# JVM虚拟机

http://blog.csdn.net/hsk256/article/details/49104955

http://blog.csdn.net/ithomer/article/details/6252552

## JVM内存申请过程

1、JVM 会试图为相关Java对象在Eden中初始化一块内存区域

2、当Eden空间足够时，内存申请结束；否则到下一步

3、JVM 试图释放在Eden中所有不活跃的对象（这属于1或更高级的垃圾回收）,

释放后若Eden空间仍然不足以放入新对象，则试图将部分Eden中活跃对象放入Survivor区

Survivor区被用来作为Eden及OLD的中间交换区域，当OLD区空间足够时，

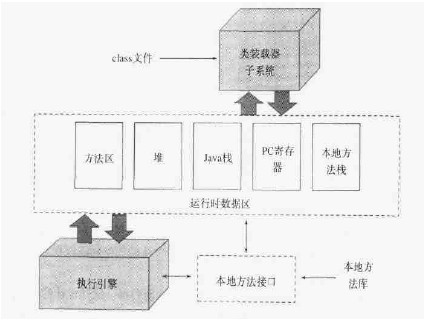
Survivor区的对象会被移到Old区，否则会被保留在Survivor区

当OLD区空间不够时，JVM 会在OLD区进行完全的垃圾收集（0级）

完全垃圾收集后，若Survivor及OLD区仍然无法存放从Eden复制过来的部分对象，

导致JVM无法在Eden区为新对象创建内存区域，则出现”out of memory”错误

## java内存模型



### 方法区（永久代）

方法区：

1. 有时候也成为永久代，在该区内很少发生垃圾回收，但是并不代表不发生GC，

在这里进行的GC主要是对方法区里的常量池和对类型的卸载

2. 方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。

3. 该区域是被线程共享的。

4. 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。

该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

### 虚拟机栈:

1. 虚拟机栈也就是我们平常所称的栈内存,它为java方法服务，

每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。

2. 虚拟机栈是线程私有的，它的生命周期与线程相同。

3. 局部变量表里存储的是基本数据类型、returnAddress类型（指向一条字节码指令的地址）

和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者

与对象相关联的位置。局部变量所需的内存空间在编译器间确定

4.操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，

而是压栈和出栈的方式

5.每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，

持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用

在运行期转化为直接引用。

### 本地方法栈

本地方法栈和虚拟机栈类似，只不过本地方法栈为Native方法服务。

### 堆

java堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

### 程序计数器

内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个java虚拟机规范没有规定任何OOM情况的区域。

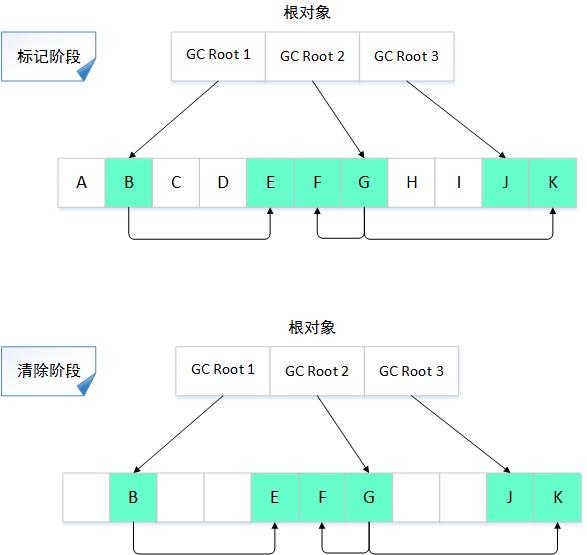
## GC算法

### **最基础的收集算法 —— 标记/清除算法**

之所以说标记/清除算法是几种GC算法中最基础的算法，是因为后续的收集算法都是基于这种思路并对其不足进行改进而得到的。标记/清除算法的基本思想就跟它的名字一样，分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

       标记阶段：标记的过程其实就是前面介绍的可达性分析算法的过程，遍历所有的GC Roots对象，对从GC Roots对象可达的对象都打上一个标识，一般是在对象的header中，将其记录为可达对象；

       清除阶段：清除的过程是对堆内存进行遍历，如果发现某个对象没有被标记为可达对象（通过读取对象header信息），则将其回收。



       上图是标记/清除算法的示意图，在标记阶段，从对象GC Root 1可以访问到B对象，从B对象又可以访问到E对象，因此从GC Root 1到B、E都是可达的，同理，对象F、G、J、K都是可达对象；到了清除阶段，所有不可达对象都会被回收。

       在垃圾收集器进行GC时，必须停止所有Java执行线程（也称"Stop The World"），原因是在标记阶段进行可达性分析时，不可以出现分析过程中对象引用关系还在不断变化的情况，否则的话可达性分析结果的准确性就无法得到保证。在等待标记清除结束后，应用线程才会恢复运行。

       前面刚提过，后续的收集算法是在标记/清除算法的基础上进行改进而来的，那也就是说标记/清除算法有它的不足。其实了解了它的原理，其缺点也就不难看出了。

       1、效率问题。标记和清除两个阶段的效率都不高，因为这两个阶段都需要遍历内存中的对象，很多时候内存中的对象实例数量是非常庞大的，这无疑很耗费时间，而且GC时需要停止应用程序，这会导致非常差的用户体验。

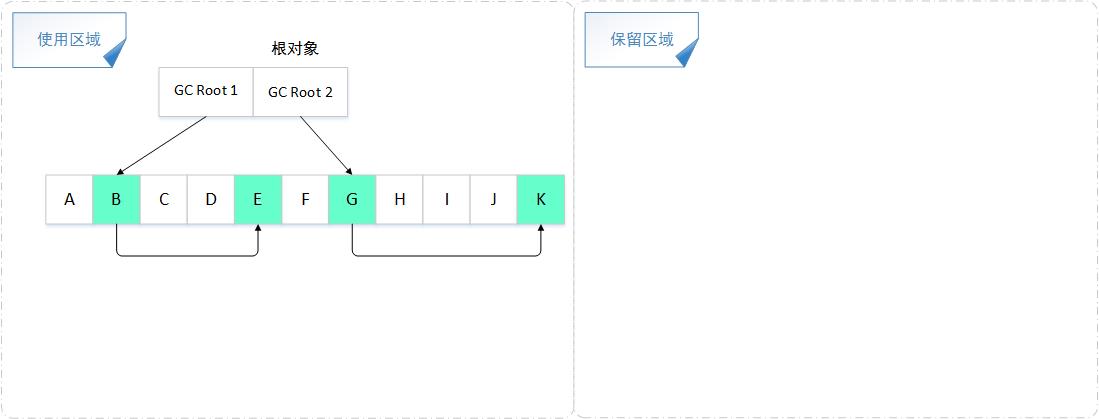
       2、空间问题。标记清除之后会产生大量不连续的内存碎片（从上图可以看出），内存空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾回收动作。

       既然标记/清除算法有这么多的缺点，那它还有存在的意义吗？别急，一个算法有缺陷，人们肯定会想办法去完善它，接下来的两个算法就是在标记/清除算法的基础上完善而来的。

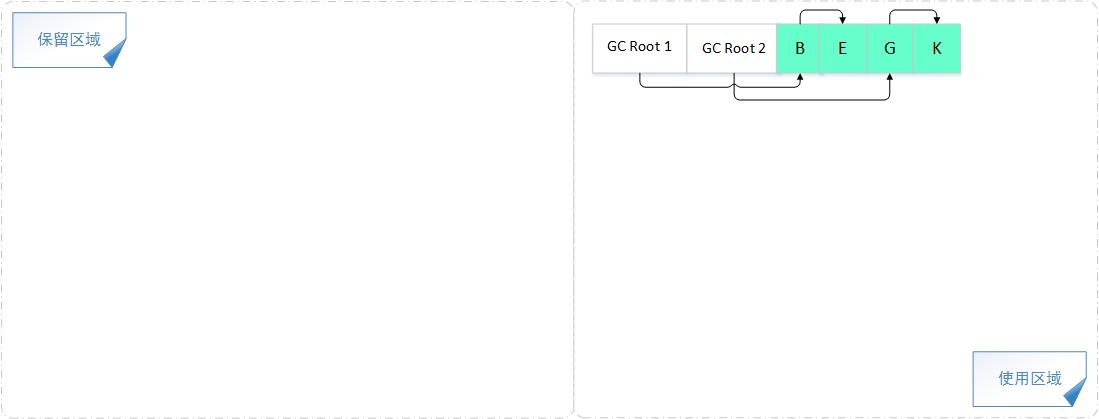
### **复制算法**

       为了解决效率问题，复制算法出现了。复制算法的原理是：将可用内存按容量划分为大小相等的两块，每次使用其中的一块。当这一块的内存用完了，就将还存活的对象复制到另一块内存上，然后把这一块内存所有的对象一次性清理掉。用图说明如下：

       回收前：



       回收后：



        复制算法每次都是对整个半区进行内存回收，这样就减少了标记对象遍历的时间，在清除使用区域对象时，不用进行遍历，直接清空整个区域内存，而且在将存活对象复制到保留区域时也是按地址顺序存储的，这样就解决了内存碎片的问题，在分配对象内存时不用考虑内存碎片等复杂问题，只需要按顺序分配内存即可。

       复制算法简单高效，优化了标记/清除算法的效率低、内存碎片多的问题。但是它的缺点也很明显：

       1、将内存缩小为原来的一半，浪费了一半的内存空间，代价太高；

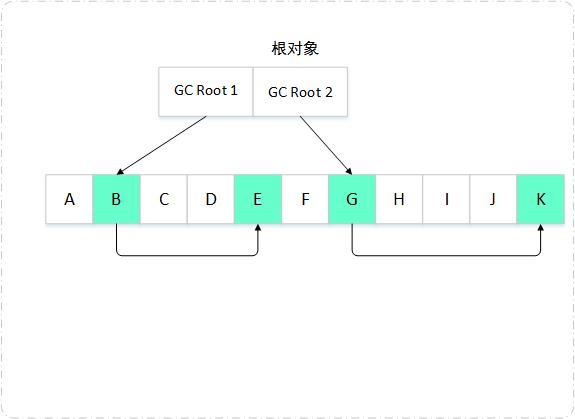
       2、如果对象的存活率很高，极端一点的情况假设对象存活率为100%，那么我们需要将所有存活的对象复制一遍，耗费的时间代价也是不可忽视的。

       基于以上复制算法的缺点，由于新生代中的对象几乎都是“朝生夕死”的（达到98%），现在的商业虚拟机都采用复制算法来回收新生代。由于新生代的对象存活率低，所以并不需要按照1：1的比例来划分内存空间，而是将内存分为一块较大的Eden空间和两块较小的From Survivor空间、To Survivor空间，三者的比例为8：1：1。每次使用Eden和From Survivor区域，To Survivor作为保留空间。GC开始时，对象只会存在于Eden区和From Survivor区，To Survivor区是空的。GC进行时，Eden区中所有存活的对象都会被复制到To Survivor区，而在From Survivor区中，仍存活的对象会根据它们的年龄值决定去向，年龄值达到年龄阀值（默认为15，新生代中的对象每熬过一轮垃圾回收，年龄值就加1）的对象会被移到老年代中，没有达到阀值的对象会被复制到To Survivor区。接着清空Eden区和From Survivor区，新生代中存活的对象都在To Survivor区。接着， From Survivor区和To Survivor区会交换它们的角色，也就是新的To Survivor区就是上次GC清空的From Survivor区，新的From Survivor区就是上次GC的To Survivor区，总之，不管怎样都会保证To Survivor区在一轮GC后是空的。GC时当To Survivor区没有足够的空间存放上一次新生代收集下来的存活对象时，需要依赖老年代进行分配担保，将这些对象存放在老年代中。

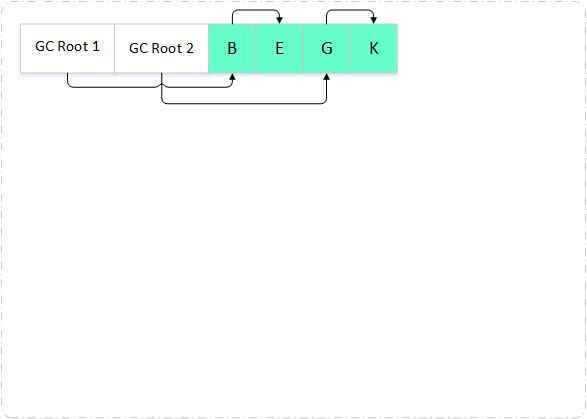
### **标记/整理算法**

复制算法在对象存活率较高时要进行较多的复制操作，效率会变得很低，更关键的是，如果不想浪费50%的内存空间，就需要有额外的内存空间进行分配担保，以应对内存中对象100%存活的极端情况，因此，在老年代中由于对象的存活率非常高，复制算法就不合适了。根据老年代的特点，高人们提出了另一种算法：标记/整理算法。从名字上看，这种算法与标记/清除算法很像，事实上，标记/整理算法的标记过程任然与标记/清除算法一样，但后续步骤不是直接对可回收对象进行回收，而是让所有存活的对象都向一端移动，然后直接清理掉端边线以外的内存。

       回收前：



       回收后：



       可以看到，回收后可回收对象被清理掉了，存活的对象按规则排列存放在内存中。这样一来，当我们给新对象分配内存时，jvm只需要持有内存的起始地址即可。标记/整理算法不仅弥补了标记/清除算法存在内存碎片的问题，也消除了复制算法内存减半的高额代价，可谓一举两得。但任何算法都有缺点，就像人无完人，标记/整理算法的缺点就是效率也不高，不仅要标记存活对象，还要整理所有存活对象的引用地址，在效率上不如复制算法。

       弄清了以上三种算法的原理，下面我们来从几个方面对这几种算法做一个简单排行。

**效率**：复制算法 > 标记/整理算法 > 标记/清除算法（标记/清除算法有内存碎片问题，给大对象分配内存时可能会触发新一轮垃圾回收）

**内存整齐率**：复制算法 = 标记/整理算法 > 标记/清除算法

**内存利用率**：标记/整理算法 = 标记/清除算法 > 复制算法

       从上面简单的评估可以看出，标记/清除算法已经比较落后了，但是吃水不忘挖井人，它是后面几种算法的前辈、是基础，在某些场景下它也有用武之地。

### **终极算法 —— 分代收集算法**

当前商业虚拟机都采用分代收集算法，说它是终极算法，是因为它结合了前几种算法的优点，将算法组合使用进行垃圾回收，与其说它是一种新的算法，不如说它是对前几种算法的实际应用。分代收集算法的思想是按对象的存活周期不同将内存划分为几块，一般是把Java堆分为新生代和老年代（还有一个永久代，是HotSpot特有的实现，其他的虚拟机实现没有这一概念，永久代的收集效果很差，一般很少对永久代进行垃圾回收），这样就可以根据各个年代的特点采用最合适的收集算法。

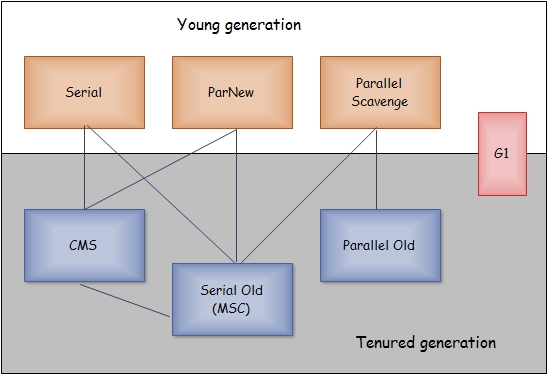
       新生代：朝生夕灭，存活时间很短。

       老年代：经过多次Minor GC而存活下来，存活周期长。

       在新生代中每次垃圾回收都发现有大量的对象死去，只有少量存活，因此采用复制算法回收新生代，只需要付出少量对象的复制成本就可以完成收集；而老年代中对象的存活率高，不适合采用复制算法，而且如果老年代采用复制算法，它是没有额外的空间进行分配担保的，因此必须使用标记/清理算法或者标记/整理算法来进行回收。

       总结一下就是，分代收集算法的原理是采用复制算法来收集新生代，采用标记/清理算法或者标记/整理算法收集老年代。

## JVM收集器



              上面有7中收集器，分为两块，上面为新生代收集器，下面是老年代收集器。如果两个收集器之间存在连线，就说明它们可以搭配使用。

### Serial(串行GC)收集器

Serial收集器是一个新生代收集器，单线程执行，使用复制算法。它在进行垃圾收集时，必须暂停其他所有的工作线程(用户线程)。是Jvm client模式下默认的新生代收集器。对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。

### ParNew(并行GC)收集器

ParNew收集器其实就是serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为与Serial收集器一样。

### Parallel Scavenge(并行回收GC)收集器

Parallel Scavenge收集器也是一个新生代收集器，它也是使用复制算法的收集器，又是并行多线程收集器。parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而parallel Scavenge收集器的目标则是达到一个可控制的吞吐量。吞吐量= 程序运行时间/(程序运行时间 + 垃圾收集时间)，虚拟机总共运行了100分钟。其中垃圾收集花掉1分钟，那吞吐量就是99%。

### Serial Old(串行GC)收集器

Serial Old是Serial收集器的老年代版本，它同样使用一个单线程执行收集，使用“标记-整理”算法。主要使用在Client模式下的虚拟机。

### Parallel Old(并行GC)收集器

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。

### CMS(并发GC)收集器

CMS(Concurrent 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。CMS收集器是基于“标记-清除”算法实现的，整个收集过程大致分为4个步骤：

①.初始标记(CMS initial mark)

②.并发标记(CMS concurrenr mark)

③.重新标记(CMS remark)

④.并发清除(CMS concurrent sweep)

     其中初始标记、重新标记这两个步骤任然需要停顿其他用户线程。初始标记仅仅只是标记出GC ROOTS能直接关联到的对象，速度很快，并发标记阶段是进行GC ROOTS 根搜索算法阶段，会判定对象是否存活。而重新标记阶段则是为了修正并发标记期间，因用户程序继续运行而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间会被初始标记阶段稍长，但比并发标记阶段要短。

     由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以整体来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

CMS收集器的优点：并发收集、低停顿，但是CMS还远远达不到完美，器主要有三个显著缺点：

CMS收集器对CPU资源非常敏感。在并发阶段，虽然不会导致用户线程停顿，但是会占用CPU资源而导致引用程序变慢，总吞吐量下降。CMS默认启动的回收线程数是：(CPU数量+3) / 4。

CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure“，失败后而导致另一次Full  GC的产生。由于CMS并发清理阶段用户线程还在运行，伴随程序的运行自热会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在本次收集中处理它们，只好留待下一次GC时将其清理掉。这一部分垃圾称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，  
即需要预留足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分内存空间提供并发收集时的程序运作使用。在默认设置下，CMS收集器在老年代使用了68%的空间时就会被激活，也可以通过参数-XX:CMSInitiatingOccupancyFraction的值来提供触发百分比，以降低内存回收次数提高性能。要是CMS运行期间预留的内存无法满足程序其他线程需要，就会出现“Concurrent Mode Failure”失败，这时候虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。所以说参数-XX:CMSInitiatingOccupancyFraction设置的过高将会很容易导致“Concurrent Mode Failure”失败，性能反而降低。

最后一个缺点，CMS是基于“标记-清除”算法实现的收集器，使用“标记-清除”算法收集后，会产生大量碎片。空间碎片太多时，将会给对象分配带来很多麻烦，比如说大对象，内存空间找不到连续的空间来分配不得不提前触发一次Full  GC。为了解决这个问题，CMS收集器提供了一个-XX:UseCMSCompactAtFullCollection开关参数，用于在Full  GC之后增加一个碎片整理过程，还可通过-XX:CMSFullGCBeforeCompaction参数设置执行多少次不压缩的Full  GC之后，跟着来一次碎片整理过程。

### G1收集器

G1(Garbage First)收集器是JDK1.7提供的一个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。还有一个特点之前的收集器进行收集的范围都是整个新生代或老年代，而G1将整个Java堆(包括新生代，老年代)。

### 垃圾收集器参数总结

-XX:+<option> 启用选项

-XX:-<option> 不启用选项

-XX:<option>=<number>

-XX:<option>=<string>

|  |  |
| --- | --- |
| **参数** | **描述** |
| -XX:+UseSerialGC | Jvm运行在Client模式下的默认值，打开此开关后，使用Serial + Serial Old的收集器组合进行内存回收 |
| -XX:+UseParNewGC | 打开此开关后，使用ParNew + Serial Old的收集器进行垃圾回收 |
| -XX:+UseConcMarkSweepGC | 使用ParNew + CMS +  Serial Old的收集器组合进行内存回收，Serial Old作为CMS出现“Concurrent Mode Failure”失败后的后备收集器使用。 |
| -XX:+UseParallelGC | Jvm运行在Server模式下的默认值，打开此开关后，使用Parallel Scavenge +  Serial Old的收集器组合进行回收 |
| -XX:+UseParallelOldGC | 使用Parallel Scavenge +  Parallel Old的收集器组合进行回收 |
| -XX:SurvivorRatio | 新生代中Eden区域与Survivor区域的容量比值，默认为8，代表Eden:Subrvivor = 8:1 |
| -XX:PretenureSizeThreshold | 直接晋升到老年代对象的大小，设置这个参数后，大于这个参数的对象将直接在老年代分配 |
| -XX:MaxTenuringThreshold | 晋升到老年代的对象年龄，每次Minor GC之后，年龄就加1，当超过这个参数的值时进入老年代 |
| -XX:UseAdaptiveSizePolicy | 动态调整java堆中各个区域的大小以及进入老年代的年龄 |
| -XX:+HandlePromotionFailure | 是否允许新生代收集担保，进行一次minor gc后, 另一块Survivor空间不足时，将直接会在老年代中保留 |
| -XX:ParallelGCThreads | 设置并行GC进行内存回收的线程数 |
| -XX:GCTimeRatio | GC时间占总时间的比列，默认值为99，即允许1%的GC时间，仅在使用Parallel Scavenge 收集器时有效 |
| -XX:MaxGCPauseMillis | 设置GC的最大停顿时间，在Parallel Scavenge 收集器下有效 |
| -XX:CMSInitiatingOccupancyFraction | 设置CMS收集器在老年代空间被使用多少后出发垃圾收集，默认值为68%，仅在CMS收集器时有效，-XX:CMSInitiatingOccupancyFraction=70 |
| -XX:+UseCMSCompactAtFullCollection | 由于CMS收集器会产生碎片，此参数设置在垃圾收集器后是否需要一次内存碎片整理过程，仅在CMS收集器时有效 |
| -XX:+CMSFullGCBeforeCompaction | 设置CMS收集器在进行若干次垃圾收集后再进行一次内存碎片整理过程，通常与UseCMSCompactAtFullCollection参数一起使用 |
| -XX:+UseFastAccessorMethods | 原始类型优化 |
| -XX:+DisableExplicitGC | 是否关闭手动System.gc |
| -XX:+CMSParallelRemarkEnabled | 降低标记停顿 |
| -XX:LargePageSizeInBytes | 内存页的大小不可设置过大，会影响Perm的大小，-XX:LargePageSizeInBytes=128m |

Client、Server模式默认GC

|  |  |  |
| --- | --- | --- |
|  | **新生代GC方式** | **老年代和持久代GC方式** |
| Client | Serial 串行GC | Serial Old 串行GC |
| Server | Parallel Scavenge  并行回收GC | Parallel Old 并行GC |

Sun/oracle JDK GC组合方式

|  |  |  |
| --- | --- | --- |
|  | **新生代GC方式** | **老年代和持久代GC方式** |
| -XX:+UseSerialGC | Serial 串行GC | Serial Old 串行GC |
| -XX:+UseParallelGC | Parallel Scavenge  并行回收GC | Serial Old  并行GC |
| -XX:+UseConcMarkSweepGC | ParNew 并行GC | CMS 并发GC  当出现“Concurrent Mode Failure”时 采用Serial Old 串行GC |
| -XX:+UseParNewGC | ParNew 并行GC | Serial Old 串行GC |
| -XX:+UseParallelOldGC | Parallel Scavenge  并行回收GC | Parallel Old 并行GC |
| -XX:+UseConcMarkSweepGC -XX:+UseParNewGC | Serial 串行GC | CMS 并发GC  当出现“Concurrent Mode Failure”时 采用Serial Old 串行GC |

参数配置：http://unixboy.iteye.com/blog/174173

## jvm调优

1堆大小设置  
JVM 中最大堆大小有三方面限制：相关操作系统的数据模型（32-bt还是64-bit）限制；系统的可用虚拟内存限制；系统的可用物理内存限制。32位系统下，一般限制在1.5G~2G；64为操作系统对内存无限制。我在Windows Server 2003 系统，3.5G物理内存，JDK5.0下测试，最大可设置为1478m。  
典型设置：

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k  
-Xmx3550m：设置JVM最大可用内存为3550M。  
-Xms3550m：设置JVM促使内存为3550m。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。  
-Xmn2g：设置年轻代大小为2G。整个JVM内存大小=年轻代大小 + 年老代大小 + 持久代大小。持久代一般固定大小为64m，所以增大年轻代后，将会减小年老代大小。此值对系统性能影响较大，Sun官方推荐配置为整个堆的3/8。  
-Xss128k：设置每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。更具应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。

java -Xmx3550m -Xms3550m -Xss128k -XX:NewRatio=4 -XX:SurvivorRatio=4 -XX:MaxPermSize=16m -XX:MaxTenuringThreshold=0  
-XX:NewRatio=4:设置年轻代（包括Eden和两个Survivor区）与年老代的比值（除去持久代）。设置为4，则年轻代与年老代所占比值为1：4，年轻代占整个堆栈的1/5  
-XX:SurvivorRatio=4：设置年轻代中Eden区与Survivor区的大小比值。设置为4，则两个Survivor区与一个Eden区的比值为2:4，一个Survivor区占整个年轻代的1/6  
-XX:MaxPermSize=16m:设置持久代大小为16m。  
-XX:MaxTenuringThreshold=0：设置垃圾最大年龄。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代。对于年老代比较多的应用，可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象再年轻代的存活时间，增加在年轻代即被回收的概论。

回收器选择  
JVM给了三种选择：串行收集器、并行收集器、并发收集器，但是串行收集器只适用于小数据量的情况，所以这里的选择主要针对并行收集器和并发收集器。默认情况下，JDK5.0以前都是使用串行收集器，如果想使用其他收集器需要在启动时加入相应参数。JDK5.0以后，JVM会根据当前[系统配置](http://java.sun.com/j2se/1.5.0/docs/guide/vm/server-class.html)进行判断。

吞吐量优先的并行收集器  
如上文所述，并行收集器主要以到达一定的吞吐量为目标，适用于科学技术和后台处理等。  
典型配置：

java -Xmx3800m -Xms3800m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20  
-XX:+UseParallelGC：选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。  
-XX:ParallelGCThreads=20：配置并行收集器的线程数，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC -XX:ParallelGCThreads=20 -XX:+UseParallelOldGC  
-XX:+UseParallelOldGC：配置年老代垃圾收集方式为并行收集。JDK6.0支持对年老代并行收集。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC  -XX:MaxGCPauseMillis=100  
-XX:MaxGCPauseMillis=100:设置每次年轻代垃圾回收的最长时间，如果无法满足此时间，JVM会自动调整年轻代大小，以满足此值。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseParallelGC  -XX:MaxGCPauseMillis=100 -XX:+UseAdaptiveSizePolicy  
-XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动选择年轻代区大小和相应的Survivor区比例，以达到目标系统规定的最低相应时间或者收集频率等，此值建议使用并行收集器时，一直打开。

响应时间优先的并发收集器  
如上文所述，并发收集器主要是保证系统的响应时间，减少垃圾收集时的停顿时间。适用于应用服务器、电信领域等。  
典型配置：

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:ParallelGCThreads=20 -XX:+UseConcMarkSweepGC -XX:+UseParNewGC  
-XX:+UseConcMarkSweepGC：设置年老代为并发收集。测试中配置这个以后，-XX:NewRatio=4的配置失效了，原因不明。所以，此时年轻代大小最好用-Xmn设置。  
-XX:+UseParNewGC:设置年轻代为并行收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。

java -Xmx3550m -Xms3550m -Xmn2g -Xss128k -XX:+UseConcMarkSweepGC -XX:CMSFullGCsBeforeCompaction=5 -XX:+UseCMSCompactAtFullCollection  
-XX:CMSFullGCsBeforeCompaction：由于并发收集器不对内存空间进行压缩、整理，所以运行一段时间以后会产生“碎片”，使得运行效率降低。此值设置运行多少次GC以后对内存空间进行压缩、整理。  
-XX:+UseCMSCompactAtFullCollection：打开对年老代的压缩。可能会影响性能，但是可以消除碎片

辅助信息  
JVM提供了大量命令行参数，打印信息，供调试使用。主要有以下一些：

-XX:+PrintGC  
输出形式：[GC 118250K->113543K(130112K), 0.0094143 secs]

                [Full GC 121376K->10414K(130112K), 0.0650971 secs]

-XX:+PrintGCDetails  
输出形式：[GC [DefNew: 8614K->781K(9088K), 0.0123035 secs] 118250K->113543K(130112K), 0.0124633 secs]

                [GC [DefNew: 8614K->8614K(9088K), 0.0000665 secs][Tenured: 112761K->10414K(121024K), 0.0433488 secs] 121376K->10414K(130112K), 0.0436268 secs]

-XX:+PrintGCTimeStamps -XX:+PrintGC：PrintGCTimeStamps可与上面两个混合使用  
输出形式：11.851: [GC 98328K->93620K(130112K), 0.0082960 secs]

-XX:+PrintGCApplicationConcurrentTime:打印每次垃圾回收前，程序未中断的执行时间。可与上面混合使用  
输出形式：Application time: 0.5291524 seconds

-XX:+PrintGCApplicationStoppedTime：打印垃圾回收期间程序暂停的时间。可与上面混合使用  
输出形式：Total time for which application threads were stopped: 0.0468229 seconds

-XX:PrintHeapAtGC:打印GC前后的详细堆栈信息  
输出形式：  
34.702: [GC {Heap before gc invocations=7:  
 def new generation   total 55296K, used 52568K [0x1ebd0000, 0x227d0000, 0x227d0000)  
eden space 49152K,  99% used [0x1ebd0000, 0x21bce430, 0x21bd0000)  
from space 6144K,  55% used [0x221d0000, 0x22527e10, 0x227d0000)  
  to   space 6144K,   0% used [0x21bd0000, 0x21bd0000, 0x221d0000)  
 tenured generation   total 69632K, used 2696K [0x227d0000, 0x26bd0000, 0x26bd0000)  
the space 69632K,   3% used [0x227d0000, 0x22a720f8, 0x22a72200, 0x26bd0000)  
 compacting perm gen  total 8192K, used 2898K [0x26bd0000, 0x273d0000, 0x2abd0000)  
   the space 8192K,  35% used [0x26bd0000, 0x26ea4ba8, 0x26ea4c00, 0x273d0000)  
    ro space 8192K,  66% used [0x2abd0000, 0x2b12bcc0, 0x2b12be00, 0x2b3d0000)  
    rw space 12288K,  46% used [0x2b3d0000, 0x2b972060, 0x2b972200, 0x2bfd0000)  
34.735: [DefNew: 52568K->3433K(55296K), 0.0072126 secs] 55264K->6615K(124928K)Heap after gc invocations=8:  
 def new generation   total 55296K, used 3433K [0x1ebd0000, 0x227d0000, 0x227d0000)  
eden space 49152K,   0% used [0x1ebd0000, 0x1ebd0000, 0x21bd0000)  
  from space 6144K,  55% used [0x21bd0000, 0x21f2a5e8, 0x221d0000)  
  to   space 6144K,   0% used [0x221d0000, 0x221d0000, 0x227d0000)  
 tenured generation   total 69632K, used 3182K [0x227d0000, 0x26bd0000, 0x26bd0000)  
the space 69632K,   4% used [0x227d0000, 0x22aeb958, 0x22aeba00, 0x26bd0000)  
 compacting perm gen  total 8192K, used 2898K [0x26bd0000, 0x273d0000, 0x2abd0000)  
   the space 8192K,  35% used [0x26bd0000, 0x26ea4ba8, 0x26ea4c00, 0x273d0000)  
    ro space 8192K,  66% used [0x2abd0000, 0x2b12bcc0, 0x2b12be00, 0x2b3d0000)  
    rw space 12288K,  46% used [0x2b3d0000, 0x2b972060, 0x2b972200, 0x2bfd0000)  
}  
, 0.0757599 secs]

-Xloggc:filename:与上面几个配合使用，把相关日志信息记录到文件以便分析。

常见配置汇总

堆设置

-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。如:为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4

-XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：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

收集器设置

-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

垃圾回收统计信息

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

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。

### 调优总结

年轻代大小选择

响应时间优先的应用：尽可能设大，直到接近系统的最低响应时间限制（根据实际情况选择）。在此种情况下，年轻代收集发生的频率也是最小的。同时，减少到达年老代的对象。

吞吐量优先的应用：尽可能的设置大，可能到达Gbit的程度。因为对响应时间没有要求，垃圾收集可以并行进行，一般适合8CPU以上的应用。

年老代大小选择

响应时间优先的应用：年老代使用并发收集器，所以其大小需要小心设置，一般要考虑并发会话率和会话持续时间等一些参数。如果堆设置小了，可以会造成内存碎片、高回收频率以及应用暂停而使用传统的标记清除方式；如果堆大了，则需要较长的收集时间。最优化的方案，一般需要参考以下数据获得：

并发垃圾收集信息

持久代并发收集次数

传统GC信息

花在年轻代和年老代回收上的时间比例

减少年轻代和年老代花费的时间，一般会提高应用的效率

吞吐量优先的应用：一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象。

较小堆引起的碎片问题  
因为年老代的并发收集器使用标记、清除算法，所以不会对堆进行压缩。当收集器回收时，他会把相邻的空间进行合并，这样可以分配给较大的对象。但是，当堆空间较小时，运行一段时间以后，就会出现“碎片”，如果并发收集器找不到足够的空间，那么并发收集器将会停止，然后使用传统的标记、清除方式进行回收。如果出现“碎片”，可能需要进行如下配置：

-XX:+UseCMSCompactAtFullCollection：使用并发收集器时，开启对年老代的压缩。

-XX:CMSFullGCsBeforeCompaction=0：上面配置开启的情况下，这里设置多少次Full GC后，对年老代进行压缩

## Java内存泄露的问题调查定位

jstack

观察jvm中当前所有线程的运行情况和线程当前状态。

jstat

jstat利用JVM内建的指令对Java应用程序的资源和性能进行实时的命令行的监控，

包括了对进程的classloader，compiler，gc情况；

jmap

监视进程运行中的jvm物理内存的占用情况，该进程内存内，所有对象的情况，

例如产生了哪些对象，对象数量；

jinfo

观察进程运行环境参数，包括Java System属性和JVM命令行参数