收集器	串行、并行or并发	新生代/老年代	算法	目标	适用场景
Serial	串行	新生代	复制算法	响应速度优先	单CPU环境下的Client模式
Serial Old	串行	老年代	标记-整理	响应速度优先	单CPU环境下的Client模式、CMS的后备预案
ParNew	并行	新生代	复制算法	响应速度优先	多CPU环境时在Server模式下与CMS配合
Parallel Scavenge	并行	新生代	复制算法	吞吐量优先	在后台运算而不需要太多交互的任务
Parallel Old	并行	老年代	标记-整理	吞吐量优先	在后台运算而不需要太多交互的任务
CMS	并发	老年代	标记-清除	响应速度优先	集中在互联网站或B/S系统服务端上的Java应用
G1	并发	both	标记-整理+复制算法	响应速度优先	面向服务端应用，将来替换CMS

1、Serial收集器：

1）特性：

这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。Stop The World

2）应用场景：

Serial收集器是虚拟机运行在Client模式下的默认新生代收集器；也可使用-XX:+UseSerialGC指定。

3）优势：

简单而高效（与其他收集器的单线程比），对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。

2、Serial Old收集器：

1）特性：

Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用标记－整理算法。每次进行全部回收，进行Compact，非常耗费时间。

2）应用场景：

Client模式：client模式下的默认GC方式，可通过-XX:+UseSerialGC强制指定。

3、ParNew收集器：

1）特性：

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。（内存分配、回收和PS相同，不同的仅在于会收拾会配合CMS做些处理）

2）应用场景：

ParNew收集器是许多运行在Server模式下的虚拟机中首选的新生代收集器，可以和cms老年代回收期配合使用。当old代采用CMS GC时new代默认采用ParNew，也可以采用-XX:+UseParNewGC指定。

很重要的原因是：除了Serial收集器外，目前只有它能与CMS收集器配合工作。

3）Serial收集器 VS ParNew收集器：

ParNew收集器在单CPU的环境中绝对不会有比Serial收集器更好的效果，甚至由于存在线程交互的开销，该收集器在通过超线程技术实现的两个CPU的环境中都不能百分之百地保证可以超越Serial收集器。 然而，随着可以使用的CPU的数量的增加，它对于GC时系统资源的有效利用还是很有好处的。

4、Parallel Scavenge收集器：

1）特性：

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器。server模式下的默认GC方式，也可用-XX:+UseParallelGC强制指定。

2）应用场景：

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

5、Parallel Old收集器：

1）特性：

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记－整理”算法。

2）应用场景：

在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器。server模式下的默认GC方式，也可用-XX:+UseParallelGC=强制指定；可以在选项后加等号来制定并行的线程数。

6、CMS收集器：

1）特性：

CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求。可以使用-XX:+UseConcMarkSweepGC=指定使用，后边接等号指定并发线程数。

CMS收集器是基于“标记—清除”算法实现的，它的运作过程相对于前面几种收集器来说更复杂一些，整个过程分为4个步骤：

A、初始标记（CMS initial mark）：初始标记仅仅只是标记一下GC Roots能直接关联到的对象，速度很快，需要“Stop The World”。

B、并发标记（CMS concurrent mark）：并发标记阶段就是进行GC Roots Tracing的过程。

C、重新标记（CMS remark）：重新标记阶段是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短，仍然需要“Stop The World”。

D、并发清除（CMS concurrent sweep）：并发清除阶段会清除对象。

由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作，所以，从总体上来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

2）优点：

CMS是一款优秀的收集器，它的主要优点在名字上已经体现出来了：并发收集、低停顿。

3）缺点：

A、CMS收集器对CPU资源非常敏感。其实，面向并发设计的程序都对CPU资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或者说CPU资源）而导致应用程序变慢，总吞吐量会降低。

CMS默认启动的回收线程数是（CPU数量+3）/ 4，也就是当CPU在4个以上时，并发回收时垃圾收集线程不少于25%的CPU资源，并且随着CPU数量的增加

而下降。但是当CPU不足4个（譬如2个）时，CMS对用户程序的影响就可能变得很大。

B、CMS收集器无法处理浮动垃圾：

CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。

由于CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。也是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。

C、CMS收集器会产生大量空间碎片 CMS是一款基于“标记—清除”算法实现的收集器，这意味着收集结束时会有大量空间碎片产生。空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。

7、G1收集器：

1、特性：

G1（Garbage-First）是一款面向服务端应用的垃圾收集器。HotSpot开发团队赋予它的使命是未来可以替换掉JDK 1.5中发布的CMS收集器。与其他GC收集器相比，G1具备如下特点。

1）并行与并发

G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop-The-World停顿的时间，部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让Java程序继续执行。

2）分代收集

与其他收集器一样，分代概念在G1中依然得以保留。虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。

3）空间整合

与CMS的“标记—清理”算法不同，G1从整体来看是基于“标记—整理”算法实现的收集器，从局部（两个Region之间）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次GC。

4）可预测的停顿

这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，但G1除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。

在G1之前的其他收集器进行收集的范围都是整个新生代或者老年代，而G1不再是这样。使用G1收集器时，Java堆的内存布局就与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合。

G1收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region（这也就是Garbage-First名称的来由）。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限的时间内可以获取尽可能高的收集效率。