

概述

G1(Garbage-First)收集器是一种server-style 回收器，主要面向多核，大内存的服务器。G1 在实现高吞吐的同时，也最大限度满足了GC 停顿时间可控的目标。在Oracle JDK7 update 4 及 以后的版本全面支持G1 回收器功能。G1收集器主要为有如下需求的程序设计：

- 可以像CMS 收集器 能同时和应用线程 一起并发的执行；
- 实现压缩空间时用更少的停顿时间；
- 满足可预测的GC停顿时间需求；
- 不要牺牲太多的吞吐性能；
- 不需要占用更多的Java Heap；

未来 G1 计划要全面取代CMS的。G1相比CMS有更多的优势，G1是压缩型收集器，G1通过依赖regions分区，可以实现压缩更充分。这样消除大部分潜在的碎片问题。G1提供更精准的可预测的垃圾停顿时间，满足用户指定垃圾回收时间的需求。

G1的收集都是STW的，但年轻代和老年代的收集界限比较模糊，采用了混合(mixed)收集的方式。即每次收集既可能只收集年轻代分区(年轻代收集)，也可能在收集年轻代的同时，包含部分老年代分区(混合收集)，这样即使堆内存很大时，也可以限制收集范围，从而降低停顿。对应GC的两个分类。(young gc、mixed gc)

GC分类

young gc、mixed gc、full gc(??)

年轻代收集集合 CSet of Young Collection

应用线程不断活动后，年轻代空间会被逐渐填满。当JVM分配对象到Eden区域失败(Eden区已满)时，便会触发一次STW式的年轻代收集。在年轻代收集中，Eden分区存活的对象将被拷贝到Survivor分区；原有Survivor分区存活的对象，将根据任期阈值(tenuring threshold)分别晋升到PLAB中，新的survivor分区和老年代分区。而原有的年轻代分区将被整体回收掉。同时，年轻代收集还负责维护对象的年龄(存活次数)，辅助判断老化(tenuring)对象晋升的时候是到Survivor分区还是到老年代分区。年轻代收集首先先将晋升对象尺寸总和、对象年龄信息维护到年龄表中，再根据年龄表、Survivor尺寸、Survivor填充容量-

XX:TargetSurvivorRatio(默认50%)、最大任期阈值-XX:MaxTenuringThreshold(默认15)，计算出一个恰当的任期阈值，凡是超过任期阈值的对象都会被晋升到老年代。

混合收集集合 CSet of Mixed Collection

年轻代收集不断活动后，老年代的空间也会被逐渐填充。当老年代占用空间超过整堆比IHOP阈值-XX:InitiatingHeapOccupancyPercent(默认45%)时，G1就会启动一次混合垃圾收集周期。为了满足暂停目标，G1可能不能一口气将所有的候选分区收集掉，因此G1可能会

产生连续多次的混合收集与应用线程交替执行，每次STW的混合收集与年轻代收集过程相类似。

为了确定包含到年轻代收集集合CSet的老年代分区，JVM通过参数混合周期的最大总次数-XX:G1MixedGCCountTarget(默认8)、堆废物百分比-XX:G1HeapWastePercent(默认5%)。通过候选老年代分区总数与混合周期最大总次数，确定每次包含到CSet的最小分区数量；根据堆废物百分比，当收集达到参数时，不再启动新的混合收集。而每次添加到CSet的分区，则通过计算得到的GC效率进行安排。

下面详述。

G1年轻代收集

堆空间被分割成大约2048个区域。最小1M，最大32M，可设置。区域没有必要像旧的收集器一样是保持连续的。

活跃对象会被疏散（复制、移动）到一个或多个survivor区域。如果达到晋升总阈值(默认15)，对象会晋升到年老代区域。

这是一个stop the world暂停。为下一次年轻代垃圾回收计算Eden和Survivor的大小。保留审计信息有助于计算大小。类似目标暂停时间的事情会被考虑在内。

这个方法使重调区域大小变得很容易，按需把它们调大或调小。

关于G1的年轻代回收做以下总结：

- 堆空间是一块单独的内存空间被分割成多个区域。
- 年轻代内存是由一组非连续的区域组成。这使得需要重调大小变得容易。
- 年轻代垃圾回收是stop the world事件，所有应用线程都会因此操作暂停。
- 年轻代垃圾收集使用多线程并行回收。
- 活跃对象被复制到新的Survivor区或者年老代区域。

G1老年代回收过程(mixed gc)

类似CMS收集器，G1收集器为年老代对象被设计成一个低暂停收集器。

G1垃圾收集器在堆上的年老代执行以下阶段。注意一些阶段是年轻代回收的一部分。

阶段	描述
(1)初始标记(stop the world事件)	这是一个stop the world事件，使用G1回收器，背负着一个常规的年轻代收集。标记那些有引用到年老代的对象的survivor区(根区)
(2)根区扫描	为到年老代的引用扫描survivor区,这个发生在应用继续运行时。这个阶段在年轻代收集前必须完成
(3)并发标记	遍历整个堆寻找活跃对象，这个发生在应用运行时，这个阶段可以被年轻代垃圾回收打断。
(4)重新标记(stop the world事件)	完全标记堆中的活跃对象，使用一个叫作snapshot-at-the-beginning(SATB)的比CMS收集器的更快的算法
(5)清理(stop the world)	在活跃对象上执行审计操作和释放区域空间(stop the world)；净化已记

(*)暂停(stop the world 事件和并发)	在垃圾对象上执行垃圾扫描和垃圾回收区域工作(stop the world), 停止垃圾回收集合(stop the world); 重置空间区域和返回它们到空闲列表(并发)
(*)复制(stop the world 事件)	这些是stop the world暂停为了疏散或者复制活跃对象到新的未使用的区域。这个可以由被记录为[GC Pause (young)]的年轻代区域或者被记录为[GC Pause (mixed)]年轻代和年老代区域完成

初始标记 Initial Mark

初始标记(Initial Mark)负责标记所有能被直接可达的根对象(原生栈对象、全局对象、JNI对象), 根是对象图的起点, 因此初始标记需要将Mutator线程(Java应用线程)暂停掉, 也就是需要一个STW的时间段。事实上, 当达到IHOP阈值时, G1并不会立即发起并发标记周期, 而是等待下一次年轻代收集, 利用年轻代收集的STW时间段, 完成初始标记, 这种方式称为借道(Piggybacking)。在初始标记暂停中, 分区的NTAMS都被设置到分区顶部Top, 初始标记是并发执行, 直到所有的分区处理完。

根分区扫描 Root Region Scanning

在初始标记暂停结束后, 年轻代收集也完成的对象复制到Survivor的工作, 应用线程开始活跃起来。此时为了保证标记算法的正确性, 所有新复制到Survivor分区的对象, 都需要被扫描并标记成根, 这个过程称为根分区扫描(Root Region Scanning), 同时扫描的Survivor分区也被称为根分区(Root Region)。根分区扫描必须在下一次年轻代垃圾收集启动前完成(并发标记的过程中, 可能会被若干次年轻代垃圾收集打断), 因为每次GC会产生新的存活对象集合。

并发标记 Concurrent Marking

和应用线程并发执行, 并发标记线程在并发标记阶段启动, 由参数-XX:ConcGCThreads(默认GC线程数的1/4, 即-XX:ParallelGCThreads/4)控制启动数量, 每个线程每次只扫描一个分区, 从而标记出存活对象图。在这一阶段会处理Previous/Next标记位图, 扫描标记对象的引用字段。同时, 并发标记线程还会定期检查和处理STAB全局缓冲区列表的记录, 更新对象引用信息。参数-XX:+ClassUnloadingWithConcurrentMark会开启一个优化, 如果一个类不可达(不是对象不可达), 则在重新标记阶段, 这个类就会被直接卸载。所有的标记任务必须在堆满前就完成扫描, 如果并发标记耗时很长, 那么有可能在并发标记过程中, 又经历了几次年轻代收集。如果堆满前没有完成标记任务, 则会触发担保机制, 经历一次长时间的串行Full GC。

存活数据计算 Live Data Accounting

存活数据计算(Live Data Accounting)是标记操作的附加产物, 只要一个对象被标记, 同时会被计算字节数, 并计入分区空间。只有NTAMS以下的对象会被标记和计算, 在标记周期的最后, Next位图将被清空, 等待下次标记周期。

重新标记 Remark

重新标记(Remark)是最后一个标记阶段。在该阶段中，G1需要一个暂停的时间，去处理剩下的SATB日志缓冲区和所有更新，找出所有未被访问的存活对象，同时安全完成存活数据计算。这个阶段也是并行执行的，通过参数-XX:ParallelGCThread可设置GC暂停时可用的GC线程数。同时，引用处理也是重新标记阶段的一部分，所有重度使用引用对象(弱引用、软引用、虚引用、最终引用)的应用都会在引用处理上产生开销。

清除 Cleanup

紧挨着重新标记阶段的清除(Clean)阶段也是STW的。Previous/Next标记位图、以及PTAMS/NTAMS，都会在清除阶段交换角色。清除阶段主要执行以下操作：

RSet梳理，启发式算法会根据活跃度和RSet尺寸对分区定义不同等级，同时RSet数埋也有助于发现无用的引用。参数-XX:+PrintAdaptiveSizePolicy可以开启打印启发式算法决策细节；

整理堆分区，为混合收集周期识别回收收益高(基于释放空间和暂停目标)的老年代分区集合；

识别所有空闲分区，即发现无存活对象的分区。该分区可在清除阶段直接回收，无需等待下次收集周期。

关键定义

分区(Region)

G1采用了分区(Region)的思路，将整个堆空间分成若干个大小相等的内存区域，每次分配对象空间将逐段地使用内存。因此，在堆的使用上，G1并不要求对象的存储一定是物理上连续的，只要逻辑上连续即可；每个分区也不会确定地为某个代服务，可以按需在年轻代和老年代之间切换。启动时可以通过参数-XX:G1HeapRegionSize=n可指定分区大小(1MB~32MB，且必须是2的幂)，默认将整堆划分为2048个分区。

Eden区、survivor区、old区、Humongous区

在G1中，还有一种特殊的区域，叫Humongous区域。如果一个对象占用的空间超过了分区容量50%以上，G1收集器就认为这是一个巨型对象。这些巨型对象，默认直接会被分配在老年代，但是如果它是一个短期存在的巨型对象，就会对垃圾收集器造成负面影响。为了解决这个问题，G1划分了一个Humongous区，它用来专门存放巨型对象。如果一个H区装不下一个巨型对象，那么G1会寻找连续的H分区来存储。为了能找到连续的H区，有时候不得不启动Full GC。

尽量别申请超大的对象，现在暂时没有优化方法。

PS: 在java 8中, 持久代也移动到了普通的堆内存空间中, 改为元空间。

对象分配策略

说起大对象的分配, 我们不得不谈谈对象的分配策略。它分为3个阶段:

TLAB(Thread Local Allocation Buffer)线程本地分配缓冲区

Eden区中分配

Humongous区分配

TLAB为线程本地分配缓冲区, 它的目的为了使对象尽可能快的分配出来。如果对象在一个共享的空间中分配, 我们需要采用一些同步机制来管理这些空间内的空闲空间指针。在Eden空间中, 每一个线程都有一个固定的分区用于分配对象, 即一个TLAB。分配对象时, 线程之间不再需要进行任何的同步。

对TLAB空间中无法分配的对象, JVM会尝试在Eden空间中进行分配。如果Eden空间无法容纳该对象, 就只能在老年代中进行分配空间。

已记忆集合 Remember Set (RSet)

在CMS中, 也有RSet的概念, 在老年代中有一块区域用来记录指向新生代的引用。这是一种point-out, 在进行Young GC时, 扫描根时, 仅仅需要扫描这一块区域, 而不需要扫描整个老年代。

但在G1中, 并没有使用point-out, 这是由于一个分区太小, 分区数量太多, 如果是用point-out的话, 会造成大量的扫描浪费, 有些根本不需要GC的分区引用也扫描了。于是G1中使用point-in来解决。point-in的意思是哪些分区引用了当前分区中的对象。这样, 仅仅将这些对象当做根来扫描就避免了无效的扫描。由于新生代有多个, 那么我们需要在新生代之间记录引用吗? 这是不必要的, 原因在于每次GC时, 所有新生代都会被扫描, 所以只需要记录老年代到新生代之间的引用即可。

需要注意的是, 如果引用的对象很多, 赋值器需要对每个引用做处理, 赋值器开销会很大, 为了解决赋值器开销这个问题, 在G1 中又引入了另外一个概念, 卡表 (Card Table) 。

卡表 (Card Table)

一个Card Table将一个分区在逻辑上划分为固定大小的连续区域, 每个区域称之为卡。卡通常较小, 介于128到512字节之间。Card Table通常为字节数组, 由Card的索引 (即数组下标) 来标识每个分区的空间地址。默认情况下, 每个卡都未被引用。当一个地址空间被引用时, 这个地址空间对应的数组索引的值被标记为"0", 即标记为脏被引用, 此外RSet也将这个数组下标记录下来。一般情况下, 这个RSet其实是一个Hash Table, Key是别的Region的起始地址, Value是一个集合, 里面的元素是Card Table的Index。

收集集合 CSet

收集集合(CSet)代表每次GC暂停时回收的一系列目标分区。在任意一次收集暂停中, CSet所有分区都会被释放, 内部存活的对象都会被转移到分配的空闲分区中。因此无论是年轻代收集, 还是混合收集, 工作的机制都是一致的。年轻代收集CSet只容纳年轻代分区, 而混

合收集会通过启发式算法，在老年代候选回收分区中，筛选出回收收益最高的分区添加到CSet中。

候选老年代分区的CSet准入条件，可以通过活跃度阈值-

XX:G1MixedGC LiveThresholdPercent(默认85%)进行设置，从而拦截那些回收开销巨大的对象；同时，每次混合收集可以包含候选老年代分区，可根据CSet对堆的总大小占比-

XX:G1OldCSetRegionThresholdPercent(默认10%)设置数量上限。

由上述可知，G1的收集都是根据CSet进行操作的，年轻代收集与混合收集没有明显的不同，最大的区别在于两种收集的触发条件。

栅栏 Barrier

我们首先介绍一下栅栏(Barrier)的概念。栅栏是指在原生代码片段中，当某些语句被执行时，栅栏代码也会被执行。而G1主要在赋值语句中，使用写前栅栏(Pre-Write Barrier)和写后栅栏(Post-Write Barrier)。事实上，写栅栏的指令序列开销非常昂贵，应用吞吐量也会根据栅栏复杂度而降低。

写前栅栏 Pre-Write Barrier

即将执行一段赋值语句时，等式左侧对象将修改引用到另一个对象，那么等式左侧对象原先引用的对象所在分区将因此丧失一个引用，那么JVM就需要在赋值语句生效之前，记录丧失引用的对象。JVM并不会立即维护RSet，而是通过批量处理，在将来RSet更新(见SATB)。

写后栅栏 Post-Write Barrier

当执行一段赋值语句后，等式右侧对象获取了左侧对象的引用，那么等式右侧对象所在分区的RSet也应该得到更新。同样为了降低开销，写后栅栏发生后，RSet也不会立即更新，同样只是记录此次更新日志，在将来批量处理(见Concurrence Refinement Threads)。

起始快照算法 Snapshot at the beginning (SATB)

Taiichi Tuasa贡献的增量式完全并发标记算法起始快照算法(SATB)，主要针对标记-清除垃圾收集器的并发标记阶段，非常适合G1的分区块的堆结构，同时解决了CMS的主要烦恼：重新标记暂停时间长带来的潜在风险。

SATB会创建一个对象图，相当于堆的逻辑快照，从而确保并发标记阶段所有的垃圾对象都能通过快照被鉴别出来。当赋值语句发生时，应用将会改变了它的对象图，那么JVM需要记录被覆盖的对象。因此写前栅栏会在引用变更前，将值记录在SATB日志或缓冲区中。每个线程都会独占一个SATB缓冲区，初始有256条记录空间。当空间用尽时，线程会分配新的SATB缓冲区继续使用，而原有的缓冲去则加入全局列表中。最终在并发标记阶段，并发标记线程(Concurrent Marking Threads)在标记的同时，还会定期检查和处理全局缓冲区列表的记录，然后根据标记位图分片的标记位，扫描引用字段来更新RSet。此过程又称为并发标记/SATB写前栅栏。

三色标记算法

提到并发标记，我们不得不了解并发标记的三色标记算法。它是描述追踪式回收器的一种有用的方法，利用它可以推演回收器的正确性。首先，我们将对象分成三种类型的。

黑色:根对象，或者该对象与它的子对象都被扫描

灰色:对象本身被扫描,但还没扫描完该对象中的子对象

白色:未被扫描对象，扫描完成所有对象之后，最终为白色的为不可达对象，即垃圾对象

当GC开始扫描对象时，按照如下图步骤进行对象的扫描：

根对象被置为黑色，子对象被置为灰色。

继续由灰色遍历,将已扫描了子对象的对象置为黑色。

遍历了所有可达的对象后，所有可达的对象都变成了黑色。不可达的对象即为白色，需要被清理。

这看起来很美好，但是如果在标记过程中，应用程序也在运行，那么对象的指针就有可能改变。这样的话，我们就会遇到一个问题：对象丢失问题。

可行的方式：

1. 在插入的时候记录对象
2. 在删除的时候记录对象

刚好这对应CMS和G1的2种不同实现方式：

在CMS采用的是增量更新（Incremental update），只要在写屏障（write barrier）里发现要有一个白对象的引用被赋值到一个黑对象的字段里，那就把这个白对象变成灰色的。即插入的时候记录下来。

在G1中，使用的是STAB（snapshot-at-the-beginning）的方式，删除的时候记录所有的对象，它有3个步骤：

- 1，在开始标记的时候生成一个快照图标记存活对象
- 2，在并发标记的时候所有被改变的对象入队（在write barrier里把所有旧的引用所指向的对象都变成非白的）
- 3，可能存在游离的垃圾，将在下次被收集

这样，G1到现在可以知道哪些老的分区可回收垃圾最多。当全局并发标记完成后，在某个时刻，就开始了Mix GC。这些垃圾回收被称作“混合式”是因为他们不仅仅进行正常的新生代垃圾收集，同时也回收部分后台扫描线程标记的分区。

混合式GC也是采用的复制的清理策略，当GC完成后，会重新释放空间。

[回到顶部](#)

最佳实践、参数

当你使用G1时，你应该遵循的一些最佳实践。

- 不用设置young区大小。通过 -Xmn设置young区的大小会干预G1回收的默认行为。设定young大小会导致G1设定的停顿时间目标失效。G1将不能按需去扩缩young区。

- 响应时间指标。不要用平均的响应时间来设置 `XX:MaxGCPauseMillis=<N>`，设置一个满足的90%目标的时间。这意味着90%的用户请求将不经历过长的响应时间。记住预设停顿时间只是个目标，不是都能被满足的。
- Evacuation Failure: 当没有足够的空间供存活对象或者晋升对象用的时候，会发生晋升失败。这时如果用 `-XX:+PrintGCDetails` 开启GC log，会打印出 `to-space overflow`。这时GC 仍然继续，以求释放空间，没有被copy成功的对象，将会直接放在老年区，在要收集区里的所有对RSet的更新都要被重新计算。这些步骤开销都比价大。
- 如何避免evacuation failure: 加大堆的大小；加大 `-XX:G1ReservePercent=n`，来为'to-space'预留更多空间；增加marking线程数，通过 `-XX:ConcGCThreads=n` 选项；

触发Full GC

在某些情况下，G1触发了Full GC，这时G1会退化使用Serial收集器来完成垃圾的清理工作，它仅仅使用单线程来完成GC工作，GC暂停时间将达到秒级别的。整个应用处于假死状态，不能处理任何请求，我们的程序当然不希望看到这些。那么发生Full GC的情况有哪些呢？有三个

(1) 并发模式失败

G1启动标记周期，但在Mix GC之前，老年代就被填满，这时候G1会放弃标记周期。这种情形下，需要增加堆大小，或者调整周期（例如增加线程数 `-XX:ConcGCThreads`等）。

(2) 晋升失败或者疏散失败

G1在进行GC的时候没有足够的内存供存活对象或晋升对象使用，由此触发了Full GC。可以在日志中看到(to-space exhausted)或者 (to-space overflow)。解决这种问题的方式是：

a,增加 `-XX:G1ReservePercent` 选项的值（并相应增加总的堆大小），为“目标空间”增加预留内存量。

b,通过减少 `-XX:InitiatingHeapOccupancyPercent` 提前启动标记周期。

c,也可以通过增加 `-XX:ConcGCThreads` 选项的值来增加并行标记线程的数目。

(3) 巨型对象分配失败

当巨型对象找不到合适的空间进行分配时，就会启动Full GC，来释放空间。这种情况下，应该避免分配大量的巨型对象，增加内存或者增大 `-XX:G1HeapRegionSize`，使巨型对象不再是巨型对象。

参数

Option and Default Value	Description
<code>-XX:+UseG1GC</code>	开启G1回收器
<code>-XX:MaxGCPaus</code>	设定最大的GC停顿时间。这是一个软目标，JVM将近最大限度去实现。因此，这个值停顿时间可能不会被满足

eMillis=n	此选项只适用于可移植虚拟机。
- XX:InitiatingHeapOccupancyPercent=n	开始并发GC的百分比。G1 通过该比例来触发GC，这个比例是基于全部的heap,而不是单单某个分区。默认值是45%。
- XX:NewRatio=n	Ratio of new/old generation sizes. The default value is 2.
- XX:SurvivorRatio=n	Ratio of eden/survivor space size. The default value is 8.
- XX:MaxTenuringThreshold=n	Maximum value for tenuring threshold. The default value is 15.
- XX:ParallelGCThreads=n	设置并行回收阶段，GC所要用到的线程数。默认值随着JVM所运行的平台不同而变化。
- XX:ConcGCThreads=n	设置GC并发的线程数。默认值随着JVM所运行的平台不同而变化。
- XX:G1ReservePercent=n	加大-XX:G1ReservePercent=n,来为 'to-space'预留更多空间，防止晋升失败。默认值是10。
- XX:G1HeapRegionSize=n	指定region的大小。默认值是根据所开堆空间大小来计算的。region最大值是32Mb,最小值是1Mb。