**JVM内存管理，Minor GC和Full GC触发机制总结**

<https://blog.csdn.net/weixin_39788856/article/details/80388002>

1、Java垃圾回收机制

        GC，即就是Java垃圾回收机制。目前主流的JVM（HotSpot）采用的是分代收集算法。作为Java开发者，一般不需要专门编写内存回收和垃圾清理代码，对内存泄露和溢出的问题。与C++不同的是，Java采用的是类似于树形结构的可达性分析法来判断对象是否还存在引用。即：从gcroot开始，把所有可以搜索得到的对象标记为存活对象。缺点就是：1. 有可能不知不觉浪费了很多内存。2. JVM花费过多时间来进行内存回收。3. 内存泄露

理解Java的垃圾回收机制，就要从：“什么时候”，“对什么东西”，“做了什么”三个方面来具体分析。

       第一：“什么时候”即就是GC触发的条件。GC触发的条件有两种。（1）程序调用System.gc时可以触发；（2）系统自身来决定GC触发的时机。系统判断GC触发的依据：根据Eden区和From Space区的内存大小来决定。当内存大小不足时，则会启动GC线程并停止应用线程。

       第二：“对什么东西”笼统的认为是Java对象。但是准确来讲，GC操作的对象分为：通过可达性分析法无法搜索到的对象和可以搜索到的对象。对于搜索不到的方法进行标记。

       第三：“做了什么”最浅显的理解为释放对象。但是从GC的底层机制可以看出，对于可以搜索到的对象进行复制操作，对于搜索不到的对象，调用finalize()方法进行释放。

       具体过程：当GC线程启动时，会通过可达性分析法把Eden区和From Space区的存活对象复制到To Space区，然后把Eden Space和From Space区的对象释放掉。当GC轮训扫描To Space区一定次数后，把依然存活的对象复制到老年代，然后释放To Space区的对象。

对于用可达性分析法搜索不到的对象，GC并不一定会回收该对象。要完全回收一个对象，至少需要经过两次标记的过程：

        第一次标记：对于一个没有其他引用的对象，筛选该对象是否有必要执行finalize()方法，如果没有执行必要，则意味可直接回收。（筛选依据：是否复写或执行过finalize()方法；因为finalize方法只能被执行一次）。

        第二次标记：如果被筛选判定位有必要执行，则会放入FQueue队列，并自动创建一个低优先级的finalize线程来执行释放操作。如果在一个对象释放前被其他对象引用，则该对象会被移除FQueue队列。

2、JVM内存管理

        根据JVM规范，JVM把内存划分了如下几个区域：. 方法区、堆区、 本地方法栈、虚拟机栈、程序计数器 。其中，方法区和堆是所有线程共享的。

2.1 方法区

        方法区存放了要加载的类的信息(如类名，修饰符)、类中的静态变量、final定义的常量、类中的field、方法信息，当开发人员调用类对象中的getName、isInterface等方法来获取信息时，这些数据都来源于方法区。方法区是全局共享的，在一定条件下它也会被GC。当方法区使用的内存超过它允许的大小时，就会抛出OutOfMemory：PermGen Space异常。

        在Hotspot虚拟机中，这块区域对应的是Permanent Generation(持久代)，一般的，方法区上执行的垃圾收集是很少的，因此方法区又被称为持久代的原因之一，但这也不代表着在方法区上完全没有垃圾收集，其上的垃圾收集主要是针对常量池的内存回收和对已加载类的卸载。在方法区上进行垃圾收集，条件苛刻而且相当困难。

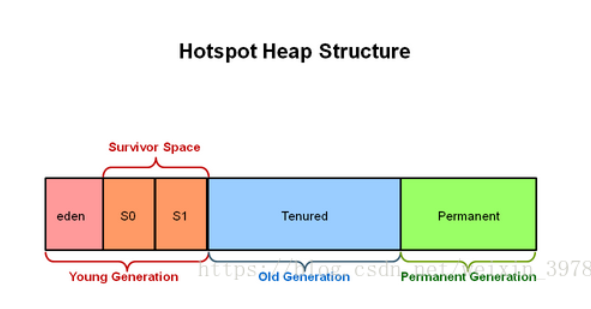
        运行时常量池（Runtime Constant Pool）是方法区的一部分，用于存储编译期就生成的字面常量、符号引用、翻译出来的直接引用（符号引用就是编码是用字符串表示某个变量、接口的位置，直接引用就是根据符号引用翻译出来的地址，将在类链接阶段完成翻译）；运行时常量池除了存储编译期常量外，也可以存储在运行时间产生的常量，比如String类的intern()方法，作用是String维护了一个常量池，如果调用的字符“abc”已经在常量池中，则返回池中的字符串地址，否则，新建一个常量加入池中，并返回地址。

2.2 堆

        堆区是理解Java GC机制最重要的区域。在JVM所管理的内存中，堆区是最大的一块，堆区也是JavaGC机制所管理的主要内存区域，堆区由所有线程共享，在虚拟机启动时创建。堆区用来存储对象实例及数组值，可以认为java中所有通过new创建的对象都在此分配。

        对于堆区大小，可以通过参数-Xms和-Xmx来控制，-Xms为JVM启动时申请的最新heap内存，默认为物理内存的1/64但小于1GB;-Xmx为JVM可申请的最大Heap内存，默认为物理内存的1/4但小于1GB,默认当剩余堆空间小于40%时，JVM会增大Heap到-Xmx大小，可通过-XX:MinHeapFreeRadio参数来控制这个比例；当空余堆内存大于70%时，JVM会减小Heap大小到-Xms指定大小，可通过-XX:MaxHeapFreeRatio来指定这个比例。对于系统而言，为了避免在运行期间频繁的调整Heap大小，我们通常将-Xms和-Xmx设置成一样。为了让内存回收更加高效，从Sun JDK 1.2开始对堆采用了分代管理方式，如下图所示：

---------------------



年轻代（Young Generation）

        对象在被创建时，内存首先是在年轻代进行分配（注意，大对象可以直接在老年代分配）。当年轻代需要回收时会触发Minor GC(也称作Young GC)。

        年轻代由Eden Space和两块相同大小的Survivor Space（又称From Space和To Space）构成，Eden区和Servior区的内存比为8:1，可通过-Xmn参数来调整新生代大小，也可通过-XX:SurvivorRadio来调整Eden Space和Survivor Space大小。不同的GC方式会按不同的方式来按此值划分Eden Space和Survivor Space，有些GC方式还会根据运行状况来动态调整Eden、From Space、To Space的大小。

        年轻代的Eden区内存是连续的，所以其分配会非常快；同样Eden区的回收也非常快（因为大部分情况下Eden区对象存活时间非常短，而Eden区采用的复制回收算法，此算法在存活对象比例很少的情况下非常高效）。如果在执行垃圾回收之后，仍没有足够的内存分配，也不能再扩展，将会抛出OutOfMemoryError:Java Heap Space异常。

老年代（Old Generation）

          老年代用于存放在年轻代中经多次垃圾回收仍然存活的对象，可以理解为比较老一点的对象，例如缓存对象；新建的对象也有可能在老年代上直接分配内存，这主要有两种情况：一种为大对象，可以通过启动参数设置-XX:PretenureSizeThreshold=1024，表示超过多大时就不在年轻代分配，而是直接在老年代分配。此参数在年轻代采用Parallel Scavenge GC时无效，因为其会根据运行情况自己决定什么对象直接在老年代上分配内存；另一种为大的数组对象，且数组对象中无引用外部对象。

        当老年代满了的时候就需要对老年代进行垃圾回收，老年代的垃圾回收称作Full GC。老年代所占用的内存大小为-Xmx对应的值减去-Xmn对应的值。

2.3 本地方法栈（Native Method Stack）

        本地方法栈用于支持native方法的执行，存储了每个native方法调用的状态。本地方法栈和虚拟机方法栈运行机制一致，它们唯一的区别就是，虚拟机栈是执行Java方法的，而本地方法栈是用来执行native方法的，在很多虚拟机中（如Sun的JDK默认的HotSpot虚拟机），会将本地方法栈与虚拟机栈放在一起使用。

2.4 程序计数器（Program Counter Register）

        程序计数器是一个比较小的内存区域，可能是CPU寄存器或者操作系统内存，其主要用于指示当前线程所执行的字节码执行到了第几行，可以理解为是当前线程的行号指示器。字节码解释器在工作时，会通过改变这个计数器的值来取下一条语句指令。 每个程序计数器只用来记录一个线程的行号，所以它是线程私有（一个线程就有一个程序计数器）的。

        如果程序执行的是一个Java方法，则计数器记录的是正在执行的虚拟机字节码指令地址；如果正在执行的是一个本地（native，由C语言编写完成）方法，则计数器的值为Undefined，由于程序计数器只是记录当前指令地址，所以不存在内存溢出的情况，因此，程序计数器也是所有JVM内存区域中唯一一个没有定义OutOfMemoryError的区域。

2.5 虚拟机栈（JVM Stack）

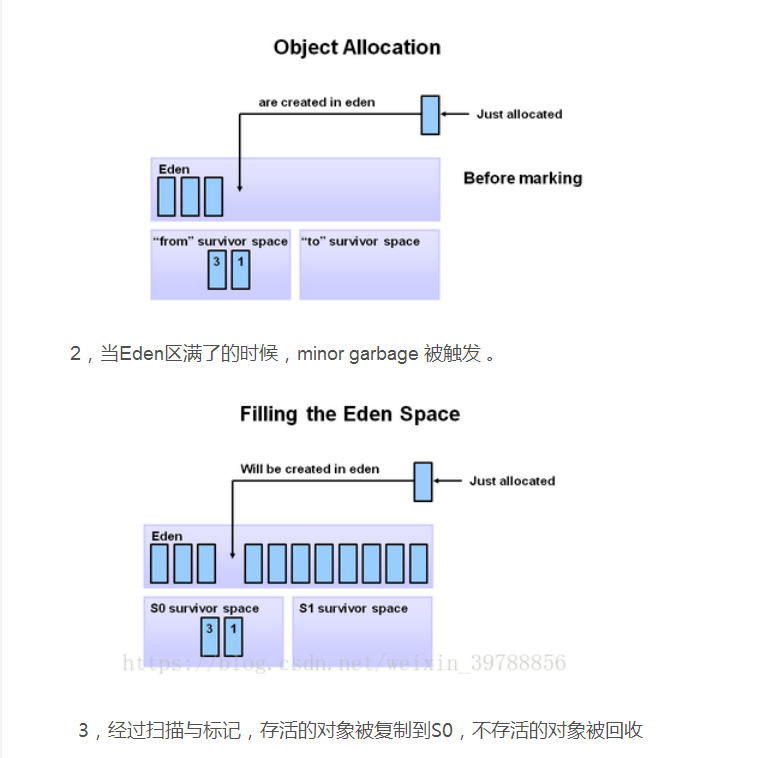
        虚拟机栈占用的是操作系统内存，每个线程都对应着一个虚拟机栈，它是线程私有的，而且分配非常高效。一个线程的每个方法在执行的同时，都会创建一个栈帧（Statck Frame），栈帧中存储的有局部变量表、操作站、动态链接、方法出口等，当方法被调用时，栈帧在JVM栈中入栈，当方法执行完成时，栈帧出栈。

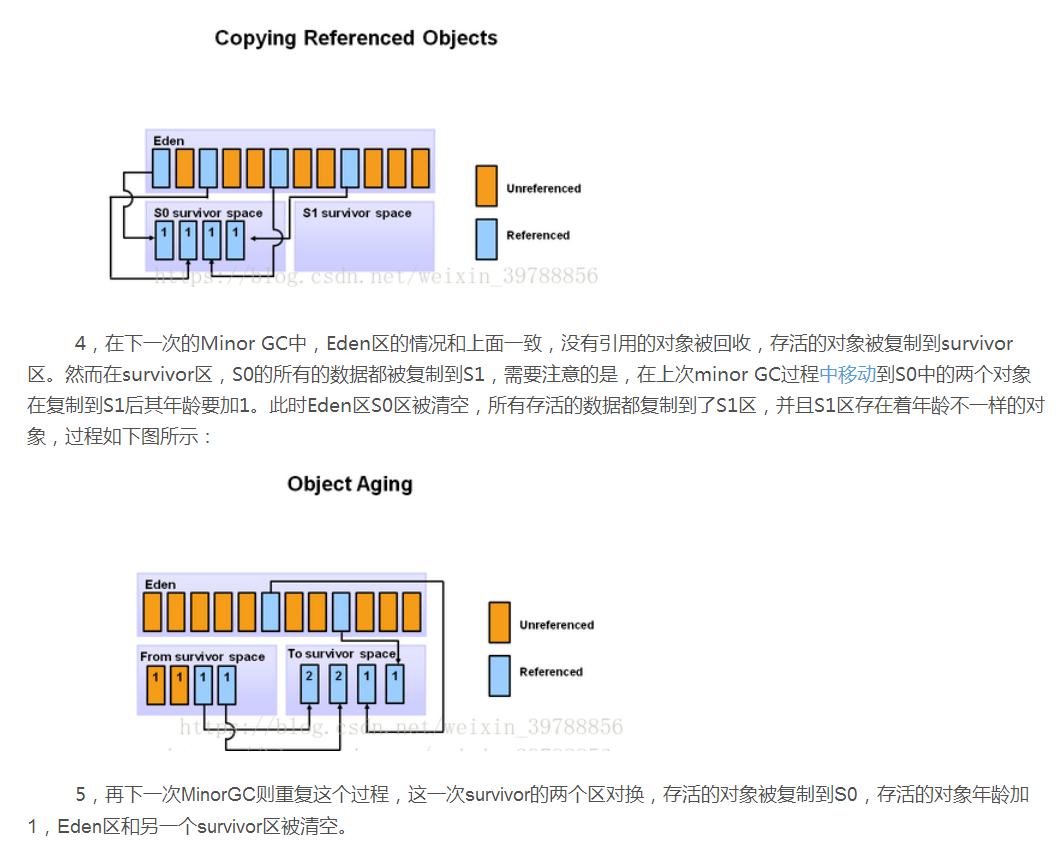
        局部变量表中存储着方法的相关局部变量，包括各种基本数据类型，对象的引用，返回地址等。在局部变量表中，只有long和double类型会占用2个局部变量空间（Slot，对于32位机器，一个Slot就是32个bit），其它都是1个Slot。需要注意的是，局部变量表是在编译时就已经确定好的，方法运行所需要分配的空间在栈帧中是完全确定的，在方法的生命周期内都不会改变。

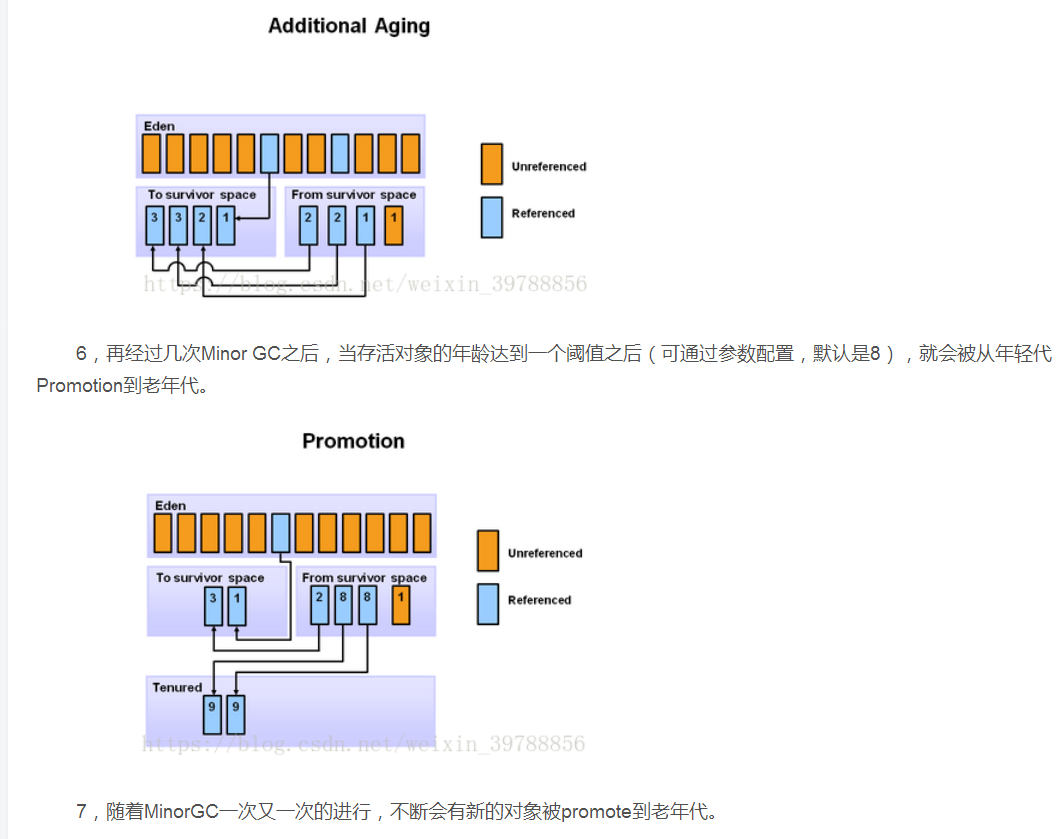
        虚拟机栈中定义了两种异常，如果线程调用的栈深度大于虚拟机允许的最大深度，则抛出StatckOverFlowError（栈溢出）；不过多数Java虚拟机都允许动态扩展虚拟机栈的大小(有少部分是固定长度的)，所以线程可以一直申请栈，直到内存不足，此时，会抛出OutOfMemoryError（内存溢出）。

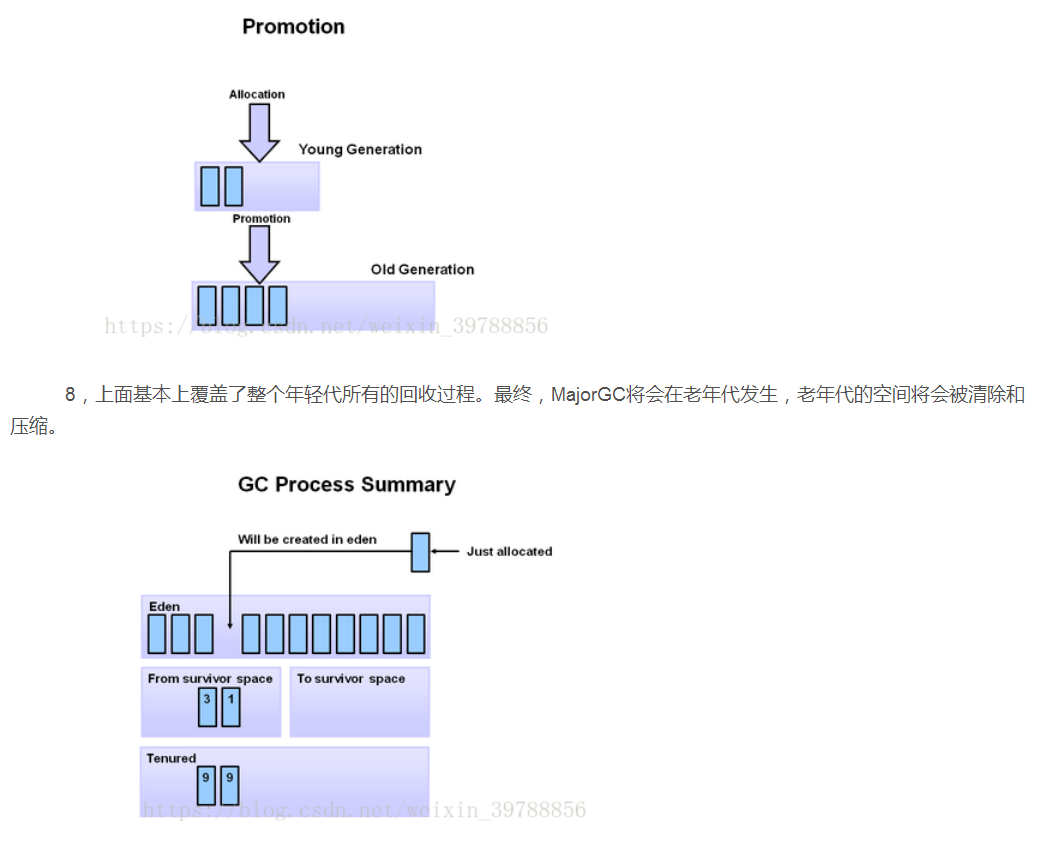
3、虚拟机中GC的过程

        1，在初始阶段，新创建的对象被分配到Eden区，survivor的两块空间都为空。









从上面的过程可以看出，Eden区是连续的空间，且Survivor总有一个为空。经过一次GC和复制，一个Survivor中保存着当前还活着的对象，而Eden区和另一个Survivor区的内容都不再需要了，可以直接清空，到下一次GC时，两个Survivor的角色再互换。因此，这种方式分配内存和清理内存的效率都极高，这种垃圾回收的方式就是著名的“停止-复制（Stop-and-copy）”清理法（将Eden区和一个Survivor中仍然存活的对象拷贝到另一个Survivor中），这不代表着停止复制清理法很高效，其实，它也只在这种情况下（基于大部分对象存活周期很短的事实）高效，如果在老年代采用停止复制，则是非常不合适的。

            老年代存储的对象比年轻代多得多，而且不乏大对象，对老年代进行内存清理时，如果使用停止-复制算法，则相当低效。一般，老年代用的算法是标记-压缩算法，即：标记出仍然存活的对象（存在引用的），将所有存活的对象向一端移动，以保证内存的连续。在发生Minor GC时，虚拟机会检查每次晋升进入老年代的大小是否大于老年代的剩余空间大小，如果大于，则直接触发一次Full GC，否则，就查看是否设置了-XX:+HandlePromotionFailure（允许担保失败），如果允许，则只会进行MinorGC，此时可以容忍内存分配失败；如果不允许，则仍然进行Full GC（这代表着如果设置-XX:+Handle PromotionFailure，则触发MinorGC就会同时触发Full GC，哪怕老年代还有很多内存，所以，最好不要这样做）。

        关于方法区即永久代的回收，永久代的回收有两种：常量池中的常量，无用的类信息，常量的回收很简单，没有引用了就可以被回收。对于无用的类进行回收，必须保证3点：

1. 类的所有实例都已经被回收。2. 加载类的ClassLoader已经被回收。3. 类对象的Class对象没有被引用（即没有通过反射引用该类的地方）。

永久代的回收并不是必须的，可以通过参数来设置是否对类进行回收。

 4、Minor GC ，Full GC 触发条件

Minor GC触发条件：当Eden区满时，触发Minor GC。

Full GC触发条件：

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

（3）方法去空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小。

部分内容选自：https://blog.csdn.net/suifeng3051/article/details/48292193

---------------------

作者：热血的少年

来源：CSDN

原文：https://blog.csdn.net/weixin\_39788856/article/details/80388002

版权声明：本文为博主原创文章，转载请附上博文链接！