

第三周作业

- 1、使用 GCLogAnalysis.java 自己演练一遍串行/并行/CMS/G1的案例。

- ①:

- 启动参数: -XX:+UseSerialGC -XX:+PrintGCDetails -Xloggc:gc.%p.%t.demo.log -XX:+PrintGCDateStamps -XX:+PrintGCApplicationStoppedTime -XX:+PrintReferenceGC -Xms128m -Xmx128m
- 造成结果: 2020-10-26T23:39:56.183+0800: 0.420: [GC (Allocation Failure) 2020-10-26T23:39:56.184+0800: 0.421: [DefNew2020-10-26T23:39:56.190+0800: 0.427: [SoftReference, 0 refs, 0.0000532 secs]2020-10-26T23:39:56.190+0800: 0.427: [WeakReference, 0 refs, 0.0000051 secs]2020-10-26T23:39:56.190+0800: 0.427: [FinalReference, 1 refs, 0.0000128 secs]2020-10-26T23:39:56.190+0800: 0.427: [PhantomReference, 0 refs, 0 refs, 0.0000044 secs]2020-10-26T23:39:56.190+0800: 0.427: [JNI Weak Reference, 0.0000062 secs]: 34341K->4351K(39296K), 0.0063055 secs] 34341K->12252K(126720K), 0.0071326 secs] [Times: user=0.00 sys=0.02, real=0.01 secs]

real = user + sys 截取第一次分配失败造成 young gc, 初始年轻代占用 34M 清理到 4M, 年轻代总容量为 39M, 初始整个堆的内存使用为 34M 清理到 12M, 堆总容量为 126M, stw 暂停时间为 0.42s

2020-10-26T23:39:56.183+0800: 0.420 GC 的开始时间, 0.420 表示相对于 JVM 启动的时间

GC 用来区分 Minor GC / Full GC 的标志, 这个表示小型 GC (Minor GC)

Allocation Failure GC 发生原因: 分配失败

DefNew 表示垃圾收集器的名字, 这个名字表示年轻代使用的单线程、标记-复制、STW 垃圾收集器

0.0071326 secs 表示整个 GC 事件持续的时间

[Times: user=0.00 sys=0.02, real=0.01 secs] 表示此次 GC 的持续时间, 通过三个部分来衡量, user: 表示所有 GC 线程消耗的 CPU 时间, sys 表示系统调用和系统等待事件消耗的时间, real 表示应用程序暂停的时间, 因为串行垃圾收集器只使用单线程所以: real = user + sys

- 分析串行 GC 日志得:



- young GC (minor GC): 小型 GC, 清理年轻代空间, DefNew: 用于清理年轻代垃圾收集器的名称
 - 34341K->4351K(39296K), 0.0063055 secs] 34341K->12252K(126720K), 0.0071326 secs] [Times: user=0.00 sys=0.02, real=0.01 secs] 年轻代使用量与堆总使用量相同得: 刚开始老年代使用量为 0, young GC 结束后老年代使用量为: 12252-4351=7901, 整个堆占用量为 126720, 可得老年代占用整个堆比例为: 0%-6%, 年轻代占用整个堆比例为: 27%-3%, 整个堆使用量比例为: 27%-9%, 程序暂停时间: 10ms
 - 39141K->4345K(39296K), 0.0103104 secs] 47042K->27228K(126720K), 0.0103425 secs], young GC 开始前老年代使用量为: 47042-39141=7901, young GC 结束后老年代使用量为: 27228-4345=22883, 老年代占用整个堆的比例为: 6%-18%, 年轻代占用整个堆比例为: 30%-3%, 整个堆使用量: 37%-21%

Minor GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前				
GC后				

- major GC :大型GC，清理老年代空间，Tenured：用于清理老年代垃圾收集器的名称
 - [DefNew: 39210K->39210K(39296K), 0.0000113 secs]2020-10-26T23:39:56.276+0800: 0.513: [Tenured2020-10-26T23:39:56.276+0800: 0.513: [SoftReference, 0 refs, 0.0000117 secs]2020-10-26T23:39:56.277+0800: 0.513: [WeakReference, 2 refs, 0.0000055 secs]2020-10-26T23:39:56.277+0800: 0.513: [FinalReference, 21 refs, 0.0000066 secs]2020-10-26T23:39:56.277+0800: 0.514: [PhantomReference, 0 refs, 0 refs, 0.0000040 secs]2020-10-26T23:39:56.277+0800: 0.514: [JNI Weak Reference, 0.0000040 secs]: 72746K->83052K(87424K), 0.0189611 secs] 111957K->83052K(126720K), [Metaspace: 3546K->3546K(1056768K)], 0.0191573 secs] [Times: user=0.01 sys=0.00, real=0.02 secs] 年轻代大小未改变，老年代大小由72746--83052，占整个堆比例为57%--65%，停顿19ms
 - [DefNew: 34496K->34496K(39296K), 0.0000197 secs]2020-10-26T23:39:56.303+0800: 0.539: [Tenured2020-10-26T23:39:56.303+0800: 0.540: [SoftReference, 0 refs, 0.0000091 secs]2020-10-26T23:39:56.303+0800: 0.540: [WeakReference, 0 refs, 0.0000040 secs]2020-10-26T23:39:56.303+0800: 0.540: [FinalReference, 3 refs, 0.0000044 secs]2020-10-26T23:39:56.303+0800: 0.540: [PhantomReference, 0 refs, 0 refs, 0.0000040 secs]2020-10-26T23:39:56.303+0800: 0.540: [JNI Weak Reference, 0.0000040 secs]: 83052K->87351K(87424K), 0.0257564 secs] 117548K->92292K(126720K), [Metaspace: 3546K->3546K(1056768K)], 0.0258450 secs] [Times: user=0.01 sys=0.00, real=0.03 secs] 年轻代大小未改变，老年代大小由83052--87351，占整个堆比例为65%--69%，停顿25ms
- full GC :完全GC，清理所有空间
 - [Full GC (Allocation Failure) 2020-10-26T23:39:56.336+0800: 0.573: [Tenured2020-10-26T23:39:56.337+0800: 0.574: [SoftReference, 0 refs, 0.0000113 secs]2020-10-26T23:39:56.337+0800: 0.574: [WeakReference, 0 refs, 0.0000044 secs]2020-10-26T23:39:56.337+0800: 0.574: [FinalReference, 2 refs, 0.0000040 secs]2020-10-26T23:39:56.337+0800: 0.574: [PhantomReference, 0 refs, 0 refs, 0.0000040 secs]2020-10-26T23:39:56.337+0800: 0.574: [JNI Weak Reference, 0.0000040 secs]: 87351K->87338K(87424K), 0.0210332 secs] 126293K->104105K(126720K), [Metaspace: 3546K->3546K(1056768K)], 0.0210886 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
 -

Full GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前	?	?		
GC后	?	?		

- ②
 - 启动参数: -XX:+UseSerialGC -XX:+PrintGCDetails -Xloggc:gc.%p.%t.demo.log -XX:+PrintGCDateStamps -Xms512m -Xmx512m, 程序运行10s
 - 共生成对象次数:48217
- ③
 - 启动参数: -XX:+UseSerialGC -XX:+PrintGCDetails -Xloggc:gc.%p.%t.demo.log -XX:+PrintGCDateStamps -Xms1024m -Xmx1024m, 程序运行10s
 - 共生成对象次数:110617
- ④
 - 启动参数: -XX:+UseSerialGC -XX:+PrintGCDetails -Xloggc:gc.%p.%t.demo.log -XX:+PrintGCDateStamps -Xms2048m -Xmx2048m, 程序运行10s
 - 共生成对象次数:133197
- 2、使用压测工具 (wrk或sb) , 演练gateway-server-0.0.1-SNAPSHOT.jar 示例。
- 3、(选做)如果自己本地有可以运行的项目, 可以按照2的方式进行演练。
- 根据上述自己对于1和2的演示, 写一段对于不同 GC 的总结, 提交到 Github
- 问题:
 - XX:-UseLargePagesIndividualAllocation什么意思?
 - 串行垃圾收集器, 使用单线程, 所以: real = user +sys, 但为什么我运行出来的GC日志大多不同, 比如: [Times: user=0.00 sys=0.02, real=0.01 secs], [Times: user=0.02 sys=0.00, real=0.01 secs], [Times: user=0.02 sys=0.00, real=0.01 secs] (之前为young GC) , [Times: user=0.02 sys=0.00, real=0.02 secs], [Times: user=0.02 sys=0.00, real=0.03 secs], [Times: user=0.00 sys=0.00, real=0.01 secs] (之前为fullGC)
 - major GC 年轻代大小未改变, 老年代大小由72746--83052, 占整个堆比例为57%--65%, 老年代反而变大? 整个堆大小由111957--83052, metaspace没变, 那么到底少了些什么?
- 总结:
 - 1.串行GC增大堆内存, 可减少major GC及Full GC、oom发生频率, 可增加程序运行效率, GC时间由存活对象数决定;
 - GC 策略
 - Serial收集器 一个单线程的收集器, 在进行垃圾收集时候, 必须暂停其他所有的工作线程直到它收集结束。特点: CPU利用率最高, 停顿时间即用户等待时间比较长。

适用场景：小型应用 通过JVM参数-XX:+UseSerialGC可以使用串行垃圾回收器。

- Parallel收集器 采用多线程来通过扫描并压缩堆 特点：停顿时间短，回收效率高，对吞吐量要求高。适用场景：大型应用，科学计算，大规模数据采集等。通过JVM参数 XX:+UseParNewGC 打开并发标记扫描垃圾回收器。
- CMS收集器 采用“标记-清除”算法实现，使用多线程的算法去扫描堆，对发现未使用的对象进行回收。（1）初始标记（2）并发标记（3）并发预处理（4）重新标记（5）并发清除（6）并发重置 特点：响应时间优先，减少垃圾收集停顿时间 适应场景：服务器、电信领域等。通过JVM参数 -XX:+UseConcMarkSweepGC设置
- G1收集器 在G1中，堆被划分成许多个连续的区域(region)。采用G1算法进行回收，吸收了CMS收集器特点。特点：支持很大的堆，高吞吐量 -支持多CPU和垃圾回收线程 -在主线程暂停的情况下，使用并行收集 -在主线程运行的情况下，使用并发收集 实时目标：可配置在N毫秒内最多只占用M毫秒的时间进行垃圾回收 通过JVM参数 -XX:+UseG1GC 使用G1垃圾回收器

• GC的选择

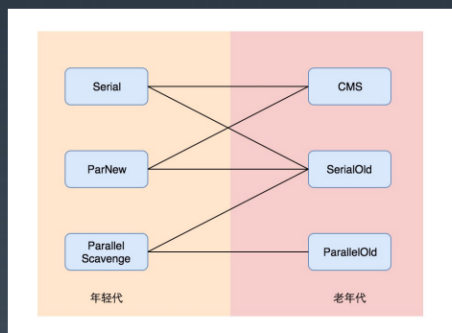
- 官方推荐，需要根据应用的实际情况进行选择。在选择之前必须要对应用的堆大小、收集频率进行估算。
- 使用SerialGC的场景：
 - 1、如果应用的堆大小在100MB以内。
 - 2、如果应用在一个单核单线程的服务器上面，并且对应用暂停的时间无需求。
- 使用ParallelGC的场景：
 - 如果需要应用在高峰期有较好的性能，但是对应用停顿时间无高要求（比如：停顿1s甚至更长）。
- 使用G1、CMS场景：
 - 1、对应用的延迟有很高的要求。
 - 2、如果内存大于6G请使用G1。

• 一、常见垃圾收集器

• 现在常见的垃圾收集器有如下几种：

- 新生代收集器：
 - Serial
 - ParNew
 - Parallel Scavenge
- 老年代收集器：
 - Serial Old
 - CMS
 - Parallel Old
- 堆内存垃圾收集器：
 - G1

常用的 GC 组合（重点）



常用的组合为：

（1）Serial+Serial Old 实现单线程的低延迟垃圾回收机制；

（2）ParNew+CMS，实现多线程的低延迟垃圾回收机制；

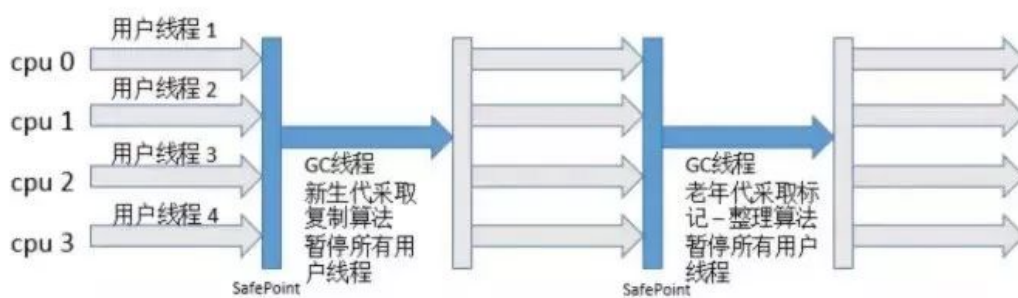
（3）Parallel Scavenge和Parallel Scavenge Old，实现多线程的高吞吐量垃圾回收机制；

二、新生代垃圾收集器

（1）Serial 收集器

- Serial 是一款用于新生代的单线程收集器，采用复制算法进行垃圾收集。Serial 进行垃圾收集时，不仅只用一条线程执行垃圾收集工作，它在收集的同时，所有的用户线程必须暂停（Stop The World）。
- 就比如妈妈在家打扫卫生的时候，肯定不会边打扫边让儿子往地上乱扔纸屑，否则一边制造垃圾，一遍清理垃圾，这活啥时候也干不完。
- 如下是 Serial 收集器和 Serial Old 收集器结合进行垃圾收集的示意图，当用户线程都执行到安全点时，所有线程暂停执行，Serial 收集器以单线程，采用复制算法进行垃圾收集工作，收集完之后，用户线程继续开始执行。

示意图

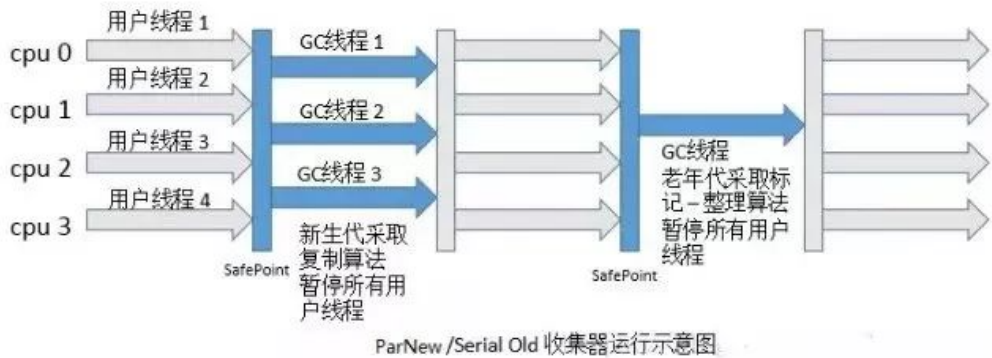


Serial/Serial Old 收集器运行示意图

- 适用场景：Client 模式（桌面应用）；单核服务器。
- 可以用 `-XX:+UserSerialGC` 来选择 Serial 作为新生代收集器。
- （2）ParNew 收集器
 - ParNew 就是一个 Serial 的多线程版本，其它与 Serial 并无区别。ParNew 在单核 CPU 环境并不会比 Serial 收集器达到更好的效果，它默认开启的收集线程数和 CPU 数量一致，可以通过 `-XX:ParallelGCThreads` 来设置垃圾收集的线程数。

- 如下是 ParNew 收集器和 Serial Old 收集器结合进行垃圾收集的示意图，当用户线程都执行到安全点时，所有线程暂停执行，ParNew 收集器以多线程，采用复制算法进行垃圾收集工作，收集完之后，用户线程继续开始执行。

- **示意图**



- 适用场景：多核服务器；与 CMS 收集器搭配使用。当使用 -XX:+UserConcMarkSweepGC 来选择 CMS 作为老年代收集器时，新生代收集器默认就是 ParNew，也可以用 -XX:+UseParNewGC 来指定使用 ParNew 作为新生代收集器。

- **(3) Parallel Scavenge 收集器**

- Parallel Scavenge 也是一款用于新生代的多线程收集器，与 ParNew 的不同之处是 ParNew 的目标是尽可能缩短垃圾收集时用户线程的停顿时间，Parallel Scavenge 的目标是达到一个可控制的吞吐量。
- 吞吐量就是 CPU 执行用户线程的时间与 CPU 执行总时间的比值【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)】，比如虚拟机一共运行了 100 分钟，其中垃圾收集花费了 1 分钟，那吞吐量就是 99%。比如下面两个场景，垃圾收集器每 100 秒收集一次，每次停顿 10 秒，和垃圾收集器每 50 秒收集一次，每次停顿时间 7 秒，虽然后者每次停顿时间变短了，但是总体吞吐量变低了，CPU 总体利用率变低了。

- **示意图**

收集频率	每次停顿时间	吞吐量
每100秒收集一次	10秒	91%
每50秒收集一次	7秒	88%

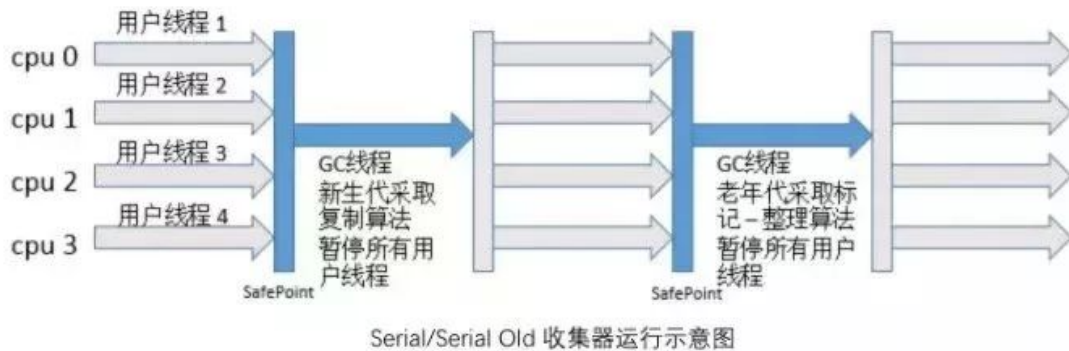
- 可以通过 -XX:MaxGCPauseMillis 来设置收集器尽可能在多长时间完成内存回收，可以通过 -XX:GCTimeRatio 来精确控制吞吐量。
- 如下是 Parallel 收集器和 Parallel Old 收集器结合进行垃圾收集的示意图，在新生代，当用户线程都执行到安全点时，所有线程暂停执行，ParNew 收集器以多线程，采用复制算法进行垃圾收集工作，收集完之后，用户线程继续开始执行；在老年代，当用户线程都执行到安全点时，所有线程暂停执行，Parallel Old 收集器以多线程，采用标记整理算法进行垃圾收集工作。
- 适用场景：注重吞吐量，高效利用 CPU，需要高效运算且不需要太多交互。

- 可以使用 -XX:+UseParallelGC 来选择 Parallel Scavenge 作为新生代收集器，jdk7、jdk8 默认使用 Parallel Scavenge 作为新生代收集器。

• 三、老年代垃圾收集器

• (1) Serial Old 收集器

- Serial Old 收集器是 Serial 的老年代版本，同样是一个单线程收集器，采用标记-整理算法。
- 如下图是 Serial 收集器和 Serial Old 收集器结合进行垃圾收集的示意图：



- 适用场景：Client 模式（桌面应用）；单核服务器；与 Parallel Scavenge 收集器搭配；作为 CMS 收集器的后备预案。
- (2) CMS(Concurrent Mark Sweep) 收集器
 - CMS 收集器是一种以最短回收停顿时间为目标的收集器，以“最短用户线程停顿时间”著称。整个垃圾收集过程分为 4 个步骤：
 - ① 初始标记：标记一下 GC Roots 能直接关联到的对象，速度较快。
 - ② 并发标记：进行 GC Roots Tracing，标记出全部的垃圾对象，耗时较长。
 - ③ 重新标记：修正并发标记阶段引用户程序继续运行而导致变化的对象的标记记录，耗时较短。
 - ④ 并发清除：用标记-清除算法清除垃圾对象，耗时较长。
 - 整个过程耗时最长的并发标记和并发清除都是和用户线程一起工作，所以从总体上来说，CMS 收集器垃圾收集可以看做是和用户线程并发执行的。

UserSerialGC 采用Serial+ Serial Old的收集器组合进行垃圾回收

UserParNewGC 使用ParNew+Serial Old组合

UseConMarkSweepGC 使用ParNew+CMS+SerialOld组合进行垃圾回收。SerialOld是在Concurrent Mode Failure失败后的后备收集器使用

UseParallelGC 虚拟机运行在Server模式下的默认值。使用Parallel Scavenge +Serial Old的收集器组合进行垃圾回收 (jdk8用的就是这个，但是老年代用的是Parallel Old)

UseParallelOldGC 使用Parallel Scavenge + Parallel Old组合进行垃圾回收

SurvivorRatio 新生代中Eden区与Survivor区域的容量比值，默认为8，代表Eden:Survivor=8:1

PretenureSizeThreshold 直接晋升到老年代的对象的大小

MaxTenuringThreshold 晋升到老年代对象的年龄，超过这个数值进入老年代

UseAdaptiveSizePolicy 动态调整Java堆中各个区域的大小以及进入老年代的年龄

HandlePromotionFailure 是否允许分配担保失败，即老年代的剩余空间不足以应付新生代的整个Eden和Survivor区的所有对象存活的极端情况

ParallelGCThreads 并行GC时进行内存回收的线程数

GCTimeRatio GC时间占总时间的比率，默认值99%，即允许1%的GC时间，仅在使用Parallel Scavenge收集器时生效

MaxGCPauseMillis GC的最大停顿时间，仅在使用Parallel Scavenge收集器时生效

CMSInitiatingOccupancyFraction 设置CMS在老年代空间被使用多少后触发垃圾回收，默认是92%，仅在使用CMS收集器时生效

UseCMSCompactAtFullCollection 设置CMS收集器在完成垃圾收集后是否要进行一次内存整理。仅在使用CMS收集器时生效

CMSFullGCsBeforeCompaction 设置CMS收集器进行若干次垃圾收集后再启动一次内存碎片整理。仅在使用CMS时生效

PrintGCDetails 查看程序运行时的GC细节

