Android 插件框架原理

# 要解决的问题

代码加载

ClassLoader可以加载代码，但是Android的组件具有生命周期，要解决组件的生命周期管理

资源加载

AssetManager的隐藏API addAssetPath

比较有代表的插件框架

DroidPlugin

# 动态代理

什么是代理

静态代理

其实就是代理模式的实现。

动态代理

关键函数

java.lang.reflect.Proxy

public static Object newProxyInstance(

ClassLoader loader,

Class<?>[] interfaces,

InvocationHandler h

) throws IllegalArgumentException

loader:一个ClassLoader对象，定义了由哪个ClassLoader对象来对生成的代理对象进行加载

interfaces**:一个Interface对象的数组，表示的是我将要给我需要代理的对象提供一组什么接口，如果我提供了一组接口给它，那么这个代理对象就宣称实现了该接口(多态)，这样我就能调用这组接口中的方法了**

h:一个InvocationHandler对象，表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个InvocationHandler对象上。

java.lang.reflect

interface InvocationHandler

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

HOOK startActivity

Context.startActivity和Activity.startActivity调用链不同

如何HOOK Context.startActivity?

实际上是HOOK ContextImpl里的方法

1 通过android.app.ActivityThread里的currentActivityThread方法拿到当前主线程

2 实现Instrumentation代理对象，代理execStartActivity方法

3 替换掉ActivityThread对象的成员变量mInstrumentation

# Binder HOOK

HOOK 住ActivityManagerService可以让插件自由使用startActivity方法

Binder本地对象：存在于某个单独的进程

Binder代理对象

系统服务的使用和获取过程

客户获取方法

ActivityManager am =

(ActivityManager)context.getSystemService(Context.ACTIVITY\_SERVICE)

context ContextImpl

在ContextImpl里有一张存放service的map SYSTEM\_SERVICE\_MAP

系统服务的实际获取过程

IBinder b = ServiceManager.getService(“service\_name”)

IXXInterface in = IXXInterface.Stub.asInterface(b) 转换为Service接口

asInterface过程

android.content.IClipboard

IClipboard.Stub.asInterface

obj.queryLocalInterface()

如果本进程存在这个对象，直接返回，否则创建一个代理对象

为了让asInterface这个函数返回我们的代理对象。需要HOOK queryLocalInterface这个函数

需要生成一个IBinder的代理对象，来完成对queryLocalInterface的HOOK

伪造一个系统服务对象

AMS和PMS

ActivityManagerService

PackageManagerService

AMS

不论读者是否知道，我们使用startActivity有两种形式：

直接调用Context类的startActivity方法；**这种方式启动的Activity没有Activity栈**，因此不能以standard方式启动，必须加上FLAG\_ACTIVITY\_NEW\_TASK这个Flag。

调用被Activity类重载过的startActivity方法，通常在我们的Activity中直接调用这个方法就是这种形式；

ActivityManager

ActivityManagerService 本地Binder对象

ActivityManagerNative Binder代理对象

三者之间的关系

ContextImpl startActivity

Instrumentation execStartActivity

ActivityManagerNative startActivity

android.app.ActivityManagerNative

private static final Singleton<IActivityManager> **gDefault** = ?

android.util.Singleton

mInstance

HOOK关键函数

Class.forName

getDeclaredField(“gDefault”)

Field.get set

HOOK PMS

android.content.pm.IPackageManager

android.app.AcitivityThread

sPackageManager

ApplicationPackageManager

mPM

Activity加载

ClassLoader可以动态加载模块，但是从ClassLoader里加载进来的Activity没有生命周期，如何将其交给AMS来管理，赋予Activity生命周期？

startActivity的启动过程

ActivityManagerNative类的startActivity 客户端

ActivityManagerService类的startActivity IPC 服务端

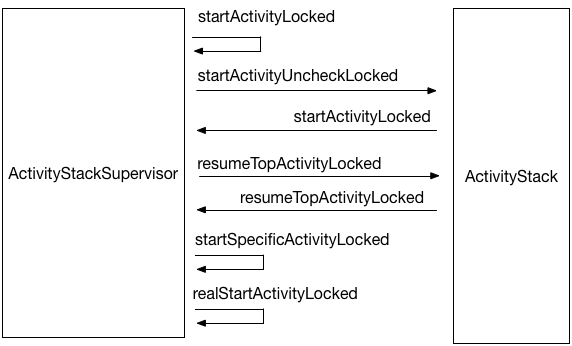
startActivityAsUser 方法

ActivityStackSupervisor 类 Activity栈管理

startActivityWait

startActivityMayWait 对Intent解析，取出需要启动的Activity信息

startActivityLocked 检查，权限检查，exported属性等



realStartActivityLocked

ApplicationThread的scheduleLaunchActivity 开始真正的Activity对象创建以及启动过程

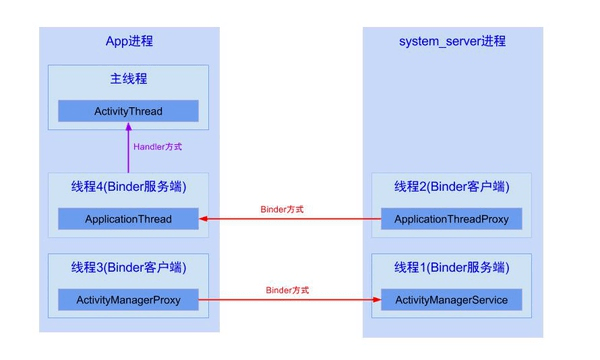
ApplicationThread是什么？

ApplicationThread实际上是一个Binder对象，是App所在的进程与AMS所在进程system\_server通信的桥梁；在Activity启动的过程中，App进程会频繁地与AMS进程进行通信：

App进程会委托AMS进程完成Activity生命周期的管理以及任务栈的管理；这个通信过程AMS是Server端，App进程通过持有AMS的client代理ActivityManagerNative完成通信过程；

AMS进程完成生命周期管理以及任务栈管理后，会把控制权交给App进程，让App进程完成Activity类对象的创建，以及生命周期回调；这个通信过程也是通过Binder完成的，App所在server端的Binder对象存在于ActivityThread的内部类ApplicationThread；AMS所在client通过持有IApplicationThread的代理对象完成对于App进程的通信。

App进程与AMS进程的通信过程如图所示：



App进程内部的ApplicationThread server端内部有**自己的Binder线程池**，它与App主线程的通信通过Handler完成，这个Handler存在于ActivityThread类，它的名字很简单就叫H

ApplicationThread::scheduleLaunchActivity

ActivityThread::handleLaunchActivity

performLaunchActivity(r,customIntent)

这个函数做了两件重要的事情：

使用ClassLoader加载并通过反射创建Activity对象

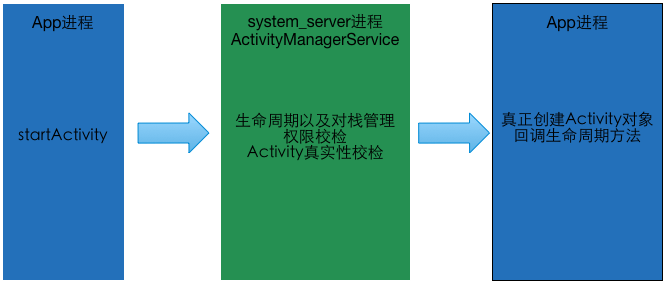
ClassLoader cl = r.packageInfo.getClassLoader()

activity = mInstrumentation.newActivity(cl,component.getClassName(),r.intent)

如果Application还没有创建，那么创建Application对象并回调相应的生命周期方法；

Application app = r.packageInfo.makeApplication(false,mInstrumentation)

简要过程



ActivityThread

H mH

H

mCallback