Java Platform, Standard Edition

HotSpot Virtual Machine Garbage Collection Tuning Guide

Release 11
E95201-02
April 2022

ORACLE

Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide, Release 11

E95201-02

Contents

9 Garbage-First Garbage Collector

10 Garbage-First Garbage Collector Tuning

11 The Z Garbage Collector

12 Other Considerations

Preface

The *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide* describes the garbage collection methods included in the Java HotSpot Virtual Machine (Java HotSpot VM) and helps you determine which one is the best for your needs.

Audience

This document is intended for users, application developers and system administrators of the Java HotSpot VM that want to improve their understanding of the Java HotSpot VM garbage collectors. This document further provides help with analysis and solutions for common problems with garbage collection to make the application meet the users' requirements.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following documents:

- Garbage Collection: Algorithms for Automatic Dynamic Memory Management. Wiley, Chichester, July 1996. With a chapter on Distributed Garabge Collection by R. Lins. Richard Jones, Anony Hosking, and Elliot Moss.
- The Garbage Collection Handbook: The Art of Automatic Memory Managmenet. CRC Applied Algorithms and Data Structures. Chapman & Hall, January 2012

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.

Convention	Meaning
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction to Garbage Collection Tuning

A wide variety of applications, from small applets on desktops to web services on large servers, use the Java Platform, Standard Edition (Java SE). In support of this diverse range of deployments, the Java HotSpot VM provides multiple garbage collectors, each designed to satisfy different requirements. Java SE selects the most appropriate garbage collector based on the class of the computer on which the application is run. However, this selection may not be optimal for every application. Users, developers, and administrators with strict performance goals or other requirements may need to explicitly select the garbage collector and tune certain parameters to achieve the desired level of performance. This document provides information to help with these tasks.

First, general features of a garbage collector and basic tuning options are described in the context of the serial, stop-the-world collector. Then specific features of the other collectors are presented along with factors to consider when selecting a collector.

Topics

What Is a Garbage Collector?

The garbage collector (GC) automatically manages the application's dynamic memory allocation requests.

A garbage collector performs automatic dynamic memory management through the following operations:

- Allocates from and gives back memory to the operating system.
- Hands out that memory to the application as it requests it.
- Determines which parts of that memory is still in use by the application.
- Reclaims the unused memory for reuse by the application.

The Java HotSpot garbage collectors employ various techniques to improve the efficiency of these operations:

- Use generational scavenging in conjunction with aging to concentrate their efforts on areas in the heap that most likely contain a lot of reclaimable memory areas.
- Use multiple threads to aggressively make operations parallel, or perform some long-running operations in the background concurrent to the application.
- Try to recover larger contiguous free memory by compacting live objects.

Why Does the Choice of Garbage Collector Matter?

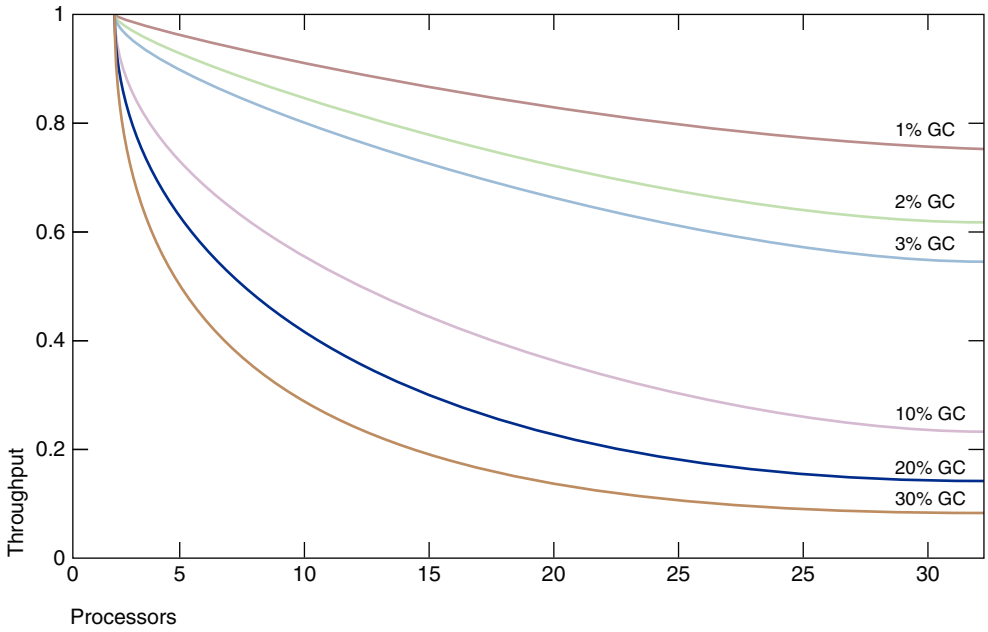
The purpose of a garbage collector is to free the application developer from manual dynamic memory management. The developer is freed of the requirement to match allocations with deallocations and closely take care of the lifetimes of allocated dynamic memory. This completely eliminates some classes of errors related to memory management at the cost of some additional runtime overhead. The Java HotSpot VM provides a selection of garbage collection algorithms to choose from.

When does the choice of a garbage collector matter? For some applications, the answer is never. That is, the application can perform well in the presence of garbage collection with pauses of modest frequency and duration. However, this isn't the case for a large class of applications, particularly those with large amounts of data (multiple gigabytes), many threads, and high transaction rates.

Amdahl's law (parallel speedup in a given problem is limited by the sequential portion of the problem) implies that most workloads can't be perfectly parallelized; some portion is always sequential and doesn't benefit from parallelism. In the Java platform, there are currently four supported garbage collection alternatives and all but one of them, the serial GC, parallelize the work to improve performance. It's very important to keep the overhead of doing garbage collection as low as possible. This can be seen in the following example.

The graph in Figure 1-1 models an ideal system that's perfectly scalable with the exception of garbage collection. The red line is an application spending only 1% of the time in garbage collection on a uniprocessor system. This translates to more than a 20% loss in throughput on systems with 32 processors. The magenta line shows that for an application at 10% of the time in garbage collection (not considered an outrageous amount of time in garbage collection in uniprocessor applications), more than 75% of throughput is lost when scaling up to 32 processors.

Figure 1-1 Comparing Percentage of Time Spent in Garbage Collection



This figure shows that negligible throughput issues when developing on small systems may become principal bottlenecks when scaling up to large systems. However, small improvements in reducing such a bottleneck can produce large gains in performance. For a sufficiently large system, it becomes worthwhile to select the right garbage collector and to tune it if necessary.

The serial collector is usually adequate for most small applications, in particular those requiring heaps of up to approximately 100 megabytes on modern processors. The other collectors have additional overhead or complexity, which is the price for specialized behavior. If the application does not need the specialized behavior of an alternate collector, use the serial collector. One situation where the serial collector isn't expected to be the best choice is a large, heavily threaded application that runs on a machine with a large amount of memory and two or more processors. When applications are run on such server-class machines, the Garbage-First (G1) collector is selected by default; see Ergonomics.

Supported Operating Systems in Documentation

This document and its recommendations apply to all JDK 11 supported system configurations, limited by actual availability of some garbage collectors in a particular configuration. See Oracle JDK Certified System Configurations.

2
Ergonomics

Ergonomics is the process by which the Java Virtual Machine (JVM) and garbage collection heuristics, such as behavior-based heuristics, improve application performance.

The JVM provides platform-dependent default selections for the garbage collector, heap size, and runtime compiler. These selections match the needs of different types of applications while requiring less command-line tuning. In addition, behavior-based tuning dynamically optimizes the sizes of the heap to meet a specified behavior of the application.

This section describes these default selections and behavior-based tuning. Use these defaults before using the more detailed controls described in subsequent sections.

Topics

Garbage Collector, Heap, and Runtime Compiler Default Selections

These are important garbage collector, heap size, and runtime compiler default selections:

- Garbage-First (G1) collector
- The maximum number of GC threads is limited by heap size and available CPU resources
- Initial heap size of 1/64 of physical memory
- Maximum heap size of 1/4 of physical memory
- Tiered compiler, using both C1 and C2

Behavior-Based Tuning

The Java HotSpot VM garbage collectors can be configured to preferentially meet one of two goals: maximum pause-time and application throughput. If the preferred goal is met, the collectors will try to maximize the other. Naturally, these goals can't always be met: Applications require a minimum heap to hold at least all of the live data, and other configuration might preclude reaching some or all of the desired goals.

Maximum Pause-Time Goal

The *pause time* is the duration during which the garbage collector stops the application and recovers space that's no longer in use. The intent of the *maximum pause-time* goal is to limit the longest of these pauses.

An average time for pauses and a variance on that average is maintained by the garbage collector. The average is taken from the start of the execution, but it's weighted so that more recent pauses count more heavily. If the average plus the variance of the pause-time is greater than the maximum pause-time goal, then the garbage collector considers that the goal isn't being met.

The maximum pause-time goal is specified with the command-line option `-XX:MaxGCPauseMillis=<nnn>`. This is interpreted as a hint to the garbage collector that a pause-time of *<nnn>* milliseconds or fewer is desired. The garbage collector adjusts the Java heap size and other parameters related to garbage collection in an attempt to keep garbage collection pauses shorter than *<nnn>* milliseconds. The default for the maximum pause-time goal varies by collector. These adjustments may cause garbage collection to occur more frequently, reducing the overall throughput of the application. In some cases, though, the desired pause-time goal can't be met.

Throughput Goal

The throughput goal is measured in terms of the time spent collecting garbage, and the time spent outside of garbage collection is the*application time*.

The goal is specified by the command-line option `-XX:GCTimeRatio=`*nnn*. The ratio of garbage collection time to application time is 1/ (1+*nnn*). For example, `-XX:GCTimeRatio=19` sets a goal of 1/20th or 5% of the total time for garbage collection.

The time spent in garbage collection is the total time for all garbage collection induced pauses. If the throughput goal isn't being met, then one possible action for the garbage collector is to increase the size of the heap so that the time spent in the application between collection pauses can be longer.

Footprint

If the throughput and maximum pause-time goals have been met, then the garbage collector reduces the size of the heap until one of the goals (invariably the throughput goal) can't be met. The minimum and maximum heap sizes that the garbage collector can use can be set using `-Xms=<nnn>` and `-Xmx=<mmm>` for minimum and maximum heap size respectively.

Tuning Strategy

The heap grows or shrinks to a size that supports the chosen throughput goal. Learn about heap tuning strategies such as choosing a maximum heap size, and choosing maximum pause-time goal.

Don't choose a maximum value for the heap unless you know that you need a heap greater than the default maximum heap size. Choose a throughput goal that's sufficient for your application.

A change in the application's behavior can cause the heap to grow or shrink. For example, if the application starts allocating at a higher rate, then the heap grows to maintain the same throughput.

If the heap grows to its maximum size and the throughput goal isn't being met, then the maximum heap size is too small for the throughput goal. Set the maximum heap size to a value that's close to the total physical memory on the platform, but doesn't cause swapping of the application. Execute the application again. If the throughput goal still isn't met, then the goal for the application time is too high for the available memory on the platform.

If the throughput goal can be met, but pauses are too long, then select a maximum pause-time goal. Choosing a maximum pause-time goal may mean that your throughput goal won't be met, so choose values that are an acceptable compromise for the application.

It's typical that the size of the heap oscillates as the garbage collector tries to satisfy competing goals. This is true even if the application has reached a steady state. The pressure to achieve a throughput goal (which may require a larger heap) competes with the goals for a maximum pause-time and a minimum footprint (which both may require a small heap).

3
Garbage Collector Implementation

One strength of the Java SE platform is that it shields the developer from the complexity of memory allocation and garbage collection.

However, when garbage collection is the principal bottleneck, it's useful to understand some aspects of the implementation. Garbage collectors make assumptions about the way applications use objects, and these are reflected in tunable parameters that can be adjusted for improved performance without sacrificing the power of the abstraction.

Topics

Generational Garbage Collection

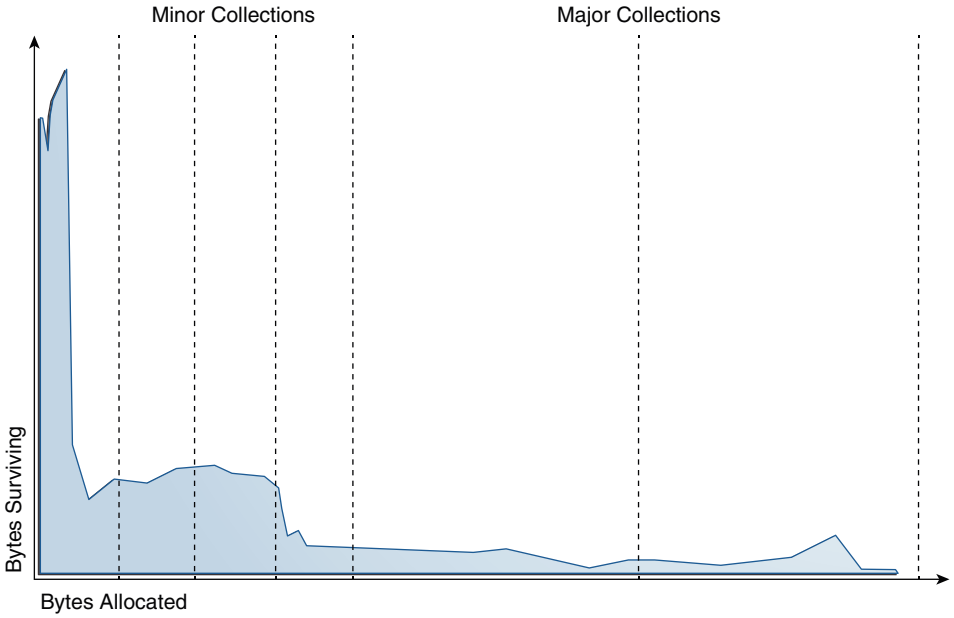
An object is considered garbage and its memory can be reused by the VM when it can no longer be reached from any reference of any other live object in the running program.

A theoretical, most straightforward garbage collection algorithm iterates over every reachable object every time it runs. Any leftover objects are considered garbage. The time this approach takes is proportional to the number of live objects, which is prohibitive for large applications maintaining lots of live data.

The Java HotSpot VM incorporates a number of different garbage collection algorithms that all use a technique called *generational collection*. While naive garbage collection examines every live object in the heap every time, generational collection exploits several empirically observed properties of most applications to minimize the work required to reclaim unused (garbage) objects. The most important of these observed properties is the *weak generational hypothesis*, which states that most objects survive for only a short period of time.

The blue area in Figure 3-1 is a typical distribution for the lifetimes of objects. The x-axis shows object lifetimes measured in bytes allocated. The byte count on the y-axis is the total bytes in objects with the corresponding lifetime. The sharp peak at the left represents objects that can be reclaimed (in other words, have "died") shortly after being allocated. For example, `iterator` objects are often only alive for the duration of a single loop.

Figure 3-1 Typical Distribution for Lifetimes of Objects



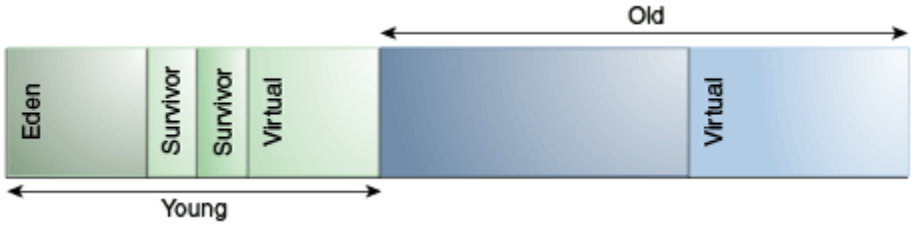
Some objects do live longer, and so the distribution stretches out to the right. For instance, there are typically some objects allocated at initialization that live until the VM exits. Between these two extremes are objects that live for the duration of some intermediate computation, seen here as the lump to the right of the initial peak. Some applications have very different looking distributions, but a surprisingly large number possess this general shape. Efficient collection is made possible by focusing on the fact that a majority of objects "die young."

Generations

To optimize for this scenario, memory is managed in *generations* (memory pools holding objects of different ages). Garbage collection occurs in each generation when the generation fills up.

The vast majority of objects are allocated in a pool dedicated to young objects (the *young generation*), and most objects die there. When the young generation fills up, it causes a *minor collection* in which only the young generation is collected; garbage in other generations isn't reclaimed. The costs of such collections are, to the first order, proportional to the number of live objects being collected; a young generation full of dead objects is collected very quickly. Typically, some fraction of the surviving objects from the young generation are moved to the *old generation* during each minor collection. Eventually, the old generation fills up and must be collected, resulting in a *major collection*, in which the entire heap is collected. Major collections usually last much longer than minor collections because a significantly larger number of objects are involved. Figure 3-2 shows the default arrangement of generations in the serial garbage collector:

Figure 3-2 Default Arrangement of Generations in the Serial Collector



At startup, the Java HotSpot VM reserves the entire Java heap in the address space, but doesn't allocate any physical memory for it unless needed. The entire address space covering the Java heap is logically divided into young and old generations. The complete address space reserved for object memory can be divided into the young and old generations.

The young generation consists of eden and two survivor spaces. Most objects are initially allocated in eden. One survivor space is empty at any time, and serves as the destination of live objects in eden and the other survivor space during garbage collection; after garbage collection, eden and the source survivor space are empty. In the next garbage collection, the purpose of the two survivor spaces are exchanged. The one space recently filled is a source of live objects that are copied into the other survivor space. Objects are copied between survivor spaces in this way until they've been copied a certain number of times or there isn't enough space left there. These objects are copied into the old region. This process is also called *aging*.

Performance Considerations

The primary measures of garbage collection are throughput and latency.

- *Throughput* is the percentage of total time not spent in garbage collection considered over long periods of time. Throughput includes time spent in allocation (but tuning for speed of allocation generally isn't needed).
- *Latency* is the responsiveness of an application. Garbage collection pauses affect the responsiveness of applications.

Users have different requirements of garbage collection. For example, some consider the right metric for a web server to be throughput because pauses during garbage collection may be tolerable or simply obscured by network latencies. However, in an interactive graphics program, even short pauses may negatively affect the user experience.

Some users are sensitive to other considerations. *Footprint* is the working set of a process, measured in pages and cache lines. On systems with limited physical memory or many processes, footprint may dictate scalability. *Promptness* is the time between when an object becomes dead and when the memory becomes available, an important consideration for distributed systems, including Remote Method Invocation (RMI).

In general, choosing the size for a particular generation is a trade-off between these considerations. For example, a very large young generation may maximize throughput, but does so at the expense of footprint, promptness, and pause times. Young generation pauses can be minimized by using a small young generation at the expense of throughput. The sizing of one generation doesn't affect the collection frequency and pause times for another generation.

There is no one right way to choose the size of a generation. The best choice is determined by the way the application uses memory as well as user requirements. Thus the virtual machine's choice of a garbage collector isn't always optimal and may be overridden with command-line options; see Factors Affecting Garbage Collection Performance.

Throughput and Footprint Measurement

Throughput and footprint are best measured using metrics particular to the application.

For example, the throughput of a web server may be tested using a client load generator, whereas the footprint of the server may be measured on the Solaris operating system using the `pmap` command. However, pauses due to garbage collection are easily estimated by inspecting the diagnostic output of the virtual machine itself.

The command-line option `-verbose:gc` prints information about the heap and garbage collection at each collection. Here is an example:

```
[15,651s][info ][gc] GC(36) Pause Young (G1 Evacuation Pause) 239M-
>57M(307M) (15,646s, 15,651s) 5,048ms
[16,162s][info ][gc] GC(37) Pause Young (G1 Evacuation Pause) 238M-
>57M(307M) (16,146s, 16,162s) 16,565ms
[16,367s][info ][gc] GC(38) Pause Full (System.gc()) 69M->31M(104M)
(16,202s, 16,367s) 164,581ms
```

The output shows two young collections followed by a full collection that was initiated by the application with a call to `System.gc()`. The lines start with a time stamp indicating the time from when the application was started. Next comes information about the log level (info) and tag (gc) for this line. This is followed by a GC identification number. In this case, there are three GCs with the numbers 36, 37, and 38. Then the type of GC and the cause for stating the GC is logged. After this, some information about the memory consumption is logged. That log uses the format "used before GC" -> "used after GC" ("heap size").

In the first line of the example this is 239M->57M(307M), which means that 239 MB were used before the GC and the GC cleared up most of that memory, but 57 MB survived. The heap size is 307 MB. Note in this example that the full GC shrinks the heap from 307 MB to 104 MB. After the memory usage information, the start and end times for the GC are logged as well as the duration (end - start).

The `-verbose:gc` command is an alias for `-Xlog:gc`. `-Xlog` is the general logging configuration option for logging in the HotSpot JVM. It's a tag-based system where `gc` is one of the tags. To get more information about what a GC is doing, you can configure logging to print any message that has the `gc` tag and any other tag. The command line option for this is `-Xlog:gc*`.

Here's an example of one G1 young collection logged with `-Xlog:gc*` :

```
[10.178s][info][gc,start ] GC(36) Pause Young (G1 Evacuation Pause)
[10.178s][info][gc,task ] GC(36) Using 28 workers of 28 for evacuation
[10.191s][info][gc,phases ] GC(36) Pre Evacuate Collection Set: 0.0ms
[10.191s][info][gc,phases ] GC(36) Evacuate Collection Set: 6.9ms
[10.191s][info][gc,phases ] GC(36) Post Evacuate Collection Set: 5.9ms
[10.191s][info][gc,phases ] GC(36) Other: 0.2ms
```

```
[10.191s][info][gc,heap ] GC(36) Eden regions: 286->0(276)
[10.191s][info][gc,heap ] GC(36) Survivor regions: 15->26(38)
[10.191s][info][gc,heap ] GC(36) Old regions: 88->88
[10.191s][info][gc,heap ] GC(36) Humongous regions: 3->1
[10.191s][info][gc,metaspace ] GC(36) Metaspace: 8152K->8152K(1056768K)
[10.191s][info][gc ] GC(36) Pause Young (G1 Evacuation Pause) 391M-
>114M(508M) 13.075ms
[10.191s][info][gc,cpu ] GC(36) User=0.20s Sys=0.00s Real=0.01s
```

Note:

The format of the output produced by `-Xlog:gc*` is subject to change in future releases.

4

Factors Affecting Garbage Collection Performance

The two most important factors affecting garbage collection performance are total available memory and proportion of the heap dedicated to the young generation.

Topics

Total Heap

The most important factor affecting garbage collection performance is total available memory. Because collections occur when generations fill up, throughput is inversely proportional to the amount of memory available.

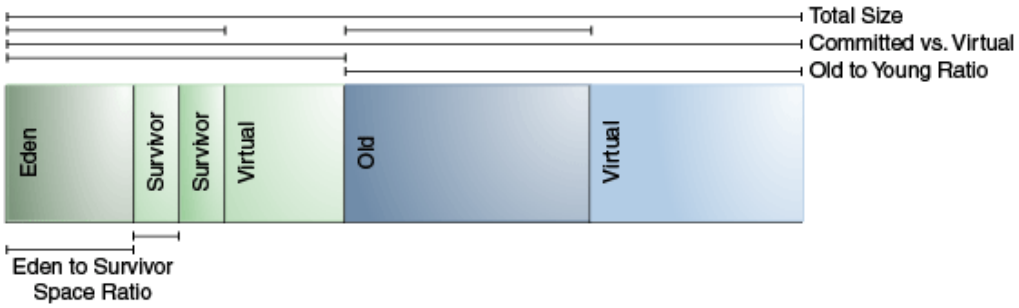
> **Note:**
>
> The following discussion regarding growing and shrinking of the heap, the heap layout, and default values uses the serial collector as an example. While the other collectors use similar mechanisms, the details presented here may not apply to other collectors. Refer to the respective topics for similar information for the other collectors.

Heap Options Affecting Generation Size

A number of options affects generation size. Figure 4-1 illustrates the difference between committed space and virtual space in the heap. At initialization of the virtual machine, the entire space for the heap is reserved. The size of the space reserved can be specified with the `-Xmx` option. If the value of the `-Xms` parameter is smaller than the value of the `-Xmx` parameter, then not all of the space that's reserved is immediately committed to the virtual machine. The uncommitted space is labeled "virtual" in this figure. The different parts of the heap, that is, the old generation and young generation, can grow to the limit of the virtual space as needed.

Some of the parameters are ratios of one part of the heap to another. For example, the parameter `-XX:NewRatio` denotes the relative size of the old generation to the young generation.

Figure 4-1 Heap Options



Default Option Values for Heap Size

By default, the virtual machine grows or shrinks the heap at each collection to try to keep the proportion of free space to live objects at each collection within a specific range.

This target range is set as a percentage by the options `-XX:MinHeapFreeRatio=`*<minimum>* and `-XX:MaxHeapFreeRatio=`*<maximum>*, and the total size is bounded below by `-Xms<min>` and above by `-Xmx<max>`. The default options for the 64-bit Solaris operating system (SPARC Platform Edition) are shown in Table 4-1.

Table 4-1 Default Options for 64-Bit Solaris Operating System

Option	Default Value
`-XX:MinHeapFreeRatio`	40
`-XX:MaxHeapFreeRatio`	70
`-Xms`	6656 KB
`-Xmx`	calculated

With these options, if the percent of free space in a generation falls below 40%, then the generation expands to maintain 40% free space, up to the maximum allowed size of the generation. Similarly, if the free space exceeds 70%, then the generation contracts so that only 70% of the space is free, subject to the minimum size of the generation.

As noted in Table 4-1, the default maximum heap size is a value that's calculated by the JVM. The calculation used in Java SE for the Parallel collector are now used for all the garbage collectors. Part of the calculation is an upper limit on the maximum heap size for 64-bit platforms. See Parallel Collector Default Heap Size. There's a similar calculation for the client JVM, which results in smaller maximum heap sizes than for the server JVM.

The following are general guidelines regarding heap sizes for server applications:

- Unless you have problems with pauses, try granting as much memory as possible to the virtual machine. The default size is often too small.
- Setting `-Xms` and `-Xmx` to the same value increases predictability by removing the most important sizing decision from the virtual machine. However, the virtual machine is then unable to compensate if you make a poor choice.
- In general, increase the memory as you increase the number of processors, because allocation can be made parallel.

Conserving Dynamic Footprint by Minimizing Java Heap Size

If you need to minimize the dynamic memory footprint (the maximum RAM consumed during execution) for your application, then you can do this by minimizing the Java heap size. Java SE Embedded applications may require this.

Minimize Java heap size by lowering the values of the options `-XX:MaxHeapFreeRatio` (default value is 70%) and `-XX:MinHeapFreeRatio` (default value is 40%) with the command-line options `-XX:MaxHeapFreeRatio` and `-XX:MinHeapFreeRatio`. Lowering `-XX:MaxHeapFreeRatio` to as low as 10% and `-XX:MinHeapFreeRatio` has shown to successfully reduce the heap size without too much performance degradation; however, results may vary greatly depending on your application. Try different values for these parameters until they're as low as possible, yet still retain acceptable performance.

In addition, you can specify `-XX:-ShrinkHeapInSteps`, which immediately reduces the Java heap to the target size (specified by the parameter `-XX:MaxHeapFreeRatio`). You may encounter performance degradation with this setting. By default, the Java runtime incrementally reduces the Java heap to the target size; this process requires multiple garbage collection cycles.

The Young Generation

After total available memory, the second most influential factor affecting garbage collection performance is the proportion of the heap dedicated to the young generation.

The bigger the young generation, the less often minor collections occur. However, for a bounded heap size, a larger young generation implies a smaller old generation, which will increase the frequency of major collections. The optimal choice depends on the lifetime distribution of the objects allocated by the application.

Young Generation Size Options

By default, the young generation size is controlled by the option `-XX:NewRatio`.

For example, setting `-XX:NewRatio=3` means that the ratio between the young and old generation is 1:3. In other words, the combined size of the eden and survivor spaces will be one-fourth of the total heap size.

The options `-XX:NewSize` and `-XX:MaxNewSize` bound the young generation size from below and above. Setting these to the same value fixes the young generation, just as setting `-Xms` and `-Xmx` to the same value fixes the total heap size. This is useful for tuning the young generation at a finer granularity than the integral multiples allowed by `-XX:NewRatio`.

Survivor Space Sizing

You can use the option `-XX:SurvivorRatio` to tune the size of the survivor spaces, but often this isn't important for performance.

For example, `-XX:SurvivorRatio=6` sets the ratio between eden and a survivor space to 1:6. In other words, each survivor space will be one-sixth of the size of eden, and thus one-eighth of the size of the young generation (not one-seventh, because there are two survivor spaces).

If survivor spaces are too small, then the copying collection overflows directly into the old generation. If survivor spaces are too large, then they are uselessly empty. At each garbage collection, the virtual machine chooses a threshold number, which is the number of times an object can be copied before it's old. This threshold is chosen to keep the survivors half full. You can use the log configuration `-Xlog:gc,age` can be used to show this threshold and the ages of objects in the new generation. It's also useful for observing the lifetime distribution of an application.

Table 4-2 provides the default values for 64-bit Solaris.

Table 4-2 Default Option Values for Survivor Space Sizing

Option	Default Value
`-XX:NewRatio`	2
`-XX:NewSize`	1310 MB
`-XX:MaxNewSize`	not limited
`-XX:SurvivorRatio`	8

The maximum size of the young generation is calculated from the maximum size of the total heap and the value of the `-XX:NewRatio` parameter. The "not limited" default value for the `-XX:MaxNewSize` parameter means that the calculated value isn't limited by `-XX:MaxNewSize` unless a value for `-XX:MaxNewSize` is specified on the command line.

The following are general guidelines for server applications:

- First decide on the maximum heap size that you can afford to give the virtual machine. Then, plot your performance metric against the young generation sizes to find the best setting.
  - Note that the maximum heap size should always be smaller than the amount of memory installed on the machine to avoid excessive page faults and thrashing.
- If the total heap size is fixed, then increasing the young generation size requires reducing the old generation size. Keep the old generation large enough to hold all the live data used by the application at any given time, plus some amount of slack space (10 to 20% or more).
- Subject to the previously stated constraint on the old generation:
  - Grant plenty of memory to the young generation.
  - Increase the young generation size as you increase the number of processors because allocation can be parallelized.

5

Available Collectors

The discussion to this point has been about the serial collector. The Java HotSpot VM includes three different types of collectors, each with different performance characteristics.

Topics

Serial Collector

The serial collector uses a single thread to perform all garbage collection work, which makes it relatively efficient because there is no communication overhead between threads.

It's best-suited to single processor machines because it can't take advantage of multiprocessor hardware, although it can be useful on multiprocessors for applications with small data sets (up to approximately 100 MB). The serial collector is selected by default on certain hardware and operating system configurations, or can be explicitly enabled with the option `-XX:+UseSerialGC`.

Parallel Collector

The parallel collector is also known as *throughput collector*, it's a generational collector similar to the serial collector. The primary difference between the serial and parallel collectors is that the parallel collector has multiple threads that are used to speed up garbage collection.

The parallel collector is intended for applications with medium-sized to large-sized data sets that are run on multiprocessor or multithreaded hardware. You can enable it by using the `-XX:+UseParallelGC` option.

Parallel compaction is a feature that enables the parallel collector to perform major collections in parallel. Without parallel compaction, major collections are performed using a single thread, which can significantly limit scalability. Parallel compaction is enabled by default if the option `-XX:+UseParallelGC` has been specified. You can disable it by using the `-XX:-UseParallelOldGC` option.

The Mostly Concurrent Collectors

Concurrent Mark Sweep (CMS) collector and Garbage-First (G1) garbage collector are the two mostly concurrent collectors. Mostly concurrent collectors perform some expensive work concurrently to the application.

- G1 garbage collector: This server-style collector is for multiprocessor machines with a large amount of memory. It meets garbage collection pause-time goals with high probability, while achieving high throughput.

  G1 is selected by default on certain hardware and operating system configurations, or can be explicitly enabled using`-XX:+UseG1GC` .

- CMS collector : This collector is for applications that prefer shorter garbage collection pauses and can afford to share processor resources with the garbage collection.

  Use the option `-XX:+UseConcMarkSweepGC` to enable the CMS collector

The CMS collector is deprecated as of JDK 9.

The Z Garbage Collector

The Z Garbage Collector (ZGC) is a scalable low latency garbage collector. ZGC performs all expensive work concurrently, without stopping the execution of application threads.

ZGC is intended for applications which require low latency (less than 10 ms pauses) and/or use a very large heap (multi-terabytes). You can enable is by using the `-XX:+UseZGC` option.

ZGC is available as an experimental feature, starting with JDK 11.

Selecting a Collector

Unless your application has rather strict pause-time requirements, first run your application and allow the VM to select a collector.

If necessary, adjust the heap size to improve performance. If the performance still doesn't meet your goals, then use the following guidelines as a starting point for selecting a collector:

- If the application has a small data set (up to approximately 100 MB), then select the serial collector with the option `-XX:+UseSerialGC`.
- If the application will be run on a single processor and there are no pause-time requirements, then select the serial collector with the option `-XX:+UseSerialGC`.
- If (a) peak application performance is the first priority and (b) there are no pause-time requirements or pauses of one second or longer are acceptable, then let the VM select the collector or select the parallel collector with `-XX:+UseParallelGC`.
- If response time is more important than overall throughput and garbage collection pauses must be kept shorter than approximately one second, then select a mostly concurrent collector with `-XX:+UseG1GC` or `-XX:+UseConcMarkSweepGC`.
- If response time is a high priority, and/or you are using a very large heap, then select a fully concurrent collector with `-XX:UseZGC`.

These guidelines provide only a starting point for selecting a collector because performance is dependent on the size of the heap, the amount of live data maintained by the application, and the number and speed of available processors.

If the recommended collector doesn't achieve the desired performance, then first attempt to adjust the heap and generation sizes to meet the desired goals. If

performance is still inadequate, then try a different collector: Use the concurrent collector to reduce pause-time, and use the parallel collector to increase overall throughput on multiprocessor hardware.

6
The Parallel Collector

The parallel collector (also referred to here as the *throughput collector*) is a generational collector similar to the serial collector. The primary difference between the serial and parallel collectors is that the parallel collector has multiple threads that are used to speed up garbage collection.

The parallel collector is enabled with the command-line option `-XX:+UseParallelGC`. By default, with this option, both minor and major collections are run in parallel to further reduce garbage collection overhead.

Topics

Number of Parallel Collector Garbage Collector Threads

On a machine with <*N*> hardware threads where <*N*> is greater than 8, the parallel collector uses a fixed fraction of <*N*> as the number of garbage collector threads.

The fraction is approximately 5/8 for large values of <*N*>. At values of <*N*> below 8, the number used is <*N*>. On selected platforms, the fraction drops to 5/16. The specific number of garbage collector threads can be adjusted with a command-line option (which is described later). On a host with one processor, the parallel collector will likely not perform as well as the serial collector because of the overhead required for parallel execution (for example, synchronization). However, when running applications with medium-sized to large-sized heaps, it generally outperforms the serial collector by a modest amount on computers with two processors, and usually performs significantly better than the serial collector when more than two processors are available.

The number of garbage collector threads can be controlled with the command-line option `-XX:ParallelGCThreads=`<*N*>. If you are tuning the heap with command-line options, then the size of the heap needed for good performance with the parallel collector is the same as needed with the serial collector. However, enabling the parallel collector should make the collection pauses shorter. Because multiple garbage collector threads are participating in a minor collection, some fragmentation is possible due to promotions from the young

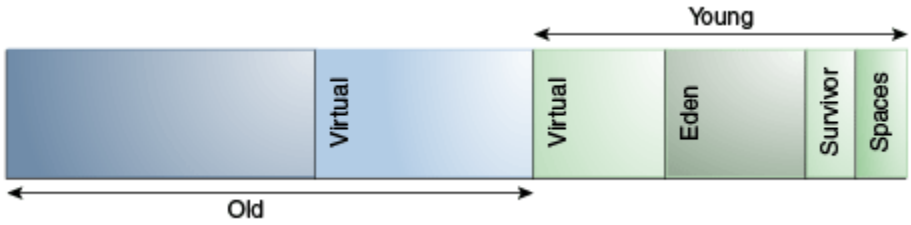
generation to the old generation during the collection. Each garbage collection thread involved in a minor collection reserves a part of the old generation for promotions and the division of the available space into these "promotion buffers" can cause a fragmentation effect. Reducing the number of garbage collector threads and increasing the size of the old generation will reduce this fragmentation effect.

Arrangement of Generations in Parallel Collectors

The arrangement of the generations is different in the parallel collector.

That arrangement is shown in Figure 6-1:

Figure 6-1 Arrangement of Generations in the Parallel Collector



Parallel Collector Ergonomics

When the parallel collector is selected by using `-XX:+UseParallelGC`, it enables a method of automatic tuning that allows you to specify behaviors instead of generation sizes and other low-level tuning details.

Options to Specify Parallel Collector Behaviors

You can specify maximum garbage collection pause time, throughput, and footprint (heap size).

- Maximum garbage collection pause time: The maximum pause time goal is specified with the command-line option `-XX:MaxGCPauseMillis=`<*N*>. This is interpreted as a hint that pause times of <*N*> milliseconds or less are desired; by default, no maximum pause- time goal. If a pause-time goal is specified, the heap size and other parameters related to garbage collection are adjusted in an attempt to keep garbage collection pauses shorter than the specified value; however, the desired pause-time goal may not always be met. These adjustments may cause the garbage collector to reduce the overall throughput of the application.
- Throughput: The throughput goal is measured in terms of the time spent doing garbage collection versus the time spent outside of garbage collection, referred to as *application time*. The goal is specified by the command-line option `-XX:GCTimeRatio=`<*N*>, which sets the ratio of garbage collection time to application time to `1 / (1 +` <*N*>`)`.

  For example, `-XX:GCTimeRatio=19` sets a goal of 1/20 or 5% of the total time in garbage collection. The default value is 99, resulting in a goal of 1% of the time in garbage collection.

- Footprint: The maximum heap footprint is specified using the option `-Xmx`<*N*>. In addition, the collector has an implicit goal of minimizing the size of the heap as long as the other goals are being met.

Priority of Parallel Collector Goals

The goals are maximum pause-time goal, throughput goal, and minimum footprint goal, and goals are addressed in that order:

The maximum pause-time goal is met first. Only after it's met is the throughput goal addressed. Similarly, only after the first two goals have been met is the footprint goal considered.

Parallel Collector Generation Size Adjustments

Statistics such as average pause time kept by the collector are updated at the end of each collection.

The tests to determine if the goals have been met are then made and any needed adjustments to the size of a generation is made. The exception is that explicit garbage collections, for example, calls to `System.gc()` are ignored in terms of keeping statistics and making adjustments to the sizes of generations.

Growing and shrinking the size of a generation is done by increments that are a fixed percentage of the size of the generation so that a generation steps up or down toward its desired size. Growing and shrinking are done at different rates. By default, a generation grows in increments of 20% and shrinks in increments of 5%. The percentage for growing is controlled by the command-line option `-XX:YoungGenerationSizeIncrement=`<*Y*> for the young generation and `-XX:TenuredGenerationSizeIncrement=`<*T*> for the old generation. The percentage by which a generation shrinks is adjusted by the command-line flag `-XX:AdaptiveSizeDecrementScaleFactor=`<*D*>. If the growth increment is *X*%, then the decrement for shrinking is *X*/*D*%.

If the collector decides to grow a generation at startup, then there's a supplemental percentage is added to the increment. This supplement decays with the number of collections and has no long-term effect. The intent of the supplement is to increase startup performance. There isn't supplement to the percentage for shrinking.

If the maximum pause-time goal isn't being met, then the size of only one generation is shrunk at a time. If the pause times of both generations are above the goal, then the size of the generation with the larger pause time is shrunk first.

If the throughput goal isn't being met, then the sizes of both generations are increased. Each is increased in proportion to its respective contribution to the total garbage collection time. For example, if the garbage collection time of the young generation is 25% of the total collection time and if a full increment of the young generation would be by 20%, then the young generation would be increased by 5%.

Parallel Collector Default Heap Size

Unless the initial and maximum heap sizes are specified on the command line, they're calculated based on the amount of memory on the machine. The default maximum heap size is one-fourth of the physical memory while the initial heap size is 1/64th of physical memory. The maximum amount of space allocated to the young generation is one third of the total heap size.

Specification of Parallel Collector Initial and Maximum Heap Sizes

You can specify the initial and maximum heap sizes using the options `-Xms` (initial heap size) and `-Xmx` (maximum heap size).

If you know how much heap your application needs to work well, then you can set `-Xms` and `-Xmx` to the same value. If you don't know, then the JVM will start by using the initial heap size and then growing the Java heap until it finds a balance between heap usage and performance.

Other parameters and options can affect these defaults. To verify your default values, use the `-XX:+PrintFlagsFinal` option and look for `-XX:MaxHeapSize` in the output. For example, on Linux or Solaris, you can run the following:

```
java -XX:+PrintFlagsFinal <GC options> -version | grep MaxHeapSize
```

Excessive Parallel Collector Time and OutOfMemoryError

The parallel collector throws an `OutOfMemoryError` if too much time is being spent in garbage collection (GC).

If more than 98% of the total time is spent in garbage collection and less than 2% of the heap is recovered, then an `OutOfMemoryError`, is thrown. This feature is designed to prevent applications from running for an extended period of time while making little or no progress because the heap is too small. If necessary, this feature can be disabled by adding the option `-XX:-UseGCOverheadLimit` to the command line.

Parallel Collector Measurements

The verbose garbage collector output from the parallel collector is essentially the same as that from the serial collector.

7

The Mostly Concurrent Collectors

The mostly concurrent collectors perform parts of their work concurrently to the application, hence their name. The Java HotSpot VM includes two mostly concurrent collectors:

- Concurrent Mark Sweep (CMS) collector: This collector is for applications that prefer shorter garbage collection pauses and can afford to share processor resources with the garbage collection.
- Garbage-First (G1) garbage collector: This server-style collector is for multiprocessor machines with a large amount of memory. It meets garbage collection pause-time goals with high probability while achieving high throughput.

Overhead of Mostly Concurrent Collectors

The mostly concurrent collector trades processor resources (which would otherwise be available to the application) for shorter major collection pause time.

The most visible overhead is the use of one or more processors during the concurrent parts of the collection. On an *N* processor system, the concurrent part of the collection uses *K*/*N* of the available processors, where 1 <= *K* <= ceiling*{N/4}*. In addition to the use of processors during concurrent phases, additional overhead is incurred to enable concurrency. Thus, while garbage collection pauses are typically much shorter with the concurrent collector, application throughput also tends to be slightly lower than with the other collectors.

On a machine with more than one processing core, processors are available for application threads during the concurrent part of the collection, so the concurrent garbage collector thread doesn't pause the application. This usually results in shorter pauses, but again fewer processor resources are available to the application and some slowdown should be expected, especially if the application uses all of the processing cores maximally. As *N* increases, the reduction in processor resources due to concurrent garbage collection becomes smaller, and the benefit from concurrent collection increases. See Concurrent Mode Failure, which discusses potential limits to such scaling.

Because at least one processor is used for garbage collection during the concurrent phases, the concurrent collectors don't normally provide any benefit on a uniprocessor (single-core) machine.

8

Concurrent Mark Sweep (CMS) Collector

The Concurrent Mark Sweep (CMS) collector is designed for applications that prefer shorter garbage collection pauses and that can afford to share processor resources with the garbage collector while the application is running.

Typically applications that have a relatively large set of long-lived data (a large old generation) and run on machines with two or more processors tend to benefit from the use of this collector. The CMS collector is enabled with the command-line option `-XX:+UseConcMarkSweepGC`.

The CMS collector is deprecated. Strongly consider using the Garbage-First collector instead.

Topics

Concurrent Mark Sweep Collector Performance and Structure

Similar to the other available collectors, the CMS collector is generational; thus both minor and major collections occur. The CMS collector attempts to reduce pause times due to major collections by using separate garbage collector threads to trace the reachable objects concurrently with the execution of the application threads.

During each major collection cycle, the CMS collector pauses all the application threads for a brief period at the beginning of the collection and again toward the middle of the collection. The second pause tends to be the longer of the two pauses. Multiple threads perform the collection work during both pauses. One or more garbage collector threads do the remainder of the collection (including most of the tracing of live objects and sweeping of unreachable objects). Minor collections can interleave with an ongoing major cycle, and are done in a manner similar to the parallel collector (in particular, the application threads are stopped during minor collections).

Concurrent Mode Failure

The CMS collector uses one or more garbage collector threads that run simultaneously with the application threads with the goal of completing the collection of the old generation before it becomes full.

As described previously, in normal operation, the CMS collector does most of its tracing and sweeping work with the application threads still running, so only brief pauses are seen by the application threads. However, if the CMS collector is unable to finish reclaiming the unreachable objects before the old generation fills up, or if an allocation cannot be satisfied with the available free space blocks in the old generation, then the application is paused and the collection is completed with all the application threads stopped. The inability to complete a collection concurrently is referred to as *concurrent mode failure* and indicates the need to adjust the CMS collector parameters. If a concurrent collection is interrupted by an explicit garbage collection (`System.gc()`) or for a garbage collection needed to provide information for diagnostic tools, then a concurrent mode interruption is reported.

Excessive GC Time and OutOfMemoryError

The CMS collector throws an `OutOfMemoryError` if too much time is being spent in garbage collection: If more than 98% of the total time is spent in garbage collection and less than 2% of the heap is recovered, then an `OutOfMemoryError` is thrown.

This feature is designed to prevent applications from running for an extended period of time while making little or no progress because the heap is too small. If necessary, this feature can be disabled by adding the option `-XX:-UseGCOverheadLimit` to the command line.

The policy is the same as that in the parallel collector, except that time spent performing concurrent collections isn't counted toward the 98% time limit. In other words, only collections performed while the application is stopped count toward excessive GC time. Such collections are typically due to a concurrent mode failure or an explicit collection request (for example, a call to `System.gc()`).

Concurrent Mark Sweep Collector and Floating Garbage

The CMS collector, like all the other collectors in Java HotSpot VM, is a tracing collector that identifies at least all the reachable objects in the heap.

Richard Jones and Rafael D. Lins in their publication *Garbage Collection: Algorithms for Automated Dynamic Memory*, it's an incremental update collector. Because application threads and the garbage collector thread run concurrently during a major collection, objects that are traced by the garbage collector thread may subsequently become unreachable by the time collection process ends. Such unreachable objects that haven't yet been reclaimed are referred to as *floating garbage*. The amount of floating garbage depends on the duration of the concurrent collection cycle and on the frequency of reference updates, also known as *mutations*, by the application. Furthermore, because the young generation and the old generation are collected independently, each acts as a source of roots to the other. As a rough guideline, try increasing the size of the old generation by 20% to account for the floating garbage. Floating garbage in the heap at the end of one concurrent collection cycle is collected during the next collection cycle.

Concurrent Mark Sweep Collector Pauses

The CMS collector pauses an application twice during a concurrent collection cycle. The first pause is to mark as live the objects directly reachable from the roots (for example, object references from application thread stacks and registers, static objects, and so on) and from elsewhere in the heap (for example, the young generation).

This first pause is referred to as the *initial mark pause*. The second pause comes at the end of the concurrent tracing phase and finds objects that were missed by the concurrent tracing due to updates by the application threads of references in an object after the CMS collector had finished tracing that object. This second pause is referred to as the *remark pause*.

Concurrent Mark Sweep Collector Concurrent Phases

The concurrent tracing of the reachable object graph occurs between the initial mark pause and the remark pause.

During this concurrent tracing phase, one or more concurrent garbage collector threads may be using processor resources that would otherwise have been available to the application. As a result, compute-bound applications may see a commensurate decrease in application throughput during this and other concurrent phases even though the application threads aren't paused. After the remark pause, a concurrent sweeping phase collects the objects identified as unreachable. After a collection cycle completes, the CMS collector waits, consuming almost no computational resources, until the start of the next major collection cycle.

Starting a Concurrent Collection Cycle

With the serial collector a major collection occurs whenever the old generation becomes full and all application threads are stopped while the collection is done. In contrast, the start of a concurrent collection in CMS collector must be timed such that the collection can finish before the old generation becomes full; otherwise, the application would observe longer pauses due to concurrent mode failure. There are several ways to start a concurrent collection.

Based on recent history, the CMS collector maintains estimates of the time remaining before the old generation will be exhausted and of the time needed for a concurrent collection cycle. Using these dynamic estimates, a concurrent collection cycle is started with the aim of completing the collection cycle before the old generation is exhausted. These estimates are padded for safety because concurrent mode failure can be very costly.

A concurrent collection also starts if the occupancy of the old generation exceeds an initiating occupancy (a percentage of the old generation). The default value for this initiating occupancy threshold is approximately 92%, but the value is subject to change from release to release. This value can be manually adjusted using the command-line option `-XX:CMSInitiatingOccupancyFraction=<N>`, where `<N>` is an integral percentage (0 to 100) of the old generation size.

Scheduling Pauses

The pauses for the young generation collection and the old generation collection occur independently.

They don't overlap, but may occur in quick succession such that the pause from one collection, immediately followed by one from the other collection, can appear to be a single, longer pause. To avoid this, the CMS collector attempts to schedule the remark pause roughly midway between the previous and next young generation pauses. This scheduling is currently not done for the initial mark pause, which is usually much shorter than the remark pause.

Concurrent Mark Sweep Collector Measurements

The following is the output from the CMS collector with the option `-Xlog:gc`:

```
[121,834s][info][gc] GC(657) Pause Initial Mark 191M->191M(485M)
(121,831s, 121,834s) 3,433ms
[121,835s][info][gc] GC(657) Concurrent Mark (121,835s)
[121,889s][info][gc] GC(657) Concurrent Mark (121,835s, 121,889s)
54,330ms
[121,889s][info][gc] GC(657) Concurrent Preclean (121,889s)
[121,892s][info][gc] GC(657) Concurrent Preclean (121,889s, 121,892s)
2,781ms
[121,892s][info][gc] GC(657) Concurrent Abortable Preclean (121,892s)
[121,949s][info][gc] GC(658) Pause Young (Allocation Failure) 324M-
>199M(485M) (121,929s, 121,949s) 19,705ms
[122,068s][info][gc] GC(659) Pause Young (Allocation Failure) 333M-
>200M(485M) (122,043s, 122,068s) 24,892ms
[122,075s][info][gc] GC(657) Concurrent Abortable Preclean (121,892s,
122,075s) 182,989ms
[122,087s][info][gc] GC(657) Pause Remark 209M->209M(485M) (122,076s,
122,087s) 11,373ms
[122,087s][info][gc] GC(657) Concurrent Sweep (122,087s)
[122,193s][info][gc] GC(660) Pause Young (Allocation Failure) 301M-
>165M(485M) (122,181s, 122,193s) 12,151ms
[122,254s][info][gc] GC(657) Concurrent Sweep (122,087s, 122,254s)
166,758ms
[122,254s][info][gc] GC(657) Concurrent Reset (122,254s)
[122,255s][info][gc] GC(657) Concurrent Reset (122,254s, 122,255s)
0,952ms
[122,297s][info][gc] GC(661) Pause Young (Allocation Failure) 259M-
>128M(485M) (122,291s, 122,297s) 5,797ms
```

Note:

The output for the CMS collection (GC ID 657) is interspersed with the output from the minor collections (GC IDs 658, 659 and 660); typically many minor collections occur during a concurrent collection cycle. Pause Initial Mark indicates the start of the concurrent collection cycle. The lines starting with "Concurrent" indicate the start and end of the concurrent phases. Pause Remark is the final pause. Not discussed previously is the precleaning phases. Precleaning represents work that can be done concurrently in preparation for the remark phase. The final phase is indicated by Concurrent Reset and is in preparation for the next concurrent collection.

The initial mark pause is typically short relative to the minor collection pause time. The concurrent phases (concurrent mark, concurrent preclean, and concurrent sweep) normally last significantly longer than a minor collection pause, as indicated in the CMS collector output example. Note, however, that the application isn't paused during these concurrent phases. The remark pause is often comparable in length to a minor collection. The remark pause is affected by certain application characteristics (for example, a high rate of object modification can increase this pause) and the time since the last minor collection (for example, more objects in the young generation may increase this pause).

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).

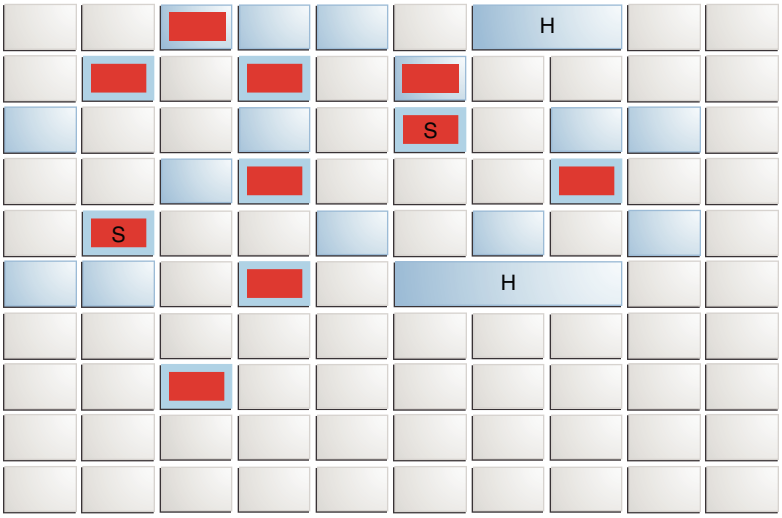
G1 reclaims space mostly by using evacuation: live objects found within selected memory areas to collect are copied into new memory areas, compacting them in the process. After an evacuation has been completed, the space previously occupied by live objects is reused for allocation by the application.

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



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.

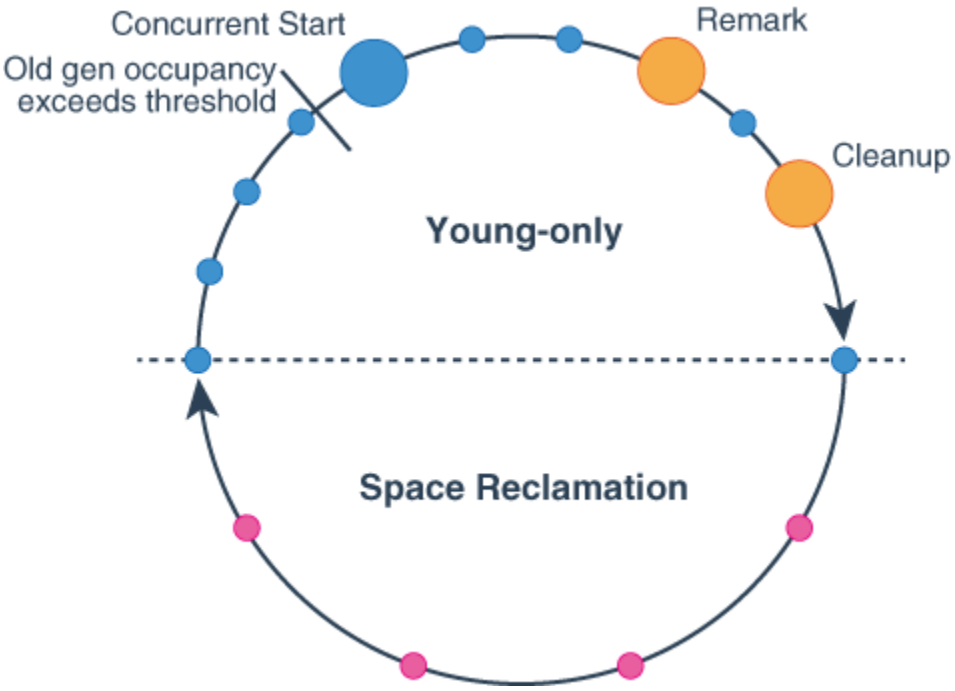
G1 garbage collection pauses can reclaim space in the young generation as a whole, and any additional set of old generation regions at any collection pause. During the pause G1 copies objects from this *collection set* to one or more different regions in the heap. The destination region for an object 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



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 Normal young 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 a Concurrent Start young collection instead of a Normal young collection.
    - Concurrent Start : This type of collection starts the marking process in addition to performing a Normal young collection. Concurrent marking determines all currently reachable (live) objects in the old generation regions to be kept for the following space-reclamation phase. While collection marking hasn't completely finished, Normal young collections may occur. Marking finishes with two special stop-the-world pauses: Remark and Cleanup.
    - Remark: This pause finalizes the marking itself, performs global reference processing and class unloading, reclaims completely empty regions and cleans up internal data structures. Between Remark and Cleanup G1 calculates information to later be able to reclaim free space in selected old generation regions concurrently, which will be finalized in the Cleanup pause.
    - Cleanup: This pause determines whether a space-reclamation phase will actually follow. If a space-reclamation phase follows, the young-only phase completes with a single Prepare Mixed young 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 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 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 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 normal young collection for the next mutator phase. 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 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 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. At the start of every pause, the maximum number of threads used is further constrained by maximum total heap size: G1 will not use more than one thread per `-XX:HeapSizePerGCThread` amount of Java heap capacity.
`-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.

Table 9-1 (Cont.) Ergonomic Defaults G1 GC

Option and Default Value	Description
`-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`.

10

Garbage-First Garbage Collector Tuning

This section describes how to adapt Garbage-First garbage collector (G1 GC) behavior in case it does not meet your requirements.

Topics

General Recommendations for G1

The general recommendation is to use G1 with its default settings, eventually giving it a different pause-time goal and setting a maximum Java heap size by using `-Xmx` if desired.

G1 defaults have been balanced differently than either of the other collectors. G1's goals in the default configuration are neither maximum throughput nor lowest latency, but to provide relatively small, uniform pauses at high throughput. However, G1's mechanisms to incrementally reclaim space in the heap and the pause-time control incur some overhead in both the application threads and in the space-reclamation efficiency.

If you prefer high throughput, then relax the pause-time goal by using `-XX:MaxGCPauseMillis` or provide a larger heap. If latency is the main requirement, then modify the pause-time target. Avoid limiting the young generation size to particular values by using options like `-Xmn`, `-XX:NewRatio` and others because the young generation size is the main means for G1 to allow it to meet the pause-time. Setting the young generation size to a single value overrides and practically disables pause-time control.

Moving to G1 from Other Collectors

Generally, when moving to G1 from other collectors, particularly the Concurrent Mark Sweep collector, start by removing all options that affect garbage collection, and only set the pause-time goal and overall heap size by using `-Xmx` and optionally `-Xms`.

Many options that are useful for other collectors to respond in some particular way, have either no effect at all, or even decrease throughput and the likelihood to meet the pause-time target. An example could be setting young generation sizes that completely prevent G1 from adjusting the young generation size to meet pause-time goals.

Improving G1 Performance

G1 is designed to provide good overall performance without the need to specify additional options. However, there are cases when the default heuristics or default configurations for them provide suboptimal results. This section gives some guidelines about diagnosing and improving in these cases. This guide describes only the possibilities that G1 provides to improve garbage collector performance in a selected metric, when given a set application. On a case-by-case basis, application-level optimizations could be more effective than trying to tune the VM to perform better, for example, by avoiding some problematic situations by less long-lived objects altogether.

For diagnosis purposes, G1 provides comprehensive logging. A good start is to use the `-Xlog:gc*=debug` option and then refine the output from that if necessary. The log provides a detailed overview during and outside the pauses about garbage collection activity. This includes the type of collection and a breakdown of time spent in particular phases of the pause.

The following subsections explore some common performance issues.

Observing Full Garbage Collections

A full heap garbage collection (Full GC) is often very time consuming. Full GCs caused by too high heap occupancy in the old generation can be detected by finding the words *Pause Full (Allocation Failure)* in the log. Full GCs are typically preceded by garbage collections that encounter an evacuation failure indicated by `to-space exhausted` tags.

The reason that a Full GC occurs is because the application allocates too many objects that can't be reclaimed quickly enough. Often concurrent marking has not been able to complete in time to start a space-reclamation phase. The probability to run into a Full GC can be compounded by the allocation of many humongous objects. Due to the way these objects are allocated in G1, they may take up much more memory than expected.

The goal should be to ensure that concurrent marking completes on time. This can be achieved either by decreasing the allocation rate in the old generation, or giving the concurrent marking more time to complete.

G1 gives you several options to handle this situation better:

- You can determine the number of regions occupied by humongous objects on the Java heap using the `gc+heap=info` logging. `Y` in the lines "`Humongous regions: X->Y`" give you the amount of regions occupied by humongous objects. If this number is high compared to the number of old regions, the best option is to try to decrease this number of objects. You can achieve this by increasing the region size using the `-XX:G1HeapRegionSize` option. The currently selected heap region size is printed at the beginning of the log.
- Increase the size of the Java heap. This typically increases the amount of time marking has to complete.
- Increase the number of concurrent marking threads by setting `-XX:ConcGCThreads` explicitly.
- Force G1 to start marking earlier. G1 automatically determines the Initiating Heap Occupancy Percent (IHOP) threshold based on earlier application behavior. If the application behavior changes, these predictions might be wrong. There are two options: Lower the target occupancy for when to start space-reclamation by increasing the buffer used in an adaptive IHOP calculation by modifying `-XX:G1ReservePercent`; or, disable the adaptive calculation of the IHOP by setting it manually using `-XX:-G1UseAdaptiveIHOP` and `-XX:InitiatingHeapOccupancyPercent`.

Other causes than Allocation Failure for a Full GC typically indicate that either the application or some external tool causes a full heap collection. If the cause is `System.gc()`, and there is no way to modify the application sources, the effect of Full GCs can be mitigated by using `-XX:+ExplicitGCInvokesConcurrent` or let the VM completely ignore them by setting `-XX:+DisableExplicitGC`. External tools may still force Full GCs; they can be removed only by not requesting them.

Humongous Object Fragmentation

A Full GC could occur before all Java heap memory has been exhausted due to the necessity of finding a contiguous set of regions for them. Potential options in this case are increasing the heap region size by using the option `-XX:G1HeapRegionSize` to decrease the number of humongous objects, or increasing size of the heap. In extreme cases, there might not be enough contiguous space available for G1 to allocate the object even if available memory indicates otherwise. This would lead to a VM exit if that Full GC can not reclaim enough contiguous space. As a result, there are no other options than either decreasing the amount of humongous object allocations as mentioned previously, or increasing the heap.

Tuning for Latency

This section discusses hints to improve G1 behavior in case of common latency problems that is, if the pause-time is too high.

Unusual System or Real-Time Usage

For every garbage collection pause, the `gc+cpu=info` log output contains a line including information from the operating system with a breakdown about where during the pause-time has been spent. An example for such output is `User=0.19s Sys=0.00s Real=0.01s`.

User time is time spent in VM code, *system time* is the time spent in the operating system, and *real time* is the amount of absolute time passed during the pause. If the system time is relatively high, then most often the environment is the cause.

Common known issues for high system time are:

- The VM allocating or giving back memory from the operating system memory may cause unnecessary delays. Avoid the delays by setting minimum and maximum heap sizes to the same value using the options `-Xms` and `-Xmx`, and pre-touching all memory using `-XX:+AlwaysPreTouch` to move this work to the VM startup phase.
- Particularly in Linux, coalescing of small pages into huge pages by the *Transparent Huge Pages (THP)* feature tends to stall random processes, not just during a pause. Because the VM allocates and maintains a lot of memory, there is a higher than usual risk that the VM will be the process that stalls for a long time. Refer to the documentation of your operating system on how to disable the Transparent Huge Pages feature.
- Writing the log output may stall for some time because of some background task intermittently taking up all I/O bandwidth for the hard disk the log is written to. Consider using a separate disk for your logs or some other storage, for example memory-backed file system to avoid this.

Another situation to look out for is real time being a lot larger than the sum of the others this may indicate that the VM did not get enough CPU time on a possibly overloaded machine.

Reference Object Processing Takes Too Long

Information about the time taken for processing of Reference Objects is shown in the `Reference Processing` phase. During the `Reference Processing` phase, G1 updates the referents of Reference Objects according to the requirements of the particular type of Reference Object. By default, G1 tries to parallelize the sub-phases of `Reference Processing` using the following heuristic: for every `-XX:ReferencesPerThread` reference Objects start a single thread, bounded by the value in `-XX:ParallelGCThreads`. This heuristic can be disabled by setting `-XX:ReferencesPerThread` to 0 to use all available threads by default, or parallelization disabled completely by `-XX:-ParallelRefProcEnabled`.

Young-Only Collections Within the Young-Only Phase Take Too Long

Normal young and, in general any young collection roughly takes time proportional to the size of the young generation, or more specifically, the number of live objects within the collection set that needs to be copied. If the *Evacuate Collection Set* phase takes too long, in particular, the *Object Copy* sub-phase, decrease `-XX:G1NewSizePercent`. This decreases the minimum size of the young generation, allowing for potentially shorter pauses.

Another problem with sizing of the young generation may occur if application performance, and in particular the amount of objects surviving a collection, suddenly changes. This may cause spikes in garbage collection pause time. It might be useful to decrease the maximum young generation size by using `-XX:G1MaxNewSizePercent`. This limits the maximum size of the young generation and so the number of objects that need to be processed during the pause.

Mixed Collections Take Too Long

Mixed collections are used to reclaim space in the old generation. The collection set of mixed collections contains young and old generation regions. You can obtain information about how much time evacuation of either young or old generation regions

contribute to the pause-time by enabling the `gc+ergo+cset=trace` log output. Look at the *predicted young region* time and *predicted old region* time for young and old generation regions respectively.

If the predicted young region time is too long, then see Young-Only Collections Within the Young-Only Phase Take Too Long for options. Otherwise, to reduce the contribution of the old generation regions to the pause-time, G1 provides three options:

- Spread the old generation region reclamation across more garbage collections by increasing `-XX:G1MixedGCCountTarget`.
- Avoid collecting regions that take a proportionally large amount of time to collect by not putting them into the candidate collection set by using `-XX:G1MixedGCLiveThresholdPercent`. In many cases, highly occupied regions take a lot of time to collect.
- Stop old generation space reclamation earlier so that G1 won't collect as many highly occupied regions. In this case, increase `-XX:G1HeapWastePercent`.

Note that the last two options decrease the amount of collection set candidate regions where space can be reclaimed for the current space-reclamation phase. This may mean that G1 may not be able to reclaim enough space in the old generation for sustained operation. However, later space-reclamation phases may be able to garbage collect them.

High Update RS and Scan RS Times

To enable G1 to evacuate single old generation regions, G1 tracks locations of *cross-region references*, that is references that point from one region to another. The set of cross-region references pointing into a given region is called that region's *remembered set*. The remembered sets must be updated when moving the contents of a region. Maintenance of the regions' remembered sets is mostly concurrent. For performance purposes, G1 doesn't immediately update the remembered set of a region when the application installs a new cross-region reference between two objects. Remembered set update requests are delayed and batched for efficiency.

G1 requires complete remembered sets for garbage collection, so the *Update RS* phase of the garbage collection processes any outstanding remembered set update requests. The *Scan RS* phase searches for object references in remembered sets, moves region contents, and then updates these object references to the new locations. Depending on the application, these two phases may take a significant amount of time.

Adjusting the size of the heap regions by using the option `-XX:G1HeapRegionSize` affects the number of cross-region references and as well as the size of the remembered set. Handling the remembered sets for regions may be a significant part of garbage collection work, so this has a direct effect on the achievable maximum pause time. Larger regions tend to have fewer cross-region references, so the relative amount of work spent in processing them decreases, although at the same time, larger regions may mean more live objects to evacuate per region, increasing the time for other phases.

G1 tries to schedule concurrent processing of the remembered set updates so that the Update RS phase takes approximately `-XX:G1RSetUpdatingPauseTimePercent` percent of the allowed maximum pause time. By decreasing this value, G1 usually performs more remembered set update work concurrently.

Spurious high Update RS times in combination with the application allocating large objects may be caused by an optimization that tries to reduce concurrent remembered set update work by batching it. If the application that created such a batch happens just before a garbage collection, then the garbage collection must process all this work in the Update RS

times part of the pause. Use `-XX:-ReduceInitialCardMarks` to disable this behavior and potentially avoid these situations.

Scan RS Time is also determined by the amount of compression that G1 performs to keep remembered set storage size low. The more compact the remembered set is stored in memory, the more time it takes to retrieve the stored values during garbage collection. G1 automatically performs this compression, called remembered set coarsening, while updating the remembered sets depending on the current size of that region's remembered set. Particularly at the highest compression level, retrieving the actual data can be very slow. The option `-XX:G1SummarizeRSetStatsPeriod` in combination with `gc+remset=trace` level logging shows if this coarsening occurs. If so, then the `X` in the line `Did <X> coarsenings` in the *Before GC Summary* section shows a high value. The `-XX:G1RSetRegionEntries` option could be increased significantly to decrease the amount of these coarsenings. Avoid using this detailed remembered set logging in production environments as collecting this data can take a significant amount of time.

Tuning for Throughput

G1's default policy tries to maintain a balance between throughput and latency; however, there are situations where higher throughput is desirable. Apart from decreasing the overall pause-times as described in the previous sections, the frequency of the pauses could be decreased. The main idea is to increase the maximum pause time by using `-XX:MaxGCPauseMillis`. The generation sizing heuristics will automatically adapt the size of the young generation, which directly determines the frequency of pauses. If that does not result in expected behavior, particularly during the space-reclamation phase, increasing the minimum young generation size using `-XX:G1NewSizePercent` will force G1 to do that.

In some cases, `-XX:G1MaxNewSizePercent`, the maximum allowed young generation size, may limit throughput by limiting young generation size. This can be diagnosed by looking at region summary output of `gc+heap=info` logging. In this case the combined percentage of Eden regions and Survivor regions is close to `-XX:G1MaxNewSizePercent` percent of the total number of regions. Consider increasing `-XX:G1MaxNewSizePercent` in this case.

Another option to increase throughput is to try to decrease the amount of concurrent work in particular, concurrent remembered set updates often require a lot of CPU resources. Increasing `-XX:G1RSetUpdatingPauseTimePercent` moves work from concurrent operation into the garbage collection pause. In the worst case, concurrent remembered set updates can be disabled by setting `-XX:-G1UseAdaptiveConcRefinement -XX:G1ConcRefinementGreenZone=2G -XX:G1ConcRefinementThreads=0`. This mostly disables this mechanism and moves all remembered set update work into the next garbage collection pause.

Enabling the use of large pages by using `-XX:+UseLargePages` may also improve throughput. Refer to your operating system documentation on how to set up large pages.

You can minimize heap resizing work by disabling it; set the options `-Xms` and `-Xmx` to the same value. In addition, you can use `-XX:+AlwaysPreTouch` to move the operating system work to back virtual memory with physical memory to VM startup time. Both of these measures can be particularly desirable in order to make pause-times more consistent.

Tuning for Heap Size

Like other collectors, G1 aims to size the heap so that the time spent in garbage collection is below the ratio determined by the `-XX:GCTimeRatio` option. Adjust this option to make G1 meet your requirements.

Tunable Defaults

This section describes the default values and some additional information about command-line options that are introduced in this topic.

Table 10-1 Tunable Defaults G1 GC

Option and Default Value	Description
`-XX:+G1UseAdaptiveConcRefinement` `-XX:G1ConcRefinementGreenZone=`*<ergo>* `-XX:G1ConcRefinementYellowZone=`*<ergo>* `-XX:G1ConcRefinementRedZone=`*<ergo>* `-XX:G1ConcRefinementThreads=`*<ergo>*	The concurrent remembered set update (refinement) uses these options to control the work distribution of concurrent refinement threads. G1 chooses the ergonomic values for these options so that `-XX:G1RSetUpdatingPauseTimePercent` time is spent in the garbage collection pause for processing any remaining work, adaptively adjusting them as needed. Change with caution because this may cause extremely long pauses.
`-XX:+ReduceInitialCardMarks`	This batches together concurrent remembered set update (refinement) work for initial object allocations.
`-XX:+ParallelRefProcEnabled` `-XX:ReferencesPerThread=1000`	`-XX:ReferencesPerThread` determines the degree of parallelization: for every *N* Reference Objects one thread will participate in the sub-phases of Reference Processing, limited by `-XX:ParallelGCThreads`. A value of 0 indicates that the maximum number of threads as indicated by the value of `-XX:ParallelGCThreads` will always be used. This determines whether processing of `java.lang.Ref.*` instances should be done in parallel by multiple threads.
`-XX:G1RSetUpdatingPauseTimePercent=10`	This determines the percentage of total garbage collection time G1 should spend in the Update RS phase updating any remaining remembered sets. G1 controls the amount of concurrent remembered set updates using this setting.
`-XX:G1SummarizeRSetStatsPeriod=0`	This is the period in a number of GCs that G1 generates remembered set summary reports. Set this to zero to disable. Generating remembered set summary reports is a costly operation, so it should be used only if necessary, and with a reasonably high value. Use `gc+remset=trace` to print anything.

Table 10-1 (Cont.) Tunable Defaults G1 GC

Option and Default Value	Description
`-XX:GCTimeRatio=12`	This is the divisor for the target ratio of time that should be spent in garbage collection as opposed to the application. The actual formula for determining the target fraction of time that can be spent in garbage collection before increasing the heap is `1 / (1 + GCTimeRatio)`. This default value results in a target with about 8% of the time to be spent in garbage collection.

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

11

The Z Garbage Collector

The Z Garbage Collector (ZGC) is a scalable low latency garbage collector. ZGC performs all expensive work concurrently, without stopping the execution of application threads for more than 10ms, which makes is suitable for applications which require low latency and/or use a very large heap (multi-terabytes).

The Z Garbage Collector is available as an experimental feature, and is enabled with the command-line options `-XX:+UnlockExperimentalVMOptions -XX:+UseZGC`.

Setting the Heap Size

The most important tuning option for ZGC is setting the max heap size `(-Xmx)`. Since ZGC is a concurrent collector a max heap size must be selected such that, 1) the heap can accommodate the live-set of your application, and 2) there is enough headroom in the heap to allow allocations to be serviced while the GC is running. How much headroom is needed very much depends on the allocation rate and the live-set size of the application. In general, the more memory you give to ZGC the better. But at the same time, wasting memory is undesirable, so it's all about finding a balance between memory usage and how often the GC needs to run.

Setting Number of Concurrent GC Threads

The second tuning option one might want to look at is setting the number of concurrent GC threads `(-XX:ConcGCThreads)`. ZGC has heuristics to automatically select this number. This heuristic usually works well but depending on the characteristics of the application this might need to be adjusted. This option essentially dictates how much CPU-time the GC should be given. Give it too much and the GC will steal too much CPU-time from the application. Give it too little, and the application might allocate garbage faster than the GC can collect it.

12

Other Considerations

This section covers other situations that affect garbage collection.

Topics

Finalization and Weak, Soft, and Phantom References

Some applications interact with garbage collection by using finalization and weak, soft, or phantom references.

However, the use of finalization is discouraged. It can lead to problems with security, performance, and reliability. For instance, relying on finalization to close file descriptors makes an external resource (descriptors) dependent on garbage collection promptness.

> **Note:**
>
> Finalization has been deprecated in JDK 9.

Finalization

A class can declare a finalizer – the method `protected void finalize()` – whose body releases any underlying resources. The GC will schedule the finalizer of an unreachable object, which is called before the GC reclaims the object's memory.

An object becomes unreachable, and thus eligible for garbage collection, when there's no path from a GC root to the object. GC roots include references from an active thread and internal JVM references; they are the references that keep objects in memory.

See Monitoring the Objects Pending Finalization in *Java Platform, Standard Edition Troubleshooting Guide* to determine if finalizable objects are building up in your system. In addition, you can use one of these tools:

- JDK Mission Control:
  1. In the **JVM Browser**, right-click your JVM and select **Start JMX Console**.
  2. In the **MBean Browser**, in the **MBean Tree**, expand **java.lang** and select **Memory**.
  3. In **MBean Features**, the attribute **ObjectPendingFinalizationCount** is the approximate number of objects that are pending finalization.
- `jcmd` tool:

– Run the following command to print information about the Java finalization queue; the value `<pid>` is the PID of your JVM:

```
jcmd <pid> GC.finalizer_info
```

Migrating from Finalization

To avoid finalization, use one of the following techniques:

- The try-with-Resources Statement
- The Cleaner API

The try-with-Resources Statement

The `try`-with-resources statement is a `try` statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The `try`-with-resources statement ensures that each resource is closed at the end of the code block, even if one or more exceptions occur. See The Try-with-resources Statement for more information.

The Cleaner API

If you foresee that the lifecycle of a resource in your application will live beyond the scope of a `try`-with-resources statement, then you can use the Cleaner API instead. The Cleaner API allows a program to register a cleaning action for an object that is run some time after the object becomes unreachable.

Cleaners enable you to avoid many of the drawbacks of finalizers:

- More secure: A cleaner must explicitly register an object. In addition, cleaning actions cannot access it so object resurrection is impossible.
- Better performance: You have more control over when you register a cleaning action, which means a cleaning action never processes an uninitialized or partially initialized object. You can also cancel an object's cleaning action.
- More reliable: You can control which threads run cleaning actions.

However, like finalizers, the garbage collector schedules cleaning actions, so they may suffer from unbounded delays. Thus, don't use the cleaner API in situations where the timely release of a resource is required.

The following is a simple example of a cleaner. It does the following:

1. Defines a cleaning action class, `State`, which initializes the cleaning action and defines the cleaning action itself (by overriding the `State::run()` method).
2. Creates an instance of `Cleaner`.
3. With this instance of `Cleaner`, registers the object `myObject1` and a cleaning action (an instance of `State`).
4. To ensure that the garbage collector schedules the cleaner and the cleaning action `State::run()` is performed before the example ends, the example:
   a. Sets `myObject1` to `null` to ensure it is phantom unreachable. See .
   b. Calls `System.gc()` in a loop to trigger garbage collection cleanup.

Figure 12-1 CleanerExample

```
import java.lang.ref.Cleaner;

public class CleanerExample {

    // This Cleaner is shared by all CleanerExample instances
    private static final Cleaner CLEANER = Cleaner.create();
    private final State state;

    public CleanerExample(String id) {
        state = new State(id);
        CLEANER.register(this, state);
    }

    // Cleaning action class for CleanerExample
    private static class State implements Runnable {
        final private String id;

        private State(String id) {
            this.id = id;
            System.out.println("Created cleaning action for " + this.id);
        }

        @Override
        public void run() {
            System.out.println("Cleaner garbage collected " + this.id);
        }
    }

    public static void main(String[] args) {
        CleanerExample myObject1 = new CleanerExample("myObject1");

        // Make myObject1 unreachable
        myObject1 = null;

        System.out.println("-- Give the GC a chance to schedule the Cleaner
--");
        for (int i = 0; i < 100; i++) {

            // Calling System.gc() in a loop is usually sufficient to trigger
            // cleanup in a small program like this.
            System.gc();
            try {
                Thread.sleep(1);
            } catch (InterruptedException e) {}
        }
        System.out.println("-- Finished --");
    }
}
```

This example prints the following:

```
Created cleaning action for myObject1
-- Give the GC a chance to schedule the Cleaner --
Cleaner garbage collected myObject1
-- Finished --
```

Consider the following if you're implementing a cleaner for a production environment:

- The cleaning action class (`State` in this example) should be a private implementation detail. In particular, it shouldn't be used from the `main(String[])` method. Thus, your cleaning action class should be immutable whenever practical. A new object should handle creating its own cleaning action class and registering itself with a cleaner within its constructor.
- Classes typically need access to objects within the cleaner action class. The simplest way to do this is for the object to save a reference to the cleaner action class.
- `Cleaner` instances should be shared. In this example, all instances of `CleanerExample` should share a single, static `Cleaner` instance.

See the JavaDoc API documentation for the `Cleaner` class for more information about implementing a cleaner.

Reference-Object Types

There are three reference-object types: `SoftReference`, `WeakReference`, and `PhantomReference`. Each reference-object type corresponds to a different level of reachability. The following are the different levels of reachability, from strongest to weakest, which reflect the life cycle of an object:

- An object is *strongly reachable* if it can be reached by some thread without traversing any reference objects. A newly-created object is strongly reachable by the thread that created it.
- An object is *softly reachable* if it is not strongly reachable but can be reached by traversing a soft reference.
- An object is *weakly reachable* if it is neither strongly nor softly reachable but can be reached by traversing a weak reference. When the weak references to a weakly-reachable object are cleared, the object becomes eligible for finalization.
- An object is *phantom reachable* if it is neither strongly, softly, nor weakly reachable, it has been finalized, and some phantom reference refers to it.
- An object is *unreachable*, and therefore eligible for reclamation, when it is not reachable in any of the previous ways.

Each reference-object type encapsulates a single reference to a particular object, which is called the *referent*. A reference object provides methods for clearing the referent.

The following are the most common uses for reference-object instances:

- To maintain access to an object while still allowing it to be garbage collected if the system needs to free up memory (such as a cached value that can be regenerated if required)

- To determine and perhaps take some action when an object has reached a particular reachability level (in combination with the `ReferenceQueue` class)

Explicit Garbage Collection

Another way that applications can interact with garbage collection is by calling full garbage collections explicitly by using `System.gc()`.

This can force a major collection to be done when it may not be necessary (for example, when a minor collection would suffice), and so in general should be avoided. The performance effect of explicit garbage collections can be measured by disabling them using the flag `-XX:+DisableExplicitGC`, which causes the VM to ignore calls to `System.gc()`.

One of the most commonly encountered uses of explicit garbage collection occurs with the distributed garbage collection (DGC) of Remote Method Invocation (RMI). Applications using RMI refer to objects in other virtual machines. Garbage cannot be collected in these distributed applications without occasionally invoking garbage collection of the local heap, so RMI forces full collections periodically. The frequency of these collections can be controlled with properties, as in the following example:

```
java -Dsun.rmi.dgc.client.gcInterval=3600000
    -Dsun.rmi.dgc.server.gcInterval=3600000 ...
```

This example specifies explicit garbage collection once per hour instead of the default rate of once per minute. However, this may also cause some objects to take much longer to be reclaimed. These properties can be set as high as `Long.MAX_VALUE` to make the time between explicit collections effectively infinite if there's no desire for an upper bound on the timeliness of DGC activity.

Soft References

Soft references are kept alive longer in the server virtual machine than in the client.

The rate of clearing can be controlled with the command-line option `-XX:SoftRefLRUPolicyMSPerMB=<N>`, which specifies the number of milliseconds (ms) a soft reference will be kept alive (once it is no longer strongly reachable) for each megabyte of free space in the heap. The default value is 1000 ms per megabyte, which means that a soft reference will survive (after the last strong reference to the object has been collected) for 1 second for each megabyte of free space in the heap. This is an approximate figure because soft references are cleared only during garbage collection, which may occur sporadically.

Class Metadata

Java classes have an internal representation within Java Hotspot VM and are referred to as class metadata.

In previous releases of Java Hotspot VM, the class metadata was allocated in the so-called permanent generation. Starting with JDK 8, the permanent generation was removed and the class metadata is allocated in native memory. The amount of native memory that can be used for class metadata is by default unlimited. Use the option `-XX:MaxMetaspaceSize` to put an upper limit on the amount of native memory used for class metadata.

Java Hotspot VM explicitly manages the space used for metadata. Space is requested from the OS and then divided into chunks. A class loader allocates space for metadata from its chunks (a chunk is bound to a specific class loader). When classes are unloaded for a class loader, its chunks are recycled for reuse or returned to the OS. Metadata uses space allocated by `mmap`, not by `malloc`.

If `-XX:UseCompressedOops` is turned on and `-XX:UseCompressedClassesPointers` is used, then two logically different areas of native memory are used for class metadata. `-XX:UseCompressedClassPointers` uses a 32-bit offset to represent the class pointer in a 64-bit process as does `-XX:UseCompressedOops` for Java object references. A region is allocated for these compressed class pointers (the 32-bit offsets). The size of the region can be set with `-XX:CompressedClassSpaceSize` and is 1 gigabyte (GB) by default. The space for the compressed class pointers is reserved as space allocated by `-XX:mmap` at initialization and committed as needed. The `-XX:MaxMetaspaceSize` applies to the sum of the committed compressed class space and the space for the other class metadata.

Class metadata is deallocated when the corresponding Java class is unloaded. Java classes are unloaded as a result of garbage collection, and garbage collections may be induced to unload classes and deallocate class metadata. When the space committed for class metadata reaches a certain level (a high-water mark), a garbage collection is induced. After the garbage collection, the high-water mark may be raised or lowered depending on the amount of space freed from class metadata. The high-water mark would be raised so as not to induce another garbage collection too soon. The high-water mark is initially set to the value of the command-line option `-XX:MetaspaceSize`. It is raised or lowered based on the options `-XX:MaxMetaspaceFreeRatio` and `-XX:MinMetaspaceFreeRatio`. If the committed space available for class metadata as a percentage of the total committed space for class metadata is greater than `-XX:MaxMetaspaceFreeRatio`, then the high-water mark will be lowered. If it's less than `-XX:MinMetaspaceFreeRatio`, then the high-water mark will be raised.

Specify a higher value for the option `-XX:MetaspaceSize` to avoid early garbage collections induced for class metadata. The amount of class metadata allocated for an application is application-dependent and general guidelines do not exist for the selection of `-XX:MetaspaceSize`. The default size of `-XX:MetaspaceSize` is platform-dependent and ranges from 12 MB to about 20 MB.

Information about the space used for metadata is included in a printout of the heap. The following is typical output:.

```
[0,296s][info][gc,heap,exit] Heap
[0,296s][info][gc,heap,exit] garbage-first heap total 514048K, used 0K
[0x00000005ca600000, 0x00000005ca8007d8, 0x00000007c0000000)
[0,296s][info][gc,heap,exit] region size 2048K, 1 young (2048K), 0
survivors (0K)
[0,296s][info][gc,heap,exit] Metaspace used 2575K, capacity 4480K,
committed 4480K, reserved 1056768K
[0,296s][info][gc,heap,exit] class space used 238K, capacity 384K,
committed 384K, reserved 1048576K
```

In the line beginning with `Metaspace`, the `used` value is the amount of space used for loaded classes. The `capacity` value is the space available for metadata in currently allocated chunks. The `committed` value is the amount of space available for chunks. The `reserved` value is the amount of space reserved (but not necessarily committed)

for metadata. The line beginning with `class space` contains the corresponding values for the metadata for compressed class pointers.