# TODO

Android类加载器ClassLoader

http://gityuan.com/2017/03/19/android-classloader/

Android JNI原理分析

<http://gityuan.com/2016/05/28/android-jni/>

debuggerd守护进程

<http://gityuan.com/2016/06/15/android-debuggerd/>

-rw-r----- 1 system u0\_a31000 16384 2018-06-15 05:00 system@priv-app@kpad-ui@kpad-ui.apk@classes.art

-rw-r--r-- 1 system u0\_a31000 13177472 2018-06-15 18:57 [system@priv-app@kpad-ui@kpad-ui.apk@classes.dex](mailto:system@priv-app@kpad-ui@kpad-ui.apk@classes.dex)

<https://blog.csdn.net/austindev/article/details/61194249>

<https://blog.csdn.net/myfriend0/article/details/54970455>

预加载类

MultiDex

<https://blog.csdn.net/DJY1992/article/details/51162013?utm_source=blogxgwz11>

<https://www.xuebuyuan.com/3184083.html>

<https://blog.csdn.net/threepigs/article/details/52884904>

<https://www.cnblogs.com/manuosex/p/3634375.html>

https://www.jianshu.com/p/61ff20b53e40

https://blog.csdn.net/threepigs/article/details/52724748

<https://www.jianshu.com/p/1f779586efdc>

https://blog.csdn.net/woai110120130/article/details/54706473

<https://blog.csdn.net/lin20044140410/article/details/79307964>

*然后每次开机为何都那么慢呢？*

*06-15 10:03:30.270 3271-3271/? E/dex2oat: Could not create image space with image file '/system/framework/boot.art'. Attempting to fall back to imageless running. Error was: Only the zygote can create the global boot image.*

*Attempted image: /system/framework/boot.art*

*06-15 10:03:30.271 3271-3271/? E/dex2oat: Dex file fallback disabled, cannot continue without image.*

*Failed to create runtime*

*06-15 10:03:30.290 3273-3273/? E/SchedPolicy: open of /dev/blkio/tasks failed: Permission denied*

*open of /dev/blkio/bg\_non\_interactive/tasks failed: Permission denied*

# 基本概念

虚拟机的gc和内核回收差异：无关，各自不同的机制

## Dalvik和ART

Android系统是以Linux为内核构建的。Google为了降低应用的开发难度，并将其适配到不同硬件配置的设备上，在Linux内核之上构建了一个虚拟机，Android应用使用java开发，运行在虚拟机之上。

Dalvik就是Android4.4及之前使用的虚拟机，它使用的是JIT（Just-In-Time）技术来进行代码转译，每次执行应用的时候，Dalvik将程序的代码编译为机器语言执行。随着硬件水平的不断发展以及人们对更高性能的需求，Dalvik虚拟机的不足日益突出。而应运而生的ART(Android RunTime)虚拟机，其处理机制根本上的区别是：它采用AOT(Ahead-Of-Time)技术，会在应用程序安装时就转换成机器语言，不再在执行时解释，从而优化了应用运行的速度。在内存管理方面，ART也有比较大的改进，对内存分配和回收都做了算法优化，降低了内存碎片化程度，回收时间也得以缩短。使用AOT进行处理（Ahead-Of-Time）

先了解Java的几种虚拟机基本的工作机制：

（1）JVM：JVM虚拟机运行的是java字节码。Java文件到JVM的过程是：java -> java bytecode(class) -> java bytecode(jar)。

（2）DVM：DVM虚拟机解析执行的dex字节码。Java文件到DVM的过程是：java -> java bytecode(class) -> dalvik bytecode(dex)。

（3）ART：ART虚拟机执行本地机器码。Java文件到ART的过程是：java -> java bytecode(class) -> dalvik bytecode(dex) -> optimized android runtime machine code(oat)。

从java文件到虚拟机执行代码，ART比DVM多了oat的过程，ART所使用的AOT（Ahead-Of-Time）编译，在应用首次安装时，字节码预编译成机器码存在本地，而DVM是典型的JIT（Just-In-Time），在此模式下，应用每次运行的时候，字节码都需要即时编译器转换为机器码再执行，ART模式相对于DVM省去了解析字节码的过程，占用内存也相应减少，进而提高APP的运行效率，所以Andorid 5.0发布时，Google号称Android系统可以跑在512 M的内存的机器上，ART发挥了举足轻重的作用。

## dex

本质上java文件编译后都是字节码ByteCode，不管是传统的JVM，还是Google Dalvik DVM。只是这两种虚拟机环境下ByteCode有所差异，最直观的是在JVM运行的是.class文件，而DVM是.dex文件，DVM专门对移动操作系统(尤其是Android)的特性进行了优化，并且DVM的设计是基于寄存器的，指令集有非常大的不同(具体未研究),还有等等等等...,好吧，到这步，了解到.dex是字节码，至于dex文件format构成，脑补阶段暂时略过吧XD~~。

## Odex

讲了dex、JIT,接下来讲讲Odex，Odex即Optimize Dex对dex文件的优化，最直观的好处：

* deodex在系统第一次开机时，需要提取所有apk里的dex文件，而odex优化是提前提取出来了，开机速度&运行速度都有提高。
* Odex优化后，APK里可以没有dex文件，而未Odex在APK包里有一份dex文件，在/data/dalvik-cache下还有提取出来的一份，浪费存储空间。
* 一定程度上保护了厂商自己的APK，因为apk里只有资源文件，反汇编没有意义，直接拷贝到别处无法安装运行...。

## AOT与ART

技术随着时间之轮毫不停歇前行。Android kitkat 4.4,新的Android Runtime(ART)出现了，做为一个可选项，供一些愿意的用户测试使用，当然这个时候Dalvik还是作为默认的虚拟机环境。ART采取了AOT(Ahead-Of-Time)技术，简单一点理解就是，在APK安装的时候就会做预先编译动作，编译好的文件是OAT文件，该文件本质上是一个ELF文件，这里与dex(Odex)文件最大的区别是OAT文件不再是字节码文件，而是一个可执行文件，可以更底层的与硬件接触，运行时也省去了预编译和转译的时间。

在Android L中我们找不到OAT文件，其实oat文件依旧以.odex作为后缀，通过file命令或者UE打开可以看到ELF头部。另外ART设计是考虑兼容性的，即在Dalvik可以运行的APP，在系统升级到5.0、5.1(Dalvik->ART)这些APP依旧可以运行，这个是通过dex2oat做到的，dalvik下的dex、odex文件均可以通过这个工具转化为oat文件，并且odex文件将比dex文件编译的更快。

## 参考

<https://www.jianshu.com/p/ac079e7fc412>

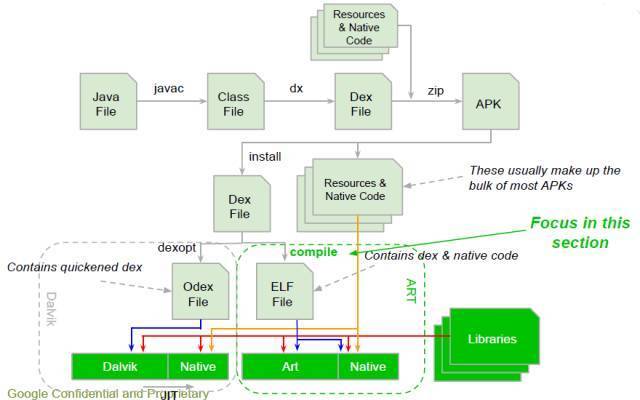
## Android 系统启动原理

Android 系统启动原理（art 虚拟机）

<https://www.2cto.com/kf/201805/743995.html>

https://blog.csdn.net/threepigs/article/details/52724748

## Android对apk的执行流程



Java文件在编译成class文件，然后经过Android平台的dx工具转换为Dex文件后，同Native code（JNI）和资源一起打包成apk，apk安装到手机后解压出Dex文件。Dalvik会通过dexopt工具将Dex优化，成为Odex文件，Odex文件的效率比Dex高，但其中大部分代码仍然需要每次执行时编译；而ART则会将Dex通过dex2oat工具编译得到一个ELF文件，它是一个可执行的文件。

关于Java代码的执行过程，以一段简单的代码为例：

int a = 1;

int b = 2;

public int test() {

① int x = a;

② int y = b;

int z = a + b;

return z;

}

在执行这段Java代码时，Dalvik虚拟机先要把test()方法的每句代码转译成Dex代码，对其中的① ② 两句赋值语句，执行时需要在虚拟机中进行“指令读取—识别指令—跳转—实例操作”的解析过程；而ART中Java代码都被以方法为单位编译成汇编指令，执行上面这个方法的时候，① ② 两句代码只需要直接拷贝两个寄存器的值，各需要一条汇编指令就可以完成，省去了跳转、指令读取的过程，执行效率也就大大提高了。

虚拟机从Dalvik换成ART后，Android系统的性能得到了一定程度的提升。不过ART与Dalvik相比也存在一些缺点，比较明显的表现就是，apk经过dex2oat预编译之后，占用的空间增加，因此Android ROM占用的空间更大。手机在安装下载的apk时，安装时间也明显变长。

# loadLibrary动态库加载过程分析出厂设置

libcore/luni/src/main/java/java/lang/System.java

libcore/luni/src/main/java/java/lang/Runtime.java

libcore/luni/src/main/native/java\_lang\_System.cpp

bionic/linker/linker.cpp

art/runtime/native/java\_lang\_Runtime.cc

art/runtime/java\_vm\_ext.cc

## 一. 概述

动态库操作,所需要的头文件的#include, 最为核心的方法如下:

void \*dlopen(const char \* pathname,int mode); //打开动态库

void \*dlsym(void \*handle,const char \*name); //获取动态库对象地址

char \*dlerror(vid); //错误检测

int dlclose(void \* handle); //关闭动态库

而对于android上层的Java代码来说,都封装好了, 只需要一行代码就即可完成动态库的加载过程,如下:

System.loadLibrary("gityuan\_jni");

接下来,解析这行代码背后的故事.

## 动态库加载过程

### 2.1 System.loadLibrary

[-> System.java]

@CallerSensitive  
**public void** loadLibrary(String libname) {  
 loadLibrary0(VMStack.*getCallingClassLoader*(), libname);  
}

此处getCallingClassLoader返回的是调用者所定义的ClassLoader.

### 2.2 Runtime.loadLibrary

[-> Runtime.java]

**synchronized void** loadLibrary0(ClassLoader loader, String libname) {  
 **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**) {

//[见小节2.4.1]  
 String filename = loader.findLibrary(libraryName);  
 **if** (filename == **null**) {  
 *// 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) + **"\""**);  
 }

////成功执行完doLoad,则返回.[见小节2.5]  
 String error = doLoad(filename, loader);  
 **if** (error != **null**) {  
 **throw new** UnsatisfiedLinkError(error);  
 }  
 **return**;  
 }  
//当loader为空的情况下执行[见2.3]  
 String filename = System.*mapLibraryName*(libraryName);  
 List<String> candidates = **new** ArrayList<String>();  
 String lastError = **null**;

//此处的mLibPaths取值 [见小节三]  
 **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);  
}

加载库的两条路径,如下:

* 当loader不为空时, 通过该loader来findLibrary()查看目标库所在路径;
* 当loader为空时, 则从默认目录mLibPaths下来查找是否存在该动态库;

不管loader是否为空, 找到目标库所在路径后,都会调用doLoad来真正用于加载动态库.

### 2.3 loader为空的情况

[-> System.java]

public static String mapLibraryName(String nickname) {

if (nickname == null) {

throw new NullPointerException("nickname == null");

}

return "lib" + nickname + ".so";

}

该方法的功能是将xxx动态库的名字转换为libxxx.so. 比如前面传递过来的nickname为gityuan\_jni, 经过该方法处理后返回的名字为libgityuan\_jni.so.

在8.1又变了，通过jni来加前缀和后缀

**public static native** String mapLibraryName(String libname);

C:\k\android-8.0.0\_r1\libcore\ojluni\src\main\native\System.c

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);  
}

C:\k\android-8.0.0\_r1\libcore\ojluni\src\main\native\jvm\_md.h

#define JNI\_LIB\_PREFIX **"lib"**#ifdef \_\_APPLE\_\_  
#define JNI\_LIB\_SUFFIX **".dylib"**#define VERSIONED\_JNI\_LIB\_NAME(NAME, VERSION) JNI\_LIB\_PREFIX NAME **"."** VERSION JNI\_LIB\_SUFFIX  
#else  
#define JNI\_LIB\_SUFFIX **".so"**

#### LibPaths

mLibPaths取值分两种情况:

* 对于64系统,则为/system/lib64和/vendor/lib64;
* 对于32系统,则为/system/lib和/vendor/lib.

假设此处为64位系统, 则会去查找/system/lib64/libgityuan\_jni.so或/vendor/lib64/libgityuan\_jni.so库是否存在.

绝大多数的动态库都在/system/lib64或/system/lib路径下.通过canOpenReadOnly方法来判定目标动态库是否存在, 找到则直接返回,否则抛出UnsatisfiedLinkError异常.

**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;  
}

### loader不为空的情况

ClassLoader一般来说都是PathClassLoader, 这就不再解释. 从该对象的findLibrary说起. 由于PathClassLoader继承于 BaseDexClassLoader对象, 并且没有覆写该方法, 故调用其父类所对应的方法.

#### findLibrary

[-> BaseDexClassLoader.java]

public class BaseDexClassLoader extends ClassLoader {

private final DexPathList pathList;

public BaseDexClassLoader(String dexPath, File optimizedDirectory, String libraryPath, ClassLoader parent) {

super(parent);

//dexPath一般是指apk所在路径【小节2.4.2】

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

}

public String findLibrary(String name) {

return pathList.findLibrary(name); //[见小节2.4.3]

}

}

#### 2.4.2 DexPathList

[-> DexPathList.java]

public DexPathList(ClassLoader definingContext, String dexPath, String libraryPath, File optimizedDirectory) {

...

this.definingContext = definingContext;

ArrayList<IOException> suppressedExceptions = new ArrayList<IOException>();

//记录所有的dexFile文件

this.dexElements = makePathElements(splitDexPath(dexPath), optimizedDirectory, suppressedExceptions);

//app目录的native库

this.nativeLibraryDirectories = splitPaths(libraryPath, false);

//系统目录的native库

this.systemNativeLibraryDirectories = splitPaths(System.getProperty("java.library.path"), true);

List<File> allNativeLibraryDirectories = new ArrayList<>(nativeLibraryDirectories);

allNativeLibraryDirectories.addAll(systemNativeLibraryDirectories);

//记录所有的Native动态库

this.nativeLibraryPathElements = makePathElements(allNativeLibraryDirectories, null,

suppressedExceptions);

...

}

DexPathList初始化过程,主要

DexPathList初始化过程,主要功能是收集以下两个变量信息:

1. dexElements: 记录所有的dexFile文件
2. nativeLibraryPathElements: 记录所有的Native动态库, 包括app目录的native库和系统目录的native库.

接下来便是从pathList中查询目标动态库.

#### findLibrary

**public** String findLibrary(String libraryName) {  
 String fileName = System.*mapLibraryName*(libraryName);  
  
 **for** (NativeLibraryElement element : **nativeLibraryPathElements**) {  
 String path = element.findNativeLibrary(fileName);  
  
 **if** (path != **null**) {  
 **return** path;  
 }  
 }  
  
 **return null**;  
}

gityuan\_jni同样也是经过mapLibraryName(), 处理后得到的名字为libgityuan\_jni.so. 接下来,从所有的动态库nativeLibraryPathElements(包含两个系统路径)查询是否存在匹配的. 这里举例说明, 一般地会在以下两个路径查找(以64位为例):

* /data/app/com.gityuan.blog-1/lib/arm64
* /vendor/lib64
* /system/lib64

#### findNativeLibrary

**public** String findNativeLibrary(String name) {  
 maybeInit();  
  
 **if** (**zipDir** == **null**) {  
 String entryPath = **new** File(**path**, name).getPath();  
 **if** (IoUtils.*canOpenReadOnly*(entryPath)) {  
 **return** entryPath;  
 }  
 } **else if** (**urlHandler** != **null**) {  
 *// Having a urlHandler means the element has a zip file.  
 // In this case Android supports loading the library iff  
 // it is stored in the zip uncompressed.* String entryName = **zipDir** + **'/'** + name;  
 **if** (**urlHandler**.isEntryStored(entryName)) {  
 **return path**.getPath() + ***zipSeparator*** + entryName;  
 }  
 }  
  
 **return null**;  
}

### Runtime.doLoad

[-> Runtime.java]

private String doLoad(String name, ClassLoader loader) {

String ldLibraryPath = null;

String dexPath = null;

if (loader == null) {

ldLibraryPath = System.getProperty("java.library.path");

} else if (loader instanceof BaseDexClassLoader) {

BaseDexClassLoader dexClassLoader = (BaseDexClassLoader) loader;

ldLibraryPath = dexClassLoader.getLdLibraryPath();

}

synchronized (this) {

//[见小节2.5.1]

return nativeLoad(name, loader, ldLibraryPath);

}

}

此处ldLibraryPath有两种情况：

* 当loader为空，　则ldLibraryPath为系统目录下的Ｎative库；
* 当lodder不为空，　则ldLibraryPath为ａpp目录下的Ｎative库；

接下来, 继续看看nativeLoad.

#### 2.5.1 nativeLoad

[-> java\_lang\_Runtime.cc]

static jstring Runtime\_nativeLoad(JNIEnv\* env, jclass, jstring javaFilename, jobject javaLoader,

jstring javaLdLibraryPathJstr) {

ScopedUtfChars filename(env, javaFilename);

if (filename.c\_str() == nullptr) {

return nullptr;

}

SetLdLibraryPath(env, javaLdLibraryPathJstr);

std::string error\_msg;

{

JavaVMExt\* vm = Runtime::Current()->GetJavaVM();

//[见小节2.5.2]

bool success = vm->LoadNativeLibrary(env, filename.c\_str(), javaLoader, &error\_msg);

if (success) {

return nullptr;

}

}

env->ExceptionClear();

return env->NewStringUTF(error\_msg.c\_str());

}

nativeLoad方法经过jni,进入该方法执行.

#### 2.5.2 LoadNativeLibrary

[-> java\_vm\_ext.cc]

bool JavaVMExt::LoadNativeLibrary(JNIEnv\* env, const std::string& path, jobject class\_loader,

std::string\* error\_msg) {

error\_msg->clear();

SharedLibrary\* library;

Thread\* self = Thread::Current();

{

MutexLock mu(self, \*Locks::jni\_libraries\_lock\_);

library = libraries\_->Get(path); //检查该动态库是否已加载

}

if (library != nullptr) {

if (env->IsSameObject(library->GetClassLoader(), class\_loader) == JNI\_FALSE) {

//不能加载同一个采用多个不同的ClassLoader

return false;

}

...

return true;

}

const char\* path\_str = path.empty() ? nullptr : path.c\_str();

//通过dlopen打开动态共享库.该库不会立刻被卸载直到引用技术为空.

void\* handle = dlopen(path\_str, RTLD\_NOW);

bool needs\_native\_bridge = false;

if (handle == nullptr) {

if (android::NativeBridgeIsSupported(path\_str)) {

handle = android::NativeBridgeLoadLibrary(path\_str, RTLD\_NOW);

needs\_native\_bridge = true;

}

}

if (handle == nullptr) {

\*error\_msg = dlerror(); //打开失败

VLOG(jni) << "dlopen(\"" << path << "\", RTLD\_NOW) failed: " << \*error\_msg;

return false;

}

bool created\_library = false;

{

std::unique\_ptr<SharedLibrary> new\_library(

new SharedLibrary(env, self, path, handle, class\_loader));

MutexLock mu(self, \*Locks::jni\_libraries\_lock\_);

library = libraries\_->Get(path);

if (library == nullptr) {

library = new\_library.release();

//创建共享库,并添加到列表

libraries\_->Put(path, library);

created\_library = true;

}

}

...

bool was\_successful = false;

void\* sym;

//查询JNI\_OnLoad符号所对应的方法

if (needs\_native\_bridge) {

library->SetNeedsNativeBridge();

sym = library->FindSymbolWithNativeBridge("JNI\_OnLoad", nullptr);

} else {

sym = dlsym(handle, "JNI\_OnLoad");

}

if (sym == nullptr) {

was\_successful = true;

} else {

//需要先覆盖当前ClassLoader.

ScopedLocalRef<jobject> old\_class\_loader(env, env->NewLocalRef(self->GetClassLoaderOverride()));

self->SetClassLoaderOverride(class\_loader);

typedef int (\*JNI\_OnLoadFn)(JavaVM\*, void\*);

JNI\_OnLoadFn jni\_on\_load = reinterpret\_cast<JNI\_OnLoadFn>(sym);

// 真正调用JNI\_OnLoad()方法的过程

int version = (\*jni\_on\_load)(this, nullptr);

if (runtime\_->GetTargetSdkVersion() != 0 && runtime\_->GetTargetSdkVersion() <= 21) {

fault\_manager.EnsureArtActionInFrontOfSignalChain();

}

//执行完成后, 需要恢复到原来的ClassLoader

self->SetClassLoaderOverride(old\_class\_loader.get());

...

}

library->SetResult(was\_successful);

return was\_successful;

}

该过程简单总结:

1. 检查该动态库是否已加载;
2. 通过dlopen打开动态共享库;
3. 创建SharedLibrary共享库,并添加到libraries\_列表;
4. 找到JNI\_OnLoad符号所对应的方法, 并调用该方法.

## LibPaths初始化

LibPaths的初始化过程, 要从libcore/luni/src/main/java/java/lang/System.java类的静态代码块初始化开始说起. 对于类的静态代码块,编译过程会将所有的静态代码块和静态成员变量的赋值过程都收集整合到clinit方法, 即类的初始化方法.如下:

public final class System {

static {

...

//[见小节3.1]

unchangeableSystemProperties = initUnchangeableSystemProperties();

systemProperties = createSystemProperties();

addLegacyLocaleSystemProperties();

}

}

### 3.1 initUnchangeableSystemProperties

[-> System.java]

private static Properties initUnchangeableSystemProperties() {

VMRuntime runtime = VMRuntime.getRuntime();

Properties p = new Properties();

...

p.put("java.boot.class.path", runtime.bootClassPath());

p.put("java.class.path", runtime.classPath());

// [见小节3.2]

parsePropertyAssignments(p, specialProperties());

...

return p;

}

这个过程会将大量的key-value对保存到Properties对象, 这种重点看specialProperties

#### 3.1.1 parsePropertyAssignments

[-> System.java]

***private static void parsePropertyAssignments(Properties p, String[] assignments) {***

***for (String assignment : assignments) {***

***int split = assignment.indexOf('=');***

***String key = assignment.substring(0, split);***

***String value = assignment.substring(split + 1);***

***p.put(key, value);***

***}***

***}***

将assignments数据解析后保存到Properties对象.

### 3.2 specialProperties

[-> java\_lang\_System.cpp]

static jobjectArray System\_specialProperties(JNIEnv\* env, jclass) {

std::vector<std::string> properties;

char path[PATH\_MAX];

...

const char\* library\_path = getenv("LD\_LIBRARY\_PATH");

if (library\_path == NULL) {

// [见小节3.3]

android\_get\_LD\_LIBRARY\_PATH(path, sizeof(path));

library\_path = path;

}

properties.push\_back(std::string("java.library.path=") + library\_path);

return toStringArray(env, properties);

}

环境变量LD\_LIBRARY\_PATH为空, 可通过adb shell env命令来查看环境变量的值. 接下来进入android\_get\_LD\_LIBRARY\_PATH方法, 该方法调用do\_android\_get\_LD\_LIBRARY\_PATH,见下文:

### 3.3 do\_android\_get\_LD\_LIBRARY\_PATH

[-> linker.cpp]

void do\_android\_get\_LD\_LIBRARY\_PATH(char\* buffer, size\_t buffer\_size) {

//[见小节3.4]

size\_t required\_len = strlen(kDefaultLdPaths[0]) + strlen(kDefaultLdPaths[1]) + 2;

char\* end = stpcpy(buffer, kDefaultLdPaths[0]);

\*end = ':';

strcpy(end + 1, kDefaultLdPaths[1]);

}

可见赋值过程还得看kDefaultLdPaths数组.

### 3.4 kDefaultLdPaths

[-> linker.cpp]

***static const char\* const kDefaultLdPaths[] = {***

***#if defined(\_\_LP64\_\_)***

***"/vendor/lib64",***

***"/system/lib64",***

***#else***

***"/vendor/lib",***

***"/system/lib",***

***#endif***

***nullptr***

***};***

对于linux 64位操作系统则定义\_\_LP64\_\_宏, 可知mLibPaths取值分两种情况:

* 对于64系统,则为/system/lib64和/vendor/lib64;
* 对于32系统,则为/system/lib和/vendor/lib.

也可以通过System.getProperty(“java.library.path”)来获取该值.

## 参考

http://gityuan.com/2017/03/26/load\_library/

<http://blog.csdn.net/mirkerson/article/details/24322207>

<http://blog.csdn.net/woshing123456/article/details/44524051>

## data/dalvik-cache

dalvik-cache名词解释： 在系统data/dalvik-cache文件夹里有很多安装卸载文件（优化过的字节码），这些文件是当你安装好一个应用程序后，系统会自动生成的一个优化过的字节码文件，但是当你频繁安装卸载某些应用软件后可能对应字节码文件不会同时删除，也即是残留的垃圾，这时就需要用到缓存清理助手来帮助增加您手机的可用空间^0^ 特别提示： 清理Dalvik缓存时，需要Root权限 。

安卓手机程序越装越多，系统提示内存空间不足了吧？卸载一些程序也不会增加多少可用空间

系统缓存文件和卸载定制程序留下来的无用垃圾，可以放心全部删除，系统所需文件重启后能自动生成的。删除后重启手机的时候，时间有点久，大概2－3分钟，期间有段时间为黑屏状态，应用不能正常打开，不要担心，一会系统就加载完成，启动加载期间，不要有任何操作，要耐心等待......系统启动加载完毕，一切正常。

至此，宣布成功！

[](http://images.cnitblog.com/blog/363476/201402/082037504515476.jpg)

[](http://images.cnitblog.com/blog/363476/201402/082037518684475.png)

# 初探boot.art与boot.oat

boot.art与boot.oat与其说是ART虚拟机的两种执行格式，不如说他俩就是ART虚拟机的一部分！！！ART离开了这两个文件，也就无法启动了。

boot.art是一个img文件，而boot.oat文件可以将其理解为ART虚拟机的启动类。

这两个文件是dex2oat命令将Android系统必须的的jar包编译生成的，这两个文件相互联系，缺一不可，boot.art这个img文件直接被映射到ART虚拟机的堆空间中，包含了boot.oat中的某些对象实例以及函数地址。

## 产生过程

删除/data/dalvik-cache/arm的boot.art和boot.oat，

Reboot

adb logcat | grep dex2oat

观察生成日志

root@cs500c:/data/dalvik-cache/arm # rm system@framework@boot.oat

root@cs500c:/data/dalvik-cache/arm # reboot

shell@cs500c:logcat | grep dex2oat

|  |
| --- |
| I/dex2oat ( 379): /system/bin/dex2oat --image=/data/dalvik-cache/arm/system@framework@boot.art --dex-file=/system/framework/core-libart.jar --dex-file=/system/  framework/conscrypt.jar --dex-file=/system/framework/okhttp.jar --dex-file=/system/framework/core-junit.jar --dex-file=/system/framework/bouncycastle.jar --dex-  file=/system/framework/ext.jar --dex-file=/system/framework/framework.jar --dex-file=/system/framework/telephony-common.jar --dex-file=/system/framework/voip-co  mmon.jar --dex-file=/system/framework/ims-common.jar --dex-file=/system/framework/mms-common.jar --dex-file=/system/framework/android.policy.jar --dex-file=/sys  tem/framework/apache-xml.jar --oat-file=/data/dalvik-cache/arm/system@framework@boot.oat --instruction-set=arm --instruction-set-features=div --base=0x708ea000  --runtime-arg -Xms64m --runtime-arg -Xmx64m --image-classes=/system/etc/preloaded-classes |

boot.oat和boot.art文件依赖的dalvik的dex来自于BOOTCLASSPATH中指定的jar包。

|  |
| --- |
| shell@cs500c:/ $ echo ${BOOTCLASSPATH}  /system/framework/core-libart.jar:/system/framework/conscrypt.jar:/system/framework/okhttp.jar:/system/framework/core-junit.jar:/system/framework/bouncycastle.jar:/system/framework/ext.jar:/system/framework/**framework**.jar:/system/framework/telephony-common.jar:/system/framework/voip-common.jar:/system/framework/ims-common.jar:/system/framework/mms-common.jar:/system/framework/android.policy.jar:/system/framework/apache-xml.jar |

[初探boot.art与boot.oat](http://www.iloveandroid.net/2015/12/19/AndroidART-2/)

第2章 Android系统的结构

2.1 Android安装文件简介

2.1.1 AndroidSDK目录结构

2.1.2 android.jar及内部结构

2.1.3 SDK帮助文档

2.1.4 解析AndroidSDK实例

2.2 分析Android的系统架构

2.2.1 Android体系结构介绍

2.2.2 Android工程文件结构

2.2.3 应用程序的生命周期

2.3 简析Android内核

2.3.1 Android继承于Linux

2.3.2 Android内核和Linux内核的区别

2.4 简析Android源码

2.4.1 获取并编译Android源码

2.4.2 Android对Linux的改造

2.4.3 为Android构建Linux的操作系统

2.4.4 分析Android源码结构

2.4.5 编译Android源码

2.4.6 运行Android源码

2.5 实践演练——演示两种编译Android程序的方法

2.5.1 编译NativeC的helloworld模块

2.5.2 手工编译C模块

第3章 虚拟机概述

3.1 虚拟机的作用

3.2 Java虚拟机

3.2.1 理解Java虚拟机

3.2.2 Java虚拟机的数据类型

3.2.3 Java虚拟机的体系结构

3.2.4 Java虚拟机的生命周期

3.3 Android虚拟机-DalvikVM

3.3.1 Dalvik架构

3.3.2 和Java虚拟机的差异

3.3.3 DalvikVM的主要特征

3.3.4 DalvikVM的代码结构

3.4 Dalvik控制VM详解

3.5 DalvikVM架构

3.5.1 Dalvik的进程管理

3.5.2 Android的初始化流程

第4章 编译和调试

4.1 Windows环境编译Dalvik

4.2 GDB调试Dalvik

4.2.1 准备工作

4.2.2 GDB调试C程序

4.2.3 GDB调试Dalvik

4.3 使用dexdump

4.3.1 dexdump的反编译功能

4.3.2 使用dexdump查看jar文件

4.4 Dalvik虚拟机编译脚本

4.4.1 Android.mk文件

4.4.2 ReconfigureDvm.mk文件

4.4.3 dvm.mk文件

4.5 Android4.0.1 源码下载、模拟器编译和运行

第5章 Dalvik虚拟机的运作流程

5.1 Dalvik虚拟机相关的可执行程序

5.1.1 dalvikvm

5.1.2 dvz

5.1.3 app\_process

5.2 Dalvik虚拟机的初始化

5.2.1 开始虚拟机的准备工作

5.2.2 初始化跟踪显示系统

5.2.3 初始化垃圾回收器

5.2.4 初始化线程列表和主线程环境参数

5.2.5 分配内部操作方法的表格内存

5.2.6 初始化虚拟机的指令码相关的内容

5.2.7 分配指令寄存器状态的内存

5.2.8 分配指令寄存器状态的内存

5.2.9 初始化虚拟机最基本用的Java库

5.2.10 进一步使用的Java类库线程类

5.2.11 初始化虚拟机使用的异常Java类库

5.2.12 释放字符串哈希表

5.2.13 初始化本地方法库的表

5.2.14 初始化内部本地方法

5.2.15 初始化JNI调用表

5.2.16 缓存Java类库里的反射类

5.2.17 最后的工作

5.3 启动zygote

5.3.1 在init.rc中配置zygote启动参数

5.3.2 启动Socket服务端口

5.3.3 加载preload-classes

5.3.4 i口载preload-resources

5.3.5 使用folk启动新进程

5.4 启动SystemServer进程

5.4.1 启动各种系统服务线程

5.4.2 启动第一个Activity

5.5 class类文件的加载

5.5.1 DexFile在内存中的映射

5.5.2 ClassObject-Class在加载后的表现形式

5.5.3 findClassNolnit-加载Class并生成相应ClassObject的函数

5.5.4 加载基本类库文件

5.5.5 加载用户类文件

5.6 解释执行类

5.6.1 Dalvik虚拟机字节码和JVM字节码的区别

5.6.2 Davik虚拟机的解释器优化

第6章 dex的优化和安全管理

6.1 Androiddex文件优化简介

6.2 dex文件的格式

6.2.1 map\_list

6.2.2 string\_idjtem

6.2.3 type\_idjtem

6.2.4 proto\_idjtem

6.2.5 field\_id\_item

6.2.6 method\_id\_item

6.2.7 class\_def\_item

6.3 dex文件结构

6.3.1 文件头（FileHeader）

6.3.2 魔数字段

6.3.3 检验码字段

6.3.4 SHA-l签名字段

6.3.5 map\_off字段

……

第7章 生命周期管理

第8章 内存分配策略

第9章 垃圾收集

第10章 线程管理

第11章 JNI接口

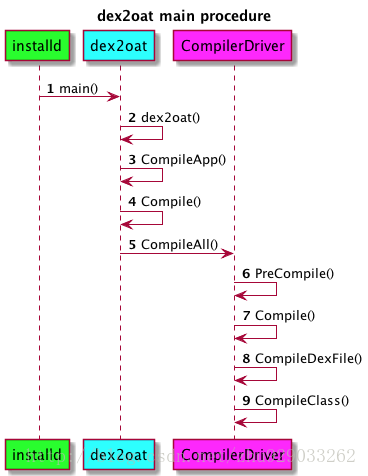
第12章 JIT编译

第13章 异常管理

# dex2oat源码流程

dex2oat是ART运行模式下虚拟机必备的一个组件，主要用来把安装的apk和动态加载的dex等文件转换成oat文件，方便下一步的加载解析，获得其中的类并执行相关方法，所以本文以Android 6.0源码为例，对dex的处理流程尝试做一下分析，了解其中的处理情况。

dex2oat源码位于art\dex2oat\Dex2oat.cc，入口函数为main：



<https://blog.csdn.net/zhu929033262/article/details/78317614>

<https://www.jianshu.com/p/5d7f4ad8b416>

<https://blog.csdn.net/qq1084283172/article/details/78513483>

<https://blog.csdn.net/long375577908/article/details/78190422>

<https://blog.csdn.net/threepigs/article/details/52724748>

<https://blog.csdn.net/threepigs/article/details/52884904>

# Android预优化那些事

Android预优化，也就是把Android在启动或APP在运行时所需要做的一些事情，把这些事情转移到编译APK时完成，来达到更快的Android系统启动速度和更快的APP运行速度。

原理：

# Trace原理

分析Art虚拟机的trace原理，相关源码都位于/art/runtime目录：

/art/runtime/

- signal\_catcher.cc

- runtime.cc

- intern\_table.cc

- thread\_list.cc

- java\_vm\_ext.cc

- class\_linker.cc

- gc/heap.cc

## 概述

Android 6.0系统采用的art虚拟机，所有的Java进程都运行在art之上，当应用发生ANR(Application Not Response，其中最终的一个环节便是向目标进程发送信号SIGNAL\_QUIT, 传统的linux则是终止程序并输出core;而对于Android进程来说当收到SIGQUIT时，Java层面的进程都是跑在虚拟机之上的，ART虚拟机会捕获该信号，并输出相应的traces信息保存到目录/data/anr/traces.txt。

当然也可以通过一条命令来获取指定进程的traces信息，例如输出pid=888的进程信息：

adb shell kill -3 888 //可指定进程pid

### traces.txt

执行完该命令后traces信息的结果保存到文件/data/anr/traces.txt，如下：

//[见小节2.2]

----- pid 888 at 2016-11-11 22:22:22 -----

Cmd line: system\_server

ABI: arm

Build type: optimized

//[见小节3.1]

Zygote loaded classes=4113 post zygote classes=3239

//[见小节3.2]

Intern table: 57550 strong; 9315 weak

//共加载16动态库 [见小节3.3]

JNI: CheckJNI is off; globals=2418 (plus 115 weak)

Libraries: /system/lib/libandroid.so /system/lib/libandroid\_servers.so /system/lib/libaudioeffect\_jni.so /system/lib/libcompiler\_rt.so /system/lib/libjavacrypto.so /system/lib/libjnigraphics.so /system/lib/libmedia\_jni.so /system/lib/librs\_jni.so /system/lib/libsechook.so /system/lib/libshell\_jni.so /system/lib/libsoundpool.so /system/lib/libwebviewchromium\_loader.so /system/lib/libwifi-service.so /vendor/lib/libalarmservice\_jni.so /vendor/lib/liblocationservice.so libjavacore.so (16)

//已分配堆内存大小40MB，其中29M已用，总分配207772个对象 [见小节3.4]

Heap: 27% free, 29MB/40MB; 307772 objects

... //省略GC相关信息

//当前进程总99个线程[见小节3.5]

DALVIK THREADS (99):

//主线程调用栈[见小节3.6]

"main" prio=5 tid=1 Native

| group="main" sCount=1 dsCount=0 obj=0x75bd9fb0 self=0x5573d4f770

| sysTid=12078 nice=-2 cgrp=default sched=0/0 handle=0x7fa75fafe8

| state=S schedstat=( 5907843636 827600677 5112 ) utm=453 stm=137 core=0 HZ=100

| stack=0x7fd64ef000-0x7fd64f1000 stackSize=8MB

| held mutexes=

//内核栈[见小节3.6.2]

kernel: \_\_switch\_to+0x70/0x7c

kernel: SyS\_epoll\_wait+0x2a0/0x324

kernel: SyS\_epoll\_pwait+0xa4/0x120

kernel: cpu\_switch\_to+0x48/0x4c

native: #00 pc 0000000000069be4 /system/lib64/libc.so (\_\_epoll\_pwait+8)

native: #01 pc 000000000001cca4 /system/lib64/libc.so (epoll\_pwait+32)

native: #02 pc 000000000001ad74 /system/lib64/libutils.so (\_ZN7android6Looper9pollInnerEi+144)

native: #03 pc 000000000001b154 /system/lib64/libutils.so (\_ZN7android6Looper8pollOnceEiPiS1\_PPv+80)

native: #04 pc 00000000000d4bc0 /system/lib64/libandroid\_runtime.so (\_ZN7android18NativeMessageQueue8pollOnceEP7\_JNIEnvP8\_jobjecti+48)

native: #05 pc 000000000000082c /data/dalvik-cache/arm64/system@framework@boot.oat (Java\_android\_os\_MessageQueue\_nativePollOnce\_\_JI+144)

at android.os.MessageQueue.nativePollOnce(Native method)

at android.os.MessageQueue.next(MessageQueue.java:323)

at android.os.Looper.loop(Looper.java:135)

at com.android.server.SystemServer.run(SystemServer.java:290)

at com.android.server.SystemServer.main(SystemServer.java:175)

at java.lang.reflect.Method.invoke!(Native method)

at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:738)

at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:628)

"Binder\_1" prio=5 tid=8 Native

| group="main" sCount=1 dsCount=0 obj=0x12c610a0 self=0x5573e5c750

| sysTid=12092 nice=0 cgrp=default sched=0/0 handle=0x7fa2743450

| state=S schedstat=( 796240075 863170759 3586 ) utm=50 stm=29 core=1 HZ=100

| stack=0x7fa2647000-0x7fa2649000 stackSize=1013KB

| held mutexes=

kernel: \_\_switch\_to+0x70/0x7c

kernel: binder\_thread\_read+0xd78/0xeb0

kernel: binder\_ioctl\_write\_read+0x178/0x24c

kernel: binder\_ioctl+0x2b0/0x5e0

kernel: do\_vfs\_ioctl+0x4a4/0x578

kernel: SyS\_ioctl+0x5c/0x88

kernel: cpu\_switch\_to+0x48/0x4c

native: #00 pc 0000000000069cd0 /system/lib64/libc.so (\_\_ioctl+4)

native: #01 pc 0000000000073cf4 /system/lib64/libc.so (ioctl+100)

native: #02 pc 000000000002d6e8 /system/lib64/libbinder.so (\_ZN7android14IPCThreadState14talkWithDriverEb+164)

native: #03 pc 000000000002df3c /system/lib64/libbinder.so (\_ZN7android14IPCThreadState20getAndExecuteCommandEv+24)

native: #04 pc 000000000002e114 /system/lib64/libbinder.so (\_ZN7android14IPCThreadState14joinThreadPoolEb+124)

native: #05 pc 0000000000036c38 /system/lib64/libbinder.so (???)

native: #06 pc 000000000001579c /system/lib64/libutils.so (\_ZN7android6Thread11\_threadLoopEPv+208)

native: #07 pc 0000000000090598 /system/lib64/libandroid\_runtime.so (\_ZN7android14AndroidRuntime15javaThreadShellEPv+96)

native: #08 pc 0000000000014fec /system/lib64/libutils.so (???)

native: #09 pc 0000000000067754 /system/lib64/libc.so (\_ZL15\_\_pthread\_startPv+52)

native: #10 pc 000000000001c644 /system/lib64/libc.so (\_\_start\_thread+16)

(no managed stack frames)

... //此处省略剩余的N个线程.

接下来从虚拟机角度说说目标进程收到该信号的处理过程，每一行关键信息都说明其所对应的输出方法

## ART信号捕获

### SignalCatcher

[-> SignalCatcher.cc]

void\* SignalCatcher::Run(void\* arg) {

SignalCatcher\* signal\_catcher = reinterpret\_cast<SignalCatcher\*>(arg);

Runtime\* runtime = Runtime::Current();

Thread\* self = Thread::Current();

//当前进程状态处于非Runnable是

DCHECK\_NE(self->GetState(), kRunnable);

{

MutexLock mu(self, signal\_catcher->lock\_);

signal\_catcher->thread\_ = self;

signal\_catcher->cond\_.Broadcast(self);

}

//设置需要handle的信号

SignalSet signals;

signals.Add(SIGQUIT); //信号3

signals.Add(SIGUSR1); //信号10

while (true) {

int signal\_number = signal\_catcher->WaitForSignal(self, signals);

if (signal\_catcher->ShouldHalt()) {

runtime->DetachCurrentThread();

return nullptr;

}

switch (signal\_number) {

case SIGQUIT:

//收到信号3 【见小节2.2】

signal\_catcher->HandleSigQuit();

break;

case SIGUSR1:

signal\_catcher->HandleSigUsr1();

break;

default:

LOG(ERROR) << "Unexpected signal %d" << signal\_number;

break;

}

}

}

### 2.2 SignalCatcher::HandleSigQuit

[-> signal\_catcher.cc]

void SignalCatcher::HandleSigQuit() {

Runtime\* runtime = Runtime::Current();

std::ostringstream os;

os << "\n" << "----- pid " << getpid() << " at " << GetIsoDate() << " -----\n";

DumpCmdLine(os);

std::string fingerprint = runtime->GetFingerprint();

os << "Build fingerprint: '" << (fingerprint.empty() ? "unknown" : fingerprint) << "'\n";

os << "ABI: '" << GetInstructionSetString(runtime->GetInstructionSet()) << "'\n";

os << "Build type: " << (kIsDebugBuild ? "debug" : "optimized") << "\n";

// [见小节2.3]

runtime->DumpForSigQuit(os);

os << "----- end " << getpid() << " -----\n";

// [见小节3.7]

Output(os.str());

}

### 2.3 Runtime::DumpForSigQuit

[-> runtime.cc]

void Runtime::DumpForSigQuit(std::ostream& os) {

GetClassLinker()->DumpForSigQuit(os); //[见小节3.1]

GetInternTable()->DumpForSigQuit(os); //[见小节3.2]

GetJavaVM()->DumpForSigQuit(os); //[见小节3.3]

GetHeap()->DumpForSigQuit(os); //[见小节3.4]

TrackedAllocators::Dump(os);

os << "\n";

thread\_list\_->DumpForSigQuit(os); //[见小节3.5]

BaseMutex::DumpAll(os);

}

## trace信息

dsf

### 3.1 ClassLinker

[-> class\_linker.cc]

void ClassLinker::DumpForSigQuit(std::ostream& os) {

Thread\* self = Thread::Current();

if (dex\_cache\_image\_class\_lookup\_required\_) {

ScopedObjectAccess soa(self);

MoveImageClassesToClassTable();

}

ReaderMutexLock mu(self, \*Locks::classlinker\_classes\_lock\_);

os << "Zygote loaded classes=" << pre\_zygote\_class\_table\_.Size() << " post zygote classes="

<< class\_table\_.Size() << "\n";

}

### 3.2 InternTable

[-> intern\_table.cc]

void InternTable::DumpForSigQuit(std::ostream& os) const {

os << "Intern table: " << StrongSize() << " strong; " << WeakSize() << " weak\n";

}

### 3.3 JavaVMExt

[-> java\_vm\_ext.cc]

void JavaVMExt::DumpForSigQuit(std::ostream& os) {

os << "JNI: CheckJNI is " << (check\_jni\_ ? "on" : "off");

if (force\_copy\_) {

os << " (with forcecopy)";

}

Thread\* self = Thread::Current();

{

ReaderMutexLock mu(self, globals\_lock\_);

os << "; globals=" << globals\_.Capacity();

}

{

MutexLock mu(self, weak\_globals\_lock\_);

if (weak\_globals\_.Capacity() > 0) {

os << " (plus " << weak\_globals\_.Capacity() << " weak)";

}

}

os << '\n';

{

MutexLock mu(self, \*Locks::jni\_libraries\_lock\_);

os << "Libraries: " << Dumpable<Libraries>(\*libraries\_) << " (" << libraries\_->size() << ")\n";

}

}

### 3.4 Heap

[-> heap.cc]

void Heap::DumpForSigQuit(std::ostream& os) {

os << "Heap: " << GetPercentFree() << "% free, " << PrettySize(GetBytesAllocated()) << "/"

<< PrettySize(GetTotalMemory()) << "; " << GetObjectsAllocated() << " objects\n";

DumpGcPerformanceInfo(os); //输出大量gc相关的信息

}

DumpGcPerformanceInfo()这个方法的参数非常多,先省略, 后续再单独用一篇文章来讲解.

### 3.5 ThreadList

[-> thread\_list.cc]

void ThreadList::DumpForSigQuit(std::ostream& os) {

{

ScopedObjectAccess soa(Thread::Current());

if (suspend\_all\_historam\_.SampleSize() > 0) {

Histogram<uint64\_t>::CumulativeData data;

suspend\_all\_historam\_.CreateHistogram(&data);

suspend\_all\_historam\_.PrintConfidenceIntervals(os, 0.99, data); // Dump time to suspend.

}

}

Dump(os); // [见小节3.5.1]

DumpUnattachedThreads(os); //[见小节3.5.2]

}

#### 3.5.1 Dump

[-> thread\_list.cc]

void ThreadList::Dump(std::ostream& os) {

{

MutexLock mu(Thread::Current(), \*Locks::thread\_list\_lock\_);

//输出当前进程的线程个数

os << "DALVIK THREADS (" << list\_.size() << "):\n";

}

DumpCheckpoint checkpoint(&os);

//执行checkpoint检查

size\_t threads\_running\_checkpoint = RunCheckpoint(&checkpoint);

if (threads\_running\_checkpoint != 0) {

checkpoint.WaitForThreadsToRunThroughCheckpoint(threads\_running\_checkpoint);

}

}

DALVIK THREADS (25)代表的是当前虚拟机中的线程个数为25. 另外, 此处RunCheckpoint方法比较重要,涉及进程的suspend状态问题.

#### 3.5.2 DumpUnattachedThreads

[-> thread\_list.cc]

void ThreadList::DumpUnattachedThreads(std::ostream& os) {

DIR\* d = opendir("/proc/self/task");

if (!d) {

return;

}

Thread\* self = Thread::Current();

dirent\* e;

while ((e = readdir(d)) != nullptr) {

char\* end;

pid\_t tid = strtol(e->d\_name, &end, 10);

if (!\*end) {

bool contains;

{

MutexLock mu(self, \*Locks::thread\_list\_lock\_);

contains = Contains(tid);

}

if (!contains) {

DumpUnattachedThread(os, tid); //[见小节3.6]

}

}

}

closedir(d);

}

获取当前进程中所有的线程

### 3.6 DumpUnattachedThread

[-> thread\_list.cc]

static void DumpUnattachedThread(std::ostream& os, pid\_t tid) NO\_THREAD\_SAFETY\_ANALYSIS {

Thread::DumpState(os, nullptr, tid); //[见小节3.6.1]

DumpKernelStack(os, tid, " kernel: ", false); //[见小节3.6.2]

os << "\n";

}

将进程中的每个线程都执行一次该方法

#### 3.6.1 Thread::DumpState

[-> thread.cc]

void Thread::DumpState(std::ostream& os, const Thread\* thread, pid\_t tid) {

std::string group\_name;

int priority;

bool is\_daemon = false;

Thread\* self = Thread::Current();

if (thread != nullptr) {

... //若为空则不进入该分支

}

if (gAborting == 0 && self != nullptr && thread != nullptr && thread->tlsPtr\_.opeer != nullptr) {

ScopedObjectAccessUnchecked soa(self);

priority = soa.DecodeField(WellKnownClasses::java\_lang\_Thread\_priority)

->GetInt(thread->tlsPtr\_.opeer);

is\_daemon = soa.DecodeField(WellKnownClasses::java\_lang\_Thread\_daemon)

->GetBoolean(thread->tlsPtr\_.opeer);

mirror::Object\* thread\_group =

soa.DecodeField(WellKnownClasses::java\_lang\_Thread\_group)->GetObject(thread->tlsPtr\_.opeer);

if (thread\_group != nullptr) {

ArtField\* group\_name\_field =

soa.DecodeField(WellKnownClasses::java\_lang\_ThreadGroup\_name);

mirror::String\* group\_name\_string =

reinterpret\_cast<mirror::String\*>(group\_name\_field->GetObject(thread\_group));

group\_name = (group\_name\_string != nullptr) ? group\_name\_string->ToModifiedUtf8() : "<null>";

}

} else {

priority = GetNativePriority();

}

std::string scheduler\_group\_name(GetSchedulerGroupName(tid));

if (scheduler\_group\_name.empty()) {

scheduler\_group\_name = "default";

}

//线程名，优先级，tid，线程状态

if (thread != nullptr) {

os << '"' << \*thread->tlsPtr\_.name << '"';

if (is\_daemon) {

os << " daemon";

}

os << " prio=" << priority

<< " tid=" << thread->GetThreadId()

<< " " << thread->GetState(); //获取线程状态

if (thread->IsStillStarting()) {

os << " (still starting up)";

}

os << "\n";

//当线程为空指针，则输出not attached

} else {

os << '"' << ::art::GetThreadName(tid) << '"'

<< " prio=" << priority

<< " (not attached)\n";

}

if (thread != nullptr) {

MutexLock mu(self, \*Locks::thread\_suspend\_count\_lock\_);

os << " | group=\"" << group\_name << "\""

<< " sCount=" << thread->tls32\_.suspend\_count

<< " dsCount=" << thread->tls32\_.debug\_suspend\_count

<< " obj=" << reinterpret\_cast<void\*>(thread->tlsPtr\_.opeer)

<< " self=" << reinterpret\_cast<const void\*>(thread) << "\n";

}

os << " | sysTid=" << tid

<< " nice=" << getpriority(PRIO\_PROCESS, tid)

<< " cgrp=" << scheduler\_group\_name;

if (thread != nullptr) {

int policy;

sched\_param sp;

CHECK\_PTHREAD\_CALL(pthread\_getschedparam, (thread->tlsPtr\_.pthread\_self, &policy, &sp),

\_\_FUNCTION\_\_);

os << " sched=" << policy << "/" << sp.sched\_priority

<< " handle=" << reinterpret\_cast<void\*>(thread->tlsPtr\_.pthread\_self);

}

os << "\n";

std::string scheduler\_stats;

//读取节点/proc/self/task/%d/schedstat

if (ReadFileToString(StringPrintf("/proc/self/task/%d/schedstat", tid), &scheduler\_stats)) {

scheduler\_stats.resize(scheduler\_stats.size() - 1);

} else {

scheduler\_stats = "0 0 0";

}

char native\_thread\_state = '?';

int utime = 0;

int stime = 0;

int task\_cpu = 0;

// 读取节点/proc/self/task/%d/stat

GetTaskStats(tid, &native\_thread\_state, &utime, &stime, &task\_cpu);

os << " | state=" << native\_thread\_state

<< " schedstat=( " << scheduler\_stats << " )"

<< " utm=" << utime

<< " stm=" << stime

<< " core=" << task\_cpu

<< " HZ=" << sysconf(\_SC\_CLK\_TCK) << "\n";

if (thread != nullptr) {

os << " | stack=" << reinterpret\_cast<void\*>(thread->tlsPtr\_.stack\_begin) << "-"

<< reinterpret\_cast<void\*>(thread->tlsPtr\_.stack\_end) << " stackSize="

<< PrettySize(thread->tlsPtr\_.stack\_size) << "\n";

os << " | held mutexes=";

for (size\_t i = 0; i < kLockLevelCount; ++i) {

if (i != kMonitorLock) {

BaseMutex\* mutex = thread->GetHeldMutex(static\_cast<LockLevel>(i));

if (mutex != nullptr) {

os << " \"" << mutex->GetName() << "\"";

if (mutex->IsReaderWriterMutex()) {

ReaderWriterMutex\* rw\_mutex = down\_cast<ReaderWriterMutex\*>(mutex);

if (rw\_mutex->GetExclusiveOwnerTid() == static\_cast<uint64\_t>(tid)) {

os << "(exclusive held)";

} else {

os << "(shared held)";

}

}

}

}

}

os << "\n";

}

}

#### 3.6.2 DumpKernelStack

[-> art/runtime/utils.cc]

//前缀prefix= “kernel:”

void DumpKernelStack(std::ostream& os, pid\_t tid, const char\* prefix, bool include\_count) {

if (tid == GetTid()) {

return;

}

//内核栈是通过读取节点/proc/self/task/[tid]/stack

std::string kernel\_stack\_filename(StringPrintf("/proc/self/task/%d/stack", tid));

std::string kernel\_stack;

if (!ReadFileToString(kernel\_stack\_filename, &kernel\_stack)) {

os << prefix << "(couldn't read " << kernel\_stack\_filename << ")\n";

return;

}

std::vector<std::string> kernel\_stack\_frames;

Split(kernel\_stack, '\n', &kernel\_stack\_frames);

kernel\_stack\_frames.pop\_back();

for (size\_t i = 0; i < kernel\_stack\_frames.size(); ++i) {

const char\* text = kernel\_stack\_frames[i].c\_str();

const char\* close\_bracket = strchr(text, ']');

if (close\_bracket != nullptr) {

text = close\_bracket + 2;

}

os << prefix;

if (include\_count) {

os << StringPrintf("#%02zd ", i);

}

os << text << "\n";

}

}

内核栈是通过读取节点/proc/self/task/[tid]/stack

### 3.7 Output

[-> signal\_catcher.cc]

void SignalCatcher::Output(const std::string& s) {

if (stack\_trace\_file\_.empty()) {

LOG(INFO) << s;

return;

}

ScopedThreadStateChange tsc(Thread::Current(), kWaitingForSignalCatcherOutput);

// 以追加的方式输出文件

int fd = open(stack\_trace\_file\_.c\_str(), O\_APPEND | O\_CREAT | O\_WRONLY, 0666);

if (fd == -1) {

PLOG(ERROR) << "Unable to open stack trace file '" << stack\_trace\_file\_ << "'";

return;

}

std::unique\_ptr<File> file(new File(fd, stack\_trace\_file\_, true));

bool success = file->WriteFully(s.data(), s.size());

if (success) {

success = file->FlushCloseOrErase() == 0;

} else {

file->Erase();

}

if (success) {

// 输出该log, 则代表traces信息已完整的输出到文件

LOG(INFO) << "Wrote stack traces to '" << stack\_trace\_file\_ << "'";

} else {

PLOG(ERROR) << "Failed to write stack traces to '" << stack\_trace\_file\_ << "'";

}

}

[小节2.2]执行完DumpForSigQuit()后,便会执行SignalCatcher::Output()方法. 将前面所有生成的信息一并输出到文件/data/anr/traces.txt.

## 四. 实例

"Binder\_1" prio=5 tid=8 Native

| group="main" sCount=1 dsCount=0 obj=0x12c610a0 self=0x5573e5c750

| sysTid=12092 nice=0 cgrp=default sched=0/0 handle=0x7fa2743450

| state=S schedstat=( 796240075 863170759 3586 ) utm=50 stm=29 core=1 HZ=100

| stack=0x7fa2647000-0x7fa2649000 stackSize=1013KB

| held mutexes=

**说明：**

* 第0行:
  + 线程名: Binder\_1（如有daemon则代表守护线程)
  + prio: 线程优先级
  + tid: 线程内部id
  + 线程状态: NATIVE
* 第1行:
  + group: 线程所属的线程组
  + sCount: 线程挂起次数
  + dsCount: 用于调试的线程挂起次数
  + obj: 当前线程关联的java线程对象
  + self: 当前线程地址
* 第2行：
  + sysTid：线程真正意义上的tid
  + nice: 调度有优先级
  + cgrp: 进程所属的进程调度组
  + sched: 调度策略
  + handle: 函数处理地址
* 第3行：
  + state: 线程状态
  + schedstat: CPU调度时间统计, 见proc/[pid]/task/[tid]/schedstat
  + utm/stm: 用户态/内核态的CPU时间(单位是jiffies), 见proc/[pid]/task/[tid]/stat
  + core: 该线程的最后运行所在核
  + HZ: 时钟频率
* 第4行：
  + stack：线程栈的地址区间
  + stackSize：栈的大小
* 第5行：
  + mutex: 所持有mutex类型，有独占锁exclusive和共享锁shared两类

## REF

# Loader

Ads

# Framework

---------------------

作者：HaleYangD

来源：CSDN

原文：https://blog.csdn.net/cscs111/article/details/77558168

版权声明：本文为博主原创文章，转载请附上博文链接！原文：https://blog.csdn.net/cscs111/article/details/77558168

版权声明：本文为博主原创文章，转载请附上博文链接！

# Android文件说明

## **jar与sources.jar**

编译后的class文件。这种文件是看不到源码的

Attach Source.关联源码。对应的source.jar里面

## **jar和aar区别：**

\*.jar：只包含了class文件与清单文件，不包含资源文件，如图片等所有res中的文件。

\*.aar：包含所有资源，class以及res资源文件全部包含

如果你只是一个简单的类库那么使用生成的\*.jar文件即可；如果你的是一个UI库，包含一些自己写的控件布局文件以及字体等资源文件那么就只能使用\*.aar文件。

注意：在更改Android的系统应用时，要引用一个aar文件。之前引用的主要是jar包，而aar文件包含Android的资源文件，如：布局、样式、图片等，如果按照源码中jar的引用方式会遇到编译不过的问题，提示找不到相关的资源文件。

LOCAL\_AAPT\_FLAGS := \ --auto-add-overlay \ --extra-packages <aar package name>

[引用一个aar文件](http://www.voidcn.com/article/p-cmriynxn-bqy.html)

[引用jar及so文件](https://blog.csdn.net/zuiwuyuan/article/details/48201655)

# Failed resolution of: Lorg/msgpack/core/MessagePack;

2018-10-17 03:16:44.770 27549-27549/system\_process E/AndroidRuntime: \*\*\* FATAL EXCEPTION IN SYSTEM PROCESS: main

java.lang.NoClassDefFoundError: Failed resolution of: Lorg/msgpack/core/MessagePack;

at com.dpad.core.analytics.util.MsgpackUtil.toMsgpackByte(MsgpackUtil.java:44)

at com.dpad.core.analytics.manager.BaseInfoManager.sendBaseInfo(BaseInfoManager.java:75)

at com.dpad.core.analytics.manager.ReportManager.distribute(ReportManager.java:78)

at com.dpad.core.analytics.manager.NetStateManager.notifyReportManager(NetStateManager.java:95)

at com.dpad.core.analytics.manager.NetStateManager$1.onReceive(NetStateManager.java:104)

at android.app.LoadedApk$ReceiverDispatcher$Args.run(LoadedApk.java:1122)

at android.os.Handler.handleCallback(Handler.java:755)

at android.os.Handler.dispatchMessage(Handler.java:95)

at android.os.Looper.loop(Looper.java:154)

at com.android.server.SystemServer.run(SystemServer.java:367)

at com.android.server.SystemServer.main(SystemServer.java:226)

at java.lang.reflect.Method.invoke(Native Method)

at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:957)

at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:825)

Caused by: java.lang.ClassNotFoundException: Didn't find class "org.msgpack.core.MessagePack" on path: DexPathList[[zip file "/system/framework/services.jar", zip file "/system/framework/ethernet-service.jar", zip file "/system/framework/wifi-service.jar"],

# REF

深入解析Android虚拟机 平装 – 2016年9月1日钟世礼 (作者)

https://www.2cto.com/ebook/201605/62817.html#download

Android Dalvik虚拟机结构及机制剖析(第2卷):Dalvik虚拟机各模块机制分析2014-08

吴艳霞、 张国印