GC标记算法

内存从小到大的过程产生了各种各样的GC

Serial + Serial Old　单线程　　GC线程执行的时候用户线程要停掉

Parallel Scavenge + Parallel Old　多线程　并行执行

PerNew　+ SMS　并发执行—(标记时间久的阶段)和并发回收是多线程, 初始标记和重新标记是并行

前面几个GC内存都是两块连续的（年轻+老年），分大块，不管是哪种GC都是要扫描整个内存，所以这个块变得很大的时候，无论是哪个GC，基本都没戏。所以诞生了G1

G1　是将内存分成一小块一小块，在某一块内存工作的时候，可以清理别的块内存。分而治之

ZGC　是对G1 的升级　每一小块内存更加灵活，有小的有大的

卡表（Card Table）：

主要用在分代模型中，帮助我们进行垃圾回收，垃圾回收速度比较快

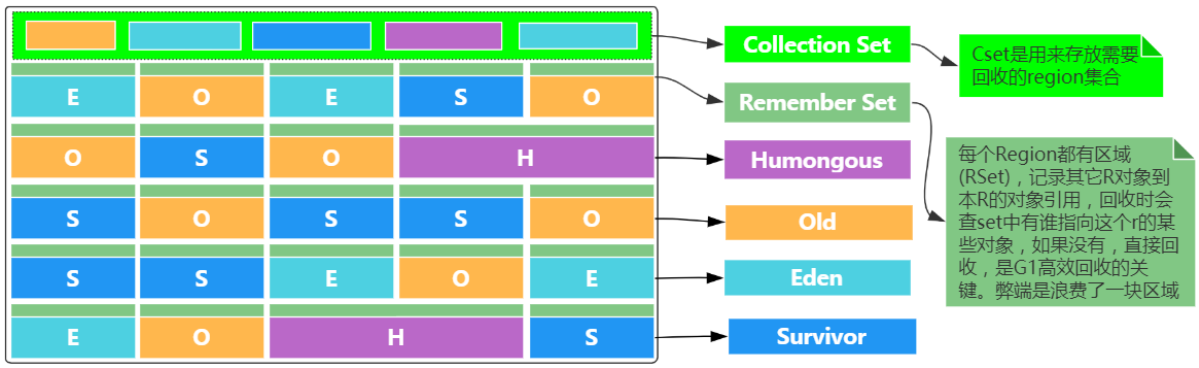
> 由于做YGC时，需要扫描整个OLD区，效率非常低，所以JVM设计了CardTable， 如果一个OLD区CardTable中有对象指向Y区，就将它设为Dirty，下次扫描时，只需要扫描Dirty Card,在结构上，Card Table用BitMap来实现

说明：

在JVM中去追踪活着的对象的时候并不容易，扫描的根对象很可能在Old区，比如一个根对象在Y区，然后下一个指向的对象在Old，Old区的这个对象又指向另一个Y区的对象。那么再找到Y区活着的这个对象的过程竟然要遍历整个的Old区，进行一次YGC竟然扫描一次Old区，这显然不合理，所以为了解决这个问题，JVM内部把整个内存分成了一个一个的card（Y 和O区都区分），具体的对象存在于一个一个card中，如果Old区域中某一个card中有指向Y区的对象，就把这个Card标记为 Dirty，说

明这个Card里面有指向Y区的对象，而在JVM中通过一个位图Bit Map (01001001001 的东西)来标识这个Dirty（一个card用一个 0或1来标识是否是Dirty的，0 标识没有指向Y的Dirty，1 表示是Dirty），所以这个东西叫card table，Y区不需要标记在CardTable

G1 GC　Garbage First Garbage Collector 垃圾优先GC



之前的GC都是两块连续的内存，每一次GC操作都是在两块大空间中进行的，空间太大，效率肯定不高。所以G1改变了内存布局（分而治之），G1 是将一大块大内存分成一块一块的Region，每一块region 在逻辑上是有分代,，分了四种（Old区，Survivor[放存活对象]，Eden区[放新生对象]，Humongous[大对象区域，这个区域可能会很大，有可能跨两个、三个连续的region]）

Humongous： 超过单个region的50%的region内存就称为大对象

入门参考：https://www.oracle.com/technical-resources/articles/java/g1gc.html

Garbage First 的意思就是里面存活对象最少的Region 就是垃圾最多的region，优先回收这样的Region

G1 是一种服务端应用使用的垃圾回收器，目标是用在多核、大内存的机器上，它在大多数情况下可以实现指定的GC暂停时间，同事还能保持较高的吞吐量

特点：

- 并发收集
- 压缩空闲空间不会延长GC暂停时间

- 更容易预测的GC暂停时间
- 适用不需要实现很高的吞吐量的场景

并发收集：CMS也是并发收集，G1和CMS的收集算法本质上没什么区别，都是三色标记，但是到了ZGC和Shenandoah 的算法就有本质上的区别了，算法是颜色指针(Colored Pointer)

能够控制GC的时间，也能预测GC时间

每个分区都可能是年轻代也可能是老年代，但是在同一时刻只能属于某个代

年轻代、幸存区、老年代这些概念还存在，成为逻辑上的概念，这样方便服用之前分代框架的逻辑。在物理上不需要连接，这样带来了额外的好处——有的分区内垃圾对象特别多，有的分区内垃圾对象很少，G1会优先回收垃圾对象特别多的分区，这样可以话费较少的时间来回收这些分区的垃圾，这也就是G1名字的由来——首先收集垃圾最多的分区。
新生代其实并不适用于这种算反，依然是在新生代满了的时候，对整个新生代进行回收——整个新生代中的对象要么被回收、要么晋升 ，至于新生代也采取分区机制的原因则是因为这样跟老年代的策略统一，方便调整代的大小。
G1 还是一种带压缩的收集器，在回收老年代分区时，是将存活的对象从一个分区拷贝到另一个可用分区，这个高倍过程就实现了局部的压缩
每个分区的大小从1M到32M不等，但是都是2的幂次方

在某个时间段，第一块内存是Eden区，在一次GC之后，下一次的时候很可能会变成Old区

CSet（Collection Set） G1 特定的概念

- Cset是用来存放需要回收的region集合
- 一组可被回收的分区的集合
- 在CSet中存货的数据会在GC过程中被移动大奥另一个可用分区
- CSet中的分区可以来自Eden、Survivor、Old
- CSet 会占用不到整个堆空间的1%的大小

RSet（Remembered Set） G1 特定的概念

- 每一个Region里面都有一个表格，一个区域（hashset），

- 记录着其它region的对象到本region的对象引用，在回收一块region的时候会查看这个set中有谁指向这个region的某些对象，如果没有，直接回收就行，
- 这个是G1高效回收的一个关键。弊端就是浪费了一块存取set的区域
- 价值是使得GC不需要扫描整个堆找到谁引用了当前分区的对象，只需要扫描RSet即可

G1 特点

- 追求吞吐量
- 追求响应时间
- G1 新老年代比例一般不需要手工指定（会自动优化）

  G1会跟踪每次STW，如果这一次STW时间（400ms）比较长，大于我设定的值（200ms），400ms才回收完整个的Y区，这样下次的时候会把Y区稍微调小一点（动态），自动优化

对象什么时候进入老年代

- 超过 XX:MaxTenuringThreshold 指定次数（YGC）

  PS默认15　　CMS默认6　　G1默认 15
- 动态年龄

  s1 → s2 超过50% 把年龄最大的放入Old区域

GC 何时触发

YGC：Eden 空间不足、多线程并行执行

FGC：Old空间不足、System.gc()

G1 有三种GC　YGC　MixedGC　FGC（jdk10 之前是Serial）

G1 是一定会产生FGC的，当分配的特别快，GC线程根本回收不过来，回收不完， 对象分配不下了，就会产生FGC

MixedGC

MixedGC 相当于CMS

MixedGC，当YGC已经不行了，对象的产生特别多，对象的的量已经到了堆内存的一个比例(可以设置)，超过这个值后默认会启动这个设置

XX:InitiatingHeapOccupacyPercent　默认45%，当O超过这个值时，启动MixedGC

MixedGC是G1 的正常回收过程，当达到这个比例时，就会启动MixedGC，年轻代也会回收，Old代也会回收，那个Region满了或者哪个Region垃圾最多，就先回收哪个Region，只有逻辑是区分Y和O的，物理不区分。没有到达这个比例的时候回收的都是Y区

MixedGC　的过程

- 出事标记 STW
- 并发标记
- 最终标记（重新标记）STW
- 筛选回收STW（并行）

  筛选过程：筛选哪些最需要回收的Region，垃圾占用的最多的哪些region，标记完后，将标记的有用的对象Copy压缩到新的区域，避免产生过多的内存碎片，将原来的region回收掉

G1　java 10 以前是串行的FullGC，之后是并行的FullGC

其中 并发标记 阶段会有漏标的问题，为解决这个问题，采用了 "三色标记算法"

并发标记算法：CMS 和 G1 的并发标记时刻

难点：在标记对象的过程中，对象引用关系正在发生改变

三色标记

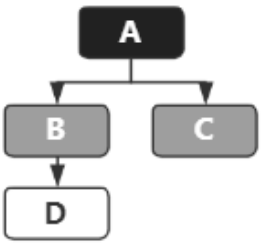
顺着root一直向下标记，标记到的说明都有对象引用，能找到，最后没有遍历到的就是垃圾

把对象在逻辑上分成三个不同的颜色，每个不同的颜色代表有没有标记过、标记到一半了、还是完全没有标记

- 白色：未标记
- 灰色：标记了一半（自身被标记、成员变量未被标记）
- 黑色：全部标记（自身和成员变量均已标记完成）

漏标：

由于并发标记和工作线程是同时进行了，A对象全部标记后，工作线程又把已经标记的A 又指向了D，原来B到D的这个指向没了，满足这两个条件的情况才会漏标。



在重新标记过程中，黑色指向了白色，如果不对黑色重新扫描，则会漏标，会把白色D对象当作没有新引用指向从而回收掉

并发标记过程中，删除了所有从灰色到白色的引用，会产生漏标，此时白色对象应该被回收

漏标是指：本来是可用对象，但是由于没有遍历到被当成Garbage回收掉了

解决漏标的思路（打破上述条件之一即可）：

1、只要能跟踪到A指向了D，这样就可以在下次扫描的时候重新扫描一遍就可以了，重新扫描后重新标记为灰色，找到了D这个对象———incremental update **CMS使用**

2、或者跟踪到到B到D的指向消失了，下次扫描的时候再扫描一次D也可以，扫描到D后找到A把A重新标记为灰色————- SATB

即 Incremental update 和 SATB：

- incremental update —— **CMS使用**

  增量更新，关注引用增加

  吧黑色的重新标记为灰色，下次重新扫描属性

- SATB —— **G1使用**

  Snapshot at the beginning —— 关注引用的删除

  当B→D消失时，要把这个引用推到GC的堆栈，保证D还能被GC扫描到

  G1 用到的算法，下次标记的时候会从GC堆栈中遍历引用找到对象D，重新标记A

  引用删除表示将该发生改变引用压到JVM堆栈里面，下一次GC的时候会从堆栈里面扫描变更的对象，然后根据这个对象D找到指向它的相关对象（通过RememberSet找关系），再进行标记即可。而增量更新中为变成灰色后需要对灰色的重新扫描，效率低

为什么G1 要用 SATB 不用 incremental update？

灰色→白色引用消失时，如果没有黑色指向白色引用会被push到堆栈，下次扫描时拿到这个引用，由于有RSet的存在，不需要扫描整个堆取查找指向白色的引用，效率比较高，SATB配合RSet，浑然天成

因为 incremental update 增量更新，会将原来已经扫描过后的对象成员重新扫描一遍，效率很低，STAB只需要关注更改过的对象即可，扫描次数变少。

颜色指针——ZGC算法

一个指针在JVM中如果没有压缩的话，占用8个字节64位，会从这64个bit中拿出来3个标记这个对象指针的变化。如果变过了，在进行GC的时候，会扫描变化过的指针，所以叫颜色指针。