GC 情报

content.log

时长: 49 min 41 sec 828 ms

内存泄漏

Our analysis tells that your application is suffering from memory leak. It can cause OutOfMemoryError, JVM to freeze, poor response time and high CPU consumption.

Read our recommendations to resolve memory leak (./gc-recommendations/memory-leak-solution.jsp)

较差吞吐量

Our analysis tells that your application is spending too much time on GC. **22.58%** of time is spent on GC. Too much GC activity degrades response time + consumes CPU. It's ideal to keep GC time under **10.0%**.

Read our recommendations to increase throughput (./gc-recommendations/through-solution.jsp)

建议

(**CAUTION:** Please do thorough testing before implementing below recommendations.)

✓ **10 min 44 sec 659 ms** of GC pause time is triggered by **'G1 Evacuation Pause'** event. This GC is triggered when copying live objects from one set of regions to another set of regions. When Young generation regions are only copied then Young GC is triggered. When both Young + Tenured regions are copied, Mixed GC is triggered..

Solution:
1. Evacuation failure might happen because of over tuning. So eliminate all the memory related properties and keep only min and max heap and a realistic pause time goal (i.e. Use only -Xms, -Xmx and a pause time goal -XX:MaxGCPauseMillis). Remove any additional heap sizing such as -Xmn, -XX:NewSize, -XX:MaxNewSize, -XX:SurvivorRatio, etc.

2. If the problem still persists then increase JVM heap size (i.e. -Xmx).

3. If you can't increase the heap size and if you notice that the marking cycle is not starting early enough to reclaim the old generation then reduce -XX:InitiatingHeapOccupancyPercent. The default value is 45%. Reducing the value will start the marking cycle earlier. On the other hand, if the marking cycle is starting early and not reclaiming, increase the -XX:InitiatingHeapOccupancyPercent threshold above the default value.

4. You can also increase the value of the '-XX:ConcGCThreads' argument to increase the number of parallel marking threads. Increasing the concurrent marking threads will make garbage collection run fast.

5. Increase the value of the '-XX:G1ReservePercent' argument. Default value is 10%. It means the G1 garbage collector will try to keep 10% of memory free always. When you try to increase this value, GC will be triggered earlier, preventing the Evacuation pauses. Note: G1 GC caps this value at 50%.

✓ **6 sec 30 ms** of GC pause time is triggered by **'G1 Humongous Allocation'** event. Humongous allocations are allocations that are larger than 50% of the region size in G1. Frequent humongous allocations can cause couple of performance issues:
1. If the regions contain humongous objects, space between the last humongous object in the region and the end of the region will be unused. If there are multiple such humongous objects, this unused space can cause the heap to become fragmented.
2. Until Java 1.8u40 reclamation of humongous regions were only done during full GC events. Where as in the newer JVMs, clearing humongous objects are done in cleanup phase.

Solution:
You can increase the G1 region size so that allocations would not exceed 50% limit. By default region size is calculated during startup based on the heap size. It can be overriden by specifying '-XX:G1HeapRegionSize' property. Region size must be between 1 and 32 megabytes and has to be a power of two. Note: Increasing region size is sensitive change as it will reduce the number of regions. So before increasing new region size, do thorough testing.

✓ **80.0 ms** of GC pause time is triggered by **'Metadata GC Threshold'** event. This type of GC event is triggered under two circumstances:
1. Configured metaspace size is too small than the actual requirement
2. There is a classloader leak (very unlikely, but possible).

Solution:

You may consider setting '-XX:MaxMetaspaceSize' to a higher value. If this property is not present already please configure it. Setting these arguments to a higher value will reduce 'Metadata GC Threshold' frequency. If you still continue to see 'Metadata GC Threshold' event reported, then you need to capture heap dump from your application and analyze it. You can learn how to do heap dump analysis from this article. (https://blog.heaphero.io/2018/03/27/how-to-diagnose-memory-leaks/)

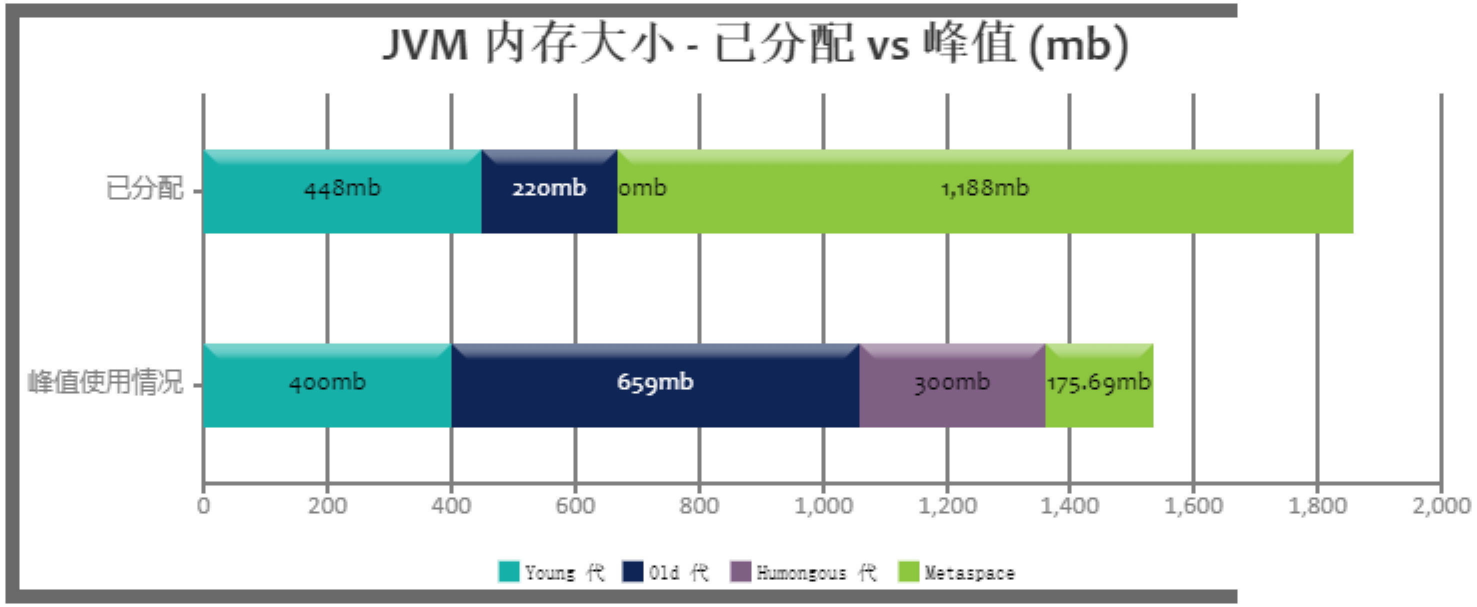
✓ It looks like you are using G1 GC algorithm. If you are running on Java 8 update 20 and above, you may consider passing **-XX:+UseStringDeduplication** to your application. It will remove duplicate strings in your application and has potential to improve overall application's performance. You can learn more about this property in this article. (./gc-recommendations/stringdeduplication-solution.jsp?)

✓ This application is using the G1 GC algorithm. If you are looking to tune G1 GC performance even further, here are the important G1 GC algorithm related JVM arguments (./gc-recommendations/important-g1-gc-arguments.jsp?)

JVM 内存大小

(To learn about JVM Memory, click here (https://www.youtube.com/watch?v=uJLOlCuOR4k))

代	已分配	峰值
Young 代	448 mb	400 mb
Old 代	220 mb	659 mb
Humongous	n/a	300 mb
Metaspace	1.16 gb	175.69 mb
Young + Old + Metaspace	1.81 gb	841.69 mb



关键性能指标（KPI）

(重要报告部分。如需了解更多有关 KPI 的信息, 请点击此处 (https://blog.gceasy.io/2016/10/01/garbage-collection-kpi/))

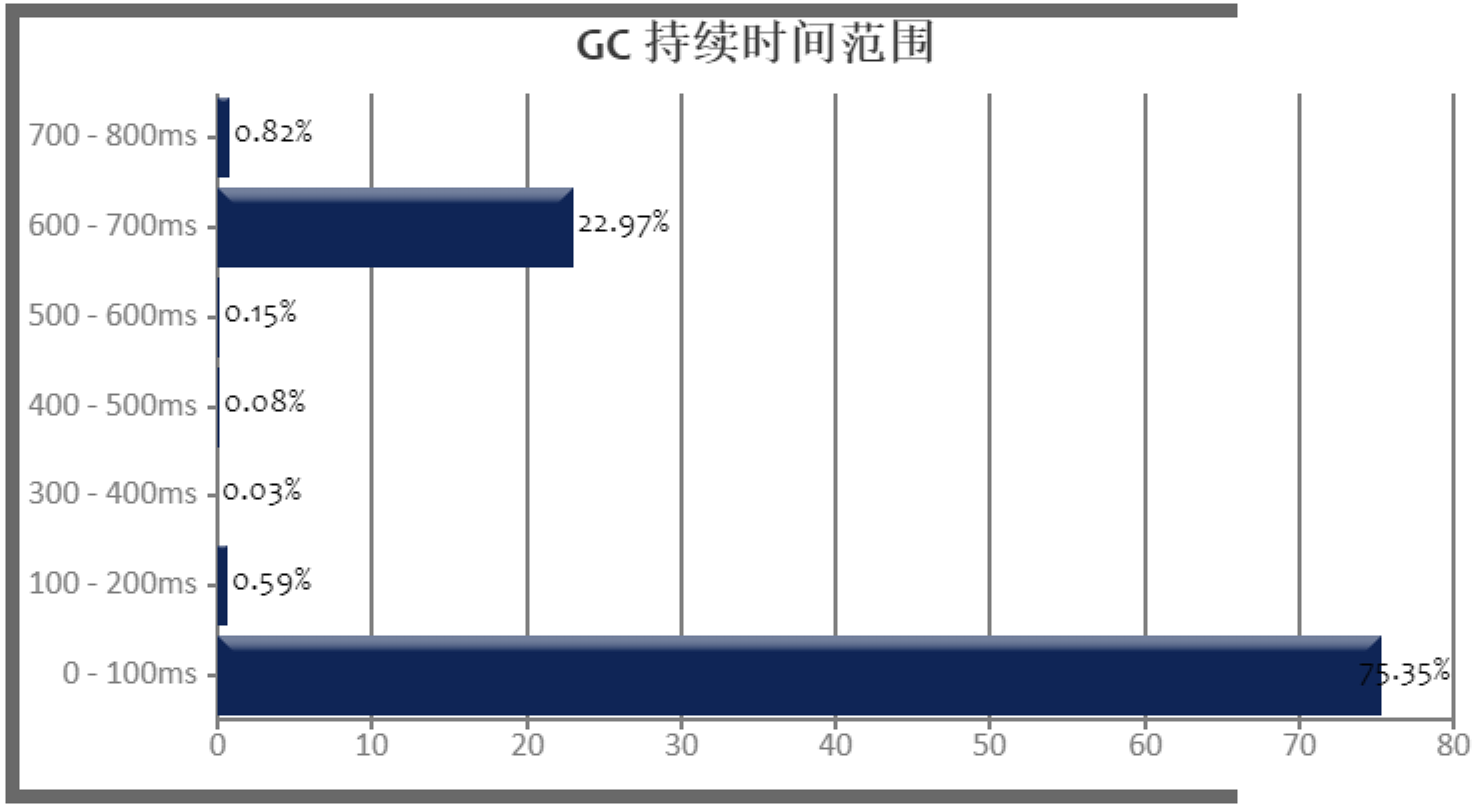
1 **吞吐❓ : 77.421%**

2 **延迟:**

平均停顿 GC 时间 ❓	**174 ms**
最大停顿 GC 时间 ❓	**760 ms**

GC停顿持续时间范围 ❓:

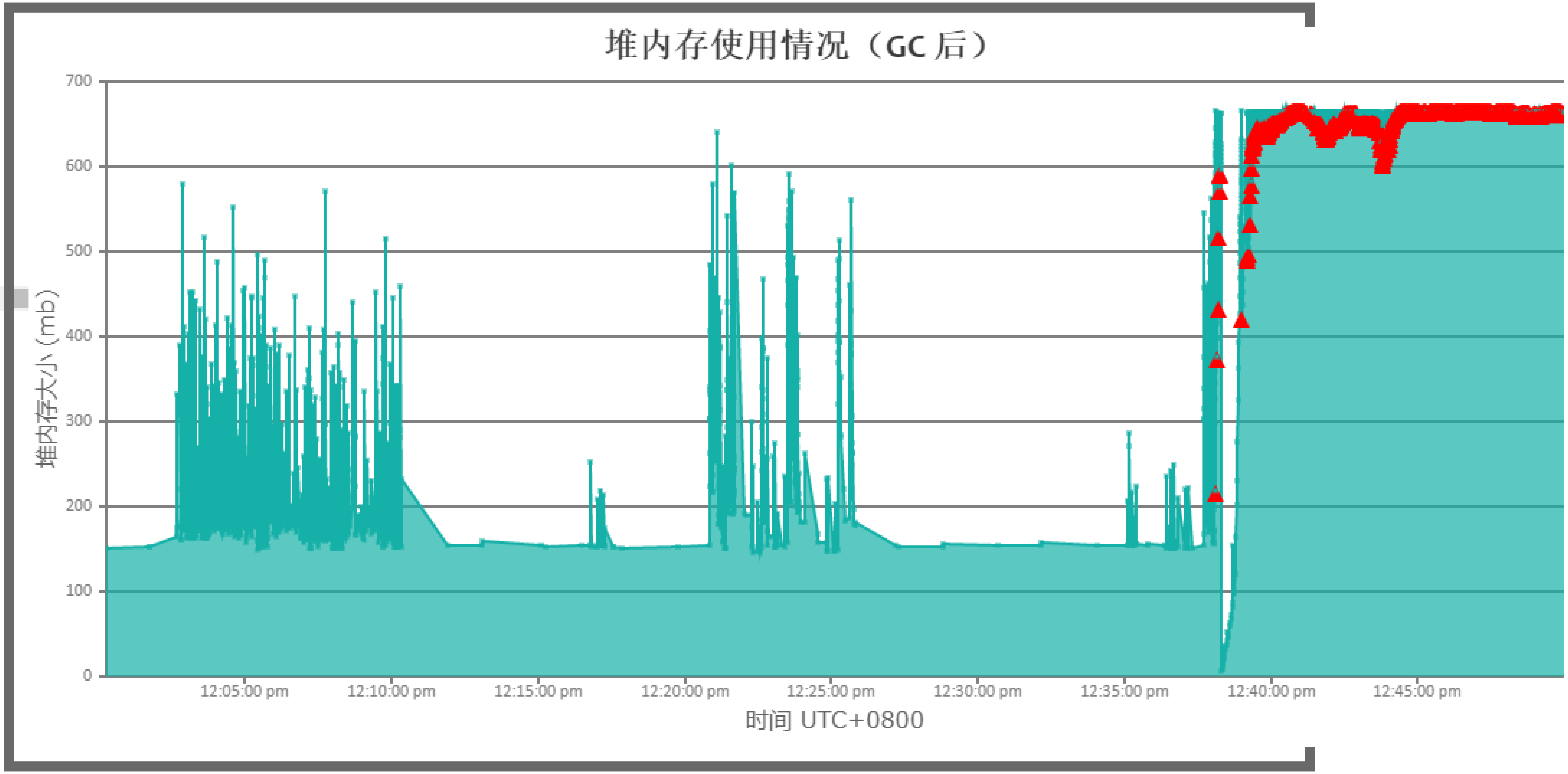
Duration (ms) 100 **ms ⌄** Change	**No. of GCs**	**Percenta**
0 - 100	2923	75.35%
100 - 200	23	0.59%
300 - 400	1	0.03%
400 - 500	3	0.08%
500 - 600	6	0.15%
600 - 700	891	22.97%

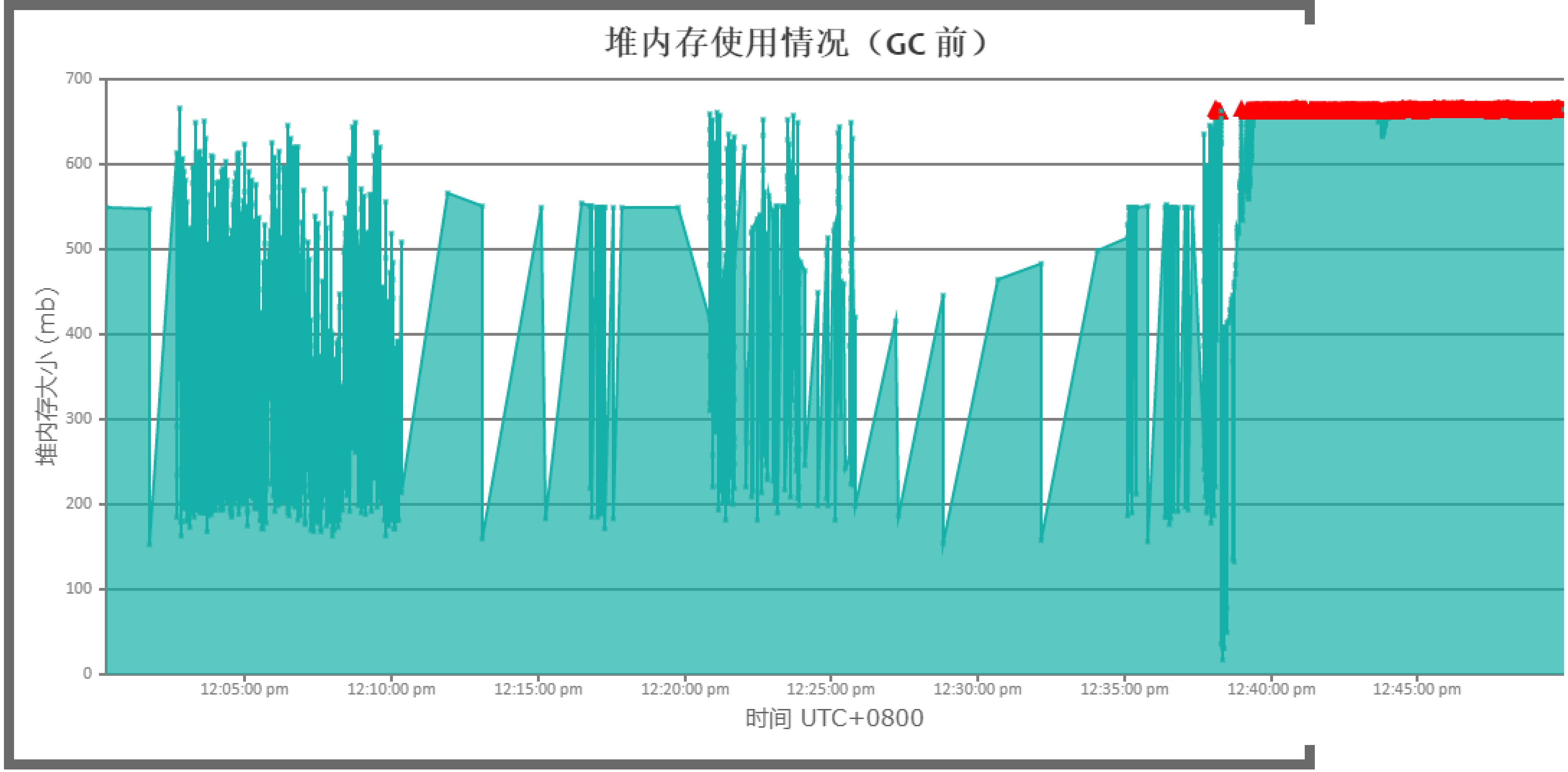


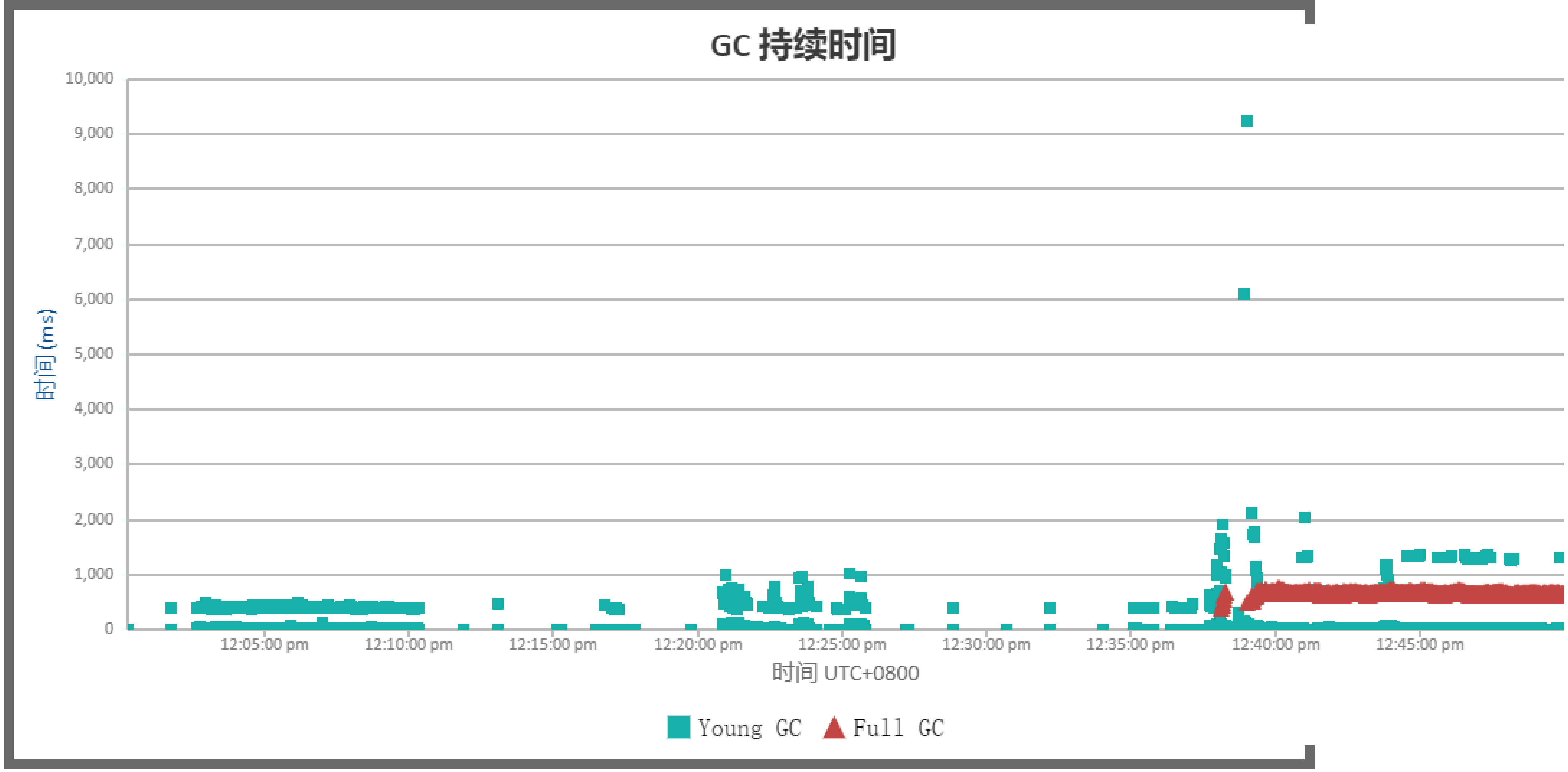
700 - 800	32	0.82%

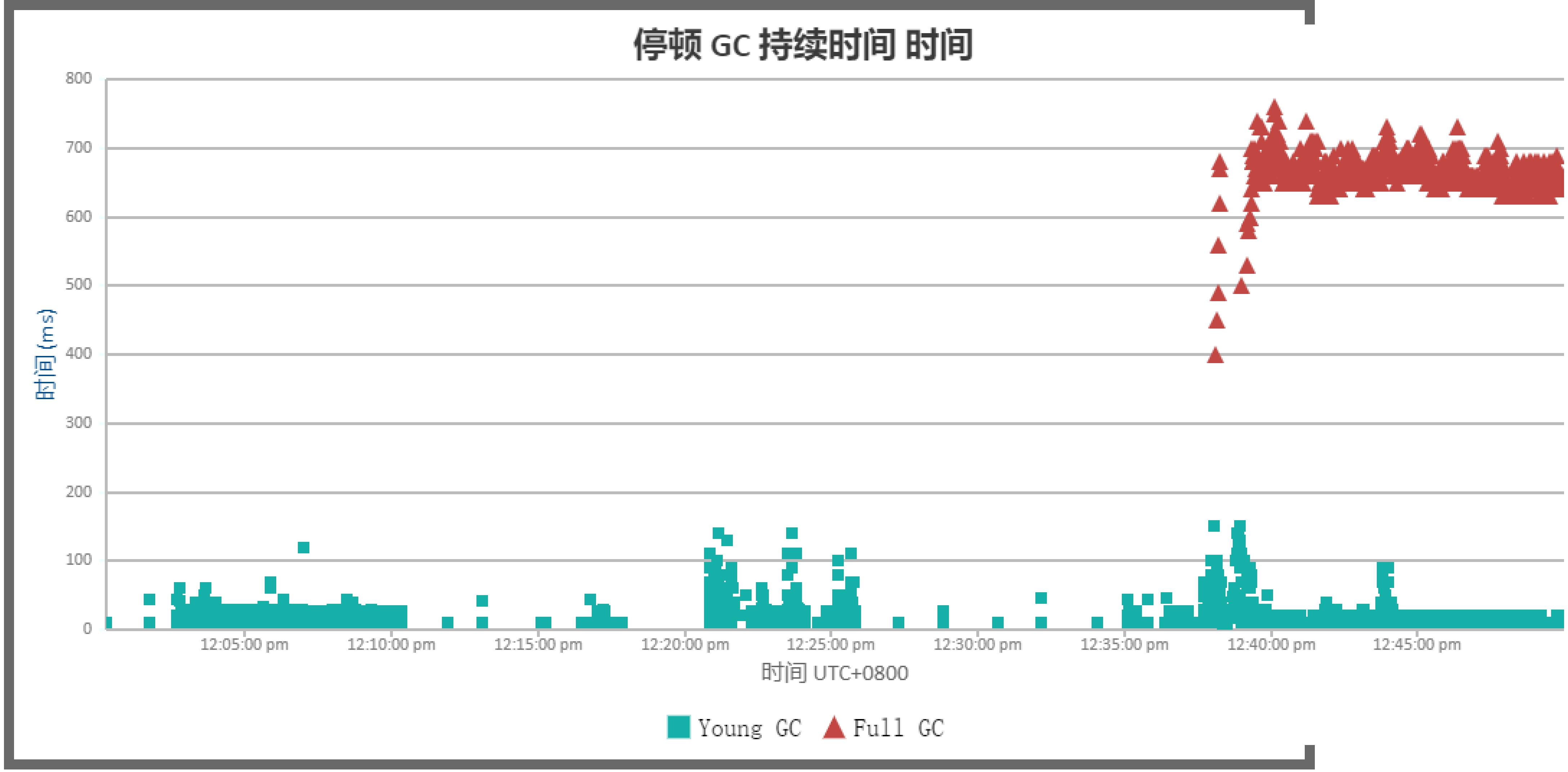
交互图表 (如何在图表中进行缩放？)

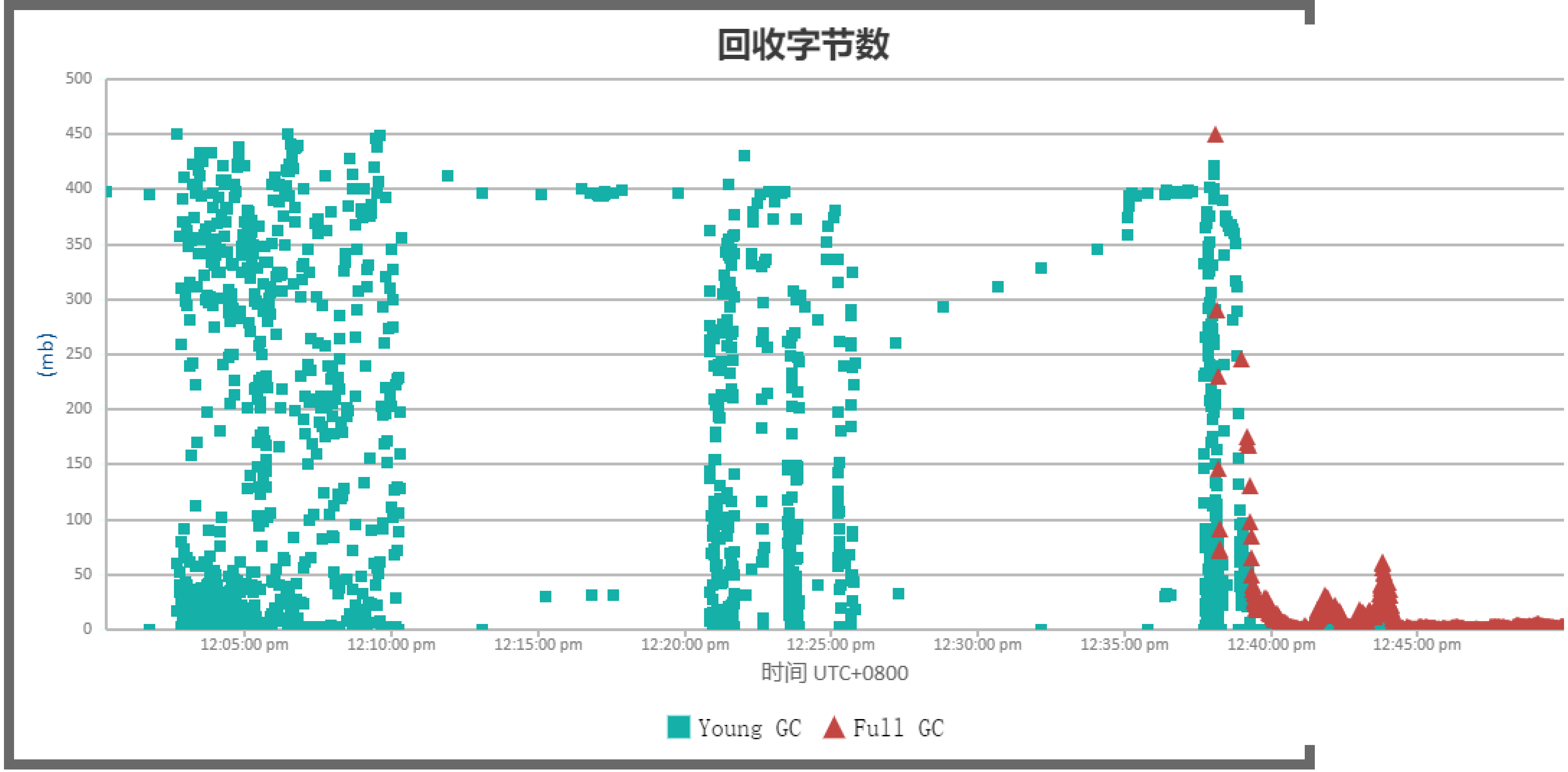
(https://www.youtube.com/watch?v=JhZFj6gJQyk)

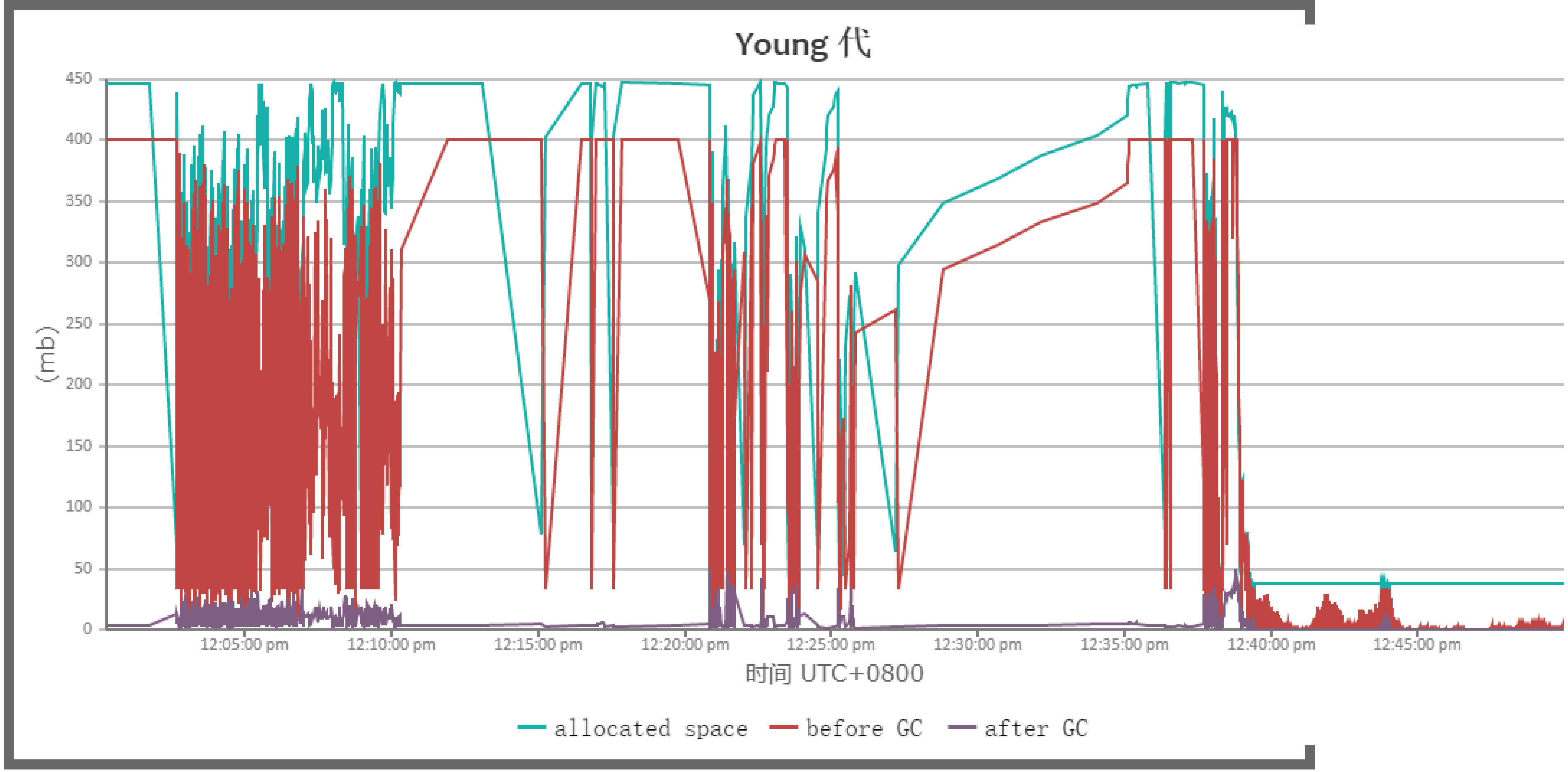


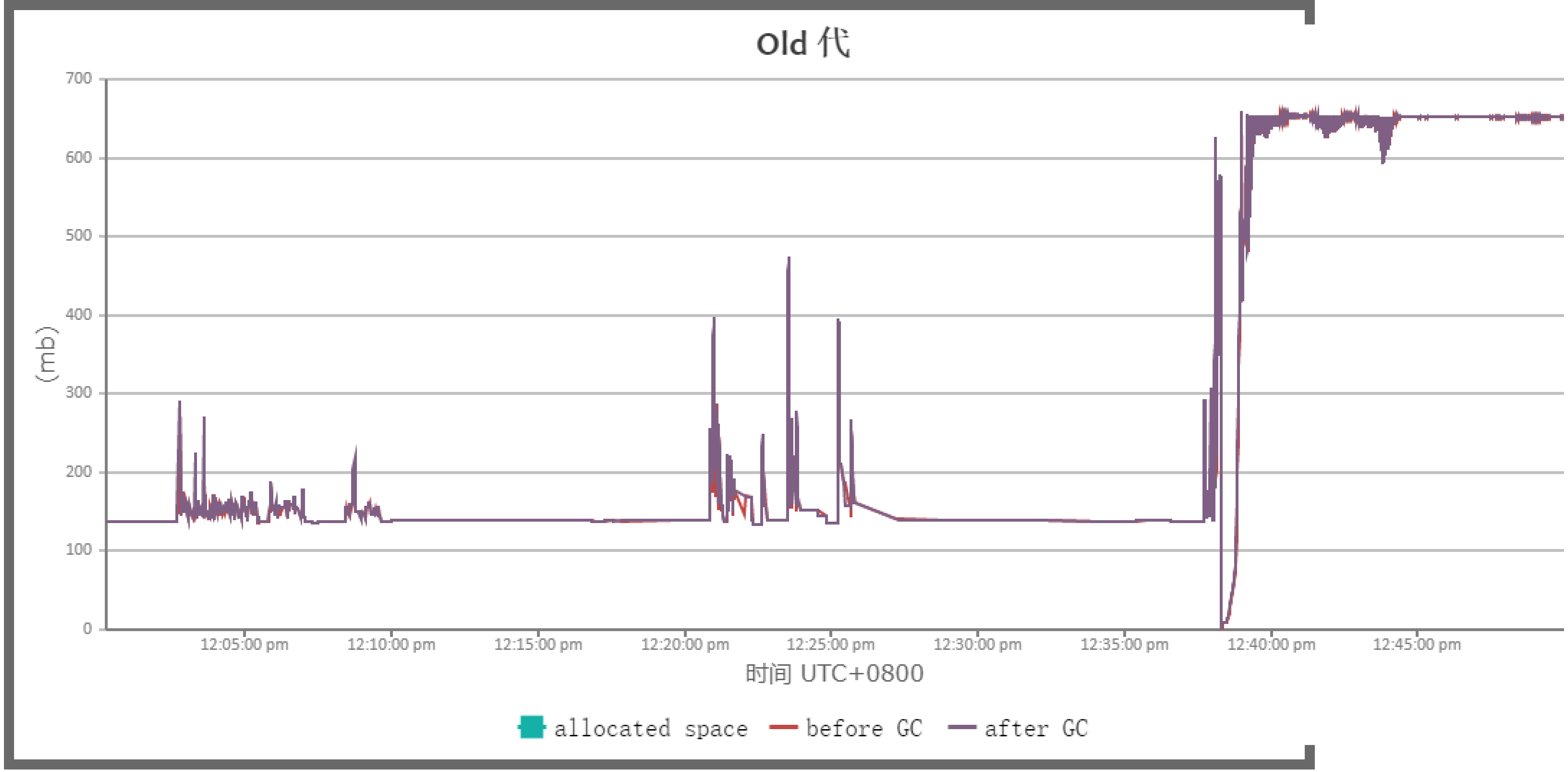

堆内存使用情况（GC 前）
堆内存大小（mb）
700
600
500
400
300
200
100
0
12:05:00 pm
12:10:00 pm
12:15:00 pm
12:20:00 pm
12:25:00 pm
12:30:00 pm
12:35:00 pm
12:40:00 pm
12:45:00 pm
时间 UTC+0800

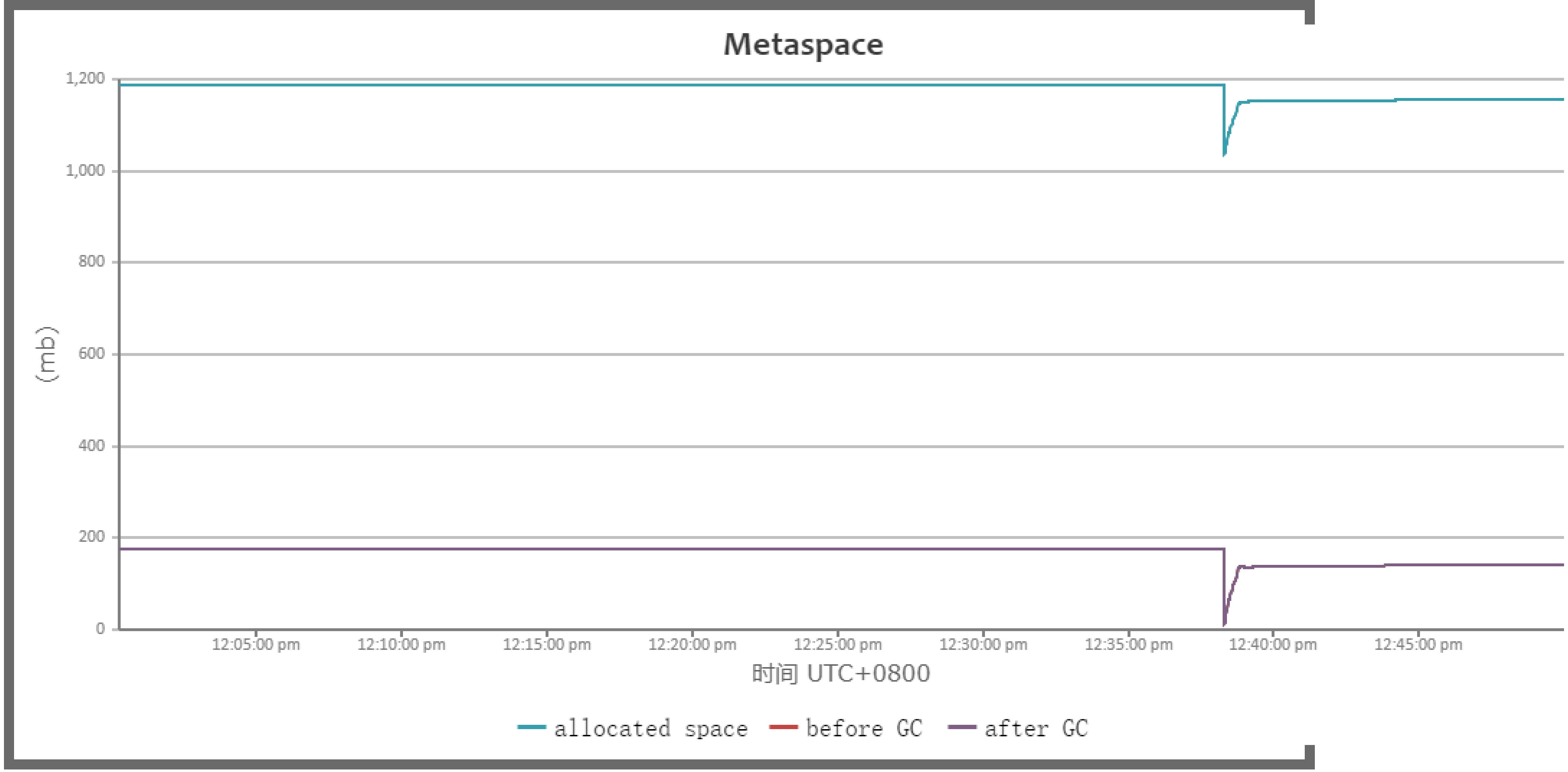

GC 持续时间
时间 (ms)
10,000
9,000
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
12:05:00 pm
12:10:00 pm
12:15:00 pm
12:20:00 pm
12:25:00 pm
12:30:00 pm
12:35:00 pm
12:40:00 pm
12:45:00 pm
时间 UTC+0800
Young GC
Full GC

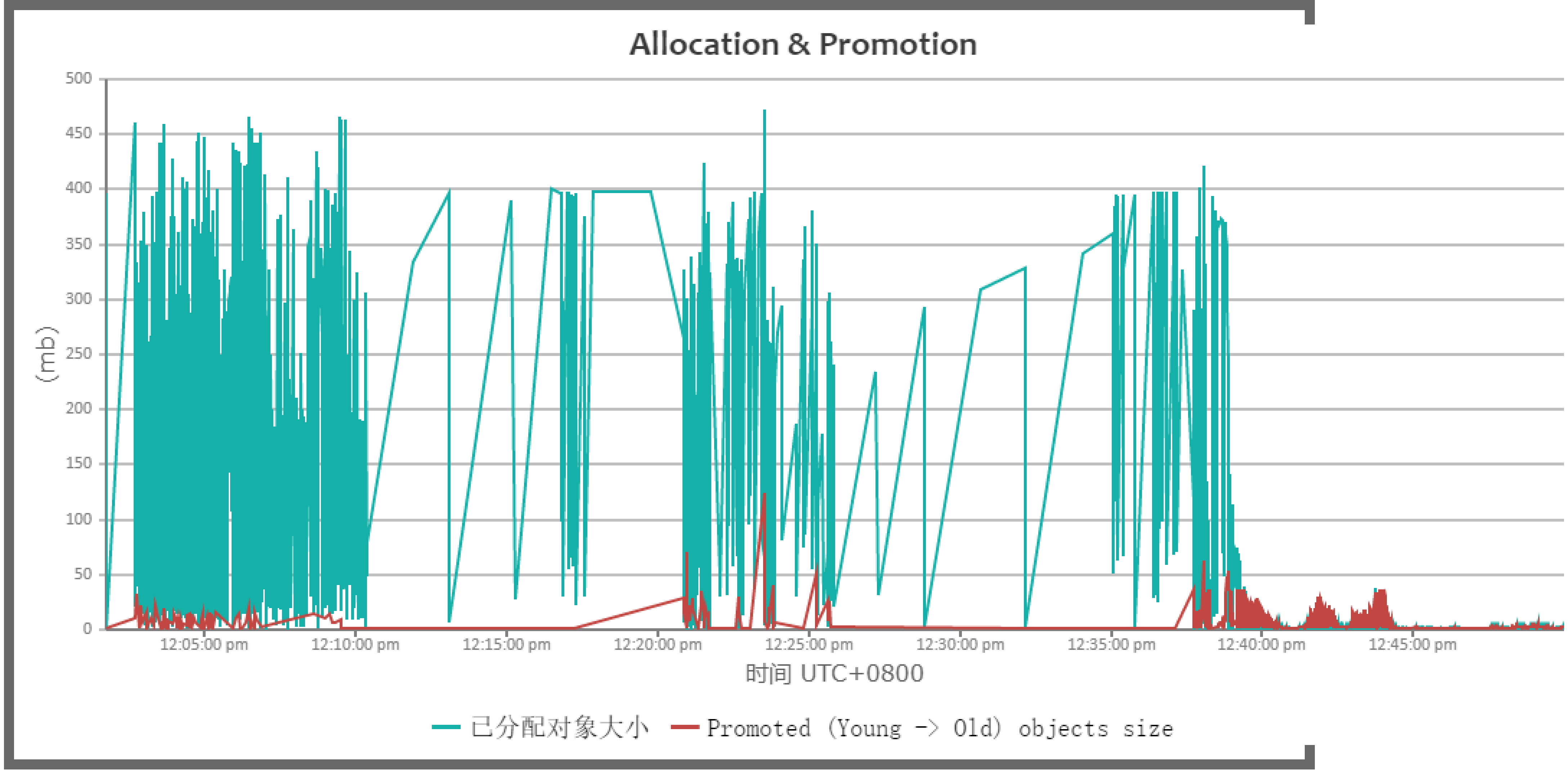

停顿 GC 持续时间 时间
时间 (ms)
800
700
600
500
400
300
200
100
0
12:05:00 pm
12:10:00 pm
12:15:00 pm
12:20:00 pm
12:25:00 pm
12:30:00 pm
12:35:00 pm
12:40:00 pm
12:45:00 pm
时间 UTC+0800
🟩 Young GC
▲ Full GC


回收字节数
500
450
400
350
300
250
200
150
100
50
0
(mb)
12:05:00 pm
12:10:00 pm
12:15:00 pm
12:20:00 pm
12:25:00 pm
12:30:00 pm
12:35:00 pm
12:40:00 pm
12:45:00 pm
时间 UTC+0800
Young GC
Full GC

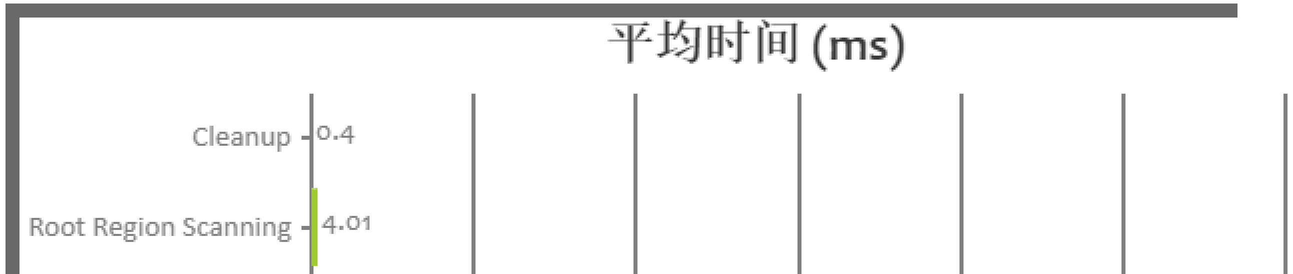

Young 代
450
400
350
300
250
200
150
100
50
0
(mb)
12:05:00 pm
12:10:00 pm
12:15:00 pm
12:20:00 pm
12:25:00 pm
12:30:00 pm
12:35:00 pm
12:40:00 pm
12:45:00 pm
时间 UTC+0800
allocated space
before GC
after GC


Old 代
700
600
500
400
300
200
100
0
(mb)
12:05:00 pm
12:10:00 pm
12:15:00 pm
12:20:00 pm
12:25:00 pm
12:30:00 pm
12:35:00 pm
12:40:00 pm
12:45:00 pm
时间 UTC+0800
allocated space
before GC
after GC

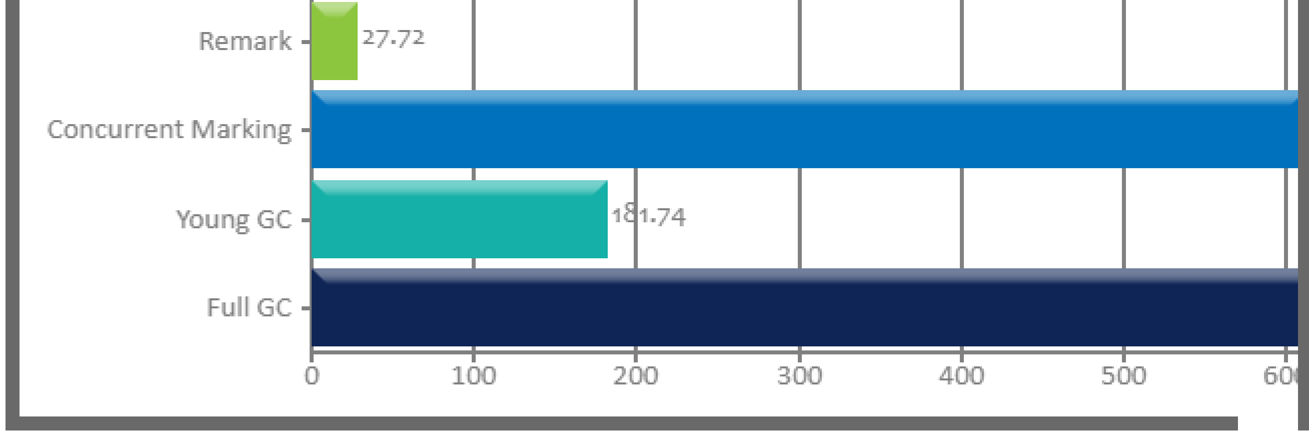

Metaspace
1,200
1,000
800
600
400
200
0
(mb)
12:05:00 pm
12:10:00 pm
12:15:00 pm
12:20:00 pm
12:25:00 pm
12:30:00 pm
12:35:00 pm
12:40:00 pm
12:45:00 pm
时间 UTC+0800
allocated space
before GC
after GC

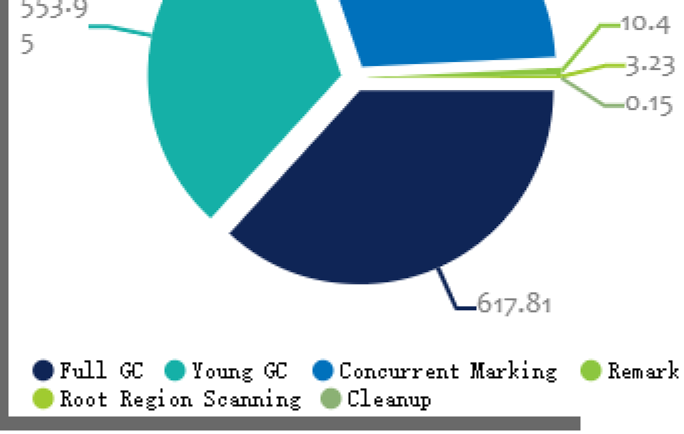

Allocation & Promotion
500
450
400
350
300
250
200
150
100
50
0
(mb)
12:05:00 pm
12:10:00 pm
12:15:00 pm
12:20:00 pm
12:25:00 pm
12:30:00 pm
12:35:00 pm
12:40:00 pm
12:45:00 pm
时间 UTC+0800
已分配对象大小
Promoted (Young -> Old) objects size

G1 回收阶段统计数据


平均时间 (ms)
Cleanup
0.4
Root Region Scanning
4.01

累计时间（秒）
493.86



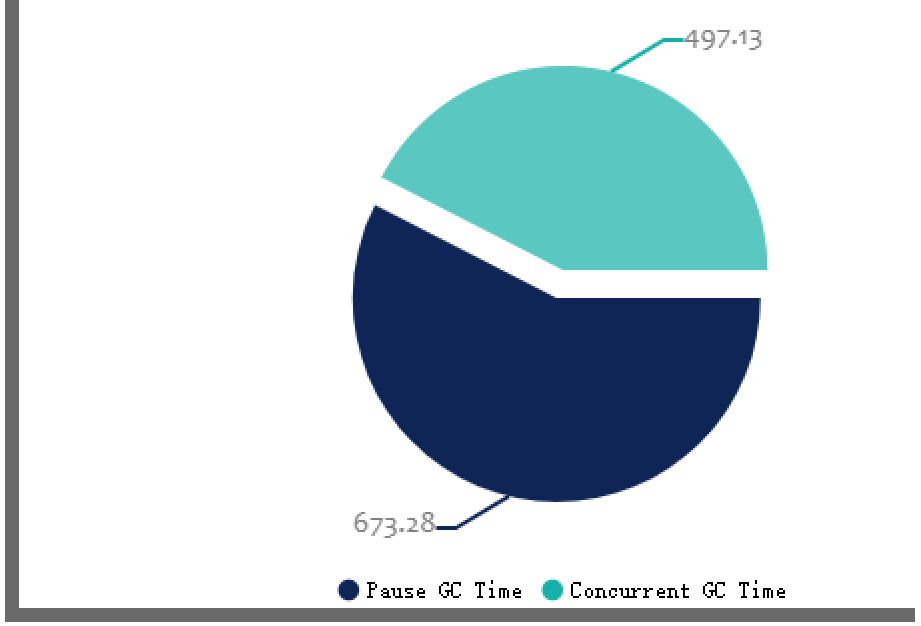


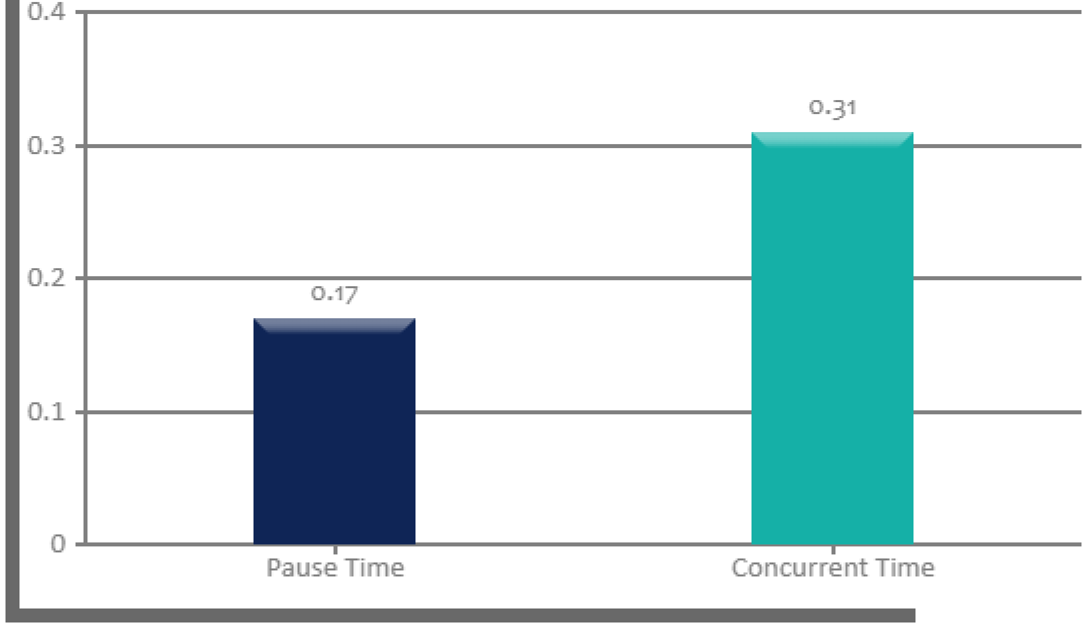
	Full GC ⏸	Young GC ⏸	Concurrent Marking	Remark ⏸	Root Region Scanning	Cleanup ⏸
Total Time ❓	10 min 17 sec 806 ms	9 min 13 sec 953 ms	8 min 13 sec 856 ms	10 sec 396 ms	3 sec 235 ms	149 ms
Avg Time ❓	662 ms	182 ms	612 ms	27.7 ms	4.01 ms	0.397 ms
Std Dev Time	25.2 ms	355 ms	448 ms	5.14 ms	9.51 ms	0.0685 ms
Min Time ❓	400 ms	0	0.187 ms	2.71 ms	0.00400 ms	0.0950 ms
Max Time ❓	760 ms	9 sec 228 ms	9 sec 138 ms	54.2 ms	163 ms	0.640 ms
Interval Time ❓	764 ms	978 ms	3 sec 591 ms	5 sec 933 ms	3 sec 591 ms	5 sec 933 ms
Count ❓	933	3048	807	375	807	375

G1 GC 时间

停顿，并发总时间（秒）

停顿，并发平均时间（秒）





停顿时间

Total Time	11 min 13 sec 279 ms
Avg Time	174 ms
Std Dev Time	276 ms
Min Time	0.0950 ms
Max Time	760 ms

并发时间

Total Time	8 min 17 sec 132 ms
Avg Time	308 ms
Std Dev Time	439 ms
Min Time	0.00400 ms
Max Time	9 sec 138 ms

对象数据

Total created bytes	188.61 gb

CPU Stats

(To learn more about CPU stats, 请点击此处 (https://blog.gceasy.io/2022/08/05/garbage-collection-cpu-statistics/))

CPU Time:	36 min 42 sec 670 ms

Total promoted bytes	11.18 gb
Avg creation rate	64.77 mb/sec
Avg promotion rate	3.84 mb/sec

User Time:	36 min 39 sec 60 ms
Sys Time:	3 sec 610 ms

连续 Full GC

None.

长时间停顿

None.

安全点持续时间

(如需了解更多有关安全点持续时间的信息, 请点击此处 (./gc-recommendations/safe-point-solution.jsp))

Not Reported in the log.

分配阻塞指标

(click here, 请点击此处 (./gc-recommendations/allocation-stall-solution.jsp))

Not Reported in the log.

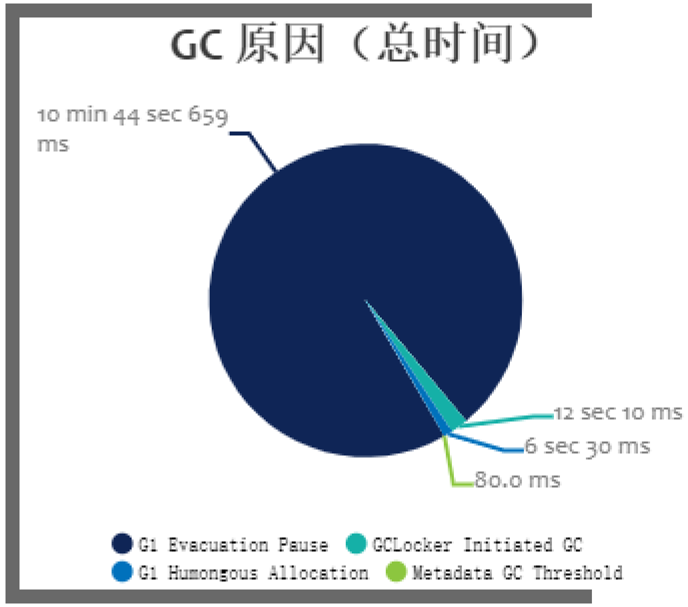
字符串重复指标

Not Reported in the log.

GC 原因

(哪些事件引发了 GC 以及这些事件消耗了多少时间？)

原因	计数	平均时间	最大时间	总时间
G1 Evacuation Pause	2781	232 ms	760 ms	10 min 44 sec 659 ms
GCLocker Initiated GC	44	273 ms	730 ms	12 sec 10 ms
G1 Humongous Allocation	346	17.4 ms	130 ms	6 sec 30 ms
Metadata GC Threshold	4	20.0 ms	30.0 ms	80.0 ms



Tenuring 摘要

未在日志中报告。

JVM 参数

(To learn about JVM Arguments, click here (https://blog.gceasy.io/2020/03/18/7-jvm-arguments-of-highly-effective-applications/))

未在日志中报告。