[Android动态加载基础 ClassLoader工作机制](https://segmentfault.com/a/1190000004062880)

https://segmentfault.com/a/1190000004062880

Last Edit: 2016-02-10

## 基本信息

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

## 类加载器ClassLoader

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。

对于Java程序来说，编写程序就是编写类，运行程序也就是运行类（编译得到的class文件），其中起到关键作用的就是类加载器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（这点后面在解释说明）。

## 有几个ClassLoader实例？

**动态加载的基础是ClassLoader**，从名字也可以看出，ClassLoader就是专门用来处理类加载工作的，所以这货也叫类加载器，而且一个运行中的APP **不仅只有一个类加载器**。

其实，在Android系统启动的时候会创建一个Boot类型的ClassLoader实例，用于加载一些系统Framework层级需要的类，我们的Android应用里也需要用到一些系统的类，所以APP启动的时候也会把这个Boot类型的ClassLoader传进来。

此外，APP也有自己的类，这些类保存在APK的dex文件里面，所以APP启动的时候，也会创建一个自己的ClassLoader实例，用于加载自己dex文件中的类。下面我们在项目里验证看看

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

ClassLoader classLoader = getClassLoader();

if (classLoader != null){

Log.i(TAG, "[onCreate] classLoader " + i + " : " + classLoader.toString());

while (classLoader.getParent()!=null){

classLoader = classLoader.getParent();

Log.i(TAG,"[onCreate] classLoader " + i + " : " + classLoader.toString());

}

}

}

输出结果为

[onCreate] classLoader 1 : dalvik.system.PathClassLoader[DexPathList[[zip file "/data/app/me.kaede.anroidclassloadersample-1/base.apk"],nativeLibraryDirectories=[/vendor/lib, /system/lib]]]

[onCreate] classLoader 2 : java.lang.BootClassLoader@14af4e32

可以看见有2个Classloader实例，一个是BootClassLoader（系统启动的时候创建的），另一个是PathClassLoader（应用启动时创建的，用于加载“/data/app/me.kaede.anroidclassloadersample-1/base.apk”里面的类）。由此也可以看出，**一个运行的Android应用至少有2个ClassLoader**。

## 创建自己ClassLoader实例

动态加载外部的dex文件的时候，我们也可以使用自己创建的ClassLoader实例来加载dex里面的Class，不过ClassLoader的创建方式有点特殊，我们先看看它的构造方法

/\*

\* constructor for the BootClassLoader which needs parent to be null.

\*/

ClassLoader(ClassLoader parentLoader, boolean nullAllowed) {

if (parentLoader == null && !nullAllowed) {

throw new NullPointerException("parentLoader == null && !nullAllowed");

}

parent = parentLoader;

}

创建一个ClassLoader实例的时候，需要使用一个现有的ClassLoader实例作为新创建的实例的Parent。这样一来，一个Android应用，甚至整个Android系统里所有的ClassLoader实例都会被一棵树关联起来，这也是ClassLoader的 **双亲代理模型**（Parent-Delegation Model）的特点。

## ClassLoader双亲代理模型加载类的特点和作用

JVM中ClassLoader通过defineClass方法加载jar里面的Class，而Android中这个方法被弃用了。

@Deprecated

protected final Class<?> defineClass(byte[] classRep, int offset, int length)

throws ClassFormatError {

throw new UnsupportedOperationException("can't load this type of class file");

}

取而代之的是loadClass方法

public Class<?> loadClass(String className) throws ClassNotFoundException {

return loadClass(className, false);

}

protected Class<?> loadClass(String className, boolean resolve) throws ClassNotFoundException {

Class<?> clazz = findLoadedClass(className);

if (clazz == null) {

ClassNotFoundException suppressed = null;

try {

clazz = parent.loadClass(className, false);

} catch (ClassNotFoundException e) {

suppressed = e;

}

if (clazz == null) {

try {

clazz = findClass(className);

} catch (ClassNotFoundException e) {

e.addSuppressed(suppressed);

throw e;

}

}

}

return clazz;

}

### 特点

从源码中我们也可以看出，loadClass方法在加载一个类的实例的时候，

1. 会先查询当前ClassLoader实例是否加载过此类，有就返回；
2. 如果没有。查询Parent是否已经加载过此类，如果已经加载过，就直接返回Parent加载的类；
3. 如果继承路线上的ClassLoader都没有加载，才由Child执行类的加载工作；

这样做有个明显的特点，如果一个类被位于树根的ClassLoader加载过，那么在以后整个系统的生命周期内，这个类永远不会被重新加载。

### 作用

首先是共享功能，一些Framework层级的类一旦被顶层的ClassLoader加载过就缓存在内存里面，以后任何地方用到都不需要重新加载。

除此之外还有隔离功能，不同继承路线上的ClassLoader加载的类肯定不是同一个类，这样的限制避免了用户自己的代码冒充核心类库的类访问核心类库包可见成员的情况。这也好理解，一些系统层级的类会在系统初始化的时候被加载，比如java.lang.String，如果在一个应用里面能够简单地用自定义的String类把这个系统的String类给替换掉，那将会有严重的安全问题。

## 使用ClassLoader一些需要注意的问题

我们都知道，我们可以通过动态加载获得新的类，从而升级一些代码逻辑，这里有几个问题要注意一下。

如果你希望通过动态加载的方式，加载一个新版本的dex文件，使用里面的新类替换原有的旧类，从而修复原有类的BUG，那么你必须保证在加载新类的时候，旧类还没有被加载，因为如果已经加载过旧类，那么ClassLoader会一直优先使用旧类。

如果旧类总是优先于新类被加载，我们也可以使用一个与加载旧类的ClassLoader没有树的继承关系的另一个ClassLoader来加载新类，因为ClassLoader只会检查其Parent有没有加载过当前要加载的类，如果两个ClassLoader没有继承关系，那么旧类和新类都能被加载。

不过这样一来又有另一个问题了，在Java中，只有当两个实例的类名、包名以及加载其的ClassLoader都相同，才会被认为是同一种类型。上面分别加载的新类和旧类，虽然包名和类名都完全一样，但是由于加载的ClassLoader不同，所以并不是同一种类型，在实际使用中可能会出现类型不符异常。

同一个Class = 相同的 ClassName + PackageName + ClassLoader

以上问题在采用动态加载功能的开发中容易出现，请注意。

## DexClassLoader 和 PathClassLoader

在Android中，ClassLoader是一个抽象类，实际开发过程中，我们一般是使用其具体的子类DexClassLoader、PathClassLoader这些类加载器来加载类的，它们的不同之处是：

* DexClassLoader可以加载jar/apk/dex，可以从SD卡中加载未安装的apk；
* PathClassLoader只能加载系统中已经安装过的apk；

## 类加载器的初始化

平时开发的时候，使用DexClassLoader就够用了，但是我们不妨挖一下这两者具体细节上的区别。

// DexClassLoader.java

public class DexClassLoader extends BaseDexClassLoader {

public DexClassLoader(String dexPath, String optimizedDirectory,

String libraryPath, ClassLoader parent) {

super(dexPath, new File(optimizedDirectory), libraryPath, parent);

}

}

// PathClassLoader.java

public class PathClassLoader extends BaseDexClassLoader {

public PathClassLoader(String dexPath, ClassLoader parent) {

super(dexPath, null, null, parent);

}

public PathClassLoader(String dexPath, String libraryPath,

ClassLoader parent) {

super(dexPath, null, libraryPath, parent);

}

}

这两者只是简单的对BaseDexClassLoader做了一下封装，具体的实现还是在父类里。不过这里也可以看出，PathClassLoader的optimizedDirectory只能是null，进去BaseDexClassLoader看看这个参数是干什么的

public BaseDexClassLoader(String dexPath, File optimizedDirectory,

String libraryPath, ClassLoader parent) {

super(parent);

this.originalPath = dexPath;

this.pathList = new DexPathList(this, dexPath, libraryPath, optimizedDirectory);

}

这里创建了一个DexPathList实例，进去看看

public DexPathList(ClassLoader definingContext, String dexPath,

String libraryPath, File optimizedDirectory) {

……

this.dexElements = makeDexElements(splitDexPath(dexPath), optimizedDirectory);

}

private static Element[] makeDexElements(ArrayList<File> files,

File optimizedDirectory) {

ArrayList<Element> elements = new ArrayList<Element>();

for (File file : files) {

ZipFile zip = null;

DexFile dex = null;

String name = file.getName();

if (name.endsWith(DEX\_SUFFIX)) {

dex = loadDexFile(file, optimizedDirectory);

} else if (name.endsWith(APK\_SUFFIX) || name.endsWith(JAR\_SUFFIX)

|| name.endsWith(ZIP\_SUFFIX)) {

zip = new ZipFile(file);

}

……

if ((zip != null) || (dex != null)) {

elements.add(new Element(file, zip, dex));

}

}

return elements.toArray(new Element[elements.size()]);

}

private static DexFile loadDexFile(File file, File optimizedDirectory)

throws IOException {

if (optimizedDirectory == null) {

return new DexFile(file);

} else {

String optimizedPath = optimizedPathFor(file, optimizedDirectory);

return DexFile.loadDex(file.getPath(), optimizedPath, 0);

}

}

/\*\*

\* Converts a dex/jar file path and an output directory to an

\* output file path for an associated optimized dex file.

\*/

private static String optimizedPathFor(File path,

File optimizedDirectory) {

String fileName = path.getName();

if (!fileName.endsWith(DEX\_SUFFIX)) {

int lastDot = fileName.lastIndexOf(".");

if (lastDot < 0) {

fileName += DEX\_SUFFIX;

} else {

StringBuilder sb = new StringBuilder(lastDot + 4);

sb.append(fileName, 0, lastDot);

sb.append(DEX\_SUFFIX);

fileName = sb.toString();

}

}

File result = new File(optimizedDirectory, fileName);

return result.getPath();

}

看到这里我们明白了，optimizedDirectory是用来缓存我们需要加载的dex文件的，并创建一个DexFile对象，如果它为null，那么会直接使用dex文件原有的路径来创建DexFile  
对象。

optimizedDirectory必须是一个内部存储路径，还记得我们之前说过的，无论哪种动态加载，加载的可执行文件一定要存放在内部存储。DexClassLoader可以指定自己的optimizedDirectory，所以它可以加载外部的dex，因为这个dex会被复制到内部路径的optimizedDirectory；而PathClassLoader没有optimizedDirectory，所以它只能加载内部的dex，这些大都是存在系统中已经安装过的apk里面的。

## 加载类的过程

上面还只是创建了类加载器的实例，其中创建了一个DexFile实例，用来保存dex文件，我们猜想这个实例就是用来加载类的。

Android中，ClassLoader用loadClass方法来加载我们需要的类

public Class<?> loadClass(String className) throws ClassNotFoundException {

return loadClass(className, false);

}

protected Class<?> loadClass(String className, boolean resolve) throws ClassNotFoundException {

Class<?> clazz = findLoadedClass(className);

if (clazz == null) {

ClassNotFoundException suppressed = null;

try {

clazz = parent.loadClass(className, false);

} catch (ClassNotFoundException e) {

suppressed = e;

}

if (clazz == null) {

try {

clazz = findClass(className);

} catch (ClassNotFoundException e) {

e.addSuppressed(suppressed);

throw e;

}

}

}

return clazz;

}

loadClass方法调用了findClass方法，而BaseDexClassLoader重载了这个方法，得到BaseDexClassLoader看看

@Override

protected Class<?> findClass(String name) throws ClassNotFoundException {

Class clazz = pathList.findClass(name);

if (clazz == null) {

throw new ClassNotFoundException(name);

}

return clazz;

}

结果还是调用了DexPathList的findClass

public Class findClass(String name) {

for (Element element : dexElements) {

DexFile dex = element.dexFile;

if (dex != null) {

Class clazz = dex.loadClassBinaryName(name, definingContext);

if (clazz != null) {

return clazz;

}

}

}

return null;

}

这里遍历了之前所有的DexFile实例，其实也就是遍历了所有加载过的dex文件，再调用loadClassBinaryName方法一个个尝试能不能加载想要的类，真是简单粗暴

public Class loadClassBinaryName(String name, ClassLoader loader) {

return defineClass(name, loader, mCookie);

}

private native static Class defineClass(String name, ClassLoader loader, int cookie);

看到这里想必大家都明白了，loadClassBinaryName中调用了Native方法defineClass加载类。

至此，ClassLoader的创建和加载类的过程的完成了。有趣的是，标准JVM中，ClassLoader是用defineClass加载类的，而Android中defineClass被弃用了，改用了loadClass方法，而且加载类的过程也挪到了DexFile中，在DexFile中加载类的具体方法也叫defineClass，不知道是Google故意写成这样的还是巧合。

## 自定义ClassLoader

平时进行动态加载开发的时候，使用DexClassLoader就够了。但我们也可以创建自己的类去继承ClassLoader，需要注意的是loadClass方法并不是final类型的，所以我们可以重载loadClass方法并改写类的加载逻辑。

通过前面我们分析知道，ClassLoader双亲代理的实现很大一部分就是在loadClass方法里，我们可以通过重写loadClass方法避开双亲代理的框架，这样一来就可以在重新加载已经加载过的类，也可以在加载类的时候注入一些代码。这是一种Hack的开发方式，采用这种开发方式的程序稳定性可能比较差，但是却可以实现一些“黑科技”的功能。

## Android程序比起一般Java程序在使用动态加载时麻烦在哪里

通过上面的分析，我们知道使用ClassLoader动态加载一个外部的类是非常容易的事情，所以很容易就能实现动态加载新的可执行代码的功能，但是比起一般的Java程序，在Android程序中使用动态加载主要有两个麻烦的问题：

1. Android中许多组件类（如Activity、Service等）是需要在Manifest文件里面注册后才能工作的（系统会检查该组件有没有注册），所以即使动态加载了一个新的组件类进来，没有注册的话还是无法工作；
2. Res资源是Android开发中经常用到的，而Android是把这些资源用对应的R.id注册好，运行时通过这些ID从Resource实例中获取对应的资源。如果是运行时动态加载进来的新类，那类里面用到R.id的地方将会抛出找不到资源或者用错资源的异常，因为新类的资源ID根本和现有的Resource实例中保存的资源ID对不上；

说到底，抛开虚拟机的差别不说，**一个Android程序和标准的Java程序最大的区别就在于他们的上下文环境（Context）不同**。Android中，这个环境可以给程序提供组件需要用到的功能，也可以提供一些主题、Res等资源，其实上面说到的两个问题都可以统一说是这个环境的问题，而现在的各种Android动态加载框架中，核心要解决的东西也正是“如何给外部的新类提供上下文环境”的问题。

* [2015年11月29日发布](https://segmentfault.com/a/1190000004062880)