February 15, 2017 | **Prvoslav Savic**

Serial GC

In this post I will go through all garbage collectors and common settings. Currently there are 4 garbage collectors that can be used:

- Serial GC
- Parallel GC (Enabled by default)
- The Concurrent Mark Sweep (CMS) Collector
- G1

Java allocated memory is split in three distinct memory sections or generations

- Java heap
    - Young generation
        - Eiden
        - Survivor 1
        - Survivor 2
    - Old generation(Tenured)
- Permanent generation(or Metaspace)
- Native heap

There are multiple tools that allow you to monitor memory and other related details of your running apps

- Jprofiler
- JVisualVM
- Jstat
- Jconsole

Serial GC as it name suggests operates in serial way for both young and old generation using single CPU. During the work of the GC world is stopped and application execution is on pause. Serial GC is a GC of choice for non server applications where only one CPU is available and heap size is couple of hundred MB. Serial GC uses next algorithms

- **mark and copy** for young generation
- **mark sweep compact** for old generation

You can explicitly request this GC using **-XX:+UseSerialGC**

Parallel GC

Parallel GC as it names suggests operates in parallel on all available cores. Number of threads can be configured , default value is the number of CPUs.

-XX:ParallelGCThreads=NNN

Parallel GC uses next algorithms for young and old generation

- **mark and copy** for young generation
- **mark sweep compact** for old generation

Parallel GC is suitable on machines that have multiple CPU cores using which throughput is increased and pauses are shorted, since multiple threads are collecting at once. Because collection has to be done without any interruptions application execution is stopped, so if latency is not your primary goal and time to time long pauses are acceptable for you this is the GC for you.

To enable parallel garbage collector use

-XX:+UseParallelGC -XX:+UseParallelOldGC

First option is to collect young generation in parallel second is to collect old generation in parallel. For old generation mark-sweep-compact algorithm is used.

Adaptive sizing can be applied, this will allow java to decide what is the optimal size for heap. There are two options to be used **MaxGCPauseMillis** and **MaxGCRatio**. First options specifies wanted max pause time and second represents the throughput. If both parameters are specified java will try to satisfy the pause times first and then it will extend the total heap size to try to meat the MaxGCRatio.

There is a formula that is used to calculate the value of MaxGCRation is

1 MaxGCRatio = Throughput/1-Throughput

So for Throughput of 0.95 MaxGcRatio is 19.

The Concurrent Mark Sweep (CMS)

CMS Collector is designed to avoid long pauses of the previously mentioned parallel collector. This is achieved firstly by **not** compacting the old generation, secondly by doing collection parallel to the application execution during the **mark sweep phase**. Because collector works parallel to application execution it uses CPU, usually 25% goes go the collector and rest is left for the application to use.

CMS does not compact old generation, because of this what can happen is that free memory exists but it can not be allocated because the longest block is shorter then what is required for the allocation. This occurs because of memory fragmentation. CMS will revert to full GC. Side effect of this issue is **promotion failure,** that occurs when collector wants to copy from young to old generation objects that are deserved to be in old generation. Copy requires that all object fit into one chunk of memory, and if chunk can not be found you have a promotion failure. Usual solution to this problem is to increase the total heap amount.

CMS does not start collecting old generation when old generation is full it actually starts a bit earlier, reason for this is that goal of CMS is to finish the collection before old generation is full. In case that old generation becomes full at some point CMS will revert to mark-sweep-compact algorithm used by parallel and serial collector. This is called **concurrent failure)** . CMS will start collecting the old generation when the initial occupancy is reached. Default value is 68% and it can be configured.

-XX:CMSInitiatingOccupancy=N -XX:+UseCMSInitialOccupancyOnly

Key is to avoid both mentioned failures by allowing collector to win by either increasing total heap or by ordering collector to start sooner.

Full GC also happens when PermGen is full. This can be avoided by telling the CMS to sweep the PermGen same as the old generation. Here is how to do that :

-XX:+CMSPermGenSweepingEnabled -XX:CMSInitiatingPermOccupancyFraction=N -XX:+CMSClassUnloadingEnabled

- Default CMSInitiatingPermOccupancyFraction is 80
- To unload unreferenced classes CMSClassUnloadingEnabled

When applications are run on machines with small number of processors sometimes it is required that CMS is run incrementally. CMS phases are split in chunks that are executed with time in between. **Incremental mode is deprecated in java 8**.

CMS uses this algorithms in parallel :

- **mark and copy** for young generation
- **mark sweep** for old generation

It can be selected by using (by default -XX:+UseParNewGC option is enabled)

-XX:+UseConcMarkSweepGC

Algorithm has low latency (short pauses) but it usually has worse throughput than the parallel collectors and requires bigger heap. This collector should be selected when main goal is the low latency, for example you want all your requests to be under 200ms, and you do not want to use parallel collector because 90% of requests will be for example 150ms and rest will be 500ms due to the long pause of the garbage collector(values are exaggerated)

Phases of the CMS

CMS has 6 phases in total , few of them are stop the world fashion and other are done parallel with the application.

- **Initial mark**, this phase is stop the world. Execution of the application is stopped while collector goes through and checks what objects are referenced from young generation objects and what are direct GC roots.
- **Concurrent mark**, this phase is done concurrently. Additionally to already marked objects almost all live objects are marked in this phase.
- **Precleaning phase**, this phase is done concurrently. While concurrent mark phase there is a chance that some references are changed by threads. This changes are identified in this phase. Start of the precleaning phase is tuned so that it after the next phase(remark phase that is stop the world) does not finish right before the start of young generation collection(that is stop the world too). This will then create two stop the world pauses next to each other.
- **The remark phase**, this is stop the world. Collector now needs to stop the execution of the application to finally determine what objects are live and what are not.
- **The sweep phase**, this phase is done concurrently. In this phase all space occupied by dead objects is reclaimed.
- **The reset phase**, this phase is done concurrently. Collector cleans its inner structures and prepares for next cycle.

G1

G1 is the collector that operates differently then previously mentioned collectors. First it does not have distinct parts of young and old generation but it has regions usually 2048 and each region can be used for young or old generation. G1 can be configured in such a way that pauses can not be

longer then X in Y period of time. Then G1 uses this configuration to determine what job can it do so that pauses fit the available time. Currently plan is to use G1 as default collector in java 9 http://openjdk.java.net/jeps/248

It can be selected by using

-XX:+UseG1GC

Let observe on configuration that uses G1

-server -Xms16G -Xmx16G -XX:PermSize=512m -XX:+UseG1GC -XX:MaxGCPauseMillis=200 -XX:ParallelGCThreads=10 -XX:ConcGCThreads=3 -XX:InitiatingHeapOccupancyPercent=70

- XX:MaxGCPauseMillis, specifies maximum pause time. G1 will try its best to achive it.
- XX:ParallelGCThreads, number of threads used in parallel phases
- XX:ConcGCThreads, number of threads concurrent collector will use
- -XX:InitiatingHeapOccupancyPercent, what is the percent of total occupancy of the heap before the GC cycle.

G1 does the young generation collection in stop the world fashion. For the old generation it has several phases , some are done concurrently and some are done in stop the world fashion.

- Initial mark (Stop the world)
- Concurrent root region scan
- Concurrent mark
- Remark (Stop the world)
- Clean up (Stop the world)
- Concurrent clean up
- Mixed

Important notice is that in mixed phase G1 collector collects the regions already marked as mostly garbage and young collection along. After all of the marked regions are collected G1 returns to normal young generation collections.

Same as with CMS there are few occasions when Full GC is performed

- Concurrent failure, happens if during marking phase old generation gets full.
- Promotion failure, happens if during the mixed phase old generation runs out of space.
- Evacuation failure, happens when collector can not promote objects to survivor space and old generation.
- Humongous allocation failure, when application tries to allocated very large object.

G1 can be auto tuned by setting the max pause time **MaxGCPauseMillis**. Here same trade of is made, if value is to small the collector will reduce the heap size and will work more often.

You can also allow more threads to be used for collection using **ParallelGCThreads** this should help collector win the race and avoid upper mentioned failures.

Additionally you can tell the collector to start sooner by setting the **InitiatingHeapOccupancyPercent** (same as with CMS) but here occupancy is related to whole heap and not per generation.

For more detail explanations of G1 visit oracle.com where G1 is explained in more detail.

Algorithms used by collectors

Mark and Copy algorithm

Memory is split into two regions, all live objects are moved from one region to other region and when work is done result is that objects are compacted. Problem with this approach is that it is required that there is enough space for this other region where the objects will be moved.

Mark and sweep algorithm

During the mark phase all free memory regions are remembered in something called free-list. Management of Free-list creates a overhead since it also uses memory and CPU. Problem with this algorithm even though it is simplest is that it can create memory fragmentation, result can be that you have OutOfMemoryError with lot of free memory but with no region of required size.

Mark sweep compact algorithm

This algorithm solves the shortcoming of mark and copy mentioned above, first it does not require space for second region. All live objects are moved and compacted to start of the memory thus allocating new memory locations is fast using pointer bumping. Memory fragmentation is low due to compacting of the used memory locations. Shortcoming of this algorithm is that it requires longer pause and more processor work to copy all the objects to the start of the memory.

Java configuration options list can be found here .

Java Ergonomics

There are default values that are used

- the default collector is **parallel collector**
- initial and maximum heaps
- server class machines(all machines except those configured by -client argument)

- Initial memory, 1/64 of the physical memory up to 1GB
- Maximum memory is 1/4 of physical memory up to 1GB

Behavioral parallel collector tuning

Parallel collector has two tricks up its sleeve , instead of configuring the size of the heap and size of the young generation you can leave this to the collector and specify maximum pause time or throughput value.

- -XX:MaxGCPauseTime=n
- -XX:GCTimeRatio=n , default is 99. So 1/1+99 means that 1% of time should go to GC collections.

Maximum pause time is specified per generation (young and old) , if specification is not met sizes are changed. Sometimes throughput can be lowered to match the desired pauses. This could mean that pauses are short as you desired but frequent.

GCTimeRatio is for both generations, when this goal is not met memory can be increased because bigger memory takes more time to fill up...

Priority, when both goals are specified collector tries to achieve **maximum pause time** first.

OutOfMemoryErros

OutOfMemoryErrors can occur due to different problems, problems can be solved by better configuration or by application and code examination using tools mentioned above.

When OutOfMemoryError occurs it is important to observe the text after it, there are few possible values

- **Java heap space**, memory for object could not be allocated in the heap. Observe how much memory did you give to the application (Xms and Xmx).
- **PermGen space**, Permanent generation is full. You can increase PermGen using -XX:PermGenSize=N.
- **Requested array size exceeds VM limit**, application attempted to allocate array of for example 521Mb but maximum heap memory is 256Mb, this exception will be thrown.

Evaluation Garbage Collection performance

To enable logs that will show you when GC collection is started and what are the results of it you can use two arguments, this will print the sizes of generations before and after the collection.

-XX:+PrintGCDetails

-XX:PrintGCTimeStamps

Additionally you can specify where this logs are to be saved using:

There are apps that can parse this logs one of them is Gchisto .

Typical configurational parameters

GC Selection

- -XX:UseSerialGC
- -XX:UseParallelGC
- -XX:UseParallelOldGC
- -XX:UseConcMarkSweepGC
- -XX:+UseG1GC

GC Statistics

- –XX:+PrintGC
- –XX:+PrintGCDetails
- –XX:+PrintGCTimeStamps

Heap and generation sizes

- –Xmsn, start heap size
- –Xmxn, maximum heap size
- –XX:MinHeapFreeRatio=minimum , when percent of free space in generation is below this value then heap is increased to match it. Default is 40%

- –XX:MaxHeapFreeRatio=maximum,maximum free heap ratio, when available heap is above then this value heap is decreased.
- –XX:NewSize=n, default initial size for young generation in bytes
- –XX:NewRatio=n, ratio between new and old generation. Default is 2.
- –XX:SurvivorRatio=n, ratio between each survivor space and Eden. Default is 8.
- –XX:MaxPermSize=n, maximum size of permanent generation
- -XX:InitialTenuringThreshold , how many times can an object move from one survivor space to other

New generation is 1/3 of old generation.
Eiden id 3/4 of young generation
Survivor spaces are 1/8 each(there are two).

Options for parallel GC

- –XX:ParallelGCThreads=n, number of GC threads used.
- –XX:MaxGCPauseMillis=n, max pause time
- –XX:GCTimeRatio=n, throughput is calculated 1/1+n

Options for parallel CMS

- -XX:CMSInitiatingOccupancyFraction=N, default is 96
- –XX:+CMSIncrementalMode, all phases are done incrementally with pausses to give CPU back to the processor

Options for G1

- -XX:ConcGCThreads=n, Number of parallel marking threads.
- -XX:InitiatingHeapOccupancyPercent=45 . Number of parallel marking threads.

Dump file on error

-XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=<path>.hprof

---