**Spring Cloud构建微服务架构（五）服务网关**

**原创**

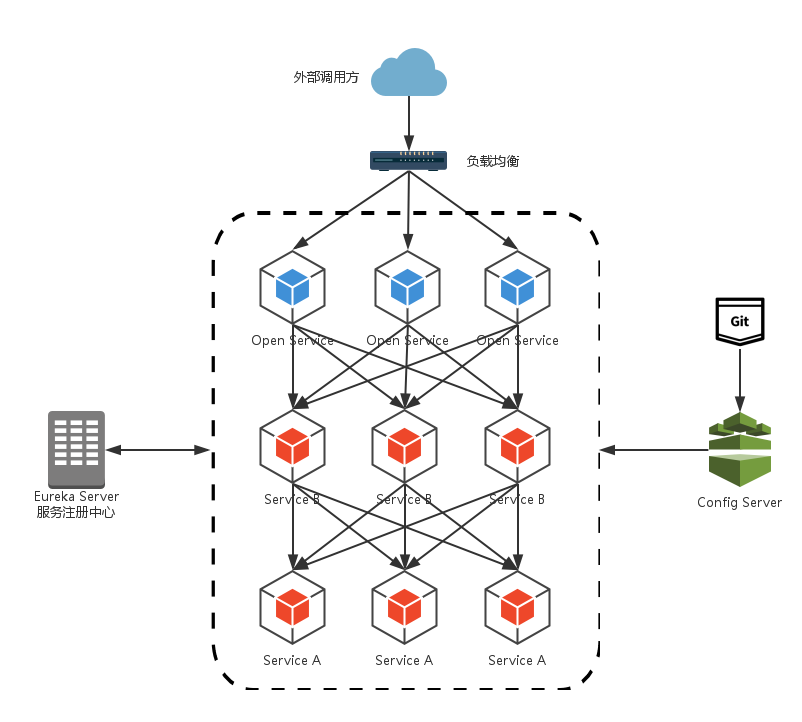
[2016-07-12](http://blog.didispace.com/springcloud5/)

 翟永超

[Spring Cloud](http://blog.didispace.com/categories/Spring-Cloud/)

被围观 **48750** 次

通过之前几篇Spring Cloud中几个核心组件的介绍，我们已经可以构建一个简略的（不够完善）微服务架构了。比如下图所示：

[](http://blog.didispace.com/content/images/2016/07/-----.png)alt

我们使用Spring Cloud Netflix中的Eureka实现了服务注册中心以及服务注册与发现；而服务间通过Ribbon或Feign实现服务的消费以及均衡负载；通过Spring Cloud Config实现了应用多环境的外部化配置以及版本管理。为了使得服务集群更为健壮，使用Hystrix的融断机制来避免在微服务架构中个别服务出现异常时引起的故障蔓延。

在该架构中，我们的服务集群包含：内部服务Service A和Service B，他们都会注册与订阅服务至Eureka Server，而Open Service是一个对外的服务，通过均衡负载公开至服务调用方。本文我们把焦点聚集在对外服务这块，这样的实现是否合理，或者是否有更好的实现方式呢？

先来说说这样架构需要做的一些事儿以及存在的不足：

* 首先，破坏了服务无状态特点。为了保证对外服务的安全性，我们需要实现对服务访问的权限控制，而开放服务的权限控制机制将会贯穿并污染整个开放服务的业务逻辑，这会带来的最直接问题是，破坏了服务集群中REST API无状态的特点。从具体开发和测试的角度来说，在工作中除了要考虑实际的业务逻辑之外，还需要额外可续对接口访问的控制处理。
* 其次，无法直接复用既有接口。当我们需要对一个即有的集群内访问接口，实现外部服务访问时，我们不得不通过在原有接口上增加校验逻辑，或增加一个代理调用来实现权限控制，无法直接复用原有的接口。

面对类似上面的问题，我们要如何解决呢？下面进入本文的正题：服务网关！

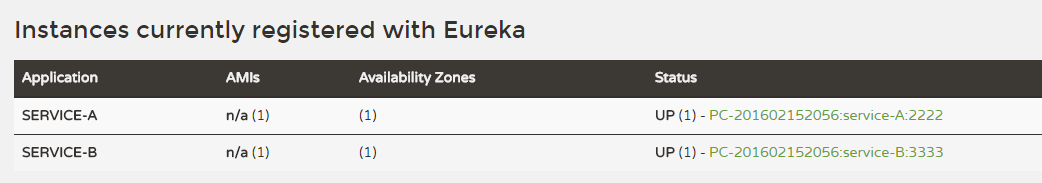
为了解决上面这些问题，我们需要将权限控制这样的东西从我们的服务单元中抽离出去，而最适合这些逻辑的地方就是处于对外访问最前端的地方，我们需要一个更强大一些的均衡负载器，它就是本文将来介绍的：服务网关。

服务网关是微服务架构中一个不可或缺的部分。通过服务网关统一向外系统提供REST API的过程中，除了具备服务路由、均衡负载功能之外，它还具备了权限控制等功能。Spring Cloud Netflix中的Zuul就担任了这样的一个角色，为微服务架构提供了前门保护的作用，同时将权限控制这些较重的非业务逻辑内容迁移到服务路由层面，使得服务集群主体能够具备更高的可复用性和可测试性。

下面我们通过实例例子来使用一下Zuul来作为服务的路有功能。

**准备工作**

在使用Zuul之前，我们先构建一个服务注册中心、以及两个简单的服务，比如：我构建了一个service-A，一个service-B。然后启动eureka-server和这两个服务。通过访问eureka-server，我们可以看到service-A和service-B已经注册到了服务中心。

[](http://blog.didispace.com/content/images/2016/07/EEE3-tm.png)alt

如果您还不熟悉如何构建服务中心和注册服务，请先阅读[Spring Cloud构建微服务架构（一）服务注册与发现](http://blog.didispace.com/springcloud1/)。

如果您不想自己动手准备，可以从这里获取示例代码：<http://git.oschina.net/didispace/SpringBoot-Learning>

**开始使用Zuul**

* 引入依赖spring-cloud-starter-zuul、spring-cloud-starter-eureka，如果不是通过指定serviceId的方式，eureka依赖不需要，但是为了对服务集群细节的透明性，还是用serviceId来避免直接引用url的方式吧。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-zuul</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency> |

* 应用主类使用@EnableZuulProxy注解开启Zuul

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @EnableZuulProxy  @SpringCloudApplication  **public** **class** **Application** {  **public** **static** **void** **main**(String[] args) {  **new** SpringApplicationBuilder(Application.class).web(**true**).run(args);  }  } |

*这里用了@SpringCloudApplication注解，之前没有提过，通过源码我们看到，它整合了@SpringBootApplication、@EnableDiscoveryClient、@EnableCircuitBreaker，主要目的还是简化配置。这几个注解的具体作用这里就不做详细介绍了，之前的文章已经都介绍过。*

* application.properties中配置Zuul应用的基础信息，如：应用名、服务端口等。

|  |  |
| --- | --- |
| 1  2 | spring.application.name=api-gateway  server.port=5555 |

**Zuul配置**

完成上面的工作后，Zuul已经可以运行了，但是如何让它为我们的微服务集群服务，还需要我们另行配置，下面详细的介绍一些常用配置内容。

**服务路由**

通过服务路由的功能，我们在对外提供服务的时候，只需要通过暴露Zuul中配置的调用地址就可以让调用方统一的来访问我们的服务，而不需要了解具体提供服务的主机信息了。

在Zuul中提供了两种映射方式：

* 通过url直接映射，我们可以如下配置：

|  |  |
| --- | --- |
| 1  2  3 | # routes to url  zuul.routes.api-a-url.path=/api-a-url/\*\*  zuul.routes.api-a-url.url=http://localhost:2222/ |

该配置，定义了，所有到Zuul的中规则为：/api-a-url/\*\*的访问都映射到http://localhost:2222/上，也就是说当我们访问http://localhost:5555/api-a-url/add?a=1&b=2的时候，Zuul会将该请求路由到：http://localhost:2222/add?a=1&b=2上。

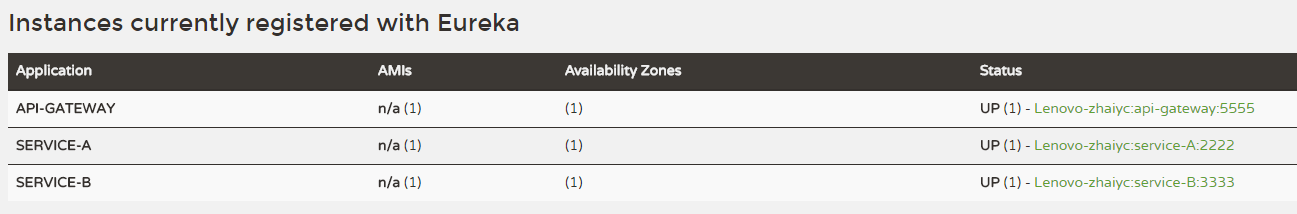
其中，配置属性zuul.routes.api-a-url.path中的api-a-url部分为路由的名字，可以任意定义，但是一组映射关系的path和url要相同，下面讲serviceId时候也是如此。

* 通过url映射的方式对于Zuul来说，并不是特别友好，Zuul需要知道我们所有为服务的地址，才能完成所有的映射配置。而实际上，我们在实现微服务架构时，服务名与服务实例地址的关系在eureka server中已经存在了，所以只需要将Zuul注册到eureka server上去发现其他服务，我们就可以实现对serviceId的映射。例如，我们可以如下配置：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | zuul.routes.api-a.path=/api-a/\*\*  zuul.routes.api-a.serviceId=service-A  zuul.routes.api-b.path=/api-b/\*\*  zuul.routes.api-b.serviceId=service-B  eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/ |

针对我们在准备工作中实现的两个微服务service-A和service-B，定义了两个路由api-a和api-b来分别映射。另外为了让Zuul能发现service-A和service-B，也加入了eureka的配置。

接下来，我们将eureka-server、service-A、service-B以及这里用Zuul实现的服务网关启动起来，在eureka-server的控制页面中，我们可以看到分别注册了service-A、service-B以及api-gateway

[](http://blog.didispace.com/content/images/2016/07/45F-tm.png)alt

尝试通过服务网关来访问service-A和service-B，根据配置的映射关系，分别访问下面的url

* http://localhost:5555/api-a/add?a=1&b=2：通过serviceId映射访问service-A中的add服务
* http://localhost:5555/api-b/add?a=1&b=2：通过serviceId映射访问service-B中的add服务
* http://localhost:5555/api-a-url/add?a=1&b=2：通过url映射访问service-A中的add服务

*推荐使用serviceId的映射方式，除了对Zuul维护上更加友好之外，serviceId映射方式还支持了断路器，对于服务故障的情况下，可以有效的防止故障蔓延到服务网关上而影响整个系统的对外服务*

**服务过滤**

在完成了服务路由之后，我们对外开放服务还需要一些安全措施来保护客户端只能访问它应该访问到的资源。所以我们需要利用Zuul的过滤器来实现我们对外服务的安全控制。

在服务网关中定义过滤器只需要继承ZuulFilter抽象类实现其定义的四个抽象函数就可对请求进行拦截与过滤。

比如下面的例子，定义了一个Zuul过滤器，实现了在请求被路由之前检查请求中是否有accessToken参数，若有就进行路由，若没有就拒绝访问，返回401 Unauthorized错误。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | **public** **class** **AccessFilter** **extends** **ZuulFilter** {  **private** **static** Logger log = LoggerFactory.getLogger(AccessFilter.class);  @Override  **public** String **filterType**() {  **return** "pre";  }  @Override  **public** **int** **filterOrder**() {  **return** 0;  }  @Override  **public** **boolean** **shouldFilter**() {  **return** **true**;  }  @Override  **public** Object **run**() {  RequestContext ctx = RequestContext.getCurrentContext();  HttpServletRequest request = ctx.getRequest();  log.info(String.format("%s request to %s", request.getMethod(), request.getRequestURL().toString()));  Object accessToken = request.getParameter("accessToken");  **if**(accessToken == **null**) {  log.warn("access token is empty");  ctx.setSendZuulResponse(**false**);  ctx.setResponseStatusCode(401);  **return** **null**;  }  log.info("access token ok");  **return** **null**;  }  } |

自定义过滤器的实现，需要继承ZuulFilter，需要重写实现下面四个方法：

* filterType：返回一个字符串代表过滤器的类型，在zuul中定义了四种不同生命周期的过滤器类型，具体如下：
  + pre：可以在请求被路由之前调用
  + routing：在路由请求时候被调用
  + post：在routing和error过滤器之后被调用
  + error：处理请求时发生错误时被调用
* filterOrder：通过int值来定义过滤器的执行顺序
* shouldFilter：返回一个boolean类型来判断该过滤器是否要执行，所以通过此函数可实现过滤器的开关。在上例中，我们直接返回true，所以该过滤器总是生效。
* run：过滤器的具体逻辑。需要注意，这里我们通过ctx.setSendZuulResponse(false)令zuul过滤该请求，不对其进行路由，然后通过ctx.setResponseStatusCode(401)设置了其返回的错误码，当然我们也可以进一步优化我们的返回，比如，通过ctx.setResponseBody(body)对返回body内容进行编辑等。

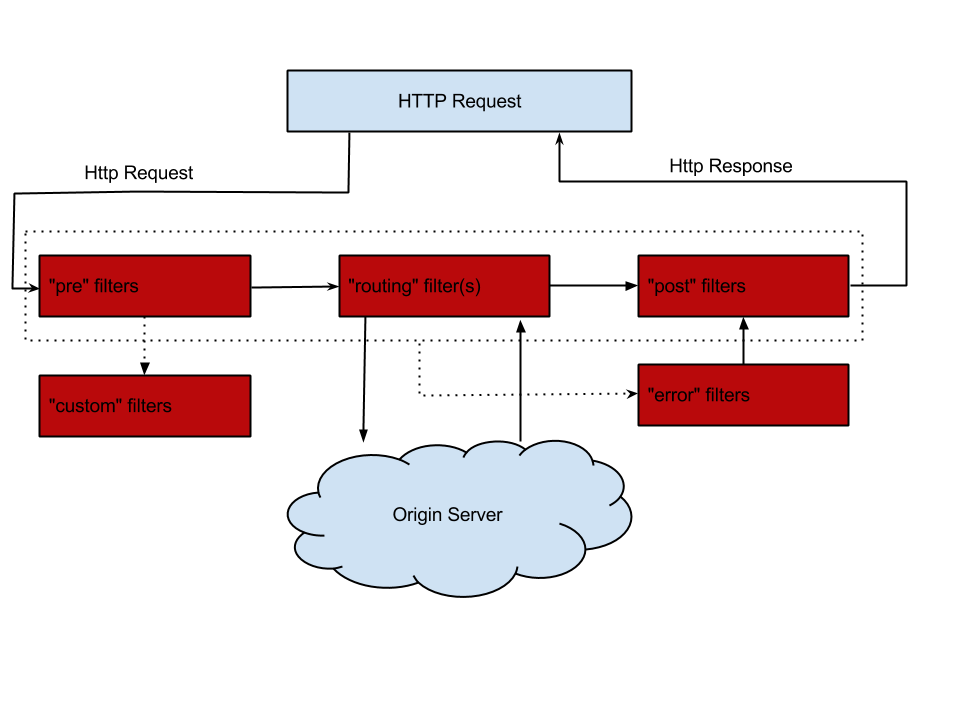
在实现了自定义过滤器之后，还需要实例化该过滤器才能生效，我们只需要在应用主类中增加如下内容：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @EnableZuulProxy  @SpringCloudApplication  public class Application {  public static void main(String[] args) {  new SpringApplicationBuilder(Application.class).web(true).run(args);  }  @Bean  public AccessFilter accessFilter() {  return new AccessFilter();  }  } |

启动该服务网关后，访问：

* http://localhost:5555/api-a/add?a=1&b=2：返回401错误
* http://localhost:5555/api-a/add?a=1&b=2&accessToken=token：正确路由到server-A，并返回计算内容

对于其他一些过滤类型，这里就不一一展开了，根据之前对filterType生命周期介绍，可以参考下图去理解，并根据自己的需要在不同的生命周期中去实现不同类型的过滤器。

[](http://blog.didispace.com/content/images/2016/07/687474703a2f2f6e6574666c69782e6769746875622e696f2f7a75756c2f696d616765732f7a75756c2d726571756573742d6c6966656379636c652e706e67.png)alt

最后，总结一下为什么服务网关是微服务架构的重要部分，是我们必须要去做的原因：

* 不仅仅实现了路由功能来屏蔽诸多服务细节，更实现了服务级别、均衡负载的路由。
* 实现了接口权限校验与微服务业务逻辑的解耦。通过服务网关中的过滤器，在各生命周期中去校验请求的内容，将原本在对外服务层做的校验前移，保证了微服务的无状态性，同时降低了微服务的测试难度，让服务本身更集中关注业务逻辑的处理。
* 实现了断路器，不会因为具体微服务的故障而导致服务网关的阻塞，依然可以对外服务。

搜索生生世世三