

一、SpringBoot应用JVM监控环境搭建

1、01-SpringBoot自带监控Actuator

SpringBoot自带监控功能Actuator，可以帮助实现对程序内部运行情况监控，比如监控内存状况、CPU、Bean加载情况、配置属性、日志信息、线程情况等。

使用步骤：

(1) 导入依赖坐标

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

(2) 访问监控地址：<http://127.0.0.1:9001/actuator>

监控应用endpoint：

路径	描述	默认开启
/beans	显示容器的全部的Bean，以及它们的关系	Y
/env	获取全部环境属性	Y
/env/{name}	根据名称获取特定的环境属性值	Y
/health	显示健康检查信息	Y
/info	显示设置好的应用信息	Y
/mappings	显示所有的@RequestMapping信息	Y
/metrics	显示应用的度量信息	Y
/scheduledtasks	显示任务调度信息	Y
/httptrace	显示Http Trace信息	Y
/caches	显示应用中的缓存	Y
/conditions	显示配置条件的匹配情况	Y
/configprops	显示@ConfigurationProperties的信息	Y
/loggers	显示并更新日志配置	Y
/shutdown	关闭应用程序	N
/threaddump	执行ThreadDump	Y
/headdump	返回HeadDump文件，格式为HPROF	Y
/prometheus	返回可供Prometheus抓取的信息	Y

配置：

```
# 暴露所有的监控点【含Prometheus】
management.endpoints.web.exposure.include: '*'
# 定义Actuator访问路径
management.endpoints.web.base-path: /actuator
# 开启endpoint 关闭服务功能
management.endpoint.shutdown.enabled: true
```

2、Micrometer

Spring Boot 2.0以上，使用了micrometer作为底层的度量工具，micrometer是监控度量的门面，它能支持按照各种格式来暴露数据，其中就有Prometheus。

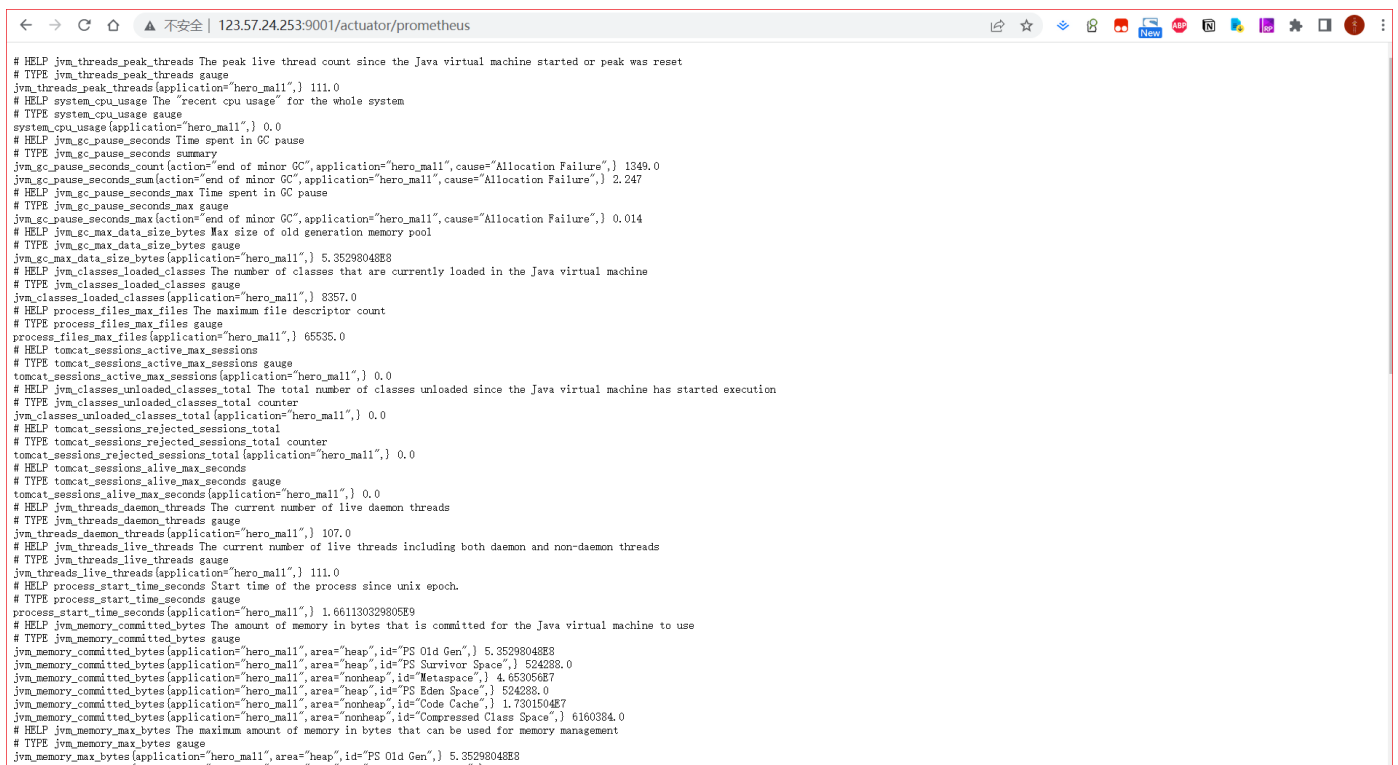
我们引入一个依赖来暴露Prometheus数据：

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

这个依赖的作用就是会开启一个endpoint，输出兼容Prometheus exporter的结果，方便Prometheus来采集。修改SpringBoot配置：

```
server.port: 9001
spring.application.name: hero_mall
# 暴露/actuator/prometheus，如果配置了*，则可以不配置这一行
management.endpoints.web.exposure.include: 'prometheus'
# 暴露的数据中添加application label
management.metrics.tags.application: ${spring.application.name}
```

然后启动应用，访问 `http://123.57.24.253:9001/actuator/prometheus` 应该会得到如下结果：



```
# HELP jvm_threads_peak_threads The peak live thread count since the Java virtual machine started or peak was reset
# TYPE jvm_threads_peak_threads gauge
jvm_threads_peak_threads(application="hero_mall",) 111.0
# HELP system_cpu_usage The "recent cpu usage" for the whole system
# TYPE system_cpu_usage gauge
system_cpu_usage(application="hero_mall",) 0.0
# HELP jvm_gc_pause_seconds Time spent in GC pause
# TYPE jvm_gc_pause_seconds summary
jvm_gc_pause_seconds_count(action="end of minor GC", application="hero_mall", cause="Allocation Failure",) 1349.0
jvm_gc_pause_seconds_sum(action="end of minor GC", application="hero_mall", cause="Allocation Failure",) 2.247
# HELP jvm_gc_pause_seconds_max Time spent in GC pause
# TYPE jvm_gc_pause_seconds_max gauge
jvm_gc_pause_seconds_max(action="end of minor GC", application="hero_mall", cause="Allocation Failure",) 0.014
# HELP jvm_gc_max_data_size_bytes Max size of old generation memory pool
# TYPE jvm_gc_max_data_size_bytes gauge
jvm_gc_max_data_size_bytes(application="hero_mall",) 5.35298048E8
# HELP jvm_classes_loaded_classes The number of classes that are currently loaded in the Java virtual machine
# TYPE jvm_classes_loaded_classes gauge
jvm_classes_loaded_classes(application="hero_mall",) 8387.0
# HELP process_files_max_files The maximum file descriptor count
# TYPE process_files_max_files gauge
process_files_max_files(application="hero_mall",) 65535.0
# HELP tomcat_sessions_active_max_sessions
# TYPE tomcat_sessions_active_max_sessions gauge
tomcat_sessions_active_max_sessions(application="hero_mall",) 0.0
# HELP jvm_classes_unloaded_classes_total The total number of classes unloaded since the Java virtual machine has started execution
# TYPE jvm_classes_unloaded_classes_total counter
jvm_classes_unloaded_classes_total(application="hero_mall",) 0.0
# HELP tomcat_sessions_rejected_sessions_total
# TYPE tomcat_sessions_rejected_sessions_total counter
tomcat_sessions_rejected_sessions_total(application="hero_mall",) 0.0
# HELP tomcat_sessions_alive_max_seconds
# TYPE tomcat_sessions_alive_max_seconds gauge
tomcat_sessions_alive_max_seconds(application="hero_mall",) 0.0
# HELP jvm_threads_daemon_threads The current number of live daemon threads
# TYPE jvm_threads_daemon_threads gauge
jvm_threads_daemon_threads(application="hero_mall",) 107.0
# HELP jvm_threads_live_threads The current number of live threads including both daemon and non-daemon threads
# TYPE jvm_threads_live_threads gauge
jvm_threads_live_threads(application="hero_mall",) 111.0
# HELP process_start_time_seconds Start time of the process since unix epoch.
# TYPE process_start_time_seconds gauge
process_start_time_seconds(application="hero_mall",) 1.66113032805E9
# HELP jvm_memory_committed_bytes The amount of memory in bytes that is committed for the Java virtual machine to use
# TYPE jvm_memory_committed_bytes gauge
jvm_memory_committed_bytes(application="hero_mall", area="heap", id="PS Old Gen",) 5.35298048E8
jvm_memory_committed_bytes(application="hero_mall", area="heap", id="PS Survivor Space",) 524288.0
jvm_memory_committed_bytes(application="hero_mall", area="nonheap", id="Metaspace",) 4.653056E7
jvm_memory_committed_bytes(application="hero_mall", area="heap", id="PS Eden Space",) 524288.0
jvm_memory_committed_bytes(application="hero_mall", area="nonheap", id="Code Cache",) 1.7301504E7
jvm_memory_committed_bytes(application="hero_mall", area="nonheap", id="Compressed Class Space",) 6160384.0
# HELP jvm_memory_max_bytes The maximum amount of memory in bytes that can be used for memory management
# TYPE jvm_memory_max_bytes gauge
jvm_memory_max_bytes(application="hero_mall", area="heap", id="PS Old Gen",) 5.35298048E8
jvm_memory_max_bytes(application="hero_mall", area="heap", id="PS Survivor Space",) 524288.0
jvm_memory_max_bytes(application="hero_mall", area="nonheap", id="Metaspace",) 4.653056E7
jvm_memory_max_bytes(application="hero_mall", area="heap", id="PS Eden Space",) 524288.0
jvm_memory_max_bytes(application="hero_mall", area="nonheap", id="Code Cache",) 1.7301504E7
jvm_memory_max_bytes(application="hero_mall", area="nonheap", id="Compressed Class Space",) 6160384.0
```

这就是Prometheus exporter的格式，可以看到里面暴露了很详细的JVM指标。接下来，配置Prometheus抓取监控数据。

3、Prometheus整合Grafana

Prometheus需要增加对 `http://123.57.24.253:9001/actuator/prometheus` 采集，我们修改prometheus.yml配置：

```
- job_name: "hero_mall"
  metrics_path: "/actuator/prometheus"
  static_configs:
    - targets: [ "172.17.187.79:9001" ]
```

启动Prometheus，没报错的话应该就已经在正常采集了。我们访问prometheus的web ui看一下数: `http://182.92.87.65:9090/graph`

PrometheusAlertsGraphStatus ▾Help

Enable query history

Expression (press Shift+Enter for newlines)

Execute

jvm_threads_live_threads ▾

Graph

Console

⏪

Moment

⏩

Element

jvm_threads_live_threads{application="test-gc",instance="172.16.0.103:9001",job="hero_mall"}

Add Graph

看到这样的结果说明数据采集正常。接下来配置Grafana的JVM监控Dashboard。

这里采用的展示模板是JVM监控大盘， dashboard-ID：12856。使用模板ID即可导入模板, 完整效果如下图：

The screenshot displays the Grafana JVM Monitoring Dashboard for the application 'test-gc' on instance '172.16.0.103:9001'. The dashboard is organized into several sections:

- Summary (Overview):** Shows process start time (17.9 min), start time (2022-09-29 21:39:53), memory usage (2.17%), and non-heap memory usage (4.60%).
- Service Golden Indicators:** Includes QPS (1 minute average), error rate (1 minute average), request latency (1 minute average), and saturation.
- JVM Memory:** Displays heap memory, non-heap memory, total memory, and JVM process memory usage.
- JVM Load:** Shows CPU usage, load, thread count, and the number of threads in various states (blocked, new, runnable).

Each section contains a line graph showing the metric's value over time, with a vertical red line indicating the current time. The dashboard also includes a 'Refresh dashboard' button and a 'Run SLS' button.

二、高延迟接口压测

1、代码

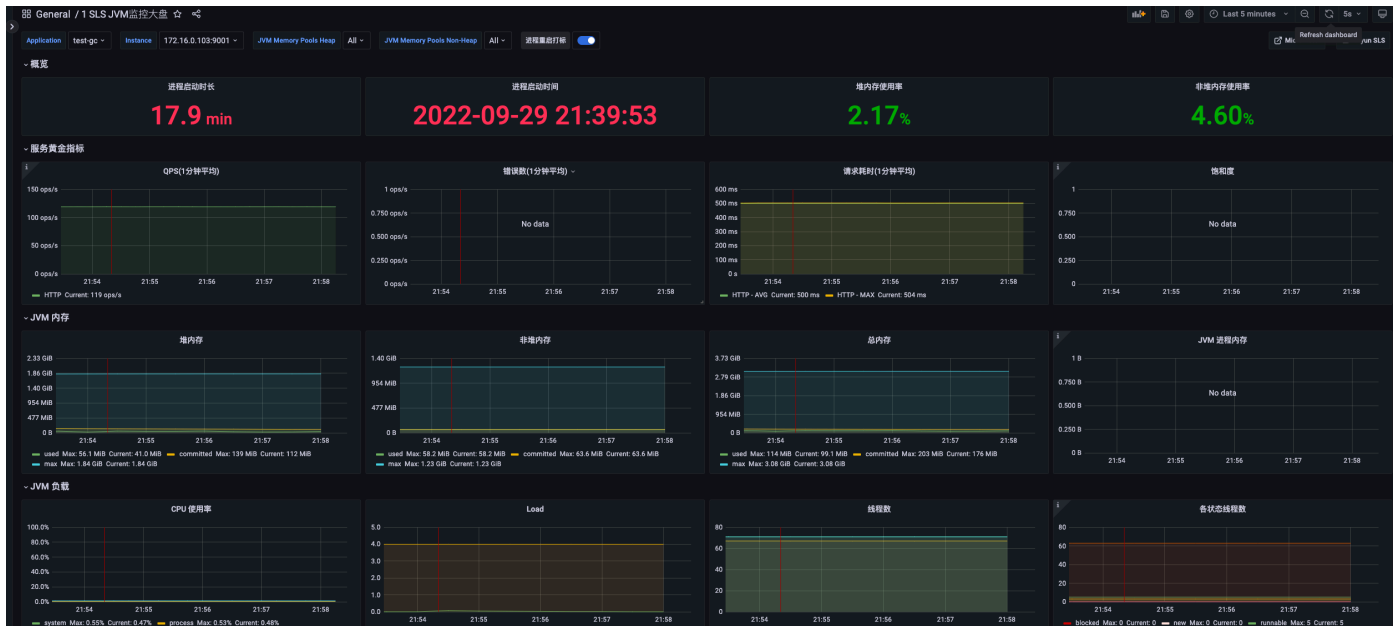
```
@GetMapping("/fast")
public String fastTest(){
    List<User> list = new ArrayList<>();
    for (int i=0; i<100; i++){
        User u = new User();
        list.add(u);
    }
    return "fast ok";
}
```

2、压测脚本

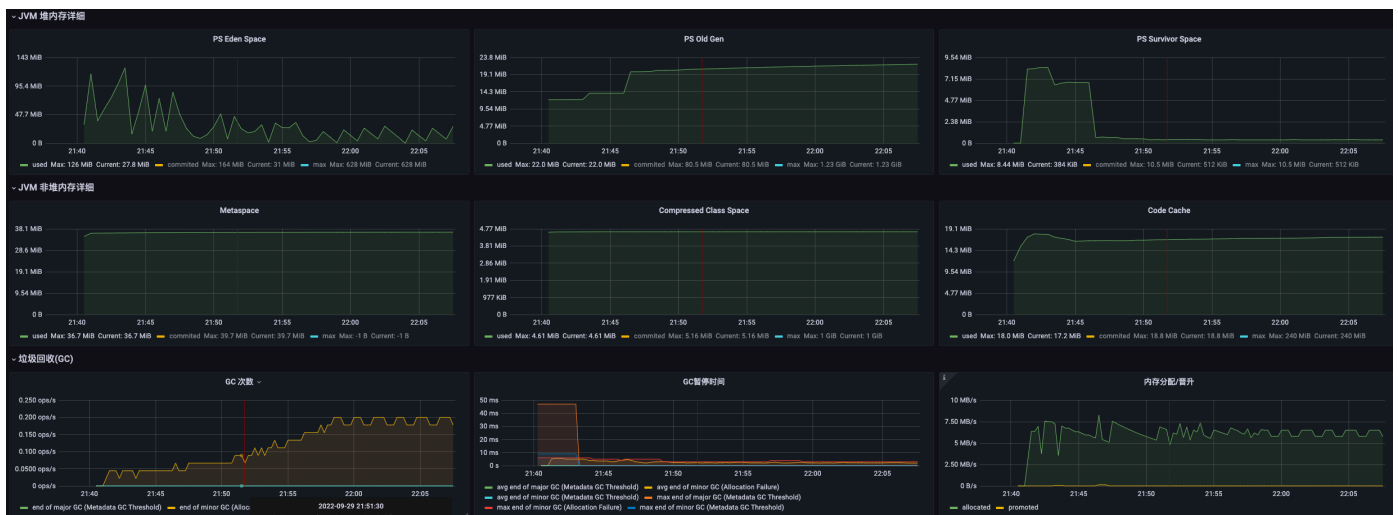
60个线程，循环5000次

3、使用监控平台查看指标

从下图可以看到，吞吐量在120QPS、时延在500ms；最大堆内存为1.86G，最大使用堆内存为50M左右；最大非堆内存为1.23G，最大使用为58.4M左右



分代内存和GC情况如下图所示



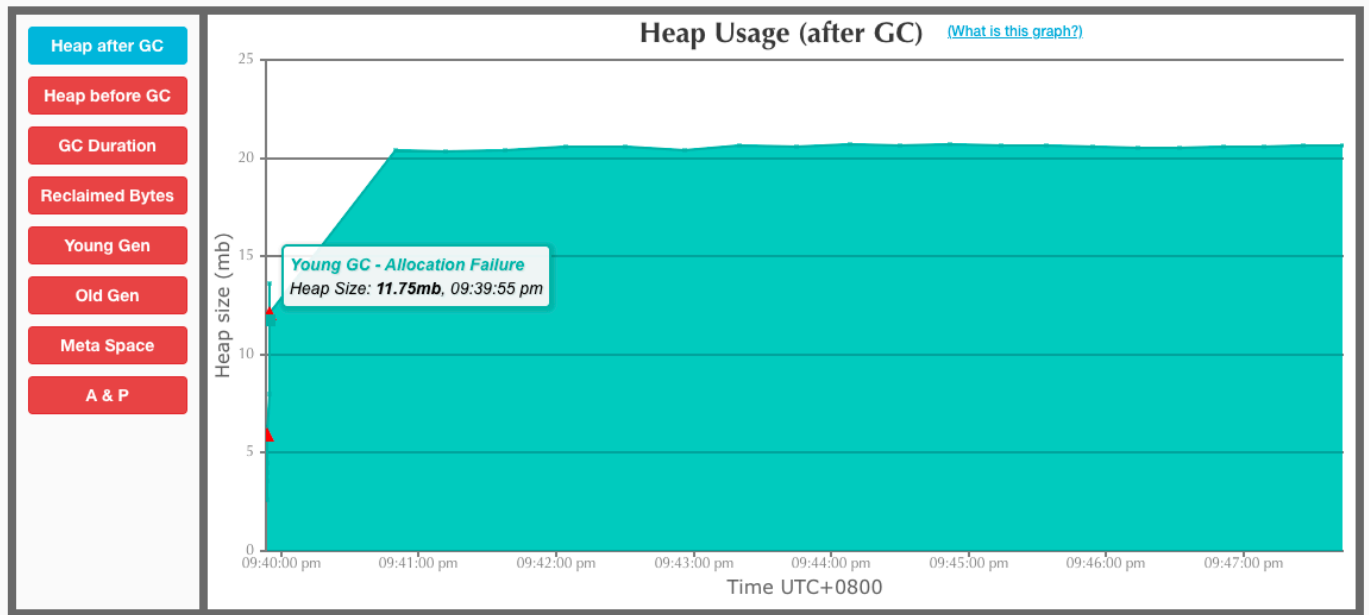
4、使用gcEasy查看

可以看到年轻代和老年代比较正常，而元空间非常不正常，使用的最大内存是32M，然而分配的确有2.87G
吞吐量为99.962%



另外还可以看到连个FullGC，都是因为元空间分配失败导致

Interactive Graphs [\(How to zoom graphs?\)](#)



总停顿时间

Total GC stats

Total GC count	30
Total reclaimed bytes	2.88 gb
Total GC time	180 ms
Avg GC time	6.00 ms
GC avg time std dev	9.17 ms
GC min/max time	0 / 40.0 ms
GC Interval avg time	16 sec 178 ms

Minor GC stats

Minor GC count	28
Minor GC reclaimed	2.88 gb
Minor GC total time	110 ms
Minor GC avg time	3.93 ms
Minor GC avg time std dev	4.88 ms
Minor GC min/max time	0 / 10.0 ms
Minor GC Interval avg	17 sec 377 ms

Full GC stats

Full GC Count	2
Full GC reclaimed	1.62 mb
Full GC total time	70.0 ms
Full GC avg time	35.0 ms
Full GC avg time std dev	5.00 ms
Full GC min/max time	30.0 ms / 40.0 ms
Full GC Interval avg	1 sec 253 ms

GC Pause Statistics

Pause Count	30
Pause total time	180 ms
Pause avg time	6.00 ms
Pause avg time std dev	0.0
Pause min/max time	0 / 40.0 ms

Object Stats

Total created bytes	2.9 gb
Total promoted bytes	9.66 mb
Avg creation rate	6.33 mb/sec
Avg promotion rate	21 kb/sec

CPU Stats

(To learn more about CPU stats, [click here](#))

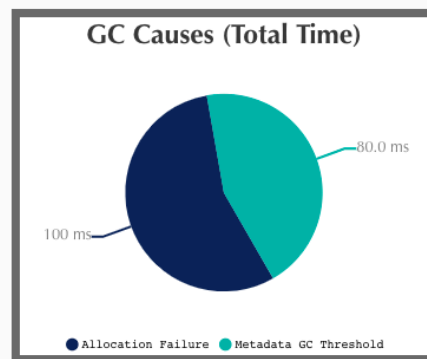
CPU Time:	560 ms
User Time:	510 ms
Sys Time:	50.0 ms

GC原因：主要是因为年轻代空间不足和元空间分配超过阈值

? GC Causes ?

(What events caused GCs & how much time they consumed?)

Cause	Count	Avg Time	Max Time	Total Time
Allocation Failure ?	26	3.85 ms	10.0 ms	100 ms
Metadata GC Threshold ?	4	20.0 ms	40.0 ms	80.0 ms



三、JVM调优

针对上述情况，对JVM进行调优

堆空间：建议设置成GC后老年代的3到4倍：20M*4=80M

元空间：参数-XX:MetaspaceSize=N，设置元空间大小为128MB

新生代：参数-Xmn，建议扩大至1-1.5倍FullGC之后的老年代空间占用。20M*(1-1.5)=(20

-30)M，设置新生代大小为30MB

设置参数

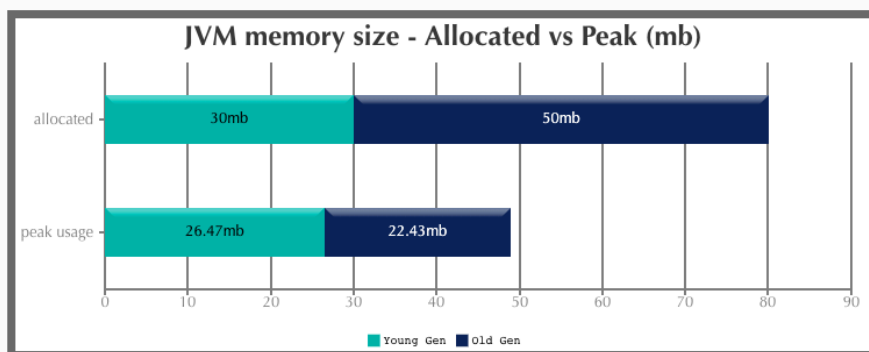
```
JAVA_OPT="{JAVA_OPT} -Xms80m -Xmx80m -Xmn30m -XX:MetaspaceSize=128m"
```

调整之后的内存使用情况

JVM memory size

(To learn about JVM Memory, [click here](#))

Generation	Allocated ?	Peak ?
Young Generation	30 mb	26.47 mb
Old Generation	50 mb	22.43 mb
Total	80 mb	48.71 mb



Key Performance Indicators

(Important section of the report. To learn more about KPIs, [click here](#))

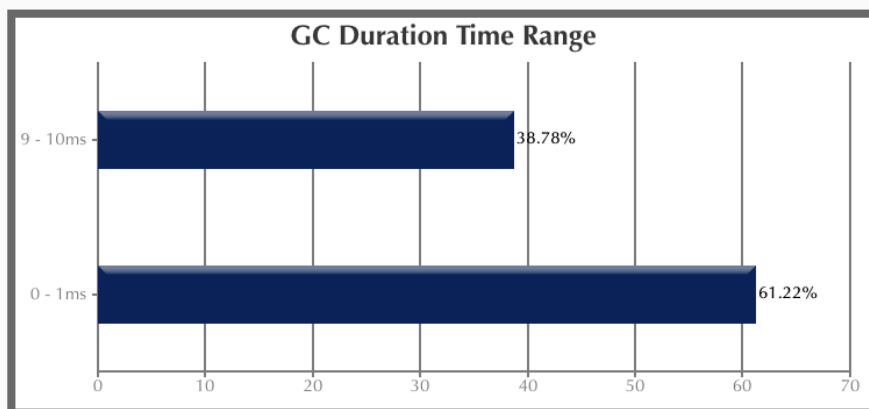
1 Throughput : 99.825%

2 Latency:

Avg Pause GC Time ?	3.88 ms
Max Pause GC Time ?	10.0 ms

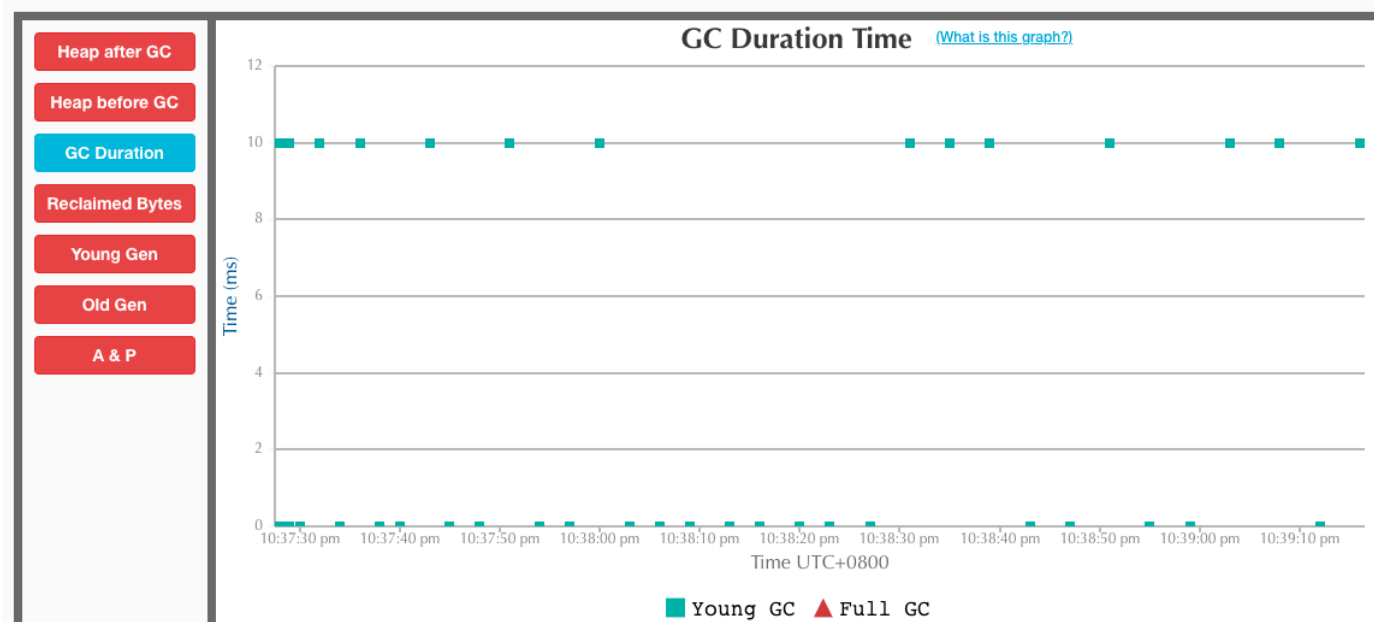
GCPauseDuration Time Range ?:

Duration (ms)	No. of GCs	Percentage
1 ms		
0 - 1	30	61.22%
9 - 10	19	38.78%



fullGC的问题已经消除

Interactive Graphs [\(How to zoom graphs?\)](#)



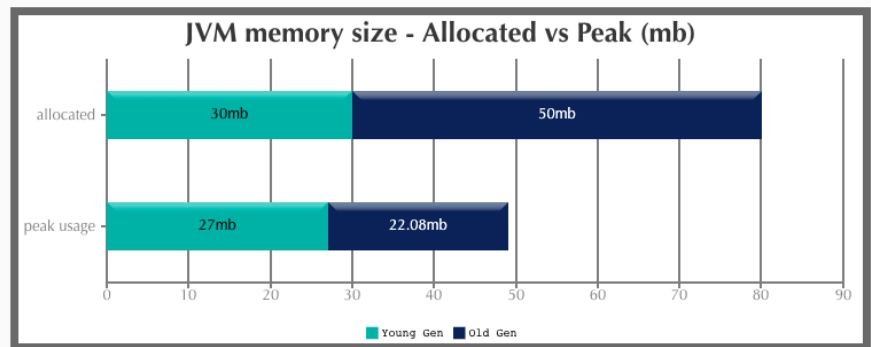
调优后的GC统计：没有了fullgc，最大的平均停顿时间和最大停顿时间明显下降



JVM memory size

(To learn about JVM Memory, [click here](#))

Generation	Allocated ?	Peak ?
Young Generation	30 mb	27 mb
Old Generation	50 mb	22.08 mb
Total	80 mb	48.33 mb



Key Performance Indicators

未选择任何文件

(Important section of the report. To learn more about KPIs, [click here](#))

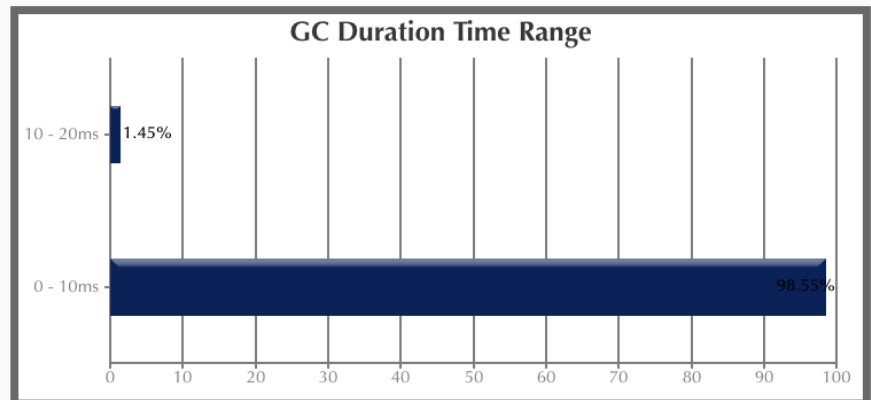
1 Throughput ? : 99.892%

2 Latency:

Avg Pause GC Time ?	3.48 ms
Max Pause GC Time ?	20.0 ms

GC Pause Duration Time Range ?:

Duration (ms)	No. of GCs	Percentage
10 ms Change		
0 - 10	68	98.55%
10 - 20	1	1.45%



CMS垃圾回收器GC情况

Total GC stats

Total GC count ?	69
Total reclaimed bytes ?	n/a
Total GC time ?	240 ms
Avg GC time ?	3.48 ms
GC avg time std dev	5.06 ms
GC min/max time	0 / 20.0 ms
GC Interval avg time ?	3 sec 267 ms

Minor GC stats

Minor GC count	69
Minor GC reclaimed ?	1.6 gb
Minor GC total time	240 ms
Minor GC avg time ?	3.48 ms
Minor GC avg time std dev	5.06 ms
Minor GC min/max time	0 / 20.0 ms
Minor GC Interval avg ?	3 sec 267 ms

Full GC stats

Full GC Count	0
Full GC reclaimed ?	n/a
Full GC total time	n/a
Full GC avg time ?	n/a
Full GC avg time std dev	n/a
Full GC min/max time	n/a
Full GC Interval avg ?	n/a

GC Pause Statistics

Pause Count	69
Pause total time	240 ms
Pause avg time ?	3.48 ms
Pause avg time std dev	0.0
Pause min/max time	0 / 20.0 ms

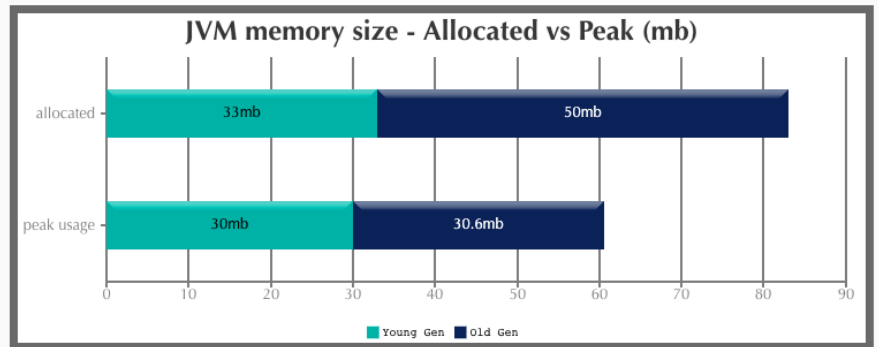
2、使用G1

```
JAVA_OPT="$ {JAVA_OPT} -XX:+UseG1GC -XX:MaxGCPauseMillis=100 "
```

JVM memory size

(To learn about JVM Memory, [click here](#))

Generation	Allocated	Peak
Young Generation	33 mb	30 mb
Old Generation	50 mb	30.6 mb
Total	80 mb	60.6 mb



Key Performance Indicators

(Important section of the report. To learn more about KPIs, [click here](#))

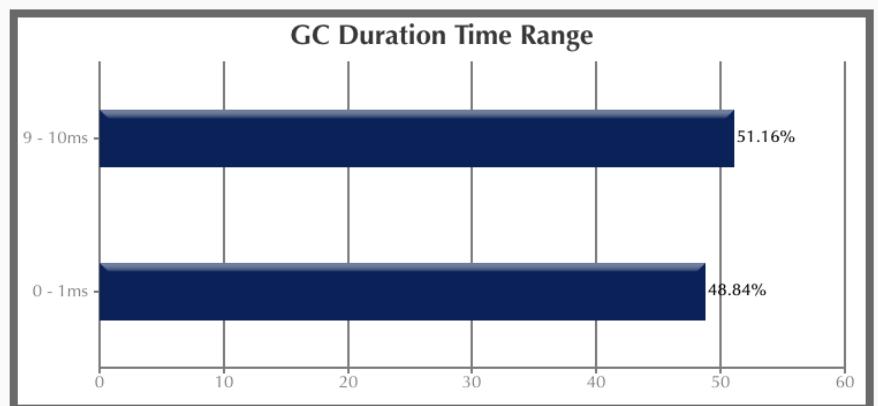
1 Throughput : 99.84%

2 Latency:

Avg Pause GC Time	5.12 ms
Max Pause GC Time	10.0 ms

GCPauseDuration Time Range :

Duration (ms)	No. of GCs	Percentage
1 ms Change		
0 - 1	21	48.84%
9 - 10	22	51.16%



G1垃圾回收器的GC情况

Total GC stats

Total GC count	43
Total reclaimed bytes	n/a
Total GC time	220 ms
Avg GC time	5.12 ms
GC avg time std dev	5.00 ms
GC min/max time	0 / 10.0 ms
GC Interval avg time	3 sec 271 ms

Other GC stats

Other GC count	43
Other GC reclaimed	1.14 gb
Other GC total time	220 ms
Other GC avg time	5.12 ms
Other GC avg time std dev	5.00 ms
Other GC min/max time	0 / 10.0 ms
Other GC Interval avg	3 sec 271 ms

Full GC stats

Full GC Count	0
Full GC reclaimed	n/a
Full GC total time	n/a
Full GC avg time	n/a
Full GC avg time std dev	n/a
Full GC min/max time	n/a
Full GC Interval avg	n/a

GC Pause Statistics

Pause Count	43
Pause total time	220 ms
Pause avg time	5.12 ms
Pause avg time std dev	0.0
Pause min/max time	0 / 10.0 ms