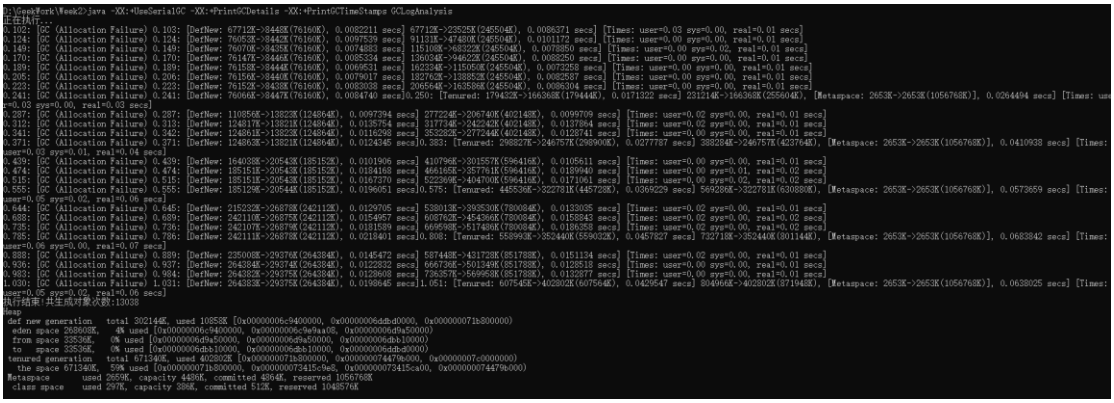
1.使用串行 GC 不调堆大小的情况下



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

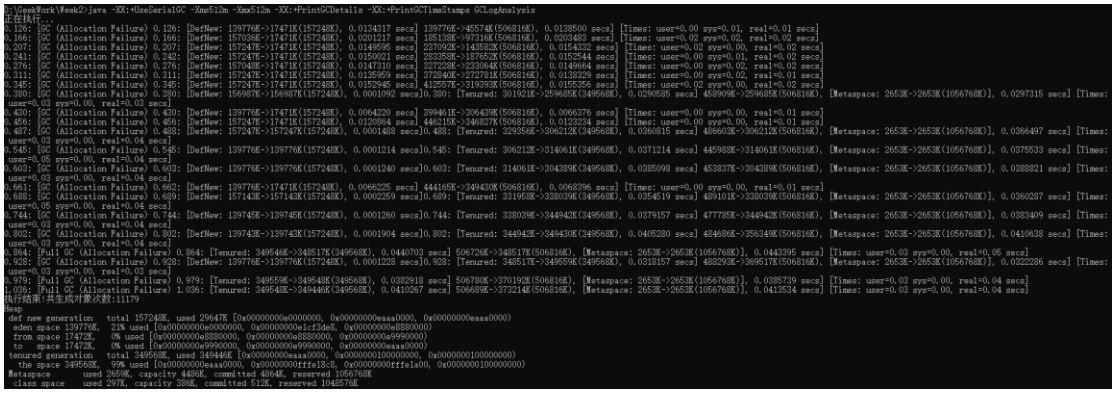
Times: user=0.03 sys=0.00, real=0.03 secs：

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

2.使用串行 GC 调堆内存大小



分析:

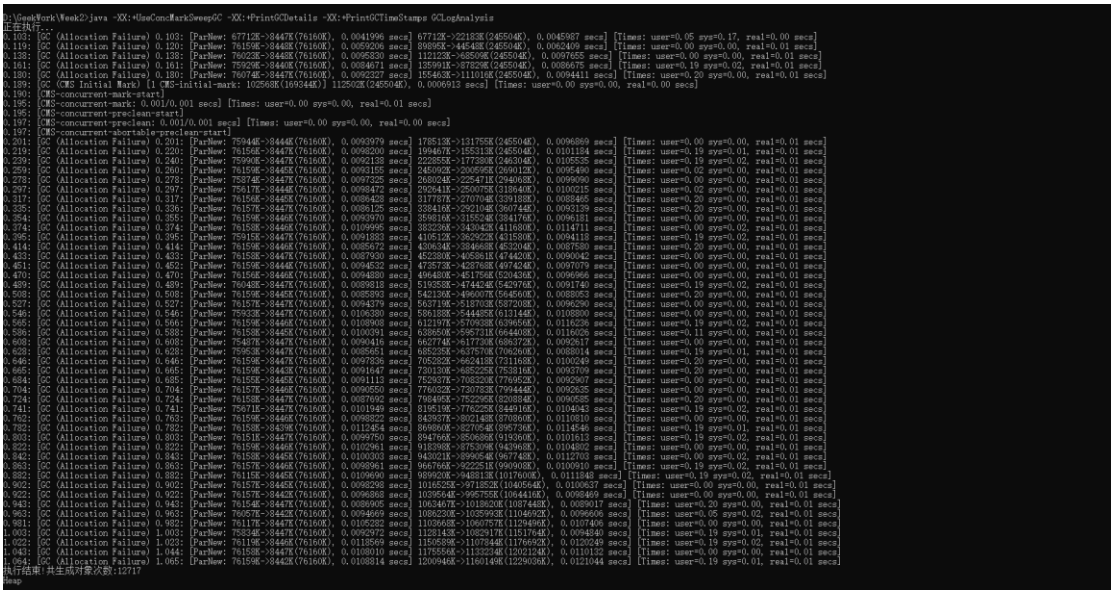
此时出现了 FullGC，而且约到之行的最后 FuIGC 执行的越频繁，这对于系统是很不友好的，因为频繁的 FullGC 会大量产生 STW，会导致系统出现很严重的卡顿，而且还不一定能回收多少数据。

3.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。

4.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;

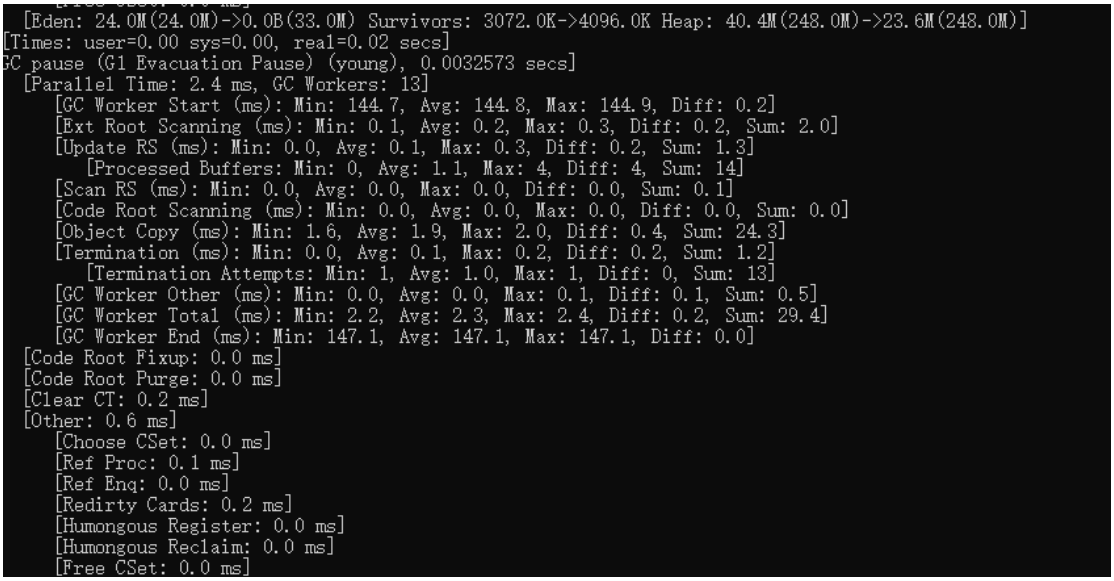
第四步是最终标记，是并行的，标记那些改变的引用;

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

5.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。