JVM GC

GC Description

- What's GC?
  - GC用于跟踪内存中的对象，并回收那些不再被其他对象引用的对象。
  - 内存中的对象类型
    - 活动对象：即当前正在其他对象引用的对象。
    - 非活动对象：这类对象不再被其他对象所引用，是孤立的对象。这类对象可以被回收，回收的堆空间用于分配给其它新创建的对象。

GC Description(2)

- GC何时会被触发 ？
  - 系统空闲
    - GC线程的优先级低于系统应用线程，当系统中没有应用线程执行时，GC会被触发。
  - 堆空间内存不足
    - 当堆空间的内存不足以创建新对象时，GC会被触发。如果第一GC仍不能获得足够的空间，第二次GC将被触发，如果这一次仍无法获取足够的空间， “Out of memory” 将被抛出。

GC Description(3)

- 影响GC执行时间、频度的因素
  - JVM 堆(heap)空间的大小
    - 堆空间设置偏大，完全GC执行比较耗时，但执行频率会降低。
    - 堆空间设置恰好符合应用内存需求，完全GC执行很快，但执行会变得更频繁。

Heap Description

- 什么是堆(heap)空间
  - 堆是java程序中对象存活的地方，其中包括：
    - 活动对象
    - 非活动对象，这类对象不再为应用程序中的任何指针能够到达。
    - 剩余内存
  - 堆空间中包含三种区域：
    - 新生代(young generation)
    - 旧生代(tenured generation)
    - 永生代(permanent generation)

Heap Description(2)

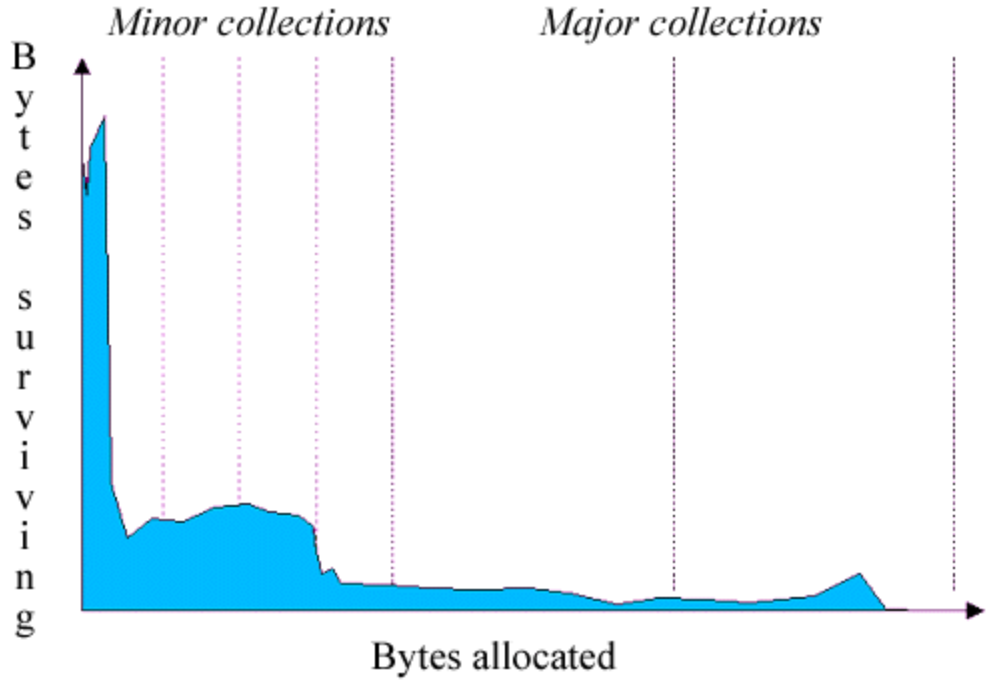
- 新生代(young generation)
  - 新生代被分为两块: Eden、Survivor spaces
    - Eden是为新对象分配的地方，很多对象分配后就变成非活动对象,即垃圾对象。这类对象具有“infant mortality”(幼儿死亡率)。如：方法体中的临时对象。
    - Survivor spaces也被称为两片生存空间，其中一片要保证任何时刻是空的,并作为下一个空间的目的地。当GC发生的时候，Eden中的存活对象被移入下一片空间。对象在生存空间之间移动，直到它们老化(达到存活时间阀值)，然后被移入旧生代。
    - Eden中对象满的时候，发生一次小收集(minor collection)，小收集执行时间取决于Eden中对象的infant mortality。infant mortality高，执行就会很快。

Heap Description(3)

- 旧生代(tenured generation)
  - 该区域用于存放那些生命周期比较长的对象，Eden中的活动对象经过minor collection后，被复制到两片生存空间，当两片生存空间中的对象老化时，这些对象被移入旧生代。
  - 旧生代中对象满的时候，发生一次大收集(major collection)，因为收集时要涉及所有存活对象，所以大收集的速度相对于小收集要慢很多。

Heap Description(4)

- 下图展示了典型应用中，对象生存周期

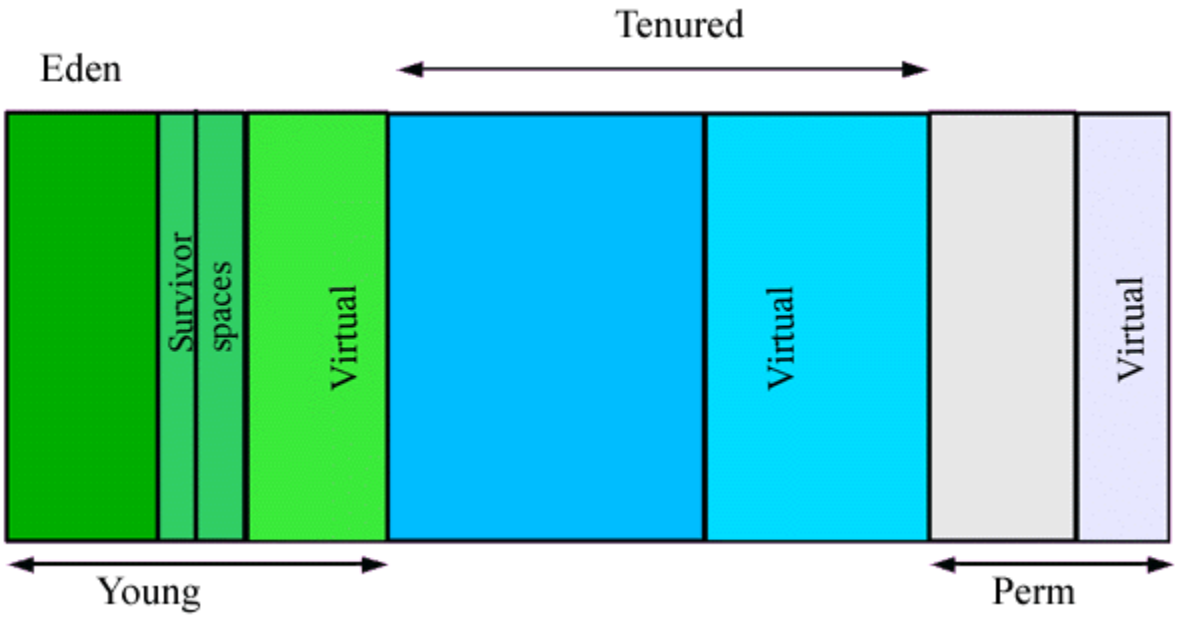


Heap Description(5)

- 永生代(permanent generation)
  - 这个代比较特别，它负责保存反射对象。这些数据是虚拟机所需的数据，用来描述在Java语言中没有等同物的对象。例如，描述类与方法的对象存储在永生代中。

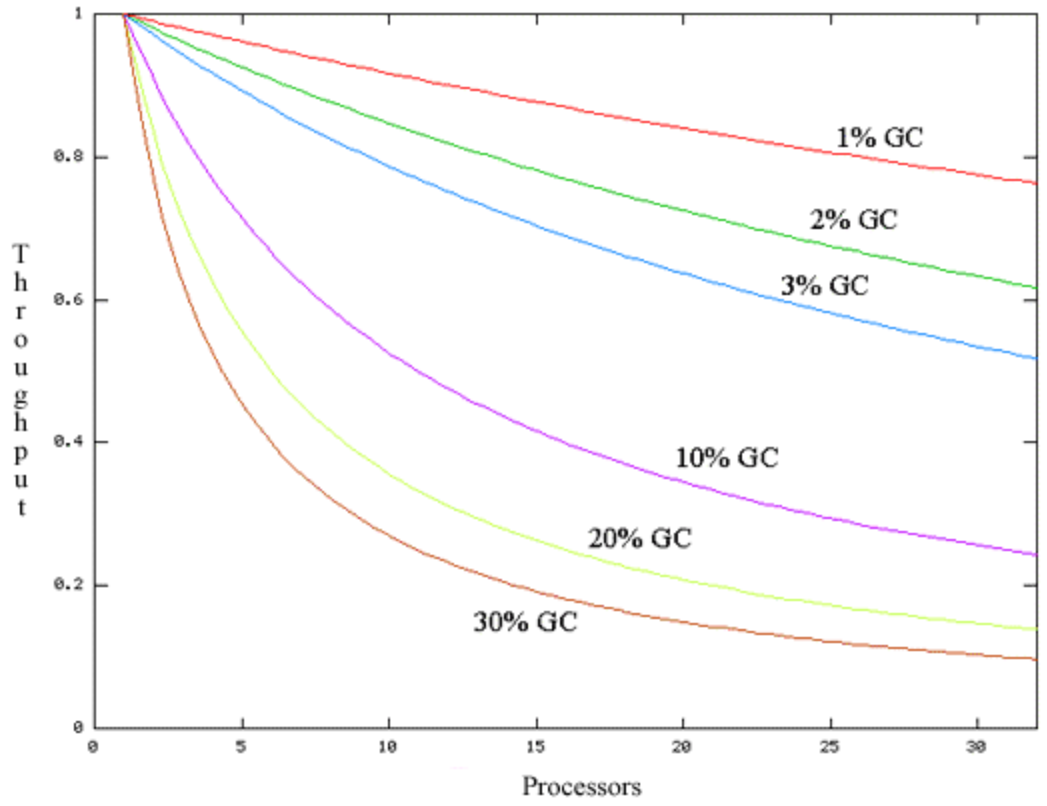
Heap Description(6)

- 分代[illegible]结构



Performance Tuning

- 下图显示了GC性能对整个系统性能的影响



Performance Tuning(2)

- 从上图可以看出:
  - 如果系统花费在GC上的时间为1%，一旦系统跨越32个处理器时，系统的吞吐量(Throughput)将减少>20%
  - 如果系统花费在GC上的时间为10%，一旦系统跨越32个处理器时，系统的吞吐量(Throughput)将减少高达75%

Note:所谓throughput，即系统用于处理非GC时间的比例。

Performance Tuning(3)

- Performance measurement
  - ***Pause:*** the times when an application appears unresponsive because garbage collection is occurring.
  - ***Footprint:*** the working set of a process, measured in pages and cache lines .
  - ***Promptness*** :the time between when an object becomes dead and when the memory becomes available.

Performance Tuning(4)

- How to leverage these measurement
  - a very large *young* generation may maximize throughput, but does so at the expense of footprint, promptness, and pause times.
  - a small *young* generation can minimize pause times at the expense of throughput

Performance Tuning(5)

- Example output of GC

```
[memory ] GC strategy: parallel
[memory ] heap size: 1433600K, maximal heap size: 1433600K
[memory ] <s>-<end>: GC <before>K-><after>K (<heap>K), <pause> ms
[memory ] <s/start> - start time of collection (seconds since jvm start)
[memory ] <end>     - end time of collection (seconds since jvm start)
[memory ] <before>  - memory used by objects before collection (KB)
[memory ] <after>   - memory used by objects after collection (KB)
[memory ] <heap>    - size of heap after collection (KB)
[memory ] <pause>   - total pause time during collection (milliseconds)
[memory ] 169.469-170.047: GC 1433600K->171510K (1433600K), 578.000 ms
[memory ] 302.484-302.812: GC 1433600K->143686K (1433600K), 328.000 ms
[memory ] 471.125-471.469: GC 1433600K->164180K (1433600K), 338.222 ms
[memory ] 615.750-616.203: GC 1433600K->174907K (1433600K), 440.966 ms
[memory ] 761.969-762.312: GC 1433600K->188344K (1433600K), 343.000 ms
[memory ] 882.687-883.031: GC 1433600K->200600K (1433600K), 344.000 ms
```

Performance Tuning(6)

- To get detailed info of GC
  - -XX:+PrintGCDetails

  Example:

  [GC [**DefNew**: 64575K->959K(64576K), 0.0457646 secs] 196016K->133633K (261184K), 0.0459067 secs]]

  - -XX:+PrintGCTimestamps

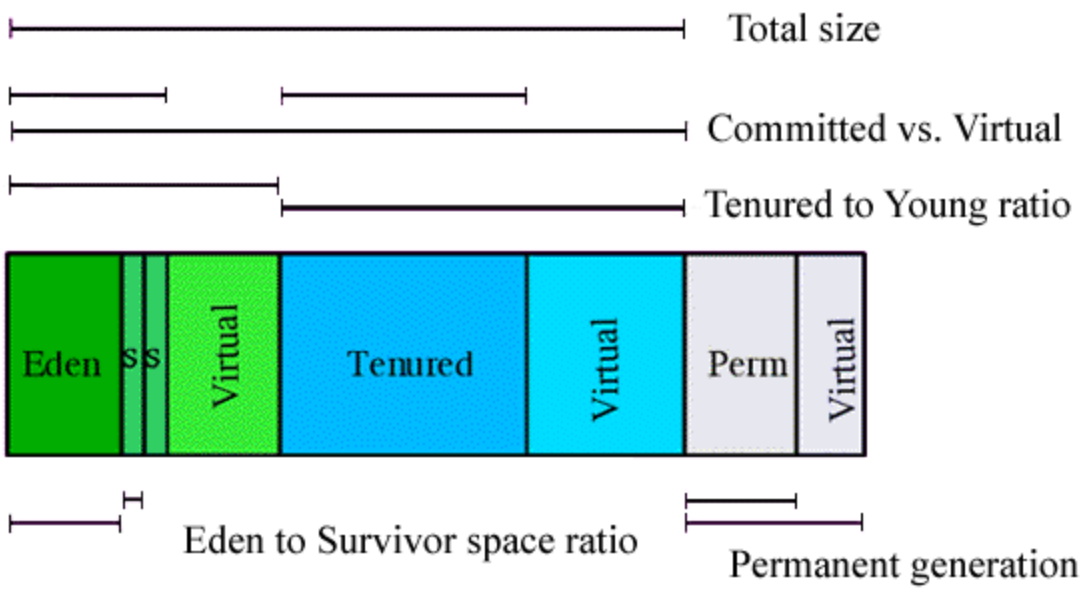
  Example:

  111.042: [GC 111.042: [DefNew: 8128K->8128K(8128K), 0.0000505 secs] 111.042: [Tenured: 18154K->2311K(24576K), 0.1290354 secs] 26282K->2311K(32704K), 0.1293306 secs]

  Note: the output format of above flag is different from each version of JVM.

Performance Tuning(7)

- Generation size diagram



Performance Tuning(8)

- Committed VS Virtual
  - 每个代分成committed和virtual，JVM初始化的时候，所有heap被reserve。如果-Xms小于-Xmx，不是所有reserved的空间都立刻被commit，没有被commit的空间标志为virtual。
  - 随着应用对内存需求的增长，commited的heap慢慢会达到-Xmx指定的数值。

Performance Tuning(9)

- Parameters which affect total heap size
  - -XX:MinHeapFreeRatio=
  - -XX:MaxHeapFreeRatio=
  - -Xms
  - -Xmx

Note: JVM每次GC的时候，都会增加或收缩heap的大小，以保证free的空间的百分比在-XX:MinHeapFreeRatio和-XX:MaxHeapFreeRatio之间。同时建议-Xms和-Xmx设为相同值。

Performance Tuning(10)

- Parameters which affect young generation size
  - -XX:NewRatio=
  - -XX:NewSize=
  - -XX:MaxNewSize=
  - -XX:SurvivorRatio=

Note: young generation的设置大点，小收集的执行频率降低，但大收集的执行频率会升高，可以依据应用中的对象生命周期进行young generation大小的调整。young generation的大小，可以由NewRatio控制，如NewRatio=4，这意味着young generation占整个heap的1/5。NewSize和MaxNewSize分别代表young generation的上下边界值。SurvivorRation用于指定survivor和eden的比例，如SurvivorRation=5，则意味着每个survivor占整个young generation的1/7(因为young generation中有两个survivor)。

Types of Collectors

- Collectors in J2SE platform, version 1.4.2
  - ***Default*** collector
  - ***Throughput*** collector
  - ***Concurrent*** low pause collector
  - ***Incremental*** low pause collector

Note: 之前的介绍，都是基于default collector的。通常情况下，在使用其他三种collector之前，推荐使用default collector。如果调整heap size仍不满足系统某些要求，则根据这些需求，再选择相应的collector。

Throughput collector

- What's Throughput collector
  - Throughput collector在young generation中使用并行版本收集器，而tenured generation中仍使用default collector。
- When to use Throughput collector
  - Throughput collector适用于应用跨越多CPU的场合。Default collector进行minor收集的时候，使用单线程，而Throughput collector使用多线程进行并行收集，这样可以减少收集时间。
- How to use Throughput collector
  - Add flag *-XX:+**UseParallelGC*** to command line。

Throughput collector(2)

- Performance improvement
  - 对于单CPU的场合，因为并行收集的额外负荷(同步等)， throughput collector性能不但得不到提升，反而不如default collector。
  - 对于双CPU的场合，throughput collector的性能基本和default collector相当。
  - 对于3CPU及以上的场合，throughput collector在收集性能上能得到一定的提升。

Throughput collector(3)

- GC threads in Throughput collector
  - 默认情况下，throughput collector中GC线程数和系统的CPU数相当。但GC线程数可通过下面的flag进行控制，

    -XX:ParallelGCThreads=
- GC statistics
  - Through collector收集时的统计数据包括：收集时间、收集速率、剩余heap大小等。可以根据这些数据调整young generation、tenured generation的大小来满足系统要求。下面的flag用于控制这些统计数据是否输出，默认为on，

    -XX:+UseAdaptiveSizePolicy

Throughput collector(4)

- Self tuning by Throughput collector
  - -XX:+AggressiveHeap 自检物理机器的内存、CPU数，然后自行调整一些参数，以优化那些长期运行、大量内存分配需求的任务。这个flag比较适合那些大内存、多CPU的系统，
  - 使用AggressiveHeap，必须保证物理内存>256M，初始化的heap大小依赖于物理内存的大小。后续内存的使用，会尽可能多的使用物理内存 (i.e., it attempt to use heaps nearly as large as the total physical memory)。

Throughput collector(5)

- Possible shortcomings
  - 因为多线程参与minor收集，每个线程都保留一部分tenured generation，用于复制数据，所以在从young generation复制对象至tenured generation时，可能引起heap中碎片。
  - 碎片产生的可能性可以通过减少GC线程数和提升tenured generation的大小来降低。

Concurrent Low Pause Collector

- What's Concurrent Low Pause Collector
  - Concurrent collector使用单线程进行tenured generation的收集，即major collection，但该GC线程和应用threads并行，这样做的目的是降低GC引起的应用暂停时间，其代价是CPU资源。
- When to use Concurrent Low Pause Collector
  - Concurrent collector适用于那些具有很多long-lived对象，即tenured generation比较大，且期望获得更短的GC引起的暂停时间的应用场合。它同样适用于多CPU的场合，对于单CPU的系统，它并不能达到预期的效果。

Concurrent Low Pause Collector(2)

- How to use Concurrent Low Pause Collector
  - To use concurrent collector，you can add follow flag to command line： -XX:+UseConcMarkSweepGC。
- Two pauses during the collection
  - Concurrent collector在收集过程中将引起两次应用threads暂停。一次发生在collection开始至collection中间，这次暂停时间相对较短，用于标记(mark)存活对象；另一次暂停时，多个线程并行的执行收集工作，用于标记那些在两次暂停以外由于并发收集而错失的对象，称作remark；剩余的收集由单一线程完成，收集线程和应用线程同时运行，这时不会引起应用的暂停。

Concurrent Low Pause Collector(3)

- Full collections
  - Concurrent collector使用单线程进行tenured generation收集，它和应用程序线程同时运行，以保证在tenured generation满之前收集成功。通常情况下，它可以正常工作。如果收集成功之前，tenured generation已经充满，那么所有应用线程将停止，进行full collection，这种情况下，gc.log中会有full collection的提示，此时可以根据实际情况，调整tenured generation的大小。
- Floating collections
  - GC收集过程中，需要查找heap中的存活对象，因为GC和应用线程并行，某些被GC认定为存活的对象在GC结束前已经不再被其他对象引用，这些对象称为floating garbage。Floating garbage的多少取决于GC时间。通常在出现floating garbage的情况下，将以加大20%的tenured generation的大小，以消除后续floating garbage的出现。

Concurrent Low Pause Collector(4)

- Concurrent phases
  - Concurrent collector收集过程中大概分成initial mark、remark、concurrent mark、sweeping四个阶段。concurrent mark介于initial mark、remark之间，这个阶段，GC和应用线程同时运行，它会占用系统资源。在remark之后，进入sweeping阶段，这个阶段进行dead对象收集，这个阶段也会占用系统资源。sweeping 阶段后，collector休眠至下次major collection。
- Parallel Minor Collection Options
  - 默认情况下，在多CPU的环境下，UseParNewGC 是打开的。如果在使用UseParNewGC的情况下，可以通过打开CMSParallelRemarkEnabled 来降低remark时间，如下:

    -XX:+CMSParallelRemarkEnabled

Concurrent Low Pause Collector(5)

- Example GC output for Concurrent collector
  - // beginning of concurrent marking
  - [GC [1 CMS-initial-mark: 13991K(20288K)] 14103K(22400K), 0.0023781 secs]
  - [GC [DefNew: 2112K->64K(2112K), 0.0837052 secs] 16103K->15476K(22400K), 0.0838519 secs]
  - ...
  - [GC [DefNew: 2077K->63K(2112K), 0.0126205 secs] 17552K->15855K(22400K), 0.0127482 secs]
  - //end of concurrent marking
  - [CMS-concurrent-mark: 0.267/0.374 secs]
  - [GC [DefNew: 2111K->64K(2112K), 0.0190851 secs] 17903K->16154K(22400K), 0.0191903 secs]
  - //beginning of preparation for remarking and is done concurrently
  - [CMS-concurrent-preclean: 0.044/0.064 secs]
  - //beginning of remarking
  - [GC[1 CMS-remark: 16090K(20288K)] 17242K(22400K), 0.0210460 secs]
  - [GC [DefNew: 2112K->63K(2112K), 0.0716116 secs] 18177K->17382K(22400K), 0.0718204 secs]
  - [GC [DefNew: 2111K->63K(2112K), 0.0830392 secs] 19363K->18757K(22400K), 0.0832943 secs]
  - ...
  - [GC [DefNew: 2111K->0K(2112K), 0.0035190 secs] 17527K->15479K(22400K), 0.0036052 secs]
  - //end of remarking and beginning of sweeping
  - [CMS-concurrent-sweep: 0.291/0.662 secs]
  - [GC [DefNew: 2048K->0K(2112K), 0.0013347 secs] 17527K->15479K(27912K), 0.0014231 secs]
  - //preparation for next collection
  - [CMS-concurrent-reset: 0.016/0.016 secs]
  - [GC [DefNew: 2048K->1K(2112K), 0.0013936 secs] 17527K->15479K(27912K), 0.0014814 secs]

Incremental Low Pause Collector

- What's Incremental Low Pause Collector
  - Incremental collector在每次minor collection的时候做一部分major collection，这样做的目的是避免一次major collection占用太长时间，也就是说将一次长时间的GC划分为多次、短时间的GC。使用Incremental collector的JVM，其young generation的收集使用default collector的收集方式，而tenured generation的收集则使用incremental collector。但incremental collection过程中也会夹杂着非incremental collection，以避免out of memory的情况。

Incremental Low Pause Collector(2)

- When to use Incremental collector
  - Incremental collector比较适合于一些tenured generation比较大(具有大量长期存活的对象)，young generaion比较小(幼儿死亡率比较高)的场合。对于这样的场合，因为young generation比较小，而且幼儿死亡率高，这样GC工作比较频繁，收集也充分，每次可以进行一部分major collection。注意：它只适用于单CPU的场合。
- How to use Incremental collector
  - 可以在命令行中加入如下option，

    -Xincgc

Note： Xincgc 不能和下面两个option同时使用，

-XX:+UseParallelGC、

-XX:+UseParNewGC

Incremental Low Pause Collector(3)

- Example GC output for Incremental collector
  - // pure Incremental collection
  - [GC [DefNew: 2074K->25K(2112K), 0.0050065 secs] [Train: 1676K->1633K(63424K), 0.0082112 secs] 3750K->1659K(65536K), 0.0138017 secs]
  - //full collection in Incremental collection, MSC means
  - //mark-sweep-compact
  - [GC [DefNew: 2049K->2049K(2112K), 0.0003304 secs] [Train MSC: 61809K->357K(63424K), 0.3956982 secs] 63859K->394K(65536K), 0.3987650 secs] ...

Note: above output can be got with -XX:+PrintGCDetails

GC log an analyzation

- Format of GC log
  - [GC [<collector>: <starting occupancy1> -> <ending occupancy1>, <pause time1> secs] <starting occupancy3> -> <ending occupancy3>, <pause time3> secs]

Collector：internal collector for minor collection
starting occupancy1：young generation size before the collection
ending occupancy1： young generation size after the collection
pause time1：pause time for minor collection
starting occupancy3：entire heap size before the collection
ending occupancy3： entire heap size after the collection
pause time3：pause time for the entire garbage collection

How to get detailed GC log

- IBM JDK
  - -verbose:gc -Xverbosegclog:<path_GC_log_file_name>
- HP JDK
  - -Xverbosegc[:help]|[0|1][:file=[stdout|stderr|<filename>]] -Xloggc
- Sun JDKBEA JRockit
  - -verbose:gc –Xloggc:logfile
- BEA JRockit
  - -verbose:gc -Xverboselog:logfile

How to get detailed GC log(2)

- To get detailed GC info
  - -XX:-PrintGC
  - -XX:+PrintGCDetails
  - -XX:+PrintGCApplicationConcurrentTime
  - -XX:+PrintGCApplicationStoppedTime
  - -XX:+PrintGCTimeStamps