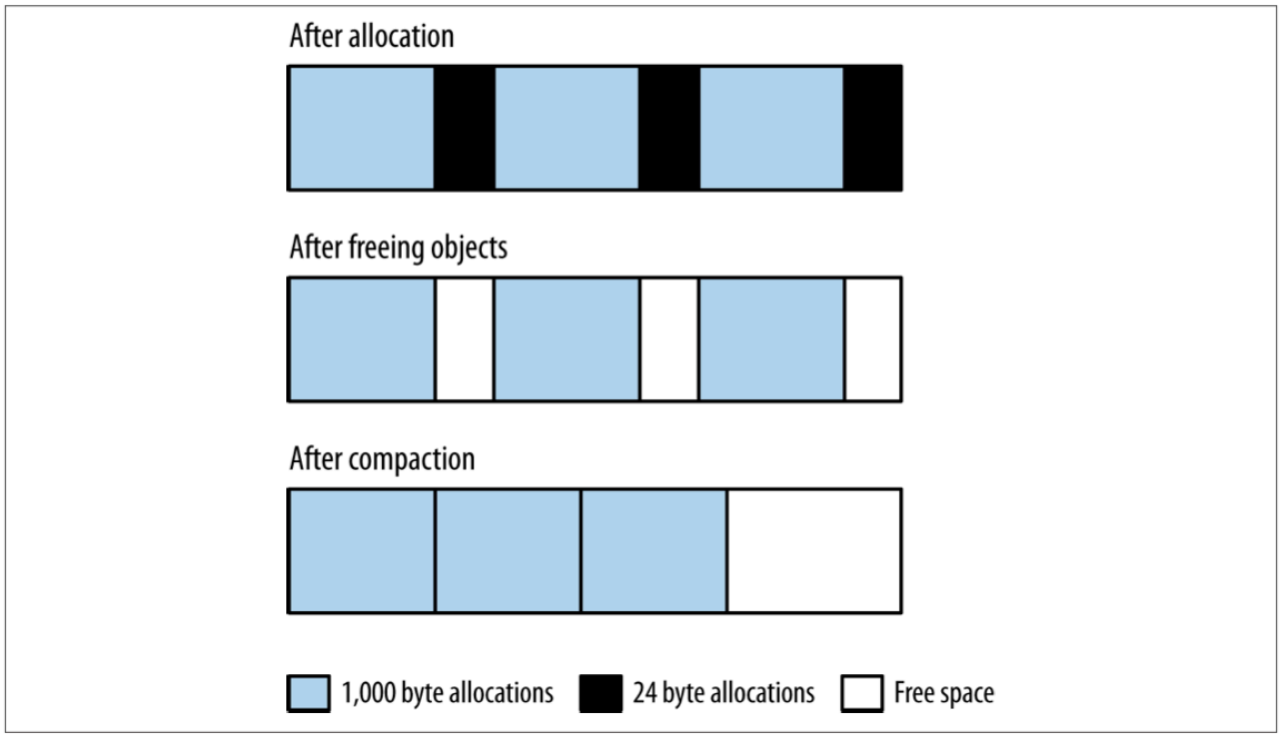
Garbage Collection

关于垃圾收集

Java的特性使开发人员不用面对对象生命周期管理，但发生问题时这却成为了绊脚石。

收集步骤



- 搜寻垃圾对象

  引用计数法
  根搜索算法

- 清理释放内存空间
- 整理内存空间

  避免内存碎片化

- stop-the-world

  GC在运作时需要暂停应用线程以保证清理工作正确执行，尤其是在挪移对象时，对象地址会发生变更。
  这些暂停将会对应用性能产生极大的影响，最小化该影响是GC调优的重点。

分代收集

堆区划分为新生代和老年代，新生代又进一步分割成一块Eden空间和两块Survivor空间。所有的GC都以这个为基础进行垃圾收集。

- IBM：98%的对象都是朝生夕死的

  大部分对象都是临时存在的

Minor GC

回收新生代，频率高，回收成效高，单次执行时间短，stop-the-world。
每次将Eden和一块Survivor空间清空，将其中存活的对象挪至另一块Survivor空间。

优势

- 新生代空间相较于堆区小，意味着处理速度会比处理整个堆区要快。

  同样表明，应用线程被暂停的时长缩短。

- 新生代的回收方式自动整理了其碎片内存

  回收时的复制动作天然按序分配内存

Full GC

全回收（同时也会触发Minor GC），频率低，回收成效低，单次执行时间长（Minor GC10倍以上），stop-the-world。

- GC差异：stop-the-world算法

  算法1：类似Minor GC
  Throughput收集器，老年代满了一次性清理。

  算法2：low-pause
  CMS、G1收集器，后台GC线程持续间隔进行（占用额外CPU），更短暂的单次暂停时间。
  因为内存整理的问题，也会出现Full GC的情况，应当通过调优避免。

垃圾收集算法

JVM提供了四种不同的收集器

Serial收集器

最简单

- 客户端级机器的默认收集器

  运行32位Windows版本JVM/单核CPU的机器

收集算法

- 新生代

  单线程、stop-the-world

- 老年代

  单线程、stop-the-world、完全的碎片整理

配置

- -XX:+UseSerialGC

Throughput（Parallel）收集器

Serial收集器的多线程版本

- 服务器级机器的默认收集器

  Unix多核CPU/运行64位JVM的机器

- 收集算法

  - 新生代

    多线程、stop-the-world

  - 老年代

    多线程、stop-the-world、完全的碎片整理

- 配置

  - -XX:+UseParallelGC
  - -XX:+UseParallelOldGC

- CMS（Concurrent）收集器

  - 为消除Serial/Throughput收集器在Full GC时产生的长停顿问题而设计

  - 收集算法

    - 新生代

      多线程、stop-the-world

    - 老年代

      后台多条线程间歇性对内存进行扫描及清理，仅在必要时进行短暂停。
      总体暂停时间要比其他收集器少的多

      Trade-off：
      CPU利用率上升
      不进行碎片整理（可能会导致Full GC，就像Serial收集器一样。）

  - 配置

    - -XX:+UseParNewGC
    - -XX:+UseConcMarkSweepGC

- G1（Concurrent）收集器

  - 用于处理大内存堆区（大等于4G）并保持最小停顿

  - 收集算法

    将堆区切分为一系列区块，新生代和老年代各由多个区块逻辑上组成。

    - 新生代

      多线程、stop-the-world

- 老年代

  和CMS收集器一样，但由于其内存布局的不同，采用类似新生代的做法（将一个区块的存活数据复制到另一区块，随后清空该区块）可完成部分的碎片整理（较CMS出现Full GC的概率低许多）。

  Trade-off：
  CPU利用率上升

配置

- -XX:+UseG1GC

触发垃圾收集

被动触发

JVM决定GC运作时机

- 新生代满（Minor GC）
- 老年代满（Full GC）

  并发收集器在Heap开始填充时启动扫描

主动触发

大部分场景中不推荐使用，此方式总是触发Full GC，导致应用停顿（所有收集器的停顿效果是一样的）。（例外场景：代码段性能测试前、拉取堆区dump前等。）

- System.gc()

  可通过-XX:+DisableExplicitGC关闭该方法的功能。

- jcmd <process id> GC.run
- jconsole的执行GC按钮

垃圾收集算法选用

取决于应用的属性和性能目标

- 关于Serial收集器

  Serial收集器只适用于那些内存使用量小于100M的应用（这样的场景即使换做并发或者并行收集器的作用都帮助不大）
  ，更多的时候我们是在其他几个收集器之间进行选择。

批处理

- 停顿是个问题

  每次GC停顿都会增加应用任务总体执行耗时

并发收集器 vs. Throughput收集器 @单线程应用场景

停顿不可避免，取平衡。

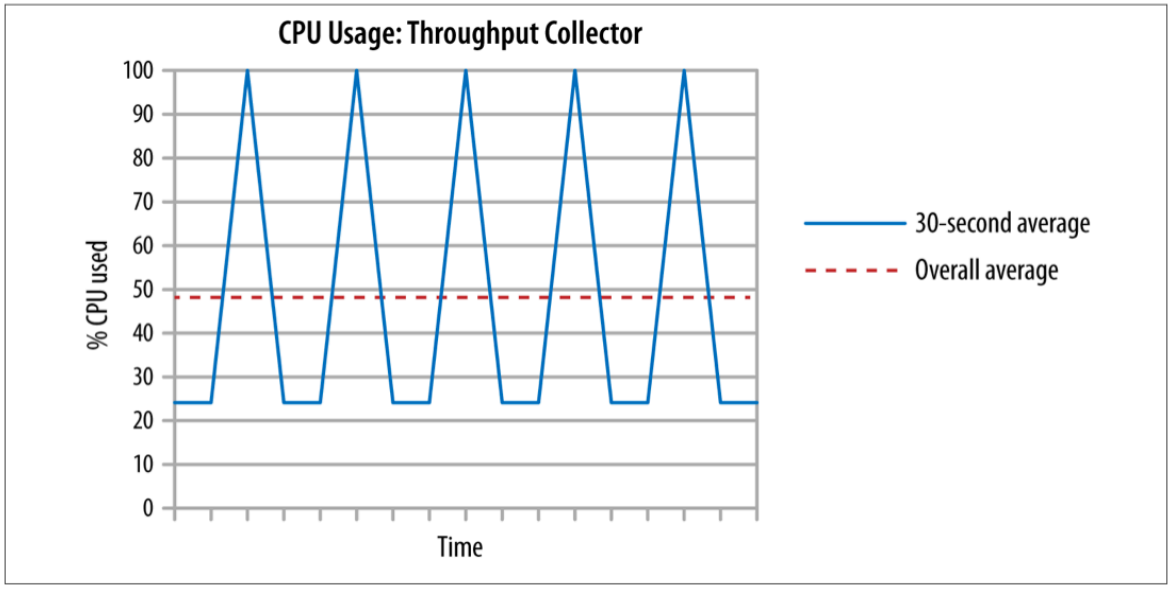
平均CPU利用率：

GC algorithm	4 CPUs (CPU utilization)	1 CPU (CPU utilization)
CMS	78.09 (30.7%)	120.0 (100%)
Throughput	81.00 (27.7%)	111.6 (100%)

CMS效率高，同时需要额外CPU资源支持。

▼ 详细CPU利用率（4CPU场景）

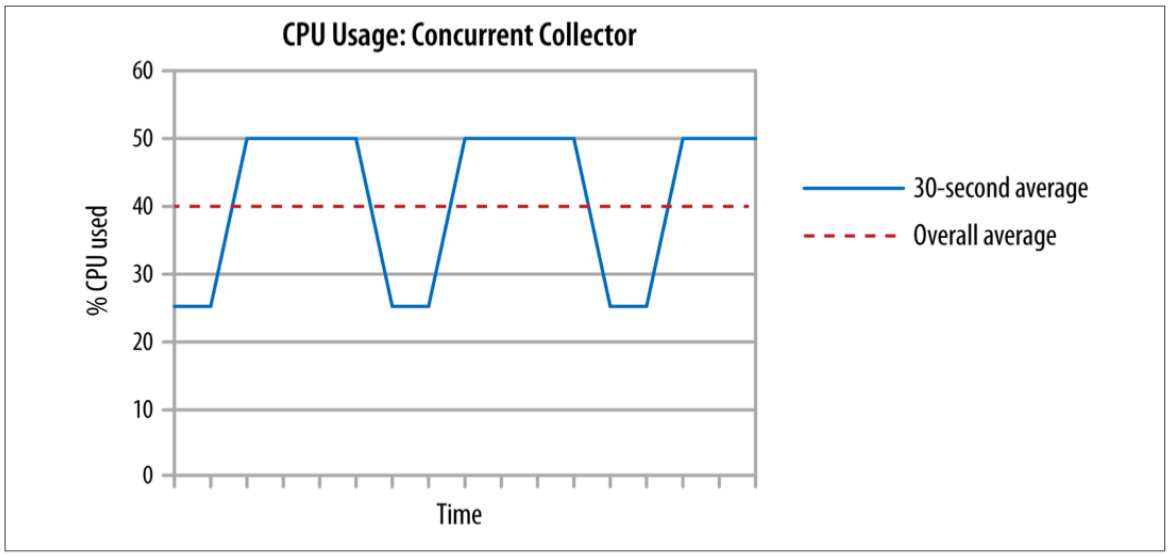
- Throughput收集器运作



25%CPU用于应用线程处理
100%CPU用于GC线程（4条）回收处理，此时应用线程暂停。

监控系统注意过滤

- CMS收集器运作



25%CPU用于应用线程处理
50%CPU用于应用线程与GC线程（1条）共同处理作用，此时应用线程仍工作。

▼ 总结

- 如应用线程已最大化利用CPU资源，则Throughput收集器表现更好。
- 如应用线程仍存有余量的CPU资源，则并发收集器表现更好。

▼ 吞吐量

基本类似批处理的结果，整个系统的性能仍旧和CPU相关。

- 并发收集器 vs. Throughput收集器（4CPU）

Number of clients	Throughput TPS (CPU usage)	CMS TPS (CPU usage)
1	30.43 (29%)	31.98 (31%)
10	81.34 (97%)	62.20 (85%)

1客户端模拟CPU存余的场景，10客户端模拟CPU全占的场景。

10客户端CMS的CPU利用率没有达到100%，是因为没有CPU资源可以供CMS后台线程使用，遭遇了“Concurrent Mode Failure”。此时，JVM只能使用单线程Full GC（只占用25%CPU，拉低了平均使用率）。

▼ CMS和G1之间的选择

- 当Heap容量小（<4G）时，CMS更好。

  CMS算法更简单，在简单的场景下，表现更优。
  CMS清理老年代会全局扫描，如果这个时间过长（老年代已溢出），会引发Concurrent Mode Failure，而这个时间属性直接和老年代的大小相关。

- 当Heap容量大（>=4G）时，G1更好。

  在大内存场景下，G1的分区块算法尽管复杂，但是优势体现的更好。

基础调整

尽管GC算法各异，但它们共享一批基本的配置参数，大多时候对这些参数的调整已能满足应用运行需要。

调整Heap大小

- 权衡

  Heap小了，GC频率高，占用应用执行时间。
  Heap大了，GC频率低，但停顿时间随空间增长而增加。此外，太大会引起频繁内存页交换，大大损耗性能。

- 总大小不要超过物理内存

  多个JVM进程的情况下为Heap总和大小。通常，会预留至少1GB的空间。

- -Xms设置初始值，-Xmx设置最大值。

  如不配置，默认值由系统按规则动态决定。

Operating system and JVM	Initial heap (Xms)	Maximum heap (Xmx)
Linux/Solaris, 32-bit client	16 MB	256 MB
Linux/Solaris, 32-bit server	64 MB	Min (1 GB, 1/4 of physical memory)
Linux/Solaris, 64-bit server	Min (512 MB, 1/64 of physical memory)	Min (32 GB, 1/4 of physical memory)
MacOS 64-bit server JVMs	64 MB	Min (1 GB, 1/4 of physical memory)
Windows 32-bit client JVMs	16 MB	256 MB
Windows 64-bit server JVMs	64 MB	Min (1 GB, 1/4 of physical memory)

Heap大小的增长由应用当前的负载决定，如GC频繁，则堆大小会增长直到频率可容忍或到设定最大值。
如单设定了最大值，应用也会从默认初始值开始按需调整。最佳实践来看，最大值为应用稳定状态下Full GC之后内存占用情况的三倍。
更推荐的做法，确切的将最小值和最大值定义为同一个值能使得GC更有效率，免去自调整的检查。

调整分代

调整新生代和老年代大小

- 权衡

  相对大的新生代带来相对低的回收频率以及相对少的对象进入老年代

- 配置参数（只设置新生代）

  -XX:NewRatio=N
  老年代/新生代比值，默认值2。

  ```
  Initial Young Gen Size = Initial Heap Size / (1 + NewRatio)
  ```

  -XX:NewSize=N
  新生代初始值

  -XX:MaxNewSize=N
  新生代最大值

  -XmnN
  快捷配置，初始值和最大值为同一设定值。

调整永久代

存有类型信息

- 配置参数

  默认值

JVM	Default initial size	Default maximum permgen size	Default maximum metaspace size
32-bit client JVM	12 MB	64 MB	Unlimited
32-bit server JVM	16 MB	64 MB	Unlimited
64-bit JVM	20.75 MB	82 MB	Unlimited

  通常很难定义永久代的最大值，取决于应用自身的情况，因此Metaspace的设计更合理一些。但即使Metaspace默认无限增长，也需要避免内存溢出问题，这和给它设定最大值是同样的，这时更需要关注的是应用本身的问题。

  设置Permgen
  -XX:PermSize=N / -XX:MaxPermSize=N

  设置Metaspace
  -XX:MetaspaceSize=N / -XX:MaxMetaspaceSize=N

  大小的调整会引发Full GC，应用启动时频繁产生这类场景时，应考虑增大初始值设定。

- 回收永久代

  旧的类加载器没有被引用即可被GC回收，包括其加载的类信息。

自适应调整

堆区、分代等的大小都可以在执行时根据制定的策略来自适应调整，JVM会更具既往的执行情况，推测将来的运行水平并作出调整。

优势

- 小应用程序不用担心预先分配内存可能带来的内存浪费情况
- 许多应用并不用为去专门调优Heap大小而烦恼

劣势

- 自适应调整会带来性能损耗

  通过-XX:-UseAdaptiveSizePolicy或将Heap设定为固定大小可关闭自适应调整。

控制并行度

除了Serial收集器，其他GC都采用多线程回收方式。

- -XX:ParallelGCThreads=N

  设置回收线程数

默认线程数

回收时，JVM倾向使用尽可能多得资源来最小化停顿时间。

```
ParallelGCThreads = 8 + ((N - 8) * 5 / 8)
```

默认分配1线程/CPU（上限8），CPU数量大于8时，每5/8个CPU分配一条线程。

- 一机多JVM场景

  应限制所有JVM使用的回收线程数和为公式结果

- 线程数过大问题

  1G Heap with 128CPUs（83 threads！）
  此类场景下应主动降低线程数，如4~6条回收线程的表现会更好。

垃圾回收相关工具

GC log

打印每次GC操作

配置

- -verbose:gc / -XX:+PrintGC

  打印简要信息

- -XX:+PrintGCDetails

  打印详细信息

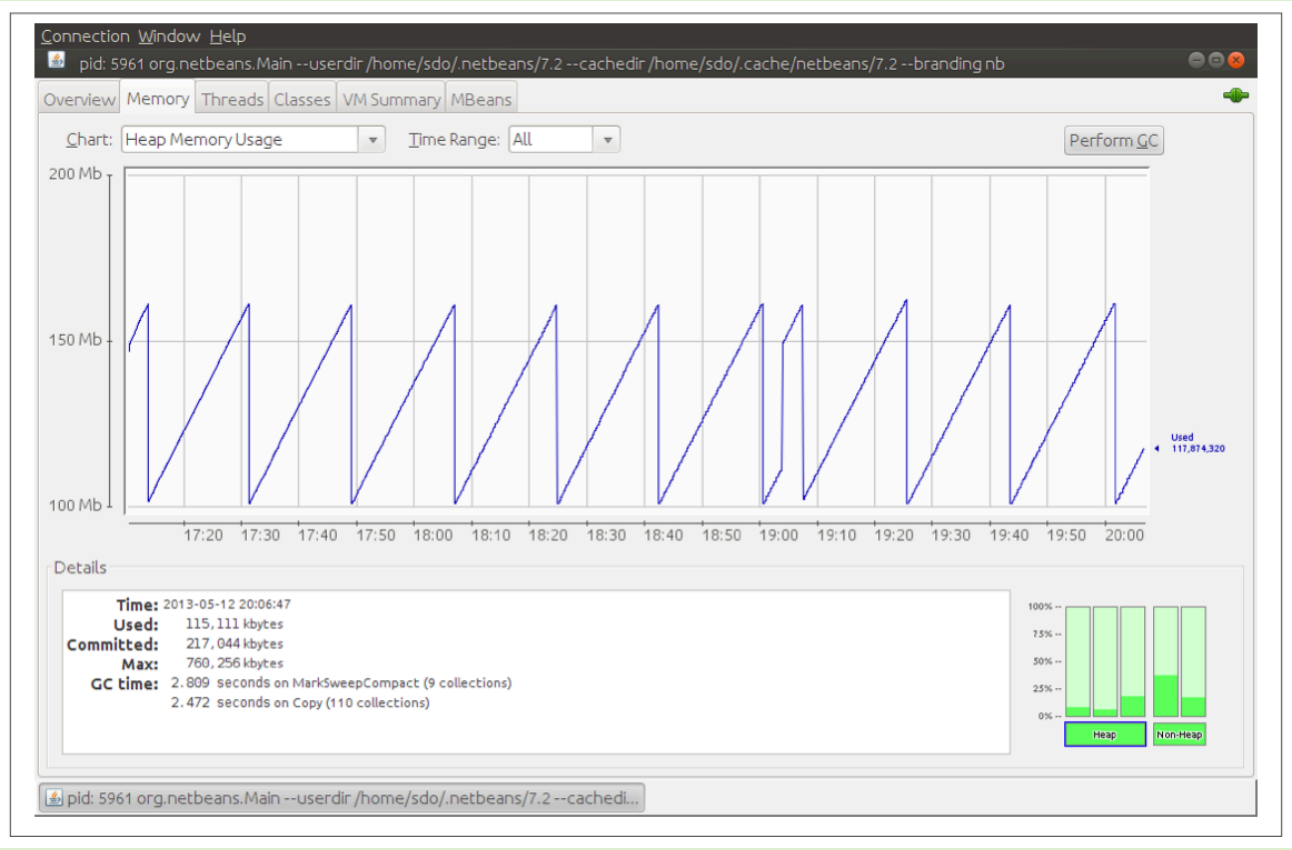
- -Xloggc:filename

  指定打印输出位置，默认为标准输出（控制台）。配合-XX:+UseGCLogFileRotation、-XX:NumberOfGCLogFiles=N、-XX:GCLogFileSize=N可控制文件滚动。

- 图形化工具：GC Histogram

- jconsole

  实时图形化Heap情况监控



- jstat

  命令行监控

```
% jstat -gcutil process_id 1000
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
 51.71   0.00  99.12  60.00  99.93     98    1.985     8    2.397    4.382
  0.00  42.08   5.55  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08   6.32  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08  68.06  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08  82.27  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08  96.67  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08  99.30  60.98  99.93     99    2.016     8    2.397    4.413
 44.54   0.00   1.38  60.98  99.93    100    2.042     8    2.397    4.439
 44.54   0.00   1.91  60.98  99.93    100    2.042     8    2.397    4.439
```

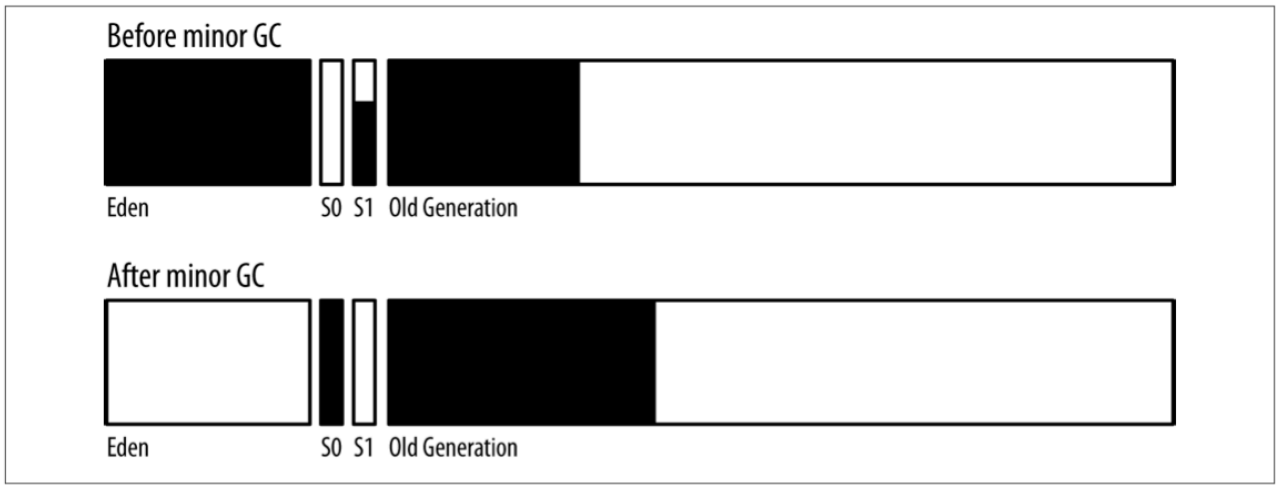
  -gcutil：显示GC消耗时间和各区域内存占比

  1000：监控每隔一秒

进阶理解

理解Throughput收集器

- 新生代回收

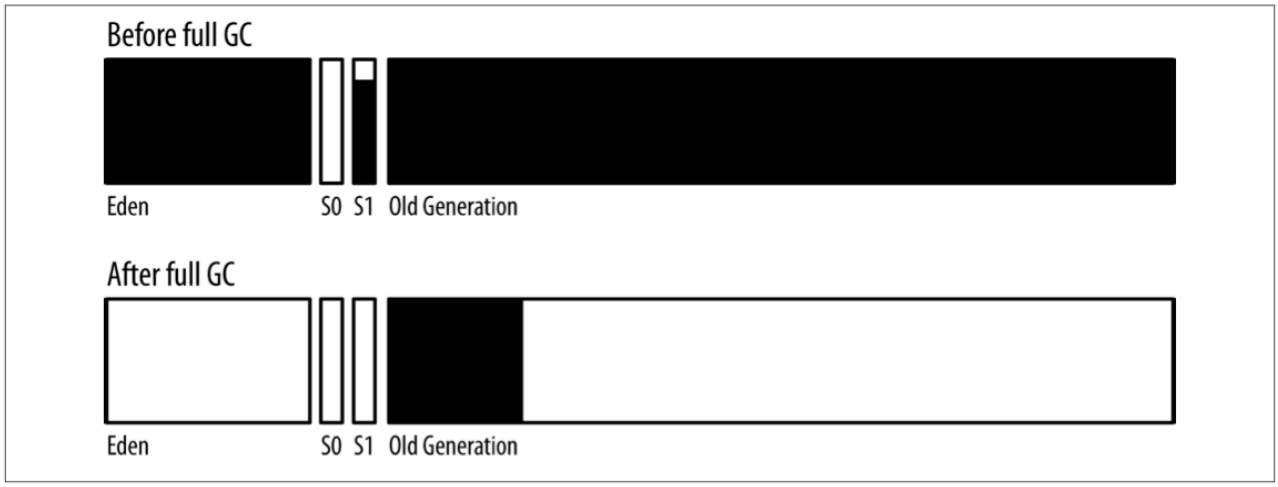


17.806: [GC [PSYoungGen: 227983K->14463K(264128K)] 280122K->66610K(613696K), 0.0169320 secs] [Times: user=0.05 sys=0.00, real=0.02 secs]

关于总耗时：

总耗时约0.02秒，实际在多核CPU情况下，时间总时间（0.05秒）大于这个数。

- 老年代回收



64.546: [Full GC [PSYoungGen: 15808K->0K(339456K)] [ParOldGen: 457753K->392528K(554432K)] 473561K->392528K(893888K) [PSPermGen: 56728K->56728K(115392K)], 1.3367080 secs] [Times: user=4.44 sys=0.01, real=1.34 secs]

关于永久代回收：

永久带的回收也通过Full GC，从GC日志中可以发现是否此区域发生垃圾回收。

▼ 自适应与固定Heap大小调整

Throughput收集器的调优全部是关于对Heap大小和停顿时间的调整，在之间取平衡。

▼ 权衡

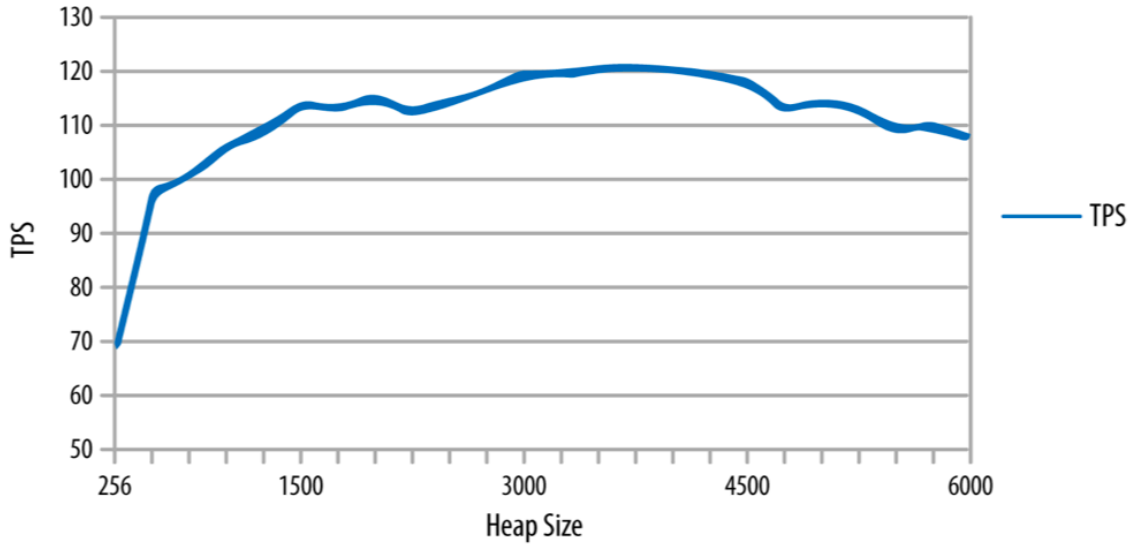
- 应用时间 vs. 空间

  消耗更多的内存空间可以有效降低GC频率，提高应用执行效率。

- 空间 vs. GC时间

  过多的内存空间会增加单次GC处理时间，降低应用执行效率。

- 固定Heap大小设定



~ 1500：吞吐率快速上升，GC频率降低带来的优势明显。
1500 ~ 4500：吞吐率上升乏力，系统瓶颈已不是GC问题。
4500 ~ ：吞吐率开始下降，单次GC时长损耗影响整体性能。

▼ 自适应Heap大小设定（默认）

▼ GC性能指标

Heap的大小会依据预定的GC性能指标来动态决定（-Xms ~ -Xmx）

两指标同时设定时，会先满足-XX:MaxGCPauseMillis，再满足-XX:GCTimeRatio。

- -XX:MaxGCPauseMillis=N（优先，默认不设）

  单次GC最大可容忍停顿时间，同时作用于Minor和Full GC，默认不设定。过小的值将导致老年代过小，从而引发频繁Full GC。

- -XX:GCTimeRatio=N

  非垃圾回收时间与垃圾回收时间的比值，默认值为99（99%时间给应用，GC占用1%）。大部分应用在3%~5%的GC时间时表现也不错。

$$ThroughputGoal = 1 - \frac{1}{(1 + GCTimeRatio)}$$

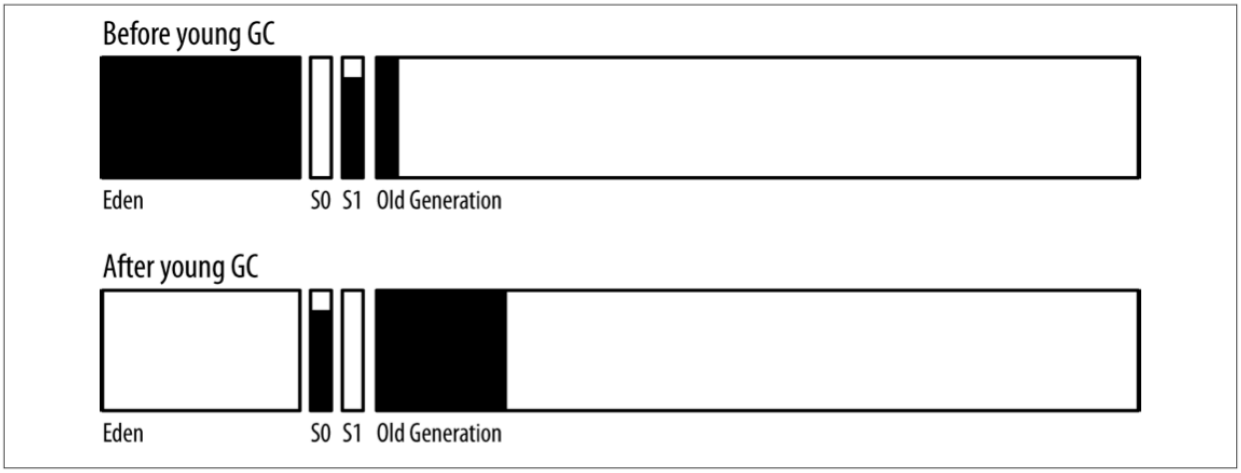
转换一下：

$$GCTimeRatio = \frac{Throughput}{(1 - Throughput)}$$

理解CMS收集器

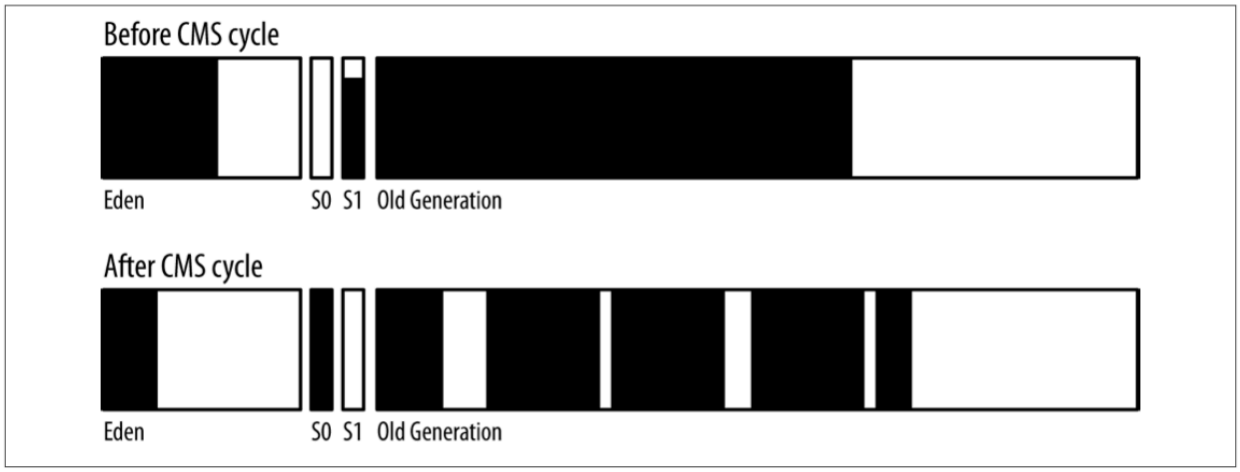
主要操作

- 新生代回收



89.853: [GC 89.853: [ParNew: 629120K->69888K(629120K), 0.1218970 secs] 1303940K->772142K(2027264K), 0.1220090 secs] [Times: user=0.42 sys=0.02, real=0.12 secs]

老年代回收



细分流程

- 初始化标记（stop-the-world）

  89.976: [GC [1 CMS-initial-mark: 702254K(1398144K)] 772530K(2027264K), 0.0830120 secs] [Times: user=0.08 sys=0.00, real=0.08 secs]

- 标记

  90.059: [CMS-concurrent-mark-start]
  90.887: [CMS-concurrent-mark: 0.823/0.828 secs] [Times: user=1.11 sys=0.00, real=0.83 secs]

- 预清理

  90.887: [CMS-concurrent-preclean-start]
  90.892: [CMS-concurrent-preclean: 0.005/0.005 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]

- 二次标记

  90.892: [CMS-concurrent-abortable-preclean-start]
  92.392: [GC 92.393: [ParNew: 629120K->69888K(629120K), 0.1289040 secs] 1331374K->803967K(2027264K), 0.1290200 secs] [Times: user=0.44 sys=0.01, real=0.12 secs]
  94.473: [CMS-concurrent-abortable-preclean: 3.451/3.581 secs] [Times: user=5.03 sys=0.03, real=3.58 secs]
  94.474: [GC[YG occupancy: 466937 K (629120 K)]
  94.474: [Rescan (parallel) , 0.1850000 secs]
  94.659: [weak refs processing, 0.0000370 secs]
  94.659: [scrub string table, 0.0011530 secs] [1 CMS-remark: 734079K(1398144K)] 1201017K(2027264K), 0.1863430 secs] [Times: user=0.60 sys=0.01, real=0.18 secs]

- 清扫

  94.661: [CMS-concurrent-sweep-start]
  95.223: [GC 95.223: [ParNew: 629120K->69888K(629120K), 0.1322530 secs] 999428K->472094K(2027264K), 0.1323690 secs] [Times: user=0.43 sys=0.00, real=0.13 secs]
  95.474: [CMS-concurrent-sweep: 0.680/0.813 secs] [Times: user=1.45 sys=0.00, real=0.82 secs]

- 并发重置

  95.474: [CMS-concurrent-reset-start]
  95.479: [CMS-concurrent-reset: 0.005/0.005 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

- 查看回收成效

  老年代清理没有给出具体的数字表明清理的结果，我们只能通过下一次Minor GC的日志和之前的进行对比：
  89.853: [GC 89.853: [ParNew: 629120K->69888K(629120K), 0.1218970 secs] 1303940K->772142K(2027264K), 0.1220090 secs] [Times: user=0.42 sys=0.02, real=0.12 secs]
  98.049: [GC 98.049: [ParNew: 629120K->69888K(629120K), 0.1487040 secs] 1031326K->504955K(2027264K), 0.1488730 secs]

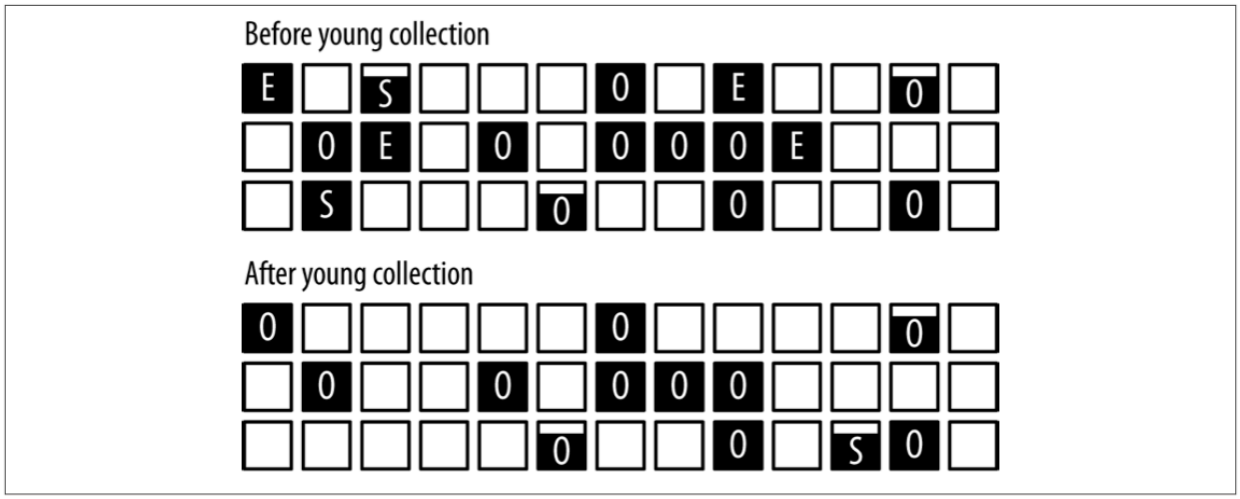
- （如必要）Full GC

理解G1收集器

G1默认将Heap切分为2048个区块，每个区块可以属于新生代或者老年代，属于同一代的区块内存上不必连续。
在老年代回收时，G1只会瞄准那些垃圾多的区块进行清理（Garbage First），所以只花少量的时间清理这些区块。
新生代的回收除了目标内存布局不同，其余基本相同。

主要操作

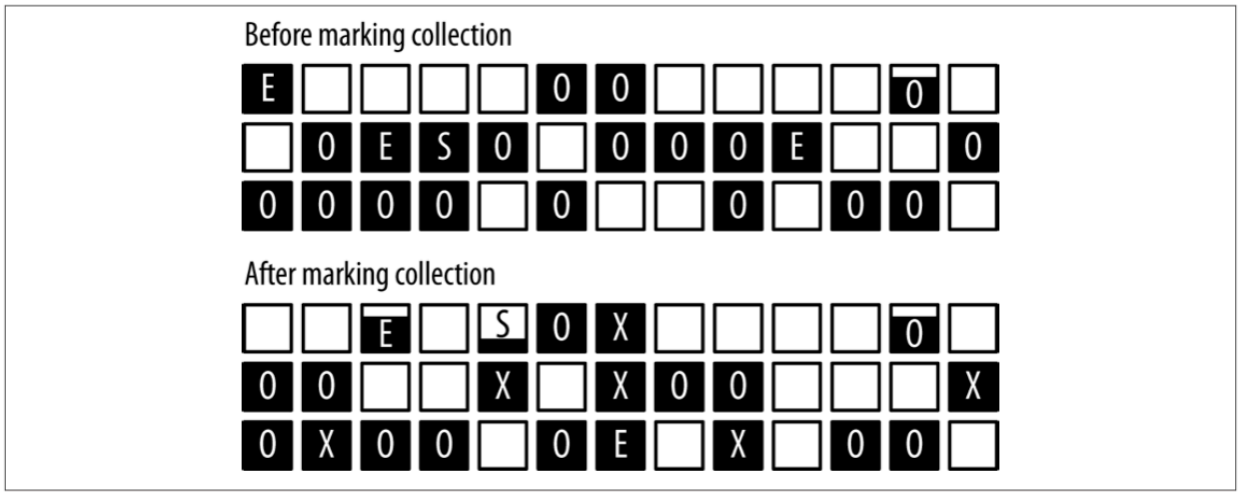
- 新生代回收



23.430: [GC pause (young), 0.23094400 secs] ... [Eden: 1286M(1286M)->0B(1212M) Survivors: 78M->152M Heap: 1454M(4096M)->242M(4096M)] [Times: user=0.85 sys=0.05, real=0.23 secs]

老年代增长

▼ （后台）并发阶段



1、新生代内存容量的变化：并发阶段会发生至少一次的新生代回收

2、老年代部分区块被标记“X”：X区块是被G1认为垃圾量比较大的区块（注意：仍存有数据）

3、老年代内存容量的变化：并发阶段并不清除垃圾，同时，新生代回收造成部分对象晋升。

▼ 细分流程

- 初始标记（stop-the-world）

  50.541: [GC pause (young) (initial-mark), 0.27767100 secs] [Eden: 1220M(1220M)->0B(1220M) Survivors: 144M->144M Heap: 3242M(4096M)->2093M(4096M)] [Times: user=1.02 sys=0.04, real=0.28 secs]

- 扫描根区块

  50.819: [GC concurrent-root-region-scan-start]
  51.408: [GC concurrent-root-region-scan-end, 0.5890230]

- 并发标记

  111.382: [GC concurrent-mark-start]
  ....
  120.905: [GC concurrent-mark-end, 9.5225160 sec]

- 二次标记（stop-the-world）

  120.910: [GC remark 120.959: [GC ref-PRC, 0.0000890 secs], 0.0718990 secs] [Times: user=0.23 sys=0.01, real=0.08 secs]

- 常规清理（stop-the-world）

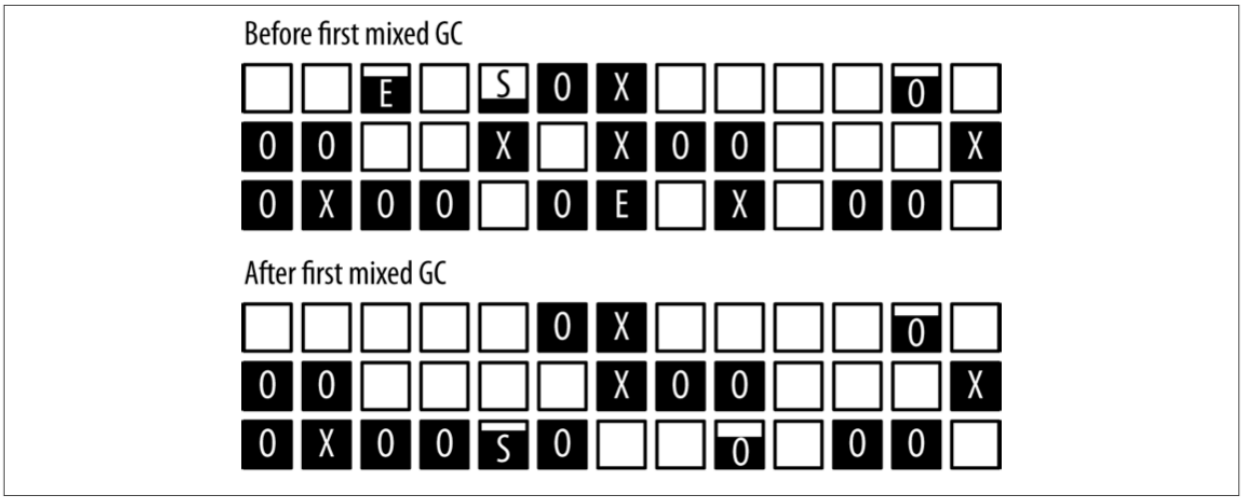
  120.985: [GC cleanup 3510M->3434M(4096M), 0.0111040 secs] [Times: user=0.04 sys=0.00, real=0.01 secs]

- 额外清理

  120.996: [GC concurrent-cleanup-start]
  120.996: [GC concurrent-cleanup-end, 0.0004520]

- 混合回收

  常规新生代回收，同时也会回收一些被标记区块。
  混合回收持续运作直到所有标记区块被回收。



79.826: [GC pause (mixed), 0.26161600 secs] .... [Eden: 1222M(1222M)->0B(1220M) Survivors: 142M->144M Heap: 3200M(4096M)->1964M(4096M)] [Times: user=1.01 sys=0.00, real=0.26 secs]

- （如必要）Full GC