JVM 優化經驗總結

importnew.com/16327.html

原文出處： 周 耀明

開始之前

Java 虛擬機有自己完善的硬件架構, 如處理器、堆棧、寄存器等，還具有相應的指令系統。JVM 屏蔽了與具體操作系統平台相關的信息，使得 Java 程序只需生成在 Java 虛擬機上運行的目標代碼 (字節碼), 就可以在多種平台上不加修改地運行。Java 虛擬機在執行字節碼時，實際上最終還是把字節碼解釋成具體平台上的機器指令執行。

注意：本文僅針對 JDK7、HotSPOT Java 虛擬機，對於 JDK8 引入的 JVM 新特性及其他 Java 虛擬機，本文不予關注。

我們以一個例子開始這篇文章。假設你是一個普通的 Java 對象，你出生在 Eden 區，在 Eden 區有許多和你差不多的小兄弟、小姐妹，可以把 Eden 區當成幼兒園，在這個幼兒園裡大家玩了很長時間。Eden 區不能無休止地放你們在裡面，所以當年紀稍大，你就要被送到學校去上學，這裡假設從小學到高中都稱為 Survivor 區。開始的時候你在 Survivor 區裡面劃分出來的的「From」區，讀到高年級了，就進了 Survivor 區的「To」區，中間由於學習成績不穩定，還經常來回折騰。直到你 18 歲的時候，高中畢業了，該去社會上闖闖了。於是你就去了年老代，年老代裡面人也很多。在年老代裡，你生活了 20 年 (每次 GC 加一歲)，最後壽終正寢，被 GC 回收。有一點沒有提，你在年老代遇到了一個同學，他的名字叫愛德華 (慕光之城裡的帥哥吸血鬼)，他以及他的家族永遠不會死，那麼他們就生活在永生代。

之前的文章《JVM 垃圾回收器工作原理及使用實例介紹》中已經介紹過年輕代、年老代、永生代，本文主要講講如何運用這些區域，為系統性能提供更好的幫助。本文不再重複這些概念，直接進入主題。

如何將新對象預留在年輕代

眾所周知，由於 Full GC 的成本遠遠高於 Minor GC，因此某些情況下需要儘可能將對象分配在年輕代，這在很多情況下是一個明智的選擇。雖然在大部分情況下，JVM 會嘗試在 Eden 區分配對象，但是由於空間緊張等問題，很可能不得不將部分年輕對象提前向年老代壓縮。因此，在 JVM 參數調優時可以為應用程序分配一個合理的年輕代空間，以最大限度避免新對象直接進入年老代的情況發生。清單 1 所示代碼嘗試分配 4MB 內存空間，觀察一下它的內存使用情況。

清單 1. 相同大小內存分配

```
1  public class PutInEden {
2   public static void main(String[] args){
3   byte[] b1,b2,b3,b4;
4   b1=new byte[1024*1024];
5   b2=new byte[1024*1024];
6   b3=new byte[1024*1024];
7   b4=new byte[1024*1024];
8   }
9  }
```

使用 JVM 參數-XX:+PrintGCDetails -Xmx20M -Xms20M 運行清單 1 所示代碼，輸出如清單 2 所示。

清單 2. 清單 1 運行輸出

```
[GC [DefNew: 5504K->640K(6144K), 0.0114236 secs] 5504K->5352K(19840K),
   0.0114595 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew: 6144K->640K(6144K), 0.0131261 secs] 10856K->10782K(19840K),
0.0131612 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew: 6144K->6144K(6144K), 0.0000170 secs][Tenured: 10142K-
>13695K(13696K),
0.1069249 secs] 16286K->15966K(19840K), [Perm : 376K->376K(12288K)],
0.1070058 secs] [Times: user=0.03 sys=0.00, real=0.11 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0302067 secs] 19839K-
>19595K(19840K),
[Perm : 376K->376K(12288K)], 0.0302635 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0311986 secs] 19839K-
>19839K(19840K),
[Perm : 376K->376K(12288K)], 0.0312515 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0358821 secs] 19839K-
>19825K(19840K),
[Perm : 376K->371K(12288K)], 0.0359315 secs] [Times: user=0.05 sys=0.00,
real=0.05 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0283080 secs] 19839K-
>19839K(19840K),
[Perm : 371K->371K(12288K)], 0.0283723 secs] [Times: user=0.02 sys=0.00,
real=0.01 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0284469 secs] 19839K-
>19839K(19840K),
[Perm : 371K->371K(12288K)], 0.0284990 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0283005 secs] 19839K-
>19839K(19840K),
[Perm : 371K->371K(12288K)], 0.0283475 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0287757 secs] 19839K-
>19839K(19840K),
[Perm : 371K->371K(12288K)], 0.0288294 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0288219 secs] 19839K-
>19839K(19840K),
[Perm : 371K->371K(12288K)], 0.0288709 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0293071 secs] 19839K-
>19839K(19840K),
[Perm : 371K->371K(12288K)], 0.0293607 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
[Full GC [Tenured: 13695K->13695K(13696K), 0.0356141 secs] 19839K-
>19838K(19840K),
[Perm : 371K->371K(12288K)], 0.0356654 secs] [Times: user=0.01 sys=0.00,
real=0.03 secs]
Heap
 def new generation total 6144K, used 6143K [0x35c10000, 0x362b0000,
```

```
0x362b0000)
 eden space 5504K, 100% used [0x35c10000, 0x36170000, 0x36170000)
 from space 640K, 99% used [0x36170000, 0x3620fc80, 0x36210000)
 to space 640K, 0% used [0x36210000, 0x36210000, 0x362b0000)
 tenured generation total 13696K, used 13695K [0x362b0000, 0x37010000,
0x37010000)
 the space 13696K, 99% used [0x362b0000, 0x3700fff8, 0x37010000, 0x37010000)
 compacting perm gen total 12288K, used 371K [0x37010000, 0x37c10000,
0x3b010000)
 the space 12288K, 3% used [0x37010000, 0x3706cd20, 0x3706ce00, 0x37c10000)
 ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
 rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

清單 2 所示的日誌輸出顯示年輕代 Eden 的大小有 5MB 左右。分配足夠大的年輕代空間，使用 JVM 參數-XX:+PrintGCDetails -Xmx20M -Xms20M-Xmn6M 運行清單 1 所示代碼，輸出如清單 3 所示。

清單 3. 增大 Eden 大小後清單 1 運行輸出

```
[GC [DefNew: 4992K->576K(5568K), 0.0116036 secs] 4992K->4829K(19904K),
 0.0116439 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew: 5568K->576K(5568K), 0.0130929 secs] 9821K->9653K(19904K),
0.0131336 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
[GC [DefNew: 5568K->575K(5568K), 0.0154148 secs] 14645K->14500K(19904K),
0.0154531 secs] [Times: user=0.00 sys=0.01, real=0.01 secs]
[GC [DefNew: 5567K->5567K(5568K), 0.0000197 secs][Tenured: 13924K-
>14335K(14336K),
0.0330724 secs] 19492K->19265K(19904K), [Perm : 376K->376K(12288K)],
0.0331624 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]
[Full GC [Tenured: 14335K->14335K(14336K), 0.0292459 secs] 19903K-
>19902K(19904K),
[Perm : 376K->376K(12288K)], 0.0293000 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
[Full GC [Tenured: 14335K->14335K(14336K), 0.0278675 secs] 19903K-
>19903K(19904K),
[Perm : 376K->376K(12288K)], 0.0279215 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
[Full GC [Tenured: 14335K->14335K(14336K), 0.0348408 secs] 19903K-
>19889K(19904K),
[Perm : 376K->371K(12288K)], 0.0348945 secs] [Times: user=0.05 sys=0.00,
real=0.05 secs]
[Full GC [Tenured: 14335K->14335K(14336K), 0.0299813 secs] 19903K-
>19903K(19904K),
[Perm : 371K->371K(12288K)], 0.0300349 secs] [Times: user=0.01 sys=0.00,
real=0.02 secs]
[Full GC [Tenured: 14335K->14335K(14336K), 0.0298178 secs] 19903K-
>19903K(19904K),
[Perm : 371K->371K(12288K)], 0.0298688 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space[Full GC
[Tenured:
14335K->14335K(14336K), 0.0294953 secs] 19903K->19903K(19904K),
[Perm : 371K->371K(12288K)], 0.0295474 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
[Full GC [Tenured
: 14335K->14335K(14336K), 0.0287742 secs] 19903K->19903K(19904K),
[Perm : 371K->371K(12288K)], 0.0288239 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
[Full GC [Tenuredat GCTimeTest.main(GCTimeTest.java:16)
: 14335K->14335K(14336K), 0.0287102 secs] 19903K->19903K(19904K),
[Perm : 371K->371K(12288K)], 0.0287627 secs] [Times: user=0.03 sys=0.00,
real=0.03 secs]
Heap
 def new generation total 5568K, used 5567K [0x35c10000, 0x36210000,
0x36210000)
 eden space 4992K, 100% used [0x35c10000, 0x360f0000, 0x360f0000)
 from space 576K, 99% used [0x36180000, 0x3620ffe8, 0x36210000)
 to space 576K, 0% used [0x360f0000, 0x360f0000, 0x36180000)
 tenured generation total 14336K, used 14335K [0x36210000, 0x37010000,
0x37010000)
 the space 14336K, 99% used [0x36210000, 0x3700ffd8, 0x37010000, 0x37010000)
 compacting perm gen total 12288K, used 371K [0x37010000, 0x37c10000,
```

```
0x3b010000)
 the space 12288K, 3% used [0x37010000, 0x3706ce28, 0x3706d000, 0x37c10000)
 ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
 rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

通過清單 2 和清單 3 對比，可以發現通過設置一個較大的年輕代預留新對象，設置合理的 Survivor 區並且提供 Survivor 區的使用率，可以將年輕對象保存在年輕代。一般來說，Survivor 區的空間不夠，或者佔用量達到 50%時，就會使對象進入年老代 (不管它的年齡有多大)。清單 4 創建了 3 個對象，分別分配一定的內存空間。

清單 4. 不同大小內存分配

```
public class PutInEden2 {
 public static void main(String[] args){
 byte[] b1,b2,b3;
 b1=new byte[1024*512];
 b2=new byte[1024*1024*4];
 b3=new byte[1024*1024*4];
 b3=null;
 b3=new byte[1024*1024*4];
 }
}
```

使用參數-XX:+PrintGCDetails -Xmx1000M -Xms500M -Xmn100M -XX:SurvivorRatio=8 運行清單 4 所示代碼，輸出如清單 5 所示。

清單 5. 清單 4 運行輸出

```
Heap
 def new generation total 92160K, used 11878K [0x0f010000, 0x15410000,
0x15410000)
 eden space 81920K, 2% used [0x0f010000, 0x0f1a9a20, 0x14010000)
 from space 10240K, 99% used [0x14a10000, 0x1540fff8, 0x15410000)
 to space 10240K, 0% used [0x14010000, 0x14010000, 0x14a10000)
 tenured generation total 409600K, used 86434K [0x15410000, 0x2e410000,
0x4d810000)
 the space 409600K, 21% used [0x15410000, 0x1a878b18, 0x1a878c00, 0x2e410000)
 compacting perm gen total 12288K, used 2062K [0x4d810000, 0x4e410000,
0x51810000)
 the space 12288K, 16% used [0x4d810000, 0x4da13b18, 0x4da13c00, 0x4e410000)
No shared spaces configured.
```

清單 5 輸出的日誌顯示，年輕代分配了 8M，年老代也分配了 8M。我們可以嘗試加上-XX:TargetSurvivorRatio=90 參數，這樣可以提高 from 區的利用率，使 from 區使用到 90%時，再將對象送入年老代，運行清單 4 代碼，輸出如清單 6 所示。

清單 6. 修改運行參數後清單 4 輸出

```
Heap
 def new generation total 9216K, used 9215K [0x35c10000, 0x36610000,
0x36610000)
 eden space 8192K, 100% used [0x35c10000, 0x36410000, 0x36410000)
 from space 1024K, 99% used [0x36510000, 0x3660fc50, 0x36610000)
 to space 1024K, 0% used [0x36410000, 0x36410000, 0x36510000)
 tenured generation total 10240K, used 10239K [0x36610000, 0x37010000,
0x37010000)
 the space 10240K, 99% used [0x36610000, 0x3700ff70, 0x37010000, 0x37010000)
 compacting perm gen total 12288K, used 371K [0x37010000, 0x37c10000,
0x3b010000)
 the space 12288K, 3% used [0x37010000, 0x3706cd90, 0x3706ce00, 0x37c10000)
 ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
 rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

如果將 SurvivorRatio 設置為 2，將 b1 對象預存在年輕代。輸出如清單 7 所示。

清單 7. 再次修改運行參數後清單 4 輸出

```
Heap
 def new generation total 7680K, used 7679K [0x35c10000, 0x36610000,
0x36610000)
 eden space 5120K, 100% used [0x35c10000, 0x36110000, 0x36110000)
 from space 2560K, 99% used [0x36110000, 0x3638fff0, 0x36390000)
 to space 2560K, 0% used [0x36390000, 0x36390000, 0x36610000)
 tenured generation total 10240K, used 10239K [0x36610000, 0x37010000,
0x37010000)
 the space 10240K, 99% used [0x36610000, 0x3700fff0, 0x37010000, 0x37010000)
 compacting perm gen total 12288K, used 371K [0x37010000, 0x37c10000,
0x3b010000)
 the space 12288K, 3% used [0x37010000, 0x3706ce28, 0x3706d000, 0x37c10000)
 ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

如何讓大對象進入年老代

我們在大部分情況下都會選擇將對象分配在年輕代。但是，對於佔用內存較多的大對象而言，它的選擇可能就不是這樣的。因為大對象出現在年輕代很可能擾亂年輕代 GC，並破壞年輕代原有的對象結構。因為嘗試在年輕代分配大對象，很可能導致空間不足，為了有足夠的空間容納大對象，JVM 不得不將年輕代中的年輕對象挪到年老代。因為大對象佔用空間多，所以可能需要移動大量小的年輕對象進入年老代，這對 GC 相當不利。基於以上原因，可以將大對象直接分配到年老代，保持年輕代對象結構的完整性，這樣可以提高 GC 的效率。如果一個大對象同時又是一個短命的對象，假設這種情況出現很頻繁，那對於 GC 來說會是一場災難。原本應該用於存放永久對象的年老代，被短命的對象塞滿，這也意味著對堆空間進行了洗牌，擾亂了分代內存回收的基本思路。因此，在軟件開發過程中，應該儘可能避免使用短命的大對象。可以使用參數-XX:PetenureSizeThreshold 設置大對象直接進入年老代的閾值。當對象的大小超過這個值時，將直接在年老代分配。參數-XX:PetenureSizeThreshold 只對串行收集器和年輕代並行收集器有效，並行回收收集器不識別這個參數。

清單 8. 創建一個大對象

```
public class BigObj2Old {
 public static void main(String[] args){
 byte[] b;
 b = new byte[1024*1024];
 }
}
```

使用 JVM 參數-XX:+PrintGCDetails –Xmx20M –Xms20MB 運行，可以得到清單 9 所示日誌輸出。

清單 9. 清單 8 運行輸出

```
Heap
 def new generation total 6144K, used 1378K [0x35c10000, 0x362b0000,
0x362b0000)
 eden space 5504K, 25% used [0x35c10000, 0x35d689e8, 0x36170000)
 from space 640K, 0% used [0x36170000, 0x36170000, 0x36210000)
 to space 640K, 0% used [0x36210000, 0x36210000, 0x362b0000)
 tenured generation total 13696K, used 0K [0x362b0000, 0x37010000, 0x37010000)
 the space 13696K, 0% used [0x362b0000, 0x362b0000, 0x362b0200, 0x37010000)
 compacting perm gen total 12288K, used 374K [0x37010000, 0x37c10000,
0x3b010000)
 the space 12288K, 3% used [0x37010000, 0x3706dac8, 0x3706dc00, 0x37c10000)
 ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
 rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

可以看到該對象被分配在了年輕代，佔用了 25%的空間。如果需要將 1MB 以上的對象直接在年老代分配，設置-XX:PetenureSizeThreshold=1000000，程序運行後輸出如清單 10 所示。

清單 10. 修改運行參數後清單 8 輸出

```
Heap
 def new generation total 6144K, used 354K [0x35c10000, 0x362b0000, 0x362b0000)
 eden space 5504K, 6% used [0x35c10000, 0x35c689d8, 0x36170000)
 from space 640K, 0% used [0x36170000, 0x36170000, 0x36210000)
 to space 640K, 0% used [0x36210000, 0x36210000, 0x362b0000)
 tenured generation total 13696K, used 1024K [0x362b0000, 0x37010000,
0x37010000)
 the space 13696K, 7% used [0x362b0000, 0x363b0010, 0x363b0200, 0x37010000)
 compacting perm gen total 12288K, used 374K [0x37010000, 0x37c10000,
0x3b010000)
 the space 12288K, 3% used [0x37010000, 0x3706dac8, 0x3706dc00, 0x37c10000)
 ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
 rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

清單 10 裡面可以看到當滿 1MB 時進入到了年老代。

如何設置對象進入年老代的年齡

堆中的每一個對象都有自己的年齡。一般情況下，年輕對象存放在年輕代，年老對象存放在年老代。為了做到這點，

虛擬機為每個對象都維護一個年齡。如果對象在 Eden 區，經過一次 GC 後依然存活，則被移動到 Survivor 區中，對象年齡加 1。以後，如果對象每經過一次 GC 依然存活，則年齡再加 1。當對象年齡達到閾值時，就移入年老代，成為老年對象。這個閾值的最大值可以通過參數-XX:MaxTenuringThreshold 來設置，默認值是 15。雖然-XX:MaxTenuringThreshold 的值可能是 15 或者更大，但這不意味著新對象非要達到這個年齡才能進入年老代。事實上，對象實際進入年老代的年齡是虛擬機在運行時根據內存使用情況動態計算的，這個參數指定的是閾值年齡的最大值。即，實際晉陞年老代年齡等於動態計算所得的年齡與-XX:MaxTenuringThreshold 中較小的那個。清單 11 所示代碼為 3 個對象申請了若干內存。

清單 11. 申請內存

```
public class MaxTenuringThreshold {
 public static void main(String args[])
{
 byte[] b1,b2,b3;
 b1 = new byte[1024*512];
 b2 = new byte[1024*1024*2];
 b3 = new byte[1024*1024*4];
 b3 = null;
 b3 = new byte[1024*1024*4];
 }
}
```

參數設置為：-XX:+PrintGCDetails -Xmx20M -Xms20M -Xmn10M -XX:SurvivorRatio=2

運行清單 11 所示代碼，輸出如清單 12 所示。

清單 12. 清單 11 運行輸出

```
[GC [DefNew: 2986K->690K(7680K), 0.0246816 secs] 2986K->2738K(17920K),
 0.0247226 secs] [Times: user=0.00 sys=0.02, real=0.03 secs]
[GC [DefNew: 4786K->690K(7680K), 0.0016073 secs] 6834K->2738K(17920K),
0.0016436 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation total 7680K, used 4888K [0x35c10000, 0x36610000,
0x36610000)
 eden space 5120K, 82% used [0x35c10000, 0x36029a18, 0x36110000)
 from space 2560K, 26% used [0x36110000, 0x361bc950, 0x36390000)
 to space 2560K, 0% used [0x36390000, 0x36390000, 0x36610000)
 tenured generation total 10240K, used 2048K [0x36610000, 0x37010000,
0x37010000)
 the space 10240K, 20% used [0x36610000, 0x36810010, 0x36810200, 0x37010000)
 compacting perm gen total 12288K, used 374K [0x37010000, 0x37c10000,
0x3b010000)
 the space 12288K, 3% used [0x37010000, 0x3706db50, 0x3706dc00, 0x37c10000)
 ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
 rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

更改參數為-XX:+PrintGCDetails -Xmx20M -Xms20M -Xmn10M -XX:SurvivorRatio=2 -XX:MaxTenuringThreshold=1，運行清單 11 所示代碼，輸出如清單 13 所示。

清單 13. 修改運行參數後清單 11 輸出

```
[GC [DefNew: 2986K->690K(7680K), 0.0047778 secs] 2986K->2738K(17920K),
 0.0048161 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC [DefNew: 4888K->0K(7680K), 0.0016271 secs] 6936K->2738K(17920K),
0.0016630 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Heap
 def new generation total 7680K, used 4198K [0x35c10000, 0x36610000,
0x36610000)
 eden space 5120K, 82% used [0x35c10000, 0x36029a18, 0x36110000)
 from space 2560K, 0% used [0x36110000, 0x36110088, 0x36390000)
 to space 2560K, 0% used [0x36390000, 0x36390000, 0x36610000)
 tenured generation total 10240K, used 2738K [0x36610000, 0x37010000,
0x37010000)
 the space 10240K, 26% used [0x36610000, 0x368bc890, 0x368bca00, 0x37010000)
 compacting perm gen total 12288K, used 374K [0x37010000, 0x37c10000,
0x3b010000)
 the space 12288K, 3% used [0x37010000, 0x3706db50, 0x3706dc00, 0x37c10000)
 ro space 10240K, 51% used [0x3b010000, 0x3b543000, 0x3b543000, 0x3ba10000)
 rw space 12288K, 55% used [0x3ba10000, 0x3c0ae4f8, 0x3c0ae600, 0x3c610000)
```

清單 13 所示，第一次運行時 b1 對象在程序結束後依然保存在年輕代。第二次運行前，我們減小了對象晉陞年老代的年齡，設置為 1。即，所有經過一次 GC 的對象都可以直接進入年老代。程序運行後，可以發現 b1 對象已經被分配到年老代。如果希望對象儘可能長時間地停留在年輕代，可以設置一個較大的閾值。

穩定的 Java 堆 VS 動盪的 Java 堆

一般來說，穩定的堆大小對垃圾回收是有利的。獲得一個穩定的堆大小的方法是使-Xms 和-Xmx 的大小一致，即最大堆和最小堆 (初始堆) 一樣。如果這樣設置，系統在運行時堆大小理論上是恆定的，穩定的堆空間可以減少 GC 的次數。因此，很多服務端應用都會將最大堆和最小堆設置為相同的數值。但是，一個不穩定的堆並非毫無用處。穩定的堆大小雖然可以減少 GC 次數，但同時也增加了每次 GC 的時間。讓堆大小在一個區間中震盪，在系統不需要使用大內存時，壓縮堆空間，使 GC 應對一個較小的堆，可以加快單次 GC 的速度。基於這樣的考慮，JVM 還提供了兩個參數用於壓縮和擴展堆空間。

-XX:MinHeapFreeRatio 參數用來設置堆空間最小空閒比例，默認值是 40。當堆空間的空閒內存小於這個數值時，JVM 便會擴展堆空間。

-XX:MaxHeapFreeRatio 參數用來設置堆空間最大空閒比例，默認值是 70。當堆空間的空閒內存大於這個數值時，便會壓縮堆空間，得到一個較小的堆。

當-Xmx 和-Xms 相等時，-XX:MinHeapFreeRatio 和-XX:MaxHeapFreeRatio 兩個參數無效。

清單 14. 堆大小設置

```
import java.util.Vector;

public class HeapSize {
 public static void main(String args[]) throws
InterruptedException{
 Vector v = new Vector();
 while(true){
 byte[] b = new byte[1024*1024];
 v.add(b);
 if(v.size() == 10){
 v = new Vector();
 }
 Thread.sleep(1);
 }
 }
}
```

清單 14 所示代碼是測試-XX:MinHeapFreeRatio 和-XX:MaxHeapFreeRatio 的作用，設置運行參數為-XX:+PrintGCDetails -Xms10M -Xmx40M -XX:MinHeapFreeRatio=40 -XX:MaxHeapFreeRatio=50 時，輸出如清單 15 所示。

清單 15. 修改運行參數後清單 14 輸出

```
[GC [DefNew: 2418K->178K(3072K), 0.0034827 secs] 2418K->2226K(9920K),
 0.0035249 secs] [Times: user=0.00 sys=0.00, real=0.03 secs]
[GC [DefNew: 2312K->0K(3072K), 0.0028263 secs] 4360K->4274K(9920K),
0.0029905 secs] [Times: user=0.00 sys=0.00, real=0.03 secs]
[GC [DefNew: 2068K->0K(3072K), 0.0024363 secs] 6342K->6322K(9920K),
0.0024836 secs] [Times: user=0.00 sys=0.00, real=0.03 secs]
[GC [DefNew: 2061K->0K(3072K), 0.0017376 secs][Tenured: 8370K-
>8370K(8904K),
0.1392692 secs] 8384K->8370K(11976K), [Perm : 374K->374K(12288K)],
0.1411363 secs] [Times: user=0.00 sys=0.02, real=0.16 secs]
[GC [DefNew: 5138K->0K(6336K), 0.0038237 secs] 13508K->13490K(20288K),
0.0038632 secs] [Times: user=0.00 sys=0.00, real=0.03 secs]
```

改用參數：-XX:+PrintGCDetails -Xms40M -Xmx40M -XX:MinHeapFreeRatio=40 -XX:MaxHeapFreeRatio=50，運行輸出如清單 16 所示。

清單 16. 再次修改運行參數後清單 14 輸出

```
[GC [DefNew: 10678K->178K(12288K), 0.0019448 secs] 10678K-
>178K(39616K),
 0.0019851 secs] [Times: user=0.00 sys=0.00, real=0.03 secs]
[GC [DefNew: 10751K->178K(12288K), 0.0010295 secs] 10751K-
>178K(39616K),
0.0010697 secs] [Times: user=0.00 sys=0.00, real=0.02 secs]
[GC [DefNew: 10493K->178K(12288K), 0.0008301 secs] 10493K-
>178K(39616K),
0.0008672 secs] [Times: user=0.00 sys=0.00, real=0.02 secs]
[GC [DefNew: 10467K->178K(12288K), 0.0008522 secs] 10467K-
>178K(39616K),
0.0008905 secs] [Times: user=0.00 sys=0.00, real=0.02 secs]
[GC [DefNew: 10450K->178K(12288K), 0.0008964 secs] 10450K-
>178K(39616K),
0.0009339 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
[GC [DefNew: 10439K->178K(12288K), 0.0009876 secs] 10439K-
>178K(39616K),
0.0010279 secs] [Times: user=0.00 sys=0.00, real=0.02 secs]
```

從清單 16 可以看出，此時堆空間的垃圾回收穩定在一個固定的範圍。在一個穩定的堆中，堆空間大小始終不變，每次 GC 時，都要應對一個 40MB 的空間。因此，雖然 GC 次數減小了，但是單次 GC 速度不如一個震盪的堆。

增大吞吐量提升系統性能

吞吐量優先的方案將會儘可能減少系統執行垃圾回收的總時間，故可以考慮關注系統吞吐量的並行回收收集器。在擁有高性能的計算機上，進行吞吐量優先優化，可以使用參數：

```
java –Xmx3800m –Xms3800m –Xmn2G –Xss128k –
XX:+UseParallelGC
   –XX:ParallelGC-Threads=20 –XX:+UseParallelOldGC
```

–Xmx380m –Xms3800m：設置 Java 堆的最大值和初始值。一般情況下，為了避免堆內存的頻繁震盪，導致系統性能下降，我們的做法是設置最大堆等於最小堆。假設這裡把最小堆減少為最大堆的一半，即 1900m，那麼 JVM 會儘可能在 1900MB 堆空間中運行，如果這樣，發生 GC 的可能性就會比較高；

-Xss128k：減少線程棧的大小，這樣可以使剩餘的系統內存支持更多的線程；

-Xmn2g：設置年輕代區域大小為 2GB；

–XX:+UseParallelGC：年輕代使用並行垃圾回收收集器。這是一個關注吞吐量的收集器，可以儘可能地減少 GC 時間。

–XX:ParallelGC-Threads：設置用於垃圾回收的線程數，通常情況下，可以設置和 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:+UseParallelOldGC：設置年老代使用並行回收收集器。

嘗試使用大的內存分頁

CPU 是通過尋址來訪問內存的。32 位 CPU 的尋址寬度是 0~0xFFFFFFFF ，計算後得到的大小是 4G，也就是說可支持的物理內存最大是 4G。但在實踐過程中，碰到了這樣的問題，程序需要使用 4G 內存，而可用物理內存小於

4G，導致程序不得不降低內存佔用。為瞭解決此類問題，現代 CPU 引入了 MMU（Memory Management Unit 內存管理單元）。MMU 的核心思想是利用虛擬地址替代物理地址，即 CPU 尋址時使用虛址，由 MMU 負責將虛址映射為物理地址。MMU 的引入，解決了對物理內存的限制，對程序來說，就像自己在使用 4G 內存一樣。內存分頁 (Paging) 是在使用 MMU 的基礎上，提出的一種內存管理機制。它將虛擬地址和物理地址按固定大小（4K）分割成頁 (page) 和頁帧 (page frame)，並保證頁與頁帧的大小相同。這種機制，從數據結構上，保證了訪問內存的高效，並使 OS 能支持非連續性的內存分配。在程序內存不夠用時，還可以將不常用的物理內存頁轉移到其他存儲設備上，比如磁盤，這就是大家耳熟能詳的虛擬內存。

在 Solaris 系統中，JVM 可以支持 Large Page Size 的使用。使用大的內存分頁可以增強 CPU 的內存尋址能力，從而提升系統的性能。

```
1  java -Xmx2506m -Xms2506m -Xmn1536m -Xss128k -XX:++UseParallelGC
2   -XX:ParallelGCThreads=20 -XX:+UseParallelOldGC -
   XX:+LargePageSizeInBytes=256m
```

–XX:+LargePageSizeInBytes：設置大頁的大小。

過大的內存分頁會導致 JVM 在計算 Heap 內部分區（perm, new, old）內存佔用比例時，會出現超出正常值的劃分，最壞情況下某個區會多佔用一個頁的大小。

使用非佔有的垃圾回收器

為降低應用軟件的垃圾回收時的停頓，首先考慮的是使用關注系統停頓的 CMS 回收器，其次，為了減少 Full GC 次數，應儘可能將對象預留在年輕代，因為年輕代 Minor GC 的成本遠遠小於年老代的 Full GC。

```
1  java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20
2   -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+SurvivorRatio=8 -
3  XX:TargetSurvivorRatio=90
    -XX:MaxTenuringThreshold=31
```

–XX:ParallelGCThreads=20：設置 20 個線程進行垃圾回收；

–XX:+UseParNewGC：年輕代使用並行回收器；

–XX:+UseConcMarkSweepGC：年老代使用 CMS 收集器降低停頓；

–XX:+SurvivorRatio：設置 Eden 區和 Survivor 區的比例為 8:1。稍大的 Survivor 空間可以提高在年輕代回收生命週期較短的對象的可能性，如果 Survivor 不夠大，一些短命的對象可能直接進入年老代，這對系統來說是不利的。

–XX:TargetSurvivorRatio=90：設置 Survivor 區的可使用率。這裡設置為 90%，則允許 90%的 Survivor 空間被使用。默認值是 50%。故該設置提高了 Survivor 區的使用率。當存放的對象超過這個百分比，則對象會向年老代壓縮。因此，這個選項更有助於將對象留在年輕代。

–XX:MaxTenuringThreshold：設置年輕對象晉陞到年老代的年齡。默認值是 15 次，即對象經過 15 次 Minor GC 依然存活，則進入年老代。這裡設置為 31，目的是讓對象儘可能地保存在年輕代區域。

結束語

通過本文的學習，讀者瞭解了如何將新對象預留在年輕代、如何讓大對象進入年老代、如何設置對象進入年老代的年齡、穩定的 Java 堆 VS 動盪的 Java 堆、增大吞吐量提升系統性能、嘗試使用大的內存分頁、使用非佔有的垃圾回收器等主題，通過實例及對應輸出解釋的形式讓讀者對於 JVM 優化有一個初步認識。如其他文章相同的觀點，沒有哪一條優化是固定不變的，讀者需要自己判斷、實踐後才能找到正確的道路。