Java Platform, Standard Edition

HotSpot Virtual Machine Garbage Collection Tuning Guide

Release 24
G16827-02
April 2025

Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide, Release 24

G16827-02

Copyright © 2015, 2025, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience vii
Documentation Accessibility vii
Diversity and Inclusion vii
Related Documents vii
Conventions viii

1 Introduction to Garbage Collection Tuning

What Is a Garbage Collector? 1-1
Why Does the Choice of Garbage Collector Matter? 1-1
Supported Operating Systems in Documentation 1-3

2 Ergonomics

Garbage Collector, Heap, and Runtime Compiler Default Selections 2-1
Behavior-Based Tuning 2-1
- Maximum Pause-Time Goal 2-2
- Throughput Goal 2-2
- Footprint 2-2

Tuning Strategy 2-2

3 Garbage Collector Implementation

Generational Garbage Collection 3-1
Generations 3-2
Performance Considerations 3-3
Throughput and Footprint Measurement 3-4

4 Factors Affecting Garbage Collection Performance

Total Heap 4-1
- Heap Options Affecting Generation Size 4-1
- Default Option Values for Heap Size 4-2

Conserving Dynamic Footprint by Minimizing Java Heap Size 4-2
The Young Generation 4-3
Young Generation Size Options 4-3
Survivor Space Sizing 4-3

5 Available Collectors

Serial Collector 5-1
Parallel Collector 5-1
Garbage-First (G1) Garbage Collector 5-1
The Z Garbage Collector 5-2
Selecting a Collector 5-2

6 The Parallel Collector

Number of Parallel Collector Garbage Collector Threads 6-1
Arrangement of Generations in Parallel Collectors 6-2
Parallel Collector Ergonomics 6-2
Options to Specify Parallel Collector Behaviors 6-2
Priority of Parallel Collector Goals 6-3
Parallel Collector Generation Size Adjustments 6-3
Parallel Collector Default Heap Size 6-3
Specification of Parallel Collector Initial and Maximum Heap Sizes 6-3
Excessive Parallel Collector Time and OutOfMemoryError 6-4
Parallel Collector Measurements 6-4

7 Garbage-First (G1) Garbage Collector

Introduction to Garbage-First (G1) Garbage Collector 7-1
Enabling G1 7-2
Basic Concepts 7-2
Heap Layout 7-2
Garbage Collection Cycle 7-3
Garbage Collection Pauses and Collection Set 7-5
Remembered Set 7-5
Collection Set 7-5
Garbage Collection Process 7-6
Garbage-First Internals 7-6
Java Heap Sizing 7-6
Young-Only Phase Generation Sizing 7-7
Space-Reclamation Phase Generation Sizing 7-7
Periodic Garbage Collections 7-8

Determining Initiating Heap Occupancy 7-8
Marking 7-9
Evacuation Failure 7-9
Humongous Objects 7-9
Ergonomic Defaults for G1 GC 7-10
Comparison to Other Collectors 7-11

8 Garbage-First Garbage Collector Tuning

General Recommendations for G1 8-1
Moving to G1 from Other Collectors 8-2
Improving G1 Performance 8-2
Observing Full Garbage Collections 8-2
Humongous Object Fragmentation 8-3
Tuning for Latency 8-3
Unusual System or Real-Time Usage 8-3
Reference Object Processing Takes Too Long 8-4
Young-Only Collections Within the Young-Only Phase Take Too Long 8-4
Mixed Collections Take Too Long 8-4
Collections Occur Back to Back 8-5
High Merge Heap Roots and Scan Heap Roots Times 8-5
Tuning for Throughput 8-5
Tuning for Heap Size 8-6
Tunable Defaults 8-6

9 The Z Garbage Collector

Setting the Heap Size 9-1
Returning Unused Memory to the Operating System 9-2
Using Large Pages 9-2
Enabling Large Pages On Linux 9-2
Enabling Transparent Huge Pages On Linux 9-3

10 Other Considerations

Finalization and Weak, Soft, and Phantom References 10-1
Finalization 10-1
Migrating from Finalization 10-2
The try-with-Resources Statement 10-2
The Cleaner API 10-2
Reference-Object Types 10-4
Explicit Garbage Collection 10-5

Soft References 10-5
Class Metadata 10-5

Preface

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

Audience

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

Documentation Accessibility

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

Access to Oracle Support

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Documents

For more information, see the following documents:

- Garbage Collection: Algorithms for Automatic Dynamic Memory Management by Richard Jones, and Rafael D Lins.
- The Garbage Collection Handbook: The Art of Automatic Memory Management (Chapman & Hall/CRC Applied Algorithms and Data Structures)

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction to Garbage Collection Tuning

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

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

Topics

- What Is a Garbage Collector?
- Why Does the Choice of Garbage Collector Matter?
- Supported Operating Systems in Documentation

What Is a Garbage Collector?

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

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

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

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

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

Why Does the Choice of Garbage Collector Matter?

The purpose of a garbage collector is to free the application developer from manual dynamic memory management. The developer is freed of the requirement to match allocations with

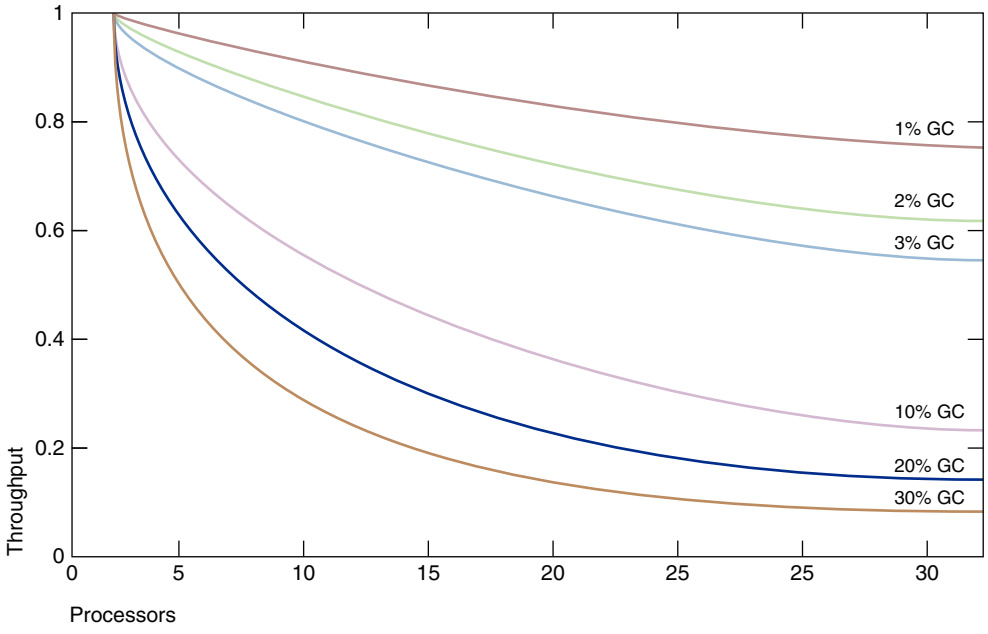
deallocations and closely take care of the lifetimes of allocated dynamic memory. This completely eliminates some classes of errors related to memory management at the cost of some additional runtime overhead. The Java HotSpot VM provides a selection of garbage collection algorithms to choose from.

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

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

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

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



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

The serial collector is usually adequate for most small applications, in particular those requiring heaps of up to approximately 100 megabytes on modern processors. The other collectors have

additional overhead or complexity, which is the price for specialized behavior. If the application does not need the specialized behavior of an alternate collector, use the serial collector. One situation where the serial collector isn't expected to be the best choice is a large, heavily threaded application that runs on a machine with a large amount of memory and two or more processors. When applications are run on such server-class machines, the Garbage-First (G1) collector is selected by default; see Ergonomics.

Supported Operating Systems in Documentation

This document and its recommendations apply to all JDK 24 supported system configurations, limited by actual availability of some garbage collectors in a particular configuration. See Oracle JDK Certified System Configurations.

2
Ergonomics

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

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

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

Topics

- Garbage Collector, Heap, and Runtime Compiler Default Selections
- Behavior-Based Tuning
  - Maximum Pause-Time Goal
  - Throughput Goal
  - Footprint
- Tuning Strategy

Garbage Collector, Heap, and Runtime Compiler Default Selections

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

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

> **Note:**
>
> The VM considers machines as server-class if the VM detects more than two processors and a heap size larger or equal to 1792 MB.

Behavior-Based Tuning

The Java HotSpot VM garbage collectors can be configured to preferentially meet one of two goals: maximum pause-time and application throughput. If the preferred goal is met, the

collectors will try to maximize the other. Naturally, these goals can't always be met: Applications require a minimum heap to hold at least all of the live data, and other configuration might preclude reaching some or all of the desired goals.

Maximum Pause-Time Goal

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

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

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

Throughput Goal

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

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

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

Footprint

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

Tuning Strategy

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

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

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

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

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

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

3
Garbage Collector Implementation

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

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

Topics

- Generational Garbage Collection
- Generations
- Performance Considerations
- Throughput and Footprint Measurement

Generational Garbage Collection

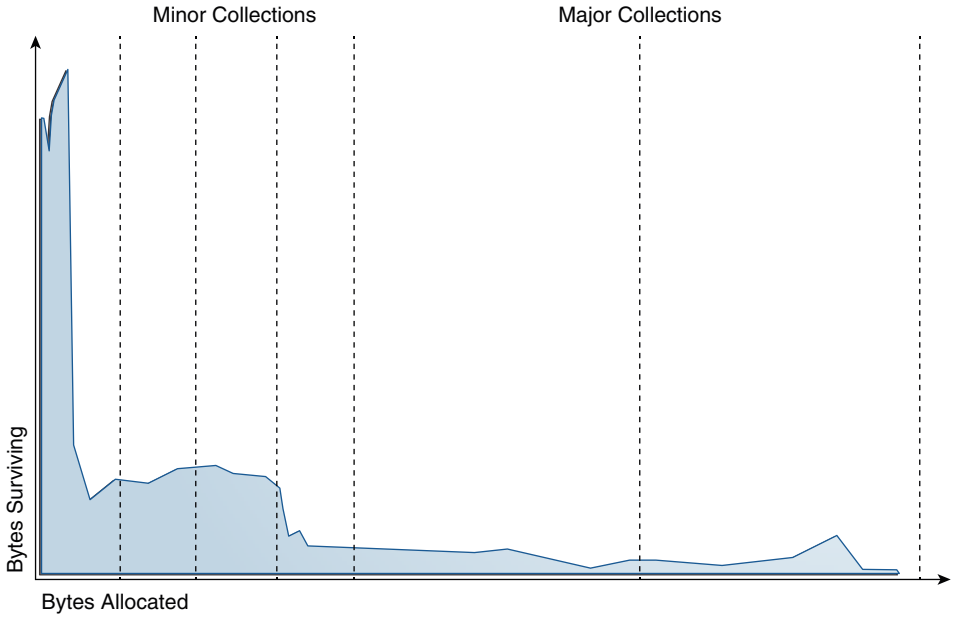
An object is considered garbage and its memory can be reused by the VM when it can no longer be reached from any reference of any other live object in the running program.

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

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

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

Figure 3-1 Typical Distribution for Lifetimes of Objects



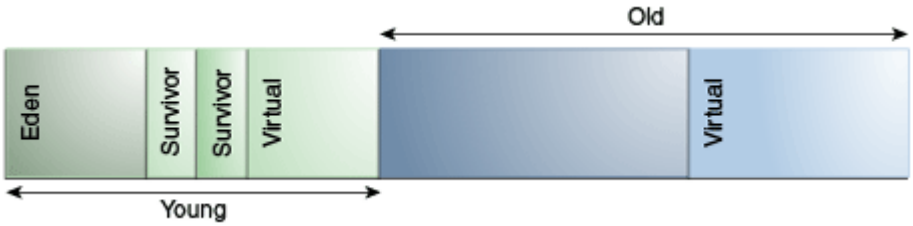
Some objects do live longer, and so the distribution stretches out to the right. For instance, there are typically some objects allocated at initialization that live until the VM exits. Between these two extremes are objects that live for the duration of some intermediate computation, seen here as the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

Generations

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

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

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



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

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

Performance Considerations

The primary measures of garbage collection are throughput and latency.

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

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

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

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

There is no one right way to choose the size of a generation. The best choice is determined by the way the application uses memory as well as user requirements. Thus the virtual machine's

choice of a garbage collector isn't always optimal and may be overridden with command-line options; see Factors Affecting Garbage Collection Performance.

Throughput and Footprint Measurement

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

For example, the throughput of a web server may be tested using a client load generator. However, pauses due to garbage collection are easily estimated by inspecting the diagnostic output of the virtual machine itself. The command-line option `-verbose:gc` prints information about the heap and garbage collection at each collection. Here is an example:

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

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

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

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

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

```
[10.178s][info][gc,start     ] GC(36) Pause Young (G1 Evacuation Pause)
[10.178s][info][gc,task      ] GC(36) Using 28 workers of 28 for evacuation
[10.191s][info][gc,phases    ] GC(36) Pre Evacuate Collection Set: 0.0ms
[10.191s][info][gc,phases    ] GC(36) Evacuate Collection Set: 6.9ms
[10.191s][info][gc,phases    ] GC(36) Post Evacuate Collection Set: 5.9ms
[10.191s][info][gc,phases    ] GC(36) Other: 0.2ms
[10.191s][info][gc,heap      ] GC(36) Eden regions: 286->0(276)
[10.191s][info][gc,heap      ] GC(36) Survivor regions: 15->26(38)
[10.191s][info][gc,heap      ] GC(36) Old regions: 88->88
[10.191s][info][gc,heap      ] GC(36) Humongous regions: 3->1
[10.191s][info][gc,metaspace ] GC(36) Metaspace: 8152K->8152K(1056768K)
[10.191s][info][gc           ] GC(36) Pause Young (G1 Evacuation Pause) 391M-
>114M(508M) 13.075ms
[10.191s][info][gc,cpu       ] GC(36) User=0.20s Sys=0.00s Real=0.01s
```

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

4

Factors Affecting Garbage Collection Performance

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

Topics

- Total Heap
  - Heap Options Affecting Generation Size
  - Default Option Values for Heap Size
  - Conserving Dynamic Footprint by Minimizing Java Heap Size
- The Young Generation
  - Young Generation Size Options
  - Survivor Space Sizing

Total Heap

The most important factor affecting garbage collection performance is total available memory. Because collections occur when generations fill up, throughput is correlated with memory size.

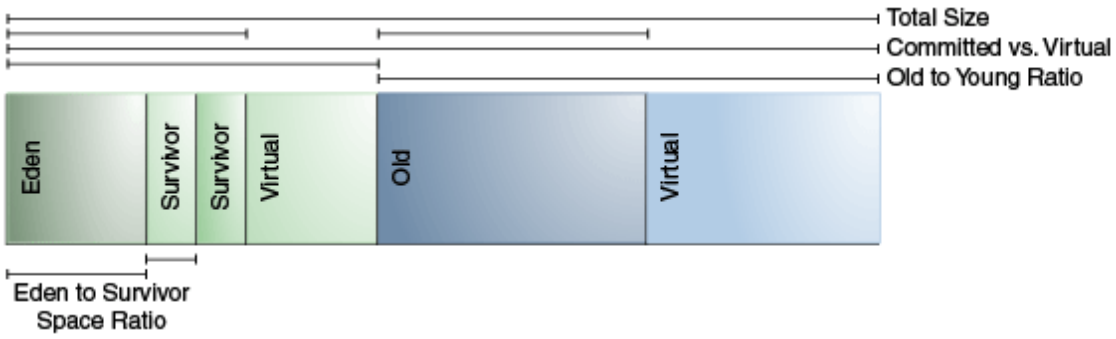
> **Note:**
>
> The following discussion regarding growing and shrinking of the heap, the heap layout, and default values uses the serial collector as an example. While the other collectors use similar mechanisms, the details presented here may not apply to other collectors. Refer to the respective topics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

Heap Options Affecting Generation Size

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

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

Figure 4-1 Heap Options



Default Option Values for Heap Size

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

This target range is set as a percentage by the options `-XX:MinHeapFreeRatio=<minimum>` and `-XX:MaxHeapFreeRatio=<maximum>`, and the total size is bounded below by `-Xms<min>` and above by `-Xmx<max>`.

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

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

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

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

Conserving Dynamic Footprint by Minimizing Java Heap Size

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

Minimize Java heap size by lowering the values of the options `-XX:MaxHeapFreeRatio` (default value is 70%) and `-XX:MinHeapFreeRatio` (default value is 40%) with the command-line options `-XX:MaxHeapFreeRatio` and `-XX:MinHeapFreeRatio`. Lowering `-XX:MaxHeapFreeRatio` to as low as 10% and `-XX:MinHeapFreeRatio` has shown to successfully reduce the heap size without too much performance degradation; however, results may vary greatly depending on

your application. Try different values for these parameters until they're as low as possible, yet still retain acceptable performance.

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

The Young Generation

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

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

Young Generation Size Options

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

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

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

Survivor Space Sizing

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

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

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

Table 4-1 provides the default values for survivor space sizing.

Table 4-1 Default Option Values for Survivor Space Sizing

Option	Default Value
`-XX:NewRatio`	2

Table 4-1 (Cont.) Default Option Values for Survivor Space Sizing

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

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

The following are general guidelines for server applications:

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

5
Available Collectors

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

Topics

- Serial Collector
- Parallel Collector
- Garbage-First (G1) Garbage Collector
- The Z Garbage Collector
- Selecting a Collector

Serial Collector

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

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

Parallel Collector

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

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

Garbage-First (G1) Garbage Collector

G1 is a mostly concurrent collector. Mostly concurrent collectors perform some expensive work concurrently to the application. This collector is designed to scale from small machines to large multiprocessor machines with a large amount of memory. It provides the capability to meet a pause-time goal with high probability, while achieving high throughput.

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

The Z Garbage Collector

ZGC provides max pause times under a millisecond, but at the cost of some throughput. It is intended for applications, which require low latency. Pause times are independent of heap size that is being used. ZGC works well for heap sizes from a few hundred megabytes to 16TB To enable this, use the `-XX:+UseZGC` option.

Selecting a Collector

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

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

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

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

If the recommended collector doesn't achieve the desired performance, then first attempt to adjust the heap and generation sizes to meet the desired goals. If performance is still inadequate, then try a different collector: Use the concurrent collector to reduce pause-time, and use the parallel collector to increase overall throughput on multiprocessor hardware.

6
The Parallel Collector

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

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

Topics

- Number of Parallel Collector Garbage Collector Threads
- Arrangement of Generations in Parallel Collectors
- Parallel Collector Ergonomics
  - Options to Specify Parallel Collector Behaviors
  - Priority of Parallel Collector Goals
  - Parallel Collector Generation Size Adjustments
  - Parallel Collector Default Heap Size
    - Specification of Parallel Collector Initial and Maximum Heap Sizes
- Excessive Parallel Collector Time and OutOfMemoryError
- Parallel Collector Measurements

Number of Parallel Collector Garbage Collector Threads

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

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

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

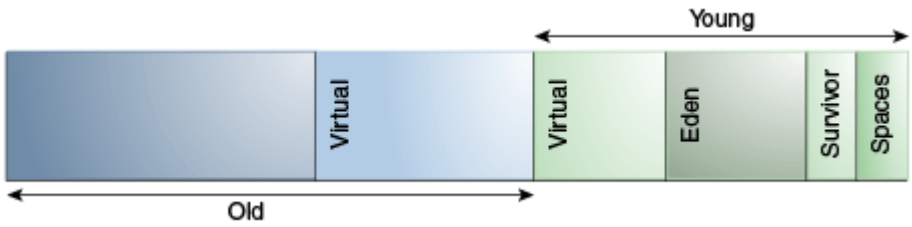
collection reserves a part of the old generation for promotions and the division of the available space into these "promotion buffers" can cause a fragmentation effect. Reducing the number of garbage collector threads and increasing the size of the old generation will reduce this fragmentation effect.

Arrangement of Generations in Parallel Collectors

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

That arrangement is shown in Figure 6-1:

Figure 6-1 Arrangement of Generations in the Parallel Collector



Parallel Collector Ergonomics

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

Options to Specify Parallel Collector Behaviors

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

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

  For example, `-XX:GCTimeRatio=19` sets a goal of 1/20 or 5% of the total time in garbage collection. The default value is 99, resulting in a goal of 1% of the time in garbage collection.
- Footprint: The maximum heap footprint is specified using the option `-Xmx`*<N>*. In addition, the collector has an implicit goal of minimizing the size of the heap as long as the other goals are being met.

Priority of Parallel Collector Goals

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

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

Parallel Collector Generation Size Adjustments

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

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

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

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

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

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

Parallel Collector Default Heap Size

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

Specification of Parallel Collector Initial and Maximum Heap Sizes

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

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

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

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

Excessive Parallel Collector Time and OutOfMemoryError

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

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

Parallel Collector Measurements

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

7

Garbage-First (G1) Garbage Collector

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

Topics

- Introduction to Garbage-First (G1) Garbage Collector
- Enabling G1
- Basic Concepts
  - Heap Layout
  - Garbage Collection Cycle
  - Garbage Collection Pauses and Collection Set
    - Remembered Set
    - Collection Set
    - Garbage Collection Process
  - Garbage-First Internals
    - Java Heap Sizing
      - Young-Only Phase Generation Sizing
      - Space-Reclamation Phase Generation Sizing
    - Periodic Garbage Collections
    - Determining Initiating Heap Occupancy
    - Marking
    - Evacuation Failure
    - Humongous Objects
- Ergonomic Defaults for G1 GC
- Comparison to Other Collectors

Introduction to Garbage-First (G1) Garbage Collector

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

- Heap sizes up to tens of GBs or larger, with more than 50% of the Java heap occupied with live data.
- Rates of object allocation and promotion that can vary significantly over time.
- A significant amount of fragmentation in the heap.

- Predictable pause-time target goals that aren't longer than a few hundred milliseconds, avoiding long garbage collection pauses.

G1 performs parts of its work at the same time as the application runs. It trades processor resources which would otherwise be available to the application for shorter collection pauses.

This is most visible in the use of one or more garbage collection threads active while the application runs. Thus, compared to throughput collectors, while garbage collection pauses are typically much shorter with the G1 collector, application throughput also tends to be slightly lower.

G1 is the default collector.

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

Enabling G1

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

Basic Concepts

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

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

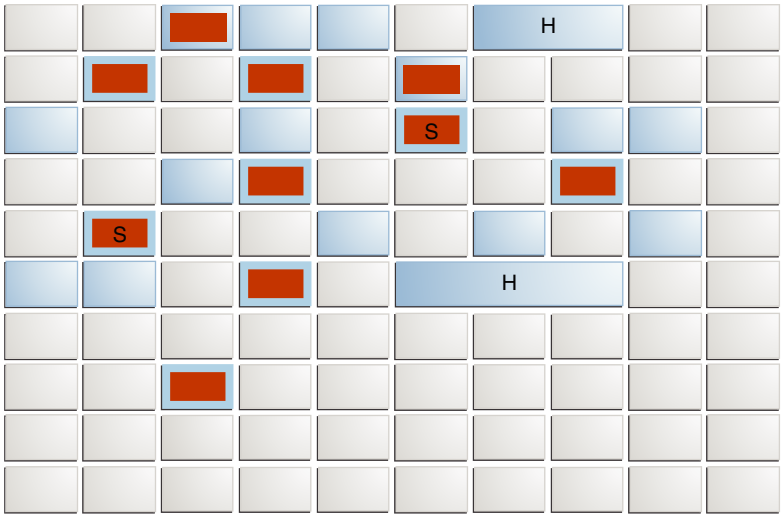
G1 reclaims space mostly by using evacuation: live objects found within selected memory areas to collect are copied into new memory areas, compacting them in the process. After an evacuation has been completed, the space previously occupied by live objects is reused for allocation by the application.

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

Heap Layout

G1 partitions the heap into a set of equally sized heap regions, each a contiguous range of virtual memory as shown in Figure 7-1. A region is the unit of memory allocation and memory reclamation. At any given time, each of these regions can be empty (light gray) or assigned to a particular generation, young or old. As requests for memory arrive, the memory manager hands out free regions. The memory manager assigns them to a generation and then returns them to the application as free space into which it can allocate.

Figure 7-1 G1 Garbage Collector Heap Layout



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

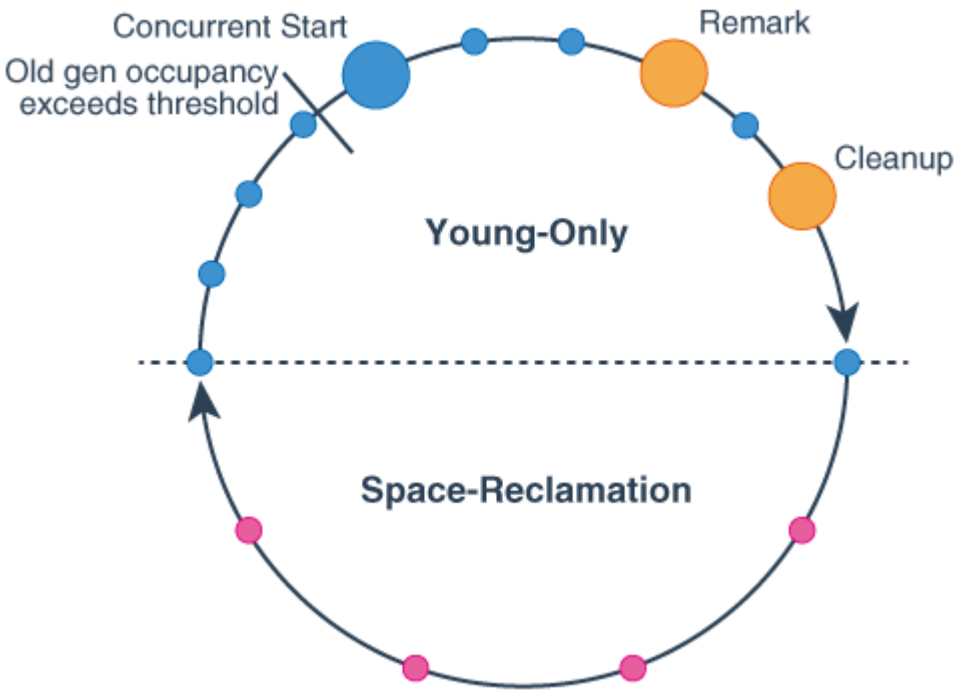
An application always allocates into a young generation, that is, eden regions, with the exception of humongous objects that are directly allocated as belonging to the old generation.

Garbage Collection Cycle

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

Figure 7-2 gives an overview about this cycle with an example of the sequence of garbage collection pauses that could occur:

Figure 7-2 Garbage Collection Cycle Overview



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

1. Young-Only phase: This phase starts with a few Normal young collections that promote objects into the old generation. The transition between the Young-Only phase and the Space-Reclamation phase starts when the old generation occupancy reaches a certain threshold, the Initiating Heap Occupancy threshold. At this time, G1 schedules a Concurrent Start young collection instead of a Normal young collection.
   - Concurrent Start : This type of collection starts the marking process in addition to performing a Normal young collection. Concurrent marking determines all currently reachable (live) objects in the old generation regions to be kept for the following Space-Reclamation phase. While marking hasn't completely finished, Normal young collections may occur. Marking finishes with two special stop-the-world pauses: Remark and Cleanup.

     The Concurrent Start pause may also determine that there is no need to follow through with marking: in this case, a short Concurrent Mark Undo phase occurs, and the Young-Only phase continues. In this case no Remark and Cleanup pauses will occur.
   - Remark: This pause finalizes the marking itself, performs reference processing and class unloading, reclaims completely empty regions and cleans up internal data structures. Between Remark and Cleanup G1 calculates information to later be able to reclaim free space in selected old generation regions concurrently, which will be finalized in the Cleanup pause.
   - Cleanup: This pause determines whether a Space-Reclamation phase will actually follow. If a Space-Reclamation phase follows, the Young-Only phase completes with a single Prepare Mixed young collection.
2. Space-Reclamation phase: This phase consists of multiple young collections that in addition to young generation regions, also evacuate live objects of sets of old generation regions. These collections are also called Mixed collections. The Space-Reclamation

phase ends when G1 determines that evacuating more old generation regions wouldn't yield enough free space worth the effort.

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

Garbage Collection Pauses and Collection Set

G1 performs garbage collections and space reclamation in stop-the-world pauses. Live objects are typically copied from source regions to one or more destination regions in the heap, and existing references to these moved objects are adjusted.

For non-humongous regions, the destination region for an object is determined from the source region of that object:

- Objects of the young generation (eden and survivor regions) are copied into survivor or old regions, depending on their age.
- Objects from old regions are copied to other old regions.

Objects in humongous regions are treated differently. G1 typically does not move these objects, but only determines their liveness, and if they are not live, reclaims the space they occupy. G1 only moves humongous objects in a very slow last-resort collection effort.

Remembered Set

To evacuate the collection set G1 manages a remembered set: the set of locations outside the collection set that contain references into the collection set. When an object from the collection set moves during garbage collection, any other references to that object from outside the collection set need to be changed to point to the new location of the object.

The remembered set entries represent approximate locations to save memory: often references close together reference objects close together, so that a single remembered set entry covers multiple locations. G1 uses cards to represent remembered set entries, small logical partitions of the heap. By default, these are 512 byte sized areas. Remembered set entries are compressed references of these cards.

G1 manages remembered sets on a per-region basis with the exception of the young generation: normally every region has its own remembered set, but young generation regions use a single remembered set for all young generation regions. It contains the locations with references to any young generation region.

The remembered sets are created mostly lazily: between the Remark and Cleanup pause G1 rebuilds the remembered set of all marking collection set candidate regions. G1 always maintains remembered sets for young generation regions as they are collected at every collection.

Collection Set

The collection set is the set of source regions to reclaim space during garbage collection. Independent of the garbage collection type, the collection set consists of different kinds of regions:

- Young generation regions,
- Humongous regions. See Humongous Objects about the restrictions,
- Collection set candidate regions. These are old generation regions that G1 determined to be good candidate regions for garbage collection due to their high collection efficiency.

This efficiency is calculated from the amount of free space, where regions with little live data are preferred over regions that contain mostly live data, and the connectivity to other regions, low connectivity being preferred over high connectivity.

There are two sources for old generation collection set candidate regions: from whole heap analysis, i.e., the marking, when G1 has good information about the liveness and connectivity of all old generation regions, and regions that experienced evacuation failure.

The former region's efficiencies have been determined directly using up-to-date liveness and connectivity data gathered in the preceding concurrent mark.

Regions that experienced evacuation failure very often contain very few objects. This makes them very efficient regions to collect, so they are made collection set candidate regions by default.

G1 discriminates between collection set candidate regions that are mandatory to collect in this garbage collection, and optional collection set candidate regions that will be garbage collected if time permits.

Garbage Collection Process

A garbage collection consists of four phases.

- The *Pre Evacuate Collection Set* phase performs some preparatory work for garbage collection: disconnecting TLABs from mutator threads, selecting the collection set for this collection as described in Java Heap Sizing, and other small preparatory work.
- During *Merge Heap Roots* G1 creates a single unified remembered set for later easier parallel processing from the collection set regions. This removes many duplicates from the individual remembered sets that would otherwise be needed to be filtered out later in a more expensive way.
- The *Evacuate Collection Set* phase represents the bulk of the work: G1 starts moving objects starting from the roots. A root reference is a reference from outside the collection set, either from some VM internal data structure (external roots), code (code roots) or from the remainder of the Java heap (heap roots, determined by the remembered sets). For all roots, G1 copies the referenced object in the collection set to its destination, processes its references into the collection set as new roots until there are no more roots.

  Individual timing for these phases can be observed with `-Xlog:gc+phases=debug` logging in the `Ext Root Scanning`, `Code Root Scan`, `Scan Heap Roots`, and `Object Copy` sub-phases respectively.

  G1 may optionally repeat main evacuation phases for optional collection sets.
- The *Post Evacuate Collection Set* consists of clean-up work including reference processing and setup for the following mutator phase.

These phases correspond to the phases printed with `-Xlog:gc+phases=info` logging.

Garbage-First Internals

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

Java Heap Sizing

G1 respects standard rules when resizing the Java heap, using `-XX:InitialHeapSize` as the initial Java heap size, `-XX:MaxHeapSize` as the maximum Java heap size, `-XX:MinHeapFreeRatio` for the minimum percentage of free memory, and `-XX:MaxHeapFreeRatio` for determining the maximum percentage of free memory after resizing.

The G1 collector resizes the Java heap according to these options during the Remark and Full GC pauses only. This process may release memory to or allocate memory from the operating system.

Heap expansion may occur within any garbage collection pause. If G1 determines that the Java heap should be shrunk, the release of this memory occurs after the pause concurrent with the application after the pause.

Young-Only Phase Generation Sizing

G1 determines an initial size for the young generation at the end of a normal young collection for the next mutator phase. As the mutator phase progresses, G1 refines this size estimate regularly.

The `-XX:GCPauseIntervalMillis` and `-XX:MaxGCPauseTimeMillis` options provide G1 with a minimum mutator utilization (MMU) to fit garbage collection activity into. For every possible time range of `-XX:GCPauseIntervalMillis`, G1 sizes the collection pauses to at most use `-XX:MaxGCPauseTimeMillis` milliseconds for garbage collection pauses. Information used for this calculation includes previous observations on how long it took young generations of similar size to evacuate, information on how many objects had to be copied during collection, and how interconnected these objects had been.

The options `-XX:G1NewSizePercent` and `-XX:G1MaxNewSizePercent` constrain the minimum and maximum eden size, which in turn constrain garbage collection pause times. The Garbage-First Garbage Collector Tuning guide provides some examples on how to decrease maximum pauses using these.

Alternatively, `-XX:NewSize` in combination with `-XX:MaxNewSize` may be used to set the minimum and maximum young generation sizes respectively.

At the start of a normal young collection, G1 selects additional old generation regions based on available time, as mentioned in the Collection Set section.

> **Note:**
>
> Only specifying one of these latter options to set eden size fixes young generation size to exactly the value passed with `-XX:NewSize` and `-XX:MaxNewSize` respectively. This disables pause time control.

At the start of a young collection, G1 selects additional old generation regions based on available time, as mentioned in the Collection Set section.

Space-Reclamation Phase Generation Sizing

During the Space-Reclamation phase, G1 tries to maximize the amount of space that will be reclaimed in the old generation in a single garbage collection pause. The size of the young generation is determined the same as for other Young-Only phase garbage collections, additionally taking the minimum set of old generation regions to take into the collection set into account.

At the start of every Mixed collection in this phase, G1 determines the collection set from regions as described in the Collection Set section. The amount of old generation regions in the collection set is determined as follows:

- A minimum set of old generation regions to ensure evacuation progress. This set of old generation regions is determined by the number of old generation regions determined as collection set candidates by the marking divided by the length of the Space-Reclamation phase as determined by `-XX:G1MixedGCCountTarget`.
- Additional old generation regions from the collection set candidates if G1 predicts that after collecting the minimum set there will be time left. Old generation regions are added until 80% of the remaining time is predicted to be used.
- A set of optional collection set regions that G1 evacuates incrementally after the other two parts have been evacuated and there is time left in this pause.

G1 collects the first two sets of regions in an initial garbage collection pass, with additional collection passes with regions from the optional collection set fit into the remaining pause time. This method ensures space reclamation progress while improving the probability to keep pause time and minimal overhead due to management of the optional collection set.

The Space-Reclamation phase ends when there are no more marking candidate regions in the collection set candidate regions set.

See Garbage-First Garbage Collector Tuning for more information about how many old generation regions G1 will use and how to avoid long mixed collection pauses.

Periodic Garbage Collections

If there has been no garbage collection for a long time because of application inactivity, the VM may unnecessarily hold on to a large amount of unused memory for a long time that could be used elsewhere. To avoid this, G1 can be forced to do regular garbage collection using the `-XX:G1PeriodicGCInterval` option during long idle periods. This option determines a minimum interval in milliseconds at which G1 considers performing a garbage collection after detecting the idle application state. If this amount of time passed since any previous garbage collection pause and there is no concurrent cycle in progress, G1 triggers additional garbage collections with the following possible effects:

- During the Young-Only phase: G1 starts a concurrent marking using a Concurrent Start pause or, if `-XX:-G1PeriodicGCInvokesConcurrent` has been specified, a Full GC.
- During the Space-Reclamation phase: G1 continues the space reclamation phase triggering the garbage collection pause type appropriate to current progress.

The `-XX:G1PeriodicGCSystemLoadThreshold` option should be used to refine what idle means for G1: if the average one-minute system load value as returned by the `getloadavg()` call on the JVM host system (for example, a container) is above this value, the VM is not considered idle and no periodic garbage collection will be run.

See JEP 346: Promptly Return Unused Committed Memory from G1 for more information about periodic garbage collections.

Determining Initiating Heap Occupancy

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

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

option-`XX:-G1UseAdaptiveIHOP`. In this case, the value of -`XX:InitiatingHeapOccupancyPercent` always determines this threshold.

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

Marking

G1 marking uses an algorithm called *Snapshot-At-The-Beginning (SATB)*. It takes a virtual snapshot of the heap at the time of the Concurrent Start pause. All objects that were live at the start of marking are considered live for the remainder of marking. This means that objects that become dead (unreachable) during marking are still considered live for the purpose of space reclamation (with some exceptions). This may cause some additional memory wrongly retained compared to other collectors. However, SATB potentially provides better latency during the Remark pause. The too conservatively considered live objects during that marking will be reclaimed during the next marking. See the Garbage-First Garbage Collector Tuning topic for more information about problems with marking.

Evacuation Failure

Evacuation failure means that G1 could not move some objects during garbage collection.

Such an occurrence is indicated in garbage collection logs with `-Xlog:gc` logging using an `Evacuation Failure: <reason>` printout where `<reason>` is one or both of `Allocation` and `Pinned` as indicated in the example below:

```
[9,740s][info ][gc] GC(26) Pause Young (Normal) (G1 Evacuation Pause) (Evacuation
Failure: Allocation/Pinned) 2159M->402M(3000M) 6,108ms
```

- `Allocation`: G1 could not find enough space in the destination area to move the object to.
- `Pinned`: There is an object that G1 could not move because G1 found an object that has been locked in place, or *pinned*, to allow safe use of native code on it using a `GetPrimitiveArrayCritical()` or similar JNI call. See JEP 423: Region Pinning for G1 for more information about pinning objects.

If G1 cannot move all objects out of a region, that region will be unavailable for allocation temporarily. G1 schedules these regions for immediate evacuation in the next garbage collections as collection set candidates.

In the worst case, if garbage collection does not manage to free any space at all during a garbage collection, G1 will schedule a Full GC. This type of garbage collection performs in-place compaction of the entire heap. This might be very slow.

See Garbage-First Garbage Collector Tuning for more information about problems with allocation failure or Full GCs before signaling out of memory.

Humongous Objects

Humongous objects are objects larger or equal the size of half a region. The current region size is determined ergonomically as described in the Ergonomic Defaults for G1 GC section, unless set using the `-XX:G1HeapRegionSize` option.

These humongous objects are sometimes treated in special ways:

- Every humongous object gets allocated as a sequence of contiguous regions in the old generation. The start of the object itself is always located at the start of the first region in that sequence. Any leftover space in the last region of the sequence will be lost for allocation until the entire object is reclaimed.
- Generally, humongous objects can be reclaimed only at the end of marking during the Remark pause, or during Full GC if they became unreachable. There is, however, a special provision for humongous objects for arrays of primitive types for example, `bool`, all kinds of integers, and floating point values. G1 opportunistically tries to reclaim humongous objects if they are not referenced by many objects at any garbage collection pause. This behavior is enabled by default but you can disable it with the option `-XX:-G1EagerReclaimHumongousObjects`.
- Allocations of humongous objects may cause garbage collection pauses to occur prematurely. G1 checks the Initiating Heap Occupancy threshold at every humongous object allocation and may force a Concurrent Start young collection immediately, if current occupancy exceeds that threshold and no marking is currently in progress.
- Humongous objects only move in a last-resort collection effort after a first Full GC failed to free enough contiguous memory for another humongous object allocation in a second Full GC in the same pause. This process is very slow. Due to space being unavailable for allocation in heap regions containing the end of humongous objects, it is still possible that G1 exits the VM with an out-of-memory condition.

Ergonomic Defaults for G1 GC

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

Table 7-1 Ergonomic Defaults G1 GC

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

Table 7-1 (Cont.) Ergonomic Defaults G1 GC

Option and Default Value	Description
`-XX:+G1UseAdaptiveIHOP` `-XX:InitiatingHeapOccupancyPercent=45`	Defaults for controlling the initiating heap occupancy indicate that adaptive determination of that value is turned on, and that for the first few collection cycles G1 will use an occupancy of 45% of the old generation as mark start threshold.
`-XX:G1HeapRegionSize=<ergo>`	The size of the heap regions. The default value is based on the maximum heap size and it is calculated to render roughly 2048 regions, with a maximum ergonomically determined value of 32 MB. A size given by the user must be a power of 2, and valid values range from 1 to 512 MB.
`-XX:G1NewSizePercent=5` `-XX:G1MaxNewSizePercent=60`	The size of the young generation in total, which varies between these two values as percentages of the current Java heap in use.
`-XX:G1HeapWastePercent=5`	The allowed unreclaimed space in the heap as a percentage of current total heap size. G1 stops adding old generation regions to the marking collection set candidates if the total free space after collecting these regions is lower than that value.
`-XX:G1MixedGCCountTarget=8`	The expected length of the Space-Reclamation phase in a number of collections.
`-XX:G1MixedGCLiveThresholdPercent=85`	Old generation regions with higher live object occupancy than this percentage will not be collected in a Space-Reclamation phase.

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

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

Comparison to Other Collectors

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

- Parallel GC can compact and reclaim space in the old generation only as a whole. G1 incrementally distributes this work across multiple much shorter collections. This substantially shortens pause time at the potential expense of throughput.
- G1 performs part of the old generation Space-Reclamation concurrently.
- G1 may exhibit higher overhead than the above collectors, affecting throughput due to its concurrent nature.
- ZGC aims to provide significantly smaller pause times at further cost of throughput.

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

- G1 can reclaim some kinds of humongous objects during any collection. This can avoid many otherwise unnecessary garbage collections, freeing a significant amount of space without much effort.
- G1 can optionally try to deduplicate duplicate strings on the Java heap concurrently.

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

8

Garbage-First Garbage Collector Tuning

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

Topics

- General Recommendations for G1
- Moving to G1 from Other Collectors
- Improving G1 Performance
  - Observing Full Garbage Collections
  - Humongous Object Fragmentation
  - Tuning for Latency
    - Unusual System or Real-Time Usage
    - Reference Object Processing Takes Too Long
    - Young-Only Collections Within the Young-Only Phase Take Too Long
    - Mixed Collections Take Too Long
    - High Merge Heap Roots and Scan Heap Roots Times
  - Tuning for Throughput
  - Tuning for Heap Size
  - Tunable Defaults

General Recommendations for G1

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

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

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

Moving to G1 from Other Collectors

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

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

Improving G1 Performance

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

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

The following subsections explore some common performance issues.

Observing Full Garbage Collections

A full heap garbage collection (Full GC) is often very time consuming. Full GCs caused by too high heap occupancy in the old generation can be detected by finding the words *Pause Full (G1 Compaction Pause)* in the log. Full GCs are typically preceded by garbage collections that encounter an evacuation failure with `Allocation` reason.

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

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

G1 gives you several options to handle this situation better:

- You can determine the number of regions occupied by humongous objects on the Java heap using the `gc+heap=info` logging. `Y` in the lines" `Humongous regions: X->Y`" give you the amount of regions occupied by humongous objects. If this number is high compared to the number of old regions, the best option is to try to decrease this number of objects. You can achieve this by increasing the region size using the `-XX:G1HeapRegionSize` option. The currently selected heap region size is printed at the beginning of the log.

- Increase the size of the Java heap. This typically increases the amount of time marking has to complete.
- Increase the number of concurrent marking threads by setting `-XX:ConcGCThreads` explicitly.
- Force G1 to start marking earlier. G1 automatically determines the Initiating Heap Occupancy Percent (IHOP) threshold based on earlier application behavior. If the application behavior changes, these predictions might be wrong. There are two options: Lower the target occupancy for when to start Space-Reclamation by increasing the buffer used in an adaptive IHOP calculation by modifying `-XX:G1ReservePercent`; or, disable the adaptive calculation of the IHOP by setting it manually using `-XX:-G1UseAdaptiveIHOP` and `-XX:InitiatingHeapOccupancyPercent`.

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

Humongous Object Fragmentation

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

Tuning for Latency

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

Unusual System or Real-Time Usage

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

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

Common known issues for high system time are:

- The VM allocating or giving back memory from the operating system memory may cause unnecessary delays. Avoid the delays by setting minimum and maximum heap sizes to the same value using the options `-Xms` and `-Xmx`, and pre-touching all memory using `-XX:+AlwaysPreTouch` to move this work to the VM startup phase.
- Particularly in Linux, coalescing of small pages into huge pages by the *Transparent Huge Pages (THP)* feature tends to stall random processes, not just during a pause. Because the VM allocates and maintains a lot of memory, there is a higher than usual risk that the

VM will be the process that stalls for a long time. Refer to the documentation of your operating system on how to disable the Transparent Huge Pages feature.

- Writing the log output may stall for some time because of some background task intermittently taking up all I/O bandwidth for the hard disk the log is written to. Consider using a separate disk for your logs or some other storage, for example memory-backed file system to avoid this. Another mitigation can be the use of asynchronous logging where the VM writes the log asynchronously using the `-Xlog:async` command line option.

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

Reference Object Processing Takes Too Long

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

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

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

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

Mixed Collections Take Too Long

Mixed young collections are used to reclaim space in the old generation. The collection set of Mixed collections contains young and old generation regions. You can obtain information about how much time evacuation of either young or old generation regions contribute to the pause-time by enabling the `gc+ergo+cset=debug` log output. Look for the following log message:

```
Added young regions to CSet. [...] predicted eden time: 4.86ms, predicted base
time: 9.98ms, target pause time: 200.00ms, [...]
```

Eden time and base time together give the predicted young region time, that is the time G1 expects evacuating the young generation will take

The log message for predicting old region time looks as follows:

```
Finish choosing collection set old regions. [...] predicted initial time:
147.70ms, predicted optional time: 15.45ms, [...]
```

Here, predicted initial time represents predicted old region time, i.e. the time G1 expects evacuating the minimum set of old generation regions will take.

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

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

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

Collections Occur Back to Back

G1 default MMU settings allow back-to-back garbage collections. The default value of `-XX:GCPauseIntervalMillis` is just slightly higher than `-XX:MaxGCPauseMillis`. In case you observe continuous back-to-back garbage collections, which results in the application not progressing, increase the value of `-XX:GCPauseIntervalMillis` to an acceptable value. G1 will then try to space out garbage collections more.

High Merge Heap Roots and Scan Heap Roots Times

One way to reduce these phases is to decrease the number of remembered set entries in the combined remembered sets. Adjusting the size of the heap regions by using the option `-XX:G1HeapRegionSize` decreases the number of cross-region references size of the remembered set. Larger regions tend to have fewer cross-region references, so the relative amount of work spent in processing them decreases, although at the same time, larger regions may mean more live objects to evacuate per region, increasing the time for other phases.

If a significant amount of time of the garbage collection, i.e. more than 60%, is spent in these two phases, one option could be decreasing the granularity of the remembered set entries by decreasing the value of the `-XX:GCCardSizeInBytes` option: finer granularity decreases the amount of work to find references, at the cost of some additional memory.

Spurious high Scan Heap Roots times in combination with the application allocating large objects may be caused by an optimization that tries to reduce concurrent remembered set updates work by batching them. If the application that created such a batch happens just before a garbage collection, this might have a negative impact on Merge Heap Roots time. Use `-XX:-ReduceInitialCardMarks` to disable this optimization and potentially avoid this situation.

Tuning for Throughput

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

particularly during the Space-Reclamation phase, increasing the minimum young generation size using `-XX:G1NewSizePercent` will force G1 to do that.

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

Another option to increase throughput is to decrease the amount of concurrent work. In particular, concurrent remembered set updates often require a lot of CPU resources. The option `-XX:G1RSetUpdatingPauseTimePercent` can be used to move work from concurrent operation into the garbage collection pause.

Increasing this value potentially decreases the refinement work scheduled concurrently to the application, conversely decreasing this value potentially increases the amount of refinement work performed concurrently to the application.

The refinement work in the garbage collection pause is tracked in the `Log Buffers` part of the `Merge Heap Roots` phase when enabling `gc+phases=debug` logging.

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

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

Tuning for Heap Size

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

Tunable Defaults

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

Table 8-1 Tunable Defaults G1 GC

Option and Default Value	Description
`-XX:+ReduceInitialCardMarks`	This batches together concurrent remembered set update (refinement) work for initial object allocations.
`-XX:+ParallelRefProcEnabled` `-XX:ReferencesPerThread=1000`	`-XX:ReferencesPerThread` determines the degree of parallelization: for every *N* Reference Objects one thread will participate in the sub-phases of Reference Processing, limited by `-XX:ParallelGCThreads`. A value of 0 indicates that the maximum number of threads as indicated by the value of `-XX:ParallelGCThreads` will always be used. This determines whether processing of `java.lang.Ref.*` instances should be done in parallel by multiple threads.

Table 8-1 (Cont.) Tunable Defaults G1 GC

Option and Default Value	Description
`-XX:G1RSetUpdatingPauseTimePercent=10`	The concurrent remembered set update (refinement) work can be controlled with this option. Refinement tries to schedule work concurrently so that at most `-XX:G1RSetUpdatingPauseTimePercent` percent of the maximum pause time goal is spent in the garbage collection pause in the Update RS phase, processing remaining work.
`-XX:G1SummarizeRSetStatsPeriod=0`	This is the period in a number of GCs that G1 generates remembered set summary reports. Set this to zero to disable.
`-XX:GCTimeRatio=12`	This is the divisor for the target ratio of time that should be spent in garbage collection as opposed to the application. The actual formula for determining the target fraction of time that can be spent in garbage collection before increasing the heap is `1 / (1 + GCTimeRatio)`. This default value results in a target with about 8% of the time to be spent in garbage collection.
`-XX:G1PeriodicGCInterval=0`	The interval in ms to check whether G1 should trigger a periodic garbage collection. Set to zero to disable.
`-XX:+G1PeriodicGCInvokesConcurrent`	If set, periodic garbage collections trigger a concurrent marking or continue the existing collection cycle, otherwise trigger a Full GC.
`-XX:G1PeriodicGCSystemLoadThreshold=0.0`	Threshold for the current system load as returned by the hosts `getloadavg()` call to determine whether a periodic garbage collection should be triggered. A current system load higher than this value prevents periodic garbage collections. A value of zero indicates that this threshold check is disabled.

Note:

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

9
The Z Garbage Collector

The *Z Garbage Collector (ZGC)* is a scalable low latency garbage collector. ZGC performs all expensive work concurrently, without stopping the execution of application threads for more than a millisecond. It is suitable for applications which require low latency. Pause times are independent of the heap size that is being used. ZGC works well with heap sizes from a few hundred megabytes to 16TB.

ZGC has been designed to be adaptive and to require minimal manual configuration. During the execution of the Java program, ZGC dynamically adapts to the workload by resizing generations, scaling the number of GC threads, and adjusting tenuring thresholds. The main tuning knob is to increase the maximum heap size.

> **Note:**
>
> As of JDK 24 ZGC is a generational garbage collector. The `ZGenerational` option has been removed.

Topics

- Setting the Heap Size
- Returning Unused Memory to the Operating System
- Using Large Pages
  - Enabling Large Pages On Linux
  - Enabling Transparent Huge Pages On Linux

Setting the Heap Size

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

ZGC has another command-line option related to the heap size named `-XX:SoftMaxHeapSize`. It can be used to set a soft limit on how large the Java heap can grow. ZGC will strive to not grow beyond this limit, but is still allowed to grow beyond this limit up to the maximum heap size. ZGC will only use more than the soft limit if that is needed to prevent the Java application from stalling and waiting for the GC to reclaim memory. For example, with the command-line options `-Xmx5g -XX:SoftMaxHeapSize=4g` ZGC will use 4GB as the limit for its heuristics, but if it can't keep the heap size below 4GB it is still allowed to temporarily use up to 5GB.

Returning Unused Memory to the Operating System

By default, ZGC uncommits unused memory, returning it to the operating system. This is useful for applications and environments where memory footprint is a concern, but might have a negative impact on the latency of Java threads. You can disable this feature with the command-line option `-XX:-ZUncommit`. Furthermore, memory will not be uncommitted so that the heap size shrinks below the minimum heap size `(-Xms)`. This means this feature will be implicitly disabled if the minimum heap size `(-Xms)` is configured to be equal to the maximum heap size `(-Xmx)`.

You can configure an uncommit delay using `-XX:ZUncommitDelay=<seconds>` (default is 300 seconds). This delay specifies for how long memory should have been unused before it's eligible for uncommit.

> **Note:**
>
> Allowing the GC to commit and uncommit memory while the application is running could have a negative impact on the latency of Java threads. If extremely low latency is the main reason for running with ZGC, consider running with the same value for `-Xmx` and `-Xms`, and use `-XX:+AlwaysPreTouch` to page in memory before the application starts.

Using Large Pages

Configuring ZGC to use large pages will generally yield better performance (in terms of throughput, latency and start up time) and comes with no real disadvantage, except that it's slightly more complicated to setup. The setup process typically requires root privileges, which is why it's not enabled by default.

Enabling Large Pages On Linux

On Linux x86, large pages (also known as "huge pages") have a size of 2MB.

Let's assume you want a 16GB Java heap. That means you need 16GB / 2MB = 8192 huge pages.

The heap requires at least 16GB (8192 pages) of memory to the pool of huge pages. The heap along with other parts of the JVM will use large pages for various internal data structures (such as code heap and marking bitmaps). In this example you will reserve 9216 pages (18GB) to allow for 2GB of non-Java heap allocations to use large pages.

Configure the system's huge page pool to have the required number of pages (requires root privileges):

```
$ echo 9216 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

Note that the above command is not guaranteed to be successful if the kernel cannot find enough free huge pages to satisfy the request. Also note that it might take some time for the kernel to process the request. Before proceeding, check the number of huge pages assigned to the pool to make sure the request was successful and has completed.

```
$ cat /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
```

```
9216
```

Enabling Transparent Huge Pages On Linux

An alternative to using explicit large pages (as described previously) is to use transparent huge pages. Use of transparent huge pages is usually not recommended for latency sensitive applications because it tends to cause unwanted latency spikes. However, it might be worth experimenting with to see if or how your workload is affected by it.

> **Note:**
>
> On Linux, using ZGC with transparent huge pages enabled requires kernel >= 4.7.

Use the following options to enable transparent huge pages in the VM:

```
-XX:+UseLargePages -XX:+UseTransparentHugePages
```

These options tell the JVM to issue `madvise(..., MADV_HUGEPAGE)` calls for memory it maps, which is useful when using transparent huge pages in `madvise` mode.

To enable transparent huge pages, you also need to configure the kernel by enabling `madvise` mode.

```
$ echo madvise > /sys/kernel/mm/transparent_hugepage/enabled
```

ZGC uses `shmem` huge pages for the heap, so the following kernel setting also needs to be configured:

```
$ echo advise > /sys/kernel/mm/transparent_hugepage/shmem_enabled
```

It is important to check these kernel settings when comparing the performance of different GCs. Some Linux distributions forcefully enable transparent huge pages for private pages by configuring `/sys/kernel/mm/transparent_hugepage/enabled` to be set to `always`, while leaving `/sys/kernel/mm/transparent_hugepage/shmem_enabled` at the default `never`. In this case all GCs but ZGC will make use of transparent huge pages for the heap. See Transparent Hugepage Support for more information.

10
Other Considerations

This section covers other situations that affect garbage collection.

Topics

- Finalization and Weak, Soft, and Phantom References
- Explicit Garbage Collection
- Soft References
- Class Metadata

Finalization and Weak, Soft, and Phantom References

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

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

> **Note:**
>
> Finalization has been deprecated in JDK 9. It also has been deprecated for removal in JDK 18; see JEP 421: Deprecate Finalization for Removal.

Finalization

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

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

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

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

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

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

- JDK Flight Recorder:
  - The JDK Flight Recorder (JFR) event, `jdk.FinalizerStatistics`, identifies classes at run time that use finalizers. The event is enabled by default in the `default.jfc` and `profile.jfc` JFR configuration files. When enabled, JFR emits a `jdk.FinalizerStatistics` event for each instantiated class with a non-empty `finalize()` method. The event includes the class that overrides `finalize()`, that class's `CodeSource`, the number of times the class's finalizer has run, and the number of objects still on the heap (and not yet finalized). See Flight Recorder in *Java Platform, Standard Edition JDK Mission Control User Guide* for more information.

Migrating from Finalization

To avoid finalization, use one of the following techniques:

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

The try-with-Resources Statement

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

The Cleaner API

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

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

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

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

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

1. Defines a cleaning action class, `State`, which initializes the cleaning action and defines the cleaning action itself (by overriding the `State::run()` method).
2. Creates an instance of `Cleaner`.

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

Figure 10-1 CleanerExample

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

public class CleanerExample {

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

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

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

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

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

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

        // Make myObject1 unreachable
        myObject1 = null;

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

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

```
        System.out.println("-- Finished --");
    }
}
```

This example prints the following:

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

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

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

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

Reference-Object Types

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

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

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

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

- To maintain access to an object while still allowing it to be garbage collected if the system needs to free up memory (such as a cached value that can be regenerated if required)
- To determine and perhaps take some action when an object has reached a particular reachability level (in combination with the `ReferenceQueue` class)

Explicit Garbage Collection

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

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

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

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

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

Soft References

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

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

Class Metadata

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

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

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

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

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

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

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

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

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