http://www.cnblogs.com/smyhvae/p/4748313.html

本文主要内容：

- 堆的回顾
- 串行收集器
- 并行收集器
- CMS收集器

零、堆的回顾：

新生代中的98%对象都是"朝生夕死"的，所以并不需要按照1:1的比例来划分内存空间，而是将内存分为一块比较大的**Eden**空间和两块较小的**Survivor**空间，每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和Survivor的大小比例是8:1，也就是说，每次新生代中可用内存空间为整个新生代容量的90%（80%+10%），只有10%的空间会被浪费。

当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于10%的对象存活，当Survivor空间不够用时，需要依赖于老年代进行分配担保，所以大对象直接进入老年代。

堆的结构如下图所示：

垃圾收集器：

如果说收集算法时内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

虽然我们在对各种收集器进行比较，但并非为了挑出一个最好的收集器。因为直到现在位置还没有最好的收集器出现，更加没有万能的收集器，所以我们选择的只是对具体应用最合适的收集器。

一、串行收集器：**Serial**收集器

- 最古老，最稳定
- 简单而高效

- 可能会产生较长的停顿
- -XX:+UseSerialGC

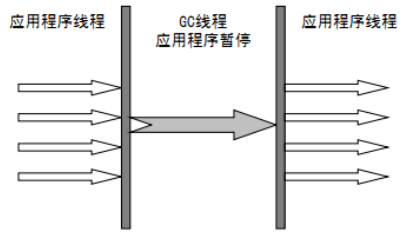
  新生代、老年代都会使用串行回收

  新生代复制算法

  老年代标记-整理

总结：**Serial收集器**对于**运行在Client模式下**的虚拟机来说是一个很好的选择。

这个收集器是一个单线程的收集器，但它的单线程的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。收集器的运行过程如下图所示：




```
0.844: [GC 0.844: [DefNew: 17472K->2176K(19648K), 0.0188339 secs] 17472K-
>2375K(63360K), 0.0189186 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
```

```
8.259: [Full GC 8.259: [Tenured: 43711K->40302K(43712K), 0.2960477 secs] 63350K-
>40302K(63360K), [Perm : 17836K->17836K(32768K)], 0.2961554 secs] [Times: user=0.28
sys=0.02, real=0.30 secs]
```

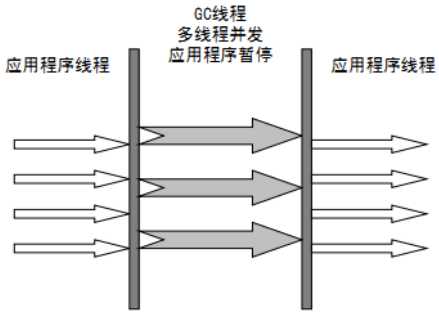
二、并行收集器：

1、ParNew收集器：

- ParNew收集器其实就是Serial收集器新生代的并行版本。
- 多线程，需要多核支持。
- -XX:+UseParNewGC

  新生代并行

  老年代串行
- -XX:ParallelGCThreads 限制线程数量




```
0.834: [GC 0.834: [ParNew: 13184K->1600K(14784K), 0.0092203 secs] 13184K->1921K(63936K),
0.0093401 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

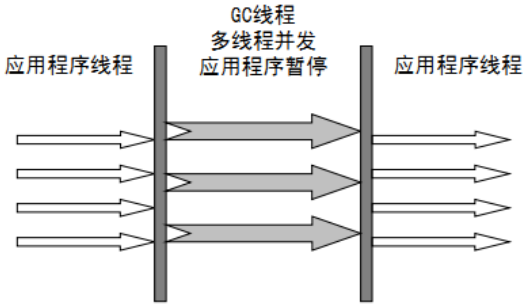
2、Parallel Scanvenge收集器：

- 类似ParNew，但更加关注吞吐量
- -XX:+UseParallelGC 使用Parallel Scanvenge收集器：新生代并行，老年代串行

3、Parallel Old收集器：

- Parallel Old收集器是Parallel Scanvenge收集器的老年代版本
- -XX:+UseParallelGC 使用Parallel Old收集器：新生代并行，老年代并行

如下图所示：




```
1.500: [Full GC [PSYoungGen: 2682K->0K(19136K)] [ParOldGen: 28035K-
>30437K(43712K)] 30717K->30437K(62848K) [PSPermGen: 10943K-
>10928K(32768K)], 0.2902791 secs] [Times: user=1.44 sys=0.03, real=0.30 secs]
```

各种参数设置：

- -XX:MaxGCPauseMills

  最大停顿时间，单位毫秒

  GC尽力保证回收时间不超过设定值

- -XX:GCTimeRatio

  0-100的取值范围

  垃圾收集时间占总时间的比

默认99，即最大允许1%时间做GC

注：这两个参数是矛盾的。因为停顿时间和吞吐量不可能同时调优。我们一方买希望停顿时间少，另外一方面希望吞吐量高，其实这是矛盾的。因为：在GC的时候，垃圾回收的工作总量是不变的，如果将停顿时间减少，那频率就会提高；既然频率提高了，说明就会频繁的进行GC，那吞吐量就会减少，性能就会降低。

吞吐量：**CPU用于用户代码的时间/CPU总消耗时间的比值**，即=运行用户代码的时间/(运行用户代码时间+垃圾收集时间)。比如，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%。

注2：以上所有的收集器当中，当执行GC时，都会stop the world，但是下面的CMS收集器却不会这样。

三、CMS收集器：

CMS收集器（Concurrent Mark Sweep：并发标记清除）是一种以获取最短回收停顿时间为目标的收集器。适合应用在互联网站或者B/S系统的服务器上，这类应用尤其重视服务器的响应速度，希望系统停顿时间最短。

- Concurrent Mark Sweep 并发标记清除，并发低停顿
- 标记-清除算法
- 并发阶段会降低吞吐量（因为停顿时间减少了，于是GC的频率会变高）
- 老年代收集器（新生代使用ParNew）
- -XX:+UseConcMarkSweepGC　打开这收集器

注：这里的并发指的是与用户线程一起执行。

2、CMS收集器运行过程：（着重实现了标记的过程）

（1）初始标记

根可以直接关联到的对象

速度快

（2）并发标记（和用户线程一起）

主要标记过程，标记全部对象

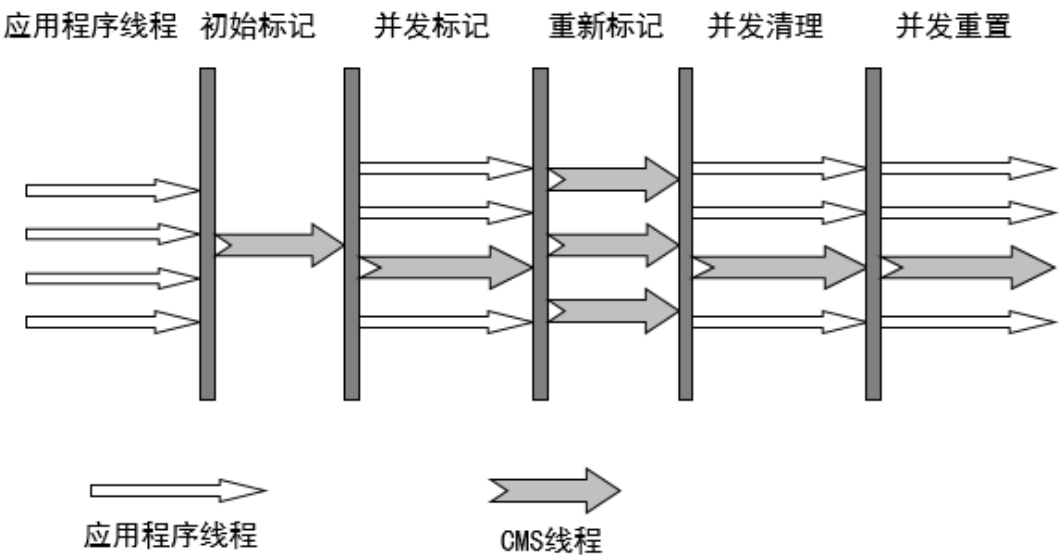
（3）重新标记

由于并发标记时，用户线程依然运行，因此在正式清理前，再做修正

（4）并发清除（和用户线程一起）

基于标记结果，直接清理对象

整个过程如下图所示：



其中，初始标记和重新标记时，需要stop the world。
整个过程中耗时最长的是并发标记和并发清除，这两个过程都可以和用户线程一起工作。
打印GC日志举例如下：

```
1.662: [GC [1 CMS-initial-mark: 28122K(49152K)]
29959K(63936K), 0.0046877 secs] [Times:
user=0.00 sys=0.00, real=0.00 secs]
1.666: [CMS-concurrent-mark-start]
1.699: [CMS-concurrent-mark: 0.033/0.033 secs]
[Times: user=0.25 sys=0.00, real=0.03 secs]
1.699: [CMS-concurrent-preclean-start]
1.700: [CMS-concurrent-preclean: 0.000/0.000
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
1.700: [GC[YG occupancy: 1837 K (14784 K)]1.700:
[Rescan (parallel) , 0.0009330 secs]1.701: [weak
refs processing, 0.0000180 secs] [1 CMS-remark:
28122K(49152K)] 29959K(63936K), 0.0010248
secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
1.702: [CMS-concurrent-sweep-start]
1.739: [CMS-concurrent-sweep: 0.035/0.037 secs]
[Times: user=0.11 sys=0.02, real=0.05 secs]
1.739: [CMS-concurrent-reset-start]
1.741: [CMS-concurrent-reset: 0.001/0.001 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
```

3、CMS收集器特点：

（1）尽可能降低停顿

（2）会影响系统整体吞吐量和性能

比如，在用户线程运行过程中，分一半CPU去做GC，系统性能在GC阶段，反应速度就下降一半

（3）清理不彻底

因为在清理阶段，用户线程还在运行，会产生新的垃圾，无法清理

（4）因为和用户线程一起运行，不能在空间快满时再清理

-XX:CMSInitiatingOccupancyFraction设置触发GC的阈值

如果不幸内存预留空间不够，就会引起concurrent mode failure

我们来看一下concurrent mode failure的日志：

```
33.348: [Full GC 33.348: [CMS33.357: [CMS-concurrent-sweep: 0.035/0.036 secs] [Times:
user=0.11 sys=0.03, real=0.03 secs]
 (concurrent mode failure): 47066K->39901K(49152K), 0.3896802 secs] 60771K-
>39901K(63936K), [CMS Perm : 22529K->22529K(32768K)], 0.3897989 secs] [Times:
user=0.39 sys=0.00, real=0.39 secs]
```

碰到上图中的情况，我们需要使用串行收集器作为后备。

4、既然标记清除算法会造成内存空间的碎片化，CMS收集器为什么使用标记清除算法而不是使用标记整理算法：

答案：

CMS收集器更加关注停顿，它在做GC的时候是和用户线程一起工作的（并发执行），如果使用标记整理算法的话，那么在清理的时候就会去移动可用对象的内存空间，那么应用程序的线程就很有可能找不到应用对象在哪里。

为了解决碎片的问题，CMS收集器会有一些整理上的参数，接下来就来讲这个。

5、整理时的各种参数：

- -XX:+ UseCMSCompactAtFullCollection

Full GC后，进行一次整理。整理过程是独占的，会引起停顿时间变长

- -XX:+CMSFullGCsBeforeCompaction

设置进行几次Full GC后，进行一次碎片整理

- -XX:ParallelCMSThreads

设定CMS的线程数量

四、GC参数的整理：

-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

最后的总结：

为了减轻GC压力，我们需要注意些什么？

- 软件如何设计架构（性能的根本在应用）
- GC参数属于微调（设置不合理会影响性能，产生大的延时）
- 堆空间如何管理和分配
- 代码如何写