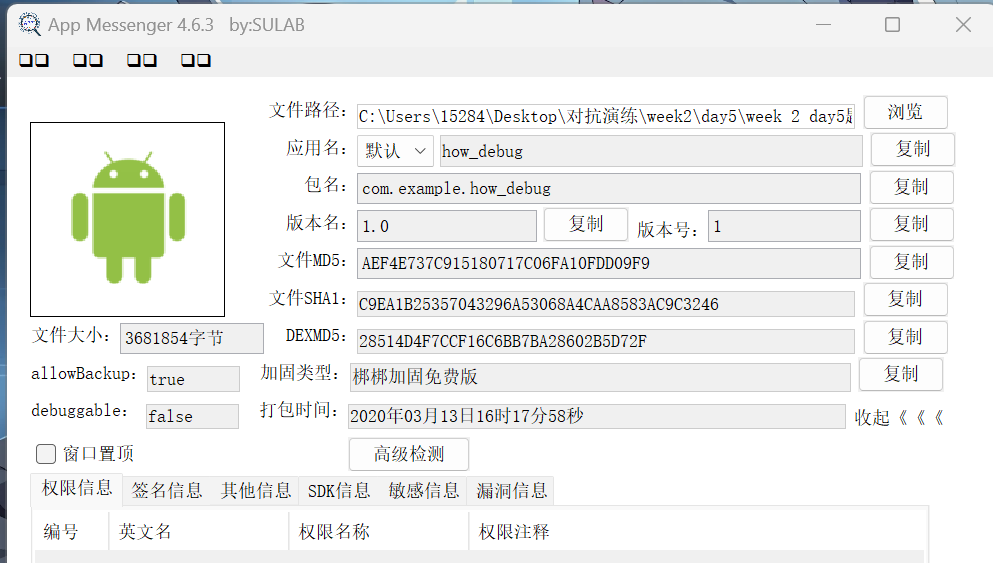
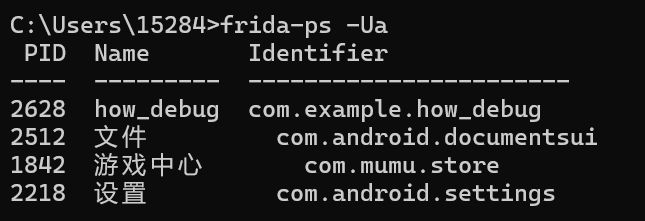
# 1.软件加固识别

首先将apk包放到app messenger中查看基本信息，可以发现该apk包使用了梆梆加固进行加壳

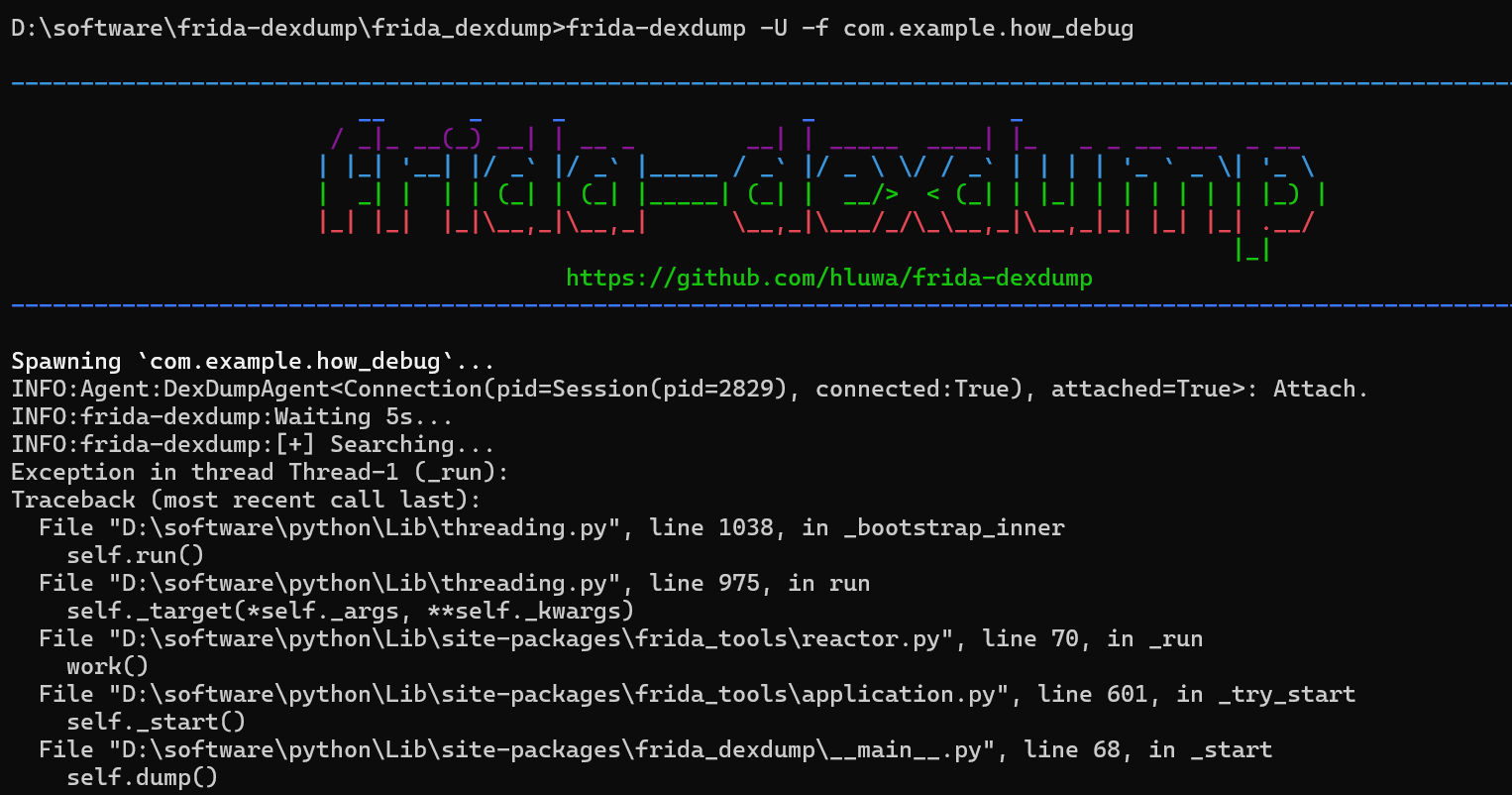


# 2.软件脱壳

在这里我们尝试使用frida-dump工具进行脱壳，首先查看对应程序的包名如下



进入到工具目录下进行脱壳但有报错



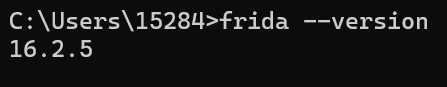
同时发现应用程序会直接闪退，由此断定是使用的模拟器的android版本有问题，更改MuMu模拟器机型，将机型设置为尽可能老的型号来确保android版本更老



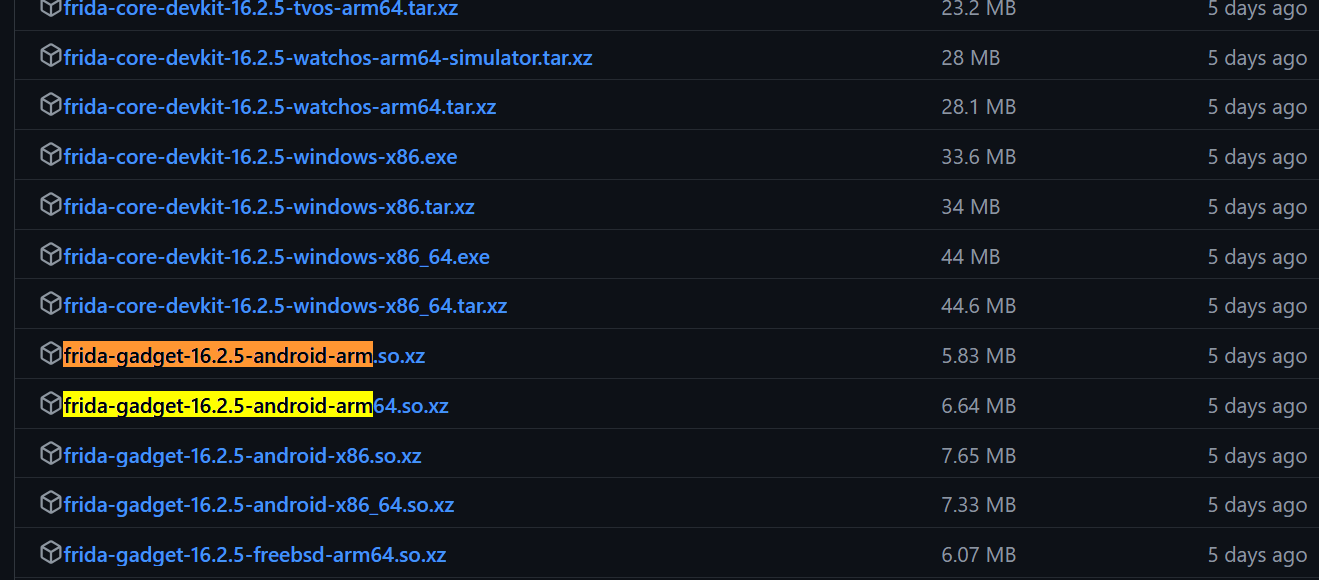
重新执行脱壳工具产生如下报错



在网络上查找原因发现可能是frida版本问题，本人使用的版本为16.2.5



根据[frida报错need Gadget....(有效解决方法)\_failed to spawn: need gadget to attach on jailed a-CSDN博客](https://blog.csdn.net/m0_65663088/article/details/135994176)的提示下载文件frida-gadget-16.2.5-android-arm64.so.xz

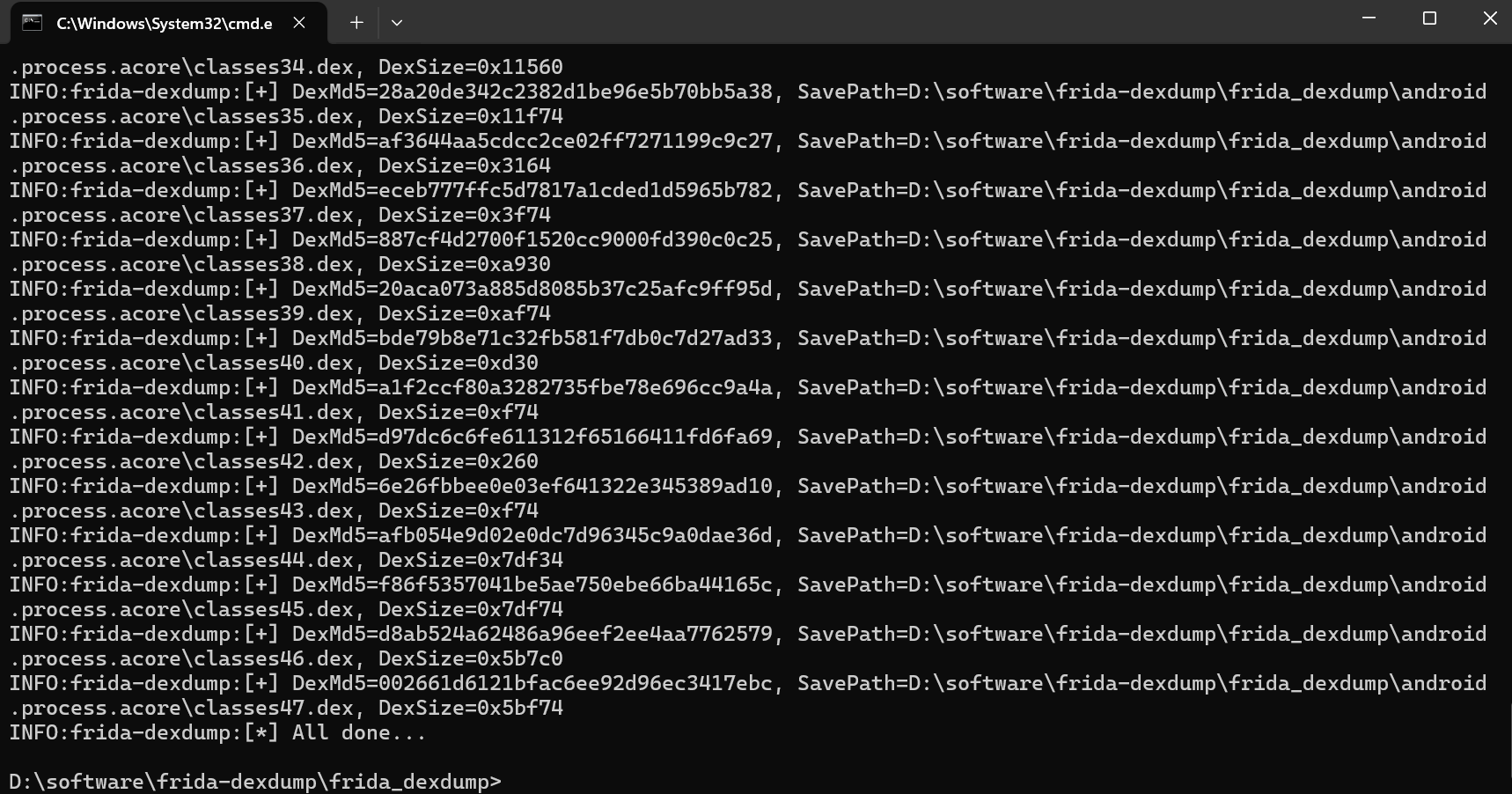


需要将下载好的文件解压之后放置到C:\Users\XXX\AppData\Local\Microsoft\Windows\INetCache\frida\目录下之后文件重命名为gadget-android-arm64.so

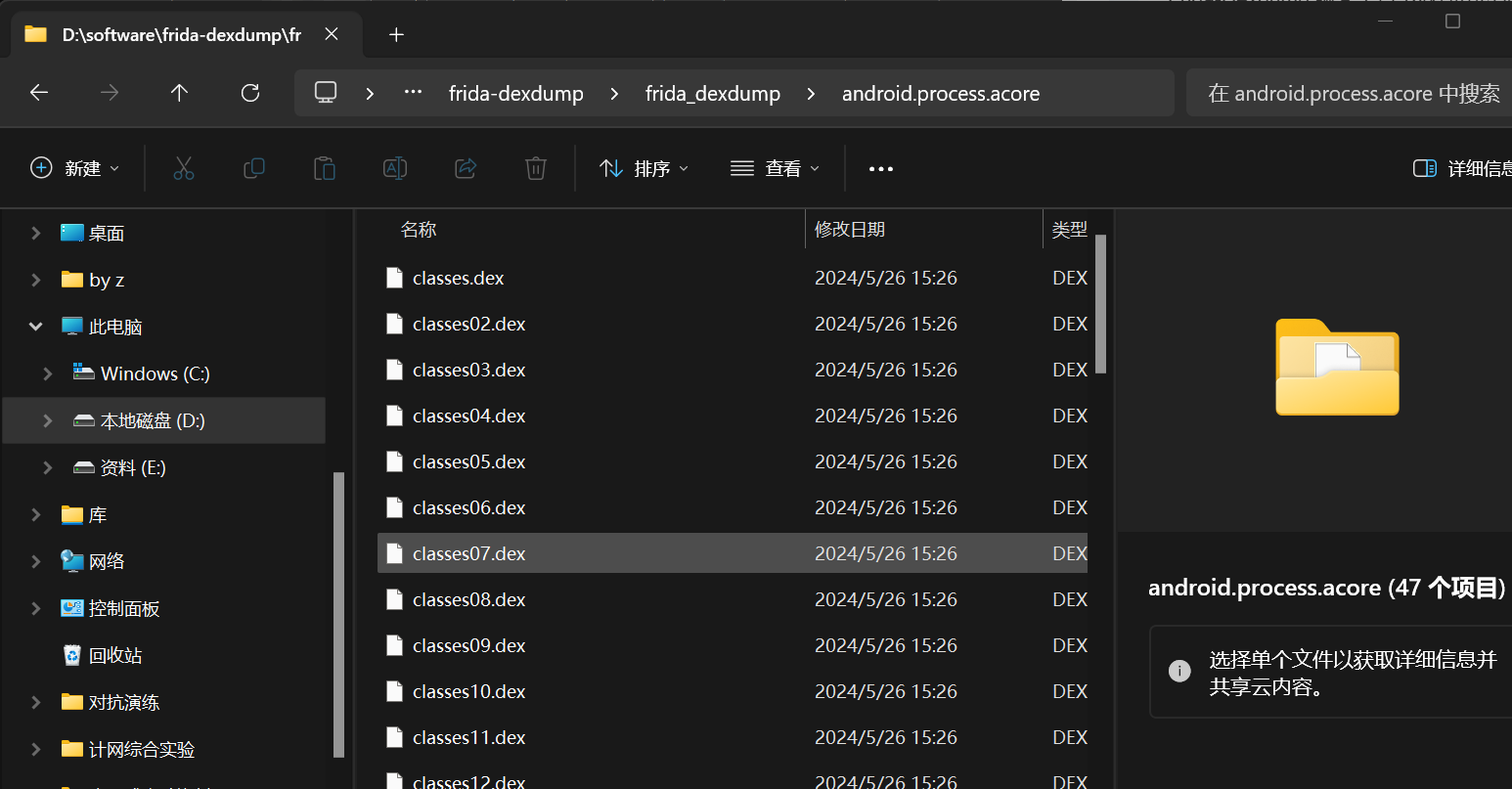
这里需要注意的是INetCache目录在文件夹下不可见，需要在终端中进入到INetCache\下创建frida目录，将文件复制到该目录如下所示



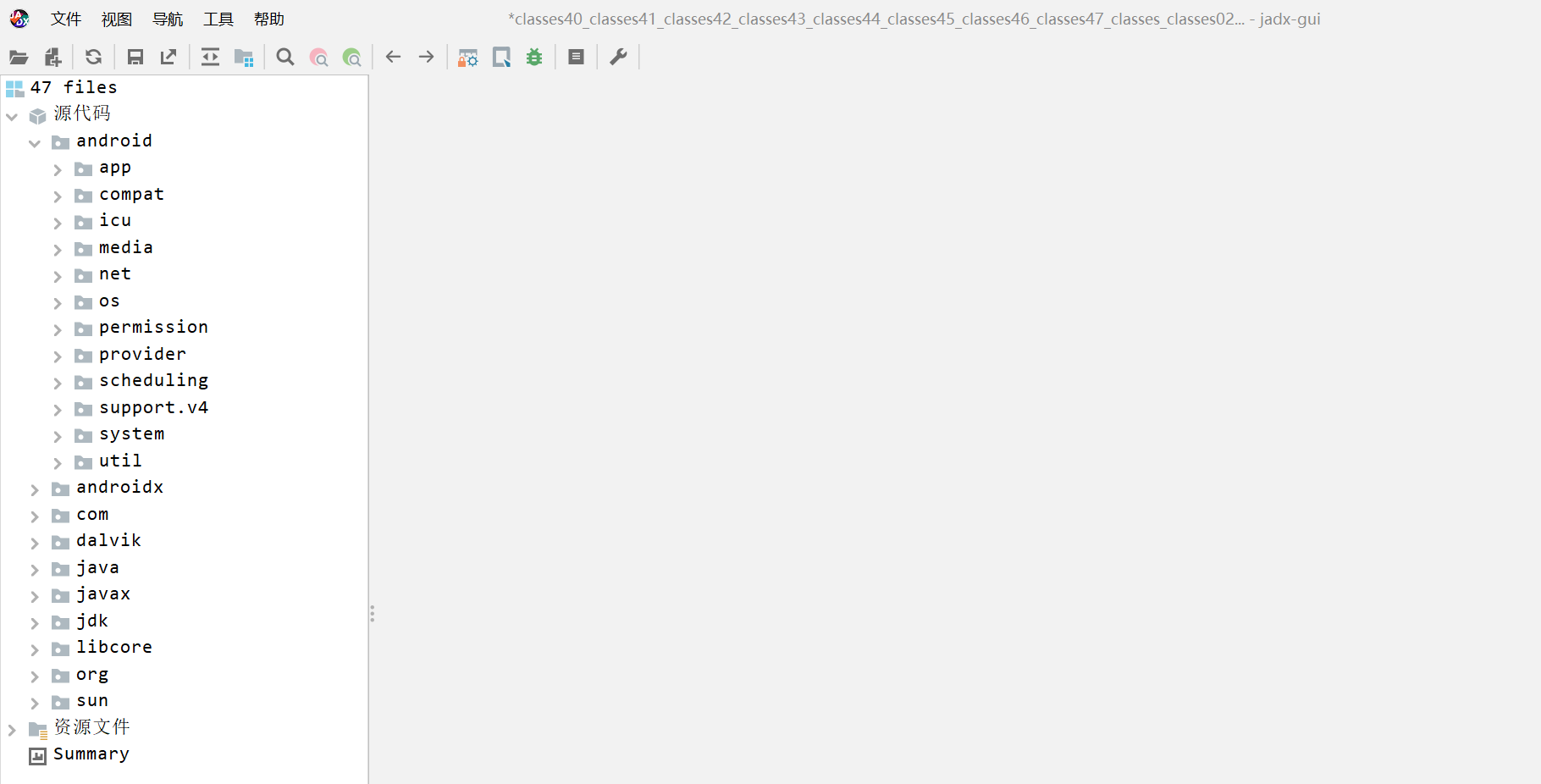
重新执行指令，在这里因为android模拟器版本问题可能会出现多次程序闪退问题，多次重新执行即可



进入到对应的文件夹可以发现生成的dex对应文件



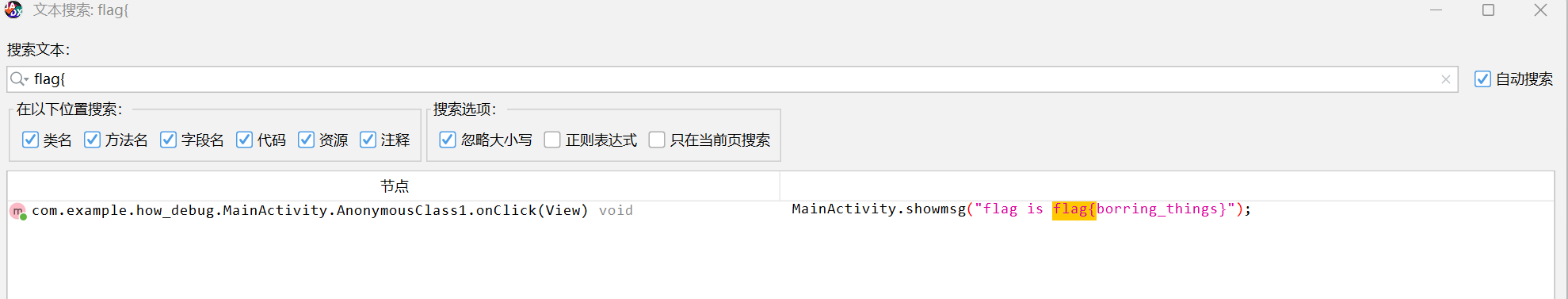
将生成的文件全部拖入jadx中进行反编译，成功获取源码实现脱壳



# 3.分析dex文件格式

# 4.查找flag

首先搜索flag{字段



定位到该字段如下



flag保存在一个onClick函数中，分析该函数的功能

public void onClick(View view) {: 这是一个公有方法，用于处理点击事件，它接受一个View类型的参数view，表示被点击的视图对象。

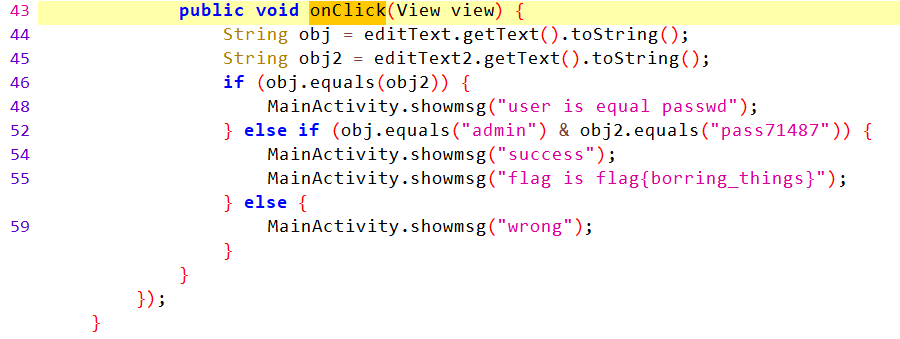
String obj = editText.getText().toString();: 这一行代码从名为editText的文本框中获取用户输入的文本，并将其转换为字符串类型，存储在变量obj中。

String obj2 = editText2.getText().toString();: 同样地，这一行代码从名为editText2的另一个文本框中获取用户输入的文本，并将其转换为字符串类型，存储在变量obj2中。

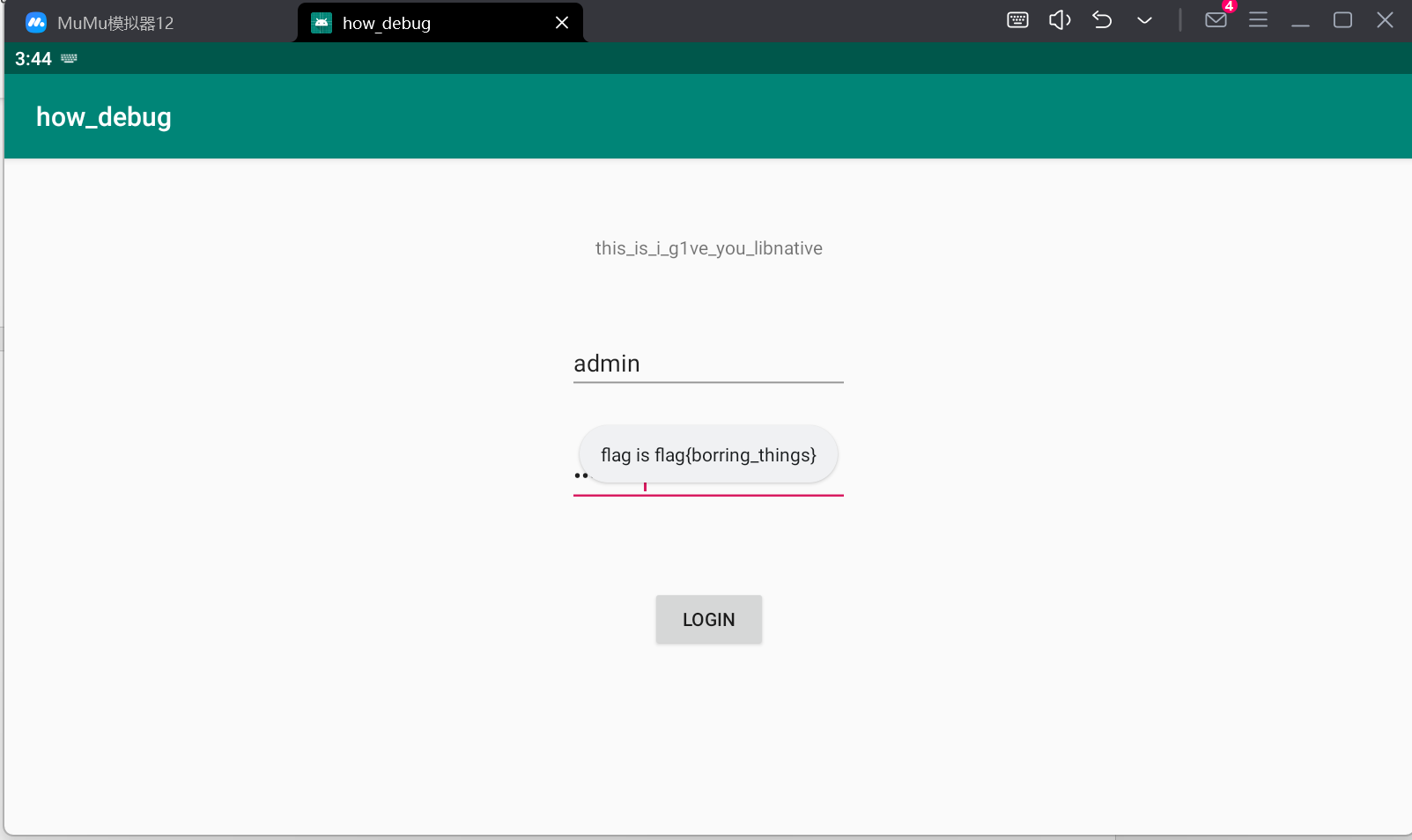
if (obj.equals(obj2)) { MainActivity.showmsg("user is equal passwd"); }: 这是一个条件判断语句，如果obj和obj2的值相等，则调用MainActivity类的showmsg方法显示消息"用户和密码相同"。

else if (obj.equals("admin") & obj2.equals("pass71487")) { MainActivity.showmsg("success"); MainActivity.showmsg("flag is flag{borring\_things}"); }: 这是另一个条件判断语句，如果obj等于"admin"且obj2等于"pass71487"，则调用MainActivity类的showmsg方法分别显示"成功"和"flag is flag{borring\_things}"。

else { MainActivity.showmsg("wrong"); }: 如果以上两个条件都不满足，则调用MainActivity类的showmsg方法显示"错误"。



因此我们只要在登陆的时候用户名使用admin 密码使用pass71487即可获取到flag如下所示



**实验2** 完成目标程序**jstz.apk**的逆向分析。

**实验目的：**

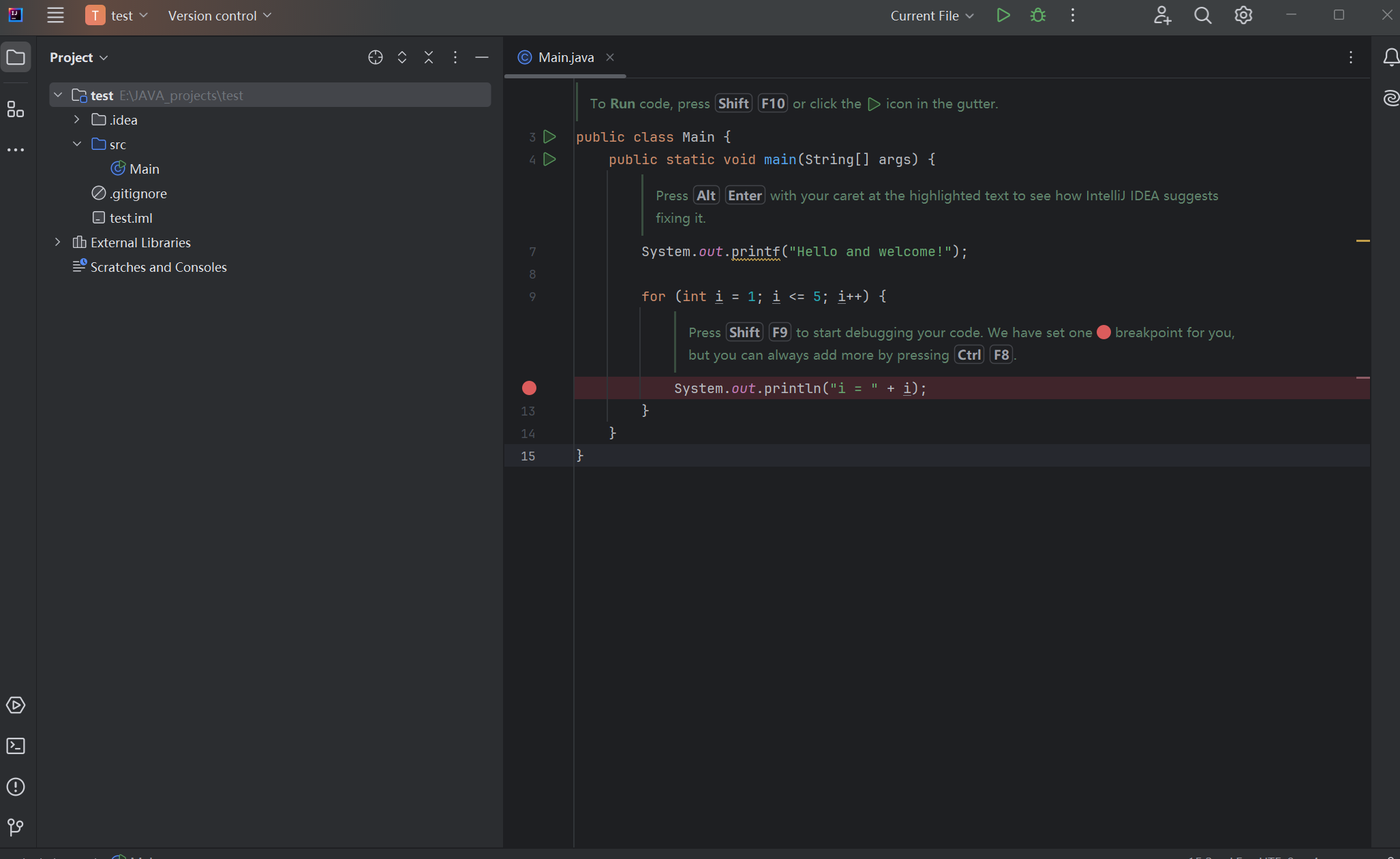
* + 学会基本的Android程序逆向分析方法，结合多种逆向框架进行基础的逆向分析，算法还原分析，得到**jstz.apk**的关键信息
  + unidbg的学习
  + unidbg环境的搭建
  + unidbg代码的编写
  + **jstz.apk**题目的分析

**实验题目：**

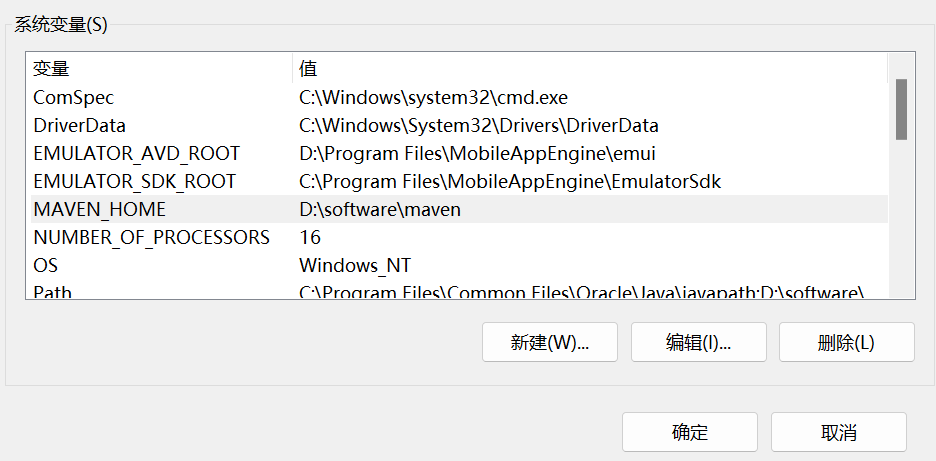
1、完成逆向题的**jstz.apk**题目，拿到真正的flag

# 1.配置环境

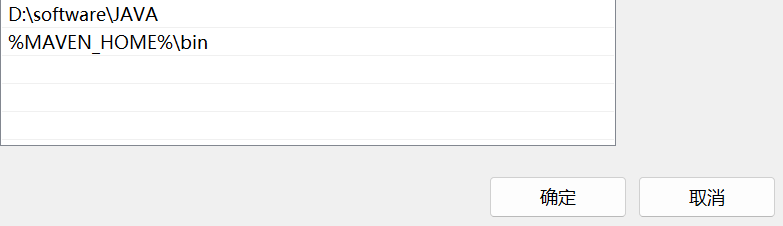
首先下载idea社区版



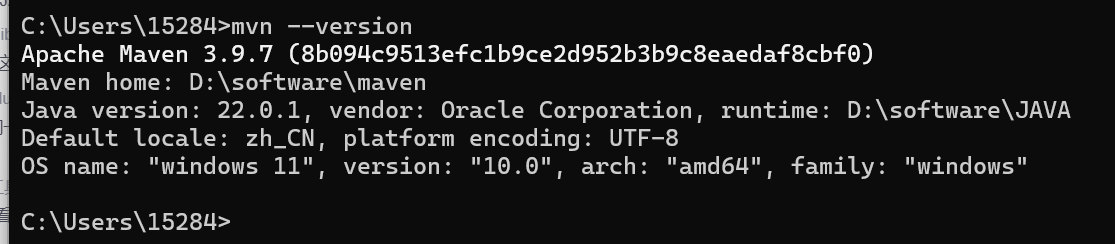
然后下载maven，将压缩包解压之后配置环境变量MAVEN\_HOME，值设置为maven的解压路径。



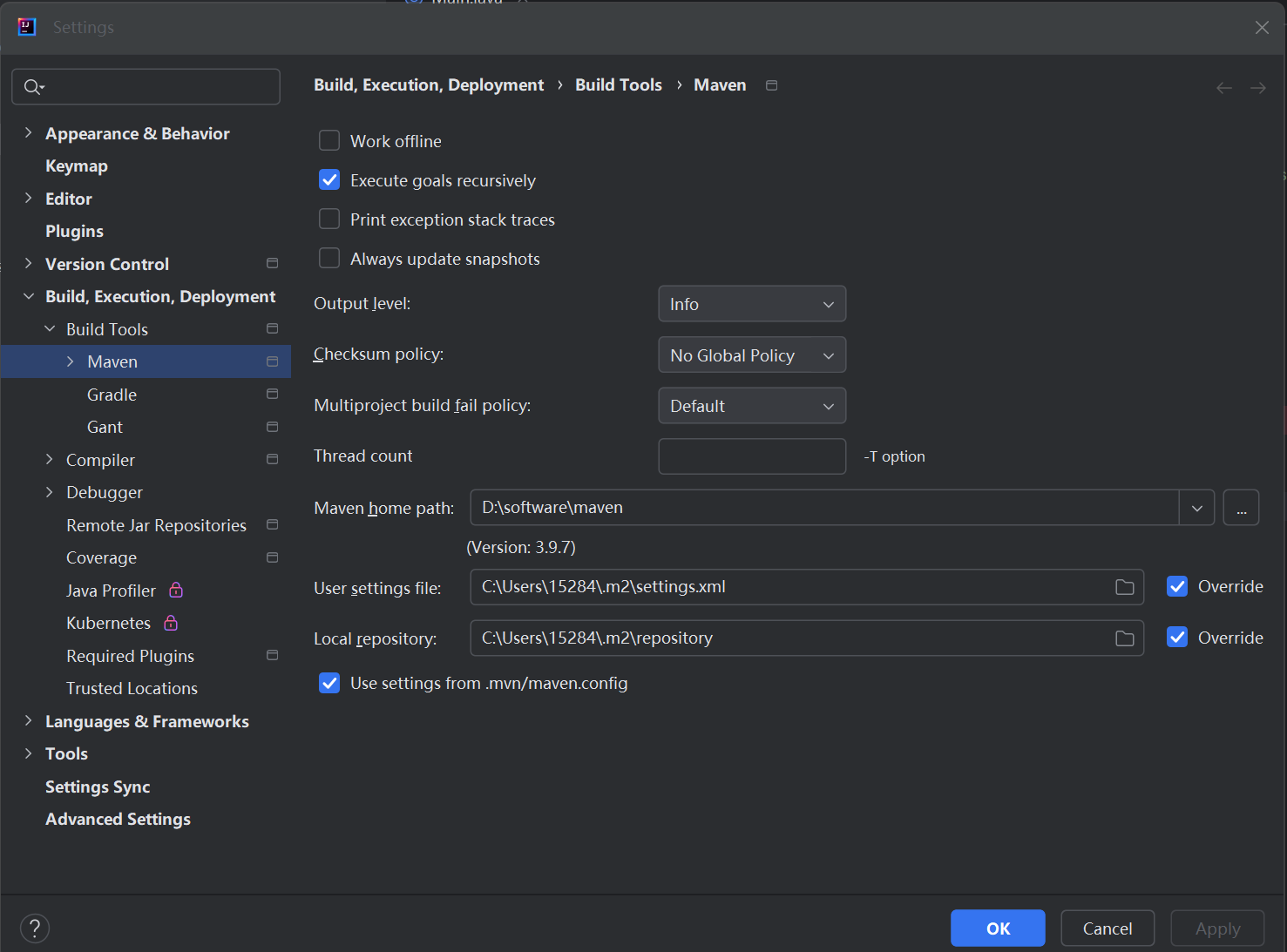
在系统变量的Path中添加环境变量



在终端中查看maven版本看是否配置完毕，已经配置完毕



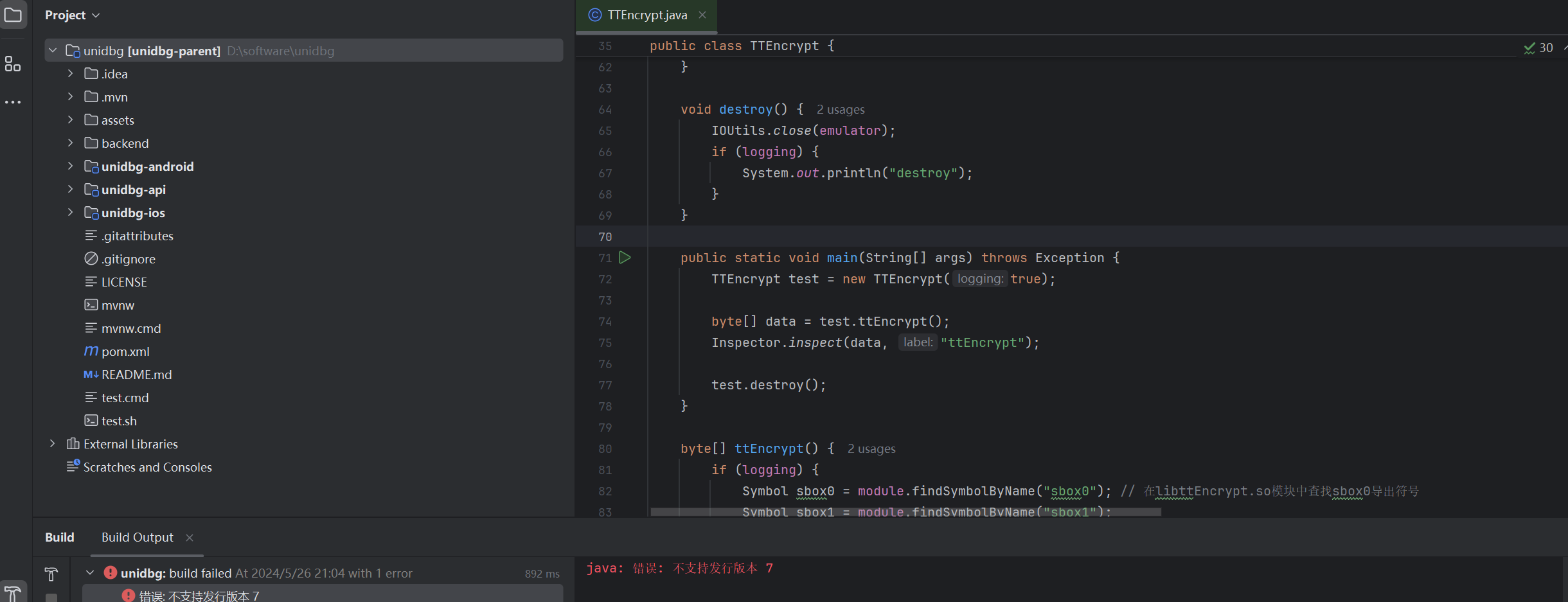
之后再idea中找到Maven模块，设置模块地址为maven的安装地址



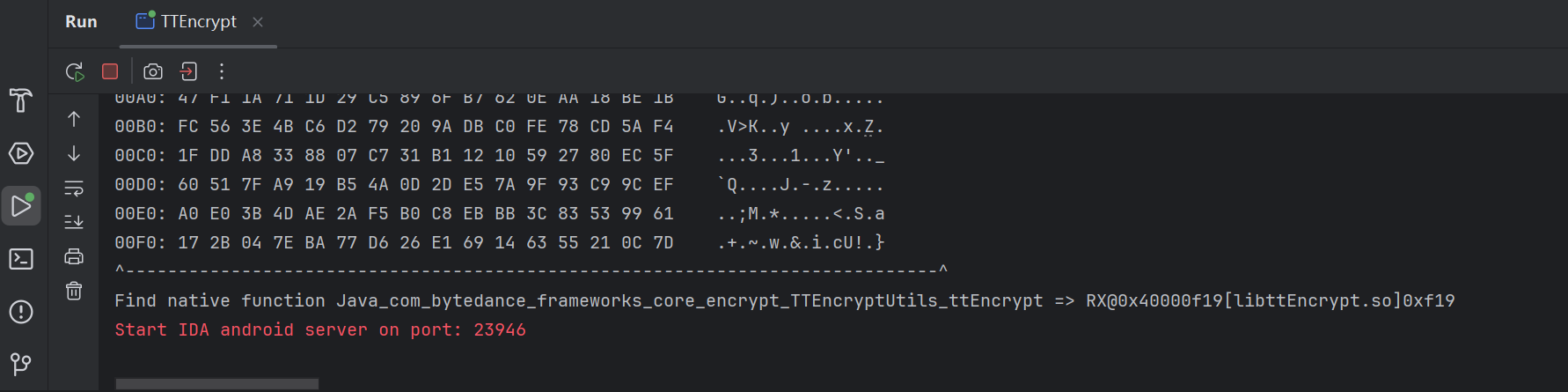
然后在github中将unidbg代码克隆到本地项目中

git clone https://github.com/zhkl0228/unidbg.git

运行代码出现如下报错



查找原因应该是本机的java版本为22，unidbg的版本是7，下载可以找到的最老的java版本java17下载后运行程序如下所示，成功配置Unidbg环境



# 2.实验步骤

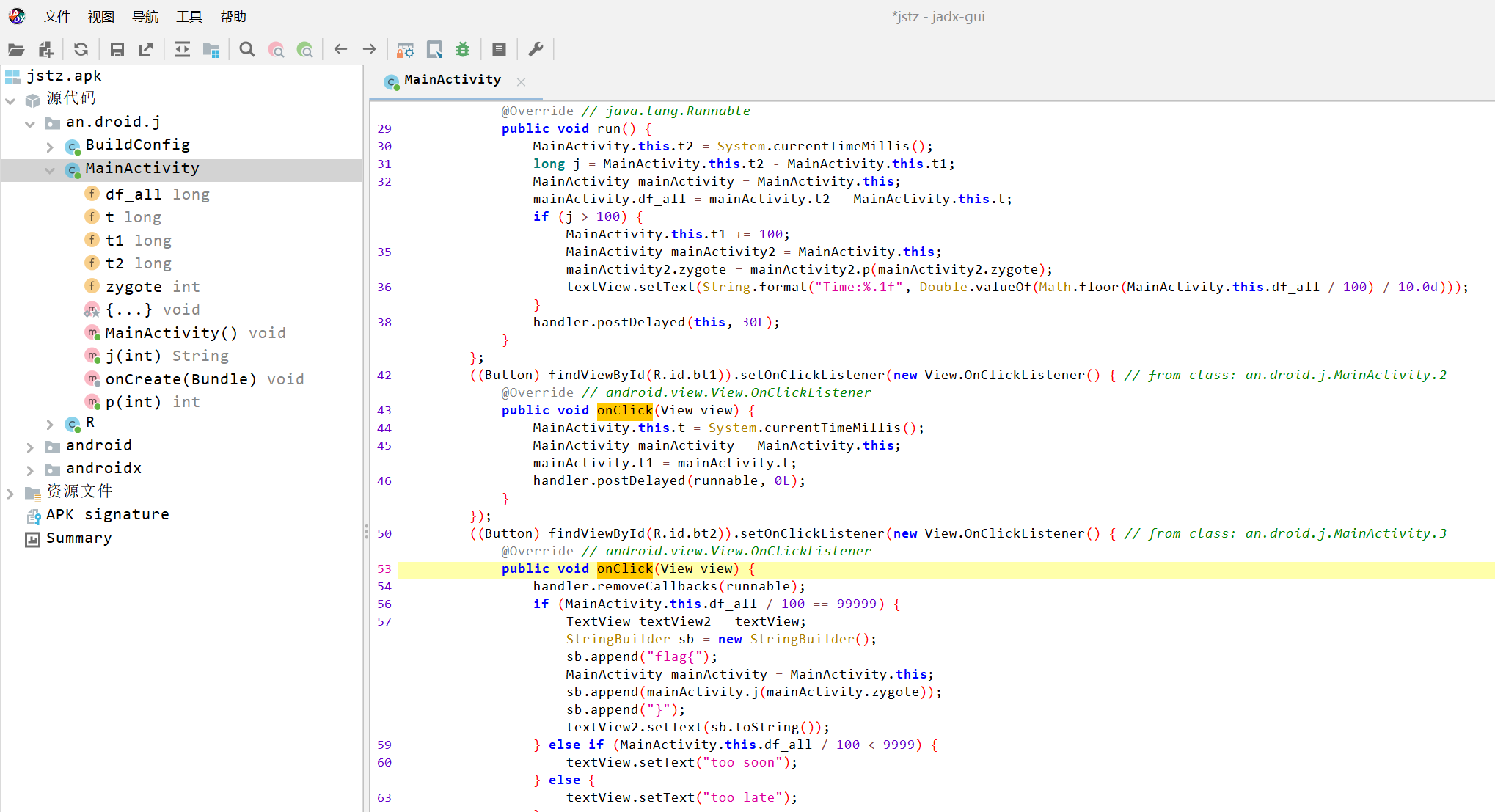
## 2.1 App Messenger分析

首先使用App Messenger查看该apk程序的基本信息如下



## 2.2 jadx分析

因为程序并没有进行加壳，所以我们可以直接将程序使用jadx打开如下所示，可以发现程序只有一个类



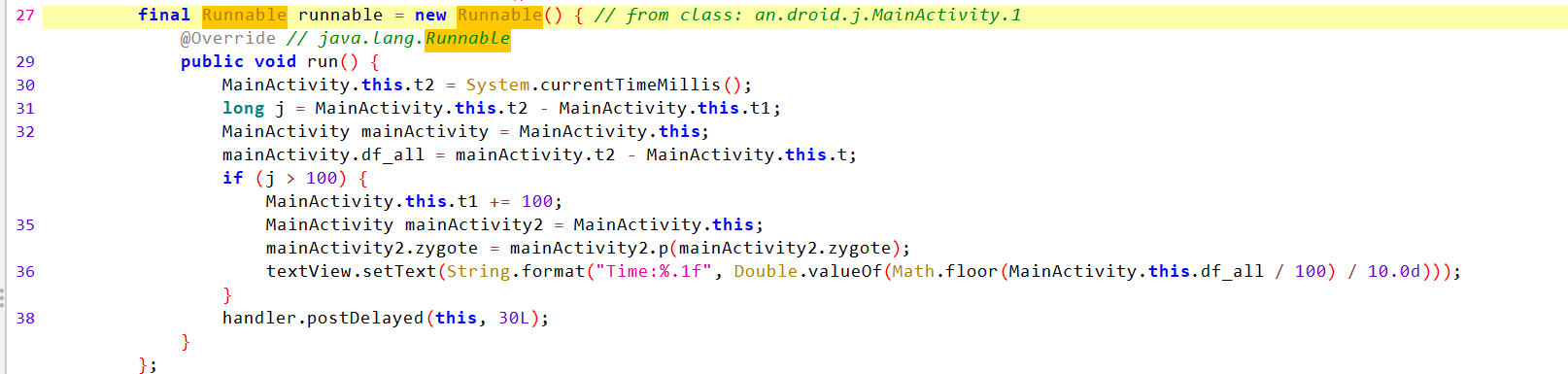
点击程序的START按钮后便会执行onClick方法



执行的runnable代码如下，

将当前的系统时间（以毫秒为单位）赋值给 t2。j 表示从上一次记录时间（t1）到现在（t2）的时间差，也就是上一次更新操作后至本次操作的时间间隔（毫秒）。df\_all 表示从第一次开始计时（t，可能在某个特定事件触发时设置）到当前时间（t2）的总时间差。

然后执行if语句，检查自上一次更新以来是否已经过了100毫秒。如果是则将起始时间点t1向前推进100毫秒。这意味着run方法将基于新的时间点继续监控，然后调用本地方法 p，传入当前的 zygote 值，并更新 zygote，最后将 df\_all（以毫秒计的总时间差）转换为秒，并保留一位小数显示在 textView 上。这通过将毫秒数除以100并向下取整，然后除以10.0来完成，得到以秒为单位的时间。



程序点击END按钮时会执行如下onClick方法，

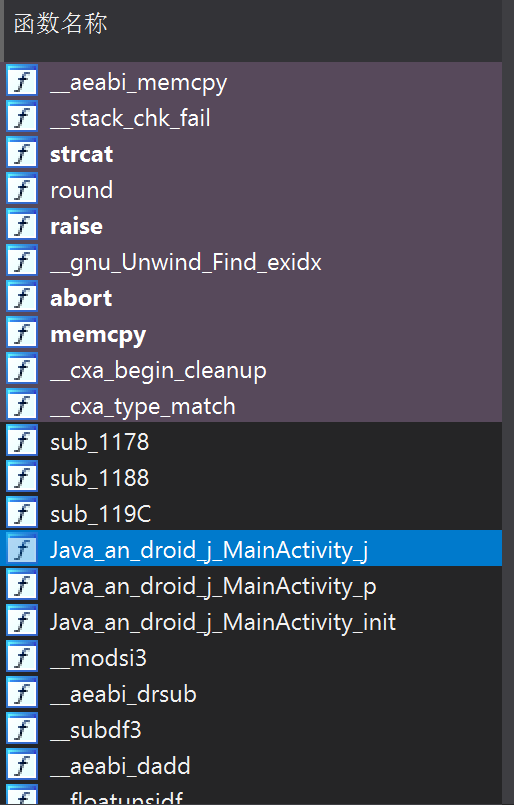
该方法是当df\_all变量的值即程序运行的时间为9999900ms时就将zygote调用j方法进行输出，输出的内容就是我们需要的flag



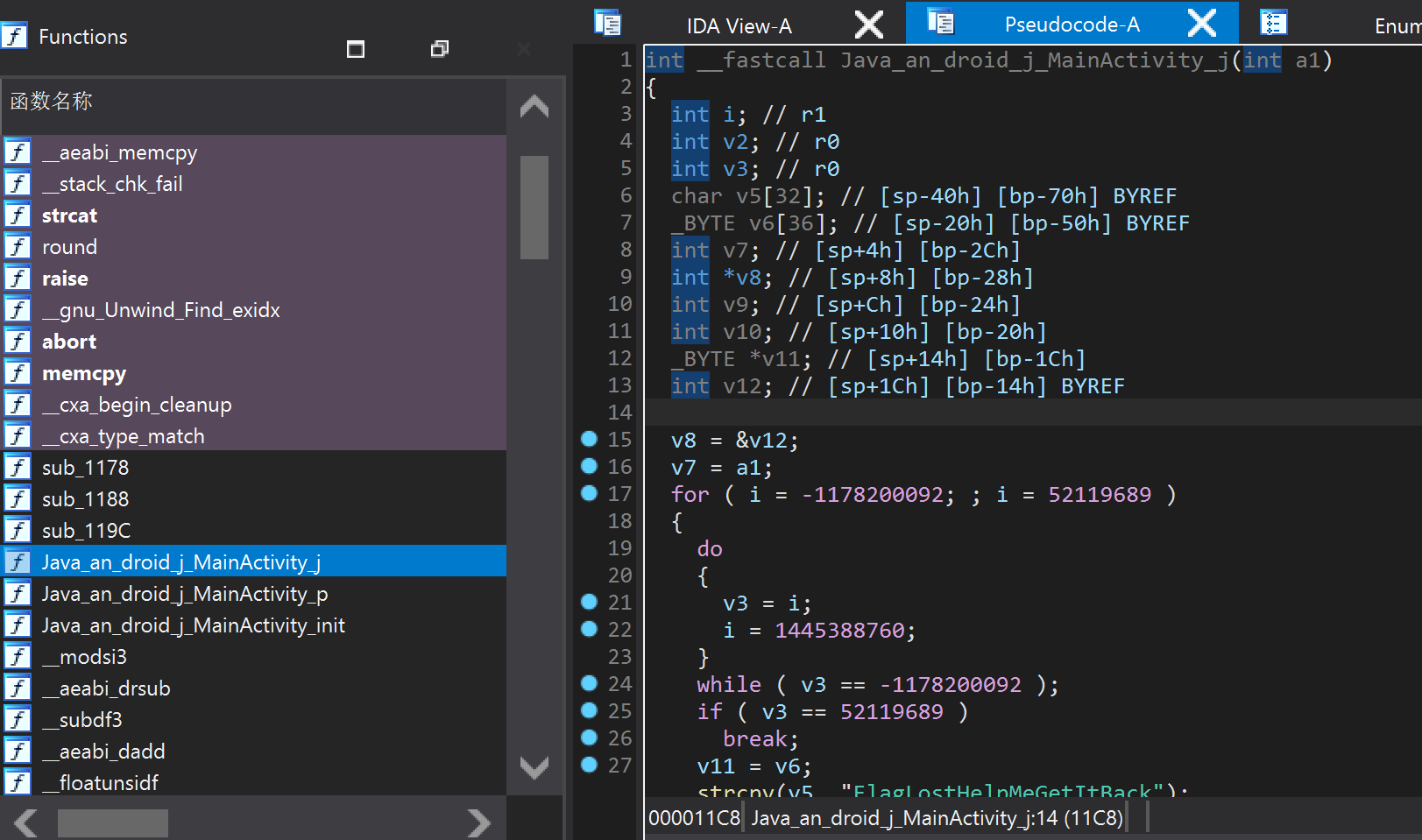
综上所述我们可以得出程序的基本逻辑是点击START按钮之后每隔100ms就使用p方法对变量zygote更新，点击END时如果时间为9999900ms则计算输出正确flag

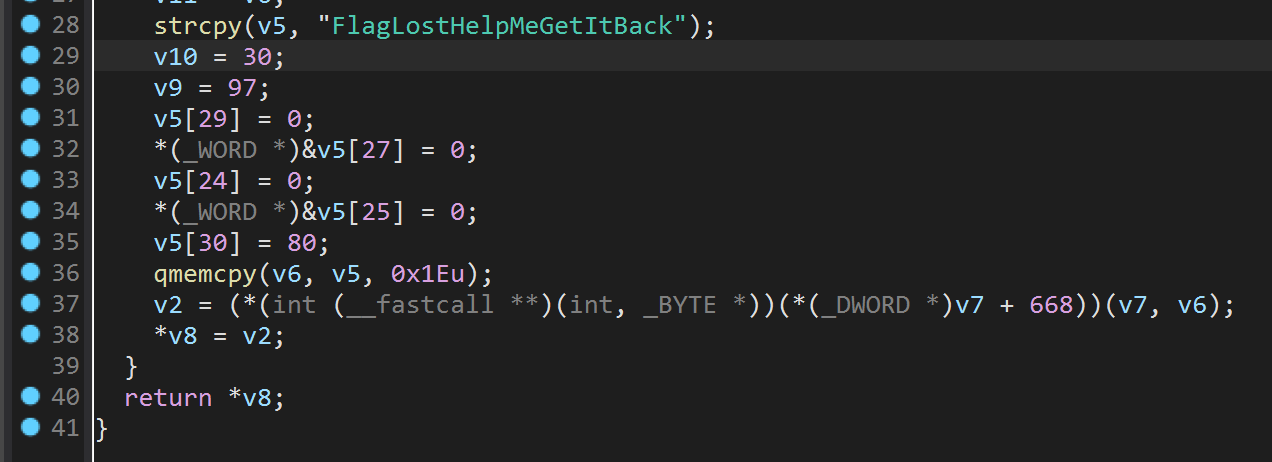
## 2.3 IDA分析

使用ida32打开lib/armeabi目录下的libj.so文件如下可以发现函数列表中确实存在三个静态注册的函数

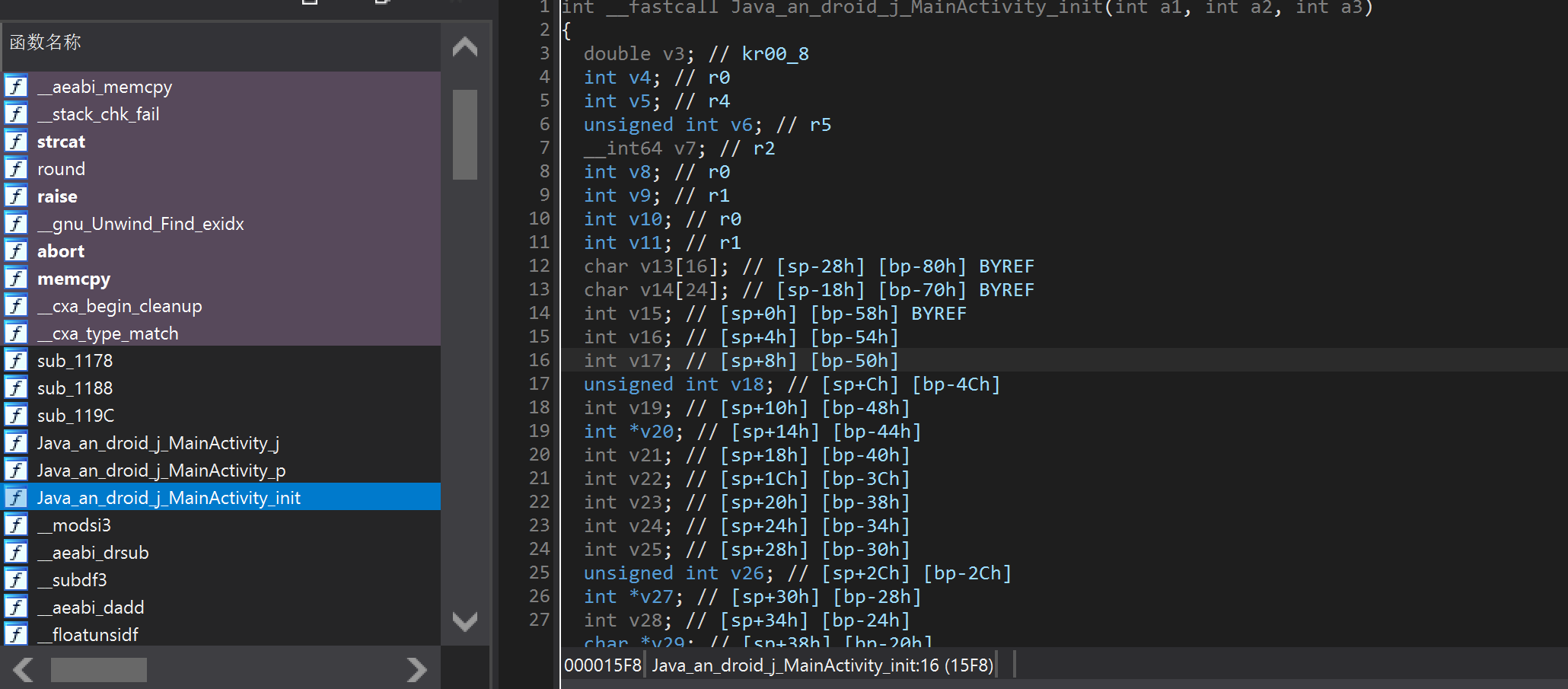


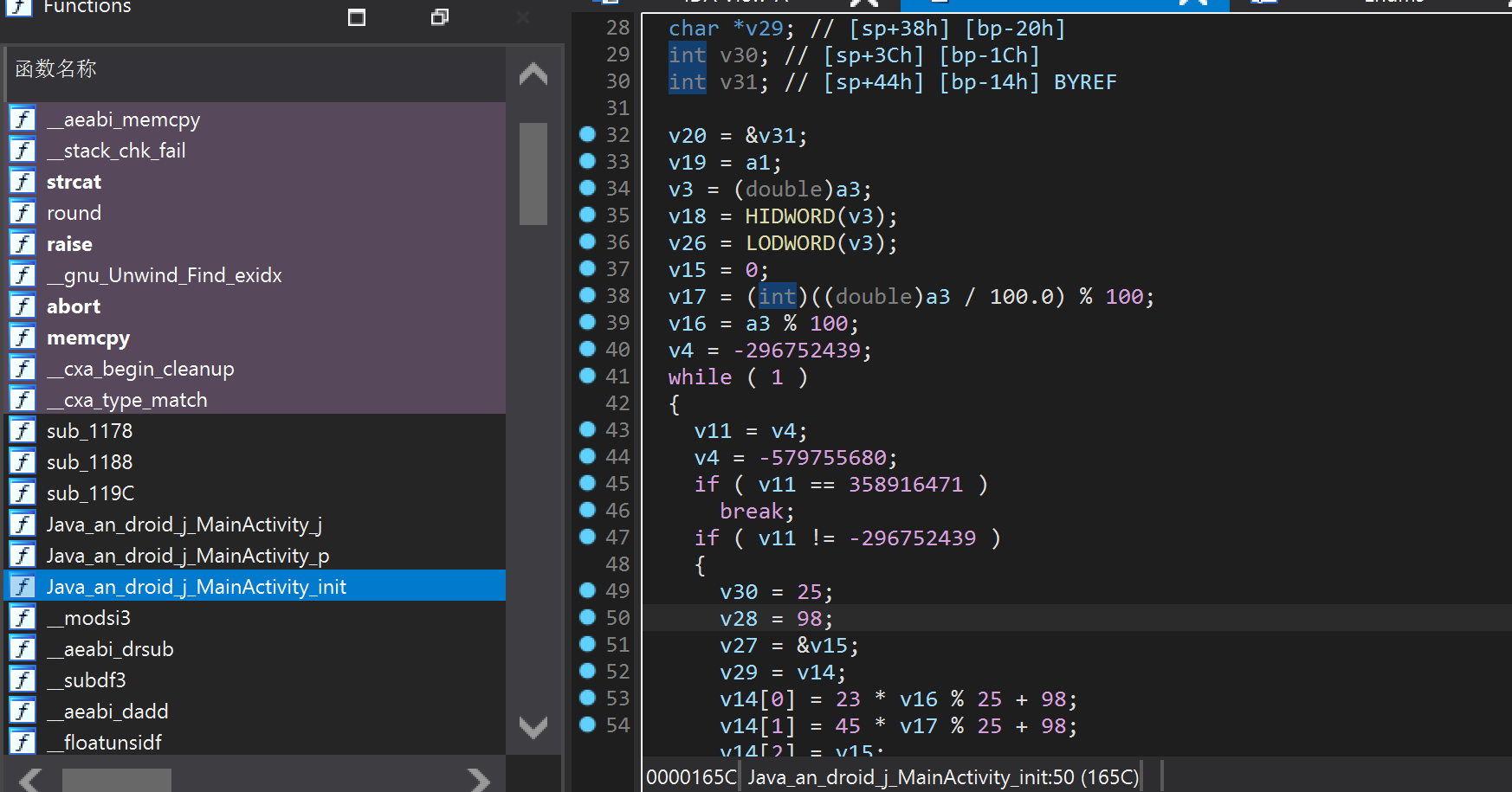
首先打开j方法F5查看反汇编后的结果，发现j方法会固定返回字符串“FlagLostHelpMeGetItBack”但这并不是正确的flag，这也说明我们之前尝试的获取flag的方法获取到的该字符串应该是错误的。

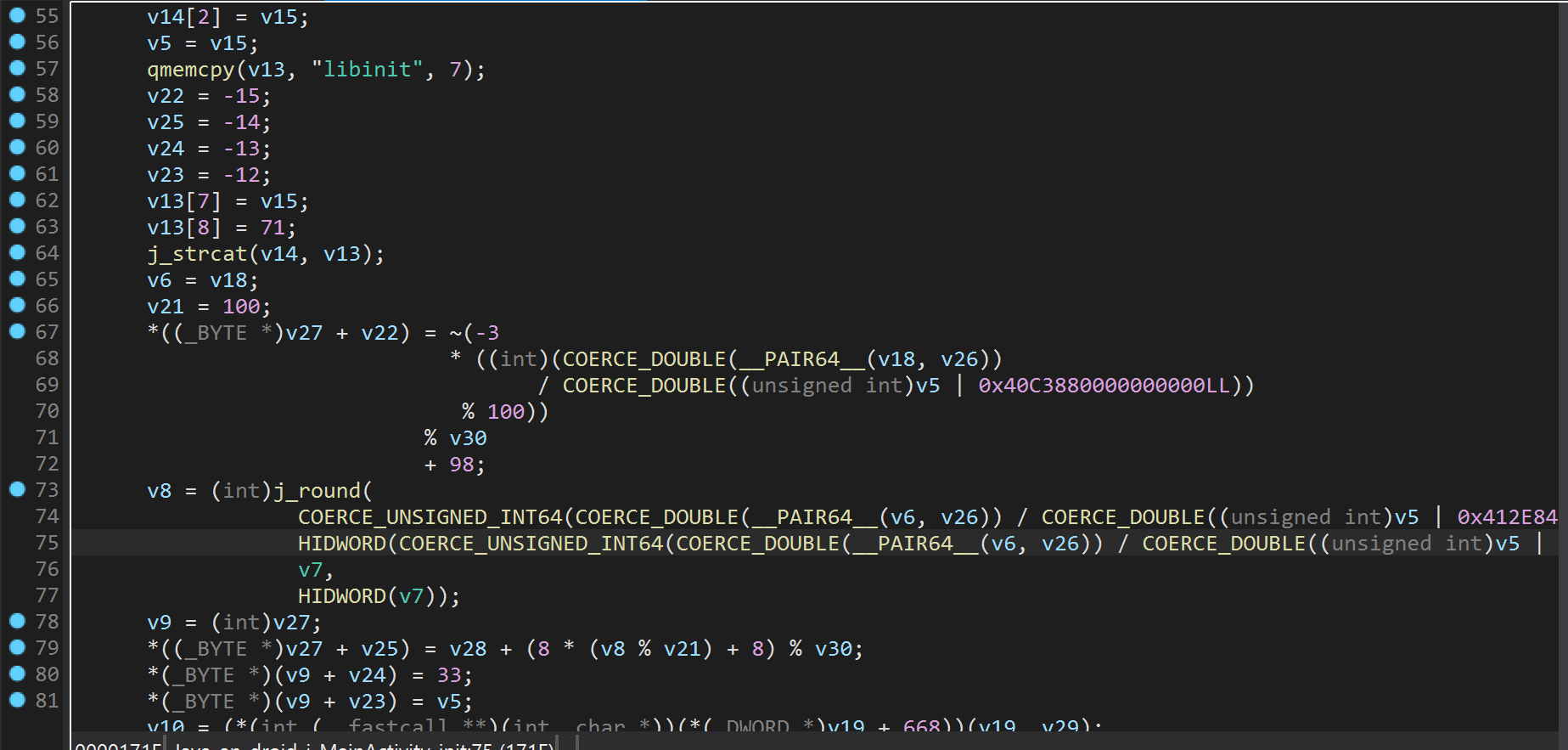


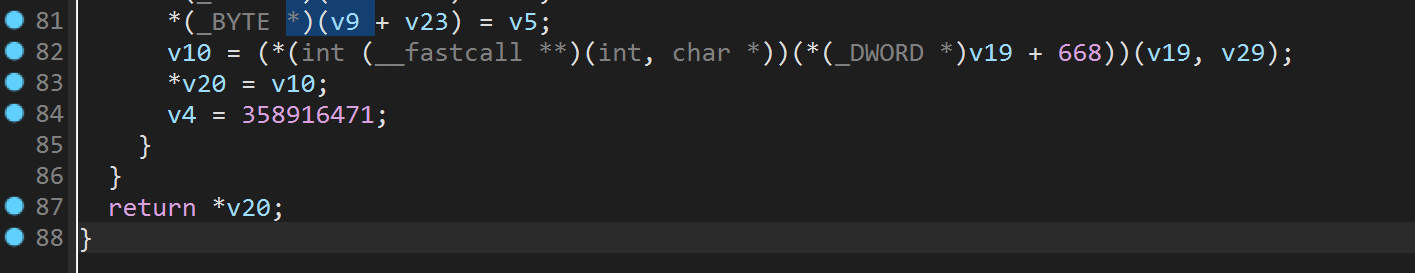


接着查看init方法的反汇编代码，该方法是接受一个double类型的变量取高四位和第四位作为两个数据计算一个含有“libinit”的字符串并返回



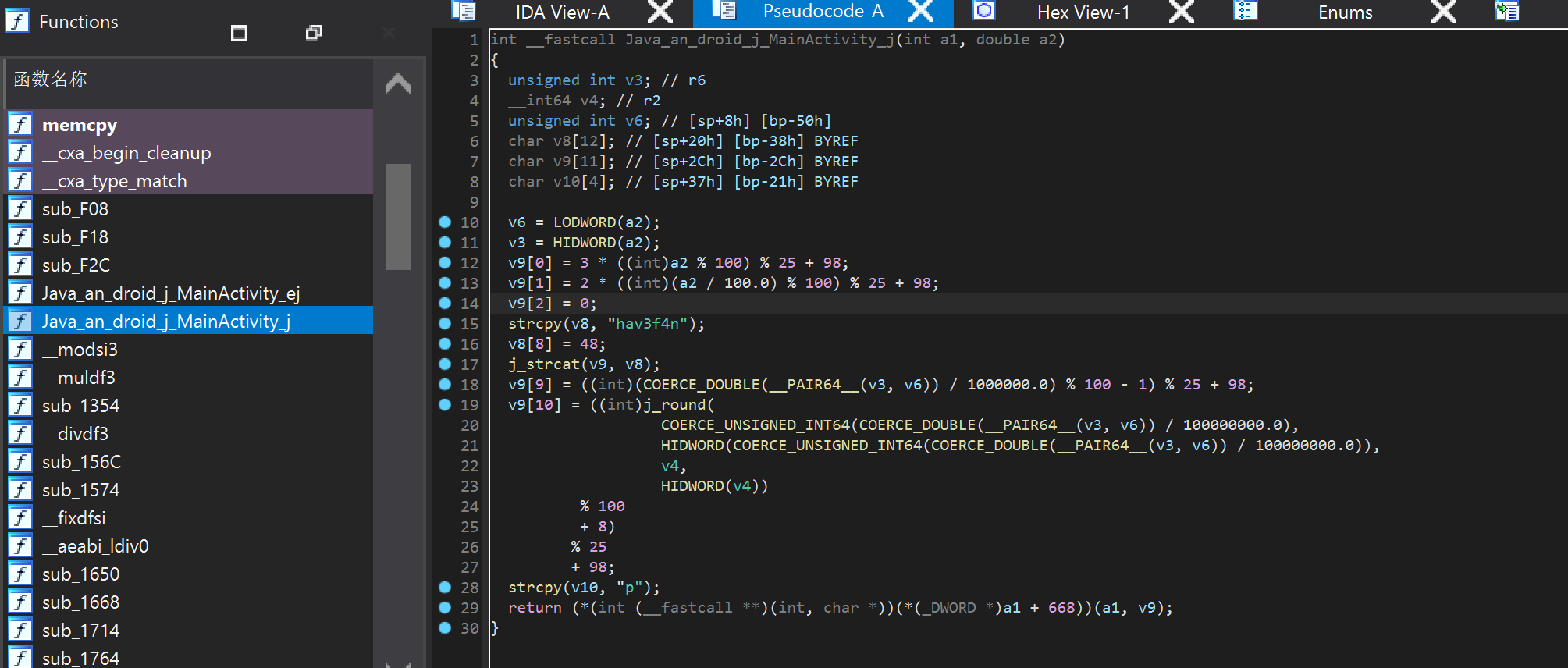






接下来查看bak\_libj.so文件

发现该文件中同样有j方法，其反编译后的代码如下所示，该方法同样是接受double类型的数据，高四位与低四位参与运算后获取一个含有“hav3f4n”字符串的值

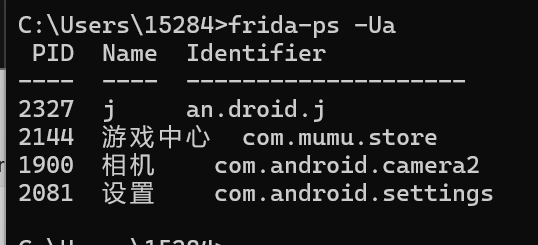


## 2.4 编写frida脚本获取zygote

使用教程中提供的脚本如下所示



查看包名和pid

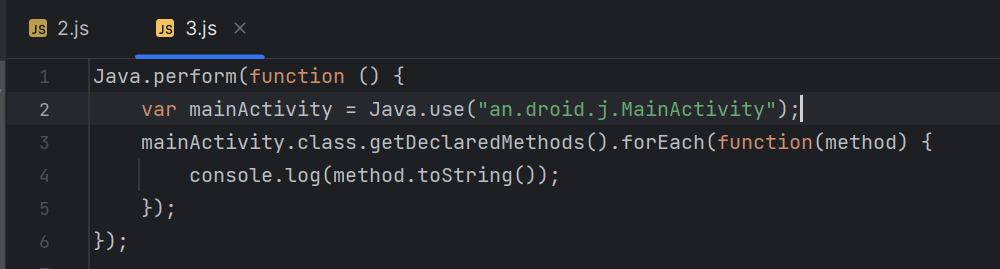


frida -U -f an.droid.j -l 2.js会出现以下报错

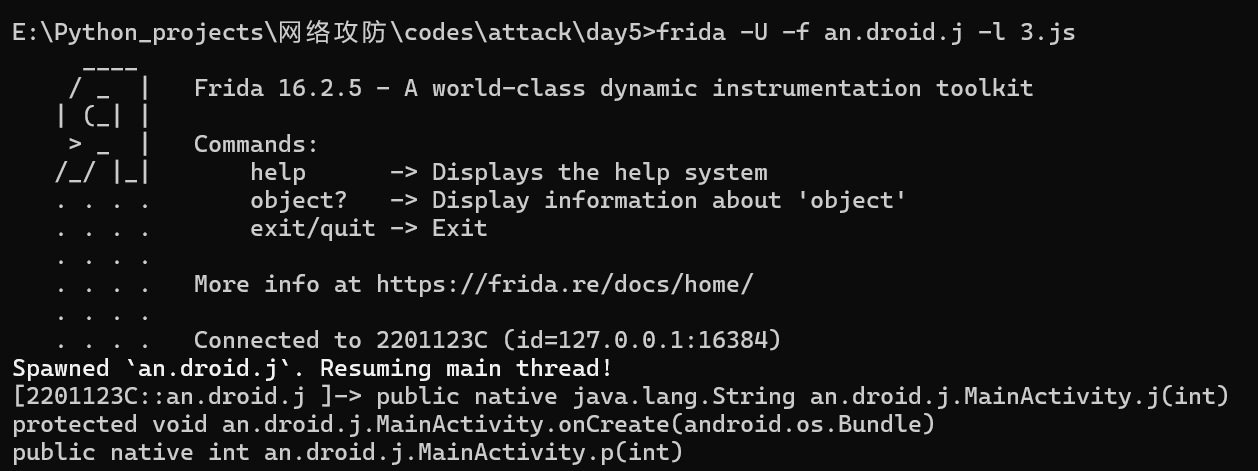


分析原因是因为p方法无法定位到

编写脚本查看mainActivity下的所有方法如下



执行结果如下，能够找到p方法

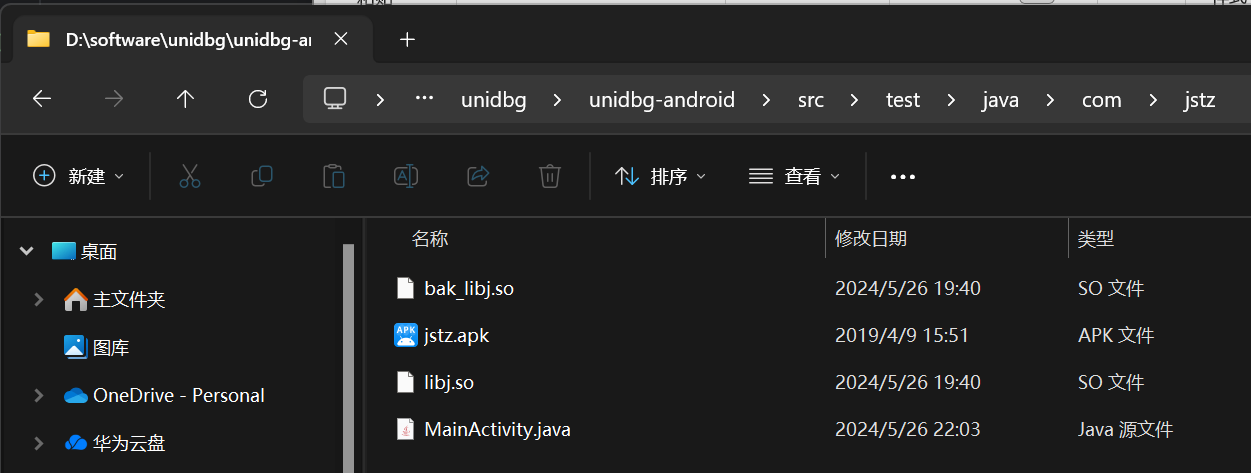


2.5 Unidbg

放弃使用frida计算下面尝试教程中所说的全程Unidbg

首先需要为unidbg搭架子

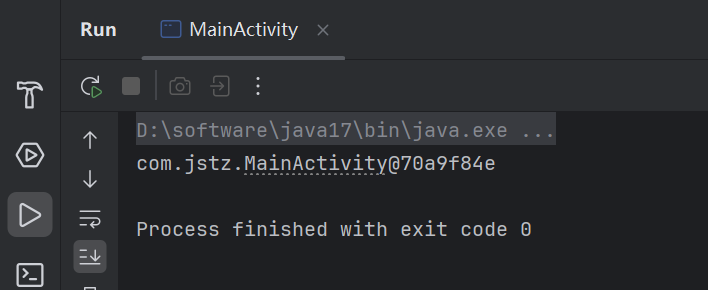
在已经下载好的Unidbg包下面D:\software\unidbg\unidbg-android\src\test\java\com目录下新建目录jstz作为我们的项目位置，然后将apk文件和两个so文件放到该项目包并创建MainActivity.java文件



编写MainActivity文件的内容如下所示

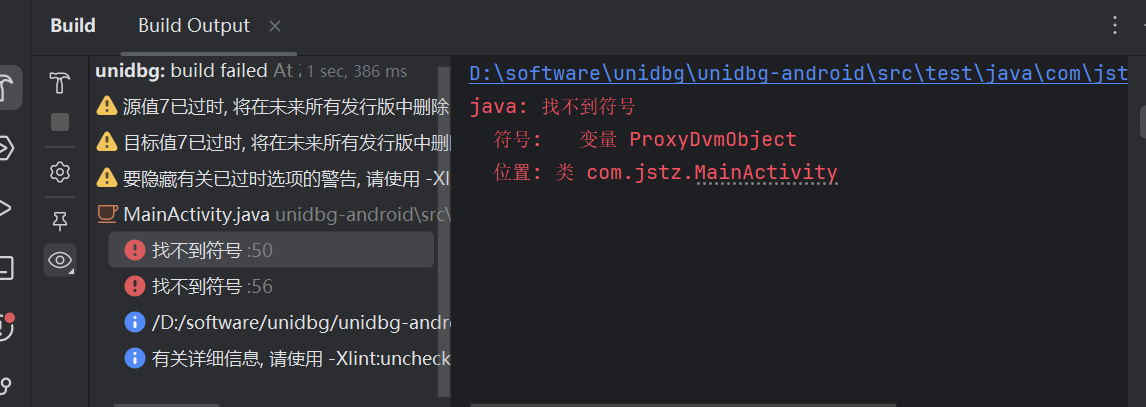
package com.jstz;  
  
import com.github.unidbg.AndroidEmulator;  
import com.github.unidbg.Module;  
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;  
import com.github.unidbg.linux.android.AndroidResolver;  
import com.github.unidbg.linux.android.dvm.\*;  
import com.github.unidbg.memory.Memory;  
import com.github.unidbg.pointer.UnidbgPointer;  
import com.github.unidbg.utils.Inspector;  
import com.sun.jna.Pointer;  
import keystone.Keystone;  
import keystone.KeystoneArchitecture;  
import keystone.KeystoneEncoded;  
import keystone.KeystoneMode;  
  
import java.io.File;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
public class MainActivity extends AbstractJni {  
 private final AndroidEmulator emulator;  
 private final VM vm;  
 private final Module module;  
  
 MainActivity() {  
 // 创建模拟器实例,进程名建议依照实际进程名填写，可以规避针对进程名的校验  
 emulator = AndroidEmulatorBuilder.*for32Bit*().setProcessName("com.jstz.MainActivity").build();  
 // 获取模拟器的内存操作接口  
 final Memory memory = emulator.getMemory();  
 // 设置系统类库解析  
 memory.setLibraryResolver(new AndroidResolver(23));  
 // 创建Android虚拟机,传入APK，Unidbg可以替我们做部分签名校验的工作  
 vm = emulator.createDalvikVM(new File("unidbg-android\\src\\test\\java\\com\\jstz\\jstz.apk"));  
 // 加载目标SO  
 DalvikModule dm = vm.loadLibrary(new File("unidbg-android\\src\\test\\java\\com\\jstz\\bak\_libj.so"), true); // 加载so到虚拟内存  
 //获取本SO模块的句柄,后续需要用它  
 module = dm.getModule();  
 vm.setJni(this); // 设置JNI  
 vm.setVerbose(true); // 打印日志  
// dm.callJNI\_OnLoad(emulator); // 调用JNI OnLoad  
 }  
  
 public static void main(String[] args) {  
 MainActivity test = new MainActivity();  
 System.*out*.println(test);  
 }  
}

运行之后得到结果如下,出现正确的回显值说明架子已经搭好

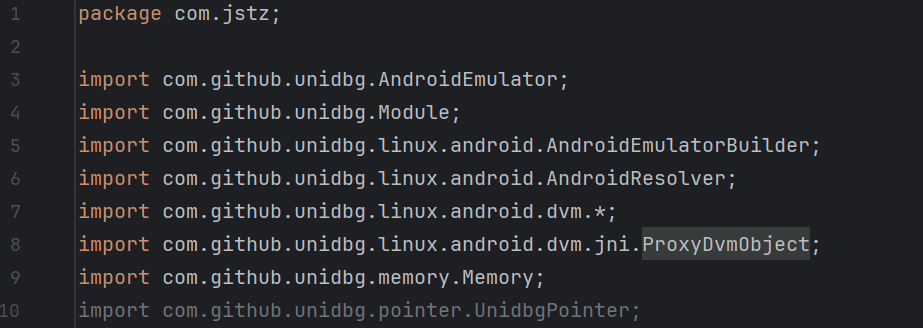


修改MainActivity的内容

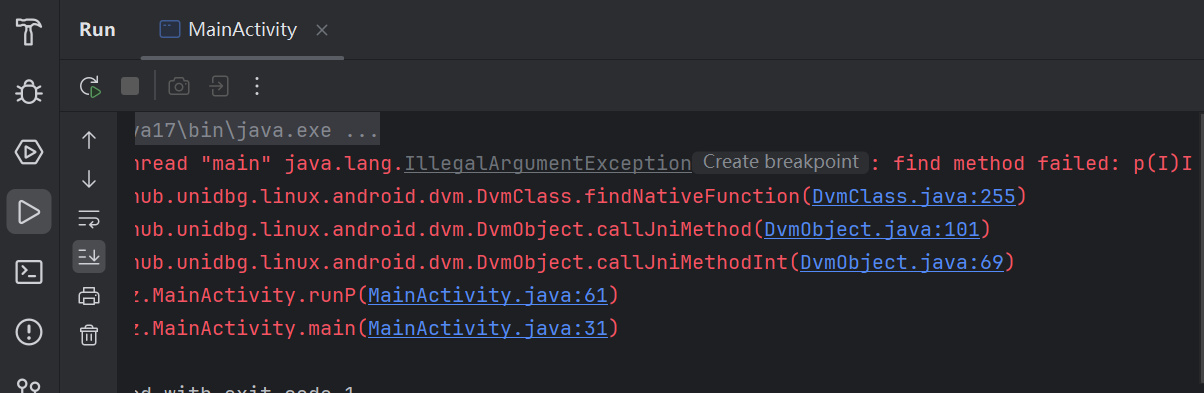
运行程序如下发现不能够识别ProxyDvmObject



在导入栏中导入该包



继续运行出现以下报错



同样是找不到对应的方法

在更换调用方法的方式之后进行调试发现第二次的zygote的值会被计算为287，与我们想要的值不符合，所以断定更换调用方法之后仍然不能解决问题

最终修改完成的代码如下所示

package com.jstz;  
  
import com.github.unidbg.AndroidEmulator;  
import com.github.unidbg.Module;  
import com.github.unidbg.Symbol;  
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;  
import com.github.unidbg.linux.android.AndroidResolver;  
import com.github.unidbg.linux.android.dvm.\*;  
import com.github.unidbg.linux.android.dvm.jni.ProxyDvmObject;  
import com.github.unidbg.linux.android.dvm.wrapper.DvmInteger;  
import com.github.unidbg.memory.Memory;  
import com.github.unidbg.pointer.UnidbgPointer;  
import com.github.unidbg.utils.Inspector;  
import com.sun.jna.Pointer;  
import keystone.Keystone;  
import keystone.KeystoneArchitecture;  
import keystone.KeystoneEncoded;  
import keystone.KeystoneMode;  
  
import java.io.File;  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
public class MainActivity {  
 int zygote = 1357024680;  
 public static void main(String[] args) {  
 MainActivity mainActivity = new MainActivity();  
  
 // 使用Unidbg模拟执行99999次并计算Flag  
 long start = System.*currentTimeMillis*();  
 for(int j =0;j<99999;j++){  
 mainActivity.zygote = mainActivity.runP(mainActivity.zygote);  
 }  
 System.*out*.println("zygote = " +mainActivity.zygote);  
 System.*out*.println("flag{" + mainActivity.runInit(mainActivity.zygote) + "}");  
 long times = System.*currentTimeMillis*() - start;  
 System.*out*.println("times = " + times);  
 }  
 private final AndroidEmulator emulator;  
 private final Memory memory;  
 private final VM vm;  
  
 public MainActivity() {  
 emulator = AndroidEmulatorBuilder  
 .*for32Bit*().build();  
 memory = emulator.getMemory();  
 memory.setLibraryResolver(new AndroidResolver(19));  
 vm = emulator.createDalvikVM(new File("unidbg-android\\src\\test\\java\\com\\jstz\\jstz.apk"));  
 vm.setVerbose(false);  
 vm.loadLibrary("j",false);  
 }  
  
 public String runInit(int i){  
 DvmObject obj = ProxyDvmObject.*createObject*(vm,this);  
 DvmObject object = obj.callJniMethodObject(emulator, "Java\_an\_droid\_j\_MainActivity\_init", i);  
 String result = (String) object.getValue();  
 return result;  
 }  
  
 public int runP(int i){  
 DvmObject obj = ProxyDvmObject.*createObject*(vm,this);  
 int number = obj.callJniMethodInt(emulator, "Java\_an\_droid\_j\_MainActivity\_p", i);  
 return number;  
 }  
}

但是该代码在执行过后仍然会产生如下报错，分析报错仍然为找不到具体包的位置

