O'REILLY®

Second Edition

Java Performance

In-Depth Advice for Tuning and Programming Java 8, 11, and Beyond

Early Release

RAW & UNEDITED

Scott Oaks

Java Performance: The Definitive Guide

SECOND EDITION

Scott Oaks

O'REILLY®

Beijing • Boston • Farnham • Sebastopol • Tokyo

Java Performance: The Definitive Guide

By Scott Oaks

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Rachel Roumeliotis

Developmental Editors: Suzanne McQuade and Amelia Blevins

Production Editor: Nan Barber

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

Revision History for the Early Release

- 2019-07-26: First Early Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492056041 for release details.

978-1-492-05604-1

Chapter 1. An Introduction to Garbage Collection

This chapter covers the basics of garbage collection within the JVM. Short of rewriting code, tuning the garbage collector is the most important thing that can be done to improve the performance of a Java application.

Since the performance of Java applications depends heavily on garbage collection technology, it is not surprising that there are quite a few collectors available. The OpenJDK has three collectors suitable for production; another that is deprecated in JDK 9 but still quite popular in JDK 8; and some experimental collectors that will (hopefully) be production-ready in future releases. Other Java implementations like J9 or the Azul JVM have their own proprietary collectors. The performance characteristics of all these collectors are quite different, so each of the OpenJDK collectors will be covered in depth in the next chapter. However, they share basic concepts, so this chapter provides a basic overview of how the collectors operate.

Garbage Collection Overview

One of the most attractive features of programming in Java is that developers needn't explicitly manage the lifecycle of objects: objects are created when needed, and when the object is no longer in use, the JVM automatically frees the object. If, like me, you spend a large amount of time optimizing the memory use of Java programs, this whole scheme might seem like a weakness instead of a feature (and the amount of time I'll spend covering GC might seem to lend credence to that position). Certainly it can be considered a mixed blessing, but I still recall the difficulties of tracking down null pointers and dangling pointers in other languages: I'd strongly argue that tuning garbage collectors is far easier (and less time-consuming) than tracking down pointer bugs.

At a basic level, GC consists of finding objects that are no longer in use, and freeing the memory associated with those objects. The JVM starts by finding objects that are no longer in use (garbage objects). This is sometimes described as finding objects that no longer have any references to them (implying that references are tracked via a count). That sort of reference counting is insufficient, though: given a linked list of objects, each object in the list (except the head) will be pointed to by another object in the list—but if nothing refers to the head of the list, the entire list is not in use and can be freed. And if the list is circular (e.g., the tail of the list points to the head), then every object in the list has a reference to it—even though no object in the list can actually be used, since no objects reference the list itself.

So references cannot be tracked dynamically via a count; instead, the JVM must periodically search the heap for unused objects. When it finds unused objects, the JVM can free the memory occupied by those objects and use it to allocate additional objects. However, it is usually insufficient simply to keep track of that free memory and use it for future allocations; at some point, memory must be coalesced to prevent memory fragmentation.

Consider the case of a program that allocates an array of 1,000 bytes, then one of 24 bytes, and repeats that process in a loop. When that process fills up the heap, it will appear like the top row in Figure 1-1: the heap is full, and the allocations of the array sizes are interleaved.

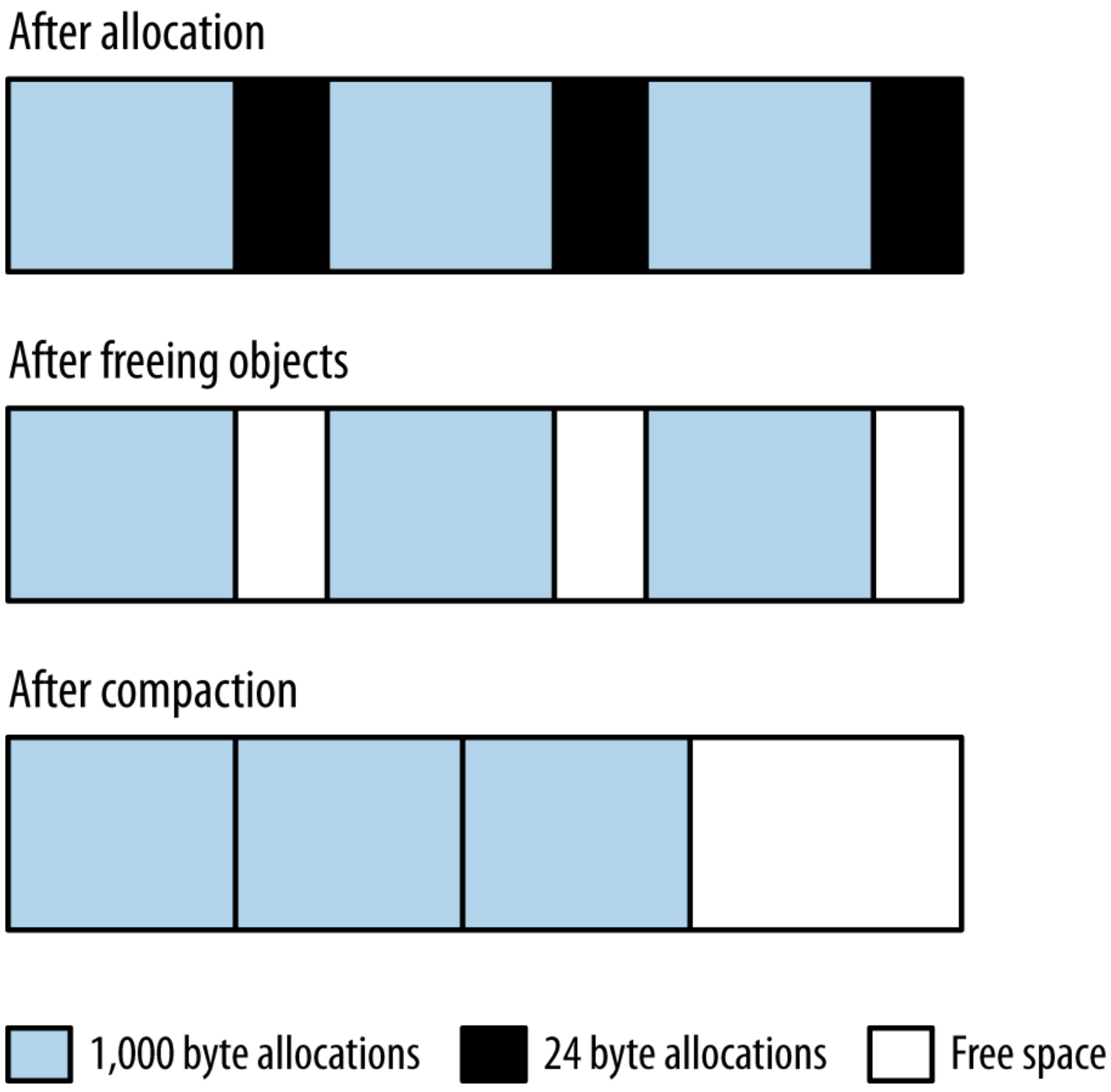


Figure 1-1. Idealized GC heap during collection

When the heap is full, the JVM will free the unused arrays. Say that all the 24-byte arrays are no longer in use, and the 1,000-byte arrays are still all in use: that yields the second row in Figure 1-1. The heap has free areas within it, but it can't actually allocate anything larger than 24 bytes—unless the JVM moves all the 1,000-byte arrays so that they are contiguous, leaving all the free memory in a region where it can be allocated as needed (the third row in Figure 1-1).

The implementations are a little more detailed, but the performance of GC is dominated by these basic operations: finding unused objects, making their memory available, and compacting the heap. Different collectors take different approaches to these operations, which is why they yield different performance characteristics.

It is simpler to perform these operations if no application threads are running while the garbage collector is running. Java programs are typically heavily multithreaded, and the garbage collector itself often runs multiple threads. This discussion considers two logical groups of threads: those performing application logic (often called mutator threads, since they are mutating objects as part of the application logic), and those performing GC. When GC threads track object references or move objects around in memory, they must make sure that application threads are not using those objects. This is particularly true when GC moves objects around: the memory location of the object changes during that operation, and hence no application threads can be accessing the object.

The pauses when all application threads are stopped are called *stop-the-world pauses*. These pauses generally have the greatest impact on the performance of an application, and minimizing those pauses is the key consideration when tuning GC.

Generational Garbage Collectors

Though the details differ somewhat, most garbage collectors work by splitting the heap into different generations. These are called the old (or tenured) generation, and the young generation. The young generation is further divided into sections known as eden and the survivor spaces (though sometimes, eden is incorrectly used to refer to the entire young generation).

The rationale for having separate generations is that many objects are used for a very short period of time. Take, for example, the loop in the stock price calculation where it sums the square of the difference of price from the average price (part of the calculation of standard deviation):

```
sum = new BigDecimal(0);
for (StockPrice sp : prices.values()) {
    BigDecimal diff = sp.getClosingPrice().subtract(averagePrice);
    diff = diff.multiply(diff);
```

```
        sum = sum.add(diff);
    }
```

Like many Java classes, the `BigDecimal` class is immutable: the object represents a particular number and cannot be changed. When arithmetic is performed on the object, a new object is created (and often, the previous object with the previous value is then discarded). When this simple loop is executed for a year's worth of stock prices (roughly 250 iterations), 750 `BigDecimal` objects are created to store the intermediate values just in this loop. Those objects are discarded on the next iteration of the loop. Within the `add()` and other methods, the JDK library code creates even more intermediate `BigDecimal` (and other) objects. In the end, a lot of objects are created and discarded very quickly in this very small amount of code.

This kind of operation is quite common in Java and so the garbage collector is designed to take advantage of the fact that many (and sometimes most) objects are only used temporarily. This is where the generational design comes in. Objects are first allocated in the young generation, which is some subset of the entire heap. When the young generation fills up, the garbage collector will stop all the application threads and empty out the young generation. Objects that are no longer in use are discarded, and objects that are still in use are moved elsewhere. This operation is called a minor GC or a young GC.

There are two performance advantages to this design. First, because the young generation is only a portion of the entire heap, processing it is faster than processing the entire heap. This means that the application threads are stopped for a much shorter period of time than if the entire heap were processed at once. You probably see a trade-off there, since it also means that the application threads are stopped more frequently than they would be if the JVM waited to perform GC until the entire heap were full; that trade-off will be explored in more detail later in this chapter. For now, though, it is almost always a big advantage to have the shorter pauses even though they will be more frequent.

The second advantage arises from the way objects are allocated in the young generation. Objects are allocated in eden (which comprises the vast majority of the young generation). When the young generation is cleared during a collection, all objects in eden are either moved or discarded: objects that are not in use can be discarded, and objects in use are moved either to one of the survivor spaces or to the old generation. Since all surviving objects are moved, the young generation is automatically compacted when it is collected.

Most common GC algorithms have stop-the-world pauses during collection of the young generation.

As objects are moved to the old generation, eventually it too will fill up, and the JVM will need to find any objects within the old generation that are no longer in use and discard them. This is where GC algorithms have their biggest differences. The simpler algorithms stop all application threads, find the unused objects, free their memory, and then compact the heap. This process is called a full GC, and it generally causes a relatively long pause for the application threads.

On the other hand, it is possible—though more computationally complex—to find unused objects while application threads are running. Because the phase where they scan for unused objects can occur without stopping application threads, these algorithms are called concurrent collectors. They are also called low-pause (and sometimes—incorrectly—pauseless) collectors, since they minimize the need to stop all the application threads. Concurrent collectors also take different approaches to compacting the old generation.

When using a concurrent collector, an application will typically experience fewer (and much shorter) pauses. The trade-off is that the application will use more CPU overall. Concurrent collectors can also be more difficult to tune in order to get their best performance (though in JDK 11, tuning concurrent collectors like the G1 GC is much easier than in previous releases, which reflects the engineering progress that has been made since the concurrent collectors were first introduced).

As you consider which garbage collector is appropriate for your situation, think about the overall performance goals that must be met. There are trade-offs in every situation. In an application (such as a Java EE server) measuring the response time of individual requests, consider these points:

- The individual requests will be impacted by pause times—and more importantly by long pause times for full GCs. If minimizing the effect of pauses on response times is the goal, a concurrent collector will be more appropriate.

- If the average response time is more important than the outliers (i.e., the 90th% response time), a non-concurrent collector may yield better results.

- The benefit of avoiding long pause times with a concurrent collector comes at the expense of extra CPU usage. If your machine lacks the spare CPU cycles needed by a concurrent collector, a non-concurrent collector may be the better choice.

Similarly, the choice of garbage collector in a batch application is guided by the following trade-off:

- If enough CPU is available, using the concurrent collector to avoid full GC pauses will allow the job to finish faster.

- If CPU is limited, then the extra CPU consumption of the concurrent collector will cause the batch job to take more time.

QUICK SUMMARY

1. GC algorithms generally divide the heap into old and young generations.
2. GC algorithms generally employ a stop-the-world approach to clearing objects from the young generation, which is usually a very quick operation.
3. Minimizing the effect of performing GC in the old generation is a trade off between pause times and CPU usage.

GC Algorithms

OpenJDK 12 provides a variety of GC algorithms with varying degrees of support in earlier releases. Table 1-1 lists various algorithms and their status in OpenJDK and Oracle Java releases.

Table 1-1. Support level of various GC algorithms (S: Fully Supported D: Deprecated E: Experimental E2: Experimental; in OpenJDK builds but not Oracle builds)

GC algorithm	Support in JDK 8	Support in JDK 11	Support in JDK 12
Serial GC	S	S	S
Throughput (Parallel) GC	S	S	S
G1 GC	S	S	S
Concurrent Mark-Sweep (CMS)	S	D	D
ZGC	N	E	E
Epsilon GC	N	E	E
Shenandoah	E2	E2	E2

Here is a brief description of each algorithm; Chapter 2 will provide more details on tuning them individually.

THE SERIAL GARBAGE COLLECTOR

The serial garbage collector is the simplest of the collectors. This is the default collector if the application is running on a client-class machine (32-bit JVMs on Windows) or on a single processor machine.

The serial collector uses a single thread to process the heap. It will stop all application threads as the heap is processed (for either a minor or full GC). During a full GC, it will fully compact the old generation.

The serial collector is enabled by using the `-XX:+UseSerialGC` flag (though usually it is the default in those cases where it might be used). Note that unlike with most JVM flags, the serial collector is not disabled by changing the plus sign to a minus sign (i.e., by specifying `-XX:-UseSerialGC`). On systems where the serial collector is the default, it is disabled by specifying a different GC algorithm.

THE THROUGHPUT COLLECTOR

In JDK 8, this is the default collector for any 64-bit machine with two or more CPUs.

The throughput collector uses multiple threads to collect the young generation, which makes minor GCs much faster than when the serial collector is used. The throughput collector uses multiple threads to process the old generation as well. Because it uses multiple threads, the throughput collector is often called the parallel collector.

The throughput collector stops all application threads during both minor and full GCs, and it fully compacts the old generation during a full GC. Since it is the default in most situations where it would be used, it needn't be explicitly enabled. To enable it where necessary, use the flag `-XX:+UseParallelGC`.

Note that old versions of the JVM enabled parallel collection in the young and old generations separately, so you might see references to the flag `-XX:+UseParallelOldGC`. This flag is obsolete (though it still functions, and you could disable this flag to collect only the young generation in parallel if for some reason you really wanted to).

THE G1 GC COLLECTOR

The G1 GC (or Garbage First Garbage Collector) uses a concurrent collection strategy to collect the heap with minimal pauses. It is the default collector in JDK 11 and later for 64-bit JVMs on machines with two or more CPUs.

G1 GC divides the heap into a number of regions, but it is still considers the heap to have two generations. Some number of those regions comprise the young generation, and the young generation is still collected by stopping all application threads and moving all objects that are alive into the old generation or the survivor spaces. This occurs using multiple threads.

In G1 GC, the old generation is processed by background threads that don't need to stop the application threads to perform most of their work. Because the old generation is divided into regions, G1 GC can clean up objects from the old generation by copying from one region into another, which means that it (at least partially) compacts the heap during normal processing. This helps to prevent G1 GC heaps from becoming fragmented, although that is still possible.

The trade-off for avoiding the full GC cycles is CPU time: the (multiple) background threads G1 GC uses to process the old generation must have CPU cycles available at the same time the application threads are running.

G1 GC is enabled by specifying the flag `-XX:+UseG1GC`. Although it is the default in JDK 11, it is fully functional in JDK 8 as well — particular in later builds of JDK 8, which contains many important bug fixes and performance enhancements that have been back-ported from later releases.

THE CMS COLLECTOR

The CMS collector was the first concurrent collector. Like other algorithms, CMS stops all application threads during a minor GC, which it performs with multiple threads.

CMS is officially deprecated in JDK 11 and beyond, and its use in JDK 8 is discouraged. From a partical standpoint, the major flaw in CMS is that it has no way to compact the heap during its background processing. If the heap becomes fragmented (which is quite likely to happen at some point), CMS must stop all application threads and compact the heap, which defeats the purpose of a concurrent collector. Between that and the advent of G1 GC, CMS is no longer recommended.

CMS is enabled by specifying the flag `-XX:+UseConcMarkSweepGC`, which is `false` by default. Historically, CMS used to require setting the `-XX:+UseParNewGC` flag as well (else the young generation would be collected by a single thread), though that is obsolete.

EXPERIMENTAL COLLECTORS

Garbage collection continues to be fertile ground for JVM engineers, and the latest versions of Java come with the three experimental algorithms mentioned earlier. We’ll have more to say about those in the next chapter; for now, let’s continue with a look at when to choose among the three collectors supported in production environments.

CAUSING AND DISABLING EXPLICIT GARBAGE COLLECTION

GC is typically caused when the JVM decides GC is necessary: a minor GC will be triggered when the new generation is full, a full GC will be triggered when the old generation is full, or a concurrent GC (if applicable) will be triggered when the heap starts to fill up.

Java provides a mechanism for applications to force a GC to occur: the `System.gc()` method. Calling that method is almost always a bad idea. This call always triggers a full GC (even if the JVM is running with G1 GC or CMS), so application threads will be stopped for a relatively long period of time. And calling this method will not make the application any more efficient; it will cause a GC to occur sooner than might have happened otherwise, but that is really just shifting the performance impact.

There are exceptions to every rule, particularly when doing performance monitoring or benchmarking. For small benchmarks that run a bunch of code to properly warm up the JVM, forcing a GC before the measurement cycle may make sense. Similarly when doing heap analysis, it is usually a good idea to force a full GC before taking the heap dump. Most techniques to obtain a heap dump will perform a full GC anyway, but there are also other ways you can force a full GC: you can execute **`jcmd <process id> GC.run`**, or you can connect to the JVM using `jconsole` and click the Perform GC button in the Memory panel.

Another exception is RMI, which calls `System.gc()` every hour as part of its distributed garbage collector. That timing can be changed by setting a different value for these two system properties: `-Dsun.rmi.dgc.server.gcInterval=`*`N`* and `-Dsun.rmi.dgc.client.gcInterval=`*`N`*. The values for `N` are in milliseconds, and the default value is 3600000 (one hour).

If you end up running third-party code that incorrectly calls the `System.gc()` method, those GCs can be prevented entirely by including `-XX:+DisableExplicitGC` in the JVM arguments; by default that flag is `false`. Java EE servers often include this argument to prevent the RMI gc calls from interfering with their operations.

QUICK SUMMARY

1. The supported GC algorithms take different approaches toward minimizing the effect of GC on an application.

2. The serial collector makes sense (and is the default) when only one CPU is available and extra GC threads would interfere with the application.

3. The throughput collector is the default in JDK8; it maximizes the total throughput of an application but may subject individual operations to long pauses.

4. G1 GC is the default in JDK11 and beyond; it concurrently collects the old generation while application threads are running, potentially avoiding full GCs. Its design makes it less likely to experience full GCs than CMS.

5. The CMS collector can concurrently collect the old generation while application threads are running. If enough CPU is available for its background processing, this can avoid full GC cycles for the application. It is deprecated in favor of G1 GC.

Choosing a GC Algorithm

The choice of a GC algorithm depends in part on the hardware available, in part on what the application looks like, and in part on the performance goals for the application. But while this choice used to be fraught with significance, things have changed: in JDK 11—and the latest versions of JDK 8—you almost certainly want to use G1 GC.

As a rule of thumb, G1 GC is the better choice, but there are exceptions to every rule. In the case of garbage collection, these exceptions involve the amount of CPU cyles the application needs relative to the available hardware, and the amount of processing the background G1 GC threads need to perform. When G1 GC is not the better choice, then the decision between the throughput and serial collectors is based on the number of CPUs on the machine.

WHEN TO USE (AND NOT USE) THE SERIAL COLLECTOR

On a machine with a single CPU, the JVM defaults to using the serial collector. That’s usually a good choice, though there are times when G1 GC will give better results. This example is also a good starting point for understanding the general trade-offs involved in choosing a GC algorithm.

The trade-off between G1 GC and other collectors involves having available CPU cyles for G1 GC background threads, so let's start with a CPU-intensive batch job. In a batch job, the CPU will be 100% busy for a long time, and then the serial collector has a marked advantage.

Table 1-2 lists the time required for a single thread to compute stock histories for 100,000 stocks over a period of 3 years.

Table 1-2. Processing time on a single CPU for different GC algorithms

GC algorithm	Elapsed Time	Time Paused for GC
Serial	434 seconds	79 seconds
Throughput	503 seconds	144 seconds
G1 GC	501 seconds	97 seconds

The advantage of the single-threaded garbage collection is most readily apparent when we compare the serial collector to the throughput collector. The time spent doing the actual calculation is the elapsed time minus the time spent paused for GC. In the serial and throughput collectors, that time is essentially the same (roughly 355 seconds), but the serial collector wins because it spends much less time paused for garbage collection. In particular, the serial collector takes on average 505 ms for a full GC, where the throughput collector requires 1392 ms. The throughput collector has a fair amount of overhead in its algorithm—that overhead is worthwhile when there are two or more threads processing the heap, but it just gets in the way when there is only a single available thread.

Now compare the serial collector to G1 GC. If we eliminate the pause time when running with G1 GC, the application takes 404 seconds for its calculation—but we know from the other examples that it should take only 355 seconds for that. What accounts for the other 49 seconds?

The calculation thread can utilize all available CPU cycles. At the same time, there are background G1 GC threads that need CPU cycles for their work. Since there isn't enough CPU to satisfy both, they end up sharing the CPU: the calculation thread will run some of the time, and a background G1 GC thread will run some of the time. The net effect is the calculation thread cannot run for 49 seconds because a "background" G1 GC thread is occupying the CPU.

That’s what is meant when we say that when you choose G1 GC, there must be sufficient CPU cycles for its backgroun threads to run. With a long-running application thread taking the only available CPU, G1 GC is not a good choice. But what about something different, like a microservice running simple REST requests on the constrained hardware. Table 1-3 shows the response time for a webserver that is handling roughly 11 requests per second on its single CPU, which takes roughly 50% of the available CPU cycles.

Table 1-3. Response times for a single CPU with different GC algorithms

GC algorithm	Average Response Time	90th% Response Time	99th% Response Time	CPU Utilization
Serial	0.10 seconds	0.18 seconds	0.69 seconds	53%
Throughput	0.16 seconds	0.18 seconds	1.40 seconds	49%
G1 GC	0.13 seconds	0.28 seconds	0.40 seconds	48%

The default (serial) algorithm still has the best average time by 30%. Again, that’s because the collections of the young generation by the serial collector are generally faster than those of the other algorithms, so an average request is delayed less by the serial collector.

Some unlucky requests will get interrupted by a full GC of the serial collector. In this experiment, the average time for a full GC by the serial collector took 592 milliseconds and some took as long as 730 milliseconds. The result is that 1% of the requests took almost 700 milliseconds.

That’s still better than the throughput collector can do. The full GCs of the throughput collector averaged 1192 milliseconds with a 1510 millisecond maximum. Hence the 99th% response time of the throughput collector is twice that of the serial collector. And the average time is skewed by those outliers as well.

G1 GC sits somewhere in the middle. In terms of average response time, it is worse than the serial collector, which is due to the fact that the simpler serial collector algorithm is faster. In this case, that applies primarily to the minor GCs, which took on average 86 milliseconds for the serial collector but required 141 milliseconds for G1 GC. So an average request will get delayed longer in the G1 GC case.

Still, G1 GC has a 99th% response time which is almost half that of the serial collector. In this example, G1 GC was able to avoid full GCs, so it had none of the 500+ millisecond delays of the serial collector.

There’s a choice of what to optimize for here: if average response time is the most important goal, then the (default) serial collector is the better choice. If you want to optimize for the 99th% response time, then G1 GC wins. It’s a judgement call, but to me, the 30 ms difference in the average time is not as important as the 300 ms difference in the 99th% time—so here’s a case where G1 GC makes sense over the platform’s default collector.

This example is quite GC intensive, and in particular the non-concurrent collectors each have to perform a significant amount of full GC operations. If we tweak the test such that all objects can be collected without requiring a full GC, then the serial algorithm can match G1 GC as Table 1-4 shows.

Table 1-4. Response times for a single CPU with different GC algorithms (no full GCs)

GC algorithm	Average Response Time	90th% Response Time	99th% Response Time	CPU Utilization
Serial	0.05 seconds	0.08 seconds	0.11 seconds	53%
Throughput	0.08 seconds	0.09 seconds	0.13 seconds	49%
G1 GC	0.05 seconds	0.08 seconds	0.11 seconds	52%

Because there are no full GCs, the advantage of the serial collector to G1 GC is eliminated. On the other hand, it’s pretty unlikely that there be no full GCs, and that’s the case where the serial collector will do best. Given sufficient CPU cycles, G1 will generally be better even where the serial collector is the default.

Single Hyper-Threaded CPU Hardware

What about a single CPU where the CPU is hyper-threaded (and hence appears to the JVM as a two CPU machine)? In that case, the JVM will not use the serial collector by default—it thinks there are two CPUs, so it will default to the throughput collector in JDK8 and G1 GC in JDK11. But it turns out that the serial collector is often advantageous on this hardware as well. Table 1-5 shows what happens when we run the previous batch experiment on a single hyper-threaded CPU.

Table 1-5. Processing time on a single hyper-threaded CPU for different GC algorithms

GC algorithm	Elapsed Time	Time Paused for GC
Serial	432 seconds	82 seconds
Throughput	478 seconds	117 seconds
G1 GC	476 seconds	72 seconds

The serial collector won't run mutiple threads, so its times are essentially unchanged from our previous test. The other algorithms have improved, but not by as much as we might hope—the throughput collector will run two threads, but instead of cutting the pause time in half, the pause time has been reduced by about 20%. (That's typical; Intel documents a hyper-thread adds "up to 30%" in performance.) Similarly, G1 GC still cannot get enough CPU cycles for its background threads

So at least in this case—a long-running batch job with frequent garbage collection—the default choice by the JVM will be incorrect, and the application will be better off using the serial collector despite the presence of "two" cpus. If there were two actual CPUs (i.e., two cores), then things would be different. The throughput collector would take only 72 seconds for its operations; less than the time required by the serial collector. At that point, the usefulness of the serial collector wanes, so we'll drop it from future examples.

One other point about the serial collector: an application with a very small heap (say, 100MB) may still perform better with the serial collector regardless of the number of cores that are available. That could apply to a simple administrative command, though Chapter 2 will have more to say about that topic.

WHEN TO USE THE THROUGHPUT COLLECTOR

In terms of garbage collection, running multiple threads on hardware with multiple cores isn't significantly different than the single-threaded experiements we just looked at. For example, Table 1-6 shows how our sample application works when running either two or four application threads on a machine with four cores (where the cores are not hyper-threaded).

Table 1-6. Batch processing time with different GC algorithms

Application Threads	G1 GC	Throughput
Two	410 seconds (60.8%)	446 seconds (59.7%)
Four	513 seconds (99.5%)	536 seconds (99.5%)

The times in this table are the number of seconds required to run the test, and the CPU utilization of the machine is shown in parentheses. When there are two application threads, G1 GC is significantly faster than the throughput collector. The main reason for that is the throughput collector spent 35 seconds paused for full GCs. G1 GC was able to avoid those collections, at the (relatively slight) increase in CPU time.

Even when there are four application threads, G1 still wins in this example. Here, the throughput collector paused the application threads for a total of 176 seconds. G1 GC paused the application threads for only 88 seconds. The application threads are keeping the CPUs 100% busy, and so G1 GC did have interrupt the application threads for a total of 65 seconds—but that still meant G1 GC was 23 seconds faster.

When the elapsed time of an application is key, the throughput collector will be advantageous when it spends less time pausing the application threads than G1 GC does. That happens when one or more of these things occur: * There are no (or few) full GCs. Full GC pauses can easily dominate the pause times of an application, but if they don't occur in the first place, then the throughput collector is no longer at a disadvantage. * The old generation is generally full, causing the background G1 GC threads to work more. * The G1 GC threads are starved for CPU.

In the next chapter, we'll examine the details of how the various algorithms work and the reasons behind these points will be clearer (as well as ways to tune the collectors around them). Fow now, we'll look at a few examples that prove the point.

Table 1-7 illustrates these points. This test is the same code we used before for batch jobs with long calculations, though it has a few modifications: there are now multiple application threads doing calculations (two in this case), the old generation is seeded with objects to keep it 65% full, and almost all objects can be collected directly from the young generation. This test is run on a system with 4 CPUs (not hyper-threaded) so that there is sufficient CPU for the G1 GC background threads to run.

Table 1-7. Batch processing with long-lived objects

Application Threads	G1 GC	Throughput
Two	212 seconds (67%)	193 seconds (51%)

Because so few objects are promoted to the old generation, the throughput collector paused the application threads for only 15 seconds, and only 1.5 seconds of that was to collect the old generation.

Although the old generation doesn't get many new objects promoted into it, the test seeds the old generation such that the G1 GC threads will scan it for garbage. This makes more work for the background GC threads, and it causes G1 GC to perform more work collecting the young generation in an attempt to compensate for the fuller old generation. The end result is that G1 GC paused the application for 30 seconds during the two thread test—more than the throughput collector did.

Similarly, when there isn't sufficient CPU for the G1 GC background thread to run, the throughput collector will perform better. This is really no different than the case we saw with a single CPU: the competition for CPU cycles between the G1 GC background threads and the application threads meant that the application threads were effectively paused even when GC wasn't happening. The same thing can happen when there are multiple CPUs and multiple application threads, as Table 1-8 shows.

Table 1-8. Batch processing with busy CPUs

Application Threads	G1 GC	Throughput
Four	225 seconds (99%)	212 seconds (99%)

If we’re more interested in interactive processing and response times, then the throughput collector has a harder time beating G1 GC. If your server is short of CPU cycles such that the G1 GC and application threads compete for CPU, then G1 GC will yield worse response times (similar to the cases we’ve already seen). If the server is tuned such that there are no full GCs, then G1 GC and the throughput collector will generally turn out similar results. But the more full GCs that the server has, the better G1 GC average, 90th% and 99th% response times will be.

AVERAGE CPU USAGE AND GC

Looking at only the average CPU during a test misses the interesting picture of what happens during GC cycles. The throughput collector will (by default) consume 100% of the CPU available on the machine while it runs, so a more accurate representation of the CPU usage during the test with two application threads is shown in Figure 1-2.

Most of the time, only the application threads are running, consuming 50% of the total CPU. When GC kicks in, 100% of the CPU is consumed. Hence, the actual CPU usage resembles the sawtooth pattern in the graph, even though the average during the test is reported as the value of the straight dashed line.

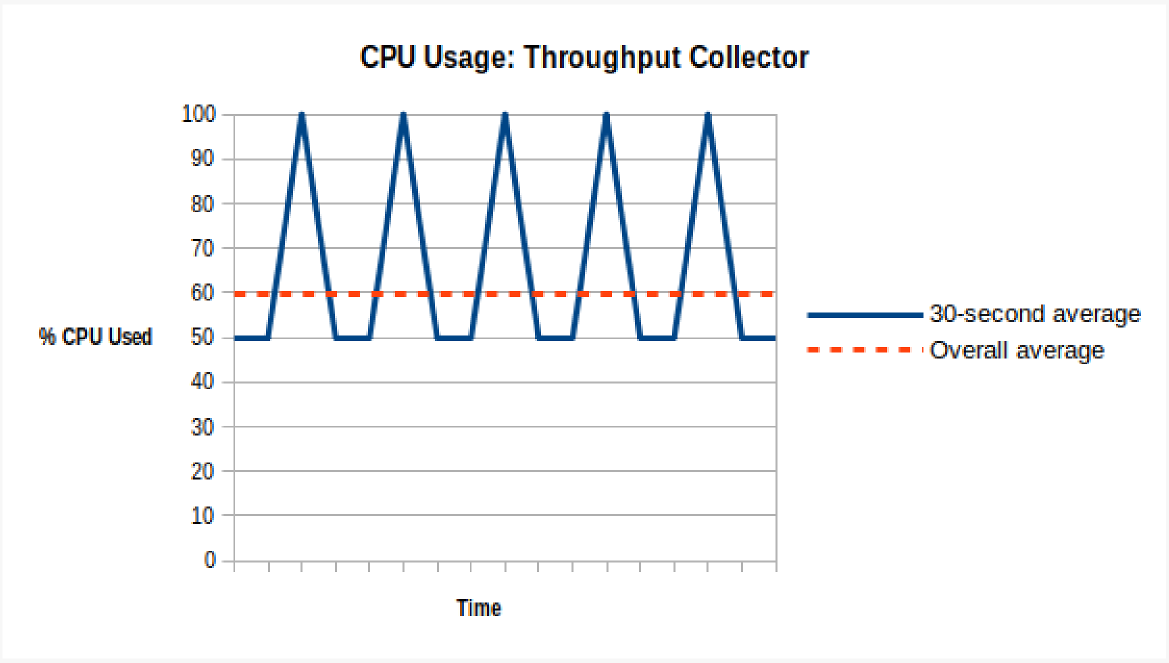


Figure 1-2. Actual versus average CPU usage (throughput)

The effect is different in a concurrent collector, when there are background thread(s) running concurrently with the application threads. In that case, a graph of the CPU looks like Figure 1-3.

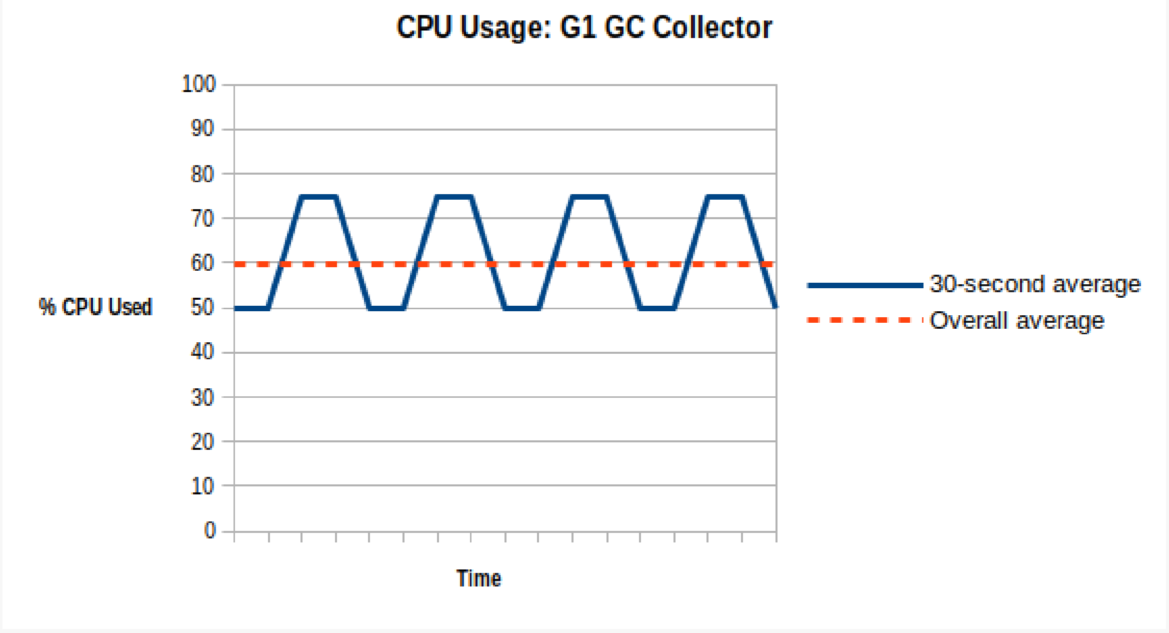


Figure 1-3. Actual versus average CPU usage (CMS)

The application thread starts by using 50% of the total CPU. Eventually it has created enough garbage for a G1 GC background thread to kick in; that thread also consumes an entire CPU, bringing the total up to 75%. When that thread finishes, CPU usage drops to 50%, and so on. Note that there are no 100% peak-CPU periods, which is a little bit of a simplification: there will be very short spikes to 100% CPU usage during the G1 GC young generation collections, but those are short enough that we can ignore them for this discussion. (Those really short spikes occur for the throughput collector as well.)

There can be multiple background threads in a concurrent collector, but the effect is similar: when those background threads run, they will consume CPU and drive up the long-term CPU average.

This can be quite important when you have a monitoring system triggered by CPU usage rules: you want to make sure CPU alerts are not triggered by the 100% CPU usage spikes in a full GC, or the much longer (but lower) spikes from background concurrent processing threads. These spikes are normal occurrences for Java programs.

QUICK SUMMARY

1. G1 GC is currently the better algorithm to chose for a majority of applications.

2. The serial collector makes sense when running CPU-bound applications on a machine with a single CPU, even if that single CPU is hyper-threaded. G1 GC will still be better on such hardware for jobs that are not CPU-bound.

3. The throughput collector makes sense on multi-CPU machines that run jobs that are CPU bound. Even for jobs that are not CPU bound, the throughput collector can be the better choice if it does relatively few full GCs, or if the old generation is generally full.

Basic GC Tuning

Although GC algorithms differ in the way they process the heap, they share basic configuration parameters. In many cases, these basic configurations are all that is needed to run an application.

Sizing the Heap

The first basic tuning for GC is the size of the application's heap. There are advanced tunings that affect the size of the heap's generations; as a first step, this section will discuss setting the overall heap size.

Like most performance issues, choosing a heap size is a matter of balance. If the heap is too small, the program will spend too much time performing GC and not enough time performing application logic. But simply specifying a very large heap isn't necessarily the answer either. The time spent in GC pauses is dependent on the size of the heap, so as the size of the heap increases, the duration of those pauses also increases. The pauses will occur less frequently, but their duration will make the overall performance lag.

A second danger arises when very large heaps are used. Computer operating systems use virtual memory to manage the physical memory of the machine. A machine may have 8 GB of physical RAM, but the OS will make it appear as if there is much more memory available. The amount of virtual memory is subject to the OS configuration, but say the OS makes it look like there is 16 GB of memory. The OS manages that by a process called swapping. (Or paging, though there is also a technical difference between those two terms that isn't important for this discussion.)

You can load programs that use up to 16 GB of memory, and the OS will copy inactive portions of those programs to disk. When those memory areas are needed, the OS will copy them from disk to RAM (usually it will first need to copy something from RAM to disk to make room).

This process works well for a system running lots of different applications, since most of the applications are not active at the same time. It does not work so well for Java applications. If a Java program with a 12 GB heap is run on this system, the OS can handle it by keeping 8 GB of the heap in RAM and 4 GB on disk (that simplifies the actual situation a little, since other programs will use part of RAM). The JVM won't know about this; the swapping is transparently handled by the OS. Hence, the JVM will happily fill up all 12 GB of heap it has been told to use. This causes a severe performance penalty as the OS swaps data from disk to RAM (which is an expensive operation to begin with).

Worse, the one time this swapping is guaranteed to occur is during a full GC, when the JVM must access the entire heap. If the system is swapping during a full GC, pauses will be an order of magnitude longer than they would otherwise be. Similarly, when you use G1 GC and the background thread sweeps through the heap, it will likely fall behind due to the long waits for data to be copied from disk to main memory—resulting in an expensive concurrent mode failure.

Hence, the first rule in sizing a heap is never to specify a heap that is larger than the amount of physical memory on the machine—and if there are multiple JVMs running, that applies to the sum of all their heaps. You also need to leave some room for the JVM itself, as well as some memory for other applications: typically, at least 1 GB of space for common OS profiles.

The size of the heap is controlled by two values: an initial value (specified with `-Xms`*N*) and a maximum value (`-Xmx`*N*). The defaults for these vary depending on the operating system, the amount of system RAM, and the JVM in use. The defaults can be affected by other flags on the command line as well; heap sizing is one of the JVM's core ergonomic tunings.

The goal of the JVM is to find a "reasonable" default initial value for the heap based on the system resources available to it, and to grow the heap up to a "reasonable" maximum if (and only if) the application needs more memory (based on how much time it spends performing GC). Absent some of the advanced tuning flags and details discussed later in this and the next chapters, the default values for the initial and maximum sizes are given in Table 1-9. (The JVM will round these values down slightly for alignment purposes; the GC logs that print the sizes will show that the values are not exactly equal to the numbers in this table.)

Table 1-9. Default heap sizes

Operating system and JVM	Initial heap (Xms)	Maximum heap (Xmx)
Linux	Min (512 MB, 1/64 of physical memory)	Min (32 GB, 1/4 of physical memory)
MacOS	64 MB	Min (1 GB, 1/4 of physical memory)
Windows 32-bit client JVMs	16 MB	256 MB
Windows 64-bit server JVMs	64 MB	Min (1 GB, 1/4 of physical memory)

On a machine with less than 192 MB of physical memory, the maximum heap size will be half of the physical memory (96 MB or less).

Having an initial and maximum size for the heap allows the JVM to tune its behavior depending on the workload. If the JVM sees that it is doing too much GC with the initial heap size, it will continually increase the heap until the JVM is doing the “correct” amount of GC, or until the heap hits its maximum size.

For many applications, that means a heap size doesn’t need to be set at all. Instead, you specify the performance goals for the GC algorithm: the pause times you are willing to tolerate, the percentage of time you want to spend in GC, and so on. The details of that will depend on the GC algorithm used and are discussed in the next chapter (though even then, the defaults are chosen such that for a wide range of applications, those values needn’t be tuned either).

That approach usually works fine if an application does not need a larger heap than the default maximum for the platform it is running on. However, if the application is spending too much time in GC, then the heap size will need to be increased by setting the `-Xmx` flag. There is no hard-and-fast rule on what size to choose (other than not specifying a size larger than the machine can support). A good rule of thumb is to size the heap so that it is 30% occupied after a full GC. To calculate this, run the application until it has reached a steady-state configuration: a point at which it has loaded anything it caches, has created a maximum number of client connections, and so on. Then connect to the application with `jconsole`, force a full GC, and observe how much memory is used when the full GC completes. (Alternately, for throughput GC, you can consult the GC log if it is available.)

Be aware that automatic sizing of the heap occurs even if you explicitly set the maximum size: the heap will start at its default initial size, and the JVM will grow the heap in order to meet the performance goals of the GC algorithm. There isn't necessarily a memory penalty for specifying a larger heap than is needed: it will only grow enough to meet the GC performance goals.

On the other hand, if you know exactly what size heap the application needs, then you may as well set both the initial and maximum values of the heap to that value (e.g., `-Xms4096m -Xmx4096m`. That makes GC slightly more efficient, because it never needs to figure out whether the heap should be resized.

QUICK SUMMARY

1. The JVM will attempt to find a reasonable minimum and maximum heap size based on the machine it is running on.
2. Unless the application needs a larger heap than the default, consider tuning the performance goals of a GC algorithm (given in the next chapter) rather than fine-tuning the heap size.

Sizing the Generations

Once the heap size has been determined, you (or the JVM) must decide how much of the heap to allocate to the young generation, and how much to allocate to the old generation. The performance implication of this should be clear: if there is a relatively larger young generation, then it will be collected less often, and fewer objects will be promoted into the old generation. But on the other hand, because the old generation is relatively smaller, it will fill up more frequently and do more full GCs. Striking a balance here is the key.

Different GC algorithms attempt to strike this balance in different ways. However, all GC algorithms use the same set of flags to set the sizes of the generations; this section covers those common flags.

The command-line flags to tune the generation sizes all adjust the size of the young generation; the old generation gets everything that is left over. There are a variety of flags that can be used to size the young generation:

`-XX:NewRatio=`*`N`*

Set the ratio of the young generation to the old generation.

`-XX:NewSize=`*`N`*

Set the initial size of the young generation.

`-XX:MaxNewSize=`*`N`*

Set the maximum size of the young generation.

`-Xmn`*`N`*

Shorthand for setting both `NewSize` and `MaxNewSize` to the same value.

The young generation is first sized by the `NewRatio`, which has a default value of 2. Parameters that affect the sizing of heap spaces are generally specified as ratios; the value is used in an equation to determine the percentage of space affected. The `NewRatio` value is used in this formula:

```
Initial Young Gen Size = Initial Heap Size / (1 + NewRatio)
```

Plugging in the initial size of the heap and the `NewRatio` yields the value that becomes the setting for the young generation. By default, then, the young generation starts out at 33% of the initial heap size.

Alternately, the size of the young generation can be set explicitly by specifying the `NewSize` flag. If that option is set, it will take precedence over the value calculated from the `NewRatio`. There is no default for this flag since the default is to calculate it from `NewRatio`.

As the heap expands, the young generation size will expand as well, up to the maximum size specified by the `MaxNewSize` flag. By default, that maximum is also set using the `NewRatio` value, though it is based on the maximum (rather than initial) heap size.

Tuning the young generation by specifying a range for its minimum and maximum sizes ends up being fairly difficult. When a heap size is fixed (by setting `-Xms` equal to `-Xmx`), it is usually preferable to use `-Xmn` to specify a fixed size for the young generation as well. If an application needs a dynamically sized heap and requires a larger (or smaller) young generation, then focus on setting the `NewRatio` value.

QUICK SUMMARY

1. Within the overall heap size, the sizes of the generations are controlled by how much space is allocated to the young generation.

2. The young generation will grow in tandem with the overall heap size, but it can also fluctuate as a percentage of the total heap (based on the initial and maximum size of the young generation).

Sizing Metaspace

When the JVM loads classes, it must keep track of certain metadata about those classes. Until JDK 8, this occupied a separate heap space (called the *permgen* (or permanent generation); it is now referred to as the *metaspace*.

As end users, the metaspace is opaque to us: we know that it holds a bunch of class-related data, and that there are certain circumstances where the size of that region needs to be tuned.

Note that the metaspace does not hold the actual instance of the class (the `Class` objects), nor reflection objects (e.g., `Method` objects); those are held in the regular heap. Information in the metaspace is really only used by the compiler and JVM runtime, and the data it holds is referred to as class metadata.

There isn't a good way to calculate in advance how much space a particular program needs for its metaspace. The size will be proportional to the number of classes it uses, so bigger applications will need bigger areas. This is another area where changes in JDK technology have made life easier: tuning the permgen used to be fairly common, but tuning the metaspace is fairly rare these days. The main reason for that is the default values for the size of the metaspace are very generous; Table 1-10 lists the default initial and maximum sizes.

Table 1-10. Default sizes of the metaspace

JVM	Default initial size	Default maximum metaspace size
32-bit client JVM	12 MB	Unlimited
32-bit server JVM	16 MB	Unlimited
64-bit JVM	20.75 MB	Unlimited

The metaspace behaves just like a separate instance of the regular heap. It is sized dynamically based on an initial size (`-XX:MetaspaceSize=`*N*) and will increase as needed to a maximum size (`-XX:MaxMetaspaceSize=`*N*).

METASPACE TOO BIG?

Because the default size of metaspace is unlimited, there is the possibility (particularly in a 32-bit JVM) that an application can run out of memory by filling up metaspace. The Native Memory Tracking (NMT) tools can help diagnose that case. If metaspace is growing too big, you can set the value of `MaxMetaspaceSize` lower—but then the application will eventually get an `OutOfMemoryError` when the metaspace fills up. Figuring out why the class metadata is too large is the real remedy in that case.

Resizing the metaspace requires a full GC, so it is an expensive operation. If there are a lot of full GCs during the startup of a program (as it is loading classes), it is often because permgen or metaspace is being resized, so increasing the initial size is a good idea to improve startup in that case. Application servers, for example, typically specify an initial metaspace size of 128 MB, 192 MB, or more.

Java classes can be eligible for GC just like anything else. This is a very common occurrence in an application server, which creates new classloaders every time an application is deployed (or redeployed). The old classloaders are then unreferenced and eligible for GC, as are any classes that they defined. Meanwhile the new classes of the application will have new metadata, and so there must be room in the metaspace for that. This often causes a full GC since the metaspace needs to grow (or discard old metadata).

One reason to limit the size of the metaspace is to guard against a classloader leak: then the application server (or other program like an IDE) continually defines new classloaders and classes while maintaining references to the old classloaders. This has the potential to fill up the metaspace and consume a lot of memory on the machine. On the other hand, the actual classloader and class objects in that case are also still in the main heap—and that heap is likely to fill up and cause an OutOfMemoryError before the memory occupied by the metaspace becomes a problem.

Heap dumps can be used to diagnose what classloaders exist, which in turn can help determine if a classloader leak is filling up metaspace. Otherwise, `jmap` can be used with the argument `-clstats` to print out information about the classloaders.

QUICK SUMMARY

1. The metaspace holds class metadata (not class objects itself). It behaves like a separate heap.

2. The initial size of this region can be based on its usage after all classes have been loaded. That will slightly speed up startup.

3. Application servers doing development (or any environment where classes are frequently redefined) will see an occasional full GC caused when metaspace fills up and old class metadata is discarded.

Controlling Parallelism

All GC algorithms except the serial collector use multiple threads. The number of these threads is controlled by the `-XX:ParallelGCThreads=`*`N`* flag. The value of this flag affects the number of threads used for the following operations:

- Collection of the young generation when using `-XX:+UseParallelGC`

- Collection of the old generation when using `-XX:+UseParallelGC`

- Collection of the young generation when using `-XX:+UseG1GC`

- Stop-the-world phases of G1 GC (though not full GCs)

Because these GC operations stop all application threads from executing, the JVM attempts to use as many CPU resources as it can in order to minimize the pause time. By default, that means the JVM will run one thread for each CPU on a machine, up to eight. Once that threshold has been reached, the JVM only adds a new thread for every five-eighths of a CPU. So the total number of threads (where `N` is the number of CPUs) on a machine with more than eight CPUs is:

```
ParallelGCThreads = 8 + ((N - 8) * 5 / 8)
```

There are times when this number is too large. An application using a small heap (say, 1 GB) on a machine with eight CPUs will be slightly more efficient with four or six threads dividing up that heap. On a 128-CPU machine, 83 GC threads is too many for all but the largest heaps.

Additionally, if more than one JVM is running on the machine, it is a good idea to limit the total number of GC threads among all JVMs. When they run, the GC threads are quite efficient and each will consume 100% of a single CPU (this is why the average CPU usage for the throughput collector was higher than expected in previous examples). In machines with eight or fewer CPUs, GC will consume 100% of the CPU on the machine. On machines with more CPUs and multiple JVMs, there will still be too many GC threads running in parallel.

Take the example of a 16-CPU machine running four JVMs; each JVM will have by default 13 GC threads. If all four JVMs execute GC at the same time, the machine will have 52 CPU-hungry threads contending for CPU time. That results in a fair amount of contention; it will be more efficient if each JVM is limited to four GC threads. Even though it may be unlikely for all four JVMs to perform a GC operation at the same time, one JVM executing GC with 13 threads means that the application threads in the remaining JVMs now have to compete for CPU resources on a machine where 13 of 16 CPUs are 100% busy executing GC tasks. Giving each JVM four GC threads provides a better balance in this case.

Note that this flag does not set the number of background threads used by G1 GC (though it does affect that). Details on that are given in the next chapter.

QUICK SUMMARY

1. The basic number of threads used by all GC algorithms is based on the number of CPUs on a machine.

2. When multiple JVMs are run on a single machine, that number will be too high and must be reduced.

Adaptive Sizing

The sizes of the heap, the generations, and the survivor spaces can vary during execution as the JVM attempts to find the optimal performance according to its policies and tunings.

This is a best-effort solution, and it relies on past performance: the assumption is that future GC cycles will look similar to the GC cycles in the recent past. That turns out to be a reasonable assumption for many workloads, and even if the allocation rate suddenly changes, the JVM will readapt its sizes based on the new information.

Adaptive sizing provides benefits in two important ways. First, it means that small applications don't need to worry about overspecifying the size of their heap. Consider the administrative command-line programs used to adjust the operations of things like an application server—those programs are usually very short-lived and use minimal memory resources. These applications will use 64 (or 16) MB of heap even though the default heap could potentially grow to 1 GB. Because of adaptive sizing, applications like that don't need to be specifically tuned; the platform defaults ensure that they will not use a large amount of memory.

Second, it means that many applications don't really need to worry about tuning their heap size at all—or if they need a larger heap than the platform default, they can just specify that larger heap and forget about the other details. The JVM can autotune the heap and generation sizes to use an optimal amount of memory given the GC algorithm's performance goals. Adaptive sizing is what allows that autotuning to work.

Still, doing the adjustment of the sizes takes a small amount of time—which occurs for the most part during a GC pause. If you have taken the time to finely tune GC parameters and the size constraints of the application's heap, adaptive sizing can be disabled. Disabling adaptive sizing is also useful for applications that go through markedly different phases, if you want to optimally tune GC for one of those phases.

At a global level, adaptive sizing is disabled by turning off the `-XX:-UseAdaptiveSizePolicy` flag (which is `true` by default). With the exception of the survivor spaces (which are examined in detail in the next chapter), adaptive sizing is also effectively turned off if the minimum and maximum heap sizes are set to the same value, and the initial and maximum sizes of the new generation are set to the same value.

To see how the JVM is resizing the spaces in an application, set the `-XX:+PrintAdaptiveSizePolicy` flag. When a GC is performed, the GC log will contain information detailing how the various generations were resized during a collection.

QUICK SUMMARY

1. Adaptive sizing controls how the JVM alters the ratio of young generation to old generation within the heap.
2. Adaptive sizing should generally be kept enabled, since adjusting those generation sizes is how GC algorithms attempt to meet their pause time goals.
3. For finely tuned heaps, adaptive sizing can be disabled for a small performance boost.

GC Tools

Since GC is central to the performance of Java, there are many tools that monitor its performance.

The best way to see what effect GC has on the performance of an application is to become familiar with the GC log, which is a record of every GC operation during the program's execution.

The details in the GC log vary depending on the GC algorithm, but the basic management of the log is the same for all algorithms. The log management is not the same, however, between JDK 8 and subsequent releases: JDK 8 uses a different set of command-line arguments to enable and manage the GC log. We'll discuss the management of GC logs here, and more details on the contents of the log are given in the algorithm-specific tuning sections in the next chapter.

Enabling GC Logging in JDK 8

In JDK 8, there are multiple ways to enable the GC log: specifying either of the flags `-verbose:gc` or `-XX:+PrintGC` will create a simple GC log (the flags are aliases for each other, and by default the log is disabled). The `-XX:+PrintGCDetails` flag will create a log with much more information. This flag is recommended (it is also `false` by default); it is often too difficult to diagnose what is happening with GC using only the simple log. In conjunction with the detailed log, it is recommended to include `-XX:+PrintGCTimeStamps` or `-XX:+PrintGCDateStamps`, so that the time between GC operations can be determined. The difference in those two arguments is that the timestamps are relative to 0 (based on when the JVM starts), while the date stamps are an actual date string. That makes the date stamps ever-so-slightly less efficient as the dates are formatted, though it is an infrequent enough operation that its effect is unlikely to be noticed.

The GC log is written to standard output, though that location can be changed with the `-Xloggc:`*`filename`* flag. Using `-Xloggc` automatically enables the simple GC log unless `PrintGCDetails` has also been enabled.

The amount of data that is kept in the GC log can be limited using log rotation; this is quite useful for a long-running server that might otherwise fill up its disk with logs over several months. Logfile rotation is controlled with these flags: `-XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=`*`N`* `-XX:GCLogFileSize=`*`N`*. By default, `UseGCLogFileRotation` is disabled. When that flag is enabled, the default number of files is 0 (meaning unlimited), and the default logfile size is 0 (meaning unlimited). Hence, values must be specified for all these options in order for log rotation to work as expected. Note that a logfile size will be rounded up to 8 KB for values less than that.

Putting that altogether, a useful set of flags for logging is:

```
-Xloggc:gc.log -XX:+PrintGCTimeStamps -XX:+UseGCLogFileRotation
-XX:NumberOfGCLogFile=8 -XX:GCLogFileSize=8m
```

That will log GC events with timestamps to correlate to other logs and limit the total amount of retained logs to 64m in eight files. This logging is minimal enough that it can be enabled even on production systems.

Enabling GC Logging in JDK 11

JDK 11 and later versions use Java's new unified logging feature. This means that all logging—GC related or not—is enabled via the flag `-Xlog`. Then you append various options to that flag that control how the logging should be performed. In order to specify logging similar to the long example from JDK 8, you would specify this flag:

```
-Xlog:gc*:file=gc.log:time:filecount=7,filesize=8M+.
```

The colons divide the command into four sections. You can run `java -Xlog:help:` to get more information on the available options, but here's how they map for this string.

The first section (`gc*`) specifies which modules should enable logging; we are enabling logging for all GC modules. There are options to log only a particular section (e.g., `gc+age` will log information about the tenuring of an object, a topic covered in the next chapter). Those specific modules often have limited output at the default logging level, so you might use something like `gc*,gc+age=debug` to log basic (info level) messages from all gc modules and debug level messages from the tenuring code. Typically, logging all modules at info level is fine.

The second section sets the destination of the log file.

The third section (`time`) is a decorator: that decorator says to log messages with a time-of-day stamp, the same as we specified for JDK 8. Multiple decorators can be specified.

Finally, the fourth section specifies output options; in this case we've said to rotate logs when they hit 8M, and keeping eight logs altogether.

One thing to note: log rotation is handled slightly differently between JDK8 and JDK11. Say that we have specified a log name of gc.log and that three files should be retained. In JDK8, the logs will be written this way:

- Start logging to *gc.log.0.current*
- When full, rename that to *gc.log.0* and start logging to *gc.log.1.current*
- When full, rename that to *gc.log.1* and start logging to *gc.log.2.current*
- When full, rename that to *gc.log.2*, remove *gc.log.0*, and start logging to a new *gc.log.0.current*
- Repeat this cycle

In JDK11, the logs will be written this way:

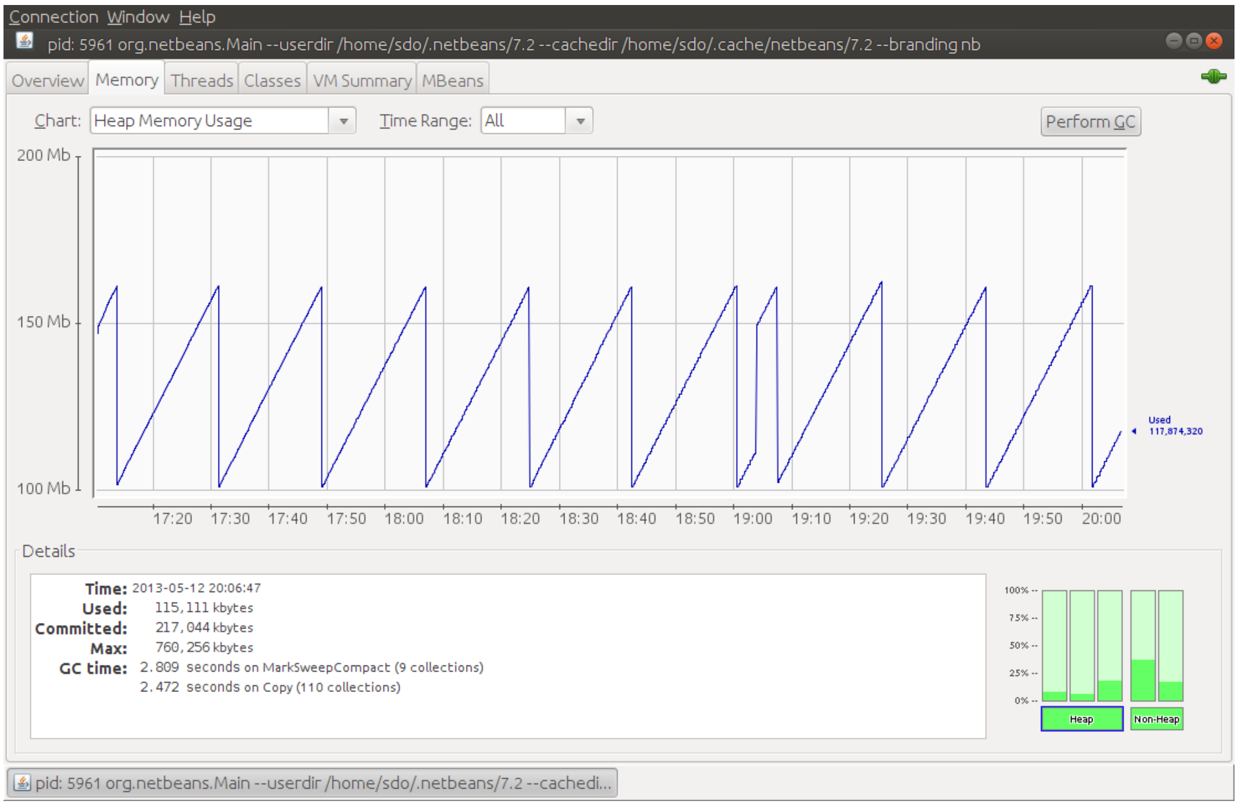
- Start logging to *gc.log*
- When that is full, rename it to *gc.log.0* and start a new *gc.log*
- When that is full, rename it to *gc.log.1* and start a new *gc.log*
- When that is full, rename it to *gc.log.2* and start a new *gc.log*
- When that is full, rename it to *gc.log.0*, removing the old *gc.log.0*, and start a new *gc.log*

If you were wondering why we specified seven logs to retain in the previous JDK 11 command, this is why: there will be eight active files in this case. Also note in either case that the number appended to the file doesn't mean anything about the order in which the files weres created. The numbers are reused in a cycle, so there is some order, but the oldest log file could be any one in the set.

The gc log contains a lot of information specific to each collector, so we'll step through them in detail in the next chapter. It is also quite useful to parse the logs for aggregate information about your application: how many pauses it had, how long the took on average and in total, and so on.

Unfortunately, there are not a lot of good open-source tools to parse log files. Like profilers, this is an area where commercial vendors have stepped in to provide support, like the offerings from jClarity (Censum) and gceasy.io. The latter of these has a free service for basic log parsing.

For real-time monitoring of the heap, use `jconsole`. The Memory panel of `jconsole` displays a real-time graph of the heap as shown in Figure 1-4.



```
      Time: 2013-05-12 20:06:47
      Used:    115,111 kbytes
 Committed:    217,044 kbytes
       Max:    760,256 kbytes
   GC time:  2.809  seconds on MarkSweepCompact (9 collections)
             2.472  seconds on Copy (110 collections)
```

Figure 1-4. Real-time heap display

This particular view shows the entire heap, which is periodically cycling between using about 100 MB and 160 MB. `jconsole` can instead display only eden, or the survivor spaces, or the old generation, or the permanent generation. If I'd selected eden as the region to chart, it would have shown a similar pattern as eden fluctuated between 0 MB and 60 MB (and, as you can guess, that means if I'd charted the old generation, it would have been essentially a flat line at 100 MB).

For a scriptable solution, `jstat` is the tool of choice. `jstat` provides nine options to print different information about the heap; `jstat -options` will provide the full list. One useful option is `-gcutil`, which displays the time spent in GC as well as the percentage of each GC area that is currently filled. Other options to `jstat` will display the GC sizes in terms of KB.

Remember that `jstat` takes an optional argument—the number of milliseconds to repeat the command—so it can monitor over time the effect of GC in an application. Here is some sample output repeated every second:

```
% jstat -gcutil process_id 1000
  S0     S1     E      O      P     YGC     YGCT    FGC    FGCT     GCT
 51.71   0.00  99.12  60.00  99.93     98    1.985     8    2.397    4.382
  0.00  42.08   5.55  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08   6.32  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08  68.06  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08  82.27  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08  96.67  60.98  99.93     99    2.016     8    2.397    4.413
  0.00  42.08  99.30  60.98  99.93     99    2.016     8    2.397    4.413
 44.54   0.00   1.38  60.98  99.93    100    2.042     8    2.397    4.439
 44.54   0.00   1.91  60.98  99.93    100    2.042     8    2.397    4.439
```

When monitoring started, the program had already performed 98 collections of the young generation (`YGC`), which took a total of 1.985 seconds (`YGCT`). It had also performed eight full GCs (`FGC`) requiring 2.397 seconds (`FGCT`); hence the total time in GC (`GCT`) was 4.382 seconds.

All three sections of the young generation are displayed here: the two survivor spaces (`S0` and `S1`) and eden (`E`). The monitoring started just as eden was filling up (99.12% full), so in the next second there was a young collection: eden reduced to 5.55% full, the survivor spaces switched places, and a small amount of memory was promoted to the old generation (`O`), which increased to using 60.98% of its space. As is typical, there is little or no change in the permanent generation (`P`) since all necessary classes have already been loaded by the application.

If you’ve forgotten to enable GC logging, this is a good substitute to watch how GC operates over time.

QUICK SUMMARY

1. GC logs are the key piece of data required to diagnose GC issues; they should be collected routinely (even on production servers).
2. A better GC logfile is obtained with the `PrintGCDetails` flag.
3. Programs to parse and understand GC logs are readily available; they are quite helpful in summarizing the data in the GC log.
4. `jstat` can provide good visibility into GC for a live program.

Summary

Performance of the garbage collector is one key feature to the overall performance of any Java application. For many applications, though, the only tuning required is to select the appropriate GC algorithm and, if needed, to increase the heap size of the application. Adaptive sizing will then allow the JVM to autotune its behavior to provide good performance using the given heap.

More complex applications will require additional tuning, particularly for specific GC algorithms. If the simple GC settings in this chapter do not provide the performance an application requires, consult the tunings described in the next chapter.

Chapter 2. Garbage Collection Algorithms

Chapter 1 examined the general behavior of all garbage collectors, including JVM flags that apply universally to all GC algorithms: how to select heap sizes, generation sizes, logging, and so on.

The basic tunings of garbage collection suffice for many circumstances. When they do not, it is time to examine the specific operation of the GC algorithm in use to determine how its parameters can be changed to minimize the impact of GC on the application.

The key information needed to tune an individual collector is the data from the GC log when that collector is enabled. This chapter starts, then, by looking at each algorithm from the perspective of its log output, which allows us to understand how the GC algorithm works and how it can be adjusted to work better. Each section then includes tuning information to achieve that better performance.

This chapter also covers the details of some new, experimental collectors. Those collectors may not be 100% solid at the time of this writing but will likely become full-fledged, production-worthy collectors by the time the next LTS version of Java is released (just as G1 GC began as an experimental collector and is now the default in JDK 11).

There are a few unusual cases that impact the performance of all GC algorithms: allocation of very large objects, objects that are neither short- nor long-lived, and so on. Those cases are covered at the end of this chapter.

Understanding the Throughput Collector

We'll start by looking at the individual garbage collectors, beginning with the throughput collector. Although we've seen that the G1 GC collector is generally preferred, the details of the throughput collector are easier and make a better foundation for understanding how things work.

Recall from Chapter 1 that garbage collectors must do three basic operations: find unused objects, free their memory, and compact the heap. The throughput collector does all of those operations in the same GC cycle; together those operations are referred to as a collection. These collectors can collect either the young generation or the old generation during a single operation.

Figure 2-1 shows the heap before and after a young collection.

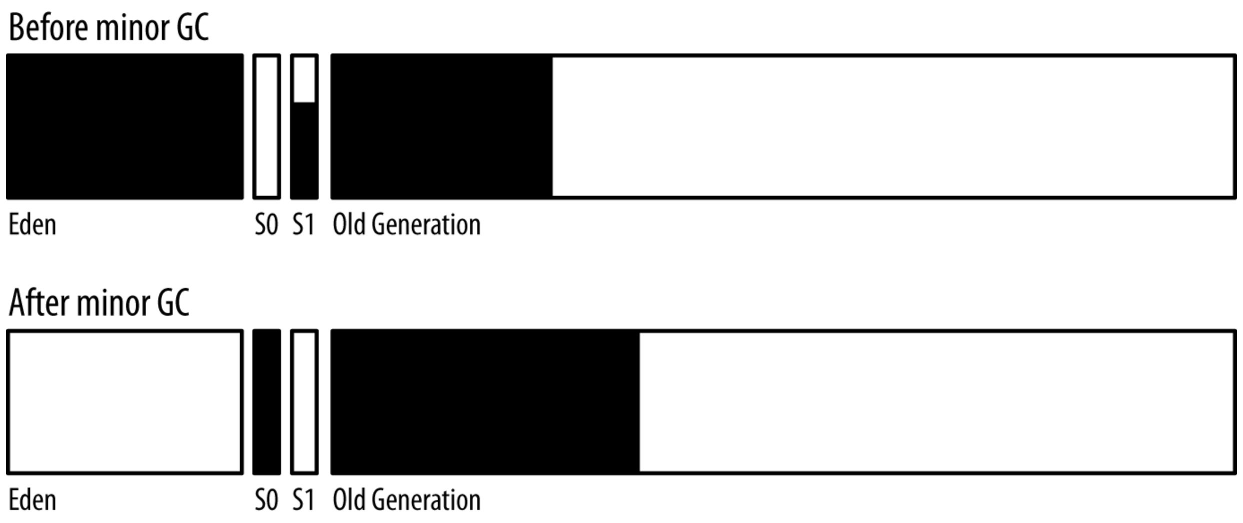


Figure 2-1. A throughput GC young collection

A young collection occurs when eden has filled up. The young collection moves all objects out of eden: some are moved to one of the survivor spaces (S0 in this diagram) and some are moved to the old generation, which now contains more objects. Many objects, of course, are discarded because they are no longer referenced.

Because eden is usually empty after this operation, it may seem unusual to consider that it has been compacted, but that’s the effect here. It is possible in some circumstances that the old generation is full and some objects remain in eden, in which case they will be moved to the beginning of eden (which is a more obvious compaction).

In the JDK 8 GC log with `PrintGCDetails`, a minor GC of the throughput collector appears like this:

```
17.806: [GC (Allocation Failure) [PSYoungGen: 227983K->14463K(264128K)]
            280122K->66610K(613696K), 0.0169320 secs]
            [Times: user=0.05 sys=0.00, real=0.02 secs]
```

This GC occurred 17.806 seconds after the program began. Objects in the young generation now occupy 14463 KB (14 MB, in the survivor space); before the GC, they occupied 227983 KB (227 MB). (Actually, 227893 KB is only 222 MB. For ease of discussion, I’ll just truncate the KBs by 1000 in this chapter. Pretend I am a disk manufacturer.) The total size of the young generation at this point is 264 MB.

Meanwhile the overall occupancy of the heap (both young and old generations) decreased from 280 MB to 66 MB, and the size of the entire heap at this point in time was 613 MB. The operation took less than 0.02 seconds (the 0.02 seconds of real time at the end of the output is 0.0169320 seconds—the actual time—rounded). The program was charged for more CPU time than real time because the young collection was done by multiple threads (in this configuration, four threads).

The same log in JDK11 would look something like this:

```
[17.805s][info][gc,start       ] GC(4) Pause Young (Allocation Failure)
[17.806s][info][gc,heap        ] GC(4) PSYoungGen: 227983K->14463K(264128K
[17.806s][info][gc,heap        ] GC(4) ParOldGen: 280122K->66610K(613696K)
[17.806s][info][gc,metaspace   ] GC(4) Metaspace: 3743K->3743K(1056768K)
[17.806s][info][gc             ] GC(4) Pause Young (Allocation Failure)
                                          496M->79M(857M) 16.932ms
[17.086s][info][gc,cpu         ] GC(4) User=0.05s Sys=0.00s Real=0.02s
```

The information here is the same; it's just a different format. And this log entry is multiple lines; the previous log entry is actually a single line (but that doesn't reproduce in this format). This log also prints out the metaspace sizes, but those will never change during a young collection. The metaspace is also not included in the total heap size reported on the fifth line of this sample.

Figure 2-2 shows the heap before and after a full GC.

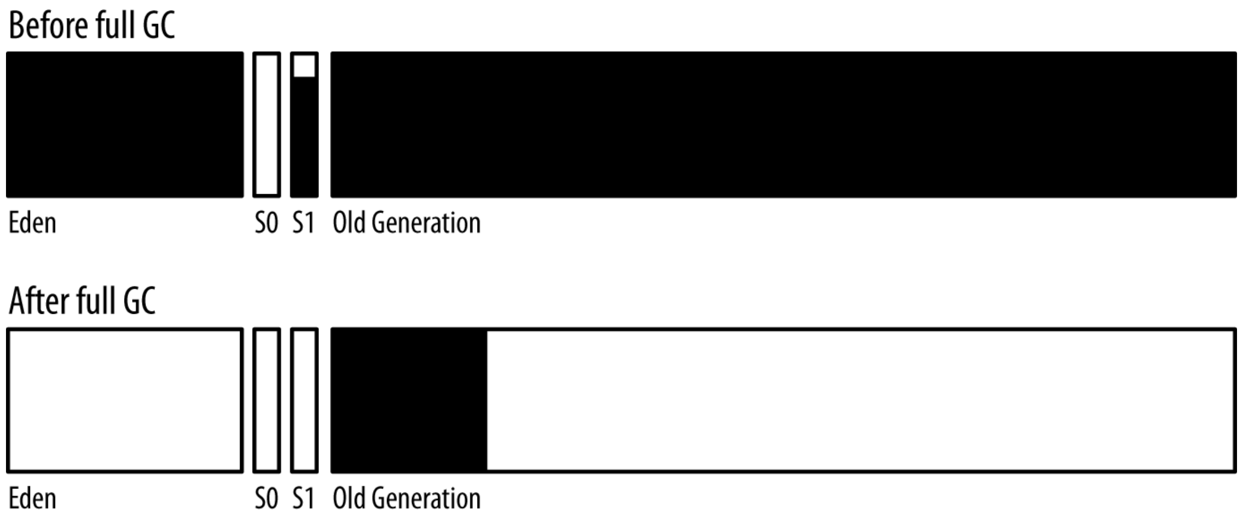


Figure 2-2. A throughput full GC

The old collection frees everything out of the young generation (including the survivor spaces). The only objects that remain in the old generation are those which have active references, and all of those objects have been compacted so that the beginning of the old generation is occupied, and the remainder is free.

The GC log reports that operation like this:

```
64.546: [Full GC (Ergonomics) [PSYoungGen: 15808K->0K(339456K)]
          [ParOldGen: 457753K->392528K(554432K)] 473561K->392528K(893888K)
          [Metaspace: 56728K->56728K(115392K)], 1.3367080 secs]
          [Times: user=4.44 sys=0.01, real=1.34 secs]
```

The young generation now occupies 0 bytes (and its size is 339 MB). The data in the old generation decreased from 457 MB to 392 MB, and hence the entire heap usage has decreased from 473 MB to 392 MB. The size of the metaspace is unchanged; it is not collected during most full GCs. (If the metaspace runs out of room, the JVM will run a full GC to collect it, and you will see the size of the metaspace change; we'll show that a little further on.) Because there is substantially more work to do in a full GC, it has taken 1.3 seconds of real time, and 4.4 seconds of CPU time (again for four parallel threads).

The similar JDK 11 log is this:

```
[63.205s][info][gc,start       ] GC(13) Pause Full (Ergonomics)
[63.205s][info][gc,phases,start] GC(13) Marking Phase
[63.314s][info][gc,phases      ] GC(13) Marking Phase 109.273ms
[63.314s][info][gc,phases,start] GC(13) Summary Phase
[63.316s][info][gc,phases      ] GC(13) Summary Phase 1.470ms
[63.316s][info][gc,phases,start] GC(13) Adjust Roots
[63.331s][info][gc,phases      ] GC(13) Adjust Roots 14.642ms
[63.331s][info][gc,phases,start] GC(13) Compaction Phase
[63.482s][info][gc,phases      ] GC(13) Compaction Phase 1150.792ms
[64.482s][info][gc,phases,start] GC(13) Post Compact
[64.546s][info][gc,phases      ] GC(13) Post Compact 63.812ms
[64.546s][info][gc,heap        ] GC(13) PSYoungGen: 15808K->0K(339456K)
[64.546s][info][gc,heap        ] GC(13) ParOldGen: 457753K->392528K(554432
[64.546s][info][gc,metaspace   ] GC(13) Metaspace: 56728K->56728K(115392K)
[64.546s][info][gc             ] GC(13) Pause Full (Ergonomics)
                                            462M->383M(823M) 1336.708ms
[64.546s][info][gc,cpu         ] GC(13) User=4.446s Sys=0.01s Real=1.34s
```

QUICK SUMMARY

1. The throughput collector has two operations: minor collections and full GCs, each of which marks, frees, and compacts the target generation.
2. Timings taken from the GC log are a quick way to determine the overall impact of GC on an application using these collectors.

Adaptive and Static Heap Size Tuning

Tuning the throughput collector is all about pause times and striking a balance between the overall heap size and the sizes of the old and young generations.

There are two trade-offs to consider here. First, there is the classic programming trade-off of time versus space. A larger heap consumes more memory on the machine, and the benefit of consuming that memory is (at least to a certain extent) that the application will have a higher throughput.

The second trade-off concerns the length of time it takes to perform GC. The number of full GC pauses can be reduced by increasing the heap size, but that may have the perverse effect of increasing average response times because of the longer GC times. Similarly, full GC pauses can be shortened by allocating more of the heap to the young generation than to the old generation, but that in turn increases the frequency of the old GC collections.

The effect of these trade-offs is shown in Figure 2-3. This graph shows the maximum throughput of the stock servlet application running in a GlassFish instance with different heap sizes. With a small 256 MB heap, the application server is spending quite a lot of time in GC (36% of total time, in fact); the throughput is restricted as a result. As the heap size is increased, the throughput rapidly increases—until the heap size is set to 1,500 MB. After that, throughput increases less rapidly: the application isn't really GC-bound at that point (about 6% of time in GC). The law of diminishing returns has crept in here: the application can use additional memory to gain throughput, but the gains become more limited.

After a heap size of 4,500 MB, the throughput starts to decrease slightly. At that point, the application has reached the second trade-off: the additional memory has caused much longer GC cycles, and those longer cycles—even though they are less frequent—can impact the overall throughput.

The data in this graph was obtained by disabling adaptive sizing in the JVM; the minimum and maximum heap sizes were set to the same value. It is possible to run experiments on any application and determine the best sizes for the heap and for the generations, but it is often easier to let the JVM make those decisions (which is what usually happens, since adaptive sizing is enabled by default).

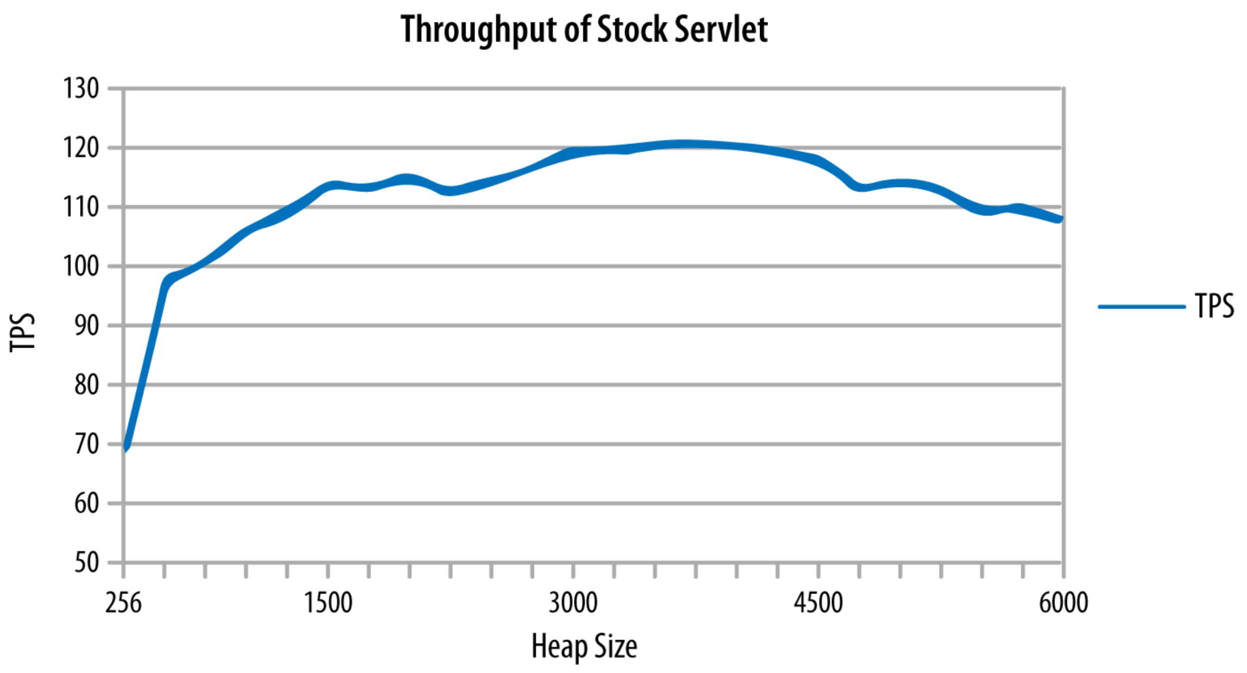


Figure 2-3. Throughput with various heap sizes

Adaptive sizing in the throughput collector will resize the heap (and the generations) in order to meet its pause time goals. Those goals are set with these flags: `-XX:MaxGCPauseMillis=`*`N`* and `-XX:GCTimeRatio=`*`N`*.

The `MaxGCPauseMillis` flag specifies the maximum pause time that the application is willing to tolerate. It might be tempting to set this to 0, or perhaps some very small value like 50 ms. Be aware that this goal applies to both minor and full GCs. If a very small value is used, the application will end up with a very small old generation: for example, one that can be cleaned in 50 ms. That will cause the JVM to perform very, very frequent full GCs, and performance will be dismal. So be realistic: set the value to something that can actually be achieved. By default, this flag is not set.

The `GCTimeRatio` flag specifies the amount of time you are willing for the application to spend in GC (compared to the amount of time its application-level threads should run). It is a ratio, so the value for *`N`* takes a little thought. The value is used in the following equation to determine the percentage of time the application threads should ideally run:

$$ThroughputGoal = 1 - \frac{1}{(1 + GCTimeRatio)}$$

The default value for `GCTimeRatio` is 99. Plugging that value into the equation yields 0.99, meaning that the goal is to spend 99% of time in application processing, and only 1% of time in GC. But don't be confused by how those numbers line up in the default case. A `GCTimeRatio` of 95 does not mean that GC should run up to 5% of the time: it means that GC should run up to 1.94% of the time.

It's easier to decide the minimum percentage of time you want the application to perform work (say, 95%), and then calculate the value of the `GCTimeRatio` from this equation:

$$GCTimeRatio = \frac{Throughput}{(1 - Throughput)}$$

For a throughput goal of 95% (0.95), this equation yields a `GCTimeRatio` of 19.

The JVM uses these two flags to set the size of the heap within the boundaries established by the initial (`-Xms`) and maximum (`-Xmx`) heap sizes. The `MaxGCPauseMillis` flag takes precedence: if it is set, the sizes of the young and old generation are adjusted until the pause time goal is met. Once that happens, the overall size of the heap is increased until the time ratio goal is met. Once both goals are met, the JVM will attempt to reduce the size of the heap so that it ends up with the smallest possible heap that meets these two goals.

Because the pause time goal is not set by default, the usual effect of automatic heap sizing is that the heap (and generation) sizes will increase until the `GCTimeRatio` goal is met. In practice, though, the default setting of that flag is quite optimistic. Your experience will vary, of course, but I am much more used to seeing applications that spend 3% to 6% of their time in GC and behave quite well. Sometimes I even work on applications in environments where memory is severely constrained; those applications end up spending 10% to 15% of their time in GC. GC has a substantial impact on the performance of those applications, but the overall performance goals are still met.

So the best setting will vary depending on the application goals. In the absence of other goals, I start with a time ratio of 19 (5% of time in GC).

Table 2-1 shows the effects of this dynamic tuning for an application that needs a small heap and does little GC (it is the stock servlet running in GlassFish where no session state is saved, and there are very few long-lived objects).

Table 2-1. Effect of dynamic GC tuning

GC settings	End heap size	Percent time in GC	OPS
Default	649 MB	0.9%	9.2
`MaxGCPauseMillis=50ms`	560 MB	1.0%	9.2
`Xms=Xmx=2048m`	2 GB	0.04%	9.2

By default, the heap will have a 64 MB minimum size and a 2 GB maximum size (since the machine has 8 GB of physical memory). In that case, the `GCTimeRatio` works just as expected: the heap dynamically resized to 649 MB, at which point the application was spending about 1% of total time in GC.

Setting the `MaxGCPauseMillis` flag in this case starts to reduce the size of the heap in order to meet that pause time goal. Because there is so little work for the garbage collector to perform in this example, it succeeds and can still spend only 1% of total time in GC, while maintaining the same throughput of 9.2 OPS.

Finally, notice that more isn't always better—a full 2 GB heap does mean that the application can spend less time in GC, but GC isn't the dominant performance factor here, so the throughput doesn't increase. As usual, spending time optimizing the wrong area of the application has not helped.

If the same application is changed so that the previous 50 requests for each user are saved in the session state, the garbage collector has to work harder. Table 2-2 shows the trade-offs in that situation.

Table 2-2. Effect of dynamic GC tuning

GC settings	End heap size	Percent time in GC	OPS
Default	1.7 GB	9.3%	8.4
`MaxGCPauseMillis=50ms`	588 MB	15.1%	7.9
`Xms=Xmx=2048m`	2 GB	5.1%	9.0
`Xmx=3560M`; `MaxGCRatio=19`	2.1 GB	8.8%	9.0

In a test that spends a significant amount of time in GC, things are different. The JVM will never be able to satisfy the 1% throughput goal in this test; it tries its best to accommodate the default goal and does a reasonable job, using 1.7 GB of space.

Things are worse when an unrealistic pause time goal is given. To achieve a 50 ms collection time, the heap is kept to 588 MB, but that means that GC now becomes excessively frequent. Consequently, the throughput has decreased significantly. In this scenario, the better performance comes from instructing the JVM to utilize the entire heap by setting both the initial and maximum sizes to 2 GB.

Finally, the last line of the table shows what happens when the heap is reasonably sized and we set a realistic time ratio goal of 5%. The JVM itself determined that approximately 2 GB was the optimal heap size, and it achieved the same throughput as the hand-tuned case.

QUICK SUMMARY

1. Dynamic heap tuning is a good first step for heap sizing. For a wide set of applications, that will be all that is needed, and the dynamic settings will minimize the JVM's memory use.

2. It is possible to statically size the heap to get the maximum possible performance. The sizes the JVM determines for a reasonable set of performance goals are a good first start for that tuning.

Understanding the G1 Garbage Collector

G1 GC operates on discrete regions within the heap. Each region (there are by default around 2,048 of them) can belong to either the old or new generation, and the generational regions need not be contiguous. The idea behind having regions in the old generation is that when the concurrent background threads look for unreferenced objects, some regions will contain more garbage than other regions. The actual collection of a region still requires that application threads be stopped, but G1 GC can focus on the regions that are mostly garbage and only spend a little bit of time emptying those regions. This approach—clearing out only the mostly garbage regions—is what gives G1 GC its name: Garbage First.

That doesn't apply to the regions in the young generation: during a young GC, the entire young generation is either freed or promoted (to a survivor space or to the old generation). Still, the young generation is defined in terms of regions, in part because it makes resizing the generations much easier if the regions are predefined.

G1 GC is called a concurrent collector because the marking of free objects within the old generation happens concurrently with the application threads (i.e., they are left running). But it is not completely concurrent because the marking and compacting of the young generation requires stopping all application threads, and the compacting of the old generation also occurs while the application threads are stopped.

G1 GC has four logical operations:

- A young collection

- A background, concurrent marking cycle

- A mixed collection

- If necessary, a full GC

We'll look at each of those in turn, starting with the G1 GC young collection shown in Figure 2-4.

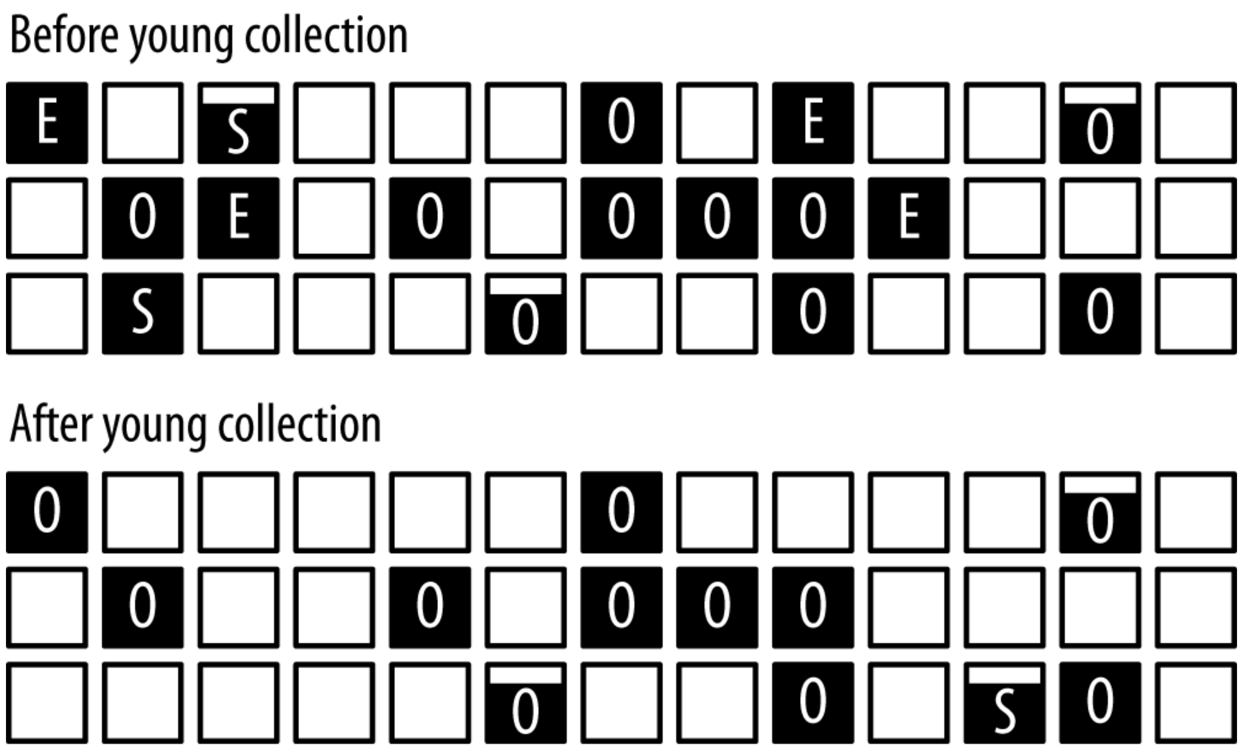


Figure 2-4. A G1 GC young collection

Each small square in this figure represents a G1 GC region. The data in each region is represented by the black area of the region, and the letter in each region identifies the generation to which the region belongs ([E]den, [O]ld generation, [S]urvivor space). Empty regions do not belong to a generation; G1 GC uses them arbitrarily for whichever generation it deems necessary.

The G1 GC young collection is triggered when eden fills up (in this case, after filling four regions). After the collection, eden is empty (though there are regions assigned to it, which will begin to fill up with data as the application proceeds). There is at least one region assigned to the survivor space (partially filled in this example), and some data has moved into the old generation.

The GC log illustrates this collection a little differently in G1 than in other collectors. The JDK 8 example log was taken using `PrintGCDetails`, but the details in the log for G1 GC are more verbose. The examples show only a few of the important lines.

Here is the standard collection of the young generation:

```
23.430: [GC pause (young), 0.23094400 secs]
...
   [Eden: 1286M(1286M)->0B(1212M)
        Survivors: 78M->152M Heap: 1454M(4096M)->242M(4096M)]
   [Times: user=0.85 sys=0.05, real=0.23 secs]
```

Collection of the young generation took 0.23 seconds of real time, during which the GC threads consumed 0.85 seconds of CPU time. 1,286 MB of objects were moved out of eden (which was adaptively resized to 1,212 MB); 74 MB of that was moved to the survivor space (it increased in size from 78 M to 152 MB) and the rest were freed. We know they were freed by observing that the total heap occupancy decreased by 1,212 MB. In the general case, some objects from the survivor space might have been moved to the old generation, and if the survivor space were full, some objects from eden would have been promoted directly to the old generation—in those cases, the size of the old generation would increase.

The similar log in JDK 11 looks like this:

```
[23.200s][info   ][gc,start     ] GC(10) Pause Young (Normal) (G1 Evacuati
[23.200s][info   ][gc,task      ] GC(10) Using 4 workers of 4 for evacuati
[23.430s][info   ][gc,phases    ] GC(10)   Pre Evacuate Collection Set: 0.
[23.430s][info   ][gc,phases    ] GC(10)   Evacuate Collection Set: 230.3m
[23.430s][info   ][gc,phases    ] GC(10)   Post Evacuate Collection Set: 0
[23.430s][info   ][gc,phases    ] GC(10)   Other: 0.1ms
[23.430s][info   ][gc,heap      ] GC(10) Eden regions: 643->606(606)
[23.430s][info   ][gc,heap      ] GC(10) Survivor regions: 39->76(76)
[23.430s][info   ][gc,heap      ] GC(10) Old regions: 67->75
[23.430s][info   ][gc,heap      ] GC(10) Humongous regions: 0->0
[23.430s][info   ][gc,metaspace ] GC(10) Metaspace: 18407K->18407K(1067008
[23.430s][info   ][gc           ] GC(10) Pause Young (Normal) (G1 Evacuati
                                           1454M(4096M)->242M(4096M) 230.1
[23.430s][info   ][gc,cpu       ] GC(10) User=0.85s Sys=0.05s Real=0.23s
```

A concurrent G1 GC cycle begins and ends as shown in Figure 2-5.

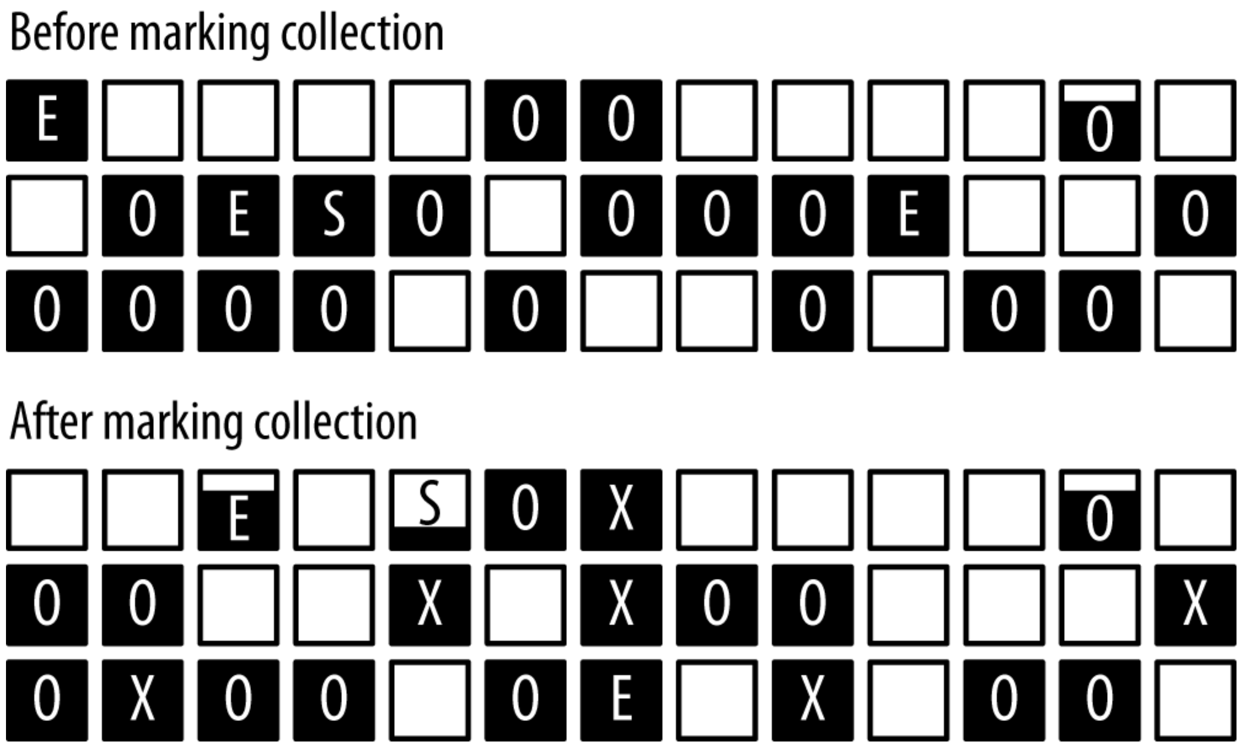


Figure 2-5. Concurrent collection performed by G1 GC

The are three important things to observe in this diagram. First, the young generation has changed its occupancy: there will be at least one (and possibly more) young collections during the concurrent cycle. Hence, the eden regions before the marking cycle have been completely freed, and new eden regions have started to be allocated.

Second, some regions are now marked with an X. Those regions belong to the old generation (and note that they still contain data)—they are regions that the marking cycle has determined contain mostly garbage.

Finally, notice that the old generation (consisting of the regions marked with an O or an X) is actually more occupied after the cycle has completed. That's because the young generation collections that occurred during the marking cycle promoted data into the old generation. In addition, the marking cycle doesn't actually free any data in the old generation: it merely identifies regions that are mostly garbage. Data from those regions is freed in a later cycle.

The G1 GC concurrent cycle has several phases, some of which stop all application threads and some of which do not. The first phase is called initial-mark (in JDK 8) or concurrent start (in JDK 11). That phase stops all application threads—partly because it also executes a young collection, and it sets up the next phases of the cycle.

In JDK 8, that looks like this:

```
50.541: [GC pause (G1 Evacuation pause) (young) (initial-mark), 0.27767100
    ... lots of other data ...
```

```
      [Eden: 1220M(1220M)->0B(1220M)
          Survivors: 144M->144M Heap: 3242M(4096M)->2093M(4096M)]
      [Times: user=1.02 sys=0.04, real=0.28 secs]
```

And in JDK 11:

```
  [50.261s][info   ][gc,start      ] GC(11) Pause Young (Concurrent Start)
                                               (G1 Evacuation Pause)
  [50.261s][info   ][gc,task       ] GC(11) Using 4 workers of 4 for evacuat
  [50.541s][info   ][gc,phases     ] GC(11)   Pre Evacuate Collection Set: 0
  [50.541s][info   ][gc,phases     ] GC(11)   Evacuate Collection Set: 25.9m
  [50.541s][info   ][gc,phases     ] GC(11)   Post Evacuate Collection Set:
  [50.541s][info   ][gc,phases     ] GC(11)   Other: 0.2ms
  [50.541s][info   ][gc,heap       ] GC(11) Eden regions: 1220->0(1220)
  [50.541s][info   ][gc,heap       ] GC(11) Survivor regions: 144->144(144)
  [50.541s][info   ][gc,heap       ] GC(11) Old regions: 1875->1946
  [50.541s][info   ][gc,heap       ] GC(11) Humongous regions: 3->3
  [50.541s][info   ][gc,metaspace  ] GC(11) Metaspace: 52261K->52261K(109977
  [50.541s][info   ][gc            ] GC(11) Pause Young (Concurrent Start)
                                               (G1 Evacuation Pause)
                                               1220M->0B(1220M) 280.055ms
  [50.541s][info   ][gc,cpu        ] GC(11) User=1.02s Sys=0.04s Real=0.28s
```

As in a regular young collection, the application threads were stopped (for 0.28 seconds), and the young generation was emptied (so Eden ends with a size of 0). 71 MB of data was moved from the young generation to the old generation. That's a little difficult to tell in JDK 8 (it is 2093 - 3242 + 1220); the JDK 11 output shows that more clearly.

On the other hand, the JDK 11 output contains references to a few things we haven't discussed yet: first is that the sizes are in regions and not in MB. We'll discuss region sizes later in this chapter, but in this example, the region size is 1 MB. In addition, JDK 11 mentions a new area: humongous regions. That is part of the old generation and is also discussed later in this chapter.

The initial-mark or concurrent start log message announces that the background concurrent cycle has begun. Since the initial mark of the marking cycle phase also requires all application threads to be stopped, G1 GC takes advantage of the young GC cycle to do that work. The impact of adding the initial mark phase to the young GC wasn't that large: it used 20% more CPU cycles than the previous collection, even though the pause was only slightly longer. (Fortunately, there were spare CPU cycles on the machine for the parallel G1 threads, or the pause would have been longer.)

Next, G1 GC scans the root region:

```
50.819: [GC concurrent-root-region-scan-start]
51.408: [GC concurrent-root-region-scan-end, 0.5890230]

[50.819s][info ][gc            ] GC(20) Concurrent Cycle
[50.819s][info ][gc,marking    ] GC(20) Concurrent Clear Claimed Marks
[50.828s][info ][gc,marking    ] GC(20) Concurrent Clear Claimed Marks 0.
[50.828s][info ][gc,marking    ] GC(20) Concurrent Scan Root Regions
[51.408s][info ][gc,marking    ] GC(20) Concurrent Scan Root Regions 589.
```

This takes 0.58 seconds, but it doesn't stop the application threads; it only uses the background threads. However, this phase cannot be interrupted by a young collection, so having available CPU cycles for those background threads is crucial. If the young generation happens to fill up during the root region scanning, the young collection (which has stopped all the application threads) must wait for the root scanning to complete. In effect, this means a longer-than-usual pause to collect the young generation. That situation is shown in the GC log like this:

```
350.994: [GC pause (young)
        351.093: [GC concurrent-root-region-scan-end, 0.6100090]
        351.093: [GC concurrent-mark-start],
        0.37559600 secs]

[350.384s][info][gc,marking   ] GC(50) Concurrent Scan Root Regions
[350.384s][info][gc,marking   ] GC(50) Concurrent Scan Root Regions 610.36
[350.994s][info][gc,marking   ] GC(50) Concurrent Mark (350.994s)
[350.994s][info][gc,marking   ] GC(50) Concurrent Mark From Roots
[350.994s][info][gc,task      ] GC(50) Using 1 workers of 1 for marking
[350.994s][info][gc,start     ] GC(51) Pause Young (Normal) (G1 Evacuation
```

The GC pause here starts before the end of the root region scanning. In JDK 8, the interleaved output in the GC log indicates that the young collection had to pause for the root region scanning to complete before it proceded. In JDK 11, that's a little more difficult to detect: you have to notice that the timestamp of the end of the root region scanning is exactly the same at which the next young collection begins.

In either case, it is impossible to know exactly how long the young collection was delayed. It wasn't necessarily delayed the entire 610 ms in this example; for some period of that time (until the young generation actually filled up), things continued. But in this case, the timestamps show that application threads waited about an extra 100 ms—that is why the duration of the young GC pause is about 100 ms longer than the average duration of other pauses in this log. (If this occurs frequently, it is an indication that G1 GC needs to be better tuned, as discussed in the next section.)

After the root region scanning, G1 GC enters a concurrent marking phase. This happens completely in the background; a message is printed when it starts and ends:

```
111.382: [GC concurrent-mark-start]
....
120.905: [GC concurrent-mark-end, 9.5225160 sec]

[111.382s][info][gc,marking   ] GC(20) Concurrent Mark (111.382s)
[111.382s][info][gc,marking   ] GC(20) Concurrent Mark From Roots
...
[120.905s][info][gc,marking   ] GC(20) Concurrent Mark From Roots 9521.994
[120.910s][info][gc,marking   ] GC(20) Concurrent Preclean
[120.910s][info][gc,marking   ] GC(20) Concurrent Preclean 0.522ms
[120.910s][info][gc,marking   ] GC(20) Concurrent Mark (111.382s, 120.910s
```

Concurrent marking can be interrupted, so young collections may occur during this phase (so there will be lots of GC output where the elipses are).

Also note that in the JDK 11 example, the output here has the same GC entry—20—as did the entry where the root region scanning occurred. We are breaking down the operations more finely than the JDK logging does: in the JDK, the entire background scanning is considered one operation. We're splitting the discussion into more fine-grained, logical operations, since, for example, the root scanning can introduce a pause when the concurrent marking cannot.

The marking phase is followed by a remarking phase and a normal cleanup phase:

```
120.910: [GC remark 120.959:
        [GC ref-PRC, 0.0000890 secs], 0.0718990 secs]
        [Times: user=0.23 sys=0.01, real=0.08 secs]
120.985: [GC cleanup 3510M->3434M(4096M), 0.0111040 secs]
        [Times: user=0.04 sys=0.00, real=0.01 secs]

[120.909s][info][gc,start     ] GC(20) Pause Remark
[120.909s][info][gc,stringtable] GC(20) Cleaned string and symbol table,
                                          strings: 1369 processed, 0 remo
                                          symbols: 17173 processed, 0 rem
[120.985s][info][gc           ] GC(20) Pause Remark 2283M->862M(3666M) 80
[120.985s][info][gc,cpu       ] GC(20) User=0.23s Sys=0.01s Real=0.08s
```

These phases stop the application threads, though usually for a quite short time. Next there is an additional cleanup phase that happens concurrently:

```
120.996: [GC concurrent-cleanup-start]
```

```
120.996: [GC concurrent-cleanup-end, 0.0004520]

[120.878s][info][gc,start       ] GC(20) Pause Cleanup
[120.879s][info][gc             ] GC(20) Pause Cleanup 1313M->1313M(3666M)
[120.879s][info][gc,cpu         ] GC(20) User=0.00s Sys=0.00s Real=0.00s
[120.879s][info][gc,marking     ] GC(20) Concurrent Cleanup for Next Mark
[120.996s][info][gc,marking     ] GC(20) Concurrent Cleanup for Next Mark 1
[120.996s][info][gc             ] GC(20) Concurrent Cycle 70,177.506ms
```

And with that, the normal G1 GC background marking cycle is complete—insofar as finding the garbage goes, at least. But very little has actually been freed yet. A little memory was reclaimed in the cleanup phase, but all G1 GC has really done at this point is to identify old regions that are mostly garbage and can be reclaimed (the ones marked with an X in Figure 2-5).

Now G1 GC executes a series of mixed GCs. They are called mixed because they perform the normal young collection, but they also collect some number of the marked regions from the background scan. The effect of a mixed GC is shown in Figure 2-6.

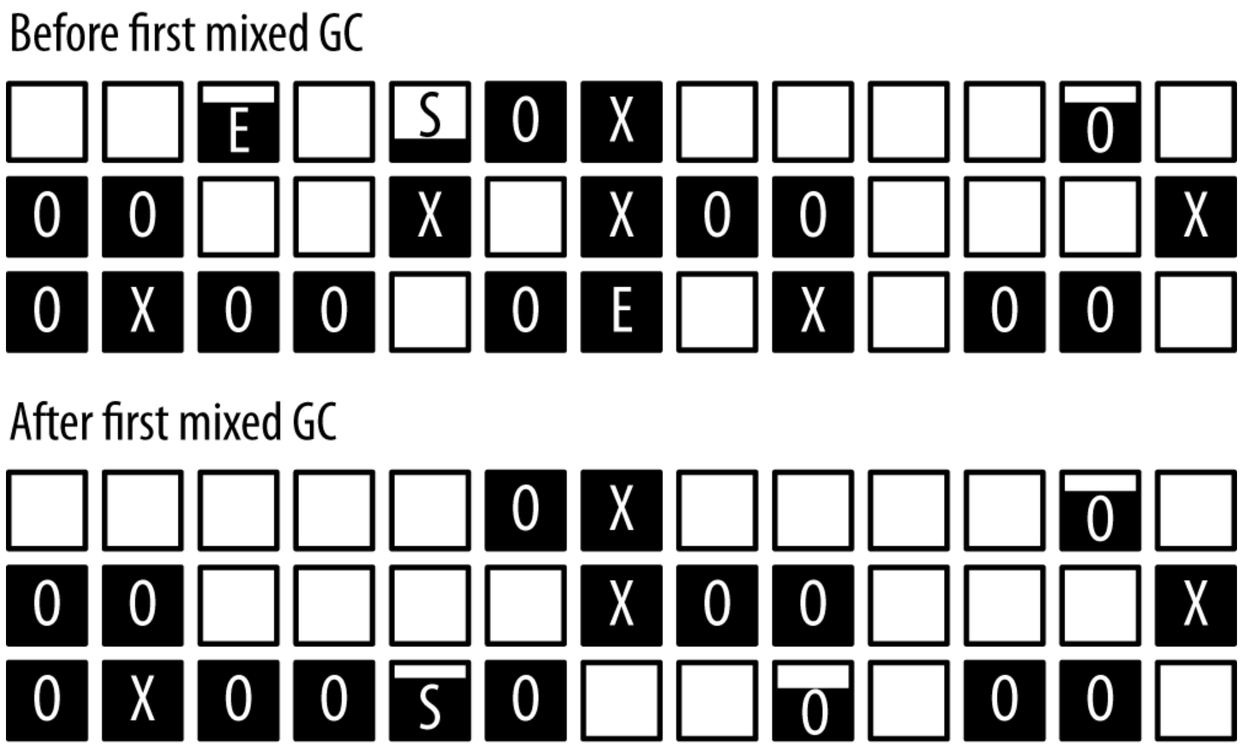


Figure 2-6. Mixed GC performed by G1 GC

As is usual for a young collection, G1 GC has completely emptied eden and adjusted the survivor spaces. Additionally, two of the marked regions have been collected. Those regions were known to contain mostly garbage, and so a large part of them was freed. Any live data in those regions was moved to another region (just as live data was moved from the young generation into regions in the old generation). This is how G1 GC compacts the old generation—moving the objects like this is essentially compacting the heap as G1 GC goes along.

The mixed GC operation looks like this in the log:

```
79.826: [GC pause (mixed), 0.26161600 secs]
....
   [Eden: 1222M(1222M)->0B(1220M)
        Survivors: 142M->144M Heap: 3200M(4096M)->1964M(4096M)]
   [Times: user=1.01 sys=0.00, real=0.26 secs]


[3.800s][info][gc,start       ] GC(24) Pause Young (Mixed) (G1 Evacuation P
[3.800s][info][gc,task        ] GC(24) Using 4 workers of 4 for evacuation
[3.800s][info][gc,phases      ] GC(24)   Pre Evacuate Collection Set: 0.2ms
[3.825s][info][gc,phases      ] GC(24)   Evacuate Collection Set: 250.3ms
[3.826s][info][gc,phases      ] GC(24)   Post Evacuate Collection Set: 0.3m
[3.826s][info][gc,phases      ] GC(24)   Other: 0.4ms
[3.826s][info][gc,heap        ] GC(24) Eden regions: 1222->0(1220)
[3.826s][info][gc,heap        ] GC(24) Survivor regions: 142->144(144)
[3.826s][info][gc,heap        ] GC(24) Old regions: 1834->1820
[3.826s][info][gc,heap        ] GC(24) Humongous regions: 4->4
[3.826s][info][gc,metaspace   ] GC(24) Metaspace: 3750K->3750K(1056768K)
[3.826s][info][gc             ] GC(24) Pause Young (Mixed) (G1 Evacuation P
[3.826s][info][gc,cpu         ] GC(24) User=1.01s Sys=0.00s Real=0.26s
[3.826s][info][gc,start       ] GC(25) Pause Young (Mixed) (G1 Evacuation P
```

Notice that the entire heap usage has been reduced by more than just the 1,222 MB removed from eden. That difference (16 MB) seems small, but remember that some of the survivor space was promoted into the old generation at the same time; in addition, each mixed GC cleans up only a portion of the targeted old generation regions. As we continue, we'll see that it is important to make sure that the mixed GCs clean up enough memory to prevent future concurrent failures.

In JDK 11, the first mixed GC is labelled "Prepared Mixed" and immediately follow the concurrent cleanup.

The mixed GC cycles will continue until (almost) all of the marked regions have been collected, at which point G1 GC will resume regular young GC cycles. Eventually, G1 GC will start another concurrent cycle to determine which regions in the old generation should be freed next.

If all goes well, that's the entire set of GC activies you'll see in your GC log. But there are some failure cases to consider.

There are times when you'll observe a full GC in the log, which is an indication that more tuning (including, possibly, more heap space) will benefit the application performance. There are primarily four times when this is triggered:

Concurrent mode failure

G1 GC starts a marking cycle, but the old generation fills up before the cycle is completed. In that case, G1 GC aborts the marking cycle:

```
51.408: [GC concurrent-mark-start]
65.473: [Full GC 4095M->1395M(4096M), 6.1963770 secs]
 [Times: user=7.87 sys=0.00, real=6.20 secs]
71.669: [GC concurrent-mark-abort]

[51.408][info][gc,marking     ] GC(30) Concurrent Mark From Roots
...
[65.473][info][gc             ] GC(32) Pause Full (G1 Evacuation Pause)
                                           4095M->1305M(4096M) 60,196.37
...
[71.669s][info][gc,marking     ] GC(30) Concurrent Mark From Roots 191m
[71.669s][info][gc,marking     ] GC(30) Concurrent Mark Abort
```

This failure means that heap size should be increased, or the G1 GC background processing must begin sooner, or the cycle must be tuned to run more quickly (e.g., by using additional background threads). Details on how to do that follow.

Promotion failure

G1 GC has completed a marking cycle and has started performing mixed GCs to clean up the old regions. Before it can clean enough space, too many objects are promoted from the young generation, and so the old generation still runs out of space. In the log, a full GC immediately follows a mixed GC:

```
2226.224: [GC pause (mixed)
        2226.440: [SoftReference, 0 refs, 0.0000060 secs]
        2226.441: [WeakReference, 0 refs, 0.0000020 secs]
        2226.441: [FinalReference, 0 refs, 0.0000010 secs]
        2226.441: [PhantomReference, 0 refs, 0.0000010 secs]
        2226.441: [JNI Weak Reference, 0.0000030 secs]
                (to-space exhausted), 0.2390040 secs]
....
    [Eden: 0.0B(400.0M)->0.0B(400.0M)
        Survivors: 0.0B->0.0B Heap: 2006.4M(2048.0M)->2006.4M(2048.0M)]
    [Times: user=1.70 sys=0.04, real=0.26 secs]
2226.510: [Full GC (Allocation Failure)
        2227.519: [SoftReference, 4329 refs, 0.0005520 secs]
        2227.520: [WeakReference, 12646 refs, 0.0010510 secs]
        2227.521: [FinalReference, 7538 refs, 0.0005660 secs]
        2227.521: [PhantomReference, 168 refs, 0.0000120 secs]
        2227.521: [JNI Weak Reference, 0.0000020 secs]
```

```
                   2006M->907M(2048M), 4.1615450 secs]
      [Times: user=6.76 sys=0.01, real=4.16 secs]


   [2226.224s][info][gc            ] GC(26) Pause Young (Mixed) (G1 Evacua
                                                   2048M->2006M(2048M) 26.129m
   ...
   [2226.510s][info][gc,start      ] GC(27) Pause Full (G1 Evacuation Paus
```

This failure means the mixed collections need to happen more quickly; each young collection needs to process more regions in the old generation.

Evacuation failure

When performing a young collection, there isn't enough room in the survivor spaces and the old generation to hold all the surviving objects. This appears in the GC logs as a specific kind of young GC:

```
60.238: [GC pause (young) (to-space overflow), 0.41546900 secs]

[60.238s][info][gc,start       ] GC(28) Pause Young (Concurrent Start)
                                        (G1 Evacuation Pause)
[60.238s][info][gc,task        ] GC(28) Using 4 workers of 4 for evacua
[60.238s][info][gc             ] GC(28) To-space exhausted
```

This is an indication that the heap is largely full or fragmented. G1 GC will attempt to compensate for this, but you can expect this to end badly: the JVM will resort to performing a full GC. The easy way to overcome this is to increase the heap size, though some possible solutions are given in "Advanced Tunings".

Humongous allocation failure

Applications that allocate very large objects can trigger another kind of full GC in G1 GC; see "G1 GC allocation of humongous objects" for more details on this (including how to avoid it). For a long time, it wasn't possible to diagnose this situation without resorting to special logging parameters, but now that is shown with this log:

```
[3023.091s][info][gc,start     ] GC(54) Pause Full (G1 Humongous Allocatio
```

Metadata GC Threshold

As we’ve mentioned, the metaspace is essentially a separate heap and is collected independently of the main heap. It is not collected via G1 GC, but still when it needs to be collected in JDK 8, G1 GC will perform a full GC (immediately preceded by a young collection) on the main heap:

```
0.0535: [GC (Metadata GC Threshold) [PSYoungGen: 34113K->20388K(291328K)]
    73838K->60121K(794112K), 0.0282912 secs]
    [Times: user=0.05 sys=0.01, real=0.03 secs]
0.0566: [Full GC (Metadata GC Threshold) [PSYoungGen: 20388K->0K(291328K)]
    [ParOldGen: 39732K->46178K(584192K)] 60121K->46178K(875520K),
    [Metaspace: 59040K->59036K(1101824K)], 0.1121237 secs]
    [Times: user=0.28 sys=0.01, real=0.11 secs]
```

In JDK 11, the metaspace can be collect/resized without requiring a full GC.

QUICK SUMMARY

1. G1 has a number of cycles (and phases within the concurrent cycle). A well-tuned JVM running G1 should only experience young, mixed, and concurrent GC cycles.

2. Small pauses occur for some of the G1 concurrent phases.

3. G1 should be tuned if necessary to avoid full GC cycles.

Tuning G1 GC

The major goal in tuning G1 GC is to make sure that there are no concurrent mode or evacuation failures that end up requiring a full GC. The techniques used to prevent a full GC can also be used when there are frequent young GCs that must wait for a root region scan to complete.

Secondarily, tuning can minimize the pauses that occur along the way.

These are the options to prevent a full GC:

- Increase the size of the old generation either by increasing the heap space overall or by adjusting the ratio between the generations.

- Increase the number of background threads (assuming there is sufficient CPU).

- Perform G1 GC background activities more frequently.
- Increase the amount of work done in mixed GC cycles.

There are a lot of tunings that can be applied here, but one of the goals of G1 GC is that it shouldn't have to be tuned that much. To that end, G1 GC is primarily tuned via a single flag: the same `-XX:MaxGCPauseMillis=N` flag that was used to tune the throughput collector.

When used with G1 GC (and unlike the throughput collector), that flag does have a default value: 200 ms. If pauses for any of the stop-the-world phases of G1 GC start to exceed that value, G1 GC will attempt to compensate—adjusting the young-to-old ratio, adjusting the heap size, starting the background processing sooner, changing the tenuring threshold, and (most significantly) processing more or fewer old generation regions during a mixed GC cycle.

Some trade-off applies here: if that value is reduced, the young size will contract to meet the pause time goal, but more frequent young GCs will be performed. In addition, the number of old generation regions that can be collected during a mixed GC will decrease to meet the pause time goal, which increases the chances of a concurrent mode failure.

If setting a pause time goal does not prevent the full GCs from happening, these various aspects can be tuned individually. Tuning the heap sizes for G1 is GC accomplished in the same way as for other GC algorithms.

TUNING THE G1 BACKGROUND THREADS

You can consider the concurrent marking of G1 GC to be in a race with the application threads: G1 GC must clear out the old generation faster than the application is promoting new data into it. To have make that happen, try increasing the number of background marking threads (assuming there is sufficient CPU available on the machine).

There are two sets of threads used by G1 GC. The first set is controlled via the `-XX:ParallelGCThreads=N` flag that we first saw in Chapter 1. That value affects the number of threads used for phases when application threads are stopped: young and mixed collections, and the phases of the concurrent remark cycle where threads must be stopped. The second flag is `-XX:ConcGCThreads=N`, which affects the number of threads used for the concurrent remarking.

The default value for the `ConcGCThreads` flag is defined as:

```
ConcGCThreads = (ParallelGCThreads + 2) / 4
```

The division here is integer-based, so there will be one background scanning thread for up to five parallel threads, two background scanning threads for between six and nine parallel threads, and so on.

Increasing the number of background scanning threads will make the concurrent cycle shorter, which should make it easier for G1 GC to finish freeing the old generation during the mixed GC cycles before other threads have filled it again. As always, this assumes that the CPU cycles are available; otherwise, the scanning threads will take CPU away from the application and effectively introduce pauses in it, as we saw when we compared the serial collector to G1 GC in Chapter 1.

TUNING G1 GC TO RUN MORE (OR LESS) FREQUENTLY

G1 GC can also win its race if it starts the background marking cycle earlier. That cycle begins when the heap hits the occupancy ratio specified by `-XX:InitiatingHeapOccupancyPercent=`*`N`*, which has a default value of 45. This percentage refers to the entire heap, not just the old generation.

The `InitiatingHeapOccupancyPercent` value is constant; G1 GC never changes that number as it attempts to meet its pause time goals. If that value is set too high, the application will end up performing full GCs because the concurrent phases don't have enough time to complete before the rest of the heap fills up. If that value is too small, the application will perform more background GC processing than it might otherwise.

At some point, of course, those background threads will have to run, so presumably the hardware has enough CPU to accomodate them. Still, there can be a significant penalty by running them too frequently, because there will be more of the small pauses for those concurrent phases that stop the application threads. Those pauses can add up quickly, so performing background sweeping too frequently for G1 GC should be avoided. Check the size of the heap after a concurrent cycle, and make sure that the `InitiatingHeapOccupancyPercent` value is set higher than that.

TUNING G1 GC MIXED GC CYCLES

After a concurrent cycle, G1 GC cannot begin a new concurrent cycle until all previously marked regions in the old generation have been collected. So another way to make G1 GC start a marking cycle earlier is to process more regions in a mixed GC cycle (so that there will end up being fewer mixed GC cycles).

The amount of work a mixed GC does is dependent on three factors. The first is how many regions were found to be mostly garbage in the first place. There is no way to directly affect that: a region is declared eligible for collection during a mixed GC if it is 85% garbage. (There is an experimental name for tuning that parameter, which is `-XX:G1MixedGCLiveThresholdPercent=N`.)

The second factor is the maximum number of mixed GC cycles over which G1 GC will process those regions, which is specified by the value of the flag `-XX:G1MixedGCCountTarget=N`. The default value for that is 8; reducing that value can help overcome promotion failures (at the expense of longer pause times during the mixed GC cycle).

On the other hand, if mixed GC pause times are too long, that value can be increased so that less work is done during the mixed GC. Just be sure that increasing that number does not delay the next G1 GC concurrent cycle too long, or a concurrent mode failure may result.

Finally, the third factor is the maximum desired length of a GC pause (i.e., the value specified by `MaxGCPauseMillis`). The number of mixed cycles specified by the `G1MixedGCCountTarget` flag is an upper bound; if time is available within the pause target, G1 GC will collect more than one-eighth (or whatever value has been specified) of the marked old generation regions. Increasing the value of the `MaxGCPauseMillis` flag allows more old generation regions to be collected during each mixed GC, which in turn can allow G1 GC to begin the next concurrent cycle sooner.

QUICK SUMMARY

1. G1 GC tuning should begin by setting a reasonable pause time target.
2. If full GCs are still an issue after that and the heap size cannot be increased, specific tunings can be applied for specific failures.
    a. To make the background threads run more frequently, adjust the `InitiatingHeapOccupancyPercent`.
    b. If additional CPU is available, adjust the number of threads via the `ConcGCThreads` flag.
    c. To prevent promotion failures, decrease the size of the `G1MixedGCCountTarget`.

Understanding the CMS Collector

Although the CMS collector is deprecated, it is still available in current JDK builds. So we'll discuss how to tune it in this section, as well as explaining why it has been deprecated.

CMS has three basic operations:

- CMS collects the young generation (stopping all application threads).
- CMS runs a concurrent cycle to clean data out of the old generation.
- If necessary, CMS performs a full GC to compact the old generation.

A CMS collection of the young generation appears in Figure 2-7.

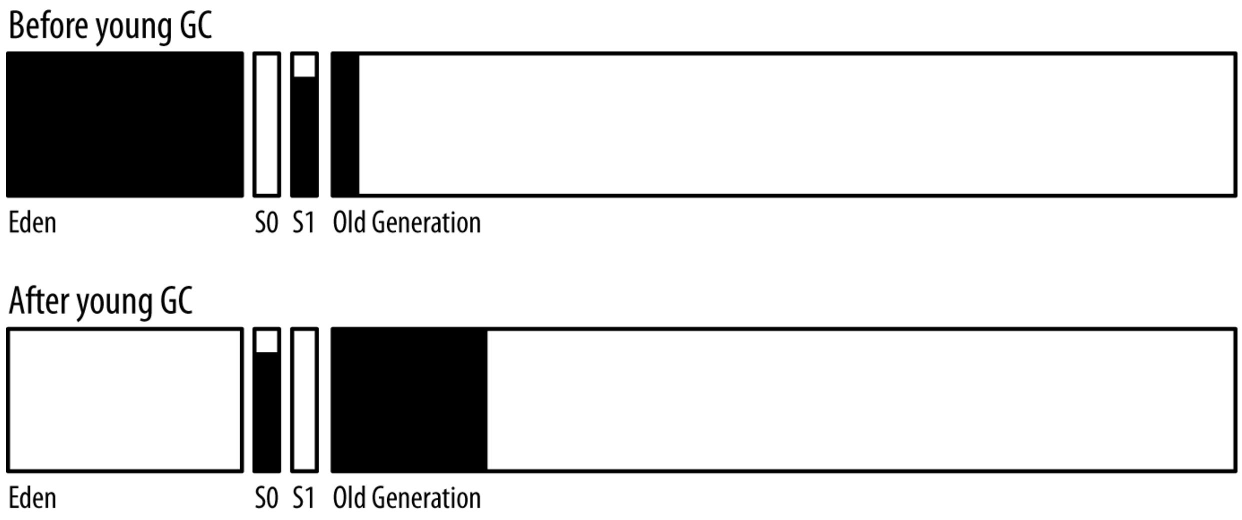


Figure 2-7. Young collection performed by CMS

A CMS young collection is very similar to a throughput young collection: data is moved from eden into one survivor space and into the old generation.

The GC log entry for CMS is also similar (we'll only show the JDK 8 log format):

```
89.853: [GC 89.853: [ParNew: 629120K->69888K(629120K), 0.1218970 secs]
               1303940K->772142K(2027264K), 0.1220090 secs]
               [Times: user=0.42 sys=0.02, real=0.12 secs]
```

The size of the young generation is presently 629 MB; after collection, 69 MB of it remains (in a survivor space). Similarly, the size of the entire heap is 2,027 MB—772 MB of which is occupied after the collection. The entire process took 0.12 seconds, though the parallel GC threads racked up 0.42 seconds in CPU usage.

A concurrent cycle is shown in Figure 2-8.

CMS starts a concurrent cycle based on the occupancy of the heap. When it is sufficiently full, the background threads that cycle through the heap and remove objects are started. At the end of the cycle, the heap looks like the bottom row in this diagram. Notice that the old generation is not compacted: there are areas where objects are allocated, and free areas. When a young collection moves objects from eden into the old generation, the JVM will attempt to use those free areas to hold the objects.

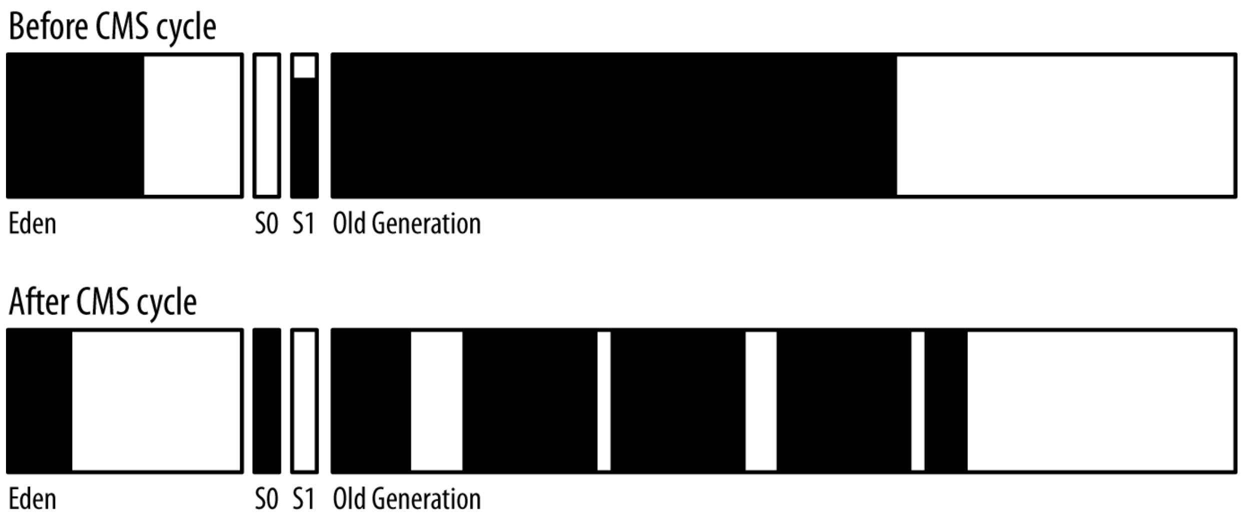


Figure 2-8. Concurrent collection performed by CMS

In the GC log, this cycle appears as a number of phases. Although a majority of the concurrent cycle uses background threads, some phases introduce short pauses where all application threads are stopped.

The concurrent cycle starts with an initial mark phase, which stops all the application threads:

```
89.976: [GC [1 CMS-initial-mark: 702254K(1398144K)]
                772530K(2027264K), 0.0830120 secs]
                [Times: user=0.08 sys=0.00, real=0.08 secs]
```

This phase is responsible for finding all the GC root objects in the heap. The first set of numbers shows that objects currently occupy 702 MB of 1,398 MB of the old generation, while the second set shows that the occupancy of the entire 2,027 MB heap is 772 MB. The application threads were stopped for a period of 0.08 seconds during this phase of the CMS cycle.

The next phase is the mark phase, and it does not stop the application threads. The phase is represented in the GC log by these lines:

```
90.059: [CMS-concurrent-mark-start]
90.887: [CMS-concurrent-mark: 0.823/0.828 secs]
                [Times: user=1.11 sys=0.00, real=0.83 secs]
```

The mark phase took 0.83 seconds (and 1.11 seconds of CPU time). Since it is just a marking phase, it hasn't done anything to the heap occupancy, and so no data is shown about that. If there were data, it would likely show a growth in the heap from objects allocated in the young generation during those 0.83 seconds, since the application threads have continued to execute.

Next comes a preclean phase, which also runs concurrently with the application threads:

```
90.887: [CMS-concurrent-preclean-start]
90.892: [CMS-concurrent-preclean: 0.005/0.005 secs]
                [Times: user=0.01 sys=0.00, real=0.01 secs]
```

The next phase is a remark phase, but it involves several operations:

```
90.892: [CMS-concurrent-abortable-preclean-start]
92.392: [GC 92.393: [ParNew: 629120K->69888K(629120K), 0.1289040 secs]
                1331374K->803967K(2027264K), 0.1290200 secs]
                [Times: user=0.44 sys=0.01, real=0.12 secs]
94.473: [CMS-concurrent-abortable-preclean: 3.451/3.581 secs]
                [Times: user=5.03 sys=0.03, real=3.58 secs]

94.474: [GC[YG occupancy: 466937 K (629120 K)]
        94.474: [Rescan (parallel) , 0.1850000 secs]
        94.659: [weak refs processing, 0.0000370 secs]
        94.659: [scrub string table, 0.0011530 secs]
                [1 CMS-remark: 734079K(1398144K)]
                1201017K(2027264K), 0.1863430 secs]
        [Times: user=0.60 sys=0.01, real=0.18 secs]
```

Wait, didn't CMS just execute a preclean phase? What's up with this abortable preclean phase?

The abortable preclean phase is used because the remark phase (which, strictly speaking, is the final entry in this output) is not concurrent—it will stop all the application threads. CMS wants to avoid the situation where a young generation collection occurs and is immediately followed by a remark phase, in which case the application threads would be stopped for two back-to-back pause operations. The goal here is to minimize pause lengths by preventing back-to-back pauses.

Hence, the abortable preclean phase waits until the young generation is about 50% full. In theory, that is halfway between young generation collections, giving CMS the best chance to avoid those back-to-back pauses. In this example, the abortable preclean phase starts at 90.8 seconds and waits about 1.5 seconds for the regular young collection to occur (at 92.392 seconds into the log). CMS uses past behavior to calculate when the next young collection is likely to occur—in this case, CMS calculated it would occur in about 4.2 seconds. So after 2.1 seconds (at 94.4 seconds), CMS ends the preclean phase (which it calls “aborting” the phase, even though that is the only way the phase is stopped). Then, finally, CMS executes the remark phase, which pauses the application threads for 0.18 seconds (the application threads were not paused during the abortable preclean phase).

Next comes another concurrent phase—the sweep phase:

```
94.661: [CMS-concurrent-sweep-start]
95.223: [GC 95.223: [ParNew: 629120K->69888K(629120K), 0.1322530 secs]
                999428K->472094K(2027264K), 0.1323690 secs]
                [Times: user=0.43 sys=0.00, real=0.13 secs]
95.474: [CMS-concurrent-sweep: 0.680/0.813 secs]
                [Times: user=1.45 sys=0.00, real=0.82 secs]
```

This phase took 0.82 seconds and ran concurrently with the application threads. It also happened to be interrupted by a young collection. This young collection had nothing to do with the sweep phase, but it is left in here as an example that the young collections can occur simultaneously with the old collection phases. In Figure 2-8, notice that the state of the young generation changed during the concurrent collection—there may have been an arbitrary number of young collections during the sweep phase (and there will have been at least one young collection because of the abortable preclean phase).

Next comes the concurrent reset phase:

```
95.474: [CMS-concurrent-reset-start]
95.479: [CMS-concurrent-reset: 0.005/0.005 secs]
        [Times: user=0.00 sys=0.00, real=0.00 secs]
```

That is the last of the concurrent phases; the CMS cycle is now complete, and the unreferenced objects found in the old generation are now free (resulting in the heap shown in Figure 2-8). Unfortunately, the log doesn’t provide any information about how many objects were freed; the reset line doesn’t give any information about the heap occupancy. To get an idea of that, look to the next young collection, which is:

```
98.049: [GC 98.049: [ParNew: 629120K->69888K(629120K), 0.1487040 secs]
                1031326K->504955K(2027264K), 0.1488730 secs]
```

Now compare the occupancy of the old generation at 89.853 seconds (before the CMS cycle began), which was roughly 703 MB (the entire heap occupied 772 MB at that point, which included 69 MB in the survivor space, so the old generation consumed the remaining 703 MB).

In the collection at 98.049 seconds, the old generation occupies about 504 MB; the CMS cycle therefore cleaned up about 199 MB of memory.

If all goes well, these are the only cycles that CMS will run and the only log messages that will appear in the CMS GC log. But there are three more messages to look for, which indicate that CMS ran into a problem. The first is a concurrent mode failure:

```
267.006: [GC 267.006: [ParNew: 629120K->629120K(629120K), 0.0000200 secs]
        267.006: [CMS267.350: [CMS-concurrent-mark: 2.683/2.804 secs]
        [Times: user=4.81 sys=0.02, real=2.80 secs]
        (concurrent mode failure):
        1378132K->1366755K(1398144K), 5.6213320 secs]
        2007252K->1366755K(2027264K),
        [CMS Perm : 57231K->57222K(95548K)], 5.6215150 secs]
        [Times: user=5.63 sys=0.00, real=5.62 secs]
```

When a young collection occurs and there isn’t enough room in the old generation to hold all the objects that are expected to be promoted, CMS executes what is essentially a full GC. All application threads are stopped, and the old generation is cleaned of any dead objects, reducing its occupancy to 1,366 MB—an operation which kept the application threads paused for a full 5.6 seconds. That operation is single-threaded, which is one reason it takes so long (and one reason why concurrent mode failures are worse as the heap grows).

This concurrent mode failure is a major reason why CMS is deprecated. G1 GC can have a concurrent mode failure, but when it reverts to a full GC, that full GC occurs in parallel (like a throughput full GC). A CMS full GC will take many times longer to execute since it must execute in a single thread.

The second problem occurs when there is enough room in the old generation to hold the promoted objects, but the free space is fragmented and so the promotion fails:

```
6043.903: [GC 6043.903:
        [ParNew (promotion failed): 614254K->629120K(629120K), 0.1619839 s
        6044.217: [CMS: 1342523K->1336533K(2027264K), 30.7884210 secs]
        2004251K->1336533K(1398144K),
        [CMS Perm : 57231K->57231K(95548K)], 28.1361340 secs]
        [Times: user=28.13 sys=0.38, real=28.13 secs]
```

Here, CMS started a young collection and assumed that there was enough free space to hold all the promoted objects (otherwise, it would have declared a concurrent mode failure). That assumption proved incorrect: CMS couldn’t promote the objects because the old generation was fragmented (or, much less likely, because the amount of memory to be promoted was bigger than CMS expected).

As a result, in the middle of the young collection (when all threads were already stopped), CMS collected and compacted the entire old generation. The good news is that with the heap compacted, fragmentation issues have been solved (at least for a while). But that came with a hefty 28-second pause time. This time is much longer than when CMS had a concurrent mode failure because the entire heap was compacted; the concurrent mode failure simply freed objects in the heap. The heap at this point appears as it did at the end of the throughput collector's full GC (Figure 2-2): the young generation is completely empty, and the old generation has been compacted.

Finally, the CMS log may show a full GC without any of the usual concurrent GC messages:

```
279.803: [Full GC 279.803:
                [CMS: 88569K->68870K(1398144K), 0.6714090 secs]
                558070K->68870K(2027264K),
                [CMS Perm : 81919K->77654K(81920K)],
                0.6716570 secs]
```

This occurs when the metaspace has filled up and needs to be collected. CMS does not collect permgen (or the metaspace), so if it fills up, a full GC is needed to discard any unreferenced classes. The advanced tuning section for CMS shows how to overcome this issue.

QUICK SUMMARY

1. CMS has several GC operations, but the expected operations are minor GCs and concurrent cycles.

2. Concurrent mode failures and promotion failures in CMS are quite expensive; CMS should be tuned to avoid these as much as possible.

3. By default, CMS does not collect permgen.

Tuning to Solve Concurrent Mode Failures

The primary concern when tuning CMS is to make sure that there are no concurrent mode or promotion failures. As the CMS GC log showed, a concurrent mode failure occurs because CMS did not clean out the old generation fast enough: when it comes time to perform a collection in the young generation, CMS calculates that it doesn't have enough room to promote those objects to the old generation and instead collects the old generation first.

The old generation initially fills up by placing the objects right next to each other. When some amount of the old generation is filled (by default, 70%), the concurrent cycle begins and the background CMS thread(s) start scanning the old generation for garbage. At this point, a race is on: CMS must complete scanning the old generation and freeing objects before the remainder (30%) of the old generation fills up. If the concurrent cycle loses the race, CMS will experience a concurrent mode failure.

There are multiple ways to attempt to avoid this failure:

- Make the old generation larger, either by shifting the proportion of the new generation to the old generation, or by adding more heap space altogether.
- Run the background thread more often.
- Use more background threads.

ADAPTIVE SIZING AND CMS

CMS uses the `MaxGCPauseMllis=`*`N`* and `GCTimeRatio=`*`N`* settings to determine how large the heap and the generations should be.

One significant difference in the approach CMS takes is that the young generation is never resized unless a full GC occurs. Since the goal of CMS is to never have a full collection, this means a well-tuned CMS application will never resize its young generation.

Concurrent mode failures can be frequent during program startup, as CMS adaptively sizes the heap and the metaspace. It can be a good idea to start CMS with a larger initial heap size (and larger metaspace), which is a special case of making the heap larger to prevent those failures.

If more memory is available, the better solution is to increase the size of the heap. Otherwise, change the way the background threads operate.

RUNNING THE BACKGROUND THREAD MORE OFTEN

One way to let CMS win the race is to start the concurrent cycle sooner. If the concurrent cycle starts when 60% of the old generation is filled, CMS has a better chance of finishing than if the cycle starts when 70% of the old generation is filled. The easiest way to achieve that is to set both these flags: `-XX:CMSInitiatingOccupancyFraction=`*`N`* and -

`XX:+UseCMSInitiatingOccupancyOnly`. Using both those flags also makes CMS easier to understand: if they are both set, then CMS determines when to start the background thread based only on the percentage of the old generation that is filled. (Note that unlike G1 GC, the occupancy ratio here is only the old generation and not the entire heap.)

By default, the `UseCMSInitiatingOccupancyOnly` flag is `false`, and CMS uses a more complex algorithm to determine when to start the background thread. If the background thread needs to be started earlier, better to start it the simplest way possible and set the `UseCMSInitiatingOccupancyOnly` flag to `true`.

Tuning the value of the `CMSInitiatingOccupancyFraction` may require a few iterations. If `UseCMSInitiatingOccupancyOnly` is enabled, then the default value for `CMSInitiatingOccupancyFraction` is 70: the CMS cycle starts when the old generation is 70% occupied.

A better value for that flag for a given application can be found in the GC log by figuring out when the failed CMS cycle started in the first place. Find the concurrent mode failure in the log, and then look back to when the most recent CMS cycle started. The CMS-initial-mark line will show how full the old generation was when the CMS cycle started:

```
89.976: [GC [1 CMS-initial-mark: 702254K(1398144K)]
                772530K(2027264K), 0.0830120 secs]
                [Times: user=0.08 sys=0.00, real=0.08 secs]
```

In this example, that works out to about 50% (702 MB out of 1,398 MB). That was not soon enough, so the `CMSInitiatingOccupancyFraction` needs to be set to something lower than 50. (Although the default value for that flag is 70, this example started the CMS threads when the old generation was 50% full because the `UseCMSInitiatingOccupancyOnly` flag was not set.)

The temptation here is just to set the value to 0 or some other small number so that the background CMS cycle runs all the time. That’s usually discouraged, but as long as you are aware of the trade-offs being made, it may work out fine.

The first trade-off comes in CPU time: the CMS background thread(s) will run continually, and they consume a fair amount of CPU—each background CMS thread will consume 100% of a CPU. There will also be very short bursts when multiple CMS threads run and the total CPU on the box spikes as a result. If these threads are running needlessly, that wastes CPU resources.

On the other hand, it isn’t necessarily a problem to use those CPU cycles. The background CMS threads have to run sometimes, even in the best case. Hence, the machine must always have enough CPU cycles available to run those CMS threads. So when sizing the machine, you must plan for that CPU usage.

The second trade-off is far more significant and has to do with pauses. As the GC log showed, certain phases of the CMS cycle stop all the application threads. The main reason CMS is used is to limit the effect of GC pauses, so running CMS more often than needed is counterproductive. The CMS pauses are generally much shorter than a young generation pause, and a particular application may not be sensitive to those additional pauses—it's a trade-off between the additional pauses and the reduced chance of a concurrent mode failure. But continually running the background GC pauses will likely lead to excessive overall pauses, which will in the end ultimately reduce the performance of the application.

Unless those trade-offs are acceptable, take care not to set the `CMSInitiatingOccupancyFraction` higher than the amount of live data in the heap, plus at least 10% to 20%.

ADJUSTING THE CMS BACKGROUND THREADS

Each CMS background thread will consume 100% of a CPU on a machine. If an application experiences a concurrent mode failure and there are extra CPU cycles available, the number of those background threads can be increased by setting the `-XX:ConcGCThreads=`*`N`* flag. CMS sets this flag differently than G1 GC; it uses this calculation:

```
ConcGCThreads = (3 + ParallelGCThreads) / 4
```

So CMS increases the value of `ConcGCThreads` one step earlier than does G1 GC.

QUICK SUMMARY

1. Avoiding concurrent mode failures is the key to achieving the best possible performance with CMS.
2. The simplest way to avoid those failures (when possible) is to increase the size of the heap.
3. Otherwise, the next step is to start the concurrent background threads sooner by adjusting the `CMSInitiatingOccupancyFraction`.
4. Tuning the number of background threads can also help.

Advanced Tunings

This section on tunings covers some fairly unusual situations. Even if those situations are not encountered frequently, many of the low-level details of the GC algorithms are explained in this section.

Tenuring and Survivor Spaces

When the young generation is collected, some objects will still be alive. This includes newly created objects that are destined to exist for a long time, but it also includes some objects that are otherwise short-lived. Consider the loop of `BigDecimal` calculations at the beginning of Chapter 1. If the JVM performs GC in the middle of that loop, some of those very-short-lived `BigDecimal` objects will be quite unlucky: they will have been just created and in use, so they can't be freed—but they aren't going to live long enough to justify moving them to the old generation.

This is the reason that the young generation is divided into two survivor spaces and eden. This setup allows objects to have some additional chances to be collected while still in the young generation, rather than being promoted into (and filling up) the old generation.

When the young generation is collected and the JVM finds an object that is still alive, that object is moved to a survivor space rather than to the old generation. During the first young generation collection, objects are moved from eden into survivor space 0. During the next collection, live objects are moved from both survivor space 0 and from eden into survivor space 1. At that point, eden and survivor space 0 are completely empty. The next collection moves live objects from survivor space 1 and eden into survivor space 0, and so on. (The survivor spaces are also referred to as the “to” and “from” spaces; during each collection, objects are moved out of the “from” space into the “to” space. “from” and “to” are simply pointers that switch between the two survivor spaces on every collection.)

Clearly this cannot continue forever, or nothing would ever be moved into the old generation. Objects are moved into the old generation in two circumstances. First, the survivor spaces are fairly small. When the target survivor space fills up during a young collection, any remaining live objects in eden are moved directly into the old generation. Second, there is a limit to the number of GC cycles during which an object can remain in the survivor spaces. That limit is called the tenuring threshold.

There are tunings to affect each of those situations. The survivor spaces take up part of the allocation for the young generation, and like other areas of the heap, the JVM sizes them dynamically. The initial size of the survivor spaces is determined by the `-XX:InitialSurvivorRatio=`*`N`* flag, which is used in this equation:

```
survivor_space_size = new_size / (initial_survivor_ratio + 2)
```

For the default initial survivor ratio of 8, each survivor space will occupy 10% of the young generation.

The JVM may increase the survivor spaces size to a maximum determined by the setting of the `-XX:MinSurvivorRatio=`*`N`* flag. That flag is used in this equation:

```
maximum_survivor_space_size = new_size / (min_survivor_ratio + 2)
```

By default, this value is 3, meaning the maximum size of a survivor space will be 20% of the young generation. Note again that the value is a ratio, so the minimum value of the ratio gives the maximum size of the survivor space. The name is hence a little counterintuitive.

To keep the survivor spaces at a fixed size, set the `SurvivorRatio` to the desired value and disable the `UseAdaptiveSizePolicy` flag (though remember that disabling adaptive sizing will apply to the old and new generations as well).

The JVM determines whether to increase or decrease the size of the survivor spaces (subject to the defined ratios) based on how full a survivor space is after a GC. The survivor spaces will be resized so that they are, by default, 50% full after a GC. That value can be changed with the `-XX:TargetSurvivorRatio=`*`N`* flag.

Finally, there is the question of how many GC cycles an object will remain ping-ponging between the survivor spaces before being moved into the old generation. That answer is determined by the tenuring threshold. The JVM continually calculates what it thinks the best tenuring threshold is. The threshold starts at the value specified by the `-XX:InitialTenuringThreshold=N` flag (the default is 7 for the throughput and G1 GC collectors, and 6 for CMS). The JVM will ultimately determine a threshold between 1 and the value specified by the `-XX:MaxTenuringThreshold=N` flag; for the throughput and G1 GC collectors, the default maximum threshold is 15, and for CMS it is 6.

ALWAYS AND NEVER TENURE

The tenuring threshold will always take on some range between 1 and `MaxTenuringThreshold`. Even if the JVM is started with an initial tenuring threshold equal to the maximum tenuring threshold, the JVM may decrease that value.

There are two flags that can circumvent that behavior at either extreme. If you know that objects that survive a young collection will always be around for a long time, you can specify `-XX:+AlwaysTenure` (by default, `false`), which is essentially the same as setting the `MaxTenuringThreshold` to 0. This is a very, very rare situation; it means that objects will always be promoted to the old generation rather than stored in a survivor space.

The second flag is `-XX:+NeverTenure` (also `false` by default). This flag affects two things: it behaves as if the initial and max tenuring thresholds are infinity, and it prevents the JVM from adjusting that threshold down. In other words, as long as there is room in the survivor space, no object will ever be promoted to the old generation.

Given all that, which values might be tuned under which circumstances? It is helpful to look at the tenuring statistics; these are not printed using the GC logging commands we've used so far.

In JDK 8, the tenuring distribution can be added to the GC log by including the flag `-XX:+PrintTenuringDistribution` (which is `false` by default). In JDK 11, it is added by including `age*=debug` or `age*=trace` to the `Xlog` argument.

The most important thing to look for is whether the survivor spaces are so small that during a minor GC, objects are promoted directly from eden into the old generation. The reason to avoid that is short-lived objects will end up filling the old generation, causing full GCs to occur too frequently.

In GC logs taken with the throughput collector, the only hint for that condition is this line:

```
Desired survivor size 39059456 bytes, new threshold 1 (max 15)
         [PSYoungGen: 657856K->35712K(660864K)]
```

```
            1659879K->1073807K(2059008K), 0.0950040 secs]
            [Times: user=0.32 sys=0.00, real=0.09 secs]
```

The JDK 11 log with `age*=debug` is similar; it will print the desired survivor size during the collection.

The desired size for a single survivor space here is 39 MB out of a young generation of 660 MB: the JVM has calculated that the two survivor spaces should take up about 11% of the young generation. But the open question is whether that is large enough to prevent overflow. There is no definitive answer from this log, but the fact that the JVM has adjusted the tenuring threshold to 1 indicates that it has determined it is directly promoting most objects to the old generation anyway, so it has minimized the tenuring threshold. This application is probably promoting directly to the old generation without fully using the survivor spaces.

When G1 GC is used, more informative output is obtained in the JDK 8 log:

```
Desired survivor size 35782656 bytes, new threshold 2 (max 6)
- age   1:   33291392 bytes,   33291392 total
- age   2:    4098176 bytes,   37389568 total
```

In JDK 11, that information comes by including `age*=trace` in the logging configuration.

The desired survivor space is similar to the last example—35 MB—but the output also shows the size of all the objects in the survivor space. With 37 MB of data to promote, the survivor space is indeed overflowing.

Whether or not this situation can be improved upon is very dependent on the application. If the objects are going to live longer than a few more GC cycles, they will eventually end up in the old generation anyway, so adjusting the survivor spaces and tenuring threshold won't really help. But if the objects would go away after just a few more GC cycles, then some performance can be gained by arranging for the survivor spaces to be more efficient.

If the size of the survivor spaces is increased (by decreasing the survivor ratio), then memory is taken away from the eden section of the young generation. That is where the objects actually are allocated, meaning fewer objects can be allocated before incurring a minor GC. So that option is usually not recommended.

Another possibility is to increase the size of the young generation. That can be counterproductive in this situation: objects might be promoted less often into the old generation, but since the old generation is smaller, the application may do full GCs more often.

If the size of the heap can be increased altogether, then both the young generation and the survivor spaces can get more memory, which will be the best solution. A good process is to increase the heap size (or at least the young generation size) and to decrease the survivor ratio. That will increase the size of the survivor spaces more than it will increase the size of eden. The application should end up having roughly the same number of young collections as before. It should have fewer full GCs, though, since fewer objects will be promoted into the old generation (again, assuming that the objects will no longer be live after a few more GC cycles).

If the sizes of the survivor spaces have been adjusted so that they never overflow, then objects will only be promoted to the old generation after the `MaxTenuringThreshold` is reached. That value can be increased to keep the objects in the survivor spaces for a few more young GC cycles. But be aware that if the tenuring threshold is increased and objects stay in the survivor space longer, there will be less room in the survivor space during future young collections: it is then more likely that the survivor space will overflow and start promoting directly into the old generation again.

QUICK SUMMARY

1. Survivor spaces are designed to allow objects (particularly just-allocated objects) to remain in the young generation for a few GC cycles. This increases the probability the object will be freed before it is promoted to the old generation.

2. If the survivor spaces are too small, objects will promoted directly into the old generation, which in turn causes more old GC cycles.

3. The best way to handle that situation is to increase the size of the heap (or at least the young generation) and allow the JVM to handle the survivor spaces.

4. In rare cases, adjusting the tenuring threshold or survivor space sizes can prevent promotion of objects into the old generation.

Allocating Large Objects

This section describes in detail how the JVM allocates objects. This is interesting background information, and it is important to applications that frequently create a significant number of large objects. In this context, “large” is a relative term; it depends, as we’ll see, on the size of a

TLAB within the JVM.

TLAB sizing is a consideration for all GC algorithms, and G1 GC has an additional consideration for very large objects (again, a relative term—but for a 2 GB heap, objects larger than 512 MB). The effects of very large objects on G1 GC can be very important—TLAB sizing (to overcome somewhat large objects when using any collector) is fairly unusual, but G1 GC region sizing (to overcome very large objects when using G1) is more common.

THREAD-LOCAL ALLOCATION BUFFERS

Chapter 1 discusses how objects are allocated within eden; this allows for faster allocation (particularly for objects that are quickly discarded).

It turns out that one reason allocation in eden is so fast is that each thread has a dedicated region where it allocates objects—a thread-local allocation buffer or TLAB. When objects are allocated directly in a shared space such as eden, some synchronization is required to manage the free-space pointers within that space. By setting up each thread with its own dedicated allocation area, the thread needn't perform any synchronization when allocating objects. (This is a variation of how thread-local variables can prevent lock contention.)

Usually, the use of TLABs is transparent to developers and end users: TLABs are enabled by default, and the JVM manages their sizes and how they are used. The important thing to realize about TLABs is that they have a small size, so large objects cannot be allocated within a TLAB. Large objects must be allocated directly from the heap, which requires extra time because of the synchronization.

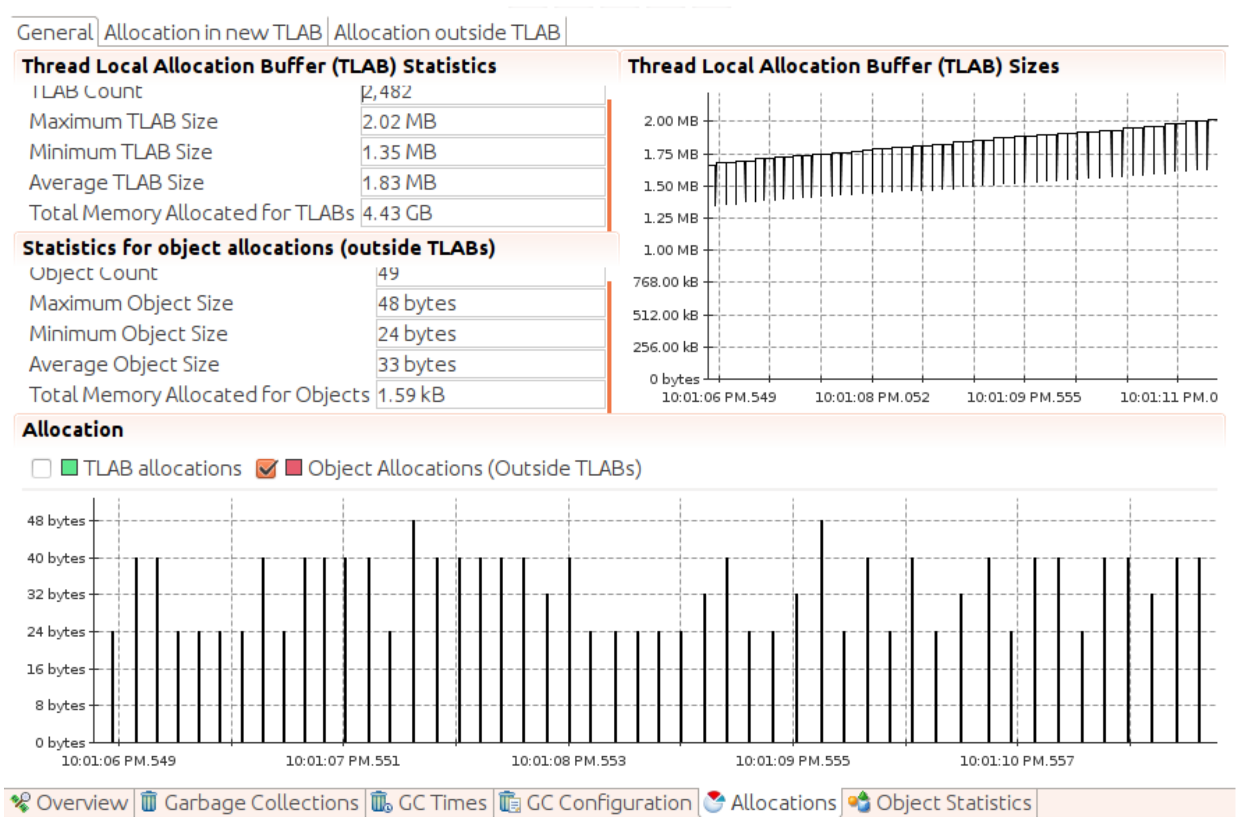
As a TLAB becomes full, objects of a certain size can no longer be allocated in it. At this point, the JVM has a choice. One option is to "retire" the TLAB and allocate a new one for the thread. Since the TLAB is just a section within eden, the retired TLAB will be cleaned at the next young collection and can be reused subsequently. Alternately, the JVM can allocate the object directly on the heap and keep the existing TLAB (at least until the thread allocates additional objects into the TLAB). Consider the case where a TLAB is 100 KB, and 75 KB has already been allocated. If a new 30 KB allocation is needed, the TLAB can be retired, which wastes 25 KB of eden space. Or the 30 KB object can be allocated directly from the heap, and the thread can hope that the next object that is allocated will fit in the 25 KB of space that is still free within the TLAB.

There are parameters to control this (which are discussed later in this section), but the key is that the size of the TLAB is crucial. By default, the size of a TLAB is based on three things: the number of threads in the application, the size of eden, and the allocation rate of threads.

Hence two types of applications may benefit from tuning the TLAB parameters: applications that allocate a lot of large objects, and applications that have a relatively large number of threads compared to the size of eden. By default, TLABs are enabled; they can be disabled by specifying `-XX:-UseTLAB`, although they give such a performance boost that disabling them is always a bad idea.

Since the calculation of the TLAB size is based in part on the allocation rate of the threads, it is impossible to definitively predict the best TLAB size for an application. What can be done instead is to monitor the TLAB allocation to see if any allocations occur outside of the TLABs. If a significant number of allocations occur outside of TLABs, then there are two choices: reduce the size of the object being allocated, or adjust the TLAB sizing parameters.

Monitoring the TLAB allocation is another case where Java Flight Recorder is much more powerful than other tools. Figure 2-9 shows a sample of the TLAB allocation screen from a JFR recording.



Thread Local Allocation Buffer (TLAB) Statistics

TLAB Count	2,482
Maximum TLAB Size	2.02 MB
Minimum TLAB Size	1.35 MB
Average TLAB Size	1.83 MB
Total Memory Allocated for TLABs	4.43 GB

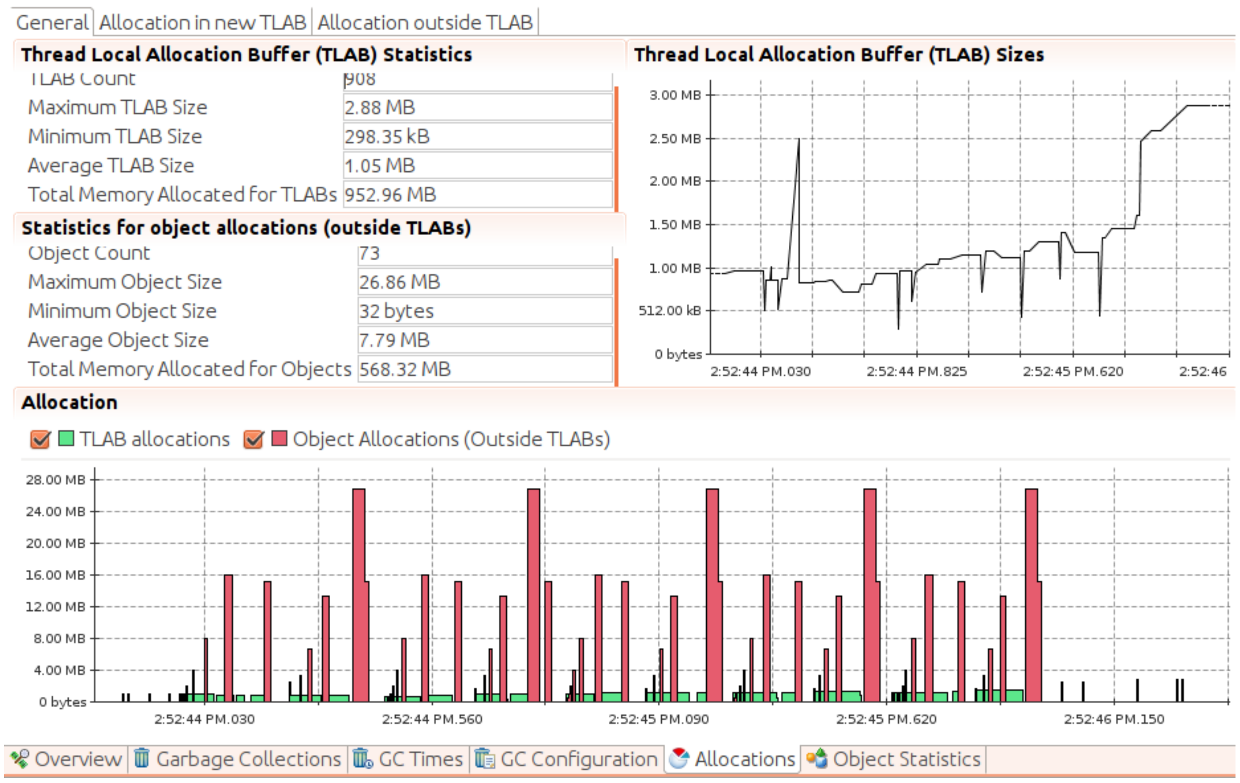
Statistics for object allocations (outside TLABs)

Object Count	49
Maximum Object Size	48 bytes
Minimum Object Size	24 bytes
Average Object Size	33 bytes
Total Memory Allocated for Objects	1.59 kB

Figure 2-9. View of TLABs in Java Flight Recorder

In the 5 seconds selected in this recording, 49 objects were allocated outside of TLABs; the maximum size of those objects was 48 bytes. Since the minimum TLAB size is 1.35 MB, we know that these objects were allocated on the heap only because the TLAB was full at the time of allocation: they were not allocated directly in the heap because of their size. That is typical immediately before a young GC occurs (as eden—and hence the TLABs carved out of eden—becomes full).

The total allocation in this period is 1.59 KB; neither the number of allocations nor the size in this example are a cause for concern. There will always be some object allocated outside of TLABs, particularly as eden approaches a young collection. Compare that example to Figure 2-10, which shows a great deal of allocation occurring outside of the TLABs.



Thread Local Allocation Buffer (TLAB) Statistics

TLAB Count	908
Maximum TLAB Size	2.88 MB
Minimum TLAB Size	298.35 kB
Average TLAB Size	1.05 MB
Total Memory Allocated For TLABs	952.96 MB

Statistics for object allocations (outside TLABs)

Object Count	73
Maximum Object Size	26.86 MB
Minimum Object Size	32 bytes
Average Object Size	7.79 MB
Total Memory Allocated For Objects	568.32 MB

Figure 2-10. Excessive allocation occurring outside of TLABs

The total memory allocated inside TLABs during this recording is 952.96 MB, and the total memory allocated for objects outside of TLABs is 568.32 MB. This is a case where either changing the application to use smaller objects, or tuning the JVM to allocate those objects in larger TLABs, can have a beneficial effect. Note that there are other tabs here that can display the actual objects that were allocated out of the TLAB; we can even arrange to get the stacks from when those objects were allocated. If there is a problem with TLAB allocation, JFR will pinpoint it very quickly.

Outside of JFR, the best way to look at this is is monitor the TLAB allocation by adding the `-XX:+PrintTLAB` flag to the command line in JDK8, or including `tlab*=trace` in the log configuration for JDK 11 (which provides the below information plus more). Then, at every young collection, the GC log will contain two kinds of line: a line for each thread describing the TLAB usage for that thread, and a summary line describing the overall TLAB usage of the JVM.

The per-thread line looks like this:

```
TLAB: gc thread: 0x00007f3c10b8f800 [id: 18519] desired_size: 221KB
     slow allocs: 8  refill waste: 3536B alloc: 0.01613    11058KB
     refills: 73 waste  0.1% gc: 10368B slow: 2112B fast: 0B
```

The `gc` in this output means that the line was printed during GC; the thread itself is a regular application thread. The size of this thread's TLAB is 221 KB. Since the last young collection, it allocated eight objects from the heap (`slow allocs`); that was 1.6% (0.01613) of the total amount of allocation done by this thread, and it amounted to 11,058 KB. 0.1% of the TLAB was "wasted," which comes from three things: 10,336 bytes were free in the TLAB when the current GC cycle started; 2,112 bytes were free in other (retired) TLABs, and 0 bytes were allocated via a special "fast" allocator.

After the TLAB data for each thread has been printed, the JVM provides a line of summary data (this data is provided in JDK 11 by configuring the log for `tlab*=debug`):

```
TLAB totals: thrds: 66  refills: 3234 max: 105
          slow allocs: 406 max 14 waste:  1.1% gc: 7519856B
          max: 211464B slow: 120016B max: 4808B fast: 0B max: 0B
```

In this case, 66 threads performed some sort of allocation since the last young collection. Among those threads, they refilled their TLABs 3,234 times; the most any particular thread refilled its TLAB was 105. Overall there were 406 allocations to the heap (with a maximum of 14 done by one thread), and 1.1% of the TLABs were wasted from the free space in retired TLABs.

In the per-thread data, if threads show a large number of allocations outside of TLABs, consider resizing them.

SIZING TLABS

Applications that spend a lot of time allocating objects outside of TLABs will benefit from changes that can move the allocation to a TLAB. If there are only a few specific object types that are always allocated outside of a TLAB, then programmatic changes are the best solution.

Otherwise—or if programmatic changes are not possible—you can attempt to resize the TLABs to fit the application use case. Because the TLAB size is based on the size of eden, adjusting the new size parameters will automatically increase the size of the TLABs.

The size of the TLABs can be set explicitly using the flag `-XX:TLABSize=`*`N`* (the default value, 0, means to use the dynamic calculation previously described). That flag sets only the initial size of the TLABs; to prevent resizing at each GC, add `-XX:-ResizeTLAB` (the default for that flag is `true` on most common platforms). This is the easiest (and, frankly, the only really useful) option for exploring the performance of adjusting the TLABs.

When a new object does not fit in the current TLAB (but would fit within a new, empty TLAB), the JVM has a decision to make: whether to allocate the object in the heap, or whether to retire the current TLAB and allocate a new one. That decision is based on several parameters. In the TLAB logging output, the `refill waste` value gives the current threshold for that decision: if the TLAB cannot accommodate a new object that is larger than that value, then the new object will be allocated in the heap. If the object in question is smaller than that value, the TLAB will be retired.

That value is dynamic, but it begins by default at 1% of the TLAB size—or, specifically, at the value specified by `-XX:TLABWasteTargetPercent=`*`N`*. As each allocation is done outside the heap, that value is increased by the value of `-XX:TLABWasteIncrement=`*`N`* (the default is 4). This prevents a thread from reaching the threshold in the TLAB and continually allocating objects in the heap: as the target percentage increases, the chances of the TLAB being retired also increases. Adjusting the `TLABWasteTargetPercent` value also adjusts the size of the TLAB, so while it is possible to play with this value, its effect is not always predictable.

Finally, when TLAB resizing is in effect, the minimum size of a TLAB can be specified with `-XX:MinTLABSize=`*`N`* (the default is 2 KB). The maximum size of a TLAB is slightly less than 1 GB (the maximum space that can be occupied by an array of integers, rounded down for object alignment purposes) and cannot be changed.

QUICK SUMMARY

1. Applications that allocate a lot of large objects may need to tune the TLABs (though often using smaller objects in the application is a better approach).

HUMONGOUS OBJECTS

Objects that are allocated outside a TLAB are still allocated within eden when possible. If the object cannot fit within eden, then it must be allocated directly in the old generation. That prevents the normal GC lifecycle for that object, so if it is short-lived, GC is negatively affected. There’s little to do in that case other than change the application so that it doesn’t need those short-lived huge objects.

Humongous objects are treated differently in G1 GC, however: G1 will allocate them in the old generation if they are bigger than a G1 region. So applications that use a lot of humongous objects in G1 GC may need special tuning to compensate for that.

G1 GC REGION SIZES

G1 GC divides the heap into a number of regions, each of which has a fixed size. The region size is not dynamic; it is determined at startup based on the minimum size of the heap (the value of `Xms`). The minimum region size is 1 MB. If the minimum heap size is greater than 2 GB, the size of the regions will be set according to this formula (using log base 2):

```
region_size = 1 << log(Initial Heap Size / 2048);
```

In short, the region size is the smallest power of 2 such that there are close to 2,048 regions when the initial heap size is divided. There are some minimum and maximum constraints in use here too; the region size is always at least 1 MB and never more than 32 MB. Table 2-3 sorts out all the possibilities.

Table 2-3. Default G1 region sizes

Heap size	Default G1 region size
Less than 4 GB	1 MB
Between 4 GB and 8 GB	2 MB
Between 8 GB and 16 GB	4 MB
Between 16 GB and 32 GB	8 MB
Between 32 GB and 64 GB	16 MB
Larger than 64 GB	32 MB

The size of a G1 region can be set with the `-XX:G1HeapRegionSize=N` flag (the default for which is nominally 0, meaning to use the dynamic value just described). The value given here should be a power of 2 (e.g., 1 MB or 2 MB); otherwise it is rounded down to the nearest power of 2.

G1 REGION SIZES AND LARGE HEAPS

Normally the G1 GC region size needs to be tuned only to handle humongous object allocation, but there is one other case where it might need to be tuned.

Consider an application that specifies a very large heap range, e.g., `-Xms2G -Xmx32G`. In that case, the region size will be 1 MB. When the heap is fully expanded, there will be 32,000 G1 GC regions. That is a lot of separate regions to process; the G1 GC algorithm is designed around the idea that the number of regions is closer to 2,048. Increasing the size of the G1 GC region will make G1 GC a little more efficient in this example; select a value so that there will be close to 2,048 regions at the expected heap size.

G1 GC ALLOCATION OF HUMONGOUS OBJECTS

If the G1 GC region size is 1 MB and a program allocates an array of 2 million bytes, the array will not fit within a single G1 GC region. But these humongous objects must be allocated in contiguous G1 GC regions. If the G1 GC region size is 1 MB, then to allocate a 3.1 MB array, G1 GC must find four regions within the old generation in which to allocate the array. (The rest of the last region will remain empty, wasting 0.9 MB of space.) This defeats the way G1 GC normally performs compaction, which is to free arbitrary regions based on how full they are.

In fact, G1 GC defines a humoungous object as one that is half of the region size, so allocating an array of 512MB (plus 1 byte) will, in this case, trigger the humongous allocation we're discussing.

Because the humongous object is allocated directly in the old generation, it cannot be freed during a young collection. So if the object is short-lived, this also defeats the generational design of the collector. The humongous object will be collected during the concurrent G1 GC cycle. On the bright side, the humongous object can be freed quickly since it is the only object in the regions it occupies. Humongous objects are freed during the cleanup phase of the concurrent cycle (rather than during a mixed GC).

Increasing the size of a G1 GC region so that all objects the program will allocate can fit within a single G1 GC region can make G1 GC more efficient.

Humongous allocation used to be a far bigger problem in G1 GC because finding the necessary regions to allocate the object would usually require a full GC (and because such full GCs were not parallelized). Improvments in G1 GC in JDK 8u60 (and in all JDK 11 builds) minimze this issue so it isn’t necessarily the critical problem it sometimes used to be.

QUICK SUMMARY

1. G1 regions are sized in powers of 2, starting at 1 MB.
2. Heaps that have a very different maximum size than initial size will have too many G1 regions; the G1 region size should be increased in that case.
3. Applications that allocate objects larger than half the size of a G1 region should increase the G1 region size, so that the objects can fit within a G1 region. An application must allocate an object that is at least 512 KB for this to apply (since the smallest G1 region is 1 MB).

AggressiveHeap

The `AggressiveHeap` flag (by default, `false`), was introduced in an early version of Java as an attempt to make it easier to easily set a variety of command-line arguments—arguments that would be appropriate for a very large machine with a lot of memory running a single JVM.

Although the flag has been carried forward since those versions and is still present, it is no longer recommended (though it is not yet officially deprecated). The problem with this flag is that it hides the actual tunings it adopts, making it quite hard to figure out what the JVM is actually setting. Some of the values it sets are now set ergonomically based on better information about the machine running the JVM, so there are actually cases where enabling this flag hurts performance. I have often seen command lines that include this flag and then later override values that it sets. (For the record, that works: later values in the command line currently override earlier values. That behavior is not guaranteed.)

Table 2-4 lists all the tunings that are automatically set when the `AggressiveHeap` flag is enabled.

Table 2-4. Tunings enabled with AggressiveHeap

Flag	Value
`Xmx`	The minimum of half of all memory, or all memory: 160 MB
`Xms`	The same as `Xmx`
`NewSize`	3/8ths of whatever was set as `Xmx`
`UseLargePages`	`true`
`ResizeTLAB`	`false`
`TLABSize`	256 KB
`UseParallelGC`	`true`
`ParallelGCThreads`	Same as current default
`YoungPLABSize`	256 KB (default is 4 KB)
`OldPLABSize`	8 KB (default is 1 KB)
`CompilationPolicyChoice`	0 (the current default)
`ThresholdTolerance`	100 (default is 10)

`ScavengeBeforeFullGC`	`false` (default is `true`)
`BindGCTaskThreadsToCPUs`	`true` (default is `false`)

Those last six flags are obscure enough that I have not discussed them elsewhere in this book. Briefly, they cover these areas:

PLAB sizing

PLABs are promotion-local allocation buffers—these are per-thread regions used during scavenging the generations in a GC. Each thread can promote into a specific PLAB, negating the need for synchronization (analogous to the way TLABs work).

Compilation policies

The JVM ships with some alternate JIT compilation algorithms. The current default algorithm was, at one time, somewhat experimental, but this is now the recommended policy.

Disabling young GCs before full GCs

Setting `ScavengeBeforeFullGC` to `false` means that when a full GC occurs, the JVM will not perform a young GC before a full GC. That is usually a bad thing, since it means that garbage objects in the young generation (which are eligible for collection) can prevent objects in the old generation from being collected. Clearly there is (or was) a point in time when that setting made sense (at least for certain benchmarks), but the general recommendation is not to change that flag.

Binding GC threads to CPUs

Setting the last flag in that list means that each parallel GC thread is bound to a particular CPU (using OS-specific calls). In limited circumstances—when the GC threads are the only thing running on the machine, and heaps are very large—that makes sense. In the general case, it is better if GC threads can run on any available CPU.

As with all tunings, your mileage may vary, and if you carefully test the `AggressiveHeap` flag and find that it improves performance, then by all means use it. Just be aware of what it is doing behind the scenes, and realize that whenever the JVM is upgraded, the relative benefit of this flag will need to be reevaluated.

QUICK SUMMARY

1. The `AggressiveHeap` flag is a legacy attempt to set a number of heap parameters to values that make sense for a single JVM running on a very large machine.

2. Values set by this flag are not adjusted as JVM technology improves, so its usefulness in the long run is dubious (even though it still is often used).

Full Control Over Heap Size

"Sizing the Heap" discussed the default values for the initial minimum and maximum size of the heap. Those values are dependent on the amount of memory on the machine as well as the JVM in use, and the data presented there had a number of corner cases to it. If you're curious about the full details about how the default heap size is actually calculated, this section will explain the details. Those details include some very low-level tuning flags; in certain circumstances, it might be more convenient to adjust the way those calculations are done (rather than simply setting the heap size). This might be the case if, for example, you want to run multiple JVMs with a common (but adjusted) set of ergonomic heap sizes. For the most part, the real goal of this section is to complete the explanation of how those default values are chosen.

The default sizes are based on the amount of memory on a machine, which can be set with the `-XX:MaxRAM=`*`N`* flag. Normally, that value is calculated by the JVM by inspecting the amount of memory on the machine. However, the JVM limits `MaxRAM` to 1 GB for the client compiler, 4 GB for 32-bit server compilers, and 128 GB for 64-bit compilers. The maximum heap size is one-quarter of `MaxRAM`. This is why the default heap size can vary: if the physical memory on a machine is less than `MaxRAM`, the default heap size is one-quarter of that. But even if hundreds of gigabytes of RAM are available, the most the JVM will use by default is 32 GB: one-quarter of 128 GB.

The default maximum heap calculation is actually this:

```
Default Xmx = MaxRAM / MaxRAMFraction
```

Hence, the default maximum heap can also be set by adjusting the value of the `-XX:MaxRAMFraction=N` flag, which defaults to 4. Finally, just to keep things interesting, the `-XX:ErgoHeapSizeLimit=N` flag can also be set to a maximum default value that the JVM should use. That value is 0 by default (meaning to ignore it); otherwise, that limit is used if it is smaller than `MaxRAM` / `MaxRAMFraction`.

On the other hand, on a machine with a very small amount of physical memory, the JVM wants to be sure it leaves enough memory for the operating system. This is why the JVM will limit the maximum heap to 96 MB or less on machines with only 192 MB of memory. That calculation is based on the value of the `-XX:MinRAMFraction=N` flag, which defaults to 2:

```
if ((96 MB * MinRAMFraction) > Physical Memory) {
    Default Xmx = Physical Memory / MinRAMFraction;
}
```

The initial heap size choice is similar, though it has fewer complications. The initial heap size value is determined like this:

```
Default Xms =  MaxRAM / InitialRAMFraction
```

As can be concluded from the default minimum heap sizes, the default value of the `InitialRAMFraction` flag is 64. The one caveat here occurs if that value is less than 5 MB—or, strictly speaking, less than the values specified by `-XX:OldSize=N` (which defaults to 4 MB) plus `-XX:NewSize=N` (which defaults to 1 MB). In that case, the sum of the old and new sizes is used as the initial heap size.

QUICK SUMMARY

1. The calculations for the default initial and maximum heap sizes are fairly straightforward on most machines.

2. Around the edges, these calculations can be quite involved.

Experimental GC Algorithms

In JDK 8 and JDK 11 production VMs with multiple CPUs, you'll use either the G1 GC or throughput collector depending on your application requirements. On small machines, you'll use the serial collector if that is appropriate for your hardware. Those are the production-supported collectors.

JDK 12 introduces some new collectors. Although these collectors are not necessarily production-ready, we'll take a peek into them for experimental purposes.

Concurrent Compaction: ZGC and Shenandoah

Existing concurrent collectors are not fully concurrent. Neither G1 GC nor CMS has concurrent collection of the young generation: freeing the young generation requires all application threads to be stopped. And neither of those collectors does concurrent compaction. In G1 GC, the old generation is compacted as an effect of the mixed GC cycles: within a target region, objects that are not freed are compacted into empty regions. In CMS, the old generation is compacted when it becomes too fragmented to allow new allocations. Collections of the young generation also compact that portion of the heap by moving surviving objects into the survivor spaces or the old generation.

During compaction, objects move their position in memory. This is the primary reason why the JVM stops all application threads during that operation—the algorithms to update the memory references are much simpler if the application threads are known to be stopped. So the pause times of an application are dominated by the time spent moving objects and making sure references to them are up-to-date.

There are two experimental collectors that are designed to address this problem. The first of these is the Z Garbage Collector, or ZGC; the second is the Shenandoah collector. ZGC first appeared in JDK 11; Shenandoah GC first appeared in JDK 12 though has now been backported to JDK 8 and JDK 11. JVM builds from AdoptOpenJDK (or that you compile yourself from source) contain both collectors; builds that come from Oracle contain only ZGC.

To use these collectors, you must specify the `-XX:+UnlockExperimentalVMOptions` flag (by default it is false). Then you specify either `-XX:+UseZGC` or `-XX:+UseShenandoahGC` in place of other GC algorithms. Like other GC algorithms, there are several tunings knobs for them, but these are changing as the algorithms are in development, so for now we'll just run with the default arguments. (And both collectors have the goal of running with minimal tuning.)

Althougth they take quite different approaches, these two collectors both allow concurrent compaction of the heap, meaning that objects in the heap can be moved without stopping all application threads. There are two main effects of that: first, the heap is no longer generational (i.e., there is no longer a young and old generation; there is simply a single heap). The idea behind the young generation is that it is faster to collect a small portion of the heap rather than the entire heap, and many (hopefully most) of those objects will be garbage. So the young generation allows for shorter pauses for much of the time.

If the application threads don't need to be paused during collection, then the need for the young generation disappears, and so these algorithms no longer need to segment the heap into generations.

The second effect is that the latency of operations performed by the application threads can be expected to be reduced (at least in many cases). Consider a REST call that normally executes in 200 milliseconds; if that call is interrupted by a young collection in G1 GC, and that collection takes 500 milliseconds, then the user will see that the REST call took 700 milliseconds. Most of the calls, of course, won't hit that situation, but some will be, and these outliers will affect the overall performance of the system. Without the need to stop the application threads, the concurrent compacting collectors will not see these same outliers.

This simplifies the situation somewhat. Recall from the discussion of G1 GC that the background threads which marked the free objects in the heap regions sometimes had short pauses. So G1 GC has three types of pauses: relatively long pauses for a full GC (well, hopefuly you've tuned well enough for that not to happen), shorter pauses for a young GC collection (including a mixed collection that frees and compacts some of the old generation), and very short pauses for the marking threads.

Both ZGC and Shenandoah have similar pauses that fall into that latter category; there will still be short periods of time when all the application threads are stopped. The goal of these collectors is to keep those times very short, on the order of 10 milliseconds.

These collectors can also introduce some latency on individual thread operations. The details differ between the algortihms, but in a nutshell access to an object by an application thread is guarded by a barrier. If the object happens to be in the process of being moved, then the application thread waits at the barrier until the move is complete. (For that matter, if the application thread is accessing the object, then the GC thread must wait at the barrier until it can relocate the object.) In effect, this is a form of locking on the object reference, but that term makes this process seem far more heavyweight than it actually is. In general, this has a very small effect on the application throughput.

LATENCY EFFECTS OF CONCURRENT COMPACTION

To get a feel for the overall impact of these algorithms, consider the data in Table 2-5. This table shows the response times from a REST server handling a fixed load of 500 operations/sec using various collectors. The operation here is very fast; it simply allocates and saves a fairly large byte array (replacing an existing pre-saved array to keep memory pressure constant).

Table 2-5. Latency effects of concurrent compacting collectors

Collector	Average Time	90th% Time	99th% Time	Max Time
Parallel GC	13 ms	60 ms	160 ms	265 ms
G1 GC	5 ms	10 ms	35 ms	87 ms
ZGC	1 ms	5 ms	5 ms	20 ms
Shenandoah GC	1 ms	5 ms	5 ms	22 ms

These results are just what we'd expect from the various collectors. The full GC times of the parallel collector cause a maximum reponse time of 265 milliseconds and lots of outliers where the response time is more than 50 milliseconds. With G1 GC, those full GC times have gone away, but there are still some shorter times for the young collections, yielding a maximum time of 87 milliseconds and outliers about 10 milliseconds. And with the concurrent collectors, those young collection pauses have disappered so that the maximum times are now around 20 milliseconds and the outliers only 5 milliseconds.

One caveat here: garbage collection pauses traditionally have been the largest contributor to latency outliers like we're discussing here. But there are other causes: temporary network congestion between server and client, OS scheduling delays, and so on. So while a lot of the outliers in the last two cases here are due to those short pauses of a few milliseconds that the concurrent collectors still have, we're now entering the realm where those other things also have a large impact on the total latency.

THROUGHPUT EFFECTS OF CONCURRENT COMPACTING COLLECTORS

The throughput effects of these collectors is harder to categorize. Like G1 GC, these collectors rely on background threads to scan and process the heap. So if sufficient CPU cycles are not available for these threads, the collectors will experience the same sort of concurrent failure we've seen before and end up doing a full GC. The concurrent compacting collectors will typically use even more background processing than the G1 GC background threads.

On the other hand, if sufficient CPU is available for those background threads, then throughput when using these collectors will be higher than the throughput of G1 GC or the parallel collector. This again is in line with what we saw in Chapter 1. Examples from that chapter showed that G1 GC can have higher throughput than the throughput collector when it offloads GC processing to background threads. The concurrent compacting collectors have that same avantage over the parallel collector, and a similar (but smaller) advantage over G1 GC.

No collection: Epsilon GC

JDK 11 also contains a collector that does nothing: the epsilon collector. When you use this collector, objects are never freed from the heap, and when the heap fills up, you will get an out of memory error.

Traditional programs will not be able to use this collector, of course; it finds its usefulness in two situations:

- Very short-lived programs
- Programs carefully written to reuse memory and never perform new allocations

That second category is useful in some embedded environments with very limited memory. That sort of programming is quite specialized; we'll not consider it here. But the first case holds some interesting possibilities.

Consider the case of a program that allocates an array list of 4096 elements, each of this is a .5 MB byte array. The time to turn that program with various collectors is shown in Table 2-6. Default GC tunings are used in this example.

Table 2-6. Running time with disabled GC

Collector	Time	Heap Required
Parallel GC	2.3s	3072m
G1 GC	3.24s	4096m
Epsilon	1.6s	2052m

Disabling garbage collection is a significant advantage in this case, yielding a 30% improvement. And the other collectors require significant memory overhead: like the other experimental collectors we've seen, the Epsilon collector is not generational (since the objects cannot be freed, there's no need to set up a separate space to be able to free them quickly). So for this test that produces an object of about 2 GB, the total heap required for the Epsilon collector is just over that; we can run that case with `-Xmx2052m`. The parallel collector needs one-third more memory to hold its young generation, while G1 GC needs even more memory to set up all its regions.

To use this collectors, you again specify the `-XX:+UnlockExperimentalVMOptions` flag with `-XX:+UseEpsilonGC`.

Running with this collector is risky unless you are certain that the program will never need more memory than you provide it. But in those cases, it can give a nice performance boost.

Summary

The past two chapters have spent a lot of time delving into the deep details of how GC (and its various algorithms) work. If GC is taking longer than you'd like, knowing how all of that works should aid you in taking the necessary steps to improve things.

Now that we understand all the details, let's take a step back to determine an approach to choosing and tuning a garbage collector. Here's a quick set of questions to ask yourself to help put everything in context:

Can your application tolerate some full GC pauses?

> If not, then G1 GC is the algorithm of choice. Even if you can tolerate some full pauses, G1 GC will often be better than parallel GC unless your application is CPU bound.

Are you getting the performance you need with the default settings?

> Try the default settings first. As GC technology matures, the ergonomic (automatic) tuning gets better all the time.
>
> If you're not getting the performance you need, make sure that GC is your problem. Look at the GC logs and see how much time you're spending in GC and how frequently the long pauses occur. For a busy application, if you're spending 3% or less time in GC, you're not going to get a lot out of tuning (though you can always try and reduce outliers if that is your goal).

Are the pause times that you have somewhat close to your goal?

If they are, then adjusting the maximum pause time may be all you need. If they aren't, then you need to do something else. If the pause times are too large but your throughput is OK, you can reduce the size of the young generation (and for full GC pauses, the old generation); you'll get more, but shorter, pauses.

Is throughput lagging even though GC pause times are short?

You need to increase the size of the heap (or at least the young generation).

More isn't always better: bigger heaps lead to longer pause times. Even with a concurrent collector, a bigger heap means a bigger young generation by default, so you'll see longer pause times for young collections. But if you can, increase the heap size, or at least the relative sizes of the generations.

Are you using a concurrent collector and seeing full GCs due to concurrent-mode failures?

If you have available CPU, try increasing the number of concurrent GC threads or starting the background sweep sooner by adjusting the `InitiatingHeapOccupancyPercent`. For G1, the concurrent cycle won't start if there are pending mixed GCs; try reducing the mixed GC count target.

Are you using a concurrent collector and seeing full GCs due to promotion failures?

In G1 GC, an evacuation failure (to-space overflow) indicates that the heap is fragmented, but that can usually be solved if G1 GC performs its background sweeping sooner and mixed GCs faster. Try increasing the number of concurrent G1 threads, adjusting the `InitiatingHeapOccupancyPercent`, or reducing the mixed GC count target.

About the Author

Scott Oaks is an architect at Oracle Corporation, where he works on the performance of Oracle’s middleware software. Prior to joining Oracle, he worked for years at Sun Microsystems, specializing in many disparate technologies from the SunOS kernel to network programming and RPCs to Windows systems and the OPEN LOOK Virtual Window Manager. In 1996, Scott became a Java evangelist for Sun and in 2001 joined their Java Performance Group—which has been his primary focus ever since. Scott also authored O’Reilly’s *Java Security*, *Java Threads*, *JXTA in a Nutshell*, and *Jini in a Nutshell* titles.