# 1 关于Android虚拟机

Android系统的应用层是采用Java开发的，由于Java语言的跨平台特性，所以Java的代码必须运行在虚拟机中。正因为这个特性，Android系统也实现了自己的一个类似JVM但是更适合嵌入式平台的虚拟机——Dalvik。Dalvik的功能等同于JVM，为Android平台上的Java代码提供了运行环境。

JVM是一种基于堆栈的机器，而DVM则是基于寄存器的。

# 2 Dalvik虚拟机

## 2.1 Dalvik虚拟机特征

1.使用的空间小；

2.常量池己被修改为只使用32位的索引，以简化解释器，寻址更快；

3.Dalvik VM 使用自己的字节码，而非Java字节码；

4.优化的dex可执行文件

5.所有的Android程序都运行在系统进程里，每一个进程对应着一个Dalvik虚拟机实例。

在DVM中，一个应用总会定义很多类，编译完成后，即会有很多响应的Class文件，Class文件间会有不少冗余的信息；而Dex文件格式会把所有的Class文件内容整合到一个文件中。这样，除了减少整体文件尺寸、IO操作，也提高了类的查找速度。原来每个类文件中的常量池，在Dex文件中由一个常量池来管理。

每一个Android应用都运行在一个DVM实例中，而每一个虚拟机实例都是一个独立的进程空间。虚拟机的线程控制、内存分配和管理、Mutex等都依赖底层的操作系统实现。所有Android应用的线程都对应一个Linux线程，虚拟机因而可以更多的依赖操作系统的线程调度和管理机制。

不同的应用在不同的进程空间里运行，加之对不同来源的应用都使用不同的Linux用户来运行，这可以最大程度的保护应用的安全和独立运行。

Zygote是一个虚拟机进程，同时也是一个虚拟机实例的孵化器，每当系统要求执行一个Android应用程序的时候，Zygote就会Fork出一个子进程来执行该应用程序。这样做的好处是：Zygote进程是在系统启动时产生的，它会完成虚拟机的初始化、库加载、预置类库的加载和初始化等操作，而在系统需要一个新的虚拟机实例时，Zygote通过复制自身，最快速的提供一个子系统。另外，对于一些只读的系统库，所有虚拟机实例都与Zygote共享一块内存区域，大大节省了内存开销。

相对于基于堆栈的虚拟机实现，基于寄存器的虚拟机实现虽然在硬件通用性上要差一些，但是它在代码的执行效率上却更胜一筹。在基于寄存器的虚拟机里，可以更有效的减少冗余指令的分发和减少内存的读写访问。

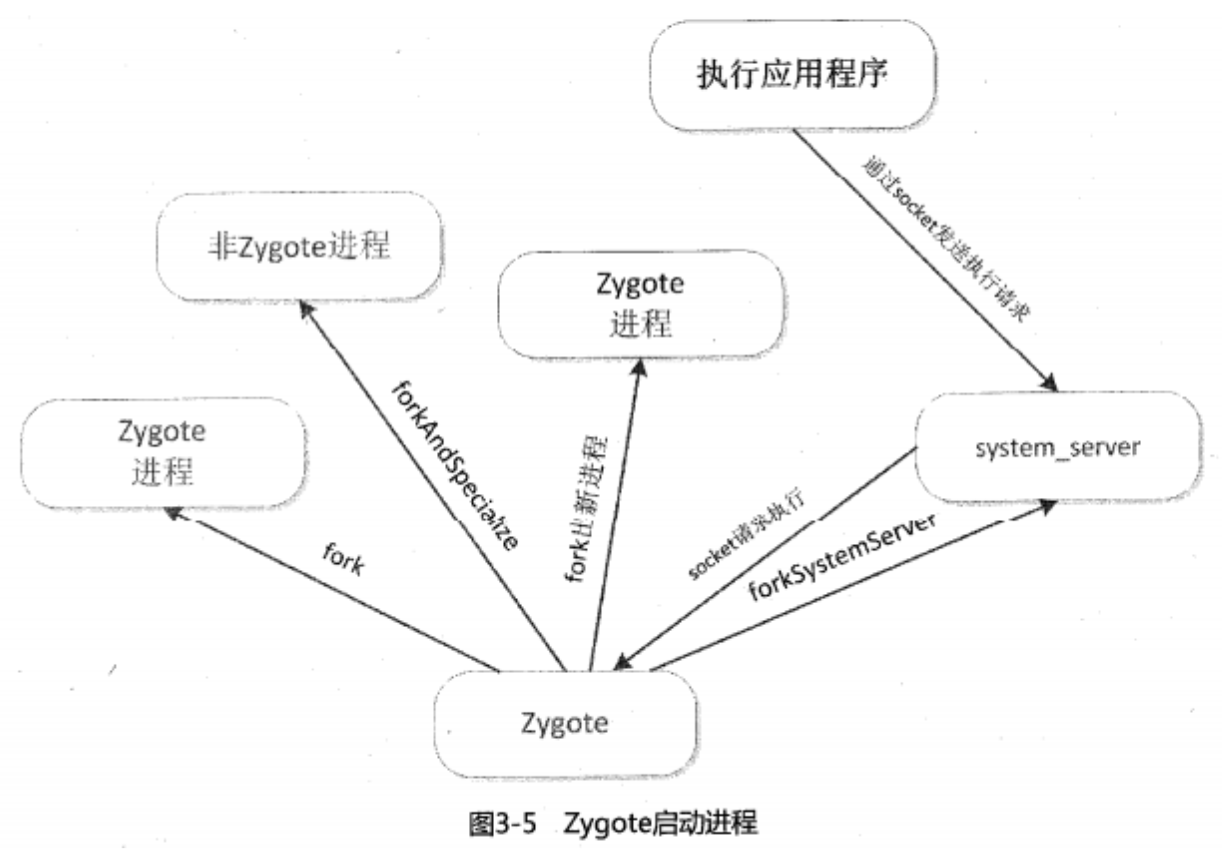
## 2.2 Dalvik的进程管理

Dalvik进程管理是依赖于Linux的进程体系结构的，如要为应用程序创建一个进程，它会使用Linux的Fork机制来复制一个进程（复制进程往往比创建进程的效率更高）。

Zygote通过Init进程启动。首先会孕育出System\_Server（Android绝大多数系统服务的守护进程，它会监听socket，等待请求命令，当有一个应用程序启动时，就会像它发出请求，Zygote就会Fork出一个新的应用程序进程）。每当系统要求执行一个Android应用程序时，Zygote就会运用Linux的Fork机制产生一个子进程来执行该应用程序。

## 2.3 Dalvik虚拟机如何执行程序

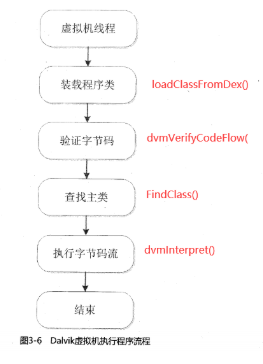
1. Android系统启动并加载内核后会执行init进程
2. 完成设备的初始化
3. 读取init.c
4. 启动系统中的重要外部程序Zygote
5. Zygote进程启动后



1. 初始化Dalvik虚拟机 ---> 启动system\_server进程 ---> 进入Zygote模式，通过socket等待命令
2. 当启动一个Android应用程序时，系统通过Binder IPC方式发送执行请求给system\_server，system\_server以socket请求方式发送给Zygote
3. Zygote收到命令后通过fork自身创建一个Dalvik虚拟机的实例，fork成功后就把工作交给Dalvik虚拟机，来执行应用程序的入口函数，这样一个程序就启动了。

**Dalvik虚拟机执行流程**

1. Dalvik虚拟机首先通过loadClassFromDex()完成类的装载工作，每个类被成功解析后，都会在运行时环境中转化一个ClassObject类型的数据结构(虚拟机使用gDvm.loadedClasses全局散列表来存储与查询所有装载进来的类)。
2. 随后字节码校验器使用dvmVerifyCodeFlow()对装入的代码进行校验
3. 接着虚拟机调用FindClass()查询并装载main()类
4. 随后调用dvmInterpret()初始化解释器并执行字节码流。



## 2.4 Dalvik虚拟机的执行方式

即时编译。即时编译 (Just-in-time compilatino JIT)又称动态编译， 是一种通过在运行时将字节码翻译为机器码使得程序的执行速度加快的技术。主流的JIT包括两种字节码编译方式：

1.method方式：以函数或方法为单位进行编译。

2.trace方式：以trace为单位进行编译。

Dalvik虚拟机默认采用trace方式编译代码，同时支持method方式。

在函数中，只有少数的代码是顺序执行的，多数代码都有好多条执行路径，而其中的一些路径在实际运行过程中是很少执行的，这部分路径称为冷路径。使用trace的方式进行编译，能快速的获取热路径的代码进行编译，用更短的时间和更少的内存来编译代码。

## 参考链接

<https://blog.csdn.net/weixin_34352005/article/details/92549308>

<https://www.jianshu.com/p/6bdbbab73705>

<https://www.jianshu.com/p/f29c04a81225>

# 3 Dalvik字节码指令及Smali文件

## 3.1 Dalvik寄存器

Dalvik中用的寄存器都是32位,64位类型数据则用两个相邻的32位寄存器表示,也就是对于double这种64位类型的数据,需要用到两个32位寄存器来存储.

我们知道Dalvik最多支持65536个寄存器(编号从0~65535),但是在ARM架构的cpu中只存在37个寄存器,那么这种不对称是怎么解决的呢?

Dalvik中的寄存器是虚拟寄存器, 通过映射真实的寄存器来实现.我们知道每个Dalvik维护了一个调用栈,该调用栈就是用来支持虚拟寄存器和真实寄存器相互映射的.在执行具体函数时,Dalvik会根据.registers指令来确定该函数要用到的寄存器数目.具体的原理,可以自行参考Davilk的实现.

下面我们谈到的寄存器都是虚拟寄存器.

**寄存器的使用规则**

对于一个使用m个寄存器(m=局部变量寄存器个数l+参数寄存器个数n)的方法而言,局部寄存器使用从v0开始的l个寄存器,而参数寄存器则使用最后的n个寄存器.举个例子说明假设实例方法test(String a,String b)一共使用了5个寄存器:0,1,2,3,4,那么参数寄存器是能使用2,3,4这三个寄存器,如图:



**寄存器的命名**

寄存器有两种不同的命名方法:v字命名法和p字命 名法.这两种命名法仅仅是影响了字节码的可读性.

v字命名法：以小写字母v开头的方式表示方法中使用的局部变量和参数.

对于上面实例方法test(String a,String b)来说,v0,v1为局部变量能够使用的寄存器,v2,v3,v4为参数能够使用的寄存器:



p字命名法：以小写字母p开头的方式表示参数,参数名称从p0开始,依次增大.局部变量能够使用的寄存器仍然是以v开头.

对于上面实例方法test(String a,String b)来说,v0,v1为局部变量能够使用的寄存器,p0,p1,p2为参数能够使用的寄存器:



## 3.2 Dalvik描述符

与JVM相类似,Davilk字节码中同样有一套用于描述类型,方法,字段的方法,这些方法结合Davilk的指令便形成了完整的汇编代码.

**字节码和数据类型**

Davilk字节码只有两种类型:基本类型和引用类型.对象和数组都是引用类型,Davilk中对字节码类型的描述和JVM中的描述符规则一致:对于基本类型和无返回值的void类型都是用一个大写字母表示,对象类型则用字母L加对象的全限定名来表示.数组则用[来表示,具体规则如下所示:

（全限定名是什么以String为例,其完整名称是java.lang.String,那么其全限定名就是java/lang/String;,即java.lang.String的”.”用”/”代替,并在末尾添加分号”;”做结束符.）







## 3.2详解smali文件

上面我们介绍了Dalvik的相关指令,下面我们则来认识一下smali文件.尽管我们使用java来写Android应用,但是Dalvik并不直接加载.class文件,而是通过dx工具将.class文件优化成.dex文件,然后交由Dalvik加载.这样说来,我们无法通过分析.class来直接分析apk文件,而是需要借助工具baksmali.jar反编译dex文件来获得对应smali文件,smali文件可以认为是Davilk的字节码文件,但是并两者并不完全等同.

通过baksmali.jar反编译出来每个.smali,都对应与java中的一个类,每个smali文件都是Davilk指令组成的,并遵循一定的结构.smali存在很多的指令用于描述对应的java文件,所有的指令都以”.”开头,常用的指令如下:

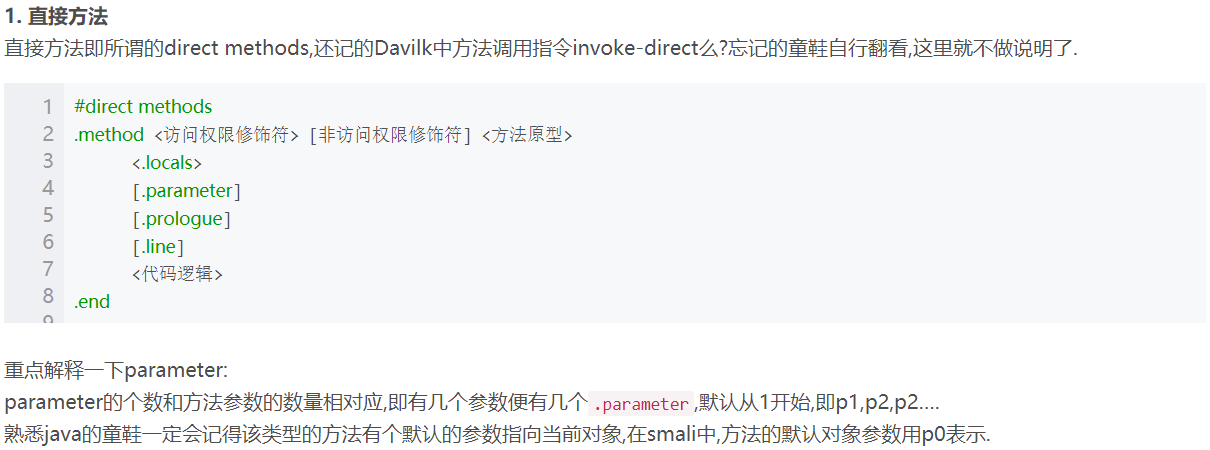




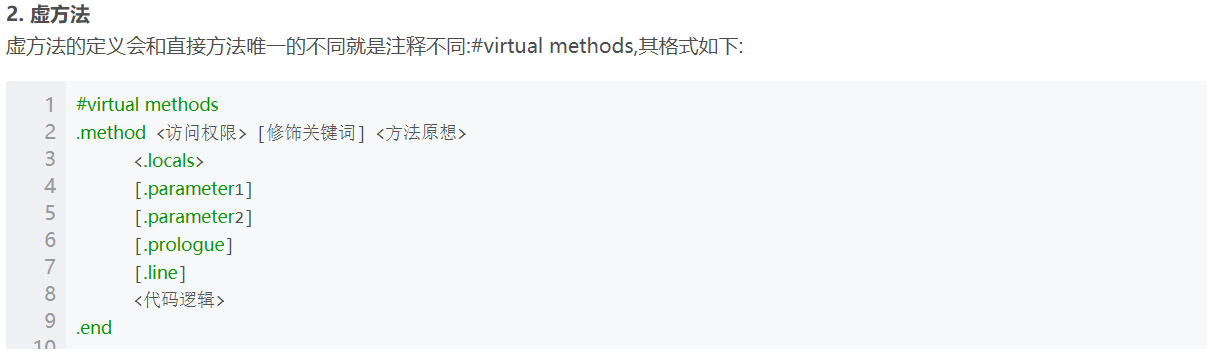


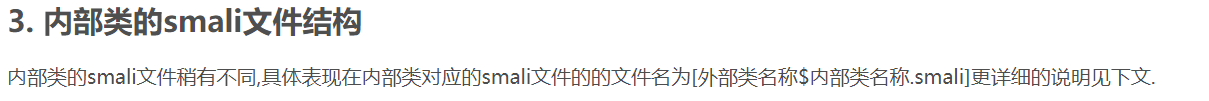


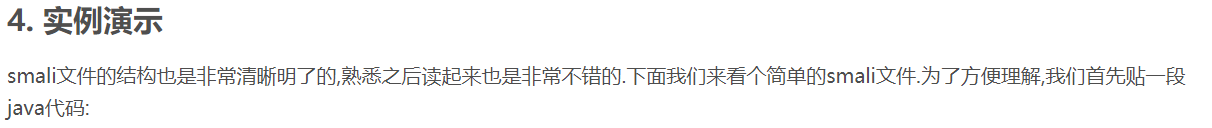




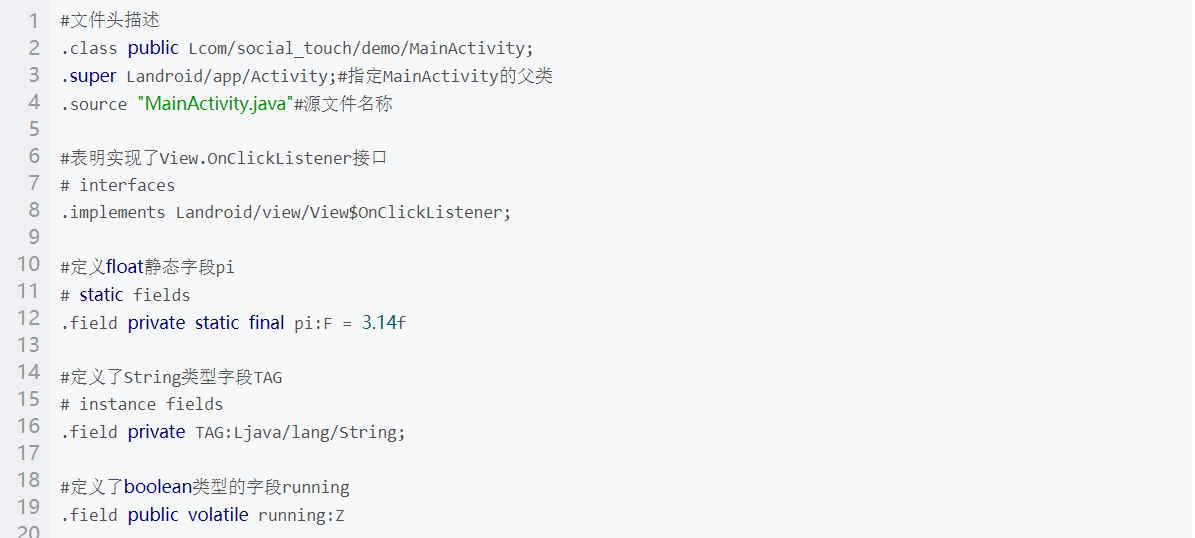






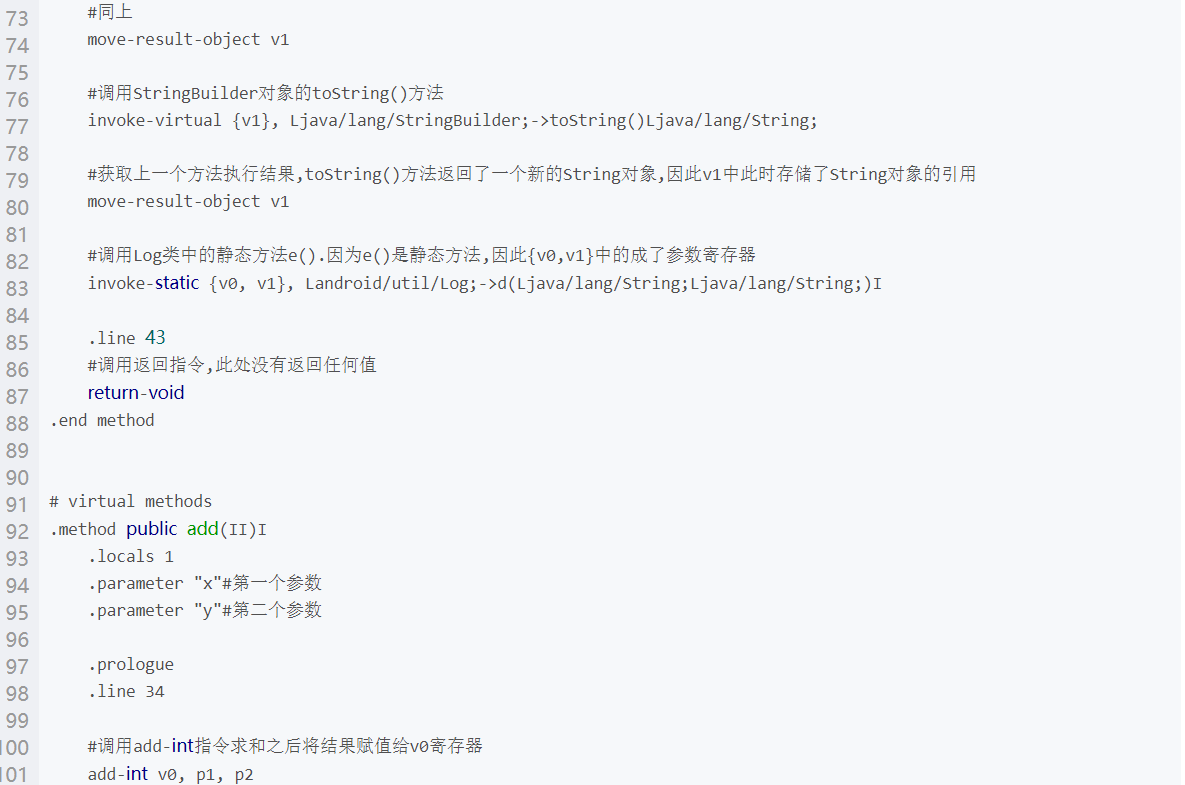


|  |
| --- |
| public class MainActivity extends Activity implements View.OnClickListener {  private String TAG = "MainActivity";  private static final float pi = (float) 3.14;  public volatile boolean running = false;  @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);  }  @Override  public void onClick(View view) {  int result = add(4, 5);  System.out.println(result);  result = sub(9, 3);  if (result > 4) {  log(result);  }  }  public int add(int x, int y) {  return x + y;  }  public synchronized int sub(int x, int y) {  return x + y;  }  public static void log(int result) {  Log.d("MainActivity", "the result:" + result);  }  } |

















## 参考链接

<https://blog.csdn.net/dd864140130/article/details/52076515>

# 4 ART虚拟机

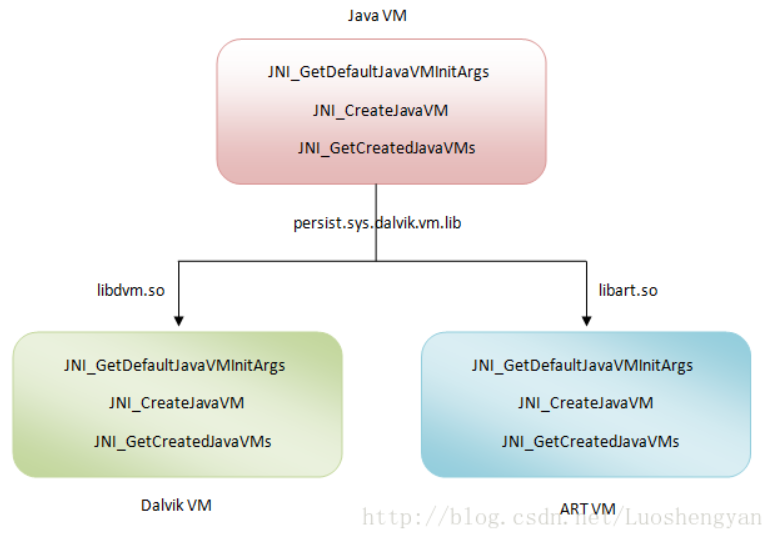
Dalvik虚拟机是android 4.4及之前使用的虚拟机。它使用的是JIT(just in time)技术来进行代码转译，每次执行应用的时候，Dalvik将程序的代码编译为机器语言执行。随着硬件水平的发展以及人民对更高性能的需求，Dalvik虚拟机的不足日益突出。

而应运而生的ART（Android Run Time）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。

## 4.1 Android ART运行时无缝替换Dalvik虚拟机的过程分析

我们知道，Dalvik虚拟机实则也算是一个Java虚拟机，只不过它执行的不是class文件，而是dex文件。因此，ART运行时最理想的方式也是实现为一个Java虚拟机的形式，这样就可以很容易地将Dalvik虚拟机替换掉。注意，我们这里说实现为Java虚拟机的形式，实际上是指提供一套完全与Java虚拟机兼容的接口。例如，Dalvik虚拟机在接口上与Java虚拟机是一致的，但是它的内部可以是完全不一样的东西。

实际上，ART运行时就是真的和Dalvik虚拟机一样，实现了一套完全兼容Java虚拟机的接口。为了方便描述，接下来我们就将ART运行时称为ART虚拟机，它和Dalvik虚拟机、Java虚拟机的关系如图1所示：



从图1可知道，Dalvik虚拟机和ART虚拟机都实现了三个用来抽象Java虚拟机的接口：

JNI\_GetDefaultJavaVMInitArgs -- 获取虚拟机的默认初始化参数

JNI\_CreateJavaVM -- 在进程中创建虚拟机实例

JNI\_GetCreatedJavaVMs -- 获取进程中创建的虚拟机实例

在Android系统中，Davik虚拟机实现在libdvm.so中，ART虚拟机实现在libart.so中。也就是说，libdvm.so和libart.so导出了JNI\_GetDefaultJavaVMInitArgs、JNI\_CreateJavaVM和JNI\_GetCreatedJavaVMs这三个接口，供外界调用。

**此外，Android系统还提供了一个系统属性persist.sys.dalvik.vm.lib，它的值要么等于libdvm.so，要么等于libart.so。**当等于libdvm.so时，就表示当前用的是Dalvik虚拟机，而当等于libart.so时，就表示当前用的是ART虚拟机。

以上描述的Dalvik虚拟机和ART虚拟机的共同之处，当然它们之间最显著还是不同之处。不同的地方就在于，Dalvik虚拟机执行的是dex字节码，ART虚拟机执行的是本地机器码。

那么，ART虚拟机执行的本地机器码是从哪里来的呢？Android的运行时从Dalvik虚拟机替换成ART虚拟机，并不要求开发者要将重新将自己的应用直接编译成目标机器码。也就是说，**开发者开发出的应用程序经过编译和打包之后，仍然是一个包含dex字节码的APK文件。**既然应用程序包含的仍然是dex字节码，而ART虚拟机需要的是本地机器码，这就必然要有一个翻译的过程。这个翻译的过程当然不能发生应用程序运行的时候，否则的话就和Dalvik虚拟机的JIT一样了。在计算机的世界里，与JIT相对的是AOT。AOT进Ahead-Of-Time的简称，它发生在程序运行之前。我们用静态语言（例如C/C++）来开发应用程序的时候，编译器直接就把它们翻译成目标机器码。这种静态语言的编译方式也是AOT的一种。但是前面我们提到，ART虚拟机并不要求开发者将自己的应用直接编译成目标机器码。这样，**将应用的dex字节码翻译成本地机器码的最恰当AOT时机就发生在应用安装的时候。**

我们知道，没有ART虚拟机之前，应用在安装的过程，其实也会执行一次“翻译”的过程。只不过这个“翻译”的过程是将dex字节码进行优化，也就是由dex文件生成odex文件。这个过程由安装服务PackageManagerService请求守护进程installd来执行的。从这个角度来说，**在应用安装的过程中将dex字节码翻译成本地机器码对原来的应用安装流程基本上就不会产生什么影响。**

我们接下来就从两个角度来了解ART虚拟机是如何做到无缝替换Dalvik虚拟机的：

1. ART虚拟机的启动过程；

2. Dex字节码翻译成本地机器码的过程。

**ART虚拟机的启动过程：**

我们知道，Android系统在启动的时候，会创建一个Zygote进程，充当应用程序进程孵化器。Zygote进程在启动的过程中，又会创建一个Dalvik虚拟机。

即然应用程序进程里面的Dalvik虚拟机都是从Zygote进程中复制过来的，那么接下来我们就继续Zygote进程是如何创建Dalvik虚拟机的。从Dalvik虚拟机的启动过程分析这篇文章可以知道，**Zygote进程中的Dalvik虚拟机是从AndroidRuntime::start这个函数开始创建**的。因此，接下来我们就看看这个函数的实现：



这个函数定义在文件frameworks/base/core/jni/AndroidRuntime.cpp中。

AndroidRuntime类的成员函数start最主要是做了以下三件事情：

1.创建一个JniInvocation实例，并且调用它的成员函数init来初始化JNI环境；

2.调用AndroidRuntime类的成员函数startVm来创建一个虚拟机及其对应的JNI接口，即创建一个JavaVM接口和一个JNIEnv接口；

3.有了上述的JavaVM接口和JNIEnv接口之后，就可以在Zygote进程中加载指定的class了。其中，第1件事情和第2件事情又是最关键的。因此，接下来我们继续分析它们所对应的函数的实现。

JniInvocation类的成员函数init所做的事情很简单。**它首先是读取系统属性persist.sys.dalvik.vm.lib的值。**前面提到，系统属性persist.sys.dalvik.vm.lib的值要么等于libdvm.so，要么等于libart.so。因此，接下来通过函数dlopen加载到进程来的要么是libdvm.so，要么是libart.so。**无论加载的是哪一个so，都要求它导出JNI\_GetDefaultJavaVMInitArgs、JNI\_CreateJavaVM和JNI\_GetCreatedJavaVMs这三个接口，**并且分别保存在JniInvocation类的三个成员变量JNI\_GetDefaultJavaVMInitArgs\_、JNI\_CreateJavaVM\_和JNI\_GetCreatedJavaVMs\_中。这三个接口也就是前面我们提到的用来抽象Java虚拟机的三个接口。

从这里就可以看出，**JniInvocation类的成员函数init实际上就是根据系统属性persist.sys.dalvik.vm.lib来初始化Dalvik虚拟机或者ART虚拟机环境。**

JniInvocation类的成员变量JNI\_CreateJavaVM\_指向的就是前面所加载的libdvm.so或者libart.so所导出的函数JNI\_CreateJavaVM，因此，JniInvocation类的成员函数JNI\_CreateJavaVM返回的JavaVM接口指向的要么是Dalvik虚拟机，要么是ART虚拟机。

通过上面的分析，我们就很容易知道，Android系统通过将ART运行时抽象成一个Java虚拟机，以及通过系统属性persist.sys.dalvik.vm.lib和一个适配层JniInvocation，就可以无缝地将Dalvik虚拟机替换为ART运行时。

**接下来我们再分析应用程序在安装过程中将dex字节码翻译为本地机器码的过程：**

Android系统通过PackageManagerService来安装APK，在安装的过程，PackageManagerService会通过另外一个类Installer的成员函数dexopt来对APK里面的dex字节码进行优化。

这个函数定义在文件frameworks/base/services/java/com/android/server/pm/Installer.java中。

Installer通过socket向守护进程installd发送一个dexopt请求，这个请求是由installd里面的函数dexopt来处理的。

函数dexopt首先是**读取系统属性persist.sys.dalvik.vm.lib的值**，接着在/data/dalvik-cache目录中**创建一个odex文件**。这个odex文件就是作为dex文件优化后的输出文件。再接下来，函数dexopt通过fork来创建一个子进程。如果系统属性persist.sys.dalvik.vm.lib的值等于libdvm.so，那么该子进程就会调用函数run\_dexopt来将dex文件优化成odex文件。另一方面，如果系统属性persist.sys.dalvik.vm.lib的值等于libart.so，那么该子进程就会调用函数run\_dex2oat来将dex文件翻译成oat文件，实际上就是将dex字节码翻译成本地机器码，并且保存在一个oat文件中。

函数run\_dexopt通过调用/system/bin/dexopt来对dex字节码进行优化，而函数run\_dex2oat通过调用/system/bin/dex2oat来将dex字节码翻译成本地机器码。注意，无论是对dex字节码进行优化，还是将dex字节码翻译成本地机器码，**最终得到的结果都是保存在相同名称的一个odex文件里面的**，但是前者对应的是一个dey文件（表示这是一个优化过的dex），后者对应的是一个oat文件（实际上是一个自定义的elf文件，里面包含的都是本地机器指令）。通过这种方式，原来任何通过绝对路径引用了该odex文件的代码就都不需要修改了。

通过上面的分析，我们就很容易知道，只需要将dex文件的优化过程替换成dex文件翻译成本地机器码的过程，就可以轻松地在应用安装过程，无缝地将Dalvik虚拟机替换成ART运行时。

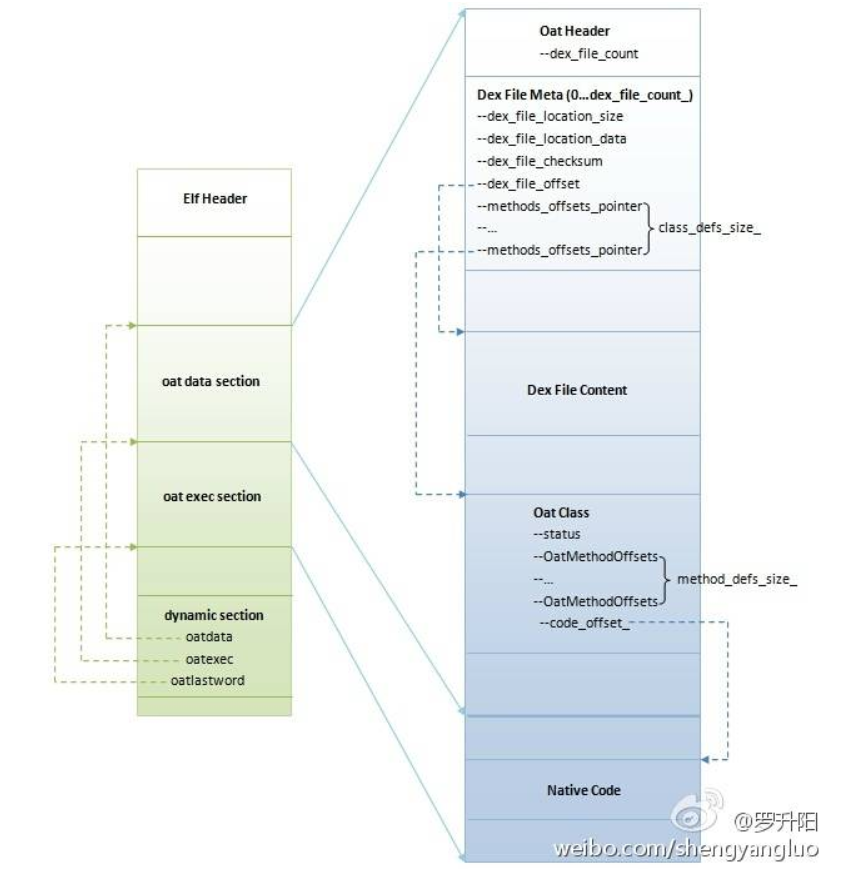
最后，还有一个地方需要注意的是，应用程序的安装发生在两个时机，第一个时机是系统启动的时候，第二个时机系统启动完成后用户自行安装的时候。在第一个时机中，系统除了会对/system/app和/data/app目录下的所有APK进行dex字节码到本地机器码的翻译之外，还会对/system/framework目录下的APK或者JAR文件，以及这些APK所引用的外部JAR，进行dex字节码到本地机器码的翻译。这样就可以保证除了应用之外，系统中使用Java来开发的系统服务，也会统一地从dex字节码翻译成本地机器码。也就是说，**将Android系统中的Dalvik虚拟机替换成ART运行时之后，系统中的代码都是由ART运行时来执行的了，这时候就不会对Dalvik虚拟机产生任何的依赖。**

## 4.2 Android运行时ART简要介绍

在Dalvik运行时中，APK在安装的时候，安装服务PackageManagerService会通过守护进程installd调用一个工具dexopt对打包在APK里面包含有Dex字节码的classes.dex进行优化，优化得到的文件保存在/data/dalvik-cache目录中，并且以.odex为后缀名，表示这是一个优化过的Dex文件。

在ART运行时中，APK在安装的时候，同样安装服务PackageManagerService会通过守护进程installd调用另外一个工具dex2oat对打包在APK里面包含有Dex字节码进翻译。这个翻译器实际上就是基于LLVM架构实现的一个编译器，它的前端是一个Dex语法分析器。翻译后得到的是一个**ELF格式的oat文件**，这个oat文件同样是以.odex后缀结束，并且也是保存在/data/dalvik-cache目录中。

ELF是Linux系统使用的一种文件格式，我们平时接触的静态库、动态库和可执行文件都是以这种格式保存的，但是由dex2oat工具生成的oat文件与上述三种文件都不一样，它有**两个特殊的段oatdata和oatexec**，分别用来储存原来打包在APK里面的dex文件和翻译这个dex文件里面的类方法得到本地机器指令，如图所示：



在oat文件的动态段（dymanic section）中，还导出了三个符号oatdata、oatexec和oatlastword，分别用来描述oatdata和oatexec段加段到内存后的起止地址。在oatdata段中，包含了两个重要的信息，一个信息是原来的classes.dex文件的完整内容，另一个信息引导ART找到classes.dex文件里面的类方法所对应的本地机器指令，这些本地机器指令就保存在oatexec段中。

举个例子说，我们在classes.dex文件中有一个类A，那么当我们知道类A的名字后，就可以通过保存在oatdata段的dex文件得到类A的所有信息，比如它的父类、成员变量和成员函数等。另一方面，类A在oatdata段中有一个对应的OatClass结构体。这个OatClass结构体描述了类A的每一个方法所对应的本地机器指令在oatexec段的位置。也就是说，当我们知道一个类及其某一个方法的名字（签名）之后，就可以通过oatdata段的dex文件内容和OatClass结构体找到其在oatexec段的本地机器指令，这样就可以执行这个类方法了。

**总结如下：**

1. 在Android系统启动过程中创建的Zygote进程利用ART运行时导出的Java虚拟机接口创建ART虚拟机。

2. APK在安装的时候，打包在里面的classes.dex文件会被工具dex2oat翻译成本地机器指令，最终得到一个ELF格式的oat文件。

3. APK运行时，上述生成的oat文件会被加载到内存中，并且ART虚拟机可以通过里面的oatdata和oatexec段找到任意一个类的方法对应的本地机器指令来执行。

## 参考链接

<https://www.kancloud.cn/alex_wsc/androids/473620>

<https://www.kancloud.cn/alex_wsc/androids/473621>