# 一、Android插件化

## 1、背景

* 从技术上讲，业务逻辑的复杂导致代码量急剧膨胀，会突破65535方法数。
* 运营为王的时代对于模块热更新提出了更高的要求。
* 在业务层面上，每个模块升级新功能都需要对整个app进行升级，发布流程不仅复杂而且效率低下。移动互联——小步快跑和持续迭代。

## 2、方法：

* H5和Hybird
* FaceBook推出了react-native
* 插件化技术

## 3、插件化的核心问题

### 3.1、代码加载

类的加载可以使用Java的**ClassLoader机制**，但是对于Android来说，并不是说类加载进来就可以用了，很多组件都是有“生命”的；因此对于这些有血有肉的类，必须给它们注入活力，也就是所谓的组件生命周期管理；

另外，如何管理加载进来的类也是一个问题。假设多个插件依赖了相同的类，是抽取公共依赖进行管理还是插件单独依赖？这就是ClassLoader的管理问题.

### 3.2、资源加载

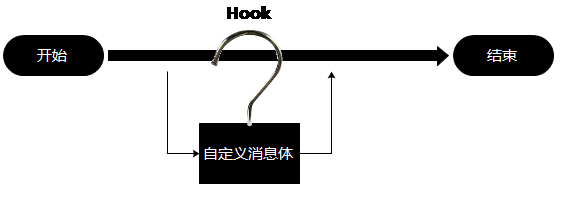
资源加载方案大家使用的原理都差不多，都是用AssetManager的隐藏方法addAssetPath；但是，不同插件的资源如何管理？是公用一套资源还是插件独立资源？共用资源如何避免资源冲突？对于资源加载，有的方案共用一套资源并采用资源分段机制解决冲突（要么修改aapt要么添加编译插件）；有的方案选择独立资源，不同插件管理自己的资源。

# 二、了解Hook

## 1、介绍

还没有接触过Hook技术读者一定会对Hook一词感觉到特别的陌生，Hook英文翻译过来就是“钩子”的意思，那我们在什么时候使用这个“钩子”呢？

我们知道，在Android操作系统中系统维护着自己的一套事件分发机制。应用程序，包括应用触发事件和后台逻辑处理，也是根据事件流程一步步的向下执行。而“钩子”的意思，就是在事件传送到终点前截获并监控事件的传输，像个钩子勾上事件一样。并且能够在勾上事件时，处理一些自己特定的事件。如下图所示：



Hook的这个本领，使它能够将自身的代码“融入”被勾住（Hook）的程序的进程中，成为目标进程的一个部分。我们也知道，在Android系统中使用了沙箱机制，普通用户程序的进程空间都是独立的，程序的运行彼此间都不受干扰。

这就使我们希望通过一个程序改变其他程序的某些行为的想法不能直接实现，但是Hook的出现给我们开拓了解决此类问题的道路。当然，根据Hook对象与Hook后处理的事件方式不同，Hook还分为不同的种类，如消息Hook、API Hook等。

## 2、框架

Cydia Substrate框架

Xposed框架

# 三、[Hook机制之代理](https://github.com/DroidPluginTeam/DroidPlugin/blob/master/DOC/tianweishu/Hook%E6%9C%BA%E5%88%B6%E4%B9%8B%E4%BB%A3%E7%90%86Hook.md)

## 1. 代理是什么

代理可以实现**方法增强**，比如常用的日志,缓存等；也可以实现方法拦截，通过代理方法修改原方法的参数和返回值，从而实现某种不可告人的目的～接下来我们用代码解释一下。

## 2.静态代理

## 3.动态代理

动态代理主要处理InvocationHandler和Proxy类；

## 4.[代理Hook](https://github.com/DroidPluginTeam/DroidPlugin/blob/master/DOC/tianweishu/Hook%E6%9C%BA%E5%88%B6%E4%B9%8B%E4%BB%A3%E7%90%86Hook.md)

Hook掉startActivity这个方法

### 4.1、寻找Hook点，原则是静态变量或者单例对象，尽量Hook pulic的对象和方法，非public不保证每个版本都一样，需要适配。

**静态变量和单例**是比较好Hook点

@Override

public void startActivity(Intent intent, Bundle options) {

warnIfCallingFromSystemProcess();

if ((intent.getFlags()&Intent.FLAG\_ACTIVITY\_NEW\_TASK) == 0) {

throw new AndroidRuntimeException(

"Calling startActivity() from outside of an Activity "

+ " context requires the FLAG\_ACTIVITY\_NEW\_TASK flag."

+ " Is this really what you want?");

}

mMainThread.getInstrumentation().execStartActivity(

getOuterContext(), mMainThread.getApplicationThread(), null,

(Activity)null, intent, -1, options);

}

使用了ActivityThread类的mInstrumentation成员的execStartActivity方法；注意到，ActivityThread 实际上是主线程，而主线程一个进程只有一个，因此这里是一个良好的Hook点。

### 4.2、选择合适的代理方式，如果是接口可以用动态代理；如果是类可以手动写代理也可以使用cglib。

### 4.3、偷梁换柱——用代理对象替换原始对象

public static void attachContext() throws Exception{

// 先获取到当前的ActivityThread对象

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

Method currentActivityThreadMethod = activityThreadClass.getDeclaredMethod("currentActivityThread");

currentActivityThreadMethod.setAccessible(true);

Object currentActivityThread = currentActivityThreadMethod.invoke(null);

// 拿到原始的 mInstrumentation字段

Field mInstrumentationField = activityThreadClass.getDeclaredField("mInstrumentation");

mInstrumentationField.setAccessible(true);

Instrumentation mInstrumentation = (Instrumentation) mInstrumentationField.get(currentActivityThread);

// 创建代理对象

Instrumentation evilInstrumentation = new EvilInstrumentation(mInstrumentation);

// 偷梁换柱

mInstrumentationField.set(currentActivityThread, evilInstrumentation);

}

# [四、Hook机制之Binder-Hook](https://github.com/DroidPluginTeam/DroidPlugin/blob/master/DOC/tianweishu/Hook%E6%9C%BA%E5%88%B6%E4%B9%8BBinder-Hook.md)

Android系统通过Binder机制给应用程序提供了一系列的系统服务，诸如ActivityManagerService，ClipboardManager， AudioManager等；这些广泛存在系统服务给应用程序提供了诸如任务管理，音频，视频等异常强大的功能。

## [1、Hook系统服务的机制称之为Binder Hook](http://blog.csdn.net/chunqiuwei/article/details/50068141)

## 2、系统服务的获取过程

ActivityManager am = (ActivityManager) context.getSystemService(Context.ACTIVITY\_SERVICE);

可是这个貌似跟ServiceManager没有什么关系啊？我们再查看getSystemService方法；(Context的实现在framework -core-java-android/app/ContextImpl里面)：

public Object getSystemService(String name) {

ServiceFetcher fetcher = SYSTEM\_SERVICE\_MAP.get(name);

return fetcher == null ? null : fetcher.getService(this);

}

很简单，所有的service对象都保存在一张map里面，我们再看这个map是怎么初始化的：

registerService(ACCOUNT\_SERVICE, new ServiceFetcher() {

public Object createService(ContextImpl ctx) {

IBinder b = ServiceManager.getService(ACCOUNT\_SERVICE);

IAccountManager service = IAccountManager.Stub.asInterface(b);

return new AccountManager(ctx, service);

}});

在ContextImpl的静态初始化块里面，有的Service是像上面这样初始化的；可以看到，确实使用了ServiceManager；当然还有一些service并没有直接使用ServiceManager，而是做了一层包装并返回了这个包装对象，比如我们的ActivityManager，它返回的是ActivityManager这个包装对象：

registerService(ACTIVITY\_SERVICE, new ServiceFetcher() {

public Object createService(ContextImpl ctx) {

return new ActivityManager(ctx.getOuterContext(), ctx.mMainThread.getHandler());

}});

但是在ActivityManager这个类内部，也使用了ServiceManager；具体来说，因为ActivityManager里面所有的核心操作都是使用ActivityManagerNative.getDefault()完成的。那么这个语句干了什么呢？

private static final Singleton<IActivityManager> gDefault = new Singleton<IActivityManager>() {

protected IActivityManager create() {

IBinder b = ServiceManager.getService("activity");

IActivityManager am = asInterface(b);

return am;

}

};

因此，通过分析我们得知，系统Service的使用其实就分为两步：

IBinder b = ServiceManager.getService("service\_name"); // 获取原始的IBinder对象

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

## 3、寻找Hook点