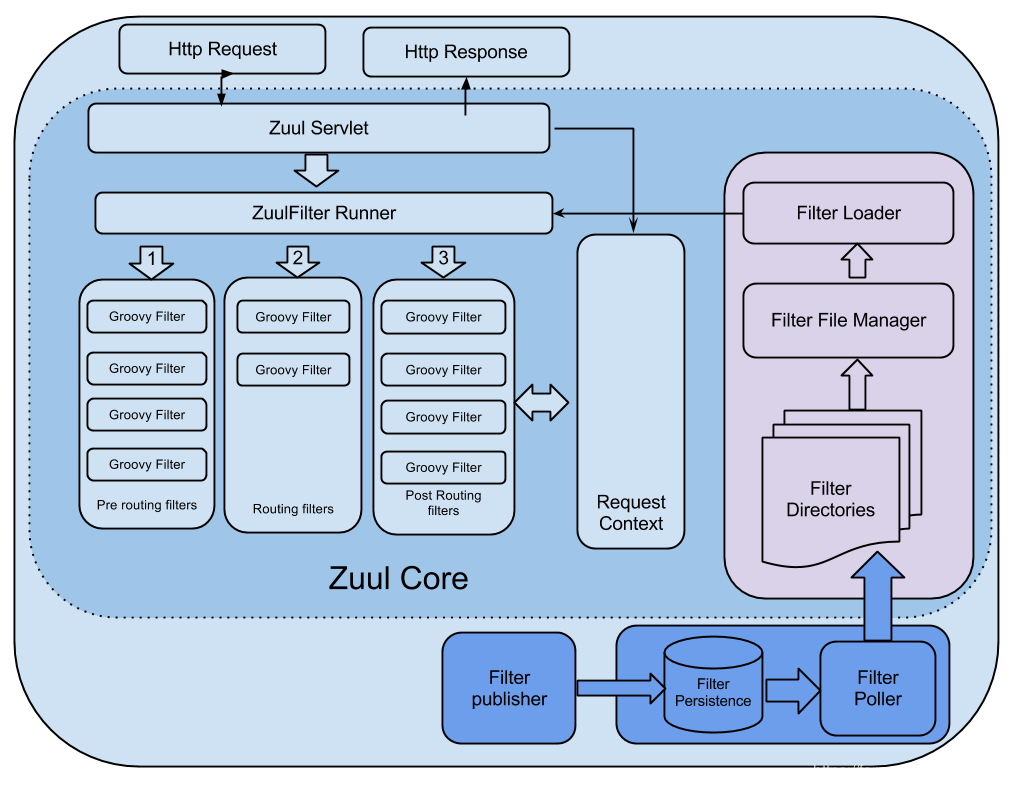
Zuul 架构图



在zuul中， 整个请求的过程是这样的＠EnableZuulProxy引用了ZuulProxyConfiguration，该类注入了DiscoveryClient、RibbonCommandFactoryConfiguration用作负载均衡相关的。注入了一些filters，比如PreDecorationFilter、RibbonRoutingFilter、SimpleHostRoutingFilter，注入了ZuulServlet，首先将请求给zuulservlet处理，zuulservlet中有一个zuulRunner对象，该对象中初始化了RequestContext：作为存储整个请求的一些数据，并被所有的zuulfilter共享。然后执行所有的过滤器，zuulservelet首先执行的Pre类型的过滤器，再执行route类型的过滤器，最后执行的是post 类型的过滤器，如果在执行这些过滤器有错误的时候则会执行error类型的过滤器。执行完这些过滤器，最终将请求的结果返回给客户端。

zuul工作原理源码分析

在之前已经讲过，如何使用zuul，其中不可缺少的一个步骤就是在程序的启动类加上＠EnableZuulProxy，该EnableZuulProxy类代码如下：

@EnableCircuitBreaker

@EnableDiscoveryClient

@Target(ElementType.TYPE)

@Retention(RetentionPolicy.RUNTIME)

@Import(ZuulProxyConfiguration.class)

public @interface EnableZuulProxy {

}

其中，引用了ZuulProxyConfiguration，跟踪ZuulProxyConfiguration，该类注入了DiscoveryClient、RibbonCommandFactoryConfiguration用作负载均衡相关的。注入了一些列的filters，比如PreDecorationFilter、RibbonRoutingFilter、SimpleHostRoutingFilter，代码如如下：

@Bean

public PreDecorationFilter preDecorationFilter(RouteLocator routeLocator, ProxyRequestHelper proxyRequestHelper) {

return new PreDecorationFilter(routeLocator, this.server.getServletPrefix(), this.zuulProperties,

proxyRequestHelper);

}

// route filters

@Bean

public RibbonRoutingFilter ribbonRoutingFilter(ProxyRequestHelper helper,

RibbonCommandFactory<?> ribbonCommandFactory) {

RibbonRoutingFilter filter = new RibbonRoutingFilter(helper, ribbonCommandFactory, this.requestCustomizers);

return filter;

}

@Bean

public SimpleHostRoutingFilter simpleHostRoutingFilter(ProxyRequestHelper helper, ZuulProperties zuulProperties) {

return new SimpleHostRoutingFilter(helper, zuulProperties);

}

它的父类ZuulConfiguration ，引用了一些相关的配置。在缺失zuulServlet bean的情况下注入了ZuulServlet，该类是zuul的核心类。

@Bean

@ConditionalOnMissingBean(name = "zuulServlet")

public ServletRegistrationBean zuulServlet() {

ServletRegistrationBean servlet = new ServletRegistrationBean(new ZuulServlet(),

this.zuulProperties.getServletPattern());

// The whole point of exposing this servlet is to provide a route that doesn't

// buffer requests.

servlet.addInitParameter("buffer-requests", "false");

return servlet;

}

同时也注入了其他的过滤器，比如ServletDetectionFilter、DebugFilter、Servlet30WrapperFilter，这些过滤器都是pre类型的。

@Bean

public ServletDetectionFilter servletDetectionFilter() {

return new ServletDetectionFilter();

}

@Bean

public FormBodyWrapperFilter formBodyWrapperFilter() {

return new FormBodyWrapperFilter();

}

@Bean

public DebugFilter debugFilter() {

return new DebugFilter();

}

@Bean

public Servlet30WrapperFilter servlet30WrapperFilter() {

return new Servlet30WrapperFilter();

}

它也注入了post类型的，比如 SendResponseFilter，error类型，比如 SendErrorFilter，route类型比如SendForwardFilter，代码如下：

@Bean

public SendResponseFilter sendResponseFilter() {

return new SendResponseFilter();

}

@Bean

public SendErrorFilter sendErrorFilter() {

return new SendErrorFilter();

}

@Bean

public SendForwardFilter sendForwardFilter() {

return new SendForwardFilter();

}

初始化ZuulFilterInitializer类，将所有的filter 向FilterRegistry注册。

@Configuration

protected static class ZuulFilterConfiguration {

@Autowired

private Map<String, ZuulFilter> filters;

@Bean

public ZuulFilterInitializer zuulFilterInitializer(

CounterFactory counterFactory, TracerFactory tracerFactory) {

FilterLoader filterLoader = FilterLoader.getInstance();

FilterRegistry filterRegistry = FilterRegistry.instance();

return new ZuulFilterInitializer(this.filters, counterFactory, tracerFactory, filterLoader, filterRegistry);

}

}

而FilterRegistry管理了一个ConcurrentHashMap，用作存储过滤器的，并有一些基本的CURD过滤器的方法，代码如下：

public class FilterRegistry {

private static final FilterRegistry INSTANCE = new FilterRegistry();

public static final FilterRegistry instance() {

return INSTANCE;

}

private final ConcurrentHashMap<String, ZuulFilter> filters = new ConcurrentHashMap<String, ZuulFilter>();

private FilterRegistry() {

}

public ZuulFilter remove(String key) {

return this.filters.remove(key);

}

public ZuulFilter get(String key) {

return this.filters.get(key);

}

public void put(String key, ZuulFilter filter) {

this.filters.putIfAbsent(key, filter);

}

public int size() {

return this.filters.size();

}

public Collection<ZuulFilter> getAllFilters() {

return this.filters.values();

}

}

FilterLoader类持有FilterRegistry，FilterFileManager类持有FilterLoader，所以最终是由FilterFileManager注入 filterFilterRegistry的ConcurrentHashMap的。FilterFileManager到开启了轮询机制，定时的去加载过滤器，代码如下：

void startPoller() {

poller = new Thread("GroovyFilterFileManagerPoller") {

public void run() {

while (bRunning) {

try {

sleep(pollingIntervalSeconds \* 1000);

manageFiles();

} catch (Exception e) {

e.printStackTrace();

}

}

}

};

poller.setDaemon(true);

poller.start();

}

Zuulservlet作为类似于Spring MVC中的DispatchServlet,起到了前端控制器的作用，所有的请求都由它接管。它的核心代码如下：

@Override

public void service(javax.servlet.ServletRequest servletRequest, javax.servlet.ServletResponse servletResponse) throws ServletException, IOException {

try {

init((HttpServletRequest) servletRequest, (HttpServletResponse) servletResponse);

// Marks this request as having passed through the "Zuul engine", as opposed to servlets

// explicitly bound in web.xml, for which requests will not have the same data attached

RequestContext context = RequestContext.getCurrentContext();

context.setZuulEngineRan();

try {

preRoute();

} catch (ZuulException e) {

error(e);

postRoute();

return;

}

try {

route();

} catch (ZuulException e) {

error(e);

postRoute();

return;

}

try {

postRoute();

} catch (ZuulException e) {

error(e);

return;

}

} catch (Throwable e) {

error(new ZuulException(e, 500, "UNHANDLED\_EXCEPTION\_" + e.getClass().getName()));

} finally {

RequestContext.getCurrentContext().unset();

}

}

跟踪init（），可以发现这个方法为每个请求生成了RequestContext,RequestContext继承了ConcurrentHashMap<String, Object>，在请求结束时销毁掉该RequestContext，RequestContext的生命周期为请求到zuulServlet开始处理，直到请求结束返回结果。

RequestContext类在存储了很多重要的信息，包括HttpServletRequest、HttpServletRespons、ResponseDataStream、ResponseStatusCode等。 RequestContext对象在处理请求的过程中，一直存在，所以这个对象为所有Filter共享。

从ZuulServlet的service（）方法可知，它是先处理pre()类型的处理器，然后在处理route()类型的处理器，最后再处理post类型的处理器。

首先来看一看pre()的处理过程，它会进入到ZuulRunner,该类的作用是将请求的HttpServletRequest、HttpServletRespons放在RequestContext类中，并包装了一个FilterProcessor，代码如下：

public void init(HttpServletRequest servletRequest, HttpServletResponse servletResponse) {

RequestContext ctx = RequestContext.getCurrentContext();

if (bufferRequests) {

ctx.setRequest(new HttpServletRequestWrapper(servletRequest));

} else {

ctx.setRequest(servletRequest);

}

ctx.setResponse(new HttpServletResponseWrapper(servletResponse));

}

public void preRoute() throws ZuulException {

FilterProcessor.getInstance().preRoute();

}

而FilterProcessor类为调用filters的类，比如调用pre类型所有的过滤器：

public void preRoute() throws ZuulException {

try {

runFilters("pre");

} catch (ZuulException e) {

throw e;

} catch (Throwable e) {

throw new ZuulException(e, 500, "UNCAUGHT\_EXCEPTION\_IN\_PRE\_FILTER\_" + e.getClass().getName());

}

}

跟踪runFilters（）方法，可以发现，它最终调用了FilterLoader的getFiltersByType(sType)方法来获取同一类的过滤器，然后用for循环遍历所有的ZuulFilter，执行了 processZuulFilter（）方法，跟踪该方法可以发现最终是执行了ZuulFilter的方法，最终返回了该方法返回的Object对象。

public Object runFilters(String sType) throws Throwable {

if (RequestContext.getCurrentContext().debugRouting()) {

Debug.addRoutingDebug("Invoking {" + sType + "} type filters");

}

boolean bResult = false;

List<ZuulFilter> list = FilterLoader.getInstance().getFiltersByType(sType);

if (list != null) {

for (int i = 0; i < list.size(); i++) {

ZuulFilter zuulFilter = list.get(i);

Object result = processZuulFilter(zuulFilter);

if (result != null && result instanceof Boolean) {

bResult |= ((Boolean) result);

}

}

}

return bResult;

}

route、post类型的过滤器的执行过程和pre执行过程类似。

Zuul默认过滤器

默认的核心过滤器一览表

Zuul默认注入的过滤器，它们的执行顺序在FilterConstants类，我们可以先定位在这个类，然后再看这个类的过滤器的执行顺序以及相关的注释，可以很轻松定位到相关的过滤器，也可以直接打开

spring-cloud-netflix-core.jar的 zuul.filters包，可以看到一些列的filter，现在我以表格的形式，列出默认注入的filter.

过滤器 order 描述 类型

ServletDetectionFilter -3 检测请求是用 DispatcherServlet还是 ZuulServlet pre

Servlet30WrapperFilter -2 在Servlet 3.0 下，包装 requests pre

FormBodyWrapperFilter -1 解析表单数据 pre

SendErrorFilter 0 如果中途出现错误 error

DebugFilter 1 设置请求过程是否开启debug pre

PreDecorationFilter 5 根据uri决定调用哪一个route过滤器 pre

RibbonRoutingFilter 10 如果写配置的时候用ServiceId则用这个route过滤器，该过滤器可以用Ribbon 做负载均衡，用hystrix做熔断 route

SimpleHostRoutingFilter 100 如果写配置的时候用url则用这个route过滤 route

SendForwardFilter 500 用RequestDispatcher请求转发 route

SendResponseFilter 1000 用RequestDispatcher请求转发 post

过滤器的order值越小，就越先执行，并且在执行过滤器的过程中，它们共享了一个RequestContext对象，该对象的生命周期贯穿于请求，可以看出优先执行了pre类型的过滤器，并将执行后的结果放在RequestContext中，供后续的filter使用，比如在执行PreDecorationFilter的时候，决定使用哪一个route，它的结果的是放在RequestContext对象中，后续会执行所有的route的过滤器，如果不满足条件就不执行该过滤器的run方法。最终达到了就执行一个route过滤器的run()方法。

而error类型的过滤器，是在程序发生异常的时候执行的。

post类型的过滤，在默认的情况下，只注入了SendResponseFilter，该类型的过滤器是将最终的请求结果以流的形式输出给客户单。

现在来看SimpleHostRoutingFilter是如何工作?

进入到SimpleHostRoutingFilter类的方法的run()方法，核心代码如下：

@Override

public Object run() {

RequestContext context = RequestContext.getCurrentContext();

//省略代码

String uri = this.helper.buildZuulRequestURI(request);

this.helper.addIgnoredHeaders();

try {

CloseableHttpResponse response = forward(this.httpClient, verb, uri, request,

headers, params, requestEntity);

setResponse(response);

}

catch (Exception ex) {

throw new ZuulRuntimeException(ex);

}

return null;

}

查阅这个类的全部代码可知，该类创建了一个HttpClient作为请求类，并重构了url,请求到了具体的服务，得到的一个CloseableHttpResponse对象，并将CloseableHttpResponse对象的保存到RequestContext对象中。并调用了ProxyRequestHelper的setResponse方法，将请求状态码，流等信息保存在RequestContext对象中。

private void setResponse(HttpResponse response) throws IOException {

RequestContext.getCurrentContext().set("zuulResponse", response);

this.helper.setResponse(response.getStatusLine().getStatusCode(),

response.getEntity() == null ? null : response.getEntity().getContent(),

revertHeaders(response.getAllHeaders()));

}

现在来看SendResponseFilter是如何工作?

这个过滤器的order为1000,在默认且正常的情况下，是最后一个执行的过滤器，该过滤器是最终将得到的数据返回给客户端的请求。

在它的run()方法里，有两个方法：addResponseHeaders()和writeResponse()，即添加响应头和写入响应数据流。

public Object run() {

try {

addResponseHeaders();

writeResponse();

}

catch (Exception ex) {

ReflectionUtils.rethrowRuntimeException(ex);

}

return null;

}

其中writeResponse（）方法是通过从RequestContext中获取ResponseBody获或者ResponseDataStream来写入到HttpServletResponse中的，但是在默认的情况下ResponseBody为null，而ResponseDataStream在route类型过滤器中已经设置进去了。具体代码如下：

private void writeResponse() throws Exception {

RequestContext context = RequestContext.getCurrentContext();

HttpServletResponse servletResponse = context.getResponse();

//代码省略

OutputStream outStream = servletResponse.getOutputStream();

InputStream is = null;

try {

if (RequestContext.getCurrentContext().getResponseBody() != null) {

String body = RequestContext.getCurrentContext().getResponseBody();

writeResponse(

new ByteArrayInputStream(

body.getBytes(servletResponse.getCharacterEncoding())),

outStream);

return;

}

//代码省略

is = context.getResponseDataStream();

InputStream inputStream = is;

//代码省略

writeResponse(inputStream, outStream);

//代码省略

}

}

..//代码省略

}

如何在zuul上做日志处理

由于zuul作为api网关，所有的请求都经过这里，所以在网关上，可以做请求相关的日志处理。

我的需求是这样的，需要记录请求的 url,ip地址，参数，请求发生的时间，整个请求的耗时，请求的响应状态，甚至请求响应的结果等。

很显然，需要实现这样的一个功能，需要写一个ZuulFliter，它应该是在请求发送给客户端之前做处理，并且在route过滤器路由之后，在默认的情况下，这个过滤器的order应该为500-1000之间。那么如何获取这些我需要的日志信息呢？找RequestContext,在请求的生命周期里这个对象里，存储了整个请求的所有信息。

现在编码，在代码的注释中，做了详细的说明，代码如下：

@Component

public class LoggerFilter extends ZuulFilter {

@Override

public String filterType() {

return FilterConstants.POST\_TYPE;

}

@Override

public int filterOrder() {

return FilterConstants.SEND\_RESPONSE\_FILTER\_ORDER - 1;

}

@Override

public boolean shouldFilter() {

return true;

}

@Override

public Object run() {

RequestContext context = RequestContext.getCurrentContext();

HttpServletRequest request = context.getRequest();

String method = request.getMethod();//氢气的类型，post get ..

Map<String, String> params = HttpUtils.getParams(request);

String paramsStr = params.toString();//请求的参数

long statrtTime = (long) context.get("startTime");//请求的开始时间

Throwable throwable = context.getThrowable();//请求的异常，如果有的话

request.getRequestURI()；//请求的uri

HttpUtils.getIpAddress(request);//请求的iP地址

context.getResponseStatusCode();//请求的状态

long duration=System.currentTimeMillis() - statrtTime);//请求耗时

return null;

}

}

现在读者也许有疑问，如何得到的statrtTime，即请求开始的时间，其实这需要另外一个过滤器，在网络请求route之前(大部分耗时都在route这一步)，在过滤器中，在RequestContext存储一个时间即可，另写一个过滤器，代码如下：

@Component

public class AccessFilter extends ZuulFilter {

@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();

ctx.set("startTime",System.currentTimeMillis());

return null;

}

}

可能还有这样的需求，我需要将响应结果，也要存储在log中，在之前已经分析了，在route结束后，将从具体服务获取的响应流存储在RequestContext中，在SendResponseFilter过滤器写入在HttpServletResponse中，最终返回给客户端。那么我只需要在SendResponseFilter写入响应流之前把响应流写入到 log日志中即可，那么会引发另外一个问题，因为响应流写入到 log后，RequestContext就没有响应流了，在SendResponseFilter就没有流输入到HttpServletResponse中，导致客户端没有任何的返回数据，那么解决的办法是这样的：

InputStream inputStream =RequestContext.getCurrentContext().getResponseDataStream();

InputStream newInputStream= copy(inputStream);

transerferTolog(inputStream);

RequestContext.getCurrentContext().setResponseDataStream(newInputStream);

从RequestContext获取到流之后，首先将流 copy一份，将流转化下字符串，存在日志中，再set到RequestContext中，

这样SendResponseFilter就可以将响应返回给客户端。这样的做法有点影响性能，如果不是字符流，可能需要做更多的处理工作。