主讲老师：Fox

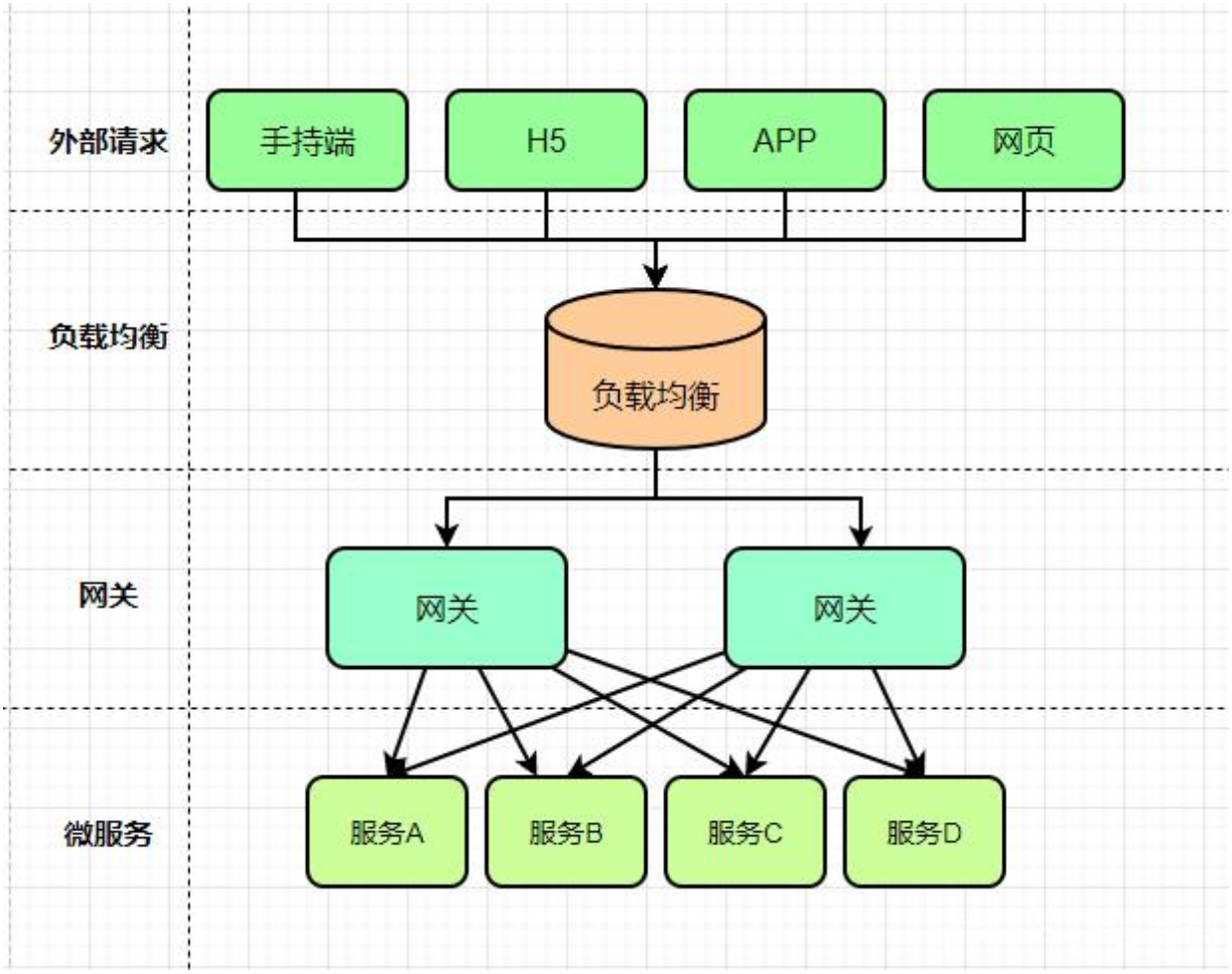
> 有道笔记地址：https://note.youdao.com/s/EaVzI6MH

1. 需求背景

在微服务架构中，通常一个系统会被拆分为多个微服务，面对这么多微服务客户端应该如何去调用呢？如果根据每个微服务的地址发起调用，存在如下问题：

- 客户端多次请求不同的微服务，会增加客户端代码和配置的复杂性，维护成本比价高
- 认证复杂，每个微服务可能存在不同的认证方式，客户端去调用，要去适配不同的认证
- 存在跨域的请求，调用链有一定的相对复杂性（防火墙 / 浏览器不友好的协议）
- 难以重构，随着项目的迭代，可能需要重新划分微服务

为了解决上面的问题，微服务引入了 API网关 的概念，**API网关为微服务架构的系统提供简单、有效且统一的API路由管理，作为系统的统一入口**，提供内部服务的路由中转，给客户端提供统一的服务，可以实现一些和业务没有耦合的公用逻辑，主要功能包含认证、鉴权、路由转发、安全策略、防刷、流量控制、监控日志等。



2. 什么是Spring Cloud Gateway

Spring Cloud Gateway 是Spring Cloud官方推出的第二代网关框架，定位于取代 Netflix Zuul。Spring Cloud Gateway 旨在为微服务架构提供一种简单且有效的 API 路由的管理方式，并基于 Filter 的方式提供网关的基本功能，例如说安全认证、监控、限流等等。

Spring Cloud Gateway 是由 WebFlux + Netty + Reactor 实现的响应式的 API 网关。它不能在传统的 servlet 容器中工作，也不能构建成 war 包。

Spring Cloud Gateway Benchmark

TL;DR

Proxy	Avg Latency	Avg Req/Sec/Thread
gateway	6.61ms	3.24k
linkered	7.62ms	2.82k
zuul	12.56ms	2.09k
none	2.09ms	11.77k

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

1.2 核心概念

- 路由（route）

路由是网关中最基础的部分，路由信息包括一个ID、一个目的URI、一组断言工厂、一组Filter组成。

- 断言(predicates)

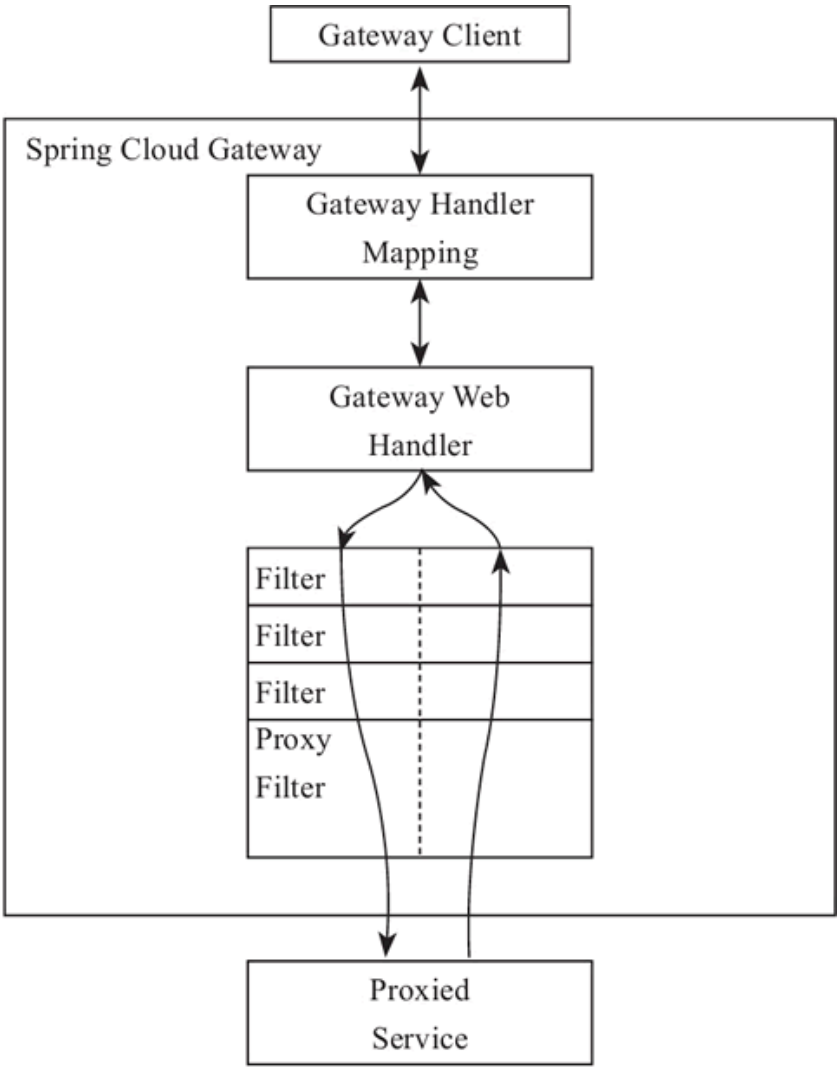
Java8中的断言函数，SpringCloud Gateway中的断言函数类型是Spring5.0框架中的ServerWebExchange。断言函数允许开发者去定义匹配Http request中的任何信息，比如请求头和参数等。如果断言为真，则说明请求的URL和配置的路由匹配。

- 过滤器（Filter）

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

1.2 工作原理

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



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

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

3. Spring Cloud Gateway实战

3.1 微服务快速接入Spring Cloud Gateway

1) 引入依赖

```
<!-- gateway网关 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>

```

```xml
<!-- nacos服务注册与发现 -->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-loadbalancer</artifactId>
</dependency>
```

注意：gateway会和spring-webmvc的依赖冲突，需要排除spring-webmvc

2) 编写yml配置文件

```yaml
spring:
  application:
    name: mall-gateway
  #配置nacos注册中心地址
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848

    gateway:
      #设置路由：路由id、路由到微服务的uri、断言
      routes:
        - id: order_route  #路由ID，全局唯一，建议配置服务名
          uri: lb://mall-order  #lb 整合负载均衡器loadbalancer
          predicates:
            - Path=/order/**   # 断言，路径相匹配的进行路由

        - id: user_route   #路由ID，全局唯一，建议配置服务名
          uri: lb://mall-user  #lb 整合负载均衡器loadbalancer
          predicates:
            - Path=/user/**   # 断言，路径相匹配的进行路由
```

3）测试

http://localhost:8888/order/findOrderByUserId/1

3.2 路由断言工厂（Route Predicate Factories）配置

predicates：路由断言，判断请求是否符合要求，符合则转发到路由目的地。application.yml配置文件中写的断言规则只是字符串，这些字符串会被Predicate Factory读取并处理，转变为路由判断的条件

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

通过网关启动日志，可以查看内置路由断言工厂：

```
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [After]
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [Before]
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [Between]
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [Cookie]
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [Header]
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [Host]
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [Method]
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [Path]
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [Query]
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [ReadBodyPredicateFactory]
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [RemoteAddr]
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [Weight]
.RouteDefinitionRouteLocator    : Loaded RoutePredicateFactory [CloudFoundryRouteService]
```

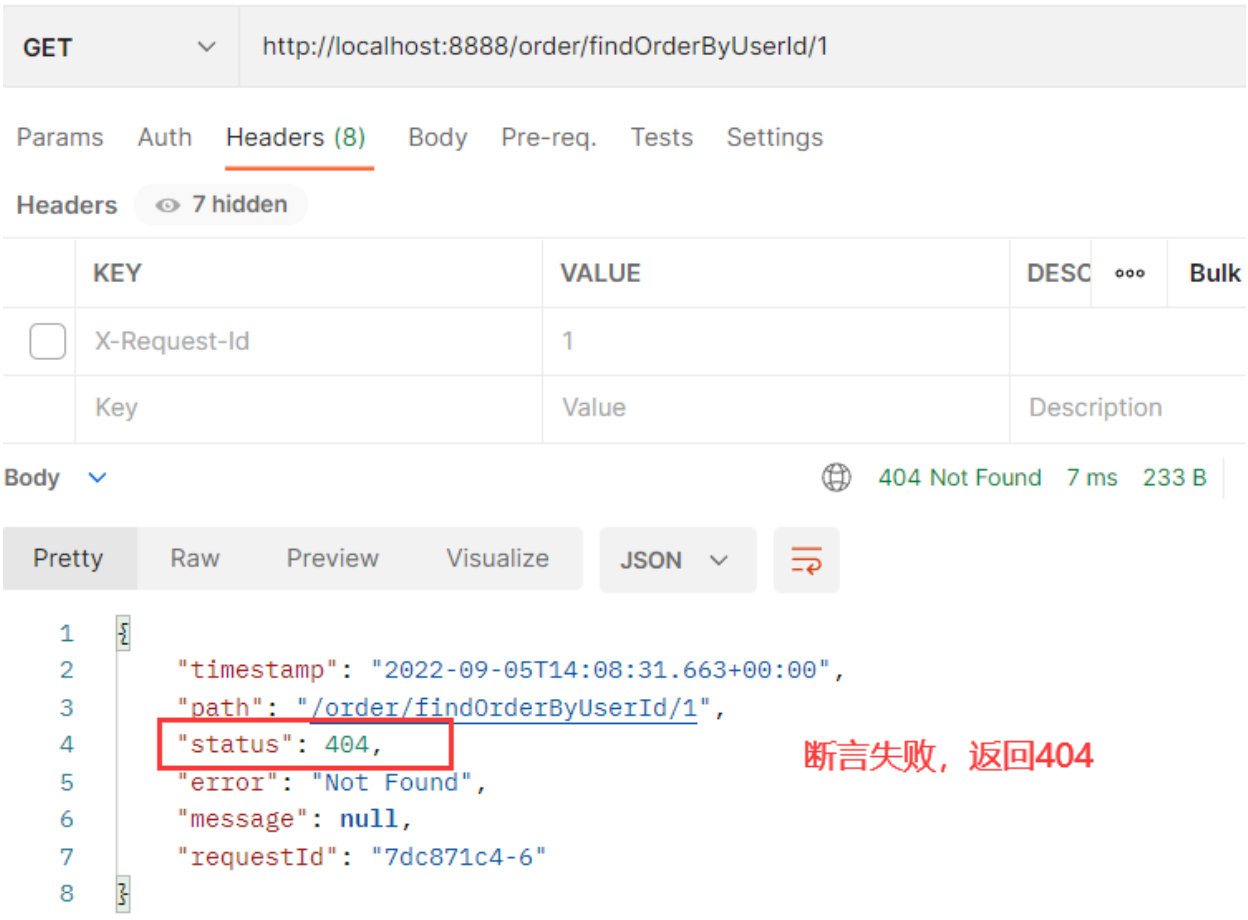
3.2.1 路径匹配

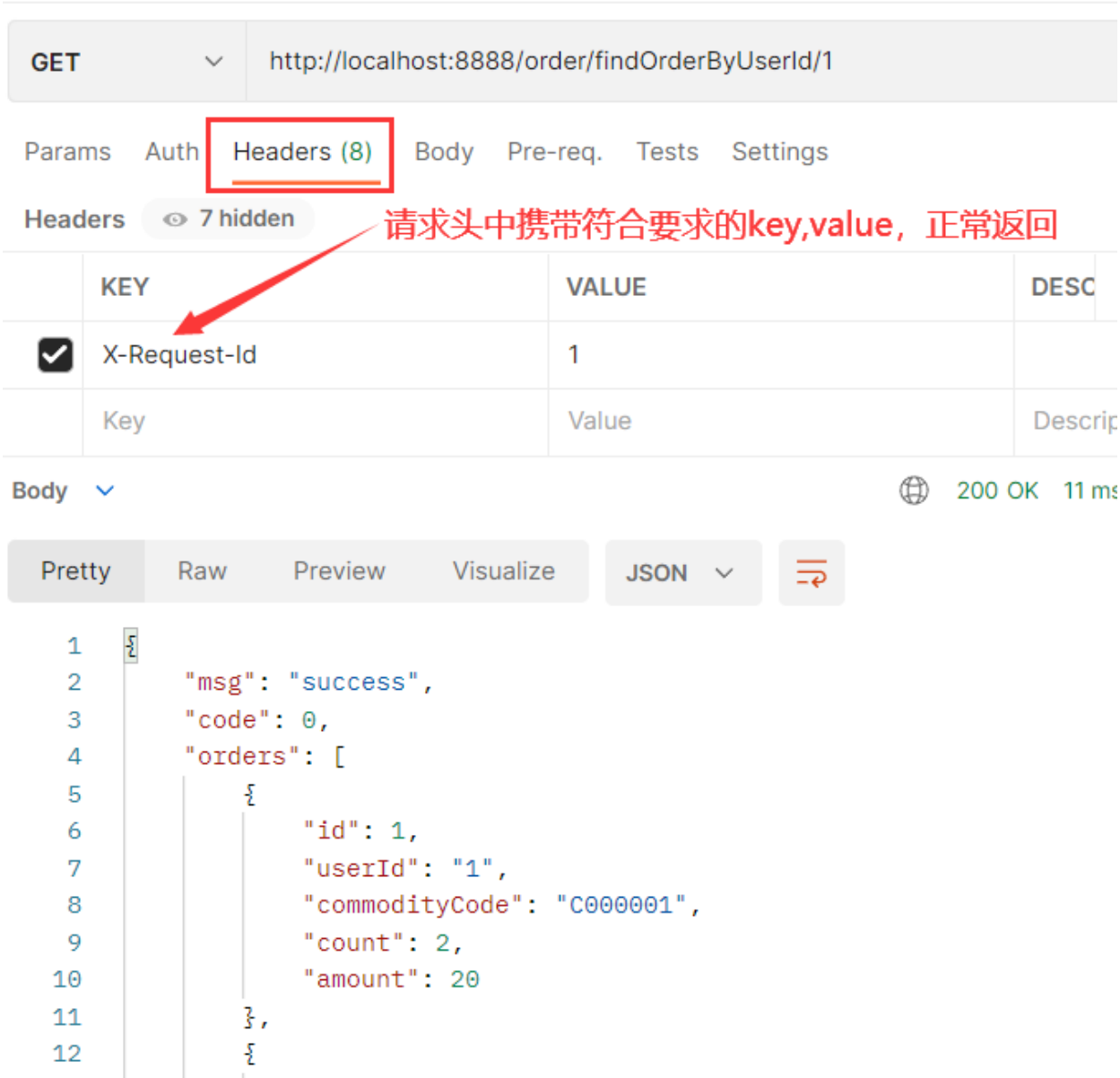
```yaml
spring:
  cloud:
    gateway:
      #设置路由：路由id、路由到微服务的uri、断言
      routes:
      - id: order_route  #路由ID，全局唯一
        uri: lb://mall-order  #目标微服务的请求地址和端口
        predicates:
         # 测试：http://localhost:8888/order/findOrderByUserId/1
        - Path=/order/**     # 断言，路径相匹配的进行路由
```

3.2.2 Header匹配

```yaml
spring:
  cloud:
    gateway:
      #设置路由：路由id、路由到微服务的uri、断言
      routes:
      - id: order_route  #路由ID，全局唯一
        uri: lb://mall-order  #目标微服务的请求地址和端口
        predicates:
          - Path=/order/**   # 断言，路径相匹配的进行路由
         # Header匹配  请求中带有请求头名为 x-request-id，其值与 \d+ 正则表达式匹配
          - Header=X-Request-Id, \d+
```

测试

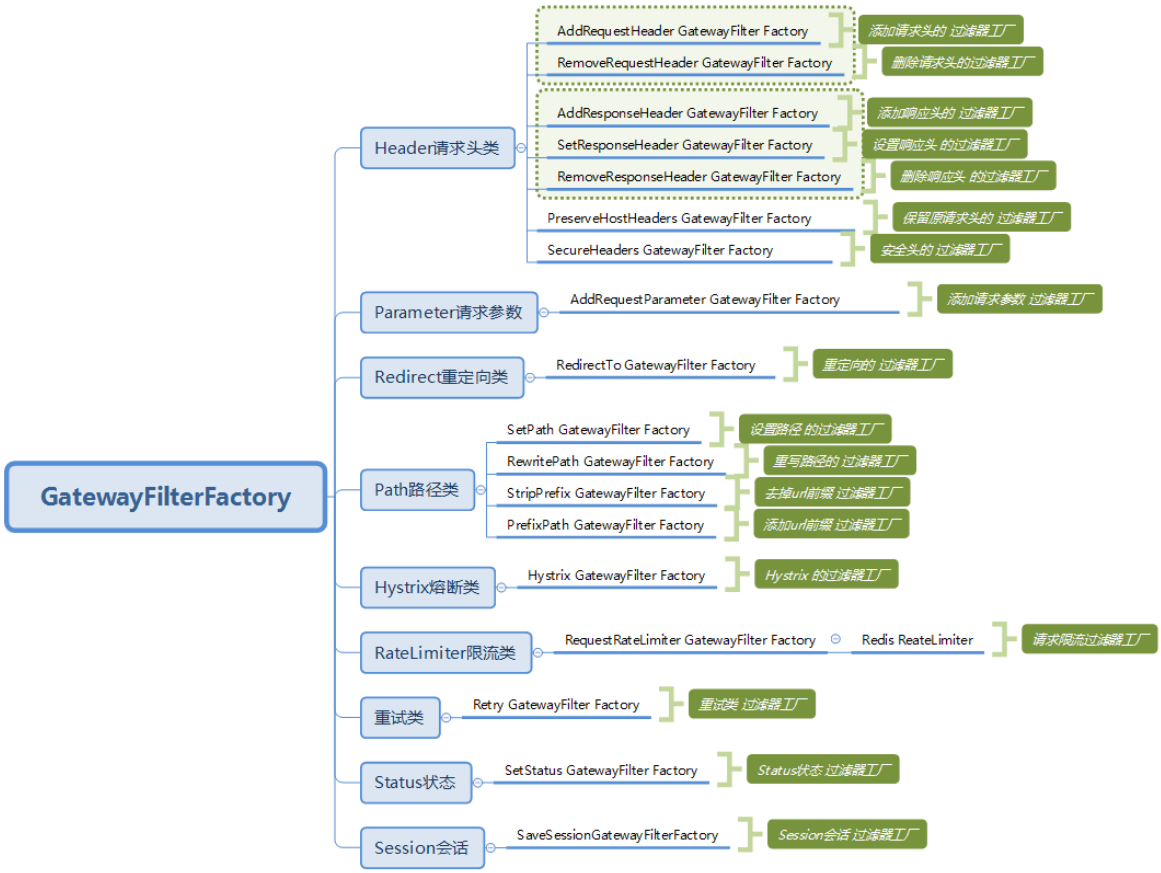




3.3 过滤器工厂（GatewayFilter Factories）配置

GatewayFilter是网关中提供的一种过滤器，可以对进入网关的请求和微服务返回的响应做处理

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



3.3.1 添加请求头

需求：给所有进入mall-order的请求添加一个请求头：X-Request-color=red。

只需要修改gateway服务的application.yml文件，添加路由过滤即可：

```yaml
spring:
  cloud:
    gateway:
      #设置路由：路由id、路由到微服务的uri、断言
      routes:
      - id: order_route  #路由ID，全局唯一
        uri: http://localhost:8020  #目标微服务的请求地址和端口
        #配置过滤器工厂
        filters:
        - AddRequestHeader=X-Request-color, red  #添加请求头
```

测试http://localhost:8888/order/testgateway

```java
@GetMapping("/testgateway")
public String testGateway(HttpServletRequest request) throws Exception {
    log.info("gateWay获取请求头X-Request-color: "
            +request.getHeader("X-Request-color"));
    return "success";
}
@GetMapping("/testgateway2")
public String testGateway(@RequestHeader("X-Request-color") String color) throws Exception {
    log.info("gateWay获取请求头X-Request-color: "+color);
    return "success";
}
```

3.3.2 添加请求参数

```yaml
spring:
  cloud:
    gateway:
      #设置路由：路由id、路由到微服务的uri、断言
      routes:
      - id: order_route  #路由ID，全局唯一
        uri: http://localhost:8020  #目标微服务的请求地址和端口
        #配置过滤器工厂
        filters:
        - AddRequestParameter=color, blue  # 添加请求参数
```

测试http://localhost:8888/order/testgateway3

```java
@GetMapping("/testgateway3")
public String testGateway3(@RequestParam("color") String color) throws Exception {
    log.info("gateWay获取请求参数color:"+color);
    return "success";
}
```

3.3.3 自定义过滤器工厂

继承AbstractNameValueGatewayFilterFactory且我们的自定义名称必须要以GatewayFilterFactory结尾并交给spring管理。

```
@Component
@Slf4j
public class CheckAuthGatewayFilterFactory extends
AbstractNameValueGatewayFilterFactory {


    @Override
    public GatewayFilter apply(NameValueConfig config) {
        return (exchange, chain) -> {
            log.info("调用CheckAuthGatewayFilterFactory==="
                    + config.getName() + ":" + config.getValue());
            return chain.filter(exchange);
        };
    }
}
```

配置自定义的过滤器工厂

```
spring:
  cloud:
    gateway:
      #设置路由：路由id、路由到微服务的uri、断言
      routes:
      - id: order_route  #路由ID，全局唯一
        uri: http://localhost:8020  #目标微服务的请求地址和端口
        #配置过滤器工厂
        filters:
        - CheckAuth=fox,男    #自定义过滤器工厂

```

测试

3.4 全局过滤器（Global Filters）配置

全局过滤器的作用也是处理一切进入网关的请求和微服务响应，与GatewayFilter的作用一样。

- GatewayFilter：网关过滤器，需要通过spring.cloud.routes.filters配置在具体的路由下，只作用在当前特定路由上，也可以通过配置spring.cloud.default-filters让它作用于全局路由上。
- GlobalFilter：全局过滤器，不需要再配置文件中配置，作用在所有的路由上，最终通过GatewayFilterAdapter包装成GatewayFilterChain能够识别的过滤器。

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

3.4.1 ReactiveLoadBalancerClientFilter

ReactiveLoadBalancerClientFilter 会查看exchange的属性ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR的值（一个URI，比如lb://mall-order/order/testgateway2?color=blue），如果该值的scheme是 lb，比如：lb://myservice ，它将会使用Spring Cloud的LoadBalancerClient 来将 myservice 解析成实际的host和port。

其实就是用来整合负载均衡器loadbalancer的

```
spring:
  cloud:
    gateway:
      routes:
      - id: order_route
        uri: lb://mall-order
        predicates:
        - Path=/order/**
```

3.4.2 自定义全局过滤器

自定义全局过滤器定义方式是实现GlobalFilter接口。每一个过滤器都必须指定一个int类型的order值，order值越小，过滤器优先级越高，执行顺序越靠前。GlobalFilter通过实现Ordered接口来指定order值

```java
@Component
@Slf4j
public class CheckAuthFilter implements GlobalFilter, Ordered {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        //获取token
        String token = exchange.getRequest().getHeaders().getFirst("token");
        if (null == token) {
            log.info("token is null");
            ServerHttpResponse response = exchange.getResponse();
            response.getHeaders().add("Content-Type",
                    "application/json;charset=UTF-8");
            // 401 用户没有访问权限
            response.setStatusCode(HttpStatus.UNAUTHORIZED);
            byte[] bytes = HttpStatus.UNAUTHORIZED.getReasonPhrase().getBytes();
            DataBuffer buffer = response.bufferFactory().wrap(bytes);
            // 请求结束，不继续向下请求
            return response.writeWith(Mono.just(buffer));
        }
        //TODO 校验token进行身份认证
        log.info("校验token");
        return chain.filter(exchange);
    }

    @Override
    public int getOrder() {
        return 2;
    }
}
```

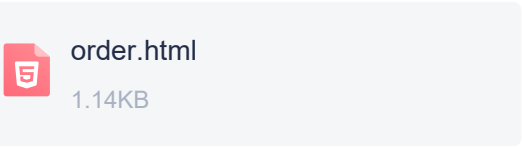
3.5 Gateway跨域资源共享配置（CORS Configuration）

在前端领域中，跨域是指浏览器允许向服务器发送跨域请求，从而克服Ajax只能同源使用的限制。

> 同源策略（Same Orgin Policy）是一种约定，它是浏览器核心也最基本的安全功能，它会阻止一个域的js脚本和另外一个域的内容进行交互，如果缺少了同源策略，浏览器很容易受到XSS、CSRF等攻击。所谓同源（即在同一个域）就是两个页面具有相同的协议（protocol）、主机（host）和端口号（port）。

> CORS： https://developer.mozilla.org/zh-CN/docs/Web/HTTP/CORS

测试代码



测试结果

如何解决gateway跨域问题?

通过yml配置的方式

https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#cors-configuration

```yaml
spring:
  cloud:
    gateway:
    globalcors:
      cors-configurations:
        '[/**]':
          allowedOrigins: "*"
          allowedMethods:
          - GET
          - POST
          - DELETE
          - PUT
          - OPTION

```

通过java配置的方式

```java
@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);
    }
}
```

3.6 Gateway基于redis+lua脚本限流

spring cloud官方提供了RequestRateLimiter过滤器工厂，基于redis+lua脚本方式采用令牌桶算法实现了限流。

https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#the-requestratelimiter-gatewayfilter-factory

请求不被允许时返回状态：HTTP 429 - Too Many Requests。

1）添加依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>
```

2）修改 application.yml，添加redis配置和RequestRateLimiter过滤器工厂配置

```
spring:
  application:
    name: mall-gateway

  data:
    #配置redis地址
```

```yaml
    redis:
      host: localhost
      port: 6379
      database: 0
      timeout: 5000
      lettuce:
        pool:
          max-active: 200
          max-wait: 10000
          max-idle: 100
          min-idle: 10
  #配置nacos注册中心地址
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848

    gateway:
      #设置路由：路由id、路由到微服务的uri、断言
      routes:
        - id: order_route  #路由ID，全局唯一，建议配置服务名
          # 测试 http://localhost:8888/order/findOrderByUserId/1
          uri: lb://mall-order  #lb 整合负载均衡器ribbon,loadbalancer
          predicates:
            - Path=/order/**   # 断言，路径相匹配的进行路由
          #配置过滤器工厂
          filters:
            - name: RequestRateLimiter   #限流过滤器
              args:
                redis-rate-limiter.replenishRate: 1 #令牌桶每秒填充速率
                redis-rate-limiter.burstCapacity: 2 #令牌桶的总容量
                key-resolver: "#{@keyResolver}" #使用SpEL表达式，从Spring容器中获取Bean对象
```

3) 配置keyResolver，可以指定限流策略，比如url限流，参数限流，ip限流等等

```
@Bean
KeyResolver keyResolver() {
    //url限流
```

```
4       return exchange -> Mono.just(exchange.getRequest().getURI().getPath());
5       //参数限流
6       //return exchange ->
    Mono.just(exchange.getRequest().getQueryParams().getFirst("user"));
7   }
```

4) 测试

url限流：http://localhost:8888/order/findOrderByUserId/1

参数限流：http://localhost:8888/order/findOrderByUserId/1?user=fox

3.7 Gateway整合sentinel限流

从 1.6.0 版本开始，Sentinel 提供了 Spring Cloud Gateway 的适配模块，可以提供两种资源维度的限流：

- route 维度：即在 Spring 配置文件中配置的路由条目，资源名为对应的 routeId
- 自定义 API 维度：用户可以利用 Sentinel 提供的 API 来自定义一些 API 分组

> sentinel网关流控：https://sentinelguard.io/zh-cn/docs/api-gateway-flow-control.html

3.7.1 Gateway整合sentinel实现网关限流

1）引入依赖

```xml
<!-- gateway接入sentinel  -->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-alibaba-sentinel-gateway</artifactId>
</dependency>

<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>
```

2）添加yml配置，接入sentinel dashboard，通过sentinel控制台配置网关流控规则

```yaml
server:
  port: 8888
spring:
  application:
    name: mall-gateway-sentinel-demo
  main:
    allow-bean-definition-overriding: true
  #配置nacos注册中心地址
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848

    sentinel:
      transport:
        # 添加sentinel的控制台地址
        dashboard: 127.0.0.1:8080

    gateway:
      #设置路由：路由id、路由到微服务的uri、断言
      routes:
      - id: order_route  #路由ID，全局唯一，建议配合服务名
        uri: lb://mall-order  #lb 整合负载均衡器loadbalancer
        predicates:
```

```
        - Path=/order/**

      - id: user_route
        uri: lb://mall-user  #lb 整合负载均衡器loadbalancer
        predicates:
        - Path=/user/**
```

注意：基于SpringBoot3的 Spring Cloud Gateway和Sentinel还存在兼容性问题，等待Sentinel官方对最新的Gateway适配包更新

3.7.2 Sentinel网关流控实现原理

当通过 GatewayRuleManager 加载网关流控规则（GatewayFlowRule）时，无论是否针对请求属性进行限流，Sentinel 底层都会将网关流控规则转化为热点参数规则（ParamFlowRule），存储在 GatewayRuleManager 中，与正常的热点参数规则相隔离。转换时 Sentinel 会根据请求属性配置，为网关流控规则设置参数索引（idx），并同步到生成的热点参数规则中。

外部请求进入 API Gateway 时会经过 Sentinel 实现的 filter，其中会依次进行 路由/API 分组匹配、请求属性解析和参数组装。Sentinel 会根据配置的网关流控规则来解析请求属性，并依照参数索引顺序组装参数数组，最终传入 SphU.entry(res, args) 中。Sentinel API Gateway Adapter Common 模块向 Slot Chain 中添加了一个 GatewayFlowSlot，专门用来做网关规则的检查。GatewayFlowSlot 会从 GatewayRuleManager 中提取生成的热点参数规则，根据传入的参数依次进行规则检查。若某条规则不针对请求属性，则会在参数最后一个位置置入预设的常量，达到普通流控的效果。