1. Serial收集器

Serial收集器是历史最悠久的收集器，在进行垃圾收集工作时，需要暂停其他的工作线程，因此是单线程收集器，这会让用户线程产生停顿时间。迄今为止，他依然是HotSpot虚拟机运行在客户端模式下的默认新生代收集器，优点是简单高效，对于内存资源受限的环境，是所有收集器里额外内存消耗最少的，能够获得较高的收集器效率。

1. ParNew收集器

Serial收集器的多线程版本，同时使用多条线程进行垃圾收集，是不少运行在服务端模式下的HotSpot虚拟机首选的新生代收集器，主要原因是除了Serial收集器，目前只有它能与CMS收集器配合工作。

1. CMS收集器

真正意义上支持用户线程与垃圾收集线程并发工作的收集器，通常作为老年代收集器。

1. Parallel Scavenge收集器

新生代收集器，基于标记-复制算法实现的，也是能够并行收集的多线程收集器，与CMS等收集器不同的点是CMS关注的是缩短垃圾收集时用户的等待时间，而Parallel Scavenge收集器的目标是达到一个可控制的吞吐量。

所谓的吞吐量是处理器用于运行用户线程的时间与处理总耗时（用户线程+垃圾收集线程）的比值，因此吞吐量越高表示能够最高效率地利用处理器资源，完成用户的任务。而停顿时间越短保证的是用户的交互体验，提高了响应速度。

当我们为收集器设定了吞吐量的优化目标后，Parallel Scavenge收集器也可以通过自适应调节策略优化参数，以达到最大的吞吐量。

1. Serial Old收集器

Serial收集器的老年代版本，同样是单线程收集器，使用标记-整理算法，也是供客户端下的HotSpot虚拟机使用。

1. Parallel Old 收集器

Parallel Scavenge收集器的老年代版本，支持多线程并发收集，基于标记-整理算法实现。

1. CMS收集器

CMS收集器是一种以获取最短回收停顿时间为目标的收集器，基于标记-清除算法实现，整体过程分为以下几步：

1. 初始标记

标记GC Roots能够关联到的对象，需要暂停所有线程。

1. 并发标记

从GC Roots的直接关联对象开始遍历整个对象图，耗时较长但是不需要暂停所有线程，且可与用户线程并发执行。

1. 重新标记

为了修正并发标记期间，因为用户线程并发运行而产生的标记变动，这个阶段的停顿时间会比初始标记长一些。

1. 并发清除

清理删除掉标记阶段判断为死亡的对象，因为不用移动存活的对象，因此这个阶段也是与用户线程并发的。

由于CMS的内存回收过程，最耗时的并发标记和并发清除过程都是和用户线程并发进行的，因此总体上来说，CMS收集器的回收过程与用户线程并发进行。

CMS的优点是并发收集，低停顿时间。但是也有缺陷，它对处理器资源很敏感，由于是并发收集的，因此垃圾回收会导致用户程序变慢，降低吞吐量。增量式并发收集器（i-CMS）是改进的CMS收集器，通过用户线程与垃圾回收线程交替进行来减少垃圾回收线程的独占资源时间，提高用户体验。

浮动垃圾指的是在与回收线程并发的用户线程进行时所产生的垃圾，一部分垃圾对象出现在标记过程结束后，因此CMS无法在档次的收集中处理他们，只能留到下一次垃圾收集时处理，因此CMS需要留出部分的内存空间给用户线程的运行和浮动垃圾，且需要提前在老年代被填满前收集垃圾，收集的预警阈值需要合理地设置。

1. Garbage First收集器
2. G1概述

简称G1收集器，面向服务端应用的收集器。从JDK10起，HotSpot提出了统一垃圾收集器接口，将内存回收的行为与实现分离。G1一改之前固定的内存分配模式，将内存中的任何部分组合成回收集进行回收，衡量的标准不再是它属于哪个分代，而是那块内存中存放的垃圾最多，回收收益最大，这就是G1收集器的Mixed GC模式。

（二） 区域

虽然G1仍是遵循分代收集理论设计的，但是它将连续的Java堆划分为多个大小相等的独立区域，每个区域都可以根据需要，扮演新生代的Eden空间，Survivor空间，或者老年代空间，这样G1能够对扮演不同角色的区域采用不同的策略来处理。

（三） 收益最大化

G1将每个区域作为单次回收的最小单元，每次收集到的内存空间都是区域的整数倍，这样可以有计划地避免在整个Java堆中进行全区域的垃圾收集。更具体的是让G1跟踪各个区域里面的垃圾堆积的价值，所谓的价值是指垃圾回收所能够获得的空间大小以及回收所需的时间的经验值，然后在后台维护一个优先级列表，优先处理收益最大的区域。

（四） G1的问题

每个区域中存在的跨区域引用问题，之前我们是通过记忆集解决的，因此G1中也是采用记忆集解决，不过是每一块区域都有自己的记忆集，这就导致G1比其他的收集器有着更高的内存占用负担。

用户线程与收集线程如何并发执行，要解决用户线程改变对象引用关系时，必须保证不能打破原有的对象图的结构，导致标记结果出差错。G1采用原始快照（SATB）算法解决。

垃圾收集过程中用户线程会创建新的对象，需要分配内存，G1为每个区域设计了两个名为TAMS（Top at Mark Start）的指针，将区域中的一部分空间划分出来用于并发回收中的新对象分配，分配的新对象地址都必须在这两个指针位置以上，这些对象都是被隐式标记过，默认是存活的，不纳入回收范围。

（五） G1的回收过程

1. 初始标记

标记GC Roots直接关联的对象，并且修改TAMS指针的值，这个阶段需要暂停线程。

1. 并发标记

从GC Roots开始递归扫描整个堆里的对象图，耗时较长，但可与用户线程并发执行。当对象图扫描完成后，还需要重新处理SATB记录下的在并发时有引用变动的对象。

1. 最终标记

暂停用户线程，处理并发阶段结束后仍遗留下来的少量SATB记录。

1. 筛选回收

更新区域的统计数据，对每个区域的回收价值和成本进行排序，根据用户所期望的停顿时间制定回收计划，可以自由选项选择任意多个区域构成回收集，然后将回收的区域的存活对象复制到空的区域中，最后直接清理掉旧的区域的全部空间。这阶段涉及到存活对象的移动，必须暂停用户线程，有多条收集线程并发执行。

（六） G1的优势与劣势

1. 优势

设计理念转变为能够应付应用的内存分配速率即可，而不追求将真个Java堆全部清理干净；

G1从整体上看是基于标记-整理算法，而局部来看是基于标记-复制算法实现的，因此不会产生内存空间碎片，垃圾收集完成后能够提供规整的可用内存，有利于程序的长时间运行。

1. 劣势

每个区域都有一个卡表，这导致G1的记忆集的内存占用率过高；

程序运行时的额外负载过高，如G1除了使用写后屏障来进行卡表维护操作，还是用了写前屏障来实现原始快照搜索（SATB）算法，跟踪并发时的指针变化情况