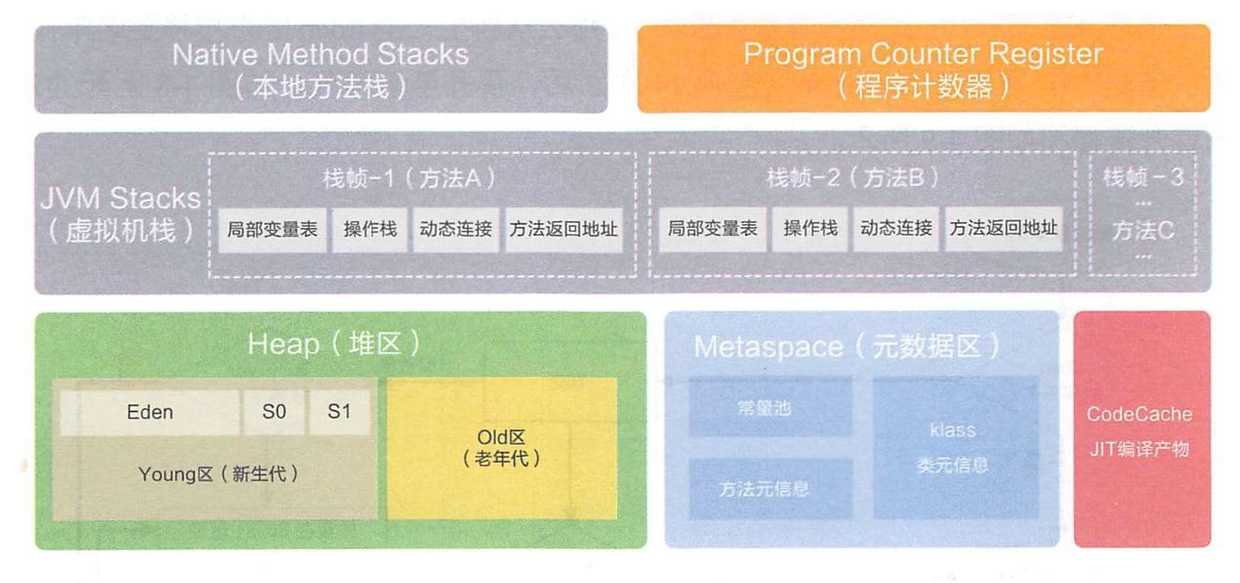
[**Java垃圾回收**](https://www.cnblogs.com/czwbig/p/11127159.html)

本文主要摘自《深入理解Java虚拟机》，内容较多，尽量全面概括了 Java 垃圾回收机制、垃圾回收器以及内存分配策略等内容。了解 Java 垃圾回收之前，需要先了解 [Java内存区域](https://www.cnblogs.com/czwbig/p/11127124.html)

**Java 垃圾回收机制**

**垃圾回收主要关注 Java 堆**

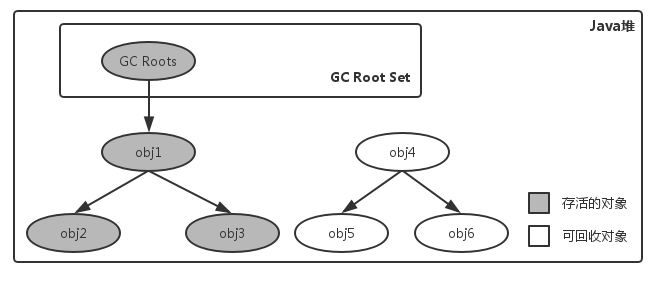


Java 内存运行时区域中的程序计数器、虚拟机栈、本地方法栈随线程而生灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈操作。每一个栈帧中分配多少内存基本上是在类结构确定下来时就已知的（尽管在运行期会由 JIT 编译器进行一些优化），因此这几个区域的内存分配和回收都具备确定性，不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟随着回收了。

而 Java 堆不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，垃圾收集器所关注的是这部分内存。

**判断哪些对象需要被回收**

有以下两种方法：

1. 引用计数法  
   给对象添加一引用计数器，被引用一次计数器值就加 1；当引用失效时，计数器值就减 1；计数器为 0 时，对象就是不可能再被使用的，简单高效，缺点是无法解决对象之间相互循环引用的问题。
2. 可达性分析算法  
   通过一系列的称为 "GC Roots" 的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的。此算法解决了上述循环引用的问题。  
   

在Java语言中，可作为 GC Roots 的对象包括下面几种：  
a. 虚拟机栈（栈帧中的本地变量表）中引用的对象。  
b. 方法区中类静态属性引用的对象。  
c. 方法区中常量引用的对象。  
d. 本地方法栈中 JNI（Native方法）引用的对象

作为 GC Roots 的节点主要在全局性的引用与执行上下文中。要明确的是，tracing gc必须以当前存活的对象集为 Roots，因此必须选取确定存活的引用类型对象。

GC 管理的区域是 Java 堆，虚拟机栈、方法区和本地方法栈不被 GC 所管理，因此选用这些区域内引用的对象作为 GC Roots，是不会被 GC 所回收的。

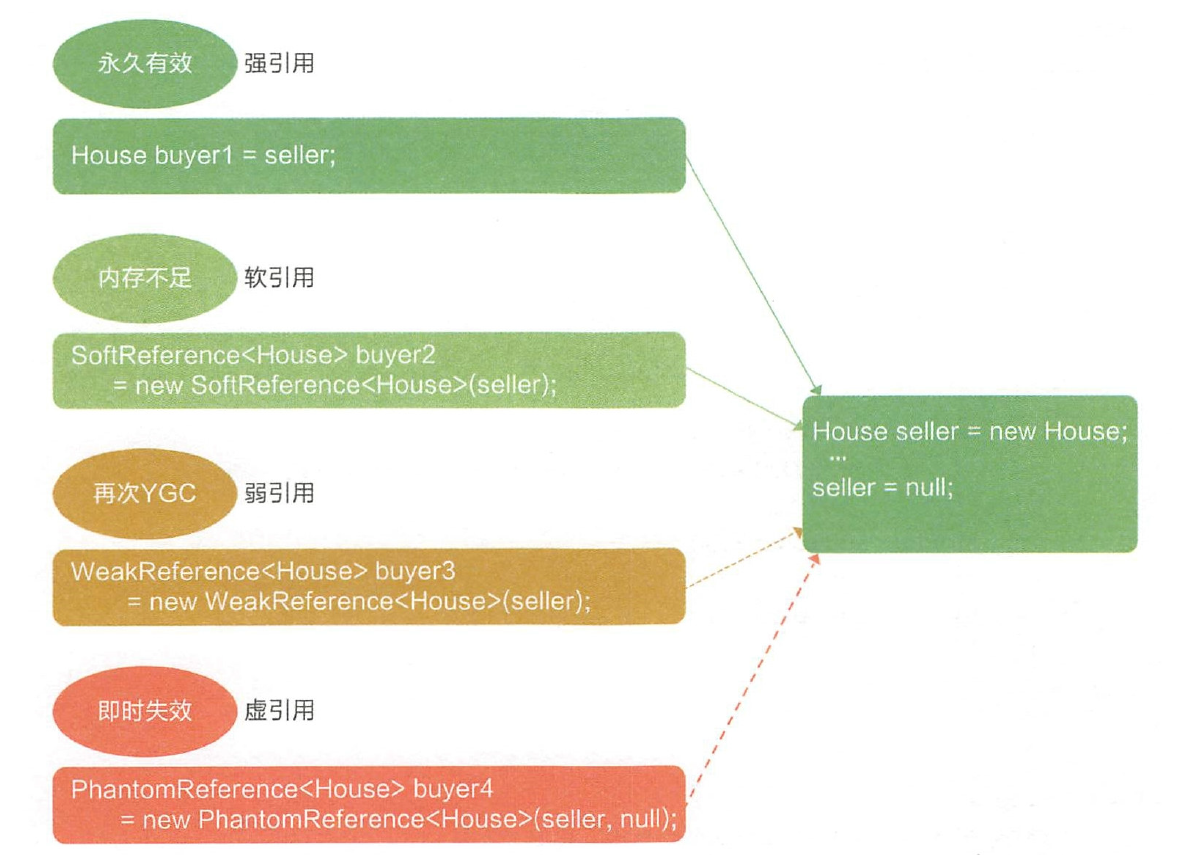
其中虚拟机栈和本地方法栈都是线程私有的内存区域，只要线程没有终止，就能确保它们中引用的对象的存活。而方法区中类静态属性引用的对象是显然存活的。常量引用的对象在当前可能存活，因此，也可能是 GC roots 的一部分。

**强、软、弱、虚引用**

JDK1.2 以前，一个对象只有被引用和没有被引用两种状态。

后来，Java 对引用的概念进行了扩充，将引用分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）4 种，这 4 种引用强度依次逐渐减弱。

1. 强引用就是指在程序代码之中普遍存在的，类似"Object obj=new Object()"这类的引用，垃圾收集器永远不会回收存活的强引用对象。
2. 软引用：还有用但并非必需的对象。在系统 **将要发生内存溢出异常之前** ，将会把这些对象列进回收范围之中进行第二次回收。
3. 弱引用也是用来描述非必需对象的，被弱引用关联的对象 **只能生存到下一次垃圾收集发生之前** 。当垃圾收集器工作时，无论内存是否足够，都会回收掉只被弱引用关联的对象。
4. 虚引用是最弱的一种引用关系。 **无法通过虚引用来取得一个对象实例** 。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。



**可达性分析算法**

不可达的对象将暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：

1. 如果对象在进行可达性分析后发现**没有与 GC Roots 相连接的引用链**，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行 finalize() 方法。
2. 当对象没有覆盖 finalize() 方法，或者 finalize() 方法已经被虚拟机调用过，虚拟机将这两种情况都视为“没有必要执行”，直接进行第二次标记。
3. 如果这个对象被判定为有必要执行 finalize() 方法，那么这个对象将会放置在一个叫做 F-Queue 的队列之中，并在稍后由一个由虚拟机自动建立的、低优先级的 Finalizer 线程去执行它。

这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束，因为如果一个对象在 finalize() 方法中执行缓慢，将很可能会一直阻塞 F-Queue 队列，甚至导致整个内存回收系统崩溃。

测试程序：

Copy

public class FinalizerTest {

public static FinalizerTest object;

public void isAlive() {

System.out.println("I'm alive");

}

@Override

protected void finalize() throws Throwable {

super.finalize();

System.out.println("method finalize is running");

object = this;

}

public static void main(String[] args) throws Exception {

object = new FinalizerTest();

// 第一次执行，finalize方法会自救

object = null;

System.gc();

Thread.sleep(500);

if (object != null) {

object.isAlive();

} else {

System.out.println("I'm dead");

}

// 第二次执行，finalize方法已经执行过

object = null;

System.gc();

Thread.sleep(500);

if (object != null) {

object.isAlive();

} else {

System.out.println("I'm dead");

}

}

}

输出如下：

Copy

method finalize is running

I'm alive

I'm dead

如果不重写 finalize()，输出将会是：

Copy

I'm dead

I'm dead

从执行结果可以看出：  
第一次发生 GC 时，finalize() 方法的确执行了，并且在被回收之前成功逃脱；  
第二次发生 GC 时，由于 finalize() 方法只会被 JVM 调用一次，object 被回收。

值得注意的是，使用 finalize() 方法来“拯救”对象是不值得提倡的，它的运行代价高昂，不确定性大，无法保证各个对象的调用顺序。finalize() 能做的工作，使用 try-finally 或者其它方法都更适合、及时。

本段程序引用自 [Java GC的那些事（上）](http://www.importnew.com/23633.html)

**Java 堆永久代的回收**

永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类。

1. 回收废弃常量与回收 Java 堆中的对象非常类似。以常量池中字面量的回收为例，假如一个字符串"abc"已经进入了常量池中，但是当前系统没有任何一个 String 对象是叫做"abc"的，也没有其他地方引用了这个字面量，如果这时发生内存回收，而且必要的话，这个"abc"常量就会被系统清理出常量池。常量池中的其他类（接口）、方法、字段的符号引用也与此类似。
2. **类需要同时满足下面 3 个条件才能算是“无用的类”：  
   a. 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。  
   b. 加载该类的 ClassLoader 已经被回收。  
   c. 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。**

虚拟机可以对满足上述 3 个条件的无用类进行回收，这里说的仅仅是“可以”，而并不是和对象一样，不使用了就必然会回收。

在大量使用反射、动态代理、CGLib 等 ByteCode 框架、动态生成 JSP 以及 OSGi 这类频繁自定义 ClassLoader 的场景都需要虚拟机具备类卸载的功能，以保证永久代不会溢出。

**垃圾收集算法**

一共有 4 种：

1. 标记-清除算法
2. 复制算法
3. 标记整理算法
4. 分代收集算法

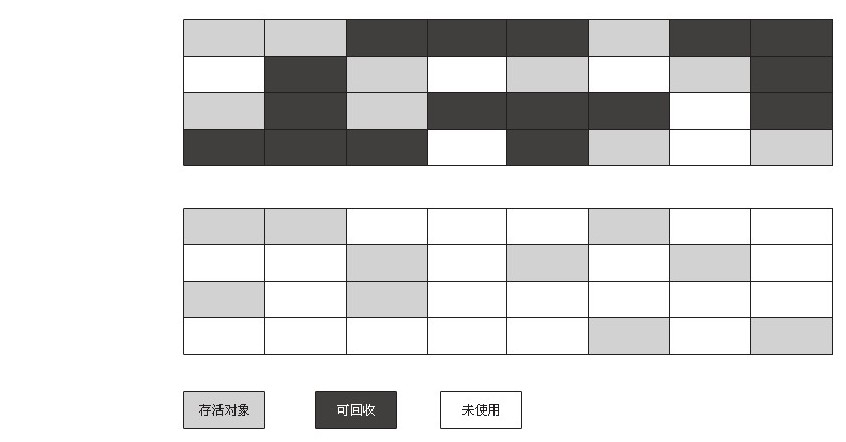
**标记-清除算法**

最基础的收集算法是“标记-清除”（Mark-Sweep）算法，分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

它的主要不足有两个：

1. 效率问题，标记和清除两个过程的效率都不高；
2. 空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

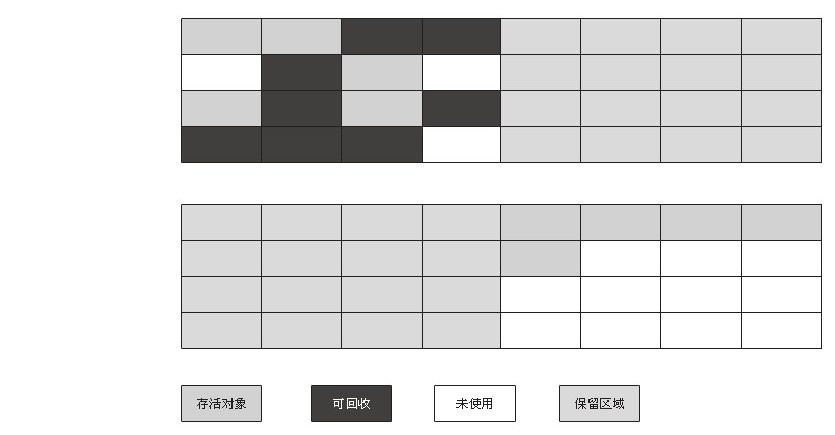
标记—清除算法的执行过程如下图。



**复制算法**

为了解决效率问题，一种称为“复制”（Copying）的收集算法出现了，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为了原来的一半。复制算法的执行过程如下图：



现在的商业虚拟机都采用这种算法来回收新生代，IBM 研究指出新生代中的对象 98% 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:Survivor = 8:1，也就是每次新生代中可用内存空间为整个新生代容量的 90%（其中一块Survivor不可用），只有 10% 的内存会被“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，需要依赖其他内存（这里指老年代）进行分配担保（Handle Promotion）。

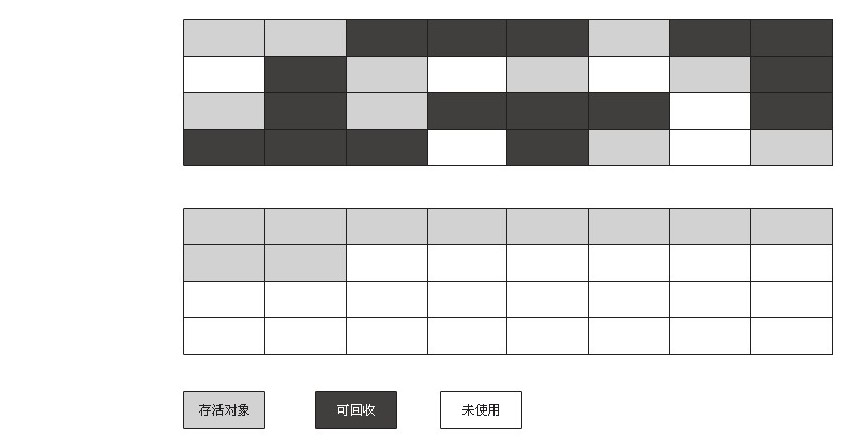
内存的分配担保就好比我们去银行借款，如果我们信誉很好，在 98% 的情况下都能按时偿还，于是银行可能会默认我们下一次也能按时按量地偿还贷款，只需要有一个担保人能保证如果我不能还款时，可以从他的账户扣钱，那银行就认为没有风险了。

内存的分配担保也一样，如果另外一块 Survivor 空间没有足够空间存放上一次新生代收集下来的存活对象时，这些对象将直接通过分配担保机制进入老年代。关于对新生代进行分配担保的内容，在本章稍后在讲解垃圾收集器执行规则时还会再详细讲解。

**标记-整理算法**

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，如果不想浪费 50% 的空间，就需要有额外的空间进行分配担保，以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。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后**直接清理掉端边界以外的内存**，“标记-整理”算法的示意图如下：



**分代收集算法**

当前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，根据对象存活周期的不同将内存划分为几块并采用不用的垃圾收集算法。

一般是把 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记—清理”或者“标记—整理”算法来进行回收。

**HotSpot的算法实现**

**枚举根节点**

以可达性分析中从 GC Roots 节点找引用链这个操作为例，可作为 GC Roots 的节点主要在全局性的引用（例如常量或类静态属性）与执行上下文（例如栈帧中的本地变量表）中，现在很多应用仅仅方法区就有数百兆，如果要逐个检查这里面的引用，那么必然会消耗很多时间。

另外，可达性分析对执行时间的敏感还体现在 GC 停顿上，因为这项分析工作必须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，否则分析结果准确性就无法得到保证。这点是导致 GC 进行时必须停顿所有 Java 执行线程（Sun将这件事情称为"Stop The World"）的其中一个重要原因，即使是在号称（几乎）不会发生停顿的 CMS 收集器中，枚举根节点时也是必须要停顿的。

因此，目前的主流 Java 虚拟机使用的都是准确式 GC（即虚拟机可以知道内存中某个位置的数据具体是什么类型。），所以当执行系统停顿下来后，并不需要一个不漏地检查完所有执行上下文和全局的引用位置，虚拟机应当是有办法直接得知哪些地方存放着对象引用。

在 HotSpot 的实现中，是使用一组称为 OopMap 的数据结构来达到这个目的的，在类加载完成的时候，HotSpot 就把对象内什么偏移量上是什么类型的数据计算出来，在 JIT 编译过程中，也会在特定的位置记录栈和寄存器中哪些位置是引用。这样， GC 在扫描时就可以直接得知这些信息了。

**安全点（Safepoint）**

在 OopMap 的协助下，HotSpot 可以快速且准确地完成 GC Roots 枚举，但一个很现实的问题随之而来：可能导致引用关系变化，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 GC 的空间成本将会变得很高。

实际上，HotSpot 也的确没有为每条指令都生成 OopMap，前面已经提到，只是在“特定的位置”记录了这些信息，这些位置称为安全点，即程序执行时并非在所有地方都能停顿下来开始 GC ，只有在到达安全点时才能暂停。

Safepoint 的选定既不能太少以致于 GC 过少，也不能过于频繁以致于过分增大运行时的负荷。

对于 Safepoint，另一个需要考虑的问题是如何在 GC 发生时让所有线程都“跑”到最近的安全点上再停顿下来。这里有两种方案可供选择：抢先式中断（Preemptive Suspension）和主动式中断（Voluntary Suspension）。

其中抢先式中断不需要线程的执行代码主动去配合，在 GC 发生时，首先把所有线程全部中断，如果发现有线程中断的地方不在安全点上，就恢复线程，让它“跑”到安全点上。现在几乎没有虚拟机实现采用抢先式中断来暂停线程从而响应 GC 事件。

而主动式中断的思想是当 GC 需要中断线程的时候，不直接对线程操作，仅仅简单地设置一个标志，各个线程执行时主动去轮询这个标志，发现中断标志为真时就自己中断挂起。轮询标志的地方和安全点是重合的，另外再加上创建对象需要分配内存的地方。

**安全区域（Safe Region）**

使用 Safepoint 似乎已经完美地解决了如何进入 GC 的问题，但实际情况却并不一定。Safepoint 机制保证了程序执行时，在不太长的时间内就会遇到可进入 GC 的Safepoint。但是，程序“不执行”的时候呢？

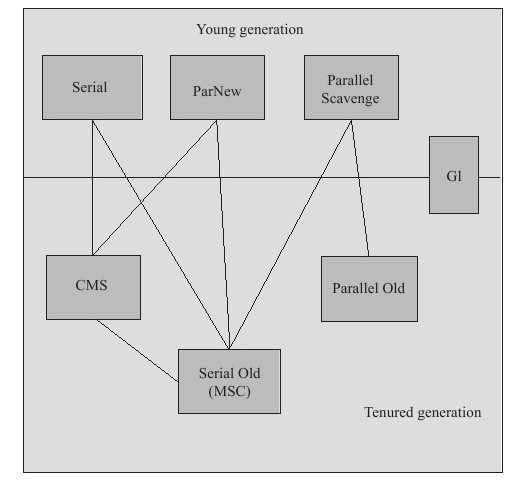
所谓的程序不执行就是没有分配 CPU 时间，典型的例子就是线程处于 Sleep 状态或者 Blocked 状态，这时候线程无法响应 JVM 的中断请求，“走”到安全的地方去中断挂起，JVM 也显然不太可能等待线程重新被分配 CPU 时间。对于这种情况，就需要安全区域（Safe Region）来解决。

**安全区域是指在一段代码片段之中，引用关系不会发生变化。**

在这个区域中的任意地方开始 GC 都是安全的。我们也可以把 Safe Region 看做是被扩展了的 Safepoint。在线程执行到 Safe Region 中的代码时，首先标识自己已经进入了 Safe Region，那样，当在这段时间里 JVM 要发起 GC 时，就不用管标识自己为 Safe Region 状态的线程了。在线程要离开 Safe Region 时，它要检查系统是否已经完成了根节点枚举（或者是整个 GC 过程），如果完成了，那线程就继续执行，否则它就必须等待直到收到可以安全离开 Safe Region 的信号为止。

**垃圾收集器**

如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。这里讨论的收集器基于JDK 1.7 Update 14之后的 HotSpot 虚拟机，这个虚拟机包含的所有收集器如下图所示

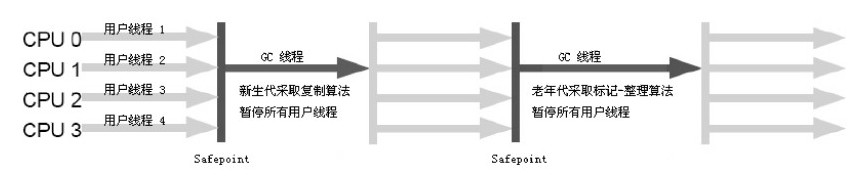


上图展示了 7 种作用于不同分代的收集器，如果两个收集器之间存在连线，就说明它们可以搭配使用。虚拟机所处的区域，则表示它是属于新生代收集器还是老年代收集器。接下来将逐一介绍这些收集器的特性、基本原理和使用场景，并重点分析 CMS 和 G1 这两款相对复杂的收集器，了解它们的部分运作细节。

**Serial收集器（串行收集器）**

Serial 收集器是最基本、发展历史最悠久的收集器，曾经是虚拟机新生代收集的唯一选择。这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。

"Stop The World"这个名字也许听起来很酷，但这项工作实际上是由虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户正常工作的线程全部停掉，这对很多应用来说都是难以接受的。下图示意了 Serial/Serial Old 收集器的运行过程。

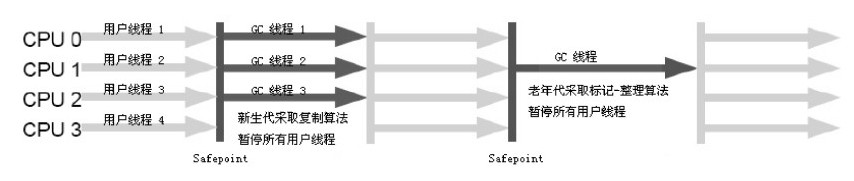


实际上到现在为止，它依然是虚拟机运行在 Client 模式下的默认新生代收集器。它也有着优于其他收集器的地方：简单而高效（与其他收集器的单线程比），对于限定单个 CPU 的环境来说，Serial 收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。

在用户的桌面应用场景中，分配给虚拟机管理的内存一般来说不会很大，收集几十兆甚至一两百兆的新生代（仅仅是新生代使用的内存，桌面应用基本上不会再大了），停顿时间完全可以控制在几十毫秒最多一百多毫秒以内，只要不是频繁发生，这点停顿是可以接受的。所以，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。

**ParNew收集器**

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多条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、回收策略等都与 Serial 收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。ParNew 收集器的工作过程如下图所示。



ParNew 收集器除了多线程收集之外，其他与 Serial 收集器相比并没有太多创新之处，但它却是许多运行在 Server 模式下的虚拟机中首选的新生代收集器，其中有一个与性能无关但很重要的原因是，除了 Serial 收集器外，目前只有它能与 CMS 收集器（并发收集器，后面有介绍）配合工作。

ParNew 收集器在单 CPU 的环境中不会有比 Serial 收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个 CPU 的环境中都不能百分之百地保证可以超越 Serial 收集器。

当然，随着可以使用的 CPU 的数量的增加，它对于 GC 时系统资源的有效利用还是很有好处的。它默认开启的收集线程数与 CPU 的数量相同，在 CPU 非常多（如 32 个)的环境下，可以使用-XX:ParallelGCThreads参数来限制垃圾收集的线程数。

注意，从 ParNew 收集器开始，后面还会接触到几款并发和并行的收集器。这里有必要先解释两个名词：并发和并行。这两个名词都是并发编程中的概念，在谈论垃圾收集器的上下文语境中，它们可以解释如下。

* 并行（Parallel）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
* 并发（Concurrent）：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，而垃圾收集程序运行于另一个 CPU 上。

**Parallel Scavenge收集器**

Parallel Scavenge 收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器……看上去和 ParNew 都一样，那它有什么特别之处呢？

Parallel Scavenge 收集器的特点是它的关注点与其他收集器不同，CMS 等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到一个可控制的吞吐量（Throughput）。

所谓吞吐量就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间），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% 。

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用 CPU 时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Parallel Scavenge收集器提供了两个参数用于精确控制吞吐量，分别是控制最大垃圾收集停顿时间的-XX:MaxGCPauseMillis参数以及直接设置吞吐量大小的-XX:GCTimeRatio参数。

MaxGCPauseMillis参数允许的值是一个大于 0 的毫秒数，收集器将尽可能地保证内存回收花费的时间不超过设定值。

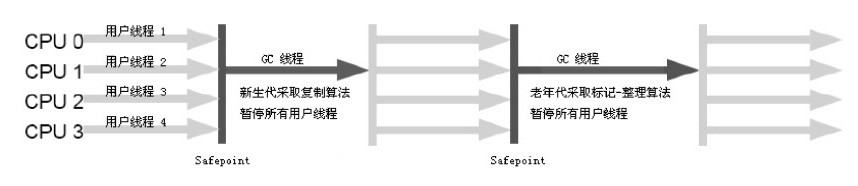
不过大家不要认为如果把这个参数的值设置得稍小一点就能使得系统的垃圾收集速度变得更快，GC停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一些，收集 300MB 新生代肯定比收集 500MB 快吧，这也直接导致垃圾收集发生得更频繁一些，原来10秒收集一次、每次停顿100毫秒，现在变成5秒收集一次、每次停顿70毫秒。停顿时间的确在下降，但吞吐量也降下来了。

GCTimeRatio 参数的值应当是一个 0 到 100 的整数，也就是垃圾收集时间占总时间的比率，相当于是吞吐量的倒数。如果把此参数设置为 19，那允许的最大 GC 时间就占总时间的 5%（即 1/（1+19）），默认值为 99 ，就是允许最大 1%（即 1/（1+99））的垃圾收集时间。

由于与吞吐量关系密切，Parallel Scavenge 收集器也经常称为“吞吐量优先”收集器。除上述两个参数之外，Parallel Scavenge 收集器还有一个参数-XX:+UseAdaptiveSizePolicy值得关注。这是一个开关参数，当这个参数打开之后，就不需要手工指定新生代的大小（-Xmn）、Eden 与 Survivor 区的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，这种调节方式称为 GC 自适应的调节策略（GC Ergonomics）。

**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。如果在 Server 模式下，那么它主要还有两大用途：一种用途是在 JDK 1.5 以及之前的版本中与 Parallel Scavenge 收集器搭配使用，另一种用途就是作为 CMS 收集器的后备预案，在并发收集发生 Concurrent Mode Failure 时使用。这两点都将在后面的内容中详细讲解。Serial Old 收集器的工作过程如下图所示。



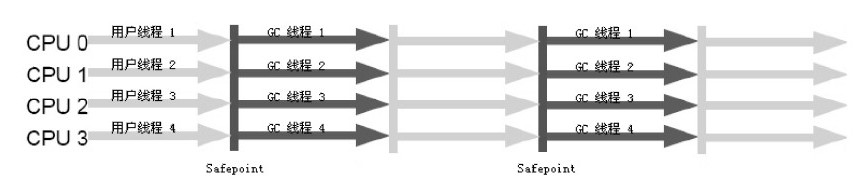
**Parallel Old收集器**

Parallel Old 是 Parallel Scavenge 收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器是在 JDK 1.6 中才开始提供的，在此之前，新生代的 Parallel Scavenge 收集器一直处于比较尴尬的状态。

原因是，如果新生代选择了 Parallel Scavenge 收集器，老年代除了 Serial Old（PS MarkSweep）收集器外别无选择（Parallel Scavenge 收集器无法与 CMS 收集器配合工作）。

由于老年代 Serial Old 收集器在服务端应用性能上的“拖累”，使用了 Parallel Scavenge 收集器也未必能在整体应用上获得吞吐量最大化的效果，由于单线程的老年代收集中无法充分利用服务器多 CPU 的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有 ParNew 加 CMS 的组合“给力”。

直到 Parallel Old 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，在注重吞吐量以及 CPU 资源敏感的场合，都可以优先考虑 Parallel Scavenge 加 Parallel Old 收集器。Parallel Old 收集器的工作过程如下图所示。



**CMS收集器**

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。

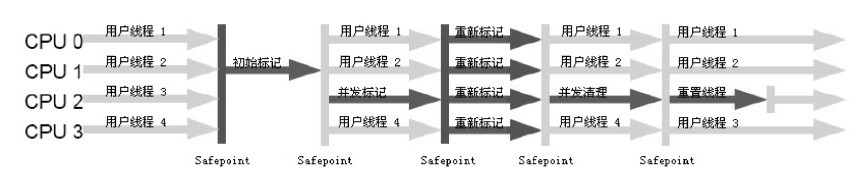
目前很大一部分的 Java 应用集中在互联网站或者 B/S 系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS 收集器就非常符合这类应用的需求。

从名字（包含"Mark Sweep"）上就可以看出，CMS 收集器是基于“标记—清除”算法实现的，它的运作过程相对于前面几种收集器来说更复杂一些，整个过程分为4个步骤，包括：

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

其中，初始标记、重新标记这两个步骤仍然需要"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，并发标记阶段就是进行 GC RootsTracing 的过程，而重新标记阶段则是为了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一部分对象的标记记录，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。

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。



CMS 是一款优秀的收集器，它的主要优点在名字上已经体现出来了：并发收集、低停顿，但是 CMS 还远达不到完美的程度，它有以下 3 个明显的缺点：

**第一、导致吞吐量降低。**CMS 收集器对 CPU 资源非常敏感。其实，面向并发设计的程序都对 CPU 资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或者说CPU资源）而导致应用程序变慢，总吞吐量会降低。

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，并发回收时垃圾收集线程不少于 25% 的 CPU 资源，并且随着 CPU 数量的增加而下降。但是当 CPU 不足 4 个（譬如2个）时，CMS 对用户程序的影响就可能变得很大，如果本来 CPU 负载就比较大，还分出一半的运算能力去执行收集器线程，就可能导致用户程序的执行速度忽然降低了 50%，其实也让人无法接受。

**第二、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 Full GC（新生代和老年代同时回收） 的产生。由于 CMS 并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS 无法在当次收集中处理掉它们，只好留待下一次 GC 时再清理掉。这一部分垃圾就称为“浮动垃圾”。

也是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。

在 JDK 1.5 的默认设置下，CMS 收集器当老年代使用了 68% 的空间后就会被激活，这是一个偏保守的设置，如果在应用中老年代增长不是太快，可以适当调高参数-XX:CMSInitiatingOccupancyFraction的值来提高触发百分比，以便降低内存回收次数从而获取更好的性能，在 JDK 1.6 中，CMS 收集器的启动阈值已经提升至 92% 。

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。所以说参数-XX:CM SInitiatingOccupancyFraction设置得太高很容易导致大量"Concurrent Mode Failure"失败，性能反而降低。

**第三、产生空间碎片。** CMS 是一款基于“标记—清除”算法实现的收集器，这意味着收集结束时会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次 Full GC 。

为了解决这个问题，CMS 收集器提供了一个-XX:+UseCMSCompactAtFullCollection开关参数（默认就是开启的），用于在CMS收集器顶不住要进行 FullGC 时开启内存碎片的合并整理过程，内存整理的过程是无法并发的，空间碎片问题没有了，但停顿时间不得不变长。虚拟机设计者还提供了另外一个参数-XX:CMSFullGCsBeforeCompaction，这个参数是用于设置执行多少次不压缩的 Full GC 后，跟着来一次带压缩的（默认值为0，表示每次进入Full GC时都进行碎片整理）。

**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 开发团队赋予它的使命是（在比较长期的）未来可以替换掉 JDK 1.5 中发布的 CMS 收集器。与其他 GC 收集器相比，G1 具备如下特点。

**并行与并发：** G1 能充分利用多 CPU、多核环境下的硬件优势，使用多个CPU（CPU或者CPU核心）来缩短 Stop-The-World 停顿的时间，部分其他收集器原本需要停顿 Java 线程执行的 GC 动作，G1 收集器仍然可以通过并发的方式让 Java 程序继续执行。

**分代收集：** 与其他收集器一样，分代概念在 G1 中依然得以保留。虽然 G1 可以不需要其他收集器配合就能独立管理整个 GC 堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次 GC 的旧对象以获取更好的收集效果。

**空间整合：** 与 CMS 的“标记—清理”算法不同，G1 从整体来看是基于“标记—整理”算法实现的收集器，从局部（两个 Region 之间）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着 G1 运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次 GC 。

**可预测的停顿：** 这是 G1 相对于 CMS 的另一大优势，降低停顿时间是 G1 和 CMS 共同的关注点，但 G1 除了追求低停顿外，还能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，这几乎已经是实时 Java（RTSJ）的垃圾收集器的特征了。

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，而 G1 不再是这样。使用 G1 收集器时，Java 堆的内存布局就与其他收集器有很大差别，它将整个 Java 堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分 Region （不需要连续）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 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（这也就是Garbage-First名称的来由），保证了 G1 收集器在有限的时间内可以获取尽可能高的收集效率。

在 G1 收集器中，Region 之间的对象引用以及其他收集器中的新生代与老年代之间的对象引用，虚拟机都是使用 **Remembered Set** 来避免全堆扫描的。

G1 中每个Region 都有一个与之对应的 Remembered Set，虚拟机发现程序在对 Reference 类型的数据进行写操作时，会产生一个 Write Barrier 暂时中断写操作，检查 Reference 引用的对象是否处于不同的 Region 之中（在分代的例子中就是检查是否老年代中的对象引用了新生代中的对象），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 Remembered Set 之中。当进行内存回收时，在 GC 根节点的枚举范围中加入 Remembered Set 即可保证不对全堆扫描也不会有遗漏。

如果不计算维护 Remembered Set 的操作，G1 收集器的运作大致可划分为以下几个步骤：

1. 初始标记（Initial Marking）
2. 并发标记（Concurrent Marking）
3. 最终标记（Final Marking）
4. 筛选回收（Live Data Counting and Evacuation）

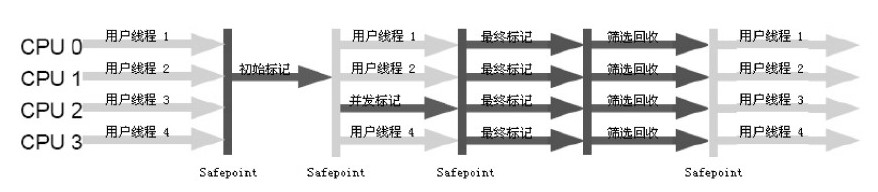
G1 的前几个步骤的运作过程和 CMS 有很多相似之处。

初始标记阶段仅仅只是标记一下 GC Roots 能直接关联到的对象，并且修改 TAMS（Next Top at Mark Start）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，这阶段需要停顿线程，但耗时很短。

并发标记阶段是从 GC Root 开始对堆中对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可与用户程序并发执行。

而最终标记阶段则是为了修正在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一部分标记记录，虚拟机将这段时间对象变化记录在线程 Remembered Set Logs 里面，最终标记阶段需要把 Remembered Set Logs 的数据合并到 Remembered Set 中，这阶段需要停顿线程，但是可并行执行。

最后在筛选回收阶段首先对各个 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，从Sun公司透露出来的信息来看，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。通过下图可以比较清楚地看到G1收集器的运作步骤中并发和需要停顿的阶段。



**GC日志**

阅读 GC 日志是处理 Java 虚拟机内存问题的基础技能，它只是一些人为确定的规则，没有太多技术含量。

每一种收集器的日志形式都是由它们自身的实现所决定的，换而言之，每个收集器的日志格式都可以不一样。但虚拟机设计者为了方便用户阅读，将各个收集器的日志都维持一定的共性，例如以下两段典型的 GC 日志：

Copy

33.125:[GC[DefNew:3324K-＞152K（3712K），0.0025925 secs]3324K-＞152K（11904K），0.0031680 secs]

100.667:[Full GC[Tenured:0 K-＞210K（10240K），0.0149142secs]4603K-＞210K（19456K），[Perm:2999K-＞2999K（21248K）]，0.0150007 secs][Times:user=0.01 sys=0.00，real=0.02 secs]

最前面的数字33.125： 和 100.667： 代表了 GC 发生的时间，这个数字的含义是从 Java 虚拟机启动以来经过的秒数。

GC 日志开头的 [GC 和 [Full GC 说明了这次垃圾收集的停顿类型，而不是用来区分新生代 GC 还是老年代 GC 的。

如果有 Full ，说明这次 GC 是发生了 Stop-The-World 的，例如下面这段新生代收集器 ParNew 的日志也会出现 [Full GC（这一般是因为出现了分配担保失败之类的问题，所以才导致 STW）。如果是调用 System.gc() 方法所触发的收集，那么在这里将显示 [Full GC（System）。

Copy

[Full GC 283.736:[ParNew:261599K-＞261599K（261952K），0.0000288 secs]

接下来的 [DefNew、[Tenured、[Perm 表示 GC 发生的区域，这里显示的区域名称与使用的 GC 收集器是密切相关的，例如上面样例所使用的 Serial 收集器中的新生代名为 "Default New Generation"，所以显示的是 [DefNew。如果是 ParNew 收集器，新生代名称就会变为 [ParNew，意为 "Parallel New Generation"。如果采用 Parallel Scavenge 收集器，那它配套的新生代称为 PSYoungGen，老年代和永久代同理，名称也是由收集器决定的。

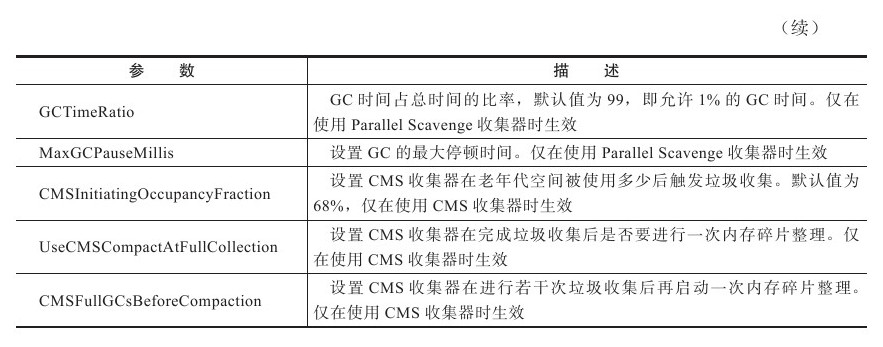
后面方括号内部的 3324K-＞152K（3712K）含义是GC 前该内存区域已使用容量 -＞ GC 后该内存区域已使用容量 （该内存区域总容量）。而在方括号之外的 3324K-＞152K（11904K） 表示 GC 前 Java 堆已使用容量 -＞ GC 后 Java 堆已使用容量 （Java 堆总容量）。

再往后，0.0025925 secs 表示该内存区域 GC 所占用的时间，单位是秒。有的收集器会给出更具体的时间数据，如 [Times:user=0.01 sys=0.00，real=0.02 secs] ，这里面的 user、sys 和 real 与 Linux 的 time 命令所输出的时间含义一致，分别代表用户态消耗的 CPU 时间、内核态消耗的 CPU 事件和操作从开始到结束所经过的墙钟时间（Wall Clock Time）。

CPU 时间与墙钟时间的区别是，墙钟时间包括各种非运算的等待耗时，例如等待磁盘 I/O、等待线程阻塞，而 CPU 时间不包括这些耗时，但当系统有多 CPU 或者多核的话，多线程操作会叠加这些 CPU 时间，所以读者看到 user 或 sys 时间超过 real 时间是完全正常的。

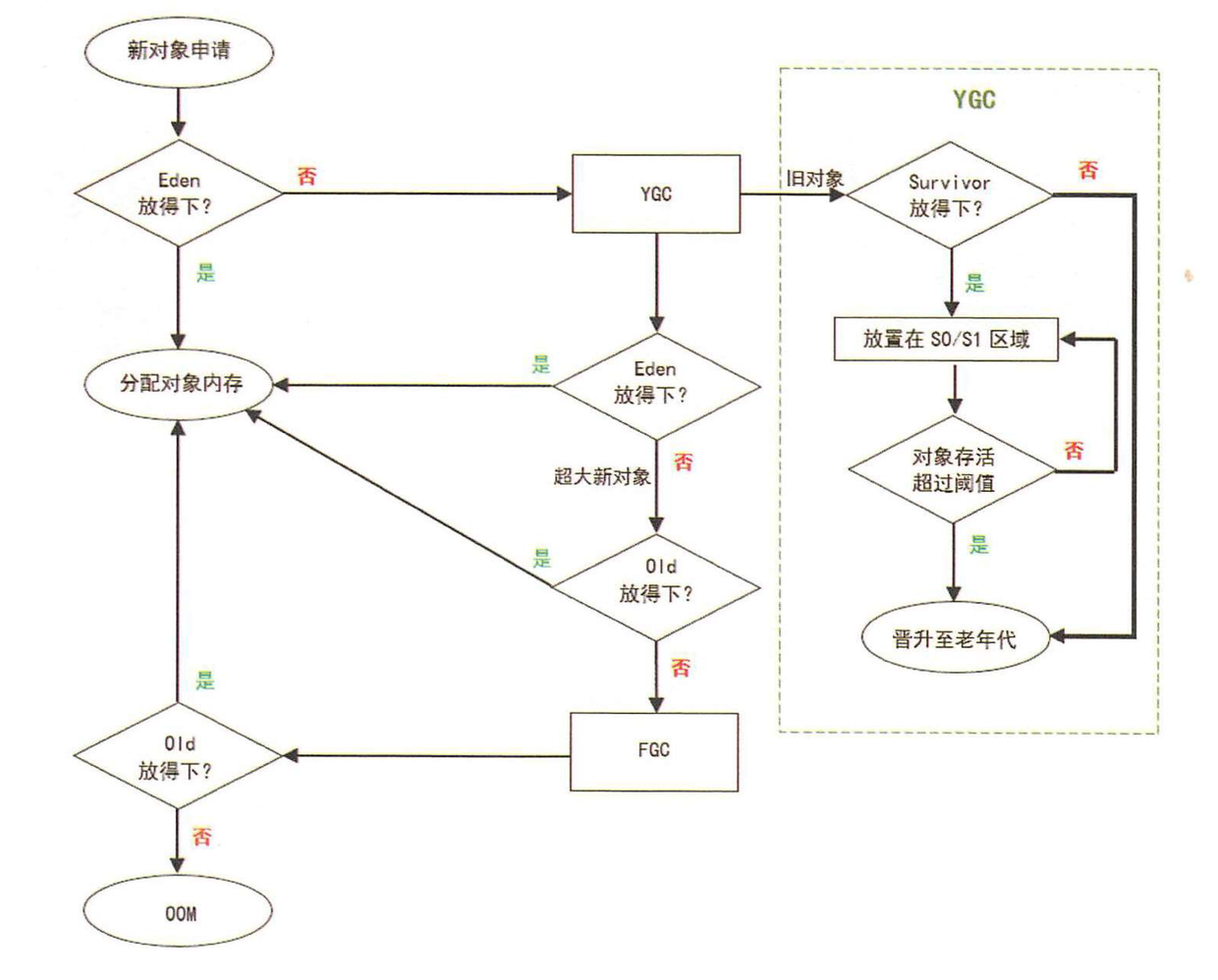
**垃圾收集器参数总结**

JDK 1.7 中的各种垃圾收集器到此已全部介绍完毕，在描述过程中提到了很多虚拟机非稳定的运行参数，在表3-2中整理了这些参数供读者实践时参考。

**内存分配与回收策略**

对象的内存分配，往大方向讲，就是在堆上分配，对象主要分配在新生代的Eden区上。少数情况下也可能会直接分配在老年代中，分配的规则并不是百分之百固定的，其细节取决于当前使用的是哪一种垃圾收集器组合，还有虚拟机中与内存相关的参数的设置。



**对象优先在Eden分配**

大多数情况下，对象在新生代 Eden 区中分配。当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC。

虚拟机提供了-XX:+PrintGCDetails这个收集器日志参数，告诉虚拟机在发生垃圾收集行为时打印内存回收日志，并且在进程退出的时候输出当前的内存各区域分配情况。

Copy

private static final int\_1MB=1024 \* 1024；

/\*\*

\*VM参数：-verbose:gc-Xms20M-Xmx20M-Xmn10M-XX:+PrintGCDetails

-XX:SurvivorRatio=8

\*/

public static void testAllocation () {

byte[] allocation1，allocation2，allocation3，allocation4；

allocation1 = new byte[2 \* \_1MB]；

allocation2 = new byte[2 \* \_1MB]；

allocation3 = new byte[2 \* \_1MB]；

allocation4 = new byte[4 \* \_1MB]；//出现一次Minor GC

}

运行结果：

Copy

[GC[DefNew:6651K-＞148K（9216K），0.0070106 secs]6651K-＞6292K（19456K），

0.0070426 secs][Times:user=0.00 sys=0.00，real=0.00 secs]

Heap

def new generation total 9216K,used 4326K[0x029d0000，0x033d0000，0x033d0000）

eden space 8192K，51%used[0x029d0000，0x02de4828，0x031d0000）

from space 1024K，14%used[0x032d0000，0x032f5370，0x033d0000）

to space 1024K，0%used[0x031d0000，0x031d0000，0x032d0000）

tenured generation total 10240K,used 6144K[0x033d0000，0x03dd0000，0x03dd0000）

the space 10240K，60%used[0x033d0000，0x039d0030，0x039d0200，0x03dd0000）

compacting perm gen total 12288K,used 2114K[0x03dd0000，0x049d0000，0x07dd0000）

the space 12288K，17%used[0x03dd0000，0x03fe0998，0x03fe0a00，0x049d0000）

No shared spaces configured.

上方代码的 testAllocation() 方法中，尝试分配 3 个 2MB 大小和 1 个 4MB 大小的对象，在运行时通过-Xms20M、-Xmx20M、-Xmn10M这 3 个参数限制了 Java 堆大小为 20MB ，不可扩展，其中 10MB 分配给新生代，剩下的 10MB 分配给老年代。-XX:SurvivorRatio=8决定了新生代中 Eden 区与一个 Survivor 区的空间比例是 8:1，从输出的结果也可以清晰地看到 eden space 8192K、from space 1024K、to space 1024K 的信息，新生代总可用空间为 9216KB（Eden区+1个Survivor区的总容量）。

执行 testAllocation() 中分配 allocation4 对象的语句时会发生一次 Minor GC，这次 GC 的结果是新生代 6651KB 变为 148KB ，而总内存占用量则几乎没有减少（因为 allocation1、allocation2、allocation3 三个对象都是存活的，虚拟机几乎没有找到可回收的对象）。

这次 GC 发生的原因是给 allocation4 分配内存的时候，发现 Eden 已经被占用了 6MB，剩余空间已不足以分配 allocation4 所需的 4MB 内存，因此发生 Minor GC。GC 期间虚拟机又发现已有的 3 个 2MB 大小的对象全部无法放入 Survivor 空间（Survivor 空间只有 1MB 大小），所以只好通过分配担保机制提前转移到老年代去。

这次 GC 结束后，4MB 的 allocation4 对象顺利分配在 Eden 中，因此程序执行完的结果是 Eden 占用 4MB（被allocation4占用），Survivor 空闲，老年代被占用 6MB（被allocation1、allocation2、allocation3占用）。通过 GC 日志可以证实这一点。

**Minor GC 和 Full GC 有什么不一样吗？**

* 新生代 GC（Minor GC）：指发生在新生代的垃圾收集动作，因为 Java 对象大多都具备朝生夕灭的特性，所以 Minor GC 非常频繁，一般回收速度也比较快。
* 老年代 GC（Major GC/Full GC）：指发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次的 Minor GC（但非绝对的，在 Parallel Scavenge 收集器的收集策略里就有直接进行 Major GC 的策略选择过程）。Major GC 的速度一般会比 Minor GC 慢 10 倍以上。

**大对象直接进入老年代**

所谓的大对象是指，需要大量连续内存空间的 Java 对象，最典型的大对象就是那种很长的字符串以及数组（ byte[] 数组就是典型的大对象）。大对象对虚拟机的内存分配来说就是一个坏消息（特别是短命大对象，写程序的时候应当避免），经常出现大对象容易导致内存还有不少空间时就提前触发垃圾收集以获取足够的连续空间来“安置”它们。

虚拟机提供了一个 -XX:PretenureSizeThreshold 参数，令大于这个设置值的对象直接在老年代分配。这样做的目的是避免在 Eden 区及两个 Survivor 区之间发生大量的内存复制。

Copy

private static final int\_1MB=1024 \* 1024；

/\*\*

\*VM参数：-verbose:gc-Xms20M-Xmx20M-Xmn10M-XX:+PrintGCDetails-XX:SurvivorRatio=8

\*-XX:PretenureSizeThreshold=3145728

\*/

public static void testPretenureSizeThreshold () {

byte[] allocation；

allocation = new byte[4 \* \_1MB]；//直接分配在老年代中

}

运行结果：

Copy

Heap

def new generation total 9216K,used 671K[0x029d0000，0x033d0000，0x033d0000）

eden space 8192K，8%used[0x029d0000，0x02a77e98，0x031d0000）

from space 1024K，0%used[0x031d0000，0x031d0000，0x032d0000）

to space 1024K，0%used[0x032d0000，0x032d0000，0x033d0000）

tenured generation total 10240K,used 4096K[0x033d0000，0x03dd0000，0x03dd0000）

the space 10240K，40%used[0x033d0000，0x037d0010，0x037d0200，0x03dd0000）

compacting perm gen total 12288K,used 2107K[0x03dd0000，0x049d0000，0x07dd0000）

the space 12288K，17%used[0x03dd0000，0x03fdefd0，0x03fdf000，0x049d0000）

No shared spaces configured.

执行以上代码中的 testPretenureSizeThreshold() 方法后，我们看到 Eden 空间几乎没有被使用，而老年代的 10MB 空间被使用了 40%，也就是 4MB 的 allocation 对象直接就分配在老年代中，这是因为 PretenureSizeThreshold 参数被设置为 3MB（就是 3145728，这个参数不能像 -Xmx 之类的参数一样直接写 3MB），因此超过 3MB 的对象都会直接在老年代进行分配。

注意 PretenureSizeThreshold 参数只对 Serial 和 ParNew 两款收集器有效，Parallel Scavenge 收集器不认识这个参数，Parallel Scavenge 收集器一般并不需要设置。如果遇到必须使用此参数的场合，可以考虑 ParNew 加 CMS 的收集器组合。

**长期存活的对象将进入老年代**

虚拟机给每个对象定义了一个对象年龄（Age）计数器。

如果对象在 Eden 出生并经过第一次 Minor GC 后仍然存活，并且能被 Survivor 容纳的话，将被移动到 Survivor 空间中，并且对象年龄设为 1 。对象在 Survivor 区中每“熬过”一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认为 15 岁），就将会被晋升到老年代中。

对象晋升老年代的年龄阈值，可以通过参数-XX:MaxTenuringThreshold设置。

**动态对象年龄判定**

为了能更好地适应不同程序的内存状况，无须等到 MaxTenuringThreshold 中要求的年龄，同年对象达到 Survivor 空间的一半后，他们以及年龄大于他们的对象都将直接进入老年代。

**空间分配担保**

在发生 Minor GC 之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代  
所有对象总空间，如果这个条件成立，那么 Minor GC 可以确保是安全的。

只要老年代的连续空间大于新生代对象总大小或者历次晋升的平均大小就会进行 Minor GC ，否则将进行 Full GC 。

感谢：  
[深入理解Java虚拟机（第2版）](https://book.douban.com/subject/24722612/)

如果觉得本文有所帮助，欢迎点【推荐】！文章错误之处烦请留言。 转载说明：转载后必须在文章开头明显地给出作者和原文链接；引用必须注明出处；需要二次修改发布请联系作者征得同意。

分类: [JVM](https://www.cnblogs.com/czwbig/category/1495542.html)

https://www.cnblogs.com/czwbig/p/11127159.html