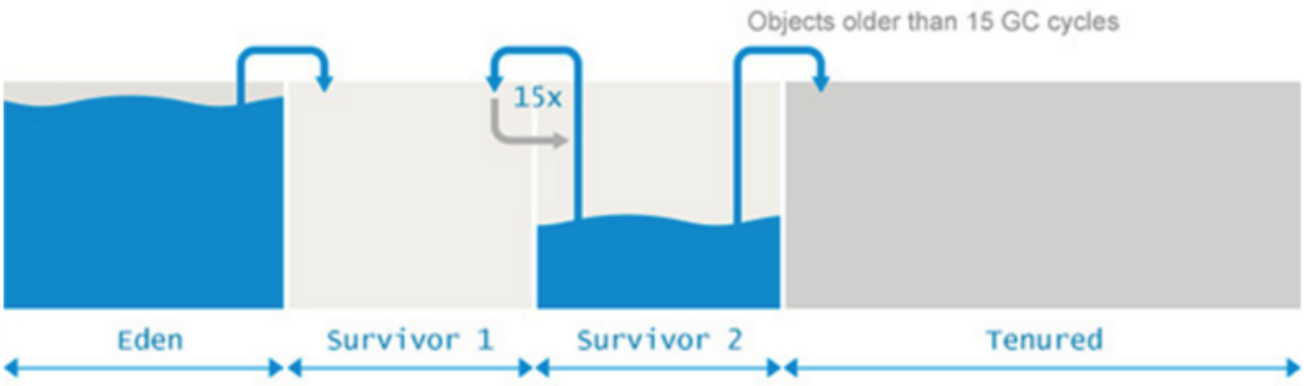
Minor GC vs Major GC vs Full GC

在Plumbr的工作过程中遇到GC间隙功能探测问题使我不得不关注相关文章，书籍，简报。自始至终，我不止一次迷惑于 Minor, Major and Full GC 的用法。为了搞清楚这些疑惑我写这篇博客。

这篇博客期望读者了解JVM 底层 GC机制。jvm heap区 分为 Eden，Survivor，Tenured/Old区。分代概念以及不同的GC算法超出了了此次讨论的范围。



Minor GC

新生代(由 Eden and Survivor 组成)的垃圾收集叫做Minor GC。该定义清晰易于理解。但是以下几点仍然需要我们注意：

1. 当jvm 无法为新建对象分配内存空间的时候Minor GC被触发，例如新生代空间被占满。因此新生代空间占用率越高，Minor GC越频繁。
2. 当空间被占满，它下面的所有对象都会被复制，而且堆顶指针从空闲空间的零位置移动（译者注：此处为复制算法）。因此取代传统的标记清除压缩算法，去清理Eden区和Survivor区，因此Eden和Survivor区无内存碎片产生。
3. 在Minor GC期间,实际上Tenured区被忽略，实际上Tenured区引用young区的对象被当作GC roots。在标记期间young区引用的Tenured区对象的对象会被忽略。
4. 反对所有Minor GC都会触发“stop-the-world”这一观点。在大多数应用中，忽略"stop-the-world"停留时长。不可否认的是新生代中的一些对象被错误当成垃圾而不会被移动到Survivor/Old区。如果笔者反对的观点成立，一些新生对象由于不合适被当作垃圾，导致Minor GC停顿将会耗费更多的时间。

因此Minor GC的情况相当清楚了，每次Minor GC只清理新生代。

寻找减少GC停顿时长的方式？automatically detect what causes GC pauses 可以解决你的难题。

Major GC vs Full GC

在目前的项目中还没有明确的定义，这点需要注意。JVM规范和垃圾收集研究论文都没有提及，但是乍一看，这些建立在我们掌握了Minor GC清理新生代上的定义并非难事：

1. Major GC清理Tenured区。
2. Full GC清理整个heap区，包括Yong区和Tenured区。

不幸的是这些有点复杂，难于解释。首先，Minor GC触发Major GC，在很多情形下，将这两者分开是不可能的。另一方面，许多现代垃圾收集平台倾向于清理Tenured区，因此，用“cleaning”术语仅仅是部分正确。

GC无论被称作Major GC还是Full GC，你应该搞清楚无论GC停止所有的应用线程还是它可以和应用线程同时进行。

这个问题甚至发生在JVM标准工具。下面是最好的解决渠道和实例，让我们比较运行在同一个JVM的两个不同工具对Concurrent Mark and Sweep收集器的输出(-XX:+UseConcMarkSweepGC)。

首先尝试使用 jstat ：

```
my-precious: me$ jstat -gc -t 4235 1s
```

```
Time S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC
CCSU   YGC     YGCT    FGC    FGCT     GCT
 5.7 34048.0 34048.0  0.0   34048.0 272640.0 194699.7 1756416.0   181419.9  18304.0 17865.1
2688.0 2497.6      3    0.275   0      0.000    0.275
 6.7 34048.0 34048.0 34048.0  0.0   272640.0 247555.4 1756416.0   263447.9  18816.0 18123.3
2688.0 2523.1      4    0.359   0      0.000    0.359
 7.7 34048.0 34048.0  0.0   34048.0 272640.0 257729.3 1756416.0   345109.8  19072.0 18396.6
2688.0 2550.3      5    0.451   0      0.000    0.451
 8.7 34048.0 34048.0 34048.0 34048.0 272640.0 272640.0 1756416.0  444982.5  19456.0 18681.3
2816.0 2575.8      7    0.550   0      0.000    0.550
 9.7 34048.0 34048.0 34046.7  0.0   272640.0 16777.0  1756416.0   587906.3  20096.0 19235.1
2944.0 2631.8      8    0.720   0      0.000    0.720
10.7 34048.0 34048.0  0.0   34046.2 272640.0 80171.6  1756416.0   664913.4  20352.0 19495.9
2944.0 2657.4      9    0.810   0      0.000    0.810
11.7 34048.0 34048.0 34048.0  0.0   272640.0 129480.8 1756416.0   745100.2  20608.0 19704.5
2944.0 2678.4     10    0.896   0      0.000    0.896
12.7 34048.0 34048.0  0.0   34046.6 272640.0 164070.7 1756416.0   822073.7  20992.0 19937.1
3072.0 2702.8     11    0.978   0      0.000    0.978
13.7 34048.0 34048.0 34048.0  0.0   272640.0 211949.9 1756416.0   897364.4  21248.0 20179.6
3072.0 2728.1     12    1.087   1      0.004    1.091
14.7 34048.0 34048.0  0.0   34047.1 272640.0 245801.5 1756416.0   597362.6  21504.0 20390.6
3072.0 2750.3     13    1.183   2      0.050    1.233
15.7 34048.0 34048.0  0.0   34048.0 272640.0 21474.1  1756416.0   757347.0  22012.0 20792.0
3200.0 2791.0     15    1.336   2      0.050    1.386
16.7 34048.0 34048.0 34047.0  0.0   272640.0 48378.0  1756416.0   838594.4  22268.0 21003.5
3200.0 2813.2     16    1.433   2      0.050    1.484
```

上面片段取自JVM启动17秒，以这些信息为基础，我们可以推断2次Full GC之前进行12次 Minor GC，一共耗费50毫秒。你可以通过一些基于GUI的工具证实，例如jconsole和jvisualvm。

在下结论之前，我们看一下JVM运行garbage collection logs。显然-XX:+PrintGCDetails 告诉我们更多细节：

```
java -XX:+PrintGCDetails -XX:+UseConcMarkSweepGC eu.plumbr.demo.GarbageProducer
```

```
3.157: [GC (Allocation Failure) 3.157: [ParNew: 272640K->34048K(306688K), 0.0844702 secs]
272640K->69574K(2063104K), 0.0845560 secs] [Times: user=0.23 sys=0.03, real=0.09 secs]
4.092: [GC (Allocation Failure) 4.092: [ParNew: 306688K->34048K(306688K), 0.1013723 secs]
342214K->136584K(2063104K), 0.1014307 secs] [Times: user=0.25 sys=0.05, real=0.10 secs]
... cut for brevity ...
11.292: [GC (Allocation Failure) 11.292: [ParNew: 306686K->34048K(306688K), 0.0857219 secs]
971599K->779148K(2063104K), 0.0857875 secs] [Times: user=0.26 sys=0.04, real=0.09 secs]
12.140: [GC (Allocation Failure) 12.140: [ParNew: 306688K->34046K(306688K), 0.0821774 secs]
1051788K->856120K(2063104K), 0.0822400 secs] [Times: user=0.25 sys=0.03, real=0.08 secs]
12.989: [GC (Allocation Failure) 12.989: [ParNew: 306686K->34048K(306688K), 0.1086667 secs]
1128760K->931412K(2063104K), 0.1087416 secs] [Times: user=0.24 sys=0.04, real=0.11 secs]
13.098: [GC (CMS Initial Mark) [1 CMS-initial-mark: 897364K(1756416K)] 936667K(2063104K),
0.0041705 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
13.102: [CMS-concurrent-mark-start]
13.341: [CMS-concurrent-mark: 0.238/0.238 secs] [Times: user=0.36 sys=0.01, real=0.24 secs]
13.341: [CMS-concurrent-preclean-start]
13.350: [CMS-concurrent-preclean: 0.009/0.009 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
13.350: [CMS-concurrent-abortable-preclean-start]
13.878: [GC (Allocation Failure) 13.878: [ParNew: 306688K->34047K(306688K), 0.0960456 secs]
1204052K->1010638K(2063104K), 0.0961542 secs] [Times: user=0.29 sys=0.04, real=0.09 secs]
14.366: [CMS-concurrent-abortable-preclean: 0.917/1.016 secs] [Times: user=2.22 sys=0.07,
real=1.01 secs]
14.366: [GC (CMS Final Remark) [YG occupancy: 182593 K (306688 K)]14.366: [Rescan (parallel) ,
0.0291598 secs]14.395: [weak refs processing, 0.0000232 secs]14.395: [class unloading, 0.0117661
secs]14.407: [scrub symbol table, 0.0015323 secs]14.409: [scrub string table, 0.0003221 secs][1
CMS-remark: 976591K(1756416K)] 1159184K(2063104K), 0.0462010 secs] [Times: user=0.14 sys=0.00,
real=0.05 secs]
14.412: [CMS-concurrent-sweep-start]
14.633: [CMS-concurrent-sweep: 0.221/0.221 secs] [Times: user=0.37 sys=0.00, real=0.22 secs]
14.633: [CMS-concurrent-reset-start]
14.636: [CMS-concurrent-reset: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

基于这些信息，我们看到12次Minor GC 之后一些事情开始发生，但是与上面2次Full GC不同的是老年代的不同阶段只有一次GC：

- 初始化标记阶段，耗费0.0041705 seconds大约4ms。为了初始化标记这个阶段进行“stop-the-world”。
- 标记和预清除并发阶段，和应用先线程并行执行。
- 最后重复标记阶段，耗费0.0462010 seconds大约46ms。再次进行“stop-the-world”。
- 并发执行清除阶段，正如名称所示，不执行“stop-the-world”，并发实施操作。

正如我们们从GC日志看到的真实情况，事实上，替代两次Full GC的仅仅一次Major GC清理Old space。

如果你考虑了基于jstat展示的数据的情况，你会做出正确结论。它正确展示出两次因为所有活动线程而进行两次“stop-the-world”,总共耗费50ms。如果你想为了吞吐量尝试优化，那么你会被误导，仅仅在初始化标记和最后重复标记阶段而进行“stop-the-world”,the jstat输出完全隐藏了并发工作。

结论

综合情况来看，这是避免考虑项目中Minor, Major or Full GC的最好方式。反之，监控你的应用延迟或者吞吐量，将结果和GC事件联系起来。连同这些事件一起，你额外需要相关信息，特别是GC事件强迫停止应用线程和并发处理部分事件。