火龙果•整理
uml.org.cn

Sun HotSpot JVM 1.4.2 调优

目录

1.序言
2.虚拟机中的“代”
  2.1.性能考虑
  2.2.测量
3.调整代大小
  3.1.堆的总体大小
  3.2.新生代
  3.3.新生代的保证
4.收集器类型
  4.1.何时使用吞吐收集器
  4.2.吞吐收集器
    4.2.1.大小自适应
    4.2.2.Aggressive 堆
    4.2.3.吞吐收集器的测量法
  4.3.何时使用并发收集器
  4.4.并发收集器
    4.4.1.并发的额外开销
    4.4.2.新生代的保证
    4.4.3.完全收集
    4.4.4.漂浮垃圾
    4.4.5.暂停
    4.4.6.并发阶段
    4.4.7.并发收集的测量
    4.4.8.并发收集器的并行次要收集
  4.5.何时使用增量收集器
  4.6.增量收集器
    4.6.1.增量收集器测量
5.其他考虑事项
6.总结
7.其他文档
  7.1.输出示例
  7.2.常见问题

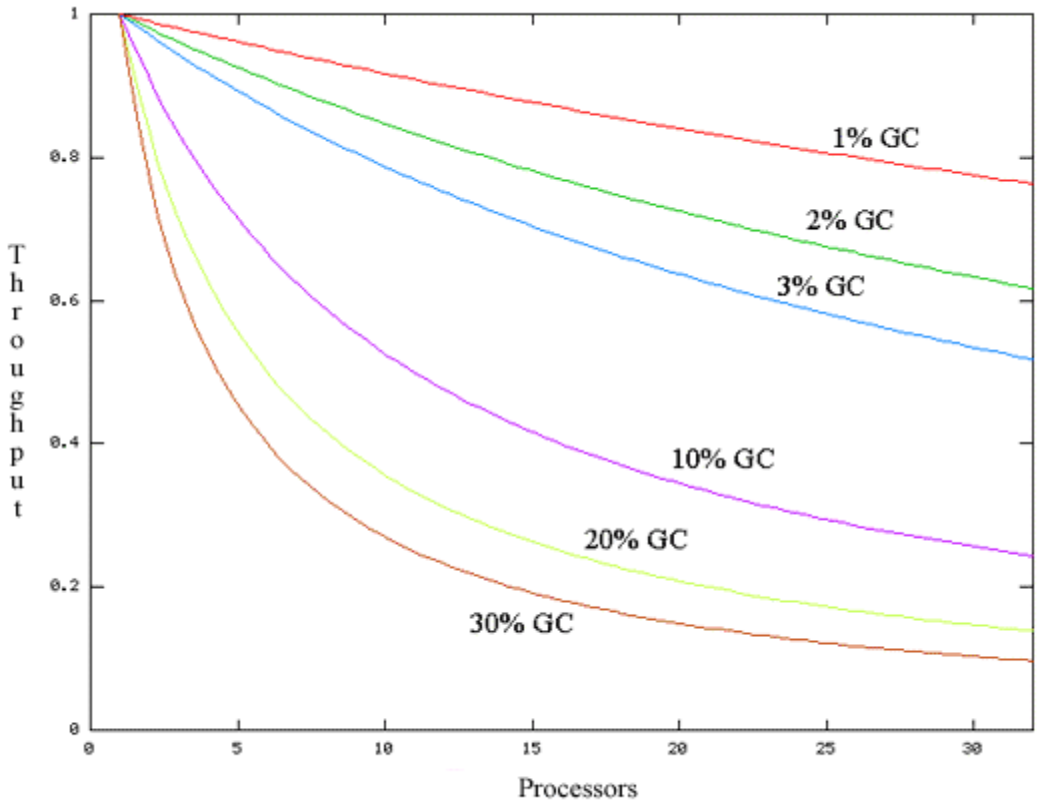
1. 序言

从小型桌面应用到运行在大型服务器的 web 服务，Java 2 平台被广泛的应用。在 1.4.1 版本的 J2SE 平台中，引入了 2 种新的*垃圾收集器*（garbage collector），现在总共有 4 种垃圾收集器供我们选择。那么，应该如何选择垃圾收集器？又有哪些因素可以作为选择的依据？本文档描述了垃圾收集器共有的特征，并对于在单线程，stop-the-world 的收集器上如何最大限度的利用这些特征给出了调整指导。最后，讨论了其他 3 种收集器独有的特征以及在 4 种垃圾收集器中做出选择的一些标准。

对于用户而言，在什么情况下，垃圾收集器会带来性能问题？对于大部分应用来说，垃圾收集器不会带来性能问题，也就是说，即使适垃圾收集器的运行会带来很短的暂停，但是应用还是可以以令人满意的性能运行。但是对于使用了大量的线程、处理器、Socket 以及内存的大型应用而言（使用默认的垃圾收集器），情况就不同了。

Amdahl 观察到，大部分工作是无法很好的并行处理的，有些工作总是顺序化的，所以，无法从并行机制中获益。对于 J2SE 平台来说，也是这样，Java 虚拟机从最初一直到 1.3.1 版本，都没有并行的垃圾收集器，所以，相对于其他的并行应用，垃圾收集器所带来的性能影响在多处理器系统上会有所增长。

下图描述的是一个具有良好弹性的理想系统，如果不考虑垃圾收集的话。红线表示在单处理器系统上的应用，垃圾收集只占用 1%的时间，当迁移到有 32 个处理器的平台上的时候，有超过 20%的*吞吐量*（throughput）的损失；在单处理器系统上垃圾收集时间占 10%的应用（不考虑那些令人无法接受的垃圾收集时间），当扩展到 32 处理器系统上时，吞吐量的损失超过了 75%。



这表明了，在开发小型应用时可以忽略不计的速度问题，当扩展到大型应用系统时却成为最为突出的性能瓶颈。但是对于这种瓶颈的小小的改良就可能收获性能上的显著提高，所以，对于大型应用来说，调整垃圾收集器是非常有价值的。

对于大部分应用来说，默认的垃圾收集器都是可以满足需要的。其他的几种垃圾收集器因为有一些特殊的行为，使用起来比较复杂。除非应用有特殊需求，一般情况下建议选用默认的垃圾收集器。当然，也有例外情况，对于那些运行在有大量内存和处理器的主机上的大型应用，可以首先尝试 aggressive 堆（heap）选项（-XX:+AggressiveHeap），稍后有详细描述。

本文选用 J2SE 1.4.2（Sun HotSpot），Sun Solaris 操作系统（SPARC 平台版本），因为这个平台对于硬件和软件来说都有很好的扩展性。当然了，本文也适用于其他平台，如 Linux，Microsoft Windows， Sun Solaris（X86 平台版本）。尽管在不同平台上虚拟机的命令行参数是一样的，但是可能其他平台上的默认值和本文描述的有所差别。

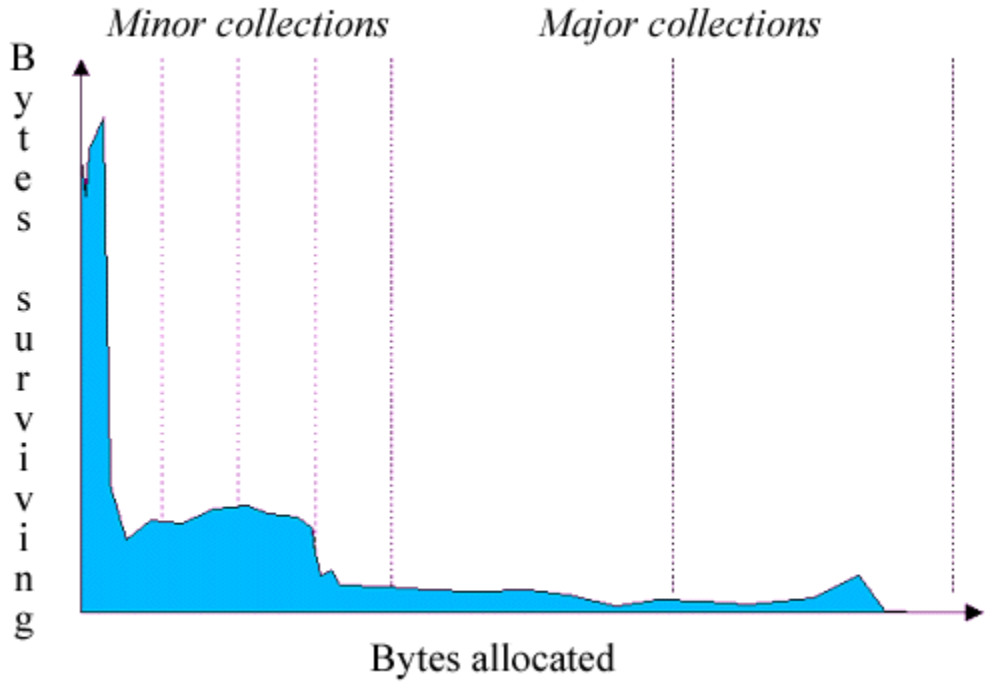
2. 虚拟机中的“代”

J2SE 平台的一个特征就是它为开发人员实现了内存的申请和释放，但是，当垃圾收集成为主要的性能瓶颈时，我们就有必要对它的实现进行深入的了解。垃圾收集器可以对应用使用对象的方式进行一个设定，反映在一些可调整的参数上，通过这些参数，在保证抽象能力的前提下提升了虚拟机的性能。

从运行的程序中，没有可以到达某一对象的指针，那么这个对象就被认为是“垃圾”。最直截了当的垃圾收集算法就是枚举每个可到达的对象，剩下的对象就是可以进行回收的垃圾对象了。这种方案所消耗的时间和活动对象的数量是成比例的，所以，对于要保持大量活动对象的大型应用，显然是不适用的。

从 J2SE 1.2 开始，虚拟机将几种不同的垃圾收集算法组合在一起使用，就是“*分代收集*”（generational collection）。在*幼儿收集器*（naive garbage collection）检查堆中活动对象的同时，分代收集器通过分析一些观测属性来避免额外的工作。

在这些观测属性中最重要的就是*幼儿死亡率*（infant mortality）。下面图表中的蓝色区域就是对象存活期的典型分布。其中 X 轴表示对象的存活时间，在分配字节时测量。Y 轴表示的是对应存活时间的对象字节总量。左侧的顶点表示对象可能在分配之后很短的时间内就被回收，例如在循环中的枚举对象，仅仅能够存活一个循环。



有些对象能够存活很长时间，所以上图的 X 轴向右延展。例如那些在初始化时就创建并且能够存活直到进程终止的对象。在这两个极端之间，就是一些只在中间计算才存活的对象，就是我们看到的到幼儿死亡率

右侧的区域。可能有些应用的对象存活周期分布有些差异，但令人惊讶的是大部分应用都和上图吻合。通过关注主要的“夭折”（die young）对象来进行垃圾收集是行之有效的。

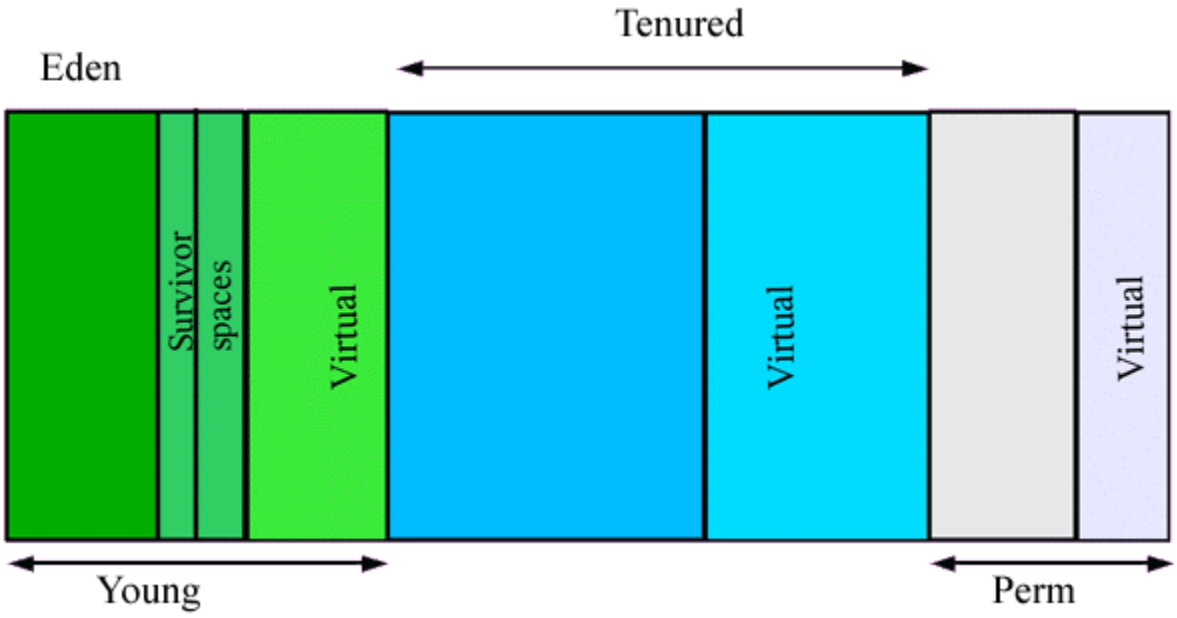
为了针对这种情况进行优化，所以将内存进行*分代*管理，也就是说内存池保持不同年龄段的对象。在分代管理的内存中，当一个代被对象充满的时候，就在这个代上进行垃圾收集。对象是在*新生代*（young generation）区域中进行分配，因为大部分对象在这个代中就已经死亡。当新生代被填充满之后，在其上进行一次*次要垃圾收集*（minor collection），由于在新生代有很高的夭折率，所以次要收集算法针对这种情况进行了优化。这种收集的消耗和被其收集的对象的数量成比例。充满死亡对象的新生代收集起来非常的快。一些仍然存活的对象被移动到*旧生代*（tenured generation），当旧生代需要进行垃圾收集的时候，就会进行*主要垃圾收集*（major collection），这种收集方式比起次要收集来要慢得多，因为它涉及了全部的活动对象。

下图显示，次要收集的间隔时间比较长，这样能够确保大部分对象都已经死亡。为了确保次要收集发生的间隔能够足够长，需要新生代有足够的空间，这样才能够使得次要收集充分利用新生代中高夭折率的特点。对于那些对象存活周期分布比较奇特的应用情况就不同了，还有，不合适的代大小设置也会使得在对象死亡之前就不得不进行收集。

默认的垃圾收集器可以应用在小型的和大型的应用，但是它的默认参数设置对于小型应用更高效。对于很多服务器应用来说，这些参数并不合适，这就引出了本文的中心原则：

> 如果垃圾收集成为性能瓶颈，建议你调整代的大小参数。并检查垃圾收集的输出信息，研究性能对于垃圾收集参数的敏感度

默认的代配置如下图所示：



在初始化时，虚拟机保持一个很大的地址空间，但并不真正的从物理内存中申请（按需申请）。用来存放对象的地址空间被划分成两部分：新生代和旧生代。

新生代由一个 Eden 和两个*存活空间*（survivor space）组成，对象在 Eden 中进行创建和内存分配。在任何时候，都有一个存活空间是空的，以用来容纳从 Eden 和另外一个存活空间复制过来的活动对象。对象以这种方式在两个存活空间之间进行复制，直到它存活时间较长而被复制到旧生代。

包括 1.2 版本的虚拟机（Solaris 操作系统）在内的其他虚拟机产品，都是使用两个大小相等的存活空间来进行对象复制，而不是像图中所示的一个 Eden 和两个存活空间。这就意味着，通过调整新生代大小来调整性能并不总是具有可比性的。

在旧生代中，有一部分比较特殊的，叫做*持久代*（permanent generation），它用来存放虚拟机自己的数据，例如*类*（class）和*方法*（method）。

2.1.性能考虑

对于垃圾收集的性能，主要有两个指标。*吞吐量*（throughput）指用来进行垃圾收集之外工作所用的时间占总时间的百分比，一般要通过长时间的观察和测量。吞吐量包括了分配内存所花费的时间在内（一般来说无需对分配进行调优）。*暂停*（Pause）指由于进行垃圾收集而导致应用无法响应的时间。

用户对于垃圾收集有不同的需求，例如，对于 Web 服务器应用来说，吞吐量是要着重考虑的，而暂停时间可能由于网络的反应时间而不那么明显；而对于一个交互式图形界面的应用来说，即使是短暂的暂停都会带来非常不好的用户体验。

通常来说，如何设置代的大小是在这些考虑因素之间作出的一个权衡。例如，将新生代设置得很大会得到很好的吞吐性能，但是会增加暂停时间；反之，较小的新生代设置会减小暂停时间，但是降低了吞吐量。一个代的大小不应该影响在其他代上进行垃圾收集的频率和暂停时间。

对于代的大小设置，没有一个精确的计算方法。同时考虑应用对内存的使用特征以及用户对垃圾收集的需求，才能够做出最好的选择。因此，虚拟机默认的垃圾收集的相关参数可能不是最优的，需要通过用户定制这些命令行选项加以调整。

2.2.测量

对于特定的应用来说，吞吐量和内存占用还是比较容易测量的。例如对于一个 web 服务应用，可以使用压力测试工具来测量它的吞吐量，内存占用可以使用 Solaris 操作系统提供的命令 pmap 来测量。另外，通过打开虚拟机的详细诊断信息来估算垃圾回收的暂停时间。

通过使用虚拟机选项-verbose:gc，能够将每次收集时的一些信息打印出来。需要注意的是，对于不同版本的 J2SE 平台来说，-verbose:gc 的输出格式可能稍有差异。下面的示例是一个大型的服务器应用的-verbose:gc 输出。

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

上面的输出中，表明虚拟机进行了两次次要收集和一次主要收集。箭头两端的数字

325407K->83000K（第一行）

表示在进行垃圾收集之前和之后活动对象的总大小。在次要收集之后的数字（83000K）包含了那些并不是必要活动的对象，但是还不能被回收。因为这些对象可能本身是活动的，或者有来自旧生代的引用。括弧中的数字

(776768K)（第一行）

表示总共的可用空间的大小。这个大小不包括持久代在内，是堆的大小减去一个存活空间的大小。此次次要收集大约用了 1/4 秒

0.2300771 secs （第一行）

主要收集的输出格式和第三行相似。如果使用了-XX:+PrintGCDetail 选项，就会将垃圾收集时的详细信息打印出来。同样，对于不同的 J2SE 平台版本，根据虚拟机的需要，-XX:+PrintGCDetail 的输出格式也有所不同。下面就是在 J2SE 平台 1.4.2 版本上使用-XX:+PrintGCDetail 参数的输出示例：

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

从上面的输出信息可以看出，次要收集释放了新生代大约 98%的空间，

DefNew: 64575K->959K(64576K)

用了大约 46 毫秒，

0.0457646 secs

整个堆空间的使用率下降到 51%左右

196016K->133633K(261184K)

最后看到，相对于新生代的收集，在时间消耗上稍微多了一点点，

0.0459067 secs

选项-XX:+PrintGCTimeStamps 在输出垃圾收集信息的时候带有时间戳。

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]

收集在 111 秒的时候开始，几乎同时次要收集也开始了。对于新生代上的主要收集也有了一些附加信息，新生代的使用率下降到 10%

18154K->2311K(24576K)

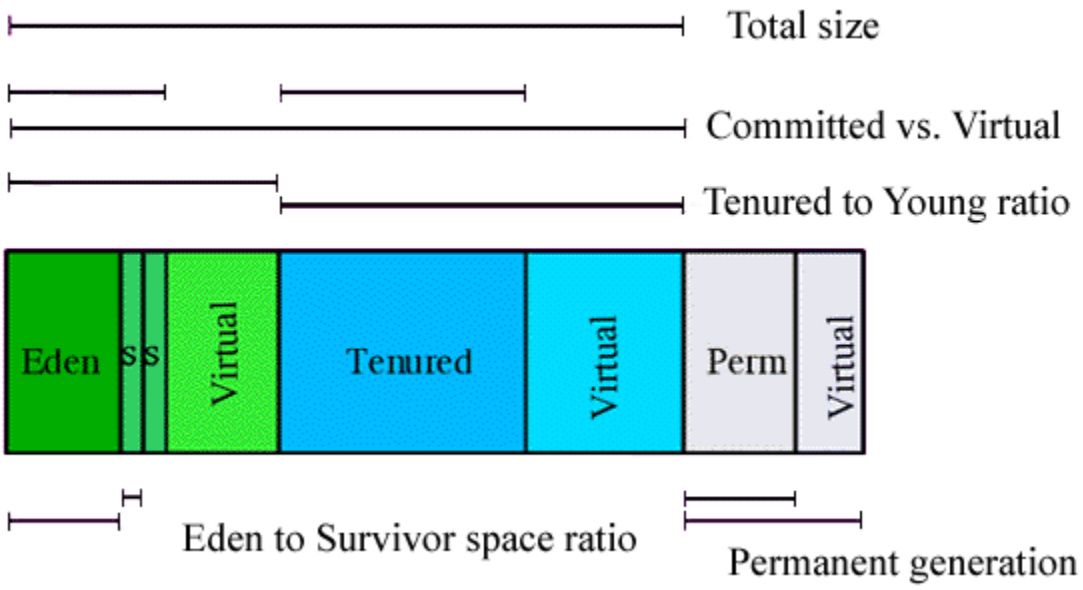
用了大约 0.13 秒

0.1293306 secs

3. 调整代大小

很多参数都能够影响代的大小，下图表示的就是*约定空间*（committed space）和*虚拟空间*（virtual space）之间的不同。在虚拟机初始化的时候，整个空间都为堆保留。可以通过-Xmx 参数指定整个空间的大小，如果-Xms 参数的值比-Xmx 小，那么在一开始，堆所保留的空间可能到不了-Xmx 参数指定的值，那么未保留的这部分空间就被表示为虚拟的。在需要更大的空间的时候，堆的不同部分（新生代，旧生代，持久代）都可能增长直到其上限。

有些参数指定的是堆中一部分空间和另外一部分的大小比率，例如，NewRatio 参数表示旧生代和新生代的比率。稍后会有关于这些参数的讨论。



3.1.堆的总体大小

当某个代被充满的时候，就会发生垃圾收集，所以，吞吐量和可用内存的大小成反比，总内存大小是影响垃圾收集的最重要因素。

默认情况下，虚拟机总是试图收缩内存大小，来保证每个代内的活动对象所占用的空间的比率维持在特定范围，这个范围就是由参数-XX:MinHeapFreeRatio=<最小值>和-XX:MaxHeapFreeRatio=<最大值>以及-Xms 和-Xmx 来确定。在 Solaris 操作系统平台（SPARC 版本）如下表所示：

-XX:MinHeapFreeRatio=	40
-XX:MaxHeapFreeRatio=	70
-Xms	3670k
-Xmx	64m

如果使用默认值，那么在某个代中，当空闲空间低于 40%，那么虚拟机就会增长这个空间以使得空闲空间能够保持在 40%以上，假设这个代的大小未超过最大限制。类似的，如果代中空闲空间的大小超过了 70%，虚拟机就会收缩这个代的大小，使空闲比率降到 70%以下，当然了，要高于 40%。

根据经验，这些默认的参数设置不适和大型的服务器应用，一个问题就是启动缓慢，因为堆的初始大小比较小，不得不进行多次的主要收集来释放空间，另外一个问题就是-Xmx 参数的默认设置对于这些服务器应用来说是很不合理的，所以，对于服务器应用，有这样的指导规则：

- 除非在垃圾收集的时候暂停现象严重，否则要尽可能多的给虚拟机分配内存，64M 通常都太小了
- 设置-Xms 和-Xmx 一样大，以避免虚拟机在收缩大小时的消耗。另外，如果你做出了糟糕的选择，虚拟机无法补偿
- 当增加了处理器之后，要相应的增加虚拟机内存大小，因为内存分配是可以并行进行的

关于虚拟机的全部参数的解释，可以参考：
http://java.sun.com/docs/hotspot/VMOptions.html

3.2.新生代

第二个重要的影响因素是虚拟机中新生代的大小。新生代越大，次要回收发生的次数就越少。当然，对于一定大小的堆来说，新生代越大，就意味着旧生代越小，那么就会使得主要回收的次数增多。最佳选择应该参考应用创建的对象的存活周期来设定。

默认情况下，新生代的大小是由 NewRatio 参数来设置的，例如设置-XX:NewRatio=3 表示新生代和旧生代之间的比率为 1:3，就是说 Eden 和存活空间总共占堆大小的 1/4。

参数 NewSize 和 MaxNewSize 指定了新生代大小的范围，设定这个参数就意味着限定了新生代的大小，就像使用-Xms 和-Xmx 参数限制堆的大小一样。最好将新生代的大小设定为经过 NewRatio 参数计算出的大小的整倍数。

3.3.新生代的保证

理想的次要收集会将活动对象从新生代的一部分（Eden 和第一个存活空间）复制到另外一部分（第二个存活空间）。但是，无法保证第二个存活空间能够容纳全部活动对象。所以，在旧生代一定要有足够的空闲空间来容纳全部的活动对象。最糟糕的情况就是在旧生代的空闲空间为 Eden 大小加上非空存活空间的大小。当旧生代也没有足够的空闲空间的时候，就需要进行一次主要收集。对于小型应用来说，这种策略是可以满足需要的，因为在旧生代的内存主要是虚拟约定的，并未真正使用。但是对于需要很大堆内存的服务器应用来说，大于堆虚拟约定内存一半的 Eden 大小是没有任何意义的：只会发生主要收集。需要指出的是，在新生代能够使用除“吞吐收集器”（throughput collector）之外的其他类型的收集器。如果旧生代无法容纳新生代复制过来的活动对象，吞吐收集器会在这个两个代上都进行垃圾收集。

如果有特殊需要，使用参数 SurvivorRatio 可以调整存活空间的大小，但是通常这个参数对性能的影响不那么重要。例如，-XX:SurvivorRatio=6 就设定了每个存活空间和 Eden 的比率为 1 :6，换句话说，就是每个存活空间占新生代大小的 1/8（不是 1/7，因为有两个存活空间）。

如果存活空间设置的太小，就会导致过于频繁的向旧生代复制对象；如果存活空间设置的太大，就会因为空闲而白白浪费。虚拟机的垃圾收集器都会为对象在复制到旧生代之前的反转复制次数选择一个阀值，通过设定这个阀值来保证存活空间一直处于半满状态。通过指定-XX:+PrintTenuringDistribution 参数可以观察到这个阀值，以及新生代中对象的“年龄”。同时，这个参数对于观察应用产生对象的存活期分布是很有帮助的。

下面是 Solaris 操作系统平台（SPARC 版本）上一些参数的默认值：

NewRatio	2（client 虚拟机是 8）
NewSize	2228k
MaxNewSize	无限
SurvivorRatio	32

新生代的最大大小是根据堆的总大小以及 NewRatio 来计算出来的，默认的“无限”表示：除非在命令行中指定了 MaxNewSize，否则该计算出来的值不限制 MaxNewSize 的大小。

对于服务器应用，有这样的指导规则：

- 首先确定能够给虚拟机使用的最大内存，然后通过调整新生代的大小来找到满足性能需求的最佳值
- 除非发现性能瓶颈在于频繁的主要收集或者暂停时间，尽可能多的给新生代分配内存
- 将新生代大小设置到接近堆大小的一半会适得其反
- 当增加了处理器之后，要相应的增加新生代大小，因为内存分配是可以并行进行的

4. 收集器类型

到目前为止，我们还只是针对默认的垃圾收集器展开讨论。从 J2SE 平台 1.4.2 版本开始，加入了另外 3 种垃圾收集器，都是着重提高吞吐能力，降低垃圾收集时的暂停时间。

1. 吞吐收集器（throughput collector）：命令行参数：-XX:+UseParallelGC。在新生代使用并行收集策略，在旧生代和默认收集器相同。
2. 并发收集器（concurrent low pause collector）：命令行参数：-XX:+UseConcMarkSweepGC。在旧生代使用并发收集策略，大部分收集工作都是和应用并发进行的，在进行收集的时候，应用的暂停时间很短。如果综合使用-XX:+UseParNewGC 和-XX:+UseConcMarkSweepGC，那么在新生代上使用并行的收集策略。
3. 增量收集器（incremental low pause collector）：命令行参数：-Xincgc。使用增量收集器要谨慎，他只是在每次进行次要收集的时候对旧生代进行一部分的收集，这样就把主要收集所带来的较长时间的停顿分散到多次的次要收集。但是，考虑到总共的吞吐，可能比旧生代上默认的收集还要慢。

注意，-XX:+UseParallelGC 和 XX:+UseConcMarkSweepGC 不能同时使用。对于 J2SE1.4.2 版本会检查垃圾收集相关参数组合的合法性，但是对于之前的版本没有这个检查，可能会导致不可预知的错误。

在尝试使用其他的收集器之前，建议先使用默认的收集器，通过调整代的大小以及相关参数来看看哪些方面不能满足需求，通过以上信息提供的参考，再决定选择其他的收集器。

4.1.何时使用吞吐收集器

当你的应用运行在多个处理器的主机上时，考虑使用吞吐收集器。因为默认的收集器是由一个线程来完成收集工作的，因此会给应用增加串行执行时间。吞吐收集器是多线程的进行次要收集的，所以可以降低应用的串行执行时间。一种典型的情况就是应用中有很多线程都在创建对象。这种情况下也需要增大新生代的大小。

4.2.吞吐收集器

吞吐收集器和默认的收集器类似，都是分代收集器。不同之处就在于吞吐收集器用多线程进行次要收集，主要收集本质上和默认收集器相同。默认情况下，在 N 个 CPU 的主机上，就会启动 N 个收集线程，收集线程的数量可以通过命令行选项进行控制。在只有 1 颗 CPU 的主机上，吞吐收集器的性能表现可能还不如默认收集器，因为一些并行执行（例如进行同步时的开销）带来了额外的开销；在 2 颗 CPU 的主机上，吞吐收集器和默认收集器性能相当；在多于 2 颗 CPU 的主机上，你会发现进行次要收集的暂停时间降低了。

通过命令行参数-XX:+UseParallelGC 来指定使用吞吐收集器，ParallelGCThreads 参数用来设置线程数量（-XX:ParallelGCThreads=<具体数值>）。吞吐收集器对堆内存的需求和默认收集器一样。使用吞吐收集器能够降低在新生代上进行次要收集时的暂停时间，但是由于有多个线程参与次要收集，在从新生代向旧生代提升（promotion）的时候可能会产生碎片。为了进行对象从新生代到旧生代的提升，每个垃圾收集线程都会保留一块旧生代的空间，旧生代就被划分成多个“提升缓冲”（promotion buffers），这样就容易产生内存碎片。减少收集线程的数量能够减少碎片的产生，同时会增加旧生代的大小。

4.2.1.大小自适应

从 J2SE 平台 1.4.1 版本开始，吞吐收集器就具有了一个特征，就是大小自适应（参数-XX:+UseAdaptiveSizePolicy），这个选项默认是打开的。该特征对于收集时间、分配比例、收集之后堆的空闲空间等数据进行统计分析，然后以此为依据调整新生代和旧生代的大小以达到最佳效果。可以使用-verbose:gc 来查看堆的大小。

4.2.2.Aggressive 堆

-XX:+AggressiveHeap 选项会检测主机的资源（内存大小、处理器数量），然后调整相关的参数，使得长时间运行的、内存申请密集的任务能够以最佳状态运行。该选项最初是为拥有大量内存和很多处理器的主机而设计的，但是从 J2SE1.4.1 以及其后继版本来看，即使是对于那些只有 4 颗 CPU 的主机，该选项都是很有帮助的。因此，吞吐收集器（-XX:+UseParallelGC）、大小自适应（-XX:+UseAdaptiveSizePolicy）以及本选项（-XX:+AggressiveHeap）经常结合在一起使用。要使用本选项，主机上至少要有 256M 的物理内存，堆内存的最初大小是基于物理内存计算出来的，然后会根据需要尽可能的利用物理内存。

4.2.3.吞吐收集器的测量法

-verbose:gc 的使用和默认收集器相同。

4.3.何时使用并发收集器

如果你的应用在运行的时候能够拥有足够的处理器资源，并且减小垃圾收集的暂停时间能够明显提升应用性能，此时考虑使用并发收集器。例如那些有大量相对长生存期的对象（较大的旧生代）的应用，并且运行在有 2 颗甚至更多的 CPU 的主机上。但是实际上这种收集器是为那些需要短暂停时间的应用所设计的。运行在单处理器上的交互式应用，如果设置一个合适的旧生代大小，能够达到非常良好的效果。

4.4.并发收集器

并发收集器和默认收集器类似，都是分代收集器。并发收集器在旧生代上并发进行垃圾收集。

并发收集器旨在降低旧生代上进行收集的暂停时间，他使用一组互相隔离的收集线程，每个线程负责一部分的主要收集，这个收集过程和应用的运行是并发进行的。通过命令行选项-XX:+UseConcMarkSweepGC 来使用并发收集器。每当发生主要收集的时候，并发收集器在收集开始的时候和中期会短时间的暂停所有的应用线程，中期的暂停时间相对长一点，在这次暂停中，多个线程同时工作完成收集任务。剩余的收集工作将由一个收集线程完成，这次是和应用并发执行的。次要收集的过程和默认收集器类似，也可以使用多线程的方式完成次要收集，参看“并发收集器的并行次要收集”章节。

在下面连接中对并发收集器（针对旧生代）中用到的技术进行了详细的阐述：
http://research.sun.com/techrep/2000/abstract-88.html

4.4.1.并发的额外开销

虽然并发收集器有效的降低了收集时的暂停时间，但却是以使用更多的处理器资源为代价的。收集过程中并发进行的那一部分是由单一的一个线程完成的。对于一个在 N 个处理器上运行的系统，并发进行的那部分会使用 1/N 的处理器资源。可能在单处理器的系统上你看到他表现也还可以，那只是偶然罢了。当然，它能够将一次长时间的暂停（这里的暂停指所有的应用线程都不可用了）分割成多个短时间的暂停，但是这个并不是它的设计初衷。同时，并发收集会有额外的开销，并且可能降低系统吞吐能力，同时对于某些类型的应用来说，并发收集是有先天缺陷的（例如容易造成内存碎片）。在拥有 2 颗处理器的系统上，并发收集在执行的时候，还有 1 颗处理器可以为应用服务，因此，执行收集不会“暂停”应用的运行。虽然降低了暂停时间，但是并发收集确实占用了一部分处理器资源，所以你可能感觉到应用会有所缓慢。N 的值越大，在并发收集上所用的处理器资源就越少，并发收集器的优势就越明显。

4.4.2.新生代的保证

如果使用默认收集器，那么在进行次要收集的时候，必须保证旧生代中有足够的空间来容纳从 Eden 和存活空间复制过去的对象。由于并发收集会产生内存碎片，所以这个保证的条件就更加苛刻：在旧生代必须要有足够的连续空间来容纳来自 Eden 和一个存活空间的对象，因为没有什么方式能够准确知道 Eden 和这个存活空间中对象大小的分布情况（主要是为了避免巨大的性能消耗）。相对于默认收集器，并发收集器往往需要更大的堆内存。在默认收集器时，堆内存需要保留但是不一定真正的使用。先使用默认收集器，找到一个新生代和旧生代大小的合适的估算值，然后将旧生代的大小设置成和新生代一样大再去使用并发收集器。这只是一个非常粗略的近似值，至于实际上的最佳设置是由应用来决定的。

4.4.3.完全收集

在旧生代被填满之前，并发收集器使用一个收集线程来完成收集工作，同时不暂停应用的执行。实际上，即使应用的线程都在运行，收集器都可以并发来做更多的工作，所以，对于应用而言，只会感觉到非常短暂的暂停。但是当旧生代在填满之前如果收集工作无法全部完成，那么就会暂停应用的线程来完成所有的收集工作。这就是我们所说的完全收集（full collections）。如果发现完全收集比较频繁，可能需要调整并发收集的相关参数。

4.4.4.漂浮垃圾

垃圾收集的工作就是查找堆中的所有活动对象。当应用的线程和垃圾收集的线程并发执行的时候，那么对于收集线程来说当时是活动的对象可能在收集工作完成之后就变成了非活动对象。这就是所说的“漂浮垃圾”（floating garbage）。漂浮垃圾的数量和并发收集的时间有关（应用线程需要花一些时间才丢弃对象），和应用的细节也有关系。可以通过增加旧生代 20%（这个是估计值）解决漂浮垃圾问题。当然了，在下一轮收集的时候，这些垃圾都将被收集掉。

4.4.5.暂停

在一次并发收集周期，需要两次暂停应用。第一次暂停时，标识所有的可以从根对象（例如线程栈，静态对象等）或者堆中的其它对象（例如新生代）直接到达的对象，这就是“初始标识”（initial mark）。紧接着就是第二次暂停，此次暂停的目的是为了找出由于收集和应用的并发执行而疏漏的、未被标识对象，这叫做“重新标识”（remark）。

4.4.6.并发阶段

在初始标识和重新标识之间有一个并发标识的过程，在并发标识的时候，收集线程需要占用一部分本来属于应用的处理器资源。在重新标识阶段之后，还有一个并发清理过程，同样，也会占用一部分处理器资源。在清理阶段之后，并发收集线程进入休眠状态，直到下一轮主要收集的发生。

4.4.7.并发收集的测量

下面是使用了-XX:+PrintGCDetails 参数的-verbose:gc 输出（已经删除了一些详细信息），我们可以看到并发收集的输出中穿插了很多次要收集，一般来说，一个并发收集周期中，会有多次的次要收集。CMS-initial-mark 表示并发收集周期的开始，CMS-concurrent-mark 表示并发标识阶段的结束，CMS-concurrent-

sweep 表示并发清理阶段的结束。我们之前没有讨论的预清理阶段（precleaning phase）由 CMS-concurrent-preclean 标识，它表示的是可以并发执行的一些工作并且是为重新标识阶段（CMS-remark）做好了准备。最后一个阶段由 CMS-concurrent-reset 标识，表示已经为下一轮的并发收集做好了准备。

```
[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]
```

初始标识的暂停相对于次要收集的暂停要短，反之，并发阶段（并发标识，并发预清理，并发清理）的暂停可能相对要长一些，但是在收集的过程中，应用不会有任何暂停。而由于标识所引起的暂停和应用的细节（例如频繁的修改对象会增加暂停时间）以及上一次次要收集的时间（例如新生代中有大量的对象时也会增加暂停时间）有关。

4.4.8.并发收集器的并行次要收集

在多处理器平台上，可以使用参数 UseParNewGC 来降低次要收集的暂停时间：

-XX:+UseParNewGC

如果使用了 UseParNewGC，那么同时使用 CMSParallelRemarkEnabled 参数可以降低标识暂停：

-XX:+CMSParallelRemarkEnabled

4.5.何时使用增量收集器

如果你的应用可以用较频繁的、较长时间的新生代上收集来换取稍短时间的旧生代收集，就可以考虑使用增量收集器。典型情况是如果需要长时间的旧生代收集时（大量的长存活期对象），小型的新生代收集也能够满足（大部分对象是短存活期的），并且只有一个处理器。

4.6.增量收集器

同样，增量收集器也是和默认收集器类似的分代收及器，在新生代上的次要收集和默认收集器一样。不要在使用增量收集器的同时使用-XX:+UseParallelGC 或者-XX:+UseParNewGC。在旧生代上的主要收集是增量完成的。

这种收集器在每次做次要收集的时候，进行一部分的主要收集，这样就避免了做完整的主要收集所带来的长时间暂停。但是有时候为了避免出现内存溢出（out of memory）的问题，也会在旧生代上进行完整的主要收集（就象默认收集器那样）。

由于这种收集器会在堆内存上产生碎片，所以相对于默认的标识-清理-压缩（mark-sweep-compact）收集器来说，可能需要更大一些的堆内存。

为了能够在每次次要收集时进行一部分的主要收集，收集器需要维护一些附加信息，所以，增量收集的总体消耗要高一些，并且吞吐可能不如默认收集器那么好。

首先使用默认收集器找到一个合适的堆大小，如果此时主要收集的暂停时间还是无法满足应用需求，尝试调整各个代的大小，并且使用增量收集器，直到找到合适的堆设置。

- 如果在使用增量收集器的时候发生了完全收集（full collection），可能在旧生代发生内存溢出之前无法完成增量的垃圾收集，这个时候，你需要减小新生代的大小，以迫使次要收集发生频率更高一些。
- 如果由于无法满足新生代保证而发生的主要收集，那么会产生内存碎片。一次次要收集没有能够回收任何空间，此时表明无法保证新生代需求了，此时尝试增大旧生代大小来弥补碎片问题，可能不会真正使用很大的旧生代，但是对于新生代保证来说是有帮助的。

4.6.1.增量收集器测量

将-verbose:gc 和-XX:+PrintGCDetail 组合使用，可以看到如下的输出：

[GC [DefNew: 2074K->25K(2112K), 0.0050065 secs][Train: 1676K->1633K(63424K), 0.0082112 secs] 3750K->1659K(65536K), 0.0138017 secs]

从上面的输出可以看出，进行次要收集用时大约 5 毫秒，同时还有一次增量收集（Train:…），用时大约 8 毫秒。如果发生了完全收集，在输出中会看到 Train:MSC 字样：

[GC [DefNew: 2049K->2049K(2112K), 0.0003304 secs][Train MSC: 61809K->357K(63424K), 0.3956982 secs] 63859K->394K(65536K), 0.3987650 secs]

同时，从上面的输出中可以看出，次要收集并没有起到作用：收集前后都是 2049K，这就表明了在旧生代上没有连续的空间能够满足新生代保证。

5. 其他考虑事项

对于大部分应用来说，持久代的大小不会影响垃圾收集的性能。但是有些应用会动态的产生或者加载大量的对象，例如 JSP 的容器，如果需要，可以使用参数 ManPermSize 来增加持久代的大小。

有些应用的 finalization 或者弱引用/软引用/幻影引用（weak/soft/phantom refrences）和垃圾收集相互影响。这种特点可能从 Java 语言本身就造成了很差的垃圾收集性能，一个典型的例子就是依赖对象的 finalize 方法来释放资源，比如关闭文件描述符（file descriptor），这样就极大地影响了垃圾收集的性能。无论如何，依赖垃圾收集来释放资源都是非常不可取的方式。

应用影响垃圾收集器的另外一种方式就是显式的调用垃圾回收，例如调用 System.gc()方法。这个调用强制进行主要收集，对于大型应用的可扩展性有很大的伤害。可以使用参数-XX:+DisableExplicitGC 来禁止应用显式的调用垃圾收集。

另外就是在 RMI 分布式垃圾收集（RMI distributed garbage collection, DGC）时经常遇到使用显式的垃圾收集，应用通过使用 RMI 引用在另外一个 Java 虚拟机中的对象，在这种分布式应用中，垃圾对象无法通过传统的垃圾收集进行清理，所以 RMI 强制进行周期性的垃圾收集。可以通过一些属性来设置收集周期：

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

默认的收集周期是 1 分钟，上面的参数指定的周期为 1 小时。但是，这样设置可能会使得某些对象要经过很长时间才能够被回收。如果对于 DGC 没有时间上的需求限制，可以设置为 Long.MAX_VALUE。

Solaris 8 操作系统平台使用另外一个版本的线程库（libthread）能够将线程绑定为轻量进程（light-weight process, LWP），对于一些应用来说可能这个版本的线程库比较有益处。要使用这个线程库，在启动 Java 虚拟机时在环境变量 LD_LIBRARY_PATH 中包含/usr/lib/lwp。在 Solaris 9 上，这个线程库是默认的。

相对于客户端模式的虚拟机（-client 选项），当使用服务器模式的虚拟机时（-server 选项），对于软引用（soft reference）的清理力度要稍微差一些。可以通过增大-XX:SoftRefLRUPolicyMSPerMB=1000 来降低收集频率。默认值是 1000，也就是说每秒一兆字节。

6. 总结

根据应用的需求不同，垃圾收集可能会成为性能的瓶颈。如果充分了解应用的需求，并且深入理解垃圾收集的机制以及相关选项，能够将垃圾收集对性能的影响降至最小。

7. 其他文档

7.1.输出示例

GC 输出示例列出了不同类型的垃圾收集的行为，以及对于垃圾收集详细信息的诊断，并描述了如何来分析问题。

http://java.sun.com/docs/hotspot/gc1.4.2/example.html

7.2.常见问题

对于常见问题的一些解答，比本文档要详细一些。

http://java.sun.com/docs/hotspot/gc1.4.2/faq.html

原文出处：http://java.sun.com/docs/hotspot/gc1.4.2/

关于译者：J.Y. 经常出没于 dev2dev.bea.com.cn（id: jadeyuan），工作之余喜欢翻译一些技术文档，但是属于业余级选手。