# 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

JVM调优原则

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

相关教程：

https://www.cnblogs.com/vana/p/10843289.html

## 概念

相关链接：

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

GC的作用区域：方法区、堆

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）。

默认-XX:SurvivorRatio=8，则代表eden:S0:S1=8:1:1

若-XX:SurvivorRatio=4，则代表eden:S0:S1=8:1:1

-XX:+PrintGCDetails 输出详细的GC收集日志信息（GC、FullGC）

## 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+From Survivor区的幸存对象被复制到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。

## 新生代GC（Minor GC）和老年代GC（Major GC/Full GC）

新生代GC：发生在新生代的GC。由于Java对象大多都具备朝生夕灭的特性，所以MinorGC非常频繁，一般回收速度也比较快。

老年代GC：发生在老年代的GC。出现MajorGC一般会有至少一次MinorGC，MajorGC速度比MinorGC慢10倍以上。

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。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；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，如果还不足则抛出OOM异常。

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。

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：MinorGC后进入老年代对象内存超过老年代剩余内存；大对象直接进入老年代但放不下；

-2）方法区满了

-3）通过MinorGC后进入老年代的对象大小大于老年代剩余空间大小

-4）动态年龄：MinorGC中，需要向To区复制的对象大小大于To内存的50%，则把较大年龄的对象转存到老年代。

注：程序中主动调用System.gc()强制执行的GC是Full GC

动态年龄判定规则：

年龄1+年龄2+年龄3的对象占据了超过50%的survivor，就会让年龄3以上的对象进入老年代。

**MinorGC过程：**

1）eden、FromSurvivor复制到ToSurvivor（若对象年龄达到15后则复制到老年区），年龄+1。

2）清空eden、FromSurvivor

3）FromSurvivor和ToSurvicor名字互换

## 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根节点**：类加载器、Thread、虚拟机栈的本地变量表、static成员、常量引用、本地方法栈的变量等

## 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(强/软/弱/虚引用)

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

软引用示例：

假如有一个应用需要读取大量的本地图片，会存在两个问题：1、如果每次读取图片都从硬盘读取则会严重影响性能；2、如果一次性全部加载到内存中又可能造成内存溢出。

此时使用软引用可以解决这个问题。

设计思路：用一个HashMap来保存图片的路径和相应图片对象关联的软引用之间的映射关系,在内存不足时，JVM会自动回收这些缓存图片对象，从而有效避免OOM问题。

Map<String, SoftReference<Bitmap>> imageCache = new HashMap<>();

弱引用示例：

ThreadLocal是被ThreadLocalMap以弱引用关联着

## JVM调优原则&两个指标

jvm调优原则：

1）减少fullGC次数

2）减少fullGC执行时间

JVM调优主要就是调整下面两个指标：

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占比：1/(1+n)，吞吐量为1-1/(1+n)。-XX:GCTimeRatio=n

## 调优步骤

调优步骤：

1）打印GC日志

2）分析日志得到关键性指标

3）分析GC原因，调优JVM参数

打印GC日志：

-XX:+PrintGCDetails –XX:+PrintGCDateStamps -Xloggc:./gc.log

Tomcat则直接加在JAVA\_OPTS变量中



# 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：就是CPU用于运行用户代码时间与CPU总消耗时间的比值。虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。

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)

吞吐量 = 1 - 1/(1+n); //n为垃圾收集时间

可通过-XX:GCTimeRation=n设置垃圾收集时间。

## JDK默认垃圾收集器

jdk1.7 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）

jdk1.8 默认垃圾收集器Parallel Scavenge（新生代）+Parallel Old（老年代）

jdk1.9 默认垃圾收集器G1

-XX:+PrintCommandLineFlagsjvm参数可查看默认设置收集器类型

-XX:+PrintGCDetails亦可通过打印的GC日志的新生代、老年代名称判断

## 并行(Parallel)与并发(Concurrent)

并行：指多条垃圾收集线程并行工作，但此时用户线程处于等待状态；

并发：指用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行），用户程序在继续运行，而垃圾收集线程运行于另一个CPU上。

## GC收集器对应的JVM参数

JVM参数：

-XX:+UseSerialGC

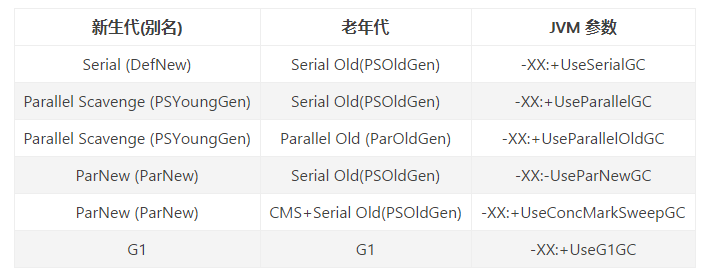
-XX:+UseParallelGC

-XX:+UseParallelOldGC

-XX:-UseParNewGC

-XX:+UseConcMarkSweepGC

-XX:+UseG1GC



## GC日志查看

参数列表：

-XX:+PrintGC 输出GC日志

-XX:+PrintGCDetails 输出GC的详细日志

-XX:+PrintGCTimeStamps 输出GC的时间戳（以基准时间的形式）

-XX:+PrintGCDateStamps 输出GC的时间戳（以日期的形式，如 2013-05-04T21:53:59.234+0800）

-XX:+PrintHeapAtGC 在进行GC的前后打印出堆的信息

-Xloggc:../logs/gc.log 日志文件的输出路径

在eclipse.ini中配置下面代码后启动，会在同目录下生成gc.log文件：

-XX:+PrintGCDateStamps -XX:+PrintGCDetails -Xloggc:./gclogs

新生代GC日志格式：...

老年代GC日志格式：...

分析GC日志工具：...

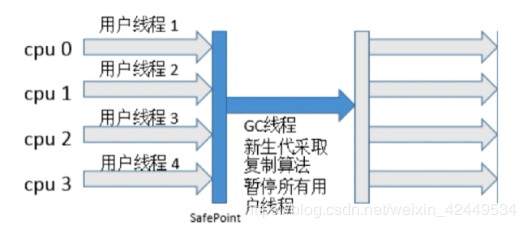
## 新生代垃圾收集器

### 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。

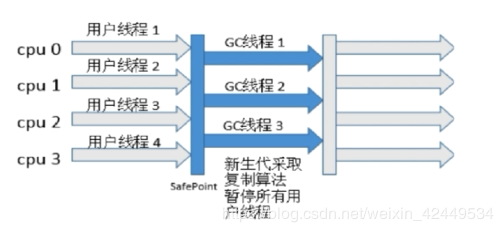


### ParNew收集器（复制算法）

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。

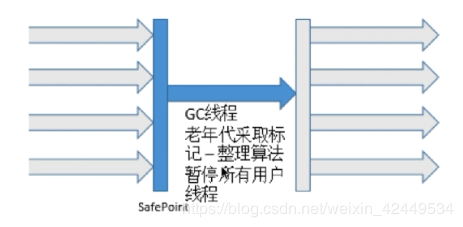
## 老年代垃圾收集器

### Serial Old收集器（标记 - 整理算法）

特点：

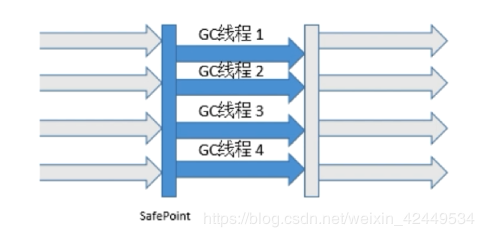
单线程收集，进行垃圾收集时，必须暂停所有工作线程；

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）

描述：

CMS收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求。

CMS的JVM参数为-XX:+UseConcMarkSweepGC

### 流程

1）初始标记（initial mark）：仅标记一下GCRoots能直接关联到的对象，速度很快。暂停用户线程（stop-the-world）

2）追溯标记（concurrent mark）：进行GC Root Tracing，并发追溯标记，用户线程不会停顿。

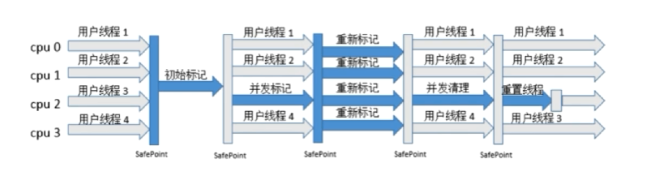
3）重新标记（remark）：修正2过程中的标记。是为了修正2过程中因用户程序继续运作而产生变动的那一部分对象，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。暂停用户线程（stop-the-world）

4）并发清理（concurrent sweep）：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）。由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

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。



缺点：

1、CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次FullGC的产生。

由于CMS并发清理阶段用户线程还在运行着，伴随着用户线程运行自然会有新的垃圾产生，这部分垃圾只能等下一次GC时在清理掉，称为“浮动垃圾”。

由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。

2、CMS收集器会产生大量空间碎片。

CMS收集器基于“标记清除”算法实现，在收集结束后会有大量内存碎片产生。碎片过多时，将会给大对象分配带来麻烦，当大对象找不到足够大连续内存空间分配时，不得不提前触发一次FullGC。

### CMS优化

CMS优化过程：

在初始化标记阶段，为了最大限度地减少STW的时间开销，我们可以使用

-XX:+CMSParallelInitialMarkEnabled

开启初始标记过程中的并行化，进一步提升初始化标记效率；

在重新标记阶段，因为重新标记的内存范围是整个堆，包含\_young\_gen和\_old\_gen。为什么要扫描新生代呢，因为对于老年代中的对象，如果被新生代中的对象引用，那么就会被视为存活对象，因此对于老年代来说，引用了老年代中对象的新生代的对象，也会被老年代视作“GC ROOTS”。如果我们分析gc日志发现ReMark阶段发生STW的时间过长，我们就可以开启：

-XX:+CMSScavengeBeforeRemark

它的作用是在重新标记之前对年轻代做一次minor GC，这样yong gen中剩余待标记的对象数量相比gc之前势必下降很多(只剩下存活的obj，大量死亡的obj被GC干掉了)，剩余被视作“GC ROOTS”的对象数量骤减，如此Remark的工作量就少很多，重新标记的时间开销也会减少；当然这里Remark减少的时间和YGC的时间开销要做一个权衡，根据实践结果选择是否要开启CMSScavengeBeforeRemark；同样在Remark中我们也可以根据实际情况选择是否开启并行化Flag：

-XX:+CMSParallelRemarkEnabled

总之，性能调优是个理论+实践的技术活，需要我们根据实际情况选择合适的VM参数。

优化总结：

优化1：CMS是标记清除算法，因此会产生内存碎片。

它有一个参数能开启压缩处理，在达到多少次FullGC后就会整理内存碎片，但是这个参数页不太好配置。

-XX:+UseCMSCompactAtFullCollection --默认为true，即做压缩处理

-XX:CMSFullGCsBeforeCompaction=0//执行多少次fullGC后才做压缩，默认为0。这个参数用来减低fullGC的压缩频率，以减少fullGC暂停时间

优化2：降低CMS GC频率或者增加频率、减少GC时长

-XX:CMSInitiatingOccupancyFraction=70//默认CMS在对内存占用率达到80%时开始GC（因为CMS有浮动垃圾，所以一般都较早启动GC）

-XX:+UseCMSInitiatingOccupancyOnl//只是用设定的回收阈值（上面指定的70%），如果不指定，JVM仅在第一次使用设定值，后续则自动调整.

优化3：在CMS GC前启动一次ygc，以减少GCRoot的对象

-XX:+CMSScavengeBeforeRemark

## 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。

### 特点

1）分代收集

2）空间整合

3）可预测停顿：建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。

4）收集范围缩小

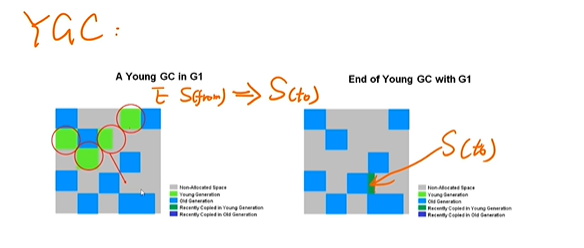
在G1之前的其他收集器进行收集的范围都是整个新生代或者老年代，G1收集器将整个Java堆划分为多个大小相等的独立区域Region。

G1收集器还保留新生代和老年代的概念，但它们不再是物理隔离，而是一部分region的集合。

G1收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Regionl里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region（名称Garbage-First的由来）。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限时间内可以获取尽可能高的手机效率。

### 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（Initial Marking）：标记GCRoot直接引用的对象。并且修改TAMS的值，让下一阶段用户程序并发运行时，能在正确可用的Region中创建新对象

暂停用户线程（stop-the-world）

-2）并发标记（Concurrent Marking）：从GCRoot开始，对堆中对象进行可达性分析，找出存活的对象。

这个阶段不会停顿用户线程，耗时较长。

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）最终标记（Final Marking）：为了修正2阶段产生的变动。

虚拟机将这段时间对象变化记录在线程RememberedSetLogs里面，最终标记阶段需要把RememberedSetLogs的数据合并到RememberedSet中。

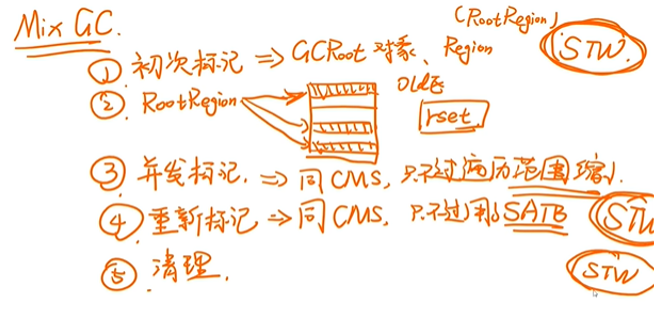
。暂停用户线程（stop-the-world）

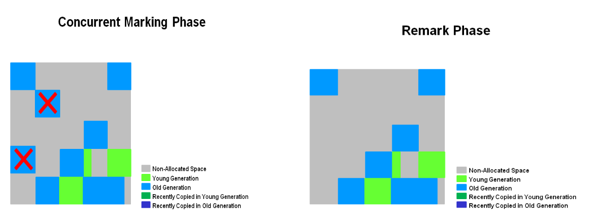
-4）最终清理：

首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划，这个阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分Region，时间是用户可控制的，而且停顿用户线程将大幅提高收集效率。

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。暂停用户线程（stop-the-world）

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：G1与CMS对比？

1、G1采用标记整理算法，CMS采用标记清理算法。因此G1避免了内存碎片，但造成了STW。

2、G1采用region化的内存结构，再标记中借助Rset进行范围的缩小，提高了并发标记的速度。

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。

CMS和G1相同点？

1、初始标记都会停顿STW，即不能与用户线程并行，都是标记能与GC Root直接关联的对象。

2、并发标记不会停顿，即可以与用户线程一起进行。并发标记主要沿着GC Reference Chain进行对象可达性分析的过程。

CMS和G1不同点？

1、CMS的并发清理不会停顿，即与用户线程一起进行。G1的最终清理需要停顿。

2、CMS的收集器不可以对young generation回收，只针对old generation回收，因此需要与其它收集器配合，如ParNew等。G1采用region化的内存结构，针对的是整个堆区。

3、CMS采用标记清除算法，会产生内存碎片，但不会STW。G1采用复制-标记整理算法，年轻代采用复制算法，old区采用标记整理算法，不会产生内存碎片，但会STW。

4、CMS的年轻代和老年代是物理隔离的，为两个不同的连续内存区域。G1是对整个Heap划分为若干不同的Region，既可以是年轻代和老年代，是逻辑上的分类，Region大小可调。

G1和CMS比较：

G1和PN+CMS调优原则都是尽可能ygc，不做老年代gc。

g1相对而言更加智能，也意味着jvm会更多的资源去判断每个region的使用情况。虽然G1在gc时不会产生碎片，但是由于每个region存在存活率85%不清理的机制，会导致内存没有充分释放问题，

因此，对于cpu性能高的、内存容器大的，对应用响应度高的系统推荐使用G1，而内存小，cpu性能比较低的系统也可以使用PN+CNS会更合适。

为什么说G1适合大内存？

假设你由32G内存，如果用ParNew+CMS，必须等待你的内存填满了才触发GC，此时GC就会回收几十G的垃圾，那么会很慢可能导致你的系统停顿几十秒。但是用了G1之后，它会更加频繁的回收Region，每次就回收一部分Region，保证停机时间不会太长。所以G1更加适合大内存机器。