**安卓ART虚拟机Java Hook框架JavaTweak的设计与实现**

**设计背景**

目前主流的JAVA层hook框架是xposed，但需要重编译art虚拟机，也过于笨重，更重要的是会被app各种检测，有些app一旦发现安装直接强制退出。为了不依赖xposed实现Java方法Hook，深入了解art运行机制，因而重新实现了一个轻型的Hook框架(JavaTweak)，目前已经在Andorid 5.0到8.1的系统上测试通过。在实现的过程中，参考了不少同行的实现思路（legend、SandHook、YAHFA），借鉴了其中做的比较好的地方再结合了自己的一些理解，力求在使用上尽可能的简单，设计上尽可能降低系统版本依赖。

**常见框架比较**

1. **Xposed**

地址：<https://github.com/rovo89/Xposed>

此框架最为知名使用最广，也是最重量级的一个框架。在持续的维护下插件众多，学习资料丰富，功能强大且支持从Andorid5到8的所有版本，但9以上版本不再支持，其源码是研究art下java方法hook实现的必读代码。

Native层它重编译了ART虚拟机，在runtime/art\_method.cc中添加了一个核心方法ArtMethod::EnableXposedHook，实现了java方法的hook。重新编译了app\_process，实现框架加载并初始化，搜索并加载指定插件。Java层则提供XposedBridgeAPI-xx.jar包，方便了插件开发。提供XposedInstaller，管理框架的安装和卸载、管理插件的启用和禁用。

**二、SandHook**

地址：<https://github.com/ganyao114/SandHook>

主要基于内联hook，条件不允许时进行入口替换方式hook。与xposed回调式的hook方式不同，sandhook采用注解式的hook方式。支持直接Hook，也支持在插件场景中的Hook。

支持Android 9的hook。非root手机也可使用，但只限于自身app内部，如果是hook外部app，则必须运行沙箱中。未提供安装和卸载框架的app，只能自行编译so和jar包，使用上个人觉得有点繁琐，但hook实现值得参考。

**三、Legend**

地址：<https://github.com/asLody/legend>

此框架是art hook的一个java实现，其hook逻辑都在java层完成，native层只做内存分配以及数据拷贝。受到在java层hook的局限，此框架只支持到Andorid 6(因为7以上无法在java层创建ArtMethod对象)。使用上与SandHook一样采用注解式的hook方，使用简单。在Android5、6上能稳定运行，与在native层实现hook相比，在java层实现hook兼容性会好一些，且提供了一种新的hook思路，值得借鉴。

**四、YAHFA**

地址：<https://github.com/PAGalaxyLab/YAHFA>

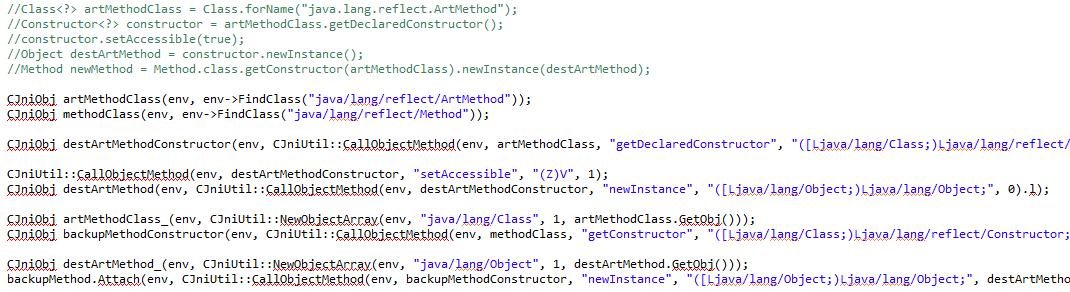
此框架专为逆向开发而设计，所以hook前没有像SandHook那样对类的加载解析状态做判断，预设类和方法都已经加载解析完成。Hook实现设计简洁精巧，内联代码高效易懂。支持到Android10。使用上没有采用注解方式hook,而是要为每个需要hook的方法编写一个HOOK类，再通过配置HOOK类的方式来完成方法的hook，所以使用起来比较繁琐。

**方法Hook过程**

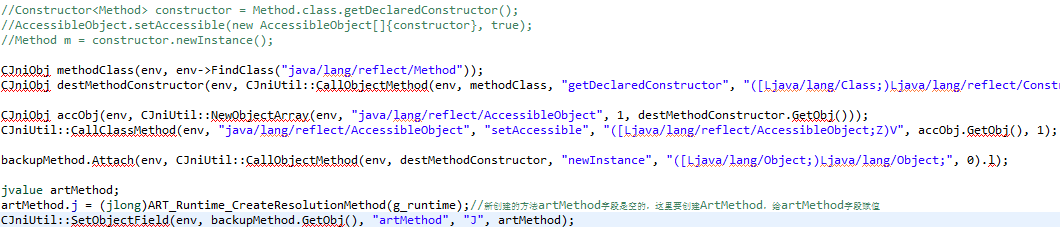
**一、动态创建一个空方法**

java层对应的是一个Method对象，native层对应一个AbstractMethod结构。AbstractMethod从8.0开始更名为Executable，AbstractMethod结构体内部包含一个核心结构体ArtMethod。ArtMethod在不同的android系统中字段不完全一致，且从6.0开始结构体中的字段在java层被完全隐藏。ArtMethod 中的【quick\_compiled\_code\_】字段，就是后续HOOK需要处理的关键字段。7.0之前创建空方法是被允许的，7.0及之后不能动态创建Method对象，只能事先在java层中定义一些空方法用来做方法备份。

5.0-5.1创建Method代码



6.0创建Method代码

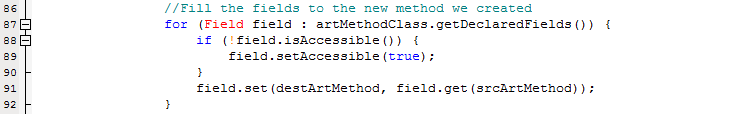


6.0之后创建Method代码



**二、拷贝原方法的所有数据覆盖空方法**

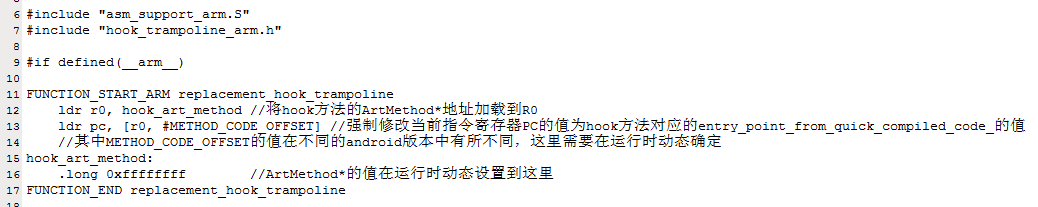
AbstractMethod结构体字段在java层是可见的，所以可以通过遍历java字段方式逐个赋值，通过反射调用在native层实现如下代码。



ArtMethod结构体从6.0开始在java层不可见，并且不同android系统字段不完全相同，结构体大小也不同，逐个字段赋值显然会有系统兼容问题，这里采用memcpy整个拷贝方式，至于求不同平台ArtMethod大小，思路是：art初始化类时会给类中所有的方法分配空间，他的底层是一个数组，通过ptr指针给ArtMethod分配空间，由于ArtMethod在art中是紧密排列的，这样就能通过自己构建一个类，里面拥有两个静态方法，去计算两个静态方法之间的差值，从而获得ArtMethod的大小。

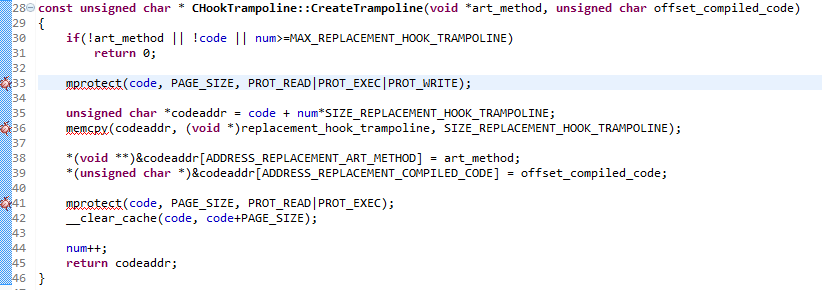
**三、构造汇编跳板代码，并替换原方法的编译代码执行入口**

当虚拟机调用java方法时，会首先跳到上面提到的【quick\_compiled\_code\_】字段指向的编译代码点，此时各个寄存器存放的内容会遵循固定的java调用约定。R0寄存器始终指向被调用方法的ArtMethod地址，所以如果需要执行我们自己的hook函数，只要将R0寄存器设置为hook方法的ArtMethod地址，并强制跳转到hook方法对应的entry\_point\_from\_quick\_compiled\_code\_代码点继续执行即可。如下图所示的汇编代码段



现在只需要将replacement\_hook\_trampoline函数，

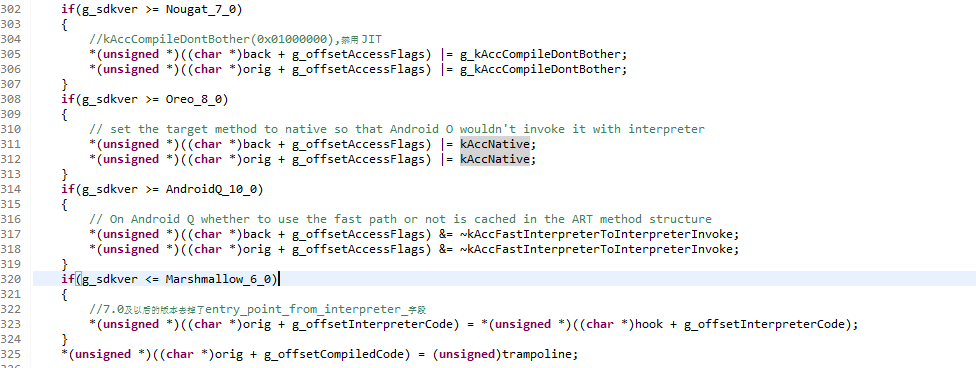
替换原方法的quick\_compiled\_code\_即可完成java方法的hook。



执行完我们的方法后，通过反射调用备份方法，即可继续完成原方法的调用。

**四、微调备份方法和原方法的字段属性**

在得到备份方法后，需要对方法熟悉进行一些细微调整，比如设置备份方法可被正常调用，因为原始方法可能是私有或保护方法，外部不能直接调用（setAccessible）。安装跳板，替换原方法的entry\_point\_from\_quick\_compiled\_code\_值为跳板汇编代码。禁用JIT，防止已经hook好的方法被再次内联编译，导致hook失效，主要是设置accessFlags字段的属性标记。这一步在不同android版本中有不同的处理，具体可以参考legend、SandHook、YAHFA等框架的实现。



设置备份方法为私有方法。



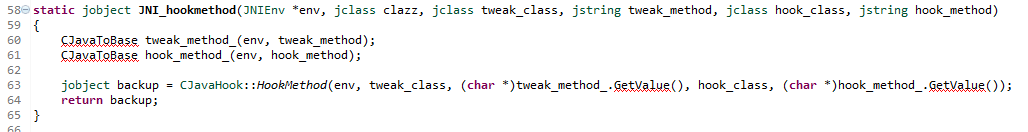
设置已hook标记，防止重复hook。



**JavaTweak Hook框架实现**

**一、Native层功能**

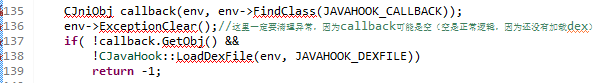
为java层提供方法hook的native实现，实现原理前面已经介绍过了。这里只需要注册JNI方法，供上层调用即可。



所有java代码(后续会介绍到)会被编译成jar包，这一步可以通过命令行完成，也可以借助ide工具中完成。 对于android系统，jar包不能直接加载，必须借助android SDK的构建工具dx将jar文件转换为dex文件。具体命令行如下：

dx --dex --output=javatweak.dex javatweak.jar

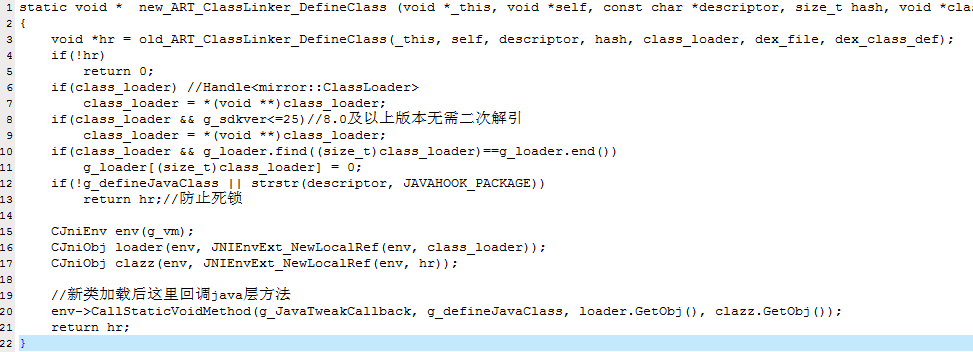
Native负责将javatweak.dex文件加载，并将文件路径添加到默认加载器的DexPathList中。如下图CJavaHook::LoadDexFile函数。



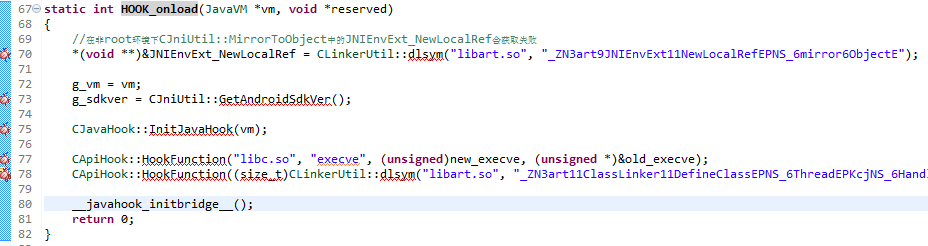
监控java层每个类的加载，每当有新类加载时，通过反射回调给java层。这样可以在第一时间hook好其中的方法。对于多dex，多classloader的app，采用这种回调模式会方便很多，不用再去考虑方法的hook时机，类加载器如何获取等问题。每个类的加载都会调用ClassLinker::DefineClass方法，具体加载流程可以参考源码。

/art/runtime/native/dalvik\_system\_DexFile.cc:DexFile\_defineClassNative

ClassLinker::DefineClass方法在android的5.0之后的定义都是固定不变的，可以hook此方法，当有类被加载时，及时回调java。



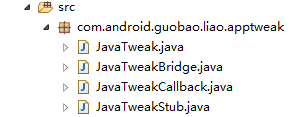
至此，Native层的主要功能就实现完成了，可以在JNI\_OnLoad函数中完成上述功能，代码片段如下。



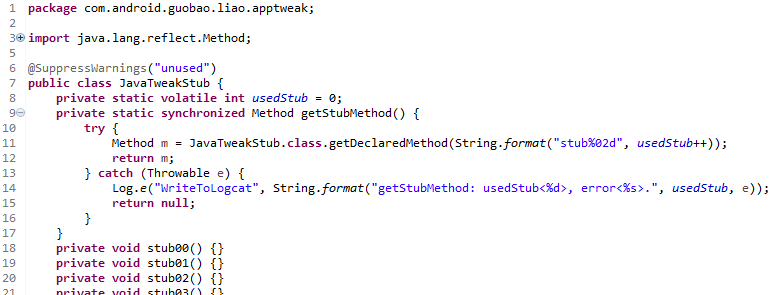
其中\_\_javahook\_initbridge\_\_函数中完成了JNI注册，dex加载，以及一些必要的全局变量设置等。

**二、Java层功能**

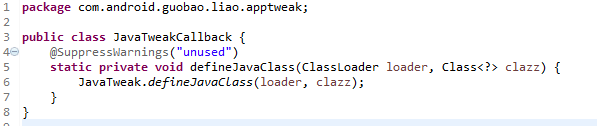
java层的所有代码全部放在com.android.guobao.liao.apptweak包中，如下图.



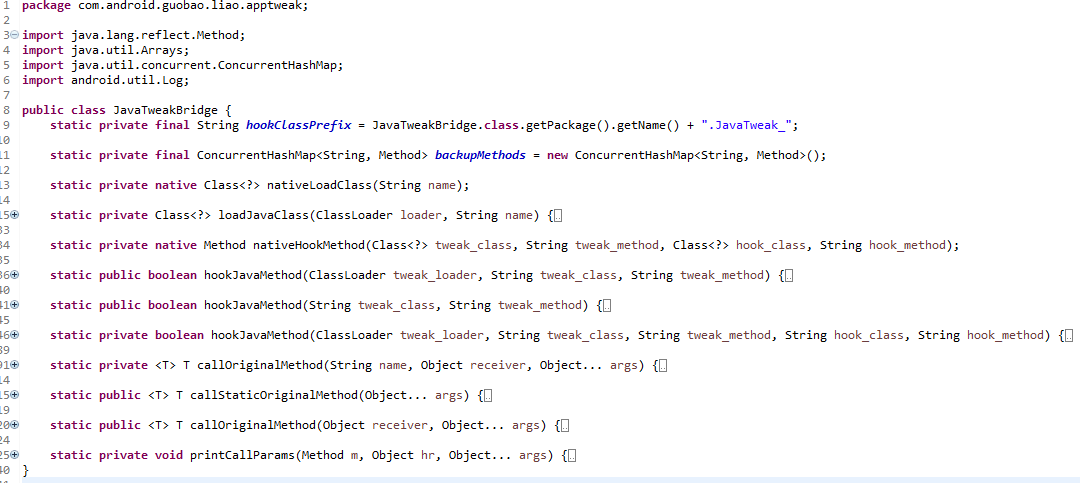
JavaTweakStub.java，主要是一些空方法存根，为了解决7.0及以后系统不再允许动态创建方法的问题，7.0以前此类没用。存根方法都是私有方法，没有传入传出参数，方法实现为空，也不会被调用，只是占据一个方法位置，预置100个存根。其中的getStubMethod方法是私有的，只会在Native中被调用，用于创建备份方法。



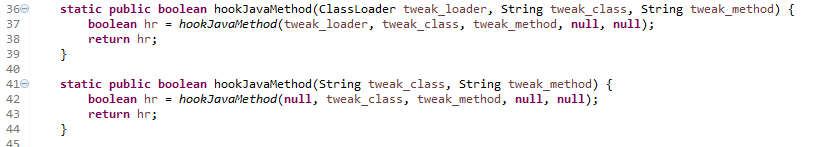
JavaTweakCallback.java，是一个回调类，实现如下。



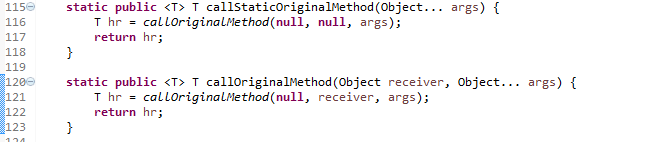
只有一个静态私有方法，每当有新类加载时会被Native层调用，方法实现很简单，简单调用了一下JavaTweak类中的同名函数。JavaTweakBridge.java，此类负责Java和Native的交互，主要是对Native函数nativeHookMethod的封装，实现方法如下：



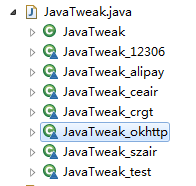
JavaTweakBridge类大多为私有方法，只提供了两组对外接口，hookJavaMethod和callOriginalMethod。hookJavaMethod函数让java层hook指定方法变的简单，只需要指定类名和方法名就可以完成hook.



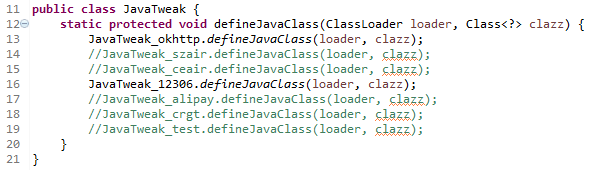
callOriginalMethod负责调用非静态的原始方法，callStaticOriginalMethod负责调用静态的原始方法.



JavaTweak.java，此类是具体的针对某个app做hook分析的业务类。其中包含一个公有类JavaTweak，以及多个以JavaTweak\_开头的针对某个app编写的内部类，如下图所示。

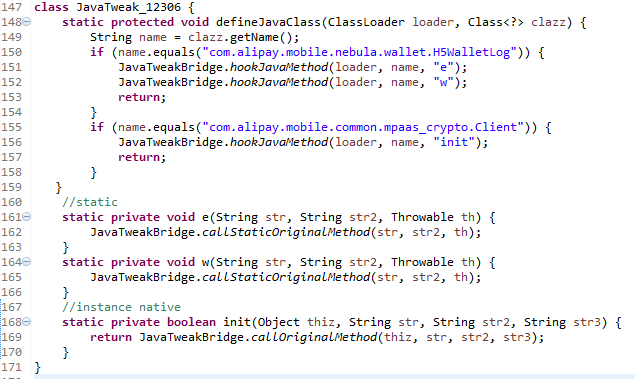


JavaTweak.类实现如下。



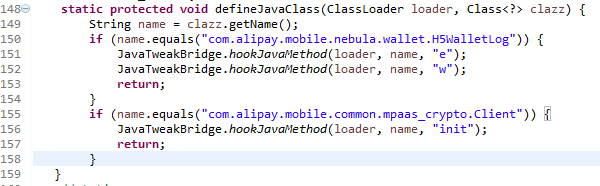
此类只有一个defineJavaClass静态方法，根据需要简单调用指定的内部类的defineJavaClass方法。

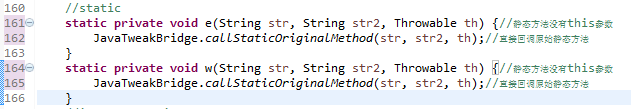
下面以JavaTweak\_12306类为例，看下如何hook一个java方法，注意内部类必须以JavaTweak\_开头, JavaTweak\_12306类实现如下：



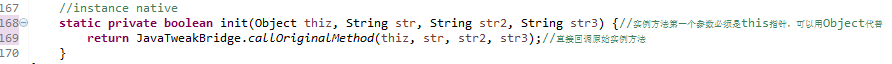
类中对app中的一个日志输出类com.alipay.mobile.nebula.wallet.H5WalletLog中的w、e方法进行了hook。

类中对app中的一个数据加解密类com.alipay.mobile.common.mpaas\_crypto.Client中的init方法进行了hook。

由于w、e方法都是静态方法，所以我们的hook方法和原方法参数必须完全一致



而init方法是实例方法，所以我们的hook方法和原方法参数相比会多一个this参数，其它参数依次后移。



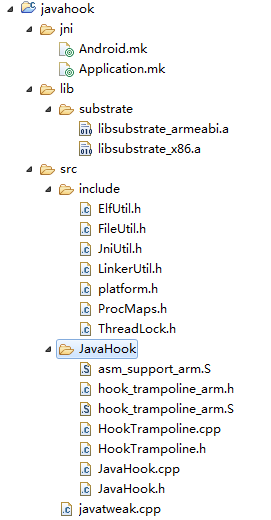
callOriginalMethod内部默认会打印传入传出参数日志，所以这里的hook方法都是直接回调原始方法。 当然可以加入任何其它代码，甚至直接返回。

上面就是Hook框架所有java层的类介绍，可以看出的java层代码也非常简洁易用，类似于xposed框架的hook功能基本具备。相比较而言hook的代码量更少，对于逆向分析来说已经大体够用，关键是完全脱离xposed的依赖。

**JavaTweak Hook框架使用**

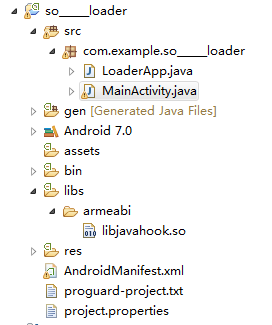
**一、编译Native库**

库目录结构以及Android.mk编译文件如下图所示。编译前必须安装好android-ndk，进入javahook目录，运行android-ndk中的ndk-build命令即可，编译完成后会生成libjavahook.so。

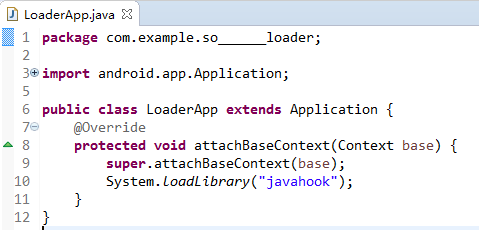


**二、创建Android测试工程**

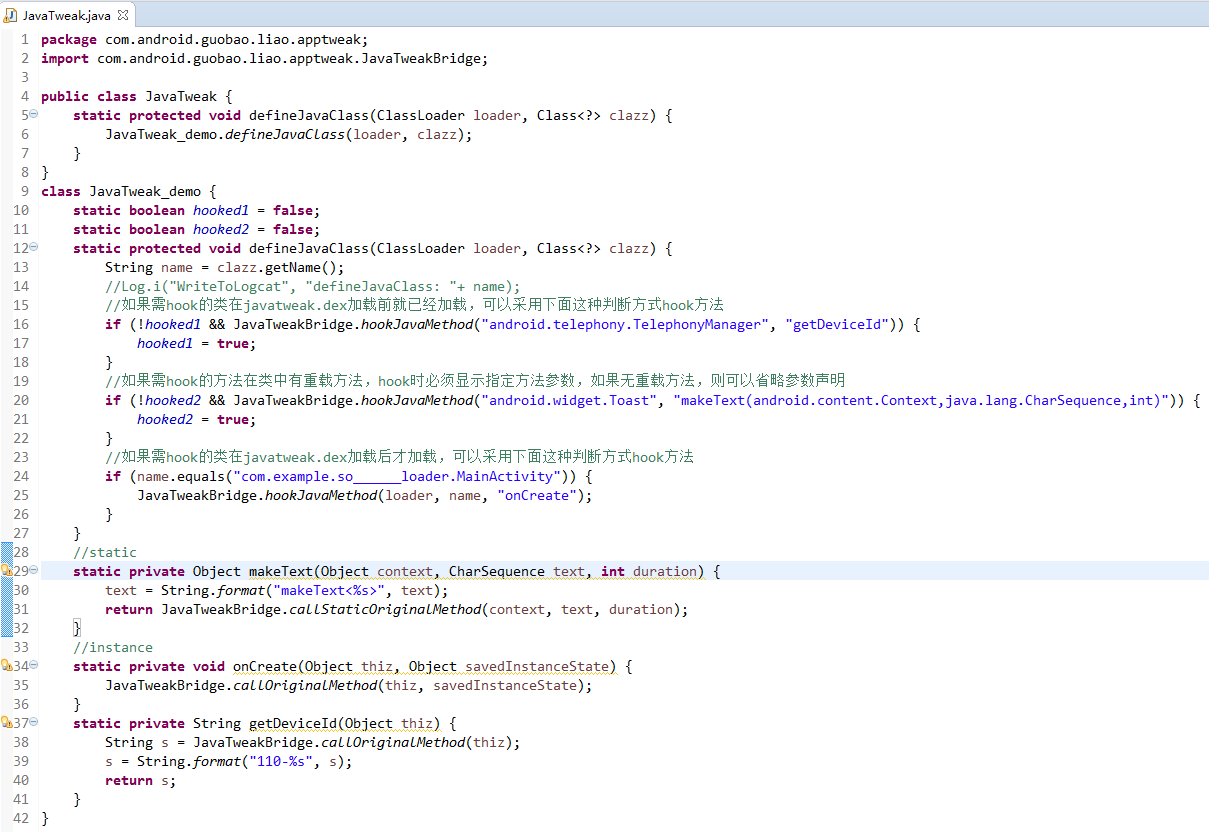
用AndroidStudio或Eclipse新建一个测试工程，工程名字任意，如下图所示。将上一步编译好的libjavahook.so库拷贝到libs/armeabi目录下。UI界面上放置两个按钮，分别用来测试static方法和instance方法hook测试工程需要开启【获取手机识别码】权限，请在手机【设置】中开启。



LoaderApp.java和MainActivity.java中代码如下所示。



**三、Hook代码添加**



**四、Hook说明**

在JavaTweak\_demo内部类中测试Hook了app中的三个方法。MainActivity类的onCreate方法是instance方法，在app启动的时候加载。Toast类的makeText方法是static方法，是一个系统类，在javatweak.dex加载前已经加载完成，且是一个重载方法。

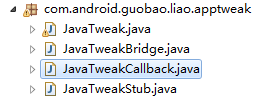
TelephonyManager类的getDeviceId方法是instance方法，是一个系统类，在javatweak.dex加载前已经加载完成。

大多数的系统类，都在javatweak.dex加载前就已经加载好了，所以只要进入defineJavaClass函数中即可开始hook。而自定义的类，大多是在javatweak.dex加载后才会加载，所以需要等待类加载的回调再hook。

我们自己的实现方法必须是static方法，我们可以在方法中更改传入传出参数，可以调用也可以不调用原始方法。内部类的类名必须为JavaTweak\_xxx格式（简化调用采取的约定）。自定义方法必须和hook方法的名称一致（简化调用采取的约定）。

**五、编译javatweak.dex**

javatweak.dex由下图所示的几个java文件编译而成，不依赖demo工程。JavaTweak.java中的Hook代码，根据不同的HOOK需求而定。其它几个文件的功能和内容上面已经介绍，且无需更改。

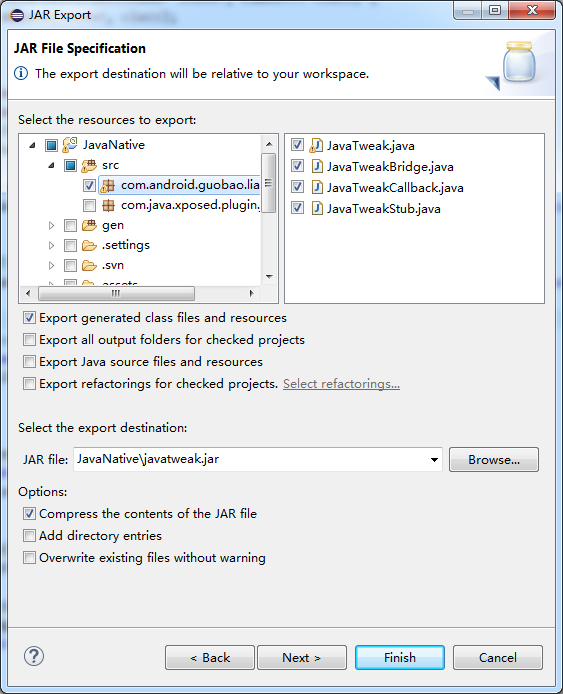


先将java文件导出为javatweak.jar，如下图，

再将jar 文件转化为javatweak.dex，命令如下，

dx --dex --output=javatweak.dex javatweak.jar

其中dx为【adroid-sdk\build-tools\版本\】目录下的命令行工具。



**六、运行效果**

将javatweak.dex放置在手机的/data/local/tmp目录下（固定放在此目录）。

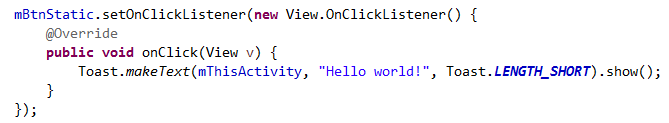
【adb push javatweak.dex /data/local/tmp】将demo程序安装到手机上。

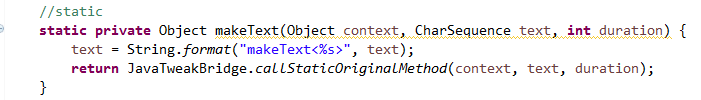
【adb install demo.apk】运行程序后，会在LogCat窗口中看到类似如下的日志。

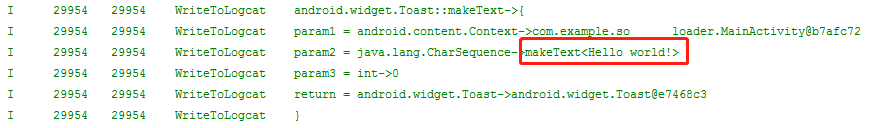
onCreate函数在程序启动时会被调用，最后几条日志输出的是函数调用时的输入输出参数类型和值。



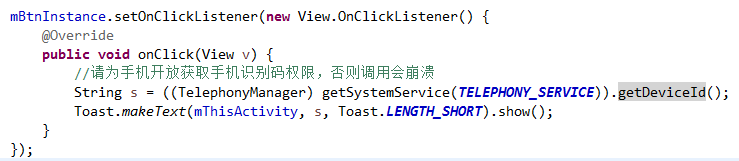
当点击静态方法测试按钮时，会在LogCat窗口中看到类似如下的日志。从弹出的提示框上可以看到，我们的弹框内容已经被更改了。

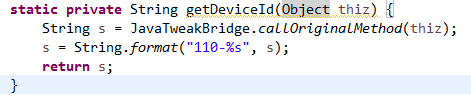


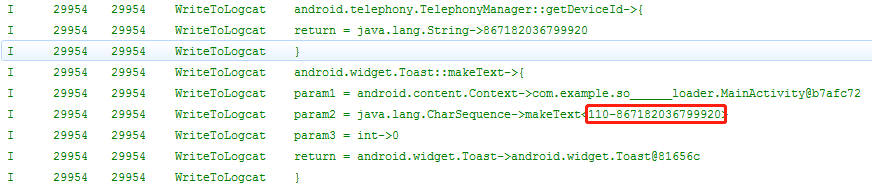




当点击实例方法测试按钮时，会在LogCat窗口中看到类似如下的日志。从弹出的提示框上可以看到，我们获取到的设备id，被自动加上了110数据。







**JavaTweak Hook总结分析**

**一、框架不足**

目前框架在从5.0到8.1的真机上测试通过，9.0及以上手机未测试，主要的问题应该是动态库符号查找的问题，后续会修复更新。hook框架只是用来辅助逆向开发或研究之用，在hook个别方法上可能会失败（具体表现是app卡死或崩溃），可以尝试延迟hook。

目前不支持构造函数hook（未提供java层的hook接口）。demo演示的是进程内hook，跨进程hook要将libjavahook.so注入到指定app中，但框架本身并没有进程注入功能。方法hook过程并没有考虑的很全面，可能会有细节方面的bug。文档中的源代码截图只供参考，完整源代码以及demo请从GitHub下载

**<https://github.com/liaoguobao/javahook>**

**二、模块化设计**

在实际的使用过程中，由于缺少注入功能，JavaTweak框架并不会独立应用，而是会以模块形式嵌入到更高一层的AndroidTweak系统中。这套系统提供了最基本的系统级的注入功能以及各种辅助逆向开发的模块，类似于一个小型的Android版本的Cydia。每个模块可以被独立注入又可以相互配合同时注入。

AndroidTweak框架设计及JavaTweak所处位置如下图。

