[JAVA・初級]：GC-垃圾回收機制 - ImportNew

importnew.com/20354.html

原文出處： 吳士龍

意義

在C++中，對象所佔的內存在程序結束運行之前一直被佔用，在明確釋放之前不能分配給其它對象；而在Java中，當沒有對象引用指向原先分配給某個對象的內存時，該內存便成為垃圾。JVM的一個系統級線程會自動釋放該內存塊。垃圾收集意味著程序不再需要的對象是」無用信息」，這些信息將被丟棄。當一個對象不再被引用的時候，內存回收它佔領的空間，以便空間被後來的新對象使用。事實上，除了釋放沒用的對象，垃圾收集也可以清除內存記錄碎片。由於創建對象和垃圾收集器釋放丟棄對象所佔的內存空間，內存會出現碎片。碎片是分配給對象的內存塊之間的空閒內存洞。碎片整理將所佔用的堆內存移到堆的一端，JVM將整理出的內存分配給新的對象。

垃圾收集能自動釋放內存空間，減輕編程的負擔。這使Java 虛擬機具有一些優點。首先，它能使編程效率提高。在沒有垃圾收集機制的時候，可能要花許多時間來解決一個難懂的存儲器問題。在用Java語言編程的時候，靠垃圾收集機制可大大縮短時間。其次是它保護程序的完整性，垃圾收集是Java語言安全性策略的一個重要部份。

垃圾收集的一個潛在的缺點是它的開銷影響程序性能。Java虛擬機必須追蹤運行程序中有用的對象，而且最終釋放沒用的對象。這一個過程需要花費處理器的時間。其次垃圾收集算法的不完備性，早先採用的某些垃圾收集算法就不能保證100%收集到所有的廢棄內存。當然隨著垃圾收集算法的不斷改進以及軟硬件運行效率的不斷提升，這些問題都可以迎刃而解。

一般來說，Java開發人員可以不重視JVM中堆內存的分配和垃圾處理收集，但是，充分理解Java的這一特性可以讓我們更有效地利用資源。同時要注意finalize()方法是Java的缺省機制，有時為確保對象資源的明確釋放，可以編寫自己的finalize方法。

原理

關於垃圾收集器，在學習GC前，你應該知道一個技術名詞:這個詞是「stop-the-world。「無論你選擇哪種GC算法，Stop-the-world都會發生。Stop-the-world意味著JVM停止應用程序，而去進行垃圾回收。當stop-the-world發生時，除了進行垃圾回收的線程，其他所有線程都將停止運行。被中斷的任務將在GC任務完成後恢復執行。GC調優往往意味著減少stop-the-world的時間。

分代垃圾收集

在Java代碼中，Java語言沒有顯式的提供分配內存和刪除內存的方法。一些開發人員將引用對象設置為null或者調用System.gc()來釋放內存。將引用對象設置為null沒有什麼大問題，但是調用system.gc()方法會大大的影響系統性能，絕對不能這個干。(謝天謝地，我還沒看到任何NHN開發者調用這個方法。)

在Java中，由於開發人員沒有在代碼中顯式刪除內存，所以垃圾收集器會去發現不需要（垃圾）的對象,然後刪除它們，釋放內存。這款垃圾收集器是基於以下兩個假設而創建的。（稱他們為前提條件更好，而不是假設。）

絕大多數對象在短時間內變得不可達，只有少量年老對象引用年輕對象。這些假設被稱為「弱代假說」。為了發揮這一假設的優勢，在HotSpot虛擬機中，物理的將內存分為兩個—年輕代(young generation)和老年代(old generation)。

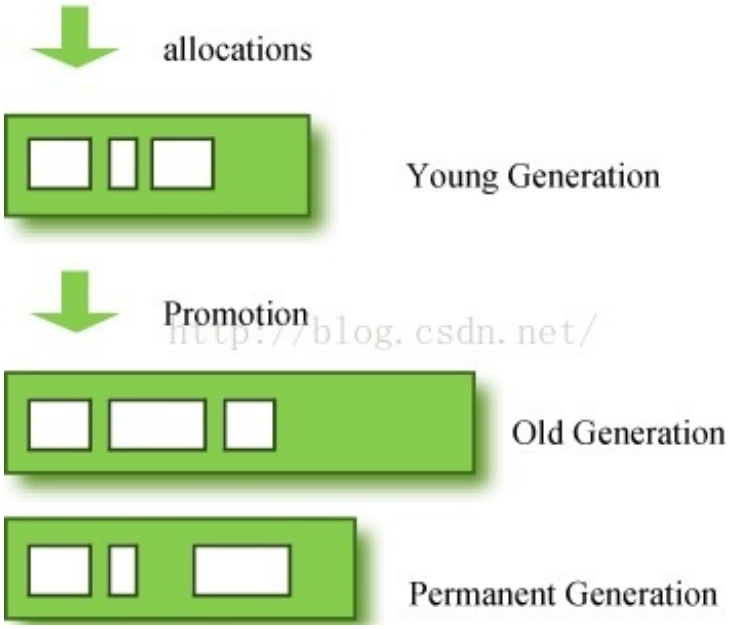
年輕代：新創建的對象都存放在這裡。因為大多數對象很快變得不可達，所以大多數對象在年輕代中創建，然後消失。當對象從這塊內存區域消失時，我們說發生了一次「minor GC」。

老年代：沒有變得不可達，存活下來的年輕代對象被覆制到這裡。這塊內存區域一般大於年輕代。因為它更大的規

模，GC發生的次數比在年輕代的少。對象從老年代消失時，我們說「major GC」（或「full GC」）發生了。

我們看一下這幅圖：

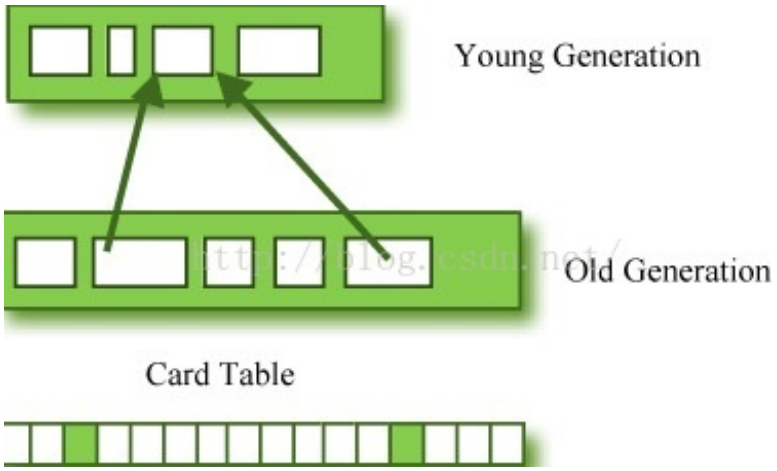
圖 1: GC區 & 數據流



上圖中的永久代(permanent generation)也稱為「方法區(method area)」，他存儲class對象和字符串常量。所以這塊內存區域絕對不是永久的存放從老年代存活下來的對象的。在這塊內存中有可能發生垃圾回收。發生在這裡垃圾回收也被稱為major GC。

一些人可能想知道:一個老年代的對象需要引用年輕代的對象，該怎麼辦？

為瞭解決這些問題，老年代中有一個被稱為「卡表(card table)」的東西，它是一個512 byte大小的塊。每當老年代的對象引用年輕代對象時，這種引用會被記錄在這張表格中。當垃圾回收發生在年輕代時，只需對這張表進行搜索以確定是否需要進行垃圾回收，而不是檢查老年代中的所有對象引用。這張表格用一個叫做「寫閘(write barrier)」的東西進行管理。「寫閘」是一種裝置，對minor GC有更好性能。雖然因為這種機制，會產生一些時間性能開銷，但降低了整體的GC時間。



年輕代組成部分

為了理解GC，我們學習一下年輕代，對象第一次創建發生在這塊內存區域。年輕代分為3塊，Eden區和2個Survivor區。

年輕代總共有3塊空間，其中2塊為Survivor區。各個空間的執行順序如下：

絕大多數新創建的對象分配在Eden區。

在Eden區發生一次GC後，存活的對象移到其中一個Survivor區。

在Eden區發生一次GC後，對象是存放到Survivor區，這個Survivor區已經存在其他存活的對象。

一旦一個Survivor區已滿，存活的對象移動到另外一個Survivor區。然後之前那個空間已滿Survivor區將置為空，沒有任何數據。

經過重複多次這樣的步驟後依舊存活的對象將被移到老年代。

通過檢查這些步驟，如你看到的樣子，其中一個Survivor區必須保持空。如果數據存在於兩個Survivor區，或兩個都沒使用，你可以將這個情況作為系統錯誤的一個標誌。

經過多次minor GC,數據被轉移到老年代過程如下面的圖表所示:

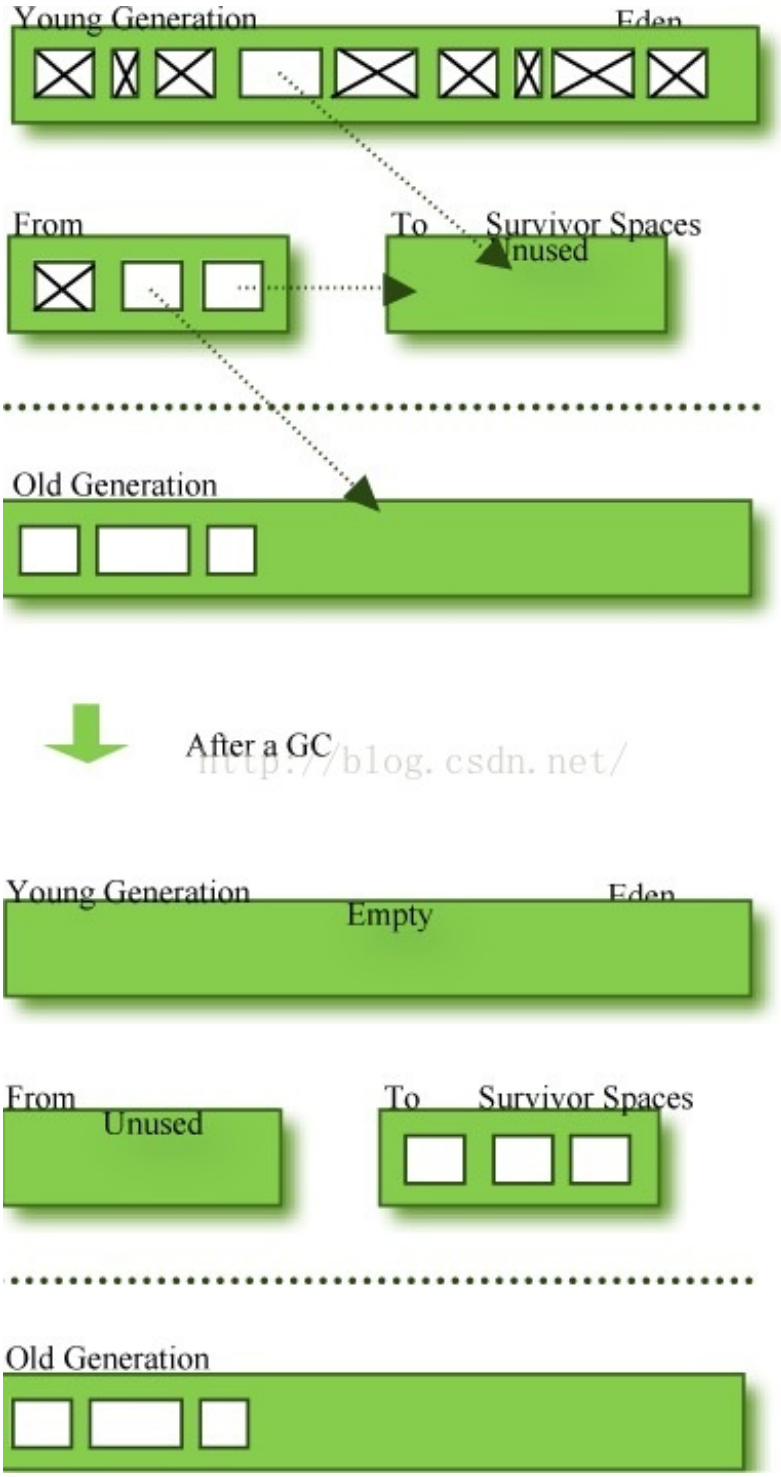


圖3: GC前和GC後

請注意，在HotSpot虛擬機中，使用兩種技術加快內存的分配。一個被稱為「指針碰撞(bump-the-pointer)」，另外一個被稱為「TLABs（線程本地分配緩衝）」。

指針碰撞技術跟蹤分配給Eden區上最新的對象。該對象將位於Eden 區的頂部。如果之後有一個對象被創建，只需檢查Eden區是否有足夠大的空間存放該對象。如果空間夠用，它將被放置在Eden區，存放在空間的頂部。因此，在創建新對象時，只需檢查最後被添加對象，看是否還有更多的內存空間允許分配。然而，如果考慮多線程的環境，則是另外一種情況。為了實現多線程環境下，在Eden 區線程安全的去創建保存對象，那麼必須加鎖，因此性能會下降。在HotSpot虛擬機中TLABs能夠解決這一問題。它允許每個線程在Eden區有自己的一小塊私有空間。因為每一個線程只能訪問自己的TLAB，所以在這個區域甚至可以使用無鎖的指針碰撞技術進行內存分配。

我們已經對年輕代有了一個快速的瀏覽。你不需要要記住我剛才提到的兩種技術。即便你不知道他們，也不會怎麼樣。但請務必記住:對象第一次被創建發生在Eden區，長期存活的對象被移動到老年代的Survivor區。

老年代GC

當老年代數據滿時，基本上會執行一次GC。執行程序根據不同GC類型而變化，所以如果你知道不同類型的垃圾收集器，會更容易理解垃圾回收過程。

在JDK7中，有5種垃圾收集器:

Serial收集器

Parallel收集器

Parallel Old收集器 (ParallelCompacting GC)收集器

Concurrent Mark & Sweep GC（or 「CMS」)收集器

Garbage First (G1) 收集器

其中，serial 收集器一定不能用於服務器端。這個收集器類型僅應用於單核CPU桌面電腦。使用serial收集器會顯著降低應用程序的性能。

現在讓我們來瞭解每個收集器類型。

Serial收集器

我們在前一段的解釋了在年輕代發生的垃圾回收算法類型。在老年代的GC使用算法被稱為「標記-清除-整理」。

該算法的第一步是在老年代標記存活的對象。

從頭開始檢查堆內存空間，並且只留下依然倖存的對象（清除）。

最後一步，從頭開始，順序地填滿堆內存空間,將存活的對象連續存放在一起，這樣堆分成兩部分：一邊有存放的對象，一邊沒有對象（整理）。

serial收集器應用於小的存儲器和少量的CPU。

Parallel收集器

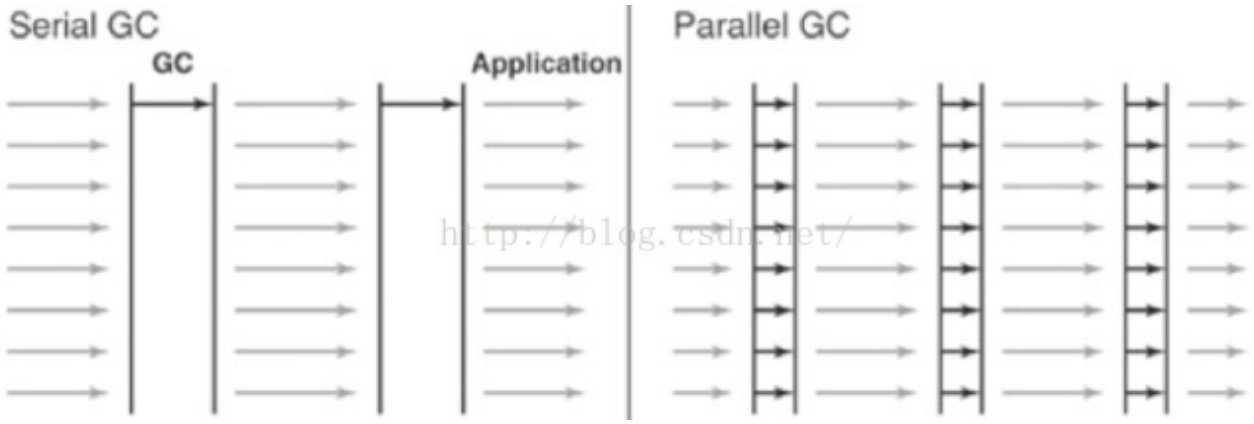


圖4: Serial收集器和 Parallel收集器的差異

從這幅圖中，你可以很容易看到Serial收集器和 Parallel收集器的差異。serial收集器只使用一個線程來處理的GC，而parallel收集器使用多線程並行處理GC，因此更快。當有足夠大的內存和大量芯數時，parallel收集器是有用的。它也被稱為「吞吐量優先垃圾收集器。」

ParallelOld 垃圾收集器

Parallel Old收集器是自JDK 5開始支持的。相比於parallel收集器，他們的唯一區別就是在老年代所執行的GC算法的不同。它執行三個步驟：標記-彙總-壓縮（mark – summary – compaction）。彙總步驟與清理的不同之處在於，其將依然倖存的對象分發到GC預先處理好的不同區域，算法相對清理來說略微複雜一點。

CMSGC

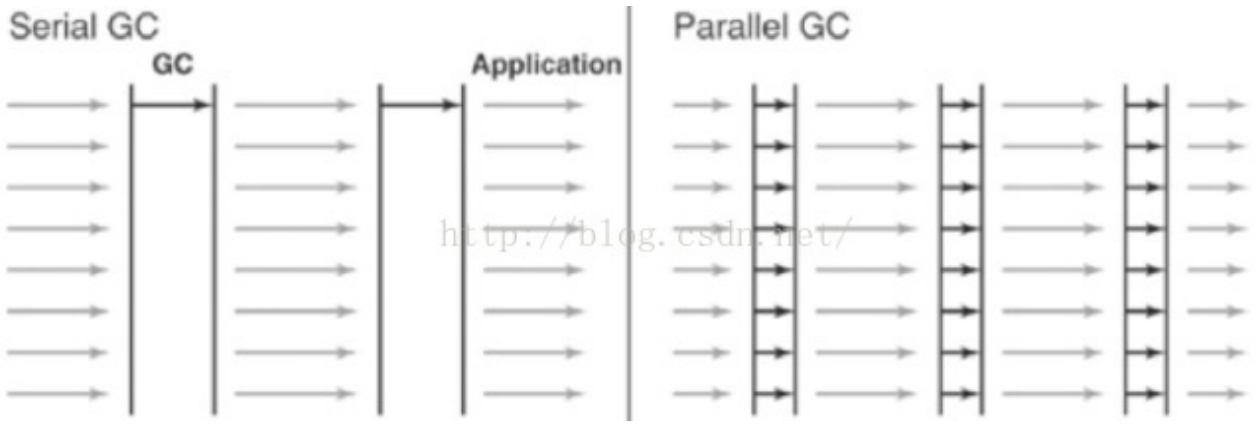


圖5: Serial GC & CMS GC

CMS垃圾收集器

如你在上圖看到的那樣, CMS垃圾收集器比之前我解釋的各種算法都要複雜很多。初始標記（initial mark）比較簡單。這一步驟只是查找距離類加載器最近的倖存對象。所以停頓時間非常短。之後的並發標記步驟，所有被倖存對象引用的對象會被確認是否已經被追蹤檢查。這一步的不同之處在於，在標記的過程中，其他的線程依然在執行。在重新標記步驟會修正那些在並發標記步驟中，因新增或者刪除對象而導致變動的那部分標記記錄。最後，在並發清除步驟，垃圾收集器執行。垃圾收集器進行垃圾收集時,其他線程的依舊在工作。一旦採取了這種GC類型，由於垃圾回收導致的停頓時間會極其短暫。CMS 收集器也被稱為低延遲垃圾收集器。它經常被用在那些對於響應時間要求十分苛刻的應用上。

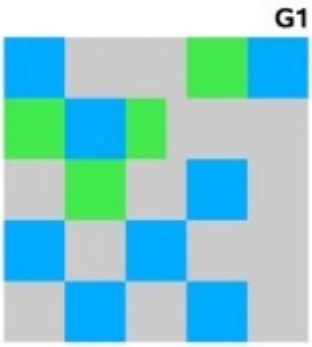
當然，這種GC類型在擁有stop-the-world時間很短的優點的同時，也有如下缺點：

它會比其他GC類型佔用更多的內存和CPU，默認情況下不支持壓縮步驟，在使用這個GC類型之前你需要慎重考慮。如果因為內存碎片過多而導致壓縮任務不得不執行，那麼stop-the-world的時間要比其他任何GC類型都長，你需要考慮壓縮任務的發生頻率以及執行時間。

G1GC

最後，我們來學習一下G1類型。

圖6: Layout of G1 GC



如果你想要理解G1收集器，首先你要忘記你所理解的新生代和老年代。正如你在上圖所看到的，每個對象被分配到不同的網格中，隨後執行垃圾回收。當一個區域填滿之後，對象被轉移到另一個區域，並再執行一次垃圾回收。在這種垃圾回收算法中，不再有從新生代移動到老年代的三部曲。這個類型的垃圾收集算法是為了替代CMS 收集器而被創建的，因為CMS 收集器在長時間持續運行時會產生很多問題。

G1最大的好處是他的性能，他比我們在上面討論過的任何一種GC都要快。但是在JDK 6中，他還只是一個早期試用版本。在JDK7之後才由官方正式發佈。就我個人看來，NHN在將JDK 7正式投入商用之前需要很長的一段測試期（至少一年）。因此你可能需要再等一段時間。並且，我也聽過幾次使用了JDK 6中的G1而導致Java虛擬機宕機的事件。請耐心的等待它更穩定吧。

建議

通過對垃圾收集器的介紹和梳理，在管理垃圾回收方面提出了五個建議，降低收集器開銷，進一步提升項目性能。

保持GC低開銷最實用的建議是什麼？

早有消息聲稱Java 9即將發佈，但如今卻一再推遲，其中比較值得關注的是G1（「Garbage-First」）垃圾收集器將成為HotSpot JVM的默認收集器。從串行收集器到CMS收集器，在整個生命週期中JVM已歷經多代GC的實現和更新，而接下來，G1收集器將譜寫新的篇章。

隨著垃圾收集器的持續發展，每一代都會進行改善和提高。在串行收集器之後的並行收集器利用多核機器強大的計算能力，實現了垃圾收集多線程。而之後的CMS（Concurrent Mark-Sweep）收集器，將收集分為多個階段執行，允許在應用線程運行同時進行大量的收集，大大降低了「stop-the-world」全局停頓的出現頻率。而現在，G1在JVM上加入了大量堆和可預測的均勻停頓，有效地提升了性能。

儘管GC不斷在完善，其致命弱點還是一樣：多餘的和不可預知的對象分配。但本文中提出了一些高效的長期實用的建議，不管你選擇哪種垃圾收集器，都可以幫助你降低GC開銷。

建議1：預測收集能力

所有的Java標準集合和大多數自定義的擴展實現（如Trove 和谷歌的Guava），都會使用底層數組（無論基於原始或基於對象）。數據的長度一旦分配後，數組就不可變了，所以在許多情況下，為集合增加項目可能會導致老的底層數組被刪除，然後需要重新分配一個更大的數組來替代。

大多數的集合實現都嘗試在集合沒有被設置為預期大小時，還能對重分配過程進行優化，並降低其開銷。但是，最好的結果還是在構造集合時就設置成預期大小。

讓我們看一下下面這個簡單的例子：

```
public static List reverse(List<?extends T> list)
{

   List result = new ArrayList();

   for (int i = list.size() - 1; i >= 0; i--) {
       result.add(list.get(i));
    }
   return result;
}
```

以上方法分配了一個新的數組，再將另一個列表的項目填充其中，但只能按倒序填充。

但是，難就難在如何優化增加項目到新列表這一步驟。每次添加後，該列表還需確保其底層數組有足夠的空槽能裝下新項目。如果能裝下，它就會直接在下一個空槽中存儲新項目；但如果空間不夠，它就會重新分配一個底層數組，將舊數組的內容複製到新數組中，然後再添加新項目。這一過程會導致分配的多個數組都會佔據內存，直到GC最後來回收。

所以，我們可以在構建時告知數組需容納多少個項目，重構後的代碼如下：

```
public static List reverse(List<?extends T> list)
{

   List result = new ArrayList(list.size());

   for (int i = list.size() - 1; i >= 0; i--) {
       result.add(list.get(i));
    }
   return result;
}
```

這樣一來，可以保證ArrayList構造函數在最初配置時就能容納下list.size()個項目，這意味著它不需要再在迭代中重新分配內存。

Guava的集合類則更加先進，允許我們用一個確切數量或估計值來初始化集合。

List result =Lists.newArrayListWithCapacity(list.size());

List result =Lists.newArrayListWithExpectedSize(list.size());

第一行代碼是我們知道有多少項目需要存儲的情況，第二行會分配一些多餘填充以適應預估誤差。

建議2：直接用處理流

當處理數據流時，如從文件中讀取數據或從網上下載數據，例如，我們通常可以從數據流中有所發現：

byte[] fileData = readFileToByteArray(newFile(「myfile.txt」));

由此產生的字節數組可以被解析為XML文檔、JSON對象或協議緩衝消息，來命名一些常用選項。

當處理大型或未知大小的文件時，這個想法則不適用了，因為當JVM無法分配文件大小的緩衝區時，則會出現OutOfMemoryErrors錯誤。

但是，即使數據大小看似能管理，當涉及到垃圾回收時，上述模式仍會造成大量開銷，因為它在堆上分配了相當大的blob來容納文件數據。

更好的處理方式是使用合適的InputStream（本例中是FileInputStream），並直接將其送到分析器，而不是提前將整個文件讀到字節數組中。所有主要庫會將API直接暴露給解析流，例如：

```
1  FileInputStream fis = newFileInputStream(fileName);
2  MyProtoBufMessage msg =
   MyProtoBufMessage.parseFrom(fis);
```

建議3：使用不可變對象

不變性有諸多優勢，但有一個優勢卻極少被重視，那就是不變性對垃圾回收的影響。

不可變對象是指對象一旦創建後，其字段（本例中指非原始字段）將無法被修改。例如：

```
1   public class ObjectPair {
2
3       private final Object first;
4       private final Object second;
5
6
7       public ObjectPair(Object first, Object second)
8   {
9           this.first = first;
10          this.second = second;
11       }
12
13       publicObject getFirst() {
14          return first;
15       }
16
17      public Object getSecond() {
18          return second;
19       }

    }
```

實例化上面類的結果為不可變對象——所有的字段一旦標記後則不能再被修改。

不變性意味著在構造容器完成之前，由不可變容器引用的所有對象都已經創建。在GC看來：容器會和其最新的新生代保持一致。這意味著當對新生代（young generations）執行垃圾回收週期時，GC可以跳過老年代（older generations）中的不可變對象，因為它知道不可變對象不能引用新生代的任何內容。

越少對象掃瞄意味著需掃瞄的內存頁越少，而越少的內存頁掃瞄意味著GC週期越短，同時也預示著更短的GC停頓和更好的整體吞吐量。

建議4：慎用字符串連接

字符串可能是任何基於JVM的應用中最普遍的非原始數據結構。但是，其隱含重量和使用便利性使得它們成為應用內存變大的罪魁禍首。

很明顯，問題不在於被內聯和拘留的文字字符串，而在於字符串在運行時被分配和構建。接下來看看構建動態字符串的簡單示例：

```
public static String toString(T[] array) {

   String result = "[";

   for (int i = 0; i < array.length; i++) {
       result += (array[i] == array ? "this" :
array[i]);
       if (i < array.length - 1) {
           result += ", ";
       }
    }

   result += "]";

   return result;
}
```

獲取數組並返回它的字符串表示是一個很不錯的方法，但這也正是對象分配的問題所在。

要看到其背後所有的語法糖並不容易，但真正的幕後場景應該是這樣：

```
public static String toString(T[] array) {

   String result = "[";

   for (int i = 0; i < array.length; i++) {

       StringBuilder sb1 = new StringBuilder(result);
       sb1.append(array[i] == array ? "this" :
array[i]);
       result = sb1.toString();

       if (i < array.length - 1) {
           StringBuilder sb2 = new
StringBuilder(result);
           sb2.append(", ");
           result = sb2.toString();
       }
    }

   StringBuilder sb3 = new StringBuilder(result);
   sb3.append("]");
   result = sb3.toString();

   return result;
}
```

字符串是不可變的，所以在其連接時並沒有被修改，而是依次分配新的字符串。此外，編譯器利用標準StringBuilder類來執行的這些鏈接。這就導致了雙重麻煩，在每次循環迭代時，我們得到（1）隱式分配臨時字符串，（2）隱式分配臨時的StringBuilder對象來幫助我們構建最終結果。

避免上述問題的最佳方法是明確使用StringBuilder並直接附加給它，而不是使用略幼稚的串聯運算符（「+」）。所以應該是這樣：

```
public static String toString(T[] array) {

    StringBuilder sb = new StringBuilder("[");

    for (int i = 0; i < array.length; i++) {
        sb.append(array[i] == array ? "this" :
array[i]);
        if (i < array.length - 1) {
            sb.append(", ");
        }
     }

    sb.append("]");
    return sb.toString();
}
```

此時，在方法開始時我們只分配了StringBuilder。從這一點來看，所有的字符串和列表項都會被添加到唯一的StringBuilder中，最終只調用一次toString方法轉換成字符串，然後返回結果。

建議5：使用專門的原始集合

Java的標準庫非常方便且通用，支持使用集合綁定半靜態類型。例如，如果要用一組字符串（Set<String>），或一對字符串映射到字符串列表（Map<Pair, List<String>>），直接利用標準庫會非常方便。

事實上，問題之所以出現是因為我們想把double類型的值放在 int 類型的list集合或map映射中。由於泛型不能調用原始集合，則可以用包裝類型代替，所以放棄List<int>而使用List<Integer>更好。

但其實這非常浪費，Integer本身就是一個完備對象，由12字節的對象頭和內部4字節的整數字段組合而成，加起來每個Integer對象佔16個字節，這是同樣大小的基類int類型長度的4倍！然而，更大的問題是所有這些Integer實際上都是垃圾回收過程中的對象實例。

為瞭解決這個問題，我們在Takipi 中使用優秀Trove 集合庫。Trove放棄了一些（但不是全部）支持專業高效內存的原始集合的泛型。例如，不用浪費的Map

```
TIntDoubleMap map =
newTIntDoubleHashMap();
map.put(5, 7.0);
map.put(-1, 9.999);
...
```

Trove底層實現了原始數組的使用，所以在操作集合時沒有裝箱（int -> Integer）或拆箱（Integer -> int）發生，因此也不會將對象存儲在基類中。

業務思想

隨著垃圾收集器不斷進步，以及實時優化和JIT編譯器變得更加智能，作為開發者的我們，可以越來越少地操心代碼的GC友好性。儘管如此，無論G1有多先進，在提高JVM方面，我們還有許多問題需要不斷探索和實踐，百尺竿頭仍需更進一步。

參考資料：

《深入理解Java虛擬機》第2版·周志明

伯樂在線·網站學習整理

OneAPM官方技術學習整理