Week2 作业一

通过分配大对象，分析各种GC日志

使用机器的基本信息：

```
Java HotSpot(TM) 64-Bit Server VM (25.261-b12) for bsd-amd64 JRE (1.8.0_261-b12), built on Jun 18 2020 06:38:55 by "java_re" with gcc 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.11.45.5) Memory: 4k page, physical 12582912k(2479692k free)
```

1. 分析SerialGC日志

- $java -XX:+UseSerialGC -Xms512m -Xmx512m -XX:+PrintGCDetails GCLogAnalysis

结果分析：在1秒钟内共生成了9616个对象，触发了8次Young GC，7次Full GC。因触发GC导致用户线程STW总时间约为540ms，平均GC导致的STW时间约为54ms。

因日志过长，因此将截取第一次发生Young GC、第一次发生Full GC的日志进行分析。

- 第一次Young GC：使用的是单线程的STW垃圾收集器，标记复制算法。

```
[GC (Allocation Failure) 2020-10-26T20:19:47.897-0800: 0.165: [DefNew: 139776K->17472K(157248K), 0.0230622 secs] 139776K->44181K(506816K), 0.0231517 secs] [Times: user=0.02 sys=0.01, real=0.03 secs]
```

可以看出，在这次垃圾收集之前，堆内存总的使用量为139776K,与年轻代使用量相同，因此可以推出GC之前老年代空间的使用量为0。GC之后年轻代的使用率约为11%，堆内存使用率约为9%。

GC前后对比，年轻代提升到老年代的对象占用了44181-17472=26708k的空间，GC使用的时间为30ms，属于延迟较高的范畴。

- 第一次Full GC：使用的是单线程的STW垃圾收集器，标记清除整理算法。

```
[GC (Allocation Failure) 2020-10-26T20:19:48.209-0800: 0.477: [DefNew: 157245K->157245K(157248K), 0.0000209 secs]2020-10-26T20:19:48.209-0800: 0.477: [Tenured: 299815K->271683K(349568K), 0.0482291 secs] 457060K->271683K(506816K), [Metaspace: 2725K->2725K(1056768K)], 0.0483414 secs] [Times: user=0.05 sys=0.00, real=0.05 secs]
```

可以看出，因为内存分配失败，先发生了一次YGC，这次YGC时间极短，且年轻代空间无任何变化，说明此次GC对年轻代无法再进行更多处理，需要触发FGC。 在FGC中，清理了28000k的老年代空间，GC之后老年代的使用率约为77.7%，数值较高，约有10%的下降，年轻代空间得到整理全部清空。内存整体使用率从90%下降至53%，个人认为在空间整理上仍然是一次比较有效的GC。从日志上表现为此次FGC后，再次发生GC时只发生了YGC。

此外，MetaSpace无明显变化，GC耗时约为50ms，约为YGC的两倍。此次GC后老年代存活对象约为270M，如果内存扩大十倍，GC后老年代内存使用量也扩大约10倍，耗时将达到500ms甚至更高，将带来非常明显的系统停顿。

- 执行结果

```
Heap
 def new generation   total 157248K, used 34721K [0x00000007a0000000,
0x00000007aaaa0000, 0x00000007aaaa0000)
  eden space 139776K,  24% used [0x00000007a0000000, 0x00000007a21e87e0,
0x00000007a8880000)
  from space 17472K,   0% used [0x00000007a8880000, 0x00000007a8880000,
0x00000007a9990000)
  to   space 17472K,   0% used [0x00000007a9990000, 0x00000007a9990000,
0x00000007aaaa0000)
 tenured generation   total 349568K, used 349432K [0x00000007aaaa0000,
0x00000007c0000000, 0x00000007c0000000)
 the space 349568K,  99% used [0x00000007aaaa0000, 0x00000007bffde1b0,
0x00000007bffde200, 0x00000007c0000000)
 Metaspace       used 2732K, capacity 4486K, committed 4864K, reserved 1056768K
 class space    used 296K, capacity 386K, committed 512K, reserved 1048576K
```

可以看到，老年代空间使用率几乎达到了100%，剩余的年轻代空间约为120MB。

2. 分析Parallel GC日志

- $java -XX:+UseParallelGC -Xms512m -Xmx512m -XX:+PrintGCDetails GCLogAnalysis

结果分析：使用4个线程，在1秒钟内共生成了9701个对象，触发了20次Young GC，8次Full GC。因触发GC导致用户线程STW总时间约为440ms，平均GC导致的STW时间约为44ms。最后剩余的对空间约为130Mb。

可以看出，ParallelGC与SerialGC对象分配的速度相似，总的STW时间有缩短，平均STW时间也有缩短。

因日志过长，因此将截取第一次发生FullGC前的Young GC、第一次发生Full GC的日志进行分析。

- 第一次发生FullGC前的Young GC：使用的是并行的STW垃圾收集器，标记复制算法。

```
[GC (Allocation Failure) [PSYoungGen: 100258K->21714K(116736K)] 380429K-
>339554K(466432K), 0.0121481 secs] [Times: user=0.02 sys=0.02, real=0.01 secs]
```

GC后年轻代的使用率约为18%，整个堆内存的使用率为73%，这个使用比例进入了一个不算低的范围。此时的使用时间，real= (user + sys) / 4(默认线程数)，可以看出并行gc相比于串行gc缩短了stw的时间，性能有较大提升。

在GC之前，老年代的使用量为280171K；GC结束之后，年轻代使用量减少了78544K，总的堆内存使用量减少了40875K，那么可以推算出有37669K的对象从年轻代提升到老年代。 老年代的使用量为317840K，老年代的大小为349696K，老年代的使用率约为91%，可以看出老年代已经快满了，紧接着触发了下面的Full GC。

- 第一次发生FullGC：并行STW垃圾收集器ParOldGen，使用标记-清除-整理算法。

```
[Full GC (Ergonomics) [PSYoungGen: 21714K->0K(116736K)] [ParOldGen: 317839K-
>245866K(349696K)] 339554K->245866K(466432K), [Metaspace: 2725K-
>2725K(1056768K)], 0.0357045 secs] [Times: user=0.13 sys=0.01, real=0.03 secs]
```

首先进行了YGC，年轻代空间得到完全回收，在ParallelGC中Full GC年轻代的结果基本都是如此。 后进行了FGC，GC之前老年代使用率为91%，GC之后老年代使用率约为70%，有明显的下降。 FGC之前堆内存的使用率约为73%，FGC之后堆内存的使用率约为52%，也有明显的下降。 此外，Metaspace没有回收任何对象。GC持续的时间仍然在多线程环境下得到缩短，串行gc可能需要140ms停顿，在并行gc中只需要约30ms。

- 最终内存结果

```
Heap
 PSYoungGen      total 116736K, used 2561K [0x00000007b5580000,
0x00000007c0000000, 0x00000007c0000000)
  eden space 58880K, 4% used
[0x00000007b5580000,0x00000007b5800438,0x00000007b8f00000)
  from space 57856K, 0% used
[0x00000007bc780000,0x00000007bc780000,0x00000007c0000000)
  to   space 57856K, 0% used
[0x00000007b8f00000,0x00000007b8f00000,0x00000007bc780000)
 ParOldGen       total 349696K, used 331752K [0x00000007a0000000,
0x00000007b5580000, 0x00000007b5580000)
  object space 349696K, 94% used
[0x00000007a0000000,0x00000007b43fa298,0x00000007b5580000)
 Metaspace       used 2732K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 296K, capacity 386K, committed 512K, reserved 1048576K
```

可以看出，与串行gc类似，老年代空间使用率仍然超过了90%，年轻代则仍然只有4%，剩余内存约为130MB。

3. 分析CMS GC

- $ java -XX:+UseConcMarkSweepGC -XX:ConcGCThreads=4 -XX:MaxGCPauseMillis=50 -Xms512m -Xmx512m -XX:+PrintGCDetails GCLogAnalysis

结果分析：CMS并发线程数为4，在60s内生成对象10798个，GC触发的频率高于并行GC。因触发GC导致用户线程STW总时间约为300ms，平均GC导致的STW时间约为30ms，相比于ParallelGC有所下降。与并行GC相比，生成对象的速率稍高。最后剩余的堆空间约为130Mb。

因日志过长，因此将截取一次Young GC、第一次发生Full GC的日志进行分析。

- Young GC：使用的是并行的STW垃圾收集器，标记复制算法。

```
[GC (Allocation Failure) 2020-10-26T20:17:20.974-0800: 0.304: [ParNew: 157248K-
>17470K(157248K), 0.0229117 secs] 302873K->208300K(506816K), 0.0230029 secs]
[Times: user=0.08 sys=0.02, real=0.02 secs]
```

GC后年轻代的使用率约为11%，整个堆内存的使用率为41%，这个使用比例尚可。 老年代的使用量为190830K，老年代的大小为349568K，老年代的使用率约为54.5%。

- Full GC：并发，标记清除算法

```
// 初始标记
2020-10-26T20:17:21.209-0800: 0.539: [GC (CMS Initial Mark) [1 CMS-initial-mark:
281640K(349568K)] 302300K(506816K), 0.0001532 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs]
// 并发标记
2020-10-26T20:17:21.209-0800: 0.539: [CMS-concurrent-mark-start]
2020-10-26T20:17:21.211-0800: 0.541: [CMS-concurrent-mark: 0.002/0.002 secs]
[Times: user=0.00 sys=0.00, real=0.01 secs]
//并发预清理
2020-10-26T20:17:21.211-0800: 0.541: [CMS-concurrent-preclean-start]
2020-10-26T20:17:21.212-0800: 0.542: [CMS-concurrent-preclean: 0.000/0.000 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
2020-10-26T20:17:21.212-0800: 0.542: [CMS-concurrent-abortable-preclean-start]
```

```
2020-10-26T20:17:21.227-0800: 0.557: [GC (Allocation Failure) 2020-10-
26T20:17:21.227-0800: 0.557: [ParNew: 157247K->17471K(157248K), 0.0117681 secs]
438887K->341620K(506816K), 0.0118471 secs] [Times: user=0.05 sys=0.00, real=0.01
secs]
2020-10-26T20:17:21.239-0800: 0.570: [CMS-concurrent-abortable-preclean:
0.001/0.028 secs] [Times: user=0.07 sys=0.00, real=0.02 secs]
// 最终标记：会STW
2020-10-26T20:17:21.239-0800: 0.570: [GC (CMS Final Remark) [YG occupancy: 26983
K (157248 K)]2020-10-26T20:17:21.239-0800: 0.570: [Rescan (parallel) , 0.0002386
secs]2020-10-26T20:17:21.240-0800: 0.570: [weak refs processing, 0.0000117
secs]2020-10-26T20:17:21.240-0800: 0.570: [class unloading, 0.0001974 secs]2020-
10-26T20:17:21.240-0800: 0.570: [scrub symbol table, 0.0002785 secs]2020-10-
26T20:17:21.240-0800: 0.571: [scrub string table, 0.0001330 secs][1 CMS-remark:
324149K(349568K)] 351132K(506816K), 0.0009137 secs] [Times: user=0.00 sys=0.00,
real=0.01 secs]
// 并发清除
2020-10-26T20:17:21.240-0800: 0.571: [CMS-concurrent-sweep-start]
2020-10-26T20:17:21.241-0800: 0.571: [CMS-concurrent-sweep: 0.000/0.000 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
// 并发重置
2020-10-26T20:17:21.241-0800: 0.571: [CMS-concurrent-reset-start]
2020-10-26T20:17:21.241-0800: 0.572: [CMS-concurrent-reset: 0.000/0.000 secs]
[Times: user=0.00 sys=0.00, real=0.00 secs]
```

- 最终堆内存情况

```
Heap
 par new generation   total 157248K, used 70388K [0x00000007a0000000,
0x00000007aaaa0000, 0x00000007aaaa0000)
  eden space 139776K,  50% used [0x00000007a0000000, 0x00000007a44bd0c0,
0x00000007a8880000)
  from space 17472K,   0% used [0x00000007a8880000, 0x00000007a8880000,
0x00000007a9990000)
  to   space 17472K,   0% used [0x00000007a9990000, 0x00000007a9990000,
0x00000007aaaa0000)
 concurrent mark-sweep generation total 349568K, used 345517K
[0x00000007aaaa0000, 0x00000007c0000000, 0x00000007c0000000)
 Metaspace       used 2732K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 296K, capacity 386K, committed 512K, reserved 1048576K
```

可以看出，与并行gc类似，老年代空间使用率仍然超过了90%，年轻代接近50%，剩余内存约为80MB。

4. 分析G1 GC

- $ java -XX:+UseG1GC -XX:ConcGCThreads=4 -XX:MaxGCPauseMillis=50 -Xms512m -Xmx512m -XX:+PrintGCDetails GCLogAnalysis

结果分析：G1并发线程数为4，在60s内生成对象10946个，GC触发的频率高于并行GC。因触发GC导致用户线程STW总时间约为90ms，平均GC导致的STW时间约为9ms，相比于CMS有显著下降，两者内存的分配速度相似。

因日志过长，因此将截取一次Young GC、第一次发生Full GC的日志进行分析。

- Young GC：标记复制算法。

```
[GC pause (G1 Evacuation Pause) (young), 0.0112341 secs]
   [Parallel Time: 10.7 ms, GC Workers: 4]
   .....
[Eden: 105.0M(105.0M)->0.0B(192.0M) Survivors: 12.0M->15.0M Heap:
281.9M(512.0M)->190.5M(512.0M)]
 [Times: user=0.02 sys=0.02, real=0.01 secs]
```

从一次YGC中可以看出，Eden区105M被清空，Survivor区增长了3M，整个堆内存减少了91.4M，可以算出约有10.6M的对象晋升到了老年代。

- Full GC：标记-清除-整理算法

```
// 纯年轻代转移模式暂停
[GC pause (G1 Evacuation Pause) (young) (to-space exhausted), 0.0044006 secs]
   [Parallel Time: 3.8 ms, GC Workers: 4]
// 纯年轻代转移模式暂停：混合模式
[GC pause (G1 Evacuation Pause) (mixed), 0.0055030 secs]
// 初始标记：STW
[GC pause (G1 Humongous Allocation) (young) (initial-mark), 0.0011497 secs]
   [Parallel Time: 0.8 ms, GC Workers: 4]
   ....
[Eden: 1024.0K(62.0M)->0.0B(64.0M) Survivors: 4096.0K->4096.0K Heap:
335.7M(512.0M)->335.1M(512.0M)]
 [Times: user=0.00 sys=0.00, real=0.00 secs]
// Root区扫描
2020-10-26T20:14:05.139-0800: 0.472: [GC concurrent-root-region-scan-start]
2020-10-26T20:14:05.139-0800: 0.472: [GC concurrent-root-region-scan-end,
0.0000638 secs]
// 并发标记
2020-10-26T20:14:05.139-0800: 0.472: [GC concurrent-mark-start]
2020-10-26T20:14:05.140-0800: 0.474: [GC concurrent-mark-end, 0.0015017 secs]
// 再次标记：STW
2020-10-26T20:14:05.140-0800: 0.474: [GC remark 2020-10-26T20:14:05.140-0800:
0.474: [Finalize Marking, 0.0001425 secs] 2020-10-26T20:14:05.141-0800: 0.474:
[GC ref-proc, 0.0000278 secs] 2020-10-26T20:14:05.141-0800: 0.474: [Unloading,
0.0005786 secs], 0.0013199 secs]
 [Times: user=0.01 sys=0.00, real=0.00 secs]
// 清理
2020-10-26T20:14:05.142-0800: 0.475: [GC cleanup 348M->347M(512M), 0.0004062
secs]
 [Times: user=0.00 sys=0.00, real=0.00 secs]
2020-10-26T20:14:05.142-0800: 0.476: [GC concurrent-cleanup-start]
2020-10-26T20:14:05.142-0800: 0.476: [GC concurrent-cleanup-end, 0.0000071 secs]
```

- 最终堆内存情况

```
Heap
 garbage-first heap   total 524288K, used 345082K [0x00000007a0000000,
0x00000007a0101000, 0x00000007c0000000)
  region size 1024K, 1 young (1024K), 0 survivors (0K)
 Metaspace       used 2732K, capacity 4486K, committed 4864K, reserved 1056768K
  class space    used 296K, capacity 386K, committed 512K, reserved 1048576K
```

整个堆内存使用率约为64%，相比于CMS和并行GC来说是一个较低的使用率。

Week2 作业2

使用SuperbenchMarker压测网关接口

1. 使用G1GC，分析此时GC的性能

内存1Gib

- $ java -server -XX:+UseG1GC -XX:ConcGCThreads=3 -XX:MaxGCPauseMillis=50 -Xms1g -Xmx1g -jar gateway-server-0.0.1-SNAPSHOT.jar io.github.kimmking.gateway.server.ApiServerApplication

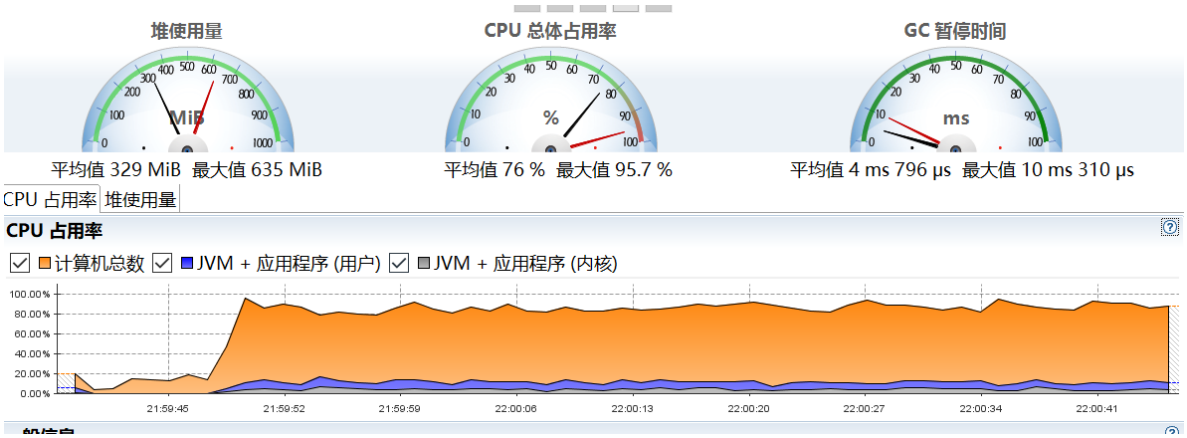
// 使用40个线程，压测约60s

- $ sb -u http://localhost:8088/api/hello -c 40 -N 60

```
Finished at 2020/10/28 22:00:46 (took 00:01:05.4848246)
Status 200:    189732

RPS: 3105.8 (requests/second)
Max: 118ms
Min: 0ms
Avg: 1.8ms

  50%   below 0ms
  60%   below 0ms
  70%   below 0ms
  80%   below 2ms
  90%   below 7ms
  95%   below 11ms
  98%   below 18ms
  99%   below 23ms
99.9%   below 41ms
```



可以看出，G1GC的请求平均响应时间为1.8ms，STW的平均时间约为4.8ms。

使用CMS GC，分析此时GC的性能

内存1Gib

- $ java -server -XX:+UseConcMarkSweepGC -XX:ConcGCThreads=3 -XX:MaxGCPauseMillis=50 -Xms1g -Xmx1g -jar gateway-server-0.0.1-SNAPSHOT.jar io.github.kimmking.gateway.server.ApiServerApplication

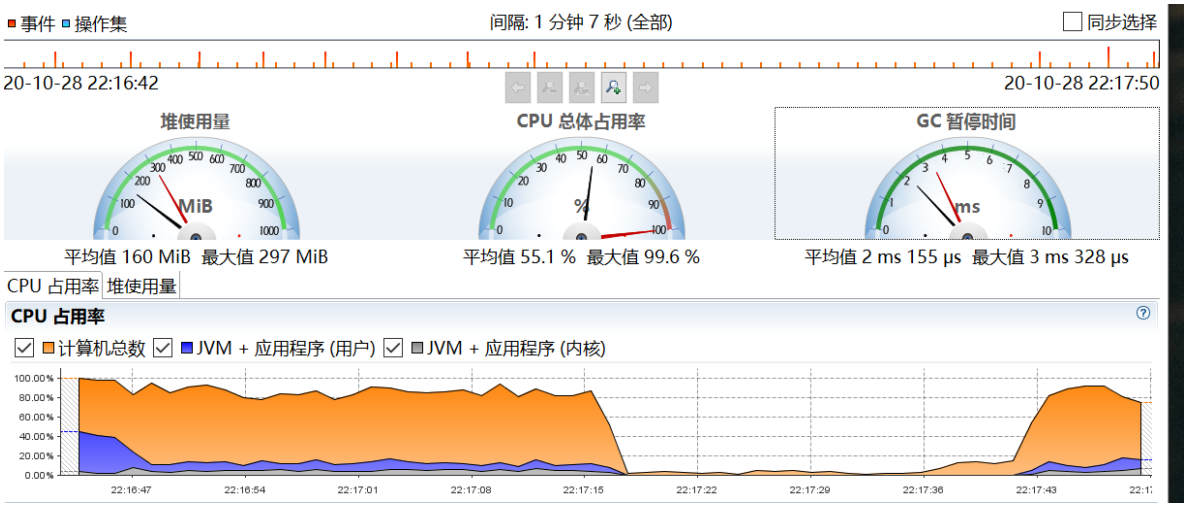
// 使用40个线程，压测约60s

- $ sb -u http://localhost:8088/api/hello -c 40 -N 60

```
Finished at 2020/10/28 22:07:52 (took 00:01:05.0299134)
Status 200:    177183

RPS: 2898.4 (requests/second)
Max: 534ms
Min: 0ms
Avg: 2.1ms

  50%   below 0ms
  60%   below 0ms
  70%   below 0ms
  80%   below 2ms
  90%   below 7ms
  95%   below 12ms
  98%   below 19ms
  99%   below 25ms
99.9%   below 48ms
```



CMS GC的平均响应时间达到2.1ms，STW的平均时间约为2ms的3倍，即达到6ms

使用并行 GC，分析此时GC的性能

内存1Gib

- $ java -server -XX:+UseParallelGC -XX:ParallelGCThreads=3 -XX:MaxGCPauseMillis=50 -Xms1g -Xmx1g -jar gateway-server-0.0.1-SNAPSHOT.jar io.github.kimmking.gateway.server.ApiServerApplication

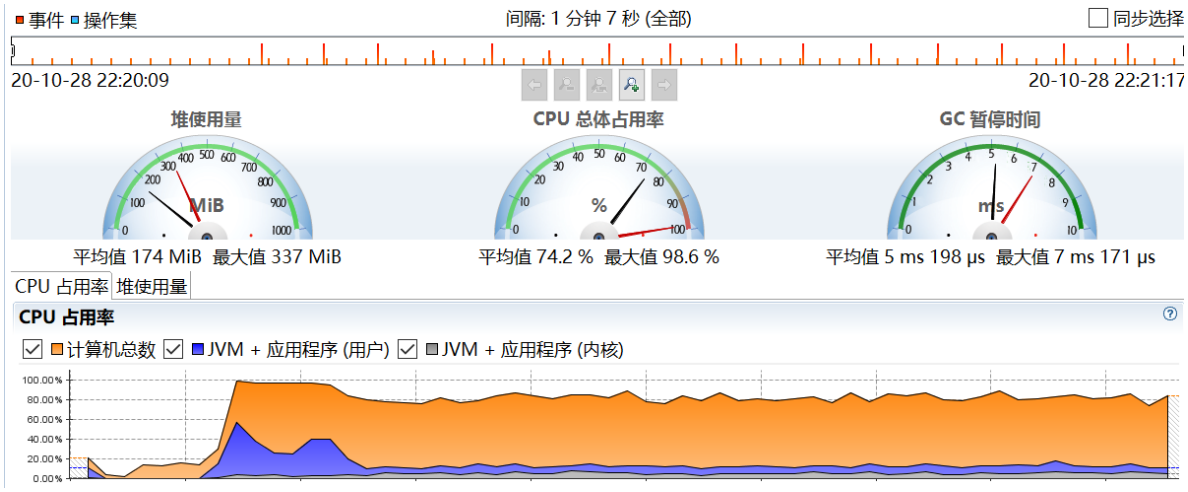
// 使用40个线程，压测约60s

- $ sb -u http://localhost:8088/api/hello -c 40 -N 60

```
Finished at 2020/10/28 22:21:17 (took 00:01:04.8646969)
Status 200:    194370

RPS: 3176.7 (requests/second)
Max: 313ms
Min: 0ms
Avg: 1.7ms

  50%    below 0ms
  60%    below 0ms
  70%    below 0ms
  80%    below 1ms
  90%    below 6ms
  95%    below 10ms
  98%    below 17ms
  99%    below 23ms
99.9%    below 42ms
```



并行GC的响应时间约为1.7ms，平均STW的时间达到5.2ms

使用串行GC，分析此时GC的性能

内存1Gib

- $ java -server -XX:+UseSerialGC -Xms1g -Xmx1g -jar gateway-server-0.0.1-SNAPSHOT.jar io.github.kimmking.gateway.server.ApiServerApplication

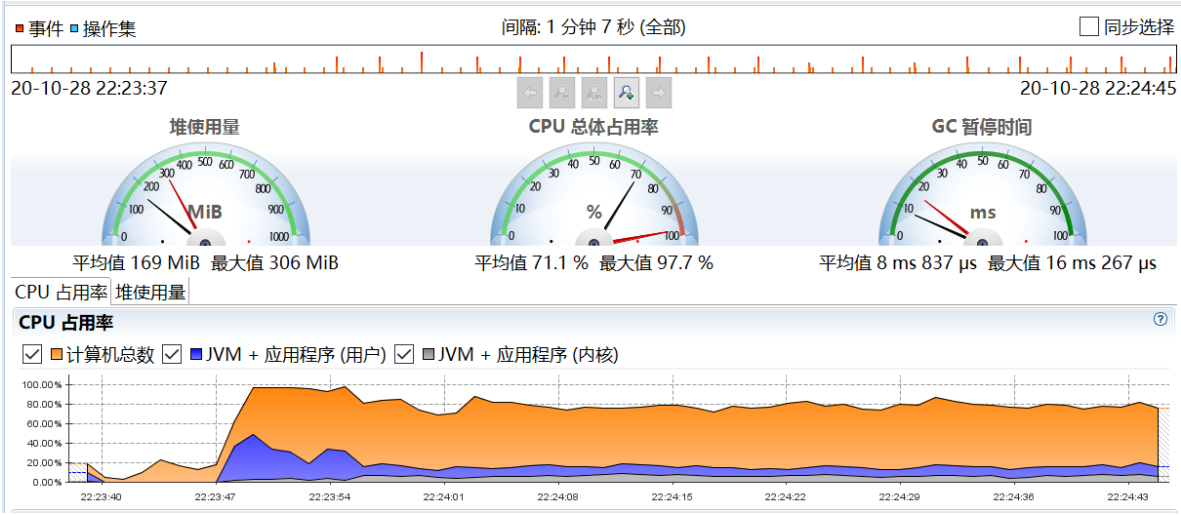
// 使用40个线程，压测约60s

- $ sb -u http://localhost:8088/api/hello -c 40 -N 60

```
Finished at 2020/10/28 22:24:46 (took 00:01:04.9493660)
Status 200:    219086

RPS: 3584.3 (requests/second)
Max: 370ms
Min: 0ms
Avg: 1.2ms

  50%    below 0ms
  60%    below 0ms
  70%    below 0ms
  80%    below 0ms
  90%    below 3ms
  95%    below 8ms
  98%    below 14ms
  99%    below 22ms
99.9%    below 45ms
```



串行GC的平均响应时间约为1.2ms， 但平均STW时间达到很高的8.9ms

对不同GC的总结

1. 串行GC
   - 单线程垃圾收集器，不能进行并行处理，年轻代和老年代都会触发STW，停止所有应用线程
   - 使用场景：在只有几百MB堆内存的JVM下可以使用
2. 并行GC：
   - 效率高，最常调整的是线程数， -XX:ParallelGCThreads=N ，默认是cpu核心数。
   - 处 理GC的线程是多线程，与业务线程没有关系。
   - 使用场景： 并行垃圾收集器适用于多核服务器，主要目标是增加吞吐量，因为对系统资源的有效使 用，能达到更高的吞吐量。 在GC期间，所有CPU内核都在并行清理垃圾，所以总暂停时间更短； 在两次GC周期的间隔期，没有GC线程在运行，不会消耗任何系统资源。
3. CMS GC：
   - 避免在老年代垃圾收集时出现长时间的卡顿，最大可能的并发
   - 使用场景：如果是多核CPU，主要调优目标是降低GC停顿导致的系统延迟，那么CMS是个明智的选择
4. G1 GC：
   - 垃圾优先，哪 一块的垃圾最多就优先清理它；
   - 目标

        - 将 STW 停顿的时间和分布，变成可预期且可配置的
        - 内存划分改变，内存切成多个小块，没有碎片
        - 停止回收：默认堆大小5%
    - 使用场景：可以做到吞吐量与效率的平衡，介于并行GC和CMS GC之间。

Week2 作业三

通过OKHttpClient来访问本机端口

```
import okhttp3.OkHttpClient;
import okhttp3.Request;
import okhttp3.Response;

import java.io.EOFException;
import java.io.IOException;
import java.io.Reader;
import java.nio.CharBuffer;

public class OKHttpClient {

    public static void main(String[] args) throws IOException {
        OkHttpClient client = new OkHttpClient();
        Request request = new Request.Builder()
                .get()
                .url("http://localhost:8801")
                .addHeader("Connection", "close")
                .build();

        Response response = client.newCall(request).execute();
        if (!response.isSuccessful()) {
            throw new IOException("Connection fail, Unexpected code " +
response.code());
        }

        Reader reader = response.body().charStream();
        CharBuffer buffer = CharBuffer.allocate(10);
        reader.read(buffer);

        buffer.flip();
        char[] res = new char[buffer.limit()];
        buffer.get(res, 0, buffer.limit());
        System.out.println(res);
    }
}
```