# XPOSED

## Java正常执行调用逻辑

dvmCallVoidMethod

dvmCallMethod

dvmCallMethodV

dvmInterpret

dvmMterpStd

dvmMterpStdRun

循环

HANDLE\_OPCODE(OP\_INVOKE\_STATIC)

GOTO\_invoke(invokeStatic)

GOTO\_invokeMethod

dvmMterp\_invokeMethod

GOTO\_TARGET(invokeMethod)

Pc=methodToCall->insns/methodToCall->nativeFunc

用反射的调用逻辑：

Constructor getDeclaredConstructor = clazz. getDeclaredConstructor()

Method m = clazz.getDeclaredMethod()

m.Invoke()

---->

Constructor. newInstance=>

Dalvik\_java\_lang\_reflect\_Constructor\_constructNative =>

dvmInvokeMethod =>

method->nativeFunc/dvmInterpret insns

Method.invoke =>

Method.invokeNative =>

Dalvik\_java\_lang\_reflect\_Method\_invokeNative =>

dvmInvokeMethod =>

method->nativeFunc/dvmInterpret

nativeFunc => dvmResolveNativeMethod

dfunc = dvmLookupInternalNativeMethod

dfunc()

GOTO\_TARGET(invokeMethod, bool methodCallRange, const Method\* \_methodToCall, u2 count, u2 regs)

Hook方式：

1. 静态修改apk中的函数调用，插入语句(droidbox)
2. 用反射获取java函数Method类型对应c层结构Method\*，修改insns域的dex字节码(未发现)
3. 修改method->nativeFunc域，自己实现dvmResolveNativeMethod以重新映射(xposed/substrate)

## 原始系统启动过程



## ART虚拟机和DALVIK虚拟机对比

1、Ahead-of-time (AOT) compilation instead of Just-in-time (JIT)

2、Improved garbage collection

3、Improved memory usage and reduce fragmentation

Ahead-of-time (AOT) compilation instead of Just-in-time (JIT)

在dalvik中(实际为android2.2以上引入的技术),如同其他大多数jvm一样,都采用的是jit来做及时翻译(动态翻译),将dex或odex中并排的dalvik code(或者叫smali指令集)运行态翻译成native code去执行.jit的引入使得dalvik提升了3~6倍的性能

而在art中,完全抛弃了dalvik的jit,使用了aot直接在安装时用dex2oat将其完全翻译成native code.这一技术的引入,使得虚拟机执行指令的速度又一重大提升

Improved garbage collection

## Xposed特点

1兼容性方面，支持selinux，对某些厂商做适配

2提供多种hook功能，可以hook系统api，应用app函数，资源操作，am pm等调用，插件扩展方便

3开源，不断更新，支持art虚拟机，android5.x/6.x

## Hook类型

Xpose 支持4种hook类型：package:robv.android.xposed

IXposedHookZygoteInit

在Zygote进程启动初始化时执行hook，framework/system

IXposedHookLoadPackage

在包加载时刻hook，此时可以hook用户app的函数

IXposedHookInitPackageResources

在资源操作时刻回调

IXposedHookCmdInit

在执行am,pm等时回调

de.rovb.android.xposed.XposedBridge.main XposedBridge.java

initForZygote:

findAndHookMethod handleBindApplication 在进程首次加载apk时hook

findAndHookMethod initAndLoop/systemMain system\_server启动时初始化hook

hookAllConstructors LoadedApk.class 在app进程中加载多个apk

loadModules:

IXposedHookZygoteInit initZygote()

IXposedHookLoadPackage 加入sLoadedPackageCallbacks表

IXposedHookCmdInit initCmdApp()

## Xposed文件结构

## Xposed => app\_process

Xposed\_service.cpp 提供文件操作接口(libc.so尚未加载)

Libxposed\_dalvik.cpp 提供dalvik虚拟机hook接口(jni层实现)

Libxposed\_art.cpp 提供art虚拟机hook接口(jni层实现)

Libxposed\_command.cpp提供其他通用接口

Xposed.cpp c++层一些实现，用于app\_main

App\_main.cpp app\_process入口，用于替换app\_process(重写main和AppRuntime) api>=21

App\_main2.cpp app\_process入口，用于替换app\_process(重写main和AppRuntime) api<21

Xposedbridge => xposedbridge.jar

XposedHelpers.java 提供反射能力

XposedBridge.java xposedbridge入口，代替系统ZygoteInit类，主要的hook逻辑

XC\_MethodHook.java 为插件提供hook接口(java层实现)

问题1：何时加载framework.jar?

ActivityThread.class => const-class ActivityThread => OP\_CONST\_CLASS.cpp

dvmResolveClass

dvmFindClassNoInit

findClassFromLoaderNoInit

dvmCallMethod(loadClass)

dvmFindSystemClassNoInit

findClassNoInit

searchBootPathForClass

java.boot.class.path bootClassPath

/system/framework/core.jar:/system/framework/ext.jar:/system/framework/framework.jar:/system/framework/android.policy.jar:/system/framework/services.jar

问题2：为何IXposedHookZygoteInit可以hook framework/system api

XposedBridge.main

initForZygote

findAndHookMethod 加载系统jar

loadModules

loadModule

moduleInstance.initZygote()

hookLoadPackage(moduleInstance)

问题3：为何IXposedHookLoadPackage可以hook app自身函数

首次加载：

initForZygote beforeHookedMethod handleBindApplication

Process.start “android.app.ActivityThread”

android.app.ActivityThread.main

android.app.ActivityThread.attach

ActivityManagerService. attachApplication

ActivityManagerService. attachApplicationLocked

android.app.ActivityThread.bindApplication

handleBindApplication

ActivityStack.realStartActivityLocked

scheduleLaunchActivity

getPackageInfoNoCheck

getPackageInfoNoCheck

LoadedApk()构造

LoadedApk.getClassLoader

ApplicationLoaders.getDefault().getClassLoader(zip,libpath,null)

问题4：何时加载xposedbridge?

Xposed app\_main->main():

xposed::initialize() -> env中加入xposedbridge.jar

runtime.start(“de.robv.android.xposed.XposedBridge”) -> AndroidRuntime

startVm()

onVmCreated()

xposed::onVmCreated 加载xposedbridge.jar

startReg()

XposedBridge.main()

问题5：底层实现hook？

强制吧函数设置为native，并修改native函数使其返调java函数 libxposed\_dalvik.cpp

hookInfo->reflectedMethod = dvmDecodeIndirectRef(dvmThreadSelf(), env->NewGlobalRef(reflectedMethodIndirect));

hookInfo->additionalInfo = dvmDecodeIndirectRef(dvmThreadSelf(), env->NewGlobalRef(additionalInfoIndirect));

SET\_METHOD\_FLAG(method, ACC\_NATIVE);//设置Method->AccessFlag强制为native函数

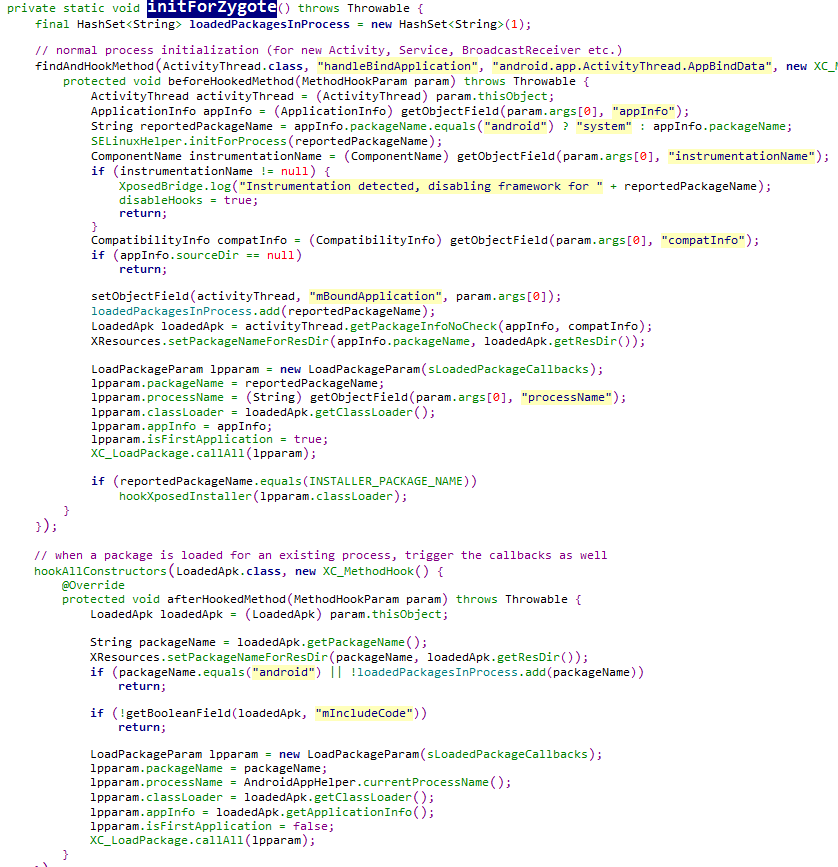
method->nativeFunc = &hookedMethodCallback;//修改默认回调dvmResolveNativeMethod为自定义函数，该函数原先从系统函数和so中的jni函数中寻找java对应的c层方法，hookedMethodCallback函数则调用dvmCallMethod执行java层方法

method->insns = (const u2\*) hookInfo;//该域原用于非native模式下保存dex字节码用于解释执行，现用于存储Method指针

method->registersSize = method->insSize;

method->outsSize = 0;











# SUBSTRATE

特点：

1 hook时机早，因此支持hook so函数

2重定向liblog.so，较易适配

3支持ios、android平台

4不支持dalvik，不支持selinux，不支持api>=5.0

文件：

substrate.h //c++ header file used in JNI layer hook

substrate-api.jar //import package used in java layer hook

substrate-bless.jar //used to remove properties(private,protect,etc...) in java layer hook

com.saurik.substrate.apk//host apk, we can only develop plugin for it to install package

\lib\armeabi \lib\x86 //real operation for hooking

libAndroidBootstrap0.so //used to fake /system/lib/liblog.so and pull up libAndroidLoader.so

libAndroidLoader.so //used to pull all \*.cy.so

//MSLoadExtensions

libAndroidCydia.cy.so //still in research

libDalvikLoader.cy.so //still in research

libsubstrate.so //provide jni layer hook low-level api

//MSFindSymbol MSGetImageByName MSCloseFunction MSDebug MSHookFunction

libsubstrate-dvm.so //provide java layer hook low-level api

//MSDecodeIndirectReference MSJavaHookClassLoad MSJavaHookBridge MSJavaHookMethod

// MSJavaCreateObjectKey MSJavaReleaseObjectKey MSJavaGetObjectKey MSJavaSetObjectKey MSJavaBlessClassLoader

libSubstrateJNI.so //used by substrate.apk to do c++ layer work

//getppid readlink grep unlink symlink mkdir kill chown chmod

libSubstrateRun.so //used by substrate.apk to do patch/unpatch/link/unlink operation

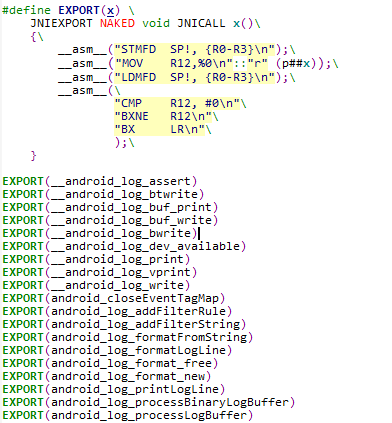
//patch unpatch link unlink nm rpl

update-binary.so //used by substrate.apk to recover patch/link operation

## Hook过程















## Xposed框架介绍

Xposed框架分为xposed版app\_process和XposedBridge.jar两部分。app\_process就是zygote，我们先看看xposed版的zygote干了些什么。

### 4.1  Xposed版zygote

注意，本章只分析32位，dalvik版的xposed app\_process，其入口main函数位于app\_main.cpp里。

图14所示的代码展示了Xposed版zygote与众不同之处。

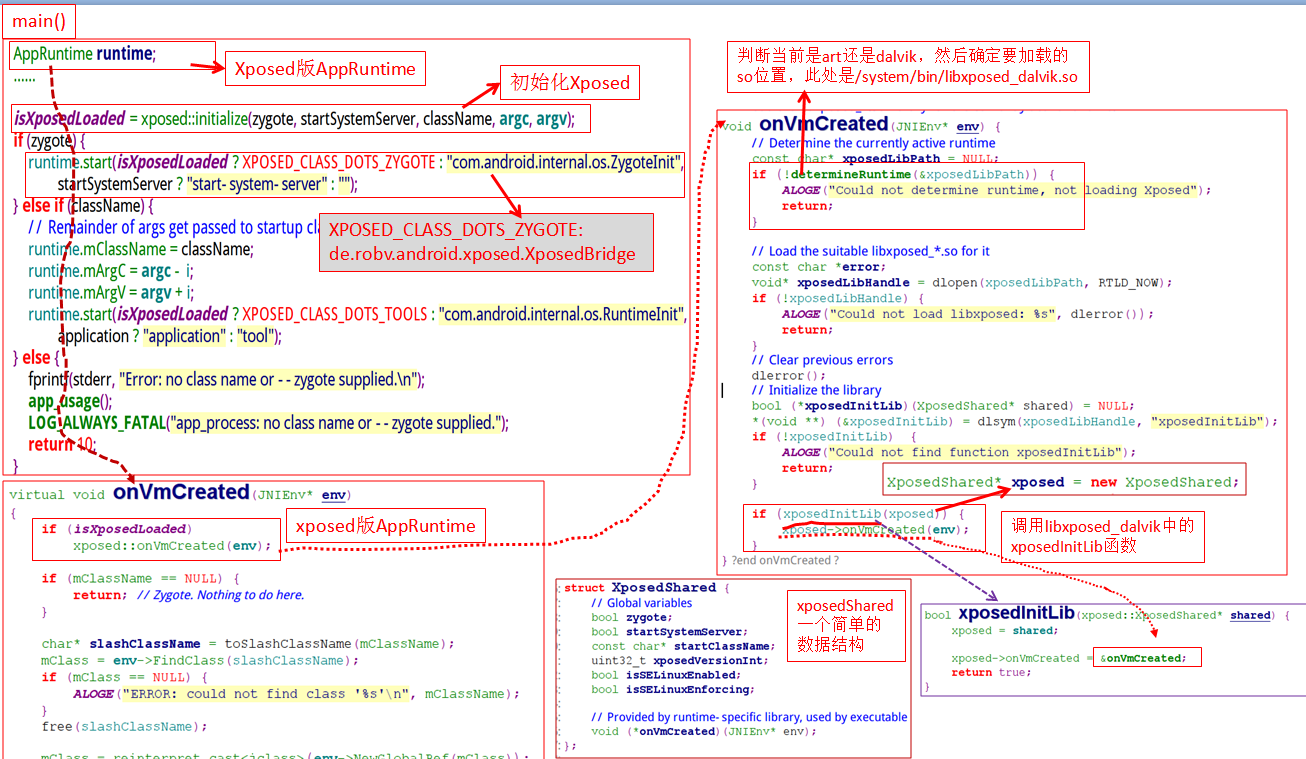


图14中，左上角的框是app\_main的main函数，里边有两处不同之处：

* AppRuntime：此处的AppRuntime类是xposed版的AppRuntime。它的与众不同之处从左下图的onVmCreated开始。当检查到isXposedLoaded为真是时，将调用xposed::onVmCreated函数。
* xposed::onVmCreated函数位于右上框，它先判断当前虚拟机是ART还是dalvik，然后加载xposed一个特殊so，此处是/system/bin/libxposed\_dalvik.so。注意，这里的xposed框架和XposedInstaller略有区别。因为XposedInstaller比较旧了（也就支持sdk 16的样子），还没有涉及到ART和dalvik的区别。在onVmCreate中，它将调用libxposed\_dalvik.so中的xposedInitLib函数，然后再调用so中设置的onVmCreated函数。这个onVmCreated函数由xposedInitLib设置。
* 左上框中还有一个xposed:initialized函数。这个函数初始化了XposedShared类型的全局变量xposed，同时还把一个重要的jar包加到了CLASSPATH里。来看代码。如图15所示：

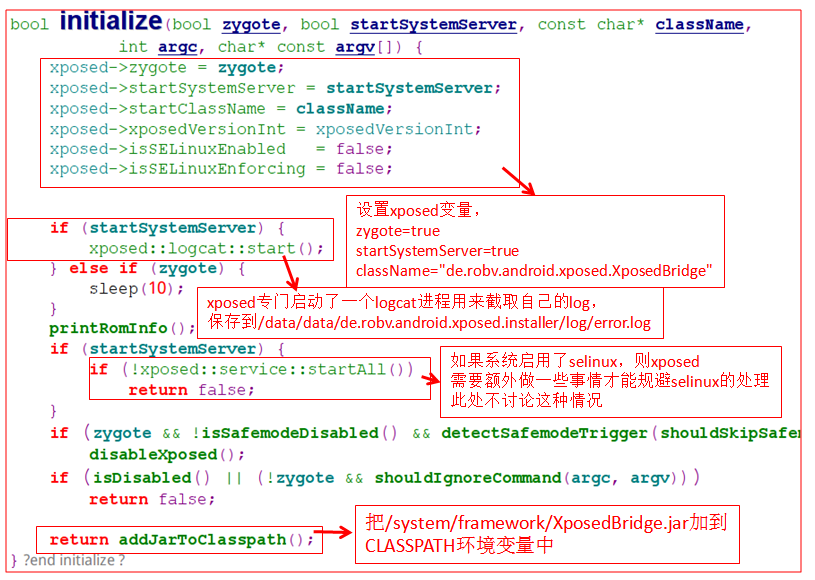


图9展示了initialize函数的内容，主要是最后一个把/system/bin/XposedBridge.jar（这个jar包的位置和我们在XposedInstaller那里看到得不同，原因前面解释过了）加到CLASSPATH比较重要。

注意，我们这里虽然对initialize介绍的内容很少，但实际上这个函数要真正看明白还是很需要技术实力的：

1  logcat:start：里边fork了logcat进程用来存储xposed自己的log

2  service:startAll：为了完美支持selinux，这里的处理更是很有技巧。selinux是一个完整的知识体系，想彻底掌握它的童鞋请参考我的三部曲文章[《深入理解SELinux SEAndroid》](http://blog.csdn.net/innost/article/details/19299937)

1&2其实很真实得反映出xposed的作者在Android、Linux上水平很高，经验很丰富。

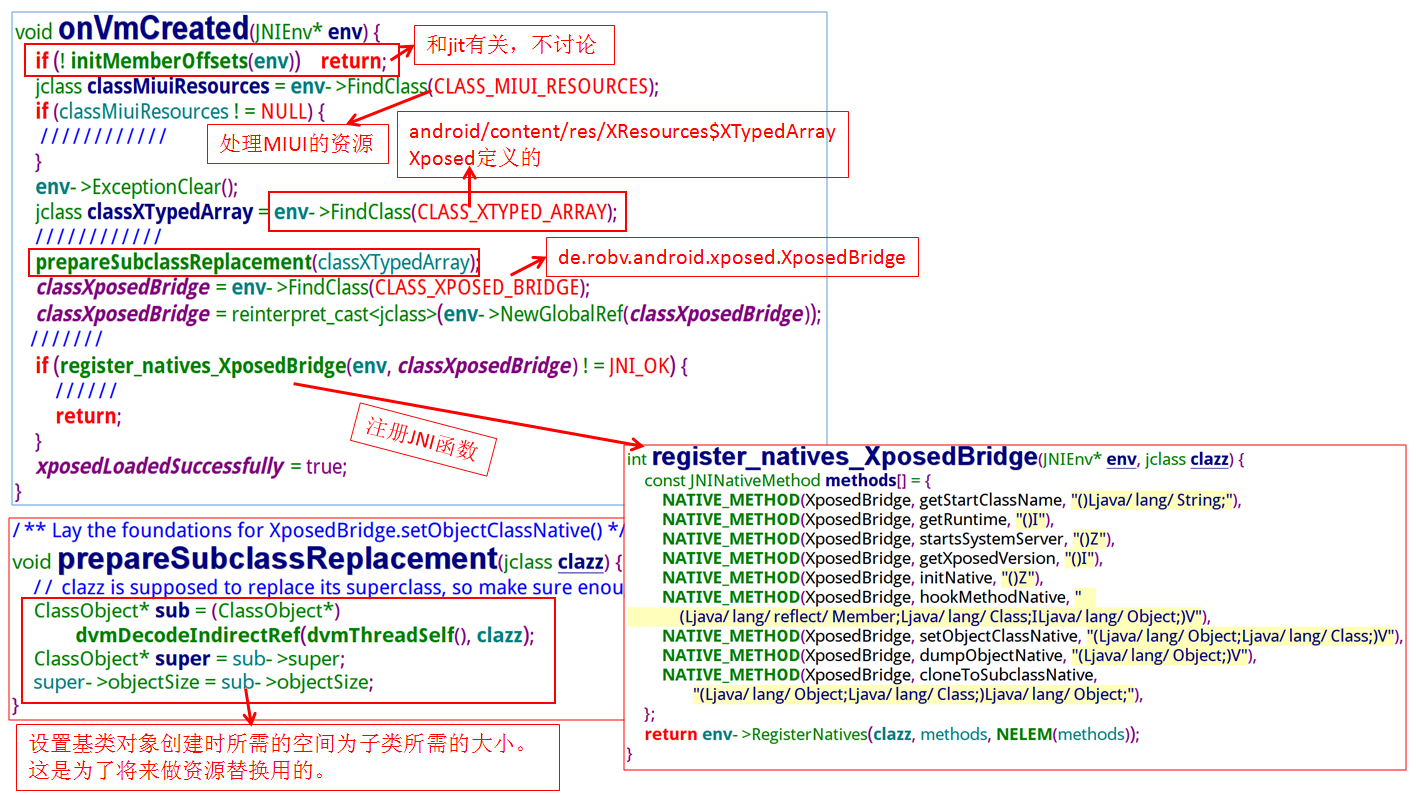
最后，如果xposed框架启用成功，那么zygote的入口类将由以前的com.android.internal.os.ZygoteInit变成de.robv.android.xposed.XposedBridge。

下面我们按照执行流程，把相关函数分析一遍：

* AppRuntime的onVmCreated，它最终会导致libxposed\_dalvik.so中的onVmCreated被调用。我们直接分析最后这个onVmCreated。
* 然后AppRuntime里会调用de.robv.android.xposed.XposedBridge.main函数。

### 4.2  onVmCreated

图16为代码：



onVmCreated比较简单了：

* 针对android/content/res/XResource&XTypedArray进行了一些处理，这是为将来资源替换时候做准备。以后我们碰到再说。
* 为XposedBridge注册JNI函数

xposed官方说MIUI大量用了xposed的东西，并且不共享，以后碰到MIUI的问题他们不再支持。Don't know what to say....

### 4.3  XposdBridge.main函数

main函数代码如图17所示：



main函数里有三个重要函数：

* initNative：好好学习下
* initForZygote：好好学习下
* loadModules：加载App插件

#### 4.3.1  initNative

initNative很重要，来看代码，如图18所示：



图18中，我们重点看一下callback\_XposedBridge\_initNative和register\_natives\_XResources这两个函数。这两个函数比较简单，我们统一放到图19中：



不多说了，没什么难度。

#### 4.3.2  initForZygote

从这个函数开始，xposed就开始给系统一些关键函数挂钩子了。我们看看它怎么玩儿的。代码如图20所示：

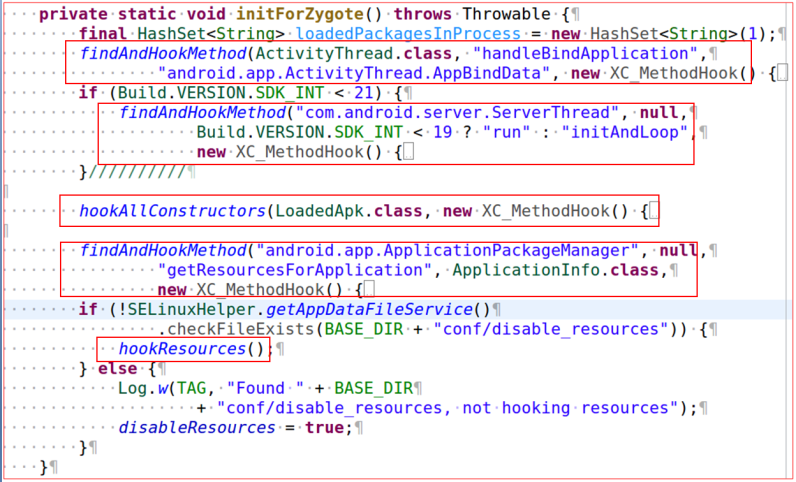
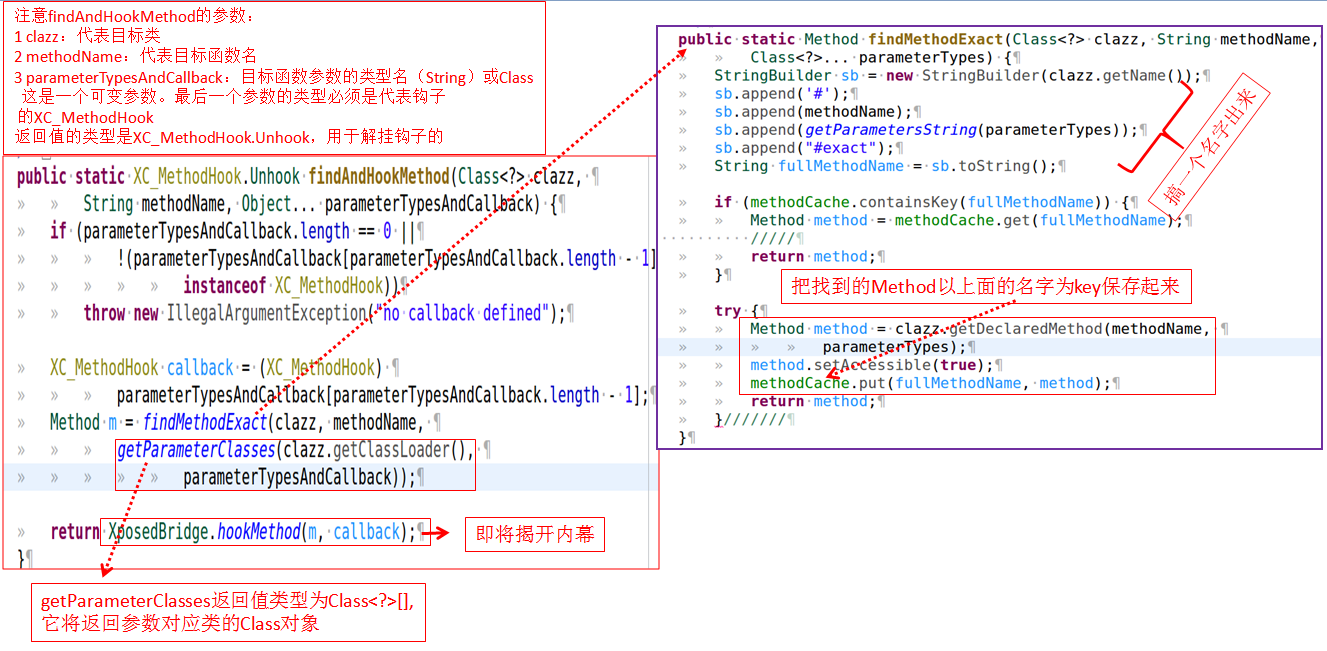


图20中我在eclipse里用了代码缩略显示的方法，可知一共有五个框，分别hook了一些关键内容。我们从上到下一次分析。先来分析下Xposed框架提供的挂钩函数findAndHookMethod。

##### （1）  findAndHookMethod介绍

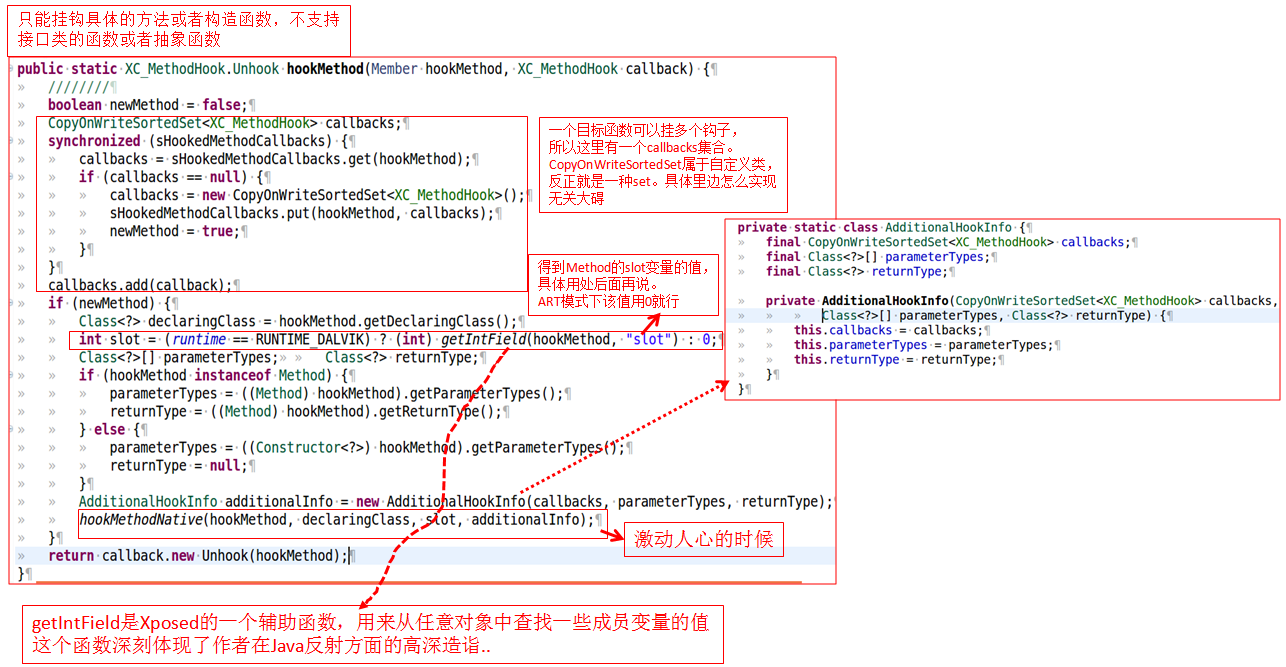
findAndHookMethod用来对指定类的指定函数进行挂钩。这个函数很重要，开发插件APP时用得最多。来看它的代码，如图21所示：



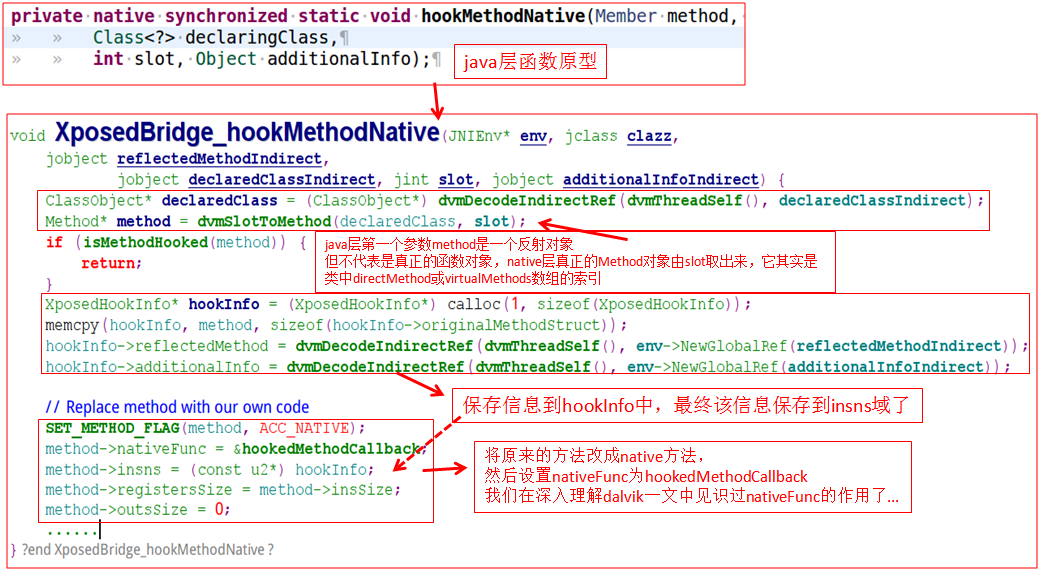
findAndHookMethod代码Java层面的逻辑还是比较好理解的：

* 首先是通过class相关的函数找到指定的函数对象。这里的查找需要指定函数所在的类，函数名，函数参数信息等，属于精确（exact）查找。注意，findAndHookMethod函数的最后一个参数必须是XC\_MethodHook类型，即钩子对象的数据类型。
* 然后就是XposedBridge.hookMethod。

来看hookMethod，如图22所示：



由hookMethod可知，一个目标函数可以挂多个钩子，这些钩子由一个集合来存储。然后我们将转到JNI层去看看hookMethodNative干了什么事情。这才是hook的核心。代码如图23所示：



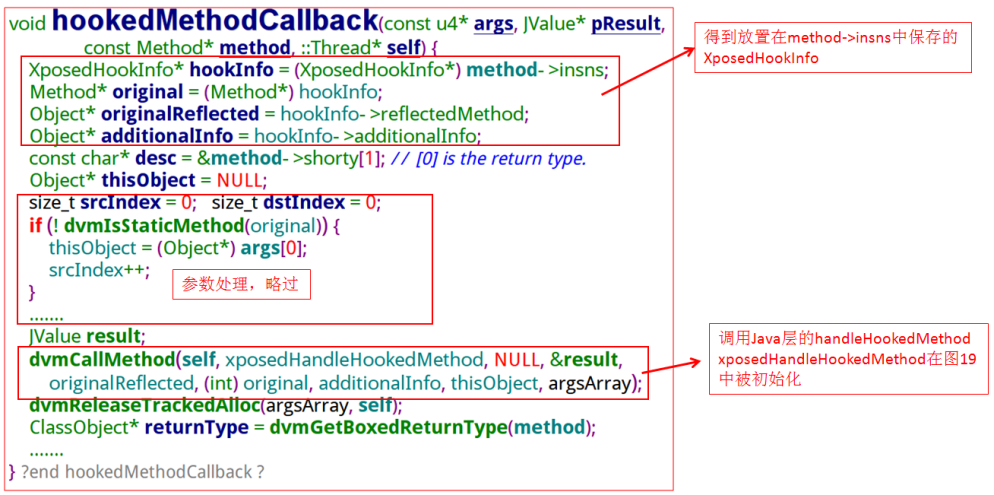
hookMethodNative完成了真正的挂钩处理，其思想很简单：

* 先找到目标函数在native层对应的Method对象。
* 修改这个Method为native方法，并设置nativeFunc为hookedMethodCallback函数。
* 然后还要保存原函数的一些信息到insns域。

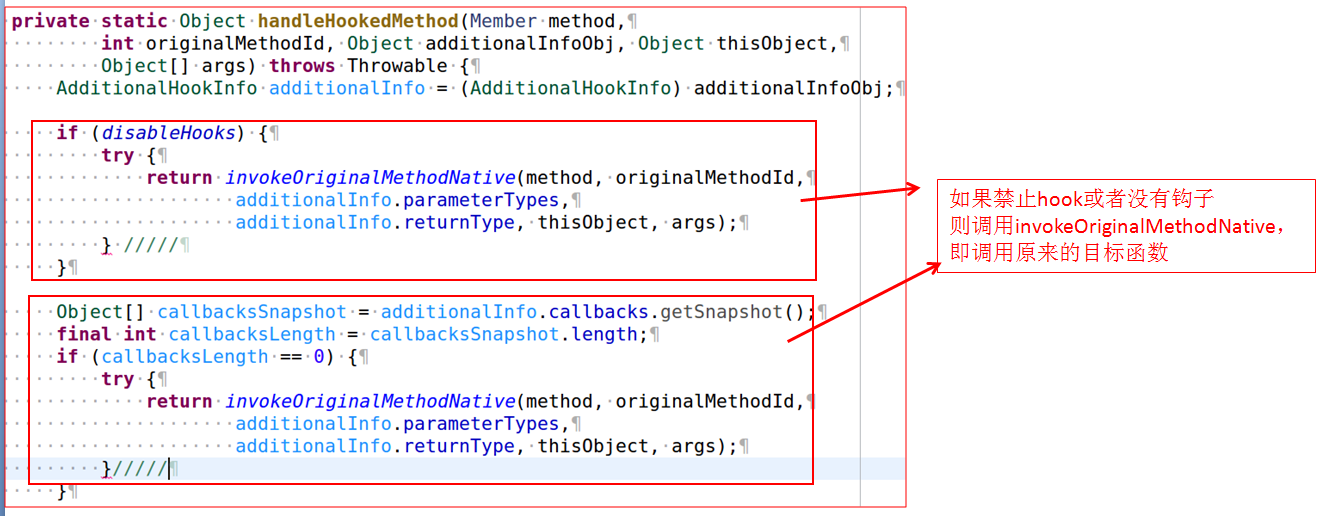
我们在[《深入理解Android之dalvik》](http://blog.csdn.net/innost/article/details/50377905)一文中介绍过，JVM调用java函数时候，发现这个函数为native的话，就调用它的nativeFunc。下面我们看看钩子函数的调用。

##### （2）  调用钩子函数

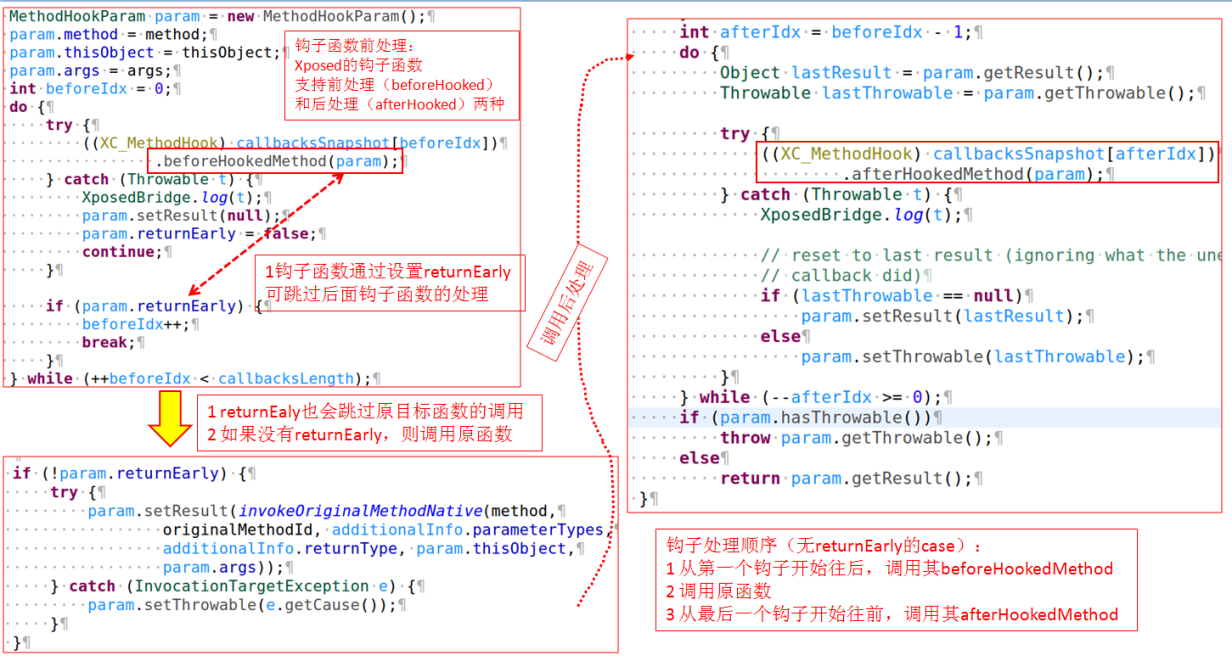
hook钩子函数后我们就要调用它。上一节我们发现xposed在挂钩子的时候会把原函数改造成native属性（即Dalvik会按native函数的方式调用它），对应的nativeFunc是hookedMethodCallback，其代码如图24所示：



注意喔，我们现在已经在处理被挂钩函数的调用了喔....从JNI会进入到java层的钩子函数dispatch总入口，及handleHookedMethod，这个函数比较复杂，我们一段一段来看它。

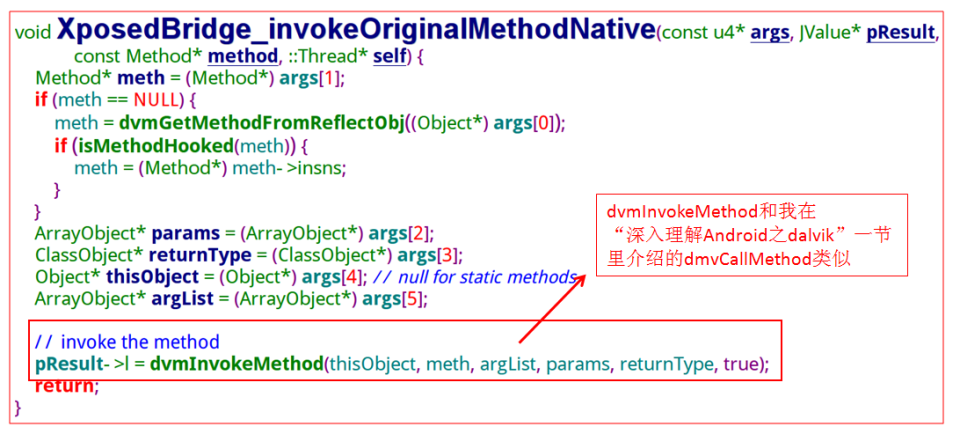


很简单。接着看第二段，如图26所示：



貌似也很简单喔...

现在来看原目标函数的调用，即invokeOriginalMethodNative。代码如图27所示：



同样很容易，不多说了。到此，我们已经看到了Xposed挂钩的所有过程。好像也没什么复杂的，只要对dalvik稍微属性点，应该是比较容易做的。

当然，本文只是给大家show了主要流程，真要自己动手做，我觉得难点在于参数传递等方面。anyway，了解了大体流程，后面的事情也好办，多尝试几次就好。

下面我们回过头来看initForZygote里加的几个钩子都干了什么。

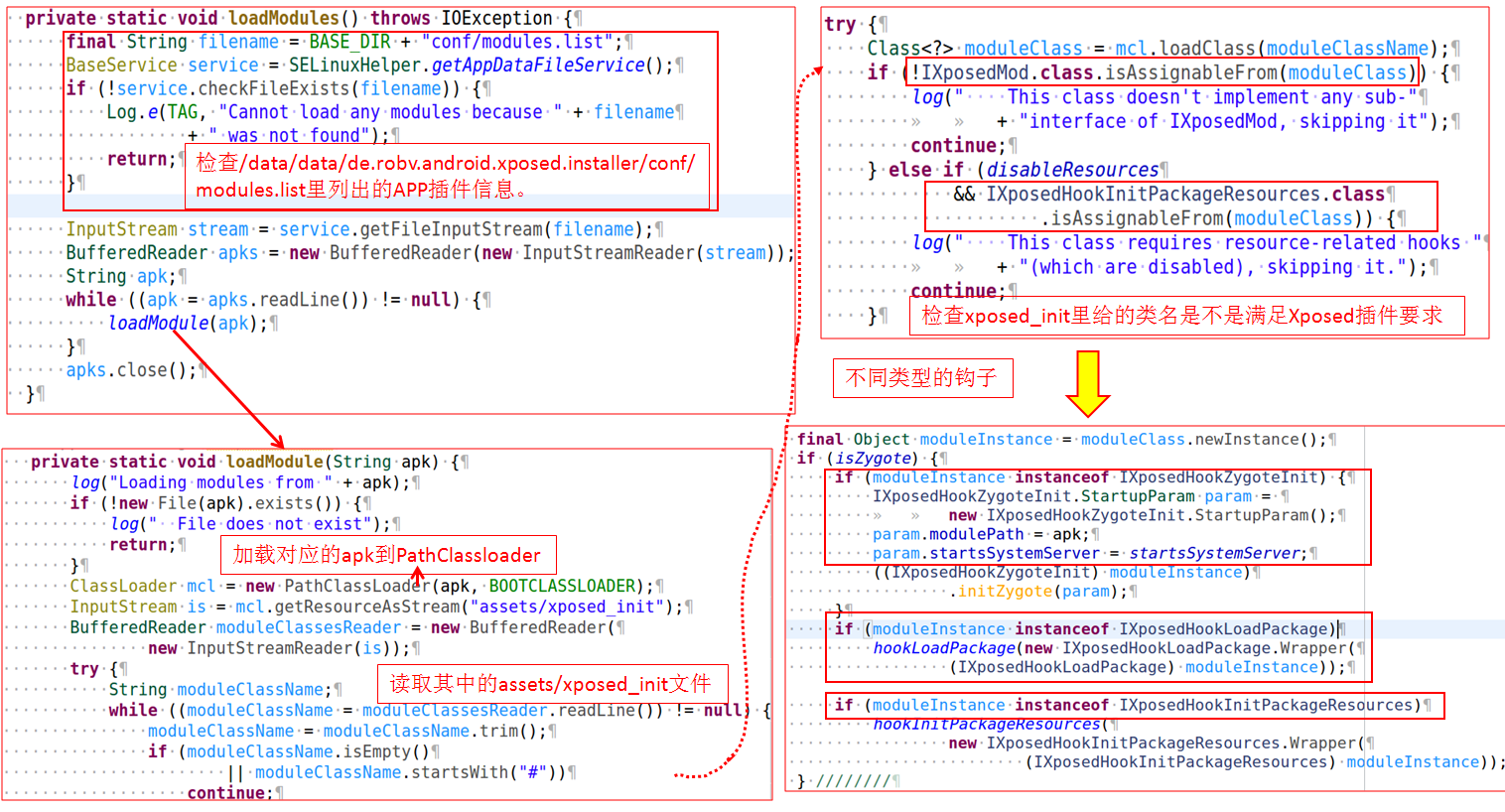
##### （3）  initForZygote钩子设置

图20的initForZygote代码示意中可知xposed对下面几个函数进行hook（此处先不讨论钩子函数干了什么）

* ActivityThread.handleBindApplication：这个函数是ActivityManagerService内部调用ActivityThread.bindApplication间接触发的。该函数的详情可参考《深入理解Android卷2》第六章[深入理解ActivityManagerService](http://blog.csdn.net/innost/article/details/47254381)的“ApplicationThread的bindApplication分析”一节。handleBindApplication主要工作是初始化APP（APP由zygote进程fork而来，在hanldeBindApplication之前，这个APP进程和zygote没什么区别。只有调用完handleBindApplication之后，这个APP进程才是APP,比如该进程有了对应的名字，Aplication对象被创建等）。
* com.android.server.ServerThread. initAndLoop函数：这个函数是给system\_server用的，用于启动系统各种重要服务。
* LoadedApk构造函数：通过hookAllConstructors来对LoadedApk各种构造函数进行挂钩。LoadedApk是APK文件在APP里的代表对象，里边有很多重要的信息。
* android.app.ApplicationPackageManager. getResourcesForApplication：挂钩PackageManager的资源获取函数。
* hookResources：对资源进行hook。这一块由于涉及到android APP对资源的管控，以后有机会我们再详细介绍。

#### 4.3.3  loadModules

main函数在initForZygote之后的下一个动作就是loadModules，这就是加载所有的插件APP。我们来看看这个函数，代码如图28所示：



我们来看下loadModules的处理：

* 先从XposedInstaller那获取当前已经安装（并启用）的插件APP列表，然后调用loadModule来加载这个APP。
* loadModule从APP的assets/xposed\_init文件里看看这个apk里声明了哪些钩子类。
* loadModule校验这些钩子类是不是符合Xposed定义的钩子类类型，然后作对应处理。图29列出了Xposed支持的钩子类类型。

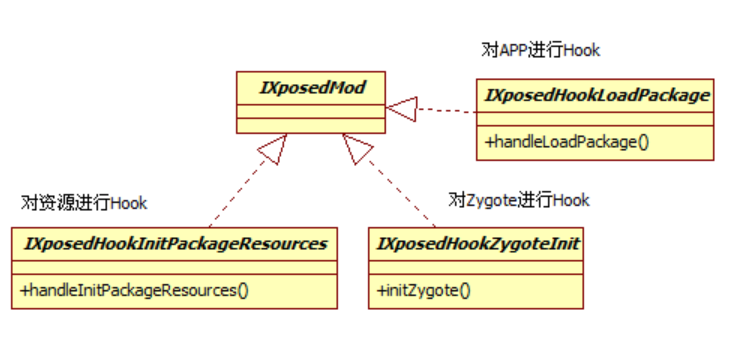
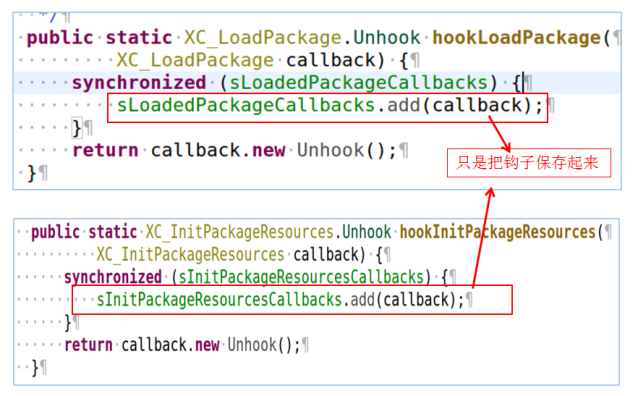


图30展示了hookLoadPackage和hookInitPackageResource两个函数的内容，特别简单。

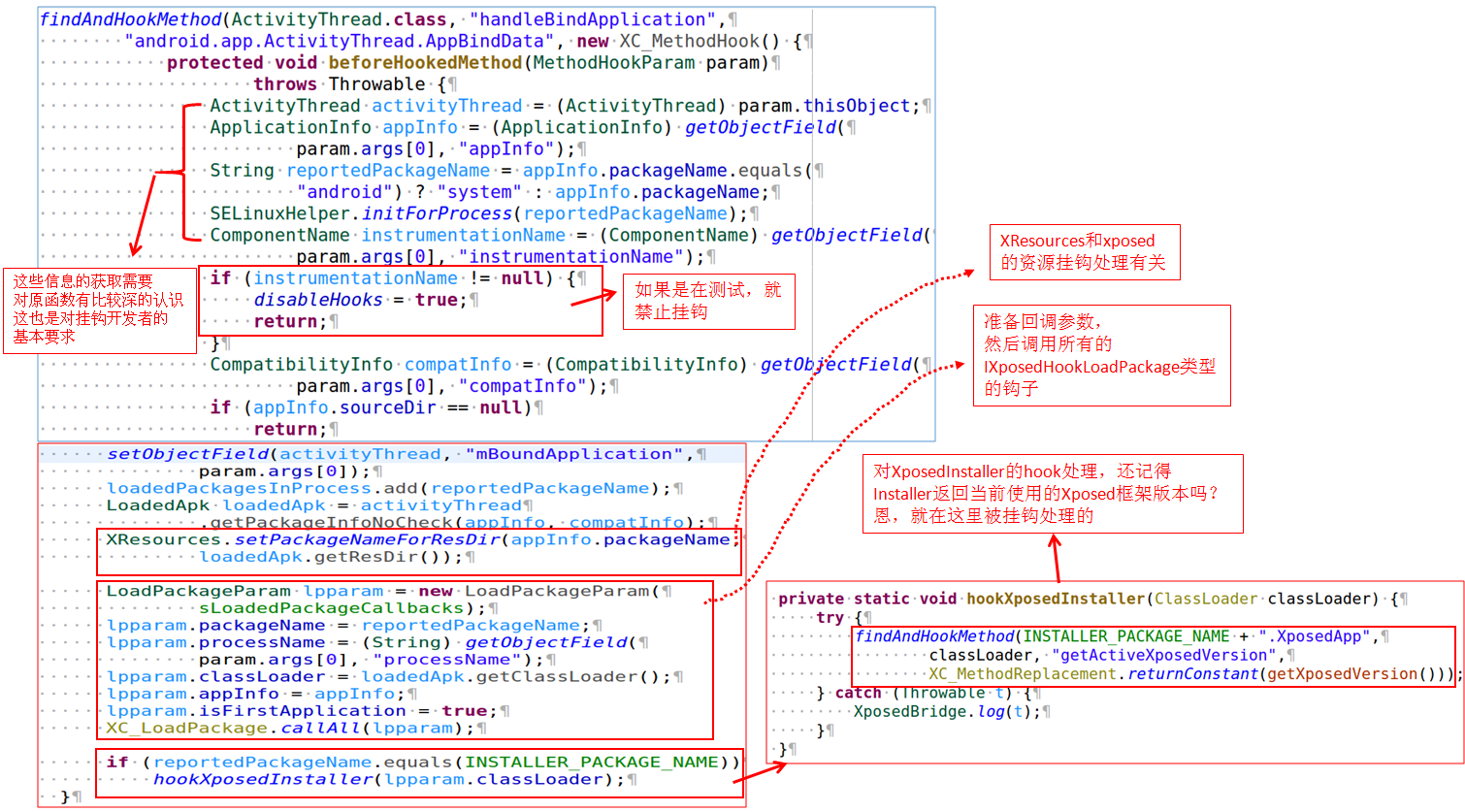


嗯嗯，就是把对应的钩子函数保存起来先。

好了，下面我们就来开始分析initForZygote里挂上的几个钩子分别有什么用。

#### 4.3.4  handleBindApplication钩子

initForZygote为hanldeBindApplication设置了前处理钩子，代码如图31所示：



前面说过，在APP生命周期内，handleBindApplication是APP刚准备好相关信息的一个重要点，在这个点去进行挂钩处理简直是最好不过了。当然，此处的挂钩处理也就是准备好这个APP的相关信息然后调用所有的IXposedHookLoadPackage类型的钩子。

注意，除了handleBindApplication之外，由于一个APP进程事实上可以加载多个APK（比如那些申明同样的uid和运行在同一进程的APP），在LoadedApk的构造函数中也做了类似的处理

IXposedHookLoadPackage钩子一般会干些什么呢？图31对XposedInstaller的处理就很明显了。一般而言，这种钩子会对目标APP中感兴趣的函数进行挂钩（调用findAndHookMethod），比如XposedInstaller对getActiveXposedVersion进行了挂钩，用于返回系统里正在使用的Xposed框架版本。

我们应该在钩子函数里干些什么？这是一个重要问题。我也不废话了，直接上XposedDemo的源码，如图32所示：

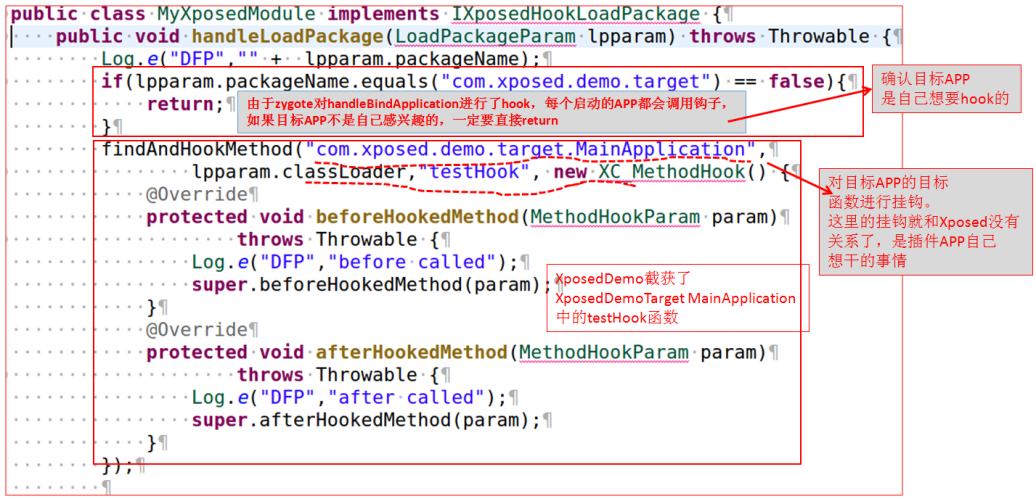


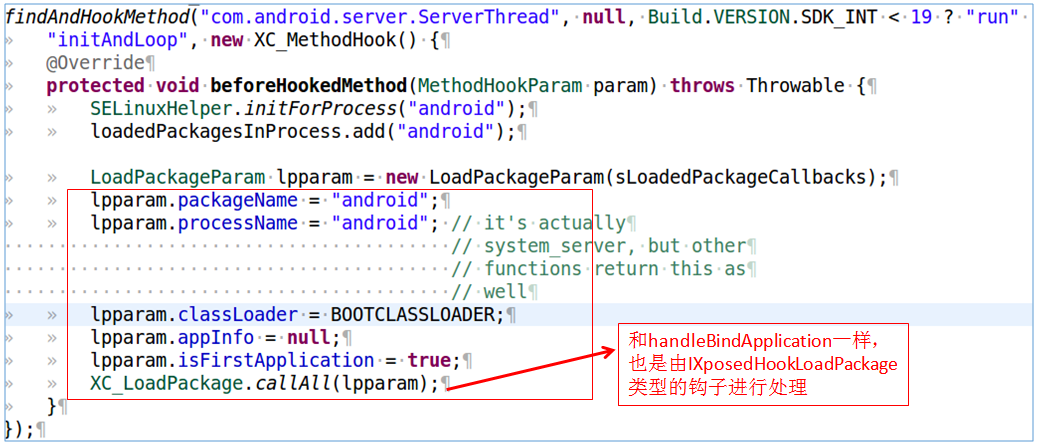
图32是我在xposed-learning项目中提供的XposedDemo示例，可知：

* Xposed框架对handleBindApplication进行hook后，当有APP启动的时候，该框架就会调用所有IXposedHookLoadPackage类型的钩子。
* 这种钩子一定要区分当前被hook的APP是不是自己想要Hook的APP。如果不是，请直接return。如果是自己的目标APP，则进一步通过findAndHookMethod进行挂钩。

简单点说，我们在IXposedHookLoadPackage的handleLoadPackage中把该挂的钩子都挂上就好。

#### 4.3.5  initAndLoop钩子

initAndLoop是system\_server进程的关键函数，在这个函数里Android Framework的绝大部分Service都将被创建。真是艺高人胆大，这个进程居然都提供了挂钩处理。其代码如图33所示：



代码倒是很简单，无非是针对system\_server进行hook。插件函数如果想区分被hook的进程是否为system\_server的话，只需要判断packageName是否为"android"即可。

再次强调，system\_server是Android Java层Framework的核心，要hook它需要万分小心，否则或导致手机系统出现会各种不稳定，崩溃，重启等情况。

下面我们介绍下Xposed框架对资源是怎么Hook的。

### 4.4  对资源的Hook

前面章节介绍了Xposed框架如何对代码调用逻辑进行hook。在Android APP中，除了代码逻辑外，Xposed还支持对资源进行Hook。对资源Hook的原理其实和对代码调用进行Hook的原理类似。这里我们简单介绍下Xposed框架如何对资源进行Hook。

在Android APP中，资源有三个重要类：

* ResourcesManager：一个APP进程有一个ResourcesManager。一个ResourcesManager可以管理多个Resouces。
* Resources：Resources是一个APK中资源的代表，注意，它不是指单独的一种类型的资源（比如字符串，图片等），它是一个APK文件中资源文件的代表。Resources对象有一个AssetManager，AssetManager能操作APK文件（其实就是一个压缩包）assets以及res目录里的内容。
* Resources通过ResourcesKey将自己保存到ResourceManager中的一个ArrayMap里。
* 对于资源挂钩来说，Xposed框架及插件APP做如下几件重要的事情：
* 和handleBindApplication类似，在APP进程某个时间点的时候（猜测：初始化资源相关模块）进行Hook，然后调用IXposedHookInitPackageResources类型的钩子。
* IXposedHookInitPackageResources类型的钩子通过Xposed提供的XResources类的setReplacement对原有的资源进行替换。在Xposed框架中，XResources是用于替代Resources的。
* App运行时候，Xposed框架截获相关的资源获取函数（比如Context的getString），先看看是否被replace了，如果是，则返回被replace的结果。

就这么简单。我们一步一步来看。

注意，XposedDemo并没有hook资源，请感兴趣的童鞋们自行加上该功能进行测试。

1.xposed框架一定会支持art框架以及安卓L

2.@Rovo89已经有在开发和测试支持art引擎的xposed框架，只是由于两个原因他没有发布，一个是由于需要经过大量的测试才行，第二是因为art本身还不够稳定，而且谷歌还在不停的修改和更新art引擎，而这些变化会导致xposed框架编码上会带来一些重构，同时还需要关注dual-stack 32/64 bit Zygote,以及严格SELinux的策略对xposed框架的影响。@Rovo89想等art引擎的正式版发布后再发布对于art的支持。

3.由于kitkat里的art测试版和android l里的art正式版是完全不同的环境，所以后续xposed框架将主要支持android l，如果测试和开发量不大的话，可能会考虑兼容kitkat里的art环境

## 待研究的点：

1.特殊情况hook，如隐藏java函数(@hide)及native函数测试

2.稳定性测试，包括api版本支持、厂商支持

