Java 垃圾收集機制

對象引用

Java 中的垃圾回收一般是在 Java 堆中進行，因為堆中幾乎存放了 Java 中所有的對象實例。談到 Java 堆中的垃圾回收，自然要談到引用。在 JDK1.2 之前，Java 中的引用定義很很純粹：如果 reference 類型的數據中存儲的數值代表的是另外一塊內存的起始地址，就稱這塊內存代表著一個引用。但在 JDK1.2 之後，Java 對引用的概念進行了擴充，將其分為強引用（Strong Reference）、軟引用（Soft Reference）、弱引用（Weak Reference）、虛引用（Phantom Reference）四種，引用強度依次減弱。

- 強引用：如「Object obj = new Object（）」，這類引用是 Java 程序中最普遍的。只要強引用還存在，垃圾收集器就永遠不會回收掉被引用的對象。
- 軟引用：它用來描述一些可能還有用，但並非必須的對象。在系統內存不夠用時，這類引用關聯的對象將被垃圾收集器回收。JDK1.2 之後提供了 SoftReference 類來實現軟引用。
- 弱引用：它也是用來描述非需對象的，但它的強度比軟引用更弱些，被弱引用關聯的對象只能生存島下一次垃圾收集發生之前。當垃圾收集器工作時，無論當前內存是否足夠，都會回收掉只被弱引用關聯的對象。在 JDK1.2 之後，提供了 WeakReference 類來實現弱引用。
- 虛引用：最弱的一種引用關係，完全不會對其生存時間構成影響，也無法通過虛引用來取得一個對象實例。為一個對象設置虛引用關聯的唯一目的是希望能在這個對象被收集器回收時收到一個系統通知。JDK1.2 之後提供了 PhantomReference 類來實現虛引用。

垃圾對象的判定

Java 堆中存放著幾乎所有的對象實例，垃圾收集器對堆中的對象進行回收前，要先確定這些對象是否還有用，判定對象是否為垃圾對象有如下算法：

引用計數算法

給對象添加一個引用計數器，每當有一個地方引用它時，計數器值就加 1，當引用失效時，計數器值就減1，任何時刻計數器都為 0 的對象就是不可能再被使用的。

引用計數算法的實現簡單，判定效率也很高，在大部分情況下它都是一個不錯的選擇，當 Java 語言並沒有選擇這種算法來進行垃圾回收，主要原因是它很難解決對象之間的相互循環引用問題。

根搜索算法

Java 和 C# 中都是採用根搜索算法來判定對象是否存活的。這種算法的基本思路是通過一系列名為「GC Roots」的對象作為起始點，從這些節點開始向下搜索，搜索所走過的路徑稱為引用鏈，當一個對象到 GC Roots 沒有任何引用鏈相連時，就證明此對象是不可用的。在 Java 語言裡，可作為 GC Roots 的兌現包括下面幾種：

- 虛擬機棧（棧幀中的本地變量表）中引用的對象。
- 方法區中的類靜態屬性引用的對象。
- 方法區中的常量引用的對象。
- 本地方法棧中 JNI（Native 方法）的引用對象。

實際上，在根搜索算法中，要真正宣告一個對象死亡，至少要經歷兩次標記過程：如果對象在進行根搜索後發現沒有與 GC Roots 相連接的引用鏈，那它會被第一次標記並且進行一次篩選，篩選的條件是此對象是否有必要執行

finalize()方法。當對象沒有覆蓋 finalize()方法，或 finalize()方法已經被虛擬機調用過，虛擬機將這兩種情況都視為沒有必要執行。如果該對象被判定為有必要執行 finalize()方法，那麼這個對象將會被放置在一個名為 F-Queue 隊列中，並在稍後由一條由虛擬機自動建立的、低優先級的 Finalizer 線程去執行 finalize()方法。finalize()方法是對象逃脫死亡命運的最後一次機會（因為一個對象的 finalize()方法最多只會被系統自動調用一次），稍後 GC 將對 F-Queue 中的對象進行第二次小規模的標記，如果要在 finalize()方法中成功拯救自己，只要在 finalize()方法中讓該對象重引用鏈上的任何一個對象建立關聯即可。而如果對象這時還沒有關聯到任何鏈上的引用，那它就會被回收掉。

垃圾收集算法

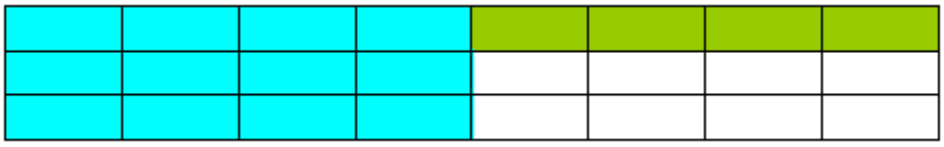
判定除了垃圾對象之後，便可以進行垃圾回收了。下面介紹一些垃圾收集算法，由於垃圾收集算法的實現涉及大量的程序細節，因此這裡主要是闡明各算法的實現思想，而不去細論算法的具體實現。

標記—清除算法

標記—清除算法是最基礎的收集算法，它分為「標記」和「清除」兩個階段：首先標記出所需回收的對象，在標記完成後統一回收掉所有被標記的對象，它的標記過程其實就是前面的根搜索算法中判定垃圾對象的標記過程。標記—清除算法的執行情況如下圖所示：

回收前狀態：

回收後狀態：



標記—整理算法

複製算法比較適合於新生代，在老年代中，對象存活率比較高，如果執行較多的複製操作，效率將會變低，所以老年代一般會選用其他算法，如標記—整理算法。該算法標記的過程與標記—清除算法中的標記過程一樣，但對標記後出的垃圾對象的處理情況有所不同，它不是直接對可回收對象進行清理，而是讓所有的對象都向一端移動，然後直接清理掉端邊界以外的內存。標記—整理算法的回收情況如下所示：

回收前狀態：

回收後狀態：

分代收集

當前商業虛擬機的垃圾收集 都採用分代收集，它根據對象的存活週期的不同將內存劃分為幾塊，一般是把 Java 堆分為新生代和老年代。在新生代中，每次垃圾收集時都會發現有大量對象死去，只有少量存活，因此可選用複製算法來完成收集，而老年代中因為對象存活率高、沒有額外空間對它進行分配擔保，就必須使用標記—清除算法或標記—整理算法來進行回收。

垃圾收集器

垃圾收集器是內存回收算法的具體實現，Java 虛擬機規範中對垃圾收集器應該如何實現並沒有任何規定，因此不同廠商、不同版本的虛擬機所提供的垃圾收集器都可能會有很大的差別。Sun HotSpot 虛擬機 1.6 版包含了如下收集器：Serial、ParNew、Parallel Scavenge、CMS、Serial Old、Parallel Old。這些收集器以不同的組合形式配合工作來完成不同分代區的垃圾收集工作。

垃圾回收分析

在用代碼分析之前，我們對內存的分配策略明確以下三點：

- 對象優先在 Eden 分配。
- 大對象直接進入老年代。
- 長期存活的對象將進入老年代。

對垃圾回收策略說明以下兩點：

- 新生代 GC（Minor GC）：發生在新生代的垃圾收集動作，因為 Java 對象大多都具有朝生夕滅的特性，因此 Minor GC 非常頻繁，一般回收速度也比較快。
- 老年代 GC（Major GC/Full GC）：發生在老年代的 GC，出現了 Major GC，經常會伴隨至少一次 Minor GC。由於老年代中的對象生命週期比較長，因此 Major GC 並不頻繁，一般都是等待老年代滿了後才進行 Full GC，而且其速度一般會比 Minor GC 慢 10 倍以上。另外，如果分配了 Direct Memory，在老年代中進行 Full GC時，會順便清理掉 Direct Memory 中的廢棄對象。

下面我們來看如下代碼:

```
public class SlotGc{
    public static void main(String[] args){
        byte[] holder = new byte[32*1024*1024];
        System.gc();
    }
}
```

代碼很簡單，就是向內存中填充了 32MB 的數據，然後通過虛擬機進行垃圾收集。在 javac 編譯後，我們執行如下指令：java -verbose:gc SlotGc 來查看垃圾收集的結果，得到如下輸出信息：

```
[GC 208K->134K(5056K), 0.0017306 secs]

[Full GC 134K->134K(5056K), 0.0121194 secs]

[Full GC 32902K->32902K(37828K), 0.0094149 sec
```

注意第三行，「->」之前的數據表示垃圾回收前堆中存活對象所佔用的內存大小，「->」之後的數據表示垃圾回收堆中存活對象所佔用的內存大小，括號中的數據表示堆內存的總容量，0.0094149 sec 表示垃圾回收所用的時間。

從結果中可以看出，System.gc()運行後並沒有回收掉這 32MB 的內存，這應該是意料之中的結果，因為變量holder還處在作用域內，虛擬機自然不會回收掉 holder 引用的對象所佔用的內存。

我們把代碼修改如下：

```
public class SlotGc{
    public static void main(String[] args){
        {
        byte[] holder = new byte[32*1024*1024];
        }
        System.gc();
    }
}
```

加入花括號後，holder 的作用域被限制在了花括號之內，因此，在執行System.gc()時，holder 引用已經不能再被訪問，邏輯上來講，這次應該會回收掉 holder 引用的對象所佔的內存。但查看垃圾回收情況時，輸出信息如下：

```
[GC 208K->134K(5056K), 0.0017100 secs]

[Full GC 134K->134K(5056K), 0.0125887 secs]

[Full GC 32902K->32902K(37828K), 0.0089226 secs]
```

很明顯，這 32MB 的數據並沒有被回收。下面我們再做如下修改：

```
public class SlotGc{
    public static void main(String[] args){
        {
        byte[] holder = new byte[32*1024*1024];
        holder = null;
        }
        System.gc();
    }
}
```

這次得到的垃圾回收信息如下：

```
[GC 208K->134K(5056K), 0.0017194 secs]

[Full GC 134K->134K(5056K), 0.0124656 secs]

[Full GC 32902K->134K(37828K), 0.0091637 secs]
```

說明這次 holder 引用的對象所佔的內存被回收了。我們慢慢來分析。

首先明確一點：holder 能否被回收的根本原因是局部變量表中的 Slot 是否還存有關於 holder 數組對象的引用。

在第一次修改中，雖然在 holder 作用域之外進行回收，但是在此之後，沒有對局部變量表的讀寫操作，holder 所佔用的 Slot 還沒有被其他變量所復用（回憶 Java 內存區域與內存溢出一文中關於 Slot 的講解），所以作為 GC Roots 一部分的局部變量表仍保持者對它的關聯。這種關聯沒有被及時打斷，因此 GC 收集器不會將 holder 引用的對象內存回收掉。 在第二次修改中，在 GC 收集器工作前，手動將 holder 設置為 null 值，就把 holder 所佔用的局部變量表中的 Slot 清空了，因此，這次 GC 收集器工作時將 holder 之前引用的對象內存回收掉了。

當然，我們也可以用其他方法來將 holder 引用的對象內存回收掉，只要復用 holder 所佔用的 slot 即可，比如在 holder 作用域之外執行一次讀寫操作。

為對象賦 null 值並不是控制變量回收的最好方法，以恰當的變量作用域來控制變量回收時間才是最優雅的解決辦法。另外，賦 null 值的操作在經過虛擬機 JIT 編譯器優化後會被消除掉，經過 JIT 編譯後，System.gc()執行時就可以正確地回收掉內存，而無需賦 null 值。

性能調優

Java 虛擬機的內存管理與垃圾收集是虛擬機結構體系中最重要的組成部分，對程序（尤其服務器端）的性能和穩定性有著非常重要的影響。性能調優需要具體情況具體分析，而且實際分析時可能需要考慮的方面很多，這裡僅就一些簡單常用的情況作簡要介紹。

- 我們可以通過給 Java 虛擬機分配超大堆（前提是物理機的內存足夠大）來提升服務器的響應速度，但分配超大堆的前提是有把握把應用程序的 Full GC 頻率控制得足夠低，因為一次 Full GC 的時間造成比較長時間的停頓。控制 Full GC 頻率的關鍵是保證應用中絕大多數對象的生存週期不應太長，尤其不能產生批量的、生命週期長的大對象，這樣才能保證老年代的穩定。
- Direct Memory 在堆內存外分配，而且二者均受限於物理機內存，且成負相關關係，因此分配超大堆時，如果用到了 NIO 機制分配使用了很多的 Direct Memory，則有可能導致 Direct Memory 的 OutOfMemoryError 異常，這時可以通過 -XX:MaxDirectMemorySize 參數調整 Direct Memory 的大小。
- 除了 Java 堆和永久代以及直接內存外，還要注意下面這些區域也會佔用較多的內存，這些內存的總和會受到操作系統進程最大內存的限制：

1、線程堆棧：可通過 -Xss 調整大小，內存不足時拋出 StackOverflowError（縱向無法分配，即無法分配新的棧幀）或 OutOfMemoryError（橫向無法分配，即無法建立新的線程）。

2、Socket 緩衝區：每個 Socket 連接都有 Receive 和 Send 兩個緩衝區，分別佔用大約 37KB 和 25KB 的內存。如果無法分配，可能會拋出 IOException：Too many open files 異常。關於 Socket 緩衝區的詳細介紹參見我的 Java 網絡編程系列中深入剖析 Socket 的幾篇文章。

3、JNI 代碼：如果代碼中使用了JNI調用本地庫，那本地庫使用的內存也不在堆中。

4、虛擬機和 GC：虛擬機和 GC 的代碼執行也要消耗一定的內存。