Garbage Collector
Magic Tuning
Explained

Simone Bordet
sbordet@intalio.com
About me

- Simone Bordet (sbordet@intalio.com)
- Senior Java Engineer @ Intalio/Webtide
  - Previously freelance, SimulaLabs, HP
- Active in Open Source and Java communities
  - Jetty, CometD, MX4J, Foxtrot, LiveTribe, etc.
  - Co-Leader of the Java User Group Torino
- Currently working on:
  - Comet client-side and server-side applications
    - Client for browsers, J2ME and Android
  - Server-side asynchronous I/O and protocols
Do you need to tune the GC?
GC Tuning Needed?

- Make your application right
- Make it even righter
- Make it fast
  - Use profilers and similar tools
- At the end, when all the rest is done, and your application has been live for a while, then you can look at the Garbage Collector
  - Rarely makes any sense doing it before
GC Tuning Needed?

- It is very difficult to replicate real load in a test environment.
- To tune the Garbage Collector, you need information taken from the live system.
- It will take a while to gather information.
  - Allocate time in the order of weeks to this activity.
- But sometimes, it really makes the difference.
Do I need to tune the GC?

A) No, but let's have some fun
B) Yes, my application needs it
Agenda

- JVM Memory Layout, Allocation and Collection
- Garbage Collector Algorithms
- Monitoring the Garbage Collector
- Tuning the Garbage Collector
JVM Memory Layout, Allocation and Collection
The JVM divides the memory it manages in 3 major “generations”:
- Young Generation (or “New”)
- Old Generation (or “Tenured”)
- Permanent Generation

Young + Old = Total Heap

-Xmx<size> sizes the total heap
- Default Young:Old ratio on 64-bit server JVM is 1:2
The Young Generation is again divided in 3 “spaces”:
- Eden Space
- Survivor Space 0
- Survivor Space 1

-Xmn<size> sizes the Young Generation
- There are other flags to fine tune it, but this works well
JVM Memory Layout

- Young Generation
- Old Generation
- Permanent Generation
Why does the JVM have “generations”?  
Careful analysis of Java applications showed that there are 2 types of garbage:

- “short-term” garbage, whose life is very short (few seconds or less)
- “long-term” garbage, whose life is longer (few minutes to application lifetime)

“Short-term” garbage is often responsible of most of the garbage generated

- An efficient GC for “short-term” garbage can free up most of the heap
JVM Memory Layout

Bytes surviving

Minor collections

Major collections

Bytes allocated
JVM Allocation Strategies

What happens when the JVM needs to allocate memory?

- It tries to allocate it in Young Generation, in the Eden space.

If that fails (not enough space left), then:
- It triggers a Young Generation collection; or
- It allocates it in the Old Generation directly (rare and possibly try to avoid it).
When the Eden Space is full, a so called “minor collection” is triggered

- Eden Space is emptied
- Survivor objects are copied into Survivor Space 0
- Survivor Space 1 is copied into Survivor Space 0
  - The Survivor Age is increased
  - Default Survivor Age on 64-bit server JVM is 4
- Older survivors overflow to the Old Generation

If not enough room in Survivor Space?
- Overflow to Old Generation
When the Old Generation is full, a so called “full collection” is triggered.

Exact behavior depends on the GC algorithm.

When the GC cannot free memory in the Old Generation, an OutOfMemoryError occurs.
Garbage Collector Algorithms
Garbage Collector Algorithms

- JDK 6 has 5 Garbage Collector Algorithms:
  - Parallel (PS – Parallel Scavenge)
    - Two available for the Young Generation
      - `-XX:+UseParallelGC`, cannot be used with CMS
      - `-XX:+UseParNewGC`, for use with CMS
    - One available for the Old Generation
      - `-XX:+UseParallelOldGC`
  - Concurrent (CMS – Concurrent Mark Sweep)
    - Only for the Old Generation
      - `-XX:+UseConcMarkSweepGC`
  - Serial
    - Only for the Old Generation
Garbage Collector Algorithms

- Parallel == full stop-the-world, multi-threaded
CMS == partial stop-the-world, multi threaded
Parallel Algorithms

- Grow and Shrink the Generation they work on
- Compact the space

Young Generation Algorithms

- Collection time depends on number of live objects
- Not on the Generation size, not on the amount of garbage
CMS Algorithm

- Does **not** compact (its worst “defect”)
- Hence it is subject to space fragmentation

What happens when the space is too fragmented and allocation fails?
- CMS falls back to the Serial algorithm
- Very long stop-the-world pause

Try to never reach that point with CMS
How many GC?

- There always are 2 collectors running in the JVM
  - One for the Young Generation
  - One for the Old Generation

- You can tune both independently, but they are related
  - This is what makes tuning difficult
Monitoring the Garbage Collector
Monitoring the collector

-XX:+PrintCommandLineFlags
  Reprints all implied options
-XX:+PrintGCDateStamps
-XX:+PrintGCTimeStamps
  GC time information
-XX:+PrintGCDetails
  GC activity information
-XX:+DisableExplicitGC
  Avoids RMI's System.gc()
-Xloggc:<file>
  Outputs to a file
Monitoring the collector

- Analyze the GC log file to understand:
  - GC overhead
    - time spent in GC / time spent in application
  - Max stop-the-world pause
  - Allocation rate and promotion rate

- Use jstat to gather further information
  - jstat -gcutil <pid>

- GC overhead vs Max stop-the-world pause
  - Overhead can be really low, but pauses really long
Tuning the Garbage Collector
Tuning the Garbage Collector

- You need to choose/tune 2 things:
  - Generation Sizes
  - GC algorithm

- Advice #1:
  - Make your heap BIG

- Big heaps reduce the frequency of collections
  - And increase the chance that objects do not survive

- Use `-Xms<size> == -Xmx<size>`
  - Saves grow/shrink time
Tuning the Garbage Collector

- Young Generation sizing: make it BIG
  - Can go up to same size as Old Generation
  - Remember: collection time does not depend on size

- Advice #2:
  - Maximize garbage in Young Generation

- Collection in Young Generation is cheap
- Usually not much tuning needed
  - The GC algorithm will be Parallel
Output example (collection frequency: ~35 s)

- `-XX:+UseParNewGC -XX:+PrintTenuringDistribution`

```
2010-06-02T06:43:08.589-0700: 940.165: [GC 940.165: [ParNew
Desired survivor size 104857600 bytes, new threshold 4 (max 4)
- age 1: 43824512 bytes, 43824512 total
- age 2: 17958408 bytes, 61782920 total
- age 3: 20590872 bytes, 82373792 total
- age 4: 14776712 bytes, 97150504 total
: 1793581K->132958K(1843200K), 0.1019750 secs] 2003784K->357779K(5939200K), 0.1021550 secs] [Times: user=0.60 sys=0.04, real=0.10 secs]
```

- Total Heap: 2003784 – 357779 = 1646005 collected (in Young)
- Young Generation: 1793581 – 132958 = 1660623
- Promoted: 1660623 – 1646005 = 14618
- Times: user/real = 6 (6x parallelism)
- Ages: ~44 MB age 1; ~18 MB age 2; ~21 MB age 3; ~15 MB age 4
Tuning the Garbage Collector

- Old Generation sizing: make it BIG
  - Bigger than or equal to Young Generation
  - Remember: collection time **does** depend on size

- Advice #3:
  - Try to avoid full collections

- Collection in Old Generation is expensive
Parallel Old Generation Collector

Has auto-tuning features ("ergonomics")
  • Not sure how good / reliable they are

Not much tuning needed anyway

Explicit tuning gives full control
Tuning the Garbage Collector

- Compact Mark Sweep (CMS) Old Generation Collector, or “low-pause” collector

Advice #4
- Try to avoid promotions

CMS does not compact space
- Need to avoid fragmentation

But you can schedule a compacting full GC
- For example, at night
Tuning the Garbage Collector

2010-06-02T10:25:06.432-0700: 14258.007: [GC [1 CMS-initial-mark: 3304088K(4096000K)] 3427806K(5939200K), 0.0678380 secs] [Times: user=0.06 sys=0.00, real=0.07 secs]

2010-06-02T10:25:06.500-0700: 14258.075: [CMS-concurrent-mark-start]
2010-06-02T10:25:07.401-0700: 14258.976: [CMS-concurrent-mark: 0.897/0.901 secs] [Times: user=2.42 sys=0.13, real=0.90 secs]

2010-06-02T10:25:07.492-0700: 14259.067: [CMS-concurrent-preclean: 0.076/0.091 secs] [Times: user=0.15 sys=0.01, real=0.09 secs]

CMS: abort preclean due to time 2010-06-02T10:25:12.589-0700: 14264.164: [CMS-concurrent-abortable-preclean: 4.970/5.097 secs] [Times: user=7.17 sys=0.41, real=5.10 secs]
Tuning the Garbage Collector

2010-06-02T10:25:12.592-0700: 14264.167: [GC[YG occupancy: 593314 K (1843200 K)]14264.168: [Rescan (parallel), 0.0766200 secs]14264.244: [weak refs processing, 0.1023280 secs]14264.347: [class unloading, 0.0059520 secs]14264.353: [scrub symbol & string tables, 0.0026240 secs] [1 CMS-remark: 3304088K(4096000K)] 3897403K(5939200K), 0.1925890 secs] [Times: user=0.70 sys=0.01, real=0.20 secs]

2010-06-02T10:25:15.655-0700: 14267.231: [CMS-concurrent-sweep: 2.860/2.860 secs] [Times: user=4.37 sys=0.28, real=2.86 secs]

2010-06-02T10:25:15.655-0700: 14267.231: [CMS-concurrent-reset-start]
2010-06-02T10:25:15.688-0700: 14267.264: [CMS-concurrent-reset: 0.033/0.033 secs] [Times: user=0.06 sys=0.01, real=0.04 secs]
Tuning the Garbage Collector

- **CMS-initial-mark** is the first stop-the-world phase
- Followed by a concurrent mark phase
- Then a concurrent preclean, that is meant to be interrupted
  - In this case by a 5 second timeout
- **CMS-remark** is the second stop-the-world phase
- Followed by a concurrent sweep phase
- Then a final reset phase
- The whole CMS cycle took 9.257 s (with a 5 s timeout)
Tuning the Garbage Collector

- It is possible to make CMS parallel
  - `-XX:ParallelCMSThreads=<number>`

- Trade off between CMS cycle time and overhead during concurrent phases
  - More threads will benefit the application during parallel phases, but hurt during concurrent phases

- CMS big risk: the collection cannot complete, so a compacting full collection is triggered
  - Which implies BIG pauses
Tuning the Garbage Collector

- CMS triggers by default when Old Generation is 92% full
  - Do not trust online sources that say 68%, try yourself
- Threshold at 92% could be too high
  - Leaves little space for big allocations
    - Remember, it's fragmented
  - A promotion from Young Generation may not find enough space
    - And a compacting full collection will trigger: big pause
- A CMS collection does not finish before the Old Generation is full
  - “Concurrent mode failure”
Tuning the Garbage Collector

- The most important tuning parameter for CMS:
  - `-XX:CMSInitiatingOccupancyFraction`

- Tells at what percentage trigger the CMS collection
  - You need trials and errors to tune it

- Trade off between collection frequency, collection overhead and risk of big pauses
Questions & Answers
References

- JDK 6 GC Reference:

- JDK 6 JVM Options:
  - http://java.sun.com/javase/technologies/hotspot/vmoptions.jsp

- Jon the Collector's blog:

- GC mailing lists archives:
  - http://mail.openjdk.java.net/mailman/listinfo/hotspot-gc-use
  - http://mail.openjdk.java.net/mailman/listinfo/hotspot-gc-dev