

第5章Spring Cloud Gateway微服务网关

5.1 网关简介

在微服务架构系统下，网关是系统唯一对外的入口，介于客户端与服务器端之间，用于对请求进行鉴权、限流、路由、监控等功能。**本身也是一个微服务。**

架构图: <https://www.processon.com/diagraming/5d2bb517e4b02015bd7c4d34>

OpenResty

5.1.1 Gateway简介

Spring Cloud Gateway 旨在提供一种简单而有效的方法来路由到 API，并为API接口提供鉴权、限流、路由、监控等功能。Gateway基于Spring 生态开发而成【Spring 5、Spring Boot 2和 project Reactor】。

Spring Cloud Gateway

3.0.0

This project provides an API Gateway built on top of the Spring Ecosystem, including: Spring 5, Spring Boot 2 and Project Reactor. Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.

1. How to Include Spring Cloud Gateway

To include Spring Cloud Gateway in your project, use the starter with a group ID of `org.springframework.cloud` and an artifact ID of `spring-cloud-starter-gateway`. See the [Spring Cloud Project page](#) for details on setting up your build system with the

5.1.2 与Zuul的对比

1. Zuul闭源：Spring Cloud Gateway 是 Zuul 网关的替代者。只所以弃用 Zuul 并不是因为 Zuul 在功能有什么大的问题。而是因为最开始的 Zuul 是开源的，所以 Spring Cloud 就集成了 Zuul 做网关。但后来 Zuul 又宣布闭源，所以 Spring Cloud 自己开发了 Spring Cloud Gateway。再后来 Zuul 2.0又开源了，但 Spring Cloud 不再集成它了。
2. Zuul：Zuul 是由 Netflix 开源的 API 网关，基于 servlet 的，使用阻塞 API，它不支持任何长连接。
3. Spring Cloud Gateway：Spring Cloud Gateway 是 Spring Cloud 自己开发的开源 API 网关，建立在 Spring5, Reactor和 Spring Boot 2 之上，使用非阻塞 API，支持异步开发。

5.1.3 重要概念

在 Spring Cloud Gateway 中有三个非常重要的概念：

1. **route路由**：路由是网关的最基本组成，由一个路由 id、一个目标地址 url，一组断言工厂及一组 filter组成。若断言为 true，则请求将经由 filter 被路由到目标 url。
2. **predicate断言**：断言即一个条件判断，根据当前的 http 请求进行指定规则的匹配，比如说 http 请求头，请求时间等。只有当匹配上规则时，断言才为 true，此时请求才会被直接路由到目标地址（目标服务器），或先路由到某过滤器链，经过过程器链的层层处理后，再路由到相应的目标地址（目标服务器）。

3. **filter过滤器**：对请求进行处理的逻辑部分。当请求的断言为 true 时，会被路由到设置好的过滤器，以对请求进行处理。例如，可以为请求添加一个请求头，或添加一个请求参数，或对请求URI进行修改等。总之，就是对请求进行处理。

5.2 网关入门案例

5.2.1 入门案例【静态路由】

以下两种路由的实现，目前都无需注册到 Nacos，即无需 Nacos 依赖。

需求：用户访问 spring cloud Gateway应用，直接跳转到百度主页。

1、配置路由到服务

目标：搭建网关微服务05-gateway-config，并且配置路由到服务

(1) 创建工程

复制工程 02-consumer-nacos，并重命名为 05-gateway-config。这里需要保证其是一个 Spring Cloud Alibaba 工程。

(2) 修改pom文件

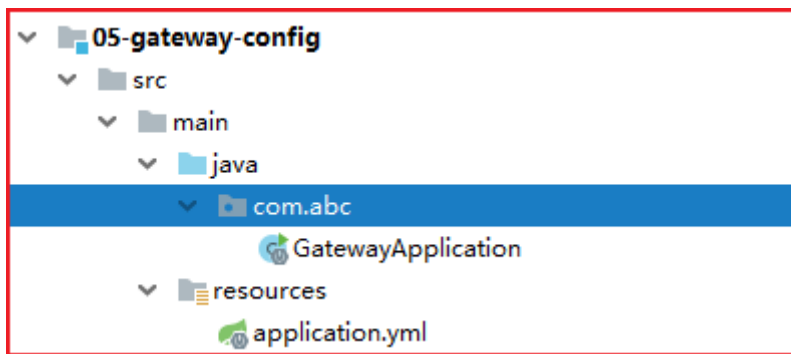
仅保留 Actuator 依赖，并添加 spring cloud gateway 依赖。

```
1 <dependencies>
2   <!--actuator依赖-->
3   <dependency>
4     <groupId>org.springframework.boot</groupId>
5     <artifactId>spring-boot-starter-actuator</artifactId>
6   </dependency>
7   <!--gateway 依赖-->
8   <dependency>
9     <groupId>org.springframework.cloud</groupId>
10    <artifactId>spring-cloud-starter-gateway</artifactId>
11  </dependency>
12 </dependencies>
```

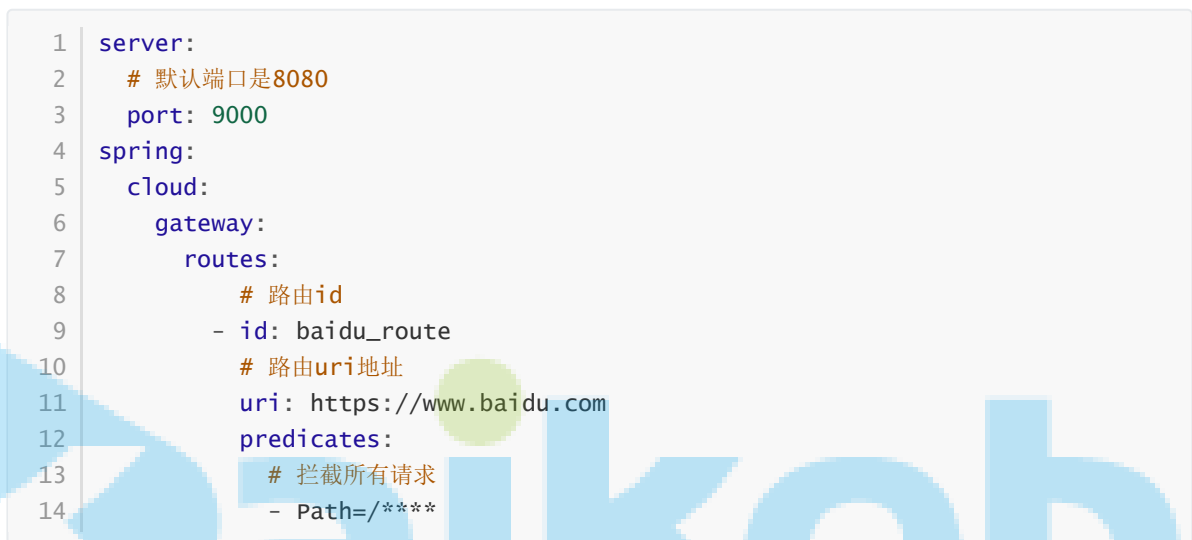
(3) 修改代码类

代码中，只需一个启动类，其它全部删除。

```
1 @SpringBootApplication
2 public class GatewayApplication {
3     public static void main(String[] args) {
4         SpringApplication.run(GatewayApplication.class, args);
5     }
6 }
```



(4) 修改配置文件



(5) 测试



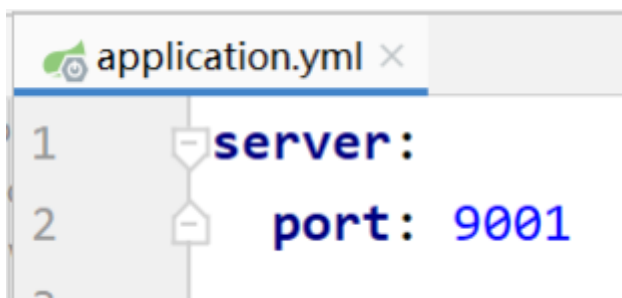
2、配置路由到百度

(1) 创建工程

复制工程 05-gateway-config，并重命名为 05-gateway-api。

(2) 修改配置文件

去掉原来配置的路由策略，仅剩如下内容。



(3) 修改启动类

在启动类中添加一个@Bean 方法，用于设置路由策略。

```
1  @SpringBootApplication//启动引导类，同时也是配置类
2  public class GatewayApiApplication {
3      public static void main(String[] args) {
4          SpringApplication.run(GatewayApiApplication.class, args);
5      }
6      //配置路由规则
7      @Bean
8      public RouteLocator someRouteLocator(RouteLocatorBuilder builder){
9          return builder.routes().route(predicateSpec -> predicateSpec
10              .path("/**")
11              .uri("https://www.baidu.com")
12              .id("baidu_route")).build();
13      }
14  }
```

(4) 测试



5.2.2 入门案例【动态路由\负载均衡】

根据微服务名称进行 Ribbon 负载均衡。有一个微服务，有三个提供者，这里要通过 spring cloud gateway 实现对该微服务的负载均衡访问。

1、配置式路由方式1

目标：配置式路由05-gateway-ribbon-config

(1) 创建工程

复制工程 05-gateway-config，并重命名为 05-gateway-ribbon-config。

(2) 添加依赖

```
1  <!--nacos discovery 依赖-->
2  <dependency>
3      <groupId>com.alibaba.cloud</groupId>
4      <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
5  </dependency>
```

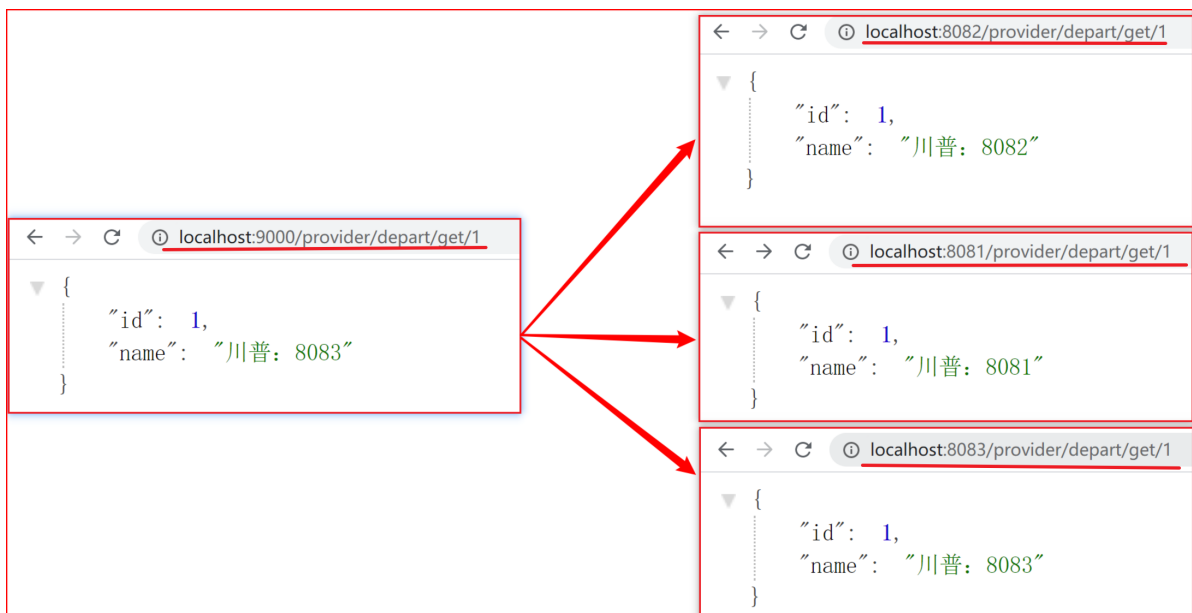
(3) 修改配置文件

```
1  server:
2    # 默认端口是8080
3    port: 9000
4  spring:
5    application:
6      name: msc-gateway-depart
7  cloud:
8    nacos:
9      # 注册中心地址
10     discovery:
11       server-addr: 127.0.0.1:8848
12     gateway:
13       discovery:
14         locator:
15           # 配置开启与DiscoveryClient整合
16           enabled: true
17     routes:
18       # 路由id
19       - id: ribbon_route
20         # 路由uri地址, loadbalancer://服务名称
21         uri: lb://msc-provider-depart
22         predicates:
23           # 拦截所有提供者的请求到提供者集群服务
24           - Path=/provider/depart/**
```

(4) 修改负载均衡策略

```
9  @SpringBootApplication
10  public class GatewayConfigApplication {
11
12  ... public static void main(String[] args) { SpringApplication.run(GatewayConfigApplication.class, args); }
13
14  ... //配置负载均衡策略
15
16  ... @Bean
17  ... public IRule loadBalanceRule(){
18  ...     return new RandomRule();
19  ... }
20
21 }
```

(5) 测试



2、配置式路由方式2

目标：配置式路由09-gateway-ribbon-api

(1) 创建工程

复制工程 05-gateway-api，并重命名为 05-gateway-ribbon-api。

(2) 添加依赖

由于要注册到 Nacos，所以需要导入 Nacos Discovery 依赖。

```
1 <!--nacos discovery 依赖-->
2 <dependency>
3   <groupId>com.alibaba.cloud</groupId>
4   <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
5 </dependency>
```

(3) 修改配置文件

```
1 server:
2   port: 9001
3   spring:
4     application:
5       name: msc-gateway-depart
6     cloud:
7       nacos:
8         discovery:
9           server-addr: 127.0.0.1:8848
```

(4) 修改启动类

在启动类中添加如下@Bean 方法。

```
1 @SpringBootApplication//启动引导类，同时也是配置类
2 public class GatewayApiApplication {
3     public static void main(String[] args) {
4         SpringApplication.run(GatewayApiApplication.class, args);
5     }
6     //配置路由规则
7     @Bean
8     public RouteLocator someRouteLocator(RouteLocatorBuilder builder){
9         //路由构建器对象，构建一个路由规则
10        return builder.routes().route(predicateSpec -> predicateSpec
11            .path("/provider/depart/**")
12            .uri("lb://abcmvc-provider-depart")
13            .id("ribbon_route")).build();
14    }
15 }
```

(4) 测试

```
← → ↻ ⓘ localhost:9001/provider/depart/get/1
▼ {
  "id": 1,
  "name": "川普: 8082"
}
```

扩展：PathMatcher

<https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#the-path-route-predicate-factory>

<https://docs.spring.io/spring-framework/docs/5.2.12.RELEASE/javadoc-api/>

`PathMatcher` 实现了Ant路径匹配风格。借鉴了ApacheAnt项目的大部分路径匹配代码实现。

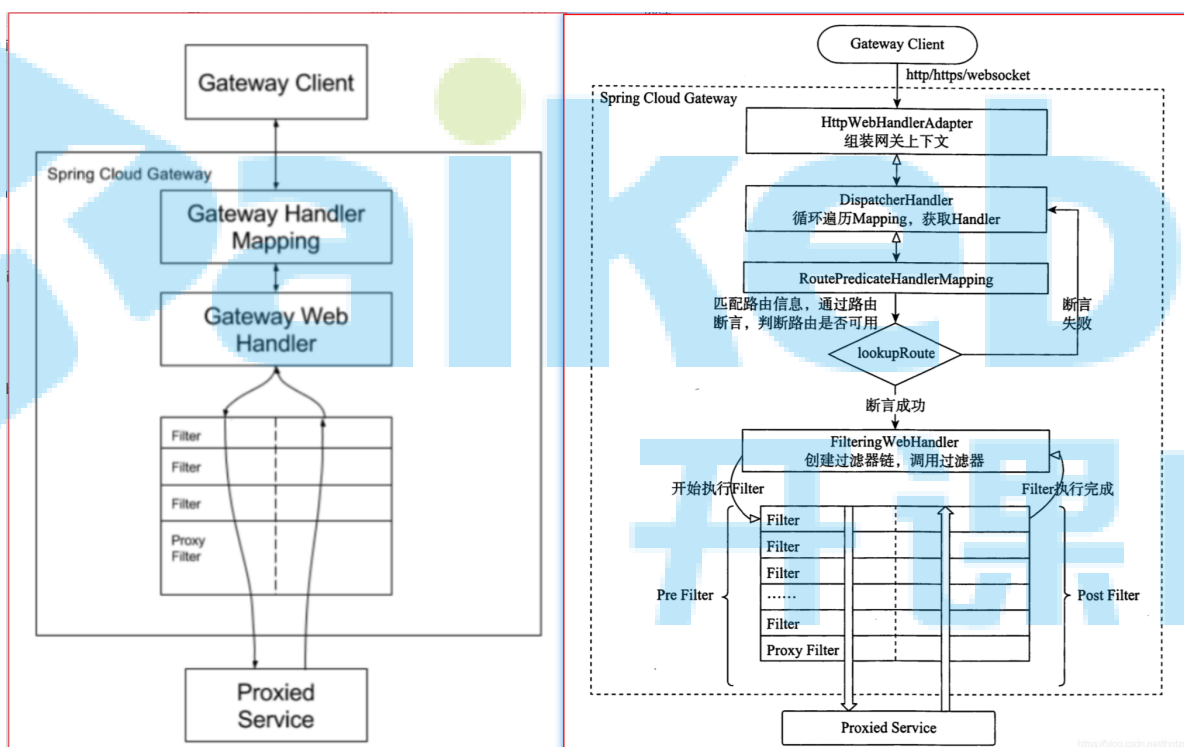
规则如下：

- `?` 只匹配正斜杠 / 路径后一个字符
- `*` 匹配正斜杠 / 路径后的多个字符串
- `**` 匹配正斜杠 / 路径后的多个正斜杠内的多个字符串
- `{spring:[a-z]+}` 匹配正则表达式 `[a-z]+`，并将匹配到的值赋值给spring变量

5.3 Gateway工作原理

<https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/>

Spring Cloud Gateway 的核心处理流程如下图，左边为官方提供High Level 流程图，右边详细流程图。



网关两端，三大件：客户端Client，服务端Proxied Service

- 网关处理器映射器组件HandlerMapping
- 网关web处理器组件Web Handler
- 网关过滤器组件 Filter

主要流程：

1. Gateway的客户端向Spring Cloud Gateway发起请求
2. 首先请求会被`HttpWebHandlerAdapter`进行提取组装成网关的上下文，然后网关的上下文会传递到`DispatcherHandler`。
3. `DispatcherHandler`是所有请求的分发处理器，`DispatcherHandler`主要负责分发请求对应的处理器，比如将请求分发到对应`RoutePredicateHandlerMapping`(路由断言处理器映射器)。
4. `RoutePredicateHandlerMapping`路由断言处理映射器主要用于路由的查找，以及找到路由后返回对应的`FilteringWebHandler`，如果没有找到则丢弃处理。
5. `FilteringWebHandler`主要负责组装Filter链表并调用Filter执行一系列Filter处理

6. 然后把请求转到后端对应的**代理服务【Proxied Service】**处理，处理完毕后，将Response返回到Gateway客户端。

在Filter链中，通过虚线分割Filter的原因是，过滤器可以在转发请求之前处理，或者在代理服务返回结果之后处理。

- PreFilter前置过滤器：所有的Pre类型的Filter执行完毕之后，才会转发请求到被代理的服务处理。
- PostFilter后置过滤器：被代理的服务把所有请求完毕之后，才会执行Post类型的过滤器。

5.4 网关路由断言工厂

什么是Predicate断言工厂？

路由匹配规则

Spring Cloud Gateway 将路由匹配作为最基本的功能。而这个功能是通过路由断言工厂完成的。Spring Cloud Gateway 中包含了许多内置的路由断言工厂。所有这些断言都可以匹配 HTTP 请求的不同属性，并且可以根据逻辑与状态，将多个路由断言工厂复合使用。路由断言工厂的使用方式有两种：yaml配置文件式与 API 配置类式。

以下讲义中的代码：

- yaml配置文件式：在05-gateway-config 工程上直接进行修改，即yaml配置文件。
- API 配置类式：在 05-gateway-api 工程上直接进行修改，即启动类中设置。

5.4.1 Path路由断言工厂

(1) 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 拼接要到要转向的目标 uri 的后面，形成一个统一的 uri。

(2) 配置式-修改配置文件

添加了两个路由策略。

```
1  # 配置Path断言工厂1，路由到provider
2  - id: path_provider_route
3    uri: http://localhost:8081
4    predicates:
5      - Path=/provider/**
6  # 配置Path断言工厂2，路由到consumer
7  - id: path_consumer_route
8    uri: http://localhost:8080
9    predicates:
10     - Path=/consumer/**
```

(3) API式 修改启动类

直接修改路由方法。添加了两个路由策略。


```

1 //Path配置路由规则
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.path("/provider/**")
6             .uri("http://localhost:8081")
7             .id("path_provider_route"))
8         .route(ps -> ps.path("/consumer/**")
9             .uri("http://localhost:8080")
10            .id("path_consumer_route"))
11        .build();
12 }

```

举一反三

5.4.2 After路由断言工厂

(1) 规则

After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对比。若请求时间在参数时间之后，则匹配成功，断言为 true。

(2) 配置式修改配置文件

```

1 # after断言工厂
2 - id: after_route
3   uri: https://www.baidu.com
4   predicates:
5     - After=2022-01-20T17:42:47.789-07:00[Asia/Shanghai]

```

(3) API式-修改启动类

```

1 //After配置路由规则
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     //ISO-8601日历系统中带有时区的日期时间
5     ZonedDateTime dateTime =
6         LocalDateTime.now().plusDays(5).atZone(ZoneId.systemDefault());
7     return builder.routes()
8         .route(ps -> ps.after(dateTime)//设置after路由断言
9             .uri("https://www.baidu.com")
10            .id("after_route"))
11        .build();
12 }

```

5.4.3 扩展了解断言工厂

1、Before路由断言工厂

(1) 规则

Before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，若请求时间在参数时间之前，则匹配成功，断言为 true。

(2) 配置式修改配置文件

```
- id: baidu_route
  uri: https://www.baidu.com
  predicates:
    - Before=2017-01-20T17:42:47.789-07:00[Asia/Shanghai]
```

```
1 # Before断言工厂
2     - id: Before_route
3       uri: https://www.baidu.com
4       predicates:
5         - Before=2022-01-20T17:42:47.789-07:00[Asia/Shanghai]
```

(3) API式 修改启动类

```
@Bean
public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
    ZonedDateTime dateTime = LocalDateTime.now().minusDays(5)
        .atZone(ZoneId.systemDefault());

    return builder.routes()
        .route("baidu_route", r -> r.before(dateTime)
            .uri("https://www.baidu.com"))
        .build();
}
```

```
1 //Before配置路由规则
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     //ISO-8601日历系统中带有时区的日期时间
5     ZonedDateTime dateTime =
6     LocalDateTime.now().minusDays(5).atZone(ZoneId.systemDefault());
7     return builder.routes()
8         .route(ps -> ps.before(dateTime)//设置before路由断言
9             .uri("https://www.baidu.com")
10             .id("before_route"))
11         .build();
12 }
```

2、Between路由断言工厂

(1) 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，若请求时间在这两个参数时间之间，则匹配成功，断言为 true。

(2) 配置式 修改配置文件

```
1 #Between断言工厂
2     - id: Between_route
3       uri: https://www.baidu.com
4       predicates:
5         - Between=2022-01-20T17:42:47.789-07:00[Asia/Shanghai]
```

(3) API式 修改启动类

直接修改路由方法。

```
1 //Between配置路由规则
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     //ISO-8601日历系统中带有时区的日期时间
5     ZonedDateTime minusTime =
6     LocalDateTime.now().minusDays(5).atZone(ZoneId.systemDefault());
7     ZonedDateTime plusTime =
8     LocalDateTime.now().plusDays(3).atZone(ZoneId.systemDefault());
9     return builder.routes()
10        .route(ps -> ps.between(minusTime, plusTime)//设置before路由断言
11           .uri("https://www.baidu.com")
12           .id("between_route"))
13        .build();
14 }
```

3、Cookie路由断言工厂

(1) 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，匹配成功，断言为 true。

(2) 配置式-修改配置文件

```
1 #Cookie断言工厂
2     - id: cookie_route
3       uri: https://www.baidu.com
4       predicates:
5         - Cookie=love,baby
```

(3) API式修改启动类

直接修改路由方法。

```
@Bean
public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
        .route(ps -> ps.cookie("chocolate", "dove")
            .uri("https://www.baidu.com")
            .id("cookie_route"))
        .build();
}
```

```

1 //cookie配置路由规则
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.cookie("chocolate", "dove")
6             .uri("https://www.baidu.com")
7             .id("cookie_route"))
8         .build();
9 }

```

4、Header路由断言工厂

(1) 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，匹配成功，断言为 true。

(2) 配置式-修改配置文件

```

1 #Header断言工厂
2     - id: header_route
3       uri: https://www.baidu.com
4     predicates:
5       - Header=X-Request-Id, \d+

```

(3) API式-修改启动类

直接修改路由方法。

```

1 //header配置路由规则
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.header("X-Request-Id", "\\d+")
6             .uri("https://www.baidu.com")
7             .id("header_route"))
8         .build();
9 }

```

5、Host路由断言工厂

(1) 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。

(2) 修改hosts文件

修改 C:\Windows\System32\drivers\etc 中的 hosts 文件，为 127.0.0.1 这个 ip 指定多个主机名。例如，在该文件中添加如下内容：

(3) 配置式-修改配置文件

```
1 #Host断言工厂
2     - id: host_route
3       uri: https://www.baidu.com
4       predicates:
5         - Host=mylocalhost:9000, myhost:9000
```

(4) API式-修改启动类

直接修改路由方法。

```
1 //host配置路由规则
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.host("mylocal:9001")
6             .uri("https://www.baidu.com")
7             .id("host_route"))
8         .build();
9 }
```

6、Method路由断言工厂

(1) 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，匹配成功，断言为 true。

(2) 配置式-修改配置文件

```
1 #Method断言工厂
2     - id: method_route
3       uri: https://www.baidu.com
4       predicates:
5         - Method=GET,POST
```

(3) API式-修改启动类

直接修改路由方法。

```
1 //method配置路由规则
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.method("POST")
6             .uri("https://www.baidu.com")
7             .id("method_route"))
8         .build();
9 }
```

7、Query路由断言工厂

(1) 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。当请求中包含了指定的参数名，或名值对时，匹配成功，断言为 true。

(2) 配置式 修改配置文件

```
1 #Query断言工厂
2     - id: query_route
3       uri: https://www.baidu.com
4     predicates:
5       - Query=color, gr.+
6       - Query=size, 5
```

(3) API式-修改启动类

直接修改路由方法。

```
1 //query配置路由规则
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.query("color", "gr.+")
6             .and()
7             .query("size", "5")
8             .uri("https://www.baidu.com")
9             .id("query_route"))
10        .build();
11 }
```

8、RemoteAddr路由断言工厂

(1) 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，匹配成功，断言为 true。

(2) 配置式-修改配置文件

```
1 #RemoteAddr断言工厂
2     - id: remoteAddr_route
3       uri: https://www.baidu.com
4     predicates:
5       - RemoteAddr=10.20.33.150/180
```

(3) API式-修改启动类

直接修改路由方法。

```

1 //remoteAddr配置路由规则
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.remoteAddr("10.20.33.150/180")
6             .uri("https://www.baidu.com")
7             .id("remoteAddr_route"))
8         .build();
9 }

```

5.5 网关过滤器Filter

5.5.1 什么是过滤工厂&过滤器?

过滤器允许以某种方式修改传入的 HTTP 请求或返回的 HTTP 响应。而过滤器作用域是某些特定路由。

过滤器分类

过滤器允许以某种方式修改传入的 HTTP 请求或返回的 HTTP 响应。Filter 根据其作用范围的不同，分为两种：局部过滤器与全局过滤器

- 局部过滤器：应用于单个路由策略上，其功能仅对路由断言为 true 的请求起作用
- 全局过滤器：应用于所有路由策略上，其功能就像前面的配置文件中设置的默认 Filter

Spring Cloud Gateway 包括许多内置的 GatewayFilter 工厂。GatewayFilter 工厂的使用方式有两种：配置式与 API 式。

5.5.2 环境准备三步走

以下讲义中的代码：

- yaml配置文件式：在 05-gateway-config-filter 工程上直接进行修改，即yaml配置文件。
- API 配置类式：在05-gateway-api-filter 工程上直接进行修改，即启动类中设置。

(1) 第一步：定义一个工程05-showinfo

该工程就是为了显示通过过滤工厂对请求处理后的结果的。

A、定义工程

就定义一个普通的 Spring Boot 工程，仅导入一个 Spring Web 依赖，即spring-boot-web-starter 依赖。

B、定义处理器

```

1 @RestController
2 @RequestMapping("/info")
3 public class SomeController {
4
5     // 暂时还没有定义任务处理器方法
6
7 }

```

(2) 第二步：创建config工程05-gateway-config-filter

直接复制05-gateway-config工程，并将配置文件中原来设置的路由策略全部删除。后续对于 filter 的配置式设置，都在该工程中进行。

(3) 第三步：创建API工程05-gateway-api-filter

直接复制 05-gateway-api 工程，并将启动类中原来设置的路由策略全部删除。后续对于 filter 的 API 式设置，都在该工程中进行。

5.5.3 PrefixPath 过滤器：添加前缀

(1) 规则

该过滤器工厂会为指定的 URI 自动添加上一个指定的 URI 前缀。

(2) 配置式

```
1 # 添加前缀 过滤器
2   - id: prefixPath_filter
3     uri: http://localhost:8080
4     predicates:
5       - Path=/**
6     filters:
7       - PrefixPath=/consumer
```

(3) API式

直接修改路由方法。

```
1 //prefixPath过滤工厂
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.path("/**")
6             .filters(fs -> fs.prefixPath("/consumer"))
7             .uri("http://localhost:8080")
8             .id("prefixPath_filter"))
9         .build();
10 }
```

5.5.4 StripPrefix 过滤器：去除前缀

(1) 规则

该过滤器工厂会为指定的 URI 去掉指定长度的前缀。

(2) 配置式

```
1 # 去除前缀 过滤器
2   - id: stripPrefix_filter
3     uri: http://localhost:8080
4     predicates:
5       - Path=/**
6     filters:
7       - StripPrefix=2
```


(3) API式

直接修改路由方法。

```
1 //stripPrefix过滤工厂
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.path("/**")
6             .filters(fs -> fs.stripPrefix(2))
7             .uri("http://localhost:8080")
8             .id("stripPrefix_filter"))
9         .build();
10 }
```

5.5.5 扩展了解过滤器工厂

1、AddRequestParameter 过滤器

(1) 规则

该过滤器工厂会对匹配上的请求添加指定的请求参数。

(2) 配置式

```
1 # AddRequestParameter 过滤器
2 - id: addRequestParameter_filter
3   uri: http://localhost:8080
4   predicates:
5     - Path=/**
6   filters:
7     - AddRequestParameter=color, blue
```

(3) 配置式-修改showinfo工程处理器

在处理器中添加如下处理器方法

```
@RequestMapping("/param")
public String paramHandler(String color) {
    return "color: " + color;
}
```

(4) API式

直接修改路由方法。

```

1 //addRequestParameter过滤器工厂
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.path("/**")
6             .filters(fs -> fs.addRequestParameter("color", "blue"))
7             .uri("http://localhost:8080")
8             .id("addRequestParameter_filter"))
9         .build();
10 }

```

2、AddResponseHeader 过滤器

(1) 规则

该过滤器工厂会给从网关返回的响应添加上指定的 header。

(2) 配置式

```

1 # AddResponseHeader 过滤器
2 - id: addResponseHeader_filter
3   uri: http://localhost:8080
4   predicates:
5     - Path=/**
6   filters:
7     - AddResponseHeader=X-Response-Red, Blue

```

(3) API式

直接修改路由方法。

```

1 //addResponseHeader过滤器工厂
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.path("/**")
6             .filters(fs -> fs.addResponseHeader("X-Response-Red",
7 "blue"))
8             .uri("http://localhost:8080")
9             .id("addResponseHeader_filter"))
10        .build();

```

3、AddRequestHeader 过滤器

(1) 规则

该过滤器工厂会对匹配上的请求添加指定的 header。

(2) 配置式

这里修改的是 09-gateway-filter-config 工程中的配置文件。

```

1 # AddRequestHeader 过滤器
2     - id: addRequestHeader_filter
3       uri: http://localhost:8080
4       predicates:
5         - Path=/**
6       filters:
7         - AddRequestHeader=X-Request-red, blue

```

(3) 配置式-修改showinfo工程处理器

在处理器中添加如下处理器方法。

```

1 @RestController
2 @RequestMapping("info")
3 public class SomeController {
4     //暂时还没有定义任务处理器方法
5     @RequestMapping("header")
6     public String headerHandler(HttpServletRequest request){
7         String header = request.getHeader("X-Request-red");
8         return "X-Request-red: " + header;
9     }
10 }

```

(4) API式

直接修改路由方法。

```

1 //addRequestHeader过滤工厂
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.path("/**")
6             .filters(fs -> fs.addRequestHeader("X-Request-red", "blue"))
7             .uri("http://localhost:8080")
8             .id("addRequestHeader_filter"))
9         .build();
10 }

```

4、RewritePath 过滤器

(1) 规则

该过滤器工厂会将请求 URI 替换为另一个指定的 URI 进行访问。RewritePath 有两个参数，第一个是正则表达式，第二个是要替换为目标表达式。

(2) 配置式

```

1 # 重写路径 过滤器
2     - id: rewritePath_filter
3       uri: http://localhost:8080
4       predicates:
5         - Path=/**
6       filters:
7         - RewritePath=/red(<segment>/?.*), ${segment}

```

(3) API式

直接修改路由方法。

```
1 //rewritePath过滤工厂
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     return builder.routes()
5         .route(ps -> ps.path("/**")
6             .filters(fs -> fs.rewritePath("/red(<segment>/?.*)",
7                 "${segment}"))
8             .uri("http://localhost:8080")
9             .id("rewritePath_filter"))
10     .build();
11 }
```

5.5.6 自定义过滤器

前面的 GatewayFilter 工厂可以创建出某种特定的 Filter 过滤效果，但这些过滤功能可能并不能满足全部业务需求，此时可以根据具体需求自定义自己的 Filter。

1、自定义GatewayFilter：修改请求

目标：

这里要实现的需求是，在自定义的 Filter 中为请求添加指定的请求头。

实现步骤：

1. 创建自定义过滤器工程05-gateway-filter-custom
2. 自定义AddHeaderGatewayFilter，实现GatewayFilter接口
3. 配置自定义过滤器AddHeaderGatewayFilter
4. 启动测试

实现过程：

(1) 创建工程05-gateway-filter-custom

复制 05-gateway-api 工程，并重命名为 05-gateway-filter-custom。在此工程基础上进行修改。当然，首先要修改端口号为 9000。

(2) 定义GatewayFilter

```
1 /**
2  * 目标：实现需求，在自定义的 Filter 中为请求添加指定的请求头。
3  */
4 public class AddHeaderGatewayFilter implements GatewayFilter {
5     @Override
6     public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
7         chain) {
8         //1. 构建改变后的Request对象
9         ServerHttpRequest request =
10             exchange.getRequest().mutate().header("X-Request-red", "blue").build();
11         //2. 将改变后的对象设置到exchange对象中
12         ServerWebExchange webExchange =
13             exchange.mutate().request(request).build();
14     }
```

```

11 //3.设置修改后的exchange
12 return chain.filter(webExchange);
13 }
14 }

```

(3) 修改启动类

仅修改路由方法。

```

1 //配置路由：自定义拦截器
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     //路由构建器对象，构建一个路由规则
5     return builder.routes().route(predicateSpec -> predicateSpec
6         .path("/**")
7         .filters(gfs -> gfs.filter(new
8             AddHeaderGatewayFilter()))
9         .uri("http://localhost:8080")
10        .id("AddHeader_route")).build();
11 }

```

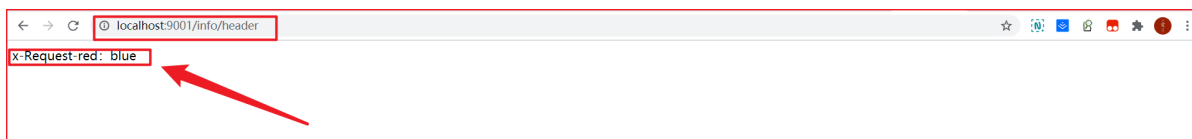
(4) 修改ShowInfo工程的处理器

```

1 @RestController
2 @RequestMapping("info")
3 public class SomeController {
4     //暂时还没有定义任务处理器方法
5     @RequestMapping("header")
6     public String headerHandler(HttpServletRequest request){
7         String header = request.getHeader("x-request-red");
8         return "x-Request-red: " + header;
9     }
10 }

```

(5) 测试



2、自定义GatewayFilter：多filter

目标：

下面我们要定义出多个 Filter，每个 Filter 都具有 pre 与 post 两部分。将所有 Filter 注册到路由中，以查看它们执行的顺序。

实现步骤：

1. 在工程05-gateway-filter-custom中定义三个GatewayFilter
2. 修改启动类，配置三个过滤器
3. 在ShowInfo工程的添加处理器
4. 启动服务测试

实现过程:

(1) 在工程05-gateway-filter-custom中定义三个GatewayFilter

A、第一个Filter

```
1  /**
2   * 目标: 查看过滤器执行顺序, 观察pre过滤和post过滤
3   */
4  //过滤器1
5  public class OneGatewayFilter implements GatewayFilter {
6      @Override
7      public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
8          //获取系统当前时间
9          long startTime = System.currentTimeMillis();
10         System.out.println("pre-filter-【111】 " + startTime);
11         //设置filter过滤器时间
12         exchange.getAttributes().put("startTime", startTime);
13         return chain.filter(exchange).then(Mono.fromRunnable(() -> {
14             System.out.println("post-filter-【111】 ");
15             //获取过滤器执行开始时间
16             Long startTimeAttr = (Long)
exchange.getAttributes().get("startTime");
17             //获取过滤器执行结束时间
18             long endTime = System.currentTimeMillis();
19             //计算开始到结束时间差值
20             System.out.println("OneGatewayFilter过滤器执行用时: " + (endTime -
startTime));
21         }));
22     }
23 }
24
```

B、第二个Filter

```
1  //过滤器2
2  public class TwoGatewayFilter implements GatewayFilter {
3      @Override
4      public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
5          System.out.println("pre-filter-【222】 ");
6          return chain.filter(exchange).then(Mono.fromRunnable(() -> {
7              System.out.println("post-filter-【222】 ");
8          }));
9      }
10 }
```

C、第三个Filter

```

1 //过滤器3
2 public class ThreeGatewayFilter implements GatewayFilter {
3     @Override
4     public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
5         System.out.println("pre-filter- 【333】 ");
6         return chain.filter(exchange).then(Mono.fromRunnable(() -> {
7             System.out.println("post-filter- 【333】 ");
8         }));
9     }
10 }

```

(2) 修改启动类，配置三个过滤器

仅修改路由方法。

```

1 //配置三个拦截器
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     //路由构建器对象，构建一个路由规则
5     return builder.routes()
6         .route(ps -> ps.path("/**")
7             .filters(gfs -> gfs
8                 .filter(new OneGatewayFilter())
9                 .filter(new TwoGatewayFilter())
10                .filter(new ThreeGatewayFilter()))
11            .uri("http://localhost:8080")
12            .id("custom_filter_route"))
13         .build();
14 }

```

(3) 在ShowInfo工程的添加处理器

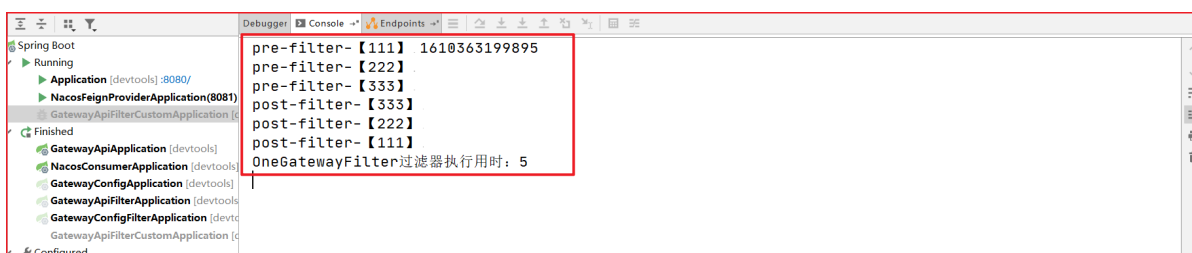
在 05-showinfo 工程中添加如下处理器。

```

1 //获取当前系统时间
2 @RequestMapping("time")
3 public String time(HttpServletRequest request){
4     return "time: " + System.currentTimeMillis();
5 }

```

(4) 测试结果



3、自定义Global Filter：模拟网关鉴权

目标：

这里要实现的需求是，访问当前系统的任意模块的 URL 都需要是合法的 URL。这里所谓合法的 URL 指的是请求中携带了 token 请求参数。

分析：

由于是对所有请求的 URL 都要进行验证，所以这里就需要定义一个 Global Filter，可以应用到所有路由中。

注意：Global Filter 不需要在任何具体的路由规则中进行注册，只需在类上添加@Component注解，将其生命周期交给 Spring 容器来管理即可。

实现步骤：

1. 在工程05-gateway-filter-custom中定义GlobalFilter
2. 配置全局过滤器：默认全局过滤器会生效
3. 测试

实现过程：

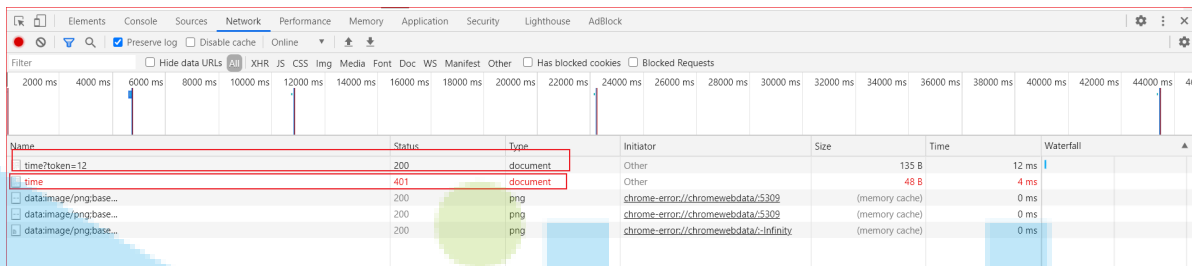
(1) 在工程05-gateway-filter-custom中定义GlobalFilter

```
1  /**
2   * 目标：模拟网关鉴权
3   */
4  @Component//注意：必须注入Spring容器，否则不能生效
5  public class URLValidateFilter implements GlobalFilter, Ordered {
6      @Override
7      public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
8          //1. 获取请求参数token
9          String token =
exchange.getRequest().getQueryParams().getFirst("token");
10         //2. 判断token是否存在
11         //如果不存在则拦截，提示用户未授权
12         if (StringUtils.isEmpty(token)){
13             //设置提示用户未授权
14             exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
15             //完结请求
16             return exchange.getResponse().setComplete();
17         }
18         //如果存在，则放行拦截器
19         return chain.filter(exchange);
20     }
21
22     @Override
23     public int getOrder() {
24         return Ordered.HIGHEST_PRECEDENCE;//最高优先级
25     }
26 }
```


(3) 修改启动类

```
1 //配置全局权限校验过滤器
2 @Bean
3 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
4     //路由构建器对象，构建一个路由规则
5     return builder.routes()
6         .route(ps -> ps.path("/**")
7             .uri("http://localhost:8080")
8             .id("custom_global_filter_route"))
9         .build();
10 }
```

(4) 测试



Name	Status	Type	Initiator	Size	Time	Waterfall
time?token=12	200	document	Other	135 B	12 ms	
time	401	document	Other	48 B	4 ms	
data:image/png;base64...	200	png	chrome-error://chromewebdata/5309	(memory cache)	0 ms	
data:image/png;base64...	200	png	chrome-error://chromewebdata/5309	(memory cache)	0 ms	
data:image/png;base64...	200	png	chrome-error://chromewebdata/5309	(memory cache)	0 ms	

包含token可以访问成功，不包含则失败

4、过滤器的默认优先级：

- 局部 filter 的优先级要高于默认 Filter 的。
- 相同路由策略，配置式的要高于 API 式的。

5、如何修改自定义过滤器执行顺序？

在自定义了一些 GlobalFilter 后，为了保证这些Filter 的执行顺序，在每个返回 GlobalFilter 的@Bean 方法上添加@Order，或直接使自定义的 GlobalFilter 类实现 Order 接口，就可以修改Filter的执行顺序。

```
@Bean
@Order(-1)
public GlobalFilter myGlobalFilter() {
    return (exchange, chain) -> {
        Log.info("first pre filter");
        return chain.filter(exchange).then(Mono.fromRunnable(() -> {
            Log.info("third post filter");
        }));
    };
}
```

```
@Component
public class URLValidateFilter implements GlobalFilter, Ordered {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilter
```

5.6 网关熔断&降级

目标：网关的熔断与服务降级

实现需求：我们下面的例子实现的需求是，浏览器访问 05-showinfo 的处理器不成功，从而服务降级到 Gateway 工程中定义的降级处理器。

实现步骤：

1. 添加依赖坐标
2. 定义降级处理器
3. 编写降级配置
 - 方式1：yml配置文件
 - 方式2：配置类

实现过程：

(1) 规则

该过滤器工厂完成网关层的服务熔断与降级。

(2) 添加依赖

05-gateway-config-filter 与 05-gateway-api-filter 两个工程中均需要添加上resilience4j 依赖。

```
1 <!--resilience4j 依赖-->
2 <dependency>
3   <groupId>org.springframework.cloud</groupId>
4   <artifactId>spring-cloud-starter-circuitbreaker-reactor-
    resilience4j</artifactId>
5 </dependency>
```

(3) 定义降级处理器

05-gateway-config-filter 与 05-gateway-api-filter 两个工程中均需要添加上该降级处理器。

```
1 @RestController
2 public class GatewayFallbackController {
3     //定义服务降级处理方法
4     @RequestMapping("fallback")
5     public String fallback(){
6         return "This is the gateway fallback~";
7     }
8 }
```

(4) 配置式

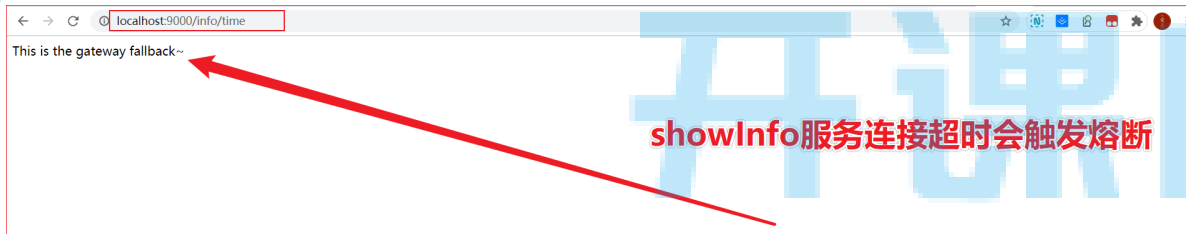
```
1 - id: circuitBreaker_filter
2   uri: http://localhost:8080
3   predicates:
4     - Path=/**
5   filters:
6     - name: CircuitBreaker
7       args:
8         name: myCircuitBreaker
9         fallbackUri: forward:/fallback
```

(5) API式

直接修改路由方法。

```
1 @Bean
2 public RouteLocator someRouteLocator(RouteLocatorBuilder builder) {
3     return builder.routes()
4         .route(ps -> ps.path("/**")
5             .filters(fs -> fs.circuitBreaker(config -> {
6                 config.setName("myCircuitBreaker");
7                 config.setFallbackUri("forward:/fallback");
8             })))
9         .uri("http://localhost:8080")
10        .id("circuitBreaker_filter")
11    .build();
12 }
```

(6) 测试熔断降级



(8) 配置全局熔断降级

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 工厂定义在默认 Filters 中。修改 gateway 工程配置文件。

```
1  spring:
2    cloud:
3      gateway:
4        default-filters:
5          - name: CircuitBreaker
6            args:
7              name: myCircuitBreaker
8              fallbackUri: forward:/fallback
9        routes:
10         - id: test_filter
11           uri: http://localhost:8080
12           predicates:
13             - Path=/**
```

5.7 网关限流

目标：配置网关限流

配置网关限流

实现步骤：

1. 启动redis服务
2. 导入依赖坐标
3. 修改配置
 - 方式1配置类
 - 方式2配置文件

实现过程：

(1) 规则

该过滤器工厂会对进来的请求进行限流。这里采用的是令牌桶算法。另外，从这里也可以看出其是基于 Redis 实现的，所以需要导入 Redis 的依赖。

(2) 导入依赖

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

(3) 配置式-修改启动类

```
1  @SpringBootApplication
2  public class GatewayConfigApplication {
3
4      public static void main(String[] args) {
5          SpringApplication.run(GatewayConfigApplication.class, args);
6      }
7
8      //配置令牌桶算法的key:将主机名称作为限流key
```

```

9      @Bean
10     public KeyResolver keyResolver(){
11         return exchange -> Mono.just(exchange
12             .getRequest()
13             .getRemoteAddress()
14             .getHostName());
15     }
16 }

```

(4) 配置式-修改配置文件

```

1  - id: requestRateLimiter_filter
2  uri: http://localhost:8080
3  predicates:
4  - Path=/**
5  filters:
6  - name: RequestRateLimiter
7    args:
8      key-resolver: "#{@keyResolver}"
9      redis-rate-limiter.replenishRate: 2
10     redis-rate-limiter.burstCapacity: 5

```

解释:

- 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。

通过在 replenishRate 和中设置相同的值来实现稳定的速率 burstCapacity。设置 burstCapacity 高于时, 可以允许临时突发 replenishRate。在这种情况下, 需要在突发之间允许速率限制器一段时间 (根据 replenishRate), 因为2次连续突发将导致请求被丢弃 (HTTP 429 - Too Many Requests)

key-resolver: "#{@userKeyResolver}" 用于通过SPEL表达式来指定使用哪一个KeyResolver.

如上配置:

表示一秒内, 允许一个请求通过, 令牌桶的填充速率也是一秒钟添加一个令牌。

最大突发状况 也只允许一秒内有一次请求, 可以根据业务来调整。

(4) 测试

- 启动本地redis
- 启动gateway网关
- 打开浏览器 <http://localhost:9101/goods/brand>
- 快速刷新, 当1秒内发送多次请求, 就会返回429错误。

(5) 测试

注意: 需要启动Redis的服务

