Android系统架构详解(2)--Android Runtime

时间 2014-03-18 16:46:01  [eoe博客](http://www.tuicool.com/sites/fuea2iY)

原文  [http://my.eoe.cn/1194759/archive/24027.html](http://my.eoe.cn/1194759/archive/24027.html?utm_source=tuicool&utm_medium=referral)

主题 [软件架构](http://www.tuicool.com/topics/11000148)[安卓开发](http://www.tuicool.com/topics/11080010)

**2. Android Runtime**   
现在再说AndroidRuntime和以前说的Android Runtime有所不同了，因为现在的AndroidRuntime是ART机制，是google android4.4新的运行机制。ART 的机制与 Dalvik 不同。在 Dalvik 下，应用每次运行的时候，字节码都需要通过即时编译器转换为机器码，这会拖慢应用的运行效率，而在 ART 环境中，应用在第一次安装的时候，字节码就会预先编译成机器码，使其成为真正的本地应用。这个过程叫做预编译（AOT,Ahead-Of-Time）。这样的话，应用的启动和执行都会变得更加快速。

开篇就说了这么多专业词是不是有点蒙了，说实话我自己也晕，因为我是应用开发，而且是半路转行做的开发，对计算机原理一类的东西认知度接近于0，先说说以前的Dalvik吧。知过去才能懂现在。

**(一) Dalvik**

关于Dalvik的文章写了好几天，本来已经总结了好几百字了，发现说的条例有点乱，后来无意中发现一位大侠写的相当好，但是太深了，所以我就直接将做了一个精简，重新整理了一下，原帖网址为 [***http://blog.csdn.net/andyxm/article/details/6126907***](http://blog.csdn.net/andyxm/article/details/6126907) ，文中大部分专业知识引用于此，有兴趣的童鞋可以自己做更深的研究。

我们通过四个问题来了解Dalvik：

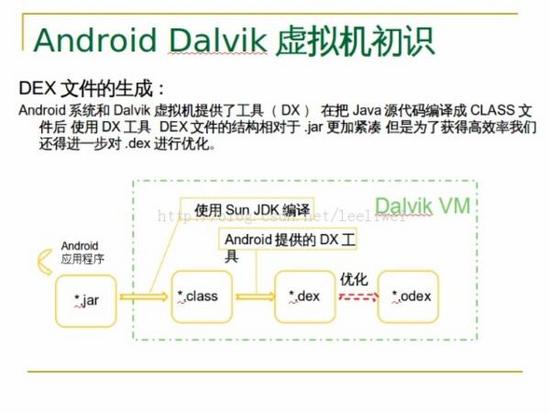
1. 什么是Dalvik虚拟机?
2. Dalvik VM与JVM有什么区别？
3. 有什么新的特点？
4. Dalvik VM的架构是怎么样的？前三个问题下文中做了总结和引导，这第四个问题。。。尼玛，这个架构看的我是头昏眼花，研究了好久，只明白了冰山一角。反正我不明白也就不随便乱说了，有兴趣的自己去原博客看吧。

1) 什么是Dalvik虚拟机?

首先，回顾一下什么是java虚拟机。Java虚拟机（JVM）是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。它有自己完善的硬件架构（如处理器、堆栈、寄存器等），还具有相应的指令系统。使用“Java虚拟机”程序就是为了支持与操作系统无关、在任何系统中都可以运行的程序。

因此，我们不妨对Dalvik虚拟机作出这样的描述：

**Dalvik虚拟机是Android程序的虚拟机，是Android中Java程序的运行基础** 。其指令集基于寄存器架构，执行其特有的文件格式——dex字节码来完成对象生命周期管理、堆栈管理、线程管理、安全异常管理、垃圾回收等重要功能。它的核心内容是实现库（libdvm.so），大体由C语言实现。依赖于Linux内核的一部分功能——线程机制、内存管理机制，能高效使用内存，并在低速CPU上表现出的高性能。每一个Android应用在底层都会对应一个独立的Dalvik虚拟机实例，其代码在虚拟机的解释下得以执行。   
[]()



2) Dalvik VM与JVM有什么区别？

Dalvik VM ≠Java VM

Dalvik 执行的是特有的DEX文件格式，而JVM运行的是\*.class文件格式。

dalvik基于寄存器，而JVM基于栈。

基于栈与基于寄存器的架构，谁更快？现在实际的处理器，大多都是基于寄存器的架构，从侧面反映出基于寄存器比基于栈的架构更与实际的处理器接近。但对于VM来说，源架构的求值栈或者寄存器都可能是用实际机器的内存来模拟的，所以性能特性与实际硬件又有不同。\*\*一般认为基于寄存器架构的Dalvik VM比基于栈架构JVM执行效率更高\*\*，原因是：虽然零地址指令更紧凑，但完成操作需要更多的load/store指令，也意味着更多的指令分派（instruction dispatch）次数与内存访问次数；访问内存是执行速度的一个重要瓶颈，二地址或三地址指令虽然每条指令占的空间较多，但总体来说可以用更少的指令完成操作，指令分派与内存访问次数都较少。

Dalvik优势：

A. 在编译时提前优化代码而不是等到运行时

B. 虚拟机很小，使用的空间也小；被设计来满足可高效运行多种虚拟机实例。

C. 常量池已被修改为只使用32位的索引，以简化解释器

3) Dalvik VM有什么新的特点？

**Dalvik VM的新特点可以概括为：一个应用，一个虚拟机实例，一个进程！！！**

每一个Android应用都运行在一个Dalvik虚拟机实例里，而每一个虚拟机实例都是一个独立的进程空间。每个进程之间可以通信（IPC，Binder机制实现）。虚拟机的线程机制，内存分配和管理，Mutex等等都是依赖底层操作系统而实现的。

不同的应用在不同的进程空间里运行，当一个虚拟机关闭或意外中止时不会对其它虚拟机造成影响，可以最大程度的保护应用的安全和独立运行。

有一张图，很牛X，总结的很精辟：



[]()

**(二) ART**

这是android4.4的新东西，还在研究呢，除了官网没有太多可参考的材料，加上时间有限，我就从百度百科里扒下来一部分内容略作修改，以后慢慢再做总结改进。

从 Android 4.4 开始，Google 开发者引进了新的Android 运行环境 ART（意思就是 Android Runtime。Android 官方页面的介绍中，也将其称作新的虚拟机），以替代旧的 Dalvik VM。它现在是实验选项，系统默认的运行环境仍然是 Dalvik。ART 会为 Android 带来怎样的改变？AndroidPolice 网站对此进行了分析。

ART 的机制与 Dalvik 不同。在 Dalvik 下，应用每次运行的时候，字节码都需要通过即时编译器转换为机器码，这会拖慢应用的运行效率，而在 ART 环境中，应用在第一次安装的时候，字节码就会预先编译成机器码，使其成为真正的本地应用。这个过程叫做预编译（AOT,Ahead-Of-Time）。这样的话，应用的启动和执行都会变得更加快速。

根据一些基准测试，新的运行环境能够使大多数应用的执行时间减半。这意味着，CPU 消耗大、运行时间长的应用能够更加快速地完成，而一般的应用也能更加流畅，比如动画效果更顺畅，触控反馈更加即时。在多核处理器的设备上，多数情况下只需激活少量的核心，或者能够更好的利用 ARM 的 big.LITTLE 架构。另外，它将会显著提升电池的续航能力以及系统的性能。

预编译也会带来一些缺点。一方面，机器码占用的存储空间更大。字节码变为机器码之后，可能会增加 10%-20%，不过在应用包中，可执行的代码常常只是一部分。比如最新的 Google+ APK 是 28.3 MB，但是代码只有 6.9 MB。另一方面，应用的安装时间会变长。至于延长多少时间，取决于应用本身，一些复杂的应用如 Facebook 和 Google+ 会让你等待更长时间。

总的来说，ART 的优点还是远远超越其缺点的，运行起来更有效率、耗电更少、占的内存也更低。毕竟，影响用户体验的要素中，电池续航和应用顺畅运行更为重要。我们仍然不知道 ART 何时能够替代 Dalvik，不过 AndroidPolice 网站说，Google 已经秘密开发了两年之久。Android 摆脱卡顿的希望，看来就是它了。

[Android运行时ART简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/39256813)

标签： [Android](http://www.csdn.net/tag/Android)[ART](http://www.csdn.net/tag/ART)[Dalvik](http://www.csdn.net/tag/Dalvik)[VM](http://www.csdn.net/tag/VM)

2014-09-15 00:58 40452人阅读 [评论](http://blog.csdn.net/luoshengyang/article/details/39256813#comments)(49) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/luoshengyang/article/details/39256813#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

Android（177） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

[**Android**](http://lib.csdn.net/base/android)在4.4就已推出新运行时ART，准备替代用了有些时日的Dalvik。不过当时尚属[**测试**](http://lib.csdn.net/base/softwaretest)版，主角仍是Dalvik。 直到今年的Google I/O大会，ART才正式取代Dalvik。这个消息在科技界引起不小轰动，也吸引不少技术人员对它的“技术分析”。可惜这些“技术分析”不过是引用了官方的数据和图表而已。这一系列文章将对ART进行真正的技术分析。老规矩，分析前先进行简要介绍和制定学习计划。

老罗的新浪微博：<http://weibo.com/shengyangluo>，欢迎关注！

**《Android系统源代码情景分析》一书正在进击的程序员网（**[**http://0xcc0xcd.com**](http://0xcc0xcd.com/)**）中连载，点击进入！**

        ART的发布之所以引起大家的关注，是因为Andoid与[**iOS**](http://lib.csdn.net/base/ios)相比，一直被人诟病它的流畅性。Android的流畅性问题，有一部分原因就归结于它的应用程序和部分系统服务是运行虚拟机之上的，也就是运行在Dalvik虚拟机之上，而iOS的应用程序和系统服务都是直接执行本地机器指令的。除了使用ART替换Dalvik之外，我们也应当看到，Android从3.0开始，就不遗余力地改进系统的流畅性。例如，3.0增加了对应用程序2D UI的硬件加速渲染，也就是GPU渲染。在此之前，应用程序的2D UI一直都是使用软件渲染，也就是CPU渲染。又如4.1通过Project Butter，在UI[**架构**](http://lib.csdn.net/base/architecture)中引入了VSYNC、Triple Buffer和HWComposer等技术，极大地提高UI的流畅性。

       ART之所以会比Dalvik快，是因为ART执行的是本地机器指令，而Dalvik执行的是Dex字节码，通过通过解释器执行。尽管Dalvik也会对频繁执行的代码进行JIT生成本地机器指令来执行，但毕竟在应用程序运行的过程中将Dex字节码翻译成本地机器机器指令也会影响到应用程序本身的执行，因此即使Dalvik使用了JIT，也在一定程度上也比不上直接就可以执行本地机器指令的运行时。

       在前面[Android ART运行时无缝替换Dalvik虚拟机的过程分析](http://blog.csdn.net/luoshengyang/article/details/18006645)一文中，我们提到，ART像Dalvik一样，都实现[**Java**](http://lib.csdn.net/base/javase)虚拟机接口，如图1所示：

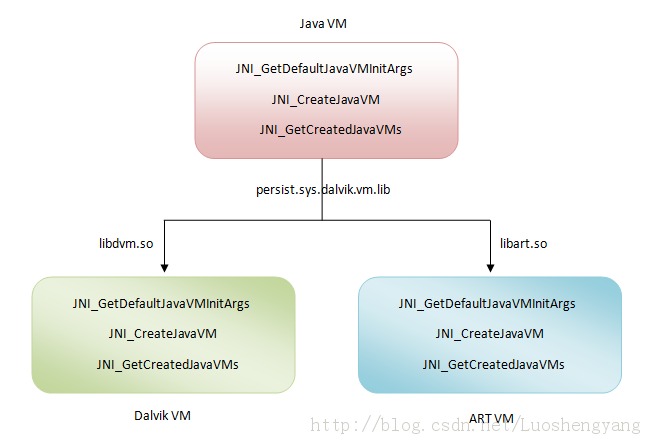


图1 Dalvik、ART和[**Java**](http://lib.csdn.net/base/java)VM的关系

        Zygote进程在启动的过程中，正是通过图1所示的接口创建Dalvik或者ART虚拟机的，这样看来，ART虽然执行的本地机器指令，但是它表面看来，又是一个不折不扣的虚拟机。也正是因为这样，ART才可以在不重新编译APK的基础上，直接可以加载和运行APK。这也是ART运行时可以无缝替换Dalvik运行时的原理。因此，我们就可以得出一个结论：ART是一个执行本地机器指令的虚拟机。这个结论似乎有点矛盾，既然是执行本地机器指令，为什么又称为虚拟机呢？从接下来的文章分析可以知道，ART除了实现Java虚拟机接口之外，其内部还有垃圾收集机制，同时还有Java核心类库调用，因此，随着对ART的深入分析，我们就认为这个结论是不矛盾的了。

        上面提到，ART才可以在不重新编译APK的基础上，直接对其进行加载和运行，这是由于APK在安装时被执行了AOT。AOT（Ahead Of Time）是相对JIT（Just In Time）而言的。也就是在APK运行之前，就对其包含的Dex字节码进行翻译，得到对应的本地机器指令，于是就可以在运行时直接执行了。这种技术不但使得我们可以不对原有的APK作任何修改，还可以使得这些APK只需要在安装时翻译一次，就可以无数次以本地机器指令的形式运行。这种技术与我们用C/C++语言编写一个程序，然后用GCC编译得到一个可执行程序，最后这个可执行程序就可以无数次地加载到系统执行，是差不多的。

        在ART中，打包在APK里面的Dex字节码是通过LLVM翻译成本地机器指令的。LLVM是一个用来快速开发自己的编译器的框架系统，关于它的介绍，可以参考它的作者之一[Chris Lattner](http://www.nondot.org/sabre/)写的这篇文章：<http://www.aosabook.org/en/llvm.html>。说起[Chris Lattner](http://www.nondot.org/sabre/)，他就是Apple今年发布的[**Swift**](http://lib.csdn.net/base/swift)语言的首席架构师啊，所以我们就可以感受到LLVM有多强大了。总体来说，基于LLVM架构开发的编译器的执行过程如图2所示：



图2 基于LLVM架构开发的编译器执行过程

       其中，前端（Frontend）对输入的源代码（Source Code）进行语法分析后，生成一棵抽象语法树（Abstract Syntax Tree，AST），并且可以进一步将得到的抽象语法树转化一种称为LLVM IR的中间语言。LLVM IR是一种与编程语言无关的中间语言，也就是说，不管是[**C语言**](http://lib.csdn.net/base/c)，还是Fortran、Ada语言编写的源文件，经过语法分析后，最终都可以得到一个对应的LLVM IR文件。这个LLVM IR文件可以作为后面的优化器（Optimizer）和后端（Backend）的输入文件。优化器对LLVM IR文件进行优化，例如消除代码里面的冗余计算，以提到最终生成的代码的执行效率。后端负责生成最终的机器指令。

       LLVM的上述架构大大简化开发编译器的流程，因为开发者需要关注的仅仅是前端，然后就可以利用现成的优化器来进行代码优化，并且利用现成的后端生成各种体系结构相关的机器指令，如图3所示：

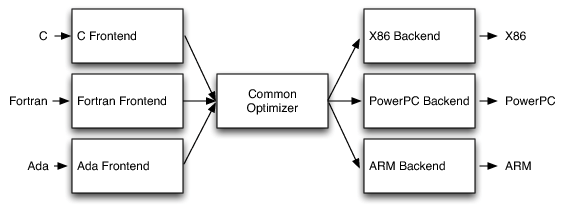


图3 利用现成的与语言无关的优化器和后端为语言相关的前端生成各种体系结构相关的机器指令

        在图3中，我们分别为C、Fortran和Ada三种语言开发三个不同的前端，然后利用现成的优化器对它们生成的LLVM IR语言进行优化，并且通过现成的后端生成X86、PowerPC和ARM三种不同体系结构的机器指令。

        如果我们没有忘记，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**](http://lib.csdn.net/base/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4所示：

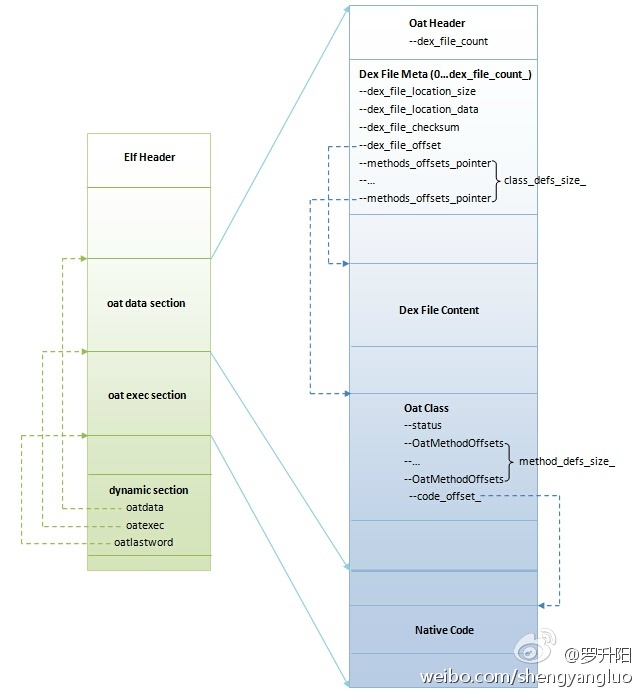


图4 ART翻译classes.dex后得到的ELF格式的oat文件

       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。

        通过上面的分析，我们就将ART的运行原理都简要地介绍了，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。

        对于第1点，ART虚拟机的创建过程中，可以参考前面[Android ART运行时无缝替换Dalvik虚拟机的过程分析](http://blog.csdn.net/luoshengyang/article/details/18006645)一文。

        对于第2点，APK里面的Dex字节码被dex2oat工具翻译生本地机器指令的过程，掌握它需要有扎实的编译知识。由于知识、能力、时间有限，因此这一部分的内容我们就略过不分析了，但是这将不会影响我们对ART运行时的理解。

        对于第3点，是我们理解ART运行时的关键所在。以ART虚拟机的启动过程为例，从前面[Android ART运行时无缝替换Dalvik虚拟机的过程分析](http://blog.csdn.net/luoshengyang/article/details/18006645)一文可以知道，在AndroidRuntime类的成员函数start中，ART虚拟机创建和初始化完成后，Zygote进程就会通过它导出的JNI接口CallStaticVoidMethod使得它以指定的类方法为入口正式进入运行状态，如下所示：

**[cpp]** [view plain](http://blog.csdn.net/luoshengyang/article/details/39256813) [copy](http://blog.csdn.net/luoshengyang/article/details/39256813)

[在CODE上查看代码片](https://code.csdn.net/snippets/489287)

1. **void** AndroidRuntime::start(**const** **char**\* className, **const** **char**\* options)
2. {
3. ......
5. /\* start the virtual machine \*/
6. JniInvocation jni\_invocation;
7. jni\_invocation.Init(NULL);
8. JNIEnv\* env;
9. **if** (startVm(&mJavaVM, &env) != 0) {
10. **return**;
11. }
13. ......
15. /\*
16. \* Start VM.  This thread becomes the main thread of the VM, and will
17. \* not return until the VM exits.
18. \*/
19. **char**\* slashClassName = toSlashClassName(className);
20. jclass startClass = env->FindClass(slashClassName);
21. **if** (startClass == NULL) {
22. ALOGE("JavaVM unable to locate class '%s'\n", slashClassName);
23. /\* keep going \*/
24. } **else** {
25. jmethodID startMeth = env->GetStaticMethodID(startClass, "main",
26. "([Ljava/lang/String;)V");
27. **if** (startMeth == NULL) {
28. ALOGE("JavaVM unable to find main() in '%s'\n", className);
29. /\* keep going \*/
30. } **else** {
31. env->CallStaticVoidMethod(startClass, startMeth, strArray);
33. #if 0
34. **if** (env->ExceptionCheck())
35. threadExitUncaughtException(env);
36. #endif
37. }
38. }
40. ......
41. }

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

       在AndroidRuntime类的成员函数start中，参数className的值等于“com.android.internal.os.ZygoteInit”，本地变量env是从调用另外一个成员函数startVm创建的ART虚拟机获得的JNI接口。函数的目标就是要找到一个名称为com.android.internal.os.ZygoteInit的类，以及它的静态成员函数main，然后就以这个函数为入口，开始运行ART虚拟机。为此，函数执行了以下步骤：

       1. 调用JNI接口FindClass加载com.android.internal.os.ZygoteInit类。

       2. 调用JNI接口GetStaticMethodID找到com.android.internal.os.ZygoteInit类的静态成员函数main。

       3. 调用JNI接口CallStaticVoidMethod开始执行com.android.internal.os.ZygoteInit类的静态成员函数main。

       注意，在第3步中，要执行的是CallStaticVoidMethod开始执行com.android.internal.os.ZygoteInit类的静态成员函数main的本地机器指令，而不是Dex字节码。这样就引出以下三个关键问题：

       1. ART如何找到com.android.internal.os.ZygoteInit类？

       2. ART如何找到com.android.internal.os.ZygoteInit类的静态成员函数main？

       3. ART如何找到com.android.internal.os.ZygoteInit类的静态成员函数main的本地机器指令？

       解决上述三个问题所需要的信息都存在于dex2oat工具生成的oat文件中。因此，在接下来的文章中，我们将通过分析dex2oat工具生成的oat文件来回答上述三个问题，使得我们可以更加好地理解ART的工作原理。

       如上所述，由于ART在查找类方法时，需要用到保存在oat文件的oatdata段的原dex文件内容，实质上就是要对dex文件进行解析，以获得相关的信息。这与Dalvik虚拟机在dex文件中查找类和方法信息的过程是一样的。这意味着要理解ART运行时，必须先要理解Dalvik虚拟机。Dalvik虚拟机的相关知识可以参考[Dalvik虚拟机简要介绍和学习计划](http://blog.csdn.net/luoshengyang/article/details/8852432)这个系列的文章。这告诉我们一个道理，旧的知识并没有过时，它对我们学习新的知识是有帮助的，有时候甚至是必须的。所以大家就不要觉得最前面写的那些基于Android 2.3版本的文章是没有用的了。我们在学习一样新东西的时候，无论是新的知识，还是旧的知识，对我们理解它的原理，都是很有帮助的！

        我们除了要以dex2oat工具生成的oat文件作为切入点来分析ART运行时之外，还会结合ART运行时的垃圾收集机制来说明ART运行时与Dalvik虚拟机一样也是一个虚拟机，以此加深对ART运行时的理解。与Dalvik虚拟机的垃圾收集机制相比，ART运行时的垃圾收集机制更为复杂，由此带来的垃圾收集效率也更高。因此我们在分析ART运行时的垃圾收集机制之前，先会分析Dalvik虚拟机的垃圾收集机制。一方面是有利于我们循序渐进、由简而繁地讲解ART运行时的垃圾收集原理，另一方面也方便我们对比ART运行时和Dalvik虚拟机的垃圾收集机制有哪些不同，从而可以更好地理解为ART运行时的垃圾收集效率更高。

        综上所述，接下来我们就按照以下几个情景来分析ART的工作原理：

        1. [ART加载oat文件的过程](http://blog.csdn.net/luoshengyang/article/details/39307813)。

        2. [ART查找类和方法的过程](http://blog.csdn.net/luoshengyang/article/details/39533503)。

        3. [ART查找类方法的本地机器指令的过程](http://blog.csdn.net/luoshengyang/article/details/40289405)。

        4. [Dalvik虚拟机的垃圾收集过程](http://blog.csdn.net/luoshengyang/article/details/41338251)。

        5. [ART的垃圾收集过程](http://blog.csdn.net/luoshengyang/article/details/42072975)。

        以上情景将基于Android 4.4源码进行分析，一方面是因为Android L版本的源码还没有放出来，另一方面是相信万变不离其宗，即使Androd L版本的ART实现有变化，但是基本的原理还是一样的。敬请关注！同时更多的信息