#### JVM工作原理

我们写的java文件,经过编译生成.class文件,然后java虚拟机经过 类加载 就成了.class类,也就是我们运行时访问的XXX.class类。

JVM是运行在系统之上的,也就是说,我们写的代码,运行在JVM中,而JVM又运行在CPU里,而CPU只认识机器码,那么我们写的代码是怎么跑在CPU中呢?其实很简单,JVM底层会将字节码转换为机器码,然后运行在CPU内.也可以这么理解:**JVM就是个转换器,他将我们写的代码转换为CPU可以识别的代码,然后运行在CPU内**.

不同平台有不同的JVM,所以我们写一套代码,就能转换成不同平台的机器码,也就可以运行在不同平台上,这就是java跨平台的原理.

DVM工作原理

DVM是Android虚拟机的一个版本,主要工作在Android4.4之前. 因为在Android4.4时,Google就引入了ART,而在Android5.0之后, Google就将Android系统虚拟机彻底切换为ART.

也就是说: Android4.4之前,系统是虚拟机DVM,Android5.0之后,系统虚拟机ART.而在Android4.4之后,Android5.0之前这段期间,是两者并存的.

那么,为什么要将DVM替换掉呢? 因为DVM效率低!

前面我们说到,CPU只认识机器码,DVM的工作原理很简单,它会在app启动后,我们执行到对应功能的时候,就将这部分功能对应的代码 转换为 机器码,保存起来然后执行,可以理解为:用到才转换,所以也被称为JIT(just in time).

这样有个优点: 节省内存!

但是,这样有个缺点: app执行速度变慢了,就像"懒汉式"单例一样,省了内存,但是要的时候,就需要临时去创建对象,等于延迟了程序的执行速度.同样的道理,这里要执行功能的时候,才去转换,肯定也延迟了app的执行速度,这也是Android4.4之前的系统卡顿的原因之一.

那么,原因之二呢?

原因之二就是DVM的垃圾回收机制比较差.

ART工作原理

ART是Android5.0之后彻底生效的,它主要有两个改善的地方.

1 将转换为机器码的过程提前到了安装apk的时候.

2 内存分配方式和垃圾回收机制做了极大的优化.

ART是在app安装的时候,就提前将所有代码转换为机器码保存下来,等到执行的时候,直接取出来在CPU中执行,也就是说,ART将转换为机器码这件事提前了. 所以叫做AOT(ahead of time).类似于饿汉式单例.

缺点也很明显,就是耗费内存了,一次加载整个app所有功能的代码 并 转换为机器码,明显费内存.

JVM&DVM&ART的关系

其实我们的app是跑在Android虚拟机上的(DVM或ART),跟JVM毛关系都没有.

Android虚拟机可以看成是JVM的魔改版,它们的类加载方式,垃圾回收策略基本都是一样的,并且Android虚拟机的一些知识是基于JVM的,所以我们需要学习JVM的相关知识.

我们知道,JVM接收的是.class文件,我们写的每一个.java文件,都会被编译为一个单独的.class文件,然后经过类加载器去加载.

这个方式有两个缺点:

1 不同的.java文件之间,会有一些相同的地方,分别编译为.class文件也会有相同的地方,这样的重复就有点浪费空间.

2 加载A.class文件的时候,发现它依赖了B.class,就需要将B.class也加载到内存,而加载到内存是一个IO操作,这样就需要两次IO操作,效率低.

基于空间和效率考虑,Android就提出了.dex文件, .dex文件可以看成是多个.class文件的集合, 也就是说,一个.dex文件可以是多个.class文件的集合, 在.class文件合并为.dex文件的过程中,会把重复的内容删除只留一份,这样就节省了空间.

我们的Android虚拟机就是执行.dex文件的,因为一个.dex文件包含了多个.class文件,所以当我们加载A.class文件时,如果需要B.class文件,就不需要再加载一次了,因为它俩本来就在同一个.dex文件中,这样就减少了IO次数,提高了效率.

现在,我们的代码转换过程就是这样的: .java -> .class -> .dex -> 机器码.

这里有个点:当我们将多个.class文件合并为.dex文件的时候,就出现方法数超过65535的问题, 这是因为合并的.class文件太多了,大家一般都会通过添加multidex来解决.

我们知道,JVM是基于栈的,所以JVM指令一般都比较短,因为只能对栈顶元素进行操作,而Android虚拟机是基于寄存器的,所以指令可以很长,因为可以指定操作数.

那么,它门有什么区别呢.区别有两点:

1 基于栈的指令集普遍短,执行同样的功能需要的指令多,需要频繁出栈入栈,所以效率低,但是可移植性更强.

2 基于寄存器的指令集可以指定操作数,所以可以很长,执行相同的功能需要的指令更少,效率更高,但是移植性差.

我们知道,Java的强大之一就在于良好的可移植性,这是因为不同平台有不同的JVM,并且它们都是基于栈的,所以可以共用一套JVM指令集. 而Android只需要运行在Android平台上就行了,对移植性要求不高,所以可以采用效率更高的基于寄存器的指令集.

综上,我们可以归纳如下:

1 Android虚拟机是基于Java虚拟机的魔改版.

2 Android应用是跑在Android虚拟机里面的.

3 Android虚拟机的指令集和文件格式跟JVM都不同.

CMS的GC过程

前面说到,DVM的卡顿原因之一是因为JIT,那么原因之二呢,是因为垃圾回收机制不太好,那么它是怎么回收的呢? 我们先来看下Java虚拟机CMS(Concurrent Mark Sweep)的回收方式.

CMS的回收方式可以分为四大步骤:

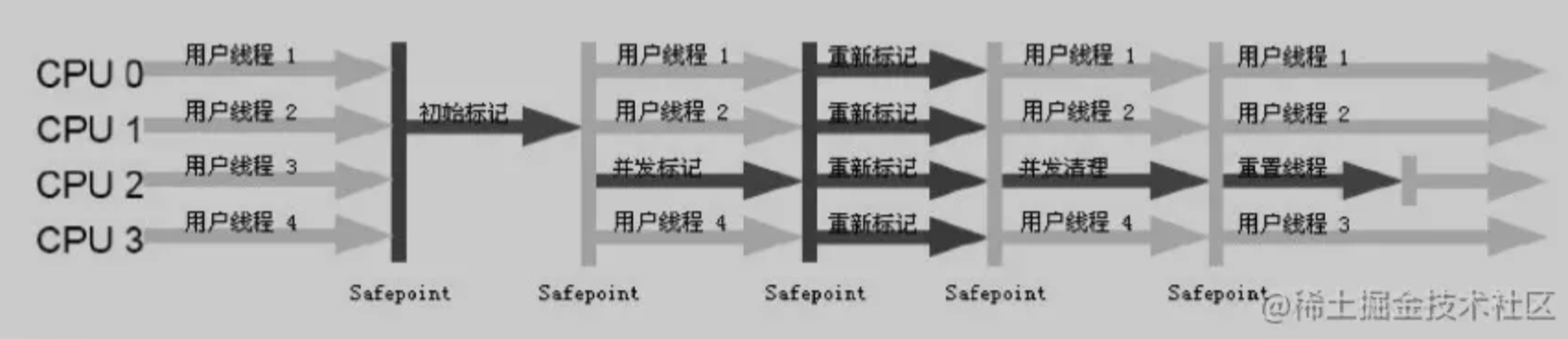
初始标记: 只标记GC Roots直接关联的对象.需要停止所有线程(Stop The World),但是速度很快.

并发标记: 进行GC Roots遍历的过程,速度很慢,但是 是和用户线程并发执行的.

重新标记: 收集浮动垃圾对象,需要Stop The World,比初试标记时间长,但是远比并发标记时间短.

并发清除: 清除所有标记的垃圾对象,时间长,但是是和用户线程并发执行的.

过程如下图



通过上述过程我们知道,整个GC过程有两次停顿,分别是"初始标记"和"重新标记"阶段,但是速度很快,对用户的影响不大.而两次耗时比较长的"并发标记"和"并发清除",因为是和用户线程并发执行的,所以用户是无感知的,我们可以归纳为:并发收集,低停顿.

DVM的GC过程

我们前面说过,Andrid虚拟机是基于JVM的魔改版本,它们中的很多地方都是通用的. DVM和JVM中的CMS的GC过程是类似的.标记过程中也有两次停顿,这是无法避免的.

那么为什么Android上就卡顿明显呢? 因为Android是移动端设备,移动端本来内存就小,更容易触发GC,就会明显感觉卡顿.

其次,在DVM中,堆空间被分为两部分:Zygote堆和Active堆.

我们知道,Android系统启动的第一个进程是init进程,然后由init进程创建出我们的Zygote进程,再然后由Zygote孵化出system\_server进程,Android是基于CS模式的系统,其中system\_server进程就是我们的S端,我们用的ActivityManagerService,WindowManagerService等XXXService都是在这个进程中的.

当我们的Zygote进程孵化第一个应用进程的之前,就会将已经使用的那部分堆划分出来,称为Zygote堆,将剩下的堆划分为另一部分,称为Active堆.后续无论是Zygote进程还是应用程序进程,要分配对象的时候,都在Active堆上进行.这样就可以减少对Zygote堆的写操作.

那么为什么要减少对Zygote进程的写操作呢?

很简单,因为Zygote进程在fork应用进程的时候,它们是使用同一个堆的,都是zygote堆,但是,如果一旦需要对该堆进行写操作时,就会将堆内容拷贝一份,Zygote进程和应用程序进程分别各拿一份,也就是说:对Zygote堆写操作会引起复制,这个叫做写时复制(COW),所以,为了减少复制操作,需要尽量少的对Zygote堆进行写操作.

那么,为什么要复制呢,所有应用程序共享一个堆空间不就行了?不行! 如果所有应用程序共享一个堆空间,那么如果其中一个应用程序爆炸式的写数据,就会导致OOM,顺便就连累了其它的应用程序.所以一定要拆开.

那为什么要写时才复制呢,fork的时候直接复制不就行了?不行!因为有的app可能就访问下数据,永远都不写数据,也就没有复制的必要,你提前复制了就是多此一举,白浪费时间和精力.

好,现在我们知道Android虚拟机是主要是往Active堆里面写对象,而且采用的是跟CMS一样的"标记-清除"法来执行垃圾回收,那么,内存碎片是避免不了的. 尤其是在分配大对象时.这就变相增加了回收频率,从而导致app卡顿.

ART的GC过程

ART的整体回收策略跟DVM类似,但是ART只需要停顿一次.

首先,ART的初始标记和并发标记阶段,都是并发执行的,可以理解为只有并发标记这一个阶段.

其次,ART在并发标记过程中,如果有新的对象,则会分配到一个叫做Allocation stack的空间里. 这样,就避免了浮动垃圾.

最后,ART再进行一次整体停顿,去扫描Allocaton stack中分配的对象. 然后再执行清除操作.

这样,ART通过新增一个Allocation stack空间来保存临时新增对象,就避免了多次停顿.这是典型的空间换时间.

而且,ART将Active堆分的更细, ART开辟了一块非连续的、离散的堆空间,叫做Large Object Space,专门用来放大对象.

这样,每次存放大对象时,都会在这里进行分配,而这个是离散的、非连续的,也就不存在内存碎片,从而变相的降低了GC频率,提高了应用性能,这属于粒度细化的思想.

总结

本文只站在宏观角度来说明Java虚拟机和Android虚拟机的异同点,不深入到细节问题,我们可以归纳如下:

1 Android虚拟机是基于Java虚拟机演变的,很多思想都是通用的,只不过针对移动端做了一些处理.

2 Android虚拟机接收的是.dex文件,是基于寄存器的;java虚拟机接收的是.class文件,是基于栈的.

3 Dalvik是基于JIT实现的,省内存但是效率低;ART是基于AOT实现的,费内存但是效率高.

4 Dalvik的回收过程跟CMS类似,标记过程需要进行两次Stop world,并且内存碎片问题严重,导致分配大对象时频繁触发GC.

5 ART的回收过程也跟CMS类似(可选),标记过程只需要进行一次Stop world,而且引入了Large Object Space,解决了内存碎片导致的分配大对象频繁GC的问题,变相提高了GC效率.