方法区

方法区的回收主要有两部分

- 废弃常量
- 无用类

类回收需要满足3个条件

- 该类所有的实例都已经被GC，也就是说JVM中不存在该Class实例的任何实例
- 加载该类的Class Loader 也已经被回收
- 该类对应的Class对象没有在任何地方被引用，就是说没有在任何地方有A.class类似的操作

可见方法区的回收条件是非常苛刻的。

GC

常见GC算法

1. 标记——整理
2. 标记——清除
3. 复制
4. 分代

GC的时机

1. Scavenge GC(Minor GC)。触发时机：新对象生成时，Eden空间满了。
2. Full GC。对整个JVM进行清理，包括Young、old、perm。触发时机：①Old满了②Perm满了③system.gc()。效率低，尽量减少Full GC。

Hotspot中，虚拟机栈和本地方法栈是一个，可以通过-Xss控制

JVM参数

-Xms 堆的初始值
-Xmx 堆的最大值
-Xmn 堆的新生代大小
-XX:PrintGCDetails 打印垃圾回收的详细信息
-XX:SurvivorRatio=8 Eden:Survivor

大对象直接进老年代

-verbose:gc
-Xms20M
-Xmx20M
-Xmn10M
-XX:PrintGCDetails
-XX:SurvivorRatio=8

-XX:PretenureSizeThreshold=4194304//大对象直接进老年代
-XX:UseSerialGC

晋升至老年代的最大年龄
//MaxTenuringThreshold 作用：在可以自动调节的对象晋升至老年代的GC中，设置
//该晋升年龄的最大值。注意，JVM会根据需要自动调整阈值，但不会超过设置的最大值
//该参数在CMS中默认为6，在G1中默认为15
-XX:MaxTenuringThreshold=5
-XX:+PrintTenuringDistribution

对象已死?

引用计数：对象中添加一个引用计数器，有地方引用时就+1，引用失效时就-1，计数器为0的时候就没有被引用，可以回收。此方式，实现简单，判定效率高，但无法解决循环引用问题。

GCRoot：通过判断GC Root 根是否可达，执行时**会STW**。可作为GC Root 根的有：

- 栈帧中局部变量表中的对象
- **方法区中常量引用的对象**
- **方法区中类静态属性引用的对象**
- 本地方法栈中引用的对象

每次GC时不需要都遍历引用，而是把信息存在OopMap里。

安全点

主动式中断：设置标志，和安全点重合，各个线程主动轮询这个标志，为真则挂起。
抢占式中断：GC发生时，先把所有的线程中断掉，如果有线程没在安全点上，恢复线程。

并行：指多个收集器同时工作，但是用户线程处于等待状态。
并发：指收集器在工作时，允许用户线程运行。

内存分配策略

- 优先在Eden分配。
- 大对象直接进入老年代。**-XX:PretenureSizeThreshold=4194304**
- 长期存活对象进入老年代。年龄超过设置，进入老年代。**-XX:MaxTenuringThreshold=5**
- **空间担保分配进入老年代。在minor GC前会检查老年代的最大连续空间，如果空间小于新生代对象所占用的空间，那么这次minor GC是有风险的。jvm会查看HandlePromotionFailure，如果jvm允许担保失败，则老年代会计算历次晋升至**

老年代对象大小的平均大小，如果最大连续空间大于平均值，那么会尝试进行一次minor GC，如果小于，则进行一次Full GC。

这里风险是指，极端情况下，新生代 Minor GC后，存在大量的存活对象，把Survivor无法容纳的对象直接担保进老年代。

CMS

Concurrent Mark Sweep，基于 标记——清除 算法，只能在老年代使用，以最短回收停顿为目标的收集器。

1. 初始标记 initial mark **(STW)**
2. 并发标记 Concurrent mark
3. Concurrent Abortable Preclean
4. Final mark **(STW)。将并发标记过程中，由于用户线程把本GC ROOT不可达的的对象变得可 达了的对象，重新标记为可达。**
5. Concurrent Sweep
6. Concurrent Reset

优点：并发收集，将大量工作分散到并发处理阶段，STW时间短。

缺点：

① 对CPU资源非常敏感。

② 无法处理浮动垃圾。并发标记的过程中，由于用户线程的执行，，会产生新的垃圾， 而这些垃圾这能等到下一次GC的时候才能进行清理。这些垃圾被称为浮动垃圾。在并发标记过程中，用户线程还在继续，所以应该预留出足够的空间给用户线程使用，可以通过适当调高参数 -XX:CMSInitiatingOccupancyFraction 来提高触发百分比，。如果运行期间，预留的内存无法满足用户线程，会出现 Concurrent Mode Failure，会启用后备预案：Serial Old，这样停顿时间就很长了。所以说参数 -XX:CMSInitiatingOccupancyFraction 设置太高很容易导致大量的Concurrent Mode Failure，从而导致长时间停顿。

③ 结束时会产生大量的碎片。由于CMS基于标记——清除算法，会产生空间碎片。当老年代无法找到足够大的连续空间时，不得不触发一次 Full GC。

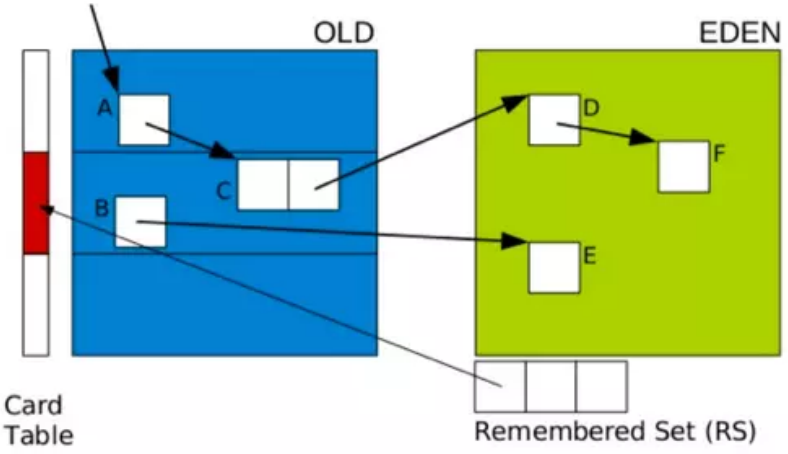
G1

RSet(已记忆集合)

Remembered Set。G1采用的时coping算法，需要移动对象，所以需要更新对象的引用地址，在普通的分代收集中也是如此，在年轻代收集时需要更新老年代到年轻代的指向，此

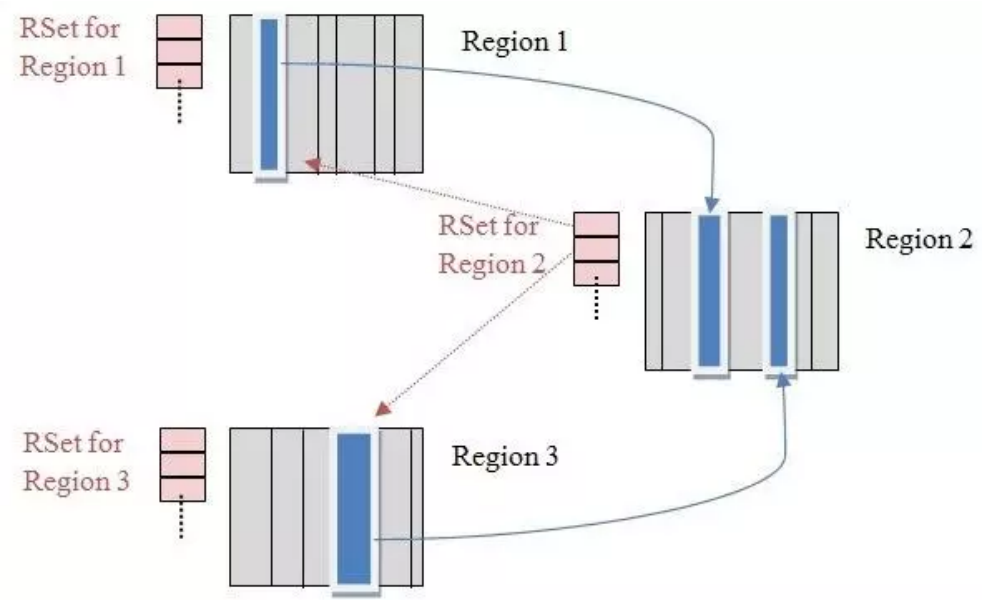
时需要Remembered Set(简称RS)来记录这个信息。CardTable是一种RS，每个Region都有自己的CardTable。维护CardTable需要mutator线程在可能修改跨Region引用的时候通知Collection，这种方式叫做write barrier。每个线程都有自己的RS log， 相当于各自修改的card的缓冲buffer，除此之外还有全局的buffer， mutator自己的remember set buffer满了之后会放入到全局buffer中，然后创建一个新的buffer。

如果仅仅GC 新生代对象，我们如何找到所有的根对象呢？ 老年代的所有对象都是根么？那这样扫描下来会耗费大量的时间。于是，G1引进了RSet的概念。它的全称是Remembered Set，作用是跟踪指向某个heap区内的对象引用。



全称是Remembered Set，是辅助GC过程的一种结构，典型的空间换时间工具，和Card Table有些类似。还有一种数据结构也是辅助GC的：Collection Set（CSet），它记录了GC要收集的Region集合，集合里的Region可以是任意年代的。在GC的时候，对于old->young和old->old的跨代对象引用，只要扫描对应的CSet中的RSet即可。

逻辑上说每个Region都有一个RSet，RSet记录了其他Region中的对象引用本Region中对象的关系，属于points-into结构（谁引用了我的对象）。而Card Table则是一种points-out（我引用了谁的对象）的结构，每个Card 覆盖一定范围的Heap（一般为512Bytes）。G1的RSet是在Card Table的基础上实现的：每个Region会记录下别的Region有指向自己的指针，并标记这些指针分别在哪些Card的范围内。 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。
下图表示了RSet、Card和Region的关系：



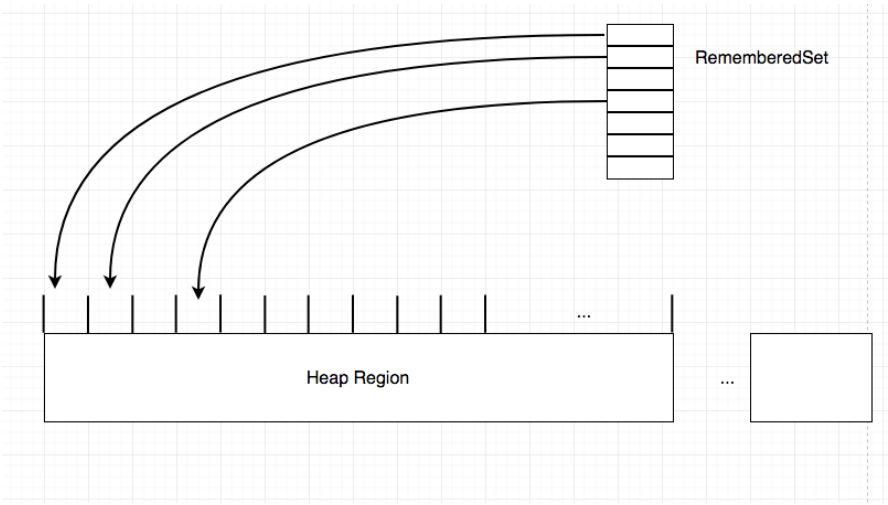
上图中有三个Region，每个Region被分成了多个Card，在不同Region中的Card会相互引用，Region1中的Card中的对象引用了Region2中的Card中的对象，蓝色实线表示的就是points-out的关系，而在Region2的RSet中，记录了Region1的Card，即红色虚线表示的关系，这就是points-into。

RSet究竟是怎么辅助GC的呢？在做YGC的时候，只需要选定young generation region的RSet作为根集，这些RSet记录了old->young的跨代引用，避免了扫描整个old generation。 而mixed gc的时候，old generation中记录了old->old的RSet，young->old的引用由扫描全部young generation region得到，这样也不用扫描全部old generation region。所以RSet的引入大大减少了GC的工作量。

CardTable

因为G1只回收一部分Region，所以回收的时候需要知道哪些其他Region的对象引用着自己Region的对象，因为采用的copying算法需要移动对象，所以要更新引用为对象的新地址，在普通的分代收集中也是如此，分代收集中年轻代收集需要老年代到年轻代的引用的记录，通常叫做remembered set(简称RS)。CardTable是一种remembered set，一个card代表一个范围的内存，目前采用512bytes表示一个card，cardtable就是一个byte数组，每个Region有自己的cardtable。维护remembered set需要mutator线程在可能修改跨Region的引用的时候通知collector，这种方式通常叫做write barrier(和GC中的Memory Barrier不同)，每个线程都会有自己的remembered set log,相当于各自的修改的card的缓冲

buffer,除此之外还有全局的buffer, mutator自己的remember set buffer满了之后会放入到全局buffer中，然后创建一个新的buffer。



只有来自其他Region的引用需要记录在RS中，所以Region内部的引用和null都不需要记录RS。

Humongous区域

YGC

当Eden区域无法申请新的对象时（满了），就会进行Young GC, Young GC将Eden和Survivor区域的Region(称为Collection Set, CSet)中的活对象Copy到一些新Region中(即新的Survivor)，当对象的GC年龄达到阈值后会Copy到Old Region中。由于采取的是Copying算法，所以就避免了内存碎片的问题，不再需要单独的压缩。

Young GC：选定所有年轻代里的Region。通过控制年轻代的region个数，即年轻代内存大小，来控制young GC的时间开销。

- 扫描
- 更新RS
- 处理RS Process Buffer
- 复制对象
- 处理引用队列 Termination

Mixed GC

当old区Heap的对象占总Heap的比例超过InitiatingHeapOccupancyPercent之后，就会开始ConcurentMarking, 完成了Concurrent Marking后，G1会从

Young GC切换到Mixed GC, 在Mixed GC中，G1可以增加若干个Old区域的Region到CSet中。
Mixed GC的次数根据候选的Old CSet和每次回收的。 Mixed GC指的不是一次GC，而是一个过程。

当old区Heap的对象占总Heap的比例超过InitiatingHeapOccupancyPercent之后，就会开 始ConcurentMarking, 完成了Concurrent Marking后，G1会从Young GC切换到Mixed GC, 在Mixed GC中，G1可以增加若干个Old区域的Region到CSet中。
Mixed GC的次数根据候选的Old CSet和每次回收的。

当堆空间占有率达到某一阈值，G1会启动一个独占的全局并发标记 global concurrent marking。全局并发标记包括以下几个阶段：

- **Initial Mark。暂停所有应用线程(STW)。它标记了从GC Root开始直接可达的对象。共用了Young GC的暂停，这是因为他们可以复用root scan操作，所以可以说global concurrent marking是伴随Young GC而发生的。**
- **Concurrent Marking。这个阶段从GC Root开始对heap中的对象标记，标记线程与应用程序线程并行执行，并且收集各个Region的存活对象信息。**
- **Remark。STW。标记那些在并发标记阶段发生变化的对象，将被回收。**
- **Cleanup。marking的最后一个阶段，G1统计各个Region的活跃性，完全没有存活对象的Region直接放入空闲可用Region列表中，然后会找出mixed GC的Region候选列表。清除空Region（没有存活对象的），加入到free list。只是回收了没有存活对象的Region，所以它并不需要STW。**

全局并发标记之后不一定会发生mixed gc。G1HeapWastePercent：在global concurrent marking结束之后，我们可以知道old gen regions中有多少空间要被回收，在每次YGC之后和再次发生Mixed GC之前，会检查垃圾占比是否达到此参数，只有达到了，下次才会发生Mixed GC。
G1MixedGCLiveThresholdPercent：old generation region中的存活对象的占

比，只有在此参数之下，才会被选入CSet。G1MixedGCCountTarget：一次global concurrent marking之后，最多执行Mixed GC的次数。G1OldCSetRegionThresholdPercent：一次Mixed GC中能被选入CSet的最多old generation region数量。

阶段统计得出收集收益高的若干老年代 Region。在用户指定的开销目标范围内，尽可能选择收益高的老年代Region进行GC，通过选择哪些老年代Region和选择多少Region来控制Mixed GC开销

Mixed GC 不仅进行新生代的GC，也会进行标记的老年代的回收。
它的GC步骤分为两步

- global concurrent marking
- evacuation（拷贝）

SATB 三色标记算法

- **黑色：根对象或该对象与其子对象都被扫描过。**
- **灰色：该对象已被扫描过，但其子对象还没有被扫描。**
- **白色：未被扫描过的对象。扫描完成后，白色对象为不可达的垃圾对象。**

GC过程中的引用改变问题：通过write barrier。
漏标情况只会发生在白色对象中，且满足以下任一条件：

- 并发标记时，黑色对象的引用指向一个白色的对象。因为，黑色对象已经被扫描过，不会再进行对其子对象的向下的扫描，导致这个白色对象被回收，发生错误。
- 并发标记阶段，应用线程删除了**所有**灰色对象对某一白色对象的引用。原因：取消引用之后，如果有一个黑色对象引用了该白色对象，那么该白色对象就不会被扫描到，而jvm会回收掉该白对象。

从gray对象引用移除的对象标为gray，black中新引用(刚new出来的)的对象标记为black。

SATB精度较低，会产生浮动垃圾，只能等到下次回收。

停顿模型预测

- 通过 -XX:MaxGCPauseMillis 参数设置，但并不是越短越好。
- 设置时间越短意味着每次回收的CSet越小，导致垃圾回收不及时越堆越多，**最终不得不退化为Serail GC**。停顿时间越长，则会影响程序的对外响应时间。

Evacuation Failure

Evacuation Failure 类似于CMS里的晋升老年代失败。对空间的垃圾太多无法完成 Region 之间的拷贝，于是不得不进行一次FULL GC做一次全局范围内的垃圾收集。