一:什么是SpringCloud gateWay

Spring Cloud Gateway是Spring Cloud官方推出的第二代网关框架，取代Zuul网关。网关作为流量的，在微服务系统中有着非常作用。据说性能是第一代网关zuul的1.5倍。(基于Netty,WebFlux)，

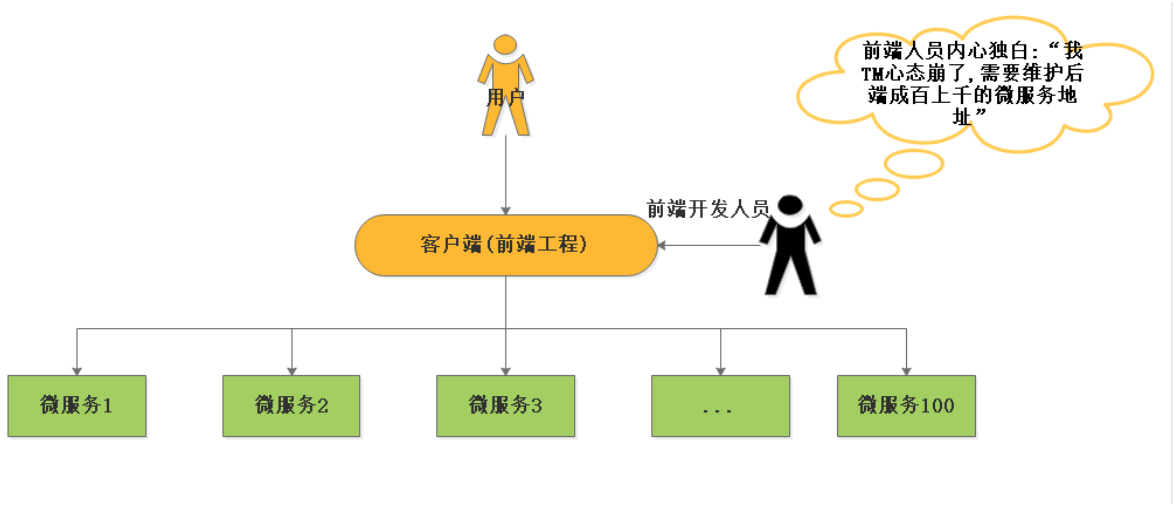
注意点:由于不是Sevlet容器，所以他不能打成war包, 只支持SpringBoot2.X不支持1.x

1.1)网关作用:

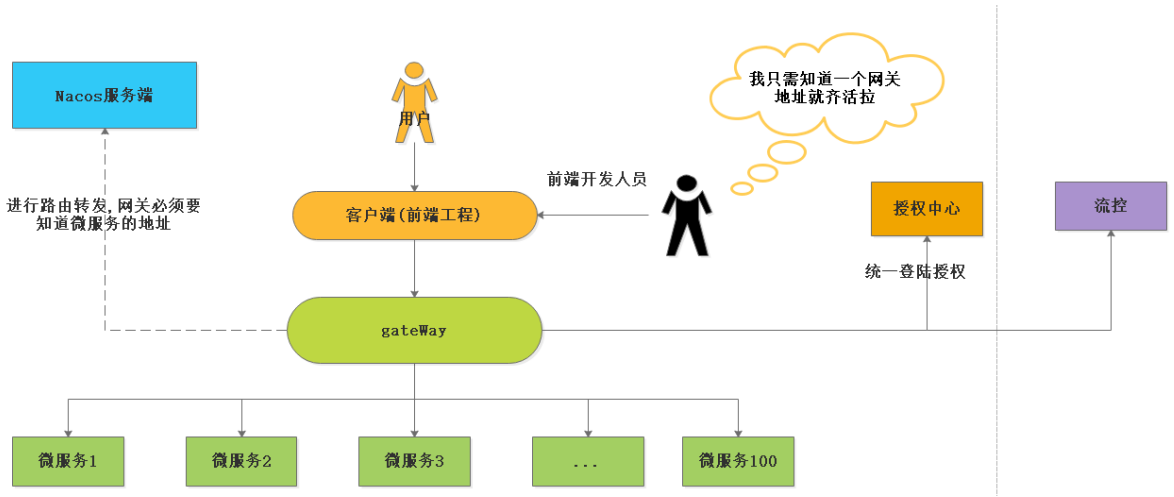
网关常见的功能有路由转发、权限校验、限流控制等作用。

1.2)为什么要使用SpringCloudGateWay。

①:没有网关



②:使用了网关



二:搭建SpringCloudGateWay的三板斧

2.1)创建一个gateWay的工程tulingvip08-ms-cloud-gateway

①添加依赖:

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

 <!--加入nacos的依赖-->
 <dependency>
 <groupId>com.alibaba.cloud</groupId>
  <artifactId>spring-cloud-alibaba-nacos-discovery</artifactId>
  </dependency>


  <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
  </dependency>
```

②：写配置文件

```
#规划GateWay的服务端口
server:
 port: 8888
#规划gateWay注册到到nacos上的服务应用名称
spring:
 application:
 name: api-gateway
 cloud:
 nacos:
  discovery:
  #gateway工程注册到nacos上的地址
  server-addr: localhost:8848
  gateway:
  discovery:
  locator:
  #开启gateway从nacos上获取服务列表
  enabled: true
#开启acutor端点
management:
 endpoints:
```

```
21  web:
22  exposure:
23  include: '*'
24  endpoint:
25  health:
26  #打开端点详情
27  show-details: always
```

③:写注解 服务发现的注解，gateway没有注解

```
1  @SpringBootApplication
2  @EnableDiscoveryClient
3  public class Tulingvip08MsCloudGatewayApplication {
4
5   public static void main(String[] args) {
6   SpringApplication.run(Tulingvip08MsCloudGatewayApplication.class, args);
7   }
8  }
```

2.2)测试网关工程,分别启动

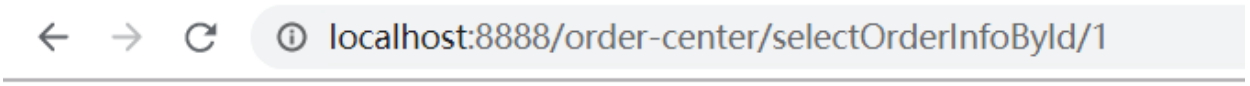
tulingvip08-ms-cloud-gateway(8888),

tulingvip08-ms-alibaba-gateway-order(8080)

tulingvip08-ms-alibaba-gateway-product(8084)

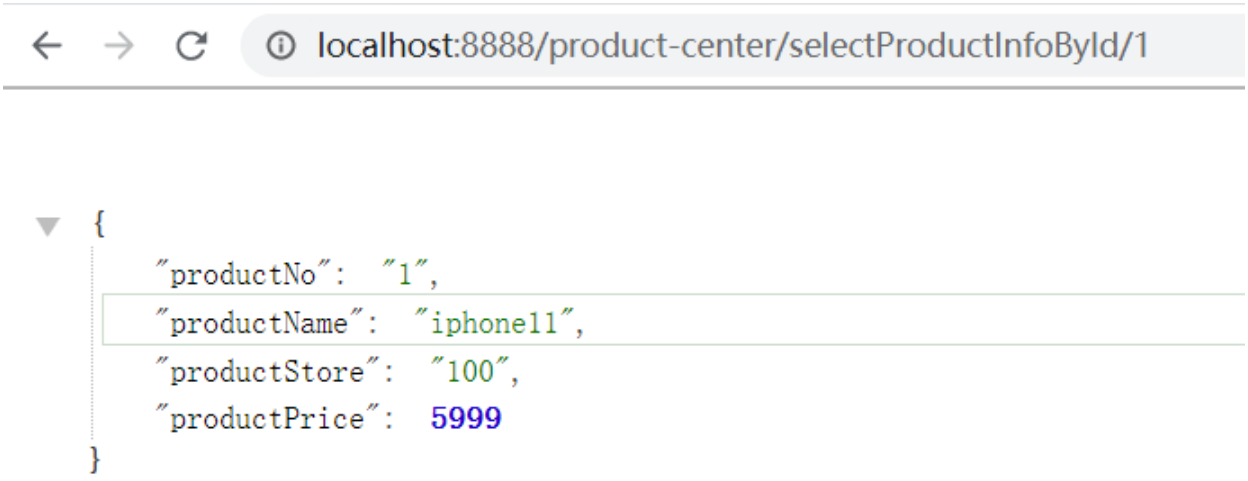
通过网关地址访问订单微服务

http://localhost:8888/order-center/selectOrderInfoById/1

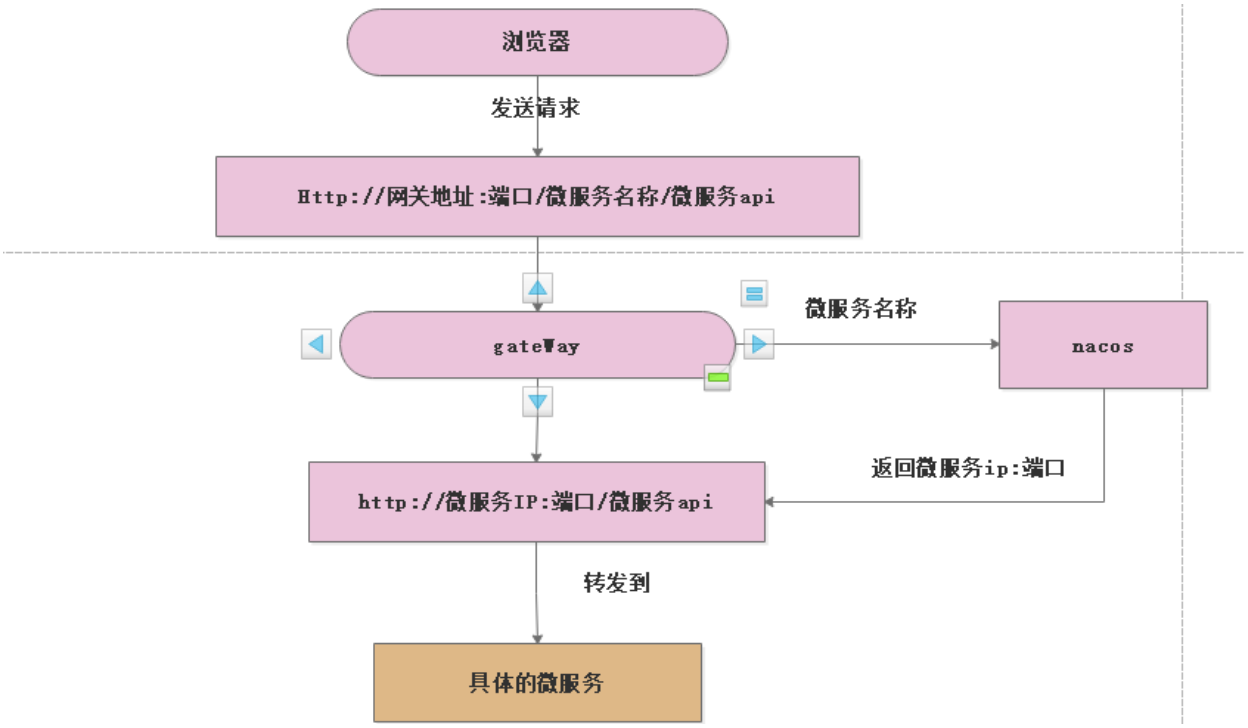


通过网关地址访问库存微服务

http://localhost:8888/product-center/selectProductInfoById/1



转发规则:



三:GateWay的核心概念

3.1）基本核心概念.

路由网关的基本构建模块，它是由ID、目标URI、断言集合和过滤器集合定义，如果集合断言为真，则匹配路由。

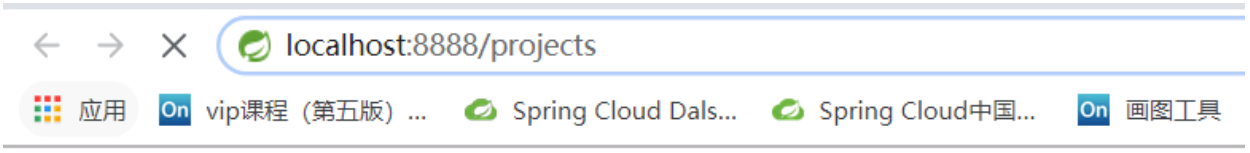
Predicate(断言)：这是java 8的一个函数式接口predicate,可以用于lambda表达式和方法引用，输入类型是：Spring Framework ServerWebExchange,允许开发人员匹配来自HTTP请求的任何内容，例如请求头headers和参数paramers

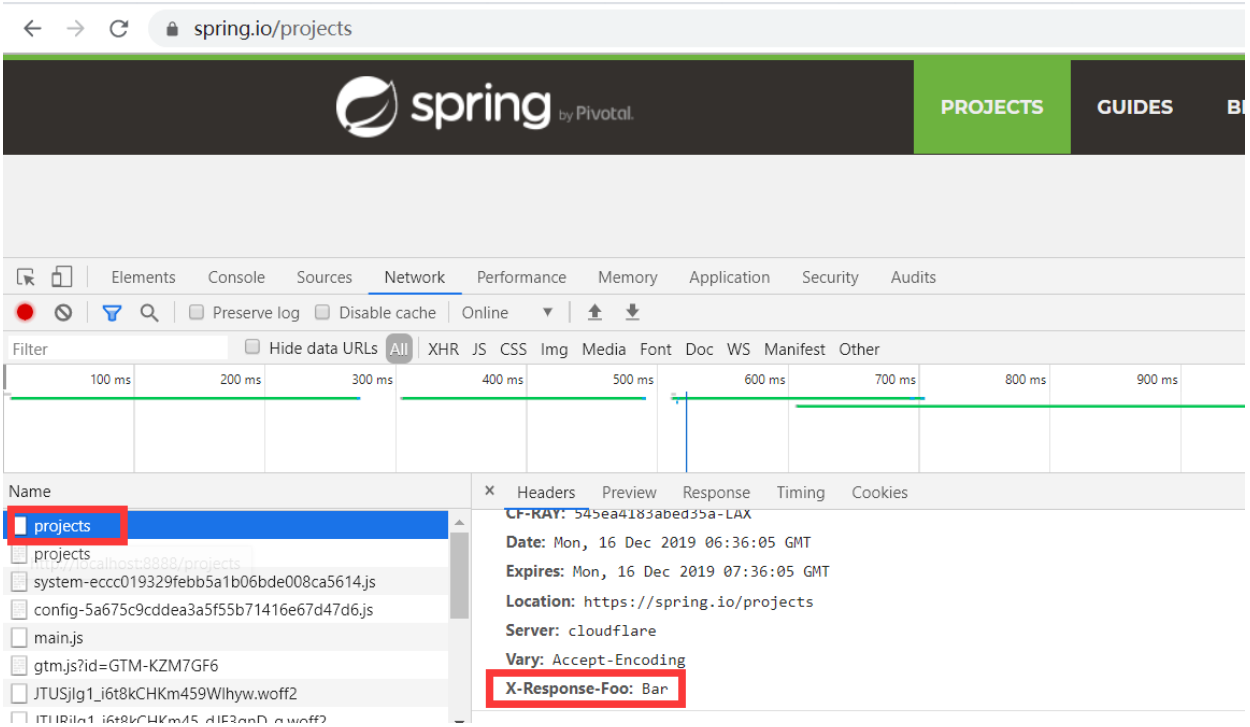
Filter(过滤器)：这些是使用特定工厂构建的Spring Framework GatewayFilter实例，这里可以在发送下游请求之前或之后修改请求和响应

如下配置:

含义:我们浏览器 http://localhost:8888/projects/ 都会转发到 http://spring.io/projects/**下 并且带入响应头部: X-Response-Foo=Bar**

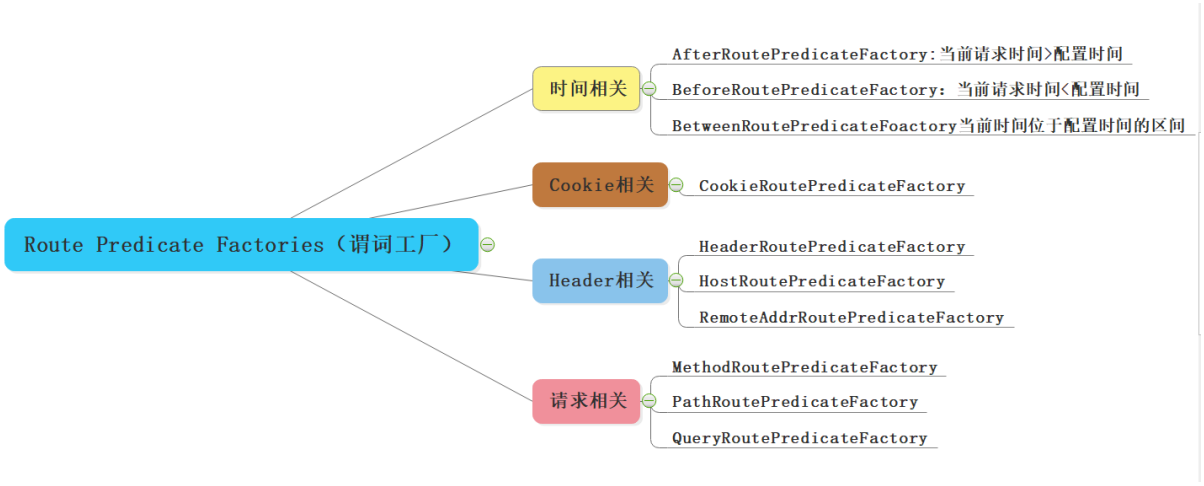
```
spring:
 application:
 name: api-gateway
 cloud:
 gateway:
 routes:
 - id: add_request_header_route
 uri: http://spring.io
 predicates:
  - Path=/projects/**
  filters:
  - AddResponseHeader=X-Response-Foo, Bar
```



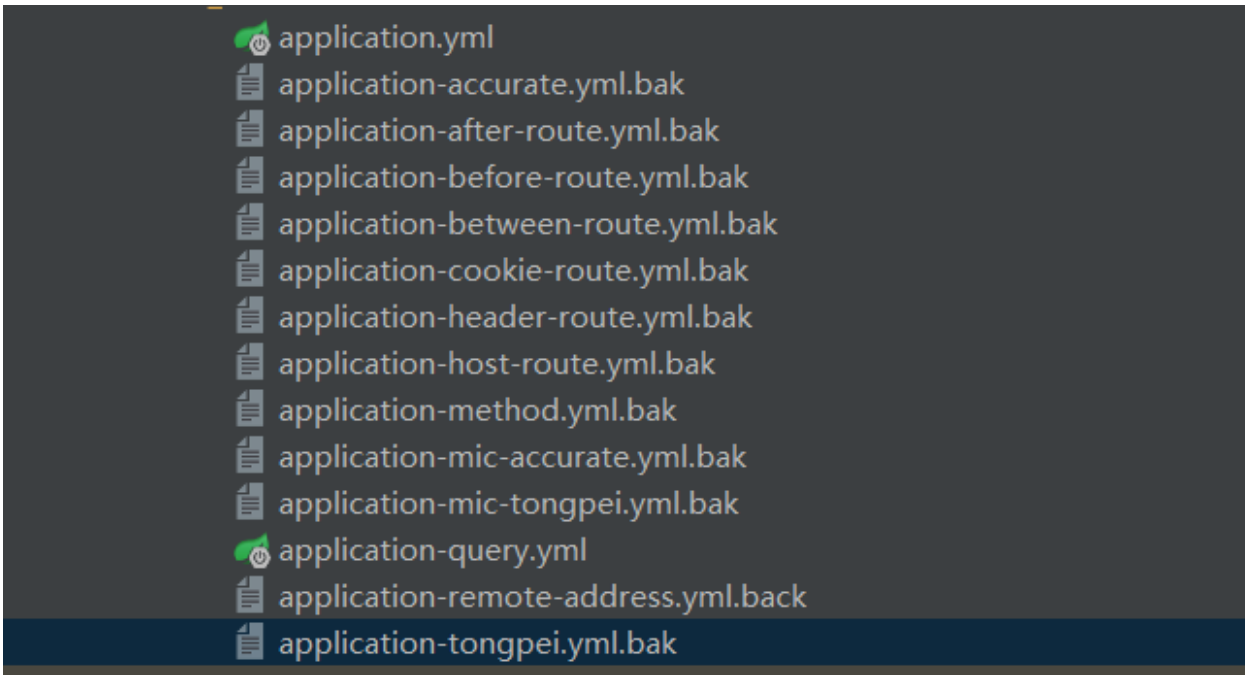


3.2)路由断言工厂

https://cloud.spring.io/spring-cloud-gateway/2.1.x/multi/multi_spring-cloud-gateway.html



谓词工厂配置老师这里都配置好了，大家可以自行的尝试其他的



3.3）自定义谓词工厂

第一步:写一个自定义谓词工厂，类名必须要以RoutePredicateFactory结尾然后继承AbstractRoutePredicateFactory

```
1  @Component
2  @Slf4j
3  public class TulingTimeBetweenRoutePredicateFactory extends AbstractRoute
PredicateFactory<TulingTimeBetweenConfig> {
4
5   public TulingTimeBetweenRoutePredicateFactory() {
6   super(TulingTimeBetweenConfig.class);
7   }
```

```
8
9   //真正的业务判断逻辑
10   @Override
11   public Predicate<ServerWebExchange> apply(TulingTimeBetweenConfig confi
g) {
12
13   LocalTime startTime = config.getStartTime();
14
15   LocalTime endTime = config.getEndTime();
16
17   return new Predicate<ServerWebExchange>(){
18   @Override
19   public boolean test(ServerWebExchange serverWebExchange) {
20   LocalTime now = LocalTime.now();
21   //判断当前时间是否在在配置的时间范围类
22   return now.isAfter(startTime) && now.isBefore(endTime);
23   }
24   };
25
26   }
27
28   //用于接受yml中的配置 - TulingTimeBetween=上午7:00,下午11:00
29   public List<String> shortcutFieldOrder() {
30   return Arrays.asList("startTime", "endTime");
31   }
32
33  }
34
```

第二步：书写一个配置类,用于接受配置

```
1  //写一个类用于接受配置
2  @Data
3  public class TulingTimeBetweenConfig {
4
5   private LocalTime startTime;
6
7   private LocalTime endTime;
8
9  }
```

第三步:在yml配置中

谓词配置是以我们自定义类名TulingTimeBetweenRoutePredicateFactory去除了RoutePredicateFactory接受开头TulingTimeBetween

```
spring:
 cloud:
 gateway:
 routes:
 - id: tuling-timeBetween #id必须要唯一
 uri: lb://product-center
 predicates:
 #当前请求的时间必须在早上7点到 晚上11点 http://localhost:8888/selectProductInfoById/1
 #才会被转发
  #到http://product-center/selectProductInfoById/1
  - TulingTimeBetween=上午7:00,下午11:00
```

3.4)过滤器工厂,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://cloud.spring.io/spring-cloud-gateway/2.1.x/multi/multi__gatewayfilter_factories.html

老师在这里拿出几个来演示。

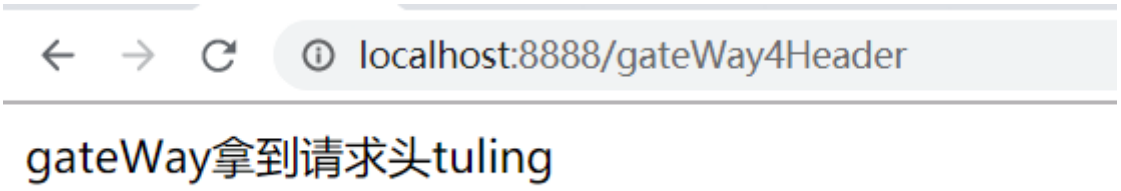
①:添加请求头。

```
spring:
 cloud:
 gateway:
 routes:
 - id: tuling-timeBetween #id必须要唯一
 uri: lb://product-center
 predicates:
 #当前请求的时间必须在早上7点到 晚上11点 http://localhost:8888/selectProductInfoById/1
 #才会被转发
  #到http://product-center/selectProductInfoById/1
  - TulingTimeBetween=上午7:00,下午11:00
  filters:
  - AddRequestHeader=X-Request-Company,tuling
```

测试:http://localhost:8888/gateWay4Header

```java
@RequestMapping("/gateWay4Header")
public Object gateWay4Header(@RequestHeader("X-Request-Company") String company) {

 return "gateWay拿到请求头"+company;
}
```



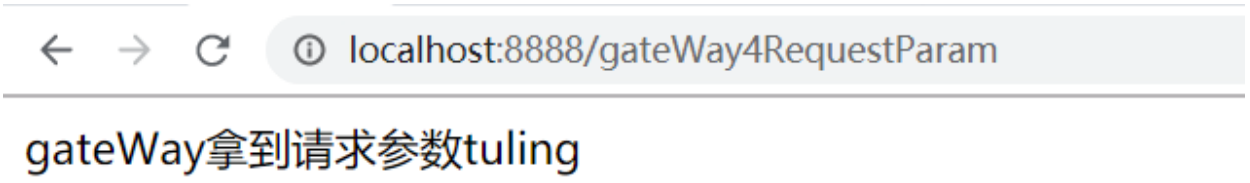
②:添加请求参数

```yaml
spring:
 cloud:
 gateway:
 routes:
 - id: tuling-timeBetween #id必须要唯一
 uri: lb://product-center
 predicates:
 - TulingTimeBetween=上午7:00,下午11:00
 filters:
  - AddRequestParameter=company, tuling
```

测试地址:http://localhost:8888/gateWay4RequestParam

```java
@RequestMapping("/gateWay4RequestParam")
public Object gateWay4RequestParam(@RequestParam("company") String company) {

 return "gateWay拿到请求参数"+company;
}
```



③:为匹配的路由统一添加前缀

```yaml
spring:
 cloud:
 gateway:
 routes:
 - id: tuling-timeBetween #id必须要唯一
 uri: lb://product-center
 predicates:
 - TulingTimeBetween=上午7:00,下午11:00
 filters:
  - PrefixPath=/product-api
  #比如
  http://localhost:8888/selectProductInfoById/1
  会转发到路径
  http://product-center/product-api/selectProductInfoById/1
```

我们的product-center的需要添加一段配置:

```yaml
server:
 servlet:
 context-path: /product-api
```

测试地址:http://localhost:8888/selectProductInfoById/2


```
{
    "productNo":  "2",
    "productName":  "华为meta30",
    "productStore":  "500",
    "productPrice":  4999
}
```


更多的配置 具体查看官网 已经详细的列出了20多种.
https://cloud.spring.io/spring-cloud-gateway/2.1.x/multi/multi__gatewayfilter_factories.html

④: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

```
@Slf4j
@Component
```

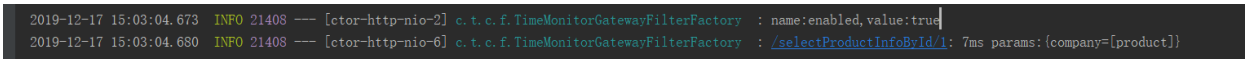
```
3  public class TimeMonitorGatewayFilterFactory extends AbstractNameValueGat
ewayFilterFactory {
4
5   private static final String COUNT_START_TIME = "countStartTime";
6
7
8   @Override
9   public GatewayFilter apply(NameValueConfig config) {
10
11   return new GatewayFilter() {
12   @Override
13   public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
14   //获取配置文件yml中的
15   filters:
16   - TimeMonitor=enabled,true
17   String name = config.getName();
18   String value = config.getValue();
19   log.info("name:{},value:{}",name,value);
20   if(value.equals("false")) {
21   return null;
22   }
23   exchange.getAttributes().put(COUNT_START_TIME,
System.currentTimeMillis());
24
25   //then方法相当于aop的后置通知一样
26   return chain.filter(exchange).then(Mono.fromRunnable(new Runnable() {
27   @Override
28   public void run() {
29   Long startTime = exchange.getAttribute(COUNT_START_TIME);
30   if (startTime != null) {
31   StringBuilder sb = new StringBuilder(exchange.getRequest().getURI().get
RawPath())
32   .append(": ")
33   .append(System.currentTimeMillis() - startTime)
34   .append("ms");
35   sb.append(" params:").append(exchange.getRequest().getQueryParams());
36   log.info(sb.toString());
37   }
38   }
39   }));
```

```
40    }
41    };
42    }
43  }
```

配置我们的自定义的过滤器工厂

```
1  spring:
2   cloud:
3   gateway:
4   routes:
5   - id: product
6   uri: lb://product-center
7   predicates:
8   - Query=company,product
9   filters:
10   - TimeMonitor=enabled,true
```

访问打印的日志



缺陷: 通过自定义过滤器工程创建出来的过滤器是不能指定优先级的，只能根据配置的先后顺序执行，若向指定优先级怎么办?

我们需要稍微改动一下代码: 写一个自定义的内部类实现GateWayFilter接口 和ordered接口,

```
1  @Slf4j
2  @Component
3  public class TimeMonitorGatewayFilterFactory extends AbstractNameValueGat
ewayFilterFactory {
4
5   private static final String COUNT_START_TIME = "countStartTime";
6
7
8   @Override
9   public GatewayFilter apply(NameValueConfig config) {
10   return new TimeMonitorGatewayFilter(config);
11   }
12
13   /**
14   * 我们自己写一个静态内部类 实现GatewayFilter,Ordered 通过Orderd可以实现顺序
的控制
```

```
15   */
16   public static class TimeMonitorGatewayFilter implements GatewayFilter,O
rdered{
17
18   private NameValueConfig nameValueConfig;
19
20   public TimeMonitorGatewayFilter(NameValueConfig nameValueConfig) {
21   this.nameValueConfig = nameValueConfig;
22   }
23
24   @Override
25   public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
26   String name = nameValueConfig.getName();
27   String value = nameValueConfig.getValue();
28   log.info("name:{},value:{}",name,value);
29   if(value.equals("false")) {
30   return null;
31   }
32   exchange.getAttributes().put(COUNT_START_TIME,
System.currentTimeMillis());
33
34   //then方法相当于aop的后置通知一样
35   return chain.filter(exchange).then(Mono.fromRunnable(new Runnable() {
36   @Override
37   public void run() {
38   Long startTime = exchange.getAttribute(COUNT_START_TIME);
39   if (startTime != null) {
40   StringBuilder sb = new StringBuilder(exchange.getRequest().getURI().get
RawPath())
41   .append(": ")
42   .append(System.currentTimeMillis() - startTime)
43   .append("ms");
44   sb.append(" params:").append(exchange.getRequest().getQueryParams());
45   log.info(sb.toString());
46   }
47   }
48   }));
49   }
50
51   @Override
```

```
52   public int getOrder() {
53   return -100;
54   }
55   }
56 }
```

⑤:自定义全局过滤器，所有的请求都会经过全局过滤器

实现GlobalGateWayFilter，那么所有的请求都会经过gateway

业务场景中。请求中必须带入token才会被转发.

```
1  /**
2   * 全局过滤器校验请求头中的token
3   * Created by smlz on 2019/12/17.
4   */
5  @Component
6  @Slf4j
7  public class AuthGateWayFilter implements GlobalFilter,Ordered {
8
9   @Override
10   public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
11   List<String> token = exchange.getRequest().getHeaders().get("token");
12   if(StringUtils.isEmpty(token)) {
13   return null;
14   }else {
15   log.info("token:{}",token);
16   return chain.filter(exchange);
17   }
18   }
19
20   @Override
21   public int getOrder() {
22   return 0;
23   }
24 }
```

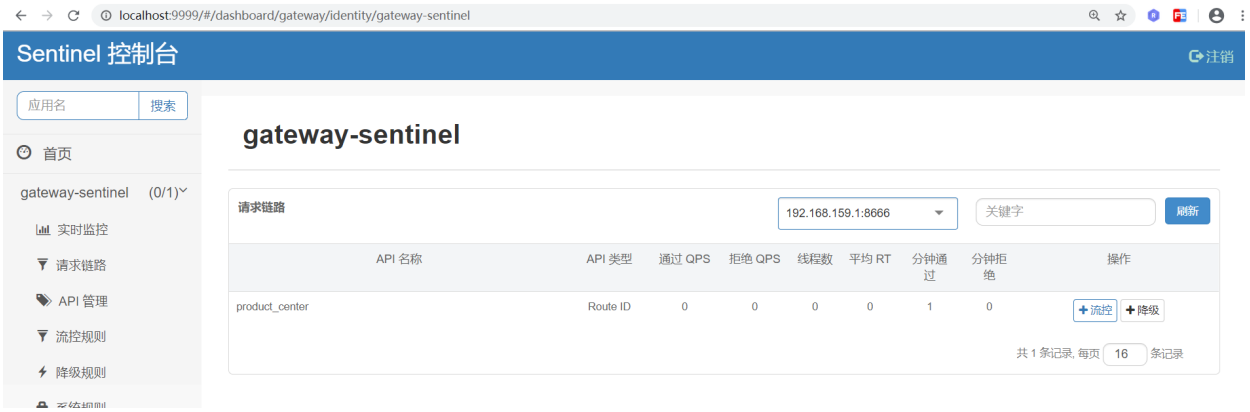
⑥:SpringCloudGateWay+Sentinel1.6.3(以上版本)

解释？为啥要1.6.3版本，若低于1.6.3版本的话，需要在gateway工程进行大量的编码进行设置流控的规则。具体见代码。

tulingvip08-ms-cloud-gateway-sentinel工程中com.tuling.config.GateWaySentinelConfig的

```
@PostConstruct
public void init() {
 //initCustomizedApis();
 //initGatewayRules();
}
```

若1.6.3版本以上，我们就可以通过sentinel页面进行配置规则



名称解释:

GatewayFlowRule：网关限流规则，针对 API Gateway 的场景定制的限流规则，可以针对不同 route 或自定义的 API 分组进行限流，支持针对请求中的参数、Header、来源 IP 等进行定制化的限流

ApiDefinition：用户自定义的 API 定义分组，可以看做是一些 URL 匹配的组合。比如我们可以定义一个 API 叫 my_api，请求 path 模式为 /foo/** 和 /baz/** 的都归到 my_api 这个 API 分组下面。限流的时候可以针对这个自定义的 API 分组维度进行限流

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。

resourceMode：规则是针对 API Gateway 的 **route**（RESOURCE_MODE_ROUTE_ID）还是用户在 Sentinel 中定义的 API 分组（RESOURCE_MODE_CUSTOM_API_NAME），默认是 route。

grade：限流指标维度，同限流规则的 grade 字段。

count：限流阈值

intervalSec：统计时间窗口，单位是秒，默认是 1 秒。

controlBehavior：流量整形的控制效果，同限流规则的 controlBehavior 字段，目前支持快速失败和匀速排队两种模式，默认是快速失败。

burst：应对突发请求时额外允许的请求数目。

axQueueingTimeoutMs：匀速排队模式下的最长排队时间，单位是毫秒，仅在匀速排队模式下生效。

paramItem：参数限流配置。若不提供，则代表不针对参数进行限流，该网关规则将会被转换成普通流控规则；否则会转换成热点规则。其中的字段：

parseStrategy：从请求中提取参数的策略，目前支持提取来源 **IP（PARAM_PARSE_STRATEGY_CLIENT_IP）、Host（PARAM_PARSE_STRATEGY_HOST）**、任意 Header（PARAM_PARSE_STRATEGY_HEADER）和任意 URL 参数（PARAM_PARSE_STRATEGY_URL_PARAM）四种模式。

fieldName：若提取策略选择 Header 模式或 URL 参数模式，则需要指定对应的 header 名称或 URL 参数名称。

pattern：参数值的匹配模式，只有匹配该模式的请求属性值会纳入统计和流控；若为空则统计该请求属性的所有值。（1.6.2 版本开始支持）

matchStrategy：参数值的匹配策略，目前支持精确匹配（PARAM_MATCH_STRATEGY_EXACT）、子串匹配（PARAM_MATCH_STRATEGY_CONTAINS）和正则匹配（PARAM_MATCH_STRATEGY_REGEX）。（1.6.2 版本开始支持）

用户可以通过 GatewayRuleManager.loadRules(rules) 手动加载网关规则，或通过 GatewayRuleManager.register2Property(property) 注册动态规则源动态推送（推荐方式）。

GateWay+Sentinel1.6.3版本整合

a)创建工程tulingvip08-ms-cloud-gateway-sentinel

导入依赖:

```
<!--加入nacos的依赖-->
<dependency>
 <groupId>com.alibaba.cloud</groupId>
 <artifactId>spring-cloud-alibaba-nacos-discovery</artifactId>
</dependency>
```

```
6
7  <dependency>
8   <groupId>org.springframework.cloud</groupId>
9   <artifactId>spring-cloud-starter-gateway</artifactId>
10  </dependency>
11  <dependency>
12   <groupId>org.springframework.boot</groupId>
13   <artifactId>spring-boot-starter-webflux</artifactId>
14  </dependency>
15
16  <dependency>
17   <groupId>com.alibaba.csp</groupId>
18   <artifactId>sentinel-spring-cloud-gateway-adapter</artifactId>
19  </dependency>
20  <dependency>
21   <groupId>com.alibaba.csp</groupId>
22   <artifactId>sentinel-transport-simple-http</artifactId>
23  </dependency>
```

增加配置类

```
1  @Configuration
2  public class GatewayConfiguration {
3
4   private final List<ViewResolver> viewResolvers;
5   private final ServerCodecConfigurer serverCodecConfigurer;
6
7   public GatewayConfiguration(ObjectProvider<List<ViewResolver>> viewResol
versProvider,
8   ServerCodecConfigurer serverCodecConfigurer) {
9   this.viewResolvers = viewResolversProvider.getIfAvailable(Collections::e
mptyList);
10   this.serverCodecConfigurer = serverCodecConfigurer;
11   }
12
13   @Bean
14   @Order(Ordered.HIGHEST_PRECEDENCE)
15   public SentinelGatewayBlockExceptionHandler sentinelGatewayBlockExcepti
onHandler() {
16   // Register the block exception handler for Spring Cloud Gateway.
17   return new SentinelGatewayBlockExceptionHandler(viewResolvers, serverCo
decConfigurer);
```

```
    }

    @Bean
    @Order(Ordered.HIGHEST_PRECEDENCE)
    public GlobalFilter sentinelGatewayFilter() {
    return new SentinelGatewayFilter();
    }
}

```

增加yml的配置

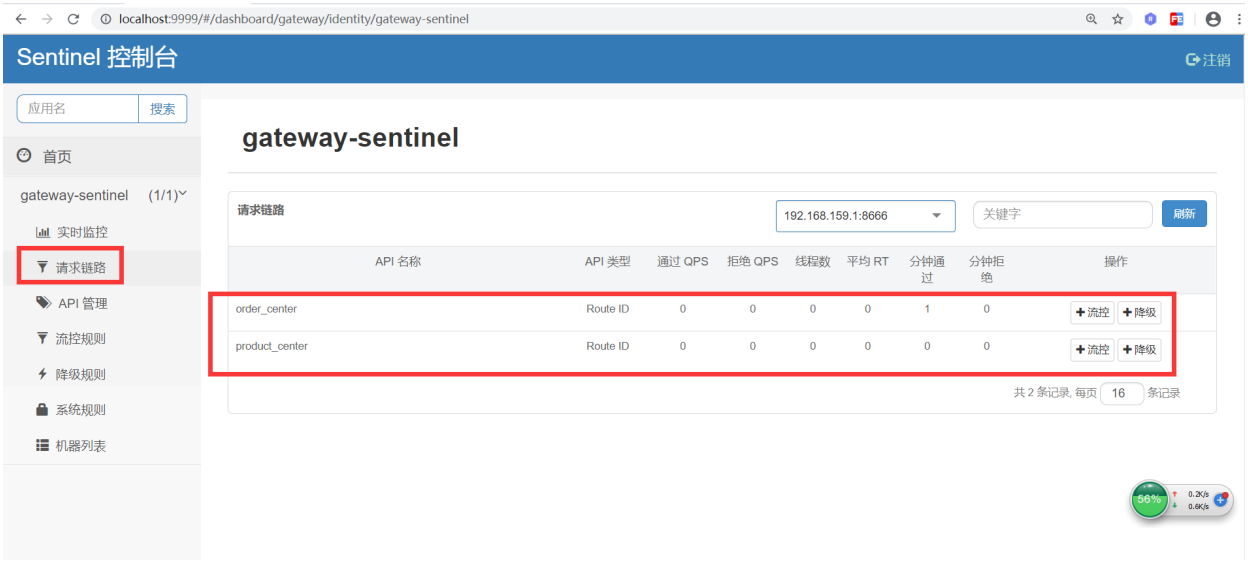
```
server:
 port: 8888
spring:
 application:
 name: gateway-sentinel
 cloud:
 gateway:
 discovery:
 locator:
  lower-case-service-id: true
  enabled: true
  routes:
  - id: product_center
  uri: lb://product-center
  predicates:
  - Path=/product/**
  - id: order_center
  uri: lb://order-center
  predicates:
  - Path=/order/**
  nacos:
  discovery:
  server-addr: localhost:8848
```

打开sentinel的控制台,由于sentinel的控制台第一次打开没有，你需要分别请求一下路径

http://localhost:8888/product/selectProductInfoById/2

http://localhost:8888/order/selectOrderInfoById/1

就会生成如下的流控节点



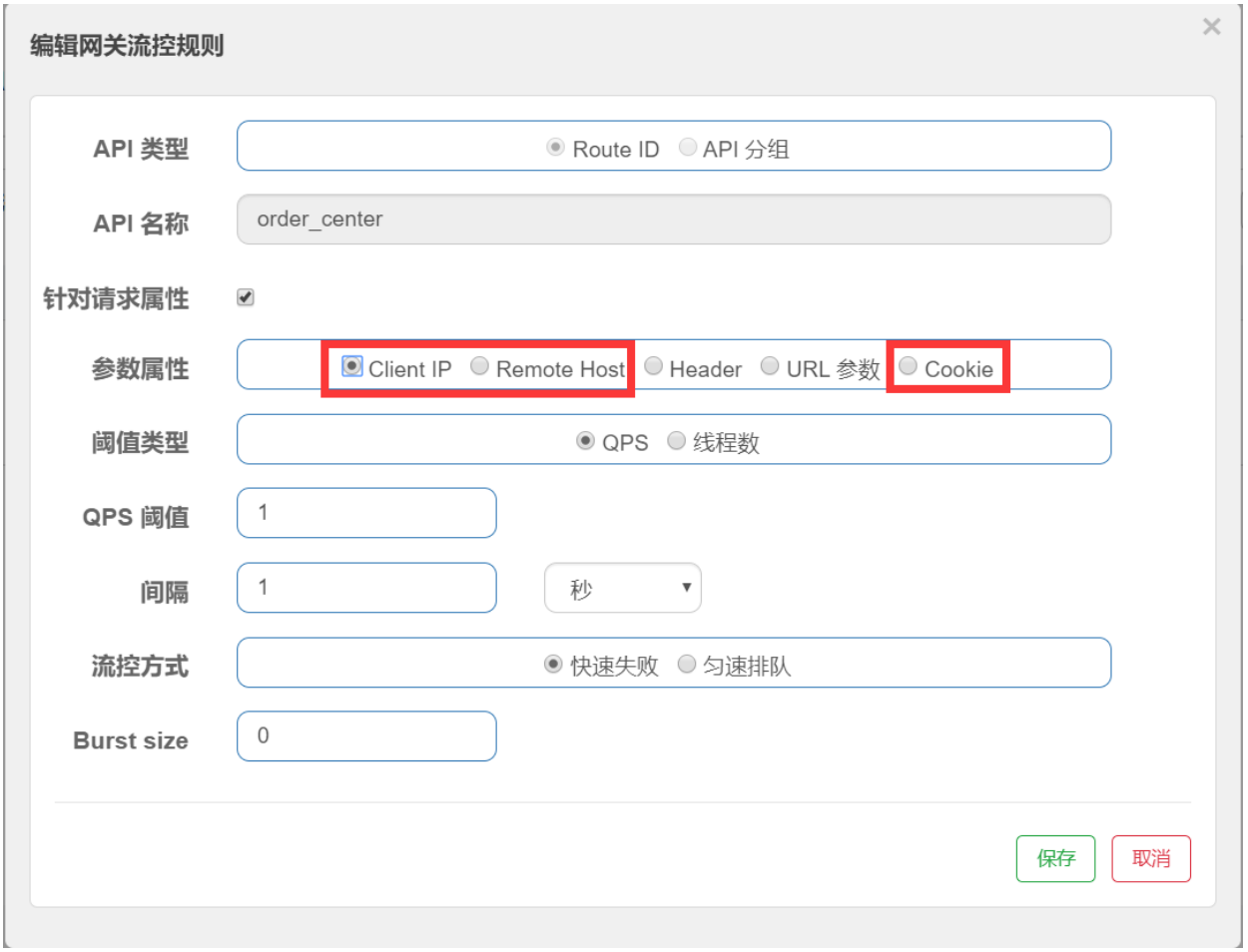
添加流控规则（如下三个 测试不出效果）

他的本意是是控制调用网关的 ip是指定的Ip进行控制

Cookie选项中意思就是每次请求中带入指定的cookie k v 就会被限流，不带就会被限制流量。（但是效果测试不出）

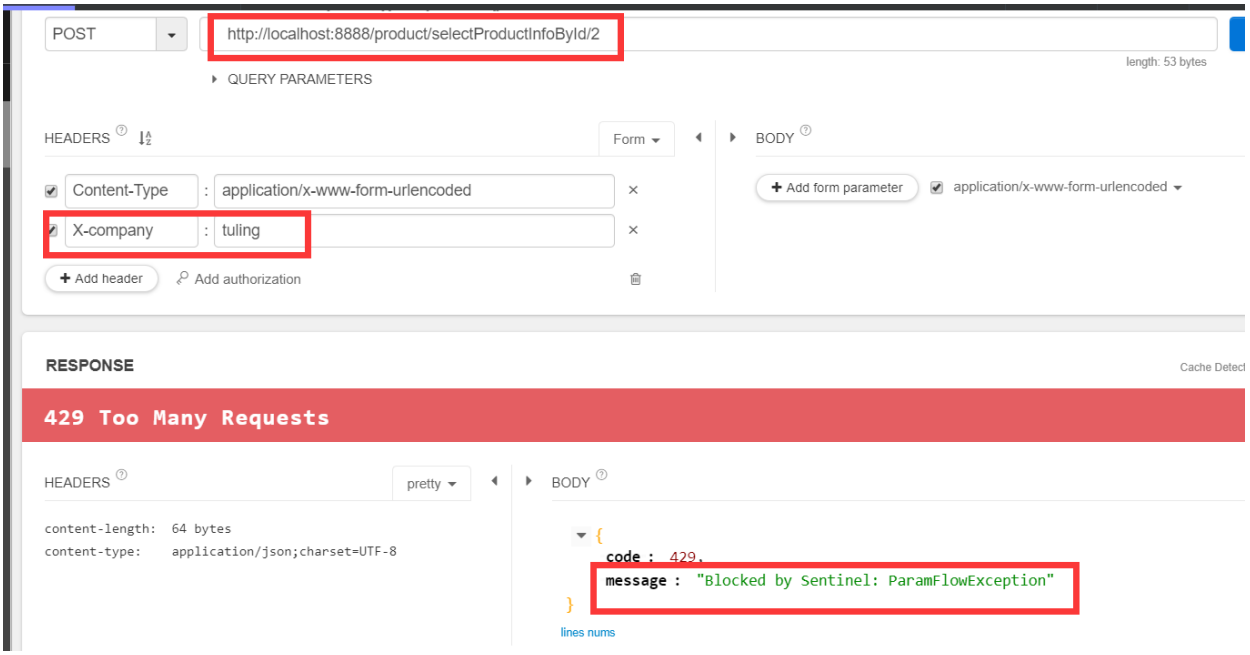
而Header模型： 指定是请求中带入的特定的header kv指就会被限流

URL参数:同理，也会针对请求参数名称进行限制流量。

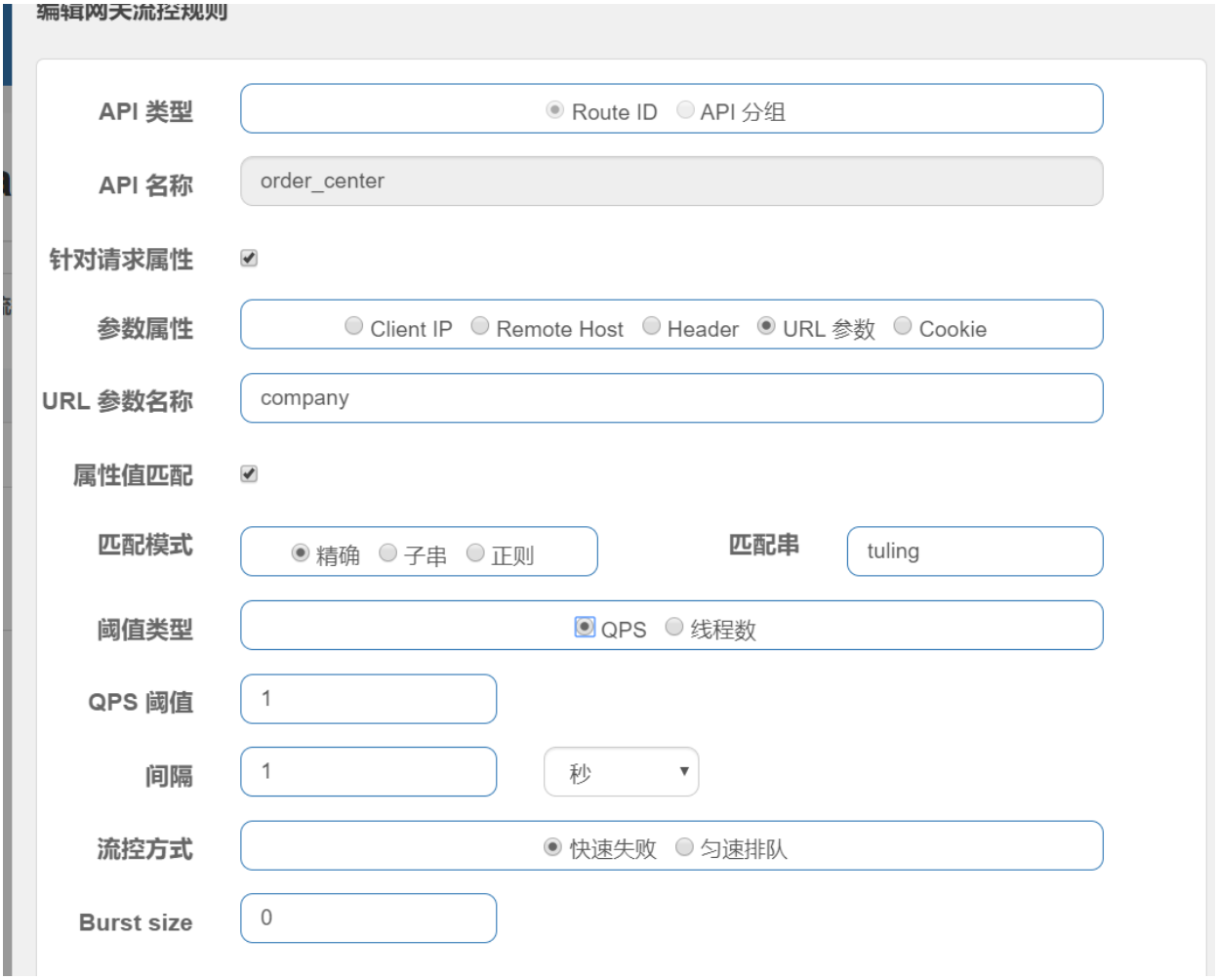


现在我们测试Header模式，如下配置

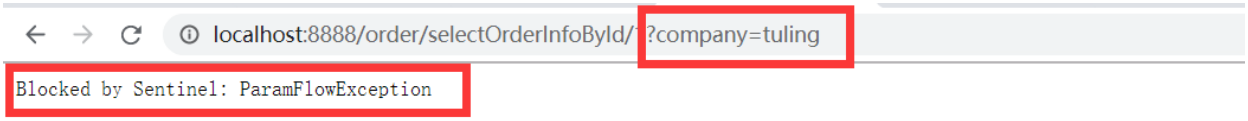
频繁的请求如下地址:



现在测试



测试:



业务场景：我们一个工程有多个请求的api，但是可能存在一种可能就是不同的

api的请求控制不一样，怎么办，那么sentienl的routeId模式流控达不到效果了。

比如:tulingvip08-ms-alibaba-gateway-product工程中有三个api

/product/selectProductInfoById/{productNo}

/product/gateWay4Header

/product/gateWay4RequestParam

若通过如下这种配置流控规则，不能做到细粒度配置，那么如何做?

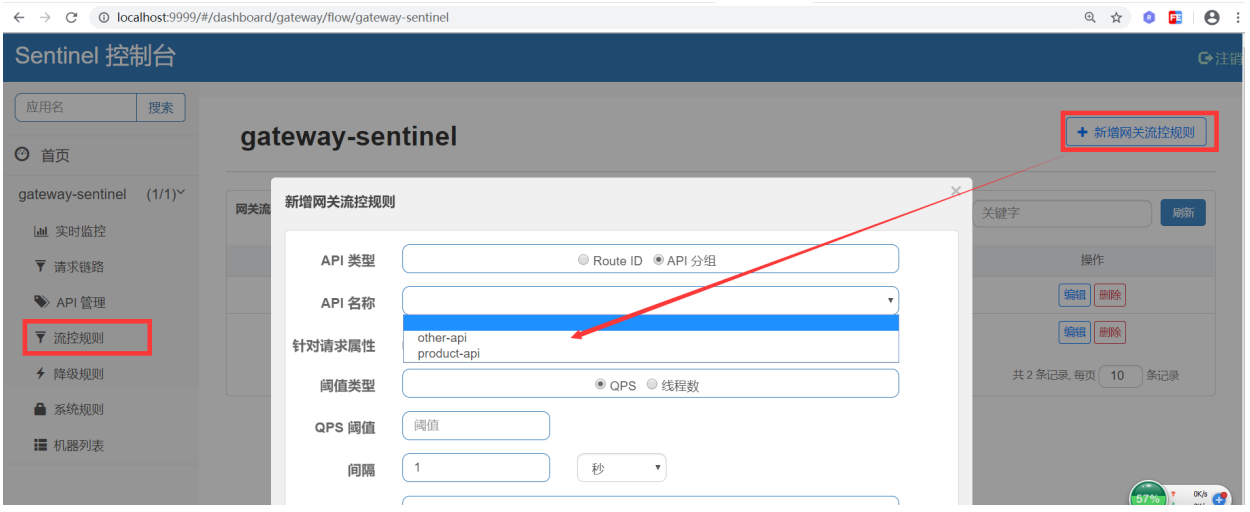
自定义APi分组,我们把 如下的api进行分组

/product/selectProductInfoById/{productNo}

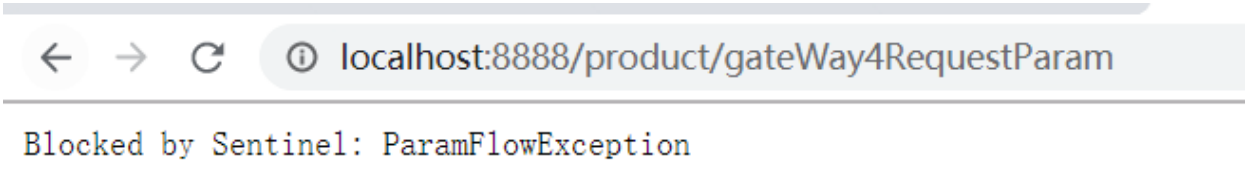
/product/gateWay4Header

/product/gateWay4RequestParam

通过API类型选择api分组可以做到细粒度配置.



GateWay+Sentienl全局异常处理。



Sentinel默认的情况下使用的是SentinelGatewayBlockExceptionHandler进行处理，我们只需要

而我们的SentinelGatewayBlockExceptionHandler底层调用了我们的

BlockRequestHandler接口的实现类DefaultBlockRequestHandler,而我们只需要自己写一个类继承父类就可以进行自定义异常处理

```
@Component
public class TulingBlockRequestHandler extends DefaultBlockRequestHandler {

 private static final String DEFAULT_BLOCK_MSG_PREFIX = "Blocked by Sentinel: ";

 //处理异常的
 @Override
 public Mono<ServerResponse> handleRequest(ServerWebExchange exchange, Throwable ex) {
 //处理html错误类型的
  if (acceptsHtml(exchange)) {
  return htmlErrorResponse(ex);
  }
  //处理Json类型的
  // JSON result by default.
  return ServerResponse.status(HttpStatus.TOO_MANY_REQUESTS)
  .contentType(MediaType.APPLICATION_JSON_UTF8)
  .body(fromObject(buildErrorResult(ex)));
  }

  private Mono<ServerResponse> htmlErrorResponse(Throwable ex) {

  return ServerResponse.status(HttpStatus.TOO_MANY_REQUESTS)
  .contentType(MediaType.TEXT_PLAIN)
  .syncBody(new String(JSON.toJSONString(buildErrorResult(ex))));
  }

  private TulingBlockRequestHandler.ErrorResult buildErrorResult(Throwable ex) {
```

```
28  if(ex instanceof ParamFlowException) {
29  return new TulingBlockRequestHandler.ErrorResult(HttpStatus.TOO_MANY_RE
QUESTS.value(),"block");
30  }else if (ex instanceof DegradeException) {
31  return new TulingBlockRequestHandler.ErrorResult(HttpStatus.TOO_MANY_RE
QUESTS.value(),"fallback");
32  }else{
33  return new
TulingBlockRequestHandler.ErrorResult(HttpStatus.BAD_GATEWAY.value(),"gatew
ay error");
34  }
35
36  }
37
38  /**
39  * Reference from {@code DefaultErrorWebExceptionHandler} of Spring
Boot.
40  */
41  private boolean acceptsHtml(ServerWebExchange exchange) {
42  try {
43  List<MediaType> acceptedMediaTypes =
exchange.getRequest().getHeaders().getAccept();
44  acceptedMediaTypes.remove(MediaType.ALL);
45  MediaType.sortBySpecificityAndQuality(acceptedMediaTypes);
46  return acceptedMediaTypes.stream()
47  .anyMatch(MediaType.TEXT_HTML::isCompatibleWith);
48  } catch (InvalidMediaTypeException ex) {
49  return false;
50  }
51  }
52
53  private static class ErrorResult {
54  private final int code;
55  private final String message;
56
57  ErrorResult(int code, String message) {
58  this.code = code;
59  this.message = message;
60  }
61
62  public int getCode() {
63  return code;
```

```
64   }
65
66   public String getMessage() {
67   return message;
68   }
69   }
70  } if (exchange.getResponse().isCommitted()) {
71   return Mono.error(ex);
72   }
73   // This exception handler only handles rejection by Sentinel.
74   if (!BlockException.isBlockException(ex)) {
75   return Mono.error(ex);
76   }
77   return handleBlockedRequest(exchange, ex)
78   .flatMap(response -> writeResponse(response, exchange));
79   }
80
81   private Mono<ServerResponse> handleBlockedRequest(ServerWebExchange exc
hange, Throwable throwable) {
82   return GatewayCallbackManager.getBlockHandler().handleRequest(exchange,
throwable);
83   }
84
85   private final Supplier<ServerResponse.Context> contextSupplier = () ->
new ServerResponse.Context() {
86   @Override
87   public List<HttpMessageWriter<?>> messageWriters() {
88   return TulingSentinelGatewayBlockExceptionHandler.this.messageWriters;
89   }
90
91   @Override
92   public List<ViewResolver> viewResolvers() {
93   return TulingSentinelGatewayBlockExceptionHandler.this.viewResolvers;
94   }
95   };
96
97
98   private Mono<Void> writeResponse(ServerResponse response, ServerWebExch
ange exchange) {
99   String reqPath = exchange.getRequest().getPath().value();
100   Map<String,Object> retMap = new HashMap<>();
101
```

```
102   ServerHttpResponse serverHttpResponse = exchange.getResponse();
103   serverHttpResponse.getHeaders().add("Content-Type", "application/json;c
harset=UTF-8");
104
105   retMap.put("msg","被限流拉");
106   retMap.put("code","-1");
107   retMap.put("reqPath",reqPath);
108   ObjectMapper objectMapper = new ObjectMapper();
109
110   byte[] datas = new byte[0];
111   try {
112   datas = objectMapper.writeValueAsString(retMap).getBytes(StandardCharse
ts.UTF_8);
113   } catch (JsonProcessingException e) {
114   e.printStackTrace();
115   }
116   DataBuffer buffer = serverHttpResponse.bufferFactory().wrap(datas);
117   return serverHttpResponse.writeWith(Mono.just(buffer));
118   }
119
120 }
```

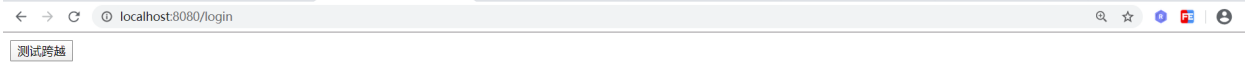
```
1
```

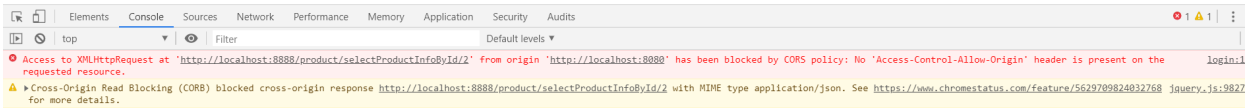
测试跨域

```
1  <script src="asserts/js/jquery.js"></script>
2  <script>
3   $(function(){
4   $('#queryUserInfo').click(function(){
5   $.ajax({
6   type: "GET",
7   url: "http://localhost:8888/product/selectProductInfoById/2",
8   dataType: "json",
9   success: function(data){
10   console.log(data);
11   alert(JSON.stringify(data));
12   }
13   });
14   });
15   });
16  </script>
```

```
17  <body>
18   <input id="queryUserInfo" value="测试跨越" name="queryUserInfo" type="bu
tton"/>
19  </body>
20  </html>
```

关闭跨域配置:





开启跨域配置: