浅谈 Java 虚拟机垃圾回收

郑帆

Base Java7

Performance Basics

- Responsiveness
- Throughput

Responsiveness

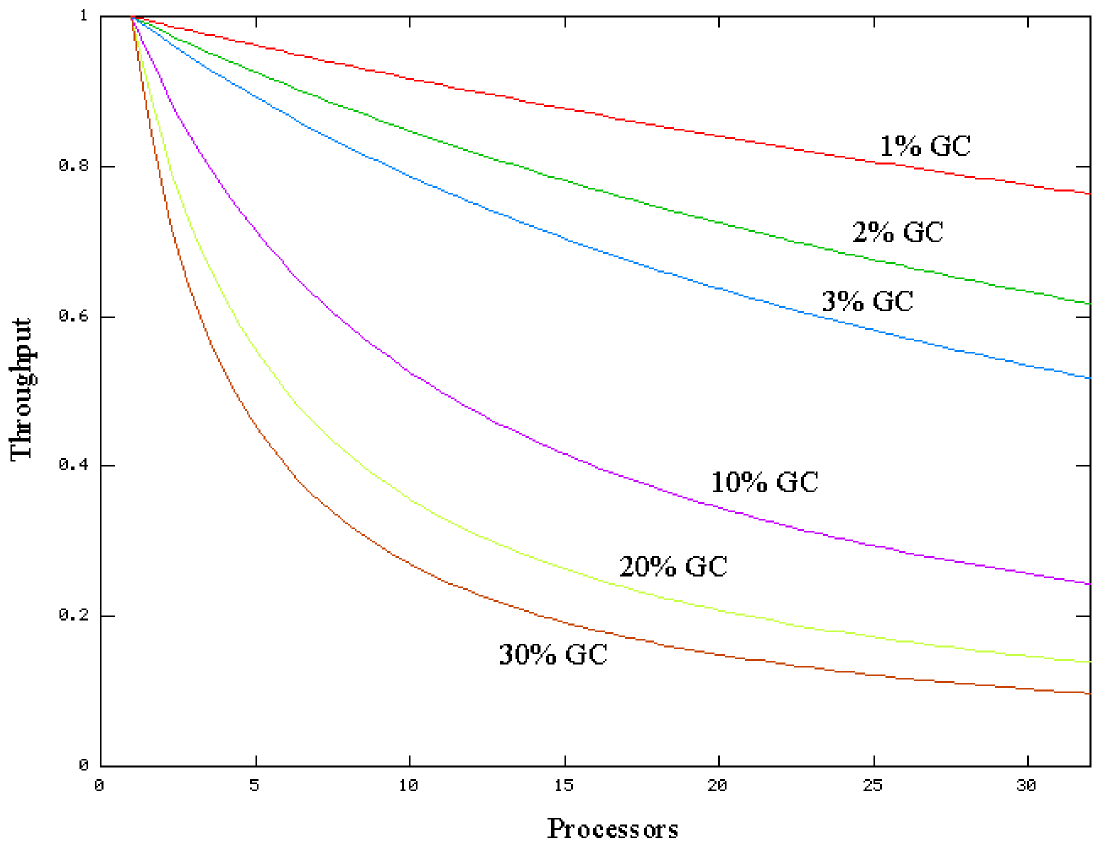
The focus is on responding in short periods of time.

(large pause times are not acceptable)

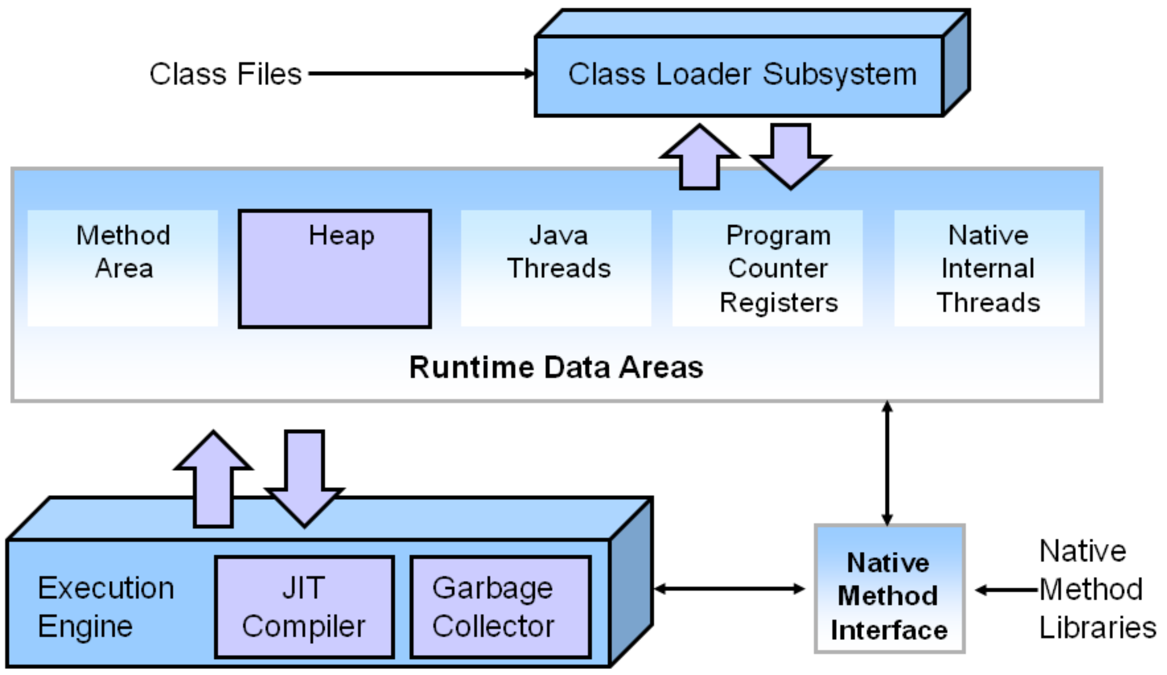
Throughput

high throughput applications focus on benchmarks over longer periods of time.

(quick response time is not a consideration)


1
0.8
0.6
0.4
0.2
0
0
5
10
15
20
25
30
Throughput
Processors
1% GC
2% GC
3% GC
10% GC
20% GC
30% GC

Key HotSpot JVM Components



垃圾回收机制中的算法

- 引用计数法 (Reference Counting)
- 根搜索算法(Tracing Root)
- 标记-清除算法 (Mark-Sweep)
- 复制算法 (Copying)
- 标记-压缩算法 (Mark-Compact)

引用计数法 (Reference Counting)

例:

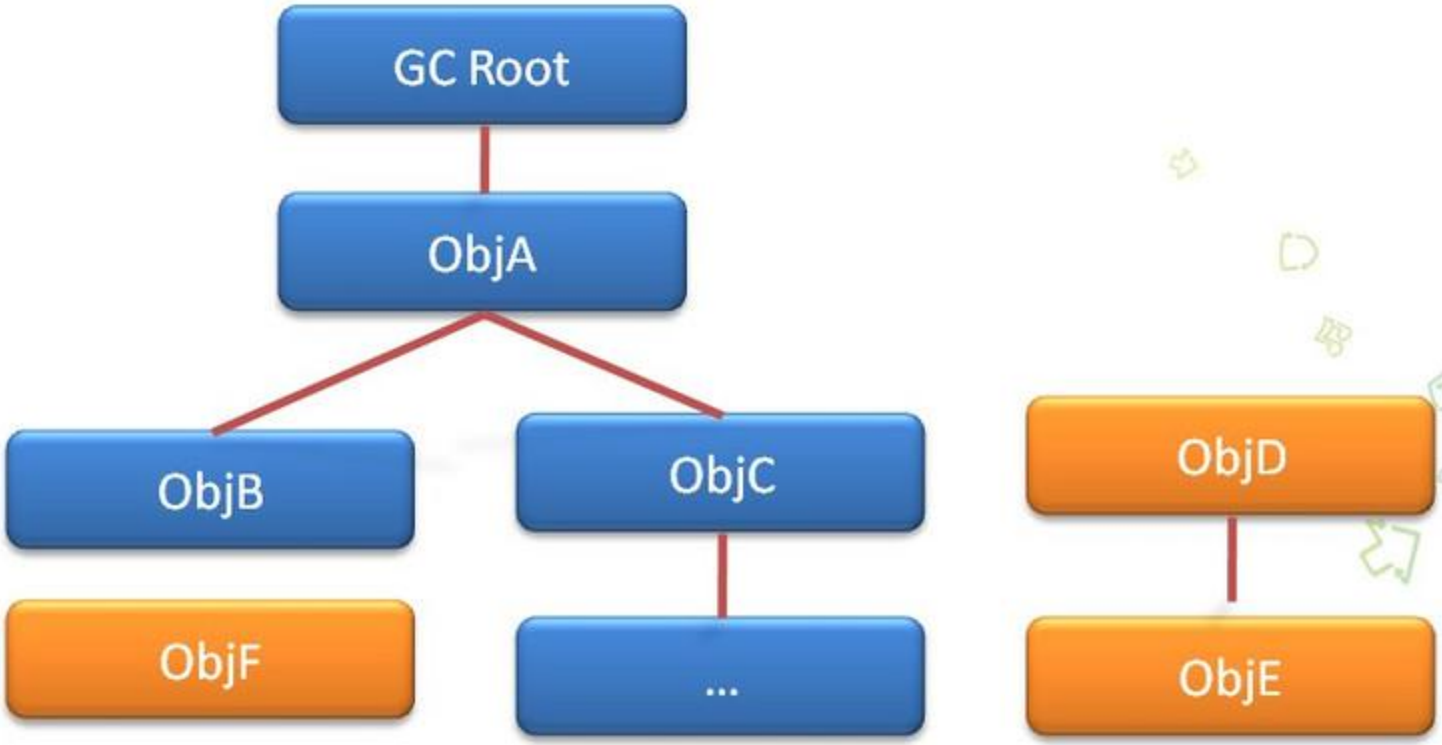
　　对于一个对象 A，只要有任何一个对象引用了 A，则 A 的引用计数器就加 1，当引用失效时，引用计数器就减 1。只要对象 A 的引用计数器的值为 0，则对象 A 就不可能再被使用。

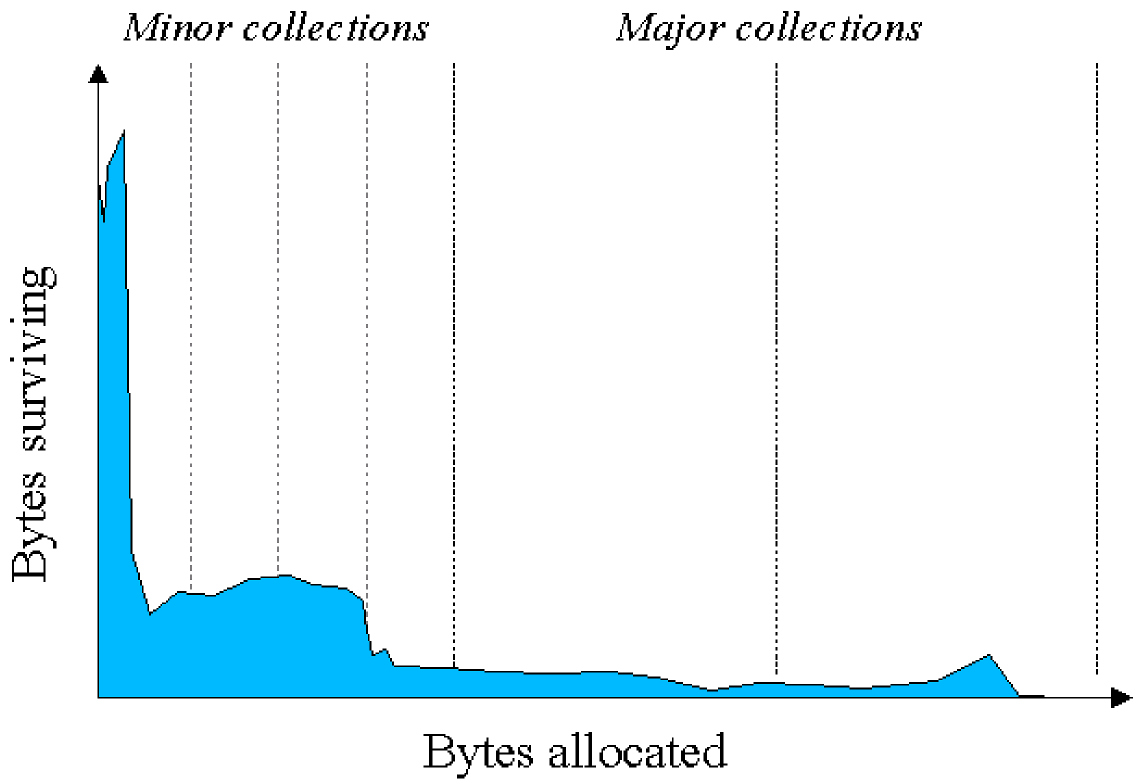
缺点：无法检测出循环引用

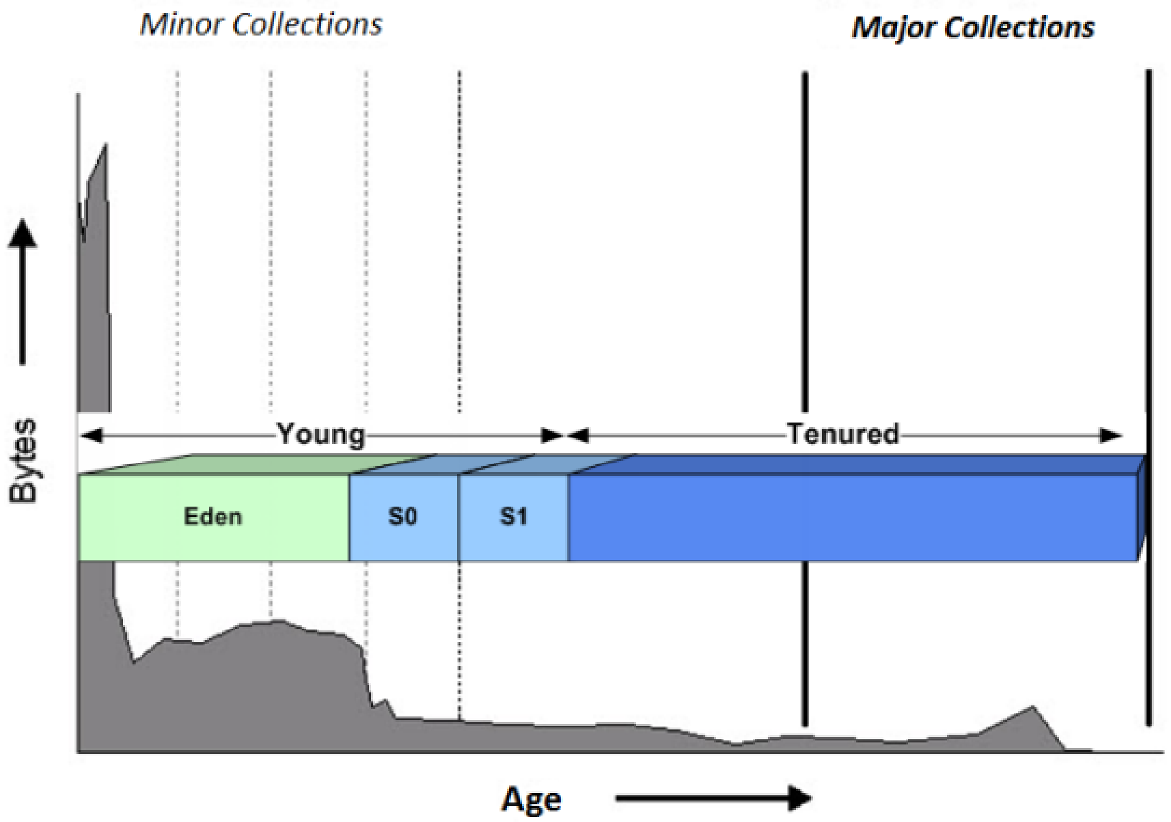
根搜索算法(Tracing Root)

　　根搜索算法是从离散数学中的图论引入的程序把所有的引用关系看作一张图，从一个节点GC ROOT开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点。

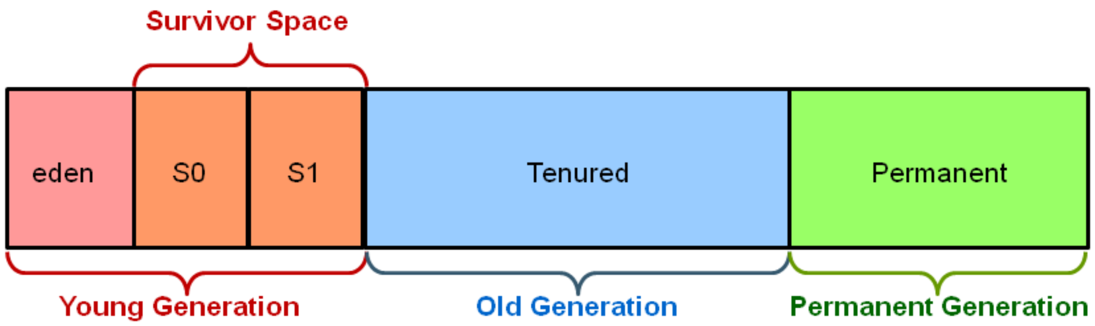
参见可以当做root节点的对象


GC Root
ObjA
ObjB
ObjC
ObjD
ObjF
...
ObjE


Minor collections
Major collections
Bytes surviving
Bytes allocated

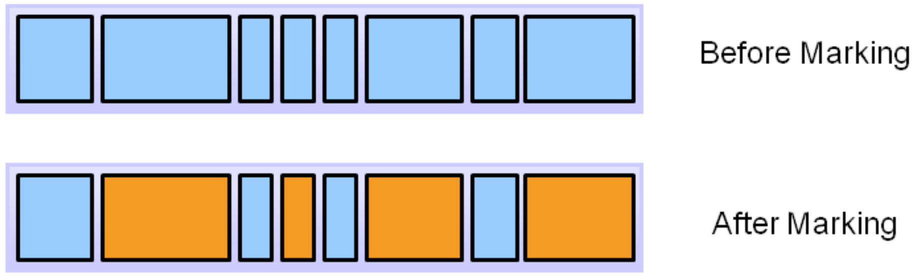

Minor Collections
Major Collections
Bytes
Young
Tenured
Eden
S0
S1
Age

Hotspot Heap Structure



标记-清除算法 (Mark-Sweep)

Marking

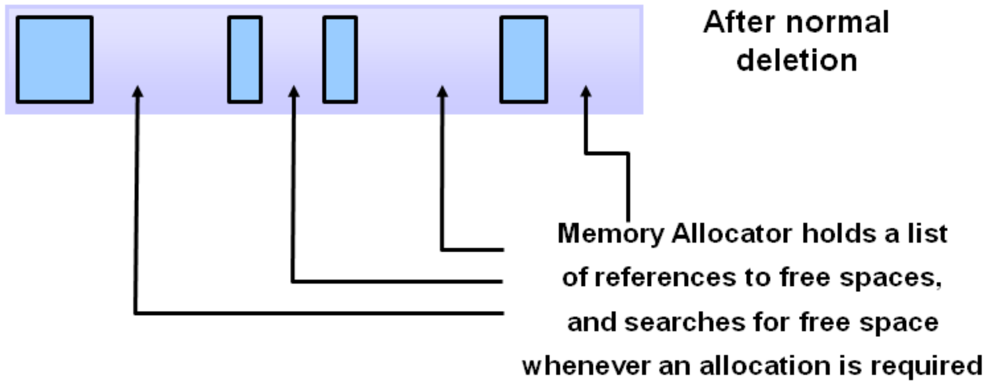


🟦 A live object

🟧 Unreferenced Objects

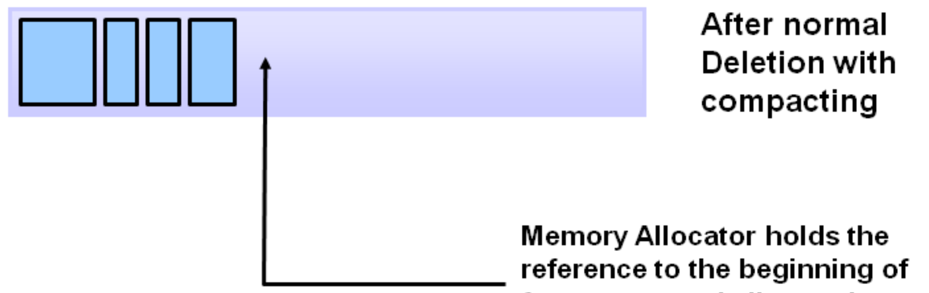
⬜ Memory space

Normal Deletion


After normal deletion
Memory Allocator holds a list of references to free spaces, and searches for free space whenever an allocation is required

标记-压缩算法 (Mark-Compact)

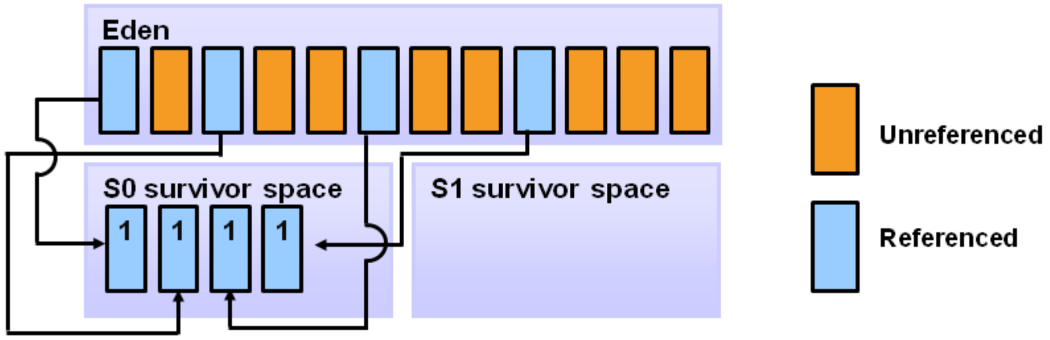
Deletion with Compacting



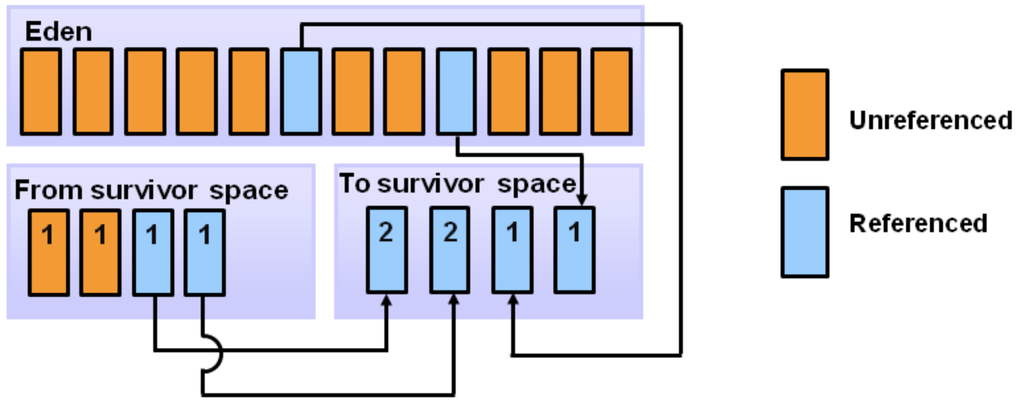
Memory Allocator holds the reference to the beginning of free space, and allocated memory sequentially then on.

复制算法 (Copying)

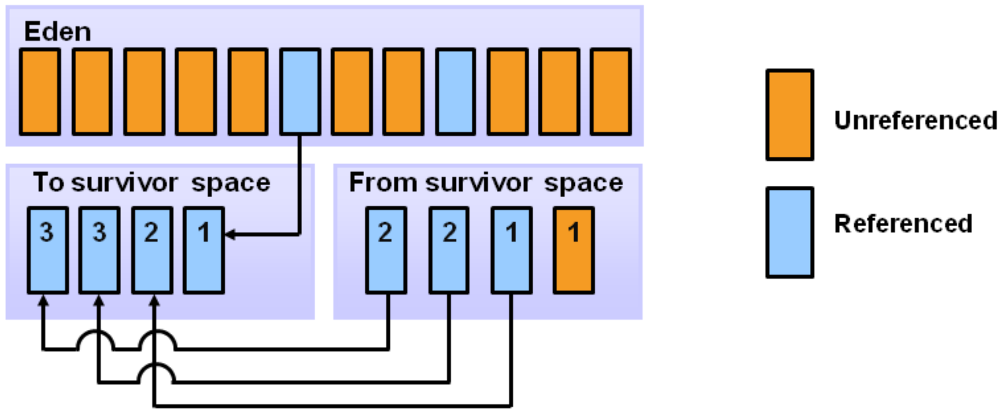
Copying Referenced Objects



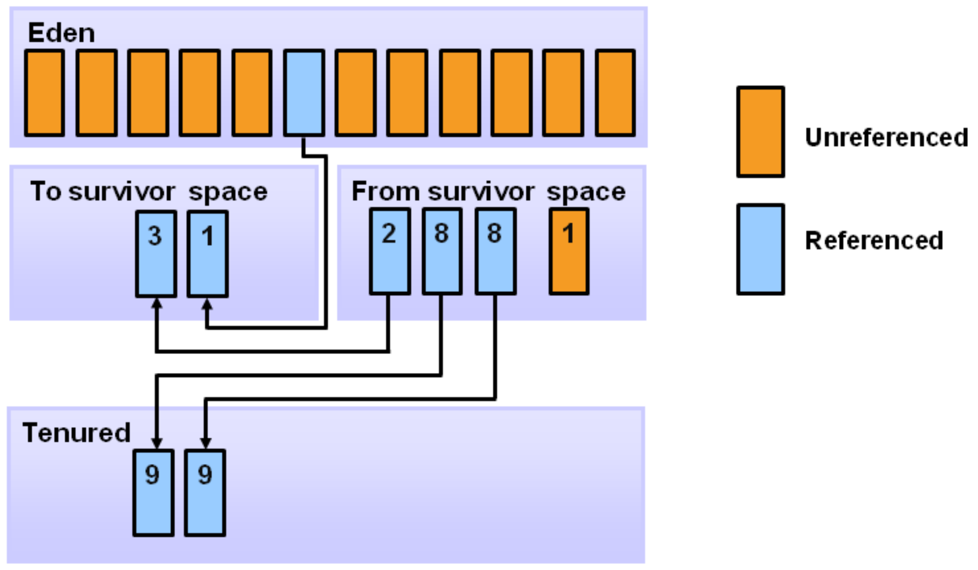
Object Aging



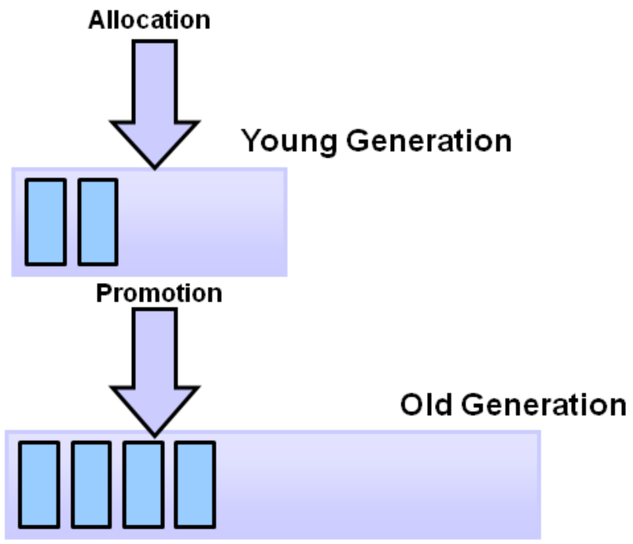
Additional Aging



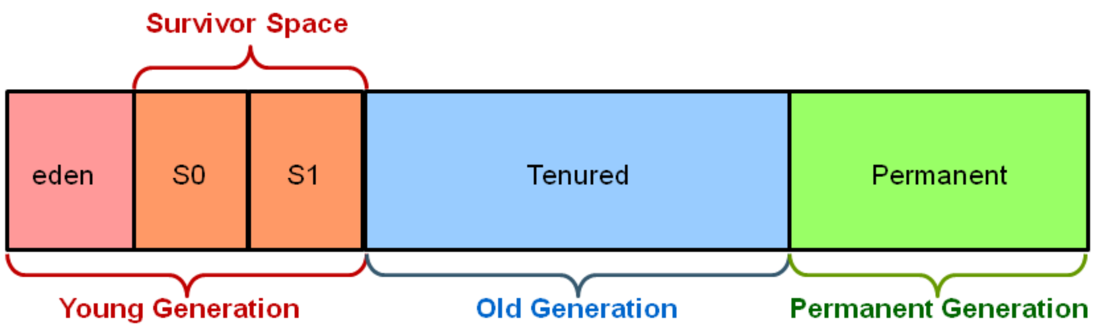
Promotion


Eden
To survivor space
3
1
From survivor space
2
8
8
1
Tenured
9
9
Unreferenced
Referenced

Promotion



Hotspot Heap Structure



Hotspot Tuning Heap Size

- Young Generation size determines
  - Frequency of minor GC
  - Number of objects reclaimed in minor GC
- Old Generation Size
  - Should hold application's steady-state live data size
  - Try to minimize frequency of major GC’s
- JVM footprint should not exceed physical memory

GC Logging in Production

- Don’t be afraid to enable GC logging in production
  - Very helpful when diagnosing production issues
- Extremely low / non-existent overhead
  - Maybe some large files in your file system.
  - We are surprised that customers are still afraid to enable it

-XX:+PrintGC

(the the alias -verbose:gc)

example:

```
[GC 246656K->243120K(376320K), 0,0929090 secs]
[Full GC 243120K->241951K(629760K), 1,5589690 secs]
```

-XX:+PrintGCTimeStamps

(a timestamp reflecting the real time passed in seconds since JVM start is added to every line)

example:

```
0,185: [GC 66048K->53077K(251392K), 0,0977580 secs]
0,323: [GC 119125K->114661K(317440K), 0,1448850 secs]
```

-XX:+PrintGCDateStamps

```
2014-01-03T12:08:38.102-0100: [GC 66048K->53077K(251392K), 0,0959470 secs]
2014-01-03T12:08:38.239-0100: [GC 119125K->114661K(317440K), 0,1421720 secs]
```

-Xloggc:<file>

Rotate GC Log, after Java 1.6_34 (or 1.7_2)

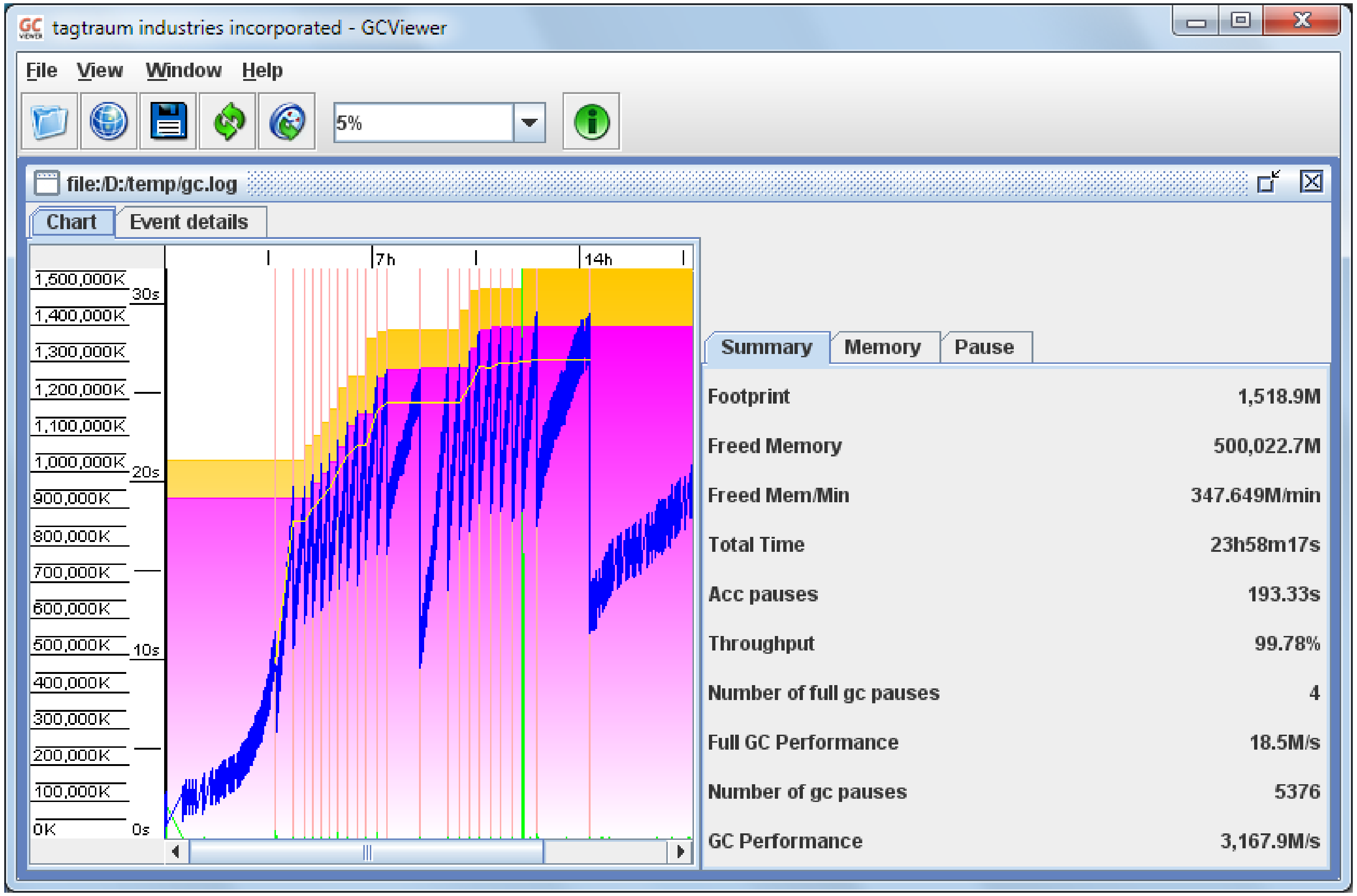
-XX:+UseGCLogFileRotaRon

-XX:NumberOfGCLogFiles=5

-XX:GCLogFileSize=10M

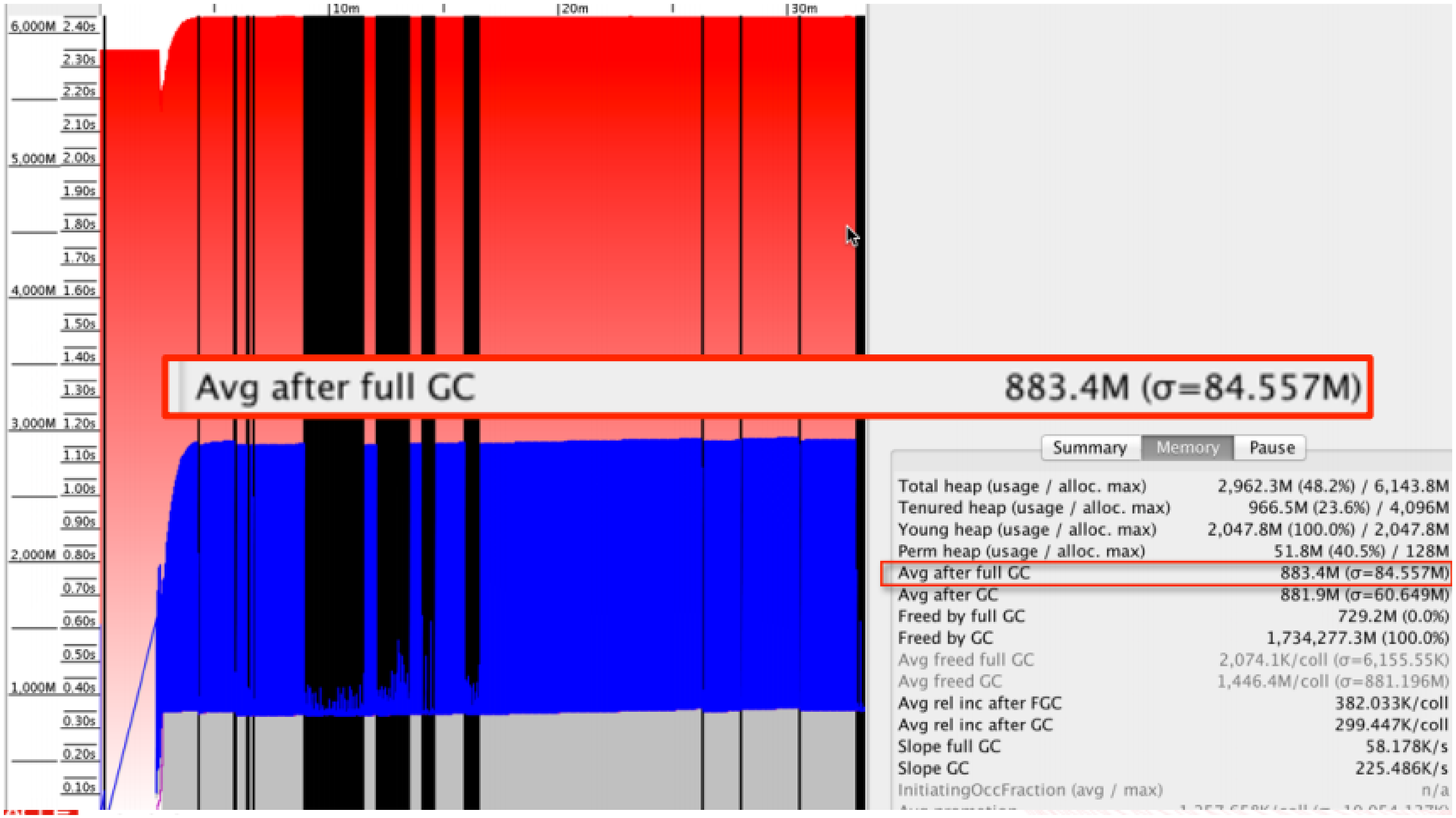
GCViewer – Offline analysis of GC logs

https://github.com/chewiebug/GCViewer



Footprint	1,518.9M
Freed Memory	500,022.7M
Freed Mem/Min	347.649M/min
Total Time	23h58m17s
Acc pauses	193.33s
Throughput	99.78%
Number of full gc pauses	4
Full GC Performance	18.5M/s
Number of gc pauses	5376
GC Performance	3,167.9M/s

Calculate Live Data Size by GCViewer

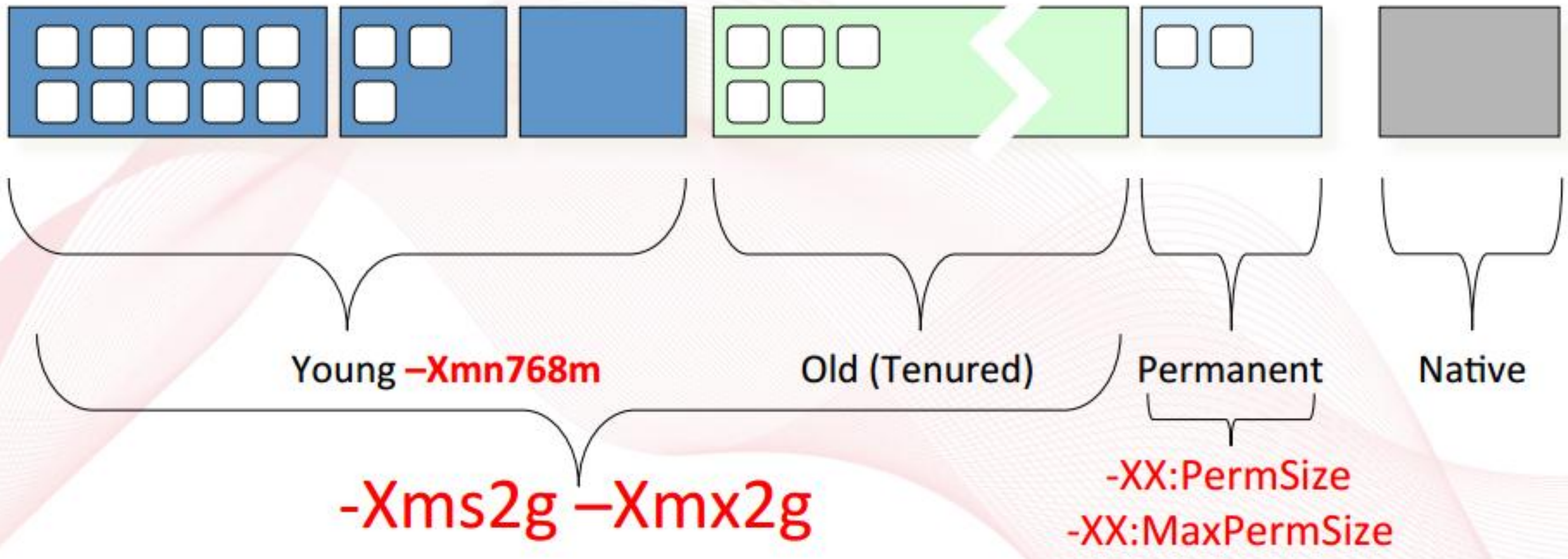


Avg after full GC 883.4M (σ=84.557M)

Total heap (usage / alloc. max)	2,962.3M (48.2%) / 6,143.8M
Tenured heap (usage / alloc. max)	966.5M (23.6%) / 4,096M
Young heap (usage / alloc. max)	2,047.8M (100.0%) / 2,047.8M
Perm heap (usage / alloc. max)	51.8M (40.5%) / 128M
Avg after full GC	883.4M (σ=84.557M)
Avg after GC	881.9M (σ=60.649M)
Freed by full GC	729.2M (0.0%)
Freed by GC	1,734,277.3M (100.0%)
Avg freed full GC	2,074.1K/coll (σ=6,155.55K)
Avg freed GC	1,446.4M/coll (σ=881.196M)
Avg rel inc after FGC	382.033K/coll
Avg rel inc after GC	299.447K/coll
Slope full GC	58.178K/s
Slope GC	225.486K/s
InitiatingOccFraction (avg / max)	n/a

- Rule of thumb
  - Set -Xms and -Xmx to 3x to 4x LDS
  - Set both -XX:PermSize and -XX:MaxPermSize to around 1.2x to 1.5x the max perm gen size
  - Old gen should be around 2x to 3x LDS
  - young gen should be around 1/3-1/4 of the heap size or 1x to 1.5x LDS
- 参考
  - Oracle Doc
  - Other

For LDS of 512m : -Xmn768m -Xms2g -Xmx2g



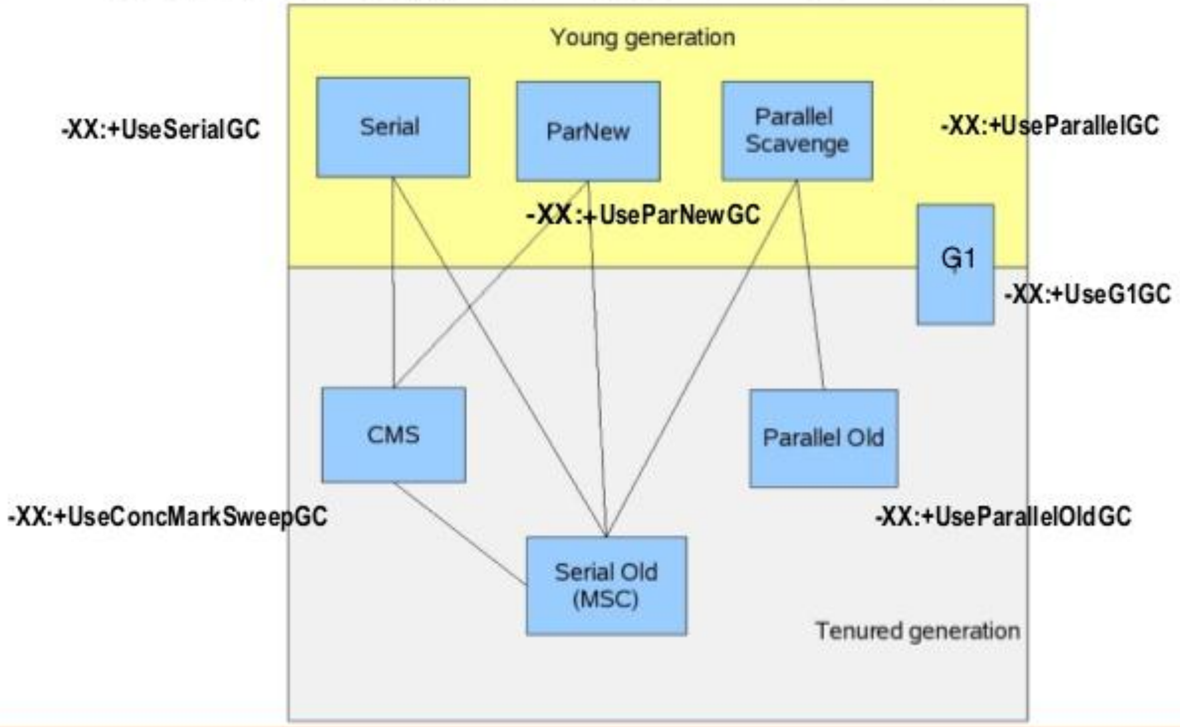
Java Garbage Collection

- Minor GC
- Major GC
- Full GC

Java Garbage Collectors

- 串行收集器(The Serial GC)
- 并行收集器(The Parallel GC)
- The Concurrent Mark Sweep (CMS) Collector
- The G1 Garbage Collector

HotSpot Garbage Collectors in Java SE 6



http://blogs.sun.com/jonthecollector/entry/our_collectors

The Serial GC

特点:
1) 它仅仅使用单线程进行垃圾回收；
1) 它独占式的垃圾回收

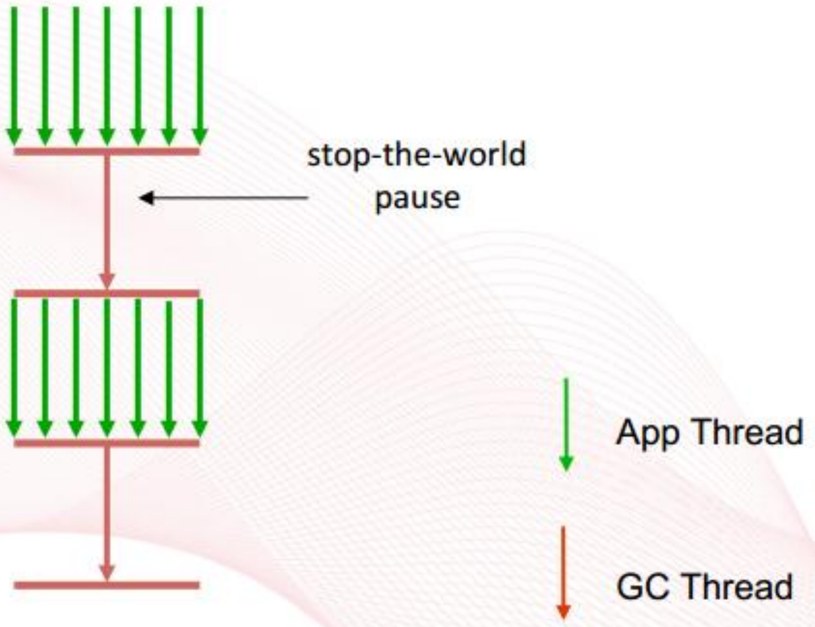
Use:

-XX：+UseSerialGC

eg:

```
java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -
XX:+UseSerialGC -jar server.jar
```

Serial Collector



The Parallel GC

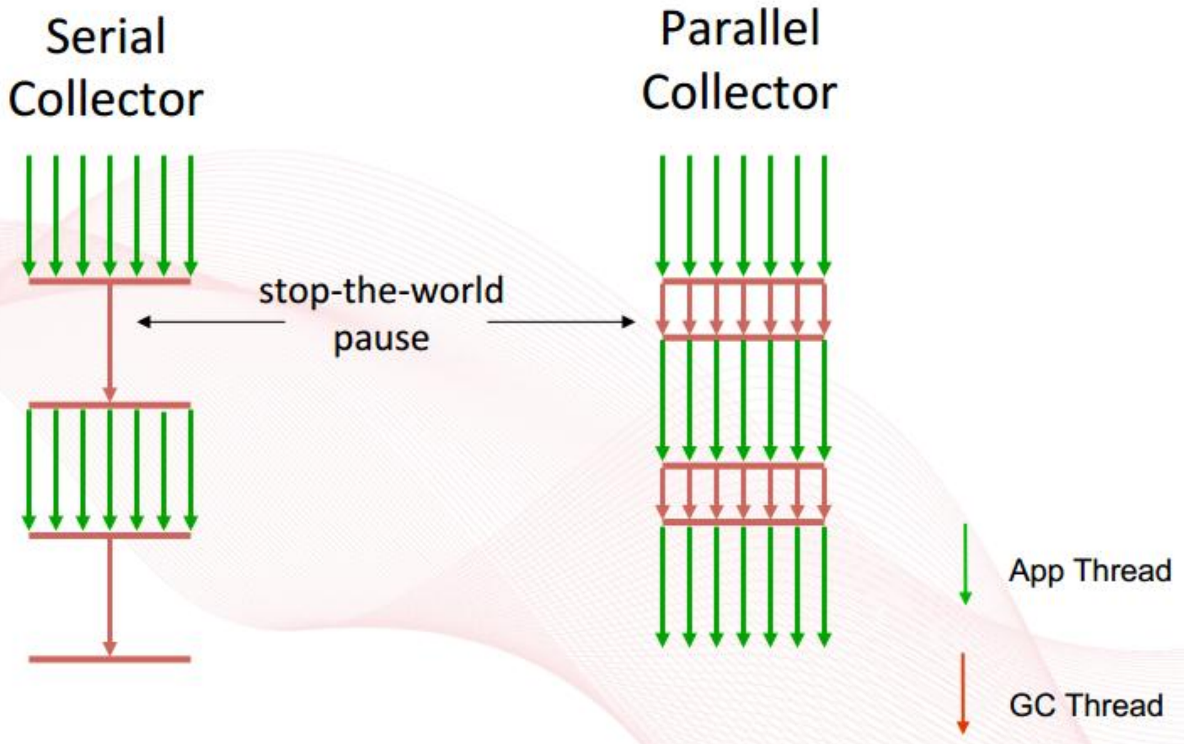
特点:

1) 它只是简单地将串行回收器多线程化.

2) 独占式的收集器.

3) throughput collector(侧重于吞吐)

Serial VS Parallel Collector



-XX:ParallelGCThreads=<desired number>
默认和CPU个数相同.

-XX:+UseParallelGC
多线程年轻代,单线程老年代.

-XX:+UseParallelOldGC
多线程年轻代,多线程老年代.

-XX:+MaxGCPauseMills

设置最大垃圾收集停顿时间

-XX:+GCTimeRatio

设置吞吐量大小,统将花费不超过 1/(1+n) 的时间用于垃圾收集

比如 GCTimeRatio 等于 19，则系统用于垃圾收集的时间不超过 1/(1+19)=5%。默认情况下，它的取值是 99，即不超过 1%的时间用于垃圾收集

-XX:+UseAdaptiveSizePolicy

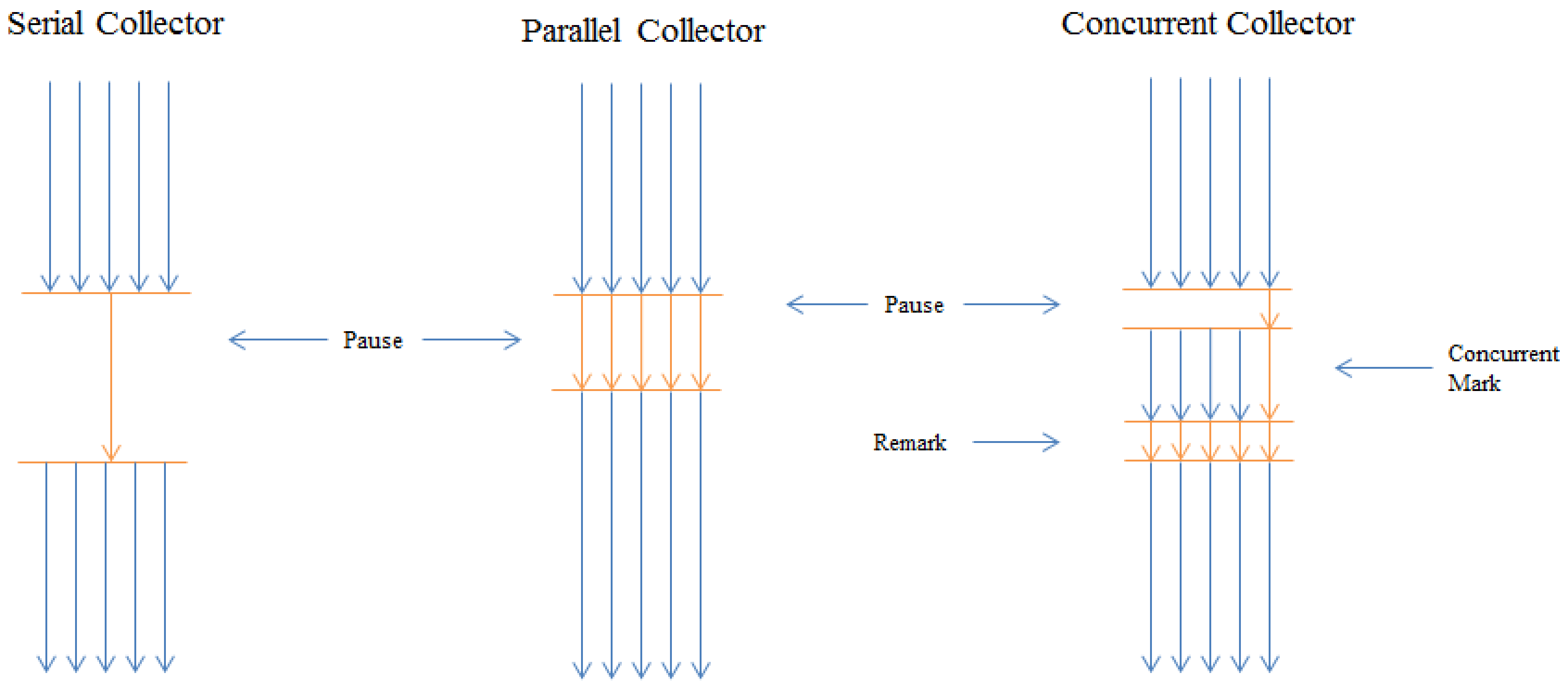
自适应 GC 策略,新生代的大小、eden 和 survivor 的比例、晋升老年代的对象年龄等参数会被自动调整

Concurrent Mark Sweep (CMS) Collector

1) 多线程,非独占
2) 关注于系统停顿时间

Use:

```
-XX:+UseConcMarkSweepGC
```


Serial Collector
Parallel Collector
Concurrent Collector
Pause
Pause
Concurrent Mark
Remark

Concurrent Mode Failure

Options

-XX:CMSInitiatingOccupancyFraction

(默认是 92)即当老年代的空间使用率达到 92%时，会执行一次 CMS 回收

-XX:+UseParNewGC

新生代2个算法两个实现方式; CMS开启式,这个参数默认启动,

‘-XX:-UseParNewGC’ 关闭.

Incremental Mode

1) 垃圾收集线程只收集一小片区域的内存空间，接着切换到应用程序线程。依次反复，直到垃圾收集完成

2) 适用于当cpu少,要求响应时间短.

-XX:+CMSIncrementalMode

要求必须开启 UseConcMarkSweepGC 才可以生效;默认是关闭;

GC Suggestions

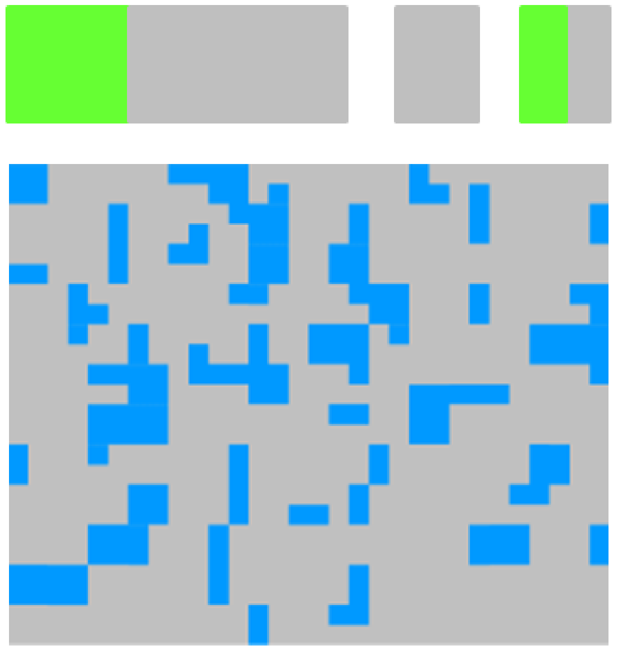
Generation	Low Latency Collectors	Throughput Collectors
	2+ CPUs	2+ CPUs
Young	`-XX:+UseParNewGC`	`-XX:+UseParallelGC`
Old	`-XX:+UseConcMarkSweepGC`	`-XX:+UseParallelOldGC`

G1 & CMS

- Garbage First
- Concurrent Mark Sweep

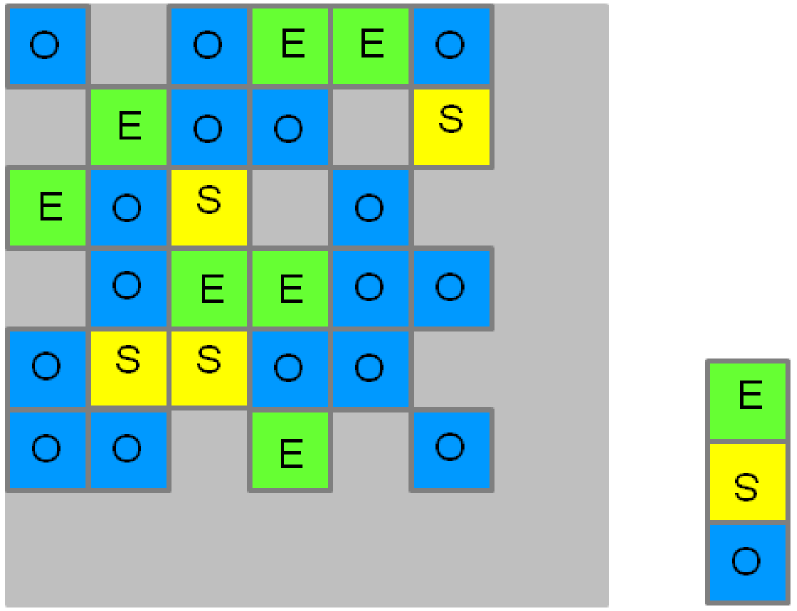
CMS

老年代收集 - 清理之后(After Sweeping)



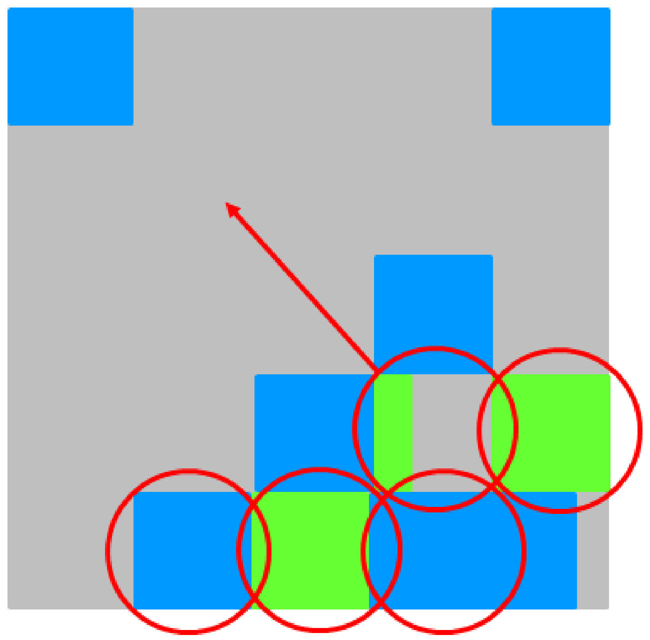
⬜ 未分配的空间
🟩 年轻代(Young Generation)
🟦 老年代(Old Generation)
🟩 新近拷贝到年轻代的部分
🟦 新近拷贝到老年代的部分

G1堆空间分配(Heap Allocation)



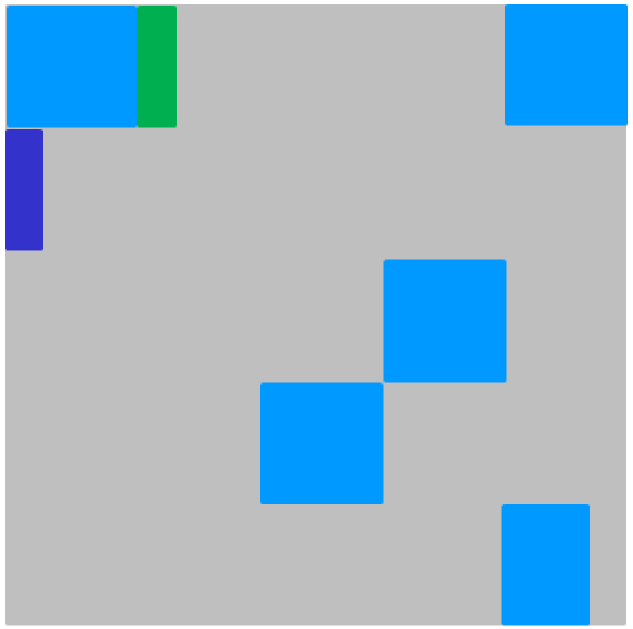
E	Eden Space	(新生代)
S	Survivor Space	(存活区)
O	Old Generation	(老年代)

拷贝/清理阶段(Copying/Cleanup)



⬜ 未分配的空间
🟩 年轻代(Young Generation)
🟦 老年代(Old Generation)
🟩 新近拷贝到年轻代的部分
🟦 新近拷贝到老年代的部分

拷贝/清理之后(After Copying/Cleanup)



⬜ 未分配的空间
🟩 年轻代(Young Generation)
🟦 老年代(Old Generation)
🟩 新近拷贝到年轻代的部分
🟪 新近拷贝到老年代的部分

G1的年轻代收集

- 堆一整块内存空间,被分为多个heap区(regions).
- 年轻代内存由一组不连续的heap区组成. 这使得在需要时很容易进行容量调整.
- 年轻代的垃圾收集,或者叫 young GCs, 会有 stop the world 事件. 在操作时所有的应用程序线程都会被暂停(stopped).
- 年轻代 GC 通过多线程并行进行.
- 存活的对象被拷贝到新的 survivor 区或者老年代.

G1对老年代的GC

- 并发标记清理阶段(Concurrent Marking Phase)
  - 活跃度信息在程序运行的时候被并行计算出来
  - 活跃度(liveness)信息标识出哪些区域在转移暂停期间最适合回收.
  - 不像CMS一样有清理阶段(sweeping phase).
- 再次标记阶段(Remark Phase)
  - 使用的 Snapshot-at-the-Beginning (SATB, 开始快照) 算法比起CMS所用的算法要快得多.
  - 完全空的区域直接被回收.
- 拷贝/清理阶段(Copying/Cleanup Phase)
  - 年轻代与老年代同时进行回收.
  - 老年代的选择基于其活跃度(liveness).

-XX:+UseG1GC

让 JVM 使用 G1 垃圾收集器

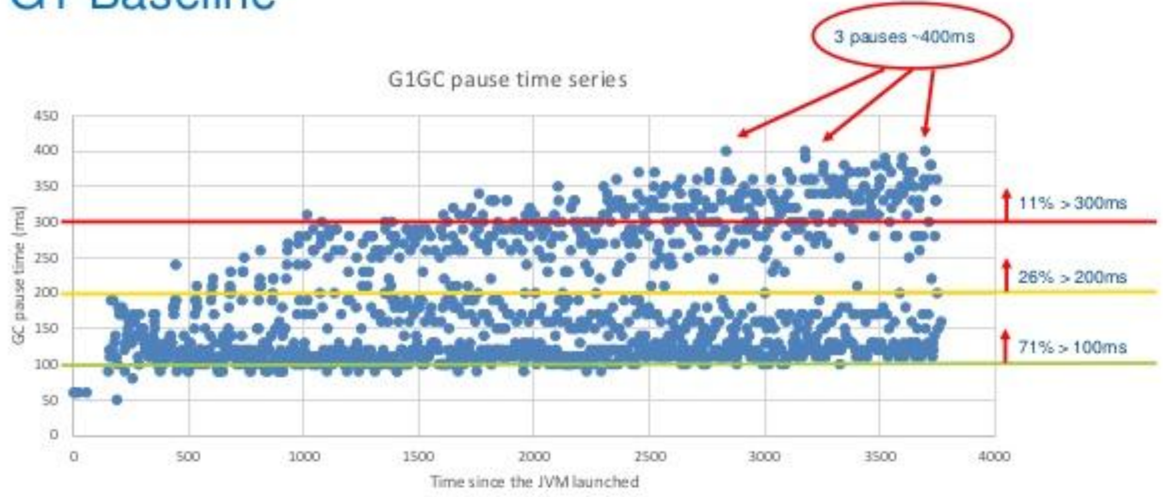
-XX:MaxGCPauseMillis=200

设置最大GC停顿时间(GC pause time)指标(target). 这是一个软性指标(soft goal), JVM 会尽力去达成这个目标. 所以有时候这个目标并不能达成. 默认值为 200 毫秒.

更多参见:

http://blog.csdn.net/renfufei/article/details/41897113

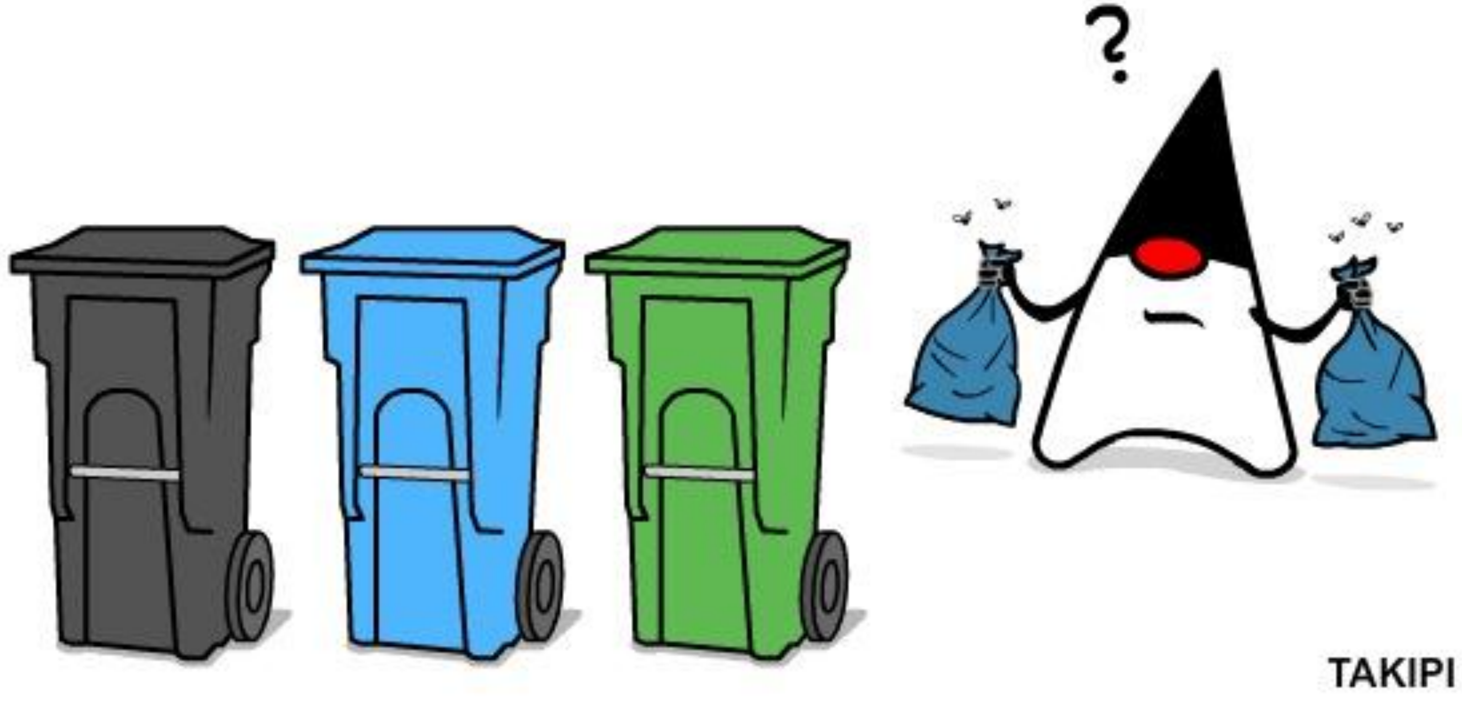
G1 Baseline



-XX:+UseG1GC -Xms100g –Xmx100g -XX:MaxGCPauseMillis=100

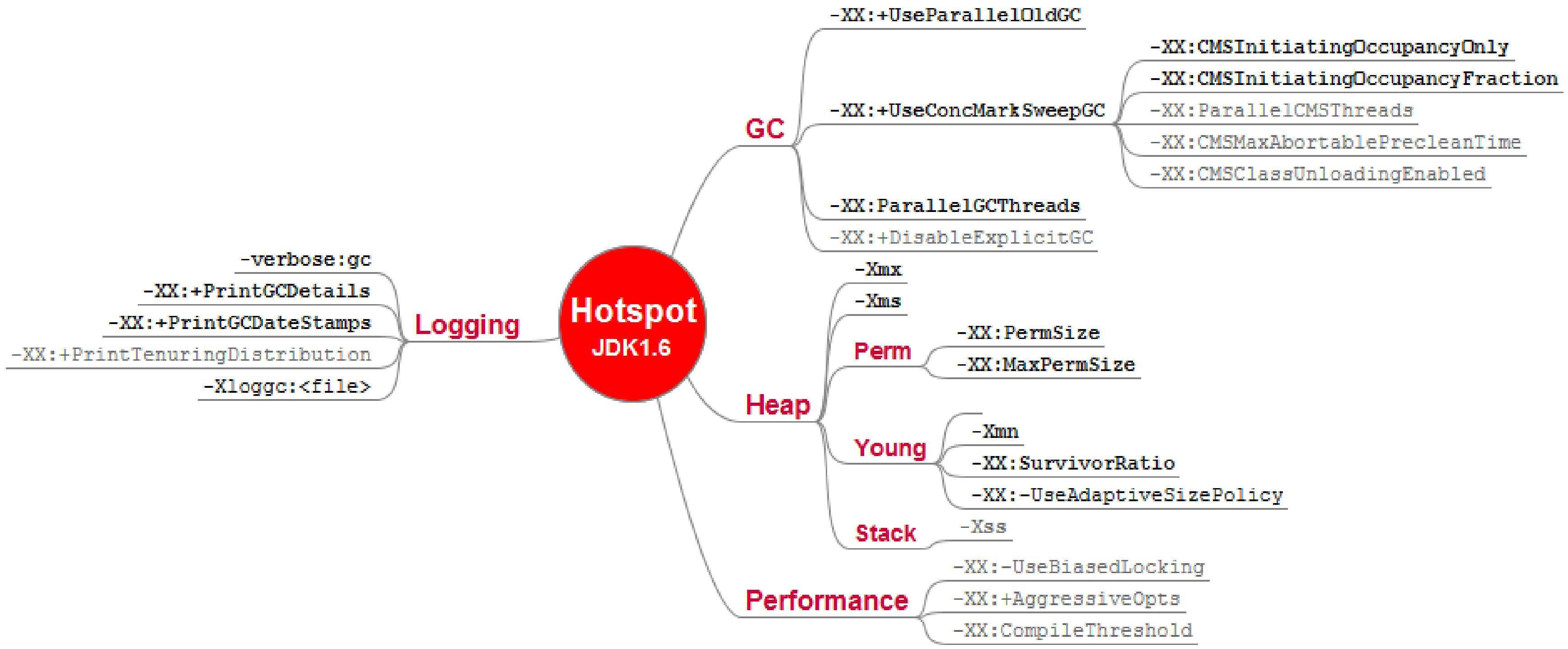
When to Use

- Supported for production use since 7u4
- Large heaps with GC latency requirements
  – Typically ~6GB or larger heaps
  – Stable and predictable GC latencies below 0.5 seconds
- Applications that have
  – more than 50% live data on the heap
  – varied object allocation rate
  – undesired long GC or compaction pauses (greater than 0.5s)


?
TAKIPI

Tuning Procedures

- Observer LDS to decide initial size setting
- Use parallel GC first
- If the GC pause time can’t be accepted, then trying to use CMS
- If heap is over 6G, and look for lower down pause time, you can consider G1
- Monitoring -> Analyzing -> Tuning -> …


Hotspot
JDK1.6
Logging
-verbose:gc
-XX:+PrintGCDetails
-XX:+PrintGCDateStamps
-XX:+PrintTenuringDistribution
-Xloggc:<file>
GC
-XX:+UseParallelOldGC
-XX:+UseConcMarkSweepGC
-XX:CMSInitiatingOccupancyOnly
-XX:CMSInitiatingOccupancyFraction
-XX:ParallelCMSThreads
-XX:CMSMaxAbortablePrecleanTime
-XX:CMSClassUnloadingEnabled
-XX:ParallelGCThreads
-XX:+DisableExplicitGC
Heap
-Xmx
-Xms
Perm
-XX:PermSize
-XX:MaxPermSize
Young
-Xmn
-XX:SurvivorRatio
-XX:-UseAdaptiveSizePolicy
Stack
-Xss
Performance
-XX:-UseBiasedLocking
-XX:+AggressiveOpts
-XX:CompileThreshold

参考

1. Oracle CMS介绍
2. JVM Options
3. G1 收集器
4. JVM 垃圾回收
5. JVM Options(2)
6. JVM Options 说明
7. Gc log 分析

THANKS

Bai du 百度