### 响应式编程中的Flux和Mono 的理解

#### 1.1 Publisher

由于响应流的特点，我们不能再返回一个简单的**POJO**对象来表示结果了。必须返回一个类似**Java**中的Future的概念，在有结果可用时通知消费者进行消费响应。

#### 2.1 Flux

是一个发出(emit)0-N个元素组成的异步序列的Publisher<T>,可以被onComplete信号或者onError信号所终止 ，在有结果可用时通知消费者进行消费响应

**传统数据处理**

public List<ClientUser> allUsers() {

return Arrays.asList(new ClientUser("felord.cn", "reactive"),

new ClientUser("Felordcn", "Reactor"));

}

**流式数据处理**

public Stream<ClientUser> allUsers() {

return Stream.of(new ClientUser("felord.cn", "reactive"),

new ClientUser("Felordcn", "Reactor"));

}

**反应式数据处理**

public Flux<ClientUser> allUsers(){

return Flux.just(new ClientUser("felord.cn", "reactive"),

new ClientUser("Felordcn", "Reactor"));

}

#### 3 .1 Mono<T>

Mono 是一个发出(emit)0-1个元素的Publisher<T>,可以被onComplete信号或者onError信号所终止。

**反应式数据处理**

public Mono<ClientUser> currentUser () {

return isAuthenticated ? Mono.just(new ClientUser("felord.cn", "reactive"))

: Mono.empty();

}

和Optional有点类似的机制，当然Mono不是为了解决NPE问题的，它是为了处理响应流中单个值（也可能是Void）而存在的

总结：Fulx和Mono不太好理解，需要结合代码及调试深入去实践

参考链接：

<https://developer.ibm.com/zh/languages/java/articles/j-cn-with-reactor-response-encode/> （详细使用案例）

<https://juejin.cn/post/6844903858599428110>

<https://segmentfault.com/a/1190000024499748>

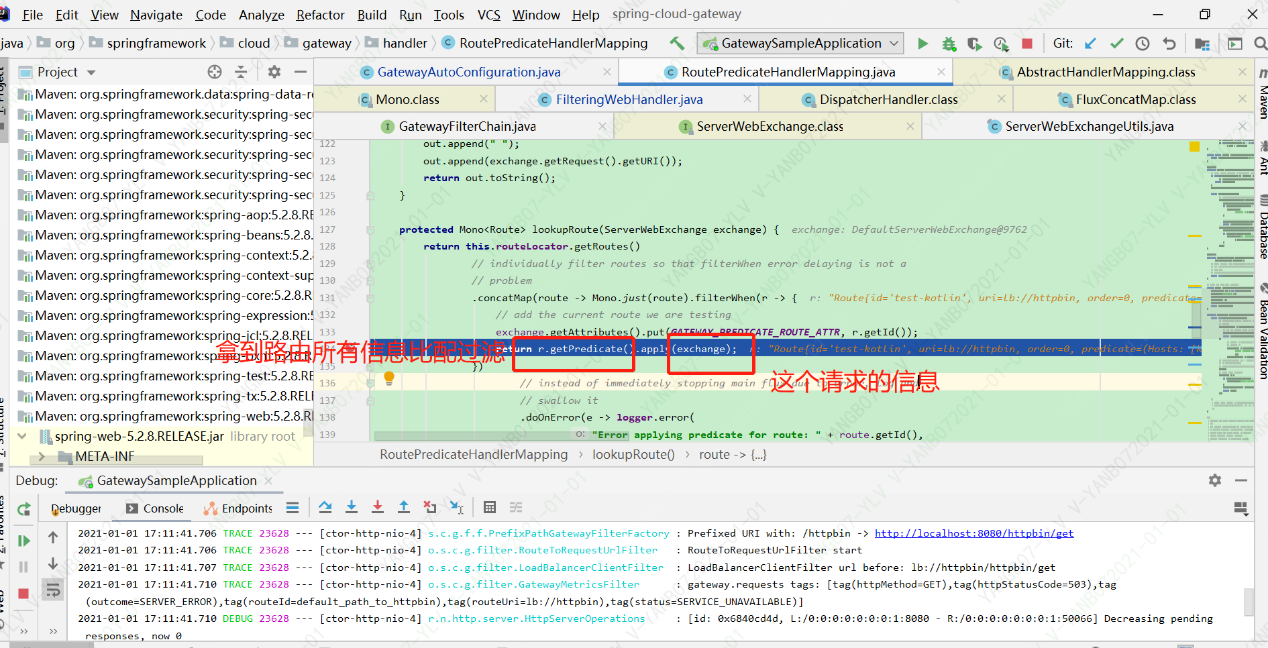
### 2 gateway是如何处理一个请求

例：post请求[http://localhost:9527/order/gateway](http://localhost:9527/order/gateway" \t "_blank) ，最后调用[http://10.201.35.189:8090/order/gateway](http://10.201.35.189:8090/order/gateway" \t "_blank)

#### 路由比配：

打断点到RoutePredicateHandlerMapping的lookupRoute，

循环每个路由，看看predicate是否匹配，一直到找到匹配的路由，这里是默认的default\_path\_to\_httpbin



debug的调用栈，目前找到这里熟悉的类DispatcherHandler，依次找到FilteringWebHandler，handle方法的大致逻辑

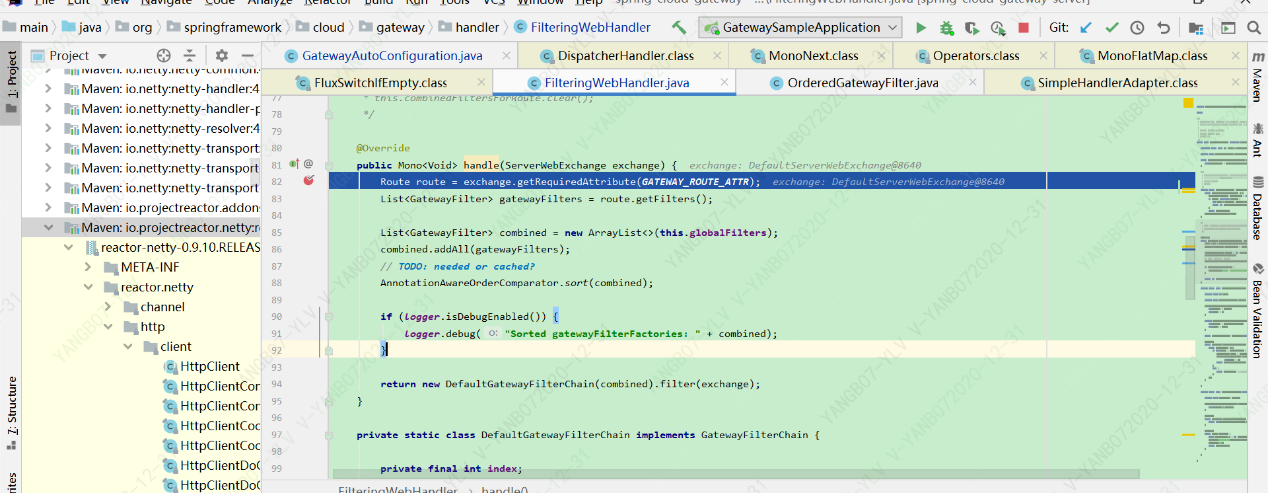
：从 GATEWAY\_ROUTE\_ATTR 获得 请求对应的 Route 。

：获得 GatewayFilter 数组，包含 route.filters 和 globalFilters 。

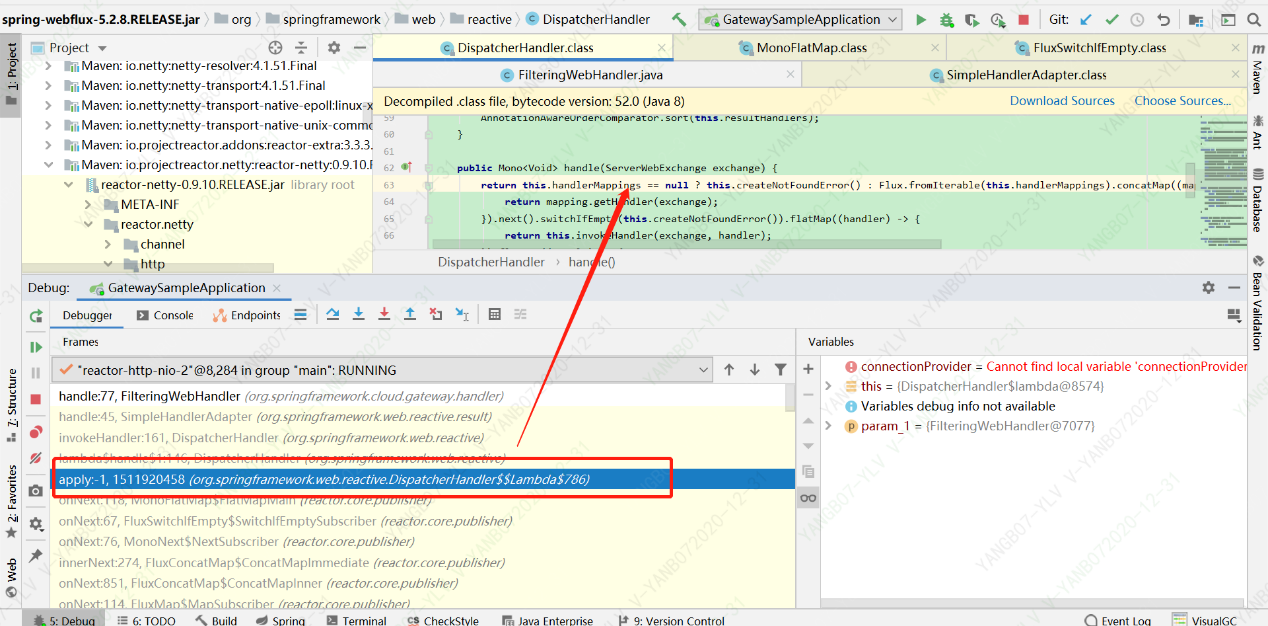
：排序获得的 GatewayFilter 数组。

：使用获得的 GatewayFilter 数组创建 DefaultGatewayFilterChain ，过滤处理请求。

下图：FilteringWebHandler 的handler方法上打上断点，可以看到 从route拿到两个filter，合并到之前的10个filter，一起来处理过滤 exchange



再走到下图的代码

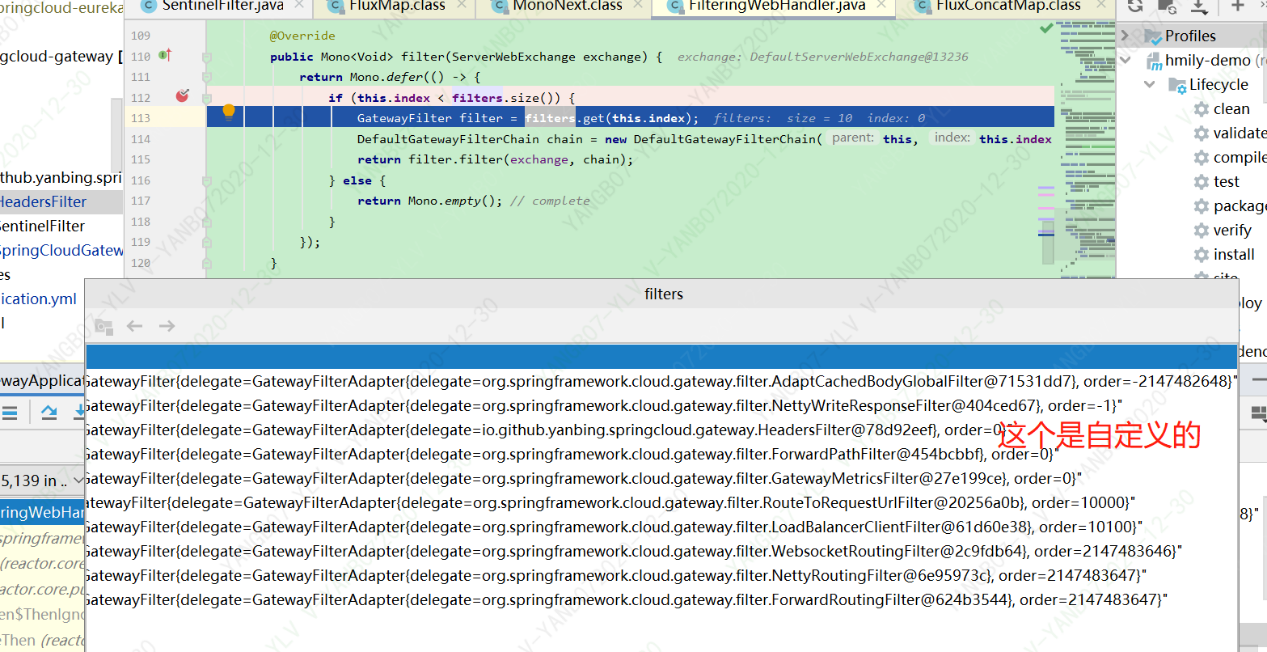


跟踪找这：

#### FilteringWebHandler

核心处理方法 ，使用调用链机制，

自定义的HeadersFilter 的order=0



NettyRoutingFilter（GlobalFilter的实现类）的方法filter，

filtered.forEach(httpHeaders::set); headers信息设置

//核心调用方法

Mono<HttpClientResponse> responseMono = this.httpClient.request(method, url, req -> {

final HttpClientRequest proxyRequest = req.options(NettyPipeline.SendOptions::flushOnEach)

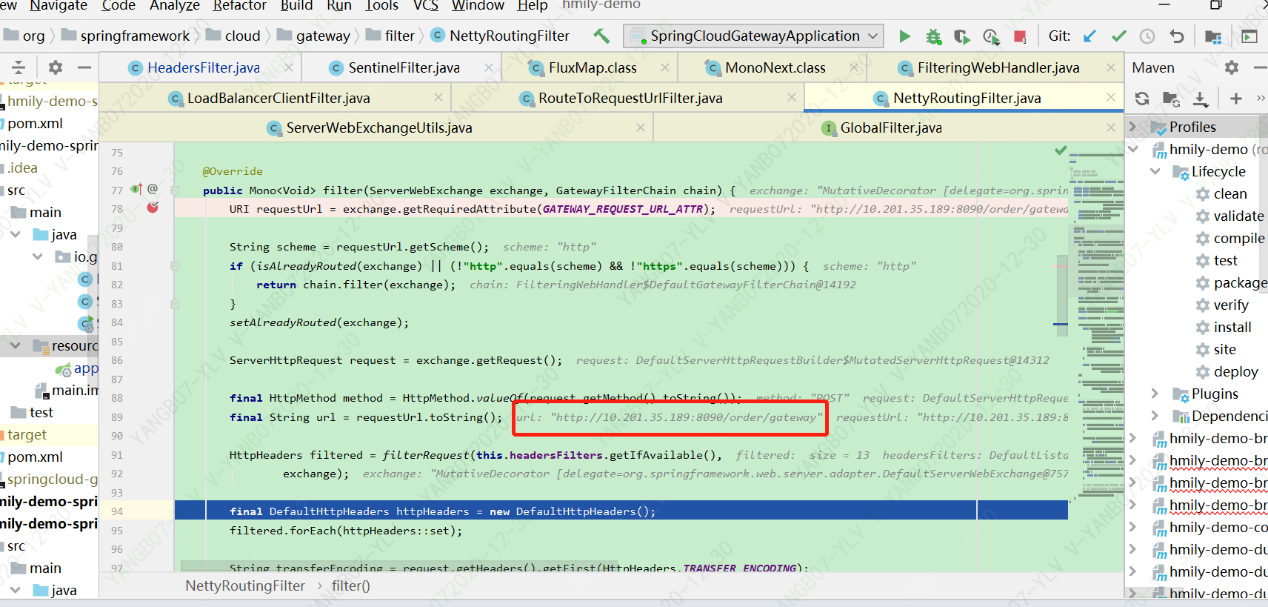
.headers(httpHeaders)

.chunkedTransfer(chunkedTransfer)

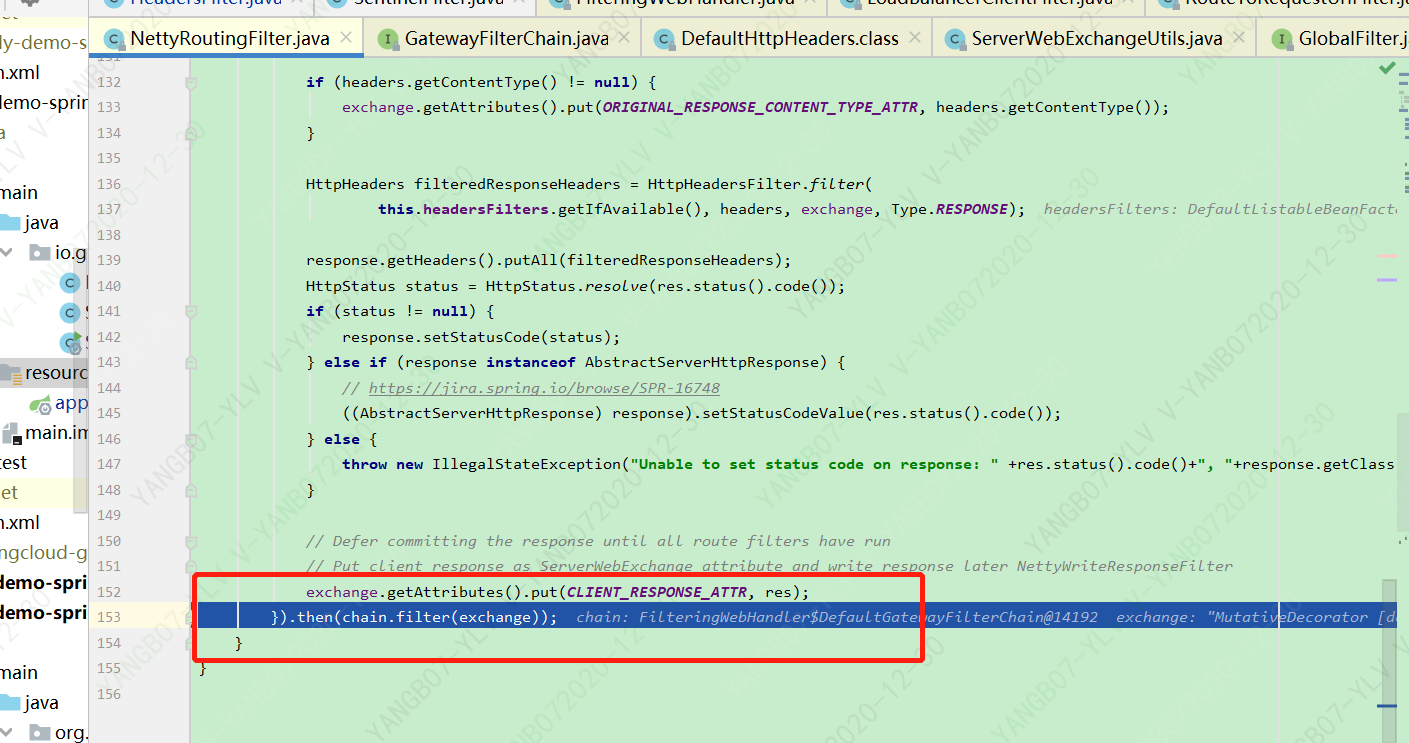
.failOnServerError(false)

.failOnClientError(false)

从bug的看到url是需要实际调用的服务地址，

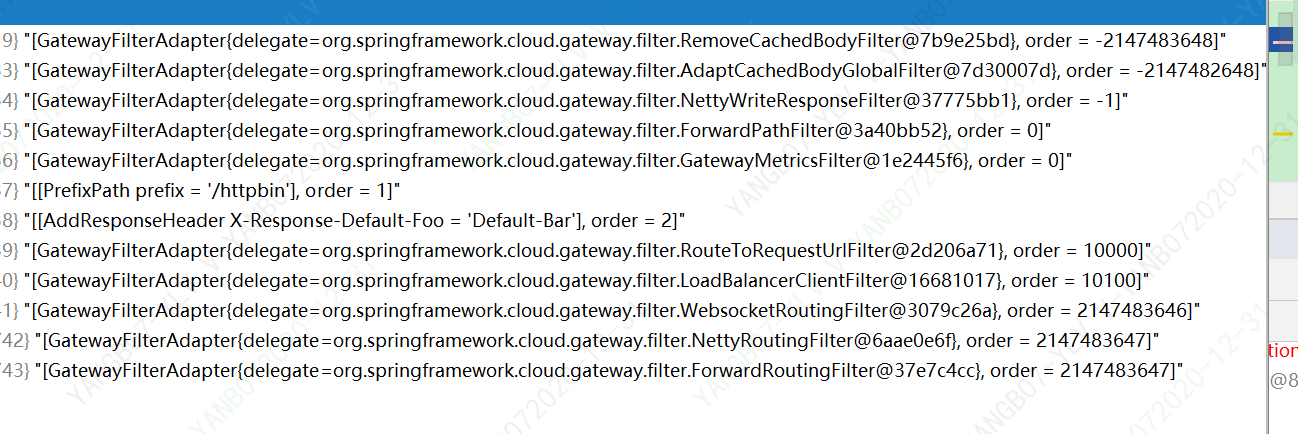


接着处理后面的filter，NettyRoutingFilter 用httpclient连接池对数据的转发



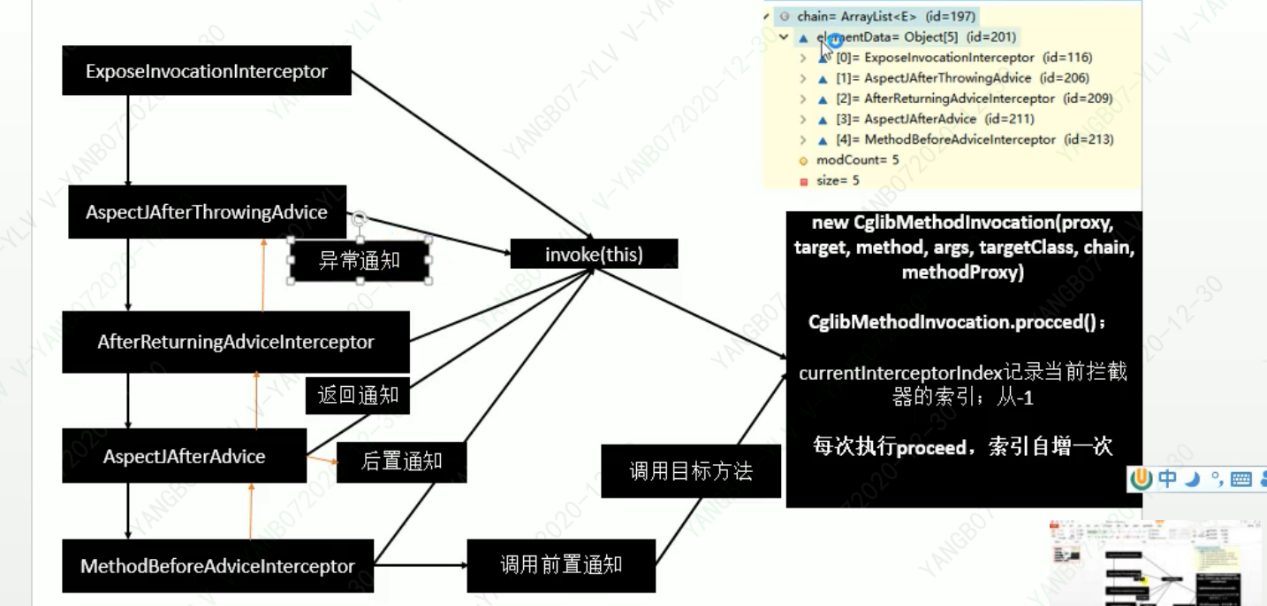
总体逻辑还需要在整理，有很多不明白的

1：具体怎么路由细节还不清楚

2：fiters一共12个，按照order从小到大一次执行，具体每个filter做什么还没去细看，只能通过类名猜想假设

### 3 链式调用机制关联学习AOP的拦截器链调用：

链式获取每一个拦截器，拦截器执行invoke方法，每一个拦截器等待下一个拦截器执行完成  返回以后再来执行，拦截器链的机制，保证通知方法与目标方法的执行顺序，运行顺序遵循“后进先出”的原则



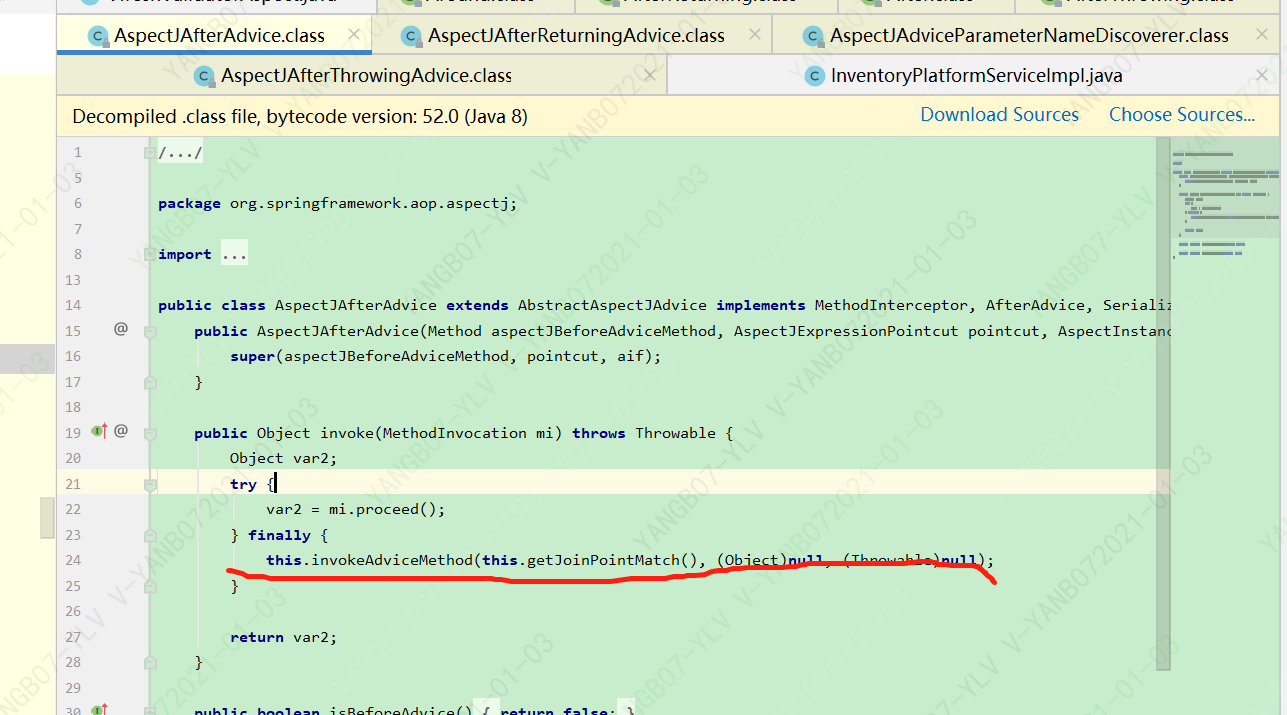
例：AspectJAfterThrowingAdvice 对应@AfterThrowing注解，如下图：在目标方法出现异常以后，才会走到invokeAdviceMethod方法，

思考：所以使用@AfterThrowing时，想要注解生效，自己的业务代码不要吃掉异常，异常必须抛出来



* AspectJAfterThrowingAdvice 对应后置通知(@After)

在目标方法运行结束之后运行（无论方法正常结束还是异常结束）



* mi.proceed() 这个方法源码如下图

解析：当前index值和增强器的数量相同执行真实目标方法，不同则执行增强器的方法，同时index+1，此处猜想index的初始值是：0

