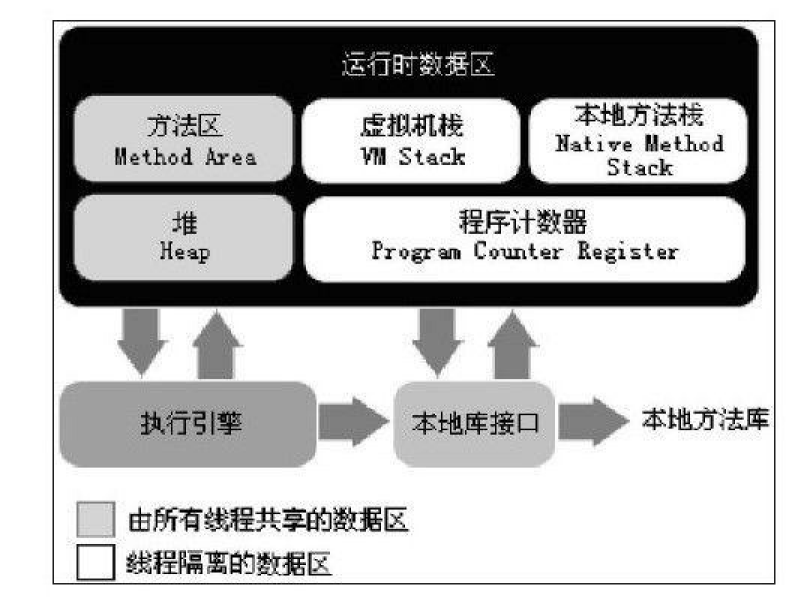
JVM学习笔记



JVM运行时数据区的调用关系

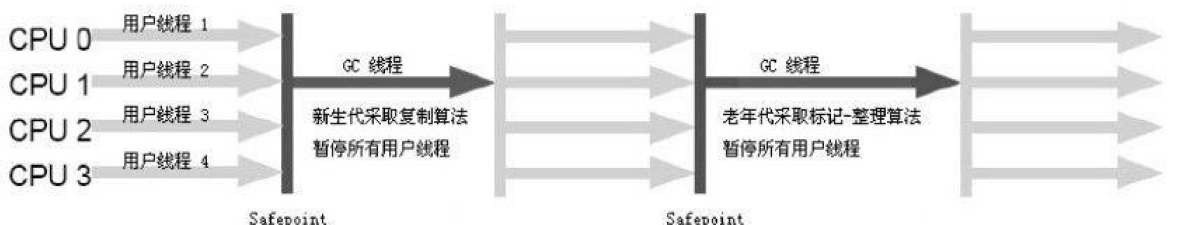
CAS （compare and swap） 如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。

垃圾收集器（GC）

Serial收集器（JDK 1.3.1之前是唯一新生代的收集选择）

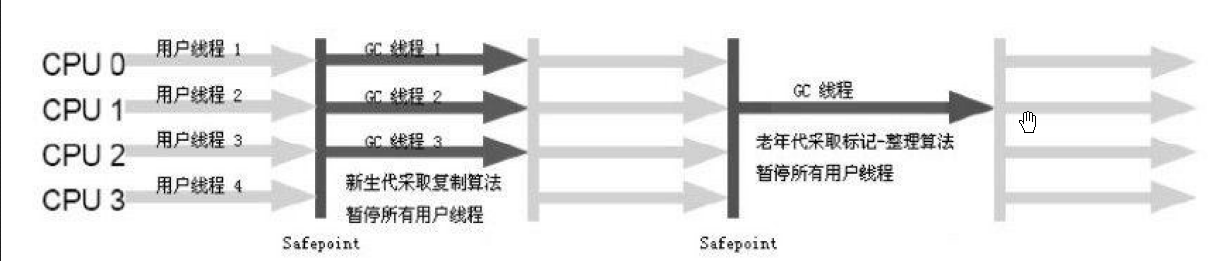
是一个单线程的收集器,在收集线程的过程中，会暂停所有其他工作线程，直到它收集结束。“Stop The World”暂停其他线程是虚拟机在后台自动发起和自动完成的，在其工作中用户是不可见的。它是虚拟机运行在Client模式下默认的新生代收集器，优点是简单而高效（与其他收集器的单线程比）对于限定单个CPU的环境来说，Serial收集器不会再线程交互上有开销，只要停顿时间不是很频繁发生，在Client模式下是一个比较好的选择。

工作过程如下：



ParNew收集器

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多线程进行垃圾收集之外，其余行为几乎一样。它在Server模式下的虚拟机中以新生代收集器为首选，ParNew和Serial能支持CMS, ParNew收集器工作过程如下图



Parallel Scavenge 收集器

Parallel Scavenge 是一个新生代收集器，在ParNew的基础上，Parallel Scavenge 收集器的目标是达到一个可控制的吞吐量 ，Parallel Scavenge收集器提供两个参数用户精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis 以及吞吐量大小的 –XX：GCTimeRatio

MaxGCPauseMillis 参数允许的值是一个大于

吞吐量公式

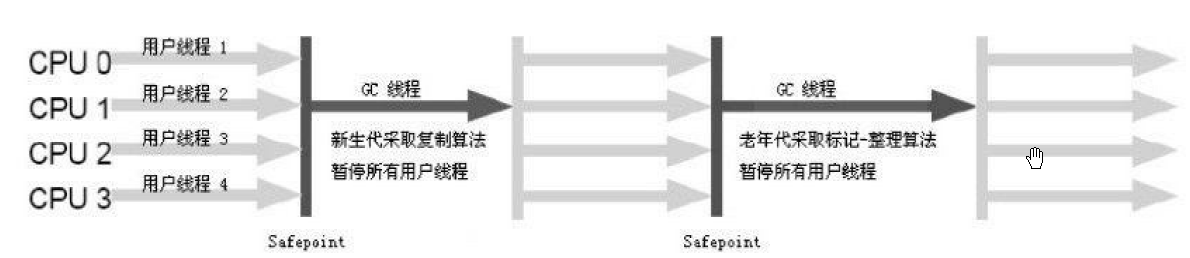
**吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）**

**如 99%=99min/100min**

Serial Old收集器

Serial Old是Serial的收集器的老年代版本，同样是单线程收集器，使用“标记-整理”算法。 这个收集器**在于给Client模式下的虚拟机使用**。如果用在Server模式下，name它主要用途是：JDK1.5以及之前的版本用来与Parallel Scavenge收集器搭配使用，另一种用途就是作为CMS收集器的后备预案 ，在并发收集发生Concurrent Mode Failure时使用

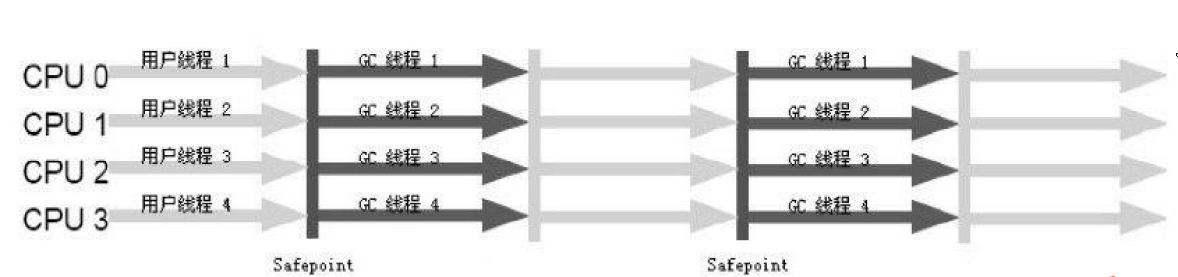
流程图如图



**Parallel Old收集器**

**是Parallel Scavenge 收集器的老年代版本，使用多线程和“标记-整理”算法。JDK1.6才开始提供的，Parallel Old 能与CMS组合使用，其组合使得“吞吐量优先”收集器终于有了比较名副其实的应用组合，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**

**工作过程如图**



**CMS收集器**

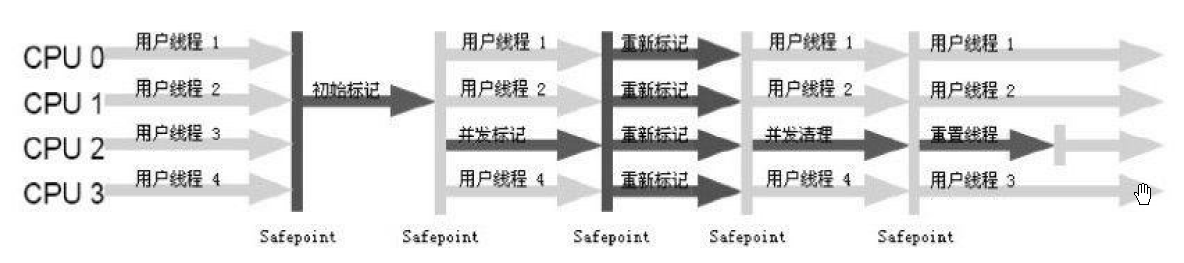
**CMS收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上。**

**CMS收集器是基于“标记—清除”算法实现的，它的运作过程细分为4个步骤：**

**初始标记（CMS initial mark）、并发标记（CMS concurrent mark）、重新标记（CMS remark）、并发清除（CMS concurrent sweep）**

**其中，初始标记、重新标记两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到对象，速度很快，并发标记阶段就是进行GC RootsTracing的过程，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操作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一般会比初始标记阶段稍长一些，但远比并发标记的时间短。**

**由于整个过程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一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。通过下图可以比较清除地看到CMS收集器的运作步骤中并发和需要停顿的时间。**



**CMS收集器的优点：并发收集，低停顿。**

**缺点：对CPU资源非常敏感， 无法处理浮动垃圾（Floating Garbage），收集结束时会有大量空间碎片产生。**

**1、CMS收集器对CPU资源非常敏感。在面向并发涉及的程序都对CPU资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用一部分线程而导致应用程序变慢，总吞吐量降低。CMS默认启动的回收线程数是（CPU数量+3）/4，也就是当CPU在4个以上时，并发回收垃圾收集线程不少于25%的CPU资源，并且随着CPU数量的增加而下降。但是当CPU不足4个时，CMS对用户程序的影响就可能变得很大，如果本来CPU负荷就比较大，还要分一半运算能力去执行收集器线程，就会导致用户速度忽然降低了50%。为了应付这种情况虚拟机提供了一种称为“增量式并发收集器”（i-CMS）的变种CMS收集器，它是在并发标记、清理的时候让GC线程、用户线程交替运行，尽量减少GC线程的独占资源的时间，这样整个垃圾收集的过程会更长，但对用户的影响就会显得少一些，也就是速度下降没有那么明显。增量时的CMS收集器效果一般，在目前版本中，i-CMS已经被声明为“deprecated”，不再提倡用户使用。**

**2、CMS收集器无法处理浮动垃圾（floating Garbage）,可能出现“Concurrent Mode Failure”失败导致另一次Full GC的产生。 浮动垃圾是伴随程序运行自然就还会有新的垃圾不断产生，这些垃圾在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。CMS收集器不能像其他收集器那样等老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在JDK1.5的默认设置下，CMS收集器当老年代使用了68%的空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数-XX:CMSInitatingOccupancyFraction的值来提高触发百分比，以降低内存回收次数从而获取更好的性能，在JDK1.6中，CMS收集器的启动阈值已经提升至92%。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这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、结束时会产生大量空间碎片，因为CMS是一款基于“标记——清除”算法实现的收集器，这就意味着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的麻烦，往往会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(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。虚拟机设计者还提供了另一个参数 –XX: CMSFullGCsBeforeCompaction,这个参数适用于设置执行多少次不压缩的Full GC后，跟着来一次带压碎的(默认值为0，表示每次进入Full GC时都要进行碎片整理)。**

**G1收集器**

**G1收集器是当年收集器技术发展的最前沿成果之一，在JDK1.7确立项目目标，Sun公司给出的JDK1.7 RoadMap里面，它就被视为JDK1.7中HotSpot虚拟机的一个重要进化特征。从JDK 6u14中开始就有Early Access版本的G1收集器供开发者实现，直至JDK 7u4，Sun公司才认为它达到足够成熟的商用程度。**