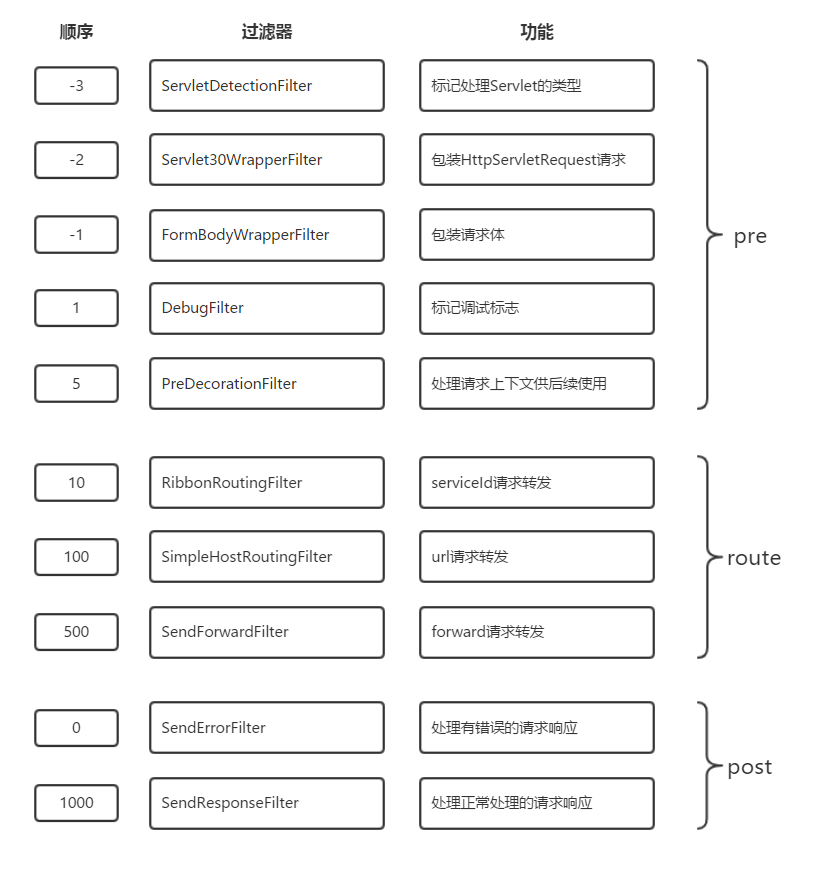
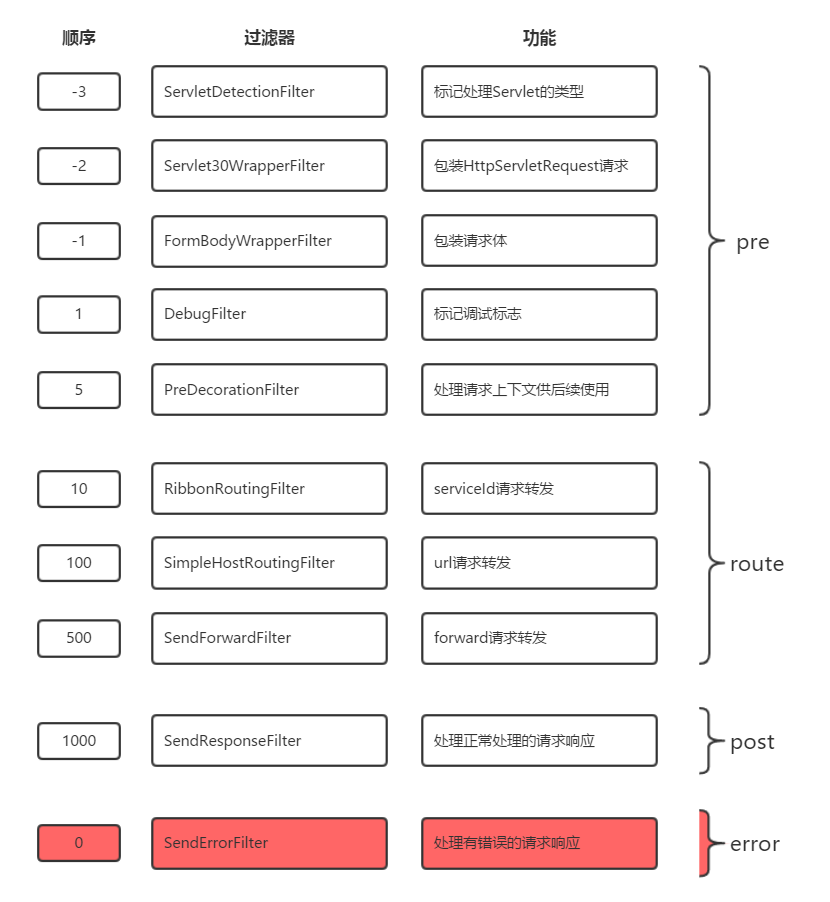
本篇作为《Spring Cloud微服务实战》一书关于Spring Cloud Zuul网关在Dalston版本对异常处理的补充。没有看过本书的读书也不要紧，可以先阅读我之前的两篇博文：[《Spring Cloud实战小贴士：Zuul统一异常处理（一）》](http://blog.didispace.com/spring-cloud-zuul-exception-1/)和[《Spring Cloud实战小贴士：Zuul统一异常处理（二）》](http://blog.didispace.com/spring-cloud-zuul-exception-2/)，这两篇文章都详细介绍和分析了Spring Cloud Zuul在过滤器设计中对异常处理的不足。同时，在这两篇文章中，也针对不足之处做了相应的解决方案。不过，这些方案都是基于Brixton版本所做的，在最新的Dalston版本中，Spring Cloud Zuul做了一些优化，所以我们不再需要做这些扩展就已经能够正确的处理异常信息了。那么，在Dalston版中，Spring Cloud Zuul中做了怎么样的修改以达到之前我们自己扩展的效果呢？

**过滤器类型的变更**

读者是否还记得我们之前分析了Spring Cloud Zuul自带的核心过滤器有哪些呢？我们先根据下图回忆一下：

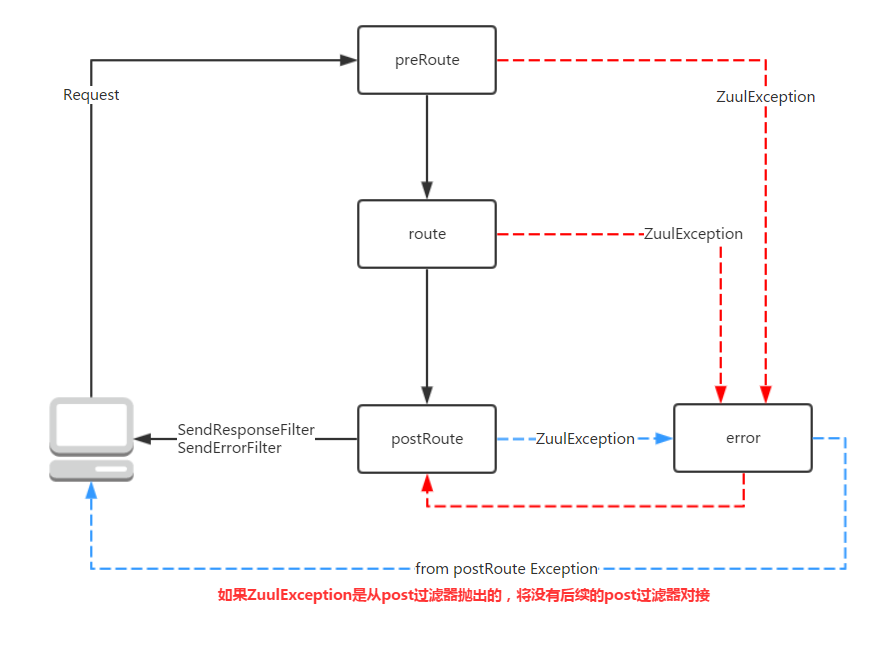
[](http://blog.didispace.com/content/images/posts/spring-cloud-zuul-exception-3-1.png)

这次主要将SendErrorFilter过滤器的类型从POST改为了ERROR，所以核心过滤器变成了如下图的结构：

[](http://blog.didispace.com/content/images/posts/spring-cloud-zuul-exception-3-2.png)

**处理逻辑的变化**

既然过滤器类型发生了变化，那么请求的处理生命周期就会有所变化。在变化之前，各阶段过滤器的流转如下图所示：

[](http://blog.didispace.com/content/images/posts/spring-cloud-zuul-exception-3-3.png)

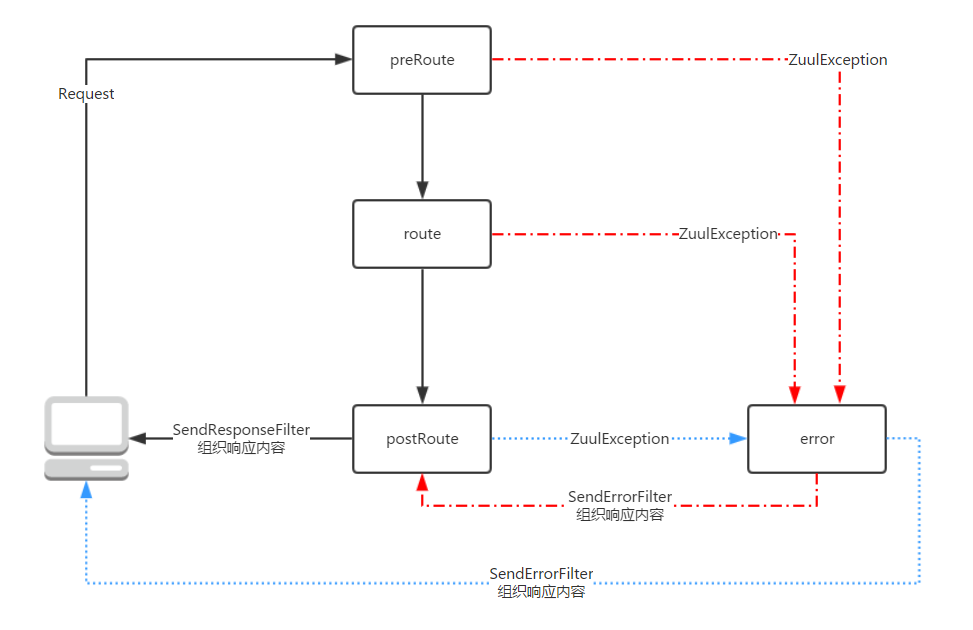
针对异常情况，在图中我们标出了不同的颜色。从pre和route阶段抛出的异常将会进入error阶段，再进入到post阶段进行返回。由于SendErrorFilter需要判断请求上下文中是否包含error.status\_code属性：有的话就用SendErrorFilter处理错误结果；没有的话就用SendResponseFilter返回正常结果，但是error.status\_code属性默认是在各个阶段过滤器中自己put进去的，这就导致，各个阶段过滤器抛出异常之后，是没有办法返回错误结果的。因此，我们扩展了一个ErrorFilter来捕获异常，然后手工的设置error.status\_code属性，让SendErrorFilter能正常运作。

通过上面你的改造，从pre和route阶段的异常都能处理了，但是post阶段抛出异常后，是不会再进入post阶段的，这使得ErrorFilter设置了设置error.status\_code属性之后，也没有过滤器去组织返回结果，所以我们通过继承SendErrorFilter在error阶段增加了一个返回错误信息的过滤器。

而这次在Dalston版本中，做了很巧妙的变动：就是上文所述的对SendErrorFilter过滤器类型的变更，这一变动使得所有阶段的异常都会被SendErrorFilter处理，直接解决的上面的第二个问题。当然只是做个变动还是不够的，为了区分SendErrorFilter和SendResponseFitler分别处理出现异常和未出现异常的情况，修改原来根据error.status\_code属性判断的逻辑，而是改为根据请求上下文中是否包含Throwable来作为基本依据，而这个对象是在过滤器出现异常之后，Zuul往请求上下文中置入的，所以可以更为准确的判断当前请求处理是否出现了异常，而不再需要我们之前扩展的ErrorFilter了。

|  |
| --- |
| public class SendErrorFilter extends ZuulFilter {   @Override  public boolean shouldFilter() {  RequestContext ctx = RequestContext.getCurrentContext();  return ctx.containsKey("error.status\_code")  && !ctx.getBoolean(SEND\_ERROR\_FILTER\_RAN, false);  }  ... }  public class SendResponseFilter extends ZuulFilter {  @Override  public boolean shouldFilter() {  RequestContext context = RequestContext.getCurrentContext();  return context.getThrowable() == null  && (!context.getZuulResponseHeaders().isEmpty()  || context.getResponseDataStream() != null  || context.getResponseBody() != null);  }  ... } |

所以，最后修改之后，整个处理逻辑变为如下图所示的流程：

[](http://blog.didispace.com/content/images/posts/spring-cloud-zuul-exception-3-4.png)

1.4 禁用过滤器

只需要在application.properties(或yml)中配置需要禁用的filter，格式为:zuul.[filter-name].[filter-type].disable=true。如:

zuul.FormBodyWrapperFilter.pre.disable=true

2. @EnableZuulServer VS. @EnableZuulProxy

Zuul为我们提供了两个主应用注解: @EnableZuulServer和@EnableZuulProxy，其中@EnableZuulProxy包含@EnableZuulServer的功能，

而且还加入了@EnableCircuitBreaker和@EnableDiscoveryClient。当我们需要运行一个没有代理功能的Zuul服务，或者有选择的开关部分代理功能时，

那么需要使用 @EnableZuulServer 替代 @EnableZuulProxy。 这时候我们可以添加任何 ZuulFilter类型实体类都会被自动加载，

这和上一篇使用@EnableZuulProxy是一样，但不会自动加载任何代理过滤器。

2.1 @EnableZuulServer默认过滤器

当我们使用@EnableZuulServer时，默认所加载的过滤器有：

2.1.1 PRE类型过滤器

ServletDetectionFilter

该过滤器是最先被执行的。其主要用来检查当前请求是通过Spring的DispatcherServlet处理运行的，还是通过ZuulServlet来处理运行的。判断结果会保存在isDispatcherServletRequest中，值类型为布尔型。

FormBodyWrapperFilter

该过滤器的目的是将符合要求的请求体包装成FormBodyRequestWrapper对象，以供后续处理使用。

DebugFilter

PRE类型过滤器。当请求参数中设置了debug参数时，该过滤器会将当前请求上下文中的RequestContext.setDebugRouting()和RequestContext.setDebugRequest()设置为true，这样后续的过滤器可以根据这两个参数信息定义一些debug信息，当生产环境出现问题时，我们就可以通过增加该参数让后台打印出debug信息，以帮助我们进行问题分析。对于请求中的debug参数的名称，我们可以通过zuul.debug.parameter进行自定义。

2.1.2 ROUTE类型过滤器

SendForwardFilter

该过滤器只对请求上下文中存在forward.to(FilterConstants.FORWARD\_TO\_KEY)参数的请求进行处理。即处理之前我们路由规则中forward的本地跳转。

2.1.3 POST类型过滤器

SendResponseFilter

该过滤器就是对代理请求所返回的响应进行封装，然后作为本次请求的相应发送回给请求者。

2.1.4 Error类型过滤器

SendErrorFilter

该过滤器就是判断当前请求上下文中是否有异常信息(RequestContext.getThrowable()不为空)，如果有则默认转发到/error页面，我们也可以通过设置error.path来自定义错误页面。

2.2 @EnableZuulProxy默认过滤器

@EnableZuulProxy则在上面的基础上增加以下过滤器：

2.2.1 PRE类型过滤器

PreDecorationFilter

该过滤器根据提供的RouteLocator确定路由到的地址，以及怎样去路由。该路由器也可为后端请求设置各种代理相关的header。

2.2.2 ROUTE类型过滤器

RibbonRoutingFilter

该过滤器会针对上下文中存在serviceId(可以通过RequestContext.getCurrentContext().get(“serviceId”)获取)的请求进行处理，使用Ribbon、Hystrix和可插拔的HTTP客户端发送请求，并将服务实例的请求结果返回。也就是之前所说的只有当我们使用serviceId配置路由规则时Ribbon和Hystrix方才生效。

SimpleHostRoutingFilter

该过滤器检测到routeHost参数(可通过RequestContext.getRouteHost()获取)设置时，就会通过Apache HttpClient向指定的URL发送请求。此时，请求不会使用Hystrix命令进行包装，所以这类请求也就没有线程隔离和断路器保护。