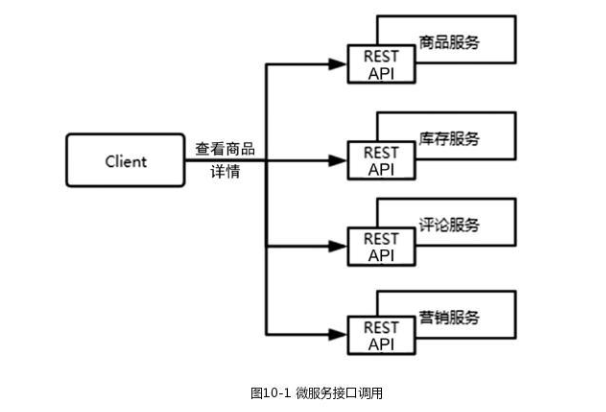
# 认识网关

## 为什么要使用网关

在微服务架构中，每个服务都是一个可以独立开发和运行的组件，而一个完整的微服务架构由一系列独立运行的微服务组成。其中每个服务都只会完成特定领域的功能，比如订单服务提供与订单业务场景有关的功能、商品服务提供商品展示功能等。各个微服务之间通过轻量级通信机制REST API或者RPC完成通信。实现微服务之后在某些层面会带来一定的影响，比如，一个用户查看一个商品的详情，对于客户端来说，可能需要调用商品服务、评论服务、库存服务、营销服务等多个服务来完成数据的渲染，如下图所示：



在这个场景中，客户端虽然能通过调用多个服务实现数据的获取，但是会存在一些问题，比如：

·客户端需要发起多次请求，增加了网络通信的成本及客户端处理的复杂性。

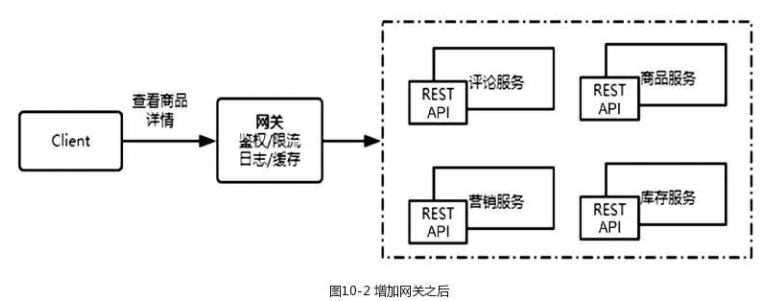
·服务的鉴权会分布在每个微服务中处理，客户端对于每个服务的调用都需要重复

鉴权

·在后端的微服务架构中，可能不同的服务采用的协议不同，比如有HTTP、RPC

等。客户端如果需要调用多个服务，需要对不同协议进行适配。

而网关可以解决这些问题，在客户端与服务端之间增加一个API网关，网关优点类似于门面，所有的外部请求都会先经过网关这一层。如下图所示：



网关不仅仅只是做一个请求的转发及服务的整合，有了网关这个统一的入口之后，它还提供一下功能：

·针对所有请求进行统一鉴权、限流、熔断、日志。

·协议转换。针对后端多种不同的协议，在网关层统一处理后以HTTP对外提供服

务。用过Dubbo框架的读者应该知道，针对Dubbo服务还需要提供一个Web应

用来进行协议转化。

·统一错误码处理。

·请求转发，并且可以基于网关实现内、外网隔离。

## 统一认证鉴权

**统一认证鉴权的组成**

统一认证鉴权包含如下两部分：

·客户端身份验证：主要用于判断当前用户是否为合法用户，一般的做法是使

用账号和密码进行验证。当然，对于一些复杂的认证场景会采用加密算法来

实现，比如公、私钥。

·访问权限控制：身份认证和访问权限一般是相互联系的，当身份认证通过后

，就需要判断该用户是否有权限访问该资源，或者该用户的访问权限是否被

限制了。

**实现统一认证鉴权**

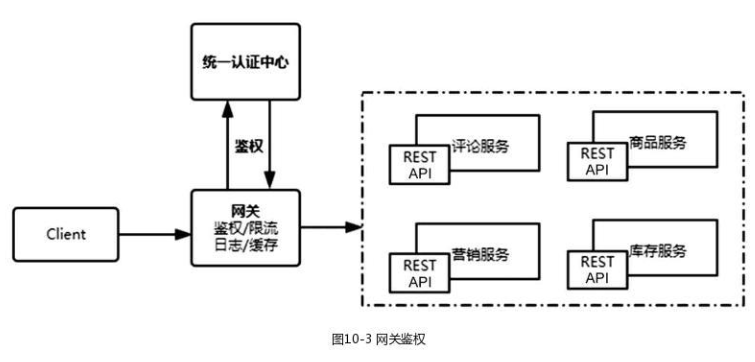
在单体应用中，客户端身份认证及访问权限的控制比较简单，只需要在服务端通过session保存该用户信息即可。但是在微服务架构下，单体应用被拆分成多个微服务，鉴权的过程就会变得很复杂。

**·**首先要解决的问题是，原来单体应用中的session方式已经无法用于微服务

场景

**·**其次就是如何实现对每个微服务进行鉴权

对于第一个问题，目前已经有非常多的成熟解决方案了，比如AccessToken、Oauth（开放API）等。对于第二个问题，我们可以把鉴权的功能抽离出一个统一认证服务，所有的微服务在被访问之前，先访问该认证服务进行鉴权。这种解决方案看似合理，但是在实际应用中，一个业务场景中可能会调用多个微服务，就会造成一次请求需要进行多次鉴权操作，增加了网络通信开销。如下图所示：



增加API网关之后，在网关层进行请求拦截，获取请求中附带的用户身份信息，调用统一认证中心对请求进行身份认证，在确认了身份之后再检查是否有访问资源的访问权限。

## 灰度发布

**为什么要使用灰度发布**

互联网公司的产品的迭代速度非常快，很多公司会采用一周发布一个版本的迭代模式。在这种高频率的迭代模式下，往往会伴随着一些风险，比如：

**·**新发布的代码程序兼容性问题

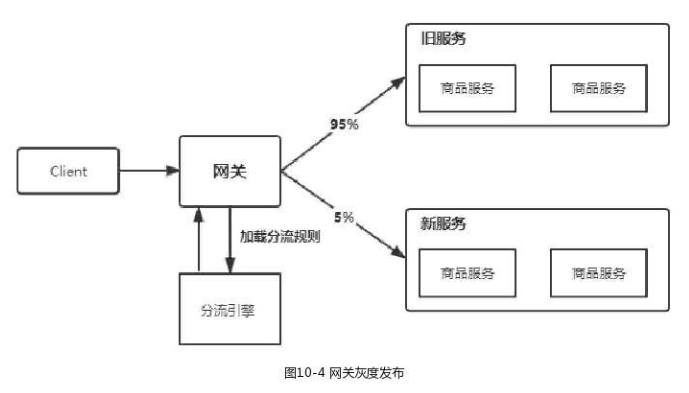
**·**新的功能发布后，用户是否能够接受，如果不能，会造成用户流式

**·**代码中存在隐藏的Bug，导致线上故障

为了避免这些问题，对于有较大的功能性改动的版本一般都会采用灰度发布（又名金丝雀发布）的方式来实现平滑过渡。

**什么是灰度发布**

所谓灰度发布，即使将要发布的功能先开发给一部分用户使用，把影响范围控制在一个非常小的范围，比如A/B Test就是一种灰度发布方式，即一部分用户继续使用A功能，另外一小部分用户使用新的B功能。通过对使用B功能的用户进行满意度调查，以及对新发布的代码的性能和稳定性指标进行评测，逐步放大该新版本的投放，直到全量或者回滚该版本。对于应用系统来说，无非就是将新的功能发布在特定的灰度机器上，然后根据设定的规则将部分请求路由到灰度服务器上。网关是所有客户端请求的入口，因此在网关可以通过灰度规则进行部分流量的路由，从而实现灰度发布。如下图所示：



# 网关的本质及技术选型

## 网关的本质

**网关的本质是对请求进行路由转发，以及对请求进行前置和后置的过滤。**

·请求的转发和路由：接收客户端的所有请求，并将请求转发到后端的微服务中。

因为微服务的粒度比较细，所以API网关有类似于门面系统，对多个微服务进行

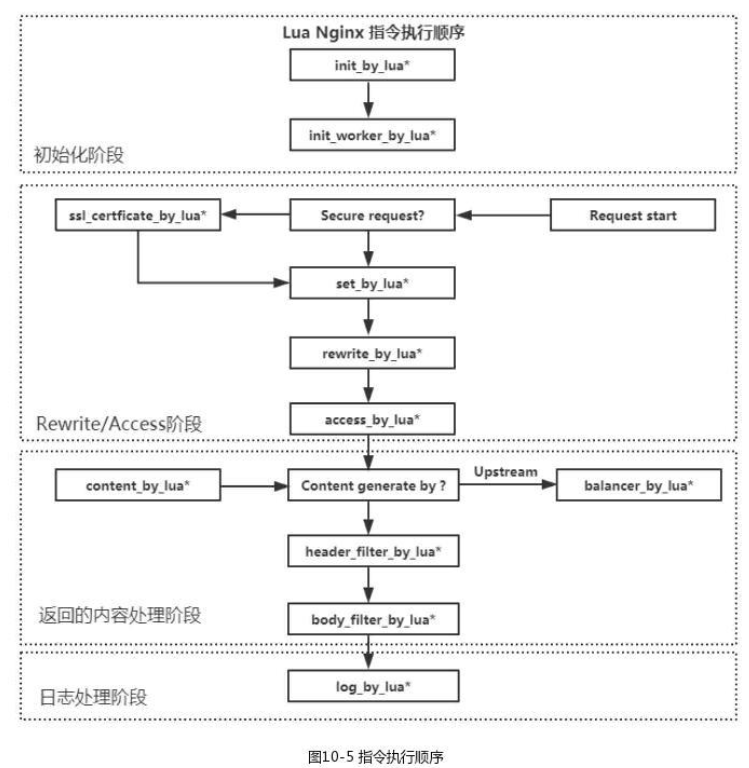
功能整合，提供唯一的业务接口给客户端。

·过滤：网关会拦截所有的请求来完成一系列的横切工作，比如鉴权、限流。

常见的开源API网关实现方案有很多，比如OpenResty、Zuul、GateWay、Orange、Kong、Tyk等。

## OpenResty

OpenResty是由Nginx和Lua集成的一个高性能Web应用服务器，它的内部集成了大量优秀的Lua库、第三方模块。并且OpenResty团队自己研发了很多优秀的Nginx模块，开发人员可以适应Lua脚本来调用Nginx支持的C模块及Lua模块。OpenResty本质上时将Lua脚本嵌入Nginx中，在每个Nginx的进程中都嵌入了一个LuaJIT虚拟机来执行Lua脚本。对于OpenResty，它本质上仍然是Nginx服务器，可以实现反向代理和负载均衡，而网关的本质是对请求进行转发及过滤，这意味着OpenResty在接收到客户端的请求时同样可以拦截请求进行前置和后置的处理，对此，OpenResty通过在不同的阶段来挂载Lua脚本实现不同阶段的自定义行为。如下图所示：



可以看到init\_by\_lua、init\_worker\_by\_lua、set\_by\_lua等11个指令，OpenResty实现网关功能的核心就是在这11个步骤中挂载Lua脚本来实现功能的扩展。一个请求进入OpenResty之后，会根据请求所在的不同阶段按照上图所示的流程执行不同的指令，每个指令的作用如下：

·init\_by\_lua：当Nginx Master进程加载Nginx配置文件时会运行这段Lua脚本

·init\_worker\_by\_lua：每个Nginx worker进程启动时会执行的Lua脚本，可以用来

做健康检查。

·ssl\_certificate\_by\_lua：当Nginx开始对下游进行SSL（HTTPS）握手连接时，该指

令执行用户Lua代码

·set\_by\_lua：设置一个变量

·rewrite\_by\_lua：在rewrite阶段执行

·access\_by\_lua：在访问阶段调用Lua脚本

·content\_by\_lua：通过Lua脚本生成content输出给HTTP响应

·balancer\_by\_lua：实现动态负载均衡，如果不走contentbylus，则走proxy\_pass，

再通过upstream进行转发。

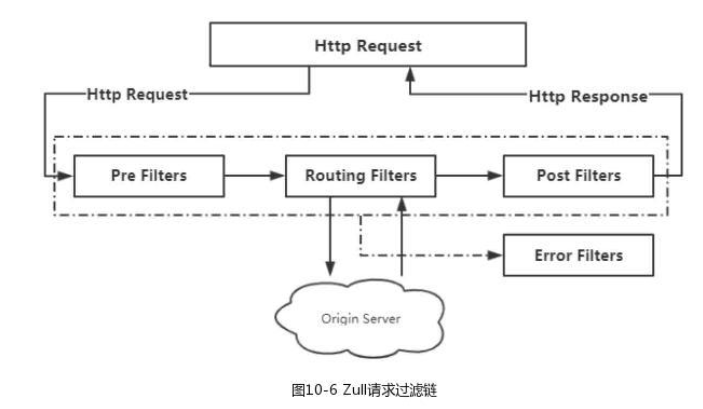
·heander\_filter\_by\_lua：通过Lua来设置Header或者Cookie。

·body\_filter\_by\_lua：对响应数据进行过滤。

·log\_by\_lua：在Log阶段执行的脚本

## Spring Cloud Zuul

Zuul是Netflix开源的微服务网关，它的主要功能是路由转发和过滤。Zuul是在Spring Cloud Netflix生态中的，它被整合到Spring Cloud中为微服务架构提供API网关的功能。Zuul的核心由一系列过滤器组成，如下：



它定义了4种标准类型的过滤器，这些会对应请求的整个生命周期。

·Pre Filters：前置过滤器，请求被路由之前调用，可以用于处理鉴权、限流等。

·Routing Filters：路由过滤器，将请求路由到后端的微服务

·Post Filter：后置过滤器，路由过滤中远程调用结束后执行。可以用于做统计、

监控、日志等。

·Error Filters：错误过滤器，任意一个过滤器出现异常或者远程调用超时时会被调

用。

## Spring Cloud Gateway

Spring Cloud Gateway是Spring官方团队研发的API网关技术，它的目的是取代Zuul为微服务提供一种简单高效的API网关。Spring团队研发Spring Cloud Gateway取代Zuul的原因主要如下：

·Zuul1.x采用的是传统的thread per connection方式来处理请求，也就是针对每一

个请求，会为这个请求专门分配一个线程来处理，知道这个请求完成之后才会释

放线程，一旦后台服务器响应很慢，就是使得该线程被阻塞，所以它的性能不是

很好。

·Zuul本身存在一些性能问题不适合于高并发的场景，虽然后来Netflix决定开发

高性能版Zuul2.x，但是Zuul2.x的发布时间一直不确定。后台Zuul2.x发布开源

了，但是Spring Cloud并没有打算集成进来。

Spring Cloud Gateway是依赖于Spring Boot2.0、Spring WebFlux和Project Reactor等技术开发的网关，它不仅提供了统一的路由请求的方式，还基于过滤链的方式提供了网关最基本的功能。

# Spring Cloud Gateway网关实战与原理分析

## Spring Cloud Gateway实战

使用一个简单的案例演示Spring Cloud Gateway的使用方法，首先准备两个Spring Boot应用：

·spring-cloud-gateway-service，模拟一个微服务

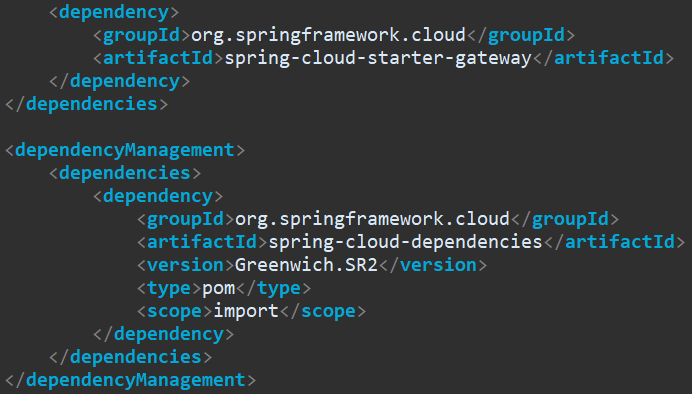
·spring-cloud-gateway-sample，独立的网关服务

**spring-cloud-gateway-service**

基于Spring Boot脚手架构建一个应用，添加spring-boot-starter-web依赖。创建一个HelloController类发布一个接口并启动该应用。

**spring-cloud-gateway-sample**

1. 创建Spring Boot应用，添加Spring Cloud Gateway依赖



1. 在application.yml文件中添加Gateway的路由配置



上述配置的字段含义说明如下：

·id：自定路由ID，需要保持唯一。

·uri：目标服务地址，支持普通URI及lb://应用注册服务名称，后者表

示从注册中心获取集群服务地址。

·predicates：路由条件，根据匹配的结果决定是否执行该请求路由。

·filters：过滤规则，包含pre和post过滤。其中StripPrefix=1，表示Gat

eway根据该配置的值去掉URL路径中的部分前缀（这里去掉一个前缀，

即在转发的目标URI中去掉gateway）。

1. 启动项目，在控制台可以获得如下信息：

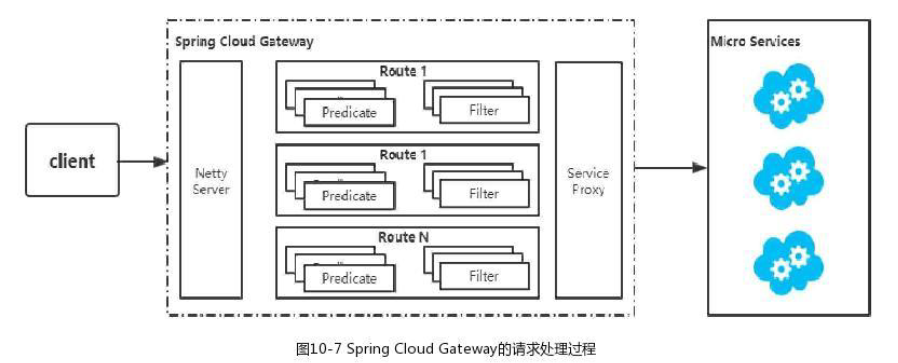
可以看到，它并没有依赖Tomcat，而是用NettyWebServer来启动一个服务监听。访问网关应用，测试是否实现请求的路由：

<http://localhost:8088/gateway/say>

在配置正确的情况下，将会获得返回结果“[spring-cloud-gateway-service]:say Hello”，即通过网关访问微服务。

## Spring Cloud Gateway原理分析

Spring Cloud Gateway的请求处理过程如下图所示：



其中有几个非常重要的概念：

·路由（Route）：它是网关的基本组件，由ID、目标URI、Predicate集合、Filter

集合组成。

·谓语（Predicate）：它是Java8引入的函数式接口，提供了断言的功能。它可以匹

配HTTP请求中的任何内容。如果Predicate的聚合判断结果为true，则意味着该

请求会被当前Router进行转发。

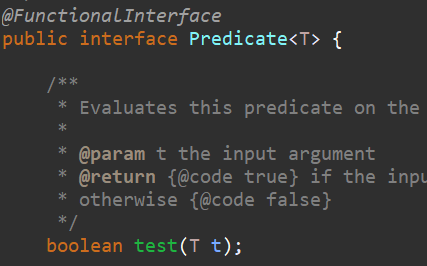
·过滤器（Filter）：为请求提供前置和后置的过滤。

Spring Cloud Gateway启动时基于Netty Server监听一个指定的端口（该端口可以通过server.port属性自定义，默认8080）。当客户端发送一个请求到网关时，网关会根据一系列Predicate的匹配结果来决定访问哪个Route路由，然后根据过滤器链进行请求的处理。过滤器可以在请求发送到后端服务器之前和之后执行，也就是首先执行Pre过滤器链，然后将请求转发到后端服务器，最后执行Post过滤器链。

# Route Predicate Factories

## 认识Predicate

Predicate是Java8提供的一个函数式接口，它允许接收一个参数并返回一个布尔值，可以用于条件过滤、请求参数的校验。部分源码如下：



在Spring Cloud Gateway中，Predicate提供了路由规则的匹配机制，比如上一节Spring Cloud Gateway实战中配置的Predicates规则spring.cloud.gateway.router[0].predicates[0]=Path=/gateway/\*\*，意思是通过Path来匹配URL前缀是gateway的请求。

如下图所示，Spring Cloud Gateway默认提供了很多Route Predicate Factory，这些Predicate会分别区分匹配HTTP请求的不同属性，并且多个Predicate可以通过and逻辑进行组合。



·ZoneDateTime：指定时间规则指定路由

·Cookie：Cookie匹配路由

·Header：Header匹配路由

·Host：Host匹配路由

·Method：请求方法匹配路由

·Path：请求路径匹配路由

Query和RemoteAddr在此不做介绍。

## 指定时间规则匹配路由

根据配置的时间来匹配路由，包含以下三种：

·请求在指定日期之前，对应BeforeRoutePredicateFactory

·请求在指定日期之后，对应AfterRoutePredicateFactory

·请求在指定的两个日期之间，对应BetweenRoutePredicateFactor

比如，我们希望在2021年1月1日之前发生的请求都路由到[www.example.com](http://www.example.com)，配置如下：



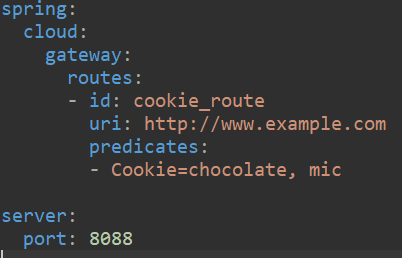
需要注意的是，配置的日期时间必须满足ZonedDateTime的格式，该格式说明如下：



如：2020-06-01T20:00:00.000+08:00[Asia/Shanghai]

## Cookie匹配路由

Cookie匹配路由规则（即CookieRoutePredicateFactory），判断请求中携带的Cookie是否匹配配置的规则，配置如下：



该配置的意思是，当前请求需要携带一个name=chocolate，并且value的需要通过正则表达式匹配mic，才能路由到<http://www. example.com>。（可用Postman测试）

## Header匹配路由

Header匹配路由规则，判断请求中Header头信息对应的name和value与Predicate配置的值是否匹配，value也是正则匹配形式的。如：



该配置中会匹配请求中Header头中的name=X-Request-Id，并且value会根据正则表达式匹配\d+，就是匹配1个以上的数字。（可用Postman测试）

## Host匹配路由

HTTP请求会携带一个Host字段，Host字段表示请求的服务网址。Host匹配路由规则即使匹配请求中的Host字段进行路由。如：



Host可以配置一个列表，列表中的每个元素通过“，”分隔，且也支持正则匹配。在上述配置中，当前请求的Host的值符合\*\*.somehost.com，\*\*antherhost.com时，才会将请求路由到<http://www.baidu.com>，路径命名及配置规则支持Ant Path，比如www.somehost.com、test.somehost.com、www.anotherhost.com都符合该规则。（可用Postman测试，只需在配置的列表中包含当前网关的Host即可）

## 请求方法匹配路由

请求方法路由规则会根据HTTP请求的Method属性来匹配以实现路由。如：

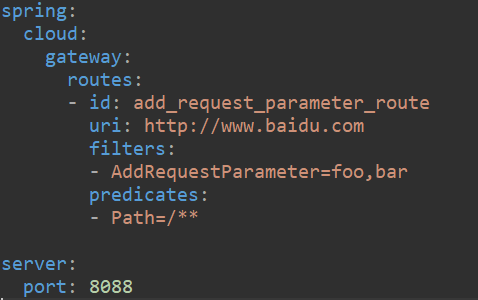


该配置表示，如果HTTP请求的方式是GET或者POST，都会路由到http://www.baidu.com。

（可用Postman测试）

## 请求路径匹配路由

请求路径匹配路由规则是比较常见的路由匹配规则。如：



${segment}是一种比较特殊的占位符，/\*表示单层路径匹配，/\*\*表示多层路径匹配。上述配置规则中，匹配请求的URI为/red/\*、/blue/\*时，才会转发到http://www.baidu.com，但配置了/\*\*，表示所有请求都会转发给http://www.baidu.com（可用Postman测试）

# Gateway Filter Factories

## 认识Gateway中的Filter

Filter分为Pre类型的过滤器和Post类型的过滤器。

·Pre类型的过滤器在请求转发到后端微服务之前执行，在Pre类型过滤器链中可

以做鉴权限流等操作。

·Post类型的过滤器在请求执行完之后，将结果返回给客户端之前执行。

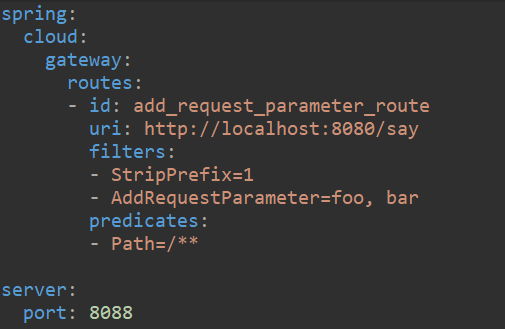
在Spring Cloud Gateway中内置了很多Filter，Filter有两种实现，分别是GatewayFilter和GlobalFilter。GlobalFilter会应用到所有的路由上，而GatewayFilter只会应用到单个路由或者一个分组的路由上。

## GatewayFilter

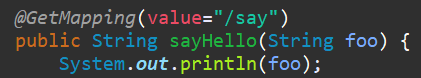
由于内置的GatewayFilter比较多，本节只列举几种进行说明。

### AddRequestParameter GatewayFilter Factory

该过滤器的功能是对所有匹配的请求添加一个查询参数。如：



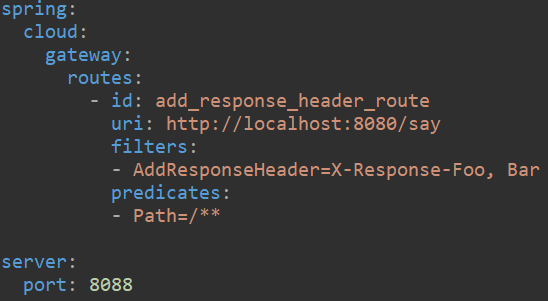
在上面的这段配置中，会对所有的请求增加foo=bar这个参数，/say是微服务中的一个接口，可以接收foo参数，验证其是否增参了，如下：



同样也可以使用Postman进行测试。

### AddResponseHeader GatewayFilter Factory

该过滤器会对所有匹配的请求，在返回结果给客户端之前，在Header中添加相应的数据。如：

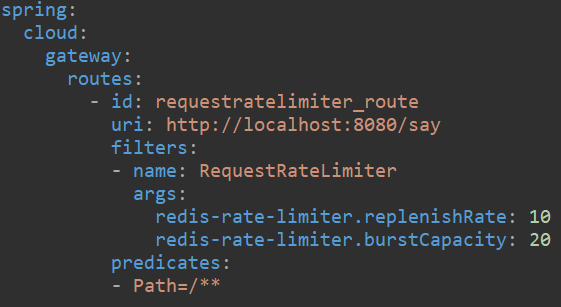


在上面的这段配置中，会在Response中添加Header头，key=X-Response-Foo，Value=Bar。

（可在Postman测试）

### RequestRateLimiter GatewayFilter Factory

该过滤器会对访问到当前网关的所有请求执行限流过滤，如果被限流，默认情况下会响应HTTP 429-Too Many Requests。RequestRateLimiterGatewayFilterFactory默认提供了RedisRateLimiter的限流实现，它采用令牌桶算法来实现限流功能（关于令牌桶算法在第7章已经讲过）。如：



redis-rate-limiter过滤器有两个配置属性，如果理解令牌桶算法，就很容易知道它们的含义：

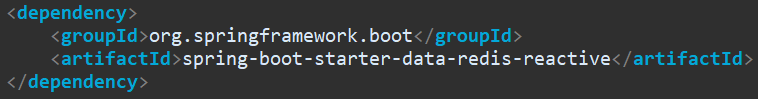
·replenishRate：令牌桶中令牌的填充速度，代表允许每秒执行的请求数

·burstCapacity：令牌桶的容量，也就是令牌桶做多能够容纳的令牌数。表示每秒

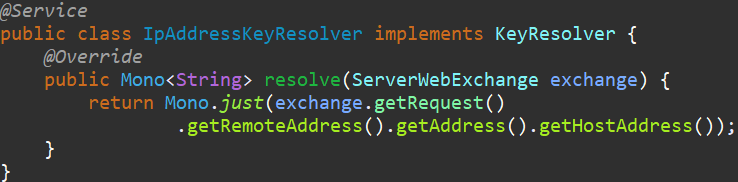
用户最大能够执行的请求数量。

下面通过一个示例来实现限制同一个IP的请求频次。

1. 添加Jar包依赖，Redis的限流器基于Stripe实现，它需要引入下面这个依赖包：



1. 创建一个KeyResolver的实现类



KeyResolver接口主要用于设置限流请求的key，我们可以实现该接口来指定需要对当前请求中的哪些因素进行流量控制。在上述代码中设置的是HostAddress，表示根据请求IP来限流。

KeyResolver的默认实现的PrincipalNameKeyResolver，它会从ServerWebExchange检索Principal并调用Principal.getName。在默认情况下，如果KeyResolver没有获取到key，请求将被拒绝。我们可以通过以下两个属性来调整

·spring.cloud.gateway.filter.request-rate-limiter.denyEmptyKey，是否允许空的key

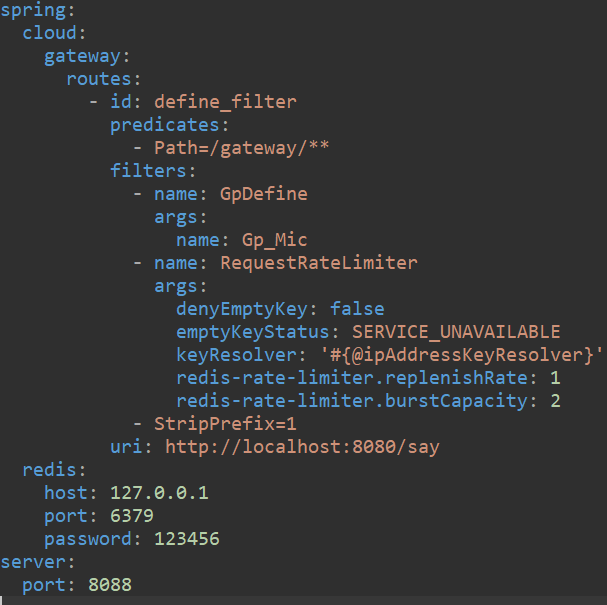
·spring.cloud.gateway.filter.request-rate-limiter.emptyKeyStatus，当denyEmptyKey

=true时返回的HttpStatus，默认为FORBIDDEN(403,”Forbidden”)。

4）定义一个类



1. 在application.yml中添加如下配置：



上述配置中，keyResolver采用的是SpEL表达式按照名称来引用Bean，#{@ipAddressKeyResolver}表示引用name=ipAddressKeyResolver的Bean。

1. 启动Redis服务器和启动项目后，通过测试工具访问网关即可看到限流的效果，默认响应HTTP ERROR 429。

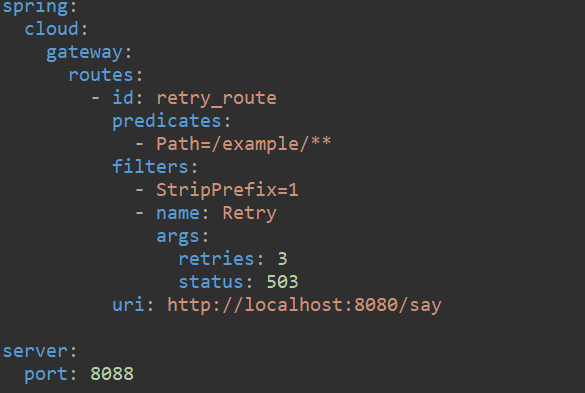
Spring Cloud Gateway目前默认只实现了基于Redis的Ratelimiter限流方式，如果我们想使用其他方式实现限流，它也提供了扩展，实现方式类似于keyResolver。

·创建自定义限流器，实现AbstractRateLimiter接口

·指定自定义限流器，rateLimiter:#{@defineRateLimiter}

### Retry GatewayFilter Factory

Retry GatewayFilter Factory为请求重试过滤器，当后端服务不可用时，网关会根据句配置参数来发起重试请求。



RetryGatewayFilter提供4个参数来控制重试请求，参数说明如下：

·retries：请求重试次数，默认值是3

·status：HTTP请求返回的状态码，针对指定状态码进行重试，比如，在上述配置

中，当服务端返回的状态是503时，才会发起重试，此处可以配置多个状态码。

·methods：指定HTTP请求中哪些方法类型需要进行重试，默认值是GET

·series：配置错误码段，表示符合某段状态码才发起重试，默认值是SERVER\_ERR

OR(5)，表示5xx段的状态码都会发起重试。如果series配置了错误码段，但是

status没有配置，则仍然会匹配series进行重试。

## GlobalFilter

GlobalFiler和GatewayFilter的作用是相同的，只是GlobalFilter针对所有的路由配置生效，也称全局过滤。Spring Cloud Gateway内置的全局过滤器也有很多，比如：

·GatewayMetricsFilter，提供监控指标

·LoadBalancerClientFilter，整合Ribbon （Spring Cloud Ribbon是一个基于HTTP和

TCP的客户端负载均衡工具，它基于Netflix Ribbon实现）针对下游服务实现负载

均衡

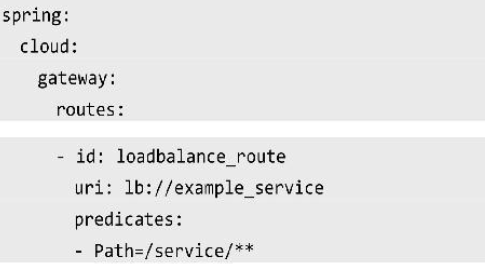
·ForwardRoutingFilter，用于本地forward，请求不转发到下游服务器

·NettyRoutingFilter，使用Netty的HttpClient转发HTTP、HTTPS请求

全局过滤器链的执行顺序是，当Gateway接收到请求时，Filtering Web Handler处理器会将所有的GlobalFilter实例及所有路由上所配置的GatewayFilter实例添加到一条过滤器链中。该过滤器链里的所有过滤器都会按照@Order注解所指定的数字大小进行排序。

### LoadBalancerClientFilter

LoadBalancerClientFilter是用于实现请求负载均衡的全局过滤器。配置如下：



如果URI配置的是lb:example\_service，那么这个过滤器会识别到lb://，并且使用Spring Cloud LoadBalancerClient将example\_service名称解析成实际访问主机和端口地址，具体的使用方法会在后面详细分析。

### GatewayMetricsFilter

GatewayMetricsFilter是网关指标过滤器，这个过滤器会添加name=gateway.requests的timermetrics，它包含以下数据：

·routeId：路由ID

·routeUri：API网关将路由到的URI

·outcome：返回的状态码，值的枚举类定义在HttpStatus.Series中

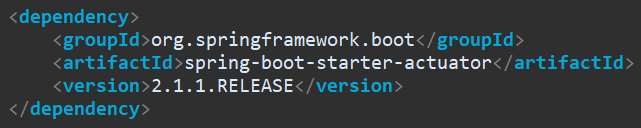
·status：返回给客户端的HTTP Status

·httpStatusCode：返回给客户端的HttpStatusCode，如200

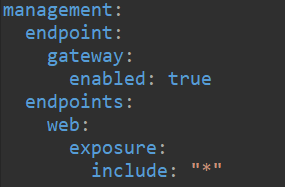
·httpMethod：请求所使用的HTTP方法

这些坐标可以通过<http://ip:port/actuator/metrics/gateway.requests>获得，前提是要添加Spring Boot Actuator依赖，具体配置步骤如下：

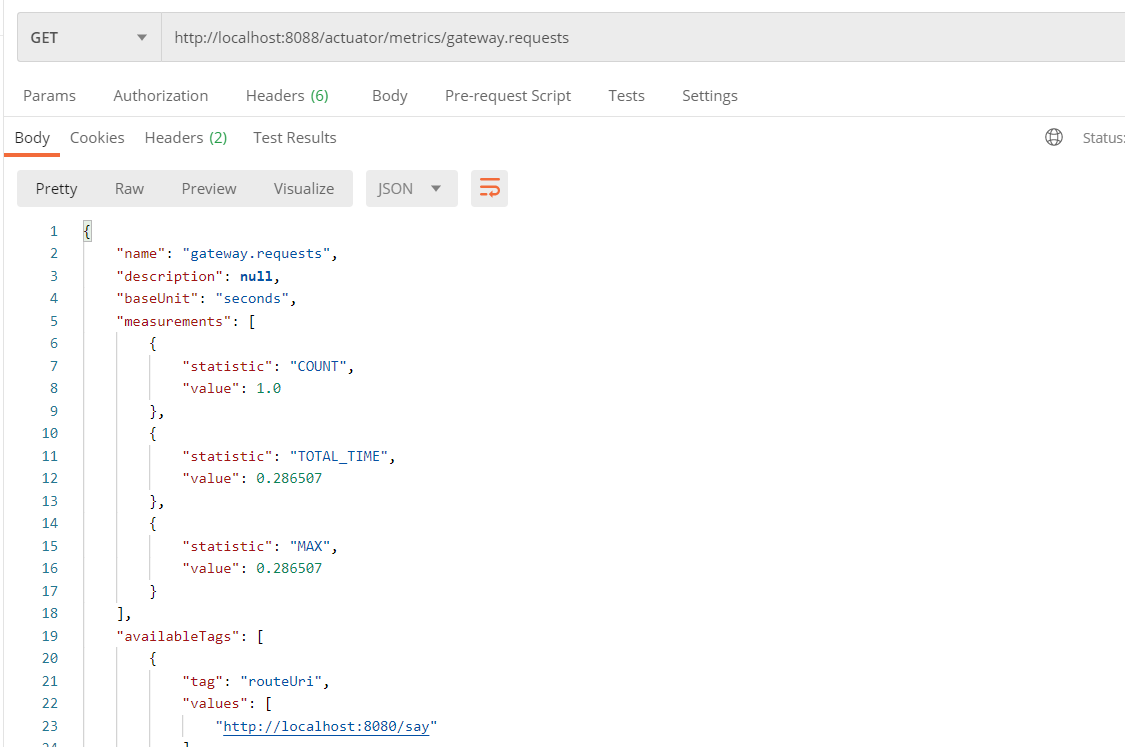
1. 添加Spring Boot Actuator依赖



1. 配置application.yml



1. 使用Postman访问gatewaymetric端点，将会获得指标数据（如果数据为空，可以先访问一次网关的接口）：



在实际应用中，我们可以将这些信息发布到监控平台上，比如可以集成Prometheus（服务监控系统），进行可视化监控。

# 自定义过滤器

Spring Cloud Gateway提供了过滤器的扩展功能，开发者可以根据实际业务需要来自定义过滤器。同样，自定义过滤器也支持GatewayFilter和GlobalFilter两种。

## 自定义GatewayFilter

1. 首先创建一个自定义过滤器GpDefineGatewayFilterFactory，继承AbstractGatewayFilterFactory。



在上述代码中，需要注意：

·类名必须要统一以GatewayFilterFactory结尾，因为默认情况下过滤器的name会

采用该自定义类的前缀。这里的name=GpDefine。

·在apply方法中，包含Pre过滤和Post过滤。在then方法中是请求执行结束之

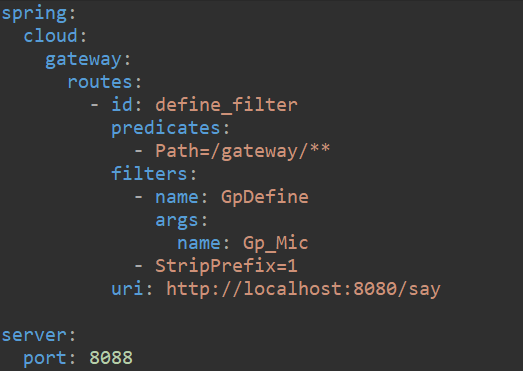
后的后置处理。

·GpConfig是一个配置类，该类中只有一个属性name。这个属性可以在yml文件

中使用。

·该类需要装载到Spring IoC容器中，此处使用@Service注解实现。

1. 在application.yml文件中配置该自定义过滤器



其中，name属性就是GpDefineGatewayFilterFactory的前缀。而args中的name属性

是GpConfig配置类中声明的属性，这个属性配置好后，可以在代码中获得这个name

对应的值Gp\_Mic

1. 启动微服务与网关，使用Postman访问网关



可以在控制台获得如下日志信息：



说明自定义Global过滤器配置正确。

## 自定义GlobaFilter

GlobalFilter的实现比较简单，它不需要遵守命名约定和额外配置，只需要实现GlobalFilter接口，自动会过滤所有的Route。



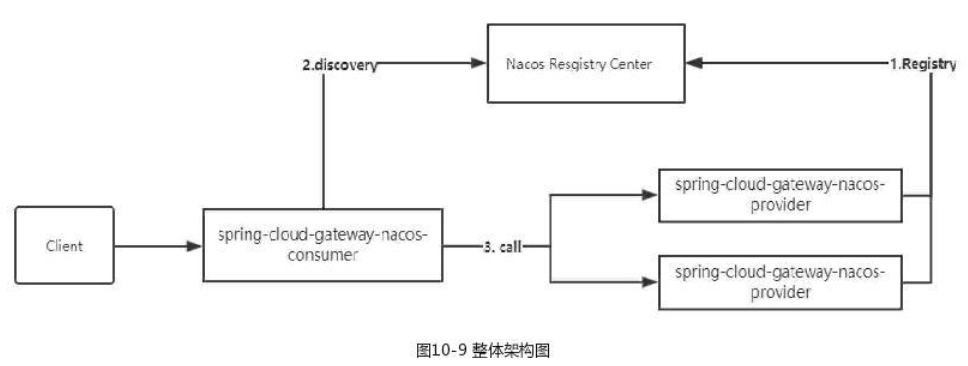
getOrder方法的返回值表示该过滤器的执行顺序，值越小，执行优先级越高。需要注意在自定义GatewayFilter类型的过滤器时没有指定order，它的默认值是0（值越小优先级越高），如果想要设置多个过滤器的执行顺序，可以实现Ordered接口覆写getOrder方法。

# Spring Cloud Gateway集成Nacos实现请求负载

## 介绍

Nacos可以用于实现Spring Cloud Gateway中网关动态路由功能，也可以基于Nacos来 实现对后端服务的负载均衡。前者利用Nacos中心配置功能，后者利用Nacos服务注册

功能。如下图是Spring Cloud Gateway集成Nacos实现负载均衡的架构图：



## 实现

基于上图的架构图来演示一下整个实现过程，首先准备如下项目：

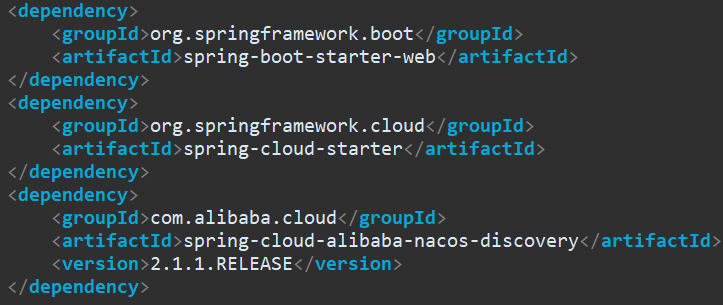
·spring-cloud-gateway-provider：提供Rest服务，并将服务注册到Nacos

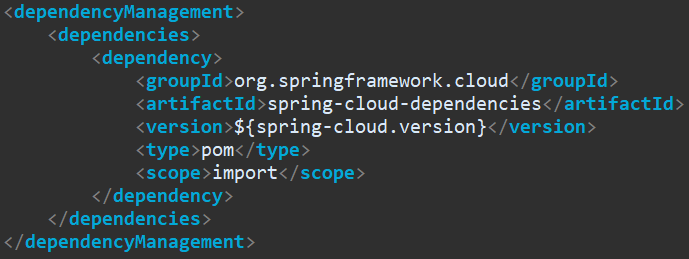
·spring-cloud-gateway-consumer：提供网关路由，基于Nacos服务注册中心

**spring-cloud-gateway-nacos-provider**

spring-cloud-gateway-nacos-provider服务的构建过程如下，该项目提供REST接口，并且将接口注册到Nacos服务器上。

1. 添加相关Jar包依赖，Spring Cloud使用的版本为Greenwich.SR2

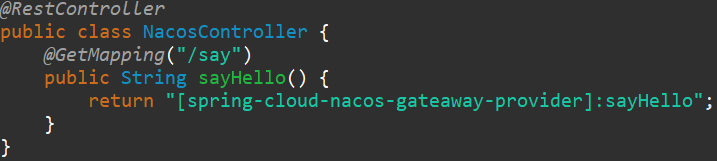




添加了nacos-disconvery依赖，该项目中就必须配置Nacos相关信息，否则启动

项目会提示“no server available”的错误。

1. 创建NacosController，提供一个say接口



1. 在application.yml中添加服务注册的地址



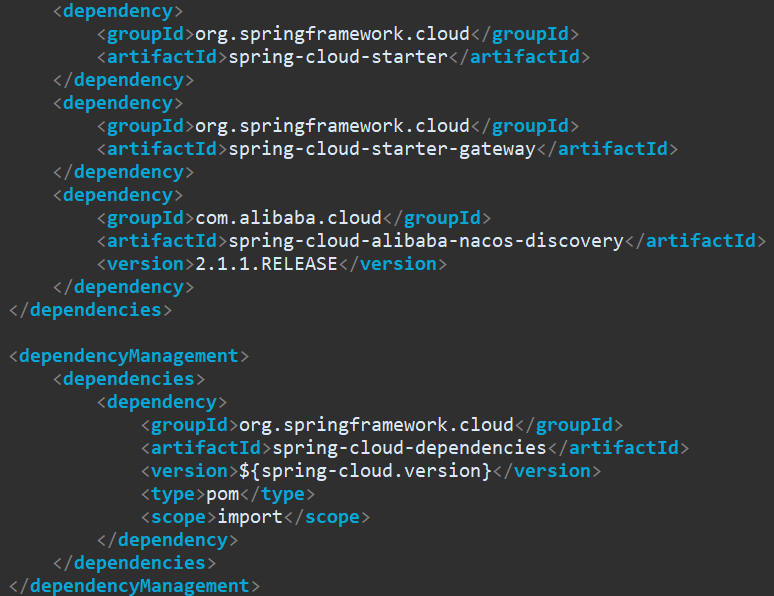
为了演示请求负载，将spring-cloud-nacos-gateway-provider部署两份，分别开放开放8080和8081接口（修改端口号启动即可）。开启Nacos服务器，服务启动成功之后，进入Nacos Dashboard的服务列表，可以看到如下图所示的服务信息：



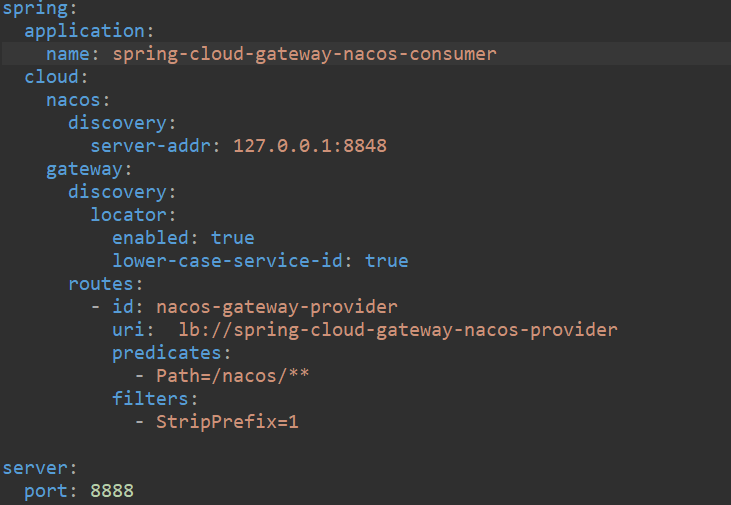
**spring-cloud-nacos-gateway-consumer**

spring-cloud-gateway-nacos -consumer作为网关，会从Nacos上根据服务名称获取目标URI进行服务调用。

1. 添加Jar依赖，Spring Cloud版本为Greenwich.SR2



1. 在application.yml中添加如下配置



关键配置说明如下：

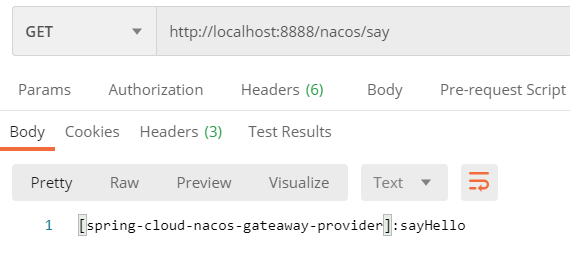
·lower-case-service-id：是否使用service-id的小写，默认是大写

·spring.cloud.gateway.locator.enabled：开启从注册中心动态创建路由的功能。

·uri：其中配置的lb://表示从注册中心获取服务，后面的spring-cloud-gateway-na

cos-provider表示目标服务在注册中心上的服务名。

1. 启动网关，通过Postman访问网关，如果配置成功，将会正确返回目标服务的内容。



# Spring Cloud Gateway集成Sentinel网关限流

## 介绍

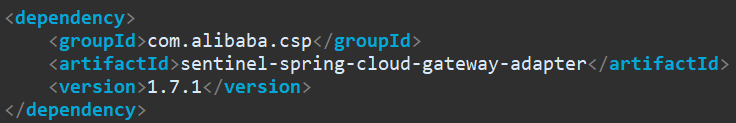
Sentinel从1.6.0版本开始，提供了Spring Cloud Gateway Adapter模块，支持两种资源维度的限流。

·Rote维度

·自定义API维度，可以利用提供的API来自定义API分组，然后针对这些分

组维度进行限流。

下面将基于上节Spring Cloud Gateway集成Nacos的两个项目代码进行改造，分别演示两种方式的限流配置。在演示前，在网关服务中引入如下依赖：



## Route维度限流

1. 在网关服务中创建一个配置类



该配置类主要完成的功能有：注入SentinelGatewayFilter，它属于全局过滤器即GlobalFilter类型的过滤器；注入限流异常处理器，限流异常处理器能够设置触发限流后返回什么内容；初始化限流规则，其中“nacos-gateway-provider”表示资源名称，它要与application.yml中指定的路由ID一致。其中Order注解可以决定该执行次序，值越小越小执行（前面也讲过），PostConstruct注解标注的方法会提前加载。该配置类的详细说明如下：

·注入一个全局限流过滤器SentinelGatewayFilter

·注入限流异常处理器

·初始化限流规则。在当前版本中，sentinel-spring-cloud-gateway-adapter仅支持手

动配置。

其中，GatewayFlowRule网关限流规则中提供了如下属性：

·resource：资源名称，可以是网关中的route的ID名称后者用户自定义的API

分组名称。（上述构造方法中的值就是resource）

·resourceMode：资源模型，限流规则是针对API Gateway的route（RESOUR

CE\_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

·cotrolBehavior：流量整形的控制效果，同限流规则的controlBehavior子段，

目前仅支持快速失败和均速排队两种模式，默认是快速失败。

·burst：应对突发请求时额外允许的请求数目。

·maxQueueingTimeoutMs：均速排队模式下的最长排队时间，单位是毫秒，

仅在均速排队模式下生效。

·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。其中的字段如

下：

·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\_STRATEG

Y\_HOST）、任意参数Header（PARAM\_PARSE\_STRATEGY\_HEADER）和任

意URL参数（PARAM\_PARSE\_STRATEGY\_URL\_PARAM）四种模式。

·filedName：若提取策略选择Header模式或URL参数模式，则需要制定

对应的Header名称或URL参数名称。

·pattern和matchStrategy：为后续参数匹配特性预留，目前未实现

网关限流规则的加载可以通过GatewayRuleManager.loadRules(rules);的方式手动加

载（如此配置类中），也可以通过GatewayRuleManager.register2Property(property)

注册动态限流规则，推荐使用这种动态的限流规则的方式。

1. 在application.yml文件中的配置如下



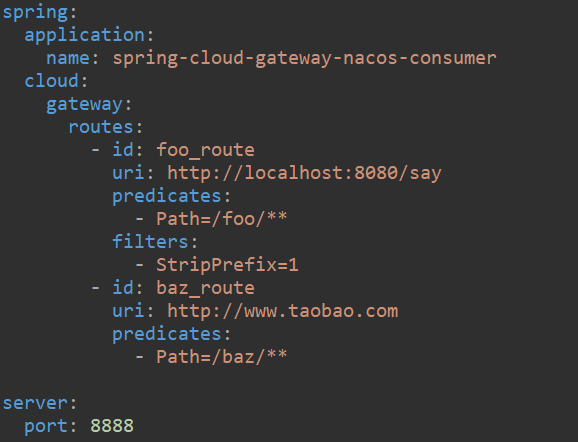
各个属性的含义都已经讲过，这里不再叙述。由于是基于上节的集成Nacos代码进行的改造，所以此处遗留了Nacos的相关配置。如果不集成nacos，则uri参数是一个路径，而不是Nacos注册中心上的一个微服务名称。

1. 最后，启动服务提供者和网关服务，这里通过Postman进行测试，当触发配置的限流规则后，会获得如下内容：



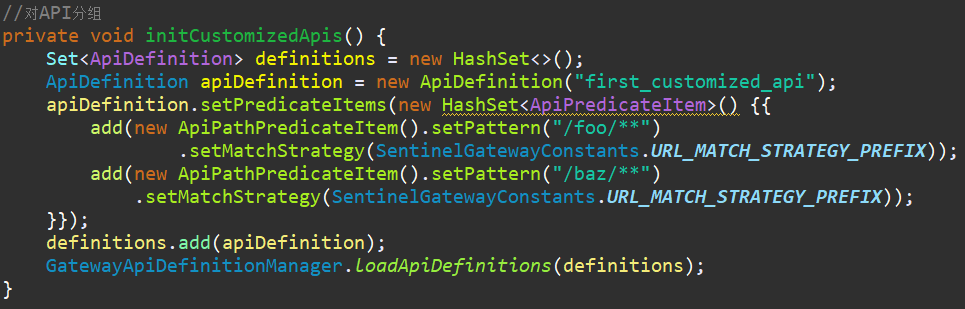
## 自定义API分组限流

自定义API分组限流实际上就是让多个Route共用一个限流规则。假设有如下两个URL匹配规则。

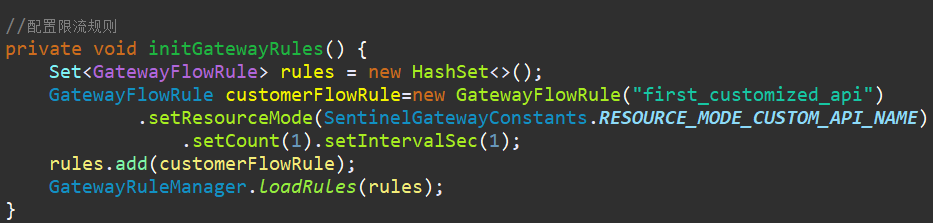


如果希望这两个路由共用同一个限流规则，则可以采用自定义API分组限流的方式来实

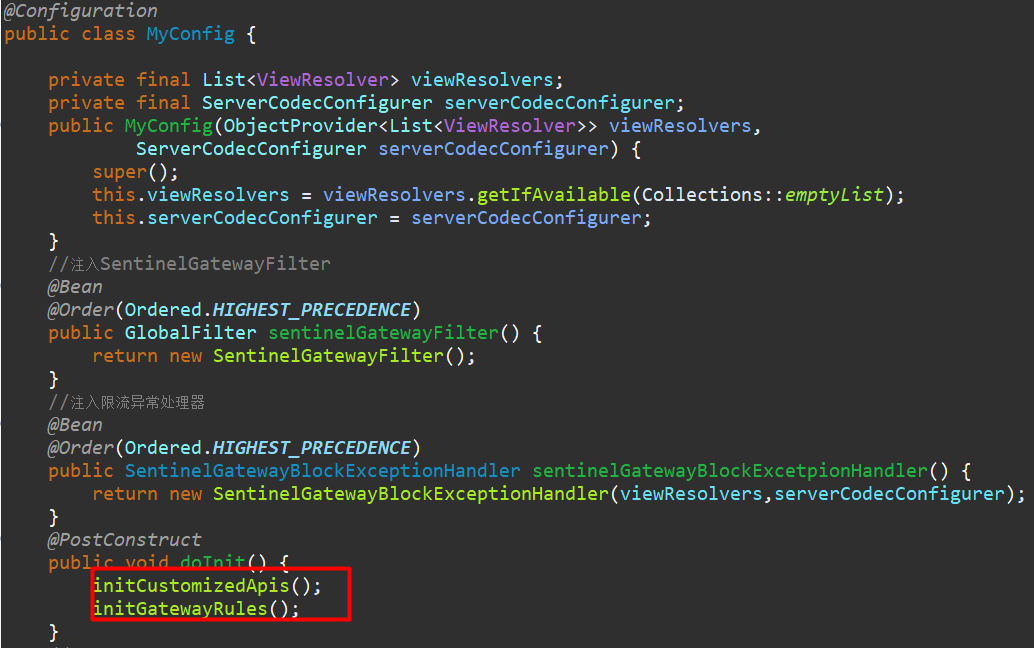
现（基于上节定义的配置类进行改造）。



上述代码主要是将/foo/\*\*和/baz/\*\*进行统一分组，并提供一个name=first\_customized\_api，然后可以针对该name设置限流规则（基于上节的配置类进行改造）：



先加载分组后加载限流规则：



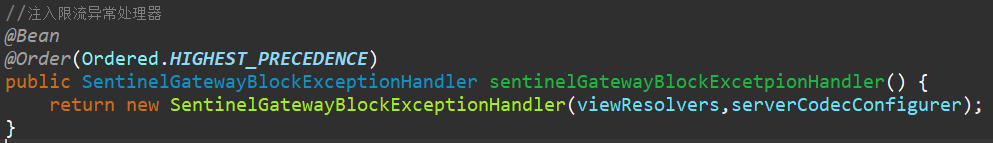
同时，我们可以通过setMatchStrategy来设置不同path下的限流参数策略。需要注意的是，在上述代码中，foo\_route和baz\_route这两个ID与first\_customized\_api都会标记位Sentinel的资源（限流资源标识）。比如，当访问网关的URI为<http://localhost:8888/foo/1>时，Sentinel会统计foo\_route、baz\_route、first\_customized\_api这些资源的流量情况。

## 自定义异常

当触发限流时，会返回如下这样的异常信息：

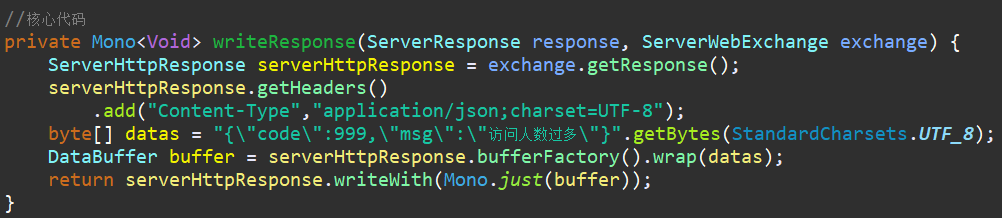


但在实际应用中，一般以JSON格式进行数据返回，那么怎么修改限流之后返回的数据格式呢？触发限流的默认处理类是通过注入限流异常处理器来实现的：

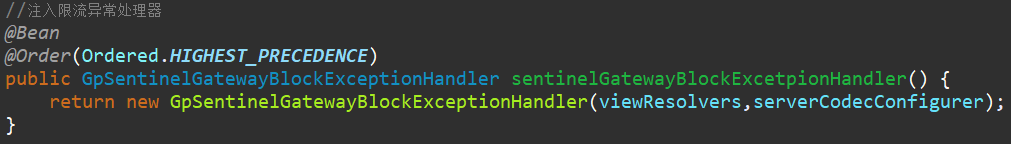


注入的是一个SentinelGatewayBlockExceptionHandler类，该类实现WebExceptionHandler

接口，这意味着我们可以实现该接口来自定义异常处理器实现消息格式的转化。接下来自定义一个限流异常处理器GpSentinelGatewayBlockExceptionHandler，所有的代码都可以直接从SentinelGatewayBlockExceptionHandler类中复制过来，我们只需要构造方法名和返回的变量的类名即可，核心是修改writeResponse方法，该方法的作用是将限流的异常信息写回客户端。



在配置类中注入自定义限流处理器：



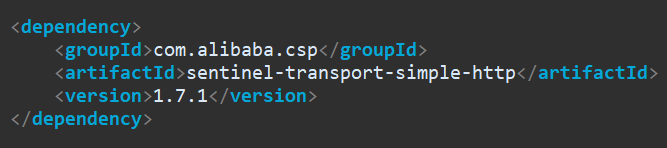
访问网关地址，触发限流，返回的限流异常信息如下：



## 网关流控控制台

Sentinel在1.6.3版本引入了网关流控控制台，使得我们可以在Sentinel控制台上查看API Gateway实时的Route和自定义API分组的监控，也可以在控制台上管理网关的流控规则和API分组配置。

1. 在网关服务中添加依赖



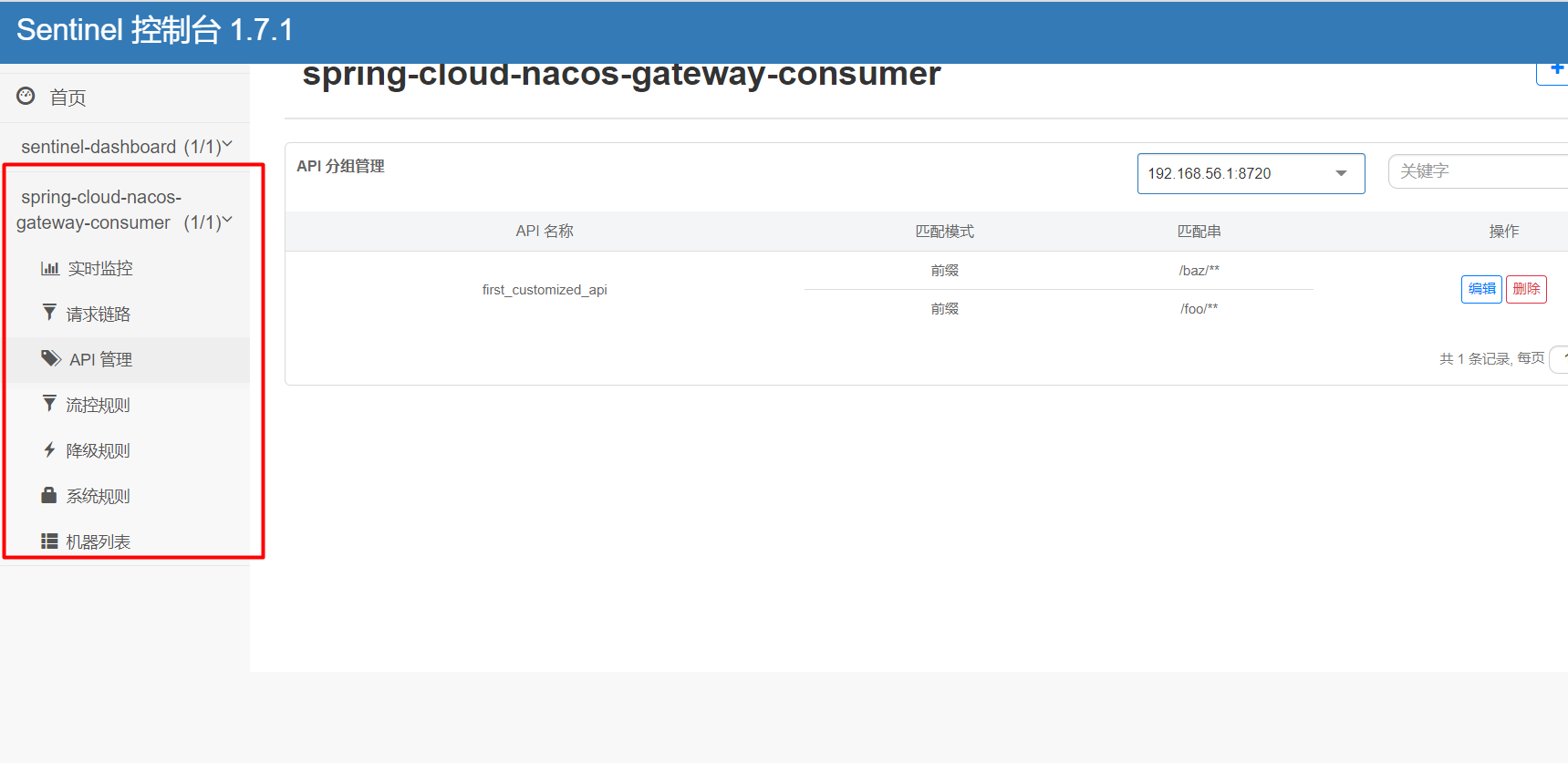
1. 启动Sentinel控制台



1. 将网关服务打成jar包，使用java命令启动



该命令中指定了Sentinel控制台的地址，同时需要添加启动参数-Dcsp.sentinel.app.type=1，它是在Spring Cloud Gateway接入Sentinel的时候才需要配置的，配置好之后Sentinel Dashboard会针对Gateway提供一个定制化的界面。启动完成之后，可以在Sentinel看到定制的网关监控服务：

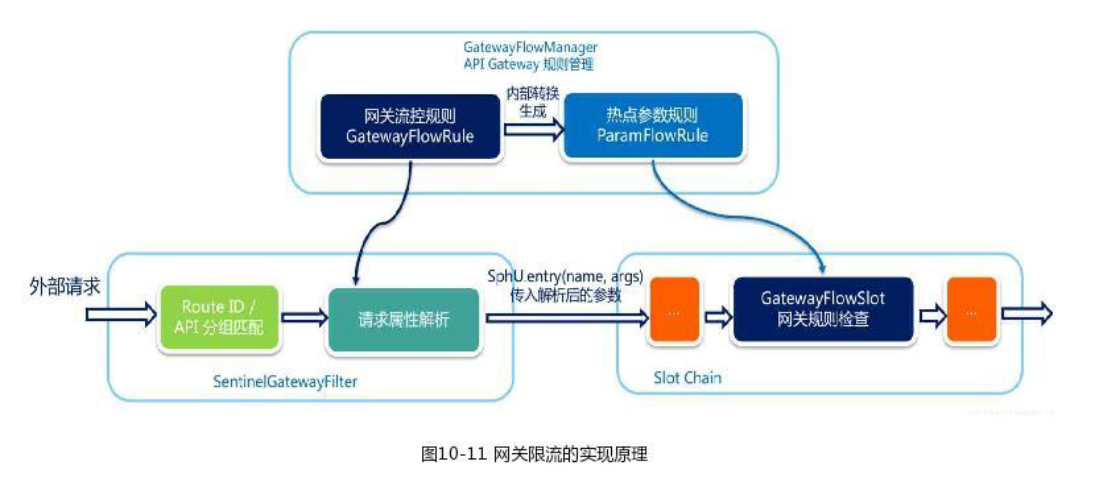


如果Sentinel Dashboard控制台没有相关服务，调用一下网关接口刷新Sentinel

Dashboard即可。

## 网关限流原理

网关限流的实现原理如下图所示：



·通过GatewayRuleManager加载网关限流规则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GatewayFlowRule转化成热点参

数规则ParamFlowRule存储在GatewayFlowManager中，与正常的热点参数规则相互

隔离。在转化时，Sentinel会根据请求属性配置，为网关流控规则设置参数索引（idx），

并添加到生成的热点参数规则中。

·在外部请求进入API网关时，会先进过SentinelGatewayFilter，在该过滤器中依次进行

Route ID/API分组匹配、请求属性解析和参数组装。

·Sentinel根据配置的网关限流规则来解析请求属性，并按照参数索引顺序组装参数数

组，最终传入SphU.entryy(name,args)中。

·在Sentinel API Gateway Adapter Common模块中在Slot Chain中添加了一个Gatewa

yFlowSlot，专门用于来处理网关限流规则的检查。

·如果当前限流规则并没有指定限流参数，则Sentinel会在参数的最后一个位置置入一

个预设的$D，最终实现普通限流。

实际上，在网关限流中，我们所配置的网关限流规则最终都会转化成参数限流规则，通

过ParamFlowChecker.passCheck进行参数限流规则检查。

# 本章小结

本章主要介绍了网关的发展过程，以及目前比较常见的API网关解决方案，通过分析发现，网关的本质是提供路由及过滤的功能。重点分析了Spring Cloud Gateway，它由三部分组成：

·路由（Route）

·谓语（Predicate）

·过滤器（Filter）

需要注意的是，Spring Cloud Gateway目前的版本只支持HTTPS、HTTP、WS、WSS这四种协议的转发，而Dubbo框架是基于Dubbo协议来实现的，因此暂时无法实现Dubbo服务的网关请求转发。