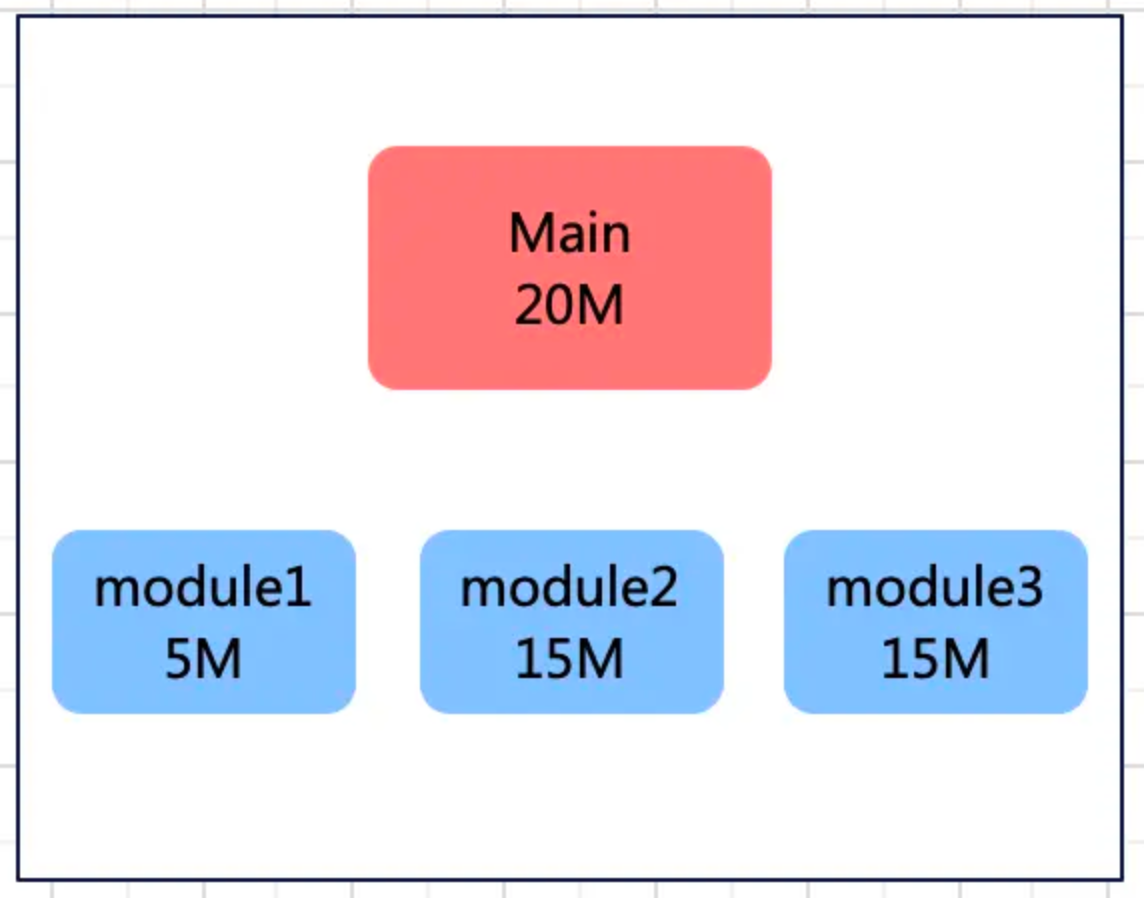
一个应用主模块20M，其他3个模块可以看做是3个App，分别占5M、15M、15M，如果打包，那么整个包体积为55M；如果我们需要做包体积压缩，那么这3个实打实的app无论怎么压缩都会占用app的体积。

那么如果使用插件化技术呢？最终打出的包体积只有20M，其他3个模块都是以插件的方式存在，而Main App则是能够支持插件的宿主App，所以插件化的特点就是不需要安装就能运行app



# 插件化解决的问题

（1）app功能模块越来越多，包体积增大

其实这是一个app成为大型app的必经之路，模块越加越多，所以就如前文讲解的一样，采用插件的方式，当需要启动一个app的时候，将插件下载下来，调用插件中的方法运行app。

（2）模块解耦

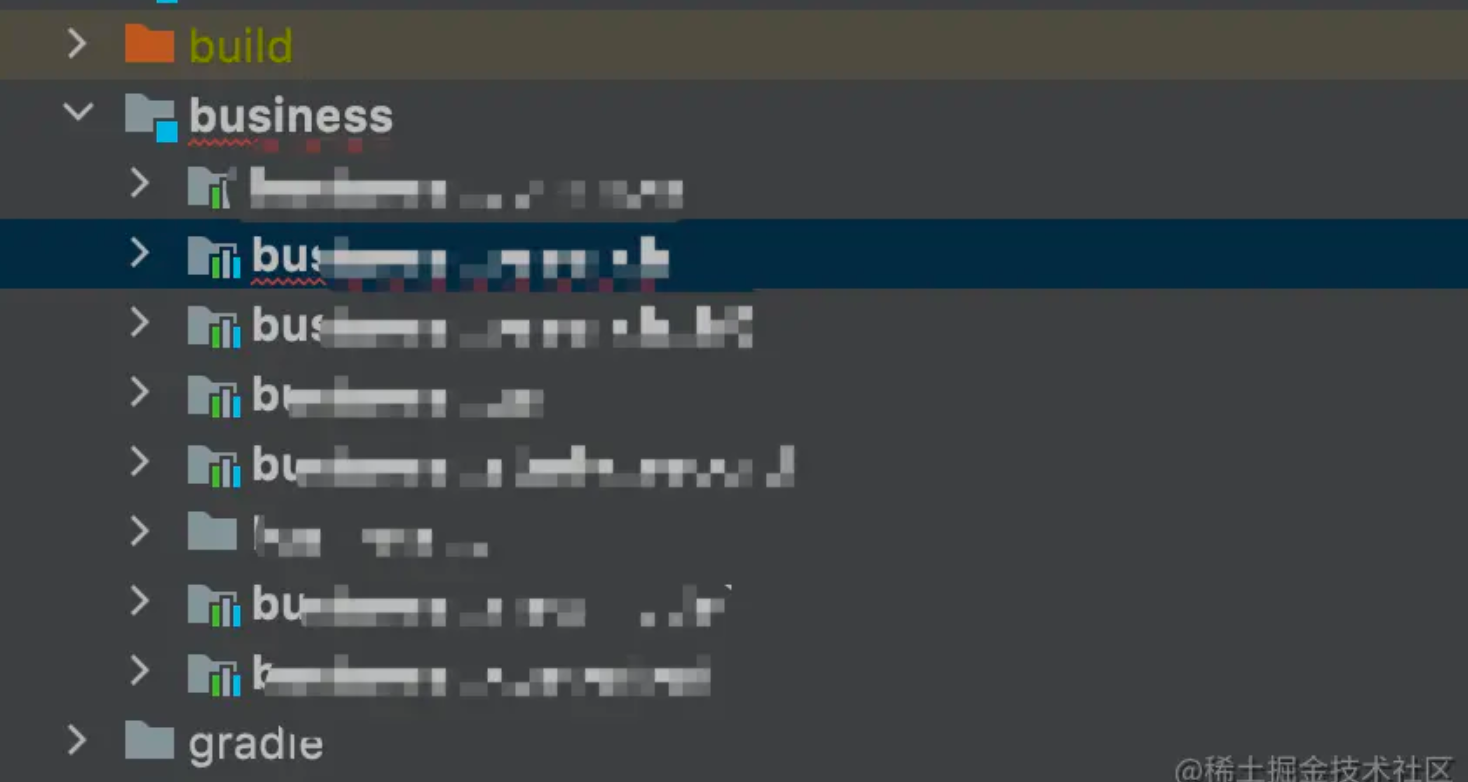
每个插件其实在app中都可以看做是一个单独的模块，如果采用插件化的方式，那么可以将每个功能抽离为单独的module，每个module可以独立运行，不会出现多个模块耦合在一块的问题

（3）多应用之间相互调用

这个其实我们在使用支付宝、淘宝的时候，经常会使用到，例如从闲鱼app中跳转到支付宝、或者跳转到淘宝，支持相互调用。

组件化和插件化的区别

在实际项目中，组件化是使用最频繁的，例如



将app分为多个模块，每个模块都是一个组件，在开发过程中，组件之间可以相互依赖，也可以单独作为app调试，最终打包的时候，是将这些组件合并到一起打包成一个apk。

而插件化和组件化类似的是，app同样被分为多个模块，但是每个模块都有一个宿主和多个插件，也就是说每个模块都是一个apk，最终打包的时候宿主apk和插件apk分开打包

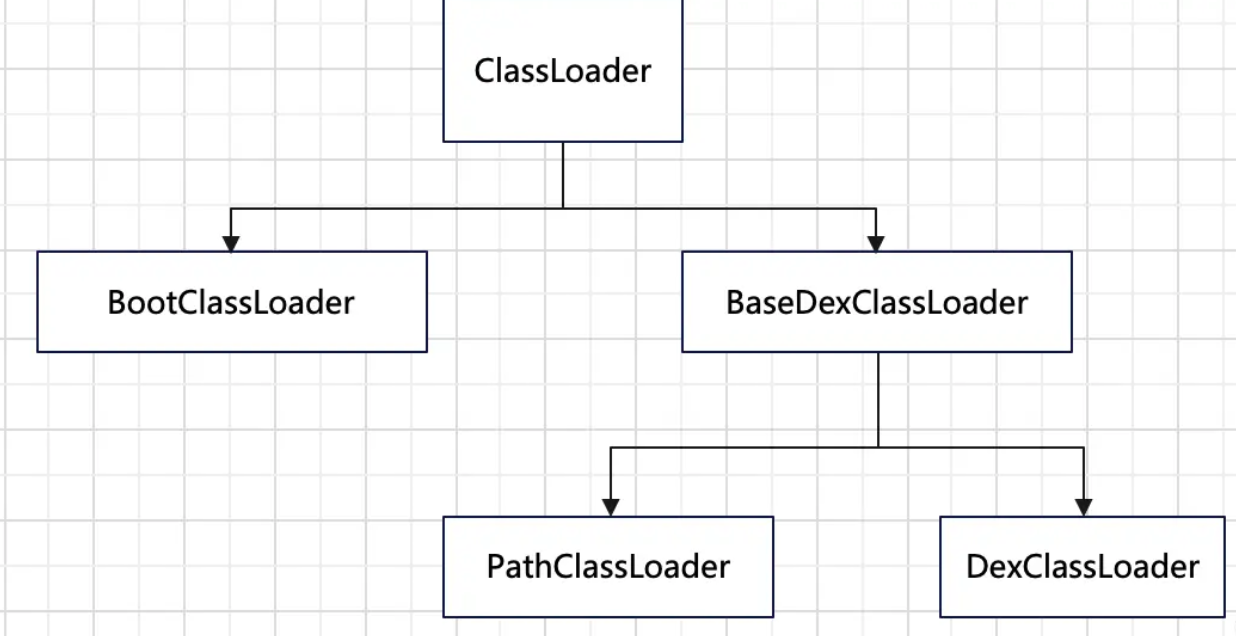
插件化设计思路

在设计一个框架的时候，往往需要想明白目的是什么？插件是一个apk，如果我们想要启动这个插件，主要有以下几个关键点：  
（1）如何启动插件  
（2）如何加载插件中的类  
（3）如何加载插件中的资源  
（4）如何调用插件中的类和方法

## Android的类加载机制

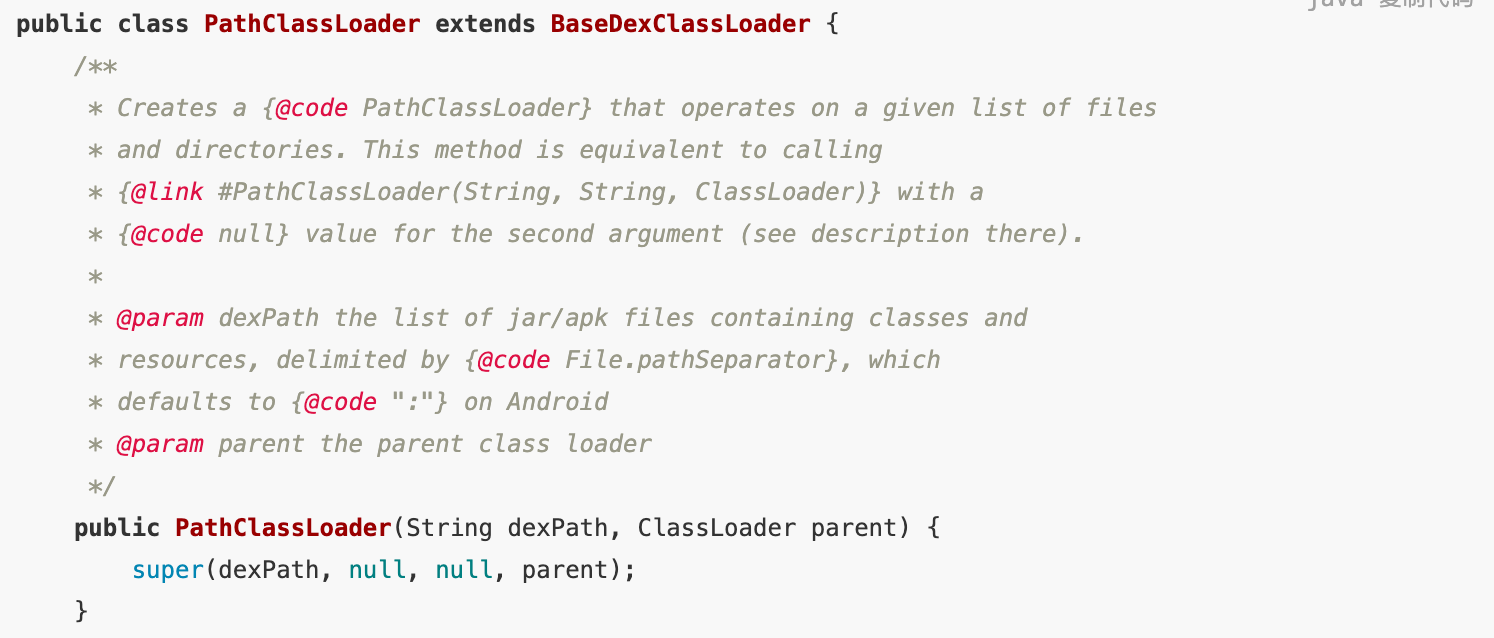
如果想要加载插件中的类，那么对于Android的类加载机制必须要了解，在之前Tinker热修复专题中，其实已经介绍了Android的类加载机制，那么这里再简单介绍一下。

Android类加载和Java不同的在于，Android拥有自己的类加载器，看下图



### PathClassLoader和DexClassLoader有啥区别？

在Android中常用的两个类加载器，分别是PathClassLoader和DexClassLoader，两者的区别我们稍后分析，先看下具体的源码分析。





PathClassLoader是继承自BaseDexClassLoader，其中只有两个构造方法，先不着急，再看下DexClassLoader的源码

public class DexClassLoader extends BaseDexClassLoader {

/\*\*

\* Creates a {@code DexClassLoader} that finds interpreted and native

\* code. Interpreted classes are found in a set of DEX files contained

\* in Jar or APK files.

\* <p>The path lists are separated using the character specified by the

\* {@code path.separator} system property, which defaults to {@code :}.

\* @param dexPath the list of jar/apk files containing classes and

\* resources, delimited by {@code File.pathSeparator}, which

\* defaults to {@code ":"} on Android

\* @param optimizedDirectory this parameter is deprecated and has no effect since API level 26.

\* @param librarySearchPath the list of directories containing native

\* libraries, delimited by {@code File.pathSeparator}; may be

\* {@code null}

\* @param parent the parent class loader

\*/

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

super(dexPath, null, librarySearchPath, parent);

}

//Android 8.0 以前的源码

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

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

}

}

DexClassLoader同样是继承自BaseDexClassLoader，而且跟PathClassLoader中构造方法传入的值是一样的，当前版本是Android10版本，其实在Android 8.0之前的版本，第二个参数是必须要传入的，optimizedDirectory是dex优化之后生成odex文件存储地址，但是Android 8.0之后，就直接传null了

所以在Android 8.0之前，PathClassLoader和DexClassLoader还是有区别的，但是在Android 8.0之后，两者就是一样的了

PathClassLoader和BootClassLoader

BaseDexClassLoader是PathClassLoader继承上的父类，但是并不代表BaseDexClassLoader是PathClassLoader的父类加载器，我们通过代码可以看一下



我们在MainActivity中打印下日志，我们可以发现就是MainActivity的类加载器是PathClassLoader，而PathClassLoader的父类加载器是BootClassLoader

E/TAG: classLoader dalvik.system.PathClassLoader[DexPathList[[zip file "/data/app/com.lay.image\_process-n8633iv\_VMBnRO2AEqJ4rg==/base.apk"],nativeLibraryDirectories=[/data/app/com.lay.image\_process-n8633iv\_VMBnRO2AEqJ4rg==/lib/x86, /system/lib]]] parent java.lang.BootClassLoader@3ab8f00

那么PathClassLoader和BootClassLoader分别加在什么类呢？



通过之前的代码，我们可以看到，应用内的类都是PathClassLoader来加载（包括三方库），而Activity的类加载器是BootClassLoader，也就是说Android SDK中的类是由BootClassLoader来加载的

### 双亲委派机制

和Java的类加载机制一样，Android类加载同样遵循双亲委派机制，我们看下ClassLoader的loadClass方法。

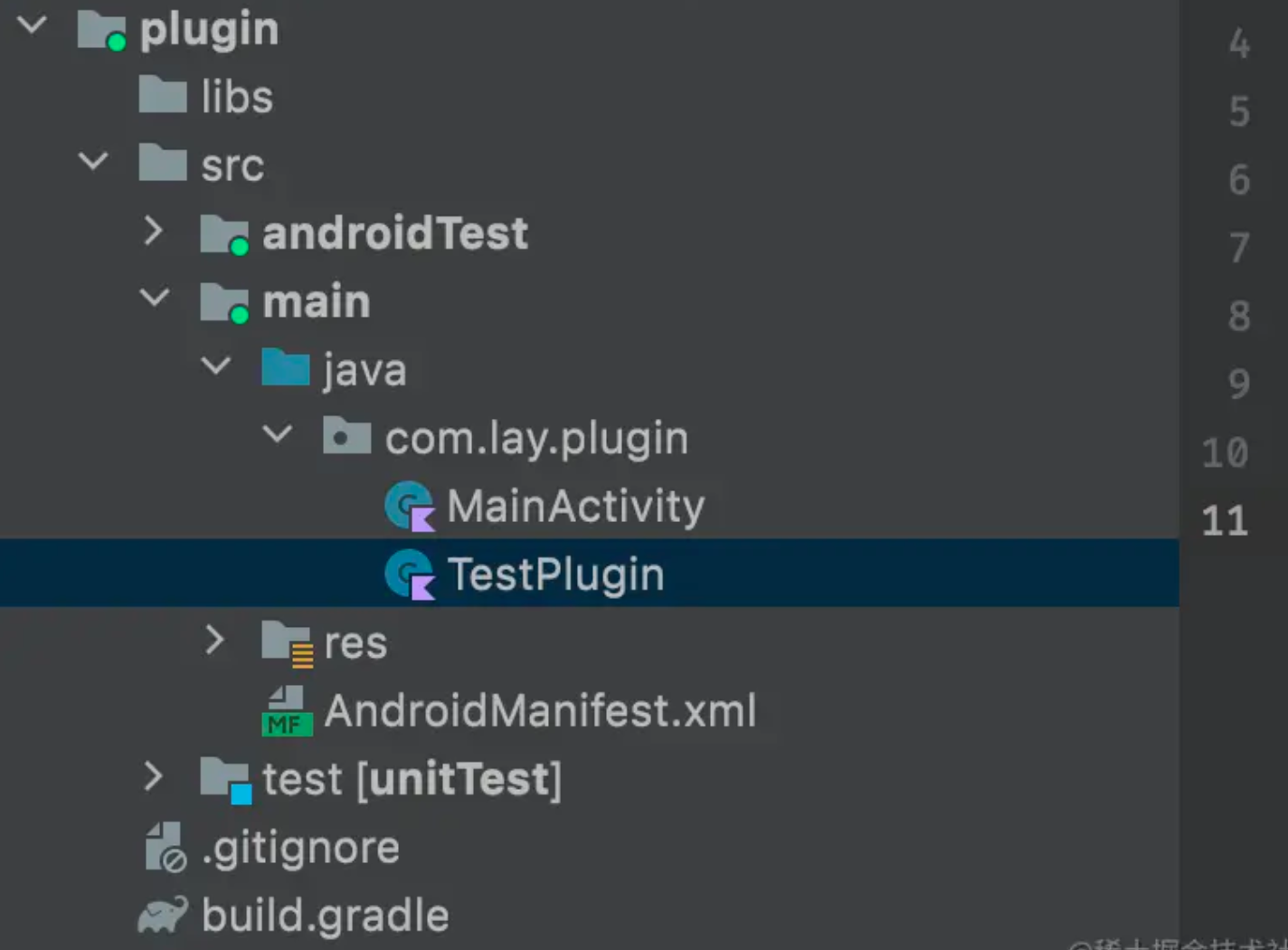


（1）首先通过类的全类名，查找这个类是不是已经被加载过了，如果已经加载过了，那么直接返回；  
（2）如果没有被加载过，那么首先会判断父类加载器是否为空，如果不为空那么就交给父类加载器去加载，依次递归，如果某个父类加载器加载过了，那么就返回，如果所有的父类加载器都遍历过了，而且不能去加载这个类，那么就自己去加载；  
（3）自己怎么加载呢？就是从DexPathList中取出dex文件加载其中类，跟我们今天讲的插件化就联系起来了，其实就是通过PathClassLoader或者DexClassLoader去加载

那么为什么要使用双亲委派机制呢？其实更多的是为了安全性考虑，假如我们自己写了一个String类，想要代替系统的String，这个其实是不可能的，因为系统SDK中的类已经被BootClassLoader加载过了，我们应用内的String类就不会再次被加载。

加载插件中的类

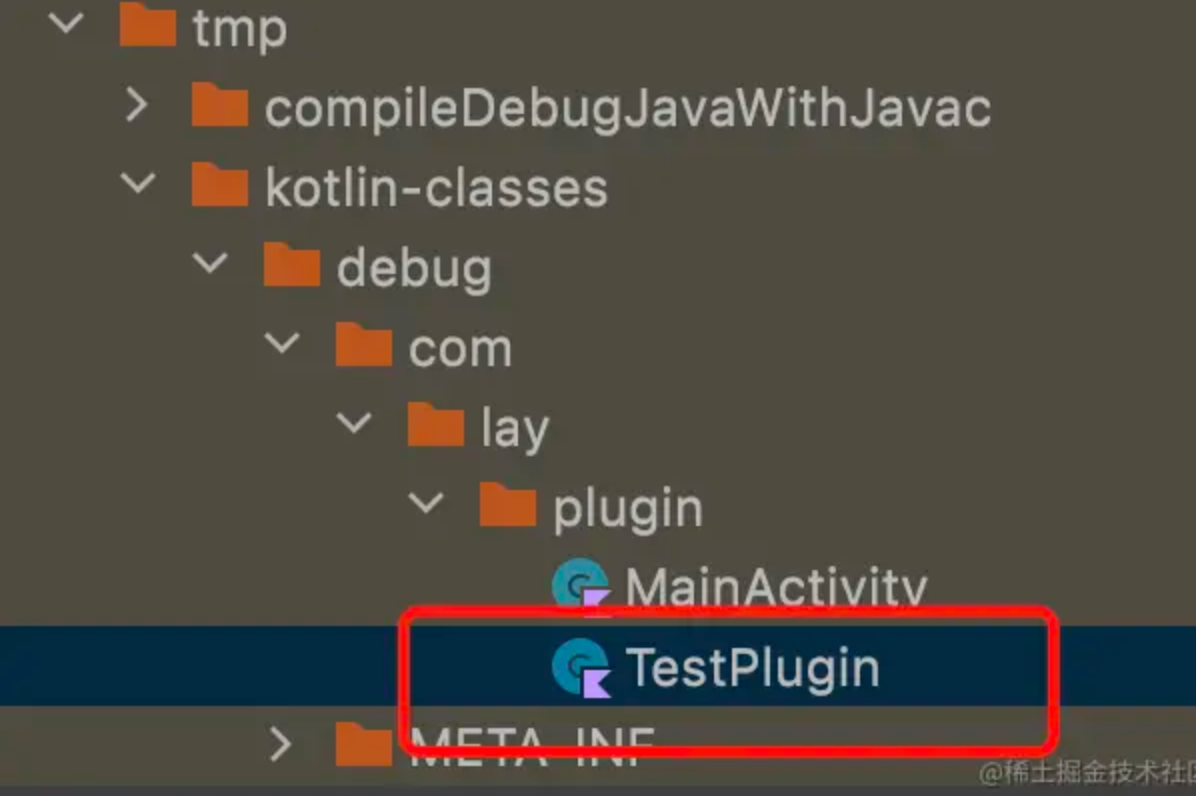
通过前面对于类加载机制的简单了解，我们知道，插件中类其实就可以通过ClassLoader来加载，所以我们先尝试加载插件中某个类，调用它的方法。



插件也是一个apk，其中有一个TestPlugin类



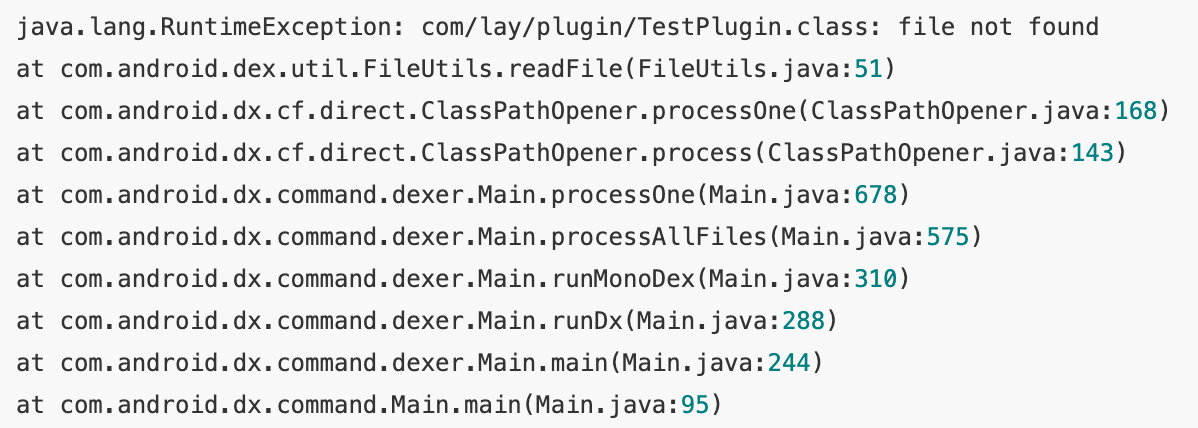
TestPlugin通过编译成class文件后，转为dex文件打包进入apk，我们可以模拟这个场景



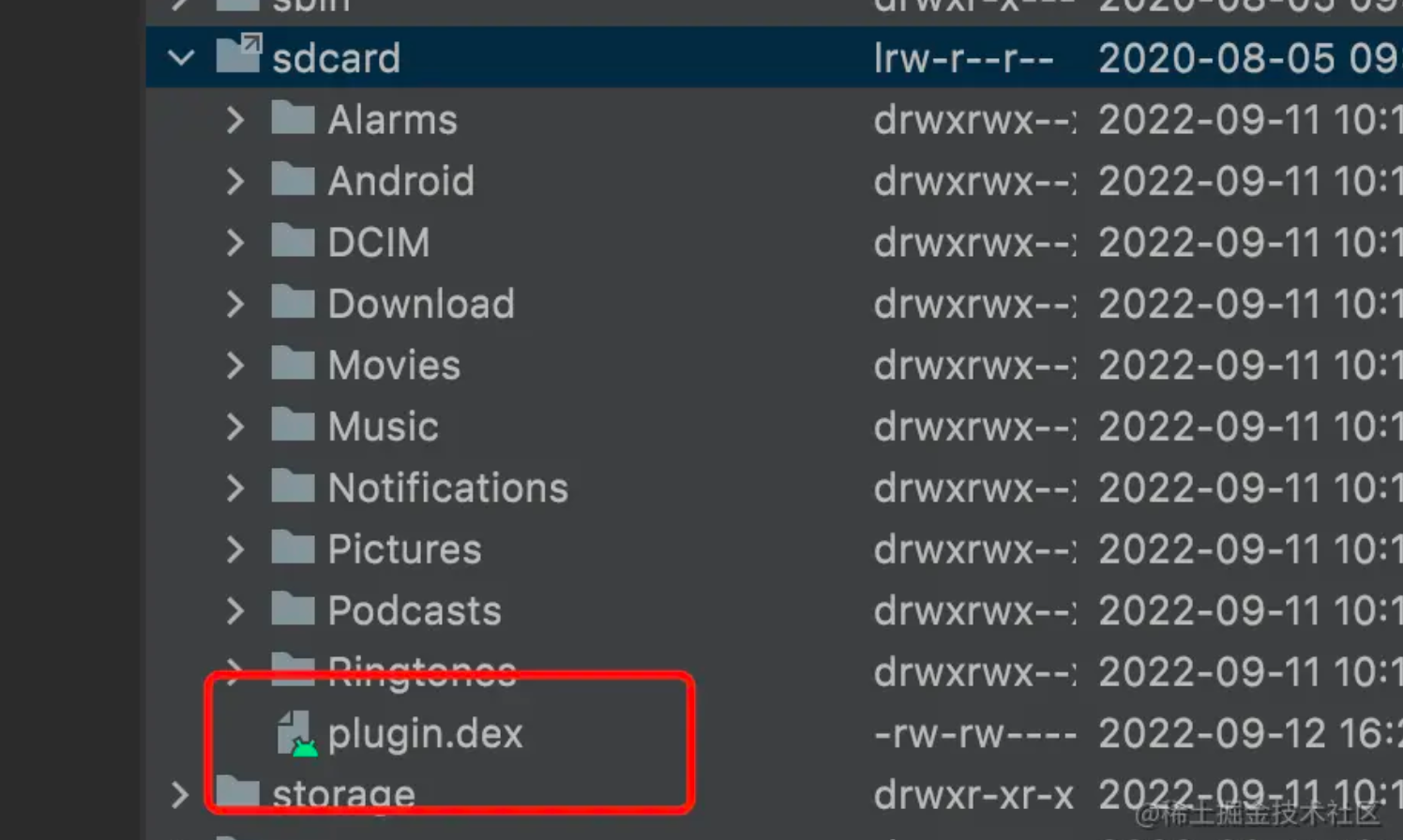
将class转换为dex文件，采用下面的命令行

dx --dex --output=/Users/xxx/Desktop/dx/plugin.dex com/lay/plugin/TestPlugin.class

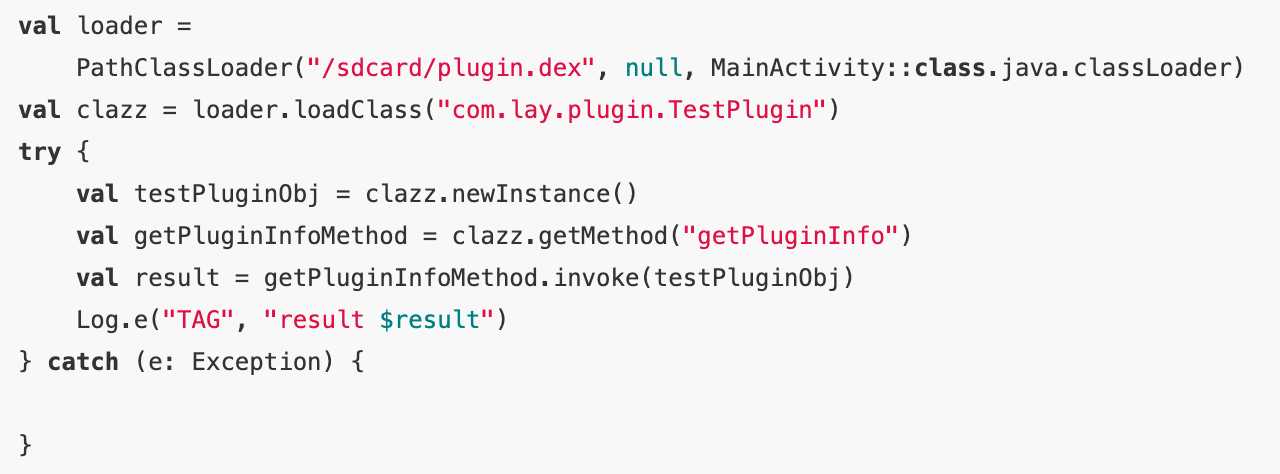
这里需要注意一点就是，/Users/xxx/Desktop/dx是class文件所在包名的前缀，/Users/xxx/Desktop/dx/com/lay/plugin/TestPlugin.class是class文件所在的全路径，只有这样才能生成dex文件，不然可能会报错



这样，我们得到dex文件之后，可以将其放在sd卡下面，通过ClassLoader去加载某个类。



通过创建PathClassLoader或者DexClassLoader，去加载插件（dex）中类，获取Class对象，通过反射可以生成一个类对象，获取到getPluginInfo方法后，调用这个方法





其实从这里就能验证，无论是PathClassLoader还是DexClassLoader，都可以加载未安装apk中的类。

## 宿主和插件dex合并

在上一小节中，我们采用了反射的方式，加载dex文件中的类，但是实际的项目开发中，伙伴们认为这种方式可取吗？显然不可取，一个插件可能有上千个方法，都采用反射的方式去调用，那岂不是太荒唐了，所以我们想，既然宿主apk能够加载apk中所有的类和资源，那么能不能把插件中的类和资源也全部捎带上呢？

首先我们先看一下宿主apk加载的流程，之前上一小节中，我们看到了类加载的双亲委派机制，其实应用中的类都是由PathClassLoader加载的，所以我们看下PathClassLoader是如何加载类的。

因为PathClassLoader只有两个构造方法，所以直接去它父类BaseDexClassLoader中查看源码；在ClassLoader的loadClass方法中，我们看到如果没有其他父类加载器能够加载这个类，就会由当前类加载器调用findClass方法区加载，所以我们看下BaseDexClassLoader中的findClass方法。

### DexPathList和dexElements



BaseDexClassLoader中的findClass中，调用了pathList的findClass方法，如果没有找到，那么就会抛出ClassNotFoundException的异常，那么pathList是什么呢？

pathList是BaseDexClassLoader中的一个变量DexPathList，是在BaseDexClassLoader的构造方法中完成初始化，会将dexPath作为参数传递进来，其实在上一小节中，我们在创建PathClassLoader的时候，其实已经初始化了这个DexPathList

/\*\*

\* @hide

\*/

public BaseDexClassLoader(String dexPath, File optimizedDirectory,String librarySearchPath, ClassLoader parent, boolean isTrusted) {

super(parent);

this.pathList = new DexPathList(this, dexPath, librarySearchPath, null, isTrusted);

if (reporter != null) {

reportClassLoaderChain();

}

}

既然后缀带有一个List，我们猜到这个数据结构应该是个数组，那么我们看下DexPathList到底是个什么

DexPathList(ClassLoader definingContext, String dexPath,String librarySearchPath, File optimizedDirectory, boolean isTrusted) {

if (definingContext == null) {

throw new NullPointerException("definingContext == null");

}

if (dexPath == null) {

throw new NullPointerException("dexPath == null");

}

if (optimizedDirectory != null) {

if (!optimizedDirectory.exists()) {

throw new IllegalArgumentException(

"optimizedDirectory doesn't exist: "

+ optimizedDirectory);

}

if (!(optimizedDirectory.canRead()

&& optimizedDirectory.canWrite())) {

throw new IllegalArgumentException(

"optimizedDirectory not readable/writable: "

+ optimizedDirectory);

}

}

this.definingContext = definingContext;

ArrayList<IOException> suppressedExceptions = new ArrayList<IOException>();

// save dexPath for BaseDexClassLoader

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

suppressedExceptions, definingContext, isTrusted);

}

在DexPathList中，有一个非常重要的成员变量dexElements，我们看过apk包的话，应该会看到有很多dex文件。所以我们传入的dexPath下，可能存在多个dex文件，那么dexElements其实就是存储这些dex文件的，我们可以看到，在DexPathList的构造方法中，调用了makeDexElements方法，其实就是将dex文件存储在dexElements数组中。



首先在makeDexElements方法中，首先调用了splitPaths方法，这个方法就是将传入的dexPath路径下全部的dex文件存储在一个List集合中，作为第一个参数，传入到makeDexElements方法中

private static Element[] makeDexElements(List<File> files, File optimizedDirectory,List<IOException> suppressedExceptions, ClassLoader loader, boolean isTrusted) {

Element[] elements = new Element[files.size()];

int elementsPos = 0;

/\*

\* Open all files and load the (direct or contained) dex files up front.

\*/

for (File file : files) {

if (file.isDirectory()) {

// We support directories for looking up resources. Looking up resources in

// directories is useful for running libcore tests.

elements[elementsPos++] = new Element(file);

} else if (file.isFile()) {

String name = file.getName();

DexFile dex = null;

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

// Raw dex file (not inside a zip/jar).

try {

dex = loadDexFile(file, optimizedDirectory, loader, elements);

if (dex != null) {

elements[elementsPos++] = new Element(dex, null);

}

} catch (IOException suppressed) {

System.logE("Unable to load dex file: " + file, suppressed);

suppressedExceptions.add(suppressed);

}

} else {

try {

dex = loadDexFile(file, optimizedDirectory, loader, elements);

} catch (IOException suppressed) {

/\*

\* IOException might get thrown "legitimately" by the DexFile constructor if

\* the zip file turns out to be resource-only (that is, no classes.dex file

\* in it).

\* Let dex == null and hang on to the exception to add to the tea-leaves for

\* when findClass returns null.

\*/

suppressedExceptions.add(suppressed);

}

if (dex == null) {

elements[elementsPos++] = new Element(file);

} else {

elements[elementsPos++] = new Element(dex, file);

}

}

if (dex != null && isTrusted) {

dex.setTrusted();

}

} else {

System.logW("ClassLoader referenced unknown path: " + file);

}

}

if (elementsPos != elements.length) {

elements = Arrays.copyOf(elements, elementsPos);

}

return elements;

}

然后，makeDexElements方法中，创建了一个Element数组，将之前传入的List集合中文件分组，将带有.dex后缀的文件和其他文件（夹）区分放置

看了这么多，核心在于宿主类加载器如何加载apk中的类呢？看下findClass方法



我们可以看到，findClass是遍历dexElements数组，调用Element的findClass方法，如果找到了这个类，那么就直接return，所以如果我们想在宿主app中，加载插件中的类，是不是就可以将插件中的dexElements合并到宿主的dexElements，就可以直接调用了。

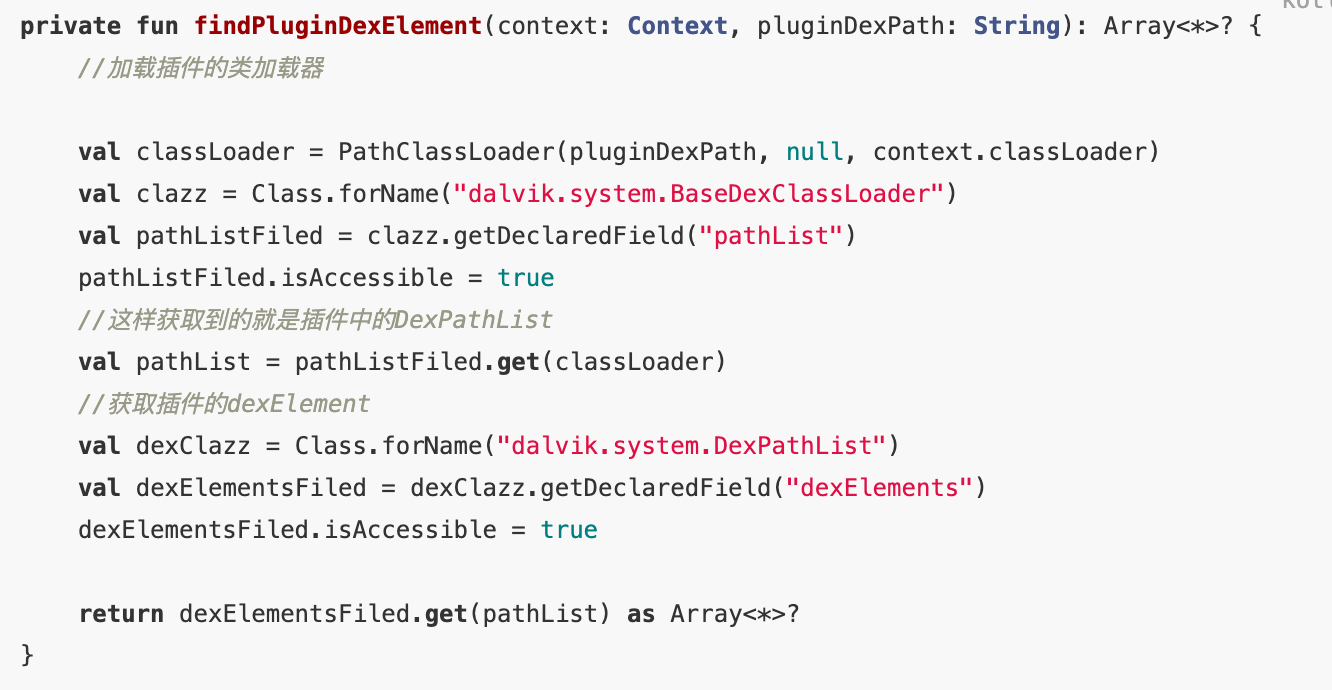
### 实现dex合并工具

通过上面的源码，我们可以得到dex合并的思路

1. 获取宿主的dexElements  
   （2）获取插件的dexElements  
   （3）将宿主的dexElements和插件的dexElements合并成新的dexElements  
   （4）将合并之后的dexElements赋值给宿主的dexElements



通过反射获取BaseDexClassLoader中的pathList成员变量，然后通过DexPathList来获取对应宿主的dexElements



对于插件类，宿主启动的时候并没有加载进来，所以不能使用宿主的类加载器，需要新建一个PathClassLoader来加载对应路径下的apk，这样就能生成对应的dexElements，才可以通过反射去获取。

接下来就是需要合并两个dexElement，因为通过反射是没法获取返回值类型，所以返回的类型是Object类型，那么我们可以创建一个Object数组，然后重新赋值给宿主的dexElements吗？显然不行，我们通过源码可以看到，宿主的dexElements需要的是Element类型的数组，所以需要通过反射来创建数组



创建了新的newDexElements数组之后，通过系统的arraycopy方法，将两个数组拷贝到新的数组中。



最终，将组合之后的Element数组重新赋值给宿主app的dexElements。



其实apk插件的存储一般是存储在服务端，然后从服务端拉取下来，下载然后注入到宿主app中，这里只是模拟放在了sdcard下面，但是这里可能存在一个问题，就是第一次启动速度比较慢，但是也只是第一次，后续下载完成之后，就直接取本地缓存即可。

其实在DexPathList中，提供了一个方法addDexPath，可以将dex文件存储的路径传进去，然后内部自动将dex文件跟与宿主dexElements组合在一起

