spring cloud gateway 介绍

1. 网关是怎么演化来的

2. 网关的基本功能

3.关于Spring Cloud Gateway

4.Spring Cloud Gateway 功能特征

5.predicate简介

6.GatewayFilter 工厂介绍

7.集成Hystrix的断路器功能

8.请求限流

pom.xml

application.yml

IP限流

用户限流

接口限流

spring cloud gateway 介绍

1. 网关是怎么演化来的

单体应用拆分成多个服务后，对外需要一个统一入口，解耦客户端与内部服务

注：图片来自网络

2. 网关的基本功能

网关核心功能是路由转发，因此不要有耗时操作在网关上处理，让请求快速转发到后端服务上

网关还能做统一的熔断、限流、认证、日志监控等 注：图片来自网络

可以和服务注册中心完美的整合，如：Eureka、Consul、Nacos

3.关于Spring Cloud Gateway

在SpringCloud微服务体系中，有个很重要的组件就是网关，在1.x版本中都是采用的Zuul网关；但在2.x版本中，zuul的升级一直跳票，SpringCloud最后自己研发了一个网关替代Zuul，那就是SpringCloud Gateway

网上很多地方都说Zuul是阻塞的，Gateway是非阻塞的，这么说是不严谨的，准确的讲Zuul1.x是阻塞的，而在2.x的版本中，Zuul也是基于Netty，也是非阻塞的，如果一定要说性能，其实这个真没多大差距。

而官方出过一个测试项目，创建了一个benchmark的测试项目：spring-cloud-gateway-bench，其中对比了：

Spring Cloud Gateway

Zuul1.x

Linkerd

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

还有一点就是Gateway是基于WebFlux的。这里引出了WebFlux名词，那什么是WebFlux？

WebFlux 介绍

注：图片来自网络

左侧是传统的基于Servlet的Spring Web MVC框架，

传统的Web框架，比如说：struts2，springmvc等都是基于Servlet API与Servlet容器基础之上运行的，在Servlet3.1之后才有了异步非阻塞的支持。

右侧是5.0版本新引入的基于Reactive Streams的Spring WebFlux框架，从上到下依次是Router Functions，WebFlux，Reactive Streams三个新组件。

Router Functions: 对标@Controller，@RequestMapping等标准的Spring MVC注解，提供一套函数式风格的API，用于创建Router，Handler和Filter。

WebFlux: 核心组件，协调上下游各个组件提供响应式编程支持。

Reactive Streams: 一种支持背压（Backpressure）的异步数据流处理标准，主流实现有RxJava和Reactor，Spring WebFlux默认集成的是Reactor。

在Web容器的选择上，Spring WebFlux既支持像Tomcat，Jetty这样的的传统容器（前提是支持Servlet 3.1 Non-Blocking IO API），又支持像Netty，Undertow那样的异步容器。不管是何种容器，Spring WebFlux都会将其输入输出流适配成Flux<DataBuffer>格式，以便进行统一处理。

值得一提的是，除了新的Router Functions接口，Spring WebFlux同时支持使用老的Spring MVC注解声明Reactive Controller。和传统的MVC Controller不同，Reactive Controller操作的是非阻塞的ServerHttpRequest和ServerHttpResponse，而不再是Spring MVC里的HttpServletRequest和HttpServletResponse。

根据官方的说法，webflux主要在如下两方面体现出独有的优势：

1）非阻塞式

其实在servlet3.1提供了非阻塞的API，WebFlux提供了一种比其更完美的解决方案。使用非阻塞的方式可以利用较小的线程或硬件资源来处理并发进而提高其可伸缩性

2) 函数式编程端点

老生常谈的编程方式了，Spring5必须让你使用java8，那么函数式编程就是java8重要的特点之一，而WebFlux支持函数式编程来定义路由端点处理请求。

IO模式和IO多路复用（阻塞IO、非阻塞IO、同步IO、异步IO等概念）这里就不在阐述，因为不在本分享范围之内!

4.Spring Cloud Gateway 功能特征

基于Spring Framework 5, Project Reactor 和 Spring Boot 2.0 进行构建；

动态路由：能够匹配任何请求属性；

集成 Spring Cloud 服务发现功能；

可以对路由指定 Predicate（断言）和 Filter（过滤器）；

易于编写的 Predicate（断言）和 Filter（过滤器）；

集成Hystrix的断路器功能；

请求限流功能；

支持路径重写。

注：图片来自网络

上图中是核心的流程图，最主要的就是Route、Predicates 和 Filters 作用于特定路由。

1）Route：\*\*路由是网关的基本构件\*\*。它由ID、目标URI、谓词集合和过滤器集合定义。如果聚合谓词为真，则匹配路由。

2）Predicate：\*\*参照Java8的新特性Predicate\*\*。这允许开发人员匹配HTTP请求中的任何内容，比如头或参数。

3）Filter：可以在发送下游请求之前或之后修改请求和响应。

我们为什么选择Gateway？

一方面因为Zuul已经进入了维护阶段，而且Gateway是SpringCloud团队研发的，是亲儿子产品，值得信赖。而且很多功能Zuul都没有；用起来也非常的简单便捷。

Gateway是基于异步非阻塞模型上进行开发的，性能方面不需要担心。虽然Netflix 早就发布了最新的 Zuul 2.x，但 Spring Cloud 貌似没有整合计划。而且Netflix相关组件都宣布进入维护期；不知前景如何？

多方面综合考虑Gateway是很理想的网关选择。

Spring Cloud Gateway 工作原理

注：该图片来自官网

客户端向 Spring Cloud Gateway 发出请求。然后在 Gateway Handler Mapping 中找到与请求相匹配的路由，将其发送到 Gateway Web Handler。Handler 再通过指 定的过滤器链来将请求发送到我们实际的服务执行业务逻辑，然后返回。过滤器之间用虚线分开是因为过滤器可能会在发送代理请求之前（“pre”）或之后（“post”）执行业务逻辑。

Filter在“pre”类型的过滤器可以做参数校验、权限校验、流量监控、日志输出、协议转换等，

在“post”类型的过滤器中可以做响应内容、响应头的修改，日志的输出，流量监控等有着非常重要的作用。

核心逻辑就是路由转发，执行过滤器链。

在上面的处理过程中，有一个重要的点就是讲请求和路由进行匹配，这时候就需要用到predicate，它是决定了一个请求走哪一个路由。

5.predicate简介

Predicate来自于java8的接口。Predicate接受一个输入参数，返回一个布尔值结果。该接口包含多种默认方法来将Predicate组合成其他复杂的逻辑（比如：与，或，非）。可以用于接口请求参数校验、判断新老数据是否有变化需要进行更新操作。add--与、or--或、negate--非。

Spring Cloud Gateway内置了许多Predict,这些Predict的源码在org.springframework.cloud.gateway.handler.predicate包中，有兴趣可以阅读一下。现在列举各种Predicate如下图：

注：图片来自网络

在上图中，有很多类型的Predicate,比如说时间类型的Predicated（AfterRoutePredicateFactory BeforeRoutePredicateFactory BetweenRoutePredicateFactory），当只有满足特定时间要求的请求会进入到此predicate中，并交由router处理；cookie类型的CookieRoutePredicateFactory，指定的cookie满足正则匹配，才会进入此router;以及host、method、path、querparam、remoteaddr类型的predicate，每一种predicate都会对当前的客户端请求进行判断，是否满足当前的要求，如果满足则交给当前请求处理。如果有很多个Predicate，并且一个请求满足多个Predicate，则按照配置的顺序第一个生效。

1. After Route Predicate Factory

After Route Predicate Factory使用的是时间作为匹配规则，只要当前时间大于设定时间，路由才会匹配请求。 application.yml：

spring:

cloud:

gateway:

routes:

- id: after\_route

uri: http://www.google.com

predicates:

- After=2018-12-25T14:33:47.789+08:00

这个路由规则会在东8区的2018-12-25 14:33:47后，将请求都转跳到google。

2. Before Route Predicate Factory

Before Route Predicate Factory也是使用时间作为匹配规则，只要当前时间小于设定时间，路由才会匹配请求。 application.yml：

spring:

cloud:

gateway:

routes:

- id: before\_route

uri: http://www.google.com

predicates:

- Before=2018-12-25T14:33:47.789+08:00

这个路由规则会在东8区的2018-12-25 14:33:47前，将请求都转跳到google。

3. Between Route Predicate Factory

Between Route Predicate Factory也是使用两个时间作为匹配规则，只要当前时间大于第一个设定时间，并小于第二个设定时间，路由才会匹配请求。

application.yml：

spring:

cloud:

gateway:

routes:

- id: between\_route

uri: http://www.google.com

predicates:

- Between=2018-12-25T14:33:47.789+08:00, 2018-12-26T14:33:47.789+08:00

这个路由规则会在东8区的2018-12-25 14:33:47到2018-12-26 14:33:47之间，将请求都转跳到google。

4. Cookie Route Predicate Factory

Cookie Route Predicate Factory使用的是cookie名字和正则表达式的value作为两个输入参数，请求的cookie需要匹配cookie名和符合其中value的正则。 application.yml：

spring:

cloud:

gateway:

routes:

- id: cookie\_route

uri: http://www.google.com

predicates:

- Cookie=cookiename, cookievalue

路由匹配请求存在cookie名为cookiename，cookie内容匹配cookievalue的，将请求转发到google。

5. Header Route Predicate Factory

Header Route Predicate Factory，与Cookie Route Predicate Factory类似，也是两个参数，一个header的name，一个是正则匹配的value。 application.yml：

spring:

cloud:

gateway:

routes:

- id: header\_route

uri: http://www.google.com

predicates:

- Header=X-Request-Id, \d+

路由匹配存在名为X-Request-Id，内容为数字的header的请求，将请求转发到google。

6. Host Route Predicate Factory

Host Route Predicate Factory使用的是host的列表作为参数，host使用Ant style匹配。 application.yml：

spring:

cloud:

gateway:

routes:

- id: host\_route

uri: http://www.google.com

predicates:

- Host=\*\*.somehost.org,\*\*.anotherhost.org

路由会匹配Host诸如：www.somehost.org 或 beta.somehost.org或www.anotherhost.org等请求。

7. Method Route Predicate Factory

Method Route Predicate Factory是通过HTTP的method来匹配路由。 application.yml：

spring:

cloud:

gateway:

routes:

- id: method\_route

uri: http://www.google.com

predicates:

- Method=GET

路由会匹配到所有GET方法的请求。

8. Path Route Predicate Factory

Path Route Predicate Factory使用的是path列表作为参数，使用Spring的PathMatcher匹配path，可以设置可选变量。 application.yml：

spring:

cloud:

gateway:

routes:

- id: host\_route

uri: http://www.google.com

predicates:

- Path=/foo/{segment},/bar/{segment}

上面路由可以匹配诸如：/foo/1 或 /foo/bar 或 /bar/baz等 其中的segment变量可以通过下面方式获取：

PathMatchInfo variables = exchange.getAttribute(URI\_TEMPLATE\_VARIABLES\_ATTRIBUTE);

Map<String, String> uriVariables = variables.getUriVariables();

String segment = uriVariables.get("segment");

在后续的GatewayFilter Factories就可以做对应的操作了。

9. Query Route Predicate Factory

Query Route Predicate Factory可以通过一个或两个参数来匹配路由，一个是查询的name，一个是查询的正则value。 application.yml：

spring:

cloud:

gateway:

routes:

- id: query\_route

uri: http://www.google.com

predicates:

- Query=baz

路由会匹配所有包含baz查询参数的请求。 application.yml：

spring:

cloud:

gateway:

routes:

- id: query\_route

uri: http://www.google.com

predicates:

- Query=foo, ba.

路由会匹配所有包含foo，并且foo的内容为诸如：bar或baz等符合ba.正则规则的请求。

10. RemoteAddr Route Predicate Factory

RemoteAddr Route Predicate Factory通过无类别域间路由(IPv4 or IPv6)列表匹配路由。 application.yml：

spring:

cloud:

gateway:

routes:

- id: remoteaddr\_route

uri: http://www.google.com

predicates:

- RemoteAddr=192.168.1.1/24

上面路由就会匹配RemoteAddr诸如192.168.1.10等请求。

10.1 Modifying the way remote addresses are resolved

RemoteAddr Route Predicate Factory默认情况下，使用的是请求的remote address。但是如果Spring Cloud Gateway是部署在其他的代理后面的，如Nginx，则Spring Cloud Gateway获取请求的remote address是其他代理的ip，而不是真实客户端的ip。

考虑到这种情况，你可以自定义获取remote address的处理器RemoteAddressResolver。当然Spring Cloud Gateway也提供了基于X-Forwarded-For请求头的XForwardedRemoteAddressResolver。 熟悉Http代理协议的，都知道X-Forwarded-For头信息做什么的，不熟悉的可以自己谷歌了解一下。

XForwardedRemoteAddressResolver提供了两个静态方法获取它的实例： XForwardedRemoteAddressResolver::trustAll得到的RemoteAddressResolver总是获取X-Forwarded-For的第一个ip地址作为remote address，这种方式就比较容易被伪装的请求欺骗，模拟请求很容易通过设置初始的X-Forwarded-For头信息，就可以欺骗到gateway。

XForwardedRemoteAddressResolver::maxTrustedIndex得到的RemoteAddressResolver则会在X-Forwarded-For信息里面，从右到左选择信任最多maxTrustedIndex个ip，因为X-Forwarded-For是越往右是越接近gateway的代理机器ip，所以是越往右的ip，信任度是越高的。 那么如果前面只是挡了一层Nginx的话，如果只需要Nginx前面客户端的ip，则maxTrustedIndex取1，就可以比较安全地获取真实客户端ip。

使用java的配置： GatewayConfig.java：

RemoteAddressResolver resolver = XForwardedRemoteAddressResolver.maxTrustedIndex(1);

...

.route("direct-route", r -> r.remoteAddr("10.1.1.1", "10.10.1.1/24") .uri("http://www.google.com")

.route("proxied-route",r -> r.remoteAddr(resolver, "10.10.1.1", "10.10.1.1/24") .uri("http://www.google.com"))

6.GatewayFilter 工厂介绍

Route filters可以通过一些方式修改HTTP请求的输入和输出，针对某些特殊的场景，Spring Cloud Gateway已经内置了很多不同功能的GatewayFilter Factories。

下面就来通过例子逐一讲解这些GatewayFilter Factories。

1. AddRequestHeader GatewayFilter Factory

AddRequestHeader GatewayFilter Factory通过配置name和value可以增加请求的header。 application.yml：

spring:

cloud:

gateway:

routes:

- id: add\_request\_header\_route

uri: http://www.google.com

filters:

- AddRequestHeader=X-Request-Foo, Bar

对匹配的请求，会额外添加X-Request-Foo:Bar的header。

2. AddRequestParameter GatewayFilter Factory

AddRequestParameter GatewayFilter Factory通过配置name和value可以增加请求的参数。 application.yml：

spring:

cloud:

gateway:

routes:

- id: add\_request\_parameter\_route

uri: http://www.google.com

filters:

- AddRequestParameter=foo, bar

对匹配的请求，会额外添加foo=bar的请求参数。

3. AddResponseHeader GatewayFilter Factory

AddResponseHeader GatewayFilter Factory通过配置name和value可以增加响应的header。 application.yml：

spring:

cloud:

gateway:

routes:

- id: add\_request\_header\_route

uri: http://www.google.com

filters:

- AddResponseHeader=X-Response-Foo, Bar

对匹配的请求，响应返回时会额外添加X-Response-Foo:Bar的header返回。

4. Hystrix GatewayFilter Factory

Hystrix是Netflix实现的断路器模式工具包，The Hystrix GatewayFilter就是将断路器使用在gateway的路由上，目的是保护你的服务避免级联故障，以及在下游失败时可以降级返回。

项目里面引入spring-cloud-starter-netflix-hystrix依赖，并提供HystrixCommand的名字，即可生效Hystrix GatewayFilter。 application.yml：

spring:

cloud:

gateway:

routes:

- id: hystrix\_route

uri: http://www.google.com

filters:

- Hystrix=myCommandName

那么剩下的过滤器，就会包装在名为myCommandName的HystrixCommand中运行。

Hystrix过滤器也是通过配置可以参数fallbackUri，来支持路由熔断后的降级处理，降级后，请求会跳过fallbackUri配置的路径，目前只支持forward:的URI协议。 application.yml：

spring:

cloud:

gateway:

routes:

- id: hystrix\_route

uri: lb://backing-service:8088

predicates:

- Path=/consumingserviceendpoint

filters:

- name: Hystrix

args:

name: fallbackcmd

fallbackUri: forward:/incaseoffailureusethis

当Hystrix降级后就会将请求转发到/incaseoffailureusethis。

整个流程其实是用fallbackUri将请求跳转到gateway内部的controller或者handler，然而也可以通过以下的方式将请求转发到外部的服务： application.yml：

spring:

cloud:

gateway:

routes:

- id: ingredients

uri: lb://ingredients

predicates:

- Path=//ingredients/\*\*

filters:

- name: Hystrix

args:

name: fetchIngredients

fallbackUri: forward:/fallback

- id: ingredients-fallback

uri: http://localhost:9994

predicates:

- Path=/fallback

以上的例子，gateway降级后就会将请求转发到http://localhost:9994。

Hystrix Gateway filter在转发降级请求时，会将造成降级的异常设置在ServerWebExchangeUtils.HYSTRIX\_EXECUTION\_EXCEPTION\_ATTR属性中，在处理降级时也可以用到。

比如下一节讲到的FallbackHeaders GatewayFilter Factory，就会通过上面的方式拿到异常信息，设置到降级转发请求的header上，来告知降级下游异常信息。

通过下面配置可以设置Hystrix的全局超时信息： application.yml：

hystrix.command.fallbackcmd.execution.isolation.thread.timeoutInMilliseconds: 5000

5. FallbackHeaders GatewayFilter Factory

FallbackHeaders GatewayFilter Factory可以将Hystrix执行的异常信息添加到外部请求的fallbackUriheader上。 application.yml：

spring:

cloud:

gateway:

routes:

- id: ingredients

uri: lb://ingredients

predicates:

- Path=//ingredients/\*\*

filters:

- name: Hystrix

args:

name: fetchIngredients

fallbackUri: forward:/fallback

- id: ingredients-fallback

uri: http://localhost:9994

predicates:

- Path=/fallback

filters:

- name: FallbackHeaders

args:

executionExceptionTypeHeaderName: Test-Header

在这个例子中，当请求lb://ingredients降级后，FallbackHeadersfilter会将HystrixCommand的异常信息，通过Test-Header带给http://localhost:9994服务。

你也可以使用默认的header，也可以像上面一下配置修改header的名字：

executionExceptionTypeHeaderName ("Execution-Exception-Type")

executionExceptionMessageHeaderName ("Execution-Exception-Message")

rootCauseExceptionTypeHeaderName ("Root-Cause-Exception-Type")

rootCauseExceptionMessageHeaderName ("Root-Cause-Exception-Message")

6. PrefixPath GatewayFilter Factory

The PrefixPath GatewayFilter Factor通过设置prefix参数来路径前缀。 application.yml：

spring:

cloud:

gateway:

routes:

- id: prefixpath\_route

uri: http://www.google.com

filters:

- PrefixPath=/mypath

如果一个请求是/hello，通过上面路由，就会将请求修改为/mypath/hello。

7. PreserveHostHeader GatewayFilter Factory

PreserveHostHeader GatewayFilter Factory会保留原始请求的host头信息，并原封不动的转发出去，而不是被gateway的http客户端重置。

application.yml:

spring:

cloud:

gateway:

routes:

- id: preserve\_host\_route

uri: http://www.google.com

filters:

- PreserveHostHeader

8. RequestRateLimiter GatewayFilter Factory

RequestRateLimiter GatewayFilter Factory使用RateLimiter来决定当前请求是否允许通过，如果不允许，则默认返回状态码HTTP 429 - Too Many Requests。

RequestRateLimiter GatewayFilter可以使用一个可选参数keyResolver来做速率限制。

keyResolver是KeyResolver接口的一个实现bean，在配置里面，通过SpEL表达式#{@myKeyResolver}来管理bean的名字myKeyResolver。

KeyResolver.java.

public interface KeyResolver {

Mono<String> resolve(ServerWebExchange exchange);

}

KeyResolver接口允许你使用不同的策略来得出限制请求的key，未来，官方也会推出一些KeyResolver的不同实现。

KeyResolver默认实现是PrincipalNameKeyResolver，通过ServerWebExchange中获取Principal，并以Principal.getName()作为限流的key。

如果KeyResolver拿不到key，请求默认都会被限制，你也可以自己配置spring.cloud.gateway.filter.request-rate-limiter.deny-empty-key：是否允许空key，spring.cloud.gateway.filter.request-rate-limiter.empty-key-status-code ：空key时返回的状态码。

RequestRateLimiter不支持捷径配置，如下面的配置是非法的

application.properties.

# INVALID SHORTCUT CONFIGURATION

spring.cloud.gateway.routes[0].filters[0]=RequestRateLimiter=2, 2, #{@userkeyresolver}

8.1 Redis RateLimiter

基于 Stripe的redis实现方案，依赖spring-boot-starter-data-redis-reactiveSpring Boot starter，使用的是令牌桶算法。

redis-rate-limiter.replenishRate配置的是每秒允许通过的请求数，其实就是令牌桶的填充速率。

redis-rate-limiter.burstCapacity配置的是一秒内最大的请求数，其实就是令牌桶的最大容量，如果设置为0，则会阻塞所有请求。

所以可以通过设置相同的replenishRate和burstCapacity来实现匀速的速率控制，通过设置burstCapacity大于replenishRate来允许系统流量瞬间突发，但是对于这种情况，突发周期为burstCapacity / replenishRate秒，如果周期内有两次请求突发的情况，则第二次会有部分请求丢失，返回HTTP 429 - Too Many Requests。

application.yml.

spring:

cloud:

gateway:

routes:

- id: requestratelimiter\_route

uri: http://www.google.com

filters:

- name: RequestRateLimiter

args:

redis-rate-limiter.replenishRate: 10

redis-rate-limiter.burstCapacity: 20

Config.java.

@Bean

KeyResolver userKeyResolver() {

return exchange -> Mono.just(exchange.getRequest().getQueryParams().getFirst("user"));

}

上面定义了每个用户每秒10个请求的速率限制，允许20的突发流量，突发完，下一秒只允许10个请求通过了，KeyResolver定义了通过请求获取请求参数user作为key。

你也可以实现RateLimiter接口自定义自己的请求速率限制器，在配置文件中使用SpEL表达式配置对应的bean的名字即可。

application.yml.

spring:

cloud:

gateway:

routes:

- id: requestratelimiter\_route

uri: http://www.google.com

filters:

- name: RequestRateLimiter

args:

rate-limiter: "#{@myRateLimiter}"

key-resolver: "#{@userKeyResolver}"

在最后有对请求限流具体介绍

9. RedirectTo GatewayFilter Factory

RedirectTo GatewayFilter Factory使用status和url两个参数，其中status必须是300系列的HTTP状态码，url则是跳转的地址，会放在响应的Location的header中（http协议中转跳的header）。

application.yml.

spring:

cloud:

gateway:

routes:

- id: prefixpath\_route

uri: http://www.google.cn

filters:

- RedirectTo=302, http://www.edjdhbb.com

上面路由会执行302重定向到http://www.edjdhbb.com。

10. RemoveNonProxyHeaders GatewayFilter Factory

RemoveNonProxyHeaders GatewayFilter Factory转发请求是会根据IETF的定义，默认会移除下列的http头信息：

Connection

Keep-Alive

Proxy-Authenticate

Proxy-Authorization

TE

Trailer

Transfer-Encoding

Upgrade

你也可以通过配置spring.cloud.gateway.filter.remove-non-proxy-headers.headers来更改需要移除的header列表。

11. RemoveRequestHeader GatewayFilter Factory

RemoveRequestHeader GatewayFilter Factory配置header的name，即可以移除请求的header。

application.yml.

spring:

cloud:

gateway:

routes:

- id: removerequestheader\_route

uri: http://www.google.com

filters:

- RemoveRequestHeader=X-Request-Foo

上面路由在发送请求给下游时，会将请求中的X-Request-Foo头信息去掉。

12. RemoveResponseHeader GatewayFilter Factory

RemoveResponseHeader GatewayFilter Factory通过配置header的name，会在响应返回时移除header。

application.yml.

spring:

cloud:

gateway:

routes:

- id: removeresponseheader\_route

uri: http://www.google.com

filters:

- RemoveResponseHeader=X-Response-Foo

上面路由会在响应返回给gateway的客户端时，将X-Response-Foo响应头信息去掉。

13. RewritePath GatewayFilter Factory

RewritePath GatewayFilter Factory使用路径regexp和替换路径replacement两个参数做路径重写，两个都可以灵活地使用java的正则表达式。

application.yml.

spring:

cloud:

gateway:

routes:

- id: rewritepath\_route

uri: http://www.google.com

predicates:

- Path=/foo/\*\*

filters:

- RewritePath=/foo/(?<segment>.\*), /$\{segment}

对于上面的例子，如果请求的路径是/foo/bar，则gateway会将请求路径改为/bar发送给下游。

注：在YAML 的格式中使用$\来代替$。

14. RewriteResponseHeader GatewayFilter Factory

RewriteResponseHeader GatewayFilter Factory的作用是修改响应返回的header内容，需要配置响应返回的header的name，匹配规则regexp和替换词replacement，也是支持java的正则表达式。

application.yml.

spring:

cloud:

gateway:

routes:

- id: rewriteresponseheader\_route

uri: http://www.google.com

filters:

- RewriteResponseHeader=X-Response-Foo, , password=[^&]+, password=\*\*\*

举个例子，对于上面的filter，如果响应的headerX-Response-Foo的内容是/42？user=ford&password=omg!what&flag=true，这个内容会修改为/42?user=ford&password=\*\*\*&flag=true。

15. SaveSession GatewayFilter Factory

SaveSession GatewayFilter Factory会在请求下游时强制执行WebSession::save方法，用在那种像Spring Session延迟数据存储的，并在请求转发前确保session状态保存情况。

application.yml.

spring:

cloud:

gateway:

routes:

- id: save\_session

uri: http://www.google.com

predicates:

- Path=/foo/\*\*

filters:

- SaveSession

如果你将Spring Secutiry于Spring Session集成使用，并想确保安全信息都传到下游机器，你就需要配置这个filter。

16. SecureHeaders GatewayFilter Factory

SecureHeaders GatewayFilter Factory会添加在返回响应中一系列安全作用的header，至于为什么，英文好的可以看一下这篇博客。

默认会添加这些头信息和默认内容：

X-Xss-Protection:1; mode=block

Strict-Transport-Security:max-age=631138519

X-Frame-Options:DENY

X-Content-Type-Options:nosniff

Referrer-Policy:no-referrer

Content-Security-Policy:default-src 'self' https:; font-src 'self' https: data:; img-src 'self' https: data:; object-src 'none'; script-src https:; style-src 'self' https: 'unsafe-inline'

X-Download-Options:noopen

X-Permitted-Cross-Domain-Policies:none

如果你想修改这些头信息的默认内容，可以在配置文件中添加下面的配置：

前缀：spring.cloud.gateway.filter.secure-headers

上面的header对应的后缀：

xss-protection-header

strict-transport-security

frame-options

content-type-options

referrer-policy

content-security-policy

download-options

permitted-cross-domain-policies

前后缀接起来即可，如：spring.cloud.gateway.filter.secure-headers.xss-protection-header

17. SetPath GatewayFilter Factory

SetPath GatewayFilter Factory采用路径template参数，通过请求路径的片段的模板化，来达到操作修改路径的母的，运行多个路径片段模板化。

application.yml.

spring:

cloud:

gateway:

routes:

- id: setpath\_route

uri: http://www.google.com

predicates:

- Path=/foo/{segment}

filters:

- SetPath=/{segment}

对于上面的例子，如果路径是/foo/bar，则对于下游的请求路径会修改为/bar。

18. SetResponseHeader GatewayFilter Factory

SetResponseHeader GatewayFilter Factory通过设置name和value来替换响应对于的header。

application.yml.

spring:

cloud:

gateway:

routes:

- id: setresponseheader\_route

uri: http://www.google.com

filters:

- SetResponseHeader=X-Response-Foo, Bar

对于上面的例子，如果下游的返回带有头信息为X-Response-Foo:1234，则会gateway会替换为X-Response-Foo:Bar，在返回给客户端。

19. SetStatus GatewayFilter Factory

SetStatus GatewayFilter Factory通过配置有效的Spring HttpStatus枚举参数，可以是类似于404的这些数字，也可以是枚举的name字符串，来修改响应的返回码。

application.yml.

spring:

cloud:

gateway:

routes:

- id: setresponseheader\_route

uri: http://www.google.com

filters:

- SetResponseHeader=X-Response-Foo, Bar

上面例子中，两种路由都会将响应的状态码设置为401。

20. StripPrefix GatewayFilter Factory

StripPrefix GatewayFilter Factory通过配置parts来表示截断路径前缀的数量。

application.yml.

spring:

cloud:

gateway:

routes:

- id: nameRoot

uri: http://nameservice

predicates:

- Path=/name/\*\*

filters:

- StripPrefix=2

如上面例子中，如果请求的路径为/name/bar/foo，则路径会修改为/foo，即将路径的两个前缀去掉了。

21. Retry GatewayFilter Factory

Retry GatewayFilter Factory可以配置针对不同的响应做请求重试，可以配置如下参数：

retries: 重试次数

statuses: 需要重试的状态码，需要根据枚举 org.springframework.http.HttpStatus来配置

methods: 需要重试的请求方法，需要根据枚举org.springframework.http.HttpMethod来配置

series: HTTP状态码系列，详情见枚举org.springframework.http.HttpStatus.Series

application.yml.

spring:

cloud:

gateway:

routes:

- id: retry\_test

uri: http://localhost:8080/flakey

predicates:

- Host=\*.retry.com

filters:

- name: Retry

args:

retries: 3

statuses: BAD\_GATEWAY

上面例子，当下游服务返回502状态码时，gateway会重试3次。

22. RequestSize GatewayFilter Factory

RequestSize GatewayFilter Factory会限制客户端请求包的大小，通过参数RequestSize来配置最大上传大小，单位字节。

application.yml.

spring:

cloud:

gateway:

routes:

- id: request\_size\_route

uri: http://localhost:8080/upload

predicates:

- Path=/upload

filters:

- name: RequestSize

args:

maxSize: 5000000

如果请求大小超过5000kb限制，则会返回状态码413 Payload Too Large。

如果不设置这个filter，默认限制5M的请求大小。

23. Modify Request Body GatewayFilter Factory

官方说这个filter目前只是beta版本，API以后可能会修改。

Modify Request Body GatewayFilter Factory可以修改请求体内容，这个只能通过java来配置。

@Bean

public RouteLocator routes(RouteLocatorBuilder builder) {

return builder.routes()

.route("rewrite\_request\_obj", r -> r.host("\*.rewriterequestobj.org")

.filters(f -> f.prefixPath("/httpbin")

.modifyRequestBody(String.class, Hello.class, MediaType.APPLICATION\_JSON\_VALUE,

(exchange, s) -> return Mono.just(new Hello(s.toUpperCase())))).uri(uri))

.build();

}

static class Hello {

String message;

public Hello() { }

public Hello(String message) {

this.message = message;

}

public String getMessage() {

return message;

}

public void setMessage(String message) {

this.message = message;

}

}

24. Modify Response Body GatewayFilter Factory

官方说这个filter目前只是beta版本，API以后可能会修改。

Modify Response Body GatewayFilter Factory用于修改响应返回的内容，同样只能通过java配置。

@Bean

public RouteLocator routes(RouteLocatorBuilder builder) {

return builder.routes()

.route("rewrite\_response\_upper", r -> r.host("\*.rewriteresponseupper.org")

.filters(f -> f.prefixPath("/httpbin")

.modifyResponseBody(String.class, String.class,

(exchange, s) -> Mono.just(s.toUpperCase()))).uri(uri)

.build();

}

7.集成Hystrix的断路器功能

分布式系统环境下，服务间类似依赖非常常见，一个业务调用通常依赖多个基础服务。如下图，对于同步调用，当库存服务不可用时，商品服务请求线程被阻塞，当有大批量请求调用库存服务时，最终可能导致整个商品服务资源耗尽，无法继续对外提供服务。并且这种不可用可能沿请求调用链向上传递，这种现象被称为雪崩效应。

雪崩效应常见场景

硬件故障：如服务器宕机，机房断电，光纤被挖断等。

流量激增：如异常流量，重试加大流量等。

缓存穿透：一般发生在应用重启，所有缓存失效时，以及短时间内大量缓存失效时。大量的缓存不命中，使请求直击后端服务，造成服务提供者超负荷运行，引起服务不可用。

程序BUG：如程序逻辑导致内存泄漏，JVM长时间FullGC等。

同步等待：服务间采用同步调用模式，同步等待造成的资源耗尽。

雪崩效应应对策略

针对造成雪崩效应的不同场景，可以使用不同的应对策略，没有一种通用所有场景的策略，参考如下：

硬件故障：多机房容灾、异地多活等。

流量激增：服务自动扩容、流量控制（限流、关闭重试）等。

缓存穿透：缓存预加载、缓存异步加载等。

程序BUG：修改程序bug、及时释放资源等。

同步等待：资源隔离、MQ解耦、不可用服务调用快速失败等。资源隔离通常指不同服务调用采用不同的线程池；不可用服务调用快速失败一般通过熔断器模式结合超时机制实现。

综上所述，如果一个应用不能对来自依赖的故障进行隔离，那该应用本身就处在被拖垮的风险中。 因此，为了构建稳定、可靠的分布式系统，我们的服务应当具有自我保护能力，当依赖服务不可用时，当前服务启动自我保护功能，从而避免发生雪崩效应。本文将重点介绍使用Hystrix解决同步等待的雪崩问题。

断路器工作原理

服务端的服务降级逻辑会因为hystrix命令调用依赖服务超时而触发，也就是说调用服务超时会进入断路回调逻辑处理。但是即使这样，受限于Hystrix超时时间的问题，调用依然会有可能产生堆积。

这个时候断路器就会发挥作用。这里涉及到断路器的三个重要参数：

快照时间窗

断路器确定是否打开需要统计一些请求和错误数据，而统计的时间范围就是快照时间窗，默认为最近的10秒。

请求总数下限

在快照时间窗内，必须满足请求总数下限才有资格熔断。默认为20，意味着在10秒内，如果该hystrix命令的调用次数不足20，即使所有的请求都超时或者其他原因失败，断路器都不会打开。

错误百分比下限

当请求总数在快照时间窗口内超过了下限，比如发生了30次调用，如果在这30次调用中有16次发生了超时异常，也就是超过了50%错误百分比，在默认设定50%下限情况下，这时候就会将断路器打开。

因此，断路器打开的条件是：在时间快照窗口期（默认为10s）内，至少发生20次服务调用，并且服务调用错误率超过50%。

不满足条件时断路器并不会打开，服务调用错误只会触发服务降级，也就是调用fallback函数，每个请求时间延迟就是近似hystrix的超时时间。如果将超时时间设置为5秒，那么每个请求都要延迟5每秒才会返回。当断路器在10秒内发现请求总数超过20并且错误率超过50%，这时候断路器会打开。之后再有请求调用的时候，将不会调用主逻辑，而是直接调用降级逻辑，这个时候就不会等待5秒之后才会返回fallback。通过断路器实现自动发现错误并将降级逻辑切换为主逻辑，减少响应延迟的效果。

在断路器打开之后，处理逻辑并没有结束，此时降级逻辑已经被切换为主逻辑了，那么原来的主逻辑要如何恢复呢？实际上hystrix也实现了这一点：当断路器打开，对主逻辑进行熔断之后，hystrix会启动一个休眠时间窗，在这个时间窗内，降级逻辑是临时的主逻辑，当休眠时间窗到期，断路器将进入半开状态，释放一次请求到原来的主逻辑，如果此次请求正常返回，那么断路器将进行闭合，主逻辑恢复，如果这次请求依然有问题，断路器继续进入打开状态，休眠时间窗重新计时。

换句话说，断路器每隔一段时间进行一次重试，看看原来的主逻辑是否可用，可用就关闭，不可用就继续打开。

通过上面的机制，hystrix的断路器实现了对依赖资源故障的处理，对降级策略的自动切换以及对主逻辑的自动恢复。这使得我们的微服务在依赖外部服务或资源的时候得到了非常好的保护，同时对于一些具备降级逻辑的业务需求可以实现自动化的切换和恢复，相比于设置开关由监控和运维来进行切换的传统实现方式显得更为智能和高效。

集成spring cloud gateway

maven依赖

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

yml配置文件配置

spring:

cloud:

gateway:

discovery:

locator:

enabled: true

default-filters:

- StripPrefix=1

- name: Hystrix

args:

name: fallback

fallbackUri: forward:/fallback

routes:

#基础架构服务

- id: orgnization-wx

uri: lb:http://orgnization-wx

predicates:

- Path=/structure/\*\*

fallback 实例

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

/\*\*

\* @author 荡漾

\* @title: PermissionFiler

\* @projectName cmp

\* @description: TODO

\* @date 2019/10/25 10:24

\*/

@RestController

@RequestMapping("/fallback")

public class FallbackController {

@RequestMapping("")

public ResponseEntity fallback(){

return ResponseEntity.status(HttpStatus.SERVICE\_UNAVAILABLE).body("服务暂不可用！！！！");

}

}

8.请求限流

在高并发的系统中，往往需要在系统中做限流，一方面是为了防止大量的请求使服务器过载，导致服务不可用，另一方面是为了防止网络攻击。

一般开发高并发系统常见的限流有：限制总并发数（比如数据库连接池、线程池）、限制瞬时并发数（如 nginx 的 limit\_conn 模块，用来限制瞬时并发连接数）、限制时间窗口内的平均速率（如 Guava 的 RateLimiter、nginx 的 limit\_req 模块，限制每秒的平均速率）；其他还有如限制远程接口调用速率、限制 MQ 的消费速率。另外还可以根据网络连接数、网络流量、CPU 或内存负载等来限流。

限流算法

计数器

简单的做法是维护一个单位时间内的 计数器，每次请求计数器加1，当单位时间内计数器累加到大于设定的阈值，则之后的请求都被拒绝，直到单位时间已经过去，再将 计数器 重置为零。此方式有个弊端：如果在单位时间1s内允许100个请求，在10ms已经通过了100个请求，那后面的990ms，只能眼巴巴的把请求拒绝，我们把这种现象称为“突刺现象”。

常用的更平滑的限流算法有两种：漏桶算法 和 令牌桶算法。下面介绍下二者。

漏桶算法

漏桶算法思路很简单，水（请求）先进入到漏桶里，漏桶以一定的速度出水（接口有响应速率），当水流入速度过大会直接溢出（访问频率超过接口响应速率），然后就拒绝请求，可以看出漏桶算法能强行限制数据的传输速率。

可见这里有两个变量，一个是桶的大小，支持流量突发增多时可以存多少的水（burst），另一个是水桶漏洞的大小（rate）。因为漏桶的漏出速率是固定的参数，所以，即使网络中不存在资源冲突（没有发生拥塞），漏桶算法也不能使流突发（burst）到端口速率。因此，漏桶算法对于存在突发特性的流量来说缺乏效率。

令牌桶算法

令牌桶算法 和漏桶算法 效果一样但方向相反的算法，更加容易理解。随着时间流逝，系统会按恒定 1/QPS 时间间隔（如果 QPS=100，则间隔是 10ms）往桶里加入 Token（想象和漏洞漏水相反，有个水龙头在不断的加水），如果桶已经满了就不再加了。新请求来临时，会各自拿走一个 Token，如果没有 Token 可拿了就阻塞或者拒绝服务。

令牌桶的另外一个好处是可以方便的改变速度。一旦需要提高速率，则按需提高放入桶中的令牌的速率。一般会定时（比如 100 毫秒）往桶中增加一定数量的令牌，有些变种算法则实时的计算应该增加的令牌的数量。

限流实现

在 Spring Cloud Gateway 上实现限流是个不错的选择，只需要编写一个过滤器就可以了。有了前边过滤器的基础，写起来很轻松。

Spring Cloud Gateway 已经内置了一个RequestRateLimiterGatewayFilterFactory，我们可以直接使用。

目前RequestRateLimiterGatewayFilterFactory的实现依赖于 Redis，所以我们还要引入spring-boot-starter-data-redis-reactive。

pom.xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-gateway</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifatId>spring-boot-starter-data-redis-reactive</artifactId>

</dependency>

application.yml

server:

port: 8080

spring:

cloud:

gateway:

routes:

- id: limit\_route

uri: http://httpbin.org:80/get

predicates:

- After=2019-02-26T00:00:00+08:00[Asia/Shanghai]

filters:

- name: RequestRateLimiter

args:

key-resolver: '#{@hostAddrKeyResolver}'

redis-rate-limiter.replenishRate: 1

redis-rate-limiter.burstCapacity: 3

application:

name: gateway-limiter

redis:

host: localhost

port: 6379

database: 0

在上面的配置文件，配置了 redis的信息，并配置了RequestRateLimiter的限流过滤器，该过滤器需要配置三个参数：

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。

IP限流

获取请求用户ip作为限流key。

@Bean

public KeyResolver hostAddrKeyResolver() {

return exchange -> Mono.just(exchange.getRequest().getRemoteAddress().getHostName());

}

用户限流

获取请求用户id作为限流key。

@Bean

public KeyResolver userKeyResolver() {

return exchange -> Mono.just(exchange.getRequest().getQueryParams().getFirst("userId"));

}

接口限流

获取请求地址的uri作为限流key。

@Bean

KeyResolver apiKeyResolver() {

return exchange -> Mono.just(exchange.getRequest().getPath().value());

}