与C/C++相比，JAVA并不要求我们去人为编写代码进行内存回收和垃圾清理。JAVA提供了垃圾回收器（garbage collector)来自动检测对象的作用域），可自动把不再被使用的存储空间释放掉，也就是说，GC机制可以有效地防止内存泄露以及内存溢出。

**JAVA 垃圾回收器的主要任务是**：

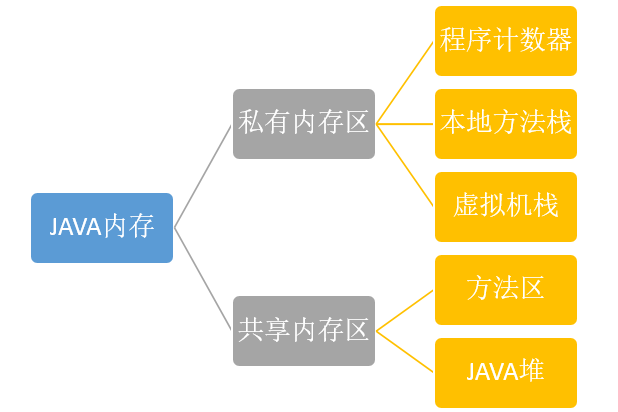
* 分配内存
* 确保被引用对象的内存不被错误地回收
* 回收不再被引用的对象的内存空间

凡事都有两面性。垃圾回收器在把程序员从释放内存的复杂工作中解放出来的同时，为了实现垃圾回收，garbage collector必须跟踪内存的使用情况，释放没用的对象，在完成内存的释放之后还需要处理堆中的碎片， 这样做必定会增加JVM的负担。

为什么要了解JAVA的GC机制? 综上所述，除了作为一个程序员，精益求精是基本要求之外，深入了解GC机制让我们的代码更有效率，尤其是在构建大型程序时，GC直接影响着内存优化和运行速度。

## 2 JAVA 内存区域

了解GC机制之前，需要首先搞清楚JAVA程序在执行的时候，内存究竟是如何划分的。



私有内存区的区域名称和相应的特性如下表所示：

| **区域名称** | **特性** |
| --- | --- |
| 程序计数器 | 指示当前程序执行到了哪一行，执行JAVA方法时纪录正在执行的虚拟机字节码指令地址；执行本地方法时，计数器值为undefined |
| 虚拟机栈 | 用于执行JAVA方法。栈帧存储局部变量表、操作数栈、动态链接、方法返回地址和一些额外的附加信息。程序执行时栈帧入栈；执行完成后栈帧出栈 |
| 本地方法栈 | 用于执行本地方法，其它和虚拟机栈类似 |

着重说一下虚拟机栈中的局部变量表，里面存放了三个信息：

* 各种基本数据类型（boolean、byte、char、short、int、float、long、double）
* 对象引用（reference）
* returnAddress地址

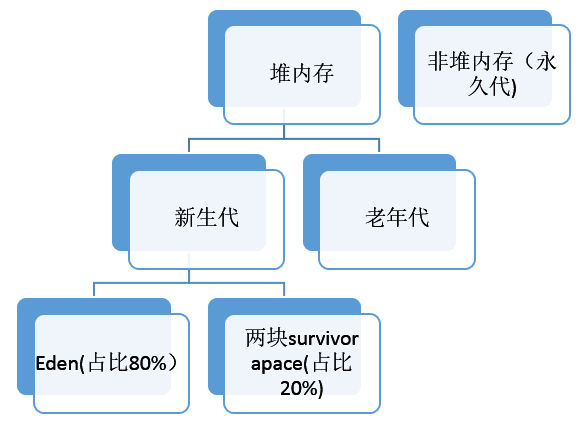
这个returnAddress和程序计数器有什么区别？前者是指示JVM的指令执行到哪一行，后者则是你的代码执行到哪一行。

**私有内存区伴随着线程的产生而产生，一旦线程中止，私有内存区也会自动消除，因此我们在本文中讨论的内存回收主要是针对共享内存区。**下面介绍一下共享内存区。

| **区域名称** | **特性** |
| --- | --- |
| JAVA堆 | JAVA虚拟机管理的内存中最大的一块，所有线程共享，几乎所有的对象实例和数组都在这类分配内存。**GC主要就是在JAVA堆中进行的**。 |
| 方法区 | 用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。但是已经被最新的 JVM 取消了。现在，被加载的类作为元数据加载到底层操作系统的本地内存区。 |

## 3 JAVA堆

既然GC主要发生在堆内存中，这部分我们会对堆内存进行比较详细的描述。

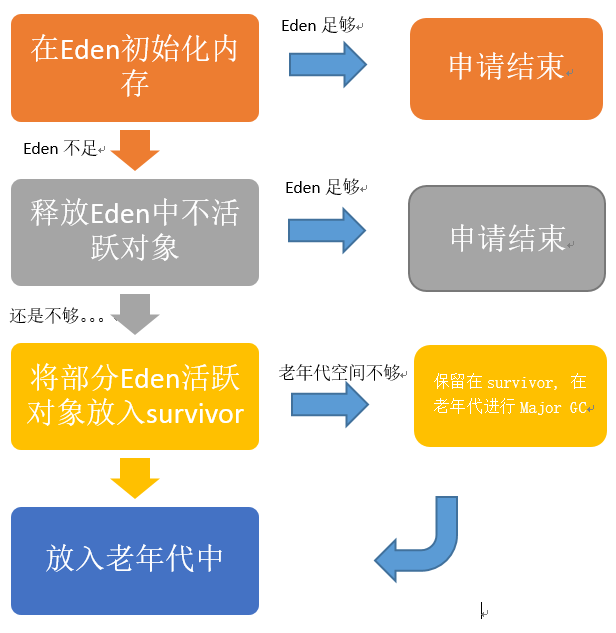
堆内存是由存活和死亡的对象组成的。存活的对象是应用可以访问的，不会被垃圾回收。死亡的对象是应用不可访问尚且还没有被垃圾收集器回收掉的对象。一直到垃圾收集器把这些对象回收掉之前，他们会一直占据堆内存空间。堆是应用程序在运行期请求操作系统分配给自己的向高地址扩展的数据结构，是不连续的内存区域。**用一句话总结堆的作用：程序运行时动态申请某个大小的内存空间**。   
   
**新生代：**刚刚新建的对象在Eden中，经历一次Minor GC，Eden中的存活对象就会被移动到第一块survivor space S0，Eden被清空；等Eden区再满了，就再触发一次Minor GC，Eden和S0中的存活对象又会被复制送入第二块survivor space S1。S0和Eden被清空，然后下一轮S0与S1交换角色，如此循环往复。如果对象的复制次数达到16次，该对象就会被送到老年代中。

至于为什么新生代要分出两个survivor区，在我的另一篇博客中有详细介绍[为什么新生代内存需要有两个Survivor区](http://blog.csdn.net/antony9118/article/details/51425581" \t "http://blog.csdn.net/antony9118/article/details/_blank)

**老年代：**如果某个对象经历了几次垃圾回收之后还存活，就会被存放到老年代中。老年代的空间一般比新生代大。

| **GC名称** | **介绍** |
| --- | --- |
| Minor GC | 发生在新生代，频率高，速度快（大部分对象活不过一次Minor GC） |
| Major GC | 发生在老年代，速度慢 |
| Full GC | 清理整个堆空间 |

不过实际运行中，Major GC会伴随至少一次 Minor GC，因此也不必过多纠结于到底是哪种GC（在有些资料中看到把full GC和Minor GC等价的说法)。

那么，当我们创建一个对象后，它会被放在堆内存的哪个部分呢?   
   
如果Major GC之后还是老年代不足，那苍天也救不了了。。。。JVM会抛出内存不足的异常。

## 4 垃圾回收机制

JAVA 并没有给我们提供明确的代码来标注一块内存并将其回收。或许你会说，我们可以将相关对象设为 null 或者用 System.gc()。然而，后者将会严重影响代码的性能，因为一般每一次显式的调用 system.gc() 都会停止所有响应，去检查内存中是否有可回收的对象。这会对程序的正常运行造成极大的威胁。另外，调用该方法并不能保证 JVM 立即进行垃圾回收，仅仅是通知 JVM 要进行垃圾回收了，具体回收与否完全由 JVM 决定。这样做是费力不讨好。

垃圾回收器是利用有向图来记录和管理内存中的所有对象，通过该有向图，就可以识别哪些对象“可达”，哪些对象“不可达”，“不可达”的对象就是可以被回收的。这里举一个很简单的例子来说明这个原理：

public class Test{

public static void main(String[] a){

Integer n1=new Integer(9);

Integer n2=new Integer(3);

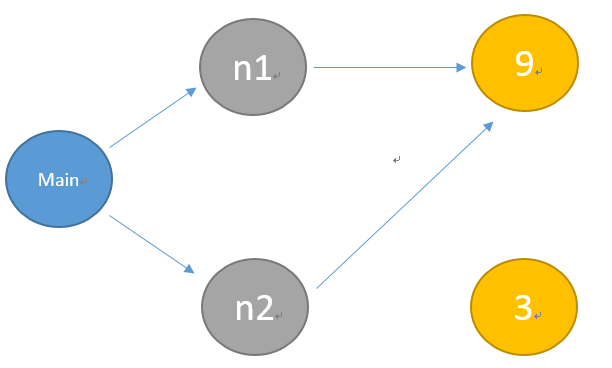
n2=n1;

// other codes

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

   
如上图所示，垃圾回收器在遍历有向图时，资源2所占的内存不可达，垃圾回收器就会回收该块内存空间。

### 4.1 垃圾回收算法概述

**追踪回收算法（tracing collector）**   
从根结点开始遍历对象的应用图。同时标记遍历到的对象。遍历完成后，没有被标记的对象就是目前未被引用，可以被回收。

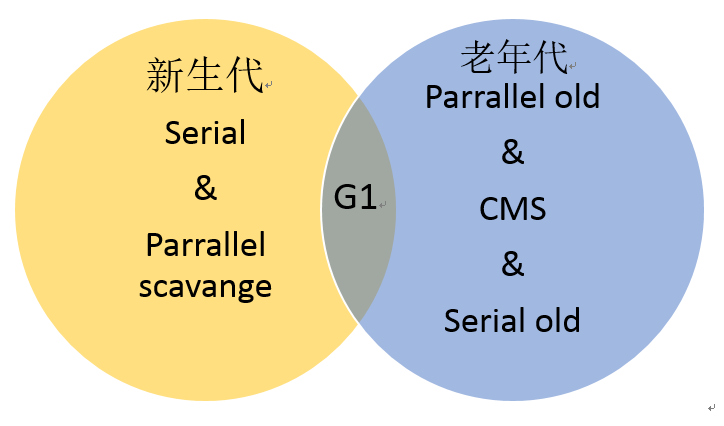
**压缩回收算法（Compacting Collector）**   
把堆中活动的对象集中移动到堆的一端，就会在堆的另一端流出很大的空闲区域。这种处理简化了消除碎片的工作，但可能带来性能的损失。

**复制回收算法（Coping Collector)**   
把堆均分成两个大小相同的区域，只使用其中的一个区域，直到该区域消耗完。此时垃圾回收器终端程序的执行，通过遍历把所有活动的对象复制到另一个区域，复制过程中它们是紧挨着布置的，这样也可以达到消除内存碎片的目的。复制结束后程序会继续运行，直到该区域被用完。   
但是，这种方法有两个缺陷:

1. 对于指定大小的堆，需要两倍大小的内存空间，
2. 需要中断正在执行的程序，降低了执行效率

**按代回收算法（Generational Collector)**   
为什么要按代进行回收？这是因为不同对象生命周期不同，每次回收都要遍历所有存活对象，对于整个堆内存进行回收无疑浪费了大量时间，对症下药可以提高垃圾回收的效率。主要思路是：把堆分成若搞个子堆，每个子堆视为一代，算法在运行的过程中优先收集“年幼”的对象，如果某个对象经过多次回收仍然“存活”,就移动到高一级的堆，减少对其扫描次数。

### 4.2 垃圾回收器



| **回收器** | **概述** | **年轻代** | **老年代** |
| --- | --- | --- | --- |
| 串行回收器（serial collector） | 客户端模式的默认回收器，所谓的串行，指的就是单线程回收，回收时将会暂停所有应用线程的执行 | 参见本文第三部分 | serial old回收器**标记-清除-合并**。标记所有存活对象，从头遍历堆，清除所有死亡对象，最后把存活对象移动到堆的前端，堆的后端就空了 |
| 并行回收器 | 服务器模式的默认回收器，利用多个线程进行垃圾回收，充分利用CPU，回收期间暂停所有应用线程 | Parallel Scavenge回收器，关注可控制的吞吐量（吞吐量=代码运行时间/（代码运行时间加垃圾回收时间）。吞吐量越大，垃圾回收时间越短，可以充分利用CPU。但是 | parrellel old回收器，多线程，同样采取“标记-清除-合并”。特点是“吞吐量优先” |
| CMS回收器 | 停顿时间最短，分为以下步骤：1初始标记；2并发标记；3重新标记；4并发清除。优点是停顿时间短，并发回收，缺点是无法处理浮动垃圾，而且会导致空间碎片产生 | X | 适用 |
| G1回收器 | 新技术，将堆内存划分为多个等大的区域，按照每个区域进行回收。工作过程是1初始标记；2并发标记；3最终标记；4筛选回收。特点是并行并发，分代收集，不会导致空间碎片，也可以由编程者自主确定停顿时间上限 | 适用 | 适用 |

附转载的GC参数汇总以及一个使用实例，转载来源是   
[JVM垃圾回收器工作原理及使用实例介绍](https://www.ibm.com/developerworks/cn/java/j-lo-JVMGarbageCollection/" \t "http://blog.csdn.net/antony9118/article/details/_blank)   
**1. 与串行回收器相关的参数**   
-XX:+UseSerialGC:在新生代和老年代使用串行回收器。   
-XX:+SuivivorRatio:设置 eden 区大小和 survivor 区大小的比例。   
-XX:+PretenureSizeThreshold:设置大对象直接进入老年代的阈值。当对象的大小超过这个值时，将直接在老年代分配。   
-XX:MaxTenuringThreshold:设置对象进入老年代的年龄的最大值。每一次 Minor GC 后，对象年龄就加 1。任何大于这个年龄的对象，一定会进入老年代。   
**2. 与并行 GC 相关的参数**   
-XX:+UseParNewGC: 在新生代使用并行收集器。   
-XX:+UseParallelOldGC: 老年代使用并行回收收集器。   
-XX:ParallelGCThreads：设置用于垃圾回收的线程数。通常情况下可以和 CPU 数量相等。但在 CPU 数量比较多的情况下，设置相对较小的数值也是合理的。   
-XX:MaxGCPauseMills：设置最大垃圾收集停顿时间。它的值是一个大于 0 的整数。收集器在工作时，会调整 Java 堆大小或者其他一些参数，尽可能地把停顿时间控制在 MaxGCPauseMills 以内。   
-XX:GCTimeRatio:设置吞吐量大小，它的值是一个 0-100 之间的整数。假设 GCTimeRatio 的值为 n，那么系统将花费不超过 1/(1+n) 的时间用于垃圾收集。   
-XX:+UseAdaptiveSizePolicy:打开自适应 GC 策略。在这种模式下，新生代的大小，eden 和 survivor 的比例、晋升老年代的对象年龄等参数会被自动调整，以达到在堆大小、吞吐量和停顿时间之间的平衡点。   
**3. 与 CMS 回收器相关的参数**   
-XX:+UseConcMarkSweepGC: 新生代使用并行收集器，老年代使用 CMS+串行收集器。   
-XX:+ParallelCMSThreads: 设定 CMS 的线程数量。   
-XX:+CMSInitiatingOccupancyFraction:设置 CMS 收集器在老年代空间被使用多少后触发，默认为 68%。   
-XX:+UseFullGCsBeforeCompaction:设定进行多少次 CMS 垃圾回收后，进行一次内存压缩。   
-XX:+CMSClassUnloadingEnabled:允许对类元数据进行回收。   
-XX:+CMSParallelRemarkEndable:启用并行重标记。   
-XX:CMSInitatingPermOccupancyFraction:当永久区占用率达到这一百分比后，启动 CMS 回收 (前提是-XX:+CMSClassUnloadingEnabled 激活了)。   
-XX:UseCMSInitatingOccupancyOnly:表示只在到达阈值的时候，才进行 CMS 回收。   
-XX:+CMSIncrementalMode:使用增量模式，比较适合单 CPU。   
**4. 与 G1 回收器相关的参数**   
-XX:+UseG1GC：使用 G1 回收器。   
-XX:+UnlockExperimentalVMOptions:允许使用实验性参数。   
-XX:+MaxGCPauseMills:设置最大垃圾收集停顿时间。   
-XX:+GCPauseIntervalMills:设置停顿间隔时间。   
**5. 其他参数**   
-XX:+DisableExplicitGC: 禁用显示 GC。

常用参数如下



**调优实例**

import java.util.HashMap;

public class GCTimeTest {

static HashMap map = new HashMap();

public static void main(String[] args){

long begintime = System.currentTimeMillis();

for(int i=0;i<10000;i++){

if(map.size()\*512/1024/1024>=400){

map.clear();//保护内存不溢出

System.out.println("clean map");

}

byte[] b1;

for(int j=0;j<100;j++){

b1 = new byte[512];

map.put(System.nanoTime(), b1);//不断消耗内存

}

}

long endtime = System.currentTimeMillis();

System.out.println(endtime-begintime);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23

通过上面的代码运行 1 万次循环，每次分配 512\*100B 空间，采用不同的垃圾回收器，输出程序运行所消耗的时间。   
使用参数-Xmx512M -Xms512M -XX:+UseParNewGC 运行代码，输出如下：   
clean map 8565   
cost time=1655   
使用参数-Xmx512M -Xms512M -XX:+UseParallelOldGC –XX:ParallelGCThreads=8 运行代码，输出如下：   
clean map 8798   
cost time=1998

## 5 JAVA性能优化

大多说针对内存的调优，都是针对于特定情况的。但是实际中，调优很难与JAVA运行动态特性的实际情况和工作负载保持一致。也就是说，几乎不可能通过单纯的调优来达到消除GC的目的。

真正影响JAVA程序性能的，就是碎片化。碎片是JAVA堆内存中的空闲空间，可能是TLAB剩余空间，也可能是被释放掉的具有较长生命周期的小对象占用的空间。

下面是一些在实际写程序的过程中应该注意的点，养成这些习惯可以在一定程度上减少内存的无谓消耗，进一步就可以减少因为内存不足导致GC不断。类似的这种经验可以多积累交流：

1. 减少new对象。每次new对象之后，都要开辟新的内存空间。这些对象不被引用之后，还要回收掉。因此，如果最大限度地合理重用对象，或者使用基本数据类型替代对象，都有助于节省内存；
2. 多使用局部变量，减少使用静态变量。局部变量被创建在栈中，存取速度快。静态变量则是在堆内存；
3. 避免使用finalize，该方法会给GC增添很大的负担；
4. 如果是单线程，尽量使用非多线程安全的，因为线程安全来自于同步机制，同步机制会降低性能。例如，单线程程序，能使用HashMap，就不要用HashTable。同理，尽量减少使用synchronized
5. 用移位符号替代乘除号。eg：a\*8应该写作a<<3
6. 对于经常反复使用的对象使用缓存；
7. 尽量使用基本类型而不是包装类型，尽量使用一维数组而不是二维数组；
8. 尽量使用final修饰符，final表示不可修改，访问效率高
9. 单线程情况下（或者是针对于局部变量），字符串尽量使用StringBuilder，比StringBuffer要快；
10. 尽量使用StringBuffer来连接字符串。这里需要注意的是，StringBuffer的默认缓存容量是16个字符，如果超过16，apend方法调用私有的expandCapacity()方法，来保证足够的缓存容量。因此，如果可以预设StringBuffer的容量，避免append再去扩展容量，示例下面两个示例·：

示例一：

StringBuffer st = new StringBuffer(50);st.append("let us cook");st.append(" ");st.append("a matcha cake for our dinner");

String s = st.toString();

* 1
* 2
* 3
* 4
* 5

示例二：

public String toString() {

return new StringBuilder().append("[").append(name).append("]")

.append("[").append(Message).append("]")

.append("[").append(salary).append("]").toString();

}