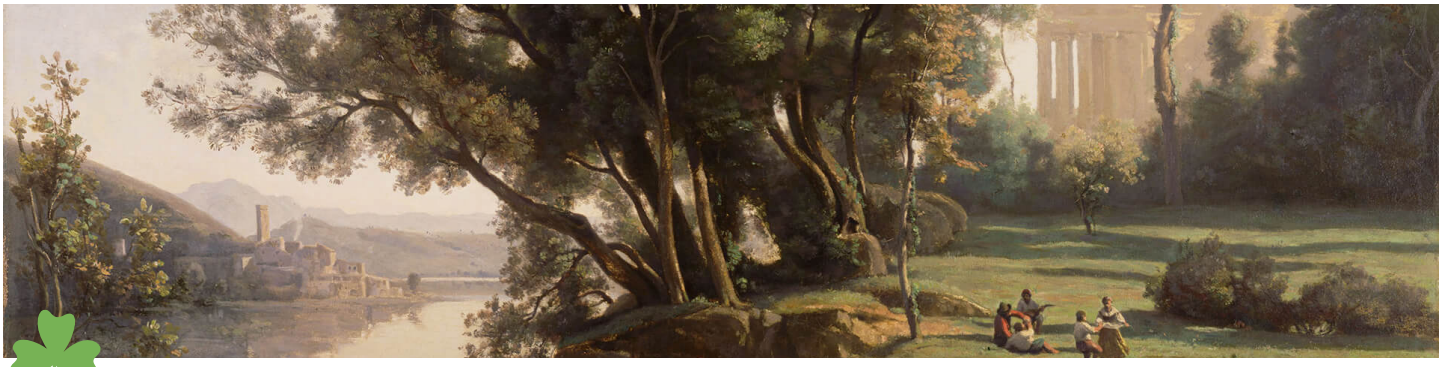
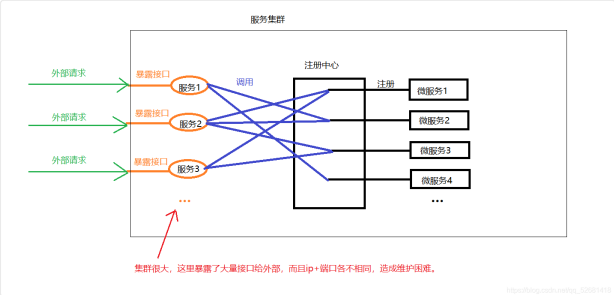


服务网关| Nginx、Zuul、GateWay

当集群中存在大量的调用者时，对外暴露的服务接口将很多，而这些服务都分布在不同的主机上，这就会出现接口IP+端口各种各样，而且当部分服务换主机或端口时，维护起来也十分困难。



情况就是上面这样，为了解决这个问题，所以使用服务网关来将这些ip端口不一致的接口代理为ip端口一致的接口。

当然，只要你想你可以代理任意接口，甚至是微服务接口，只不过使用它主要的目的还是在于解决对外提供接口不同IP过多的问题。

网关提供的功能：

- IP代理
- 身份认证
- 请求限流（网关限流）

Nginx

Nginx：自由且开源，是高性能的http服务器的反向代理服务器。安装使用

- 优点：使用Nginx的反向代理和负载均衡可实现对api服务器的负载均衡及高可用。
- 缺点：自注册的问题和网关本身的扩展性。

1.服务 IP 代理

nginx可以进行代理，实现网关功能，只需要在conf/nginx.conf中server下添加

```nginx
#路由到图片服务(后面/不能省)，img为映射路径
location /img {
  proxy_pass http://127.0.0.1:9090/;
}
```

此时就将原路径的IP:Port 变成了Nginx的IP:Nginx的端口/img 原本的api直接接在img后面即可。/api变成/img/api

Zuul

- 优点：Netflix公司开源、java开发，易于二次开发。
- 缺点：需要运行在web容器，如tomcat，缺乏管控、无法动态配置、依赖组件多，http请求依赖web容器，性能不如nginx。实际上是同步的Servlet，请求量过大时可能会阻塞，不支持websocket。

几个概念：

- 动态路由：动态将请求路由到不同后端集群
- 压力测试：逐渐增加指向集群的流量，以了解性能
- 负载分配：为每一种负载类型分配对应容量，并弃用超出限定值的请求
- 静态响应处理：边缘位置进行响应，避免转发到内部集群
- 权限认证：识别每一个资源的验证要求，并拒绝那些不符的请求。Spring Cloud对Zuul进行 了整合和增强。

创建 Zuul 网关服务

创建一个新模块作为Zuul网关，并引入依赖：

XML

```xml
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
  <version>2.1.0.RELEASE</version>
</dependency>
```

为网关服务添加配置：

YAML

```yaml
server:
  port: 7001 #服务端口
spring:
  application:
    name: api-zuul #指定服务名
```

使用注解@EnableZuulProxy启用Zuul服务：

Java

```java
@SpringBootApplication
@EnableZuulProxy // 开启Zuul的网关功能
public class ZuulApplication {

    public static void main(String[] args) {
        SpringApplication.run(ZuulApplication.class, args);
    }

}
```

服务 IP 代理

普通路由：根据请求的url将请求分配到对应的微服务中进行处理。

普通路由只需在配置文件中设置即可：

YAML

```yaml
#路由基本配置
zuul:
    routes:
        img-service:                    # 这里是路由id，随意写
            path: /img1/**              # 这里是映射路径
            url: http://127.0.0.1:9002 # 映射路径对应的实际url地址
            sensitiveHeaders:           #默认zuul会屏蔽cookie，cookie不会传到下游服务，这里设置为空则取消默认的黑名单，如果设置了具体的头信息则不会传到
```

访问：路由ip+路由端口+/img/+请求方法

这种方式需要为每一个服务进行配置，服务较多时，会有大量IP+端口，配置起来是很麻烦的。

动态路由：根据服务名从注册中心获取服务的IP+端口。

> 以Eureka为例，如果想从Eureka获取服务信息，必须引入依赖Eureka-Client：

XML

```xml
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

配置Eureka地址：

YAML

```yaml
#eureka配置
eureka:
  client:
    service-url:                                #配置注册中心地址
      defaultZone: http://localhost:8081/eureka/ #8081为Eureka端口
  instance:
    prefer-ip-address: true                     #使用ip地址注册(可选)
```

上面只是为了能够从Eureka获取服务信息而做的准备，而使用路由，其实也只是修改配置即可：

YAML

```yaml
#路由配置
zuul:
  routes:
    img-service:                    # 这里是路由id，随意写
        path: /img1/**              # 这里是映射路径
        #url: http://127.0.0.1:9091  # 映射路径对应的实际url地址
        serviceId: img-service      #配置转发的微服务名称
        sensitiveHeaders:
```

IP+端口都变成了服务名，可是每个服务都需要配置好几行。

简化路由：使配置变得更简短。

YAML

```yaml
#路由id=服务id时可简化
zuul:
  routes:
    img-service: /img1/**   #key为路由id，和服务id一致
                            #映射的url有默认值/img-service/**，所以有2个访问链接
```

可简化的前提是：路由id=服务id。

过滤器

过滤器用来对请求进行过滤，可以限流、权限认证、记录日志等。其使用时机有4种：

- 请求被路由前（PRE）：实现身份验证，在集群中选择微服务、记录调试信息等。
- 路由请求时（ROUTING）：构建发送给服务的请求，并使用Apache HttpClient或Netfilx Ribbon请求微服务。
- 路由到服务后（POST）：可为响应添加标准的HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端等。
- 发生错误时（ERROR）：发生错误时触发。

自定义过滤器：继承ZuulFilter或实现IZuulFilter接口。

通过继承ZuulFilter来自定义过滤器：

Java

```java
@Component//注册为bean
public class LoginFilter extends ZuulFilter {

    @Override    //过滤器类型：pre、routing、post、error
    public String filterType() {
        return "pre";
    }

    @Override    //过滤器执行序号（第几个执行）
    public int filterOrder() {
        return 1;
    }

    @Override    //当前过滤器是否开启（可根据参数改变值）
    public boolean shouldFilter() {
        return true;
    }

    @Override     //过滤器中执行的业务逻辑
    public Object run() throws ZuulException {
        System.out.println("执行了过滤器");
        return null;
    }

}
```

身份认证：Zuul 可以通过 RequestContext 的上下文对象可以获取 request 对象，通过此对象获取 token 参数就行了。

```java
public Object run() throws ZuulException {

    RequestContext ctx=RequestContext.getCurrentContext();    //获取RequestContext
    HttpServletRequest request=ctx.getRequest();              //获取HttpServletRequest
    String token=request.getParameter("login_token");         //获取请求参数token
    //判断token是否合法
    if ( !token.equals("DFHTSSGESGEEGSEG") ) {
        ctx.setSendZuulResponse(false);                       //拦截请求
        ctx.setResponseStatusCode(HttpStatus.SC_UNAUTHORIZED);//返回错误状态码401
    }
    //返回null，继续执行
    return null;
}
```

GateWay

SpringCloud 提供的网关服务，比较好，实际作用和 zuul 差不多，性能是 zuul 的 1.6 倍。

三个概念：

- 路由（route）：路由是网关最基础的部分，路由信息由一个 ID、一个目的 URL、一组断言工厂和一组 Filter 组成。如果断言为真，则说明请求 URL 和配置的路由匹配。
- 断言（predicates）：允许开发者去定义匹配来自 Http Request 中的任何信息，比如请求头和参数等。
- 过滤器（filter）：一个标准的 Spring webFilter，分为 Gateway Filter 和 Global Filter，可以对请求和响应进行过滤处理。

创建 GateWay 网关服务

引入项目依赖：注意 SpringCloud Gateway 使用的 web 框架为 webflux，和 SpringMVC 不兼容。

```xml
<!--内部是通过netty+webflux实现的，而webflux实现和springMVC冲突-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

服务 IP 代理

普通路由：用网关 ip、端口代理服务 ip、端口。只需如下配置即可：

```yaml
spring:
  #配置路由
  cloud:
    gateway:
      routes:
      #路由id、目标服务的url、断言（判断条件，可以有多条）
      - id: img-service
        uri: http://127.0.0.1:9093
        predicates:
        - Path=/img/**   #此处判断链接开头是否是/img/，是则通
        #其它几种断言匹配方式
        # - After=xxx                                #断言后匹配
        # - Before=xxx                               #断言前匹配
        # - Between=xxx,xxx                          #断言间匹配
        # - Cookie=chocolate, ch.p                   #Cookie匹配
        # - Header=X-Request-Id, \d+                 #Header匹配
        # - Host=**.somehost.org,**.anotherhost.org  #Host匹配
        # - Method=GET                               #请求方式匹配
        # - Path=/foo/{segment},/bar/{segment}       #请求路径匹配
        # - Query=baz 或 Query=foo,ba.               #请求参数匹配
        # - RemoteAddr=192.168.1.1/24                #IP地址范围匹配
```

与 zuul 不同的是，访问时不需要添加/img/

启动时会报错：Spring MVC found on classpat...
这是因为 gateway 与 springMVC 冲突所致，springMVC 在父项目 spring-boot-starter-web 包中，所以要将其移入模块中（也可以在运行其它服务后，将此包注释后再运行 gateway 服务）。

动态路由：根据服务名去注册中心获取

根据服务调用配置好注册中心后，GateWay 网关配置动态路由如下：

YAML
```yaml
spring:
  #配置路由
  cloud:
    gateway:
      routes:
      #路由id、目标服务的url、断言（判断条件，可以有多条）
      - id: img-service
        #uri: http://127.0.0.1:9093
        uri: lb://img4-service  #lb://注册中心中的服务名
        predicates:
        - Path=/img/**          #此处判断链接开头是否是/img/，是则通
```

可以发现，服务 ip、端口变成了服务名。

简化路由：简化路由配置。

YAML
```yaml
spring:
  cloud:    #配置路由
    gateway:
      discovery:
          locator:
            enabled: true               #开启根据服务名称转发（Eureka上所有服务都被代理）
            lower-case-service-id: true #服务名称小写显示

```

访问路径： http://网关 IP:网卡端口/服务名/接口路径

过滤器

GateWay 的过滤器使用时机有 2 种：

- 请求被路由前（PRE）：实现身份验证，在集群中选择微服务、记录调试信息等。
- 路由到服务后（POST）：可为响应添加标准的 HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端等。

如果你不希望访问路径开始部分为请求链接开始部分，而是自定义的，可以通过过滤器实现，此时功能和 zuul 类似。

YAML
```yaml
routes:
   - id: img4-service                #路由id
   #uri: http://127.0.0.1:9093
   uri: lb://img4-service  #lb:     #要路由的服务
   predicates:                      #断言（判断条件，可以有多条）
      #- Path=/img/**
      - Path=/img4-service/**
      filters:                      #过滤器
          #路径重写过滤器：http://127.0.0.1:7002/img4-service/img/findimg/1
          - RewritePath=/img4-service/(?<segment>.*), /$\{segment}
          # yml文档中 $ 要写成 $\


```

实际上是通过过滤器 + 表达式来重写请求路径。

局部过滤器：对单个路由或单个分组进行过滤（**GatewayFilter** ）。

GateWay 有很多过滤器工厂，它们都有实现类，名称以 工厂 +GatewayFilterFactory ：

请求过滤：

- AddRequestHeader：为请求添加 Header（提供参数：Header 的名称及值）。
- RemoveRequestHeader：为请求删除 Header（提供参数：Header 的名称）。

- AddRequestParameter：为请求添加参数（提供参数：添加的参数名、添加的参数值）。
- PrefixPath：为请求添加前缀（提供参数：前缀路径 ）。
- PreserveHostHeader：为请求添加 preserveHostHeader=true，是否要发送原始的 Host。
- RequestRateLimiter：使用令牌桶算法进行请求限流（提供参数：keyResolver、rateLimiter、statusCode、denyEmptyKey、emptyKeyStatus ）。
- RedirectTo：重定向请求（提供参数：http 状态码、重定向的 url ）。
- RewritePath：重写请求路径（提供参数：原始及重写的路径正则表达式 ）。
- SetPath：修改请求路径（提供参数：新路径 ）。
- RemoveHopByHopHeadersFilter：为请求删除 IETF 组织规定的 Header，默认启用，可通过配置来指定删除。
- SaveSession：转发请求前，保存 WebSession。
- ModifyRequestBody：转发请求前，修改请求体（提供参数：新请求体）。
- StripPrefix：截断请求路径（提供参数：截断数量）。
- RequestSize：设置请求包容量上限（提供参数：字节数，默认 5M）。
- Hystrix：为路由引入 Hystrix 的断路器保护（提供参数：HystrixCommand 名称）。
- FallbackHeaders：为 fallbackUri 的请求头中添加具体的异常信息（提供参数：Header 的名称）。

响应过滤：

- AddResponseHeader：为响应添加 Header（提供参数：Header 的名称及值）。
- secureHeaders：为响应添加安全 Header（提供参数：Header 的名称及值）。
- RemoveResponseHeader：为响应删除 Header（提供参数：Header 的名称）。
- DedupeResponseHeader：为响应 Header 去重（提供参数：Header 的名称及去重策略）。
- RewriteResponseHeader：重写响应中的 Header（提供参数：Header 的名称、值的正则表达式、重写后的值 ）。
- SetResponseHeader：修改响应 Header（提供参数：Header 的名称、新值）。
- SetStatus：修改响应状态码（提供参数：新状态码）。
- Retry：重试响应（提供参数：retries、statuses、methods、series ）。
- ModifyResponseBody：修改响应体（提供参数：新响应体）。

全局过滤器：对所有路由过滤（GlobalFilter 接口）。

可以实现对权限的统一校验，安全性验证等功能，比较常用。

Java

```java
/**自定义一个全局过滤器：
 *  实现 GlobalFilter、Ordered接口
 * **/
@Component
public class LoginFilter implements GlobalFilter, Ordered {

    @Override //过滤器中执行的业务逻辑
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        System.out.println("执行了自定义的全局过滤器");
        return chain.filter(exchange);//继续向下执行
    }

    @Override //指定过滤器执行顺序
    public int getOrder() {
        return 0;
    }
}
```

自定义过滤器：

ServerWebExchange 和 Zuul 里的 RequestContext 类似，可以获取 request、response 对象

Java

```java
@Component
public class AuthorizeFilter implements GlobalFilter, Ordered {

    @Override
```

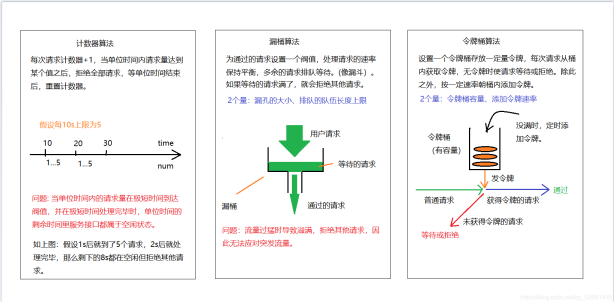
```
    public Mono<Void> filter(ServerWebExchange ec, GatewayFilterChain chain) {
        //获取请求参数token
        String token = ec.getRequest().getQueryParams().getFirst("token");
        if ( !token.equlas("AFDFAFASAFDGGD"))) {//验证token
            ec.getResponse().setStatusCode( HttpStatus.UNAUTHORIZED );
            return ec.getResponse().setComplete();
        }
        return chain.filter(ec);
    }

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

    //忽略部分url请求
    //String url = ec.getRequest().getURI().getPath();
    //if(url.indexOf("/login") >= 0){
    // return chain.filter(exchange);
     // }

}

```

网关限流

GateWay官方提供了基于令牌桶的限流支持，使用过滤器工厂RequestRateLimiterGatewayFilterFactory实现，通过Redis和lua脚本结合实现流量控制。

网关限流算法：

- 计数器算法
- 漏桶算法
- 令牌桶算法



基于Filter的限流

需要使用Redis，因此需要下载Redis，并运行redis-server.exe、redis-xli.exe，然后输入monitor开启监控。

网关服务引入基于Reactive的Redis依赖：

XML

```
<!--监控依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<!--redis依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>
```

为网关服务添加配置：

YAML

```
spring:
  redis:            #Redis配置
    host: localhost
    port: 6379
    database: 0
  cloud:
    gateway:        #GateWay配置
      routes:
```

```yaml
        - id: img4-service
          uri: lb://img4-service
          predicates:
          - Path=/img4-service/**
          filters:
            - RewritePath=/img4-service/(?<segment>.*), /$\{segment}
            - name: RequestRateLimiter #使用限流过滤器
              args:
                key-resolver: '#{@pathKeyResolver}'     # 使用SpEL从容器中获取对象
                redis-rate-limiter.replenishRate: 1     # 令牌桶每秒填充平均速率
                redis-rate-limiter.burstCapacity: 3     # 令牌桶的总容量
```

配置 redis 中的 KeySesolver：

Java

```java
@Component    //请求限流方法
public class KeyResolverConfiguration {

    @Bean    //对请求路径限流
    public KeyResolver pathKeResolver(){
        return new KeyResolver() {//自定义的KeyResolver，即实现KeyResolver接口。
            @Override    //参数为springcloud的上下文参数
            public Mono<String> resolve(ServerWebExchange ec) {
                return Mono.just(ec.getRequest().getPath().toString());
            }
        };
    }



    @Bean    //对请求IP限流
    public KeyResolver userKeyResolver(){
        return ec-> Mono.just(ec.getRequest().getQueryParams().getFirst("userId"));
    }

    @Bean    //对请求参数限流
    public KeyResolver ipKeyResolver(){
        return ec-> Mono.just(ec.getRequest().getHeaders().getFirst("X-Forwarded-For"));
    }

}
```

基于 sentinel 的限流

注入 SentinelGatewayFilter、SentinelGatewayBlockExceptionHandler 实例即可。

@PostConstruct 定义初始化的加载方法，用于指定资源的限流规则。

Java

```java
@Configuration
public class GatewayConfiguration {
    private final List<ViewResolver> vr;
    private final ServerCodecConfigurer scc;

    //构造函数
    public GatewayConfiguration(ObjectProvider<List<ViewResolver>> vrs,ServerCodecConfigurer scc) {
        this.vr = vrs.getIfAvailable(Collections::emptyList);
        this.scc = scc;
    }

    @Bean    //配置限流的异常处理器:SentinelGatewayBlockExceptionHandler
    @Order(Ordered.HIGHEST_PRECEDENCE)
    public SentinelGatewayBlockExceptionHandler sentinelGatewayBlockExceptionHandler() {
        return new SentinelGatewayBlockExceptionHandler(vr,scc);
    }

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

     /**
       此处添加限流规则的方法

      **/
```

```
}
```

添加配置：

```yaml
server:
  port: 7002 #服务端口
spring:
  application:
    name: api-gateway #指定服务名
  redis:                #配置Redis
    host: localhost
    port: 6379
    database: 0
  cloud:
    gateway:            #配置GateWay
      routes:
        - id: img4-service
          uri: lb://img4-service
          predicates:
              - Path=/img4-service/**
              filters:
                  - RewritePath=/product-service/(?<segment>.*), /$\{segment}

```

限流规则

全局限流、局部限流：

```java
    //全局限流
    @PostConstruct
    public void initGatewayRules() {
        Set<GatewayFlowRule> rules = new HashSet<>();
        //添加：资源名称、 限流阈值、统计时间窗口（单位是秒，默认是 1 秒）
        rules.add(new GatewayFlowRule("img4-service").setCount(1).setIntervalSec(1));
        GatewayRuleManager.loadRules(rules);
    }

    //局部限流（参数限流）
    @PostConstruct
    public void initGatewayRules() {
        Set<GatewayFlowRule> rules = new HashSet<>();
        //从url获取参数，参数名为id
        GatewayParamFlowItem gpf= new GatewayParamFlowItem()
          .setParseStrategy(SentinelGatewayConstants.PARAM_PARSE_STRATEGY_URL_PARAM)
          .setFieldName("id");
        //添加：资源名称、 限流阈值、统计时间窗口（单位是秒，默认是 1 秒）
        rules.add(new GatewayFlowRule("img4-service").setCount(1).setIntervalSec(1).setParamItem(gpf));
        GatewayRuleManager.loadRules(rules);
     }
```

分组限流：

```java
//自定义API限流分组
@PostConstruct
private void initCustomizedApis() {

    Set<ApiDefinition> defs = new HashSet<>();//API限流分组集合

    //定义API限流分组1：路径限流（路径头部匹配）
    ApiDefinition api1 = new ApiDefinition("img_api")
        .setPredicateItems(new HashSet<ApiPredicateItem>() {{
            add(new ApiPathPredicateItem().setPattern("/img4-service/img/**").setMatchStrategy(SentinelGatewayConstants.URL_MATCH_STRATI
            );
        }});

    //定义API限流分组2：路径限流（路径完全匹配）
    ApiDefinition api2 = new ApiDefinition("user_api")
        .setPredicateItems( new HashSet<ApiPredicateItem>() {{
            add(new ApiPathPredicateItem().setPattern("/user-service/user"));
         }});

    defs.add(api1);
    defs.add(api2);
```

```
    //将所有自定义分组加入分组管理器
    GatewayApiDefinitionManager.loadApiDefinitions(defs);

}


@PostConstruct    //限流规则
public void initGatewayRules() {
    Set<GatewayFlowRule> rules = new HashSet<>();
  //小组限流设置
    //添加：小组名称、 限流阈值、统计时间窗口（单位是秒，默认是 1 秒）
    rules.add(new GatewayFlowRule("img_api").setCount(1).setIntervalSec(1));
    rules.add(new GatewayFlowRule("user_api").setCount(1).setIntervalSec(1));
  GatewayRuleManager.loadRules(rules);
}

```

限流-自定义异常处理

触发限流时默认显示Blocked by Sentinel: FlowException，这看起来不是很友好。

可以设置一个限流触发异常处理器，限流异常处理接口为**BlockRequestHandler** 。

自定义异常处理：

Java

```
/**自定义的限流异常处理：
 *      将GatewayCallbackManager里的BlockHandler改为自定义的BlockRequestHandler。
 **/
@PostConstruct//服务器加载Servlet的时候运行,且只运行一次
public void myBlockHandlers(){

    //自定义返回处理器（限流触发异常返回处理器）
    BlockRequestHandler brh=new BlockRequestHandler() {
        @Override
        public Mono<ServerResponse> handleRequest(ServerWebExchange ec,Throwable throwable) {
            Map map=new HashMap<>();
            map.put("code",001);
            map.put("msessage","对不起，此时比较拥堵");
            return ServerResponse.status(HttpStatus.OK).contentType(MediaType.APPLICATION_JSON).body(BodyInserters.fromValue(map));
        }
     };
     //gateway调用返回管理器,设置为自定义的返回处理器（修改了默认值）
     GatewayCallbackManager.setBlockHandler(brh);
}
```

网关高可用

网关集群：多个网关同时启动。客户端需要维护多个网关信息，因此客户端与网关集群之间需要添加一层nginx。

启动多个gateway服务，并编写nginx.conf，使用nginx代理gateway服务：

nginx

```
    #集群配置
  upstream gateway {
    server 127.0.0.1:7002;
    server 127.0.0.1:7003;
  }

    server {
        listen       9999;
        server_name  localhost;
    #------------------------------gateway修改
    #访问127.0.0.1时会调用
      location / {
      proxy_pass http://gateway;    #会自动选择调用上面的2个server
        }

  }
```

启动后访问：http://localhost/img4-service/img/findimg/1

只要有一个[illegible]服务还在，就还能访问