类加载机制系列2——深入理解Android中的类加载器

sososeen09 2017-12-20 3,306 阅读12分钟

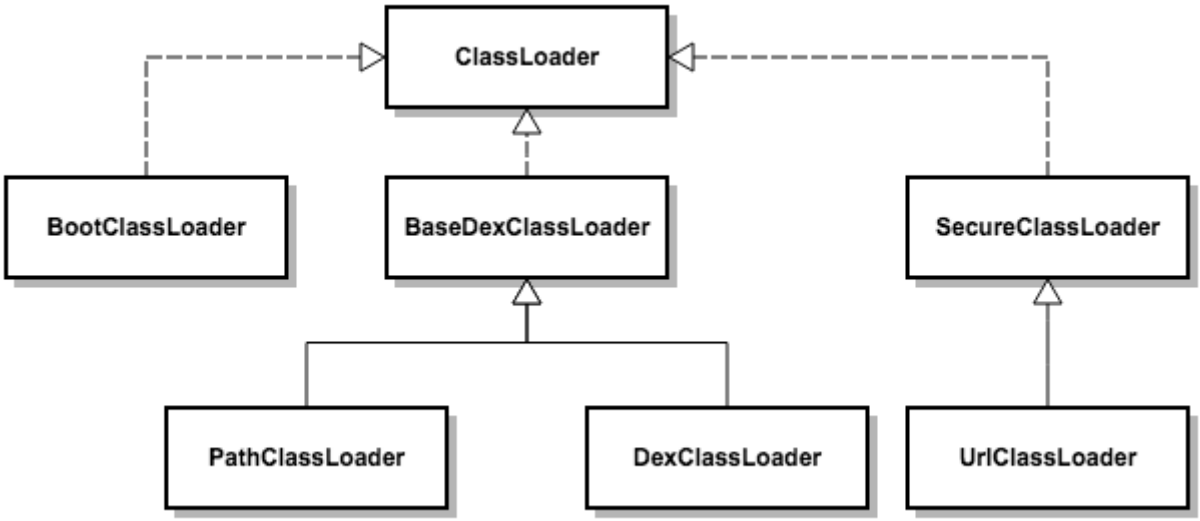
原文链接： www.jianshu.com

1 Android中的ClassLoader

Java中的ClassLoader是加载class文件，而Android中的虚拟机无论是dvm还是art都只能识别dex文件。因此Java中的ClassLoader在Android中不适用。Android中的 `java.lang.ClassLoader` 这个类也不同于Java中的 `java.lang.ClassLoader` 。
Android中的ClassLoader类型也可分为系统ClassLoader和自定义ClassLoader。其中系统ClassLoader包括3种分别是：

- BootClassLoader，Android系统启动时会使用BootClassLoader来预加载常用类，与Java中的Bootstrap ClassLoader不同的是，它并不是由C/C++代码实现，而是由Java实现的。BootClassLoader是ClassLoader的一个内部类。
- PathClassLoader，全名是 `dalvik/system.PathClassLoader` ，可以加载已经安装的Apk，也就是 `/data/app/package` 下的apk文件，也可以加载 `/vendor/lib, /system/lib` 下的nativeLibrary。
- DexClassLoader，全名是 `dalvik/system.DexClassLoader` ，可以加载一个未安装的apk文件。

`PathClassLoader` 和 `DexClasLoader` 都是继承自 `dalviksystem.BaseDexClassLoader` ，它们的类加载逻辑全部写在 `BaseDexClassLoader` 中。
下图展示了Android中的ClassLoader中的继承体系，其中SecureClassLoader和UrlClassLoader是在Java中的类加载器，在Android中是没法办使用的。



Android中的ClassLoader.png

在MainActivity中打印当前的ClassLoader，

```scala
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ClassLoader classLoader = getClassLoader();
        while (classLoader != null) {
            System.out.println("classLoader: " + classLoader);
            classLoader = classLoader.getParent();
        }
    }
}
```

结果如下：

```gradle
dalvik.system.PathClassLoader[DexPathList[[zip file "/data/app/com.sososeen09.classloadtest-1/base.apk"],
java.lang.BootClassLoader@aced87d
```

从打印的结果也可以证实：App系统类加载器是PathClassLoader，而BootClassLoader是其parent类加载器。

2 ClassLoader源码分析

在Android中我们主要关心的是PathClassLoader和DexClassLoader。
PathClassLoader用来操作本地文件系统中的文件和目录的集合。并不会加载来源于网络中的类。Android采用这个类加载器一般是用于加载系统类和它自己的应用类。这个应用类放置在data/data/包名下。
看一下PathClassLoader的源码，只有2个构造方法：

scala 复制代码

```scala
package dalvik.system;

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);
    }
}
```

DexClassLoader可以加载一个未安装的APK，也可以加载其它包含dex文件的JAR/ZIP类型的文件。DexClassLoader需要一个对应用私有且可读写的文件夹来缓存优化后的class文件。而且一定要注意不要把优化后的文件存放到外部存储上，避免使自己的应用遭受代码注入攻击。看一下它的源码，只有1个构造方法:

scala 复制代码

```scala
package dalvik.system;
import java.io.File;
public class DexClassLoader extends BaseDexClassLoader {
    public DexClassLoader(String dexPath, String optimizedDirectory,
            String libraryPath, ClassLoader parent) {
        super(dexPath, new File(optimizedDirectory), libraryPath, parent);
    }
}
```

可以看到，PathClassLoader和DexClassLoader除了构造方法传参不同，其它的逻辑都是一样的。要注意的是DexClassLoader构造方法第2个参数指的是dex优化缓存路径，这个值是不能为空的。而

PathClassLoader对应的dex优化缓存路径为null是因为Android系统自己决定了缓存路径。

先提前漏一嘴，Android中具体负责类加载的并不是哪个ClassLoader，而是通过DexFile的defineClassNative()方法来加载的。

接下来我们看一下BaseDexClassLoader这个类：

BaseDexClassLoader的构造方法有四个参数：

- dexPath，指的是在Androdi包含类和资源的jar/apk类型的文件集合，指的是包含dex文件。多个文件用“：”分隔开，用代码就是`File.pathSeparator`。
- optimizedDirectory，指的是odex优化文件存放的路径，可以为null，那么就采用默认的系统路径。
- libraryPath，指的是native库文件存放目录，也是以“：”分隔。
- parent，parent类加载器

可以看到，在BaseDexClassLoader类中初始化了DexPathList这个类的对象。这个类的作用是存放指明包含dex文件、native库和优化目录。

```arduino
# dalvik.system.BaseDexClassLoader
    public BaseDexClassLoader(String dexPath, File optimizedDirectory,
            String libraryPath, ClassLoader parent) {
        super(parent);
        this.pathList = new DexPathList(this, dexPath, libraryPath, optimizedDirectory);
    }
```

`dalvik.system.DexPathList`封装了dex路径，是一个final类，而且访问权限是包权限，也就是说外界不可继承，也不可访问这个类。
BaseDexClassLoader在其构造方法中初始化了DexPathList对象，我们来看一下DexPathList的源码，我们需要重点关注一下它的成员变量dexElements，它是一个Element[]数组，是包含dex的文件集合。Element是DexPathList的一个静态内部类。DexPathList的构造方法有4个参数。从其构造方法中也可以看到传递过来的classLoade对象和dexPath不能为null，否则就抛出空指针异常。

```haxe
# dalvik.system.DexPathList

private final Element[] dexElements;

public DexPathList(ClassLoader definingContext, String dexPath,
```

```
        String libraryPath, File optimizedDirectory) {
    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);
        }
        // 如果文件不是可读可写的也会抛出异常
        if (!(optimizedDirectory.canRead()
                        && optimizedDirectory.canWrite())) {
            throw new IllegalArgumentException(
                    "optimizedDirectory not readable/writable: "
                    + optimizedDirectory);
        }
    }

    this.definingContext = definingContext;
    ArrayList<IOException> suppressedExceptions = new ArrayList<IOException>();
    // 通过makeDexElements方法来获取Element数组
    // splitDexPath(dexPath)方法是用来把我们之前按照“:”分隔的路径转为File集合。
    this.dexElements = makeDexElements(splitDexPath(dexPath), optimizedDirectory,
                                       suppressedExceptions);
    if (suppressedExceptions.size() > 0) {
        this.dexElementsSuppressedExceptions =
            suppressedExceptions.toArray(new IOException[suppressedExceptions.size()]);
    } else {
        dexElementsSuppressedExceptions = null;
    }
    this.nativeLibraryDirectories = splitLibraryPath(libraryPath);
}
```

makeDexElements方法的作用是获取一个包含dex文件的元素集合。

gradle 复制代码

```
# dalvik.system.DexPathList
private static Element[] makeDexElements(ArrayList<File> files, File optimizedDirectory,
                                         ArrayList<IOException> suppressedExceptions) {
```

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

    // 遍历打开所有的文件并且加载直接或者间接包含dex的文件。
    for (File file : files) {
        File zip = null;
        DexFile dex = null;
        String name = file.getName();

        if (file.isDirectory()) {
            // We support directories for looking up resources.
            // This is only useful for running libcore tests.
            // 可以发现它是支持传递目录的，但是说只测试libCore的时候有用
            elements.add(new Element(file, true, null, null));
        } else if (file.isFile()){
            // 如果文件名后缀是.dex，说明是原始dex文件
            if (name.endsWith(DEX_SUFFIX)) {
                // Raw dex file (not inside a zip/jar).
                try {
                    //调用loadDexFile()方法，加载dex文件，获得DexFile对象
                    dex = loadDexFile(file, optimizedDirectory);
                } catch (IOException ex) {
                    System.logE("Unable to load dex file: " + file, ex);
                }
            } else {
                // dex文件包含在其它文件中
                zip = file;

                try {
                    // 同样调用loadDexFile()方法
                    dex = loadDexFile(file, optimizedDirectory);
                } catch (IOException suppressed) {
                    // 和加载纯dex文件不同的是，会把异常添加到异常集合中
                    /*
                     * 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);
                }
            }
        } else {
            System.logW("ClassLoader referenced unknown path: " + file);
        }

        // 如果zip或者dex二者一直不为null，就把元素添加进来
        // 注意，现在添加进来的zip存在不为null也不包含dex文件的可能。
        if ((zip != null) || (dex != null)) {
```

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

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

通过上面的代码也可以看到，加载一个dex文件调用的是loadDexFile()方法。

reasonml 复制代码

```
# dalvik.system.DexPathList
private static DexFile loadDexFile(File file, File optimizedDirectory)
        throws IOException {
    // 如果缓存存放目录为null就直接创建一个DexFile对象返回
    if (optimizedDirectory == null) {
        return new DexFile(file);
    } else {
        // 根据缓存存放目录和文件名得到一个优化后的缓存文件路径
        String optimizedPath = optimizedPathFor(file, optimizedDirectory);
        // 调用DexFile的loadDex()方法来获取DexFile对象。
        return DexFile.loadDex(file.getPath(), optimizedPath, 0);
    }
}
```

DexFile的loadDex()方法如下，内部也做了一些调用。抛开这些细节来讲，它的作用就是加载DexFile文件，而且会把优化后的dex文件缓存到对应目录。

processing 复制代码

```
# dalvik.system.DexFile
static public DexFile loadDex(String sourcePathName, String outputPathName,
    int flags) throws IOException {

    /*
     * TODO: we may want to cache previously-opened DexFile objects.
     * The cache would be synchronized with close().  This would help
     * us avoid mapping the same DEX more than once when an app
     * decided to open it multiple times.  In practice this may not
     * be a real issue.
     */
    //loadDex方法内部就是调用了DexFile的一个构造方法
    return new DexFile(sourcePathName, outputPathName, flags);
}
```

```
private DexFile(String sourceName, String outputName, int flags) throws IOException {
    if (outputName != null) {
        try {
            String parent = new File(outputName).getParent();
            if (Libcore.os.getuid() != Libcore.os.stat(parent).st_uid) {
                throw new IllegalArgumentException("Optimized data directory " + parent
                        + " is not owned by the current user. Shared storage cannot protect"
                        + " your application from code injection attacks.");
            }
        } catch (ErrnoException ignored) {
            // assume we'll fail with a more contextual error later
        }
    }

    mCookie = openDexFile(sourceName, outputName, flags);
    mFileName = sourceName;
    guard.open("close");
    //System.out.println("DEX FILE cookie is " + mCookie + " sourceName=" + sourceName + " outputName=" +
}

private static long openDexFile(String sourceName, String outputName, int flags) throws IOException {
    // Use absolute paths to enable the use of relative paths when testing on host.
    return openDexFileNative(new File(sourceName).getAbsolutePath(),
                             (outputName == null) ? null : new File(outputName).getAbsolutePath(),
                             flags);
}

private static native long openDexFileNative(String sourceName, String outputName, int flags);
```

分析到这，我们可以小结一下：在BaseDexClassLoader对象构造方法内，创建了PathDexList对象。而在PathDexList构造方法内部，通过调用一系列方法，把直接包含或者间接包含dex的文件解压缩并缓存优化后的dex文件，通过PathDexList的成员变量 `Element[] dexElements` 来指向这个文件。

到此我们就分析完了BaseDexClassLoader的构造方法。

我们在之前讲Java类加载器的时候已经说了，类加载是按需加载，也就是说当明确需要使用class文件的时候才会加载。我们来看一下在Android中ClassLoader的loadeClass()方法。 与在Java中的loadClass()方法主要流程是类似的，不过因为Android中BootClassLoader是用Java代码写的，所以可以直接当作系统类加载器的parent类加载器。在Android中如果parent类加载器找不到类，最终还是会调用ClassLoader对象自己的findClass()方法。这个与在Java中逻辑是一样的。

```
                                                                    gradle  复制代码
```

```
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;
}
```

我们可以去看一下BaseDexClassLoader类的findClass()方法。

reasonml 复制代码

```
# dalvik.system.BaseDexClassLoader
@Override
protected Class<?> findClass(String name) throws ClassNotFoundException {
    List<Throwable> suppressedExceptions = new ArrayList<Throwable>();
    // 调用DexPathList对象的findClass()方法
    Class c = pathList.findClass(name, suppressedExceptions);
    if (c == null) {
        ClassNotFoundException cnfe = new ClassNotFoundException("Didn't find class \"" + name + "\" on p
        for (Throwable t : suppressedExceptions) {
            cnfe.addSuppressed(t);
        }
        throw cnfe;
    }
    return c;
}
```

可以看到，实际上BaseDexClassLoader调用的是其成员变量 `DexPathList pathList` 的findClass()方法。

```pgsql
# dalvik.system.DexPathList
public Class findClass(String name, List<Throwable> suppressed) {
    // 遍历Element
    for (Element element : dexElements) {
        // 获取DexFile，然后调用DexFile对象的loadClassBinaryName()方法来加载Class文件。
        DexFile dex = element.dexFile;

        if (dex != null) {
            Class clazz = dex.loadClassBinaryName(name, definingContext, suppressed);
            if (clazz != null) {
                return clazz;
            }
        }
    }
    if (dexElementsSuppressedExceptions != null) {
        suppressed.addAll(Arrays.asList(dexElementsSuppressedExceptions));
    }
    return null;
}
```

从上面的代码中我们也可以看到，实际上DexPathList最终还是遍历其自身的Element[]数组，获取DexFile对象来加载Class文件。我们之前讲DexPathList构造方法内是调用其makeDexElements()方法来创建Element[]数组的，而且也提到了如果zip文件或者dex文件二者之一不为null，就把元素添加进来，而添加进来的zip存在不为null也不包含dex文件的可能。从上面的代码中也可以看到，获取Class的时候跟这个zip文件没什么关系，调用的是dex文件对应的DexFile的方法来获取Class。

数组的遍历是有序的，假设有两个dex文件存放了二进制名称相同的Class，类加载器肯定就会加载在放在数组前面的dex文件中的Class。现在很多热修复技术就是把修复的dex文件放在DexPathList中Element[]数组的前面，这样就实现了修复后的Class抢先加载了，达到了修改bug的目的。

Android加载一个Class是调用DexFile的defineClass()方法。而不是调用ClassLoader的defineClass()方法。这一点与Java不同，毕竟Android虚拟机加载的dex文件，而不是class文件。

```wren
# dalvik.system.DexFile
public Class loadClassBinaryName(String name, ClassLoader loader, List<Throwable> suppressed) {
    return defineClass(name, loader, mCookie, suppressed);
}
```

```
private static Class defineClass(String name, ClassLoader loader, long cookie,
                                 List<Throwable> suppressed) {
    Class result = null;
    try {
        result = defineClassNative(name, loader, cookie);
    } catch (NoClassDefFoundError e) {
        if (suppressed != null) {
            suppressed.add(e);
        }
    } catch (ClassNotFoundException e) {
        if (suppressed != null) {
            suppressed.add(e);
        }
    }
    return result;
}
```

在Android中ClassLoader的defineClass()方法已经不能用了。可以看到它的方法体里直接抛出异常了，而且在BaseDexClassLoader中也没有重写这个方法，毕竟BaseDexClassLoader加载类的逻辑已经变了。

gradle 复制代码

```
# java.lang.ClassLoader
protected final Class<?> defineClass(String className, byte[] classRep, int offset, int length,
        ProtectionDomain protectionDomain) throws java.lang.ClassFormatError {
    throw new UnsupportedOperationException("can't load this type of class file");
}
```

好了，分析到此我们可以小结一下： Android中加载一个类是遍历PathDexList的Element[]数组，这个Element包含了DexFile，调用DexFile的方法来获取Class文件，如果获取到了Class，就跳出循环。否则就在下一个Element中寻找Class。

3 使用DexClassLoader加载类

写一个Test类。

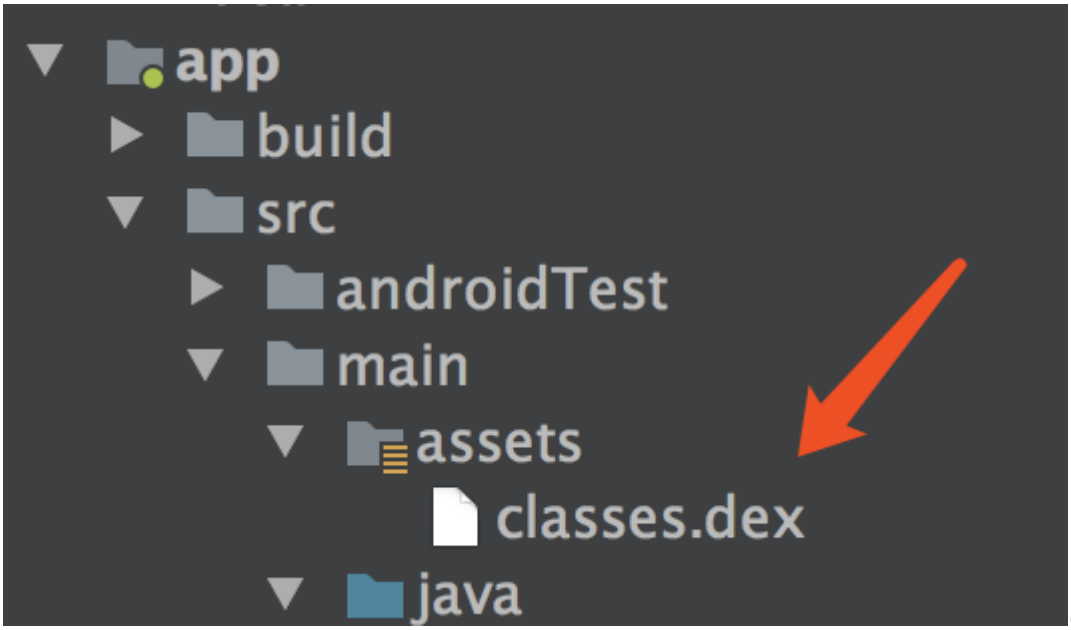
csharp 复制代码

```
```

```
class Test {
    public Test() {
    }

    public void print() {
        System.out.println("this is Test Class");
    }
}
```

通过 `javac` 命令生成class文件，然后通过Android中的 `dex` 工具生成dex文件。
为了方便测试，我把生成的dex文件放在了assets文件中。



dex文件.png

测试的时候，先把assets中的classes.dex文件复制到本地一个目录。 主要代码如下：

reasonml 复制代码

```
// 1. 先把assets中的classes.dex文件复制到一个本地目录中
File originDex = null;
try {
    InputStream open = getAssets().open("classes.dex");
    File dexOutputDir = getCacheDir();
    originDex = new File(dexOutputDir, "classes.dex");
    FileOutputStream fileOutputStream = new FileOutputStream(originDex);
    byte[] bytes = new byte[1024];
    int len = 0;
    while ((len = open.read(bytes)) != -1) {
        fileOutputStream.write(bytes, 0, len);
    }
    fileOutputStream.close();
    open.close();
```

```
} catch (IOException e) {
    e.printStackTrace();
}

// 2. 创建DexClassLoader加载dex文件中的类
if (originDex != null) {
    File dexOptimizeDir = getDir("dex", Context.MODE_PRIVATE);
    String dexOutputPath = dexOptimizeDir.getAbsolutePath();
    DexClassLoader dexClassLoader = new DexClassLoader(originDex.getAbsolutePath(), dexOutputPath, null,

    try {
        Class<?> clazz = dexClassLoader.loadClass("com.sososeen09.Test");
        System.out.println("loaded class: " + clazz);
        System.out.println("class loader: " + clazz.getClassLoader());
        System.out.println("class loader parent: " + clazz.getClassLoader().getParent());
        Constructor constructor = clazz.getConstructor();
        constructor.setAccessible(true);
        Object o = constructor.newInstance();
        Method print = clazz.getDeclaredMethod("print");
        print.setAccessible(true);
        print.invoke(o);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

打印结果：

```
stata 复制代码

I/System.out: loaded class: class com.sososeen09.Test
I/System.out: class loader: dalvik.system.DexClassLoader[DexPathList[[dex file "/data/data/com.sososeen09
I/System.out: class loader parent: dalvik.system.PathClassLoader[DexPathList[[zip file "/data/app/com.sos
I/System.out: this is Test Class
```

可以看到我们通过DexClassLoader对象正确的加载到了我们自己的dex文件中的类。

4 总结

Android中的类加载器是BootClassLoader、PathClassLoader、DexClassLoader，其中BootClassLoader是虚拟机加载系统类需要用到的，PathClassLoader是App加载自身dex文件中的类用到的，DexClassLoader可以加载直接或间接包含dex文件的文件，如APK等。

PathClassLoader和DexClassLoader都继承自BaseDexClassLoader，它的一个DexPathList类型的成员变量pathList很重要。DexPathList中有一个Element类型的数组dexElements，这个数组中存放了包含dex文件（对应的是DexFile）的元素。BaseDexClassLoader加载一个类，最后调用的是DexFile的方法进行加载的。

无论是热修复还是插件化技术中都利用了类加载机制，所以深入理解Android中的类加载机制对于理解这些技术的原理很有帮助。

参考文章

- Android解析ClassLoader（二）Android中的ClassLoader
- 一看你就懂，超详细java中的ClassLoader详解
- Android动态加载之ClassLoader详解
- Android动态加载入门 简单加载模式
- 唯一插件化Replugin源码及原理深度剖析–唯一Hook点原理
- Android插件化框架系列之类加载器