#### 腾讯---系统如何加载一个dex文件，他的底层原理是怎么实现的

本专栏专注分享大型Bat面试知识，后续会持续更新，喜欢的话麻烦点击一个star

**面试官: 系统如何加载一个dex文件，他的底层原理是怎么实现的**

**心理分析**：面试官想知道你是否有过对dex加载相关经验。此题主要为tinker热修复做铺垫。dex加载与热修复是有关系的，求职者一定要注意 面试官后续会面试到tinker

**求职者:**应该从DexClassLoader 加载出发

**DexClassLoader 是加载包含classes.dex文件的jar文件或者apk文件**； 通过构造函数发现需要一个应用私有的，可写的目录去缓存优化的classes。可以用使用File dexoutputDir = context.getDir(“dex”,0);创建一个这样的目录，不要使用外部缓存，以保护你的应用被代码注入。

其源码如下：

public classDexClassLoaderextendsBaseDexClassLoader {
 public DexClassLoader(String dexPath, String optimizedDirectory,
 String libraryPath, ClassLoader parent) {
 super(dexPath, new File(optimizedDirectory), libraryPath, parent);
 }
}

##### 再解释下几个构造函数参数的意义：

1. dexpath为jar或apk文件目录。
2. optimizedDirectory为优化dex缓存目录。
3. libraryPath包含native lib的目录路径。
4. parent父类加载器。

然后执行的是父类的构造函数：

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

##### BaseDexClassLoader 的构造函数如下：

```

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

```

第一句调用的还是父类的构造函数，也就是ClassLoader的构造函数：

protected ClassLoader(ClassLoader parentLoader) {
 this(parentLoader, false);
 }

 ClassLoader(ClassLoader parentLoader, boolean nullAllowed) {
 if (parentLoader == null && !nullAllowed) {
 throw new NullPointerException(“parentLoader == null && !nullAllowed”);
 }
 parent = parentLoader;
}

该构造函数把传进来的父类加载器赋给了私有变量parent。

再来看第二句：

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

pathList为该类的私有成员变量，类型为DexPathList，进入到DexPathList函数：

 Constructs an instance.
79 \*
80 \* @param definingContext the context in which any as-yet unresolved
81 \* classes should be defined
82 \* @param dexPath list of dex/resource path elements, separated by
83 \* {@code File.pathSeparator}
84 \* @param libraryPath list of native library directory path elements,
85 \* separated by {@code File.pathSeparator}
86 \* @param optimizedDirectory directory where optimized {@code .dex} files
87 \* should be found and written to, or {@code null} to use the default
88 \* system directory for same
89 \*/
90 public DexPathList(ClassLoader definingContext, String dexPath,
91 String libraryPath, File optimizedDirectory) {
92
93 if (definingContext == null) {
94 throw new NullPointerException("definingContext == null");
95 }
96
97 if (dexPath == null) {
98 throw new NullPointerException("dexPath == null");
99 }
100
101 if (optimizedDirectory != null) {
102 if (!optimizedDirectory.exists()) {
103 throw new IllegalArgumentException(
104 "optimizedDirectory doesn't exist: "
105 + optimizedDirectory);
106 }
107
108 if (!(optimizedDirectory.canRead()
109 && optimizedDirectory.canWrite())) {
110 throw new IllegalArgumentException(
111 "optimizedDirectory not readable/writable: "
112 + optimizedDirectory);
113 }
114 }
115
116 this.definingContext = definingContext;
117
118 ArrayList<IOException> suppressedExceptions = new ArrayList<IOException>();
119 // save dexPath for BaseDexClassLoader
120 this.dexElements = makePathElements(splitDexPath(dexPath), optimizedDirectory,
1 suppressedExceptions);
122
123 // Native libraries may exist in both the system and
124 // application library paths, and we use this search order:
125 //
126 // 1. This class loader's library path for application libraries (libraryPath):
127 // 1.1. Native library directories
128 // 1.2. Path to libraries in apk-files
129 // 2. The VM's library path from the system property for system libraries
130 // also known as java.library.path
131 //
132 // This order was reversed prior to Gingerbread; see http://b/2933456.
133 this.nativeLibraryDirectories = splitPaths(libraryPath, false);
134 this.systemNativeLibraryDirectories =
135 splitPaths(System.getProperty("java.library.path"), true);
136 List<File> allNativeLibraryDirectories = new ArrayList<>(nativeLibraryDirectories);
137 allNativeLibraryDirectories.addAll(systemNativeLibraryDirectories);
138
139 this.nativeLibraryPathElements = makePathElements(allNativeLibraryDirectories, null,
140 suppressedExceptions);
141
142 if (suppressedExceptions.size() > 0) {
143 this.dexElementsSuppressedExceptions =
144 suppressedExceptions.toArray(new IOException[suppressedExceptions.size()]);
145 } else {
146 dexElementsSuppressedExceptions = null;
147 }
148 }

前面是一些对于传入参数的验证，然后调用了makeDexElements。

private static Element[] makeDexElements(ArrayList<File> files, File optimizedDirectory,
 ArrayList<IOException> suppressedExceptions) {
ArrayList<Element> elements = new ArrayList<Element>();
 for (File file : files) {
 File zip = null;
 DexFile dex = null;
 String name = file.getName();

 if (name.endsWith(DEX\_SUFFIX)) { //dex文件处理
 // 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 if (name.endsWith(APK\_SUFFIX) || name.endsWith(JAR\_SUFFIX)
 || name.endsWith(ZIP\_SUFFIX)) { //apk，jar，zip文件处理
 zip = file;

 try {
 dex = loadDexFile(file, optimizedDirectory);
 } catch (IOException suppressed) {
 suppressedExceptions.add(suppressed);
 }
 } else if (file.isDirectory()) {
 elements.add(new Element(file, true, null, null));
 } else {
 System.logW(“Unknown file type for: ” + file);
 }

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

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

不管是dex文件，还是apk文件最终加载的都是loadDexFile，跟进这个函数：

如果optimizedDirectory为null就会调用openDexFile(fileName, null, 0);加载文件。

否则调用DexFile.loadDex(file.getPath(), optimizedPath, 0);

而这个函数也只是直接调用new DexFile(sourcePathName, outputPathName, flags);

里面调用的也是openDexFile(sourceName, outputName, flags);

所以最后都是调用openDexFile，跟进这个函数：

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);
 }
}

private static int openDexFile(String sourceName, String outputName,
 int flags) throws IOException {
 return openDexFileNative(new File(sourceName).getCanonicalPath(),
 (outputName == null) ? null : new File(outputName).getCanonicalPath(),
 flags);

而这个函数调用的是so的openDexFileNative这个函数。打开成功则返回一个cookie。

接下来就是分析native函数的实现部分了。

**-openDexFileNative**函数

static void Dalvik\_dalvik\_system\_DexFile\_openDexFileNative(const u4\* args,JValue\* pResult)
{
 ……………
if (hasDexExtension(sourceName)
 && dvmRawDexFileOpen(sourceName, outputName, &pRawDexFile, false) == 0) {
 ALOGV(“Opening DEX file ‘%s’ (DEX)”, sourceName);

 pDexOrJar = (DexOrJar\*) malloc(sizeof(DexOrJar));
 pDexOrJar->isDex = true;
 pDexOrJar->pRawDexFile = pRawDexFile;
 pDexOrJar->pDexMemory = NULL;
 } else if (dvmJarFileOpen(sourceName, outputName, &pJarFile, false) == 0) {
 ALOGV(“Opening DEX file ‘%s’ (Jar)”, sourceName);

 pDexOrJar = (DexOrJar\*) malloc(sizeof(DexOrJar));
 pDexOrJar->isDex = false;
 pDexOrJar->pJarFile = pJarFile;
 pDexOrJar->pDexMemory = NULL;
 } else {
 ALOGV(“Unable to open DEX file ‘%s’”, sourceName);
 dvmThrowIOException(“unable to open DEX file”);
 }
 ……………
}

这里会根据是否为dex文件或者包含classes.dex文件的jar，分别调用函数dvmRawDexFileOpen和dvmJarFileOpen来处理，最终返回一个DexOrJar的结构。

首先来看dvmRawDexFileOpen函数的处理：

int dvmRawDexFileOpen(const char\* fileName, const char\* odexOutputName,
    RawDexFile\*\* ppRawDexFile, bool isBootstrap)
{
    .................
    dexFd = open(fileName, O\_RDONLY);
    if (dexFd < 0) goto bail;

    /\* If we fork/exec into dexopt, don't let it inherit the open fd. \*/
    dvmSetCloseOnExec(dexFd);

    //校验前8个字节的magic是否正确，然后把校验和保存到adler32
    if (verifyMagicAndGetAdler32(dexFd, &adler32) < 0) {
        ALOGE("Error with header for %s", fileName);
        goto bail;
    }
    //得到文件修改时间以及文件大小
   if (getModTimeAndSize(dexFd, &modTime, &fileSize) < 0) {
        ALOGE("Error with stat for %s", fileName);
        goto bail;
    }
    .................
    //调用函数dexOptCreateEmptyHeader,构造了一个DexOptHeader结构体，写入fd并返回
    optFd = dvmOpenCachedDexFile(fileName, cachedName, modTime,
        adler32, isBootstrap, &newFile, /\*createIfMissing=\*/true);

    if (optFd < 0) {
        ALOGI("Unable to open or create cache for %s (%s)",
                fileName, cachedName);
        goto bail;
    }
    locked = true;

       //如果成功生了opt头
    if (newFile) {
        u8 startWhen, copyWhen, endWhen;
        bool result;
       off\_t dexOffset;

        dexOffset = lseek(optFd, 0, SEEK\_CUR);
        result = (dexOffset > 0);

        if (result) {
            startWhen = dvmGetRelativeTimeUsec();
            // 将dex文件中的内容写入文件的当前位置，也就是从dexOffset的偏移处开始写
            result = copyFileToFile(optFd, dexFd, fileSize) == 0;
            copyWhen = dvmGetRelativeTimeUsec();
        }

        if (result) {
            //对dex文件进行优化
            result = dvmOptimizeDexFile(optFd, dexOffset, fileSize,
                fileName, modTime, adler32, isBootstrap);
        }

        if (!result) {
            ALOGE("Unable to extract+optimize DEX from '%s'", fileName);
            goto bail;
        }

        endWhen = dvmGetRelativeTimeUsec();
        ALOGD("DEX prep '%s': copy in %dms, rewrite %dms",
            fileName,
            (int) (copyWhen - startWhen) / 1000,
            (int) (endWhen - copyWhen) / 1000);
    }

     //dvmDexFileOpenFromFd这个函数最主要在这里干了两件事情
     // 1.将优化后得dex文件（也就是odex文件）通过mmap映射到内存中，并通过mprotect修改它的映射内存为只读权限
     // 2.将映射为只读的这块dex数据中的内容全部提取到DexFile这个数据结构中去
    if (dvmDexFileOpenFromFd(optFd, &pDvmDex) != 0) {
        ALOGI("Unable to map cached %s", fileName);
        goto bail;
    }

    if (locked) {
        /\* unlock the fd \*/
       if (!dvmUnlockCachedDexFile(optFd)) {
            /\* uh oh -- this process needs to exit or we'll wedge the system \*/
            ALOGE("Unable to unlock DEX file");
            goto bail;
        }
        locked = false;
    }

    ALOGV("Successfully opened '%s'", fileName);
    //填充结构体 RawDexFile
    \*ppRawDexFile = (RawDexFile\*) calloc(1, sizeof(RawDexFile));
    (\*ppRawDexFile)->cacheFileName = cachedName;
   (\*ppRawDexFile)->pDvmDex = pDvmDex;
    cachedName = NULL;      // don't free it below
    result = 0;

bail:
    free(cachedName);
    if (dexFd >= 0) {
        close(dexFd);
    }
    if (optFd >= 0) {
        if (locked)
            (void) dvmUnlockCachedDexFile(optFd);
        close(optFd);
    }
    return result;
}

最后成功的话，填充RawDexFile。

##### dvmJarFileOpen的代码处理也是差不多的。

int dvmJarFileOpen(const char\* fileName, const char\* odexOutputName,
    JarFile\*\* ppJarFile, bool isBootstrap)
{
    ...
    ...
    ...
    //调用函数dexZipOpenArchive来打开zip文件，并缓存到系统内存里
    if (dexZipOpenArchive(fileName, &archive) != 0)
        goto bail;
    archiveOpen = true;
    ...
    //这行代码设置当执行完成后，关闭这个文件句柄
    dvmSetCloseOnExec(dexZipGetArchiveFd(&archive));
    ...
    //优先处理已经优化了的Dex文件
    fd = openAlternateSuffix(fileName, "odex", O\_RDONLY, &cachedName);
    ...
    //从压缩包里找到Dex文件，然后打开这个文件
    entry = dexZipFindEntry(&archive, kDexInJarName);
    ...
    //把未经过优化的Dex文件进行优化处理，并输出到指定的文件
    if (odexOutputName == NULL) {
                cachedName = dexOptGenerateCacheFileName(fileName,
                                kDexInJarName);
    }
    ...
    //创建缓存的优化文件
    fd = dvmOpenCachedDexFile(fileName, cachedName,
                    dexGetZipEntryModTime(&archive, entry),
                    dexGetZipEntryCrc32(&archive, entry),
                    isBootstrap, &newFile, /\*createIfMissing=\*/true);
    ...
    //调用函数dexZipExtractEntryToFile从压缩包里解压文件出来
    if (result) {
                    startWhen = dvmGetRelativeTimeUsec();
                    result = dexZipExtractEntryToFile(&archive, entry, fd) == 0;
                    extractWhen = dvmGetRelativeTimeUsec();
                 }
    ...
    //调用函数dvmOptimizeDexFile对Ｄex文件进行优化处理
    if (result) {
                    result = dvmOptimizeDexFile(fd, dexOffset,
                                dexGetZipEntryUncompLen(&archive, entry),
                                fileName,
                                dexGetZipEntryModTime(&archive, entry),
                                dexGetZipEntryCrc32(&archive, entry),
                                isBootstrap);
                }
    ...
    //调用函数dvmDexFileOpenFromFd来缓存dex文件
    //并分析文件的内容。比如标记是否优化的文件，通过签名检查Dex文件是否合法
    if (dvmDexFileOpenFromFd(fd, &pDvmDex) != 0) {
        ALOGI("Unable to map %s in %s", kDexInJarName, fileName);
        goto bail;
    }
    ...
    //保存文件到缓存里，标记这个文件句柄已经保存到缓存
    if (locked) {
        /\* unlock the fd \*/
        if (!dvmUnlockCachedDexFile(fd)) {
            /\* uh oh -- this process needs to exit or we'll wedge the system \*/
            ALOGE("Unable to unlock DEX file");
            goto bail;
        }
        locked = false;
    }
    ...
     //设置一些相关信息返回前面的函数处理。
    \*ppJarFile = (JarFile\*) calloc(1, sizeof(JarFile));
    (\*ppJarFile)->archive = archive;
    (\*ppJarFile)->cacheFileName = cachedName;
    (\*ppJarFile)->pDvmDex = pDvmDex;
    cachedName = NULL;      // don't free it below
    result = 0;
    ...

}

最后成功的话，填充JarFile。