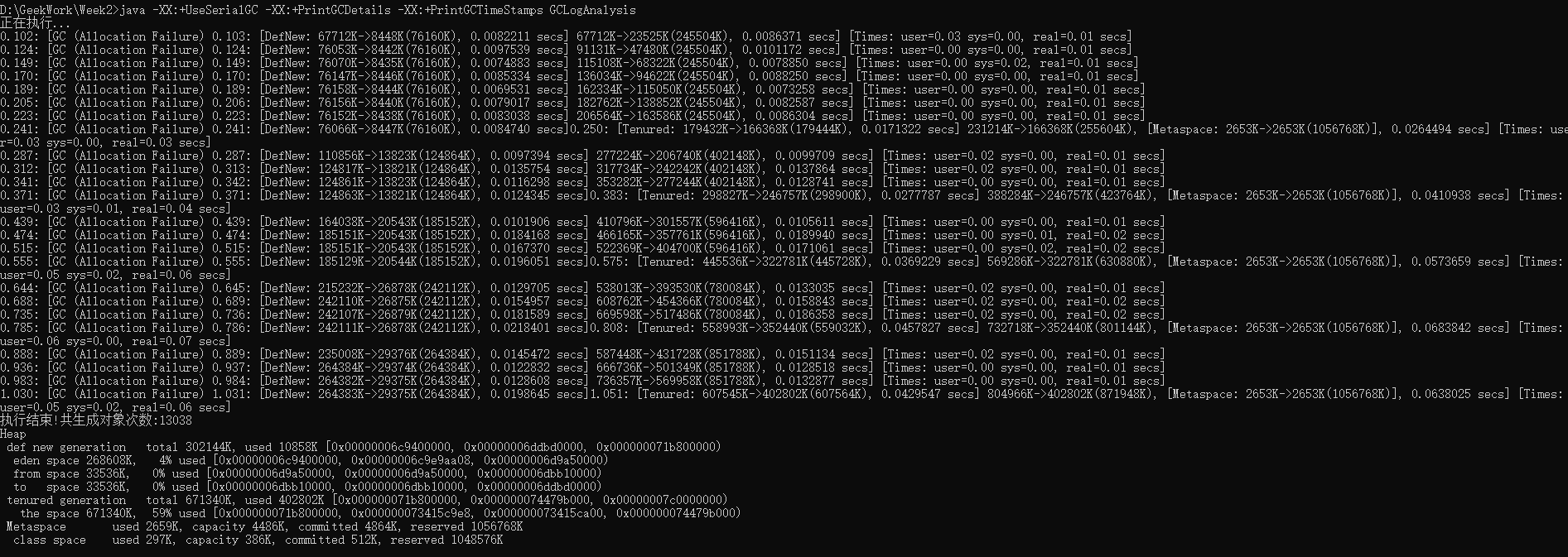
1. **使用串行GC不调堆大小的情况下**



**完整分析：**

一共生成了13038个对象，一共进行了24次GC操作，其中5次MojorGC，19次MinorGC，GC引起的原因都是没有足够的内存存储新数据。

**分步分析（以第一次MojorGC为例）：**

**0.241：**

第一次执行MojorGC的时间是在程序运行到0.241s处

**GC:**

SerialGC执行，表示进行了一次GC

**DefNew：**

串行GC新生代使用的回收算法

**76066K->8447K(76160K)：**

表示新生代从76066K减少到8447K，新生代的使用率从99%减少到11%

**0.0084740 secs：**

这个区域的GC一共使用了0.0084740s，相当于8ms

**0.250:**

在0.250这个时刻开始执行Tenured进行老年代回收

**Tenured:**

串行GC老年代使用算法

**179432K->166368K(179444K)**：

老年代的大小从179432K减少到166368K，使用率从接近100%减少到92%

**0.0171322 secs：**

消耗了0.0171322s，相当于17ms

**231214K->166368K(255604K)：**

表示整个堆的使用从231214K减少到166368K，堆使用率从90%减少到65%，我们通过计算可以看到其中有2773k的数据从年轻代升级到老年代

**Metaspace:**

Meta区GC执行

**2653K->2653K(1056768K)：**

Meta区数据没变化

**0.0264494 secs：**

Meta区GC一共消耗26ms

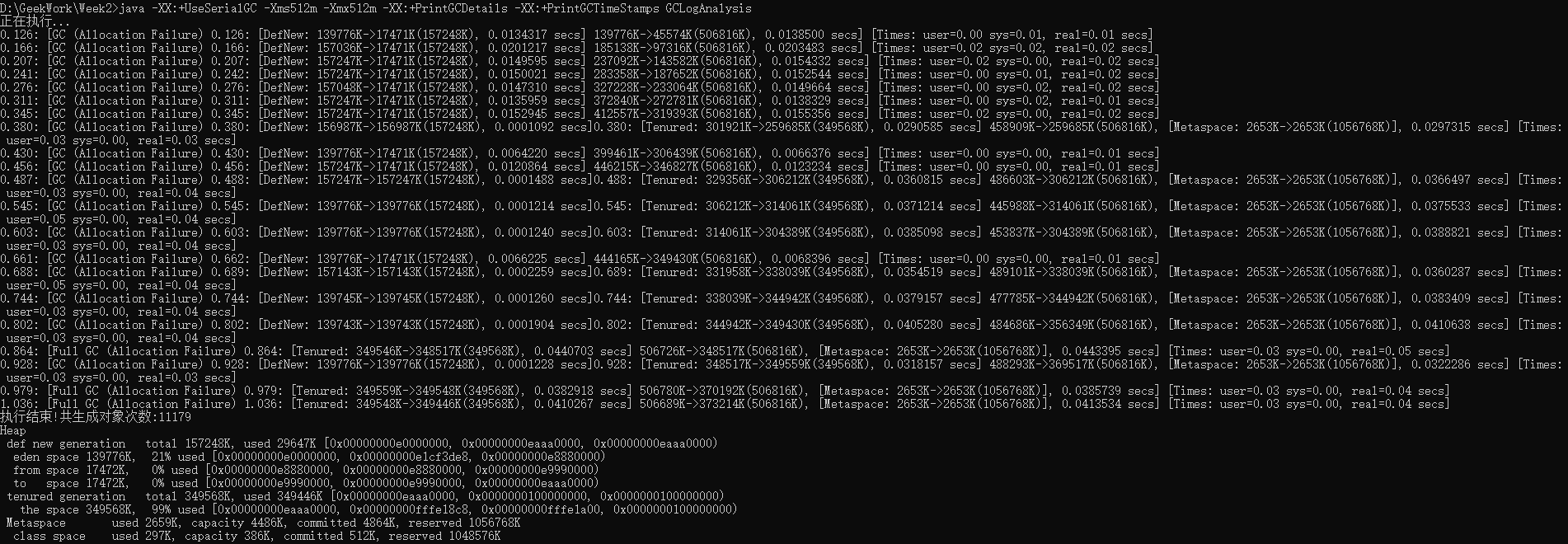
**Times: user=0.03 sys=0.00, real=0.03 secs：**

用户态消耗30ms，内核态没有消耗，真实时间为30ms

**总结：**

串行GC执行一次时间较长，而且无论MinorGC还是FullGC都会导致STW，而且随着内存的增大，GC的时间也在不断的增加，回收效率也较低

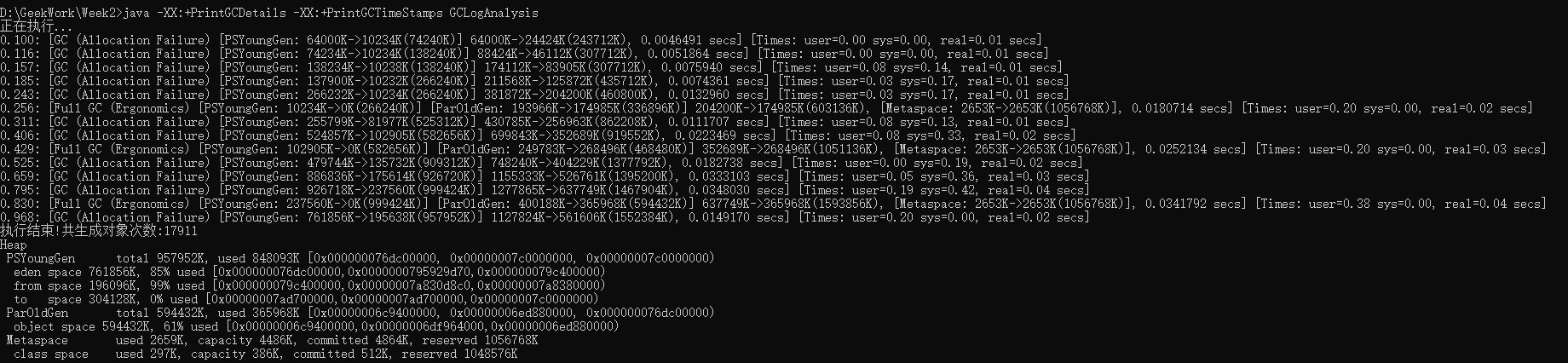
1. **使用串行GC调堆内存大小**



分析：

此时出现了FullGC，而且约到之行的最后FulGC执行的越频繁，这对于系统是很不友好的，因为频繁的FullGC会大量产生STW，会导致系统出现很严重的卡顿，而且还不一定能回收多少数据。

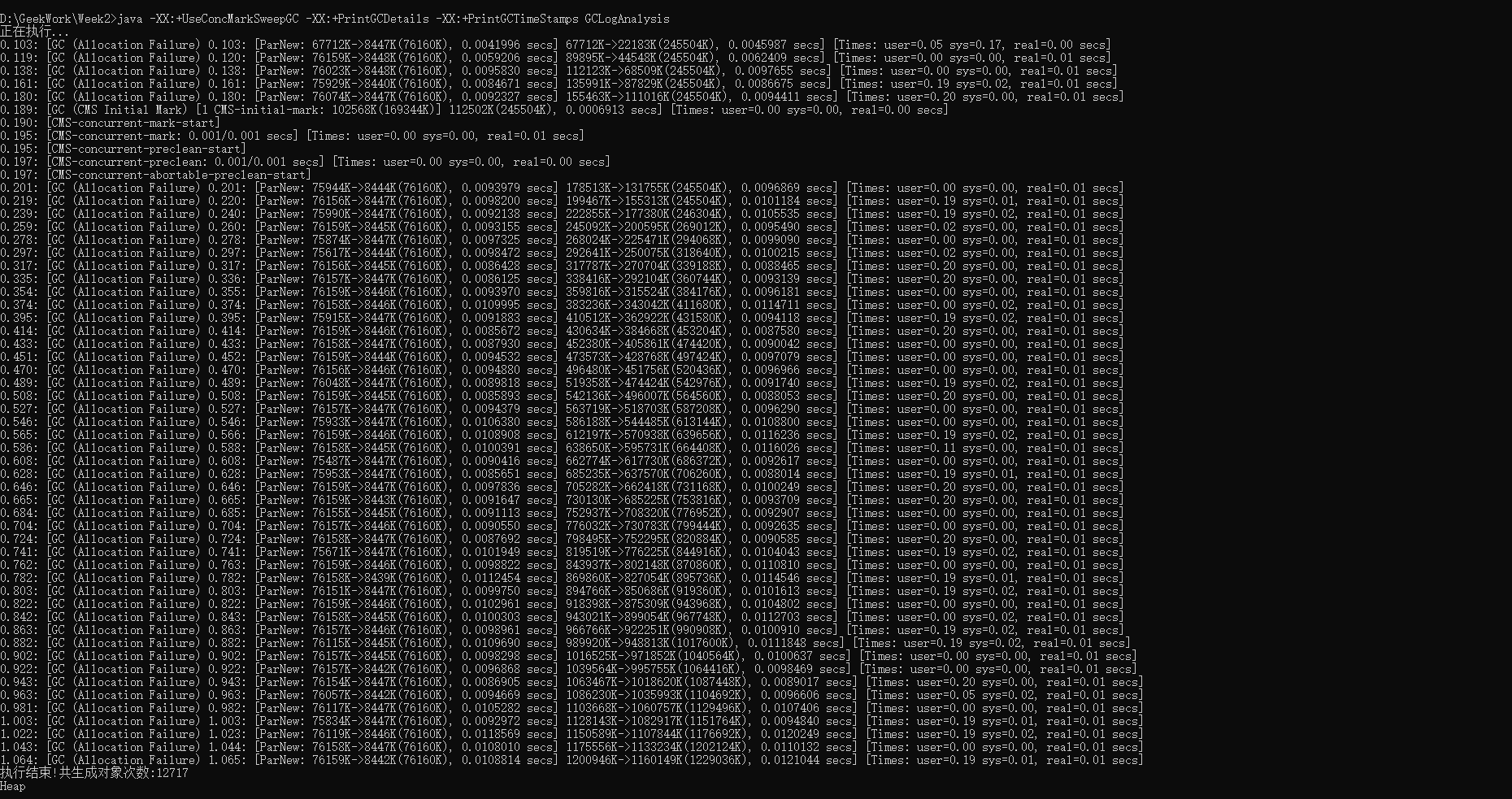
1. **使用ParallelGC的情况下**



分析：

没有规定堆大小的情况下，默认堆大小为1.5G左右，一共生成了17911个对象，对比SerialGC，多生成了1/3的对象，内存结构被更好的优化，使用的新生代算法为PSYoungGen，老年代算法为ParOldGen，虽然并行GC两次都会导致STW，但是整体来看还是比串行快很多。

1. **使用CMS的情况下**



分析：

从上图可以看出整个新生代使用的是parNew的算法，进行老年代回收会使用CMS算法，而且可以完整的看出CMS执行的六步：

第一步的CMS是初始标记，是串行的，只标记GCRoots的第一个引用；

第二步是并发标记，是并行的，标记所有存活对象；

第三步是并行预处理，是并行的 ，识别在此期间改变的引用；

第四步是最终标记，是并行的，标记那些改变的引用；

第五步是并发清除，是并行的，清除垃圾数据；

第六步会进行并发重置。

虽然整体内存存储数据不多，但是整个堆大小相对并行GC小了300M左右，而且回收速率快了很多，而且解决了STW长时间卡顿问题

1. **使用G1GC的情况**



G1GC的回收效率非常快，虽然会产生STW但是这么快也不会受什么影响，但是对于没有对堆大小进行配置的情况下数据会比较少，只有10000左右，但是在堆参数很大时G1有明显的优势。