1. API网关

1.1 什么是API网关

API Gateway 是随着微服务（Microservice）这个概念一起兴起的一种架构模式，它用于解决微服务过于分散，没有一个统一的出入口进行流量管理的问题。

API网关可以看做系统与外界联通的入口，我们可以在网关进行处理一些非业务逻辑的逻辑，比如权限验证，监控，缓存，请求路由等等。

1.2 为什么需要API网关

- RPC协议转成HTTP。

由于在内部开发中我们都是以RPC协议(thrift or dubbo)去做开发，暴露给内部服务，当外部服务需要使用这个接口的时候往往需要将RPC协议转换成HTTP协议。

- 请求路由

在我们的系统中由于同一个接口新老两套系统都在使用，我们需要根据请求上下文将请求路由到对应的接口。

- 统一鉴权

对于鉴权操作不涉及到业务逻辑，那么可以在网关层进行处理，不用下层到业务逻辑。

- 统一监控

由于网关是外部服务的入口，所以我们可以在这里监控我们想要的数据，比如入参出参，链路时间。

- 流量控制，熔断降级

对于流量控制，熔断降级非业务逻辑可以统一放到网关层。
有很多业务都会自己去实现一层网关层，用来接入自己的服务，但是对于整个公司来说这还不够。

1.3 统一API网关

统一的API网关不仅有API网关的所有的特点，还有下面几个好处:

- 统一技术组件升级

在公司中如果有某个技术组件需要升级，那么是需要和每个业务线沟通，通常几个月都搞不定。举个例子如果对于入口的安全鉴权有重大安全隐患需要升级，如果速度还

是这么慢肯定是不行，那么有了统一的网关升级是很快的。

- 统一服务接入

对于某个服务的接入也比较困难，比如公司已经研发出了比较稳定的服务组件，正在公司大力推广，这个周期肯定也特别漫长，由于有了统一网关，那么只需要统一网关统一接入。

- 节约资源

不同业务不同部门如果按照我们上面的做法应该会都自己搞一个网关层，用来做这个事，可以想象如果一个公司有100个这种业务，每个业务配备4台机器，那么就需要400台机器。并且每个业务的开发RD都需要去开发这个网关层，去随时去维护，增加人力。如果有了统一网关层，那么也许只需要50台机器就可以做这100个业务的网关层的事，并且业务RD不需要随时关注开发，上线的步骤。

2. 统一网关的设计

2.1 异步化请求

对于我们自己实现的网关层，由于只有我们自己使用，对于吞吐量的要求并不高，所以，我们一般同步请求调用即可。

对于我们统一的网关层，如何用少量的机器接入更多的服务，这就需要我们采用异步，用来提高更多的吞吐量。对于异步化一般有下面两种策略：

- **Tomcat/Jetty+NIO+servlet3**

这种策略使用的比较普遍，京东，有赞，Zuul，都选取的是这个策略，这种策略比较适合HTTP。在Servlet3中可以开启异步。

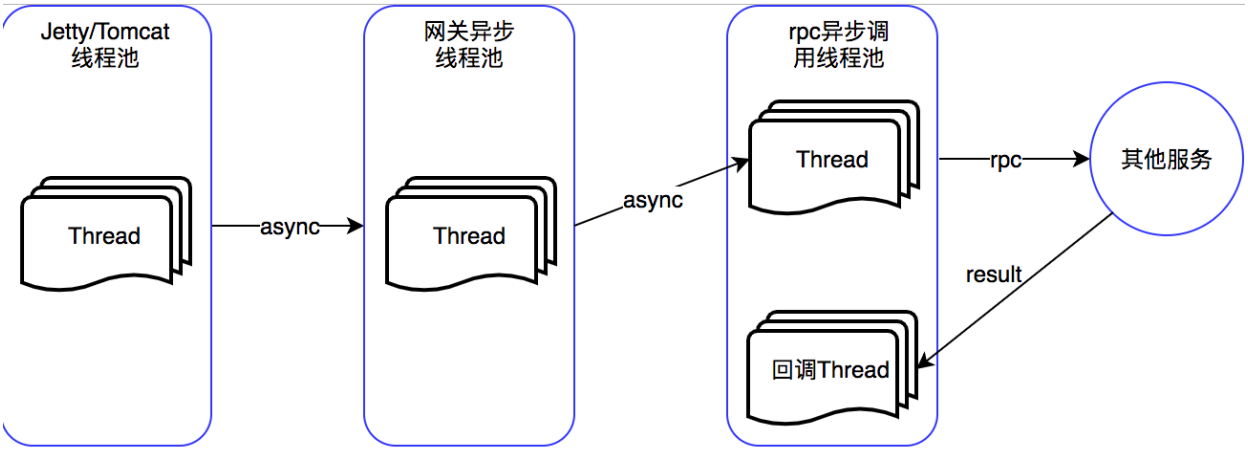
- **Netty+NIO**

Netty为高并发而生，目前唯品会的网关使用这个策略，在唯品会的技术文章中在相同的情况下Netty是每秒30w+的吞吐量，Tomcat是13w+,可以看出是有一定的差距的，但是Netty需要自己处理HTTP协议，这一块比较麻烦。

对于网关是HTTP请求场景比较多的情况可以采用Servlet，毕竟有更加成熟的处理HTTP协议。如果更加重视吞吐量那么可以采用Netty。

2.1.1 全链路异步

对于来的请求我们已经使用异步了，为了达到全链路异步，所以我们需要对去的请求也进行异步处理，对于去的请求我们可以利用我们rpc的异步支持进行异步请求，所以基本可以达到下图：

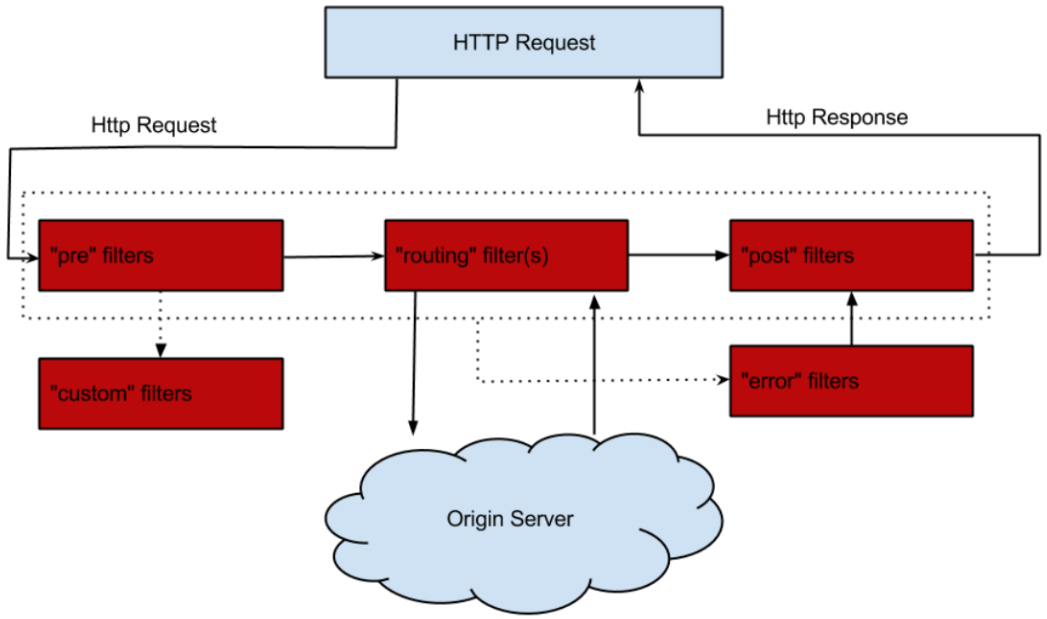


由在web容器中开启servlet异步，然后进入到网关的业务线程池中进行业务处理，然后进行rpc的异步调用并注册需要回调的业务，最后在回调线程池中进行回调处理。

2.2 链式处理

在设计模式中有一个模式叫责任链模式，他的作用是避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止。通过这种模式将请求的发送者和请求的处理者解耦了。在我们的各个框架中对此模式都有实现，比如servlet里面的filter，springmvc里面的Interceptor。

在Netflix Zuul中也应用了这种模式，如下图所示：



这种模式在网关的设计中我们可以借鉴到自己的网关设计:

- preFilters：前置过滤器，用来处理一些公共的业务，比如统一鉴权，统一限流，熔断降级，缓存处理等，并且提供业务方扩展。
- routingFilters: 用来处理一些泛化调用，主要是做协议的转换，请求的路由工作。
- postFilters: 后置过滤器，主要用来做结果的处理，日志打点，记录时间等等。
- errorFilters: 错误过滤器，用来处理调用异常的情况。

这种设计在有赞的网关也有应用。

2.3 业务隔离

上面在全链路异步的情况下不同业务之间的影响很小，但是如果在提供的自定义Fillter中进行了某些同步调用，一旦超时频繁那么就会对其他业务产生影响。所以我们需要采用隔离技术，降低业务之间的互相影响。

2.3.1 信号量隔离

信号量隔离只是限制了总的并发数，服务还是主线程进行同步调用。这个隔离如果远程调用超时依然会影响主线程，从而会影响其他业务。因此，如果只是想限制某个服务的总并发调用量或者调用的服务不涉及远程调用的话，可以使用轻量级的信号量来实现。有赞的网关由于没有自定义filter所以选取的是信号量隔离。

2.3.2 线程池隔离

最简单的就是不同业务之间通过不同的线程池进行隔离，就算业务接口出现了问题由于线程池已经进行了隔离那么也不会影响其他业务。在京东的网关实现之中就是采用的线程池隔离，比较重要的业务比如商品或者订单 都是单独的通过线程池去处理。但是由于是统一网关平台，如果业务线众多，大家都觉得自己的业务比较重要需要单独的线程池隔离，如果使用的是Java语言开发的话那么，在Java中线程是比较重的资源比较受限，如果需要隔离的线程池过多不是很适用。如果使用一些其他语言比如Golang进行开发网关的话，线程是比较轻的资源，所以比较适合使用线程池隔离。

2.3.3 集群隔离

如果有某些业务就需要使用隔离但是统一网关又没有线程池隔离那么应该怎么办呢？那么可以使用集群隔离，如果你的某些业务真的很重要那么可以为这一系列业务单独申请一个集群或者多个集群，通过机器之间进行隔离。

2.4 请求限流

流量控制可以采用很多开源的实现，比如阿里最近开源的Sentinel和比较成熟的Hystrix。

一般限流分为集群限流和单机限流:

- 利用统一存储保存当前流量的情况，一般可以采用Redis，这个一般会有一些性能损耗。
- 单机限流:限流每台机器我们可以直接利用Guava的令牌桶去做，由于没有远程调用性能消耗较小。

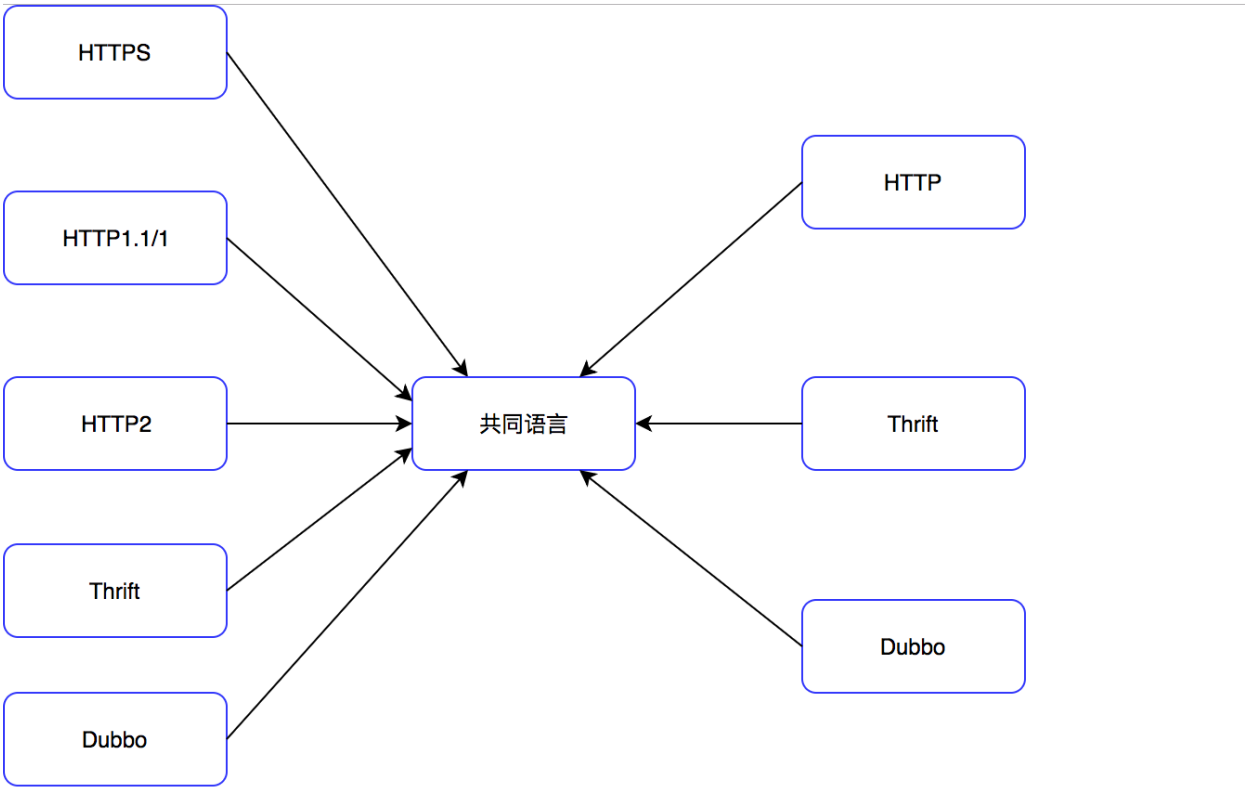
2.5 熔断降级

这一块也可以参照开源的实现Sentinel和Hystrix，这里不是重点就不多提了。

2.6 泛化调用

泛化调用指的是一些通信协议的转换，比如将HTTP转换成Thrift。在一些开源的网关中比如Zuul是没有实现的，因为各个公司的内部服务通信协议都不同。比如在唯品会中支持HTTP1，HTTP2，以及二进制的协议，然后转化成内部的协议，淘宝的支持HTTPS，HTTP1，HTTP2这些协议都可以转换成，HTTP，HSF，Dubbo等协议。

如何去实现泛化调用呢？由于协议很难自动转换，那么其实每个协议对应的接口需要提供一种映射。简单来说就是把两个协议都能转换成共同语言，从而互相转换。



一般来说共同语言有三种方式指定：

- json：json数据格式比较简单，解析速度快，较轻量级。在Dubbo的生态中有一个HTTP转Dubbo的项目是用JsonRpc做的，将HTTP转化成JsonRpc再转化成Dubbo。

比如可以将一个 www.baidu.com/id = 1 GET 可以映射为json：

```
{
 “method”: "getBaidu"
 "param" : {
 "id" : 1
 }
}
```

- xml：xml数据比较重，解析比较困难，这里不过多讨论。
- 自定义描述语言：一般来说这个成本比较高需要自己定义语言来进行描述并进行解析，

但是其扩展性，自定义个性化性都是最高。比如：spring自定义了一套自己的SPEL表达式语言
对于泛化调用如果要自己设计的话JSON基本可以满足，如果对于个性化的需要特别多的话倒是可以自己定义一套语言。

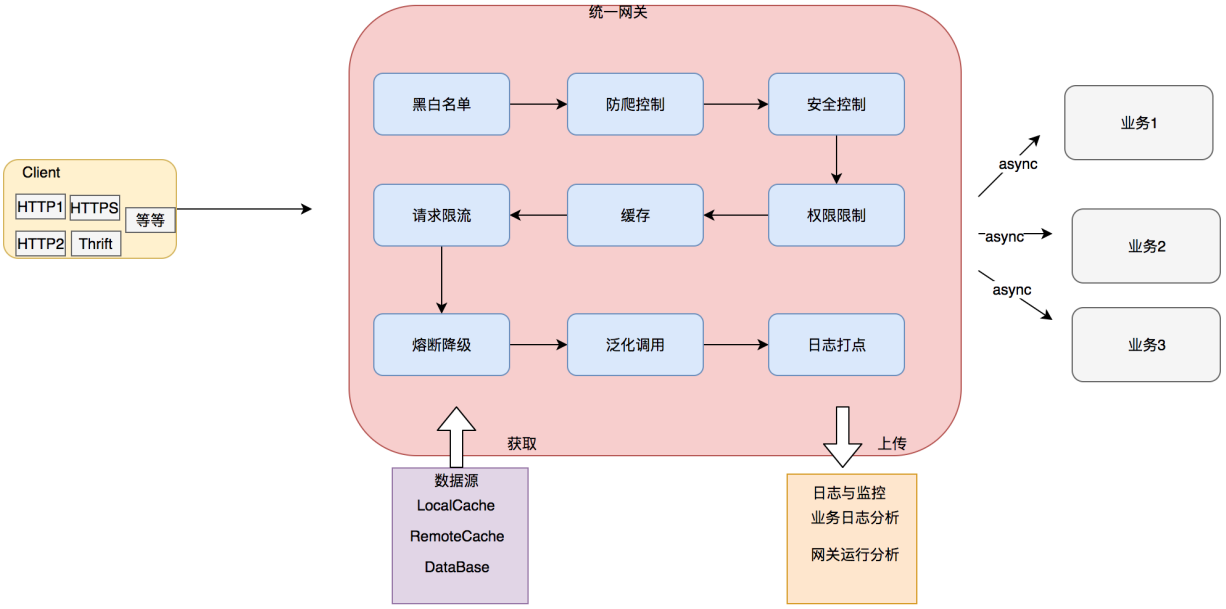
2.7 管理平台

上面介绍的都是如何实现一个网关的技术关键。这里需要介绍网关的一个业务关键。有了网关之后，需要一个管理平台如何去对我们上面所描述的技术关键进行配置,包括但不限于下面这些配置:

- 限流
- 熔断
- 缓存
- 日志
- 自定义filter
- 泛化调用

3. 总结

最后一个合理的标准网关应该按照如下去实现：



---	京东	唯品会	有赞	阿里
实现关键	servlet3.0	netty	servlet3.0	servlet3.0
异步情况	servlet异步，rpc是否异步不清楚	全链路异步	全链路异步	全链路异步
限流	---	---	平滑限流。最初是codis，后续换到每个单机的令牌桶限流。	1.基本流控 QPS做限流 控:支持AP APP+API 控33.大促 问API的权 里开源:Se
熔断降级	---	---	Hystrix	---
隔离	线程池隔离	---	信号量隔离	---
缓存	redis	---	二级缓存，本地缓存	HDCC 本地

			+Codis	程缓存，数
泛化调用	---	http,https,http1,http2,二进制	dubbo,http,nova	hsf,dubbo http2,http

参考：

- 京东:http://www.yunweipai.com/archives/23653.html
- 有赞网关:https://tech.youzan.com/api-gateway-in-practice/
- 唯品会:https://mp.weixin.qq.com/s/gREMe-G7nqNJJLzbZ3ed3A
- Zuul:http://www.scienjus.com/api-gateway-and-netflix-zuul/

4. Spring Cloud Zuul

Spring Cloud Zuul 是Spring Cloud Netflix 子项目的核心组件之一，可以作为微服务架构中的API网关使用，支持动态路由与过滤功能。

4.1 Zuul简介

API网关为微服务架构中的服务提供了统一的访问入口，客户端通过API网关访问相关服务。API网关的定义类似于设计模式中的门面模式，它相当于整个微服务架构中的门面，所有客户端的访问都通过它来进行路由及过滤。它实现了请求路由、负载均衡、校验过滤、服务容错、服务聚合等功能。

4.2 创建一个zuul-proxy模块

这里我们创建一个zuul-proxy模块来演示zuul的常用功能。

在pom.xml中添加相关依赖

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

在application.yml中进行配置

```yaml
server:
 port: 8801
spring:
 application:
 name: zuul-proxy
eureka:
 client:
 register-with-eureka: true
 fetch-registry: true
  service-url:
  defaultZone: http://localhost:8001/eureka/
```

在启动类上添加@EnableZuulProxy注解来启用Zuul的API网关功能

```java
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@EnableZuulProxy
@EnableDiscoveryClient
@SpringBootApplication
public class ZuulProxyApplication {

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

}
```

4.3常用功能

启动相关服务

我们通过启动eureka-server，两个user-service，feign-service和zuul-proxy来演示Zuul的常用功能，启动后注册中心显示如下。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
FEIGN-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-K1F7O7Q:feign-service:8701
USER-SERVICE	n/a (2)	(2)	UP (2) - DESKTOP-K1F7O7Q:user-service:8202 , DESKTOP-K1F7O7Q:user-service:8201
ZUUL-PROXY	n/a (1)	(1)	UP (1) - DESKTOP-K1F7O7Q:zuul-proxy:8801

配置路由规则

- 我们可以通过修改application.yml中的配置来配置路由规则，这里我们将匹配/userService/**的请求路由到user-service服务上去，匹配/feignService/**的请求路由到feign-service上去。

```
zuul:
 routes: #给服务配置路由
 user-service:
 path: /userService/**
 feign-service:
 path: /feignService/**
```

- 访问http://localhost:8801/userService/user/1可以发现请求路由到了user-service上了；
- 访问http://localhost:8801/feignService/user/1可以发现请求路由到了feign-service上了。

默认路由规则

- Zuul和Eureka结合使用，可以实现路由的自动配置，自动配置的路由以服务名称为匹配路径，相当于如下配置：

```
zuul:
 routes: #给服务配置路由
 user-service:
 path: /user-service/**
 feign-service:
 path: /feign-service/**
```

- 访问http://localhost:8801/user-service/user/1同样可以路由到了user-service上了；
- 访问http://localhost:8801/feign-service/user/1同样可以路由到了feign-service上了。
- 如果不想使用默认的路由规则，可以添加以下配置来忽略默认路由配置：

```
zuul:
```

```
  ignored-services: user-service,feign-service #关闭默认路由配置
```

负载均衡功能

多次调用http://localhost:8801/user-service/user/1进行测试，可以发现运行在8201和8202的user-service服务交替打印如下信息。

```
2020-03-06 23:30:03.132 INFO 19276 --- [nio-8201-exec-8] c.macro.cloud.controller.UserController : 根据id获取用户信息，用户名称为：macro
2020-03-06 23:31:30.556 INFO 19276 --- [nio-8201-exec-1] c.macro.cloud.controller.UserController : 根据id获取用户信息，用户名称为：macro
2020-03-06 23:31:31.362 INFO 19276 --- [nio-8201-exec-2] c.macro.cloud.controller.UserController : 根据id获取用户信息，用户名称为：macro
```

配置访问前缀

我们可以通过以下配置来给网关路径添加前缀，此处添加了/proxy前缀，这样我们需要访问http://localhost:8801/proxy/user-service/user/1才能访问到user-service中的接口。

```
zuul:
  prefix: /proxy #给网关路由添加前缀
```

Header过滤及重定向添加Host

- Zuul在请求路由时，默认会过滤掉一些敏感的头信息，以下配置可以防止路由时的Cookie及Authorization的丢失：

```
zuul:
  sensitive-headers: Cookie,Set-Cookie,Authorization #配置过滤敏感的请求头信息，设置为空就不会过滤
```

- Zuul在请求路由时，不会设置最初的host头信息，以下配置可以解决：

```
zuul:
  add-host-header: true #设置为true重定向是会添加host请求头
```

查看路由信息

我们可以通过SpringBoot Actuator来查看Zuul中的路由信息。

- 在pom.xml中添加相关依赖：

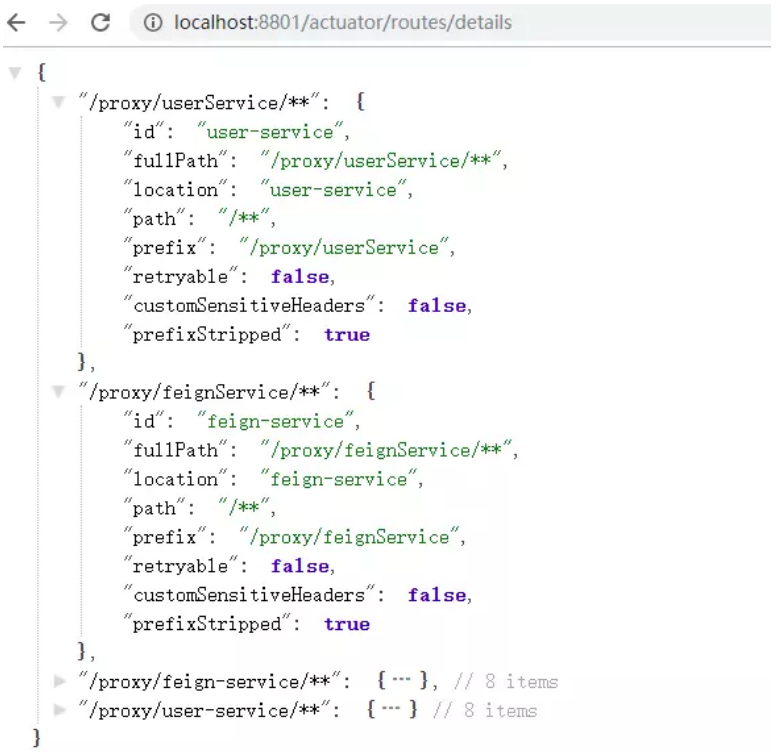
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- 修改application.yaml配置文件，开启查看路由的端点：

```yaml
management:
  endpoints:
  web:
  exposure:
  include: 'routes'
```

- 通过访问http://localhost:8801/actuator/routes查看简单路由信息：

- 通过访问http://localhost:8801/actuator/routes/details查看详细路由信息：



4.4 过滤器

路由与过滤是Zuul的两大核心功能，路由功能负责将外部请求转发到具体的服务实例上去，是实现统一访问入口的基础，过滤功能负责对请求过程进行额外的处理，是请求校验过滤及服务聚合的基础。

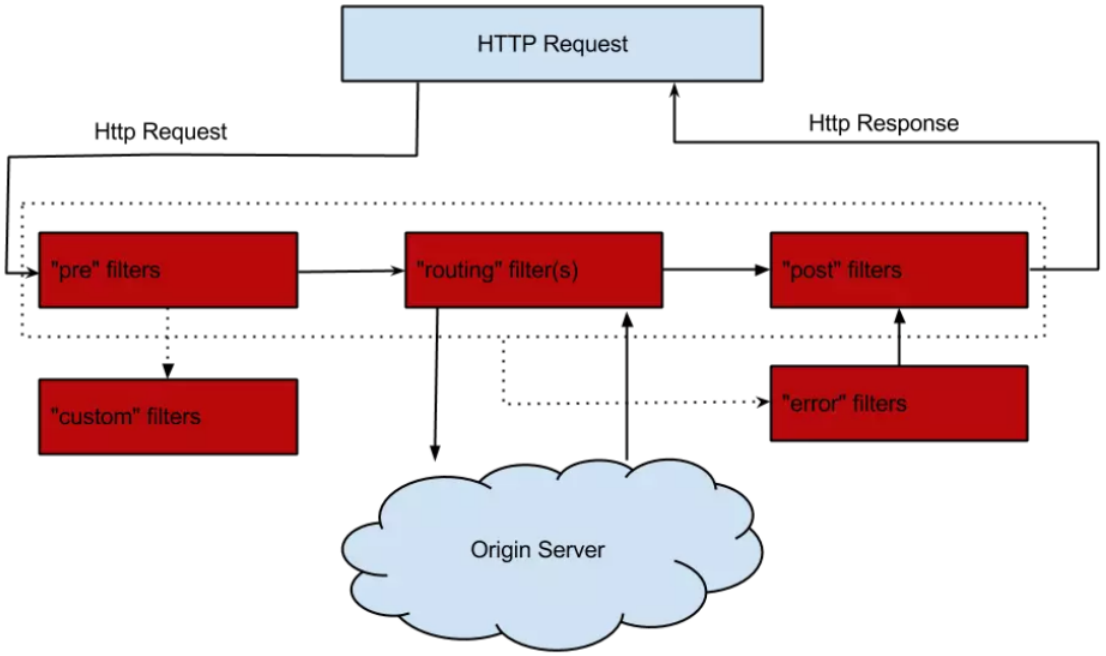
过滤器类型

Zuul中有以下几种典型的过滤器类型。

- pre：在请求被路由到目标服务前执行，比如权限校验、打印日志等功能；
- routing：在请求被路由到目标服务时执行，这是使用Apache HttpClient或Netflix Ribbon构建和发送原始HTTP请求的地方；
- post：在请求被路由到目标服务后执行，比如给目标服务的响应添加头信息，收集统计数据等功能；
- error：请求在其他阶段发生错误时执行。

过滤器的生命周期

下图描述了一个HTTP请求到达API网关后，如何在各种不同类型的过滤器中流转的过程。



自定义过滤器

接下来我们自定义一个过滤器来演示下过滤器的作用。

添加PreLogFilter类继承ZuulFilter

这是一个前置过滤器，用于在请求路由到目标服务前打印请求日志。

```
@Component
public class PreLogFilter extends ZuulFilter {
```

```
3  private Logger LOGGER = LoggerFactory.getLogger(this.getClass());
4
5  /**
6  * 过滤器类型，有pre、routing、post、error四种。
7  */
8  @Override
9  public String filterType() {
10  return "pre";
11  }
12
13  /**
14  * 过滤器执行顺序，数值越小优先级越高。
15  */
16  @Override
17  public int filterOrder() {
18  return 1;
19  }
20
21  /**
22  * 是否进行过滤，返回true会执行过滤。
23  */
24  @Override
25  public boolean shouldFilter() {
26  return true;
27  }
28
29  /**
30  * 自定义的过滤器逻辑，当shouldFilter()返回true时会执行。
31  */
32  @Override
33  public Object run() throws ZuulException {
34  RequestContext requestContext = RequestContext.getCurrentContext();
35  HttpServletRequest request = requestContext.getRequest();
36  String host = request.getRemoteHost();
37  String method = request.getMethod();
38  String uri = request.getRequestURI();
39  LOGGER.info("Remote host:{},method:{},uri:{}", host, method, uri);
40  return null;
41  }
42 }
```

过滤器功能演示

添加过滤器后，我们访问http://localhost:8801/user-service/user/1测试下，会打印如下日志。

```
2020-03-07 15:51:37.097 INFO 73684 --- [nio-8801-exec-1] com.macro.cloud.filter.PreLogFilter : Remote host:0:0:0:0:0:0:0:1,method:GET,uri:/user-service/user/1
```

核心过滤器

过滤器名称	过滤类型	优先级	过滤器的作用
ServletDetectionFilter	pre	-3	检测当前请求是通过DispatcherServlet处理运行的还是ZuulServlet运行处理的。
Servlet30WrapperFilter	pre	-2	对原始的HttpServletRequest进行包装。
FormBodyWrapperFilter	pre	-1	将Content-Type为application/x-www-form-urlencoded或multipart/form-data的请求包装成FormBodyRequestWrapper对象。
DebugFilter	route	1	根据zuul.debug.request的配置来决定是否打印debug日志。
PreDecorationFilter	route	5	对当前请求进行预处理以便执行后续操作。
RibbonRoutingFilter	route	10	通过Ribbon和Hystrix来向服务实例发起请求，并将请求结果进行返回。
SimpleHostRoutingFilter	route	100	只对请求上下文中有routeHost参数的进行处理，直接使用HttpClient向routeHost对应的物理地址进行转发。
SendForwardFilter	route	500	只对请求上下文中有forward.to参数的进行处理，进行本地跳转。
SendErrorFilter	post	0	当其他过滤器内部发生异常时的会由它来进行处理，产生错误响应。
SendResponseFilter	post	1000	利用请求上下文的响应信息来组织请求成功的响应内容。

禁用过滤器

- 我们可以对过滤器进行禁用的配置，配置格式如下：

```
zuul:
 filterClassName:
 filter:
 disable: true
```

- 以下是禁用PreLogFilter的示例配置：

```
zuul:
```

```
  PreLogFilter:
  pre:
  disable: true
```

4.5 Ribbon和Hystrix的支持

由于Zuul自动集成了Ribbon和Hystrix，所以Zuul天生就有负载均衡和服务容错能力，我们可以通过Ribbon和Hystrix的配置来配置Zuul中的相应功能。

- 可以使用Hystrix的配置来设置路由转发时HystrixCommand的执行超时时间：

```
hystrix:
  command: #用于控制HystrixCommand的行为
  default:
  execution:
  isolation:
  thread:
  timeoutInMilliseconds: 1000 #配置HystrixCommand执行的超时时间，执行超过该时间会进行服务降级处理
```

- 可以使用Ribbon的配置来设置路由转发时请求连接及处理的超时时间：

```
ribbon: #全局配置
  ConnectTimeout: 1000 #服务请求连接超时时间（毫秒）
  ReadTimeout: 3000 #服务请求处理超时时间（毫秒）
```

常用配置

```
zuul:
  routes: #给服务配置路由
  user-service:
  path: /userService/**
  feign-service:
  path: /feignService/**
  ignored-services: user-service,feign-service #关闭默认路由配置
  prefix: /proxy #给网关路由添加前缀
  sensitive-headers: Cookie,Set-Cookie,Authorization #配置过滤敏感的请求头信息，设置为空就不会过滤
   add-host-header: true #设置为true重定向是会添加host请求头
   retryable: true # 关闭重试机制
   PreLogFilter:
   pre:
   disable: false #控制是否启用过滤器
```

5. Spring Cloud Gateway：新一代API网关服务

Spring Cloud Gateway 为 SpringBoot 应用提供了API网关支持，具有强大的智能路由与过滤器功能，下面将对其用法进行详细介绍。

5.1 Gateway 简介

Gateway是在Spring生态系统之上构建的API网关服务，基于Spring 5，Spring Boot 2和Project Reactor等技术。Gateway旨在提供一种简单而有效的方式来对API进行路由，以及提供一些强大的过滤器功能， 例如：熔断、限流、重试等。

Spring Cloud Gateway 具有如下特性：

- 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；
- 可以对路由指定 Predicate（断言）和 Filter（过滤器）；
- 集成Hystrix的断路器功能；
- 集成 Spring Cloud 服务发现功能；
- 易于编写的 Predicate（断言）和 Filter（过滤器）；
- 请求限流功能；
- 支持路径重写。

5.2 相关概念

- **Route（路由）**：路由是构建网关的基本模块，它由ID，目标URI，一系列的断言和过滤器组成，如果断言为true则匹配该路由；
- **Predicate（断言）**：指的是Java 8 的 Function Predicate。 输入类型是Spring框架中的ServerWebExchange。 这使开发人员可以匹配HTTP请求中的所有内容，例如请求头或请求参数。如果请求与断言相匹配，则进行路由；
- **Filter（过滤器）**：指的是Spring框架中GatewayFilter的实例，使用过滤器，可以在请求被路由前后对请求进行修改。

5.3 创建 api-gateway模块

创建一个api-gateway模块来演示Gateway的常用功能。

在pom.xml中添加相关依赖

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

两种不同的配置路由方式

Gateway 提供了两种不同的方式用于配置路由，一种是通过yml文件来配置，另一种是通过Java Bean来配置，下面我们分别介绍下。

使用yml配置

```
server:
 port: 9201
service-url:
 user-service: http://localhost:8201
spring:
 cloud:
 gateway:
 routes:
 - id: path_route #路由的ID
  uri: ${service-url.user-service}/user/{id} #匹配后路由地址
  predicates: # 断言，路径相匹配的进行路由
  - Path=/user/{id}
```

- 启动eureka-server，user-service和api-gateway服务，并调用该地址测试：http://localhost:9201/user/1
- 我们发现该请求被路由到了user-service的该路径上：http://localhost:8201/user/1

使用Java Bean配置

- 添加相关配置类，并配置一个RouteLocator对象：

```
@Configuration
public class GatewayConfig {

 @Bean
 public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
 return builder.routes()
 .route("path_route2", r -> r.path("/user/getByUsername")
 .uri("http://localhost:8201/user/getByUsername"))
 .build();
  }
}
```

- 重新启动api-gateway服务，并调用该地址测试：
http://localhost:9201/user/getByUsername?username=macro
- 我们发现该请求被路由到了user-service的该路径上：
http://localhost:8201/user/getByUsername?username=macro

5.4 Route Predicate 的使用

Spring Cloud Gateway将路由匹配作为Spring WebFlux HandlerMapping基础架构的一部分。 Spring Cloud Gateway包括许多内置的Route Predicate工厂。 所有这些Predicate都与HTTP请求的不同属性匹配。 多个Route Predicate工厂可以进行组合，下面我们来介绍下一些常用的Route Predicate。
注意：Predicate中提到的配置都在application-predicate.yml文件中进行修改，并用该配置启动api-gateway服务。

After Route Predicate

在指定时间之后的请求会匹配该路由。

```
spring:
 cloud:
 gateway:
 routes:
 - id: after_route
 uri: ${service-url.user-service}
 predicates:
 - After=2020-03-08T16:30:00+08:00[Asia/Shanghai]
```

Before Route Predicate

在指定时间之前的请求会匹配该路由。

```
spring:
 cloud:
 gateway:
 routes:
 - id: before_route
 uri: ${service-url.user-service}
 predicates:
 - Before=2020-03-09T16:30:00+08:00[Asia/Shanghai]
```

Between Route Predicate

在指定时间区间内的请求会匹配该路由。

```
spring:
 cloud:
 gateway:
 routes:
```

```
- id: before_route
uri: ${service-url.user-service}
predicates:
- Between=2020-03-08T16:30:00+08:00[Asia/Shanghai], 2020-03-09T16:30:00+08:00[Asia/Shanghai]
```

Cookie Route Predicate

带有指定Cookie的请求会匹配该路由。

```
spring:
 cloud:
 gateway:
 routes:
 - id: cookie_route
 uri: ${service-url.user-service}
 predicates:
 - Cookie=username,macro
```

使用curl工具发送带有cookie为username=macro的请求可以匹配该路由。

```
curl http://localhost:9201/user/1 --cookie "username=macro"
```

Header Route Predicate

带有指定请求头的请求会匹配该路由。

```
spring:
 cloud:
 gateway:
 routes:
 - id: header_route
 uri: ${service-url.user-service}
 predicates:
 - Header=X-Request-Id, \d+
```

使用curl工具发送带有请求头为X-Request-Id:123的请求可以匹配该路由。

```
curl http://localhost:9201/user/1 -H "X-Request-Id:123"
```

Host Route Predicate

带有指定Host的请求会匹配该路由。

```
spring:
 cloud:
```

```
  gateway:
  routes:
  - id: host_route
  uri: ${service-url.user-service}
  predicates:
  - Host=**.macrozheng.com
```

使用curl工具发送带有请求头为Host:www.macrozheng.com的请求可以匹配该路由。

```
curl http://localhost:9201/user/1 -H "Host:www.macrozheng.com"
```

Method Route Predicate

发送指定方法的请求会匹配该路由。

```
spring:
  cloud:
  gateway:
  routes:
  - id: method_route
  uri: ${service-url.user-service}
  predicates:
  - Method=GET
```

使用curl工具发送GET请求可以匹配该路由。

```
curl http://localhost:9201/user/1
```

使用curl工具发送POST请求无法匹配该路由。

```
curl -X POST http://localhost:9201/user/1
```

Path Route Predicate

发送指定路径的请求会匹配该路由。

```
spring:
  cloud:
  gateway:
  routes:
  - id: path_route
  uri: ${service-url.user-service}/user/{id}
  predicates:
  - Path=/user/{id}
```

使用curl工具发送/user/1路径请求可以匹配该路由。

```
curl http://localhost:9201/user/1
```

使用curl工具发送/abc/1路径请求无法匹配该路由。

```
curl http://localhost:9201/abc/1
```

Query Route Predicate

带指定查询参数的请求可以匹配该路由。

```
spring:
 cloud:
 gateway:
 routes:
 - id: query_route
 uri: ${service-url.user-service}/user/getByUsername
 predicates:
 - Query=username
```

使用curl工具发送带username=macro查询参数的请求可以匹配该路由。

```
curl http://localhost:9201/user/getByUsername?username=macro
```

使用curl工具发送带不带查询参数的请求无法匹配该路由。

```
curl http://localhost:9201/user/getByUsername
```

RemoteAddr Route Predicate

从指定远程地址发起的请求可以匹配该路由。

```
spring:
 cloud:
 gateway:
 routes:
 - id: remoteaddr_route
 uri: ${service-url.user-service}
 predicates:
 - RemoteAddr=192.168.1.1/24
```

使用curl工具从192.168.1.1发起请求可以匹配该路由。

```
curl http://localhost:9201/user/1
```

Weight Route Predicate

使用权重来路由相应请求，以下表示有80%的请求会被路由到localhost:8201，20%会被路由到localhost:8202。

```
spring:
 cloud:
 gateway:
 routes:
 - id: weight_high
 uri: http://localhost:8201
 predicates:
 - Weight=group1, 8
 - id: weight_low
  uri: http://localhost:8202
  predicates:
  - Weight=group1, 2
```

5.5 Route Filter 的使用

路由过滤器可用于修改进入的HTTP请求和返回的HTTP响应，路由过滤器只能指定路由进行使用。Spring Cloud Gateway 内置了多种路由过滤器，他们都由GatewayFilter的工厂类来产生，下面我们介绍下常用路由过滤器的用法。

AddRequestParameter GatewayFilter

给请求添加参数的过滤器。

```
spring:
 cloud:
 gateway:
 routes:
 - id: add_request_parameter_route
 uri: http://localhost:8201
 filters:
 - AddRequestParameter=username, macro
 predicates:
  - Method=GET
```

以上配置会对GET请求添加username=macro的请求参数，通过curl工具使用以下命令进行测试。

```
curl http://localhost:9201/user/getByUsername
```

相当于发起该请求：

```
curl http://localhost:8201/user/getByUsername?username=macro
```

StripPrefix GatewayFilter

对指定数量的路径前缀进行去除的过滤器。

```
spring:
 cloud:
 gateway:
 routes:
 - id: strip_prefix_route
 uri: http://localhost:8201
 predicates:
 - Path=/user-service/**
 filters:
  - StripPrefix=2
```

以上配置会把以/user-service/开头的请求的路径去除两位，通过curl工具使用以下命令进行测试。

```
curl http://localhost:9201/user-service/a/user/1
```

相当于发起该请求：

```
curl http://localhost:8201/user/1
```

PrefixPath GatewayFilter

```
spring:
 cloud:
 gateway:
 routes:
 - id: prefix_path_route
 uri: http://localhost:8201
 predicates:
 - Method=GET
 filters:
  - PrefixPath=/user
```

以上配置会对所有GET请求添加/user路径前缀，通过curl工具使用以下命令进行测试。

```
curl http://localhost:9201/1
```

相当于发起该请求：

```
curl http://localhost:8201/user/1
```

Hystrix GatewayFilter

Hystrix 过滤器允许你将断路器功能添加到网关路由中，使你的服务免受级联故障的影响，并提供服务降级处理。

- 要开启断路器功能，我们需要在pom.xml中添加Hystrix的相关依赖：

```
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
```

- 然后添加相关服务降级的处理类：

```
@RestController
public class FallbackController {

 @GetMapping("/fallback")
 public Object fallback() {
 Map<String,Object> result = new HashMap<>();
 result.put("data",null);
 result.put("message","Get request fallback!");
 result.put("code",500);
  return result;
  }
}
```

- 在application-filter.yml中添加相关配置，当路由出错时会转发到服务降级处理的控制器上：

```
spring:
 cloud:
 gateway:
 routes:
 - id: hystrix_route
 uri: http://localhost:8201
```

```
  predicates:
  - Method=GET
  filters:
   - name: Hystrix
   args:
   name: fallbackcmd
   fallbackUri: forward:/fallback
```

- 关闭user-service，调用该地址进行测试：http://localhost:9201/user/1 ，发现已经返回了服务降级的处理信息。

RequestRateLimiter GatewayFilter

RequestRateLimiter 过滤器可以用于限流，使用RateLimiter实现来确定是否允许当前请求继续进行，如果请求太大默认会返回HTTP 429-太多请求状态。

- 在pom.xml中添加相关依赖：

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

- 添加限流策略的配置类，这里有两种策略一种是根据请求参数中的username进行限流，另一种是根据访问IP进行限流；

```
@Configuration
public class RedisRateLimiterConfig {
 @Bean
 KeyResolver userKeyResolver() {
 return exchange -> Mono.just(exchange.getRequest().getQueryParams().getFirst("username"));
 }

 @Bean
 public KeyResolver ipKeyResolver() {
```

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

- 我们使用Redis来进行限流，所以需要添加Redis和RequestRateLimiter的配置，这里对所有的GET请求都进行了按IP来限流的操作；

```
server:
  port: 9201
spring:
  redis:
  host: localhost
  password: 123456
  port: 6379
  cloud:
  gateway:
   routes:
   - id: requestratelimiter_route
   uri: http://localhost:8201
   filters:
   - name: RequestRateLimiter
   args:
   redis-rate-limiter.replenishRate: 1 #每秒允许处理的请求数量
   redis-rate-limiter.burstCapacity: 2 #每秒最大处理的请求数量
   key-resolver: "#{@ipKeyResolver}" #限流策略，对应策略的Bean
   predicates:
   - Method=GET
 logging:
  level:
  org.springframework.cloud.gateway: debug
```

- 多次请求该地址：http://localhost:9201/user/1 ，会返回状态码为429的错误；

Retry GatewayFilter

对路由请求进行重试的过滤器，可以根据路由请求返回的HTTP状态码来确定是否进行重试。

- 修改配置文件：

```yaml
spring:
 cloud:
 gateway:
 routes:
 - id: retry_route
 uri: http://localhost:8201
 predicates:
 - Method=GET
 filters:
  - name: Retry
  args:
  retries: 1 #需要进行重试的次数
  statuses: BAD_GATEWAY #返回哪个状态码需要进行重试，返回状态码为5XX进行重试
  backoff:
  firstBackoff: 10ms
  maxBackoff: 50ms
  factor: 2
  basedOnPreviousValue: false
```

- 当调用返回500时会进行重试，访问测试地址：http://localhost:9201/user/111
- 可以发现user-service控制台报错2次，说明进行了一次重试。

```
2020-03-08 14:08:53.435 ERROR 2280 --- [nio-8201-exec-2] o.a.c.c.C.[.[.[/].[dispatcherServlet] : Servlet.service() for servlet [dispatcherServlet] in context with path [] threw exception [Request processing failed; nested exception is java.lang.NullPointerException] with root cause

java.lang.NullPointerException: null
 at com.macro.cloud.controller.UserController.getUser(UserController.java:34) ~[classes/:na]
```

5.6 结合注册中心使用

我们上面讲到使用Zuul作为网关结合注册中心进行使用时，默认情况下Zuul会根据注册中心注册的服务列表，以服务名为路径创建动态路由，Gateway同样也实现了该功能。下面我们演示下Gateway结合注册中心如何使用默认的动态路由和过滤器。

使用动态路由

- 在pom.xml中添加相关依赖：

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

- 添加application-eureka.yml配置文件：

```
server:
 port: 9201
spring:
 application:
 name: api-gateway
 cloud:
 gateway:
 discovery:
 locator:
 enabled: true #开启从注册中心动态创建路由的功能
 lower-case-service-id: true #使用小写服务名，默认是大写
eureka:
```

```
  client:
  service-url:
  defaultZone: http://localhost:8001/eureka/
logging:
  level:
  org.springframework.cloud.gateway: debug
```

- 使用application-eureka.yml配置文件启动api-gateway服务，访问 http://localhost:9201/user-service/user/1 ，可以路由到user-service的 http://localhost:8201/user/1 处。

使用过滤器

在结合注册中心使用过滤器的时候，我们需要注意的是uri的协议为ib（https://blog.csdn.net/swingwang/article/details/72887367），这样才能启用Gateway的负载均衡功能。

- 修改application-eureka.yml文件，使用了PrefixPath过滤器，会为所有GET请求路径添加/user路径并路由；

```
server:
  port: 9201
spring:
  application:
  name: api-gateway
  cloud:
  gateway:
  routes:
  - id: prefixpath_route
  uri: ib://user-service #此处需要使用ib协议
  predicates:
  - Method=GET
  filters:
  - PrefixPath=/user
  discovery:
  locator:
  enabled: true
eureka:
  client:
```

```
  service-url:
  defaultZone: http://localhost:8001/eureka/
logging:
  level:
  org.springframework.cloud.gateway: debug
```

- 使用application-eureka.yml配置文件启动api-gateway服务，访问 http://localhost:9201/1，可以路由到user-service的http://localhost:8201/user/1 处。

6. Spring-Cloud-Gateway 源码解析

6.1 Spring GateWay架构

Spring Cloud提供了两套方便我们编写网关的中间件，分别是**zuul和Spring GateWay**，在zuul1的IO模型是使用**BIO**(图1-1)。而zuul2对IO模型使用**NIO**进行了重构(图1-2)。而Spring GateWay的IO模型是使用NIO。而在Netflix发布zuul2的时候Spring Cloud已经开始不集成到Spring Cloud中，因为Spring Cloud 等着zuul2集成太久，才有了Spring Gateway。Spring GateWay的架构是基于Spring webflux的基础上开发的。而对webflux的RP中涉及的Back Pressure、Stream、asynchronous好处就不多了。

图1-1

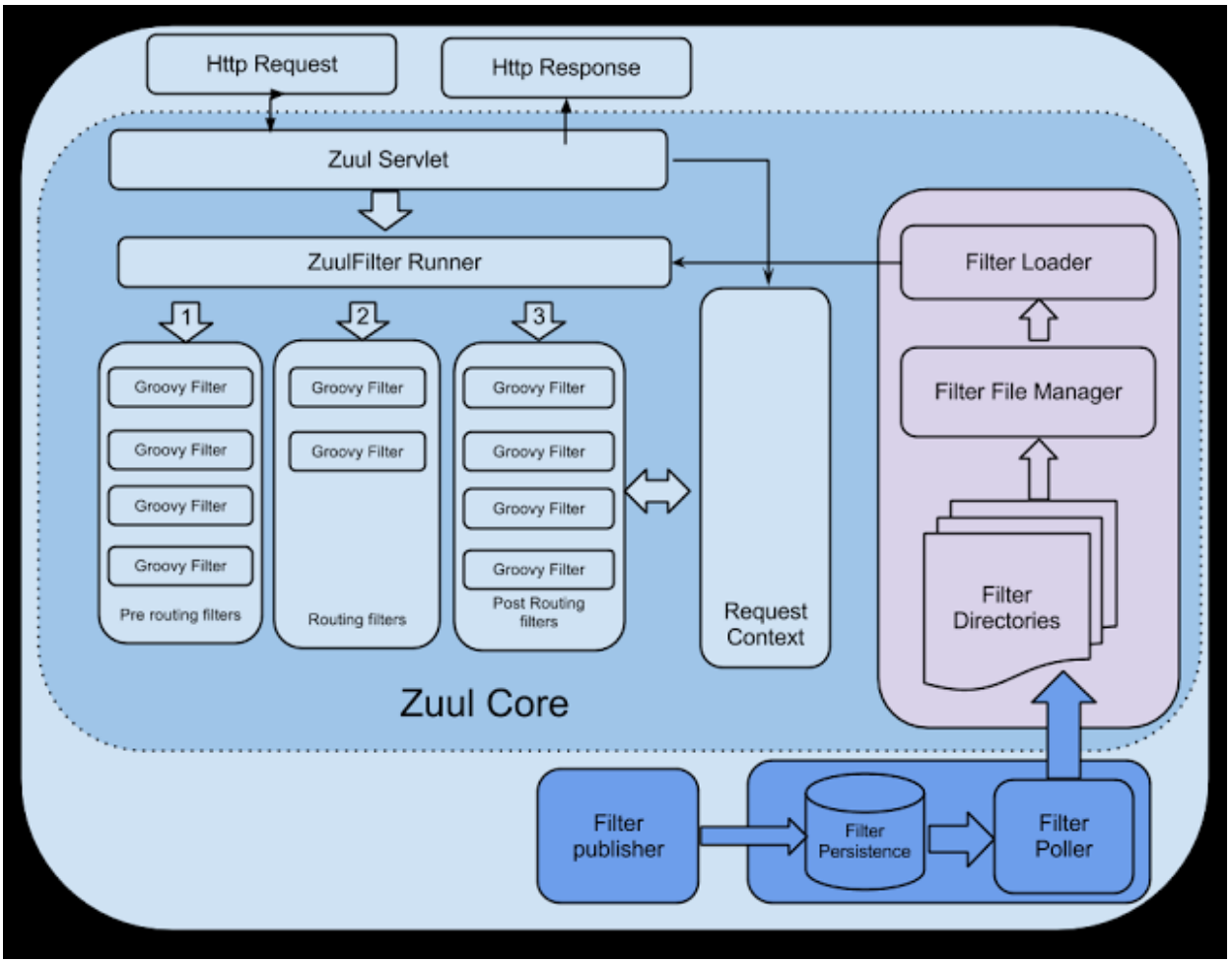
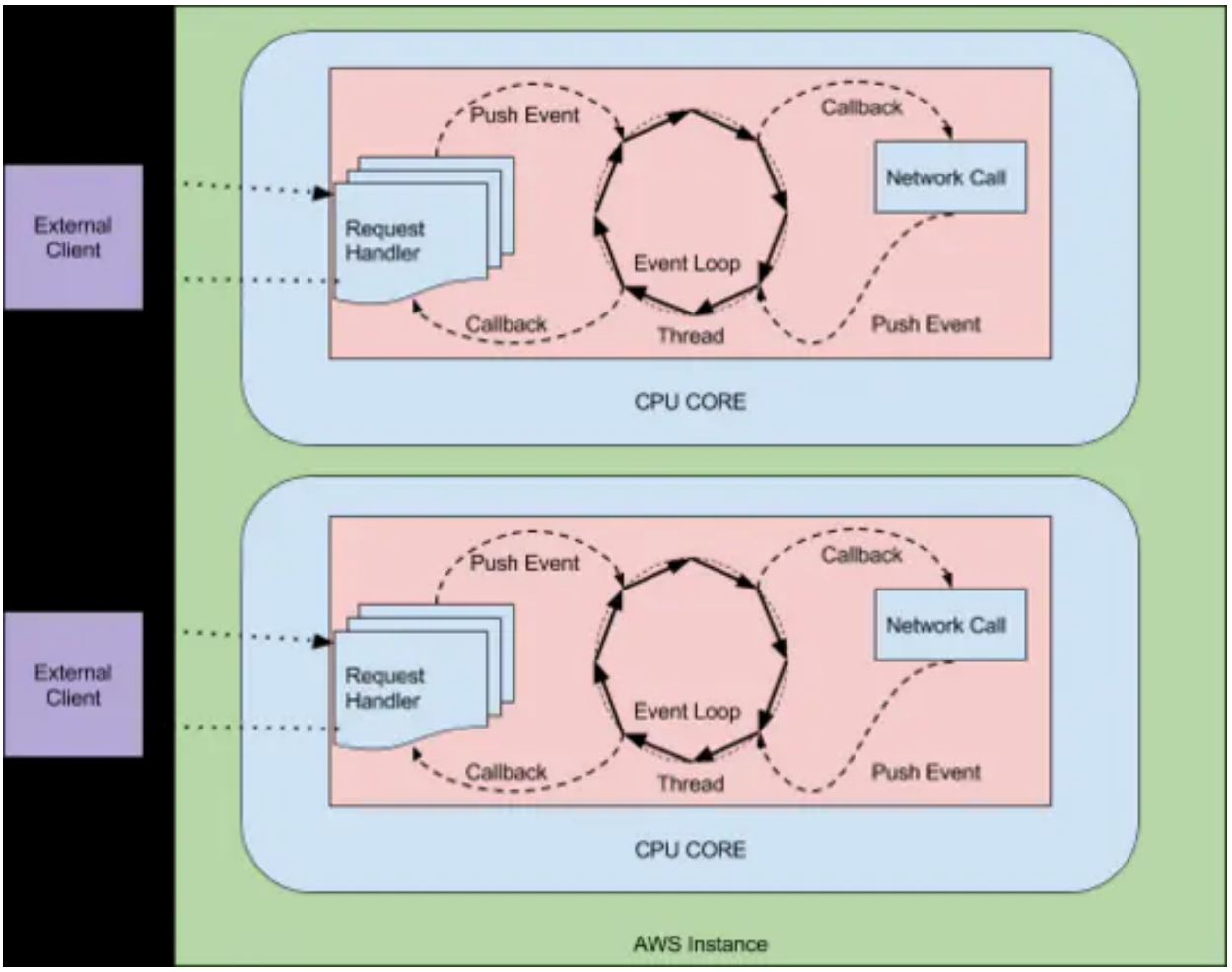

Http Request
Http Response
Zuul Servlet
ZuulFilter Runner
1
2
3
Groovy Filter
Groovy Filter
Groovy Filter
Groovy Filter
Pre routing filters
Groovy Filter
Groovy Filter
Routing filters
Groovy Filter
Groovy Filter
Groovy Filter
Groovy Filter
Post Routing filters
Request Context
Filter Loader
Filter File Manager
Filter Directories
Zuul Core
Filter publisher
Filter Persistence
Filter Poller

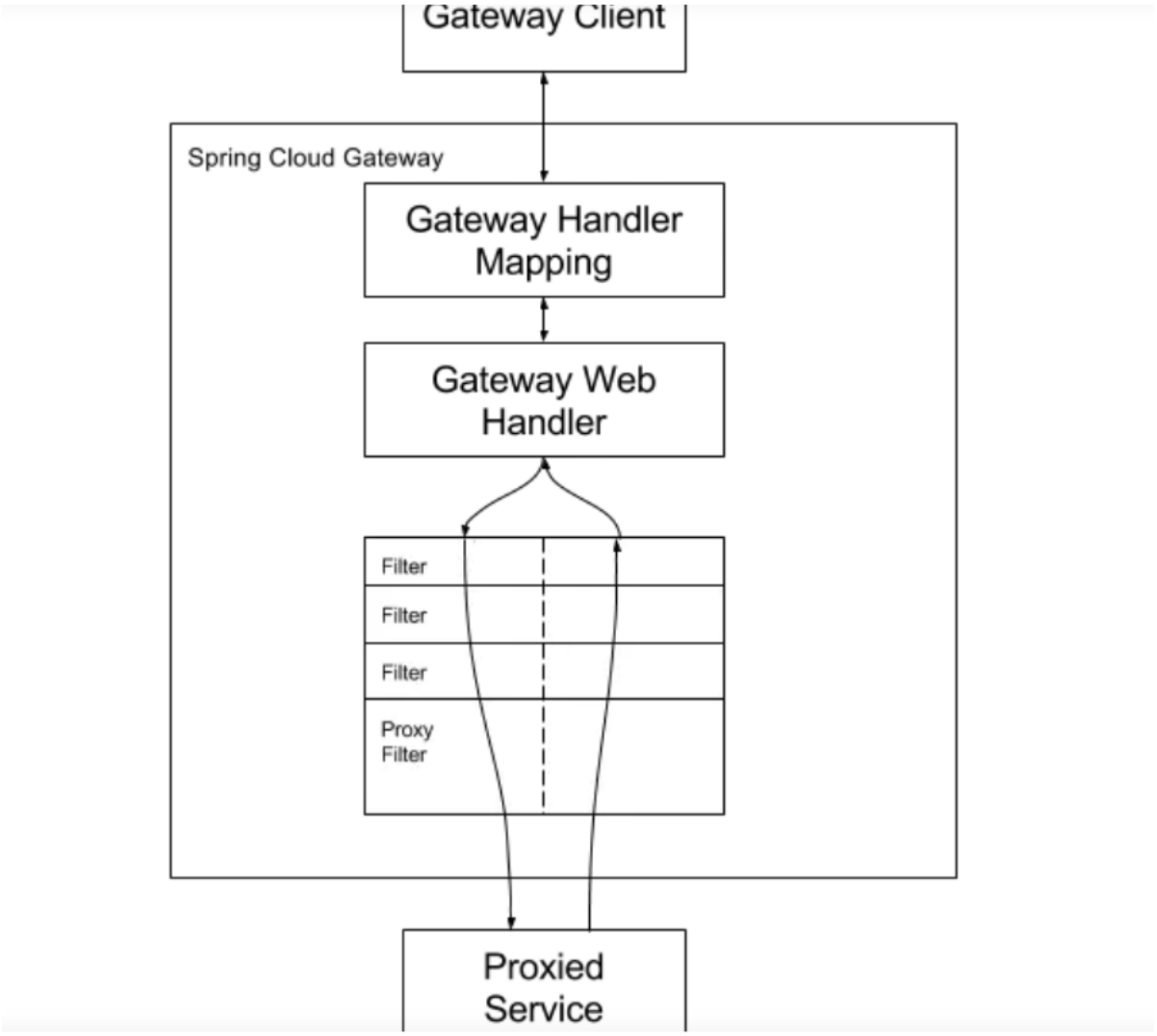
图1-2



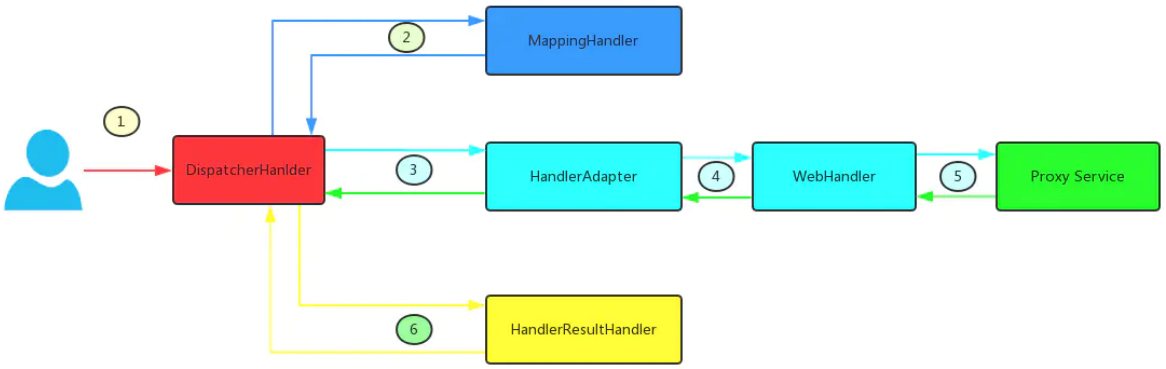
6.2 Spring GateWay 转发请求过程

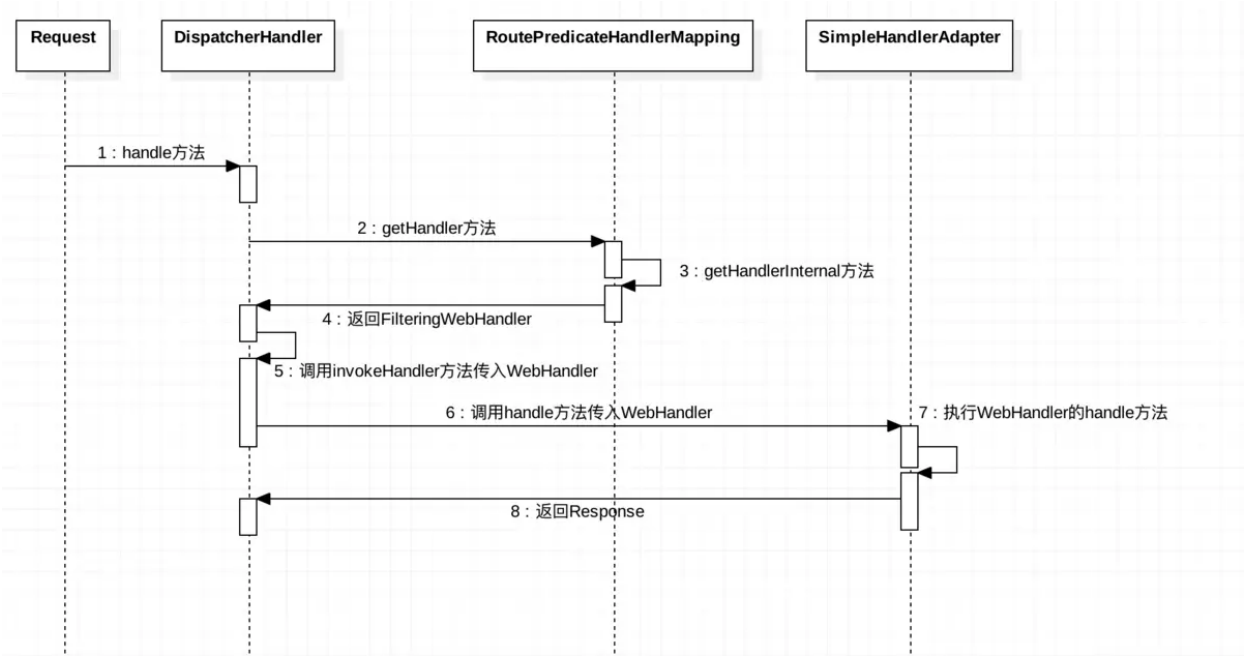
转发请求过程

在Spring mvc是通过HandlerMapping解析请求链接，然后根据请求链接找到执行这个请求Controller类 。而在Spring GateWay中也是使用HandlerMapping对请求的链接进行解析匹配对应的Route进行代理转发到对应的服务。图2-1为整个请求的流程，用户请求先通过DispatcherHandler找到对应GateWwayHandlerMapping,再通过GateWwayHandlerMapping解析匹配到对应的Handler。Handler处理完后，再经过Filter，最终到Proxied Service.


Gateway Client
Spring Cloud Gateway
Gateway Handler Mapping
Gateway Web Handler
Filter
Filter
Filter
Proxy Filter
Proxied Service

转发请求过程代码分析


1
2
MappingHandler
DispatcherHanlder
3
HandlerAdapter
4
WebHandler
5
Proxy Service
6
HandlerResultHandler



DispatchHandler.png

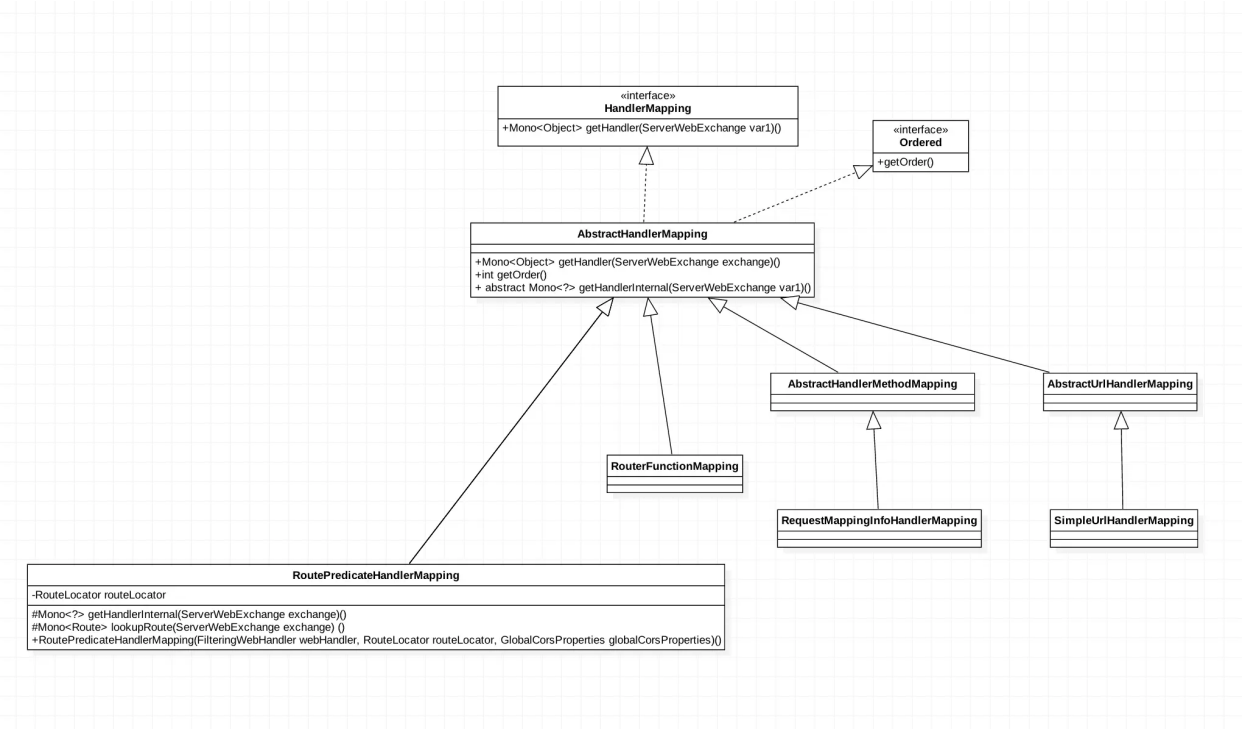
- 1.请求先由DispatcherHanlder进行处理，DispatcherHanlder初始化的时候会从IOC中查找实现HandlerMapping接口的实现类。然后保存到内部变量handlerMappings中，DispatcerHandler调用Handler方法迭代handlerMappings中的HandlerMapping,
- 2.这里只讲解RoutePredicateHandlerMapping，因此然后调用RoutePredicateHandlerMapping中的获取路由的方法，当RoutePredicateHandlerMapping获取到对应的路由的时候会将Route存储到ServerWebExchanges的属性中，然后返回实现了WebHandler接口的FilteringWebHandler。FilteringWebHandler是一个存放过滤器的Handler。
- 3.最后DispatcherHanlder通过SimpleHandlerAdapter适配器的方式调用FilteringWebHandler的handler方法，FilteringWebHandler调用所有的过滤器，包括proxy filter。通过proxyFilter请求被代理的服务。处理完毕后，并将Response响应回去。

通过流程的分析，这里我们可以学习到适配的设计模式的使用。GateWay中的HandlerMapping有RouterFunctionMapping、RoutePredicateHandlerMapping等。这些HandlerMapping返回的结果都是不一样的。那么DispatcherHanlder的后续处理也不会一样。那么可以通过适配器的方式，根据HandlerMapping返回的结果进行适配调用。

HandlerMapping \ RouteLocator\GlobalFilter(WebHandler)

HandlerMapping

下面为handler类关系图。这里主要涉及到Spring GateWay相关类的探讨。如：Spring Webflux使用到的RouteFuntionMapping和SimpleUrlHandlerMapping等不做探讨。



AbstractHandlerMapping

HandlerMapping和Ordered接口主要定义了获取getHandler和当前hanler加载顺序。AbstractHandlerMapping在getHanlder封装了CORS处理。因为所有Handler都可能会涉及到CORS的处理，抽象到AbstractHandlerMapping处理，再提供了getHandlerInternal让子类实现具体的查找Handler的方法。

```
1  public Mono<Object> getHandler(ServerWebExchange exchange) {
2   return this.getHandlerInternal(exchange).map((handler) -> {
3   if (CorsUtils.isCorsRequest(exchange.getRequest())) {
4   CorsConfiguration configA = this.globalCorsConfigSource.getCorsConfigura
tion(exchange);
5   CorsConfiguration configB = this.getCorsConfiguration(handler,
exchange);
6   CorsConfiguration config = configA != null ? configA.combine(configB) :
configB;
7   if (!this.getCorsProcessor().process(config, exchange) || CorsUtils.isPr
eFlightRequest(exchange.getRequest())) {
8   return REQUEST_HANDLED_HANDLER;
9   }
10   }
11
12   return handler;
13   });
```

```
14   }
15  protected abstract Mono<?> getHandlerInternal(ServerWebExchange var1);
```

RoutePredicateHandlerMapping

RoutePredicateHandlerMapping是处理获取路由的hanlder。RoutePredicateHandlerMapping中的RouteLocator是存储了我们启动的时候加载的路由对象信息。获取路由的时候，调用RoutePredicateHanlderMapping的getHandlerInternal方法从RouteLocator获取路由存放在ServerWebExchange中，返回webFilter。

```
1   private final RouteLocator routeLocator;
2
3   @Override
4   protected Mono<?> getHandlerInternal(ServerWebExchange exchange) {
5   exchange.getAttributes().put(GATEWAY_HANDLER_MAPPER_ATTR, getClass().get
SimpleName());
6
7   return lookupRoute(exchange)
8   // .log("route-predicate-handler-mapping", Level.FINER) //name this
9   .flatMap((Function<Route, Mono<?>>) r -> {
10   exchange.getAttributes().remove(GATEWAY_PREDICATE_ROUTE_ATTR);
11   if (logger.isDebugEnabled()) {
12   logger.debug("Mapping [" + getExchangeDesc(exchange) + "] to " + r);
13   }
14
15   exchange.getAttributes().put(GATEWAY_ROUTE_ATTR, r);
16   return Mono.just(webHandler);
17   }).switchIfEmpty(Mono.empty().then(Mono.fromRunnable(() -> {
18   exchange.getAttributes().remove(GATEWAY_PREDICATE_ROUTE_ATTR);
19   if (logger.isTraceEnabled()) {
20   logger.trace("No RouteDefinition found for [" + getExchangeDesc(exchang
e) + "]");
21   }
22   })));
23   }
24
25
26   protected Mono<Route> lookupRoute(ServerWebExchange exchange) {
27   return this.routeLocator
28   .getRoutes()
29   //individually filter routes so that filterWhen error delaying is not a
problem
```

```
30   .concatMap(route -> Mono
31   .just(route)
32   .filterWhen(r -> {
33   // add the current route we are testing
34   exchange.getAttributes().put(GATEWAY_PREDICATE_ROUTE_ATTR, r.getId());
35   return r.getPredicate().apply(exchange);
36   })
37   //instead of immediately stopping main flux due to error, log and swall
ow it
38   .doOnError(e -> logger.error("Error applying predicate for route: "+rou
te.getId(), e))
39   .onErrorResume(e -> Mono.empty())
40   )
41   // .defaultIfEmpty() put a static Route not found
42   // or .switchIfEmpty()
43   // .switchIfEmpty(Mono.<Route>empty().log("noroute"))
44   .next()
45   //TODO: error handling
46   .map(route -> {
47   if (logger.isDebugEnabled()) {
48   logger.debug("Route matched: " + route.getId());
49   }
50   validateRoute(route, exchange);
51   return route;
52   });
53
54   /* TODO: trace logging
55   if (logger.isTraceEnabled()) {
56   logger.trace("RouteDefinition did not match: " +
routeDefinition.getId());
57   }*/
```

RoutePredicateHandlerMapping的创建是在GatewayAutoConfiguration进行自动创建。创建的时候指定webHandler和routeLocator。webHandler就是一个封装了GlobalFilter的对象，而routeLocator就是保存了所有Route的对象。

```
1  @Bean
2   public RoutePredicateHandlerMapping routePredicateHandlerMapping(
3   FilteringWebHandler webHandler, RouteLocator routeLocator,
4   GlobalCorsProperties globalCorsProperties) {
5   return new RoutePredicateHandlerMapping(webHandler, routeLocator,
```

```
globalCorsProperties);
}


```

RouteLocator

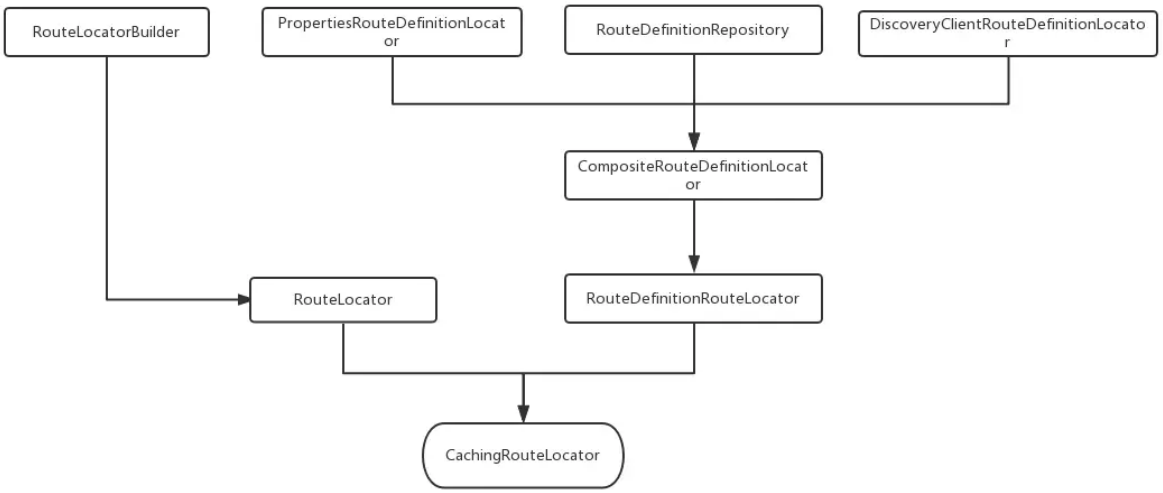
RouteLocator主要作用是提供获取路由的类型。我们在分析RoutePredicateHandlerMapping的时候，知道RoutePredicateHandlerMapping获取路由是通过RouteLocator进行获取的。那么我们这里分析下RouteLocator加载路由。

Route组成

Route主要为三部分：

- Proxy：代理的信息包括被代理的uri。
- Predicate:包含接受请求的方法、path等信息
- Filter:Route自定义的过滤器。

RouteLocator加载过程



RouteLocator加载过程.jpg

最终的 RouteLocator是CachingRoutelocator。加载过程是自上而下进行创建。

Route加载来源分为三种来源:

- 第一种是通过RouteLocatorBuilder方式，在创建RouteLocator的方法上通过注入RouteLocatorBuilder来创建路由。如：

```
@Bean
public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
RouteLocator routeLocator = builder.routes()
```

```
  .route("get",r->r.path("/security/**").uri("lb://security-server"))
  .build();
  return routeLocator;
  }
```

- 第二种方式是通过Properties文件进行创建路由。Properties路由的创建包括：PropertiesRouteDefinitionLocator和DiscoveryClientRouteDefinitionLocator.
    - PropertiesRouteDefinitionLocator 通过代码创建Route
    - DiscoveryClientRouteDefinitionLocator 是通过配置文件中lb://security-serve 指定的服务名称，在Eurek查找Service中的uri进行创建Route。

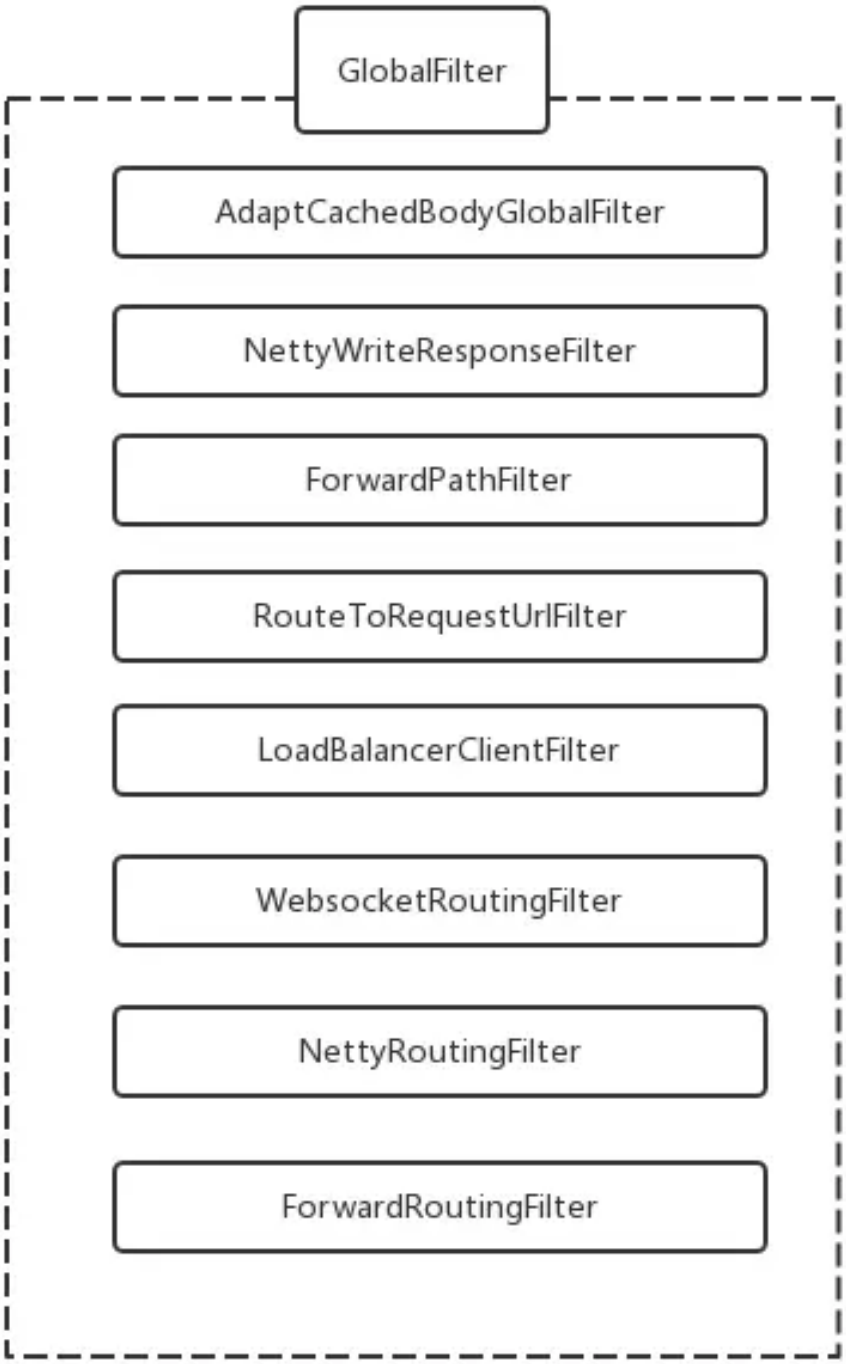
```
spring:
 cloud:
 gateway:
 routes:
 - id: cookie_route
 uri: http://example.org
 predicates:
 - Cookie=chocolate, ch.p
```

- 第三种方式是通过MYSQL或者Reids、内存（InMemoryRouteDefinitionRepository）方式创建路由。实现RouteDefinitionRepository接口实现接口中的方式。InMemoryRouteDefinitionRepository为默认方式。

```
public class InMemoryRouteDefinitionRepository implements RouteDefinitionRepository {

 private final Map<String, RouteDefinition> routes = synchronizedMap(new LinkedHashMap<String, RouteDefinition>());

 @Override
 public Mono<Void> save(Mono<RouteDefinition> route) {
 return route.flatMap( r -> {
 routes.put(r.getId(), r);
 return Mono.empty();
  });
  }

```

```
13  @Override
14  public Mono<Void> delete(Mono<String> routeId) {
15  return routeId.flatMap(id -> {
16  if (routes.containsKey(id)) {
17  routes.remove(id);
18  return Mono.empty();
19  }
20  return Mono.defer(() -> Mono.error(new NotFoundException("RouteDefiniti
on not found: "+routeId)));
21  });
22  }
23
24  @Override
25  public Flux<RouteDefinition> getRouteDefinitions() {
26  return Flux.fromIterable(routes.values());
27  }
28 }
29
30
```

GlobalFilter



全局Filter的加载过程

Filter我们区分为全局Filter和RouteFilter

- Globle Filter是上图所列的Filter。而这些全局的Filter是在GatewayAutoConfiguration中通过@Bean的方式进行创建。
- Route Filter是在创建的Route的时候指定。这种路由属于自定义的路由。

NettyRoutingFilter

在转发过程分析中我们知道最终的代理请求是通过一个Proxy Filter进行请求Proxy Service，那么这个Proxy Filter就是NettyRoutingFilter。通过下面的源码我们可以看到在`proxyRequest.sendHeaders() .send(request.getBody().map(dataBuffer -> ((NettyDataBuffer) dataBuffer).getNativeBuffer()));`中请求Proxy Service.

```
@Override
 public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
 URI requestUrl = exchange.getRequiredAttribute(GATEWAY_REQUEST_URL_ATTR);

 String scheme = requestUrl.getScheme();
 if (isAlreadyRouted(exchange) || (!"http".equals(scheme) && !"https".equals(scheme))) {
 return chain.filter(exchange);
 }
 setAlreadyRouted(exchange);

  ServerHttpRequest request = exchange.getRequest();

 final HttpMethod method = HttpMethod.valueOf(request.getMethod().toString());
 final String url = requestUrl.toString();

 HttpHeaders filtered = filterRequest(this.headersFilters.getIfAvailable(),
 exchange);

 final DefaultHttpHeaders httpHeaders = new DefaultHttpHeaders();
 filtered.forEach(httpHeaders::set);

 String transferEncoding = request.getHeaders().getFirst(HttpHeaders.TRANSFER_ENCODING);
 boolean chunkedTransfer = "chunked".equalsIgnoreCase(transferEncoding);

 boolean preserveHost = exchange.getAttributeOrDefault(PRESERVE_HOST_HEADER_ATTRIBUTE, false);

 Mono<HttpClientResponse> responseMono = this.httpClient.request(method, url, req -> {
```

```
28  final HttpClientRequest proxyRequest = req.options(NettyPipeline.SendOptions::flushOnEach)
29  .headers(httpHeaders)
30  .chunkedTransfer(chunkedTransfer)
31  .failOnServerError(false)
32  .failOnClientError(false);
33
34  if (preserveHost) {
35  String host = request.getHeaders().getFirst(HttpHeaders.HOST);
36  proxyRequest.header(HttpHeaders.HOST, host);
37  }
38
39  return proxyRequest.sendHeaders() //I shouldn't need this
40  .send(request.getBody().map(dataBuffer ->
41  ((NettyDataBuffer) dataBuffer).getNativeBuffer()));
42  });
43
44  if (properties.getResponseTimeout() != null) {
45  responseMono.timeout(properties.getResponseTimeout(),
46  Mono.error(new TimeoutException("Response took longer than timeout: " +
47  properties.getResponseTimeout())));
48  }
49
50  return responseMono.doOnNext(res -> {
51  ServerHttpResponse response = exchange.getResponse();
52  // put headers and status so filters can modify the response
53  HttpHeaders headers = new HttpHeaders();
54
55  res.responseHeaders().forEach(entry -> headers.add(entry.getKey(), entry.getValue()));
56
57  if (headers.getContentType() != null) {
58  exchange.getAttributes().put(ORIGINAL_RESPONSE_CONTENT_TYPE_ATTR, headers.getContentType());
59  }
60
61  HttpHeaders filteredResponseHeaders = HttpHeadersFilter.filter(
62  this.headersFilters.getIfAvailable(), headers, exchange,
Type.RESPONSE);
63
64  response.getHeaders().putAll(filteredResponseHeaders);
65  HttpStatus status = HttpStatus.resolve(res.status().code());
```

```
66   if (status != null) {
67   response.setStatusCode(status);
68   } else if (response instanceof AbstractServerHttpResponse) {
69   // https://jira.spring.io/browse/SPR-16748
70   ((AbstractServerHttpResponse)
response).setStatusCodeValue(res.status().code());
71   } else {
72   throw new IllegalStateException("Unable to set status code on response:
" +res.status().code()+", "+response.getClass());
73   }
74
75   // Defer committing the response until all route filters have run
76   // Put client response as ServerWebExchange attribute and write respons
e later NettyWriteResponseFilter
77   exchange.getAttributes().put(CLIENT_RESPONSE_ATTR, res);
78   }).then(chain.filter(exchange));
79   }
```

Spring GateWay 主要相关类思维导图