# 微服务专题-一站式微服务架构SpringCloud

主讲：Cat 老师

**北京动力节点教育科技有限公司**  
**2019 • 北京**

**动力节点版权所有，禁止私自传播**

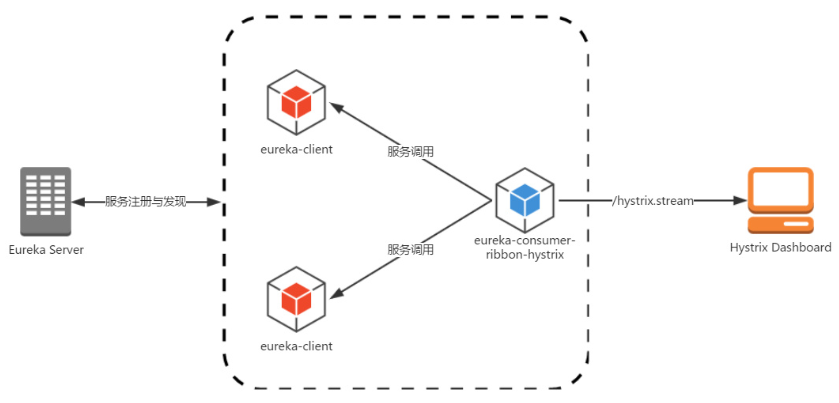
# 快速回顾

1. 分布式与微服务架构的理论梳理；
2. 什么是Spring Cloud？
3. Spring Cloud的整体架构 （与Dubbo比较）
4. 服务消费者Controller直连调用服务提供者Controller（http协议的restful）
5. Spring Cloud的注册中心Eureka
6. Spring Cloud Eureka与Zookeeper比较
7. Spring Cloud Eureka高可用集群
8. Spring Cloud Eureka自我保护机制
9. Spring Cloud Ribbon负载均衡
10. Spring Cloud Feign声明式服务调用（与Dubbo接口层比较）
11. Spring Cloud Hystrix 服务熔断降级、服务限流
12. Spring Cloud Hystrix DashBoard仪表盘监控

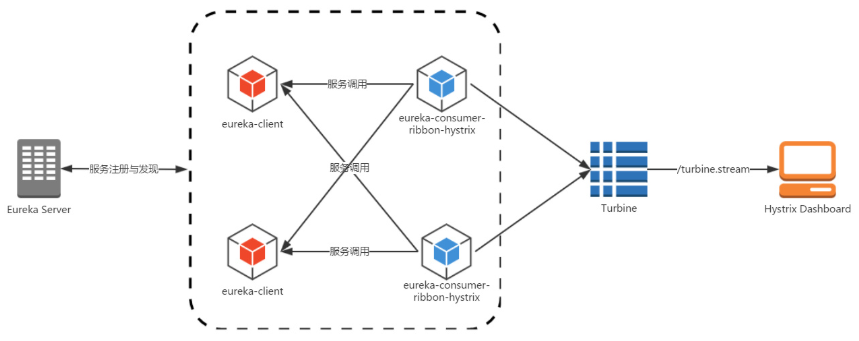
# Spring Cloud Hystrix Turbine

Hystrix Dashboard 前面已经知道了，它的主要功能是可以对某一项微服务进行监控，但真实情况下，不可能只对一个微服务进行监控，我们有很多微服务，所以我们需要对很多微服务进行监控，这个时候就需要使用到turbine [ˈtɜːbaɪn] 来完成；

**单个hystrix服务的监控（如下图）：**



**多个hystrix服务的监控（如下图）：**



具体步骤：

首先准备一个turbine模块

1. 创建一个34-springcloud-service-turbine项目，该项目依然是一个Springboot项目；
2. 添加依赖：

*<!-- spring-cloud-starter-netflix-turbine -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-turbine</**artifactId**>  
</**dependency**>

3、配置文件：

**server.port**=**3722**

*#不向注册中心注册自己* **eureka.client.register-with-eureka**=**false***#eureka注册中心的连接地址***eureka.client.service-url.defaultZone**=**http://192.168.10.128:8761/eureka,http://192.168.10.128:8762/eureka,http://192.168.10.128:8763/eureka***#配置turbine***turbine.app-config**=**34-SPRINGCLOUD-SERVICE-PORTAL,34-SPRINGCLOUD-SERVICE-PORTAL-2**

*#需要有这个，没有的话聚合不了多个项目* **turbine.cluster-name-expression**=**"default"**

4、在main方法的入口类上添加注解：

@EnableTurbine *//开启turbine*@SpringBootApplication  
**public class** TurbineApplication {  
 **public static void** main(String[] args) {  
 SpringApplication.*run*(TurbineApplication.**class**, args);  
 }  
}

接下来为了能对多个使用了hystrix的项目进行监控，我们再准备一份项目，并且里面使用了hystrix：34-springcloud-service-portal-2（可以拷贝一份34-springcloud-service-portal项目来制作）

注意点就是要先访问一下各个使用了hystrix的优熔断功能的接口，然后再测试turbine，否则你不访问的话，测试是不会有数据的；

<http://localhost:8080/cloud/goodsFeignHystrix>

<http://localhost:8081/cloud/goodsLimit>

<http://localhost:8080/actuator/hystrix.stream>

<http://localhost:8081/actuator/hystrix.stream>

<http://localhost:3722/turbine.stream>

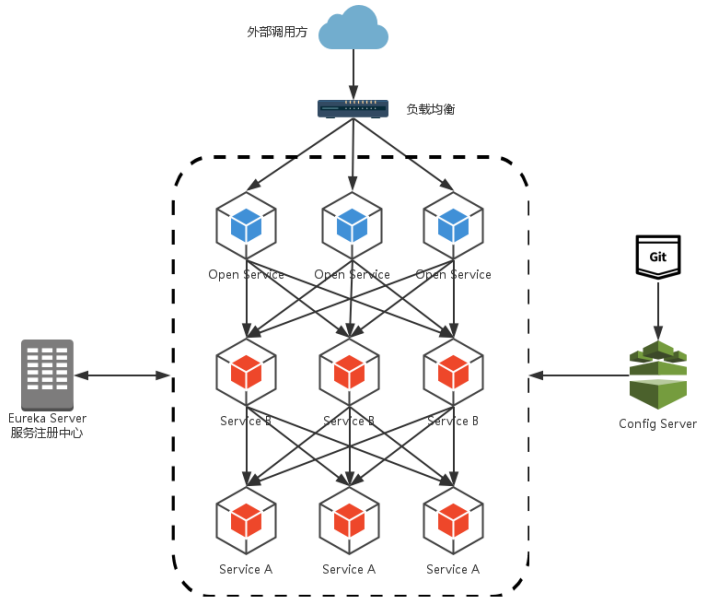
# Spring Cloud Zuul

通过前面内容的学习，我们已经可以基本搭建出一套简略版的微服务架构了，我

们有注册中心 Eureka，可以将服务注册到该注册中心中，我们有 Ribbon 或

Feign 可以实现对服务负载均衡地调用，我们有 Hystrix 可以实现服务的熔断；

我们来看一下下面的微服务架构图



在上面的架构图中，我们的服务包括：内部服务 Service A 和内部服务 Service

B，这两个服务都是集群部署，每个服务部署了 3 个实例，他们都会通过 Eureka

Server 注册中心注册与订阅服务，而 Open Service 是一个对外的服务，也是

集群部署，外部调用方通过负载均衡设备调用 Open Service 服务，比如负载均衡使用 Nginx、LVS、HAProxy，这样的实现是否合理，或者是否有更好的实现方式呢？接下来我们主要围绕该问题展开讨论。

1、如果我们的微服务中有很多个独立服务都要对外提供服务，那么我们要如何

去管理这些接口？特别是当项目非常庞大的情况下要如何管理？

2、在微服务中，一个独立的系统被拆分成了很多个独立的服务，为了确保安全，

权限管理也是一个不可回避的问题，如果在每一个服务上都添加上相同的权限验

证代码来确保系统不被非法访问，那么工作量也就太大了，而且维护也非常不方

便。

为了解决上述问题，微服务架构中提出了API网关的概念，它就像一个安检站

一样，所有外部的请求都需要经过它的调度与过滤，然后 API 网关来实现请

求路由、负载均衡、权限验证等功能；

那么 Spring Cloud 这个一站式的微服务开发框架基于 Netflix Zuul 实现了

Spring Cloud Zuul，采用 Spring Cloud Zuul 即可实现一套 API 网关服务；

## Zuul

Zuul包含了对请求的路由和过滤两个最主要的功能：

其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部访问统一入口的基础，过滤功能则负责对请求的处理过程进行干预，是实现请求校验、服务聚合等功能的基础；

Zuul和Eureka进行整合，将Zuul自身注册为Eureka服务治理下的应用，同时从Eureka中获得其他微服务的信息，也即以后的访问微服务都是通过Zuul跳转后获得。

## 路由功能：

项目加入依赖:

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

*<!-- spring-cloud-starter-netflix-zuul -->*<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-zuul</**artifactId**>  
</**dependency**>

由于Zuul最终会注册进eureka，所以我们此处也依赖了eureka；

配置文件：

**server.port**=**80***#是eureka注册中心首页的Application这一栏***spring.application.name**=**34-springcloud-service-zuul***#每间隔2s，向服务端发送一次心跳，证明自己依然"存活"***eureka.instance.lease-renewal-interval-in-seconds**=**2***#告诉服务端，如果我10s之内没有给你发心跳，就代表我故障了，将我踢出掉***eureka.instance.lease-expiration-duration-in-seconds**=**10***#告诉服务端，服务实例以IP作为链接，而不是取机器名***eureka.instance.prefer-ip-address**=**true***#告诉服务端，服务实例的id，id必须要唯一，是eureka注册中心首页的Status这一栏***eureka.instance.instance-id**=**34-springcloud-service-zuul***#eureka注册中心的连接地址***eureka.client.service-url.defaultZone**=**http://192.168.10.128:8761/eureka,http://192.168.10.128:8762/eureka,http://192.168.10.128:8763/eureka**

启动类上：

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

这样简单的zuul就搭建好了， 启动项目我们即可通过zuul然后加上对应的微服务名字访问微服务，比如：

<http://localhost/34-springcloud-service-portal/cloud/goodsFeign>

[http://localhost:80/](http://localhost/)

这个是zuul本身的

1. springcloud-service-portal

这个是要调用的项目名称

/cloud/goodsFeign

这个是被调用的contrller上的接口路径；

在实际开发当中我们肯定不会通过微服务名去调用，比如我要调用消费者可能只要一个/cloud/goodsFeign就好了，而不是

/34-springcloud-service-portal/cloud/goodsFeign

加入以下配置即可:

*#配置路由规则***zuul.routes.portal.service-id**=**34-springcloud-service-portal  
zuul.routes.portal.path**=**/portal/\*\***

**然后：<http://localhost/portal/cloud/goodsFeignHystrix>**

/ \*\*代表是所有（多个）层级 /cloud/goodsFeignHystrix

/ \* 是代表一层；

如果是/ \* 的话 /api/goods 就不会被路由；

此时我们能通过自定义的规则进行访问，但是我们现在依然能用之前的微服务名调用，这是不合理的，第一是有多重地址了， 第二一般微服务名这种最好不要暴露在外，所以我们一般会禁用微服务名方式调用。

加入配置:

**zuul.ignored-services**=34-springcloud-service-portal

这里能发现我们不能通过微服务名来调用了， 不过这个配置

如果一个一个通过微服务名来配置难免有点复杂，所以一般这样配置来禁用所有:

**zuul.ignored-services**=**\***

可能有时候我们的接口调用需要一定的规范，比如调用微服务的API URL前缀需要加上/api 对于这种情况，zuul也考虑到了并给出了解决方案：

**zuul.prefix**=**/api**

**比如：<http://localhost/api/portal/cloud/goodsFeignHystrix>**

### 通配符

|  |  |  |  |
| --- | --- | --- | --- |
| 通配符 | 含义 | 举例 | 说明 |
| ？ | 匹配任意单个字符 | /34-springcloud-service-feign/? | 匹配  /34-springcloud-service-feign/a,  /34-springcloud-service-feign/b,  /34-springcloud-service-feign/c 等 |
| \* | 匹配任意数量的字符 | /34-springcloud-service-feign/\* | 匹配  /34-springcloud-service-feign/aaa,  /34-springcloud-service-feign/bbb,  /34-springcloud-service-feign/ccc 等，  无法匹配  /34-springcloud-service-feign/a/b/c |
| \*\* | 匹配任意数量的字符 | /34-springcloud-service-feign/\*\* | 匹配  /34-springcloud-service-feign/aaa,  /34-springcloud-service-feign/bbb,  /34-springcloud-service-feign/ccc 等，  也可以匹配  /34-springcloud-service-feign/a/b/c |

## 过滤器:

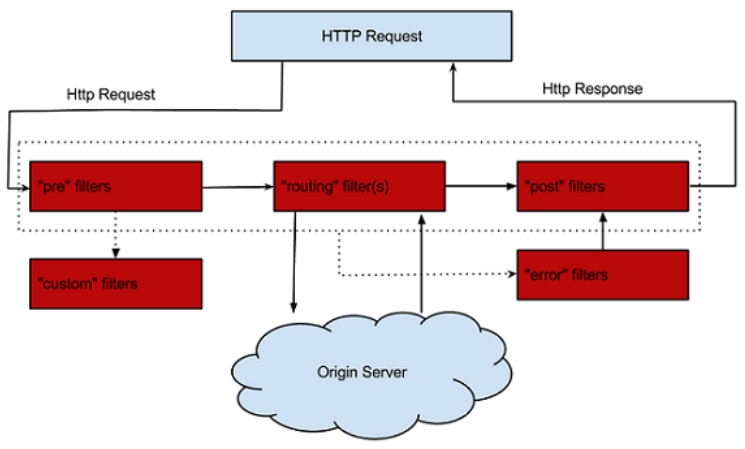
限流、权限验证、记录日志

过滤器 (filter) 是zuul的核心组件，zuul大部分功能都是通过过滤器来实现的。 zuul中定义了4种标准过滤器类型，这些过滤器类型对应于请求的典型生命周期。

PRE：这种过滤器在请求被路由之前调用。可利用这种过滤器实现身份验证、在 集群中选择请求的微服务、记录调试信息等。

ROUTING：这种过滤器将请求路由到微服务。这种过滤器用于构建发送给微服 务的请求，并使用 Apache HttpClient或 Netfilx Ribbon请求微服务

POST:这种过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准 的 HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端等。 ERROR：在其他阶段发生错误时执行该过滤器。



如果要编写一个过滤器，则需继承ZuulFilter类实现其中的方法:

@Component  
**public class** LogFilter **extends** ZuulFilter {  
  
 @Override  
 **public** String filterType() {  
 **return** FilterConstants.***ROUTE\_TYPE***;  
 }  
  
 @Override  
 **public int** filterOrder() {  
 **return** FilterConstants.***PRE\_DECORATION\_FILTER\_ORDER***;  
 }  
  
 @Override  
 **public boolean** shouldFilter() {  
 **return true**;  
 }  
  
 @Override  
 **public** Object run() **throws** ZuulException {  
 RequestContext currentContext = RequestContext.*getCurrentContext*();  
 HttpServletRequest request = currentContext.getRequest();  
 String remoteAddr = request.getServerName();  
 System.***out***.println(**"访问地址:"**+request.getRequestURI());  
 **return null**;  
 }  
}

由代码可知，自定义的 zuul Filter需实现以下几个方法。

filterType: 返回过滤器的类型。有 pre、 route、 post、 error等几种取值，分别对应上文的几种过滤器。

详细可以参考 com.netflix.zuul.ZuulFilter.filterType()中的注释；

filter0rder: 返回一个 int值来指定过滤器的执行顺序，不同的过滤器允许返回相同的数字。

shouldFilter：返回一个 boolean值来判断该过滤器是否要执行， true表示执行， false表示不执行。

run：过滤器的具体逻辑；

## zuul过滤器的禁用

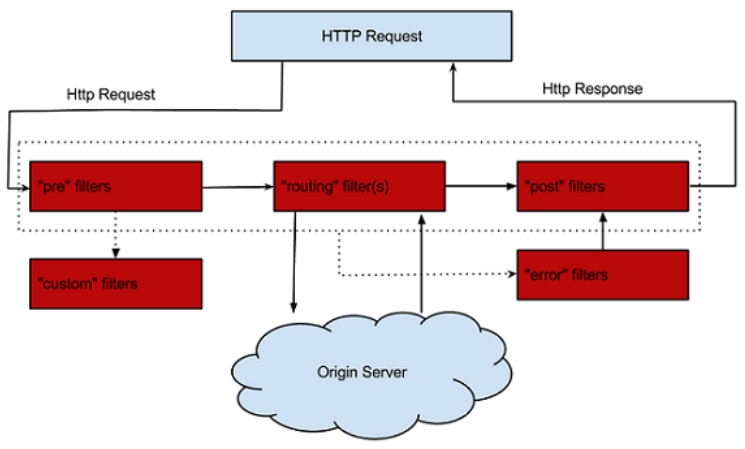
Spring Cloud默认为Zuul编写并启用了一些过滤器，例如DebugFilter、 FormBodyWrapperFilter等，这些过滤器都存放在spring-cloud-netflix-zuul这个jar包里，一些场景下，想要禁用掉部分过滤器，该怎么办呢？ 只需在application.properties里设置zuul...disable=true 例如，要禁用上面我们写的过滤器，这样配置就行了：

**zuul.LogFilter.route.disable**=**true**

## **Zuul 的异常处理**

Spring Cloud Zuul 对异常的处理是非常方便的，但是由于 Spring Cloud 处于迅速发展中，各个版本之间有所差异，本案例是以 Finchley.RELEASE 版本为例， 来说明 Spring Cloud Zuul 中的异常处理问题。

首先我们来看一张官方给出的 Zuul 请求的生命周期图：



1.正常情况下所有的请求都是按照 pre、route、post 的顺序来执行，然后由 post 返回 response

2.在 pre 阶段，如果有自定义的过滤器则执行自定义的过滤器

3.pre、routing、post 的任意一个阶段如果抛异常了，则执行 error 过滤器

我们可以统一处理异常：

怎么实现，步骤：

1、禁用 zuul 默认的异常处理 SendErrorFilter 过滤器，然后自定义我们自己的Errorfilter 过滤器

**zuul.SendErrorFilter.error.disable**=**true**

@Component  
**public class** ErrorFilter **extends** ZuulFilter {  
  
 @Override  
 **public** String filterType() {  
 **return "error"**;  
 }  
 @Override  
 **public int** filterOrder() {  
 **return** 1;  
 }  
 @Override  
 **public boolean** shouldFilter() {  
 **return true**;  
 }  
 @Override  
 **public** Object run() **throws** ZuulException {  
 **try** {  
 RequestContext context = RequestContext.*getCurrentContext*();  
 ZuulException exception = (ZuulException)context.getThrowable();  
 System.***out***.println(**"进入系统异常拦截"** + exception.getMessage());  
 HttpServletResponse response = context.getResponse();  
 response.setContentType(**"application/json; charset=utf8"**);  
 response.setStatus(exception.**nStatusCode**);  
 PrintWriter writer = **null**;  
 **try** {  
 writer = response.getWriter();  
 writer.print(**"{code:"**+ exception.**nStatusCode** +**",message:\""**+  
 exception.getMessage() +**"\"}"**);  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } **finally** {  
 **if**(writer!=**null**){  
 writer.close();  
 }  
 }  
 } **catch** (Exception e) {  
 ReflectionUtils.*rethrowRuntimeException*(e);  
 }  
 **return null**;  
 }  
}

## Zuul的熔断降级

zuul是一个代理服务，但如果被代理的服务突然断了，这个时候zuul上面会有出错信息，例如，停止了被调用的微服务；

一般服务方自己会进行服务的熔断降级，但对于zuul本身，也应该进行zuul的降级处理；

我们需要有一个zuul的降级，实现如下：

@Component  
**public class** ProviderFallback **implements** FallbackProvider {  
 @Override  
 **public** String getRoute() {  
 **return "\*"**;  
 }  
  
 @Override  
 **public** ClientHttpResponse fallbackResponse(String route, Throwable cause) {  
 **return new** ClientHttpResponse() {  
 @Override  
 **public** HttpHeaders getHeaders() {  
 HttpHeaders headers = **new** HttpHeaders();  
 headers.set(**"Content-Type"**, **"text/html; charset=UTF-8"**);  
 **return** headers;  
 }  
  
 @Override  
 **public** InputStream getBody() **throws** IOException {  
 *// 响应体* **return new** ByteArrayInputStream(**"服务正在维护，请稍后再试."**.getBytes());  
 }  
  
 @Override  
 **public** HttpStatus getStatusCode() **throws** IOException {  
 **return** HttpStatus.***BAD\_REQUEST***;  
 }  
  
 @Override  
 **public int** getRawStatusCode() **throws** IOException {  
 **return** HttpStatus.***BAD\_REQUEST***.value();  
 }  
  
 @Override  
 **public** String getStatusText() **throws** IOException {  
 **return** HttpStatus.***BAD\_REQUEST***.getReasonPhrase();  
 }  
  
 @Override  
 **public void** close() {  
  
 }  
 };  
 }  
}