既然插件是一个apk，其实最重要的是启动插件中的Activity、Service等组件，因为像Activity这些组件，需要在清单文件中注册，否则启动的时候会报下面的错误





那么如果要启动插件中的Activity，就需要将其注册到宿主app的清单文件中

# Activity的启动流程

其实想要启动插件中的四大组件，例如Activity，就需要熟悉Activity的启动流程，为什么在没有注册的情况下，不能启动

## 启动插件Activity思路梳理

开头我们提到，为什么没有注册过的Activity不能启动？是因为当系统启动一个Activity的时候，需要通过AMS检测，当前被启动的Activity是否被注册，如果没有注册，那么就会报错，所以我们需要采用Hook的方式来欺骗系统已达到我们的目的。

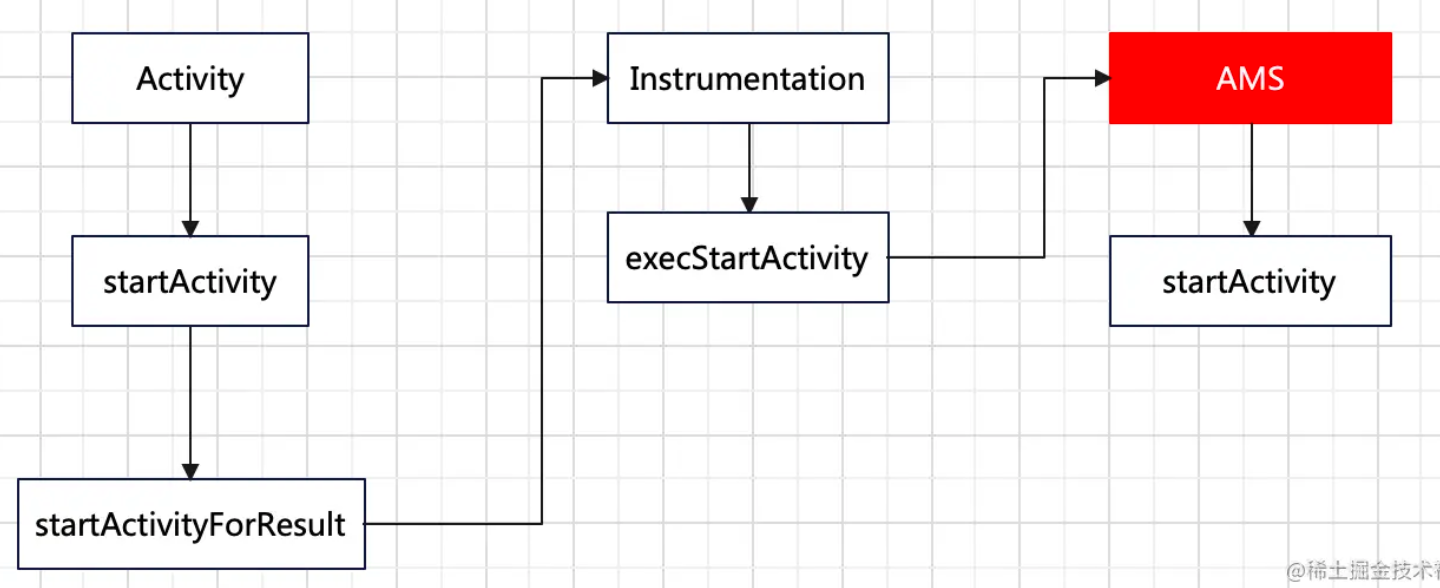
在应用层，我们能看到的就是，宿主app启动了插件中的Activity，其实在内部做了很多处理：

（1）首先将插件中的Activity替换成宿主中已经注册过的ProxyActivity，在启动的时候，通过AMS检测发现当前启动的Activity已经被注册过了，那么就会继续往下执行；  
（2）当AMS检测完成之后，再将宿主中的ProxyActivity替换成插件中的Activity，最终启动的就是插件中的Activity。



## 从源码理解Activity启动流程

## 对于Activity的启动流程，我想大家都非常了解了，记住关键的几个类Instrument、ActivityManagerService、ApplicationThread......，下面简单介绍下流程。

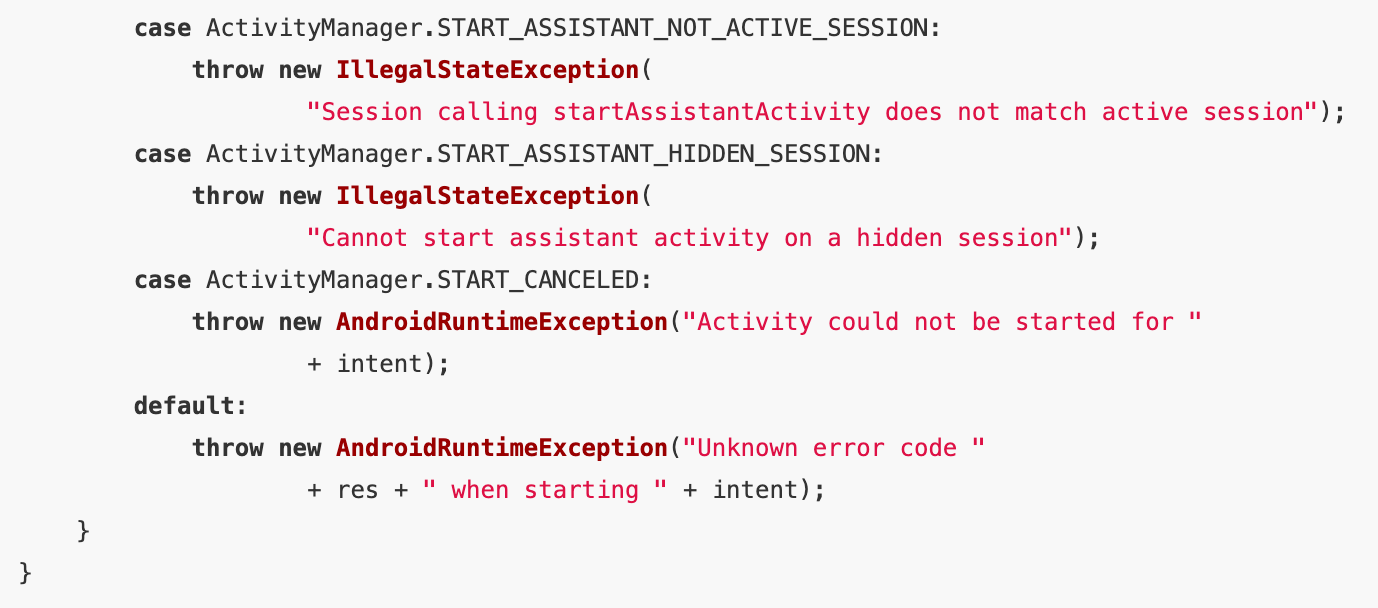


当调用Instrument的execStartActivity方法时，通过ActivityTaskManager.getService()获取AMS服务，调用AMS的startActivity方法，并返回检测的结果。



我们看一下checkStartActivityResult方法究竟做了什么？我们看一下START\_CLASS\_NOT\_FOUND这个结果，最终抛出的错误是不是就是文章开头的错误





也就是说在这个位置就已经检测完成，当前启动的Activity是否在清单文件中注册，也就是说，Hook点要在这个检测方法之前，完成对插件Activity的替换，这是Hook点1.

我们接着往下看AMS里主要做了啥



ActivityTaskManagerService继承自IActivityTaskManager.Stub接口，作为一个服务端的角色，接受客户端的请求，例如启动Activity，AMS除了做Intent启动的Activity检测之外，还做了什么事呢？

在ATMS中调用startActivity方法，最终是调用了startActivityAsUser方法，这个方法中，首先检查了调用者的权限，然后调用了ActivityStartController一系列方法，用于启动前的任务栈处理



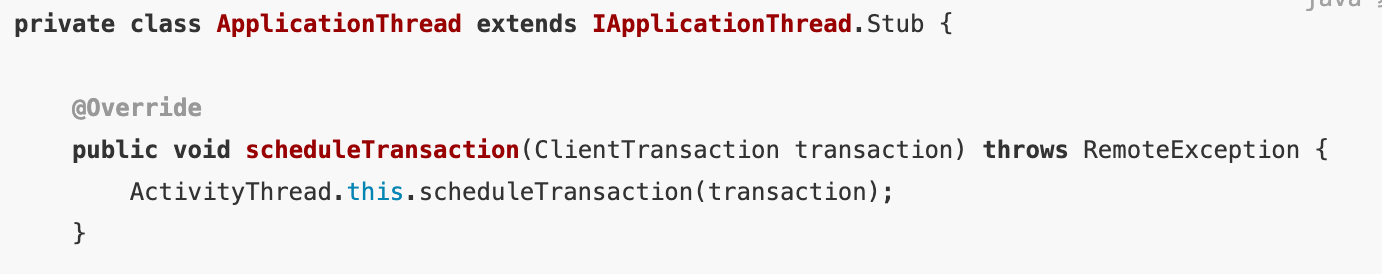
最终AMS会判断启动这个Activity的进程是否启动，没有启动的话就启动app进程，如果已经启动了会调用realStartActivityLocked方法



在realStartActivityLocked方法中，首先会创建一个启动Activity的Transaction，会通过ApplicationThread回调给客户端启动Activity，在AMS和客户端通信时，有两个对象需要注意：ActivityThread和ApplicationThread，ApplicationThread其实是ActivityThead在AMS的代理对象，AMS通过调用ApplicationThread与ActivityThead建立通信

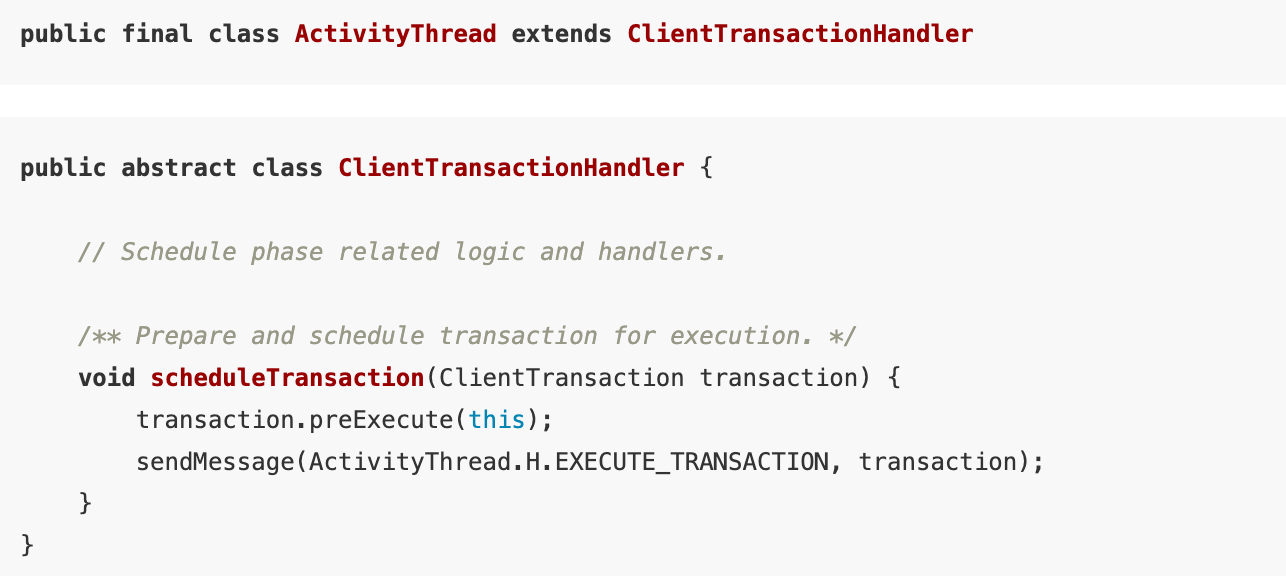


所以AMS这里的作用就是：****检测Activity合法性、检测app进程是否启动、通过ApplicationThead与ActivityThead建立通信。****

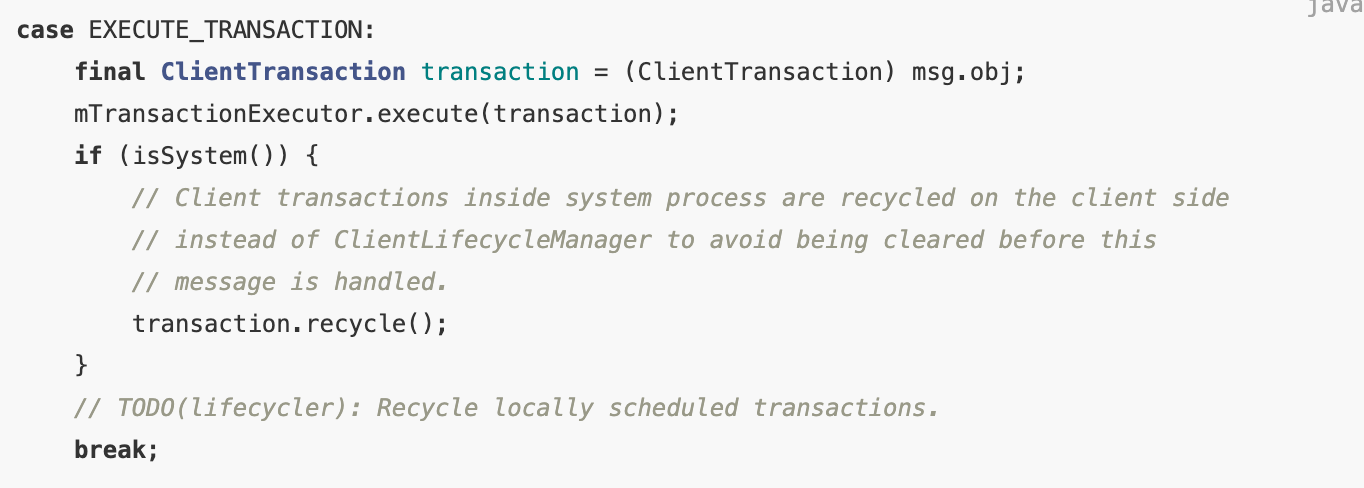


我们看下服务端的代理对象ApplicationThread，主要是处理AMS端的请求，例如realStartActivityLocked方法中创建的启动Activity的ClientTransaction，接收到之后，交给了ActivityThread处理。

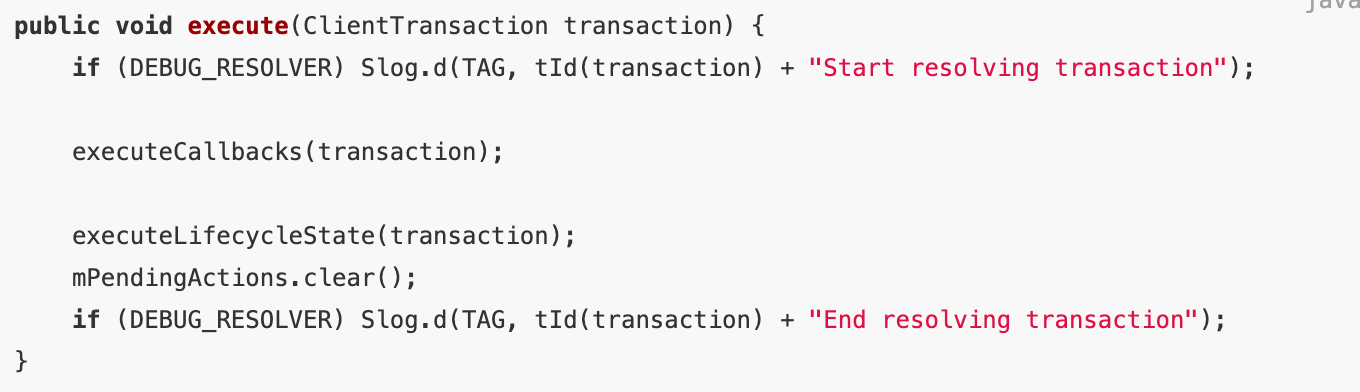
首先我们看下ActivityThread是什么，我们看到它是继承自ClientTransactionHandler，也就是说，在App进程内的处理，都是通过Handler来发送消息



对于EXECUTE\_TRANSACTION消息类型，就是用来处理AMS端发送来的消息



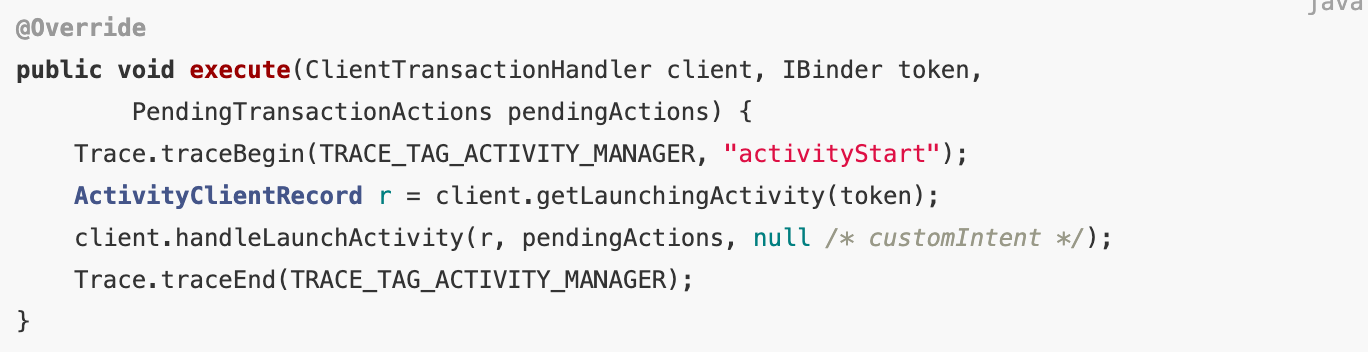
最终是调用了TransactionExecutor的execute方法，在execute方法中，调用executeCallbacks方法，我们可以在realStartActivityLocked方法中看到，在transaction添加了callback，所以这个方法就是执行这些callback

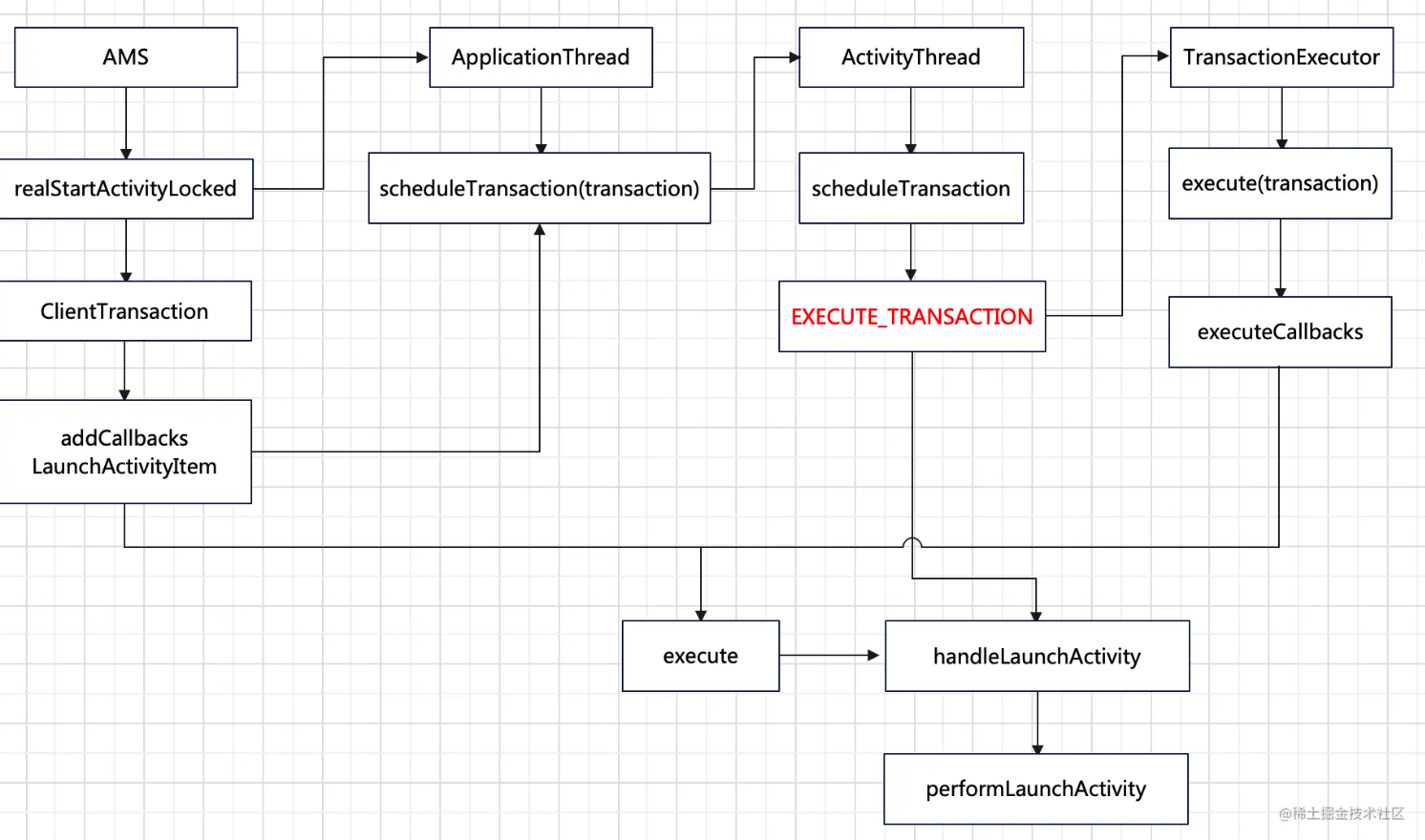


executeCallbacks方法比较简单，就是遍历获取所有的callback，然后调用callback的execute方法，也就是LaunchActivityItem的execute方法



在LaunchActivityItem的execute方法中，执行了ClientTransactionHandler的handleLaunchActivity方法，也就是ActivityThread的handleLaunchActivity方法，这里才是真正要启动这个Activity。





在performLaunchActivity方法中，通过Instrument类来创建新的Activity，并执行Activity的onCreate、onStart、onResume方法等，Activity就算是正式启动，所以第二个Hook点，我们知道了吗？

因为在performLaunchActivity方法调用之前，Activity都不算是真正地启动，也就是说，将宿主中的ProxyActivity替换成插件的Activity就需要在performLaunchActivity方法之前

# Hook方式实现插件四大组件启动

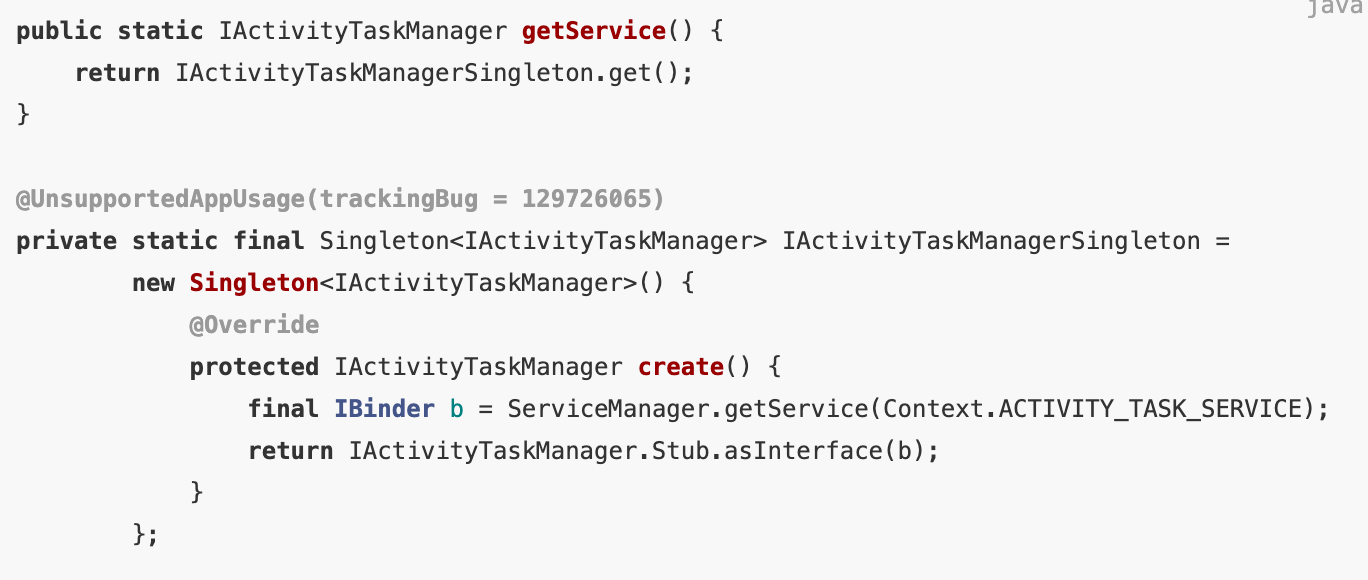
通过上面的源码，我们知道了AMS在什么时机去检测Activity合法性，以及Activity什么时候真正地启动，所以我们分步骤，先处理AMS合法性校验问题。

## Hook AMS

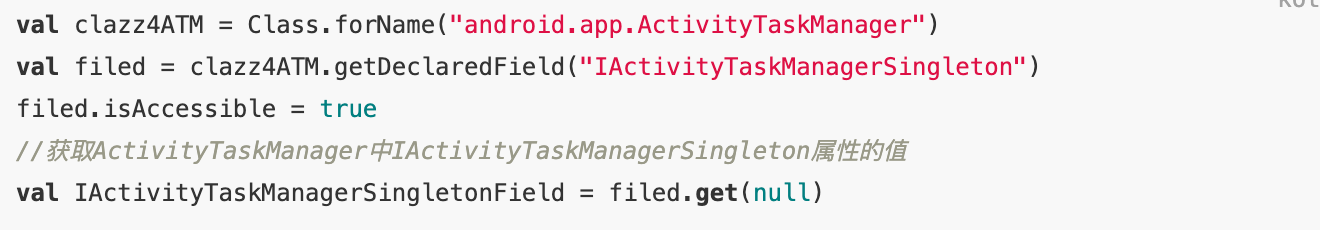
首先我们先看一下，如何通过Hook的方式，欺骗AMS过Activity合法性检测这一关，因为从前面的源码我们知道，调用startActivity最终会在Instrument的execStartActivity方法中检测，就是下面这两行代码



首先我们先看一下ActivityTaskManager.getService()最终返回的是一个IActivityTaskManager代理对象，是从IActivityTaskManagerSingleton中取出来的；



所以想要代替系ActivityTaskManager.getService()的返回值，首先反射获取IActivityTaskManagerSingleton这个属性



我们看到，IActivityTaskManagerSingleton其实是一个单例，需要通过get方法才能获取真正的实例，那么我们先看下Singleton是啥



其实还是比较简单的一个单例类，通过get方法拿到的其实就是mInstance，也就是说getService()拿到的就是这个mInstance，只不过是对IActivityTaskManager做了一层封装。

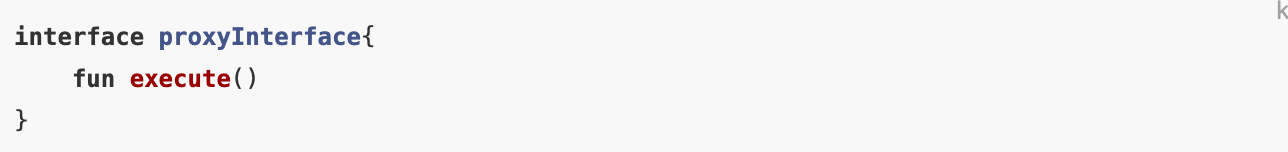


那么之前通过反射，拿到了Singleton对象，那么现在通过反射获取mInstance属性就拿到了getService()的返回值。

既然我们要hook系统的执行方式，当我们调用startActivity的时候，系统流程一步一步执行到位置①，这个时候，我们把getService()的返回值替换，换成我们自己处理过的IActivityTaskManager

## 动态代理在Hook中的使用

## 因为我们想要替换getService()的返回值，而且返回值是一个IActivityTaskManager接口对象，其实第一时间就会想到动态代理，那么动态代理是什么原理呢？



例如有一个接口，如果我们想要执行execute方法，那么就需要实现一个具体类，然后调用execute方法，这样其实在程序运行之前，就已经知道执行者是谁了；而动态代理则是程序在运行时，动态生成一个对象，这个时候才能知道执行者是谁，例如：

val instance = Proxy.newProxyInstance(classLoader, arrayOf(proxyInterface::class.java),object : InvocationHandler{

override fun invoke(

proxy: Any?,

method: java.lang.reflect.Method?,

args: Array<out Any>?

): Any {

Log.e("TAG","执行execute方法之前")

return method?.invoke(proxy,args)!!

}

})

(instance as proxyInterface).execute()

通过Proxy调用newProxyInstance方法，传入的参数为类加载器、接口的class对象，需要实现一个接口InvocationHandler，其中method为调用者调用的某个方法，这个方法调用之前，可以插入一些逻辑判断，更为灵活；

这样就意味着，我们可以创建一个IActivityTaskManager的动态代理对象，替换系统的getService()返回的IActivityTaskManager对象，那么在动态代理对象调用startActivity之前，可以对入参做处理。

val proxyClazz = Class.forName("android.app.IActivityTaskManager")

val newTaskManagerInstance = Proxy.newProxyInstance(Thread.currentThread().contextClassLoader, arrayOf(proxyClazz),object : InvocationHandler{

override fun invoke(proxy: Any?, method: Method?, args: Array<out Any>?): Any {

//做相应参数的处理

return method?.invoke(taskManagerInstance,args)!!

}

})

//替换

mInstance.set(singletonClazz,newTaskManagerInstance)

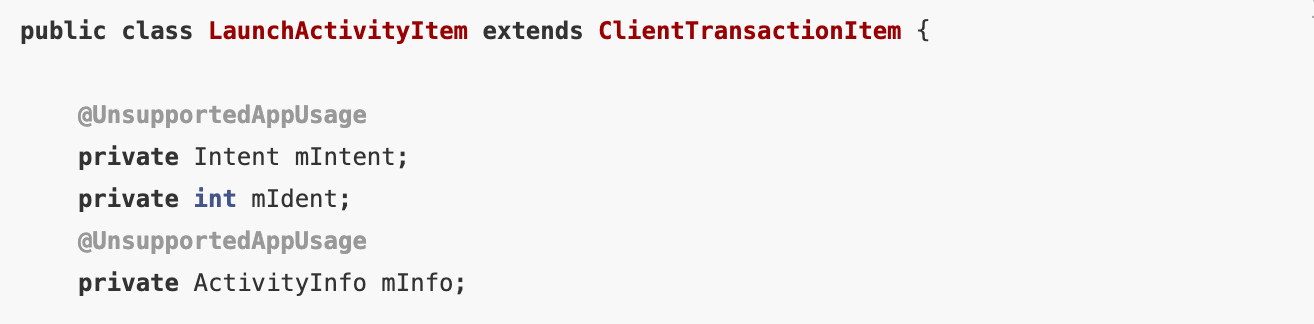
因为IActivityTaskManager中的方法非常多，因此需要根据方法名去过滤，只能对startActivity方法进行处理，当startActivity执行时，首先会将Intent类型参数取出来，把Intent替换成宿主中的ProxyActivity，但是真实的Intent不能丢掉，等合适的时间会重置回来，因此可以保存在代理的Intent中



这个时候，当我们启动插件中的Activity时，启动的就是这个代理Activity；这里需要说明一点就是，因为源码是Java写的，所以我们在Hook时，尽量还是用跟源码一致的语言，否则可能会出现类型不匹配的问题

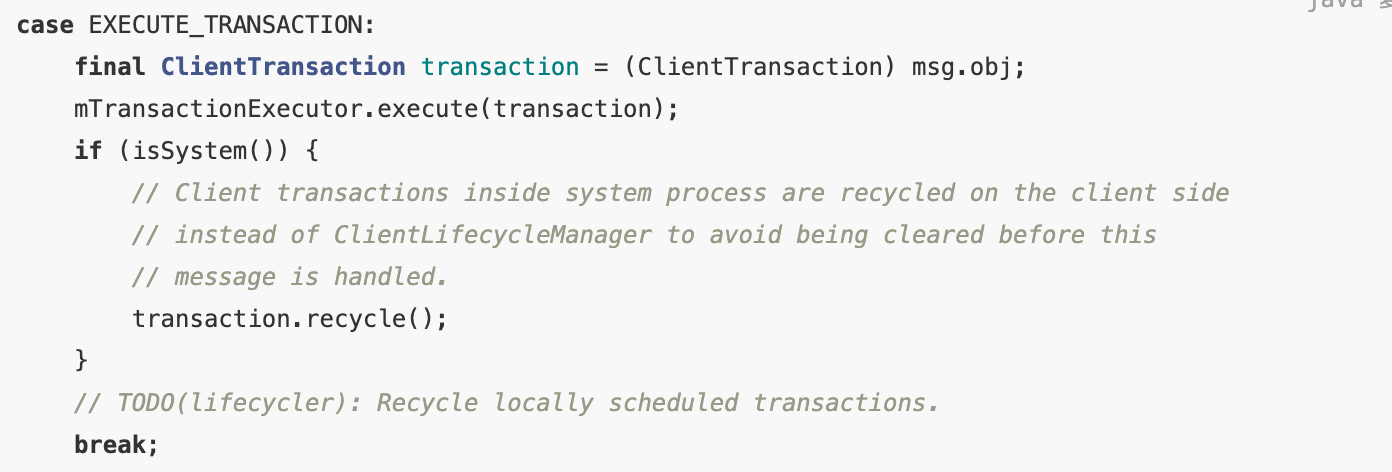
## Hook ActivityThread

通过前面的源码，我们知道在AMS的realStartActivityLocked方法中（辛苦伙伴们自己爬楼），是创建了一个ClientTransaction对象，然后在ClientTransaction对象中添加了callback，在启动Activity的时候，添加的callback对象为LaunchActivityItem，我们看下LaunchActivityItem源码



其他的先不用看，第一个参数就是mIntent，这个mIntent其实就是在之前AMS中替换的Intent，现在的目标就是将其替换成插件中的Activity。

那么在哪拿到这个对象呢？我们继续爬楼，在之前提到ActivityThread其实是一个Handler，通过Message的tag来分别处理数据。

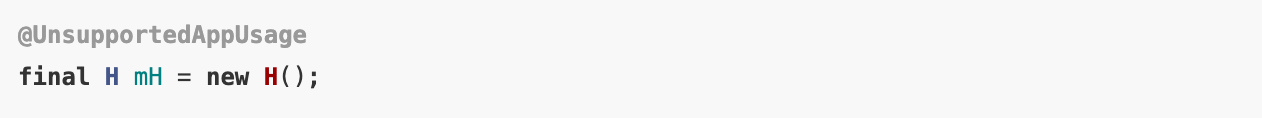


当启动Activity的时候，AMS通过ApplicationThread代理发送的消息类型为EXECUTE\_TRANSACTION，也就是说，我们可以Hook系统的Handler，当接收到EXECUTE\_TRANSACTION消息时，把Intent给替换掉。

在Activity中有一个Handler类H，这个类就负责接收Message然后处理



我们可以看到，在ActivityThread中是直接new出来，我们知道，当调用Handler的send方法时，在底层是调用了dispatchMessage方法从消息池中取出数据分发



因为空参构造方法，mCallback是空的，所以会直接走handleMessage方法，所以我们如果想拿消息自己去处理，那么可以自己创建一个mCallback对象赋值给mH的callback，那么是不就能走我们自己的逻辑处理了呢



首先我们先把前面梳理的思路通过反射梳理出来，有几个属性先强调一下，一个是sCurrentActivityThread，这个是ActivityThread类中的一个属性，就是当前对象，通过获取这个属性的值，来获取ActivityThread中的非静态属性值；还有一个就是Handler中的mCallback，因为系统Handler创建时没有设置callback，所以我们自己创建了一个callback，并给它赋值

public static void hookHandler() {

try {

Class<?> activityThreadClazz = Class.forName("android.app.ActivityThread");

//获取到ActivityThread对象

Field sCurrentActivityThreadField = activityThreadClazz.getDeclaredField("sCurrentActivityThread");

sCurrentActivityThreadField.setAccessible(true);

Object sCurrentActivityThread = sCurrentActivityThreadField.get(null);

//获取mH属性

Field mHField = activityThreadClazz.getDeclaredField("mH");

mHField.setAccessible(true);

//获取mH Handler对象

Object mH = mHField.get(sCurrentActivityThread);

//反射Handler

Class<?> handlerClazz = Class.forName("android.os.Handler");

Field mCallbackField = handlerClazz.getDeclaredField("mCallback");

mCallbackField.setAccessible(true);

//创建callback对象

Handler.Callback callback = new Handler.Callback() {

@Override

public boolean handleMessage(@NonNull Message msg) {

//处理Intent替换的逻辑

doIntentReplace(msg)

return false;

}

};

//给系统Handler赋值

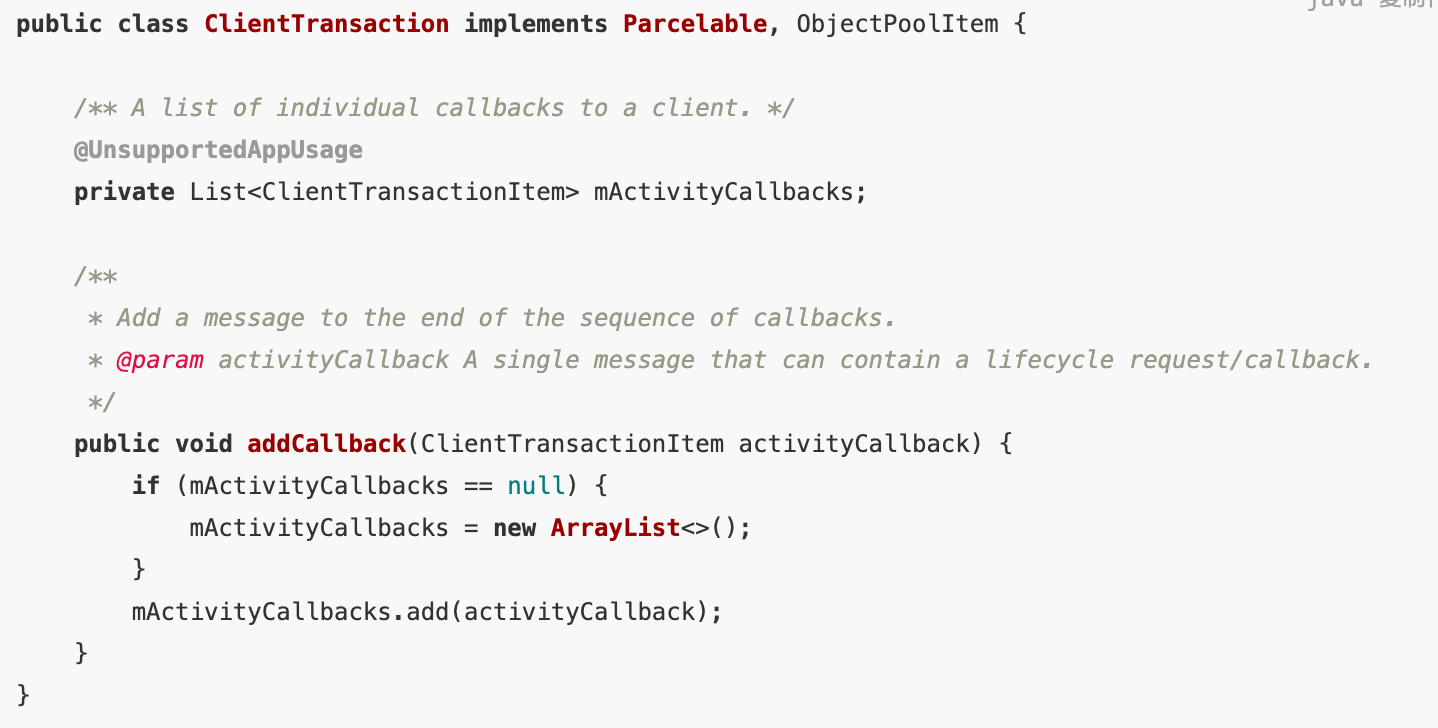
mCallbackField.set(mH, callback);

} catch (Exception exception) {

}

}

那么下面的逻辑，主要就是给Intent替换。我们主要关心EXECUTE\_TRANSACTION这个tag的处理。



当我们获取到Message携带的对象，其实就是ClientTransaction，我们看下源码，当调用addCallback的时候，就是将ClientTransactionItem放在了mActivityCallbacks中，所以拿到mActivityCallbacks就能拿到我们想要的LaunchActivityItem

private static void doIntentReplace(Message msg) {

switch (msg.what) {

case 159:

//获取

Class<?> transactionClazz = msg.obj.getClass();

try {

Field mActivityCallbacksField = transactionClazz.getDeclaredField("mActivityCallbacks");

mActivityCallbacksField.setAccessible(true);

List callbacks = (List) mActivityCallbacksField.get(msg.obj);

for (int i = 0; i < callbacks.size(); i++) {

Object transactionItem = callbacks.get(i);

//判断是不是LaunchActivityItem

if (transactionItem.getClass().getName().equals("android.app.servertransaction.LaunchActivityItem")) {

//获取mIntent属性

Field mIntentField = transactionItem.getClass().getDeclaredField("mIntent");

mIntentField.setAccessible(true);

Log.e("TAG", "intent " + mIntentField.get(transactionItem));

Intent mIntent = (Intent) mIntentField.get(transactionItem);

Intent oldIntent = mIntent.getParcelableExtra("old\_intent");

mIntentField.set(transactionItem, oldIntent);

// mIntent.setComponent(oldIntent.getComponent()); 这种方式同样有效

}

}

} catch (Exception e) {

e.printStackTrace();

}

break;

}

}

这样我们拿到了LaunchActivityItem中的mIntent之后，这个其实是代理的Intent，需要从中取出我们之前在HookAMS的时候保存的真正的Intent，并赋值，这个时候，Activity启动的时候启动的就是插件中的Activity。

其实，我们在学习插件化的时候，其实知识点还是很多的，像：Android类加载机制、Activity的启动流程、Hook原理等，并且能够深入源码，这一块其实是面试的重点和难点；这一节我们已经启动了插件中的Activity，其实像Service、BroadcastReceiver等组件，原理是一致，像Activity已经启动了，但是资源文件并没有被加载，所以下一节将着重介绍如何加载插件中的资源。

Hook AMS(Android Q版本)

public static void HookAMS() {

try {

Class<?> atmClazz = Class.forName("android.app.ActivityTaskManager");

Field iActivityTaskManagerSingletonField = atmClazz.getDeclaredField("IActivityTaskManagerSingleton");

iActivityTaskManagerSingletonField.setAccessible(true);

Object atmSingleton = iActivityTaskManagerSingletonField.get(null);

Class<?> singletonClazz = Class.forName("android.util.Singleton");

Field mInstanceField = singletonClazz.getDeclaredField("mInstance");

mInstanceField.setAccessible(true);

final Object atmService = mInstanceField.get(atmSingleton);

Class<?> proxyClazz = Class.forName("android.app.IActivityTaskManager");

Object newProxyInstance = Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(), new Class[]{proxyClazz}, new InvocationHandler() {

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

if (method.getName().equals("startActivity")) {

int targetIndex = 0;

//修改Intent的值

for (int i = 0; i < args.length; i++) {

if (args[i] instanceof Intent) {

targetIndex = i;

break;

}

}

Intent oldIntent = (Intent) args[targetIndex];

Intent proxyIntent = new Intent();

proxyIntent.setComponent(new ComponentName("com.lay.image\_process", "com.lay.image\_process.MainActivity2"));

proxyIntent.putExtra("old\_intent", oldIntent);

args[targetIndex] = proxyIntent;

}

Log.e("TAG", "atmService=" + atmService + " method=" + method + ", args=" + args);

return method.invoke(atmService, args);

}

});

mInstanceField.set(atmSingleton, newProxyInstance);

} catch (Exception e) {

Log.e("TAG", "e--" + e.getMessage());

e.printStackTrace();

}

}

Hook Handler(Android Q版本)

public static void hookHandler() {

try {

Class<?> activityThreadClazz = Class.forName("android.app.ActivityThread");

//获取到ActivityThread对象

Field sCurrentActivityThreadField = activityThreadClazz.getDeclaredField("sCurrentActivityThread");

sCurrentActivityThreadField.setAccessible(true);

Object sCurrentActivityThread = sCurrentActivityThreadField.get(null);

//获取mH属性

Field mHField = activityThreadClazz.getDeclaredField("mH");

mHField.setAccessible(true);

//获取mH Handler对象

Object mH = mHField.get(sCurrentActivityThread);

//反射Handler

Class<?> handlerClazz = Class.forName("android.os.Handler");

Field mCallbackField = handlerClazz.getDeclaredField("mCallback");

mCallbackField.setAccessible(true);

//创建callback对象

Handler.Callback callback = new Handler.Callback() {

@Override

public boolean handleMessage(@NonNull Message msg) {

//处理Intent替换的逻辑

doIntentReplace(msg);

return false;

}

};

//给系统Handler赋值

mCallbackField.set(mH, callback);

} catch (Exception exception) {

}

}

private static void doIntentReplace(Message msg) {

switch (msg.what) {

case 159:

//获取

Class<?> transactionClazz = msg.obj.getClass();

try {

Field mActivityCallbacksField = transactionClazz.getDeclaredField("mActivityCallbacks");

mActivityCallbacksField.setAccessible(true);

List callbacks = (List) mActivityCallbacksField.get(msg.obj);

for (int i = 0; i < callbacks.size(); i++) {

Object transactionItem = callbacks.get(i);

//判断是不是LaunchActivityItem

if (transactionItem.getClass().getName().equals("android.app.servertransaction.LaunchActivityItem")) {

//获取mIntent属性

Field mIntentField = transactionItem.getClass().getDeclaredField("mIntent");

mIntentField.setAccessible(true);

Log.e("TAG", "intent " + mIntentField.get(transactionItem));

Intent mIntent = (Intent) mIntentField.get(transactionItem);

Intent oldIntent = mIntent.getParcelableExtra("old\_intent");

mIntentField.set(transactionItem, oldIntent);

// mIntent.setComponent(oldIntent.getComponent()); 这种方式同样有效

}

}

} catch (Exception e) {

e.printStackTrace();

}

break;

}

}