JVM 垃圾回收器工作原理及使用實例介紹

importnew.com/16325.html

原文出處： 周 明耀

垃圾收集基礎

Java 語言的一大特點就是可以進行自動垃圾回收處理，而無需開發人員過於關注系統資源，例如內存資源的釋放情況。自動垃圾收集雖然大大減輕了開發人員的工作量，但是也增加了軟件系統的負擔。

擁有垃圾收集器可以說是 Java 語言與 C++語言的一項顯著區別。在 C++語言中，程序員必須小心謹慎地處理每一項內存分配，且內存使用完後必須手工釋放曾經佔用的內存空間。當內存釋放不夠完全時，即存在分配但永不釋放的內存塊，就會引起內存洩漏，嚴重時甚至導致程序癱瘓。

以下列舉了垃圾回收器常用的算法及實驗原理：

- 引用計數法 (Reference Counting)

引用計數器在微軟的 COM 組件技術中、Adobe 的 ActionScript3 種都有使用。

引用計數器的實現很簡單，對於一個對象 A，只要有任何一個對象引用了 A，則 A 的引用計數器就加 1，當引用失效時，引用計數器就減 1。只要對象 A 的引用計數器的值為 0，則對象 A 就不可能再被使用。

引用計數器的實現也非常簡單，只需要為每個對象配置一個整形的計數器即可。但是引用計數器有一個嚴重的問題，即無法處理循環引用的情況。因此，在 Java 的垃圾回收器中沒有使用這種算法。

一個簡單的循環引用問題描述如下：有對象 A 和對象 B，對象 A 中含有對象 B 的引用，對象 B 中含有對象 A 的引用。此時，對象 A 和對象 B 的引用計數器都不為 0。但是在系統中卻不存在任何第 3 個對象引用了 A 或 B。也就是說，A 和 B 是應該被回收的垃圾對象，但由於垃圾對象間相互引用，從而使垃圾回收器無法識別，引起內存洩漏。

- 標記-清除算法 (Mark-Sweep)

標記-清除算法將垃圾回收分為兩個階段：標記階段和清除階段。一種可行的實現是，在標記階段首先通過根節點，標記所有從根節點開始的較大對象。因此，未被標記的對象就是未被引用的垃圾對象。然後，在清除階段，清除所有未被標記的對象。該算法最大的問題是存在大量的空間碎片，因為回收後的空間是不連續的。在對象的堆空間分配過程中，尤其是大對象的內存分配，不連續的內存空間的工作效率要低於連續的空間。

- 複製算法 (Copying)

將現有的內存空間分為兩快，每次只使用其中一塊，在垃圾回收時將正在使用的內存中的存活對象複製到未被使用的內存塊中，之後，清除正在使用的內存塊中的所有對象，交換兩個內存的角色，完成垃圾回收。

如果系統中的垃圾對象很多，複製算法需要複製的存活對象數量並不會太大。因此在真正需要垃圾回收的時刻，複製算法的效率是很高的。又由於對象在垃圾回收過程中統一被覆制到新的內存空間中，因此，可確保回收後的內存空間是沒有碎片的。該算法的缺點是將系統內存摺半。

Java 的新生代串行垃圾回收器中使用了複製算法的思想。新生代分為 eden 空間、from 空間、to 空間 3 個部分。其中 from 空間和 to 空間可以視為用於複製的兩塊大小相同、地位相等，且可進行角色互換的空間塊。from 和 to 空間也稱為 survivor 空間，即倖存者空間，用於存放未被回收的對象。

在垃圾回收時，eden 空間中的存活對象會被覆制到未使用的 survivor 空間中 (假設是 to)，正在使用的 survivor 空間 (假設是 from) 中的年輕對象也會被覆制到 to 空間中 (大對象，或者老年對象會直接進入老年帶，如果 to 空間已滿，

則對象也會直接進入老年代)。此時，eden 空間和 from 空間中的剩餘對象就是垃圾對象，可以直接清空，to 空間則存放此次回收後的存活對象。這種改進的複製算法既保證了空間的連續性，又避免了大量的內存空間浪費。

- 標記-壓縮算法 (Mark-Compact)

複製算法的高效性是建立在存活對象少、垃圾對象多的前提下的。這種情況在年輕代經常發生，但是在老年代更常見的情況是大部分對象都是存活對象。如果依然使用複製算法，由於存活的對象較多，複製的成本也將很高。

標記-壓縮算法是一種老年代的回收算法，它在標記-清除算法的基礎上做了一些優化。也首先需要從根節點開始對所有可達對象做一次標記，但之後，它並不簡單地清理未標記的對象，而是將所有的存活對象壓縮到內存的一端。之後，清理邊界外所有的空間。這種方法既避免了碎片的產生，又不需要兩塊相同的內存空間，因此，其性價比比較高。

- 增量算法 (Incremental Collecting)

在垃圾回收過程中，應用軟件將處於一種 CPU 消耗很高的狀態。在這種 CPU 消耗很高的狀態下，應用程序所有的線程都會掛起，暫停一切正常的工作，等待垃圾回收的完成。如果垃圾回收時間過長，應用程序會被掛起很久，將嚴重影響用戶體驗或者係統的穩定性。

增量算法的基本思想是，如果一次性將所有的垃圾進行處理，需要造成系統長時間的停頓，那麼就可以讓垃圾收集線程和應用程序線程交替執行。每次，垃圾收集線程只收集一小片區域的內存空間，接著切換到應用程序線程。依次反覆，直到垃圾收集完成。使用這種方式，由於在垃圾回收過程中，間斷性地還執行了應用程序代碼，所以能減少系統的停頓時間。但是，因為線程切換和上下文轉換的消耗，會使得垃圾回收的總體成本上升，造成系統吞吐量的下降。

- 分代 (Generational Collecting)

根據垃圾回收對象的特性，不同階段最優的方式是使用合適的算法用於本階段的垃圾回收，分代算法即是基於這種思想，它將內存區間根據對象的特點分成幾塊，根據每塊內存區間的特點，使用不同的回收算法，以提高垃圾回收的效率。以 Hot Spot 虛擬機為例，它將所有的新建對象都放入稱為年輕代的內存區域，年輕代的特點是對象會很快回收，因此，在年輕代就選擇效率較高的複製算法。當一個對象經過幾次回收後依然存活，對象就會被放入稱為老生代的內存空間。在老生代中，幾乎所有的對象都是經過幾次垃圾回收後依然得以倖存的。因此，可以認為這些對象在一段時期內，甚至在應用程序的整個生命週期中，將是常駐內存的。如果依然使用複製算法回收老生代，將需要複製大量對象。再加上老生代的回收性價比也要低於新生代，因此這種做法也是不可取的。根據分代的思想，可以對老年代的回收使用與新生代不同的標記-壓縮算法，以提高垃圾回收效率。

從不同角度分析垃圾收集器，可以將其分為不同的類型。

1. 按線程數分，可以分為串行垃圾回收器和並行垃圾回收器。串行垃圾回收器一次只使用一個線程進行垃圾回收；並行垃圾回收器一次將開啟多個線程同時進行垃圾回收。在並行能力較強的 CPU 上，使用並行垃圾回收器可以縮短 GC 的停頓時間。

2. 按照工作模式分，可以分為並發式垃圾回收器和獨佔式垃圾回收器。並發式垃圾回收器與應用程序線程交替工作，以儘可能減少應用程序的停頓時間；獨佔式垃圾回收器 (Stop the world) 一旦運行，就停止應用程序中的其他所有線程，直到垃圾回收過程完全結束。

3. 按碎片處理方式可分為壓縮式垃圾回收器和非壓縮式垃圾回收器。壓縮式垃圾回收器會在回收完成後，對存活對象進行壓縮整理，消除回收後的碎片；非壓縮式的垃圾回收器不進行這步操作。

4. 按工作的內存區間，又可分為新生代垃圾回收器和老年代垃圾回收器。

可以用以下指標評價一個垃圾處理器的好壞。

吞吐量：指在應用程序的生命週期內，應用程序所花費的時間和系統總運行時間的比值。系統總運行時間=應用程序耗時+GC 耗時。如果系統運行了 100min，GC 耗時 1min，那麼系統的吞吐量就是 (100-1)/100=99%。

垃圾回收器負載：和吞吐量相反，垃圾回收器負載指來記回收器耗時與系統運行總時間的比值。

停頓時間：指垃圾回收器正在運行時，應用程序的暫停時間。對於獨佔回收器而言，停頓時間可能會比較長。使用並發的回收器時，由於垃圾回收器和應用程序交替運行，程序的停頓時間會變短，但是，由於其效率很可能不如獨佔垃圾回收器，故系統的吞吐量可能會較低。

垃圾回收頻率：指垃圾回收器多長時間會運行一次。一般來說，對於固定的應用而言，垃圾回收器的頻率應該是越低越好。通常增大堆空間可以有效降低垃圾回收發生的頻率，但是可能會增加回收產生的停頓時間。

反應時間：指當一個對象被稱為垃圾後多長時間內，它所佔據的內存空間會被釋放。

堆分配：不同的垃圾回收器對堆內存的分配方式可能是不同的。一個良好的垃圾收集器應該有一個合理的堆內存區間劃分。

JVM 垃圾回收器分類

- 新生代串行收集器

串行收集器主要有兩個特點：第一，它僅僅使用單線程進行垃圾回收；第二，它獨佔式的垃圾回收。

在串行收集器進行垃圾回收時，Java 應用程序中的線程都需要暫停，等待垃圾回收的完成，這樣給用戶體驗造成較差效果。雖然如此，串行收集器卻是一個成熟、經過長時間生產環境考驗的極為高效的收集器。新生代串行處理器使用複製算法，實現相對簡單，邏輯處理特別高效，且沒有線程切換的開銷。在諸如單 CPU 處理器或者較小的應用內存等硬件平台不是特別優越的場合，它的性能表現可以超過並行回收器和並發回收器。在 HotSpot 虛擬機中，使用-XX：+UseSerialGC 參數可以指定使用新生代串行收集器和老年代串行收集器。當 JVM 在 Client 模式下運行時，它是默認的垃圾收集器。一次新生代串行收集器的工作輸出日誌類似如清單 1 信息 (使用-XX:+PrintGCDetails 開關) 所示。

清單 1. 一次新生代串行收集器的工作輸出日誌

```
1  [GC [DefNew: 3468K->150K(9216K), 0.0028638 secs][Tenured:
2    1562K->1712K(10240K), 0.0084220 secs] 3468K-
3  >1712K(19456K),
4    [Perm : 377K->377K(12288K)],
     0.0113816 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
```

它顯示了一次垃圾回收前的新生代的內存佔用量和垃圾回收後的新生代內存佔用量，以及垃圾回收所消耗的時間。

- 老年代串行收集器

老年代串行收集器使用的是標記-壓縮算法。和新生代串行收集器一樣，它也是一個串行的、獨佔式的垃圾回收器。由於老年代垃圾回收通常會使用比新生代垃圾回收更長的時間，因此，在堆空間較大的應用程序中，一旦老年代串行收集器啟動，應用程序很可能會因此停頓幾秒甚至更長時間。雖然如此，老年代串行回收器可以和多種新生代回收器配合使用，同時它也可以作為 CMS 回收器的備用回收器。若要啟用老年代串行回收器，可以嘗試使用以下參數：-XX:+UseSerialGC: 新生代、老年代都使用串行回收器。

清單 2. 一次老年代串行收集器的工作輸出日誌

```
Heap
 def new generation total 4928K, used 1373K [0x27010000, 0x27560000,
0x2c560000)
 eden space 4416K, 31% used [0x27010000, 0x27167628, 0x27460000)
 from space 512K, 0% used [0x27460000, 0x27460000, 0x274e0000)
 to space 512K, 0% used [0x274e0000, 0x274e0000, 0x27560000)
 tenured generation total 10944K, used 0K [0x2c560000, 0x2d010000, 0x37010000)
 the space 10944K, 0% used [0x2c560000, 0x2c560000, 0x2c560200, 0x2d010000)
 compacting perm gen total 12288K, used 376K [0x37010000, 0x37c10000,
0x3b010000)
 the space 12288K, 3% used [0x37010000, 0x3706e0b8, 0x3706e200, 0x37c10000)
 ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
 rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

如果使用-XX:+UseParNewGC 參數設置，表示新生代使用並行收集器，老年代使用串行收集器，如清單 3 所示。

清單 3. 一次串並行收集器混合使用的工作輸出日誌

```
Heap
 par new generation total 4928K, used 1373K [0x0f010000, 0x0f560000,
0x14560000)
 eden space 4416K, 31% used [0x0f010000, 0x0f167620, 0x0f460000)
 from space 512K, 0% used [0x0f460000, 0x0f460000, 0x0f4e0000)
 to space 512K, 0% used [0x0f4e0000, 0x0f4e0000, 0x0f560000)
 tenured generation total 10944K, used 0K [0x14560000, 0x15010000, 0x1f010000)
 the space 10944K, 0% used [0x14560000, 0x14560000, 0x14560200, 0x15010000)
 compacting perm gen total 12288K, used 2056K [0x1f010000, 0x1fc10000,
0x23010000)
 the space 12288K, 16% used [0x1f010000, 0x1f2121d0, 0x1f212200, 0x1fc10000)
No shared spaces configured.
```

如果使用-XX:+UseParallelGC 參數設置，表示新生代和老年代均使用並行回收收集器。如清單 4 所示。

清單 4. 一次老年代並行回收器的工作輸出日誌

```
[Full GC [Tenured: 1712K->1699K(10240K), 0.0071963 secs] 1712K-
>1699K(19456K),
      [Perm : 377K->372K(12288K)], 0.0072393 secs] [Times: user=0.00 sys=0.00,
      real=0.01 secs]
```

它顯示了垃圾回收前老年代和永久區的內存佔用量，以及垃圾回收後老年代和永久區的內存使用量。

- 並行收集器

並行收集器是工作在新生代的垃圾收集器，它只簡單地將串行回收器多線程化。它的回收策略、算法以及參數和串行回收器一樣。

並行回收器也是獨佔式的回收器，在收集過程中，應用程序會全部暫停。但由於並行回收器使用多線程進行垃圾回收，因此，在並發能力比較強的 CPU 上，它產生的停頓時間要短於串行回收器，而在單 CPU 或者並發能力較弱的

系統中，並行回收器的效果不會比串行回收器好，由於多線程的壓力，它的實際表現很可能比串行回收器差。

開啟並行回收器可以使用參數-XX:+UseParNewGC，該參數設置新生代使用並行收集器，老年代使用串行收集器。

清單 5. 設置參數-XX:+UseParNewGC 的輸出日誌

```
[GC [ParNew: 825K->161K(4928K), 0.0155258 secs][Tenured: 8704K->661K(10944K),
  0.0071964 secs] 9017K->661K(15872K),
  [Perm : 2049K->2049K(12288K)], 0.0228090 secs] [Times: user=0.01 sys=0.00,
real=0.01 secs]
Heap
 par new generation total 4992K, used 179K [0x0f010000, 0x0f570000, 0x14560000)
 eden space 4480K, 4% used [0x0f010000, 0x0f03cda8, 0x0f470000)
 from space 512K, 0% used [0x0f470000, 0x0f470000, 0x0f4f0000)
 to space 512K, 0% used [0x0f4f0000, 0x0f4f0000, 0x0f570000)
 tenured generation total 10944K, used 8853K [0x14560000, 0x15010000,
0x1f010000)
 the space 10944K, 80% used [0x14560000, 0x14e057c0, 0x14e05800, 0x15010000)
 compacting perm gen total 12288K, used 2060K [0x1f010000, 0x1fc10000,
0x23010000)
 the space 12288K, 16% used [0x1f010000, 0x1f213228, 0x1f213400, 0x1fc10000)
No shared spaces configured.
```

設置參數-XX:+UseConcMarkSweepGC 可以要求新生代使用並行收集器，老年代使用 CMS。

清單 6. 設置參數-XX:+UseConcMarkSweepGC 的輸出日誌

```
[GC [ParNew: 8967K->669K(14784K), 0.0040895 secs] 8967K->669K(63936K),
  0.0043255 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 par new generation total 14784K, used 9389K [0x03f50000, 0x04f50000,
0x04f50000)
 eden space 13184K, 66% used [0x03f50000, 0x047d3e58, 0x04c30000)
 from space 1600K, 41% used [0x04dc0000, 0x04e67738, 0x04f50000)
 to space 1600K, 0% used [0x04c30000, 0x04c30000, 0x04dc0000)
 concurrent mark-sweep generation total 49152K, used 0K [0x04f50000,
0x07f50000, 0x09f50000)
 concurrent-mark-sweep perm gen total 12288K, used 2060K [0x09f50000,
0x0ab50000, 0x0df50000)
```

並行收集器工作時的線程數量可以使用-XX:ParallelGCThreads 參數指定。一般，最好與 CPU 數量相當，避免過多的線程數影響垃圾收集性能。在默認情況下，當 CPU 數量小於 8 個，ParallelGCThreads 的值等於 CPU 數量，大於 8 個，ParallelGCThreads 的值等於 3+[5*CPU_Count]/8]。以下測試顯示了筆者筆記本上運行 8 個線程時耗時最短，本人筆記本是 8 核 IntelCPU。

清單 7. 設置為 8 個線程後 GC 輸出

```
[GC [ParNew: 8967K->676K(14784K), 0.0036983 secs] 8967K->676K(63936K),
  0.0037662 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 par new generation total 14784K, used 9395K [0x040e0000, 0x050e0000,
0x050e0000)
 eden space 13184K, 66% used [0x040e0000, 0x04963e58, 0x04dc0000)
 from space 1600K, 42% used [0x04f50000, 0x04ff9100, 0x050e0000)
 to space 1600K, 0% used [0x04dc0000, 0x04dc0000, 0x04f50000)
 concurrent mark-sweep generation total 49152K, used 0K [0x050e0000,
0x080e0000, 0x0a0e0000)
 concurrent-mark-sweep perm gen total 12288K, used 2060K [0x0a0e0000,
0x0ace0000, 0x0e0e0000)
```

清單 8. 設置為 128 個線程後 GC 輸出

```
[GC [ParNew: 8967K->664K(14784K), 0.0207327 secs] 8967K-
>664K(63936K),
 0.0208077 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
```

清單 9. 設置為 640 個線程後 GC 輸出

```
[GC [ParNew: 8967K->667K(14784K), 0.2323704 secs] 8967K-
>667K(63936K),
 0.2324778 secs] [Times: user=0.34 sys=0.02, real=0.23 secs]
```

清單 10. 設置為 1280 個線程後 GC 輸出

```
Error occurred during initialization of
VM
Too small new size specified
```

- 新生代並行回收 (Parallel Scavenge) 收集器

新生代並行回收收集器也是使用複製算法的收集器。從表面上看，它和並行收集器一樣都是多線程、獨佔式的收集器。但是，並行回收收集器有一個重要的特點：它非常關注系統的吞吐量。

新生代並行回收收集器可以使用以下參數啟用：

-XX:+UseParallelGC:新生代使用並行回收收集器，老年代使用串行收集器。

-XX:+UseParallelOldGC:新生代和老年代都是用並行回收收集器。

清單 11. 設置為 24 個線程後 GC 輸出

```
Heap
 PSYoungGen total 4800K, used 893K [0x1dac0000, 0x1e010000, 0x23010000)
 eden space 4160K, 21% used [0x1dac0000,0x1db9f570,0x1ded0000)
 from space 640K, 0% used [0x1df70000,0x1df70000,0x1e010000)
 to space 640K, 0% used [0x1ded0000,0x1ded0000,0x1df70000)
 ParOldGen total 19200K, used 16384K [0x13010000, 0x142d0000,
0x1dac0000)
 object space 19200K, 85% used [0x13010000,0x14010020,0x142d0000)
 PSPermGen total 12288K, used 2054K [0x0f010000, 0x0fc10000, 0x13010000)
 object space 12288K, 16% used [0x0f010000,0x0f2119c0,0x0fc10000)
```

新生代並行回收收集器可以使用以下參數啟用：

-XX:+MaxGCPauseMills:設置最大垃圾收集停頓時間，它的值是一個大於 0 的整數。收集器在工作時會調整 Java 堆大小或者其他一些參數，儘可能地把停頓時間控制在 MaxGCPauseMills 以內。如果希望減少停頓時間，而把這個值設置得很小，為了達到預期的停頓時間，JVM 可能會使用一個較小的堆 (一個小堆比一個大堆回收快)，而這將導致垃圾回收變得很頻繁，從而增加了垃圾回收總時間，降低了吞吐量。

-XX:+GCTimeRatio：設置吞吐量大小，它的值是一個 0-100 之間的整數。假設 GCTimeRatio 的值為 n，那麼系統將花費不超過 1/(1+n) 的時間用於垃圾收集。比如 GCTimeRatio 等於 19，則系統用於垃圾收集的時間不超過 1/(1+19)=5%。默認情況下，它的取值是 99，即不超過 1%的時間用於垃圾收集。

除此之外，並行回收收集器與並行收集器另一個不同之處在於，它支持一種自適應的 GC 調節策略，使用-XX:+UseAdaptiveSizePolicy 可以打開自適應 GC 策略。在這種模式下，新生代的大小、eden 和 survivor 的比例、晉陞老年代的對象年齡等參數會被自動調整，以達到在堆大小、吞吐量和停頓時間之間的平衡點。在手工調優比較困難的場合，可以直接使用這種自適應的方式，僅指定虛擬機的最大堆、目標的吞吐量 (GCTimeRatio) 和停頓時間 (MaxGCPauseMills)，讓虛擬機自己完成調優工作。

清單 12. 新生代並行回收收集器 GC 輸出

```
Heap
 PSYoungGen total 4800K, used 893K [0x1dac0000, 0x1e010000, 0x23010000)
 eden space 4160K, 21% used [0x1dac0000,0x1db9f570,0x1ded0000)
 from space 640K, 0% used [0x1df70000,0x1df70000,0x1e010000)
 to space 640K, 0% used [0x1ded0000,0x1ded0000,0x1df70000)
 PSOldGen total 19200K, used 16384K [0x13010000, 0x142d0000,
0x1dac0000)
 object space 19200K, 85% used [0x13010000,0x14010020,0x142d0000)
 PSPermGen total 12288K, used 2054K [0x0f010000, 0x0fc10000,
0x13010000)
 object space 12288K, 16% used [0x0f010000,0x0f2119c0,0x0fc10000)
```

它也顯示了收集器的工作成果，也就是回收前的內存大小和回收後的內存大小，以及花費的時間。

- 老年代並行回收收集器

老年代的並行回收收集器也是一種多線程並發的收集器。和新生代並行回收收集器一樣，它也是一種關注吞吐量的收集器。老年代並行回收收集器使用標記-壓縮算法，JDK1.6 之後開始啟用。

使用-XX:+UseParallelOldGC 可以在新生代和老生代都使用並行回收收集器，這是一對非常關注吞吐量的垃圾收集器組合，在對吞吐量敏感的系統中，可以考慮使用。參數-XX:ParallelGCThreads 也可以用於設置垃圾回收時的線程數

量。

清單 13 是設置線程數量為 100 時老年代並行回收收集器輸出日誌：

清單 13. 老年代並行回收收集器設置 100 線程時 GC 輸出

```
Heap
 PSYoungGen total 4800K, used 893K [0x1dac0000, 0x1e010000, 0x23010000)
 eden space 4160K, 21% used [0x1dac0000,0x1db9f570,0x1ded0000)
 from space 640K, 0% used [0x1df70000,0x1df70000,0x1e010000)
 to space 640K, 0% used [0x1ded0000,0x1ded0000,0x1df70000)
 ParOldGen total 19200K, used 16384K [0x13010000, 0x142d0000,
0x1dac0000)
 object space 19200K, 85% used [0x13010000,0x14010020,0x142d0000)
 PSPermGen total 12288K, used 2054K [0x0f010000, 0x0fc10000, 0x13010000)
 object space 12288K, 16% used [0x0f010000,0x0f2119c0,0x0fc10000)
```

- CMS 收集器

與並行回收收集器不同，CMS 收集器主要關注於系統停頓時間。CMS 是 Concurrent Mark Sweep 的縮寫，意為並發標記清除，從名稱上可以得知，它使用的是標記-清除算法，同時它又是一個使用多線程並發回收的垃圾收集器。

CMS 工作時，主要步驟有：初始標記、並發標記、重新標記、並發清除和並發重置。其中初始標記和重新標記是獨佔系統資源的，而並發標記、並發清除和並發重置是可以和用戶線程一起執行的。因此，從整體上來說，CMS 收集不是獨佔式的，它可以在應用程序運行過程中進行垃圾回收。

根據標記-清除算法，初始標記、並發標記和重新標記都是為了標記出需要回收的對象。並發清理則是在標記完成後，正式回收垃圾對象；並發重置是指在垃圾回收完成後，重新初始化 CMS 數據結構和數據，為下一次垃圾回收做好準備。並發標記、並發清理和並發重置都是可以和應用程序線程一起執行的。

CMS 收集器在其主要的工作階段雖然沒有暴力地徹底暫停應用程序線程，但是由於它和應用程序線程並發執行，相互搶佔 CPU，所以在 CMS 執行期內對應用程序吞吐量造成一定影響。CMS 默認啟動的線程數是 (ParallelGCThreads+3)/4),ParallelGCThreads 是新生代並行收集器的線程數，也可以通過-XX:ParallelCMSThreads 參數手工設定 CMS 的線程數量。當 CPU 資源比較緊張時，受到 CMS 收集器線程的影響，應用程序的性能在垃圾回收階段可能會非常糟糕。

由於 CMS 收集器不是獨佔式的回收器，在 CMS 回收過程中，應用程序仍然在不停地工作。在應用程序工作過程中，又會不斷地產生垃圾。這些新生成的垃圾在當前 CMS 回收過程中是無法清除的。同時，因為應用程序沒有中斷，所以在 CMS 回收過程中，還應該確保應用程序有足夠的內存可用。因此，CMS 收集器不會等待堆內存飽和時才進行垃圾回收，而是當前堆內存使用率達到某一閾值時，便開始進行回收，以確保應用程序在 CMS 工作過程中依然有足夠的空間支持應用程序運行。

這個回收閾值可以使用-XX:CMSInitiatingOccupancyFraction 來指定，默認是 68。即當老年代的空間使用率達到 68%時，會執行一次 CMS 回收。如果應用程序的內存使用率增長很快，在 CMS 的執行過程中，已經出現了內存不足的情況，此時，CMS 回收將會失敗，JVM 將啟動老年代串行收集器進行垃圾回收。如果這樣，應用程序將完全中斷，直到垃圾收集完成，這時，應用程序的停頓時間可能很長。因此，根據應用程序的特點，可以對-XX:CMSInitiatingOccupancyFraction 進行調優。如果內存增長緩慢，則可以設置一個稍大的值，大的閾值可以有效降低 CMS 的觸發頻率，減少老年代回收的次數可以較為明顯地改善應用程序性能。反之，如果應用程序內存使用率增長很快，則應該降低這個閾值，以避免頻繁觸發老年代串行收集器。

標記-清除算法將會造成大量內存碎片，離散的可用空間無法分配較大的對象。在這種情況下，即使堆內存仍然有較

大的剩餘空間，也可能會被迫進行一次垃圾回收，以換取一塊可用的連續內存，這種現象對系統性能是相當不利的，為瞭解決這個問題，CMS 收集器還提供了幾個用於內存壓縮整理的算法。

-XX:+UseCMSCompactAtFullCollection 參數可以使 CMS 在垃圾收集完成後，進行一次內存碎片整理。內存碎片的整理並不是並發進行的。-XX:CMSFullGCsBeforeCompaction 參數可以用於設定進行多少次 CMS 回收後，進行一次內存壓縮。

-XX:CMSInitiatingOccupancyFraction 設置為 100，同時設置-XX:+UseCMSCompactAtFullCollection 和-XX:CMSFullGCsBeforeCompaction，日誌輸出如下：

清單 14.CMS 垃圾回收器 GC 輸出

```
[GC [DefNew: 825K->149K(4928K), 0.0023384 secs][Tenured: 8704K->661K(10944K),
  0.0587725 secs] 9017K->661K(15872K),
  [Perm : 374K->374K(12288K)], 0.0612037 secs] [Times: user=0.01 sys=0.02,
real=0.06 secs]
Heap
 def new generation total 4992K, used 179K [0x27010000, 0x27570000, 0x2c560000)
 eden space 4480K, 4% used [0x27010000, 0x2703cda8, 0x27470000)
 from space 512K, 0% used [0x27470000, 0x27470000, 0x274f0000)
 to space 512K, 0% used [0x274f0000, 0x274f0000, 0x27570000)
 tenured generation total 10944K, used 8853K [0x2c560000, 0x2d010000,
0x37010000)
 the space 10944K, 80% used [0x2c560000, 0x2ce057c8, 0x2ce05800, 0x2d010000)
 compacting perm gen total 12288K, used 374K [0x37010000, 0x37c10000,
0x3b010000)
 the space 12288K, 3% used [0x37010000, 0x3706db10, 0x3706dc00, 0x37c10000)
 ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
 rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

- G1 收集器 (Garbage First)

G1 收集器的目標是作為一款服務器的垃圾收集器，因此，它在吞吐量和停頓控制上，預期要優於 CMS 收集器。

與 CMS 收集器相比，G1 收集器是基於標記-壓縮算法的。因此，它不會產生空間碎片，也沒有必要在收集完成後，進行一次獨佔式的碎片整理工作。G1 收集器還可以進行非常精確的停頓控制。它可以讓開發人員指定當停頓時長為 M 時，垃圾回收時間不超過 N。使用參數-XX:+UnlockExperimentalVMOptions –XX:+UseG1GC 來啟用 G1 回收器，設置 G1 回收器的目標停頓時間：-XX:MaxGCPauseMills=20,-XX:GCPauseIntervalMills=200。

收集器對系統性能的影響

通過清單 15 所示代碼運行 1 萬次循環，每次分配 512*100B 空間，採用不同的垃圾回收器，輸出程序運行所消耗的時間。

清單 15. 性能測試代碼

```
import java.util.HashMap;

public class GCTimeTest {
 static HashMap map = new HashMap();

 public static void main(String[] args){
 long begintime =
System.currentTimeMillis();
 for(int i=0;i<10000;i++){
 if(map.size()*512/1024/1024>=400){
 map.clear();
 System.out.println("clean map");
 }
 byte[] b1;
 for(int j=0;j<100;j++){
 b1 = new byte[512];
 map.put(System.nanoTime(), b1);
 }
 }
 long endtime = System.currentTimeMillis();
 System.out.println(endtime-begintime);
 }
}
```

使用參數-Xmx512M -Xms512M -XX:+UseParNewGC 運行代碼，輸出如下：

clean map 8565

cost time=1655

使用參數-Xmx512M -Xms512M -XX:+UseParallelOldGC –XX:ParallelGCThreads=8 運行代碼，輸出如下：

clean map 8798

cost time=1998

使用參數-Xmx512M -Xms512M -XX:+UseSerialGC 運行代碼，輸出如下：

clean map 8864

cost time=1717

使用參數-Xmx512M -Xms512M -XX:+UseConcMarkSweepGC 運行代碼，輸出如下：

clean map 8862

cost time=1530

上面例子的 GC 輸出可以看出，採用不同的垃圾回收機制及設定不同的線程數，對於代碼段的整體執行時間有較大的影響。需要讀者有針對性地選用適合自己代碼段的垃圾回收機制。

GC 相關參數總結

1. 與串行回收器相關的參數

-XX:+UseSerialGC:在新生代和老年代使用串行回收器。

-XX:+SuivivorRatio:設置 eden 區大小和 survivor 區大小的比例。

-XX:+PretenureSizeThreshold:設置大對象直接進入老年代的閾值。當對象的大小超過這個值時，將直接在老年代分配。

-XX:MaxTenuringThreshold:設置對象進入老年代的年齡的最大值。每一次 Minor GC 後，對象年齡就加 1。任何大於這個年齡的對象，一定會進入老年代。

2. 與並行 GC 相關的參數

-XX:+UseParNewGC: 在新生代使用並行收集器。

-XX:+UseParallelOldGC: 老年代使用並行回收收集器。

-XX:ParallelGCThreads：設置用於垃圾回收的線程數。通常情況下可以和 CPU 數量相等。但在 CPU 數量比較多的情況下，設置相對較小的數值也是合理的。

-XX:MaxGCPauseMills：設置最大垃圾收集停頓時間。它的值是一個大於 0 的整數。收集器在工作時，會調整 Java 堆大小或者其他一些參數，儘可能地把停頓時間控制在 MaxGCPauseMills 以內。

-XX:GCTimeRatio:設置吞吐量大小，它的值是一個 0-100 之間的整數。假設 GCTimeRatio 的值為 n，那麼系統將花費不超過 1/(1+n) 的時間用於垃圾收集。

-XX:+UseAdaptiveSizePolicy:打開自適應 GC 策略。在這種模式下，新生代的大小，eden 和 survivor 的比例、晉陞老年代的對象年齡等參數會被自動調整，以達到在堆大小、吞吐量和停頓時間之間的平衡點。

3. 與 CMS 回收器相關的參數

-XX:+UseConcMarkSweepGC: 新生代使用並行收集器，老年代使用 CMS+串行收集器。

-XX:+ParallelCMSThreads: 設定 CMS 的線程數量。

-XX:+CMSInitiatingOccupancyFraction:設置 CMS 收集器在老年代空間被使用多少後觸發，默認為 68%。

-XX:+UseFullGCsBeforeCompaction:設定進行多少次 CMS 垃圾回收後，進行一次內存壓縮。

-XX:+CMSClassUnloadingEnabled:允許對類元數據進行回收。

-XX:+CMSParallelRemarkEndable:啟用並行重標記。

-XX:CMSInitatingPermOccupancyFraction:當永久區佔用率達到這一百分比後，啟動 CMS 回收 (前提是-XX:+CMSClassUnloadingEnabled 激活了)。

-XX:UseCMSInitatingOccupancyOnly:表示只在到達閾值的時候，才進行 CMS 回收。

-XX:+CMSIncrementalMode:使用增量模式，比較適合單 CPU。

4. 與 G1 回收器相關的參數

-XX:+UseG1GC：使用 G1 回收器。

-XX:+UnlockExperimentalVMOptions:允許使用實驗性參數。

-XX:+MaxGCPauseMills:設置最大垃圾收集停頓時間。

-XX:+GCPauseIntervalMills:設置停頓間隔時間。

5. 其他參數

-XX:+DisableExplicitGC: 禁用顯示 GC。

結束語

通過本文的學習，讀者可以掌握基本的 JVM 垃圾回收器設計原理及使用規範。基於筆者多年的工作經驗，沒有哪一條優化是可以照本宣科的，它一定是基於您對 JVM 垃圾回收器工作原理及自身程序設計有一定瞭解前提下，通過大量的實驗來找出最適合自己的優化方案。