HotSpot垃圾回收算法概述

引言

在内存管理中，垃圾是指那些不再被使用的对象。对于一个垃圾回收器（Garbage Collector），它需要完成三件事：

1. 分配内存：垃圾回收算法的设计往往制约了内存分配的方式；
2. 确保存活对象不会被回收
3. 回收垃圾对象

很显然的是，一个垃圾回收器并不提供回收全部垃圾对象的保证。这意味着一次垃圾回收结束之后，内存中依然可能存活着不会再被使用的对象。

对于HotSpot中的垃圾回收算法而言，其设计都是建立在两个经验法则之上的：
1. 对于大部分对象来说，它会在“年轻”的时候死去；
2. 很少引用存在于“年老的”和“年轻的”对象之间；

这里使用的“年轻的”和“年老的”的两种说法，是指存活时间的长短。这两条的经验法则表明，大部分的对象存活时间都很短，无法活过一个回收周期。并且，存活时间长的对象倾向于引用存活时间长的对象，而存活时间短的对象倾向于引用存活时间短的对象。对于大部分的应用来说，这两个经验法则都是适用的。

设计因素

在垃圾算法回收的设计中，需要考虑的因素有：
1. 串行（Serial）和并行（Parallel）
2. 并发（Concurrent）和暂停式（Stop the world）
3. 压缩（Compacting）和非压缩（Non-Compacting）：这一组概念是指，在垃圾回收结束之后，是否需要把所有的存活对象挪到一起，占据一个连续空间。在Compacting之后，也意味着可用内存占据了一个连续的空间，这个时候就可以使用bump-the-pointer的分配内存技术。在这种技术中，只需要持有一个指针指向已分配内存的尾部。每次分配的时候只需要检查剩余空闲空间能否容纳新的对象，而后分配内存并且将指针指向新的尾部。这种Compacting的计数，在一些算法中需要付出额外的代价，这个代价要么是需要额外的内存空间，要么是额外的回收时间。

并发和并行是一对比较容易搞混的概念。并发是指，垃圾回收和应用可以在一段时间内同时运行，这个概念和操作系统上的概念是一致的。在HotSpot中，并发垃圾回收算法中大部分垃圾回收的工作都是在并发的情况下完成的，但是并不能完全免除Stop-the-world。并行是指利用多个CPU，多个线程同时进行垃圾回收。

度量

在衡量一个垃圾回收算法上，最为主要的两个度量是：
1. 暂停时间（pause time）：是指在一次垃圾回收中，Stop-the-world状态下占用的时间。暂停时间主要受到算法和堆大小的影响。相同条件下，堆越小，暂停时间就越短。但是堆越小，那么回收频率就越高。
2. 吞吐量（throughput）：一般而言，堆越大，吞吐量越高，回收频率越低。

可以看到一个有趣的地方：暂停时间和吞吐量对堆的大小要求是不一样的。暂停时间要想短，那么应该有更小堆；而吞吐量要大，需要更大的堆。

分代

前面提到，大部分的对象都会在年轻的时候死去。因此将那些年轻对象放在一起，将那些年老的对象放在一起，在垃圾回收的时候区别对待，就是很有价值的。这就是分代（generation）的想法。在HotSpot中，所有的垃圾回收算法都是分代垃圾回收算法。年轻代垃圾回收更加频繁，并且在一次回收中能够存活下来的对象也是及其稀少的。在年轻代存活过一段时间（通常是活过了几次年轻代的垃圾回收周期）后，对象会被提升（promoted or tenured）到老年代。如图：

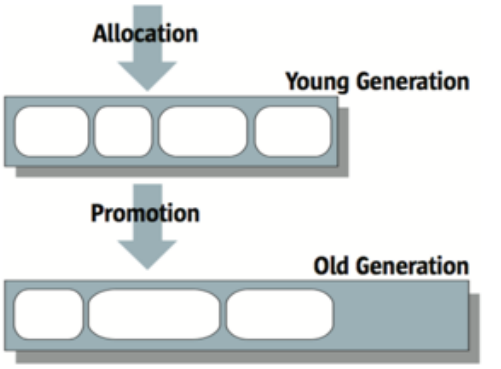

Figure 1. Generational garbage collection

注：图片引自Memory Managerment in the Java HotSpot Virtual Machine

除此以外，大对象也会直接分配在老年代。总体来说，老年代的对象存活时间更长，并且老年代分配出去的空间增长缓慢，由此导致了老年代垃圾回收不频繁。

这种提升策略会有一个问题，就是老年代可能并没有足够的空间容纳这些从年轻代提升上来的对象。在这种情况下，会触发一次老年代的垃圾回收。在极端的情况下，整个年轻代的所有对象都会被提升到老年代。因此，老年代比年轻代要大是顺理成章的事情。

> *关于老年代什么时候检测剩余空闲空间，判断能否容纳被提升的对象，我阅读的文档文章中并没有明确提出。它可能发生在每提升一个对象，就执行一次检测；也可能是在年轻代回收之前，根据有多少待提升对象进行一次检测；最后一种可能是，检测发生在年轻代垃圾回收之前——也就是意味着每次检测都是拿年轻代的大小和老年代空闲空间的大小进行比较。在Tuning Garbaga Collection with the 5.0 Java Virtual Machine中有一段话，“But for applications needing the largest possible heap, an eden bigger than half the virtually committed size of the heap is useless: only major collections would occur”。从这句话来看，老年代似乎并不管究竟有多少需要提升，也不管会回收多少垃圾，每一次比较都是用空闲空间和整个年轻代的大小进行比较。*

在HotSpot中的分代为：
1. 年轻代（young generation）分成Eden和Survivor。（注：在一些特殊的平台上，HotSpot并没有Eden和Survivor的概念）
1. Eden：年轻代只有一个Eden。对象分配都是直接分配在Eden中的；
2. Survivor：年轻代中会有两个Survivor，拿来存放在一次垃圾回收算法中存活的对象。每次算法运行的时候，会把Eden和一个Survivor（标记为from）中的存活对象复制到另外一个Survivor（标记为to）中；
2. 老年代（old generation）：
3. 持久代（perm generation）：持久代几乎不会出现垃圾回收（有些应用或者有些平台会有这种需求），本文将不会涉及这个地方；

在HotSpot中，Eden和Survivor的比例可以通过使用++NewRatio来

Serial Collector

原理

Serial Collector（串行回收器）在年轻代是一个使用标记-复制算法的回收器。标记-复制垃圾回收算法可以分成两个阶段：

1. 标记阶段：从根（root）出发，沿着引用链标记存活对象；
2. 复制：将存活对象复制到特定的区域；
   如图：

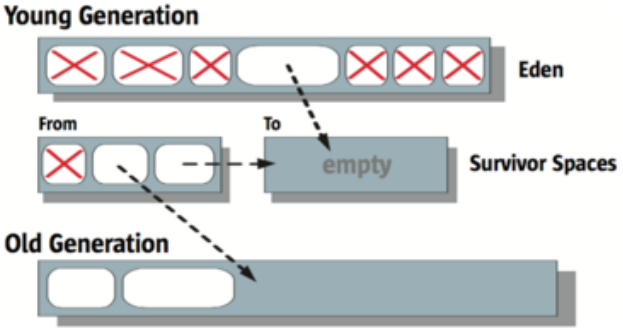

Figure 3. Serial young generation collection

GC完成后如图：

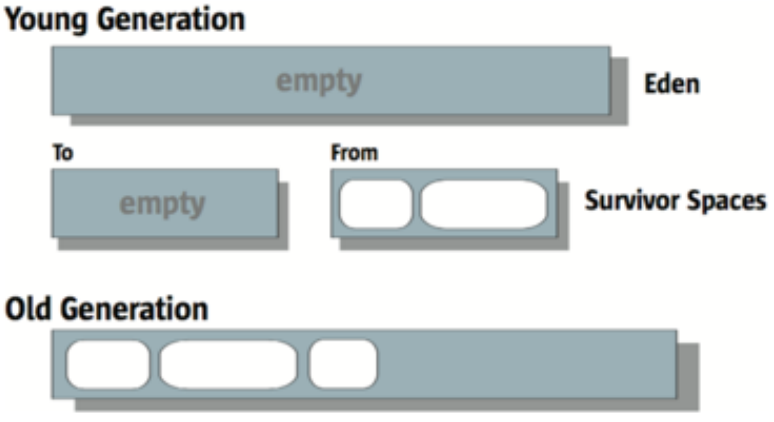

Figure 4. After a young generation collection

注：图片引自Memory Managerment in the Java HotSpot Virtual Machine

在标记阶段，Serial Collector将Eden中和这次被标记为from（恰好是上次标记为to）的Survivor中的存活对象标记出来，而后将存活对象复制到另外一个Survivor中。在这个过程中，可能有部分对象的存活时间达到了提升到老年代的标准，因此会有一些对象被复制到老年代。

Serial Collector也可以被应用在老年代，所不同的是，老年代并没有什么Eden和Survivor的划分。在老年代使用的是标记-清扫-整理算法。该算法流程为：

1. 标记：比较存活的对象
2. 清扫：在该阶段，会识别出垃圾对象。“清扫”这个概念带有一些误导的色彩，在算法的实现中，并不需要真的对垃圾对象占据的内存进行清扫，仅仅标记一下就够了。（注：在新对象被分配到该被清扫的区域的时候，会执行一次JVM层面上的初始化过程，该过程会把该内存区域重置，因而有了Java语言中的不同数据类型的默认值一说）
3. 整理：将所有的存活对象都挪到一端

可以看到的是，在使用Serial Collector的情况下，无论是老年代还是年轻代，其内存都经过了Compacting，所有的存活对象占据了一块连续的空间，而空闲空间也是占据了一个连续的空间。因此在分配内存空间的时候都可以使用bump-the-pointer的技术。

使用场景

Serial Collector主要适用于对pause time要求不高，可用内存和CPU数量都比较小的应用上。在新的HotSpot中，如果虚拟机运行的平台是client-style类型的，那么就会采用Serial Collector。

影响因素和JVM选项

Serial Collector的性能和几个因素有关：
1. 堆大小。如前面所言，堆的大小直接影响了吞吐量和停顿时间（pause time）
2. NewRatio：这个因素调节的是年轻代和老年代的比例。比如说值是3，那么意味着年轻代和老年代的比例是1：3。对于Serial Collector来说，年轻代过大，会造成相应老年代过小，这会导致更加频繁地触发老年代的垃圾回收（也即是major GC）。而如果年轻代过小，那么年轻代就会更加频繁GC，并且相对而言，更加多的大对象会被直接分配到老年代。在这种情况下，触发老年代GC的频率要降低，但是pause time就要提升；
3. SurvivorRatio：该选项调节的是年轻代中Suvivor部分占据的大小。举例来说，如果值是8，那么意味着，每个Survivor占据的大小是1/(8+2)。Survivor的大小是比较关键的。如果Survivor比较小，那么Survivor很容易就装不下存活对象，因此有更加多的存活对象被逼提升到老年代。这会带来两个影响：一个是老年代更加快装满，另外一个是老年代指向年轻代的引用会增加——这个因素会影响mark阶段根的大小。而如果Survivor过大，那么每次装完存活对象之后还会剩下一段没有利用的空间，这段空间就会被浪费，影响年轻代的GC频率和吞吐量；

还有其余的一些选项，如NewSize和MaxNewSize等，读者可以自己去找相关的资料阅读。

Parallel Collector

原理

Parallel Collector（并行垃圾回收器）和Serial Collector（串行回收器）比起来，要复杂微妙很多。总体的算法思想是一致的，不过Serial Collector在任何阶段都是单线程在运行的，而Parallel Collector则是多线程运行的。额外要注意的是，即便在虚拟机中指定了Parallel Collector，但是老年代的回收，也就是major collection还是使用和Serail Collector一样的单线程！换句话来说，只有对年轻代的回收（minor collection）将会采用多线程。如图：

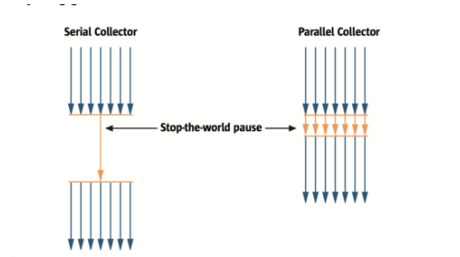

Figure 6. Comparison between serial and parallel young generation collection

注：图片来自Memory Managerment in the Java HotSpot Virtual Machine。

Parallel Collector与Serial Collector比起来，以下这些方面是要注意的：
1. 工作分配。这是一个算法设计的难点。举例来说，我们可以将整个标记阶段看成是一个大的任务，由一系列的小的工作组成，那么在多线程的情况下，要考虑哪个线程负责标记哪一块，还要考虑负载均衡——即任务的分配要公平，不能一些线程很快完成任务停下来，等待其余线程完成自己的工作；

> *在标记阶段，还有一个很有意思的话题。就是如果多线程在进行标记的时候，可能会重复标记一个对象为存活对象，这并不会影响算法的正确性，只是会影响算法的性能。关于并行回收算法的更加细致的描述可以阅读《垃圾回收算法手册——自动内存管理的艺术》第14章。*

1. 在多线程的情况下，提升对象到老年代会遇到新的问题。按照原来的bump-the-point的算法，每一次分配都是指针的递增。假设说现在有两个线程同时在老年代分配空间，分配前的指针指向100，其中一个线程将指针增加到150，而另外一个线程将指针设置为200.那么最终的结果就可能是150，也可能是200。一种自然的想法是加锁，但是和一般应用里面锁争用会带来损耗一般，这也会导致老年代的分配损耗增加（这会极大影响pause time和吞吐量）。而Paraller Collector采用了另外一种被称为TLABs（thread-local allocation buffers）技术：它将老年代划分成一个个固定大小的Buffer，给每一个回收线程分配一个，回收线程就只在自己的Buffer内分配空间。在这种情况下，只有线程重新申请一个Buffer的时候，才会引入并发的问题。但是这会带来另外一个问题，就是每一个线程并不能恰好用完一个Buffer，可能出现的情况是，一个线程检测到Buffer里面的空闲空间已经不够了，于是只能申请另外一个Buffer。而原来的Buffer的那部分不足的空间就会被浪费掉。这就是所谓的float garbage。

还有一个误解要澄清。应用有多少个线程在运行并不决定有多少个线程回收。举例说一个应用有200个线程正在运行，但是在垃圾回收的时候，回收的线程可能只有10个。

Parallel Collector有一个变种叫做Parallel Compacting Collector。从名字上很令人困惑，因为从前面对Serial Collector上的叙述中可以看出来，major collection使用的算法，本身就是compacting的。我简单描述一下两者的区别：对于Parallel Collector来说，每一次回收都会Compacting整个老年代；而对于Parallel Compacting Collector来说并不是。它会检测一个区域的存活对象的密度，来决定是否进行compacting。更多相关信息可以阅读《Memory Managerment in the Java HotSpot Virtual Machine》和《垃圾回收算法手册——自动内存管理的艺术》。

使用场景

Parallel Collector又被叫做Throughput Collector。所以很显然，它适用于要求吞吐量高的场景。

影响因素和JVM选项

前面在Serial Collector中谈及的影响因素，对Parallel Collector同样是适用的，而且还额外收到以下这些因素的影响：
1. GC线程数量：线程数量和float garbage的数量是成正相关关系；
2. Buffer大小
3. CPU数量：在单个CPU的情况下，Parallel Collector的表现不如Serial Collector；在两个CPU的情况下，Parallel Collector可以达到和Serial Collector相当的表现，甚至会超过一点。在CPU超过两个的情况下， 表现都会比Serial Collector好。但是并不是CPU越多性能越好。在当前算法的实现中，CPU数量超过32（16？有点忘记了，阅读的资料也忘了是哪个了）性能反而会下降；

CMS Collector

原理

CMS Collector（Concurrent Mark-Sweep Collector），是一个并发垃圾回收器，这意味着在垃圾回收的过程中间，大部分时间里，应用还是可以继续运行的。所以，这里所谓的并发，更加多的是指应用和垃圾回收的并发。同时CMS Collector也是多线程的，也即它也是Parallel（并行）的。

在CMS Collector里，年轻代的回收是和Parallel Collector一样的，也就是说，年轻代的回收是stop-the-world式的。只有在老年代，相应的major collection里面才会使用CMS算法。
CMS的过程如图：

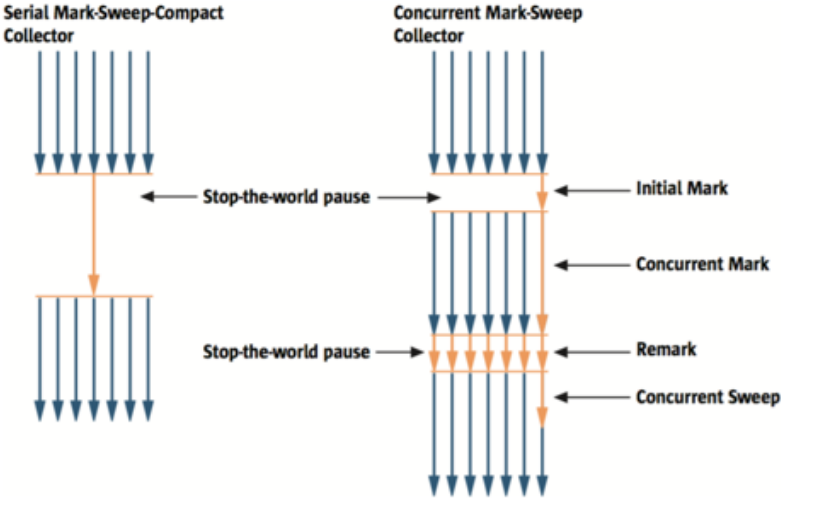

Figure 7. Comparison between serial and CMS old generation collection

注：图片来自Memory Managerment in the Java HotSpot Virtual Machine

1. initial mark：initial mark是stop-the-world式的，也就是说在这个阶段是需要暂停应用的执行。initial mark只是识别出来标记的根（准确说法是”identifies the initial set of live objects directly reachable from application code”， 另外在StackOverFlow上看到一段话”This includes: Reference from thread stacks, Reference from young space.”）
2. concurrent mark：并发标记阶段。在该阶段，应用可以继续运行；

3. remark：在concurrent mark阶段，因为应用依旧在运行，所以可能原本标记为垃圾的对象又“复活”了，也可能分配了新的对象。所以会引入找一个remark阶段。该阶段也是stop-the-world的；
4. concurrent sweep：并发清扫。该阶段应用会继续运行；

因为引入两个并发阶段，所以会造成很多问题：
1. 如何快速的remark？显然，如果要是在remark中还是需要扫描对象，那么该Collector就没多大必要存在了。CMS Collector采用了一种称为“card table”的技术。card table简单理解是并发回收器的工作列表。CMS使用该技术会在concurrent mark阶段，将改变了引用关系的对象标记为“dirty”，在remark阶段中重新扫描；
2. 还有一个问题是，在concurrent mark阶段，可能触发minor collection。CMS Collector会采用一种称为Mod Union Table的技术来记录GC前后card的信息。所以综合这两种情况，CMS Collector在remark阶段会从Mod Union Table和Card Table出发；
3. 因为在回收阶段还有可能分配对象，所以垃圾回收不能等内存满了才开始，必须要提前开始。但是CMS Collector并不提供在回收阶段一定含有足够的空闲给应用分配对象。这就会造成一个问题，就是在垃圾回收阶段，空闲空间不足了。这个时候，应用会被停下来，也就是进入到stop-the-world状态，直到回收完成；
4. 该回收的垃圾没有被回收。这也被称为floating garbage。这主要是出现在，原本一个对象被标记为存活，但是在concurrent阶段，应用修改了指向该对象的引用，使得它称为了垃圾。但是CMS Collector无法将其检测出来。因此它能够躲过这一轮的垃圾回收，直到下一次的回收周期；

CMS Collector不是compacting的，这意味着垃圾回收之后得到的空闲空间并不是连续的。这会带来一些问题：
1. 分配方式改变：前面提到的最多的分配方式就是bump-the-pointer。但是该分配方式只使用于连续的空闲空间。CMS采用了新的分配方式：空闲链表分配方式，该概念和操作系统中内存管理中的空闲链表是一样的；
2. 空闲空间不连续会导致空间有效利用率下降。比如说空闲空间总和是足以容纳分配对象的，但是因为不能容纳，所以反而会触发GC，或者会触发扩容。所以对于CMS Collector来说，它会要求更大的堆；
3. CMS Collector会合并相邻的空闲空间：这是一个优化，但是因为合并这个空闲空间需要操作空闲链表，而分配对象又需要操作空闲链表，所以在concurrent sweep会出现空闲链表的争用。CMS Collector使用了Mutual exclusion locks来保证JVM分配优先；

CMS Collector是允许整理空闲空间的，用户可以通过命令行选项UseCMSCompactAtFullCollection来指定。

CMS Collector还有一种模式，增量模式（Incremental Mode）。在该模式下，CMS Collector会使用少量的线程来并发标记或者并发清扫，整个过程会持续多个minor collection周期。该模式的好处是，可以降低pause time，并且减少对应用的影响。所付出的代价，就是需要更加大的堆。该模式一般适用于CPU数量较少的情况。

使用场景

适用于要求pause time尽可能短，并且拥有多个CPU的应用。CMS Collector的别名是Latency Collector。

影响因素和选项

在Serial Collector中谈及的因素对CMS Collector都会有影响，此外还受到：
1. 线程数量
2. CPU数量
3. CMSInitiatingOccupancyFraction：该选项设置了当被分配内存占据了多大比例的时候，就会触发major collection。若是设置的太小，那么会导致更加频繁的GC；但是设置得太大，就更有可能出现回收过程中空闲空间不足的现象，而这会导致应用被停下来，直到GC完成；
4. 是否使用增量模式

G1 Collector

原理

G1 Collector（Garbage-First Collector）可以被看做是CMS Collector的升级加强版。G1 Collector的算法流程和CMS类似，所不同的有：
1. G1 Collector采用的是标记-整理算法。这意味着每次算法结束得到的都是连续空间；
2. G1 Collector虽然还采用分代的方式，但是它的内存模型有了巨大的变化。它的内存基本结构被分成了一个个Region。G1 Collector维护了一个Region的列表，每次判断哪个Region的回收价值最大，便回收该Region。也就是说，G1 Collector回收并不是回收整个区域，而是分区域收集的；

其具体流程是：
1. initial marking phase：标记根，该阶段是stop-the-world式的；
2. root region scanning phase：该阶段标记Survivor Region中的指向老年代的引用，及其引用对象；
3. Concurrent marking phase：
4. Remark phase：
5. Cleanup phase：

所以CMS Collector因为并发引发的问题G1也同样存在。但是G1 Collector能够避开各种因为空闲空间不连续所导致的问题。

> *G1 Collector实现的算法是比较复杂的，详细内容可以阅读Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide和《垃圾回收算法手册-自动内存管理的艺术》10.3.1，11.8.6等*

G1 Collector有一个很重要的特性，就是“软实时”。G1 Collector可以让使用者指定在一个长度为M毫秒的时间段内，消耗在垃圾收集上的时间不得超过N毫秒。这已经是期望达到一种类似于实时垃圾回收的效果了。

> *所谓的实时垃圾回收，是指必须能够精确地控制由垃圾回收所导致的赋值器的中断。*

结语

强烈推荐阅读Oracle上关于HotSpot的相关文档，以及《垃圾回收算法手册-自动内存管理的艺术》。