Garbage Collector Implementation

Wednesday, February 7, 2018 2:05 PM

Java SE平台的一个优势是它可以让开发人员从内存分配和垃圾回收的复杂性中解脱出来。

然而，当垃圾收集是主要性能瓶颈的时候，理解GC实现的某些方面就非常有用。垃圾收集器对应用程序使用对象的方式做出了假设，并且这些假设都反映在可调参数中。我们可以在不牺牲抽象能力的情况下调整这些参数以提高性能。

Generational Garbage Collector

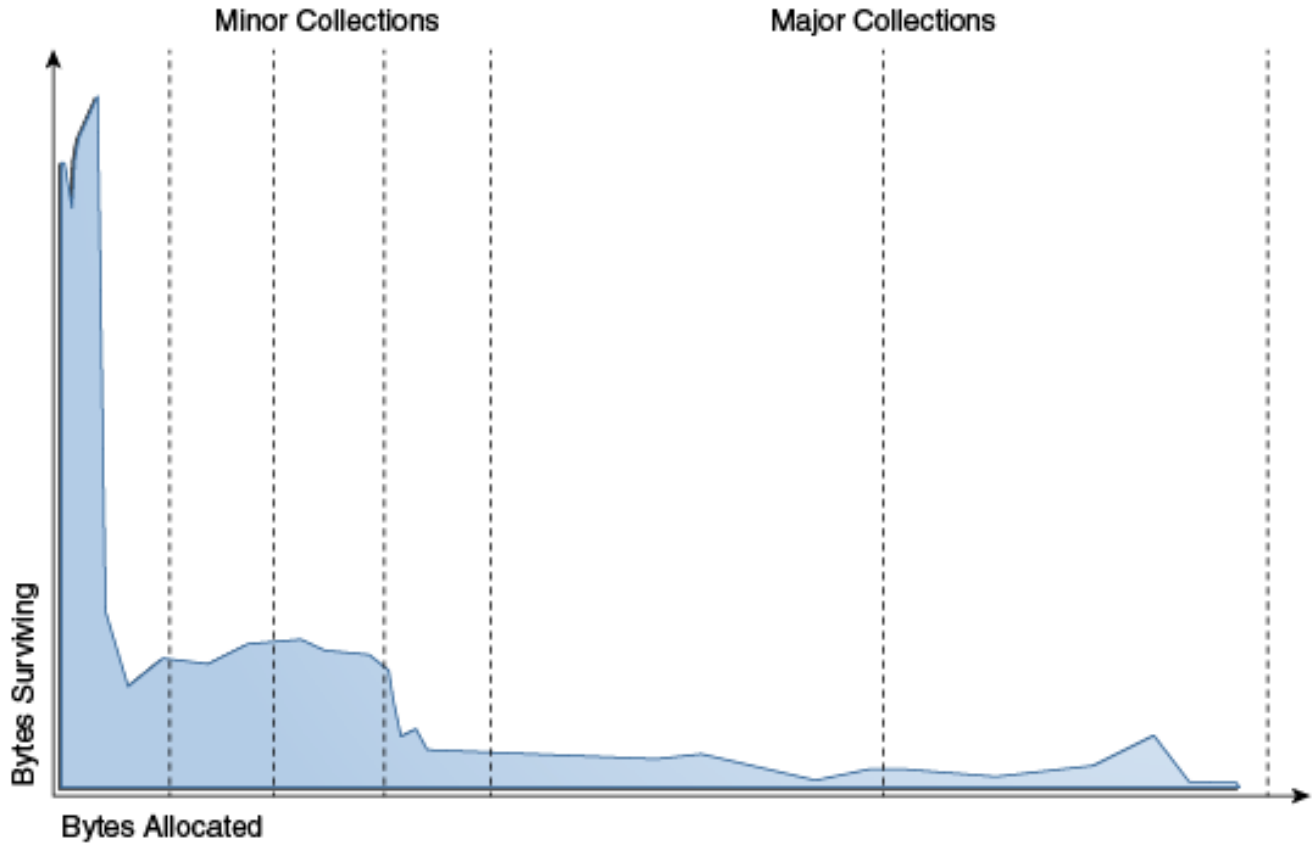
当一个对象不能被运行程序中任何其他活动对象引用时，就会被认为是垃圾，它的内存可以被VM重用。

一个理论上的、最直接的垃圾收集算法每次运行时都会遍历每个可访问的对象。任何剩余的对象都被认为是垃圾。这种方法所花费的时间与活动对象的数量成正比，这对于维护大量实时数据的大型应用程序来说是禁止的。

Java Hotspot VM包含了许多不同的垃圾收集算法，这些算法都使用了分代收集的技术。尽管朴素的垃圾收集每次都检查堆中的每个活动对象，但是分代收集利用了大多数应用程序的经验观察特性，以最小化回收未使用(垃圾)对象所需的成本。这些特性中最重要的是弱世代假说，即大多数对象只会存活很短的时间。

图3-1中的蓝色区域是对象生命周期的典型分布。x轴显示以字节为单位分配的对象寿命。y轴上的字节数是对象中对应的生存期的总字节数。左边的尖峰代表了可以被回收的对象(换句话说，在分配后不久就已经“死亡”)。例如，迭代器对象通常只存活在单个循环的持续时间。

Figure 3-1 Typical Distribution for Lifetimes of Objects



有些物体寿命更长，所以分布向右延伸。例如，通常在初始化时分配一些对象，直到VM退出。介于这两个极端之间的是一些在中间计算过程中存在的对象，在这里被看作是初始峰值的右边的块。有些应用程序的分布非常不同，但是一个非常大的数字具有这个通用的形状。有效的收集是通过关注大多数对象“英年早逝”这一事实而得以实现的。

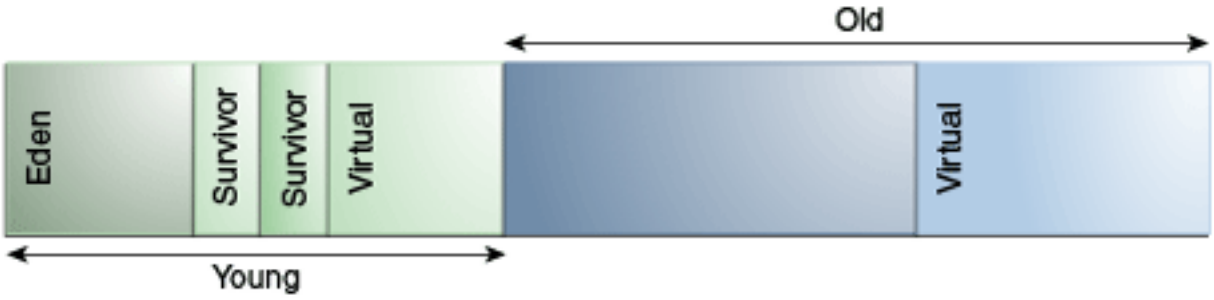
Generations

为了优化这个场景，内存是分代管理的(内存池中存放着不同年龄的对象)，而垃圾收集在每一代中都会发生。

绝大多数的对象会被分配到一个专用于年轻对象(年轻代)的池中，且其中大部分都会死在那里。当年轻代被填满时，它会触发一个Minor GC，只有年轻的代被收集，其他世代则不受影响。Minor GC的开销，首先，与被收集的对象的数目成正比，如果年轻代中所有的对象都需要被回收，那么回收的速度会非常的快。通常情况下，在每一次Minor GC中都有一部分幸存对象从年轻代被转移到老年代。最终，老年代被填满并必须收集，从而触发Major GC对整个堆进行垃圾手机。因为有大量的对象参与，Major GC的持续的时间通常比Minor GC长很多。图3-2显示了

串行垃圾收集器中几个世代的默认设置:

Figure 3-2 Default Arrangement of Generations in the Serial Collector



在启动的时候，Java Hotspot VM将整个Java堆保留在地址空间中，但是除非需要，它不会为它分配任何物理内存。覆盖Java堆的整个地址空间在逻辑上被划分为年轻的和年老的。为对象内存预留的完整地址空间可以分为年轻和年老两代。

年轻代由eden(伊甸园)和两个survivor space(幸存者空间)组成。大多数对象最初是在eden中分配的。一个幸存者空间(to)在任何时候都是空的，在垃圾收集过程中作为eden和另一个幸存者空间(from)中存活对象的目标空间。在垃圾收集结束之后，eden和源幸存者空间(from)是空的。在下一次垃圾收集中，两个survivor space的功能互换。填充在一个survivor space中对象会被复制到另一个survivor space。对象就以这种方式在survivor space中来回复制，直到它们被复制了一定次数或者没有足够的空间。这些对象就会被复制到老年代中。这个过程也叫做衰老。

Performance Considerations

衡量垃圾收集的主要指标是吞吐率和延迟。

- 吞吐量是在长时间内没有进行垃圾收集的时间占总时间的百分比。吞吐量包括内存分配的时间(但是通常不需要优化内存分配)。
- 延迟是指应用程序的响应性。因垃圾收集而产生的暂停会影响应用程序的响应能力。

用户对垃圾收集有不同的要求。例如，一些人认为web服务器的衡量指标是吞吐量，在垃圾收集期间暂停可能是可以忍受的，或者会因为网络延迟

而变得不明显。但在交互式图形程序中，即使是短暂的停顿也会对用户体验产生负面影响。

有些用户对其他的考虑很敏感。内存占用(Footprint)是以页面和缓存行来度量的进程的工作集。在有限的物理内存或多进程的系统上，内存占用可能决定系统的可伸缩性。及时性指的是从一个对象死后到内存可用所需的时间，这是分布式系统的一个重要性能考量，包括远程方法调用(RMI)。

一般而言，如何选择某一个世代的大小是以上所有因素之间的权衡。例如，一个非常大的年轻代可能会最大限度地提高吞吐量，但是这样做会影响内存占用、及时性和暂停时间。使用小的年轻代可以最小化年轻代的暂停时间，而代价则是降低吞吐量。一个世代的大小不会影响另一个世代的收集频率和暂停时间。

在如何决定每一个时代的大小这个问题上，没有银弹。最好的选择取决于应用程序使用内存方式和用户需求。因此，虚拟机对垃圾收集器的选择并不总是最优的，他们还可能会被命令行选项覆盖。看影响垃圾收集性能的因素（下个章节）。

Throughput and Footprint Measurement

吞吐量和内存使用量最好使用基于特定应用程序的标准来度量。

例如，web服务器的吞吐量可以使用客户端负载生成器进行测试，而服务器的内存占用率可以使用pmap命令在Solaris操作系统上进行测量。但是，通过检查虚拟机本身的诊断输出，可以很容易地预估垃圾收集引起的暂停。

命令行选项-verbose:gc会打印每次垃圾收集中关于堆和垃圾收集的信息。示例如下:

```
[15,651s][info ][gc] GC(36) Pause Young (G1 Evacuation Pause) 239M->57M(307M) (15,646s, 15,651s) 5,048ms
[16,162s][info ][gc] GC(37) Pause Young (G1 Evacuation Pause) 238M->57M(307M) (16,146s, 16,162s) 16,565ms
[16,367s][info ][gc] GC(38) Pause Full (System.gc()) 69M->31M(104M) (16,202s, 16,367s) 164,581ms
```

输出显示两个年代的回收，然后是应用程序调用System.gc()发起的一个Full GC。
这些行以一个指示应用程序启动的时间戳为开始，然后是关于这一行的日志级别(info
及标记(gc)的信息。再后面是一个GC识别号。在这个例子，有3个GC，36、37和38。
然后记录GC的类型和说明GC的原因。在此之后，是关于内存消耗的信息。该日志
使用“GC前”->“GC后”(“堆大小”)的格式。

这个示例的第一行显示239M->57M(307M)，这意味着在GC之前使用了239M的内存，
并且本次GC清除了大部分内存，57 M存活了下来。堆大小是307 MB。在这个例子中，
Full GC将堆从307 M压缩到104M。在内存使用信息之后，记录了GC的开始和
结束时间，并由此可以算出持续时间(结束-开始)。

-verbose:gc命令是-Xlog:gc的别名。-Xlog是在HotSpot JVM中通用的日志配置选项。
它(-Xlog)是一个基于标签的系统，gc是其中一个标签。要获得关于GC操作的
更多信息，可以配置日志以打印任何具有gc标签或者任何其他标签的消息。
这个命令行选项是-Xlog:gc*。

下面是用-Xlog:gc*作为选项打印出的G1年轻代收集的示例:

```
[10.178s][info][gc,start ] GC(36) Pause Young (G1 Evacuation Pause)
[10.178s][info][gc,task ] GC(36) Using 28 workers of 28 for evacuation
[10.191s][info][gc,phases ] GC(36) Pre Evacuate Collection Set: 0.0ms
[10.191s][info][gc,phases ] GC(36) Evacuate Collection Set: 6.9ms
[10.191s][info][gc,phases ] GC(36) Post Evacuate Collection Set: 5.9ms
[10.191s][info][gc,phases ] GC(36) Other: 0.2ms
[10.191s][info][gc,heap ] GC(36) Eden regions: 286->0(276)
[10.191s][info][gc,heap ] GC(36) Survivor regions: 15->26(38)
[10.191s][info][gc,heap ] GC(36) Old regions: 88->88
[10.191s][info][gc,heap ] GC(36) Humongous regions: 3->1
[10.191s][info][gc,metaspace ] GC(36) Metaspace: 8152K->8152K(1056768K)
[10.191s][info][gc ] GC(36) Pause Young (G1 Evacuation Pause) 391M->114M(508M) 13.075ms
[10.191s][info][gc,cpu ] GC(36) User=0.20s Sys=0.00s Real=0.01s
```