第3章 垃圾回收器和内存分配策略

3.1 概述

==1960年==诞生==第一门真正使用内存动态分配和垃圾回收技术的语言Lisp==。

由于程序计数器、虚拟机栈、本地方法栈是线程私有的，当线程结束就会被回收，同时虚拟机栈和本地方法栈在方法结束时，内存也会被回收，因此我们只用考虑==Java堆==和==方法区==的内存分配与回收问题。

3.2 对象已死吗？

3.2.1 引用计数法

引用计数法：==给对象中添加一个引用计数器，每当一个地方引用它时，计数器值加1，当引用失效时，计数器值减1，任何计数器为0的对象就是不可能再被使用的。==

优点：==实现简单、效率高==。

缺点：==很难解决对象之间的相互之间的循环引用问题==。例如对象A中一个引用指向B，对象B中的一个引用指向A，现在将A、B均指向空，由于AB之间存在相互指向的情况，两者的引用计数器均不为0，但是已经没有有效的引用可以访问到A和B了。

3.2.2 可达性分析算法

可达性分析算法：==通过一系列“GC Roots”的对象作为起始点，从这些点向下搜索，搜索所走过的链称为引用链。当一个对象到达GC Roots不存在任何一条引用链，我们称其不可用==。

==GC Roots可以取如下对象==：

1. 虚拟机栈（栈帧中的本地变量表）中的引用对象。
2. 方法区中类静态属性引用的对象。
3. 方法区中常量引用对象。
4. 本地方法栈中JNI（即一般说的Native方法）引用的对象。

3.2.3 再谈引用

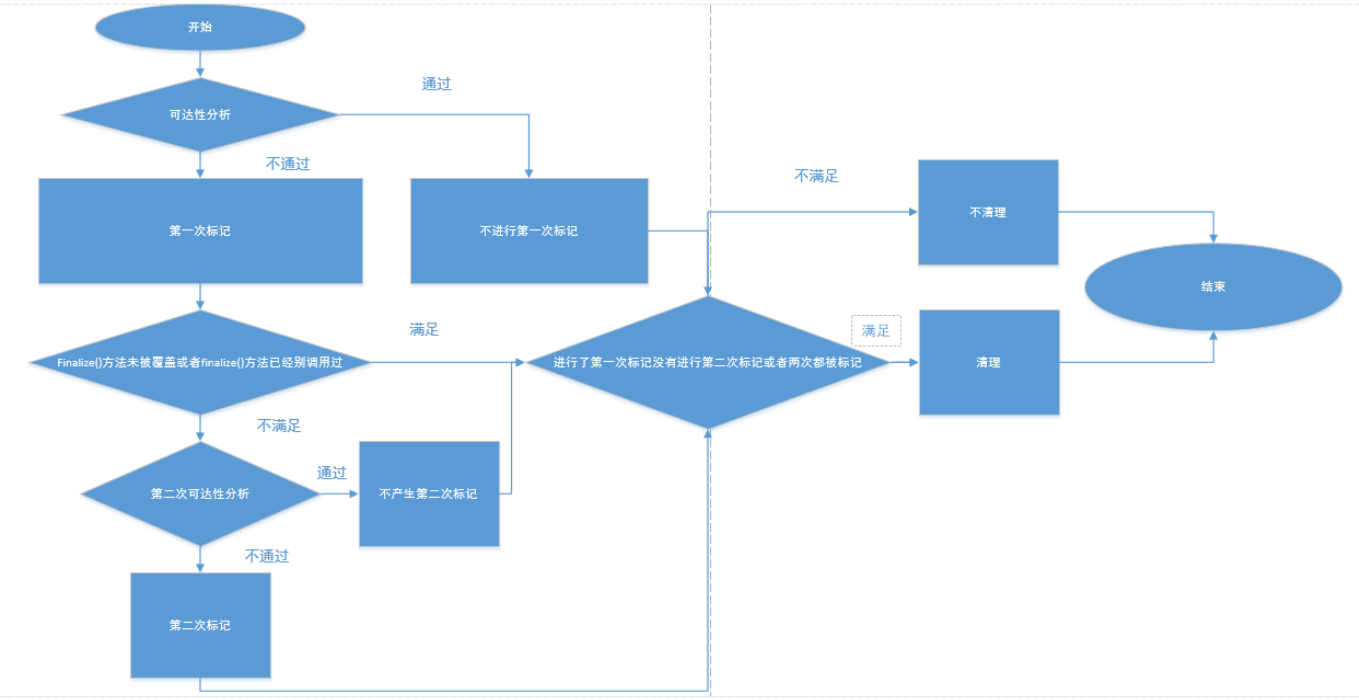
在JDK1.2后，Java对引用的概念进行了扩充，将引用分为强引用、软引用、弱引用、虚引用。这四种引用强度依次递减。

1. 强引用是在代码中普遍存在的，类似“Object o = new Object();”的引用，只要强引用存在，对象就不会被回收。
2. 软引用是用来描述一系列还有用但并非必要的对象。 ==在系统将要发生内存溢出异常之前，将会把被弱引用相连的对象列进回收范围之中进行第二次回收，如果仍然没有足够的内存，则抛出内存溢出异常==。
3. 弱引用也是描述非必要对象的，但是强度比软引用弱一些。==被弱引用关联的对象只能生存岛下一次垃圾收集发生之前==。当垃圾收集器工作时，无论内存是否足够，都会回收掉只被弱引用关联的对象。
4. 虚引用也成为幽灵引用或者幻影引用，它是最弱的一种引用关系。==一个对象是否有虚引用存在完全不会对其生存时间构成影响，也无法通过虚引用取得一个对象实例==。==为一个对象设置虚引用的关联

的唯一目的就是能在这个对象被收集器收集时收到一个系统通知==。

3.2.4 生存还是死亡

对象被销毁的过程：



3.2.5 回收方法区

Java虚拟机规范中没有明确要求要在方法区中进行垃圾回收，甚至可以不实现，==在方法区中进行垃圾回收的“性价比”一般比较低==。

永久代的垃圾回收主要回收==废弃常量==和==无用的类==。当没有任何对象引用了该常量，那么在必要情况下，该常量就会被回收。“无用的类”的要求比较苛刻：

1. ==该类的所有实例均被回收==，Java堆中不存在该类的实例。
2. ==加载该类的ClassLoader已经被回收==。
3. ==该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法==。

HotSpot 虚拟机提供了-Xnoclassgc参数进行控制，还可以使用-verbose:class以及-XX:+TraceClassLoading、-XX:+TraceClassUnLoading查看类加载和卸载信息，其中-verbose:class和-XX:+TraceClassLoading可以在Product版的虚拟机中使用，-XX:+TraceClassUnLoading参数需要FastDebug版的虚拟机支持。

在大量使用反射、动态代理、CGLib等ByteCode框架、动态生成JSP以及OSGi这类频繁自定义ClassLoader的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

3.3 垃圾清除算法

白色的空间表示未使用、粉色的表示可回收、绿色的表示存货对象、黑色的表示保留空间。

3.3.1 标记-清除算法

标记-清除算法分为“标记”、“清除”两部分：==首先标记处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。==

缺点：

1. 效率问题，==标记和清除两个过程的效率都不高==；
2. 空间问题，==产生大量不连续的内存碎片==。

清理前：

清理后：

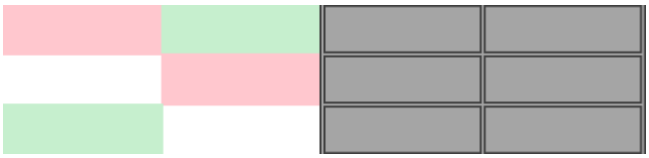
3.3.2 复制算法

==复制算法将内存分为大小相同的两块，每次只使用其中的一块，当这块的内存用完了，就将还存活着的对象复制到另一块上面，然后再把已使用过的内存空间一次性清理掉。==

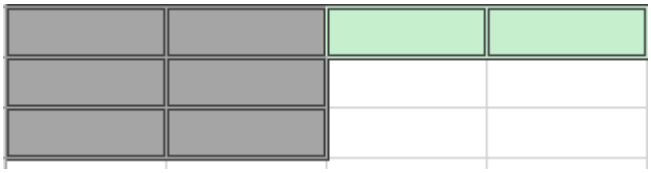
优点：不会出现内存碎片、实现简单、运行高效。

缺点：==减少一半的内存，开销太大==。

清理前：



清理后：



但是由于资源浪费太严重，新生代98%是“朝生夕死”，所以不需要1:1的比例划分内存空间，而==将内存分为一块较大的Eden空间和两块较小的Survivor空间，每次只使用Eden空间和其中的一块。当回收时，将Eden和Survivor空间中存活的对象复制到另一块中。然后清理Eden和刚才那块Survivor空间。==

HotSpot虚拟机==默认的Eden和Survivor的比例为8:1。==

3.3.3 标记-整理算法

标记-整理算法将所有可回收的空间进行标记，然后将标记后的空间进行清理并且整理在一起。

优点：不产生内存碎片。

缺点：效率低。

回收前状态：

回收后状态：

3.3.4 分代收集算法

按对象存活周期分为几块，一般分为新生代和老年代，新生代的对象大部分都是朝生夕死的，老年代则不同，然后对这两个年代的内存采用不同的收集方式。

3.4 HotSpot的算法实现

3.4.1 枚举根节点

在可达性分析过程中，要逐个检查这里面的引用，检查过程中必然会消耗一定时间，但是在这段时间里要在确保一致性的快照中进行，因此，进行GC停顿，进行GC时必须停顿所有Java执行线程（Stop The World）。

主流Java虚拟机使用的都是准确式GC，不需要一个不漏的检查完所有空间。虚拟机使用一组称为OopMap的数据结构实现直接得知哪些地方存放着对象引用的作用。

3.4.2 安全点

如果为每一条指令都生成对应的OopMap，但是会需要大量的额外空间。因此我们只在特定的地方记录这些信息，这些位置称为安全点，即程序只有在达到安全点才能停顿开始GC。

GC发生时让所有线程全部跑到最近的安全点再停顿下来，这里有两种方案可供选择：

1. 抢先式中断。在GC发生时，首先把所有线程全部中断，如果发现线程中断的地方不在安全点上，就恢复线程，让它“跑”到安全点上。
2. 主动式中断。当GC需要中断线程时，不直接对线程进行操作，仅仅简单地设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就将自己挂起，轮询标志的地方和安全点是重合的，另外再加上创建对象需要分配内存的地方。

3.4.3 安全区域

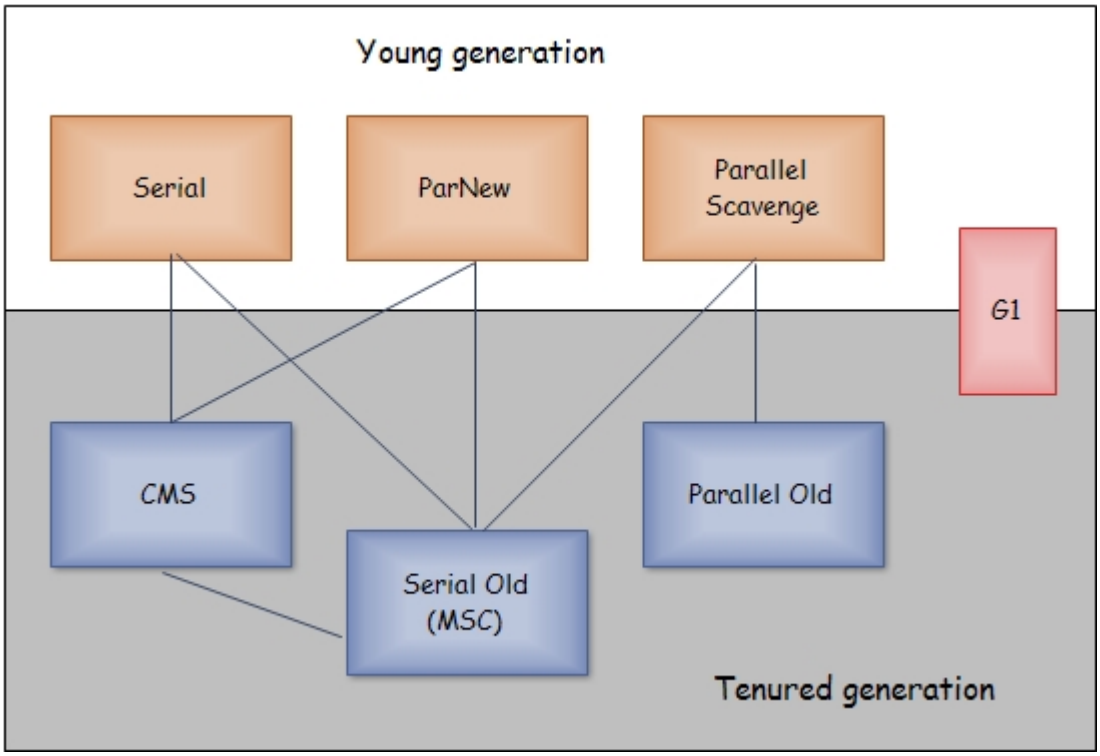
Safepoint机制保证了程序执行时，在不太长的时间内就会遇到可进入GC的Safepoint。但是程序不执行时呢，即没有分配CPU时间，例如线程处于sleep状态或者Blocked状态。

安全区域是指在一段代码片段中，引用关系不会发生变化。在这个区域中的任何地方开始GC都是安全的。

在线程执行到Safe Region中的代码时，首先标识自己已经进入Safe Region，当在这段时间内JVM发起GC时，就不用管标识自己是Safe Region状态的线程了。当线程要离开Safe Region时，它要检查系统是否已经完成了根节点枚举（或者是整个GC过程），完成，则线程继续执行，否则，等待知道收到完成信号再退出。

3.5 垃圾回收器

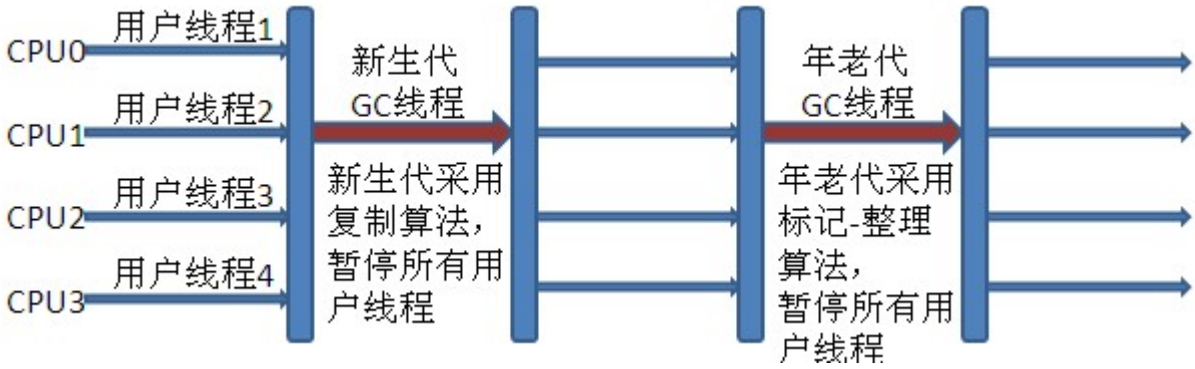
HotSpot虚拟机中的垃圾回收器：



3.5.1 Serial收集器

Serial收集器是一个单线程收集器，使用复制算法，它进行垃圾收集时，只会使用一个CPU或一条线程完成垃圾收集工作，必须暂停其他所有工作线程，直到它收集结束。

Serial/Serial Old收集器运行示意图：

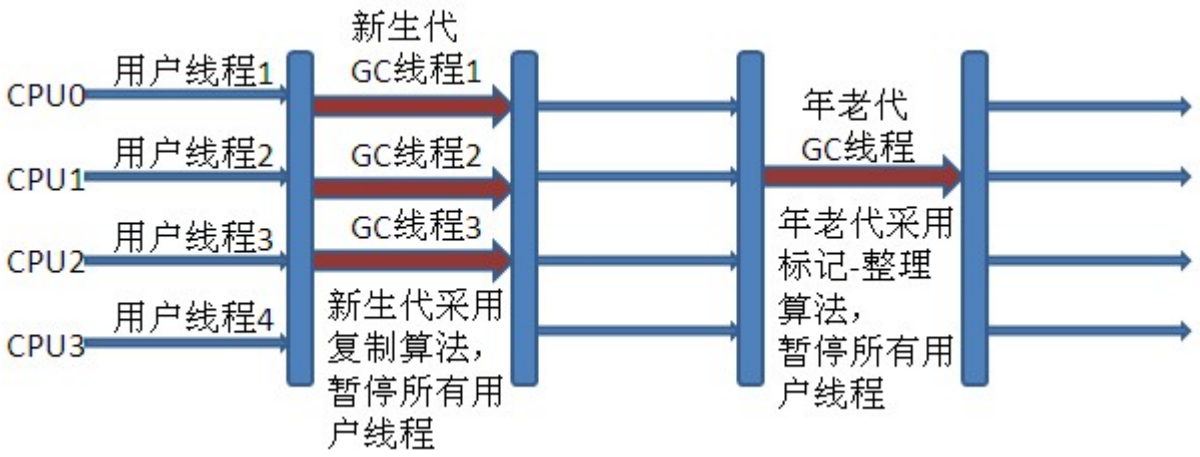


是Jvm client模式下默认的新生代收集器。对于限定单个CPU的环境来说，简单高效，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率，因此是运行在Client模式下的虚拟机的不错选择（比如桌面应用场景）。

3.5.2 ParNew 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，其余行为包括Serial收集器可用的所有参数、收集算法、Stop The World、对象分配规则、回收策略等都和Serial完全相同。

ParNew/Serial Old收集器运行示意图：



ParNew是运行在Server模式下的首选新生代收集器，除了Serial收集器外，只有它能与CMS收集器配合工作。ParNew收集器在单CPU的环境中绝不会有比Serial更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个CPU环境中都不能百分之百保证超过Serial收集器。默认开启的线程收集线程数和CPU的数量相同，可以通过-XX:ParallelGCThreads参数限制垃圾收集的线程数。

3.5.3 Parallel Scavenge收集器

Parallel Scavenge收集器是一个新生代收集器，使用复制算法，是并行多线程的收集器。Parallel Scavenge收集器的目的是达到一个可控制的吞吐量。吞吐量即CPU用于运行用户代码的时间与CPU总消耗时间的比值，吞吐量=运行用户代码时间/（用户运行代码时间+垃圾收集时间）。主要适合在后台运算而不需要太多交互的任务。

-XX:MaxGCPauseMillis参数控制最大垃圾收集停顿时间。 -XX:GCTimeRatio参数用来直接设置吞吐量大小。

GC停顿时间的减少是以牺牲吞吐量和新生代空间来换取的。

-XX：+UseAdaptiveSizePolicy参数设置虚拟机是否根据当前系统运行情况自动调节新生代大小、Eden和Survivor区比例、晋升老年代对象大小等细节参数。这种调节方式称为GC自适应调节策略。

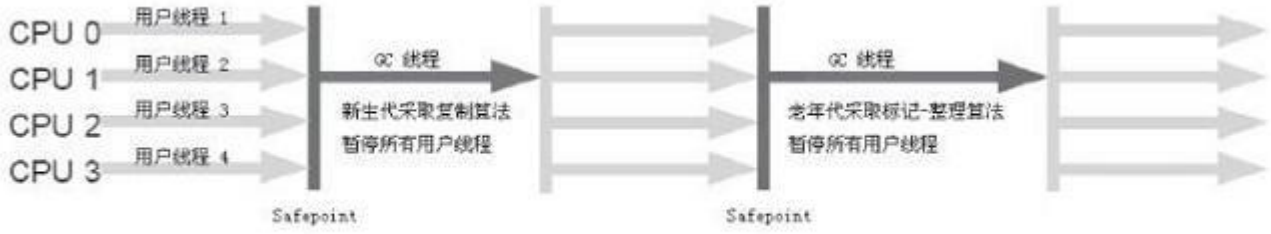
自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个重要区域。

3.5.4 Serial Old 收集器

Serial Old收集器是Serial收集器的老年代版本，单线程，使用标记整理算法，Client模式下的虚拟机使用。Server模式下有2大用途：

1. 在JDK1.5以及以前的版本与Parallel Scavenge收集器搭配使用；
2. 作为CMS的后备预案，在Concurrent Mode Failure时使用。

Serial Old 收集器工作过程：



Serial / Serial Old 收集器运行示意图

3.5.5 Parallel Old 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。

在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器。

其工作过程如图：

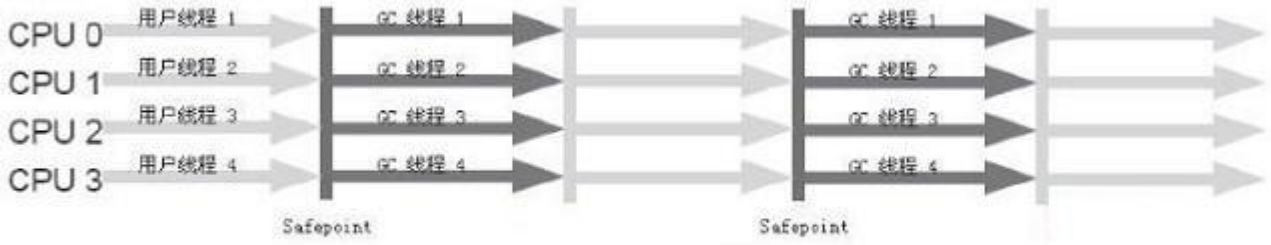


图 3-9 Parallel Scavenge / Parallel Old 收集器运行示意图

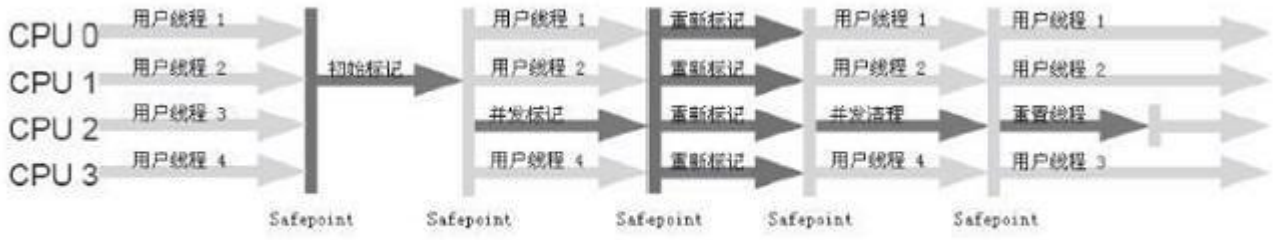
3.5.6 CMS收集器

CMS收集器是一种以获取最短回收停顿时间为目标的收集器，使用“标记-清除”算法，运作过程分为四个步骤：

1. 初始标记
2. 并发标记
3. 重新标记
4. 并发清除

其中，初始标记和重新标记仍需“Stop The World”，其中，初始标记仅仅是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，而重新标记则是为了修正并发标记期间因用户程序继续运作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一般会比初始标记阶段稍长些，但是比并发标记的时间短。

由于CMS收集器中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程工作，所以CMS收集器的内存回收过程是和用户线程一起并发执行的，工作流程如下：



Concurrent Mark Sweep 收集器运行示意图

优点：并发收集、低停顿

缺点：

1. CMS收集器对CPU资源非常敏感。
2. CMS无法处理浮动垃圾。
3. CMS基于“标记-清除”算法，会产生大量内存碎片。

CMS收集器对CPU资源敏感，并发程序对CPU资源都比较敏感，由于并发阶段会占用部分CPU资源，导致用户进行变慢，总吞吐量降低，CMS默认启动的回收线程数为（CPU数量+3）/4，那么占用CPU资源为(1/4+3/(4*CPU数量)))，随着CPU数量增多而下降，但总不低于1/4。

CMS无法处理浮动垃圾，由于CMS并发清理阶段用户线程还在运行，期间伴随垃圾产生，这部分垃圾无法被标记，无法处理，这部分垃圾就称为“浮动垃圾”。

3.5.7 G1收集器

G1收集器是面向服务端应用的垃圾收集器。具备如下特点：

1. 并行与并发，G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短STW的时间，部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让Java程序继续执行。
2. 分带收集
3. 空间整合，G1从整体来看是基于“标记-整理”算法，从局部看，使用的是“复制”算法。
4. 可预测的停顿

G1收集器，它将整个Java堆分为多个大小相等的独立区域，虽然还保留新生代和老年代的概念但新生代和老年代不再是物理隔离，他们都是一部分Region的集合。

G1可以有计划地避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的价值大小，在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region。这保证了G1能建立可预测的停顿时间模型。

G1收集器中，虚拟机使用Remembered Set来避免全堆扫描。

如不计算维护Remembered Set的操作，G1收集器的运作大致可分为如下几个步骤：

1. 初始标记
2. 并发标记
3. 最终标记
4. 筛选回收

初始标记仅仅只是标记一下GC Roots能直接关联到的对象，并且修改TAMS的值(Next Top at Mark Start)，让下一阶段用户程序并发运行时，能在正确可用的Region中创建新对象，这阶段需要停顿线程，但耗时很短。

并发标记阶段是从GC Roots开始对堆中的对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可以并发执行。

最终标记阶段是为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程Remembered Set Logs里面，最终标记阶段需要把Remembered Set Logs的数据合并到Remembered Set中，这阶段需要线程停顿，但是可以并行执行。

最后在筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来执行回收计划。

G1收集器运行过程如下：

G1 收集器运行示意图

3.5.8 理解GC日志

```
33.125: [GC [DefNew: 3324K->152K(3712K),0.0025925 secs]
3324K->152K(11904K),0.0031680 secs]

100.667: [Full GC [Tenured: 0K->210K(10240K),0.0149142 secs]
4603K->210K(19456K),[Perm: 2999K->2999K(21248K)],0.0150007 secs]
[Times: user=0.01 sys=0.00,real:0.02 secs]
```

最前面的数字表示 GC发生的时间，这个数字的含义是从Java虚拟机启动以来经过的秒数。

GC和Full GC说明这次垃圾回收的停顿类型，带有Full证明产生了STW。

接下来的[DefNew、[Tenured、[Perm表示GC发生的区域。

后面的"3324K->152K(3712K)"含义是"GC前该内存区域已使用的容量->GC后该内存区域已使用的容量(该内存区域总容量)"。

最后的0.0025925 secs表明该内存区域GC所占用的时间，单位是秒。有的收集器给出更详细的数据[Times:user = 0.01 sys=0.00 real = 0.02 secs]，user、sys、real分别代表用户态消耗的CPU时间、内核态消耗的CPU时间、和操作从开始到结束所经过的墙钟时间。

3.5.9 垃圾收集器参数总结

3.6 内存分配和回收策略

3.6.1 对象优先在**Eden**分配

3.6.2 大对象直接进入老年代

3.6.3 长期存活的对象将进入老年代

3.6.4 动态对象年龄判定

3.6.5 空间分配担保