Tuning Garbage Collection Outline

This document is a summary or outline of Sun's document: Tuning Garbage collection with the 1.4.2 Hotspot JVM located here: http://java.sun.com/docs/hotspot/gc1.4.2/

1.0 Introduction

- For many applications garbage collection performance is not significant
- Default collector should be first choice

2.0 Generations

- Most straightforward GC will just iterate over every object in the heap and determine if any other objects reference it.
  - This gets really slow as the number of objects in the heap increase
- GC's therefor make assumptions about how your application runs.
- Most common assumption is that an object is most likely to die shortly after it was created: called infant mortality
- This assumes that an object that has been around for a while, will likely stay around for a while.
- GC organizes objects into generations (young, tenured, and perm) *This is important!*

2.1 Performance Considerations

- Ways to measure GC Performance
  - Throughput - % of time not spent in GC over a long period of time.
  - Pauses - app unresponsive because of GC
  - Footprint - overall memory a process takes to execute
  - Promptness - time between object death, and time when memory becomes available
- There is no one right way to size generations, make the call based on your applications usage.

2.2 Measurement

- Throughput and footprint are best measured using metrics particular to the application.
- Command line argument -verbose:gc output

```
[GC 325407K->83000K(776768K), 0.2300771 secs]
```

  - `GC` - Indicates that it was a minor collection (young generation). If it had said `Full GC` then that indicates that it was a major collection (tenured generation).
  - `325407K` - The combined size of live objects before garbage collection.
  - `83000K` - The combined size of live objects after garbage collection.

    - `(776768K)` - the total available space, not counting the space in the permanent generation, which is the total heap minus one of the survivor spaces.
    - `0.2300771 secs` - time it took for garbage collection to occur.
- You can get more detailed output using `-XX:+PrintGCDetails` and `-XX:+PrintGCTimeStamps`

3 Sizing the Generations

- The `-Xmx` value determines the size of the heap to reserve at JVM initialization.
- The `-Xms` value is the space in memory that is committed to the VM at init. The JVM can grow to the size of `-Xmx`.
- The difference between `-Xmx` and `-Xms` is virtual memory (virtually committed)

3.1 Total Heap

- Total available memory is the most important factor affecting GC performance
- By default the JVM grows or shrinks the heap at each GC to keep the ratio of free space to live objects at each collection within a specified range.
    - `-XX:MinHeapFreeRatio` - when the percentage of free space in a generation falls below this value the generation will be expanded to meet this percentage. Default is 40
    - `-XX:MaxHeapFreeRatio` - when the percentage of free space in a generation exceeded this value the generation will shrink to meet this value. Default is 70
- For server applications
    - Unless you have problems with pauses grant as much memory as possible to the JVM
    - Set `-Xms` and `-Xmx` close to each other or equal for a faster startup (removes constant resizing of JVM). But if you make a poor choice the JVM can't compensate for it.
    - Increase memory sa you increase # of processors because memory allocation can be parallelized.

3.2 The Young Generation

- The bigger the young generation the less minor GC's, but this implies a smaller tenured generation which increases the frequency of major collections.
- You need to look at your application and determine how long your objects live for to tune this.
- `-XX:NewRatio=3` - the young generation will occupy 1/4 the overall heap
- `-XX:NewSize` - Size of the young generation at JVM init. Calculated automatically if you specify -XX:NewRatio
- `-XX:MaxNewSize` - The largest size the young generation can grow to (unlimited if this value is not specified at command line)

3.2.1 Young Generation Guarantee

- The `-XX:SurvivorRatio` option can be used to tune the number of survivor spaces.
- Not often important for performance
    - `-XX:SurvivorRatio=6` - each survivor space will be 1/8 the young generation
    - If survivor spaces are too small copying collection overflows directly into the tenured generation.
    - Survivor spaces too large uselessly empty
    - `-XX:+PrintTenuringDistribution` - shows the threshold chosen by JVM to keep survivors half full, and the ages of objects in the new generation.
- Server Applications
    - First decide the total amount of memory you can afford to give the virtual machine. Then graph your own performance metric against young generation sizes to find the best setting.

    - Unless you find problems with excessive major collection or pause times, grant plenty of memory to the young generation.
    - Increasing the young generation becomes counterproductive at half the total heap or less (whenever the young generation guarantee cannot be met).
    - Be sure to increase the young generation as you increase the number of processors, since allocation can be parallelized.

4 Types of Collectors

- Everything to this point talks about the default garbage collector, there are other GC's you can use
- Throughput Collector - Uses a parallel version of the young generation collector
    - `-XX:+UseParallelGC`
    - Tenured collector is the same as in default
- Concurrent Low Pause Collector
    - Collects tenured collection concurrently with the execution of the app.
    - The app is paused for short periods during collection
    - `-XX:+UseConcMarkSweepGC`
    - To enable a parallel young generation GC with the concurrent GC add `-XX:+UseParNewGC` to the startup. Don't add `-XX:+UseParallelGC` with this option.
- Incremental Low Pause Collector
    - Sometimes called Train Collector
    - Collects a portion of the tenured generation at each minor collection.
    - Tries to minimize large pause of major collections
    - Slower than the default collector when considering overall throughput
    - Good for client apps (my observation)
    - `-Xincgc`
- Don't mix these options, JVM may not behave as expected.

4.1 When to use Throughput Collector

- Large number of processors
- Reduces serial execution time of app, by using multiple threads for GC
- App with lots of threads allocating objects should use this with a large young generation
- Server Applications (my observation)

4.2 The Throughput collector

- By default the throughput collector uses the number of CPU's as its value for number of GC threads.
- On a computer with one CPU it will not perform as well as the default collector
- Overhead from parallel execution (synchronization costs)
- With 2 CPU's the throughput collector performs as well as the default garbage collector.
- With more then 2 CPU's you can expect to see a reduction in minor GC pause times
- You can control the number of threads with `-XX:ParallelGCThreads=n`
- Fragmentation can occur
    - Reduce GC threads
    - Increase Tenured Generation size

4.2.1 Adaptive Sizing

- Keeps stats about GC times, allocation rates, and free space then sizes young and tenured generation to best fit the app.
- J2SE 1.4.1 and later

- `-XX:+UseAdaptiveSizePolicy` (on by default)

4.2.2 Aggressive Heap

- Attempts to make maximum use of physical memory for the heap
- Inspects computer resources (memory, num processors) and sets params optimal for long running memory allocation intensive jobs.
- Must have at least 256MB of RAM
- For lots of CPU's and RAM, but 1.4.1+ has shown improvements on 4-Way machines.
- `-XX:+AggressiveHeap`

4.3 When to use the Concurrent Low Pause Collector

- Apps that benefit from shorter GC pauses, and can share resources with GC during execution.
- Apps with large sets of long living data (tenured generation)
- Two or more processors
- Interactive apps with modest tenured generation size, and one CPU

4.4 The Concurrent Low Pause Collector

- Uses a separate GC thread to do parts of the major collection concurrently with the app threads.
- Pauses App threads in the beginning of a collection and toward the middle (longer pause in middle)
- The rest of the GC is in a single thread that runs at the same time as the app

4.4.1 Overhead of Concurrency

- Doesn't provide much of an advantage on single processor machines.
- Fragmentation can occur.
- Two processor machine eliminates pauses due to the GC thread.
- The more CPU's the advantages of concurrent collector increase.

4.4.2 Young Generation Guarantee

- There has to be enough contiguous space available in the tenured generation for all objects in the eden and one survivor space.
- A larger heap is needed compared to the default collector.
- Add the size of the young generation to the tenured generation.

4.4.3 Full Collections

- If the concurrent collector is unable to finish collecting the tenured generation before the tenured generation fills up, the application is paused and the collection is completed.
- When this happens you should make some adjustments to your GC params

4.4.4 Floating Garbage

- Floating Garbage - Objects that die while the GC is running (after they have been checked).
- Increase the tenured generation by 20% to reduce floating garbage.

4.4.5 Pauses

- First Pause - marks live objects - initial marking
- Second Pause - remarking phase - checks objects that were missed during the concurrent marking phase due to the concurrent execution of the app threads.

4.4.6 Concurrent Phases

- Concurrent Marking phase occurs between initial mark and remarking phase.
- Concurrent sweeping phase collects dead objects after the remarking phase.

4.4.7 Measurements with the Concurrent Collector

- Use `-verbose:gc` with `-XX:+PrintGCDetails`
- vCMS-initial-mark shows GC stats for the initial marking phase
- `CMS-concurrent-mark` - shows GC stats for concurrent marking phase.
- `CMS-concurrent-sweep` - shows stats for concurrent sweeping phase
- `CMS-concurrent-preclean` - stats for determining work that can be done concurrently
- `CMS-remark` - stats for the remarking phase.
- `CMS-concurrent-reset` - concurrent stuff is done, ready for next collection.

4.4.8 Parallel Minor Collection Options with Concurrent Collector

- `-XX:+UseParNewGC` - for multiprocessor machines, enables multi threaded young generation collection.
- `-XX:+CMSParallelRemarkEnabled` - reduce remark pauses

4.5 When to use the Incremental Low Pause Collector

- Use when you can afford to tradeoff longer and more frequent young generation GC pauses for shorter tenured generation pauses
- You have a large tenured generation
- Single Processor

4.6 The Incremental Low Pause Collector

- Minor collections same as default collector.
- Don't use try to use parallel GC with this collector
- Incrementally Collects parts of the tenured generation at each young collection.
- Tries to avoid long major collections by doing small chunks each minor collection.
- Can cause fragmentation of the heap. Sometimes need to increase tenured generation size compared to the default.
- There is some overhead required to maintain the position of the incremental collector. Less overhead than is required by the default collector.
- First try the default collector, and adjust heap sizing. If major pauses are too long try incremental.
- If the incremental collector can't collect the tenured generation fast enough you will run out of memory, try reducing the young generation.
- If young generation collections do not free any space, could be because of fragmentation. Increase tenured generation size.

4.6.1 Measurements with the Incremental Collector

- `-verbose:gc` and `-XX:+PrintGCDetails`
- Look for the Train: to see the stats for the incremental collection.

5 Other Considerations

- The permanent generation may be a factor on apps that dynamically generate and load many classes (JSP, CFM application servers)
- You may need to increase the `MaxPermSize`, eg: `-XX:MaxPermSize=128m`
- Apps that rely on finalization (finalize method, or finally clauses) will cause lag in garbage collection. This is a bad idea, use only for errorious situations.
- Explicit garbage collection calls (`System.gc()`) force a major collection. You can measure the effectiveness of these calls by disabling them with `-XX:+DisableExplicitGC`
- RMI garbage collection intervals can be controlled with
    - `-Dsun.rmi.dgc.client.gcInteraval=3600000`

    - `-Dsun.rmi.dgc.server.gcInterval=3600000`
- On Solaris 8+ you can enable libthreads, lightweight thread processes, these may increase thread performance.
- To enable add `/usr/lib/lwp` to `LD_LIBRARY_PATH`
- Soft References cleared less aggressively in server.
- `-XX:SoftRefLRUPolicyMSPerMB=10000`
- Default value is 1000, or one second per MB

6 Conclusion

- GC can be bottleneck in your app.

More Information

- FAQ: http://java.sun.com/docs/hotspot/gc1.4.2/faq.html
- Concurrent collector research paper: http://research.sun.com/techrep/2000/abstract-88.html
- HotSpot VM command line options: http://java.sun.com/docs/hotspot/VMOptions.html

Java Performance Books

Archives:

2019 2018 2017 2016

2015 2014 2013 2012

2011 2010 2009 2008

2007 2006 2005 2004

2003 2002

Pete Freitag

Pete is a husband and father located in scenic Central New York area. He owns a ColdFusion Consulting & Products company, Foundeo Inc. Pete is a frequent speaker at national conferences including Adobe ColdFusion Summit, Into The Box and others. He holds a BS in Software Engineering from Clarkson University. Read more about pete here.