网址:

https://segmentfault.com/a/1190000004062866

# [Android动态加载技术 简单易懂的介绍方式](https://segmentfault.com/a/1190000004062866)

基本信息

* Author：[kaedea](https://github.com/kaedea)
* GitHub：[android-dynamical-loading](https://github.com/kaedea/android-dynamical-loading)

我们很早开始就在Android项目中采用了动态加载技术，主要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（特别是 SDK项目），这样一来不但可以大大提高应用新版本的覆盖率，也减少了服务器对旧版本接口兼容的压力，同时如果也可以快速修复一些线上的BUG。

这种技术并不是常规的Android开发方式，早期并没有完善的解决方案。从“不明觉厉”到稳定投入生产，一直以来我总想对此编写一些文档，这也是这篇日志的由来，没想到前前后后竟然拖沓着编辑了一年多，所以日志里有的地方思路可能有点衔接得不是很好，如果有修正建议请直接回复。

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，这是常有的事情。

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时节相关的图片，APP启动时候再把原有的启动图换成这些新的图片，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，如果应用里包含有广告，那么有可能会审核不通过。那么就通过在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了；安卓市场审核通过后，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里面的Manifest甚至dex文件，查看开发者的APK包里是否有广告的代码，如果有就有可能审核不通过。

通过服务器怕配置开关参数的方法行不通了，开发者们开始想，“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，再现实广告呢？”。答案是肯定的，这就是动态加载：

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的代码逻辑。

看起来就像是应用从服务器下载了一些代码，然后再执行这些代码！

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，然后再执行截图功能。

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软件则是通过调用这些DLL里面的代码执行特定的功能的，这就是一种动态加载技术。

熟悉Java的同学应该比较清楚，Java的可执行文件是Jar，运行在虚拟机上JVM上，虚拟机通过ClassLoader加载Jar文件并执行里面的代码。所以Java程序也可以通过动态调用Jar文件达到动态加载的目的。

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。

动态加载的定义

开始正题之前，在这里可以先给动态加载技术做一个简单的定义。真正的动态加载应该是

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;

Android动态加载的类型

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（后面我们谈到“动态加载”如果没有特别指定，均默认是这种）。

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。同时，使用动态加载技术，一般来说会使得Android开发工作变得更加复杂，这中开发方式不是官方推荐的，不是目前主流的Android开发方式，**Github** 和 **StackOverflow** 上面外国的开发者也对此不是很感兴趣，外国相关的教程更是少得可怜，目前只有在大天朝才有比较深入的研究和应用，特别是一些SDK组件项目和 **BAT家族** 的项目上，Github上的相关开源项目基本是国人在维护，偶尔有几个外国人请求更新英文文档。

Android动态加载的大致过程

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；

以下分别对这两种动态加载的实现方式做比较深入的介绍。

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。

详细的应用方式请参考后续日志 [Android动态加载补充 加载SD卡的SO库](http://segmentfault.com/a/1190000004062899)。

动态加载 **dex/jar/apk文件**

我们经常讲到的那种Android动态加载技术就是这种，后面我们谈到“动态加载”如果没有特别指定，均默认是这个。

基础知识：类加载器ClassLoader和dex文件

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。

关于类加载器ClassLoader的工作机制，请参考 [Android动态加载基础 ClassLoader的工作机制](http://segmentfault.com/a/1190000004062880)。

现在网上有多种基于ClassLoader的Android动态加载的开源项目，大部分核心思想都殊途同归，按照复杂程度以及具体实现的框架，大致可以分为以下三种形式，或者说模式 [1]。

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（做过Android1.6兼容的同学可能深有体会），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。

详细的应用方式请参考后续日志 [Android动态加载入门 简单加载模式](http://segmentfault.com/a/1190000004062952)。

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；

详细的应用方式请参考后续日志 [Android动态加载进阶 代理Activity模式](http://segmentfault.com/a/1190000004062972)。

代理Activity模式的核心在于“使用宿主的一个代理Activity为插件所有的Activity提供组件工作需要的环境”，随着代理模式的逐渐成熟，现在还出现了“使用Hack手段给插件的Activity注入环境”的模式，这里暂时不展开，以后会继续分析。

我们目前有投入到生产中的开发方式只有简单模式和代理模式，在设计的前期遇到不少兼容性的问题，不过好在Android 4.0以后的机型上就比较少了。

动态创建Activity模式

天了噜，到了这个阶段就真的是“黑科技”的领域了，从而使其可以正常运行。可以试想“从网络下载一个Flappy Bird的APK，不用安装就直接运行游戏”，或者“同时运行两个甚至多个微信”。

动态创建Activity模式的核心是“运行时字节码操作”，现在宿主注册一个不存在的Activity，启动插件的某个Activity时都把想要启动的Activity替换成前面注册的Activity，从而是后者能正常启动。

这个模式有以下特点：

1. 主APK可以启动一个未安装的插件APK；
2. 插件APK可以是任意第三方APK，无需接入指定的接口，理所当然也可以独立运行；

详细的应用方式请参考后续日志 [Android动态加载黑科技 动态创建Activity模式](http://segmentfault.com/a/1190000004077469)。

为什么我们要使用动态加载技术

说实话，作为开发我们也不想使用的，这是产品要求的！（警察蜀黍就是他，他只问我能不能实现，并木有问我实现起来难不难……好吧我们知道他们也没得选。）

Android开发中，最先使用动态加载技术的应该是SDK项目吧。现在网上有一大堆Android SDK项目，比如Google的Goole Play Service，向开发者提供支付、地图等功能，又比如一些Android游戏市场的SDK，用于向游戏开发者提供账号和支付功能。和普通Android应用一样，这些SDK项目也是要升级的，比如现在别人的Android应用里使用了我们的SDK1.0版本，然后发布到安卓市场上去。现在我们发现SDK1.0有一些紧急的BUG，所以升级了一个SDK1.1版本，没办法，只能让人家重新接入1.1版本再发布到市场。万一我们有SDK1.2、1.3等版本呢，本来让人家每个版本都重新接入也无可厚非，不过产品可关心体验啊，他就会问咯，“虽然我不懂技术，但是我想知道有没有办法，能让人家只接入一次我们的SDK，以后我们发布新的SDK版本的时候他们的项目也能跟着自动升级？”，答曰，“有，使用动态加载的技术能办到，只不过（开发工作量会剧增…）”，“那就用吧，我们要把产品的体验做到极致”。

好吧，我并没有黑产品的意思，现在团队的产品也不错，不过与上面类似的对话确实发生在我以前的项目里。这里提出来只是为了强调一下Android项目中采用动态加载技术的 **作用** 以及由此带来的 **代价**。

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。

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（虽然现在在Android Studio中很容易开启MultiDex，这个问题也不难解决）；

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；
5. 采用动态加载的插件在使用系统资源（特别是Theme）时经常有一些兼容性问题，特别是部分三星的手机；

其他动态修改代码的技术

上面说到的都是基于ClassLoader的动态加载技术（除了加载SO库外），使用ClassLoader的一个特点就是，如果程序不重新启动，加载过一次的类就无法重新加载。因此，如果使用ClassLoader来动态升级APP或者动态修复BUG，都需要重新启动APP才能生效。

除了使用ClassLoader外，还可以使用jni hook的方式修改程序的执行代码。前者是在虚拟机上操作的，而后者做的已经是Native层级的工作了，直接修改应用运行时的内存地址，所以使用jni hook的方式时，不用重新应用就能生效。

目前采用jni hook方案的项目中比较热门的有阿里的dexposed和AndFix，有兴趣的同学可以参考 [各大热补丁方案分析和比较](http://blog.zhaiyifan.cn/2015/11/20/HotPatchCompare/)。

动态加载开源项目

* [ACDD](https://github.com/bunnyblue/ACDD)
* [DL dynamic-load-apk](https://github.com/singwhatiwanna/dynamic-load-apk)
* [android-pluginmgr](https://github.com/houkx/android-pluginmgr)
* [Direct-Load-apk](https://github.com/FinalLody/Direct-Load-apk)
* [360 DroidPlugin](https://github.com/Qihoo360/DroidPlugin)
* [携程网 DynamicAPK](https://github.com/CtripMobile/DynamicAPK)
* [女娲 Nuwa](https://github.com/jasonross/Nuwa)
* [Android-Plugin-Framework](https://github.com/limpoxe/Android-Plugin-Framework)

脚注

[1] 其实也说不上什么模式，这不过这些动态加载的开发方式都有自己明显的特征，所以姑且用“形式或者模式”来称呼好了。

[2] 为了方便区分概念，阐述一些术语：  
宿主：Host，主项目APK、主APK，也就是我们希望采用动态加载技术的主项目；  
插件：Plugin，可以是dex、jar或者apk文件，从主项目分离开来，我们能通过动态加载加载到主项目里面来的模块，一个主APK可以同时加载多个插件APK；

* [2015年11月29日发布](https://segmentfault.com/a/1190000004062866)