

# SpringCloud服务网关

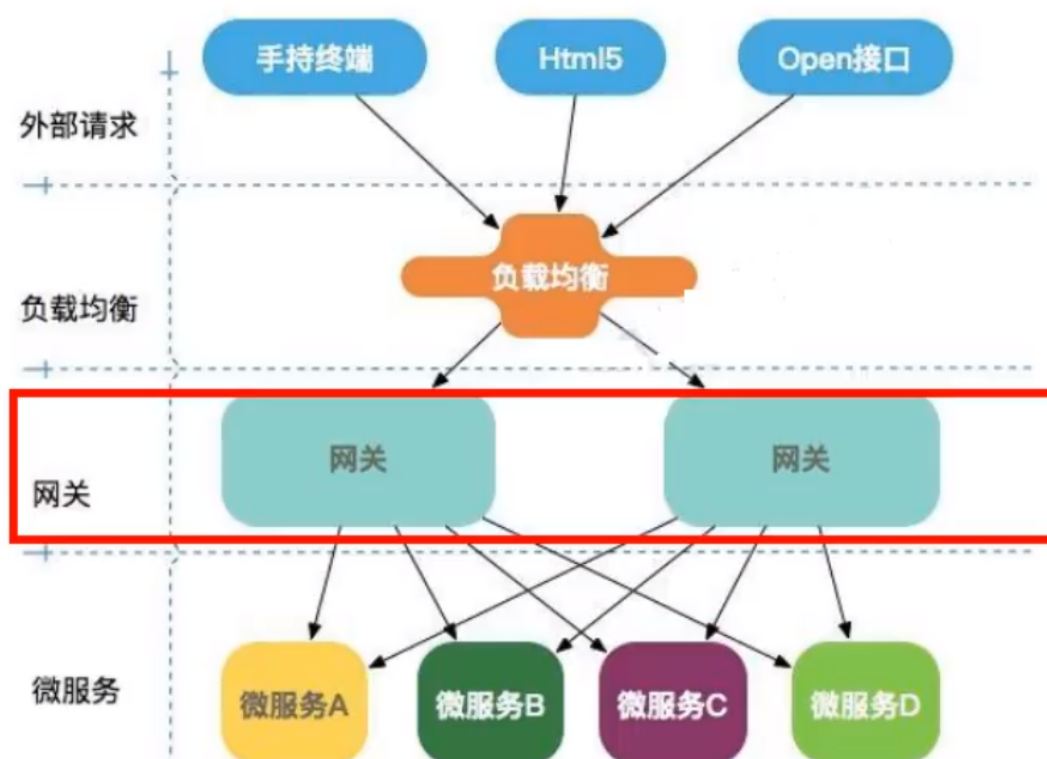
## 1.由于某些历史原因，技术选型使用了Gateway作为服务网关

### 1.简介

SpringCloud Gateway是在Spring生态系统之上构建的API网关服务，基于Spring 5, Spring Boot 2和Project Reactor等技术。Gateway旨在提供一种简单而有效的方式来对API进行路由，以及提供一些强大的过滤器功能，例如：熔断、限流、重试等。

SpringCloud Gateway是基于WebFlux框架实现的，而WebFlux框架底层则是使用了高性能的Reactor模式通信框架Netty，Spring Cloud Gateway的目标是提供统一的路由方式且基于Filter链的方式提供了网关基本的功能，例如：安全、监控/指标和限流。

### 2.位置



## 2.Gateway非阻塞异步模型

### 1.Spring Cloud Gateway特性

基于Spring Framework 5, Project Reactor和Spring boot 2.0进行构建

动态路由：能够匹配任何请求属性

可对路由指定Predicate(断言)和Filter(过滤器)

#### 集成Hystrix断路器功能

集成Spring Cloud服务发现功能

易于编写的Predicate(断言)和Filter(过滤器)

请求限流功能

支持路径重写

## 2.Spring Cloud Gateway与Zuul的区别

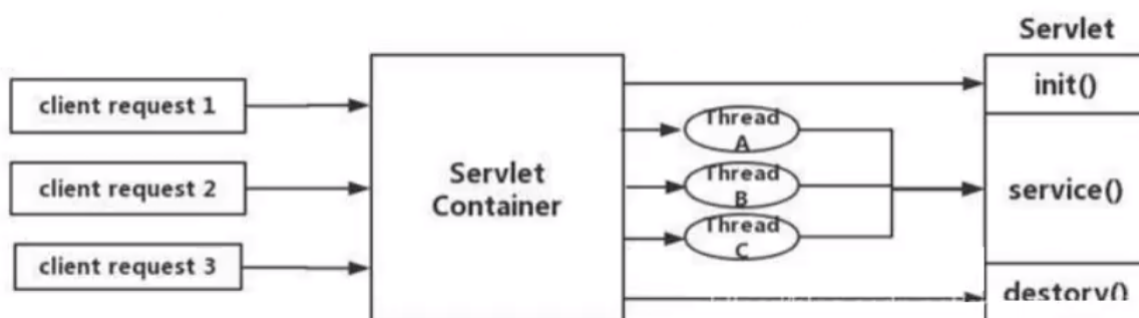
在Spring Cloud Finchley正式版之前，Spring Cloud推荐的网关是Netflix提供的Zuul:

- Zuul 1.x是一个基于阻塞I/O的API Gateway
- Zuul 1.x是基于Servlet2.5使用阻塞架构，它不支持任何长连接（如WebSocket）Zuul的设计模式和Nginx比较像，每次I/O操作都是从工作线程中选择一个执行，请求线程被阻塞到工作线程完成，但是差别是Nginx是用C++实现，Zuul使用java实现，而JVM本身会有第一次加载比较慢的请求，使得Zuul的性能相对较差
- Zuul 2.x理念更为先进，想基于Netty非阻塞和支持长连接，但SpringCloud目前还没有整合。Zuul 2.x的性能较Zuul 1.x有较大的提升。在性能方面，根据官方提供的基准测试，Spring Cloud Gateway的RPS（每秒请求数）是Zuul的1.6倍。
- Spring Cloud Gateway建立在Spring Framework 5、Project Reactor和Spring Boot 2之上，使用非阻塞API。
- Spring Cloud Gateway还支持WebSocket,并且与Spring紧密集成，拥有更好的开发体验

## 3.Zuul 1.x的模型

Spring Cloud中所集成的Zuul版本，采用的是Tomcat容器。使用的是传统的Servlet IO处理模型。

Servlet由Servlet container进行生命周期管理。container（容器，如tomcat）启动时，构造Servlet对象并调用Servlet init()进行初始化。container运行时接受请求，并为每一个请求分配一个线程（一般从线程池中获得空闲线程），然后调用service()。container关闭时调用Servlet destroy()销毁Servlet。



## 4.上述模式的缺点:

Servlet是一个简单的网络I/O模型，当请求进入Servlet container时，Servlet container就会为其绑定一个线程，在并发不高的场景下，这种模型是适用的。但是一旦在高并发场景下，线程数量上涨，而线程资源代价是昂贵的，（上下文切换，内存消耗大）严重影响请求的处理时间，在一些简单的业务场景下，不希望为每一个request分配一个线程，只需要1个或几个线程就能应对极大并发的请求。这种业务场景下，Servlet模型没有优势。

所以Zuul 1.x是基于Servlet上的一个阻塞式处理模型，即Spring实现了所有request请求的一个Servlet（DispatcherServlet）并由该Servlet阻塞式处理，所以SpringCloud Zuul无法摆脱Servlet模型的弊端

## 5.Spring Cloud Gateway模型

Spring Cloud Gateway是基于WebFlux的，传统的Web框架，比如说：struts2,springmvc等都是基于Servlet API与Servlet容器基础上运行的。但是，在Servlet3.1之后有了异步非阻塞的支持。而WebFlux是一个典型非阻塞异步的框架，它的核心是基于Reactor的相关API实现的。相对于传统的web框架来说，它可以运行在诸如Netty，Undertow以及支持Servlet3.1的容器上。非阻塞式+函数式编程(Spring5必须要求试用Java8)

Spring WebFlux是Spring 5.0引入的新的响应式框架，区别于Spring MVC,它不需要依赖Servlet API,它是完全异步非阻塞的，并且基于Reactor来实现响应式流规范。

### 3. Gateway 工作流程

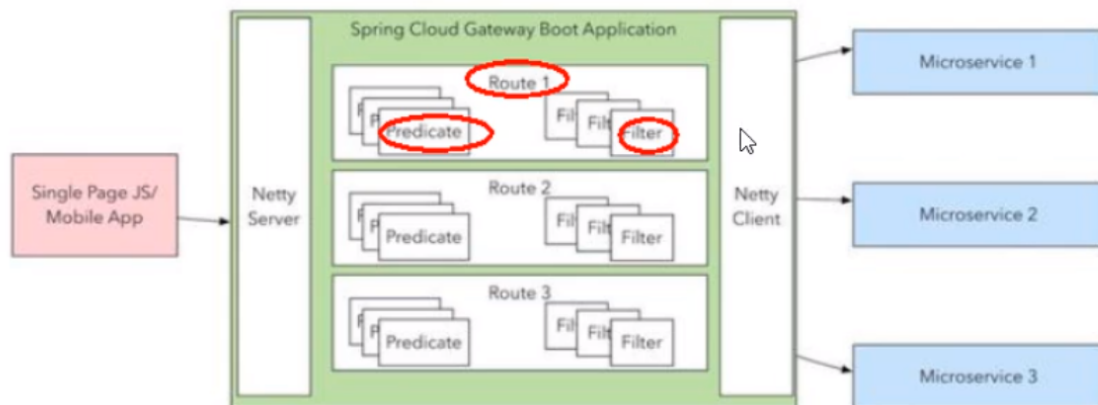
#### 1. 三大核心概念

**Route(路由)**: 路由是构建网关的基本模块，它由ID，目标URI，一系列的断言和过滤器组成，如果断言为true则匹配该路由

**Predicate(断言)**: 开发人员可以匹配HTTP请求中的所有内容（例如请求头或请求参数），如果请求与断言相匹配则进行路由

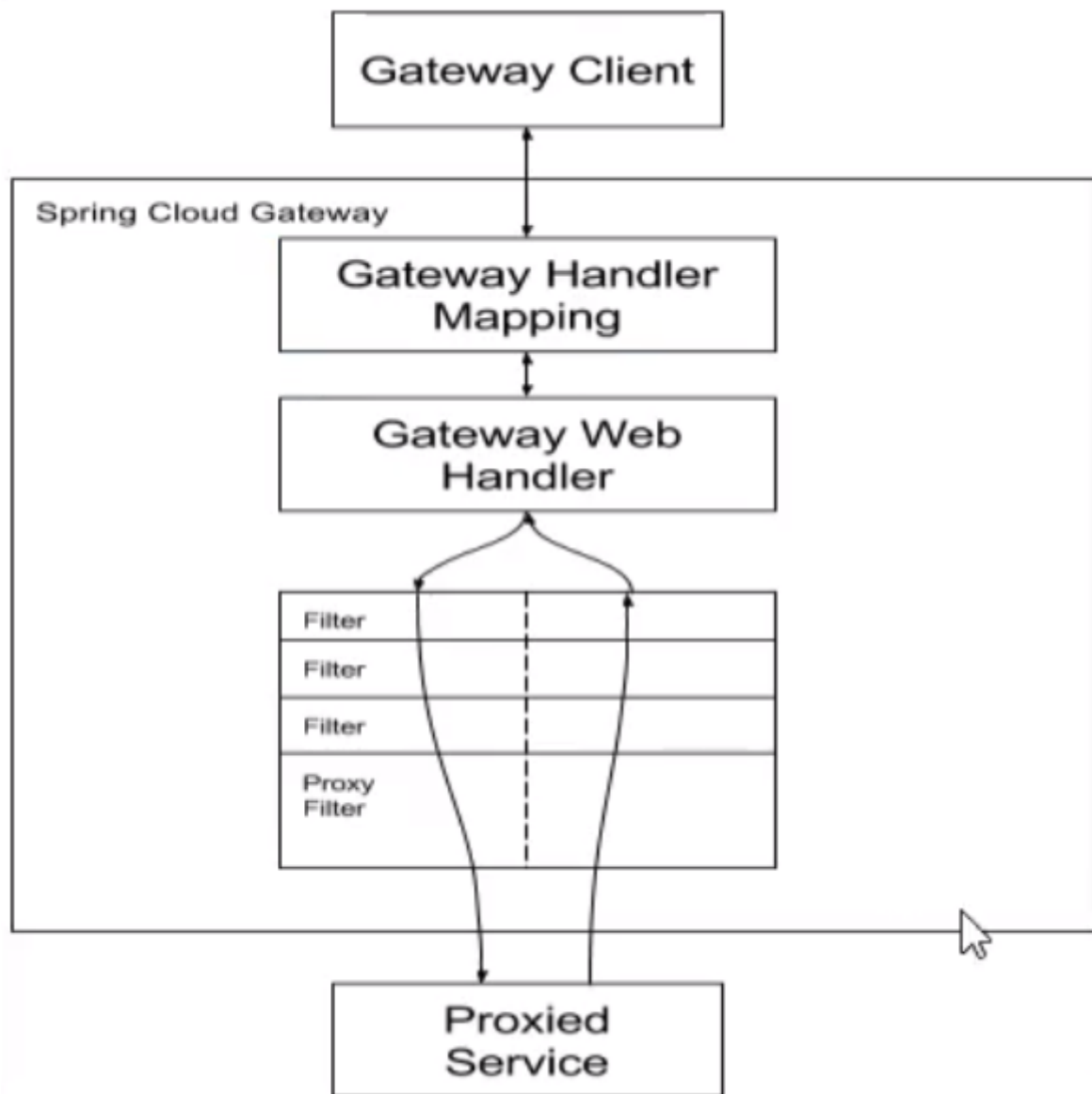
**Filter(过滤)**: 指的是Spring框架中GatewayFilter的实例，使用过滤器，可以在请求被路由前或者之后对请求进行修改。

#### 2. 整体



web请求，通过一些匹配条件，定位到真正的服务节点。并在这个转发过程前后，进行一些精细化控制。predicate就是我们匹配的条件；而filter，就可以理解为一个无所不能的拦截器，有了这两个元素，再加上目标uri，就可以实现一个具体的路由了

#### 3. 工作流程



客户端向Spring Cloud Gateway发出请求，然后在Gateway Handler Mapping中找到与请求相匹配的路由，将其发送到Gateway Web Handler.

Gateway Web Handler再通过指定的过滤器链来将请求发送到我们实际的服务执行业务逻辑，然后返回。

过滤器之间用虚线分开开始因为过滤器可能会在发送代理请求之前("pre")（就是filter发送请求之前）或之后("post")（就是filter发送请求之后）执行业务逻辑。

Filter在“pre”（之前）类型的过滤器可以做参数校验、权限校验、流量监控、日志输出、协议转换等。在“post”(之后)可以做相应内容、响应头的修改、日志的输出，流量监控等有着非常重要作用的功能。

## 4.cloud-gateway-gateway-9527工程搭建

pom

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
    <artifactId>SpringCloudStudyNew</artifactId>
    <groupId>com.echo</groupId>
```

```

    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>cloud-gateway-gateway-9527</artifactId>

  <properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
  </properties>
  <dependencies>
    <!--      引入gateway-->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
    <!--      引入EurekaClient-->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    </dependency>
    <!--      引入自定义的公共模块-->
    <dependency>
      <groupId>com.echo</groupId>
      <artifactId>cloud-api-common</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-actuator</artifactId>
    </dependency>
    <!--      springboot热部署-->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <!--      runtime表示被依赖项目无需参与项目的编译，不过后期的测试和运行
      周期需要其参与。与compile相比，跳过编译而已，-->
      <scope>runtime</scope>
      <!--      <optional>true</optional>表示两个项目之间依赖不传递；不设置
      optional或者optional是false，表示传递依赖。-->
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.projectlombok</groupId>
      <artifactId>lombok</artifactId>
      <optional>true</optional>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>

```

yml

```
server:
  port: 9527

spring:
  application:
    name: cloud-gateway

eureka:
  instance:
    hostname: cloud-gateway-service
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka
```

主启动

```
@SpringBootApplication
@EnableEurekaClient
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}
```

## 1.路由映射

映射cloud-provider-payment-8001,修改其yml文件为单机版eureka, 对其controller中的lb和get接口进行路由, 不想暴露8001端口, 在8001端口外面套一层9527

yml中新增网关配置

```
server:
  port: 9527

spring:
  application:
    name: cloud-gateway
  cloud:
    gateway:
      routes:
        - id: payment_routh1 #路由的ID, 没有固定规则, 但是要求唯一, 建议配合服务名
          uri: http://localhost:8001 #匹配后提供服务的路由地址
          predicates:
            - Path=/payment/get/** #断言, 路径相匹配的进行路由
        - id: payment_routh2
          uri: http://localhost:8001
          predicates:
            - Path=/payment/lb/**
```

```
eureka:
  instance:
    hostname: cloud-gateway-service
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka
```

## 2.测试

启动7001 启动 cloud-payment-service-8001 启动9527

这里在启动9527的时候有一个坑

```
*****
APPLICATION FAILED TO START
*****

Description:

Parameter 0 of method modifyResponseBodyGatewayFilterFactory in org.springframework.cloud.gateway.config.Gat

Action:
```

Parameter 0 of method modifyResponseBodyGatewayFilterFactory in org.springframework.cloud.gateway.config.GatewayAutoConfiguration required a bean of type 'org.springframework.http.codec.ServerCodecConfigurer' that could not be found.

这是因为在pom以来中添加了springboot web启动器和actuator监视器。gateway在启动时不需要这两个依赖，所以要注释掉

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

然后再重新启动

添加网关之前是使用<http://localhost:8001/payment/get/1>访问

添加网关之后是使用<http://localhost:9527/payment/get/1>访问

```
localhost:9527/payment/get/1
{
  code: 200,
  message: "success,the port is :8001",
  data: {
    id: 1,
    serial: "aabbbaa"
  }
}
```

```
1 // 20210513165546
2 // http://localhost:9527/payment/get/1
3
4 {
5   "code": 200,
6   "message": "success,the port is :8001",
7   "data": {
8     "id": 1,
9     "serial": "aabbbaa"
10  }
11 }
```

## 5. Gateway路由配置的两种方式

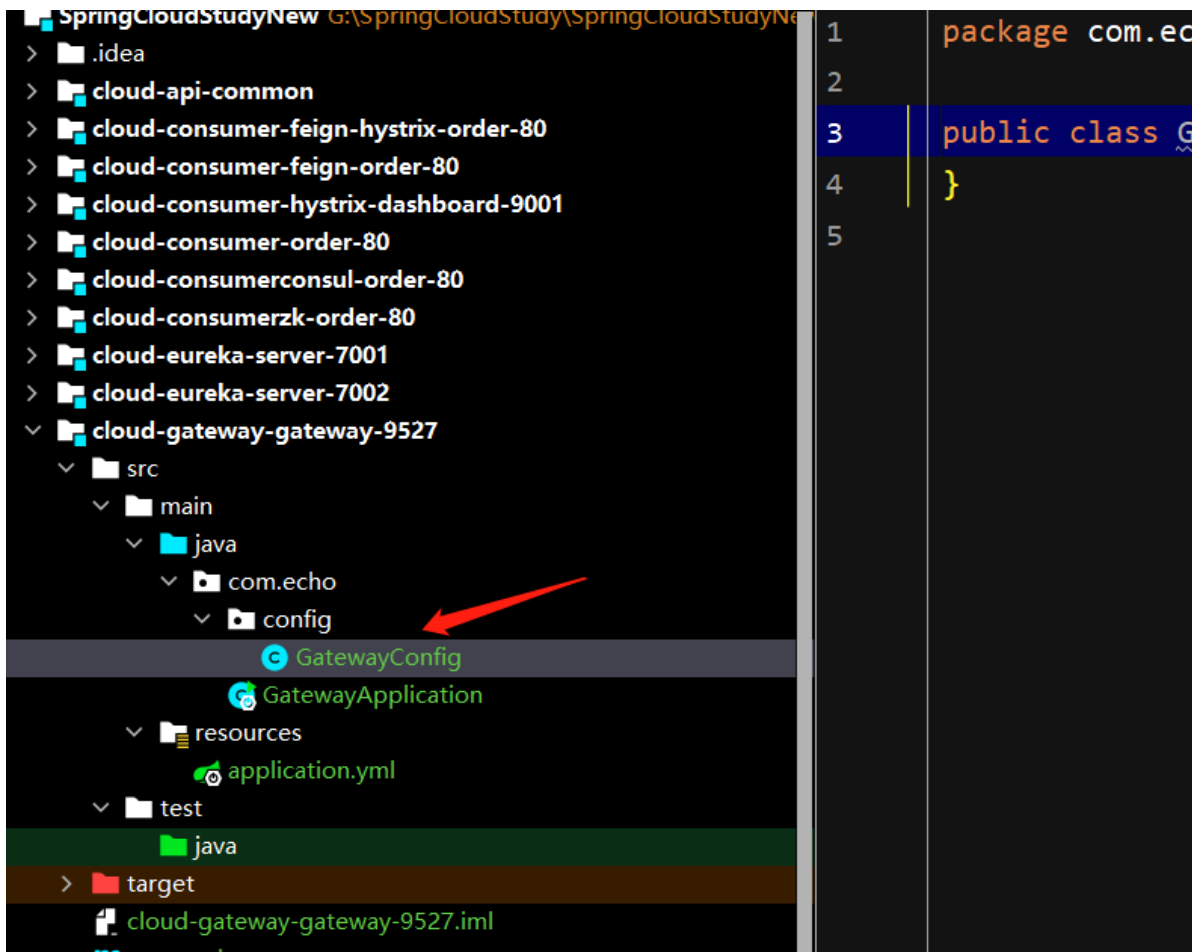
1. 上述yml中的配置

2. 代码中注入RouteLocator的Bean

实现一个业务，通过9527网关访问到外网的百度新闻地址，不再使用yml配置，而是采取编码的方式

3. 创建一个config包，新建一个Config配置类



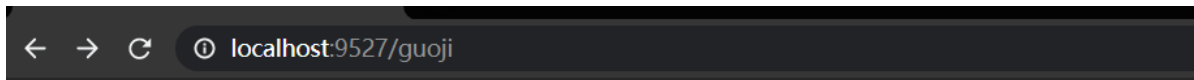


```
@Configuration
public class GatewayConfig {
    @Bean
    public RouteLocator customRouteLocator(RouteLocatorBuilder
routeLocatorBuilder){
        //相当于yaml中的routes 配置
        RouteLocatorBuilder.Builder routes = routeLocatorBuilder.routes();
        //第一个参数是这个路由的ID
        //第二个是一个函数型参数，定义了一个路径，以及该路径路由到哪个URL,路由到了百度新
        闻。。。
        routes.route("path_route_echo1",
            r -> r.path("/guonei").uri("http://news.baidu.com/guonei")
        ).build();
        return routes.build();
    }
}
```

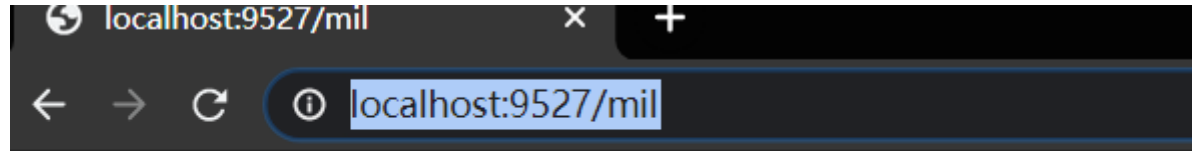
#### 4.启动测试



#### 5.一件很神奇的事情



点击这个链接会直接跳到



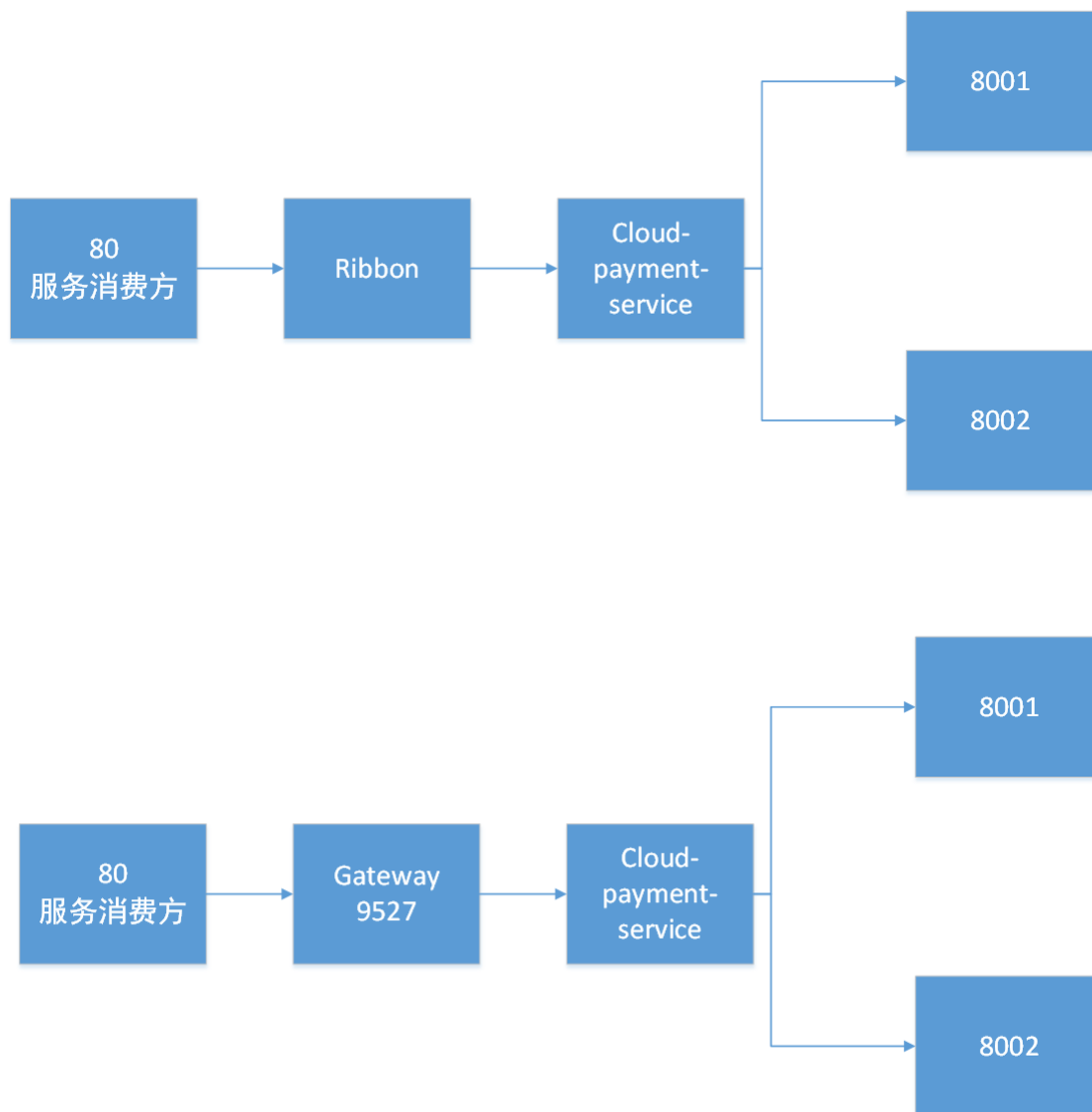
# Whitelabel Error Page

This application has no configured error view, so you are seeing this page.

Thu May 13 17:13:59 CST 2021

## 6.Gateway配置动态路由

### 1.架构演变



## 2.通过微服务名实现动态路由

默认情况下Gateway会根据注册中心注册的服务列表，以注册中心上微服务名为路径创建动态路由进行转发，从而实现动态路由的功能

启动：一个eureka 7001 + 两个服务提供者8001 8002 ， 网关**Gateway是带有负载均衡功能的**

## 3.修改gateway的yml，进行动态路由

```

server:
  port: 9527

spring:
  application:
    name: cloud-gateway
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true      #开启从注册中心动态创建路由的功能，利用微服务名进行路由
      routes:
        - id: payment_routh1 #路由的ID，没有固定规则，但是要求唯一，建议配合服务名
          uri: lb://cloud-payment-service #动态路由，匹配后提供服务的路由地址
          # 需要注意的是uri的协议为lb，表示启用Gateway的负载均衡功能。 lb://微
          # 服务名 是spring cloud gateway在微服务中自动为我们创建的负载均衡uri
          predicates:

```

```

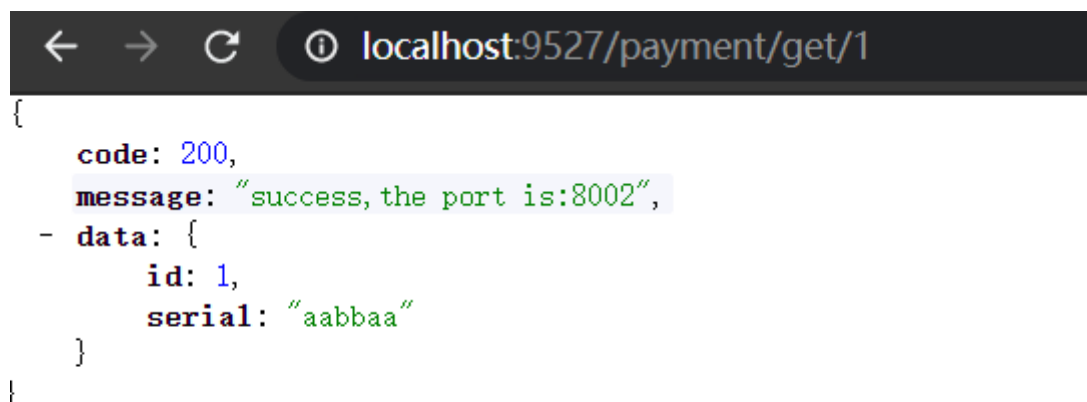
- Path=/payment/get/**      #断言，路径相匹配的进行路由

- id: payment_routh2
  uri: lb://cloud-payment-service #动态路由，匹配后提供服务的路由地址
  predicates:
    - Path=/payment/lb/**

eureka:
  instance:
    hostname: cloud-gateway-service
  client:
    register-with-eureka: true
    fetch-registry: true
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka

```

#### 4.测试



```

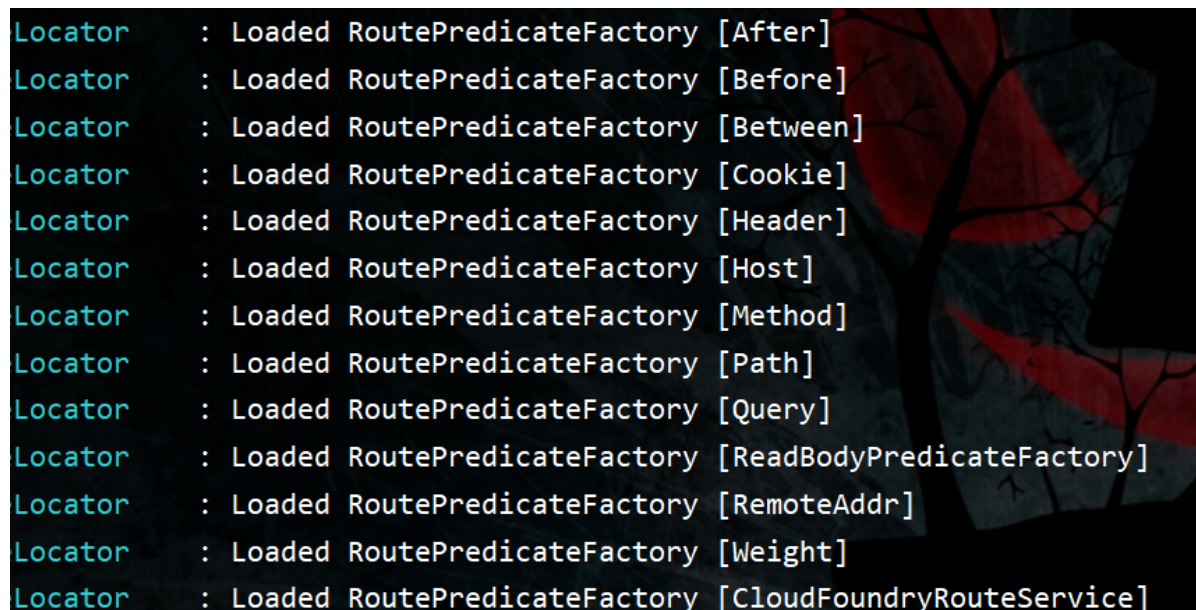
{
  code: 200,
  message: "success, the port is:8002",
  data: {
    id: 1,
    serial: "aabbbaa"
  }
}

```

访问成功，多次测试也可以发现负载均衡功能。

## 7.Gateway常用的Predicate

### 1.是什么？



```

Locator      : Loaded RoutePredicateFactory [After]
Locator      : Loaded RoutePredicateFactory [Before]
Locator      : Loaded RoutePredicateFactory [Between]
Locator      : Loaded RoutePredicateFactory [Cookie]
Locator      : Loaded RoutePredicateFactory [Header]
Locator      : Loaded RoutePredicateFactory [Host]
Locator      : Loaded RoutePredicateFactory [Method]
Locator      : Loaded RoutePredicateFactory [Path]
Locator      : Loaded RoutePredicateFactory [Query]
Locator      : Loaded RoutePredicateFactory [ReadBodyPredicateFactory]
Locator      : Loaded RoutePredicateFactory [RemoteAddr]
Locator      : Loaded RoutePredicateFactory [Weight]
Locator      : Loaded RoutePredicateFactory [CloudFoundryRouteService]

```

其实就是越来越详细的匹配规则，一个Predicate就是一种匹配条件，官方自带的就有十一种

String Cloud Gateway将路由匹配作为Spring WebFlux HandlerMapping基础架构的一部分。Spring Cloud Gateway包括许多内置的Route Predicate工厂。所有这些Predicate都与HTTP请求的不同属性匹配。多个Route Predicate工厂可以进行组合。

String Cloud Gateway 创建Route对象是，使用RoutePredicateFactory创建Predicate对象， Predicate对象可以赋值给Route， Spring Cloud Gateway包含许多内置的Route Predicate Factories

所有这些谓词都匹配HTTP请求的不同的属性，多种谓词工厂可以组合，并通过逻辑And.

## 2.常用的Predicate

- After Route Predicate
- Before Route Predicate
- Between Route Predicate
- Cookie Route Predicate

Cookie Route Predicate需要两个参数，一个是Cookie Name,一个是正则表达式，路由规则会通过获取相的Cookie Name值和正则表达式去匹配，如果匹配上就会执行路由，如果没有匹配上就不执行路由

配置一下

- `Cookie=username,echo` # `cookie`中的`username`属性中包含 正则表达式中定义的`echo`，请求才会有效

## 不带cookie的访问出错

```
C:\Users\Echo>curl http://localhost:9527/payment/lb
{"timestamp": "2021-05-14T02:10:35.139+0000", "path": "/payment/lb", "status": 404, "error": "Not Found", "message": null, "requestId": "0A0d56f7", "trace": "org.springframework.web.server.ResponseStatusException: 404 NOT_FOUND\r\n\tat org.springframework.web.reactive.handler.WebHandler.lambda$handle$0(ResourceWebHandler.java:325)\r\n\tSuppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException: \nError has been observed at the following site(s) :\r\n\tcheckpoint [org.springframework.cloud.gateway.filter.WeightCalculatorWebFilter [DefaultWebFilterChain]\r\n\tcheckpoint [HTTP GET ^\r\n\tpayment/lb ] [ExceptionHandlerWebHandler] \nStack trace :\r\n\t\tat org.springframework.web.reactive.resource.ResourceWebHandler.lambda$handle$0(ResourceWebHandler.java:325)\r\n\t\t\r\n\t\tat reactor.core.publisher.MonoDefer.subscribe(MonoDefer.java:44)\r\n\t\t\r\n\t\tat reactor.core.publisher.Mono.subscribe(Mono.java:4105)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxSwitchIfEmpty$SwitchIfEmptySubscriber.onComplete(FluxSwitchIfEmpty.java:75)\r\n\t\t\r\n\t\tat reactor.core.publisher.MonoFlatMap$FlatMapMain.onComplete(MonoFlatMap.java:174)\r\n\t\t\r\n\t\tat reactor.core.publisher.MonoNext$NextSubscriber.onComplete(MonoNext.java:96)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxConcatMap$ConcatMapImmediate.drain(FluxConcatMap.java:359)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxConcatMap$ConcatMapImmediate.onSubscribe(FluxConcatMap.java:211)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxIterable.subscribe(FluxIterable.java:63)\r\n\t\t\r\n\t\tat reactor.core.publisher.Mono.subscribe(Mono.java:4105)\r\n\t\t\r\n\t\tat reactor.core.publisher.MonoIgnoreThen$ThenIgnoreMain.run(MonoIgnoreThen.java:172)\r\n\t\t\r\n\t\tat reactor.core.publisher.MonoIgnoreThen.subscribe(MonoIgnoreThen.java:56)\r\n\t\t\r\n\t\tat reactor.core.publisher.MonoFlatMap$FlatMapMain.onNext(MonoFlatMap.java:150)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxSwitchIfEmpty$SwitchIfEmptySubscriber.onNext(FluxSwitchIfEmpty.java:67)\r\n\t\t\r\n\t\tat reactor.core.publisher.MonoNext$NextSubscriber.onNext(MonoNext.java:76)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxConcatMap$ConcatMapImmediate.innerNext(FluxConcatMap.java:274)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxConcatMap$ConcatMapInner.onNext(FluxConcatMap.java:851)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onNext(FluxMapFuseable.java:121)\r\n\t\t\r\n\t\tat reactor.core.publisher.Operators$ScalarSubscription.request(Operators.java:2186)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.request(FluxMapFuseable.java:162)\r\n\t\t\r\n\t\tat reactor.core.publisher.Operators$MultiSubscriptionSubscriber.set(Operators.java:1994)\r\n\t\t\r\n\t\tat reactor.core.publisher.Operators$MultiSubscriptionSubscriber.onSubscribe(Operators.java:1868)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onSubscribe(FluxMapFuseable.java:90)\r\n\t\t\r\n\t\tat reactor.core.publisher.MonoJust.subscribe(MonoJust.java:54)\r\n\t\t\r\n\t\tat reactor.core.publisher.Mono.subscribe(Mono.java:4105)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxConcatMap$ConcatMapImmediate.drain(FluxConcatMap.java:441)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxConcatMap$ConcatMapImmediate.onSubscribe(FluxConcatMap.java:211)\r\n\t\t\r\n\t\tat reactor.core.publisher.FluxIterable.subscribe(FluxIterable.java:63)\r\n\t\t\r\n\t\tat reactor.core.publisher.InternalMonoOperator.subscribe(InternalMonoOperator.java:55)\r\n\t\t\r\n\t\tat reactor.core.publ
```

## 带Cookie访问

```
C:\Users\Echo>curl http://localhost:9527/payment/1b --cookie "username=echo"
8002
C:\Users\Echo>curl http://localhost:9527/payment/1b --cookie "username=echo"
8001
C:\Users\Echo>curl http://localhost:9527/payment/1b --cookie "username=echo"
8002
C:\Users\Echo>curl http://localhost:9527/payment/1b --cookie "username=echo"
8001
C:\Users\Echo>curl http://localhost:9527/payment/1b --cookie "username=echo"
8002
C:\Users\Echo>_
```

访问成功

- Header Route Predicate



```

- Cookie=username,echo # cookie中的username属性中包含 正则表达式中定义的
echo, 请求才会有效
#- Header=X-Request-Id, \d+ #请求头中有X-Request-Id这个参数, 而且该参数的
值满足定义的正则表达式, 请求才会有效
- Host=**.echo.com #请求中携带匹配该规则的主机名时, 本次请求才有效
- Method=GET #get请求才允许访问
- Query=username, \d+ #要有参数名username且值满足后面定义的正则表达式的, 该
请求才有效

```

## 8. Gateway中的过滤器

### 1. 概念

指的是Spring框架中GatewayFilter的实例, 使用过滤器, 可以在请求被路由前或者之后对请求进行修改

2. web请求, 通过一些匹配条件, 定位到真正的服务节点。并在这个转发过程前后, 进行一些精细化的控制, predicate就是我们的匹配条件, 而filter就可以理解为一个无所不能的拦截器。有了这两个元素, 再加上目标uri, 就可以实现一个具体的路由。

### 3. 是什么?

路由过滤器可以用于修改进入的HTTP请求和返回的HTTP响应, 路由过滤器只能指定路由进行使用。Spring Cloud Gateway内置了多种过滤器, 他们都由GatewayFilter的工厂类来产生。

### 4. 生命周期

两个: 路由前pre和路由后post

### 5. 种类

两个: GatewayFilter单一过滤器 (有31种) 和GlobalFilter全局过滤器 (十几种), 其实重要的是自定义过滤器

### 6. 使用

```

- id: payment_routh2
  # uri: http://localhost:8001
  uri: lb://cloud-payment-service #动态路由, 匹配后提供服务的路由地址
  filters:
    - AddRequestParameter=X-Request-Id,1024 #过滤工厂会在匹配的请求头上加一对
    请求头, 名称为X-Request-Id值为1024

```

## 9. 自定义全局GlobalFilter过滤器

### 1. 两个主要接口

implements GlobalFilter, Ordered

### 2. 能干吗

全局日志记录、统一网关鉴权等

### 3. 使用

新建一个filter的包, 并在里面创建一个MyLogGatewayFilter的类, 实现全局日志

```

@Component
@Slf4j
public class MyLogGatewayFilter implements GlobalFilter, Ordered {

```



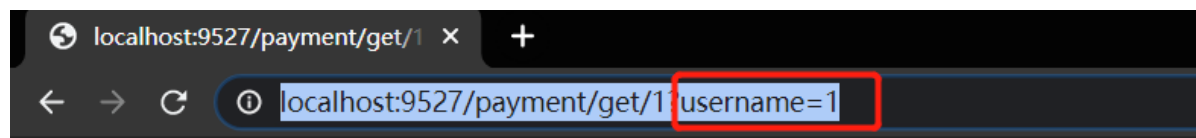
```

@Override
public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
    log.info("*****come in MyLogGatewayFilter : " + new Date());
    //从请求参数中取出来username
    String username =
exchange.getRequest().getQueryParams().getFirst("username");
    //如果username为空
    if(username == null){
        //日志提示
        log.info("*****用户名为null,非法用户*****");
        //过滤器设置一个响应
        exchange.getResponse().setStatusCode(HttpStatus.NOT_ACCEPTABLE);
        //过滤器返回这个响应, 请求结束
        return exchange.getResponse().setComplete();
    }
    //正常情况, 将请求转发给下一个过滤器
    return chain.filter(exchange);
}

//该过滤器的优先级
@Override
public int getOrder() {
    return 0;
}
}

```

#### 4.测试



```

{
  code: 200,
  message: "success, the port is:8002",
  data: {
    id: 1,
    serial: "aabbbaa"
  }
}

```

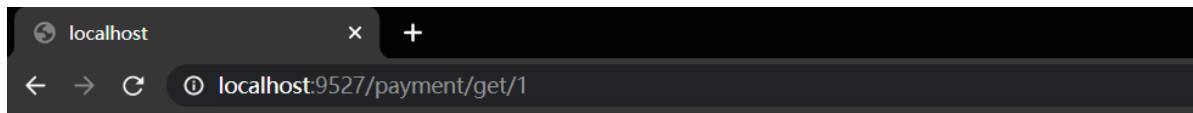
```

1 // 20210515102205
2 // http://localhost:9527/payment/get/1?username=1
3
4 {
5   "code": 200,
6   "message": "success, the port is:8002",
7   "data": {
8     "id": 1,
9     "serial": "aabbbaa"
10  }
11 }

```



携带username参数，可以访问成功。



## 该网页无法正常工作

如果问题仍然存在，请与网站所有者联系。

HTTP ERROR 406

重新加载

不携带则无法访问。