本文我们将具体介绍一下Spring Cloud Zuul的另一项核心功能：过滤器。

过滤器的作用

通过上面两节，我们已经能够实现请求的路由功能，所以我们的微服务应用提供的接口就可以通过统一的API网关入口被客户端访问到了。但是，每个客户端用户请求微服务应用提供的接口时，它们的访问权限往往都需要有一定的限制，系统并不会将所有的微服务接口都对它们开放。然而，目前的服务路由并没有限制权限这样的功能，所有请求都会被毫无保留地转发到具体的应用并返回结果，为了实现对客户端请求的安全校验和权限控制，最简单和粗暴的方法就是为每个微服务应用都实现一套用于校验签名和鉴别权限的过滤器或拦截器。不过，这样的做法并不可取，它会增加日后的系统维护难度，因为同一个系统中的各种校验逻辑很多情况下都是大致相同或类似的，这样的实现方式会使得相似的校验逻辑代码被分散到了各个微服务中去，冗余代码的出现是我们不希望看到的。所以，比较好的做法是将这些校验逻辑剥离出去，构建出一个独立的鉴权服务。在完成了剥离之后，有不少开发者会直接在微服务应用中通过调用鉴权服务来实现校验，但是这样的做法仅仅只是解决了鉴权逻辑的分离，并没有在本质上将这部分不属于业余的逻辑拆分出原有的微服务应用，冗余的拦截器或过滤器依然会存在。

对于这样的问题，更好的做法是通过前置的网关服务来完成这些非业务性质的校验。由于网关服务的加入，外部客户端访问我们的系统已经有了统一入口，既然这些校验与具体业务无关，那何不在请求到达的时候就完成校验和过滤，而不是转发后再过滤而导致更长的请求延迟。同时，通过在网关中完成校验和过滤，微服务应用端就可以去除各种复杂的过滤器和拦截器了，这使得微服务应用的接口开发和测试复杂度也得到了相应的降低。

为了在API网关中实现对客户端请求的校验，我们将需要使用到Spring Cloud Zuul的另外一个核心功能：**过滤器**。

Zuul允许开发者在API网关上通过定义过滤器来实现对请求的拦截与过滤，实现的方法非常简单，我们只需要继承ZuulFilter抽象类并实现它定义的四个抽象函数就可以完成对请求的拦截和过滤了。

过滤器的实现

例如下面的代码，我们定义了一个简单的Zuul过滤器，它实现了在请求被路由之前检查HttpServletRequest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。

```
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("send {} request to {}", request.getMethod(),
request.getRequestURL().toString());

        //检查HttpServletRequest中是否有accessToken参数
        Object accessToken = request.getParameter("accessToken");
        //若没有就拒绝访问
        if(accessToken == null) {
            log.warn("access token is empty");
            ctx.setSendZuulResponse(false);
            //返回401 Unauthorized错误
            ctx.setResponseStatusCode(401);
            return null;
        }
        log.info("access token ok");
        return null;
    }
}
```

在上面实现的过滤器代码中，我们通过继承`ZuulFilter`抽象类并重写了下面的四个方法来实现自定义的过滤器。这四个方法分别定义了：

- `filterType`：过滤器的类型，它决定过滤器在请求的哪个生命周期中执行。这里定义为`pre`，代表会在请求被路由之前执行。
- `filterOrder`：过滤器的执行顺序。当请求在一个阶段中存在多个过滤器时，需要根据该方法返回的值来依次执行。

- shouldFilter：判断该过滤器是否需要被执行。这里我们直接返回了true，因此该过滤器对所有请求都会生效。实际运用中我们可以利用该函数来指定过滤器的有效范围。
- run：过滤器的具体逻辑。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，它并不会直接生效，我们还需要为其创建具体的Bean才能启动该过滤器，在应用主类中增加如下内容：

```
@EnableZuulProxy
@SpringCloudApplication
public class ApiGatewayApplication {
    public static void main(String[] args) {
        new
SpringApplicationBuilder(ApiGatewayApplication.class).web(true).run(args);
    }

    @Bean
    public AccessFilter accessFilter() {
        return new AccessFilter();
    }
}
```

在对api-gateway服务完成了上面的改造之后，我们可以重新启动它，并发起下面的请求，对上面定义的过滤器做一个验证：

- http://localhost:2340/eureka-client/dc：返回401错误
- http://localhost:2340/eureka-client/dc?accessToken=token：正确路由到eureka-client的/dc接口，返回正确结果：

Services: [eureka-consumer, eureka-client, api-gateway]---端口号：2001

到这里，对于Spring Cloud Zuul过滤器的基本功能就以介绍完毕。读者可以根据自己的需要在服务网关上定义一些与业务无关的通用逻辑实现对请求的过滤和拦截，比如：签名校验、权限校验、请求限流等功能。