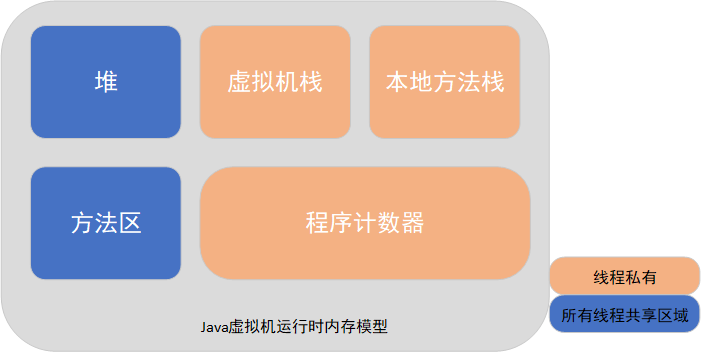
# Java内存区域



## 线程私有区域

**虚拟机栈**：Java虚拟机栈是线程私有的，生命周期与线程相同。虚拟机栈描述的是java方法执行的内存模型，与数据结构的栈结构相同，每个方法被执行的时候都会同时创建一个栈帧，并压入栈中**（这就是被称为虚拟机栈的原因）**，用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

***局部变量表***：存放变量，对于对象，存的是对象实例在堆中的指针

***操作数栈***：用于计算表达式

***动态链接***：

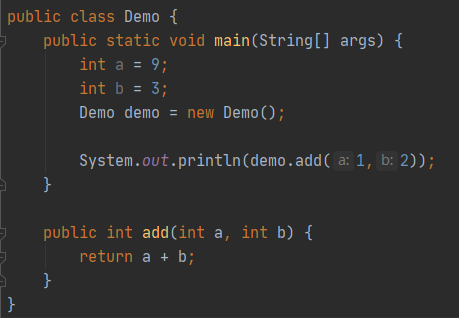
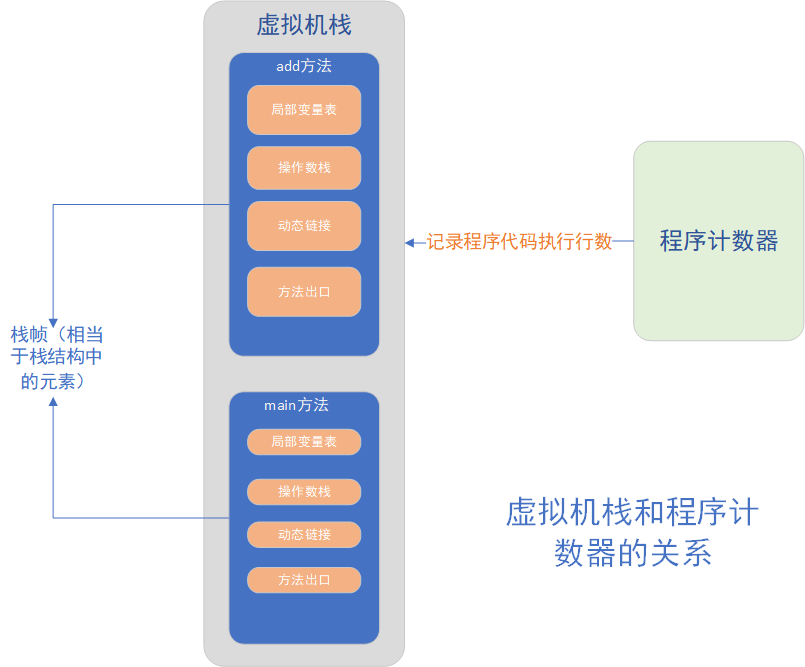
***方法出口***：如main调用add方法，方法出口中保存了main的执行位置。add方法计算执行完成后，返回main方法的某一行代码

**StackOverflowError异常**：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常。

**OutOfMemoryError异常**：如果虚拟机可以动态扩展，在扩展时无法申请到足够的内存，则抛出OutOfMemoryError异常。

**程序计数器：**程序计数器是一块较小的内存空间，作用是当前线程所执行的字节码的行号指示器。在任何时刻，一个处理器只会执行一条线程中的指令。为了线程切换后能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。

**本地方法栈：**本地方法栈与虚拟机栈类似，区别不过是虚拟机栈为虚拟机执行Java方法服务，而本地方法栈则为虚拟机使用到的Native方法服务。



## 线程共享区域

**Java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，此区域的唯一目的就是存放对象实例。

Java堆是垃圾收集器管理的主要区域，因此也被称为GC堆。从内存回收角度看，由于现在收集器基本都采用了分代收集算法，所以java堆中还可以细分为新生代和老年代，在细致一点可以分为Eden空间，From Survivor空间，To Survivor空间等

**方法区：**方法区与java堆一样，是线程共享的内存区域，用于存储已经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。

***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，用于存放编译器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中存放。

# 垃圾收集器和内存分配策略

问题：

哪些内存需要回收

什么时候回收

如何回收

程序计数器，虚拟机栈，本地方法栈3个区域随线程而生，随线程而灭；栈中的栈帧随着方法的进入和退出执行着出栈和入栈的操作。每一个栈帧中分配多少内存基本上是在类结构确定下来时就一直了。因此这几个区域的内存分配和回收都具备确定性。所以这几个区域内就不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟着回收了。

而java堆和方法区不一样，一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间才能知道会创建哪些对象，这部分内存的创建和回收都是动态的，垃圾收集器所关注的是这部分内存。

## 如何判断对象是否死了

**引用计数算法：**

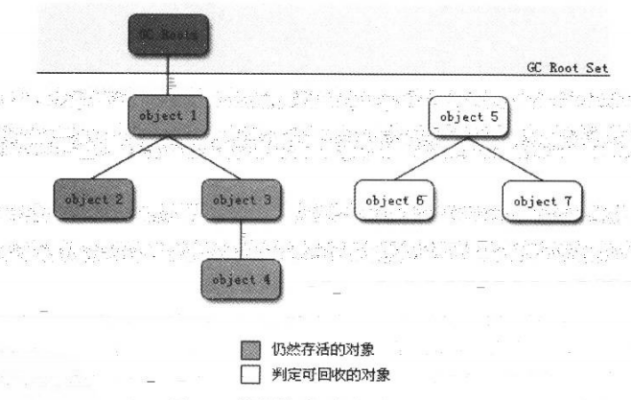
给对象添加一个引用计数器，每当有一个地方引用它时，计数器的值就加1；当引用失效时，计数器值就减1，任何时刻计数器为0的对象就是不可能再被引用的。

主流的java虚拟机都没有使用引用计数算法来管理内存，主要原因是很难解决对象之间相互循环引用的问题。

如：对象objA和objB都有字段instance，赋值令objA.instance=objB 以及 objB.instance=objA，除此之外，这两个对象再没有其他引用，实际这两个对象已经不可能再被访问，但是他们互相引用对方，导致他们的引用计数都不为0，无法通知GC收集器回收他们。

**可达性分析算法：**

在主流的商用程序语言的主流实现中，都是通过可达性分析来判定对象是否存活的。这个算法的基本思路是通过一些列的成为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径成为引用链，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。



在java语言中，可作为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（Native方法）引用的对象。

## 引用

在JDK1.2之前，Java中一个对象只有被引用和没有被引用两种状态。对于如何描述一些“食之无味，弃之可惜”的对象就显得无能为力。我们希望能描述这样一类对象：当内存空间还足够时，则能保存在内存之中，如果内存空间在进行垃圾收集后还是非常紧张，则可以抛弃这些对象。很多系统的缓存功能都符合这样的应用场景。

在JDK1.2之后，Java对引用的概念进行了扩充，将引用分为强引用，软引用，弱引用，虚引用4种，这4种引用强度一次逐渐减弱。

**强引用**：值在程序代码之中普遍存在的，类似“Object obj = new Object()”这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。

**软引用**：描述一些有用但是并非必需的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。

**弱引用**：被弱引用关联的对象只能生存到下次垃圾收集之前。

**虚引用**：

## 生存还是死亡

即使在可达性分析算法中不可达的对象，也并非是非死不可的，只是他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：

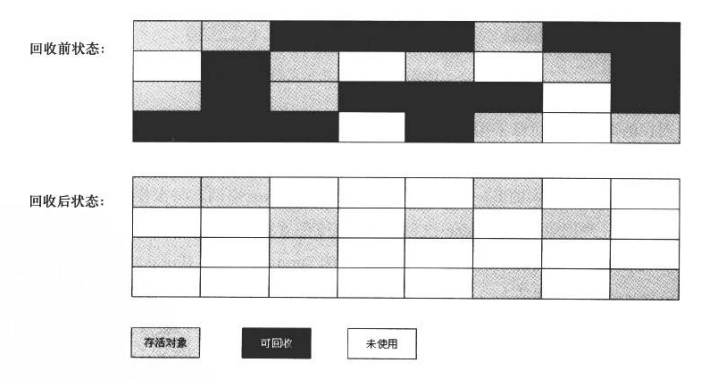
1. 如果对象在惊醒可达性分析后法相没有与GC Roots相连的引用链，那他将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否有必要执行finalize()方法。当对象没有覆盖finalize()方法，或者finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都视为没有必要执行。
2. 如果这个对象被判定为有必要执行finalize()方法，那么这个对象将被放置在一个叫做F-Queue队列之中，并在稍后由一个由虚拟机自动建立的，低优先级的Finalizer线程曲执行它。稍后GC将对F-Queue中的对象进行第二次小规模的标记，如果对象要在finalize()中成功拯救自己，只要重新与引用链上的任何一个对象建立关系即可。

## 垃圾收集算法

**标记-清除算法**

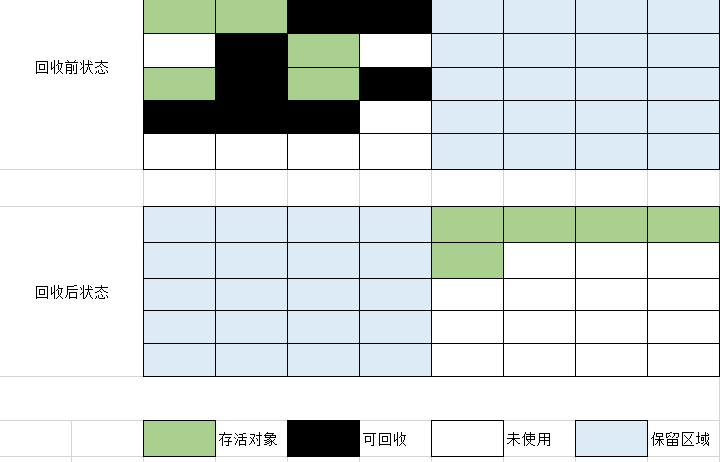
如同他的名字一样，算法分为标记和清除两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。后续的收集算法都是基于这种思路并对其不足进行改进得到的。

不足：效率问题：标记和清除两个过程的效率都不高。空间问题：清除后会产生大量比连续的内存碎片，空间碎片太多可能会导致以后在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出发另一次垃圾收集动作。



**复制算法**

为了解决效率问题，一种称为复制的手机算法出现了，他将可用内存按容量划分为大小相等的两块，每次值使用其中一块。当这一块内存用完了，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。但是这种算法的代价是将内存缩小为了原来的一般，代价太高了一点儿。



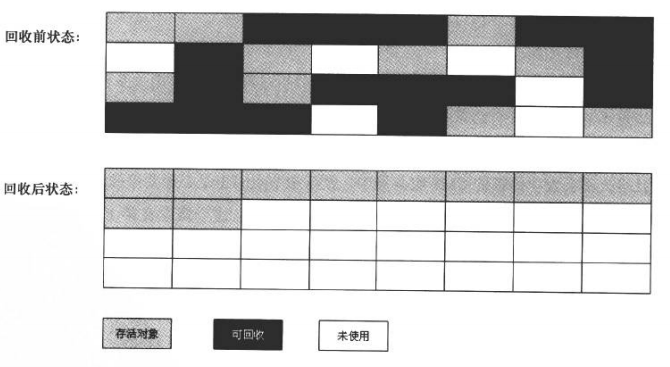
现在的商业虚拟机都采用这种收集算法来回收新生代，IBM公司的专门研究表明，新生代中的对象98%是朝生暮死的，所以并不需要按照1：1的比例来划分内存空间，而是将内存分为一块较大的Eden和两块较小的Survivor空间 ，每次使用Eden和其中一块Survivor。当回收时，将Eden和Survivor中还存活着的对象一次性地复制到另外一块Survivor空间上，最后清理掉Eden和刚才用过的Survivor空间。HotSpot虚拟机默认Eden和两个Survivor的大小比例是8：1：1，也就是只有10%的内存会浪费。

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（老年代）进行分配担保。

**标记-整理算法（适用于老年代）**

复制手机算法再对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端请款给，所以老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了标记-整理算法，标记过程仍与标记-清除算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉端边界意外的内存。



## 分代收集算法（重要）

当前商业虚拟机的垃圾收集都是采用分代收集算法，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。

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，没有额外空间对他进行分配担保，就必须使用标记-整理或标记-清理算法来进行回收。

## 垃圾收集器

Serial收集器，parnew收集器，parallel scavenge收集器等

**CMS收集器（Concurrent Mark Sweep）**

CMS收集器是一种以获得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。

CMS收集器是基于“标记-清除”算法实现的，收集过程分为4个步骤，包括：

* 初始标记
* 并发标记
* 重新标记
* 并发清除

其中，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，而重新标记阶段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，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记时间短。

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。

**G1收集器**

G1收集器是当今收集器技术发展的最前沿成果之一，可能会替换掉JDK1.5中发布的CMS收集器。

## 内存分配与回收策略

**====================都是抄的=====================**

**-Xms20M**

表示设置JVM启动内存的最小值为20M，必须以M为单位

**-Xmx20M**

表示设置JVM启动内存的最大值为20M，必须以M为单位。将-Xmx和-Xms设置为一样可以避免JVM内存自动扩展。大的项目-Xmx和-Xms一般都要设置到10G、20G甚至还要高

**-verbose:gc**

表示输出虚拟机中GC的详细情况

**-Xss128k**

表示可以设置虚拟机栈的大小为128k

**-Xoss128k**

表示设置本地方法栈的大小为128k。不过HotSpot并不区分虚拟机栈和本地方法栈，因此对于HotSpot来说这个参数是无效的

**-XX:PermSize=10M**

表示JVM初始分配的永久代的容量，必须以M为单位

**-XX:MaxPermSize=10M**

表示JVM允许分配的永久代的最大容量，必须以M为单位，大部分情况下这个参数默认为64M

**-Xnoclassgc**

表示关闭JVM对类的垃圾回收

**-XX:+TraceClassLoading**

表示查看类的加载信息

**-XX:+TraceClassUnLoading**

表示查看类的卸载信息

**-XX:NewRatio=4**

表示设置年轻代：老年代的大小比值为1：4，这意味着年轻代占整个堆的1/5

**-XX:SurvivorRatio=8**

表示设置2个Survivor区：1个Eden区的大小比值为2:8，这意味着Survivor区占整个年轻代的1/5，这个参数默认为8

**-Xmn20M**

表示设置年轻代的大小为20M

**-XX:+HeapDumpOnOutOfMemoryError**

表示可以让虚拟机在出现内存溢出异常时Dump出当前的堆内存转储快照

**-XX:+UseG1GC**

表示让JVM使用G1垃圾收集器

**-XX:+PrintGCDetails**

表示在控制台上打印出GC具体细节

**-XX:+PrintGC**

表示在控制台上打印出GC信息

**-XX:PretenureSizeThreshold=3145728**

表示对象大于3145728（3M）时直接进入老年代分配，这里只能以字节作为单位

**-XX:MaxTenuringThreshold=1**

表示对象年龄大于1，自动进入老年代

**-XX:CompileThreshold=1000**

表示一个方法被调用1000次之后，会被认为是热点代码，并触发即时编译

**-XX:+PrintHeapAtGC**

表示可以看到每次GC前后堆内存布局

**-XX:+PrintTLAB**

表示可以看到TLAB的使用情况