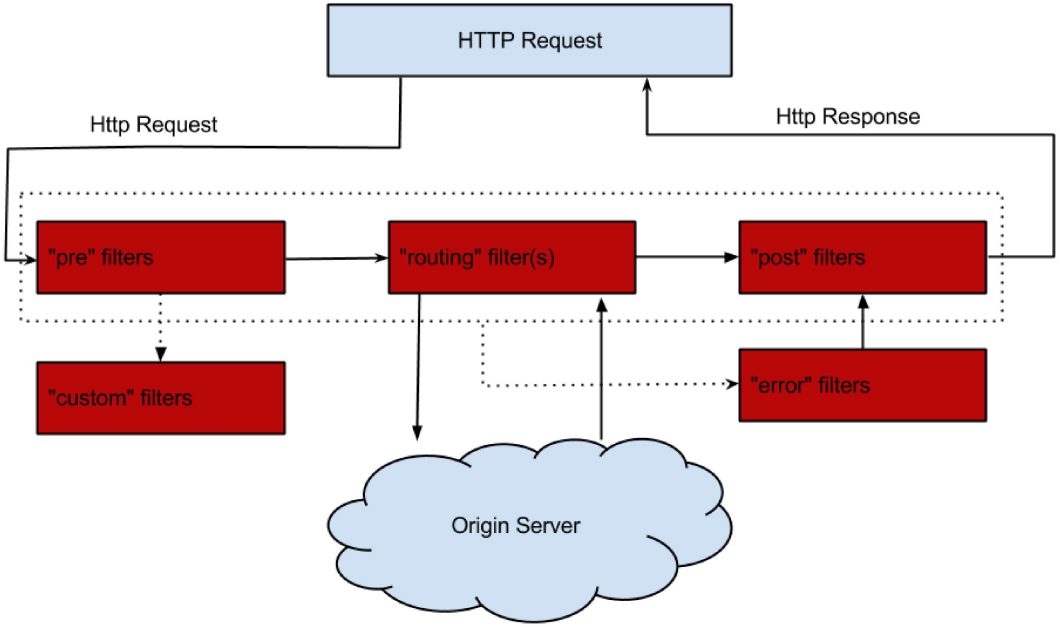
时间过的很快，写springcloud(十)：服务网关zuul初级篇还在半年前，现在已经是2018年了，我们继续探讨Zuul更高级的使用方式。

上篇文章主要介绍了Zuul网关使用模式，以及自动转发机制，但其实Zuul还有更多的应用场景，比如：鉴权、流量转发、请求统计等等，这些功能都可以使用Zuul来实现。

Zuul的核心

Filter是Zuul的核心，用来实现对外服务的控制。Filter的生命周期有4个，分别是“PRE”、“ROUTING”、“POST”、“ERROR”，整个生命周期可以用下图来表示。



Zuul大部分功能都是通过过滤器来实现的，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。我们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。
  除了默认的过滤器类型，Zuul还允许我们创建自定义的过滤器类型。例如，我们可以定制一种STATIC类型的过滤器，直接在Zuul中生成响应，而不将请求转发到后端的微服务。

Zuul中默认实现的Filter

类型	顺序	过滤器	功能
pre	-3	ServletDetectionFilter	标记处理Servlet的类型
pre	-2	Servlet30WrapperFilter	包装HttpServletRequest请求
pre	-1	FormBodyWrapperFilter	包装请求体
route	1	DebugFilter	标记调试标志
route	5	PreDecorationFilter	处理请求上下文供后续使用
route	10	RibbonRoutingFilter	serviceId请求转发
route	100	SimpleHostRoutingFilter	url请求转发
route	500	SendForwardFilter	forward请求转发
post	0	SendErrorFilter	处理有错误的请求响应
post	1000	SendResponseFilter	处理正常的请求响应

禁用指定的Filter

可以在application.yml中配置需要禁用的filter，格式：

```
zuul:
    FormBodyWrapperFilter:
        pre:
            disable: true
```

自定义Filter

实现自定义Filter，需要继承ZuulFilter的类，并覆盖其中的4个方法。

```
public class MyFilter extends ZuulFilter {
    @Override
    String filterType() {
        return "pre"; //定义filter的类型，有pre、route、post、error四种
    }

    @Override
    int filterOrder() {
        return 10; //定义filter的顺序，数字越小表示顺序越高，越先执行
    }

    @Override
    boolean shouldFilter() {
        return true; //表示是否需要执行该filter，true表示执行，false表示不执行
    }

    @Override
    Object run() {
        return null; //filter需要执行的具体操作
    }
}
```

自定义Filter示例

我们假设有这样一个场景，因为服务网关应对的是外部的所有请求，为了避免产生安全隐患，我们需要对请求做一定的限制，比如请求中含有Token便让请求继续往下走，如果请求不带Token就直接返回并给出提示。

首先自定义一个Filter，在run()方法中验证参数是否含有Token。

```
public class TokenFilter extends ZuulFilter {

    private final Logger logger = LoggerFactory.getLogger(TokenFilter.class);

    @Override
    public String filterType() {
        return "pre"; // 可以在请求被路由之前调用
    }

    @Override
    public int filterOrder() {
        return 0; // filter执行顺序，通过数字指定 ,优先级为0，数字越大，优先级越低
    }

    @Override
    public boolean shouldFilter() {
        return true;// 是否执行该过滤器，此处为true，说明需要过滤
    }

    @Override
    public Object run() {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();

        logger.info("--->>> TokenFilter {},{}", request.getMethod(), request.getRequestURL().toString());

        String token = request.getParameter("token");// 获取请求的参数

        if (StringUtils.isNotBlank(token)) {
            ctx.setSendZuulResponse(true); //对请求进行路由
            ctx.setResponseStatusCode(200);
            ctx.set("isSuccess", true);
            return null;
        } else {
            ctx.setSendZuulResponse(false); //不对其进行路由
            ctx.setResponseStatusCode(400);
```

```
            ctx.setResponseBody("token is empty");
            ctx.set("isSuccess", false);
            return null;
        }
    }

}
```

将TokenFilter加入到请求拦截队列，在启动类中添加以下代码：

```
@Bean
public TokenFilter tokenFilter() {
    return new TokenFilter();
}
```

这样就将我们自定义好的Filter加入到了请求拦截中。

测试

我们依次启动示例项目：`spring-cloud-eureka`、`spring-cloud-producer`、`spring-cloud-zuul`，这个三个项目均为上一篇示例项目，`spring-cloud-zuul`稍微进行改造。

访问地址：`http://localhost:8888/spring-cloud-producer/hello?name=neo`，返回：token is empty，请求被拦截返回。
访问地址：`http://localhost:8888/spring-cloud-producer/hello?name=neo&token=xx`，返回：hello neo，this is first messge，说明请求正常响应。

通过上面这例子我们可以看出，我们可以使用“PRE"类型的Filter做很多的验证工作，在实际使用中我们可以结合shiro、oauth2.0等技术去做鉴权、验证。

路由熔断

当我们的后端服务出现异常的时候，我们不希望将异常抛出给最外层，期望服务可以自动进行一降级。Zuul给我们提供了这样的支持。当某个服务出现异常时，直接返回我们预设的信息。

我们通过自定义的fallback方法，并且将其指定给某个route来实现该route访问出问题的熔断处理。主要继承ZuulFallbackProvider接口来实现，ZuulFallbackProvider默认有两个方法，一个用来指明熔断拦截哪个服务，一个定制返回内容。

```
public interface ZuulFallbackProvider {
   /**
     * The route this fallback will be used for.
     * @return The route the fallback will be used for.
     */
    public String getRoute();

    /**
     * Provides a fallback response.
     * @return The fallback response.
     */
    public ClientHttpResponse fallbackResponse();
}
```

实现类通过实现getRoute方法，告诉Zuul它是负责哪个route定义的熔断。而fallbackResponse方法则是告诉 Zuul 断路出现时，它会提供一个什么返回值来处理请求。

后来Spring又扩展了此类，丰富了返回方式，在返回的内容中添加了异常信息，因此最新版本建议直接继承类 `FallbackProvider` 。

我们以上面的spring-cloud-producer服务为例，定制它的熔断返回内容。

```
@Component
public class ProducerFallback implements FallbackProvider {
    private final Logger logger = LoggerFactory.getLogger(FallbackProvider.class);

    //指定要处理的 service。
    @Override
    public String getRoute() {
        return "spring-cloud-producer";
    }

    public ClientHttpResponse fallbackResponse() {
        return new ClientHttpResponse() {
            @Override
            public HttpStatus getStatusCode() throws IOException {
                return HttpStatus.OK;
            }

            @Override
            public int getRawStatusCode() throws IOException {
```

```
                return 200;
            }

            @Override
            public String getStatusText() throws IOException {
                return "OK";
            }

            @Override
            public void close() {

            }

            @Override
            public InputStream getBody() throws IOException {
                return new ByteArrayInputStream("The service is unavailable.".getBytes());
            }

            @Override
            public HttpHeaders getHeaders() {
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_JSON);
                return headers;
            }
        };
    }

    @Override
    public ClientHttpResponse fallbackResponse(Throwable cause) {
        if (cause != null && cause.getCause() != null) {
            String reason = cause.getCause().getMessage();
            logger.info("Excption {}",reason);
        }
        return fallbackResponse();
    }
}
```

当服务出现异常时，打印相关异常信息，并返回"The service is unavailable."。

启动项目spring-cloud-producer-2，这时候服务中心会有两个spring-cloud-producer项目，我们重启Zuul项目。再手动关闭spring-cloud-producer-2项目，多次访问地址：`http://localhost:8888/spring-cloud-producer/hello?name=neo&token=xx`，会交替返回：

```
hello neo, this is first messge
The service is unavailable.
...
```

根据返回结果可以看出：spring-cloud-producer-2项目已经启用了熔断，返回：`The service is unavailable.`

> Zuul 目前只支持服务级别的熔断，不支持具体到某个URL进行熔断。

路由重试

有时候因为网络或者其它原因，服务可能会暂时的不可用，这个时候我们希望可以再次对服务进行重试，Zuul也帮我们实现了此功能，需要结合Spring Retry 一起来实现。下面我们以上面的项目为例做演示。

添加Spring Retry依赖

首先在spring-cloud-zuul项目中添加Spring Retry依赖。

```
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
</dependency>
```

开启Zuul Retry

再配置文件中配置启用Zuul Retry

```
#是否开启重试功能
zuul.retryable=true
#对当前服务的重试次数
ribbon.MaxAutoRetries=2
#切换相同Server的次数
ribbon.MaxAutoRetriesNextServer=0
```

这样我们就开启了Zuul的重试功能。

测试

我们对spring-cloud-producer-2进行改造，在hello方法中添加定时，并且在请求的一开始打印参数。

```
@RequestMapping("/hello")
public String index(@RequestParam String name) {
    logger.info("request two name is "+name);
    try{
        Thread.sleep(1000000);
    }catch ( Exception e){
        logger.error(" hello two error",e);
    }
    return "hello "+name+", this is two messge";
}
```

重启 spring-cloud-producer-2和spring-cloud-zuul项目。

访问地址：`http://localhost:8888/spring-cloud-producer/hello?name=neo&token=xx`，当页面返回：`The service is unavailable.`时查看项目spring-cloud-producer-2后台日志如下：

```
2018-01-22 19:50:32.401  INFO 19488 --- [io-9001-exec-14] o.s.c.n.z.f.route.FallbackProvider       : request two name is neo
2018-01-22 19:50:33.402  INFO 19488 --- [io-9001-exec-15] o.s.c.n.z.f.route.FallbackProvider       : request two name is neo
2018-01-22 19:50:34.404  INFO 19488 --- [io-9001-exec-16] o.s.c.n.z.f.route.FallbackProvider       : request two name is neo
```

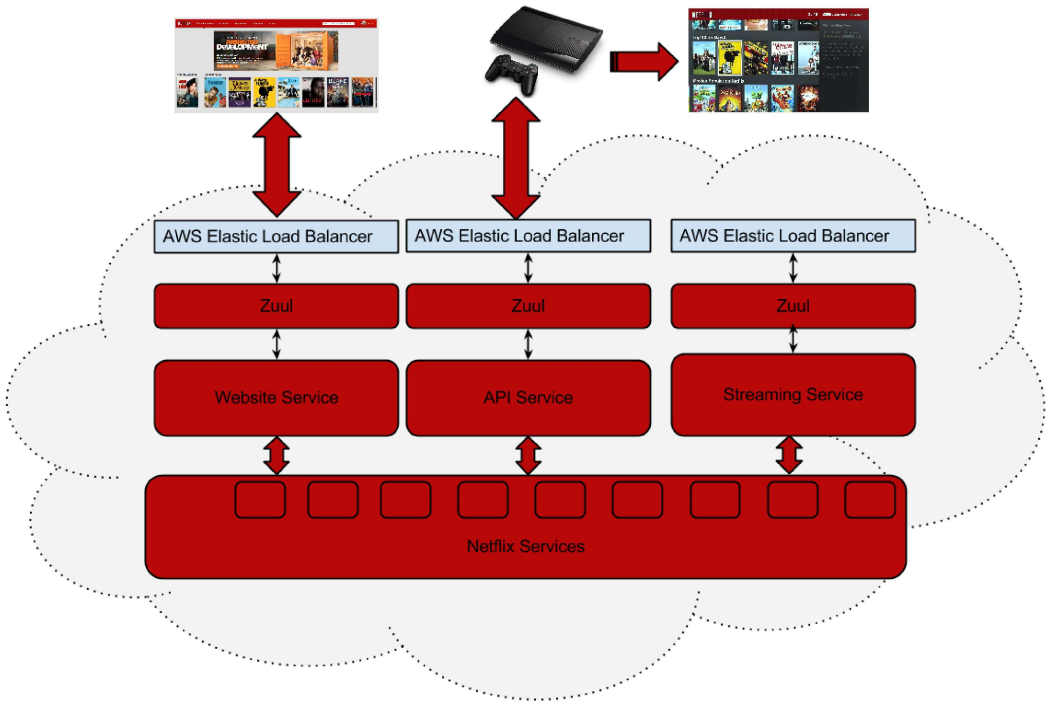
说明进行了三次的请求，也就是进行了两次的重试。这样也就验证了我们的配置信息，完成了Zuul的重试功能。

注意

开启重试在某些情况下是有问题的，比如当压力过大，一个实例停止响应时，路由将流量转到另一个实例，很有可能导致最终所有的实例全被压垮。说到底，断路器的其中一个作用就是防止故障或者压力扩散。用了retry，断路器就只有在该服务的所有实例都无法运作的情况下才能起作用。这种时候，断路器的形式更像是提供一种友好的错误信息，或者假装服务正常运行的假象给使用者。

不用retry，仅使用负载均衡和熔断，就必须考虑到是否能够接受单个服务实例关闭和eureka刷新服务列表之间带来的短时间的熔断。如果可以接受，就无需使用retry。

Zuul高可用



我们实际使用Zuul的方式如上图，不同的客户端使用不同的负载将请求分发到后端的Zuul，Zuul在通过Eureka调用后端服务，最后对外输出。因此为了保证Zuul的高可用性，前端可以同时启动多个Zuul实例进行负载，在Zuul的前端使用Nginx或者F5进行负载转发以达到高可用性。