# 垃圾收集器与内存分配策略

## 3.1 概述

既然已经是自动垃圾回收了，为何要了解GC和内存分配？答案很简单：当需要排查各种内存溢出，内存泄露问题时，当垃圾回收成为系统达到更高并发量的瓶颈时。我就需更好的了解其背后的原理。

## 3.2 判断对象是否已死的方法

### 3.2.1引用计数算法：

给对象中添加一个引用计数器，每当有一个地方要引用他的时候，计数器就加一。当引用失效时，技术器就减一；任何时刻计数器为0的对象就是不能再使用的对象。但是主流java虚拟机并没有使用。其主要原因是应为它很难解决对象之间相互循环引用的问题【具体见P62】

### 3.2.2可达性算法：

Java虚拟机的主流实现。基本思想是通过一系列称为“GC Roots”的对象作为起点，这些节点向下搜索，搜索走过的路径称之为引用链，当一个对象到GC Roots没有任何引用链相连时（Gc Roots到这个对象不可达）时证明对象不可用

### 3.2.3生存还是死亡（自我拯救）：

即使可达性算法不可达对象，也不是非死不可的，此时他们暂存缓刑。要真正宣告一个对象死亡，要经历两次标记过程

如果一个对象再经历一次可达性分析之后发现没有引用连相连，那么将会进行第一次标记并行进行一次筛选，筛选条件是是否执行fiinalize（）方法。若果对象有必要执行该方法那么将会被放置在一个叫F-Queue的队列之中（P66）。Finalize（）方法是对象逃脱死亡的最后一次机会

## 3.3 垃圾回收机制

垃圾回收算法：标记-清除算法

标记清除：最基础的算法（后继算法都是在此基础上改进的），算法分为“标记”，“清除”两个阶段，首先标记处所要回收的对象，标记之后统一回收所被标记的对象。他的不足，一是效率问题，标记和清除效率都不高，宁一个是空间问题，标记之后会产生大量的不连续内存碎片，空间碎片太多会到导致后期分配较大的对象时无法找到足够的空间而不得不提前触发一次垃圾回收动作

复制算法：解决效率问题。它可讲内存按照容量划分大小相等的两块，每次只使用其中一块，当一块内存使用完之后，就讲内存中存活的对象移动到另一块内存上去，然后把使用过的内存空间一次性清楚掉，这样每次都是对整个半区进行内存回收，内存分配也不用考虑内存碎片的问题了。代价：内存代价太高。现在商业虚拟机都是采用这个方法来回收新生代。新生代中对象大多都是朝生夕死，并不需要按照1：1比列来划分内存，而是将内存划分为一块较大的Eden区和两块内存较小的survivor空间，每次使用Eden和其中一块Survivor，当回收时，讲Eden区和Survivor区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和刚才使用的survivor空间。Hotspot默认Eden与Survivor比列是8:1。所以只有10%的内存空间不会使用到分配内存。但是我们并不能保证每次回收都有不多于10%的对象存活，当survivor对象空间不够用时，需要其他内存（这里指老年代）进行分配担保（关于分配担保后继知识会讲解）。

标记—整理算法：复制算法再存活率较高时效率会降低。更关键的是如果不想浪费50%的内存空间，就需要有额外的空间进行分配担保，老年代没有额外的担保空间。所以老年代一般一般不能直接使用这种方法。所以该方法是适用于老年代的算法，算法过程与标记清除一样，但是后继步骤不是直接对可回收对象进行清理，然是让所有存活的对象都向一段移动，然后直接清除掉边界以外的内存。

## 3.4 HotSpot的算法实现

### 3.4.1 枚举根节点

可达性分析对时间的敏感还体现在GC停顿上，因为这项分析工作必须在一个能确保一致性的快照中进行，这里一致性的意思是指在整个分析期间整个执行系统看起来就像被冻结再某个时间点。不可以出现分析过程中对象引用关系还在不断变化的情况，该点不满足的话分析结果准确性就无法得到保证。这点是导致GC进行时必须停止所有的java执行线程（sun讲这个事情称之为stop the world）的其中一个重要的原因，即使在号称几乎不会发生停顿CMS收集器中，枚举节点时也是必须要停顿的。

由于目前主流java虚拟机使用的都是准确式GC(虚拟机知道某个内存的位置具体数据了类型)，所以虚拟机停顿下来的话，并不需要一个不漏的检查所有执行上下文和全局的引用位置。再HotSpot中使用的是OopMap的数据结构来达到这个目的，再类加载完之后，Hotspot就把对象内什么偏移量上是什么类型数据计算出来，再jit编译过程中，也会再特定的位置记录下栈和寄存器中那些位置是引用。

### 3.4.2 安全点

在OopMap的协助下，hotstop可以快速准确的完成GC Roots枚举，但是一个很现实的问题是：可能会导致引用关系的变化，或者说OopMap内容变化的指令非常多，如果为每一条指令都生成对应的OopMap，那将会需要大量的额外空间，这样就会照成GC的空间成本将会变的很高。

实际上，Hotspot并没有为每条指令都生成OopMap,只是再特定的位置上记录了这些信息，这些位置称之为安全点，即程序在执行时并非再所有的地方都能停顿下来GC,只有达到安全点时才能暂停。

如何选定安全点？

安全点太多，GC 过于频繁，增大运行时负荷；安全点太少，GC 等待时间太长。一般会在如下几个位置选择安全点：1、循环的末尾 2、方法临返回前 3、调用方法之后 4、抛异常的位置。

为什么选定这些位置作为安全点：

主要的目的就是避免程序长时间无法进入 Safe Point。比如 JVM 在做 GC 之前要等所有的应用线程进入安全点，如果有一个线程一直没有进入安全点，就会导致 GC 时 JVM 停顿时间延长。比如这里，超大的循环导致执行 GC 等待时间过长。

对于安全点，另一个需要考虑的问题是如何在GC发生时让所有的线程都跑到最近的安全点再停顿下来。这里有两个方案：抢断式中断，主动式中断（P74）

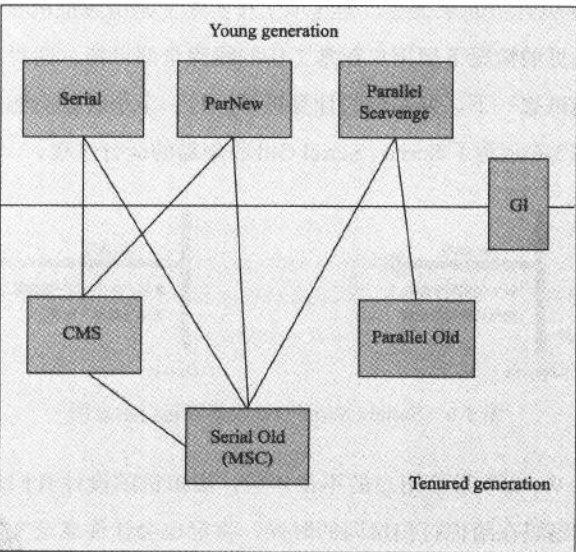
### 3.4.3 安全区域（待完善）

使用安全点似乎已经完美的解决了如何进入GC的问题，但实际情况却并不一定。Safe Point 是对正在执行的线程设定的。如果一个线程处于 Sleep 或中断状态，它就不能响应 JVM 的中断请求，再运行到 Safe Point 上。因此 JVM 引入了 Safe Region。Safe Region 是指在一段代码片段中，引用关系不会发生变化。在这个区域内的任意地方开始 GC 都是安全的。

线程在进入 Safe Region 的时候先标记自己已进入了 Safe Region，等到被唤醒时准备离开 Safe Region 时，先检查能否离开，如果 GC 完成了，那么线程可以离开，否则它必须等待直到收到安全离开的信号为止。

## 3.5 垃圾收集器

本文中的垃圾收集器研究背景为：HotSpot+JDK1.7



如上图所示，垃圾回收算法一共有7个，3个属于年轻代、三个属于年老代，G1属于横跨年轻代和年老代的算法。

7种作用于不同分代的收集器，如果两个收集器之间存在连线，就说明它们可以搭配使用。

JVM会从年轻代和年老代各选出一个算法进行组合，连线表示哪些算法可以组合使用

新生代收集器： Serial收集器 ParNew收集器 Parallel Scavenge收集器

老年代收集器： Serial Old收集器 Parallel Old收集器 CMS收集器（Concurrent Mark Sweep）

GC评价标准：在评估一个GC时，有几个方面需要关注。

1、GC的停顿时间: 大部分GC执行时需要将应用程序停止以便内存保持一个一致的状态来进行回收，这会对应用程序的执行性能造成影响。

2、GC的吞吐量（专注于应用在一段时间的的最大工作量）: 在不面向用户的应用程序中，可能并不关注停顿时间，而是关注总体的执行效率，这时GC的吞吐量可以理解为回收一定内存所需的时间。