浙江大學

硕 士 研 究 生 读 书 报 告

题目 Android 的插件化开发原理

作者姓名 郭大魁

作者学号 21651151

指导教师 李启雷

学科专业 软件工程

所在学院 软件学院

提交日期 二○一七年一月

the Principle of Android Plugin development

A Dissertation Submitted to

Zhejiang University

in partial fulfillment of the requirements for

the degree of

Master of Engineering

Major Subject: Software Engineering

Advisor: Li Qilei

By

Guo Dakui

Zhejiang University, P.R. China

2017

摘要

本文重点探讨了 Android 开发中一个至关重要的问题——Android 的插件化开发。在 Android 项目中采用了动态加载技术，主要的目的是为了达到让用户不用重新安装 APK 就能升级应用的功能，这样一来不仅大大提高了应用新版本的覆盖率，也减少了服务器对旧版本接口兼容性的压力，同时可以快速修复一些线上的 BUG。所以插件化开发对于 Android 项目来说就显得非常的重要，本文重点介绍当前流行的 Android 插件化开发的技术方法以及这些技术背后的原理，介绍一些实用的 Android 插件化开发框架。

关键词：Android， 插件化开发，动态加载技术

Abstract

This paper discusses the development of Android is a crucial issue——Android Plugin Development. The dynamic loading in the Android project, the main purpose is to let users do not have to re install the APK will be able to upgrade the application function, thus not only greatly improve the application of the new version of the coverage, but also reduce the pressure on the old version of the server interface compatibility, and can quickly repair some of the online BUG. So the plugin development for the Android project is very important. The principle behind Android plugin development techniques this paper focuses on the current and the technology, introduces some practical Android plugin development framework.

Keywords：Android, the plugin development, dynamic loading technique

1 引言

随着 Android 项目的越来越大，互联网的迅速发展，业务迭代占据了大量的时间，这导致了 Android APP 的业务逻辑将变化的很快，而且有很多新功能的不断的出现。而现在 APP 的发布要经过签名、打包、发布到应用市场、用户下载安装等等一系列流程，这导致了 APP 退出新版本到用户实际使用的成本非常的高，这就与互联网迅速发展局势形成了矛盾，APP 的发布到安装的高成本无意成为了一个瓶颈。

我们知道，APP 必须安装才能运行，如果不安装要是也能运行该多好呀。事实上，这不是完全不可能的，尽管它比较难实现。这就设计到了 Android 的动态加载的技术和 Android 项目的插件化开发，如果我们把 APP 的各个模块做成插件，通过宿主 APK 可以动态的加载未安装的 APK，实现 Android 项目的热插拔，从而来动态的更新升级 APP，和修复一些线上的 BUG。

2 技术背景

通过服务器配置一些参数，Android APP 获取这些参数再做出相应的逻辑，这是常用的事情。

比如现在大部分 APP 都有一个启动页面，如果到了一些重要的节日，APP 的服务器会配置一些与节日相关的图片，APP 启动时候再把原始的启动图片换成这些新的图片，这样就能提高用户的体验了。

再则，早期个人开发者在安卓市场上发布应用的时候，如果应用里面包含广告，那么有可能审核不通过。那么就可以通过服务器配置一个开关，审核应用的时候先把开关关闭，这样应用就不会显示广告；安卓市场审核通过之后，再把服务器的广告开关给打开，以这样的手段规避市场的审核。

道高一尺魔高一丈。安卓市场开始扫描 APK 中的 Manifest 甚至是 dex 文件，查看开发者的 APK 包里是否有广告的代码，如果有就审核不通过。

通过服务器配置开关参数饭方法行不通了，开发者就开始想，既然这样，能不能先不要在 APK 写广告的代码，在用户运行 APP 的时候，再从服务器下载广告的代码，运行，在实现广告呢？答案是肯定的，这就是动态加载。动态加载就是在程序运行的时候，加载一些程序自身原本不存在的可执行文件并运行这些文件里面的逻辑，开起来就像应用从服务器下载了一些代码，然后去执行这些代码。当然，这技术也能应用于动态的更新和升级 app，修复线上 APP 的 BUG。

3 传统 PC 软件中的动态加载技术

动态加载技术在 PC 软件领域广泛使用，比如输入法的截图功能。刚刚安装好的输入法软件可能没有截图的功能，当你第一次使用的时候，输入法会先从服务器下载并安装截图软件，然后在执行截图功能。

此外，许多的 PC 软件的安装目录里面都有大量的 DLL 文件(Dynamic Link Library)，PC 软件则通过调用这些 DLL 里面的代码执行特定的功能，这就是一种动态加载技术。

对于 Java 程序，Java 的可执行文件是 Jar，运行在 JVM 上，JVM 通过 ClassLoader 加载 Jar 文件并执行里面的代码。所以说 Java 程序可以通过动态的调用 Jar 文件达到动态加载的目的。

4 Android 应用的动态加载技术

Android 应用类似于 Java 程序，虚拟机换成了 Dalvik/ART，而 Jar 换成了 Dex。在 Android APP 运行的时候，我们是不是也可以通过下载新的应用，或者通过外部调用外部的 Dex 文件来实现动态加载呢？

然而在 Android 上实现起来可没那么容易，如果下载一个新的 APK 下来，不安装这个 APK 的话可不能运行。如果让用户手动安装完这个 APK 再启动，那可不像是动态加载，纯粹就是用户安装了一个新的应用，然后再启动这个新的应用这种做法也叫做“静默安装”。

动态调用外部的 Dex 文件则是完全没有问题的。在 APK 文件中往往有一个或者多个 Dex 文件，我们写的每一句代码都会被编译到这些文件里面，Android 应用运行的时候就是通过执行这些 Dex 文件完成应用的功能的。虽然一个 APK 一旦构建出来，我们是无法更换里面的 Dex 文件的，但是我们可以通过加载外部的 Dex 文件来实现动态加载，这个外部文件可以放在外部存储，或者从网络下载。

6 Android 动态加载的类型

在介绍动态加载之前，我们应该先给动态加载技术做一个简单的定义，真正的加载加载应该是这样的：

1. 应用在运行的时候通过加载一些本地不存在的可执行文件实现一些特定的功能;
2. 这些可执行文件是可以替换的;
3. 更换静态资源（比如换启动图、换主题、或者用服务器参数开关控制广告的隐藏现实等）不属于动态加载;
4. Android 中动态加载的核心思想是动态调用外部的 dex 文件，极端的情况下，Android APK 自身带有的 Dex 文件只是一个程序的入口，或者说空壳，所有的功能都通过从服务器下载最新的 Dex 文件完成。

在 Android 项目中，动态加载技术按照加载的可执行文件的不同大致可以分

为两种：

1.动态加载 so 库；

2.动态加载 dex/jar/apk 文件（现在动态加载普遍说的是这种）；

其一，Android 中 NDK 中其实就使用了动态加载，动态加载.so 库并通过 JNI 调用其封装好的方法。后者一般是由 C/C++编译而成，运行在 Native 层，效率会比执行在虚拟机层的 Java 代码高很多，所以 Android 中经常通过动态加载.so 库来完成一些对性能比较有需求的工作（比如 T9 搜索、或者 Bitmap 的解码、图片高斯模糊处理等）。此外，由于 so 库是由 C/C++编译而来的，只能被反编译成汇编代码，相比中 dex 文件反编译得到的 Smali 代码更难被破解，因此 so 库也可以被用于安全领域。这里为后面要讲的内容提前说明一下，一般情况下我们是把 so 库一并打包在 APK 内部的，但是 so 库其实也是可以从外部存储文件加载的。

其二，“基于 ClassLoader 的动态加载 dex/jar/apk 文件”，就是我们上面提到的“在 Android 中动态加载由 Java 代码编译而来的 dex 包并执行其中的代码逻辑”，这是常规 Android 开发比较少用到的一种技术，目前网络上大多文章说到的动态加载指的就是这种。

Android 项目中，所有 Java 代码都会被编译成 dex 文件，Android 应用运行时，就是通过执行 dex 文件里的业务代码逻辑来工作的。使用动态加载技术可以在 Android 应用运行时加载外部的 dex 文件，而通过网络下载新的 dex 文件并替换原有的 dex 文件就可以达到不安装新 APK 文件就升级应用（改变代码逻辑）的目的。

7 Android 动态加载的过程

无论上面的哪种动态加载，其实基本原理都是在程序运行时加载一些外部的

可执行的文件，然后调用这些文件的某个方法执行业务逻辑。需要说明的是，因为文件是可执行的(so 库或者 dex 包，也就是一种动态链接库)，出于安全问题，Android 并不允许直接加载手机外部存储这类 noexec（不可执行）存储路径上的可执行文件。

对于这些外部的可执行文件，在 Android 应用中调用它们前，都要先把他们拷贝到 data/packagename/内部储存文件路径，确保库不会被第三方应用恶意修改或拦截，然后再将他们加载到当前的运行环境并调用需要的方法执行相应的逻辑，从而实现动态调用。

动态加载的大致过程就是：

1. 把可执行文件（.so/dex/jar/apk）拷贝到应用 APP 内部存储；
2. 加载可执行文件；
3. 调用具体的方法执行业务逻辑；

7.1 动态加载 so 库

动态加载 so 库应该就是 Android 最早期的动态加载了，不过 so 库不仅可以存放在 APK 文件内部，还可以存放在外部存储。Android 开发中，更换 so 库的情形并不多，但是可以通过把 so 库挪动到 APK 外部，减少 APK 的体积，毕竟许多 so 库文件的体积可是非常大的。

7.2 动态加载 dex/jar/apk 文件

为了更好的理解动态加载 dex/jar/apk 文件，我们首先需要一些相关的知识。

7.2.1 ClassLoader 与 dex

动态加载 dex/jar/apk 文件的基础是类加载器 ClassLoader，它的包路径是 java.lang，由此可见其重要性，虚拟机就是通过类加载器加载其需要用的 Class，

这是 Java 程序运行的基础。

早期使用过 Eclipse 等 Java 编写的软件的同学可能比较熟悉，Eclipse 可以加载许多第三方的插件（或者叫扩展），这就是动态加载。这些插件大多是一些 Jar 包，而使用插件其实就是动态加载 Jar 包里的 Class 进行工作。这其实非常好理解，Java 代码都是写在 Class 里面的，程序运行在虚拟机上时，虚拟机需要把需要的 Class 加载进来才能创建实例对象并工作，而完成这一个加载工作的角色就是 ClassLoader。

Android 的 Dalvik/ART 虚拟机如同标准 JAVA 的 JVM 虚拟机一样，在运行程序时首先需要将对应的类加载到内存中。因此，我们可以利用这一点，在程序运行时手动加载 Class，从而达到代码动态加载可执行文件的目的。Android 的 Dalvik/ART 虚拟机虽然与标准 Java 的 JVM 虚拟机不一样，ClassLoader 具体的加载细节不一样，但是工作机制是类似的，也就是说在 Android 中同样可以采用类似的动态加载插件的功能，只是在 Android 应用中动态加载一个插件的工作要比 Eclipse 加载一个插件复杂许多

7.2.2 简单的动态加载模式

理解 ClassLoader 的工作机制后，我们知道了 Android 应用在运行时使用 ClassLoader 动态加载外部的 dex 文件非常简单，不用覆盖安装新的 APK，就可以更改 APP 的代码逻辑。但是 Android 却很难使用插件 APK 里的 res 资源，这意味着无法使用新的 XML 布局等资源，同时由于无法更改本地的 Manifest 清单文件，所以无法启动新的 Activity 等组件。

不过可以先把要用到的全部 res 资源都放到主 APK 里面，同时把所有需要的 Activity 先全部写进 Manifest 里，只通过动态加载更新代码，不更新 res 资源，如

果需要改动 UI 界面 ,可以通过使用纯 Java 代码创建布局的方式绕开 XML 布局。同时也可以使用 Fragment 代替 Activity，这样可以最大限度得避开“无法注册新组件的限制”。

某种程度上，简单的动态加载功能已经能满足部分业务需求了，特别是一些早期的 Android 项目 ,那时候 Android 的技术还不是很成熟 ,而且早期的 Android 设备更是有大量的兼容性问题 ,只有这种简单的加载方式才能稳定运行。这种模式的框架比较适用一些 UI 变化比较少的项目 ,比如游戏 SDK ,基本就只有登陆、注册界面，而且基本不会变动，更新的往往只有代码逻辑。

7.2.3 代理 Activity 模式

简单加载模式还是不够用，所以代理模式出现了。从这个阶段开始就稍微有点“黑科技”的味道了，比如我们可以通过动态加载，让现在的 Android 应用启动一些“新”的 Activity，甚至不用安装就启动一个“新”的 APK。宿主 APK[2]需要先注册一个空壳的 Activity 用于代理执行插件 APK 的 Activity 的生命周期。

主要有以下特点：

1.宿主 APK 可以启动未安装的插件 APK；

2.插件 APK 也可以作为一个普通 APK 安装并且启动；

3.插件 APK 可以调用宿主 APK 里的一些功能；

4.宿主 APK 和插件 APK 都要接入一套指定的接口框架才能实现以上功能。

同时也主要有一下几点限制：

1.需要在 Manifest 注册的功能都无法在插件实现 ,比如应用权限、LaunchMode、静态广播等；

2.宿主一个代理用的 Activity 难以满足插件一些特殊的 Activity 的需求 ,插件

Activity 的开发受限于代理 Activity；

3.宿主项目和插件项目的开发都要接入共同的框架，大多时候，插件需要依附宿主才能运行，无法独立运行。

7.2.4 动态创建 Activity 模式

我们在代理 Activity 模式里谈到启动插件 APK 里的 Activity 的两个难题吗，由于插件里的 Activity 没在主项目的 Manifest 里面注册，所以无法经历系统 Framework 层级的一系列初始化过程，最终导致获得的 Activity 实例并没有生命周期和无法使用 res 资源。

使用代理 Activity 能够解决这两个问题，但是有一些限制

1.实际运行的 Activity 实例其实都是 ProxyActivity，并不是真正想要启动的 Activity；

2.ProxyActivity 只能指定一种 LaunchMode，所以插件里的 Activity 无法自定义 LaunchMode；

3.不支持静态注册的 BroadcastReceiver；

4.往往不是所有的 apk 都可作为插件被加载，插件项目需要依赖特定的框架，还有需要遵循一定的"开发规范"。

总不能把插件 APK 所有的 Activity 都事先注册到主项目里面吧，想到代理模式需要注册一个代理的 ProxyActivity，那么能不能在主项目里注册一个通用的 Activity（比如 TargetActivity）给插件里所有的 Activity 用呢？解决对策就是，在需要启动插件的某一个 Activity（比如 PlugActivity）的时候，动态创建一个 TargetActivity，新创建的 TargetActivity 会继承 PlugActivity 的所有共有行为，而这个 TargetActivity 的包名与类名刚好与我们事先注册的 TargetActivity 一致，我

们就能以标准的方式启动这个 Activity。

运行时动态创建并编译一个 Activity 类，这种想法不是天方夜谭，动态创建类的工具有 dexmaker 和 asmdex，二者均能实现动态字节码操作，最大的区别是前者是创建 dex 文件，而后者是创建 class 文件。

动态类创建的方式 ,使得注册一个通用的 Activity 就能给多给 Activity 使用，对这种做法存在的问题也是明显的：

1.使用同一个注册的 Activity ,所以一些需要在 Manifest 注册的属性无法做到每个 Activity 都自定义配置；

2.插件中的权限，无法动态注册，插件需要的权限都得在宿主中注册，无法动态添加权限；

3.插件的 Activity 无法开启独立进程，因为这需要在 Manifest 里面注册；

其中不稳定的问题出现在对 Service 的支持上，使用动态创建类的方式可以搞定 Activity 和 Broadcast Receiver，但是使用类似的方式处理 Service 却不行，因为“ContextImpl.getApplicationContext” 期待得到一个非 ContextWrapper 的 context，如果不是则继续下次循环，目前的 Context 实例都是 wrapper，所以会进入死循环。

代理 Activity 模式与动态创建 Activity 模式的区别是什么呢，简单的说，最大的不同是代理模式使用了一个代理的 Activity，而动态创建 Activity 模式使用了一个通用的 Activity。

代理模式中，使用一个代理 Activity 去完成本应该由插件 Activity 完成的工作，这个代理 Activity 是一个标准的 Android Activity 组件，具有生命周期和上下文环境（ContextWrapper 和 ContextCompl），但是它自身只是一个空壳，并没有

承担什么业务逻辑；而插件 Activity 其实只是一个普通的 Java 对象，它没有上下文环境，但是却能正常执行业务逻辑的代码。代理 Activity 和不同的插件 Activity 配合起来，就能完成不同的业务逻辑了。所以代理模式其实还是使用常规的 Android 开发技术，只是在处理插件资源的时候强制调用了系统的隐藏 API，因此这种模式还是可以稳定工作和升级的。

动态创建 Activity 模式，被动态创建出来的 Activity 类是有在主项目里面注册的，它是一个标准的 Activity，它有自己的 Context 和生命周期，不需要代理的 Activity。

8.作用与代价

凡事都有两面性，特别是这种 非官方支持 的 非常规 开发方式，在采用前一定要权衡清楚其作用与代价。如果决定了要采用动态加载技术，个人推荐可以现在实际项目的一些比较独立的模块使用这种框架，把遇到的一些问题解决之后，再慢慢引进到项目的核心模块；如果遇到了一些无法跨越的问题，要有能够迅速投入生产的替代方案。

8.1 作用

1.规避 APK 覆盖安装的升级过程，提高用户体验，顺便能 规避 一些安卓市场的限制；

2.动态修复应用的一些 紧急 BUG，做好最后一道保障；

3.当应用体积太庞大的时候，可以把一些模块通过动态加载以插件的形式分割出去，这样可以减少主项目的体积，提高项目的编译速度，也能让主项目和插件项目并行开发；

4.插件模块可以用懒加载的方式在需要的时候才初始化，从而 提高应用的启

动速度；

5.从项目管理上来看，分割插件模块的方式做到了 项目级别的代码分离，大大降低模块之间的耦合度，同一个项目能够分割出不同模块在多个开发团队之间 并行开发，如果出现 BUG 也容易定位问题；

6.在 Android 应用上 推广 其他应用的时候，可以使用动态加载技术让用户优先体验新应用的功能，而不用下载并安装全新的 APK；

7.减少主项目 DEX 的方法数，65535 问题 彻底成为历史。

8.2 代价

1.开发方式可能变得比较诡异、繁琐，与常规开发方式不同；

2.随着动态加载框架复杂程度的加深，项目的构建过程也变得复杂，有可能要主项目和插件项目分别构建，再整合到一起；

3.由于插件项目是独立开发的，当主项目加载插件运行时，插件的运行环境已经完全不同，代码逻辑容易出现 BUG，而且在主项目中调试插件十分繁琐；

4.非常规的开发方式，有些框架使用反射强行调用了部分 Android 系统 Framework 层的代码，部分 Android ROM 可能已经改动了这些代码，所以有存在兼容性问题的风险，特别是在一些古老 Android 设备和部分三星的手机上；

参考文献

[1]Activity 开发艺术探索．任玉刚 电子工业出版社，2015．

[2] https://segmentfault.com/a/1190000004077469，Android 动态加载黑科技 动态创建 Activity 模式

[3] http://blog.csdn.net/u013478336/article/details/50734108，Android 动态加载 dex 技术初探

[4] http://blog.csdn.net/singwhatiwanna/article/details/40283117，DL 动态加载框架技术文档