一、什么是Spring Cloud Gateway

1、网关简介

网关作为流量的入口，常用的功能包括路由转发，权限校验，限流等。

2、Gateway简介

Spring Cloud Gateway 是Spring Cloud官方推出的第二代网关框架，定位于取代 Netflix Zuul。相比 Zuul 来说，Spring Cloud Gateway 提供更优秀的性能，更强大的有功能。

Spring Cloud Gateway 是由 WebFlux + Netty + Reactor 实现的响应式的 API 网关。

它不能在传统的 servlet 容器中工作，也不能构建成 war 包。

Spring Cloud Gateway 旨在为微服务架构提供一种简单且有效的 API 路由的管理方式，并基于 Filter 的方式提供网关的基本功能，例如说安全认证、监控、限流等等。

网文档：https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#gateway-starter

3、核心概念

- 路由（route)

  路由是网关中最基础的部分，路由信息包括一个ID、一个目的URI、一组谓 词工厂、一组Filter组成。如果谓词为真，则说明请求的URL和配置的路由匹配。

- 谓词(predicates)

  即java.util.function.Predicate，Spring Cloud Gateway使用Predicate实现路由的匹配条件。

- 过滤器（Filter)

  SpringCloud Gateway中 的filter分为Gateway FilIer和Global Filter。Filter可以对请求和响应进行处理。

  【路由就是转发规则，谓词就是是否走这个路径的条件，过滤器可以为路由添加业务逻辑，修改请求以及响应】

示例演示

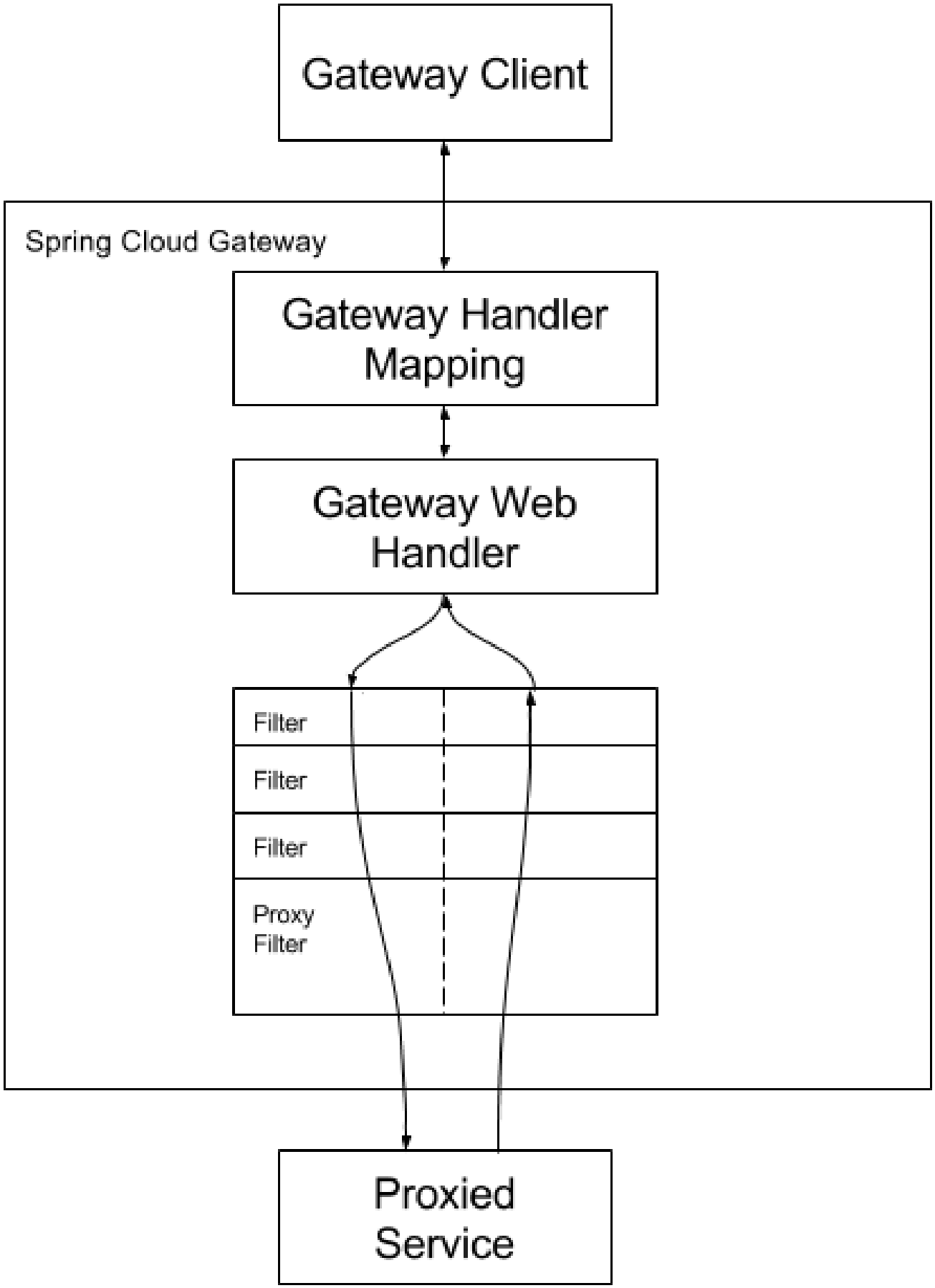
```
spring:
  cloud:
```

```
gateway:
  routes:
    #路由ID全局唯一
    - id: msb-order-route
      #目标微服务的请求地址和端口
      uri: http://localhost:8001
      predicates:
      - Path=/order/*
      filters:
      - AddRequestHeader=X-Request-Foo, Bar
```

这里就演示了请求 /order/*的路径就会路由到上边这个url上去

2、 工作原理

Spring Cloud Gateway 的工作原理跟 Zuul 的差不多，最大的区别就是 Gateway 的 Filter 只有 pre 和 post 两种。


Gateway Client
Spring Cloud Gateway
Gateway Handler Mapping
Gateway Web Handler
Filter
Filter
Filter
Proxy Filter
Proxied Service

客户端向 Spring Cloud Gateway 发出请求，如果请求与网关程序定义的路由匹配，则该请求就会被发送到网关 Web 处理程序，此时处理程序运行特定的请求过滤器链。

过滤器之间用虚线分开的原因是过滤器可能会在发送代理请求的前后执行逻辑。所有 pre 过滤器逻辑先执行，然后执行代理请求；代理请求完成后，执行 post 过滤器逻辑。

4、快速入门

3.1环境搭建

引入依赖

```
<!--  gateway网关-->
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
        </dependency>
</dependencies>
<!--  nacos注册中心-->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

父工程的pom

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
    </dependency>
</dependencies>
```

```
spring:
  application:
    name: msb-gateway
```

```
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
    gateway:
      discovery:
        locator:
          #默认值是false,如果设为true开启通过微服务创建路由的功能，即可以通过微服务名访问服务
          #http://localhost:13001/msb-order/order/create
          #不建议打开，因为这样暴露了服务名称
          enabled: true
      #是否开启网关
      enabled: true
```

这里注意不要引入springmvc工程否则会报错如下：

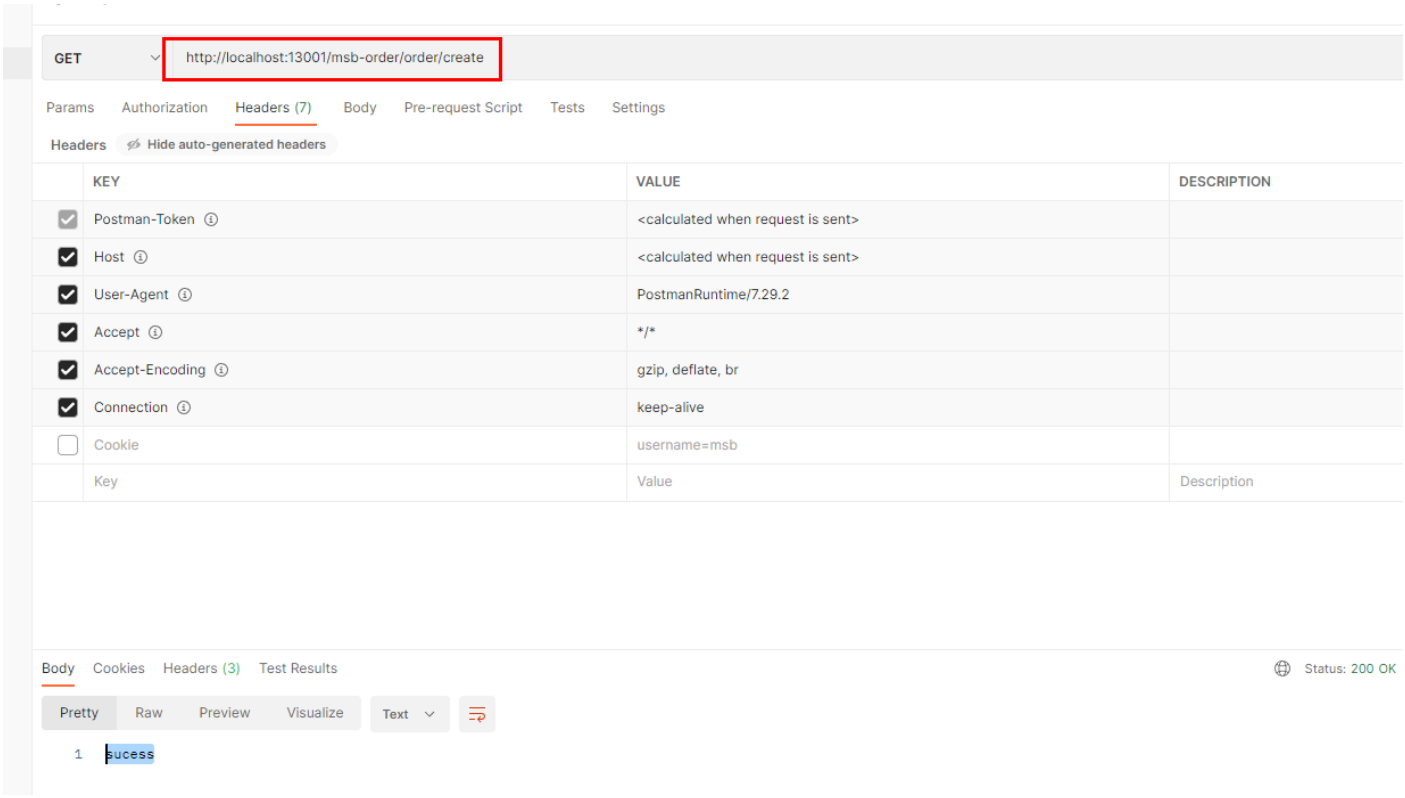
```
***************************************************

Spring MVC found on classpath, which is incompatible with Spring Cloud Gateway at this time. Please remove
 spring-boot-starter-web dependency.

***************************************************
```

我们工程中如下：

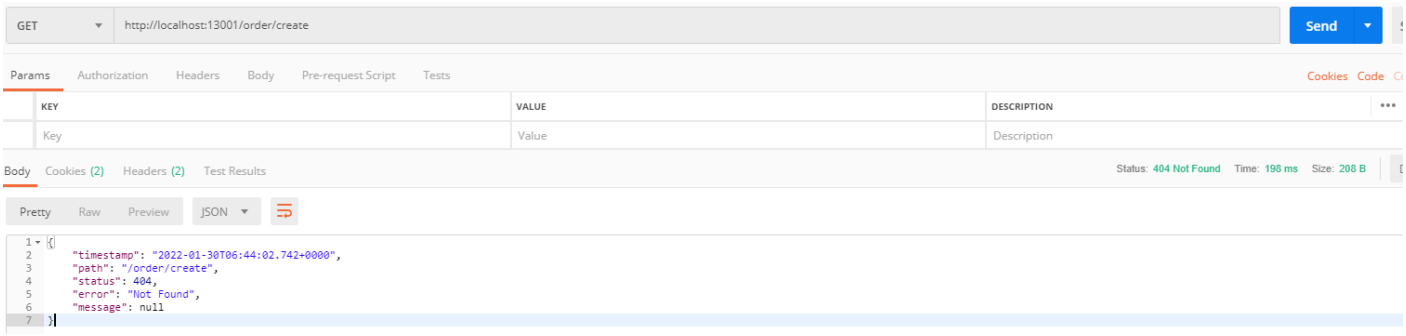
msb-gateway工程的pom

```
<dependencies>
  <!--  gateway网关-->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <scope>test</scope> <!-- 特殊处理，不引入父工程依赖 -->
    </dependency>
</dependencies>
```

3.2 测试



我们刚才说了我们不建议开启通过微服务创建路由的功能，这样就把微服务名称暴露了，如果设置位默认值在请求一下，是否能成功呢？ 我们测试一下：



是失败的，刚才能成功主要是

http://localhost:13001/msb-order/order/create 可以进行转化：

http://localhost:13001/msb-order/order/create可以替换为http://localhost:8001/order/create

然而此时的请求是转化不了的

http://localhost:13001/order/create

这是我们就需要我们的gateway 中的 route& predicates 建立映射

二、路由谓词工厂（Route Predicate Factories）配置

1、路由配置的两种形式

1.1 路由到指定URL

- 通配

```
spring:
  cloud:
    gateway:
      routes:
      - id: {唯一标识}
        uri: http://localhost:8001
```

表示访问 GATEWAY_URL/** 会转发到 http://localhost:8001/**

注：上面路由的配置必须和下面谓词（Predicate）配合使用才行

- 精确匹配

```
spring:
  cloud:
    gateway:
      routes:
      - id: {唯一标识}
        uri: http://localhost:8001/
        predicates:
            - Path=/order/*
```

表示访问 GATEWAY_URL/order/*会转发到 http://localhost:8001/order/*

1.2 路由到服务发现组件上的微服务

- 通配

```
spring:
  cloud:
    gateway:
      routes:
      - id: {唯一标识}
        uri: lb://msb-order
```

表示访问 GATEWAY_URL/** 会转发到 msb-order 微服务的 /**

注：上面路由的配置必须和下面谓词（Predicate）配合使用才行

- 精确匹配

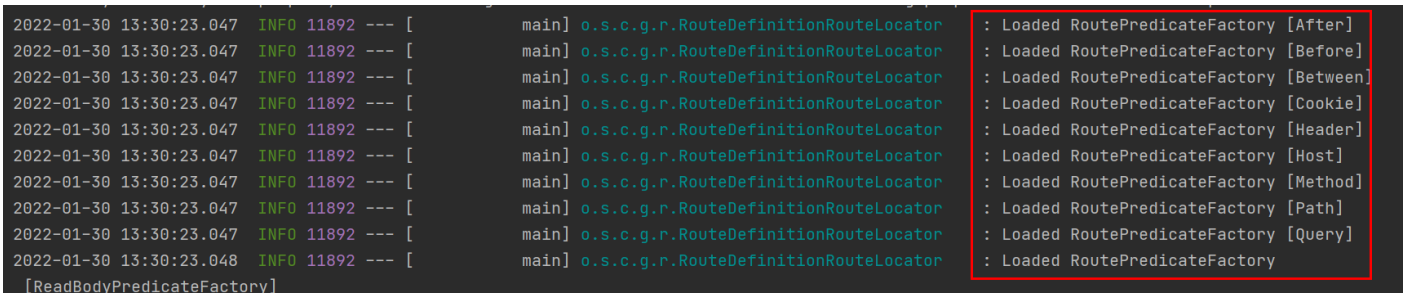
```
spring:
  cloud:
    gateway:
      routes:
      - id: {唯一标识}
        uri: lb://msb-order/
        predicates:
            - Path=/order/*
```

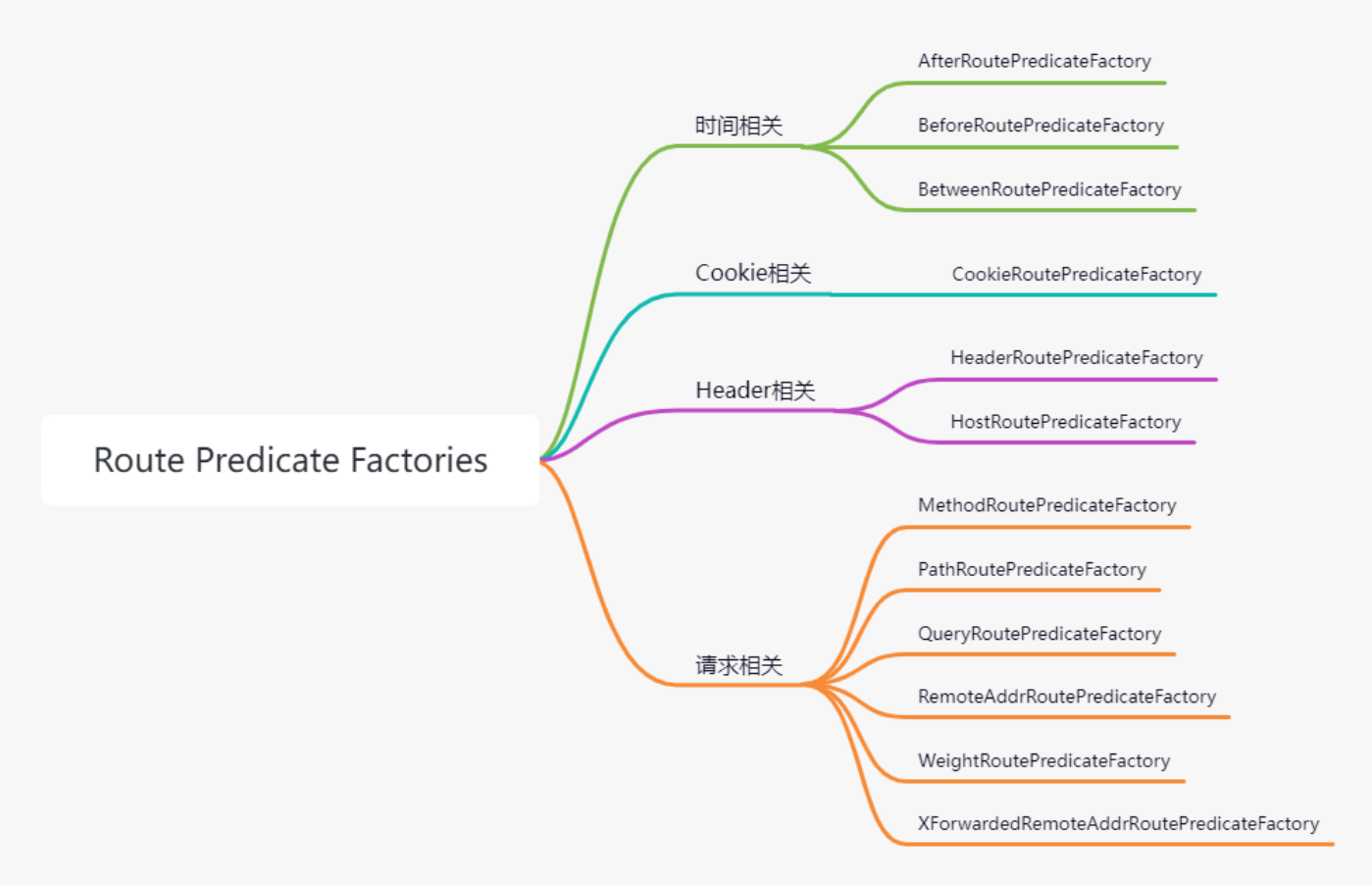
表示访问 GATEWAY_URL/order/ 会转发到 msb-order 微服务的 /order/

2、谓词工厂分类

官网：https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#gateway-request-predicates-factories

网关启动日志：





3、谓词介绍

3.1 After路由断言工厂

规则：

该断言工厂的参数是一个 UTC 格式的时间。其会将请求访问到 Gateway 的时间与该参数时间相比，若请求时间在参数

配置：

```
spring:
  cloud:
    gateway:
      routes:
        - id: after_route
          uri: lb://msb-user
          predicates:
            # 当且仅当请求时的时间After配置的时间时，才会转发到用户微服务
            # 目前配置不会进该路由配置，所以返回404
            # 将时间改成 < now的时间，则访问localhost:8040/** -> msb-user/**
            # eg. 访问http://localhost:8040/users/1 -> msb-user/users/1
            - After=2030-01-20T17:42:47.789-07:00[America/Denver]
```

技巧：时间可使用 System.out.println(ZonedDateTime.now()); 打印，然后即可看到时区。例如：2019-08-10T16:50:42.579+08:00[Asia/Shanghai]

时间格式的相关逻辑：

- 默认时间格式：org.springframework.format.support.DefaultFormattingConversionService#addDefaultFormatters
- 时间格式注册：org.springframework.format.datetime.standard.DateTimeFormatterRegistrar#registerFormatters

3.2 Before路由断言工厂

规则：

该断言工厂的参数是一个 UTC 格式的时间。其会将请求访问到 Gateway 的时间与该参数时间相比，若请求时间在参数

配置：

```
spring:
  cloud:
    gateway:
      routes:
        - id: before_route
          uri: lb://msb-user
          predicates:
            # 当且仅当请求时的时间Before配置的时间时，才会转发到用户微服务
            # 目前配置不会进该路由配置，所以返回404
            # 将时间改成 > now的时间，则访问localhost:8040/** -> msb-user/**
            # eg. 访问http://localhost:8040/users/1 -> msb-user/users/1
            - Before=2018-01-20T17:42:47.789-07:00[America/Denver]
```

3.3 Between路由断言工厂

规则：

该断言工厂的参数是两个 UTC 格式的时间。其会将请求访问到 Gateway 的时间与这两个参数时间相比，若请求时间在

配置：

```
spring:
  cloud:
    gateway:
```

```
    routes:
      - id: between_route
        uri: lb://msb-user
        predicates:
          # 当且仅当请求时的时间Between配置的时间时，才会转发到用户微服务
          # 因此，访问localhost:8040/** -> msb-user/**
          # eg. 访问http://localhost:8040/users/1 -> msb-user/users/1
          - Between=2017-01-20T17:42:47.789-07:00[America/Denver], 2027-01-21T17:42:47.789-07:00[Americ
```

3.4 Cookie路由断言工厂

规则：

该断言工厂中包含两个参数，分别是 cookie 的 key 与 value。当请求中携带了指定 key与 value 的 cookie 时，匹配成

配置：

```
spring:
  cloud:
    gateway:
      routes:
        - id: cookie_route
          uri: lb://msb-user
          predicates:
            # 当且仅当带有名为somecookie，并且值符合正则ch.p的Cookie时，才会转发到用户微服务
            # 如Cookie满足条件，则访问http://localhost:8040/** -> msb-user/**
            # eg. 访问http://localhost:8040/users/1 -> msb-user/users/1
            - Cookie=somecookie, ch.p
```

3.5 Header路由断言工厂

规则：

该断言工厂中包含两个参数，分别是请求头 header 的 key 与 value。当请求中携带了指定 key 与 value 的 header 时

配置：

```
spring:
  cloud:
    gateway:
      routes:
        - id: header_route
          uri: lb://msb-user
```

```
        predicates:
          # 当且仅当带有名为X-Request-Id，并且值符合正则\d+的Header时，才会转发到用户微服务
          # 如Header满足条件，则访问http://localhost:8040/** -> msb-user/**
          # eg. 访问http://localhost:8040/users/1 -> msb-user/users/1
          - Header=X-Request-Id, \d+
```

3.6 Host路由断言工厂

规则：

该断言工厂中包含的参数是请求头中的 Host 属性。当请求中携带了指定的 Host 属性值时，匹配成功，断言为 true。

配置：

```
spring:
  cloud:
    gateway:
      routes:
        - id: host_route
          uri: lb://user-center
          predicates:
            # 当且仅当名为Host的Header符合**.somehost.org或**.anotherhost.org时，才会转发用户微服务
            # 如Host满足条件，则访问http://localhost:8040/** -> user-center/**
            # eg. 访问http://localhost:8040/users/1 -> user-center/users/1
            - Host=**.somehost.org,**.anotherhost.org
```

3.7 Method路由断言工厂

规则：

该断言工厂用于判断请求是否使用了指定的请求方法，是 POST，还是 GET 等。当请求中使用了指定的请求方法时，匹

配置：

```
spring:
  cloud:
    gateway:
      routes:
        - id: method_route
          uri: lb://msb-user
          predicates:
            # 当且仅当HTTP请求方法是GET时，才会转发用户微服务
            # 如请求方法满足条件，访问http://localhost:8040/** -> msb-user/**
```

```
        # eg. 访问http://localhost:8040/users/1 -> msb-user/users/1
        - Method=GET
```

3.8 Path路由断言工厂

规则：

```
该断言工厂用于判断请求路径中是否包含指定的uri。若包含，则匹配成功，断言为true，此时会将该匹配上的 uri 拼接
```

配置：

```
spring:
  cloud:
    gateway:
      routes:
        - id: path_route
          uri: lb://msb-user
          predicates:
            # 当且仅当访问路径是/users/*或者/some-path/**，才会转发用户微服务
            # segment是一个特殊的占位符，单层路径匹配
            # eg. 访问http://localhost:8040/users/1 -> msb-user/users/1
            - Path=/users/{segment},/some-path/**
```

3.9 Query路由断言工厂

规则：

```
该断言工厂用于从请求中查找指定的请求参数。其可以只查看参数名称，也可以同时查看参数名与参数值。当请求中包
```

参数：

param 请求参数的key值
regexp 请求参数的值，配置的值是 Java中的 正则表达式形式。

配置：

示例1：

```
spring:
  cloud:
    gateway:
      routes:
        - id: query_route
          uri: lb://msb-user
```

```
        predicates:
          # 当且仅当请求带有baz的参数，才会转发到用户微服务
          # eg. 访问http://localhost:8040/users/1?baz=xx -> msb-user的/users/1
          - Query=baz
```

示例2：

```
spring:
  cloud:
    gateway:
      routes:
        - id: query_route
          uri: lb://msb-user
          predicates:
            # 当且仅当请求带有名为foo的参数，且参数值符合正则ba.，才会转发到用户微服务
            # eg. 访问http://localhost:8040/users/1?baz=baz -> msb-user的/users/1?baz=baz
            - Query=foo, ba.
```

3.10 RemoteAddr路由断言工厂

规则：

```
该断言工厂用于判断请求提交的所要访问的 IP 地址是否在断言中指定的 IP 范围。当请求中的 IP 在指定范围时，匹配成
```

配置：

```
spring:
  cloud:
    gateway:
      routes:
        - id: remoteaddr_route
          uri: lb://msb-user
          predicates:
            # 当且仅当请求IP是192.168.1.1/24网段，例如192.168.1.10，才会转发到用户微服务
            # eg. 访问http://localhost:8040/users/1 -> msb-user的/users/1
            - RemoteAddr=192.168.1.1/24
```

> 如果我们的 Spring Cloud Gateway是位于代理后面，那么获取到 远程地址可能不正确，此时我们可以自己编写一个 RemoteAddressResolver来解决。

3.11 Weight路由断言工厂

规则：

该断言工厂中包含两个参数，分别是用于表示组 group，与权重 weight。对于同一组中的多个 uri 地址，路由器会根

配置：

```
spring:
  cloud:
    gateway:
      routes:
      - id: weight_high
        uri: https://weighthigh.org
        predicates:
        - Weight=group1, 8
      - id: weight_low
        uri: https://weightlow.org
        predicates:
        - Weight=group1, 2
```

group 组，权重根据组来计算
weight 权重值，是一个 Int 的值

3.12 XForwarded Remote Addr路由断言工厂

规则：

这可用于反向代理，如负载平衡器或web应用程序防火墙，其中只有当请求来自这些反向代理使用的IP地址的受信任列表时，才应允许该请求。

配置：

```
spring:
  cloud:
    gateway:
      routes:
      - id: xforwarded_remoteaddr_route
        uri: https://example.org
        predicates:
        - XForwardedRemoteAddr=192.168.1.1/24
```

4、谓词使用

4.1 betwee

例如：在某个时间段下单我们送5元红包

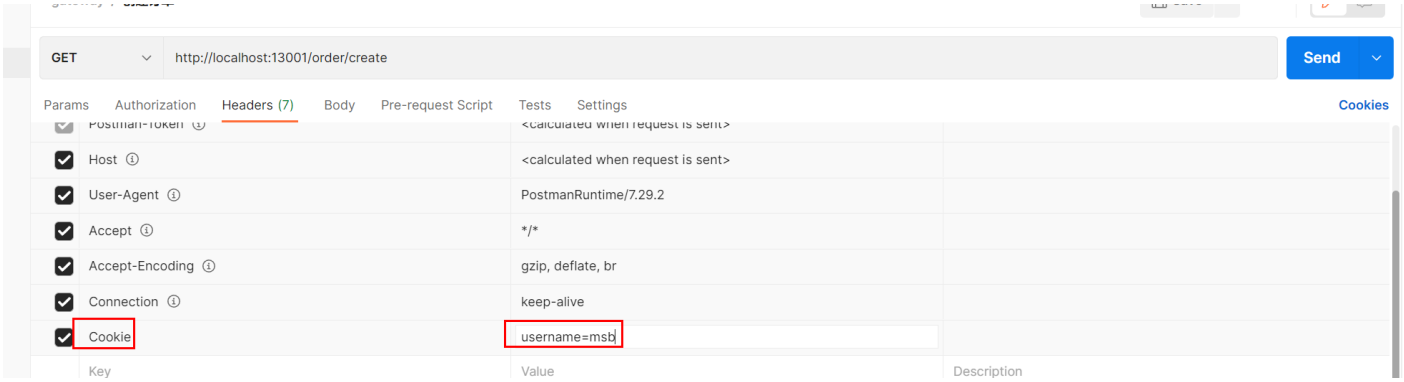
```
spring:
  cloud:
    gateway:
      routes:
        #路由ID全局唯一
        - id: create_order
          #目标微服务的请求地址和端口
          uri: http://localhost:8001
          predicates:
            #请求某个接口在这个时间段是有效的
            - Between=2022-01-30T11:59:59+08:00[Asia/Shanghai], 2022-02-01T11:59:59+08:00[Asia/Shanghai]
```

获取某个时区的指定时间

```
public class TestTime {
    public static void main(String[] args) {
        // 获取某个时区当前时间
        ZonedDateTime zonedDateTime= ZonedDateTime.now(ZoneId.of("Asia/Shanghai"));
        // 获取指定时区开始时间
        ZonedDateTime startTime = ZonedDateTime.of(2022, 1, 30,
                11, 59, 59, 0, ZoneId.of("Asia/Shanghai"));
        // 获取指定时区结束时间
        ZonedDateTime endTime = ZonedDateTime.of(2022, 2, 1,
                11, 59, 59, 0, ZoneId.of("Asia/Shanghai"));
        System.out.println("获取指定时区当前时间：" +zonedDateTime);
        System.out.println("获取指定时区开始时间："+ startTime);
        System.out.println("获取指定时区结束时间：" + endTime);
    }

}
```

4.2 Cookie匹配

```
spring:
  cloud:
    gateway:
      routes:
        - id: cookie_route
          uri: lb://msb-user
          predicates:
          #Cookie配置
            - Cookie=username, msb
```



4.3 head匹配

```
spring:
  cloud:
    gateway:
      routes:
        - id: header_route
          uri: lb://msb-user
          predicates:
            # 当且仅当带有名为X-Request-Id，并且值符合正则\d+（单个或者多个数字）的Header时，才会转发到用户
            # 如Header满足条件，则访问http://localhost:8040/** -> msb-user/**
            # eg. 访问http://localhost:8040/users/1 -> msb-user/users/1
            - Header=X-Request-Id, \d+
```

4.4 Weight路由断言工厂

我们可以将配置放到nacos配置中心，使用这个路由谓词实现灰度发布

```
spring:
  cloud:
    gateway:
      routes:
        - id: weight_high
          uri: http://localhost:8081
          predicates:
```

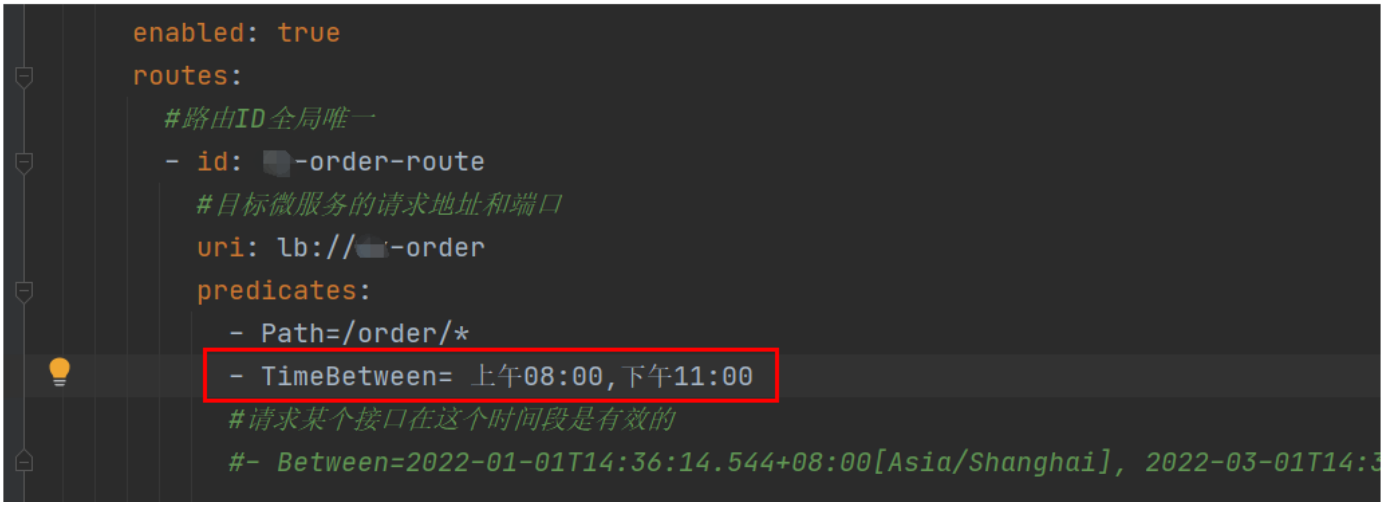
```
        - Weight=group1, 8
        - Path=/order/**
      - id: weight_low
        uri: http://localhost:8082
        predicates:
        - Weight=group1, 2
        - Path=/order/**
```

5、自定义谓词

比如我们有这样一个需求：

- 限制08:00 - 23:00 才能访问

我们自定义一下配置：假设这个配置是这样配置的 - TimeBetween= 08:00，23:00



自定义路由断言工厂需要继承 AbstractRoutePredicateFactory 类，重写 apply 方法的逻辑。在 apply 方法中可以通

注意： 命名需要以 RoutePredicateFactory 结尾

```
@Component
public class TimeBetweenRoutePredicateFactory extends AbstractRoutePredicateFactory<TimeBetweenConfi

    public TimeBetweenRoutePredicateFactory() {
        super(TimeBetweenConfig.class);
    }

    @Override
    public Predicate<ServerWebExchange> apply(TimeBetweenConfig config) {
        LocalTime start = config.getStart();
```

```
        LocalTime end = config.getEnd();
        return new Predicate<ServerWebExchange>() {
            @Override
            public boolean test(ServerWebExchange serverWebExchange) {
                LocalTime now = LocalTime.now();
                return now.isAfter(start) && now.isBefore(end);
            }
        };
    }

    @Override
    public List<String> shortcutFieldOrder() {
        return Arrays.asList("start","end");
    }
}
```

1. 继承

```
AbstractRoutePredicateFactory<>
```

需要用到泛型，这里的泛型是需要定义一个配置类来给配置参数，下面是配置类，配置开始时间和结束时间

```
@Data
public class TimeBetweenConfig {
    private LocalTime start;
    private LocalTime end;
}
```

2. 重写方法

```
public List<String> shortcutFieldOrder() {
        return Arrays.asList("start","end");
    }
```

用来映射参数，asList("配置文件的第1个参数","配置文件的第2个参数","配置文件的第3个参数","配置文件的第4个参数"....)，接给配置类，配置类得到参数后里面的参数有值然后在重写的

```
public Predicate<ServerWebExchange> apply(TimeBetweenConfig config) {
        LocalTime start = config.getStart();
        LocalTime end = config.getEnd();
        return new Predicate<ServerWebExchange>() {
            @Override
            public boolean test(ServerWebExchange serverWebExchange) {
                LocalTime now = LocalTime.now();
```

```
            return now.isAfter(start) && now.isBefore(end);
        }
    }
```

中进行谓词配置，实现自定义谓词工厂

3. 利用JDK8特性查看时间格式

```
public static void main(String[] args) {
    DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT);
    System.out.println(dateTimeFormatter.format(LocalTime.now()));
}
```

三、过滤器工厂（ GatewayFilter Factories ）配置

SpringCloudGateway 内置了很多的过滤器工厂，我们通过一些过滤器工厂可以进行一些业务逻辑处理器，比如添加剔除响应头，添加去除参数等

官网：https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#gatewayfilter-factories

1、内置过滤器

1.1 AddRequestHeader 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_request_header_route
        uri: https://example.org
        filters:
        - AddRequestHeader=X-Request-Foo, Bar
```

为原始请求添加名为 X-Request-Foo ，值为 Bar 的请求头。

1.2 AddRequestParameter 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
```

```
      - id: add_request_parameter_route
        uri: https://example.org
        filters:
        - AddRequestParameter=foo, bar
```

为原始请求添加请求参数 foo=bar

1.3 AddResponseHeader 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_response_header_route
        uri: https://example.org
        filters:
        - AddResponseHeader=X-Response-Foo, Bar
```

添加名为 X-Request-Foo ，值为 Bar 的响应头。

1.4 DedupeResponseHeader 过滤工厂

> Spring Cloud Greenwich SR2提供的**新特性**，低于这个版本无法使用。
>
> **强烈建议**阅读一下类org.springframework.cloud.gateway.filter.factory.DedupeResponseHeaderGatewayFilterFactory上的注释，比官方文档写得还好。

```
spring:
  cloud:
    gateway:
      routes:
      - id: dedupe_response_header_route
        uri: https://example.org
        filters:
        - DedupeResponseHeader=Access-Control-Allow-Credentials Access-Control-Allow-Origin, RETAIN_FIR
```

剔除重复的响应头。

举个例子：

我们在Gateway以及微服务上都设置了CORS（解决跨域）header，如果不做任何配置，请求 -> 网关 -> 微服务，获得的响应就是这样的：

```
Access-Control-Allow-Credentials: true, true
Access-Control-Allow-Origin: https://www.msbedu.com, https://www.msbedu.com
```

也就是Header重复了。要想把这两个Header去重，只需设置成如下即可。

```
filters:
- DedupeResponseHeader=Access-Control-Allow-Credentials Access-Control-Allow-Origin
```

也就是说，想要去重的Header如果有多个，用空格分隔即可；

去重策略：

```
RETAIN_FIRST: 默认值，保留第一个值
RETAIN_LAST: 保留最后一个值
RETAIN_UNIQUE: 保留所有唯一值，以它们第一次出现的顺序保留
```

1.5 PrefixPath 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: prefixpath_route
        uri: https://example.org
        filters:
        - PrefixPath=/mypath
```

为匹配的路由添加前缀。例如：访问 ${GATEWAY_URL}/hello 会转发到 https://example.org/mypath/hello

1.6 PreserveHostHeader 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: preserve_host_route
        uri: https://example.org
        filters:
        - PreserveHostHeader
```

如果不设置，那么名为 Host 的Header由Http Client控制；如果设置了，那么会设置一个请求属性（preserveHostHeader=true），路由过滤器会检查从而去判断是否要发送原始的、名为Host的Header。

1.7 RequestRateLimiter 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: requestratelimiter_route
        uri: https://example.org
        filters:
        - name: RequestRateLimiter
          args:
            redis-rate-limiter.replenishRate: 10
            redis-rate-limiter.burstCapacity: 20
```

1.8 RedirectTo 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: prefixpath_route
        uri: https://example.org
        filters:
        # 配置成HTTP状态码, URL的形式
        - RedirectTo=302, http://www.msb.com
```

- HTTP状态码应该是HTTP状态码300序列，例如301
- URL必须是合法的URL，并且该值会作为名为 Location 的Header。

上面配置表达的意思是： ${GATEWAY_URL}/hello 会重定向到 https://ecme.org/hello ，并且携带一个 Location:http://www.msb.com 的Header。

1.9 RemoveHopByHopHeadersFilter 过滤工厂

```
spring.cloud.gateway.filter.remove-hop-by-hop.headers: Connection,Keep-Alive
```

移除转发请求的Header，多个用，分隔。默认情况下，移除如下Header。这些Header是由 IETF 组织规定的。

- Connection
- Keep-Alive
- Proxy-Authenticate
- Proxy-Authorization
- TE
- Trailer
- Transfer-Encoding
- Upgrade

1.10 RemoveRequestHeader 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: removerequestheader_route
        uri: https://example.org
        filters:
        - RemoveRequestHeader=X-Request-Foo
```

原始请求删除名为 X-Request-Foo 的请求头。

1.11 RemoveResponseHeader 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: removeresponseheader_route
        uri: https://example.org
        filters:
        - RemoveResponseHeader=X-Response-Foo
```

删除名为 X-Request-Foo 的响应头。

1.12 RewritePath 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: rewritepath_route
        uri: https://example.org
        predicates:
        - Path=/foo/**
        filters:
        # 配置成原始路径正则, 重写后的路径的正则
        - RewritePath=/foo/(?<segment>.*), /$\{segment}
```

重写请求路径。如上配置，访问 /foo/bar 会将路径改为 /bar 再转发，也就是会转发到 https://example.org/bar 。需要注意的是，由于YAML语法，需用 $\ 替换 $ 。

1.13 RewriteResponseHeader 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: rewriteresponseheader_route
        uri: https://example.org
        filters:
        - RewriteResponseHeader=X-Response-Foo, password=[^&]+, password=***
```

如果名为 X-Response-Foo 的响应头的内容是 /42 ? user=ford&password=omg! what&flag=true，则会被修改为 /42?user=ford&password=***&flag=true。

1.14 SaveSession 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: save_session
        uri: https://example.org
        predicates:
        - Path=/foo/**
        filters:
        - SaveSession
```

在转发到后端微服务请求之前，强制执行 WebSession::save 操作。用在那种像 Spring Session 延迟数据存储（笔者注：数据不是立刻持久化）的，并希望在请求转发前确保session状态保存情况。

如果你将 Spring Secutiry于 Spring Session集成使用，并想确保安全信息都传到下游机器，你就需要配置这个filter。

1.15 SecureHeaders 过滤工厂

添加一系列起安全作用的响应头。Spring Cloud Gateway参考了这篇博客的建议：https://blog.appcanary.com/2017/http-security-headers.html

默认会添加如下Header（包括值）：

- X-Xss-Protection:1; mode=block
- Strict-Transport-Security:max-age=631138519
- X-Frame-Options:DENY
- X-Content-Type-Options:nosniff
- Referrer-Policy:no-referrer
- Content-Security-Policy:default-src 'self' https:; font-src 'self' https: data:; img-src 'self' https: data:; object-src 'none'; script-src https:; style-src 'self' https: 'unsafe-inline'
- X-Download-Options:noopen
- X-Permitted-Cross-Domain-Policies:none

如果你想修改这些Header的值，可使用如下配置：

前缀：spring.cloud.gateway.filter.secure-headers

上面的header对应的后缀：

- xss-protection-header
- strict-transport-security
- frame-options
- content-type-options
- referrer-policy
- content-security-policy
- download-options
- permitted-cross-domain-policies

例如：spring.cloud.gateway.filter.secure-headers.xss-protection-header: 你想要的值

如果想禁用某些Header，可使用如下配置：spring.cloud.gateway.filter.secure-headers.disable ，多个用 , 分隔。例如：spring.cloud.gateway.filter.secure-headers.disable=frame-options, download-options 。

1.16 SetPath 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: setpath_route
        uri: https://example.org
        predicates:
        - Path=/foo/{segment}
        filters:
        - SetPath=/{segment}
```

采用路径 template参数，通过请求路径的片段的模板化，来达到操作修改路径的母的，运行多个路径片段模板化。

如上配置，访问 ${GATEWAY_PATH}/foo/bar ，则对于后端微服务的路径会修改为 /bar 。

1.17 SetResponseHeader 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: setresponseheader_route
        uri: http://example.org
        filters:
        - SetResponseHeader=X-Response-Foo, Bar
```

如果后端服务响应带有名为 X-Response-Foo 的响应头，则将值改为替换成 Bar 。

1.18 SetStatus 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: setstatusstring_route
        uri: http://example.org
```

```
        filters:
        - SetStatus=BAD_REQUEST
      - id: setstatusint_route
        uri: http://example.org
        filters:
        - SetStatus=401
```

修改响应的状态码，值可以是数字，也可以是字符串。但一定要是Spring HttpStatus 枚举类中的值。如上配置，两种方式都可以返回HTTP状态码401。

1.19 StripPrefix 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: nameRoot
        uri: http://nameservice
        predicates:
        - Path=/name/**
        filters:
        - StripPrefix=2
```

数字表示要截断的路径的数量。如上配置，如果请求的路径为 /name/bar/foo ，则路径会修改为 /foo ，也就是会截断2个路径。

1.20 Retry 过滤工厂

```
spring:
  cloud:
    gateway:
      routes:
      - id: retry_test
        uri: http://localhost:8080/flakey
        predicates:
        - Host=*.retry.com
        filters:
        - name: Retry
          args:
            retries: 3
            statuses: BAD_GATEWAY
```

针对不同的响应做重试，可配置如下参数：

- retries: 重试次数
- statuses: 需要重试的状态码，取值在 org.springframework.http.HttpStatus 中
- methods: 需要重试的请求方法，取值在 org.springframework.http.HttpMethod 中
- series: HTTP状态码系列，取值在 org.springframework.http.HttpStatus.Series 中

1.21 RequestSize 过滤工厂

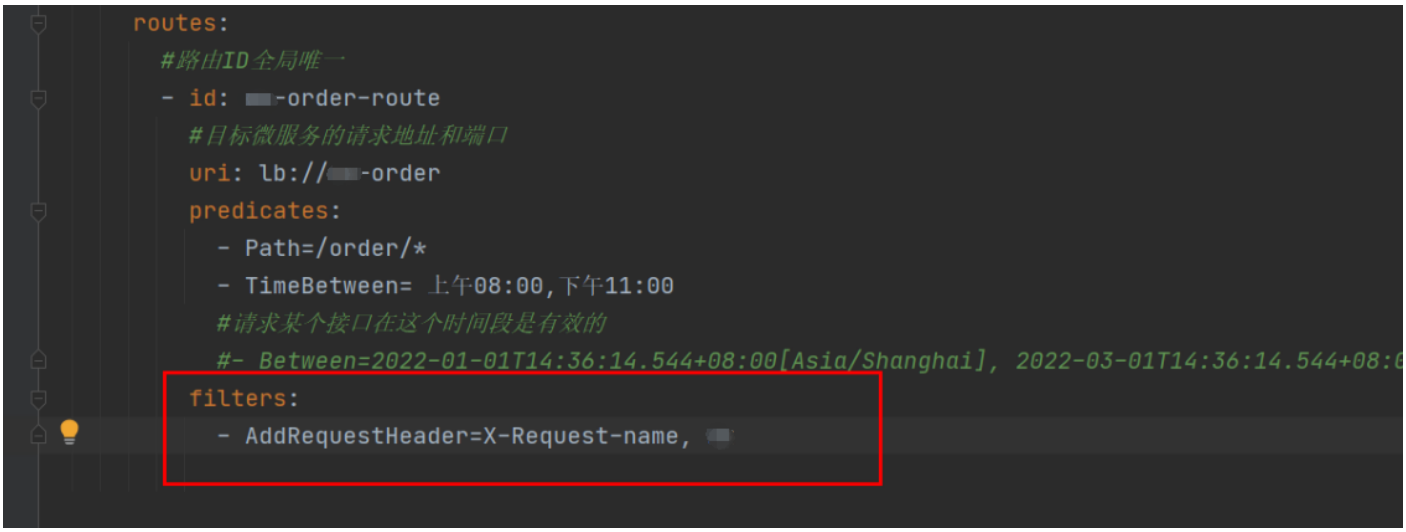
```
spring:
  cloud:
    gateway:
      routes:
      - id: request_size_route
      uri: http://localhost:8080/upload
      predicates:
      - Path=/upload
      filters:
      - name: RequestSize
        args:
          # 单位字节
          maxSize: 5000000
```

为后端服务设置收到的最大请求包大小。如果请求大小超过设置的值，则返回 413 Payload Too Large 。默认值是5M

1.22 默认过滤器工厂

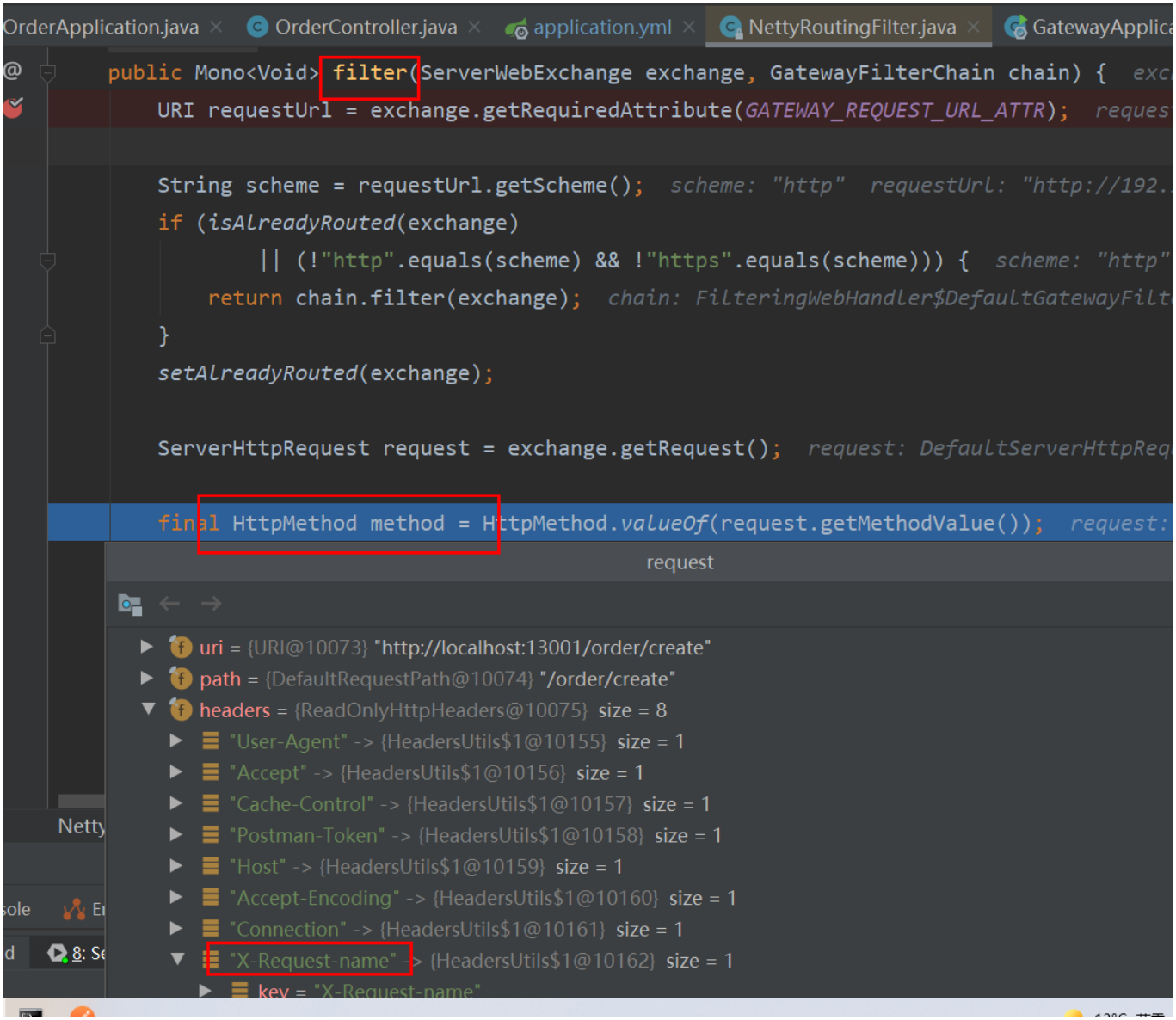
```
spring:
  cloud:
    gateway:
      default-filters:
      - AddResponseHeader=X-Response-Default-Foo, Default-Bar
      - PrefixPath=/httpbin
```

2、AddRequestHeader 过滤工厂



- 断点打在 org.springframework.cloud.gateway.filter.NettyRoutingFilter#filter ，就可以调试Gateway转发的具体细节了
- 添加如下配置，可观察到一些请求细节：

```
logging:
  level:
    org.springframework.cloud.gateway: trace
    org.springframework.http.server.reactive: debug
    org.springframework.web.reactive: debug
    reactor.ipc.netty: debug
```



3、自定义过滤器

3.1 过滤器生命周期

- pre : Gateway转发请求之前
- post : Gateway转发请求之后

3.2 自定义过滤器工厂的方式

- 自定义过滤器工厂-方式1

  继承: AbstractGatewayFilterFactory

  参考示例:org.springframework.cloud.gateway.filter.factory.
  RequestSizeGatewayFilterFactory

```
spring:
  cloud:
    gateway:
      routes:
      filters:
      - name: RequestSize
        args:
          # 单位字节
          maxSize: 5000000
```

- 自定义过滤器工厂-方式2

  继承: AbstractNaeValueGatewayFilterFactory 【AbstractNaeValueGatewayFilterFactory 其实继承了AbstractGatewayFilterFactory，所以AbstractNaeValueGatewayFilterFactory 对AbstractGatewayFilterFactory的简化】

  参考示例:org.springframework.cloud.gateway.filter.factory.AddRequestHeaderGatewayFilterFactory

```
spring:
  cloud:
    gateway:
      routes:
      - id: add_response_header_route
        uri: https://example.org
        filters:
        - AddResponseHeader=X-Response-Foo, Bar
```

3.3 核心API

- exchange.getRequest().mutate().xxx//修改request
- exchange.mutate().xxx//修改exchange
- chain.filter(exchange)//传递给下一个过滤器处理
- exchange.getResponse()//拿到响应

```
// 这里的过滤器和我们以前的过滤器还不一样，tomcat过滤器是属于HttpServletRequst\HttpServletResponse
// 我们这里底层是基于Netty的，它是将我们的请求和响应封装到我们的ServerWebExchange然后进行出里这里是对
```

```
*/
public class AddResponseHeaderGatewayFilterFactory
        extends AbstractNameValueGatewayFilterFactory {

    @Override
    public GatewayFilter apply(NameValueConfig config) {
        return new GatewayFilter() {
            @Override
            public Mono<Void> filter(ServerWebExchange exchange,
                    GatewayFilterChain chain) {
                String value = ServerWebExchangeUtils.expand(exchange, config.getValue());
                exchange.getResponse().getHeaders().add(config.getName(), value);

                return chain.filter(exchange);
            }
```

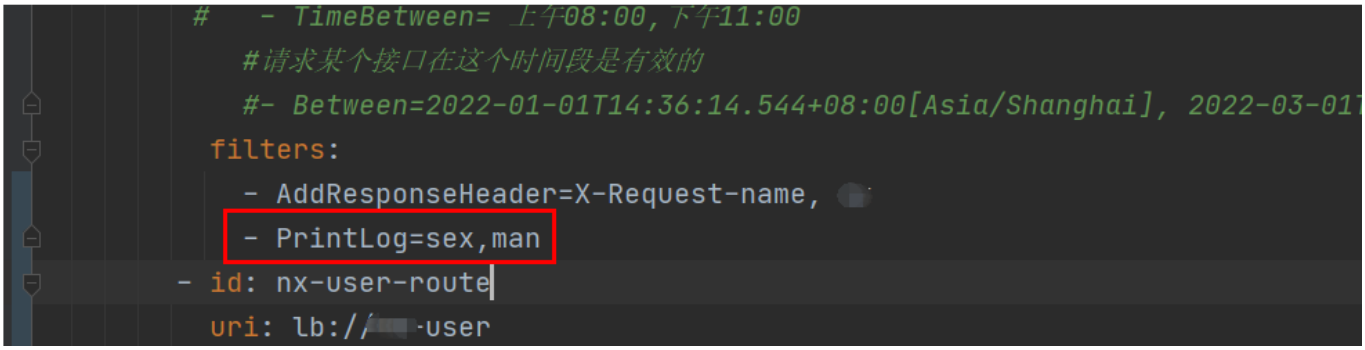
3.4 编写一个过滤器工厂

记录日志

- 编写的类一定以GatewayFilterFactory结尾
- 代码

```
@Slf4j
@Component
public class PrintLogGatewayFilterFactory extends AbstractNameValueGatewayFilterFactory {
    @Override
    public GatewayFilter apply(NameValueConfig config) {
        // 放到这里应用启动的时候会执行两次次，这是一个坑，需要放到方法里面
       //   log.info("打印请求信息:{}:{}",config.getName(),config.getValue());
        return new GatewayFilter() {

            @Override
            public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
                log.info("打印请求信息:{}:{}",config.getName(),config.getValue());
                // 获取request就可以进行修改了
                ServerHttpRequest modifiedRequest = exchange.getRequest().mutate().build();
                // 设置请求值
//                ServerHttpRequest modifiedRequest = exchange.getRequest().mutate()
//                        .header(config.getName(), value).build();
                ServerWebExchange modifiedExchange = exchange.mutate().request(modifiedRequest).build(
                return chain.filter(modifiedExchange);
            }
        };
    }
}
```

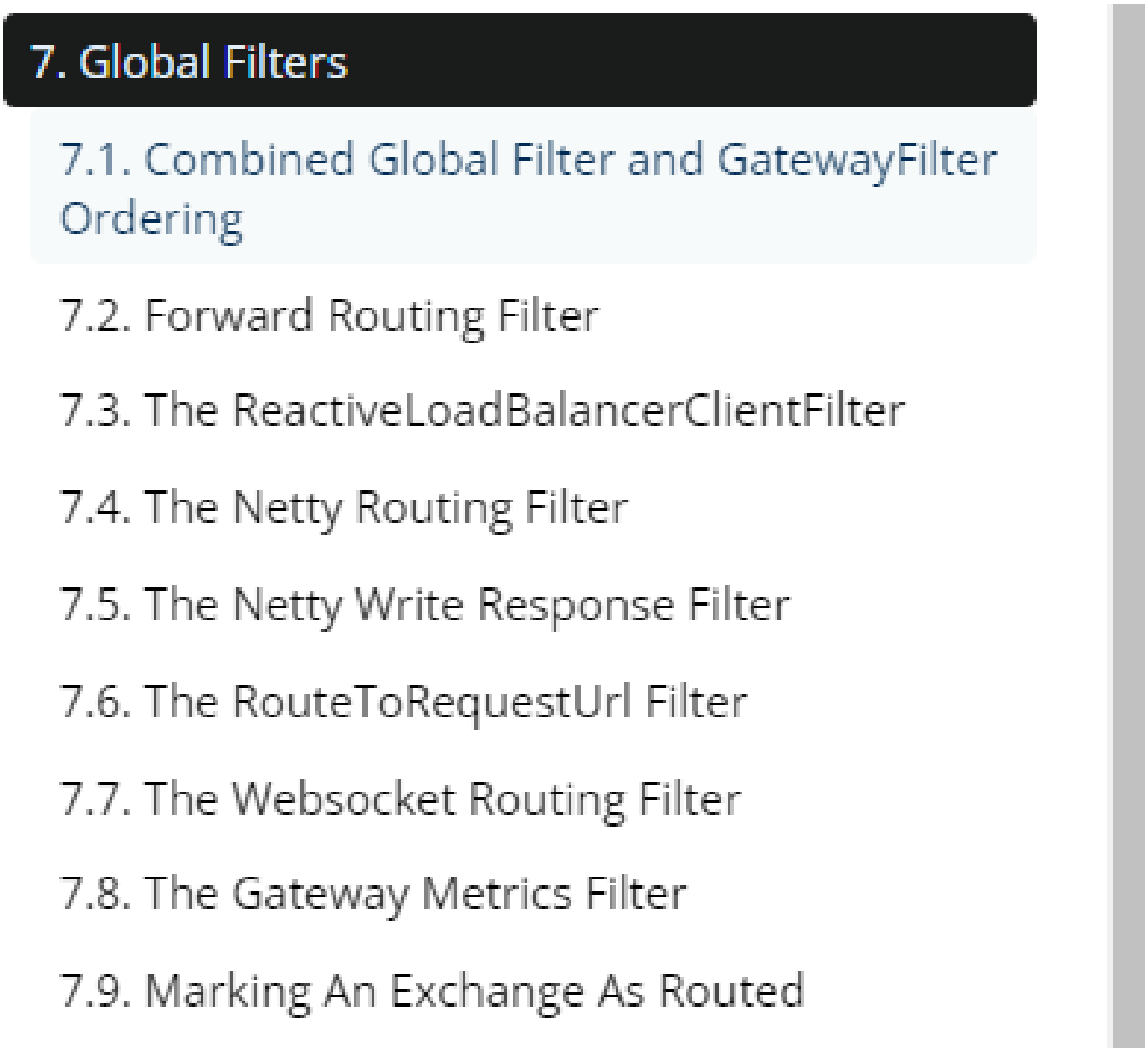
- 配置



4、全局过滤器

前面的 GatewayFilter 工厂是在某一特定路由策略中设置的，仅对这一种路由生效。若要使某些过滤效果应用到所有路由策略中，就可以将该 GatewayFilter 工厂定义在全局Filters中。修改 gateway 工程配置文件。

https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#global-filters



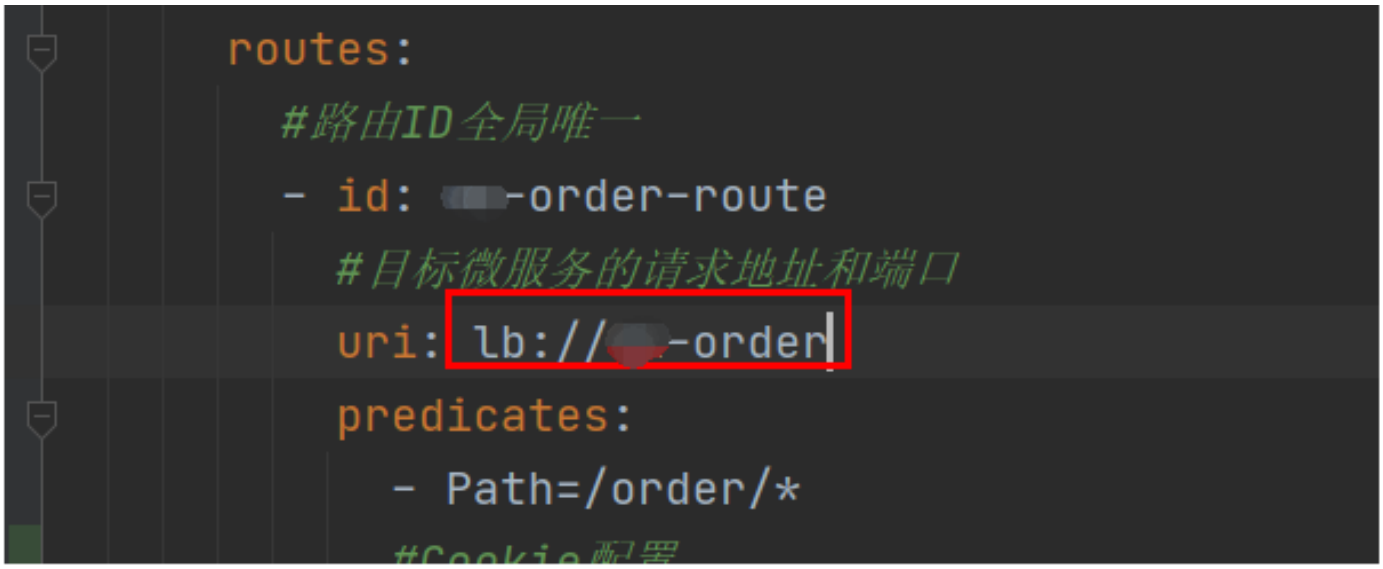
GlobalFilter 接口和 GatewayFilter 有一样的接口定义，只不过， GlobalFilter 会作用于所有路由。

官方声明：GlobalFilter的接口定义以及用法在未来的版本可能会发生变化。

4.1 LoadBalancerClientFilter

LoadBalancerClientFilter 会查看exchange的属性 ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR 的 值（一个URI），如果该值的scheme是 lb，比如：lb://msb-user，它将会使用Spring Cloud的LoadBalancerClient 来 将 msb-user解析成实际的host和port，并替换掉 ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR 的内 容。

其实就是用来整合负载均衡器Ribbon的



4.2 自定义过滤器

1. 实现GlobalFilter接口
2. 增加Order注解，值越小越靠前
3. @Component

```
@Slf4j
@Component
public class IPAddressStatisticsFilter implements GlobalFilter, Ordered {
```

```
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        InetSocketAddress host = exchange.getRequest().getHeaders().getHost();
        if (host == null || host.getHostName() == null) {
            exchange.getResponse().setStatusCode(HttpStatus.BAD_REQUEST);
            return exchange.getResponse().setComplete();
        }
        String hostName = host.getHostName();
        AtomicInteger count = IpCache.CACHE.getOrDefault(hostName, new AtomicInteger(0));
        count.incrementAndGet();
        IpCache.CACHE.put(hostName, count);
        log.info("IP地址：" + hostName + ",访问次数：" + count.intValue());
        return chain.filter(exchange);
    }

    @Override
    public int getOrder() {
        return -1;
    }
}
```

```
public class IpCache {
    public static final Map<String, AtomicInteger> CACHE = new ConcurrentHashMap<>();
}
```

四、源码分析