**《深入理解Java虚拟机》笔记:**

一.Java内存区域:

1、程序计数器:

程序计数器是一块较小的内存空间,可以看作是当前线程所执行的字节码的行号指示器。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。线程私有。唯一一个没有任何OutOfMemoryError情况的区域。

2、Java虚拟机栈：

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帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。每个方法从调用直至执行结束完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

局部变量表存放了编译期可知的各种基本数据类型、对象引用和returnAddress类型。

线程私有。该区域有两种异常状况：如果线程请求的栈深度大于虚拟机栈所允许的深度，将抛出StackOverFlowError异常；如果虚拟机栈可以动态扩展，如果扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。

3、本地方法栈：

和虚拟机栈发挥的作用相似，不过虚拟机栈为虚拟机执行Java方法服务，本地方法栈则为虚拟机执行Native方法服务。

4、堆：

Java堆是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块区域。在虚拟机启动时创建。次内存区域的唯一目的就是存放对象实例，几乎所有的对象实例以及数组都在这里分配内存。

如果堆中没有内存完成实例分配，并且无法扩展时，就会抛出OutOfMemoryError异常。

5、方法区：

方法区也是线程共享的内存区域，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。

该区域的内存回收目标主要是针对常量池的回收和对类型的卸载。

当方法区无法满足内存分配需求时，将抛出OutOfMemoryError异常。

6、运行时常量池：

运行时常量池是方法区的一部分。Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池。运行时常量池相对于Class文件常量池的另一个重要特性是具备动态性，运行期间也可能将新的常量放入池中，用的较多的是String类的intern（）方法。

二.Java对象:

1.对象内存布局：

分为对象头、示例数据、对齐填充三部分。

对象头：一部分用于存储对象自身的运行时数据，如哈希码、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等；另一部分是类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例。

实例数据:对象真正存储的有效信息,也是代码中所定义的各种类型的字段内容。

对齐填充：并不是必然存在的，没有什么特殊含义，仅仅起着占位符的作用。由于对象的大小必须是8字节的整数倍，而对象头部分正好是8字节的倍数，因此，当对象实例数据部分没有对齐时，就需要通过对齐填充来补全。

2、对象访问定位：

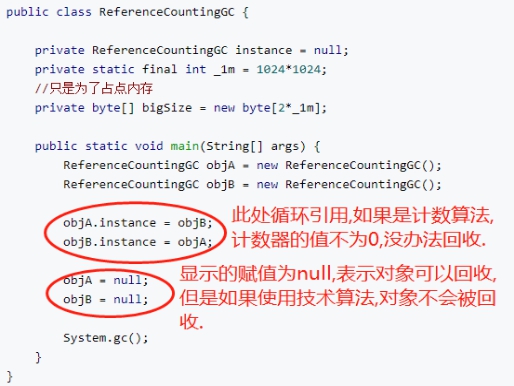
2.1、使用句柄访问：Java堆中会划分出一块内存用来作为句柄池，reference中存储的就是对象的句柄地址，句柄中包含了对象实例数据与类型数据的具体地址信息。使用句柄来访问的最大好处就是存储的句柄地址稳定，对象被移动时只会改变句柄中的实例数据指针，而reference本身不需要修改。

2.2、使用直接指针访问：Java堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，而reference中存储的直接就是对象地址。使用直接指针的最大好处就是速度更快，节省了一次指针定位的时间开销。虚拟机Sun HotSpot就是使用直接指针访问。

三.判断对象存活算法:

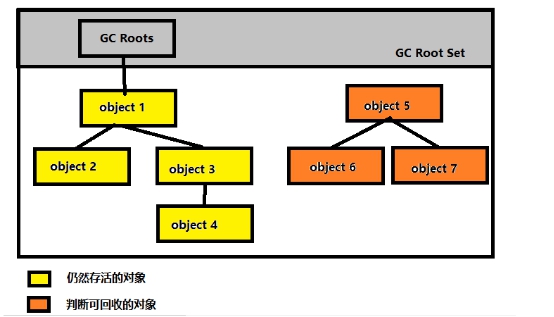
1.引用计数算法:

给对象添加一个引用计数器,每当一个地方引用它时,计数器值加1;当引用失效时,计数器值减1;任何时候计数器为0的对象就是不可能再被使用的.该算法没办法解决循环引用的问题.



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（用图论的话来说，就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。如下图所示，对象object 5、object 6、object 7虽然互相有关联，但是它们到GC Roots是不可达的，所以它们将会被判定为是可回收的对象。(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.当对象没有覆盖finalize()方法或者finalize()方法以及被JVM虚拟机调用过一次,此种情况下会被视为”没有必要执行”.finalize()方法最多会被自动调用一次.)



在Java语言中，可作为GC Roots的对象包括下面几种：

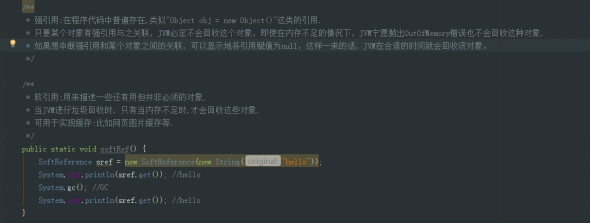
1. 虚拟机栈（栈帧中的本地变量表）中引用的对象。

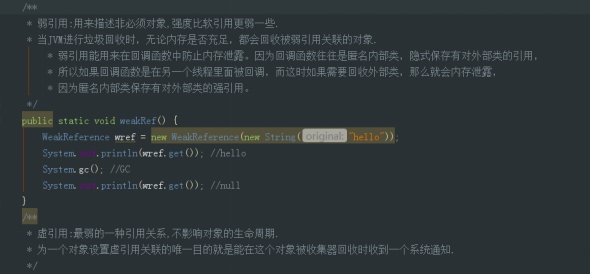
2. 方法区中类静态属性引用的对象。

3. 方法区中常量引用的对象。

4. 本地方法栈中JNI（即一般说的Native方法）引用的对象。

四.Java四种引用:







五.方法区回收:

永久代的垃圾回收主要两个部分:废弃常量和无用的类。

1.“废弃常量”条件:假如常量池中有”abc”,但是系统中并没有任何地方引用该字面量,必要情况下,该常量就废弃并被清理。常量池中的其他类(接口)、方法、字段的符号引用与此类似。

2.“无用的类”条件：该类所有的实例已经被回收；加载该类的ClassLoader已经被回收；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类的方法。

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。

**六.垃圾收集算法:**

**1.标记-清除算法:**

首先标记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。

标记过程：判断对象是否存活主要采用引用计数算法和可达性分析算法，Java采用可达性分析算法；要宣告一个对象死亡，至少要经历两次标记过程：如果对象进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被标记并且进行一次筛选，筛选条件是次对象是否执行finalize（）方法。

不足：1.效率问题，标记和清楚的效率都不高。

2.空间问题，标记清除之后会产生大量的不连续内存碎片，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一次垃圾收集动作。

**2.复制算法:**

它将可用内存按容量划分为大小相等的两块，每次都使用其中的一块。当这一块内存用完了，就将还存活的对象复制到另一块上面，然后再对已使用过的内存空间一次清理掉。这样使得每次都是对半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆栈指针，按顺序分配内存即可。

该算法主要用于回收新生代。内存划分时并不需要按照1:1的比例划分，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和其中的一块Survivor。当回收时，将Eden和Survivor中还存活的对象一次性的复制到另外一块Survivor上，然后清理掉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。

**3.标记-整理算法:**

根据老年代的特点，有人提出了“标记-整理”算法，标记过程与“标记-清理”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，然后直接清理端边界外的内存。

**4.分代收集算法:**

当前商业虚拟机都采用“分代收集”算法。在新生代中，每次垃圾收集时都发现有大量对象死亡，只有少量存活，那就采用复制算法，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“标记-整理”算法来进行回收。

**七.HotSpot算法实现:**

1.枚举根节点：

可作为GC Roots的节点主要在全局性的引用（例如常量和静态属性）与执行上下文（例如栈帧中的本地变量表）中。可达性分析对执行时间的敏感体现在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--这里的“一致性”的意思是指在分析过程中整个执行系统看起来就像被冻结在某个时间点上，不可以出现分析过程中对象引用关系还在不断的变化的情况。所以GC进行时必须停顿所有的Java执行线程（Stop The World）。

虚拟机应当是有办法直接得知那些地方存放着对象引用，在Hot Spot的实现中，是使用一组成为OopMap的数据结构来达到这个目的的，在类加载完成时，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下栈和寄存器中哪些位置是引用。

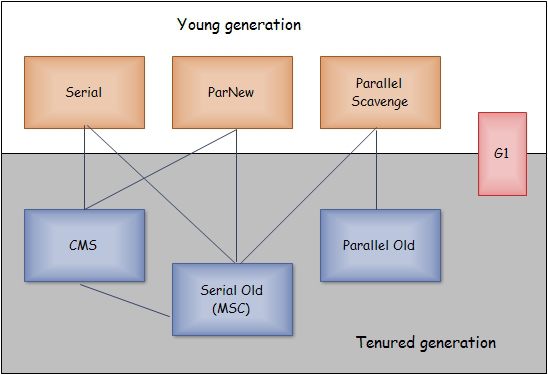
2.安全点：

HotSpot没有为每条指令都生成OopMap，只是在“特定的位置”记录了这些信息，这些位置称为安全点（Safepoint），即程序执行时并非在所有地方都能停顿下来开始GC，只有在到达安全点才能停顿。安全点的选定基本上是以程序“是否具有让程序长时间执行的特征”为标准进行选定的。“长时间执行”的最明显特征就是指令序列复用，例如方法调用、循环跳转、异常跳转等，所以具有这些功能的指令才会产生Safepoint。

3.安全区域：

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。可以吧Safe Region看做是被扩展了的Safepoint。

**八.垃圾收集器:**



1.Serial收集器:

最基本、发展历史最悠久的收集器，在jdk1.3.1之前是新生代的唯一选择。是一个单线程的收集器。在它进行垃圾回收时必须暂停其他所有的工作现场，知道它收集结束。对于运行在Clients模式下的虚拟机来说是一个很好的选择。



2.ParNew收集器：

是Serial收集器的多线程版本，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。



3.Parallel Scavenge收集器：

是一个新生代的收集器，它也是采用复制算法的收集器，又是并行的多线程收集器。由于与吞吐量关系密切，又称为“吞吐量优先”收集器。该收集器有一个参数-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的比例（-XX:SurvivorRatio）、晋升老年代对象大小（-XX:PretenureSizeThreshold）等细节参数，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）



4.Serial Old收集器：

是Serial收集器的老年代版本，也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在于给Clients模式下的虚拟机使用。



5.Parallel Old收集器：

是Parallel Scavenge收集器的老年代版本,使用多线程和”标记-整理”算法。



6.CMS收集器：

第一款真正意义上的并发收集器，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。是一种以获取最短回收停顿时间为目标的收集器。是老年代收集器，基于“标记-清除”算法。

步骤：

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）

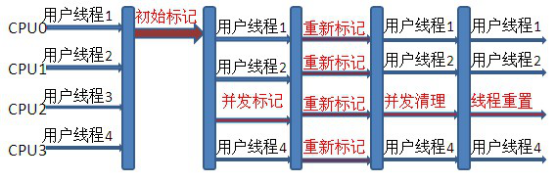
初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，并发标记阶段就是进行GC Roots Tracing的过程，重新标记阶段则是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录。

缺点：

1.CMS收集器对CPU资源非常敏感。

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Full GC的产生。

3.因为基于“标记-清除”算法实现，意味着收集结束时会有大量的空间碎片产生。



7.G1收集器：

G1收集器一款面向服务端应用的收集器。

特点：

1.并行与并发

2.分代收集

3.空间整合

4.可预测的停顿

步骤：

1.初始标记（Initial mark）

2.并发标记（Concurrent mark）

3.最终标记（Final Marking）

4.筛选回收（Live Data Counting and Evacuation）

并行与并发：

并行：指多条垃圾收集线程并行工作，但此时用于线程处于等待状态。

并发：指用户线程与垃圾收集线程同时执行（但并不一定是并行的，可能交替执行），用户程序在继续运行，而垃圾收集器程序运行于另一个CPU上。

吞吐量：

指CPU用于运行用户代码的时间与CPU总消耗时间的比值。

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）

**垃圾收集器参数总结：**

|  |  |
| --- | --- |
| **参数** | **描述** |
| **-XX:+UseSerialGC** | Jvm运行在Client模式下的默认值，打开此开关后，使用Serial + Serial Old的收集器组合进行内存回收 |
| **-XX:+UseParNewGC** | 打开此开关后，使用ParNew + Serial Old的收集器进行垃圾回收 |
| **-XX:+UseConcMarkSweepGC** | 使用ParNew + CMS +  Serial Old的收集器组合进行内存回收，Serial Old作为CMS出现“Concurrent Mode Failure”失败后的后备收集器使用。 |
| **-XX:+UseParallelGC** | Jvm运行在Server模式下的默认值，打开此开关后，使用Parallel Scavenge +  Serial Old的收集器组合进行回收 |
| **-XX:+UseParallelOldGC** | 使用Parallel Scavenge +  Parallel Old的收集器组合进行回收 |
| **-XX:SurvivorRatio** | 新生代中Eden区域与Survivor区域的容量比值，默认为8，代表Eden:Subrvivor = 8:1 |
| **-XX:PretenureSizeThreshold** | 直接晋升到老年代对象的大小，设置这个参数后，大于这个参数的对象将直接在老年代分配 |
| **-XX:MaxTenuringThreshold** | 晋升到老年代的对象年龄，每次Minor GC之后，年龄就加1，当超过这个参数的值时进入老年代 |
| **-XX:UseAdaptiveSizePolicy** | 动态调整java堆中各个区域的大小以及进入老年代的年龄 |
| **-XX:+HandlePromotionFailure** | 是否允许新生代收集担保，进行一次minor gc后, 另一块Survivor空间不足时，将直接会在老年代中保留 |
| **-XX:ParallelGCThreads** | 设置并行GC进行内存回收的线程数 |
| **-XX:GCTimeRatio** | GC时间占总时间的比列，默认值为99，即允许1%的GC时间，仅在使用Parallel Scavenge 收集器时有效 |
| **-XX:MaxGCPauseMillis** | 设置GC的最大停顿时间，在Parallel Scavenge 收集器下有效 |
| **-XX:CMSInitiatingOccupancyFraction** | 设置CMS收集器在老年代空间被使用多少后出发垃圾收集，默认值为68%，仅在CMS收集器时有效，-XX:CMSInitiatingOccupancyFraction=70 |
| **-XX:+UseCMSCompactAtFullCollection** | 由于CMS收集器会产生碎片，此参数设置在垃圾收集器后是否需要一次内存碎片整理过程，仅在CMS收集器时有效 |
| **-XX:+CMSFullGCBeforeCompaction** | 设置CMS收集器在进行若干次垃圾收集后再进行一次内存碎片整理过程，通常与UseCMSCompactAtFullCollection参数一起使用 |
| **-XX:+DisableExplicitGC** | 是否关闭手动System.gc |

**九．GC日志：**

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

最前面的数字33.125代表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或者[Full GC说明了垃圾收集的停顿类型,不是用来区分新生代GC和老年代GC的,如果有Full说明GC发生了Stop-The-World.

[DefNew或者[Tenured或[Perm表示GC发生的区域,是与GC收集器密切相关的。Serial收集器中的新生代名为“Default New Generation”，所以显示“[DefNew”；ParNew收集器的新生代名为“[ParNew”，意为“Parallel New Generation”；Parallel Scavenge收集器的新生代名为“PSToungGen”。

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)”.

方括号外的”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)”

再往后的”0.0031680 secs”表示该内存区域GC所占用的时间.

**十.内存分配策略:**

**1.对象优先在Eden分配**

**大多数情况下,对象在新生代Eden区分配,当Eden区没有足够内存进行分配时,虚拟机将会发起一次Minor GC。GC期间采用复制算法进行垃圾回收，存活的对象大小如果大于Survivor区内存大小，将不会进入Survivor区，而是通过分配担保机制转移到老年代。**

**2.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。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”他们。虚拟机提供了-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区以及两个Survivor区之间发生大量的内存复制。**

**3.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计数器。如果对象在Eden出生并经过第一次Minor GC后仍存活，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区中每熬过一个Minor GC，年龄就加1，当它的年龄增加到一定程度（默认是15岁），就会晋升到老年代。对象晋升老年代的年龄阙值可以通过参数-XX：MaxTenuringThreshold设置。**

**4.动态对象年龄判定：**

**为了更好的适应不同程序的内存情况，虚拟机并不是永远的要求对象的年龄必须达到MaxTenuringThreshold才能晋升到老年代，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。**

**5.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，如果条件成立，那么Minor GC可以确保是安全的。如果不成立，则虚拟机会查看HandlePromotionFailure设置值是否允许担保失败。如果允许，那么会继续检查老年代最大可用内存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，尽管是有风险的；如果小于或者参数设置为不允许，则进行一次Full GC。在JDK6 Update24之后，HandlePromotionFailure参数不会再影响到虚拟机的空间分配担保策略，规则变为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，否则进行Full GC。**

**Minor GC与Major GC：**

**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特性，所以Minor GC非常频繁，一般回收速度也很快。**

**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。Major GC的速度一般会比Minor GC慢10倍以上。**