**类加载器 ClassLoader**

在 JVM（Java Virtual Machine）中，类加载器（ClassLoader）负责将 .class 文件加载到内存中，并将其转换为 JVM 可以使用的 Class 对象。

JVM 中主要有以下几种类加载器（ClassLoader）：

**1. 引导类加载器（Bootstrap ClassLoader）**

加载内容：JDK 的核心类库（JAVA\_HOME/lib 下的类，如 rt.jar、java.lang、java.util 等系统类）。

由 C/C++ 实现，是 JVM 的一部分，JVM 的启动就是通过 Bottstrap ，不是 Java 类。

没有父类加载器（即 getParent() 返回 null）。

**2. 拓展类加载器（Extension ClassLoader）**

加载内容：JAVA\_HOME/lib/ext/ 目录或由 java.ext.dirs 指定的路径中的类。

实现类：sun.misc.Launcher$ExtClassLoader

父加载器：Bootstrap ClassLoader

**3. 应用类加载器（Application ClassLoader）**

加载内容：用户类路径（classpath）上的类，比如通过 -cp 或 -classpath 指定的类和 jar。

实现类：sun.misc.Launcher$AppClassLoader

父加载器：Extension ClassLoader

我们编写的大部分 Java 应用程序类就是由它加载的，ClassLoader.getSystemClassLoader 返回的就是它。

**自定义类加载器**

通过继承 java.lang.ClassLoader 自定义类加载器，实现自己的类加载逻辑，比如从网络、数据库、加密文件中加载类。

类加载顺序，比如：YourCustomClassLoader.findClass( classX.class)

YourCustomClassLoader （一般继承 ClassLoader 或 URLClassLoader）

|

(委托给父类)

↓

AppClassLoader

|

(委托给父类)

↓

ExtClassLoader

|

(委托给父类)

↓

Bootstrap ClassLoader

|

→ 尝试加载 classX.class

|

┌──┴──┐

↓ ↓

[成功] [失败]

↓

ExtClassLoader 执行 findClass() → 从 ext 路径找 classX

↓

┌──┴──┐

↓ ↓

[成功] [失败]

↓

AppClassLoader 执行 findClass() → 从 classpath 找 classX

↓

┌──┴──┐

↓ ↓

[成功] [失败]

↓

回退给 YourCustomClassLoader

（开始尝试自己加载 classX）

**双亲委派**

双亲委派是一种 类加载机制。

它的核心思想是：一个类加载器收到类加载请求时，不会自己去尝试加载，而是先把请求委托给它的“父类加载器”。只有当父加载器无法完成这个请求，它才会尝试自己加载。

为什么需要双亲委派？

避免类的重复加载

确保 Java 核心类（如 java.lang.String）不会被用户自定义的同名类替代。

加载流程（伪代码）

Class loadClass(String name) {

// 1. 检查是否已经加载过

if (已经加载过) return 缓存的Class对象;

// 2. 委托父类加载器尝试加载

try {

return parent.loadClass(name);

} catch (ClassNotFoundException e) {

// 3. 父加载器加载失败，自己尝试加载

return findClass(name);

}

}

Java 的类加载器是分层结构：

+---------------------------+

| Bootstrap ClassLoader | (C/C++ 实现)

+---------------------------+

↑ （委托） ↓ （回退）

+---------------------------+

| ExtClassLoader | --> 继承自 URLClassLoader

| (sun.misc.Launcher$ExtClassLoader)

+---------------------------+

↑ （委托） ↓ （回退）

+---------------------------+

| AppClassLoader | --> 继承自 URLClassLoader

| (sun.misc.Launcher$AppClassLoader)

+---------------------------+

↑ （委托） ↓ （回退）

+---------------------------+

| YourCustomClassLoader | --> 可继承 ClassLoader 或 URLClassLoader

+---------------------------+

[注]：

↑ 表示 loadClass() 方法的双亲委托链（优先给父类加载器加载）

↓ 表示当父类无法加载类时，逐层回退到下层加载器自己加载

加载顺序是自顶向下委托，如果父加载器无法加载才交给子类加载器。

**类加载**

**1. 隐式加载（Implicit Class Loading）**

隐式加载是指 当你在代码中首次使用某个类时，JVM 会自动加载这个类，不需要你手动调用任何方法。

示例：

val list = ArrayList<String>() // JVM 在这里自动加载 java.util.ArrayList 类

你没有写 Class.forName("java.util.ArrayList")

但 JVM 会自动检测到你用到了它，于是加载进内存

常见触发时机：

创建类实例（new）

访问静态字段或静态方法

继承/实现某个类或接口

反序列化对象（如果需要某个类）

**2. 显式加载（Explicit Class Loading）**

你通过反射等手段，主动告诉 JVM：“嘿，帮我加载这个类！” JVM 就会立即加载它，不管你有没有实际用它。

示例：

val clazz = Class.forName("com.example.MyClass")

这段代码会：

主动加载 com.example.MyClass 类；

触发静态初始化块（如果有）；

返回对应的 Class 对象。

常见显式加载方法：

Class.forName(String className)

ClassLoader.loadClass(String name)

ClassLoader.findClass(String name)

ClassLoader.defineClass(...) 字节码转换成类对象

在你自己写插件加载器或框架时，如果你手动读取字节码文件（.class 或 .dex），可以用这个方法把它变成 JVM 中的 Class 对象：

val bytes = ... // 读取 .class 文件内容

val clazz = ClassLoader.defineClass("com.example.MyClass", bytes, 0, bytes.size)

**findClass、loadClass、forName 的区别**

ClassLoader.loadClass(name)：先委托父加载器加载类（触发双亲委派）。

ClassLoader.findClass(name)：不委托，直接由当前类加载器查找（一般自定义类加载器要重写它）。

Class.forName(name)：是调用 loadClass + 初始化类的快捷方式，属于 Class 工具方法。

**区别对比表：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 方法 | 所属类/接口 | 是否委托父类加载 | 是否初始化类 | 常用于 |
| loadClass(name) | ClassLoader | ✅ 是 | ❌ 否 | 加载一个类，但不初始化 |
| findClass(name) | ClassLoader | ❌ 否 | ❌ 否 | 自定义类加载逻辑 |
| Class.forName() | java.lang.Class | ✅ 是（底层用 loadClass） | ✅ 是 | 加载并初始化类（执行 <clinit> ） |

<clinit> 是类的初始化方法，由编译器自动生成，不是程序员手写的。用于执行类中静态变量初始化和静态代码块。

**类加载和初始化的过程**

一个类从**“加载”** 到 **“可以使用”** 的完整生命周期：**加载（Loading）→ 连接（Linking）→ 初始化（Initialization）**。

JVM 规范中，一个类从加载到完成使用，主要经过以下几个阶段：

加载（Loading）

↓

连接（Linking）

→ 验证（Verification）

→ 准备（Preparation）

→ 解析（Resolution）

↓

初始化（Initialization）

↓

使用（Use）

↓

卸载（Unload）

每个阶段发生了什么？

|  |  |
| --- | --- |
| 阶段 | 发生了什么 |
| 加载 | 字节码文件 -> Class 对象 |
| 验证 | 检查字节码合法性 |
| 准备 | 静态变量分配内存，默认值 |
| 解析 | 符号引用 → 真实引用 |
| 初始化 | 调用 <clinit> 函数，static 代码块执行，静态变量赋值 |

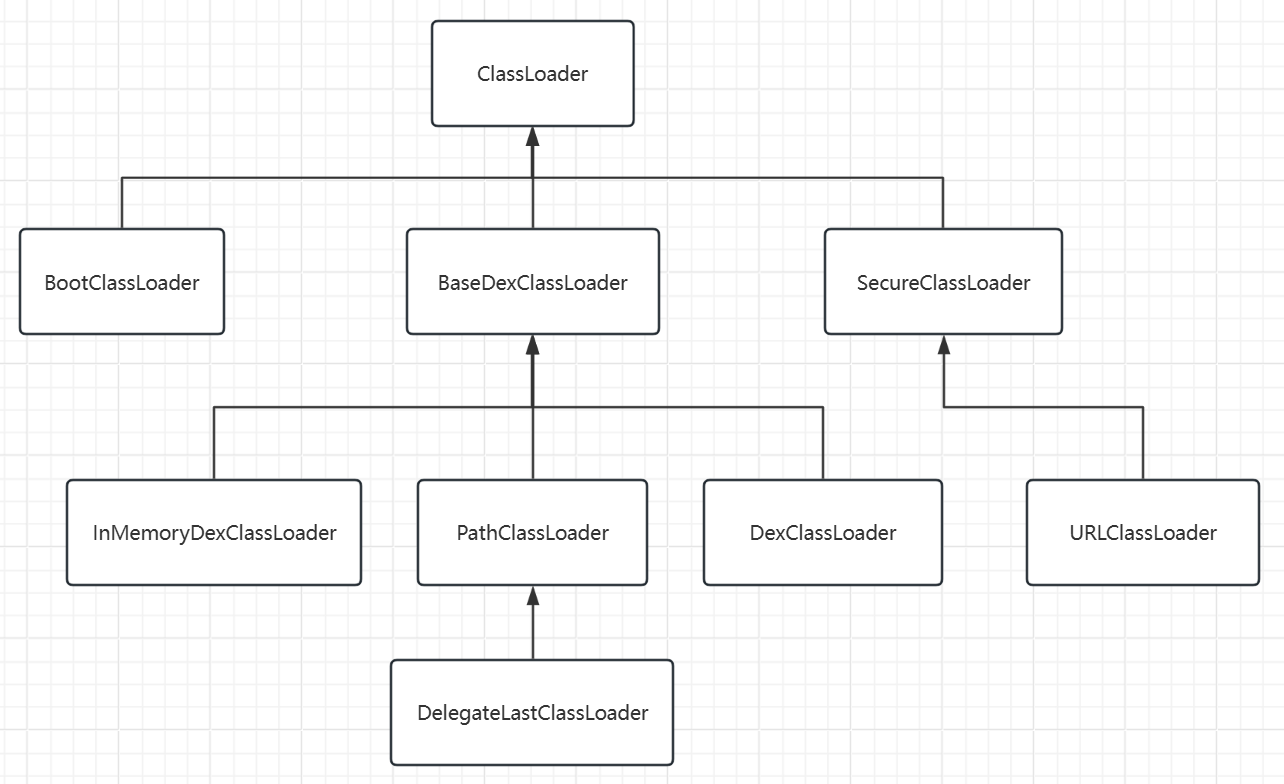
**Android 的类加载器**

Android 中的类加载器的作用就是将 .dex、.jar 或 .apk 中的类文件（字节码）加载到内存中，转化为 Android 虚拟机 可用的 Class<?> 对象。

常见类加载器（ClassLoader）

|  |  |
| --- | --- |
| 类加载器 | 说明 |
| ClassLoader | 所有类加载器的基类，负责加载类文件并将其转化为 Class 对象。 |
| BootClassLoader | 加载 Android 的核心类库（如 java.\*, android.\*） |
| BaseDexClassLoader | 是 DexClassLoader 和 PathClassLoader 的父类，专门处理 .dex 文件的加载 |
| SecureClassLoader | 引入了安全机制，限制了可以加载的类的范围和权限 |
| URLClassLoader | 用于从指定的 URL 路径加载类，常用于加载 JAR 文件 |
| PathClassLoader | 默认加载应用和系统库中的 dex |
| DexClassLoader | 加载外部 dex、apk、jar，支持动态加载（插件化常用） |
| InMemoryDexClassLoader | Android 8.0+，支持直接加载内存中的 dex |
| DelegateLastClassLoader | Android 8.0+，先尝试自己加载，再委托父类（破坏双亲委派） |

Android 系统中的 ClassLoader 的继承关系



在线UML建模：<https://www.processon.com/uml>

**BootClassLoader**

BootClassLoader 负责加载 Android 系统的核心类库，单例模式。

与 JVM 不同的是 在 Android 中，BootClassLoader 是由 Java 实现的，而不是通过 C++ 层实现。

源码如下：

1339 class BootClassLoader extends ClassLoader {

1340

1341 private static BootClassLoader instance;

1342

1343 @FindBugsSuppressWarnings("DP\_CREATE\_CLASSLOADER\_INSIDE\_DO\_PRIVILEGED")

1344 public static synchronized BootClassLoader getInstance() {

1345 if (instance == null) {

1346 instance = new BootClassLoader();

1347 }

1348

1349 return instance;

1350 }

1351

1352 public BootClassLoader() {

1353 super(null);

1354 }

1355

1356 @Override

1357 protected Class<?> findClass(String name) throws ClassNotFoundException {

1358 return Class.classForName(name, false, null);

1359 }

1360

1412 @Override

1413 protected Class<?> loadClass(String className, boolean resolve)

1414 throws ClassNotFoundException {

1415 Class<?> clazz = findLoadedClass(className);

1416

1417 if (clazz == null) {

1418 clazz = findClass(className);

1419 }

1420

1421 return clazz;

1422 }

1428 }

<http://aospxref.com/android-10.0.0_r47/xref/libcore/ojluni/src/main/java/java/lang/ClassLoader.java#1339>

**BaseDexClassLoader**

BaseDexClassLoader 是 PathClassLoader、DexClassLoader、InMemoryDexClassLoader 的父类，加载和解析 .dex 格式的类文件主要逻辑都是在 BaseDexClassLoader 完成的。

**SecureClassLoader**

SecureClassLoader 是 ClassLoader 的子类，主要用于增强类加载过程的安全性。它提供了在类加载过程中对类的访问控制和安全检查的能力。

SecureClassLoader 的常见子类是 URLClassLoader，用于加载 URL 指定 jar 文件的类和资源。

**PathClassLoader**

PathClassLoader 是 Android 应用的 默认类加载器，用于加载 APK 文件中的 .dex 文件。

app 如果没有加壳都是用 PathClassLoader，如果加壳了用的 DexClassLoader 比较多。

**DexClassLoader**

DexClassLoader 可以加载任意路径下的 dex，或者 jar、apk、zip 文件（包含classes.dex）。常用于插件化、热修复以及 dex 加壳。

**InMemoryDexClassLoader（Android 8.0+）**

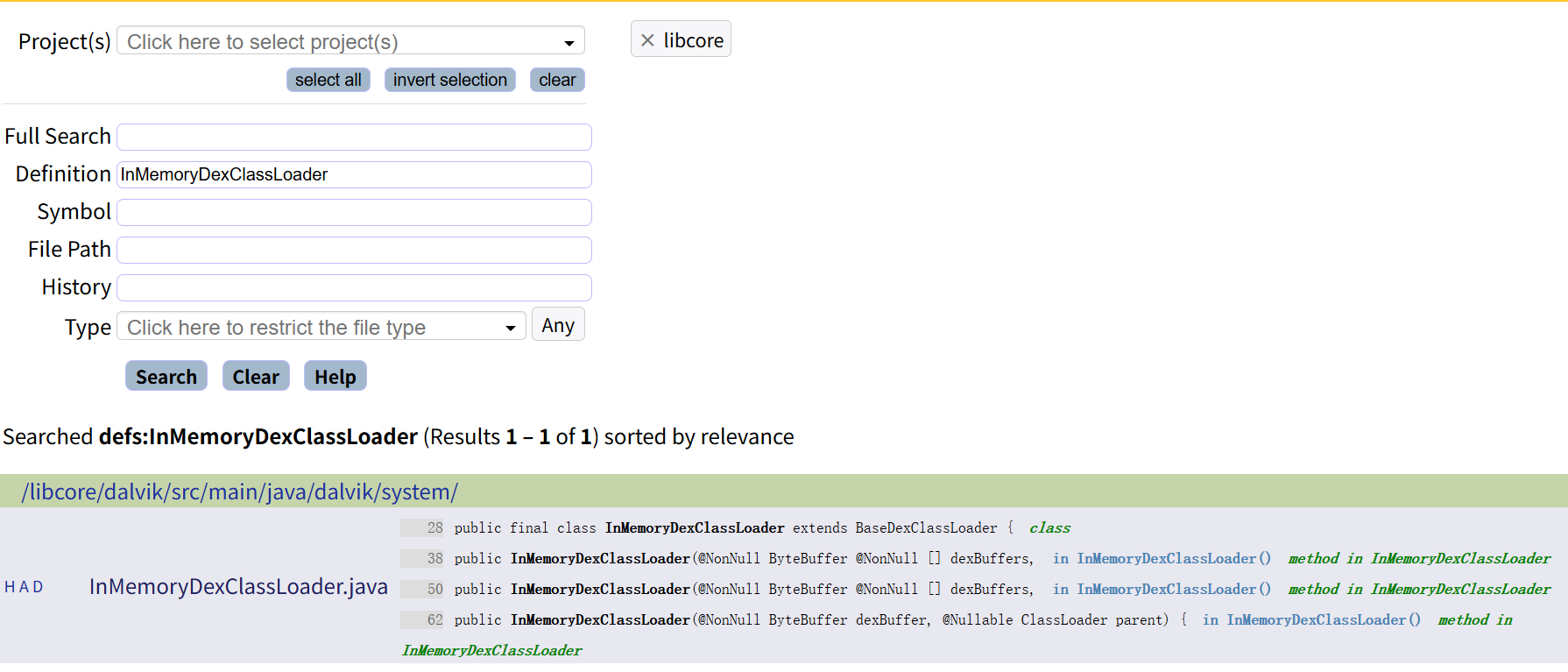
InMemoryDexClassLoader 支持直接从内存中的 ByteBuffer 加载 dex，适合做即时加载、代码保护（比如壳技术）；

val loader = InMemoryDexClassLoader(

byteBuffer, // ByteBuffer 里的 dex

parentClassLoader

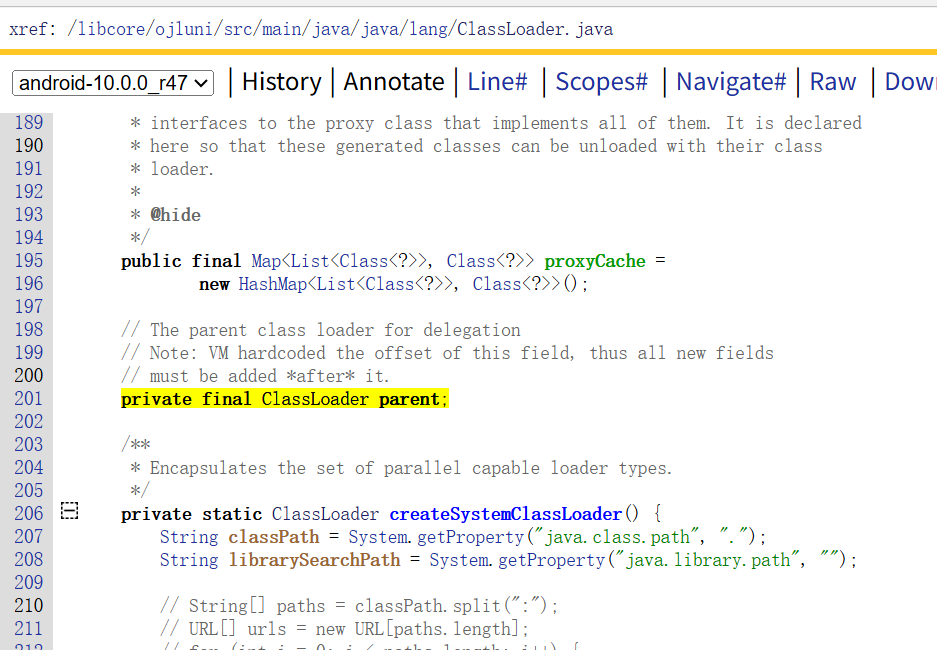
)



**Android 中默认的 ClassLoader 分层结构**

如果 app 没有加壳，我们编写的 Activity 类它在什么 ClassLoader 当中呢？默认的 ClassLoader 分层结构是怎么样的？

所有 ClassLoader 都有一个这样的属性，可以拿到 parent ClassLoader



<http://aospxref.com/android-10.0.0_r47/xref/libcore/ojluni/src/main/java/java/lang/ClassLoader.java>

打印一下 Activity 的 ClassLoader 链

package com.cyrus.example.classloader

import android.os.Bundle

import android.util.Log

import androidx.activity.ComponentActivity

import androidx.activity.compose.setContent

import androidx.compose.foundation.layout.\*

import androidx.compose.material3.\*

import androidx.compose.runtime.\*

import androidx.compose.ui.Modifier

import androidx.compose.ui.unit.dp

class ClassLoaderActivity : ComponentActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContent {

ClassLoaderScreen()

}

}

@Composable

fun ClassLoaderScreen() {

var output by remember { mutableStateOf("") }

Surface(

modifier = Modifier.fillMaxSize(),

color = MaterialTheme.colorScheme.background

) {

Column(

modifier = Modifier

.fillMaxSize()

.padding(16.dp),

verticalArrangement = Arrangement.spacedBy(16.dp)

) {

Button(onClick = {

// 打印并更新输出信息

val builder = StringBuilder()

var loader: ClassLoader? = this@ClassLoaderActivity::class.java.classLoader

var level = 0

while (loader != null) {

val line = "[$level] ${loader.javaClass.name}"

Log.d("ClassLoaderChain", line)

builder.appendLine(line)

loader = loader.parent

level++

}

output = builder.toString()

}) {

Text("打印 ClassLoader 链")

}

Text(

text = output,

style = MaterialTheme.typography.bodyMedium

)

}

}

}

}

输出如下：



可以看到默认是 PathClassLoader，parent 是 BootClassLoader。

**ClassLoader 的类列表**

PathClassLoader 源码如下：

25 public class PathClassLoader extends BaseDexClassLoader {

36

37 public PathClassLoader(String dexPath, ClassLoader parent) {

38 super(dexPath, null, null, parent);

39 }

40

63 public PathClassLoader(String dexPath, String librarySearchPath, ClassLoader parent) {

64 super(dexPath, null, librarySearchPath, parent);

65 }

69

70 @libcore.api.CorePlatformApi

71 public PathClassLoader(

72 String dexPath, String librarySearchPath, ClassLoader parent,

73 ClassLoader[] sharedLibraryLoaders) {

74 super(dexPath, librarySearchPath, parent, sharedLibraryLoaders);

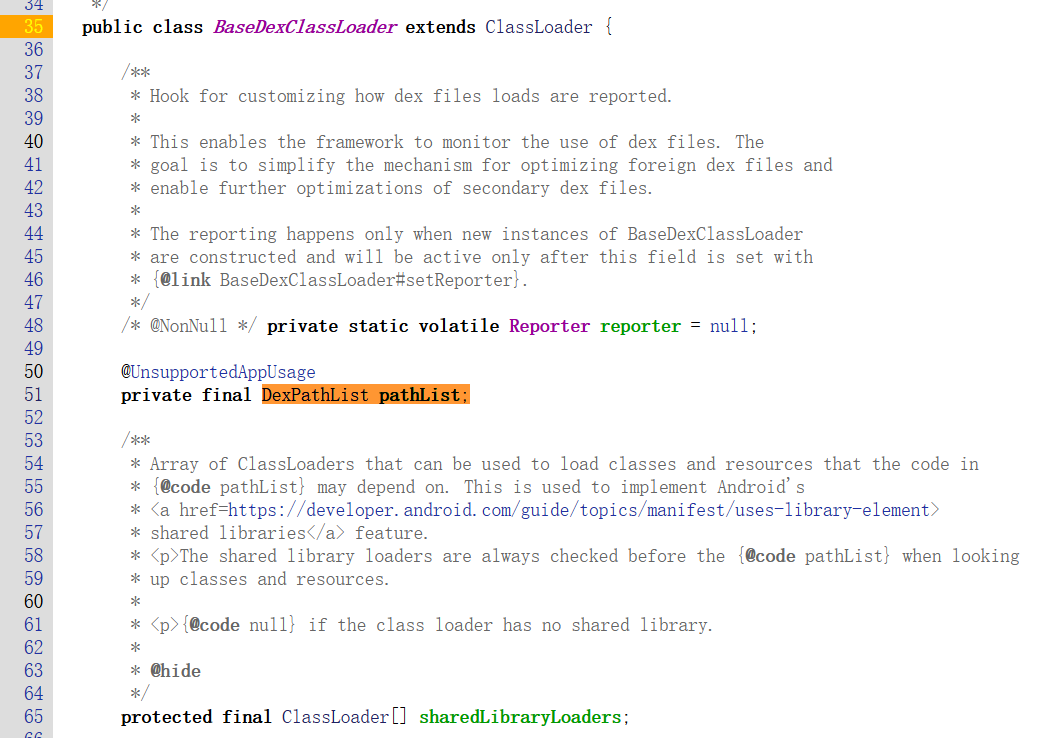
75 }

76 }

<http://aospxref.com/android-10.0.0_r47/xref/libcore/dalvik/src/main/java/dalvik/system/PathClassLoader.java>

主要实现都是在父类 BaseDexClassLoader 中。

在 BaseDexClassLoader 中有一个 DexPathList 类型字段 pathList



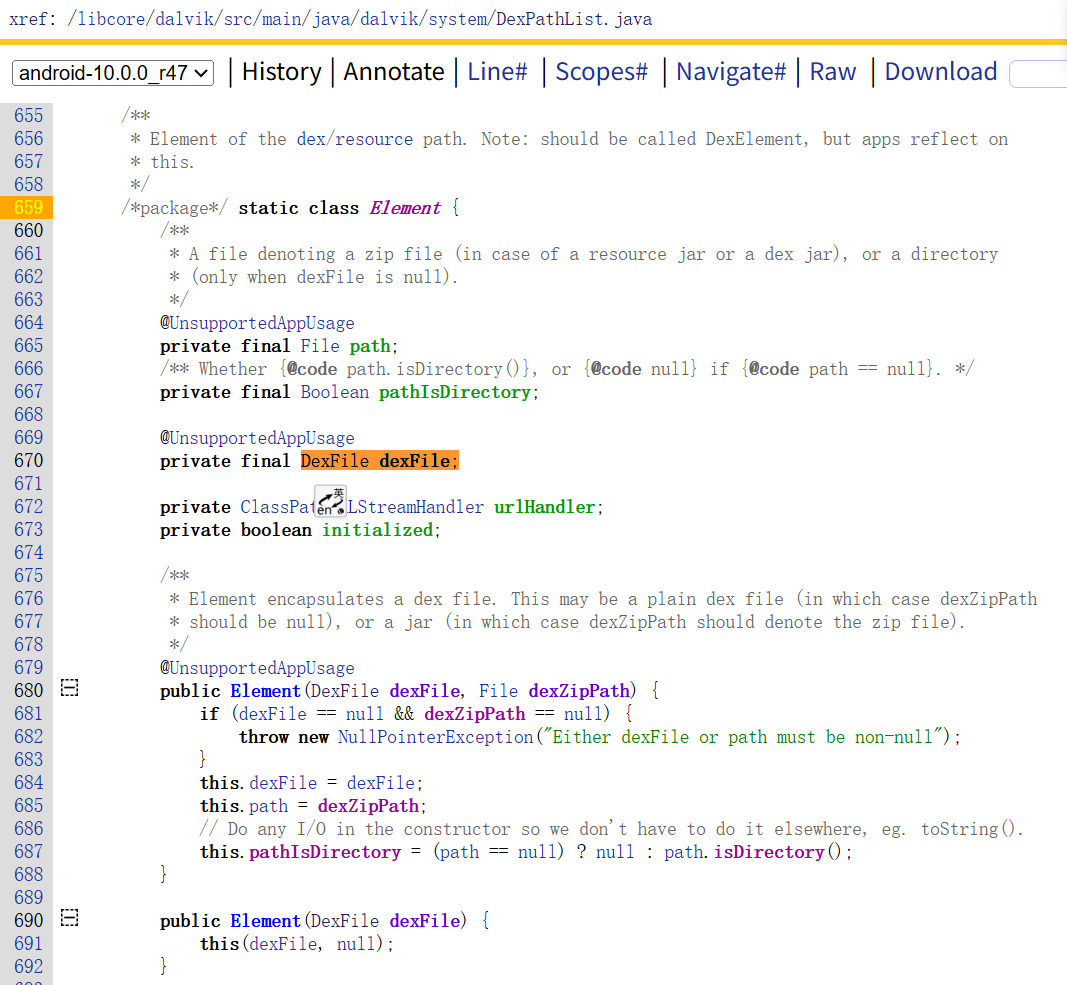
<http://aospxref.com/android-10.0.0_r47/xref/libcore/dalvik/src/main/java/dalvik/system/BaseDexClassLoader.java#51>

在 DexPathList 中有一个 Element[] dexElements 字段



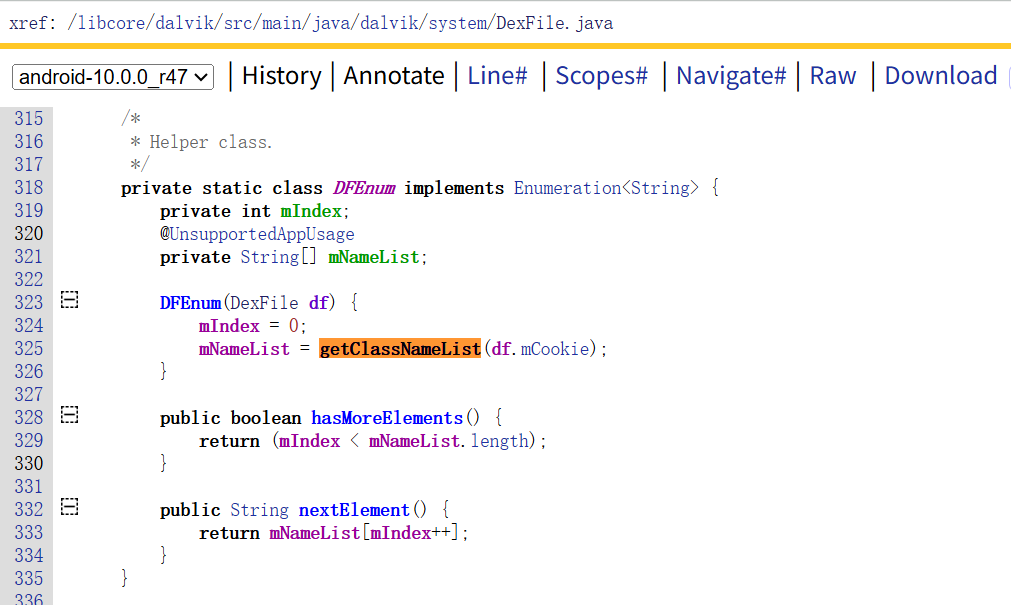
<http://aospxref.com/android-10.0.0_r47/xref/libcore/dalvik/src/main/java/dalvik/system/DexPathList.java#69>

当前 ClassLoader 中的所有 DexFile 对象就在 Element 中



<http://aospxref.com/android-10.0.0_r47/xref/libcore/dalvik/src/main/java/dalvik/system/DexPathList.java#659>

在 DexFile 中就包含了获取 dex 中类列表的 api，就是 getClassNameList，可以获得当前 ClassLoader 中包含的所有类



<http://aospxref.com/android-10.0.0_r47/xref/libcore/dalvik/src/main/java/dalvik/system/DexFile.java#432>

getClassNameList 方法需要一个参数 mCookie，mCookie 是一个 native 层的句柄（handle）或指针，它指向底层的 ART 或 Dalvik 虚拟机中加载的 DEX 文件对象，是 DexFile 对象的唯一标识。

@UnsupportedAppUsage

private static native String[] getClassNameList(Object cookie);

在 Android 中，可以通过反射的方式从 ClassLoader 中获取所有类的名称。

以下是 Kotlin 示例代码，该方法接受一个 ClassLoader 参数，利用反射访问 pathList -> dexElements -> dexFile -> getClassNameList()，从而获取类列表：

@SuppressLint("DiscouragedPrivateApi")

fun getAllClassesFromClassLoader(classLoader: ClassLoader): List<String> {

val classNames = mutableListOf<String>()

try {

// 获取 BaseDexClassLoader 的 pathList 字段

val pathListField = Class.forName("dalvik.system.BaseDexClassLoader")

.getDeclaredField("pathList")

pathListField.isAccessible = true

val pathList = pathListField.get(classLoader)

// 获取 pathList 中的 dexElements 字段

val dexElementsField = pathList.javaClass.getDeclaredField("dexElements")

dexElementsField.isAccessible = true

val dexElements = dexElementsField.get(pathList) as Array<\*>

for (element in dexElements) {

// 获取 dexElement 中的 dexFile 字段

val dexFileField = element!!::class.java.getDeclaredField("dexFile")

dexFileField.isAccessible = true

val dexFile = dexFileField.get(element)

val mCookieField = dexFile.javaClass.getDeclaredField("mCookie")

mCookieField.isAccessible = true

val mCookie = mCookieField.get(dexFile)

// 调用 dexFile.getClassNameList()

val getClassNameListMethod = dexFile.javaClass.getDeclaredMethod("getClassNameList", Any::class.java)

getClassNameListMethod.isAccessible = true

val result = getClassNameListMethod.invoke(dexFile, mCookie)

// 将结果添加到列表中

if (result is Array<\*>) {

classNames.addAll(result.filterIsInstance<String>())

}

}

} catch (e: Exception) {

e.printStackTrace()

}

return classNames

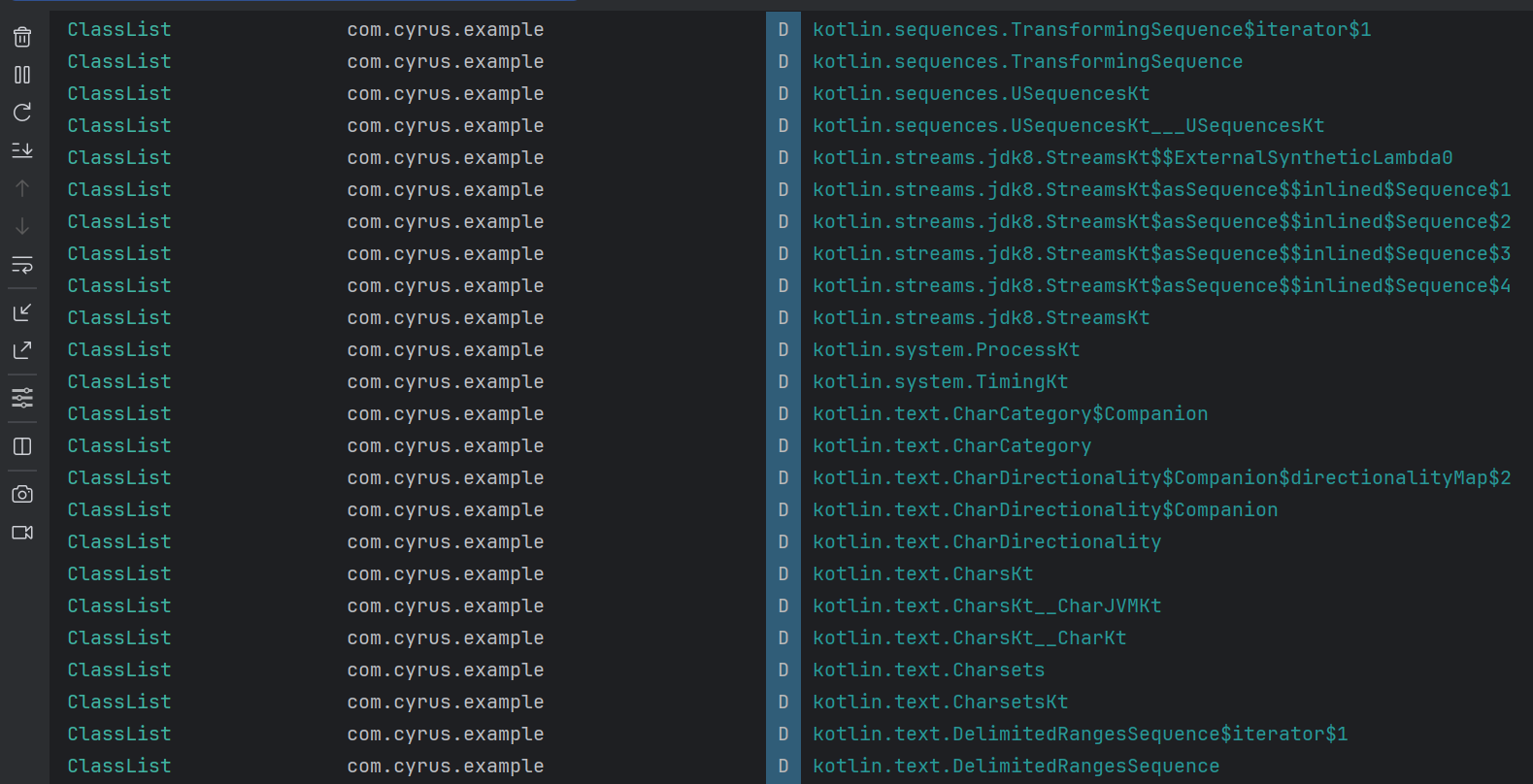
}

注意：只有继承于 BaseDexClassLoader 的子类才可以，因为 BaseDexClassLoader 的子类才有 pathList

输出如下：



这样就可以直观的看到 ClassLoader 中有哪些类，而且会比 frida 枚举类列表更全，因为这是包含了 dex 文件所有类，不只是已经加载的类。



或者通过 dexFile.entries() 方法迭代 dex 文件的类列表。

使用示例（Kotlin）：假设你想从当前 App 的 dexFile 中打印所有类名：

val apkPath = context.applicationInfo.sourceDir

val dexFile = DexFile(apkPath)

val entries = dexFile.entries()

while (entries.hasMoreElements()) {

val className = entries.nextElement()

Log.d("DexClass", className)

}

**使用 Frida 打印类列表**

关于 Frida 的使用可以参考这篇文章：[使用 Frida Hook Android App](https://cyrus-studio.github.io/blog/posts/%E4%BD%BF%E7%94%A8-frida-hook-android-app/)

**1. 打印 app 中已经加载的类**

编写 firda 脚本：classloader\_utils.js

/\*\*

\* 打印 app 中已经加载的类

\*

\* @param filterPrefix 如果提供了前缀，就按前缀过滤

\*/

function printLoadedClasses(filterPrefix) {

Java.perform(() => {

const loadedClasses = [];

Java.enumerateLoadedClasses({

onMatch(className) {

// 如果提供了前缀，就按前缀过滤

if (!filterPrefix || className.startsWith(filterPrefix)) {

loadedClasses.push(className);

}

},

onComplete() {

loadedClasses.sort();

console.log(`\n=== Loaded Classes (${loadedClasses.length}) ===`);

loadedClasses.forEach(className => console.log(className));

console.log("=== End ===\n");

}

});

});

}

// 打印全部已加载的类

// printLoadedClasses();

// 打印指定包名开头的类，比如你的应用类

// printLoadedClasses("com.cyrus.example");

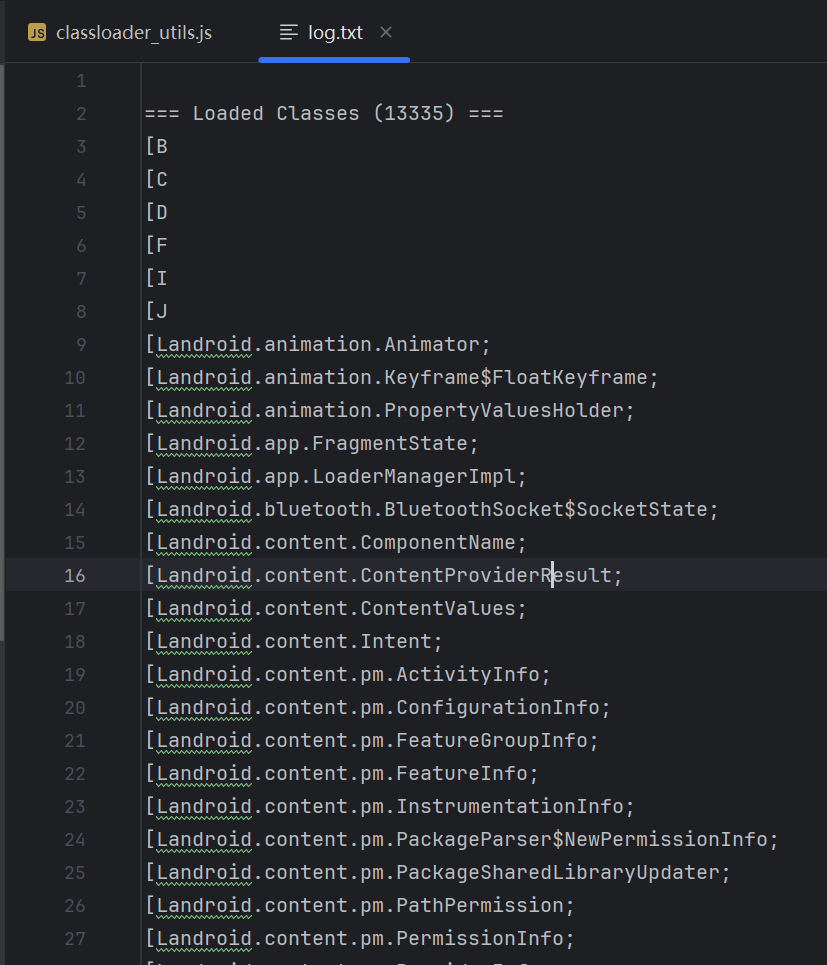
// frida -H 127.0.0.1:1234 -F -l classloader\_utils.js

// frida -H 127.0.0.1:1234 -F -l classloader\_utils.js -o log.txt

指向脚本并输入日志到 log.txt

frida -H 127.0.0.1:1234 -F -l classloader\_utils.js -o log.txt

日志输入如下：



**2. 打印所有 ClassLoader 中的类列表**

/\*\*

\* 获取 classLoader 的类列表

\* @param classLoader

\* @returns {\*[]}

\*/

function getAllClassesFromClassLoader(classLoader) {

const classNames = [];

const BaseDexClassLoader = Java.use("dalvik.system.BaseDexClassLoader");

if (!BaseDexClassLoader.class.isInstance(classLoader)) {

console.log("Not a BaseDexClassLoader, skip.");

return [];

}

const castedLoader = Java.cast(classLoader, BaseDexClassLoader);

const pathList = castedLoader.pathList.value;

const dexElements = pathList.dexElements.value;

for (let i = 0; i < dexElements.length; i++) {

const element = dexElements[i];

const dexFile = element.dexFile.value;

if (dexFile == null) {

console.log(` [dexFile] is null for element ${i}, skipping.`);

continue;

}

const enumeration = dexFile.entries();

while (enumeration.hasMoreElements()) {

const className = enumeration.nextElement();

classNames.push(className);

}

}

return classNames;

}

function dumpAllClassLoaderClasses() {

Java.perform(() => {

console.log(">> Enumerating all ClassLoaders...");

Java.enumerateClassLoaders({

onMatch: function (loader) {

console.log("\n[ClassLoader] => " + loader);

const classes = getAllClassesFromClassLoader(loader);

console.log(" [class count] => " + classes.length);

if (classes.length > 0) {

classes.forEach(cls => console.log(" " + cls));

}

},

onComplete: function () {

console.log(">> Done.");

}

});

});

}

// 列出所有类加载器的类列表

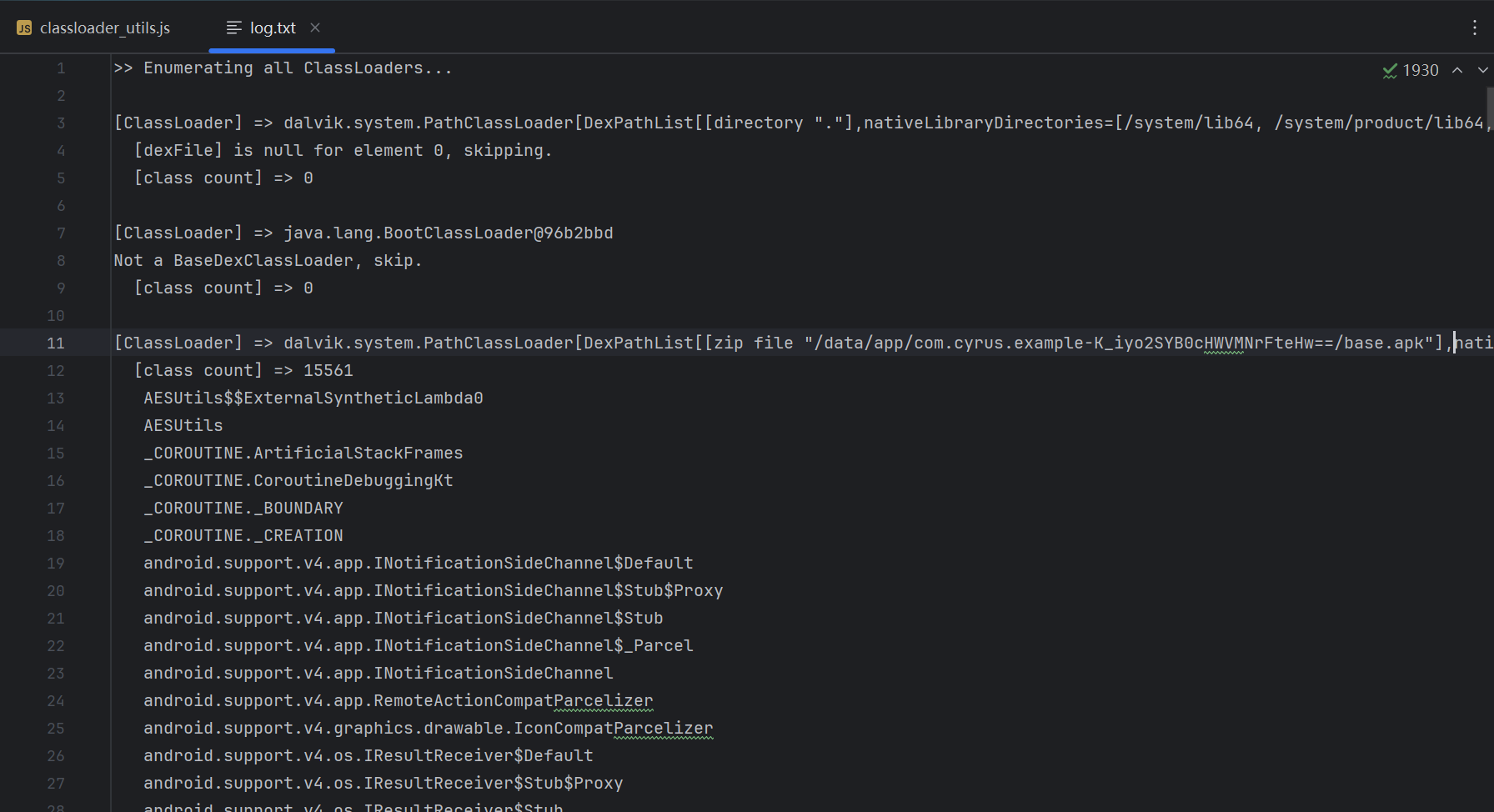
// dumpAllClassLoaderClasses()

执行脚本

frida -H 127.0.0.1:1234 -F -l classloader\_utils.js -o log.txt

执行 dumpAllClassLoaderClasses() 方法

日志输出如下：



**Android 中的类加载**

当类没有被加载过时，类加载器的 loadClass 方法最终会调用到 findClass 方法。

Android 应用的类加载器（PathClassLoader）继承自 BaseDexClassLoader。

BaseDexClassLoader 重写了 findClass()，直接把查找任务委托给内部的 pathList.findClass(name, ...)。



<http://aospxref.com/android-10.0.0_r47/xref/libcore/dalvik/src/main/java/dalvik/system/BaseDexClassLoader.java#findClass>

DexPathList 维护了一组 dexElements（多个 .dex / .jar / .apk）。

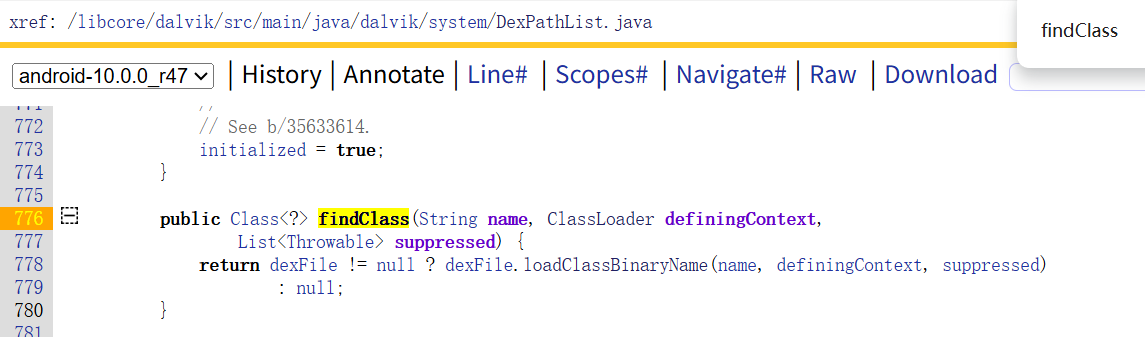
遍历每个 Element，调用 element.findClass() 逐个查找类。



<http://aospxref.com/android-10.0.0_r47/xref/libcore/dalvik/src/main/java/dalvik/system/DexPathList.java#536>

每个 Element 可能关联一个 DexFile。

如果当前元素持有 dexFile，就调用 dexFile.loadClassBinaryName(...)。



<http://aospxref.com/android-10.0.0_r47/xref/libcore/dalvik/src/main/java/dalvik/system/DexPathList.java#776>

将类名传入 defineClass(...) 方法，开始进入 native 加载流程。



<http://aospxref.com/android-10.0.0_r47/xref/libcore/dalvik/src/main/java/dalvik/system/DexFile.java?fi=loadClassBinaryName#loadClassBinaryName>

调用 native 方法 defineClassNative(...)，完成真正的类定义。

private static native Class defineClassNative(String name, ClassLoader loader, Object cookie, DexFile dexFile) throws ClassNotFoundException, NoClassDefFoundError;

JNI 实现（C++），调用 ART 虚拟机内部逻辑：

查找 dex 中的 class def

验证字节码

创建 Class 对象

注册到类加载器关联的 ClassTable 中

如果类成功加载，将 Java 层的 Class<?> 返回给上层。

**类加载过程调用链**

PathClassLoader.loadClass()

↓

BaseDexClassLoader.findClass()

↓

DexPathList.findClass()

↓

DexPathList.Element.findClass()

↓

DexFile.loadClassBinaryName()

↓

DexFile.defineClass()

↓

DexFile.defineClassNative() ← native 通过 ART 注册类

**完整源码**

开源地址：<https://github.com/CYRUS-STUDIO/AndroidExample>