# Java虚拟机：垃圾收集器与内存分配策略

# 概述

说起垃圾收集器（Garbage Collection,GC），大部分人都把这项技术当做Java的伴生产物。实际上GC的历史远比Java久远，1960年诞生于MIT的Lisp是第一门真正使用内存动态分配和垃圾收集技术的语言。当Lisp还在胚胎时期时，人们就在思考GC需要完成的3件事情:

* 那些内存需要回收？
* 什么时候回收内存？
* 如何回收？

# 垃圾收集器关注那些数据区域

* 程序计数器
* 虚拟机栈
* 本地方法栈
* 堆
* 方法区

程序计数器、Java虚拟机栈、本地方法栈这3个区域都是随线程而生，随线程而灭；栈中的栈帧随着方法的进入和退出而有条不紊地执行着出栈和入栈操作。每个栈帧分配多少内存基本上是在类结构确定下来的时候就已知的，因此这几个区域的内存分配和回收都具备确定性，在这几个区域内就不需要过多考虑回收的问题，以为方法结束或者线程结束时，内存自然就跟随着回收了。而Java堆区和方法区则不一样，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那些对象，这部分内存的分配和回收都是动态的，垃圾回收器关注的是这部分内存。

先讲述Java堆区中的对象回收。

# 判断对象是否存活

* 引用计数：通过判断对象被引用的次数(为0，则表示不可被使用)，但这很难解决对象相互循环引用的问题。
* 根搜索算法：即采用有向图的方式，判断从GC Roots到某个对象是否可达。

img

img

GC-可达性分析.png

什么样的对象能作为GC的Root节点呢？

* 虚拟机栈中局部变量引用的对象
* 类静态属性引用的对象
* 常量引用的对象
* JNI中引用的对象

# 对象的回收

要宣告一个对象死亡，只少要经历两次标记过程：如果对象在进行可达行分析后发现没有与GC Roots相链接的引用链，那它将会被进行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”。

img

img

对象回收.png

# 对象的引用类型

说起对象的回收我们就不能不说对象的引用了，因为无论【引用计数法】判断对象的引用数量，或者【根搜索算法】判断对象的应用链是否可达，判定对象是否存活都与引用有关。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（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）与虚引用（Phantom Reference）四种，这四中引用程序依次逐渐减弱。

* 强引用就是指在程序代码之中普遍存在的，类似"Object obj = new Object()"这类的引用，只要有强引用还存在，垃圾收集器永远不会回收掉被引用的对象。
* 软引用是用来描述一些还有用但并非必需的对象。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二次回收。如果这次回收还没有足够的内存，才会抛出内存异常。在JDK1.2之后，提供了SoftReference类来实现软引用。
* 弱引用也是用来描述非必须对象的，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存早下一次垃圾收集发生之前，当来及收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在JDK1.2之后，提供了WeakReference类来实现弱引用。
* 虚引用也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来获得一个对象实例，为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。在JDK1.2之后，提供了PhantomReference类来实现虚引用。

# 方法区的回收

方法区或者是HotSpot虚拟机的永久代的垃圾回收主要回收的内容有两部分：废弃的常量和无用的类。

废弃的常量回收和Java堆中的对象回收时类似的。

判断一个类是否是【无用的类】却比判断一个对象是否被可以被回收苛刻的多，该类需要满足同时满足一下三个条件：

* 改类的所有实力都以及被回收，也就是说Java堆中存在改类的任何实力；
* 加载该类的ClassLoader都以及被回收。
* 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。

# 垃圾回收算法

* 标记-清除算法（Mark-Sweep）
* 复制算法（Copying）
* 标记-整理算法（Mark-Compact）

## 标记-清除算法（Mark-Sweep）

【标记-清除】是最基础的收集算法，算法分为“标记”和“清除”两个阶段，首先标记处所有需要回收的对象，在标记完成后统一回收所被标记的对象，它的标记过程就是上边讲的对象的回收中的标记。 特点：

* 标记和清除效率都不高
* 标记清除后会产生大量内存碎片

img

img

标记-清除

## 复制算法（Copying）

为了解决效率问题，一种称为“复制”的收集算法出现了，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，就将其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一次清理掉。

* 不会产生碎片
* 运行效率高
* 内存缩小了一半

img

img

复制算法

## 标记-整理算法（Mark Compact）

标记-整理算法是介于【标记-清除】和【复制】之间的收集算法，标记过程任然与【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，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

img

img

标记-整理

## 分代收集算法（Generational Collection）

当前商业虚拟机的垃圾收集都是采用“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。在新生代中，每次来及收集时都发现有大批对象死去，只有少量存活，那就选用复制算犯法，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，没有额外控件对它进行分配担保，就必须使用“标记-清理”或者“标记-整理”算法来进行回收。

# 内存分配策略

img

img

Java内存分配

这里所说的内存分配，主要至的是在堆上的分配，一半的，对象的内存分配都是在堆上进行，但现代技术页支持将对象拆程标量类型（标量类型即原子类型，表示单个值，可以是基本类型或String类型），然后在栈上分配，在栈上分配很少见，我们这里不考虑。

Java内存分配和回收的机制概括的说，就是分代分配，分代回收。对象根据存活的时间被分为：年轻代（Young Generation）、老年代（Old Generation）、永久代（Permanent Generation，也就是方法区）。

* 年轻代（Young Generation）：对象被创建时，内存的分配首先发生在年轻代（大对象可以直接 被创建在年老代），大部分的对象在创建后很快就不再使用，因此很快变得不可达，于是被年轻代的GC机制清理掉（IBM的研究表明，98%的对象都是很快消 亡的），这个GC机制被称为Minor GC或叫Young GC。注意，Minor GC并不代表年轻代内存不足，它事实上只表示在Eden区上的GC。

Minor GC:采用复制算法（Copying）

* 年老代（Old Generation）：对象如果在年轻代存活了足够长的时间而没有被清理掉（即在几次 Young GC后存活了下来），则会被复制到年老代，年老代的空间一般比年轻代大，能存放更多的对象，在年老代上发生的GC次数也比年轻代少。当年老代内存不足时， 将执行Major GC，也叫 Full GC。

Full GC:标记-整理算法(Mark-Compact)

img

img

GC.png

年轻代上的内存分配是这样的，年轻代可以分为3个区域：Eden区（伊甸园，亚当和夏娃偷吃禁果生娃娃的地方，用来表示内存首次分配的区域，再 贴切不过）和两个存活区（Survivor 0 、Survivor 1）。

绝大多数刚创建的对象会被分配在Eden区，其中的大多数对象很快就会消亡。Eden区是连续的内存空间，因此在其上分配内存极快；

当Eden区满的时候，执行Minor GC，将消亡的对象清理掉，并将剩余的对象复制到一个存活区Survivor0（此时，Survivor1是空白的，两个Survivor总有一个是空白的）；

此后，每次Eden区满了，就执行一次Minor GC，并将剩余的对象都添加到Survivor0；

当Survivor0也满的时候，将其中仍然活着的对象直接复制到Survivor1，以后Eden区执行Minor GC后，就将剩余的对象添加Survivor1（此时，Survivor0是空白的）。

当两个存活区切换了几次（HotSpot虚拟机默认15次，用-XX:MaxTenuringThreshold控制，大于该值进入老年代）之后，仍然存活的对象（其实只有一小部分，比如，我们自己定义的对象），将被复制到老年代。

## 对象优先在Eden区分配

## 大对象直接进入老年代

## 长期存活的对象将进入老年代

第一次进入Survivor区域的时候对象年龄设置为1，对象在Survivor区域中每“熬过”一次MinorGC，年龄增加一岁，当它的年龄增加到一定程度（默认为15岁），将会被晋升到老年代中。

## 动态对象年龄判断

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直接进入老年代，无须等到MaxTenuringThreshold中要求的年龄。

## 空间分配担保

在发生MinorGC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手代所有对象总空间，如果这个条件成立，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的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，将会尝试着一次Minor GC，尽管这次Minor GC是有风险的；如果小于，或者HandlePromotionFailure设置不允许冒险（冒险：当出现大量对象在Minor GC后任然存活的情况，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），那这时改为进行一次Full GC。