# QA

-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

# 基本概念

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）

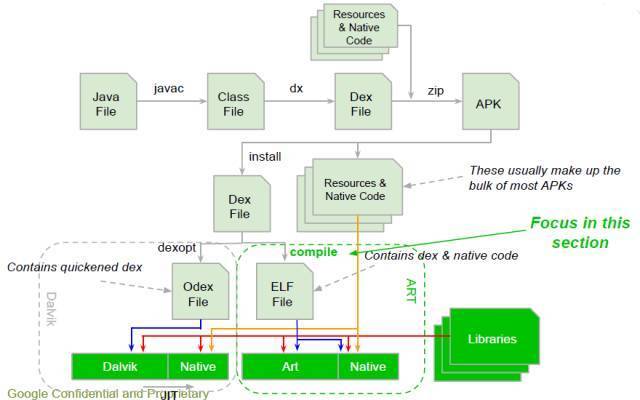
## Android 系统启动原理

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。

作者：点融黑帮

链接：https://www.jianshu.com/p/1f779586efdc

來源：简书

简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

式的转载都请联系作者获得授权并注明出处。

# 出厂设置

<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

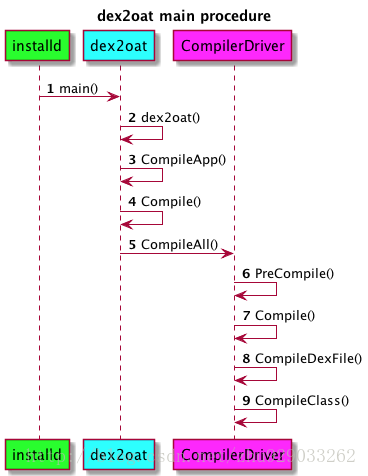
第12章 JIT编译

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

# 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