# Java虚拟机：自动内存管理机制

之前看过JVM的相关知识，当时没有留下任何学习成果物，有些遗憾。这次重新复习了下，并通过博客来做下笔记(只能记录一部分，因为写博客真的很花时间)，也给其他同行一些知识分享。

Java自动内存管理机制包含两部分：内存分配和内存回收，要想理解内存分配和回收的机制，则需要了解下Java内存区域（Java运行时数据区），这篇随笔将按照下面的线索进行逐步解析：

1. Java运行时数据区
2. 对象“已死”的判定算法
3. 垃圾收集算法
4. 垃圾收集器
5. 结束语

好，接下来我们一一来看。

**一、Java运行时数据区**

根据《Java虚拟机规范》的规定，Java虚拟机所管理的内存将会包括如下几个运行时数据区域



img

* 程序计数器：用来记录当前线程所执行的字节码指令的行号指示器。字节码计时器需要通过改变改值来选取下一条需要执行的字节码指定，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个指示器来完成。程序计数器是唯一一个没有规定任何OutOfMemoryError情况的区域。
* Java虚拟机栈：虚拟机栈描述的是Java方法执行的内存模型，每个方法执行时都会创建一个栈帧用来存储局部变量表（存放编译器可知的各种基本数据类型、对象引用和returnAddress类型，所需的内存空间在编译器完成分配）、操作数栈、动态链接、方法出口等信息。Java虚拟机栈有两种异常情况：OutOfMemoryError（扩展时无法申请到足够内存）和StackOverflowError（线程请求的栈深度大于虚拟机所允许的深度）。
* 本地方法栈：同Java虚拟机栈类似，只不过Java虚拟机栈为虚拟机执行Java方法服务，本地方法栈为虚拟机使用Native方法服务。HotSpot直接将两个栈合二为一。也规定了两种异常：OutOfMemoryError和StackOverflowError。
* 堆：JVM所管理的内存中最大的一块，也是GC管理的主要区域。理论上所有的对象实例和数组都要在堆上分配。堆的大小是可以扩展的，通过-Xms和-Xms控制，并且堆无法扩展的时候就会报OutOfMemoryError异常。
* 方法区：用来存储JVM加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但是为了和堆区分开来，它也叫Non-Heap（非堆）。方法区无法满足内存分配需求时，报OutOfMemoryError异常。
* 直接内存：并不是虚拟机运行时数据区的一部分，也不是JVM规范中定义的内存区域，但是却被经常使用。JDK1.4中新加入的NIO类，引入了基于通道和缓冲区的I/O方式，他可以直接分配对外内存，以提高性能。不收堆大小的限制，但是会受物理内存的约束。也会报OutOfMemoryError异常。

附栈到堆的关联例子（基于HotSpot）：



img

**二、对象“已死”的判定算法**

由于程序计数器、Java虚拟机栈、本地方法栈都是线程独享，其占用的内存也是随线程生而生、随线程结束而回收。而Java堆和方法区则不同，线程共享，是GC的所关注的部分。

在堆中几乎存在着所有对象，GC之前需要考虑哪些对象还活着不能回收，哪些对象已经死去可以回收。

有两种算法可以判定对象是否存活：

1. ）引用计数算法：给对象中添加一个引用计数器，每当一个地方应用了对象，计数器加1；当引用失效，计数器减1；当计数器为0表示该对象已死、可回收。但是它很难解决两个对象之间相互循环引用的情况。
2. ）可达性分析算法：通过一系列称为“GC Roots”的对象作为起点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC Roots没有任何引用链相连（即对象到GC Roots不可达），则证明此对象已死、可回收。Java中可以作为GC Roots的对象包括：虚拟机栈中引用的对象、本地方法栈中Native方法引用的对象、方法区静态属性引用的对象、方法区常量引用的对象。

在主流的商用程序语言（如我们的Java）的主流实现中，都是通过可达性分析算法来判定对象是否存活的。

**三、垃圾收集算法**

1、标记-清除算法

最基础的算法，分标记和清除两个阶段：首先标记处所需要回收的对象，在标记完成后统一回收所有被标记的对象。

它有两点不足：一个效率问题，标记和清除过程都效率不高；一个是空间问题，标记清除之后会产生大量不连续的内存碎片（类似于我们电脑的磁盘碎片），空间碎片太多导致需要分配大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾回收动作。



img

2、复制算法

为了解决效率问题，出现了“复制”算法，他将可用内存按容量划分为大小相等的两块，每次只需要使用其中一块。当一块内存用完了，将还存活的对象复制到另一块上面，然后再把刚刚用完的内存空间一次清理掉。这样就解决了内存碎片问题，但是代价就是可以用内容就缩小为原来的一半。



img

3、标记-整理算法

复制算法在对象存活率较高时就会进行频繁的复制操作，效率将降低。因此又有了标记-整理算法，标记过程同标记-清除算法，但是在后续步骤不是直接对对象进行清理，而是让所有存活的对象都向一侧移动，然后直接清理掉端边界以外的内存。



img

4、分代收集算法

当前商业虚拟机的GC都是采用分代收集算法，这种算法并没有什么新的思想，而是根据对象存活周期的不同将堆分为：新生代和老年代，方法区称为永久代（在新的版本中已经将永久代废弃，引入了元空间的概念，永久代使用的是JVM内存而元空间直接使用物理内存）。

这样就可以根据各个年代的特点采用不同的收集算法。



img

新生代中的对象“朝生夕死”，每次GC时都会有大量对象死去，少量存活，使用复制算法。新生代又分为Eden区和Survivor区（Survivor from、Survivor to），大小比例默认为8:1:1。

老年代中的对象因为对象存活率高、没有额外空间进行分配担保，就使用标记-清除或标记-整理算法。

* 新产生的对象优先进去Eden区，当Eden区满了之后再使用Survivor from，当Survivor from 也满了之后就进行Minor GC（新生代GC），将Eden和Survivor from中存活的对象copy进入Survivor to，然后清空Eden和Survivor from，这个时候原来的Survivor from成了新的Survivor to，原来的Survivor to成了新的Survivor from。复制的时候，如果Survivor to 无法容纳全部存活的对象，则根据老年代的分配担保（类似于银行的贷款担保）将对象copy进去老年代，如果老年代也无法容纳，则进行Full GC（老年代GC）。
* 大对象直接进入老年代：JVM中有个参数配置-XX:PretenureSizeThreshold，令大于这个设置值的对象直接进入老年代，目的是为了避免在Eden和Survivor区之间发生大量的内存复制。
* 长期存活的对象进入老年代：JVM给每个对象定义一个对象年龄计数器，如果对象在Eden出生并经过第一次Minor GC后仍然存活，并且能被Survivor容纳，将被移入Survivor并且年龄设定为1。没熬过一次Minor GC，年龄就加1，当他的年龄到一定程度（默认为15岁，可以通过XX:MaxTenuringThreshold来设定），就会移入老年代。但是JVM并不是永远要求年龄必须达到最大年龄才会晋升老年代，如果Survivor 空间中相同年龄（如年龄为x）所有对象大小的总和大于Survivor的一半，年龄大于等于x的所有对象直接进入老年代，无需等到最大年龄要求。

**四、垃圾收集器**

垃圾收集算法是方法论，垃圾收集器是具体实现。JVM规范对于垃圾收集器的应该如何实现没有任何规定，因此不同的厂商、不同版本的虚拟机所提供的垃圾收集器差别较大，这里只看HotSpot虚拟机。

JDK7/8后，HotSpot虚拟机所有收集器及组合（连线）如下：



img

**1.Serial收集器**

Serial收集器是最基本、历史最久的收集器，曾是新生代手机的唯一选择。他是单线程的，只会使用一个CPU或一条收集线程去完成垃圾收集工作，并且它在收集的时候，必须暂停其他所有的工作线程，直到它结束，即“**Stop the World**”。停掉所有的用户线程，对很多应用来说难以接受。比如你在做一件事情，被别人强制停掉，你心里奔腾而过的“羊驼”还数的过来吗？

尽管如此，它仍然是虚拟机运行在**client模式下的默认新生代收集器**：简单而高效（与其他收集器的单个线程相比，因为没有线程切换的开销等）。

工作示意图：



img

**2.ParNew收集器**

ParNew收集器是Serial收集器的多线程版本，除了使用了多线程之外，其他的行为（收集算法、stop the world、对象分配规则、回收策略等）同Serial收集器一样。

是许多运行在**Server模式下**的JVM中**首选的新生代收集器**，其中一个很重还要的原因就是除了Serial之外，只有他能和老年代的CMS收集器配合工作。

工作示意图：



img

**3.Parallel Scavenge收集器**

新生代收集器，并行的多线程收集器。它的目标是达到一个可控的**吞吐量（**就是CPU运行用户代码的时间与CPU总消耗时间的比值，即 吞吐量=行用户代码的时间/[行用户代码的时间+垃圾收集时间]**）**，这样可以**高效率的利用CPU时间**，**尽快完成程序的运算任务**，适合在**后台运算而不需要太多交互**的任务。

**4.Serial Old收集器**

Serial 收集器的老年代版本，单线程，“标记整理”算法，**主要是给Client模式下**的虚拟机使用。

另外还可以在Server模式下：

* JDK 1.5之前的版本中雨Parallel Scavenge 收集器搭配使用
* 可以作为CMS的后背方案，在CMS发生Concurrent Mode Failure是使用

工作示意图：



img

**5.Parallel Old收集器**

Parallel Scavenge的老年代版本，多线程，“标记整理”算法，JDK 1.6才出现。在此之前Parallel Scavenge只能同Serial Old搭配使用，由于Serial Old的性能较差导致Parallel Scavenge的优势发挥不出来，尴了个尬~~

Parallel Old收集器的出现，使“**吞吐量优先**”收集器终于有了名副其实的组合。在**吞吐量和CPU敏感**的场合，都可以使用Parallel Scavenge/Parallel Old组合。组合的工作示意图如下：



img

**6.CMS收集器**

CMS(Concurrent Mark Sweep)收集器是一种以获取最短回收停顿时间为目标的收集器，停顿时间短，用户体验就好。

基于“标记清除”算法，**并发收集、低停顿**，运作过程复杂，分4步：

1. )初始标记：仅仅标记GC Roots能直接关联到的对象，速度快，但是需要“Stop The World”
2. )并发标记：就是进行追踪引用链的过程，可以和用户线程并发执行。
3. )重新标记：修正并发标记阶段因用户线程继续运行而导致标记发生变化的那部分对象的标记记录，比初始标记时间长但远比并发标记时间短，需要“Stop The World”
4. )并发清除：清除标记为可以回收对象，可以和用户线程并发执行

由于整个过程耗时最长的并发标记和并发清除都可以和用户线程一起工作，所以总体上来看，CMS收集器的内存回收过程和用户线程是并发执行的。

工作示意图：



img

CSM收集器有3个缺点：

1. )对CPU资源非常敏感并发收集虽然不会暂停用户线程，但因为占用一部分CPU资源，还是会**导致应用程序变慢，总吞吐量降低**。 CMS的默认收集线程数量是=(CPU数量+3)/4；当CPU数量多于4个，收集线程占用的CPU资源多于25%，对用户程序影响可能较大；不足4个时，影响更大，可能无法接受。
2. )无法处理浮动垃圾（在并发清除时，用户线程新产生的垃圾叫浮动垃圾）,可能出现"Concurrent Mode Failure"失败。并发清除时需要预留一定的内存空间，不能像其他收集器在老年代几乎填满再进行收集；如果CMS预留内存空间无法满足程序需要，就会出现一次"Concurrent Mode Failure"失败；这时JVM启用后备预案：临时启用Serail Old收集器，而导致另一次Full GC的产生；
3. )产生大量内存碎片：CMS基于"标记-清除"算法，清除后不进行压缩操作产生大量不连续的内存碎片，这样会导致分配大内存对象时，无法找到足够的连续内存，从而需要提前触发另一次Full GC动作。

**7.G1收集器**

G1（Garbage-First）是**JDK7-u4**才正式推出商用的收集器。G1是**面向服务端**应用的垃圾收集器。它的使命是未来可以替换掉CMS收集器。

G1收集器特性：

并行与并发：能充分利用多CPU、多核环境的硬件优势，缩短停顿时间；能和用户线程并发执行。
分代收集：G1可以不需要其他GC收集器的配合就能独立管理整个堆，采用不同的方式处理新生对象和已经存活一段时间的对象。
空间整合：整体上看采用标记整理算法，局部看采用复制算法（两个Region之间），不会有内存碎片，不会因为大对象找不到足够的连续空间而提前触发GC，这点优于CMS收集器。
可预测的停顿：除了追求低停顿还能建立可以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超N毫秒，这点优于CMS收集器。

为什么能做到可预测的停顿？

是因为可以有计划的避免在整个Java堆中进行全区域的垃圾收集。
G1收集器将内存分大小相等的独立区域（Region），新生代和老年代概念保留，但是已经不再物理隔离。
G1跟踪各个Region获得其收集价值大小，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（名称Garbage-First的由来）；
这就保证了在有限的时间内可以获取尽可能高的收集效率。

对象被其他Region的对象引用了怎么办？



img

判断对象存活时，是否需要扫描整个Java堆才能保证准确？在其他的分代收集器，也存在这样的问题（而G1更突出）：新生代回收的时候不得不扫描老年代？
无论G1还是其他分代收集器，JVM都是使用Remembered Set来避免全局扫描：
每个Region都有一个对应的Remembered Set；
每次Reference类型数据写操作时，都会产生一个Write Barrier暂时中断操作；
然后检查将要写入的引用指向的对象是否和该Reference类型数据在不同的Region（其他收集器：检查老年代对象是否引用了新生代对象）；
如果不同，通过CardTable把相关引用信息记录到引用指向对象的所在Region对应的Remembered Set中；
进行垃圾收集时，在GC根节点的枚举范围加入Remembered Set，就可以保证不进行全局扫描，也不会有遗漏。



img

不计算维护Remembered Set的操作，回收过程可以分为4个步骤（与CMS较为相似）：

1. )初始标记：仅仅标记GC Roots能直接关联到的对象，并修改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，需要“Stop The World”
2. )并发标记：从GC Roots开始进行可达性分析，找出存活对象，耗时长，可与用户线程并发执行
3. )最终标记：修正并发标记阶段因用户线程继续运行而导致标记发生变化的那部分对象的标记记录。并发标记时虚拟机将对象变化记录在线程Remember Set Logs里面，最终标记阶段将Remember Set Logs整合到Remember Set中，比初始标记时间长但远比并发标记时间短，需要“Stop The World”
4. )筛选回收：首先对各个Region的回收价值和成本进行排序，然后根据用户期望的GC停顿时间来定制回收计划，最后按计划回收一些价值高的Region中垃圾对象。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；可以并发进行，降低停顿时间，并增加吞吐量。

工作示意图：



img