# GC回收

## 学习目标

GC分类(MinorGC、FullGC)、触发条件、MinorGC流程

无用对象判断方法、GCRoot对象

GC算法：复制、标记清理、标记整理

GC收集器：

CMS（标记清理算法）：初始标记、追溯标记、重新标记、并发清理

G1（复制+标记整理）：初始标记、追溯标记、重新标记、并发清理

CMS与G1区别

## 概念相关

相关链接：

http://www.cnblogs.com/sunada2005/p/3577799.html

由于栈区的内存是随着线程而释放的，因此垃圾回收主要针对的是堆区的回收。堆区被划分成两个不同的区域：新生代Young和老年代Old，其比例为1：2（可通过-XX:NewRatio指定）。新生代又被划分成三个区域：Eden、From Survivor、To Survivor。其比例为Eden：From：To=8:1:1（可通过-XX:SurvivorRatio指定）。

针对新生代的垃圾回收称为Minor GC，针对老年代的垃圾回收称为Major GC。当老年代满后，会触发FullGC，针对整个堆（包括新生代、老年代、持久代）进行垃圾回收。

新生代GC：简化了新对象的分配(只在新生代分配内存)。使用的是复制回收算法，每次使用Eden区和一个幸存区（FromSurvivor），回收时将其还存活的对象一次性复制到另一幸存区（ToSurvivor），最后清理刚才使用的Eden和幸存区。

老年代GC：在新生代中经历了多次MinorGC后仍然存活的对象，就会被放到老年代中。因此可以认为老年代中存放的都是一些生命周期较长的对象。

## GC什么时候开始？

1. 对象优先在新生代的Eden区中分配，当Eden区满后，发生一次MinorGC:

-1）Eden区幸存对象+From Survivor区存储的对象被复制到To Survivor区；

-2）清空Eden区和From Survivor区；

-3）颠倒From Survivor和To Survivor的逻辑关系：From变To，To变From

附：因为Java大多数对象都是朝生夕灭，所以Minor GC非常频繁，而且速度也很快。

1. 老年代也满后，会触发FullGC（MinorGC+MajorGC），针对整个堆（包括新生代、老年代、持久代）进行垃圾回收。

## MinorGC、MajorGC和FullGC

MinorGC发生在新生代，采用复制算法；

MajorGC发生在老年代，一般也伴随着MinorGC，它采用标记-清除算法。发生MajorGC频率小于MinorGC，速度比MinorGC慢10倍以上。

当新生代Eden区没有足够空间进行分配时，虚拟机将发起一次MinorGC；老年代空间不足时发生FullGC(MajorGC+MinorGC)，如果还不足则抛出OOM异常。

MinorGC触发条件：

当Eden区满时触发。

FullGC触发条件：

触发Full GC时，将针对整个堆（包括新生代、老年代、持久代）进行垃圾回收。

-1）老年代满了

-2）方法区满了

-3）通过MinorGC后进入老年代的平均大小大于老年代剩余空间大小

-4）由Eden区、From区向To区复制时，对象大小大于To可用内存，则把该对象转存到老年代，但老年代的可用内存小于该对象的大小。

注：程序中主动调用System.gc()强制执行的GC是Full GC

## 如何判断是否为无用对象？

1）引用计数法

为每个创建的对象分配一个引用计数器，用来存储该对象被引用的次数。

该对象的引用增加则+1，减少则-1。当引用数为0时，表示该对象不可达，GC可以回收掉这些引用次数为0的对象。

缺点：两个对象存在互相引用，使得它们的引用次数都不为0，从而被认定不可回收。这也是Java没有采用它的原因。

2）可达性分析算法

通过“GC Roots”对象作为起始点，从这些节点开始向下搜索，搜索所有走过的路径为引用链，当一个对象到GC Roots没有任何引用链项链时, 则证明此对象时不可用的。

JVM怎么判断无用对象？

根搜索算法，从GCRoot出发，对象没有引用，就判定为无用对象

GCRoot对象包括：

类对象，虚拟机栈的对象，常量引用的对象，本地方法栈的对象

## GC回收算法

1）复制算法

思路：两个区域A和B，初始对象在A，存活的对象被转移到B，最后清空A。

该算法直接把A清空了，因此不会出现内存碎片。

新生代采用复制算法，新生代垃圾对象多，存活少。

2）标记-清除算法：

思路：第一阶段从引用根节点开始标记所有被引用的对象（可达性分析），第二阶段遍历整个堆，把未标记的对象清除。

该算法会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次GC动作。

老年代采用标记-清除算法。

3）标记-整理算法：

相比标记-清除算法，多了碎片整理，整理出更大的内存来存放更大的对象。

## 4种对象引用类型

强引用：默认对象采用的均为强引用（该对象的实例没有其它对象的引用，GC时才会被回收）

软引用：软引用是Java中提供的一种比较适合缓存场景的应用（只有在内存不够用的情况下才会被GC）

弱引用：在GC时一定会被GC

虚引用：由于虚引用只是用来得知对象是否被GC

# 几种GC收集器

## STW(stop-the-world)

gc有两个重要指标：

停顿时间：垃圾收集器做垃圾回收中断应用执行的时间。-XX:MaxGcPauseMillis

吞吐量：垃圾收集的时间和总时间的占比。1/(1+n)，吞吐量为1-1/(1+n)。-XX:GCTimeRation=n

jvm由于要执行GC而停止了应用程序执行，任何一种gc算法都会发生。多数gc优化通过减少stop-the-world发生的时间来提高程序性能（高吞吐、低停顿）。

吞吐量=运行用户代码时间 / (运行用户代码时间 + 垃圾回收时间)

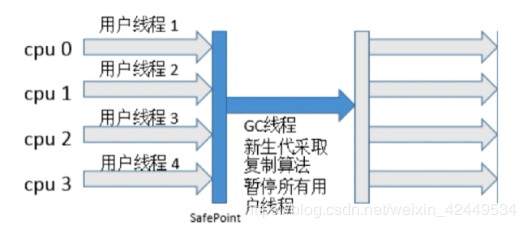
## 新生代垃圾收集器

### Serial New收集器（复制算法）

特点：

单线程收集，进行垃圾收集时，必须暂停所有工作线程(stop-the-world)；

简单高效，Client模式下默认的新生代收集器。

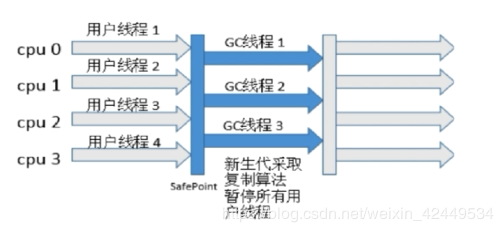


### ParNew收集器（复制算法）

特点：

实现算法跟Serial完全一样，可认为是Serial收集器的多线程版本，在多核CPU环境下有着比Serial更好的表现。

多线程可能存在上下文切换的问题。



### Parallel Scavenge（复制算法）

并行收集，更关注吞吐量，多用于客户端。

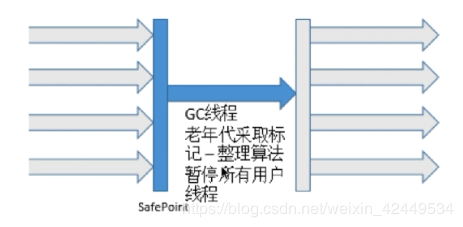
## 老年代垃圾收集器

### Serial Old收集器（标记 - 整理算法）

特点：

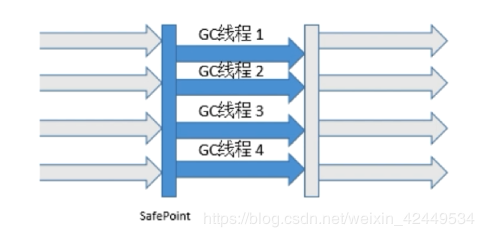
单线程收集，进行垃圾收集时，必须暂停所有工作线程；

简单高效，Client模式下默认的老年代收集器。



### Parallel Old收集器（标记 - 整理算法）

特点：多线程，吞吐量优先。



## CMS收集器（标记-清理算法）

流程：

1）初始标记：标记Old区的GCRoot对象和被Y区引用的对象，耗时短。暂停用户线程（stop-the-world）

2）追溯标记：GCRootTracing，并发追溯标记，程序不会停顿。

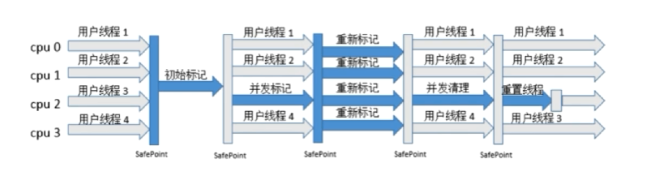
3）重新标记：修正2过程中的标记。暂停用户线程（stop-the-world）

4）并发清理：清理垃圾对象，程序不会停顿（采用标记清理算法，而不影响其它位置内存，所以可以并发搞，但会产生内存碎片）

说明：

【1】中是做了GCRoot可达性的初始化标记，【3】是对【2】中变动的对象。

由于使用标记-清除算法，可能产生内存碎片，而导致大对象无法分配内存，从而提前触发一次full gc。



## G1收集器（复制 + 标记 - 整理算法）-官方推荐

G1收集器将Java堆内存划分成多个大小相等的Region，每个region既用于新生代也用于老年代，新生代和老年代不再物理隔离。

将堆内存划分成2000块左右的小块，每块大小1-32M（2的幂次），每块region都可以作为E、S、O任意一种，分配灵活，但是存在大对象问题。

大对象问题解决：

小于一半region size的可以正常存入E区；

一半到一个region size的直接存入O区一个region中，这个region又叫Humongous region，我们也可以把它叫做H区（他本质还是O区的）；

比一个region size还要大的对象，需要存入连续的多个region中，这多个region都是H区。

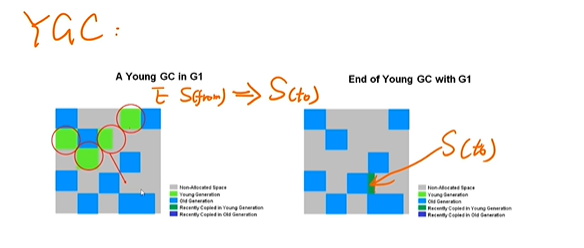
两个概念：

RememberSets：又叫Rsets是每个region中都有的一份存储空间，用于存储本region的对象被其他region对象的引用记录。

CollectionSets：又叫Csets是一次GC中需要被清理的regions集合，注意G1每次GC不是全部region都参与的，可能只清理少数几个，这几个就被叫做Csets。

### YGC（复制算法）

新生代的GC，STW，复制算法。将E和S(from)区复制到S(to)，注意S(to)一开始是没有标识的，就是个free region。下图中没有标出YGC进入老年代的对象，有可能有一部分会进入O区！



### MixGC

MixGC针对老年代和部分新生代。

流程如下：

https://github.com/sunwu51/notebook/blob/master/19.09/java\_jvm%E5%9E%83%E5%9C%BE%E6%94%B6%E9%9B%86%E5%99%A8.md #相关链接

-1）初始标记：标记GCRoot直接引用的对象和所在Region

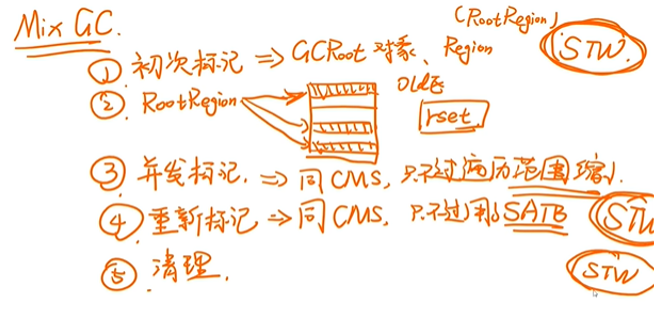
-2）追溯标记：时间较长，分析GCRoot到所有对象的可达性分析。

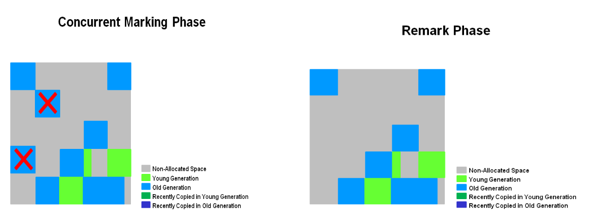
如果从GC Root节点开始遍历所有对象会比较耗时，实际上JVM也不是这么做的。JVM是使用Remembered Set保存了对象引用的调用信息，在可达性分析的时候只需要同时遍历remembered set就好了，不需要从根节点开始挨个遍历。

-3）重新标记：修正3过程的错误

-4）并发清理：通过标记整理的算法，根据用户配置的回收时间，和维护的优先级列表，优先收集价值最大的region。收集阶段是基于标记-整理和复制算法实现。

MixGC过程：





### G1问题列表

问：G1回收器和其他回收器有什么区别？

问：RootRegionScan这个阶段是干嘛的？

标记出RootRegion指向O区的region，标记这些region是为了降低并发标记的扫描范围，因为并发标记需要扫描GCROOT引用或间接的所有对象，而这些对象一定是在RootRegion出发指向的Region中的。MIXGC中Y区本来就要全扫，所以这里再按照O区过滤下，这样就缩小了扫描范围。该阶段的操作为遍历O区region查询Rset是否有来自RootRegion的，（RootRegion是初始标记得到的）

问：Rset作用有哪些？

上题中的作用是一个，还有个作用是YGC时，O区不GC因而认为O区全为‘GCroot’，需扫描全部O区。有了Rset只需要查看所有Y区region的Rset就知道被哪些O区region跨带引用了，避免了扫描整个O区。

问：G1提高效率的点有哪些？

1 重新标记时X区域直接删除。

2 Rset降低了扫描的范围，上题中两点。

3 重新标记阶段使用SATB速度比CMS快。

4 清理过程为选取部分存活率低的Region进行清理，不是全部，提高了清理的效率。

问：对比CMS，有哪些不同？

1 region化的内存结构，采用复制清理的方式，避免了内存碎片。但是这种清理也造成了STW。

2 SATB速度更快。

3 初始标记，并发标记，重新标记，清理垃圾四个阶段很像，但是G1中有很多标记region的操作，并借助Rset进行了范围的缩小，提高了并发标记的速度。小结下就是初始标记和YGC的STW一起了，提高了效率；并发标记因为rset的设计，扫描范围缩小了，提高了效率；重新标记因为使用了SATB提高了效率；清理虽然造成了STW，但是复制使内存紧凑，避免了内存碎片。同时只清理垃圾较多的region，最大限度的降低了STW时间。