DexClassLoader 分析

2017年9月6日 星期三　　下午5:14

DexClassLoader 被调用后， 会调用父类的DexPathList传递参数，然后DexPathList 又回去调用makeDexElements ， 接下来就是调用loaddexFile 加载dex。

而这个loaddexFile 内部又继续调用DexFile.loaddex

```
    */
    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.
         */
        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);
}
```

可以看到内部又调用了一个openDexFile，同时返回了一个mCookie ，找找代码，发现openDexFile 有两种，一种是给文件名，一种是给一块内存地址，所以后者可以用来不落地加载。

```
static void Dalvik_dalvik_system_DexFile_openDexFile(const u4* args,
    JValue* pResult)
{
    StringObject* sourceNameObj = (StringObject*) args[0];
    StringObject* outputNameObj = (StringObject*) args[1];
    DexOrJar* pDexOrJar = NULL;
    JarFile* pJarFile;
    RawDexFile* pRawDexFile;
    char* sourceName;
    char* outputName;

    if (sourceNameObj == NULL) {
        dvmThrowNullPointerException("sourceName == null");
        RETURN_VOID();
    }

    sourceName = dvmCreateCstrFromString(sourceNameObj);
    if (outputNameObj != NULL)
        outputName = dvmCreateCstrFromString(outputNameObj);
    else
        outputName = NULL;

    /*
     * We have to deal with the possibility that somebody might try to
     * open one of our bootstrap class DEX files.  The set of dependencies
     * will be different, and hence the results of optimization might be
     * different, which means we'd actually need to have two
```

```
versions of
     * the optimized DEX: one that only knows about part of the boot class
     * path, and one that knows about everything in it.  The latter might
     * optimize field/method accesses based on a class that appeared later
     * in the class path.
     *
     * We can't let the user-defined class loader open it and start using
     * the classes, since the optimized form of the code skips some of
     * the method and field resolution that we would ordinarily do, and
     * we'd have the wrong semantics.
     *
     * We have to reject attempts to manually open a DEX file from the boot
     * class path.  The easiest way to do this is by filename, which works
     * out because variations in name (e.g. "/system/framework/./ext.jar")
     * result in us hitting a different dalvik-cache entry.  It's also fine
     * if the caller specifies their own output file.
     */
    if (dvmClassPathContains(gDvm.bootClassPath, sourceName)) {
        ALOGW("Refusing to reopen boot DEX '%s'", sourceName);
        dvmThrowIOException(
            "Re-opening BOOTCLASSPATH DEX files is not allowed");
        free(sourceName);
        free(outputName);
        RETURN_VOID();
    }

    /*
     * Try to open it directly as a DEX if the name ends with ".dex".
     * If that fails (or isn't tried in the first place), try it as a
     * Zip with a "classes.dex" inside.
     */
    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);
        ALOGV("Unable to open DEX file '%s'", sourceName);
        dvmThrowIOException("unable to open DEX file");
    }

    if (pDexOrJar != NULL) {
        pDexOrJar->fileName = sourceName;
        addToDexFileTable(pDexOrJar);
    } else {
        free(sourceName);
    }

    RETURN_PTR(pDexOrJar);
}
```

再往下就调用dexopt 对dexfile 进行优化了，这里的主要技巧就是看那些注释的说明，然后调用dexopt 对dexfile进行优化。

```
*/
bool dvmOptimizeDexFile(int fd, off_t dexOffset, long dexLength,
    const char* fileName, u4 modWhen, u4 crc, bool isBootstrap)
{
    const char* lastPart = strrchr(fileName, '/');
    pid = fork();
    if (pid == 0) {
        static const int kUseValgrind = 0;
        static const char* kDexOptBin = "/bin/dexopt";
        static const char* kValgrinder = "/usr/bin/valgrind";
        static const int kFixedArgCount = 10;
        static const int kValgrindArgCount = 5;
        static const int kMaxIntLen = 12;   // '-'+10dig+'\0' -OR- 0x+8dig
        int bcpSize = dvmGetBootPathSize();
        int argc = kFixedArgCount + bcpSize
            + (kValgrindArgCount * kUseValgrind);
        const char* argv[argc+1];           // last entry is NULL
        char values[argc][kMaxIntLen];
        char* execFile;
        const char* androidRoot;
        int flags;

        /* change process groups, so we don't clash with ProcessManager */
        setpgid(0, 0);

        /* full path to optimizer */
        androidRoot = getenv("ANDROID_ROOT");
        if (androidRoot == NULL) {
            ALOGW("ANDROID_ROOT not set, defaulting to /system");
            androidRoot = "/system";
        }
        execFile = (char*)alloca(strlen(androidRoot) + strlen(kDexOptBin) + 1);
        strcpy(execFile, androidRoot);
        strcat(execFile, kDexOptBin);

        /*
         * Create arg vector.
         */
        int curArg = 0;
        int curArg = 0;

        if (kUseValgrind) {
            /* probably shouldn't ship the hard-coded path */
            argv[curArg++] = (char*)kValgrinder;
            argv[curArg++] = "--tool=memcheck";
            argv[curArg++] = "--leak-check=yes";        // check for leaks too
            argv[curArg++] = "--leak-resolution=med";   // increase from 2 to 4
            argv[curArg++] = "--num-callers=16";        // default is 12
            assert(curArg == kValgrindArgCount);
        }
        argv[curArg++] = execFile;

        argv[curArg++] = "--dex";

        sprintf(values[2], "%d", DALVIK_VM_BUILD);
        argv[curArg++] = values[2];

        sprintf(values[3], "%d", fd);
        argv[curArg++] = values[3];

        sprintf(values[4], "%d", (int) dexOffset);
        argv[curArg++] = values[4];

        sprintf(values[5], "%d", (int) dexLength);
        argv[curArg++] = values[5];

        argv[curArg++] = (char*)fileName;

        sprintf(values[7], "%d", (int) modWhen);
        argv[curArg++] = values[7];

        sprintf(values[8], "%d", (int) crc);
        argv[curArg++] = values[8];

        flags = 0;
        if (gDvm.dexOptMode != OPTIMIZE_MODE_NONE) {
            flags |= DEXOPT_OPT_ENABLED;
            if (gDvm.dexOptMode == OPTIMIZE_MODE_ALL)
                flags |= DEXOPT_OPT_ALL;
        }
        if (gDvm.classVerifyMode != VERIFY_MODE_NONE) {
            flags |= DEXOPT_VERIFY_ENABLED;
            if (gDvm.classVerifyMode == VERIFY_MODE_ALL)
                flags |= DEXOPT_VERIFY_ALL;
        }
        if (isBootstrap)
            flags |= DEXOPT_IS_BOOTSTRAP;
        if (gDvm.generateRegisterMaps)
            flags |= DEXOPT_GEN_REGISTER_MAPS;
        sprintf(values[9], "%d", flags);
        argv[curArg++] = values[9];

        assert(((!kUseValgrind && curArg == kFixedArgCount) ||
                ((kUseValgrind && curArg == kFixedArgCount+kValgrindArgCount))));

        ClassPathEntry* cpe;
        for (cpe = gDvm.bootClassPath; cpe->ptr != NULL; cpe++) {
            argv[curArg++] = cpe->fileName;
        }
        assert(curArg == argc);

        argv[curArg] = NULL;

        if (kUseValgrind)
            execv(kValgrinder, const_cast<char**>(argv));
        else
            execv(execFile, const_cast<char**>(argv));
```