JDK8 GC 调优文档，总共 2.1w 字，需要队友一起翻译，有意扫码，有错误，请反馈

序言

Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide 描述 Java HotSpot 虚拟机的垃圾搜集方法，并且能够帮助你决定最适合你的垃圾搜集方法。

1. 使用者

此文档适用于希望提升程序性能的程序开发者和系统管理员，特别是优化使用多线程技术处理大量数据，有大量事务的程序。

2. 文档使用权限

关于 Oracle's 对文档的权限，可以查看链接

http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc

3. 获取 Oracle 的帮助

已经购买 Oracle 帮助的客户可以通过登陆电子账户获取帮助，更多信息参考

http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info

http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs

4. 其他相关文档

更多的信息可以参考以下文档。

- Java 虚拟机技术

  http://docs.oracle.com/javase/8/docs/technotes/guides/vm/index.html

- 快速预览 Java SE HotSpot

  http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html

- 常见 FAQ

  http://www.oracle.com/technetwork/java/hotspotfaq-138619.html

- 怎么处理 Java Finalization's 内存问题：处理和避免 Finalization 陷阱

http://www.devx.com/Java/Article/30192

- Richard Jones and Rafael Lins, *Garbage Collection: Algorithms for Automated Dynamic Memory Management*, Wiley and Sons (1996), ISBN 0-471-94148-4

5. 术语

在此文档后面使用以下约定

约定	含义
黑体	黑体表示用户定义接口中的元素，文本或术语表中的术语
斜体	斜体表示书籍标题，强调，你提特定的值来替换的占位符
等宽字体	等宽字体表示图形框内的命令，URL，样例代码，屏幕上的文字，你输入的内容

1. 介绍

从桌面 Applet 程序到大型服务器的 web 服务，各种各样的程序都使用 Java SE。为了支持各种各样的程序需求，Java HotSpot 虚拟机实现了多种垃圾搜集器，每种搜集器都有它自己独有的特性。这样做是为了满足各种不同程序的需求。Java SE 根据运行程序的 PC 种类自动选取最合适的垃圾搜集器。然而，这个选取对所有应用程序来说可能并不是最优方案，用户，开发者，管理员可能会显式选择垃圾搜集器，调整相关相关参数来达到期望的性能目标。此文档提供完成这些工作的相关信息。首先，垃圾搜集器通用特性和基本选项参数在 serial（stop-the-world）搜集器的上下文内描述。然后介绍其他相关搜集器特性和选择这些搜集器需要考虑的因素。

垃圾搜集器（GC）是内存管理工具。它使用以下几个方法实现自动内存管理。

- 在年轻代为对象开辟空间，提升对象年纪并让它进入年老代。
- 并发标记阶段在年老代寻找存活的对象。在 Java 堆使用率超过默认阀值后，Java HotSpot 虚拟机自动触发标记阶段。具体查看 CMS 搜集器和 G1 搜集器。
- 并行复制，压缩存活对象，回收空闲空间。具体查看并行搜集器和 G1 搜集器。

什么时候应该手动选择搜集器？对某些程序来说，永远都不需要手动选择搜集器。这些程序在自动选择的垃圾搜集器下运行良好，停顿时间，搜集频率也适中。然而，对许多程序来说并不那么适用，特别是大内存，多线程，高并发程序。

Amdahl 定律（加速比受限于的串行部分）暗示着大多数工作并不能很好的并行执行；任务的某些部分只能串行，并行处理这部分任务并不会得到任何收益。这条定律同样适用于 Java 平台。通常，Oracle Java 平台的 Java SE 1.4 以下的都不支持并行垃圾搜集，所以，和并行程序相比，在多核系统上单线程的垃圾搜集器造成的影响越来越严重。

图 1-1,"comparing Percentage of Time Spend in Garbage Collection" 描述一个理想系统（完全可伸缩，除了垃圾搜集器）中垃圾搜集器的时间占比。红色线条表明单核系统中的垃圾搜集器占用了 1%的时间。但是在 32 核的系统中，系统吞吐量损失超过 20%。在紫红色线条中，垃圾搜集器花费 10%的时间（不考虑单核系统中垃圾回收的时间），当伸缩到 32

核时，损失的的吞吐量超过 75%。

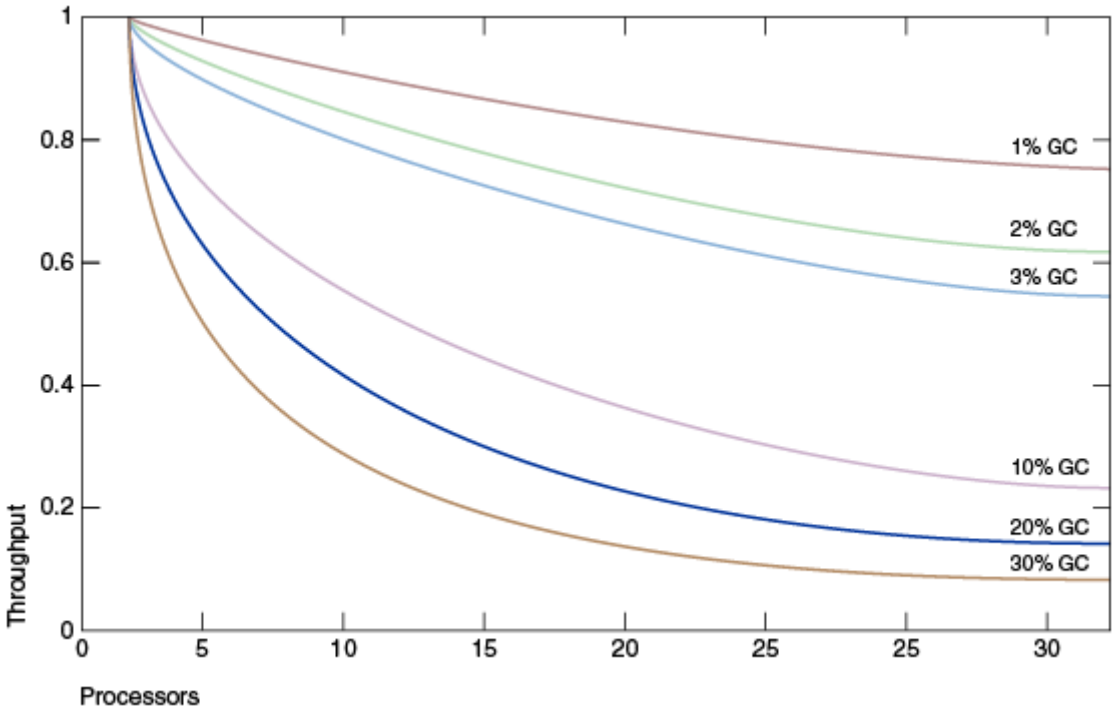


图 1.1-comparing Percentage of Time Spend in Garbage Collection

上图表明在小型系统往大型系统发展时，微不足道的小问题会成为整个系统的瓶颈。然而，小小的提升可以获取非常大的收益。对于一个足够大的系统，选择一个合适的垃圾搜集器，并且调整它是必要的。

串行搜集器通常在小型程序中使用（最高大约 100MB 内存）。其他搜集器有其他的花费和复杂性，这是为了实现独有功能的代价。如果一个应用程序对搜集器没有特殊的要求，可使用串行搜集器。对于多核处理器，大内存的环境使用串行搜集器不适用。当程序运行在这种类似服务器类型机器上时，会自动选择并行搜集器，具体查看人类工程学。

此文档使用 Solaris 操作系统的 Java SE 8 开发环境做为参考。然而，概念和建议对所有支持的平台都通用，包括 Linux，Microsoft Windows，Solaris，OS X。命令行参数在所有平台都有，只是默认值可能会不同。

2. 人类工程学

*人类工程学*是自动选择 JVM 类型和调整垃圾搜集器的过程，比如调整基本行为和提升程序性能。JVM 根据平台的不同自动选择垃圾搜集器，堆大小，运行时编译器，这些默认的选择让用户在调整 GC 时使用更少的参数。另外，基于行为动态调整堆的大小满足用户指定的应用程序行为。

此章节描述默认选择和基于行为调整。在使用后续章节中描述的搜集器前请先使用这些默认值。

1. 垃圾搜集器，堆，运行时编译器的默认值

服务器类型机器的定义如下：

- 2 核或者更多的物理核
- 2G 或者更多的物理内存

服务器类型机器，使用如下默认参数：

- 吞吐量搜集器
- 堆初始大小是物理内存的 1/64，最大 1GB
- 堆最大大小是物理内存的 1/4，最大时 1GB
- Server runtime compiler

对于 64 位系统的初始堆大小和最大堆大小，查看并行搜集器内的默认堆大小一节。

除了 Win32 平台不是 Server 类型的机器，其他平台都是 Server 类型的机器。

表 Table 2-1 Default Runtime Compiler 列出了各个平台相关的运行时编译器。

Table 2-1 Default Runtime Compiler

Platform	**Operating System**	**Default**[Foot1]	**Default if Server-Class**[Footref1]
i586	Linux	Client	Server
i586	Windows	Client	Client[Foot2]

Platform	Operating System	Default[Foot1]	Default if Server-Class[Footref1]
SPARC (64-bit)	Solaris	Server	Server[Foot3]
AMD (64-bit)	Linux	Server	Server[Footref3]
AMD (64-bit)	Windows	Server	Server[Footref3]

[Footnote1]Client 表示使用 client 类型的运行时编译器。Server 表示使用 Server 类型的运行时编译器。

[Footnote2] 策略说明即使是在 Server 类型的机器也会使用 client 类型的编译器。使用这个选择是因为历史原因，以前客户端程序（比如交互程序）通常运行在这些操作系统和平台的组合上。

[Footnote3] 仅支持 server 类型的运行时编译器。

2. 基于行为调整

对于 parallel 搜集器，Java SE 提供基于程序行为调整搜集器的参数：最大停顿时间目标和程序吞吐量目标；更多信息查看并行搜集器。（这俩个目标在其他搜集器中不可用）注意这俩个目标不是总能达到。应用程序需要足够的堆内存保存至少一半的存活数据。另外，最小堆大小可能影响满足这些目标。

最大停顿时间

停顿时间是指垃圾搜集器让应用程序停止的时间。最大停顿时间目标限制这些停顿的最长时间。垃圾搜集器维护平均停顿时间和方差。平均停顿时间从程序开始执行时进行加权计算，最近的停顿时间权值更重。如果平均时间加方差时间大于最大停顿时间，搜集器就会认定没有满足最大停顿时间目标。

使用命令行参数-XX:MaxGCPauseMillis=<nnn>指定最大停顿时间。这表示搜集停顿时间应该小于<nnn>毫秒。搜集器会调整堆大小和其他相关参数来保持停顿时间小于<nnn>毫秒。默认无最大停顿时间。这些调整可能导致搜集行为频繁发生，减少程序的吞吐量。垃圾搜集器先满足停顿时间，再满足吞吐量目标。在某些情况下，无法达到指定的停顿时间目标。

吞吐量目标

吞吐量目标基于垃圾搜集器时间和非垃圾搜集时间（程序时间）测量，这个目标值使用命令行参数-XX:GCTimeRatio=<nnn>指定。使用在垃圾搜集器上的比例是 1/（1+<nnn>），比如，-XX:GCTimeRatio=19 表示 1/20 时间使用在垃圾搜集上。

垃圾搜集时间指的是花费在年轻代和年老代的时间之和。如果不能满足吞吐量目标，增长各个代的大小让程序运行时间变长。

Footprint Goal-占用内存大小

如果吞吐量和最大停顿时间目标都达到，垃圾搜集器会尝试减少堆大小直到某个目标不满足（通常是吞吐量目标），然后开始解决无法满足的目标。

3. 调整策略

不要设置最大堆大小，除非你知道需要的最大堆大小大于默认的最大堆大小。选择一个足以满足你程序要求的吞吐量目标。

为了满足吞吐量目标，堆大小会自动变大，减少。程序的行为也会导致堆大小增长和收缩。比如，如果程序频繁申请空间，堆的大小就会增长，以此保持吞吐量不变。

如果堆增长到最大大小，吞吐量目标仍然不满足，那就说明对于当前吞吐量目标最大堆大小太小。设置尽可能大的物理内存，并且不会引起程序的内存页交换，然后再次运行程序。如果吞吐量目标还未达到，那就说明对于当前的环境，吞吐量目标太高了。

如果吞吐量目标能够达到，但是停顿时间太长，设置最大停顿时间。设置了最大停顿时间意味着可能吞吐量目标不能满足，所以，选择一个合适的值就好。

当垃圾搜集器试图满足矛盾的目标时，堆大小通常会震荡。即使是稳定状态的程序也会这样。吞吐量目标（需要较大堆）通常与最大停顿时间目标，占用空间目标（需要小堆）矛盾。

3. 代

使用 Java SE 平台的好处就是开发人员不用管理内存和垃圾搜集。然而，当垃圾搜集变成瓶颈后，知道与垃圾搜集器实现相关的概念就会变成非常有用。垃圾搜集器假设程序使用对象的方式，这些假设反应在可调整的参数中，同时又不损失垃圾搜集器概念上的抽象性。

如果一个对象不再可达，就意味着是垃圾。最简单的垃圾搜集算法直接遍历所有可达的对象，剩余的对象就是垃圾。这个方法耗时和存活对象数目成正比，对于一个有大量存活对象的程序来说，这样做是不合适的。

虚拟机使用分代策略在不同的代使用不同的搜集算法。缺乏经验的垃圾搜集器测试堆内每个对象是否存活，分代垃圾搜集器利用几个已知的事实经验来搜集垃圾。最重要的经验就是，大多数对象仅存活一小段时间。

图 3-1 中蓝色区域是一个典型的对象存活时间分布图，x 轴表示对象存活的时间，y 表示存活时间中字节总数，左边的尖峰表示已经死亡的对象，比如对于循环体内的对象，对象仅在一次循环中存活。

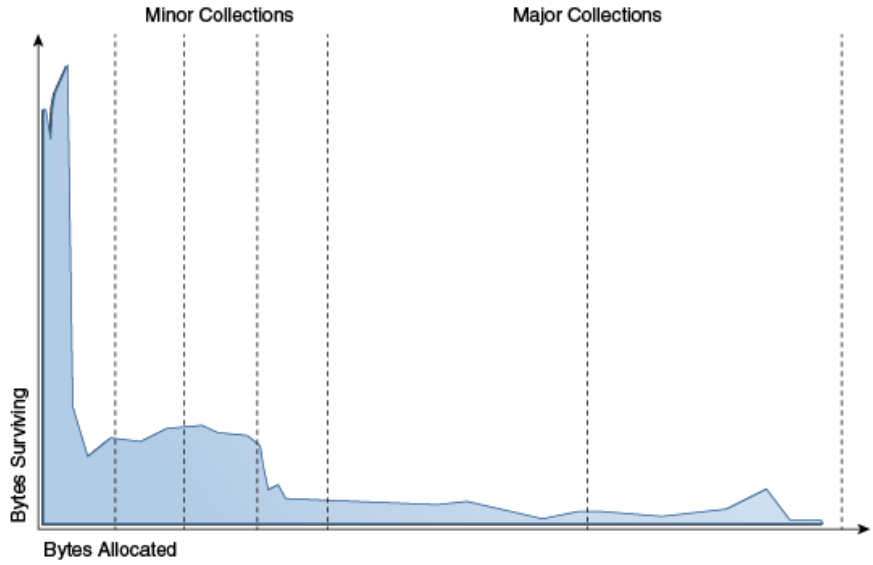


图 3-1-对象生存时间分布

某些对象会存活许久，它们分布在图的右边。比如，那些程序初始化就存在，在程序退出时才会死亡的对象。在这俩者之间的对象是一些中间计算的对象（图的中间部分），图中初始峰值的右侧。某些程序的对象存活时间可能和上图不同，但是，令人惊奇的是大部分程序都如上图。通过大多数对象在年轻时就会死亡这一事实可以高效的进行垃圾搜集。

基于上述优化方案，内存基于代管理（不同年龄对象存在不同的对象池）。当代满了的时候进行垃圾搜集。绝大多数对象分配在年轻代，然后死在年轻代。当年轻代填满后，触发一次 minor gc 搜集年轻代；其他代不进行回收。Minor gc 可以更加优化。这个搜集方案的花费和存活对象的数量成正比；搜集充满死亡对象的年轻代非常快。更明确来说，在每次 minor gc 后某些幸存的对象会移动到年老代。最后，年老代也会被填满，也必须被搜集，

这就会触发 major gc，这个时候整个堆都会被搜集。Major GC 通常比 minor GC 花费更多的时间，因为它涉及对象的数量比 minor gc 多。

正如人类工程学章节描述的那样，虚拟机自动选择垃圾搜集器，在各种类型的程序中提供良好的性能。Serial 搜集器用于小数据集的程序，它也是许多小应用的默认搜集器。Parallel 和吞吐量搜集器使用在程序有中等到大数据集的情况下。自动选择的堆大小和自适应策略能够给服务器程序提供良好的性能。在大多数情况，它们都能运行的很好，但并不是在所有情况下都运行的很好，这就是本文档的核心内容：

Note：

如果垃圾搜集已经成为瓶颈，你必须手动调整堆大小，各位代大小。检查搜集器的输出，测试垃圾搜集器参数对性能的影响。

图 3-2 展示了默认的代排列顺序（除了 G1 和并行搜索器）

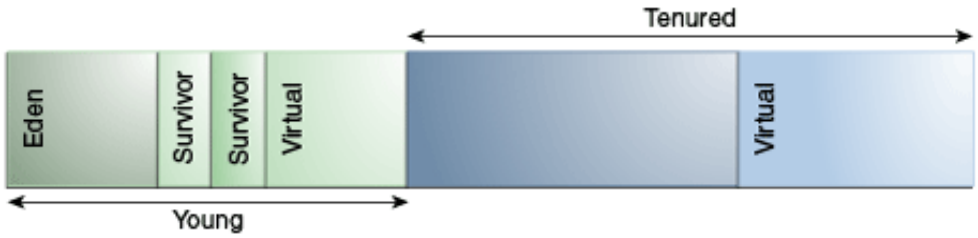


图 3-2-代的默认排列（除了并行和 G1 搜集器）

在初始时，最大地址空间 virtually，不会还给物理内存，除非需要被还回。对象可使用的地址空间分为年轻代和年老代。

年轻代由 eden 和俩个 survivor 组成。大多数对象在 eden 中申请空间。Survivor 中一个一直保持空闲，并且用于保存从 eden 来的存活对象；另一块 survivor 作为下一次复制的目的地。在 survivor 之间复制对象时，如果对象已经足够老，会被放到年老代中。

1. 性能考虑

使用以下俩个关键点测量垃圾搜集器性能：

吞吐量：花费在程序上的时间百分之。吞吐量时间包括申请内存所用的时间（通常不需要调整分配速度）。

停顿时间：发生搜集器导致应用程序停止的时间。

每个用户对垃圾搜集器都有不同的需求。比如，某些 web 服务器程序认为吞吐量才是正确的策略指标，因为垃圾搜集造成停顿时间可能会被网络延时覆盖。然而，在交互程序中，即使是很小的停顿也会让用户体验不好。

某些用户对其他因素比较敏感。*Footprint*（垃圾对象占用的量）表示程序的工作集，使用页和缓存行为单位。在物理内存受限和有许多进程的系统内，footprint 是一个影响伸缩的

因素。*Promptness* 是对象变成垃圾到被回收所间隔的时间。在分布式系统中这是一个重要的因素，包括远程方法调用（RMI）。

通常来说，设置一个特定代的大小需要权衡以下因素。比如，非常大的年轻代会造成高的吞吐量，但是也会造成大的 footprint，promptness，停顿时间。小的年轻代会降低吞吐量。一个代的大小不会影响另外一个代的搜集器频率和停顿时间。

不存在一个正确选择代的方法。最优的方案根据用户需求的不同而不同。因此，虚拟机自动选择的方案通常不是最优，可以通过命令行参数来覆盖默认参数。更多信息参考章节代的大小。

2. 测量

吞吐量和 footprint 通常是最好的测量指标。比如，web 服务器的吞吐量通常可以用客户端负载生成器来测试，footprint 相关的信息，在 Solaris 操作系统中可以使用 pmap 来生成。同样，通过检查虚拟机的输出信息，很容易算出由于垃圾搜集导致的停顿时间。

命令行参数-verbose:gc 可以让虚拟机打印出每次搜集时堆和垃圾搜集器信息。比如，以下的输出来自一个大型的服务器程序：

```
[GC 325407K->83000K(776768K), 0.2300771 secs]
[GC 325816K->83372K(776768K), 0.2454258 secs]
[Full GC 267628K->83769K(776768K), 1.8479984 secs]
```

输出信息包括俩次 minor 和一次 major 搜集的信息。箭头前后的数字表示搜集前后存活对象的大小。在 minor 搜集后，某些不再存活对象可能没有被回收，这些对象包含在年老代中或者被年老代引用。

括号内的下一个数字（比如 `776768K`）表示提交的堆大小：Java 对象占用的所有空间，此时不需要从操作系统申请更多的内存。注意，这个数字仅包含一块 survivor 空间。在垃圾搜集时，仅有一块空间用于保存对象。

行尾的时间（比如，`0.2300771 secs`）表示执行搜集花费的时间，这个输出大概是四分之一秒（0.25 秒）。

Major 搜集输出信息的含义和上面一样。

Note：

-verbose:gc 输出的格式在将来版本可能会被修改。

-XX:+PrintGCDetails 打出 GC 的其他信息，下面是一个 Serial 搜集器输出的样例信息。

```
[GC [DefNew: 64575K->959K(64576K), 0.0457646 secs]
196016K->133633K(261184K), 0.0459067 secs]
```

上面信息表示年轻代回收率大约 `98%`，DefNew: 64575K->959K(64576K)，花费时间 0.0457646 secs 。

整个堆的使用率减少了 51%（196016K->133633K(261184K)），总花费时间 0.0459067 secs，搜集器的其他工作占用了一些时间。

Note：

-XX:+PrintGCDetails 输出格式在将来发行版中可能会被改变

-XX:+PrintGCTimeStamps 选项会在输出格式中加上每次搜集开始的时间，这个信息可以用来检查搜集的频率：

```
111.042: [GC 111.042: [DefNew: 8128K->8128K(8128K), 0.0000505
secs]111.042: [Tenured: 18154K->2311K(24576K), 0.1290354 secs]
26282K->2311K(32704K), 0.1293306 secs]
```

在程序开始执行 111 秒后发生了一次垃圾搜集行为，minor 开始于同样的时间，另外，信息也描述了一次年老代的搜集信息，年老代使用率大概减少到 10%(18154K->2311K(24576K))，花费 0.1290354 secs （大概 130 毫秒）

4. 代的大小

有一些参数影响代的大小，图 4-1“Heap Parameters”说明堆中虚拟空间和已提交空间的区别。在虚拟机初始化时，一次开辟整个堆空间（-Xmx 选项）。如果-Xms 参数比-Xmx 小，并不是所有空间都提交，这些未提交的空间在图中的标记为 virtual。堆的不同部分（年轻代和年老代）都可以生长。

有些参数是堆中一部分和另外一部的比率。比如，NewRatio 表示年老代和年轻代的相对大小。

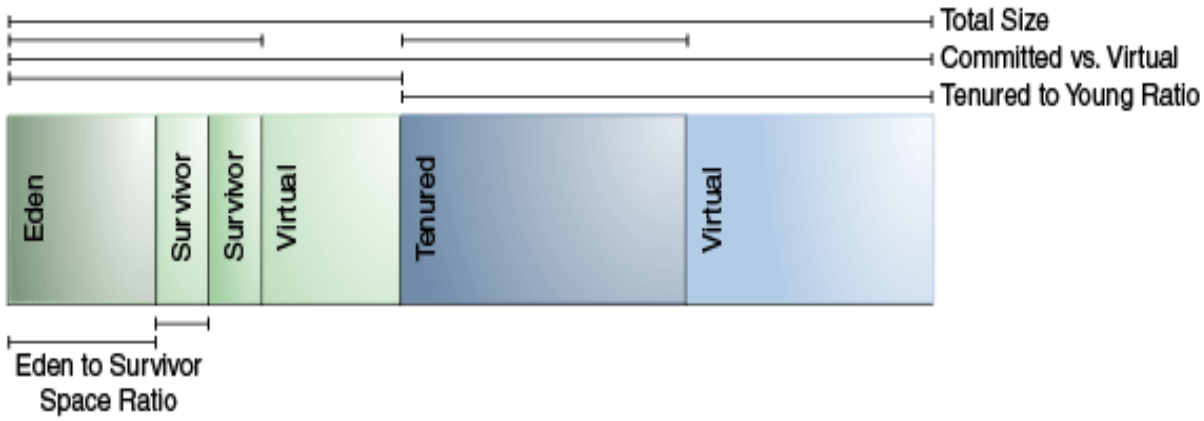


图 4-1“Heap Parameters”

1. 堆的总大小

以下讨论堆增长和收缩，默认堆大小不适用于 parallel 搜集器（更多查看 Parallel 搜集器和代的大小章节）。但是，控制堆的总大小和各个代大小的参数适用于 Parallel 搜集器。

影响搜集器性能最重要的因素是总的可用空间大小，因为在代满了以后就会触发搜集行为，吞吐量与可用空间成反比（可用空间越大，垃圾搜集越慢，停顿时间越长，程序吞吐量越低）。

默认情况下，在每次搜集后，虚拟机增长或收缩堆来保证空闲空间的比例在指定的范围内。这个范围通过使用参数-XX:MinHeapFreeRatio=<minimum> 和 -XX:MaxHeapFreeRatio=<maximum>来指定，总大小的范围由-Xms<min>和

-Xmx<max>约束。64 位的 Solaris 操作系统的默认参数如下表。

Parameter	**Default Value**
MinHeapFreeRatio	40
MaxHeapFreeRatio	70

Parameter	Default Value
-Xms	6656k
-Xmx	calculated

根据以上的参数，如果某个代中空闲空间百分比小于 40%，对应的代就会扩张到满足 40%的空闲空间，上限是代允许的最大大小。同样，如果空闲空间超过 70%，代会被收缩到只有 70%空闲，最小是代的最小大小。

注意表中的另外一个参数，默认最大堆大小是 JVM 计算得出。Parallel 搜集器和 server 类型 JVM 的计算现在适用于所有搜集器。32 位平台和 64 平台的计算上限不同，更多参考章节 Parallel 搜集器的默认堆大小。对于客户端类型 JVM，计算方式相似，结果就是客户端类型 JVM 最大堆大小比服务器类型 JVM 的小。

以下是设置服务器程序堆大小的一般规则：

- 除非你的问题是停顿时间，要不然尽可能给虚拟机内存，默认堆大小通常来说太小。
- -Xms 和-Xmx 设置成一样的值提升可预测性，这样会移除虚拟机对堆大小的决策。然而，如果你设置了一个不合适的值，虚拟机无法补救。
- 通常，随着处理器的增加需要增加内存，因为可以并行申请空间。

2. 年轻代

在决定可用内存后，第二个影响垃圾回收器性能的因素是年轻代的比例。越大的年轻代，minor gc 次数越少。然而，对于有限的堆大小，大的年轻代意味者小的年老代，这样就会提升 major gc 的频率。最优的选择取决于对象存活时间的分布。

默认情况下，年轻代的大小由参数 NewRatio 控制。比如，设置-XX:NewRatio=3 意味着年轻代和年老代的比例是 1:3。换句话来说，eden 和 survivor 占用整个堆的四分之一。

参数 NewSize 和 MaxNewSize 表示年轻代的下限和上限，如果这俩个参数的值一样，那么年轻代的大小就被固定，就像设置-Xms 和-Xmx 的值一样来固定整个堆大小。

3. Survivor 空间大小

你可以使用 SurvivorRatio 参数指定 Survivor 空间的大小，但是这个对性能的影响通常不大。比如，-XX:SurvivorRation=6 表示 eden 和一个 survivor 的比例是 6：1.也就是说每个 survivor 空间是 eden 的六分之一，也就是整个年轻代的八分之一（不是七分之一，因为存在俩个 survivor 空间）。

eden:one survivor= 6:1，eden: survivor = 6:2，意思就是 eden 占年轻代的八分之六，survivor 占八分之二。

如果 survivor 空间太小，对象复制时，如果空间溢出，对象直接复制到年老代。如果 survivor 太大，纯粹浪费内存。在每次垃圾回收时，虚拟机会选择一个阀值表示对象可以复制到年老代。这个阀值是为了保持 survivor 半满。使用参数-XX:+PrintTenuringDistribution（不是所有垃圾搜集器都有用）打印出这个阀值。同样也可以用于观察对象的分布。

64 位-Solaris 平台下的默认值

Parameter	**Server JVM Default Value**
NewRatio	2
NewSize	1310M
MaxNewSize	not limited
SurvivorRatio	8

年轻代的大小通过最大堆大小和 NewRatio 参数来决定。MaxNewSize 默认值是 not limited 表示计算后的值不受限制，除非你指定了 MaxNewSize 参数值。

以下是服务器类型程序设置指南：

- 首先设置把堆的最大值尽可能的大。然后根据年轻代的大小来设置性能指标，以此找到最好的设定。
    - 注意最大堆大小应该总是小于机器的物理内存，避免内存页交换。
- 如果最大堆大小已经确定，提升年轻代大小，减少年老代大小。保证年老代足够大，永远可以保存存活的对象，并且增加一些缓冲（10%-20%，或者更多）。
- 根据先前对年老代的约束：
    - 给年轻代足够的内存。
    - 处理器数目的增加，相应增加年轻代大小，因为可以并行申请内存。

5. 可用的搜集器

Serial 搜集器前面已经讨论过，Java HotSpot VM 包含三种不同类型的搜集器，每种搜集器都有它自己都有的性能特点。

- Serial 搜集器使用单线程执行所有垃圾搜集工作，这是一个相对高效的方法，因为没有和其他线程交互的花费。在单核处理器机器中这是最合适的方法，虽然也能在多核，小数据集（最大大约 100MB）的程序中使用它。在某些硬件和操作系统中 Serial 搜集器是默认搜集器，也可以通过显示使用-XX:+UserSerialGC 选定它。

- Parallel 搜集器（也叫做吞吐量搜集器）并行执行 minor GC，这样能大幅度减少 GC 花费的时间。这也就意味着它适合大数据集，多核或者多线程的环境。某些硬件和操作系统环境下自动选择 Parallel 搜集器，也可以手动使用-XX:+UseParallelGC 参数来指定。

  - Parallel 搜集器在 major GC 时可以并行压缩。没有并行压缩时使用单线程执行 major GC，这样严重影响伸缩性，指定-XX:+UseparallelGC 参数默认开启并行压缩，使用选项-XX:-UseParallelOldGC 关闭这个功能。

- 大多数并发搜集器并发执行垃圾搜集器任务，以此减少停顿时间。它适用于中型-大型数据集的程序并且程序的响应时间比吞吐量重要，因为最小化停顿时间会减少程序的性能。Java HotSpot VM 提供俩种并发搜集器；参考并发搜集器。使用选项-XX:+UseConcMarkSweepGC 使用 CMS 搜集器，使用选项-XX:+UseG1GC 使用 G1 搜集器。

1. 选择搜集器

如果没有严格停顿时间约束，首先让 VM 自动选择搜集器。如果有需要，再调整堆大小来提升性能。如果性能还是不满意，根据以下指南来选择搜集器。

- 如果程序的数据集很小（最大大约 100MB），那么你应该用 Serial 搜集器，使用 -XX:+UseSerialGC
- 如果程序运行在单线程环境，没有停顿时间要求，让 VM 自动选择或者使用 Serial 搜集器。
- 如果程序性能是第一要素，没有停顿时间要求，停顿 1s 或者更长时间也能接受，让 VM 自动选择或者使用 ParallelGC，命令行参数-XX:+UseParallelGC
- 如果响应时间比吞吐量重要，停顿时间必须短，使用并发 GC，-XX:+UseConcMarkSweepGC 或者-XX:+UseG1GC。

此指南仅包含选择搜集器的第一步，因为性能还和堆大小，程序维护的存活数据集，可用处理器数目的速度相关。停顿时间通常也与上述因素相关，所以前面提到的阀值 1s 也只是一个大概值：在许多大量数据集和硬件的组合下，parallel 搜集器的停顿时间可能远远大于 1s，同样，在某些组合下，并发搜集器也有可能无法保持停顿时间短于 1s。

如果建议选取的搜集器还是不满足要求，首先应该调整堆和各个代的大小，如果性能还是不满足，然后尝试换个搜集器：使用并发搜集器减少停顿时间，使用并行搜集器提升吞吐量。

6. 并行搜集器

Parallel 搜集器（也叫做吞吐量搜集器）和 Serial 搜集器一样，是一个分代搜集器；最大的区别就是使用多线程加快垃圾搜集速度。使用参数-XX:+UseParallelGC 来使用 Parallel 搜集器。默认情况下，minor gc 和 major gc 都并行处理，进一步减少垃圾搜集的开销。

如果机器有 N 个硬件线程，并且 N 大于 8，Parallel 搜集器使用固定比率的线程进行垃圾搜集器。比率大概是 5/8。如果 N 小于 8，使用 N 个线程。在特定的平台上，大概下降到 5/16。也可以使用命令行参数调整垃圾搜集线程数。在单核机器上，Parallel 搜集器的表现和 Serial 搜集器不一样，因为单核处理器使用多线程技术会有额外的花销（比如，同步）。然而，对于运行在中大型堆，双核处理器的程序来说，使用 Parallel 搜集器通常优于 Serial 搜集器，如果可用的处理器超出双核，性能超出更多。

可以使用参数-XX:ParallelGCThreads=<N>控制并行搜集器的线程数目。如果已经手动指定好堆的大小，Parallel 搜集器获得良好性能所需的堆大小和 Serial 搜集器一致。但是，使用 Parallel 搜集器会缩短搜集时间。因为在 minor gc 中使用多个垃圾回收线程，因此，在年轻代对象提升到年老代时会出现一些碎片。minor gc 中每个垃圾回收线程保留年老代中的一部分空间用于提升对象，从年老代可用空间中划分出这些“提升 buffer”时会导致内存碎片。减少垃圾垃圾回收线程数目并且提升年老代大小可能会减少这个碎片。

1. 代

就像前面讨论的那样，Parallel gc 的代分布和其他的搜集器不同，如下图 6-1 所示。

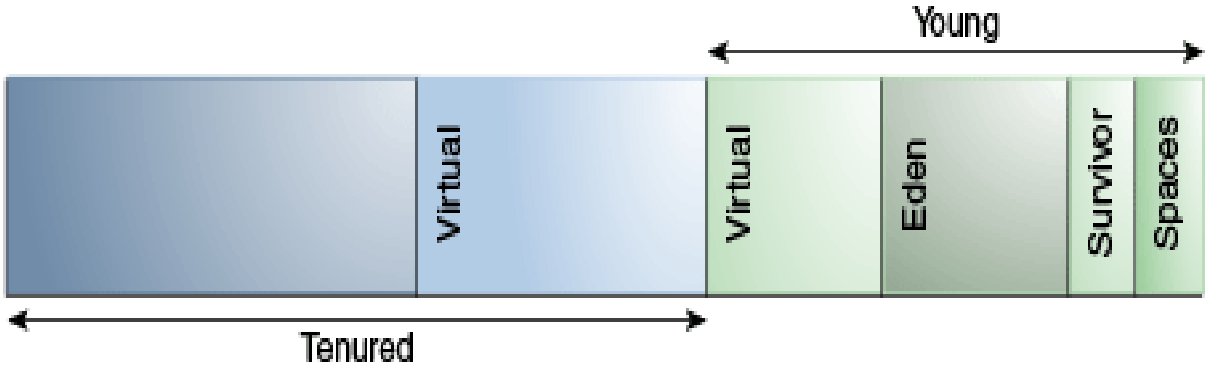


图 6-1

2. Parallel 搜集器自动选择项

在 server 类型的机器上，默认选择 Parallel 搜集器。同样，Parallel 搜集器会根据你指定的行为自动调整，并且不需要你指定类似代大小这种低级别的调整信息。你可以指定最大停顿时间，吞吐量，footprint（堆大小）。

- **垃圾搜集器最大停顿时间：**最大停顿时间目标使用选项-XX：MaxGCPauseMillis=<N>。

  这个参数用于提示搜集器停顿时间应该小于等于<N>;默认情况下，不存在最大停顿时间目标。如果指定了最大停顿时间，自动调整堆大小和其他相关参数来满足停顿时间小于等于指定的值。这些调整可能会导致程序吞吐量降低，而且也没达到目标停顿时间。

- **吞吐量：**吞吐量根据垃圾搜集器花费时间和程序所有时间来衡量。使用参数-XX:GCTimeRatio=<N>。垃圾搜集花费的时间为 1/(1+<N>)。

  比如，-XX:GCTimeRatio=19，表示百分之 5 的时间用于 GC，默认值是 99。

- **Footprint：**使用-Xmx<N>指定堆的最大值，但是，搜集器还有一个隐含的目标，只要满足其他条件，就会把堆最小化。

3. 目标优先级

目标优先级如下：

1. 最大停顿时间

2. 吞吐量目标

3. 最小化 footprint 目标

最大停顿时间在先，在满足这个目标后，然后满足吞吐量目标，再满足 footprint 目标。

4. 调整代的大小

搜集器保存的静态统计信息（比如平均停顿时间）在每次搜集后都会被更新。GC 会测试必要的目标是否满足，然后对代大小进行调整。但是有一个特例，忽略显式 GC 的信息（System.gc()，不进行统计和调整）。

使用代的固定百分比增长，收缩代的大小。这样代大小直接增长，伸缩到期望的大小。增长和收缩的百分比值不同。默认情况下，增长百分比是 20%，收缩的百分比是 5%，使用命令行选项-XX:YoungGenerationSizeIncrement=<Y>设置年轻代增长百分比，-XX:TenuredGenerationSizeIncrement=<T>设置老年代增长百分比。调整收缩的百分比使用参数-XX:AdaptiveSizeDecrementScaleFactor=<D>。如果增长比例是 X，那么收缩百分比就是百分之 X / D。

在启动时，如果搜集器决定扩张代的大小，会在增量上再附加一个百分比。这个附加参数会逐渐减少，没有持久影响。这个附加百分比用于提升启动时的性能。对于收缩没有附加百分比。

如果停顿时间目标不能满足，仅收缩当前搜集的代，如果年轻代和年老代停顿时间都不满足，那么较大的代首先被收缩。

如果吞吐量目标不能满足，增长所有代的大小。增长的大小与占用的垃圾搜集时间成比例。比如，如果搜集年轻代花费的时间占用总时间的百分之 25，如果年轻代的全增长率是百分之 20，那么最终年轻代的增长率是百分之 5。

5. 默认堆大小

除非手动指定堆大小，要不然就根据机器的内存大小来计算。

6. Client 类型 JVM 初始堆大小和最大堆大小

默认最大堆大小是物理内存的一半，最大到 192MB 的二分之一，要不然就是物理内存的四分之一，最大到 1GB 的四分之一。

比如，如果你的机器有 128MB 内存，那么最大内存就是 64MB。如果你的机器内存大于或者等于 1GB 内存，那么最大内存就是 256MB。

JVM 并不会真的使用这么大的内存，除非程序真的创建了许多对象。在 JVM 初始化的时候开辟一个最小内存空间。最小是 8MB，要不然就是内存的 64 分之一，最大到 1GB 的 64 分之一。

年轻代开辟空间的最大值是堆的总大小的 3 分之一。

7. Server 类型 JVM 初始堆大小和最大堆大小

Server 类型 JVM 的初始堆大小和最大堆大小的设定和 Client 类型 JVM 类似。只是值比较高。在 32 位 JVM 中，如果物理内存是 4GB 或者更大，默认最大堆大小能够到 1GB。在 64 位 JVM 内，如果物理内存是 128GB 或者更大，默认最大堆大小能够到 32G。你可以设置更高或者更低的初始值和最大值，具体参考下一章节。

8. 指定堆的初始值和最大值

你可以使用-Xms 指定初始值，使用-Xmx 指定最大值。也可以把这俩个选项的值设置成一样。如果没有设置成一样，JVM 会从初始化堆开始，逐步增长 Java 堆，直到在性能和堆使用率之间找到平衡。

其他一些参数同样会影响默认值。为了确定你使用的默认值，使用-XX:+PrintFlagsFinal 选项，检查输出中 MaxHeapSize 信息。比如，在 Linux 和 Solaris 中可以这样。

Java –XX:+PrintFlagFinal <GC options> -version | grep MaxHeapSize

9. GC 花费太多的时间导致 OutOfMemoryError

在 Parallel 搜集器花费太多时间时会抛出 OutOfMemoryError 错误。如果在 GC 上花了 98%的时间但是回收的空间小于 2%，那么就会抛出 OutOfMemoryError 错误。这个功能用于阻止长时间运行，但是使用内存太小的 JVM 启动。可以使用选项-XX：-UseGCOverheadLimit 参数关闭这个功能。

10. 测量

Parallel 搜集器打印出的信息和 Serial gc 打印的信息基本相同。

7. 并发搜集器

JDK8 中的 Hotspot 虚拟机有俩种并发搜集器：

- CMS 搜集器（Concurrent Mark Sweep Collector）：CMS 搜集器用于需要较短停顿时间的程序，并且程序能够尽量分享处理器资源给搜集器。
- G1 搜集器（Garbage-First Garbage Collector）：G1 使用在多核和大内存的机器上。它在满足高吞吐量的前提下尽可能让停顿时间短。

1. 并发的开销

并发搜集器使用处理器资源（如果搜集器不用，这部分处理器资源给应用程序用）换取较短的 major gc 停顿时间，这是显而易见的开销。在有 N 个处理器系统中，并发搜集阶段使用 K/N 个处理器，1<=K<=ceiling{N/4}（注意，精度范围的选择往后可能会变）。为了在并发阶段使用处理器资源，另外一个开销就是并发开销。因此，搜集器导致的停顿时间通常比较短，程序的吞吐量也会有一些的降低。

多核处理器系统中，垃圾搜集器进入并发阶段时，其他可用的核用于服务应用程序，因此，并发搜集线程不用停止应用程序。而且会让停顿时间变短，同样，这样程序能使用的处理器资源也会变少，甚至有时候会被停止。随着处理器核数 N 的提升，垃圾搜集导致的影响也会相应变小，从并发搜集得到的收益也会变大。CMS 搜集器的并发模型缺陷将会讨论这种潜在的限制。

因为并发阶段至少使用一个处理器核，因此，单核环境中使用并发搜集器不会有任何好处。然而，CMS 有一个模型可以在单核或双核环境中提供低停顿的垃圾回收，更多请查看 CMS 搜集器内的增量模型。这个功能在 Java SE8 中已经被废弃并且在往后大版本中移除。

2. 其他参考

- G1 搜集器

  http://www.oracle.com/technetwork/java/javase/tech/g1-intro-jsp-135488.html

- 调整 G1 搜集器

  http://www.oracle.com/technetwork/articles/java/g1gc-1984535.html

8. CMS 搜集器

需要低停顿时间，并且能够分享处理器资源给搜集器的程序可以使用 CMS 搜集器。这类程序通常有相当大，存活时间久的对象，并且运行在俩个或者更多核处理器的系统上。但是，任何希望得到低停顿的应用都可以用 CMS 搜集器。使用命令行参数 -XX:+UseConcMarkSweepGC 使用 CMS 搜集器。

和其他搜集器类似，CMS 也是分代搜集器；因此也就分为 minor gc 和 major gc。CMS 搜集器使用单独的 gc 线程跟踪正在运行的程序存活的对象来减少 major gc 的停顿时间。在每次 major gc 周期中，CMS 搜集器进行搜集前先短暂停止应用程序线程一会，在搜集期间，也会再次暂停应用线程。第二次暂停的时间比第一次长。在俩次停顿期间，都会使用多个线程进行工作。搜集器剩下的工作（包括追踪存活对象和清理垃圾）和程序并发执行。minor gc 在 major gc 的过程中可以交错完成，执行过程和 Parallel 搜集器类似（通常来说，在 minor gc 过程，暂停应用程序线程）。

1. 并发模型失败

在年老代变满之前，CMS 搜集器使用多个线程与程序并发运行完成垃圾清理任务。就像先前描述的那样，在正常操作过程中，大多数标记过程和清理过程和程序线程同时运行，程序线程只有短暂的暂停过程。但是，如果在年老代填满前，CMS 搜集器没有结束垃圾搜集，或者此时从年老代申请空间失败，这样，在搜集工作完成前应用都会被停止。这种现象就叫做并发模型失败，如果出现这种情况，说明需要调整 CMS 搜集器参数。如果并发搜集被显式垃圾搜集中断（System.gc()）或者被诊断工具中断，也会报告并发失败。

2. GC 时间太长和 OutOfMemoryError

当垃圾搜集器花费太多时间时，CMS 搜集器抛出 OutOfMemoryError:如果花了超过百分之 98 的时间但是只回收了百分之 2 的内存，就会抛出 OutOfMemoryError。这个功能用于阻止在小内存上长期运行程序。可以使用-XX:-UseGCOverheadLimit 选项关闭它。

这个策略和 Parallel 搜集器的一样，除了并发搜集的时间不计入到 98%的时间内。换句话来说，只包含程序停止时的垃圾时间。出现这种搜集器情况时因为显式的 GC 请求和并发模型失败。

3. 浮动垃圾

CMS 搜集器和其他垃圾搜集器一样，它是标记堆中可达对象的追踪型搜集器。在 Richard Jones 和 Rafael D. Lins 的 *Garbage Collection：Algorithms for Automated Dynamic Memory* 论文中，CMS 是一种增量更新搜集器。因为在 major gc 中，程序线程和垃圾线程并发运行，在并发运行的时候对象可能会从变成不可达状态。这种没有回收的不可达对象就叫做浮动垃圾。浮动垃圾总数和并发搜集周期，对象引用更新频率有关。此外，因为独立搜集年轻代和年老代，每个都是对方的 root。粗略考虑，年老代增大 20%用于保留浮动垃圾。在下一次搜集时浮动垃圾将被清理。

4. 停顿

在并发搜集器周期内，CMS 停止应用程序俩次。第一次停止是从 root（比如，线程栈内的引用，寄存器内的引用，静态对象等）或者从堆内的其他地方（比如，年轻代）开始标记所有的存活的对象。第一次停顿叫做 *initial mark pause*。第二次停顿在并发标记阶段后，它标记在并发标记阶段时程序生成的垃圾。第二次停顿叫做 *remark pause*。

5. 并发阶段

在 *initial mark pause* 和 *remark pause* 之间进行可达对象标记。在并发标记阶段，垃圾搜集器使用多个线程占用多个处理器资源进行处理。因此，对于计算密集型程序，即使程序线程没有被暂停，但是吞吐量也有一定的下降。在 *remark pause* 之后进行 *concurrent sweeping phase*，它并发清理垃圾。一旦搜集器周期结束，CMS 就会停止，此时它不消耗任何计算资源，直到开始下一轮 major gc。

6. 开始并发搜集周期

在年老代满了以后，Serial 搜集器会停止应用程序进行 major gc。相反，CMS 搜集器在年老代变满之前进行搜集行为。要不然，会因为并发模型失败导致应用长时间的停顿。以下是几个开始并发搜集的条件。

基于最近的历史记录，CMS 搜集器估计年老代被填满的时间，维护进行搜集消耗的时间，使用这些动态的估计值，可以在年老代用完之前结束搜集周期。为了安全，这个估值会被保守，因为并发模型失败的代价太高。

当年老代占用率超过一定比例值后也会开始并发搜集。这个比例的默认值是 92%，这个值在往后版本可能会修改。可以使用-XX:CMSInitiatingOccupancyFraction=<N>，N 表示年老代的百分比。

7. 调度停顿

年轻代和年老代搜集行为的停顿单独发生。它们不会重叠，但是结束的时间很快，可能在某次搜集发生后，紧接着就发生另一次搜集的停顿，这样看起来就像一次长的停顿。为了避免这个，CMS 搜集器会试着将 *remark pause* 放在上次和下次年轻代停顿之间。目前还未对 *initial mark pause* 使用这个调度，因为 *initial mark pause* 通常比 *remark pause* 短。

8. 增量模型

注意：增量模型在 Java SE 8 中已经被标记成废弃，在将来版本中可能会被删除。

CMS 搜集器的并发阶段能够增量完成。正如前面说的那样，在并发搜集阶段的线程占用多个处理器。增量模型指的是减少长时间并发阶段的影响，定期的停止并发阶段，让出 CPU 资源给应用程序。这个模型，通常也叫做 i-cms,它把并发任务分成很小的块，并且在年轻代搜集之间调度它们。这个功能对于运行在小数目处理器上，但是需要低的停顿时间的程序非常有用（比如，只有 1 或 2 个核）。

并发搜集周期通常包括以下几步：

- 停止所有应用线程，标识出从 root 可达的对象，然后恢复程序所有线程。
- 使用一个或多个核并发标记可达的对象，此时程序线程还在执行。
- 重新标记在先前步骤中被更改过的对象，使用一个处理器。
- 并发清理不可达的对象并且空间返还给空闲列表，使用一个处理器。
- 并发调整堆大小，准备下次搜集器周期所需的数据结构，使用一个处理器。

正常而言，在整个并发标记阶段，CMS 搜集器使用一个或多个处理器，同样，在清理阶段也使用一个核处理。在只有一个处理器或者俩个处理器的环境中，这会导致程序响应时间超过约束。增量模型通过把并发阶段分成短小的活动，然后在俩个 minor gc 之间调度它们。

i-cms 模型使用 *duty cycle* 控制 CMS 搜集器在放弃处理器之前的工作总量。*duty cycle* 是 CMS 搜集器在俩次年轻代搜集之间能使用的时间的百分比。i-cms 模型能根据程序行为自动计算 *duty cycle*（推荐使用这个方法，让 GC 自动调整），或者使用命令行参数使用固定值。

命令行参数

以下是控制 i-cms 模型的命令行参数，建议选项章节列出了建议的初始值。

Table 8-1 Command-Line Options for i-cms

Option	**Description**	**Default Value, Java SE 5 and Earlier**	**Default Value, Java SE 6 and Later**
-XX:+CMSIncrementalMode	开启增量模型，使用 -XX:+UseConcMarkSweepGC 也可以开启这个功能	disabled	disabled
-XX:+CMSIncrementalPacing	开启自动调整，由 JVM 自动调整 *duty cycle*	disabled	disabled
-XX:CMSIncrementalDutyCycle=<N>	增量模型中 *duty cycle* 的大小 如果开启 CMSIncrementalPacing，这个值表示初始值	50	10

Option	Description	Default Value, Java SE 5 and Earlier	Default Value, Java SE 6 and Later
-XX:CMSIncrementalDutyCycleMin=<N>	开启 CMSIncrementalPacing 后最小的 *duty cycle* 值	10	0
-XX:CMSIncrementalSafetyFactor=<N>	计算 *duty cycle* 时加入的保守值	10	10
-XX:CMSIncrementalOffset=<N>	The percentage (0 to 100) by which the incremental mode duty cycle is shifted to the right within the period between minor collections.	0	0
-XX:CMSExpAvgFactor=<N>	CMS 搜集器进行统计时的计算权重	25	25

建议选项

在 Java SE 8 中，可以使用如下的命令行选项使用 i-cms 功能。

```
-XX:+UseConcMarkSweepGC   -XX:+CMSIncrementalMode \
-XX:+PrintGCDetails   -XX:+PrintGCTimeStamps
```

第一行的俩个参数于开启 CMS 搜集器和 i-cms 功能。第二行的参数用于输出 GC 诊断信息。

对于 Java SE 5 和更早的版本，Oracle 建议使用如下参数：

```
-XX:+UseConcMarkSweepGC   -XX:+CMSIncrementalMode \
-XX:+PrintGCDetails   -XX:+PrintGCTimeStamps \
-XX:+CMSIncrementalPacing   -XX:CMSIncrementalDutyCycleMin=0
-XX:CMSIncrementalDutyCycle=10
```

在 Java SE 8 中也建议使用相同的值，虽然它们在 Java SE 6 就已经是默认值。

基本的故障排除

i-cms 自动调整功能使用程序的统计信息来计算 *duty cycle*，这样就能在堆满之前完成并发搜集。然而，过去的行为并不代表将来的行为，并且估计值可能不够大，从而导致堆满。如果发生了太多的搜集行为，可以参考表 8-2-调整 i-cms 的自动调整功能。

Table 8-2 Troubleshooting the i-cms Automatic Pacing Feature

Step	**Options**
1.增加保守因子.	-XX:CMSIncrementalSafetyFactor=<N>
2. 增大最小 *duty cycle*.	-XX:CMSIncrementalDutyCycleMin=<N>
3. 关闭自动调整，使用固定的 *duty cycle*	-XX:-CMSIncrementalPacing -XX:CMSIncrementalDutyCycle=<N>

9. 测量

Example 8-1，展示了 CMS 搜集器的输出信息，使用的命令行参数是-verbose:gc 和 -XX:+PrintGCDetails，删除了某些 minor gc 的日志。注意里面仍然夹杂着一些 minor gc 的信息。通常是在并发搜集周期发生多次 minor gc。CMS-initial-mark 表示并发搜集周期开始，CMS-concurrent-mark 表示 concurrent marking 阶段结束，CMS-concurrent-sweep 表示 concurrent sweeping 阶段结束。预清理在先前没有讨论，它表示的是 CMS-concurrent-preclean。预清理工作可以和 remark 阶段并发执行。最后一个阶段 CMS-concurrent-reset 表明为下次并发搜集做准备。

Example 8-1 Output from the CMS Collector

[GC [1 CMS-initial-mark: 13991K(20288K)] 14103K(22400K), 0.0023781 secs]

[GC [DefNew: 2112K->64K(2112K), 0.0837052 secs] 16103K->15476K(22400K), 0.0838519 secs]

...

[GC [DefNew: 2077K->63K(2112K), 0.0126205 secs] 17552K->15855K(22400K), 0.0127482 secs]

[CMS-concurrent-mark: 0.267/0.374 secs]

[GC [DefNew: 2111K->64K(2112K), 0.0190851 secs] 17903K->16154K(22400K), 0.0191903 secs]

[CMS-concurrent-preclean: 0.044/0.064 secs]

[GC [1 CMS-remark: 16090K(20288K)] 17242K(22400K), 0.0210460 secs]

[GC [DefNew: 2112K->63K(2112K), 0.0716116 secs] 18177K->17382K(22400K), 0.0718204 secs]

[GC [DefNew: 2111K->63K(2112K), 0.0830392 secs] 19363K->18757K(22400K), 0.0832943 secs]

...

[GC [DefNew: 2111K->0K(2112K), 0.0035190 secs] 17527K->15479K(22400K), 0.0036052 secs]

[CMS-concurrent-sweep: 0.291/0.662 secs]

[GC [DefNew: 2048K->0K(2112K), 0.0013347 secs] 17527K->15479K(27912K), 0.0014231 secs]

[CMS-concurrent-reset: 0.016/0.016 secs]

[GC [DefNew: 2048K->1K(2112K), 0.0013936 secs] 17527K->15479K(27912K), 0.0014814 secs

]

initial mark 的停顿时间通常比 minor gc 停顿时间短。并发阶段（concurrent mark，concurrent preclean，concurrent sweep）的耗时通常比 minor gc 的停顿时间长。注意。并发阶段程序不会停止。remark 阶段的停顿时间通常核 minor gc 的时间一样。remark 阶段的停顿时间通常受程序性能（比如，程序修改对象引用的速度快会增加这个停顿的时长）和离最后一次进行 minor gc 时长的影响（年轻代对象太多会增加这个时长）。

GC 信息解读：

Example 8-1 Output from the CMS Collector

[GC [1 CMS-initial-mark: 13991K(20288K)] 14103K(22400K), 0.0023781 secs]

//CMS 搜集器开始进回收

13991K 表示年老代内对象占用的空间

20288K 表示当前的年老代大小

14103K 表示年轻代和年老代对象总用的总空间大小

22499K 表示当前堆的总大小

0.0023781 secs 表示 init-mark 的耗时

[GC [DefNew: 2112K->64K(2112K), 0.0837052 secs] 16103K->15476K(22400K), 0.0838519 secs]

//minor gc 的信息解读

2112K 表示年轻代对象占用的总大小

64K 表示搜集器后，仍然存活对象的大小

2122K 表示年轻代大小

搜集年轻代耗时 0.0837052

16103K 表示搜集前整个堆中存活对象占用的大小

15476K 表示搜集后整个堆中存活对象占用的大小

22400K 表示整个堆大小

总共耗时 0.0838519 secs //有其他一些耗时

...

[GC [DefNew: 2077K->63K(2112K), 0.0126205 secs] 17552K->15855K(22400K), 0.0127482 secs]

[CMS-concurrent-mark: 0.267/0.374 secs]

//concurrent marking 结束，在这个点之后，开始预清理

//0.267s 用于并发标记阶段，0.374 是系统时间

[GC [DefNew: 2111K->64K(2112K), 0.0190851 secs] 17903K->16154K(22400K), 0.0191903 secs]

[CMS-concurrent-preclean: 0.044/0.064 secs]

//到这里，预清理结束，与 concurrent marking 的意义类似

[GC [1 CMS-remark: 16090K(20288K)] 17242K(22400K), 0.0210460 secs]

//到这里，remark 结束，

[GC [DefNew: 2112K->63K(2112K), 0.0716116 secs] 18177K->17382K(22400K), 0.0718204 secs]

[GC [DefNew: 2111K->63K(2112K), 0.0830392 secs] 19363K->18757K(22400K), 0.0832943 secs]

...

[GC [DefNew: 2111K->0K(2112K), 0.0035190 secs] 17527K->15479K(22400K), 0.0036052 secs]

[CMS-concurrent-sweep: 0.291/0.662 secs]

[GC [DefNew: 2048K->0K(2112K), 0.0013347 secs] 17527K->15479K(27912K), 0.0014231 secs]

[CMS-concurrent-reset: 0.016/0.016 secs]

[GC [DefNew: 2048K->1K(2112K), 0.0013936 secs] 17527K->15479K(27912K), 0.0014814 secs

]

9. Garbage-First 搜集器

Garbage-First（G1）搜集器也是 server 类型的垃圾搜集器，它使用在多核，大内存的环境中。它的目标是保证吞吐量的前提下实现低的停顿时间。在整个堆上的操作，比如全局标记行为，都是和程序线程并发运行。这可以防止堆和存活数据成比例的中断。

G1 使用以下几个技术来实现高性能和和停顿时间目标。

堆被划分成大小一致的区域，并且每个区域都是一片连续的虚拟内存。G1 并发执行标记阶段寻找存活的对象。在标记阶段完成后，G1 就能知道哪些区域是最空的（充满垃圾的区域）。G1 首先回收空闲区域，这样总是能够让出大量的空闲空间，这就是为什么叫 Garbage-First 的原因。和它的名字一样，G1 集中搜集，压缩那些充满垃圾的区域。G1 使用预测模型处理停顿时间目标，基于用户指定的停顿时间选择进行搜集的区域。

G1 复制从多个区域复制对象到一个区域，然后压缩，释放内存。这个过程是并行操作，这样就能减少停顿时间并且增大吞吐量。因此，在每次搜集时，G1 连续这样工作就能减少内存碎片。这个功能超过前面垃圾搜集器许多，CMS 搜集器不进行压缩，Parallel 搜集器仅执行整个堆的压缩-会造成可观的停顿时间。

需要注意的是 G1 不是实时搜集器。它只是大概率满足停顿时间，但不是绝对满足。基于以前的搜集数据，G1 预测在目标时间内能搜集器多少个区域。G1 有一个精确可信的代价模型决定在停顿时间内存搜集哪些和多少区域。

G1 的第一个关注点是给大内存，低 GC 延时的程序提供了解决方案。这意味着堆大小大概是 6GB 或者更大，并且停顿时间在 0.5s 左右。

运行在 CMS 或者 Parallel 搜集器上的程序，如果有以下特点，切换到 G1 应该更加合适。

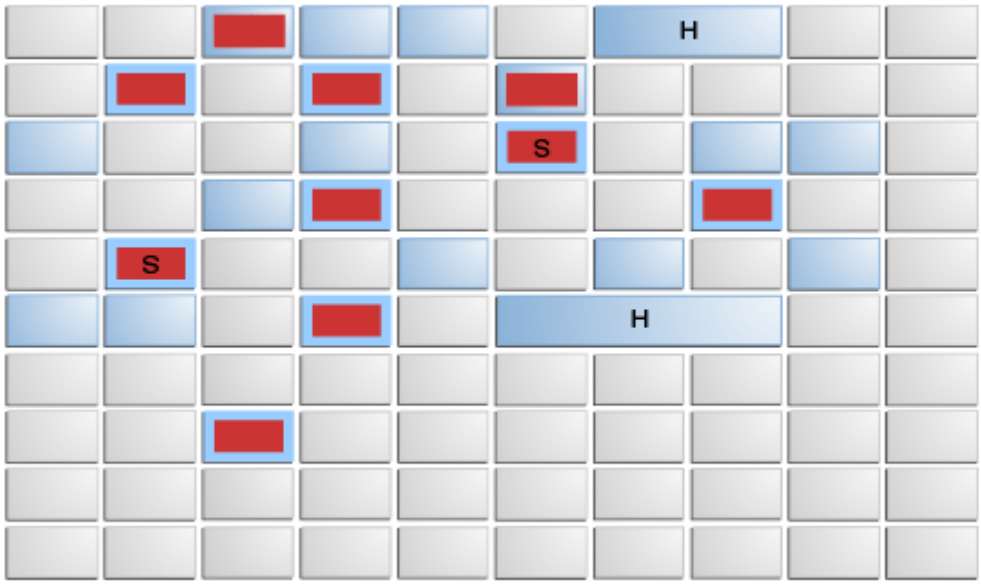
- 存活数据占用内存超过 50%。
- 申请内存的速度和对象提升速度变化大。
- 无法接受长时间的 GC 停顿（超过 0.5s）。

G1 是预计替换 CMS 的搜集器。比较发现，G1 是一个比 CMS 更好的方案。唯一一点不同的是 G1 是压缩型的搜集器。同样，G1 允许用户指定期望的停顿时间。

和 CMS 比较来说，G1 用于期望更短停顿时间的程序。

G1 把堆划分成固定大小的区域（灰色格子），如下图 9-1 所示。

Figure 9-1 Heap Division by G1



G1 对内存进行逻辑分代。以上图来说，年轻代是淡蓝色。内存分配在逻辑上是年轻代的区域进行，当年轻代满了以后，这些区域就会被搜集（年轻代搜集）。在某些情形中，年轻代以外的区域（深蓝色）同时被搜集，这个过程叫做混合搜集。图中红色的格子表示正在搜集的区域。由图可知，正在进行混合搜集，年轻代和年老代同时被搜集。垃圾搜集会伴有压缩行为，它复制存活的对象到选定的空闲区域。根据对象幸存的次数，对象可以被复制到 survivor（图中标记为 S 的区域）或者复制到年老代的区域（图中未指出）。H 区域包含对象的大小大于区域的一半。更多请查看 G1 搜集器中大对象和给大对象开辟空间一章。

1. 申请内存（回收）失败

和 CMS 一样，G1 搜集器和程序并发执行，所以也会出现垃圾回收速度低于程序申请内存的速度。CMS 相似的行为查看 CMS 搜集器中的并发模型失败

如果在 G1 复制存活数据到其他区域时发生内存不足错误。G1 尝试压缩存活数据来完成复制行为。如果找不到空闲区域保存存活数据，那就会发生 stop-the-world（STW）搜集。

2. 浮动垃圾

在 G1 搜集过程中，某些对象可能会死亡。G1 使用 snapshot-at-the-beginning（SATB）技术保证在搜集过程中找到所有存活的对象。SATB 假设并发标记时存活的对象就是存活的对象。SATB 允许浮动垃圾的方式和 CMS 类似。

3. 停顿

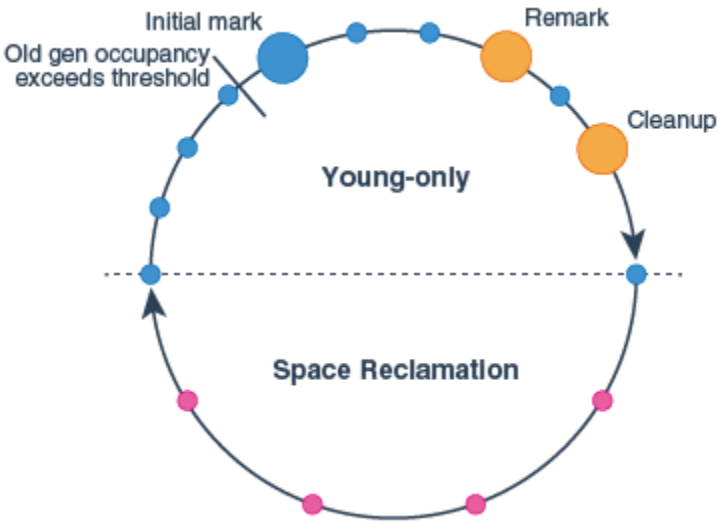
在复制对象到新区域时，G1 让程序停顿。搜集年轻代和进行混合搜集时都会让应用停顿。CMS 只在 *re-marking* 和 *initial-mark* 阶段让程序停顿，在 G1 中，*initial marking* 是驱逐停顿的一部分。在搜集末尾，G1 有一个 *cleanup* 阶段，这个阶段部分是 STW，部分是并发进行。

4. Card Tables 和并发阶段

如果不是对整个堆进行搜集（增量搜集），G1 需要知道从堆未搜集部分到正在搜集部分的指针在哪里。未搜集的部分通常是老年代，正在搜集的部分通常是年轻代。保持年老代到年轻代指针的数据结构叫做 *remembered set*。Card table 是一个特殊的 *remembered set*。Java HotSpot VM 使用一个字节数组作为 card table。一个 byte 就是一个 card。一个 card 对应着堆中的一个地址范围。把 byte 值变成脏值就表示 card 变脏，脏值包含着 card 地址范围内从年老代到年轻代的新指针。

处理 card 就是查看 card 是否是从年老代到年轻代的指针，并且可能会处理这个指针，比如传递它给其他的数据结构。

G1 的并发标记阶段标记程序存活的对象。并发标记阶段在驱逐停顿（*init marking* 已经结束）和 *remark* 之间。（如下图-来自 jdk9）



并发 *cleanup* 阶段添加空闲区域到空闲内存列表，并且清空这些区域的 *remembered sets*。另外，可能会有一个线程处理被修改过的 card table 项。

5. 开始并发搜集器周期

正如前面提到的那样，混合搜集时搜集年轻代和年老代。为了搜集年老代，G1 在堆中并发标记存活对象。当堆的使用率达到 InitiatingHeapOccupancyPercent 时开始进行并发标记阶段。使用命令行参数-XX:InitiatingHeapOccupancyPercent=<N>来设置这个参数。默认值是 45。

6. 停顿时间目标

使用 MaxGCPauseMillis 参数设置停顿时间目标值。G1 使用先验模型决定在目标时间内进行多少搜集工作。在一次搜集后，G1 选择下一次搜集能够搜集的区域。这些搜集区域也包含年轻代区域（它的大小决定了逻辑年轻代的大小）。G1 通过控制搜集区域中年轻代的数目来控制搜器的停顿时间。你可以像使用其他搜集器一样，通过命令行指定年轻代的

大小，但是手动设置这个参数可能会影响 G1 满足目标停顿时间的能力。另外，你也可以设置俩次停顿时间之间的间隔（GCPauseIntervalMillis）。默认的 MaxGCPauseMillis 是 200ms。默认的 GCPauseIntervalMillis 是 0，表示没有限制。

10. 调整 G1 搜集器

此章节描述如何调整 G1 搜集器。

正如前面章节 Garbage-First 搜集器描述的那样，G1 搜集器是分区域和分代的搜集器，它把堆划分成多个大小相等的区域。在 JVM 启动后，G1 搜集器根据堆的大小把区域划分成 1MB 到 32MB 的大小。总的区域大小不超过 2048 个区域。Eden，Survivor，old 都是逻辑上的代。

G1 会尽量满足停顿时间约束（软实时）。在年轻代搜集时，G1 调整年轻代的大小来满足软实时目标。更多请查看 Garbage-First 搜集器章节中的停顿和停顿时间目标。

在混合搜集期间，G1 搜集器根据混合搜集区域的数目，每个区中存活对象的百分比，能接受的堆浪费百分比来调整搜集的年老代区域数目。

G1 GC 使用增量并行复制多个区域（Collection Sets（CSet））中的对象到新的区域来完成内存压缩，这样可以减少堆中的内存碎片。在不超过目标停顿时间的前提下，从那些包含垃圾最多的区域开始，回收尽可能多的区域。

G1 使用单独的 Remembered Sets（RSet）保存区域内的引用。每个单独的 RSets 能够单独并行进行搜集，这样就不用扫描整个堆。The G1 GC uses a post-write barrier to record changes to the heap and update the RSets（搜一下 post-write-barrier）。

1. 垃圾搜集阶段

回收空间的停顿时间由 STW-年轻代和混合搜集停顿组成（查看申请内存（回收）失败章节），G1 同样有并行，并发，多阶段标记周期。在标记周期内，G1 使用 SATB 算法，在标记阶段开始时，该算法获取堆中存活的对象的快照。获取到的存活对象中同样包含了新生成的对象。The G1 GC marking algorithm uses a pre-write barrier to record and mark objects that are part of the logical snapshot.（搜一下 pre-write）。

2. 年轻代搜集

G1 搜集器让大多数开辟空间的行为发生在 eden 区域。在年轻代搜集时，G1 同时搜集上次搜集时标记的 eden 和 survivor 区域。eden 和 survivor 区域中存活的对象复制到新的区域。年龄足够老的对象复制到年老代，要不然复制到下一次搜集（年轻代搜集或者混合搜集）时会被回收的 survivor 区域。

3. 混合搜集

一旦成功完成并发标记周期，G1 从年轻代搜集切换到混合搜集。混合搜集时，G1 搜集一些年老代区域。可以通过命令行参数精确控制这个数目（查看建议章节）。在 G1 搜集足够多的年老代区域后，G1 恢复执行年轻代搜集直到下一个标记周期完成。

4. 标记周期阶段

标记周期有如下阶段：

- **Initial marking phase**：在这个阶段中，G1 标记 root 集合，这个阶段通常在普通的年轻代搜集（普通的年轻代搜集会有 STW）内。
- **Root Region Scanning phase**：G1 搜集器扫描 initial marking 阶段标记的 survivor 区域的引用，这些引用都指向年老代，同样，年老代被引用的对象也被标记。这个阶段和程序并发执行（没有 STW），必须在下一次 STW 年轻代搜集开始之前完成。
- **Concurrent marking phase**：G1 在整个堆中寻找存活的对象。这个过程和程序并发执行，但是可能被 STW 年轻代搜集中断。
- **Remark phase**：这个阶段 STW，用于帮助完成标记周期。G1 释放 SATB buffer，追踪未遍历的存活对象，然后开始处理引用。
- **Cleanup phase**： 最后一个阶段，G1 执行 STW 操作来清理和统计 Rset，在统计时，G1 标记完全空闲的区域和用于混合搜集的区域。在 Cleanup 阶段，清理空闲区域和返还空闲区域给内存列表的过程和程序并发执行。

5. 重要的默认值

G1 是一个自适应搜集器，使用默认值的时候效率通常也很高效。表 10-1 列出了 HotSpot JVM build 24 中重要的默认值。你可以调整以下参数来让 G1 更适合你的应用程序。

Table 10-1 Default Values of Important Options for G1 Garbage Collector

Option and Default Value	**Option**
-XX:G1HeapRegionSize=n	设置 G1 区域的大小，这个应该是 2 的幂次方并且应该在 1MB 到 32MB 之间。根据最小堆大小来算应该有 2048 区域
-XX:MaxGCPauseMillis=200	设置期望的最大停顿时间。默认值是 200ms，这个默认值可能不适合你的堆大小。

Option and Default Value	Option
-XX:G1NewSizePercent=5	用于设置年轻代的最小百分比，默认值是 5.[Foot1] 这是一个实验性参数。更多请参考怎么解锁实验性 VM 参数。这个参数会覆盖 -XX:DefaultMinNewGenPercent。
-XX:G1MaxNewSizePercent=60	设置年轻代的最大百分比。默认值是百分之 60[Footref1] 这是一个实验性参数。更多参考如何解锁实验性参数 这个设置会覆盖 -XX:DefaultMaxNewGenPercent
-XX:ParallelGCThreads=n	设置 STW 时的工作线程。n 默认设置成处理器逻辑数目。最大值是 8。如果超过 8 个处理器，n 会被设置成处理器数目的 5/8。在 SPARC 系统 n 的默认值会被设置成 5/16。
-XX:ConcGCThreads=n	设置并行标记的线程数。n 大概设置成并行搜集器线程数目的 1/4(ParallelGCThreads)。
-XX:InitiatingHeapOccupancyPercent=45	设置触发标记周期的阀值。默认值是堆的百分之 45
-XX:G1MixedGCLiveThresholdPercent=85	设置年老代区域参与混合搜集的占用阀值。默认值百分之 85[Footref1] 这是一个实验性参数。更多请参考如何解锁实验性参数 这个参数会覆盖 -XX:G1OldCSetRegionLiveThresholdPercent

Option and Default Value	Option
-XX:G1HeapWastePercent=5	设置能接受的浪费堆的百分比。Java HotSpot VM 当回收的百分比小于这个值时不会初始化混合搜集。默认值是百分之 5.Footref1
-XX:G1MixedGCCountTarget=8	设置混合搜集周期中混合搜集的目标数目。年老代最多有 G1MixedGCLiveThresholdPercent 大小的存活数据，混合搜集的目标值。默认值是 8 个混合搜集。混合搜集目标数目在这个数目内 Footref1
-XX:G1OldCSetRegionThresholdPercent=10	设置参与混合搜集周期的年老代区域数目的上线。默认值是 Java 堆的百分之 10Footref1
-XX:G1ReservePercent=10	设置用于防止空间 overflow 而保留的内存大小。默认值是百分之 10.当你增加或减少这个值。确保堆也的做同样总量的调整 Footref1

Footnote1 的参数在 Java HotSpot VM build 32 和更早版本之前无法使用

6. 怎么解锁实验性参数

为了使用实验性的参数，你必须首先解锁它们。你可以使用-XX：+UnlockExperimentalVMOptions 显式的开启。如下：

```
java -XX:+UnlockExperimentalVMOptions -XX:G1NewSizePercent=10
-XX:G1MaxNewSizePercent=75 G1test.jar
```

7. 建议

在评估和微调 G1 搜集器时，请牢记以下几点：

- 年轻代大小:应该避免显式使用-Xmn 显式设置年轻代大小和其他相关的参数，比如 -XX:NewRatio。固定年轻代的大小会覆盖目标停顿时间。
- 目标停顿时间：在你调整和评估任何一个 GC 时，都会遇到延迟和吞吐量之间的平衡。

G1 搜集器是有一致停顿的增量搜集器，但应用程序线程同样有更多的花销。吞吐量目标是百分之 90 时间用于程序，百分之 10 时间用于垃圾搜集。而 Parallel 搜集器是百分之 99 时间用于程序，百分之 1 时间用于垃圾搜集。因此，如果你更注重吞吐量，那么你应该对停顿时间更松一些。如果设置的目标值太过于激进，那就表示你愿意接受搜集器占用更多的花销，这会直接影响吞吐量。当你评估 G1 的延迟时，设置你期望的软实时值，G1 会尽量满足这个目标，同样，吞吐量会受到影响。更多请查看章节 G1 搜集器内的停顿时间目标。

- 征服混合搜集:在优化混合搜集时，可以尝试以下几个参数，更多请查看重要的默认值：

  -XX:InitiatingHeapOccupancyPercent:改变标记的阀值。

  -XX:G1MixedGCLiveThreholdPercent 和-XX:G1HeapWastePercent:改变混合搜集的决策。

  -XX:G1MixedGCCountTarget 和-XX:G1OldCSetRegionThresholdPercent:调整年老代区域的 CSet。

8. 溢出和耗尽时的日志

当你看到日志里面有溢出和耗尽的日志。这个表示 G1 没有足够的空间给 survivor 区域或者提升对象到年老代。这说明此时的堆已经是最大值。比如下面这个信息：

- 924.897: [GC pause (G1 Evacuation Pause) (mixed) (to-space exhausted), 0.1957310 secs]
- 924.897: [GC pause (G1 Evacuation Pause) (mixed) (to-space overflow), 0.1957310 secs]

为了缓解这个问题，可以使用下面这些调整：

- 提升-XX：G1ReservePercent 的值来提升保留的内存大小（根据堆的总大小来决定）。
- 通过减少-XX:InitiatingHeapOccupancyPercent 来提前开启标记周期。
- 提升-XX:ConcGCThread 选项的值来增加并行标记的线程数目。

  这些值的更多描述参考重要的默认值。

9. 超大对象和申请超大内存

对于 G1 搜集器，超过半个区域大小的对象都是超大对象。这些对象直接在年老代的超大区域中申请空间。这些超大区域都是连续的区域。StartsHumongous 标志连续区域的开始，ContinuesHumongous 标志连续区域的继续。

在开始申请任何超大区域之前，如果需要，会检查标志的阀值，初始化一个并发周期。

在清理阶段的标志阶段后和整个垃圾搜集期间会释放超大对象垃圾。

为了减少复制对象的花销，超大对象不包含在任何驱逐停顿中。在 full gc 周期内直接在原地压缩超大对象。

因为每个 StartsHumongous 和 ContinuesHumongous 区域内仅包含一个超大对象，在超对象的末尾到区域的末尾是未使用的空间。这样对于刚好大于区域数倍的对象就会导致内存碎片。

如果你看到由于大量超大对象导致连续初始化并发周期，如果这些申请操作导致你的年老代碎片，可以增大-XX:G1HeapRegionSize 来提升区域的大小，这样可以使以前是超大对象的对象不再是超大对象。

11. 其他考虑

此章节包含一些其他影响垃圾搜集的因素。

1. Finalization，Weak，Soft，Phantom 引用

某些程序使用 finalization，weak，soft，phantom 引用和 GC 交互。这些功能在 Java 语言层面可以用来构建高性能组件。一个常见的列子就是使用 finalization 来关闭文件描述符，文件描述符关闭的实际依赖 GC 回收的及时性。依靠 GC 管理与内存无关的资源通常是一个坏注意。

序言内相关文档章节包含的一篇文章深入讨论 finalization 的陷阱，并且给出了怎么避免这个问题。

2. 显式垃圾回收

另外一个和 GC 交互的方式是调用 System.gc()来主动发起 Full GC。这样会导致没必要的 major gc，所以应该避免这样做。可以使用-XX:DisableExplictGC 让 VM 忽略 System.gc()的调用。

一个常见的显式 GC 是 RMI 的分布式垃圾回收（DGC）。应用程式使用 RMI 引用到另一个虚拟机中的对象。如果不偶尔调用本地的垃圾搜集，则无法搜集这些垃圾，所以 RMI 定期调用 GC 进行 Full GC。搜集频率能够使用下面这些参数来干预。如下：

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

默认是一分钟一次 GC，样例指定一小时进行一次显式 GC。但是，这样会导致一些对象长久存在。最大值是 Long.MAX_VALUE。

3. Soft 引用

Soft 引用在 server 类型 VM 中的存活时间比在 client 类型 VM 中长。回收速率可以用参数-XX:SOftRefLRUPolicyMSPerMB=<N>来控制，它表示 Soft 引用在堆空闲每兆字节时保持存活（没有强引用引用）的毫秒数。默认是 1000MS 每兆字节，这表示堆空闲每兆字节保持 Soft 引用 1s 的存活（最后一个引用它的强引用已经被回收）。这只是一个大约值，因为 Soft 引用只在垃圾回收时才会被清理，垃圾回收只是偶然发生。

4. Class Metadata

Java 类在 Java HotSpot VM 里的表现形式叫做 Class Metadata。在以前的版本中，Class Metadata 在持久代中申请空间。在 JDK 8 中，持久代被移除，Class Metadata 直接在本地内存中申请空间。默认情况下，Class Metadata 使用的本地内存不受限制。使用选项 MaxMetaspaceSize 来限制 Class Metadata 能够使用的最大本地内存。

Java Hotspot VM 显式管理 metadata 使用的内存。直接从 OS 请求空间，然后把空间分成块。一个 class loader 从它自己的内存块中给 metadata 开辟空间（块绑定到对应的 class loader）。当从 class loader 上卸载一个 class 的时候，块就会被回收，还给 OS。Metadata 使用 mmp 调用申请空间，而不是使用 malloc 调用。

如果开启选项 UseCompressedOops，并且使用了 UseCompressedPointers，那么将本地内存分成俩个不同逻辑的区域用于 class metadata。UseCompressedClassPointers 使用 32 位偏移代表 64 位系统中的指针。一个区域用于保存压缩的 32 位指针（32 位的偏移量）。这个区域的大小可用 CompressedClassSpaceSize 参数来控制，默认是 1GB。用于类压缩指针的空间是保留空间，使用 mmp 开辟空间并且在需要时被提交。MaxMetaspaceSize 包含压缩指针已提交的空间和其他 class metadata。

当 Java class 被 unload 时，对应的 class metadata 被释放。由于 GC 的原因，Java 类被卸载，同样， unload class 和释放 class metadata 也会引起 GC。当 class metadata 提交的空间到达一定程度就会导致 GC。在 GC 后，根据从 class metadata 中释放出的空间总数，程度值可能会变高或者降低。程度值被提高以免过快的导致另一次 GC。程度值使用参数 MetaspaceSize 来设置。这个值是提高还是降低由参数 MaxMetaspaceFreeRatio 和 MinMetaspaceFreeRatio 来决定。如果 class metadata 可用的已提交空间的占总的已提交空间的百分比大于 MaxMetaspaceFreeRatio，程度值就会变低。如果低于 MinMetaspaceFreeRatio，程度值就会变高。

指定一个较大的 MetaspaceSize 值来避免 class metadata 过早的触发 GC。给 class metadata 指定多大的空间根据程序的不同而不同，这个并不存在一个通用的值。根据平台不同，MetaspaceSize 的值默认从 12MB 到 20MB。

关 metadata 的信息也会打印到堆的输出中。比较典型的输出如下：

```
Heap
  PSYoungGen      total 10752K, used 4419K
    [0xfffffffff6ac00000, 0xfffffffff6b800000, 0xfffffffff6b800000)
    eden space 9216K, 47% used
      [0xfffffffff6ac00000,0xfffffffff6b050d68,0xfffffffff6b500000)
    from space 1536K, 0% used
      [0xfffffffff6b680000,0xfffffffff6b680000,0xfffffffff6b800000)
    to   space 1536K, 0% used
      [0xfffffffff6b500000,0xfffffffff6b500000,0xfffffffff6b680000)
  ParOldGen       total 20480K, used 20011K
      [0xfffffffff69800000, 0xfffffffff6ac00000, 0xfffffffff6ac00000)
    object space 20480K, 97% used
      [0xfffffffff69800000,0xfffffffff6ab8add8,0xfffffffff6ac00000)
```

Metaspace used 2425K, capacity 4498K, committed 4864K, reserved 1056768K

class space used 262K, capacity 386K, committed 512K, reserved 1048576K

在包含 Metaspace 那行，used 表示 loaded classes 使用的总空间。capacity 表示当前已开辟的块中 metadata 可用的空间（ArrayList 的容量）。committed 表示块可用空间的总值。reserved 表示给 metadata 保留的总空间（没有必要提交）。class space 开头的行包含 compressed class pointers 的 class metadata 的值。