

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](https://github.com/crittercism/dexmaker" \t "_blank)和[asmdex](http://asm.ow2.org/" \t "_blank)，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