**Java虚拟机9：垃圾收集（GC）-4（垃圾收集器）**

**1.前言**

垃圾收集器是前一章垃圾收集算法理论知识的具体实现了，不同虚拟机所提供的垃圾收集器可能会有很大差别，另外我们必须提前说明一个道理：没有最好的垃圾收集器，更加没有万能的收集器，只能选择对具体应用最合适的收集器。这也是HotSpot为什么要实现这么多收集器的原因，下面我们以HotSpot为例讲解。在写之前，先介绍几个概念。

**1.1.并行和并发的区别**

这个区别之前在你专门的一节介绍过，这里再重点提一下：这两个名词都是并发编程中的概念，在谈论垃圾收集器的上下文语境中，可以这么理解这两个名词：

**1、并行Parallel**

多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。

**2、并发Concurrent**

指用户线程与垃圾收集线程同时执行（但并不一定是并行的，可能会交替执行），用户程序在继续运行，而垃圾收集程序运行于另一个CPU上。

**1.2.Minor GC和Full GC的区别**

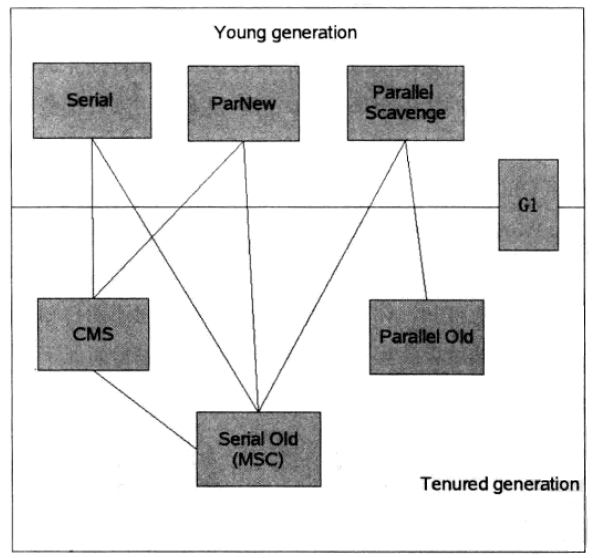
* **新生代GC（Minor GC）**：指发生在新生代的垃圾收集动作，因为Java对象大多都具备朝生夕灭的特性（存活率不高），所以Minor GC非常频繁，一般回收速度也比较快。
* **老年代GC（Major GC / Full GC）**：指发生在老年代的垃圾收集动作，出现了Major GC，经常会伴随至少一次的Minor GC（但非绝对的，在Parallel Scavenge收集器的收集策略里就有直接进行Major GC的策略选择过程）。Major GC的速度一般会比Minor GC慢10倍以上。

**1.3.吞吐量**

吞吐量：**就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即吞吐量 = 运行用户代码时间 /（运行用户代码时间 + 垃圾收集时间）。虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。**

**2.垃圾收集器组合**

 下面一张图是HotSpot虚拟机包含的所有收集器：



（A）：图中展示了**7种**不同分代的收集器：

       Serial、ParNew、Parallel Scavenge、Serial Old、Parallel Old、CMS、G1；

（B）：而它们所处区域，则表明其是属于新生代收集器还是老年代收集器：

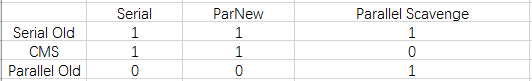
**新生代收集器**：Serial、ParNew、Parallel Scavenge；

**老年代收集器**：Serial Old、Parallel Old、CMS；

**整堆收集器**：G1；

（C）：两个收集器间有连线，表明它们**可以搭配使用**：

       Serial/Serial Old、Serial/CMS、ParNew/Serial Old、ParNew/CMS、Parallel Scavenge/Serial Old、Parallel Scavenge/Parallel Old、G1；



（D）：其中Serial Old作为CMS出现**"Concurrent Mode Failure"失败**的后备预案（后面介绍）；

**2.1 Serial收集器**

1. **特性:**  
   最基本、发展历史最久的收集器，**采用复制算法的单线程收集器**，**单线程一方面意味着它只会使用一个CPU或一条线程去完成垃圾收集工作，另一方面也意味着在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束为止，这个过程也称为 Stop The world。**后者意味着，在用户不可见的情况下要把用户正常工作的线程全部停掉，这显然对很多应用是难以接受的。
2. **应用场景：**  
   **Serial收集器依然是虚拟机运行在Client模式下的默认新生代收集器。**在用户的桌面应用场景中，可用内存一般不大（几十M至一两百M），可以在较短时间内完成垃圾收集（几十MS至一百多MS）,只要不频繁发生，这是可以接受的
3. **优势：**  
   **简单而高效**（与其他收集器的单线程相比），对于限定单个CPU的环境来说，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。比如在用户的桌面应用场景中，可用内存一般不大（几十M至一两百M），可以在较短时间内完成垃圾收集（几十MS至一百多MS）,只要不频繁发生，这是可以接受的。

**Stop TheWorld 说明：**

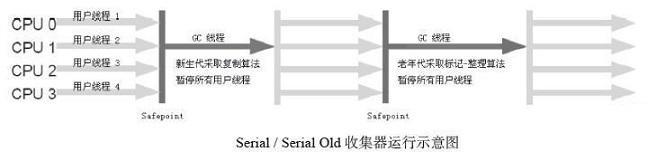
      GC在后台自动发起和自动完成的，在用户不可见的情况下，把用户正常的工作线程全部停掉，即**GC停顿，**会带给用户不良的体验；

      从JDK1.3到现在，从Serial收集器-》Parallel收集器-》CMS-》G1，用户线程停顿时间不断缩短，但仍然无法完全消除；

**设置参数**

**"-XX:+UseSerialGC"**：添加该参数来显式的使用串行垃圾收集器；

**Serial/Serial Old组合收集器运行示意图如下：**



**2.2 ParNew收集器**

1. **特性:**

**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外，其余行为和Serial收集器完全一样，包括Serial收集器可用的所有控制参数、收集算法、Stop The world、对象分配规则、回收策略等都一样。在实现上也共用了相当多的代码。

1. **应用场景：**  
   **ParNew收集器是许多运行在Server模式下的虚拟机中首选的新生代收集器。**很重要的原因是：除了Serial收集器之外，目前**只有它能与CMS收集器配合工作**（看图）。在JDK1.5时期，HotSpot推出了一款几乎可以认为具有划时代意义的垃圾收集器-----**CMS收集器**，这款收集器是HotSpot虚拟机中第一款真正意义上的并发收集器，它**第一次实现了让垃圾收集线程与用户线程同时工作**。
2. **优势：**  
   在单CPU中的环境中，不会比Serail收集器有更好的效果，因为存在线程交互开销，甚至由于线程交互的开销，该收集器在两个CPU的环境中都不能百分百保证可以超越Serial收集器。当然，随着可用CPU数量的增加，它对于GC时系统资源的有效利用还是很有好处的，它默认开启的收集线程数与CPU数量相同。

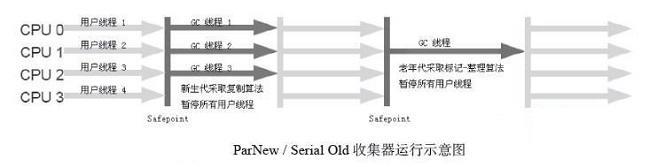
**设置参数：**

**"-XX:+UseConcMarkSweepGC"**：指定使用CMS后，会默认使用ParNew作为新生代收集器；

**"-XX:+UseParNewGC"**：强制指定使用ParNew；

**"-XX:ParallelGCThreads"**：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；

**ParNew/Serial Old组合收集器运行示意图如下：**



* 1. **Parallel Scavenge 收集器**

1.特性:

Parallel Scavenge收集器是一个**新生代收集器**，它也是使用**复制算法**的收集器，也是**并行的多线程收集器**。

2.对比分析：

* + **Parallel Scavenge收集器 VS CMS等收集器：**  
    Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是达到一个**可控制的吞吐量**（Throughput）。  
    由于与吞吐量关系密切，Parallel Scavenge收集器也经常称为“吞吐量优先”收集器。
  + **Parallel Scavenge收集器 VS ParNew收集器：**  
    Parallel Scavenge收集器与ParNew收集器的一个重要区别是它具有自适应调节策略。

3.应用场景：  
**Parallel Scavenge收集器是虚拟机运行在Server模式下的默认垃圾收集器。**  
停顿时间短适合需要与用户交互的程序，良好的响应速度能提升用户体验；高吞吐量则可以高效率利用CPU时间，尽快完成运算任务，主要适合在后台运算而不需要太多交互的任务。  
该收集器以高吞吐量为目标，就是减少垃圾收集时间，从而让用户代码获得更长的运行时间。所以适合那些运行在多个CPU上，并且专注于后台计算的应用程序，例如：执行批量处理任务、订单处理，工资支付，科学计算等。

**设置参数：**

虚拟机提供了-XX:MaxGCPauseMillis和-XX:GCTimeRatio两个参数来精确控制最大垃圾收集停顿时间和吞吐量大小。不过不要以为前者越小越好，GC停顿时间的缩短是以牺牲吞吐量和新生代空间换取的。

**"-XX:+MaxGCPauseMillis"**：控制最大垃圾收集停顿时间，大于0的毫秒数；这个参数设置的越小，停顿时间可能会缩短，但也会导致吞吐量下降，导致垃圾收集发生得更频繁。

**"-XX:GCTimeRatio"**：设置垃圾收集时间占总时间的比率，0<n<100的整数，就相当于设置吞吐量的大小。

                                          垃圾收集执行时间占应用程序执行时间的比例的计算方法是：

                                         1 / (1 + n)

                                          例如，选项-XX:GCTimeRatio=19，设置了垃圾收集时间占总时间的5%--1/(1+19)；

                                          默认值是1%--1/(1+99)，即n=99；

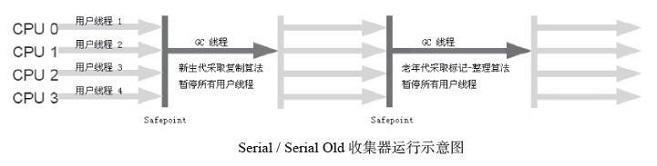
                                          垃圾收集所花费的时间是年轻一代和老年代收集的总时间；

**GC自适应的调节策略**：

Parallel Scavenge收集器有一个参数-XX:+UseAdaptiveSizePolicy。当这个参数打开之后，就不需要手工指定新生代的大小、Eden与Survivor区的比例、晋升老年代对象年龄等细节参数了，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或者最大的吞吐量，这种调节方式称为GC自适应的调节策略（GC Ergonomics）。**如果对于垃圾收集器运作原理不太了解，以至于在优化比较困难的时候，使用Parallel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将是一个不错的选择**。

**2.4 Serial Old收集器**

1. **特性**  
   Serial Old是**Serial收集器的老年代版本**，它同样是一个**单线程收集器**，使用**标记-整理算法**。
2. **应用场景**
   * **Client模式**  
     Serial Old收集器的主要意义也是在于给Client模式下的虚拟机使用。
   * **Server模式**  
     如果在Server模式下，那么它主要还有两大用途：一种用途是在JDK 1.5以及之前的版本中与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。

**Serial/Serial Old收集器运行示意图如下：**  


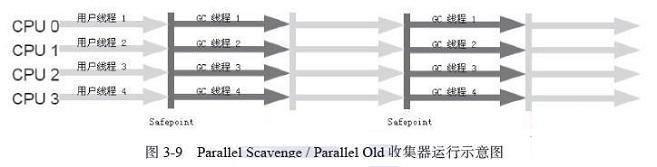
**2.5 Parallel Old收集器**

1. **特性：**  
   **Parallel 收集器的老年代版本，使用多线程和”标记-整理“算法。**
2. **应用场景：**  
   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。  
   这个收集器是在JDK1.6中才开始提供的，特别是在Server模式，多CPU的情况下。

**设置参数:**

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

**Parallel Scavenge/Parallel Old收集器运行示意图如下：**



**2.6 CMS收集器**

1. **特性：**  
   **CMS（Concurrent Mark Sweep）收集器：基于”标记-清除“算法实现的（不进行压缩，会产生内存碎片），特点是：并发收集，低停顿。**

是HotSpot在JDK1.5推出的第一款真正意义上的并发（Concurrent）收集器；第一次实现了让垃圾收集线程与用户线程（基本上）同时工作；

1. **应用场景**  
   与用户交互较多的场景。**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应用集中在互联网或者B/S系统的服务端上，这类应用尤其注重服务的响应速度，希望系统停顿时间最短，以给用户带来极好的体验。CMS收集器就非常符合这类应用的需求。
2. **运作过程：**  
   对于前面几种收集器来说更复杂一些，整个过程分为4个步骤：
   * **初始标记（CMS initial mark）**  
     初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，但需要“Stop The World”。
   * **并发标记（CMS concurrent mark）**  
     并发标记阶段就是进行GC Roots Tracing的过程，刚才产生的集合中标记出存活对象；应用程序也在运行；并不能保证可以标记出所有的存活对象；
   * **重新标记（CMS remark）**  
     重新标记阶段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；仍然需要需要”Stop The World“，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。
   * **并发清除（CMS concurrent sweep）**  
     并发清除阶段会清除对象，回收所有的垃圾对象。

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。

1. **缺点：**
   * **CMS收集器对CPU资源非常敏感**  
     其实，面向并发设计的程序都对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，但是会因为占用了一部分线程（或者说CPU资源）而导致应用程序变慢，总吞吐量会降低。  
     **CMS默认启动的回收线程数=（CPU数量+3）/ 4**，也就是当CPU在4个以上时，并发回收时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（譬如2个）时，CMS对用户程序的影响就可能变得更大，可能会无法接受。  
     了解：  
     增量式并发收集器：针对上述这种情况，曾出现了”增量式并发收集器“，类似使用抢占式来模拟多任务机制的思想，让收集线程和用户线程交替运行，减少收集线程运行时间；但效果并不理想，**JDK1.6后官方就不再提倡用户使用。**
   * **CMS收集器无法处理浮动垃圾**  
     CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。

**(1):浮动垃圾：**

由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，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。这一部分垃圾就称为“**浮动垃圾**”。  
由于在垃圾收集阶段用户线程还需要运行，那就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，也可以热为CMS所需要的空间比其他垃圾收集器大；

      "-XX:CMSInitiatingOccupancyFraction"：设置CMS预留内存空间；

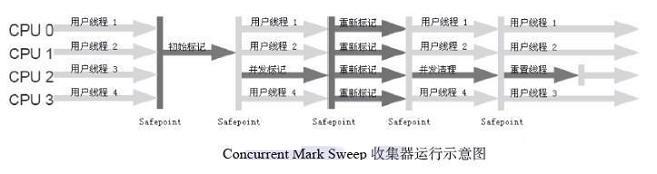
      JDK1.5默认值为68%；

      JDK1.6变为大约92%；

**(2):"Concurrent Mode Failure"失败：**  
如果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，这样会导致另一次Full GC的产生。**这样停顿时间就更长了，代价会更大，所以 "-XX:CMSInitiatingOccupancyFraction"不能设置得太大。

* + **CMS收集器会产生大量空间碎片**  
    CMS是一款基于“标记—清除”算法实现的收集器，清除后不进行压缩操作，这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。  
    解决办法：  
    **（1）、"-XX:+UseCMSCompactAtFullCollection"**  
    使得CMS出现上面这种情况时不进行Full GC，而开启内存碎片的合并整理过程； 但合并整理过程无法并发，停顿时间会变长； 默认开启（但不会进行，结合下面的CMSFullGCsBeforeCompaction）；    **（2）、"-XX:+CMSFullGCsBeforeCompaction"**  
    设置执行多少次不压缩的Full GC后，来一次压缩整理； 为减少合并整理过程的停顿时间； 默认为0，也就是说每次都执行Full GC，不会进行压缩整理；

1. **运行示意图如下：**

****

**2.7 G1 收集器**

G1（Garbage-First）是一款**面向服务端应用**的垃圾收集器，JDK 7 Update4 后开始进入商用。HotSpot开发团队赋予它的使命是未来可以替换掉JDK 1.5中发布的CMS收集器。

在G1之前的其他收集器进行收集的范围都是整个新生代或者老年代，而G1不再是这样。使用G1收集器时，Java堆的内存布局就与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有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的了，它们都是一部分Region（不需要连续）的集合。

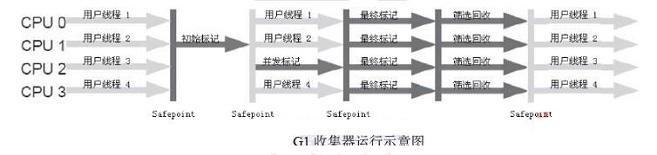
**G1收集器之所以能建立可预测的停顿时间模型**，是因为它可以有计划地避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region（这也就是Garbage-First名称的来由）。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限的时间内可以获取尽可能高的收集效率。

与其他GC收集器相比，G1具备如下特点。

* 1. **特性：**
     + **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Stop-The-World停顿的时间，**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。
     + **分代收集（收集范围包括新生代和老年代）**  
       与其他收集器一样，分代概念在G1中依然得以保留。**G1可以不需要其他收集器配合就能独立管理整个GC堆，它能够采用不同的方式去处理不同时期的对象。**使用G1收集器时，Java堆的内存布局有了很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有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的了，它们都是一部分Region（不需要连续）的集合。
     + **空间整合（结合多种垃圾收集算法，不产生碎片）**  
       与CMS的“标记—清理”算法不同，G1从**整体来看是基于“标记—整理”**算法实现的收集器，从**局部（两个Region之间）上来看是基于“复制”**算法实现的，**但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次GC。**
     + **可预测的停顿（低停顿的同时实现高吞吐量）**  
       这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，但G1除了追求低停顿外，**还能建立可预测的停顿时间模型**，**能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。**
  2. **应用场景：**  
     **1.面向服务端应用，针对具有大内存、多处理器的机器**；**最主要的应用是为需要低GC延迟，并具有大堆的应用程序提供解决方案**；  
     如：在堆大小约6GB或更大时，可预测的暂停时间可以低于0.5秒；  
     **2.用来替换掉JDK1.5的CMS收集器；**  
     （1）、超过50%的Java堆被活动数据占用；  
     （2）、对象分配频率或年代提升频率变化很大；  
     （3）、GC停顿时间过长（长与0.5至1秒）。  
     是否一定采用G1呢？也未必：  
     如果现在采用的收集器没有出现问题，不用着急去选择G1;  
     如果应用程序追求低停顿，可以尝试选择G1;  
     是否替代CMS需要实际场景测试才知道。

**3.执行过程：**  
G1收集器的运作大致可划分为以下几个步骤：

* **初始标记（Initial Marking）**  
  初始标记阶段仅仅只是标记一下GC Roots能直接关联到的对象，并且修改TAMS（Next Top at Mark Start）的值，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（Concurrent Marking）**  
  并发标记阶段是从GC Root开始对堆中对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可与用户程序并发执行。
* **最终标记（Final Marking）**  
  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，这阶段需要停顿线程，但是可并行执行。
* **筛选回收（Live Data Counting and Evacuation）**  
  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停顿时间来制定回收计划，这个阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分价值高的Region区的垃圾对象，时间是用户可控制的，而且停顿用户线程将大幅提高收集效率。回收时，采用“复制”算法，从一个或多个Region复制存活对象到堆上的另一个空的Region，并且在此过程中压缩和释放内存。
* **4.运行示意图如下：**



**设置参数：**

**"-XX:+UseG1GC"：指定使用G1收集器；**

**"-XX:InitiatingHeapOccupancyPercent"：当整个Java堆的占用率达到参数值时，开始并发标记阶段；默认为45；**

**"-XX:MaxGCPauseMillis"：为G1设置暂停时间目标，默认值为200毫秒；**

**"-XX:G1HeapRegionSize"：设置每个Region大小，范围1MB到32MB；目标是在最小Java堆时可以拥有约2048个**

**3.垃圾收集器总结**

来看一下对垃圾收集器的总结，列了一张表

|  |  |  |  |
| --- | --- | --- | --- |
| GC组合 | Minor GC | Full GC | 描述 |
| -XX:+UseSerialGC | Serial收集器串行回收 | Serial Old收集器串行回收 | 该选项可以手动指定Serial收集器+Serial Old收集器组合执行内存回收 |
| -XX:+UseParNewGC | ParNew收集器并行回收 | Serial Old收集器串行回收 | 该选项可以手动指定ParNew收集器+Serilal Old组合执行内存回收 |
| -XX:+UseParallelGC | Parallel收集器并行回收 | Serial Old收集器串行回收 | 该选项可以手动指定Parallel收集器+Serial Old收集器组合执行内存回收 |
| -XX:+UseParallelOldGC | Parallel收集器并行回收 | Parallel Old收集器并行回收 | 该选项可以手动指定Parallel收集器+Parallel Old收集器组合执行内存回收 |
| -XX:+UseConcMarkSweepGC | ParNew收集器并行回收 | 缺省使用CMS收集器并发回收，备用采用Serial Old收集器串行回收 | 该选项可以手动指定ParNew收集器+CMS收集器+Serial Old收集器组合执行内存回收。优先使用ParNew收集器+CMS收集器的组合，当出现ConcurrentMode Fail或者Promotion Failed时，则采用ParNew收集器+Serial Old收集器的组合 |
| -XX:+UseConcMarkSweepGC  -XX:-UseParNewGC | Serial收集器串行回收 |
| -XX:+UseG1GC | G1收集器并发、并行执行内存回收 | | 暂无 |

**GC日志**

每种收集器的日志形式都是由它们自身的实现所决定的，换言之，每种收集器的日志格式都可以不一样。不过虚拟机为了方便用户阅读，将各个收集器的日志都维持了一定的共性，就以最前面的对象间相互引用的那个类ReferenceCountingGC的代码为例：

**虚拟机参数为“-XX:+PrintGCDetails -XX:+UseSerialGC”，使用Serial+Serial Old组合进行垃圾回收的日志**

[复制代码](javascript:void(0);)

复制代码

[GC [DefNew: 310K->194K(2368K), 0.0269163 secs] 310K->194K(7680K), 0.0269513 secs] [Times: user=0.00 sys=0.00, real=0.03 secs]

[GC [DefNew: 2242K->0K(2368K), 0.0018814 secs] 2242K->2241K(7680K), 0.0019172 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[Full GC (System) [Tenured: 2241K->193K(5312K), 0.0056517 secs] 4289K->193K(7680K), [Perm : 2950K->2950K(21248K)], 0.0057094 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

Heap

def new generation total 2432K, used 43K [0x00000000052a0000, 0x0000000005540000, 0x0000000006ea0000)

eden space 2176K, 2% used [0x00000000052a0000, 0x00000000052aaeb8, 0x00000000054c0000)

from space 256K, 0% used [0x00000000054c0000, 0x00000000054c0000, 0x0000000005500000)

to space 256K, 0% used [0x0000000005500000, 0x0000000005500000, 0x0000000005540000)

tenured generation total 5312K, used 193K [0x0000000006ea0000, 0x00000000073d0000, 0x000000000a6a0000)

the space 5312K, 3% used [0x0000000006ea0000, 0x0000000006ed0730, 0x0000000006ed0800, 0x00000000073d0000)

compacting perm gen total 21248K, used 2982K [0x000000000a6a0000, 0x000000000bb60000, 0x000000000faa0000)

the space 21248K, 14% used [0x000000000a6a0000, 0x000000000a989980, 0x000000000a989a00, 0x000000000bb60000)

No shared spaces configured.

复制代码

[复制代码](javascript:void(0);)

**虚拟机参数为“-XX:+PrintGCDetails -XX:+UseParNewGC”，使用ParNew+Serial Old组合进行垃圾回收的日志**

[复制代码](javascript:void(0);)

复制代码

[GC [ParNew: 310K->205K(2368K), 0.0006664 secs] 310K->205K(7680K), 0.0007043 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[GC [ParNew: 2253K->31K(2368K), 0.0032525 secs] 2253K->2295K(7680K), 0.0032911 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[Full GC (System) [Tenured: 2264K->194K(5312K), 0.0054415 secs] 4343K->194K(7680K), [Perm : 2950K->2950K(21248K)], 0.0055105 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

Heap

par new generation total 2432K, used 43K [0x0000000005550000, 0x00000000057f0000, 0x0000000007150000)

eden space 2176K, 2% used [0x0000000005550000, 0x000000000555aeb8, 0x0000000005770000)

from space 256K, 0% used [0x0000000005770000, 0x0000000005770000, 0x00000000057b0000)

to space 256K, 0% used [0x00000000057b0000, 0x00000000057b0000, 0x00000000057f0000)

tenured generation total 5312K, used 194K [0x0000000007150000, 0x0000000007680000, 0x000000000a950000)

the space 5312K, 3% used [0x0000000007150000, 0x0000000007180940, 0x0000000007180a00, 0x0000000007680000)

compacting perm gen total 21248K, used 2982K [0x000000000a950000, 0x000000000be10000, 0x000000000fd50000)

the space 21248K, 14% used [0x000000000a950000, 0x000000000ac39980, 0x000000000ac39a00, 0x000000000be10000)

No shared spaces configured.

复制代码

[复制代码](javascript:void(0);)

**虚拟机参数为“-XX:+PrintGCDetails -XX:+UseParallelGC”，使用Parallel+Serial Old组合进行垃圾回收的日志**

[复制代码](javascript:void(0);)

复制代码

[GC [PSYoungGen: 4417K->288K(18688K)] 4417K->288K(61440K), 0.0007910 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]

[Full GC (System) [PSYoungGen: 288K->0K(18688K)] [PSOldGen: 0K->194K(42752K)] 288K->194K(61440K) [PSPermGen: 2941K->2941K(21248K)], 0.0032663 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]

Heap

PSYoungGen total 18688K, used 321K [0x0000000034190000, 0x0000000035660000, 0x0000000048f90000)

eden space 16064K, 2% used [0x0000000034190000,0x00000000341e05c0,0x0000000035140000)

from space 2624K, 0% used [0x0000000035140000,0x0000000035140000,0x00000000353d0000)

to space 2624K, 0% used [0x00000000353d0000,0x00000000353d0000,0x0000000035660000)

PSOldGen total 42752K, used 194K [0x000000000a590000, 0x000000000cf50000, 0x0000000034190000)

object space 42752K, 0% used [0x000000000a590000,0x000000000a5c0810,0x000000000cf50000)

PSPermGen total 21248K, used 2982K [0x0000000005190000, 0x0000000006650000, 0x000000000a590000)

object space 21248K, 14% used [0x0000000005190000,0x0000000005479980,0x0000000006650000)

复制代码

[复制代码](javascript:void(0);)

**虚拟机参数为“-XX:+PrintGCDetails -XX:+UseConcMarkSweepGC”，使用ParNew+CMS+Serial Old组合进行垃圾回收的日志**

[复制代码](javascript:void(0);)

复制代码

[Full GC (System) [CMS: 0K->194K(62656K), 0.0080796 secs] 4436K->194K(81792K), [CMS Perm : 2941K->2940K(21248K)], 0.0081589 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]

Heap

par new generation total 19136K, used 340K [0x0000000005540000, 0x0000000006a00000, 0x0000000006a00000)

eden space 17024K, 2% used [0x0000000005540000, 0x0000000005595290, 0x00000000065e0000)

from space 2112K, 0% used [0x00000000065e0000, 0x00000000065e0000, 0x00000000067f0000)

to space 2112K, 0% used [0x00000000067f0000, 0x00000000067f0000, 0x0000000006a00000)

concurrent mark-sweep generation total 62656K, used 194K [0x0000000006a00000, 0x000000000a730000, 0x000000000a940000)

concurrent-mark-sweep perm gen total 21248K, used 2981K [0x000000000a940000, 0x000000000be00000, 0x000000000fd40000)

复制代码

[复制代码](javascript:void(0);)

这四段GC日志中提炼出一些共性：

1、日志的开头“GC”、“Full GC”表示这次垃圾收集的停顿类型，而不是用来区分新生代GC还是老年代GC的。如果有Full，则说明本次GC停止了其他所有工作线程。看到Full GC的写法是“Full GC(System)”，这说明是调用System.gc()方法所触发的GC。

2、“GC”中接下来的“DefNew”、“ParNew”、“PSYoungGen”、“CMS”表示的是老年代垃圾收集器的名称，“PSYoungGen”中的“PS”指的是“Parallel Scavenge”，它是Parallel收集器的全称。

3、以第一个为例，方括号内部的“320K->194K(2368K)”、“2242K->0K(2368K)”，指的是**该区域已使用的容量->GC后该内存区域已使用的容量(该内存区总容量)**。方括号外面的“310K->194K(7680K)”、“2242K->2241K(7680K)”则指的是**GC前Java堆已使用的容量->GC后Java堆已使用的容量(Java堆总容量)**。

4、还以第一个为例，再往后“0.0269163 secs”表示该内存区域GC所占用的时间，单位是秒。最后的“[Times: user=0.00 sys=0.00 real=0.03 secs]”则更具体了，user表示用户态消耗的CPU时间、内核态消耗的CPU时间、操作从开始到结束经过的钟墙时间。后面两个的区别是，钟墙时间包括各种非运算的等待消耗，比如等待磁盘I/O、等待线程阻塞，而CPU时间不包括这些耗时，但当系统有多CPU或者多核的话，多线程操作会叠加这些CPU时间所以如果user或sys超过real是完全正常的。

5、“Heap”后面就列举出堆内存目前各个年代的区域的内存情况

**4.触发GC的时机**

最后总结一下什么时候会触发一次GC，个人经验看，有三种场景会触发GC：

1、第一种场景应该很明显，当年轻代或者老年代满了，Java虚拟机无法再为新的对象分配内存空间了，那么Java虚拟机就会触发一次GC去回收掉那些已经不会再被使用到的对象

2、手动调用System.gc()方法，通常这样会触发一次的Full GC以及至少一次的Minor GC

3、程序运行的时候有一条低优先级的GC线程，它是一条守护线程，当这条线程处于运行状态的时候，自然就触发了一次GC了。这点也很好证明，不过要用到WeakReference的知识，后面写WeakReference的时候会专门讲到这个。