## 阿里巴巴--Android(虚拟机)内存模型

本专栏专注分享大型Bat面试知识，后续会持续更新，喜欢的话麻烦点击一个star

从Jvm内存模型中入手对于理解GC会有很大的帮助，不过这里只需要了解一个大概，说多了反而混淆视线。

Jvm(Java虚拟机)主要管理两种类型内存：堆和非堆。 堆是运行时数据区域，所有类实例和数组的内存均从此处分配。 非堆是JVM留给自己用的，包含方法区、JVM内部处理或优化所需的内存（如 JIT Compiler，Just-in-time Compiler，即时编译后的代码缓存）、每个类结构（如运行时常数池、字段和方法数据）以及方法和构造方法的代码。

简言之，Java程序内存主要（这里强调主要二字）分两部分，堆和非堆。大家一般new的对象和数组都是在堆中的，而GC主要回收的内存也是这块堆内存。

配一张示意图总结一下：

堆内存（Heap Memory）： 存放Java对象 非堆内存（Non-Heap Memory）： 存放类加载信息和其它meta-data 其它（Other）： 存放JVM 自身代码等

Java内存模型

Java内存模型

## 堆内存模型

既然重点是堆内存，我们就再看看堆的内存模型。

堆内存由垃圾回收器的自动内存管理系统回收。 堆内存分为两大部分：新生代和老年代。比例为1：2。 老年代主要存放应用程序中生命周期长的存活对象。 新生代又分为三个部分：一个Eden区和两个Survivor区，比例为8：1：1。 Eden区存放新生的对象。 Survivor存放每次垃圾回收后存活的对象。

堆内存模型

堆内存模型

看晕了吧，关注这几个问题：

1. 为什么要分新生代和老年代？
2. 新生代为什么分一个Eden区和两个Survivor区?
3. 一个Eden区和两个Survivor区的比例为什么是8:1:1？

现在还不能解释为什么，但这几个问题都是垃圾回收机制所采用的算法决定的。 所以问题转化为，是何种算法？为什么要采用此种算法？

## 可回收对象的判定

讲算法之前，我们先要搞清楚一个问题，什么样的对象是垃圾（无用对象），需要被回收？ 目前市面上有两种算法用来判定一个对象是否为垃圾。

#### 1. 引用计数算法

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。 引用计数算法 优点是简单，高效，现在的objective-c用的就是这种算法。 缺点是很难处理循环引用，比如图中相互引用的两个对象则无法释放。 这个缺点很致命，有人可能会问，那objective-c不是用的好好的吗？ 我个人并没有觉得objective-c好好的处理了这个循环引用问题，它其实是把这个问题抛给了开发者。

#### 2. 可达性分析算法（根搜索算法）

为了解决上面的循环引用问题，Java采用了一种新的算法：可达性分析算法。 从GC Roots（每种具体实现对GC Roots有不同的定义）作为起点，向下搜索它们引用的对象，可以生成一棵引用树，树的节点视为可达对象，反之视为不可达。 ![可达性分析算法](data:image/svg+xml;utf8;base64,xmlversiog==) OK，即使循环引用了，只要没有被GC Roots引用了依然会被回收，完美！ 但是，这个GC Roots的定义就要考究了，Java语言定义了如下GC Roots对象：

虚拟机栈（帧栈中的本地变量表）中引用的对象。 方法区中静态属性引用的对象。 方法区中常量引用的对象。 本地方法栈中JNI引用的对象。

## Stop The World

有了上面的垃圾对象的判定，我们还要考虑一个问题，请大家做好心里准备，那就是Stop The World。 因为垃圾回收的时候，需要整个的引用状态保持不变，否则判定是判定垃圾，等我稍后回收的时候它又被引用了，这就全乱套了。所以，GC的时候，其他所有的程序执行处于暂停状态，卡住了。 幸运的是，这个卡顿是非常短（尤其是新生代），对程序的影响微乎其微 （关于其他GC比如并发GC之类的，在此不讨论）。 所以GC的卡顿问题由此而来，也是情有可原，暂时无可避免。

## 几种垃圾回收算法

有了上面两个大基础，我们的GC才能开始。 那么问题来了，已经知道哪些是垃圾对象了，怎么回收呢？目前主流有以下几种算法。 *PS：大家可以先猜猜Java虚拟机（这里默认指Hotspot）采用的是那种算法，…，答对了，是分代回收算法，现在是不是明白了前面堆内存为什么要分新生代和老年代了吧。但是即使猜对了，也要看其他几种算法哦，不然不要说我没提醒你，你会直接看不懂分代回收算法的。*

#### 1. 标记清除算法 (Mark-Sweep)

标记-清除算法分为两个阶段：标记阶段和清除阶段。标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。 优点是简单，容易实现。 缺点是容易产生内存碎片，碎片太多可能会导致后续过程中需要为大对象分配空间时无法找到足够的空间而提前触发新的一次垃圾收集动作。 示意图如下（不用我解说了吧）： 标记清除算法示意图

#### 2. 复制算法 (Copying)

复制算法将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。 优缺点就是，实现简单，运行高效且不容易产生内存碎片，但是却对内存空间的使用做出了高昂的代价，因为能够使用的内存缩减到原来的一半。 从算法原理我们可以看出，Copying算法的效率跟存活对象的数目多少有很大的关系，如果存活对象很多，那么Copying算法的效率将会大大降低。 示意图如下（不用我解说了吧）： 复制算法示意图

#### 3. 标记整理算法 (Mark-Compact)

该算法标记阶段和Mark-Sweep一样，但是在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉端边界以外的内存。 所以，特别适用于存活对象多，回收对象少的情况下。 示意图如下（不用我解说了吧）： 标记整理算法示意图

#### 4. 分代回收算法

分代回收算法其实不算一种新的算法，而是根据复制算法和标记整理算法的的特点综合而成。这种综合是考虑到java的语言特性的。 这里重复一下两种老算法的适用场景：

复制算法：适用于存活对象很少。回收对象多 标记整理算法: 适用用于存活对象多，回收对象少

刚好互补！不同类型的对象生命周期决定了更适合采用哪种算法。 于是，我们根据对象存活的生命周期将内存划分为若干个不同的区域。一般情况下将堆区划分为老年代（Old Generation）和新生代（Young Generation），老年代的特点是每次垃圾收集时只有少量对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点采取最适合的收集算法。 这就是分代回收算法。 现在回头去看堆内存为什么要划分新生代和老年代，是不是觉得如此的清晰和自然了？

我们再说的细一点：

1. 对于新生代采取Copying算法，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数较少，采用Copying算法效率最高。但是，但是，但是，实际中并不是按照上面算法中说的1：1的比例来划分新生代的空间的，而是将新生代划分为一块较大的Eden空间和两块较小的Survivor空间，比例为8：1：1.。为什么？下一节深入分析。
2. 由于老年代的特点是每次回收都只回收少量对象，一般使用的是Mark-Compact算法。

## 深入理解分代回收算法

对于这个算法，我相信很多人还是有疑问的，我们来各个击破，说清楚了就很简单。

#### 为什么不是一块Survivor空间而是两块？

这里涉及到一个新生代和老年代的存活周期的问题，比如一个对象在新生代经历15次（仅供参考）GC，就可以移到老年代了。问题来了，当我们第一次GC的时候，我们可以把Eden区的存活对象放到Survivor A空间，但是第二次GC的时候，Survivor A空间的存活对象也需要再次用Copying算法，放到Survivor B空间上，而把刚刚的Survivor A空间和Eden空间清除。第三次GC时，又把Survivor B空间的存活对象复制到Survivor A空间，如此反复。 所以，这里就需要两块Survivor空间来回倒腾。

#### 为什么Eden空间这么大而Survivor空间要分的少一点？

新创建的对象都是放在Eden空间，这是很频繁的，尤其是大量的局部变量产生的临时对象，这些对象绝大部分都应该马上被回收，能存活下来被转移到survivor空间的往往不多。所以，设置较大的Eden空间和较小的Survivor空间是合理的，大大提高了内存的使用率，缓解了Copying算法的缺点。 我看8：1：1就挺好的，当然这个比例是可以调整的，包括上面的新生代和老年代的1：2的比例也是可以调整的。 新的问题又来了，从Eden空间往Survivor空间转移的时候Survivor空间不够了怎么办？直接放到老年代去。

#### Eden空间和两块Survivor空间的工作流程

这里本来简单的Copying算法被划分为三部分后很多朋友一时理解不了，也确实不好描述，下面我来演示一下Eden空间和两块Survivor空间的工作流程。

现在假定有新生代Eden，Survivor A， Survivor B三块空间和老生代Old一块空间。

// 分配了一个又一个对象  
放到Eden区  
// 不好，Eden区满了，只能GC(新生代GC：Minor GC)了  
把Eden区的存活对象copy到Survivor A区，然后清空Eden区（本来Survivor B区也需要清空的，不过本来就是空的）  
// 又分配了一个又一个对象  
放到Eden区  
// 不好，Eden区又满了，只能GC(新生代GC：Minor GC)了  
把Eden区和Survivor A区的存活对象copy到Survivor B区，然后清空Eden区和Survivor A区  
// 又分配了一个又一个对象  
放到Eden区  
// 不好，Eden区又满了，只能GC(新生代GC：Minor GC)了  
把Eden区和Survivor B区的存活对象copy到Survivor A区，然后清空Eden区和Survivor B区  
// ...  
// 有的对象来回在Survivor A区或者B区呆了比如15次，就被分配到老年代Old区  
// 有的对象太大，超过了Eden区，直接被分配在Old区  
// 有的存活对象，放不下Survivor区，也被分配到Old区  
// ...  
// 在某次Minor GC的过程中突然发现：  
// 不好，老年代Old区也满了，这是一次大GC(老年代GC：Major GC)  
Old区慢慢的整理一番，空间又够了  
// 继续Minor GC  
// ...  
// ...

从这段流程中，我相信大家应该有了一个清晰的认识了，当然为了说明原理，这只是最简化版本。

## 触发GC的类型

了解这些是为了解决实际问题，Java虚拟机会把每次触发GC的信息打印出来来帮助我们分析问题，所以掌握触发GC的类型是分析日志的基础。

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