# 垃圾收集器与内存分配策略

为什么要了解GC和内存分配策略

1、面试需要

2、GC对应用的性能是有影响的；

3、写代码有好处

# 判断对象的存活

引用计数法：快，方便，实现简单，缺点：对象相互引用时，很难判断对象是否改回收。

## 可达性分析

（**面试时重要的知识点，牢记**）

来判定对象是否存活的。这个算法的基本思路就是通过一系列的称为“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。

作为GC Roots的对象包括下面几种：

* 虚拟机栈（栈帧中的本地变量表）中引用的对象。
* 方法区中类静态属性引用的对象。
* 方法区中常量引用的对象。
* 本地方法栈中JNI（即一般说的Native方法）引用的对象。

# 各种引用

## 强引用

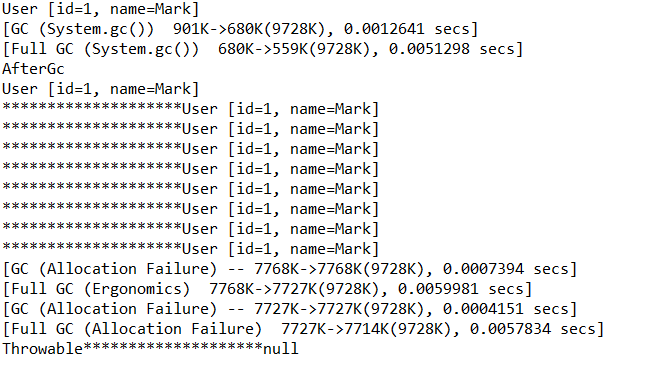
一般的Object obj = new Object() ，就属于强引用。

## 软引用 SoftReference

一些有用但是并非必需，用软引用关联的对象，系统将要发生OOM之前，这些对象就会被回收。参见代码：



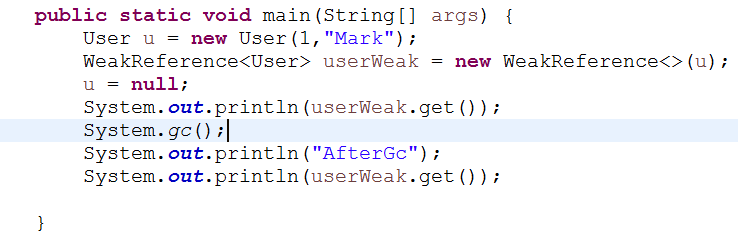
运行结果

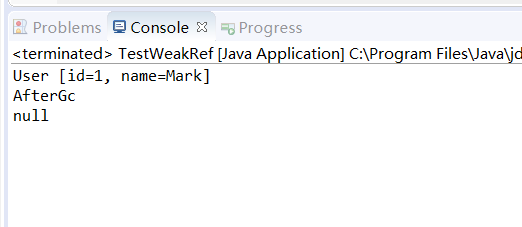


## 弱引用 WeakReference

一些有用（程度比软引用更低）但是并非必需，用弱引用关联的对象，只能生存到下一次垃圾回收之前，GC发生时，不管内存够不够，都会被回收。

参看代码：





## 虚引用 PhantomReference

幽灵引用，最弱，被垃圾回收的时候收到一个通知

**注意：**软引用 SoftReference和弱引用 WeakReference，可以用在内存资源紧张的情况下以及创建不是很重要的数据缓存。当系统内存不足的时候，缓存中的内容是可以被释放的。

例如，一个程序用来处理用户提供的图片。如果将所有图片读入内存，这样虽然可以很快的打开图片，但内存空间使用巨大，一些使用较少的图片浪费内存空间，需要手动从内存中移除。如果每次打开图片都从磁盘文件中读取到内存再显示出来，虽然内存占用较少，但一些经常使用的图片每次打开都要访问磁盘，代价巨大。这个时候就可以用软引用构建缓存。

# 标记-清除算法（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。

# 复制算法（Copying）

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。

# 标记-整理算法（Mark-Compact）

首先标记出所有需要回收的对象，在标记完成后，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。

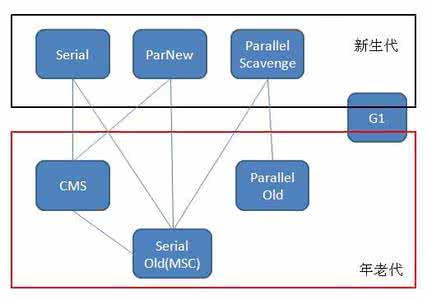
# 把算法们都用上

当前商业虚拟机的垃圾收集都采用“分代收集”（Generational Collection）算法，这种算法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。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。

专门研究表明，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[1]。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）。

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。

**请记住下图的垃圾收集器和之间的连线关系。**



# 垃圾回收器列表

并行：垃圾收集的多线程的同时进行。

并发：垃圾收集的多线程和应用的多线程同时进行。

# 垃圾回收器工作示意图

## Serial/Serial Old

最古老的，单线程，独占式，成熟，适合单CPU 服务器

-XX:+UseSerialGC 新生代和老年代都用串行收集器

-XX:+UseParNewGC 新生代使用ParNew，老年代使用Serial Old

-XX:+UseParallelGC 新生代使用ParallerGC，老年代使用Serial Old

## ParNew

和Serial基本没区别，唯一的区别：多线程，多CPU的，停顿时间比Serial少

-XX:+UseParNewGC 新生代使用ParNew，老年代使用Serial Old

## Parallel Scavenge（ParallerGC）/Parallel Old

关注吞吐量的垃圾收集器，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。

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%。

-XX:+ UseParallelOldGC：新生代使用ParallerGC，老年代使用Parallel Old

-XX:MaxGCPauseMills ：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。

-XX:GCTimeRatio参数的值应当是一个大于0且小于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。

-XX:+UseAdaptiveSizePolicy 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。

如果对于收集器运作原来不太了解，手工优化存在困难的时候，使用Parallel Scavenge收集器配合自适应调节策略，把内存管理的调优任务交给虚拟机去完成将是一个不错的选择。只需要把基本的内存数据设置好（如-Xmx设置最大堆），然后使用MaxGCPauseMillis参数（更关注最大停顿时间）或GCTimeRatio（更关注吞吐量）参数给虚拟机设立一个优化目标，那具体细节参数的调节工作就由虚拟机完成了。自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个重要区别。

## Concurrent Mark Sweep （CMS）

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：

* **初始标记**-短暂，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**-和用户的应用程序同时进行，进行GC RootsTracing的过程
* **重新标记**-短暂，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。
* **并发清除**

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。

-XX:+UseConcMarkSweepGC ，表示新生代使用ParNew，老年代的用CMS

**浮动垃圾：**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”。

同时用户的线程还在运行，需要给用户线程留下运行的内存空间。

-XX:CMSInitialOccupyFraction ，因为以上两点，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 早期版本的默认设置下，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：CMSInitiatingOccupancyFraction设置得太高很容易导致大量“Concurrent Mode Failure”失败，性能反而降低。

-XX:+UseCMSCompactAtFullCollection为了解决这个问题，CMS收集器提供了一个这个开关参数（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。

-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FullGC时都进行碎片整理）。

## G1

-XX:+UseG1GC

**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。

**内存布局：**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。

* 新生代GC

回收Eden区和survivor区，回收后，所有eden区被清空，存在一个survivor区保存了部分数据。老年代区域会增多，因为部分新生代的对象会晋升到老年代。

* 并发标记周期

**初始标记：**短暂，仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，产生一个全局停顿，都伴随有一次新生代的GC。

**根区域扫描：**扫描survivor区可以直接到达的老年代区域。

**并发标记阶段：**扫描和查找整个堆的存活对象，并标记。

**重新标记：**会产生全局停顿，对并发标记阶段的结果进行修正。

**独占清理**：会产生全局停顿，对GC回收比例进行排序，供混合收集阶段使用

**并发清理**：识别并清理完全空闲的区域，并发进行

* 混合收集

对含有垃圾比例较高的Region进行回收。

G1当出现内存不足的的情况，也可能进行的FullGC回收。

G1中重要的参数：

-XX:MaxGCPauseMillis 指定目标的最大停顿时间，G1尝试调整新生代和老年代的比例，堆大小，晋升年龄来达到这个目标时间。

-XX:ParallerGCThreads：设置GC的工作线程数量

# 未来的垃圾回收

ZGC通过技术手段把stw的情况控制在仅有一次，就是第一次的初始标记才会发生，这样也就不难理解为什么GC停顿时间不随着堆增大而上升了，再大我也是通过并发的时间去回收了

关键技术

1. **有色指针（Colored Pointers）**
2. **加载屏障（Load Barrier）**

# Stop The World现象

GC收集器和我们GC调优的目标就是尽可能的减少STW的时间和次数。

# 内存分配与回收策略

**对象优先在Eden分配**，如果说Eden内存空间不足，就会发生Minor GC

**大对象直接进入老年代**，大对象：需要大量连续内存空间的Java对象，比如很长的字符串和大型数组，1、导致内存有空间，还是需要提前进行垃圾回收获取连续空间来放他们，2、会进行大量的内存复制。

-XX:PretenureSizeThreshold 参数 ，大于这个数量直接在老年代分配，缺省为0 ，表示绝不会直接分配在老年代。

**长期存活的对象将进入老年代**，默认15岁，-XX:MaxTenuringThreshold调整

**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，虚拟机并不是永远地要求对象的年龄必须达到了MaxTenuringThreshold才能晋升老年代，如果在Survivor空间中相同年龄所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到MaxTenuringThreshold中要求的年龄

**空间分配担保**：新生代中有大量的对象存活，survivor空间不够，当出现大量对象在MinorGC后仍然存活的情况（最极端的情况就是内存回收后新生代中所有对象都存活），就需要老年代进行分配担保，把Survivor无法容纳的对象直接进入老年代.只要老年代的连续空间大于新生代对象的总大小或者历次晋升的平均大小，就进行Minor GC，否则FullGC。

# 新生代配置

新生代大小配置参数的优先级：

高：-XX:NewSize/MaxNewSize

中间 -Xmn （NewSize= MaxNewSize）

低：-XX:NewRatio 表示比例，例如=2，表示 新生代：老年代 = 1:2

-XX:SurvivorRatio 表示Eden和Survivor的比值，

缺省为8 表示 Eden:FromSurvivor:ToSurvivor= 8:1:1

**同样的代码情况下：**

-Xms20M -Xmx20M -XX:+PrintGCDetails –Xmn2m -XX:SurvivorRatio=2

没有垃圾回收

数组都在老年代

-Xms20M -Xmx20M -XX:+PrintGCDetails -Xmn7m -XX:SurvivorRatio=2

发生了垃圾回收

新生代存了部分数组，老年代也保存了部分数组，发生了晋升现象

-Xms20M -Xmx20M -XX:+PrintGCDetails -Xmn15m -XX:SurvivorRatio=8

新生代可以放下所有的数组

老年代没放

-Xms20M -Xmx20M -XX:+PrintGCDetails -XX:NewRatio=2

发生了垃圾回收

出现了空间分配担保，而且发生了FullGC

# 内存泄漏和内存溢出辨析

内存溢出：实实在在的内存空间不足导致；

内存泄漏：该释放的对象没有释放，多见于自己使用容器保存元素的情况下。

# JDK为我们提供的工具

**jps**

列出当前机器上正在运行的虚拟机进程

-p  :仅仅显示VM 标示，不显示jar,class, main参数等信息.

-m:输出主函数传入的参数. 下的hello 就是在执行程序时从命令行输入的参数

-l: 输出应用程序主类完整package名称或jar完整名称.

-v: 列出jvm参数, -Xms20m -Xmx50m是启动程序指定的jvm参数

**jstat**

是用于监视虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据，在没有GUI图形界面，只提供了纯文本控制台环境的服务器上，它将是运行期定位虚拟机性能问题的首选工具。

假设需要每250毫秒查询一次进程2764垃圾收集状况，一共查询20次，那命令应当是：jstat-gc 2764 250 20

常用参数：

-class (类加载器)

-compiler (JIT)

-gc (GC堆状态)

-gccapacity (各区大小)

-gccause (最近一次GC统计和原因)

-gcnew (新区统计)

-gcnewcapacity (新区大小)

-gcold (老区统计)

-gcoldcapacity (老区大小)

-gcpermcapacity (永久区大小)

-gcutil (GC统计汇总)

-printcompilation (HotSpot编译统计)

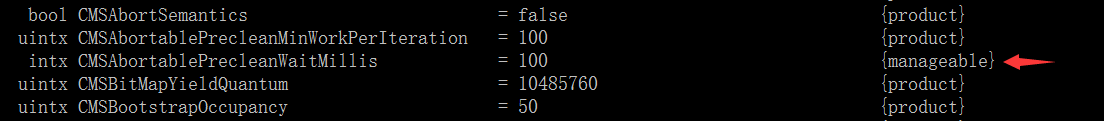
**jinfo**

查看和修改虚拟机的参数

jinfo –sysprops 可以查看由System.getProperties()取得的参数

jinfo –flag 未被显式指定的参数的系统默认值

jinfo –flags（注意s）显示虚拟机的参数

jinfo –flag +[参数] 可以增加参数，但是仅限于由java -XX:+PrintFlagsFinal –version查询出来且

为manageable的参数

jinfo –flag -[参数] 可以去除参数

Thread.*getAllStackTraces*();

**jmap**

用于生成堆转储快照（一般称为heapdump或dump文件）。jmap的作用并不仅仅是为了获取dump文件，它还可以查询finalize执行队列、Java堆和永久代的详细信息，如空间使用率、当前用的是哪种收集器等。和jinfo命令一样，jmap有不少功能在Windows平台下都是受限的，除了生成dump文件的-dump选项和用于查看每个类的实例、空间占用统计的-histo选项在所有操作系统都提供之外，其余选项都只能在Linux/Solaris下使用。

jmap -dump:live,format=b,file=heap.bin <pid>

Sun JDK提供jhat（JVM Heap Analysis Tool）命令与jmap搭配使用，来分析jmap生成的堆转储快照。

**jhat**

jhat dump文件名

后屏幕显示“Server is ready.”的提示后，用户在浏览器中键入http://localhost：7000/就可以访问详情

**jstack**

（Stack Trace for Java）命令用于生成虚拟机当前时刻的线程快照。线程快照就是当前虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等都是导致线程长时间停顿的常见原因。

在代码中可以用java.lang.Thread类的getAllStackTraces（）方法用于获取虚拟机中所有线程的StackTraceElement对象。使用这个方法可以通过简单的几行代码就完成jstack的大部分功能，在实际项目中不妨调用这个方法做个管理员页面，可以随时使用浏览器来查看线程堆栈。

**管理远程进程需要在远程程序的启动参数中增加：**

-Djava.rmi.server.hostname=…..

-Dcom.sun.management.jmxremote

-Dcom.sun.management.jmxremote.port=8888

-Dcom.sun.management.jmxremote.authenticate=false

-Dcom.sun.management.jmxremote.ssl=false

**visualvm**

插件中心地址

[https://visualvm.github.io](https://visualvm.github.io/archive/uc/8u40/updates.xml.gz)

但是注意版本问题，不同的JDK所带的visualvm是不一样的，下载插件时需要下对应的版本。

**浅堆和深堆**

浅堆 :（Shallow Heap）是指一个对象所消耗的内存。例如，在32位系统中，一个对象引用会占据4个字节，一个int类型会占据4个字节，long型变量会占据8个字节，每个对象头需要占用8个字节。

深堆 ：这个对象被GC回收后，可以真实释放的内存大小，也就是只能通过对象被**直接**或**间接**访问到的所有对象的集合。通俗地说，就是指仅被对象所持有的对象的集合。深堆是指对象的保留集中所有的对象的浅堆大小之和。

举例：对象A引用了C和D，对象B引用了C和E。那么对象A的浅堆大小只是A本身，不含C和D，而A的实际大小为A、C、D三者之和。而A的深堆大小为A与D之和，由于对象C还可以通过对象B访问到，因此不在对象A的深堆范围内。