文章编号：2095-2163(2019)01-0235-04 中图分类号：TP311.52 文献标志码：A

Android APP 功能插件化机制的研究与实现

熊建芳

（岭南师范学院 信息工程学院，广东 湛江 524048）

摘 要：本文结合动态加载技术与相关机制实现 Android APP 的插件化，该插件机制不仅可以实现模块化的动态加载和 bug 的修复，并且可以随时安装和卸载。与传统的开发方式相比，以插件化开发方式开发的 APP 不论在开发效率还是用户体验性上都会有较大的提高。

关键词：Android APP；插件化机制；粒度；安全机制；模块化更新；热修复

Research and implementation of plug-in mechanism of Android APP function

XIONG Jianfang

（School of Information Engineering, Lingnan Normal University, Zhanjiang Guangdong 524048, China）

【Abstract】 This paper combines dynamic loading technology with related mechanisms to achieve Android APP plug-in. it proves that the plug-in mechanism can achieve modular dynamic loading and bugs fixing, and the module can be installed and uninstalled. Comparing with the traditional development methods, regardless of the development efficiency or the user´s experience, the developed APP using plug-in method has a greater improvement.

【Key words】 Android APP; plugin mechanism; granularity; security mechanism; modular update; hot fixed

0 引 言

随着智能手机的大众化，移动应用层出不穷，目前已经开源的插件化框架有很多，但大多数针对性比较强，要么针对新功能的增加，要么是 bug 的修复，并没有一个框架对二者进行融合；并且这些框架也没有采取任何安全机制来保障应用的安全。

本文提出了一种 Android App 插件化机制。这种插件机制按照插件化的粒度分成模块化更新和热修复。模块化更新针对新功能的增加，而热修复针对类文件中方法级别的修复。在这种机制下用户无需手动安装新模块，APP 通过动态加载的方式就可以进行更新，大大提高了用户的体验性，更为开发者部署应用、更新应用、修复应用的 bug 提供了很大的方便。

1 总体设计

对于插件化的开发方式，保证模块代码在网络上的安全性是十分重要的，尤其对于一些敏感性比较高的应用，安全就显的尤为重要。目前开源的插件化框架对安全性这方面并没有做相应的保障。基于这些需求，本文所研究的机制将通过 HOOK 技术接管系统的部分管理机制，实现动态加载 APK 或者是单个 dex 文件，同时让这些插件化模块能够正常运行。安卓四大组件（Activity、Service、Broadcast Receiver、Content Provider）插件化都会预先注册代理组件，然后利用这个代理组件向系统发送数据请求，接着利用借尸还魂的方式还原成真正要启动的组件。在插件化的过程中利用 HOOK 系统各类关键的 API 和 Binder 来获取系统的资源以及管理各插件中组件的生命周期。设计 APK 的解析模块，通过这个模块来解析出 APK 中的关键信息，其中 META.INF 下的签名文件对于判断插件模块的安全性起到了关键性的作用。对于保障应用安全方面，本文采用对比签名文件信息，验证签名是否一致和对 dex 文件加壳的方式来维护。为实现热修复，解决 mutildex 的问题，使用了 javassist 动态代码注入技术。插件化机制的总体框架如图 1 所示。

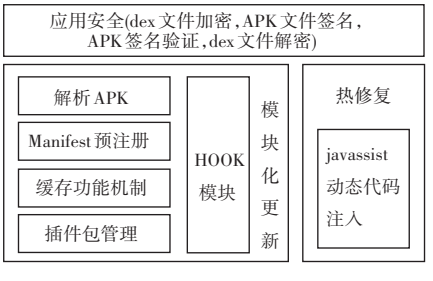

图 1 插件化机制总体框架

Fig. 1 Overall design of the plug-in mechanism

作者简介：熊建芳（1980-），女，硕士，讲师，主要研究方向：数据处理、软件应用。

收稿日期：2018-07-09

利用这个框架可以让应用在相对安全的情况下实现模块化更新和热修复功能,并且也可以让系统认为插件模块与宿主应用程序始终为一体。使用该机制开发的应用,APP 在用户体验方面将会有质的提升。

2 插件加载模块

由于在加载插件之前,从网络上获取的更新文件下载到本地的固定目录下。因此在加载插件时可以直接遍历这个固定目录,若该目录下存在符合条件的 APK 文件或 dex 文件,系统就会将这些文件加载进入内存。加载插件模块的流程如图 2 所示。

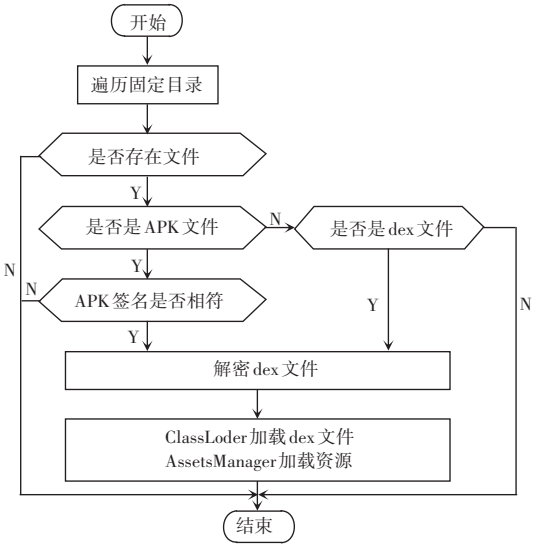

图 2 加载插件模块流程

Fig. 2 The flow of loading the plug-in module

3 模块化更新

3.1 APK 解析

普通的应用都是以 APK 包进行安装的,APK 在正常安装的过程中,系统会对 APK 进行自动解析,并且将信息缓存到系统中。在应用请求启动某个组件时,会自动匹配组件信息并初始化组件、启动组件。但本文中的 APK 并不能通过正常方式进行安装,因此必须对 Android 系统这一过程进行 HOOK,然后执行本文的逻辑,完成这一过程。在前面提到需要对 dex 文件进行保护并且验证 APK 签名,因此可以在该模块中利用解析 APK 文件的过程获取 dex 文件和签名文件,并对签名文件进行验证和对 dex 文件进行解密。

通过上面的分析,APK 包的解析将是本文研究机制中必须设计的一个模块。当有新的插件模块下载到本地固定目录时,该模块就会自动解析出对应的信息,并且将其放入系统的缓存目录下。通过对应用安装过程源码的分析,可以得出 APK 的解析是由 PackageParser 这个类来负责的,因此本文通过继承该类,实现自己的逻辑来解析插件的 APK 包。

3.2 AndroidManifest 预注册

想要启动插件中的四大组件势必需要在宿主程序中进行预注册。对于插件模块中四大组件的数量在开发宿主应用时并不可预测,但是因为系统本身的限制,能同时运行的进程数量是有限的,同时运行的插件数量也就有限,因此本文对四大组件中代理组件的注册数就可有限化。

为了让每个插件都能够独立的运行在自己的进程中,必须对预注册组件的 process 属性进行赋值,用来区分不同的进程。因为开发 APP 时,在 AndroidManifest 中注册的组件可以通过指定 process 属性来指定该组件所在的进程。对于 Activity 而言其启动模式有 4 种,除了由于 Standard 模式每次启动都是新的 Activity,在一个插件中只需为其预先注册一个 Activity,其余模式下启动的 Activity 则必须预注册多个。广播也具有一定的特殊性,具有 2 种注册方式,本文采取的方式是静态转动态注册,所以并不需要对其进行预注册。对于其它的 2 个组件并不像 Activity 的生命周期和启动模式那样复杂,只需在每个插件中预注册一个代理组件。

3.3 四大组件的插件化

由于插件模块中的组件并未在宿主应用的文件中进行注册,所以必须使用上述预先注册的四大组件作为代理,利用代理组件向系统发出相应的请求。在获得应答后再还原成插件中对应的目标组件,在此过程中本文利用 HOOK 技术结合动态代理进行拦截和替换。四大组件拦截替换过程如图 3 所示。

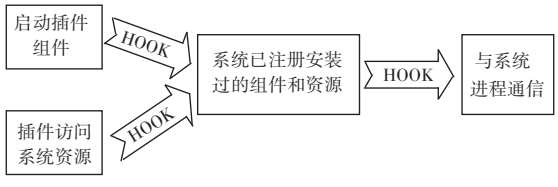

图 3 HOOK 模块设计

Fig. 3 HOOK module design

因为 Activity 启动模式复杂,在图中的拦截替换过程中,本机制会建立一个映射关系,方便系统在做任务栈管理时能够正确的识别目标组件,从而实现不同的启动模式。对于 Service 与 ContentProvider 的启动不像 Activity 那样复杂,本文只需将目标组件替换为代理组件,最后再进行代理分发还原。而对

于广播,有动态和静态之分,对于静态本文则将其转动态进行注册的。所以本文在 APK 进行解析时,将获取的广播信息通过 HOOK 的方式加入到动态广播的队列当中,然后当广播被发送后,系统会对该队列中对该广播感兴趣的接收者回调相应的方法。

3.4　缓存机制与插件包管理

Android 应用的沙箱机制,是 Android 系统为了保证应用安全专门设计的[2]。这种安全机制为每个成功安装的 Android 应用分配一个以应用包名为路径的私有空间,程序中很多本地存储都会选择该路径下的私有空间来进行数据的存储。由于插件并没有进行安装注册,所以系统并不会为其分配这样的空间。而插件本身肯定会遇到需要将数据存储到本地的境况,因此本文通过结合 HOOK 技术与 Java 的反射机制来获取应用程序的 IO 路径,在宿主程序私有空间下为插件创建一个空间。然后重定向到这个 IO 路径。获取到 IO 路径后,插件就可以进行本地存储。

插件 APK 解析完成后,将解析出来的信息缓存到本地目录,以便应用加载插件,同时也需要将一些关键的对象进行缓存,例如加载插件的 ClassLoder。在插件进行卸载的同时,需要根据插件的包名找到对应的本地目录,将该插件在本地的所有缓存清空。

3.5　HOOK 模块

HOOK 模块主要是为系统在通信过程中插件模块没有进行安装和注册,导致无法正常访问系统资源或是加载组件而设计的。本文通过 HOOK 技术和动态代理技术来解决这个问题,从而保障插件模块的正常运行。这项技术主要分为以下 2 点:一是用于 PMS 和 AMS。应用的启动和运行都与 PMS 和 AMS 相关,尤其是应用与系统进程进行通信,更是与 AMS 息息相关,因此 HOOK PMS 和 AMS 是至关重要的;二是 Binder。Android 中进程间通信使用最广泛的方式就是 Binder,请求系统服务和获取系统权限等都需要 Binder。因此 HOOK Binder 对于启动插件也是不可或缺的。HOOK 模块的设计如图 3 所示。

Package Manager Service(下文简称 PMS)在系统中的作用是完成信息校验、APK 信息获取和四大组件信息获取等重要功能。在 APK 解析、系统启动组件等过程中系统都会检查对应包的信息,因为插件并没有进行安装注册,就会抛出异常。因此要正常启动插件就必须对 PMS 进行 HOOK。从源码中可以看到真正获取 PMS 的方法是应用 ContextImpl 中的 getPackageManager( )。从该方法中可以看到 PMS 对象是在 ActivityThread 中并被 ApplicationPackageManager 包裹了一层。由 ActivityThread 源码中可以看出 PMS 的代理对象是一个静态对象,可以对这个对象进行 HOOK。因此可以将此作为一个 HOOK 点。

Ctivity Manager Service(下文简称 AMS)。AMS 对于 FrameWork 层的作用是非常重要的,四大组件启动运行都和 AMS 有着密切的关系。四大组件都是通过调用 AMS 与系统进程通信的。因此,HOOK AMS 显得尤为关键。通过查看 startActivity( ) 的源码,启动 Activity 最终都是 execStartActivity( ) 方法。这个方法存在于 Instrumentation 类中。从该方法中能够发现其实 ActivityManagerNative 是 AMS 的一个远程代理对象,这个代理对象是一个单例。因此本文将此作为一个 HOOK 点,对 AMS 的这个代理对象进行 HOOK。

本框架研究的机制主要是插件请求系统服务或资源,是以 Client 端的身份进行 HOOK Binder 的。因此,本文所要 HOOK 的对象是 Binder 的代理对象。通过对系统源码的分析可知获取系统的服务需要如下 2 个步骤:

```
//获取原始 IBinder 对象
IBinder b = ServiceManager.getService("service_name");
//转换为 Service 接口
IXXInterface in = IXXInterface.Stub.asInterface(b);
//asInterface( )方法中返回的对象。
android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
```

当插件化机制需要访问对应的系统服务、获取求权限、访问系统资源时,都需要用到 HOOK 模块,这是插件化的核心。

3.6　热修复模块

当应用出现需要修复的 bug 并需要立即更新时,本文会将需要更新的文件打包成 dex 文件,而不再是一个 APK 包。因此热修复文件不需要经过机制中的 APK 解析模块,可以直接进行加载。新类与 bug 类的包名、类名必须一致,且新类在数组中的位置又必须优先于应用中的 bug 类。根据类加载机制的双亲委托模型可知,系统将不再加载 bug 类。同时为了解决 dex 文件的引用问题,本文在加载 dex 文件的过程中需要进行动态代码的注入。图 4 是热

修复的流程。

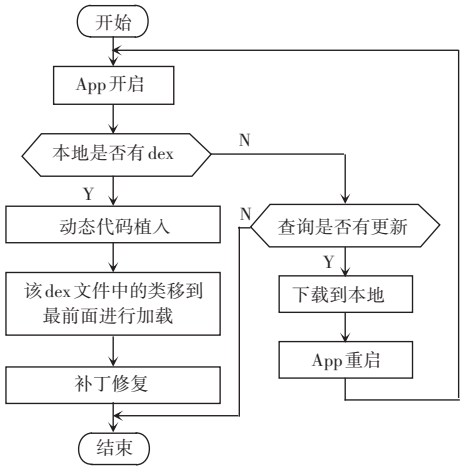

图 4 热修复流程

Fig. 4 The flow of hotfix

在 Java 层面通常用到的动态特性是反射,但是反射的效率比较低,并且在 java 层面使用动态代码注册,也不可能实现一个类去引用其它 dex 文件的类。因此为实现上面的需求,本文使用 javassist 类库进行代码注入。利用 Javassist 类库中提供的方法直接在运行时操作 Java 字节码。相比于其它动态注入代码的方式,javassist 的性能虽然略低,但是提供了一层抽象,相比于直接操作字节码的方式 javassitst 源码级别的 api 成本要低,本文选择使用 javassist 来进行代码的注入。在宿主程序开发完毕后,本文的机制要求去打包第二个 dex 文件,这个 dex 文件的目的是让类去引用不同 dex 文件中的类,避免被打上特殊的标志。在宿主应用程序中利用 javassist 类库编写相应的代码对需要代码注入的类进行动态注入。

3.7 插件化机制安全模块

采用验证 APK 签名文件和对 dex 文件在打包时进行加密、加载前进行解密这两种方式维护应用的安全。

4 结束语

本文所研究的机制具有以下特点:一是可以在用户无感知的情况下下载更新模块、对新功能模块更新和修复类文件中的 bug;二是每个插件模块都是独立的,插件之间互不影响,并且可以进行热插拔;三是热修复模块与模块化更新结合,既实现了大粒度的模块更新,又实现了小粒度的热修复功能;四是具有一定的安全机制,可以在一定程度上保障应用的安全。

参考文献

[1] 葛志忠. Android 应用搜索的设计与实现[D]. 上海:复旦大学,2014.

[2] 刘瑞斌,唐敏,宁学军. 基于 Android 的应用插件化实现方案:中国, CN103399792A[P].2013-11-20.

[3] 谢晋. 基于 Android 的阿里巴巴移动客户端的设计与实现[D]. 哈尔滨:哈尔滨工业大学,2012.

[4] 袁向英. 基于 Android 系统的数据库开发和插件技术的应用开发[J]. 电脑编程技巧与维护,2014(2):33-35.

[5] 丁丽萍. Android 操作系统的安全性分析[J]. 信息网络安全,2012(3):28-31,41.

[6] ARNOLD K, GOSLING J, HOLMES D. Java(TM) Programming Language[M]. 4th ed. USA:Addison-Wesley Professional, 2005.

[7] 清宏计算机工作室. JAVA 编程技巧[M]. 北京:机械工业出版社.2004.

[8] MOIRM, SHAVIT N. Concurrent data structures, in handbook of data structures and applications[M]. USA: Chapman and Hall/CRC Press,2004.