

CMS 垃圾回收器

枚举根节点

- 当执行系统停顿下来后，并不需要一个不漏地检查完所有执行上下文和全局的引用位置，虚拟机应当是有办法直接得知哪些地方存放着对象引用。在HotSpot的实现中，是使用一组称为OopMap的数据结构来达到这个目的

安全点

- 在OopMap的协助下，HotSpot可以快速且准确的完成GC Roots枚举，但一个很现实的问题随之而来：可能导致引用关系发生变化，或者说OopMap内容变化的指令非常多，如果为每一条指令都生成对应的OopMap, 那将会需要大量的额外空间，这样GC的空间成本将会变得更高。
- 实际上，HotSpot 并没有为每一条指令都生成OopMap, 而只是在"特定位置"记录了这些信息，这些位置称为安全点（Safepoint），即程序执行时并非在所有的地方都停顿下来GC，只有在达到安全点的时候才能暂停。
- Safepoint 的选定既不能太少以至于让GC等待时间太长，也不能过于频繁以至于过分增大运行时负载。所以，安全点的选定基本上是以"是否具有让程序长时间执行的特征"为标准进行选定的--因为每条指令执行的时间非常短暂，程序不太可能因为指令流长度太长这个原因而过长时间运行，"长时间执行"的最明显特征就是指令序列服用，例如方法调用、循环跳转、异常跳转等，所以具有这些功能的指令才会产生 Safepoint.
- 对于Safepoint，另一个需要考虑的问题是如何在GC发生时让所有线程（这里不包括执行JNI调用的线程）都"跑"到最近的安全点上再停顿下来：抢占式中断（Preemptive Suspension）和主动式中断（Voluntary Suspension）
 - 抢占式中断: 它不需要线程的执行代码主动去配合，在GC发生时，首先把所有的线程全部中断，如果有线程中断的地方不在安全点上，就恢复线程，让它"跑"到安全点上。

- 主动式中断: 当GC需要中断线程的时候, 不直接对线程操作, 仅仅简单地设置一个标志, 各个线程执行时主动去轮询这个标志, 发现中断标志位真时就自己中断挂起。轮询标志的地方和安全点是重合的, 另外再创建对象需要分配内存的地方。

现在几乎没有虚拟机采用抢占式中断来暂停线程从而响应GC事件
安全区域

- 在使用Safepoint似乎已经完美的解决了如何引进 GC的问题, 但实际上情况却并不一定。Safepoint 机制保证了程序执行时, 在不太长的时间内就会遇到可以进入 GC的Safepoint. 但如果程序在"不执行"的时候呢? 所谓恒旭不执行就没有分配CPU的时间, 典型的例子就是处于Sleep状态或者Blocked状态, 这个时候线程无法 响应JVM的中断请求, JVM也显然不太可能等待线程重新分配CPU时间。对于这种情况, 就需要安全区域 (Safe Region) 了。
- 在线程执行到了 Safe Region 中的代码时, 首先标识自己进入了Safe Region, 那样, 当在这段时间里JVM要发起GC时, 就不用管标识自己为 Safe Region 状态 的线程了。在线程要离开 Safe Region时, 它要检查系统是否已经完成了根节点枚举 (或者是整个GC过程), 如果完成了, 那线程就继续执行, 否则它就必须等待直到收到可 以安全离开 Safe Region 的信号为止。

CMS 收集器

- CMS (Concurrent Mark Sweep) 收集器, 以获取最短回收停顿时间为目标, 多数应用于互联网网站或者B/网站的服务器端上。
- CMS 是基于 "标记-清除" 算法的实现的, 整个过程分为4个步骤:
 - 初始标记 (CMS initial mark)
 - 并发编辑 (CMS concurrent mark)
 - 重新标记 (CMS remark)
 - 并发清除 (CMS concurrent sweep)
- 其中初始标记、重新标记这个两个步骤需要"Stop The World";

- 初始标记只是标记一下GC Roots 能直接关联到的对象速度很快；
- 并发标记阶段就是进行GC Roots Tracing 的过程；
- 重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。
- CMS 收集器的运作步骤如下图所示，在整个过程中最费时的并发标记和并发清除过程都可以与用户线程一起工作。因此，从总体上来看，CMS 收集器 的内存回收过程是与用户线程一起并发执行的。



CMS 优点和缺点

- 优点
 - 并发收集、低停顿，Oracle公司的一些官方文档也称为并发低停顿收集器（Concurrent Low Pause Collector）
- 缺点
 - CMS 收集器对CPU资源非常敏感
 - CMS 收集器无法处理 浮动垃圾（Floating Garbage），可能出现"Concurrent Mode Failure"失败而导致另一个Full GC的产生。如果在 中老年代增长不是太快，可以适当调高参数-
XX:CMSInitiatingOccupancyFraction 的值来提高触发百分比，以便降低内存回收次数从而获取更好的性能。是要CMS运行期间预留的内存无法满足程序所需时，虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数：-
XX:CMSInitiatingOccupancyFraction 设置太高很容易导致大量"Concurrent Mode Failure" 失败，反而性能降低。

- 收集结束会有大量的空间碎片产生。空间碎片过多时，会给大对象分配带来很大的麻烦，往往出现老年代还有很大的空间，但是无法找到足够大的连续空间来分配大对象，不得不提前进行一次Full GC，CMS收集器提供了一个XX:+UseCMSCompactAtCollection开关参数（默认就是开启的），用于在CMS收集器顶部主要进行Full GC时开启内存碎片的合并整理过程，内存整理的过程是无法并发的。空间碎片问题没有了，但停顿时间不得不变长。

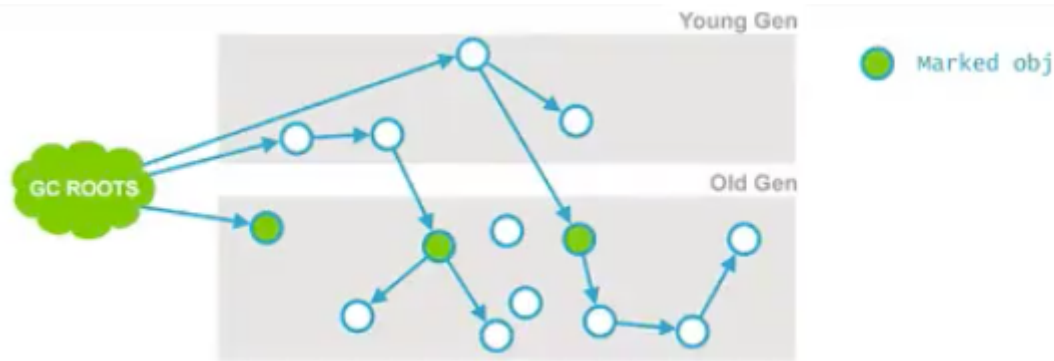
空间分配担保

- 在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么Minor GC可以确保是安全的。当大量对象在Minor GC仍然存活，就需要老年代进行空间分配担保，把Survivor无法容纳的对象直接进入老年代。如果老年代判断到剩余空间不足（更具以往每一次回收晋升到老年代独享容量的平均值作为经验值），则进行一次Full GC

CMS 收集器执行步骤

- **CMS 收集器执行步骤总览**
 - Phase 1: Initial Mark
 - Phase 2: Concurrent Mark
 - Phase 3: Concurrent Preclean
 - Phase 4: Concurrent Abortable Preclean
 - Phase 5: Final Remark
 - Phase 6: Concurrent Sweep
 - Phase 7: Concurrent Reset
- **Phase 1: Initial Mark** （初始标记）
 - 这个是CMS两次stop-the-world事件的其中一次，这个阶段的目标是：标记哪些直接被GC Root引用或者被年

轻代存活对象所引用的所有对象

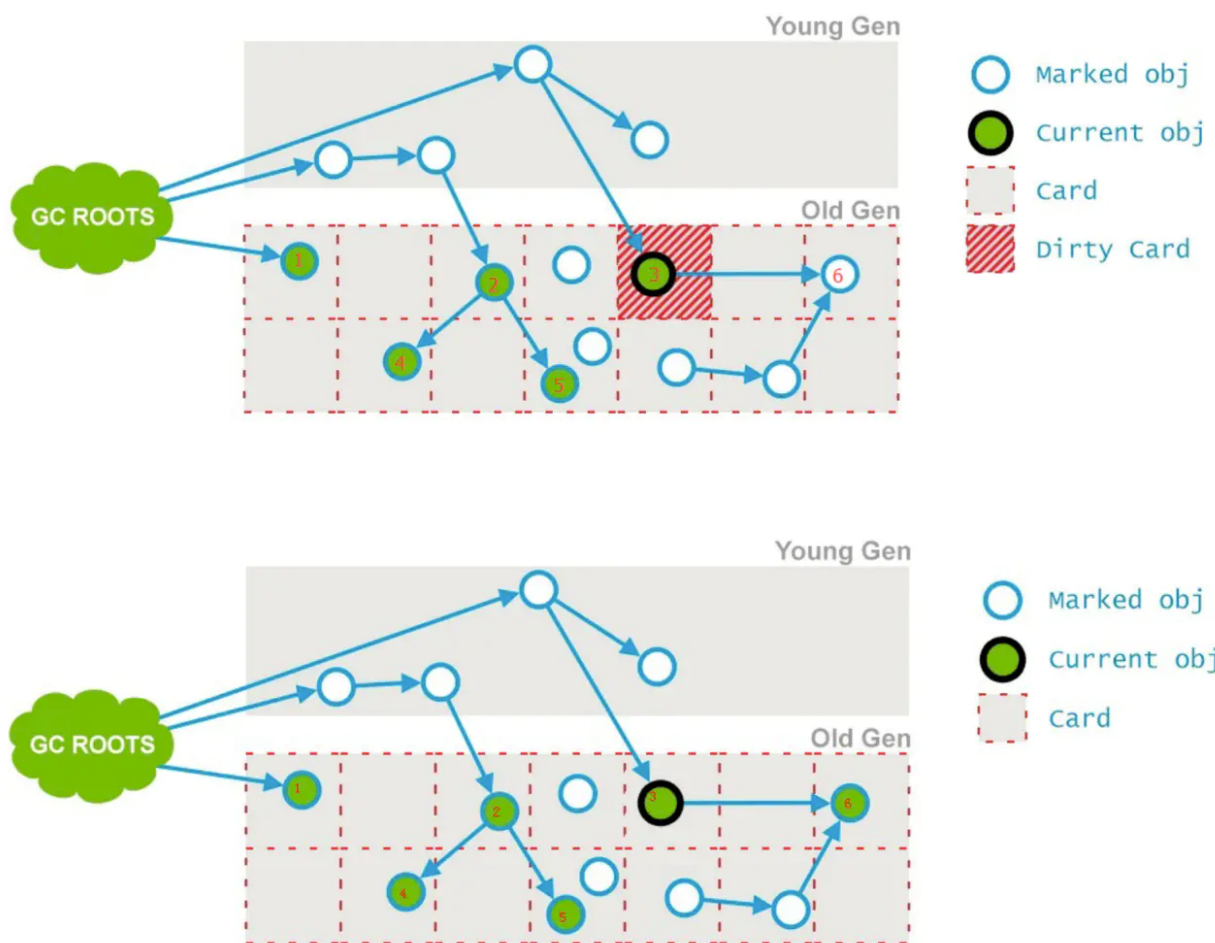


- **Phase 2: Concurrent Mark** (并发标记)
 - 在这个阶段 Garbage Collector 会遍历老年代，然后标记所有存活的对象，它会更具上个阶段找到的GC Roots 遍历查找。并发标记阶段， 他会与用户的应用程序并发运行。并不是老年代所有的存活对象都会被标记， 因为在标记期间用户的程序可能会改变一些引用



- 在上面的图中，与阶段1的图进行对比就会发现有一个对象的引用已经发生额变化。
- **Phase 3: Concurrent Preclean** (并发预先清理)
 - 这个也是一个并发阶段，与应用的线程并发运行，并不会stop应用的线程。在并发运行的过程中，一些对象可能会发生变化，但是这种情况发生时。JVM 将会包含这个对象的区域 (Card) 标记为Dirty, 这也就是Card Marking.
 - 在pre-clean 阶段，哪些能够从Dirty 对象到达的对象也会被标记，这个标记做完后，dirty card 标记就会被清

除了。



- **Phase 4: Concurrent Abortable Preclean** (并发预先可能失败的清理)
 - 这也是一个并发阶段，但是同样不会影响用户的应用线程，这个阶段是为了尽量承担 STW (stop-the-world) 中最终标记阶段的工作。这个阶段 持续时间依赖于很多的因素，由于这个阶段是在重复做很多相同的工作，直接满足一些条件 (比如：重复迭代的次数、完成的工作量或者时钟时间等)。
- **Phase 5: Final Remark** (最终重新标记)
 - 这个阶段是第二个STW阶段，也是 CMS的最后一个，这个阶段的目标是标记老年代所有的存活对象，由于之前的阶段是并发执行的，GC线程可能跟不上应用程序 的变

化，为了完成标记老年代所有存活对象的目标，STW就非常有必要了

- 通常CMS的Final Remark 阶段会在年轻代尽可能干净的时候运行，目的是为了减少STW发生的可能性（年轻代存回对象过多的话，也会导致老年代设计的存活对象会很多）。这个阶段会比前面几个阶段更复杂一些。
- 标记阶段完成
 - 经历过这5个阶段之后，老年代所有存活的对象都被标记过了，现在可以通过清除算法去清理哪些老年代不再使用的对象。
- **Phase 6: Concurrent Sweep**（并发清理）
 - 这里不需要STW，它是与用户的应用程序并发运行的，这个阶段是：清除哪些不再使用的对象，回收他们占用的空间为将来使用。
- **Phase 7: Concurrent Reset**（并发重置）
 - 这个阶段是并发执行的，它会重设 CMS 内部的数据结构，为下次的GC做准备。

总结

CMS 通过将大量工作分散到并发处理阶段来减少STW时间，在这块做的非常优秀，但是CMS也有一些其他的问题。

- CMS 收集器无法清理浮动垃圾（Floating Garbage），可能出现 "Concurrent Mode Failure" 失败而导致另一次 Full GC 的产生，可能引发串行Full GC
- 空间碎片，导致无法分配大对象，CMS收集器提供了一个 — `XX:+UseCMSCompactAtFullCollection` 开关参数（默认就是开启的），用于在CMS收集器顶部主要 记性Full GC时开启内存碎片的合并整理过程，内存整理的过程是无法并发的，空间碎片问题没有了，但停顿时间不得不变长。

- 对于堆较大的应用，GC的时间难以预估

CMS GC日志实例

- JVM 参数

```
-verbose:gc
-Xms20M
-Xmx20M
-Xmn10M
-XX:+PrintGCDetails
-XX:SurvivorRatio=8
-XX:+UseConcMarkSweepGC //设置使用CMS回收器
```

- GC 日志

```
[GC (CMS Initial Mark) [1 CMS-initial-mark: 8194K(10240K)] 12935K(19456K),
0.0004987 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
[CMS-concurrent-mark-start]
[CMS-concurrent-mark: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
[CMS-concurrent-preclean-start]
[CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
[CMS-concurrent-abortable-preclean-start]
[CMS-concurrent-abortable-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs]
[GC (CMS Final Remark) [YG occupancy: 4741 K (9216 K)][Rescan (parallel) ,
0.0010423 secs][weak refs processing, 0.0000103 secs][class unloading, 0.0003310
secs][scrub symbol table, 0.0006947 secs][scrub string table, 0.0001765 secs][1
CMS-remark: 8194K(10240K)] 12935K(19456K), 0.0023344 secs] [Times: user=0.00
sys=0.00, real=0.00 secs]
[CMS-concurrent-sweep-start]
[CMS-concurrent-sweep: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
[CMS-concurrent-reset-start]
[CMS-concurrent-reset: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00
secs]
```