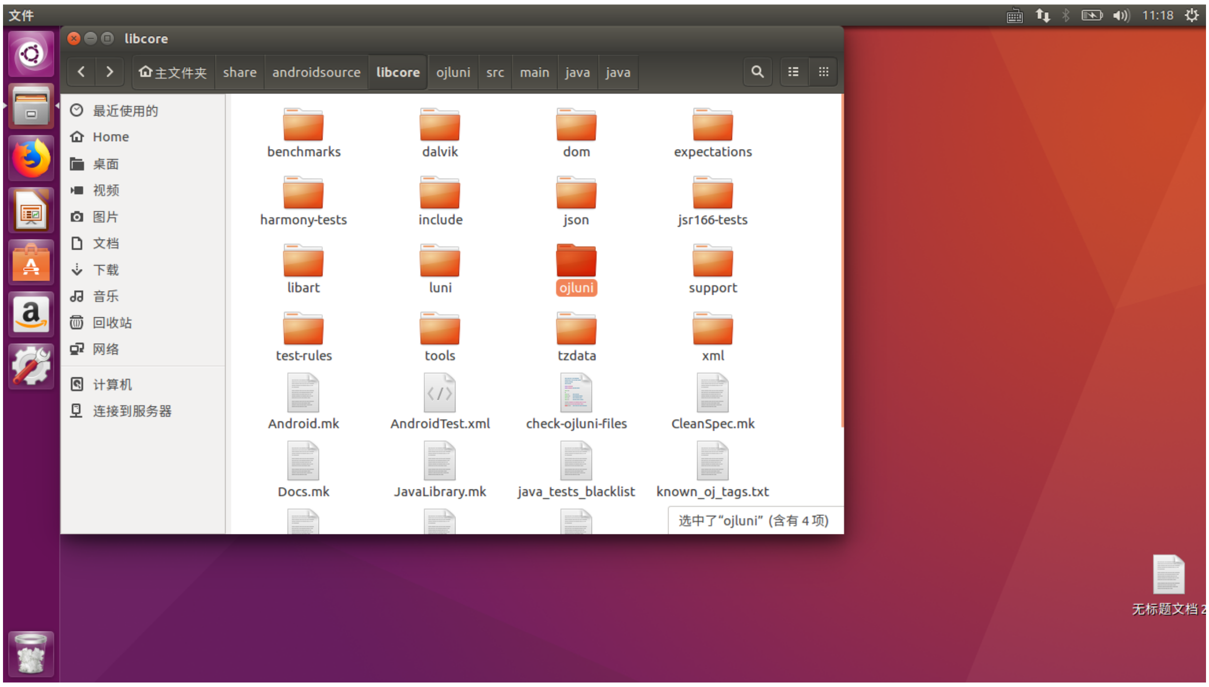
Android8.1 学习笔记--JNI加载SO流程学习+实验数据

表格：

目标平台\编译平台	X86_64	X86	ARM64-V8A
X86_64	√	√ （如需要包含引用32位库编译64位库，需要在引用时加入字段）	✘
X86	✘	√	✘
ARM64-V8A	✘	✘	√
ARMEABI-V7A	✘	✘	✘

Android 是如何加载so的，首先我们得看system.load的源代码。首先得找到Android系统的根目录，找到核心libcore模块，通常在AOSP源码的根目录，libcore文件夹下。在这里，我们可以看到的目录拥有虚拟机，json，dom，ART，expectations等源代码。



这里应该是Android比较核心的模块了，然后我们进入这个目录./libcore/ojluni/src/main/java/java/lang/System.java。这里是系统System的方法，我们找到load的方法。

```
    /**
     * Loads the native library specified by the filename argument.  The filename
     * argument must be an absolute path name.
     *
     * If the filename argument, when stripped of any platform-specific library
     * prefix, path, and file extension, indicates a library whose name is,
     * for example, L, and a native library called L is statically linked
     * with the VM, then the JNI_OnLoad_L function exported by the library
     * is invoked rather than attempting to load a dynamic library.
     * A filename matching the argument does not have to exist in the
     * file system.
     * See the JNI Specification for more details.
     *
     * Otherwise, the filename argument is mapped to a native library image in
     * an implementation-dependent manner.
     *
     * <p>
     * The call <code>System.load(name)</code> is effectively equivalent
     * to the call:
     * <blockquote><pre>
     * Runtime.getRuntime().load(name)
     * </pre></blockquote>
     *
     * @param      filename   the file to load.
     * @exception  SecurityException  if a security manager exists and its
     *             <code>checkLink</code> method doesn't allow
     *             loading of the specified dynamic library
     * @exception  UnsatisfiedLinkError  if either the filename is not an
     *             absolute path name, the native library is not statically
     *             linked with the VM, or the library cannot be mapped to
     *             a native library image by the host system.
     * @exception  NullPointerException if <code>filename</code> is
     *             <code>null</code>
     * @see        java.lang.Runtime#load(java.lang.String)
     * @see        java.lang.SecurityManager#checkLink(java.lang.String)
     */
    @CallerSensitive
    public static void load(String filename) {
        Runtime.getRuntime().load0(VMStack.getStackClass1(), filename);
    }

    /**
     * Loads the native library specified by the <code>libname</code>
     * argument.  The <code>libname</code> argument must not contain any platform
     * specific prefix, file extension or path. If a native library
     * called <code>libname</code> is statically linked with the VM, then the
     * JNI_OnLoad_<code>libname</code> function exported by the library is invoked.
     * See the JNI Specification for more details.
     *
     * Otherwise, the libname argument is loaded from a system library
     * location and mapped to a native library image in an implementation-
     * dependent manner.
     * <p>
     * The call <code>System.loadLibrary(name)</code> is effectively
     * equivalent to the call
     * <blockquote><pre>
     * Runtime.getRuntime().loadLibrary(name)
     * </pre></blockquote>
     *
     * @param      libname   the name of the library.
     * @exception  SecurityException  if a security manager exists and its
     *             <code>checkLink</code> method doesn't allow
     *             loading of the specified dynamic library
     * @exception  UnsatisfiedLinkError if either the libname argument
     *             contains a file path, the native library is not statically
     *             linked with the VM,  or the library cannot be mapped to a
     *             native library image by the host system.
     * @exception  NullPointerException if <code>libname</code> is
     *             <code>null</code>
     * @see        java.lang.Runtime#loadLibrary(java.lang.String)
     * @see        java.lang.SecurityManager#checkLink(java.lang.String)
     */
    @CallerSensitive
    public static void loadLibrary(String libname) {
        Runtime.getRuntime().loadLibrary0(VMStack.getCallingClassLoader(), libname);
    }
```

这里我们看到，systemLoadLibrary实际上调用的是Runtime的loadLibrary0

这个方法，传入的参数是VM的classLoader和对应的mode 名字。在同级目录下，我们去找Runtime.java这个class，然后搜索loadLibrary0这个方法，会发现有3个地方描述到。

```
  //出现地方1描述
/**
 * Loads the native library specified by the <code>libname</code>
 * argument.  The <code>libname</code> argument must not contain any platform
 * specific prefix, file extension or path. If a native library
 * called <code>libname</code> is statically linked with the VM, then the
 * JNI_OnLoad_<code>libname</code> function exported by the library is invoked.
 * See the JNI Specification for more details.
 *
 * Otherwise, the libname argument is loaded from a system library
 * location and mapped to a native library image in an implementation-
 * dependent manner.
 * <p>
 * First, if there is a security manager, its <code>checkLink</code>
 * method is called with the <code>libname</code> as its argument.
 * This may result in a security exception.
 * <p>
 * The method {@link System#loadLibrary(String)} is the conventional
 * and convenient means of invoking this method. If native
 * methods are to be used in the implementation of a class, a standard
 * strategy is to put the native code in a library file (call it
 * <code>LibFile</code>) and then to put a static initializer:
 * <blockquote><pre>
 * static { System.loadLibrary("LibFile"); }
 * </pre></blockquote>
 * within the class declaration. When the class is loaded and
 * initialized, the necessary native code implementation for the native
 * methods will then be loaded as well.
 * <p>
 * If this method is called more than once with the same library
 * name, the second and subsequent calls are ignored.
 *
 * @param      libname   the name of the library.
 * @exception  SecurityException  if a security manager exists and its
 *             <code>checkLink</code> method doesn't allow
 *             loading of the specified dynamic library
 * @exception  UnsatisfiedLinkError if either the libname argument
 *             contains a file path, the native library is not statically
 *             linked with the VM,  or the library cannot be mapped to a
 *             native library image by the host system.
 * @exception  NullPointerException if <code>libname</code> is
 *             <code>null</code>
 * @see        java.lang.SecurityException
 * @see        java.lang.SecurityManager#checkLink(java.lang.String)
 */
@CallerSensitive
```

```
    public void loadLibrary(String libname) {
        loadLibrary0(VMStack.getCallingClassLoader(), libname);
    }

// 出现地方2 调用入口
  /**
    * Temporarily preserved for backward compatibility. Applications call this
    * method using reflection.
    *
    * **** THIS METHOD WILL BE REMOVED IN A FUTURE ANDROID VERSION ****
    *
    * http://b/26217329
    *
    * @hide
    */
    public void loadLibrary(String libname, ClassLoader classLoader) {
        checkTargetSdkVersionForLoad("java.lang.Runtime#loadLibrary(String, ClassLoader)");
        java.lang.System.logE("java.lang.Runtime#loadLibrary(String, ClassLoader)" +
                              " is private and will be removed in a future Android release");
        loadLibrary0(classLoader, libname);
    }

// 出现地方3 实际实现部分元am
synchronized void loadLibrary0(ClassLoader loader, String libname) {
      //  出错点1  判断传入参数是否合法
        if (libname.indexOf((int)File.separatorChar) != -1) {
            throw new UnsatisfiedLinkError(
    "Directory separator should not appear in library name: " + libname);
        }
        String libraryName = libname;
        if (loader != null) {
            //调用classloader去加载对应的lib,如果找不到就报错link
            String filename = loader.findLibrary(libraryName);
            if (filename == null) {
                // 备注提示我们加载的方式规则。
                // It's not necessarily true that the ClassLoader used
                // System.mapLibraryName, but the default setup does, and it's
                // misleading to say we didn't find "libMyLibrary.so" when we
                // actually searched for "liblibMyLibrary.so.so".
                throw new UnsatisfiedLinkError(loader + " couldn't find \"" +
                                               System.mapLibraryName(libraryName) + "\"");
            }
            String error = doLoad(filename, loader);
            if (error != null) {
                throw new UnsatisfiedLinkError(error);
            }
            return;
        }
        // 当classLoader为空的时候的做法，作为APP是不可能为空的，但是系统应用也可能为空。
        String filename = System.mapLibraryName(libraryName);
        List<String> candidates = new ArrayList<String>();
    String lastError = null;
    for (String directory : getLibPaths()) {
      String candidate = directory + filename;
      candidates.add(candidate);

      if (IoUtils.canOpenReadOnly(candidate)) {
        String error = doLoad(candidate, loader);
        if (error == null) {
          return; // We successfully loaded the library. Job done.
        }
        lastError = error;
      }
    }

    if (lastError != null) {
      throw new UnsatisfiedLinkError(lastError);
    }
    throw new UnsatisfiedLinkError("Library " + libraryName + " not found; tried " + candidates);
  }
```

看了上面的描述，我们知道了，加载Library的规则、调用流程这两件事。然后我们再看看doLoad方法到底做了那些事。

```
private String doLoad(String name, ClassLoader loader) {
        // Android apps are forked from the zygote, so they can't have a
custom LD_LIBRARY_PATH,
        // which means that by default an app's shared library directory
isn't on LD_LIBRARY_PATH.

        // The PathClassLoader set up by frameworks/base knows the
appropriate path, so we can load
        // libraries with no dependencies just fine, but an app that has
multiple libraries that
        // depend on each other needed to load them in most-dependent-first
order.

        // We added API to Android's dynamic linker so we can update the
library path used for
        // the currently-running process. We pull the desired path out of
the ClassLoader here
        // and pass it to nativeLoad so that it can call the private
dynamic linker API.

        // We didn't just change frameworks/base to update the
LD_LIBRARY_PATH once at the
        // beginning because multiple apks can run in the same process and
third party code can
        // use its own BaseDexClassLoader.

        // We didn't just add a dlopen_with_custom_LD_LIBRARY_PATH call
because we wanted any
```

```
        // dlopen(3) calls made from a .so's JNI_OnLoad to work too.

        // So, find out what the native library search path is for the
ClassLoader in question...
        String librarySearchPath = null;
        if (loader != null && loader instanceof BaseDexClassLoader) {
            BaseDexClassLoader dexClassLoader = (BaseDexClassLoader)
loader;
            librarySearchPath = dexClassLoader.getLdLibraryPath();
        }
        // nativeLoad should be synchronized so there's only one
LD_LIBRARY_PATH in use regardless
        // of how many ClassLoaders are in the system, but dalvik doesn't
support synchronized
        // internal natives.
        synchronized (this) {
            return nativeLoad(name, loader, librarySearchPath);
        }
    }
```

我们可以看到，上层runtime.java做的事情是调用了native方法去load对应的so。也就是说，这一部分代码在对应的runtime.c中。

```
//JNI入口
JNIEXPORT jstring JNICALL
Runtime_nativeLoad(JNIEnv* env, jclass ignored, jstring javaFilename,
                   jobject javaLoader, jstring javaLibrarySearchPath)
{
    return JVM_NativeLoad(env, javaFilename, javaLoader,
javaLibrarySearchPath);
}
```

这一部分代码调用了JVM的方法，JVM_NativeLoad方法。这个方法不再libcore当中，而是存在于AIT openJDKjvm当中。

```
//jvm入口处
JNIEXPORT jstring JVM_NativeLoad(JNIEnv* env,
                                 jstring javaFilename,
                                 jobject javaLoader,
                                 jstring javaLibrarySearchPath) {
  ScopedUtfChars filename(env, javaFilename);
  if (filename.c_str() == NULL) {
    return NULL;
  }

  std::string error_msg;
  {
    art::JavaVMExt* vm = art::Runtime::Current()->GetJavaVM();
    bool success = vm->LoadNativeLibrary(env,
                                         filename.c_str(),
                                         javaLoader,
                                         javaLibrarySearchPath,
                                        &error_msg);
    if (success) {
      return nullptr;
    }
  }

  // Don't let a pending exception from JNI_OnLoad cause a CheckJNI issue
with NewStringUTF.
  env->ExceptionClear();
  return env->NewStringUTF(error_msg.c_str());
}
```

追述进去，就是在Java_vm_ext.cc当中的实现代码。

```


  //在我们要加载so库中查找JNI_OnLoad方法，
  //如果没有系统就认为是静态注册方式进行的，直接返回true，代表so库加载成功，
  //如果找到JNI_OnLoad就会调用JNI_OnLoad方法，JNI_OnLoad方法中一般存放的是方法注册的
函数，
  //所以如果采用动态注册就必须要实现JNI_OnLoad方法，否则调用java中申明的native方法时会
抛出异常，
  //下面有JNI_OnLoad的实现

  // Create a new entry.
  // TODO: move the locking (and more of this logic) into Libraries.
  bool created_library = false;
  {
    // Create SharedLibrary ahead of taking the libraries lock to maintain
lock ordering.
    std::unique_ptr<SharedLibrary> new_library(
        new SharedLibrary(env,
                          self,
                          path,
                          handle,
                          needs_native_bridge,
                          class_loader,
                          class_loader_allocator));

    MutexLock mu(self, *Locks::jni_libraries_lock_);
    library = libraries_->Get(path);
    if (library == nullptr) {  // We won race to get libraries_lock.
      library = new_library.release();
      libraries_->Put(path, library);
      created_library = true;
    }
  }
  if (!created_library) {
    LOG(INFO) << "WOW: we lost a race to add shared library: "
        << "\"" << path << "\" ClassLoader=" << class_loader;
    return library->CheckOnLoadResult();
  }
  VLOG(jni) << "[Added shared library \"" << path << "\" for ClassLoader "
<< class_loader << "]";
  //系统加载，加载系统库
  bool was_successful = false;
  void* sym = library->FindSymbol("JNI_OnLoad", nullptr);
  if (sym == nullptr) {
    VLOG(ani) << "[No JNI_OnLoad found in \"" << path << "\"]";
    was_successful = true;
  } else {
    // Call JNI_OnLoad.  We have to override the current class
    // loader, which will always be "null" since the stuff at the
    // top of the stack is around Runtime.loadLibrary().  (See
    // the comments in the JNI FindClass function.)
    ScopedLocalRef<jobject> old_class_loader(env, env->NewLocalRef(self-
>GetClassLoaderOverride()));
    self->SetClassLoaderOverride(class_loader);

    VLOG(jni) << "[Calling JNI_OnLoad in \"" << path << "\"]";
    typedef int (*JNI_OnLoadFn)(JavaVM*, void*);
    JNI_OnLoadFn jni_on_load = reinterpret_cast<JNI_OnLoadFn>(sym);
    int version = (*jni_on_load)(this, nullptr);

    if (runtime_->GetTargetSdkVersion() != 0 && runtime_-
>GetTargetSdkVersion() <= 21) {
      // Make sure that sigchain owns SIGSEGV.
      EnsureFrontOfChain(SIGSEGV);
    }

    self->SetClassLoaderOverride(old_class_loader.get());

    if (version == JNI_ERR) {
      StringAppendF(error_msg, "JNI_ERR returned from JNI_OnLoad in
\"%s\"", path.c_str());
    } else if (JavaVMExt::IsBadJniVersion(version)) {
      StringAppendF(error_msg, "Bad JNI version returned from JNI_OnLoad in
\"%s\": %d",
                    path.c_str(), version);
      // It's unwise to call dlclose() here, but we can mark it
      // as bad and ensure that future load attempts will fail.
      // We don't know how far JNI_OnLoad got, so there could
      // be some partially-initialized stuff accessible through
      // newly-registered native method calls.  We could try to
      // unregister them, but that doesn't seem worthwhile.
    } else {
      was_successful = true;
    }
    VLOG(jni) << "[Returned " << (was_successful ? "successfully" :
"failure")
              << " from JNI_OnLoad in \"" << path << "\"]";
  }
  library->SetResult(was_successful);
  return was_successful;
}
```

就优先级而言，loadNativeLibrary可以加载一次so，多次会出现warming。还有一个是关于shareLibrary的声明。

```
class SharedLibrary {
public:
  SharedLibrary(JNIEnv* env, Thread* self, const std::string& path, void*
handle,
                bool needs_native_bridge, jobject class_loader, void*
class_loader_allocator)
      : path_(path),
        handle_(handle),
        needs_native_bridge_(needs_native_bridge),
        class_loader_(env->NewWeakGlobalRef(class_loader)),
        class_loader_allocator_(class_loader_allocator),
        jni_on_load_lock_("JNI_OnLoad lock"),
        jni_on_load_cond_("JNI_OnLoad condition variable",
jni_on_load_lock_),
        jni_on_load_thread_id_(self->GetThreadId()),
        jni_on_load_result_(kPending) {
    CHECK(class_loader_allocator_ != nullptr);
  }

  ~SharedLibrary() {
    Thread* self = Thread::Current();
    if (self != nullptr) {
      self->GetJniEnv()->DeleteWeakGlobalRef(class_loader_);
    }

    android::CloseNativeLibrary(handle_, needs_native_bridge_);
  }
```

那么加载64位机器的so,默认有几个文件夹呢》？

我们改下路径，假定原先预加载的是libnative-lib.so，我们人为在代码中改变路径为native-libs。这样，APP就会因为找不到路径而报错。报错的异常会打印指定的目录：

```
32 arm
[/data/app/com.genesis.myjnidemo-1/lib, /system/vendor/lib,/system/lib]]

64 arm 64
[/data/app/com.genesis.myjnidemo-1/lib/arm64, /vendor/lib64,
/system/lib64]]] couldn't find "libnative-libs.so"
```

按照上述标红的代码段，我们可以得到的结论，默认优先级是本地、system/lib、然后system/vendor/lib。当然这不一定是这样的一个情况，但是有一点可以明确，

data/app/packagename/lib一定会优先去寻找。那么map到底传了什么东西过来？我们得看system.cc文件源码：

```
JNIEXPORT jstring JNICALL
System_mapLibraryName(JNIEnv *env, jclass ign, jstring libname)
{
    int len;
    int prefix_len = (int) strlen(JNI_LIB_PREFIX);
    int suffix_len = (int) strlen(JNI_LIB_SUFFIX);

    jchar chars[256];
    if (libname == NULL) {
        JNU_ThrowNullPointerException(env, 0);
        return NULL;
    }
    len = (*env)->GetStringLength(env, libname);
    if (len > 240) {
        JNU_ThrowIllegalArgumentException(env, "name too long");
        return NULL;
    }
    cpchars(chars, JNI_LIB_PREFIX, prefix_len);
    (*env)->GetStringRegion(env, libname, 0, len, chars + prefix_len);
    len += prefix_len;
    cpchars(chars + len, JNI_LIB_SUFFIX, suffix_len);
    len += suffix_len;

    return (*env)->NewString(env, chars, len);
}
```

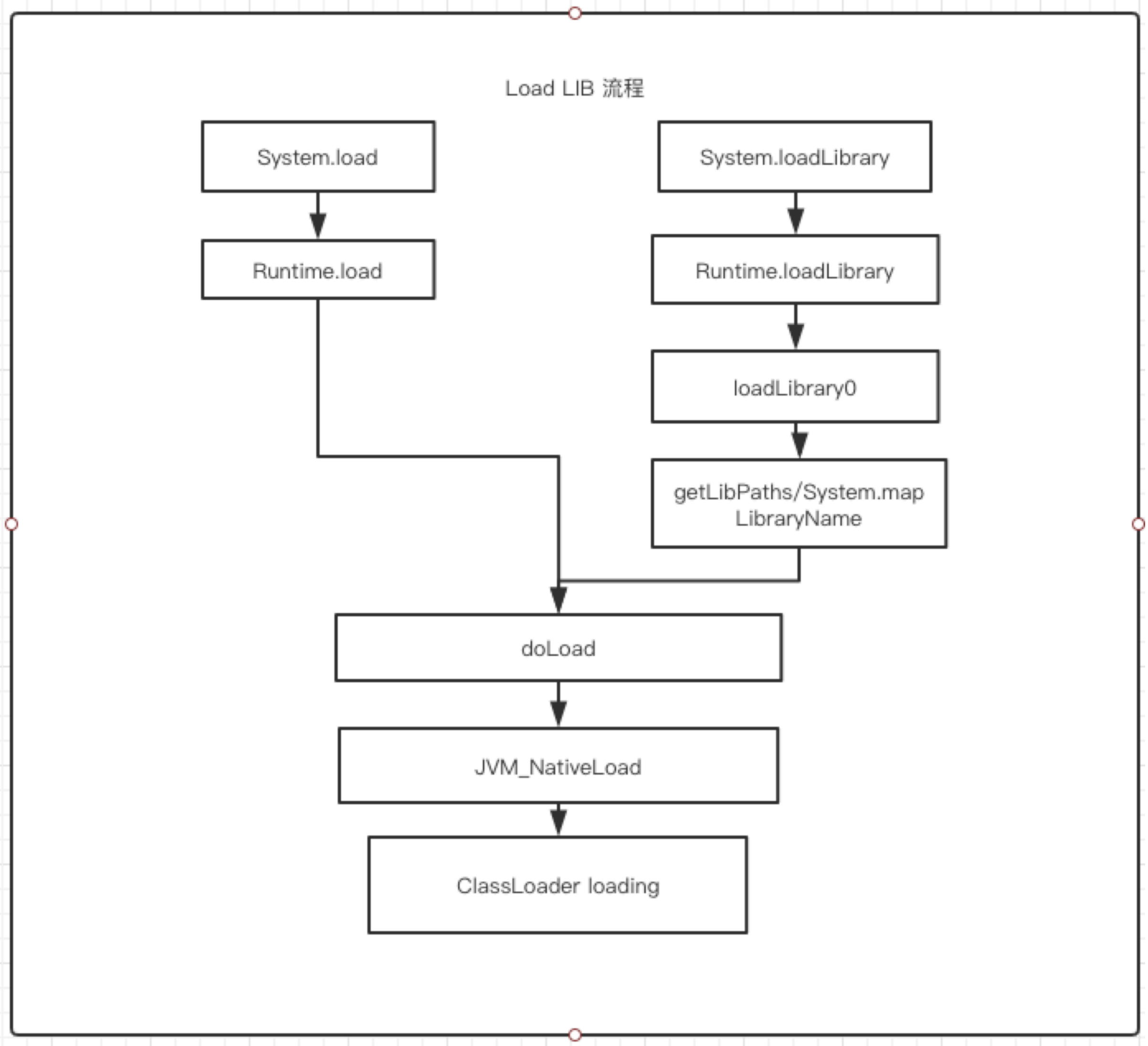
这里我们了解到的是把对应的lib组合规范。比如native-lib，会加之libnative-lib.so这样的形式输出返回。下一个，取路径方法，getLibPaths这个方法又做了那些事。我们在看到RunTime部分代码。

```
  private String[] getLibPaths() {
      if (mLibPaths == null) {
          synchronized(this) {
              if (mLibPaths == null) {
                  mLibPaths = initLibPaths();
              }
          }
      }
      return mLibPaths;
   }

   private static String[] initLibPaths() {
      String javaLibraryPath = System.getProperty("java.library.path");
      if (javaLibraryPath == null) {
          return EmptyArray.STRING;
      }
      String[] paths = javaLibraryPath.split(":");
```

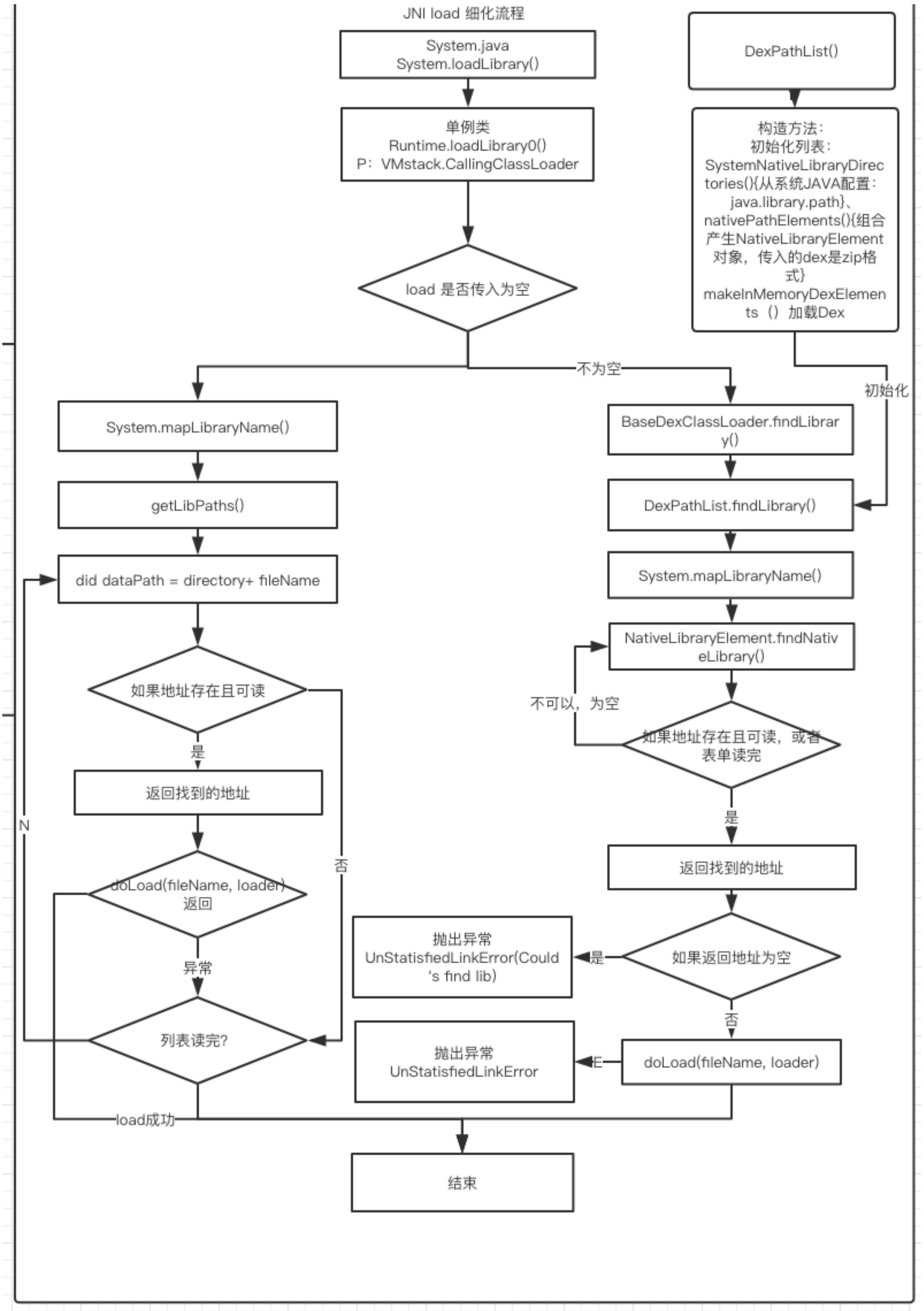
```
    // Add a '/' to the end of each directory so we don't have to do it every time.
    for (int i = 0; i < paths.length; ++i) {
        if (!paths[i].endsWith("/")) {
            paths[i] += "/";
        }
    }
    return paths;
}
```

相信看到这里，我们就明白了，Android系统是如何去执行的找so的部分咯。流程：



- 深入ClassLoader代码分析

这里我们找到同级目录的ClassLoader方法


JNI load 细化流程
System.java
System.loadLibrary()
单例类
Runtime.loadLibrary0()
P：VMstack.CallingClassLoader
load 是否传入为空
不为空
System.mapLibraryName()
getLibPaths()
did dataPath = directory+ fileName
如果地址存在且可读
是
返回找到的地址
doLoad(fileName, loader)
返回
异常
列表读完?
N
否
load成功
DexPathList()
构造方法：
初始化列表：
SystemNativeLibraryDirectories(){从系统JAVA配置：java.library.path}、
nativePathElements(){组合产生NativeLibraryElement对象，传入的dex是zip格式}
makeInMemoryDexElements（）加载Dex
初始化
BaseDexClassLoader.findLibrary()
DexPathList.findLibrary()
System.mapLibraryName()
NativeLibraryElement.findNativeLibrary()
不可以，为空
如果地址存在且可读，或者表单读完
是
返回找到的地址
如果返回地址为空
是
抛出异常
UnStatisfiedLinkError(Could's find lib)
否
doLoad(fileName, loader)
E
抛出异常
UnStatisfiedLinkError
结束

还有一个事情，就是系统应用，如果默认64位机器，如果so是32位，加载错误可能会报错。

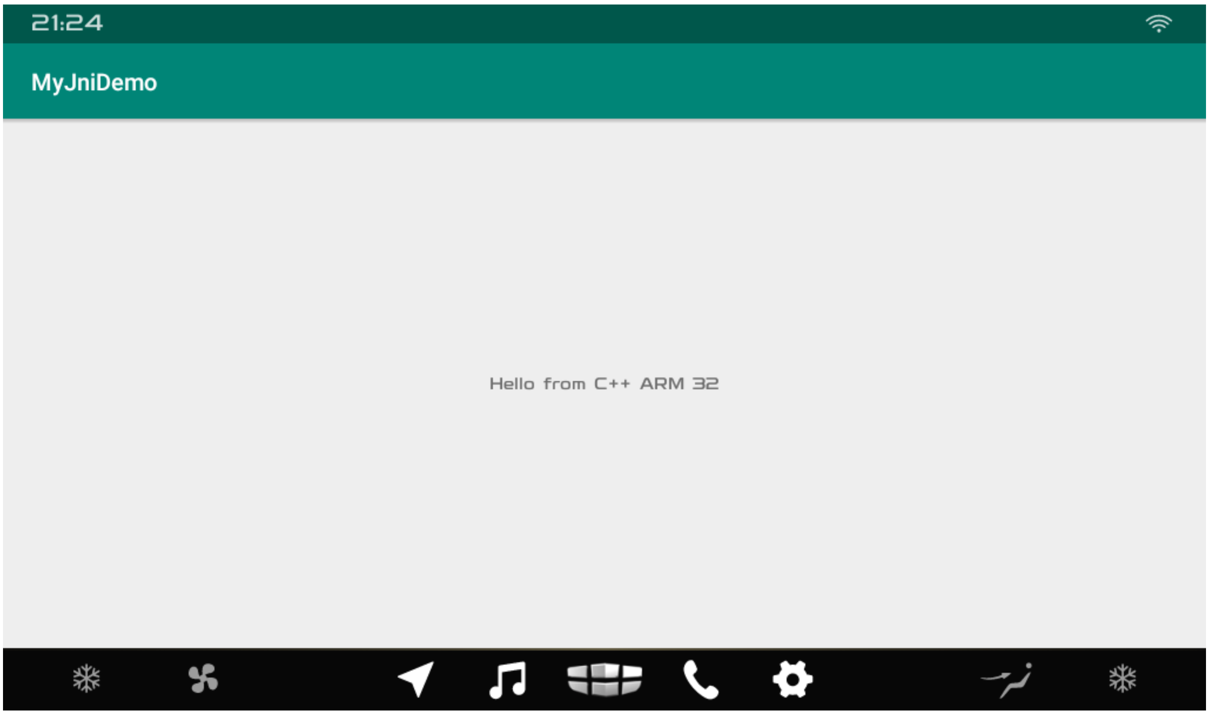
```
01-29 20:28:48.711 3640-3640/com.genesis.myjnidemo E/art:
dlopen("/system/lib/libnative-libs.so", RTLD_LAZY) failed: dlopen failed:
"/system/lib/libnative-libs.so" is 32-bit instead of 64-bit
01-29 20:28:48.714 3640-3640/com.genesis.myjnidemo E/AndroidRuntime: FATAL
EXCEPTION: main
    Process: com.genesis.myjnidemo, PID: 3640
    java.lang.UnsatisfiedLinkError: dlopen failed: "/system/lib/libnative-
libs.so" is 32-bit instead of 64-bit
        at java.lang.Runtime.load(Runtime.java:332)
        at java.lang.System.load(System.java:981)
        at com.genesis.myjnidemo.MainActivity.<clinit>
(MainActivity.java:11)
        at java.lang.reflect.Constructor.newInstance(Native Method)
        at java.lang.Class.newInstance(Class.java:1606)
        at
android.app.Instrumentation.newActivity(Instrumentation.java:1071)
        at
android.app.ActivityThread.performLaunchActivity(ActivityThread.java:2455)
        at
android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:2653)
        at android.app.ActivityThread.access$900(ActivityThread.java:190)
        at
android.app.ActivityThread$H.handleMessage(ActivityThread.java:1488)
        at android.os.Handler.dispatchMessage(Handler.java:111)
        at android.os.Looper.loop(Looper.java:194)
        at android.app.ActivityThread.main(ActivityThread.java:5682)
        at java.lang.reflect.Method.invoke(Native Method)
        at java.lang.reflect.Method.invoke(Method.java:372)
        at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:
982)
        at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:777)
```

如果考虑兼容32位so.apk运行以32位运行，可以参考下步：

编写 helloJNI.apk, 并且解压缩so.push到sdcard目录。

```
  ~ adb push helloJNI.apk  /sdcard/helloJNI.apk
  ~ adb push libnative-libs.so /sdcard/libnative-libs.so
    Adb shell
shell@xe3b0fl:/ #  cp /sdcard/helloJNI.apk /system/app
shell@xe3b0fl:/ #  mkdir system/lib/helloJNI
shell@xe3b0fl:/ #  cp /sdcard/libnative-libs.so /system/lib
shell@xe3b0fl:/ #  chmod 777  system/lib/helloJNI/libnative-libs.so
shell@xe3b0fl:/ #  chmod 777  system/app/helloJNI/helloJNI.apk
shell@xe3b0fl:/ #  reboot
```

最后结果



Hello from C++ ARM 32

By Genesis.Ling
2019.01.29