插件化中 Classloader 的加载 dex 分析

RXlee29764
2017年09月26日 17:26 · 阅读 3390

关注

原文链接： solart.cc

作者：solart

1、Java 类加载器

提到 Android 插件化，一个基础的知识点就是 Java 的类加载机制。这部分知识请参考深入探讨 Java 类加载器，以下摘录部分内容。

1.1 类加载器的树状组织结构

Java 中的类加载器大致可以分成两类，一类是系统提供的，另外一类则是由 Java 应用开发人员编写的。系统提供的类加载器主要有下面三个：

- 引导类加载器（bootstrap class loader）：它用来加载 Java 的核心库，是用原生代码来实现的，并不继承自 `java.lang.ClassLoader` 。
- 扩展类加载器（extensions class loader）：它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
- 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 `ClassLoader.getSystemClassLoader()` 来获取它。

1.2 类加载器的代理模式

类加载器 在尝试自己去查找某个类的字节代码并定义它时，会先代理给其父类加载器，由父类加载器先去尝试加载这个类，依次类推。在介绍代理模式背后的动机之前，首先需要说明一下 Java 虚拟机是如何判定两个 Java 类是相同的。Java 虚拟机不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。即便是同样的字节代码，被不同的类加载器加载之后所得到的类，也是不同的。

1.3 加载类的过程

在前面介绍类加载器的代理模式的时候，提到过类加载器会首先代理给其它类加载器来尝试加载某个类。这就意味着真正完成类的加载工作的类加载器和启动这个加载过程的类加载器，有可能不是同一个。真正完成类的加载工作是通过调用 `defineClass` 来实现的；而启动类的加载过程是通过调用 `loadClass` 来实现的。前者称为一个类的定义加载器（defining loader），后者称为初始加载器（initiating loader）。在 Java 虚拟机判断两个类是否相同的时候，使用的是类的定义加载器。也就是说，哪个类加载器启动类的加载过程并不重要，重要的是最终定义这个类的加载器。两种类加载器的关联之处在于：一个类的定义加载器是它引用的其它类的初始加载器。更多细节可参考 深入探讨 Java 类加载器。

2、DexClassLoader 和 PathClassLoader

Java的类加载是一个相对复杂的过程；它包括加载、验证、准备、解析和初始化五个阶段；对于开发者来说，可控性最强的是**加载阶段**；加载阶段主要完成三件事：

1. 根据一个类的全限定名来获取定义此类的二进制字节流
2. 将这个字节流所代表的静态存储结构转化为 JVM 方法区中的运行时数据结构
3. 在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口。

Android Framework简化了 通过一个类的全限定名获取描述次类的二进制字节流 这个过程；我们只需要告诉ClassLoader 一个 dex 文件或者 apk 文件的路径就能完成类的加载。

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、PathClassLoade r这些类加载器来加载类的，它们的不同之处是：

- **DexClassLoader:** 从 .jar 和 .apk 文件加载包含 classes.dex 条目的类。 这可以用于执行未作为应用程序的一部分安装的代码。
- **PathClassLoader:** 对本地文件系统中的文件和目录列表进行操作，但不尝试从网络加载类。 Android 将此类用于其系统类加载器及其应用程序类加载器。

看一下 DexClassLoader 和 PathClassLoader 细节上的区别:

```arduino 复制代码
public class DexClassLoader extends BaseDexClassLoader {
    public DexClassLoader(String dexPath, String optimizedDirectory,
            String libraryPath, ClassLoader parent) {
        super(dexPath, new File(optimizedDirectory), libraryPath, parent);
```

```
    }
}
```

java 复制代码

```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);
    }
}
```

以上可以看出来这两者只是对 `BaseDexClassLoader` 做了简单的封装，不同的是 `PathClassLoader` 在父类的构造中 optimizedDirectory 是 null，再来看 BaseDexClassLoader：

arduino 复制代码

```arduino
public class BaseDexClassLoader extends ClassLoader {
    private final DexPathList pathList;

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

在构造函数中创建了一个 `DexPathList` 实例，我们再来看看这个类的实现：

java 复制代码

```java
/*package*/ final class DexPathList {

    public DexPathList(ClassLoader definingContext, String dexPath,
            String libraryPath, File optimizedDirectory) {
        ...
        this.definingContext = definingContext;
        ArrayList<IOException> suppressedExceptions = new ArrayList<IOException>();
        this.dexElements = makeDexElements(splitDexPath(dexPath), optimizedDirectory,
                                           suppressedExceptions);
        ...
        this.nativeLibraryDirectories = splitLibraryPath(libraryPath);
    }

    /**
```

```
     * Makes an array of dex/resource path elements, one per element of
     * the given array.
     */
    private static Element[] makeDexElements(ArrayList<File> files, File optimizedDirectory,
                                             ArrayList<IOException> suppressedExceptions) {
        ArrayList<Element> elements = new ArrayList<Element>();
        /*
         * Open all files and load the (direct or contained) dex files
         * up front.
         */
        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.
                elements.add(new Element(file, true, null, null));
            } else if (file.isFile()){
                if (name.endsWith(DEX_SUFFIX)) {
                    // Raw dex file (not inside a zip/jar).
                    try {
                        dex = loadDexFile(file, optimizedDirectory);
                    } catch (IOException ex) {
                        System.logE("Unable to load dex file: " + file, ex);
                    }
                } else {
                    zip = file;

                    try {
                        dex = loadDexFile(file, optimizedDirectory);
                    } catch (IOException suppressed) {
                        suppressedExceptions.add(suppressed);
                    }
                }
            } else {
                System.logW("ClassLoader referenced unknown path: " + file);
            }

            if ((zip != null) || (dex != null)) {
                elements.add(new Element(file, false, zip, dex));
            }
        }

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

     /**
```

```
         * Constructs a {@code DexFile} instance, as appropriate depending
         * on whether {@code optimizedDirectory} is {@code null}.
         */
        private static DexFile loadDexFile(File file, File optimizedDirectory)
                throws IOException {
            if (optimizedDirectory == null) { //如果为空则直接创建DexFile
                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) {
            /*
             * Get the filename component of the path, and replace the
             * suffix with ".dex" if that's not already the suffix.
             *
             * We don't want to use ".odex", because the build system uses
             * that for files that are paired with resource-only jar
             * files. If the VM can assume that there's no classes.dex in
             * the matching jar, it doesn't need to open the jar to check
             * for updated dependencies, providing a slight performance
             * boost at startup. The use of ".dex" here matches the use on
             * files in /data/dalvik-cache.
             */
            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();
        }
    }
```

DexFileList 创建了一个 dex 资源路径的元素数组，在向这个数组添加元素时，会根据 optimizedDirectory 参数是否为 null 来区分创建 DexFile 对象。所以呢，PathClassLoader 最后调用的是 new DexFile(pathFile)，而DexClassLoader 调用的是 DexFile.loadDex(dexPathList[i], outputName, 0)。

这里还需要提一下的是官方文档中 DexClassLoader 中写到这样一段话：

> **Do not cache optimized classes on external storage.** External storage does not provide access controls necessary to protect your application from code injection attacks.

3、Dex 加载

前面我们提到了 ClassLoader 的代理模式，这种双亲委托的模式，很好的满足了 Android Framework 的系统代码的共享以及应用代码的隔离。

3.1 有几个 ClassLoader 实例

一般一个应用启动后有几个 ClassLoader 的实例呢？我们使用如下代码来打印一下 ClassLoader 的实例：

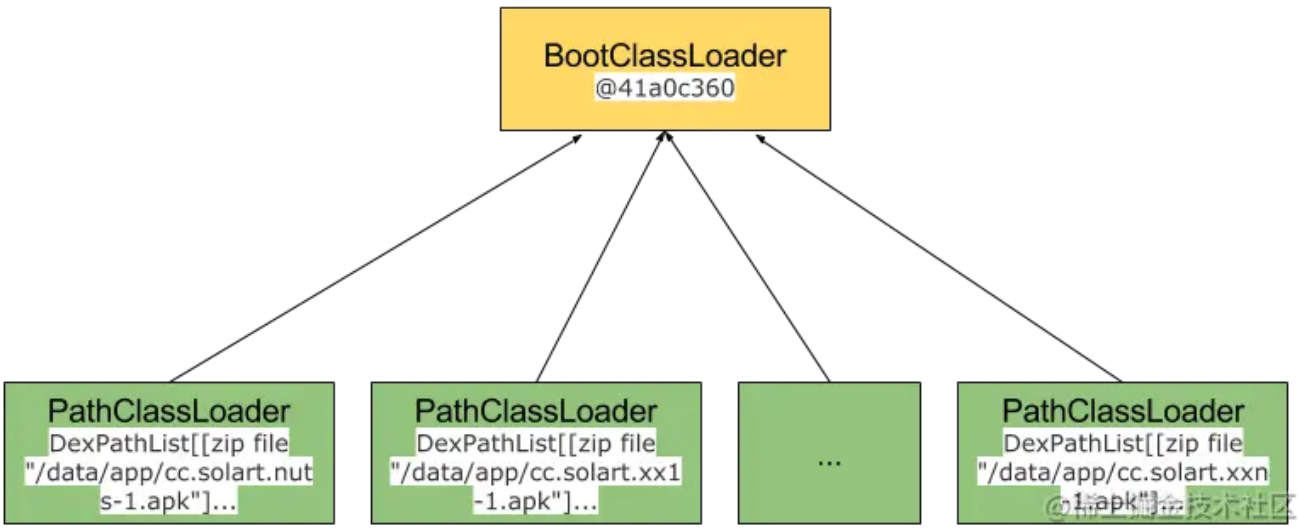
```ini 复制代码
@Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        int i = 1;
        ClassLoader classLoader = getClassLoader();
        if (classLoader != null){
            Log.i(TAG, "[onCreate] classLoader " + i + " : " + classLoader.toString());
            while (classLoader.getParent() != null){
                classLoader = classLoader.getParent();
                i++;
                Log.i(TAG,"[onCreate] classLoader " + i + " : " + classLoader.toString());
            }
        }
    }
```

Log 输出为：

```
javascript 复制代码
olart.nuts I/HomeActivity: [onCreate] classLoader 1 : dalvik.system.PathClassLoader[DexPathLi
olart.nuts I/HomeActivity: [onCreate] classLoader 2 : java.lang.BootClassLoader@41a0c360
```

可以看到有两个 ClassLoader 的实例输出，一个 `PathClassLoader` ，应用启动时创建的，用于加载 apk 中的类，另一个是 `BootClassLoader` ，系统启动时创建的，用于加载系统相关的类。

其实看到这里我们可以看出，任何运行的Android应用至少包含有两个 ClassLoader，每个应用中的 `PathClassLoader` 拥有同一个 `parent` 即是 `BootClassLoader` ，这样就保证了 系统代码共享以及应用代码隔离 ，如下图。



额外提一下的是，由于系统代码是全局共享的，那么这就产生了一个问题，那就是给了开发者Hook系统代码的机会，所以系统安全性是一个很大的考验，不知道Google是如何考量这个问题的，当然正因为这种机会的存在才给了插件化发展的契机。假设这样一个场景：两个应用都使用了Hook的机制，那么有可能导致一方Hook失效，所以一般在使用Hook后的服务时，一般要去做环境检测，看是否需要重新注入。这就属于Hook框架的博弈了。

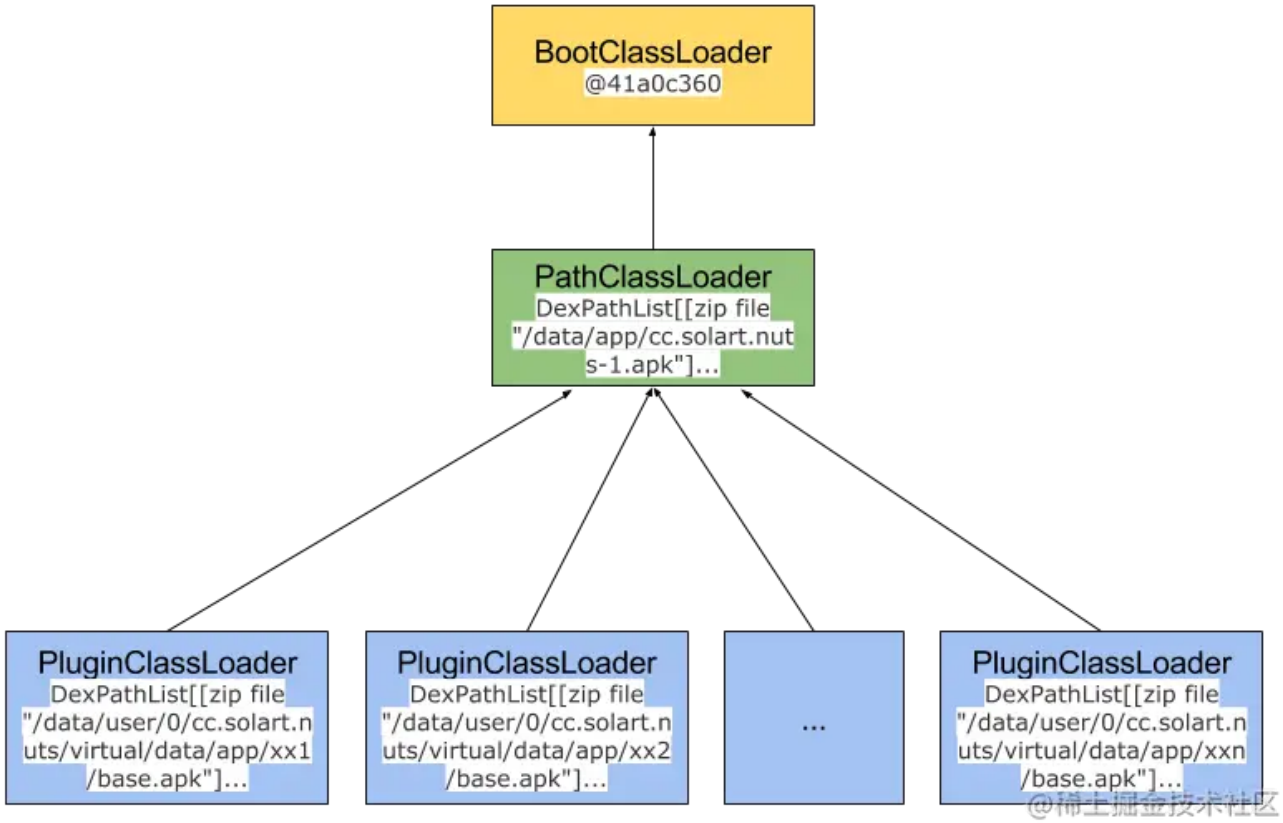
3.2 插件化 Dex 加载策略

我们可以借鉴这种思路应用到插件化的框架中，根据不同的出发点，通常插件化的 Dex 加载策略有两种：

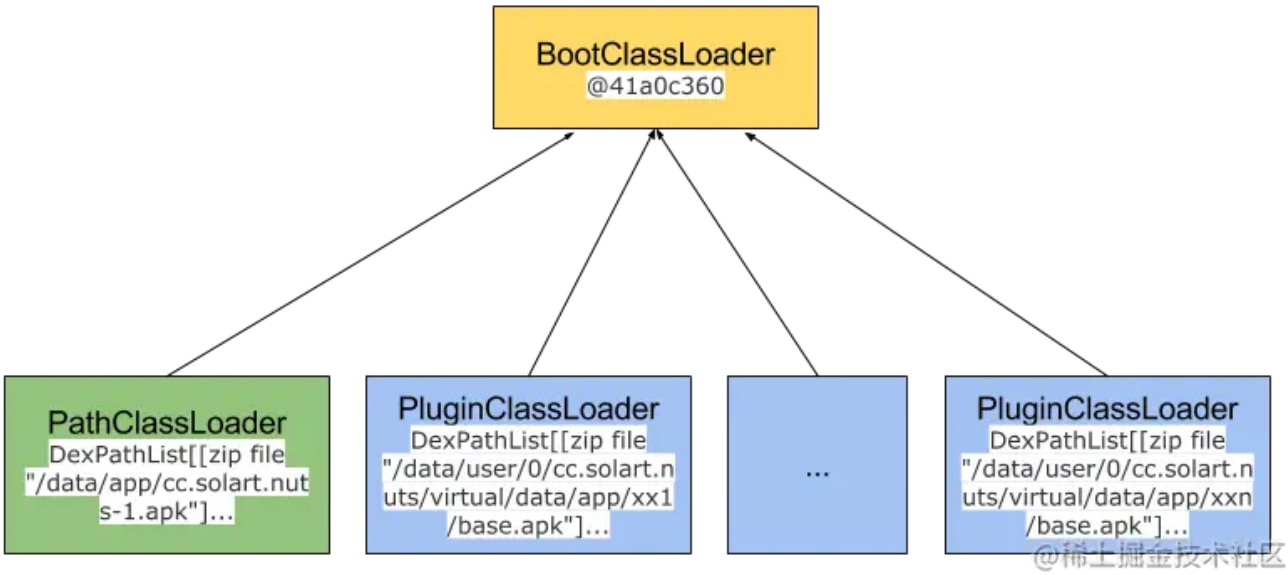
- 可以对每一个插件分配一个 `ClassLoader` 分别加载 `.dex` (这是目前最常见的方式)
- 也可以动态得把插件 `.dex` 加载到当前运行环境的 `Classloader` 中

我们先来看多 ClassLoader 加载的方案，多 ClassLoader 的方案，还可以细分为两种：一是每个自定义 ClassLoader 的 parent 为当前宿主应用的 ClassLoader 即是 PathClassLoader，这种

方案将宿主视为运行环境，插件需依赖宿主运行，插件之间互相隔离，如下图：



一是每个自定义 ClassLoader 的 parent 为 BootClassLoader，这种方案类似原生应用隔离的方案，宿主与插件、插件与插件互相独立，如下图：



多 ClassLoader 的难度在于 Hook 系统服务以及兼容性适配等工作，本身 ClassLoader 的代码量并不大。

我们再来看单ClassLoader方案，这种方案是委托给应用的PathClassLoader加载 `.dex` ，宿主与插件共享同一个 ClassLoader。

那怎么让 PathClassLoader 帮忙加载 dex 呢，我们在上面分析 BaseDexClassLoader 时，BaseDexClassLoader 在构造时生创建一个 `DexPathList` ，而 `DexPathList` 内部有一个叫做 `dexElements` 数组，我们要做的就是将 dex 文件插入到这个 `dexElements` 数组中，在 PathClassLoader 中查找类时，就会遍历这个数组中 DexFile 的信息，完成插件类的加载。

来看看 BaseDexClassLoader 中 `findClass` 的过程：

java 复制代码

```
public class BaseDexClassLoader extends ClassLoader {
    private final DexPathList pathList;

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        List<Throwable> suppressedExceptions = new ArrayList<Throwable>();
        //通过DexPathList查找类
        Class c = pathList.findClass(name, suppressedExceptions);
        if (c == null) {
            ClassNotFoundException cnfe = new ClassNotFoundException("Didn't find class \""
            for (Throwable t : suppressedExceptions) {
                cnfe.addSuppressed(t);
            }
            throw cnfe;
        }
        return c;
    }
}
```

可以看到，通过 `DexPathList` 完成查找Class，它的 `findClass` 方法如下：

java 复制代码

```
/*package*/ final class DexPathList {
    private final Element[] dexElements;

    public Class findClass(String name, List<Throwable> suppressed) {
        for (Element element : dexElements) {
            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;
        }
    }
```

看到这里就验证了我们将 dex 文件插入到 `dexElements` 数组中这个结论的可行性，接下的工作就是通过反射注入 dex 信息，这里就不在细说了。

回想一下单 ClassLoader 的方案，有没有觉得似曾相识？想必大家对 `MultiDex` 并不陌生吧，而这种单 ClassLoader 的方案与 `MultiDex` 方案如出一辙。所以这种方案实现起来代码量比较小(参照 MultiDex)，难度相对不大。再扯远一点，微信 Android 团队开源的热修复框架 Tinker 中也采用了类似的方式去处理，有兴趣的话可以去读读 SystemClassLoaderAdder 的代码。

3.3 Dex 加载策略的优劣

方案	优势	劣势
多ClassLoader	隔离性较好，热更新不必重启进程	存在多个相同的类包，加载 dex 文件大，实现交为复杂
单ClassLoader	实现相对简单，dex 文件可以做到比较小（毕竟没有重复类库）	隔离性弱（需要避免不同 dex 文件引用类库冲突），热更新需重启进程

在不同的出发点上，优势劣势各有不同，对于 Android 系统来说，程序间的隔离性显然要比其他因素重要的多，系统采用这种应用层隔离的方案是鸡蛋落在鸡窝里—刚刚好。在插件化框架中，多 ClassLoader 也是比较常见的一种方案，在多数情况下，多 ClassLoader 的方案要优于单一 ClassLoader。