Java 平台标准版
HotSpot 虚拟机垃圾回收调优指南

第 24 版
G16827-02
2025年4月

ORACLE

Java 平台标准版 HotSpot 虚拟机垃圾收集调整指南，版本 24

G16827-02

内容

5 可用的收集器

6 并行收集器

7 垃圾优先（G1）垃圾收集器

8 Garbage-First 垃圾收集器调优

9 Z 垃圾收集器

10 其他考虑因素

前言

这*Java 平台标准版 HotSpot 虚拟机垃圾回收调优指南*描述 Java HotSpot 虚拟机（Java HotSpot VM）中包含的垃圾收集方法，并帮助您确定哪一种方法最适合您的需求。

观众

本文档面向希望深入了解 Java HotSpot VM 垃圾收集器的 Java HotSpot VM 用户、应用程序开发者和系统管理员。本文档进一步分析并解决垃圾收集器常见问题，帮助应用程序满足用户需求。

文档可访问性

有关 Oracle 对可访问性的承诺的信息，请访问 Oracle 可访问性计划网站http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc。

访问 Oracle 支持

Oracle 客户对 Oracle 支持服务的访问和使用将遵守其 Oracle 适用服务订单中规定的条款和条件。

多元化与包容性

Oracle 致力于多元化和包容性。Oracle 尊重并重视多元化的员工队伍，这有助于提升思想领导力和创新能力。为了构建更具包容性的文化，为员工、客户和合作伙伴带来积极影响，我们正在努力从产品和文档中删除不敏感的术语。我们也深知，必须与客户现有技术保持兼容性，并确保随着 Oracle 产品和行业标准的不断发展，服务能够保持连续性。由于这些技术限制，我们仍在持续努力删除不敏感的术语，这需要时间和外部合作。

相关文件

有关详细信息，请参阅以下文档：

- 垃圾收集：Richard Jones 和 Rafael D Lins 编写的自动动态内存管理算法。
- 垃圾收集手册：自动内存管理的艺术（Chapman & Hall/CRC 应用算法和数据结构）

公约

本文档中使用以下文本约定：

习俗	意义
粗体	粗体表示与操作相关的图形用户界面元素，或文本或词汇表中定义的术语。
斜体	斜体表示书名、重点或您提供特定值的占位符变量。
等宽字体	等宽字体表示段落内的命令、URL、示例中的代码、屏幕上显示的文本或您输入的文本。

1

垃圾回收调优简介

从桌面上的小型应用程序到大型服务器上的 Web 服务，各种各样的应用程序都使用 Java 平台标准版 (Java SE)。为了支持这种多样化的部署，Java HotSpot VM 提供了多种垃圾收集器，每种垃圾收集器都旨在满足不同的需求。Java SE 会根据运行应用程序的计算机类型选择最合适的垃圾收集器。但是，这种选择可能并非适用于所有应用程序。具有严格性能目标或其他要求的用户、开发人员和管理员可能需要明确选择垃圾收集器并调整某些参数以达到所需的性能水平。本文档提供了帮助完成这些任务的信息。

首先，本文以串行、stop-the-world 垃圾收集器为例，介绍垃圾收集器的一般特性和基本调优选项。然后，本文将介绍其他垃圾收集器的具体特性，以及选择垃圾收集器时需要考虑的因素。

主题

什么是垃圾收集器?

垃圾收集器 (GC) 自动管理应用程序的动态内存分配请求。

垃圾收集器通过以下操作执行自动动态内存管理：

- 从操作系统分配内存并将内存返还给操作系统。
- 根据应用程序的请求将该内存分发给应用程序。
- 确定内存的哪些部分仍被应用程序使用。
- 回收未使用的内存以供应用程序重复使用。

Java HotSpot 垃圾收集器采用各种技术来提高这些操作的效率：

- 将分代清理与老化结合使用，将精力集中在堆中最有可能包含大量可回收内存区域的区域。
- 使用多个线程积极地使操作并行，或者在后台与应用程序并发执行一些长时间运行的操作。
- 尝试通过压缩活动对象来恢复更大的连续可用内存。

为什么垃圾收集器的选择很重要?

垃圾收集器的目的是将应用程序开发人员从手动动态内存管理中解放出来。开发人员无需再将分配与

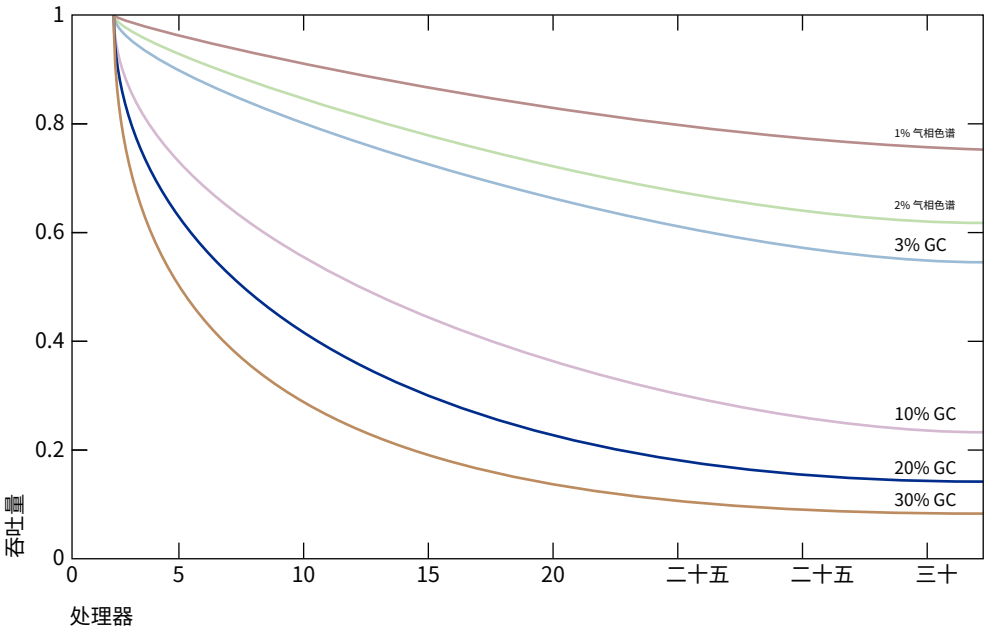
释放内存并密切关注已分配动态内存的生命周期。这完全消除了与内存管理相关的某些错误，但会付出一些额外的运行时开销。Java HotSpot VM 提供了多种垃圾收集算法供您选择。

垃圾收集器的选择在什么时候至关重要？对于某些应用程序来说，答案是永远无关紧要。也就是说，应用程序在垃圾收集的情况下，只要暂停频率和持续时间适中，就能表现良好。然而，对于大部分应用程序来说，情况并非如此，尤其是那些拥有大量数据（数 GB）、多线程和高事务率的应用程序。

阿姆达尔定律（给定问题的并行加速受问题的顺序部分限制）意味着大多数工作负载无法完美并行化；某些部分始终是顺序的，无法从并行性中获益。Java 平台目前支持四种垃圾收集方案，除了串行 GC 之外，其他所有方案都通过并行化工作来提升性能。尽可能降低垃圾收集的开销至关重要。以下示例展示了这一点。

图表图 1-1模拟了一个理想的系统，该系统除了垃圾收集之外，完全可扩展。红线表示一个应用程序在单处理器系统上仅花费 1% 的时间进行垃圾收集。这意味着在 32 个处理器的系统上，吞吐量损失超过 20%。洋红色线表示，对于一个垃圾收集时间占 10% 的应用程序（在单处理器应用程序中，垃圾收集时间并不算过长），扩展到 32 个处理器时，吞吐量损失超过 75%。

图 1-1 比较垃圾收集所花费的时间百分比



该图显示，当扩展到大型系统时，可忽略的吞吐量问题在小型系统上开发时，可能会影响系统性能。成为主要瓶颈，为了减少这种瓶颈，可以产生大型的ga然而，性能上的小改进并不重要。我们需要一系统，选择正确的gar变得值得 个足够大的 Bage 收集器，并在必要时进行调整。

串行收集器通常适用于大多数小型堆，单机容量可达 100 兆字节 所有应用程序，特别是那些需要 Dern 处理器的应用程序。其他收集器有

额外的开销或复杂性，这是特殊行为的代价。如果应用程序不需要替代收集器的特殊行为，请使用串行收集器。串行收集器并非最佳选择的一种情况是，大型、多线程的应用程序运行在拥有大量内存和两个或更多处理器的机器上。当应用程序在这样的服务器级机器上运行时，默认选择垃圾优先 (G1) 收集器；参见人体工程学。

文档中支持的操作系统

本文档及其建议适用于所有支持 JDK 24 的系统配置，但受特定配置中某些垃圾收集器实际可用性的限制。请参阅 Oracle JDK 认证系统配置。

2

人体工程学

*人体工程学*是 Java 虚拟机 (JVM) 和垃圾收集启发式方法（例如基于行为的启发式方法）提高应用程序性能的过程。

JVM 为垃圾收集器、堆大小和运行时编译器提供了与平台相关的默认选项。这些选项可以满足不同类型应用程序的需求，同时减少命令行调优的需要。此外，基于行为的调优功能可以动态优化堆大小，以满足应用程序的特定行为。

本节介绍这些默认选择和基于行为的调整。在使用后续章节中描述的更详细的控制之前，请使用这些默认设置。

主题

垃圾收集器、堆和运行时编译器默认选择

这些是重要的垃圾收集器、堆大小和运行时编译器默认选择：

- 服务器级机器上使用垃圾优先（G1）收集器，否则使用串行收集器。
- GC 线程的最大数量受堆大小和可用 CPU 资源的限制
- 初始堆大小为物理内存的 1/64
- 最大堆大小为物理内存的 1/4
- 分层编译器，同时使用 C1 和 C2

> **笔记：**
>
> 如果虚拟机检测到两个以上的处理器并且堆大小大于或等于 1792 MB，则虚拟机将机器视为服务器类。

基于行为的调整

Java HotSpot VM 垃圾收集器可以配置为优先满足以下两个目标之一：最大暂停时间和应用程序吞吐量。如果满足了优先目标，则

收集器会尝试最大化另一个目标。当然，这些目标并非总能实现：应用程序需要最小堆来容纳至少所有实时数据，而其他配置可能会阻碍部分或全部期望目标的实现。

最大暂停时间目标

这*暂停时间*是垃圾收集器停止应用程序并回收不再使用的空间的持续时间。*最大暂停时间*目标是限制这些暂停的最长时间。

垃圾收集器会维护一个平均暂停时间及其方差。该平均值从执行开始时计算，但会进行加权，以便越近的暂停计算得越多。如果平均值加上方差大于最大暂停时间目标，则垃圾收集器会认为未达到目标。

最大暂停时间目标是通过命令行选项指定的 - XX:MaxGCPauseMillis=*<nnn>*。这被解释为对垃圾收集器的提示，暂停时间为*<nnn>*毫秒或更短是理想的。垃圾收集器会调整 Java 堆大小和其他与垃圾收集相关的参数，以尽量使垃圾收集暂停时间小于*<nnn>*毫秒。最大暂停时间目标的默认值因垃圾收集器而异。这些调整可能会导致垃圾收集更频繁地发生，从而降低应用程序的整体吞吐量。但在某些情况下，所需的暂停时间目标无法达到。

吞吐量目标

吞吐量目标是根据垃圾收集所花费的时间来衡量的，垃圾收集之外所花费的时间是*申请时间*。

目标由命令行选项指定 -XX:GCTimeRatio=*嗯*。垃圾收集时间与申请时间之比为1/（1+*嗯*）。例如，-XX：GCTimeRatio = 19将垃圾收集时间设定为总时间的 1/20 或 5%。

垃圾收集所花费的时间是所有垃圾收集引起的暂停的总时间。如果吞吐量目标未达到，垃圾收集器可能采取的一种措施是增加堆的大小，以便应用程序在两次垃圾收集暂停之间所花费的时间更长。

脚印

如果吞吐量和最大暂停时间目标均已达到，垃圾收集器就会减少堆的大小，直到其中一个目标（通常是吞吐量目标）无法满足为止。垃圾收集器可以使用的最小和最大堆大小可以通过以下方式设置：Xms=*<nnn>*和 -Xmx=*<嗯>*分别为最小和最大堆大小。

调优策略

堆会增大或缩小到支持所选吞吐量目标的大小。了解堆调优策略，例如选择最大堆大小以及选择最大暂停时间目标。

除非您确定所需的堆大小大于默认最大堆大小，否则请勿为堆选择最大值。请选择一个足以满足您的应用程序需求的吞吐量目标。

应用程序行为的变化可能会导致堆增大或缩小。例如，如果应用程序开始以更高的速率分配内存，则堆会增大以维持相同的吞吐量。

如果堆增长到其最大大小后仍未达到吞吐量目标，则说明最大堆大小对于吞吐量目标而言太小。请将最大堆大小设置为接近平台总物理内存的值，但不会导致应用程序交换。然后再次执行该应用程序。如果仍然未达到吞吐量目标，则说明应用程序时间目标对于平台的可用内存而言过高。

如果吞吐量目标可以满足，但停顿时间过长，则应选择最大停顿时间目标。选择最大停顿时间目标可能意味着无法满足吞吐量目标，因此请选择应用程序可接受的折衷值。

通常情况下，当垃圾收集器试图满足相互竞争的目标时，堆的大小会波动。即使应用程序已经达到稳定状态，这种情况也会出现。实现吞吐量目标（可能需要更大的堆）的压力与最大暂停时间和最小占用空间的目标（这两者都可能需要较小的堆）相互竞争。

3

垃圾收集器实现

Java SE 平台的一个优势是它使开发人员免受内存分配和垃圾收集的复杂性的影响。

然而，当垃圾收集成为主要瓶颈时，了解其实现的某些方面会很有帮助。垃圾收集器会对应用程序使用对象的方式做出假设，这些假设反映在可调参数中，这些参数可以通过调整来提高性能，而不会牺牲抽象能力。

主题

分代垃圾收集

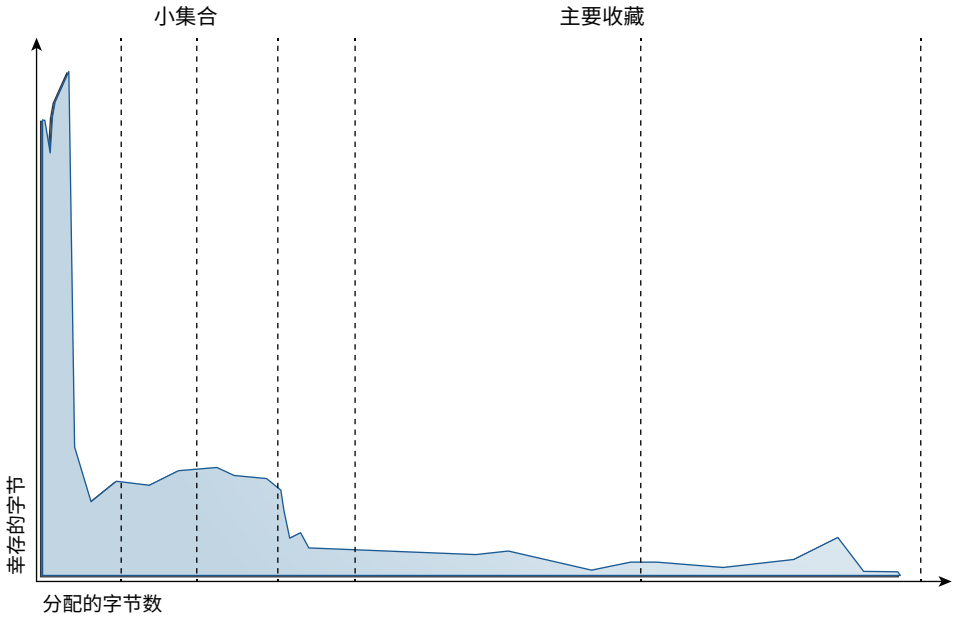
当正在运行的程序中任何其他活动对象的引用都无法再访问某个 对象时，该对象将被视为垃圾，并且其内存可由虚拟机重用。

理论上，最简单的垃圾收集算法每次运行时都会迭代所有可访问的对象。任何剩余的对象都被视为垃圾。这种方法所需的时间与活动对象的数量成正比，这对于维护大量活动数据的大型应用程序来说是难以承受的。

Java HotSpot VM 集成了多种不同的垃圾收集算法，这些算法使用一种称为分代收集的技术。简单垃圾收集每次都会检查堆中的每个活动对象，而分代收集则利用了大多数应用程序的一些经验性特性，以最大限度地减少回收未使用（垃圾）对象所需的工作量。这些观察到的特性中最重要的是*弱代际假说*，其中指出大多数物体只能存活很短的时间。

蓝色区域图 3-1这是对象生命周期的典型分布。x 轴表示对象生命周期（以分配的字节数为单位）。y 轴上的字节数表示相应生命周期的对象的总字节数。左侧的尖峰表示分配后不久即可回收（即“消亡”）的对象。例如，迭代器 对象通常仅在单个循环期间有效。

图 3-1 对象生命周期的典型分布



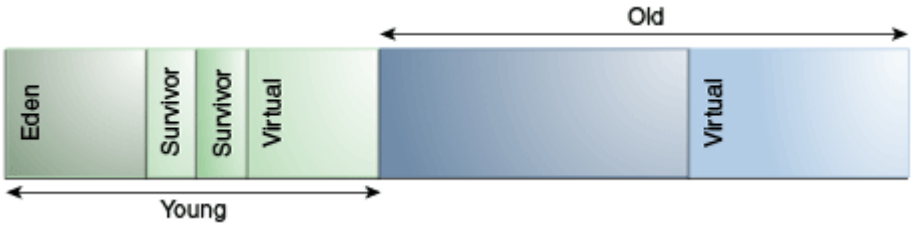
有些对象确实存活时间更长，因此分布会向右延伸。例如，通常有一些在初始化时分配的对象会一直存活到虚拟机退出。在这两个极端之间，有一些对象会存活一段时间，直到完成一些中间计算，这里我们将其视为初始峰值右侧的块状。有些应用程序的分布看起来非常不同，但令人惊讶的是，绝大多数应用程序都具有这种大致的形状。高效的垃圾收集是通过关注大多数对象“早逝”这一事实来实现的。

世代

为了优化垃圾收集，内存管理*世代*(内存池用于存放不同年龄的对象。当某一代内存填满时，就会进行垃圾收集。

绝大多数对象都分配在专门用于年轻对象的池中（*年轻一代*），大多数对象都会死在那里。当年轻代填满时，会导致*小集合*只回收年轻代中的垃圾；其他代中的垃圾不会被回收。这种回收的成本与被回收的存活对象数量成正比；充满死对象的年轻代回收速度非常快。通常，年轻代中存活的部分对象会被移动到 *老一辈*每次小垃圾收集期间。最终，老年代会被填满，必须被收集，导致*主要收藏*，其中收集整个堆。主收集通常比次收集持续时间长得多，因为涉及的对象数量要多得多。图 3-2显示了串行垃圾收集器中各代的默认排列：

图 3-2 串行收集器中默认的世代排列



Java HotSpot VM 启动时，会在地址空间中保留整个 Java 堆，但除非需要，否则不会为其分配任何物理内存。覆盖 Java 堆的整个地址空间在逻辑上分为年轻代和老年代。为对象内存保留的完整地址空间可以分为年轻代和老年代。

年轻代由伊甸园 (Eden) 和两个幸存者空间 (Survivor Space) 组成。大多数对象最初分配在伊甸园 (Eden) 中。其中一个幸存者空间在任何时候都是空的，在垃圾回收期间，它作为伊甸园 (Eden) 和另一个幸存者空间中存活对象的目的地；垃圾回收结束后，伊甸园 (Eden) 和源幸存者空间将变为空。在下一次垃圾回收中，两个幸存者空间的用途将互换。最近被填满的那个幸存者空间将成为存活对象的来源，这些存活对象会被复制到另一个幸存者空间。对象以这种方式在幸存者空间之间复制，直到它们被复制了一定次数或那里没有足够的空间。这些对象会被复制到老生代 (Old Generation) 区域。这个过程也称为*老化*。

性能考虑

垃圾收集的主要衡量指标是吞吐量和延迟。

- *吞吐量*是指长期来看未用于垃圾收集的总时间占比。吞吐量包括分配时间（但通常不需要调整分配速度）。

- *延迟*是应用程序的响应能力。垃圾收集暂停会影响应用程序的响应能力。

用户对垃圾回收的要求各不相同。例如，有些人认为 Web 服务器的正确指标是吞吐量，因为垃圾回收期间的停顿可能可以忍受，或者只是被网络延迟所掩盖。然而，在交互式图形程序中，即使是短暂的停顿也可能对用户体验产生负面影响。

一些用户对其他考虑因素很敏感。*脚印*是进程的工作集，以页面和缓存行数为单位。在物理内存有限或进程较多的系统上，占用空间可能决定可扩展性。*迅速*是对象死亡和内存可用之间的时间，这是分布式系统（包括远程方法调用 (RMI)）的一个重要考虑因素。

一般来说，选择特定代的大小需要在这些考虑因素之间进行权衡。例如，一个非常大的年轻代可以最大化吞吐量，但这是以牺牲内存占用、响应速度和暂停时间为代价的。使用较小的年轻代可以最小化年轻代的暂停时间，但会牺牲吞吐量。一个代的大小不会影响另一个代的垃圾收集频率和暂停时间。

选择代的大小没有唯一正确的方法。最佳选择取决于应用程序使用内存的方式以及用户需求。因此，虚拟机的

垃圾收集器的选择并不总是最佳的，并且可能会被命令行选项覆盖；参见影响垃圾回收性能的因素。

吞吐量和占用空间测量

最好使用特定于应用程序的指标来衡量吞吐量和占用空间。

例如，可以使用客户端负载生成器测试 Web 服务器的吞吐量。但是，通过检查虚拟机本身的诊断输出，可以轻松估计由于垃圾回收导致的 暂停。命令行选项 -详细：gc打印每次垃圾回收时堆和垃圾回收的相关信息。以下是示例：

[15,651 秒][info][gc] GC(36) 暂停年轻（G1 疏散暂停）239M->57M(307M)（15,646 秒，15,651 秒）5,048 毫秒
[16,162 秒][info][gc] GC(37) 暂停年轻（G1 疏散暂停）238M->57M(307M)（16,146 秒，16,162 秒）16,565 毫秒
[16,367 秒][信息][gc] GC(38) 暂停已满 (System.gc()) 69M->31M(104M) (16,202 秒，16,367 秒) 164,581 毫秒

输出显示两次年轻垃圾收集，随后是一次完整垃圾收集，由应用程序通过调用系统.gc()。每行以时间戳开头，指示应用程序启动的时间。接下来是有关此行日志级别 (info) 和标记 (gc) 的信息。之后是 GC 标识号。在本例中，有三个 GC，编号分别为 36、37 和 38。然后记录 GC 的类型和引发 GC 的原因。之后，记录一些有关内存消耗的信息。该日志使用“GC 前使用量”->“GC 后使用量”（“堆大小”）的格式。

示例的第一行是 239M->57M(307M)，这意味着在 GC 之前使用了 239 MB 内存，并且 GC 清理了大部分内存，但仍有 57 MB 剩余。堆大小为 307 MB。请注意，此示例中的完整 GC 将堆大小从 307 MB 缩小到 104 MB。在内存使用情况信息之后，会记录 GC 的开始和结束时间以及持续时间（结束时间 - 开始时间）。

这 -详细：gc命令是 - 的别名Xlog：gc。-Xlog是 HotSpot JVM 中日志记录的通用配置选项。它是一个基于标签的系统，其中气相色谱是标签之一。要获取有关 GC 正在执行的操作的更多信息，您可以配置日志记录以打印任何包含以下内容的消息气相色谱标签和任何其他标签。命令行选项为 -Xlog:gc*。

以下是 G1 年轻代收集记录的示例 -Xlog：gc*：

[10.178s][信息][gc，开始] GC(36) 暂停年轻进程（G1 疏散暂停）] GC(36) 使用 28
[10.178s][信息][gc，任务 个工作线程中的 28 个进行疏散] GC(36) 疏散前收集集：
[10.191s][信息][gc，阶段 0.0ms] GC(36) 疏散收集集：6.9ms] GC(36) 疏散后收集
[10.191s][信息][gc，阶段 集：5.9ms] GC(36) 其他：0.2ms
[10.191s][信息][gc，阶段
[10.191s][信息][gc，阶段
[10.191s][信息][gc，阶段] GC(36) 伊甸园区域：286->0(276)] GC(36) 幸
[10.191s][信息][gc，堆 [10.191s] 存者区域：15->26(38)] GC(36) 老年代区域：
[信息][gc，堆 [10.191s][信息] 88->88
[gc，堆 [10.191s][信息][gc，堆] GC(36) 巨型区域: 3->1
[10.191秒][信息][gc，元空间] GC(36) 元空间：8152K->8152K(1056768K) [10.191秒][信
息][gc] GC(36) 暂停年轻（G1 疏散暂停）391M-
> 114M(508M) 13.075毫秒
[10.191秒][信息][gc，cpu] GC(36) 用户=0.20秒 系统=0.00秒 实际=0.01秒

> **笔记：**
>
> 输出格式如下：Xlog：gc*在未来的版本中可能会发生变化。

4

影响垃圾回收性能的因素

影响垃圾收集性能的两个最重要的因素是总可用内存和专用于年轻代的堆比例。

主题

总堆

影响垃圾回收性能的最重要因素是总可用内存。由于垃圾回收发生在代填满时，因此吞吐量与内存大小相关。

> **笔记：**
>
> 以下关于堆的增长和收缩、堆布局以及默认值的讨论以串行收集器为例。虽然其他收集器也使用类似的机制，但此处介绍的细节可能不适用于其他收集器。有关其他收集器的类似信息，请参阅相应的主题。

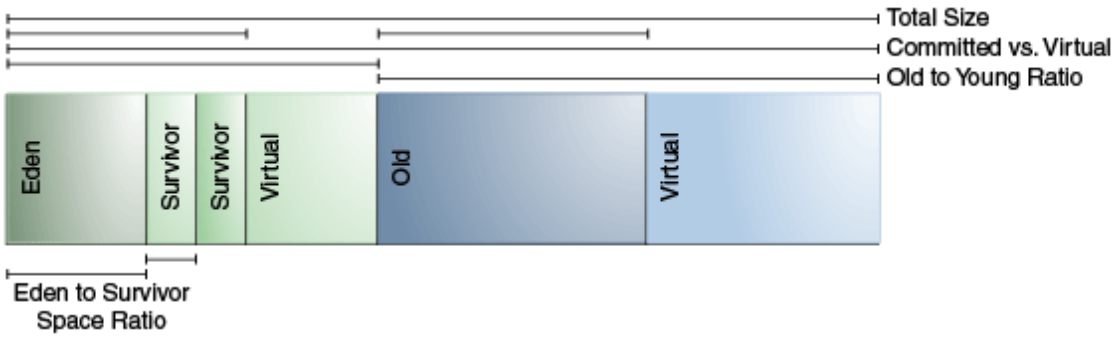
影响生成大小的堆选项

许多选项都会影响生成的大小。图 4-1说明了堆中已提交空间和虚拟空间之间的区别。在虚拟机初始化时，堆的整个空间都被保留。保留空间的大小可以通过

-Xmx选项。如果值为 -Xms参数小于 - 的值Xmx 参数，那么并非所有预留的空间都会立即提交给虚拟机。图中未提交的空间标记为“虚拟”。堆的不同部分，即老生代和年轻代，可以根据需要增长到虚拟空间的极限。

有些参数是堆的一部分与另一部分的比例。例如，参数 –XX：新比率表示老一代相对于年轻一代的相对大小。

图 4-1 堆选项



堆大小的默认选项值

默认情况下，虚拟机在每次收集时都会增大或缩小堆，以尝试将每次收集时可用空间与活动对象的比例保持在特定范围内。

该目标范围通过以下选项设置为百分比 -XX:MinHeapFreeRatio=<*最低*>和 -XX:MaxHeapFreeRatio=<*最大值*>，总规模受以下约束 –Xms<分钟> 以上由 –Xmx<最大值>。

使用这些选项，如果某个代中的可用空间百分比低于 40%，则该代会扩展以维持 40% 的可用空间，直至达到该代允许的最大大小。同样，如果可用空间超过 70%，则该代会收缩，以便只有 70% 的空间可用，但要遵守该代允许的最小大小。

Java SE 中用于并行收集器的计算方法现在适用于所有垃圾收集器。部分计算方法用于确定 64 位平台的最大堆大小上限。请参阅并行收集器默认堆大小。客户端 JVM 也有类似的计算，这会导致最大堆大小比服务器 JVM 要小。

以下是有关服务器应用程序堆大小的一般准则：

- 除非您遇到暂停问题，否则请尝试为虚拟机分配尽可能多的内存。默认大小通常太小。
- 环境 -Xms和 -Xmx设置为相同的值可以通过从虚拟机中移除最重要的大小决策来提高可预测性。但是，如果您做出了错误的选择，虚拟机将无法进行补偿。
- 一般来说，随着处理器数量的增加，内存也应该增加，因为分配可以并行进行。

通过最小化 Java 堆大小来节省动态内存占用

如果您需要最小化应用程序的动态内存占用（执行期间消耗的最大 RAM），可以通过最小化 Java 堆大小来实现。Java SE Embedded 应用程序可能需要这样做。

通过降低选项值来最小化 Java 堆大小 -XX:MaxHeapFreeRatio（默认值为 70%）和 -XX:MinHeapFreeRatio（默认值为 40%），命令行选项为 -XX:最大堆空闲率和 -XX:MinHeapFreeRatio。降低 -XX:最大堆空闲率 低至 10% 和 -XX:最小堆空闲率已证明可以成功减少堆大小，而不会造成太多性能下降；然而，结果可能会有很大差异，具体取决于

您的应用程序。尝试不同的参数值，直到它们尽可能低，但仍能保持可接受的性能。

在串行 GC 中，您可以指定 -XX:-ShrinkHeapInSteps，这会立即将 Java 堆减少到目标大小（由参数指定 -XX:MaxHeapFreeRatio）。使用此设置可能会导致性能下降。否则，Java 运行时会逐步将 Java 堆减少到目标大小；此过程需要多个垃圾回收周期。

年轻一代

除总可用内存之外，影响垃圾收集性能的第二个最重要的因素是专用于年轻代的堆比例。

年轻代越大，小垃圾收集发生的频率就越低。然而，对于有限的堆大小，较大的年轻代意味着较小的老年代，这会增加大垃圾收集的频率。最佳选择取决于应用程序分配的对象的生命周期分布。

年轻代大小选项

默认情况下，年轻代的大小由以下选项控制 -XX：新比率。

例如，设置 -XX：新比率=3意味着年轻代和老年代的比例是 1:3。换句话说，Eden 区和 Survivor 区的总大小将是堆总大小的四分之一。

选项 -XX:新尺寸和 -XX：最大新尺寸限制年轻代的大小。将这些值设置为相同的值可以固定年轻代的大小，就像设置 -Xms 和 -Xmx设置为相同的值可以固定堆的总大小。这对于以比整数倍更细的粒度调整年轻代很有用 -XX：新比率。

幸存者空间大小

您可以使用以下选项 -XX:幸存者比例调整幸存者空间的大小，但这通常对性能来说并不重要。

例如， -XX:幸存者比例=6将 eden 区和 survivor 区的比例设置为 1:6。换句话说，每个 survivor 区的大小将是 eden 区的六分之一，也就是年轻代大小的八分之一（而不是七分之一，因为有两个 survivor 区）。

如果幸存者空间太小，复制收集会直接溢出到老年代。如果幸存者空间太大，它们就会空空如也，毫无意义。每次垃圾收集时，虚拟机都会选择一个阈值，即对象在老年代之前可以被复制的次数。选择这个阈值是为了保持幸存者空间半满。您可以使用日志配置 -Xlog：gc，年龄可以用来显示此阈值以及新生代中对象的年龄。它对于观察应用程序的生命周期分布也很有用。

表 4-1提供幸存者空间大小的默认值。

表 4-1 Survivor 空间大小的默认选项值

选项	默认值
-XX：新比率	2

表 4-1（续）幸存者空间大小的默认选项值

选项	默认值
XX：新尺寸	1310 MB
-XX：最大新尺寸	不受限制
-XX：幸存者比例	8

年轻代的最大大小是根据整个堆的最大大小和以下值计算得出的 -XX：新比率参数。“不受限制”的默认值为 - XX：最大新尺寸参数意味着计算值不受 - 的限制XX：最大新尺寸 除非值 -XX：最大新尺寸在命令行上指定。

以下是服务器应用程序的一般准则：

- 首先确定虚拟机能够承受的最大堆大小。然后，绘制性能指标与年轻代大小的关系图，找到最佳设置。
  - 请注意，最大堆大小应始终小于机器上安装的内存量，以避免过多的页面错误和抖动。
- 如果堆内存总量固定，那么增加年轻代的大小就需要减少老年代的大小。保持老年代足够大，以便随时容纳应用程序使用的所有实时数据，并留出一些空闲空间（10% 到 20% 或更多）。
- 受前面提到的老一代的限制：
  - 为年轻一代提供充足的内存。
  - 随着处理器数量的增加，年轻代的大小也会增加，因为分配可以并行化。

5

可用的收集器

到目前为止，讨论的都是串行收集器。Java HotSpot VM 包含三种不同类型的收集器，每种都有不同的性能特征。

主题

串行收集器

串行收集器使用单个线程执行所有垃圾收集工作，这使得它相对高效，因为线程之间没有通信开销。

它最适合单处理器机器，因为它无法利用多处理器硬件的优势，尽管对于处理小数据集（最多约 100 MB）的应用程序，它在多处理器上很有用。在某些硬件和操作系统配置中，串行收集器是默认选择的，也可以使用以下选项显式启用 - XX:+使用串行GC。

并行收集器

并行收集器也称为*吞吐量收集器*它是一个类似于串行收集器的分代收集器。串行收集器和并行收集器之间的主要区别在于并行收集器具有多个线程，用于加快垃圾收集速度。

并行收集器适用于在多处理器或多线程硬件上运行的具有中型到大型数据集的应用程序。您可以使用以下命令启用它 - XX:+使用并行GC选项。

垃圾优先（G1）垃圾收集器

G1 是一款“主要并发”收集器。“主要并发”收集器会并发地执行一些对应用程序来说开销很大的工作。该收集器的设计目标是从小型计算机扩展到拥有大量内存的大型多处理器计算机。它能够以高概率满足暂停时间目标，同时实现高吞吐量。

在大多数硬件和操作系统配置中，G1 是默认选择的，或者可以使用以下方式明确启用 -XX:+使用G1GC。

Z 垃圾收集器

ZGC 提供的最大暂停时间低于一毫秒，但会牺牲一些吞吐量。它适用于需要低延迟的应用程序。暂停时间与正在使用的堆大小无关。ZGC 适用于几百兆字节到 16TB 的堆大小。要启用此功能，请使用 -XX:+使用ZGC选项。

选择收集器

除非您的应用程序有相当严格的暂停时间要求，否则请首先运行您的应用程序并允许虚拟机选择一个收集器。

如有必要，请调整堆大小以提高性能。如果性能仍然未达到您的目标，请参考以下准则来选择收集器：

- 如果应用程序的数据集较小（最多约 100 MB），则选择带有以下选项的串行收集器 -XX:+使用串行GC。
- 如果应用程序将在单个处理器上运行并且没有暂停时间要求，则选择带有以下选项的串行收集器 -XX:+使用串行GC。
- 如果 (a) 峰值应用程序性能是第一要务，并且 (b) 没有暂停时间要求，或者可以接受一秒或更长时间的暂停，那么让虚拟机选择收集器或选择并行收集器 -XX:+使用并行GC。
- 如果响应时间比整体吞吐量更重要，并且必须保持更短的垃圾收集暂停时间，那么选择最并发的收集器 -XX:+使用G1GC。
- 如果响应时间是高优先级，那么选择一个完全并发的收集器 -XX:+使用ZGC。

这些指导原则仅提供了选择收集器的起点，因为性能取决于堆的大小、应用程序维护的实时数据量以及可用处理器的数量和速度。

如果推荐的收集器未能达到预期性能，请先尝试调整堆和代的大小以达到预期目标。如果性能仍然不足，请尝试其他收集器：使用并发收集器来减少暂停时间，并使用并行收集器来提高多处理器硬件上的整体吞吐量。

6

并行收集器

并行收集器（此处也称为*吞吐量收集器*）是一种类似于串行收集器的分代收集器。串行收集器和并行收集器的主要区别在于，并行收集器使用多个线程来加速垃圾收集。

并行收集器通过命令行选项启用 -XX:+使用并行GC。默认情况下，使用此选项，小集合和大集合将并行运行，以进一步减少垃圾收集开销。

主题

并行垃圾收集器线程数

在一台机器上*<N>*硬件线程*<N>*大于 8，并行收集器使用固定比例的*<N>*作为垃圾收集器线程的数量。

对于较大的值，分数约为 5/8*<N>*. 值为*<N>*低于 8，使用的数字是*<N>*在某些平台上，该比例会降至 5/16。具体的垃圾收集器线程数可以通过命令行选项进行调整（稍后介绍）。在单处理器主机上，由于并行执行所需的开销（例如同步），并行收集器的性能可能不如串行收集器。但是，在运行具有中型到大型堆的应用程序时，在双处理器计算机上，并行收集器的性能通常会略优于串行收集器；而在有两个以上处理器可用的计算机上，并行收集器的性能通常会明显优于串行收集器。

垃圾收集器线程的数量可以通过命令行选项控制 - XX:ParallelGCThreads=*<N>*如果您使用命令行选项调整堆，那么并行收集器获得良好性能所需的堆大小与串行收集器相同。但是，启用并行收集器应该可以缩短收集暂停时间。由于多个垃圾收集器线程参与小垃圾收集，因此在收集过程中，由于从年轻代到老年代的提升，可能会产生一些碎片。每个参与小垃圾收集的垃圾收集线程

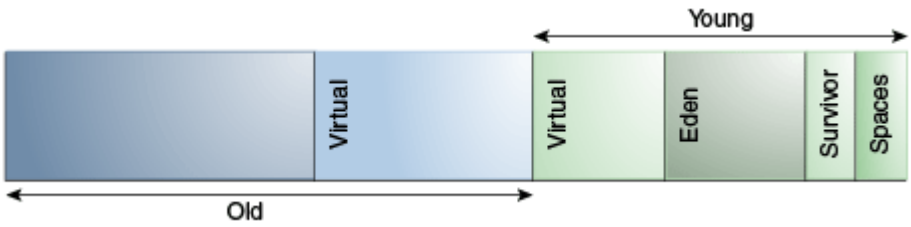
垃圾收集器会保留一部分老生代空间用于提升，而将可用空间划分为这些“提升缓冲区”可能会导致碎片化。减少垃圾收集器线程数量并增加老生代空间的大小可以减轻这种碎片化的影响。

并行收集器中的分代安排

在并行收集器中，各代的排列方式有所不同。

该安排显示在图 6-1：

图 6-1 并行收集器中的分代排列



平行收集器人体工程学

当使用以下方式选择并行收集器时 -XX:+使用并行GC，它支持一种自动调整方法，允许您指定行为而不是生成大小和其他低级调整细节。

指定并行收集器行为的选项

您可以指定最大垃圾收集暂停时间、吞吐量和占用空间（堆大小）。

- 最大垃圾收集暂停时间：最大暂停时间目标是通过命令行选项指定的 -XX:MaxGCPauseMillis=<*N*>。这被解释为一个提示，暂停时间<*N*>期望的暂停时间不超过毫秒；默认情况下，没有最大暂停时间目标。如果指定了暂停时间目标，则会调整堆大小和其他与垃圾收集相关的参数，以尝试使垃圾收集暂停时间短于指定值；但是，期望的暂停时间目标可能并不总是能够实现。这些调整可能会导致垃圾收集器降低应用程序的整体吞吐量。

- 吞吐量：吞吐量目标是根据垃圾收集所花费的时间与垃圾收集之外所花费的时间进行衡量的，称为*申请时间*。目标由命令行选项指定 -XX:GCTimeRatio=<*N*>，将垃圾收集时间与应用程序时间的比率设置为1 / (1 +<*N*>）。

  例如， -XX：GCTimeRatio = 19将垃圾收集时间的目标设置为 1/20，即总时间的 5%。默认值为 99，即垃圾收集时间的目标为 1%。

- 占用空间：使用选项指定最大堆占用空间 -Xmx<*N*>。此外，只要满足其他目标，收集器就有一个隐含的目标，即最小化堆的大小。

并行收集器目标的优先级

这些目标是最大暂停时间目标、吞吐量目标和最小占用空间目标，并按以下顺序实现目标：

首先要满足最大暂停时间目标。只有满足此目标后，才会考虑吞吐量目标。同样，只有满足前两个目标后，才会考虑占用空间目标。

并行收集器生成大小调整

收集器保存的平均暂停时间等统计数据会在每次收集结束时更新。

然后进行测试以确定目标是否已实现，并对代的大小进行任何必要的调整。例外情况是显式垃圾回收，例如，调用系统.gc()在保存统计数据和调整代数大小方面被忽略。

代的大小增长和收缩是通过代大小的固定百分比增量来实现的，这样代就会逐步增加或减少到其期望的大小。增长和收缩的速率不同。默认情况下，代的增长增量为 20%，收缩增量为 5%。增长百分比由命令行选项控制 -XX:YoungGenerationSizeIncrement=*<是>*对于年轻一代来说——XX:TenuredGenerationSizeIncrement=*<T>*对于老一代。一代收缩的百分比可以通过命令行标志来调整 - XX:AdaptiveSizeDecrementScaleFactor=*<D>*. 如果增长增量是 *十*%，则收缩的减量为 *十*/*D* %。

如果收集器在启动时决定增加某个代，则会在增量中添加一个补充百分比。此补充百分比会随着收集次数的增加而衰减，并且不会产生长期影响。此补充百分比的目的是提高启动性能。收缩时不会对百分比进行补充。

如果未达到最大暂停时间目标，则每次仅缩小一个代的规模。如果两个代的暂停时间都超过了目标，则首先缩小暂停时间较长的代的规模。

如果吞吐量目标未达到，则两个代的大小都会增加。每个代的大小都会根据其对总垃圾收集时间的贡献按比例增加。例如，如果年轻代的垃圾收集时间占总收集时间的 25%，并且年轻代的完整增量为 20%，则年轻代的大小将增加 5%。

并行收集器默认堆大小

除非在命令行中指定了初始和最大堆大小，否则它们将根据计算机的内存量计算。默认最大堆大小为物理内存的四分之一，而初始堆大小为物理内存的 1/64。分配给年轻代的最大空间量是总堆大小的三分之一。

并行收集器初始和最大堆大小的规范

您可以使用以下选项指定最小和最大堆大小 -Xms（最小堆大小）和 -Xmx（最大堆大小）。

如果您知道应用程序需要多少堆才能正常运行，那么您可以设置 -Xms和 - Xmx设置为相同的值。如果您不知道，JVM 会先使用初始堆大小，然后逐渐增大 Java 堆，直到找到堆使用率和性能之间的平衡。

其他参数和选项可能会影响这些默认值。要验证您的默认值，请使用 - XX:+PrintFlagsFinal选项并寻找 -XX:最大堆大小在输出中。例如，在 Linux 上，您可以运行以下命令：

java -XX:+PrintFlagsFinal*<GC 选项>*-版本 | grep MaxHeapSize

并行收集器时间过长和 OutOfMemoryError

并行收集器抛出一个内存不足错误如果在垃圾收集（GC）上花费了太多时间。

如果总时间的 98% 以上用于垃圾收集，并且回收的堆不到 2%，那么内存不足错误，抛出。此功能旨在防止应用程序因堆太小而长时间运行，却进展缓慢或完全没有进展。如有必要，可以通过添加以下选项来禁用此功能 - XX:-UseGCOverheadLimit到命令行。

并行收集器测量

并行收集器的详细垃圾收集器输出与串行收集器的详细垃圾收集器输出基本相同。

7

垃圾优先（G1）垃圾收集器

本节介绍垃圾优先（G1）垃圾收集器（GC）。

主题

Garbage-First（G1）垃圾收集器简介

Garbage-First (G1) 垃圾收集器专为能够扩展到大内存的多处理器机器而设计。它力求以极高的概率满足垃圾收集暂停时间目标，同时以极低的配置实现高吞吐量。G1 旨在利用当前目标应用程序和环境，在延迟和吞吐量之间实现最佳平衡，其功能包括：

- 堆大小高达数十 GB 或更大，其中 50% 以上的 Java 堆被实时数据占用。
- 对象分配和提升的速度可能会随时间发生很大变化。
- 堆中存在大量碎片。

- 可预测的暂停时间目标不超过几百毫秒，避免长时间的垃圾收集暂停。

G1 在应用程序运行时同时执行部分工作。它用原本可供应用程序使用的处理器资源来换取更短的收集暂停时间。

这在应用程序运行时使用一个或多个活跃的垃圾收集线程时最为明显。因此，与吞吐量收集器相比，虽然 G1 收集器的垃圾收集暂停时间通常要短得多，但应用程序的吞吐量也往往略低。

G1 是默认收集器。

G1 收集器实现了高性能，并尝试通过以下章节中描述的几种方式来满足暂停时间目标。

启用 G1

Garbage-First 垃圾收集器是默认收集器，因此通常无需执行任何其他操作。您可以通过提供以下信息显式启用它：XX:+使用G1GC在命令行上。

基本概念

G1 是一个分代式、增量式、并行、主要并发、支持 Stop-the-World 和清除垃圾收集器，它会监控每次 Stop-the-World 暂停时的暂停时间目标。与其他收集器类似，G1 将堆划分为（虚拟的）年轻代和老年代。空间回收工作主要集中在效率最高的年轻代，偶尔也会在老年代进行空间回收。

为了提高吞吐量，某些操作总是在 Stop-the-world 暂停期间执行。其他操作在应用程序停止时会花费更多时间，例如全堆操作，例如*全球标记*与应用程序并行并发执行。为了缩短空间回收的 Stop-the-World 暂停时间，G1 以逐步并行的方式执行空间回收。G1 通过跟踪应用程序先前行为和垃圾收集暂停的信息来构建相关成本模型，从而实现可预测性。它使用这些信息来调整暂停期间完成的工作量。例如，G1 首先回收效率最高的区域（即垃圾填充最多的区域，因此得名）。

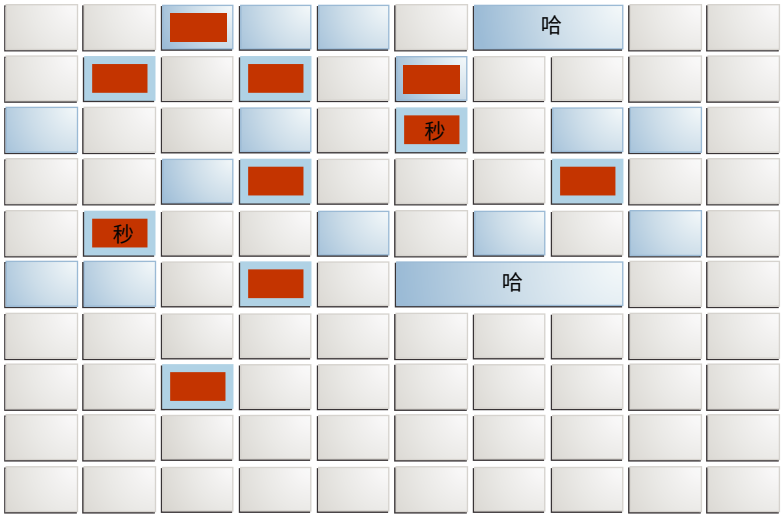
G1 主要通过“撤离”机制回收空间：在选定的待回收内存区域内找到的存活对象会被复制到新的内存区域，并在此过程中进行压缩。撤离完成后，先前存活对象占用的空间将被应用程序重新利用，用于分配。

Garbage-First 收集器并非实时收集器。它会尝试在较长时间内以较高的概率满足设定的暂停时间目标，但对于给定的暂停，并非总是绝对确定。

堆布局

G1 将堆划分为一组大小相等的堆区域，每个区域都是一段连续的虚拟内存，如图 7-1 所示。区域是内存分配和内存回收的单位。在任何给定时间，这些区域可以是空的（浅灰色），也可以分配给特定的代（年轻代或老代）。当内存请求到达时，内存管理器会释放空闲区域。内存管理器将它们分配给某一代，然后将它们作为空闲空间返回给应用程序，应用程序可以将其分配到其中。

图 7-1 G1 垃圾收集器堆布局



年轻代包含伊甸园区域（红色）和幸存者区域（红色带“S”）。这些区域的功能与其他收集器中相应的连续空间相同，不同之处在于，在 G1 中，这些区域通常以非连续的模式布局在内存中。老生代区域（浅蓝色）构成老生代。对于跨多个区域的对象，老生代区域可能非常巨大（浅蓝色带“H”）。

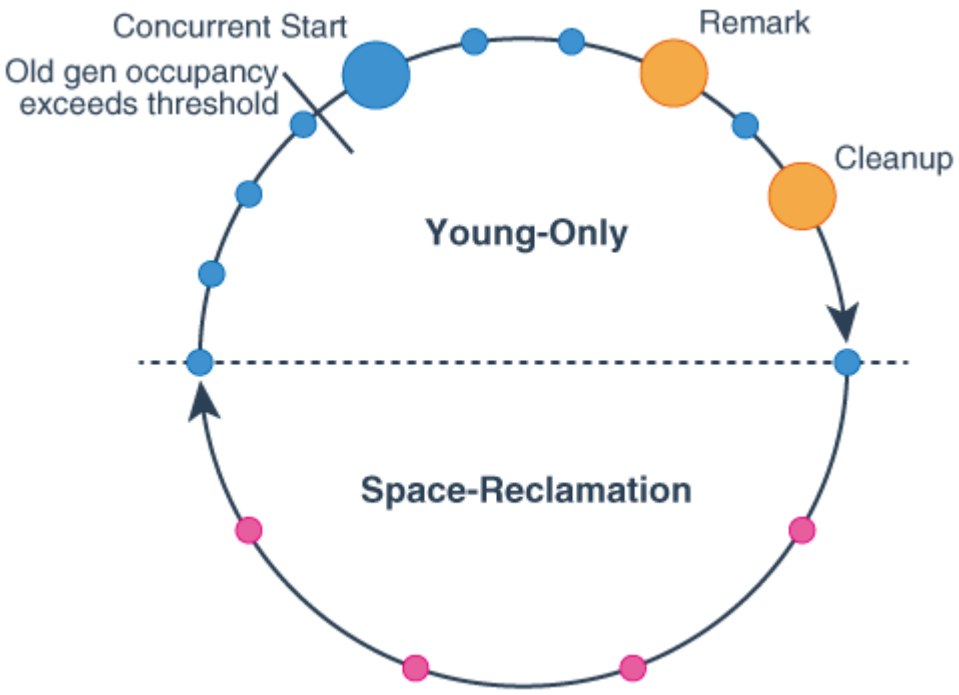
应用程序总是分配到年轻代，即伊甸园区域，但直接分配为属于老一代的巨大对象除外。

垃圾收集周期

从高层次上讲，G1 收集器在两个阶段之间交替运行。Young-Only 阶段包含垃圾收集，它会逐渐用老生代中的对象填满当前可用的内存。Space-Reclamation 阶段是 G1 除了处理年轻代空间之外，还会逐步回收老生代的空间。然后，循环从 Young-Only 阶段重新开始。

图 7-2 给出了该周期的概述，并给出了可能发生的垃圾收集暂停序列的示例：

图 7-2 垃圾收集周期概览



以下列表详细描述了 G1 垃圾收集周期的各个阶段、暂停以及阶段之间的转换：

1.Young-Only 阶段：此阶段以一些普通的年轻代垃圾收集开始，这些垃圾收集会将对象提升到老年代。当老年代占用率达到某个阈值（即初始堆占用率阈值）时，Young-Only 阶段和 Space-Reclamation 阶段之间的转换就开始了。此时，G1 会调度并发启动年轻代垃圾收集，而不是普通的年轻代垃圾收集。

   - 并发启动：这种类型的收集除了执行普通的年轻代收集外，还会启动标记过程。并发标记会确定老生代区域中所有当前可访问（存活）的对象是否要保留，以备后续的空间回收阶段使用。在标记尚未完全完成时，可能会进行普通的年轻代收集。标记过程以两个特殊的“暂停世界”阶段结束：重新标记和清理。

     并发启动暂停也可能决定无需继续进行标记：在这种情况下，会有一个短暂的并发标记撤销阶段，然后继续进入 Young-Only 阶段。在这种情况下，不会发生重新标记和清理暂停。

   - 标记：此暂停会完成标记本身，执行引用处理和类卸载，回收完全空的区域并清理内部数据结构。在标记和清理之间，G1 会计算信息，以便稍后能够同时回收选定老生代区域中的可用空间，这些操作将在清理暂停中完成。

   - 清理：此暂停决定是否真正进入空间回收阶段。如果进入空间回收阶段，则 Young-Only 阶段将以一次准备混合年轻代垃圾收集 (Prepare Mixed Young-Occupation) 结束。

2.空间回收阶段：此阶段包含多个年轻代收集，除了年轻代区域外，还会清理老生代区域中的存活对象。这些收集也称为混合收集。空间回收阶段

当 G1 确定撤离更多老一代区域不会产生足够的可用空间时，阶段结束。

空间回收 (Space-Reclamation) 结束后，收集周期将重新开始，进入新的 Young-Only 阶段。作为备份，如果应用程序在收集活跃度信息时内存耗尽，G1 会像其他收集器一样执行一次就地 Stop-the-World 的全堆压缩（Full GC）。

垃圾收集暂停和收集设置

G1 在 Stop-the-World 暂停期间执行垃圾收集和空间回收。活动对象通常会从源区域复制到堆中的一个或多个目标区域，并且对这些移动对象的现有引用会进行调整。

对于非巨型区域，对象的目标区域由该对象的源区域确定：

- 年轻一代（伊甸园和幸存者区域）的对象被复制到幸存者区域或老年代区域，取决于它们的年龄。
- 旧区域的对象被复制到其他旧区域。

巨型区域中的对象将受到不同的处理。G1 通常不会移动这些对象，而只会判断它们的存活状态，如果它们不存活，则回收它们占用的空间。G1 只会在非常缓慢的最后回收机制中移动巨型对象。

记忆集

为了清空回收集合，G1 管理着一个记忆集：该集合是回收集合外部包含对回收集合引用的位置的集合。当回收集合中的对象在垃圾回收期间移动时，回收集合外部对该对象的任何其他引用都需要更改为指向该对象的新位置。

记忆集条目表示近似位置以节省内存：通常引用彼此相邻的引用对象，因此单个记忆集条目可以覆盖多个位置。G1 使用卡片来表示记忆集条目，它们是堆的小型逻辑分区。默认情况下，这些区域大小为 512 字节。记忆集条目是这些卡片的压缩引用。

G1 以每个区域为单位管理记忆集，年轻代除外：通常每个区域都有自己的记忆集，但年轻代区域对所有年轻代区域使用一个记忆集。它包含指向任何年轻代区域的引用位置。

记忆集的创建大多是惰性的：在标记和清理暂停之间，G1 会重建所有标记收集候选区域的记忆集。G1 始终维护年轻代区域的记忆集，因为它们在每次收集时都会被回收。

收藏集

收集集是垃圾回收期间用于回收空间的源区域的集合。无论垃圾回收类型如何，收集集都包含不同类型的区域：

- 年轻一代地区，
- 巨大的区域。请参阅巨型物体关于限制，
- 垃圾收集集候选区域。这些是老生代区域，由于其较高的收集效率，G1 将其确定为垃圾收集的良好候选区域。

此效率是根据可用空间量计算得出的，其中，活动数据较少的区域优先于包含大部分活动数据的区域，并且与其他区域的连接性，低连接性优先于高连接性。

老生代收集集候选区域有两个来源：来自整个堆分析，即标记，当 G1 具有关于所有老生代区域的活跃度和连通性的良好信息时，以及经历疏散失败的区域。

前一个区域的效率是直接使用在前一个并发标记中收集的最新活性和连接性数据来确定的。

发生疏散失败的区域通常包含非常少的对象。这使得这些区域成为非常高效的收集区域，因此默认将其设为收集集候选区域。

G1 区分本次垃圾收集中必须收集的收集集候选区域，以及如果时间允许则将被垃圾收集的可选收集集候选区域。

垃圾收集过程

垃圾收集由四个阶段组成。

- 这*撤离前收集套件*阶段执行一些垃圾收集的准备工作：断开 TLAB 与变量线程的连接，选择本次收集的收集集合，具体描述如下Java 堆大小，以及其他一些小的准备工作。

- 期间*合并堆根*G1 会创建一个统一的记忆集，以便后续从集合区域进行并行处理。这会从各个记忆集中移除许多重复项，否则这些重复项需要稍后以更昂贵的方式过滤掉。

- 这*撤离收集装置*阶段代表了大部分工作：G1 开始从根开始移动对象。根引用是来自集合外部的引用，可以是来自虚拟机内部数据结构（外部根）、代码（代码根），也可以是来自 Java 堆的剩余部分（堆根，由记忆集决定）。对于所有根，G1 将集合中引用的对象复制到其目标位置，并将其引用作为新的根处理到集合中，直到不再有根。

  可以通过以下方式观察这些阶段的具体时间：Xlog：gc+阶段=调试登录扩展根扫描、代码根扫描、扫描堆根、和对象复制子阶段。

  G1 可以选择为可选的收集集重复主要疏散阶段。

- 这*撤离后收集套装*包括清理工作，包括参考处理和以下变量阶段的设置。

这些阶段对应于打印有 - 的阶段Xlog：gc+阶段=信息伐木。

垃圾优先内部结构

本节介绍 Garbage-First（G1）垃圾收集器的一些重要细节。

Java 堆大小

G1 在调整 Java 堆大小时遵循标准规则，使用 -XX：初始堆大小作为初始 Java 堆大小，-XX:最大堆大小作为最大 Java 堆大小，- XX:最小堆空闲率获得最小可用内存百分比，以及 - XX:最大堆空闲率用于确定调整大小后可用内存的最大百分比。

G1 收集器仅在 Remark 和 Full GC 暂停期间根据这些选项调整 Java 堆的大小。此过程可能会向操作系统释放内存或从操作系统分配内存。

堆扩展可能发生在任何垃圾收集暂停期间。如果 G1 确定 Java 堆应该缩小，则该内存的释放将在暂停之后与暂停之后的申请同时进行。

仅限年轻阶段的生成大小

G1 在正常年轻代收集结束时，会为下一个修改阶段确定年轻代的初始大小。随着修改阶段的推进，G1 会定期优化该大小预估。

这 -XX:GCPauseIntervalMillis和 -XX:MaxGCPauseTimeMillis选项为 G1 提供了最低的内存使用率 (MMU)，以便进行垃圾收集活动。对于每个可能的时间范围 -XX:GCPauseIntervalMillis，G1 大小集合暂停最多使用 - XX:MaxGCPauseTimeMillis垃圾收集暂停时间（毫秒）。用于此计算的信息包括：先前对类似大小的年轻代清理所需时间的观察、收集期间需要复制的对象数量，以及这些对象的互连程度。

选项 -XX：G1新尺寸百分比和 -XX：G1MaxNewSizePercent限制最小和最大 eden 大小，进而限制垃圾收集暂停时间。 Garbage-First 垃圾收集器调优指南提供了一些关于如何使用这些来减少最大暂停的示例。

或者， -XX:新尺寸结合 -XX：最大新尺寸可用于分别设置最小和最大年轻代大小。

在正常年轻代收集开始时，G1 根据可用时间选择额外的老年代区域，如收藏集部分。

> **笔记：**
>
> 仅指定其中一个选项来设置 eden 大小，即可将年轻代大小固定为通过以下方式传递的值 -XX:新尺寸和 -XX：最大新尺寸分别。这将禁用暂停时间控制。

在年轻代收集开始时，G1 根据可用时间选择额外的老年代区域，如收藏集部分。

空间回收阶段生成大小

在空间回收阶段，G1 会尝试最大化单次垃圾收集暂停中老生代可回收的空间量。年轻代的大小与其他 Young-Only 阶段的垃圾收集相同，此外还会考虑纳入收集集合的老生代区域集的最小值。

在此阶段的每个混合收集开始时，G1 都会根据收藏集部分。收集集中老生代区域的数量确定如下：

- 确保清理工作顺利进行的最小老年代区域集。该老年代区域集的确定方法如下：通过标记确定为回收集候选的老年代区域数量，除以空间回收阶段的时长，空间回收阶段的时长由以下公式确定：XX:G1MixedGCCountTarget。

- 如果 G1 预测收集完最小集合后仍有剩余时间，则从收集候选集合中添加额外的老生代区域。老生代区域会一直添加，直到预测剩余时间的 80% 被使用为止。

- 在其他两个部分都已撤离并且在此暂停中还剩余时间后，G1 将逐步撤离一组可选的收集集区域。

G1 在初始垃圾收集过程中收集前两组区域，并在剩余的暂停时间内，使用可选收集组中的区域进行额外的收集。这种方法确保了空间回收的进度，同时提高了保持暂停时间的概率，并最大限度地降低了由于管理可选收集组而产生的开销。

当收集集合候选区域集中不再有标记候选区域时，空间回收阶段结束。

看Garbage-First 垃圾收集器调优有关 G1 将使用多少个老一代区域以及如何避免长时间的混合收集暂停的更多信息。

定期垃圾收集

如果由于应用程序长时间不活动而导致垃圾回收未进行，虚拟机可能会不必要地长时间占用大量未使用的内存，而这些内存本可用于其他用途。为了避免这种情况，可以使用以下方法强制 G1 定期进行垃圾回收： XX:G1周期GC间隔选项在长时间空闲期间有效。此选项确定 G1 在检测到应用程序空闲状态后考虑执行垃圾收集的最小间隔（以毫秒为单位）。如果自上次垃圾收集暂停以来已经过了此时间间隔，并且没有正在进行的并发循环，G1 会触发额外的垃圾收集，并可能产生以下影响：

- 在 Young-Only 阶段：G1 使用并发启动暂停来启动并发标记，或者，如果 -XX:-G1PeriodicGCInvokesConcurrent已指定，一次完整的 GC。
- 在空间回收阶段：G1 继续空间回收阶段，触发适合当前进度的垃圾收集暂停类型。

这 -XX：G1P eriodicGCSystemLoadThreshold选项应该用来细化 G1 的空闲含义：如果返回的平均一分钟系统负载值获取平均负载（） JVM 主机系统（例如，容器）上的调用次数高于此值，则 VM 不会被视为空闲，并且不会运行定期垃圾收集。

看JEP 346：立即从 G1 归还未使用的已提交内存有关定期垃圾收集的更多信息。

确定启动堆占用率

这*启动堆占用百分比 (IHOP)*是触发并发开始收集的阈值，它定义为老一代大小的百分比。

G1 默认会通过观察标记周期中老年代的内存分配情况以及标记所需的时间，自动确定最佳的 IHOP。此功能称为*自适应IHOP*。如果此功能处于活动状态，则选项 - XX:启动堆占用百分比只要没有足够的观测数据来准确预测初始堆占用率阈值，就会将初始值确定为当前老年代大小的百分比。使用

选项-XX:-G1使用自适应IHOP。在这种情况下，值为 - XX:启动堆占用百分比始终决定这个阈值。

在内部，自适应 IHOP 尝试设置启动堆占用率，以便当旧代占用率等于当前最大旧代大小减去以下值时，空间回收阶段的第一次混合垃圾收集开始 -XX：G1堆预留百分比作为额外的缓冲区。

标记

G1 标记使用称为*开头快照（SATB）*它会在并发启动暂停时对堆进行虚拟快照。所有在标记开始时处于活动状态的对象在剩余的标记时间内都被视为活动状态。这意味着在标记期间变为“死亡”（无法访问）的对象在空间回收时仍被视为活动状态（有一些例外）。与其他收集器相比，这可能会导致错误地保留一些额外的内存。但是，SATB 可能会在重新标记暂停期间提供更好的延迟。在该标记期间被认为过于保守的存活对象将在下一次标记期间被回收。请参阅Garbage-First 垃圾收集器调优主题以获取有关标记问题的更多信息。

疏散失败

疏散失败意味着 G1 在垃圾收集期间无法移动某些对象。

垃圾收集日志中会显示此类事件 -Xlog：gc使用 疏散失败：*<原因>*打印输出*<原因>*是以下其中之一或两者分配和 已固定如下例所示：

[9,740s][info][gc] GC(26) 暂停年轻（正常）（G1 疏散暂停）（疏散失败：分配/固定）2159M->402M(3000M) 6,108ms

- 分配：G1 无法在目标区域找到足够的空间来移动对象。
- 已置顶：存在 G1 无法移动的对象，因为 G1 发现了一个已锁定在原地的对象，或者*固定*，以便使用 获取原始数组关键值()或类似的 JNI 调用。请参阅JEP 423：G1 的区域固定有关固定对象的更多信息。

如果 G1 无法将所有对象移出某个区域，该区域将暂时无法分配。G1 会将这些区域安排为收集集候选，并在下次垃圾收集时立即撤出。

在最坏的情况下，如果垃圾回收期间没有释放任何空间，G1 将安排一次完整 GC。这种类型的垃圾回收会对整个堆执行就地压缩。这可能会非常慢。

看Garbage-First 垃圾收集器调优在发出内存不足信号之前，了解有关分配失败或完整 GC 问题的更多信息。

巨型物体

巨型对象是指大于或等于半个区域大小的对象。当前区域大小是根据人体工程学原理确定的，具体定义在G1 GC 的人体工程学默认值部分，除非使用 -XX:G1堆区域大小选项。

这些巨大的物体有时会被以特殊的方式处理：

- 每个巨型对象都会在老生代中分配为一系列连续的区域。对象本身的起始位置始终位于该序列中第一个区域的起始位置。序列最后一个区域中剩余的空间将无法分配，直到整个对象被回收为止。

- 通常，巨型对象只能在标记结束时（Remark 暂停期间）或在 Full GC 期间（如果它们变得不可达）被回收。但是，对于原始类型的数组，有一个特殊的规定，例如：布尔值，所有类型的整数和浮点值。如果巨型对象在垃圾收集暂停期间未被太多对象引用，G1 会尝试回收它们。此行为默认启用，但您可以使用以下选项禁用它：XX：-G1EagerReclaimHumongousObjects。

- 巨型对象的分配可能会导致垃圾收集过早暂停。G1 会在每次巨型对象分配时检查初始堆占用率阈值，如果当前占用率超过该阈值且当前没有正在进行的标记操作，则可能会立即强制执行并发启动年轻代垃圾收集 (Concurrent Start young collection)。

- 巨型对象仅在第一次完整 GC 未能释放足够的连续内存以在同一暂停期间的第二次完整 GC 中分配另一个巨型对象后，才会在最后的回收措施中移动。此过程非常缓慢。由于包含巨型对象末尾的堆区域中没有可用于分配的空间，G1 仍然有可能因内存不足而退出虚拟机。

G1 GC 的人体工程学默认值

本主题概述了 G1 最重要的默认设置及其默认值。它们粗略地概述了在不使用任何附加选项的情况下使用 G1 的预期行为和资源使用情况。

表 7-1 G1 GC 人体工程学默认值

选项和默认值	描述
-XX:MaxGCPauseMillis=200	最大暂停时间的目标。
-XX:GCPauseTimeInterval=***<因此>***	最大暂停时间间隔的目标。默认情况下，G1 不设置任何目标，允许 G1 在极端情况下连续执行垃圾收集。
-XX:ParallelGCThreads=***<因此>***	垃圾回收暂停期间用于并行工作的最大线程数。此值由虚拟机所在计算机的可用线程数计算得出，计算方式如下：如果进程可用的 CPU 线程数小于或等于 8，则使用该值。否则，将大于该值的八分之五添加到最终线程数中。 每次暂停开始时，使用的最大线程数受到最大总堆大小的进一步限制：G1 每次不会使用超过一个线程 - XX:每个GC线程的堆大小Java 堆的数量 容量。
-XX:ConcGCThreads=***<因此>***	用于并发工作的最大线程数。默认情况下，此值为 - XX:并行GC线程除以 4。

表 7-1（续）人体工程学默认值 G1 GC

选项和默认值	描述
-XX:+G1使用自适应IHOP - XX：InitiatingHeapOccupancyPercent=45	控制启动堆占用率的默认值表明该值的自适应确定已打开，并且对于前几个收集周期，G1 将使用旧代 45% 的占用率作为标记启动阈值。
-XX:G1HeapRegionSize=*<因此>*	堆区域的大小。默认值基于最大堆大小，计算结果约为 2048 个区域，符合人体工程学的最大值为 32 MB。用户指定的大小必须是 2 的幂，有效值范围为 1 到 512 MB。
-XX：G1NewSizePercent=5 -XX：G1MaxNewSizePercent=60	年轻代的总大小，以当前正在使用的 Java 堆的百分比在这两个值之间变化。
-XX:G1HeapWastePercent=5	堆中允许的未回收空间占当前堆总大小的百分比。如果回收老年代区域后的总可用空间低于该值，G1 将停止将老年代区域添加到标记收集集合候选对象中。
-XX：G1MixedGCCountTarget=8	多次收集中空间回收阶段的预期长度。
-XX:G1MixedGCLiveThresholdPercent=85	活动对象占用率高于此百分比的老一代区域将不会在空间回收阶段被收集。

> **笔记：**
>
> <因此>意味着实际值是根据环境以人体工程学的方式确定的。

回收老年代中空的大对象始终处于启用状态。您可以使用以下选项禁用此功能：XX:-G1EagerReclaimHumongousObjects。字符串去重默认关闭。可以使用以下选项启用 -XX:+G1EnableStringDeduplication。

与其他收藏家的比较

以下是 G1 与其他收集器的主要区别的总结：

- 并行 GC 只能整体压缩并回收老年代的空间。G1 会将这项工作逐步分散到多个更短的收集中。这显著缩短了暂停时间，但可能会牺牲吞吐量。
- G1 同时执行部分老生代空间回收。
- G1 可能比上述收集器表现出更高的开销，由于其并发特性，会影响吞吐量。
- ZGC 的目标是以进一步牺牲吞吐量为代价，提供更短的暂停时间。

由于其工作方式，G1 有一些机制可以提高垃圾收集效率：

- G1 可以在任何垃圾回收过程中回收某些类型的巨型对象。这可以避免许多原本不必要的垃圾回收，从而轻松释放大量空间。

- G1 可以选择同时尝试对 Java 堆上的重复字符串进行重复数据删除。

回收老年代中空的大对象始终处于启用状态。您可以使用以下选项禁用此功能：XX:-G1EagerReclaimHumongousObjects。字符串去重默认关闭。您可以使用以下选项启用它：XX:+G1EnableStringDeduplication。

8

Garbage-First 垃圾收集器调优

本节介绍如何在 Garbage-First 垃圾收集器 (G1 GC) 行为无法满足您的要求的情况下对其进行调整。

主题

G1 的一般建议

一般建议使用 G1 的默认设置，最终为其指定不同的暂停时间目标，并使用以下方法设置最大 Java 堆大小 -Xmx如果需要的话。

G1 的默认配置与其他收集器有所不同。G1 的默认配置目标既不是最大化吞吐量，也不是最小化延迟，而是在高吞吐量下提供相对较短且均匀的停顿时间。然而，G1 的堆空间增量回收机制以及停顿时间控制会给应用程序线程和空间回收效率带来一些开销。

如果您更喜欢高吞吐量，那么可以通过使用以下方法放宽暂停时间目标：XX:MaxGCPauseMillis 或者提供更大的堆。如果延迟是主要要求，则修改暂停时间目标。避免将年轻代大小限制为特定值，可以使用以下选项：Xmn,-XX:新比率以及其他原因，因为年轻代大小是 G1 满足暂停时间要求的主要手段。将年轻代大小设置为单个值会覆盖并实际上禁用暂停时间控制。

从其他收集器移至 G1

一般来说，当从其他收集器移到 G1 时，首先删除所有影响垃圾收集的选项，然后仅使用以下方法设置暂停时间目标和整体堆大小 -Xmx并且可选地-Xms。

许多有助于其他收集器以特定方式响应的选项，要么完全没有效果，甚至会降低吞吐量，并降低达到暂停时间目标的可能性。例如，设置年轻代大小可能会完全阻止 G1 调整年轻代大小以满足暂停时间目标。

提高 G1 性能

G1 旨在提供良好的整体性能，无需指定其他选项。然而，在某些情况下，默认的启发式算法或默认配置并不能提供理想的结果。本节提供了一些关于诊断和改进这些情况的指南。本指南仅介绍 G1 在特定应用场景下，针对特定指标提升垃圾收集器性能的可能性。根据具体情况，应用程序级别的优化可能比尝试调整虚拟机以提高性能更有效，例如，通过减少对象寿命来避免某些问题情况。

为了便于诊断，G1 提供了全面的日志记录。一个好的开始是使用 - Xlog：gc*=调试选项，然后根据需要优化输出。日志提供了暂停期间和暂停之外的垃圾收集活动的详细概述。这包括收集类型以及暂停特定阶段所花费时间的细分。

以下小节探讨一些常见的性能问题。

观察完整的垃圾收集

完整的堆垃圾收集（Full GC）通常非常耗时。由于老年代堆占用率过高而导致的完整 GC 可以通过查找以下语句来检测：*完全暂停（G1 压缩暂停）*在日志中。完整的 GC 通常先于垃圾收集，并遇到疏散失败，分配原因。

发生完整 GC 的原因是应用程序分配了过多的对象，而这些对象无法快速回收。通常情况下，并发标记无法及时完成，导致空间回收阶段无法启动。大量超大对象的分配会加剧发生完整 GC 的可能性。由于这些对象在 G1 中的分配方式，它们可能占用比预期更多的内存。

目标应该是确保并发标记按时完成。这可以通过降低老年代的分配率，或者给予并发标记更多时间来实现。

G1 为您提供了几种选项来更好地处理这种情况：

- 您可以使用以下方法确定 Java 堆上巨型对象所占用的区域数量gc+heap=信息伐木。是在行中”巨大区域：X->Y”给出巨型对象占用的区域数量。如果此数量与旧区域数量相比较高，则最佳选择是尝试减少此对象数量。您可以通过使用以下方法增加区域大小来实现此目的：XX:G1堆区域大小选项。当前选定的堆区域大小打印在日志的开头。

- 增加 Java 堆的大小。这通常会增加完成标记所需的时间。
- 通过设置增加并发标记线程的数量 -XX:ConcGCThreads 明确地。
- 强制 G1 提前开始标记。G1 会根据之前的应用程序行为自动确定初始堆占用率 (IHOP) 阈值。如果应用程序行为发生变化，这些预测可能会出错。有两种选择：通过修改以下参数来降低启动空间回收的目标占用率：增加自适应 IHOP 计算中使用的缓冲区；XX:G1储备百分比；或者，通过以下方式手动设置禁用 IHOP 的自适应计算 -XX:-G1使用自适应IHOP和

  -XX：InitiatingHeapOccupancyPercent。

除了分配失败之外，Full GC 的其他原因通常表明应用程序或某些外部工具导致了完整的堆回收。如果原因是系统.gc()，并且没有办法修改应用程序源，可以通过使用以下方法减轻完整 GC 的影响 -XX:+显式GC调用并发或者通过设置让虚拟机完全忽略它们 - XX:+DisableExplicitGC。外部工具可能仍会强制执行完整的 GC；只有不请求它们才能将其删除。

巨型对象碎片

由于需要为所有 Java 堆内存找到一组连续的区域，因此在耗尽所有 Java 堆内存之前，可能会发生完整 GC。在这种情况下，可以使用以下选项来增加堆区域的大小：XX:G1堆区域大小减少巨型对象的数量，或者增加堆的大小。在极端情况下，即使可用内存指示情况并非如此，G1 也可能没有足够的连续空间来分配对象。如果 Full GC 无法回收足够的连续空间，这将导致虚拟机退出。因此，除了像前面提到的那样减少巨型对象分配的数量，或者增加堆大小之外，别无选择。

调整延迟

本节讨论在出现常见延迟问题（即暂停时间过长）时改善 G1 行为的提示。

异常系统或实时使用情况

对于每次垃圾收集暂停，gc+cpu=信息日志输出包含一行来自操作系统的信息，其中详细列出了暂停期间的具体操作。例如，用户=0.19s 系统=0.00s 实际=0.01s。

用户时间是花费在虚拟机代码上的时间，*系统时间*是在操作系统中花费的时间，并且 *即时的*是暂停期间经过的绝对时间量。如果系统时间相对较高，则通常是环境原因造成的。

系统时间过长的常见已知问题有：

- 虚拟机从操作系统内存中分配或归还内存可能会导致不必要的延迟。请使用以下选项将最小和最大堆大小设置为相同的值，以避免延迟 -Xms和 -Xmx，并使用以下方式预触所有内存： XX:+AlwaysPreTouch将此工作移至 VM 启动阶段。
- 特别是在 Linux 中，将小页面合并成大页面*透明大页面（THP）*该功能往往会拖延随机进程，而不仅仅是在暂停期间。由于虚拟机分配并维护大量内存，因此存在比平常更高的风险，

VM 进程将会长时间停滞。请参阅操作系统文档，了解如何禁用透明大页面功能。

- 由于某些后台任务间歇性地占用了日志写入硬盘的所有 I/O 带宽，日志输出的写入可能会暂停一段时间。请考虑为日志使用单独的磁盘或其他存储方式（例如，基于内存的文件系统）来避免这种情况。另一个缓解措施是使用异步日志记录，即虚拟机使用以下方式异步写入日志：Xlog：异步命令行选项。

需要注意的另一种情况是实际时间比其他时间的总和大得多，这可能表明虚拟机在可能超载的机器上没有获得足够的 CPU 时间。

引用对象处理耗时过长

有关处理参考对象所用时间的信息显示在 参考处理阶段。在此期间参考处理阶段，G1 根据特定类型的引用对象的需求更新引用对象的引用项。默认情况下，G1 会尝试并行化参考处理 使用以下启发式方法：对于每一个 -XX：每个线程的引用数引用对象启动一个线程，受以下值的限制 -XX：并行GC线程。可以通过设置禁用此启发式方法 -XX：每个线程的引用数设置为 0 表示默认使用所有可用线程，或者通过以下方式完全禁用并行化 -XX：-ParallelRefProcEnabled。

Young-Only 阶段的 Young-Only 回收耗时过长

正常的年轻代收集，以及一般任何年轻代收集所花费的时间大致与年轻代的大小成正比，或者更具体地说，与收集集合中需要复制的存活对象的数量成正比。如果*撤离收集装置*阶段耗时太长，特别是 *对象复制子*阶段，减少 -XX:G1NewSizePercent。这会减少年轻一代的最小规模，从而允许更短的暂停时间。

如果应用程序性能（尤其是垃圾收集中存活的对象数量）突然发生变化，年轻代大小调整还可能出现另一个问题。这可能会导致垃圾收集暂停时间激增。以下方法可能会有助于降低年轻代的最大大小：XX:G1MaxNewSizePercent。这限制了年轻一代的最大大小，因此也限制了暂停期间需要处理的对象的数量。

混合收集耗时太长

混合年轻代回收用于回收老年代空间。混合回收的集合包含年轻代和老年代区域。您可以通过启用以下选项来获取年轻代或老年代区域的清理时间对暂停时间的影响：gc+ergo+cset=调试日志输出。查找以下日志消息：

向 CSet 添加了年轻区域。[...] 预测伊甸园时间：4.86 毫秒，预测基数时间：9.98 毫秒，目标暂停时间：200.00 毫秒，[...]

Eden 时间和 Base 时间一起给出了预测的年轻区域时间，即 G1 预计疏散年轻代所需的时间

预测旧区域时间的日志消息如下：

完成选择收集集旧区域。[...] 预测初始时间：147.70 毫秒，预测可选时间：15.45 毫秒，[...]

这里，预测的初始时间表示预测的老年代区域时间，即 G1 预计疏散老年代区域的最小集合所需的时间。

如果预测的年轻区域时间太长，则参见Young-Only 阶段的 Young-Only 回收耗时过长选项。另外，为了减少老生代区域对暂停时间的贡献，G1 提供了三个选项：

- 通过增加以下方式将老一代区域回收扩展到更多的垃圾收集中：XX:G1MixedGCCountTarget。
- 避免收集需要花费大量时间的区域，方法是不将它们放入候选收集集中，方法是使用-

  XX:G1MixedGCLiveThresholdPercent。在许多情况下，占用率较高的区域需要花费大量时间来收集。
- 尽早停止老年代空间回收，以免 G1 回收过多占用率过高的区域。在这种情况下，请增加 -XX:G1HeapWastePercent。

请注意，后两个选项会减少当前 Space-Reclamation 阶段可回收空间的回收集候选区域数量。这可能意味着 G1 可能无法在老年代回收足够的空间来维持持续运行。然而，后续的 Space-Reclamation 阶段或许能够回收这些区域。

收集连续发生

G1 默认的 MMU 设置允许连续垃圾回收。默认值为 - XX:GCPauseIntervalMillis略高于 -XX:MaxGCPauseMillis。如果您观察到连续的背靠背垃圾收集，导致应用程序无法继续运行，请增加以下值 -XX:GCPauseIntervalMillis达到可接受的值。然后，G1 将尝试进一步延长垃圾收集间隔。

高合并堆根和扫描堆根时间

减少这些阶段的一种方法是减少组合记忆集中记忆集条目的数量。使用以下选项调整堆区域的大小 -XX:G1堆区域大小减少记忆集的跨区域引用数量。较大的区域往往具有较少的跨区域引用，因此处理这些引用所花费的相对工作量会减少。但与此同时，较大的区域可能意味着每个区域需要清除的存活对象更多，从而增加其他阶段所需的时间。

如果垃圾收集的大量时间（即超过 60%）花在了这两个阶段，那么一个选项是降低记忆集条目的粒度，方法是降低 -XX：GCCardSizeInBytes选项：更细的粒度减少了查找引用的工作量，但需要一些额外的内存。

扫描堆根 (Scan Heap Roots) 时间过长，再加上应用程序分配大型对象，可能是由于某种优化尝试通过批处理来减少并发记忆集更新工作量。如果创建此类批处理的应用程序恰好发生在垃圾回收之前，这可能会对合并堆根 (Merge Heap Roots) 时间产生负面影响。请使用

-XX:-减少初始卡片标记禁用此优化并可能避免这种情况。

吞吐量调优

G1 的默认策略试图在吞吐量和延迟之间保持平衡；然而，在某些情况下，更高的吞吐量是可取的。除了像上一节中提到的那样减少总体暂停时间外，还可以降低暂停的频率。主要思路是通过以下方式增加最大暂停时间：XX:MaxGCPauseMillis。代大小启发式算法会自动调整年轻代的大小，这直接决定了暂停的频率。如果这没有导致预期的行为，

特别是在空间回收阶段，使用以下方法增加最小年轻代大小：XX：G1新尺寸百分比将强制 G1 这样做。

在某些情况下，-XX:G1MaxNewSizePercent，允许的最大年轻代大小，可能会通过限制年轻代大小来限制吞吐量。这可以通过查看区域摘要输出来诊断gc+heap=信息日志记录。在这种情况下，Eden 区域和 Survivor 区域的总百分比接近 -XX：G1MaxNewSizePercent占区域总数的百分比。考虑增加-XX：G1MaxNewSizePercent在这种情况下。

提高吞吐量的另一个方法是减少并发工作量。特别是，并发的记忆集更新通常需要大量的 CPU 资源。选项如下：XX:G1RSetUpdatingPauseTimePercent可用于将工作从并发操作移至垃圾收集暂停中。

增加此值可能会减少与应用程序同时安排的细化工作，相反，减少此值可能会增加与应用程序同时执行的细化工作量。

垃圾收集暂停中的改进工作在日志缓冲区的一部分 合并堆根启用时阶段gc+阶段=调试伐木。

通过使用以下方式启用大页面：XX:+使用大页面也可能提高吞吐量。请参阅操作系统文档，了解如何设置大页面。

您可以通过禁用它来最小化堆调整大小的工作；设置选项 -Xms和 -Xmx设置为相同的值。此外，您还可以使用 -XX:+AlwaysPreTouch将操作系统工作移回虚拟内存，并将物理内存移回虚拟机启动时间。为了使暂停时间更加一致，这两种措施都特别有用。

调整堆大小

与其他收集器一样，G1 的目标是调整堆的大小，以便垃圾收集所花费的时间低于以下确定的比率 -XX:GC时间比率选项。调整此选项以使 G1 满足您的要求。

可调默认值

本节介绍本主题中介绍的命令行选项的默认值和一些附加信息。

表 8-1 G1 GC 可调默认值

选项和默认值	描述
- XX:+减少初始卡片标记	这将初始对象分配的并发记忆集更新（细化）工作分批处理在一起。
-XX:+ParallelRefProcEnabled -XX：ReferencesPerThread=1000	-XX：每个线程的引用数决定了 并行化：对于每一个*北*引用对象一个线程将参与引用处理的子阶段，受以下限制：XX：并行GC线程。值为 0 表示最大线程数，如值所示 - XX:并行GC线程将会一直被使用。 这决定了是否处理 java.lang.Ref.*实例应该由多个线程并行完成。

表 8-1（续）可调默认值 G1 GC

选项和默认值	描述
-XX：G1RSetUpdatingPauseTimePercent=10	可以使用此选项控制并发的记忆集更新（优化）工作。优化会尝试并发地安排工作，以便最多 -XX:G1RSetUpdatingPauseTimePercent百分比最大暂停时间目标是在更新 RS 阶段的垃圾收集暂停中花费，处理剩余的工作。
-XX：G1SummarizeRSetStatsPeriod=0	这是指在多次 GC 中，G1 生成记忆集摘要报告的时间间隔。设置为零即可禁用。
-XX:GCTimeRatio=12	这是垃圾收集相对于应用程序应花费的时间目标比率的除数。在增加堆之前，确定垃圾收集可花费时间目标比例的实际公式是1 / (1 + GCTimeRatio)。此默认值导致目标大约有 8% 的时间花在垃圾收集上。
-XX：G1P eriodicGCInterval=0	检查 G1 是否应触发定期垃圾回收的时间间隔（以毫秒为单位）。设置为零则禁用。
-XX:+G1PeriodicGCInvokesConcurrent	如果设置，定期垃圾收集将触发并发标记或继续现有的收集周期，否则触发完整 GC。
-XX：G1P eriodicGCSystemLoadThreshold=0.0	主机返回的当前系统负载阈值获取平均负载（）调用以确定是否应触发定期垃圾回收。当前系统负载高于此值时，将阻止定期垃圾回收。值为零表示此阈值检查已禁用。

笔记：

<因此>意味着实际值是根据环境以人体工程学的方式确定的。

9

Z 垃圾收集器

这*Z 垃圾收集器（ZGC）*ZGC 是一款可扩展的低延迟垃圾收集器。ZGC 并发执行所有高开销工作，且不会使应用程序线程的执行暂停超过一毫秒。它适用于需要低延迟的应用程序。暂停时间与正在使用的堆大小无关。ZGC 适用于从几百兆字节到 16TB 的堆大小。

ZGC 的设计旨在实现自适应性，并最大限度地减少手动配置。在 Java 程序执行过程中，ZGC 会通过调整代大小、扩展 GC 线程数量以及调整老年代阈值来动态适应工作负载。主要的调整选项是增加最大堆大小。

> **笔记：**
>
> 从 JDK 24 开始，ZGC 是一个分代垃圾收集器。Z世代选项已被删除。

主题

设置堆大小

ZGC 最重要的调整选项是设置最大堆大小，您可以使用以下命令设置 -Xmx命令行选项。由于 ZGC 是一个并发收集器，您必须选择一个最大堆大小，以便堆能够容纳应用程序的活跃集，并且堆中有足够的空间，以便在 GC 运行时进行分配。所需的空间大小很大程度上取决于应用程序的分配率和活跃集大小。通常，分配给 ZGC 的内存越多越好。但同时，浪费内存也是不可取的，因此关键在于在内存使用量和 GC 运行频率之间找到平衡。

ZGC 还有另一个与堆大小相关的命令行选项，名为 -XX：SoftMaxHeapSize。 它可以用来设置 Java 堆大小的软限制。ZGC 会尽量不超出此限制，但仍允许其增长到最大堆大小。ZGC 仅在必要时才会使用超过软限制的内存，以防止 Java 应用程序停滞并等待 GC 回收内存。例如，使用以下命令行选项：Xmx5g -XX:SoftMaxHeapSize=4gZGC 将使用 4GB 作为其启发式方法的限制，但如果它不能将堆大小保持在 4GB 以下，则仍然允许暂时使用最多 5GB。

将未使用的内存返回给操作系统

默认情况下，ZGC 会取消提交未使用的内存，并将其归还给操作系统。这对于需要关注内存占用的应用程序和环境非常有用，但可能会对 Java 线程的延迟产生负面影响。您可以使用以下命令行选项禁用此功能：XX:-Z取消提交。此外，内存不会被取消提交，因此堆大小会缩小到最小堆大小以下（-Xms)。这意味着如果最小堆大小（-Xms)配置为等于最大堆大小（-Xmx)。

您可以使用以下方式配置取消提交延迟 -XX:ZUncommitDelay=<秒数> (默认值为 300 秒）。此延迟指定内存在符合取消提交条件之前应处于闲置状态的时间。

> **笔记：**
>
> 允许 GC 在应用程序运行时提交和取消提交内存可能会对 Java 线程的延迟产生负面影响。如果使用 ZGC 运行的主要原因是极低的延迟，请考虑使用相同的值运行 - Xmx和 -Xms，并使用 -XX:+AlwaysPreTouch在应用程序启动之前将其分页到内存中。

使用大页面

将 ZGC 配置为使用大页面通常会带来更好的性能（在吞吐量、延迟和启动时间方面），并且没有实际的缺点，只是设置起来稍微复杂一些。设置过程通常需要 root 权限，因此默认情况下不启用此功能。

在 Linux 上启用大页面

在 Linux x86 上，大页面（也称为“巨页”）的大小为 2MB。

假设您需要一个 16GB 的 Java 堆。这意味着您需要 16GB / 2MB = 8192 个大页面。

堆需要至少 16GB（8192 个页面）的内存用于大页面池。堆以及 JVM 的其他部分将使用大页面来存储各种内部数据结构（例如代码堆和标记位图）。在本例中，您将预留 9216 个页面（18GB），以便 2GB 的非 Java 堆分配能够使用大页面。

配置系统的大页池，使其具有所需的页面数量（需要root权限）：

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

请注意，如果内核找不到足够的可用大页面来满足请求，则上述命令不保证成功。另请注意，内核可能需要一些时间来处理请求。在继续操作之前，请检查分配给池的大页面数量，以确保请求已成功完成。

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

9216

在 Linux 上启用透明大页面

除了使用显式大页面（如前所述）之外，还可以使用透明大页面。对于延迟敏感的应用程序，通常不建议使用透明大页面，因为它容易导致不必要的延迟峰值。但是，您可以尝试一下，看看您的工作负载是否或会受到它的影响。

> **笔记：**
>
> 在 Linux 上，使用启用透明大页面的 ZGC 需要内核 >= 4.7。

使用以下选项在虚拟机中启用透明大页面：

-XX:+使用大页面 -XX:+使用透明大页面

这些选项告诉 JVM 发出madvise（...，MADV_HUGEPAGE）调用它映射的内存，这在使用透明大页面时很有用疯狂建议模式。

要启用透明大页面，您还需要配置内核，启用疯狂建议 模式。

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

ZGC 使用共享内存大页面用于堆，因此还需要配置以下内核设置：

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

在比较不同 GC 的性能时，检查这些内核设置非常重要。一些 Linux 发行版会通过配置 / 强制启用透明大页面来处理私有页面。sys/kernel/mm/transparent_hugepage/enabled设置为总是，离开时 /sys/kernel/mm/transparent_hugepage/shmem_enabled默认绝不。在这种情况下，除 ZGC 之外的所有 GC 都会使用透明大页面来处理堆。参见透明大页面支持了解更多信息。

10

其他考虑因素

本节介绍影响垃圾收集的其他情况。

主题

终结和弱引用、软引用和虚引用

一些应用程序通过使用终止和弱引用、软引用或幻像引用与垃圾收集进行交互。

然而，我们不建议使用 finalization。它可能会导致安全性、性能和可靠性方面的问题。例如，依赖 finalization 来关闭文件描述符会使外部资源（描述符）依赖于垃圾回收的及时性。

> **笔记：**
>
> 在 JDK 9 中，Finalization 已被弃用。在 JDK 18 中，它也已被弃用并被删除；请参阅 JEP 421：弃用“删除”的最终确定性。

最终确定

类可以声明终结器 – 方法受保护的 void finalize() –其主体会释放所有底层资源。GC 会调度不可达对象的终结器，该终结器会在 GC 回收对象内存之前被调用。

当从 GC 根到对象的路径不存在时，对象将变得无法访问，因此可以被垃圾回收。GC 根包含来自活动线程的引用和 JVM 内部的引用；这些引用将对象保留在内存中。

请参阅监控待完成的对象*Java 平台标准版故障排除指南*确定系统中是否正在构建可终结对象。此外，您还可以使用以下工具之一：

- JDK 任务控制：
  1. 在**JVM浏览器**，右键单击您的 JVM 并选择**启动 JMX 控制台**。
  2. 在**MBean浏览器**，在**MBean 树**， 扩张**java.lang**并选择**记忆**。
  3. 在**MBean 功能**，属性**对象待处理完成计数**是等待最终确定的对象的大约数量。
- jcmd工具：

  - 运行以下命令打印有关 Java 终止队列的信息；值*<进程ID>*是你的 JVM 的 PID：

    jcmd *<进程ID>*GC.finalizer_info

- JDK 飞行记录器：
  - JDK 飞行记录器 (JFR) 事件，jdk.Finalizer统计信息，识别运行时使用终结器的类。该事件在默认.jfc和 配置文件.jfcJFR 配置文件。启用后，JFR 会发出 jdk.Finalizer统计信息每个实例化类的事件，具有非空 finalize()方法。事件包含覆盖finalize()，那个班的代码源，类的终结器已运行的次数，以及仍在堆上（且尚未完成）的对象数量。请参阅 Flight Recorder *Java 平台，标准版 JDK 任务控制用户指南*了解更多信息。

从 Finalization 迁移

为了避免最终确定，请使用以下技术之一：

- try-with-Resources 语句
- 更清洁的 API

try-with-Resources 语句

这尝试-with-resources 语句是尝试声明一个或多个资源的语句。资源是一个对象，程序使用完毕后必须关闭。尝试-withresources 语句确保每个资源在代码块结束时关闭，即使发生一个或多个异常。参见Try-with-resources 语句了解更多信息。

更清洁的 API

如果你预见到应用程序中资源的生命周期将超出 尝试-如果您使用 with-resources 语句，则可以改用 Cleaner API。Cleaner API 允许程序为对象注册清理操作，该操作会在对象无法访问后运行一段时间。

清洁器可以让你避免终结器的许多缺点：

- 更安全：清理者必须明确注册一个对象。此外，清理操作无法访问该对象，因此无法恢复对象。
- 性能更佳：您可以更好地控制何时注册清理操作，这意味着清理操作永远不会处理未初始化或部分初始化的对象。您还可以取消对象的清理操作。
- 更可靠：您可以控制哪些线程运行清理操作。

然而，与终结器类似，垃圾收集器会安排清理操作，因此它们可能会受到无限延迟的影响。因此，在需要及时释放资源的情况下，请勿使用清理器 API。

以下是一个简单的清理器示例。它执行以下操作：

1.定义清洁动作类，状态，初始化清洁操作并定义清洁操作本身（通过覆盖状态::运行()方法）。

2.创建一个实例清洁工。

3.有了这个例子清洁工，注册对象myObject1以及清洁动作（状态）。

4.确保垃圾收集器安排清洁人员和清洁行动 状态::运行()在示例结束之前执行，示例：

一个。套myObject1到无效的以确保它是幻像不可达的。请参阅。

b.呼叫系统.gc()循环触发垃圾收集清理。

图 10-1 CleanerExample

```
导入 java.lang.ref.Cleaner;

公共类 CleanerExample {

    // 此 Cleaner 由所有 CleanerExample 实例共享 private static final
    Cleaner CLEANER = Cleaner.create(); private final State state;


    公共 CleanerExample（字符串 id）{
        状态 = 新状态 (id);
        CLEANER.register(this, state);
    }

    // CleanerExample 的清洁操作类 private static class State
    implements Runnable {
        最终的私有字符串id;

        私有状态（字符串 id）{
            这个.id = id;
            System.out.println("为 " + this.id 创建了清理操作);
        }

        @Override
        公共无效运行（）{
            System.out.println("清洁垃圾收集" + this.id);
        }
    }

    公共静态void main（String [] args）{
        CleanerExample myObject1 = new CleanerExample("myObject1");

        // 使 myObject1 不可达 myObject1
        = null;

        System.out.println("-- 给 GC 一个机会来安排清理器
- - ");
        对于（int i = 0; i < 100; i++）{

            // 在循环中调用 System.gc() 通常足以触发 // 像这样的小程序中的清理。

            系统.gc();
            尝试 {
                线程.睡眠(1);
            } 捕获（中断异常 e）{}
        }
```

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

此示例打印以下内容：

```
为 myObject1 创建清理操作
- - 给 GC 一个机会来安排清理程序 - 清理程序收集了
myObject1
- - 完成的  -
```

如果您要为生产环境实施清洁器，请考虑以下事项：

- 清洁动作类别（状态在这个例子中）应该是私有的实现细节。特别是，它不应该被从主要（字符串[]）方法。因此，您的清洁操作类在任何可行的情况下都应该是不可变的。新对象应该负责创建自己的清洁操作类，并在其构造函数中向清洁器注册自身。

- 类通常需要访问清洁操作类中的对象。最简单的方法是让对象保存对清洁操作类的引用。

- 清洁工实例应该共享。在本例中，清洁器示例 应该共享一个静态的清洁工实例。

请参阅 JavaDoc API 文档清洁工类以获取有关实施清洁器的更多信息。

引用对象类型

有三种引用对象类型：软引用、弱引用、和 PhantomReference。每种引用对象类型都对应不同的可达性级别。以下是可达性级别（从强到弱）的排列顺序，它们反映了对象的生命周期：

- 一个对象是*强可达*是否某个线程无需遍历任何引用对象即可访问该对象。新创建的对象对于创建它的线程来说是强可达的。

- 一个对象是*软可达*如果它不是强可达的，但可以通过遍历软引用来到达。

- 一个对象是*弱可达*如果它既不是强可达也不是软可达，但可以通过遍历弱引用到达。当指向弱可达对象的弱引用被清除后，该对象就具备被终结的条件。

- 一个对象是*幻影可达*如果它既不是强可达、软可达也不是弱可达，那么它就已经完成了，并且有一些幻影引用指向它。
- 一个对象是*无法访问*，因此当无法通过任何先前的方式到达时，该地块有资格进行回收。

每个引用对象类型都封装了对特定对象的单个引用，称为*指称*。引用对象提供了清除引用的方法。

以下是引用对象实例最常见的用途：

- 保持对对象的访问，同时在系统需要释放内存时仍允许对其进行垃圾回收（例如，可以根据需要重新生成的缓存值）
- 当对象达到特定的可达性级别时，确定并可能采取一些行动（结合参考队列班级）

显式垃圾回收

应用程序与垃圾收集交互的另一种方式是通过使用以下方式显式调用完整垃圾收集系统.gc()。

这可能会在没有必要的情况下强制执行主要回收（例如，当次要回收就足够时），因此通常应避免这种情况。显式垃圾回收的性能影响可以通过使用以下标志禁用它们来衡量 - XX:+DisableExplicitGC，这会导致虚拟机忽略对系统.gc()。

显式垃圾回收最常见的用途之一是远程方法调用 (RMI) 的分布式垃圾回收 (DGC)。使用 RMI 的应用程序会引用其他虚拟机中的对象。如果不偶尔调用本地堆的垃圾回收，这些分布式应用程序中的垃圾就无法被回收，因此 RMI 会定期强制执行完全回收。这些回收的频率可以通过属性来控制，如下例所示：

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

此示例指定每小时一次的显式垃圾回收，而不是默认的每分钟一次。然而，这也可能导致某些对象需要更长时间才能被回收。这些属性可以设置为长整型.MAX_VALUE如果不希望 DGC 活动的及时性设置上限，那么可以使显式收集之间的时间实际上无限大。

软引用

软引用在服务器虚拟机中保持活跃的时间比在客户端中更长。

清除率可以通过命令行选项控制 - XX:SoftRefLRUPolicyMSPerMB=<N>，它指定软引用在堆中每兆字节可用空间中（一旦不再强可达）保持活动状态的毫秒数 (ms)。默认值为每兆字节 1000 毫秒，这意味着软引用在堆中每兆字节可用空间中（在对象的最后一个强引用被回收之后）将存活 1 秒。这是一个近似值，因为软引用仅在垃圾回收期间清除，而垃圾回收可能偶尔发生。

类元数据

Java 类在 Java Hotspot VM 中具有内部表示，并被称为类元数据。

在 Java Hotspot VM 的早期版本中，类元数据被分配在所谓的永久代中。从 JDK 8 开始，永久代已被移除，类元数据被分配在本机内存中。默认情况下，可用于类元数据的本机内存量是无限的。使用以下选项 -XX：最大元空间大小对用于类元数据的本机内存量设置上限。

Java Hotspot VM 明确管理元数据所用的空间。系统会向操作系统请求空间，然后将空间划分为多个块。类加载器会从其块中分配元数据空间（每个块绑定到特定的类加载器）。当类加载器卸载类时，其块会被回收重用或返回给操作系统。元数据使用由以下程序分配的空间：mmap，不是通过malloc。

如果 -XX:使用压缩的Oops已打开并且 -XX:使用压缩类指针被使用，那么本机内存的两个逻辑上不同的区域将用于类元数据。- XX:使用压缩类指针使用 32 位偏移量来表示 64 位进程中的类指针，如下所示 -XX:使用压缩的Oops用于 Java 对象引用。系统会为这些压缩的类指针（32 位偏移量）分配一个区域。该区域的大小可以通过以下方式设置： XX:压缩类空间大小默认为 1 GB。压缩类指针的空间由以下分配空间保留：XX：mmap在初始化时并根据需要提交。-XX：最大元空间大小适用于已提交的压缩类空间和其他类元数据的空间的总和。

类元数据在相应的 Java 类卸载时会被释放。Java 类是垃圾回收的结果，卸载类并释放类元数据时可能会触发垃圾回收。当用于类元数据的空间达到一定水平（高水位线）时，就会触发垃圾回收。垃圾回收结束后，高水位线可能会根据类元数据释放的空间量而升高或降低。为了避免过早触发另一次垃圾回收，会提高高水位线。高水位线的初始设置为命令行选项的值 -XX：元空间大小。根据选项提高或降低 - XX:MaxMetaspaceFreeRatio和 -XX：MinMetaspaceFreeRatio。如果可用于类元数据的已提交空间占类元数据总已提交空间的百分比大于 -XX:最大元空间可用比例，那么高水位线将会降低。如果低于-XX:最小元空间可用比例，那么高水位线将会被提高。

为选项指定一个更高的值 -XX：元空间大小避免因类元数据而引发的早期垃圾回收。分配给应用程序的类元数据的数量取决于应用程序，并且不存在用于选择以下对象的通用准则： XX：元空间大小。默认大小为 -XX：元空间大小取决于平台，范围从 12 MB 到 20 MB 左右。

有关元数据所用空间的信息包含在堆的打印输出中。以下是典型的输出：

[0,296s][info][gc,heap,exit] 堆
[0,296s][info][gc,heap,exit] 垃圾优先堆总计 514048K，已用 0K [0x00000005ca600000、0x00000005ca8007d8、0x00000007c0000000) [0,296s][info][gc,heap,exit] 区域大小 2048K，1 个年轻对象（2048K），0 个幸存者（0K）

[0,296s][info][gc,heap,exit] 元空间已用 2575K，容量 4480K，已提交 4480K，已保留 1056768K
[0,296s][info][gc,heap,exit] 类空间已用 238K，容量 384K，已提交 384K，已保留 1048576K

在以元空间，这用过的值是用于加载类的空间量。容量值是当前分配的块中可用于元数据的空间。坚定的值是块的可用空间量。预订的value 是为元数据保留（但不一定已提交）的空间量。以类空间包含压缩类指针的元数据的相应值。