JAVA垃圾收集器算法分析及垃圾收集器的运行透视

Algorithm Analysis of Java Garbage Collector and Its Running Perspective

摘　要：本文分析了JVM垃圾收集器所使用的多种收集算法，讨论如何通过命令行参数和终止化方法，透视JAVA自动垃圾收集器的运行，以理解JVM垃圾自动回收机制。程序开发人员应当使用finalize方法明确释放对象资源，并通过GC请求JVM执行垃圾收集，使得JVM尽可能多地回收内存供程序使用。

关键字：JAVA虚拟机、垃圾收集器、运行透视

浦云明　（厦门集美大学信息工程学院　361021）

1 引言

垃圾是原先分配给对象的堆内存，由于程序运行中对象的丢弃或失效，这些堆内存就不再有用，垃圾收集是指堆内存的回收，即重新使用。随着JAVA垃圾的堆积，将减少总的可用内存，如果JAVA虚拟机（JVM）不及时回收那些垃圾，JVM会耗尽内存，导致JAVA程序运行的灾难，为防止这种情况的发生，JVM自动执行垃圾收集GC（garbage collection）处理。一般来说，JAVA开发人员不需重视JVM中内存的分配和垃圾收集，但是，充分理解JAVA的这一内部机制能更有效使用资源。

2 JAVA垃圾收集器

在面向对象程序设计中，我们使用关键字new来创建并实例化对象，这需要JVM为对象分配内存，只要这个对象存在，JVM就不会把该内存块分给其他的对象。请看下面的源程序代码段Manager.java。

```
class Employee{
    String name,id;
}
public class Manager extends Employee{
    String dept;
    Employee[] subordinates = {new Employee(),new Employee()};//实例化2个数组元素，用new分配内存
    Public static void main(String args[])
    {
    Manager m =  new Manager(); //为对象m分配存储空间，并实例化为默认值
    m.dept = "operating";
    m.subordinates[0].name = "zhaohong"; //设定m的雇员的名字，即Employee数组对象的数据,
    m.subordinates[0].id = "1001";
    ...
    for(int i =0;i<m.subordinates.length;i++)
        System.out.println(m.dept + m.subordinates[i].name + m.subordinates[i].id);
    }
    }
```

程序中，类Manager继承类Employee，用new关键字创建类Manager的对象m并分配存储空间（使用缺省构造器），创建类型为Employee的数组对象subordinates并分配存储空间（共2个数组元素），我们用点运算符访问m的属性dept和m的所有雇员subordinates的属性name和id，最后打印输出对象m所在的部门名称和他的雇员姓名和id，在这之后对象m和对象subordinates在程序中不再使用，那么原来对象m和对象subordinates所分配的内存如何处理呢？在C++中，对象所占的内存在程序结束运行之前一直被占用，在明确释放前不能分配给其他对象。在JAVA中，当没有对象引用指向原先分配给某个对象的内存块，该块内存便成为垃圾，可以回收再利用。JVM的一个系统级线程会自动释放该内存块，这就是JAVA的自动垃圾回收机制。

垃圾收集器是JVM的一部分，GC的效率是JVM性能的一个重要指标。JVM的不同，垃圾收集的处理算法也会有所不同。当JAVA程序运行时，垃圾收集器作为后台线程运行，执行垃圾回收工作。由于这个线程在不确定的时间偶然运行，因此程序不知道垃圾收集何时执行。为了执行垃圾收集，垃圾收集线程一般执行以下两个任务。

（1）释放丢弃对象所占的内存以供再利用，防止某一瞬间出现内存耗尽的情况。

（2）碎片整理。由于创建对象和垃圾收集器释放丢弃对象所占的内存，内存会出现碎片。碎片是分配给对象的内存块之间的空闲内存洞（memory holes）。碎片整理将所占用的堆内存移至堆的一端，JVM将整理出的内

存分配给新的对象。

随着SUN公司JVM新版本的推出，垃圾收集器功能和性能也得到了较大改进。如JVM1.2.2版的 Exact虚拟机引入了精确垃圾收集功能，JVM1.3版的HotSpot虚拟机中引入了代式GC功能。代式GC把对象分成若干代，分别放到不同的堆空间里，这样就解决了每次GC前，为查找和移动无效对象必须扫描所有对象花费较长时间的问题，而JVM1.4版的HotSpot虚拟机实现连续的GC算法并同时使用多种GC算法。

3 JAVA 垃圾收集器的算法分析

JLS没有指定JVM使用何种垃圾收集算法，但任何一个JVM的垃圾收集器必须能检测到不再有用的对象，以实现垃圾回收。下面介绍常用的一些收集算法，每一种算法的性能各有所长。

多数垃圾收集器算法使用了根集（root set）这一概念，根集是正在执行的JAVA程序可以访问的引用变量的集合（包括局部变量、参数、类变量），程序可以使用引用变量访问对象的属性和调用对象的方法。垃圾收集器首先需要确定从根开始哪些对象可达（reachability）或不可达，从根集可达的对象都是活动（live）对象，它们不能作为垃圾被回收，这也包括从根集间接可达的对象。而从根集通过任意路径不可达的对象符合垃圾收集的条件，应予回收。

3.1 引用计数法

引用记数法是唯一没有使用根集的GC算法，该算法使用引用计数器来区分存活（live）对象和不再使用的对象。一般来说，堆中的每个对象对应一个引用计数器。当第一次创建一个对象并赋给一个变量时，引用计数器置为1。当对象被赋给任意变量时，引用计数器每次加1。当对象出了作用域后（该对象丢弃不再使用），引用计数器减1，一旦引用计数器为0，对象就满足了垃圾收集的条件。

基于引用计数器的垃圾收集器运行较快，不会长时间中断程序执行，适宜于必须实时运行的程序。但引用计数器增加了程序执行的开销，因为每次对象赋给新的变量，计数器加1，而每次现有对象出了作用域后，计数器减1。

3.2 tracing 算法

tracing算法是为解决引用计数法的问题而提出，它使用了根集概念。基于tracing的垃圾收集器从根集开始扫描，识别出哪些对象可达，哪些对象不可达，并用某种方式标记可达对象，例如对每个可达的对象设置一个或多个位。在扫描识别过程中，基于tracing的收集器回收那些没有标记的对象。基于tracing算法的垃圾收集器也称为标记和清除（mark-and-sweep）垃圾收集器。

3.3 Compacting 算法

为了解决堆碎片问题，基于tracing的GC吸收了Compacting算法的思想，在清除的过程中，算法将所有的对象移到堆的一端，堆的另一端就变成了一个相邻的空闲内存区，收集器会对它移动的所有对象的所有引用进行更新，使得这些引用在新的位置能识别原来的对象。在基于 Compacting算法的收集器的实现中，一般增加句柄和句柄表。

3.4 Copying 算法

Copying算法的提出是为了克服句柄的开销和解决堆碎片的垃圾收集。它开始时把堆分成一个对象面和多个空闲面，程序从对象面为对象分配空间，当对象满了，基于Copying算法的垃圾收集器GC从根集中扫描活动对象，并将每个活动对象复制到空闲面（使得活动对象所占的内存之间没有内存洞），这样空闲面变成了对象面，原来的对象面变成了空闲面，程序会在新的对象面中分配内存。

一种典型的基于Copying算法的垃圾收集器是stop-and-copy垃圾收集器，它将堆分成对象面和空闲区域面，在对象面与空闲区域面的切换过程中，暂停程序的执行。

3.5 Generation 算法

stop-and-copy垃圾收集器的一个缺陷是收集器必须复制所有的活动对象，这增加了程序的等待时间，这是Copying算法低效的原因。在程序设计中有这样的规律：多数对象存在的时间较短，少数对象被反复地复制并长时间存在。因此，Generation算法将堆分成两个或多个，每个子堆作为对象的一代（Generation）。由于多数对象存在的时间较短，随着程序丢弃不使用的对象，垃圾收集器将从最年轻的子堆中收集这些对象。在分代式的垃圾收集器运行后，上次运行存活下来的对象移到下一最高代的子堆中，由于老一代的子堆不会经常回收垃圾，因而节省了时间。分代式GC堆内存的组成见图1。

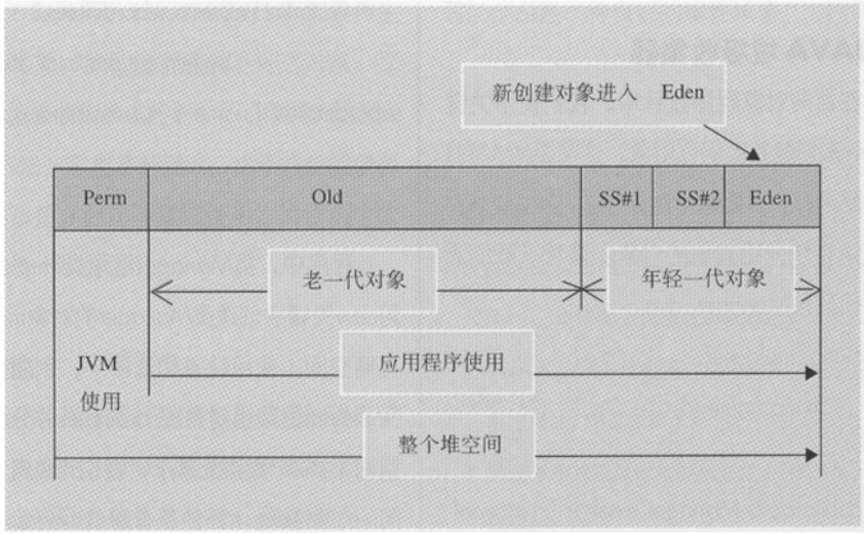

图 1 分代式 GC 堆空间的组成

图1中，年轻一代的对象空间包括Eden和两个存活空间SS#1、SS#2，新创建对象被分配到Eden，那些存活时间长的对象则从年轻一代转移到老一代对象空间OLD，永久代空间（perment generation）的内存段用来存放JVM的类和方法对象，其大小可以用命令行参数-xx:MaxPermSize=64m来调整。Eden空间小于堆内存总容量的1/2，同时老一代的容量必须大于年轻一代的容量。一般来说老新代比率为5：1，在java hotspot client VM中比率为9：1，而在java hotspot server VM中比率为2：1.

3.6 Adaptive 算法

在特定的情况下，一些垃圾收集算法会优于其他算法。基于Adaptive算法的垃圾收集器就是监控当前堆的使用情况，并选择适当算法的垃圾收集器。

4 JAVA 垃圾收集器运行透视

4.1 命令行参数透视垃圾收集器的运行

不论JVM使用何种垃圾收集算法，我们都可以使用System.gc()请求JAVA执行垃圾收集。一般来说，在执行一个较多消耗内存的计算程序，可以先进行垃圾收集清除不可达的对象，以产生更多的空闲可用内存，提高计算程序的执行速度。为了透视JVM垃圾收集的运行，查看内存的回收情况，我们可以使用命令行参数 - verbosegc通知JVM输出堆数据（JAVA使用的堆内存），格式如下：

```
>java -verbosegc javaclassfile
```

在Windows XP平台上，使用J2SDK1.4.1_02版本，输出如下信息：

```
[Full GC 268K->151K(1984K), 0.0110559 secs]
```

这是一个JVM的major收集，其中箭头前后的数据268K和151K分别表示在垃圾收集GC前后所有存活对象使用的内存容量，这也说明有117K（268-151）的对象容量被回收，括号内的数据是堆内存总容量，收集所需要的时间是0.0110559秒（这个时间每次执行会有所不同）。请看程序Demo1.java，它创建一个对象，由于没有进一步使用该对象，对象迅速被丢弃，使得该对象变为不可达，该对象符合垃圾收集的条件，命令System.gc()请求JVM执行垃圾收集。

```
class Demo1
{
    public static void main(String args[])
    {
        new Demo1();
            System.gc();
            System.runFinalization(); //在XP机器上，不必要
    }
}
```

程序编译后，执行命令行java -verbosegc Demo1后，运行结果为：

```
[Full GC 268K->151K(1984K), 0.0110439 secs]
```

4.2 finalize 方法透视垃圾收集器的运行

在JVM垃圾收集器收集一个对象前，一般要求程序调用适当的方法释放资源，但在你没有明确释放资源的情况下，JAVA提供了缺省机制来终止化（finalize）该对象以释放资源，finalize是JAVA中对象的方法之一，由于finalize方法只能在JAVA垃圾收集之前调用，当一个对象超过作用域，就不能调用finalize方法了。因此，良好的程序设计风格是自己设计对象释放程序，以明确释放对象资源，确保释放的可靠性。

finalize方法运行在垃圾收集前，在finalize方法中，对象释放所有获得的有限资源。finalize()方法的原型如下：

```
protected void finalize() throws Throwable
```

在finalize方法返回后，对象消失，垃圾收集开始执行。原型中的throws Throwable子句表示finalize方法可以抛出任意类型的例外。

```
class Demo2
{
    public static void main(String args[])
    {
        new Demo2();
    }
    public void finalize() throws Throwable
    {
            System.out.println("Demo2 finalize called");
            super.finalize(); //显式调用超类的finalize()方法
    }
}
```

这个应用程序中，有一打印“Demo2 finalize called”消息的finalize()方法，并调用super.finalize()方法，使得它的超类释放任何获取的资源，这是必须的，也是良好的编程风格，因为垃圾收集器不会自动调用任何超类的finalize()方法。在main()中，创建了一个Demo2对象，由于没有进一步使用该对象，对象迅速被丢弃，使得该对象变为不可达，该对象符合垃圾收集的条件，但是，finalize()方法没有运行，程序没有输出“Demo2 finalize called”，因为java垃圾收集器是一后台线程，我们不知它何时运行，在垃圾收集器运行前，我们已经退出程序，一个不可达的Demo1对象就留在了内存里，由于finalize方法是JAVA的一种缺省机制，因此JAVA不能保证每个对象的finalize()方法都得到调用。因此，为确保退出运行程序前调用finalize方法，我们必须使用System.gc()请求JAVA执行垃圾收集，请看改进后的程序Demo3.java

```
class Demo3
{
    public static void main(String args[])
    {
        new Demo3();
            System.gc();
            System.runFinalization(); //在
```

下转第 38 页 >>

XP机器上，不必要

```
        }
        public void finalize() throws Throwable
        {
                System.out.println("Demo3 finalize called");
                super.finalize();
        }
    }
```

在main( )中，我们使用System.gc()方法和System.runFinalization()方法，来明确释放资源，请求JVM执行垃圾收集、调用Finalize方法，在windows XP上，程序运行finalize()，输出了"Demo3 finalize called"和垃圾收集的运行信息。程序输出如下：

```
[Full GC 268K->151K(1984K), 0.0109503 secs]
Demo3 finalize called
```

由于在一个对象的finalize( )方法调用后，JAVA垃圾收集器不一定立刻回收该对象，因此，我们总是要求明确地释放对象资源，使用System.gc()请求JVM执行垃圾收集。另外，在finalize方法中，可以将正在丢弃的对象赋给一个对象引用，这能够防止对象被收集，这种行为称为再生（resurrection）。如果再生了一个对象，然后使它成为不可达对象，那么下次垃圾收集器运行时，不必调用对象的finalize方法，它也将被回收。由于再生会使源代码不清楚，因此不建议使用，关于再生本文不作进一步的说明。

5 结论

当JAVA程序运行后，不再使用的对象会累积在内存中，JVM的垃圾收集器每次运行时，都会自动回收这些内存。我们也可以使用JAVA的System类的gc()方法请求JVM运行垃圾收集器，我们可以使用命令行参数和终止化方法透视JVM垃圾收集的运行情况。终止化finalize()方法是垃圾收集器在收集对象之前必须调用的方法，它保证在GC前已释放对象资源。由于finalize方法是JAVA的缺省机制，因此为确保对象所占资源的明确释放，自己编写finalize是一种良好的编程风格。

参考文献

1 JAVA How to Program, 4th edition, Harvey M. Deitel，电子工业出版社，2002 年 6 月

2 JAVA 2 应用开发指南，飞思科技，电子工业出版社，2002 年 1 月

3 通过分布式垃圾收集提高性能，《程序员》，2002 年