# ­­­jvm7 based learning note

## 第三章：垃圾收集器和内存分配策略

### 判定对象已死的算法：

1. 引用计数Reference Counting:（实现简单，判定效率高，缺点：无法解决循环应用问题）
2. 可达性算法分析（Reachability Analysis）

通过一系列“GCRoots”的对象作为起始点，从这些节点开始向下搜索，搜索走过的路径称为引用链（Reference　Chain），当一个对象到GCRoot没有任何引用链相连，则说明对象不可用。

GC Roots对象包括：

1. 虚拟机栈（栈帧中本地变量表）中引用的对象
2. 方法区中类静态属性引用的对象
3. 方法区中常量引用的对象
4. 本地方法栈中（即一般说的Native方法）引用的对象

四种引用：

强引用

软引用

弱引用

虚引用

### 生存还是死亡—对象的大逃亡

对象非可达到对象回收经历的两个问题：

第一次标记：进行第一次标记并进行筛选，根据是否有必要执行finalize方法进行筛选。

如果对象没有覆盖finalize方法，或者finalize方法已经被虚拟机调用过，则视为“没有必要执行”。

如果被视为有必要执行finalize方法，这个对象被放置到F-Queue的队列中，稍后由虚拟机自动创建的、优先级低的Finalizer线程去执行它。这里的“执行”并不承若等待它执行结束。

避免极端情况下，finalize中死循环代码造成整个内存回收系统奔溃。Finalize方法是对象逃亡的最后机会。

|  |
| --- |
| /\*\*  \* 演示:  \* 1.对象可以在GC时自我拯救  \* 2.这种自我拯救只有一次,因为对象的finalize只能被系统自动调用一次  \*  \* 说明：仅为了做技术研究演示，实际开发中应避免使用finalize方法  \* **@author** 周志明  \*  \*/  **public** **class** FinalizeEscapeGC {  **private** **static** FinalizeEscapeGC *STRONG\_REFERENCE* = **null**;    **public** **void** isAlive() {  System.***out***.println("yes, i am still alive.");  }    @Override  **protected** **void** finalize() **throws** Throwable {  **super**.finalize();  System.***out***.println("finalize method executed.");  *STRONG\_REFERENCE* = **this**; // 重新关联引用，GC Root关联到  }    **public** **static** **void** main(String[] args) **throws** InterruptedException {  *STRONG\_REFERENCE* = **new** FinalizeEscapeGC();    // 对象第一次成功自我拯救  *STRONG\_REFERENCE* = **null**;  System.*gc*();    Thread.*sleep*(1000); // Finalize线程级别很低，睡眠1s等待线程执行  **if**(*STRONG\_REFERENCE* != **null**) {  *STRONG\_REFERENCE*.isAlive();  } **else** {  System.***out***.println("no, i am dead.");  }  // 一下代码和上述代码相同，但这次拯救失败了，因为finalize方法只能被系统调用一次  *STRONG\_REFERENCE* = **null**;  System.*gc*();    Thread.*sleep*(1000); // Finalize线程级别很低，睡眠1s等待线程执行  **if**(*STRONG\_REFERENCE* != **null**) {  *STRONG\_REFERENCE*.isAlive();  } **else** {  System.***out***.println("no, i am dead.");  }  }  } |

### 回收方法区

方法区（hotspot虚拟机中的永久代），永久代的回收分为两部分：废弃常量和无用的类。

判断一个废弃常量很简单，当前系统中没有任何一个String对象叫做“abc”了。

判断一个无用的类比较复杂：

1. 该类的所有实例都已被回收，也就是java堆中不存在该类的任何实例
2. 加载该类的ClassLoader已经被回收
3. 该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法

可以通过如下参数控制： -Xnoclassgc

可以使用一下参数观察类加载和卸载信息：-verbose:class以及-XX:+TraceClassLoading、-XX:+TraceClassUnLoading

其中-verbose:class以及-XX:+TraceClassLoading可以在Product版的虚拟机中使用，-XX:+TraceClassUnLoading需要在FastDebug版虚拟机中支持。

实际开发过程中，一下情况会造成方法区溢出：

1. 反射
2. 动态代理、CGLIB等动态字节码生成技术
3. 动态JSP，JSP会转化为Servlet类
4. OSGI这类需要频繁自定义CLassLoader的场景

### 垃圾收集算法

#### 标记-清除（Mark-Sweep）

最基础的回收算法。分为标记、清除两个过程

不足：1. 标记和清除两个阶段的效率不高 2.清除之后会产生大量不连续的内存碎片

#### 复制算法（Copying）

（新生代内存回收）

将内存空间分按容量划分为大小相等的两块，每次只使用其中一块。当这一块的内存用完了，就将还存活的对象复制到另一块上去，然后再把已使用过的内存空间一次清理掉。这就使得每次仅对半区进行回收，内存分配时就不用考虑内存碎片等复杂问题。

不足：内存空间利用率低，只能使用一半。

现在商业虚拟机都是用这种算法来回收新生代。IBM公司的专门研究表明，98%的对象是“朝生夕死”的，所以不用按1:1的比例来划分内存空间。而是将内存分为一个较大的Eden区间和两块较小的Survivor区间，每次使用Eden和其中一块Survivor。当回收时，将Eden和Survior中还存活的对象一次性复制到另一块Survivor空间上。最后清除掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认的Eden和Survivor的大小比例是8:1，每次新生代可利用内存为整个新生代内存的90%（80%+10%）。

当然，98%的对象可回收只是一般场景下的数据，我们无法保证每次回收存活对象少于10%，所以当存活对象不能全部放入到10%的Survivor空间时，则需要依赖其他内存，即向老年代进行分配担保（handle promotion）。

#### 标记-整理算法

（老年代内存回收）

复制算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如不想让费50%的空间，就需要有额外的空间进行担保，一应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

标记-整理算法，标记过程和标记-清除算法一致，整理过程是直接将所有存活的对象向一端移动，然后清除掉端边界外的内存。

#### 分代收集算法（Generation Collection）

根据对象的存活周期不同将内存划分为几块，一般讲堆分为新生代和老年代。

新生代：使用复制算法，少量对象复制成本

老年代：使用标记-清除或标记-整理算法，大量对象存活，避免复制

### Hotspot算法实现

#### 枚举根节点

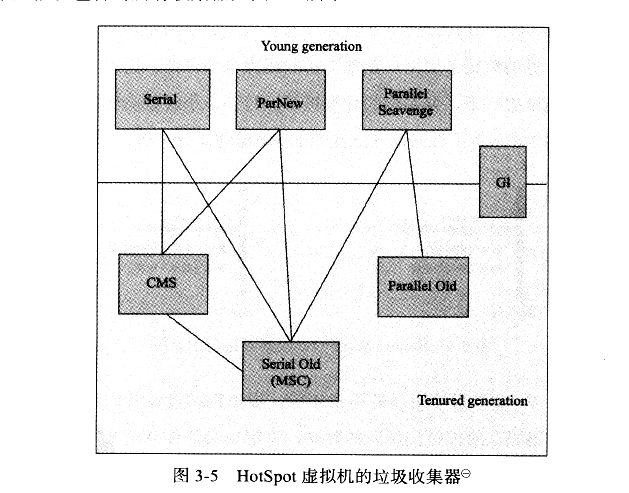
#### 安全点

#### 安全区域

### 垃圾收集器

重点：了解CMS和G1（garbage first）这两块收集器，了解它们部分运作细节

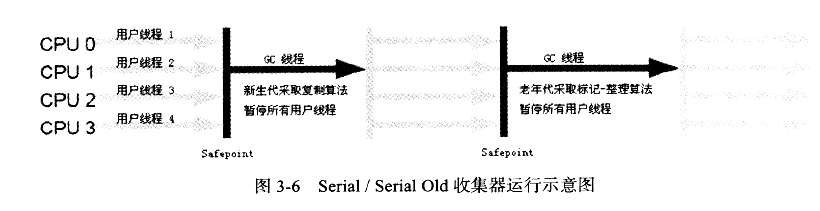
使用时，根据具体的 应用选择合适的回收器。



#### Serial收集器

最基本的、发展历史最悠久的收集器。

是一个单线程收集器。



Serial是Client模式下默认的新生代收集器。

Serial使用单线程回收，没有多线程切换到开销，专心做垃圾收集自然可以获取最高的单线程收集效率。

#### Parnew收集器

Parnew是serial的多线程版本，除了使用多线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数（例如：-XX：SurvivorRatio、-XX:PretenureSizeThreshold、-XX:HandlePromotionFailure等）、收集算法、Stop The World、对象分配规则、回收策略都和Serial收集器一样。



Parnew除了多线程外，相比Serial并没有太多创新。单它是运行在Server模式下首选的垃圾回收器。

更重要的是，除了CMS收集器外，只用Parnew收集器可以和CMS配合使用。

Jdk1.5时期，HosSpot退出了一款在强交互应用中几乎可认为有划时代意义的垃圾收集器——CMS收集器（Concurrent Mark Sweep）,这块收集器是HotSpot虚拟机中第一款真正意思上的并发收集器，它第一次实现了让垃圾收集线程与用于线程（基本上）同时工作。

Unfortunately,CMS和JDK1.4后出现的新生代收集器Parallel Scavenge配合工作，所以在jdk1.5中使用cms，只能和serial和parnew配合使用。

Parnew是使用-XX:+UseConcMarkSweepGC选项后的默认收集器，也可以使用-XX:+UseParNewGC选项来强制指定它。

Parnew收集器使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。

#### Parallel scavenge收集器

Parallel sca是新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器。。。这样说起来和parnew差不多，那区别在哪里？

Ps侧重于吞吐量（throughput）即cpu用于运行用户代码的时间与cpu总消耗时间的比值。

而cms则尽量缩短用户线程的停顿时间。

停顿时间短，更适合 与用户的交互，良好的响应速度提升用户体验。

高吞吐则可以提高cpu利用率，尽快完成程序的运算任务，主要适合后台运算不需要太多交互的任务。

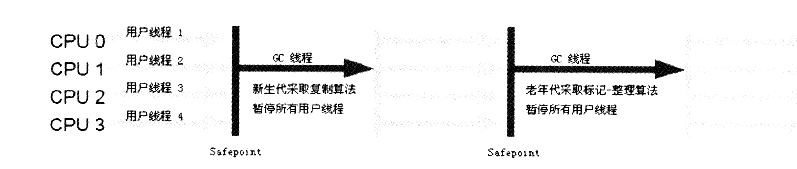
控制最大垃圾收集停顿时间-XX:MaxGCPauseMilli

直接设置吞吐量大小：-XX:GCTimeRatio

-XX：+UseAdpativeSizePolicy,会自动调节-Xmn、Eden和Survivor的比例，晋升到老年代对象的年龄（-XX:PretenureSizeThreshold）等细节参数。

#### Serial old收集器

Serial old是Serial收集器的老年代版本， 它同样是一个单线程收集器，使用“标记-整理”算法。这个收集器的主要意义也是在于给Client模式下的虚拟机使用。如在Server模式下，主要还有两大用途：1）：在jdk1.5以及之前的版本中与parallel scavange收集器配合使用2）：作为CMS收集器的后备方案，在并发收集器Concurrent Mode Failure时使用。

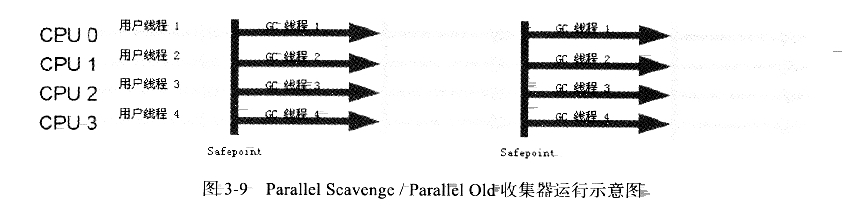


#### Parallel old 收集器

Parallel old是Parallel Scavenge收集器的老年代版本，使用多线程+“标记-整理”算法。这个收集器在Jdk1.6中开始提供的，在此之前，新生代Parallel scavenge处于尴尬位置。

原因是，如果新生代选择了Parallel Scavenge收集器，老年代只能选择Serial old收集器。老年大很大而且硬件条件比较高级的环境下，这种组合吞吐量甚至还一定有parnew加cms组合。

知道Parallel Scavenge收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合。在注重cpu资源敏感的场合，都可以考虑使用Parallel Scavenge加Parallel Old收集器。



#### CMS收集器

CMS（concurrent mark sweep）收集器是为了获取最短回收停顿时间为目的的收集器。很适合互联网或者B/S系统的服务上，这类应用尤其注重服务的响应速度，希望系统停顿时间最短，以带来更好的体验。

CMS是基于“标记-清除”算法实现的，分为以下四个步骤：

1. 初始标记（cms initial mark）
2. 并发标记（concurrent mark）
3. 重新标记（remark）
4. 并发清除（concurrent sweep）

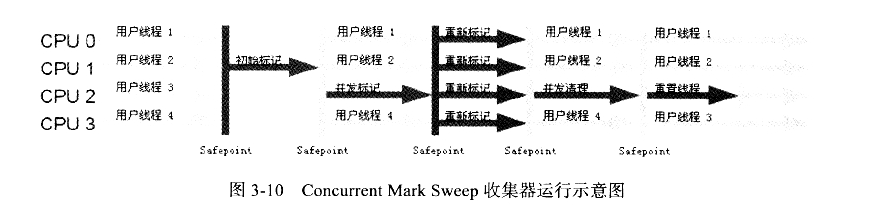
初始标记和重新标记着两个步骤仍需要“Stop The World”。

初始标记：仅仅只是标记一下GC Roots能直接关联到的对象，速度很快。

并发标记：是进行GC Roots Tracing的过程

重新标记：修正并发标记期间因 用户程序继续运作而导致标记产生变动的那部分对象的标记记录，这个标记阶段比初始化标记阶段稍长一点，单远比并发标记的时间短。

由于整个过程中耗时最长的是并发标记和并发清除过程收集器线程都可以和用户线程一起工作，所以，总体来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。



不足：

1. CPU敏感，在并发阶段会占用一部分cpu资源而导致应用程序变慢，总吞吐量降低。Cms默认启动的回收线程数是（cpu数量+3）/4，也就是当cpu在4个以上时，并发回收时的垃圾收集线程不少于25%的cpu资源。当cpu不足4个时，cms对用户程序的影响可能更大。
2. CMS收集器无法处理浮动垃圾（Floating Garbage）,可能出现“Concurrent mode failure”失败而导致另一次fullgc。在并发清理阶段还有用户线程还在运行着，伴随着运行自然就会有新的垃圾不断产生。这部分垃圾无法在当次收集中处理它们。也是因为用户线程还需要运行，那也就需要预留有足够的内存空间给用户线程使用。为了给用户线程预留空间，CMS收集器不能像其他收集器那样等到老年代几乎完全填满了在进行收集。Jdk1.5默认设置下，cms收集器当老年代使用68%的空间就会被激活，这是一个保守设置，如果老年代增速不是很快，可以适当调高-XX:CMSInitiatingOccupancyFraction的值来提高触发百分比。在jdk1.6中，已经提高到92%。 要是CMS运行期预留的内存无法满足程序需要，就会出现“Concurrent mode failure”失败，这时虚拟机启动备选方案，即采用Serial old收集器重新收集老年代，这样停顿时间更长了。所以CMSInitiatingOccupancyFraction过高很容易造成“Concurrent mode failure”。
3. 标记-清除算法很容易产生大量碎片，CMS收集器提供-XX:+UseCMSCompactAtFullCollection开关参数（默认开启），用于CMS在顶不住要进行fgc时开启碎片的合并整理过程。这个过程没法并发。

-XX:CMSFullGCsBeforeCompaction,这个参数设置多少次不压缩的Fullgc后，跟着来一次带压缩的。

### G1收集器

JDK7u4中引入商用G1收集器。

G1(Garbage-First)收集器是当今收集器技术发展的最前沿成果之一。

G1是一款面向服务端应用的垃圾收集器。HotSpot开发团队赋予了它的使命是（在比较长期的）本来可以替换掉1.5中发布的CMS收集器。G1的特点：

1. 并发和并行
2. 分代收集
3. 空间整合，采用“标记-整理”算法，从局部（两个Region）来看是基于“复制”算法实现。
4. 可预测的停顿，region

G1收集器的运作大致分为以下几个步骤：

1. 初始标记
2. 并发标记
3. 重新标记
4. 筛选回收，首先根据Region的回收价值和，成本进行排序，根据用户期望的GC停顿时间来指定回收计划。

G1收集器使用RememberSet来避免全堆扫描。虚拟机发现程序在堆Reference类型进行数据写操作时，会产生一个Write Barrier暂停中断写操作，检查Reference是否处于不同的Region之中，如果是，便通过CardTable把相关引用信息记录到被引用对象所属的Region的RememberSet中。当进行垃圾回收时，在GC根节点的枚举范围中加入RememberedSet即可保证不对全堆扫描也不会有遗漏。

