

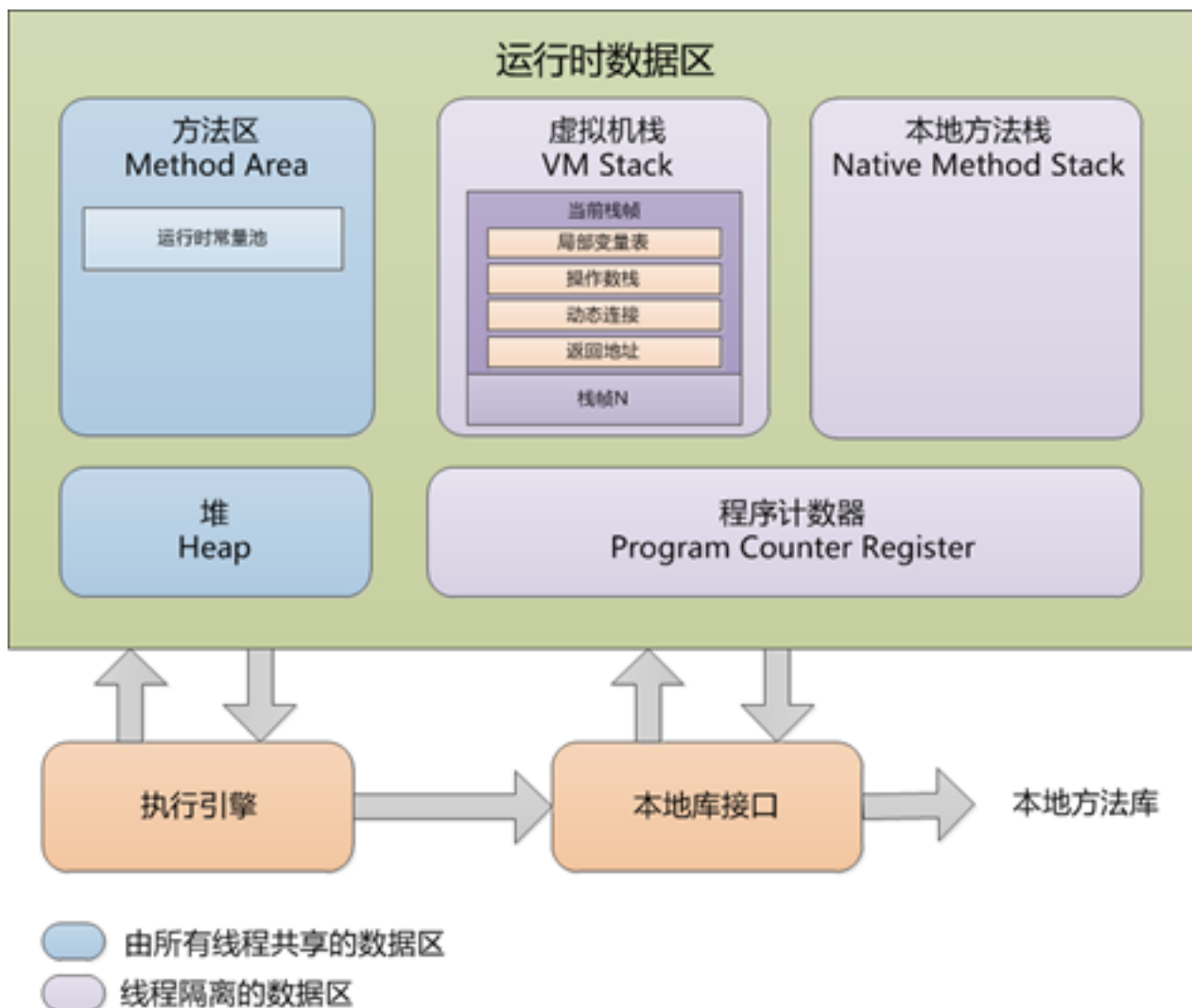
Java虚拟机：内存管理与垃圾回收

2016年7月11日 星期一

11:59

本文主要是基于Sun JDK 1.6 Garbage Collector (作者：毕玄) 的整理与总结，原文请读者在网上搜索。

1、Java虚拟机运行时的数据区



2、常用的内存区域调节参数

- Xms**：初始堆大小，默认为物理内存的1/64(<1GB)；默认(MinHeapFreeRatio参数可以调整)空余堆内存小于40%时，JVM就会增大堆直到-Xmx的最大限制
- Xmx**：最大堆大小，默认(MaxHeapFreeRatio参数可以调整)空余堆内存大于70%时，JVM会减少堆直到 -Xms的最小限制
- Xmn**：新生代的内存空间大小，**注意**：此处的大小是 (eden+ 2 survivor space)。与 jmap -heap中显示的New gen是不同的。整个堆大小=新生代大小 + 老生代大小 + 永久代大小。

在保证堆大小不变的情况下,增大新生代后,将会减小老生代大小。此值对系统性能影响较大,Sun官方推荐配置为整个堆的3/8。

-XX:SurvivorRatio : 新生代中Eden区域与Survivor区域的容量比值,默认值为8。两个Survivor区与一个Eden区的比值为2:8,一个Survivor区占整个年轻代的1/10。

-Xss : 每个线程的堆栈大小。JDK5.0以后每个线程堆栈大小为1M,以前每个线程堆栈大小为256K。应根据应用的线程所需内存大小进行适当调整。在相同物理内存下,减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的,不能无限生成,经验值在3000~5000左右。一般小的应用,如果栈不是很深,应该是128k够用的,大的应用建议使用256k。这个选项对性能影响比较大,需要严格的测试。和threadstacksize选项解释很类似,官方文档似乎没有解释,在论坛中有这样一句话:"-Xss is translated in a VM flag named ThreadStackSize"一般设置这个值就可以了。

-XX:PermSize : 设置永久代(perm gen)初始值。默认值为物理内存的1/64。

-XX:MaxPermSize : 设置持久代最大值。物理内存的1/4。

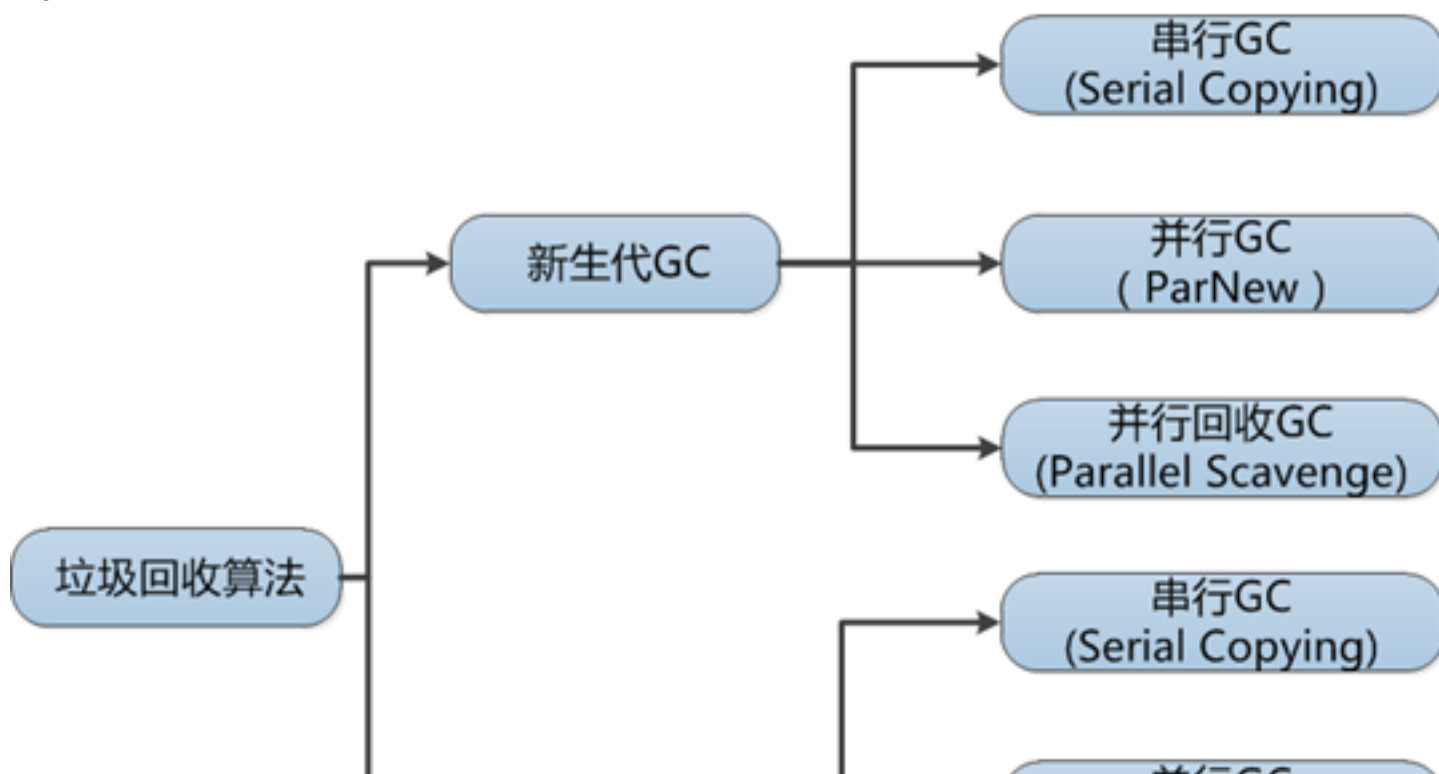
3、内存分配方法

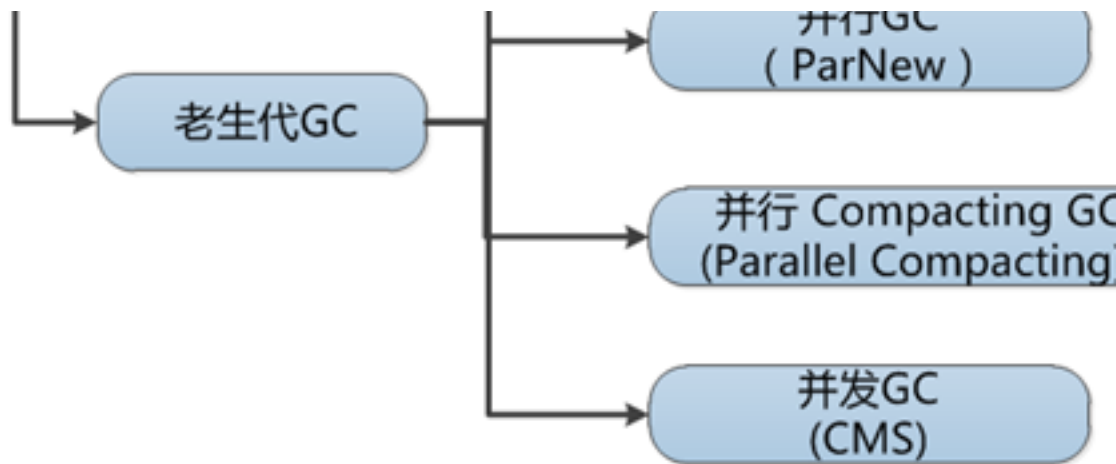
1) 堆上分配 2) 栈上分配 3) 堆外分配 (DirectByteBuffer或直接使用Unsafe.allocateMemory,但不推荐这种方式)

4、监控方法

- 1) 系统程序运行时可通过jstat -gcutil来查看堆中各个内存区域的变化以及GC的工作状态;
- 2) 启动时可添加-XX:+PrintGCDetails -Xloggc:<file>输出到日志文件来查看GC的状况;
- 3) jmap -heap可用于查看各个内存空间的大小;

5) 断代法可用GC汇总





一、新生代可用GC

1) 串行GC(Serial Copying) : client模式下默认GC方式，也可通过-XX:+UseSerialGC来强制指定；默认情况下 eden、s0、s1的大小通过-XX:SurvivorRatio来控制，默认为8，含义

为eden:s0的比例，启动后可通过jmap -heap [pid]来查看。

默认情况下，仅在TLAB或eden上分配，只有两种情况下会在老生代分配：

- 1、需要分配的内存大小超过eden space大小；
- 2、在配置了PretenureSizeThreshold的情况下，对象大小大于此值。

默认情况下，触发Minor GC时：

之前Minor GC晋级到old的平均大小 < 老生代的剩余空间 < eden+from Survivor的使用空间。当HandlePromotionFailure为true，则仅触发minor gc；如为false，则触发full GC。

默认情况下，新生代对象晋升到老生代的规则：

1、经历多次minor gc仍存活的对象，可通过以下参数来控制：以MaxTenuringThreshold值为准，默认为15。

2、to space放不下的，直接放入老生代；

2) 并行GC (ParNew) : CMS GC时默认采用，也可采用-XX:+UseParNewGC强制指定；垃圾回收的时候采用多线程的方式。

3) 并行回收GC(Parallel Scavenge) : server模式下默认的GC方式，也可采用-XX:+UseParallelGC强制指定；eden、s0、s1的大小可通过-XX:SurvivorRatio来控制，但默认情况下

以-XX:InitialSurvivorRatio为准，此值默认为8，**代表的为新生代大小：s0**，这点要特别注意。

默认情况下，当TLAB、eden上分配都失败时，判断需要分配的内存大小是否 >= eden space的一半大小，如是就直接在老生代上分配；

默认情况下的垃圾回收规则：

1、在回收前PS GC会先检测之前每次PS GC时，晋升到老生代的平均大小是否大于老生代的剩余空间，如大于则直接触发full GC。

C
)

1. 回收空间，如不足则直接触发Full GC，

2、在回收后，也会按照上面的规则进行检测。

默认情况下的新生代对象晋升到老年代的规则：

1、经历多次minor gc仍存活的对象，可通过以下参数来控制：AlwaysTenure，默认false，表示只要minor GC时存活，就晋升到老年代；NeverTenure，默认false，表示永不晋升到老年代；上面两个都没设置的情况下，如UseAdaptiveSizePolicy，启动时以InitialTenuringThreshold值作为存活次数的阈值，在每次ps gc后会动态调整，如不使用UseAdaptiveSizePolicy，则以MaxTenuringThreshold为准。

2、to space放不下的，直接放入老年代。

在回收后，如UseAdaptiveSizePolicy，PS GC会根据运行状态动态调整eden、to以及TenuringThreshold的大小。如果不希望动态调整可设置-XX:-UseAdaptiveSizePolicy。如希望跟踪每次的变化情况，可在启动参数上增加：PrintAdaptiveSizePolicy。

二、老年代可用GC

1、串行GC(Serial Copying)：client方式下默认GC方式，可通过-XX:+UseSerialGC强制指定。

触发机制汇总：

- 1) old gen空间不足；
- 2) perm gen空间不足；
- 3) minor gc时的悲观策略；
- 4) minor GC后在eden上分配内存仍然失败；
- 5) 执行heap dump时；
- 6) 外部调用System.gc，可通过-XX:+DisableExplicitGC来禁止。

2、并行回收GC(Parallel Scavenge)：server模式下默认GC方式，可通过-XX:+UseParallelGC强制指定；并行的线程数为当cpu core<=8 ? cpu core : 3+(cpu core*5)/8或通过-XX:ParallelGCThreads=x来强制指定。如ScavengeBeforeFullGC为true（默认值），则先执行minor GC。

3、并行Compacting：可通过-XX:+UseParallelOldGC强制指定。

4、并发CMS：可通过-XX:+UseConcMarkSweepGC来强制指定。并发的线程数默认为：(并行GC线程数+3)/4，也可通过ParallelCMSThreads指定。

触发机制：

1、当老年代空间的使用到达一定比率时触发；

Hotspot V 1.6中默认为65%，可通过PrintCMSInitiationStatistics（此参数在V 1.5中不能用）来查看这个值到底是多少；可通过CMSInitiatingOccupancyFraction来强制指定，默认值并不是赋值在了这个值上，是根据如下公式计算出来的： $((100 - \text{MinHeapFreeRatio}) + (\text{double})(\text{CMSTriggerRatio} * \text{MinHeapFreeRatio}) / 100.0) / 100.0$ ；其中，MinHeapFreeRatio默认值：40 CMSTriggerRatio默认值：80。

2、当CMS收集器空间使用到一定比率时触发；

2、当perm gen采用CMS收集且空间使用到一定比率时触发；

perm gen采用CMS收集需设置：-XX:+CMSClassUnloadingEnabled Hotspot V 1.6中默认为65%；可通过CMSInitiatingPermOccupancyFraction来强制指定，同样，它是根据如下公式计算出来的： $((100 - \text{MinHeapFreeRatio}) + (\text{double}) (\text{CMSTriggerPermRatio} * \text{MinHeapFreeRatio}) / 100.0) / 100.0$ ；其中，MinHeapFreeRatio默认值：40 CMSTriggerPermRatio默认值：80。

3、Hotspot根据成本计算决定是否需要执行CMS GC；可通过-XX: +UseCMSInitiatingOccupancyOnly来去掉这个动态执行的策略。

4、外部调用了System.gc，且设置了ExplicitGCInvokesConcurrent；需要注意，在hotspot 6中，在这种情况下如应用同时使用了NIO，可能会出现bug。

6、GC组合

1) 默认GC组合

	新生代GC方式	旧生代和持久代GC方式
Client	串行GC	串行GC
Server	并行回收GC	并行 MSC GC (JDK 5.0 Update 6)

2) 可选的GC组合

	新生代GC方式	旧生代和持久代GC方式
-XX:+UseSerialGC	串行GC	串行GC
-XX:+UseParallelGC	PS GC	并行MSC GC
-XX:+UseConcMarkSweepGC	ParNew GC	并发GC 当出现concurrent failure时采用串行GC
-XX:+UseParNewGC	并行GC	串行GC
-XX:+UseParallelOldGC	PS GC	并行Compacting GC
-XX:+UseConcMarkSweepGC -XX:-UseParNewGC	串行GC	并发GC 当出现Concurrent failure或promotion failure时则采用串行GC

5以后)

方式
Mode C
C
Mode n failed

不支持的组合方式	1、-XX:+UseParNewGC -XX:+UseParallelOldGC 2、-XX:+UseParNewGC -XX:+UseSerialGC
----------	---

7、GC监测

1) jstat -gcutil [pid] [interval] [count]

2) -verbose:gc // 可以辅助输出一些详细的GC信息；-XX:+PrintGCDetails // 输出GC详细信息；-XX:+PrintGCApplicationStoppedTime // 输出GC造成应用暂停的时间
-XX:+PrintGCDateStamps // GC发生的时间信息；-XX:+PrintHeapAtGC // 在GC前后输出堆中各个区域的大小；-Xloggc:[file] // 将GC信息输出到单独的文件中，建议都加上，这个消耗不大，而且对查问题和调优有很大的帮助。gc的日志拿下来后可使用GCLogViewer或gchisto进行分析。

3) 图形化的情况下可直接用jvisualvm进行分析。

4) 查看内存的消耗状况

(1) 长期消耗，可以直接dump，然后MAT(内存分析工具)查看即可

(2) 短期消耗，图形界面情况下，可使用jvisualvm的memory profiler或jprofiler。

8、系统调优方法

步骤：1、评估现状 2、设定目标 3、尝试调优 4、衡量调优 5、细微调整

设定目标：


- 1) 降低Full GC的执行频率？
- 2) 降低Full GC的消耗时间？
- 3) 降低Full GC所造成的应用停顿时间？
- 4) 降低Minor GC执行频率？
- 5) 降低Minor GC消耗时间？

例如某系统的GC调优目标：降低Full GC执行频率的同时，尽可能降低minor GC的执行频率、消耗时间以及GC对应用造成的停顿时间。

衡量调优：

1、衡量工具

1) 打印GC日志信息：-XX:+PrintGCDetails -XX:+PrintGCApplicationStoppedTime -Xloggc: {文件名} -XX:+PrintGCTimeStamps

2) jmap：（由于每个版本jvm的默认值可能会有改变，建议还是用jmap首先观察下目前每个代的内存大小、GC方式）

3) 运行状况监测工具：jstat、jvisualvm、sar、gclogviewer

2、应收集的信息

1) minor gc的执行频率；full gc的执行频率，每次GC耗时多少？

2) 内存消耗情况



3) minor gc回收的效果如何？survivor的消耗状况如何，每次有多少对象会进入老年代？

4) full gc回收的效果如何？（简单的**memory leak**判断方法）

5) 系统的load、cpu消耗、qps or tps、响应时间

QPS每秒查询率：是对一个特定的查询服务器在规定时间内所处理流量多少的衡量标准。在因特网上，作为域名服务器的机器性能经常用每秒查询率来衡量。对应fetches/sec，即每秒的响应请求数，也即是最大吞吐能力。

TPS(Transaction Per Second)：每秒钟系统能够处理的交易或事务的数量。

尝试调优：

注意Java RMI的定时GC触发机制，可通过：`-XX:+DisableExplicitGC`来禁止或通过 `-Dsun.rmi.dgc.server.gcInterval=3600000`来控制触发的时间。

1) 降低Full GC执行频率 - 通常瓶颈

老年代本身占用的内存空间就一直偏高，所以只要稍微放点对象到老年代，就full GC了；

通常原因：系统缓存的东西太多；

例如：使用oracle 10g驱动时preparedstatement cache太大；

查找办法：现执行Dump然后再进行MAT分析；

（1）Minor GC后总是有对象不断的进入老年代，导致老年代不断的满

通常原因：Survivor太小了

系统表现：系统响应太慢、请求量太大、每次请求分配的内存太多、分配的对象太大...

查找办法：分析两次minor GC之间到底哪些地方分配了内存；

利用jstat观察Survivor的消耗状况，`-XX:PrintHeapAtGC`，输出GC前后的详细信息；

对于系统响应慢可以采用系统优化，不是GC优化的内容；

（2）老年代的内存占用一直偏高

调优方法：① 扩大老年代的大小（减少新生代的大小或调大heap的大小）；

减少new注意对minor gc的影响并且同时有可能造成full gc还是严重；

调大heap注意full gc的时间的延长，cpu够强悍嘛，os是32 bit的吗？

② 程序优化（去掉一些不必要的缓存）

（3）Minor GC后总是有对象不断的进入老年代

前提：这些进入老年代的对象在full GC时大部分都会被回收

调优方法：

① 降低Minor GC的执行频率；

② 让对象尽量在Minor GC中就被回收掉：增大Eden区、增大survivor、增大TenuringThreshold；注意这些可能会造成minor gc执行频繁；

③ 切换到CMS GC：老年代还没有满就回收掉，从而降低Full GC触发的可能性；

④ 程序优化：提升响应速度、降低每次请求分配的内存、

（4）降低单次Full GC的执行时间

通常原因：老生代太大了...

调优方法：1) 是并行GC吗？ 2) 升级CPU 3) 减小Heap或老生代

(5) 降低Minor GC执行频率

通常原因：每次请求分配的内存多、请求量大

通常办法：1) 扩大heap、扩大新生代、扩大eden。注意点：降低每次请求分配的内存；横向增加机器的数量分担请求的数量。

(6) 降低Minor GC执行时间

通常原因：新生代太大了，响应速度太慢了，导致每次Minor GC时存活的对象多

通常办法：1) 减小点新生代吧；2) 增加CPU的数量、升级CPU的配置；加快系统的响应速度

细微调整：

首先需要了解以下情况：

① 当响应速度下降到多少或请求量上涨到多少时，系统会宕掉？

② 参数调整后系统多久会执行一次Minor GC，多久会执行一次Full GC，高峰期会如何？

需要计算的量：

① 每次请求平均需要分配多少内存？系统的平均响应时间是多少呢？请求量是多少、多常时间执行一次Minor GC、Full GC？

② 现有参数下，应该是多久一次Minor GC、Full GC，对比真实状况，做一定的调整；

必杀技：提升响应速度、降低每次请求分配的内存？

9、系统调优举例

现象：1、系统响应速度大概为100ms；2、当系统QPS增长到40时，机器每隔5秒就执行一次minor gc，每隔3分钟就执行一次full gc，并且很快就一直full GC了；4、每次Full gc后旧生代大概会消耗400M，有点多了。

解决方案：解决Full GC次数过多的问题

(1) 降低响应时间或请求次数，这个需要重构，比较麻烦；——这个是终极方法，往往能够顺利的解决问题，因为大部分的问题均是由程序自身造成的。

(2) 减少老生代内存的消耗，比较靠谱；——可以通过分析Dump文件(jmap dump)，并利用MAT查找内存消耗的原因，从而发现程序中造成老生代内存消耗的原因。

(3) 减少每次请求的内存的消耗，貌似比较靠谱；——这个是海市蜃楼，没有太好的办法。

(4) 降低GC造成的应用暂停的时间——可以采用CMS GS垃圾回收器。参数设置如下：

-Xms1536m -Xmx1536m -Xmn700m -XX:SurvivorRatio=7 -XX:

+UseConcMarkSweepGC -XX:+UseCMSCompactAtFullCollection

-XX:CMSMaxAbortablePrecleanTime=1000 -XX:+CMSClassUnloadingEnabled -

XX:+UseCMSInitiatingOccupancyOnly -XX:+DisableExplicitGC

(5) 减少每次minor gc晋升到old的对象。可选方法：1) 调大新生代。2) 调大Survivor。3) 调大TenuringThreshold。

调大Survivor：当前采用PS GC，Survivor space会被动态调整。由于调整幅度很小，

导致了经常有对象直接转移到了老年代；于是禁止Survivor区的动态调整了，-XX:-UseAdaptiveSizePolicy，并计算Survivor Space需要的大小，于是继续观察，并做微调...。最终将Full GC推迟到2小时1次。

10、垃圾回收的实现原理

内存回收的实现方法：1) 引用计数：不适合复杂对象的引用关系，尤其是循环依赖的场景。2) 有向图Tracing：适合于复杂对象的引用关系场景，Hotspot采用这种。常用算法：Copying、Mark-Sweep、Mark-Compact。

Hotspot从root set开始扫描有引用的对象并对Reference类型的对象进行特殊处理。

以下是Root Set的列表：1) 当前正在执行的线程；2) 全局/静态变量；3) JVM Handles；4) JNI 【Java Native Interface】Handles；

另外：minor GC只扫描新生代，当老年代的对象引用了新生代的对象时，会采用如下的处理方式：在给对象赋引用时，会经过一个write barrier的过程，以便检查是否有老年代引用新生代对象的情况，如有则记录到remember set中。并在minor gc时，remember set指向的新生代对象也作为root set。

新生代串行GC(Serial Copying)：

新生代串行GC(Serial Copying)完整内存的分配策略：

- 1) 首先在TLAB(本地线程分配缓冲区)上尝试分配；
- 2) 检查是否需要在新生代上分配，如需要分配的大小小于PretenureSizeThreshold，则在eden区上进行分配，分配成功则返回；分配失败则继续；
- 3) 检查是否需要尝试在老年代上分配，如需要，则遍历所有代并检查是否可在该代上分配，如可以则进行分配；如不需要在老年代上尝试分配，则继续；
- 4) 根据策略决定执行新生代GC或Full GC，执行full gc时不清除soft Ref；
- 5) 如需要分配的大小大于PretenureSizeThreshold，尝试在老年代上分配，否则尝试在新生代上分配；
- 6) 尝试扩大堆并分配；
- 7) 执行full gc，并清除所有soft Ref，按步骤5继续尝试分配。

新生代串行GC(Serial Copying)完整内存回收策略

- 1) 检查to是否为空，不为空返回false；
- 2) 检查老年代剩余空间是否大于当前eden+from已用的大小，如大于则返回true，如小于且HandlePromotionFailure为true，则检查剩余空间是否大于之前每次minor gc晋级到老年代的平均大小，如大于返回true，如小于返回false。
- 3) 如上面的结果为false，则执行full gc；如上面的结果为true，执行下面的步骤；
- 4) 扫描引用关系，将活的对象copy到to space，如对象在minor gc中的存活次数超过tenuring_threshold或分配失败，则往老年代复制，如仍然复制失败，则取决于HandlePromotionFailure，如不需要处理，直接抛出OOM，并退出vm，如需处理，则保持这些新生代对象不动。

新生代可用GC-PS

完整内存分配策略

- 1) 先在TLAB上分配，分配失败则直接在eden上分配；
- 2) 当eden上分配失败时，检查需要分配的大小是否 \geq eden space的一半，如是，则直接在老生代分配；
- 3) 如分配仍然失败，且gc已超过频率，则抛出OOM；
- 4) 进入基本分配策略失败的模式；
- 5) 执行PS GC，在eden上分配；
- 6) 执行非最大压缩的full gc，在eden上分配；
- 7) 在旧生代上分配；
- 8) 执行最大压缩full gc，在eden上分配；
- 9) 在旧生代上分配；
- 10) 如还失败，回到2。

最悲惨的情况，分配触发多次PS GC和多次Full GC，直到OOM。

完整内存回收策略

- 1) 如gc所执行的时间超过，直接结束；
- 2) 先调用invoke_nopolicy
 - 2.1 先检查是不是要尝试scavenge；
 - 2.1.1 to space必须为空，如不为空，则返回false；
 - 2.1.2 获取之前所有minor gc晋级到old的平均大小，并对比目前eden+from已使用的大小，取更小的一个值，如老生代剩余空间小于此值，则返回false，如大于则返回true；
 - 2.2 如不需要尝试scavenge，则返回false，否则继续；
 - 2.3 多线程扫描活的对象，并基于copying算法回收，回收时相应的晋升对象到旧生代；
 - 2.4 如UseAdaptiveSizePolicy，那么重新计算to space和tenuringThreshold的值，并调整。
- 3) 如invoke_nopolicy返回的是false，或之前所有minor gc晋级到老生代的平均大小 $>$ 旧生代的剩余空间，那么继续下面的步骤，否则结束；
- 4) 如UseParallelOldGC，则执行PSParallelCompact，如不是UseParallelOldGC，则执行PSMarkSweep。

老生代并行CMS GC：

优缺点：

- 1) 大部分时候和应用并发进行，因此只会造成很短的暂停时间；
- 2) 浮动垃圾，没办法，所以内存空间要稍微大一点；
- 3) 内存碎片，-XX:+UseCMSCompactAtFullCollection 来解决；
- 4) 争抢CPU，这GC方式就这样；

一、在垃圾回收中，新生代和老生代的垃圾回收策略是不同的。

- 5) 多次remark, 所以总的gc时间会比开行的长;
- 6) 内存分配, free list方式, so性能稍差, 对minor GC会有一点影响;
- 7) 和应用并发, 有可能分配和回收同时, 产生竞争, 引入了锁, JVM分配优先。

11、TLAB的解释

堆内的对象数据是各个线程所共享的, 所以当在堆内创建新的对象时, 就需要进行锁操作。锁操作是比较耗时, 因此JVM为每个线程在堆上分配了一块“自留地”——TLAB(全称是Thread Local Allocation Buffer), 位于堆内存的新生代, 也就是Eden区。每个线程在创建新的对象时, 会首先尝试在自己的TLAB里进行分配, 如果成功就返回, 失败了再到共享的Eden区里去申请空间。在线程自己的TLAB区域创建对象失败一般有两个原因: 一是对象太大, 二是自己的TLAB区剩余空间不够。通常默认的TLAB区域大小是Eden区域的1%, 当然也可以手工进行调整, 对应的JVM参数是-XX:TLABWasteTargetPercent。

参考文献:

- 1、Sun JDK 1.6 GC (Garbage Collector) 作者: 毕玄

