**[深入理解JVM(五)——HotSpot垃圾收集器详解](https://blog.csdn.net/u010425776/article/details/51199767)**

<https://blog.csdn.net/u010425776/article/details/51199767>

HotSpot虚拟机提供了多种垃圾收集器，每种收集器都有各自的特点，没有最好的垃圾收集器，只有最适合的垃圾收集器。我们可以根据自己实际的应用需求选择最适合的垃圾收集器。

根据新生代和老年代各自的特点，我们应该分别为它们选择不同的收集器，以提升垃圾回收效率。 

**新生代垃圾收集器**

**1. Serial垃圾收集器**

1. 单线程   
   只开启一条GC线程进行垃圾回收，并且在垃圾回收过程中停止一切用户线程，从而用户的请求或图形化界面会出现卡顿。
2. 适合客户端应用   
   一般客户端应用所需内存较小，不会创建太多的对象，而且堆内存不大，因此垃圾回收时间比较短，即使在这段时间停止一切用户线程，用户也不会感受到明显的停顿，因此本垃圾收集器适合客户端应用。
3. 简单高效   
   由于Serial收集器只有一条GC线程，因此避免了线程切换的开销，从而简单高效。
4. 采用“复制”算法

**2. ParNew垃圾收集器**

ParNew是Serial的多线程版本。   
1. 多线程并行执行   
ParNew由多条GC线程并行地进行垃圾清理。但清理过程仍然需要停止一切用户线程。但由于有多条GC线程同时清理，清理速度比Serial有一定的提升。

1. 适合多CPU的服务器环境   
   由于使用了多线程，因此适合CPU较多的服务器环境。
2. 与Serial性能对比   
   ParNew和Serial唯一的区别就是使用了多线程进行垃圾回收，在多CPU的环境下性能比Serial会有一定程度的提升；但线程切换需要额外的开销，因此在单CPU环境中表现不如Serial。
3. 采用“复制”算法
4. 追求“降低停顿时间”   
   和Serial相比，ParNew使用多线程的目的就是缩短垃圾收集时间，从而减少用户线程被停顿的时间。

**3. Parallel Scavenge垃圾收集器**

Parallel Scavenge和ParNew一样都是多线程、新生代收集器，都使用“复制”算法进行垃圾回收。但它们有个巨大的不同点：ParNew收集器追求降低用户线程的停顿时间，因此适合交互式应用；而Parallel Scavenge追求CPU吞吐量，能够在较短的时间内完成指定任务，因此适合没有交互的后台计算。

1. 什么是“吞吐量”？   
   吞吐量是指用户线程运行时间占CPU总时间的比例。   
   CPU总时间包括：用户线程运行时间 和 GC线程运行的时间。   
   因此，吞吐量越高表示用户线程运行时间越长，从而用户线程能够被快速处理完。
2. 降低停顿时间的两种方式   
   1.在多CPU环境中使用多条GC线程，从而垃圾回收的时间减少，从而用户线程停顿的时间也减少；   
   2.实现GC线程与用户线程并发执行。所谓并发，就是用户线程与GC线程交替执行，从而每次停顿的时间会减少，用户感受到的停顿感降低，但线程之间不断切换意味着需要额外的开销，从而垃圾回收和用户线程的总时间将会延长。
3. Parallel Scavenge提供的参数
   * 设置“吞吐量”   
     通过参数-XX:GCTimeRadio设置垃圾回收时间占总CPU时间的百分比。
   * 设置“停顿时间”   
     通过参数-XX:MaxGCPauseMillis设置垃圾处理过程最久停顿时间。Parallel Scavenge会根据这个值的大小确定新生代的大小。如果这个值越小，新生代就会越小，从而收集器就能以较短的时间进行一次回收。但新生代变小后，回收的频率就会提高，因此要合理控制这个值。
   * 启用自适应调节策略   
     通过命令-XX:+UseAdaptiveSizePolicy就能开启自适应策略。我们只要设置好堆的大小和MaxGCPauseMillis或GCTimeRadio，收集器会自动调整新生代的大小、Eden和Survior的比例、对象进入老年代的年龄，以最大程度上接近我们设置的MaxGCPauseMillis或GCTimeRadio。

**老年代垃圾收集器**

**1. Serial Old垃圾收集器**

Serial Old收集器是Serial的老年代版本，它们都是单线程收集器，也就是垃圾收集时只启动一条GC线程，因此都适合客户端应用。

它们唯一的区别就是Serial Old工作在老年代，使用“标记-整理”算法；而Serial工作在新生代，使用“复制”算法。 

**2. Parallel Old垃圾收集器**

Parallel Old收集器是Parallel Scavenge的老年代版本，一般它们搭配使用，追求CPU吞吐量。   
它们在垃圾收集时都是由多条GC线程并行执行，并停止一切用户线程。因此，由于在垃圾清理过程中没有使垃圾收集和用户线程并行执行，因此它们是追求吞吐量的垃圾收集器。 

**3. CMS垃圾收集器**

CMS收集器是一款追求停顿时间的老年代收集器，它在垃圾收集时使得用户线程和GC线程并行执行，因此在垃圾收集过程中用户也不会感受到明显的卡顿。但用户线程和GC线程之间不停地切换会有额外的开销，因此垃圾回收总时间就会被延长。

**垃圾回收过程**

1. 初始标记   
   停止一切用户线程，仅使用一条初始标记线程对所有与GC ROOTS直接关联的对象进行标记。速度很快。
2. 并发标记   
   使用多条并发标记线程并行执行，并与用户线程并发执行。此过程进行可达性分析，标记出所有废弃的对象。速度很慢。
3. 重新标记   
   停止一切用户线程，并使用多条重新标记线程并行执行，将刚才并发标记过程中新出现的废弃对象标记出来。这个过程的运行时间介于初始标记和并发标记之间。
4. 并发清除   
   只使用一条并发清除线程，和用户线程们并发执行，清除刚才标记的对象。这个过程非常耗时。

**CMS的缺点**

1. 吞吐量低   
   由于CMS在垃圾收集过程使用用户线程和GC线程并行执行，从而线程切换会有额外开销，因此CPU吞吐量就不如在垃圾收集过程中停止一切用户线程的方式来的高。
2. 无法处理浮动垃圾，导致频繁Full GC   
   由于垃圾清除过程中，用户线程和GC线程并发执行，也就是用户线程仍在执行，那么在执行过程中会产生垃圾，这些垃圾称为“浮动垃圾”。   
   如果CMS在垃圾清理过程中，用户线程需要在老年代中分配内存时发现空间不足时，就需要再次发起Full GC，而此时CMS正在进行清除工作，因此此时只能由Serial Old临时对老年代进行一次Full GC。
3. 使用“标记-清除”算法产生碎片空间   
   由于CMS使用了“标记-清除”算法， 因此清除之后会产生大量的碎片空间，不利于空间利用率。不过CMS提供了应对策略：   
   * 开启-XX:+UseCMSCompactAtFullCollection   
     开启该参数后，每次FullGC完成后都会进行一次内存压缩整理，将零散在各处的对象整理到一块儿。但每次都整理效率不高，因此提供了以下参数。
   * 设置参数-XX:CMSFullGCsBeforeCompaction   
     本参数告诉CMS，经过了N次Full GC过后再进行一次内存整理。

**通用垃圾收集器——G1垃圾收集器**

G1是目前最牛逼的垃圾收集器。

**G1的特点**

1. 追求停顿时间
2. 多线程GC
3. 面向服务端应用
4. 标记-整理和复制算法合并   
   不会产生碎片内存。
5. 可对整个堆进行垃圾回收
6. 可预测停顿时间

**G1的内存模型**

G1垃圾收集器没有新生代和老年代的概念了，而是将堆划分为一块块独立的Region。当要进行垃圾收集时，首先估计每个Region中的垃圾数量，每次都从垃圾回收价值最大的Region开始回收，因此可以获得最大的回收效率。

**Remembered Set**

一个对象和它内部所引用的对象可能不在同一个Region中，那么当垃圾回收时，是否需要扫描整个堆内存才能完整地进行一次可达性分析？

当然不是，每个Region都有一个Remembered Set，用于记录本区域中所有对象引用的对象所在的区域，从而在进行可达性分析时，只要在GC ROOTs中再加上Remembered Set即可防止对所有堆内存的遍历。

**G1垃圾收集过程**

1. 初始标记   
   标记与GC ROOTS直接关联的对象，停止所有用户线程，只启动一条初始标记线程，这个过程很快。
2. 并发标记   
   进行全面的可达性分析，开启一条并发标记线程与用户线程并行执行。这个过程比较长。
3. 最终标记   
   标记出并发标记过程中用户线程新产生的垃圾。停止所有用户线程，并使用多条最终标记线程并行执行。
4. 筛选回收   
   回收废弃的对象。此时也需要停止一切用户线程，并使用多条筛选回收线程并行执行。