**【Spring Cloud Gateway】**

**【主要内容】**

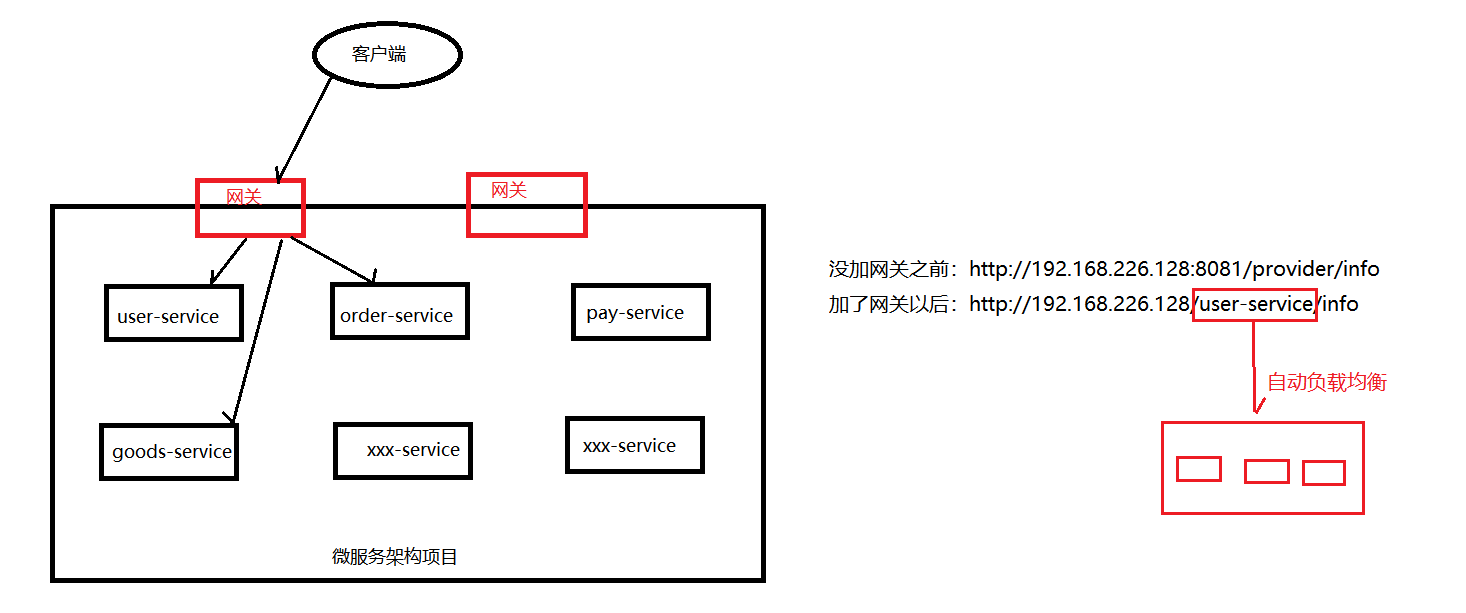
1. 网关简介
2. Gateway简介
3. Gateway快速入门
4. Gateway集群
5. Gateway断言
6. Gateway过滤
7. Gateway实战

**【学习目标】**

|  |  |
| --- | --- |
| 知识点 | 要求 |
| 网关简介 | 了解 |
| Gateway简介 | 掌握 |
| Gateway快速入门 | 掌握 |
| Gateway集群 | 掌握 |
| Gateway断言 | 了解 |
| Gateway过滤 | 掌握 |
| Gateway实战 | 掌握 |

# 什么是网关

网关是**微服务最边缘的服务**，**直接暴露给用户，用来做用户和微服务的桥梁**



1. 没有网关：客户端直接访问我们的微服务，会需要在客户端配置很多的ip：port，如果user-service并发比较大，则无法完成负载均衡
2. 有网关：客户端访问网关，网关来访问微服务，这样只需要使用**服务名称即可访问微服务**，可以实现负载均衡，可以实现token拦截，权限验证，限流等操作

# Spring Cloud Gateway简介

它是Spring Cloud**官方**提供的用来取代**zuul（netflix）**的新一代网关组件

（zuul：1.0 , 2.0 ，zuul的本质，一组过滤器，根据自定义的过滤器顺序来执行，本质就是servlet）

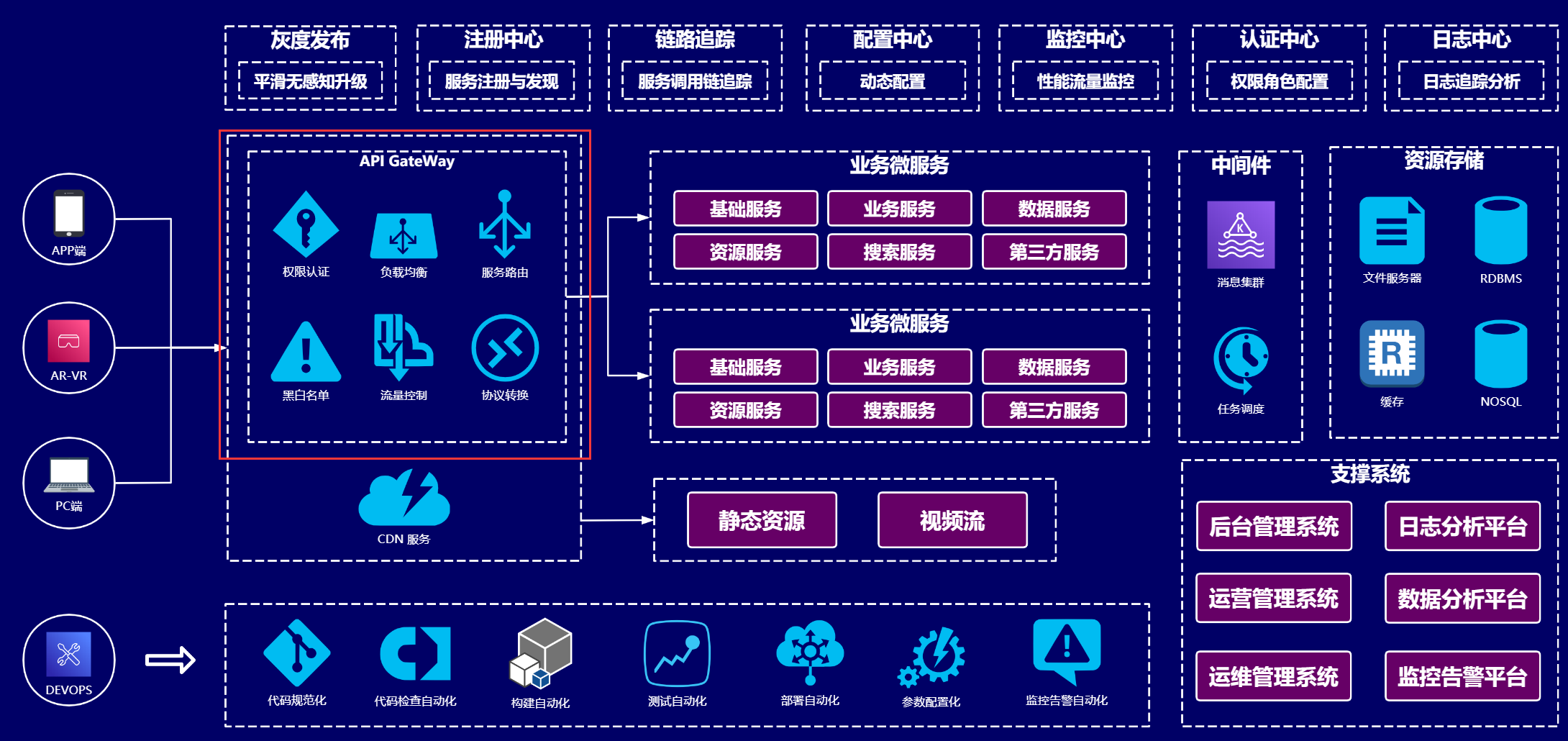
它基于spring5.x，springboot2.x和ProjectReactor等技术。

它的目地是让路由更加简单，灵活，还提供了一些强大的过滤器功能，例如：熔断、限流、重试等

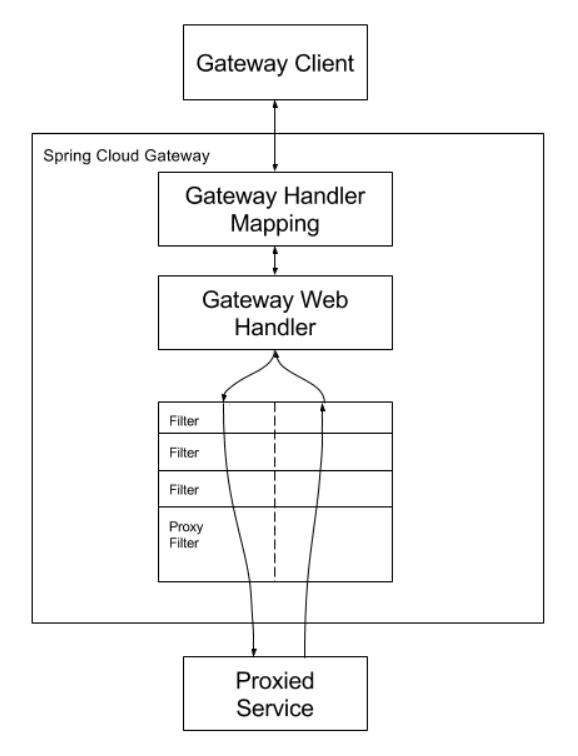
SpringCloud Gateway作为Spring Cloud生态的网关，目标是替代Zuul，在SpringCloud2.0以上的版本中，没有对新版本的zuul2.0以上的最新高性能版本进行集成，仍然还是使用的zuul1.x[可以看项目依赖找到]非Reactor模式的老版本。而为了提升网关的性能，SpringCloud Gateway是基于webFlux 框架实现的，而webFlux框架底层则使用了高性能的Reactor模式通信框架的Netty



网关具体在微服务架构中的哪个位置



# Spring Cloud Gateway工作流程



客户端向springcloud 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。

Handler 再通过指定的过滤器来将请求发送到我们实际的服务的业务逻辑，然后返回。 过滤器之间用虚线分开是因为过滤器可能会在发送爱丽请求之前【pre】或之后【post】执行业务逻辑，对其进行加强或处理。

Filter在 【pre】 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】 类型的过滤器中可以做响应内容、响应头的修改、日志的输出，流量监控等有着非常重要的作用。

**总结：Gateway 的核心逻辑也就是 路由转发 + 执行过滤器链**

# Spring Cloud Gateway三大核心概念

## Route(路由)（重点）

**路由信息的组成：**

**由一个ID、一个目的URL、一组断言工厂、一组Filter组成。**

**如果路由断言为真，说明请求URL和配置路由匹配。**

## Predicate(断言)（判断 true false）

**Java 8中的断言函数。**

**Spring Cloud Gateway中的断言函数输入类型是Spring 5.0框架中的ServerWebExchange。Spring Cloud Gateway的断言函数允许开发者去定义匹配来自于Http Request中的任何信息比如请求头和参数。**

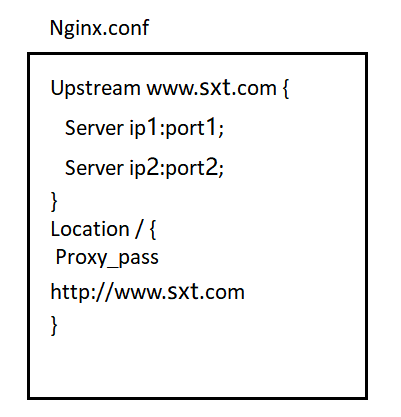
## Filter(过滤) (重点)

**一个标准的Spring WebFilter。**

**Spring Cloud Gateway中的Filter分为两种类型的Filter，分别是Gateway Filter和Global Filter。过滤器Filter将会对请求和响应进行修改处理。**

# Nginx和Gateway的区别

Nginx在做路由，负载均衡，限流之前，都有修改nginx.conf的配置文件，把需要负载均衡，路由，限流的规则加在里面。Eg:使用nginx 做tomcat的负载均衡

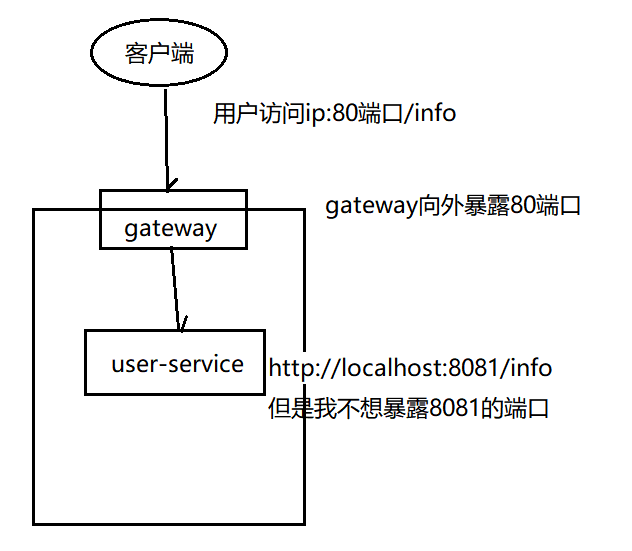


但是gateway不同，gateway自动的负载均衡和路由，gateway和eureka 高度集成，实现自动的路由，和Ribbon 结合，实现了负载均衡（lb），gateway也能轻易的实现限流和权限验证。

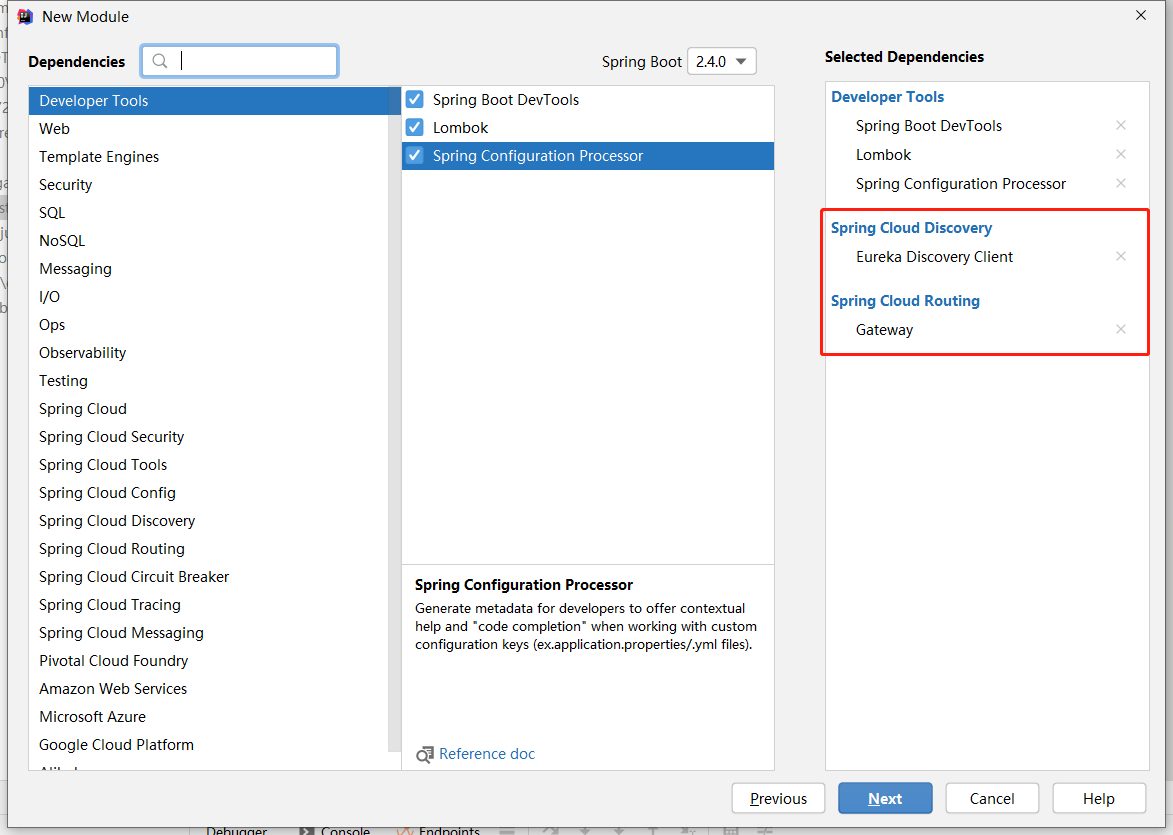
Nginx比gateway的性能高一点。

# Gateway 快速入门

## 本次访问流程



## 新建项目选择依赖(不要选web)



## 修改启动类

|  |
| --- |
| @SpringBootApplication @EnableEurekaClient //网关也是eureka的客户端 public class Gateway80Application {   public static void main(String[] args) {  SpringApplication.*run*(Gateway80Application.class, args);  }  } |

## 修改配置文件

|  |
| --- |
| server:  port: 80 spring:  application:  name: gateway-80  **cloud:  gateway:  enabled: true #开启网关，默认是开启的  routes: #设置路由，注意是数组，可以设置多个，按照id做隔离  - id: user-service-router #路由id，没有要求，保持唯一即可  uri: http://localhost:8081 #设置真正的服务ip:port  predicates: #断言匹配  - Path=/info/\*\* #和服务中的路径匹配,是正则匹配的模式  - id: provider-service-router  uri: http://localhost:8082  predicates:  - Path=/info/\*\* #如果匹配到第一个路由，则第二个就不会走了，注意这不是负载均衡** #eureka的配置 eureka:  instance:  instance-id: ${spring.application.name}:${server.port}  prefer-ip-address: true  client:  service-url:  defaultZone: http://localhost:8761/eureka/ |

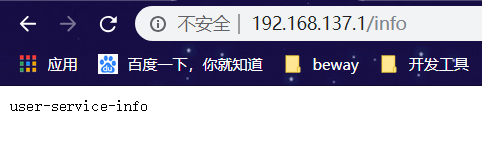
## 启动测试

**启动eureka-server**

**启动consumer-user-service**

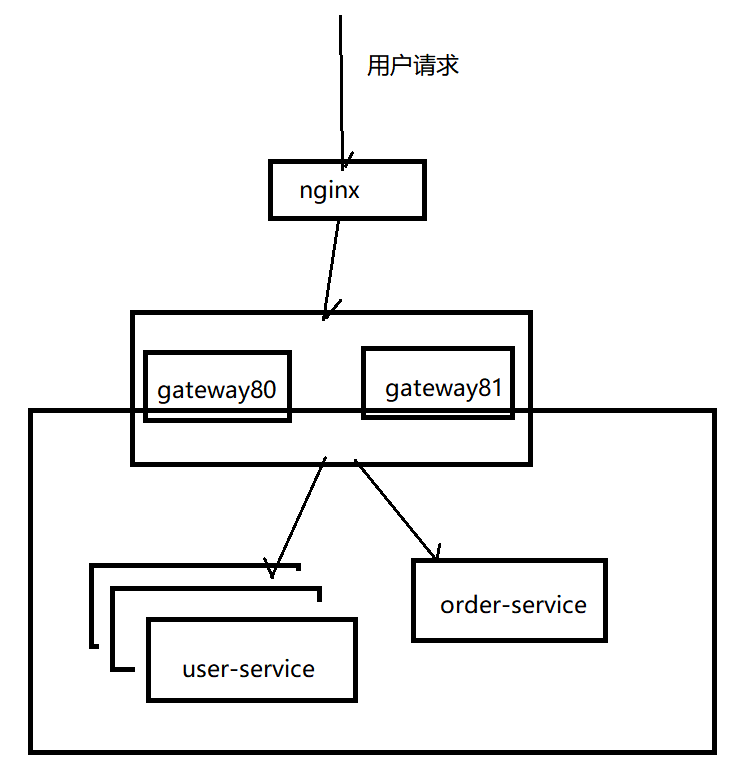
**启动gateway**

访问：<http://192.168.137.1/info>



# Gateway集群

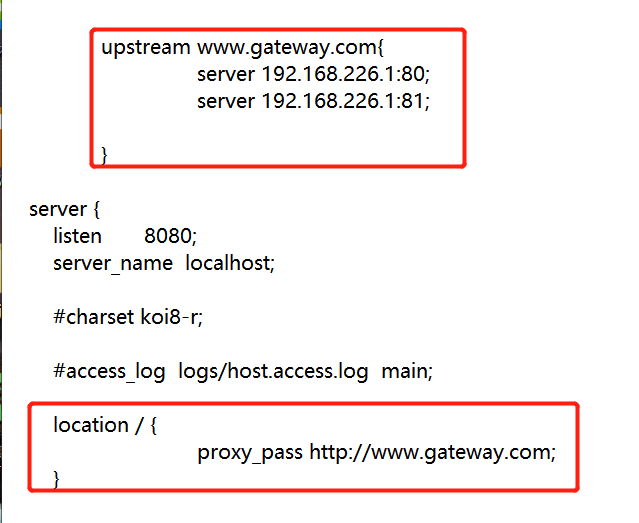
这里使用虚拟机实现



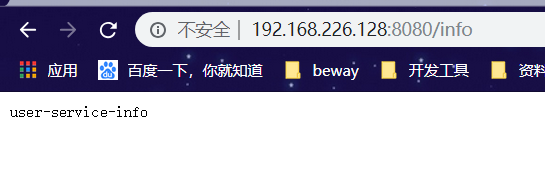
## 创建两个gateway，端口分别为80和81

|  |
| --- |
| server:  port: 80 spring:  application:  name: gateway-80  cloud:  gateway:  enabled: true #开启网关，默认是开启的  discovery:  locator:  enabled: true  lower-case-service-id: true  routes: #设置路由，注意是数组，可以设置多个，按照id做隔离  - id: user-service-router #路由id，没有要求，保持唯一即可   **uri: http://192.168.226.1:8081 #设置真正的服务ip:port**  predicates: #断言匹配  - Path=/info/\*\* #和服务中的路径匹配,是正则匹配的模式  - id: provider-service-router  uri: http://192.168.226.1:8082  predicates:  - Path=/info/\*\* #如果匹配到第一个路由，则第二个就不会走了，注意这不是负载均衡 #eureka的配置 eureka:  instance:  instance-id: ${spring.application.name}:${server.port}  prefer-ip-address: true  client:  service-url:  defaultZone: http://localhost:8761/eureka/ |

## Nginx的配置文件修改



## 访问测试



# Gateway的两种路由配置方式

## 代码注入方式（了解）

官网给出的配置类，我们照葫芦画瓢

<https://docs.spring.io/spring-cloud-gateway/docs/2.2.5.RELEASE/reference/html/#modifying-the-way-remote-addresses-are-resolved>



### 创建配置类GatewayConfig

|  |
| --- |
| @Configuration public class GatewayConfig {   @Bean  public RouteLocator routeLocator(RouteLocatorBuilder routeLocatorBuilder) {  RouteLocatorBuilder.Builder routes = routeLocatorBuilder.routes();  routes  .route("path\_rote\_guonei", r -> r.path("/guonei").uri("http://news.baidu.com/guonei"))  .route("path\_rote\_guoji", r -> r.path("/guoji").uri("http://news.baidu.com/guoji"))  .route("path\_rote\_tech", r -> r.path("/tech").uri("http://news.baidu.com/tech"))  .route("path\_rote\_lady", r -> r.path("/lady").uri("http://news.baidu.com/lady"))  .build();  return routes.build();  } |

### 启动测试



## 使用yml方式（重点）

和上面的快速入门一样，使用yml的方式，在开发中是常用的



# Gateway微服务名动态路由，负载均衡

## 概述

从之前的配置里面我们可以看到我们的URL都是写死的，这不符合我们微服务的要求，我们微服务是只要知道服务的名字，根据名字去找，而直接写死就没有负载均衡的效果了

默认情况下**Gateway会根据注册中心的服务列表，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**

**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://serviceName是spring cloud gateway在微服务中自动为我们创建的负载均衡uri**

## 最佳实践

### 修改gateway配置

|  |
| --- |
| server:  port: 80 spring:  application:  name: gateway-80  cloud:  gateway:  **discovery:  locator:  enabled: true #开启动态路由  lower-case-service-id: true #动态路由小驼峰规则**  routes: #设置路由，注意是数组，可以设置多个，按照id做隔离  - id: user-service-router #路由id，没有要求，保持唯一即可  uri: lb://provider #使用lb协议 微服务名称做负均衡  predicates: #断言匹配  - Path=/info/\*\* #和服务中的路径匹配,是正则匹配的模式  - id: provider-service-router  uri: http://localhost:8082  predicates:  - Path=/info/\*\* #如果匹配到第一个路由，则第二个就不会走了，注意这不是负载均衡 #eureka的配置 eureka:  instance:  instance-id: ${spring.application.name}:${server.port}  prefer-ip-address: true  client:  service-url:  defaultZone: http://localhost:8761/eureka/ |

### 启动测试

**启动eureka-server**

**启动两个服务名为provider的服务，和uri里面lb://服务名一致**

**在provider里面提供两个接口/info**

**访问测试：<http://192.168.137.1/info> 正常访问**



当我们新起一个服务，那么gateway可以实现服务发现功能，我们并没有再routers里面配置路由规则，然而我们访问 新起的provider-order-service，测试访问

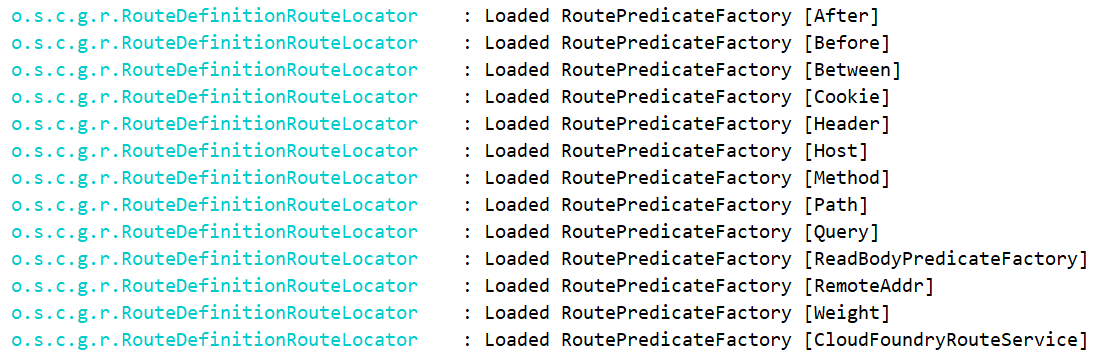
<http://localhost/provider-order-service/info> 可以成功，这就是动态路由和服务发现



# Predicate断言工厂的使用

在gateway启动时会去加载一些路由断言工厂

<https://docs.spring.io/spring-cloud-gateway/docs/2.2.5.RELEASE/reference/html/#gateway-request-predicates-factories>

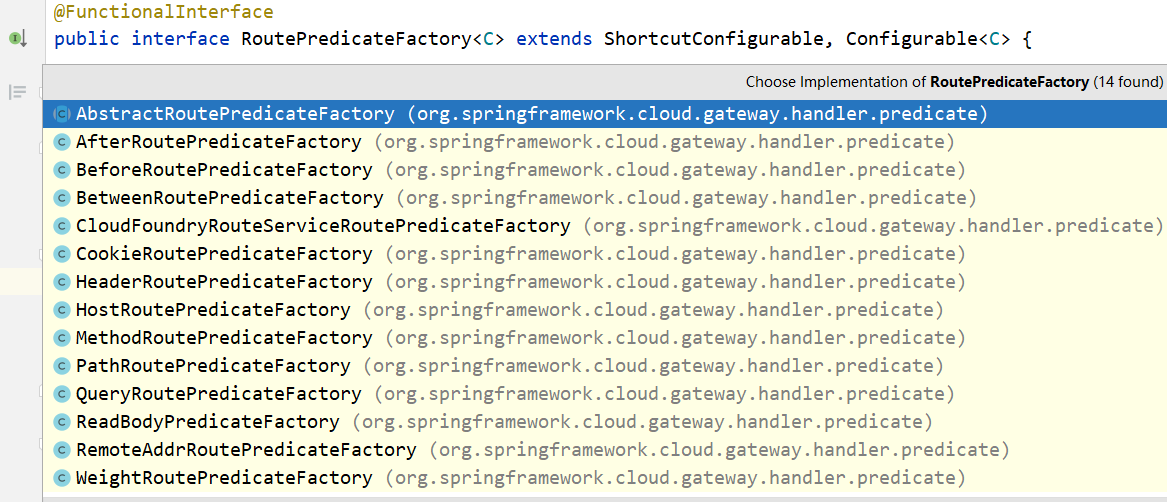


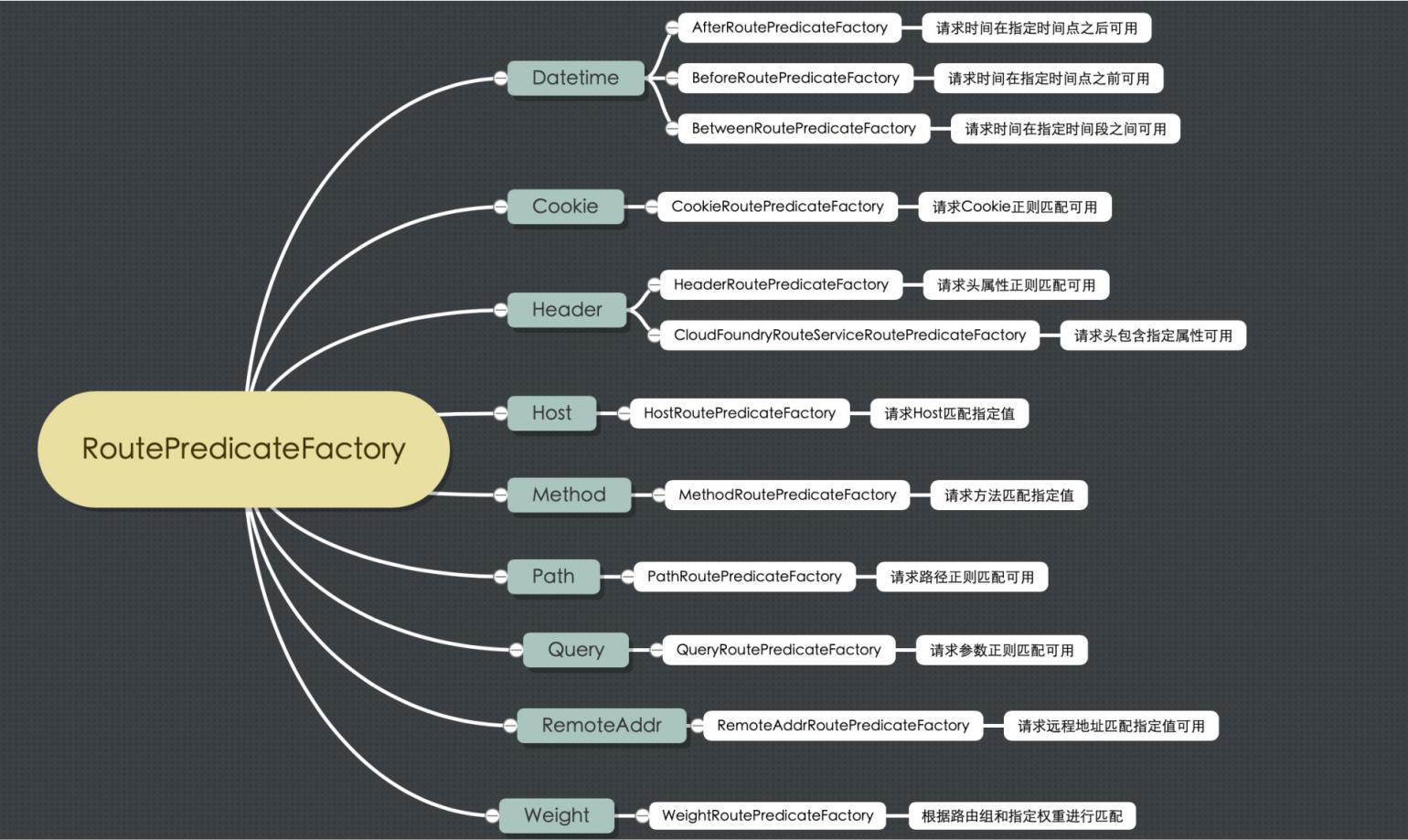
## 什么是断言，Gateway里面有哪些断言

通俗的说，**断言就是一些布尔表达式**，满足条件的返回true，不满足的返回false。

Spring Cloud Gateway将路由作为Spring WebFlux HandlerMapping基础架构的一部分进行匹配。Spring Cloud Gateway包括许多内置的路由断言工厂。所有这些断言都与HTTP请求的不同属性匹配。您可以将**多个路由断言可以组合**使用

Spring Cloud Gateway创建对象时，使用RoutePredicateFactory创建Predicate对象，Predicate对象可以赋值给Route。





## 如何使用这些断言

使用断言判断时，我们常用yml配置文件的方式进行配置

|  |
| --- |
| server:  port: 80 spring:  application:  name: gateway-80  cloud:  gateway:  enabled: true #开启网关，默认是开启的  routes: #设置路由，注意是数组，可以设置多个，按照id做隔离  - id: user-service #路由id，没有要求，保持唯一即可  uri: lb://provider #使用lb协议 微服务名称做负均衡  predicates: #断言匹配  - Path=/info/\*\* #和服务中的路径匹配,是正则匹配的模式  - After=2020-01-20T17:42:47.789-07:00[Asia/Shanghai] #此断言匹配发生在指定日期时间之后的请求，ZonedDateTime dateTime=ZonedDateTime.now()获得  - Before=2020-06-18T21:26:26.711+08:00[Asia/Shanghai] #此断言匹配发生在指定日期时间之前的请求  - Between=2020-06-18T21:26:26.711+08:00[Asia/Shanghai],2020-06-18T21:32:26.711+08:00[Asia/Shanghai] #此断言匹配发生在指定日期时间之间的请求  - Cookie=name,xiaobai #Cookie路由断言工厂接受两个参数，Cookie名称和regexp(一个Java正则表达式)。此断言匹配具有给定名称且其值与正则表达式匹配的cookie  - Header=token,123456 #头路由断言工厂接受两个参数，头名称和regexp(一个Java正则表达式)。此断言与具有给定名称的头匹配，该头的值与正则表达式匹配。  - Host=\*\*.bai\*.com:\* #主机路由断言工厂接受一个参数:主机名模式列表。该模式是一个ant样式的模式。作为分隔符。此断言匹配与模式匹配的主机头  - Method=GET,POST #方法路由断言工厂接受一个方法参数，该参数是一个或多个参数:要匹配的HTTP方法  - Qurey=username #查询路由断言工厂接受两个参数:一个必需的param和一个可选的regexp(一个Java正则表达式)。  - RemoteAddr=192.168.1.1/24 #RemoteAddr路由断言工厂接受一个源列表(最小大小1)，这些源是cidr符号(IPv4或IPv6)字符串，比如192.168.1.1/24(其中192.168.1.1是IP地址，24是子网掩码)。 |

还有一个访问权重的设置，意思是说：

80%的请求，由https://weighthigh.org这个url去处理

20%的请求由https://weightlow.org去处理

|  |
| --- |
| 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 |

## 断言总结

Predicate就是为了实现一组匹配规则，让请求过来找到对应的Route进行处理

## 自定义断言工厂（了解）

其实gateway默认为我提供的断言工厂已经够用了，但是我们想自己定义呢

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 方法中可以通过 exchange.getRequest() 拿到 ServerHttpRequest 对象，从而可以获取到请求的参数、请求方式、请求头等信息。

apply 方法的参数是自定义的配置类，在使用的时候配置参数，在 apply 方法中直接获取使用。

命名需要以 RoutePredicateFactory 结尾，比如 CheckAuthRoutePredicateFactory，那么在使用的时候 CheckAuth 就是这个路由断言工厂的名称。

### 最佳实践

#### 创建配置类

|  |
| --- |
| @Component public class CheckAuthRoutePredicateFactory  extends AbstractRoutePredicateFactory<CheckAuthRoutePredicateFactory.Config> {   public CheckAuthRoutePredicateFactory() {  super(Config.class);  }   @Override  public Predicate<ServerWebExchange> apply(Config config) {  return exchange -> {  System.*err*.println("进入了CheckAuthRoutePredicateFactory\t" + config.getName());  if (config.getName().equals("xiaobai")) {  return true;  }  return false;  };  }   @Data  static class Config {  private String name;  } } |

#### 修改配置文件

|  |
| --- |
| spring:  application:  name: gateway-80  cloud:  gateway:  enabled: true #开启网关，默认是开启的  routes: #设置路由，注意是数组，可以设置多个，按照id做隔离  - id: user-service #路由id，没有要求，保持唯一即可  uri: lb://provider #使用lb协议 微服务名称做负均衡  **predicates: #断言匹配  - name: CheckAuth  args:  name: xiaobai** |

#### 访问测试是否进入自定义断言器

随便访问，测试一下而已

# Filter过滤器工厂（重点）

## 概述

gateway里面的过滤器和Servlet里面的过滤器，功能差不多，路由过滤器可以用于修改进入Http请求和返回Http响应，过滤器只能指定路由进行使用

## 分类

### 按生命周期分两种

**pre 在业务逻辑之前**

**post 在业务逻辑之后**

### 按种类分也是两种

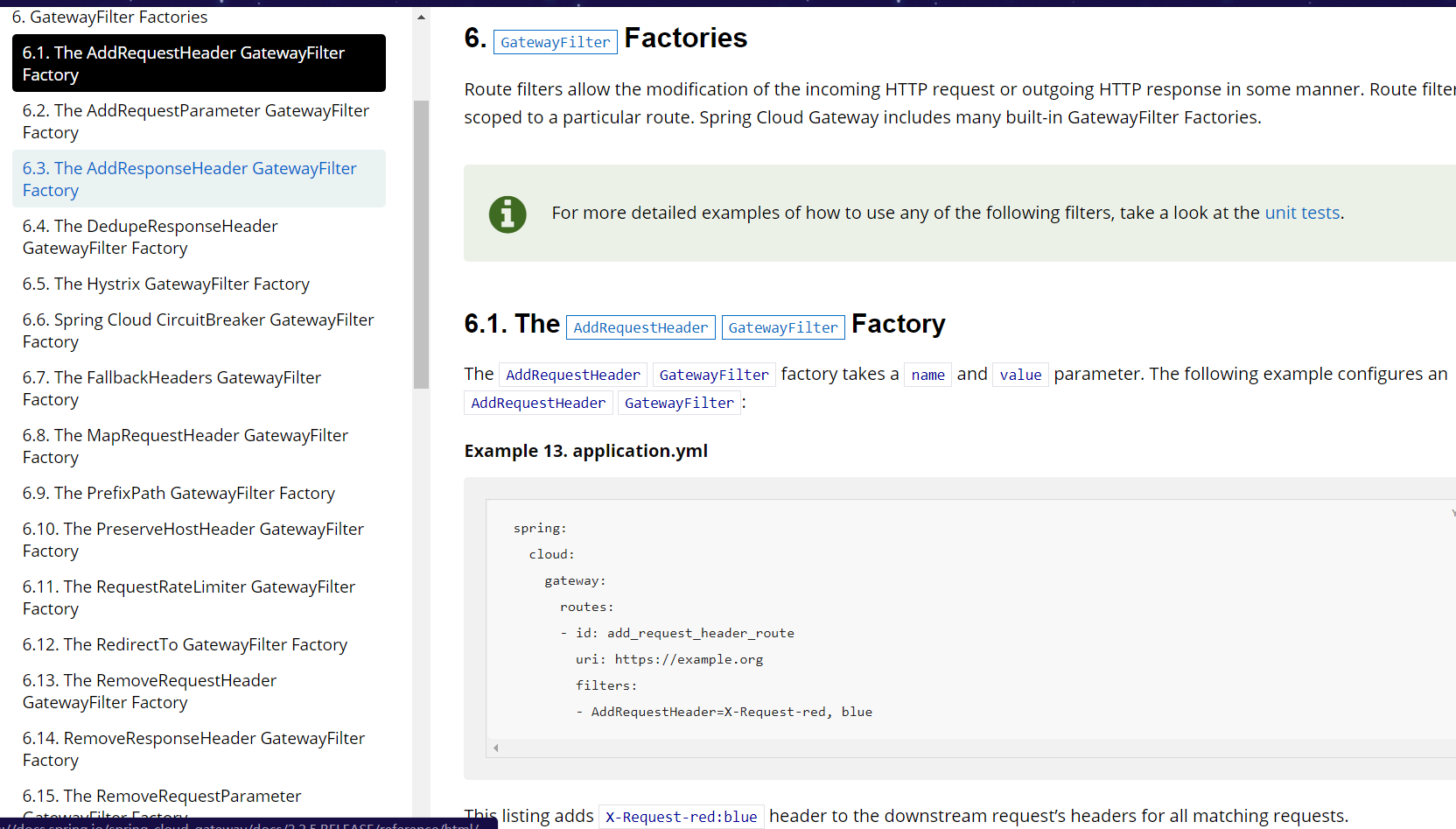
**GatewayFilter 需要配置某个路由，才能过滤。如果需要使用全局路由，需要配置Default Filters。**

**GlobalFilter 全局过滤器，不需要配置路由，系统初始化作用到所有路由上**

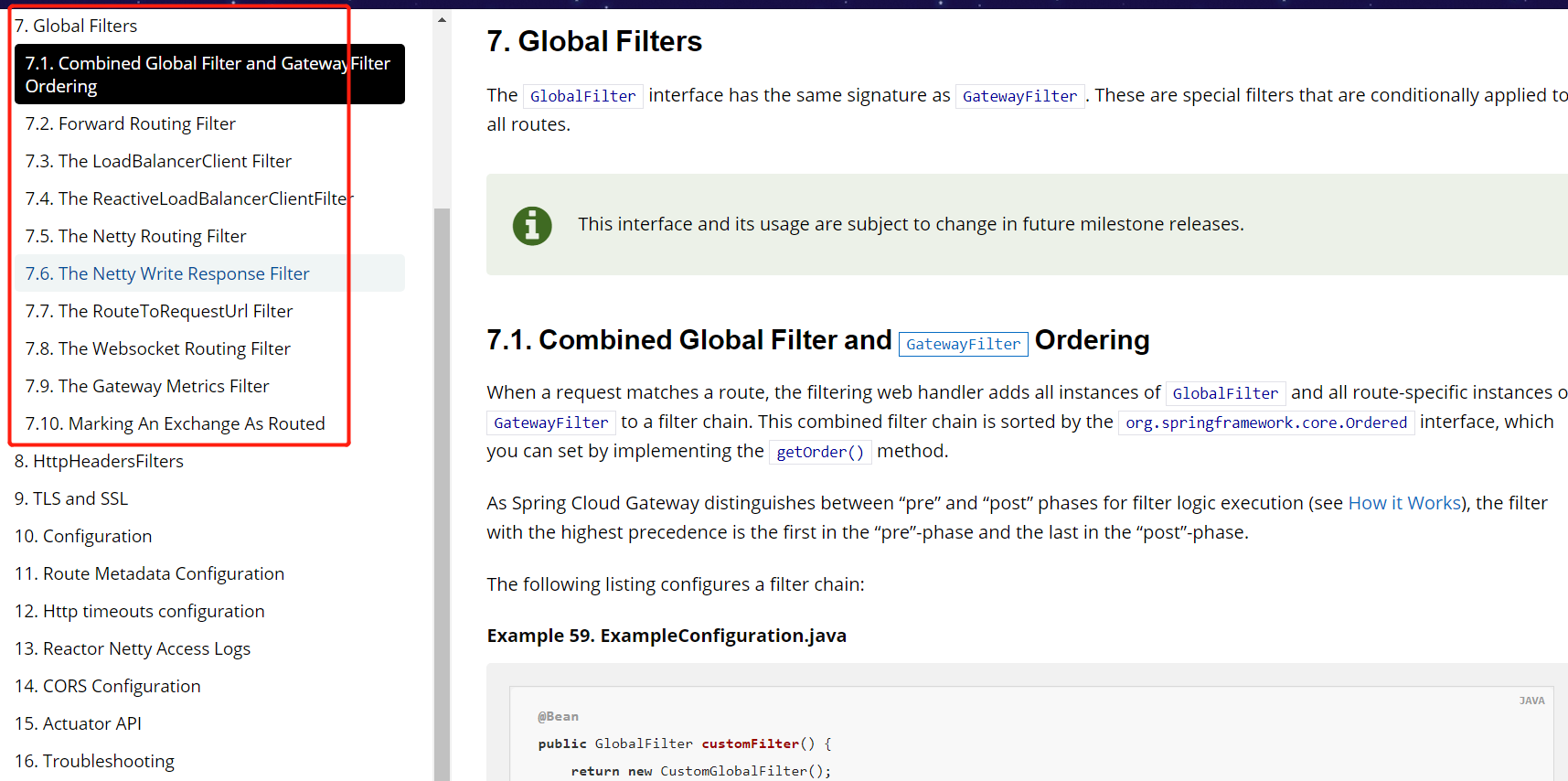
## 官方文档查看过滤器

### 单一过滤器（31个）

<https://docs.spring.io/spring-cloud-gateway/docs/2.2.5.RELEASE/reference/html/#gatewayfilter-factories>



### 全局过滤器（9个）



## 自定义网关过滤器(重点)

### 自定义全局过滤器

全局过滤器的优点的初始化时默认挂到所有路由上，我们可以使用它来完成IP过滤，限流等功能

### 创建配置类GlobalFilterConfig

|  |
| --- |
| @Component @Slf4j public class GlobalFilterConfig **implements GlobalFilter, Ordered** {   @Override  public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {   *log*.info("进入了我自己的全局过滤器");  String token = exchange.getRequest().getQueryParams().getFirst("token");  if (token == null) {  *log*.error("token为空，说明没有认证");  exchange.getResponse().setStatusCode(HttpStatus.*NOT\_ACCEPTABLE*);  return exchange.getResponse().setComplete();  }  *log*.info("验证通过");  return chain.filter(exchange);  }   /\*\*  \* order越小 越先执行  \*  \* @return  \*/  @Override  public int getOrder() {  return 0;  } } |

### 访问测试

<http://localhost/info?token=asdad>

# IP认证拦截实战

## 创建IPGlobalFilter

|  |
| --- |
| @Component @Slf4j public class IPCheckFilter implements GlobalFilter, Ordered {  @SneakyThrows  @Override  public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  String ip = exchange.getRequest().getHeaders().getHost().getHostName();  //这里写死了，只做演示  if (ip.equals("localhost")) {  //说明是黑名单里面的ip  ServerHttpResponse response = exchange.getResponse();  response.setStatusCode(HttpStatus.*UNAUTHORIZED*);  Map<String, Object> map = new HashMap<>();  map.put("code", HttpStatus.*UNAUTHORIZED*);  map.put("msg", "非法访问");  response.getHeaders().add("content-Type", "application/json;charset=UTF-8");  ObjectMapper objectMapper = new ObjectMapper();  byte[] bytes = objectMapper.writeValueAsBytes(map);  DataBuffer buffer = response.bufferFactory().wrap(bytes);  return response.writeWith(Mono.*just*(buffer));  }  return chain.filter(exchange);  }   /\*\*  \* 设置此过滤器的执行顺序  \*  \* @return  \*/  @Override  public int getOrder() {  return 1;  } } |

## 测试访问



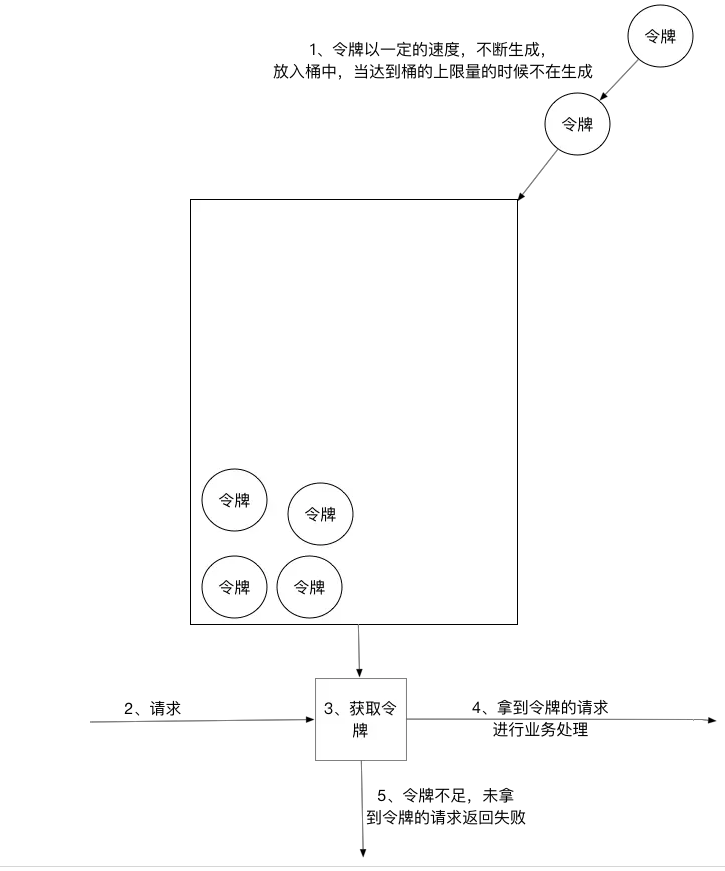
# 限流实战（会问）

## 什么是限流

通俗的说，限流就是**限制一段时间内，用户访问资源的次数**，减轻服务器压力，限流大致分为两种：

1. **IP限流（5s内同一个ip访问超过3次，则限制不让访问，过一段时间才可继续访问）**
2. **请求量限流（只要在一段时间内(窗口期)，请求次数达到阀值，就直接拒绝后面来的访问了，过一段时间才可以继续访问）（粒度可以细化到一个api（url），一个服务）**

## 本次限流模型



**入不敷出**

1）、所有的请求在处理之前都需要拿到一个可用的令牌才会被处理；

2）、根据限流大小，设置按照一定的**速率**往桶里添加令牌；

3）、**桶设置最大的放置令牌限制**，当桶满时、新添加的令牌就被丢弃或者拒绝；

4）、请求达到后首先要获取令牌桶中的令牌，拿着令牌才可以进行其他的业务逻辑，处理完业务逻辑之后，将令牌直接删除；

5）、令牌桶有最低限额，当桶中的令牌达到最低限额的时候，请求处理完之后将不会删除令牌，以此保证足够的限流；

## Gateway结合redis实现请求量限流

Spring Cloud Gateway 已经内置了一个RequestRateLimiterGatewayFilterFactory，我们可以直接使用。

目前RequestRateLimiterGatewayFilterFactory的实现依赖于Redis，所以我们还要引入spring-boot-starter-data-redis-reactive。

### 添加依赖

|  |
| --- |
| <!--限流要引入Redis--> <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis-reactive</artifactId> </dependency> |

### 修改配置文件

|  |
| --- |
| server:  port: 80 spring:  application:  name: gateway-80  cloud:  gateway:  enabled: true  routes:  - id: user-service  uri: lb://consumer-user-service  predicates:  - Path=/info/\*\*  filters:  - name: RequestRateLimiter  args:  key-resolver: '#{@hostAddrKeyResolver}'  redis-rate-limiter.replenishRate: 1  redis-rate-limiter.burstCapacity: 3   redis: #redis的配置  host: 192.168.226.128  port: 6379  database: 0 eureka:  instance:  instance-id: ${spring.application.name}:${server.port}  prefer-ip-address: true  client:  service-url:  defaultZone: http://localhost:8761/eureka/ |

### 配置文件说明

**在上面的配置文件，配置了 redis的信息，并配置了RequestRateLimiter的限流过滤器，该过滤器需要配置三个参数：**

**burstCapacity：令牌桶总容量。**

**replenishRate：令牌桶每秒填充平均速率。**

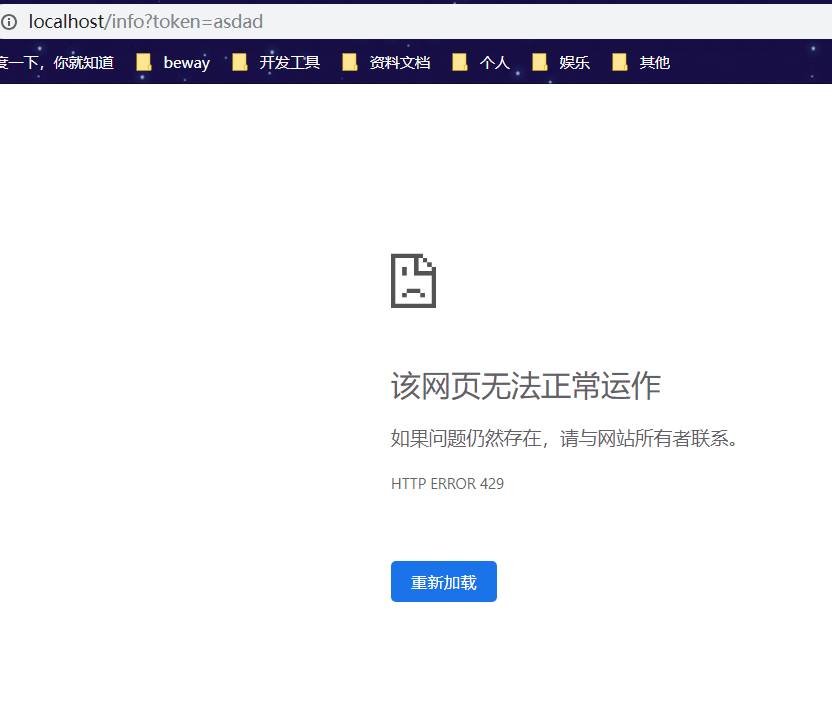
**key-resolver：用于限流的键的解析器的 Bean 对象的名字。它使用 SpEL 表达式根据#{@beanName}从 Spring 容器中获取 Bean 对象。**

### 创建配置类RequestRateLimiterConfig

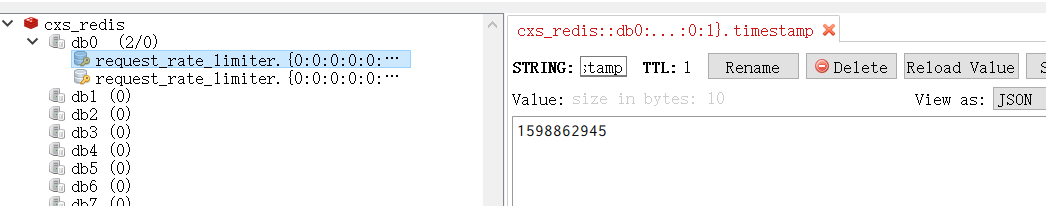
|  |
| --- |
| @Configuration public class RequestRateLimiterConfig {   /\*\*  \* IP限流  \* 把用户的IP作为限流的Key  \*  \* @return  \*/  @Bean  @Primary  public KeyResolver hostAddrKeyResolver() {  return (exchange) -> Mono.*just*(exchange.getRequest().getRemoteAddress().getHostName());  }    /\*\*  \* 用户id限流  \* 把用户ID作为限流的key  \*  \* @return  \*/  @Bean  public KeyResolver userKeyResolver() {  return exchange -> Mono.*just*(exchange.getRequest().getQueryParams().getFirst("userId"));  }    /\*\*  \* 请求接口限流  \* 把请求的路径作为限流key  \*  \* @return  \*/  @Bean  public KeyResolver apiKeyResolver() {  return exchange -> Mono.*just*(exchange.getRequest().getPath().value());  }  } |

### 启动快速访问测试

<http://localhost/info?token=asdad> 快速访问后报429



查看redis



# 跨域配置

|  |
| --- |
| @Configuration public class CorsConfig {  @Bean  public CorsWebFilter corsFilter() {  CorsConfiguration config = new CorsConfiguration();  config.addAllowedMethod("\*");  config.addAllowedOrigin("\*");  config.addAllowedHeader("\*");   UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource(new PathPatternParser());  source.registerCorsConfiguration("/\*\*", config);   return new CorsWebFilter(source);  } } |

# 总结和面试

**微服务的网关，可以很好地将具体的服务和浏览器隔离开，只暴露网关的地址给到浏览器**

**在微服务网关中，可以很好的实现校验认证，负载均衡（lb），黑名单拦截，限流等。**

## Gateway和zuul的区别 ZuulFilter

**Zuul也是web网关，本质上就是一组过滤器，按照定义的顺序，来执行过滤操作**

**二者的区别：**

1. **两者均是web网关，处理的是http请求**
2. **Gateway是springcloud官方的组件，zuul则是netflix的产品**

**springcloud，netflix ，alibaba（nacos，sentinel，dubbo zk）**

1. **gateway在spring的支持下，内部实现了限流、负载均衡等，扩展性也更强，但同时也限制了仅适合于Spring Cloud套件。而zuul则可以扩展至其他微服务框架中，其内部没有实现限流、负载均衡等。**
2. **Gateway（Netty NIO）很好的支持异步(spring5.x ,webFlux 响应式编程默认是异步的)，而zuul仅支持同步**

## Nginx在微服务中的地位

最后简单聊一下nginx，在过去几年微服务架构还没有流行的日子里，nginx已经得到了广大开发者的认可，其性能高、扩展性强、可以灵活利用lua脚本构建插件的特点让人没有抵抗力。（nginx的请求转发 最大并发是多个次，5w-10w左右）

（lua脚本，原子性）（redis的分布式锁 jedis.setnx() setifAbsent() redisson ）

有一个能满足我所有需求还很方便我扩展的东西，还免费，凭啥不用？？

但是，如今很多微服务架构的项目中不会选择nginx，我认为原因有以下几点：

微服务框架一般来说是配套的，集成起来更容易

如今微服务架构中，仅有很少的公司会面对无法解决的性能瓶颈，而他们也不会因此使用nginx，而是选择开发一套适合自己的微服务框架

spring boot对于一些模板引擎如FreeMarker、themleaf的支持是非常好的，很多应用还没有达到动、静态文件分离的地步，对nginx的需求程度并不大。

**不是为了用技术而用技术 按照实际业务来 目的是盈利**

无论如何，nginx作为一个好用的组件，最终使不使用它都是由业务来驱动的，只要它能为我们方便的解决问题，那用它又有何不可呢？

## 关于限流，面试不会直接问，而是间接来问 问 不卖超

比如：如果在抢购过程中，用户量请求非常大，怎么确保商品不会卖超

Redis 单线程 （IO为什么快，因为我们现在的处理器是多核心数的，redis底层使用的是IO的**多路复用**）

一般人只会在意商品卖超，而忘记了限流的重要性

Mq（限流 削峰，异步，解耦合）

手写限流：

Ip来限流 ： 1.拿到请求的ip地址 在一定时间内 请求次数到达一个阈值就限制访问

多线程 redis

起一个线程 定期删除

拿到ip value 次数-时间（未来的时间）

时间窗口滑动

## 健康状态检查等

健康检查的依赖

|  |
| --- |
| *<!-- 健康检查的依赖-->* <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-actuator</artifactId> </dependency> |

添加配置文件

|  |
| --- |
| management:  endpoints:  web:  exposure:  include: '\*' *#暴露检查的端点* |

