SpringCloud之十

路由网关GateWay

一、Gateway介绍

Spring Cloud Gateway是Spring官方基于Spring 5.0，Spring Boot 2.0和Project Reactor等技术开发的网关，Spring Cloud Gateway旨在为微服务架构提供一种简单而有效的统一的API路由管理方式。Spring Cloud Gateway作为Spring Cloud生态系中的网关，目标是替代Netflix ZUUL，其不仅提供统一的路由方式，并且基于Filter链的方式提供了网关基本的功能，例如：安全，监控/埋点，和限流等。

类似的网关技术有：NGINX、ZUUL、Spring Cloud Gateway、Linkerd……

Spring Cloud Gateway是一个很有前途的项目，上手简单，功能也比较强大。

Linkerd也是一个非常有前途的项目，是基于Scala实现的、目前市面上仅有的生产级别的Service Mesh（其他诸如Istio、Conduit暂时还不能用于生产）。

Zuul已经发布了Zuul 2.x，基于Netty，也是非阻塞的，支持长连接，但Spring Cloud暂时还没有整合计划。

API 网关

API 网关出现的原因是微服务架构的出现，不同的微服务一般会有不同的网络地址，而外部客户端可能需要调用多个服务的接口才能完成一个业务需求，如果让客户端直接与各个微服务通信，会有以下的问题：

1. 客户端会**多次请求**不同的微服务，增加了客户端的复杂性。
2. 存在**跨域请求**，在一定场景下处理相对复杂。
3. 认证复杂，每个服务都需要**独立认证**。
4. 难以**重构**，随着项目的**迭代**，可能需要重新划分微服务。例如，可能将多个服务合并成一个或者将一个服务拆分成多个。如果客户端直接与微服务通信，那么重构将会很难实施。
5. 某些微服务可能使用了**防火墙**/浏览器**不友好的协议**，直接访问会有一定的困难。

以上这些问题可以借助 API 网关解决。API 网关是**介于客户端和服务器端之间的中间层**，所有的外部请求都会先经过 API 网关这一层。也就是说，API 的实现方面更多的考虑业务逻辑，而安全、性能、监控可以交由 API 网关来做，这样既提高业务灵活性又不缺安全性，典型的架构图如图所示：

使用 API 网关后的优点如下：

- 易于监控。可以在网关收集监控数据并将其推送到外部系统进行分析。
- 易于认证。可以在网关上进行认证，然后再将请求转发到后端的微服务，而无须在每个微服务中进行认证。
- 减少了客户端与各个微服务之间的交互次数。

NGINX 服务

Nginx 由内核和模块组成，内核的设计非常微小和简洁，完成的工作也非常简单，仅仅通过查找配置文件与客户端请求进行 URL 匹配，用于启动不同的模块去完成相应的工作。

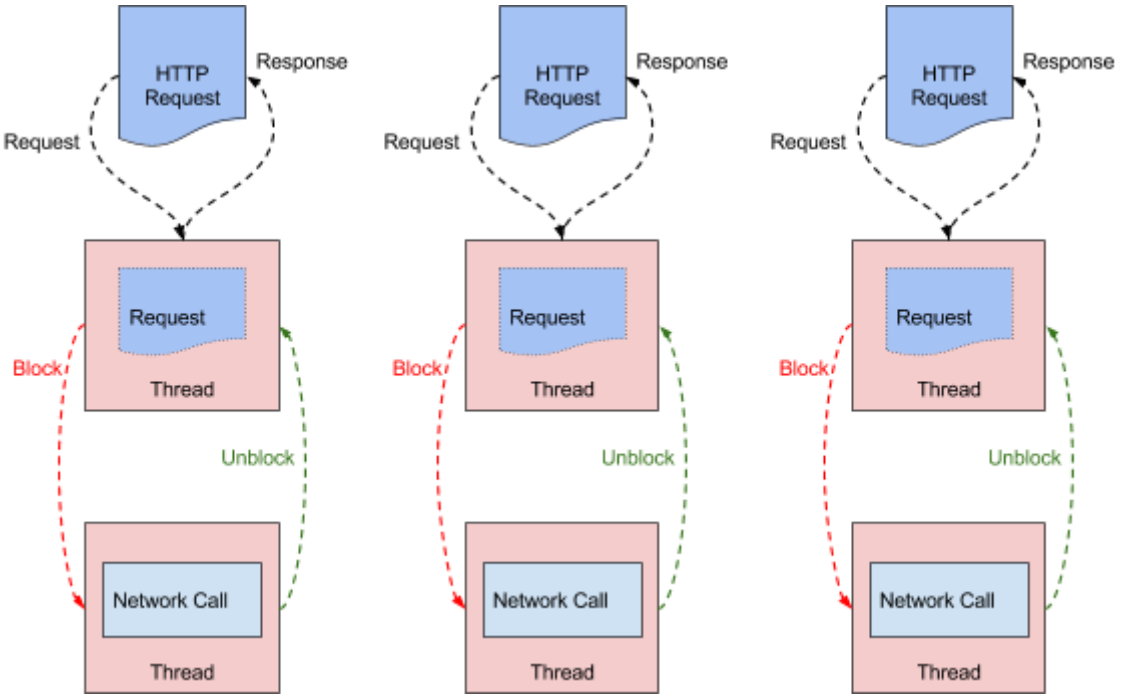
Netflix 的 Zuul

Zuul 是 Netflix 开源的微服务网关组件，它可以和 Eureka、Ribbon、Hystrix 等组件配合使用。Zuul 的核心是一系列的过滤器，这些过滤器可以完成以下功能：

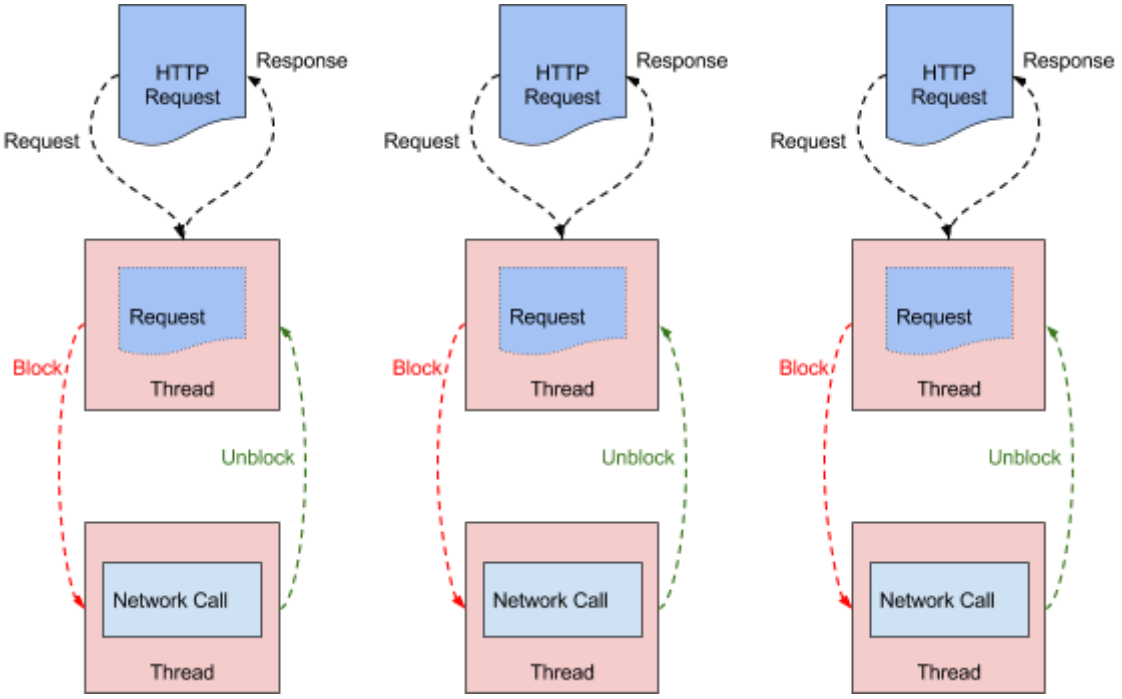
- 身份认证与安全：识别每个资源的验证要求，并拒绝那些与要求不符的请求。
- 审查与监控：与边缘位置追踪有意义的数据和统计结果，从而带来精确的生产视图。
- 动态路由：动态地将请求路由到不同的后端集群。
- 压力测试：逐渐增加指向集群的流量，以了解性能。
- 负载分配：为每一种负载类型分配对应容量，并弃用超出限定值的请求。
- 静态响应处理：在边缘位置直接建立部分响应，从而避免其转发到内部集群。
- 多区域弹性：跨越 AWS Region 进行请求路由，旨在实现 ELB（Elastic Load Balancing，弹性负载均衡）使用的多样化，以及让系统的边缘更贴近系统的使用者。

上面提及的这些特性是 Nigix 所没有的，这是因为 Netflix 公司创造 Zuul 是为了解决云端的诸多问题（特别是帮助 AWS 解决跨 Region 情况下的这些特性实现），而不仅仅是做一个类似于 Nigix 的反向代理，当然，我们可以仅使用反向代理功能，这里不多做描述。

Zuul1 是基于 Servlet 框架构建，如图所示，采用的是阻塞和多线程方式，即一个线程处理一次连接请求，这种方式在内部延迟严重、设备故障较多情况下会引起存活的连接增多和线程增加的情况发生。



Zuul2 的巨大区别是它运行在异步和无阻塞框架上，每个 CPU 核一个线程，处理所有的请求和响应，请求和响应的生命周期是通过事件和回调来处理的，这种方式减少了线程数量，因此开销较小。又由于数据被存储在同一个 CPU 里，可以复用 CPU 级别的缓存，前面提及的延迟和重试风暴问题也通过队列存储连接数和事件数方式减轻了很多（较线程切换来说轻量级很多，自然消耗较小）。这一变化一定会大大提升性能，我们在后面的测试环节看看结果。



至于zuul部署，以及注册到eureka上可以参考

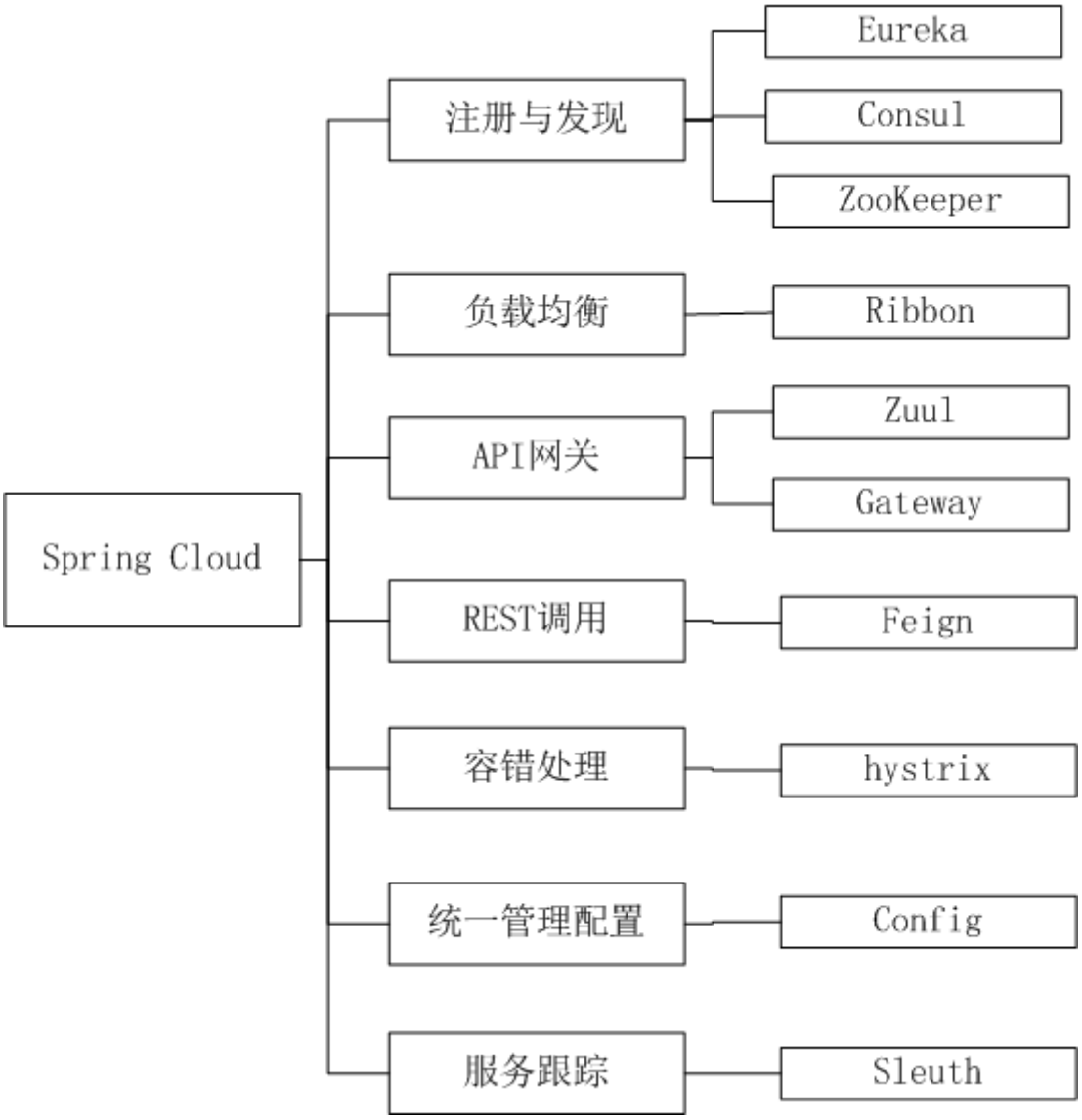
https://www.sohu.com/a/221110905_467759

SpringCloud

虽然 Spring Cloud 带有“Cloud”，但是它并不是针对云计算的解决方案，而是在 Spring Boot 基础上构建的，用于快速构建分布式系统的通用模式的工具集。

使用 Spring Cloud 开发的应用程序非常适合在 Docker 或者 PaaS 上部署，所以又叫云原生应用。云原生可以简单理解为面向云环境的软件架构。

既然是工具集，那么它一定包含很多工具，我们来看下面这张图：



这里由于仅涉及到 API 网关的对比，因此我不逐一介绍其他工具了。

Spring Cloud 对 Zuul 进行了整合，但从 Zuul 来看，没有大变化，但是 Spring Cloud 整个框架经过了组件的集成，提供的功能远多于 Netflix Zuul，可能对比时会出现差异。

Service Mesh 之 Linkerd

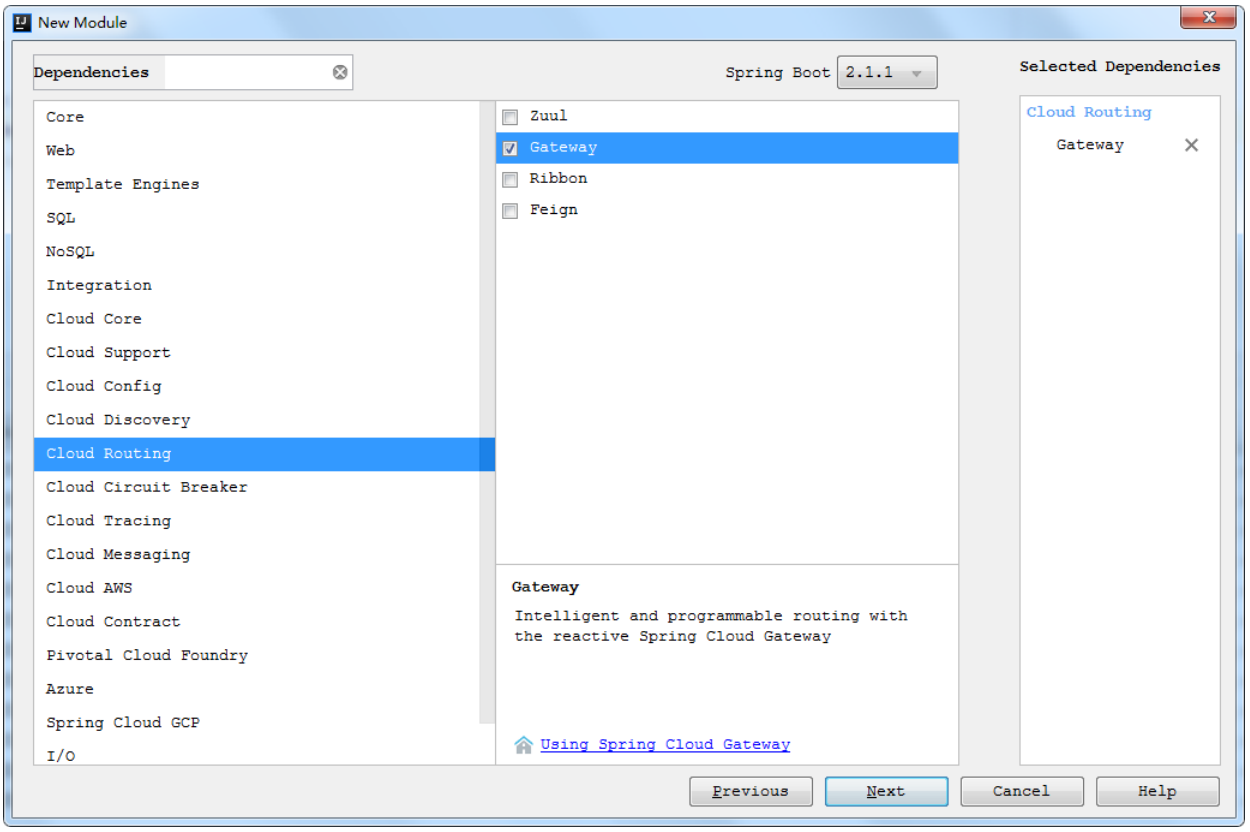
Linkerd 为云原生应用提供弹性的 Service Mesh，而 Service Mesh 能够提供轻量级高性能网络代理，并且也提供微服务框架支撑。linkerd 是面向微服务的开源 RPC 代理，它直接立足于 Finagle（Twitter 的内部核心库，负责管理不同服务间之通信流程。

和 Spring Cloud 类似，Linkerd 也提供了负载均衡、熔断机器、服务发现、动态请求路由、重试和离线、TLS、HTTP 网关集成、透明代理、gRPC、分布式跟踪、运维等诸多功能，功能是相当全了，为微服务框架的技术选型又增加了一个。

二、Gateway的使用

2.1、创建项目

创建gate-service项目，



pom.xml的内容如下：

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apach
e.org/xsd/maven-4.0.0.xsd">
4    <modelVersion>4.0.0</modelVersion>
5    <groupId>xyz.jiangnanke</groupId>
6    <artifactId>gateway-service</artifactId>
7    <version>0.0.1-SNAPSHOT</version>
8    <name>gateway-service</name>
9    <description>Demo project for Spring Boot</description>
10
11    <parent>
12    <groupId>xyz.jiangnanke</groupId>
13    <artifactId>main</artifactId>
```

```
<version>0.0.1-SNAPSHOT</version>
</parent>

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

</dependencies>

</project>
```

并在main的pom.xml 里面添加上父子项目的关联。

然后有两种方式来进行路由配置：

1、类

如下RouterConfig.java:

```
package xyz.jiangnanke.gatewayservice.config;

import org.springframework.cloud.gateway.route.RouteLocator;
import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilde
r;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @Auther: zhengfeng
 * @Date: 2019\1\10 0010 14:41
 * @Description:
 */
@Configuration
public class RouterConfig {

 @Bean
 public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
 return builder.routes()
 //basic proxy
 .route(r -> r.path("/baidu")
 .uri("http://baidu.com:80/")
 ).build();
 }
```

```
24
25
26 }
```

有编译报错没有关系的。

2、application.yml

内容如下：

```
1
2 spring:
3  application:
4  name: gateway-service
5  cloud:
6  gateway:
7  routes:
8  - id: jiangnanke_route
9  uri: http://www.baidu.com/
10  predicates:
11  - Path=/jiangnanke
12
13 server:
14  port: 8780
15
16 logging:
17  level:
18  org.springframework.cloud.gateway: TRACE
19  org.springframework.http.server.reactive: DEBUG
20  org.springframework.web.reactive: DEBUG
21  reactor.ipc.netty: DEBUG
```

上面spring.cloud.gateway.routes中的属性中：

id：固定，不同 id 对应不同的功能，可参考 官方文档

uri：目标服务地址

predicates：路由条件

filters：过滤规则

2.2、启动运行

无论访问：http://localhost:8780/jiangnanke 还是 http://localhost:8780/baidu，都会自动跳到：https://www.baidu.com/

由于没有动态图，所以没法贴图在这里。

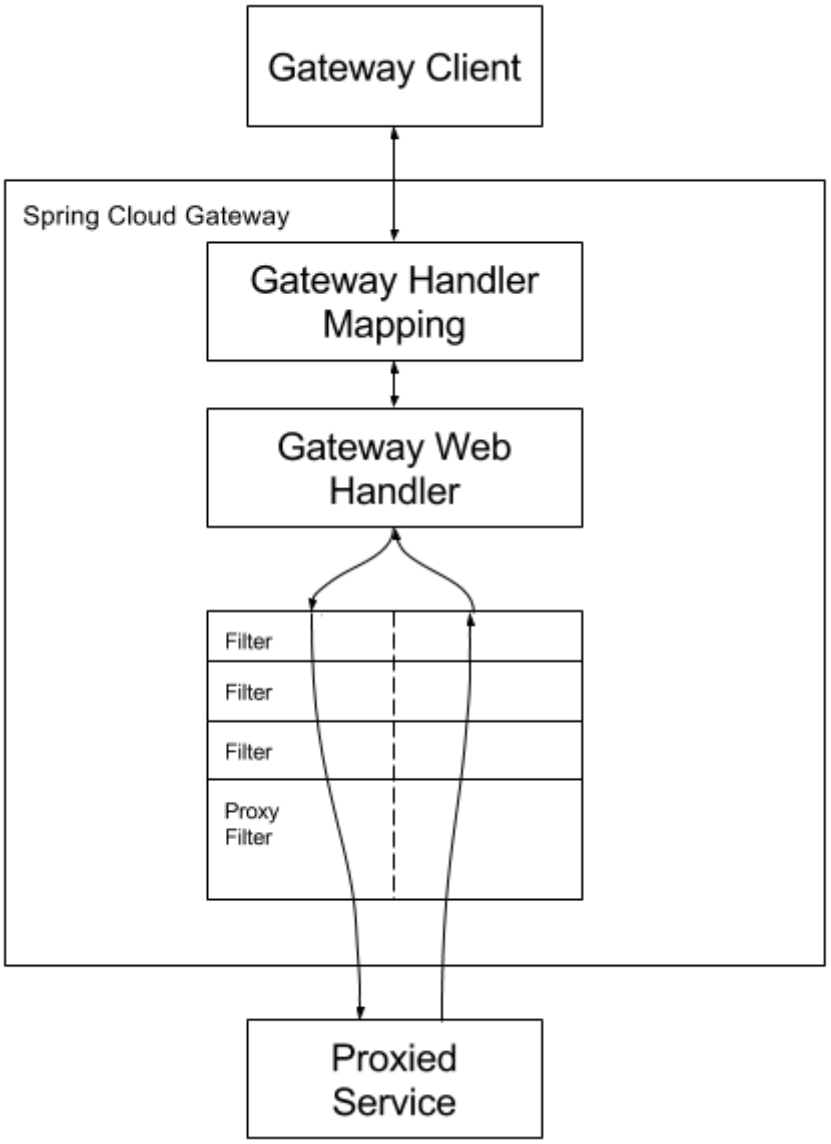
三、Gateway Predicate

网关作为一个系统的流量的入口，有着举足轻重的作用，通常的作用如下：

- 协议转换，路由转发
- 流量聚合，对流量进行监控，日志输出
- 作为整个系统的前端工程，对流量进行控制，有限流的作用
- 作为系统的前端边界，外部流量只能通过网关才能访问系统
- 可以在网关层做权限的判断
- 可以在网关层做缓存

Spring Cloud Gateway作为Spring Cloud框架的第二代网关，在功能上要比Zuul更加的强大，性能也更好。随着Spring Cloud的版本迭代，Spring Cloud官方有打算弃用Zuul的意思。在笔者调用了Spring Cloud Gateway的使用和功能上，Spring Cloud Gateway替换掉Zuul的成本上是非常低的，几乎可以无缝切换。Spring Cloud Gateway几乎包含了zuul的所有功能。

官网图片：



如上图所示，客户端向Spring Cloud Gateway发出请求。 如果Gateway Handler Mapping确定请求与路由匹配（这个时候就用到predicate），则将其发送到Gateway web handler处理。 Gateway web handler处理请求时会经过一系列的过滤器链。 过滤器链被虚线划分的原因是过滤器链可以在发送代理请求之前或之后执行过滤逻辑。 先执行所有“pre”过滤器逻辑，然后进行代理请求。 在发出代理请求之后，收到代理服务的响应之后执行“post”过滤器逻辑。这跟zuul的处理过程很类似。在执行所有“pre”过滤器逻辑时，往往进行了鉴权、限流、日志输出等功能，以及请求头的更改、协议的转换；转发之后收到响应之后，会执行所有“post”过滤器的逻辑，在这里可以响应数据进行了修改，比如响应头、协议的转换等。

在上面的处理过程中，有一个重要的点就是将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。

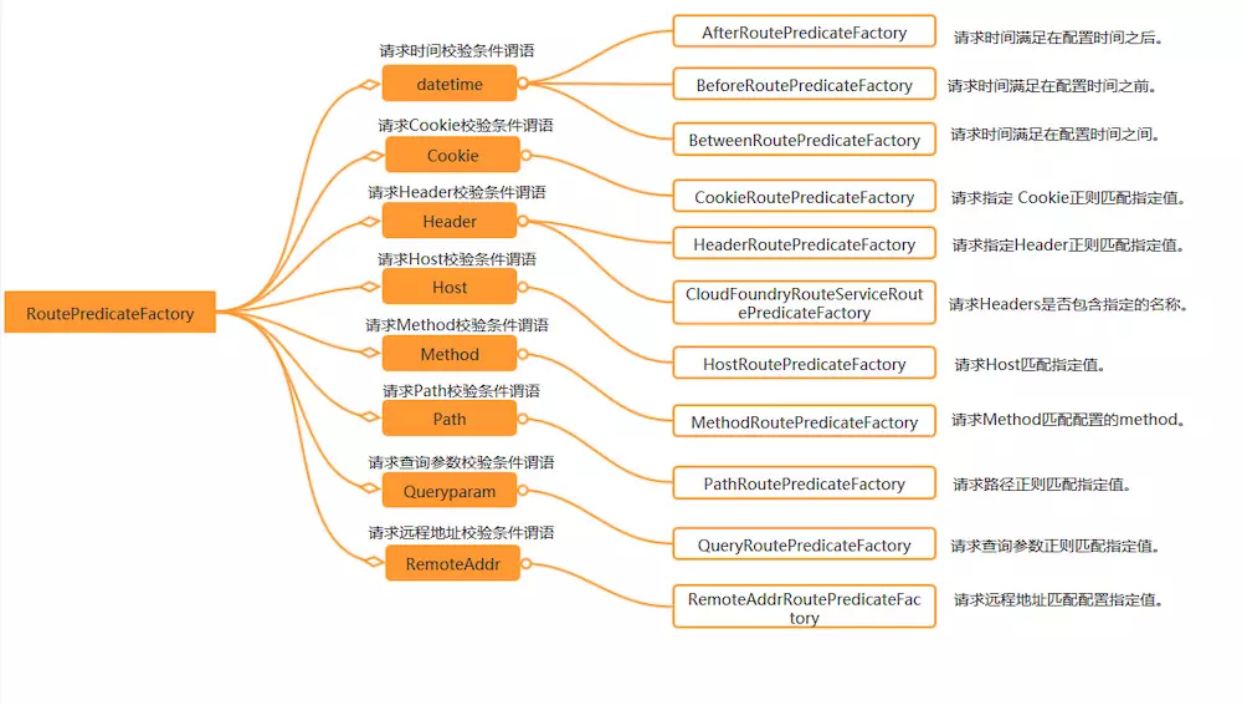
3.1、predicate 介绍

官方中描述：

https://cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.1.0.RC3/single/spring-cloud-gateway.html

> 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，如果读者有兴趣可以阅读一下。现在列举各种Predicate如下图（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。

3.2、predicate 使用

使用如下：

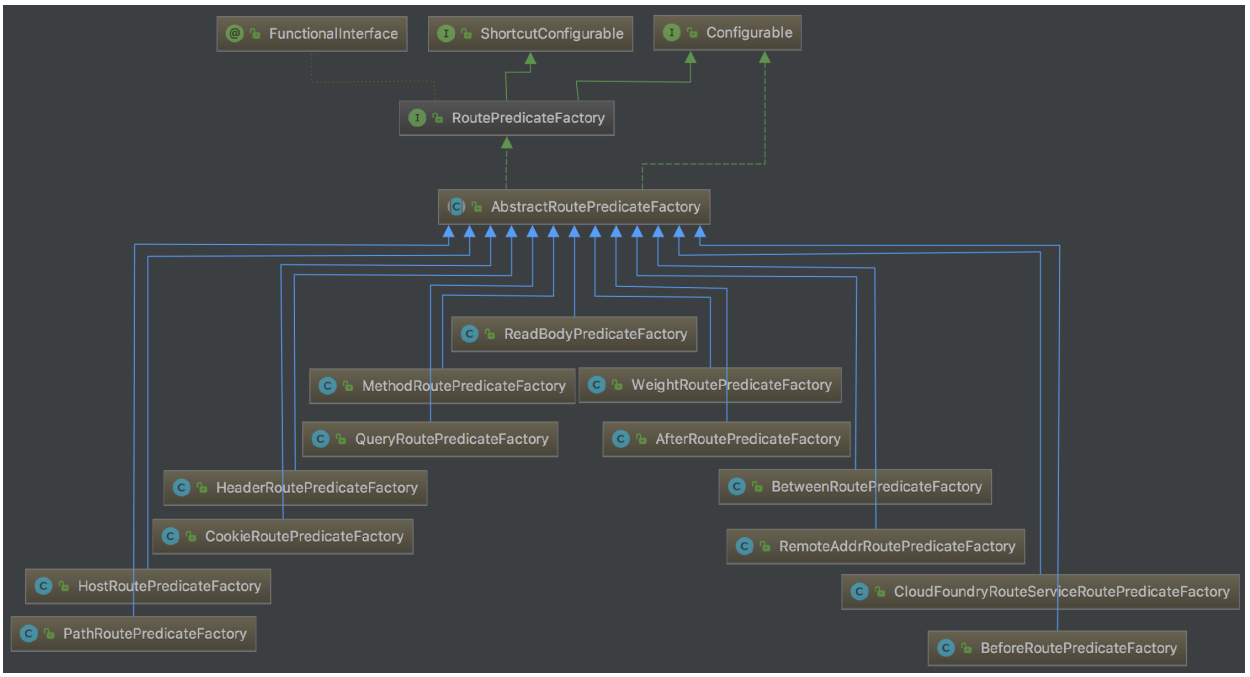
在application.yml文件中配置如下内容：

```
# After Route Predicate Factory:此谓词匹配当前日期时间之后发生的请求。
 - id: after_route
 uri: http://example.org
 predicates:
 - After=2017-01-20T17:42:47.789-07:00[America/Denver]
# Before Route Predicate Factory:此谓词匹配在当前日期时间之前发生的请求。
 - id: before_route
 uri: http://example.org
 predicates:
  - Before=2017-01-20T17:42:47.789-07:00[America/Denver]
 # Between Route Predicate Factory:此谓词匹配datetime1之后和datetime2之前发生的请求。 datetime2参数必须在datetime1之后。
  - id: between_route
  uri: http://example.org
  predicates:
  - Between=2017-01-20T17:42:47.789-07:00[America/Denver], 2017-01-21T17:42:47.789-07:00[America/Denver]
 # Cookie Route Predicate Factory:有两个参数，cookie名称和正则表达式。此谓词匹配具有给定名称且值与正则表达式匹配的cookie。
  - id: cookie_route
  uri: http://example.org
  predicates:
 # 此路由匹配请求有一个名为chocolate的cookie，其值与ch.p正则表达式匹配。
  - Cookie=chocolate, ch.p
 # Header Route Predicate Factory:有两个参数，标题名称和正则表达式。此谓词与具有给定名称且值与正则表达式匹配的标头匹配。
  - id: header_route
  uri: http://example.org
  predicates:
 #如果请求具有名为X-Request-Id的标头，则该路由匹配，其值与\ d +正则表达式匹配（具有一个或多个数字的值）。
  - Header=X-Request-Id, \d+
```

```
28  # Host Route Predicate Factory:采用一个参数：主机名模式。该模式是一种Ant样式模式“.”作为分隔符。此谓词匹配与模式匹配的Host标头。
29    - id: host_route
30    uri: http://example.org
31    predicates:
32  # 如果请求的主机头具有值www.somehost.org或beta.somehost.org，则此路由将匹配。
33    - Host=**.somehost.org
34  # Method Route Predicate Factory:采用一个参数：要匹配的HTTP方法。
35    - id: method_route
36    uri: http://example.org
37    predicates:
38    - Method=GET
39
40    - id: host_route
41    uri: http://example.org
42    predicates:
43  # 可以匹配 /foo/1 or /foo/bar.
44    - Path=/foo/{segment}
45  # Query Route Predicate Factory：有两个参数：一个必需的参数和一个可选的正则表达式。
46    - id: query_route
47    uri: http://example.org
48    predicates:
49  # 如果请求包含baz查询参数，则此路由将匹配。
50    - Query=baz
51    - id: query_route
52    uri: http://example.org
53    predicates:
54  # 如果请求包含其值与ba匹配的foo查询参数，则此路由将匹配。 regexp，所以bar和baz匹配。
55    - Query=foo, ba.
56  # RemoteAddr Route Predicate Factory采用CIDR符号（IPv4或IPv6）字符串的列表（最小值为1），例如， 192.168.0.1/16（其中192.168.0.1是IP地址，16是子网掩码）。如果请求的远程地址是例如192.168.1.10，则此路由将匹配。
57    - id: remoteaddr_route
58    uri: http://example.org
59    predicates:
60    - RemoteAddr=192.168.1.1/24
```

测试的话可以使用postman进行测试：

Spring-Cloud-Gateway通过RoutePredicateFactory创建Predicate。其中预制了很多RoutePredicateFactory使其可以通过简单的配置就可以创建出理想的Predicate。
查看类Uml图

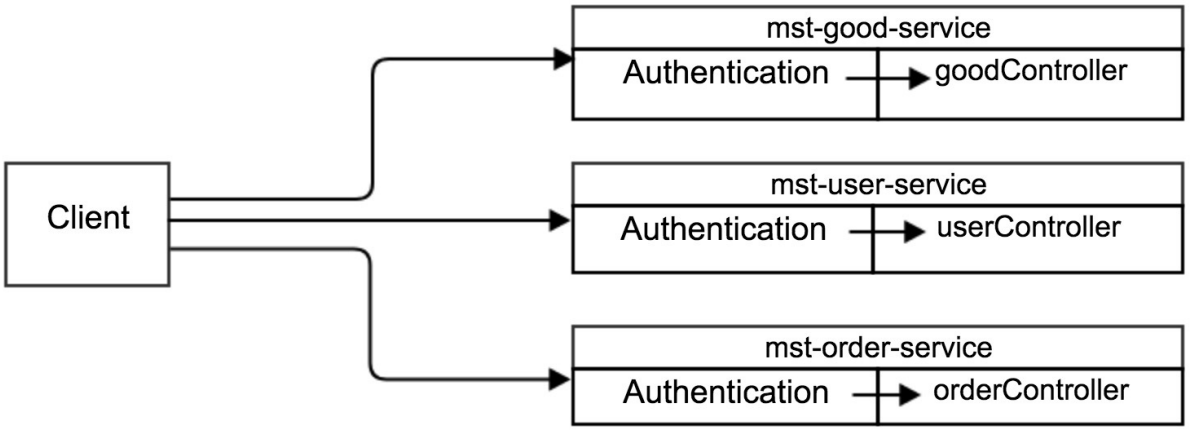


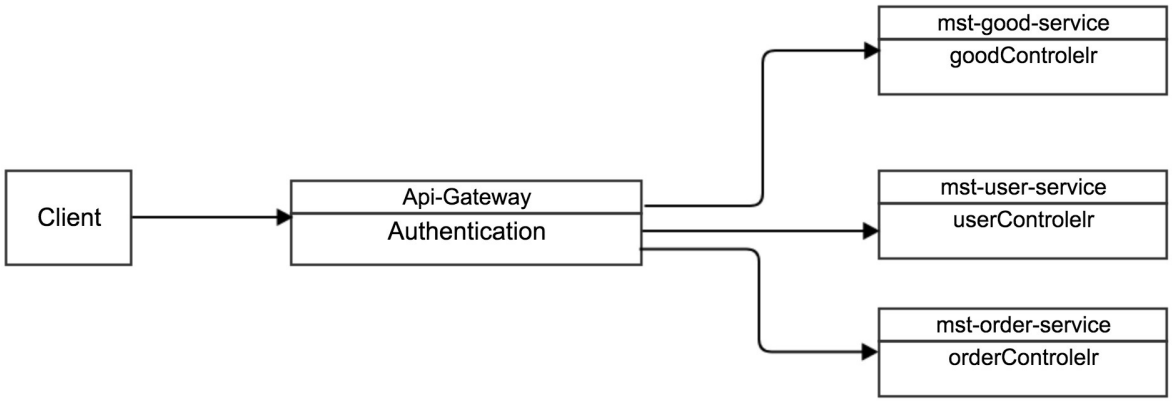
四、Gateway Filter

由filter工作流程点，可以知道filter有着非常重要的作用，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。

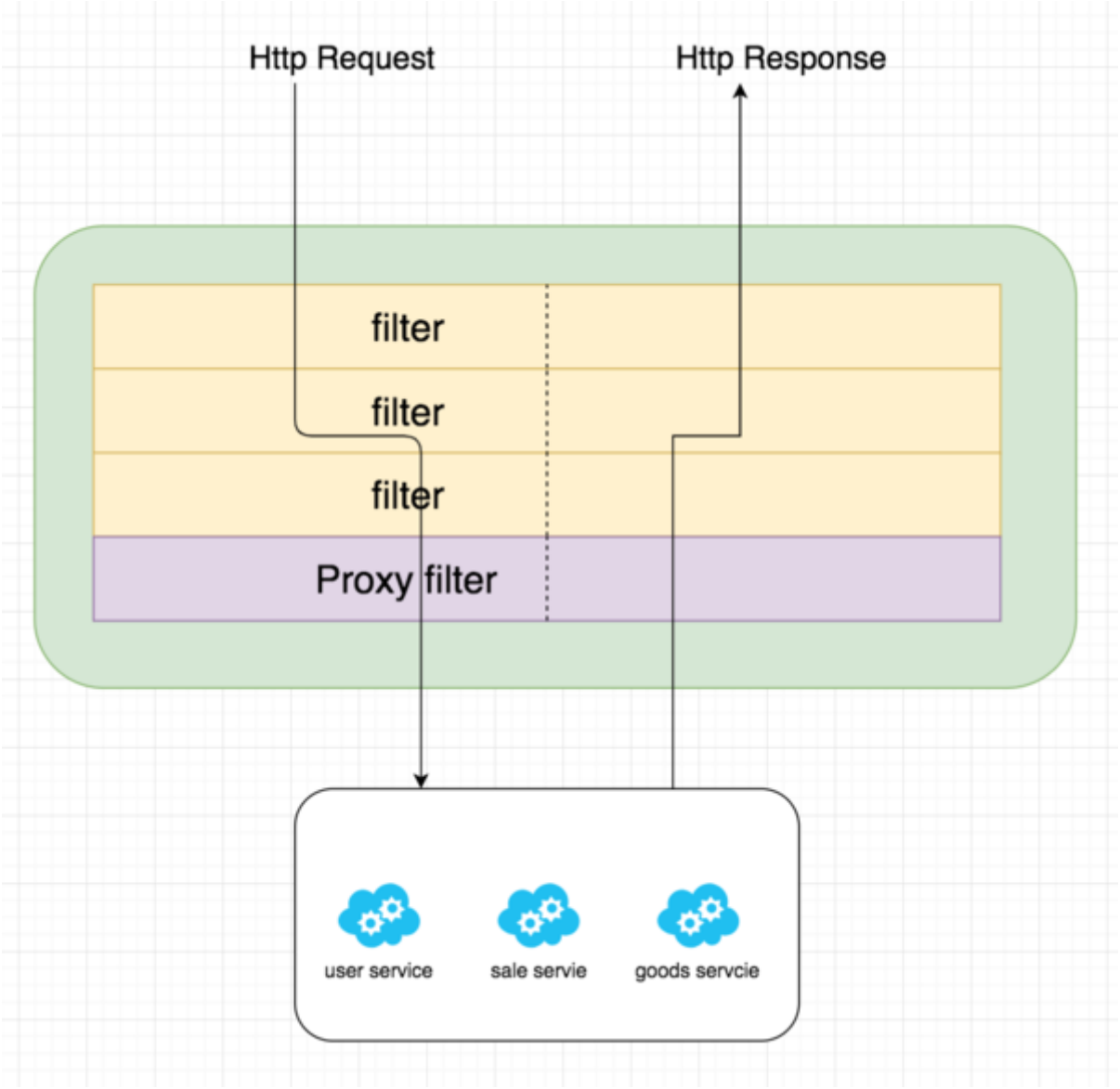
4.1、Gateway Filter 介绍

Spring Cloud Gateway同zuul类似，有“pre”和“post”两种方式的filter。客户端的请求先经过“pre”类型的filter，然后将请求转发到具体的业务服务，比如图中的user-service，收到业务服务的响应之后，再经过“post”类型的filter处理，最后返回响应到客户端。





filter作用如下图，



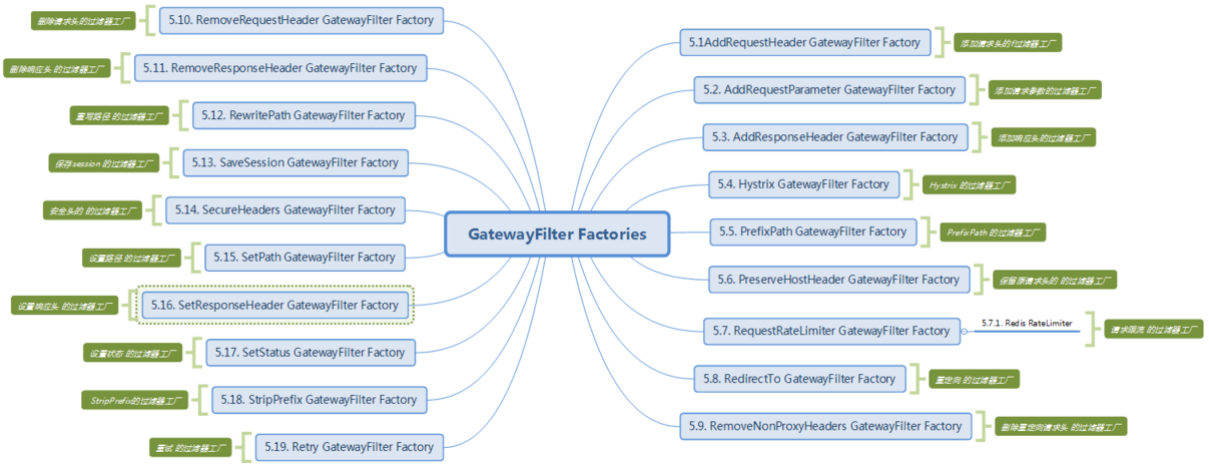
与zuul不同的是，filter除了分为“pre”和“post”两种方式的filter外，在Spring Cloud Gateway中，filter从作用范围可分为另外两种，一种是针对于单个路由的gateway filter，它在配置文件中的写法同predict类似；另外一种是针对于所有路由的global gateway filer。现在从作用范围划分的维度来讲解这两种filter。

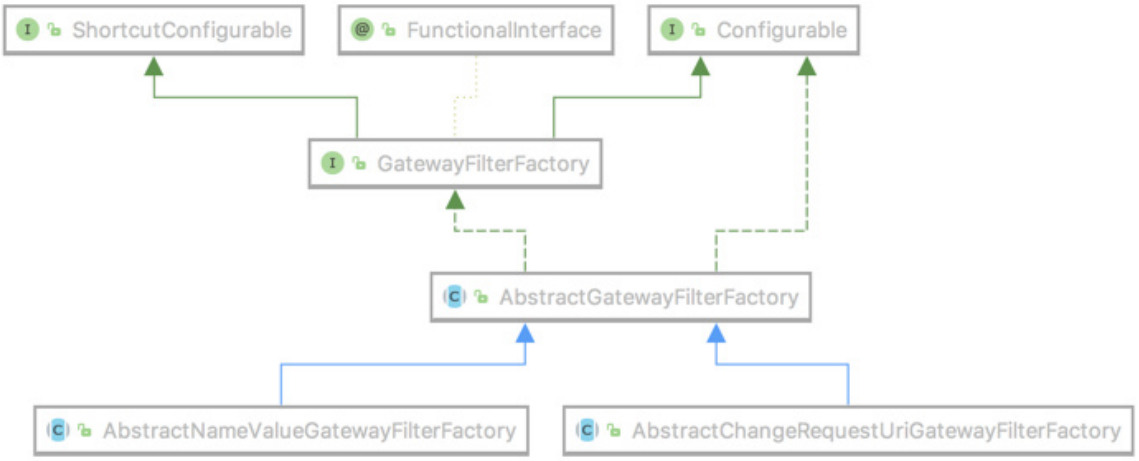
Filter过滤器允许以某种方式修改传入的HTTP请求或传出的HTTP响应。过滤器可以限定作用在某些特定请求路径上。 Spring Cloud Gateway包含许多内置的GatewayFilter工厂。

GatewayFilter工厂同上介绍的Predicate工厂类似，都是在配置文件application.yml中配置，遵循了约定大于配置的思想，只需要在配置文件配置GatewayFilter Factory的名称，而不需要写全部的类名，比如AddRequestHeaderGatewayFilterFactory只需要在配置文件中写

AddRequestHeader，而不是全部类名。在配置文件中配置的GatewayFilter Factory最终都会相应的过滤器工厂类处理。

Spring Cloud Gateway 内置的过滤器工厂一览表如下：





过滤器工厂的顶级接口是GatewayFilterFactory，我们可以直接继承它的两个抽象类来简化开发AbstractGatewayFilterFactory和AbstractNameValueGatewayFilterFactory，这两个抽象类的区别就是前者接收一个参数（像StripPrefix和我们创建的这种），后者接收两个参数（像AddResponseHeader）。

在Spring-Cloud-Gateway之请求处理流程中最终网关是将请求交给过滤器链表进行处理。

核心接口：GatewayFilter，GlobalFilter，GatewayFilterChain。

4.1.1、GatewayFilterChain--网关过滤链表

```
1  /**
```

```
 * 网关过滤链表接口
 * 用于过滤器的链式调用
 */
public interface GatewayFilterChain {

 /**
 * 链表启动调用入口方法*/
 Mono<Void> filter(ServerWebExchange exchange);

}
```

对应的默认实现是:

```
/**
 * 网关过滤的链表，用于过滤器的链式调用
 * 过滤器链表接口的默认实现，
 * 包含2个构建函数：
 * 1.集合参数构建用于初始化吧构建链表
 * 2. index，parent参数用于构建当前执行过滤对应的下次执行的链表
 */
 private static class DefaultGatewayFilterChain implements GatewayFilterC
hain {

  /**
  * 当前过滤执行过滤器在集合中索引
  */
  private final int index;
  /**
  * 过滤器集合
  */
  private final List<GatewayFilter> filters;

  public DefaultGatewayFilterChain(List<GatewayFilter> filters) {
  this.filters = filters;
  this.index = 0;
  }

  /**
  * 构建
  * @param parent 上一个执行过滤器对应的FilterChain
  * @param index 当前要执行过滤器的索引
  */
```

```
29  private DefaultGatewayFilterChain(DefaultGatewayFilterChain parent, int
index) {
30  this.filters = parent.getFilters();
31  this.index = index;
32  }
33
34  public List<GatewayFilter> getFilters() {
35  return filters;
36  }
37
38  /**
39  * @param exchange the current server exchange
40  * @return
41  */
42  @Override
43  public Mono<Void> filter(ServerWebExchange exchange) {
44  return Mono.defer(() -> {
45  if (this.index < filters.size()) {
46  //获取当前索引的过滤器
47  GatewayFilter filter = filters.get(this.index);
48  //构建当前索引的下一个过滤器的FilterChain
49  DefaultGatewayFilterChain chain = new DefaultGatewayFilterChain(this, t
his.index + 1);
50  //调用过滤器的filter方法执行过滤器
51  return filter.filter(exchange, chain);
52  } else {
53  //当前索引大于等于过滤集合大小，标识所有链表都已执行完毕，返回空
54  return Mono.empty(); // complete
55  }
56  });
57  }
58  }
```

过滤器的GatewayFilterChain 执行顺序

1. 通过GatewayFilter集合构建顶层的GatewayFilterChain
2. 调用顶层GatewayFilterChain，获取第一个Filter，并创建下一个Filter索引对应的GatewayFilterChain
3. 调用filter的filter方法执行当前filter，并将下次要执行的filter对应GatewayFilterChain传入。

4.1.2、GatewayFilter--网关路由过滤器

```java
/**
 * 网关路由过滤器，
 * Contract for interception-style, chained processing of Web requests that may
 * be used to implement cross-cutting, application-agnostic requirements such
 * as security, timeouts, and others. Specific to a Gateway
 *
 */
public interface GatewayFilter extends ShortcutConfigurable {

 String NAME_KEY = "name";
 String VALUE_KEY = "value";

 /**
 * 过滤器执行方法
 * Process the Web request and (optionally) delegate to the next
 * {@code WebFilter} through the given {@link GatewayFilterChain}.
 * @param exchange the current server exchange
 * @param chain provides a way to delegate to the next filter
 * @return {@code Mono<Void>} to indicate when request processing is complete
 */
 Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain);

}
```

网关过滤器接口，有且只有一个方法filter，执行当前过滤器，并在此方法中决定过滤器链表是否继续往下执行。

1/OrderedGatewayFilter--排序

OrderedGatewayFilter实现类主要目的是为了将目标过滤器包装成可排序的对象类型。是目标过滤器的包装类

```java
/**
 * 排序的网关路由过滤器，用于包装真实的网关过滤器，已达到过滤器可排序
 */
public class OrderedGatewayFilter implements GatewayFilter, Ordered {

 //目标过滤器
 private final GatewayFilter delegate;
 //排序字段
 private final int order;
```

```
10
11   public OrderedGatewayFilter(GatewayFilter delegate, int order) {
12   this.delegate = delegate;
13   this.order = order;
14   }
15
16   @Override
17   public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
18   return this.delegate.filter(exchange, chain);
19   }
20  }
```

2/GatewayFilterAdapter

GatewayFilterAdapter实现类主要目的是为了将GlobalFilter过滤器包装成GatewayFilter类型的对应。是GlobalFilter过滤器的包装类

```
1  /**
2   * 全局过滤器的包装类，将全局路由包装成统一的网关过滤器
3   */
4   private static class GatewayFilterAdapter implements GatewayFilter {
5
6   /**
7   * 全局过滤器
8   */
9   private final GlobalFilter delegate;
10
11   public GatewayFilterAdapter(GlobalFilter delegate) {
12   this.delegate = delegate;
13   }
14
15   @Override
16   public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
17   return this.delegate.filter(exchange, chain);
18   }
19   }
```

4.1.3、GlobalFilter

GlobalFilter 为请求业务以及路由的URI转换为真实业务服务的请求地址的核心过滤器，**不需要配置，模式系统初始化时加载，并作用在每个路由上**。

1. 初始化加载，通过GatewayAutoConfiguration自动创建
2. GlobalFilter转换成GatewayFilter，并作用于每个路由上，在FilteringWebHandler实现

4.2、Gateway Filter 使用

示例如下配置：

```
spring:
  cloud:
  gateway:
  default-filters:
  - AddResponseHeader=X-Response-Default-Foo, Default-Bar
```

- AddResponseHeader=X-Response-Default-Foo, Default-Bar 会被解析成FilterDefinition对象 （ name =AddResponseHeader ，args= [X-Response-Default-Foo,Default-Bar] ）
- 通FilterDefinition的Name找到AddResponseHeaderGatewayFilterFactory工厂
- 通过FilterDefinition 的args 创建Config对象（ name=X-Response-Default-Foo,value=Default-Bar ）
- 通过 AddResponseHeaderGatewayFilterFactory工厂的apply方法传入config创建GatewayFilter对象。

全部配置如下：

1、请求头

```
spring:
  cloud:
  gateway:
  routes:
  - id: add_request_header_route
  uri: http://example.org
  filters:
  - AddRequestHeader=X-Request-Foo, Bar
```

名称和值，这将为所有匹配请求的下游请求标头添加X-Request-Foo：Bar标头。

移除请求头

```
  filters:
  - RemoveRequestHeader=X-Request-Foo
```

2、请求参数

```
  filters:
  - AddRequestParameter=foo, bar
```

这会将foo = bar添加到下游请求的所有匹配请求的查询字符串中。

3、添加响应头

```
  filters:
  - AddResponseHeader=X-Response-Foo, Bar
```

这会将X-Response-Foo：Bar标头添加到所有匹配请求的下游响应标头中。

移除响应头

```
filters:
- RemoveResponseHeader=X-Response-Foo
```

设置响应头

```
filters:
- SetResponseHeader=X-Response-Foo, Bar
```

此GatewayFilter将替换具有给定名称的所有标头，而不是添加。

4、路径前缀

```
filters:
- PrefixPath=/mypath
```

这将使/ mypath前缀为所有匹配请求的路径。所以对/ hello的请求会被发送到/ mypath / hello。

5、原始主机头

没有参数，此过滤器设置路由过滤器将检查的请求属性，以确定是否应发送原始主机头，而不是http客户端确定的主机头。

```
filters:
- PreserveHostHeader
```

6、重定向

```
filters:
- RedirectTo=302, http://acme.org
```

这将发送带有Location：http：//acme.org标头的状态302以执行重定向。

7、重写路径

```
predicates:
- Path=/foo/**
filters:
- RewritePath=/foo/(?<segment>.*), /$\{segment}
```

对于/ foo / bar的请求路径，这将在发出下游请求之前将路径设置为/ bar。注意由于YAML规范，$ \替换为$。

8、保存Session

```
predicates:
- Path=/foo/**
filters:
- SaveSession
```

9、路径模板

SetPath GatewayFilter Factory采用路径模板参数。它提供了一种通过允许模板化路径段来操作请求路径的简单方法。

```
predicates:
```

```
- Path=/foo/{segment}
filters:
- SetPath=/{segment}
```

对于/ foo / bar的请求路径，这将在发出下游请求之前将路径设置为/ bar。

10、设置响应状态

```
spring:
 cloud:
 gateway:
 routes:
 - id: setstatusstring_route
 uri: http://example.org
 filters:
 - SetStatus=BAD_REQUEST
 - id: setstatusint_route
  uri: http://example.org
  filters:
  - SetStatus=401
```

11、请求参数剥离

parts参数指示在将请求发送到下游之前从请求中剥离的路径中的部分数。

```
predicates:
- Path=/name/**
filters:
- StripPrefix=2
```

当通过网关向/ name / bar / foo发出请求时，对nameservice的请求将类似于http：// nameservice / foo。

12、重试

retries：重试:应该尝试的重试次数

statuses：状态:应该重试的HTTP状态代码，使用org.springframework.http.HttpStatus表示

methods：方法:应该重试的HTTP方法，使用org.springframework.http.HttpMethod表示

series：系列:要重试的状态代码系列，使用org.springframework.http.HttpStatus.Series表示

```
routes:
- id: retry_test
uri: http://localhost:8080/flakey
predicates:
- Host=*.retry.com
```

```
  filters:
  - name: Retry
  args:
  retries: 3
   statuses: BAD_GATEWAY
```

13、Hystrix GatewayFilter Factory

```
spring:
 cloud:
 gateway:
 routes:
 - id: hystrix_route
 uri: http://example.org
 filters:
 - Hystrix=myCommandName
```

或者

```
spring:
 cloud:
 gateway:
 routes:
 - id: hystrix_route
 uri: lb://backing-service:8088
 predicates:
 - Path=/consumingserviceendpoint
 filters:
  - name: Hystrix
  args:
  name: fallbackcmd
  fallbackUri: forward:/incaseoffailureusethis
  - RewritePath=/consumingserviceendpoint, /backingserviceendpoint
```

```
hystrix.command.fallbackcmd.execution.isolation.thread.timeoutInMilliseco
nds: 5000
```

可以参考：https://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_hystrix_gatewayfilter_factory

14、请求限速　RequestRateLimiter GatewayFilter Factory

```
spring:
```

```
  cloud:
  gateway:
  routes:
  - id: requestratelimiter_route
  uri: http://example.org
  filters:
  - name: RequestRateLimiter
  args:
   redis-rate-limiter.replenishRate: 10
   redis-rate-limiter.burstCapacity: 20
```

或者：

```
spring:
  cloud:
  gateway:
  routes:
  - id: requestratelimiter_route
  uri: http://example.org
  filters:
  - name: RequestRateLimiter
  args:
   rate-limiter: "#{@myRateLimiter}"
   key-resolver: "#{@userKeyResolver}"
```

可以参考：https://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_requestratelimiter_gatewayfilter_factory

15、安全头 SecureHeaders GatewayFilter Factory

可以参考：https://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#_secureheaders_gatewayfilter_factory

16、SetPath GatewayFilter Factory

```
spring:
  cloud:
  gateway:
  routes:
  - id: setpath_route
```

```
  uri: http://example.org
  predicates:
  - Path=/foo/{segment}
  filters:
   - SetPath=/{segment}
```

17、SetResponseHeader GatewayFilter Factory

```
spring:
 cloud:
 gateway:
 routes:
 - id: setresponseheader_route
 uri: http://example.org
 filters:
 - SetResponseHeader=X-Response-Foo, Bar
```

18、SetStatus GatewayFilter Factory

```
spring:
 cloud:
 gateway:
 routes:
 - id: setstatusstring_route
 uri: http://example.org
 filters:
 - SetStatus=BAD_REQUEST
 - id: setstatusint_route
  uri: http://example.org
  filters:
   - SetStatus=401
```

19、StripPrefix GatewayFilter Factory

```
spring:
 cloud:
 gateway:
 routes:
 - id: nameRoot
 uri: http://nameservice
 predicates:
 - Path=/name/**
```

```
filters:
 - StripPrefix=2
```

20、Retry GatewayFilter Factory

```
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
```

五、Gateway 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。

常见的限流方式，比如Hystrix适用线程池隔离，超过线程池的负载，走熔断的逻辑。在一般应用服务器中，比如tomcat容器也是通过限制它的线程数来控制并发的；也有通过时间窗口的平均速度来控制流量。常见的限流纬度有比如通过Ip来限流、通过uri来限流、通过用户访问频次来限流。

一般限流都是在网关这一层做，比如Nginx、Openresty、kong、zuul、Spring Cloud Gateway等；也可以在应用层通过Aop这种方式去做限流。

5.1、常见的限流算法

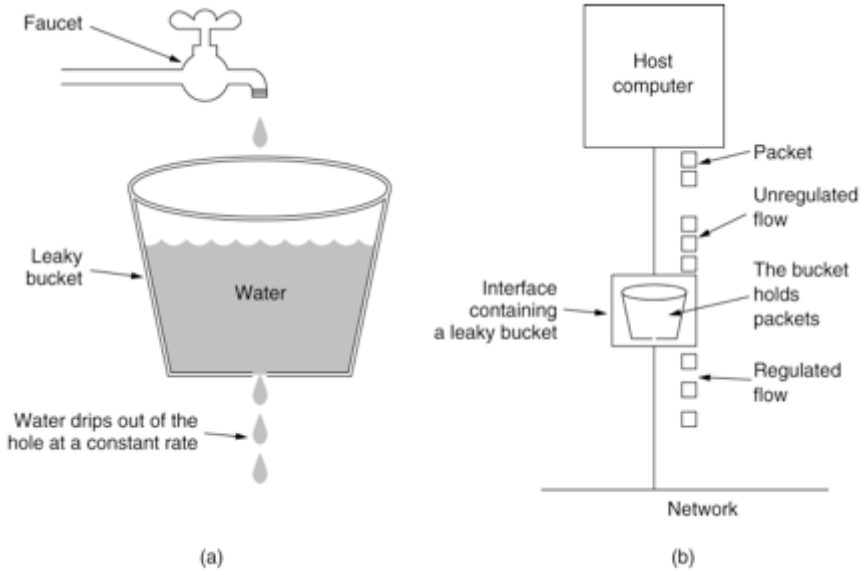
1、计数器算法

计数器算法采用计数器实现限流有点简单粗暴，一般我们会限制一秒钟的能够通过的请求数，比如限流qps为100，算法的实现思路就是从第一个请求进来开始计时，在接下去的1s内，每来一个请求，就把计数加1，如果累加的数字达到了100，那么后续的请求就会被全部拒绝。等到1s结束后，把计数恢复成0，重新开始计数。具体的实现可以是这样的：对于每次服务调用，可以通过AtomicLong#incrementAndGet()方法来给计数器加1并返回最新值，通过这个最新值和阈值进行比较。这种实现方式，相信大家都知道有一个弊端：如果

我在单位时间1s内的前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”

2、漏桶算法

漏桶算法为了消除"突刺现象"，可以采用漏桶算法实现限流，漏桶算法这个名字就很形象，算法内部有一个容器，类似生活用到的漏斗，当请求进来时，相当于水倒入漏斗，然后从下端小口慢慢匀速的流出。不管上面流量多大，下面流出的速度始终保持不变。不管服务调用方多么不稳定，通过漏桶算法进行限流，每10毫秒处理一次请求。因为处理的速度是固定的，请求进来的速度是未知的，可能突然进来很多请求，没来得及处理的请求就先放在桶里，既然是个桶，肯定是有容量上限，如果桶满了，那么新进来的请求就丢弃。

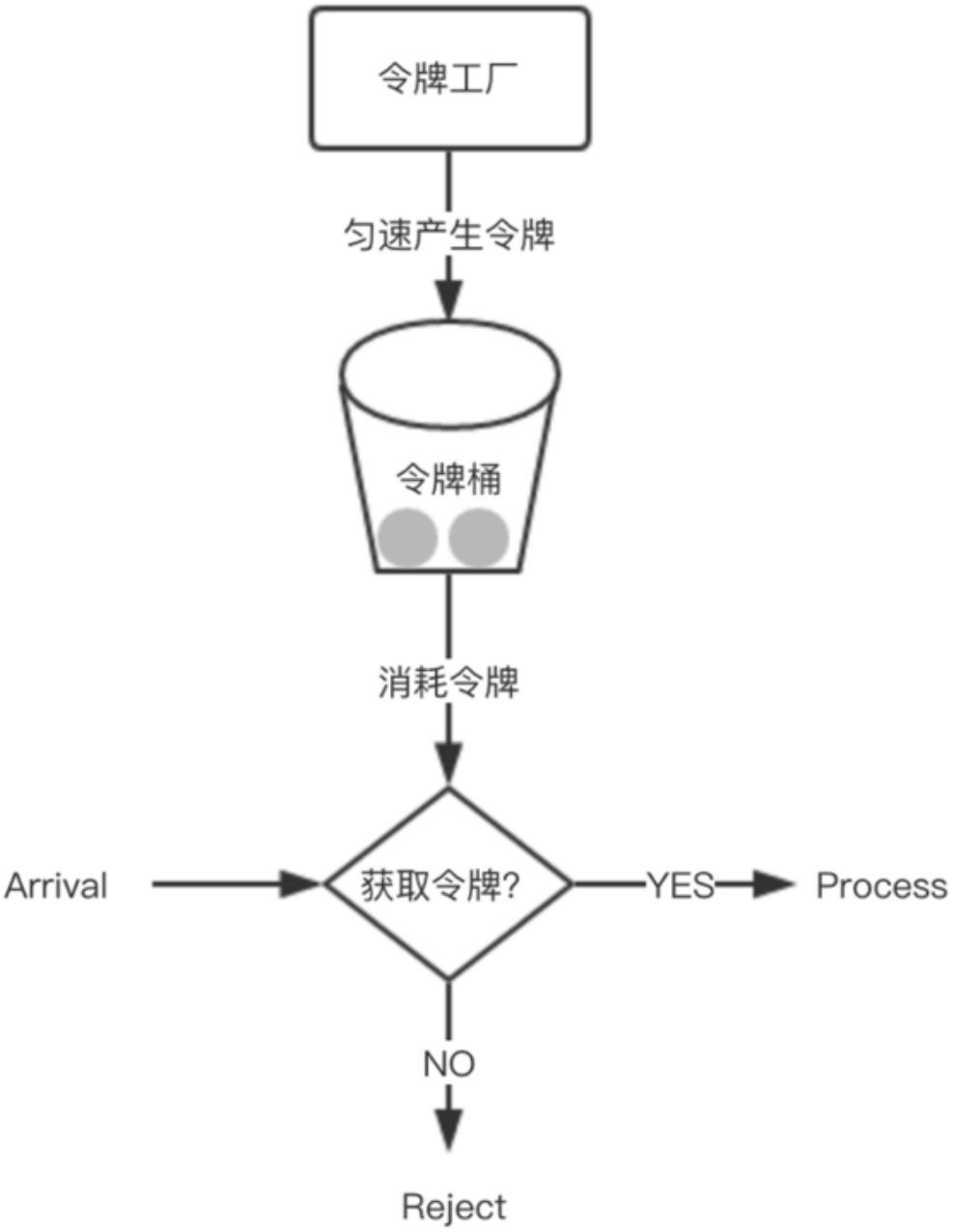


在算法实现方面，可以准备一个队列，用来保存请求，另外通过一个线程池（ScheduledExecutorService）来定期从队列中获取请求并执行，可以一次性获取多个并发执行。

这种算法，在使用过后也存在弊端：无法应对短时间的突发流量。

3、令牌桶算法

从某种意义上讲，令牌桶算法是对漏桶算法的一种改进，桶算法能够限制请求调用的速率，而令牌桶算法能够在限制调用的平均速率的同时还允许一定程度的突发调用。在令牌桶算法中，存在一个桶，用来存放固定数量的令牌。算法中存在一种机制，以一定的速率往桶中放令牌。每次请求调用需要先获取令牌，只有拿到令牌，才有机会继续执行，否则选择选择等待可用的令牌、或者直接拒绝。放令牌这个动作是持续不断的进行，如果桶中令牌数达到上限，就丢弃令牌，所以就存在这种情况，桶中一直有大量的可用令牌，这时进来的请求就可以直接拿到令牌执行，比如设置qps为100，那么限流器初始化完成一秒后，桶中就已经有100个令牌了，这时服务还没完全启动好，等启动完成对外提供服务时，该限流器可以抵挡瞬时的100个请求。所以，只有桶中没有令牌时，请求才会进行等待，最后相当于以一定的速率执行。



实现思路：可以准备一个队列，用来保存令牌，另外通过一个线程池定期生成令牌放到队列中，每来一个请求，就从队列中获取一个令牌，并继续执行。

在Spring Cloud Gateway中，有Filter过滤器，因此可以在“pre”类型的Filter中自行实现上述三种过滤器。但是限流作为网关最基本的功能，Spring Cloud Gateway官方就提供了RequestRateLimiterGatewayFilterFactory这个类，适用Redis和lua脚本实现了令牌桶的方式。

```
server:
 port: 8081
spring:
 cloud:
```

```yaml
gateway:
routes:
- id: limit_route
uri: http://httpbin.org:80/get
predicates:
 - After=2017-01-20T17:42:47.789-07:00[America/Denver]
 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，指定程序的端口为8081，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。

目前限流提供了基于Redis的实现,我们需要增加对应的依赖：

```xml
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>
```

可以通过KeyResolver来指定限流的Key,比如我们需要根据用户来做限流，IP来做限流，接口来做限流 等等。

KeyResolver需要实现resolve方法，比如根据Hostname进行限流，则需要用hostAddress去判断。实现完KeyResolver之后，需要将这个类的Bean注册到Ioc容器中。

```java
public class HostAddrKeyResolver implements KeyResolver {

 @Override
```

```
4   public Mono<String> resolve(ServerWebExchange exchange) {
5   return Mono.just(exchange.getRequest().getRemoteAddress().getAddress().g
etHostAddress());
6   }
7  }
8
9   @Bean
10   public HostAddrKeyResolver hostAddrKeyResolver() {
11   return new HostAddrKeyResolver();
12   }
```

通过exchange对象可以获取到请求信息，这边用了HostName

```
1  @Bean
2  public KeyResolver ipKeyResolver() {
3   return exchange -> Mono.just(exchange.getRequest().getRemoteAddress().ge
tHostName());
4  }
```

可以根据uri去限流，获取请求地址的uri作为限流key，这时KeyResolver代码如下：

```
1  public class UriKeyResolver implements KeyResolver {
2
3   @Override
4   public Mono<String> resolve(ServerWebExchange exchange) {
5   return Mono.just(exchange.getRequest().getURI().getPath());
6   }
7
8  }
9
10   @Bean
11   public UriKeyResolver uriKeyResolver() {
12   return new UriKeyResolver();
13   }
```

也可以以用户的维度去限流：请求路径中必须携带userId参数

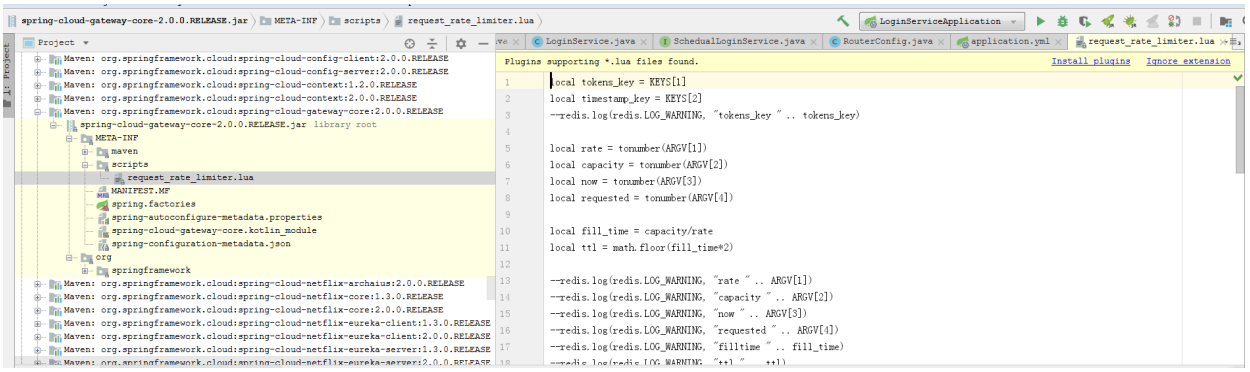
```
1   @Bean
2   KeyResolver userKeyResolver() {
3   return exchange -> Mono.just(exchange.getRequest().getQueryParams().getF
irst("user"));
4   }
```

用jmeter进行压测，配置10thread去循环请求lcoalhost:8081，循环间隔1s。从压测的结果上看到有部分请求通过，由部分请求失败。通过redis客户端去查看redis中存在的key。如下：

```
127.0.0.1:6379> keys *
1) "request_rate_limiter.{127.0.0.1}.timestamp"
2) "key1"
3) "name"
4) "nane"
5) "request_rate_limiter.{127.0.0.1}.tokens"
```

可见，RequestRateLimiter是使用Redis来进行限流的，并在redis中存储了2个key。关注这两个key含义可以看lua源代码。

具体实现逻辑在RequestRateLimiterGatewayFilterFactory类中，lua脚本在如下图所示的文件夹中：



具体源码不打算在这里讲述，读者可以自行查看。

```lua
local tokens_key = KEYS[1]
local timestamp_key = KEYS[2]
--redis.log(redis.LOG_WARNING, "tokens_key " .. tokens_key)

local rate = tonumber(ARGV[1])
local capacity = tonumber(ARGV[2])
local now = tonumber(ARGV[3])
local requested = tonumber(ARGV[4])

local fill_time = capacity/rate
local ttl = math.floor(fill_time*2)

--redis.log(redis.LOG_WARNING, "rate " .. ARGV[1])
--redis.log(redis.LOG_WARNING, "capacity " .. ARGV[2])
--redis.log(redis.LOG_WARNING, "now " .. ARGV[3])
--redis.log(redis.LOG_WARNING, "requested " .. ARGV[4])
--redis.log(redis.LOG_WARNING, "filltime " .. fill_time)
--redis.log(redis.LOG_WARNING, "ttl " .. ttl)

```

```
local last_tokens = tonumber(redis.call("get", tokens_key))
if last_tokens == nil then
 last_tokens = capacity
end
--redis.log(redis.LOG_WARNING, "last_tokens " .. last_tokens)

local last_refreshed = tonumber(redis.call("get", timestamp_key))
if last_refreshed == nil then
 last_refreshed = 0
end
--redis.log(redis.LOG_WARNING, "last_refreshed " .. last_refreshed)

local delta = math.max(0, now-last_refreshed)
local filled_tokens = math.min(capacity, last_tokens+(delta*rate))
local allowed = filled_tokens >= requested
local new_tokens = filled_tokens
local allowed_num = 0
if allowed then
 new_tokens = filled_tokens - requested
 allowed_num = 1
end

--redis.log(redis.LOG_WARNING, "delta " .. delta)
--redis.log(redis.LOG_WARNING, "filled_tokens " .. filled_tokens)
--redis.log(redis.LOG_WARNING, "allowed_num " .. allowed_num)
--redis.log(redis.LOG_WARNING, "new_tokens " .. new_tokens)

redis.call("setex", tokens_key, ttl, new_tokens)
redis.call("setex", timestamp_key, ttl, now)

return { allowed_num, new_tokens }

```

参考资料：

https://yq.aliyun.com/ziliao/343093

https://springcloud.cc/

http://spring.io/projects/spring-cloud-gateway

https://www.sohu.com/a/221110905_467759

https://www.2cto.com/kf/201805/746203.html

https://www.jianshu.com/p/44a0d6adcdea

http://cloud.spring.io/spring-cloud-static/spring-cloud-gateway/2.0.0.RELEASE/single/spring-cloud-gateway.html

https://blog.csdn.net/forezp/article/details/84926662

https://blog.csdn.net/forezp/article/details/85057268

https://blog.csdn.net/forezp/article/details/85081162

https://blog.csdn.net/forezp/article/details/85210153

https://www.cnblogs.com/bjlhx/p/9785926.html

https://www.jianshu.com/p/35b60946b8ce

https://www.cnblogs.com/bjlhx/p/9786478.html

https://blog.csdn.net/forezp/article/details/85081162