feign:基于Ribbon实现负载均衡,由于Ribbon都会利用RestTemplate的请求拦截来实现对依赖服务的调用,然而一个服务依赖的调用可能不止一个地方,然后Feign对此做了进一步封装,是用接口对远端服务进行定义.通过定义的接口来完成对依赖服务的调用.针对接口的定义集成了Spring MVC和JAXRS注解.
7. Spring Cloud Cluster:针对Zookeeper、Redis、Hazelcast、Consul的选举算法和通用状态模式的实现。
8. Spirng Cloud Cloudfoundry：与Pivotal Cloudfoundry的整合支持
9. Spring Cloud Consul：服务发现与配置管理工具。
10. Spring Cloud Stream：通过Redis、Rabbit或者Kafka实现的消费微服务。可以通过简单的声明式模型来发送和接收消息.
11. Spring Cloud AWS:用于简化整合Amazon Web Service的组件.
12. Spring Cloud Security:安全工具包,提供Zuul代理中对OAuth2客户端请求的中继器.
13. Spring Cloud Sleuth:应用的分布式跟踪实现,可以完美整合Zipkin.
14. Spring Cloud Cli:用于在Groovy中快速创建Spring Cloud 应用的Cli插件.

# 二.Eureka

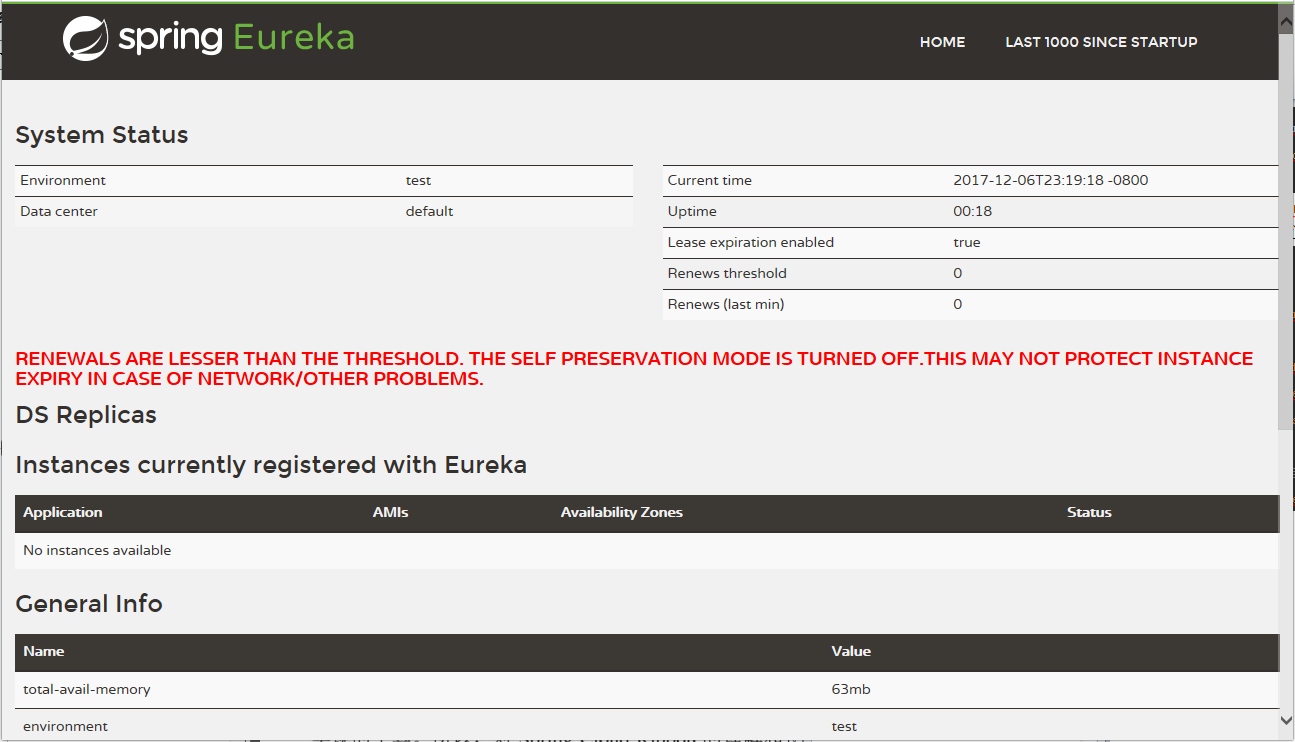
Eureka是Spring Cloud Netflix微服务套件中的一部分,它基于Netflix Eureka做了二次封装,主要负责完成微服务架构中的服务治理功能.Eureka的依赖包括spring-cloud-starter-eureka-server和spring-cloud-starter-eureka，其中server是注册中心的依赖。其中server的依赖是注册中心的依赖，两个依赖都会自动引用通用的Spring框架（如Spring.boot.web），并且都会引用Ribbon用来实现对注册中心服务的客户端负载调用.其中注册中页面的模板引擎使用的freemarker.

1. 引入依赖配置

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-eureka-server</artifactId>  
</dependency>

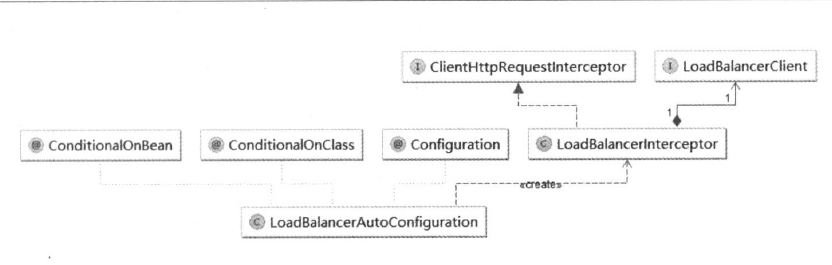
1. 启动注解@EnableEurekaServer
2. 修改注册中心的配置

#配置eureka  
#当前应用为注册中心,不用向注册中心注册自己  
eureka.client.register-with-eureka=false  
#当前应用是注册中心,所以它不需要去检索服务  
eureka.client.fetch-registry=false  
eureka.instance.hostname=localhost  
eureka.client.serviceUrl.defaultZone=http://${eureka.instance.hostname}:${server.port}/eureka/  
#关闭eureka的自我保护  
eureka.server.enable-self-preservation=false

1. 启动注册中心

# 三.Ribbon

Ribbon是基于HTTP和TCP的客户端负载均衡工具，它基于Netflit Ribbon实现，通过Spring Cloud 的封装，可以让我们轻松地讲面向服务的REST模板请求自动转换成客户端负载均衡的服务调用。



Ribbon实现负载均衡的原理,它通过LoadBalancerInterceptor拦截器对RestTemplate的请求进行拦截,并利用Spring Cloud的负载均衡器LoadBalancerClient将以逻辑服务名为host的URI转换成具体的服务实例地址的过程(所以在使用Ribbon的时候必须为RestTemplate Bean添加@LoadBalanced注解,从而让拦截器生效).同时通过分析LoadBalancerClient的Ribbon实现RibbonLoadBalancerClient,可以知道在使用Ribbon实现负载均衡器的时候,实际使用的还是Ribbon中定义的ILoadBalancer接口的实现,自动化配置会采用ZoneAwareLoadBalancer的实例来实现客户端负载均衡.

在Spring Cloud的微服务架构中使用客户端负载均衡调用需要两步:

1. 服务提供者只需要启动多个服务事例并且注册到一个注册中心或者多个相关联的服务注册中心.
2. 服务消费者直接通过调用被@LoadBalanced注解修饰过的RestTemplate来实现面向服务的接口调用.

# 四.Feign

feign是对Robbin的进一步封装,形成一种声明式、模板化的HTTP客户端.将远端服务通过接口定义注册成本地Bean,并且能够使用Spring mvc, jax-rs的注解来对远端服务的描述进而实现远端服务本地化(类似于CXF/wsimport将远端的webService注册成本地方法)

## 1.使用

1.在启动类上添加注解@EnableFeignClients

2.在Service层新建接口,使用注解@FeignClient("EUREKACLIENT-01")在接口来指定调用的远端服务

## 2.整合Hystrix配置

Feign会将所有Feign客户端的方法都封装到Hystrix命令中进行服务保护.

Hystrix的全局配置.

熔断时间:

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=5000

Hystrix支持的启用/关闭(默认情况下是关闭对Hystrix的支持):

feign.hystrix.enabled=ture/false

服务降级:

当Feign客户端的调用被熔断之后,Ribbon的做法是通过fallback参数来指定降级的方法.而在Feign中只需要编写一个客户端的实现类,重写具体方法的实现逻辑来定义对应的服务降级逻辑.(fallback的实现类必须是Spring的Component)

## 3.传输压缩

Feign支持对请求和响应进行GZIP的压缩.以减少在通讯过程中的性能损耗.

## 4.注意和配置:

Ribbon的超时和Hystrix的超时是两个概念.为了让Ribbon的配置生效.我们必须要让Hystrix的超时时间大于Ribbon的超时时间.否则Hystrix超时后改命令就会直接熔断,重试机制就没有任何意义了.

由于Feign是基于Ribbon来实现的,所以Ribbon的配置会对远端服务的调用产生影响.

全局服务配置:

ribbon.ConnectTimeout=XXX

ribbon.ReadTimeout=XXX

另外Ribbon也可以针对指定的服务进行配置

<client>.ribbon.key=value

keys:

ConnectTimeout:访问连接超时时间.

ReadTimeout:数据返回超时时间

OkToRetryOnAllOperations:true.重试全部方法调用

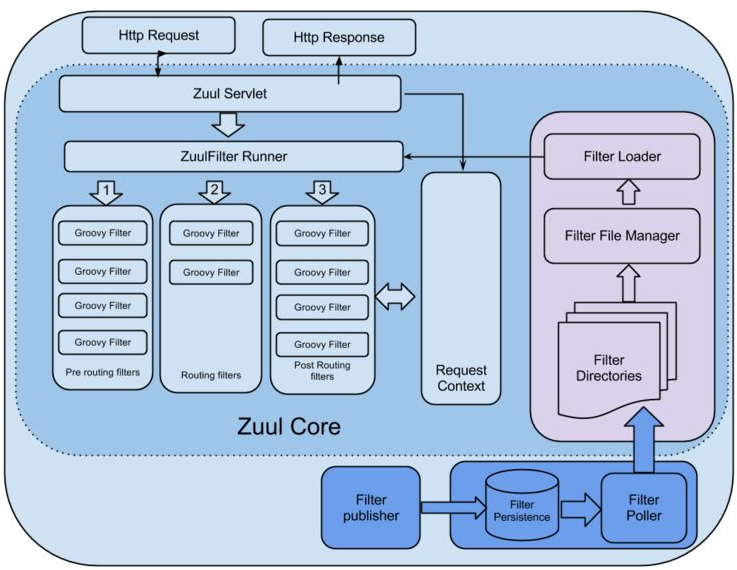
MaxAutoRetries:重试的最大次数

MaxAutoRetriesNextServer:最大尝试切换服务实例个数

在重试机制中默认为0,Ribbon从注册中心获取到服务端的地址列表之后,根据负载均衡机制对第一个实例进行请求,如果超时则对该实例进行重试,当达到了重试的最大次数时,则会从列表中获取下一个实例再对其进行重试次数的尝试.

# 五.Zuul

Zuul是用来服务”边缘服务”,API网关的定义类似于面向对象设计模式中的Facade模式,它的存在就像在整个微服务架构系统的门面,所有的外部客户端访问都需要经过它来进行调度和过滤.实现请求路由、负载均衡、校验过滤、与服务治理框架的结合、请求转发的熔断机制、服务的聚合、安全、日志、协议转换、鉴权、监控等一系列高级高能。



## 1.请求路由

引入依赖:

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-zuul</artifactId>  
</dependency>

启动Zuul代理注解

@EnableZuulProxy

为请求路径设置转发服务

zuul.routes.testApi.path=/testApi/\*\*  
zuul.routes.testApi.url=EUREKACLIENT-01

## 2.Cookie与头消息

Zuul在请求路由的时候,会默认过滤掉HTTP请求头信息中的一些敏感信息.防止它们被传递到下游的外部服务器.默认的敏感头消息通过

zuul.sensitiveHeaders参数来定义,默认Cookie、Set-Cookie、Authorization

但是在微服务架构的API网关之内,对于无状态的Restful APi请求肯定是要远远多于这些Web类应用的请求,甚至还有一些架构设计会将Web应用和App客户端一类都归类为Api网关之外的客户端应用.

如果着类应用需要从网关进行路由,可以针对指定的路由开启自定义敏感头

#对指定路由开启自定义敏感头

zuul.routes.<router>.customSensitiveHeaders=true

#将知道你个路由的敏感头设置为空

zuul.routes.<router>.sensitiveHeaders=XXX

## 3.Zuul、Hystrix和Ribbon的整合

由于Zuul依赖中已经包含了Hystrix和Ribbon,所以Zuul天生拥有线程隔离和熔断器的自我保护功能.以及对服务调用的客户端负载均衡功能.但是需要注意的是,路由的时候使用path和serviceId的组合来进行配置才能有保证APi网关的健壮和稳定,也能用到Ribbon的客户端负载均衡功能.

#关闭全局路由重试

zuul.retryable=false

#关闭指定路由重试

zuul.routes.<route>.retryable=false

## 4.Zuul过滤器

Zuul中实现的过滤器必须包含4个基本特征:过滤类型、顺序、执行条件、具体操作(Zuul拦截器接口中的四个重写方法)

filterType:该函数需要返回一个字符串来代表过滤器的类型,而这个类型就是HTTP请求过程中定义的各个阶段.在Zuul中定义了4中不同生命周期的过滤类型.

pre:可以在请求被路由之前调用.

routing:在路由请求时被调用.

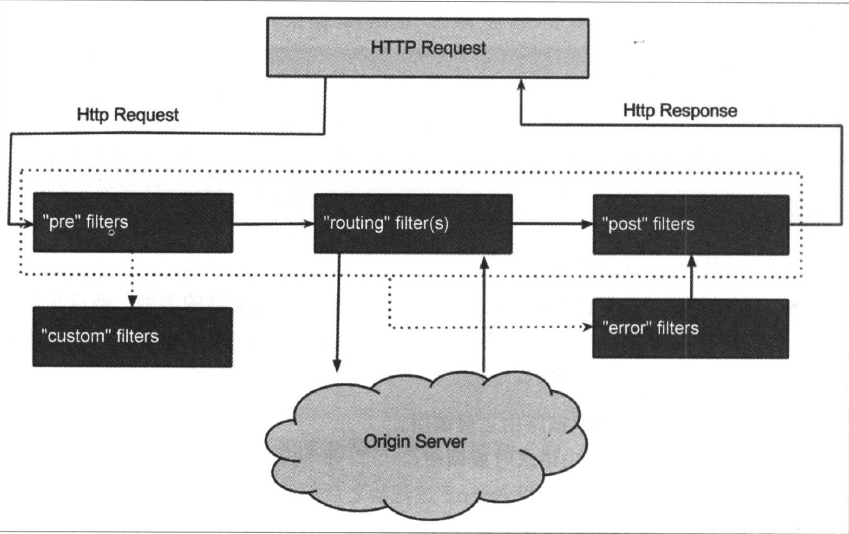
post:在routing和error过滤器之后被调用.

error:在处理请求时发生错误时被调用.

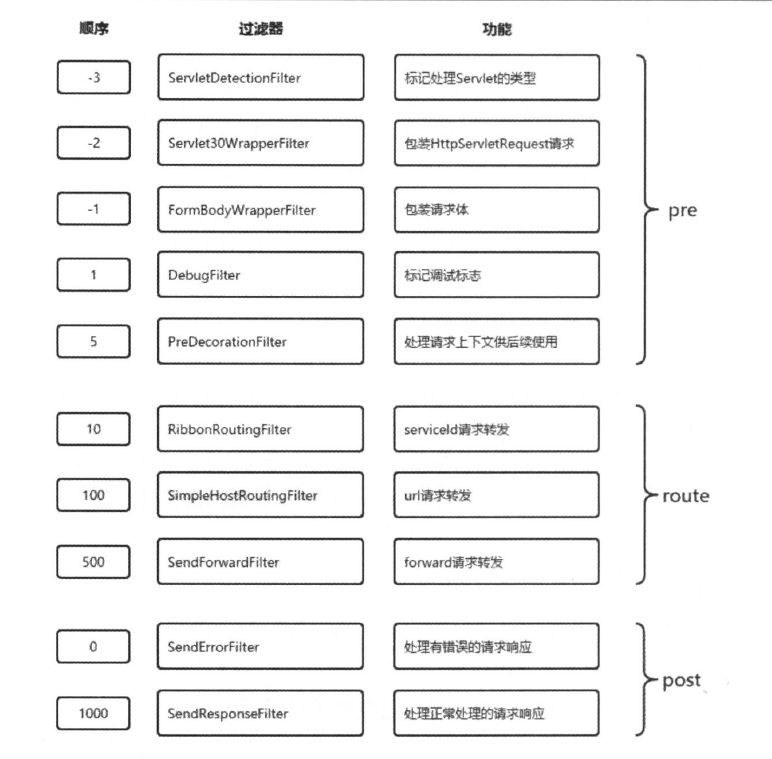
filterOrder:通过int值来定义过滤器的执行顺序,数值越小优先级越高.

shouldFilter:返回一个boolean值来判断该顾虑器是否要执行,我们可以通过此方法来指定过滤器的有效范围.

run:过滤器的具体逻辑.在该函数中,我们可以实现自定义的过滤逻辑,来确定是否要拦截当前的请求,不对其进行后续的路由,或是在请求路由返回结果之后,对处理结果做一些加工等.



在Zuul中原生就带有的拦截器有如下:



禁用过滤器:

在Zuul中提供一个参数来禁用指定的过滤器

zuul.<SimpleClassName>.<filterType>.disable=true

如:zuul.AccessFilter.pre.disable=true

动态加载

由于Zuul为了方便用户定制并且动态化加载配置引入了Groovy,提供了动态路由和动态过滤的能力,我们可以在不重启API网关服务的前提下,为其动态修改路由规则和添加或删除过滤器.

# 六.Config