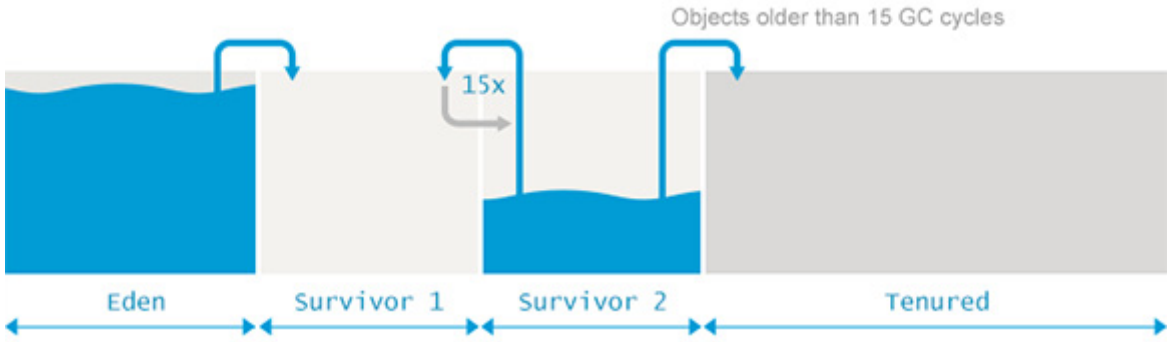
疯狂的tiger

Minor GC、Major GC和Full GC之间的区别

在 Plumbr 从事 GC 暂停检测相关功能的工作时，我被迫用自己的方式，通过大量文章、书籍和演讲来介绍我所做的工作。在整个过程中，经常对 Minor、Major、和 Full GC 事件的使用感到困惑。这也是我写这篇博客的原因，我希望能清楚地解释这其中的一些疑惑。

文章要求读者熟悉 JVM 内置的通用垃圾回收原则。堆内存划分为 Eden、Survivor 和 Tenured/Old 空间，代假设和其他不同的 GC 算法超出了本文讨论的范围。



Minor GC

从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC。这一定义既清晰又易于理解。但是，当发生 Minor GC事件的时候，有一些有趣的地方需要注意到：

昵称：疯狂的tiger
园龄：2年5个月
粉丝：18
关注：4
+加关注

< 2018年8月 >

日	一	二	三	四	五	六
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

搜索

找找看
谷歌搜索

常用链接

我的随笔

1. 当 JVM 无法为一个新的对象分配空间时会触发 Minor GC，比如当 Eden 区满了。所以分配率越高，越频繁执行 Minor GC。

2. 内存池被填满的时候，其中的内容全部会被复制，指针会从0开始跟踪空闲内存。Eden 和 Survivor 区进行了标记和复制操作，取代了经典的标记、扫描、压缩、清理操作。所以 Eden 和 Survivor 区不存在内存碎片。写指针总是停留在所使用内存池的顶部。

3. 执行 Minor GC 操作时，不会影响到永久代。从永久代到年轻代的引用被当成 GC roots，从年轻代到永久代的引用在标记阶段被直接忽略掉。

4. 质疑常规的认知，所有的 Minor GC 都会触发"全世界的暂停（stop-the-world）"，停止应用程序的线程。对于大部分应用程序，停顿导致的延迟都是可以忽略不计的。其中的真相就 是，大部分 Eden 区中的对象都能被认为是垃圾，永远也不会被复制到 Survivor 区或者老年代空间。如果正好相反，Eden 区大部分新生对象不符合 GC 条件，Minor GC 执行时暂停的时间将会长很多。

所以 Minor GC 的情况就相当清楚了——每次 Minor GC 会清理年轻代的内存。

Major GC vs Full GC

大家应该注意到，目前，这些术语无论是在 JVM 规范还是在垃圾收集研究论文中都没有正式的定义。但是我们一看就知道这些在我们已经知道的基础之上做出的定义是正确的，Minor GC 清理年轻带内存应该被设计得简单：

- Major GC 是清理永久代。
- Full GC 是清理整个堆空间—包括年轻代和永久代。

很不幸，实际上它还有点复杂且令人困惑。首先，许多 Major GC 是由 Minor GC 触发的，所以很多情况下将这两种 GC 分离是不太可能的。另一方面，许多现代垃圾收集机制会清理部分永久代空间，所以使用"cleaning"一词只是部分正确。

这使得我们不用去关心到底是叫 Major GC 还是 Full GC，大家应该关注当前的 GC 是否停止了所有应用程序的线程，还是能够并发的处理而不用停掉应用程序的线程。

这种混乱甚至内置到 JVM 标准工具。下面一个例子很好的解释了我的意思。让我们比较两个不同的工具 Concurrent Mark 和 Sweep collector (-XX:+UseConcMarkSweepGC)在 JVM 中运行时输出的跟踪记录。

第一次尝试通过 jstat 输出：

```
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

```

我的评论
我的参与
最新评论
我的标签

随笔分类

JMeter(15)
JAVA(5)
JDK(1)
JDK-IO流(6)
JVM(5)
linux-shell(23)
MongoDB(14)
spring4(34)
Vue2.0
windows 批处理(9)
项目管理(3)

随笔档案

2018年2月 (1)
2016年11月 (10)
2016年10月 (37)
2016年9月 (8)
2016年8月 (30)
2016年4月 (14)
2016年3月 (15)

文章分类

JMeter
Vue2.0(24)

最新评论

```
 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
```

这个片段是 JVM 启动后第17秒提取的。基于该信息，我们可以得出这样的结果，运行了12次 Minor GC、2次 Full GC，时间总跨度为50毫秒。通过 jconsole 或者 jvisualvm 这样的基于GUI的工具你能得到同样的结果。

```
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 ...
```

1. Re:java jvm内存管理/gc策略/参数设置
四个例子用java8 实践了下，完全不是一回事啊
--乐可2016
2. Re:java jvm内存管理/gc策略/参数设置
大神，能带一波嘛
--乐可2016

阅读排行榜

1. Shell if else语句(19790)
2. 批处理命令 For循环命令详解!(16838)
3. xcopy-参数详解(13995)
4. DOS/BAT批处理if exist else 语句的几种用法(8795)
5. shell printf命令：格式化输出语句(8222)

评论排行榜

1. java jvm内存管理/gc策略/参数设置(2)

推荐排行榜

1. Shell if else语句(2)
2. 怎样梳理属于自己的项目管理套路？(1)
3. xcopy-参数详解(1)
4. bat(续五)-获取批处理文件所在路径(1)

```
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]
```

在点头同意这个结论之前，让我们看看来自同一个 JVM 启动收集的垃圾收集日志的输出。显然- XX： + PrintGCDetails 告诉我们一个不同且更详细的故事：

基于这些信息，我们可以看到12次 Minor GC 后开始有些和上面不一样了。没有运行两次 Full GC，这不同的地方在于单个 GC 在永久代中不同阶段运行了两次：

- 最初的标记阶段，用了0.0041705秒也就是4ms左右。这个阶段会暂停“全世界（ stop-the-world）”的事件，停止所有应用程序的线程，然后开始标记。

- 并行执行标记和清洗阶段。这些都是和应用程序线程并行的。
- 最后 Remark 阶段，花费了0.0462010秒约46ms。这个阶段会再次暂停所有的事件。
- 并行执行清理操作。正如其名，此阶段也是并行的，不会停止其他线程。

所以，正如我们从垃圾回收日志中所看到的那样，实际上只是执行了 Major GC 去清理老年代空间而已，而不是执行了两次 Full GC。

如果你是后期做决 定的话，那么由 jstat 提供的数据会引导你做出正确的决策。它正确列出的两个暂停所有事件的情况，导致所有线程停止了共计50ms。但是如果你试图优化吞吐量，你会被误导的。清 单只列出了回收初始标记和最终 Remark 阶段，jstat的输出看不到那些并发完成的工作。

结论

考虑到这种情况，最好避免以 Minor、Major、Full GC 这种方式来思考问题。而应该监控应用延迟或者吞吐量，然后将 GC 事件和结果联系起来。

随着这些 GC 事件的发生，你需要额外的关注某些信息，GC 事件是强制所有应用程序线程停止了还是并行的处理了部分事件。

原文链接： javacodegeeks 翻译： ImportNew.com - 光光头去打酱油
译文链接： http://www.importnew.com/15820.html

分类: JVM

疯狂的tiger
关注 - 4
粉丝 - 18
+加关注

0 0

« 上一篇：使用本地JConsole监控远程JVM
» 下一篇：Linux Shell脚本简介

posted @ 2016-10-11 10:06 疯狂的tiger 阅读(3022) 评论(0) 编辑 收藏

最新IT新闻:

- 旗帜鲜明地反对下线顺风车
- 蝙蝠体内发现第六种埃博拉病毒：可能感染人类
- 特朗普指责谷歌操控搜索结果 有关特朗普都是坏消息|Google回应
- 小赢科技赴美提交IPO招股书：拟最多筹资2.5亿美元
- 华住5亿条用户信息疑泄露 媒体实测竟然不少是真的

» 更多新闻...

最新知识库文章:

- 如何招到一个靠谱的程序员
- 一个故事看懂"区块链"
- 被踢出去的用户
- 成为一个有目标的学习者
- 历史转折中的"杭派工程师"

» 更多知识库文章...