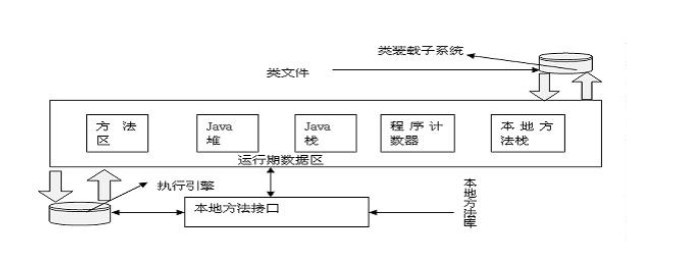
**JVM内存管理-------Java语言的内存管理概述**

**1、内存的划分**

**pc寄存器、Java虚拟机栈、Java堆、方法区、运行时常量池以及本地方法栈**

**运行时常量池由本地方法栈分配出来，所以图中没有运行时常量池**



**JVM结构图**

**解释：**

**1、矩形框中运行期数据这部分就是内存**

**2、PC寄存器（线程独有）：用来支持多线程，线程的阻塞、恢复、挂起等一系列的 操作，每一个线程都有一个PC寄存器，也就是PC寄存器是线程独有的。**

**3、Java虚拟机栈（线程独有）**

**创建线程的同时创建的，存储栈帧，也是线程独有的**

**栈帧：一个方法运行时，临时数据的存储区域，包含了数据和部分的过程结果，与此同时又肩负着处理方法的返回值、动态链接以及异常分派的任务。**

**栈帧随着方法的创建而创建，随着方法的结束而销毁，方法抛出异常也算方法结束。**

**在每一个栈帧中都有着自己的局部变量表以及操作数栈以及对当前类的运行时常量池的引用**

**局部变量表：一个方法局部变量的列表，在编译时就写入了class文件当中，可以将他理解为一个对象数组，里面按照索引0到length-1分别对应于每一个局部变量，特别的，如果是实例方法（static 实例方法在第一次创建对象时不分配入口地址，创建对象后分配入口地址并共享）的局部变量表，第0个局部变量会是一个指向当前实例的应用，也就是this关键字，其余变量则从索引1开始**

**操作数栈：是一个后进先出的栈，他的长度在编译时期就写入了class文件当中，是固定的。他的作用是提供字节码指令操作变量计算的空间，比如简单的，对于int a = 9 需要将9压入操作数栈，在将9赋值给a这个变量。**

**4、Java堆（全局共享）**

**随着java虚拟机的启动而创建，储存着所有对象实例以及数组对象，而且内置了“自动内存管理系统”也就是常说的垃圾收集器（GC）**

**Java堆中的内存释放是不受开发人员控制的，Java由java虚拟机一手操办**

**5、方法区（全局共享）**

**堆的一部分，主要存储运行时常量池、字段信息、方法信息、构造方法、普通函数的字节码内容以及一些特殊方法（java虚拟机规范不强制要求实现内存管理）**

**6、本地方法栈（线程独有）**

**本地方法栈是一个传统的栈，用来支持native方法扥执行，如果java虚拟机是使用的其他语言实现指令集解释器的时候，也会用到本地方法栈，需要本地方法栈的话它是随每一个线程启动而创建的。**

**内存管理**

**五个内存区域，大致可分为两部分一部分是（全局共享） 一部分是（线程独有）**

**对于线程独有这部分内存，随着线程的的启动而创建，线程销毁随之释放，这一部分内存，不需要垃圾收集器的管理，而是java虚拟机主动管理。**

**每当一个线程被创建的时候，java虚拟机就会为其分配相应的pc寄存器和java虚拟机栈，如果需要的话，还会有本地方法栈，当一个线程被销毁的时候，java虚拟机也会java这个线程所占有的内存全部释放。**

**全局共享的这部分内存（简称堆） 内存分配主要是由程序员显示的使用new关键字来触发，new出来的这部分内存在哪分配，如何分配，则由java虚拟机来决定，这部分内存的释放，由自动内存管理系统（简称GC）来管理**

**二、obj=null**

**1、为何不将对象赋值为空，会造成内存泄漏**

**import java.util.Arrays;
public class Stack {
private static final int INIT\_SIZE = 10;
private Object[] datas;
private int size;
public Stack() {
super();
datas = new Object[INIT\_SIZE];
}
public void push(Object data){
if (size == datas.length) {
extend();
}
datas[size++] = data;
}
public Object pop(){
if (size == 0) {
throw new IndexOutOfBoundsException("size is zero");
}
return datas[--size]; // datas[size] 引用并没有设置为空，数组只会越来越大
}
private void extend(){
datas = Arrays.copyOf(datas, 2 \* size + 1);
}
}**

**java ArrayList中remove的实现**

**private void fastRemove(int index) {**

**modCount++;**

**int numMoved = size - index - 1;**

**if (numMoved > 0)**

**System.arraycopy(elementData, index+1, elementData, index,**

**numMoved);**

**elementData[--size] = null; // clear to let GC do its work**

**}**

**三、GC简介**

**1、内存溢出与内存泄漏问题**

**自动GC的策略解决的问题**

**（1）哪些对象可以被回收**

**（2）何时回收这些对象**

**（3）采用什么样的方式回收**

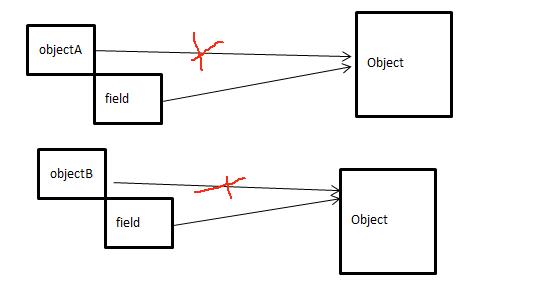
**2、GC策略采用何种算法**

**（1）引用计算算法（循环引用的对象无法回收）**

**以下这个就是循环引用**

**public void run() {
Object objectA = new Object();
Object objectB = new Object();//1
objectA.field = objectB;
objectB.field = objectA;//2
//to do something
objectA = null;
objectB = null;//3
}**

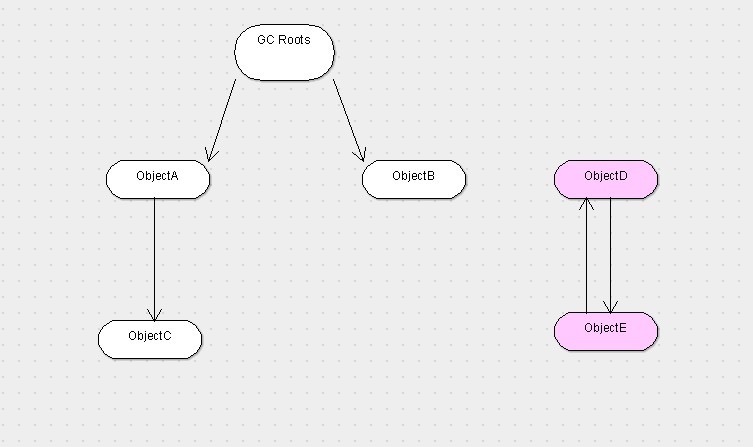
**如下，最终会有两个不会被回收**



当第1个地方的语句执行完以后，两个对象的引用计数全部为1。当第2个地方的语句执行完以后，两个对象的引用计数就全部变成了2。当第3个地方的语句执行完以后，也就是将二者全部归为空值以后，二者的引用计数仍然为1。根据引用计数算法的回收规则，引用计数没有归0的时候是不会被回收的。

（2）根搜索算法

**当任何一个根对象到某一个对象均不可达时，则认为这个对象是可以被回收的。**



**GC roots到D和E不可达，最终会被当做GC的对象**

**在java中可以被当做 GC roots 的对象有：**

**1、虚拟机栈中的引用的对象**

**2、方法区中的类静态属性引用的对象**

**3、方法区中的常量引用的对象**

**4、本地方法中JNI的引用的对象**

   第一和第四种都是指的方法的本地变量表，第二种表达的意思比较清晰，第三种主要指的是声明为final的常量值。

**垃圾搜集的算法：**

**标记-清除算法、复制算法、标记-整理算法**。

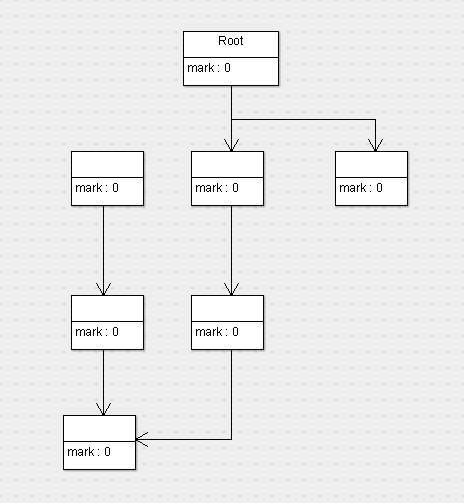
四、GC算法

1、标记/清除算法

堆中有效内存空间被耗尽，停止整个程序，然后进行两项工作，第一项 标记，第二项 清除

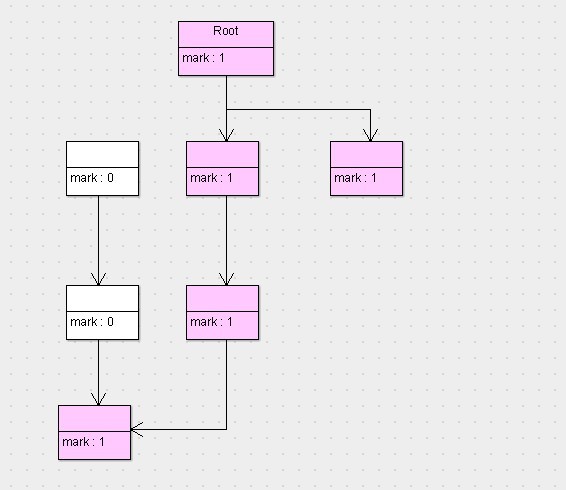
标记 遍历所有GC Roots 将所有GC Roots可达的对象标记为存活的对象，然后将所有可达的对象标记为存活的对象

清除 清除的过程将遍历堆中所有的对象，将没有标记的对象全部清除掉

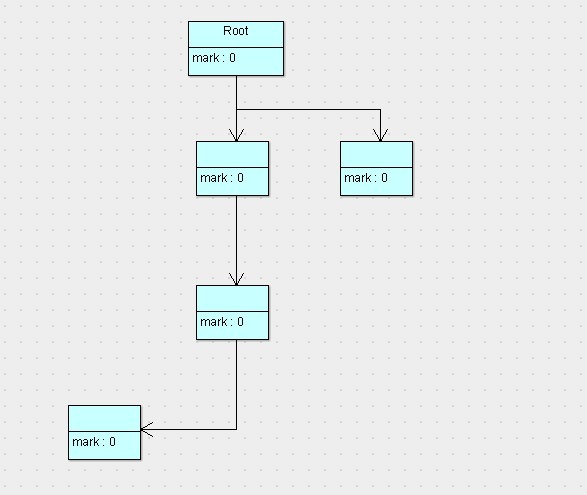


**未标记前，标志位是0**

**按照根搜索算法标记完后**



清除后的状态



**再唤醒停止的程序线程，让程序继续运行**

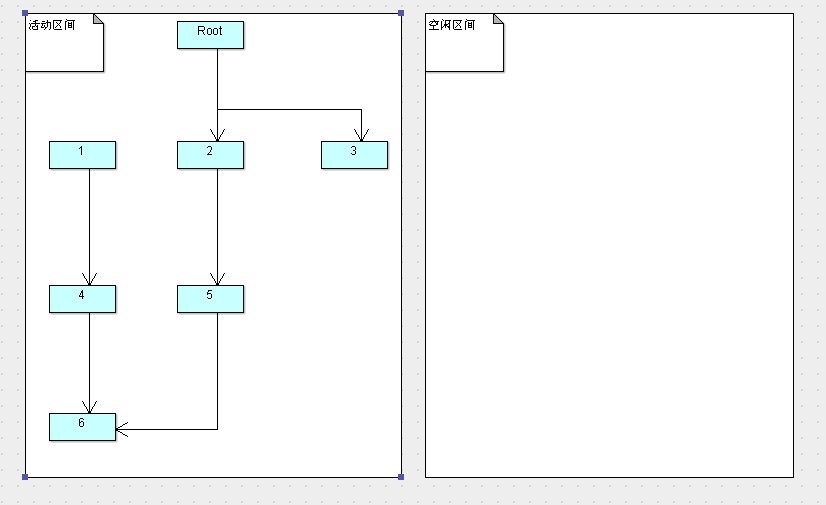
**五、复制算法与标记整理算法**

**1、复制算法**

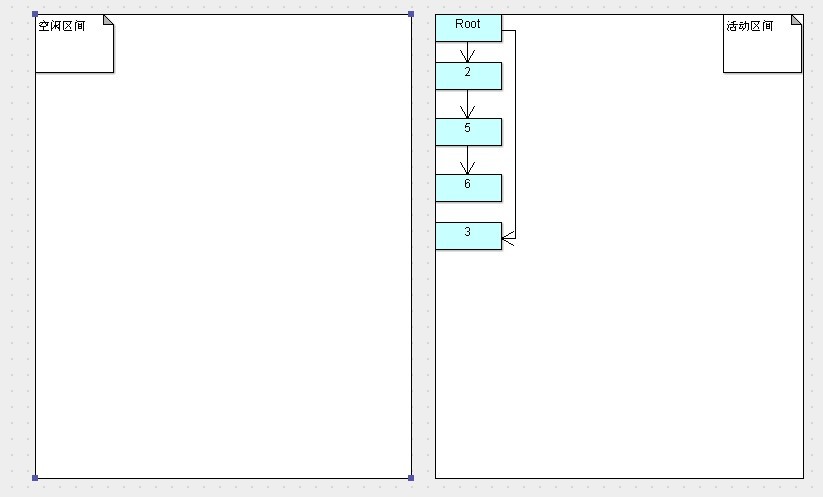
**复制算法将内存划分为两个区间，在任意时间点，多有动态分配的对象都只能分配在其中一个区间（成为活动区间），而另外一个区间（成为空闲区间）则是空闲的**

**有效内存耗尽时，JVM将暂停程序运行，开启复制算法GC线程，接下来GC想成会将活动区间内的存活对象，全部复制到空闲区间，且严格按照内存地址依次排列，与此同时，GC线程将更新存活对象内存引用地址指向新的内存地址**

**此时 空闲区间已经与活动区间交换，而垃圾对象全部留在了原来的活动区间，也就是现在的空闲区间，事实上在活动区间转换为空闲区间的时候，垃圾对象已经被一次性全部回收。**



**复制之后**



**优点：解决了标记清除算法内存布局很乱的缺点**

**缺点：浪费了一般的内存**

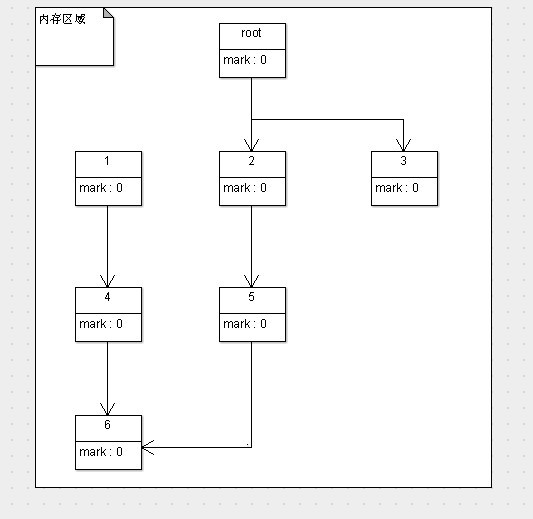
**六、标记整理算法**

**1、标记**

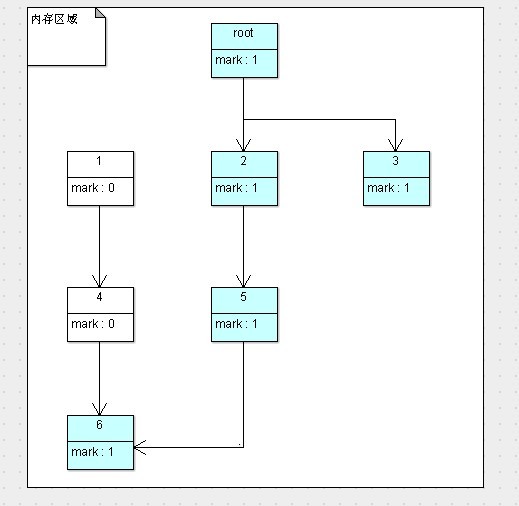
**遍历GC Roots ，将存活的对象标记**

**整理**

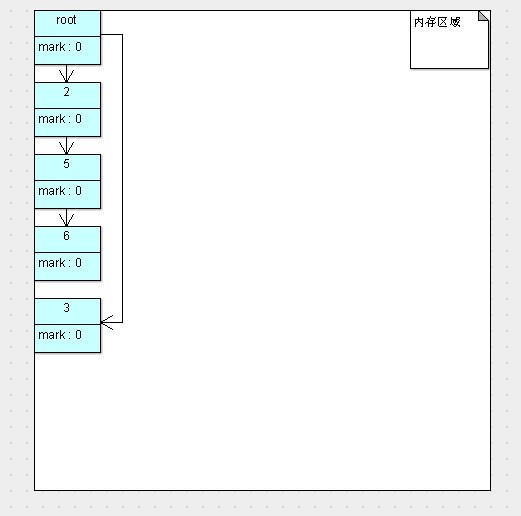
**移动所有存活的对象，按照内存地址一次排列，然后将末端地址以后的内存全部回收。**



标记



整理



**总结：**

**1、三个算法都是基于根搜索算法去判断一个对象是否应该回收，而搜索算法的理论依据就是 语法中变量作用域**

**2、GC线程开启，都要暂停应用程序**

**3、在时间复杂度上 复制算法>标记整理算法>标记清除算法**

**内存的整齐度 复制算法=标记整理算法>标记清除算法**

**内存的利用率 标记整理算法=标记清除算法>复制算法**

**七、分代搜索算法**

**内存中按照声明周期长短进行分类可分为三种：**

**1、夭折对象 朝生夕灭**

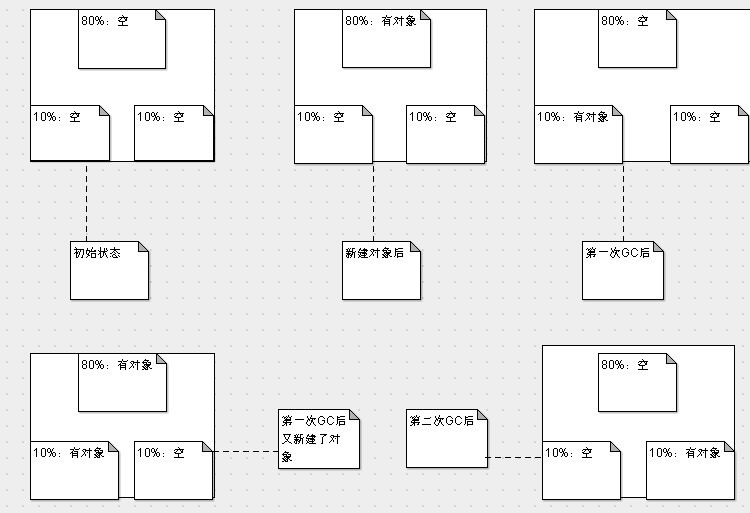
**2、老不死对象 存活时间长，但仍旧会消亡**

**3、不灭对象**

**夭折对象**

**存活时间短，复制算法要求对象存活率不太高，夭折对象最适合使用复制算法**

**一般的使用10%的内存区域作为空闲和活动区间，另外的80%内存用来给新建对象分配内存的，一旦发生GC，将10%的活动区间与另外80%的存活的对象转移到10%的空闲区间，接下来将90%的内存全部释放，以此类推。**



**两次GC**

       不过有两点LZ需要提一下，**第一点是使用这样的方式，我们只浪费了10%的内存，这个是可以接受的，因为我们换来了内存的整齐排列与GC速度。第二点是，这个策略的前提是，每次存活的对象占用的内存不能超过这10%的大小，一旦超过，多出的对象将无法复制**。

         为了解决上面的意外情况，也就是存活对象占用的内存太大时的情况，高手们将JAVA堆分成两部分来处理，上述三个区域则是第一部分，**称为新生代或者年轻代**。而余下的一部分，专门存放老不死对象的则称为**年老代**。

         是不是很贴切的名字呢？下面我们看看老不死对象的处理方式。

         **老不死对象**：这一类对象存活率非常高，因为它们大多是从新生代转过来的。就像人一样，活的年月久了，就变成老不死了。

         通常情况下，以下两种情况发生的时候，对象会从新生代区域转到年老带区域。

         **1、**在新生代里的每一个对象，都会有一个**年龄**，当这些对象的年龄到达一定程度时（年龄就是熬过的GC次数，每次GC如果对象存活下来，则年龄加1），则会被转到年老代，而这个转入年老代的年龄值，一般在JVM中是可以设置的。

         **2、**在新生代存活对象占用的内存超过10%时，则多余的对象会放入年老代。这种时候，年老代就是新生代的“备用仓库”。

         针对老不死对象的特性，显然不再适合使用复制算法，因为它的存活率太高，而且不要忘了，如果年老代再使用复制算法，它可是没有备用仓库的。因此一般针对**老不死对象只能采用标记/整理或者标记/清除算法**。

**八、方法区的对象回收**

以上解决了GC的大部分问题，因为JAVA堆是GC的主要对象，不灭对象存于方法区，方法区也被成为**永久代，**和GC年老代采用相似的方法，由于没有备用仓库，二者只能使用标记/清除和标记/整理算法

**九、回收的时机**

GC按照回收区域又分了两种类型，一种是普通GC,一种是全局GC

普通GC：只针对新生代区域的GC

全局GC：针对老年代的GC，偶尔伴随新生代GC及永久代的GC

**由于年老代与永久代相对来说GC效果不好，而且二者的内存使用增长速度也慢，因此，一般情况下，需要经过好几次普通GC，才会触发一次全局GC**

**垃圾搜集器**

**我们平时使用的JDK中，默认的JVM是hotspot**

一、垃圾搜集器的分类

1、串行搜集器（serial collector）：只有一条GC线程，运行的时候需要暂停用户程序。

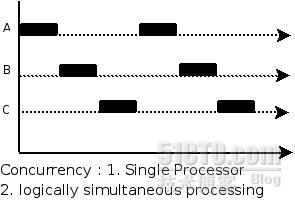
2、并行搜集器（parallel collector）：有多条GC线程，也需要暂停用户程序

3、并发搜集器（concurrent collector）：有一条或多条GC线程，需要在部分阶段暂停用户程序，部分阶段与用户程序并发执行。

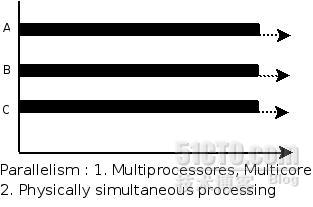
二、并发（concurrent）与并行（parallel）

并发:两个任务相互独立的运行，并且A任务开始后，B任务在A任务之前开始了

（一个cpu,任一时刻只有一个程序在处理机上运行，并发程序走走停停）



并行：同一时刻，不同程序在不同的cpu运行。



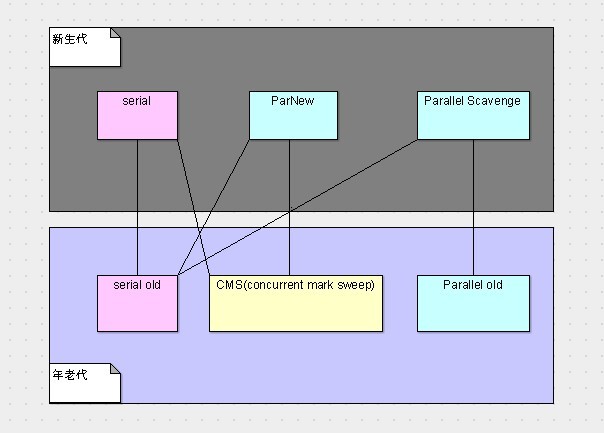
**实现**

 **串行搜集器的实现：**

**serial（用于新生代，采用复制算法）、serial old（用于年老代，采用标记/整理算法）**

**并行搜集器的实现：ParNew（用于新生代，采用复制算法）、Parallel Scavenge（用于新生代，采用复制算法）、Parallel old（用于年老代，采用标记/整理算法）**

**并发搜集器的实现：concurrent mark sweep[CMS]（用于年老代，采用标记/清除算法）**



**连线表示可以配合**

**垃圾搜集器精解**

**一、JVM的两种模式，client模式和server模式**

平时开发使用的是client模式，也可以使用命令行参数-server强制开启server模式，server模式下JVM做了很多优化，server模式程序启动慢，由于优化，长时间运行，速度会越来越快，相反client启动快，不适合长时间运行。

二、详解

1、Serial Garbage Collector

算法：采用复制算法

内存区域：针对新生代设计

执行方式：单线程、串行

执行过程：当新生代内存不够用时，先暂停全部用户程序，然后开启一条GC线程对垃圾进行回收，这一过程有一些对象提升到年老代

特点：由于是单线程，整个GC阶段都要暂停用户程序，因此会造成应用程序停顿时间较长，对于小规模的程序来说，非常适合。

适用：平时的开发与调试程序，以及桌面应用交互程序

开启参数： -XX：+UseSerialGC（client模式默认值）

2、Serial Old Garbage Collector

**针对年老代设计，因此采用标记/整理算法**

3、ParNewGarbage Colector

算法：采用复制算法

内存区域：针对新生代设计

执行方式：多线程、并行

执行过程：当新生代内存不够用时，先暂停全部用户程序，然后开启若干条GC线程对垃圾进行回收，这一过程有一些对象提升到年老代

特点：并行，至少需要多于一个处理器，有几个处理器就会开几个线程（不过线程数是可以使用参数-XX:ParallelGCThreads=<N>控制的）适合于多核多处理器的系统。

尽管整个GC阶段还是要暂停用户程序，但多线程并行处理并不会造成太长的停顿时间。因此就吞吐量来说，ParNew要大于serial，在处理器越多的时候，效果越明显。但是这并非绝对，对于单个处理器来说，由于并行执行的开销（比如同步），ParNew的性能将会低于serial搜集器。不仅是单个处理器的时候，如果在容量较小的堆上，甚至在两个处理器的情况下，ParNew的性能都并非一定可以高过serial。

         适用场景：在中到大型的堆上，且系统处理器至少多于一个的情况

         开启参数：-XX:+UseParNewGC

4、**Parallel Scavenge ([ˈskævɪndʒ] 清除)Garbage Collector**

        这个搜集器与ParNew几乎一模一样，都是针对新生代设计，采用复制算法的并行搜集器。它与ParNew最大的不同就是可设置的参数不一样，它可以让我们更精确的控制GC停顿时间以及吞吐量。

        parallel scavenge搜集器提供参数主要包括控制最大的停顿时间（使用-XX:MaxGCPauseMillis=<N>），以及控制吞吐量（使用-XX:GCTimeRatio=<N>）。由此可以看出，parallel scavenge就是为了提供吞吐量控制的搜集器。

        不过千万不要以为把最大停顿时间调的越小越好，或者吞吐量越大越好，在使用parallel scavenge搜集器时，主要有三个性能指标，最大停顿时间、吞吐量以及新生代区域的最小值。

        parallel scavenge搜集器具有相应的调节策略，它将会优先满足最大停顿时间的目标，次之是吞吐量，最后才是新生代区域的最小值。

        因此，如果将最大停顿时间调的过小，将会牺牲整体的吞吐量以及新生代大小来满足你的私欲。手心手背都是肉，我们最好还是不要这么干。不过parallel scavenge有一个参数可以让parallel scavenge搜集器全权接手内存区域大小的调节，这其中还包括了晋升为年老代（可使用-XX:MaxTenuringThreshold=n调节）的年龄，也就是使用-XX:UseAdaptiveSizePolicy打开内存区域大小自适应策略。

        parallel scavenge搜集器可使用参数-XX:+UseParallelGC开启，同时它也是server模式下默认的新生代搜集器。

**Parallel Old Garbage Collector**

         Parallel Old与ParNew或者Parallel Scavenge的关系就好似serial与serial old一样，相互之间的区别并不大，只不过parallel old是针对年老代设计的并行搜集器而已，因此它采用标记/整理算法。

         Parallel Old搜集器还有一个重要的意义就是，它是除了serial old以外唯一一个可以与parallel scavenge搭配工作的年老代搜集器，因此为了避免serial old影响parallel scavenge可控制吞吐量的名声，parallel old就作为了parallel scavenge真正意义上的搭档。

         它可以使用参数-XX:-UseParallelOldGC开启，不过在JDK6以后，它也是在开启parallel scavenge之后默认的年老代搜集器。

**Concurrent Mark Sweep Garbage Collector**

         concurrent mark sweep（以下简称CMS）搜集器是唯一一个真正意义上实现了应用程序与GC线程一起工作（一起是针对客户而言，而并不一定是真正的一起，有可能是快速交替）的搜集器。

         CMS是针对年老代设计的搜集器，并采用标记/清除算法，它也是唯一一个在年老代采用标记/清除算法的搜集器。

         采用标记/清除算法是因为它特殊的处理方式造成的，它的处理分为四个阶段。

         1、初始标记：需要暂停应用程序，快速标记存活对象。

         2、并发标记：恢复应用程序，并发跟踪GC Roots。

         3、重新标记：需要暂停应用程序，重新标记跟踪遗漏的对象。

         4、并发清除：恢复应用程序，并发清除未标记的垃圾对象。

         它比原来的标记/清除算法复杂了点，主要表现在并发标记和并发清除这两个阶段，而这两个阶段也是整个GC阶段中耗时最长的阶段，不过由于这两个阶段皆是与应用程序并发执行的，因此CMS搜集器造成的停顿时间是非常短暂的。这点还是比较好理解的。

1、由于GC线程与应用程序并发执行时会抢占CPU资源，因此会造成整体的吞吐量下降。也就是说，从吞吐量的指标上来说，CMS搜集器是要弱于parallel scavenge搜集器的。LZ这里从oracle官网上摘录下一段关于CMS的描述，里面提到CMS性能与CPU个数的关系。

         Since at least one processor is utilized for garbage collection during the concurrent phases, the concurrent collector does not normally provide any benefit on a uniprocessor (single-core) machine. However, there is a separate mode available that can achieve low pauses on systems with only one or two processors; see incremental mode below for details.

         LZ的英文很一般（四级都没过，惭愧，0.0），不过在借助工具的情况下也能大致翻译出来这段话的意思，如下。

         中文大意：由于在并发阶段垃圾搜集至少使用了一个处理器，因此在单处理器的情况下使用并发搜集器，将得不到任何好处。不过，在单个或两个处理器的系统上，有一种独立的方式可以有效的达到低停顿的目的，详情见下方的增量模式（incremental mode）。

         很明显，oracle的文档指出，在单处理器的情况下，并发搜集器会因为抢占处理器，而造成性能降低。最后给出了一种增量模式的处理方式，不过在《深入理解JAVA虚拟机》一书中指出，增量模式已经被定义为不推荐使用。由于LZ摘录的这段官方介绍是基于JDK5.0的介绍，而《深入理解JAVA虚拟机》一书中则是指的JDK6.0的版本，因此LZ暂且猜测，增量模式是在JDK6.0发布的时候被废弃了，不过这个废弃的时间或者说版本其实已经不重要了。

         2、标记/清除很大的一个缺点，那就是内存碎片的存在。因此JVM提供了-XX:+UseCMSCompactAtFullCollection参数用于在全局GC（full GC）后进行一次碎片整理的工作，由于每次全局GC后都进行碎片整理会较大的影响停顿时间，JVM又提供了参数-XX:CMSFullGCsBeforeCompaction去控制在几次全局GC后会进行碎片整理。

       3、CMS最后一个缺点涉及到一个术语---并发模式失败（Concurrent Mode Failure）。对于这个术语，官方是这样解释的。

         if the concurrent collector is unable to finish reclaiming the unreachable objects before the tenured generation fills up, or if an allocation cannot be satisfied with the available free space blocks in the tenured generation, then the application is paused and the collection is completed with all the application threads stopped.The inability to complete a collection concurrently is referred to as concurrent mode failure and indicates the need to adjust the concurrent collector parameters.

         中文大意：如果并发搜集器不能在年老代填满之前完成不可达（unreachable）对象的回收，或者年老代中有效的空闲内存空间不能满足某一个内存的分配请求，此时应用会被暂停，并在此暂停期间开始垃圾回收，直到回收完成才会恢复应用程序。这种无法并发完成搜集的情况就成为并发模式失败（concurrent mode failure），而且这种情况的发生也意味着我们需要调节并发搜集器的参数了。

         为了尽量避免并发模式失败发生，我们可以调节-XX:CMSInitiatingOccupancyFraction=<N>参数，去控制当年老代的内存占用达到多少的时候（N%），便开启并发搜集器开始回收年老代。

**组合**

**三个典型的组合，其余三种组合一般不常用**

**1、serial & serial old**

         这个组合是我们最常见的组合之一，也是client模式下的默认垃圾搜集器组合，也可以使用参数-XX:+UseSerialGC强制开启。

         由于它实现相对简单，没有线程相关的额外开销（主要指线程切换与同步），因此非常适合运行于客户端PC的小型应用程序，或者桌面应用程序（比如swing编写的用户界面程序），以及我们平时的开发、调试、测试等。

         上面三种情况都有共同的特点。

         1、由于都是在PC上运行，因此配置一般不会太高，或者说处理器个数不会太多。

         2、上面几种情况的应用程序都不会运行太久。

         3、规模不会太大，也就是说，堆相对较小，搜集起来也比较快，停顿时间会比较短。

**2、Parallel Scavenge & Parallel Old**

         这个组合我们并不常见，毕竟它不会出现在我们平时的开发当中，但是它却是很多对吞吐量（throughout）要求较高或者对停顿时间（pause time）要求不高的应用程序的首选，并且这个组合是server模式下的默认组合（JDK6或JDK6之后）。当然，它也可以使用-XX:+UseParallelGC参数强制开启。

         该组合无论是新生代还是年老代都采用并行搜集，因此停顿时间较短，系统的整体吞吐量较高。它适用于一些需要长期运行且对吞吐量有一定要求的后台程序。

         这些运行于后台的程序都有以下特点。

         1、系统配置较高，通常情况下至少四核（以目前的硬件水平为准）。

         2、对吞吐量要求较高，或需要达到一定的量。

         3、应用程序运行时间较长。

         4、应用程序规模较大，一般是中到大型的堆。

**3、ParNew & CMS（Serial Old作为替补）**

         这个组合与上面的并行组合一样，在平时的开发当中都不常见，而它则是对相应时间（response time）要求较高的应用程序的首选。该组合需要使用参数-XX:+UseConcMarkSweepGC开启。

         该组合在新生代采用并行搜集器，因此新生代的GC速度会非常快，停顿时间很短。而年老代的GC采用并发搜集，大部分垃圾搜集的时间里，GC线程都是与应用程序并发执行的，因此造成的停顿时间依然很短。它适用于一些需要长期运行且对相应时间有一定要求的后台程序。

         这些运行于后台的程序的特点与并行模式下的后台程序十分类似，不同的是第二点，采用ParNew & CMS组合的后台应用程序，一般都对相应时间有一定要求，最典型的就是我们的WEB应用程序。

**垃圾搜集器开启的参数**

          UseSerialGC：开启此参数使用serial & serial old搜集器（client模式默认值）。

          UseParNewGC：开启此参数使用ParNew & serial old搜集器（不推荐）。

          UseConcMarkSweepGC：开启此参数使用ParNew & CMS（serial old为替补）搜集器。

          UseParallelGC：开启此参数使用parallel scavenge & parallel old搜集器（server模式默认值）。

          UseParallelOldGC：开启此参数在年老代使用parallel old搜集器（该参数在JDK1.5之后已无用）。

**JVM各个内存区域大小相关参数**

         Xms：堆的初始值。默认为物理内存的1/64，最大不超1G。

          Xmx：堆的最大值。默认为物理内存的1/4，最大不超1G。

          Xmn：新生代的大小。

          Xss：线程栈大小。

          PermSize：永久代初始大小。默认为物理内存的1/64，最大不超1G。

          MaxPermSize：永久代最大值。默认为物理内存的1/4，最大不超1G。

          NewRatio：新生代与年老代的比例。比如为3，则新生代占堆的1/4，年老代占3/4。

          SurvivorRatio：新生代中调整eden区与survivor区的比例，默认为8，即eden区为80%的大小，两个survivor分别为10%的大小。（备注：这个参数设定是讲解复制算法那一章中，解决复制算法内存减半的办法。eden区即是复制算法一章中80%的那部分，而survivor区则是两个10%的那部分。）