# JVM——内存管理和垃圾回收

1. **何为GC**

Java与C语言相比的一个优势是，可以通过自己的JVM自动分配和回收内存空间。

垃圾回收机制是由垃圾收集器Garbage Collection来实现的，GC是后台一个低优先级的守护进程。在内存中低到一定限度时才会自动运行，因此垃圾回收的时间是不确定的。

为何要这样设计：因为GC也要消耗CPU等资源，如果GC执行过于频繁会对Java的程序的执行产生较大影响，因此实行不定期的GC。

与GC有关的是：JVM运行时数据区中的堆（对象实例会存储在这里）和gabagecollector方法。

垃圾回收GC只能回收通过new关键字申请的内存（在堆上），但是堆上的内存并不完全是通过new申请分配的。还有一些本地方法，这些内存如果不手动释放，就会导致内存泄漏，所以需要在finalize中用本地方法（nativemethod）如free操作等，再使用gc方法。



1. **何为垃圾**

Java中那些不可达的对象就会变成垃圾。对象之间的引用可以抽象成树形结构，通过树根（GC Roots）作为起点，从这些树根往下搜索，搜索走过的链称为引用链。

当一个对象到GC Roots没有任何引用链相连时，则证明这个对象为可回收的对象。

可以作为GC Roots的主要有以下几种：

1. 栈帧中的本地变量表所引用的对象。
2. 方法区中类静态属性和常量引用的对象。
3. 本地方法栈中JNI（Native方法）引用的对象。

//垃圾产生的情况举例：

//1.改变对象的引用，如置为null或者指向其他对象

Object obj1 = new Object();

Object obj2 = new Object();

obj1 = obj2; //obj1成为垃圾

obj1 = obj2 = null ; //obj2成为垃圾

//2.引用类型

//第2句在内存不足的情况下会将String对象判定为可回收对象，第3句无论什么情况下String对象都会被判定为可回收对象

String str = new String("hello");

SoftReference<String> sr = new SoftReference<String>(new String("java"));

WeakReference<String> wr = new WeakReference<String>(new String("world"));

//3.循环每执行完一次，生成的Object对象都会成为可回收的对象

for(int i=0;i<10;i++) {

Object obj = new Object();

System.out.println(obj.getClass());

}

//4.类嵌套

class A{

A a;

}

A x = new A();//分配了一个空间

x.a = new A();//又分配了一个空间

x = null;//产生两个垃圾

//5.线程中的垃圾

class A implements Runnable{

void run(){

//....

}

}

//main

A x = new A();

x.start();

x = null; //线程执行完成后x对象才被认定为垃圾

1. **四种引用类型**

**3.1强引用**

Object obj = new Object();

这里的obj引用便是一个强引用，强引用不会被GC回收。即使抛出OutOfMemoryError错误，使程序异常终止。

**3.2软引用**

如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。

**3.3弱引用**

弱引用与软引用的区别在于：垃圾回收器一旦发现了弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

**3.4虚引用**

虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器发现一个对象有虚引用时，首先执行所引用对象的finalize()方法，在回收内存之前，把这个虚引用对象加入到引用队列中，你可以通过判断引用队列中是否有该虚引用对象，来了解这个对象是否将要被垃圾回收。然后就可以利用虚引用机制完成对象回收前的一些工作。

1. **典型的垃圾回收算法**

在确定了哪些垃圾可以被回收后，垃圾收集器要做的事情就是开始进行垃圾回收，但是这里面涉及到一个问题是：如何高效地进行垃圾回收。

下面讨论几种常见的垃圾收集算法：

**4.1Mark-Sweep（标记-清除）算法**

标记-清除算法分为两个阶段：标记阶段和清除阶段。

标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。

标记-清除算法实现起来比较容易，但是有一个比较严重的问题就是容易产生内存碎片，碎片太多可能会导致后续过程中需要为大对象分配空间时无法找到足够的空间而提前触发GC。

**4.2Copying（算法）算法**

Copying算法将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把第一块内存上的空间一次清理掉，这样就不容易出现内存碎片的问题，并且运行高效。

但是该算法导致能够使用的内存缩减到原来的一半。而且，该算法的效率跟存活对象的数目多少有很大的关系，如果存活对象很多，那么Copying算法的效率将会大大降低。（这也是为什么后面提到的新生代采用Copying算法）

**4.3Mark-Compact（标记-整理）算法**

为了解决Copying算法的缺陷，充分利用内存空间，提出了Mark-Compact算法。

该算法标记阶段标记出所有需要被回收的对象，但是在完成标记之后不是直接清理可回收对象，而是将存活的对象都移向一端，然后清理掉端边界以外的所有内存（只留下存活对象）。

**4.4Generational Collection（分代收集）算法**

分代收集算法是目前大部分JVM的垃圾收集器采用的算法。

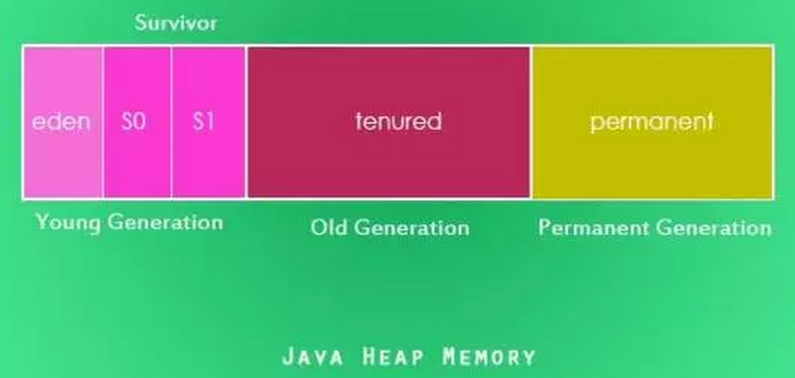
它的核心思想是将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），老年代的特点是每次垃圾收集时只有少量对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以在不同代的采取不同的最适合的收集算法。

目前大部分垃圾收集器对于新生代都采取Copying算法，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数较少，该算法效率在新生代也较高。但是实际中并不是按照1:1的比例来划分新生代的空间的，一般来说是将新生代划分为一块较大的Eden空间和两块较小的Survivor空间（比例8：1：1），每次使用Eden空间和其中的一块Survivor空间，当进行回收时，将还存活的对象复制到另一块Survivor空间中，然后清理掉Eden和A空间。在进行了第一次GC之后，使用的便是Eden space和B空间了，下次GC时会将存活对象复制到A空间，如此反复循环。

当对象在Survivor区躲过一次GC的话，其对象年龄便会加1，默认情况下，对象年龄达到15时，就会移动到老年代中。一般来说，大对象会被直接分配到老年代，所谓的大对象是指需要大量连续存储空间的对象，最常见的一种大对象就是大数组，比如：byte[] data = new byte[4\*1024\*1024]。

当然分配的规则并不是百分之百固定的，这要取决于当前使用的是哪种垃圾收集器组合和JVM的相关参数。这些搬运工作都是GC完成的，GC不仅负责在Heap中搬运实例，同时负责回收存储空间。最后，因为每次回收都只回收少量对象，所以老年代一般使用的是标记整理算法。

注意：在方法区中有一个永久代（Permanet Generation），它用来存储class文件、静态方法、方法描述等。对永久代的回收主要回收两部分内容：废弃常量和无用的类。



Minor GC是新生代Copying算法。MinorGC触发条件：

1. 当Eden区满时，触发MinorGC。

Full GC的老年代，采取的Mark-Compact。Full GC触发条件：

1. 调用System.gc时，系统建议执行Full GC，但是不必然执行。
2. 老年代空间不足。
3. 方法区空间不足。
4. 通过Minor GC后进入老年代的平均大小大于老年代的可用内存。

有关查看垃圾回收信息的JVM常见配置方式：

-XX:+PrintGCDetails

最后介绍一下有关堆的JVM常见配置方式：

-Xss //选置栈内存的大小

-Xms: //初始堆大小

-Xmx: //最大堆大小

-XX:NewSize=n: //设置年轻代大小

-XX:NewRatio=n: //设置年轻代和年老代的比值。比如设置为3，表示年轻代与年老代比值为1：3

-XX:SurvivorRatio=n: //年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。比如设置为3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的1/5。

-XX:MaxPermSize=n: //设置持久代大小

1. **典型的垃圾回收器**

垃圾收集算法是内存回收的理论基础，而垃圾收集器就是内存回收的具体实现。

下面介绍一下HotSpot（JDK7）虚拟机提供的几种垃圾收集器，用户可以根据自己的需求组合出各个年代使用的收集器。

**5.1 Serial&Serial Old**

Serial和Serial Old收集器是最基本最古老的收集器，是一个单线程收集器，并且在它进行垃圾收集时，必须暂停所有用户线程。Serial收集器是针对新生代的收集器，采用的是Copying算法，Serial Old收集器是针对老年代的收集器，采用的是Mark-Compact算法。它的优点是实现简单高效，但是缺点是会给用户带来停顿。

**5.2 ParNew**

ParNew收集器是Serial收集器的多线程版本，使用多个线程进行垃圾收集。

**5.3 Parallel Scavenge**

Parallel Scavenge收集器是一个新生代的多线程收集器（并行收集器），其采用的是Copying算法，该收集器与前两个收集器有所不同，它主要是为了达到一个可控的吞吐量。

**5.4 Parallel Old**

Parallel Old是Parallel Scavenge收集器的老年代版本（并行收集器），使用多线程和Mark-Compact算法。

**5.5 CMS**

CMS（Current Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器，它是一种并发收集器，采用的是Mark-Sweep算法。

**5.6 G1**

G1收集器是当今收集器技术发展最前沿的成果，它是一款面向服务端应用的收集器，它能充分利用多CPU、多核环境。因此它是一款并行与并发收集器，并且它能建立可预测的停顿时间模型。

最后介绍一下有关收集器设置的JVM常见配置方式：

-XX:+UseSerialGC://设置串行收集器

-XX:+UseParallelGC://设置并行收集器

-XX:+UseParalledlOldGC://设置并行年老代收集器

-XX:+UseConcMarkSweepGC://设置并发收集器

//并行收集器设置

-XX:ParallelGCThreads=n://设置并行收集器收集时使用的CPU数，并行收集线程数

-XX:MaxGCPauseMillis=n://设置并行收集最大暂停时间

-XX:GCTimeRatio=n://设置垃圾回收时间占程序运行时间的百分比，公式为1/(1+n)

//并发收集器设置

-XX:+CMSIncrementalMode://设置为增量模式。适用于单CPU情况

-XX:ParallelGCThreads=n://设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数