JVM參數整理

jvm參數較多，常用的就是之前學習筆記整理的關於OOM異常的調整。

參數分類含義：

標準參數：例如javap -verbose

X參數：所有的這類參數都以-X開始，例如常用的-Xmx,

- 對於布爾類型的參數，我們有」+」或」-「，然後才設置JVM選項的實際名稱。例如，-XX:+用於激活選項，而-XX:-用於註銷選項。
- 對於需要非布爾值的參數，如string或者integer，我們先寫參數的名稱，後面加上」=」，最後賦值。例如，-XX:=給賦值。

1 **-version**

這個不細介紹了吧，從一開始學習安裝jdk 開始就用這個命令。

注意一個點，HotSpot默認的運行模式混合模式(mixed mode)，意味著JVM在運行時可以動態的把字節碼編譯為本地代碼，讓JIT編譯器充分發揮其動態潛力。

2-XX:+PrintCommandLineFlags

這個參數讓JVM列印出那些已經被用戶或者JVM設置過的詳細的XX參數的名稱和值。

3.內存調優有關：

-Xms and -Xmx

-Xms和-Xmx可以說是最流行的JVM參數，它們可以允許我們指定JVM的初始和最大堆內存大小。

-Xms和-Xmx實際上是-XX:InitialHeapSize和-XX:MaxHeapSize的縮寫。

-XX:+HeapDumpOnOutOfMemoryError and -XX:HeapDumpPath

之前學習筆記關於mat分析dump提到過分析堆內存快照（Heap Dump）是一個很好的定位手段。設置-XX:+HeapDumpOnOutOfMemoryError 讓JVM在發生內存溢出時自動的生成堆內存快照。默認情況下，堆內存快照會保存在JVM的啟動目錄下名為java_pid.hprof 的文件里（在這裡就是JVM進程的進程號）。也可以通過設置-XX:HeapDumpPath=來改變默認的堆內存快照生成路徑，可以是相對或者絕對路徑。注意一點：dump文件很大，通常在1G以上，磁碟空間要夠。

-XX:OnOutOfMemoryError

當內存溢發生時，我們甚至可以可以執行一些指令。

-XX:PermSize and -XX:MaxPermSize

-XX:MaxPermSize 用於設置永久代大小的最大值，-XX:PermSize 用於設置永久代初始大小。

-XX:InitialCodeCacheSize and -XX:ReservedCodeCacheSize

代碼緩存區。在JVM里，Java字節碼被解釋運行，但是它沒有直接運行本地代碼快。為了提高性能，Oracle Hotspot VM會尋找字節碼的」熱點」區域，它指頻繁被執行的代碼，然後編譯成本地代碼。這些本地代碼會被保存在堆外內存的代碼緩存區。
通常我們注意不到這個區域，如果代碼緩存被占滿，JVM會列印出一條警告消息，並切換到interpreted-only 模式：JIT編譯器被停用，字節碼將不再會被編譯成機器碼。因此，應用程式將繼續運行，但運行速度會降低一個數量級。

所以說參數還得跟之前學習jvm結合起來看。

4.新生代GC回收

-XX:NewSize and -XX:MaxNewSize

可以通過參數指定新生代大小。設置 XX:MaxNewSize 參數時，應該考慮到新生代只是整個堆的一部分，新生代設置的越大，老年代區域就會減少。一般不允許新生代比老年代還大，因為要考慮GC時最壞情況，所有對象都晉升到老年代。(譯者:會發生OOM錯誤) -XX:MaxNewSize 最大可以設置為-Xmx/2 .

先寫到這裡，明天接著整理

XX:NewRatio
可以設置新生代和老年代的相對大小。這種方式的優點是新生代大小會隨著整個堆大小動態擴展。參數 -XX:NewRatio 設置老年代與新生代的比例。例如 -XX:NewRatio=3 指定老年代/新生代為3/1. 老年代占堆大小的 3/4 ，新生代占 1/4 .

如果針對新生代,同時定義絕對值和相對值,絕對值將起作用。下面例子：
```
$ java -XX:NewSize=32m -XX:MaxNewSize=512m -XX:NewRatio=3 MyApp
```

以上設置, JVM 會嘗試為新生代分配四分之一的堆大小，但不會小於32MB或大於521MB

-XX:SurvivorRatio
參數 -XX:SurvivorRatio 與 -XX:NewRatio 類似，作用於新生代內部區域。-XX:SurvivorRatio 指定伊甸園區(Eden)與倖存區大小比例. 例如, -XX:SurvivorRatio=10 表示伊甸園區(Eden)是 倖存區To 大小的10倍(也是倖存區From的10倍).所以,伊甸園區(Eden)占新生代大小的10/12, 倖存區From和倖存區To 每個占新生代的1/12 .注意,兩個倖存區永遠是一樣大的..

我們希望最小化短命對象晉升到老年代的數量，同時也希望最小化新生代GC 的次數和持續時間.我們需要找到針對當前應用的折中方案, 尋找適合方案的起點是 了解當前應用中對象的年齡分布情況。

-XX:InitialTenuringThreshold, -XX:MaxTenuringThreshold and -XX:TargetSurvivorRatio
參數 -XX:+PrintTenuringDistribution 輸出中的部分值可以通過其它參數控制。通過 -XX:InitialTenuringThreshold 和 -XX:MaxTenuringThreshold 可以設定老年代閥值的初始值和最大值。另外,可以通過參數 -XX:TargetSurvivorRatio 設定倖存區的目標使用率.例如，-XX:MaxTenuringThreshold=10 -XX:TargetSurvivorRatio=90 設定老年代閥值的上限為10,倖存區空間目標使用率為90%。

有多種方式,設置新生代行為，沒有通用準則。我們必須清楚以下2中情況：
1 如果從年齡分布中發現，有很多對象的年齡持續增長，在到達老年代閥值之前。這表示 -XX:MaxTenuringThreshold 設置過大
2 如果 -XX:MaxTenuringThreshold 的值大於1，但是很多對象年齡從未大於1.應該看下倖存區的目標使用率。如果倖

存區使用率從未到達，這表示對象都被GC回收，這正是我們想要的。 如果倖存區使用率經常達到，有些年齡超過1的對象被移動到老年代中。這種情況，可以嘗試調整倖存區大小或目標使用率。

-XX:+PrintTenuringDistribution
參數 -XX:+PrintTenuringDistribution 指定JVM 在每次新生代GC時，輸出倖存區中對象的年齡分布。例如:

```
Desired survivor size 75497472 bytes, new threshold 15 (max 15)
- age 1: 19321624 bytes, 19321624 total
- age 2: 79376 bytes, 19401000 total
- age 3: 2904256 bytes, 22305256 total
```

第一行說明倖存區To大小為 75 MB. 也有關於老年代閥值(tenuring threshold)的信息, 老年代閥值，意思是對象從新生代移動到老年代之前，經過幾次GC(即, 對象晉升前的最大年齡). 上例中,老年代閥值為15,最大也是15.之後行表示，對於小於老年代閥值的每一個對象年齡,本年齡中對象所占字節 (如果當前年齡沒有對象,這一行會忽略).

當調整堆和GC設置時，我們總是應該同時考慮新生代和老年代。

5 老年代GC

1. 吞吐量越高算法越好
2. 暫停時間越短算法越好

首先讓我們來明確垃圾收集(GC)中的兩個術語:吞吐量(throughput)和暫停時間(pause times)。不幸的是」高吞吐量」和」低暫停時間」是一對相互競爭的目標（矛盾）。對於年老代，HotSpot虛擬機提供兩類垃圾收集算法(除了新的G1垃圾收集算法)，第一類算法試圖最大限度地提高吞吐量，而第二類算法試圖最小化暫停時間。 下面是跟面向吞吐量垃圾收集算法有關的重要JVM配置參數。

-XX:+UseSerialGC

我們使用該標誌來激活串行垃圾收集器，例如單線程面向吞吐量垃圾收集器。 無論年輕代還是年老代都將只有一個線程執行垃圾收集。 該標誌被推薦用於只有單個可用處理器核心的JVM。 在這種情況下，使用多個垃圾收集線程甚至會適得其反，因為這些線程將爭用CPU資源，造成同步開銷，卻從未真正並行運行。

-XX:+UseParallelGC

有了這個標誌，我們告訴JVM使用多線程並行執行年輕代垃圾收集。 在我看來，Java 6中不應該使用該標誌因為-XX:+UseParallelOldGC顯然更合適。 需要注意的是Java 7中該情況改變了一點(詳見本概述)，就是-XX:+UseParallelGC能達到-XX:+UseParallelOldGC一樣的效果。

-XX:+UseParallelOldGC

該標誌的命名有點不巧，因為」老」聽起來像」過時」。 然而，」老」實際上是指年老代，這也解釋了為什麼-XX:+UseParallelOldGC要優於-XX:+UseParallelGC：除了激活年輕代並行垃圾收集，也激活了年老代並行垃圾收集。 當期望高吞吐量，並且JVM有兩個或更多可用處理器核心時，我建議使用該標誌。
作為旁註，HotSpot的並行面向吞吐量垃圾收集算法通常稱為」吞吐量收集器」，因為它們旨在通過並行執行來提高吞吐量。

通過-XX:ParallelGCThreads=我們可以指定並行垃圾收集的線程數量。 例如，-XX:ParallelGCThreads=6表示每次並行垃圾收集將有6個線程執行。 如果不明確設置該標誌，虛擬機將使用基於可用(虛擬)處理器數量計算的默認值。 決定因素是由Java Runtime。availableProcessors方法的返回值N，如果N<=8，並行垃圾收集器將使用N個垃圾收集線程，如果N>8個可用處理器，垃圾收集線程數量應為3+5N/8。
當JVM獨占地使用系統和處理器時使用默認設置更有意義。 但是，如果有多個JVM(或其他耗CPU的系統)在同一台機器上運行，我們應該使用-XX:ParallelGCThreads來減少垃圾收集線程數到一個適當的值。 例如，如果4個以伺服器方式運行的JVM同時跑在在一個具有16核處理器的機器上，設置-XX:ParallelGCThreads=4是明智的，它能使不同

JVM的垃圾收集器不會相互干擾。-XX:-UseAdaptiveSizePolicy

吞吐量垃圾收集器提供了一個有趣的(但常見，至少在現代JVM上)機制以提高垃圾收集配置的用戶友好性。 這種機制被看做是HotSpot在Java 5中引入的」人體工程學」概念的一部分。 通過人體工程學，垃圾收集器能將堆大小動態變動像GC設置一樣應用到不同的堆區域，只要有證據表明這些變動將能提高GC性能。 「提高GC性能」的確切含義可以由用戶通過-XX:GCTimeRatio和-XX:MaxGCPauseMillis(見下文)標記來指定。
重要的是要知道人體工程學是默認激活的。 這很好，因為自適應行為是JVM最大優勢之一。 不過，有時我們需要非常清楚對於特定應用什麼樣的設置是最合適的，在這些情況下，我們可能不希望JVM混亂我們的設置。 每當我們發現處於這種情況時，我們可以考慮通過-XX:-UseAdaptiveSizePolicy停用一些人體工程學。

通過-XX:GCTimeRatio=我們告訴JVM吞吐量要達到的目標值。 更準確地說，-XX:GCTimeRatio=N指定目標應用程式線程的執行時間(與總的程序執行時間)達到N/(N+1)的目標比值。 例如，通過-XX:GCTimeRatio=9我們要求應用程式線程在整個執行時間中至少9/10是活動的(因此，GC線程占用其餘1/10)。 基於運行時的測量，JVM將會嘗試修改堆和GC設置以期達到目標吞吐量。 -XX:GCTimeRatio的默認值是99，也就是說，應用程式線程應該運行至少99%的總執行時間。告訴JVM最大暫停時間的目標值(以毫秒為單位)。 在運行時，吞吐量收集器計算在暫停期間觀察到的統計數據(加權平均和標準偏差)。 如果統計表明正在經歷的暫停其時間存在超過目標值的風險時，JVM會修改堆和GC設置以降低它們。 需要注意的是，年輕代和年老代垃圾收集的統計數據是分開計算的，還要注意，默認情況下，最大暫停時間沒有被設置。

6.CMS收集器參數

HotSpot JVM的並發標記清理收集器(CMS收集器)的主要目標就是：低應用停頓時間。為了實現安全且正確的並發執行，CMS收集器的GC周期由6個階段組成。其中4個階段(名字以Concurrent開始的)與實際的應用程式是並發執行的，而其他2個階段需要暫停應用程式線程。

1. 初始標記：為了收集應用程式的對象引用需要暫停應用程式線程，該階段完成後，應用程式線程再次啟動。
2. 並發標記：從第一階段收集到的對象引用開始，遍歷所有其他的對象引用。
3. 並發預清理：改變當運行第二階段時，由應用程式線程產生的對象引用，以更新第二階段的結果。
4. 重標記：由於第三階段是並發的，對象引用可能會發生進一步改變。因此，應用程式線程會再一次被暫停以更新這些變化，並且在進行實際的清理之前確保一個正確的對象引用視圖。這一階段十分重要，因為必須避免收集到仍被引用的對象。
5. 並發清理：所有不再被應用的對象將從堆里清除掉。
6. 並發重置：收集器做一些收尾的工作，以便下一次GC周期能有一個乾淨的狀態。

當我們在真實的應用中使用CMS收集器時，我們會面臨兩個主要的挑戰，可能需要進行調優：

1. 堆碎片
2. 對象分配率高

當這些情形之一出現在實踐中時(經常會出現在生產系統中)，經常被證實是老年代有大量不必要的對象。一個可行的辦法就是增加年輕代的堆大小，以防止年輕代短生命的對象提前進入老年代。另一個辦法就似乎利用分析器，快照運行系統的堆轉儲，並且分析過度的對象分配，找出這些對象，最終減少這些對象的申請。

下面我看看大多數與CMS收集器調優相關的JVM標誌參數。

-XX：+UseConcMarkSweepGC

該標誌首先是激活CMS收集器。默認HotSpot JVM使用的是並行收集器。

-XX：UseParNewGC

當使用CMS收集器時，該標誌激活年輕代使用多線程並行執行垃圾回收。這令人很驚訝，我們不能簡單在並行收集器中重用-XX：UserParNewGC標誌，因為概念上年輕代用的算法是一樣的。然而，對於CMS收集器，年輕代GC算法和老年代GC算法是不同的，因此年輕代GC有兩種不同的實現，並且是兩個不同的標誌。

注意最新的JVM版本，當使用-XX：+UseConcMarkSweepGC時，-XX：UseParNewGC會自動開啟。因此，如果年輕代的並行GC不想開啟，可以通過設置-XX：-UseParNewGC來關掉。

-XX：+CMSConcurrentMTEnabled

當該標誌被啟用時，並發的CMS階段將以多線程執行(因此，多個GC線程會與所有的應用程式線程並行工作)。該標誌已經默認開啟，如果順序執行更好，這取決於所使用的硬體，多線程執行可以通過-XX：-CMSConcurremntMTEnabled禁用。

-XX：ConcGCThreads

標誌-XX：ConcGCThreads=(早期JVM版本也叫-XX:ParallelCMSThreads)定義並發CMS過程運行時的線程數。比如value=4意味著CMS周期的所有階段都以4個線程來執行。儘管更多的線程會加快並發CMS過程，但其也會帶來額外的同步開銷。因此，對於特定的應用程式，應該通過測試來判斷增加CMS線程數是否真的能夠帶來性能的提升。ParallelGCThreads參數的值來計算出默認的並行CMS線程數。該公式是ConcGCThreads = (ParallelGCThreads + 3)/4。因此，對於CMS收集器， `-XX:ParallelGCThreads`標誌不僅影響「`stop-the-world`」垃圾收集階段，還影響並發階段。

總之，有不少方法可以配置CMS收集器的多線程執行。正是由於這個原因,建議第一次運行CMS收集器時使用其默認設置, 然後如果需要調優再進行測試。只有在生產系統中測量(或類生產測試系統)發現應用程式的暫停時間的目標沒有達到，就可以通過這些標誌應該進行GC調優。

-XX:CMSInitiatingOccupancyFraction

當堆滿之後，並行收集器便開始進行垃圾收集，例如，當沒有足夠的空間來容納新分配或提升的對象。對於CMS收集器，長時間等待是不可取的，因為在並發垃圾收集期間應用持續在運行(並且分配對象)。因此，為了在應用程式使用完內存之前完成垃圾收集周期，CMS收集器要比並行收集器更先啟動。

因為不同的應用會有不同對象分配模式，JVM會收集實際的對象分配(和釋放)的運行時數據，並且分析這些數據，來決定什麼時候啟動一次CMS垃圾收集周期。為了引導這一過程， JVM會在一開始執行CMS周期前作一些線索查找。該線索由 -XX:CMSInitiatingOccupancyFraction=來設置，該值代表老年代堆空間的使用率。比如，value=75意味著第一次CMS垃圾收集會在老年代被占用75%時被觸發。通常CMSInitiatingOccupancyFraction的默認值為68(之前很長時間的經歷來決定的)。

-XX：+UseCMSInitiatingOccupancyOnly

我們用-XX+UseCMSInitiatingOccupancyOnly標誌來命令JVM不基於運行時收集的數據來啟動CMS垃圾收集周期。而是，當該標誌被開啟時，JVM通過CMSInitiatingOccupancyFraction的值進行每一次CMS收集，而不僅僅是第一次。然而，請記住大多數情況下，JVM比我們自己能作出更好的垃圾收集決策。因此，只有當我們充足的理由(比如測試)並且對應用程式產生的對象的生命周期有深刻的認知時，才應該使用該標誌。

-XX:+CMSClassUnloadingEnabled

相對於並行收集器，CMS收集器默認不會對永久代進行垃圾回收。如果希望對永久代進行垃圾回收，可用設置標誌-XX:+CMSClassUnloadingEnabled。在早期JVM版本中，要求設置額外的標誌-

XX:+CMSPermGenSweepingEnabled。注意，即使沒有設置這個標誌，一旦永久代耗盡空間也會嘗試進行垃圾回收，但是收集不會是並行的，而再一次進行Full GC。

-XX:+CMSIncrementalMode

該標誌將開啟CMS收集器的增量模式。增量模式經常暫停CMS過程，以便對應用程式線程作出完全的讓步。因此，收集器將花更長的時間完成整個收集周期。因此，只有通過測試後發現正常CMS周期對應用程式線程干擾太大時，才應該使用增量模式。由於現代伺服器有足夠的處理器來適應並發的垃圾收集，所以這種情況發生得很少。

-XX:+ExplicitGCInvokesConcurrent and -XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses

如今,被廣泛接受的最佳實踐是避免顯式地調用GC(所謂的「系統GC」)，即在應用程式中調用system.gc。然而，這個建議是不管使用的GC算法的，值得一提的是，當使用CMS收集器時，系統GC將是一件很不幸的事，因為它默認會觸發一次Full GC。幸運的是，有一種方式可以改變默認設置。標誌-XX:+ExplicitGCInvokesConcurrent命令JVM無論什麼時候調用系統GC，都執行CMS GC，而不是Full GC。第二個標誌-XX:+ExplicitGCInvokesConcurrentAndUnloadsClasses保證當有系統GC調用時，永久代也被包括進CMS垃圾回收的範圍內。因此，通過使用這些標誌，我們可以防止出現意料之外的」stop-the-world」的系統GC。

-XX:+DisableExplicitGC

然而在這個問題上…這是一個很好提到- XX:+ DisableExplicitGC標誌的機會，該標誌將告訴JVM完全忽略系統的GC調用(不管使用的收集器是什麼類型)。對於我而言，該標誌屬於默認的標誌集合中，可以安全地定義在每個JVM上運行，而不需要進一步思考。

7 GC日誌參數

-XX:+PrintGC

參數-XX:+PrintGC（或者-verbose:gc）開啟了簡單GC日誌模式，為每一次新生代（young generation）的GC和每一次的Full GC列印一行信息

-XX:PrintGCDetails

就開啟了詳細GC日誌模式。在這種模式下，日誌格式和所使用的GC算法有關。

-XX:+PrintGCTimeStamps和-XX:+PrintGCDateStamps

使用-XX:+PrintGCTimeStamps可以將時間和日期也加到GC日誌中。表示自JVM啟動至今的時間戳會被添加到每一行中。

jvm參數比較多，除了與JVM內存模型有關外，還有GC有關。

本次沒有整理G1回收的參數，待專門整理。