1、JVM内存回收机制

1.1 回收算法

- 标记回收算法（Mark and Sweep GC）

从"GC Roots"集合开始，将内存整个遍历一次，保留所有可以被GC Roots直接或间接引用到的对象，而剩下的对象都当作垃圾对待并回收，这个算法需要中断进程内其它组件的执行并且可能产生内存碎片

- 复制算法 (Copying）

将现有的内存空间分为两快，每次只使用其中一块，在垃圾回收时将正在使用的内存中的存活对象复制到未被使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收。

- 标记-压缩算法 (Mark-Compact)

先需要从根节点开始对所有可达对象做一次标记，但之后，它并不简单地清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。这种方法既避免了碎片的产生，又不需要两块相同的内存空间，因此，其性价比比较高。

- 分代

将所有的新建对象都放入称为年轻代的内存区域，年轻代的特点是对象会很快回收，因此，在年轻代就选择效率较高的复制算法。当一个对象经过几次回收后依然存活，对象就会被放入称为老生代的内存空间。对于新生代适用于复制算法，而对于老年代则采取标记-压缩算法。

1.2 复制和标记-压缩算法的区别

乍一看这两个算法似乎并没有多大的区别，都是标记了然后挪到另外的内存地址进行回收，那为什么不同的分代要使用不同的回收算法呢?

其实2者最大的区别在于前者是用空间换时间后者则是用时间换空间。

前者的在工作的时候是不没有独立的“mark”与“copy”阶段的，而是合在一起做一个动作，就叫scavenge（或evacuate，或者就叫copy）。也就是说，每发现一个这次收集中尚未访问过的活对象就直接copy到新地方，同时设置forwarding pointer。这样的工作方式就需要多一份空间。

后者在工作的时候则需要分别的mark与compact阶段，mark阶段用来发现并标记所有活的对象，然后compact阶段才移动对象来达到compact的目的。如果compact方式是sliding compaction，则在mark之后就可以按顺序一个个对象“滑动”到空间的某一侧。因为已经先遍历了整个空间里的对象图，知道所有的活对象了，所以移动的时候就可以在同一个空间内而不需要多一份空间。

所以新生代的回收会更快一点，老年代的回收则会需要更长时间，同时压缩阶段是会暂停应用的，所以给我

们应该尽量避免对象出现在老年代。

2、Dalvik虚拟机

2.1 java堆

Java堆实际上是由一个Active堆和一个Zygote堆组成的，其中，Zygote堆用来管理Zygote进程在启动过程中预加载和创建的各种对象，而Active堆是在Zygote进程fork第一个子进程之前创建的。以后启动的所有应用程序进程是被Zygote进程fork出来的，并都持有一个自己的Dalvik虚拟机。在创建应用程序的过程中，Dalvik虚拟机采用COW策略复制Zygote进程的地址空间。

COW策略：一开始的时候（未复制Zygote进程的地址空间的时候），应用程序进程和Zygote进程共享了同一个用来分配对象的堆。当Zygote进程或者应用程序进程对该堆进行写操作时，内核就会执行真正的拷贝操作，使得Zygote进程和应用程序进程分别拥有自己的一份拷贝，这就是所谓的COW。因为copy是十分耗时的，所以必须尽量避免copy或者尽量少的copy。

为了实现这个目的，当创建第一个应用程序进程时，会将已经使用了的那部分堆内存划分为一部分，还没有使用的堆内存划分为另外一部分。前者就称为Zygote堆，后者就称为Active堆。这样只需把zygote堆中的内容复制给应用程序进程就可以了。以后无论是Zygote进程，还是应用程序进程，当它们需要分配对象的时候，都在Active堆上进行。这样就可以使得Zygote堆尽可能少地被执行写操作，因而就可以减少执行写时拷贝的操作。在Zygote堆里面分配的对象其实主要就是Zygote进程在启动过程中预加载的类、资源和对象了。这意味着这些预加载的类、资源和对象可以在Zygote进程和应用程序进程中做到长期共享。这样既能减少拷贝操作，还能减少对内存的需求。

2.2 和GC有关的一些指标

记得我们之前在优化魅族某手机的gc卡顿问题时，发现他很容易触发GC*FOR*MALLOC，这个GC类别后续会说到，是分配对象内存不足时导致的。可是我们又设置了很大的堆Size为什么还会内存不够呢，这里需要了解以下几个概念：分别是Java堆的起始大小（Starting Size）、最大值（Maximum Size）和增长上限值（Growth Limit）。

在启动Dalvik虚拟机的时候，我们可以分别通过-Xms、-Xmx和-XX:HeapGrowthLimit三个选项来指定上述三个值，以上三个值分别表示表示

Starting Size : Dalvik虚拟机启动的时候，会先分配一块初始的堆内存给虚拟机使用。

Growth Limit:是系统给每一个程序的最大堆上限,超过这个上限，程序就会OOM

Maximum Size：不受控情况下的最大堆内存大小，起始就是我们在用largeheap属性的时候，可以从系统获取的最大堆大小

同时除了上面的这个三个指标外，还有几个指标也是值得我们关注的，那就是堆最小空闲值（Min Free）、堆最大空闲值（Max Free）和堆目标利用率（Target Utilization）。假设在某一次GC之后，存活对象占用内

存的大小为LiveSize，那么这时候堆的理想大小应该为(LiveSize / U)。但是(LiveSize / U)必须大于等于(LiveSize + MinFree)并且小于等于(LiveSize + MaxFree)，每次GC后垃圾回收器都会尽量让堆的利用率往目标利用率靠拢。所以当我们尝试手动去生成一些几百K的对象，试图去扩大可用堆大小的时候，反而会导致频繁的GC，因为这些对象的分配会导致GC，而GC后会让堆内存回到合适的比例，而我们使用的局部变量很快会被回收理论上存活对象还是那么多，我们的堆大小也会缩减回来无法达到扩充的目的。 与此同时这也是产生CONCURRENT GC的一个因素，后文我们会详细讲到。

2.3 GC的类型

- GC*FOR*MALLOC: 表示是在堆上分配对象时内存不足触发的GC。
- GC_CONCURRENT: 当我们应用程序的堆内存达到一定量，或者可以理解为快要满的时候，系统会自动触发GC操作来释放内存。
- 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。
- 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。

实际上，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。而并发和非并发GC的区别主要在于前者在GC过程中，有条件地挂起和唤醒非GC线程，而后者在执行GC的过程中，一直都是挂起非GC线程的。并行GC通过有条件地挂起和唤醒非GC线程，就可以使得应用程序获得更好的响应性。但是同时并行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，所以也需要花费更多的CPU资源。后文在Art的并发和非并发GC中我们也会着重说明下这两者的区别。

2.4 对象的分配和GC触发时机

1. 调用函数dvmHeapSourceAlloc在Java堆上分配指定大小的内存。如果分配成功，那么就将分配得到的地址直接返回给调用者了。函数dvmHeapSourceAlloc在不改变Java堆当前大小的前提下进行内存分配，这是属于轻量级的内存分配动作。

2. 如果上一步内存分配失败，这时候就需要执行一次GC了。不过如果GC线程已经在运行中，即gDvm.gcHeap->gcRunning的值等于true，那么就直接调用函数dvmWaitForConcurrentGcToComplete等到GC执行完成就是了。否则的话，就需要调用函数gcForMalloc来执行一次GC了，参数false表示不要回收软引用对象引用的对象。

3. GC执行完毕后，再次调用函数dvmHeapSourceAlloc尝试轻量级的内存分配操作。如果分配成功，那么就将分配得到的地址直接返回给调用者了。

4. 如果上一步内存分配失败，这时候就得考虑先将Java堆的当前大小设置为Dalvik虚拟机启动时指定的Java堆最大值，再进行内存分配了。这是通过调用函数dvmHeapSourceAllocAndGrow来实现的。

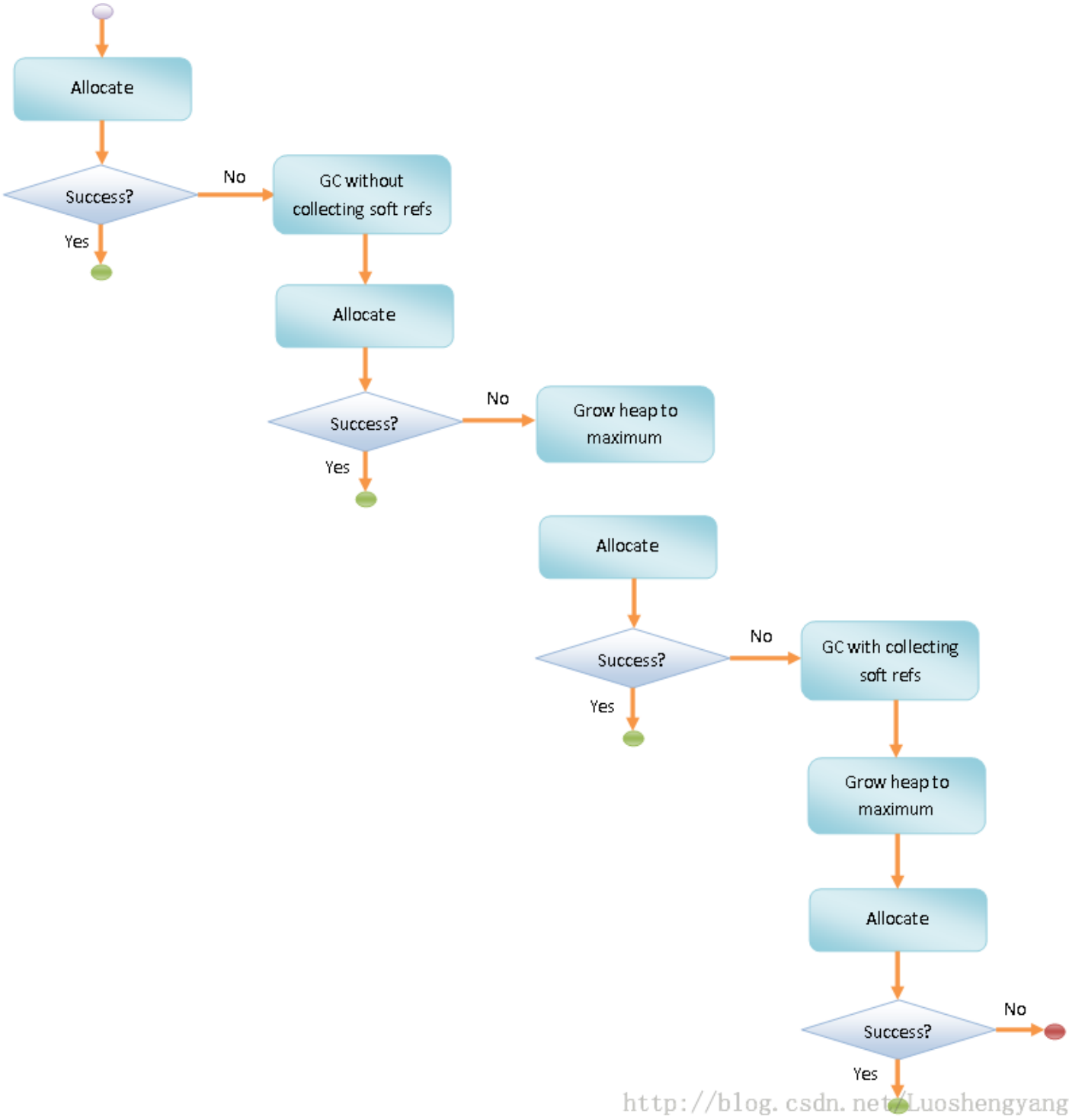
5. 如果调用函数dvmHeapSourceAllocAndGrow分配内存成功，则直接将分配得到的地址直接返回给调用者了。

6. 如果上一步内存分配还是失败，这时候就得出狠招了。再次调用函数gcForMalloc来执行GC。参数true

表示要回收软引用对象引用的对象。

7. GC执行完毕，再次调用函数dvmHeapSourceAllocAndGrow进行内存分配。这是最后一次努力了，成功与事都到此为止。

示例图如下：



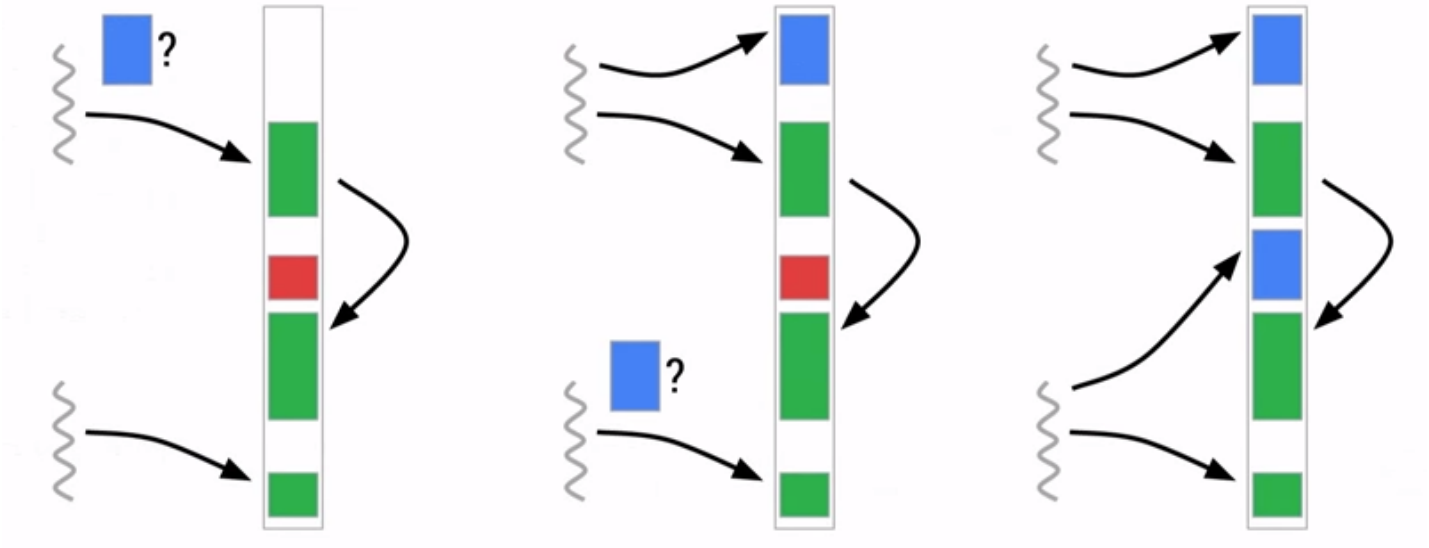
通过这个流程可以看到，在对象的分配中会导致GC，第一次分配对象失败我们会触发GC但是不回收Soft的引用，如果再次分配还是失败我们就会将Soft的内存也给回收，前者触发的GC是GC*FOR*MALLOC类型的GC，后者是GC*BEFORE*OOM类型的GC。而当内存分配成功后，我们会判断当前的内存占用是否是达到了GC*CONCURRENT的阀值，如果达到了那么又会触发GC*CONCURRENT。

那么这个阀值又是如何来的呢，上面我们说到的一个目标利用率，GC后我们会记录一个目标值，这个值理论上需要再上述的范围之内，如果不在我们会选取边界值做为目标值。虚拟机会记录这个目标值，当做当前允许总的可以分配到的内存。同时根据目标值减去固定值（200~500K),当做触发GC_CONCURRENT事件的阈值。

2.5 回收算法和内存碎片

主流的大部分Davik采取的都是标注与清理（Mark and Sweep）回收算法，也有实现了拷贝GC的，这一点和HotSpot是不一样的，具体使用什么算法是在编译期决定的，无法在运行的时候动态更换。如果在编译dalvik虚拟机的命令中指明了"WITH*COPYING*GC"选项，则编译"/dalvik/vm/alloc/Copying.cpp"源码 – 此是Android中拷贝GC算法的实现，否则编译"/dalvik/vm/alloc/HeapSource.cpp" – 其实现了标注与清理GC算法。

由于Mark and Sweep算法的缺点，容易导致内存碎片，所以在这个算法下，当我们有大量不连续小内存的时候，再分配一个较大对象时，还是会非常容易导致GC，比如我们在该手机上decode图片，具体情况如下：



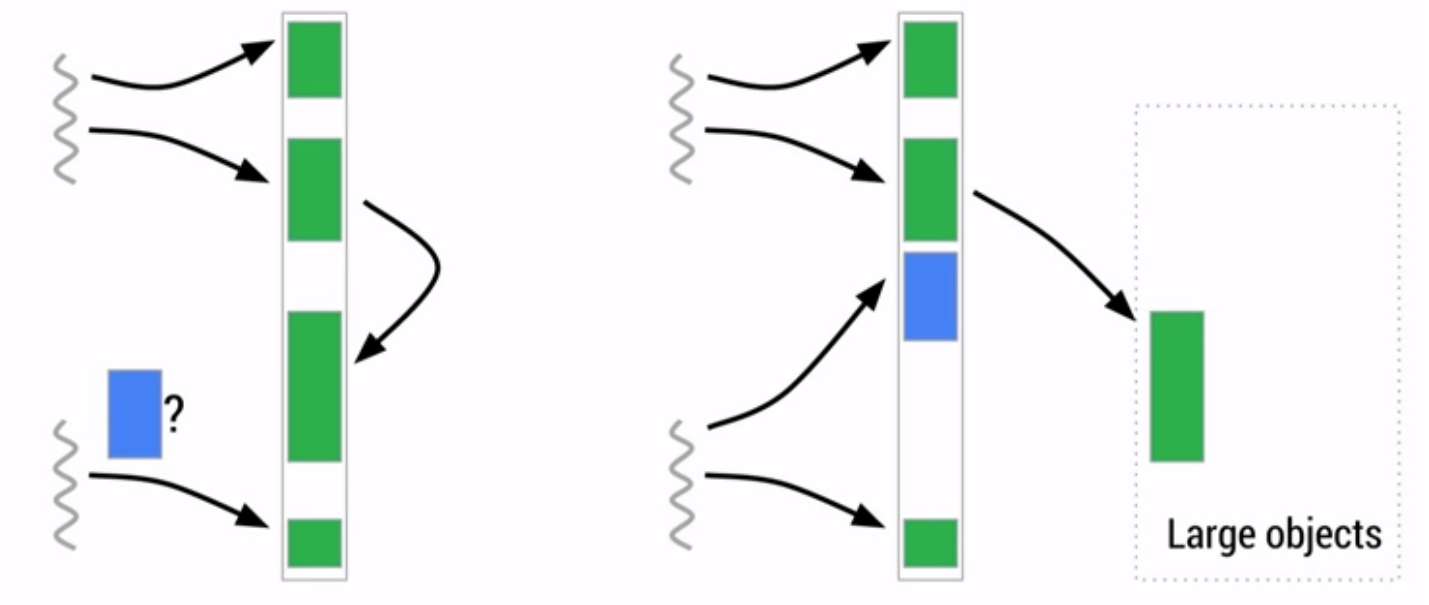
所以对于Dalvik虚拟机的手机来说，我们首先要尽量避免掉频繁生成很多临时小变量（比如说：getView，onDraw等函数），另一个又要尽量去避免产生很多长生命周期的大对象。

3、ART内存回收机制

3.1 Java堆

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，Image Space用来存在一些预加载的类， Zygote Space和Allocation Space与Dalvik虚拟机垃圾收集机制中的Zygote堆和Active堆的作用是一样的，

Large Object Space就是一些离散地址的集合，用来分配一些大对象从而提高了GC的管理效率和整体性能，类似如下图：



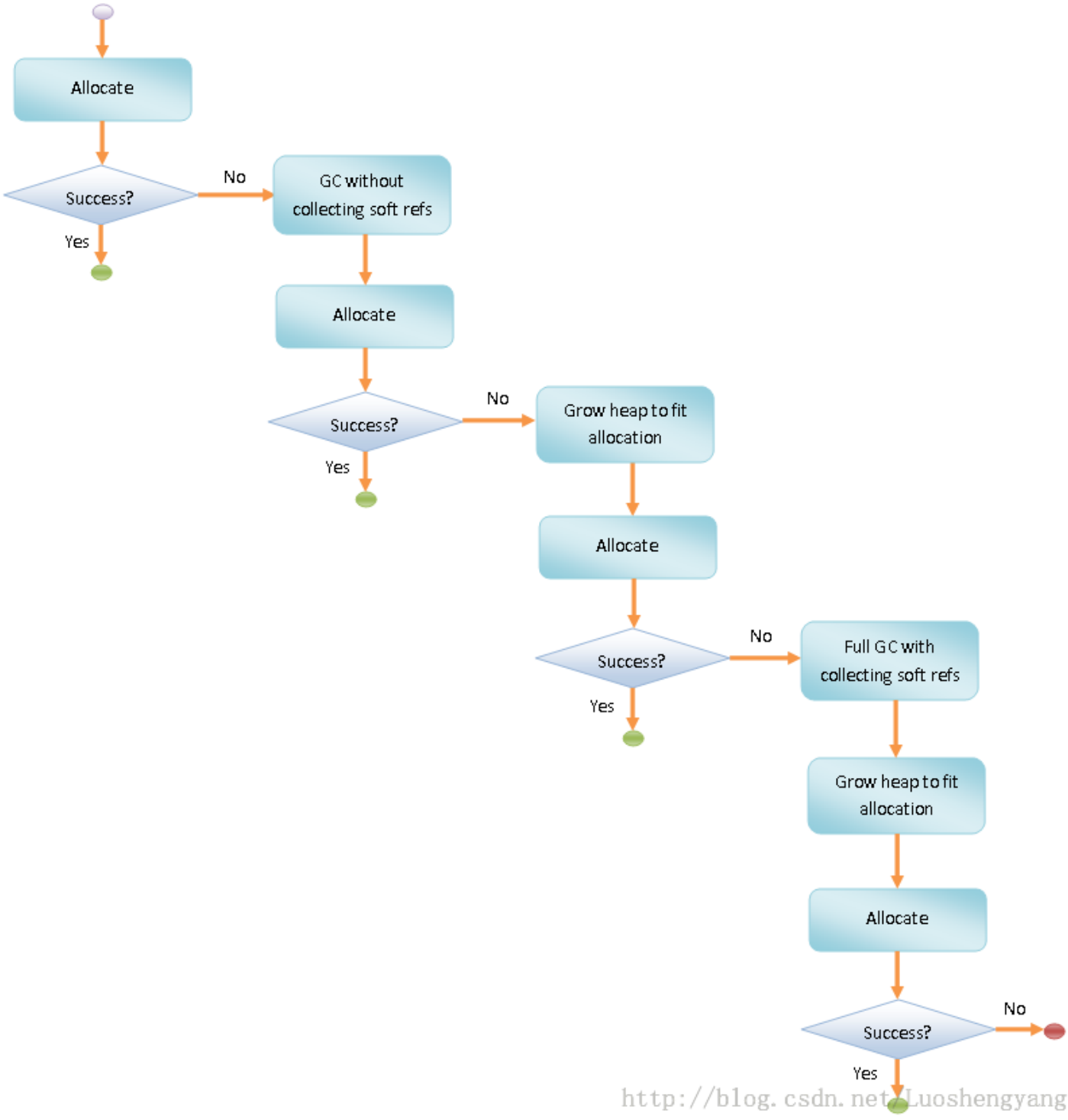
在下文的GC Log中，我们也能看到在art的GC Log中包含了LOS的信息，方便我们查看大内存的情况。

3.2 GC的类型

- kGcCauseForAlloc ，当要分配内存的时候发现内存不够的情况下引起的GC，这种情况下的GC会stop world
- kGcCauseBackground，当内存达到一定的阀值的时候会去出发GC，这个时候是一个后台gc，不会引起stop world
- kGcCauseExplicit，显示调用的时候进行的gc，如果art打开了这个选项的情况下，在system.gc的时候会进行gc
- 其他更多

3.3 对象的分配和GC触发时机

由于Art下内存分配和Dalvik下基本没有任何区别，我直接贴图带过了。



3.4 并发和非并发GC

Art在GC上不像Dalvik仅有一种回收算法，Art在不同的情况下会选择不同的回收算法，比如Alloc内存不够的时候会采用非并发GC，而在Alloc后发现内存达到一定阀值的时候又会触发并发GC。同时在前后台的情况下GC策略也不尽相同，后面我们会一一给大家说明。

- **非并发GC**

步骤1. 调用子类实现的成员函数InitializePhase执行GC初始化阶段。

步骤2. 挂起所有的ART运行时线程。

步骤3. 调用子类实现的成员函数MarkingPhase执行GC标记阶段。

步骤4. 调用子类实现的成员函数ReclaimPhase执行GC回收阶段。

步骤5. 恢复第2步挂起的ART运行时线程。

步骤6. 调用子类实现的成员函数FinishPhase执行GC结束阶段。

- **并发GC**

步骤1. 调用子类实现的成员函数InitializePhase执行GC初始化阶段。

步骤2. 获取用于访问Java堆的锁。

步骤3. 调用子类实现的成员函数MarkingPhase执行GC并行标记阶段。

步骤4. 释放用于访问Java堆的锁。

步骤5. 挂起所有的ART运行时线程。

步骤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。。

步骤7. 恢复第4步挂起的ART运行时线程。

步骤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。

步骤9. 获取用于访问Java堆的锁。

步骤10. 调用子类实现的成员函数ReclaimPhase执行GC回收阶段。

步骤11. 释放用于访问Java堆的锁。

步骤12. 调用子类实现的成员函数FinishPhase执行GC结束阶段。

所以不论是并发还是非并发，都会引起stopworld的情况出现，并发的情况下单次stopworld的时间会更短，基本区别和。

3.5 Art并发和Dalvik并发GC的差异

首先可以通过如下2张图来对比下

Dalvik GC： dalvik的日志格式基本如下：

D/dalvikvm: , , ,

gc*reason：就是我们上文提到的，是gc*alloc还是gc_concurrent，了解到不同的原因方便我们做不同的处理。

amount_freed：表示系统通过这次GC操作释放了多少内存

Heap_stats：中会显示当前内存的空闲比例以及使用情况（活动对象所占内存 / 当前程序总内存）

Pause*time：表示这次GC操作导致应用程序暂停的时间。关于这个暂停的时间，在2.3之前GC操作是不能并发进行的，也就是系统正在进行GC，那么应用程序就只能阻塞住等待GC结束。而自2.3之后，GC操作改成了并发的方式进行，就是说GC的过程中不会影响到应用程序的正常运行，但是在GC操作的开始和结束的时候会短暂阻塞一段时间，所以还有后续的一个total*time。

Total*time:表示本次GC所花费的总时间和上面的Pause*time,也就是stop all是不一样的，卡顿时间主要看上面的pause_time

4.2 Art GC日志

I/art: , , , ,

基本情况和Dalvik没有什么差别，GC的Reason更多了，还多了一个OS*Space*Status

LOS*Space*Status：Large Object Space，大对象占用的空间，这部分内存并不是分配在堆上的，但仍属于应用程序内存空间，主要用来管理 bitmap 等占内存大的对象，避免因分配大内存导致堆频繁 GC。