Java-GC-ref

2022年1月8日　　17:25

-Xss -Xmn
所有参数的默认值，full gc的触发。

https://docs.oracle.com/en/java/javase/17/vm/

https://docs.oracle.com/en/java/javase/17/gctuning/

CMS (Concurrent Mark Sweep)
https://docs.oracle.com/javase/10/gctuning/concurrent-mark-sweep-cms-collector.htm#JSGCT-GUID-FF8150AC-73D9-4780-91DD-148E63FA1BFF

==========

HotSpot Virtual Machine Garbage Collection Tuning Guide
https://docs.oracle.com/en/java/javase/17/gctuning/

Release 17
2021-9

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

Java SE(Standard Edition) selects the most appropriate garbage collector based on the class of the computer on which the application is run.

首先，GC的一般功能和基本调优选项在 串行的，stop-the-world的 收集器的 上下文中进行描述。
然后，介绍其他收集器的 具体特征，以及选择收集器时需要考虑的因素。

GC(garbage collector) 自动管理 应用的动态内存分配(allocation) 请求。

GC通过下面的操作来进行自动动态内存管理：

从OS分配和返还内存
给予应用它要求的内存
决定内存的哪一部分依然被应用使用
回收不使用的内存以便重复使用

Java HotSpot GC 使用多种技术来提高这些操作的效率：
将世代清理(generational scavenging)和老化(aging)结合起来，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
使用多线程来积极地并发操作，或在后台与应用并发地执行一些长时间运行的操作。
尝试压缩活动对象来恢复更大的连续可用内存

Amdahl's law
阿姆达尔定律
当提升系统的一部分性能时，对整个系统性能的影响取决于:1、这一部分有多重要 2、这一部分性能提升了多少。
S = 1/[(1-a)+a/k]
S是系统提升倍数，a是被优化部分占总系统的比例，k是被优化部分优化的倍数。

当一个占总系统60%的部分优化成3倍，则整个系统提升 1/(0.4 + 0.6/3) = 1.6667 倍。

可以看到，即使一个系统的主要部分的性能提升了很多，整个系统的性能提升远远小于这部分的提升。

极端化：60%的部分优化成瞬间完成，系统的性能提升是 1/0.4 = 2.5 倍。

。。。还有一种解释，不过我觉得下面的是对的。。因为Doc上也是并行

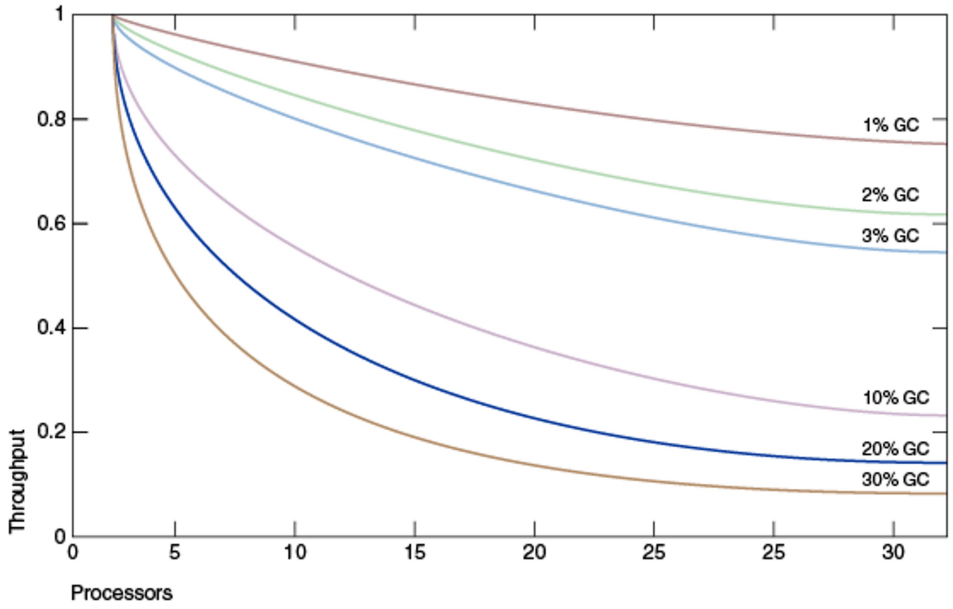
S=1/((1-a)+a/n)
其中，a为并行计算部分所占比例，n为并行处理结点个数。这样，当1-a=0时，(即没有串行，只有并行)最大加速比s=n；当a=0时（即只有串行，没有并行），最小加速比s=1；当n→∞时，极限加速比s→ 1/（1-a），这也就是加速比的上限。

阿姆达尔定律 说明了 大部分工作负载不能完美并行化。
一些部分始终是顺序的，不能从并行中获益。

Java中有4个GC，除了一个(serial GC)，都能并行工作来提升效率。

要尽可能使GC的开销低 使很重要的。

下面使理想系统，除了GC外，该系统具有完美的可扩展性。
红线是一个应用在单处理器系统上只花费1%的时间进行GC，在32核的系统上吞吐量损失超过20%。
洋红线是，单处理器花费10%，当扩展到32核时，超过75%的吞吐量损失



。。估计是因为GC不能并行，所以导致 多核的时候，线程得 全部 停下来，来GC。stop the world。

上图展示了，在小系统上的非常微小的问题，当扩展为大系统时，会变成巨大的问题。
减少这种瓶颈方面做出的 微小改动 可以让性能获得巨大提升。

对于一个充分大的系统，选择合适的GC并进行调整是值得的。

serial(串行) 收集器 对大部分小应用足够了，尤其是那些最高需要100mb堆的应用。

其他收集器有额外开销或复杂度 来获得 更专业的行为。

一种情况下，serial 收集器不是最好的选择，这种情况是：大型，重量级 多线程应用 运行在 有着大内存，2核及以上的机器上。

当应用运行在 服务器级别的 机器上时， 默认使用 Garbage-First (G1) 收集器。

Chapter 2 Ergonomics (人体工程学，人机工程)

Ergonomics 是JVM和GC 的启发方法(如基于行为的启发方法) 提高应用性能的过程

JVM提供了基于平台的 对于 GC，堆大小，运行时编译器 的默认选择。
这些选择符合不同类型的应用的需要，同时需要更少的命令行的调整。
此外，基于行为的优化 动态调整堆大小来 满足程序的行为

- Garbage Collector, Heap, and Runtime Compiler Default Selections

Gargabe-First(G1) 收集器
GC线程的最大数量 受 heap大小和可用cpu资源的 限制
初始heap大小是 1/64 的物理内存
最大heap是 1/4 物理内存
Tiered(分层) compiler, using both C1 and C2。

- Behavior-Based Tuning

Java HotSpot VM GC 能被配置来 优先满足 下面2个目标之一：最大暂停时间 和 应用吞吐量。
如果预定的目标满足了，收集器会尝试最大化另一个。

- Maximum Pause-Time Goal

pause time 是GC 停止应用 恢复不再使用的空间 的 这段时间。

暂停的平均时间和该平均值的差异 有GC维护，平均时间是从执行开始计算的，但是它是经过加权的，因此最近的暂停更重要。 如果暂停时间的平均值加上方差大于最大暂停时间目标，则GC认为没有达到目标。

通过 -XX:MaxGCPauseMillis=<nnn> 来指定。
这个被解释为一个hint，GC调整heap size 和其他参数 来 使得GC pause短于 nnn毫秒。

不同GC的 默认 最大暂停时间 不同。

这个调整可能导致 GC 更频繁地发生，降低 应用吞吐量。

- Throughput Goal

throughput goal 是根据GC花费的时间衡量的，GC之外的时间是 应用时间。

-XX:GCTimeRatio=nnn
GC时间比上应用时间是 1/(1+nnn)。
比如 -XX:GCTimeRatio=19，设置了 5%的时间 给 GC。

如果没有达到吞吐率，GC的可能操作是增加heap大小，这样 暂停时间 会变长。

- Footprint

如果吞吐率和最大暂停时间 都满足了，GC会减少heap大小 直到某个目标无法达到。
最小和最大heap size通过 -Xms=<nnn> -Xmx=<nnn> 来设置。

- Tuning Strategy

堆增加或缩小到 支持所选的 吞吐量目标的大小。了解堆调整策略，例如选择最大堆大小，选择最大暂停时间目标。

不要为 堆尺寸 选择一个最大值，除非你知道你需要一个比默认的最大 还大的 heap size。
而是选择一个 足以满足应用的 吞吐量目标。

应用行为的改变 可以导致 heap的增长或缩小。例如，如果应用开始以更高的速率进行分配，

则堆会增长以保持吞吐量不变

如果堆达到了最大size，但是吞吐量没有满足，那么 最大size 对于这个吞吐量来说 还是太小了。 设置最大size 到接近 物理内存 但不会导致应用交换 的 大小。重新运行应用，如果还是无法达到吞吐率，说明 应用时间的 目标 对于 可用内存 设置得太高了。

如果吞吐率可以达到，但是暂停太久了，那么设置 最大暂停时间。设置最大暂停时间 可能导致无法达到吞吐率目标，所以选择 对于应用来说 是可接受的 折衷值。

当GC尝试满足 冲突的目标时，会导致 heap size的震荡。即使应用已经达到稳定状态，也是如此。实现吞吐率目标(可能需要更大的堆) 与 最大暂停时间和最小占用空间(这2个都需要更小的堆) 相冲突。

Chapter 3 Garbage Collector Implementation

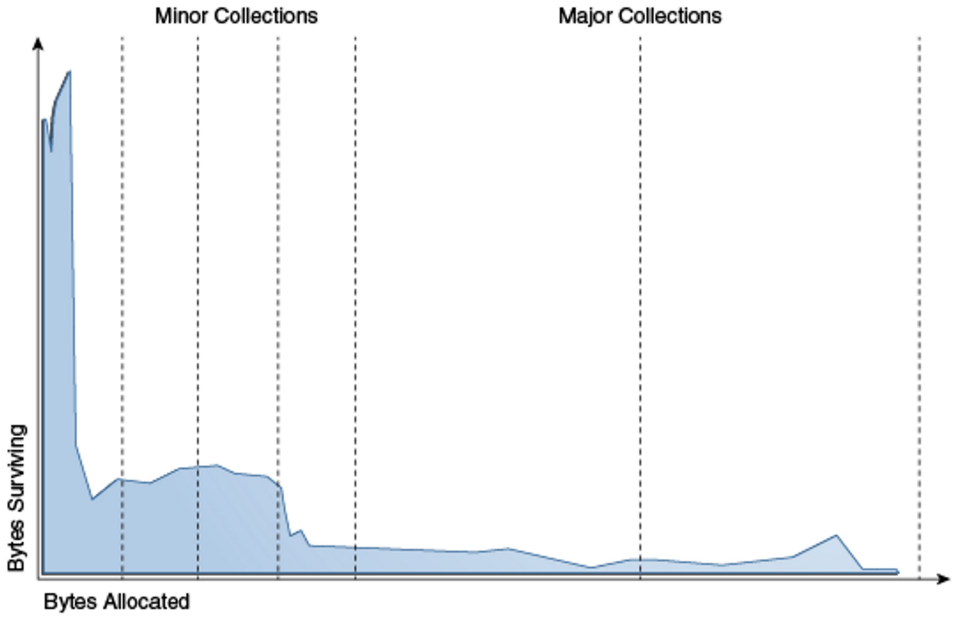
- Generational Garbage Collection

一个对象被视为垃圾，它的内存会被回收，当它 不被 任何运行的程序中存活的对象 引用到。

理论上，最直接的GC算法 每次运行时 遍历所有 可达的对象。其他对象被视为垃圾。这种实现的时间消耗 和 存活对象的数量 成比例，对于维护大量存活数据的大型应用来说是令人望而却步的。

Java HotSpot VM 包含了 一些 不同的GC算法，这些算法除了ZGC，都使用了 generational collection(分代收集) 的技术
之前的naive算法每次测试堆中的每个对象，分代收集利用大多数应用程序的几个经验性属性来最小化 GC 所需要的工作。最重要的属性是 weak generational hypothesis (弱世代假说)，认为大多数对象只存活很短的时间。

下面是一个典型的对象生存时间分布图。x轴是以分配的字节数衡量的对象生命周期，y轴的字节数是具有相应生命周期的对象的总字节数。左侧尖峰表示在分配后不久就可以回收的对象



。。。看不懂。

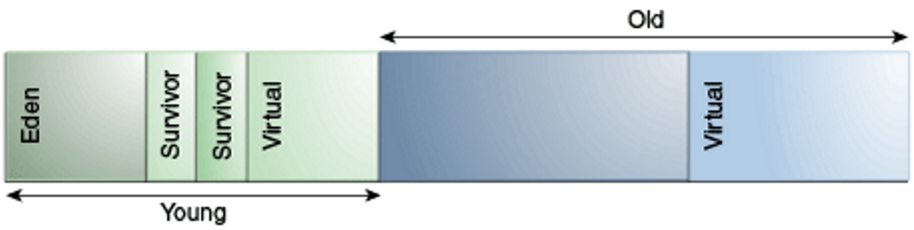
一些对象活得更久，因此分布向右延伸。比如，通常有一些对象在初始化时存活并且直到VM退出。
这2个极端之间是在一些中间计算期间存在的对象，此处视为初始峰右侧的隆起块。
一些应用有非常不同的外观分布，但是大部分应用都具有这种形状。
通过关注大多数对象的"die young(英年早逝)"这一事实，可以实现高效收集。

- Generations

针对这种设想，进行了优化：内存是分代管理的(存放不同年龄对象的内存池)。GC发生，当这代被填满。

绝大多数对象都分配在 专用于年轻对象的池中(年轻代，the young generation)，大部分都在这里死去。
当年轻代满，会导致一个 minor collection，只会对 年轻代进行 GC。
这种收集的消耗 和 被收集的存活对象的数量成正比，一个充满死亡对象的年轻代很快就会被收集起来。
通常，在每次minor collection(次要收集) 期间，来自年轻代的部分幸存对象会被移动到 old generation。
最终，老年代满了，必须被GC，会导致一个 major collection，整个heap 会被GC。
major collection 比 minor collection 持续更长时间，因为 涉及更多的对象。

下面是 serial GC 的默认的 代的安排



启动时，Java HotSpot VM 保留整个Java Heap在地址空间(address space)中，但不会分配物理内存，除非需要。

覆盖Java heap的 整个地址空间 从逻辑上被分为 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 包含 eden 和 2个survivor 空间。大部分对象初始分配在eden中。任何时候，一个survivor是空的，这个在GC时，作为eden 和 另一个survivor 中存活对象 的目的地。GC后，eden 和 source survivor 是空的。下一次GC时，2个survivor 的用途 互换。最近被填充的空间 是复制到另一个空间的 存活对象的来源。对象在2个survivor中复制，直到它们被复制了一定次数 或 没有足够的空间。这些对象被复制到 老年代。这个过程被称为aging

- Performance Considerations 性能度量

GC的主要度量是 吞吐量 和 延迟

吞吐量是一段较长时间内 非GC时间的 比例。吞吐量包含分配所花费的时间(但通常不需要调整分配速度)

延迟是应用的反应。GC的pause 会影响 应用的反应

用户对GC 有不同的要求。例如，一些用户认为web服务器的正确指标是吞吐量，因为GC的 暂停是可以容忍的，或者会被网络延迟所掩盖。但是在交互式图形程序中，即使短暂的停顿也会对用户体验产生负面影响。

一些用户对其他度量敏感，Footprint 是进程的工作集，以page 和 缓存行为单位。
在物理内存有限 或进程众多的系统上，占用空间(footprint)可能决定伸缩性。
Promptness(及时性) 是 对象死亡 与 内存重新可用 之间的时间，对于分布式系统是一个重要度量，包括RMI remote method invocation

一般来说，为特定的某代 选择尺寸 是 这些度量的权衡。例如，一个非常大的年轻代可能 最大化吞吐量，但这样做是牺牲了 footprint,promptness,pause time。年轻代pause 可以被降低，通过使用一个 小的年轻代，牺牲了 吞吐量。 代的大小不会影响另一个代的收集频率和暂停时间。

没有一种正确的方法来选择 代的大小。最佳选择 取决于应用使用内存的方式以及用户需求。

- Throughput and Footprint Measurement 吞吐量和footprint测量

吞吐量和 占用空间 最好使用特定于应用程序的指标 来衡量。

例如，web服务器的吞吐量 可以被测试 通过 一个客户端负载生成器。但是，通过检查GC本身的诊断输出 很容易就能估计出 GC pause。命令行参数 -verbose:gc，打印每次gc时，heap 和 gc的信息，如：

```
[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
```

上面的输出显示了 2次 年轻代收集，然后一次 System.gc() 的 full gc。
行首是时间戳 表示 应用启动到现在的时间。接下来是 日志级别(info) 和 标记(gc)。然后

是 GC的ID，这个例子中是36，37，38。然后是 GC类型和 GC原因。然后，一些有关内存消耗的信息，格式是 "used before gc" -> "used after gc" ("heap size")

第一行是 239M->57M(307M)，意味者 gc前使用239mb，gc清除了大部分内存，但只有57mb 存活。堆大小是307mb。full gc 把堆从307 减少到 104mb。在内存使用信息之后，记录gc的开始和结束时间以及持续时间。

-verbose:gc 是 -Xlog:gc 的别称。-Xlog是用在HotSopt JVM中进行 日志记录的 通用日志记录配置选项。这是一个基于tag的系统，gc是tag之一。为了获得更多 关于GC做了什么 的信息，你可以配置 日志来 打印有gc标记和其他tag 的信息。命令行参数是 -Xlog:gc*

下面是 G1 年轻代收集 ，使用了 -Xlog:gc* 参数：

```
[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
```

chapter 4 Factors Affecting Garbage Collection Performance GC性能因素

影响GC性能的最重要因素是 总可用内存 和 年轻代的堆的比例。

- Total Heap

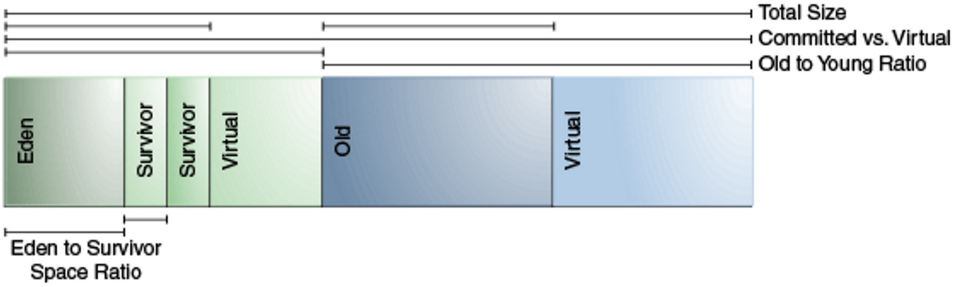
最终要的GC性能因素是 总可用内存。因为GC只有当 代 充满时发生。吞吐量和可用内存量成反比。

下面关于 堆的增长和收缩、堆布局，默认值的讨论是 使用serial 收集器作为例子的。
虽然其他收集器使用类似的机制，但是这里提供的详细信息可能不适用于其他收集器。

- Heap Options Affecting Generation Size

许多选项影响 代的大小。
下图说明了 堆中 committed 空间 和 virtual空间的区别。
在JVM初始化时，heap的整个空间被保留。保留的空间的大小可以通过 -Xmx 来指定。
如果-Xms 参数 小于 -Xmx，那么不是所有的保留空间都会立刻提交给虚拟机。没有committed的空间被标记为 virtual。heap的不同部分，即老年代 和年轻代 可以在必要时 增长到 virtual 空间的极限。

一些其他参数 设置 堆中 不同部分的比例。例如， -XX:NewRatio 表示了 老年代 对于 年轻代 的相对大小。



- Default Option Values for Heap Size

默认下，虚拟机 每次GC时，增长或减少 heap 来尝试 保持每次收集时 可用空间 与活动对象的比例 在一个特定范围内。

目标范围是一个 百分比，通过 -XX:MinHeapFreeRatio=<minimum> 和 -XX:MaxHeapFreeRatio=<maximum> ，总大小通过 -Xms<min> -Xmx<max>

通过这些选项，如果 代 中的空闲空间比例 小于40%，代 会扩展 以维持40%空闲空间，直到 代 允许的最大size。 类似的，如果 空闲空间超过70%，代 会收缩 保证只有70%空闲空间，直到 代的最小sz。

Java SE中用于 并行收集器的 计算 现在用于所有的 GC。计算中的一部分 是64位平台的 最大heap size 上限。 客户端JVM也有类似的计算，这导致 最大堆size 小于 服务器JVM。

下面是有关服务器应用堆大小的一般准则：
除非pause会带来问题，否则应尝试给予JVM尽可能多的内存。默认的size太小了。
设置 -Xms -Xmx 为相同值，来删除JVM中关于堆大小的计算/决定，以增加可预测性
一般情况下，增加CPU的时候增加内存，因为 allocation 能并行。

- Conserving Dynamic Footprint by Minimizing Java Heap Size

如果你需要最小化 动态内存footprint(在执行时计算最大RAM) for 你的应用，那么你可以通过 最小化heap size 来达到这种目的。 Java SE 嵌入式应用可能需要这个。

最小化Java heap size 通过 降低 -XX:MaxHeapFreeRatio (默认70%) 这个选项的值，和降低 -XX:MinHeapFreeRatio (默认40%) 的值。将 Max 降低到10% 和 降低Min 已经被证明了可以成功地减少 堆大小 而不会造成太多的性能下降。当然，具体结果还是依赖于你的应用。尝试不同的 尽可能低 的 同时又满足性能的 值。
。。。没有 min 降低到多少。。估计0？还是 都是10% ？

另外，你可以 指定 -XX:-ShrinkHeapInSteps， 这个会立刻降低 heap到 -XX:MaxHeapFreeRatio 指定的 size。 使用这个设置 会导致性能下降。默认下，java runtime 逐步降低heap到目标size；这个过程需要多次GC。

- The Young Generation

在 总可用内存之后，第二对GC性能有影响的是 年轻代的比例

年轻代越大，minor gc的频率越低。但是对于一个 确定的heap size，年轻代大了，老年代就小了，这会增加major gc 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。

- Young Generation Size Options

默认下，年轻代size 通过 -XX:NewRatio 控制

例如，设置 -XX:NewRatio=3 意味者 年轻代，老年代比例是 1:3。也就是说 eden+survivor 占了 heap的 1/4

-XX:Size, -XX:MaxNewSize 限制了 年轻代的 下界和上界。设置相同的值 来固定 年轻代大小，就像-Xms -Xmx 设置为相同值来 固定heap 大小 一样。 这个比 只能整数调整的-XX:NewRatio 更有用，更精细。

- Survivor Space Sizing

你可以使用 -XX:SurvivorRatio 来调整 survivor空间，但大多数时候 这个对于性能 并不重要。

例如 -XX:SurvivorRatio=6，设置 eden 和 一个survivor 比例为 1:6 (? 应该是6：1)。即，每个 survivor 是 1/6 的eden， 就是 年轻代的1/8 (因为有2个 survivor)。

如果survivor 太小，那么 复制收集(copying collection)直接 溢出到 老年代。如果太大，那么有 无用的空间。
每次gc时，vm选择一个 临界值，这个是 一个对象在 变老之前 最多可以复制几次。这个值用来 使得 survivor 半满。 你可以使用 -Xlog:gc,age 来显示这个 阈值 和 新生代中的对象的年龄 。 也可以用于 观察应用的 生命周期分布。

参数	默认值
-XX:NewRatio	2
-XX:NewSize	1310mb
-XX:MaxNewSize	无限制
-XX:SurvivorRatio	8

年轻代的最大值 通过 total heap最大值 和 -XX:NewRatio 来计算。 -XX:MaxNewSize的 无限制 意味者 计算出来的值 不受 -XX:MaxNewSize 限制，除非 命令行 显示 设置-XX:MaxNewSize。

服务器应用的一般准则：

首先决定 你可以为JVM 负担的 最大heap size。然后，将你的 性能指标和年轻代大小进行对比，来找到最大设置。

记住，最大heap size 总是 小于 物理内存 以避免 过多的 page错误和 抖动。

如果total heap size 确定了，那么增加 年轻代 就会 减少 老年代。 确保 老年代 任何时候 都 足够放入 所有的 存活的数据，以及一些 松弛空间(10%-20%或更多)

受限于前面对老年代的约束：

为 年轻代 提供 足够的内存

随着cpu的增加，年轻代的大小也要增加，因为 分配是可以并行的。

- 5 Available Collectors

到目前为止 都是讨论的 serial 收集器。
HotSpot 有3种不同类型的 收集器，有不同的性能特点。
。。但是下面有4种， serial ， parallel， g1， z

- Serial Collector

使用 单线程 来 执行所有的 gc工作，使得它相对高效，因为没有线程间通信。

最适合 单CPU的机器，因为它 不能利用 多CPU，尽管对于 小数据集(100MB以内) 的应用，它在多CPU 上很有用。 在某些硬件 和 OS 上默认 选择 serial collector，或者可以通过显式指定：-XX:+UseSerialGC

- Parallel Collector

并行collector 也被称为 吞吐量collector。它是类似 serial收集器的 分代收集器。串行和并行 收集器的 主要区别是 并行收集器 有多个线程 用来 加速gc。

并行收集器 适用于 在多CPU 或多线程 硬件 上运行的中型到大型数据集的 应用。通过-XX:+UseParallelGC 激活

parallel compaction(压缩) 是一个特点，可以让 并行收集器 并行地执行major collection。 没有 并行压缩，major collection 通过 单线程 执行，极大限制了 可扩展性。 如果指定了-XX:+UseParallelGC，并行压缩 默认启用。禁用通过-XX:-UseParallelOldGC.

- Garbage-First (G1) Garbage Collector

G1 是一个 主要并发(mostly concurrent) 的收集器。 Mostly concurrent collectors 并发地 执行一些昂贵的任务。 这个收集器 被设计 可以用于 从小型机器 到 具有大量内存的大型多CPU机器。 它能在提供高吞吐量的同时 大概率满足 pause-time 目标。

大部分的硬件 和 OS 默认使用 G1，或者可以通过显式指定： -XX:+UseG1GC

- The Z Garbage Collector

简称 ZGC，是一个可扩展的低延迟垃圾收集器。ZGC并发执行所有昂贵的工作，不会使得 应用暂停。

ZGC提供了 最多 几毫秒的 pause-time，但是 降低了一些 吞吐量。 用于需要 低延迟的 应用。 pause time 和堆大小 无关。 ZGC 支持堆大小 从 8mb到 16tb。 激活通过 -XX:+UseZGC

- Selecting a Collector

除非你的应用 有严格的 pause-time 要求，不然就 直接允许应用 让VM选择收集器。

如果有必要，修改heap size 来提高性能。如果性能 仍无法满足，那么使用下面的 指导方针作为 选择收集器的 起点：

如果 应用有一个 小的数据集(最多100mb)，那么使用 serial collector，通过-XX:+UseSerialGC

如果 应用在 单核CPU 的机器上运行，且没有 pause-time 要求，那么使用 serial collector

如果 (a)峰值应用性能是 首要的 且 (b)没有pause-time 要求或 1秒及更长是 可以接受的，那么让VM 选择 collector，或者 显式使用prallel collector: -XX:+UseParallelGC

如果 响应时间 比 整体吞吐量更重要，且 pause 必须更短，那么使用 mostly concurrent collector， -XX:+UseG1GC

如果 响应时间非常重要，那么使用 fully concurrent collector，-XX:+UseZGC

这些指南 提供的 只是选择收集器的开始， 因为性能 依赖于 heap size，存活数据的总数，CPU的数量和性能。

如果推荐的收集器不能满足 性能要求，那么 首先尝试调整 heap和 代 的size。如果性能还是不够，那么尝试 其他的收集器：使用 concurrent 收集器来 减少 pause-time，使用 parallel 收集器 在 多核CPU 上 增加吞吐量。

- chapter 6 The Parallel Collector

parallel 收集器(这里也被称为 throughout 收集器) 是一个 分代 收集器，类似 serial 收集器。 主要的区别是 parallel 使用多线程 来提升 gc速度。

parallel collector 通过 -XX:+UseParallelGC 激活。默认下，minor，major 收集 并行执行，以进一步减少 gc 开销

。。。并行 是指 一根minor ，一根major 一起？ 还是 minor有多根线程，major有多根线程？ 感觉是 后者。..后者，下面就是 线程数。。

- Number of Parallel Collector Garbage Collector Threads

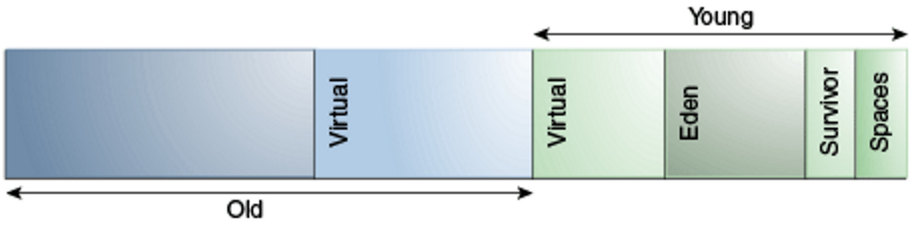
在一个 有 N个 hardware thread (硬件线程，可能是指CPU？) 且 N > 8的 机器上，paralle 收集器使用 一个固定系数 乘以 N 来得到 gc线程的数量。

对于 大的N，系数大约是 5/8， 如果N < 8，数字就是 N。 在特定的平台上，系数降到 5/16。 gc线程数量可以通过 命令行参数修改。 在 单CPU的主机上，parallel 收集器 可能比 serial 收集器性能更差，因为并行执行的开销。 但是，当运行的应用有着 中等到 大的 heap，在 双核电脑上，paralle 比 serial 更好。 双核以上的时候，paralle 比 serial 好很多。

gc线程 数量 通过 -XX:ParallelGCThreads=<N> 来控制。如果你使用命令行调整 堆，那么 parallel 需要的 heap size 和 serial 的相同。当然，parallel 的 pause-time 更短。因为minor collection 的时候 是 多线程 垃圾收集器 参与的，由于 收集期间 从年轻代 到老年代的 提升，可能产生一些碎片。minor收集时的 每个线程都会 reserve(预定，保留) 一部分 老年代 用于 对象的升代(young->old)，这种可用空间的划分会导致碎片。减少 gc线程数量，增加老年代大小可以降低这种碎片效果

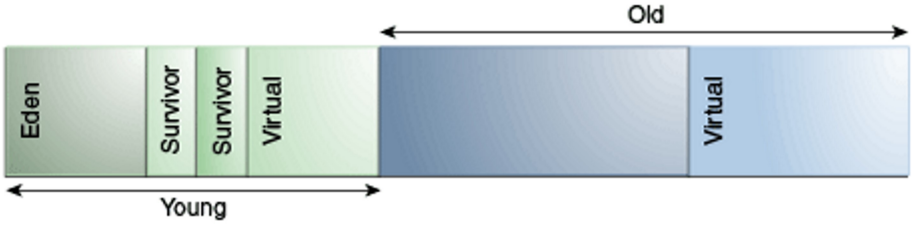
- Arrangement of Generations in Parallel Collectors

代 的分配在parallel 中不同。



。。没有具体说。感觉是 顺序(这里先 old，再young)，还有 这里只有一个 survivor。

。。。下面是 之前的 serial 的 堆划分。



。。还有增长，serial 如果young 增长，需要 移动 survivor 。 但是 parallel 似乎不需要。不过，不清楚 survivor 是不是也要增长，应该是要的。。 还有 最后的 spaces 是什么意思？

- Parallel Collector Ergonomics

通过 -XX:+UseParallelGC 使用 parallel 收集器，它也会 启用 用于自动调节的 方法，允许你 指定行为 而不是 代 大小和其他 低层 的 调整细节。

- Options to Specify Parallel Collector Behaviors

可以指定 最大gc pause-time，吞吐量，footprint。

最大gc pause-time：-XX:MaxGCPauseMillis=<N>，被解释为一个 提示，pause time需要小于 这个 时间。 默认，没有 pause time 限制。 如果指定了pause time目标， heap size 和其他 gc的参数 会被调整 来 维持 gc的pause time 小于这个指定值；当然，可能永远 无法 满足 这个pause time goal。这些调整 可能导致 垃圾收集器 降低 应用的

整体吞吐量。

吞吐量：被量化为 gc时间 对比 非gc时间。 -XX:GCTimeRatio=<N>，这个会设置 gc时间为 整个应用时间的 1/(1+N)。
例如，-XX:GCTimeRatio=19，设置 总时间的 1/20 (5%) 用于 gc。默认是99，即 1% 的时间用于 gc

footprint：堆最大size通过 -Xmx<N>。此外，只要满足其他目标，收集器就有一个隐含的目标，这个目标是最小化 堆的大小。

- Priority of Parallel Collector Goals

目标是：最大pause time，吞吐量，最小footprint，顺序是：
先满足 最大pause time，满足后 解决 吞吐量目标，前2个满足后，考虑footprint。

- Parallel Collector Generation Size Adjustments

collector 维护者 统计数据，如平均pause time， 这些统计数据会在每次 收集的最后 被更新。

测试 来决定 是否已经达到 目标，并对 代 的大小进行必要的调整。 例外的是 显式gc，如System.gc()，触发的 gc 不会 加入到统计数据，也不会 触发 代的调整。

代的 增长和缩小 通过 增加 代的 固定百分比(根据后面，这个应该是一次调整 百分之多少。) 来实现的， 这样代会 逐步增加或减少到 设定的 大小。 增长和缩小 通过不同的 比例 来做。 默认下，增加是20%，缩小是5%。 增长的百分比 通过 -XX:YoungGenerationSizeIncrement=<Y> 控制年轻代，通过-XX:TenuredGenerationSizeIncrement=<Y> 来控制 老年代。 缩小的百分比 通过 -XX:AdaptiveSizeDecrementScaleFator=<D> 来控制，如果增加的比例是X%，那么缩小的比例是 X/D%

如果在启动时，收集器决定增长 代，那么 有一个补充百分比被添加到增量中。这个补充量随着 gc的数量增加 而减少，没有长期的 影响。 这个补充量 是为了提高 启动性能。缩小没有补充百分比。

如果 pause time目标没有达到，那么 一次只 缩小 一个代 的规模。如果2个代的 pause time 都达到了，那么 pause time 大的那个代 先缩小。

如果 吞吐量 没有达到，那么 2个代 都增加。每个都根据 它对 总gc时间的 贡献 来增加。例如，如果 年轻代的gc时间 占了 总gc时间的 25%， 且如果 年轻代的 一次 full 增加 是20%，那么 年轻代会增加 5%

- Parallel Collector Default Heap Size

除非 指定了heap size的 初始 和 最大值，否则它们会 根据机器的 内存 计算。默认 最大heap size 是 1/4 物理内存，初始内存是 1/64 物理内存。 年轻代的 最大空间 是 总heap size的 1/3

- Specification of Parallel Collector Initial and Maximum Heap Sizes

-Xms (initial heap size), -Xmx (maximum heap size).

如果你直到 应用需要多少堆，那么你可以 设置-Xms ，-Xmx 为相同的值。 如果你不知道，那么JVM 会从initial heap size 开始，增长java heap 直到 heap 大小 和 性能的平衡。

其他参数也可以 影响默认值， 要确定你的 默认值， 使用 -XX:+PrintFlagsFinal ，然后在输出中 查看 -XX:MaxHeapSize。 例如，在linux ，你可以执行下面的命令：
java -XX:+PrintFlagsFinal <GC options> -version | grep MaxHeapSize

- Excessive(过多) Parallel Collector Time and OutOfMemoryError

如果太多的时间用于gc，parallel collector 会抛出 OutOfMemoryError。

如果超过 98% 的时间用于 gc，并且 小于2%的heap 被恢复，那么 抛出 OutOfMemoryError。这个功能 旨在 防止 应用长时间运行，但由于堆太小 而几乎没有进行进展。 如果有必要，这个功能可以通过 -XX:-UseGCOverheadLimit 来 禁止。

- Parallel Collector Measurements

parallel 的输出 和 serial 的基本一样。

- chapter 7 Garbage-First (G1) Garbage Collector

- Introduction to Garbage-First (G1) Garbage Collector

G1 垃圾收集器 的目标是 多CPU 机器 扩展到 大内存。它试图 在高概率 满足pause time，且 几乎不需要配置的情况下 实现 高吞吐量。g1 旨在 使用当前目标应用程序和环境 提供的延迟和 吞吐量 间的最好 平衡，功能包括：

堆大小 高达 几十gb 或者更大，超过50%的 堆 被存活数据 占用
随时间变化的 对象分配 和 晋升 率。
堆中有大量碎片
可预见的 pause time 目标 不会大于 几百ms，避免 长pause。

G1 执行 它的部分工作 在app运行时。它 获得 本该app使用的 处理器资源 来减少 pause。

这在应用运行时 使用一个或多个gc线程时最为明显。对比吞吐量收集器，g1的pause 短很多，吞吐量 降低一点点。

G1 是默认收集器

G1收集器实现了 高性能，并尝试通过以下几节 中描述的 方式来满足 pause-time 目标。

- Enabling G1

garbage-first garbage collector 是默认 收集器，所以通常 你不需要额外的动作。可以显式激活：-XX:+UseG1GC。

- Basic Concepts

G1 是一个 分代的，递增的，并行的，大部分并发的，stop-the-world，疏散/撤离的 逻辑收集器，在每次 stop the world 暂停中监控 pause time 目标。类似于其他收集器，G1 将堆分为(虚拟的) 年轻代和老年代。 空间回收 集中于 最有效的 年轻代，偶尔在 老年代回收空间。

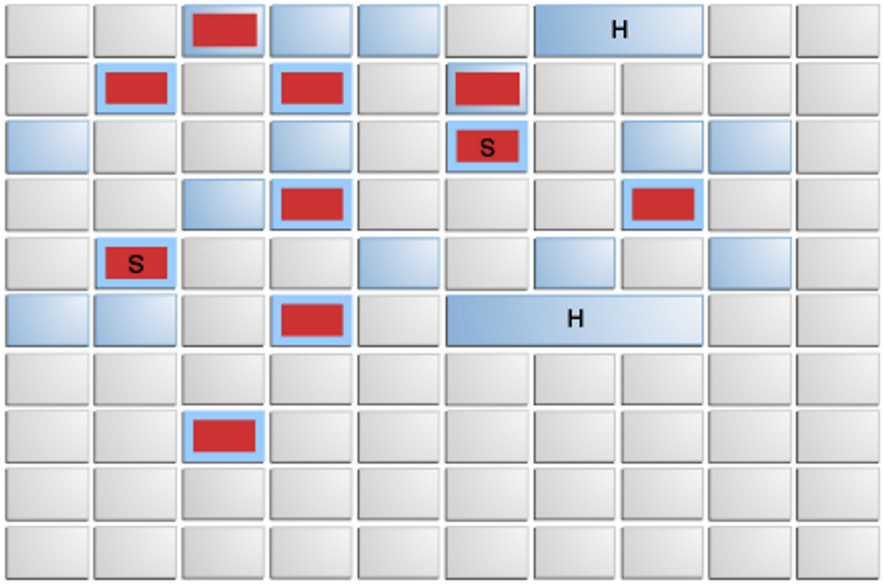
某些操作 总是在 stop the world 暂停中执行 以提高吞吐量。其他操作(如果在pause的时候执行需要更多时间的操作(如 全局标记等 对整个堆的操作)) 与应用并行。 为了让 空间回收的 pause 更短，G1 并行，逐步 执行空间回收。 G1 通过追踪 应用先前行为 和gc pause 来构建一个 成本模型，来实现 可预测性。 它使用 此信息 来确定在暂停中完成的工作的大小。例如，G1 首先在 最高效的区域回收空间(那是大部分被垃圾填满的区域)

G1 主要通过 evacuation(疏散) 回收空间：在选定的 要回收的 区域中找到的 活着的对象被复制到 新的内存区域，并在此过程中 压缩它们。 在 evacuation 完成后，存活对象 之前占据的 空间被回收，被重用。

G1 收集器 不是一个 实时收集器，它尝试 在较长时间内 以高概论 达到设定的 pause time 目标，但是 对于给定的 pause 并不总是 绝对确定。

- Heap Layout

G1 划分heap 成 等大小的heap 区域的集合。每个虚拟内存的 连续范围 就像下面：
一个区域是 内存分配 和回收的 单元。在任何时间，这些区域的 每个 可能是 空的(灰色标识)，或分配给 某一代(年轻或老年代)。 内存请求进来后，内存管理 发放 空闲区域。内存管理 分配它们(。。估计指空闲区域)到 某个代，然后 返回它们 给应用，应用可以自行分配到其中。



年轻代 包含eden (红色)，survivor(红色+S)， 这些区域提供了 其他收集器 中相应的 连续

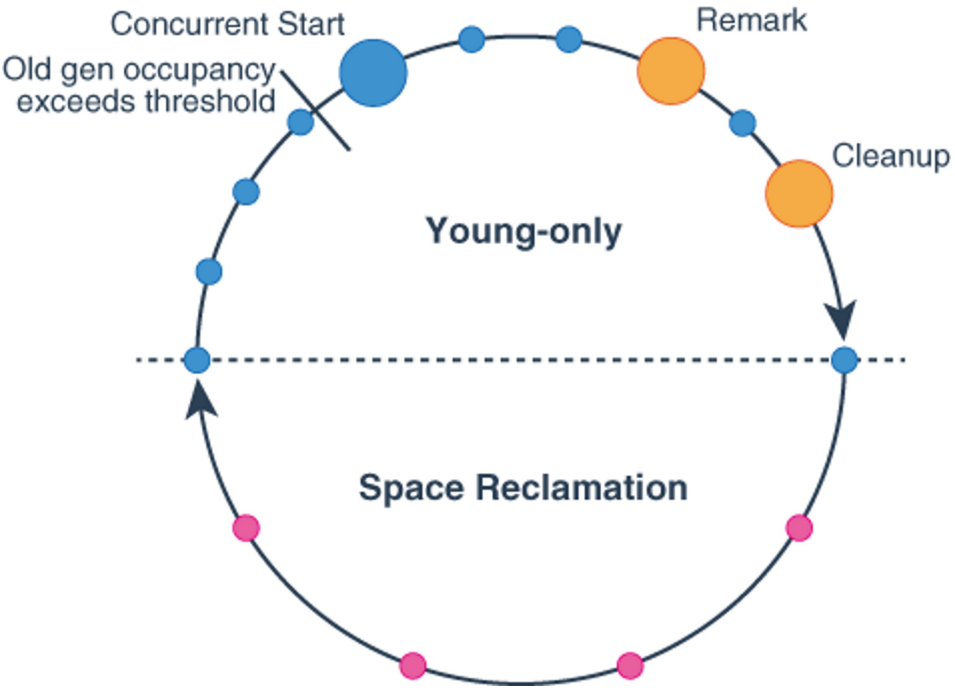
空间 相同的功能，不同是，G1这些区域 通常 在内存中 是不连续的。老年代(蓝色) 可能是巨大的 (蓝色+H)，横跨多个区域。

应用总是分配到 年轻代，即eden，除了超大对象 会直接分配到 老年代。

- Garbage Collection Cycle

在高层中，G1 收集器 在2个 阶段交替。young-only 阶段 包含 gc，gc会逐渐用 老年代中对象 填充当前可用的内存。space-reclamation 阶段是 G1 逐步回收 老年代中的空间，还处理年轻代。 然后继续从 young-only 开始。

下面是循环的例子



下面是gc阶段的描述，它们的pause 和阶段 之间的 转换

young-only阶段：这个阶段 从 一些 将对象提升到老年代的 正常 年轻代 收集开始，当老年代占用率 达到一定阈值(初始堆占用阈值)，开始 young-only 和 space-reclamation 的转换。此时，G1 使用 并发 年轻代收集 而不是 普通的 年轻代收集。

concurrent start：这种类型的 收集 除了普通的young 收集外，还会启动 标记过程。并发的标记过程 确定 老年代中 所有可达(存活) 的对象，以便在 接下来的 space-reclamation 阶段保留。在 标记 没有完全结束前，普通 young 收集可能发生。标记结束 有2个 pause：Remark 和 Cleanup。

Remark: 这个暂停 最终确定 标记，执行 全局引用处理 和 类卸载，回收完全空的区域 并清理内部数据结构。 在remark 和 cleanup 间 G1 会计算信息，以便稍后能 并发回收 选定的老年代中的空闲空间，这将在 cleanup 暂停中 完成

Cleanup: 这个暂停决定 稍后 是否会有 空间回收阶段。如果有 空间回收，young-only结算 将以一个Prepare Mixed young collection 结束。

space-reclamation阶段：包含 多个混合收集，除了年轻代外，还evacuate 老年代区域中 存活的对象。 空间回收阶段结束，当g1 确定 evacuate 更多的老年代空间 也不会产生足够的 空闲空间 时。

space-reclamation 后，gc循环重新开始。 作为备份，如果应用在 收集存活信息时 内存不足，G1 就像其他收集器一样 执行一个 in-place, stop-the-world full heap compaction(Full GC)

- Garbage Collection Pauses and Collection Set

g1 在 stop the world 暂停中 执行 gc和空间回收。 存活对象通常 从堆的 源区域s 复制到1个或多个 目的区域s，对这些被移动的对象的 ref 也会被调整。

对于 不是巨大的 区域，对象的目的地区域 由 它的源区域决定：
年轻代(eden+survivor)的 被复制到 survivor 或 老年代，基于它们的 年龄
老年代的 被复制到 老年代

巨大区域的对象 和上面不同， g1只决定 它们是否存活，如果它们已死亡，回收它们的内存。巨大区域中的对象 不会被 g1 移动.

collection set 是 将要被回收空间的 源区域 的集合。根据gc类型的不同，collection set 包含了不同的 区域：
在 young-only 阶段，collection set 只包含 年轻代的区域，和 包含可能被回收的对象的 巨大区域
space-reclamation 阶段，collection set 包含 年轻代的区域， 包含可能被回收对象的 巨大区域，collection set 候选区域的一些 老年代区域。

g1 准备 collection set 候选区域 during并发周期中。 在remark阶段，g1 选择 低占用率的区域， 在remark 和cleanup 之间，这些区域被 并行准备 for稍后的收集。cleanup pause 排序这些区域 by 它们的效率。在随后的mixed collection中， 更高效的区域 看起来需要更少的 时间 收集，包含更多的 free空间。

- Garbage-First Internals

本节描述 g1 一些重要的 细节。

- Java Heap Sizing

g1 在调整 heap 大小时，遵守标准规则，-XX:InitialHeapSize 作为最小heap size，-XX:MaxHeapSize 作为最大heap size，-XX:MinHeapFreeRatio 作为最小空闲百分比，-XX:MaxHeapFreeRatio 是可用内存的 最大百分比。
g1 只在 remark 和 full gc 暂停中考虑 调整heap大小。 这个过程可能导致 向OS 申请或释放 内存。

- Young-Only Phase Generation Sizing

g1 在 普通young collection的最后 调整 年轻代大小 for 下一个阶段。这样，g1可以满足通过-XX:MaxGCPauseTimeMillis 和 -XX:PauseTimeIntervalMillis 根据实际暂停时间的长期观察 设置的 pause time 目标。它考虑了类似规模的 年轻代 evacuate 要多久。这包括以下信息：收集期间 多少对象需要被复制，这些对象之间的 关联关系 等

如果没有其他的约束，g1 会调整 年轻代的大小的值 在 -XX:G1NewSizePercent 和 -XX:G1MaxNewSizePercent 之间 且 满足 pause time 目标。

或者 -XX:NewSize， -XX:MaxNewSize 能用于 指定 最小和最大 年轻代。

仅指定 后面选项中的一个 来将年轻代大小 固定为 通过-XX:NewSize, -XX:MaxNewSize 传递的值。这将禁用 pause time 控制。
Note:Only specifying one of these latter options fixes young generation size to exactly the value passed with -XX:NewSize and -XX:MaxNewSize respectively. This disables pause time control.

- Space-Reclamation Phase Generation Sizing

在空间回收阶段，g1尝试在 一次gc暂停中 最大化老年代中 回收的空间量。 年轻代的大小 设置为允许的最小值，通常通过 -XX:G1NewSizePercent 决定

在这个阶段的 每个 mixed collection 开始时，g1 从 collection set 候选中 选择 一组区域 添加到 collection set。这组额外的 老年代区域 由3部分组成：

保证evacuation 进度的 老年代 的最小 集合。 这个老年代 区域的集合 是由 collection set 候选中的 区域数量 除以 由-XX:G1MixedGCCountTarget 决定的 空间回收阶段长度 确定的。

如果g1预测 收集 最小集合后还有时间，那么会从 collection set候选中 选取 额外的老年代区域。 添加老年代区域 直到 剩余时间的80% 被使用。

在上面2部分 被evacuate后 还有时间，那么g1 会逐步 evacuate 一组可选的collection set区域

头2个区域集合 在 初始收集过程中 被收集，来自可选collection set的 额外区域 在剩余的 pause time 中。 这个方法确保 空间回收的执行 且 提高 维持pause time的概率 和 最小化 optional collection set 的管理 造成的 开销。

当collection set 候选区域 中可以被回收的 空间总数 小于-XX:G1HeapWastePercent 设置的百分比时， space reclamation 结束。

- Periodic(周期) Garbage Collections

如果由于 app 不活动，长时间没有gc，jvm可能长时间 保持着 大量无用的内存。为了避免这个，g1 能被 定时强制gc，通过 -XX:G1PeriodicGCInterval。 这个决定g1考虑gc时 的最小间隔毫秒数。 如果从上次 gc pause 开始 经过 设置的 ms，且 没有 concurrent cycle 在执行，g1触发 额外的gc， 可能产生 下面的影响：

在 young-only阶段：g1开始一个使用并发start pause的 并发marking 或者 如果-XX:-G1PeriodicGCInvokesConcurrent 被指定了 就 full gc

在space reclamation 阶段：g1继续 空间回收阶段 触发 适合当前进度的 gc pause

-XX:G1PeriodicGCSystemLoadThreshold 用来 改进(refine) gc是否被触发：如果 jvm主机系统上的 getloadavg() 返回的 平均一分钟的 系统负载 高于此值，则不会 运行 periodic gc。

- Determining Initiating Heap Occupancy

The Initiating Heap Occupancy Percent (IHOP) 是临界点 用于 第一次的Mark collection

的触发，被定义为 老年代的 百分比。

默认下，g1自动决定 最佳的 IHOP 通过观察 marking 的耗时 和 marking周期中，老年代通常分配多少内存。 这个功能叫做 Adapative IHOP。
如果这个功能启动着，那么 -XX:InitiatingHeapOccupancyPercent 决定 最初值 as 当前老年代的百分比， 只要没有足够的观察值来对 IHOP 进行良好的预测。
关闭G1的这个功能：-XX:-G1UseAdaptiveIHOP。这种情况下，-XX:InitiatingHeapOccupancyPercent 始终是 临界点。

在内部，Adaptive IHOP 尝试设置 initiating heap occupancy，这样，space-reclamation 阶段的 第一次mixed gc 开始，当 老年代占用率是当前老年代最大值 减去 -XX:G1HeapReservePercent(作为额外缓存区) 的值

- Marking

G1 marking 使用算法 称为 Snapshot-At-The-Beginning(SATB)，
在最初的mark pause阶段(。或者可能是mark pause阶段的最开始时)，它获得 堆的 虚拟的 snapshot，当marking 开始时 存活的对象 都被认为 存活在marking剩余阶段。
这意味者 在marking 期间 死亡的对象 依然被认为是 活的 for space-reclamation 的目的(有一些例外)。 和其他收集器相比， 这个可能导致 一些额外的内存被错误地保留。
不过，SATB 在 remark pause 期间 能提供更好的延迟。
在marking 期间由于 过度保守的策略 而导致存活的对象 在下次marking时被回收。

- Behavior in Very Tight Heap Situations

当应用 使用了很多内存 导致 无法找到 足够空间 来复制，evacuation 失败 发生了。
evacuation 失败意味者 g1 尝试完成当前gc 通过 保持已经移动的对象 在它们的新位置，对于没有复制没有移动的对象 只是调整引用。
evacuation 失败可能 导致额外的开销，但是通常 和其他young collection 一样快。
在这次 evacuation 失败的 gc后，g1会恢复app ，和平时一样，不需要其他措施。
g1假设 evacuation 失败 发生在 gc 快结束时；这样，大部分对象已经被移动了，有足够的空间 来继续 运行 app 直到 marking 结束 且 space-reclamation 开始。

如果不能满足这个假设，g1 最终开始 full gc。这个类型的gc 会元素压缩整个 heap，这可能非常慢。

- Humongous Objects

指 >= 区域 一半的对象。 当前region size 通过 后续的 Ergonomic Defaults for G1 GC 中 ergonomically 决定， 除非使用了 -XX:G1HeapRegionSize

执行巨大对象 有时会以特殊的方式处理：

> 每个巨大对象被 分配为 老年代中的一系列连续区域。对象的开始就是连续区域的 第一个区域的开始。连续区域的 最后一个区域的 剩余空间 不会再被分配，直到整个对象被回收。
>
> 一般来说，巨大对象只能在 Cleanup pause 期间的 marking结束时 被回收，或 full gc

时(如果它不可达)。然而，对于 原始类型(比如，bool，所有的整型，浮点型) 数组 的巨大对象有一个特殊的规定。 g1投机地尝试 回收巨大对象 如果在任何类型的gc pause时，它们没有被许多对象引用。这个行为 默认开启，但你可以 禁用它： -XX:G1EagerReclaimHumongObjects

巨大对象的分配 可能导致 gc暂停 过早发生。g1 在每个巨大对象分配时 检查 Initiating Heap Occupancy threshold，可能立刻强制一个 initial mark young collection，如果当前 占有率 超过 threshold。

- Ergonomic Defaults for G1 GC

这里提供 特定于g1 的 最重要的默认配置和它们的默认值。它们粗略地概述了 在没有额外配置的情况下，使用g1的预期行为 和 资源使用情况。

参数和默认值	描述
-XX:MaxGCPauseMillis=200	最大pause time goal
-XX:GCPauseTimeInterval=<ergo>	最大pause time 间隔 goal。默认下，g1不会设置任何目标，允许g1 在极端情况下连续 gc。
-XX:ParallelGCThreads=<ergo>	在gc pause期间， 用于并发工作的 最多线程数。这是从运行VM的 计算机的可用线程数 得出的，方法如下：如果 可用cpu数<=8，那么就是8。否则，将大于 最终线程数的线程数 增加5/8。 在每个pause 的开始时，使用的线程数 进一步 受到最大heap size的限制：g1不会在 每个 -XX:HeapSizePerGCThread 的 java 堆容量中 使用超过1个线程。
-XX:ConcGCThreads=<ergo>	用于并发工作的 最大线程数。默认下，是 -XX:ParallelGCThreads 除以 4。
-XX:+G1UseAdaptiveIHOP -XX:InitiatingHeapOccupancyPercent=45	用于控制 初始堆 占有率的默认值，来表示 该值的自适应确定已打开。 在最初的几个 gc里，g1将使用 45%的 老年代占有率 作为标记 开始阈值。
-XX:G1HeapRegionSize=<ergo>	heap区域的 size。默认值是 基于 最大heap size 并且 计算后大约分为 2048个区域。必须是 2的乘方，有效值是 1到 32mb。
-XX:G1NewSizePercent=5 -XX:G1MaxNewSizePercent=60	年轻代的 总大小 在这2个 值之间变化，是当前使用的heap的 百分比。
-XX:G1HeapWastePercent=5	collection set 候选中 允许的 未回收空间 百分比。 g1停止 space-reclamation 阶段，如果 collection set 候选 中的 空闲空间百分比 低于这个值
-XX:G1MixedGCCountTarget=8	许多收集中，space-reclamation 阶段的 预期长度。
-XX:G1MixedGCLiveThresholdPercent=	在本次space-reclamation 阶段，不会回收那些

85	存活对象占用率超过 此百分比的 老年代区域。

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

- Comparison to Other Collectors

g1 和其他收集器的 主要区别的 概括：

parallel gc 只能作为一个整体 来 压缩和回收 老年代。g1将这项工作 逐步分布在 多个更短的gc中。这大大缩短了 pause time，但可能会 减少吞吐量

g1 并发执行 部分老年代的 space-reclamation

g1可能表现出 比上述 收集器 更高的开销，由于其 并发性 而影响吞吐量。

ZGC 的目标是 非常大的 堆，旨在 提供 更短的 pause time 但是 吞吐量成本更大。

由于它的工作方式，g1 有一些独特的机制 来提高 gc效率：

g1 可以在 任何收集期间 回收一些完全空的，大面积的 老年代。这可以避免 许多 不必要的gc，无需太大努力就可以释放大量空间。

g1可以选择 并发 尝试 对堆上的 重复字符串进行 重复数据删除。

从老年代回收 空的 大对象总是启用的。 你可以禁用：-XX:-G1EagerReclaimHumongousObjects。 string 去重默认 不启用。启用：-XX:+G1EnableStringDeduplication

- chapter 8 Garbage-First Garbage Collector Tuning

本节描述了 如何配置g1，如果它不能满足你的引用

- General Recommendations for G1

一般推荐 使用默认设置的g1，最终给它一个 不同的pause-time(MaxGCPauseMillis) 目标，并在需要的时候使用 -Xmx 设置最大heap size

g1默认值的平衡方式 和其他收集器不同。 默认配置下的g1 既不是最大化吞吐量，也不是最小化延迟，而是提供 在高吞吐量下 相对较小的，均匀的 pause。 然而，g1的 在堆上增量回收空间 和 pause time的控制 会导致 在 应用的线程 和 space-reclamation 效率 方面产生开销。

如果你想要更大吞吐量，那么放松 pause-time 目标(-XX:MaxGCPauseMillis)，或者提供一个更大的heap。

如果想要 低延迟，那么修改 pause-time。不要限制 年轻代的size 为特定值(by -Xmn，-XX:NewRatio 等)，因为 年轻代size 是 g1满足 pause-time 的主要手段。 设置年轻代size 为 某个值，会 覆盖 并实际上 禁用 pause-time 控制。
。。感觉只是说 不能为特定值，如果这个值会变就没事。那么 -Xmn 应该没有问题啊。堆大了，年轻代也大了啊。。。Xmn Xms 。。。
。。但是 没有介绍过 Xmn啊。。
。。。。。。。。。。
。。。。搜索Xmn的时候，看到了 jdk9的 调优指南。里面也是这段话。。
而且 这里有CMS (Concurrent Mark Sweep) 。。。jdk17 没有了。。
。。。。。。。

- Moving to G1 from Other Collectors

一般来说，当 从其他收集器(特别是Concurrent Mark Sweep 收集器) 迁移到 g1时， 首先要做的是 移除所有影响 gc的参数，只设置pause-time，和-Xmx， 和可选的-Xms。

其他收集器 为了获得特定的结果 的配置 中许多 要么无效，要么反而 降低 满足pause-time 和吞吐量 的概率。 一个例子是 设置年轻代大小，完全阻止了 g1 调整年轻代 来 满足 pause time。

- Improving G1 Performance

g1 被设计 来 提供良好的整体性能，而不需要指定额外参数。
当然， 在有些场景下，默认的启发式 和 配置 提供的 并不是最优的结果。
本节介绍一些针对特定指标的 调整。
根据具体情况，app级别的调整可能比 jvm的调整 更有效。比如，减少一些短命的对象。。

为了诊断，g1提供了 全面的日志记录。 一个好的开始是 使用-Xlog:gc*=debug 参数，然后提炼信息。 日志提供了 详细的信息 关于 gc活动。包括了 gc类型，每个pause的阶段的时间细分。

下面是常见的性能问题

- Observing Full Garbage Collections

一个 full heap garbage collection (Full GC) 是非常耗时的。
full gc 是由于 老年代的 占用率太高 导致的。可以通过在日志中 搜索 Pause Full (Allocation Failure) 来找到。
full gc 之前 通常会遇到 to-space exhauseted 标记 (表示 evacuation failure)。

发生full gc的原因主要是 app 分配了太多 无法 足够快地回收的对象。通常concurrent marking 无法 及时完成 以启动 space-reclamation阶段。 full gc的可能性 会由于 许多巨大对象的分配而增加。 由于g1中 这些对象的分配方式，它们可能占用 比预期更多的内存。

目标应该是 确保 concurrent marking 按时完成。 这可以通过 减低老年代的分配率，或 给 concurrent marking 更多的时间 来完成。

g1给了一些选项来更好地处理这种情况：

你可以 heap上 巨大对象 占用的 region数 通过 gc+heap=info 日志。 "Humongous regions: X->Y"中的 Y 是 巨大对象占用的 region数量。如果这个数字 和 老年代 相比起来 很高，最好的选择是 降低这个对象的数量。 你可以通过增加region size(-XX:G1HeapRegionSize) 来 达到这个目的。 当前选择的 region size 在 日志开头打印

增加heap size， 这通常会增加 marking 的时间。

增加 concurrent marking 线程 -XX:ConcGCThreads

强制g1 更早地开始 marking。g1根据先前的应用行为 自动决定 Initiating Heap Occupancy Percent(IHOP) 阈值。 如果应用的行为改变了，这些 预测可能是错的。有2个选项：修改-XX:G1ReservePercent 来增加自适应IHOP计算中 使用的 缓冲区，从而降低 启动空间回收的 目标占用率； 或 禁用 IHOP的 自适应计算，改为手工指定：-XX:-G1UseAdaptiveIHOP, -XX:InitiatingHeapOccupancyPercent。

除了 allocation failure 外的 导致 full gc的 原因通常 表明 应用 或某些 外部工具 导致了full gc。 如果是 System.gc()，并且无法修改代码，则可以通过 -XX:+ExplicitGCInvokesConcurrent 来 减轻full gc的影响 或 让jvm 无视它们 通过 -XX:+DisableExplicitGC。 外部工具可能依然强制 full gc，可以移除它们或不要发送给它们请求。

- Humongous Object Fragmentation (碎片化)

为了给 巨大对象 找一组连续的region， 可能导致 heap耗尽之前 full gc。
这种情况下，可能的选项是 通过使用-XX:G1HeapRegionSize 来增加 region size 以减少 巨大对象的数量， 或增加 heap size。 极端情况下，可能还是无法 找到连续空间。 如果full gc无法找到足够的 连续空间，JVM 会退出。。除了前面的减少巨大对象分配的数量 或 增加堆 外， 没有其他选择。
。。G1HeapRegionSize 只是 每个region 的大小啊， 不是 数量的大小。。这种 变大 有用？ 怎么觉得 减少 才有用啊， 因为 巨大对象的 最后一个region的 剩余空间是 不用的。

- Tuning for Latency

讨论 pause-time 太长的情况下，如何改进g1 行为 的提示。

- Unusual System or Real-Time Usage

对于每次 gc pause， gc+cpu=info 的日志 包含了一行，其中包含 来自OS 的信息，以及有关 pause time 在何处花费的 细节。 例子是： User=0.19s Sys=0.00s Real=0.01s.

User time 是花费在VM code 的时间， system time 是花费在OS的时间，real time是 pause 期间 经过的 时间的绝对量。
如果 system time 相对较高，那么大多数情况下是 环境造成的。

高system time的 常见原因：

VM 从OS分配 或归还内存可能导致 不必要的延迟。通过将-Xms -Xmx 设置为 相同值 来避免这种延迟， 并使用-XX:AlwaysPreTouch 来 在 VM 启动阶段 预先加载所有内存。

特别是linux中，通过 Transparent Huge Pages(THP) 功能 将小page 合并为 大page 会 暂停随机进程，而不仅仅是在 pause期间。 因为VM 分配和维护了 很多内存，所以对VM 的处理的 时间更长。 参考的你的OS的文档 来禁用 THP 功能。

写日志可能会暂停一段时间 因为 后台任务 间歇性地 占用了日志硬盘的 所有I/O带宽。考虑为你的日志 或其他一些存储 使用单独的 磁盘，例如 memory-backed file system 来避免这种情况。

另一种需要注意的情况是， real time 比 其他时间的和 大很多，这可能表明 vm 所在机器可能过载了，导致无法获得 足够的 cpu时间片。

- Reference Object Processing Takes Too Long

关于 Reference Objects 处理的耗时的信息 在 Reference Processing 阶段中 展现。
在Reference Processing阶段期间，g1 根据 Reference Object的 特定类型的要求 来更新 Reference Objects 的 所指的对象(referent)。 默认下，g1 尝试 并行 Reference Processing 的 子阶段 使用下面的 启发式：对每个 -XX:ReferencesPerThread 引用对象开启一个 单独的线程，受-XX:ParallelGCThreads 限制。 这个启发式 能被禁止 通过设置 -XX:ReferencesPerThread 为0，以默认是哟个 所有可用线程， 或完全 禁止 并发：-XX:-ParallelRefProcEnabled

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

年轻代收集的花费 和 年轻代中 需要复制的存活对象数量 成正比。
如果 Evacuate Collection Set 阶段，特别是 Object Copy 子阶段 耗时过长，则减少 -XX:G1NewSizePercent。 这减少了 年轻代的最小size，允许更短的pause。

如果app性能，特别是 收集中的 存活对象的数量 发生突然变化，则会引起另外一个与 年轻代size 有关的 问题。 这可能会导致 pause time 出现峰值。 通过降低 年轻代最大size (-XX:G1MaxNewSizePercent) 可能有用。 这个限制了 pause 阶段的 年轻代最大size 和 需要处理的 对象的个数。

- Mixed Collections Take Too Long

Mixed Collections 用于回收 老年代空间。Mixed collection的 收集区域包括 年轻代和老年代。
你可以通过 gc+ergo+cset=trace 的日志输出 来 获得 年轻代 或 老年代的 evacuation 对 pause time的 贡献。 查看 年轻代，老年代 它们各自的 预测时间(predicated yound/old region time)

如果 predicated young region time 太长，则查看上一节： Young-Only Collections Within the Young-Only Phase Take Too Long
否则，为了降低pause time 中 老年代的消耗，g1提供了3个选项：

将老年代区域回收 分散到 更多次的gc中，通过增加 -XX:G1MixedGCCountTarget

通过-XX:G1MixedGCLiveThresholdPercent 不将 它们放入 候选collection set中，来避免 收集时需要大量时间 来收集region。 在许多情况下，gc 高占有率的区域会 消耗大量时间

更早地 停止 老年代空间回收，这样 g1不会 gc 那么多 高占有率的区域，通过增加 -XX:G1HeapWastePercent

后2个选项 降低了 当前space-reclamation 阶段的 collection set 候选 个数。这可能导致g1无法在 老年代回收足够的 空间来继续运行。当然，稍后的 space-reclamation 阶段可能会回收它们。

- High Update RS and Scan RS Times

为了允许g1 evacuate 单个老年代区域，g1 跟踪 cross-region references 的位置，即从一个区域指向另一个区域的引用。
指向给定区域 的 cross-region reference 集合 称为该区域的 remembered set。
remembered set 必须被更新，当移动 region的 内容时。
对区域的 remembered set 的维护 大多是 并发的。
为了性能，g1不会立刻 更新区域的 remembered set，当app 在2个对象之间 安装新的 cross-region reference时。 remembered set 的更新请求的 延后和批量处理 能提高效率。

g1 为了 gc 需要完整的 remembered set，所以gc的 Update RS 阶段 处理所有未完成的 remembered set 更新请求。 Scan RS 阶段 搜索 object reference 在 remembered set中，移动region 内容，然后将 这些对象引用更新到 新位置。 根据应用，这2个阶段可能需要 大量时间。

通过-XX:G1HeapRegionSize 调整 heap region 的size 会影响cross-region references 的数量，也会影响 remembered set的size。
处理 region的 remembered set 可能是gc 的重要/主要 部分，因此这对 实现最大pause time目标有直接影响。
更大的region 导致 更少的 cross-region reference，所以处理它们的 时间就少了，但是，更大的区域可能意味着 每个区域需要 evacuate 更多的存活对象，增加了其他阶段的时间。

g1尝试 使用并发的 remembered set update，使得 Update RS阶段 大约占用 允许的最大 pause time的 -XX:G1RSetUpdatingPauseTimePercent 百分比。 通过降低这个值，g1 会更多地并发执行 remembered set update。

与分配大型对象的应用 相结合的 虚假(supurious) 高Update RS time 可能是由于 试图通过批处理来 减少 并发 remembered set update 的优化引起的。
如果应用 在gc前 创建了 这么一个批处理，那么gc 必须处理完这些 在Update RS 时间。使用 -XX:-ReduceInitialCardMarks 来 禁用这个行为，并可能避免这些情况

Scan RS 时间 还取决于 g1执行的 用于 为了保持 remembered set 存储size 较低 而 进行压缩的次数。
内存中 remembered set存储 越紧凑，gc时，检索 存储的值 就更耗时。
g1自动执行这种压缩，称之为 remembered set coarsening，同时根据 该region的 remembered set的当前size 来更新它。
特别是 在最高压缩级别下，检索数据可能 非常慢。
参数-XX:G1SummarizeRSetStatsPeriod 结合 gc+remset=trace 级别的日志 展示了 是否发生这种 coarsening。 如果是的，那么 在Before GC Summary 部分的 Did <X> coarsening 的X会显示为一个 高值。

可以 显著 增加 -XX:G1RSetRegionEntries 的值 来降低 coarsening 的数量。
不要再prod 使用这种详细的日志，因为收集这些数据可能需要大量时间。

- Tuning for Throughput

g1 的默认策略 尝试维持 吞吐量和 延迟 之间的平衡。然而，在某些情况下 需要更高的吞吐量。除了前几节所述的 减少整体pause time 外，还可以减少 pause的频率。
主要的思想 是 增加最大pause time 通过 -XX:MaxGCPauseMillis。
代大小的启发式 会自动 调整 年轻代的大小，这直接决定了 pause的频率。
如果这没有导致 想要的 行为(。。估计是指调整年轻代大小)，特别是在 空间回收 阶段，使用-XX:G1NewSizePercent 来增加 年轻代最小size 来 强制 g1 这样做。

某些情况下，-XX:G1MaxNewSizePercent，年轻代的最大size，可能 通过 限制 年轻代大小来限制吞吐量。这可以通过 gc+heap=info 日志的 region summary 来确定。在这种情况下，eden + survivor的 总和 接近 总区域的 -XX:G1MaxNewSizePercent 。此时考虑增加 -XX:G1MaxNewSizePercent。

另一种增加吞吐量的选择是 尝试减少并发，特别是 remembered set update 的并发 通常需要大量 cpu 资源。
增加 -XX:G1RSetUpdatingPauseTimePercent 会将 工作 从并发操作 转移到 gc pause。
在最坏的情况下，可以通过设置 -XX:-G1UseAdaptiveConcRefinement -XX:G1ConcRefinementGreenZone=2G -XX:G1ConcRefinementThreads=0 来禁用 并发 remembered set 更新。 这主要禁用此机制 并 将所有remembered set update 工作移动到下一次gc pause。

通过-XX:+UseLargePages 启用大页面的使用 也可以 提高吞吐量。参考你的OS的文档 关于如何设置大页面。

你可以最小化 heap resize 工作 by 禁用它。 设置 -Xmx -Xms 为相同的值。 而且，你可以使用 -XX:+AlwaysPreTouch 来 将 OS 的将虚拟内存 映射到 物理内存的 工作 移动到 VM 启动时。 这2种措施 都是特别可取的，为了 pause time 更加一致。

- Tuning for Heap Size

和其他收集器一样，g1 旨在 size heap 使得 gc花费的时间 低于 -XX:GCTimeRatio 选项确定的比例。 调整这个选项来让g1 满足你 应用。

- Tunable Defaults

参数和默认值	描述
-XX:+G1UseAdaptiveConcRefinement -XX:G1ConcRefinementGreenZone=<ergo> -XX:G1ConcRefinementYellowZone=<ergo>	并发 remembered set update(refinement) 使用这些 选项来 控制并发 refinement线程的工作分配。 g1为这些 选项选择符合 ergonomic 的值，使得 在gc pause 中花费-XX:G1RSetUpdatingPauseTimePercent 的时间来处理 任何剩余的工作。 谨慎修改，因为这

-XX:G1ConcRefinementRedZone=<ergo> -XX:G1ConcRefinementThreads=<ergo>	可能导致一个 非常长的pause。
-XX:+ReduceInitialCardMarks	这会把 initial object allocation 的 并发 remembered set update 批处理到一起
-XX:+ParallelRefProcEnabled -XX:ReferencesPerThread=1000	-XX:ReferencesPerThread 决定并行化程度：每N个 Reference Objects 一个线程 参与到Reference Processing的子阶段，被-XX:ParallelGCThreads限制。值0代表使用-XX:ParallelGCThreads的值。 这决定了 java.lang.Ref.* 的实例 是否应该由多根线程 并行完成。
-XX:G1RSetUpdatingPauseTimePercent=10	这个决定 g1 应该花费 总gc时间的 百分之几用于 在Update RS(。应该是remembered set)阶段 来更新 任何剩余的 remembeered set。g1通过这个设置来控制 remembered set update 的并发数量。
-XX:G1SummarizeRSetStatsPeriod=0	这是g1 生成 remembered set 摘要报告中的gc的 周期。0禁用。 生成remembered set 摘要报告 是昂贵的 操作，有 相当高的价值。只有在必要的时候使用。使用 gc+remset=trace 来打印 所有信息。
-XX:GCTimeRatio=12	花在gc上的时间 和 花在应用上的时间的 比例，在增加堆之前 可以用于gc的时间是 1/(1+GCTimeRatio)。 默认值 导致大约8% 的时间用于gc。
-XX:G1PeriodicGCInterval=0	毫秒间隔 检查g1 是否应该触发一个 周期性的gc。0禁用。
-XX:+G1PeriodicGCInvokesConcurrent	如果设置了，周期gc 触发 一个并发的marking 或继续现有的回收周期， 否则触发full gc。
-XX:G1PeriodicGCSystemLoadThreshold=0.0	主机的 getloadavg() 返回的 系统当前负载的 阈值，来确定是否需要触发 定期gc。当前系统负载大于这个值就阻止 周期gc。 0禁用这个阈值检查。

- chapter 9 The Z Garbage Collector

Z Garbage Collector (ZGC) 是 可扩展的 低延迟的 收集器。
ZGC 并发执行所有 昂贵的工作，不会让 app 线程的停止 超过 几毫秒。用于低延迟的应用。
pause time 和heap size 无关。
ZGC 支持 8mb 到 16tb 的heap

启用 -XX:+UseZGC

- Setting the Heap Size

ZGC的最重要的调优参数是 max heap size (-Xmx)。
由于zgc是并发收集器，所以必须 选择 Xmx 使得：
heap 可以容纳你的app的 实时集
有足够的空间用于 gc时 分配空间
多少空间算足够，取决于 分配率 和 app的存活集合的size。 一般来说，越多越好。但是浪费也越大，所以要找到 内存使用情况 和 gc 频率的 平衡。

- Setting Number of Concurrent GC Threads

第二个调整参数 应该是 设置 并发gc线程 数量(-XX:ConcGCThreads).
ZGC 有启发式来自动确定 这个值。启发式通常很好，但是根据app的特征， 可能需要调整。这个参数本质上 决定了 应该给GC多少CPU时间。太多会占用app时间，太少，app分配的速度可能比 gc的速度快。

- Returning Unused Memory to the Operating System

默认下，ZGC 取消提交(uncommit) 未使用的内存，而是返回这些内存给OS。 这对于关注内存占用的app和环境很有用。 可以禁用：-XX:-ZUncommit。此外，内存不会被 取消提交，所以堆大小会缩小到最小堆以下(-Xms)。这因为这 如果 -Xmx 和 -Xms 一样大，这个功能会被隐式禁用。

可以配置 未提交延迟：-XX:ZUncommitDelay=<seconds> ，默认300秒。 这个延迟表示 内存在 有资格 uncommit 之前 应该 未使用多少时间。
。。。。。。

- chapter 10 Other Considerations

影响gc 的其他事项

- Finalization and Weak, Soft, and Phantom References

一些app 通过 finalization，weak/soft/phantom reference 来和 gc 交互

这些功能可以在 java 语言层面 改变性能。一个例子是，依赖 finalization 来关闭文件描述符，这使得 外部资源 依赖于 gc的及时性。 依赖gc来管理 外部资源 总是一个坏主意。

- Explicit Garbage Collection

另一种 和gc交互 是通过 System.gc() 来 调用 full gc

这强制一个 major collection， 尽量避免。
可以通过-XX:+DisableExplicitGC 来，使得VM 忽略 System.gc()

显式GC的 最常见的用途是 远程方法调用RMI的 分布式垃圾回收DGC。使用RMI的应用 引用了其他VM中的对象。如果不偶尔调用本地堆的gc，就无法回收这些分布式应用中的垃圾，所以RMI会定期强制full gc。这个频率可以被控制：

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

这个使用 1小时一次，而不是默认的 1分钟。
如果不需要DGC活动的及时性，这个可以设置为 Long.MAX_VALUE

- Soft References

soft reference 在服务器 VM中 存活时间比 客户端中 长。

可以通过 -XX:SoftRefLRUPolicyMSPerMB=<N> 来控制 清除 频率，代表了对于堆中每mb的可用空间， 一个soft reference 不可达后 可以存活的 ms。
默认 1000ms 每兆字节， 这意味着对于堆中每兆字节的可用空间，软引用将存活 1秒。这是一个近似值，因为soft ref 只能在gc时 被收回。

- Class Metadata

java 类在Hotspot VM中有一个内部表示，称为 class metadata。

在以前的Hotspot VM 版本中，class metadata 被分配在 称为 永久代的地方
从JDK8 开始，永久代被移除，class metadata 分配到 native memory。native memory 默认无限制。-XX:MaxMetaspaceSize 加入一个上限

Hotspot显式管理 用于 metadata 的空间。从os申请空间，然后分为chunk。类加载器从它的chunk中 为metadata 分配空间(一个chunk 绑定到一个 特定的class loader)。当class loader 卸载类，它的chunk被回收 for reuse or 还给os。 元数据使用 mmap 分配空间，而不是malloc

如果 -XX:UseCompressedOops 被开启，且-XX:UseCompressedClassesPointers 被使用，那么2个逻辑上不同的 native memory区域 用于 class metadata。
-XX:UseCompressedClassPointers 使用 32bit offset 来表示 64bit 进程中的 类指针，就像 -XX:UseCompressedOops 用于 java 对象引用一样。
为这些压缩的指针(32bit offset) 分配了一个 region。region 大小可以通过-XX:CompressedClassSpaceSize 设置，默认1gb。
压缩的指针的空间 保留为 -XX:mmap 在初始化时分配的空间，并根据需要提交。
-XX:MaxMetaspaceSize 应用到 已提交的压缩类空间 和 其他类元数据空间 的和。

当对应的类被卸载时，类元数据被释放。
java类作为 垃圾被gc 卸载，并且gc 可能被诱导 卸载卸载类和释放类元数据空间。
当类元数据 提交的空间道道某个级别(高水位标记，high-water mark)时，就会引发gc。gc后，高水位标记 可能根据类元数据中释放的空间 来提高或降低。提高 高水位标记 是为了防止 不久后再次触发 gc。 通过 -XX:MetaspaceSize 来设置 高水位标记的 初始值。基于-XX:MaxMetaspaceFreeRatio, -XX:MinMetaspaceFreeRatio 来 升高或降低。 如果 可以用于

class metadata的 已提交空间 占 class metadata的总提交空间 的百分比 大于 MaxMetaspaceFreeRatio，高水位标记会降低。 如果小于 MinMetaspaceFreeRatio，会升高。

为-XX:MetaspaceSize 设置 更大的值，避免 为class metadata 引发过早的gc。
为app分配的 类元数据 取决于app，不存在 MetaspaceSize 的一般准则。
-XX:MetaspaceSize 默认值 取决于 平台，从12mb到 20mb。

关于metadata 的空间使用 包含在heap的日志中，下面是典型输出：
[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

Metaspace 开头的行，used 值是已加载类 使用的空间，capacity是当前分配的chunk的 可用于 metadata 的空间，committed 是chunk的 可用空间。reserved 是为metadata预定的空间。class space开头的行 包含 压缩的指针的 元数据的 相应值。

==========

https://docs.oracle.com/javase/10/gctuning/concurrent-mark-sweep-cms-collector.htm#JSGCT-GUID-FF8150AC-73D9-4780-91DD-148E63FA1BFF

- chapter 8 Concurrent Mark Sweep (CMS) Collector

CMS 为了那些 需要 更短gc pause 和 能在app运行时，与gc共享 CPU资源的 应用。

通常，具有相对较大的 长时间存活的数据集(即一个大的老年代)，在 >=2核 机器上运行的

应用可以从 CMS 获益。

-XX:+UseConcMarkSweepGC 启用CMS

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

- Concurrent Mark Sweep Collector Performance and Structure

和其他收集器相似，CMS是 分代的。 会发生minor 和 major collection。
CMS 尝试 降低 major collection 的 pause time 通过 和应用 并发的 特定gc线程 来 追踪可达对象。

在每个major collection cycle，在收集的开始 暂停应用所有线程 一个短暂的时间，然后在收集的中间 再次暂停。第二次暂停比 第一次长。2个暂停中 都有多线程执行收集工作。 一个或多个 gc 线程 执行 剩余的 收集任务(包括 most of 存活对象追踪 和 不可达对象的 sweep)。 minor 可以和major cycle 交错，以类似 parallel 收集器的方式完成 (特别的，minor 期间 应用线程 暂停)

- Concurrent Mode Failure

CMS 使用 1或多根 gc线程 与app 并发，尝试：在老年代 满之前 完成 gc。
一般情况下，CMS 完成它的 大部分的 tracing 和 sweeping 在 app运行时，所以 app线程只收到很短的pause。
如果CMS 无法在 老年代 满之前 完成释放不可达对象，或者 老年代的空间不够 分配，那么 app 被pause，在所有app 线程停止的情况下 完成收集。无法完成 并发收集的情况被 称为 concurrent mode failure，需要调整参数。
如果并发收集 被 显式gc(System.gc()) 或 需要为 诊断工具提供信息的 gc 中断，则报告一个concurrent mode interruption。

- Excessive GC Time and OutOfMemoryError

如果花了太多时间在gc上，则抛出 OutOfMemoryError: 如果 超过98%的总时间 用于gc，且小于2%的heap 被释放。抛出OutOfMemoryError。
这是为了防止， 由于heap太小，所以 运行很久，但是只有执行了一点点。 -XX:-UseGCOverheadLimit 来禁用这个功能

- Concurrent Mark Sweep Collector and Floating Garbage

CMS 和其他收集器一样，是 一个 追踪收集器，它至少识别heap中 所有可达对象。

Richard Jones and Rafael D. Lins in their publication Garbage Collection: Algorithms for Automated Dynamic Memory，它是一个 增量更新收集器。
因为 major 期间，app线程 和 gc线程 同步运行。 gc线程追踪到的 对象可能 随后在gc结束时 就不可达。这些不可达对象 没有被回收，被称为floating garbage。

floating garbage 的数量取决于 并发gc cycle的持续时间 和 reference 更新频率。
年轻代和老年代 是分开收集的，所以 每个 可以作为 另一个的root。
作为一个粗略的指导，尝试增加 老年代 20% 来解决 floating garbage。

- Concurrent Mark Sweep Collector Pauses

在一个 并发收集循环中， 暂停app 2次。
第一次 用于 mark 从root 直接可达的 对象(例如，app线程栈和寄存器 引用的对象，静态对象，等)，和 从堆中 其他地方(如，年轻代) 直接可达的对象 标记为 活动的
第一次暂停 被称为 initial mark pause。

第二次暂停在 并行tracing 阶段 的末尾，并 在CMS 完成跟踪对象后，查找由于app线程跟新对象中的应用而 被并发tracing 遗漏的对象。
第二个pause 被称为 remark pause

- Concurrent Mark Sweep Collector Concurrent Phases

可达对象的 并发追踪 发生在 initial mark pause 和 remark pause 之间。

在此并发追踪阶段，一个或多个并发gc 收集器线程 可能 使用 原本用于app的 CPU资源。
因此，即使没有pause，受计算力限制的 app 也会在 这个阶段 看到吞吐量下降。
remark pause 后，一个并发 sweeping 阶段 收集不可达的 对象。
在一个 收集周期完成后，CMS等待，几乎不消耗计算资源，直到下一个 major collection cycle开始。

- Starting a Concurrent Collection Cycle

使用serial 收集器，当老年代满 触发major collection，所有的app线程 在gc时 暂停。
相反，CMS 收集器中 并发收集的 开始 必须定时，以便在 老年代满之前 收集完；否则 app 会有更长的pause 因为 concurrent mode failure。

有几个方法来开始 concurrent collection

根据最近的历史，CMS收集器 维护 老年代耗尽之前的 剩余时间的估计 以及并发收集周期 需要的时间的估计。
使用这些动态估计，开启 并发收集周期，目的是在 老年代耗尽之前 完成 gc。

如果老年代的占用率 超过初始占用率， 也会触发并发收集。
初始值默认92%，可能因版本而异。
通过-XX:CMSInitiatingOccupancyFraction=<N> 来指定，0-100 之间。老年代的百分比。

- Scheduling Pauses

年轻代 和 老年代 的 pause 独立发生。

它们不会重叠，但是可能会 快速连续发生，因此一个pause，紧接着另一个pause，可能看起

来是一个较长的pause。 为了避免这个，CMS尝试将remark pause 安排在 上下2个 young pause 的 大致中间。

- Concurrent Mark Sweep Collector Measurements

下面是 -Xlog:gc 的日志

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

GC657 被 GC658，649，660 中断。
通常，一次 并发 收集周期 中会发生 多次 minor 收集。
pause initial mark 代表 并发收集周期的 开始。
以"Concurrent" 开头的行 表示 并发 阶段的 开始和结束。
Pause Remark 是最后一个阶段。
之前没有讨论的是 precleaning 阶段。precleaning代表 工作可以并发，为remark 阶段做准备。
最后阶段由 Concurrent Reset 表示，并且为下一次并发收集做准备。

initial mark pause 通常比 minor collection pause 短。
并发的阶段(concurrent mark,concurrent preclean,concurrent sweep) 比 minor gc pause 明显长。这些阶段 app 不停止。
remark pause 和 minor collection 差不多。 remark pause 收到 app特性(object修改率高会增加pause) 和 上次minor 收集 以来的 时间(年轻代有很多对象会增加pause) 影响。

==========

https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html

The concurrent collection cycle typically includes the following steps:

Stop all application threads, identify the set of objects reachable from roots, and then resume all application threads.
停止app线程，判断 从root 可达的对象集合。恢复线程运行

Concurrently trace the reachable object graph, using one or more processors, while the application threads are executing.
并发追踪可达对象图。

Concurrently retrace sections of the object graph that were modified since the tracing in the previous step, using one processor.
单线程 和 app 并发 追踪 上一步后修改的 对象图

Stop all application threads and retrace sections of the roots and object graph that may have been modified since they were last examined, and then resume all application threads.
停止app，重新追踪 上次检查后修改的对象图。 app恢复

Concurrently sweep up the unreachable objects to the free lists used for allocation, using one processor.
单线程，和app并发 ，将 无法访问的 对象 清扫到 用于分配的空闲列表中。

Concurrently resize the heap and prepare the support data structures for the next collection cycle, using one processor.
单线程，并发 resize heap，准备 下次 循环的 数据结构。

The i-cms mode uses a duty cycle to control the amount of work the CMS collector is allowed to do before voluntarily giving up the processor.

i-cms参数	描述	J6默认值
-XX:+CMSIncrementalMode		不开启
-XX:+CMSIncrementalPacing		不开启
-XX:CMSIncrementalDutyCycle=<N>		10，JDK5是50
-XX:CMSIncrementalDutyCycleMin=<N>		0，JDK5是10
-XX:CMSIncrementalSafetyFactor=<N>		10
-XX:CMSIncrementalOffset=<N>		0
-XX:CMSExpAvgFactor=<N>		25

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========

==========