Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide

9 Garbage-First Garbage Collector

This section describes the Garbage-First (G1) Garbage Collector (GC).

Topics

Introduction to Garbage-First Garbage Collector

The Garbage-First (G1) garbage collector is targeted for multiprocessor machines with a large amount of memory. It attempts to meet garbage collection pause-time goals with high probability while achieving high throughput with little need for configuration. G1 aims to provide the best balance between latency and throughput using current target applications and environments whose features include:

- Heap sizes up to ten of GBs or larger, with more than 50% of the Java heap occupied with live data.
- Rates of object allocation and promotion that can vary significantly over time.
- A significant amount of fragmentation in the heap.
- Predictable pause-time target goals that aren't longer than a few hundred milliseconds, avoiding long garbage collection pauses.

G1 replaces the Concurrent Mark-Sweep (CMS) collector. It is also the default collector.

The G1 collector achieves high performance and tries to meet pause-time goals in several ways described in the following sections.

Enabling G1

The Garbage-First garbage collector is the default collector, so typically you don't have to perform any additional actions. You can explicitly enable it by providing `–XX:+UseG1GC` on the command line.

Basic Concepts

G1 is a generational, incremental, parallel, mostly concurrent, stop-the-world, and evacuating garbage collector which monitors pause-time goals in each of the stop-the-world pauses. Similar to other collectors, G1 splits the heap into (virtual) young and old generations. Space-reclamation efforts concentrate on the young generation where it is most efficient to do so, with occasional space-reclamation in the old generation

Some operations are always performed in stop-the-world pauses to improve throughput. Other operations that would take more time with the application stopped such as whole-heap operations like *global marking* are performed in parallel and concurrently with the application. To keep stop-the-world pauses short for space-reclamation, G1 performs space-reclamation incrementally in steps and in parallel. G1 achieves predictability by tracking information about previous application behavior and garbage collection pauses to build a model of the associated costs. It uses this information to size the work done in the pauses. For example, G1 reclaims space in the most efficient areas first (that is the areas that are mostly filled with garbage, therefore the name).

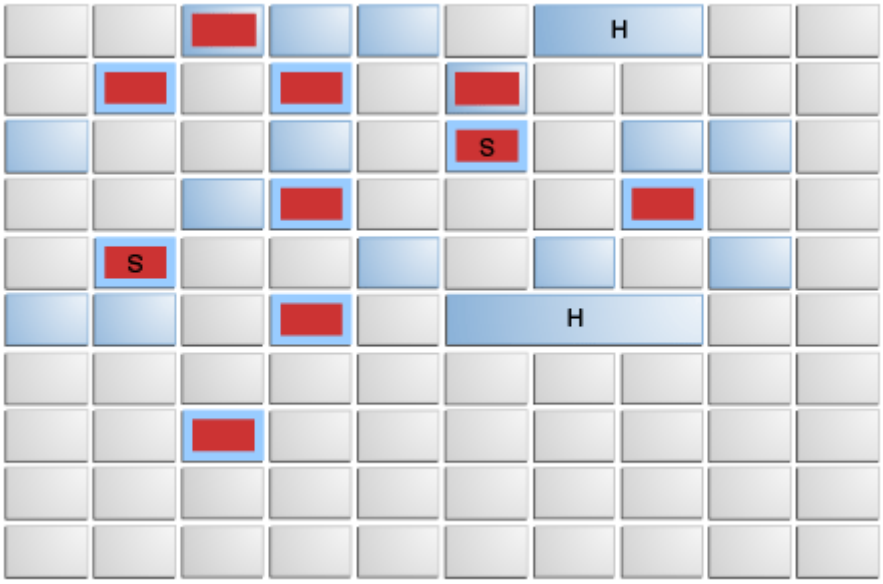
G1 reclaims space mostly by using evacuation: live objects found within selected memory areas to collect are copied into new memory areas, compacting them in the process. After an evacuation has been completed, the space previously occupied by live objects is reused for allocation by the application.

The Garbage-First collector is not a real-time collector. It tries to meet set pause-time targets with high probability over a longer time, but not always with absolute certainty for a given pause.

Heap Layout

G1 partitions the heap into a set of equally sized heap regions, each a contiguous range of virtual memory as shown in Figure 9-1. A region is the unit of memory allocation and memory reclamation. At any given time, each of these regions can be empty (light gray), or assigned to a particular generation, young or old. As requests for memory comes in, the memory manager hands out free regions. The memory manager assigns them to a generation and then returns them to the application as free space into which it can allocate itself.

Figure 9-1 G1 Garbage Collector Heap Layout



Description of "Figure 9-1 G1 Garbage Collector Heap Layout " (img_text/jsgct_dt_004_grbg_frst_hp.htm)

The young generation contains eden regions (red) and survivor regions (red with "S"). These regions provide the same function as the respective contiguous spaces in other collectors, with the difference that in G1 these regions are typically laid out in a noncontiguous pattern in memory. Old regions (light blue) make up the old generation. Old generation regions may be humongous (light blue with "H") for objects that span multiple regions.

An application always allocates into a young generation, that is, eden regions, with the exception of humongous, objects that are directly allocated as belonging to the old generation.

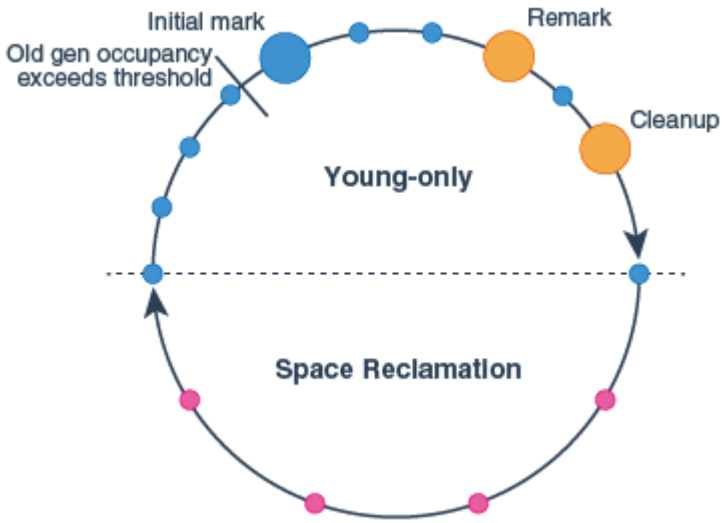
G1 garbage collection pauses can reclaim space in the young generation as a whole, and any additional set of old generation regions at any collection pause. During the pause G1 copies objects from this *collection set* to one or more different regions in the heap. The destination region for an object depends on the source region of that object: the entire young generation is copied into either survivor or old regions, and objects from old regions to other, different old regions using aging.

Garbage Collection Cycle

On a high level, the G1 collector alternates between two phases. The young-only phase contains garbage collections that fill up the currently available memory with objects in the old generation gradually. The space-reclamation phase is where G1 reclaims space in the old generation incrementally, in addition to handling the young generation. Then the cycle restarts with a young-only phase.

Figure 9-2 gives an overview about this cycle with an example of the sequence of garbage collection pauses that could occur:

Figure 9-2 Garbage Collection Cycle Overview



Description of "Figure 9-2 Garbage Collection Cycle Overview " (img_text/jsgct_dt_001_grbgcltncyl.htm)

The following list describes the phases, their pauses and the transition between the phases of the G1 garbage collection cycle in detail:

1. Young-only phase: This phase starts with a few young-only collections that promote objects into the old generation. The transition between the young-only phase and the space-reclamation phase starts when the old generation occupancy reaches a certain threshold, the Initiating Heap Occupancy threshold. At this time, G1 schedules an Initial Mark young-only collection instead of a regular young-only collection.

    - Initial Mark : This type of collection starts the marking process in addition to performing a regular young-only collection. Concurrent marking determines all currently reachable (live) objects in the old generation regions to be kept for the following space-reclamation phase. While marking hasn't completely finished, regular young collections may occur. Marking finishes with two special stop-the-world pauses: Remark and Cleanup.

    - Remark: This pause finalizes the marking itself, and performs global reference processing and class unloading. Between Remark and Cleanup G1 calculates a summary of the liveness information concurrently, which will be finalized and used in the Cleanup pause to update internal data structures.

    - Cleanup: This pause also reclaims completely empty regions, and determines whether a space-reclamation phase will actually follow. If a space-reclamation phase follows, the young-only phase completes with a single young-only collection.

2. Space-reclamation phase: This phase consists of multiple mixed collections that in addition to young generation regions, also evacuate live objects of sets of old generation regions. The space-reclamation phase ends when G1 determines that evacuating more old generation regions wouldn't yield enough free space worth the effort.

After space-reclamation, the collection cycle restarts with another young-only phase. As backup, if the application runs out of memory while gathering liveness information, G1 performs an in-place stop-the-world full heap compaction (Full GC) like other collectors.

Garbage-First Internals

This section describes some important details of the Garbage-First (G1) garbage collector.

Determining Initiating Heap Occupancy

The *Initiating Heap Occupancy Percent (IHOP)* is the threshold at which an Initial Mark collection is triggered and it is defined as a percentage of the old generation size.

G1 by default automatically determines an optimal IHOP by observing how long marking takes and how much memory is typically allocated in the old generation during marking cycles. This feature is called *Adaptive IHOP*. If this feature is active, then the option `-XX:InitiatingHeapOccupancyPercent` determines the initial value as a percentage of the size of the current old generation as long as there aren't enough observations to make a good prediction of the Initiating Heap Occupancy threshold. Turn off this behavior of G1 using the option`-XX:-G1UseAdaptiveIHOP`. In this case, the value of `-XX:InitiatingHeapOccupancyPercent` always determines this threshold.

Internally, Adaptive IHOP tries to set the Initiating Heap Occupancy so that the first mixed garbage collection of the space-reclamation phase starts when the old generation occupancy is at a current maximum old generation size minus the value of `-XX:G1HeapReservePercent` as the extra buffer.

Marking

G1 marking uses an algorithm called *Snapshot-At-The-Beginning (SATB)* . It takes a virtual snapshot of the heap at the time of the Initial Mark pause, when all objects that were live at the start of marking are considered live for the remainder of marking. This means that objects that become dead (unreachable) during marking are still considered live for the purpose of space-reclamation (with some exceptions). This may cause some additional memory wrongly retained compared to other collectors. However, SATB potentially provides better latency during the Remark pause. The too conservatively considered live objects during that marking will be reclaimed during the next marking. See the Garbage-First Garbage Collector Tuning (garbage-first-garbage-collector-tuning.htm#GUID-90E30ACA-8040-432E-B3A0-1E0440AB556A) topic for more information about problems with marking.

Behavior in Very Tight Heap Situations

When the application keeps alive so much memory so that an evacuation can't find enough space to copy to, an evacuation failure occurs. Evacuation failure means that G1 tries to complete the current garbage collection by keeping any objects that have already been moved in their new location, and not copying any not yet moved objects, only adjusting references between the object. Evacuation failure may incur some additional overhead, but generally should be as fast as other young collections. After this garbage collection with the evacuation failure, G1 will resume the application as normal without any other measures. G1 assumes that the evacuation failure occurred close to the end of the garbage collection; that is, most objects were already moved and there is enough space left to continue running the application until marking completes and space-reclamation starts.

If this assumption doesn't hold, then G1 will eventually schedule a Full GC. This type of collection performs in-place compaction of the entire heap. This might be very slow.

See Garbage-First Garbage Collector Tuning (garbage-first-garbage-collector-tuning.htm#GUID-90E30ACA-8040-432E-B3A0-1E0440AB556A) for more information about problems with allocation failure or Full GC's before signalling out of memory.

Humongous Objects

Humongous objects are objects larger or equal the size of half a region. The current region size is determined ergonomically as described in the Ergonomic Defaults for G1 GC (garbage-first-garbage-collector.htm#GUID-082C967F-2DAC-4B59-8A81-0CEC6EEB9016) section, unless set using the `-XX:G1HeapRegionSize` option.

These humongous objects are sometimes treated in special ways:

- Every humongous object gets allocated as a sequence of contiguous regions in the old generation. The start of the object itself is always located at the start of the first region in that sequence. Any leftover space in the last region of the sequence will be lost for allocation until the entire object is reclaimed.
- Generally, humongous objects can be reclaimed only at the end of marking during the Cleanup pause, or during Full GC if they became unreachable. There is, however, a special provision for humongous objects for arrays of primitive types for example, `bool`, all kinds of integers, and floating point values. G1 opportunistically tries to reclaim humongous objects if they are not referenced by many objects at any kind

of garbage collection pause. This behavior is enabled by default but you can disable it with the option –XX:G1EagerReclaimHumongousObjects.

- Allocations of humongous objects may cause garbage collection pauses to occur prematurely. G1 checks the Initiating Heap Occupancy threshold at every humongous object allocation and may force an initial mark young collection immediately, if current occupancy exceeds that threshold.
- The humongous objects never move, not even during a Full GC. This can cause premature slow Full GCs or unexpected out-of-memory conditions with lots of free space left due to fragmentation of the region space.

Young-Only Phase Generation Sizing

During the young-only phase, the set of regions to collect (collection set), consists only of young generation regions. G1 always sizes the young generation at the end of a young-only collection. This way, G1 can meet the pause time goals that were set using `–XX:MaxGCPauseTimeMillis` and `–XX:PauseTimeIntervalMillis` based on long-term observations of actual pause time. It takes into account how long it took young generations of similar size to evacuate. This includes information like how many objects had to be copied during collection, and how interconnected these objects had been.

If not otherwise constrained, then G1 adaptively sizes the young generation size between the values that `–XX:G1NewSizePercent` and `–XX:G1MaxNewSizePercent` determine to meet pause-time. See Garbage-First Garbage Collector Tuning (garbage-first-garbage-collector-tuning.htm#GUID-90E30ACA-8040-432E-B3A0-1E0440AB556A) for more information about how to fix long pauses.

Space-Reclamation Phase Generation Sizing

During the space-reclamation phase, G1 tries to maximize the amount of space that's reclaimed in the old generation in a single garbage collection pause. The size of the young generation is set to minimum allowed, typically as determined by `–XX:G1NewSizePercent`, and any old generation regions to reclaim space are added until G1 determines that adding further regions will exceed the pause time goal. In a particular garbage collection pause, G1 adds old generation regions in order of their reclamation efficiency, highest first, and the remaining available time to get the final collection set.

The number of old generation regions to take per garbage collection is bounded at the lower end by the number of potential candidate old generation regions (*collection set candidate regions*) to collect, divided by the length of the space-reclamation phase as determined by `–XX:G1MixedGCCountTarget`. The collection set candidate regions are all old generation regions that have an occupancy that's lower than `–XX:G1MixedGCLiveThresholdPercent` at the start of the phase.

The phase ends when the remaining amount of space that can be reclaimed in the collection set candidate regions is less than the percentage set by `–XX:G1HeapWastePercent`.

See Garbage-First Garbage Collector Tuning (garbage-first-garbage-collector-tuning.htm#GUID-90E30ACA-8040-432E-B3A0-1E0440AB556A) for more information about how many old generation regions G1 will use and how to avoid long mixed collection pauses.

Ergonomic Defaults for G1 GC

This topic provides an overview of the most important defaults specific to G1 and their default values. They give a rough overview of expected behavior and resource usage using G1 without any additional options.

Table 9-1 Ergonomic Defaults G1 GC

Option and Default Value	Description
`-XX:MaxGCPauseMillis=200`	The goal for the maximum pause time.
`-XX:GCPauseTimeInterval=`*<ergo>*	The goal for the maximum pause time interval. By default G1 doesn't set any goal, allowing G1 to perform garbage collections back-to-back in extreme cases.
`-XX:ParallelGCThreads=`*<ergo>*	The maximum number of threads used for parallel work during garbage collection pauses. This is derived from the number of available threads of the computer that the VM runs on in the following way: if the number of CPU threads available to the process is fewer than or equal to 8, use that. Otherwise add five eighths of the threads greater than to the final number of threads.
`-XX:ConcGCThreads=`*<ergo>*	The maximum number of threads used for concurrent work. By default, this value is `-XX:ParallelGCThreads` divided by 4.

`-XX:+G1UseAdaptiveIHOP`
 	
`-XX:InitiatingHeapOccupancyPercent=45`
	Defaults for controlling the initiating heap occupancy indicate that adaptive determination of that value is turned on, and that for the first few collection cycles G1 will use an occupancy of 45% of the old generation as mark start threshold.

`-XX:G1HeapRegionSize=<ergo>`	The set of the heap region size based on initial and maximum heap size. So that heap contains roughly 2048 heap regions. The size of a heap region can vary from 1 to 32 MB, and must be a power of 2.

`-XX:G1NewSizePercent=5`
 	
`-XX:G1MaxNewSizePercent=60`
	The size of the young generation in total, which varies between these two values as percentages of the current Java heap in use.

`-XX:G1HeapWastePercent=5`	The allowed unreclaimed space in the collection set candidates as a percentage. G1 stops the space-reclamation phase if the free space in the collection set candidates is lower than that.
`-XX:G1MixedGCCountTarget=8`	The expected length of the space-reclamation phase in a number of collections.
`-XX:G1MixedGCLiveThresholdPercent=85`	Old generation regions with higher live object occupancy than this percentage aren't collected in this space-reclamation phase.

Note:

`<ergo>` means that the actual value is determined ergonomically depending on the environment.

Comparison to Other Collectors

This is a summary of the main differences between G1 and the other collectors:

- Parallel GC can compact and reclaim space in the old generation only as a whole. G1 incrementally distributes this work across multiple much shorter collections. This substantially shortens pause time at the potential expense of throughput.
- Similar to the CMS, G1 concurrently performs part of the old generation space-reclamation concurrently. However, CMS can't defragment the old generation heap, eventually running into long Full GC's.
- G1 may exhibit higher overhead than other collectors, affecting throughput due to its concurrent nature.

Due to how it works, G1 has some unique mechanisms to improve garbage collection efficiency:

- G1 can reclaim some completely empty, large areas of the old generation during any collection. This could avoid many otherwise unnecessary garbage collections, freeing a significant amount of space without much effort.
- G1 can optionally try to deduplicate duplicate strings on the Java heap concurrently.

Reclaiming empty, large objects from the old generation is always enabled. You can disable this feature with the option `-XX:-G1EagerReclaimHumongousObjects`. String deduplication is disabled by default. You can enable it using the option `-XX:+G1EnableStringDeduplication`.

(https://docs.oracle.com/javase/9/gctuning/garbage-first-garbage-collector-tuning.htm)