##### 一．Jvm内存模型

###### 1. java虚拟机内存模型由下面几部分组成：

**堆：(线程共享)**

**线程栈：(线程私有)**由一个或多个栈帧构成，每个栈帧由：局部变量表、操作数栈、动态链接、方法出口。

局部变量表：程序运行的局部变量信息。

操作数栈：程序计算临时用到内存。

动态链接：存储方法名称对应的具体代码的内存地址。

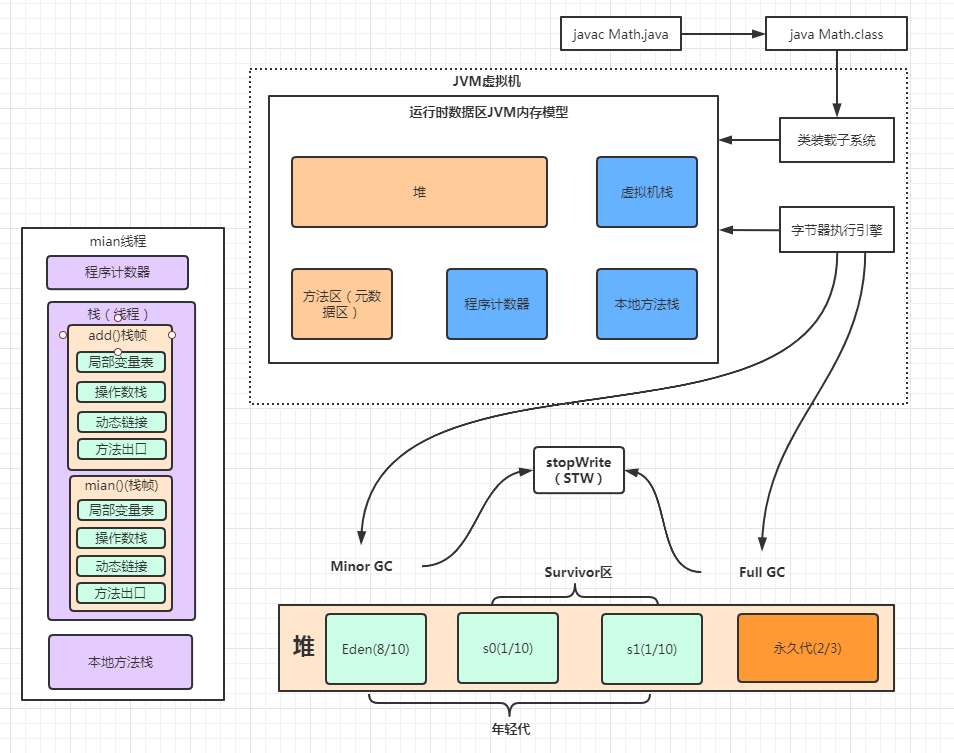
方法出口：存储调用外部方法后接着要执行代码的位置。

**计数器：(线程私有)**存储程序运行的位置。在多线程Cpu时间分片的情况下执行其他线后，再次执行能找到上次执行的位置接着执行。

**本地方法栈：(线程私有)：**执行本地方法的一块内存区域。本地方法就是用native修饰的方法，它会找到底层c语言类库对应的方法执行。

**方法区**（元数据区）**(线程共享)**jdk1.8用的是直接物理内存，是用来存储：静态变量、常量、类型信息。

**内存模型图：**



##### 垃圾回收器

垃圾回收器会回收Eden区和非空的survivor区，将标记为非垃圾对象的一次全部移动到survivor区并将这些对象的年龄加1，并将未标记垃圾对象全陪清理，这个过程会eden区中反复发生，当对象年龄达到15后，如果对象依然存活会将这些对象移到老年代中，（**大于survivor区50%的大对象直接移动到老年代**），当老年代要满了就会发生Full GC 会对整个堆全局回收，如果由于程序bug导致回不了，最后满了就会发生OOM 内存回收异常。

Stop The Word (**STW**) **停止应用程序的线程** 在GC垃圾回收器回收时发生，会停止应用线程。

1.可达性分析算法(Gc-Roots)Minor GC将“**Gc-Roots**”对象作为起点，从这些节点开始向下搜索引用的对象，找到的对象都标记为非垃圾对象，其余未标记的都是垃圾对象**GC Roots**根节点：**线程栈的本地变量、静态变量、本地方法栈的变量**等等。

即使在可达性分析算法中不可达的对象，也并非是“非死不可”，这时候它们暂时处于“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。

  第一次标记：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；

  第二次标记：第一次标记后接着会进行一次筛选，筛选的条件是此对象是否有必要执行finalize()方法。在finalize()方法中没有重新与引用链建立关联关系的，将被进行第二次标记。第二次标记成功的对象将真的会被回收，如果对象在finalize()方法中重新与引用链建立了关联关系，那么将会逃离本次回收，继续存活。

通过system.gc()通知finalize()并不是必须要执行的，它只能执行1次或者0次。Finalizer线程不保证一定执行finalize方法，因为此线程的优先级很低，获得CPU资源有限;而且这样会避免finalize执行缓慢或者发生死循环，从而导致整个GC奔溃

在Java语言中，可作为GC Roots的对象包括下面几种：

  a) 虚拟机栈中引用的对象（栈帧中的本地变量表）；

  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。

###### 2.引用计数算法

 引用计数是垃圾收集器中的早期策略。在这种方法中，堆中每个对象实例都有一个引用计数。当一个对象被创建时，就将该对象实例分配给一个变量，该变量计数设置为1。当任何其它变量被赋值为这个对象的引用时，计数加1（a = b,则b引用的对象实例的计数器+1），但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减1。任何引用计数器为0的对象实例可以被当作垃圾收集。当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减1。

**优点：**引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。

**缺点：**无法检测出循环引用。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为0。

###### 3. 方法区如何判断是否需要回收

 方法区主要回收的内容有：废弃常量和无用的类。对于废弃常量也可通过引用的可达性来判断，但是对于无用的类则需要同时满足下面3个条件：

1)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；

2)加载该类的ClassLoader已经被回收；

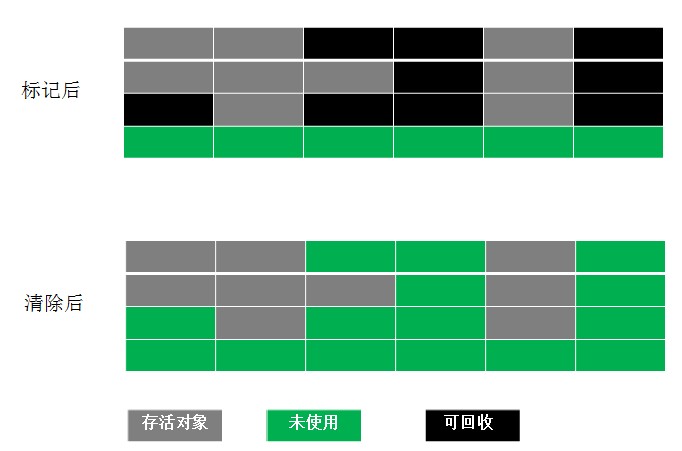
3)该类对应的java.lang.Class对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法;

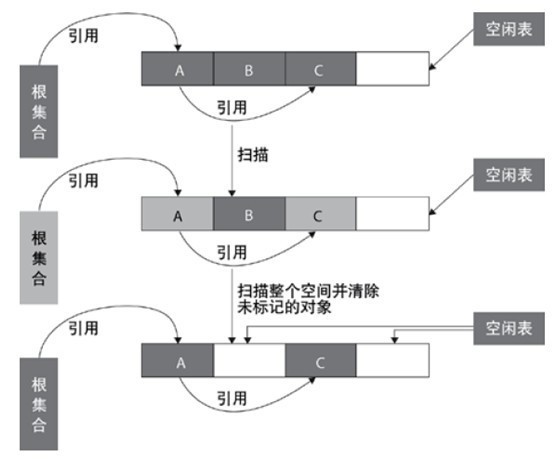
##### 常用的垃圾收集算法

###### 引用计数算法

###### 标记-清除算法（Mark-Sweep）

标记-清除算法分为两个阶段：标记阶段和清除阶段。标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。具体过程如下图所示：



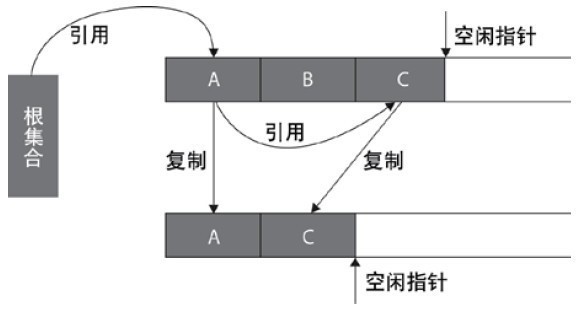


从图中可以很容易看出标记-清除算法实现起来比较容易，但是有一个比较严重的问题就是容易产生内存碎片，碎片太多可能会导致后续过程中需要为大对象分配空间时无法找到足够的空间而提前触发新的一次垃圾收集动作。

标记-清除算法采用从根集合（**GC Roots**）进行扫描，对存活的对象进行标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收。标记-清除算法不需要进行对象的移动，只需对不存活的对象进行处理，在存活对象比较多的情况下极为高效，但由于标记-清除算法直接回收不存活的对象，因此会造成内存碎片。

###### 复制算法(Copying)

为了解决Mark-Sweep算法的缺陷，Copying算法就被提了出来。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，然后再把已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。具体过程如下图所示：



这种算法虽然实现简单，运行高效且不容易产生内存碎片，但是却对内存空间的使用做出了高昂的代价，因为能够使用的内存缩减到原来的一半。

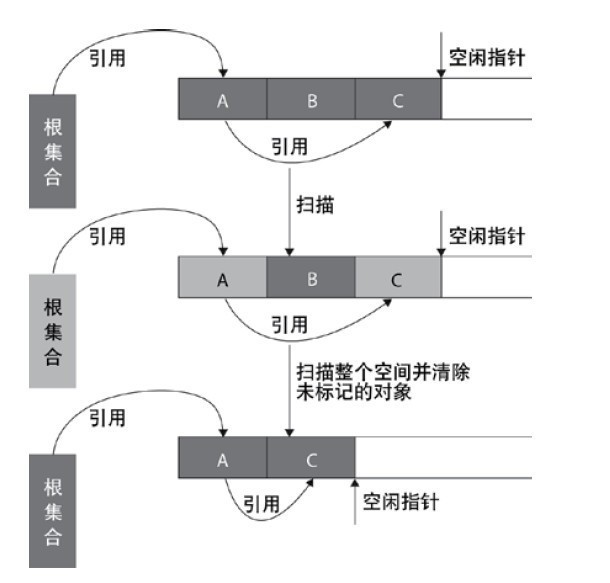
很显然，Copying算法的效率跟存活对象的数目多少有很大的关系，如果存活对象很多，那么Copying算法的效率将会大大降低。

复制算法的提出是为了克服句柄的开销和解决内存碎片的问题。它开始时把堆分成 一个对象 面和多个空闲面， 程序从对象面为对象分配空间，当对象满了，基于copying算法的垃圾 收集就从根集合（GC Roots）中扫描活动对象，并将每个 活动对象复制到空闲面(使得活动对象所占的内存之间没有空闲洞)，这样空闲面变成了对象面，原来的对象面变成了空闲面，程序会在新的对象面中分配内存。

###### 标记-整理算法(Mark-compact)（又称标记压缩）

###### 为了解决Copying算法的缺陷，充分利用内存空间，提出了Mark-Compact算法。该算法标记阶段和Mark-Sweep一样，但是在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动(记住是完成标记之后，先不清理，先移动再清理回收对象)，然后清理掉端边界以外的内存。

标记-整理算法采用标记-清除算法一样的方式进行对象的标记，但在清除时不同，在回收不存活的对象占用的空间后，会将所有的存活对象往左端空闲空间移动，并更新对应的指针。标记-整理算法是在标记-清除算法的基础上，又进行了对象的移动，因此成本更高，但是却解决了内存碎片的问题。具体流程见下图：

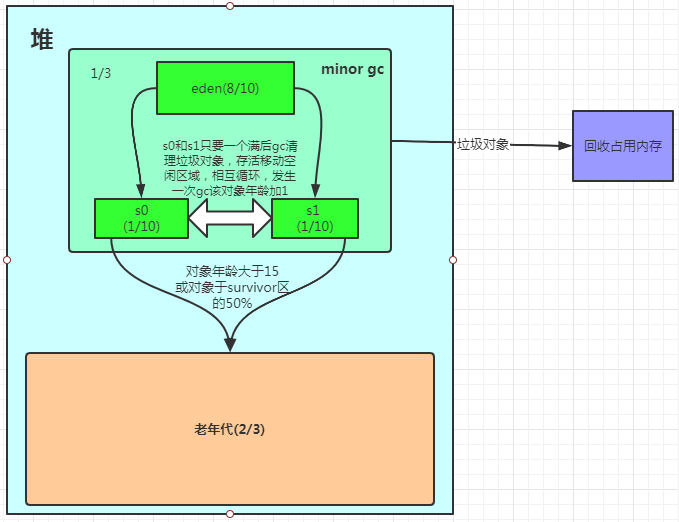


###### 分代收集算法 Generational Collection（分代收集）算法

分代收集算法是目前大部分JVM的垃圾收集器采用的算法。它的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。一般情况下将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），在堆区之外还有一个代就是永久代（Permanet Generation）。老年代的特点是每次垃圾收集时只有少量对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点采取最适合的收集算法。

目前大部分垃圾收集器对于**新生代都采取Copying算法**，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数较少，但是实际中并不是按照1：1的比例来划分新生代的空间的，**一般来说是将新生代划分为一块较大的Eden空间和两块较小的Survivor空间（一般为8:1:1），每次使用Eden空间和其中的一块Survivor空间，当进行回收时，将Eden和Survivor中还存活的对象复制到另一块Survivor空间中，然后清理掉Eden和刚才使用过的Survivor空间。**

而由于**老年代的特点是每次回收都只回收少量对象，一般使用的是Mark-Compact算法。**



5.1年轻代（Young Generation）的回收算法 (回收主要以Copying为主)

Minor GC 新生代垃圾回收器。

a) 所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。

b) 新生代内存按照8:1:1的比例分为一个eden区和两个survivor(survivor0,survivor1)区。一个Eden区，两个 Survivor区(一般而言)。大部分对象在Eden区中生成。回收时先将eden区存活对象复制到一个survivor0区，然后清空eden区，当这个survivor0区也存放满了时，则将eden区和survivor0区存活对象复制到另一个survivor1区，然后清空eden和这个survivor0区，此时survivor0区是空的，然后将survivor0区和survivor1区交换，即保持survivor1区为空(美团面试，问的太细，为啥保持survivor1为空，答案：为了让eden和survivor0 交换存活对象)， 如此往复。当Eden没有足够空间的时候就会 触发jvm发起一次Minor GC

c) 当survivor1区不足以存放 eden和survivor0的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次Full GC(Major GC)，也就是新生代、老年代都进行回收。

d) 新生代发生的GC也叫做Minor GC，MinorGC发生频率比较高(不一定等Eden区满了才触发)。

5.2年老代（Old Generation）的回收算法（回收主要以Mark-Compact为主）

Major GC 年老代垃圾回收器。

Full GC 整个堆垃圾回收器。

a) 在年轻代中经历了N次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

b) 内存比新生代也大很多(大概比例是1:2)，当老年代内存满时触发Major GC即Full GC，Full GC发生频率比较低，老年代对象存活时间比较长，存活率标记高。

##### 常见的垃圾收集器

###### Serial收集器（复制算法)

新生代单线程收集器，标记和清理都是单线程，优点是简单高效。是client级别默认的GC方式，可以通过-XX:+UseSerialGC来强制指定。

###### Serial Old收集器(标记-整理算法)

老年代单线程收集器，Serial收集器的老年代版本。

###### ParNew收集器(停止-复制算法)

新生代收集器，可以认为是Serial收集器的多线程版本,在多核CPU环境下有着比Serial更好的表现。

###### Parallel Scavenge收集器(停止-复制算法)

并行收集器，追求高吞吐量，高效利用CPU。吞吐量一般为99%， 吞吐量= 用户线程时间/(用户线程时间+GC线程时间)。适合后台应用等对交互相应要求不高的场景。是server级别默认采用的GC方式，可用-XX:+UseParallelGC来强制指定，用-XX:ParallelGCThreads=4来指定线程数。

一般情况下，当新对象生成，并且在Eden申请空间失败时，就会触发Scavenge GC，对Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区。然后整理Survivor的两个区。这种方式的GC是对年轻代的Eden区进行，不会影响到年老代。因为大部分对象都是从Eden区开始的，同时Eden区不会分配的很大，所以Eden区的GC会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使Eden去能尽快空闲出来。

###### Parallel Old收集器(停止-复制算法)

Parallel Scavenge收集器的老年代版本，并行收集器，吞吐量优先。

###### CMS(Concurrent Mark Sweep)收集器（标记-清理算法）

高并发、低停顿，追求最短GC回收停顿时间，cpu占用比较高，响应时间快，停顿时间短，多核cpu 追求高响应时间的选择。

###### G1(Garbage-First)收集器