理解GC日志

---

title: 理解GC日志
comments: false
date: 2019-07-16 10:25:52
description: 理解 GC 打印的回收日志内容
categories: GC

概述

每一种收集器的日志形式都是由它们自身的实现决定的，换而言之，每个收集器的日志格式都可以不一样，但是虚拟机的设计者为了方便用户阅读，将每个收集器的日志都维持一定的共性。

使用 -XX:+PrintGC

使用 `-XX:+PrintGC` 开启简单的 GC 日志。

以启动 SpringBoot 为例：

```
[GC (Allocation Failure)  262144K->24800K(1005056K), 0.0182439 secs]
[GC (Metadata GC Threshold)  45740K->14793K(1005056K), 0.0099653 secs]
[Full GC (Metadata GC Threshold)  14793K->14558K(1005056K), 0.0803518 secs]
```

开头是 GC 或者 FULL GC，然后是 GC 前和 GC 后使用的堆空间的大小，括号中是堆的大小，最后是 GC 执行耗费的时间。

简单模式的 GC 日志是与收集器无关的，也没有提供其它信息。

使用 -XX:+PrintGCDetails

使用 `-XX:+PrintGCDetails` 开启详细的 GC 日志，我们再加上一个参数 `-XX:+PrintGCTimeStamps` 用来打印发生 GC 时距离 Java 虚拟机启动所经过的秒数。

以启动 SpringBoot 为例：

```
[GC (Allocation Failure) [PSYoungGen: 262144K->24659K(305664K)] 262144K->24747K(1005056K), 0.0190895 secs] [Times: user=0.06 sys=0.01, real=0.
[GC (Metadata GC Threshold) [PSYoungGen: 56032K->15248K(305664K)] 56120K->15344K(1005056K), 0.0114648 secs] [Times: user=0.07 sys=0.01, real=0
[Full GC (Metadata GC Threshold) [PSYoungGen: 15248K->0K(305664K)] [ParOldGen: 96K->15116K(699392K)] 15344K->15116K(1005056K), [Metaspace: 212
```

我的 JDK 版本是 1.8.0_212，默认的垃圾处理器是 **Parallel Scavenge（新生代）** 和 **Parallel Old（老年代）**。

[GC 和 [FULL GC

表示这次垃圾收集的停顿类型，而不是用来区分是新生代 GC 还是老年代 GC，如果有“FULL”，说明这次垃圾收集是发生了“Stop The World”的。如果是调用了 `System.gc();` 方法所触发的收集，那么这将显示 `[GC (System.gc())` 或 `[Full GC (System.gc())` 。

[PSYoungGen 和 [PSOldGen

这里表示 GC 发生的区域。这里的命名是根据收集器的改变而改变的，默认使用 Parallel Scavenge/Old 收集器，所以显示 PSYoungGen 和 PSOldGen；如果使用 Serial + CMS 收集器的话，这里就显示 ParNew（新生代）和 CMS（老年代）由于 CMS 的日志类别比较多，后面再详细说明；如果使用 Serial + Serial Old 收集器的话，这就显示 ParNew（新生代）和 Tenured（老年代）。

262144K->24659K(305664K)]

这里表示的含义是 “GC 前该内存区域已使用的容量 -> GC 后该内存区域已使用的容量（该内存区域的总容量）”。而在方括号之外的 “262144K->24747K(1005056K)” 则表示 “GC 前堆已使用容量 -> GC 后堆已使用容量（Java 堆的总容量）”。

0.0190895 secs]

在 Java 堆的总容量之后，是一个时间，表示内存区域 GC 所占用的时间，单位是秒。

**[Times: user=0.06 sys=0.01, real=0.02 secs] **

这里是收集器给出的 GC 所占用时间的更具体的数据。user：用户态消耗的 CPU 时间；sys：内核态消耗的 CPU 时间；real：操作从开始到结束所经过的墙钟时间。

墙钟时间：墙钟时间包括各种非运算的等待耗时，例如等待磁盘 I/O、等待线程堵塞，而 CPU 时间不包括这些耗时，但当系统有多 CPU 或者多核的话，多线程操作会**叠加**这些 CPU 时间，所以看到 user 或 sys 时间超过 real 时间完全是正常的。

CMS 日志讲解

在 JDK1.8.0_212下在运行时。

加入 `-XX:+PrintGCDetails -XX:+UseConcMarkSweepGC -XX:-UseAdaptiveSizePolicy` 参数。

```
[GC (Allocation Failure) [ParNew: 167808K->20287K(188736K), 0.0174186 secs] 167808K->20287K(1027648K), 0.0174666 secs] [Times: user=0.08 sys=0
[GC (CMS Initial Mark) [1 CMS-initial-mark: 13213K(838912K)] 35700K(1027648K), 0.0018935 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
[CMS-concurrent-mark-start]
[CMS-concurrent-mark: 0.023/0.024 secs] [Times: user=0.07 sys=0.00, real=0.02 secs]
[CMS-concurrent-preclean-start]
[CMS-concurrent-preclean: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[CMS-concurrent-abortable-preclean-start]
[CMS-concurrent-abortable-preclean: 0.241/0.328 secs] [Times: user=0.68 sys=0.02, real=0.33 secs]
[GC (CMS Final Remark) [YG occupancy: 108205 K (188736 K)][Rescan (parallel) , 0.0160882 secs][weak refs processing, 0.0000512 secs][class unl
[CMS-concurrent-sweep-start]
[CMS-concurrent-sweep: 0.006/0.006 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
[CMS-concurrent-reset-start]
[CMS-concurrent-reset: 0.007/0.007 secs] [Times: user=0.01 sys=0.01, real=0.01 secs]
```

回收新生代

```
[GC (Allocation Failure) [ParNew: 167808K->20287K(188736K), 0.0174186 secs] 167808K->20287K(1027648K), 0.0174666 secs] [Times: user=0.08
sys=0.01, real=0.01 secs]
```

- GC (Allocation Failure)：新生代分配内存失败，发生 GC。
- ParNew：说明新生代的垃圾收集器是 ParNew。（使用 `-XX:+UseConcMarkSweepGC` 指定老年代收集器为 CMS 时，新生代默认收集器是 ParNew）。
- 167808K->20287K(188736K)：167808K 表示回收之前新生代的使用容量，20287K 表示回收之后新生代的使用容量，188736K 表示新生代的总容量。
- 0.0174186 secs：表示收集新生代时的耗时。
- 167808K->20287K(1027648K)：167808K 表示回收之前整个堆的使用量，20287K 表示回收之后整个堆的使用量，1027648K 表示整个堆的总容量。
- 0.0174666 secs：表示整个堆回收的耗时。

初始化标记

```
[GC (CMS Initial Mark) [1 CMS-initial-mark: 13213K(838912K)] 35700K(1027648K), 0.0018935 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
```

- GC (CMS Initial Mark) ：初始化标记，开始进行老年代的回收。
- 13213K(838912K)：表示在老年代使用了 13213K 时进行回收，838912K 表示老年代的总容量。
- 35700K(1027648K)：35700K 表示整个堆的使用情况，1027648K 表示整个堆的容量。
- 0.0018935 secs：初始化标记的耗时。

这个阶段会从跟对象开始扫描，会标记由根可以直接到达的对象（直接，不是间接），标记期间会暂停整个应用程序。

并发标记

```
[CMS-concurrent-mark-start]
[CMS-concurrent-mark: 0.023/0.024 secs] [Times: user=0.07 sys=0.00, real=0.02 secs]
```

并发标记阶段。在初始化标记阶段被暂停的线程重新运行，从初始化标记阶段标记过的对象出发，所有可到达的对象都在本阶段中标记，也就是遍历整个老年代。

0.023：表示并发标记时间。

0.024：表示并发标记的墙钟时间。

由于该阶段的过程中用户线程也在运行，这就可能会发生这样的情况，已经被遍历过的对象的引用被用户线程改变，如果发生了这样的情况，JVM 就会标记这个区域为 Dirty Card。

预清理

```
[CMS-concurrent-preclean-start]
[CMS-concurrent-preclean: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

这个阶段会把上一个阶段被标记为 Dirty Card 的对象以及可达的对象重新遍历标记，完成后清除 Dirty Card 标记，另外，还会做一些必要的清扫工作，还会做一些 final remark 阶段需要的准备工作。

这个阶段也是并发执行的，与用户线程同时工作。

说实话，这个是我参考网络上的解释，我也不太懂。

可中止的预清理

```
[CMS-concurrent-abortable-preclean-start]
[CMS-concurrent-abortable-preclean: 0.241/0.328 secs] [Times: user=0.68 sys=0.02, real=0.33 secs]
```

这个阶段尝试着去承担接下来 "Stop The World" 的 final remark 阶段足够多的工作，由于这个阶段是重复的做相同的事情直到发生 aboart 的条件（比如：重复的次数、多少量的工作、持续的时间等等）之一才会停止。这个阶段很大程度的影响着即将来临的 final remark 的停顿。

这个阶段也是并发执行的，与用户线程同时工作。

说实话，这个是我参考网络上的解释，我也不太懂。

重新标记阶段

```
[GC (CMS Final Remark) [YG occupancy: 108205 K (188736 K)][Rescan (parallel) , 0.0160882 secs][weak refs processing, 0.0000512 secs][class unl
```

- [GC (CMS Final Remark)：表示这是CMS的重新标记阶段，会 STW，该阶段的任务是完成标记整个年老代的所有的存活对象，尽管先前的 pre clean 阶段尽量应对处理了并发运行时用户线程改变的对象应用的标记，但是不可能跟上对象改变的速度，只是为 final remark 阶段尽量减少了负担。
- [YG occupancy: 108205 K (188736 K)]：表示年轻代当前的内存占用情况，通常 final remark 阶段要尽量运行在年轻代是足够干净的时候，这样可以消除紧接着的连续的几个 STW 阶段。
- [Rescan (parallel) , 0.0160882 secs]：这是整个 final remark 阶段扫描对象的用时总计，该阶段会重新扫描 CMS 堆中剩余的对象，重新从“根对象”开始扫描，并且也会处理对象关联。本次扫描共耗时 0.0160882。
- [weak refs processing, 0.0000512 secs]：第一个子阶段，表示对弱引用的处理耗时为 0.0000512。
- [class unloading, 0.0026681 secs]：第二个子阶段，表示卸载无用的类的耗时为 0.0026681。
- [scrub symbol table, 0.0039342 secs]：最后一个子阶段，表示清理分别包含类级元数据和内部化字符串的符号和字符串表的耗时。
- [1 CMS-remark: 13213K(838912K)]：表示经历了上面的阶段后老年代的内存使用情况。再后面的 `121419K(1027648K), 0.0238344 secs` 表示 final remark 后整个堆的内存使用情况和整个final remark的耗时。

并发清除阶段

```
[CMS-concurrent-sweep-start]
[CMS-concurrent-sweep: 0.006/0.006 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
[CMS-concurrent-reset-start]
[CMS-concurrent-reset: 0.007/0.007 secs] [Times: user=0.01 sys=0.01, real=0.01 secs]
```

该阶段和用户线程并发执行，不会STW，作用是清除之前标记阶段没有被标记的无用对象并回收内存。整个过程分为两个子阶段。

- CMS-concurrent-sweep：第一个子阶段，任务是清除那些没有标记的无用对象并回收内存。后面的参数是耗时，不再多提。
- CMS-concurrent-reset：第二个子阶段，作用是重新设置CMS算法内部的数据结构，准备下一个CMS生命周期的使用。

知识点

我们上面提到CMS收集器会在老年代内存使用到一定程度时就触发垃圾回收，这是因为CMS收集器的一个缺陷导致的这种设定，也就是无法处理“浮动垃圾”，“浮动垃圾”就是指标记清除阶段用户线程运行产生的垃圾，而这部分对象由于没有做标记处理所以在本次CMS收集中是无法处理的。如果CMS是在老年代空间快用完时才启动垃圾回收，那很可能会导致在并发阶段由于用户线程持续产生垃圾而导致老年代内存不够用而导致“Concurrent Mode Failure”失败，那这时候虚拟机就会启用后备预案，临时启用Serial Old收集器来重新进行老年代的垃圾收集，Serial Old是基于“标记-整理”算法的单线程收集器，这样停顿时间就会很长。

这个CMS启动的内存使用阈值可以通过参数 `-XX:CMSInitiatiingOccupancyFranction` 来设置。我通过在命令行输入 `java -XX:+PrintFlagsInitial` 命令查看到该参数的值是 -1，那么 -1 是什么意思呢?

看 JVM 源码可知:

```
product(intx, CMSInitiatingOccupancyFraction, -1,
          "Percentage CMS generation occupancy to start a CMS collection "
          "cycle. A negative value means that CMSTriggerRatio is used")
```

注释里也说了，如果CMSInitiatingOccupancyFraction是个负值，那么CMSTriggerRatio将被用到。

那么具体是如何用到的呢?

```
_cmsGen ->init_initiating_occupancy(CMSInitiatingOccupancyFraction, CMSTriggerRatio);

void ConcurrentMarkSweepGeneration::init_initiating_occupancy(intx io, uintx tr) {
  assert(io <= 100 && tr <= 100, "Check the arguments");
  if (io >= 0) {
    _initiating_occupancy = (double)io / 100.0;
  } else {
    _initiating_occupancy = ((100 - MinHeapFreeRatio) +
                             (double)(tr * MinHeapFreeRatio) / 100.0)
                            / 100.0;
  }
}
```

如果CMSInitiatingOccupancyFraction在0~100之间，那么由CMSInitiatingOccupancyFraction决定。

否则由按 ((100 - MinHeapFreeRatio) + (double)(CMSTriggerRatio * MinHeapFreeRatio) / 100.0) / 100.0 决定。

那么MinHeapFreeRatio，CMSTriggerRatio的初始值是多少?

```
uintx MinHeapFreeRatio                         = 40
uintx CMSTriggerRatio                          = 80
```

即最终当老年代达到 ((100 - 40) + (double) 80 * 40 / 100) / 100 = 92 %时，会触发CMS回收。

参考:

https://www.jianshu.com/p/61bf0e9011c4

https://www.jianshu.com/p/ba768d8e9fec

不要因为知识简单就忽略，不积跬步无以至千里.