根据上述自己对于 1 和 2 的演示，写一段对于不同 GC 和堆内存的总结，提交到 GitHub

GC 的背景与一般原理:

1. 清除清除算法（清除算法）
2. 标记复制算法（复制算法）
3. 标记-清除-整理算法（整理算法）

串行GC:

通过JVM命令-XX:+UseSerialGC可以使用串行垃圾回收器

对年轻代使用复制算法，对老年代使用整理算法。

GC线程只有一个，它会暂停所有工作线程，一个一个内存区域来收集，不适合服务器环境；通过JVM命令-XX:+UseParNewGC可以使用串行垃圾回收器的改进版本，配合CMS使用。

适合几百MB的堆内存的JVM，单核，CPU利用率高，但，容易卡死。

并行GC:

其实是串行GC的多线程版本；对年轻代使用复制算法，对老年代使用整理算法。对年轻代和老年代垃圾的回收都会触发STW。

ParNew收集器默认开启和CPU数目相同的线程数，可以通过-XX:ParallelGCThreads参数来限制垃圾收集器的线程数。

多核服务，主要增加吞吐量。

另外：Java8的默认GC算法是G1并行GC算法

CMSGC:

-XX:+UseConcMarkSweepGC

对年轻代使用复制算法，对老年代使用清除算法。但是，CMSGC收集器是一种年老代垃圾收集器，其最主要目标是获取最短垃圾回收停顿时间。

1.初始标记：为了收集应用程序的对象引用需要暂停应用程序线程，该阶段完成后，应用程序线程再次启动。

2.并发标记：从第一阶段收集到的对象引用开始，遍历所有其他的对象引用。

3.并发预清理：改变当运行第二阶段时由应用程序线程产生的对象引用，以更新第二阶段的结果。

4.重标记：由于第三阶段是并发的，对象引用可能会发生进一步改变。因此，应用程序线程会再一次被暂停以更新这些变化，并且在进行实际的清理之前确保一个正确的对象引用视图。这一阶段十分重要，因为必须避免收集到仍被引用的对象。

5.并发清理：所有不再被应用的对象将从堆里清除掉。

6.并发重置：收集器做一些收尾的工作，以便下一次GC周期能有一个干净的状态。

G1GC:

全称Garbage-First Garbage Collector，通过-XX:+UseG1GC参数来启用。

G1收集器是工作在堆内不同分区上的收集器，分区既可以是年轻代也可以是老年代，同一个代的分区不需要连续。并且每个代分区的数量是可以动态调整的。为老年代设置分区的目的是老年代里有的分区垃圾多，有的分区垃圾少，这样在回收的时候可以专注于收集垃圾多的分区，这也是G1名称的由来。不过这个算法并不适合新生代垃圾收集，因为新生代的垃圾收集算法是复制算法，但是新生代也使用了分区机制主要是因为便于代大小的调整。

Garbage first垃圾收集器是目前垃圾收集器理论发展的最前沿成果，相比CMS收集器，G1收集器两个最突出的改进是：

a.基于标记-整理算法，不产生内存碎片。

b.可以非常精确控制停顿时间，在不牺牲吞吐量前提下，实现低停顿垃圾回收。

另外：Java9的默认GC算法是G1算法

和CMS类似，并发收集让应用程序和GC线程交替工作，因此在特别繁忙的情况下无可避免的会发生回收过程中内存不足的情况，当遇到这种情况，G1会转入一个Full GC 进行回收。

1. G1启动标记周期，但在Mix GC之前，老年代就被填满，这时候G1会放弃标记周期。这种情形下，需要增加堆大小，或者调整周期（例如增加线程数-XX:ConcGCThreads等）可以在日志中看到GC concurrent-mark-abort
2. G1收集器完成了标记阶段，开始启动混合式垃圾回收，清理老年代的分区，不过，老年代空间在垃圾回收释放出足够内存之前就会被耗尽。（G1在进GC的时候没有足够的内存供存活对象或晋升对象使用），由此触发了Full GC。可以在日志中看到(to-space exhausted)

解决这种问题的方式是：

2.1增加 -XX:G1ReservePercent 选项的值（并相应增加总的堆大小），为“目标空间”增加预留内存量。

2.2通过减少 -XX:InitiatingHeapOccupancyPercent 提前启动标记周期。

2.3也可以通过增加 -XX:ConcGCThreads 选项的值来增加并行标记线程的数目。

1. 进行新生代垃圾收集是，Survivor空间和老年代中没有足够的空间容纳所有的幸存对象。可以在日志中看到（to-space overflow）
2. 当巨型对象找不到合适的空间进行分配时，就会启动Full GC，来释放空间。这种情况下，应该避免分配大量的巨型对象，增加内存或者增大-XX:G1HeapRegionSize，使巨型对象不再是巨型对象。