Android逆向 | JVM、DVM(Dalvik VM)和ART虚拟机对比

evan_man 咸鱼学Python

作者：**evan_man**
来源：**https://blog.csdn.net/evan_man/article/details/52414390**

Android 系统使用 Dalvik Virtual Machine (DVM) 作为其虚拟机，所有安卓程序都运行在安卓系统进程里，每个进程对应着一个 Dalvik 虚拟机实例。

他们都提供了对象生命周期管理、堆栈管理、线程管理、安全和异常管理以及垃圾回收等重要功能，各自拥有一套完整的指令系统。

Android 之所以不直接使用 JVM 作为其虚拟机的原因有很多，版权问题我们暂且搁置一边，本文将首先在技术上对 DVM 和 JVM 进行比较，然后重点对 Dalvik 虚拟机的垃圾回收机制进行介绍，文章末尾再对Android5.0之后使用的新型虚拟机—— ART 虚拟机进行简单介绍。

DVM vs JVM

共同点：

- 都是解释执行
- 都是每个 OS 进程运行一个 VM，并运行一个单独的程序
- 在较新版本中（Froyo / Sun JDK 1.5）都实现了相当程度的 JIT compiler（即时编译） 用于提速。
- JIT（Just In Time，即时编译技术）对于热代码（使用频率高的字节码）直接转换成汇编代码；

不同点：

- dvm 执行的是 .dex 格式文件，jvm执行的是.class文件。class文件和dex之间可以相互转换具体流程如下图，多个class 文件转变成一个 dex 文件会引发一些问题，具体如下：
  - 方法数受限：多个 class 文件变成一个 dex 文件所带来的问题就是方法数超过 65535 时报错，由此引出 MultiDex 技术，具体资料同学可以 google 下。
  - class 文件去冗余：class 文件存在很多的冗余信息， dex 工具会去除冗余信息(多个 class 中的字符串常量合并为一个，比如对于 Ljava/lang/Oject 字符常量，每个 class 文件基本都有该字符常量，存在很大的冗余)，并把所有的 .class 文件整合到 .dex 文件中。减少了 I/O 操作，提高了类的查找速度。
- 许多 GC 实现都是在对象开头的地方留一小块空间给 GC 标记用。Dalvik VM 则不同，在进行 GC 的时候会单独申请一块空间，以位图的形式来保存整个堆上的对象的标记，在 GC 结束后就释放该空间。（关于这一点后面的 Dalvik 垃圾回收机制还会更加深入的介绍）
- dvm 是基于寄存器的虚拟机 而 jvm 执行是基于虚拟栈的虚拟机。这类的不同是最要命的，因为它将导致一系列的问题，具体如下：
  - dvm 速度快！寄存器存取速度比栈快的多，dvm 可以根据硬件实现最大的优化，比较适合移动设备。JAVA 虚拟机基于栈结构，程序在运行时虚拟机需要频繁的从栈上读取写入数据，这个过程需要更多的指令分派与内存访问次数，会耗费很多 CPU 时间。
  - 指令数小！dvm 基于寄存器，所以它的指令是二地址和三地址混合，指令中指明了操作数的地址；jvm 基于栈，它的指令是零地址，指令的操作数对象默认是操作数栈中的几个位置。这样带来的结果就是 dvm 的指令数相对于jvm 的指令数会小很多， jvm 需要多条指令而dvm可能只需要一条指令。
  - jvm 基于栈带来的好处是可以做的足够简单，真正的跨平台，保证在低硬件条件下能够正常运行。而 dvm 操作平台一般指明是 ARM 系统，所以采取的策略有所不同。需要注意的是 dvm 基于寄存器，但是这也是个映射关系，如果硬件没有足够的寄存器， dvm 将多出来的寄存器映射到内存中。

Dalvik虚拟机

谈到垃圾回收自然而然的想到了堆， Dalvik 的堆结构相对于 JVM 的堆结构有所区别，这主要体现在 Dalvik 将堆分成了Active 堆和 Zygote 堆。

这里大家只要知道 Zygote 堆是 Zygote 进程在启动的时候预加载的类、资源和对象(具体 gygote 进程预加载了哪些类，详见文末的附录)，除此之外的所有对象都是存储在 Active 堆中的。

对于为何要将堆分成 gygote 和 Active 堆，这主要是因为Android通过fork方法创建到一个新的 gygote 进程，为了尽可能的避免父进程和子进程之间的数据拷贝。

fork 方法使用写时拷贝技术，写时拷贝技术简单讲就是 fork 的时候不立即拷贝父进程的数据到子进程中，而是在子进程或者父进程对内存进行写操作时是才对内存内容进行复制， Dalvik 的 gygote 堆存放的预加载的类都是 Android核心类和 java 运行时库，这部分内容很少被修改，大多数情况父进程和子进程共享这块内存区域。

通常垃圾回收重点对 Active 堆进行回收操作，Dalvik 为了对堆进行更好的管理创建了一个 Card Table 、两个 Heap Bitmap 和一个 Mark Stack数据结构。

Dalvik创建对象流程

当 Dalvik 虚拟机的解释器遇到一个 new 指令时，它就会调用函数`Object* dvmAllocObject(ClassObject* clazz, int flags)`。期间完成的动作有(注意：Java 堆分配内存前后，要对 Java 堆进行加锁和解锁，避免多个线程同时对 Java 堆进行操作。下面所说的堆指的是 Active 堆)：

1. 调用函数dvmHeapSourceAlloc在Java堆上分配指定大小的内存，成功则返回，否则下一步。
2. 执行一次GC， GC执行完毕后，再次调用函数dvmHeapSourceAlloc在Java堆上分配指定大小的内存，成功则返回，否则下一步。
3. 首先将堆的当前大小设置为Dalvik虚拟机启动时指定的Java堆最大值，然后进行内存分配，成功返回失败下一步。这里调用的函数是 dvmHeapSourceAllocAndGrow
4. 调用函数gcForMalloc来执行GC，这里的GC和第二步的GC，区别在于这里回收软引用对象引用的对象，如果还是失败抛出OOM异常。这里调用的函数是dvmHeapSourceAllocAndGrow

Dalvik回收对象流程

Dalvik 的垃圾回收策略默认是标记擦除回收算法，即 Mark 和 Sweep 两个阶段。标记与清理的回收算法一个明显的区别就是会产生大量的垃圾碎片，因此程序中应该避免有大量不连续小碎片的时候分配大对象，同时为了解决碎片问题，Dalvik 虚拟机通过使用 dlmalloc 技术解决，关于后者读者另行 google 。下面我们对 Mark 阶段进行简单介绍。

Mark 阶段使用了两个 Bitmap 来描述堆的对象，一个称为 Live Bitmap ，另一个称为 Mark Bitmap 。Live Bitmap 用来标记上一次 GC 时被引用的对象，也就是没有被回收的对象，而 Mark Bitmap 用来标记当前 GC 有被引用的对象。当 Live Bitmap 被标记为 1 ，但是在 Mark Bitmap 中标记为 0 的对象表明该对象需要被回收。此外在 Mark 阶段往往要求其它线程处于停止状态，因此 Mark 又分为并行和串行两种方式，并行的 Mark 分为两个阶段：

1)、只标记 gc_root 对象，即在 GC 开始的瞬间被全局变量、栈变量、寄存器等所引用的对象，该阶段不允许垃圾回收线程之外的线程处于运行状态。

2)、有条件的并行运行其它线程，使用 Card Table 记录在垃圾收集过程中对象的引用情况。整个 Mark 阶段都是通过 Mark Stack 来实现递归检查被引用的对象，即在当前 GC 中存活的对象。标记过程类似用一个栈把第一阶段得到的 gc_root 放入栈底，然后依次遍历它们所引用的对象(通过出栈入栈)，即用栈数据结构实现了对每个 gc_root 的递归。

Dalvik 的 GC 类型共有四种

- GC_CONCURRENT: 表示是在已分配内存达到一定量之后触发的 GC。
- GC_FOR_MALLOC: 表示是在堆上分配对象时内存不足触发的 GC。
- GC_BEFORE_OOM: 表示是在准备抛 OOM 异常之前进行的最后努力而触发的 GC。
- GC_EXPLICIT: 表示是应用程序调用 System.gc、VMRuntime.gc 接口或者收到 SIGUSR1 信号时触发的 GC。

其中 GC_FOR_MALLOC、GC_CONCURRENT 和 GC_BEFORE_OOM 三种类型的 GC 都是在分配对象的过程触发的。

垃圾回收具体都是通过调用函数`void dvmCollectGarbageInternal(const GcSpec* spec)`来执行垃圾回收，该函数的参数GcSpec结构体定义见本文的附录。

对于函数 dvmCollectGarbageInternal 的内部逻辑，即垃圾回收流程，根据垃圾回收线程和工作线程的关系分为并行 GC 和非并行 GC 。

前者在回收阶段有选择性的停止当前工作线程，后者在垃圾回收阶段停止所有工作线程。

但是并行 GC 需要多执行一次标记根集对象以及递归标记那些在 GC 过程被访问了的对象的操作，意味着并行 GC 需要花费更多的 CPU 资源。`dvmCollectGarbageInternal`函数的内部逻辑如下：（本文末尾的附录中给出了一个对应的流程图）

1. 调用函数 dvmSuspendAllThreads 挂起所有的线程，以免它们干扰 GC 。
   - 这里如何挂起其它线程呢？其实就是每个线程在运行过程中会周期性的检测自身的一个标志位，通过这个标志位我们可以告知线程停止运行。
2. 调用函数 dvmHeapBeginMarkStep 初始化 Mark Stack ，并且设定好 GC 范围。
   - Mark Stack其实是一个object指针数组
3. 调用函数dvmHeapMarkRootSet标记根集对象。
   - Mark的第一阶段，主要分为两大类：
     - 1) Dalvik虚拟机内部使用的全局对象（ 维护在一个 hash 表中);
     - 2) 应用程序正在使用的对象（维护在一个调用栈中）
4. 调用函数dvmClearCardTable清理Card Table。（只在并行gc发生）
   - Card Table记录记录在Zygote堆上分配的对象在垃圾收集执行过程中对在Active堆上分配的对象的引用。
5. 调用函数 dvmUnlock 解锁堆。这个是针对调用函数dvmCollectGarbageInternal执行GC前的堆锁定操作。（只在并行gc发生）
6. 调用函数dvmResumeAllThreads唤醒第1步挂起的线程。（只在并行gc发生）
   - 此时非gc线程可以开始工作，这部分线程对堆的操作记录在CardTable上面，gc则进行Mark的第二阶段
7. 用函数 dvmHeapScanMarkedObjects 从第 3 步获得的根集对象开始，归递标记所有被根集对象引用的对象。
8. 调用函数 dvmLockHeap 重新锁定堆。这个是针对前面第5步的操作。（只在并行gc发生）
9. 调用函数 dvmSuspendAllThreads 重新挂起所有的线程。这个是针对前面第6步的操作。（只在并行gc发生）
   - 这里需要再次停止工作线程，用来解决前面线程对堆的少部分的操作，这个过程很快。
10. 用函数 dvmHeapReMarkRootSet 更新根集对象。因为有可能在第 4 步到第 6 步的执行过程中，有线程创建了新的根集对象。（只在并行gc发生）
11. 调用函数 dvmHeapReScanMarkedObjects 归递标记那些在第4步到第6步的执行过程中被修改的对象。这些对象记录在 Card Table 中。（只在并行gc发生）
12. 调用函数 dvmHeapProcessReferences 处理那些被软引用（Soft Reference）、弱引用（Weak Reference）和影子引用（Phantom Reference）引用的对象，以及重写了 finalize 方法的对象。这些对象都是需要特殊处理的。
13. 调用函数 dvmHeapSweepSystemWeaks 回收系统内部使用的那些被弱引用引用的对象。
14. 调用函数 dvmHeapSourceSwapBitmaps 交换 Live Bitmap 和 Mark Bitmap 。
    - 执行了前面的 13 步之后，所有还被引用的对象在Mark Bitmap中的bit都被设置为1。
    - Live Bitmap记录的是当前GC前还被引用着的对象。
    - 通过交换这两个 Bitmap，就可以使得当前 GC 完成之后，使得 Live Bitmap 记录的是下次 GC 前还被引用着的对象。
15. 调用函数 dvmUnlock 解锁堆。这个是针对前面第8步的操作。（只在并行gc发生）
16. 调用函数 dvmResumeAllThreads 唤醒第9步挂起的线程。（只在并行gc发生）
17. 调用函数 dvmHeapSweepUnmarkedObjects 回收那些没有被引用的对象。没有被引用的对象就是那些在执行第 14 步之前，在 Live Bitmap 中的 bit 设置为 1 ，但是在 Mark Bitmap 中的 bit 设置为 0 的对象。
18. 调用函数 dvmHeapFinishMarkStep 重置 Mark Bitmap 以及 Mark Stack 。这个是针对前面第2步的操作。
19. 调用函数 dvmLockHeap 重新锁定堆。这个是针对前面第 15 步的操作。（只在并行gc发生）
20. 调用函数 dvmHeapSourceGrowForUtilization 根据设置的堆目标利用率调整堆的大小。
21. 调用函数 dvmBroadcastCond 唤醒那些等待 GC 执行完成再在堆上分配对象的线程。（只在并行gc发生）

22. 调用函数 dvmResumeAllThreads 唤醒第 1 步挂起的线程。（只在非并行gc发生）

23. 调用函数 dvmEnqueueClearedReferences 将那些目标对象已经被回收了的引用对象增加到相应的Java队列中去，以便应用程序可以知道哪些引用引用的对象已经被回收了。

总结

通过上面的流程分析，我们知道了并行和串行gc的区别在于：

1. 并行gc会在mark第二阶段将非gc线程唤醒；当mark的第二阶段完成之后，再次停止非gc线程；利用cardtable的信息再次进行一个mark操作，此时的mark操作比第一个mark操作要快得多。
2. 并行gc会在sweep阶段将非gc线程唤醒。
3. 串行gc会在垃圾回收开始就暂停所有非gc线程，知道垃圾回收结束。
4. 并行gc涉及到两次的mark操作，消耗cpu时间。

ART虚拟机

在Android5.0中，ART取代了Dalvik虚拟机（安卓在4.4中发布了ART）。ART虚拟机直接执行本地机器码；而Dalvik虚拟机运行的是DEX字节码需要通过解释器执行。

安卓运行时从Dalvik虚拟机替换成ART虚拟机，并不要求开发者重新将自己的应用直接编译成目标机器码，应用程序仍然是一个包含dex字节码的apk文件，这主要得益于AOT技术，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。

ART应用安装的时候把dex中的字节码将被编译成本地机器码，之后每次打开应用，执行的都是本地机器码。去除了运行时的解释执行，效率更高，启动更快。

ART运行时内部使用的Java堆的主要组成包括Image Space、Zygote Space、Allocation Space和Large Object Space四个Space，两个Mod Union Table，一个Card Table，两个Heap Bitmap，两个Object Map（Live 和 Mark Object Map），以及三个Object Stack （Live、Mark、Allocation Stack）。具体结构图参考附录。

Image Space 和 Zygote Space 之间，隔着一段用来映射 system@framework@boot.art[1]@classes.oat 文件的内存。

system@framework@boot.art[2]@classes.oat 是一个OAT文件，它是由在系统启动类路径中的所有DEX文件翻译得到的，Image Space 映射的是一个 `system@framework@boot.art@classes.dex` 文件，这个文件保存的是在生成system@framework@boot.art[3]@classes.oat 这个OAT文件的时候需要预加载的类对象，这些需要预加载的类由 `/system/framework/framework.jar` 文件里面的preloaded-classes文件指定。

以后只要系统启动类路径中的 DEX 文件不发生变化（即不发生更新升级），那么以后每次系统启动只需要将文件 `system@framework@boot.art@classes.dex` 直接映射到内存即可。

由于 `system@framework@boot.art@classes.dex` 文件保存的是一些预先创建的对象，并且这些对象之间可能会互相引用，因此我们必须保证 `system@framework@boot.art@classes.dex` 文件每次加载到内存的地址都是固定的。

这个固定的地址保存在`system@framework@boot.art@classes.dex`文件开头的一个 Image Header 中。此外，`system@framework@boot.art@classes.dex`文件也依赖于`system@framework@boot.art@classes.oat`文件，因此也会将后者固定加载到Image Space的末尾。

Image Space是不能分配新对象的。Image Space和Zygote Space在Zygote进程和应用程序进程之间进行共享，而Allocation Space是每个进程都独立地拥有一份。

ART的运行原理

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。
   - oat文件中的oatdata包含用来生成本地机器指令的dex文件内容
   - oat文件中的oatexec包含有生成的本地机器指令。

注意：

这里将 DEX 文件中的类和方法称之为 DEX 类和 DEX 方法，将 OTA 中的类和方法称之为 OTA 类和 OTA 方法，ART 运行时将类和方法称之为 Class 和 ArtMethod 。

ART中一个已经加载的 Class 对象包含了一系列的 ArtField 对象和 ArtMethod 对象，其中，ArtField 对象用来描述成员变量信息，而 ArtMethod 用来描述成员函数信息。对于每一个 ArtMethod 对象，它都有一个解释器入口点和一个本地机器指令入口点。

ART找到一个类和方法的流程：

- 在DEX文件中找到目标DEX类的编号，并且以这个编号为索引，在OAT文件中找到对应的OAT类。
- 在DEX文件中找到目标DEX方法的编号，并且以这个编号为索引，在上一步找到的OAT类中找到对应的OAT方法。
- 使用上一步找到的OAT方法的成员变量begin_和code_offset_，计算出该方法对应的本地机器指令。

上面的流程对应给出了流程图，具体内容参考附录。

ART运行时对象的创建过程：

可以分配内存的Space有三个：Zygote Space、Allocation Space和Large Object Space。不过，Zygote Space在还没有划分出Allocation Space之前，就在Zygote Space上分配，而当Zygote Space划分出Allocation Space之后，就只能在Allocation Space上分配。因此实际上应用运行的时候能够分配内存也就Allocation 和 Large Object Space两个。

而分配的对象究竟是存入上面的哪个Space呢？满足如下三个条件的内存，存入Large Object Space：1）Zygote Space已经划分除了Allocation Space，2）分配对象是原子类型数组，如int[] byte[] boolean[], 3）分配的内存大小大于一定的门限值。

对于分配对象时内存不足的问题，是通过垃圾回收和在允许范围内增长堆大小解决的。由于垃圾回收会影响程序，因此ART运行时采用力度从小到大的进垃圾回收策略。一旦力度小的垃圾回收执行过后能满足分配要求，那就不需要进行力度大的垃圾回收了。这跟dalvik虚拟机的对象分配策略也是类似的。

ART垃圾回收流程

并行GC流程图如下：

1. 调用子类实现的成员函数 InitializePhase 执行 GC 初始化阶段。
2. 获取用于访问 Java 堆的锁。
3. 调用子类实现的成员函数 MarkingPhase 执行 GC 并行标记阶段。
4. 释放用于访问 Java 堆的锁。
5. 挂起所有的 ART 运行时线程。
6. 调用子类实现的成员函数 HandleDirtyObjectsPhase 处理在 GC 并行标记阶段被修改的对象。
7. 恢复第4步挂起的 ART 运行时线程。
8. 重复第 5 到第 7 步，直到所有在 GC 并行阶段被修改的对象都处理完成。
9. 获取用于访问 Java 堆的锁。
10. 调用子类实现的成员函数 ReclaimPhase 执行 GC 回收阶段。
11. 释放用于访问 Java 堆的锁。
12. 调用子类实现的成员函数 FinishPhase 执行 GC 结束阶段

非并行GC流程图如下：

1. 调用子类实现的成员函数 InitializePhase 执行 GC 初始化阶段。
2. 挂起所有的 ART 运行时线程。
3. 调用子类实现的成员函数 MarkingPhase 执行 GC 标记阶段。
4. 调用子类实现的成员函数 ReclaimPhase 执行 GC 回收阶段。
5. 恢复第2步挂起的 ART 运行时线程。
6. 调用子类实现的成员函数 FinishPhase 执行 GC 结束阶段

通过两者的对比可以得出如下结论(与Dalvik大同小异)

- 非并行GC在垃圾回收的整个过程中暂停了所有非gc线程
- 并行GC在一开始只是对堆进行加锁，对于那些暂时并不会在堆中分配的内存的线程不起作用，它们依然可以运行，但是会造成对象的引用发生变化，但是这段时间的引用发生的变化被记录了下来。之后系统会停止所有线程，对上面记录的数据进行处理，然后唤起所有线程，系统进入垃圾回收阶段。

附录

Gygote堆预加载的类

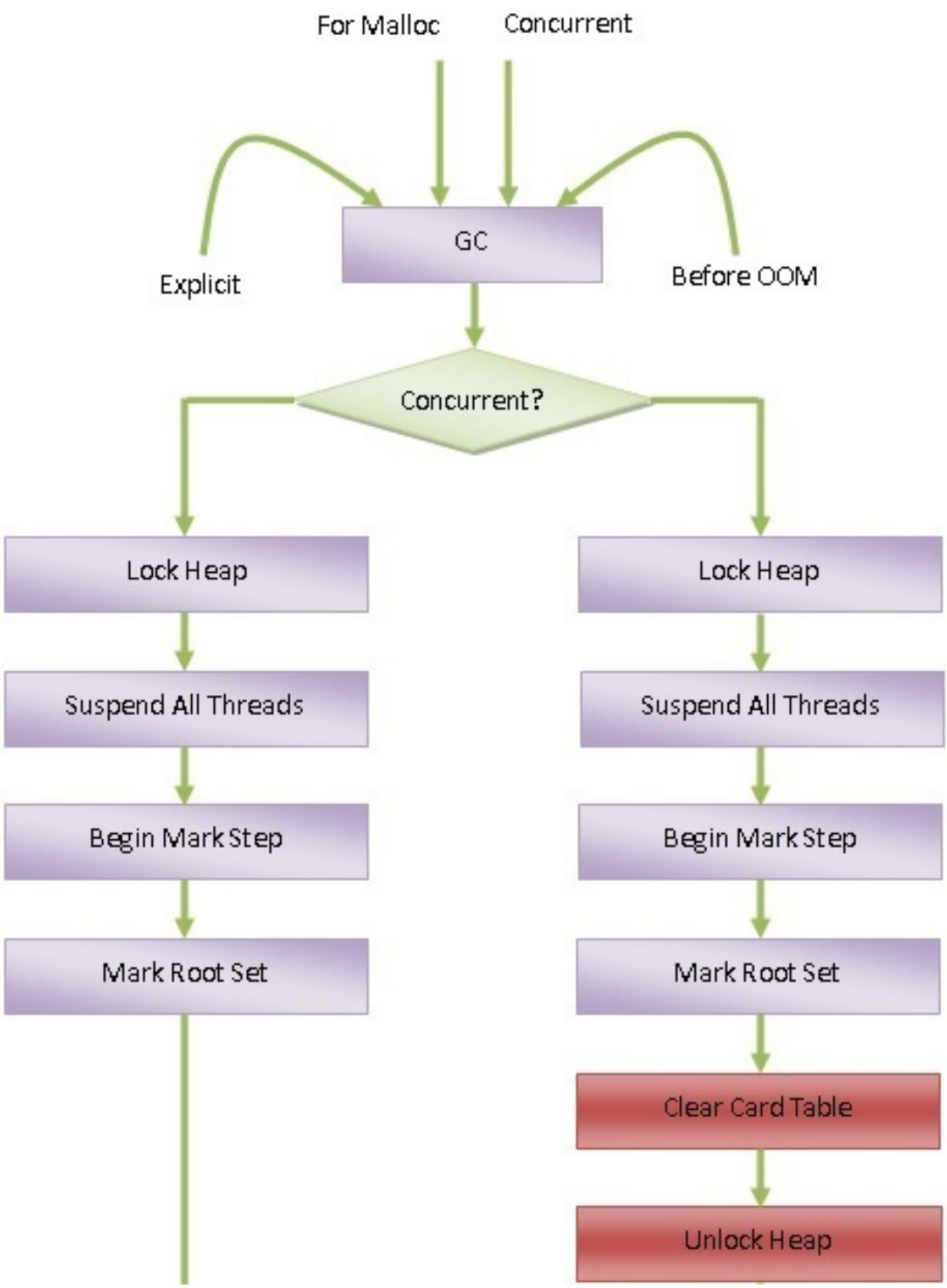
该文件[4]所指明的类就是通常gygote进程在创建时预加载的类，基本上囊括Android开发中大部分使用到的类，如View、Activity以及java运行时库等都会进行预加载。

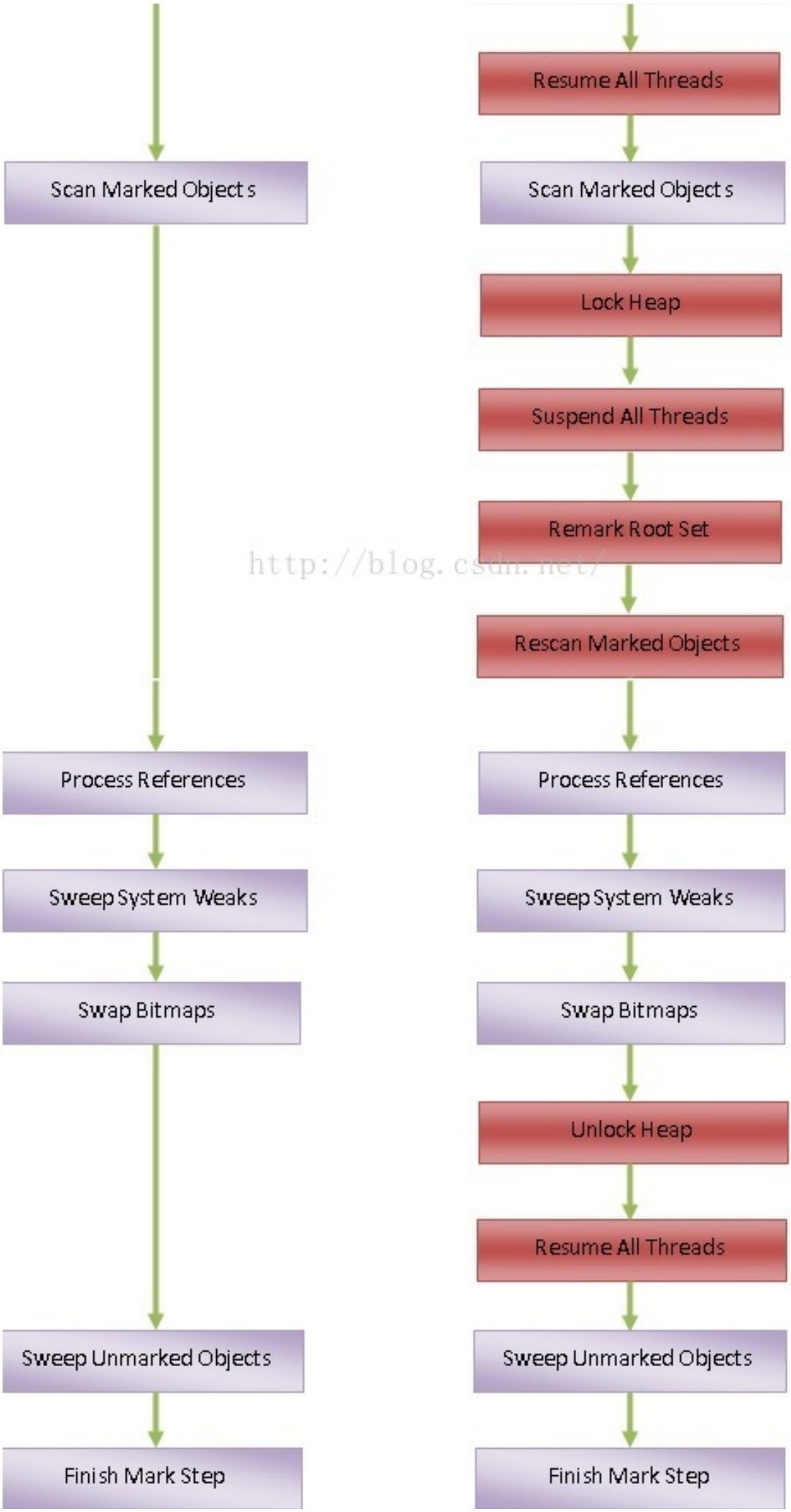
Dalvik对应的GC类型结构体定义如下：

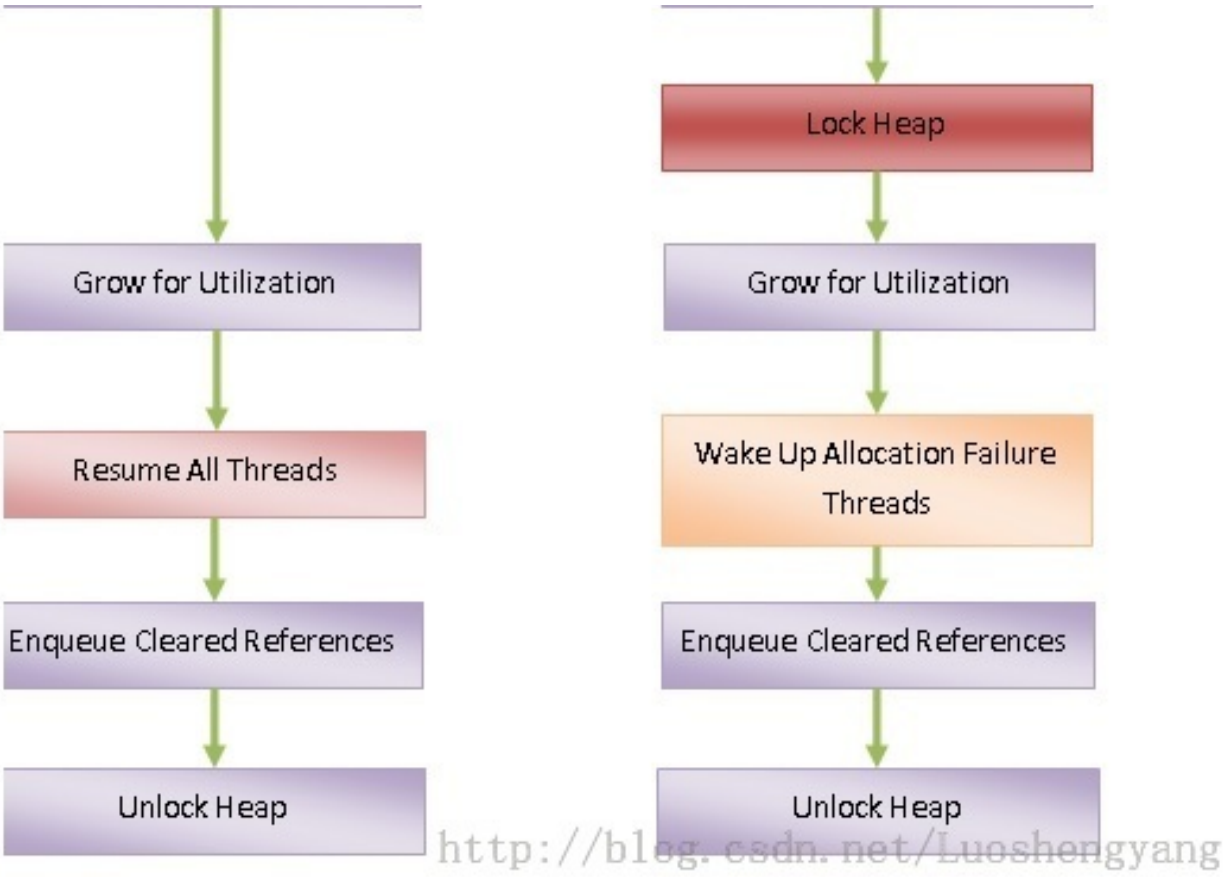
```
1
```

```
struct GcSpec {
/* If true, only the application heap is threatened. */
bool isPartial;
/* If true, the trace is run concurrently with the mutator. */
bool isConcurrent;
/* Toggles for the soft reference clearing policy. */
bool doPreserve;
/* A name for this garbage collection mode. */
const char *reason;
};
```

下图就是根据**Dalvik**回收阶段调用的`dvmCollectGarbageInternal()`函数所得到的流程图

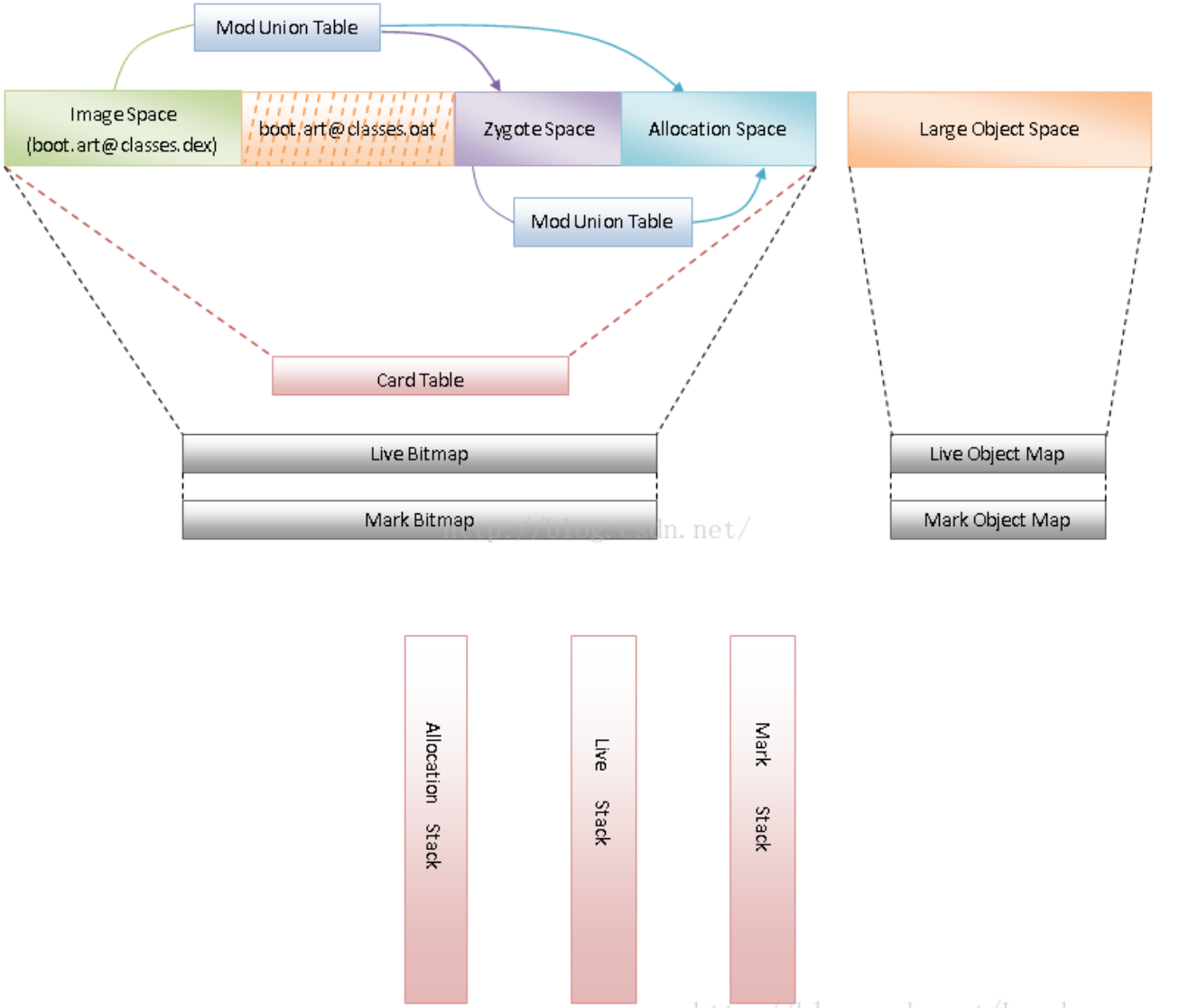






dvmCollectGarbageInternal函数针对并行和串行两种gc的流程图

下图是ART的堆结构图



ART的堆结构

Mod Union Table对象

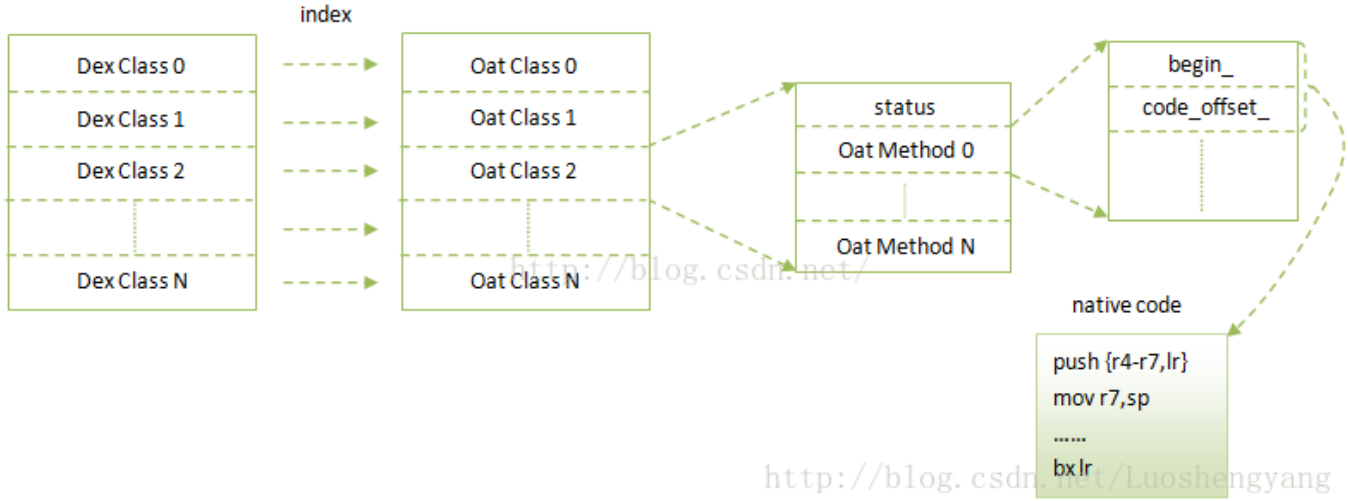
- 一个用来记录在GC并行阶段在Image Space上分配的对象对在Zygote Space和Allocation Space上分配的对象的引用。
- 另一个用来记录在GC并行阶段在Zygote Space上分配的对象对在Allocation Space上分配的对象的引用。

Allocation Stack：用来记录上一次GC后分配的对象，用来实现类型为Sticky的Mark Sweep Collector。

Live Stack：配合allocation_stack_一起使用，用来实现类型为Sticky的Mark Sweep Collector。

Mark Stack：用来在GC过程中实现递归对象标记

ART找到一个类和方法的流程



在OAT文件中查找类方法的本地机器指令的过程

我们从左往右来看上图。

先是根据类签名信息从包含在OAT文件里面的DEX文件中查找目标Class的编号，然后再根据这个编号找到在OAT文件中找到对应的OatClass。

接下来再根据方法签名从包含在OAT文件里面的DEX文件中查找目标方法的编号，然后再根据这个编号在前面找到的OatClass中找到对应的OatMethod。

有了这个OatMethod之后，我们就根据它的成员变量 begin 和 code_offset_ 找到目标类方法的本地机器指令了。其中，从DEX文件中根据签名找到类和方法的编号要求对DEX文件进行解析，这就需要利用Dalvik虚拟机的知识了。

参考资料

[1]http://blog.csdn.net/luoshengyang/article/details/41338251

[2]http://blog.csdn.net/luoshengyang/article/details/39256813

References

[1] framework@boot.art: *mailto:framework@boot.art*
[2] framework@boot.art: *mailto:framework@boot.art*
[3] framework@boot.art: *mailto:framework@boot.art*
[4] 该文件: *https://github.com/android/platform_frameworks_base/blob/master/preloaded-classes*

Love & Share

[完]

咸鱼学Python

微信扫描二维码，关注我的公众号

朕已阅