**垃圾收集GC**

（1）程序计数器、虚拟机栈、本地方法栈3个区域的内存分配可以认为在编译期可知，因此这三个区域的内存分配和回收都具备确定性；

（2）Java堆和方法区，只有在程序处于运行期间才能知道会创建哪些对象，这部分内存的分配和回收都是动态。垃圾收集器重点关注这部分内存。

回收堆区：

垃圾收集器对堆进行回收时：判断哪些对象还“存活”着，哪些对象已“死去”（即不可能再被使用到的对象）

**×**（1）引用计数算法：为每个对象添加一个引用计数器，每当有一个地方引用它时，计数器值+1；当引用失效时，计数器值-1。任何时刻计数器为0的对象就是不可能再被使用的。但是由于难以解决对象之间相互循环引用的问题，因此没有被JVM采用。

**√**（2）**可达性分析算法**：一系列“GC Roots”对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain）。

当一个对象到GC Roots没有任何引用链相连（图论称GC Roots到这个对象不可达）时，将会判定为**可回收对象**。这时候这些可回收对象暂时处于“缓刑”期，并被第一次标记，

接下来进行一次筛选：判断该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，则虚拟机将这两种情况视为“没有必要执行”。其他情况为“有必要执行finalize()方法”，那么这个对象会被放置到F-Queue的队列中，之后会由一个虚拟机自动建立的、低优先级的Finalizer线程去执行它。这里所谓的“执行”是指虚拟机会触发这个方法，但不保证会等待它运行结束。finalize()方法是对象逃脱被回收的最后一次机会，之后GC会对F-Queue中的对象进行二次标记，如果对象在finalize()方法成功拯救自己——重新与引用链上的任何一个对象建立关联即可，如把自己(this)赋值给某一个类变量或者对象的成员变量，那么二次标记时会被移出回收集合。如果对象没有逃脱，则基本上会被回收掉。

可作为GC Roots的对象包括：

1、虚拟机栈（栈帧中的本地变量表）中引用的对象；

2、方法区中类静态属性引用的对象、

3、方法区中常量引用的对象

4、本地方法栈中JNI（Native方法）引用的对象。

引用分类：

（1）强引用

Object obj = new Object();

只要强引用还存在，垃圾收集器就永远不会回收被引用对象。

（2）软引用

SoftReference类。在系统将发生内存溢出异常时，将会对软引用对象列入回收范围进行第二次回收，如果回收后还没有足够内存空闲，才会抛出内存溢出异常。

（3）弱引用

WeakReference类。弱引用对象只能生存到下一次垃圾收集发生之前。当垃圾收集器工作时，无论内存是否足够都一定回收掉弱引用对象。

（4）虚引用

PhantomReference类。一个对象是否有虚引用，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为对象设置虚引用的唯一目的是为了在当对象被收集器回收时收到一个系统通知。

回收方法区

JVM规范中不要求虚拟机在方法区实现垃圾回收，并且在方法区垃圾收集性价比比较低。

永久代垃圾收集主要有两部分内容：（1）废弃常量（2）无用的类

回收废弃常量：类似于回收java堆对象。比如以常量池中的字面量回收为例。假如一个字符串“abc”已经进入常量池中，但是当前系统没有一个String对象引用常量池的“abc”常量，则如果此时方法区进行内存回收，就会将“abc”清理出常量池。常量池中的其他类/接口、方法、字段的符号引用也与此类似。

回收无用类：

判定无用类需同时满足3个条件：

（1）该类所有实例都已经被回收，也就是Java堆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，无法再任何地方通过反射访问该类的方法

虚拟机可以对满足上述3个条件的无用类进行回收，注意这里是可以，并不是像回收堆一样，对象不使用了就一定回收掉。是否对类进行回收，由虚拟机参数控制。

垃圾收集算法

（1）标记-清除算法（最基础、效率不高，大量不连续的空间碎片太多，较大对象无法找到足够内存而再次触发垃圾收集动作）

标记阶段：二次标记得到回收对象集合。

清除阶段：回收对象

（2）复制算法

思路：将内存按容量划分为大小相等的两块，每次只使用其中一块，直到该块内存用完了，就将还存活的对象复制到另外一块上面，再把该块内存已使用的空间一次清理掉，这样使得每次都是对整个半区进行内存回收。内存分配时也不用考虑到内存碎片等问题，只要移动堆顶指针，按顺序分配内存即可。代价是将内存缩小为原来一半。

实际应用：将新生代内存分成一块较大的Eden空间和两块较小的Survivor空间。每次使用Eden和其中一块Survivor空间。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Survivor的大小比例为8:1，也就是每次新生代中可用内存空间为整个新生代容量的90%。只有10%的内存会被“浪费”。**如果另一块Survivor没有足够空间存放上一次新生代收集下来的存活对象时**，**无法存放的存活对象将直接通过分配担保机制进入老年代**。但是如果有足够空间能够分配给其中一些存活对象（按对象创建顺序与剩余的Survivor空间比较大小，小于则放入），也就是这些存活对象能够熬过一次GC，则年龄+1。当达到某个年龄（默认15），则下次GC移到老年代中。

（3）标记-整理算法

针对老年代的特点而提出的算法。标记过程同“标记-清除”算法一样。但后续不是直接对回收对象进行清理，而是让存活对象都向一端移动，然后直接清理掉除了端边界以外的内存。

（4）分代收集算法

当代商业虚拟机垃圾收集都采用“分代收集”算法，根据对象存活周期的不同对内存进行划分。根据各个年代特点采用最适当的收集算法。

Java堆 = 新生代（Eden + Survivor + Survivor） + 老年代

新生代：复制算法

老年代：标记-清理/标记-整理

HotSpot的算法实现

（1）枚举根节点（GC Roots）

准确式GC：当执行系统停顿下来后，使用OopMap数据结构来得知哪些地方存放着对象引用。当类加载完成时，就会计算出对象内什么偏移量上是什么类型的数据；在JIT编译过程中，也会在**特定的位置（安全点）**中记录下栈和寄存器的哪些位置存放着引用，这样GC扫描时就可以直接得知这些对象引用信息。

（2）安全点

程序执行时，必须到达安全点才能暂停开始GC。

安全点的选择是以程序“是否具有让程序长时间执行的特征”为标准进行选定的。如：方法调用、循环跳转、异常跳转等。具有这些功能的指令才会产生安全点。

GC即将发生时，所有线程应“跑”到最近的安全点进行停顿。有“抢先试中断”和**“主动式中断”**两种方式。

主动式中断：当GC需要中断线程时，仅设置一个标志，让每个线程执行时自己主动去轮询这个标志，发现中断标志为真时就将自己中断挂起。轮询标志的指令位置和安全点是重合的，另外再加上创建对象需要分配内存的地方。

（3）安全区域

安全点只能解决程序执行时进入GC的问题。当程序“不执行”时，即处于Sleep或者Blocked状态时，此时线程无法响应JCM中断请求，去“跑”到安全点中断挂起。这里需要引入安全区。安全区域指一段代码片段中，引用关系不会发生变化。在这个区域中的任意地方开始GC都是安全的。当线程执行到安全区代码时，首先标识自己已经进入安全区，所以当这段时间里JVM要发起GC时，就不用管已经处于安全区的线程。当线程要离开安全区时，必须检查系统是否完成了根节点枚举（或整个GC过程），如果完成了，线程就继续执行，否则必须等待直到收到可以离开安全区的信号为止。

垃圾收集器

Serial收集器、ParNew收集器、Parallel Scavenge收集器、Serial Old收集器、Parallel Old收集器**、**CMS收集器、G1收集器等

G1：并行与并发、分代收集、空间整合（“标记-整理”算法 + 复制算法）、可预测的停顿

理解GC日志

每个收集器的日志格式都可以不一样，但有一定的共性：

33.125: [GC [DefNew: 3324K->152K(3712K), 0.0025925 secs] 3324K->152(11904K), 0.0030680 secs]

100.667: [Full GC [Tenured: 0K->210K(10240K), 0.0149142 secs] 4603K->210K(19456K), [Perm: 2999K->2999K(21248K)], 0.0150007 secs] [Times: user=0.01 sys=0.00, real=0.02 secs ]

33.125、100.667：GC发生时间，值为从JVM启动以来经过的秒数

[GC、[Full GC：本次垃圾收集的停顿类型，Full说明这次GC是发生了Stop-The-World的，这一般是由于出现了分配担保失败的问题，才会导致STW。如果是调用System.gc()方法所触发的收集，那么在这里将显示“[Full GC(System)”。

[DefNew、[Tenured、[Perm表示GC发生的区域（如新生代、老年代等），这里显示的区域名称与使用的GC收集器密切相关。

DefNew：Default New generation，Serial收集器中的新生代名

ParNew：Parallel New Generation，ParNew收集器中的新生代名

PSYoungGen：Parallel Scavenge New Generation ，Parallel Scavenge收集器的新生代名

3324K->152K(3712K)：

GC前该内存区域已使用容量->GC后该内存区域已使用容量（该内存区域总容量）

0.0025925 secs：该内存区域GC所占用的时间，单位是秒

Times： user=0.01 sys=0.00 real=0.02 secs：GC具体时间。用户态消耗的CPU时间、内核态消耗的CPU时间、操作从开始到结束所经过的墙钟时间（Wall Clock Time）。CPU时间与墙钟时间的区别是：墙钟时间包括各种非运算的等待耗时，如等待磁盘I/O、等待线程阻塞，而CPU时间不包括这些耗时。但如果系统是多CPU/多核，多线程GC操作会叠加这些CPU时间，所以user或sys时间超过real时间是完全正常的。

内存分配与回收策略

对象的内存分配，大方向上说，就是在Java堆上分配（也可能经过JIT编译后被拆散为标量类型并间接地栈上分配）。

对象主要分配在新生代的Eden区上，如果启动本地线程分配缓冲，将按线程优先在TLAB上分配。少数情况下也可能会直接分配在老年代中。分配规则主要取决于当前使用哪一种垃圾收集器组合。

几条最普遍的内存分配规则

（1）对象优先在新生代Eden分配

当Eden区没有足够空间进行分配，将会触发Java虚拟机来发起一次新生代GC(Minor GC,因为Java对象大多朝生夕灭，所以Minor GC非常频繁，回收速度也比较快；而老年代 GC——**Major GC或Full GC，通常会伴随至少一次Minor GC**，而且速度会比Minor慢10倍以上);

（2）大对象直接进入老年代

大对象指需要大量连续内存空间的Java对象，比如长字符串以及数组（byte[]）。直接进入老年代是为了避免GC时在Eden区及两个Survivor区之间发生大量的内存复制（因为新生代采用复制算法收集内存）。

（3）长期存活的对象将进入老年代

针对对象回收时使用。JVM给每个对象定义了一个年龄(Age)计数器，如果对象在Eden出生并熬过第一次Minor GC（①对象经过二次标记后被判定为存活对象；②Survivor区有足够空间能够容纳存活对象），则将该对象年龄设为1。接下来对象在Survivor区毎熬过一次Minor GC，年龄+1岁。当年龄增加一定程度（默认15岁），下一次Minor GC则会被移动到老年代中。

（4）动态对象年龄判断

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，则年龄大于或等于该年龄的对象就可以直接进入老年代，无须等到MaxTenuringThreshold中要求的年龄。

（5）空间分配担保

在发生Minor GC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生代所有对象总空间**;

if条件成立

{

那么Minor GC可以确保安全;

}

else

{

虚拟机会查看HandlePromotionFailure设置值是否允许担保失败;

if允许担保失败

{

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;

if大于

{

虚拟机将尝试进行一次具有风险的Minor GC;

else

{

改为进行一次 Full GC 让老年代腾出更多空间;

}

else

{

改为进行一次Full GC 让老年代腾出更多空间;

}

}

注：JDK 6 Update 24 之后，HandlePromotionFailure参数不会再影响到虚拟机的空间分配担保策略。并且规则改为：

只要老年代的连续空间大于新生代对象大小或者历次晋升的平均大小就会进行Minor GC，否则将进行Full GC。