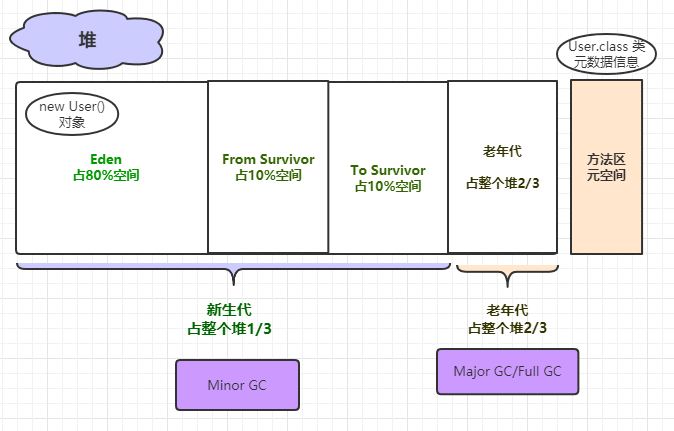
高薪必备JVM面试题解惑

主讲：Cat 老师

北京动力节点教育科技有限公司  
2021 • 北京

动力节点•版权所有•禁止传播

## 堆为什么要分成年轻代和老年代？



因为年轻代和老年代不同的特点，需要采用不同的垃圾回收算法；

年轻代的对象，它的特点是创建之后很快就会被回收，所以需要用一种垃圾回收算法；

老年代的对象，它的特点是需要长期存活，所以需要另外一种垃圾回收算法 ；

所以需要分成两个区域来放不同的对象；

1、绝大多数对象都是朝生夕灭的；

如果一个区域中大多数对象都是朝生夕灭，那么把它们集中放在一起，每次回收时只关注如何保留少量存活对象，而不是去标记那些大量将要被回收的对象，就能以较低的代价回收到大量的空间；

2、熬过越多次垃圾收集的对象就越难以回收；

如果是需要长期存活的对象，那把它们集中放在一块，虚拟机便可以使用较低的频率来回收这个区域，这就同时兼顾了垃圾收集的时间开销和内存的空间有效利用；

3、JVM划分出新生代、老年代之后，垃圾收集器可以每次只回收其中某一个或者某些部分的区域 ，同时也有了“Minor GC”“Major GC”“Full GC”这样的回收类型的划分；

Minor GC/Young GC ：新生代收集

Major GC/Old GC：老年代收集

Full GC：整堆收集，收集整个Java堆和元空间/方法区的垃圾收集；

Mixed GC：混合收集，收集整个新生代以及部分老年代的垃圾收集，目前只有G1收集器会有这种行为；

4、针对不同的区域对象存亡特征采用不同的垃圾收集算法：

（1）复制算法

（2）标记-清除算法

（3）标记-整理算法

## JVM堆的年轻代为什么要有两个Survivor区？

**1、如果没有Survivor区会怎么样？**

此时每触发一次Minor GC，就会把Eden区的对象复制到老年代，这样当老年代满了之后会触发Major Gc/Full GC(通常伴随着MinorGC)，比较耗时，所以必须有Survivor区；

**2、如果只有1个Survivor区会怎么样？**

刚刚创建的对象在Eden中，一旦Eden满了，触发一次Minor GC，Eden中存活的对象就会被移动到Survivor区，下一次Eden满了的时候，此时进行Minor GC，Eden和Survivor各有一些存活对象，因为只有一个Survivor，所以Eden区第二次GC发现的存活对象也是放入唯一的一个Survivor区域中，但此时把Eden区的存活对象硬放到Survivor区，很明显这两部分对象所占有的内存是不连续的，也就导致了内存碎片化问题，并且由于不连续的空间会导致再分配大对象的时候，由于没有连续的空间来分配，会导致提前垃圾回收；

如果将Survivor中的所有存活对象进行整理消除碎片，然后将所有的存活对象放入其中，这样做会降低效率；

如果把两个区域中的所有存活对象都复制转移到一个完全独立的空间中，也就是第二块Survivor中，这样就可以留出一块完全空着的Eden和Survivor了，下次GC的时候再重复这个流程，所以我们便要有两个Survivor区；

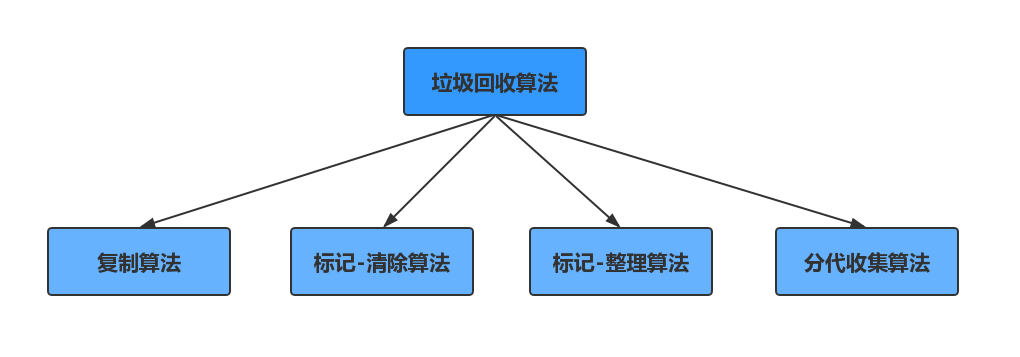
## Eden区与Survivor区的空间大小比值为什么默认是8:1:1？

一个eden区 ，新生代对象出生的地方；

两个survivor区，一个用来保存上次新生代GC存活下来的对象，还有一个空着，在新生代GC时把eden+survivor中存活对象复制到这个空的survivor中；

统计和经验表明，90%的对象朝生夕死存活时间极短，每次gc会有90%对象被回收，剩下的10%要预留一个survivor空间去保存；

## 请介绍下JVM中的垃圾回收算法？



### 标记-清除算法

标记-清除算法是最基础的收集算法，后续的很多垃圾回收算法是基于该算法而发展出来的，它分为‘ 标记 ’和‘ 清除 ’两个阶段；

**1、标记**

标记出所有需要回收的对象，在标记完成后，统一回收掉所有被标记的对象，也可以反过来，标记出所有存活的对象，在标记完成后，统一回收所有未被标记的对象，标记过程就是对象是否属于垃圾的判定过程，基于可达性分析算法判断对象是否可以回收；

**2、清除**

标记后，对所有被标记的对象进行回收；



优点：基于最基础的可达性分析算法，实现简单，后续的收集算法都是基于这种思想实现的；

缺点：

1. 执行效率不稳定，如果Java堆中包含大量对象，而且其中大部分是需要被回收的，这时必须进行大量标记和清除的动作，导致标记和清除两个过程的执行效率都随对象数量增长而降低；

2、内存空间的碎片化问题，标记、清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致当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收集；

### 复制算法

复制算法是标记-复制算法的简称，将可用内存按容量分为大小相等的两块，每次只使用其中一块，当这一块的内存用完了，就将还存活的对象复制到另外一块内存上，然后再把已使用过的内存空间一次清理掉；



优点：实现简单，效率高，解决了标记-清除算法导致的内存碎片问题；

缺点：

1. 代价太大，将可分配内存缩小了一半，空间浪费太多了；
2. 对象存活率较高时就要进行较多的复制操作，效率将会降低；

一般虚拟机都会采用该算法来回收新生代，但是JVM对复制算法进行了改进，JVM并没有按照1:1的比例来划分新生代的内存空间，因为通过大量的统计和研究表明，90%以上的对象都是朝生夕死的，所以JVM把新生代分为一块较大的Eden空间和两块较小的Survivor空间，每次分配内存只使用Eden和其中一块Survivor，发生垃圾收集时，将Eden和Survivor中仍然存活的对象一次性复制到另外一块Survivor空间上，然后直接清理掉Eden和已用过的那块Survivor空间，HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也即每次新生代中可用内存空间为整个新生代容量的90%（Eden的80%加上一个Survivor的10%），只有另外一个Survivor空间即10%的新生代会被“浪费”；

当然，90%的对象可被回收仅仅是大部分情况下，我们无法百分百保证每次回收都只有不多于10%的对象存活，因此JVM还有一个空间担保机制的安全设计，当Survivor空间不足以容纳一次Minor GC之后存活的对象时，就需要依赖其它内存区域（实际上就是老年代）进行空间分配担保（Handle Promotion，也就是冒险Minor GC一下）；

### 标记-整理算法

标记-整理算法是根据老年代的特点而产生的；

1、标记

标记过程与上面的标记-清理算法一致，也是基于可达性分析算法进行标记；

2、整理

和标记-清理不同的是，该算法不是针对可回收对象进行清理，而是根据存活对象进行整理，让存活对象都向一端移动，然后直接清理掉边界以外的内存；

而标记-清除算法不移动存活对象，导致有大量不连续空间，即内存碎片，而老年代这种每次回收都有大量存活对象的区域，移动存活对象并更新所有引用这些对象的引用，这是一种比较耗时的操作，而且这种对象移动操作必须全程暂停用户应用程序才能进行，像这样的停顿我们也称为“Stop The World”即STW；

但是即便是移动存活对象是耗时的操作，但是如果不这么做，那么在充满内存碎片的空间中分配对象，又影响了对象的分配和访问的效率，所以JVM权衡两者之后，还是采用了移动存活对象的方式，也就是对内存进行了整理；

另外像cms垃圾收集器，平时多数时间都采用标记-清除算法，暂时容忍内存碎片的存在，直到内存空间的碎片化程度已经大到影响对象分配时，再采用标记-整理算法收集一次，以获得规整的内存空间，所以像基于标记-清除算法的CMS收集器面临空间碎片过多时就会进行一次整理；



优点：

1. 不会像复制算法那样划分两个区域，提高了空间利用率；

2、不会产生不连续的内存碎片；

缺点：效率问题，除了像标记-清除算法的标记过程外，还多了一步整理过程，效率变低；

### 分代收集算法

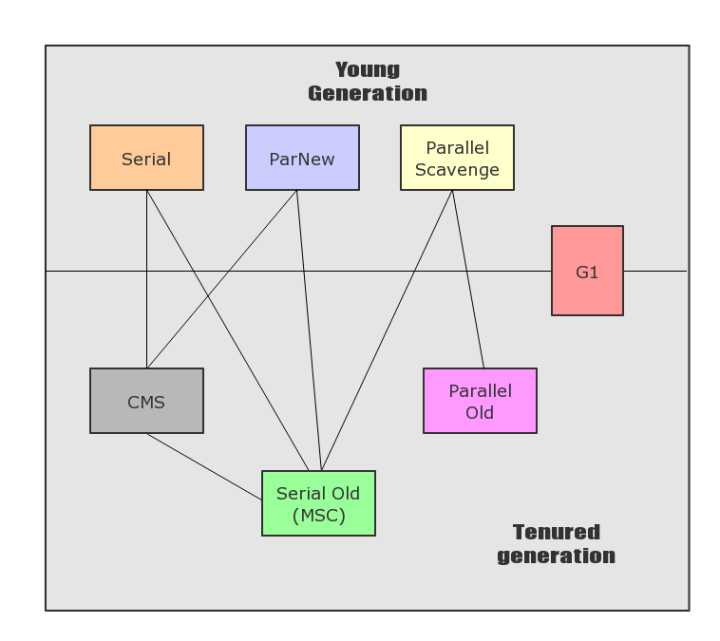
现在一般虚拟机的垃圾收集都是采用“ 分代收集 ”算法；

根据对象存活周期的不同将内存划分为几块，一般把java堆分为新生代和老年代，JVM根据各个年代的特点采用不同的收集算法；

新生代中，每次进行垃圾回收都会发现大量对象死去，只有少量存活，因此采用复制算法，只需要付出少量存活对象的复制成本就可以完成收集；

老年代中，因为对象存活率较高，采用标记-清理、标记-整理算法来进行回收；

## 请介绍一下JVM垃圾收集器？



如上图，一共有7种作用于不同分代的垃圾收集器，如果两个收集器之间存在连线，则说明它们可以搭配使用，垃圾收集器所处区域表示它是属于新生代收集器还是老年代收集器；

新生代收集器：Serial、ParNew、Parallel Scavenge [ˈpærəlel] [ˈskævɪndʒ]

老年代收集器：CMS、Serial Old、Parallel Old

整堆收集器： G1

垃圾收集器的最前沿成果：ZGC 和 Shenandoah

### Serial收集器

新生代收集器，最早的收集器，单线程的，收集时需暂停用户线程的工作，所以有卡顿现象，效率不高，致使java语言的开发团队一直在改进垃圾收集器的算法和实现，但Serial收集器简单，不会有线程切换的开销，是Client模式下默认的垃圾收集器，-client， -server；

参数： -XX:+UseSerialGC

java -XX:+PrintFlagsFinal -version 打印jvm默认的参数值；



### ParNew收集器

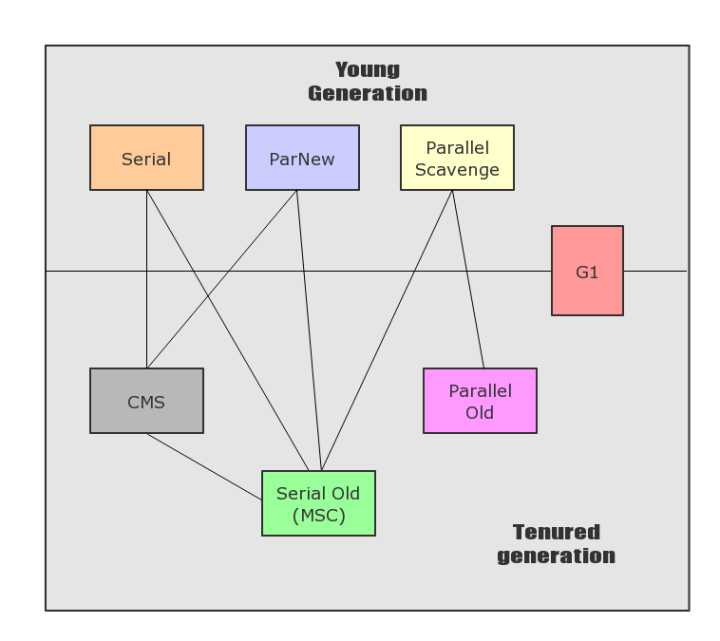
它是新生代收集器，就是Serial收集器的多线程版本，大部分基本一样，单CPU下，ParNew还需要切换线程，可能还不如Serial；

Serial和ParNew收集器可以配合CMS收集器，前者收集新生代，后者CMS收集老年代，

"-XX:+UseConcMarkSweepGC"：指定使用CMS后，会默认使用ParNew作为新生代垃圾收集器；  
"-XX:+UseParNewGC"：强制指定使用ParNew；  
"-XX:ParallelGCThreads=2"：指定垃圾收集的线程数量，ParNew默认开启的收集线程与CPU的数量相同；



### Parallel Scavenge收集器



简称Parallel，它是新生代收集器，基于复制算法，并行的多线程收集器（与ParNew收集器类似），侧重于达到一个可控的吞吐量，虚拟机运行100分钟，垃圾收集花1分钟，则吞吐量为99%，有时候我们也把该垃圾收集器叫吞吐量垃圾收集器或者是吞吐量优先的垃圾收集器；而且这个垃圾收集器是jvm默认的新生代的垃圾收集器；

它提供一个参数设置吞吐量：

-XX:MaxGCPauseMillis 该参数设置大于0的毫秒数，每次GC的时间将尽量保持不超过设置的值，但是这个值也不是设置得越小就越好，GC暂停时间越短，那么GC的次数会变得更频繁；

-XX:+UseAdaptiveSizePolicy 自适应新生代大小策略，默认这个参数是开启的，当这个参数被开启之后，就不需要人工指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX：SurvivorRatio）、晋升老年代对象大小（-XX：PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间获得最大的吞吐量，这种调节方式称为垃圾收集的自适应的调节策略（GC Ergonomics）；

如果我们不知道怎么对jvm调优，我们可以使用Parallel Scavenge收集器配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成也许是一个很不错的选择，只需要把基本的内存数据设置好（如-Xmx设置最大堆），然后使用-XX:MaxGCPauseMillis参数（最大停顿时间）给虚拟机设立一个优化目标，那具体细节参数的调节工作就由虚拟机完成，自适应调节策略也是Parallel Scavenge收集器区别于ParNew收集器的一个重要特性；

参数：-XX:+UseParallelGC 指定使用Parallel Scavenge垃圾收集器

java -XX:+PrintCommandLineFlags -version 打印jvm默认初始堆和最大堆大小以及垃圾收集器

java -XX:+PrintFlagsFinal -version 打印jvm所有的默认的参数值；

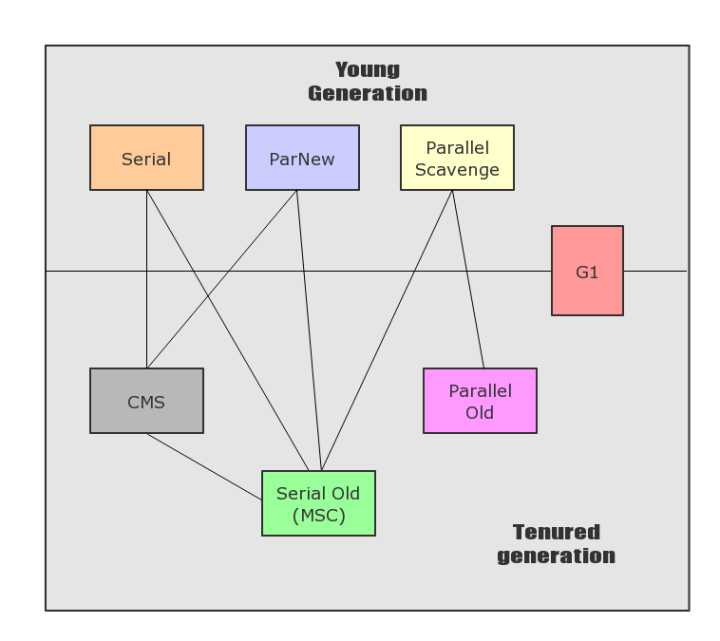
-XX:+PrintGCDetails

-XX:+PrintGCDateStamps

-Xloggc:d:/dev/gc.log

Parallel Scavenge垃圾收集器中的Ergonomics负责自动的调节gc暂停时间和吞吐量之间的平衡，自动优化虚拟机的性能；

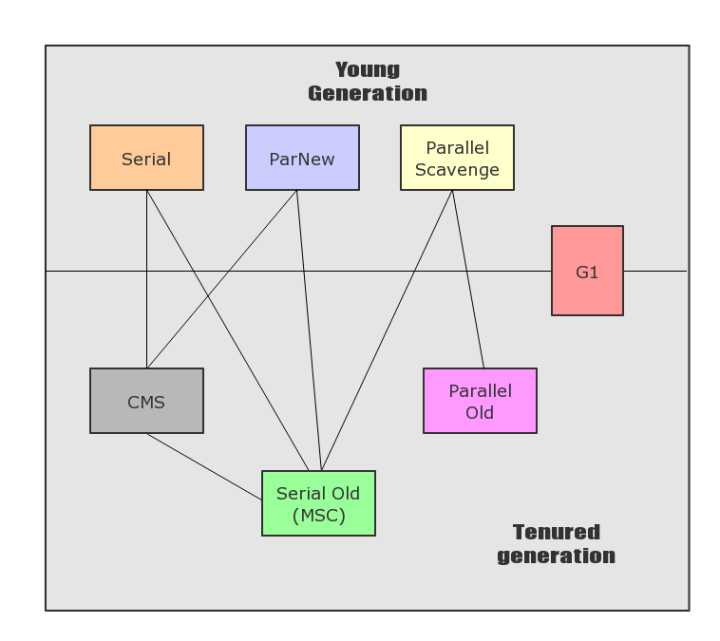
### Serial Old收集器



它是Serial收集器的老年代版本，同Serial一样，单线程，可在Client模式下使用，也可在Server模式下使用，采用标记-整理算法，Serial Old收集器也可以作为CMS 收集器发生失败时的后备预案，在并发收集发生Concurrent Mode Failure时使用；



### Parallel Old收集器



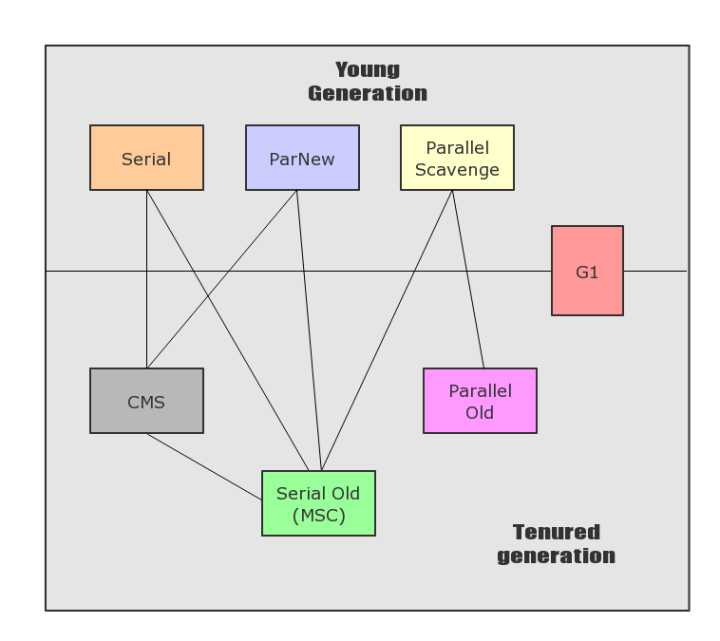
是Parallel Scavenge的老年代版本，多线程，标记整理算法，它是在jdk1.6开始才提供；

在注重吞吐量和CPU资源的情况下， Parallel Scavenge新生代+ Parallel Old老年代是一个很好的搭配；

参数：-XX:+UseParallelOldGC 指定使用Parallel Old收集器；



### CMS收集器



CMS全称Concurrent Mark Sweep，是一款老年代的垃圾收集器，它是追求最短回收停顿时间为目标的收集器，互联网B/S结构的服务器端特别适合此收集器；

我们知道垃圾回收会带来Stop the World（stw）的问题，会导致系统卡死时间过长，很多响应无法处理，所以CMS垃圾回收器采取的是垃圾回收线程和系统工作线程尽量同时执行的模式来处理的，基于标记-清除算法；

参数：-XX:+UseConcMarkSweepGC 指定使用CMS垃圾收集器；

-XX:+PrintGCDetails

-XX:+PrintGCDateStamps

-Xloggc:D:/dev/gc.log

CMS垃圾收集器的运作过程，分为4个阶段：

1、初始标记（stw，标记一下GC Roots能直接关联到的对象，那么这些对象也就是需要存活的对象，速度很快）；

2、并发标记（不会stw，追踪GC Roots的整个链路，从GC Roots的直接关联对象开始遍历整个对象引用链路，这个过程耗时较长但是不需要停顿用户线程，可以与垃圾收集线程一起并发运行）

3、重新标记（stw，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化的那一部分对象的标记记录，这个阶段的停顿时间通常会比初始标记阶段稍长一些，但也远比并发标记阶段的时间短，它其实就是对在第二阶段中被系统程序运行变动过的少数对象进行标记，所以运行速度很快的）

4、并发清除（不会stw，清理删除掉标记阶段判断的已经死亡的对象，这个阶段其实是很耗时的，但由于不需要移动存活对象，并且这个阶段也是与用户线程同时并发执行的）

其中初始标记和重新标记需要暂停用户线程（Stop The World），其它阶段都是并发执行，所以总体上暂停时间更短；

**CMS垃圾收集器的缺点：**

1、并发收集会占用CPU资源，特别是cpu数量小的服务器下，会占用用户线程，导致性能下降，CMS默认启动的回收线程数是（处理器核心数量 + 3）/ 4；

2、会产生浮动垃圾，因为你并发清除的时候用户线程可能还在产生垃圾，这些垃圾没有清除，而且你不能让老年代填满了再清除，你要给用户线程留一定空间，所以jdk1.5默认是老年代68%了就触发回收，jdk1.6则提升到92%；

通过-XX:CMSInitiatingOccupancyFraction参数设置；

默认现在是取 -XX:CMSTriggerRatio 的值，默认是80%；

-XX:CMSInitiatingOccupancyFraction设置得更高，可以降低内存回收频率，获取更好的性能，但这又会更容易面临另一种风险：要是CMS运行期间预留的内存无法满足程序分配新对象的需要，就会出现一次并发清除失败“并发失败”（Concurrent Mode Failure），如果预留老年代不够用户线程使用，则启用Serial Old收集，这就会暂停用户线程，导致性能下降；

3、CMS基于标记-清除算法，清理后会产生碎片空间，空间碎片过多时，将会导致大对象无法分配，往往会出现老年代还有很多剩余空间，但没有足够大的连续空间来分配当前对象，而不得不提前触发一次Full GC，CMS垃圾收集器有一个

-XX:+UseCMSCompactAtFullCollection开关参数（默认是开启的），用于在CMS收集器进行Full GC时开启内存碎片的整理过程，由于这个内存整理必须移动存活对象，整理过程是无法并发的，就导致停顿时间变长，因此虚拟机设计者们还提供了另外一个参数

-XX:CMSFullGCsBeforeCompaction，即CMS收集器在执行过若干次（数量由参数值决定）不整理空间的Full GC之后，下一次进入Full GC前会先进行碎片整理（默认值为0，表示每次进入Full GC时都进行碎片整理）

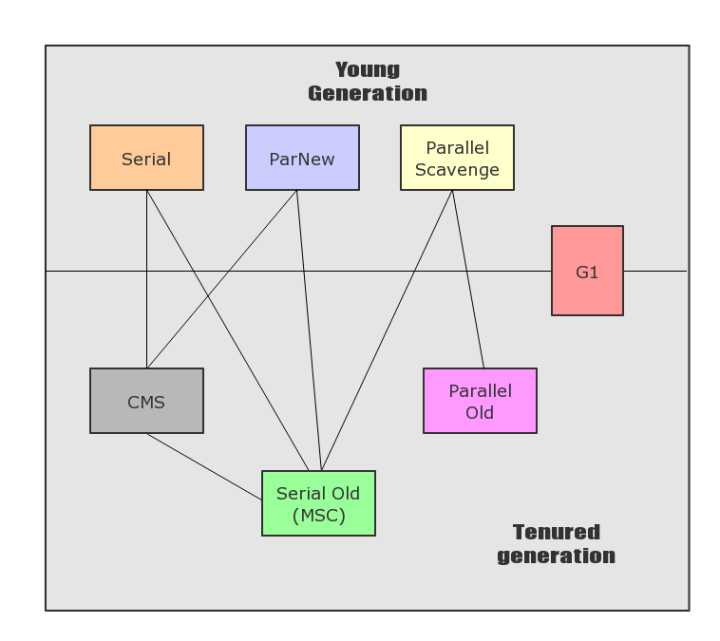


java -XX:+PrintCommandLineFlags -version 打印jvm默认初始堆和最大堆大小以及垃圾收集器

java -XX:+PrintFlagsFinal -version 打印jvm所有的默认的参数值；

### G1收集器

#### G1垃圾收集器的基本介绍



G1全称Garbage First，G1垃圾回收器可以同时回收新生代和老年代，不需要两个垃圾回收器配合起来使用；

G1垃圾收集器是目前可用于生产环境的最前沿最先进的垃圾收集器，从JDK1.6u14开始试验，到JDK1.7u4达到成熟，直到JDK1.8u40才正式完成，开始可以使用；

JDK 9发布时，G1宣告取代Parallel Scavenge加Parallel Old组合，成为服务端模式下的默认垃圾收集器，而CMS则被声明为不推荐使用（Deprecate）的收集器，如果对JDK 9及以上版本的HotSpot虚拟机使用参数-XX:+UseConcMarkSweepGC来开启CMS收集器的话，用户会收到一个警告信息，提示CMS未来将会被废弃，CMS后续将退出历史舞台；

#### G1垃圾收集器的基本原理

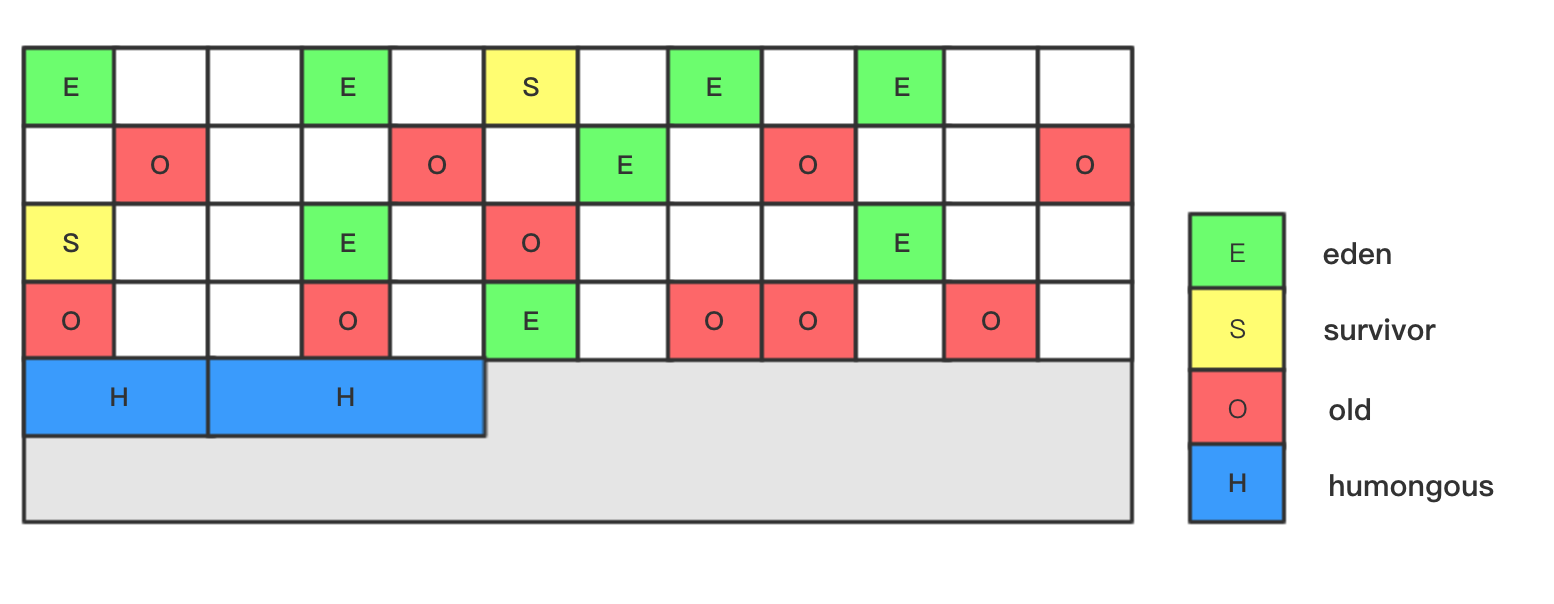
G1是一款可以让我们设置垃圾回收的预期停顿时间的垃圾收集器，设置参数是-XX:MaxGCPauseMillis，默认值是200ms；

其实我们对内存合理分配，优化jvm参数，就是为了尽可能减少新生代（Minor GC），或者是整个老年代（Major GC），或者是整个Java堆（Full GC），尽量减少GC带来的系统停顿，避免影响系统处理请求，G1可以指定垃圾回收导致的系统停顿时间不能超过多久，不管垃圾的多与少，垃圾回收的时间都不要超过我们设置的值（并不是绝对的），G1全权给你负责，保证达到这个目标，这相当于我们就可以直接控制垃圾回收对系统性能的影响了；

所以G1垃圾收集器是尽量把垃圾回收对系统造成的影响控制在你指定的时间范围内，同时在有限的时间内尽量回收尽可能多的垃圾对象，这就是G1垃圾收集器的核心原理；

#### G1垃圾收集器如何做到可预测的停顿时间？

1、这与G1垃圾收集器独特的设计有关，它最大的特点就是把Java整个堆内存拆分为多个大小相等的Region [ˈridʒən] ；区域、分区



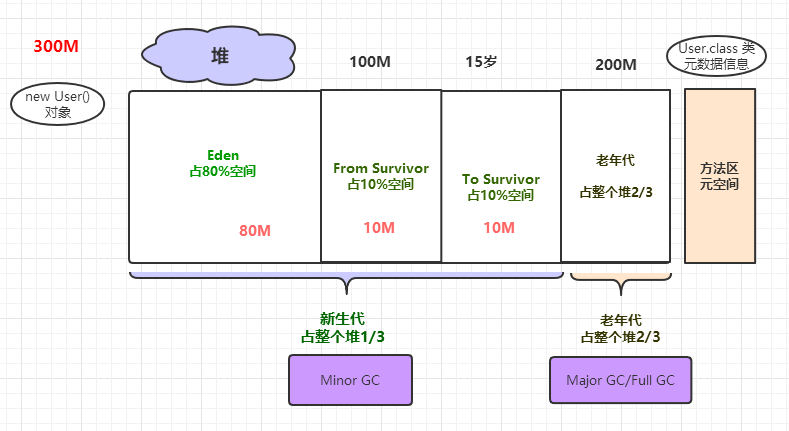
1. G1它会追踪每个Region的回收价值，即它会计算每个Region里的对象有多少是垃圾，如果对这个Region进行垃圾回收，需要耗费多长时间，可以回收掉多少垃圾？

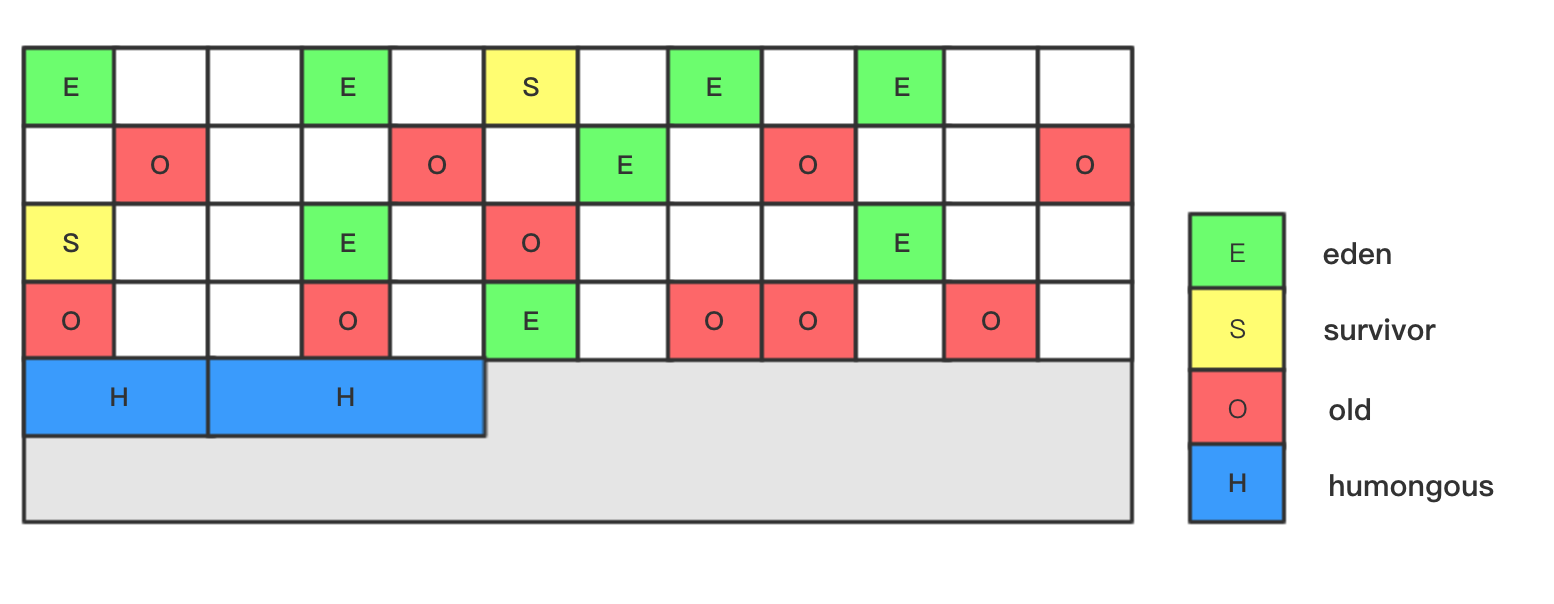
3、G1收集器之所以能建立可预测的停顿时间模型，是因为它将Region作为垃圾回收的最小单元，即每次可以选择一部分Region进行收集，避免在整个Java堆中进行全区域的垃圾收集，让G1收集器去跟踪各个Region里面的垃圾的“回收价值”，然后根据用户设定的收集停顿时间（使用参数-XX:MaxGCPauseMillis指定，默认值是200毫秒），然后在后台维护一个优先级列表，优先处理回收价值大的那些Region，这也是“Garbage First”名字的由来，这种使用Region划分堆内存空间，基于回收价值的回收方式，保证了G1收集器在有限的时间内尽可能收集更多的垃圾；

比如：G1通过追踪发现，1个Region中的垃圾对象有10MB，回收它需要耗费500毫秒，另一个Region中的垃圾对象有20MB，回收它需要耗费100毫秒，那么G1垃圾收集器基于回收价值计算会选择回收20MB只需要100毫秒的Region；

#### G1垃圾收集器是否还有年代的划分？

G1也有新生代和老年代的概念，但只不过是逻辑上的概念，也就是说一个Region此时是属于新生代的Eden空间，过一会儿可能就属于老年代空间，也就是一个Region在运行过程中动态地扮演着新生代的Eden空间、Survivor空间，或者老年代空间，每个Region并不是固定属于某一个空间，另外新生代、老年代也不一定是连续空间，可能是分开的；



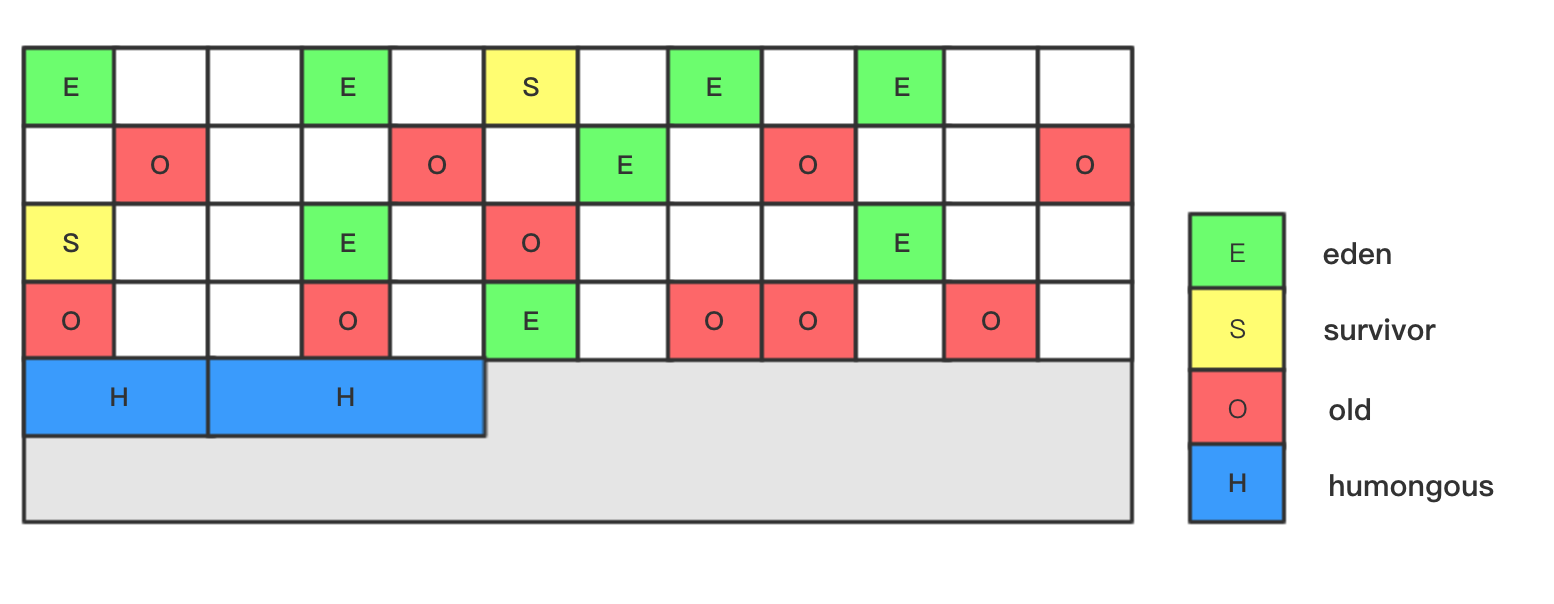


刚开始Region是空的，可能谁都不属于，然后系统创建对象就分配给了新生代，这个Region被新生代对象放满之后，后续垃圾回收了这个Region，然后下一次同一个Region可能又被分配了老年代，用来放老年代的长时间需要存活的对象，所以Region随时会属于新生代也会属于老年代；

新生代和老年代各自的内存区域在不停地变动，由G1自动控制，也就是Region动态分配给新生代或者老年代，按需分配，然后触发垃圾回收的时候，可以根据设定的预期系统停顿时间，来选择最少回收时间和最多回收对象的Region进行垃圾回收，保证GC对系统停顿的影响在可控范围内，同时还能尽可能回收最多的对象；

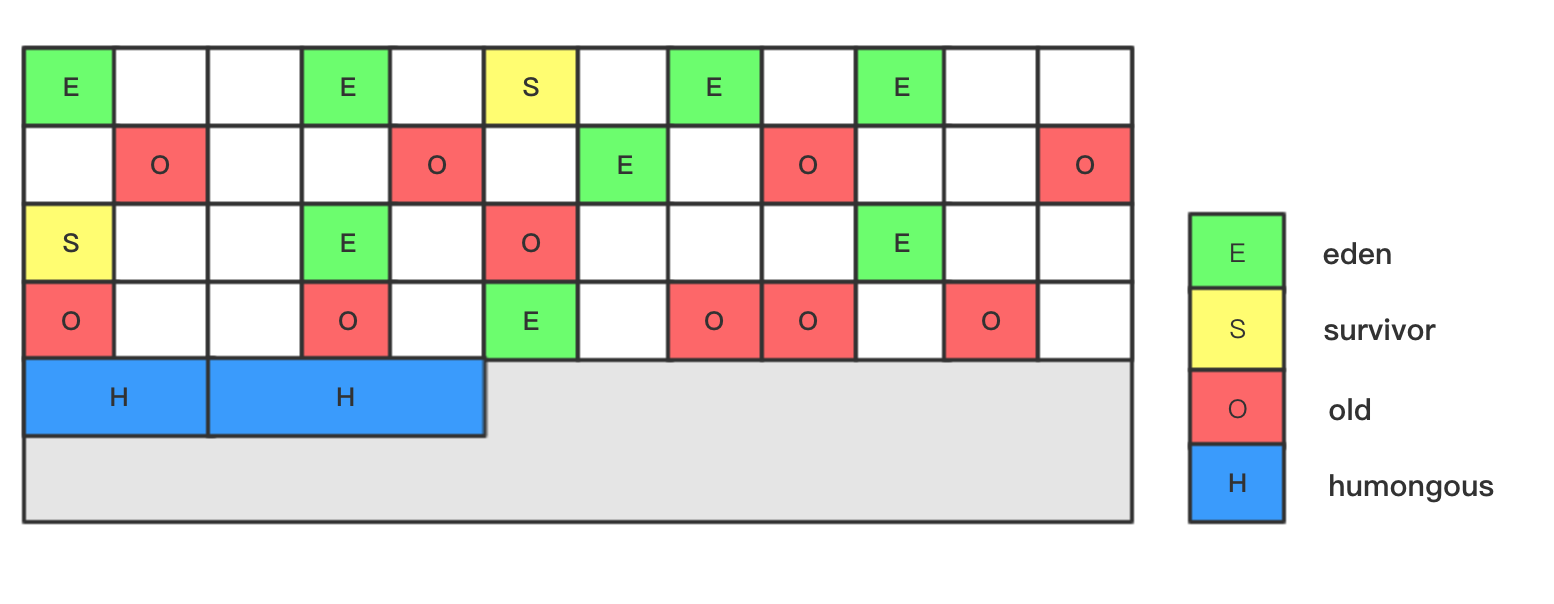
#### G1垃圾收集器中的大对象？

Region中有一类特殊的Humongous [hjuːˈmʌŋgəs]区域，专门用来存储大对象；



G1认为只要大小超过了一个Region容量一半的对象即可判定为大对象，每个Region的大小可以通过参数-XX:G1HeapRegionSize设定，取值范围为1MB～32MB，而对于那些超过了整个Region容量的超级大对象，将会被存放在N个连续的Humongous Region之中，G1的大多数行为都把Humongous Region作为老年代的一部分来进行看待；

#### G1垃圾收集器内存大小如何设置？



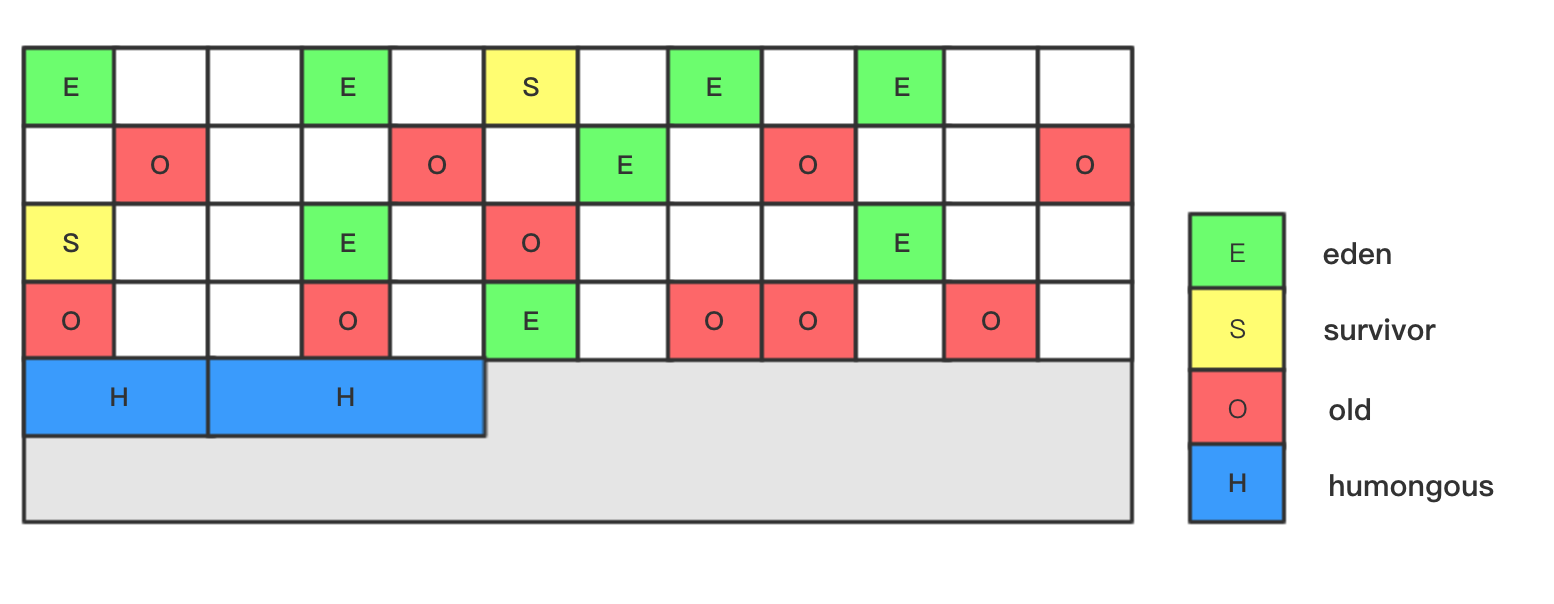
每个region = 1m ~32m，最多有2048个region；

G1对应的是一大堆的Region内存区域，最多可以有2048个Region，比如说堆大小是4G（4096MB），那么每个Region的大小就是2MB，Region的取值范围是1M-32M，可以通过参数“-XX:G1HeapRegionSize”指定每个Region是多少兆；

比如说堆大小是4G（4096MB），刚开始，默认新生代对堆内存的占比是5%，也就是占据200MB左右的内存，对应大概是100个Region，可以通过“-XX:G1NewSizePercent”来设置新生代初始占比，一般默认值即可，因为在系统运行中，JVM会不停的给新生代增加更多的Region，但是最多新生代的占比不会超过60%，可以通过“-XX:G1MaxNewSizePercent”设置，并且一旦Region进行了垃圾回收，此时新生代的Region数量就会减少，这些都是动态的；

新生代 ：老年代 = 60% ：40%

#### G1垃圾收集器新生代还有Eden和Survivor吗？



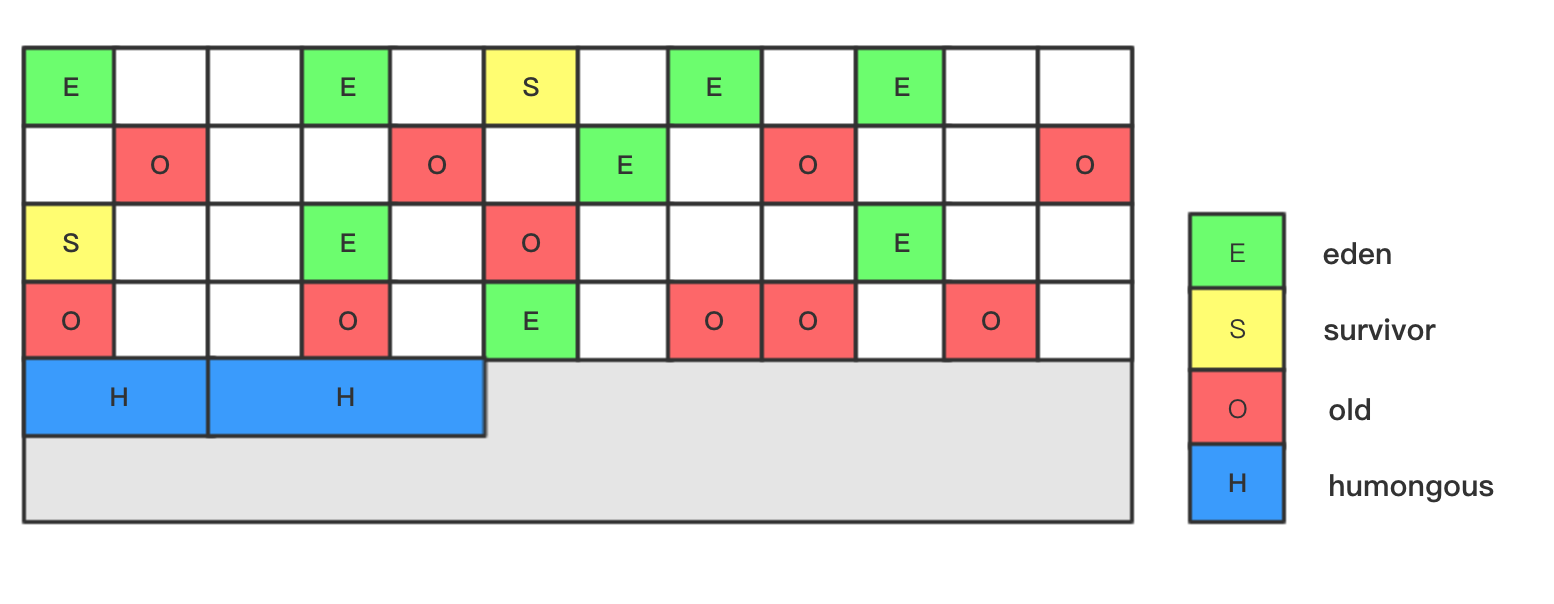
G1垃圾收集器依然有新生代、老年代的概念，新生代里依然有Eden和Survivor的划分，G1是从CMS发展过来的，以后是要完全取代CMS垃圾收取器的，从jdk9开始G1已经是默认的垃圾收集器，之前的很多技术原理在G1中依然可用，我们知道新生代有一个参数“-XX:SurvivorRatio=8”，所以G1还是可以区分出来属于新生代的Region里哪些属于Eden，哪些属于Survivor；

比如新生代刚开始初始化时有100个Region，那么可能有80个Region是Eden，10个Region分别是两个Survivor，所以G1中依然有Eden和Survivor的概念，它们会各自占据不同的Region；

只不过随着对象不停的在新生代里分配，属于新生代的Region会不断增加，Eden和Survivor对应的Region也会不断增加；

5% -- 60%

#### G1垃圾收集器的新生代垃圾回收



G1的新生代也有Eden和Survivor，其触发垃圾回收的机制也是类似的，随着不停在新生代Eden对应的Region中放对象，JVM就会不停的给新生代加入更多的Region，直到新生代占据堆大小的最大比例60%；

假设堆 4G，最大2048个region，每个region为2M，新生代最大60%=2.4G；

一旦新生代达到了设定的占据堆内存的最大大小60%，按照上面的数据大概就是有1200个Region，里面的Eden可能占据了1000个Region，每个Survivor是100个Region，而且Eden区满了，此时触发新生代的GC，G1就会依然用复制算法来进行垃圾回收，进入一个“Stop the World”状态，然后把Eden对应的Region中的存活对象复制到S0对应的Region中，接着回收掉Eden对应的Region中的垃圾对象；

但这个过程与之前是有区别的，因为G1是可以设定目标GC停顿时间的，也就是G1执行GC的时候最多可以让系统停顿多长时间，可以通过“-XX:MaxGCPauseMills”参数来设定，默认值是200ms，那么G1就会通过对每个Region追踪回收它需要多少时间，可以回收多少对象来选择回收一部分Region，保证GC停顿时间控制在指定范围内，尽可能多地回收对象；

#### G1垃圾收集器的老年代垃圾回收

1、初始标记，需要Stop the World，不过仅仅标记一下GC Roots直接能引用的对象，这个过程速度很快，而且是借用进行Minor GC的时候同步完成的，所以G1收集器在这个阶段实际并没有额外的停顿；

2、并发标记，不需要Stop the World，这个阶段会从GC Roots开始追踪所有的存活对象，初始标记阶段仅仅只是标记GC Roots直接关联的对象，而在并发标记阶段，就会进行GC Roots追踪，从这个GC Root对象直接关联的对象开始往下追踪，追踪全部的存活对象，这个阶段是很耗时的，但可以和系统程序并发运行，所以对系统程序的影响不大；

3、重新标记（最终标记），需要Stop the World，用户程序停止运行，最终标记一下有哪些存活对象，有哪些是垃圾对象；

4、筛选回收，需要Stop the World，对各个Region的回收价值和成本进行排序，根据用户所期望的停顿时间来制定回收计划，可以自由选择任意多个Region构成回收集，然后把决定回收的那一部分Region的存活对象复制到空的Region中，再清理掉整个旧Region的全部空间，这里的操作涉及存活对象的移动，是必须暂停用户线程，由多条收集器线程并行完成的；

从整体上看，G1垃圾收集像是一种标记-整理算法，它不存在内存碎片问题，实际上它是一种复制算法，G1收集器除了并发标记外，其余阶段也是要完全暂停用户线程的， 所以它并不是纯粹地追求低延迟，而是给它设定暂停目标，使其在延迟可控的情况下获得尽可能高的吞吐量；

#### G1垃圾收集器的混合垃圾回收？

混合垃圾收集即mixed gc，它不是一个old gc，除了回收整个young region，还会回收一部分的old region，是回收一部分老年代，而不是全部老年代，可以选择部分old region进行收集，从而可以对垃圾回收的耗时时间进行控制；

G1有一个参数，是“-XX:InitiatingHeapOccupancyPercent”，它的默认值是45%，即如果老年代占据了堆内存的45%的Region的时候，此时就会尝试触发一个新生代+老年代一起回收的混合回收阶段；

比如堆内存有2048个Region，如果老年代占据了其中45%的Region，也就是接近1000个Region的时候，就会开始触发一个混合回收；

#### G1回收失败时的Full GC

在进行Mixed回收的时候，无论是年轻代还是老年代都基于复制算法进行回收，把各个Region中存活的对象复制到其他空闲的Region中；

如果万一出现复制时没有空闲Region可以存放存活对象了，就会停止系统程序，然后采用单线程进行标记清除和压缩整理，空闲出来一批Region，这个过程很慢；

与CMS中的“Concurrent Mode Failure”失败会导致Full GC类似，如果内存回收的速度赶不上内存分配的速度，G1收集器也要被迫暂停用户线程，导致Full GC而产生长时间Stop The World；

1、新生代垃圾回收

2、老年代垃圾回收

3、混合回收

4、Full GC

#### 什么时候使用G1垃圾收集器？

-XX:+UseG1GC

1. 针对大内存、多处理器的机器推荐采用G1垃圾收集器，比如堆大小至少6G或以上；
2. 超过50%的堆空间都被活动数据占用；

3、在要求低延迟的场景，也就是GC导致的程序暂停时间要比较少，0.5-1秒之间；

4、对象在堆中分配频率或者年代升级频率变化比较大，防止高并发下应用雪崩现象的场景；

### ZGC收集器

-XX:+UseZGC

ZGC（Z Garbage Collector）是一款由Oracle公司研发的，以低延迟为首要目标的一款垃圾收集器，它是基于动态Region内存布局，（暂时）不设年龄分代，使用了读屏障、染色指针和内存多重映射等技术来实现可并发的标记-整理算法的收集器，在JDK 11新加入，还在实验阶段，主要特点是：回收TB级内存（最大4T），停顿时间不超过10ms；

### Shenandoah收集器

**Shenandoah**作为第一款不由Oracle(包括Sun)公司的虚拟机团队所领导开发的HotSpot垃圾收集器，Oracle官方明确拒绝在 OracleJDK12 中支持 Shenandoah 收集器，它是一款只有OpenJDK才会包含的收集器，最初由RedHat公司创建的，在2014年的时候贡献给了OpenJDK，Shenandoah收集器能实现在任何堆内存大小下都把垃圾停顿时间限制在10ms以内；