# 电脑环境：

I79700k

16G内存

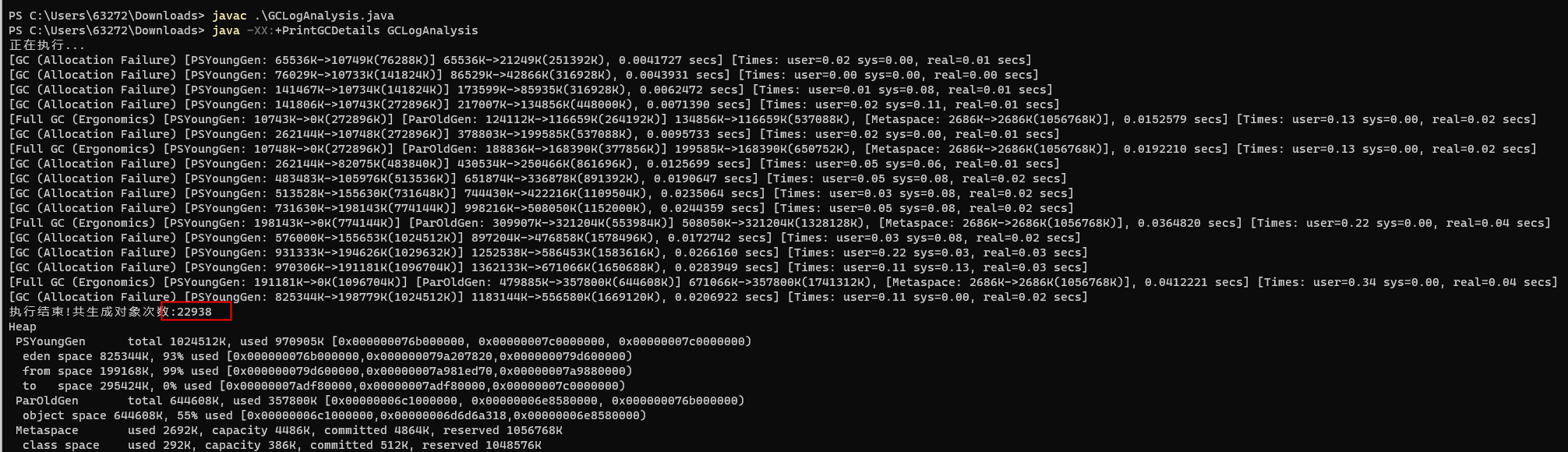
# 并行GC：

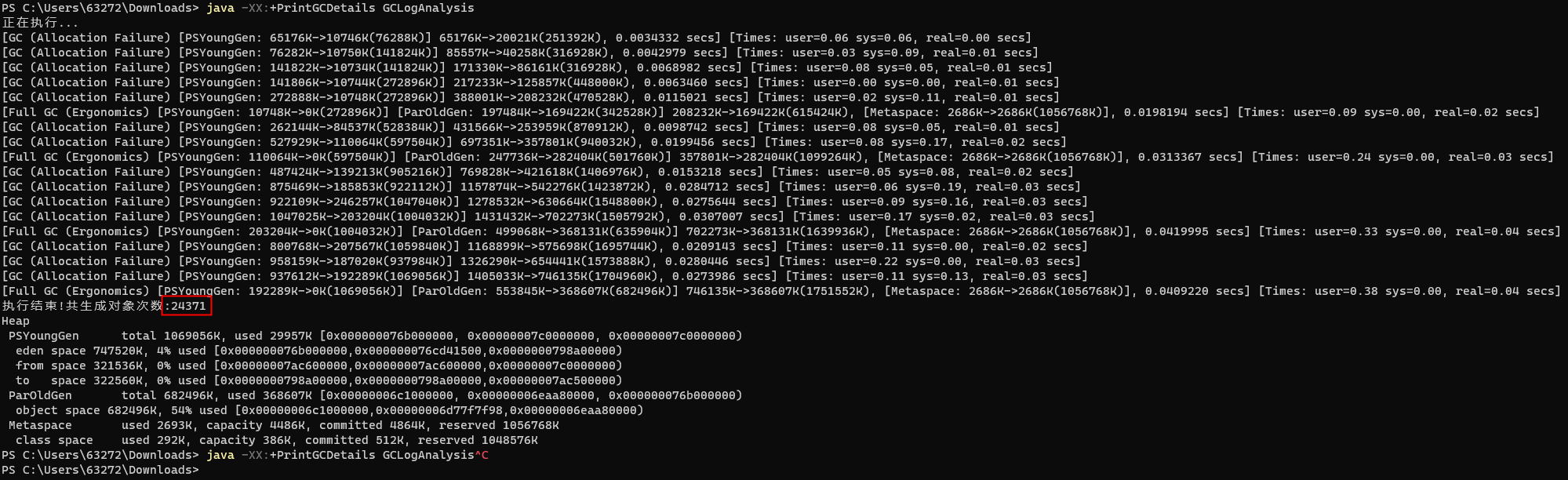
我首先执行了

java -XX:+PrintGCDetails GCLogAnalysis

指令

[我的电脑是i7-9700K@3.6GHZ、16G](mailto:我的电脑是i7-9700K@3.6GHZ、16G)内存 台式机，执行2次的结果如下：

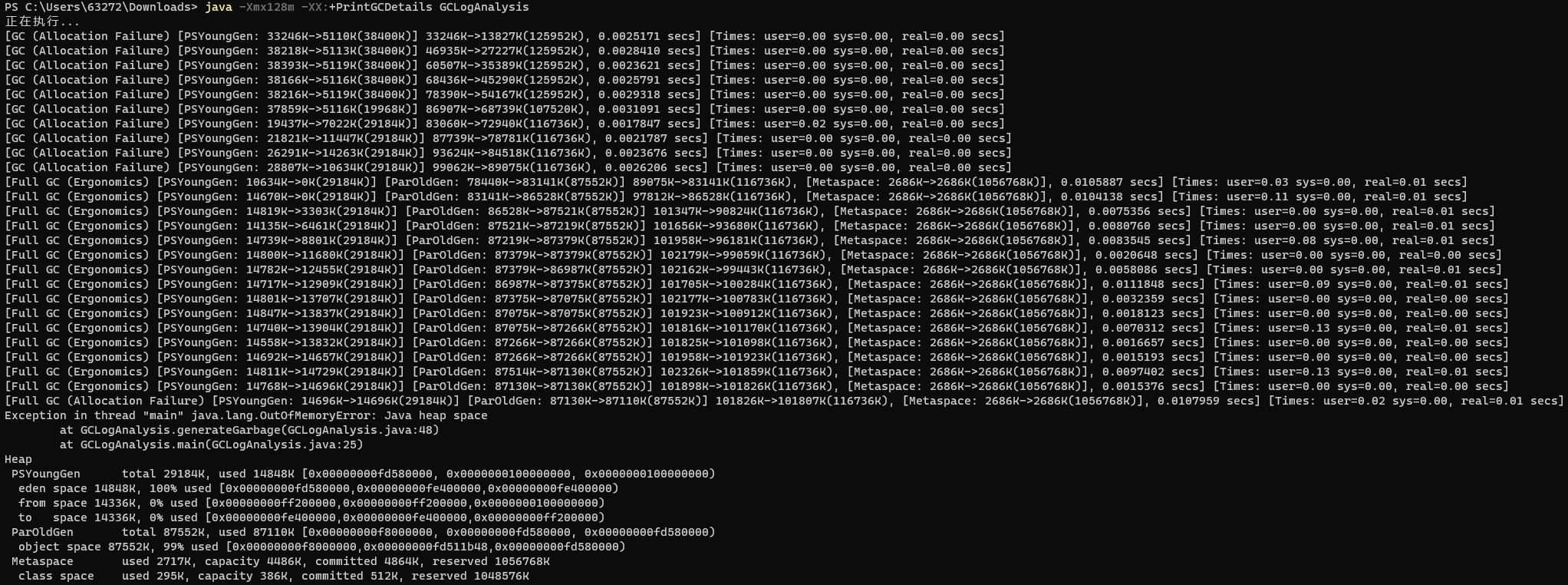


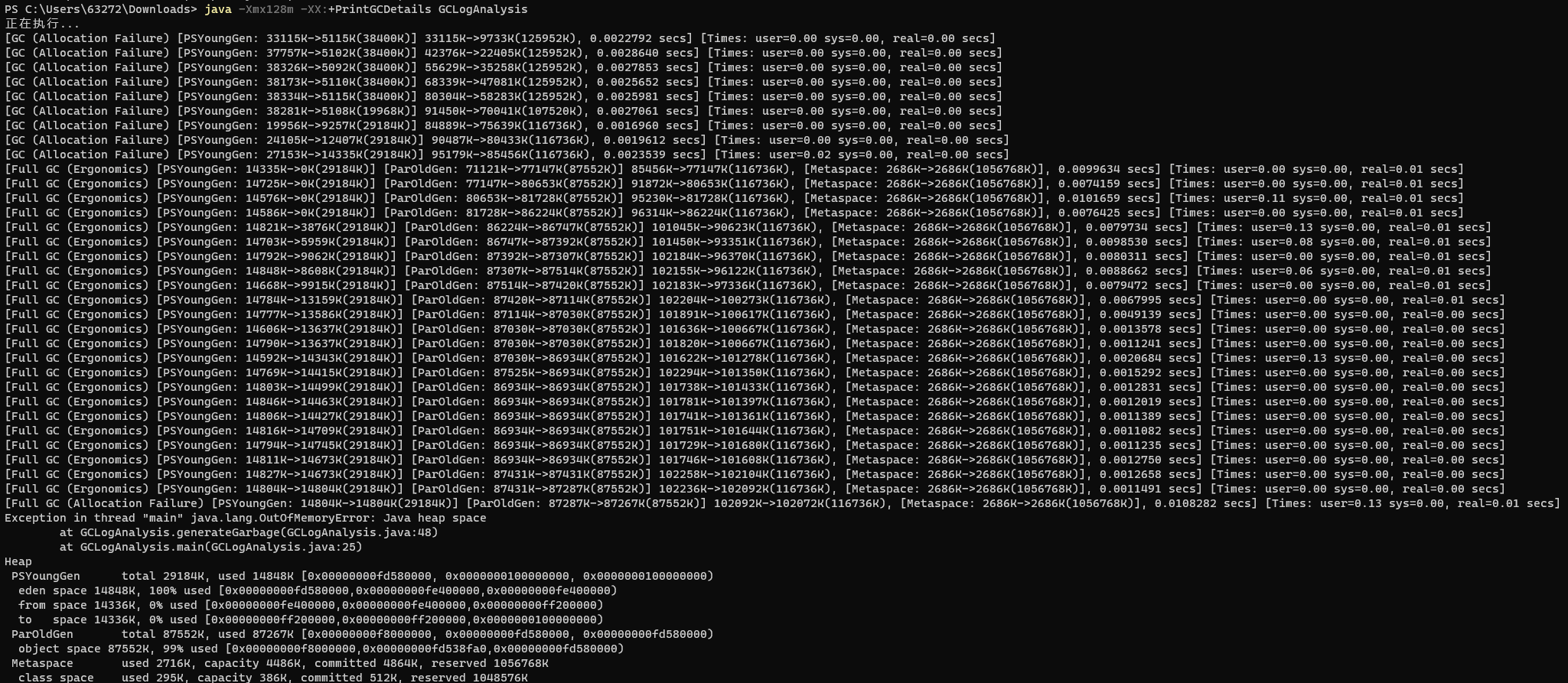


老师做演示的时候说是随机数，可是之后我都试了几次，发现都是2w+，按照台式机与笔记本CPU的差别，生成对象次数相差如此之大的主要因素在于核心、线程、CPU单核频率、三级缓存以及功耗有一定的关系。

java -Xmx128m -XX:+PrintGCDetails GCLogAnalysis

此命令执行2次，一共出现两次OOM，一次是16次FullGC后OOM，一次是24次FullGC后OOM。额，实际上应该是这两次第N（FullGC执行次数）-1次后再次执行FullGC分配失败出现的OOM。





在执行了2次Xmx512m跟Xmx1g后，发现了一个规律，就是Xmx512m的时候，两次FullGC的次数在16~17次。而变成1g后，2次FullGC的次数都在6次。后面又试验了好几次。

把Xmx从1g变成2g或4g后，发现FullGC的次数只是减少了1到2次，甚至是不减少。但是，生成对象次数截然不同。2g的时候，生成对象基本在1.7w~2.1w之间，但是，4g的时候，对象生成次数基本都在2.3w+前后，这让我想到了最开始没有指定Xmx的时候，如果不指定默认是内存的1/4。16g的1/4不就正好是4g吗。

以上为JDK默认的并行GC，并未指定其他GC。

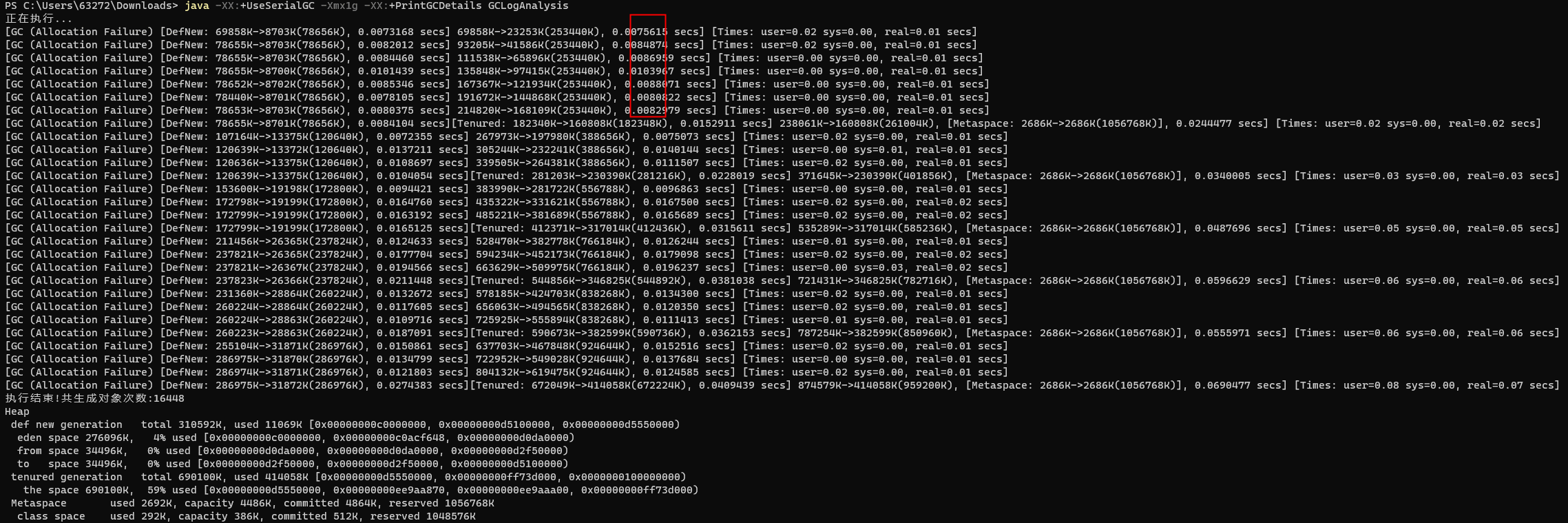
# 总结1：

并行GC执行的FullGC完全由堆内存大小决定，堆内存小的时候，比如128m这种极端情况（一般生产不会这么小，起步都4g）下由于堆内存过小，JVM需要不停的GC用以释放空间，否则容易导致OOM。通过与老师课上演示比对，发现代码的一定执行情况与机器配置有着一定的关系，具体体现在CPU核心数、核心线程、CPU缓存有关。由此可作出一个假设，是否我们的代码跑在服务器上的时候可以根据头几次运行的时候的GC日志，对机器的配置进行删减（PS：有这个假设主要是我们公司开始按业务模块区分机器配置了，以前是无脑怼机器，现在是针对代码在new对象以及其他API级操作时候，对CPU、JVM的影响从而决定机器配置的升降）

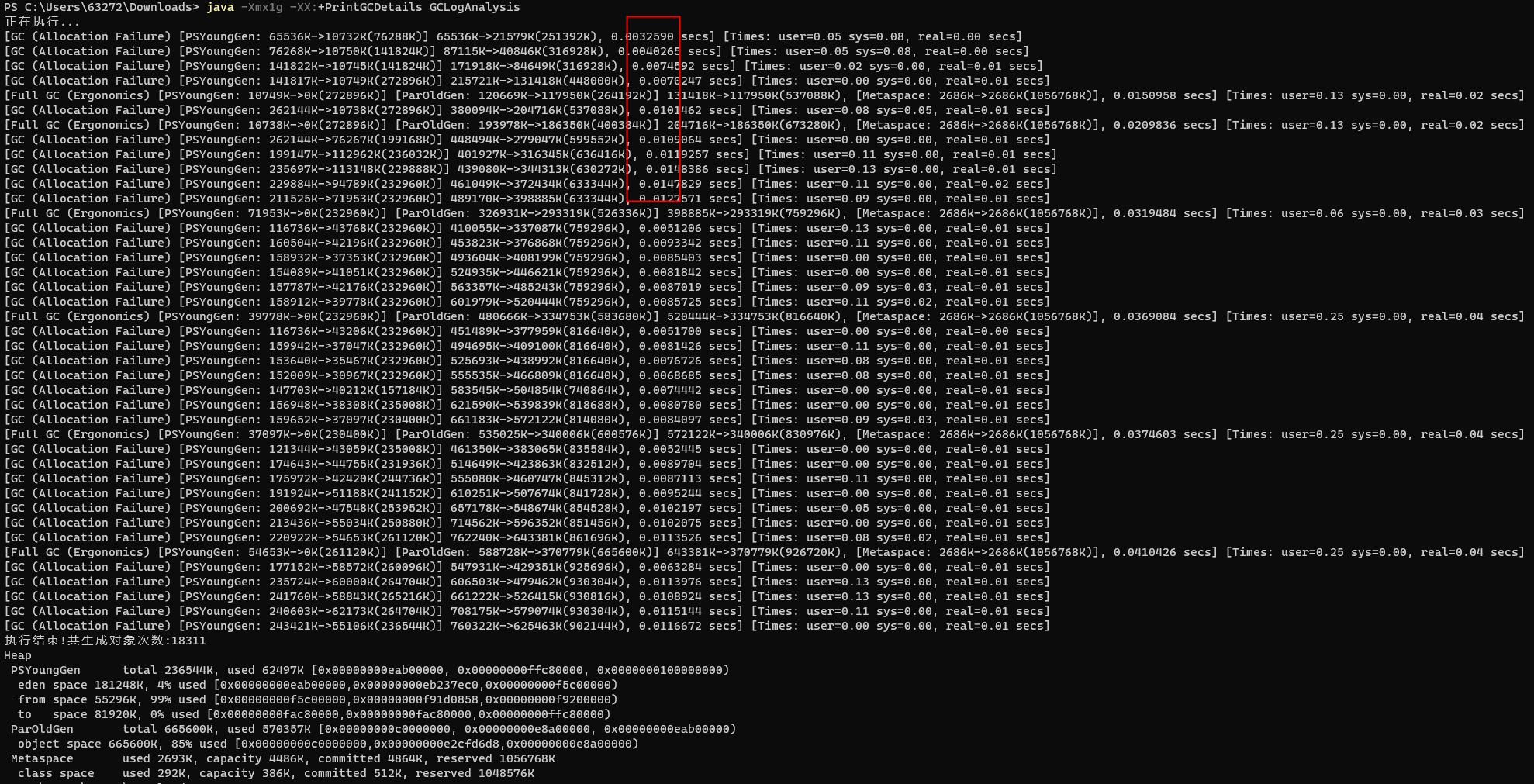
# 串行GC：

在使用串行GC执行GC代码的时候，也是像使用默认并行GC设置Xmx大小的时候，发现一个规律。使用串行GC的时候，发现一个比较有趣的事情，就是每次YoungGC的时候都是78%上下浮动的比例清空Young区。而并行GC，每次YoungGC的时候大概是65%上下浮动的清空。而这两个GC，每次YoungGC的时候，总的堆的清空比例基本都在20%上下浮动。而串行GC在FullGC的时候，每次总的堆的清空比例在30%以上。并行GC在30%左右，不会太往上浮动。

串行GC的Xmx1g执行情况



并行GC的Xmx1g执行情况

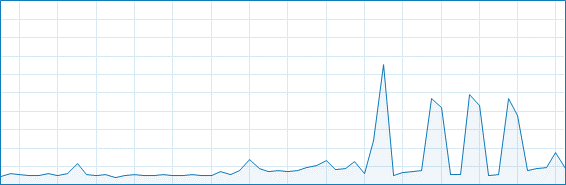


# 总结2：

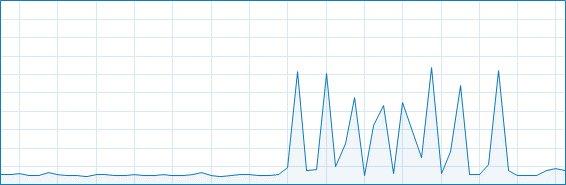
按照上面所述，串行GC的回收效率会比并行GC效率高。但是，代价是串行GC的STW时间会比并行GC的STW时间要高。因为串行GC在STW时间，只利用CPU其中一个核心去处理GC回收，而并行GC则是利用CPU多个核心去处理。也就相当于大家都去STW，串行GC是一个人工作，并行GC是多人工作，效率不一样。

# CMSGC：

并行GC执行CPU图



CMSGC执行CPU图



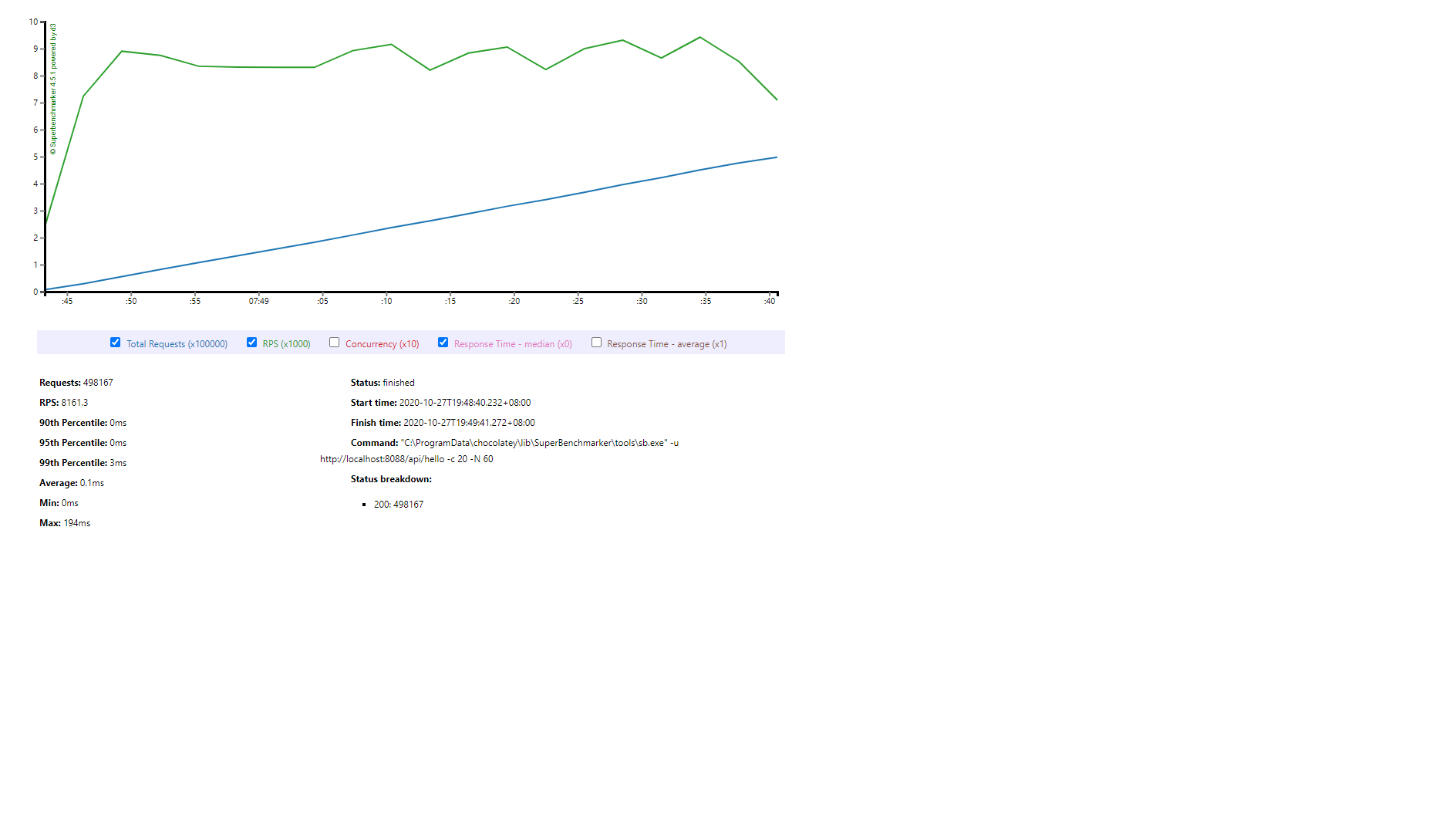
如图所示，CMSGC在新生代回收的时候与并行GC并无太大区别，而在对老年代进行处理的时候采用的是并发标记-清除算法，且对老年代进行回收的时候，标记-清除跟着应用线程一起走，很大一部分的工作提前做了，并不会拉起太高的CPU，这一点CMSGC比并行GC要好的多。对机器特别友好。

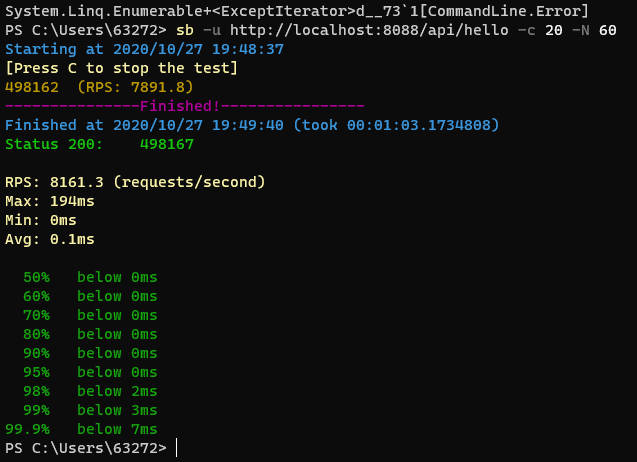
# G1GC：

使用G1GC后，Xmx在1g及以上的情况下，没有出现过FullGC，倒是出现过mixed，这个是GC混合模式，意思是不止是回收Young区也回收old区。以JDK8+Xmx1g为基准，串行GC、并行GC、CMSGC都出现过FullGC，只有G1GC未出现过，这与G1GC的设计有关。首先，使用G1GC，堆不再分新生代、老年代了，而是划分为多个区块，每个区块有可能这一刻是Young区，下一刻是Survivor区，也就是S0或者S1，后面可能是old区。这样的话，每次STW的时候，无论回收Young区还是old区，或者是S0&S1，只需要STW对应的区块即可，则不会影响到其他的区（PS：回收Young区的时候可能会影响到部分old区区块）

# 压测：

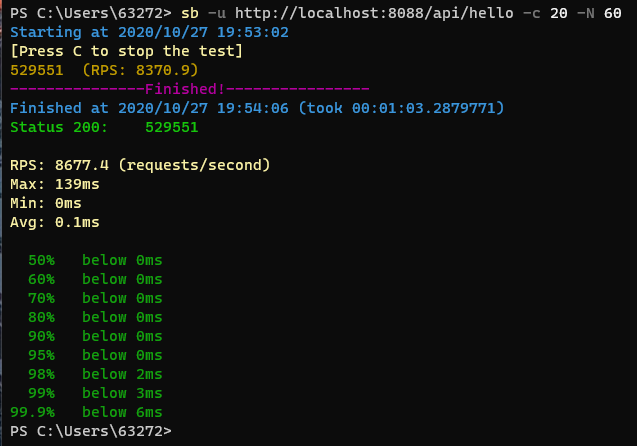
使用并行GC运行jar包压测结果

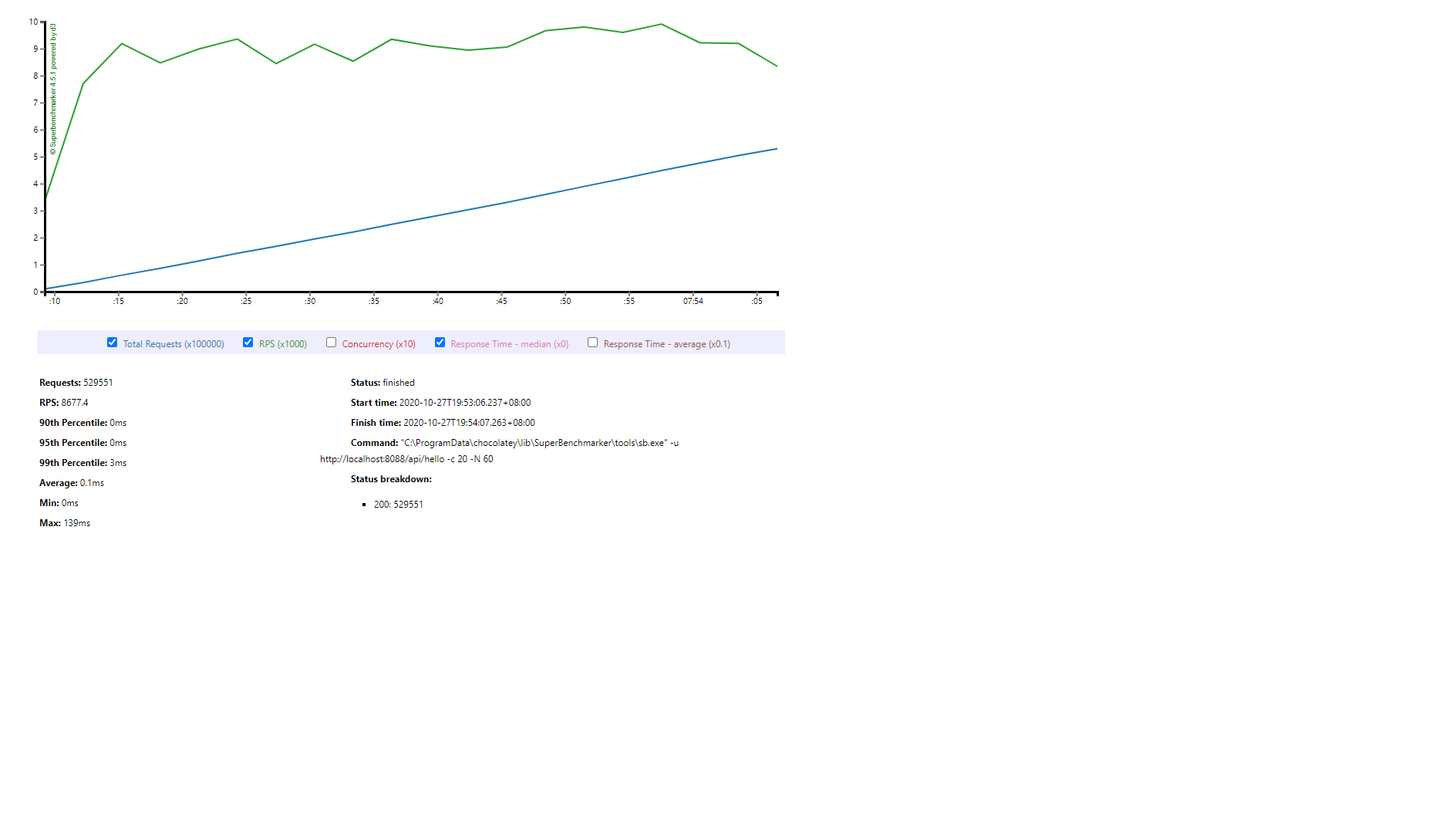




8161.3RPS

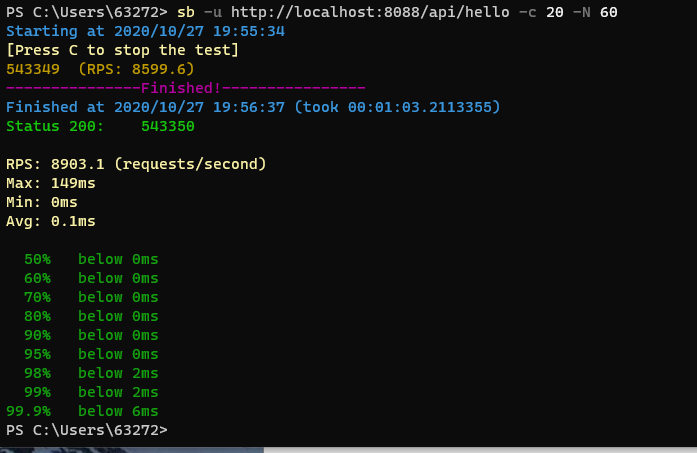
使用串行GC运行jar包压测结果

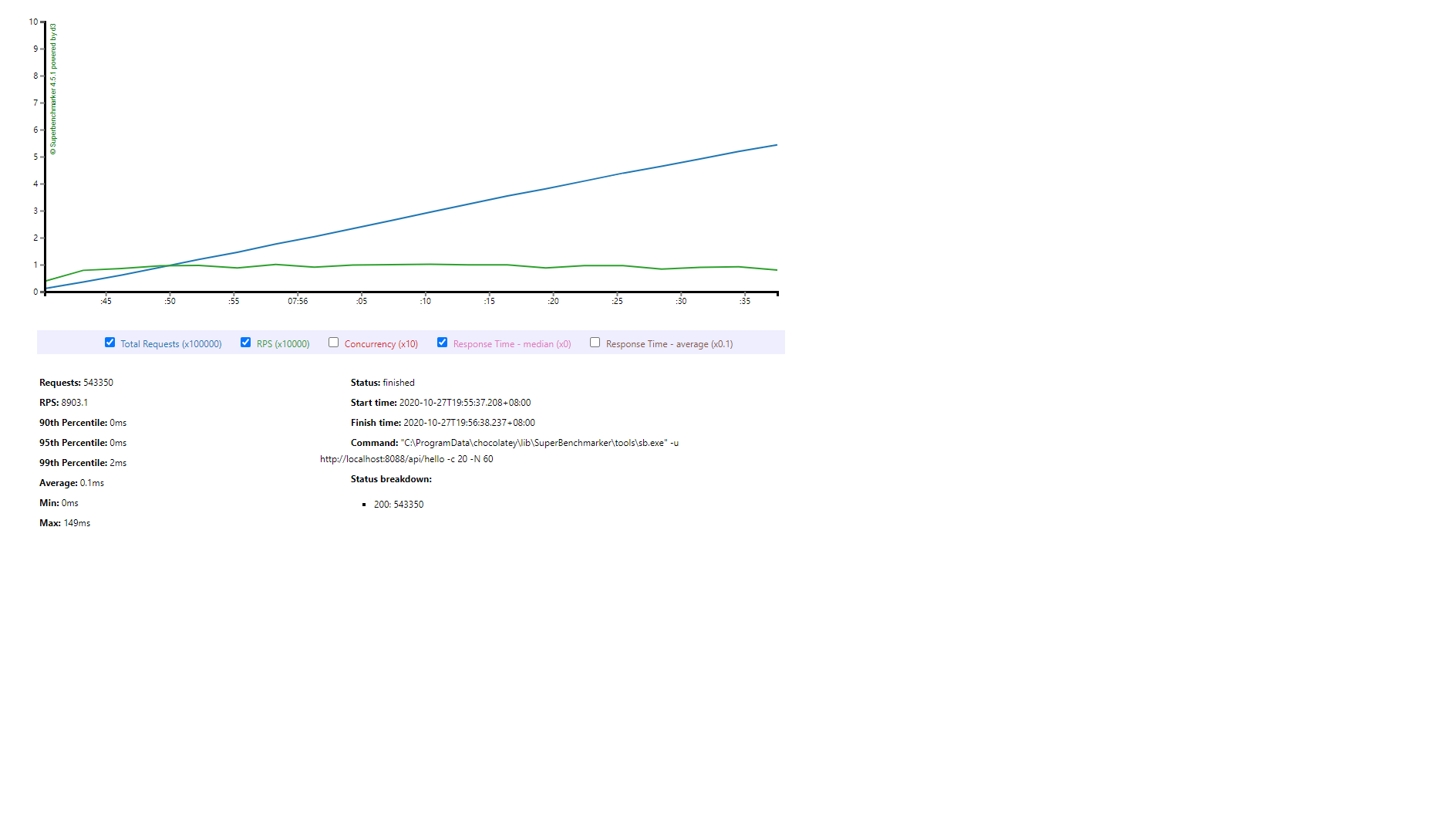




8677.4RPS

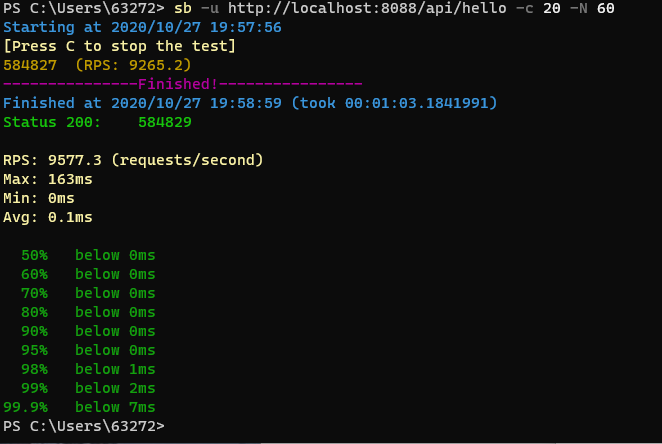
使用CMSGC运行jar包压测结果

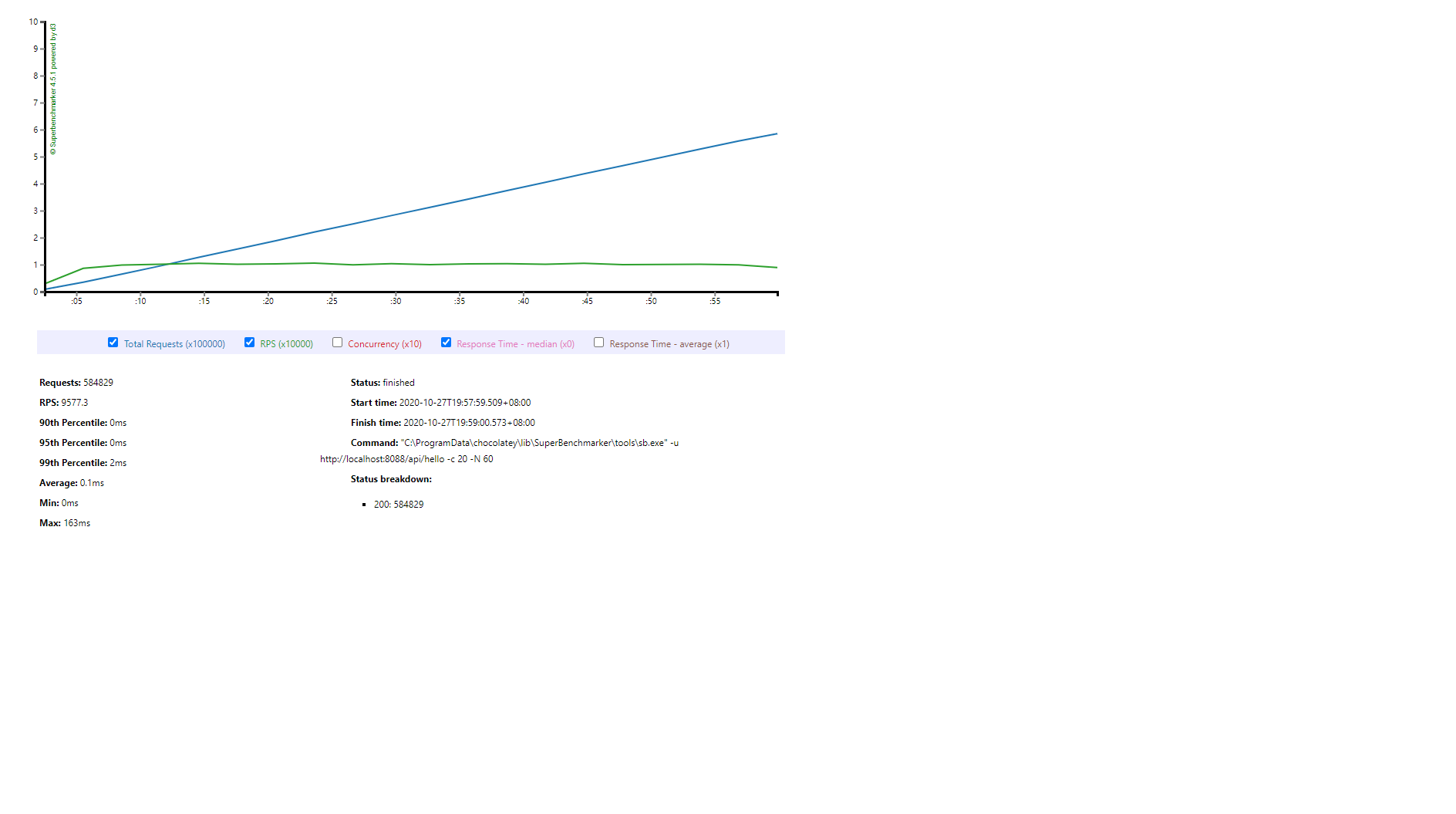




8903.1RPS

使用G1GC运行jar包压测结果





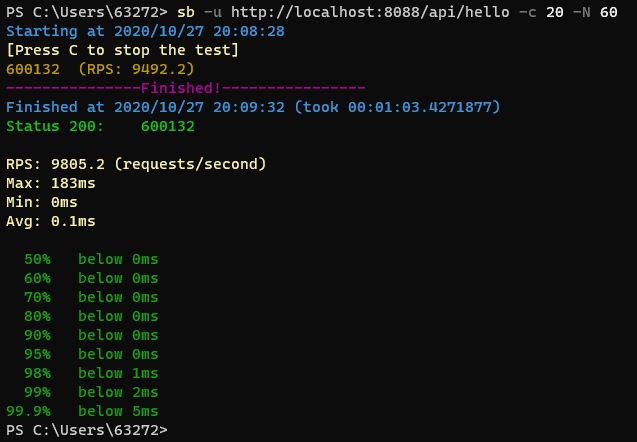
9577.3RPS

JDK8下，4个GC运行同一个老师提供的jar包，压测结果，截然不同

G1GC>CMSGC>串行GC>并行GC

这个时候我是有个疑惑的，为什么并行GC会比串行GC低。通过回忆老师课上的内容，想到了JDK8实际上默认的GC是ParallelScavenge（新生代）+Parallel Old（老年代）这两个，分管着年轻代与老年代。而这俩GC对于系统吞吐量支持的是比较好的，而对响应速度支持的不是那么的好。

串行GC虽然是利用CPU的单核来进行GC垃圾回收，但是，串行GC第一优先是响应速度。想到了这儿，我又回过头去，用ParNewGC运行了一下程序，结果如下：



9805.2RPS

看到这儿，果然并行GC还是可以完爆串行GC的，只不过启动的时候需要指定使用ParNewGC。

其实作业写到这儿，我有个疑问，为什么换成ParNewGC后，RPS会比G1GC要高，担心是随机的原因，经过好几次同样条件的压测，发现9800只是碰巧，平均值在8900~9000左右。并不会超过G1GC。

# 最终总结：

经过随机生成对象查看GC日志，压测接口，发现如果是写单独的客户端Java程序，使用串行GC较好，因为串行GC更多的是在Xmx堆小的情况下，执行效果会比较好。

而并行GC，最好是ParNew + Parallel Old 这两个结合使用或者与CMSGC结合使用。

实际上在JDK8的并行GC里面，一共有3个并行的垃圾回收器，分别是ParNew（新生代）、ParallelScavenge（新生代）、Parallel Old（老年代）。Sc与Old垃圾回收器是JDK8的默认垃圾回收器组合，很不推荐。

因为，按照压测数据的报告。发现用默认的情况下，甚至比不上串行GC，推荐ParNew+CMSGC。因为这两个GC，在对新生代和年轻代做回收的时候，不会在老年代GC的时候触发时间太长的STW。

但是，无论是串行、并行、CMS在GC的效率以及稳定情况下，都比不上G1GC。这是因为G1GC出现后，堆不再是原先的年轻代（Eden+S0+S1）与老年代了，G1GC在堆上做了区块的这么一个概念，在触发GC的时候，不会触发太多的FullGC，这是因为G1GC在日常GC的时候，会把部分年轻代+老年代混合的区块回收的时候，顺手把这个区块上老年代不需要的对象给回收（回收类型是mixed混合）。

但是，Java项目选用什么GC，还是要看这个Java项目所服务的业务，业务要求吞吐量较高，建议使用JDK8默认GC，要求响应速度（比如电商）G1GC是优先。

并没有什么所谓的最新的GC最牛逼，因为，从串行到G1GC、ZGC甚至是ShenandoahGC，它们都是在最原始的GC设计上，不断衍生出来的，只有最合适的，没有最牛的。