[Android动态加载入门 简单加载模式](https://segmentfault.com/a/1190000004062952)

https://segmentfault.com/a/1190000004062952

## 基本信息

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

## 初步了解Android动态加载

Java程序中，JVM虚拟机是通过类加载器ClassLoader加载.jar文件里面的类的。Android也类似，不过Android用的是Dalvik/ART虚拟机，不是JVM，也不能直接加载.jar文件，而是加载dex文件。

先要通过Android SDK提供的DX工具把.jar文件优化成.dex文件，然后Android的虚拟机才能加载。注意，有的Android应用能直接加载.jar文件，那是因为这个.jar文件已经经过优化，只不过后缀名没改（其实已经是.dex文件）。

如果对ClassLoader的工作机制有兴趣，具体过程请参考 [Android 动态加载基础 ClassLoader工作机制](http://segmentfault.com/a/1190000004062880)，这里不再赘述。

## 如何获取能够加载的.dex文件

首先我们可以通过JDK的编译命令javac把Java代码编译成.class文件，再使用jar命令把.class文件封装成.jar文件，这与编译普通Java程序的时候完全一样。

之后再用Android SDK的DX工具把.jar文件优化成.dex文件（在“android-sdk\build-tools\具体版本\”路径下）

dx --dex --output=target.dex origin.jar // target.dex就是我们要的了

此外，我们可以现把代码编译成APK文件，再把APK里面的.dex文件解压出来，或者直接把APK文件当成.dex使用（只是APK里面的静态资源文件我们暂时还用不到）。至此我们发现，无论加载.jar，还是.apk，其实都和加载.dex是等价的，Android能加载.jar和.apk，是因为它们都包含有.dex，直接加载.apk文件时，ClassLoader也会自动把.apk里的.dex解压出来。

## 加载并调用****.dex****里面的方法

与JVM不同，Android的虚拟机不能用ClassCload直接加载.dex，而是要用DexClassLoader或者PathClassLoader,他们都是ClassLoader的子类，这两者的区别是

1. DexClassLoader：可以加载jar/apk/dex，可以从SD卡中加载未安装的apk；
2. PathClassLoader：要传入系统中apk的存放Path，所以只能加载已经安装的apk文件；

使用前，先看看DexClassLoader的构造方法

public DexClassLoader(String dexPath, String optimizedDirectory, String libraryPath, ClassLoader parent) {

super((String)null, (File)null, (String)null, (ClassLoader)null);

throw new RuntimeException("Stub!");

}

注意，我们之前提到的，DexClassLoader并不能直接加载外部存储的.dex文件，而是要先拷贝到内部存储里。这里的dexPath就是.dex的外部存储路径，而optimizedDirectory则是内部路径，libraryPath用null即可，parent则是要传入当前应用的ClassLoader，这与ClassLoader的“双亲代理模式”有关。

实例使用DexClassLoader的代码

File optimizedDexOutputPath = new File(Environment.getExternalStorageDirectory().getAbsolutePath() + File.separator + "test\_dexloader.jar");// 外部路径

File dexOutputDir = this.getDir("dex", 0);// 无法直接从外部路径加载.dex文件，需要指定APP内部路径作为缓存目录（.dex文件会被解压到此目录）

DexClassLoader dexClassLoader = new DexClassLoader(optimizedDexOutputPath.getAbsolutePath(),dexOutputDir.getAbsolutePath(), null, getClassLoader());

到这里，我们已经成功把.dex文件给加载进来了，接下来就是如何调用.dex里面的代码，主要有两种方式。

### 使用反射的方式

使用DexClassLoader加载进来的类，我们本地并没有这些类的源码，所以无法直接调用，不过可以通过反射的方法调用，简单粗暴。

DexClassLoader dexClassLoader = new DexClassLoader(optimizedDexOutputPath.getAbsolutePath(), dexOutputDir.getAbsolutePath(), null, getClassLoader());

Class libProviderClazz = null;

try {

libProviderClazz = dexClassLoader.loadClass("me.kaede.dexclassloader.MyLoader");

// 遍历类里所有方法

Method[] methods = libProviderClazz.getDeclaredMethods();

for (int i = 0; i < methods.length; i++) {

Log.e(TAG, methods[i].toString());

}

Method start = libProviderClazz.getDeclaredMethod("func");// 获取方法

start.setAccessible(true);// 把方法设为public，让外部可以调用

String string = (String) start.invoke(libProviderClazz.newInstance());// 调用方法并获取返回值

Toast.makeText(this, string, Toast.LENGTH\_LONG).show();

} catch (Exception exception) {

// Handle exception gracefully here.

exception.printStackTrace();

}

### 使用接口的方式

毕竟.dex文件也是我们自己维护的，所以可以把方法抽象成公共接口，把这些接口也复制到主项目里面去，就可以通过这些接口调用动态加载得到的实例的方法了。

pulic interface IFunc{

public String func();

}

// 调用

IFunc ifunc = (IFunc)libProviderClazz;

String string = ifunc.func();

Toast.makeText(this, string, Toast.LENGTH\_LONG).show();

到这里，我们已经成功从外部路径动态加载一个.dex文件，并执行里面的代码逻辑了。通过从服务器下载最新的.dex文件并替换本地的旧文件，就能初步实现“APP的动态升级了”。

## 如何动态更改XML布局

虽然已经能动态更改代码逻辑了，但是UI界面要怎么更改啊？Android开发中大部分的情况下，UI界面都是通过XML布局实现的，放在res目录下，可是.dex库里面并没有这些静态资源啊，所以无法改变XML布局。（这里即使直接动态加载APK文件，但是通过DexClassLoader只能加载新的APK其中的.dex文件，并无法加载其中的res资源文件，所以如果在动态加载的.dex中直接使用新的APK的res资源的话会抛出异常。）

大家都知道，所有的XML布局在运行的时候都要通过LayoutInflator渲染成View的实例，这个实例与我们使用纯Java代码创建的View实例几乎是等价的，而且后者可能效率还更高，所有的XML布局实现的UI界面都有等价的纯代码的创建方案。由此伸展开来，res目录下所有XML资源都有等价的纯代码的实现方式，比如XML动画、XML Drawable等。

所以，如果想要动态更改应用的UI界面的话，可以通过用纯代码创建布局的形式来解决。此外，还可以模仿LayoutInflator的工作方式，自己写一套布局解析器来解析XML文件，这样就能在完全不依赖res资源的情况下创建UI界面了，当然这样的工作量不少，而且，完全避开res资源的话，所有的分辨率、国际化等自适应问题都要自己在应用层写代码维护了，显然脱离res资源框架不是一个很明智的做法，**但是这种做法确实可行**，在我们之前的实际生产中的项目中也稳定使用着，这里出于责任问题就不方便公开细节了。

（说实在，这种方案非常繁琐，不好维护，一方面，这是产品一句“技术可行就做呗”而产生的解决方案；另一方面，但是动态加载技术还很不成熟，也没有什么实际投入到生产的项目，所以采取了非常保守的开发方式）。

## 使用Fragment代替Activity

Activity需要在Manifest里注册，然后一标准的Intent启动才会具有生命周期，很明显，如果想要动态加载的.dex里的Activity没有注册的话，是无法启动的。

有一种简单粗暴的做法就是可以把.dex里所有需要用到的Activity都事先注册到原项目里，不过这样一来如果.dex里的Activity有变化，原项目就必须跟着升级。

另外一种方案是使用Fragment，Fragment自带生命周期，不需要在Manifest里注册，所以可以在.dex里使用Fragment来代替Activity，代价就是Fragment之间的切换会繁琐许多。

## ART模式的兼容性问题

当初我们开始设计动态加载方案的时候，还没有ART模式。随着Kitkat的发布以及ART模式的出现，我们开始担心“用DexClassLoader加载.dex文件”的方案会不会在ART模式上面存在兼容性问题。

其实，ART模式相比原来的Dalvik，会在安装APK的时候，使用Android系统自带的**dex2oat工具**把APK里面的.dex文件转化成OAT文件，OAT文件是一种Android私有ELF文件格式，它不仅包含有从DEX文件翻译而来的本地机器指令，还包含有原来的DEX文件内容。这使得我们无需重新编译原有的APK就可以让它正常地在ART里面运行，也就是我们不需要改变原来的APK编程接口。ART模式的系统里，同样存在DexClassLoader类，包名路径也没变，只不过它的具体实现与原来的有所不同，但是接口是一致的。

package dalvik.system;

import dalvik.system.BaseDexClassLoader;

import java.io.File;

public class DexClassLoader extends BaseDexClassLoader {

public DexClassLoader(String dexPath, String optimizedDirectory, String libraryPath, ClassLoader parent) {

super((String)null, (File)null, (String)null, (ClassLoader)null);

throw new RuntimeException("Stub!");

}

}

也就是说，ART模式在加载.dex文件的方法上，对Dalvik做了向下兼容，所以使用DexClassLoader加载进来的.dex文件同样也会被转化成OAT文件再被执行，“以DexClassLoader为核心的动态加载方案”在ART模式上可以稳定运行。

关于ART模式以及OAT文件的详细分析，请参考官方的[ART and Dalvik](https://source.android.com/devices/tech/dalvik/)，以及老罗的[Android ART运行时无缝替换Dalvik虚拟机的过程分析](http://blog.csdn.net/luoshengyang/article/details/18006645)。

## 存在的问题与改进方案

以上大致就是“Android动态性加载初级阶段”的解决方案，虽然现在已经能投入到具体的生产中去，但是还有一些问题无法忽略。

1. 无法使用res目录下的资源，特别是使用XML布局，以及无法通过res资源到达自适应
2. 无法动态加载新的Activity等组件，因为这些组件需要在Manifest中注册，动态加载无法更改当前APK的Manifest

以上问题可以通过**反射调用Framework层代码**以及**代理Activity**的方式解决，可以把这种的动态加载框架成为“代理模式”。

## 参考日志

<http://44289533.iteye.com/blog/1954453>  
<http://blog.csdn.net/bboyfeiyu/article/details/11710497>  
<http://www.cnblogs.com/over140/archive/2011/11/23/2259367.html>

* [2015年11月29日发布](https://segmentfault.com/a/1190000004062952)