**一、为什么需要服务网关：**

**1、什么是服务网关：**

传统的单体架构中只需要开放一个服务给客户端调用，但是微服务架构中是将一个系统拆分成多个微服务，如果没有网关，客户端只能在本地记录每个微服务的调用地址，当需要调用的微服务数量很多时，它需要了解每个服务的接口，这个工作量很大。那有了网关之后，能够起到怎样的改善呢？

网关作为系统的唯一流量入口，封装内部系统的架构，所有请求都先经过网关，由网关将请求路由到合适的微服务，所以，使用网关的好处在于：

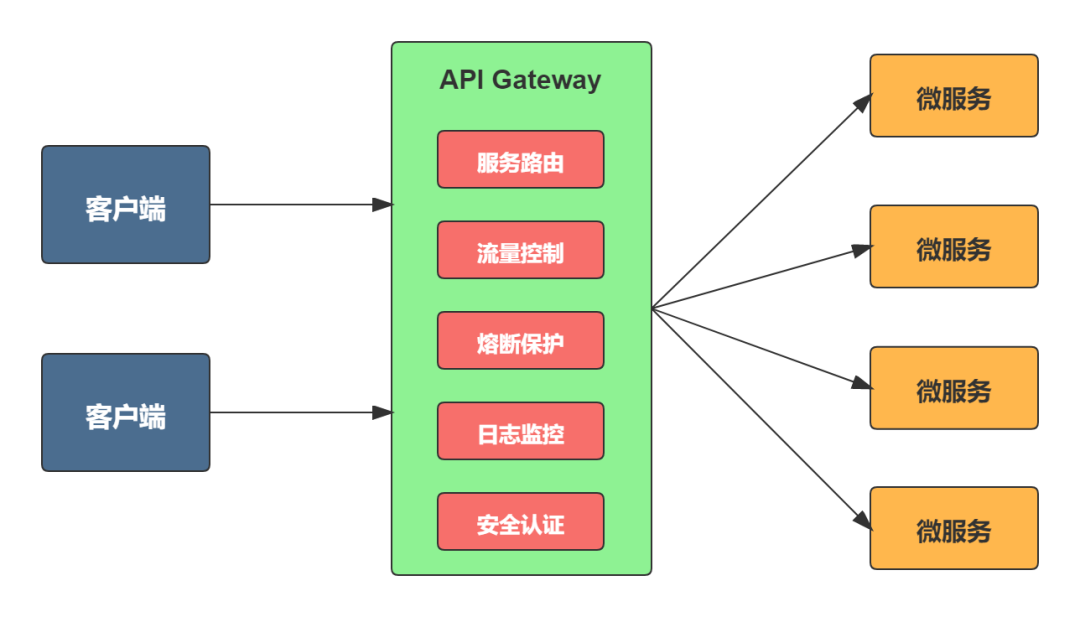
* **简化客户端的工作。** 网关将微服务封装起来后，客户端只需同网关交互，而不必调用各个不同服务；
* **降低函数间的耦合度。** 一旦服务接口修改，只需修改网关的路由策略，不必修改每个调用该函数的客户端，从而减少了程序间的耦合性
* **解放开发人员把精力专注于业务逻辑的实现。** 由网关统一实现服务路由(灰度与ABTest)、负载均衡、访问控制、流控熔断降级等非业务相关功能，而不需要每个服务 API 实现时都去考虑

但是 API 网关也存在不足之处，在微服务这种去中心化的架构中，网关又成了一个中心点或瓶颈点，它增加了一个我们必须开发、部署和维护的高可用组件。正是由于这个原因，在网关设计时必须考虑即使 API 网关宕机也不要影响到服务的调用和运行，所以需要对网关的响应结果有数据缓存能力，通过返回缓存数据或默认数据屏蔽后端服务的失败。

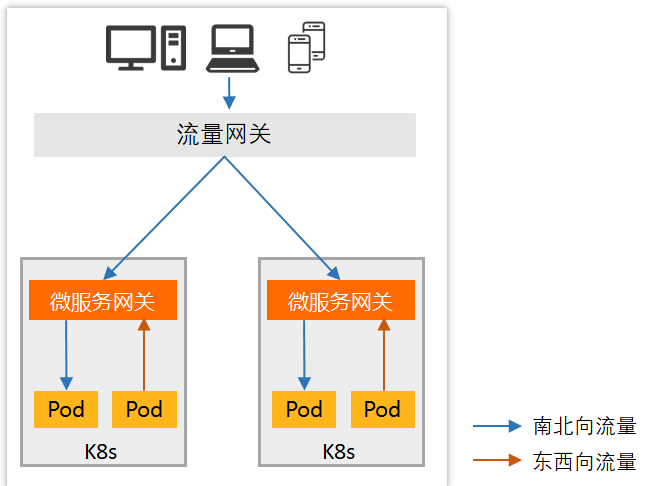
在服务的调用方式上面，网关也有一定的要求，API 网关最好是支持 I/O 异步、同步非阻塞的，如果服务是同步阻塞调用，可以理解为微服务模块之间是没有彻底解耦的，即如果A依赖B提供的API，如果B提供的服务不可用将直接影响到A不可用，除非同步服务调用在API网关层或客户端做了相应的缓存。

因此为了彻底解耦，在微服务调用上更建议选择异步方式进行。而对于 API 网关需要通过底层多个细粒度的 API 组合的场景，推荐采用响应式编程模型进行而不是传统的异步回调方法组合代码，其原因除了采用回调方式导致的代码混乱外，还有就是对于 API 组合本身可能存在并行或先后调用，对于采用回调方式往往很难控制。

**2、服务网关的基本功能：**



### 3、流量网关与服务网关的区别：

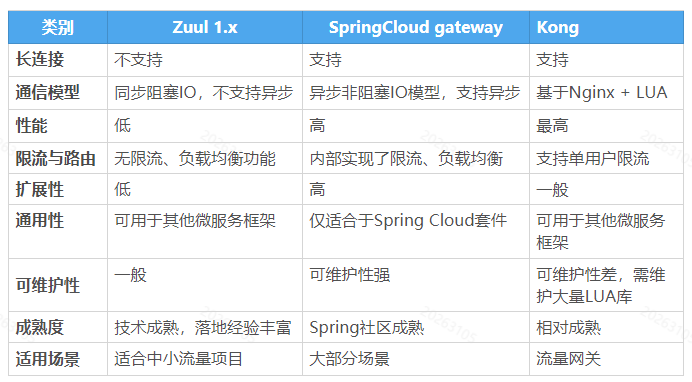


流量网关和服务网关在系统整体架构中所处的位置如上图所示，流量网关（如Nignx）是指提供全局性的、与后端业务应用无关的策略，例如 HTTPS证书卸载、Web防火墙、全局流量监控等。

而微服务网关（如Spring Cloud Gateway）是指与业务紧耦合的、提供单个业务域级别的策略，如服务治理、身份认证等。也就是说，流量网关负责南北向流量调度及安全防护，微服务网关负责东西向流量调度及服务治理。

## 二、服务网关的部署：

### 1、主流网关的对比与选型：



* **Kong 网关**：Kong 的性能非常好，非常适合做流量网关，但是对于复杂系统不建议业务网关用 Kong，主要是工程性方面的考虑
* **Zuul1.x 网关**：Zuul 1.0 的落地经验丰富，但是性能差、基于同步阻塞IO，适合中小架构，不适合并发流量高的场景，因为容易产生线程耗尽，导致请求被拒绝的情况
* **gateway 网关**：功能强大丰富，性能好，官方基准测试 RPS (每秒请求数)是Zuul的1.6倍，能与 SpringCloud 生态很好兼容，单从流式编程+支持异步上也足以让开发者选择它了。
* **Zuul 2.x**：性能与 gateway 差不多，基于非阻塞的，支持长连接，但 SpringCloud 没有集成 zuul2 的计划，并且 Netflix 相关组件都宣布进入维护期，前景未知。

综上，gateway 网关更加适合 SpringCloud 项目，而从发展趋势上看，gateway 替代 zuul 也是必然的。

### 2、Spring Cloud Gateway 网关的搭建：

（1）声明依赖版本号：

 <properties>  
  <spring-boot.version>2.3.2.RELEASE</spring-boot.version>  
  <spring-cloud.version>Hoxton.SR9</spring-cloud.version>  
  <spring-cloud-alibaba.version>2.2.6.RELEASE</spring-cloud-alibaba.version>  
 </properties>  
   
 *<!-- 只声明依赖，不引入依赖 -->*  
 <dependencyManagement>  
  <dependencies>  
   *<!-- 声明springBoot版本 -->*  
   <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-dependencies</artifactId>  
    <version>${spring-boot.version}</version>  
    <type>pom</type>  
    <scope>import</scope>  
   </dependency>  
   *<!-- 声明springCloud版本 -->*  
   <dependency>  
    <groupId>org.springframework.cloud</groupId>  
    <artifactId>spring-cloud-dependencies</artifactId>  
    <version>${spring-cloud.version}</version>  
    <type>pom</type>  
    <scope>import</scope>  
   </dependency>  
   *<!-- 声明 springCloud Alibaba 版本 -->*  
   <dependency>  
    <groupId>com.alibaba.cloud</groupId>  
    <artifactId>spring-cloud-alibaba-dependencies</artifactId>  
    <version>${spring-cloud-alibaba.version}</version>  
    <type>pom</type>  
    <scope>import</scope>  
   </dependency>  
  </dependencies>  
 </dependencyManagement>

（2）添加依赖：

*<!-- 引入gateway网关 -->*  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-gateway</artifactId>  
 <exclusions>  
        <exclusion>  
   <groupId>org.springframework.boot</groupId>  
         <artifactId>spring-boot-starter-web</artifactId>  
        </exclusion>  
    </exclusions>  
</dependency>

注意：一定要排除掉 spring-boot-starter-web 依赖，否则启动报错

（3）配置项目名与端口：

server:  
  port: 9023  
  servlet:  
    context-path: /${spring.application.name}  
spring:  
  application:  
    name: gateway

好了，网关项目搭建完成，其实就添加这么一个依赖，关于详细的配置以及作用下文介绍。

### 3、Spring Cloud Gateway 配置项的说明：

在介绍 Spring Cloud Gateway 的配置项之前，我们先了解几个 Spring Cloud Gateway 的核心术语：

* **断言（Predicate）**：参照 Java8 的新特性Predicate，允许开发人员匹配 HTTP 请求中的任何内容，比如请求头或请求参数，最后根据匹配结果返回一个布尔值。
* **路由（route）**：由ID、目标URI、断言集合和过滤器集合组成。如果聚合断言结果为真，则转发到该路由。
* **过滤器（filter）**：可以在返回请求之前或之后修改请求和响应的内容。

#### 3.1、路由 Route：

Route 主要由 路由id、目标uri、断言集合和过滤器集合组成，那我们简单看看这些属性到底有什么作用。

* **id**：路由标识，要求唯一，名称任意（默认值 uuid，一般不用，需要自定义）
* **uri**：请求最终被转发到的目标地址
* **order**：路由优先级，数字越小，优先级越高
* **predicates**：断言数组，即判断条件，如果返回值是boolean，则转发请求到 uri 属性指定的服务中
* **filters**：过滤器数组，在请求传递过程中，对请求做一些修改

#### 3.2、断言 Predicate：

Predicate 来自于 Java8 的接口。Predicate 接受一个输入参数，返回一个布尔值结果。该接口包含多种默认方法来将 Predicate 组合成其他复杂的逻辑（比如：与，或，非）。

Predicate 可以用于接口请求参数校验、判断新老数据是否有变化需要进行更新操作。Spring Cloud Gateway 内置了许多 Predict，这些 Predict 的源码在 org.springframework.cloud.gateway.handler.predicate 包中，有兴趣可以阅读一下。内置的一些断言如下图：



以上11种断言这里就不再介绍如何配置了，官方文档写的很清楚：

https://docs.spring.io/spring-cloud-gateway/docs/2.2.9.RELEASE/reference/html/

下面就以最后一种权重断言为例介绍一下如何配置。配置如下：

spring:  
  cloud:  
    gateway:  
      *# 路由数组：指当请求满足什么样的断言时，转发到哪个服务上*  
      routes:  
        *# 路由标识，要求唯一，名称任意*  
        - id: gateway-provider\_1  
    *# 请求最终被转发到的目标地址*  
          uri: http://localhost:9024  
          *# 设置断言*  
          predicates:  
            *# Path Route Predicate Factory 断言，满足 /gateway/provider/\*\* 路径的请求都会被路由到 http://localhost:9024 这个uri中*  
            - Path=/gateway/provider/\*\*  
            *# Weight Route Predicate Factory 断言，同一分组按照权重进行分配流量，这里分配了80%*  
            *# 第一个group1是分组名，第二个参数是权重*  
            - Weight=group1, 8  
          *# 配置过滤器（局部）*  
          filters:  
            *# StripPrefix：去除原始请求路径中的前1级路径，即/gateway*  
            - StripPrefix=1              
              
        - id: gateway-provider\_2  
          uri: http://localhost:9025  
          *# 设置断言*  
          predicates:  
            - Path=/gateway/provider/\*\*  
            *# Weight Route Predicate Factory，同一分组按照权重进行分配流量，这里分配了20%*  
            - Weight=group1, 2  
    *# 配置过滤器（局部）*  
          filters:  
            *# StripPrefix：去除原始请求路径中的前1级路径，即/gateway*  
            - StripPrefix=1

Spring Cloud Gateway 中的断言命名都是有规范的，格式：“xxx + RoutePredicateFactory”，比如权重断言 WeightRoutePredicateFactory，那么配置时直接取前面的 “Weight”。

如果路由转发匹配到了两个或以上，则是的按照配置先后顺序转发，上面都配置了路径：“ Path=/gateway/provider/\*\* ”，如果没有配置权重，则肯定是先转发到 “http://localhost:9024”，但是既然配置配置了权重并且相同的分组，则按照权重比例进行分配流量。

#### 3.3、过滤器 filter：

**Gateway 过滤器的生命周期：**

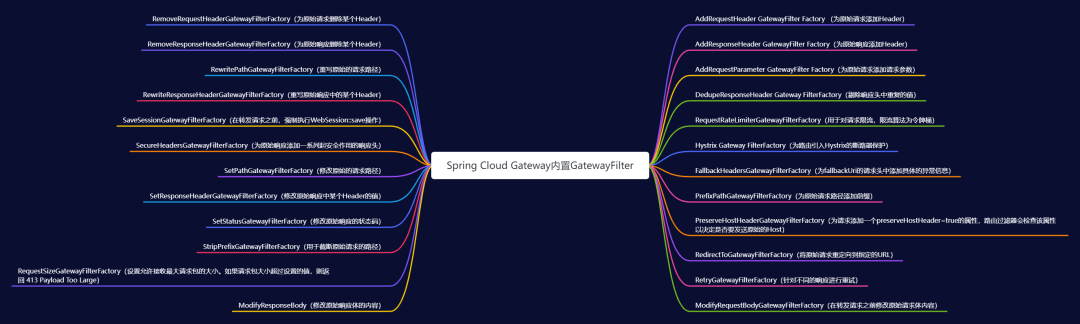
* **PRE**：这种过滤器在请求被路由之前调用。我们可利用这种过滤器实现身份验证、在集群中选择请求的微服务、记录调试信息等。
* **POST**：这种过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准的 HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端等。

**Gateway 过滤器从作用范围可分为两种:**

* **GatewayFilter**：应用到单个路由或者一个分组的路由上（需要在配置文件中配置）
* **GlobalFilter**：应用到所有的路由上（无需配置，全局生效）

（1）局部过滤器 GatewayFilter：

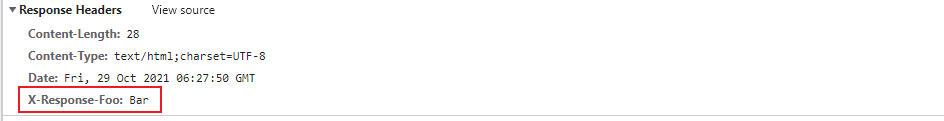
Spring Cloud Gateway 中内置了许多的局部过滤器，如下图：



局部过滤器需要在指定路由配置才能生效，默认是不生效的。以 “AddResponseHeaderGatewayFilterFactory” 这个过滤器为例，为原始响应添加Header，配置如下：

spring:  
  cloud:  
    gateway:  
      routes:  
        - id: gateway-provider\_1  
          uri: http://localhost:9024  
          predicates:  
            - Path=/gateway/provider/\*\*  
          *# 配置过滤器（局部）*  
          filters:  
            - AddResponseHeader=X-Response-Foo, Bar  
            *# StripPrefix：去除原始请求路径中的前1级路径，即/gateway*  
            - StripPrefix=1

浏览器请求，发现响应头中已经有了 X-Response-Foo=Bar 这个键值对，如下图：



在前面的示例中，我们也使用到了另一个局部过滤器 StripPrefixGatewayFilterFactory，该过滤器主要用于截断原始请求的路径，当我们请求 localhost:9023/gateway/provider/test 时，实际请求会被转发到 http://localhost:9024 服务上，并被截断成 “http://localhost:9024/provider/test"

注意：过滤器的名称只需要写前缀，过滤器命名必须是 "xxx + GatewayFilterFactory“（包括自定义）。

更多过滤器的配置参考官方文档：

https://docs.spring.io/spring-cloud-gateway/docs/2.2.9.RELEASE/reference/html/#gatewayfilter-factories

（2）自定义局部过滤器：

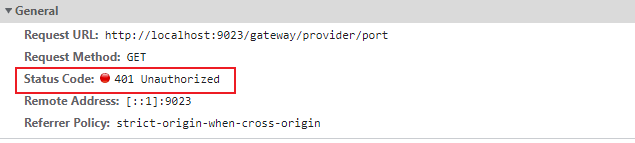
虽说内置的过滤器能够解决很多场景，但是难免还是有些特殊需求需要定制一个过滤器，下面就来介绍一下如何自定义局部过滤器。

*/\*\*  
 \* 名称必须是xxxGatewayFilterFactory形式  
 \* todo：模拟授权的验证，具体逻辑根据业务完善  
 \*/*  
@Component  
@Slf4j  
public class AuthorizeGatewayFilterFactory extends AbstractGatewayFilterFactory<AuthorizeGatewayFilterFactory.Config> {  
   
    private static final String AUTHORIZE\_TOKEN = "token";  
   
    *//构造函数，加载Config*  
    public AuthorizeGatewayFilterFactory() {  
        *//固定写法*  
        super(AuthorizeGatewayFilterFactory.Config.class);  
        log.info("Loaded GatewayFilterFactory [Authorize]");  
    }  
   
    *//读取配置文件中的参数 赋值到 配置类中*  
    @Override  
    public List<String> shortcutFieldOrder() {  
        *//Config.enabled*  
        return Arrays.asList("enabled");  
    }  
   
    @Override  
    public GatewayFilter apply(AuthorizeGatewayFilterFactory.Config config) {  
        return (exchange, chain) -> {  
            *//判断是否开启授权验证*  
            if (!config.isEnabled()) {  
                return chain.filter(exchange);  
            }  
   
            ServerHttpRequest request = exchange.getRequest();  
            HttpHeaders headers = request.getHeaders();  
            *//从请求头中获取token*  
            String token = headers.getFirst(AUTHORIZE\_TOKEN);  
            if (token == null) {  
                *//从请求头参数中获取token*  
                token = request.getQueryParams().getFirst(AUTHORIZE\_TOKEN);  
            }  
   
            ServerHttpResponse response = exchange.getResponse();  
            *//如果token为空，直接返回401，未授权*  
            if (StringUtils.isEmpty(token)) {  
                response.setStatusCode(HttpStatus.UNAUTHORIZED);  
                *//处理完成，直接拦截，不再进行下去*  
                return response.setComplete();  
            }  
            */\*\*  
             \* todo chain.filter(exchange) 之前的都是过滤器的前置处理  
             \*  
             \* chain.filter().then(  
             \*  过滤器的后置处理...........  
             \* )  
             \*/*  
            *//授权正常，继续下一个过滤器链的调用*  
            return chain.filter(exchange);  
        };  
    }  
   
    @Data  
    @AllArgsConstructor  
    @NoArgsConstructor  
    public static class Config {  
        *// 控制是否开启认证*  
        private boolean enabled;  
    }  
}

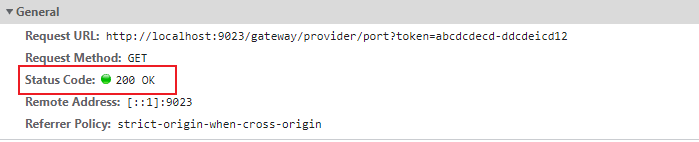
局部过滤器需要在路由中配置才能生效，配置如下：

spring:  
  cloud:  
    gateway:  
      routes:  
        - id: gateway-provider\_1  
          uri: http://localhost:9024  
          predicates:  
            - Path=/gateway/provider/\*\*  
          *## 配置过滤器（局部）*  
          filters:  
            - AddResponseHeader=X-Response-Foo, Bar  
            *## AuthorizeGatewayFilterFactory自定义过滤器配置，值为true需要验证授权，false不需要*  
            - Authorize=true

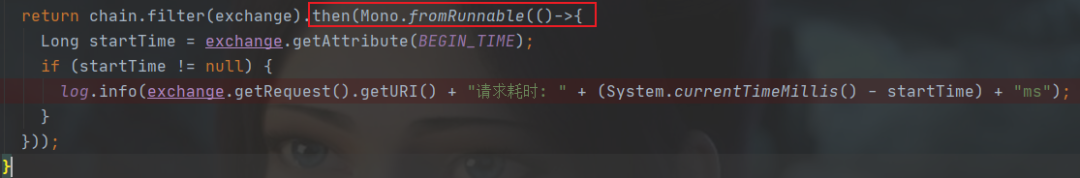
此时直接访问：http://localhost:9023/gateway/provider/port，不携带token，返回如下图：



请求参数带上token：http://localhost:9023/gateway/provider/port?token=abcdcdecd-ddcdeicd12，成功返回，如下图：

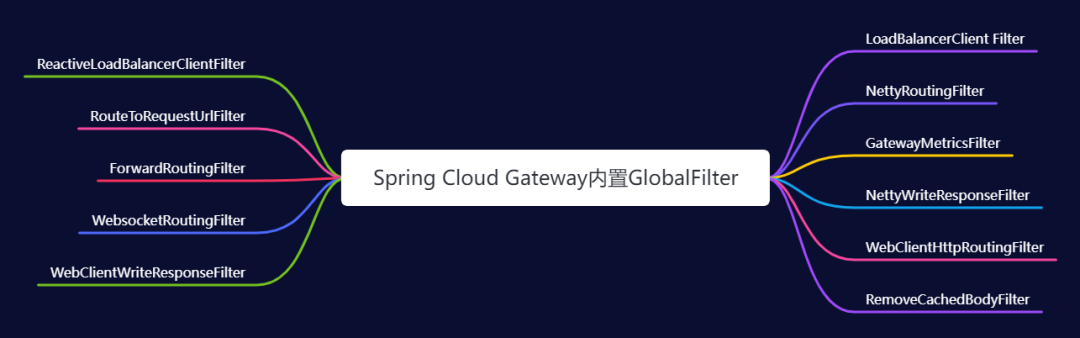


上述的 AuthorizeGatewayFilterFactory 只是涉及到了过滤器的前置处理，后置处理是在 chain.filter().then() 中的 then() 方法中完成的，具体可以看下项目源码中的 TimeGatewayFilterFactory，代码就不再贴出来了，如下图：



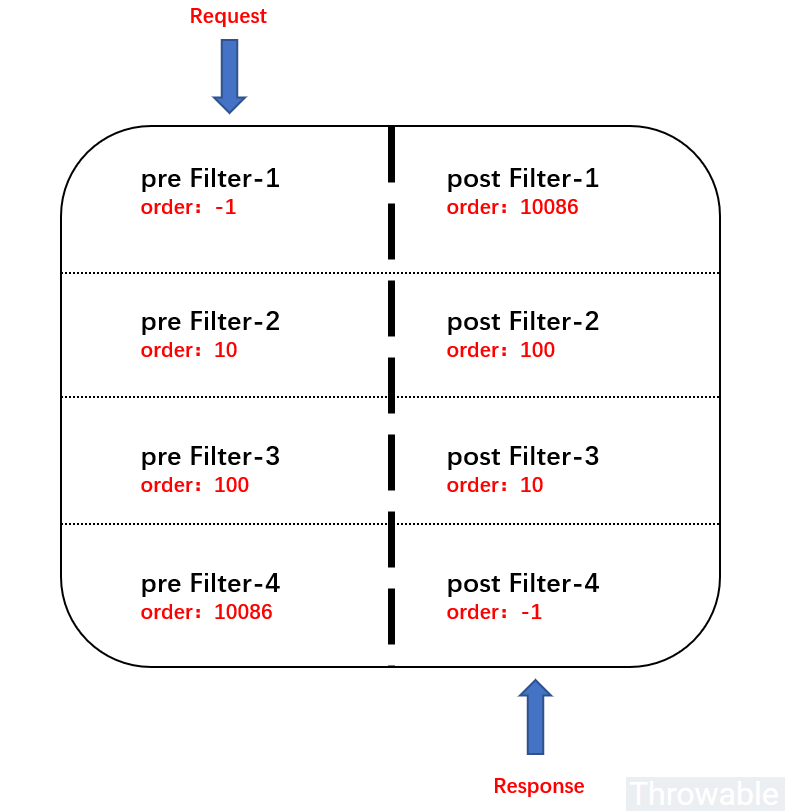
（3）GlobalFilter 全局过滤器：

全局过滤器应用全部路由上，无需开发者配置，Spring Cloud Gateway 也内置了一些全局过滤器，如下图：



GlobalFilter 的功能其实和 GatewayFilter 是相同的，只是 GlobalFilter 的作用域是所有的路由配置，而不是绑定在指定的路由配置上。多个 GlobalFilter 可以通过 @Order 或者 getOrder() 方法指定执行顺序，order值越小，执行的优先级越高。

注意，由于过滤器有 pre 和 post 两种类型，pre 类型过滤器如果 order 值越小，那么它就应该在pre过滤器链的顶层，post 类型过滤器如果 order 值越小，那么它就应该在 post 过滤器链的底层。示意图如下：



（4）自定义全局过滤器：

当然除了内置的全局过滤器，实际工作中还需要定制过滤器，下面来介绍一下如何自定义。我们模拟 Nginx 的 Access Log 功能，记录每次请求的相关信息。代码如下：

@Slf4j  
@Component  
@Order(value = Integer.MIN\_VALUE)  
public class AccessLogGlobalFilter implements GlobalFilter {  
   
    @Override  
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  
        *//filter的前置处理*  
        ServerHttpRequest request = exchange.getRequest();  
        String path = request.getPath().pathWithinApplication().value();  
        InetSocketAddress remoteAddress = request.getRemoteAddress();  
        return chain  
                *//继续调用filter*  
                .filter(exchange)  
                *//filter的后置处理*  
                .then(Mono.fromRunnable(() -> {  
            ServerHttpResponse response = exchange.getResponse();  
            HttpStatus statusCode = response.getStatusCode();  
            log.info("请求路径:{},远程IP地址:{},响应码:{}", path, remoteAddress, statusCode);  
        }));  
    }  
}

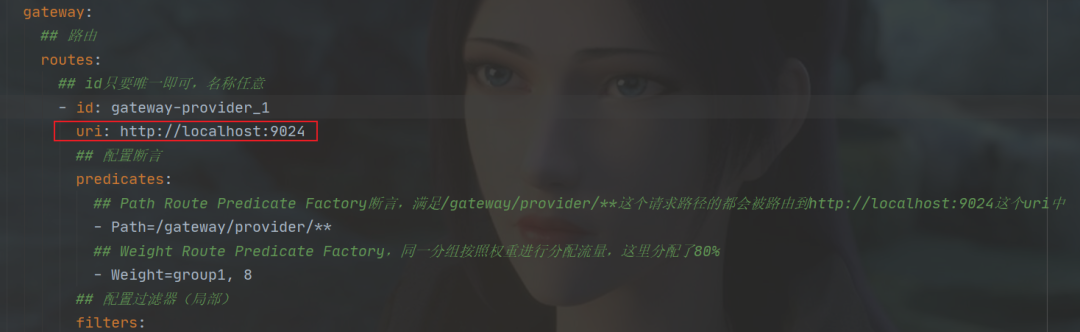
好了，全局过滤器不必在路由上配置，注入到IOC容器中即可全局生效。

此时发出一个请求，控制台打印信息如下：

请求路径:/gateway/provider/port，远程IP地址:/0:0:0:0:0:0:0:1:64114，响应码:200 OK

### 4、Gateway 集成 nacos 注册中心实现服务发现：

上述 demo 中并没有集成注册中心，每次路由配置都是指定固定的服务uri，如下图：



这样做有什么坏处呢？

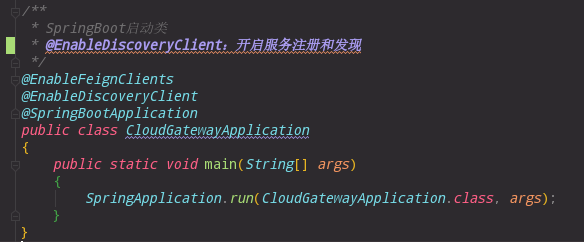
* 网关服务需要知道所有服务的域名或IP地址，另外，一旦服务的域名或IP地址发生修改，路由配置中的 uri 就必须修改
* 服务集群中无法实现负载均衡

那么此时我们可以集成的注册中心，使得网关能够从注册中心自动获取uri，并实现负载均衡，这里我们以 nacos 注册中心为例介绍一下

（1）pom 文件中新增依赖：

*<!--nacos注册中心-->*  
<dependency>  
 <groupId>com.alibaba.cloud</groupId>  
 <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>  
</dependency>

（2）启动类添加 @EnableDiscoveryClient 注解开启注册中心功能，如下图：



（3）配置 nacos 注册中心的地址：

nacos:  
  namespace: 856a40d7-6548-4494-bdb9-c44491865f63  
  url: 120.76.129.106:80  
spring:  
  cloud:  
    nacos:  
      discovery:  
       server-addr: ${nacos.url}  
        namespace: ${nacos.namespace}  
        register-enabled: true

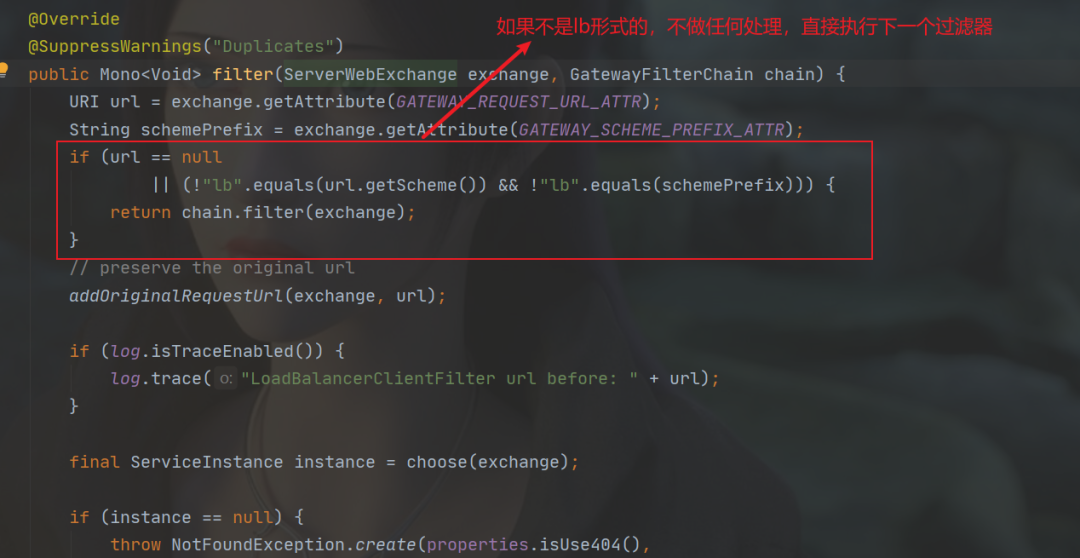
（4）服务路由配置：

spring:  
  cloud:  
    gateway:  
      routes:  
        - id: gateway-provider\_1  
          *## 使用了lb形式，从注册中心负载均衡的获取uri*  
          uri: lb://gateway-provider  
          *## 配置断言*  
          predicates:  
            - Path=/gateway/provider/\*\*  
          filters:  
            - AddResponseHeader=X-Response-Foo, Bar

路由配置中唯一不同的就是路由的 uri，格式：lb://service-name，这是固定写法：

* lb：固定格式，指的是从nacos中按照名称获取微服务，并遵循负载均衡策略
* service-name：nacos注册中心的服务名称，这里并不是IP地址形式的

为什么指定了 lb 就可以开启负载均衡，前面说过全局过滤器 LoadBalancerClientFilter 就是负责路由寻址和负载均衡的，可以看到如下源码：



（5）开启 gateway 自动路由配置：

随着我们的系统架构不断地发展，系统中微服务的数量肯定会越来越多，我们不可能每添加一个服务，就在网关配置一个新的路由规则，这样的维护成本很大；特别在很多种情况，我们在请求路径中会携带一个路由标识方便进行转发，而这个路由标识一般都是服务在注册中心中的服务名，因此这是我们就可以开启 spring cloud gateway 的自动路由功能，网关自动根据注册中心的服务名为每个服务创建一个router，将以服务名开头的请求路径转发到对应的服务，配置如下：

*# enabled：默认为false，设置为true表明spring cloud gateway开启服务发现和路由的功能，网关自动根据注册中心的服务名为每个服务创建一个router，将以服务名开头的请求路径转发到对应的服务*  
spring.cloud.gateway.discovery.locator.enabled = true  
*# lowerCaseServiceId：启动 locator.enabled=true 自动路由时，路由的路径默认会使用大写ID，若想要使用小写ID，可将lowerCaseServiceId设置为true*  
spring.cloud.gateway.discovery.locator.lower-case-service-id = true

这里需要注意的是，由于我们的网关项目配置了 server.servlet.context-path 属性，这会导致自动路由失败的问题，因此我们需要做如下两个修改：

*# 重写过滤链，解决项目设置了 server.servlet.context-path 导致 locator.enabled=true 默认路由策略404的问题*  
spring.cloud.gateway.discovery.locator.filters[0] = PreserveHostHeader

@Configuration  
public class GatewayConfig  
{  
    @Value ("${server.servlet.context-path}")  
    private String prefix;  
   
    */\*\*  
     \* 过滤 server.servlet.context-path 属性配置的项目路径，防止对后续路由策略产生影响，因为 gateway 网关不支持 servlet  
     \*/*  
    @Bean  
    @Order (-1)  
    public WebFilter apiPrefixFilter()  
    {  
        return (exchange, chain) ->  
        {  
            ServerHttpRequest request = exchange.getRequest();  
            String path = request.getURI().getRawPath();  
   
            path = path.startsWith(prefix) ? path.replaceFirst(prefix, "") : path;  
            ServerHttpRequest newRequest = request.mutate().path(path).build();  
   
            return chain.filter(exchange.mutate().request(newRequest).build());  
        };  
    }  
}

至此，我们就开启了 spring cloud gateway 的自动路由功能，网关自动根据注册中心的服务名为每个服务创建一个router，将以服务名开头的请求路径转发到对应的服务。

假设我们的服务提供者在 nacos 注册中心的服务名为 “gateway-provider”，这时我们只需要访问 “http://localhost:9023/gateway/gateway-provider/test”，就可以将请求成功转发过去了

### 5、Gateway 整合 Apollo 实现动态路由配置：

上述例子都是将网关的一系列配置写到项目的配置文件中，一旦路由策略发生改变必须要重启项目，这样维护成本很高，特别是服务网关作为系统的中心点，一旦重启出现问题，影响面将是十分巨大的，因此，我们将网关的配置存放到配置中心中，这样由配置中心统一管理，一旦路由发生改变，只需要在配置中心修改即可，降低风险且实时失效。本部分就以 Apollo 配置中心为例介绍下如下实现动态路由配置：

（1）添加 apollo 配置中心依赖：

*<!-- Apollo 统一配置中心 -->*  
<dependency>  
    <groupId>com.ctrip.framework.apollo</groupId>  
    <artifactId>apollo-client</artifactId>  
    <version>1.7.0</version>  
</dependency>

（2）添加 Apollo 路由更改监听刷新类：

import com.ctrip.framework.apollo.enums.PropertyChangeType;  
import com.ctrip.framework.apollo.model.ConfigChange;  
import com.ctrip.framework.apollo.model.ConfigChangeEvent;  
import com.ctrip.framework.apollo.spring.annotation.ApolloConfigChangeListener;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.beans.BeansException;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.cloud.context.environment.EnvironmentChangeEvent;  
import org.springframework.cloud.gateway.config.GatewayProperties;  
import org.springframework.cloud.gateway.event.RefreshRoutesEvent;  
import org.springframework.cloud.gateway.route.RouteDefinitionWriter;  
import org.springframework.context.ApplicationContext;  
import org.springframework.context.ApplicationContextAware;  
import org.springframework.context.ApplicationEventPublisher;  
import org.springframework.context.ApplicationEventPublisherAware;  
import org.springframework.context.annotation.Configuration;  
   
import java.util.ArrayList;  
   
*/\*\*  
 \* Apollo路由更改监听刷新  
 \*/*  
@Configuration  
public class GatewayPropertRefresher implements ApplicationContextAware, ApplicationEventPublisherAware  
{  
    private static final Logger logger = LoggerFactory.getLogger(GatewayPropertRefresher.class);  
   
    private static final String ID\_PATTERN = "spring\\.cloud\\.gateway\\.routes\\[\\d+\\]\\.id";  
   
    private static final String DEFAULT\_FILTER\_PATTERN = "spring\\.cloud\\.gateway\\.default-filters\\[\\d+\\]\\.name";  
   
   
    private ApplicationContext applicationContext;  
   
    private ApplicationEventPublisher publisher;  
   
    @Autowired  
    private  GatewayProperties gatewayProperties;  
   
    @Autowired  
    private RouteDefinitionWriter routeDefinitionWriter;  
   
   
    @Override  
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {  
        this.applicationContext = applicationContext;  
    }  
   
    @Override  
    public void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher) {  
        this.publisher = applicationEventPublisher;  
    }  
   
   
    */\*\*  
     \* 监听路由修改  
     \*/*  
    @ApolloConfigChangeListener(interestedKeyPrefixes = "spring.cloud.gateway.")  
    public void onChange(ConfigChangeEvent changeEvent)  
    {  
        refreshGatewayProperties(changeEvent);  
    }  
   
    */\*\*  
     \* 刷新路由信息  
     \*/*  
    private void refreshGatewayProperties(ConfigChangeEvent changeEvent)  
    {  
        logger.info("gateway网关配置 刷新开始！");  
   
        preDestroyGatewayProperties(changeEvent);  
        *//更新配置*  
        this.applicationContext.publishEvent(new EnvironmentChangeEvent(changeEvent.changedKeys()));  
        *//更新路由*  
        refreshGatewayRouteDefinition();  
   
        logger.info("gateway网关配置 刷新完成！");  
    }  
   
    */\*\*\*  
     \* GatewayProperties没有@PreDestroy和destroy方法  
     \* org.springframework.cloud.context.properties.ConfigurationPropertiesRebinder#rebind(java.lang.String)中destroyBean时不会销毁当前对象  
     \* 如果把spring.cloud.gateway.前缀的配置项全部删除（例如需要动态删除最后一个路由的场景），initializeBean时也无法创建新的bean，则return当前bean  
     \* 若仍保留有spring.cloud.gateway.routes[n]或spring.cloud.gateway.default-filters[n]等配置，initializeBean时会注入新的属性替换已有的bean  
     \* 这个方法提供了类似@PreDestroy的操作，根据配置文件的实际情况把org.springframework.cloud.gateway.config.GatewayProperties#routes  
     \* 和org.springframework.cloud.gateway.config.GatewayProperties#defaultFilters两个集合清空  
     \*/*  
    private synchronized void preDestroyGatewayProperties(ConfigChangeEvent changeEvent)  
    {  
        logger.info("Pre Destroy GatewayProperties 操作开始!");  
   
        final boolean needClearRoutes = this.checkNeedClear(changeEvent, ID\_PATTERN, this.gatewayProperties.getRoutes().size());  
        if (needClearRoutes)  
        {  
            this.gatewayProperties.setRoutes(new ArrayList());  
        }  
   
        final boolean needClearDefaultFilters = this.checkNeedClear(changeEvent, DEFAULT\_FILTER\_PATTERN, this.gatewayProperties.getDefaultFilters().size());  
        if (needClearDefaultFilters)  
        {  
            this.gatewayProperties.setRoutes(new ArrayList());  
        }  
   
        logger.info("Pre Destroy GatewayProperties 操作完成!");  
    }  
   
   
    private void refreshGatewayRouteDefinition()  
    {  
        logger.info("Refreshing Gateway RouteDefinition 操作开始!");  
   
        this.publisher.publishEvent(new RefreshRoutesEvent(this));  
   
        logger.info("Gateway RouteDefinition refreshed 操作完成!");  
    }  
   
    */\*\*\*  
     \* 根据changeEvent和定义的pattern匹配key，如果所有对应PropertyChangeType为DELETED则需要清空GatewayProperties里相关集合  
     \*/*  
    private boolean checkNeedClear(ConfigChangeEvent changeEvent, String pattern, int existSize) {  
   
        return changeEvent.changedKeys().stream().filter(key -> key.matches(pattern)).filter(key ->  
        {  
            ConfigChange change = changeEvent.getChange(key);  
            return PropertyChangeType.DELETED.equals(change.getChangeType());  
        }).count() == existSize;  
    }  
}

（3）暴露endpoint端点：

*# 暴露endpoint端点，暴露路由信息，有获取所有路由、刷新路由、查看单个路由、删除路由等方法*  
management.endpoints.web.exposure.include = \*  
management.endpoint.health.show-details = always

至此，我们就完成了 Gateway 网关整合 Apollo 配置中心实现动态路由配置，一旦路由发生改变，只需要在配置中心修改即可被监听到并实时失效

如果有整合 Nacos 或 MySQL 进行动态路由配置的读者可以参考以下两篇文章：

（1）整合 Nacos 进行动态路由配置：

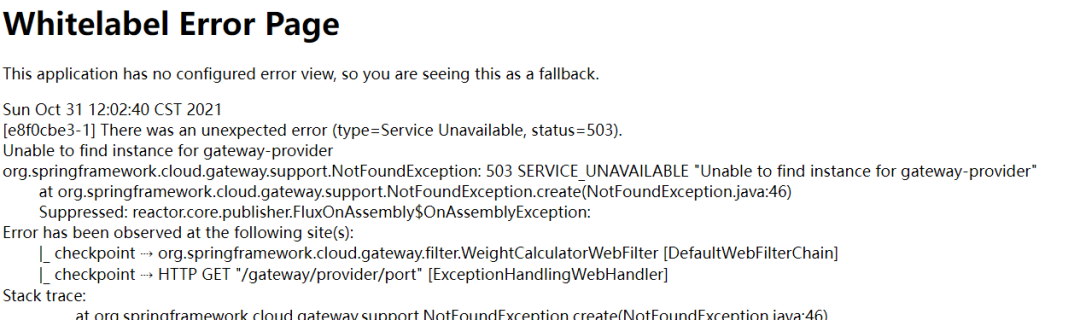
https://www.cnblogs.com/jian0110/p/12862569.html

（2）整合 MySQL 进行动态路由配置：

https://blog.csdn.net/qq\_42714869/article/details/92794911

### 6、自定义全局异常处理器：

通过前面的测试可以看到一个现象：一旦路由的微服务下线或者失联了，Spring Cloud Gateway直接返回了一个错误页面，如下图：



显然这种异常信息不友好，前后端分离架构中必须定制返回的异常信息。传统的Spring Boot 服务中都是使用 @ControllerAdvice 来包装全局异常处理的，但是由于服务下线，请求并没有到达。因此必须在网关中也要定制一层全局异常处理，这样才能更加友好的和客户端交互。

Spring Cloud Gateway提供了多种全局处理的方式，今天只介绍其中一种方式，实现还算比较优雅：

直接创建一个类 GlobalErrorExceptionHandler，实现 ErrorWebExceptionHandler，重写其中的 handle 方法，代码如下：

*/\*\*  
 \* 用于网关的全局异常处理  
 \* @Order(-1)：优先级一定要比ResponseStatusExceptionHandler低  
 \*/*  
@Slf4j  
@Order(-1)  
@Component  
@RequiredArgsConstructor  
public class GlobalErrorExceptionHandler implements ErrorWebExceptionHandler {  
   
 private final ObjectMapper objectMapper;  
   
 @SuppressWarnings({"rawtypes", "unchecked", "NullableProblems"})  
 @Override  
 public Mono<Void> handle(ServerWebExchange exchange, Throwable ex) {  
  ServerHttpResponse response = exchange.getResponse();  
  if (response.isCommitted()) {  
   return Mono.error(ex);  
  }  
   
  *// JOSN格式返回*  
  response.getHeaders().setContentType(MediaType.APPLICATION\_JSON);  
  if (ex instanceof ResponseStatusException) {  
   response.setStatusCode(((ResponseStatusException) ex).getStatus());  
  }  
   
  return response.writeWith(Mono.fromSupplier(() -> {  
   DataBufferFactory bufferFactory = response.bufferFactory();  
   try {  
    *//todo 返回响应结果，根据业务需求，自己定制*  
    CommonResponse resultMsg = new CommonResponse("500",ex.getMessage(),null);  
    return bufferFactory.wrap(objectMapper.writeValueAsBytes(resultMsg));  
   }  
   catch (JsonProcessingException e) {  
    log.error("Error writing response", ex);  
    return bufferFactory.wrap(new byte[0]);  
   }  
  }));  
 }  
}

好了，全局异常处理已经定制完成了，在测试一下，此时正常返回JSON数据了（JSON的样式根据架构需要自己定制），如下图：

