24.热修复&插件化

https://zhuanlan.zhihu.com/p/33017826

https://juejin.cn/post/6844903613865672718#heading-1

https://fashare2015.github.io/2018/01/24/dynamic-load-learning-load-activity/

24.1.插件化的定义

所谓插件化，就是让我们的应用不必再像原来一样把所有的内容都放在一个apk中，可以把一些功能和逻辑单独抽出来放在插件apk中，然后主apk做到 [按需调用] 。

24.2.插件化的优势

1.减少主apk的体积、65535 问题。让应用更轻便

2.让用户不用重新安装 APK 就能升级应用功能，减少发版本频率，增加用户体验。

3.模块化、解耦合

应用程序的工程和功能模块数量会越来越多，如果功能模块间的耦合度较高，修改一个模块会影响其它功能模块，势必会极大地增加成本。

24.3.插件化框架对比

1.最早的插件化框架：2012年大众点评的屠毅敏就推出了AndroidDynamicLoader框架。

(1) 首先通过 DLPluginManager 的 loadApk 函数加载插件，这步每个插件只需调用一次。

(2) 通过 DLPluginManager 的 startPluginActivity 函数启动代理 Activity。

(3) 代理 Activity 启动过程中构建、启动插件 Activity

2.目前主流的插件化方案有滴滴任玉刚的VirtualApk、 Wequick的Small框架。

Hook IActivityManager和Hook Instrumentation。主要方案就是先用一个在AndroidManifest.xml中注册的Activity来进行占坑，用来通过AMS的校验，接着在合适的时机用插件Activity替换占坑的Activity。

24.4.插件化流程

在Android中应用插件化技术，其实也就是动态加载的过程，分为以下几步：

把可执行文件（ .so/dex/jar/apk 等）拷贝到应用 APP 内部。

加载可执行文件，更换静态资源

调用具体的方法执行业务逻辑

24.5.插件化原理

一.类加载原理

1.1 apk被安装之后，APK文件的代码以及资源会被系统存放在固定的目录（比如/data/app/package_name/base-1.apk)系统在进行类加载的时候，会自动去这一个或者几个特定的路径来寻找这个类；

系统无法加载我们插件中的类。需要我们自己处理 这个类加载的过程。

1.2 这里用的DexClassLoader Android中的类加载器中主要包括三类BootClassLoader（继承ClassLoader），PathClassLoader和DexClassLoader,后两个继承于BaseDexClassLoader。

BootClassLoader:主要用于加载系统的类，包括java和android系统的类库。（比如TextView,Context，只要是系统的类都是由BootClassLoader加载完成）。

PathClassLoader:主要用于加载我们应用程序内的类。路径是固定的，只能加载 /data/app，无法指定解压释放dex的路径，无法动态加载。对于我们的应用默认为PathClassLoader

DexClassLoader：可以用来加载任意路径的zip,jar或者apk文件。

DexClassLoader重载findClass方法，在加载类时会调用其内部的DexPathList去加载。 而DexPathList的loadClass会去遍历DexFile直到找到需要加载的类。

1.3 插件化中 有单DexClassLoader和多DexClassLoader两种结构。

插件化要解决的是

主工程调用插件

有单DexClassLoader

对于每个插件都会生成一个DexClassLoader，

当加载该插件中的类时需要通过对应DexClassLoader加载 需要先通过插件的ClassLoader加载该类再通过反射调用其方法

多DexClassLoader

将插件的DexClassLoader中的pathList合并到主工程的DexClassLoader中。 主工程则可以直接通过类名去访问插件中的类

插件调用主工程

在构造插件的ClassLoader时会传入主工程的ClassLoader作为父加载器， 所以插件是可以直接可以通过类名引用主工程的类。

双亲委派机制: ClassLoader在加载一个字节码时，首先会询问 当前的 ClassLoader是否已经加载过此类，如果已经加载过就直接返回，不在重复的去 加载，如果没有的话，会查询它的parent是否已经加载过此类，如果加载过那 么就直接返回parent加载过的字节码文件，如果整个继承线路上都没有加载过 此类，最后由子ClassLoader执行真正的加载。

二.资源加载原理

2.1 Android系统通过Resource对象加载资源,Resource对象的生成只要将插件apk的路径加入到AssetManager中，便能够实现对插件资源的访问。

和代码加载相似，插件和主工程的资源关系也有两种处理方式：

2.2 合并式：addAssetPath时加入所有插件和主工程的路径；

独立式：各个插件只添加自己apk路径

三.Activity加载原理

代理：dynamic-load-apk采用。 Hook：主流。

Hook实现方式有两种：Hook IActivityManager和Hook Instrumentation。

主要方案就是先用一个在AndroidManifest.xml中注册的Activity来进行占坑，用来通过AMS的校验，接着在合适的时机用插件Activity替换占坑的Activity。

3.1 Hook IActivityManager：

3.1.1 占坑、通过校验：

hook点

IActivityManager

Activity启动过程

startActivity-Instrumentation.execStartActivity()->ActivityManager.getService().startActivity()->IActivityManager.Stub.asInterface->AMS.startActivity()

ActivityManager中getService()借助Singleton类实现单例,而且该单例是静态的,IActivityManager是一个比较好的Hook点。由于Hook点IActivityManager是一个接口(源码中IActivityManager.aidl文件)，建议这里采用动态代理。

过程

1.1 AndroidManifest.xml中注册SubActivity

1.2 拦截startActivity方法，获取参数args中保存的Intent对象，它是原本要启动插件TargetActivity的Intent。

1.3 新建一个subIntent用来启动StubActivity，并将前面得到的TargetActivity的Intent保存到subIntent中，便于以后还原TargetActivity。

代码

1、反射获取ActivityManager类中的静态实例IActivityManagerSingleton

2.反射获取Singleton中的mInstance实例

3、获取此IActivityManagerSingleton内部的mInstance

4.动态代理创建代理IActivityManager

5、将重写的代理IActivityManager设置给mInstance

3.1.2 还原插件Activity：

hook点

HandeIr中的mCallback

activity启动过程中,AMS会远程掉用applicationThread的scheduleLaunchActivity。

ActivityThread中的Handler-》h的handleLaunchActivity处理LAUNCH_ACTIVITY类型的消息-》ActivityThread#handleLaunchActivity-》instmentiong启动activity->Activity的onCreate方法。

在Handler的dispatchMessage处理消息的这个方法中，看到如果HandeIr的Callback类型的mCallBack不为null，就会执行mCallback的handleMessage方法，因此mCallback可以作为Hook点。我们可以用自定义的Callback来替换mCallback。

过程

重写callback，当收到消息的类型为LAUNCH_ACTIVITY时，将启动SubActivity的Intent替换为启动TargetActivity的Intent。

反射获取ActivityThread，反射获取mH

替换callback

使用时则在application的attachBaseContext方法中进行hook即可。

3.2 Hook Instrumentation：

与Hook IActivity实现不同的是，用占坑Activity替换插件Activity以及还原插件Activity的地方不同。

分析：

在Activity通过AMS校验前，会调用Activity的startActivityForResult方法

并会调用了Instrumentation的execStartActivity方法来激活Activity的生命周期。并且在ActivityThread的performLaunchActivity中使用了mInstrumentation的newActivity方法，其内部会用类加载器来创建Activity的实例。

方案：

1.在Instrumentation的execStartActivity方法中用占坑SubActivity来通过AMS的验证

首先检查TargetActivity是否已经注册，如果没有则将TargetActivity的ClassName保存起来用于后面还原。接着把要启动的TargetActivity替换为StubActivity，最后通过反射调用execStartActivity方法，这样就可以用StubActivity通过AMS的验证。

2.在Instrumentation的newActivity方法中还原TargetActivity

在newActivity方法中创建了此前保存的TargetActivity，完成了还原TargetActivit。

用InstrumentationProxy替换mInstrumentation。

3、插件Activity的生命周期：

AMS和ActivityThread之间的通信采用了token来对Activity进行标识，并且此后的Activity的生命周期处理也是根据token来对Activity进行标识的，因为我们在Activity启动时用插件TargetActivity替换占坑SubActivity，这一过程在performLaunchActivity之前，因此performLaunchActivity的r.token就是TargetActivity。所以TargetActivity具有生命周期。