# 运行时数据区

## 分区图



## 程序计数器

* 程序计数器是一块较小的内存空间，它可以看作是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里，字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。
* 由于Java虚拟机的多线程是通过线程轮流切换并分配处理器执行时间的方式来实现的，所以在任何一个确定的时刻，一个处理器（对于多核处理器来说是一个内核）都只会执行一条线程中的指令。因此，为了线程切换后能够恢复到正确的执行位置，每条线程都需要有一个独立的程序计数器。
* 各条线程之间计数器互不影响，独立存储，我们称这类内存区域为“ 线程私有 ”的内存。

## Java虚拟机栈

* 线程私有
* 虚拟机栈描述的是Java方法执行的内存模型：每个方法在执行的同时都会创建一个栈桢（Stack Frame）用于存储该方法的信息，如局部变量表、操作数栈、方法的出口等信息。
* 每个方法从调用直至执行完成的过程，就对应着一个栈桢入栈到出栈的过程。
* 局部变量表存放了编译器可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（reference类型，不是对象本身）和returnAddress类型（指向了一条字节码指令的位置）。

## 本地方法栈

* 本地方法栈与虚拟机栈的作用非常相似，它们的区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则为了虚拟机使用到的Native方法服务。
* 在虚拟机规范中队本地方法栈方法使用的语言、使用的法师与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它。甚至有的虚拟机（如 Sun HotSpot 虚拟机）直接就把本地方法栈和虚拟机栈合二为一。

## Java堆

* Java堆是Java虚拟机所管理的内存中最大的一块，该内存是被所有线程共享的，在虚拟机启动时创建。
* 此内存区域存在的唯一目的就是存放对象实例及数组，几乎所有的对象实例及数组都在这里分配内存。既然Java堆中存放了几乎所有的实例，那么这里自然就成了垃圾收集器管理的主要区域。
* 根据Java虚拟机规范规定，Java堆可以处于物理上不连续的内存空间中，只要逻辑上连续即可。既可以实现成固定大小，也可以是扩展的。
  + 1. -Xmx JVM最大允许分配的堆内存，按需分配
    2. -Xms JVM初始分配的堆内存

## 方法区

* 方法区与Java堆一样，是各个线程共享的内存区域。
* 它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。
* 虽然Java虚拟机规范把方法区描述为堆的一个逻辑部分，但它却有一个别名叫Non-Heap（非堆）
* 方法区还包括了运行时常量池。

-XX:PermSize=64M JVM初始分配的非堆内存  
-XX:MaxPermSize=128M JVM最大允许分配的非堆内存，按需分配

## 运行时常量池

* 运行时常量池属于方法区的一部分。虽然Java虚拟机没有单独为运行时常量池划分内存空间，但是由于该区域在平时开发中也是一个重要的部分，所以也为它单独列一个条目。
* Class文件中除了有类的版本、字段、方法、接口的描述信息外，还有一项信息是常量池（Constant Pool Table），用于存放编译期生成的各种字面量的引用，这部分内容将在类加载后进入方法区的运行时常量池中存放。
* 运行时常量池相对于Class文件常量池的另外一个重要特征是具有动态性，Java语言并不要求常量一定只有编译期才能产生，也就是说并非预置入Class文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，如String类的intern()方法。

# Java内存模型

## 内存模型图

* + 1. HashMap

# 垃圾回收

## 引用

* + 1. 强引用

object

* + 1. 软引用

SoftReference

* + 1. 弱引用
    2. 虚引用

## 垃圾回收算法

* + 1. 标记-清除算法
    2. 复制回收算法
    3. 标记-整理算法

## Minor GC vs Major GC vs Full GC

* + 1. Minor GC

在年轻代Young space(包括Eden区和Survivor区)中的垃圾回收称之为 **Minor GC**. 这个定义既清晰又无异议。 但仍有一些有趣的关于Minor GC事件的东西你需要了解：

1. Minor GC总是在不能为新的对象分配空间的时候触发, 例如 Eden区满了，分配空间的越快，Minor GC越频繁。
2. 当内存池慢了后, 它的完整的内容会被复制出去，指针可以从0开始重新跟踪空闲内存。所以取代传统的标记-交换-压缩（Mark, Sweep , Compact）, Eden区和Survivor区使用标记-复制方式（Mark , Copy）. 因此在Eden区和Survivor区无内存碎片。写指针总是指向内存池的顶部。
3. 在Minor GC时, 年老代（Tenured generation）可以被忽略. 年老代对年轻代的引用被认为是实际的GC根root。 在标记阶段年轻代对年老代的引用可以被简单的忽略。
4. **出于常理, 所有的Minor GC都会触发stop-the-world暂停, 它意味着会暂停应用的所有线程**. 对于大部分应用而言，暂停的延迟可以忽略不计。这是因为Eden中大部分的对象都可以垃圾回收掉，而不会被复制到Survivor/Old区。但如果相反，大部分的新对象不能被回收， Minor GC暂停会占用更多的时间。

综上所述，Minor GC概念相当清晰 – **每次Minor GC只会清理年轻代**.

## Major GC vs Full GC

有人可能会注意到没有关于Major GC和Full GC正式的定义， 即使在JVM规范和垃圾回收论文中也没有。但是轻轻一瞥，从我们对Minor GC定义上来看， 它们的定义也应该很简单:

* **Major GC** 清理年老区（Tenured space）.
* **Full GC** 清理整个内存堆 – 既包括年轻代也包括年老代.

不幸的是， 它有点复杂和令人不解. 首先来说，很多Major GC都是由Minor GC触发的,所以很多情况下将这两个概念分开是不可能的，另一方面，很多现代的垃圾回收会部分的执行年老代（Tenured space）清理，所以使用清理这个词也只能部分的正确。

这会引导我们了解到这一点： **与其担心GC被称作 Major 还是 Full GC, 你更应该关心GC是否会暂停程序的所有线程，还是和应用程序并行的处理**.

这种困惑甚至内置于JVM的标准工具中. 最好通过例子来说明. 让我们比较一下两个GC跟踪工具的输出,此时JVM使用Concurrent Mark and Sweep collector (-XX:+UseConcMarkSweepGC)

首先看一下[jstat](http://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstat.html) 的输出:

|  |  |
| --- | --- |
| 1 | my-precious: me$ jstat -gc -t 4235 1s |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Time S0C S1C S0U S1U EC EU OC OU MC MU CCSC CCSU YGC YGCT FGC FGCT GCT  5.7 34048.0 34048.0 0.0 34048.0 272640.0 194699.7 1756416.0 181419.9 18304.0 17865.1 2688.0 2497.6 3 0.275 00.000 0.275  6.7 34048.0 34048.0 34048.0 0.0 272640.0 247555.4 1756416.0 263447.9 18816.0 18123.3 2688.0 2523.1 4 0.359 00.000 0.359  7.7 34048.0 34048.0 0.0 34048.0 272640.0 257729.3 1756416.0 345109.8 19072.0 18396.6 2688.0 2550.3 5 0.451 00.000 0.451  8.7 34048.0 34048.0 34048.0 34048.0 272640.0 272640.0 1756416.0 444982.5 19456.0 18681.3 2816.0 2575.8 70.550 0 0.000 0.550  9.7 34048.0 34048.0 34046.7 0.0 272640.0 16777.0 1756416.0 587906.3 20096.0 19235.1 2944.0 2631.8 8 0.720 00.000 0.720  10.7 34048.0 34048.0 0.0 34046.2 272640.0 80171.6 1756416.0 664913.4 20352.0 19495.9 2944.0 2657.4 9 0.810 00.000 0.810  11.7 34048.0 34048.0 34048.0 0.0 272640.0 129480.8 1756416.0 745100.2 20608.0 19704.5 2944.0 2678.4 10 0.8960 0.000 0.896  12.7 34048.0 34048.0 0.0 34046.6 272640.0 164070.7 1756416.0 822073.7 20992.0 19937.1 3072.0 2702.8 11 0.9780 0.000 0.978  13.7 34048.0 34048.0 34048.0 0.0 272640.0 211949.9 1756416.0 897364.4 21248.0 20179.6 3072.0 2728.1 12 1.0871 0.004 1.091  14.7 34048.0 34048.0 0.0 34047.1 272640.0 245801.5 1756416.0 597362.6 21504.0 20390.6 3072.0 2750.3 13 1.1832 0.050 1.233  15.7 34048.0 34048.0 0.0 34048.0 272640.0 21474.1 1756416.0 757347.0 22012.0 20792.0 3200.0 2791.0 15 1.3362 0.050 1.386  16.7 34048.0 34048.0 34047.0 0.0 272640.0 48378.0 1756416.0 838594.4 22268.0 21003.5 3200.0 2813.2 16 1.4332 0.050 1.484 |

这个片段摘自JVM启动的前17秒。基于这些信息我们可以得出结论， 经过12次Minor GC后运行了两次Full GC，总共花费50ms （译者按：查看YGC和FGC数）. 通过GUI工具你也应该能得到相同的信息，比如 [jconsole](http://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html) 或 [jvisualvm](http://download.oracle.com/javase/6/docs/technotes/tools/share/jvisualvm.html).

在得出我们的结论之前，让我们看一下同样的JVM启动时垃圾回收日志的输出，显然-XX:+PrintGCDetails可以告诉我们垃圾回收器工作的细节:

|  |  |
| --- | --- |
| 1 | java -XX:+PrintGCDetails -XX:+UseConcMarkSweepGC eu.plumbr.demo.GarbageProducer |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | 3.157: [GC (Allocation Failure) 3.157: [ParNew: 272640K->34048K(306688K), 0.0844702 secs] 272640K->69574K(2063104K), 0.0845560 secs] [Times: user=0.23 sys=0.03, real=0.09 secs]  4.092: [GC (Allocation Failure) 4.092: [ParNew: 306688K->34048K(306688K), 0.1013723 secs] 342214K->136584K(2063104K), 0.1014307 secs] [Times: user=0.25 sys=0.05, real=0.10 secs]  ... cut for brevity ...  11.292: [GC (Allocation Failure) 11.292: [ParNew: 306686K->34048K(306688K), 0.0857219 secs] 971599K->779148K(2063104K), 0.0857875 secs] [Times: user=0.26 sys=0.04, real=0.09 secs]  12.140: [GC (Allocation Failure) 12.140: [ParNew: 306688K->34046K(306688K), 0.0821774 secs] 1051788K->856120K(2063104K), 0.0822400 secs] [Times: user=0.25 sys=0.03, real=0.08 secs]  12.989: [GC (Allocation Failure) 12.989: [ParNew: 306686K->34048K(306688K), 0.1086667 secs] 1128760K->931412K(2063104K), 0.1087416 secs] [Times: user=0.24 sys=0.04, real=0.11 secs]  13.098: [GC (CMS Initial Mark) [1 CMS-initial-mark: 897364K(1756416K)] 936667K(2063104K), 0.0041705 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]  13.102: [CMS-concurrent-mark-start]  13.341: [CMS-concurrent-mark: 0.238/0.238 secs] [Times: user=0.36 sys=0.01, real=0.24 secs]  13.341: [CMS-concurrent-preclean-start]  13.350: [CMS-concurrent-preclean: 0.009/0.009 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]  13.350: [CMS-concurrent-abortable-preclean-start]  13.878: [GC (Allocation Failure) 13.878: [ParNew: 306688K->34047K(306688K), 0.0960456 secs] 1204052K->1010638K(2063104K), 0.0961542 secs] [Times: user=0.29 sys=0.04, real=0.09 secs]  14.366: [CMS-concurrent-abortable-preclean: 0.917/1.016 secs] [Times: user=2.22 sys=0.07, real=1.01 secs]  14.366: [GC (CMS Final Remark) [YG occupancy: 182593 K (306688 K)]14.366: [Rescan (parallel) , 0.0291598secs]14.395: [weak refs processing, 0.0000232 secs]14.395: [class unloading, 0.0117661 secs]14.407: [scrub symbol table, 0.0015323 secs]14.409: [scrub string table, 0.0003221 secs][1 CMS-remark: 976591K(1756416K)]1159184K(2063104K), 0.0462010 secs] [Times: user=0.14 sys=0.00, real=0.05 secs]  14.412: [CMS-concurrent-sweep-start]  14.633: [CMS-concurrent-sweep: 0.221/0.221 secs] [Times: user=0.37 sys=0.00, real=0.22 secs]  14.633: [CMS-concurrent-reset-start]  14.636: [CMS-concurrent-reset: 0.002/0.002 secs] [Times: user=0.00 sys=0.00, real=0.00 secs] |

从上面的日志我们可以看到经过12次 Minor GC后一些“不同的东西”发生了。 不是两次Full GC， 而是单一的在年老代（Tenured generation）的GC, 包括两个阶段:

* 初始标记Mark阶段, 大概花费0.0041705秒，约等于4毫秒. 这个阶段是stop-the-world事件，会暂停所有应用的线程以便标记.
* 并发执行Markup 和 Preclean阶段. 它和应用程序的线程并发执行
* 最终Remark阶段, 花费0.0462010秒，大约46毫秒. 这个阶段还是stop-the-world 事件.
* 并发执行Sweep操作. 就像名字一样，这个阶段并发执行，不会暂停应用的线程.

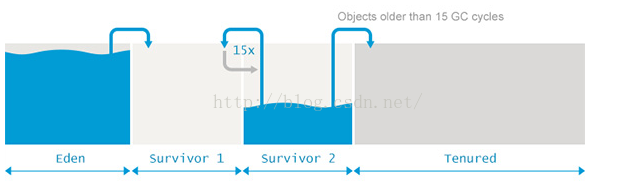
就像我们从gc log中看到的，不是两次Full GC操作，只有一次Major GC用来清理年老区。

如果你遇到延迟的问题，然后基于jstat的结果做出决定， 这没问题。它正确的列出了两次stop-the-world事件的总耗时：50毫秒，它会导致所有的应用线程的延迟。但是如果你想优化吞吐率，你可能被误导了– 它只列出了导致stop-the-world的初始mark和最终remark阶段，jstat输出结果完全隐藏了Major GC并发工作。

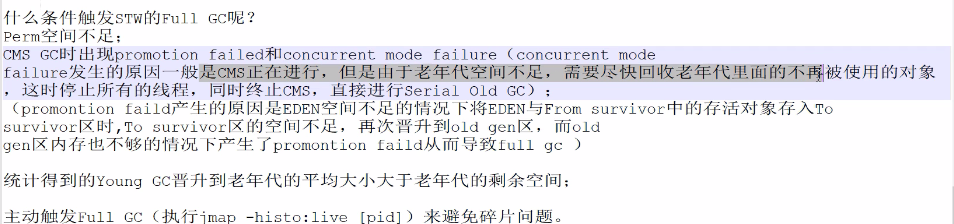
## 结论

考虑到上面的情况，最好不要考虑Minor,Major和Full GC的术语， 相反，监控你的程序的延迟和吞吐率，以及和GC事件的关联。检查这些事件是否强制暂停应用程序的线程，或者事件是并发的执行。

堆内存划分为 Eden、Survivor 和 Tenured/Old 空间，如下图所示：



从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC，对老年代GC称为Major GC,而Full GC是对整个堆来说的，在最近几个版本的JDK里默认包括了对永生带即方法区的回收（JDK8中无永生带了），出现Full GC的时候经常伴随至少一次的Minor GC,但非绝对的。Major GC的速度一般会比Minor GC慢10倍以上。下边看看有那种情况触发JVM进行Full GC及应对策略。



**1、System.gc()方法的调用**

此方法的调用是建议JVM进行Full GC,虽然只是建议而非一定,但很多情况下它会触发 Full GC,从而增加Full GC的频率,也即增加了间歇性停顿的次数。强烈影响系建议能不使用此方法就别使用，让虚拟机自己去管理它的内存，可通过通过-XX:+ DisableExplicitGC来禁止RMI调用System.gc。

## 2、老年代代空间不足

老年代空间只有在新生代对象转入及创建为大对象、大数组时才会出现不足的现象，当执行Full GC后空间仍然不足，则抛出如下错误：  
java.lang.OutOfMemoryError: Java heap space   
为避免以上两种状况引起的Full GC，调优时应尽量做到让对象在Minor GC阶段被回收、让对象在新生代多存活一段时间及不要创建过大的对象及数组。

## 3、永生区空间不足

JVM规范中运行时数据区域中的方法区，在HotSpot虚拟机中又被习惯称为永生代或者永生区，Permanet Generation中存放的为一些class的信息、常量、静态变量等数据，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation可能会被占满，在未配置为采用CMS GC的情况下也会执行Full GC。如果经过Full GC仍然回收不了，那么JVM会抛出如下错误信息：  
java.lang.OutOfMemoryError: PermGen space   
为避免Perm Gen占满造成Full GC现象，可采用的方法为增大Perm Gen空间或转为使用CMS GC。

## 4、CMS GC时出现promotion failed和concurrent mode failure

对于采用CMS进行老年代GC的程序而言，尤其要注意GC日志中是否有promotion failed和concurrent mode failure两种状况，当这两种状况出现时可能会触发Full GC。promotion failed是在进行Minor GC时，survivor space放不下、对象只能放入老年代，而此时老年代也放不下造成的；concurrent mode failure是在执行CMS GC的过程中同时有对象要放入老年代，而此时老年代空间不足造成的（有时候“空间不足”是CMS GC时当前的浮动垃圾过多导致暂时性的空间不足触发Full GC）。  
对措施为：增大survivor space、老年代空间或调低触发并发GC的比率，但在JDK 5.0+、6.0+的版本中有可能会由于JDK的bug29导致CMS在remark完毕后很久才触发sweeping动作。对于这种状况，可通过设置-XX: CMSMaxAbortablePrecleanTime=5（单位为ms）来避免。

## 5、统计得到的Minor GC晋升到旧生代的平均大小大于老年代的剩余空间

这是一个较为复杂的触发情况，Hotspot为了避免由于新生代对象晋升到旧生代导致旧生代空间不足的现象，在进行Minor GC时，做了一个判断，如果之前统计所得到的Minor GC晋升到旧生代的平均大小大于旧生代的剩余空间，那么就直接触发Full GC。  
例如程序第一次触发Minor GC后，有6MB的对象晋升到旧生代，那么当下一次Minor GC发生时，首先检查旧生代的剩余空间是否大于6MB，如果小于6MB，则执行Full GC。  
当新生代采用PS GC时，方式稍有不同，PS GC是在Minor GC后也会检查，例如上面的例子中第一次Minor GC后，PS GC会检查此时旧生代的剩余空间是否大于6MB，如小于，则触发对旧生代的回收。  
除了以上4种状况外，对于使用RMI来进行RPC或管理的Sun JDK应用而言，默认情况下会一小时执行一次Full GC。可通过在启动时通过- java -

Dsun.rmi.dgc.client.gcInterval=3600000来设置Full GC执行的间隔时间或通过-XX:+ DisableExplicitGC来禁止RMI调用System.gc。

## 6、堆中分配很大的对象

所谓大对象，是指需要大量连续内存空间的java对象，例如很长的数组，此种对象会直接进入老年代，而老年代虽然有很大的剩余空间，但是无法找到足够大的连续空间来分配给当前对象，此种情况就会触发JVM进行Full GC。

为了解决这个问题，CMS垃圾收集器提供了一个可配置的参数，即-XX:+UseCMSCompactAtFullCollection开关参数，用于在“享受”完Full GC服务之后额外免费赠送一个碎片整理的过程，内存整理的过程无法并发的，空间碎片问题没有了，但提顿时间不得不变长了，JVM设计者们还提供了另外一个参数 -XX:CMSFullGCsBeforeCompaction,这个参数用于设置在执行多少次不压缩的Full GC后,跟着来一次带压缩的。

## Ergonomics

## 垃圾收集器

在《[Java内存区域 JVM运行时数据区](http://blog.csdn.net/tjiyu/article/details/53915869)》中了解到各数据区有些什么特点、以及相关参数的调整，知道了：**程序计数器、虚拟机栈、本地方法栈这3个区域是随线程而生而灭的，内存分配和回收都具备确定性，而Java堆和方法区则不一样，各线程共享，在运行时内存的分配与回收都是动态的，垃圾收集器所关注的是这部分内存**。

       接下来几篇文章我们来详细了解Java堆和方法区的内存分配与回收，下面先来了解Java虚拟机垃圾回收的基础内容：如何判断对象是存活还是已经死亡？ 介绍相关的垃圾回收基础算法：引用计数算法、可达性分析算法，以及说明finalize()方法作用，最后再来说说HotSpot虚拟机中实现对象可达性分析的一些问题。

**1、Java虚拟机垃圾回收**

       垃圾回收，或称垃圾收集（Garbage Collection，GC）是指自动管理回收不再被引用的内存数据。

       在1960年诞生于MIT的Lisp语言首次使用了动态内存分配和垃圾收集技术，可以实现垃圾回收的一个基本要求是语言是类型安全的，现在使用的包括Java、Perl、ML等。

**1-1、为什么需要了解垃圾回收**

       目前内存的动态分配与内存回收技术已经相当成熟，但为什么还需要去了解内存分配与GC呢？

       1、当需要排查各种内存溢出、内存泄漏问题时；

       2、当垃圾收集成为系统达到更高并发量的瓶颈时；

       我们就需要对这些"自动化"技术实话必要的监控和调节；

**1-2、垃圾回收需要了解什么**

       思考GC完成的3件事：

**1、哪些内存需要回收？即如何判断对象已经死亡；**

**2、什么时候回收？即GC发生在什么时候？需要了解GC策略，与垃圾回收器实现有关；**

**3、如何回收？即需要了解垃圾回收算法，及算法的实现--垃圾回收器；**

      第一点就是本文下面的主题，这是垃圾回收的基础，如：可达性分析算法是后面垃圾回收算法的基础，而判断哪些对象可以回收是垃圾回收的首要任务。

**2、判断对象可以回收**

       垃圾收集器对堆进行回收前，首先要确定堆中的对象哪些还"存活"，哪些已经"死去"；

       下面先来了解两种判断对象不再被引用的算法，再来谈谈对象的引用，最后来看如何真正宣告一个对象死亡。

**2-1、引用计数算法（Recference Counting）**

**1、算法基本思路**

       给对象添加一个引用计数器，每当有一个地方引用它，计数器加1；

       当引用失效，计数器值减1；

       任何时刻计数器值为0，则认为对象是不再被使用的；

**2、优点**

      实现简单，判定高效，可以很好解决大部分场景的问题，也有一些著名的应用案例；

**3、缺点**

（A）、很难解决对象之间相互循环引用的问题

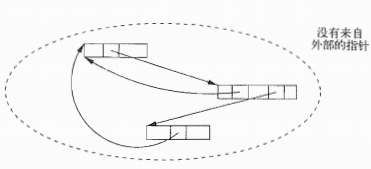
       如：

**[java]** [view plain](https://blog.csdn.net/tjiyu/article/details/53982412) [copy](https://blog.csdn.net/tjiyu/article/details/53982412)

1. ReferenceCountingGC objA = **new** ReferenceCountingGC();
2. ReferenceCountingGC objB = **new** ReferenceCountingGC();
3. objA.instance = objB;
4. objB.instance = objA;
5. objA = **null**;
6. objB = **null**;

       当两个对象不再被访问时，因为相互引用对方，导致引用计数不为0；

       更复杂的循环数据结构，如图（《编译原理》7-18）：



（B）、并且开销较大，频繁且大量的引用变化，带来大量的额外运算；

      主流的JVM都没有选用引用计数算法来管理内存；

**2-2、可达性分析算法（Reachability Analysis）**

      也称为传递跟踪算法；

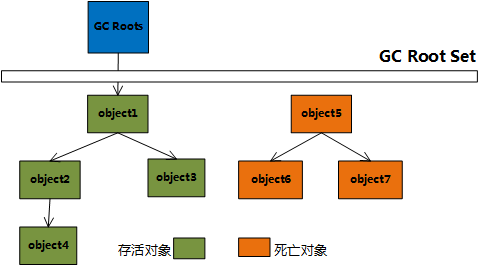
**主流的调用程序语言（Java、C#等）在主流的实现中，都是通过可达性分析来判定对象是否存活的。**

**1、算法基本思路**

      通过一系列"GC Roots"对象作为起始点，开始向下搜索；

      搜索所走过和路径称为引用链（Reference Chain）；

      当一个对象到GC Roots没有任何引用链相连时（从GC Roots到这个对象不可达），则证明该对象是不可用的；



**2、GC Roots对象**

      Java中，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JNI（Native方法）引用的对象；

      主要在执行上下文中和全局性的引用；

**3、优点**

      更加精确和严谨，可以分析出循环数据结构相互引用的情况；

**4、缺点**

      实现比较复杂；

      需要分析大量数据，消耗大量时间；

      分析过程需要GC停顿（引用关系不能发生变化），即**停顿所有Java执行线程**（称为**"Stop The World"，是垃圾回收重点关注的问题**）；

后面会针对HotSpot虚拟机实现的可达性分析算法进行介绍，看看是它**如何解决这些缺点的**。

**2-3、再谈对象引用**

      在《[Java对象在Java虚拟机中的引用访问方式](http://blog.csdn.net/tjiyu/article/details/53932199)》曾详细介绍过对象的引用问题，这与对象回收算法有很大关系，下面再来了解下。

      java程序通过reference类型数据操作堆上的具体对象；

**1、JVM层面的引用**

      reference类型是引用类型（Reference Types）的一种；

      JVM规范规定reference类型来表示对某个对象的引用，可以想象成类似于一个指向对象的指针；

      对象的操作、传递和检查都通过引用它的reference类型的数据进行操作；

**2、Java语言层面的引用**

**（i）、JDK1.2前的引用定义**

      如果reference类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用；

      这种定义太过狭隘，无法描述更多信息；

**（ii）、JDK1.2后，对引用概念进行了扩充，将引用分为：**

（1）、强引用（Strong Reference）

      程序代码普遍存在的，类似"Object obj=new Object()"；

      只要强引用还存在，GC永远不会回收被引用的对象；

（2）、软引用（Soft Reference）

      用来描述还有用但并非必需的对象；

      直到内存空间不够时（抛出OutOfMemoryError之前），才会被垃圾回收；

      最常用于实现对内存敏感的缓存；

      SoftReference类实现；

（3）、弱引用（Weak Reference）

      用来描述非必需对象；

      只能生存到下一次垃圾回收之前，无论内存是否足够；

       WeakReference类实现；

（4）、虚引用（Phantom Reference）

      也称为幽灵引用或幻影引用；

      完全不会对其生存时间构成影响；

      唯一目的就是能在这个对象被回收时收到一个系统通知；

      PhantomRenference类实现；

      更多请参考JDK相关API说明；

      对于软引用，可以使用命令行选项"-XX：SoftRefLRUPolicyMSPerMB = <N>"来控制清除速率；

      更多请参考：[http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/considerations.html#sthref65](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/considerations.html%23sthref65)

**2-4、判断对象生存还是死亡**

      要真正宣告一个对象死亡，至少要经历两次标记过程。

**1、第一次标记**

      在可达性分析后发现到GC Roots没有任何引用链相连时，被第一次标记；

      并且进行一次筛选：此对象是否必要执行finalize()方法；

**（A）、没有必要执行**

      没有必要执行的情况：

      （1）、对象没有覆盖finalize()方法；

      （2）、finalize()方法已经被JVM调用过；

      这两种情况就**可以认为对象已死**，可以回收；

**（B）、有必要执行**

      对有必要执行finalize()方法的对象，被放入F-Queue队列中；

      稍后在JVM自动建立、低优先级的Finalizer线程（可能多个线程）中触发这个方法；

**2、第二次标记**

      GC将对F-Queue队列中的对象进行第二次小规模标记；

      finalize()方法是对象逃脱死亡的最后一次机会：

      （A）、如果对象在其finalize()方法中重新与引用链上任何一个对象建立关联，第二次标记时会将其移出"即将回收"的集合；

      （B）、如果对象没有，也可以认为对象已死，可以回收了；

      一个对象的finalize()方法只会被系统自动调用一次，经过finalize()方法逃脱死亡的对象，第二次不会再调用；

**2-5、finalize()方法**

      上面已经说到finalize()方法与垃圾回收第二次标记相关，下面了解下**在Java语言层面有哪些需要注意的**。

      finalize()是Object类的一个方法，是Java刚诞生时为了使C/C++程序员容易接受它所做出的一个妥协，但不要当作类似C/C++的析构函数；

      因为它执行的时间不确定，甚至是否被执行也不确定（Java程序的不正常退出），而且运行代价高昂，无法保证各个对象的调用顺序（甚至有不同线程中调用）；

      如果需要"释放资源"，可以定义显式的终止方法，并在"try-catch-finally"的finally{}块中保证及时调用，如File相关类的close()方法；

      此外，finalize()方法主要有两种用途：

**1、充当"安全网"**

      当显式的终止方法没有调用时，在finalize()方法中发现后发出警告；

      但要考虑是否值得付出这样的代价；

      如FileInputStream、FileOutputStream、Timer和Connection类中都有这种应用；

**2、与对象的本地对等体有关**

      本地对等体：普通对象调用本地方法（JNI）委托的本地对象；

      本地对等体不会被GC回收；

      如果本地对等体不拥有关键资源，finalize()方法里可以回收它（如C/C++中malloc()，需要调用free()）；

      如果有关键资源，必须显式的终止方法；

**一般情况下，应尽量避免使用它，甚至可以忘掉它。**

更多请参考：

      《How to Handle Java Finalization's Memory-Retention Issues》：<http://www.devx.com/Java/Article/30192>

      《Effective Java》第二版 第2章 第7条：避免使用终结方法；

      《Thinking in Java》第四版 5.5 清理：终结处理和垃圾回收；

      《Java语言规范》12.6 类实例的终结；

**3、HotSpot虚拟机中对象可达性分析的实现**

      前面对可达性分析算法进行介绍，并看到了它在判断对象存活与死亡的作用，下面看看是HotSpot虚拟机是如何实现可达性分析算法，**如何解决相关缺点的。**

**3-1、可达性分析的问题**

**1、消耗大量时间**

      从前面可达性分析知道，GC Roots主要在全局性的引用（常量或静态属性）和执行上下文中（栈帧中的本地变量表）；

      要在这些大量的数据中，逐个检查引用，会消耗很多时间；

**2、GC停顿**

      可达性分析期间需要保证整个执行系统的一致性，对象的引用关系不能发生变化；

**导致GC进行时必须停顿所有Java执行线程**（称为**"Stop The World"**）；

      （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）

**Stop The World：**

      是JVM在后台自动发起和自动完成的；

      在用户不可见的情况下，把用户正常的工作线程全部停掉；

**3-2、枚举根节点**

      枚举根节点也就是查找GC Roots；

      目前主流JVM都是准确式GC，可以直接得知哪些地方存放着对象引用，所以执行系统停顿下来后，并不需要全部、逐个检查完全局性的和执行上下文中的引用位置；

      在HotSpot中，是使用一组称为OopMap的数据结构来达到这个目的的；

      在类加载时，计算对象内什么偏移量上是什么类型的数据；

      在JIT编译时，也会记录栈和寄存器中的哪些位置是引用；

      这样GC扫描时就可以直接得知这些信息；

**3-3、安全点**

**1、安全点是什么，为什么需要安全点**

      HotSpot在OopMap的帮助下可以快速且准确的完成GC Roots枚举，但是这有一个问题：

      运行中，非常多的指令都会导致引用关系变化；

      如果为这些指令都生成对应的OopMap，需要的空间成本太高；

      问题解决：

**只在特定的位置记录OopMap引用关系**，这些位置称为**安全点（Safepoint）**；

      即程序执行时并非所有地方都能停顿下来开始GC；

**2、安全点的选定**

      不能太少，否则GC等待时间太长；也不能太多，否则GC过于频繁，增大运行时负荷；

      所以，基本上是以程序"是否具有让程序长时间执行的特征"为标准选定；

      "长时间执行"最明显的特征就是指令序列复用，如：方法调用、循环跳转、循环的末尾、异常跳转等；

      只有具有这些功能的指令才会产生Safepoint；

**3、如何在安全点上停顿**

      对于Safepoint，如何在GC发生时让所有线程（不包括JNI线程）运行到其所在最近的Safepoint上再停顿下来？

      主要有两种方案可选：

**（A）、抢先式中断（Preemptive Suspension）**

      不需要线程主动配合，实现如下：

      （1）、在GC发生时，首先中断所有线程；

      （2）、如果发现不在Safepoint上的线程，就恢复让其运行到Safepoint上；

**现在几乎没有JVM实现采用这种方式；**

**（B）、主动式中断（Voluntary Suspension）**

      （1）、在GC发生时，不直接操作线程中断，而是仅简单设置一个标志；

      （2）、让各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；

      而轮询标志的地方和Safepoint是重合的；

      在JIT执行方式下：test指令是HotSpot生成的轮询指令；

      一条test汇编指令便完成Safepoint轮询和触发线程中断；

**3-4、安全区域**

**1、为什么需要安全区域**

      对于上面的Safepoint还有一个问题：

      程序不执行时没有CPU时间（Sleep或Blocked状态），无法运行到Safepoint上再中断挂起；

      这就需要安全区域来解决；

**2、什么是安全区域（Safe Region）**

      指一段代码片段中，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；

**3、如何用安全区域解决问题**

      安全区域解决问题的思路：

      （1）、线程执行进入Safe Region，首先标识自己已经进入Safe Region；

      （2）、线程被唤醒离开Safe Region时，其需要检查系统是否已经完成根节点枚举（或整个GC）；

      如果已经完成，就继续执行；

      否则必须等待，直到收到可以安全离开Safe Region的信号通知；

这样就不会影响标记结果；

      虽然HotSpot虚拟机中采用了这些方法来解决对象可达性分析的问题，但只是大大减少了这些问题影响，**并不能完全解决，如GC停顿"Stop The World"是垃圾回收重点关注的问题，后面介绍垃圾回收器时应注意：低GC停顿是其一个关注。**

**到这里，我们大体了解Java虚拟机垃圾回收的一些基础内容，后面我们将分别去了解JVM垃圾收集算法、垃圾收集器、以及相关调优方法……**

在[《Java虚拟机垃圾回收(一) 基础》](http://blog.csdn.net/tjiyu/article/details/53982412)中了解到如何判断对象是存活还是已经死亡？ 介绍了垃圾回收基础算法：引用计数算法、可达性分析算法，以及HotSpot虚拟机中实现对象可达性分析的一些问题。

       下面先来了解Java虚拟机垃圾回收的几种常见算法：标记-清除算法、复制算法、标记-整理算法、分代收集算法、火车算法，介绍它们的算法思路，有什么优点和缺点，以及主要应用场景。

**1、标记-清除算法**

       标记-清除（Mark-Sweep）算法是一种基础的收集算法。

**1、算法思路**

       "标记-清除"算法，分为两个阶段：

**（A）、标记**

**首先标记出所有需要回收的对象；**

       标记过程如[《Java虚拟机垃圾回收(一) 基础》](http://blog.csdn.net/tjiyu/article/details/53982412)"2-4、判断对象生存还是死亡"中所述--分为两个标记过程（详细请参考前文）：

**（1）、第一次标记**

       在可达性分析后发现对象到GC Roots没有任何引用链相连时，被第一次标记；

       并且进行一次筛选：此对象是否必要执行finalize()方法；

       对有必要执行finalize()方法的对象，被放入F-Queue队列中；

**（2）、第二次标记**

       GC将对F-Queue队列中的对象进行第二次小规模标记；

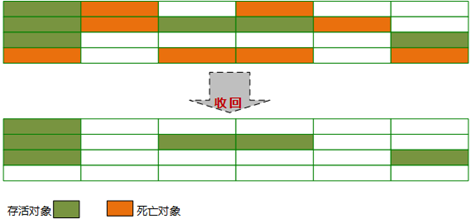
       在其finalize()方法中重新与引用链上任何一个对象建立关联，第二次标记时会将其移出"即将回收"的集合；

**对第一次被标记，且第二次还被标记（如果需要，但没有移出"即将回收"的集合），就可以认为对象已死，可以进行回收**。

**（B）、清除**

      两次标记后，还在"即将回收"集合的对象将被统一回收；

**执行过程如下图：**



**2、优点**

       基于最基础的可达性分析算法，它是最基础的收集算法；

       而后续的收集算法都是基于这种思路并对其不足进行改进得到的；

**3、缺点**

       主要有两个缺点：

（A）、效率问题

       标记和清除两个过程的效率都不高；

（B）、空间问题

       标记清除后会产生大量不连续的内存碎片；

       这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垃圾收集动作；

**4、应用场景**

**针对老年代的CMS收集器；**

**2、复制算法算法**

       "复制"（Copying）收集算法，为了解决标记-清除算法的效率问题；

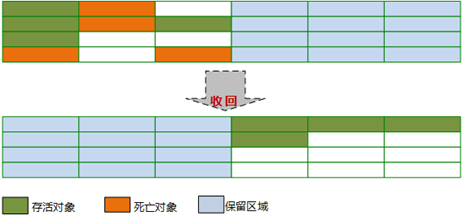
**1、算法思路**

       （A）、把内存划分为大小相等的两块，每次只使用其中一块；

       （B）、当一块内存用完了，就将还存活的对象复制到另一块上（而后使用这一块）；

       （C）、再把已使用过的那块内存空间一次清理掉，而后重复步骤2；

**执行过程如下图：**



**2、优点**

       这使得每次都是只对整个半区进行内存回收；

       内存分配时也不用考虑内存碎片等问题（可使用"指针碰撞"的方式分配内存）；

      实现简单，运行高效；

       （关于"指针碰撞"请参考《[Java对象在HotSpot虚拟机中的创建过程](http://blog.csdn.net/tjiyu/article/details/53923392)》）

**3、缺点**

（A）、空间浪费

      可用内存缩减为原来的一半，太过浪费（解决：可以改良，不按1:1比例划分）；

（B）、效率随对象存活率升高而变低

      当对象存活率较高时，需要进行较多复制操作，效率将会变低（解决：后面的标记-整理算法）；

**4、应用场景**

      现在商业JVM都采用这种算法（通过改良缺点1）来回收新生代；

**如Serial收集器、ParNew收集器、Parallel Scavenge收集器、、G1（从局部看）；**

**5、HotSpot虚拟机的改良算法**

**（A）、弱代理论**

       分代垃圾收集基于弱代理论（weak generational hypothesis），具体描述如下：

       （1）、大多数分配了内存的对象并不会存活太长时间，在处于年轻代时就会死掉；

       （2）、很少有对象会从老年代变成年轻代；

       其中IBM研究表明：新生代中98%的对象都是"朝生夕死"；

        所以并不需要按1:1比例来划分内存（解决了缺点1）；

**（B）、HotSpot虚拟机新生代内存布局及算法**

                      （1）、将新生代内存分为一块较大的Eden空间和两块较小的Survivor空间；

                      （2）、每次使用Eden和其中一块Survivor；

                      （3）、当回收时，将Eden和使用中的Survivor中还存活的对象一次性复制到另外一块Survivor；

                      （4）、而后清理掉Eden和使用过的Survivor空间；

                      （5）、后面就使用Eden和复制到的那一块Survivor空间，重复步骤3；

         默认Eden：Survivor=8:1，即每次可以使用90%的空间，只有一块Survivor的空间被浪费；

**（C）、分配担保**

       如果另一块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（Handle Promotion）进入老年代；

       分配担保在以后讲解垃圾收集器执行规则时再详解；

       更多请参考：[http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/generations.html#sthref16](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/generations.html%23sthref16)

**3、标记-整理算法**

       "标记-整理"（Mark-Compact）算法是根据老年代的特点提出的。

**1、算法思路**

（1）、标记

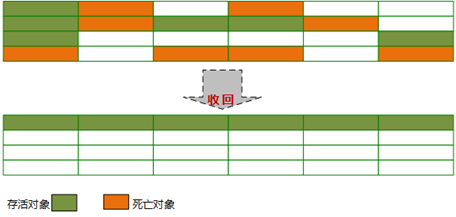
      标记过程与"标记-清除"算法一样；

（2）、整理

       但后续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；

**执行过程如下图：**



**2、优点**

**（A）、不会像复制算法，效率随对象存活率升高而变低**

       老年代特点：

       对象存活率高，没有额外的空间可以分配担保；

       所以老年代一般不能直接选用复制算法算法；

       而选用标记-整理算法；

**（B）、不会像标记-清除算法，产生内存碎片**

       因为清除前，进行了整理，存活对象都集中到空间一侧；

**3、缺点**

       主要是效率问题：除像标记-清除算法的标记过程外，还多了需要整理的过程，效率更低；

**4、应用场景**

       很多垃圾收集器采用这种算法来回收老年代；

**如Serial Old收集器、G1（从整体看）；**

**4、分代收集算法**

       "分代收集"（Generational Collection）算法结合不同的收集算法处理不同区域。

**1、算法思路**

       基于前面说的弱代理论，其实并没有什么新的思想；

       只是根据对象存活周期的不同将内存划分为几块；

       这样就可以根据各个年代的特点采用最适当的收集算法；

       一般把Java堆分为新生代和老年代；

**（A）、新生代**

       每次垃圾收集都有大批对象死去，只有少量存活；

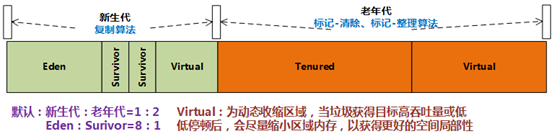
       所以可采用复制算法；

**（B）、老年代**

       对象存活率高，没有额外的空间可以分配担保；

      使用"标记-清理"或"标记-整理"算法；

      结合上面对新生代的内存划分介绍和上篇文章对Java堆的介绍，可以得出HotSpot虚拟机一般的年代内存划分，如下图：



**2、优点**

       可以根据各个年代的特点采用最适当的收集算法；

**3、缺点**

       仍然不能控制每次垃圾收集的时间；

**4、应用场景**

      目前几乎所有商业虚拟机的垃圾收集器都采用分代收集算法；

      如HotSpot虚拟机中全部垃圾收集器：Serial、ParNew、Parallel Scavenge、Serial Old、Parallel Old、CMS、G1（也保留）；

**5、火车算法**

      火车算法也称列车算法，是一种更彻底的分区域处理收集算法，是对分代收集算法的一个有力补充。

**1、算法思路**

      在火车算法中，内存被分为块，多个块组成一个集合。为了形象化，一节车厢代表一个块，一列火车代表一个集合，如下图；

      火车与车箱都按创建顺序标号，每个车厢大小相等，但每个火车包含的车厢数不一定相等；

      每节车箱有一个被记忆集合，而每辆火车的记忆集合是它所有车厢记忆集合的总和；

      记忆集合由指向车箱中对象的引用组成，这些引用来自同一辆火车中序号较高的车箱中的对象，以及序号较高中的对象；

      垃圾收集以车厢为单位，整体算法流程如下：

（1）、选择标号最小的火车；

（2）、如果火车的记忆集合是空的, 释放整列火车并终止, 否则进行第三步操作；

（3）、选择火车中标号最小的车厢；

（4）、对于车厢记忆集合的每个元素：

      如果它是一个被根引用引用的对象, 那么, 将拷贝到一列新的火车中去；

      如果是一个被其它火车的对象指向的对象, 那么, 将它拷贝到这个指向它的火车中去.；

      假设有一些对象已经被保留下来了, 那么通过这些对象可以触及到的对象将会被拷贝到同一列火车中去；

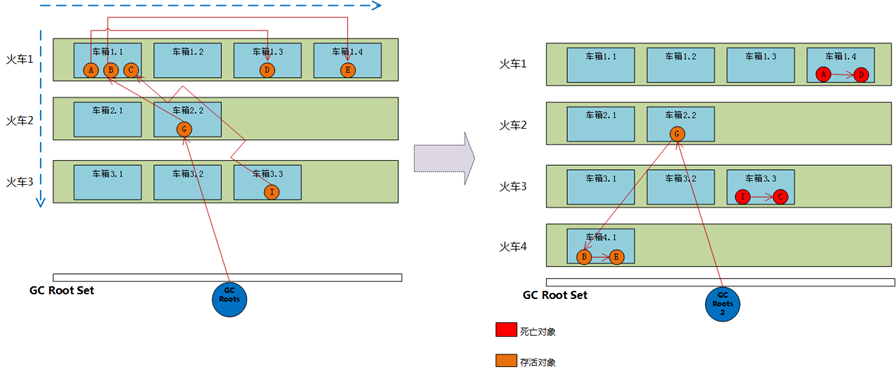
      如果一个对象被来自多个火车的对象引用, 那么它可以被拷贝到任意一个火车去；

      这个步骤中, 有必要对受影响的引用集合进行相应地更新；

（5）、释放车厢并且终止；

      收集过程会删除一些空车箱和空车，当需要的时候也会创建一些车箱和火车，更多信息请参考：《编译原理》第二版7.75"列车算法"、[《渐进式地垃圾回收: 火车算法》](http://nileader.blog.51cto.com/1381108/402609)；

**执行过程如下图：**



**2、优点**

      可以在成熟对象空间提供限定时间的渐近收集；

      而不需要每次都进行一个大区域的垃圾回收过程；

**即可以控制垃圾回收的时间，在指定时间内进行一些小区域的回收；**

**3、缺点**

      实现较为复杂，如采用类似的算法的G1收集器在JDK7才实现；

      一些场景下可能性价比不高；

**4、应用场景**

      JDK7后HotSpot虚拟机**G1收集器**采用**类似的算法**，能建立可预测的停顿时间模型；

**到这里，我们大体了解Java虚拟机垃圾回收的几种常见算法，后面我们将分别去了解JVM垃圾收集器、以及相关调优方法……**

在[《Java虚拟机垃圾回收(一) 基础》](http://blog.csdn.net/tjiyu/article/details/53982412)中了解到如何判断对象是存活还是已经死亡？在[《Java虚拟机垃圾回收(二) 垃圾回收算法》](http://blog.csdn.net/tjiyu/article/details/53983064)了解到Java虚拟机垃圾回收的几种常见算法。

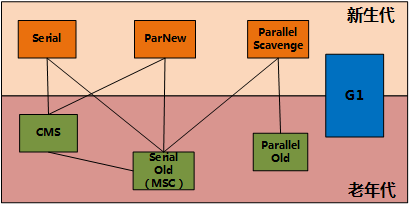
       下面先来了解HotSpot虚拟机中的7种垃圾收集器：Serial、ParNew、Parallel Scavenge、Serial Old、Parallel Old、CMS、G1，先介绍一些垃圾收集的相关概念，再介绍它们的主要特点、应用场景、以及一些设置参数和基本运行原理。

**1、垃圾收集器概述**

       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、火车算法）的具体实现，不同商家、不同版本的JVM所提供的垃圾收集器可能会有很在差别，本文主要介绍HotSpot虚拟机中的垃圾收集器。

**1-1、垃圾收集器组合**

       JDK7/8后，HotSpot虚拟机所有收集器及组合（连线），如下图：



（A）、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；

（B）、而它们所处区域，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；

（C）、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；

（D）、其中Serial Old作为CMS出现**"Concurrent Mode Failure"失败**的后备预案（后面介绍）；

**1-2、并发垃圾收集和并行垃圾收集的区别**

**（A）、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；

       如**ParNew、Parallel Scavenge、Parallel Old**；

**（B）、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，而垃圾收集程序线程运行于另一个CPU上；

       如**CMS、G1**（也有并行）；

**1-3、Minor GC和Full GC的区别**

**（A）、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；

**（B）、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；

下面将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；但需要明确一个观点：

       没有最好的收集器，更没有万能的收集；

      选择的只能是适合具体应用场景的收集器。

**2、Serial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；

**1、特点**

      针对新生代；

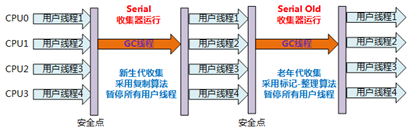
      采用复制算法；

      单线程收集；

       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：**



**2、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）,只要不频繁发生，这是可以接受的

**3、设置参数**

**"-XX:+UseSerialGC"**：添加该参数来显式的使用串行垃圾收集器；

**4、Stop TheWorld说明**

      JVM在后台自动发起和自动完成的，在用户不可见的情况下，把用户正常的工作线程全部停掉，即**GC停顿**；

      会带给用户不良的体验；

      从JDK1.3到现在，从Serial收集器-》Parallel收集器-》CMS-》G1，用户线程停顿时间不断缩短，但仍然无法完全消除；

      更多"Stop The World"信息请参考：[《Java虚拟机垃圾回收(一) 基础》](http://blog.csdn.net/tjiyu/article/details/53982412)"2-2、可达性分析算法"

更多Serial收集器请参考：

      《Memory Management in the Java HotSpot™ Virtual Machine》 4.3节 Serial Collector（内存管理白皮书）：<http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf>

      《Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide》 第5节 Available Collectors（官方的垃圾收集调优指南）：<http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html#sthref27>

**3、ParNew收集器**

      ParNew垃圾收集器是**Serial收集器的多线程版本**。

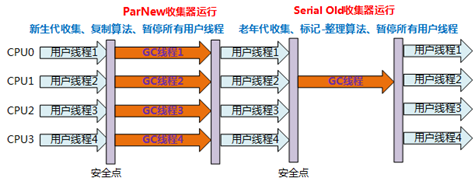
**1、特点**

      除了多线程外，其余的行为、特点和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；

      两个收集器共用了不少代码；

**ParNew/Serial Old组合收集器运行示意图如下：**



**2、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。

**3、设置参数**

**"-XX:+UseConcMarkSweepGC"**：指定使用CMS后，会默认使用ParNew作为新生代收集器；

**"-XX:+UseParNewGC"**：强制指定使用ParNew；

**"-XX:ParallelGCThreads"**：指定垃圾收集的线程数量，ParNew默认开启的收集线程与CPU的数量相同；

**4、为什么只有ParNew能与CMS收集器配合**

      CMS是HotSpot在JDK1.5推出的第一款真正意义上的并发（Concurrent）收集器，第一次实现了让垃圾收集线程与用户线程（基本上）同时工作；

      CMS作为老年代收集器，但却无法与JDK1.4已经存在的新生代收集器Parallel Scavenge配合工作；

      因为Parallel Scavenge（以及G1）都没有使用传统的GC收集器代码框架，而另外独立实现；而其余几种收集器则共用了部分的框架代码；

      关于CMS收集器后面会详细介绍。

**4、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）**。

**1、特点**

**（A）、**有一些特点与**ParNew收集器相似**

      新生代收集器；

      采用复制算法；

      多线程收集；

**（B）、主要特点是：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**达一个可控制的吞吐量（Throughput）**；

**关于吞吐量与收集器关注点说明详见本节后面；**

**2、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；

      当应用程序运行在具有多个CPU上，对暂停时间没有特别高的要求时，即程序主要在**后台进行计算，而不需要与用户进行太多交互**；

      例如，那些**执行批量处理**、订单处理、工资支付、科学计算的应用程序；

**3、设置参数**

      Parallel Scavenge收集器提供两个参数用于精确控制吞吐量：

**（A）、"-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；

**（B）、"-XX:GCTimeRatio"**

      设置垃圾收集时间占总时间的比率，0<n<100的整数；

      GCTimeRatio相当于设置吞吐量大小；

      垃圾收集执行时间占应用程序执行时间的比例的计算方法是：

      1 / (1 + n)

      例如，选项-XX:GCTimeRatio=19，设置了垃圾收集时间占总时间的5%--1/(1+19)；

      默认值是1%--1/(1+99)，即n=99；

垃圾收集所花费的时间是年轻一代和老年代收集的总时间；

如果没有满足吞吐量目标，则增加代的内存大小以尽量增加用户程序运行的时间；

      此外，还有一个值得关注的参数：

**（C）、"-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:PretenureSizeThreshold）等；

      JVM会根据当前系统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最大的吞吐量，这种调节方式称为**GC自适应的调节策略（GC Ergonomiscs）**；

**这是一种值得推荐的方式**：

      (1)、只需设置好内存数据大小（如"-Xmx"设置最大堆）；

      (2)、然后使用"-XX:MaxGCPauseMillis"或"-XX:GCTimeRatio"给JVM设置一个优化目标；

      (3)、那些具体细节参数的调节就由JVM自适应完成；

      这也是Parallel Scavenge收集器与ParNew收集器一个重要区别；

      更多目标调优和GC自适应的调节策略说明请参考：

      《Memory Management in the Java HotSpot™ Virtual Machine》 5节 Ergonomics -- Automatic Selections and Behavior Tuning：<http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf>

      《Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide》 第2节 Ergonomics：[http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/ergonomics.html#ergonomics](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/ergonomics.html%23ergonomics)

**4、吞吐量与收集器关注点说明**

**（A）、吞吐量（Throughput）**

      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）；**

      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；

**（B）、垃圾收集器期望的目标（关注点）**

**（1）、停顿时间**

      停顿时间越短就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；

**（2）、吞吐量**

      高吞吐量则可以高效率地利用CPU时间，尽快完成运算的任务；

      主要适合在后台计算而不需要太多交互的任务；

**（3）、覆盖区（Footprint）**

      在达到前面两个目标的情况下，尽量减少堆的内存空间；

      可以获得更好的空间局部性；

更多Parallel Scavenge收集器的信息请参考：

      官方的垃圾收集调优指南 第6节：[http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html#parallel\_collector](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/parallel.html%23parallel_collector)

**上面介绍的都是新生代收集器，接下来开始介绍老年代收集器；**

**5、Serial Old收集器**

      Serial Old是**Serial收集器的老年代版本**；

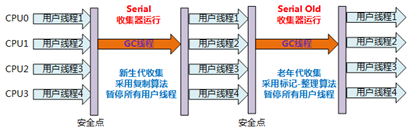
**1、特点**

      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：



**2、应用场景**

      主要用于Client模式；

      而在Server模式有两大用途：

      （A）、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）；

      （B）、作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用（后面详解）；

更多Serial Old收集器信息请参考：

      内存管理白皮书 4.3.2节：<http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf>

**6、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；

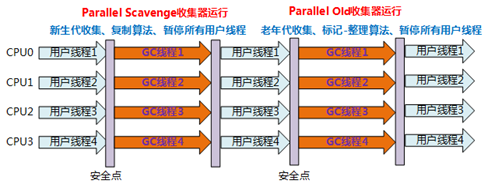
**1、特点**

      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：**



**2、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，就有了Parallel Scavenge加Parallel Old收集器的"给力"应用组合；

**3、设置参数**

**"-XX:+UseParallelOldGC"：指定使用Parallel Old收集器；**

更多Parallel Old收集器收集过程介绍请参考：

      《内存管理白皮书》 4.5.2节：    [http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf](http://blog.csdn.net/tjiyu/article/details/53983650)

**7、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-latency）垃圾收集器；

      在前面ParNew收集器曾简单介绍过其特点；

**1、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；

      并发收集、低停顿；

      需要更多的内存（看后面的缺点）；

      是HotSpot在JDK1.5推出的第一款真正意义上的并发（Concurrent）收集器；

      第一次实现了让垃圾收集线程与用户线程（基本上）同时工作；

**2、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**；

**3、设置参数**

**"-XX:+UseConcMarkSweepGC"：指定使用CMS收集器；**

**4、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:

**（A）、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"；

**（B）、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；

**（C）、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记稍长，但远比并发标记短；

      采用多线程并行执行来提升效率；

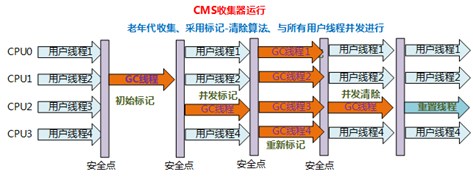
**（D）、并发清除（CMS concurrent sweep）**

      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行；

**CMS收集器运行示意图如下：**



**5、CMS收集器3个明显的缺点**

**（A）、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。

**增量式并发收集器：**

      针对这种情况，曾出现了"增量式并发收集器"（Incremental Concurrent Mark Sweep/i-CMS）；

      类似使用抢占式来模拟多任务机制的思想，让收集线程和用户线程交替运行，减少收集线程运行时间；

      但效果并不理想，**JDK1.6后就官方不再提倡用户使用**。

更多请参考：

      官方的《垃圾收集调优指南》8.8节 Incremental Mode：[http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html#CJAGIIEJ](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html%23CJAGIIEJ)

      《内存管理白皮书》 4.6.3节可以看到一些描述；

**（B）、无法处理浮动垃圾,可能出现"Concurrent Mode Failure"失败**

**（1）、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%；

**（2）、"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。

**（C）、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**；

      前面[《Java虚拟机垃圾回收(二) 垃圾回收算法》](http://blog.csdn.net/tjiyu/article/details/53983064)"标记-清除"算法介绍时曾说过：

      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；

      由于空间不再连续，CMS需要使用可用"空闲列表"内存分配方式，这比简单实用"碰撞指针"分配内存消耗大；

      更多关于内存分配方式请参考：《[Java对象在Java虚拟机中的创建过程](http://blog.csdn.net/tjiyu/article/details/53923392)》

**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；**

更多CMS收集器信息请参考：

      《垃圾收集调优指南》 8节 Concurrent Mark Sweep (CMS) Collector：[http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html#concurrent\_mark\_sweep\_cms\_collector](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html%23concurrent_mark_sweep_cms_collector)

      《内存管理白皮书》 4.6节 Concurrent Mark-Sweep (CMS) Collector：<http://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf>

**8、G1收集器**

      G1（**Garbage-First**）是**JDK7-u4**才推出商用的收集器；

**1、特点**

**（A）、并行与并发**

      能充分利用多CPU、多核环境下的硬件优势；

      可以并行来缩短"Stop The World"停顿时间；

      也可以并发让垃圾收集与用户程序同时进行；

**（B）、分代收集，收集范围包括新生代和老年代**

      能独立管理整个GC堆（新生代和老年代），而不需要与其他收集器搭配；

      能够采用不同方式处理不同时期的对象；

      虽然保留分代概念，但Java堆的内存布局有很大差别；

      将整个堆划分为多个大小相等的独立区域（Region）；

      新生代和老年代不再是物理隔离，它们都是一部分Region（不需要连续）的集合；

      更多G1内存布局信息请参考：

      《垃圾收集调优指南》 9节：[http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1\_gc.html#garbage\_first\_garbage\_collection](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc.html%23garbage_first_garbage_collection)

**（C）、结合多种垃圾收集算法，空间整合，不产生碎片**

      从整体看，是基于标记-整理算法；

      从局部（两个Region间）看，是基于复制算法；

      这是一种类似火车算法的实现；

      都不会产生内存碎片，有利于长时间运行；

**（D）、可预测的停顿：低停顿的同时实现高吞吐量**

      G1除了追求低停顿处，还能建立可预测的停顿时间模型；

      可以明确指定M毫秒时间片内，垃圾收集消耗的时间不超过N毫秒；

**2、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；

      如：在堆大小约6GB或更大时，可预测的暂停时间可以低于0.5秒；

      用来替换掉JDK1.5中的CMS收集器；

**在下面的情况时，使用G1可能比CMS好**：

      （1）、超过50％的Java堆被活动数据占用；

      （2）、对象分配频率或年代提升频率变化很大；

      （3）、GC停顿时间过长（长于0.5至1秒）。

**是否一定采用G1呢？也未必：**

      如果现在采用的收集器没有出现问题，不用急着去选择G1；

      如果应用程序追求低停顿，可以尝试选择G1；

      是否代替CMS需要实际场景测试才知道。

**3、设置参数**

**"-XX:+UseG1GC"：指定使用G1收集器；**

**"-XX:InitiatingHeapOccupancyPercent"：当整个Java堆的占用率达到参数值时，开始并发标记阶段；默认为45；**

**"-XX:MaxGCPauseMillis"：为G1设置暂停时间目标，默认值为200毫秒；**

**"-XX:G1HeapRegionSize"：设置每个Region大小，范围1MB到32MB；目标是在最小Java堆时可以拥有约2048个Region；**

      更多关于G1参数设置请参考：

      《垃圾收集调优指南》 10.5节：[http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1\_gc\_tuning.html#important\_defaults](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc_tuning.html%23important_defaults)

**4、为什么G1收集器可以实现可预测的停顿**

      G1可以建立可预测的停顿时间模型，是因为：

      可以有计划地避免在Java堆的进行全区域的垃圾收集；

      G1跟踪各个Region获得其收集价值大小，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（名称Garbage-First的由来）；

      这就保证了在有限的时间内可以获取尽可能高的收集效率；

**5、一个对象被不同区域引用的问题**

      一个Region不可能是孤立的，一个Region中的对象可能被其他任意Region中对象引用，判断对象存活时，是否需要扫描整个Java堆才能保证准确？

      在其他的分代收集器，也存在这样的问题（而G1更突出）：

      回收新生代也不得不同时扫描老年代？

      这样的话会降低Minor GC的效率；

**解决方法：**

      无论G1还是其他分代收集器，JVM都是使用Remembered Set来避免全局扫描：

      每个Region都有一个对应的Remembered Set；

      每次Reference类型数据写操作时，都会产生一个Write Barrier暂时中断操作；

      然后检查将要写入的引用指向的对象是否和该Reference类型数据在不同的Region（其他收集器：检查老年代对象是否引用了新生代对象）；

      如果不同，通过CardTable把相关引用信息记录到引用指向对象的所在Region对应的Remembered Set中；

      当进行垃圾收集时，在GC根节点的枚举范围加入Remembered Set；

      就可以保证不进行全局扫描，也不会有遗漏。

**6、G1收集器运作过程**

      不计算维护Remembered Set的操作，可以分为4个步骤（与CMS较为相似）。

**（A）、初始标记（Initial Marking）**

      仅标记一下GC Roots能直接关联到的对象；

      且修改TAMS（Next Top at Mark Start）,让下一阶段并发运行时，用户程序能在正确可用的Region中创建新对象；

      需要"Stop The World"，但速度很快；

**（B）、并发标记（Concurrent Marking）**

      进行GC Roots Tracing的过程；

      刚才产生的集合中标记出存活对象；

      耗时较长，但应用程序也在运行；

      并不能保证可以标记出所有的存活对象；

**（C）、最终标记（Final Marking）**

      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；

      上一阶段对象的变化记录在线程的Remembered Set Log；

      这里把Remembered Set Log合并到Remembered Set中；

      需要"Stop The World"，且停顿时间比初始标记稍长，但远比并发标记短；

      采用多线程并行执行来提升效率；

**（D）、筛选回收（Live Data Counting and Evacuation）**

      首先排序各个Region的回收价值和成本；

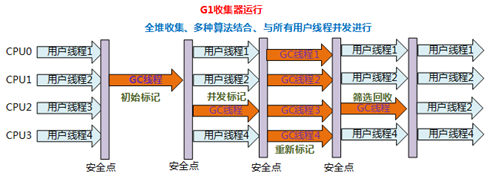
      然后根据用户期望的GC停顿时间来制定回收计划；

      最后按计划回收一些价值高的Region中垃圾对象；

      回收时采用"复制"算法，从一个或多个Region复制存活对象到堆上的另一个空的Region，并且在此过程中压缩和释放内存；

      可以并发进行，降低停顿时间，并增加吞吐量；

**G1收集器运行示意图如下：**



更多G1收集器信息请参考：

      《垃圾收集调优指南》 9节 Garbage-First Garbage Collector：[http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1\_gc.html#garbage\_first\_garbage\_collection](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc.html%23garbage_first_garbage_collection)

      《垃圾收集调优指南》 10节 Garbage-First Garbage Collector Tuning：[http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1\_gc\_tuning.html#g1\_gc\_tuning](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc_tuning.html%23g1_gc_tuning)

**到这里，我们大体了解HotSpot虚拟机中的所有垃圾收集器，后面我们将去了解JVM的一些内存分配与回收策略、JVM垃圾收集相关调优方法……**

**Java虚拟机垃圾回收(四) 总结：**

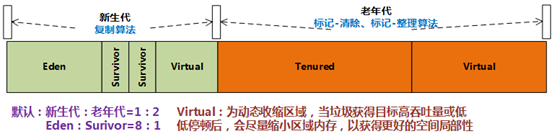
**内存分配与回收策略 方法区垃圾回收 以及 JVM垃圾回收的调优方法**

       在《[Java对象在Java虚拟机中的创建过程](http://blog.csdn.net/tjiyu/article/details/53923392)》了解到对象创建的内存分配，在《[Java内存区域 JVM运行时数据区](http://blog.csdn.net/tjiyu/article/details/53915869)》中了解到各数据区有些什么特点、以及相关参数的调整，在[《Java虚拟机垃圾回收(一) 基础》](http://blog.csdn.net/tjiyu/article/details/53982412)中了解到如何判断对象是存活还是已经死亡？在[《Java虚拟机垃圾回收(二) 垃圾回收算法》](http://blog.csdn.net/tjiyu/article/details/53983064)了解到Java虚拟机垃圾回收的几种常见算法，在《[Java虚拟机垃圾回收(三) 7种垃圾收集器](http://blog.csdn.net/tjiyu/article/details/53983650)》了解到几种收集器的特点和应用等。

       下面来了解总结前面的一些内容：主要包括内存分配与回收策略、方法区垃圾回收、以及JVM垃圾回收的调优方法、垃圾收集器选择。

## ****1、内存分配与回收策略****

       通过在[《Java虚拟机垃圾回收(二) 垃圾回收算法》](http://blog.csdn.net/tjiyu/article/details/53983064)"4、分代收集算法"中，我们知道目前几乎所有商业虚拟机的垃圾收集器都采用分代收集算法，对于HotSpot一般的年代内存划分，如下图：



      对象的内存分配从大体上讲：

       在堆上分配（JIT编译优化后可能在栈上分配），主要在新生代的Eden区中分配；

       如果启用了本地线程分配缓冲，将线程优先在TLAB上分配；

       少数情况下，可能直接分配在老年代中。

       分配的细节取决于当前使用哪种垃圾收集器组合，以及JVM中内存相关参数设置。

       接下来将会讲解几条最普遍的内存分配规则。

### ****1-1、对象优先在Eden分配****

       前面文章曾介绍**HotSpot虚拟机新生代内存布局及算法**

       （1）、将新生代内存分为一块较大的Eden空间和两块较小的Survivor空间；

       （2）、每次使用Eden和其中一块Survivor；

       （3）、当回收时，将Eden和使用中的Survivor中还存活的对象一次性复制到另外一块Survivor；

       （4）、而后清理掉Eden和使用过的Survivor空间；

       （5）、后面就使用Eden和复制到的那一块Survivor空间，重复步骤3；

      默认Eden：Survivor=8:1，即每次可以使用90%的空间，只有一块Survivor的空间被浪费；

       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，JVM将发起一次Minor GC（新生代GC）；

       Minor GC时，如果发现存活的对象无法全部放入Survivor空间，只好通过分配担保机制提前转移到老年代。

### ****1-2、大对象直接进入老年代****

       大对象指需要大量连续内存空间的Java对象，如，很长的字符串、数组；

       经常出现大对象容易导致内存还有不少空间就提前触发GC,以获取足够的连续空间来存放它们，所以应该尽量避免使用创建大对象；

**"-XX:PretenureSizeThreshold"：**

       可以设置这个阈值，大于这个参数值的对象直接在老年代分配；

       默认为0（无效），且只对Serail和ParNew两款收集器有效；

       如果需要使用该参数，可考虑ParNew+CMS组合。

### ****1-3、长期存活的对象将进入老年代****

       JVM给每个对象定义一个对象年龄计数器，其计算流程如下：

       在Eden中分配的对象，经Minor GC后还存活，就复制移动到Survivor区，年龄为1；

       而后每经一次Minor GC后还存活，在Survivor区复制移动一次，年龄就增加1岁；

       如果年龄达到一定程度，就晋升到老年代中；

**"-XX:MaxTenuringThreshold"：**

      设置新生代对象晋升老年代的年龄阈值，默认为15；

### ****1-4、动态对象年龄判定****

       JVM为更好适应不同程序，不是永远要求等到MaxTenuringThreshold中设置的年龄；

**如果在Survivor空间中相同年龄的所有对象大小总和大于Survivor空间的一半，大于或等于该年龄的对象就可以直接进入老年代**；

### ****1-5、空间分配担保****

       在前面曾简单介绍过分配担保：

       当Survivor空间不够用时，需要依赖其他内存（老年代）进行分配担保（Handle Promotion）；

      分配担保的流程如下：

       在发生Minor GC前，JVM先检查老年代最大可用的连续空间是否大于新生所有对象空间；

       如果大于，那可以确保Minor GC是安全的；

       如果不大于，则JVM查看HandlePromotionFailure值是否允许担保失败；

       如果允许，就继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小；

       如果大于，将尝试进行一次Minor GC，但这是有风险的；

       如果小于或HandlePromotionFailure值不允许冒险，那这些也要改为进行一次Full GC；

      尝试Minor GC的风险--担保失败：

       因为尝试Minor GC前面，无法知道存活的对象大小，所以使用历次晋升到老年代对象的平均大小作为经验值；

       假如尝试的Minor GC最终存活的对象远远高于经验值的话，会导致担保失败（Handle Promotion Failure）；

       失败后只有重新发起一次Full GC，这绕了一个大圈，代价较高；

       但一般还是要开启HandlePromotionFailure，避免Full GC过于频繁，而且担保失败概率还是比较低的；

      JDK6-u24后，JVM代码中已经不再使用HandlePromotionFailure参数了；

      规则变为：

**只要老年代最大可用的连续空间大于新生所有对象空间或历次晋升到老年代对象的平均大小，就会进行Minor GC；否则进行Full GC；**

      即老年代最大可用的连续空间小于新生所有对象空间时，不再检查HandelPromotionFailure，而直接检查历次晋升到老年代对象的平均大小；

## ****2、回收方法区****

      在《[Java内存区域 JVM运行时数据区](http://blog.csdn.net/tjiyu/article/details/53915869)》曾介绍过方法区及相关的回收问题，虽然JVM规范规定这个区域可以不实现垃圾收集，且针对常量池和类型卸载的收回效果不佳，但方法区实现垃圾回收是必要的，下面再来详细了解。

### ****2-1、方法区（永久代）的主要回收对象****

**1、废弃常量**

       与回收Java堆中对象非常类似；

**2、无用的类**

       同时满足下面3个条件才能算"无用的类"：

       （1）、该类所有实例都已经被回收（即Java椎中不存在该类的任何实例）；

       （2）、加载该类的ClassLoader已经被回收，也即通过引导程序加载器加载的类不能被回收；

       （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；

### ****2-2、需要注意方法区回收的应用****

       在**大量使用反射、动态代理、经常动态生成大量类的应用**，要注意类的回收；

       如运行时动态生成类的应用：

       1、CGLib在Spring、Hibernate等框架中对类进行增强时会使用；

       2、VM的动态语言也会动态创建类来实现语言的动态性；

       3、另外，JSP（第一次使用编译为Java类）、基于OSGi频繁自定义ClassLoader的应用（同一个类文件，不同加载器加载视为不同类）等；

### ****2-3、HotSpot虚拟机的相关调整****

**1、在JDK7中**

       使用永久代（Permanent Generation）实现方法区，这样就可以不用专门实现方法区的内存管理，但这容易引起内存溢出问题；

       有规划放弃永久代而改用Native Memory来实现方法区；

       不再在Java堆的永久代中生成中分配字符串常量池，而是在Java堆其他的主要部分（年轻代和老年代）中分配；

       更多请参考：<http://docs.oracle.com/javase/8/docs/technotes/guides/vm/enhancements-7.html>

**2、在JDK8中**

      永久代已被删除，类元数据（Class Metadata）存储空间在本地内存中分配，并用显式管理元数据的空间：

       从OS请求空间，然后分成块；

       类加载器从它的块中分配元数据的空间（一个块被绑定到一个特定的类加载器）；

       当为类加载器卸载类时，它的块被回收再使用或返回到操作系统；

       元数据使用由mmap分配的空间，而不是由malloc分配的空间；

**3、相关参数**

      "-XX：MaxMetaspaceSize" （JDK8）：指定类元数据区的最大内存大小；

      "-XX：MetaspaceSize" （JDK8）：指定类元数据区的内存阈值--超过将触发垃圾回收；

**"-Xnolassgc"**：控制是否对类进行回收；

       "-verbose:class"、"-XX:TraceClassLoading"、"-XX:TraceClassUnloading"：查看类加载和卸载信息；

       更多请参考：

 《Java语言规范》12.7 卸载类和接口；

JDK8类元数据说明： [http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/considerations.html#sthref62](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/considerations.html%23sthref62)

## ****3、JVM垃圾回收的调优方法****

       内存回收与垃圾收集器是影响系统性能、并发能力的主要因素之一，一般都需要进行一些手动的测试、调整优化；

       下面介绍的是一些思路，并非是具体的参数设置。

### ****3-1、明确期望的目标（关注点）****

       首先应该明确我们的应用程序调整垃圾回收期望的目标（关注点）是什么？

       在前文曾介绍过通常有这些关注点：

**（1）、停顿时间**

       GC停顿时间越短就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；

       与用户交互较多的场景，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；

**（2）、吞吐量**

       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）；

       高吞吐量可以高效率地利用CPU时间，尽快完成运算的任务，主要适合在后台计算而不需要太多交互的任务；

       应用程序运行在具有多个CPU上，对暂停时间没有特别高的要求；

       程序主要在后台进行计算，而不需要与用户进行太多交互；

       例如，那些执行批量处理、订单处理、工资支付、科学计算的应用程序；

**（3）、覆盖区（Footprint）**

       在达到前面两个目标的情况下，尽量减少堆的内存空间，以获得更好的空间局部性；

       可以减少到不满足前两个目标为止，然后再解决未满足的目标；

       如果是动态收缩的堆设置，堆的大小将随着垃圾收集器试图满足竞争目标而振荡；

**总结就是：低停顿、高吞吐量、少用内存资源；**

        一般这些目标都相互影响的，增大堆内存获得高吞吐量但会增长停顿时间，反之亦然，有时需折中处理。

### ****3-2、JVM自适应调整（Ergonomics）****

       JVM有自适应选择、调整相关设置的功能；

       一般都会先根据平台性能来选择好垃圾收集器，以及设置好其参数；

       在运行中，一些收集器还会收集监控信息来自动地、动态的调整垃圾回收策略；

       所以当我们不知道何如选择收集器和调整时，应该首先让JVM自适应调整；

       然后通过输出GC日志进行分析，看能不能满足明确期望的目标（第一步）；

       如果不能满足，或者通过打印设置的参数信息，发现可以有更好的调优时，可以进行手动指定参数进行设置，并测试；

### ****3-3、实践调优：选择垃圾收集器，并进行相关设置****

       需要明确一个观点：

       没有最好的收集器，更没有万能的收集；

      选择的只能是对具体应用最适合的收集器；     

       我们知道HotSpot有这些组合可以搭配使用：

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

      到实践调优阶段，那必须要了解每个具体收集器的行为特点、优势和劣势、调节参数等（请参考前面的文章内容）；

       然后根据明确期望的目标，选择具体应用最适合的收集器；

       当选择使用某种并行垃圾收集器时，应该指定期望的具体目标而不是指定堆的大小；

       让垃圾收集器自动地、动态的调整堆的大小来满足期望的行为；

       即堆的大小将随着垃圾收集器试图满足竞争目标而振荡；

       当然有时发现问题，堆的大小、划分也是需要进行一些调整的，一般规则：

       除非应用程序无法接受长时间的暂停，否则可以将堆调的尽可能大一些；

       除非发现问题的原因在于老年代的垃圾收集或应用程序暂停次数过多，否则你应该将堆的较大部分分给年轻代；

       等等…    

       例如，使用Parallel Scavenge/Parallel Old组合，这是一种值得推荐的方式：

       1、只需设置好内存数据大小（如"-Xmx"设置最大堆）；

       2、然后使用"-XX:MaxGCPauseMillis"或"-XX:GCTimeRatio"给JVM设置一个优化目标；

       3、那些具体细节参数的调节就由JVM自适应完成；

      设置调整后，应该通过在产生环境下进行不断测试，来分析是否达到我们的目标；

       更多"期望的目标和JVM自适应调整"信息请参考：

       《垃圾收集调优指南》 2节 Ergonomics：[http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/ergonomics.html#ergonomics](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/ergonomics.html%23ergonomics)

       更多"垃圾收集器选择"信息请参考：

       《垃圾收集调优指南》 5节 Available Collectors：[http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html#sthref27](http://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/collectors.html%23sthref27%09)

**到这里，我们大体了解HotSpot虚拟机中的内存分配与回收策略、方法区垃圾回收、以及JVM垃圾回收的调优方法、垃圾收集器选择……**