1. 分代收集理论
2. 假说

当前商用虚拟机的垃圾收集器大都遵循了分代收集理论进行设计，该理论建立在两个假说之上：弱分代假说和强分代假说。弱分代假说中绝大多数对象都是朝生夕灭的；强分代假说中熬过越多次垃圾收集过程的对象越难以消亡。

1. 划分区域

收集器根据这两个假说将Java堆划分出不同的区域，一个区域中的对象都是朝生夕灭的，每次回收该区域只需要标记存活的少量对象即可，可以较低代价回收大量的空间；一个区域中的对象都是难以灭亡的，那么减少回收该区域的次数即可。根据不同区域对象的特点来进行回收，兼顾时间开销和内存开销。这两个区域一般为新生代和老年代，每次新生代中存活的少量对象能够晋升到老年代中存放。

1. 跨代引用

当收集一个新生代的内存时，新生代中的对象可能被老年代所引用，因此为了正确地找出存活对象，需要在固定的GC Root之外，额外遍历老年代的所有对象确保可达性分析结果的正确性。这样性能消耗非常大，因此引入第三个假说：跨代引用假说，跨代引用相对于同代引用来说占极少数。

因此，根据这个假说，我们不需要为了少量的跨代引用扫描整个老年代，只需要在新生代中创建一个全局的数据结构记忆集，标识出老年代中哪一块内存存在跨代引用，将这些内存纳入GC Roots中进行扫描。

1. 收集简写

部分收集： Partial GC，收集部分区域

新生代收集： Minor GC / Young GC

老年代收集： Major GC / Old GC

混合收集： Mixed GC，收集整个新生代和部分老年代

整堆收集： Full GC，收集整个Java堆和方法区

1. 垃圾收集算法
2. 标记-清除算法
3. 算法

首先标记所有需要回收的对象，标记完成后，统一回收所有被标记的对象。

2． 缺点

执行效率不稳定，对于大量对象会消耗大量资源；

内存空间的碎片化问题，产生大量不连续的内存碎片；

1. 标记-复制算法（半区复制）
2. 算法

将可用内存划分为大小相等的两块，每次使用其中的一块，当这一块的内存用完时，将还存活着的对象复制到另一块上面，然后将已使用过的内存块全部清理掉。

1. 缺点

内存缩小了一半；

1. 优点

运行高效，执行效率高

1. 标记-复制算法（Appel式回收）
2. 算法

更加优化的半区复制算法，通常用于新生代的内存布局，将新生代分为一块较大的Eden空间和两块较小的Survivor空间。每次分配内存只使用Eden和其中一块Survivor空间，需要垃圾收集时，将Eden和Survivor上存活的对象一次性复制到另一块Survivor上，清理掉Eden和用过的Survivor空间。

HotSpot虚拟机中默认Eden和Survivor的比例是8:1，因此只有10%的空间是浪费的。同时，还设置了极端情况处理方式，若Survivor空间不足以容纳一次垃圾收集后存活的对象时，需要依赖其他内存区域（多是老年代）分配担保，即进入老年代的内存空间中存储。

1. 标记-整理算法

上面介绍的标记-复制算法适用于存活对象不多的情况，因此可用于新生代中，而对于老年代，就不能使用。针对老年代存活对象较多的特点，将所有存活的对象都向内存空间的一端进行移动，然后直接清理掉边界以外的内存。

使用标记-整理算法，移动存活对象并更新所有引用这些对象的地方会耗费很多的资源，同时移动这些对象的时候必须全称暂停用户线程的运行，有停顿时间，这种资源消耗是需要考虑的因素。但是，如果不移动存活对象，使用标记-清除算法，垃圾收集时停顿时间更短，但是会造成空间碎片化问题，又得需要更复杂的内存分配器和内存访问器解决，且内存分配和访问的频率比垃圾收集要高很多，因此耗时耗资源更多。

更加优化的方式是多数时间采用标记-清除算法，容忍内存碎片的存在，当内存碎片已经大到影响对象分配时，再采用标记-整理算法收集一次，获得规整的内存空间。

1. HotSpot算法细节
2. 根节点枚举

固定可作为GC Roots的节点主要在全局性的引用（如常量或类静态属性）与执行上下文（栈帧中的本地变量表）中。即使现在查找引用链的过程已经可以和用户线程同步了，所有收集器在根节点枚举时都需要暂停用户线程的。

当用户线程停下来后，不需要检查完所有的上下文和全局的引用位置，HotSpot通过一组称为OopMap的数据结构来存放对象引用的情况，一旦类加载动作完成，HotSpot就能够将对象内什么偏移量上是什么类型的数据计算出来，收集器在扫描时就可以得知这些信息了。

1. 安全点

有许多指令会导致OopMap内容发生变化，如果为每一条指令都生成对应的OopMap，那么将会需要大量额外的空间。HotSpot在特定的位置记录了这些对象引用信息，这些位置称为安全点。只有到达了安全点后，才能够停顿下来进行垃圾收集。因此，安全点的位置选定既不能太少让收集器等待时间过程；也不能太多增加内存负荷。选取标准是“是否具有让程序长时间执行的特征”为标准选定的，因此对于指令序列的复用才会产生安全点，如方法调用，循环跳转等。

那么，如果在垃圾收集时让所有线程都跑到最近的安全点呢？有两种方案：

1. 抢先式中断

中断所有用户线程，如果此时中断的线程不在安全点上，恢复该线程到安全点上，这种方案几乎没有虚拟机采用。

1. 主动式中断

当垃圾收集需要中断线程时，设置一个标志位，所有线程都会主动轮询这个标志，若该标志为真，则跑到最近的安全点上主动中断挂起。轮询标志与安全点是重合的，同时还要在所有创建对象和需要在Java堆上分配内存的地方设置轮询标志，检查是否要发生垃圾收集，避免没有足够内存分配新对象。

1. 安全区域

当用户线程执行时，可以通过安全点来停止用户线程，但是若线程处于阻塞状态时，无法走到安全点中断线程，就要引入安全区域了。安全区域是指在某一段代码片段之中，引用关系不会发生变化，在这个区域中任意地方开始收集垃圾都是安全的。

当用户线程执行到安全区域里面的代码时，首先会标识自己已经进入了安全区域中；当用户线程要离开安全区域时，会检查虚拟机是否完成了根节点枚举等需要暂停用户线程的操作，如果完成了，就继续执行线程，否则，持续等待。

1. 记忆集和卡表

记忆集是一种用于记录从非收集区域指向收集区域的指针集合的抽象数据结构，收集器通过记忆集来判断出某一块非收集区域是否存在有指向了收集区域的指针即可。记录的精度有三种：

字长精度： 每个记录精确到机器字长，包含跨代指针；

对象精度： 每个记录精确到对象，对象中包含跨代指针；

卡精度 ： 每个记录精确到一块内存区域，该区域包含对象和跨代指针；

卡精度指的是用一种称为“卡表”的形式实现记忆集，这是最常用的实现方式，定义了记忆集的记录精度，与堆内存的映射关系。卡表的最简单形式是一个字节数组，数组中的每个元素都对应着其标识的内存区域中一块特定大小的内存块，称为卡页，HotSpot中的卡页是2的9次幂，512字节。一个卡页的内存中不止一个对象，只要卡页内有一个对象的字段存在跨代指针，则对应卡表的数组元素的指标识为1，称为该元素变脏了，否则为0。垃圾收集时，筛选出卡表中变脏的元素，即可选出对应的内存，加入GC Roots中扫描。

1. 写屏障

写屏障用于解决卡表元素的维护问题。当卡表元素需要更新时，需要在引用字段赋值的那一刻更新卡表，对于即时编译后的机器指令流，需要在机器码层面上将维护卡表的动作放到每一个复制操作之中。

写屏障类似于Spring中的AOP切面编程，在引用对象赋值时会产生一个环形通知，共程序会执行额外的操作，也就是说赋值的前后都在写屏障的范围之内，可写前屏障，也可写后屏障。写屏障应用于所有的赋值操作，会产生额外的开销。

卡表除了写屏障的开销，在高并发场景下还有“伪共享”问题，伪共享在处理并发底层细节时经常遇到。CPU的缓存系统是以缓存行为单位存储的，当多线程修改互相独立的变量时，若干这些变量共享同一个缓存行，则会互相影响导致性能降低。简单的解决办法是，不采用无条件的写屏障，而是先检查卡表标记，只有当该卡表元素未被标记过时才标记为变脏。