主讲老师：Fox

1. 什么是Spring Cloud Gateway

网关作为流量的入口，常用的功能包括路由转发，权限校验，限流等。

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。相比 Zuul 来说，Spring Cloud Gateway 提供更优秀的性能，更强大的有功能。

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 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 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/#gateway-request-predicates-factories

1.1 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。如果断言为真，则说明请求的URL和配置的路由匹配。

- 断言(predicates)

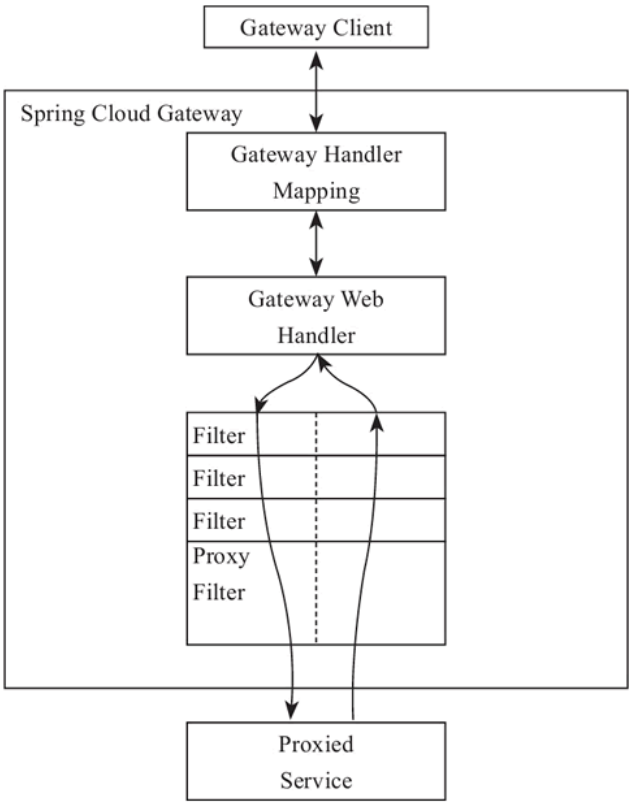
Java8中的断言函数，SpringCloud Gateway中的断言函数类型是Spring5.0框架中的ServerWebExchange。断言函数允许开发者去定义匹配Http request中的任何信息，比如请求头和参数等。

- 过滤器（Filter)

SpringCloud Gateway中的filter分为Gateway Filler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。

2. Spring Cloud Gateway快速开始

2.1 环境搭建

1) 引入依赖

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

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

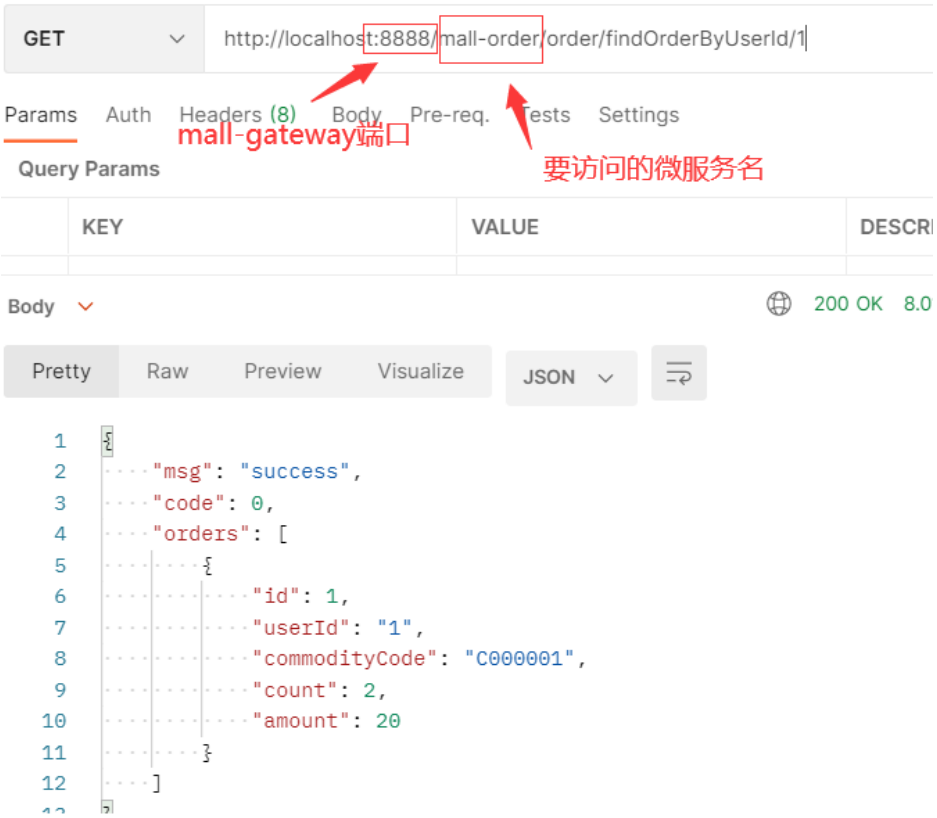
注意：会和spring-webmvc的依赖冲突，需要排除spring-webmvc

2) 编写yml配置文件

```
server:
 port: 8888
spring:
 application:
 name: mall-gateway
 #配置nacos注册中心地址
 cloud:
```

```yaml
nacos:
discovery:
 server-addr: 127.0.0.1:8848

 gateway:
 discovery:
 locator:
 # 默认为false，设为true开启通过微服务创建路由的功能，即可以通过微服务名访问服务
 # http://localhost:8888/mall-order/order/findOrderByUserId/1
 enabled: true
 # 是否开启网关
 enabled: true
```

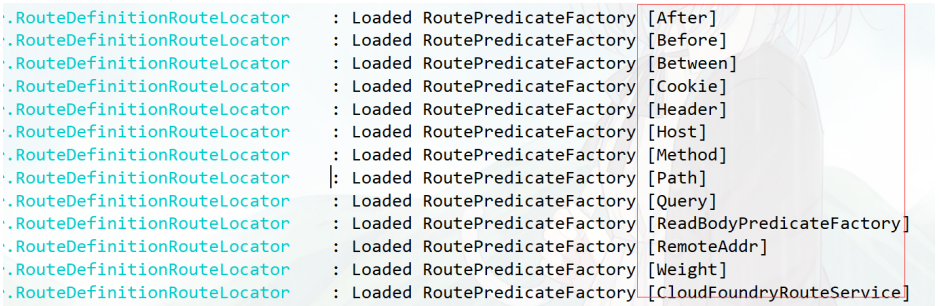
3）测试



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

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

网关启动日志：



2.2.1 时间匹配

可以用在限时抢购的一些场景中。

5.1. The After Route Predicate Factory

5.2. The Before Route Predicate Factory

5.3. The Between Route Predicate Factory

```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
 predicates:
 # 测试：http://localhost:8888/order/findOrderByUserId/1
  # 匹配在指定的日期时间之后发生的请求 入参是ZonedDateTime类型
  - After=2021-01-31T22:22:07.783+08:00[Asia/Shanghai]
```

获取ZonedDateTime类型的指定日期时间

```java
ZonedDateTime zonedDateTime = ZonedDateTime.now();//默认时区
// 用指定时区获取当前时间
ZonedDateTime zonedDateTime2 = ZonedDateTime.now(ZoneId.of("Asia/Shanghai"));
```

设置时间之前发起请求：

```
{
    "timestamp": "2021-01-31T14:21:50.803+0000",
    "path": "/order/findOrderByUserId/1",
    "status": 404,
    "error": "Not Found",
    "message": null,
    "requestId": "0ab3a9e6-2"
}
```

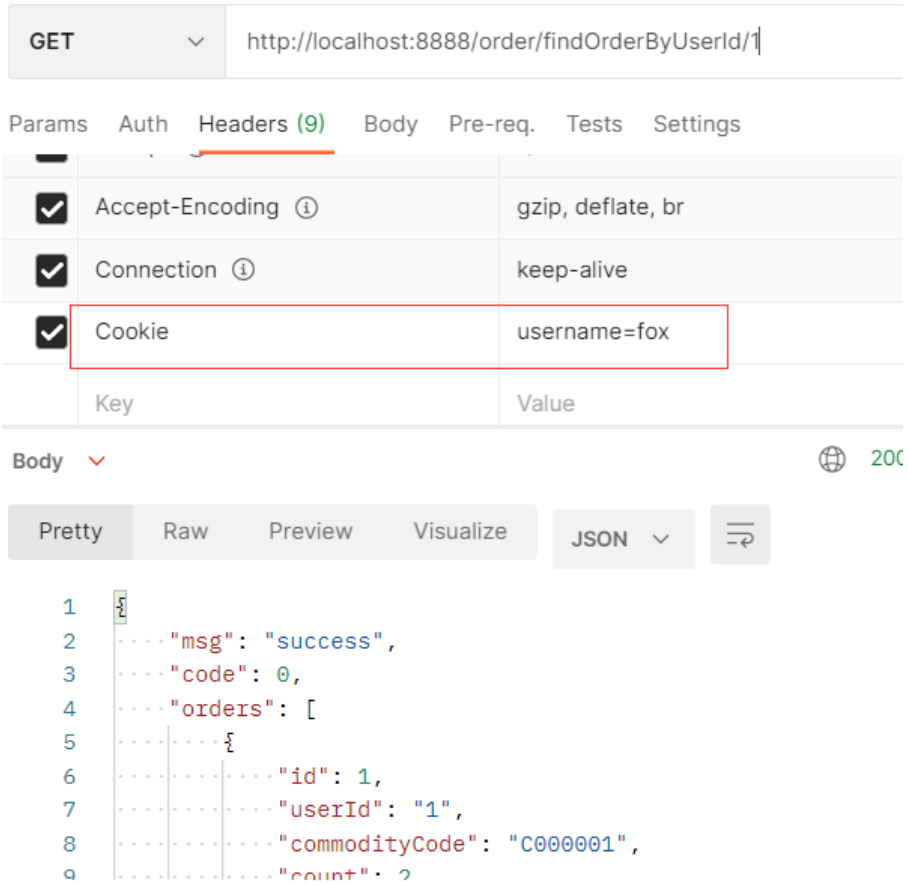
超过设置时间之后再次请求：

```
{
    "msg": "success",
    "code": 0,
    "orders": [
        {
            "id": 1,
            "userId": "1",
            "commodityCode": "C000001",
            "count": 2,
            "amount": 20
        }
    ]
```

2.2.2 Cookie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: http://localhost:8020 #目标微服务的请求地址和端口
 predicates:
 # Cookie匹配
  - Cookie=username, fox
```

postman测试



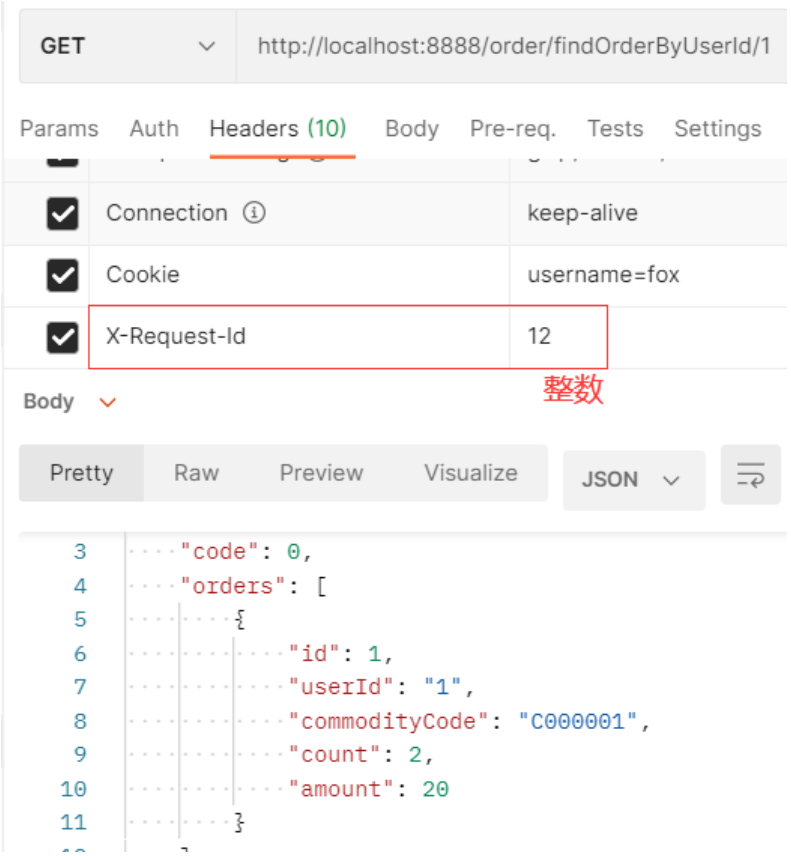
curl测试

```
F:\Resource\nacos\vip-spring-cloud-alibaba>curl http://localhost:8888/order/findOrderByUserId/1 --Cookie username=fox
{"msg":"success","code":0,"orders":[{"id":1,"userId":"1","commodityCode":"C000001","count":2,"amount":20}]}
```

2.2.3 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: http://localhost:8020 #目标微服务的请求地址和端口
 predicates:
 # 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



2.2.4 路径匹配

```
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
 predicates:
 # 测试：http://localhost:8888/order/findOrderByUserId/1
  - Path=/order/** #Path路径匹配
```

2.2.5 自定义路由断言工厂

自定义路由断言工厂需要继承 AbstractRoutePredicateFactory 类，重写 apply 方法的逻辑。在 apply 方法中可以通过 exchange.getRequest() 拿到 ServerHttpRequest 对象，从而可以获取到请求的参数、请求方式、请求头等信息。

注意： 命名需要以 RoutePredicateFactory 结尾

```
@Component
@Slf4j
public class CheckAuthRoutePredicateFactory extends AbstractRoutePredicateFactory<CheckAuthRoutePredic
ateFactory.Config> {

 public CheckAuthRoutePredicateFactory() {
 super(Config.class);
 }

 @Override
  public Predicate<ServerWebExchange> apply(Config config) {
  return new GatewayPredicate() {

  @Override
```

```
14  public boolean test(ServerWebExchange serverWebExchange) {
15  log.info("调用CheckAuthRoutePredicateFactory" + config.getName());
16  if(config.getName().equals("fox")){
17  return true;
18  }
19  return false;
20  }
21  };
22  }
23
24  /**
25  * 快捷配置
26  * @return
27  */
28  @Override
29  public List<String> shortcutFieldOrder() {
30  return Collections.singletonList("name");
31  }
32
33  public static class Config {
34
35  private String name;
36
37  public String getName() {
38  return name;
39  }
40
41  public void setName(String name) {
42  this.name = name;
43  }
44  }
45 }
```

yml中配置

```
1  spring:
2   cloud:
3   gateway:
4   #设置路由：路由id、路由到微服务的uri、断言
5   routes:
6   - id: order_route #路由ID，全局唯一
7   uri: http://localhost:8020 #目标微服务的请求地址和端口
8   predicates:
9   # 测试：http://localhost:8888/order/findOrderByUserId/1
10   - Path=/order/** #Path路径匹配
11   #自定义CheckAuth断言工厂
12  # - name: CheckAuth
13  # args:
14  # name: fox
15   - CheckAuth=fox
```

2.3 过滤器工厂（GatewayFilter Factories）配置

SpringCloudGateway 内置了很多的过滤器工厂，我们通过一些过滤器工厂可以进行一些业务逻辑处理器，比如添加剔除响应头，添加去除参数等

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

2.3.1 添加请求头

```
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

```
@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";
}
```

```
controller.OrderController    : gateWay获取请求头X-Request-color: red
controller.OrderController    : gateWay获取请求头X-Request-color: red
```

2.3.2 添加请求参数

```
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

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

```
c.t.m.order.controller.OrderController    : gateWay获取请求参数color:blue
```

2.3.3 为匹配的路由统一添加前缀

```
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:
 - PrefixPath=/mall-order # 添加前缀 对应微服务需要配置context-path
```

mall-order中需要配置

```
server:
 servlet:
 context-path: /mall-order
```

测试：http://localhost:8888/order/findOrderByUserId/1 ====》 http://localhost:8020/mall-order/order/findOrderByUserId/1

2.3.4 重定向操作

```
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:
 - RedirectTo=302, http://baidu.com #重定向到百度
```

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

2.3.5 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:
```

```
10  - CheckAuth=fox,男
11
```

测试

```
.g.f.CheckAuthGatewayFilterFactory  : 调用CheckAuthGatewayFilterFactory===fox:男
```

2.4 全局过滤器（Global Filters）配置

GlobalFilter 接口和 GatewayFilter 有一样的接口定义，只不过， GlobalFilter 会作用于所有路由。

官方声明：GlobalFilter的接口定义以及用法在未来的版本可能会发生变化。

2.4.1 LoadBalancerClientFilter

LoadBalancerClientFilter 会查看exchange的属性 ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR 的值（一个URI），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，并替换掉 ServerWebExchangeUtils.GATEWAY_REQUEST_URL_ATTR 的内容。

其实就是用来整合负载均衡器Ribbon的

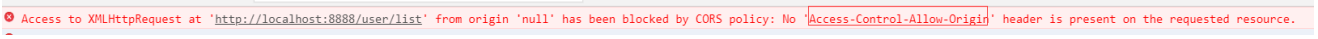
```yaml
spring:
 cloud:
 gateway:
 routes:
 - id: order_route
 uri: lb://mall-order
 predicates:
 - Path=/order/**
```

2.4.2 自定义全局过滤器

```java
@Component
@Order(-1)
@Slf4j
public class CheckAuthFilter implements GlobalFilter {
 @Override
 public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
 //校验请求头中的token
 List<String> token = exchange.getRequest().getHeaders().get("token");
 log.info("token:"+ token);
 if (token.isEmpty()){
 return null;
 }
 return chain.filter(exchange);
 }
```

```
}

@Component
public class CheckIPFilter implements GlobalFilter, Ordered {

 @Override
 public int getOrder() {
 return 0;
 }

 @Override
 public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
 HttpHeaders headers = exchange.getRequest().getHeaders();
 //模拟对 IP 的访问限制，即不在 IP 白名单中就不能调用的需求
 if (getIp(headers).equals("127.0.0.1")) {
 return null;
 }
 return chain.filter(exchange);
 }

 private String getIp(HttpHeaders headers) {
 return headers.getHost().getHostName();
 }
}
```

2.5 Gateway跨域配置（CORS Configuration）



通过yml配置的方式

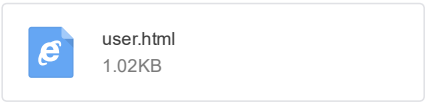
https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/#cors-configuration

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

```

通过java配置的方式


```
@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);
  }
}
```

测试



2.6 gateway整合sentinel限流

https://github.com/alibaba/Sentinel/wiki/%E7%BD%91%E5%85%B3%E9%99%90%E6%B5%81

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

2.6.1 快速开始

使用时需引入依赖：

```
<dependency>
 <groupId>com.alibaba.csp</groupId>
 <artifactId>sentinel-spring-cloud-gateway-adapter</artifactId>
 <version>x.y.z</version>
</dependency>
```

接入sentinel dashboard，添加yml配置

```
spring:
```

```
  application:
  name: mall-gateway-sentinel-demo
  #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
```

使用时只需注入对应的 SentinelGatewayFilter 实例以及 SentinelGatewayBlockExceptionHandler 实例即可

```
@Configuration
public class GatewayConfiguration {

 private final List<ViewResolver> viewResolvers;
 private final ServerCodecConfigurer serverCodecConfigurer;

 public GatewayConfiguration(ObjectProvider<List<ViewResolver>> viewResolversProvider,
 ServerCodecConfigurer serverCodecConfigurer) {
 this.viewResolvers = viewResolversProvider.getIfAvailable(Collections::emptyList);
  this.serverCodecConfigurer = serverCodecConfigurer;
  }

  /**
  * 限流异常处理器
  * @return
  */
  @Bean
  @Order(Ordered.HIGHEST_PRECEDENCE)
  public SentinelGatewayBlockExceptionHandler sentinelGatewayBlockExceptionHandler() {
  // Register the block exception handler for Spring Cloud Gateway.
  return new SentinelGatewayBlockExceptionHandler(viewResolvers, serverCodecConfigurer);
  }

  /**
  * 限流过滤器
  * @return
  */
  @Bean
  @Order(Ordered.HIGHEST_PRECEDENCE)
  public GlobalFilter sentinelGatewayFilter() {
  return new SentinelGatewayFilter();
  }

}
```

用户可以通过 GatewayRuleManager.loadRules(rules) 手动加载网关规则
GatewayConfiguration中添加

```
@PostConstruct
public void doInit() {
```

```
//初始化自定义的API
initCustomizedApis();
//初始化网关限流规则
initGatewayRules();
//自定义限流异常处理器
initBlockRequestHandler();
}

 private void initCustomizedApis() {
 Set<ApiDefinition> definitions = new HashSet<>();
 ApiDefinition api = new ApiDefinition("user_service_api")
 .setPredicateItems(new HashSet<ApiPredicateItem>() {{
 add(new ApiPathPredicateItem().setPattern("/user/**")
 .setMatchStrategy(SentinelGatewayConstants.URL_MATCH_STRATEGY_PREFIX));
 }});
 definitions.add(api);
 GatewayApiDefinitionManager.loadApiDefinitions(definitions);
 }

 private void initGatewayRules() {
 Set<GatewayFlowRule> rules = new HashSet<>();
 //resource：资源名称，可以是网关中的 route 名称或者用户自定义的 API 分组名称。
 //count：限流阈值
 //intervalSec：统计时间窗口，单位是秒，默认是 1 秒。
 rules.add(new GatewayFlowRule("order_route")
 .setCount(2)
 .setIntervalSec(1)
 );
 rules.add(new GatewayFlowRule("user_service_api")
 .setCount(2)
 .setIntervalSec(1)
 );

 // 加载网关规则
 GatewayRuleManager.loadRules(rules);
 }

 private void initBlockRequestHandler() {
 BlockRequestHandler blockRequestHandler = new BlockRequestHandler() {
 @Override
 public Mono<ServerResponse> handleRequest(ServerWebExchange exchange, Throwable t) {
 HashMap<String, String> result = new HashMap<>();
 result.put("code",String.valueOf(HttpStatus.TOO_MANY_REQUESTS.value()));
 result.put("msg", HttpStatus.TOO_MANY_REQUESTS.getReasonPhrase());

 return ServerResponse.status(HttpStatus.TOO_MANY_REQUESTS)
 .contentType(MediaType.APPLICATION_JSON)
 .body(BodyInserters.fromValue(result));
 }
 };
 //设置自定义异常处理器
 GatewayCallbackManager.setBlockHandler(blockRequestHandler);
 }
```

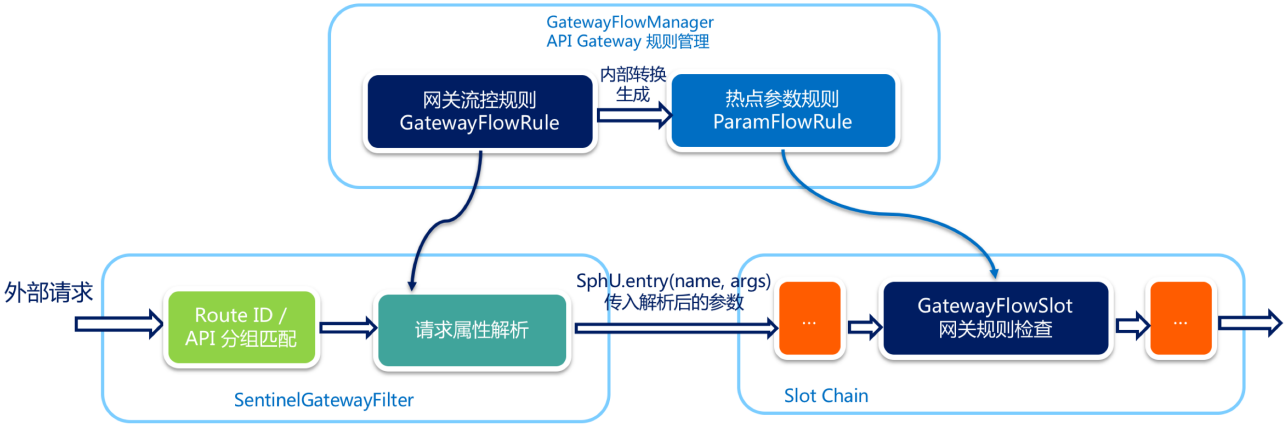
2.6.2 网关流控控制台

Sentinel 1.6.3 引入了网关流控控制台的支持，用户可以直接在 Sentinel 控制台上查看 API Gateway 实时的 route 和自定义 API 分组监控，管理网关规则和 API 分组配置。

在 API Gateway 端，用户只需要在原有启动参数的基础上添加如下启动参数即可标记应用为 API Gateway 类型：

```
# 注：通过 Spring Cloud Alibaba Sentinel 自动接入的 API Gateway 整合则无需此参数
-Dcsp.sentinel.app.type=1
```

2.6.3 网关流控实现原理



2.7 网关高可用

为了保证 Gateway 的高可用性，可以同时启动多个 Gateway 实例进行负载，在 Gateway 的上游使用 Nginx 或者 F5 进行负载转发以达到高可用。

```
upstream gateway{
    server localhost:8889;
    server localhost:8888;
}

server {
    listen       80;
    server_name  localhost;

    #charset koi8-r;

    #access_log  logs/host.access.log  main;

    location / {
        #root   html;
        #index  index.html index.htm;
        proxy_pass http://gateway;

    }
```

文档：17 微服务网关Gateway实战.note

链接：http://note.youdao.com/noteshare?id=e6a0c6530154a553fd11593f56b78c9a&sub=514AF1BC0A7D40899470143D797E895C