## 1：jvm参数设置

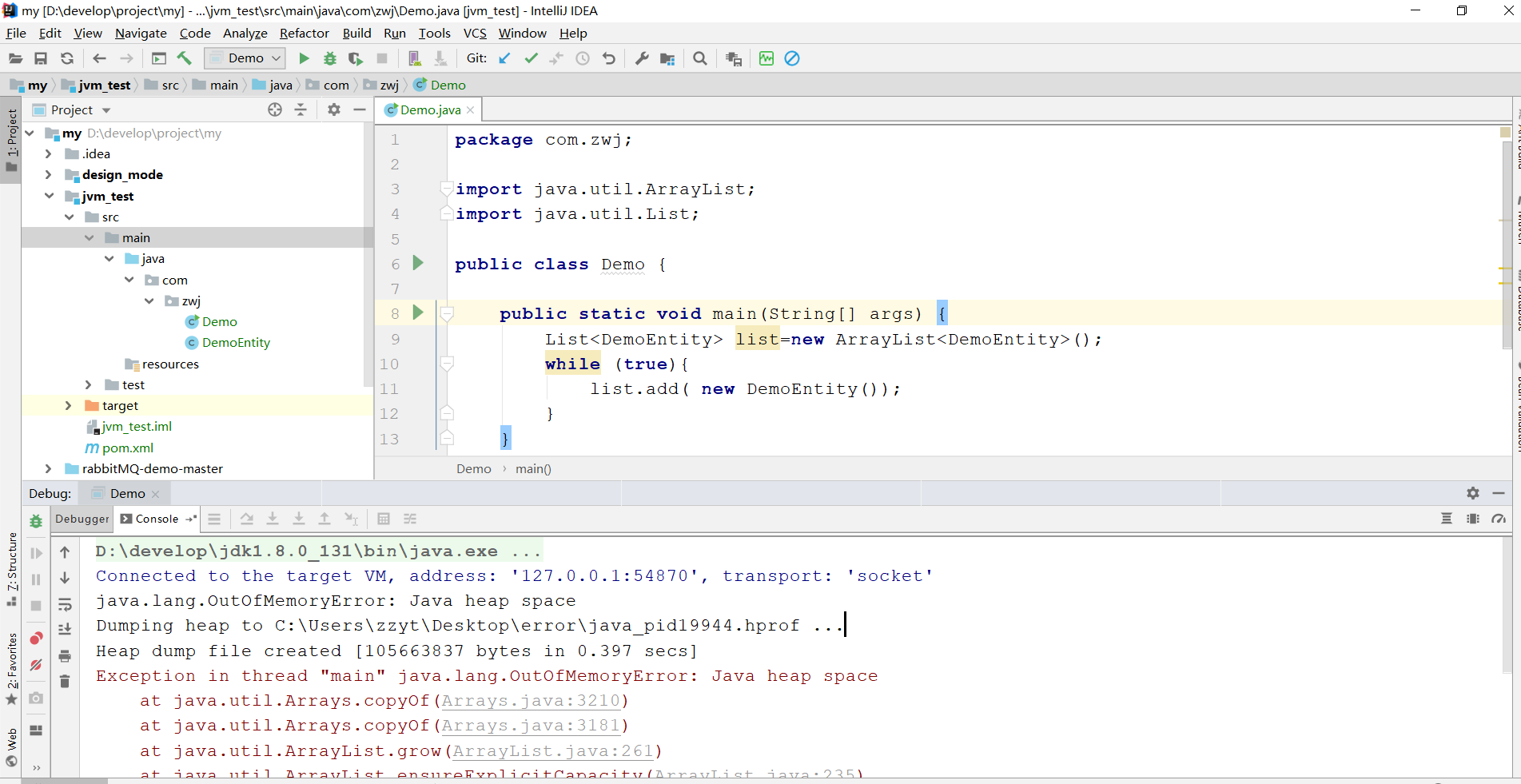
### jvm快照

内存溢出大小：-Xms60m -Xmx60m (表示初始化内存为60MB，可以使用的最大内存为62MB)

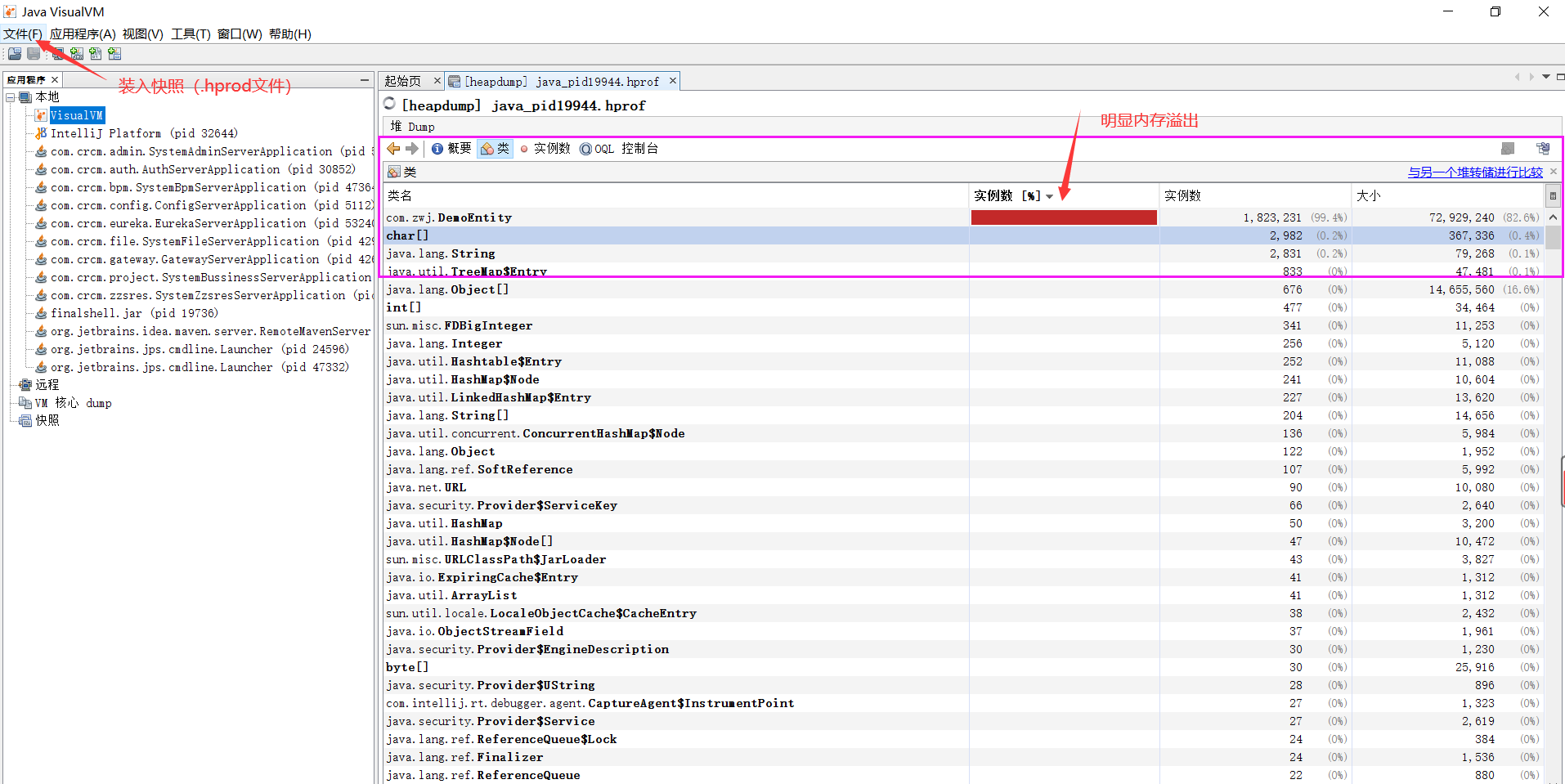
内存溢出快照：-XX:+HeapDumpOnOutOfMemoryError

内存溢出快照保存地址：-XX:HeapDumpPath=C:\Users\zzyt\Desktop\error

#### （1）分析快照

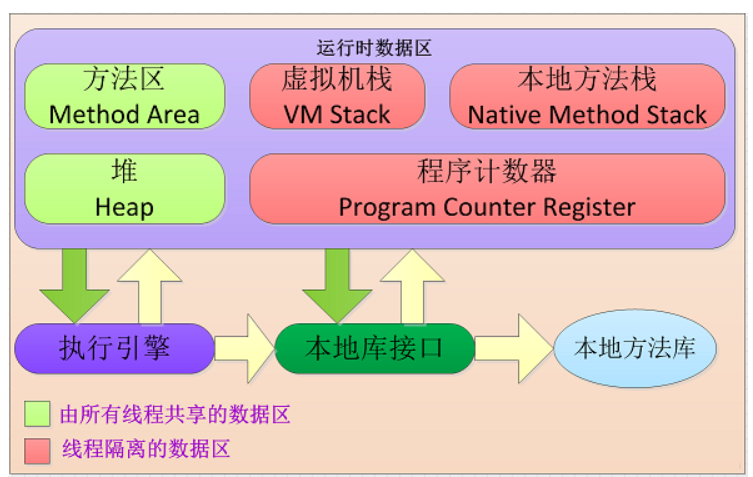
代码如图：

jvisualVm分析图：



## java虚拟机内存

内存管理模型图：



### 程序计数器

占用内存区域较小，可以看出当前线程执行的行号（通俗讲就是记录当前线程执行到哪一行的标记）。

在虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来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、线程恢复等基础功能都需要依赖这个计数器来完成。

由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，在任何一个确定的时刻，一个处理器都只会执行一条线程中的指令。因此，为了线程切换后能恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器，各条线程之间计数器互不影响，独立存储，我们称这类内存区域为“线程私有”的内存。

      如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Native方法，这个计数器值则为空（Undefined）。此内存区域是唯一 一个在Java虚拟机规范中没有规定OutOfMemoryError情况的区域。

goto(跳转到某行)关键字，java保留关键字。

### java虚拟机栈

与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。虚拟机栈描述的是Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧（Stack Frame）用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每一个方法从调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

经常有人把Java内存区分为堆内存（Heap）和栈内存（Stack），其中所指的“堆”就是Java堆，而所指的“栈”就是现在所讲的虚拟机栈，或者说是虚拟机栈中局部变量表部分。

局部变量表存放了编译期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型，它不等同于对象本身，可能是一个指向对象起始地址的引用指针，也可能是指向一个代表对象的句柄或其他与此对象相关位置）和returnAddress类型（指向了一条字节码指令的地址）。

其中64为长度的long和double类型的数据会占用2个局部变量空间（Slot），其余的数据类型只占用1个。局部变量表所需的内存空间在编译期间完成分配，当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。

在Java虚拟机规范中，对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常(例如：无限递归方法)；如果虚拟机栈可以动态扩展（当前大部分的Java虚拟机都可动态扩展，只不过Java虚拟机规范中也允许固定长度的虚拟机栈），如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

### 本地方法栈

虚拟机栈为虚拟机执行java方法服务，本地方法栈为虚拟机执行native方法服务。

本地方法栈（Native Method Stack）与虚拟机栈所发挥的作用是非常相似的，它们之间的区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的Native方法服务。在虚拟机规范中对本地方法栈中方法使用的语言、使用方式与数据结构并没有强制规定。HotSpot虚拟机直接把本地方法栈和虚拟机栈合二为一。与虚拟机栈一样，本地方法栈区域也会抛出StackOverflowError和OutOfMemoryError异常。

　　参数设置：

-Xss 设置每个线程的栈大小。JDK1.5+ 每个线程栈大小为1M，一般来说如果栈不是很深的话，1M是绝对够用的啦。

　　参数含义解析：

以-X开头的参数是和实现有关的，第一个s表示stack，第二个s表示size；

　　注意：

　　　　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堆

Java堆（Java Heap）是Java虚拟机所管理的内存中最大的一块

存放对象实例

垃圾回收器管理的主要区域

新生代、老年代、Eden空间、From Survivor空间、To Survivor空间等

根据Java虚拟机规范的规定，Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。在实现时，既可以实现成固定大小的，也可以是扩展的，不过当前主流的虚拟机都是按照可扩展来实现的（通过-Xmx和-Xms控制）。如果在堆中没有内存完成实例分配，并且堆也无法再扩展时，将会抛出OutOfMemoryError异常。

　　参数设置：

-Xms 设置堆的最小空间大小；通常为操作系统可用内存的1/64大小即可。

-Xmx 设置堆的最大空间大小；通常为操作系统可用内存的1/4大小。

-Xmn 设置新生代大小，是对-XX：newSize、-XX：MaxnewSize两个参数的同时配置，这个参数是在JDK1.4版本以后出现的；通常为Xmx的1/3或1/4。新生代 = Eden + 2个Survivor空间。实际可用空间 = Eden + 1个Survivor，即90%。

-XX：NewSize 设置新生代最小空间大小；

-XX：MaxNewSize 设置新生代最大空间大小；

-XX：NewRatio 新生代与老年代的比例，如-XX：NewRatio=2，则新生代占整个堆空间的1/3，老年代占2/3。

-XX：SurvivorRatio 新生代中 Eden 与 Survivor的比值。默认值为 8 。即Eden占新生代空间的8/10，另外两个Survivor各占1/10。

　　参数含义解析：

以-X开头的参数是和实现有关的，并不是适用于所有的参数；

最开始只有 -Xms的参数，表示‘初始’ memory size，m表示memory，s表示size；

紧接是参数 -Xmx，为了对齐三字符，压缩了其表示形式，采用计算机中约定表示方式：用 x 表示“”大“ （可以联想到衣服的号码大小，S、M、L、XL、XXL），因此 -Xmx中的m应当还是memory。既然有了最大内存的概念，那么一开始的 -Xms所表示的”初始“内存也就有了一个”最小“内存的概念（其实常用的做法中初始内存采用的也就是最小内存）。如果不对齐参数长度的话，其表示应当是-Xmsx。

　　注意：

　　　　开发过程中，通常会将-Xms与-Xmx两个参数的配置相同的值，其目的是为了能够在Java垃圾回收机制清理完堆区后不需要重新分隔计算堆区的大小而浪费资源。

### 方法区

存储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

类的版本、字段、方法、接口

方法区和永久代（HotSpot虚拟机中两者不完全等价）

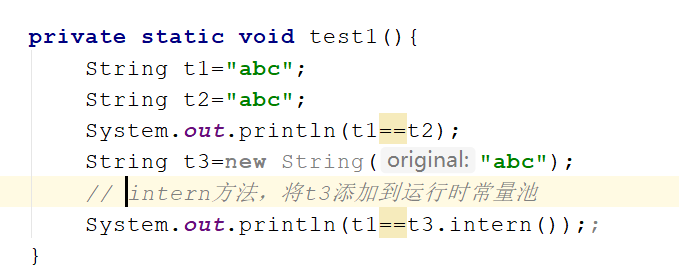
垃圾回收的方法区的行为

异常的定义

根据Java虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

### 运行时常量池

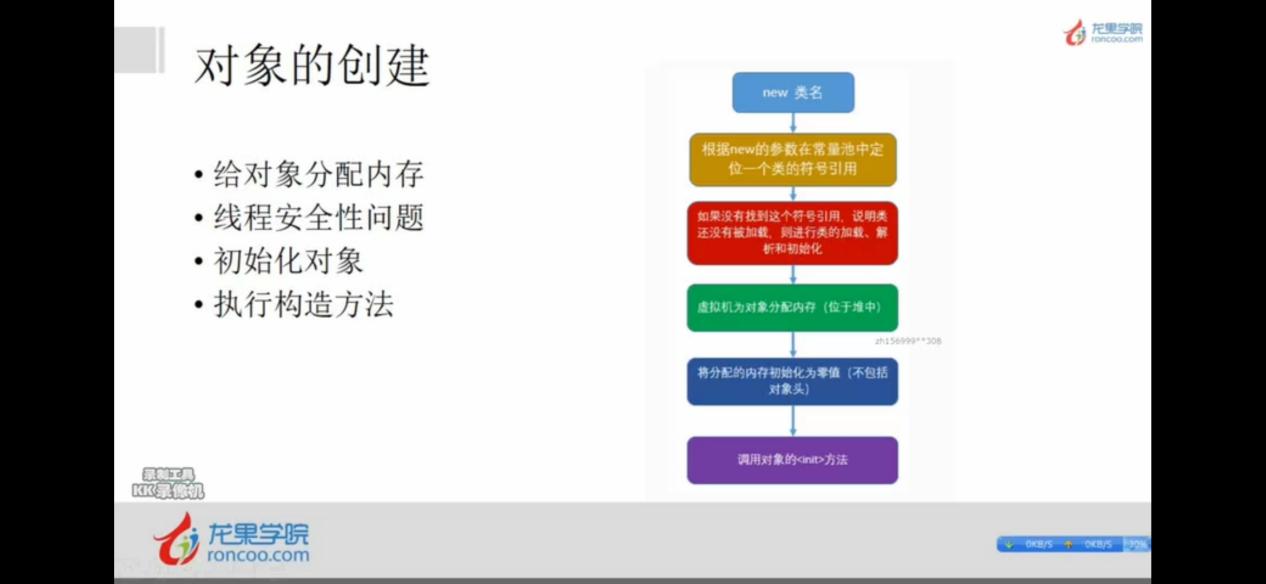
简称：常量池，属于方法区



t1和t2指向常量池同一个常量。t3指向堆内存对象地址。

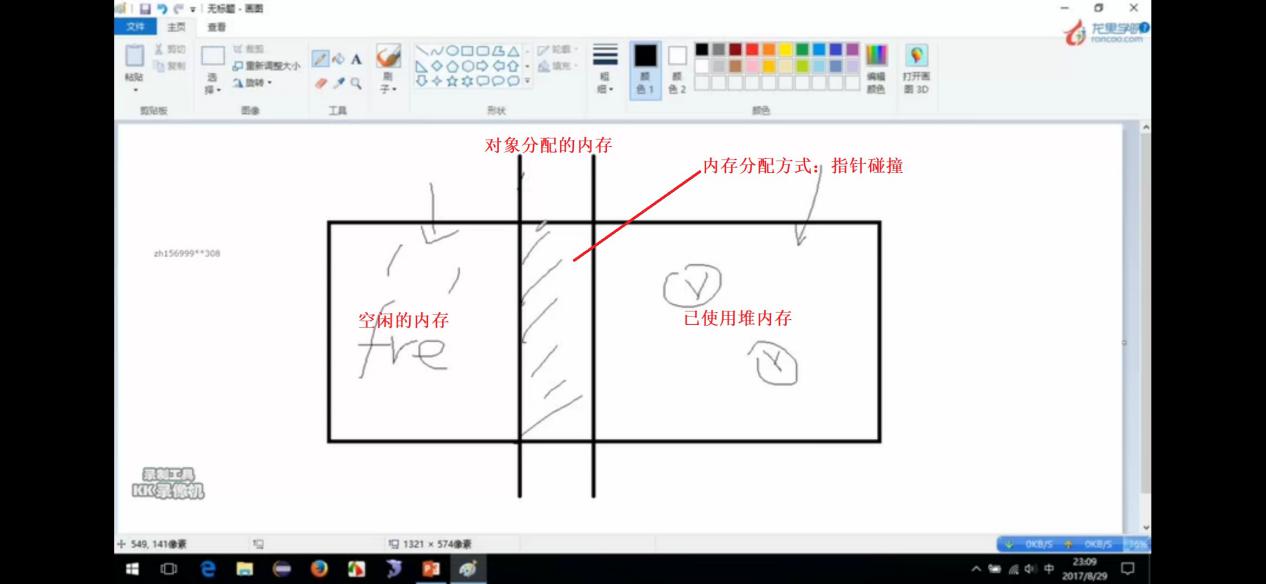
### 直接内存

## 3、对象的创建



### 给对象分配内存

指针碰撞；



空闲列表：

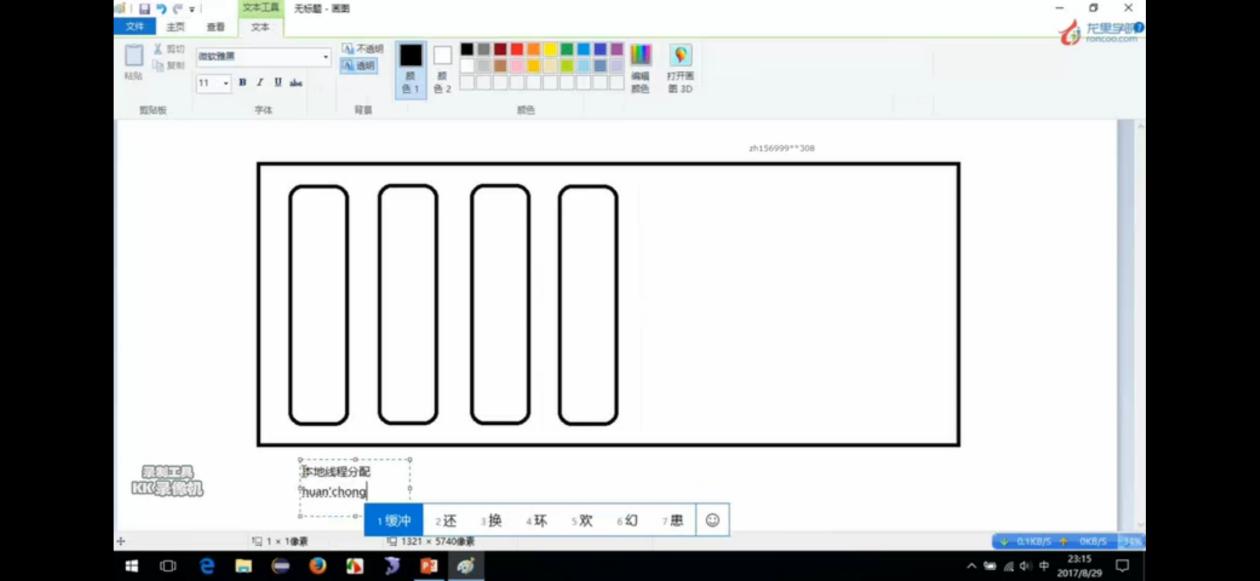
只记录未分配内存

### 2、线程安全性问题：

多线程时指针频繁移动，未分配内存频繁分配。

A.每个线程加锁-执行效率低

B.本地线程分配缓冲（TLAB）：每个线程操作不同的区域



## 4、对象的结构

Header(对象头)

自身运行时数据（Mark Word）：哈希值、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间数



类型指针:对象指向类的类型指针，数组类型还会存储它的长度。

InstanceData(数据实列)

Padding(对象填充)

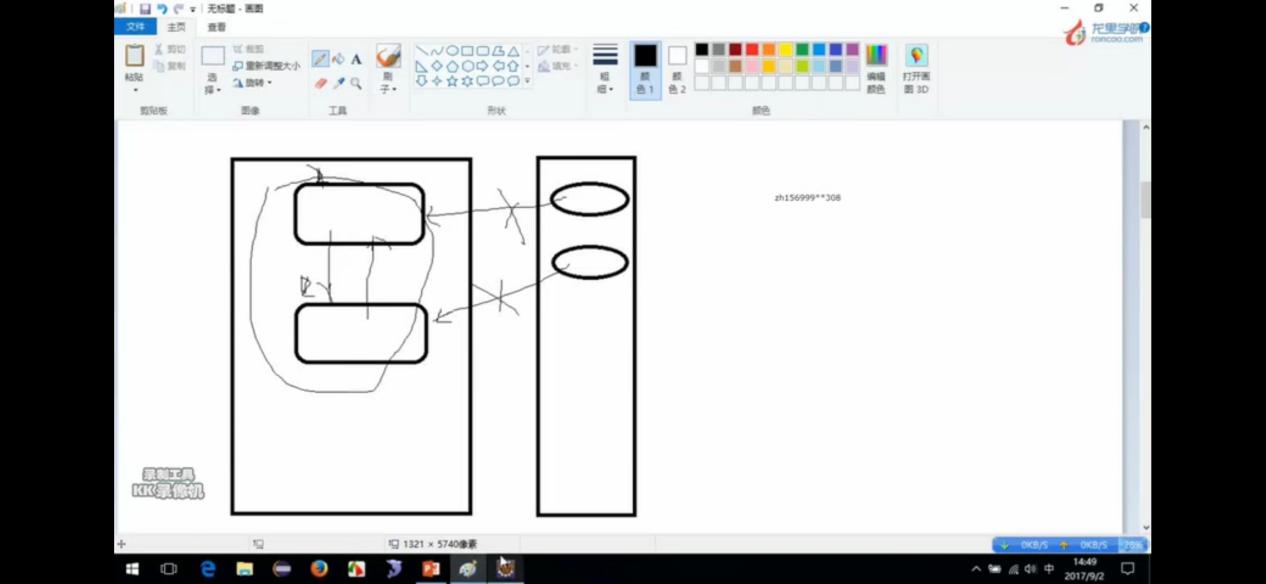
## 对象的访问定位

使用句柄：指向内存一块区域，再根据句柄查找对象

直接指针：引用类型直接指向内存区域的对象地址（hostpot采用直接指针）

## 垃圾回收

1. 如何判断对象为垃圾对象
   1. 引用计数法：对象中添加一个引用计数器，当有地方引用这个对象的时候，引用计数器的值+1，当引用失效的时候，计数器的值-1，当引用为0，就会被回收。

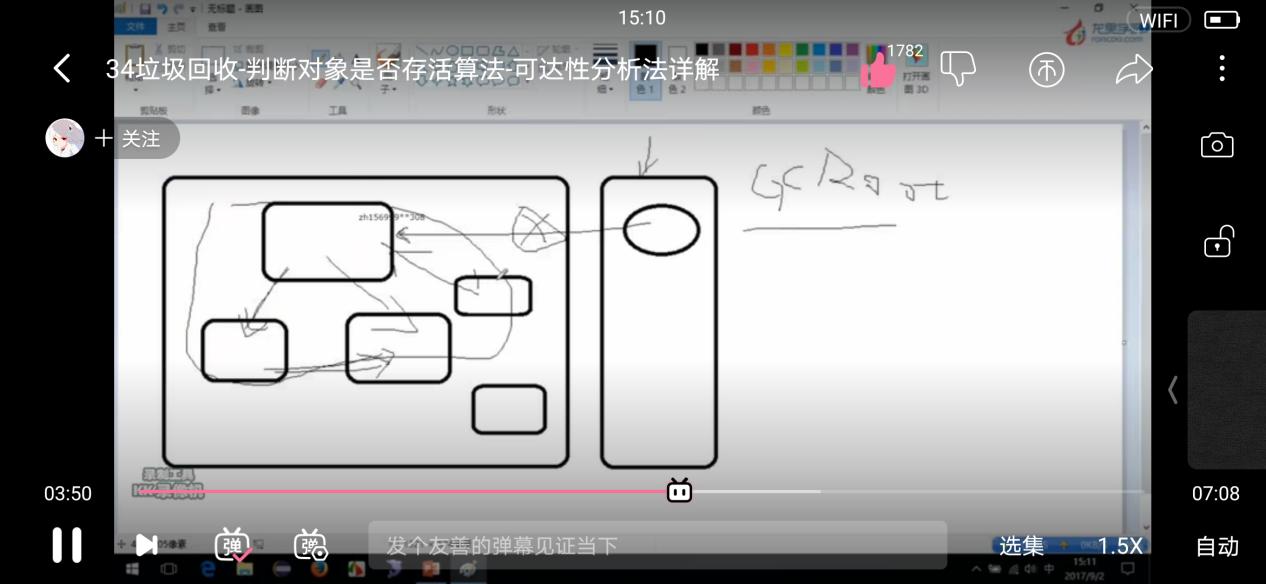


注：栈引用断开，但是堆里两个对象互相引用，导致引用计数器都不为0，两个本该为垃圾的对象不会被回收。

打印垃圾回收信息jvm参数设置：

-verbose:gc ： 开启gc日志  
-XX:+PrintGCDetails ： 打印gc详情  
-XX:+PrintGCDateStamps ： 打印gc时间戳  
-Xloggc:gcc.log ： 将日志输出到文件xx(默认位置为桌面)

* 1. 可达性分析法：



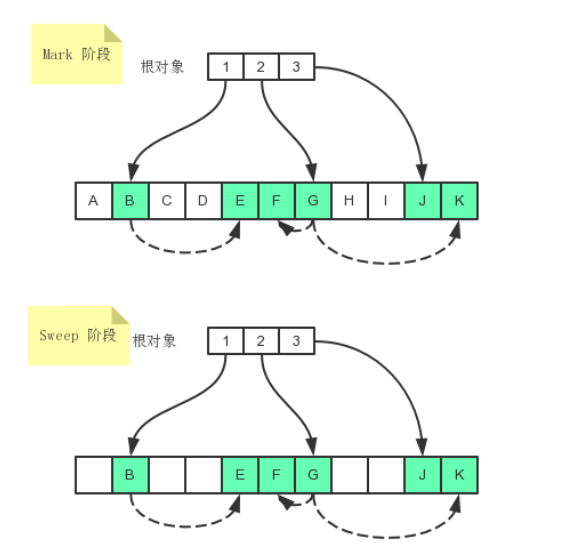
思路：从最外层的栈引用GcRoot开始，可到达的对象引用为有用对象，到达不了即为垃圾对象，若栈引用断开，则堆中相应的对象都为垃圾对象。

1. 如何回收
   1. 回收策略
      * 1. 标记-清除算法：
        2. 复制算法
        3. 标记-整理算法
        4. 分代收集算法
   2. 垃圾算法
      * 1. Serial
        2. Parnew
        3. Cms
2. 何时回收

### 垃圾回收算法

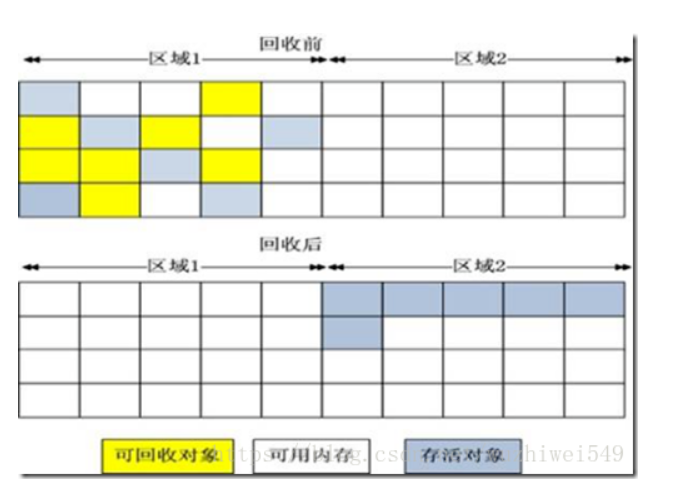
#### 标记-清除算法

* + 1. 回收原理
       1. 标记阶段从GcRoot根节点开始，对可以访问到的对象打上标识，一般是在对象的header中，将其记录为可达对象
       2. 而在清除阶段，collector对堆内存(heap memory)从头到尾进行线性的遍历，如果发现某个对象没有标记为可达对象-通过读取对象的header信息，则就将其回收

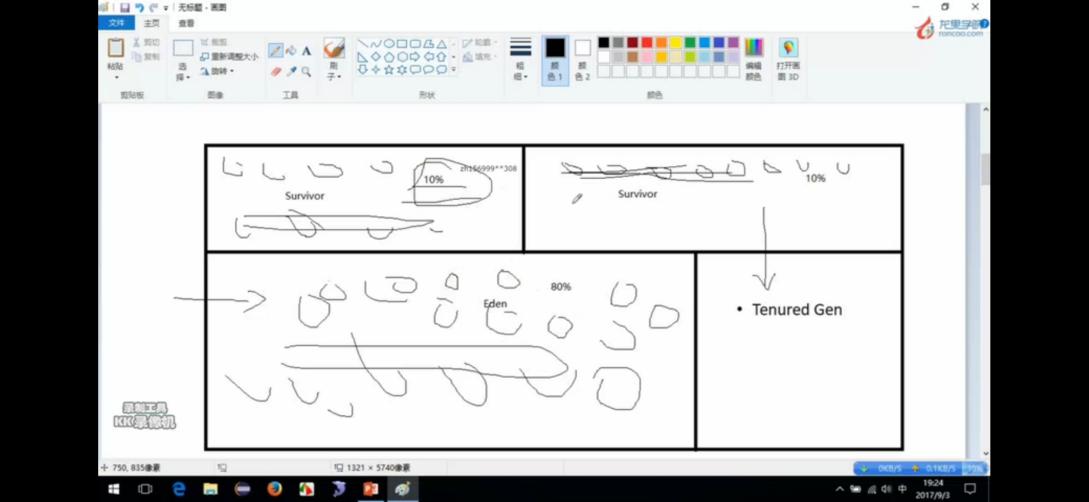


* + 1. 问题：
       1. 效率问题
       2. 空间问题
       3. 标记-清除算法的比较大的缺点就是垃圾收集后有可能会造成大量的内存碎片，像上面的图片所示，垃圾收集后内存中存在三个内存碎片，假设一个方格代表1个单位的内存，如果有一个对象需要占用3个内存单位的话，那么就会导致Mutator一直处于暂停状态，而Collector一直在尝试进行垃圾收集，直到Out of Memory。

#### 复制算法



* + 1. 回收原理：
       1. 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，然后再把已使用过的内存空间一次清理掉。  
            这样使得每次都是对其中的一块进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。
    2. 主要缺点：
       1. 效率问题：在对象存活率较高时，复制操作次数多，效率降低；
       2. 空间问题：內存缩小了一半；需要額外空间做分配担保(老年代)
    3. 内存浪费解决方案：



说明：详细划分三个区域，两个Survivor区域，一个Eden区域,每次对象放入Eden区域，Eden区域满了还会放入第一个Survivor区域，垃圾回收时回收Eden区域（主要）和第一个Survivor区域，存活对象放入第二个Survivor区域。如此内存只浪费10%，比之前50%优化很多。

#### 标记-整理算法

* + 1. 原理：

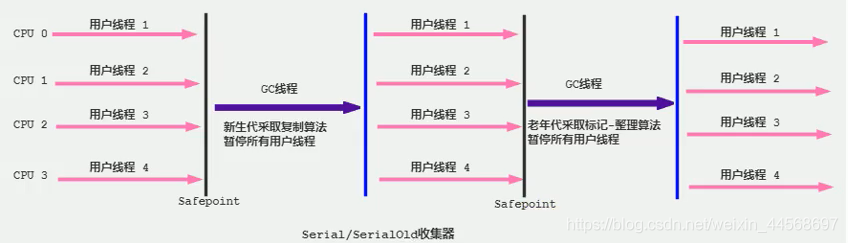
不直接对可回收对象进行清理，而是让所有可用的对象都向一端移动。然后直接清理掉边界意外的内存。

#### 分代收集算法

* + 1. 原理：新生代和老年代采用不同的回收算法
       1. 新生代：每次垃圾收集器都发现有大批对象死去，只有少量存活，采用复制算法，只需要付出少量存活对象的复制成本就可以完成收集
       2. 老年代：中因为对象存活率高、没有额外空间对它进行分配担保，就必须“标记－整理－清除”算法进行回收
       3. 新创建的对象被分配在新生代，如果对象经过几次回收后仍然存活，那么就把这个对象划分到老年代。

### 收集器

#### Serial收集器



Serial（串行）垃圾收集器是最基本、发展历史最悠久的收集器；

单线程的收集器，但它的”单线程“的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。

特点：

针对新生代；采用复制算法；单线程收集；进行垃圾收集时，必须暂停所有工作线程，直到完成

优点：

1、简单而高效

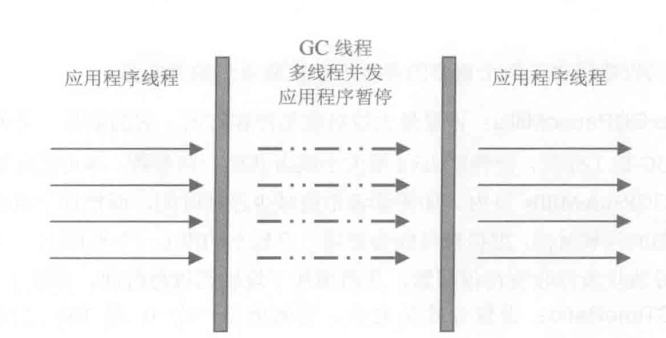
2、在用户的桌面应用场景中，可用内存一般不大，可以在较短时间内完成垃圾收集，只要不频繁发生，使用串行回收器是可以接受的

3、在HotSpot虚拟机中，使用-XX:+UseSerialGC参数可以指定年轻代和老年代都使用串行收集器

总结：

这种垃圾收集器仅做了解，目前已经不再用串行的了。而且在限定单核CPU才可以用。现在都不是单核了。对于交互较强的应用而言，这种垃圾收集器是不能被接受的。一般在Java web应用程序中是不会采用串行垃圾收集器的。

#### ParNew收集器



ParNew 回收器是一个工作在新生代的垃圾回收器，它只是简单地将串行回收器多线程化 ，它的回收策略，算法以及参数和新生代串行回收器一样

ParNew 回收器也是独占式的回收器，在收集过程中，应用程序会全部暂停。但是由于并行回收器使用多线程进行垃圾回收，因此在并发能力比较强的CPU上，它产生的停顿时间要短于串行回收器。在单CPU或者并发能力较弱的系统中，并行回收器的效果不会比串行回收器好。

特点：除了多线程外，其余的行为、特点和Serial收集器一样； 如Serial收集器可用控制参数、收集算法、Stop The World、内存分配规则、回收策略等

#### Parallel Scavenge收集器

吞吐量优先收集器，和ParNew收集器类似，是一个新生代收集器。使用复制算法的并行多线程收集器。Parallel Scavenge是Java1.8默认的收集器，特点是并行的多线程回收，以吞吐量优先。

特点：关注点与其他收集器不同， Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量（Throughput）  
(吞吐量=运行用户代码时间/(运行用户代码时间+垃圾收集时间))

参数设置：

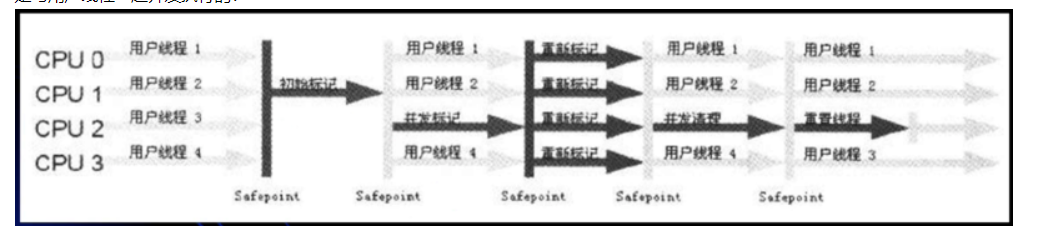
-XX:MaxGCPauseMillis参数GC停顿时间，500MB ——>300MB,这个参数配置太⼩的话会发⽣频繁GC

-XX:GCTimeRatio参数(吞吐量)，99%

#### CMS收集器

并发标记清理（Concurrent Mark Sweep，CMS） 收集器也称为 并发低停顿收集器（Concurrent Low Pause Collector）或低延迟（low-latency）垃圾收集器。

CMS是基于“标记-清除”算法实现的，整个过程分为4个步骤：  
 1、初始标记（CMS initial mark）。  
 2、并发标记（CMS concurrent mark）。  
 3、重新标记（CMS remark）。  
 4、并发清除（CMS concurrent sweep）。  
 注意：“标记”是指将存活的对象和要回收的对象都给标记出来，而“清除” 是指清除掉将要回收的对象。



优点：

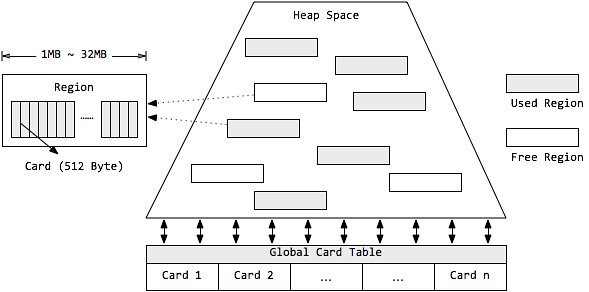
并发收集、低停顿【注意：这里的停顿指的是停止用户线程】，Oracle公司的一些官方文档中也称之为并发低停顿收集器（Concurrent Low Pause Collector）。

缺点：  
 1、CMS收集器对CPU资源非常敏感。  
 2、CMS收集器无法处理浮动垃圾（Floating Garbage，就是指在之前判断该 对象不是垃圾，由于用户线程同时也是在运行过程中的，所以会导致判断不准确的， 可能在判断完成之后在清除之前这个对像已经变成了垃圾对象，所以有可能本该此 垃圾被回收但是没有被回收，只能等待下一次GC再将该对象回收，所以这种对像 就是浮动垃圾）,可能出现“Concurrent Mode Failure”失败而导致另一次Full GC 的产生。如果在应用中老年代增长不是太快，可能适当调高参数-XX： CMSInitiatingOccupancyFraction的值来提高触发百分比，以便降低内存回收次 数从而获取更好的性能。要是CMS运行期间预留的内存无法满足程序需要时，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设置得 太高很容易导致大量“Concurrent Mode Failure”失败，性能反而降低。  
 3、收集结束时会有大量空间碎片产生，空间碎片过多时，将会给大对象分配 带来很大麻烦，往往出现老年代还有很大空间剩余，但是无法找到足够大的连续空 间来分配当前对象，不得不提前进行一次Full GC。CMS收集器提供了一个 -XX:+UseCMSCompactAtFullCollection开关参数（默认就是开启的），用于在CMS 收集器顶不住要进行Full GC时开启内存碎片的合并整理过程，内存整理的过程是 无法并发的，空间碎片问题没有了，但停顿时间不得不变长。

CMS收集器收集完整步骤：

* Phase1 ：Initial Mark【初始标记】
* Phase2 : Concurrent Mark 【并发标记】
* Phase3 : Concurrent Preclean【并发预先清除】
* Phase4 : Concurrent Abortable Preclean【并发可能失败的预先清除】
* Phase5 : Final Remark【最终重新标记】
* Phase6 : Concurrent Sweep【并发清除】
* Phase7 : Concurrent Reset【并发重置】

#### (5)G1(Garbage First)收集器



## **3、内存分配**

内存分配策略

