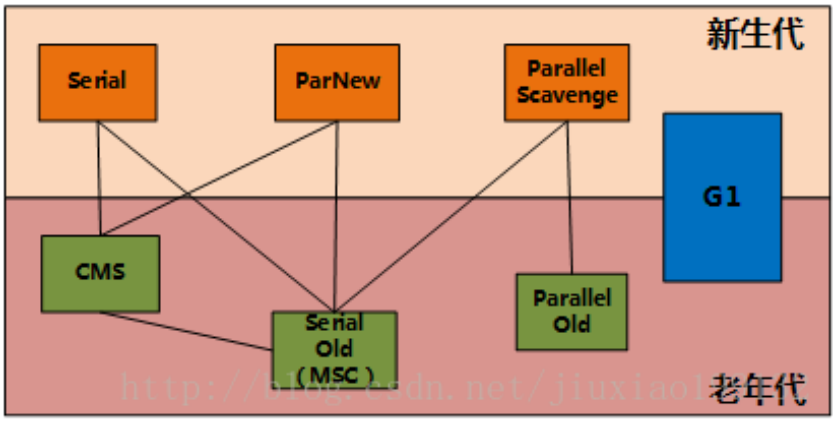
简述

垃圾收集器是垃圾回收算法（标记-清除算法、复制算法、标记-整理算法等）的具体实现，不同商家、不同版本的JVM所提供的垃圾收集器
HotSpot虚拟机中的垃圾收集器。不同的垃圾收集器有不同的特性，适用于不同的场景。没有最好的收集器，更没有万能的收集；选择的只

垃圾收集器的组合



- 图中展示了7种不同分代的收集器： Serial、ParNew、Parallel Scavenge、Serial Old、Parallel Old、CMS、G1
- 它们所处区域，则表明其是属于新生代收集器还是老年代收集器：
  新生代收集器：Serial、ParNew、Parallel Scavenge；
  老年代收集器：Serial Old、Parallel Old、CMS；
  整堆收集器：G1；
- 两个收集器间有连线，表明它们可以搭配使用：

Serial/Serial Old、Serial/CMS、ParNew/Serial Old、ParNew/CMS、Parallel Scavenge/Serial Old、Parallel Scavenge/Parallel Old、G1；

并发垃圾收集和并行垃圾收集的区别

并行

真实的多个垃圾回收线程同时进行，但是不管怎么样垃圾回收线程执行的时候，用户线程都是等待状态

并发

单cpu

一会执行垃圾回收线程，一会儿用户线程，来回切换的速度非常快，导致错觉认为是并行执行，实际上同一时间只有一个线程执行，要么是多cpu
可能用户线程和垃圾回收线程在不同的cpu上同时执行

Minor GC和Full GC的区别

Minor GC

又称新生代GC，指发生在新生代的垃圾收集动作；
因为Java对象大多是朝生夕灭，所以Minor GC非常频繁，一般回收速度也比较快；

Full GC

又称Major GC或老年代GC，指发生在老年代的GC；
出现Full GC经常会伴随至少一次的Minor GC（不是绝对，Parallel Sacvenge收集器就可以选择设置Major GC策略）；
Major GC速度一般比Minor GC慢10倍以上

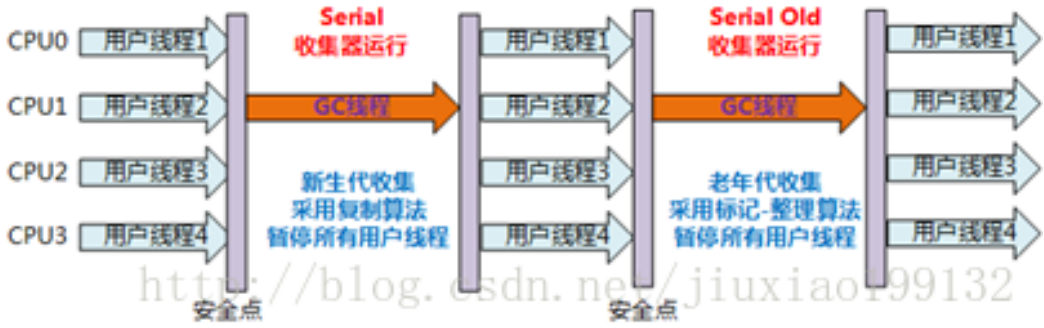
Serial收集器（串行收集器）

Serial（串行）垃圾收集器是最基本、发展历史最悠久的收集器。JDK1.3.1前是HotSpot新生代收集的唯一选择

特点

- 针对新生代
- 采用复制算法
- 单线程收集器
- 进行垃圾收集时，必须暂停所有工作线程，直到完成 即会”Stop The World”

Serial/Serial Old组合收集器运行示意图如下



应用场景

依然是HotSpot在Client模式下默认的新生代收集器；
也有优于其他收集器的地方：
简单高效（与其他收集器的单线程相比）；
对于限定单个CPU的环境来说，Serial收集器没有线程交互（切换）开销，可以获得最高的单线程收集效率；
在用户的桌面应用场景中，可用内存一般不大（几十M至一两百M），可以在较短时间内完成垃圾收集（几十MS至一百多MS）,只要不频繁

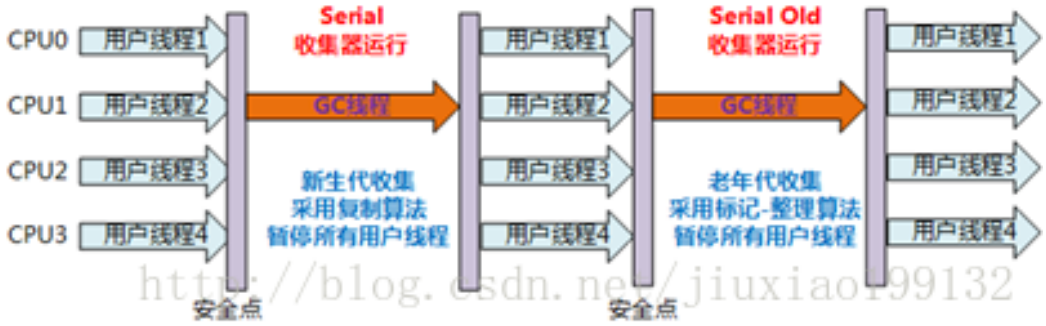
Serial Old收集器

Serial Old是 Serial收集器的老年代版本

特点

- 针对老年代
- 采用”标记-整理”算法（还有压缩，Mark-Sweep-Compact）
- 单线程收集

Serial/Serial Old收集器运行示意图如下：



应用场景

主要用于Client模式，而在Server模式有两大用途：

- 在JDK1.5及之前，与Parallel Scavenge收集器搭配使用（JDK1.6有Parallel Old收集器可搭配）
- 作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用

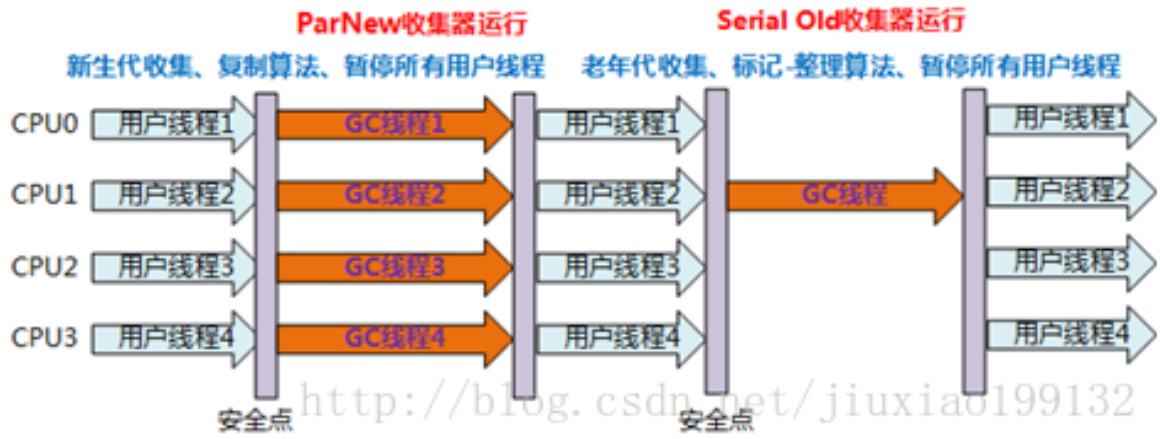
ParNew收集器

ParNew垃圾收集器是Serial收集器的多线程版本

特点

多线程收集器，其他与Serial 收集器一样。 如Serial收集器可用控制参数、收集算法、Stop The World、内存分配规则、回收策略等

ParNew/Serial Old组合收集器运行示意图如下：



应用场景

- 在Server模式下，ParNew收集器是一个非常重要的收集器，因为除Serial外，目前只有它能与CMS收集器配合工作
- 但在单个CPU环境中，不会比Serail收集器有更好的效果，因为存在线程交互开销

Parallel Scavenge收集器

Parallel Scavenge垃圾收集器因为与吞吐量关系密切，也称为吞吐量收集器（Throughput Collector）

特点

- 新生代收集器
- 采用复制算法
- 多线程收集
- 关注吞吐量：其他收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是达一个

应用场景

高吞吐量为目标，即减少垃圾收集时间，让用户代码获得更长的运行时间

吞吐量

高吞吐量即减少垃圾收集时间，让用户代码获得更长的运行时间

```
1  吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）
```

Parallel Scavenge收集器提供两个参数用于精确控制吞吐量：

- -XX:MaxGCPauseMillis
  控制最大垃圾收集停顿时间，大于0的毫秒数
  MaxGCPauseMillis设置得稍小，停顿时间可能会缩短，但也可能会使得吞吐量下降；因为可能导致垃圾收集发生得更频繁

- -XX:GCTimeRatio
  设置垃圾收集时间占总时间的比率，取值范围是0到100之间.
  即：垃圾回收时间/(用户运行代码时间+垃圾回收时间)

- -XX:+UseAdptiveSizePolicy
  开启这个参数后，就不用手工指定一些细节参数，如：
  新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRation）、晋升老年代的对象年龄（-XX:PretenureSizeThresho

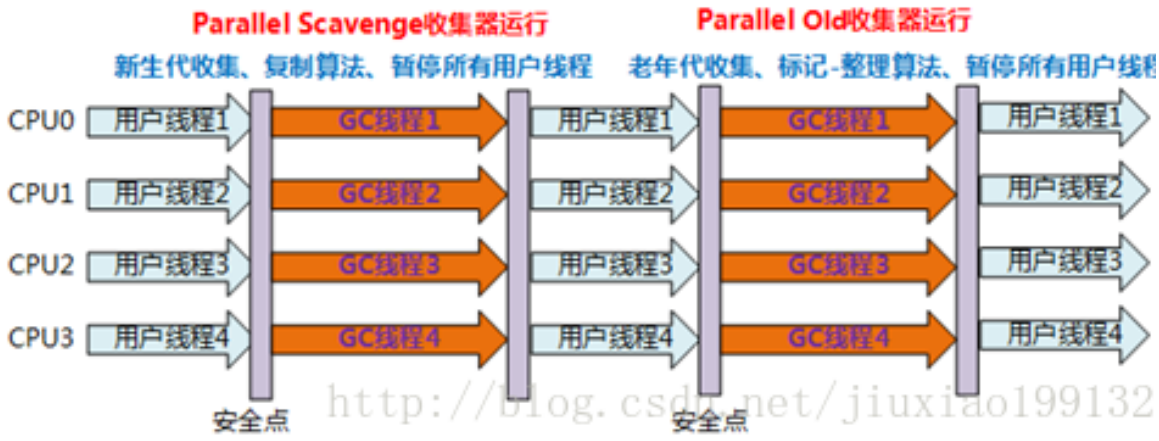
Parallel Old收集器

Parallel Old垃圾收集器是Parallel Scavenge收集器的老年代版本，JDK1.6中才开始提供。

特点

- 针对老年代
- 采用”标记-整理”算法
- 多线程收集

Parallel Scavenge/Parallel Old收集器运行示意图如下：



应用场景

JDK1.6及之后用来代替老年代的Serial Old收集器特别是在Server模式，多CPU的情况下。这样在注重吞吐量以及CPU资源敏感的场景，
Parallel Old收集器的"给力"应用组合

CMS收集器

并发标记清理（Concurrent Mark Sweep，CMS）收集器也称为并发低停顿收集器（Concurrent Low Pause Collector）或低延迟（low-late
停顿时间

特点

- 针对老年代
- 基于”标记-清除”算法(不进行压缩操作，产生内存碎片)
- 以获取最短回收停顿时间为目标
- 多线程收集器
- 需要更多的内存

应用场景

与用户交互较多的场景
希望系统停顿时间最短，注重服务的响应速度
以给用户带来较好的体验
如常见WEB、B/S系统的服务器上的应用

CMS收集器运作过程

比前面几种收集器更复杂，可以分为4个步骤：

初始标记（CMS initial mark）

仅标记一下GC Roots能直接关联到的对象，速度很快但需要"Stop The World"

并发标记（CMS concurrent mark）

进行GC Roots Tracing的过程，刚才产生的集合中标记出存活对象，应用程序也在运行，并不能保证可以标记出所有的存活对象

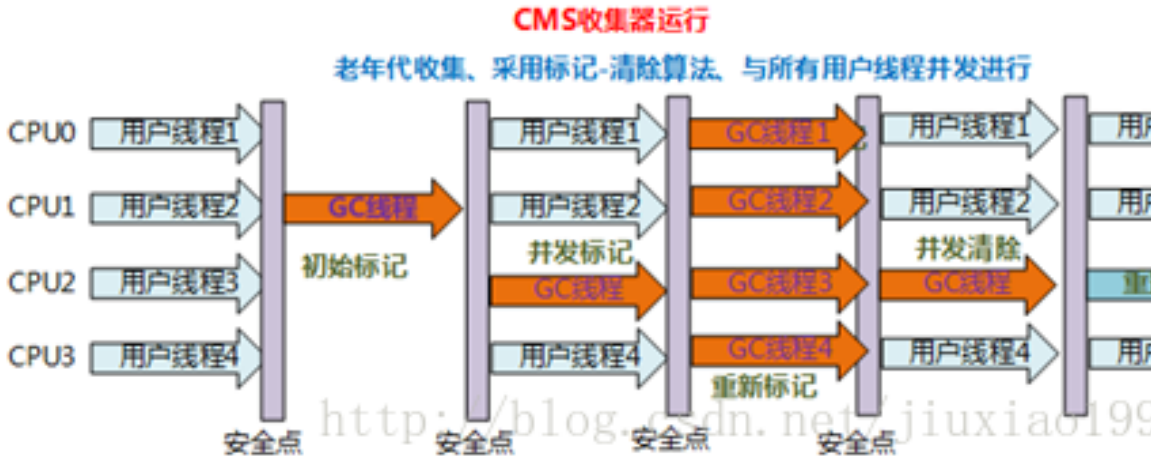
重新标记（CMS remark）

为了修正并发标记期间因用户程序继续运作而导致标记变动的那一部分对象的标记记录，需要"Stop The World"，且停顿时间比初始标
程并行执行来提升效率；

并发清除（CMS concurrent sweep）

回收所有的垃圾对象

整个过程中耗时最长的并发标记和并发清除都可以与用户线程一起工作。所以总体上说，CMS收集器的内存回收过程与用户线程一起并发执



CMS收集器3个明显的缺点

对CPU资源非常敏感

- 并发收集虽然不会暂停用户线程，但因为占用一部分CPU资源，还是会导致应用程序变慢，总吞吐量降低
- CMS的默认收集线程数量是=(CPU数量+3)/4
- 当CPU数量多于4个，收集线程占用的CPU资源多于25%，对用户程序影响可能较大；不足4个时，影响更大，可能无法接受

无法处理浮动垃圾,可能出现”Concurrent Mode Failure”失败

- 浮动垃圾（Floating Garbage）

```
在并发清除时，用户线程新产生的垃圾，称为浮动垃圾；
这使得并发清除时需要预留一定的内存空间，不能像其他收集器在老年代几乎填满再进行收集；
也要可以认为CMS所需要的空间比其他垃圾收集器大；
-XX:CMSInitiatingOccupancyFraction：设置CMS预留内存空间；
JDK1.5默认值为68%；JDK1.6变为大约92%；
```

- “Concurrent Mode Failure”失败

```
如果CMS预留内存空间无法满足程序需要，就会出现一次"Concurrent Mode Failure"失败
这时JVM启用后备预案：临时启用Serail Old收集器，而导致另一次Full GC的产生；
这样的代价是很大的，所以CMSInitiatingOccupancyFraction不能设置得太大
```

产生大量内存碎片

```
由于CMS基于"标记-清除"算法，清除后不进行压缩操作
产生大量不连续的内存碎片会导致分配大内存对象时，无法找到足够的连续内存，从而需要提前触发另一次Full GC动作。
解决方法：
    (1)、"-XX:+UseCMSCompactAtFullCollection"
     使得CMS出现上面这种情况时不进行Full GC，而开启内存碎片的合并整理过程；
     但合并整理过程无法并发，停顿时间会变长；
     默认开启（但不会进行，结合下面的CMSFullGCsBeforeCompaction）；
    (2)、"-XX:+CMSFullGCsBeforeCompaction"
     设置执行多少次不压缩的Full GC后，来一次压缩整理；
      为减少合并整理过程的停顿时间；
      默认为0，也就是说每次都执行Full GC，不会进行压缩整理；

```

总体来看，与Parallel Old垃圾收集器相比，CMS减少了执行老年代垃圾收集时应用暂停的时间；

但却增加了新生代垃圾收集时应用暂停的时间、降低了吞吐量而且需要占用更大的堆空间；

G1收集器

G1（Garbage-First）是JDK7-u4才推出商用的收集器，未来有可能替代CMS收集器。

特点

- 分代GC：G1收集器与其他收集器不同，虽然也区分新生代与老年代，但是G1即可处理新生代，也可以处理老年代。
- 分区算法：G1收集器采用分区算法，将堆内存空间分成若干个同样同等大小的区域，每个区域单独管理，每个区域划分新生代和老
- 并行性：G1在回收期间，可以由多个GC线程同时工作，有效利用多核计算能力。
- 并发性：G1可以不用整个用户线程停顿，分开清理一些区域。
- 可预测的停顿：低停顿的同时实现高吞吐量

应用场景

- 面向服务端应用，针对具有大内存、多处理器的机器；
- 最主要的应用是为需要低GC延迟，并具有大堆的应用程序提供解决方案

垃圾收集器常用参数

```
-XX:+UseSerialGC：在新生代和老年代使用串行收集器
-XX:SurvivorRatio：设置eden区大小和survivior区大小的比例
-XX:NewRatio:新生代和老年代的比
-XX:+UseParNewGC：在新生代使用并行收集器
-XX:+UseParallelGC ：新生代使用并行回收收集器
-XX:+UseParallelOldGC：老年代使用并行回收收集器
-XX:ParallelGCThreads：设置用于垃圾回收的线程数
-XX:+UseConcMarkSweepGC：新生代使用并行收集器，老年代使用CMS+串行收集器
-XX:ParallelCMSThreads：设定CMS的线程数量
-XX:CMSInitiatingOccupancyFraction：设置CMS收集器在老年代空间被使用多少后触发
-XX:+UseCMSCompactAtFullCollection：设置CMS收集器在完成垃圾收集后是否要进行一次内存碎片的整理
-XX:CMSFullGCsBeforeCompaction：设定进行多少次CMS垃圾回收后，进行一次内存压缩
-XX:+CMSClassUnloadingEnabled：允许对类元数据进行回收
-XX:CMSInitiatingPermOccupancyFraction：当永久区占用率达到这一百分比时，启动CMS回收
-XX:UseCMSInitiatingOccupancyOnly：表示只在到达阀值的时候，才进行CMS回收
```

垃圾收集器常用组合

GC参数	Young（新生代）	tenu
-XX:+UseSerialGC	Serial	Seria
-XX:+UseParallelGC	Parallel Scavenge	Seria
-XX:+UseConcMarkSweepGC	ParNew	CMS
-XX:+UseParNewGC	ParNew	Seria
-XX:+UseParallelOldGC	Parallel Scavenge	Para
-XX:+UseConcMarkSweepGC -XX:+UseParNewGC	Serial	CMS
-XX:+UnlockExperimentalVMOptions -XX:+UseG1GC	G1	G1