

1. 网关介绍

1.1 问题

前面的课程中, 我们通过Eureka, Nacos解决了服务注册, 服务发现的问题, 使用Spring Cloud LoadBalance解决了负载均衡的问题, 使用OpenFeign解决了远程调用的问题.

但是当前所有微服务的接口都是直接对外暴露的, 可以直接通过外部访问. 为了保证对外服务的安全性, 服务端实现的微服务接口通常都带有一定的权限校验机制. 由于使用了微服务, 原本一个应用的多个模块拆分成了多个应用, 我们不得不实现多次校验逻辑. 当这套逻辑需要修改时, 我们需要修改多个应用, 加重了开发人员的负担.

针对以上问题, 一个常用的解决方案是使用API网关.

比如企业管理

外部人员去公司办理业务, 公司需要先核实对方的身份再去进行办理.

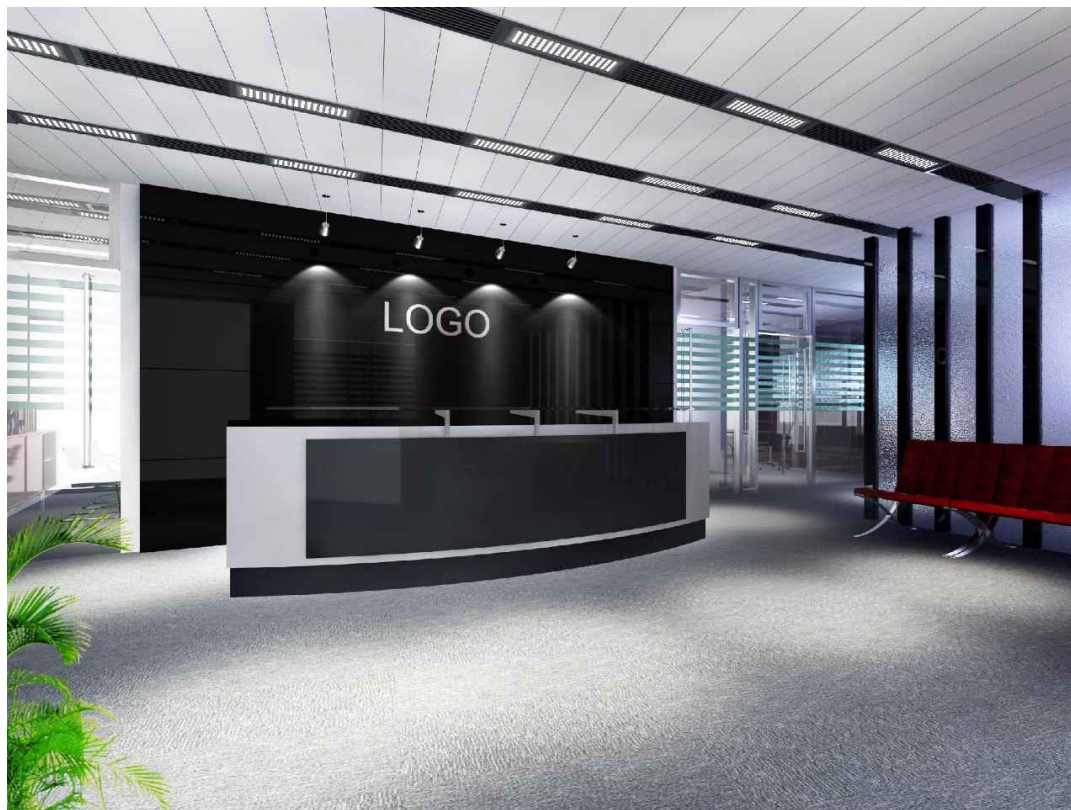
最开始只有一个员工, 这个员工核实之后直接办理即可. (单体架构)

随着公司的发展, 划分了多个部门, 每个部门负责的事情不同, 每个部门都需要先核实对方的身份再进行办理. (微服务架构)

这个流程存在一些问题:

1. 办事效率低
2. 增加了员工的工作流程

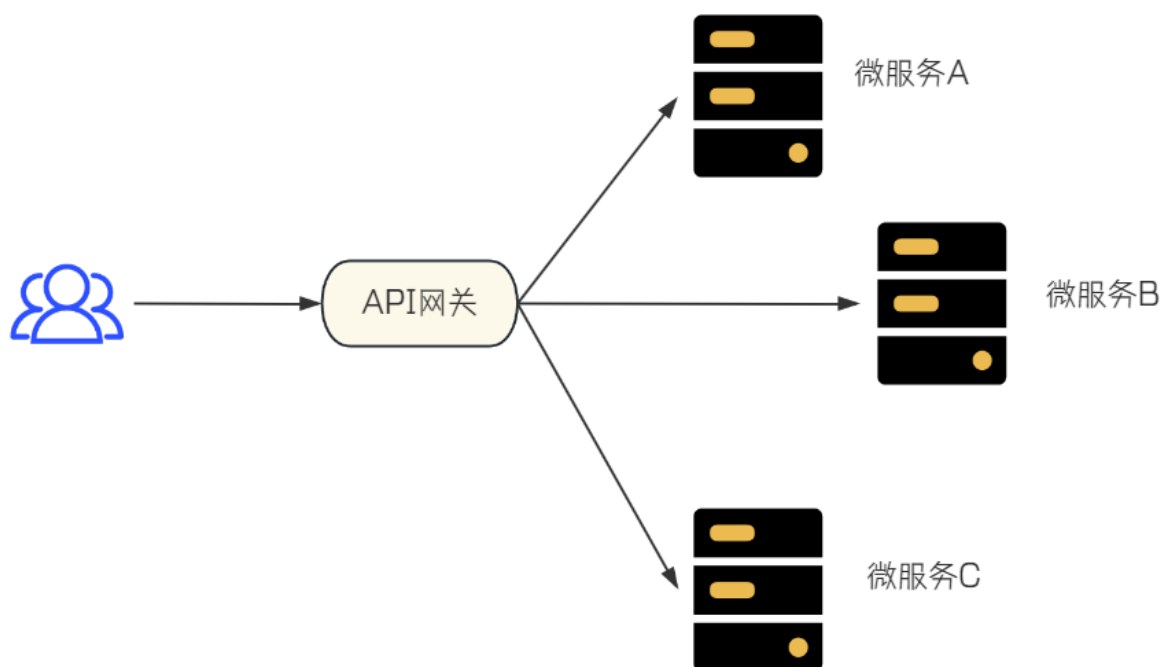
我们对此进行改进, 设立前台, 统一由前台来进行身份的校验. 前台身份校验通过后, 其他部门就设置信任, 直接办理.



前台就类似API网关。

1.2 什么是API网关

API网关(简称网关)也是一个服务,通常是后端服务的唯一入口. 它的定义类似设计模式中的Facade模式(门面模式,也称外观模式). 它就类似整个微服务架构的门面,所有的外部客户端访问,都需要经过它来进行调度和过滤.



网关核心功能:

权限控制: 作为微服务的入口, 对用户进行权限校验, 如果校验失败则进行拦截

动态路由: 一切请求先经过网关, 但网关不处理业务, 而是根据某种规则, 把请求转发到某个微服务

负载均衡: 当路由的目标服务有多个时, 还需要做负载均衡

限流: 请求流量过高时, 按照网关中配置微服务能够接受的流量进行放行, 避免服务压力过大.

类似前台的工作

1. 权限控制: 身份验证
2. 动态路由: 根据外来客户的需求, 把客户带到指定的部门去处理
3. 负载均衡: 一个部门有很多人时, 前台会帮客户选择具体某个人处理
4. 限流: 公司到访客户较多时, 进行流量限制, 比如告知明天再来

1.3 常见网关实现

业界常用的网关方式有很多, 技术方案也较成熟, 其中不乏很多开源产品, 比如Nginx, Kong, Zuul, Spring Cloud Gateway等. 下面介绍两种常见的网关方案.

Zuul

Zuul 是 Netflix 公司开源的一个API网关组件, 是Spring Cloud Netflix 子项目的核心组件之一, 它可以和 Eureka、Ribbon、Hystrix 等组件配合使用.

在Spring Cloud Finchley正式版之前, Spring Cloud推荐的网关是Netflix提供的Zuul(此处指Zuul 1.X). 然而Netflix在2018年宣布一部分组件进入维护状态, 不再进行新特性的开发. 这部分组件中就包含Zuul.

Spring Cloud Gateway

Spring Cloud Gateway 是Spring Cloud的一个全新的API网关项目, 基于Spring + SpringBoot等技术开发, 目的是为了替换掉Zuul. 旨在为微服务架构提供一种简单而有效的途径来转发请求, 并为他们提供横切关注点, 比如: 安全性, 监控/指标和弹性.

在性能方面, 根据官方提供的测试报告, Spring Cloud Gateway的RPS(每秒请求数)是Zuul的1.6倍. 测试报告参考: <https://github.com/spencergibb/spring-cloud-gateway-bench>

2. Spring Cloud Gateway

2.1 快速上手

我们通过以下的演示, 先来了解网关的基本功能

2.1.1 创建网关项目

API网关也是一个服务.

New Module

New Module

Generators

m

Maven Archetype

JavaFX

Compose Multiplatform

IDE Plugin

Android

Spring Initializr

Name:

gateway

Location:

D:\Git\spring-cloud\spring-cloud-gateway

Module will be created in: D:\Git\spring-cloud

Language:

JavaGroovyHTML+

Build system:

IntelliJ MavenGradle

JDK:

Project SDK 17

Parent:

m spring-c...d-gateway

☐ Add sample code

> Advanced Settings

?

Create

Cancel

2.1.2 引入网关依赖

```
1 <!--网关-->
2 <dependency>
3     <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-starter-gateway</artifactId>
5 </dependency>
6 <!--基于nacos实现服务发现依赖-->
7 <dependency>
8     <groupId>com.alibaba.cloud</groupId>
9     <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
10 </dependency>
11 <!--负载均衡-->
12 <dependency>
```

```
13     <groupId>org.springframework.cloud</groupId>
14     <artifactId>spring-cloud-starter-loadbalancer</artifactId>
15 </dependency>
```

2.1.3 编写启动类

```
1 package com.bite.gateway;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class GatewayApplication {
8     public static void main(String[] args) {
9         SpringApplication.run(GatewayApplication.class,args);
10    }
11 }
```

2.1.4 添加Gateway的路由配置

创建application.yml文件, 添加如下配置:

```
1 server:
2   port: 10030 # 网关端口
3 spring:
4   application:
5     name: gateway # 服务名称
6   cloud:
7     nacos:
8       discovery:
9         server-addr: 110.41.51.65:10020
10  gateway:
11    routes: # 网关路由配置
12      - id: product-service #路由ID, 自定义, 唯一即可
13        uri: lb://product-service #目标服务地址
14        predicates: #路由条件
15          - Path=/product/**
16      - id: order-service
17        uri: lb://order-service
18        predicates:
19          - Path=/order/**
```

配置字段说明:

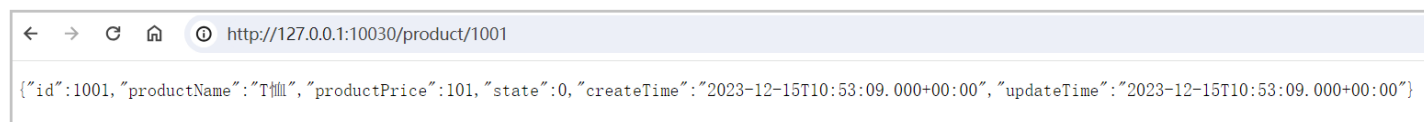
- id: 自定义路由ID, 保持唯一
- uri: 目标服务地址, 支持普通URI 及 `lb://应用注册服务名称`. lb表示负载均衡, 使用 `lb://` 方式表示从注册中心获取服务地址.
- predicates: 路由条件, 根据匹配结果决定是否执行该请求路由, 上述代码中, 我们把符合Path规则的一切请求, 都代理到uri参数指定的地址.

2.1.5 测试

启动API网关服务

1. 通过网关服务访问product-service

<http://127.0.0.1:10030/product/1001>



url符合 yml文件中配置的 `/product/**` 规则, 路由转发到product-service: <http://product-service/product/1001>

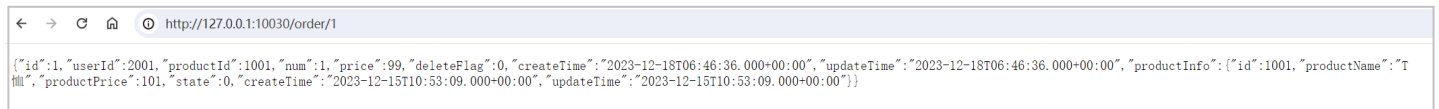
访问时, 观察网关日志, 可以看到网关服务从Nacos时获取服务列表;

```
1 2024-01-04T19:26:12.457+08:00 INFO 9004 --- [oundedElastic-1]
  com.alibaba.nacos.client.naming          : [SUBSCRIBE-SERVICE] service:product-
  service, group:DEFAULT_GROUP, clusters:
2 2024-01-04T19:26:12.498+08:00 INFO 9004 --- [oundedElastic-1]
  com.alibaba.nacos.client.naming          : init new ips(1) service:
  DEFAULT_GROUP@@product-service ->
  [{"ip":"192.168.31.90","port":9090,"weight":1.0,"healthy":true,"enabled":true,"
  ephemeral":true,"clusterName":"SH","serviceName":"DEFAULT_GROUP@@product-
  service","metadata":
  {"preserved.register.source":"SPRING_CLOUD"},"ipDeleteTimeout":30000,"instanceH
  eartBeatInterval":5000,"instanceHeartBeatTimeOut":15000}]
3 2024-01-04T19:26:12.498+08:00 INFO 9004 --- [oundedElastic-1]
  com.alibaba.nacos.client.naming          : current ips:(1) service:
  DEFAULT_GROUP@@product-service ->
  [{"ip":"192.168.31.90","port":9090,"weight":1.0,"healthy":true,"enabled":true,"
  ephemeral":true,"clusterName":"SH","serviceName":"DEFAULT_GROUP@@product-
  service","metadata":
  {"preserved.register.source":"SPRING_CLOUD"},"ipDeleteTimeout":30000,"instanceH
  eartBeatInterval":5000,"instanceHeartBeatTimeOut":15000}]
```

```
4 2024-01-04T19:26:13.009+08:00 INFO 9004 --- [110.41.51.65-31]
com.alibaba.nacos.common.remote.client : [8e0a372f-7243-49ba-86ac-
173f0309a0a0] Receive server push request, request = NotifySubscriberRequest,
requestId = 6
5 2024-01-04T19:26:13.009+08:00 INFO 9004 --- [110.41.51.65-31]
com.alibaba.nacos.common.remote.client : [8e0a372f-7243-49ba-86ac-
173f0309a0a0] Ack server push request, request = NotifySubscriberRequest,
requestId = 6
```

2. 通过网关服务访问order-service

<http://127.0.0.1:10030/order/1>



```
{ "id": 1, "userId": 2001, "productId": 1001, "num": 1, "price": 99, "deleteFlag": 0, "createTime": "2023-12-18T06:46:36.000+00:00", "updateTime": "2023-12-18T06:46:36.000+00:00", "productInfo": { "id": 1001, "productName": "T恤", "productPrice": 101, "state": 0, "createTime": "2023-12-15T10:53:09.000+00:00", "updateTime": "2023-12-15T10:53:09.000+00:00" }}
```

url符合yml文件中配置的 `/order/**` 规则, 路由转发到product-service: <http://order-service/product/1001>

2.2 Route Predicate Factories

2.2.1 Predicate

Predicate是Java 8提供的一个函数式编程接口, 它接收一个参数并返回一个布尔值, 用于条件过滤, 请求参数的校验。

```
1 @FunctionalInterface
2 public interface Predicate<T> {
3     boolean test(T t);
4     //...
5 }
```

代码演示:

1. 定义一个Predicate

```
1 class StringPredicate implements Predicate<String>{
2     @Override
3     public boolean test(String str) {
4         return str.isEmpty();
5     }
6 }
```

2. 使用这个Predicate

```
1 public class PredictTest {
2     public static void main(String[] args) {
3         Predicate<String> predicate = new StringPredicate();
4         System.out.println(predicate.test(""));
5         System.out.println(predicate.test("bite666"));
6     }
7
8 }
```

3. 运行结果

```
"D:\Program Files\Java\jdk-17\bin\java.exe" ...
true
false

Process finished with exit code 0
```

4. Predicate的其他写法

1) 内置函数

```
1 public class PredictTest {
2     public static void main(String[] args) {
3         Predicate<String> predicate = new Predicate<String>(){
4             @Override
5             public boolean test(String s) {
6                 return s.isEmpty();
7             }
8         };
9         System.out.println(predicate.test(""));
10        System.out.println(predicate.test("bite666"));
11    }
12 }
```

2) lambda写法

```
1 public class PredictTest {
2     public static void main(String[] args) {
```



```

3      Predicate<String> predicate = s -> s.isEmpty();
4      System.out.println(predicate.test(""));
5      System.out.println(predicate.test("bite666"));
6  }
7  }

```

`Predicate<String> predicate = s -> s.isEmpty();` 也可以写成

`Predicate<String> isEmpty = String::isEmpty;`

5. Predicate 的其他方法

- `isEqual(Object targetRef)` :比较两个对象是否相等,参数可以为Null
- `and(Predicate other)`: 短路与操作,返回一个组成Predicate
- `or(Predicate other)` :短路或操作,返回一个组成Predicate
- `test(T t)` :传入一个Predicate参数,用来做判断
- `negate()` : 返回表示此Predicate逻辑否定的Predicate

方法	说明
<code>boolean test(T t)</code>	判断条件, 可以理解为 条件A 根据逻辑返回布尔值
<code>Predicate<T> and(Predicate<? super T> other)</code>	条件A && 条件B 当前Predicate的test方法 && other的test方法, 相当于进行两次判断
<code>default Predicate<T> negate()</code>	! 条件A 对当前判断进行"!"操作,即取非操作
<code>default Predicate<T> or(Predicate<? super T> other)</code>	条件A 条件B 当前Predicate的test方法 other的test方法

2.2.2 Route Predicate Factories

Route Predicate Factories (路由断言工厂, 也称为路由谓词工厂, 此处谓词表示一个函数), 在Spring Cloud Gateway中, Predicate提供了路由规则的匹配机制.

我们在配置文件中写的断言规则只是字符串, 这些字符串会被Route Predicate Factory读取并处理, 转变为路由判断的条件.

比如前面章节配置的 `Path=/product/**`, 就是通过Path属性来匹配URL前缀是 `/product` 的请求.

这个规则是由 `org.springframework.cloud.gateway.handler.predicate.PathRoutePredicateFactory` 来实现的.

Spring Cloud Gateway 默认提供了很多Route Predicate Factory, 这些Predicate会分别匹配HTTP请求的不同属性, 并且多个Predicate可以通过and逻辑进行组合.

名称	说明	示例
After	这个工厂需要一个日期时间(Java的 ZonedDateTime对象), 匹配指定日期之后的请求	<pre>1 predicates: 2 - After=2017-01-20T17:42:47.789-07:00[America/Denver]</pre>
Before	匹配指定日期之前的请求	<pre>1 predicates: 2 - Before=2017-01-20T17:42:47.789-07:00[America/Denver]</pre>
Between	匹配两个指定时间之间的请求 <code>datetime2</code> 的参数必须在 <code>datetime1</code> 之后	<pre>1 predicates: 2 - Between=2017-01-20T17:42:47.789-07:00[America/Denver], 2017-01-21T17:42:47.789-07:00[America/Denver]</pre>
Cookie	请求中包含指定Cookie, 且该Cookie值符合指定的正则表达式	<pre>1 predicates: 2 - Cookie=chocolate, ch.p</pre>
Header	请求中包含指定Header, 且该Header值符合指定的正则表达式	<pre>1 predicates: 2 - Header=X-Request-Id, \d+</pre>
Host	请求必须是访问某个host(根据请求中的Host字段进行匹配)	<pre>1 predicates: 2 - Host=**.somehost.org,**.anotherhost.org</pre>

Method	匹配指定的请求方式	<pre> 1 predicates: 2 - Method=GET,POST </pre>
Path	匹配指定规则的路径	<pre> 1 predicates: 2 - Path=/red/{segment},/blue/{segment} </pre>
Remote Addr	请求者的IP必须为指定范围	<pre> 1 predicates: 2 - RemoteAddr=192.168.1.1/24 </pre>

更多请参考: <https://docs.spring.io/spring-cloud-gateway/reference/spring-cloud-gateway/request-predicates-factories.html>

2.2.3 代码演示

1. 添加Predicate规则

在application.yml中添加如下规则

```

1 spring:
2   cloud:
3     gateway:
4       routes: # 网关路由配置
5         - id: product-service #路由ID, 自定义, 唯一即可
6           uri: lb://product-service #目标服务地址
7           predicates: #路由条件
8             - Path=/product/**
9             - After=2025-01-01T00:00:00.000+08:00[Asia/Shanghai]

```

增加限制路由规则: 请求时间为2025年1月1日之后

2. 测试

访问: <http://127.0.0.1:10030/product/1001>

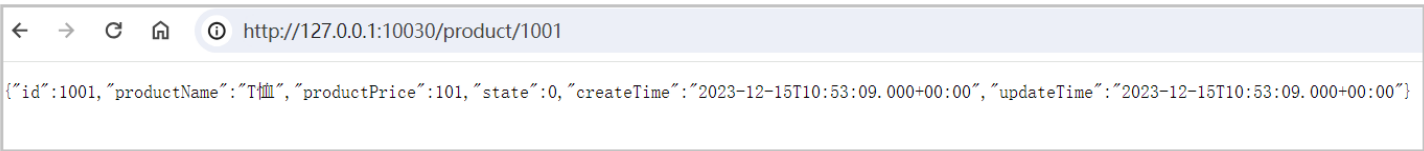
返回404



3. 修改时间为2024-01-01, 再次访问

```
1 - After=2024-01-01T00:00:00.000+08:00[Asia/Shanghai]
```

访问: <http://127.0.0.1:10030/product/1001>



2.3 Gateway Filter Factories(网关过滤器工厂)

Predicate决定了请求由哪一个路由处理, 如果在请求处理前后需要加一些逻辑, 这就是Filter(过滤器)的作用范围了.

Filter分为两种类型: Pre类型和Post类型.

Pre类型过滤器: 路由处理之前执行(请求转发到后端服务之前执行), 在Pre 类型过滤器中可以做鉴权, 限流等.

Post类型过滤器: 请求执行完成后, 将结果返回给客户端之前执行.

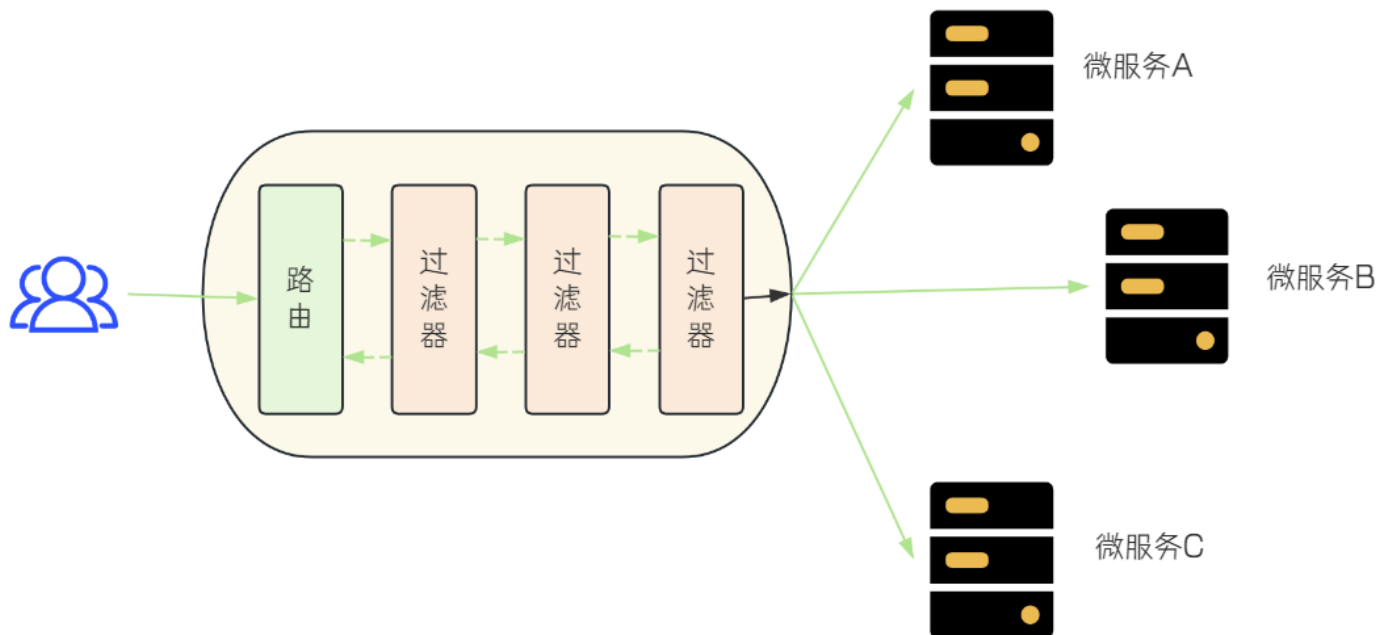
比如去景区玩

1. 进景区之前需要先安检, 验票(鉴权), 如果今日进景区的人超过了规定的人数, 就会进行限流

2. 接下来进景区游玩

3. 游玩之后, 对景区服务进行评价

1 就类似Pre类型过滤器, 3就类似Post类型过滤器, 过滤器可有可无.



Spring Cloud Gateway 中内置了很多Filter, 用于拦截和链式处理web请求. 比如权限校验, 访问超时等设定.

Spring Cloud Gateway从作用范围上, 把Filter可分为GatewayFilter 和GlobalFilter.

GatewayFilter: 应用到单个路由或者一个分组的路由上.

GlobalFilter: 应用到所有的路由上, 也就是对所有的请求生效.

2.3.1 GatewayFilter

GatewayFilter 同 Predicate 类似, 都是在配置文件 `application.yml` 中配置, 每个过滤器的逻辑都是固定的. 比如 `AddRequestParameterGatewayFilterFactory` 只需要在配置文件中写 `AddRequestParameter`, 就可以为所有的请求添加一个参数, 我们先通过一个例子来演示 GatewayFilter 如何使用.

快速上手

1. 在application.yml中添加filter

```
1 server:
2   port: 10030 # 网关端口
3 spring:
4   application:
5     name: gateway # 服务名称
6   cloud:
7     nacos:
8       discovery:
9         server-addr: 110.41.51.65:10020
10    gateway:
```

```

11     routes: # 网关路由配置
12         - id: product-service    #路由ID, 自定义, 唯一即可
13           uri: lb://product-service    #目标服务地址
14           predicates: #路由条件
15             - Path=/product/**
16             - After=2024-01-01T00:00:00.000+08:00[Asia/Shanghai]
17           filters:
18             - AddRequestParameter=username, bite
19         - id: order-service
20           uri: lb://order-service
21           predicates:
22             - Path=/order/**

```

该filter只添加在了 `product-service` 路由下, 因此只对 `product-service` 路由生效, 也就是对 `/product/**` 的请求生效.

2. 接收参数并打印

在 `product-service` 服务中接收请求的参数,并打印出来

```

1 @RequestMapping("/product")
2 @RestController
3 public class ProductController {
4     @Autowired
5     private ProductService productService;
6
7     @RequestMapping("/{productId}")
8     public ProductInfo getProductById(@PathVariable("productId") Integer
productId, String userName){
9         System.out.println("收到请求,Id:"+productId);
10        System.out.println("userName:"+userName);
11        return productService.selectProductById(productId);
12    }
13
14 }

```

3. 测试

重启gateway 和product-service服务,访问请求, 观察日志

<http://127.0.0.1:10030/product/1001>

控制台打印日志:

```

1 收到请求,Id:1001

```

GatewayFilter说明

Spring Cloud Gateway提供了的Filter非常多, 下面列出一些常见过滤器的说明.

详细可参考官方文档 [GatewayFilter Factories](#)

名称	说明	示例
AddRequestHeader	为当前请求添加Header	- AddRequestHeader=X-Request-red, blue 参数: Header的名称及值
AddRequestParameter	为当前请求添加请求参数	- AddRequestParameter=red, blue 参数: 参数的名称及值
AddResponseHeader	为响应结果添加Header	- AddResponseHeader=X-Response-Red, Blue 参数: Header的名称及值
RemoveRequestHeader	从当前请求删除某个Header	- RemoveRequestHeader=X-Request-Foo 参数: Header的名称
RemoveResponseHeader	从响应结果删除某个Header	- RemoveResponseHeader=X-Response-Foo 参数: Header的名称
RequestRateLimiter	为当前网关的所有请求执行限流过滤, 如果被限流, 默认会响应HTTP 429-Too ManyRequests 默认提供了RedisRateLimiter的限流实现, 采用令牌桶算法实现限流功能. 此处不做具体介绍	<pre>1 filters: 2 - name: RequestRateLimiter 3 args: 4 redis-rate-limiter.replenishRate: 10 5 redis-rate-limiter.burstCapacity: 20 6 redis-rate-limiter.requestedTokens: 1</pre> <div>redis-rate-limiter.replenishRate : 令牌填充速度, 即每秒钟允许多少个请求(不丢弃任何请求)</div> <div>redis-rate-limiter.burstCapacity : 令牌桶容量, 即每秒用户最大能够执行的请求数量(不丢弃任何请求). 将此值设置为零将阻止所有请求</div>

		<code>redis-rate-limiter.requestedTokens</code> : 每次请求占用几个令牌, 默认为 <code>1</code> 。
Retry	针对不同的响应进行重试. 当后端服务不可用时, 网关会根据配置参数来发起重试请求.	<pre> 1 filters: 2 - name: Retry 3 args: 4 retries: 3 5 statuses: BAD_REQUEST </pre> <p>retries: 重试次数, 默认为3</p> <p>status: HTTP请求返回的状态码, 针对指定状态码进行重试. 对应 <code>org.springframework.http.HttpStatus</code></p>
RequestSize	设置允许接收最大请求包的大小. 如果请求包大小超过设置的值, 则返回 413 Payload Too Large. 请求包大小, 单位为字节, 默认值为5M.	<pre> 1 filters: 2 - name: RequestSize 3 args: 4 maxSize: 5000000 </pre>
默认过滤器	添加一个filter并将其应用于所有路由, 这个属性需要一个filter的列表, 详细参考下面章节	

Default Filters

前面的filter添加在指定路由下, 所以只对当前路由生效, 若需要对全部路由生效, 可以使用

`spring.cloud.gateway.default-filters` 这个属性需要一个filter的列表.

配置举例:

```

1 spring:
2   cloud:
3     gateway:
4       default-filters:
5         - AddResponseHeader=X-Response-Default-Red, Default-Blue
6         - PrefixPath=/httpbin

```

2.3.2 GlobalFilter

GlobalFilter是Spring Cloud Gateway中的全局过滤器, 它和GatewayFilter的作用是相同的. GlobalFilter 会应用到所有的路由请求上, 全局过滤器通常用于实现与安全性, 性能监控和日志记录等相关的全局功能.

Spring Cloud Gateway 内置的全局过滤器也有很多, 比如:

- Gateway Metrics Filter: 网关指标, 提供监控指标
- Forward Routing Filter: 用于本地forward, 请求不转发到下游服务器
- LoadBalancer Client Filter: 针对下游服务, 实现负载均衡.
- ...

更多过滤器参考: [Global Filters](#)

快速上手

1. 添加依赖

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-actuator</artifactId>
4 </dependency>
```

2. 添加配置

```
1 spring:
2   cloud:
3     gateway:
4       metrics:
5         enabled: true
6 management:
7   endpoints:
8     web:
9       exposure:
10        include: "*"
11 endpoint:
12   health:
13     show-details: always
14 shutdown:
15   enabled: true
```

3. 测试

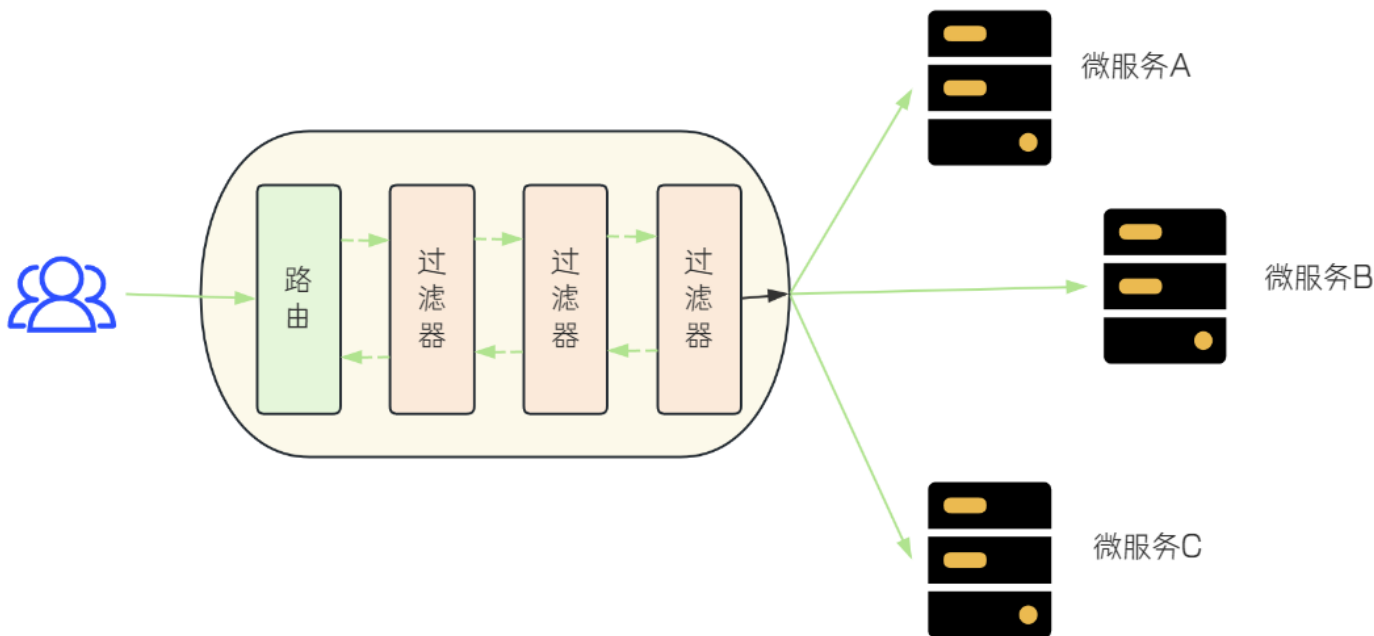
<http://127.0.0.1:10030/actuator>, 显示所有监控的信息链接

```
{
  "_links": {
    "self": {
      "href": "http://127.0.0.1:10030/actuator",
      "templated": false
    },
    "nacosdiscovery": {
      "href": "http://127.0.0.1:10030/actuator/nacosdiscovery",
      "templated": false
    },
    "beans": {
      "href": "http://127.0.0.1:10030/actuator/beans",
      "templated": false
    },
    "caches": {
      "href": "http://127.0.0.1:10030/actuator/caches",
      "templated": false
    },
    "caches-cache": {
      "href": "http://127.0.0.1:10030/actuator/caches/{cache}",
      "templated": true
    },
    "health": {
      "href": "http://127.0.0.1:10030/actuator/health/{#path}",
      "templated": true
    },
    "health": {
      "href": "http://127.0.0.1:10030/actuator/health",
      "templated": false
    },
    "info": {
      "href": "http://127.0.0.1:10030/actuator/info",
      "templated": false
    },
    "conditions": {
      "href": "http://127.0.0.1:10030/actuator/conditions",
      "templated": false
    },
    "configprops-prefix": {
      "href": "http://127.0.0.1:10030/actuator/configprops/{prefix}",
      "templated": true
    },
    "configprops": {
      "href": "http://127.0.0.1:10030/actuator/configprops",
      "templated": false
    },
    "env-toMatch": {
      "href": "http://127.0.0.1:10030/actuator/env/{toMatch}",
      "templated": true
    },
    "env": {
      "href": "http://127.0.0.1:10030/actuator/env",
      "templated": false
    },
    "loggers": {
      "href": "http://127.0.0.1:10030/actuator/loggers",
      "templated": false
    },
    "loggers-name": {
      "href": "http://127.0.0.1:10030/actuator/loggers/{name}",
      "templated": true
    },
    "heapdump": {
      "href": "http://127.0.0.1:10030/actuator/heapdump",
      "templated": false
    },
    "threaddump": {
      "href": "http://127.0.0.1:10030/actuator/threaddump",
      "templated": false
    },
    "metrics": {
      "href": "http://127.0.0.1:10030/actuator/metrics",
      "templated": false
    },
    "requiredMetricName": {
      "href": "http://127.0.0.1:10030/actuator/metrics/{requiredMetricName}",
      "templated": true
    },
    "scheduledtasks": {
      "href": "http://127.0.0.1:10030/actuator/scheduledtasks",
      "templated": false
    },
    "mappings": {
      "href": "http://127.0.0.1:10030/actuator/mappings",
      "templated": false
    },
    "refresh": {
      "href": "http://127.0.0.1:10030/actuator/refresh",
      "templated": false
    },
    "features": {
      "href": "http://127.0.0.1:10030/actuator/features",
      "templated": false
    }
  }
}
```

2.4 过滤器执行顺序

一个项目中, 既有GatewayFilter, 又有 GlobalFilter时, 执行的先后顺序是什么呢?

请求路由后, 网关会把当前项目中的GatewayFilter和GlobalFilter合并到一个过滤器链(集合)中, 并进行排序, 依次执行过滤器。



每一个过滤器都必须指定一个int类型的order值, 默认值为0, 表示该过滤的优先级. **order值越小, 优先级越高, 执行顺序越靠前。**

- Filter通过实现Order接口或者添加@Order注解来指定order值。
- Spring Cloud Gateway提供的Filter由Spring指定. 用户也可以自定义Filter, 由用户指定。
- 当过滤器的order值一样时, 会按照 defaultFilter > GatewayFilter > GlobalFilter的顺序执行。

2.5 自定义过滤器

Spring Cloud Gateway提供了过滤器的扩展功能, 开发者可以根据实际业务来自定义过滤器, 同样自定义过滤器也支持GatewayFilter 和 GlobalFilter两种。

2.5.1 自定义GatewayFilter

自定义GatewayFilter, 需要去实现对应的接口 `GatewayFilterFactory`, Spring Boot 默认帮我们实现的抽象类是 `AbstractGatewayFilterFactory`, 我们可以直接使用。

2.5.1.1 定义GatewayFilter

```

1  import lombok.Data;
2  import lombok.extern.slf4j.Slf4j;
3  import org.springframework.cloud.gateway.filter.GatewayFilter;
4  import
    org.springframework.cloud.gateway.filter.factory.AbstractGatewayFilterFactory;
5  import org.springframework.core.Ordered;
6  import org.springframework.stereotype.Service;
7  import reactor.core.publisher.Mono;
8
9  @Slf4j
10 @Service
11 public class CustomGatewayFilterFactory extends
    AbstractGatewayFilterFactory<CustomGatewayFilterFactory.CustomConfig>
    implements Ordered {
12     public CustomGatewayFilterFactory() {
13         super(CustomConfig.class);
14     }
15
16     @Override
17     public GatewayFilter apply(CustomConfig config) {
18         /**
19          * Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
    chain)
20          * ServerWebExchange: HTTP请求-响应交互的契约, 提供对HTTP请求和响应的访问, 服
    务器端请求属性, 请求实例, 响应实例等, 类似Context角色
21          * GatewayFilterChain: 过滤器链
22          * Mono: Reactor核心类, 数据流发布者, Mono最多只触发一个事件, 所以可以把
    Mono 用于在异步任务完成时发出通知.
23          * Mono.fromRunnable: 创建一个包含Runnable元素的数据流
24          */
25         return ((exchange, chain) -> {
26             log.info("[Pre] Filter Request, name:"+config.getName());
27             return chain.filter(exchange).then(Mono.fromRunnable(()->{
28                 log.info("[Post] Response Filter");
29             }));
30         });
31     }
32
33     @Override
34     public int getOrder() {
35         return Ordered.LOWEST_PRECEDENCE; //配置优先级, order越大, 优先级越低
36     }
37 }

```

针对这个Filter的配置, 使用CustomConfig 定义

```

1 @Data
2 public class CustomConfig {
3     private String name;
4 }

```

代码说明:

1. 类名统一以GatewayFilterFactory结尾, 因为默认情况下, 过滤器的name会采用该定义类的前缀. 这里的name=Custom(yml配置中使用)
2. apply方法中, 同时包含Pre和Post过滤, then方法中是请求执行结束之后处理的
3. CustomConfig 是一个配置类, 该类只有一个属性name, 和yml的配置对应
4. 该类需要交给Spring管理, 所以需要加 `@Service` 注解
5. getOrder表示该过滤器的优先级, 值越大, 优先级越低.

2.5.1.2 配置过滤器

```

1 spring:
2   cloud:
3     gateway:
4       routes: # 网关路由配置
5         - id: product-service    #路由ID, 自定义, 唯一即可
6           uri: lb://product-service #目标服务地址
7           predicates: #路由条件
8             - Path=/product/**
9           filters:
10            - name: Custom
11              args:
12                name: custom filter

```

2.5.1.3 测试

重启服务, 访问接口, 观察日志

<http://127.0.0.1:10030/product/1001>

```

1 2024-01-06T14:34:10.374+08:00 INFO 21260 --- [ctor-http-nio-2]
  c.b.g.filter.CustomGatewayFilterFactory : [Pre] Filter Request, name:custom
  filter
2 2024-01-06T14:34:10.385+08:00 INFO 21260 --- [ctor-http-nio-5]
  c.b.g.filter.CustomGatewayFilterFactory : [Post] Response Filter

```

2.5.2 自定义GlobalFilter

GlobalFilter的实现比较简单, 它不需要额外的配置, 只需要实现GlobalFilter接口, 会自动过滤所有的Filter.

2.5.2.1 定义GlobalFilter

```
1 import lombok.extern.slf4j.Slf4j;
2 import org.springframework.cloud.gateway.filter.GatewayFilterChain;
3 import org.springframework.cloud.gateway.filter.GlobalFilter;
4 import org.springframework.core.Ordered;
5 import org.springframework.stereotype.Service;
6 import org.springframework.web.server.ServerWebExchange;
7 import reactor.core.publisher.Mono;
8
9 @Slf4j
10 @Service
11 public class CustomGlobalFilter implements GlobalFilter, Ordered {
12     @Override
13     public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
        chain) {
14         log.info("[Pre] CustomGlobalFilter enter...");
15         return chain.filter(exchange).then(Mono.fromRunnable(()->{
16             log.info("[Post] CustomGlobalFilter return...");
17         }));
18     }
19
20     @Override
21     public int getOrder() {
22         return Ordered.LOWEST_PRECEDENCE; //配置优先级, order越大, 优先级越低
23     }
24 }
```

2.5.2.2 测试

重启服务, 访问接口, 观察日志

<http://127.0.0.1:10030/product/1001>

```
1 2024-01-06T14:58:55.869+08:00 INFO 37832 --- [ctor-http-nio-2]
   c.b.g.filter.CustomGatewayFilterFactory : [Pre] Filter Request, name:custom
   filter
2 2024-01-06T14:58:55.870+08:00 INFO 37832 --- [ctor-http-nio-2]
   c.b.gateway.filter.CustomGlobalFilter   : [Pre] CustomGlobalFilter enter...
```

```
3 2024-01-06T14:58:55.933+08:00 INFO 37832 --- [ctor-http-nio-5]
  c.b.gateway.filter.CustomGlobalFilter : [Post] CustomGlobalFilter return...
4 2024-01-06T14:58:55.934+08:00 INFO 37832 --- [ctor-http-nio-5]
  c.b.g.filter.CustomGatewayFilterFactory : [Post] Response Filter
5
```

从日志中,也可以看出来,当GatewayFilter 和GlobalFilter 过滤器order一样时,会先执行GatewayFilter

3. 服务部署

1. 修改数据库, Nacos等相关配置
2. 对三个服务进行打包: product-service, order-service, gateway
3. 上传jar到Linux服务器
4. 启动Nacos

启动前最好把data数据删除掉.

5. 启动服务

```
1 #后台启动order-service, 并设置输出日志到logs/order.log
2 nohup java -jar order-service.jar >logs/order.log &
3
4 #后台启动product-service, 并设置输出日志到logs/order.log
5 nohup java -jar product-service.jar >logs/product-9090.log &
6
7 #启动网关
8 nohup java -jar gateway.jar >logs/gateway.log &
```

观察Nacos控制台

NACOS

首页 文档 博客 社区 Nacos企业版 En

NACOS 2.2.3

配置管理 服务管理 服务列表 订阅者列表 命名空间 集群管理

当前集群没有开启鉴权, 请参考[文档](#)开启鉴权~

服务列表

public

服务名称 请输入服务名称 分组名称 请输入分组名称 隐藏空服务 查询 创建服务

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
gateway	DEFAULT_GROUP	1	1	1	false	详情 示例代码 订阅者 删除
product-service	DEFAULT_GROUP	1	1	1	false	详情 示例代码 订阅者 删除
order-service	DEFAULT_GROUP	1	1	1	false	详情 示例代码 订阅者 删除

每页显示: 10 < 上一页 1 下一页 >

6. 测试

访问接口: <http://110.41.51.65:10030/product/1001>

观察远程调用的结果:

不安全 http://110.41.51.65:10030/product/1001

```
{ "id": 1001, "productName": "T恤", "productPrice": 101, "state": 0, "createTime": "2023-12-29T10:14:24.000+00:00", "updateTime": "2023-12-29T10:14:24.000+00:00" }
```

观察日志:

- 1

2024-01-06T16:09:39.321+08:00

INFO 888001

[or-http-epoll-2]

c.b.g.filter.CustomGatewayFilterFactory : [Pre] Filter Request, name:custom filter
- 2

2024-01-06T16:09:39.322+08:00

INFO 888001

[oundedElastic-1]

c.b.gateway.filter.CustomGlobalFilter : [Pre] CustomGlobalFilter enter...
- 3

2024-01-06T16:09:39.338+08:00

INFO 888001

[or-http-epoll-3]

c.b.gateway.filter.CustomGlobalFilter : [Post] CustomGlobalFilter return...
- 4

2024-01-06T16:09:39.338+08:00

INFO 888001

[or-http-epoll-3]

c.b.g.filter.CustomGatewayFilterFactory : [Post] Response Filter