## 01-测试目的

通过模拟高并发的场景观察jvm内部内存、gc的情况，通过不同的jvm参数配置，观察不同配置下应用的性能。

## 02-测试工具

Grafana+Jmeter5.4.1+Prometheus2.15.1+Micrometer

## 03-测试环境

|  |  |
| --- | --- |
| 机器 | 配置 |
| hero应用服务器 | 4C8G |
| mysql数据库服务器 | 4C8G |
| Grafana、Jmeter、Prometheus部署服务器 | 4C8G |

## 04-测试场景

### 04-1默认参数启动

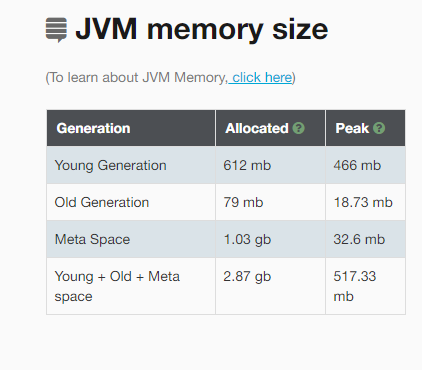
启动参数仅输出GC日志：

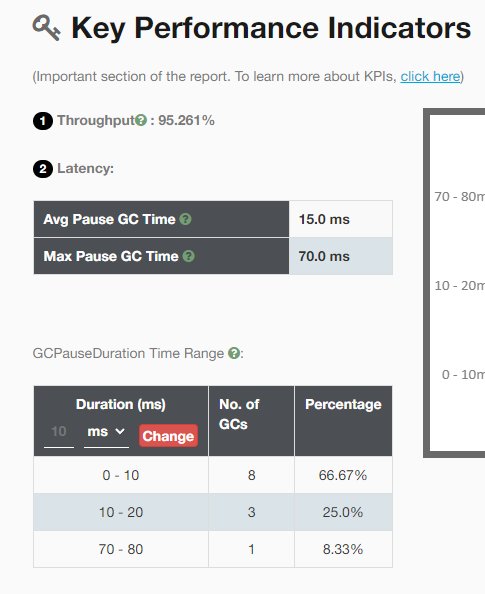
JAVA\_OPT="${JAVA\_OPT} -Xms1096m -Xmx1096m -Xmn408m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=128m"

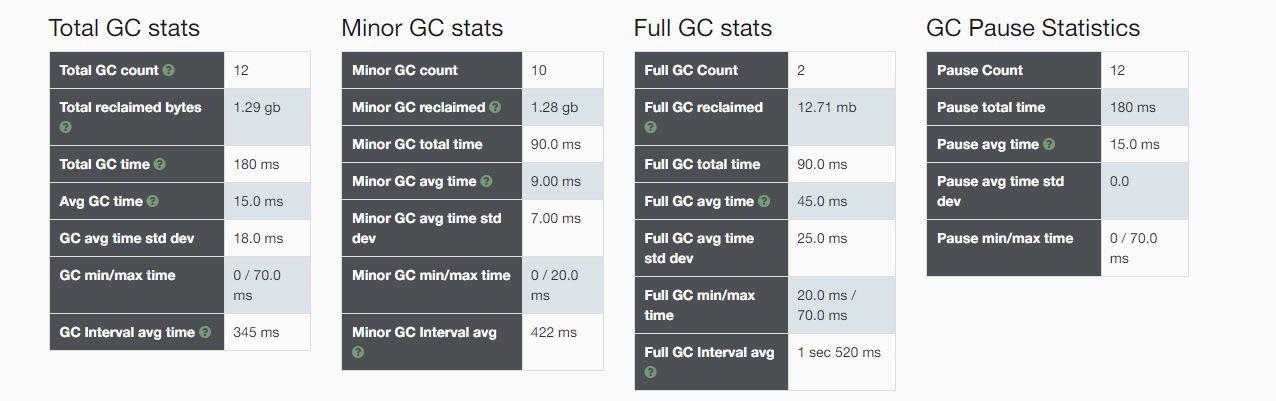
JAVA\_OPT="${JAVA\_OPT} -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -

XX:+PrintHeapAtGC -Xloggc:${BASE\_DIR}/logs/gc-best-heap-metaspace.log"

jvm内存、吞吐量与延时、fullgc次数如图：







通过gc日志分析，项目启动时，就分配了1.03gb的元空间，并且进行了两次fullGc,这是不合理的。

### 04-2选用合适的堆内存参数配置

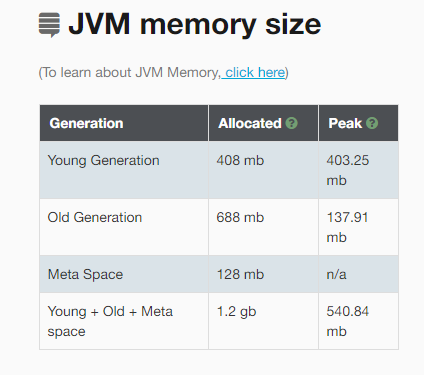
启动参数如下：

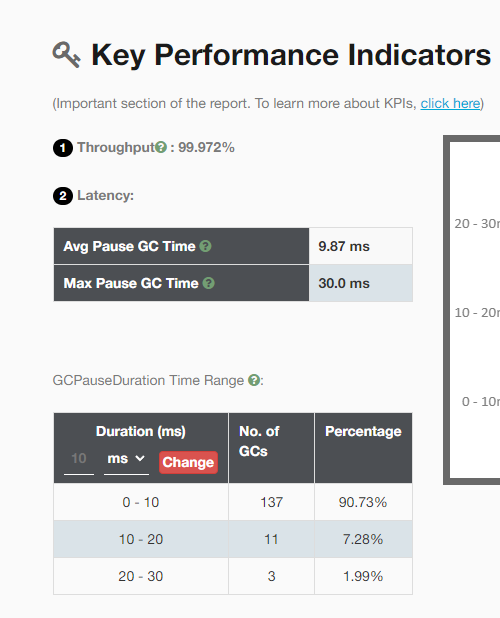
JAVA\_OPT="${JAVA\_OPT} -Xms1096m -Xmx1096m -Xmn408m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=128m"

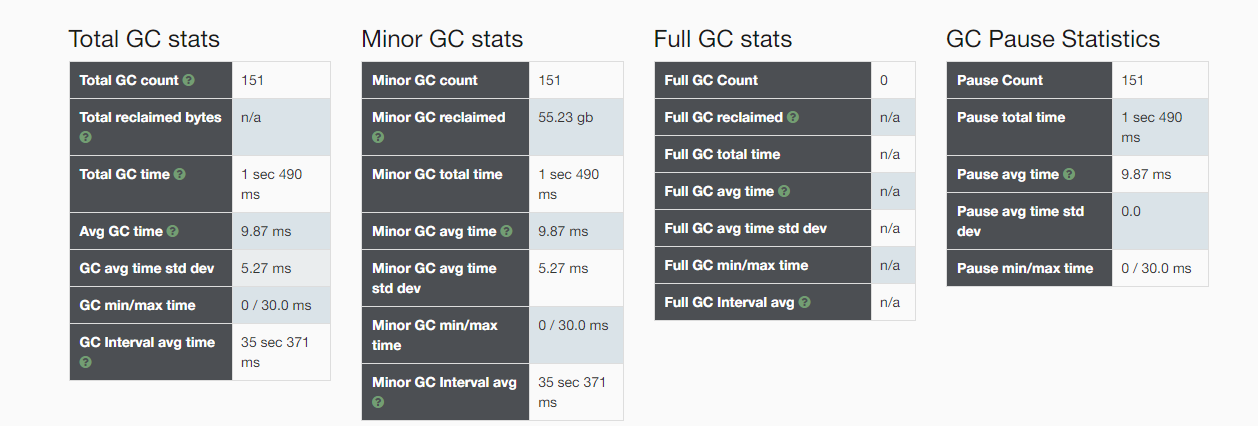
JAVA\_OPT="${JAVA\_OPT} -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -

XX:+PrintHeapAtGC -Xloggc:${BASE\_DIR}/logs/gc-best-heap-metaspace.log"

jvm内存、吞吐量与延时、fullgc次数如图：







**结论：通过合适的堆内存、元空间参数的配置，发现gc的吞吐量从95.261%上升到99.972%，平均时延由70ms降低到30ms。并且没有fullGc。**

### 04-3 配置PS+PO,模拟高负载

启动参数配置：

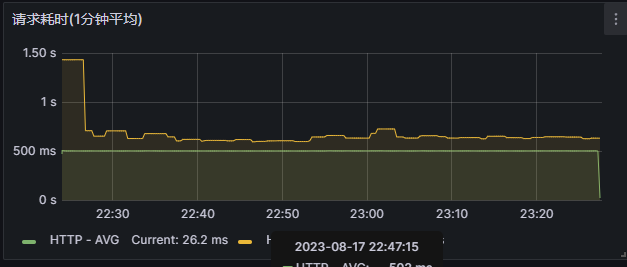
JAVA\_OPT="${JAVA\_OPT} -Xms256m -Xmx256m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=128m -Xss512k"

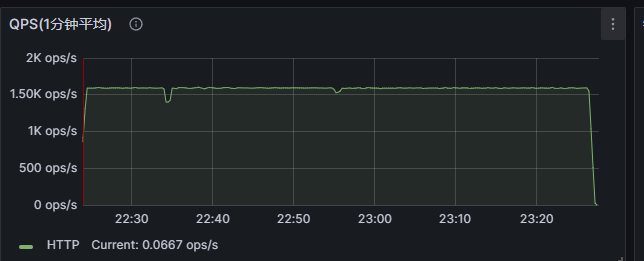
JAVA\_OPT="${JAVA\_OPT} -XX:+UseParallelGC -XX:+UseParallelOldGC "

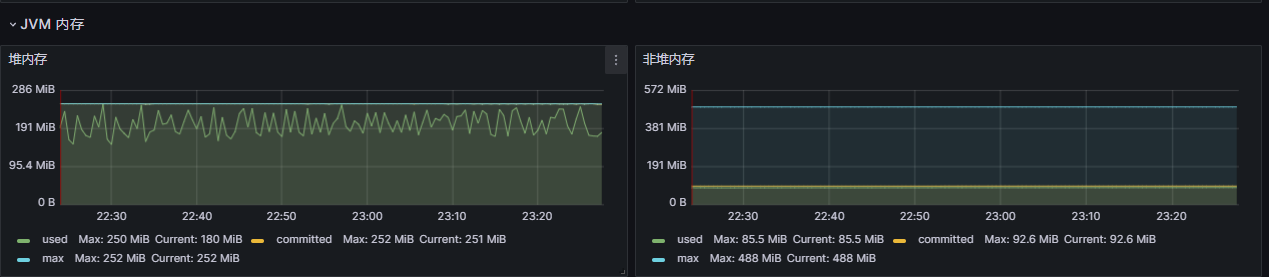
JAVA\_OPT="${JAVA\_OPT} -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -

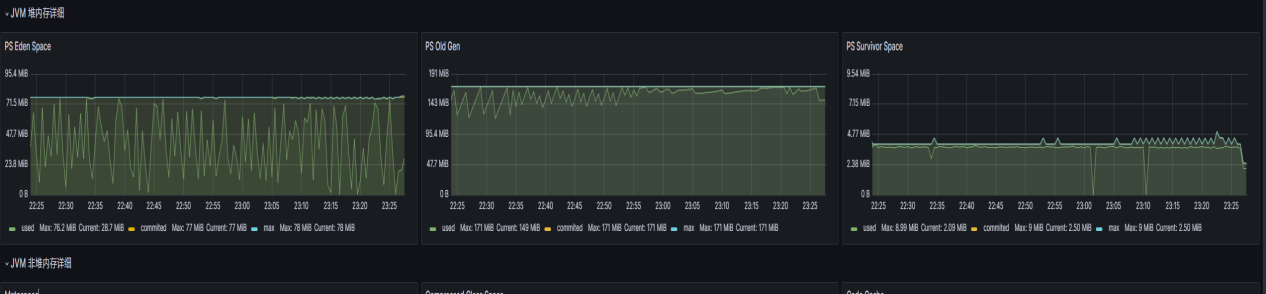
XX:+PrintHeapAtGC -Xloggc:${BASE\_DIR}/logs/gc-ps-po.log"

通过Grafana观测RT,TPS,JVM内存如图：

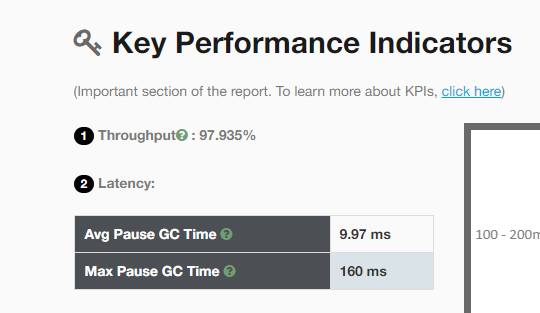


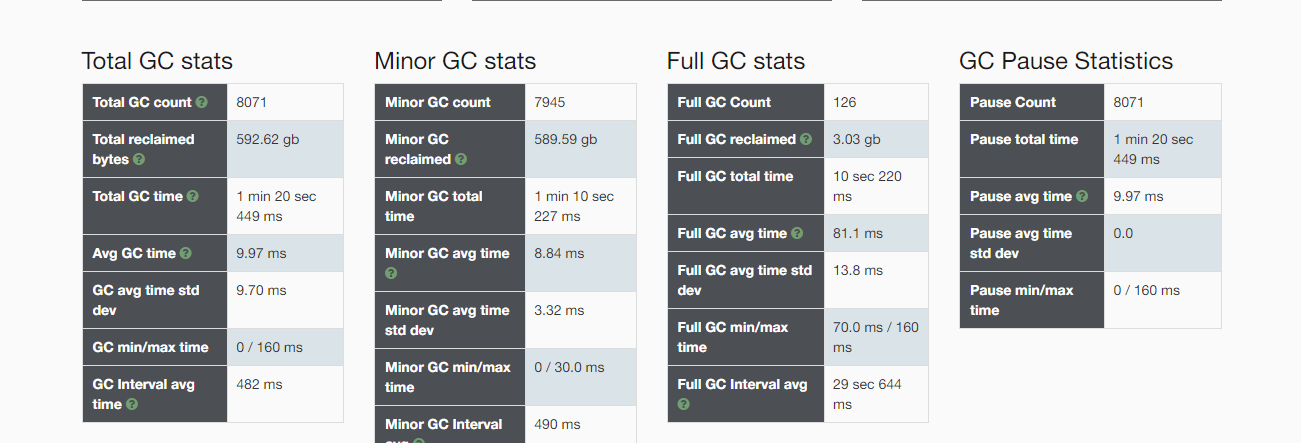






通过GcEasy观测垃圾回收的吞吐量和延时





### 04-4 配置parNew+CMS,模拟高负载

启动参数配置：

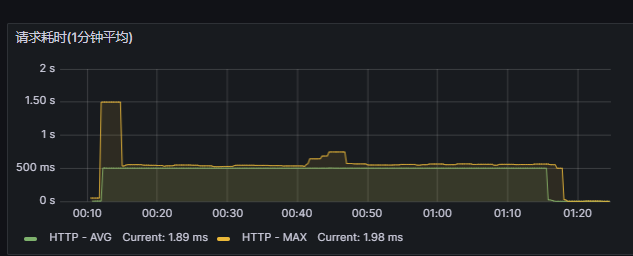
JAVA\_OPT="${JAVA\_OPT} -Xms256m -Xmx256m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=128m -Xss512k"

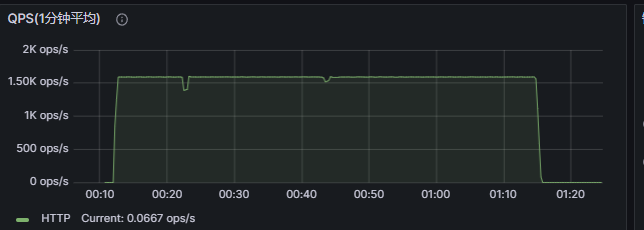
JAVA\_OPT="${JAVA\_OPT} -XX:+UseParNewGC -XX:+UseConcMarkSweepGC "

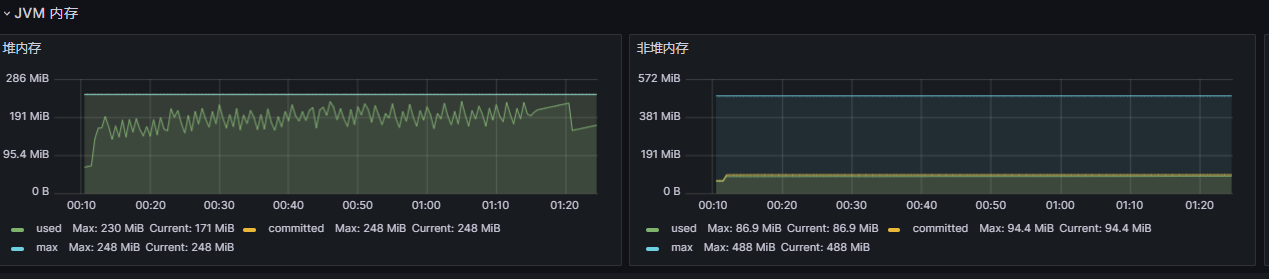
JAVA\_OPT="${JAVA\_OPT} -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -

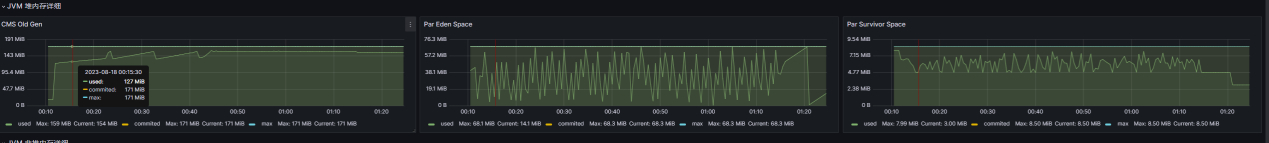
XX:+PrintHeapAtGC -Xloggc:${BASE\_DIR}/logs/gc-parnew-cms.log"

通过Grafana观测RT,TPS,JVM内存如图：

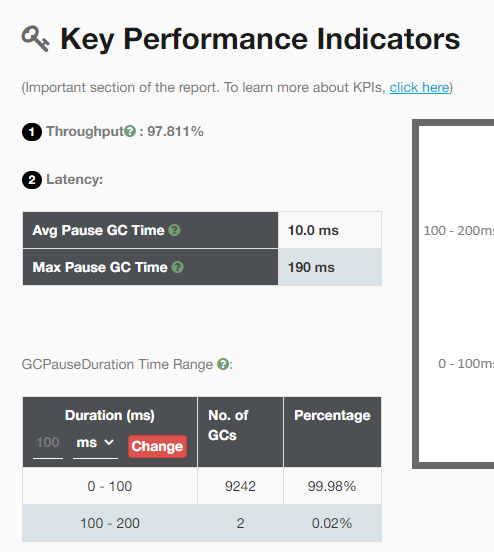


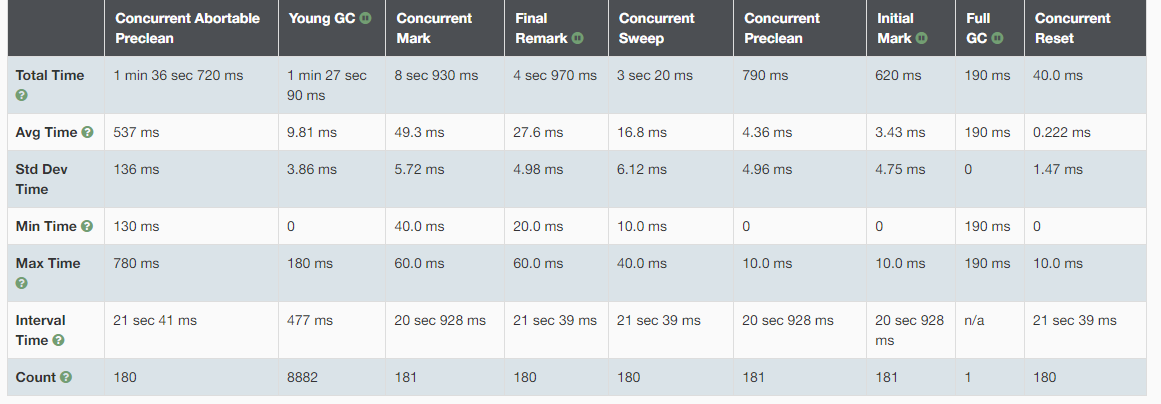






通过GcEasy观测垃圾回收的吞吐量和延时





### 04-5 配置G1，模拟高负载

启动参数配置：

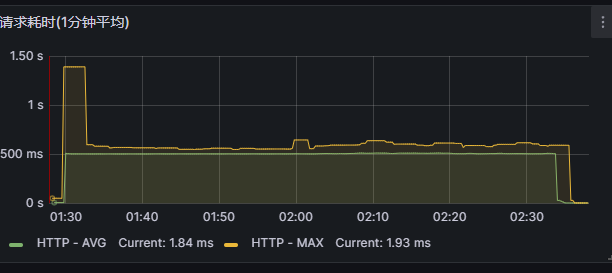
JAVA\_OPT="${JAVA\_OPT} -Xms256m -Xmx256m -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=128m -Xss512k"

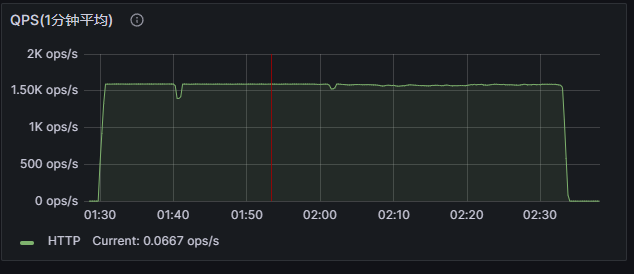
JAVA\_OPT="${JAVA\_OPT} -XX:+UseG1GC -XX:MaxGCPauseMillis=100"

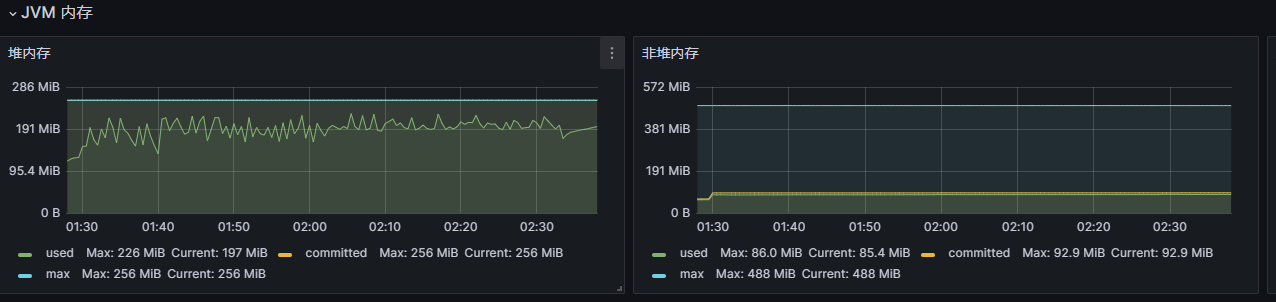
JAVA\_OPT="${JAVA\_OPT} -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -

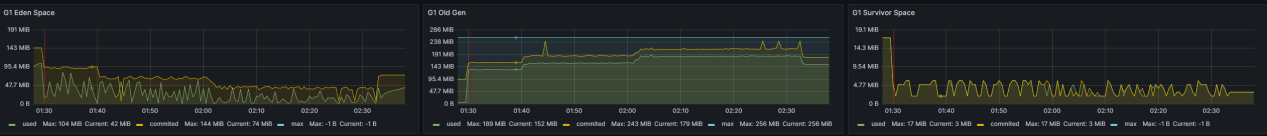
XX:+PrintHeapAtGC -Xloggc:${BASE\_DIR}/logs/gc-g-one.log"

通过Grafana观测RT,TPS,JVM内存如图：









通过GcEasy观测垃圾回收的吞吐量和延时

## 05 测试结果

通过04-1与04-2的测试对比，合适的堆内存、元空间配置能有效的减少FulGc的次数，提高垃圾回收的吞吐量，降低垃圾回收的时延。

通过04-3、04-4、04-5的测试对比，发现jvm的gc收集器性能都很强大，各个gc收集器表现都很好。Ps+po的吞吐量稍高，parnew+cms能将RT的最大值波动降下来，g1在各个方面都表现很好。