回首页

HotSpot JVM 内存管理

创建时间: 2018-03-27 00:00:00

关于 JVM 内存管理或者说垃圾收集，大家可能看过很多的文章了，笔者准备给大家总结下。这算是系列的第一篇，接下来一段时间会持续更新。

本文主要是翻译《Memory Management in the Java HotSpot Virtual Machine》白皮书的前四章内容，这是 2006 的老文章了，当年发布这篇文章的还是 **Sun Microsystems**，以后应该会越来越少人记得这家曾经无比伟大的公司了。

虽然这个白皮书有点老了，不过那个时候 Sun 在 **J2SE 5.0** 版本的 HotSpot 虚拟机上已经有了 Parallel 并行垃圾收集器和 CMS 这种并发收集器了，所以其实内容也没那么过时。

> 其实本文应该有挺多人都翻译过，我大体上是意译的，增、删了部分内容。

其他的知识，包括 Java5 之后的垃圾收集器，如 Java8 的 MetaSpace 取代了永久代、G1 收集器等，将在日后的文章中进行介绍。

目录

垃圾收集概念

GC 需要做 3 件事情：

- 分配内存，为每个新建的对象分配空间
- 确保还在使用的对象的内存一直还在，不能把有用的空间当垃圾回收了
- 释放不再使用的对象所占用的空间

我们把还被 **GC Roots** 引用的对象称为**活的**，把不再被引用的对象认为是**死的**，也就是我们说的垃圾，GC 的工作就是找到死的对象，回收它们占用的空间。

在这里，我们总结一下 GC Roots 有哪些：

- 当前各线程执行方法中的局部变量（包括形参）引用的对象
- 已被加载的类的 static 域引用的对象
- 方法区中常量引用的对象

- JNI 引用

> 以上不完全，不过我觉得了解到这些就够了，了解更多

我们把 GC 管理的内存称为 **堆（heap）**，垃圾收集启动的时机取决于各个垃圾收集器，通常，垃圾收集发生于整个堆或堆的部分已经被使用光了，或者使用的空间达到了某个百分比阈值。这些后面都会具体说，这里的每一句话都是对应了某些场景的。

对于内存分配请求，实现的难点在于在堆中找到一块没有被使用的确定大小的内存空间。所以，对于大部分垃圾回收算法来说**避免内存碎片化**是非常重要的，它将使得空间分配更加高效。

垃圾收集器的理想特征

- 安全和全面：活的对象一定不能被清理掉，死的对象一定不能在几个回收周期结束后还在内存中。
- 高效：不能将我们的应用程序挂起太长时间。我们需要在时间、空间、频次上作出权衡。比如，如果堆内存很小，每次垃圾收集就会很快，但是频次会增加。如果堆内存很大，很久才会被填满，但是每一次回收需要的时间很长。
- 尽量少的内存碎片：每次将垃圾对象释放以后，这些空间可能分布在各个地方，最糟糕的情况就是，内存中到处都是碎片，在给一个大对象分配空间的时候没有内存可用，实际上内存是够的。消除碎片的方式就是**压缩**。
- 可扩展性：在多核多线程应用中，内存分配和垃圾回收都不应该成为可扩展性的瓶颈。**原文提到的这一点，我的理解是：单线程垃圾回收在多核系统中会浪费 CPU 资源，如果我理解错误，请指正我。**

设计上的权衡

往下看之前，我们需要先分清楚这里的两个概念：并发和并行

- **并行**：多个垃圾回收线程同时工作，而不是只有一个垃圾回收线程在工作
- **并发**：垃圾回收线程和应用程序线程同时工作，应用程序不需要挂起

在设计或选择垃圾回收算法的时候，我们需要作出以下几个权衡：

- **串行 vs 并行**

  串行收集的情况，即使是多核 CPU，也只有一个核心参与收集。使用并行收集器的话，垃圾收集的工作将分配给多个线程在不同的 CPU 上同时进行。并行可以让收集工作更快，缺点是带来

的复杂性和内存碎片问题。

- **并发 vs Stop-the-world**

  当 stop-the-world 垃圾收集器工作的时候，应用将完全被挂起。与之相对的，并发收集器在大部分工作中都是并发进行的，也许会有少量的 stop-the-world。

  stop-the-world 垃圾收集器比并发收集器简单很多，因为应用挂起后**堆空间不再发生变化**，它的缺点是在某些场景下挂起的时间我们是不能接受的（如 web 应用）。

  相应的，并发收集器能够降低挂起时间，但是也更加复杂，因为在收集的过程中，也会有新的垃圾产生，同时，需要有额外的空间用于在垃圾收集过程中应用程序的继续使用。

- **压缩 vs 不压缩 vs 复制**

  当垃圾收集器标记出内存中哪些是活的，哪些是垃圾对象后，收集器可以进行压缩，将所有活的对象移到一起，这样新的内存分配就可以在剩余的空间中进行了。经过压缩后，分配新对象的内存空间是非常简单快速的。

  相对的，不压缩的收集器只会就地释放空间，不会移动存活对象。优点就是快速完成垃圾收集，缺点就是潜在的碎片问题。通常，这种情况下，分配对象空间会比较慢比较复杂，比如为新的一个大对象找到合适的空间。

  还有一个选择就是复制收集器，将活的对象复制到另一块空间中，优点就是原空间被清空了，这样后续分配对象空间非常迅速，缺点就是需要进行复制操作和占用额外的空间。

性能指标

以下几个是评估垃圾收集器性能的一些指标：

- 吞吐量：应用程序的执行时间占总时间的百分比，当然是越高越好
- 垃圾收集开销：垃圾收集时间占总时间的百分比（1 – 吞吐量）
- 停顿时间：垃圾收集过程中导致的应用程序挂起时间
- 频次：相对于应用程序来说，垃圾收集的频次
- 空间：垃圾收集占用的内存
- 及时性：一个对象从成为垃圾到该对象空间再次可用的时间

在交互式程序中，通常希望是低延时的，而对于非交互式程序，总运行时间比较重要。实时应用程序既要求每次停顿时间足够短，也要求总的花费在收集的时间足够短。在小型个人计算机和嵌入式系统中，则希望占用更小的空间。

分代收集介绍

当我们使用分代垃圾收集器时，内存将被分为不同的**代(generation)**，最常见的就是分为**年轻代**和**老年代**。

在不同的分代中，可以根据不同的特点使用不同的算法。分代垃圾收集基于 weak generational hypothesis 假设（通常国人会翻译成 **弱分代**假设）：

- 大部分对象都是短命的，它们在年轻的时候就会死去
- 极少老年对象对年轻对象的引用

年轻代中的收集是非常频繁的、高效的、快速的，因为年轻代空间中，通常都是小对象，同时有非常多的不再被引用的对象。

那些**经历过多次年轻代垃圾收集还存活的对象**会晋升到老年代中，老年代的空间更大，而且占用空间增长比较慢。这样，老年代的垃圾收集是不频繁的，但是进行一次垃圾收集需要的时间更长。

对于新生代，需要选择速度比较快的垃圾回收算法，因为新生代的垃圾回收是频繁的。

对于老年代，需要考虑的是空间，因为老年代占用了大部分堆内存，而且针对该部分的垃圾回收算法，需要考虑到这个区域的**垃圾密度比较低**。

J2SE 5.0 HotSpot JVM 中的垃圾收集器

J2SE 5.0 HotSpot 虚拟机包含四种垃圾收集器，都是采用分代算法。包括**串行收集器**、**并行收集器**、**并行压缩收集器** 和 **CMS 垃圾收集器**。

HotSpot 分代

在 HotSpot 虚拟机中，内存被组织成三个分代：年轻代、老年代、永久代。

大部分对象初始化的时候都是在年轻代中的。

老年代存放经过了几次年轻代垃圾收集依然还活着的对象，还有部分大对象因为比较大所以分配的时候直接在老年代分配。

> 如 –XX:PretenureSizeThreshold=1024，这样大于 1k 的对象就会直接分配在老年代

永久代，通常也叫 **方法区**，用于存储已加载类的元数据，以及存储运行时常量池等。

垃圾回收类型

当年轻代被填满后，会进行一次年轻代垃圾收集（也叫做 **minor GC**）。

> 下面这两段我也没有完全弄明白，弄明白会更新。至少读者要明白一点，"minor gc 收集年轻代，full gc 收集老年代" 这句话是错的。

当老年代或永久代被填满了，会触发 **full GC**（也叫做 **major GC**），full GC 会收集所有区域，先进行年轻代的收集，使用年轻代专用的垃圾回收算法，然后使用老年代的垃圾回收算法**回收老年代和永久代**。如果算法带有压缩，每个代分别独立地进行压缩。

如果先进行年轻代垃圾收集，会使得老年代不能容纳要晋升上来的对象，这种情况下，不会先进行 young gc，所有的收集器都会（除了 CMS）**直接采用老年代收集算法对整个堆进行收集**（CMS 收集器比较特殊，因为它不能收集年轻代的垃圾）。

> 基于统计，计算出每次年轻代晋升到老年代的平均大小，if (老年代剩余空间 < 平均大小) 触发 full gc。

快速分配

如果垃圾收集完成后，存在大片连续的内存可用于分配给新对象，这种情况下分配空间是非常简单快速的，只要一个简单的指针碰撞就可以了（**bump-the-pointer**），每次分配对象空间只要检测一下是否有足够的空间，如果有，指针往前移动 N 位就分配好空间了，然后就可以初始化这个对象了。

对于多线程应用，对象分配必须要保证线程安全性，如果使用全局锁，那么分配空间将成为瓶颈并降低程序性能。HotSpot 使用了称之为 **Thread-Local Allocation Buffers (TLABs)** 的技术，该技术能改善多线程空间分配的吞吐量。首先，给予每个线程一部分内存作为缓存区，每个线程都在自己的缓存区中进行指针碰撞，这样就不用获取全局锁了。只有当一个线程使用完了它的 TLAB，它才需要使用同步来获取一个新的缓冲区。HotSpot 使用了多项技术来降低 TLAB 对于内存的浪费。比如，TLAB 的平均大小被限制在 Eden 区大小的 1% 之内。TLABs 和使用指针碰撞的线性分配结合，使得内存分配非常简单高效，只需要大概 10 条机器指令就可以完成。

串行收集器

使用串行收集器，年轻代和老年代都使用单线程进行收集（使用一个 CPU），收集过程中会 stop-the-world。所以当在垃圾收集的时候，应用程序是完全停止的。

在年轻代中使用串行收集器

下图展示了年轻代中使用串行收集器的流程。

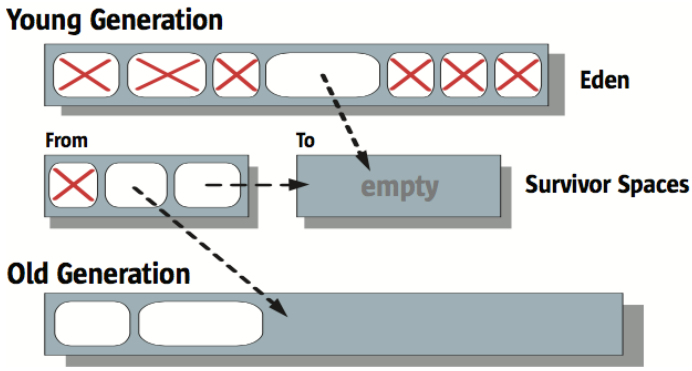

Figure 3. Serial young generation collection

年轻代分为**一个 Eden 区和两个 Survivor 区（From 区和 To 区）**。年轻代垃圾收集时，将 Eden 中活着的对象复制到空的 Survivor-To 区，Survivor-From 区的对象分两类，一类是年轻的，也是复制到 Survivor-To 区，还有一类是老家伙，晋升到老年代中。

> Survivor-From 和 Survivor-To 是我瞎取的名字。。。

如果复制的过程中，发现 Survivor-To 空间满了，将剩下还没复制到 Survivor-To 的来自于 Eden 和 Survivor-From 区的对象直接晋升到老年代。

年轻代垃圾收集完成后，Eden 区和 Survivor-From 就干净了，此时，将 Survivor-From 和 Survivor-To 交换一下角色。得到下面这个样子：

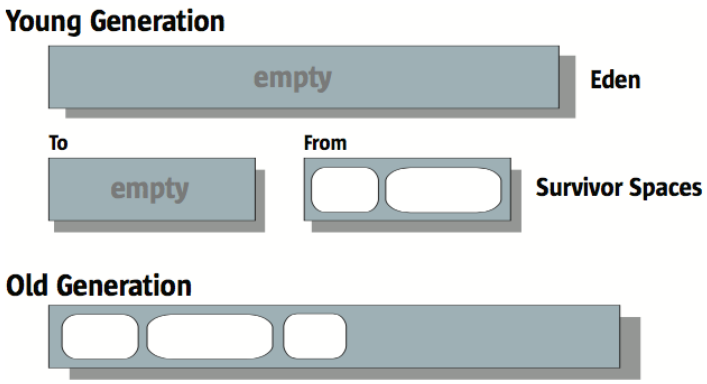

Figure 4. After a young generation collection

在老年代中使用串行收集器

如果使用串行收集器，在老年代和永久代将通过使用 **标记 -> 清除 -> 压缩** 算法。标记阶段，收集器识别出哪些对象是活的；清除阶段将遍历一下老年代和永久代，识别出哪些是垃圾；然后执行压缩，将活的对象左移到老年代的起始端（永久代类似），这样就留下了右边一片连续可用的空间，后续就可以通过指针碰撞的方式快速分配对象空间。

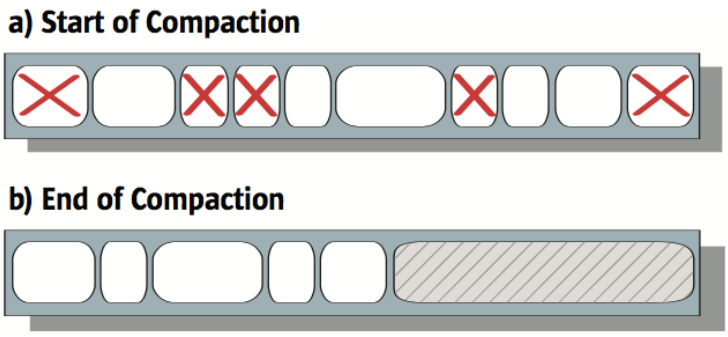

Figure 5. Compaction of the old generation

何时应该使用串行收集器

串行收集器适用于运行在 client 模式下的大部分程序，它们不要求低延时。在现代硬件条件下，串行收集器可以高效管理 64M 堆内存，并且能将 full GC 控制在半秒内完成。

使用串行收集器

它是 J2SE 5.0 版本 HotSpot 虚拟机在非服务器级别硬件的默认选择。你也可以使用 `-XX:+UseSerialGC` 来强制使用串行收集器。

并行收集器

现在大多数 Java 应用都运行在大内存、多核环境中，**并行收集器**，也就是大家熟知的**吞吐量收集器**，利用多核的优势来进行垃圾收集，而不是像串行收集器一样将程序挂起后只使用单线程来收集垃圾。

在年轻代中使用并行收集器

并行收集器在年轻代中其实就是串行收集器收集算法的并行版本。它仍然使用 stop-the-world 和复制算法，只不过使用了多核的优势并行执行，降低垃圾收集的时间，从而提高吞吐量。下图示意了在年轻代中，串行收集器和并行收集器的区别：

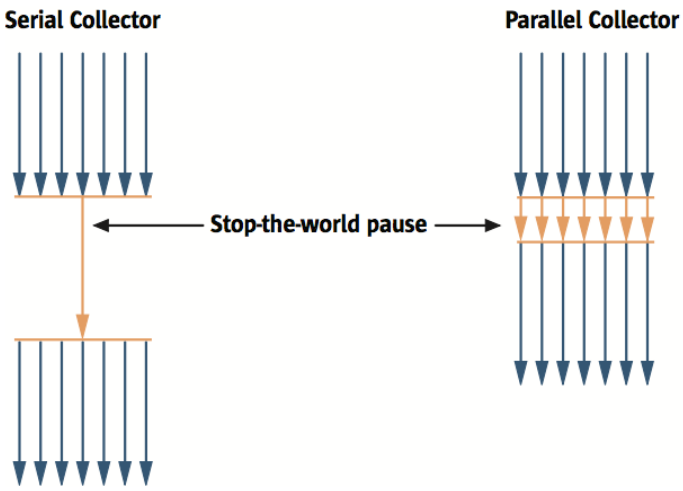

Figure 6. Comparison between serial and parallel young generation collection

在老年代中使用并行收集器

在老年代中，并行收集器使用的是和串行收集器一样的算法：**单线程，标记 -> 清除 -> 压缩**。

> 是的，并行收集器只能在年轻代中并行

何时使用并行收集器

其适用于多核、不要求低停顿的应用，因为老年代的收集虽然不频繁，但是每次老年代的**单线程垃圾收集**依然可能会需要很长时间。比如说，它可以应用在批处理、账单计算、科学计算等。

你应该不会想要这个收集器，而是要一个可以对每个代都采用并行收集的**并行压缩收集器**，下一节将介绍这个。

使用并行收集器

前面我们说了，J2SE 5.0 中 client 模式自动选择使用串行收集器，如果是 server 模式，那么将自动使用并行收集器。在其他版本中，显示使用 -XX:+UseParallelGC 可以指定并行收集器。

并行压缩收集器

并行压缩收集器于 J2SE 5.0 update 6 引入，和并行收集器的区别在于它在老年代也使用并行收集算法。注意：并行压缩收集器终将会取代并行收集器。

在年轻代中使用并行压缩收集器

并行压缩收集器在年轻代中使用了和并行收集器一样的算法。即使用 **并行、stop-the-world、复制** 算法。

在老年代中使用并行压缩收集器

在老年代和永久代中，其使用 **并行、stop-the-world、滑动压缩** 算法。

一次收集分三个阶段，首先，将老年代或永久代逻辑上分为固定大小的区块。

- **标记阶段**，将 GC Roots 分给多个垃圾收集线程，每个线程并行地去标记存活的对象，一旦标记一个存活对象，在**该对象所在的区块**记录这个对象的大小和对象所在的位置。
- **汇总阶段**，此阶段针对区块进行。由于之前的垃圾回收影响，老年代和永久代的左侧是 **存活对象密集区**，对这部分区域直接进行压缩的代价是不值得的，能清理出来的空间有限。所以第一件事就是，检查每个区块的密度，从左边第一个开始，直到找到一个区块满足：**对右侧的所有区块进行压缩获得的空间抵得上压缩它们的成本**。这个区块左边的区域过于密集，不会有对象移动到这个区域中。然后，计算并保存右侧区域中每个区块被压缩后的新位置首字节地址。

  右侧的区域将被压缩，对于右侧的每个区块，由于每个区块中保存了该区块的存活对象信息，所以很容易计算每个区块的新位置。注意：**汇总阶段目前被实现为串行进行**，这个阶段修改为并行也是可行的，不过没有在标记阶段和下面的压缩阶段并行那么重要。
- **压缩阶段**，在汇总阶段已经完成了每个区块新位置的计算，所以压缩阶段每个回收线程**并行**将每个区块复制到新位置即可。压缩结束后，就清出来了右侧一大片连续可用的空间。

何时使用并行压缩收集器

首先是多核上的并行优势，这个就不重复了。其次，前面的并行收集器对于老年代和永久代使用串行，而并行压缩收集器在这些区域使用并行，能降低停顿时间。

并行压缩收集器不适合运行在大型共享主机上（如 SunRays），因为它在收集的时候会独占几个 CPU，在这种机器上，可以考虑减少垃圾收集的线程数（通过 -XX:ParallelGCThreads=n），或者就选择其他收集器。

使用并行压缩收集器

显示指定：`-XX:+UseParallelOldGC`

Concurrent Mark-Sweep（CMS）收集器

> 重头戏 CMS 登场了，至少对于我这个 web 开发者来说，**目前** CMS 最常用（使用 JDK8 的应用一般都切换到 G1 收集器了）。前面介绍的都是并行收集，这里要介绍并发收集了，也就是垃圾回收线程和应用程序线程同时运行。

对于许多程序来说，吞吐量不如响应时间来得重要。通常年轻代的垃圾收集不会停顿多长时间，但是，老年代垃圾回收，虽然不频繁，但是可能导致长时间的停顿，尤其当堆内存比较大的时候。为了解决这个问题，HotSpot 虚拟机提供了 CMS 收集器，也叫做 **低延时收集器**。

在年轻代中使用 CMS 收集器

在年轻代中，CMS 和 **并行收集器** 一样，即：**并行、stop-the-world、复制**。

在老年代中使用 CMS 收集器

在老年代的垃圾收集过程中，大部分收集任务是和应用程序**并发**执行的。

CMS 收集过程首先是一段小停顿 stop-the-world，叫做 **初始标记阶段（initial mark）**，用于确定 GC Roots。然后是 **并发标记阶段（concurrent mark）**，标记 GC Roots 可达的所有存活对象，由于这个阶段应用程序同时也在运行，所以并发标记阶段结束后，并不能标记出所有的存活对象。为了解决这个问题，需要再次停顿应用程序，称为 **再次标记阶段（remark）**，遍历在并发标记阶段应用程序修改的对象（标记出应用程序在这个期间的活对象），由于这次停顿比初始标记要长得多，**所以会使用多线程并行执行来增加效率**。

再次标记阶段结束后，能保证所有存活对象都被标记完成，所以接下来的 **并发清理阶段（concurrent sweep）** 将就地回收垃圾对象所占空间。下图示意了老年代中 **串行、标记 -> 清理 -> 压缩收集器**和 CMS 收集器的区别：

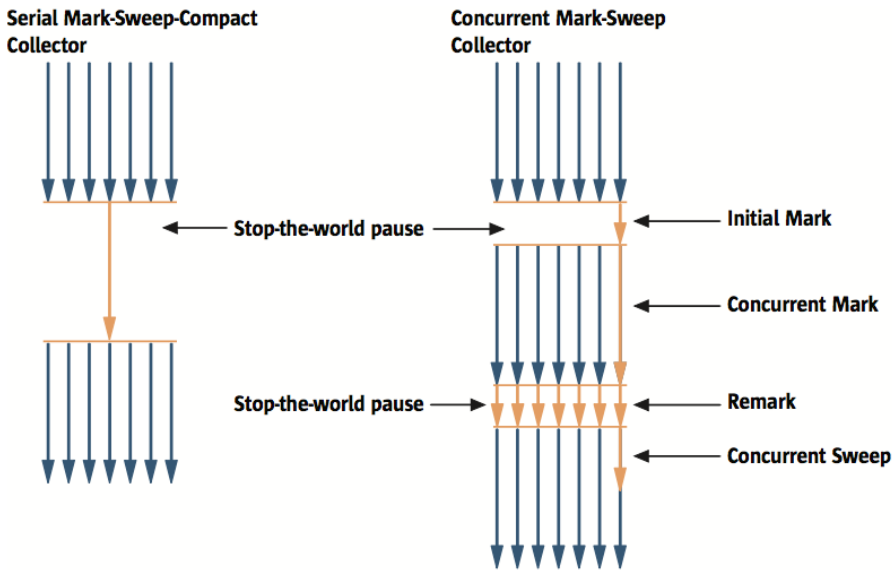

Figure 7. Comparison between serial and CMS old generation collection

由于部分任务增加了收集器的工作，如遍历并发阶段应用程序修改的对象，所以增加了 CMS 收集器的负载。对于大部分试图降低停顿时间的收集器来说，这是一种权衡方案。

CMS 收集器是**唯一不进行压缩的收集器**，在它释放了垃圾对象占用的空间后，它不会移动存活对象到一边去。

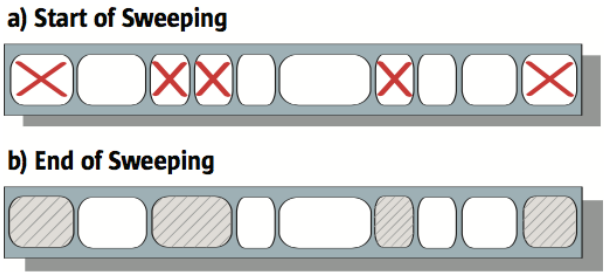

Figure 8. CMS sweeping (but not compacting) of old generation

这将节省垃圾回收的时间，但是由于之后空闲空间不是连续的，所以也就不能使用简单的 **指针碰撞（bump-the-pointer）** 进行对象空间分配了。它需要维护一个 **空闲列表**，将所有的空闲区域连接起来，当分配空间时，需要寻找到一个可以容纳该对象的区域。显然，它比使用简单的指针碰撞成本要高。同时它也会加大年轻代垃圾收集的负载，因为年轻代中的对象如果要晋升到老年代中，需要老年代进行空间分配。

另外一个缺点就是，CMS 收集器相比其他收集器需要使用更大的堆内存。因为在并发标记阶段，程序还需要执行，所以需要留足够的空间给应用程序。另外，虽然收集器能保证在标记阶段识别出所有的存活对象，但是由于应用程序并发运行，所以刚刚标记的存活对象很可能立马成为垃圾，而且这部分由于已经被标记为**存活对象**，所以只能到下次老年代收集才会被清理，这部分垃圾称为 **浮动垃圾**。

最后，由于缺少压缩环节，堆将会出现碎片化问题。为了解决这个问题，CMS 收集器需要追踪统计最常用的对象大小，评估将来的分配需求，可能还需要分割或合并空闲区域。

不像其他垃圾收集器，CMS 收集器不能等到老年代满了才开始收集。否则的话，CMS 收集器将退化到使用更加耗时的 **stop-the-world、标记-清除-压缩** 算法。为了避免这个，CMS 收集器需要统计之前每次垃圾收集的时间和老年代空间被消耗的速度。另外，如果老年代空间被消耗了 **预设占用率（initiating occupancy）**，也将会触发一次垃圾收集，这个占用率通过 **-XX:CMSInitiatingOccupancyFraction=n** 进行设置，n 为老年代空间的占用百分比，默认值是 68。

> 这个数字到 Java8 的时候已经变为默认 92 了。如果老年代空间不足以容纳从新生代垃圾回收晋升上来的对象，那么就会发生 concurrent mode failure，此时会退化到发生 Full GC，清除老年代中的所有无效对象，这个过程是单线程的，比较耗时

> 另外，即使在晋升的时候判断出老年代有足够的空间，但是由于老年代的碎片化问题，其实最终没法容纳晋升上来的对象，那么此时也会发生 Full GC，这次的耗时将更加严重，因为需要对整个堆进行压缩，压缩后年轻代彻底就空了。

总结下来，和并行收集器相比，CMS 收集器**降低了老年代收集时的停顿时间**（有时是显著降低），**稍微增加了一些年轻代收集的时间**、**降低了吞吐量** 以及 **需要更多的堆内存**。

增量模式

CMS 收集器可以使用增量模式，在并发标记阶段，周期性地将自己的 CPU 时钟周期让出来给应用程序。这个功能适用于需要 CMS 的低延时，但是 CPU 核心只有 1 个或 2 个的情况。

> 增量模式在 Java8 已经不推荐使用。
>
> 目前我了解到的是，在所有的并发或并行收集器中，都提供了控制垃圾收集线程数量的参数设置。

何时使用 CMS 收集器

适用于应用程序要求低停顿，同时能接受在垃圾收集阶段和垃圾收集线程一起共享 CPU 资源的场景，典型的就是 web 应用了。

> 在 web 应用中，低延时非常重要，所以 CMS 几乎就是唯一选择，直到后来 G1 的出现。

使用 CMS 收集器

显示指定：-XX:+UseConcMarkSweepGC

如果需要增量模式：一XX:+CMSIncrementalModeoption

> 当然，CMS 还有好些参数可以设置，这里就不展开了，想要了解更多 CMS 细节，建议读者可以参考《Java 性能权威指南》，非常不错的一本书。

小结

虽然是翻译的文章，也小结一下吧。

串行收集器：在年轻代和老年代都采用单线程，年轻代中使用 **stop-the-world**、**复制** 算法；老年代使用 **stop-the-world**、**标记 -> 清理 -> 压缩** 算法。

并行收集器：在年轻代中使用 **并行**、**stop-the-world**、**复制** 算法；老年代使用串行收集器的 **串行**、**stop-the-world**、**标记 -> 清理 -> 压缩** 算法。

并行压缩收集器：在年轻代中使用并行收集器的 **并行、stop-the-world、复制** 算法；老年代使用 **并行、stop-the-world、标记 -> 清理 -> 压缩** 算法。和并行收集器的区别是老年代使用了并行。

CMS 收集器：在年轻使用并行收集器的 **并行、stop-the-world、复制** 算法；老年代使用 **并发、标记 -> 清理** 算法，不压缩。本文介绍的唯一一个并发收集器，也是唯一一个不对老年代进行压缩的收集器。

另外，在 HotSpot 中，永久代使用的是和老年代一样的算法。到了 J2SE 8.0 的 HotSpot JVM 中，永久代被 MetaSpace 取代了，这个以后再介绍。

（全文完）

留下你的评论

昵称 用于接收回复提醒 -> 使用 登录 <-

请使用 markdown 语法进行编辑

预览 提 交

评论区

最 早 最 新

lee 2 months ago

牛逼。。。。。。。。。。。。。。。。。。。。。。。。。。。。

回复

tau 3 months ago

特别好奇博主在哪就职，感觉好厉害

回复

bufflu a year ago

博主发的文章都是精品，赞