垃圾回收GC

垃圾回收器的作用

- 垃圾回收器复制JVM堆内存管理和垃圾回收

垃圾回收算法

- 标记清除算法（Mark and Sweep）
  - 能解决循环引用问题，且只需扫描部分对象，且空间利用率高，但是存在内存碎片
- 标记复制算法 (Mark-Copy)
  - 采用复制方式把活跃对象引动到另外的区域，没有内存碎片，但是需要额外的空间进行复制
- 标记清除整理算法 (Mark-Sweep-Compact)
  - 弥补了标记清除算法的缺点，通过清除后压缩整理内存消除了内存碎片。但需要消耗更多的时间

Marking（标记）：从根对象（方法参数、局部变量、对象静态属性、活动线程、JNI 引用）出发标记可达的对象，并在本地内存(native)中分门别类记下。
Sweeping（清除）：这一步保证了，不可达对象所占用的内存，在之后进行内存分配时可以重用。

垃圾回收器

串行GC

Serial GC

- 采用算法
  - 串行 GC 对年轻代使用 mark-copy（标记-复制） 算法，对老年代使用 mark-sweep-compact（标记-清除-整理）算法
- 开启参数
  - -XX:+UseSerialGC
- 使用场景
  - 只适合几百 MB 堆内存的 JVM，而且是单核 CPU 时比较有用

并行GC

Parallel GC

- 主要目的
  - 增加吞吐量。因为对系统资源的有效使用，能达到更高的吞吐量：在 GC 期间，所有 CPU 内核都在并行清理垃圾，所以总暂停时间更短。在两次 GC 周期的间隔期，没有 GC 线程在运行，不会消耗任何系统资源
- 采用算法
  - 在年轻代使用 标记-复制（mark-copy）算法，在老年代使用 标记-清除-整理（mark-sweep compact）算法
- 开启参数
  - -XX:+UseParallelGC -XX:+UseParallelOldGC
- 设置参数
  - -XX：ParallelGCThreads=N 来指定 GC 线程数， 其默认值为 CPU 核心数
- 使用场景
  - 保证高效的垃圾收集和系统吞吐量的场景

并发GC

CMS GC

- 主要目的
  - 避免在老年代垃圾收集时出现长时间的卡顿。实现方式：不对老年代进行整理，而是使用空闲列表（free-lists）来管理内存空间的回收。在 mark-and-sweep （标记-清除） 阶段的大部分工作和应用线程一起并发执行。但值得注意的是，它仍然和应用线程争抢CPU 时间。默认情况下，CMS 使用的并发线程数等于 CPU 核心数的 1/4
- 垃圾回收阶段
  - Initial Mark（初始标记）
    - 这个阶段伴随着 STW 暂停。初始标记的目标是标记所有的根对象，包括根对象直接引用的对象，以及被年轻代中所有存活对象所引用的对象（老年代单独回收）
  - Concurrent Mark（并发标记）
    - 在此阶段，CMS GC遍历老年代，标记所有的存活对象，从前一阶段 “Initial Mark” 找到的根对象开始算起。 “并发标记”阶段，就是与应用程序同时运行，不用暂停的阶段。
  - Concurrent Preclean（并发预清理）
    - 此阶段同样是与应用线程并发执行的，不需要停止应用线程。 因为前一阶段【并发标记】与程序并发运行，可能有一些引用关系已经发生了改变。如果在并发标记过程中引用关系发生了变化，JVM 会通过“Card（卡片）”的方式将发生了改变的区域标记为“脏”区，这就是所谓的 卡片标记（Card Marking）。
  - Final Remark（最终标记）
    - 最终标记阶段是此次 GC 事件中的第二次（也是最后一次）STW 停顿。本阶段的目标是完成老年代中所有存活对象的标记. 因为之前的预清理阶段是并发执行的，有可能 GC 线程跟不上应用程序的修改速度。所以需要一次STW 暂停来处理各种复杂的情况。通常 CMS 会尝试在年轻代尽可能空的情况下执行 Final Remark 阶段，以免连续触发多次 STW 事件。
  - Concurrent Sweep（并发清除）
    - 此阶段与应用程序并发执行，不需要 STW 停顿。JVM 在此阶段删除不再使用的对象，并回收他们占用的内存空间。
  - Concurrent Reset（并发重置）
    - 此阶段与应用程序并发执行，重置 CMS 算法相关的内部数据，为下一次 GC 循环做准备
- 开启参数
  - -XX:+UseConcMarkSweepGC
- 使用场景
  - 服务器是多核 CPU，并且主要调优目标是降低GC停顿导致的系统延迟

G1 GC

- 主要目的
  - 将STW停顿的时间和分布，变成可预期且可配置的
- 特点
  - 堆不再分成年轻代和老年代，而是划分为多个（通常是2048个）可以存放对象的 小块堆区域(smaller heap regions)。每个小块，可能一会被定义成 Eden 区，一会被指定为 Survivor区或者Old区。在逻辑上，所有的Eden区和Survivor区合起来就是年轻代，所有的Old区拼在一起那就是老年代。这样划分之后，使得 G1不必每次都去收集整个堆空间，而是以增量的方式来进行处理: 每次只处理一部分内存块，称为此次GC的回收集(collection set)。每次GC暂停都会收集所有年轻代的内存块，但一般只包含部分老年代的内存块。G1的另一项创新是，在并发阶段估算每个小堆块存活对象的总数。构建回收集的原则是：垃圾最多的小块会被优先收集。这也是G1名称的由来。
- 开启参数
  - -XX:+UseG1GC
- 设置参数
  - -XX:MaxGCPauseMillis=50
    - 预期G1每次执行GC操作的暂停时间，单位是毫秒，默认值是200毫秒，G1会尽量保证控制在这个范围内
  - -XX:G1NewSizePercent
    - 初始年轻代占整个Java Heap的大小，默认值为5%
  - -XX:G1MaxNewSizePercent
    - 最大年轻代占整个Java Heap的大小，默认值为60%
  - -XX:G1HeapRegionSize
    - 设置每个Region的大小，单位MB，需要为1，2，4，8，16，32中的某个值，默认是 堆内存的1/2000。如果这个值设置比较大，那么大对象就可以进入Region了
  - -XX:ConcGCThreads
    - 与Java应用一起执行的GC线程数量，默认是Java线程的1/4
  - -XX:+InitiatingHeapOccupancyPercent
    - G1内部并行回收循环启动的阈值，默认为Java Heap的45%。这个可以理解为老年代使用大于等于45%的时候，JVM会启动垃圾回收。这个值非常重要，它决定了在什么 时间启动老年代的并行回收
  - -XX:G1HeapWastePercent
    - G1停止回收的最小内存大小，默认是堆大小的5%。GC会收集所有的Region中的对 象，但是如果下降到了5%，就会停下来不再收集了。就是说，不必每次回收就把所有的垃圾都处理完，可以遗留 少量的下次处理，这样也降低了单次消耗的时间
  - -XX:G1MixedGCCountTarget
    - 设置并行循环之后需要有多少个混合GC启动，默认值是8个。老年代Regions的回 收时间通常比年轻代的收集时间要长一些。所以如果混合收集器比较多，可以允许G1延长老年代的收集时间
  - -XX:+G1PrintRegionLivenessInfo
    - 这个参数需要和 -XX:+UnlockDiagnosticVMOptions 配合启动，打印JVM的调试信息，每个Region里的对象存活信息
  - -XX:G1ReservePercent
    - G1为了保留一些空间用于年代之间的提升，默认值是堆空间的10%。因为大量执行回收的地方在年轻代（存活时间较短），所以如果你的应用里面有比较大的堆内存空间、比较多的大对象存活，这里需要保留一些内存
  - -XX:+G1SummarizeRSetStats
    - 这也是一个VM的调试信息。如果启用，会在VM退出的时候打印出RSets的详细总结信息。如果启用-XX:G1SummaryRSetStatsPeriod参数，就会阶段性地打印RSets信息
  - -XX:+G1TraceConcRefinement
    - 这个也是一个VM的调试信息，如果启用，并行回收阶段的日志就会被详细打印出来
  - -XX:+GCTimeRatio
    - 这个参数就是计算花在Java应用线程上和花在GC线程上的时间比率，默认是9，跟新生代内存的分配比例一致。这个参数主要的目的是让用户可以控制花在应用上的时间，G1的计算公式是100/（1+GCTimeRatio）。这样如果参数设置为9，则最多10%的时间会花在GC工作上面。Parallel GC的默认值是99，表示1%的时间被用在GC上面，这是因为Parallel GC贯穿整个GC，而G1则根据Region来进行划分，不需要全局性扫描整个内存堆
  - -XX:+UseStringDeduplication
    - 手动开启Java String对象的去重工作，这个是JDK8u20版本之后新增的参数，主要用于相同String避免重复申请内存，节约Region的使用
- 垃圾回收阶段
  - 处理步骤1：年轻代模式转移暂停（Evacuation Pause）
    - G1 GC会通过前面一段时间的运行情况来不断的调整自己的回收策略和行为，以此来比较稳定地控制暂停时间。在应用程序刚启动时，G1还没有采集到什么足够的信息，这时候就处于初始的 fully young 模式。当年轻代空间用满后，应用线程会被暂停，年轻代内存块中的存活对象被拷贝到存活区。如果还没有存活区，则任意选择一部分空闲的内存块作为存活区。拷贝的过程称为转移(Evacuation)，这和前面介绍的其他年轻代收集器是一样的工作原理。
  - 处理步骤2：并发标记（Concurrent Marking）
    - G1并发标记的过程与CMS基本上是一样的。G1的并发标记通过 Snapshot-At-The-Beginning(起始快照) 的方式，在标记阶段开始时记下所有的存活对象。即使在标记的同时又有一些变成了垃圾。通过对象的存活信息，可以构建出每个小堆块的存活状态，以便回收集能高效地进行选择。
      这些信息在接下来的阶段会用来执行老年代区域的垃圾收集。有两种情况是可以完全并发执行的:
      一、如果在标记阶段确定某个小堆块中没有存活对象，只包含垃圾;
      二、在STW转移暂停期间，同时包含垃圾和存活对象的老年代小堆块。
      当堆内存的总体使用比例达到一定数值，就会触发并发标记。这个默认比例是 45%，但也可以通过JVM参数 InitiatingHeapOccupancyPercent 来设置。和CMS一样，G1的并发标记也是由多个阶段组成，其中一些阶段是完全并发的，还有一些阶段则会暂停应用线程。
    - 阶段
      - 阶段 1: Initial Mark(初始标记)：此阶段标记所有从GC根对象直接可达的对象。
      - 阶段 2: Root Region Scan(Root区扫描)：此阶段标记所有从 "根区域" 可达的存活对象。根区域包括：非空的区域，以及在标记过程中不得不收集的区域。
      - 阶段 3: Concurrent Mark(并发标记)：此阶段和CMS的并发标记阶段非常类似：只遍历对象图，并在一个特殊的位图中标记能访问到的对象
      - 阶段 4: Remark(再次标记)：和CMS类似，这是一次STW停顿(因为不是并发的阶段)，以完成标记过程。 G1收集器会短暂地停止应用线程，停止并发更新信息的写入，处理其中的少量信息，并标记所有在并发标记开始时未被标记的存活对象
      - 阶段 5: Cleanup(清理)：最后这个清理阶段为即将到来的转移阶段做准备，统计小堆块中所有存活的对象，并将小堆块进行排序，以提升GC的效率，维护并发标记的内部状态。所有不包含存活对象的小堆块在此阶段都被回收了。有一部分任务是并发的：例如空堆区的回收，还有大部分的存活率计算。此阶段也需要一个短暂的STW暂停
  - 处理步骤3：转移暂停: 混合模式（Evacuation Pause (mixed)）
    - 并发标记完成之后，G1将执行一次混合收集（mixed collection），就是不只清理年轻代，还将一部分老年代区域也加入到 回收集 中。混合模式的转移暂停不一定紧跟并发标记阶段。有很多规则和历史数据会影响混合模式的启动时机。比如，假若在老年代中可以并发地腾出很多的小堆块，就没有必要启动混合模式。因此，在并发标记与混合转移暂停之间，很可能会存在多次 young 模式的转移暂停。具体添加到回收集的老年代小堆块的大小及顺序，也是基于许多规则来判定的。其中包括指定的软实时性能指标，存活性，以及在并发标记期间收集的GC效率等数据，外加一些可配置的JVM选项。混合收集的过程，很大程度上和前面的fully-young gc是一样的
- 注意事项
  - 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。解决办法：增加堆大小，或者 调整周期（例如增加线程数-XX:ConcGCThreads等）。
  - 晋升失败
    - 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(to-space exhausted/to-space overflow）。解决办法：
      a)增加 -XX:G1ReservePercent 选项的值（并相应增加总的堆大小）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
  - 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。解决办法：增加内存或者增大-XX:G1HeapRegionSize

特别需要注意的是，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

ZGC

- 主要特点
  - GC 最大停顿时间不超过 10ms
  - 堆内存支持范围广，小至几百 MB 的堆空间，大至4TB 的超大堆内存（JDK13升至16TB）
  - 与 G1 相比，应用吞吐量下降不超过15%
  - 当前只支持 Linux/x64 位平台，JDK15后支持MacOS和Windows系统

SGC

- Shenandoah GC立项比ZGC更早，设计为GC线程与应用线程并发执行的方式，通过实现垃圾回收过程的并发处理，改善停顿时间，使得GC执行线程能够在业务处理线程运行过程中进行堆压缩、标记和整理，从而消除了绝大部分的暂停时间。Shenandoah 团队对外宣称Shenandoah GC的暂停时间与堆大小无关，无论是200 MB 还是 200 GB的堆内存，都可以保障具有很低的暂停时间（注意:并不像ZGC那样保证暂停时间在10ms以内）。

GC选择一般指导原则

- 如果系统考虑吞吐优先，CPU资源都用来最大程度处理业务，用Parallel GC
- 如果系统考虑低延迟有限，每次GC时间尽量短，用CMS GC
- 如果系统内存堆较大，同时希望整体来看平均GC时间可控，使用G1 GC
- 一般4G以上，算是比较大，用G1的性价比较高。一般超过8G，比如16G-64G内存，非常推荐使用G1 GC

常用GC组合

- Serial+Serial Old实现单线程的低延迟垃圾回收机制
- ParNew+CMS，实现多线程的低延迟垃圾回收机制
- Parallel Scavenge和Parallel Scavenge Old，实现多线程的高吞吐量垃圾回收机制