# 1.垃圾回收机制

## 判断哪些内存需要回收

1. 判断对象是否还存活，有两种方法

引用计数法：给对象添加一个引用计数器，每增加一个该对象得引用，一直到该对象的引用为0时才回收它。当存在两个对象互相引用时，引用计数法可能导致这两个对象永远无法回收，所以java虚拟机采用的并不是引用计数法。

可达性分析法：从任意GC roots节点开始，依此遍历，若是有引用链相连，就是可达的，若是没有引用链相连，就是不可达的。不可达的节点就是可回收的对象。可以做为GCroots的对象包括，虚拟机栈中引用的对象，方法区中静态属性类引用的对象，本地方法栈中引用的对象。但是需要注意的是不可达对象也不是非死不可的，因为宣判一个对象不可用，至少要经历两次标记过程：第一次经过可达性分析之后，如果发现不可达就会被标记和筛选，筛选的条件是看你是否需要执行finalize()方法，如果没有覆盖这个方法或者已经调用过了，就不用执行，就可以回收掉了。如果需要执行，就把这个对象加入一个F-queue队列中，稍后由虚拟机的一个专门的线程执行，如果对象再finalize（）方法中完成了自救就不会被回收，如果没有的话就会被回收。第二次标记的时候，如果这些对象还在即将回收的集合里，就真的会被回收。

1. 判断方法区中的常量和类是否可回收。

废弃常量：没有任何对象引用常量池中的常量，也没有任何字面量是这个常量，这个常量就是废弃常量，可被回收。

类的回收：该类的所有对象都已经被回收。加载该类的ClassLoader被回收。也无法在任何地方通过反射访问该类。满足着三个条件，这是可回收类。

## 垃圾收集算法‘

1. 标记-消除算法

这种算法是指使用一半的内存区域，把内存区域标记为存活对象，可回收对象和未使用区域。Gc操作发生时就把内存区域标记为可回收的给回收掉。这种算法会产生很多的碎片，可能导致需要分配一大块存储空间的时候无法分配连续存储空间，而不得不提前促发一次gc操作。

1. 复制算法

复制算法是队标记消除算法的改进，它只使用了一半的存储空间，当gc操作发生时，把存活对象全部复制到未使用的那一半区域，然后把正在使用的这一半区域清除掉。这样虽然不会产生内存碎片但是只有效利用了一半的存储区域。现在的商业虚拟机的新生代都是采用的这种算法，但是不是采用1:1的比例，而是Eden空间和survivor空间8：1的比例，有两个survivor空间和一个Eden空间，对象一般都是在Eden空间创建，每次把存活的对象移到未使用的survivor空间，如果survivor空间不够，就移到老年代空间，这就是空间担保，后面会介绍。所以这种算法的空间利用率时90%。

1. 标记-整理算法

标记整理算法也是对标记-整理算法的改进，它解决了复制算法空间利用率的问题。每次gc操作是把存活对象放到内存区域的一端，然后清除这个边界以外的其它区域。老年代使用的就是标记整理算法。

1. 分代收集算法

当代商业虚拟机采用的都是分代收集算法，就是把堆划分为新生代和老年代，新生代每次都有大批的对象死去，所以采用复制算法，有老年代的空间作为担保，而老年代对象存活率高，也没有额外空间做担保，必须使用标记-清理或者标记整理算法。

## 内存分配与回收策略

1. 内存分配

对象优先分配在新生代的Eden区域上，如果开启了本地线程分配缓冲，将按线程优先分配在TABLE上，TABLE是为了解决指针移动时线程不安全的问题，减少了同步锁的使用。大对象直接进入老年代。长期存活的对象进入老年代。

1. 动态年龄判定

除了年龄达到设定的值外，相同年龄的所有对象的大小，超过了survivor空间的一半的时候，也会让超过该年龄的对象进入老年代。

1. 空间分配担保

当进行Minor GC之前，会检查老年代的最大连续空间是否大于新生代所有对象的空间，如果不大于，就要检查HandlePromotionFailue设置值时否允许担保失败，如果允许，则继续检查老年代的最大连续空间是否大于历代升入老年代的对象的平均值，若大于则可以冒险一次，失败了再进行Full GC。

Full GC比Minor GC要慢10倍以上，所以要避免频繁Full GC.

## 垃圾收集器

1. Serial收集器

新生代收集器，采用单线程收集器，采用复制算法，也就是收集时会把其它的用户线程都停止掉，单线程效率高。

1. Parnew收集器

新生代收集器，与Serial收集器的区别就是，收集时可以多线程同时收集，但是也要停止用户线程。

1. Parallel scavenge收集器

新生代收集齐，也是多线程的，复制算法收集器，但是它关注的是吞吐量。

1. Serial Old收集器

老年代收集器，单线程，采用标记-整理算法。

1. Parallel old收集器

老年代收集器，多线程，标记-整理，吞吐量优先

1. CMS收集器

CMS收集器以最短停顿时间为目标的一个收集器，使用标记-清理算法它分为四步：

第一步：初始化标记，这一步是会停止所有用户线程的，但是只标记和GCroots直接相连的节点，所以时间短，就是停顿时间短。

第二步：并发标记，这一步是不用停止所有用户线程的，标记所有的的对象，但是因为用户线程再运行，可能会改变原来的对象状态，所以有要进行第二次标记。

第三步：重新标记，这一步是会停止所有用户线程的，这一步是重新标记在并发标记过程中，状态改变的对象，时间会比并发标记短很多，所以停顿时间短。

第四步：并发清理，这一步不会停止用户线程

CMS收集器的缺点：

1. 对CPU资源敏感，占用了一部分线程，可能会导致程序变慢。
2. 因为是并发清理，所以无法清理在并发清理过程中产生的浮动垃圾。
3. 使用的标记-清理算法，可能会产生内存碎片。
4. G1收集器

面向服务端应用的收集器，标记整理算法，可预测的停顿，并行和并发，可以使用多个CPU来缩短停顿时间。分为如下四步：

1. 初始标记
2. 并发标记
3. 最终标记
4. 筛选回收

# 2.如何在垃圾回收中存活一次

1. 要判断一个对象死亡要经过两次标记，如果可达性分析之后，发现没有被GCroots链相连，这时候会进行第一次标记。
2. 再判断是否需要执行finalize()方法，如果没有重写，或者已经执行过，就不需要再执行，否则就是要执行。如果要执行的话就会把这个对象，加入到一个叫做F-queue的队列中去，稍后会有一个由虚拟机自动创建的，低优先级的Finalizer对象去执行这些对象的fibalize方法。
3. 执行finalize方法的时候，如果一个对象再这个方法里把自己与引用链上的任一对象关联，例如把自己（this）赋值某个类变量或者对象的成员变量，就可以存活一次，不过也仅一次，因为每个对象的finalize()方法只会执行一次。

Gcroots对象包括一下几种：

1. 虚拟机栈的栈帧中的局部变量表中引用的对象。
2. 方法区内的静态类变量引用的对象。
3. 方法区内的常量变量引用的对象。
4. 本地方法区内本地方法引用的对象。

# 3.java内存区域和内存溢出

为什么要了解java内存，因为程序要把内存的分配和回收都交给了java虚拟机，所以，如果不了解java虚拟机是如何是内存的，当出现问题是，排除错误将是一件很艰难的错误。

## java运行时区域

1. 程序计数器

程序计数器时一块较小的内存，是程序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这个计数器来选取下一条要执行的指令。为了线程切换后能够恢复到正常位置，每个线程都有一个单独的程序计数器。各个线程之间的程序计数器时互不干扰的，运行java方法时，程序计数器存放的就是正在执行的字节码的地址。运行本地方法时为空。

(2) 虚拟机栈（VM stack）

与程序计数器一样，java虚拟机栈也是线程私有的，它的生命周期和线程同步。它描述的java方法的内存模型，每个java方法执行的时候都会创建一个栈帧，这个栈帧用于存储局部变量表，操作数栈，方法出口等信息，每个方法在执行到完成的过程就是一个栈帧在虚拟机栈种进栈到出栈的过程。

其中需要特殊注意的时栈帧中的局部变量表，它存储了基本类型变量和引用。double和long占用了两个局部变量空间，其它只占一个，而且方法执行过程中。

当线程请求的虚拟机栈深度超出了允许的最大深度，就会报StackOverflowError异常。有动态扩展的功能，自动扩展时无法申请到足够的内存，就会报OutOfmemoryError异常。

（3）本地方法栈

本地方法栈和虚拟机栈类似，只不过本地方法栈时针对本地方法的。

（4）java堆

Java堆时所有线程共享的一块区域，也是大部分java应用最大的一块程序，java中所有的堆和数组都时在这上面分配的，gc操作也基本上都发生在这个区域。所以也成为gc堆。当内存没有完成实例分配，也无法扩展的时候，就会报OutOfmemoryError异常。新生代和老生待也是在堆中分的。

（5）方法区

方法区是存放虚拟机加载的类信息，常量，静态变量，即时编译编译后的代码等。它hotSpot把GC分代收集管理扩展自方法去，用java自动内存管理来管理。方法区无法满足内存分配的时候也会报出OutOfmemoryError异常。

（6）运行时常量池

运行时常量池时方法区的一部分，class文件中的常量池的信息在类加载以后会被放到方法区的运行时常量池中。而且这个常量池时可以动态加载的。当然方法区的东西，肯定在内存不足时报OutOfmemoryError异常.

## java对象在内存的创建，布局和定位

1. java对象的创建

当虚拟机遇到一条new指令的时候，首先会去常量池中看能否定位到一个符号引用，并检查这个类是否被加载，解析和初始化过，如果没有就要执行相应的加载过程。加载之后就是为对象分配内存，有两种分配方法，如果堆是齐整的就会采用“指针碰撞”法，如果不齐整就会采用“空闲列表”法。而是否齐整又是由所采用的垃圾收集过程时候带有功能决定。Serial，ParNew等带压缩，CMS不带。

因为对象创建时很频繁的行为所以还要考虑修改指针时的线程安全问题。有一种本地线程缓冲方案比较不错。

1. java对象的内存布局

对象在内存中存布局可以分为3块区域：对象头，实例数据和对齐填充。

对象头分为两部分信息，一部分时自身运行信息，如hash码等，一部分时类型指针，指向对应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。

实例数据是真正存储有效信息，如对象的字段内容，其存储顺序会受虚拟机分配策略和java中源码定义的顺序影响。

第三部分对齐填充不是必然存在的。

1. 对象的访问定位

在java虚拟机栈的栈帧的局部变量表中，只存储了一个引用，要根据这个引用去定位对象在堆中的位置。目前主流的有两种方法，一种是句柄定位，一种是直接指针。

句柄定位就是java虚拟机会在堆中划分出一个句柄池，存储对象在堆中的地址和对象对应的类的元数据在方法区中存放的地址。栈帧中存放对象地址在句柄池中的位置。

直接指针就是在栈帧中存放对象在堆中的地址，而堆中除了存放实例信息之外还存放了类型信息在方法区中的存放地址。

句柄的优势在于稳定，对象修改也只用修改句柄池的信息，而栈帧中 的信息不用修改。

直接指针的优势在于访问速度快。

## 内存溢出

内存溢出包括java堆溢出，虚拟机和本地方法栈溢出，方法区和运行时常量池溢出以及本机直接内存溢出。