Java GC系列（2）：Java垃圾回收是如何工作的？

本文由 ImportNew - 伍翀 翻譯自 javapapers。歡迎加入翻譯小組。轉載請見文末要求。

目錄

1. 垃圾回收介紹
2. 垃圾回收是如何工作的？
3. 垃圾回收的類別
4. 垃圾回收監視和分析

本教程是為了理解基本的Java垃圾回收以及它是如何工作的。這是垃圾回收教程系列的第二部分。希望你已經讀過了第一部分：《Java 垃圾回收介紹》。

Java 垃圾回收是一項自動化的過程，用來管理程序所使用的運行時內存。通過這一自動化過程，JVM 解除了程序員在程序中分配和釋放內存資源的開銷。

啟動Java垃圾回收

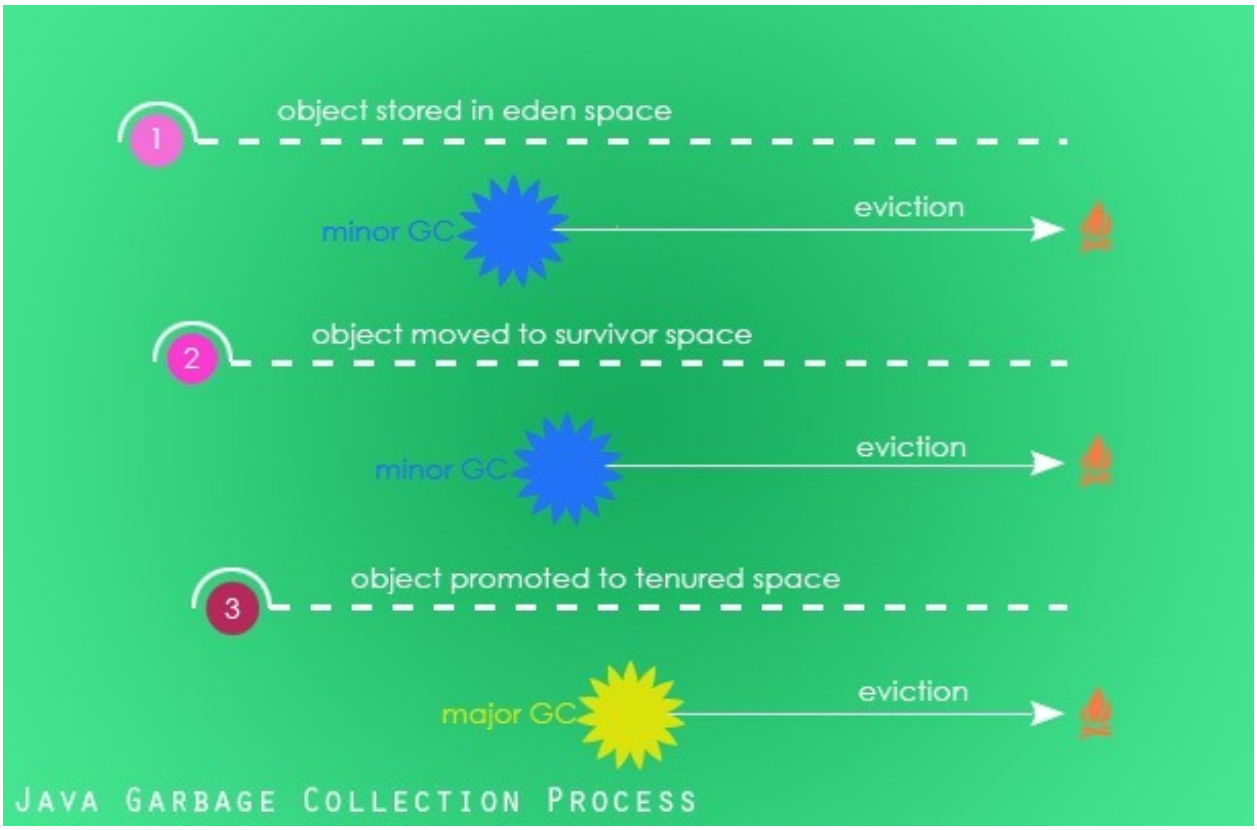
作為一個自動的過程，程序員不需要在代碼中顯示地啟動垃圾回收過程。`System.gc()`和`Runtime.gc()`用來請求JVM啟動垃圾回收。

雖然這個請求機制提供給程序員一個啟動 GC 過程的機會，但是啟動由 JVM負責。JVM可以拒絕這個請求，所以並不保證這些調用都將執行垃圾回收。啟動時機的選擇由JVM決定，並且取決於堆內存中Eden區是否可用。JVM將這個選擇留給了Java規範的實現，不同實現具體使用的算法不盡相同。

毋庸置疑，我們知道垃圾回收過程是不能被強制執行的。我剛剛發現了一個調用`System.gc()`有意義的場景。通過這篇文章瞭解一下適合調用System.gc() 這種極端情況。

Java垃圾回收過程

垃圾回收是一種回收無用內存空間並使其對未來實例可用的過程。



Eden 區：當一個實例被創建了，首先會被存儲在堆內存年輕代的 Eden 區中。

注意：如果你不能理解這些詞彙，我建議你閱讀這篇 垃圾回收介紹 ，這篇教程詳細地介紹了內存模型、JVM 架構以及這些術語。

Survivor 區（S0 和 S1）：作為年輕代 GC（Minor GC）週期的一部分，存活的對象（仍然被引用的）從 Eden 區被移動到 Survivor 區的 S0 中。類似的，垃圾回收器會掃瞄 S0 然後將存活的實例移動到 S1 中。

（譯註：此處不應該是Eden和S0中存活的都移到S1麼，為什麼會先移到S0再從S0移到S1？）

死亡的實例（不再被引用）被標記為垃圾回收。根據垃圾回收器（有四種常用的垃圾回收器，將在下一教程中介紹它們）選擇的不同，要麼被標記的實例都會不停地從內存中移除，要麼回收過程會在一個單獨的進程中完成。

老年代： 老年代（Old or tenured generation）是堆內存中的第二塊邏輯區。當垃圾回收器執行 Minor GC 週期時，在 S1 Survivor 區中的存活實例將會被晉陞到老年代，而未被引用的對象被標記為回收。

老年代 GC（Major GC）：相對於 Java 垃圾回收過程，老年代是實例生命週期的最後階段。Major GC 掃瞄老年代的垃圾回收過程。如果實例不再被引用，那麼它們會被標記為回收，否則它們會繼續留在老年代中。

內存碎片：一旦實例從堆內存中被刪除，其位置就會變空並且可用於未來實例的分配。這些空出的空間將會使整個內存區域碎片化。為了實例的快速分配，需要進行碎片整理。基於垃圾回收器的不同選擇，回收的內存區域要麼被不停地被整理，要麼在一個單獨的GC進程中完成。

垃圾回收中實例的終結

在釋放一個實例和回收內存空間之前，Java 垃圾回收器會調用實例各自的 `finalize()` 方法，從而該實例有機會釋放所持有的資源。雖然可以保證 `finalize()` 會在回收內存空間之前被調用，但是沒有指定的順序和時間。多個實例間的順序是無法被預知，甚至可能會並行發生。程序不應該預先調整實例之間的順序並使用 `finalize()` 方法回收資源。

- 任何在 finalize過程中未被捕獲的異常會自動被忽略，然後該實例的 finalize 過程被取消。
- JVM 規範中並沒有討論關於弱引用的垃圾回收機制，也沒有很明確的要求。具體的實現都由實現方決定。
- 垃圾回收是由一個守護線程完成的。

對象什麼時候符合垃圾回收的條件？

- 所有實例都沒有活動線程訪問。
- 沒有被其他任何實例訪問的循環引用實例。

Java 中有不同的引用類型。判斷實例是否符合垃圾收集的條件都依賴於它的引用類型。

引用類型	垃圾收集
強引用（Strong Reference）	不符合垃圾收集
軟引用（Soft Reference）	垃圾收集可能會執行，但會作為最後的選擇
弱引用（Weak Reference）	符合垃圾收集
虛引用（Phantom Reference）	符合垃圾收集

在編譯過程中作為一種優化技術，Java 編譯器能選擇給實例賦 `null` 值，從而標記實例為可回收。

```
class Animal {
    public static void main(String[] args) {
        Animal lion = new Animal();
        System.out.println("Main is
completed.");
    }

    protected void finalize() {
        System.out.println("Rest in Peace!");
    }
}
```

在上面的類中，`lion` 對象在實例化行後從未被使用過。因此 Java 編譯器作為一種優化措施可以直接在實例化行後賦值`lion = null`。因此，即使在 SOP 輸出之前， finalize 函數也能夠打印出 `'Rest in Peace!'`。我們不能證明這確定會發生，因為它依賴JVM的實現方式和運行時使用的內存。然而，我們還能學習到一點：如果編譯器看到該實例在未來再也不會被引用，能夠選擇並提早釋放實例空間。

- 關於對象什麼時候符合垃圾回收有一個更好的例子。實例的所有屬性能被存儲在寄存器中，隨後寄存器將被訪問並讀取內容。無一例外，這些值將被寫回到實例中。雖然這些值在將來能被使用，這個實例仍然能被標記為符合垃圾回收。這是一個很經典的例子，不是嗎？
- 當被賦值為null時，這是很簡單的一個符合垃圾回收的示例。當然，複雜的情況可以像上面的幾點。這是由 JVM 實現者所做的選擇。目的是留下儘可能小的內存佔用，加快響應速度，提高吞吐量。為了實現這一目標， JVM 的實現者可以選擇一個更好的方案或算法在垃圾回收過程中回收內存空間。
- 當 `finalize()` 方法被調用時，JVM 會釋放該線程上的所有同步鎖。

GC Scope 示例程序

```
Class GCScope {
    GCScope t;
    static int i = 1;

    public static void main(String args[]) {
        GCScope t1 = new GCScope();
        GCScope t2 = new GCScope();
        GCScope t3 = new GCScope();



        t1.t = t2;
        t2.t = t3;
        t3.t = t1;

        t1 = null;


        t2 = null;


        t3 = null;







    }

    protected void finalize() {
        System.out.println("Garbage collected from object" +
i);
        i++;
    }

class GCScope {
    GCScope t;
    static int i = 1;

    public static void main(String args[]) {
        GCScope t1 = new GCScope();
        GCScope t2 = new GCScope();
        GCScope t3 = new GCScope();



        t1.t = t2;
        t2.t = t3;
        t3.t = t1;

        t1 = null;


        t2 = null;


        t3 = null;

```

```
60
61
62
63          }

            protected void finalize() {
                System.out.println("Garbage collected from object" +
        i);
                i++;
            }
```

GC OutOfMemoryError 的示例程序

GC並不保證內存溢出問題的安全性，粗心寫下的代碼會導致 `OutOfMemoryError`。

```
1   import java.util.LinkedList;
2   import java.util.List;
3
4   public class GC {
5       public static void main(String[] main) {
6           List l = new LinkedList();
7
8
9
10          do {
11              l.add(new String("Hello,
12  World"));
13          } while (true);
14      }
15  }
```

輸出：

```
1   Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
2       at java.util.LinkedList.linkLast(LinkedList.java:142)
3       at java.util.LinkedList.add(LinkedList.java:338)
4       at com.javapapers.java.GCScope.main(GCScope.java:12)
```

接下來是垃圾收集系列教程的第三部分，我們將會看到常用的 不同 的Java垃圾收集器。

原文鏈接： javapapers 翻譯： ImportNew.com - 伍狨
譯文鏈接： http://www.importnew.com/13493.html
[轉載請保留原文出處、譯者和譯文鏈接。]

關於作者： 伍狨

（新浪微博：**@黑桃夾克** ，博客：**http://wuchong.me**）

查看伍狨的更多文章 >>