22 如何优化垃圾回收机制?

你好，我是刘超。

我们知道，在 Java 开发中，开发人员是无需过度关注对象的回收与释放的，JVM 的垃圾回收机制可以减轻不少工作量。但完全交由 JVM 回收对象，也会增加回收性能的不确定性。在一些特殊的业务场景下，不合适的垃圾回收算法以及策略，都有可能导致系统性能下降。

面对不同的业务场景，垃圾回收的调优策略也不一样。例如，在对内存要求苛刻的情况下，需要提高对象的回收效率；在 CPU 使用率高的情况下，需要降低高并发时垃圾回收的频率。可以说，垃圾回收的调优是一项必备技能。

这讲我们就把这项技能的学习进行拆分，看看回收（后面简称 GC）的算法有哪些，体现 GC 算法好坏的指标有哪些，又如何根据自己的业务场景对 GC 策略进行调优?

垃圾回收机制

掌握 GC 算法之前，我们需要先弄清楚 3 个问题。第一，回收发生在哪里？第二，对象在什么时候可以被回收？第三，如何回收这些对象?

1. 回收发生在哪里?

JVM 的内存区域中，程序计数器、虚拟机栈和本地方法栈这 3 个区域是线程私有的，随着线程的创建而创建，销毁而销毁；栈中的栈帧随着方法的进入和退出进行入栈和出栈操作，每个栈帧中分配多少内存基本是在类结构确定下来的时候就已知的，因此这三个区域的内存分配和回收都具有确定性。

那么垃圾回收的重点就是关注堆和方法区中的内存了，堆中的回收主要是对象的回收，方法区的回收主要是废弃常量和无用的类的回收。

2. 对象在什么时候可以被回收?

那 JVM 又是怎样判断一个对象是可以被回收的呢？一般一个对象不再被引用，就代表该对象可以被回收。目前有以下两种算法可以判断该对象是否可以被回收。

引用计数算法：这种算法是通过一个对象的引用计数器来判断该对象是否被引用了。每当对象被引用，引用计数器就会加 1；每当引用失效，计数器就会减 1。当对象的引用计数器的值为 0 时，就说明该对象不再被引用，可以被回收了。这里强调一点，虽然引用计数算法的实现简单，判断效率也很高，但它存在着对象之间相互循环引用的问题。

可达性分析算法：GC Roots 是该算法的基础，GC Roots 是所有对象的根对象，在 JVM 加载时，会创建一些普通对象引用正常对象。这些对象作为正常对象的起始点，在垃圾回收时，会从这些 GC Roots 开始向下搜索，当一个对象到 GC Roots 没有任何引用链相连时，就证明此对象是不可用的。目前 HotSpot 虚拟机采用的就是这种算法。

以上两种算法都是通过引用来判断对象是否可以被回收。在 JDK 1.2 之后，Java 对引用的概念进行了扩充，将引用分为了以下四种：

引用类型	功能特点
强引用（Strong Reference）	被强引用关联的对象永远不会被垃圾收集器回收掉
软引用（Soft Reference）	软引用关联的对象，只有当系统将要发生内存溢出时，才会去回收软引用引用的对象
弱引用（Weak Reference）	只被弱引用关联的对象，只要发生垃圾收集事件，就会被回收
虚引用（Phantom Reference）	被虚引用关联的对象的唯一作用是能在这个对象被回收器回收时收到一个系统通知

3. 如何回收这些对象？

了解完 Java 程序中对象的回收条件，那么垃圾回收线程又是如何回收这些对象的呢？JVM 垃圾回收遵循以下两个特性。

自动性：Java 提供了一个系统级的线程来跟踪每一块分配出去的内存空间，当 JVM 处于空闲循环时，垃圾收集器线程会自动检查每一块分配出去的内存空间，然后自动回收每一块空闲的内存块。

不可预期性：一旦一个对象没有被引用了，该对象是否立刻被回收呢？答案是不可预期的。我们很难确定一个没有被引用的对象是不是会被立刻回收掉，因为有可能当程序结束后，这个对象仍在内存中。

垃圾回收线程在 JVM 中是自动执行的，Java 程序无法强制执行。我们唯一能做的就是通过调用 System.gc 方法来"建议"执行垃圾收集器，但是否可执行，什么时候执行？仍然不可预期。

GC 算法

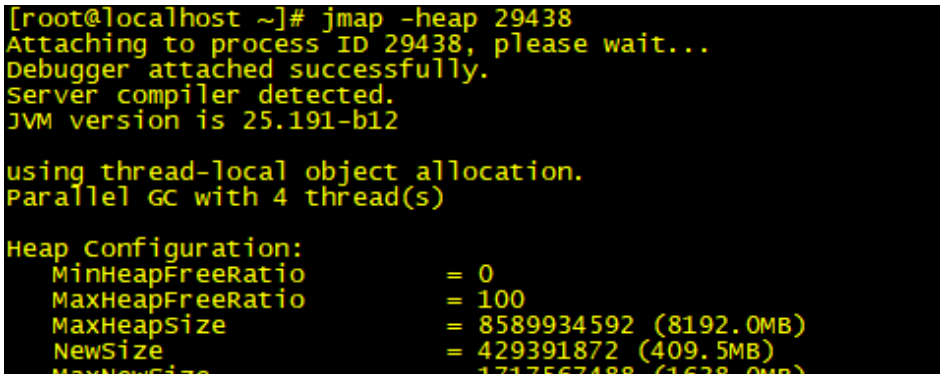
JVM 提供了不同的回收算法来实现这一套回收机制，通常垃圾收集器的回收算法可以分为以下几种：

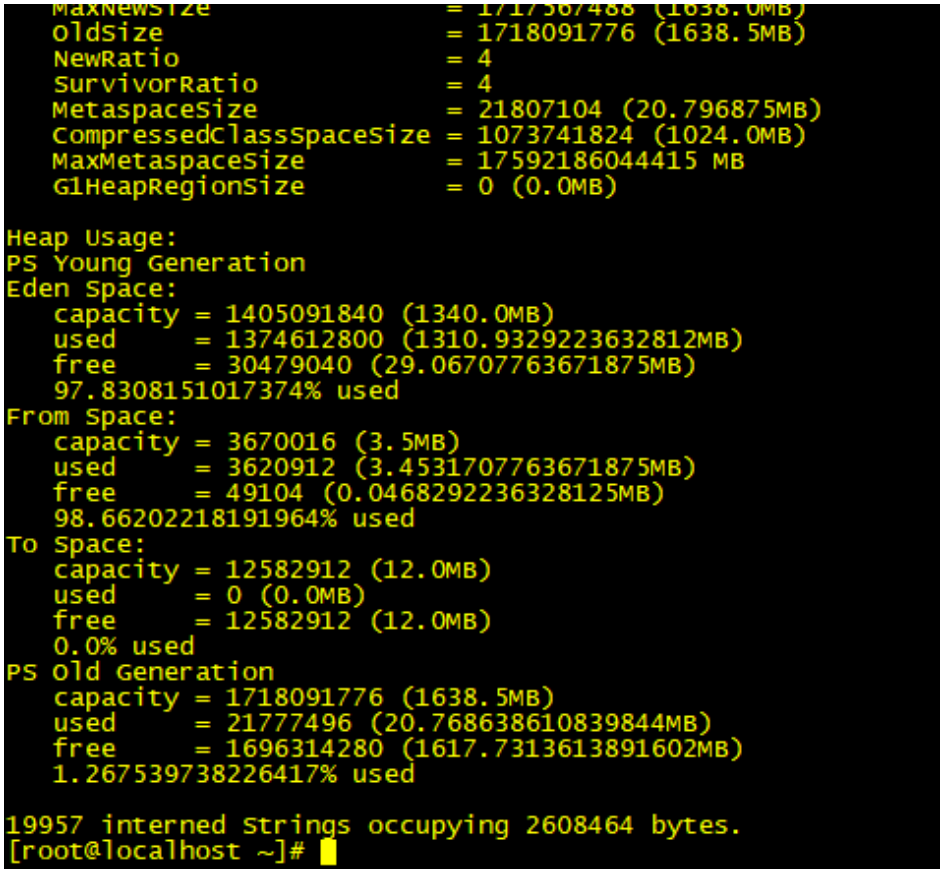
回收算法类型	优点	缺点
标记-清除算法（Mark-Sweep）	不需要移动对象，简单高效	标记-清除过程效率低，GC产生内存碎片
复制算法（Copying）	简单高效，不会产生内存碎片	内存使用率低，且有可能产生频繁复制问题
标记-整理算法（Mark-Compact）	综合了前两种算法的优点	仍需要移动局部对象
分代收集算法（Generational Collection）	分区回收	对于长时间存活对象的场景回收效果不明显，甚至起到反作用

如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现，JDK1.7 update14 之后 Hotspot 虚拟机所有的回收器整理如下（以下为服务端垃圾收集器）：

回收器类型	回收算法	特点	设置参数
Serial New / Serial Old回收器	复制算法/标记-清除算法	单线程复制回收，简单高效，但会暂停程序导致停顿	-XX:+UseSerialGC（年轻代、老年代回收器为：Serial New、Serial Old）
ParNew New / ParNew Old回收器	复制算法/标记-整理算法	多线程复制回收，降低了停顿时间，但容易增加上下文切换	-XX:+UseParNewGC（年轻代、老年代回收器为：ParNew New、Serial Old，JDK1.8中无效） -XX:+UseParallelOldGC（年轻代、老年代回收器为：Parallel Scavenge、Parallel Old）
Parallel Scavenge回收器	复制算法	并行回收器，追求高吞吐量，高效利用CPU	-XX:+UseParallelGC（年轻代、老年代回收器为：Parallel Scavenge、Serial Old） -XX:ParallelGCThreads=4（设置并发线程）
CMS回收器	标记-清理算法	老年代回收器，高并发、低停顿，追求最短GC回收停顿时间，CPU占用比较高，响应时间快，停顿时间短	-XX:+UseConcMarkSweepGC（年轻代、老年代回收器为：ParNew New、CMS（Serial Old作为备用））
G1回收器	标记-整理+复制算法	高并发、低停顿，可预测停顿时间	-XX:+UseG1GC（年轻代、老年代回收器为：G1、G1） -XX:MaxGCPauseMillis=200（设置最大暂停时间）

其实在 JVM 规范中并没有明确 GC 的运作方式，各个厂商可以采用不同的方式实现垃圾收集器。我们可以通过 JVM 工具查询当前 JVM 使用的垃圾收集器类型，首先通过 ps 命令查询出经常 ID，再通过 jmap -heap ID 查询出 JVM 的配置信息，其中就包括垃圾收集器的设置类型。





GC 性能衡量指标

一个垃圾收集器在不同场景下表现出的性能也不一样，那么如何评价一个垃圾收集器的性能好坏呢？我们可以借助一些指标。

吞吐量：这里的吞吐量是指应用程序所花费的时间和系统总运行时间的比值。我们可以按照这个公式来计算 GC 的吞吐量：系统总运行时间 = 应用程序耗时 +GC 耗时。如果系统运行了 100 分钟，GC 耗时 1 分钟，则系统吞吐量为 99%。GC 的吞吐量一般不能低于 95%。

停顿时间：指垃圾收集器正在运行时，应用程序的暂停时间。对于串行回收器而言，停顿时间可能会比较长；而使用并发回收器，由于垃圾收集器和应用程序交替运行，程序的停顿时间就会变短，但其效率很可能不如独占垃圾收集器，系统的吞吐量也很可能会降低。

垃圾回收频率：多久发生一次指垃圾回收呢？通常垃圾回收的频率越低越好，增大堆内存空间可以有效降低垃圾回收发生的频率，但同时也意味着堆积的回收对象越多，最终也会增加回收时的停顿时间。所以我们只要适当地增大堆内存空间，保证正常的垃圾回收频率即可。

查看 & 分析 GC 日志

已知了性能衡量指标，现在我们需要通过工具查询 GC 相关日志，统计各项指标的信息。首先，我们需要通过 JVM 参数预先设置 GC 日志，通常有以下几种 JVM 参数设置：

```
-XX:+PrintGC 输出 GC 日志
-XX:+PrintGCDetails 输出 GC 的详细日志
-XX:+PrintGCTimeStamps 输出 GC 的时间戳（以基准时间的形式）
-XX:+PrintGCDateStamps 输出 GC 的时间戳（以日期的形式，如 2013-05-04T21:53:59.234+0800）
-XX:+PrintHeapAtGC 在进行 GC 的前后打印出堆的信息
-Xloggc:../logs/gc.log 日志文件的输出路径
```

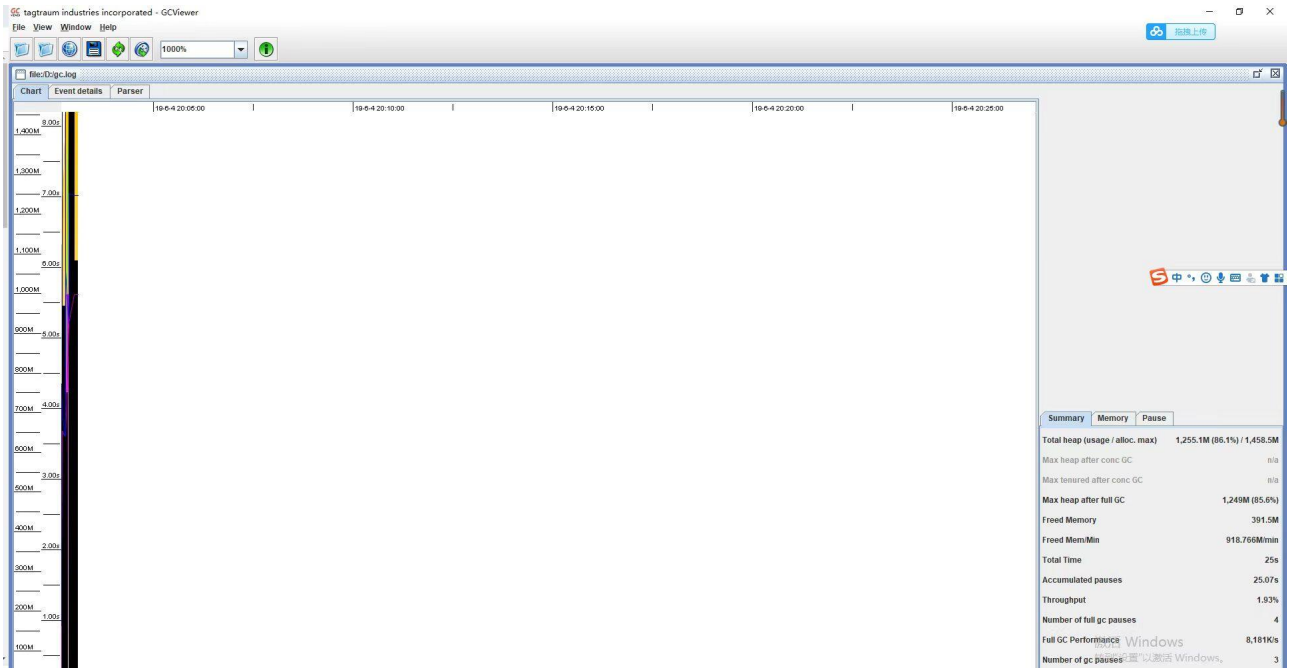
这里使用如下参数来打印日志：

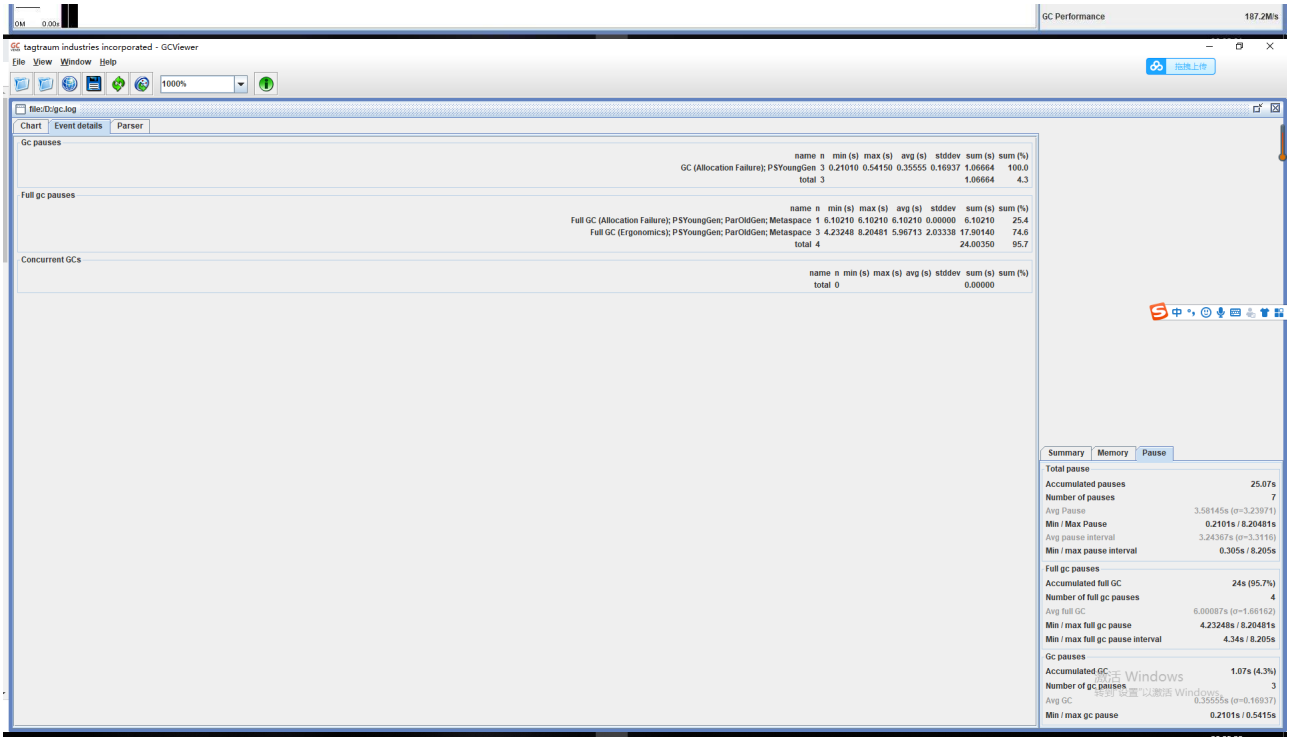
```
-XX:+PrintGCDateStamps -XX:+PrintGCDetails -Xloggc:./gclogs
```

打印后的日志为：

```
Java HotSpot(TM) 64-Bit Server VM (25.181-b13) for windows-amd64 JRE (1.8.0_181-b13), built on Jul  7 2018 04:01:33 by "java_re" with MS VC++ 10.0 (VS2010)
Memory: 4k page, physical 16696608k(4055756k free), swap 54445344k(33996432k free)
CommandLine flags: -XX:InitialHeapSize=1048576000 -XX:MaxHeapSize=1572864000 -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPoint
0.640: [GC (Allocation Failure) [PSYoungGen: 234769K->42492K(298496K)] 234769K->170029K(981504K), 0.2100955 secs] [Times: user=0.67 sys=0.13, real=0.21 secs]
0.945: [GC (Allocation Failure) [PSYoungGen: 298492K->42472K(469504K)] 426029K->366523K(1152512K), 0.3150446 secs] [Times: user=0.97 sys=0.16, real=0.32 secs]
1.432: [GC (Allocation Failure) [PSYoungGen: 469480K->42472K(469504K)] 793531K->713297K(1152512K), 0.5414956 secs] [Times: user=1.25 sys=0.27, real=0.54 secs]
1.973: [Full GC (Ergonomics) [PSYoungGen: 42472K->0K(469504K)] [ParOldGen: 670825K->657410K(1024000K)] 713297K->657410K(1493504K), [Metaspace: 4887K->4887K(10567
7.557: [Full GC (Ergonomics) [PSYoungGen: 427008K->0K(469504K)] [ParOldGen: 657410K->950235K(1024000K)] 1084418K->950235K(1493504K), [Metaspace: 5305K->5305K(105
11.897: [Full GC (Ergonomics) [PSYoungGen: 334950K->254985K(469504K)] [ParOldGen: 950235K->1023975K(1024000K)] 1285186K->1278961K(1493504K), [Metaspace: 5305K->5
20.102: [Full GC (Allocation Failure) [PSYoungGen: 254985K->254985K(469504K)] [ParOldGen: 1023975K->1023898K(1024000K)] 1278961K->1278884K(1493504K), [Metaspace:
Heap
 PSYoungGen      total 469504K, used 274663K [0x00000000e0c00000, 0x0000000100000000, 0x0000000100000000)
  eden space 427008K, 64% used [0x00000000e0c00000,0x00000000f1839d38,0x00000000fad00000)
  from space 42496K, 0% used [0x00000000fad00000,0x00000000fad00000,0x00000000fd680000)
  to   space 42496K, 0% used [0x00000000fd680000,0x00000000fd680000,0x0000000100000000)
 ParOldGen       total 1024000K, used 1023898K [0x00000000a2400000, 0x00000000e0c00000, 0x00000000e0c00000)
  object space 1024000K, 99% used [0x00000000a2400000,0x00000000e0be6910,0x00000000e0c00000)
 Metaspace       used 5338K, capacity 5420K, committed 5504K, reserved 1056768K
  class space    used 565K, capacity 592K, committed 640K, reserved 1048576K
```

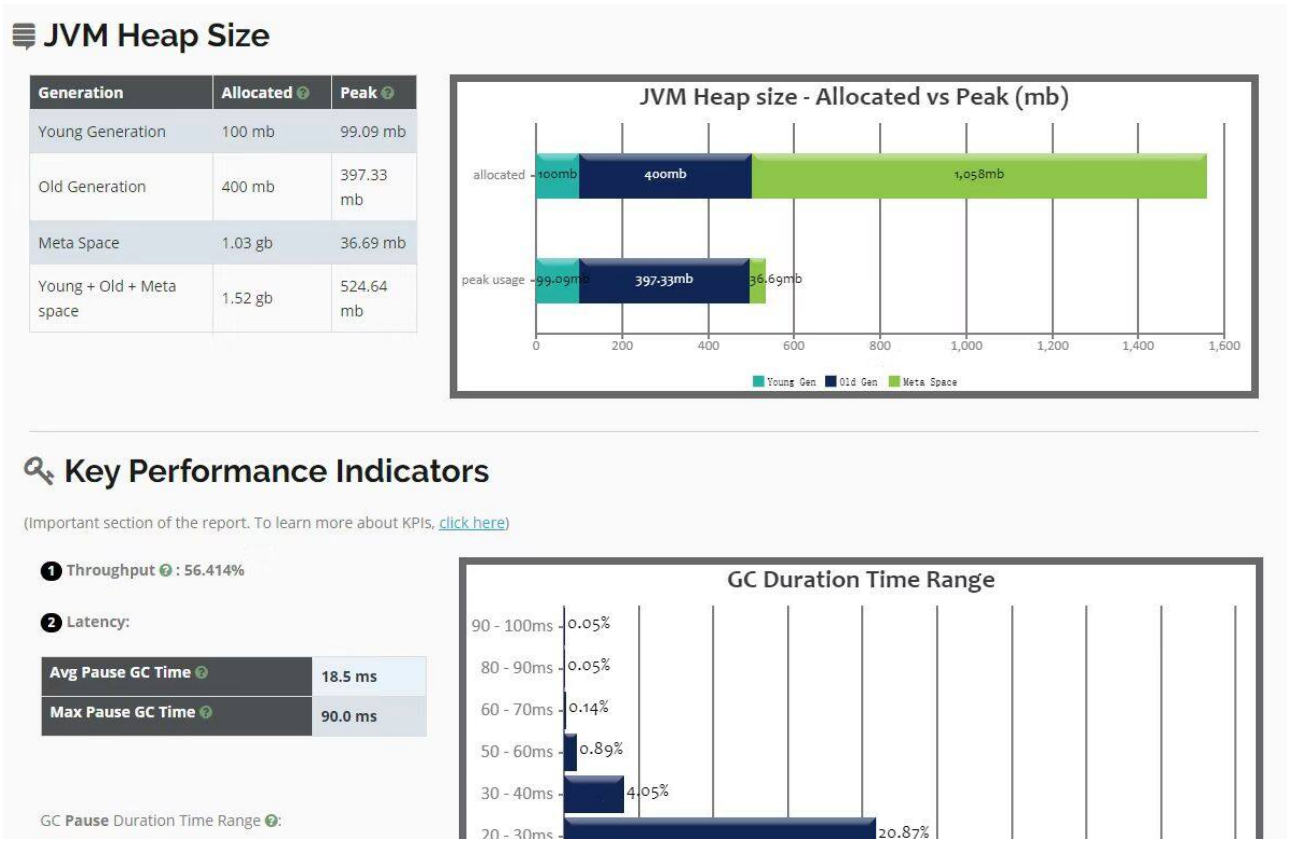
上图是运行很短时间的 GC 日志，如果是长时间的 GC 日志，我们很难通过文本形式去查看整体的 GC 性能。此时，我们可以通过GCView工具打开日志文件，图形化界面查看整体的 GC 性能，如下图所示：



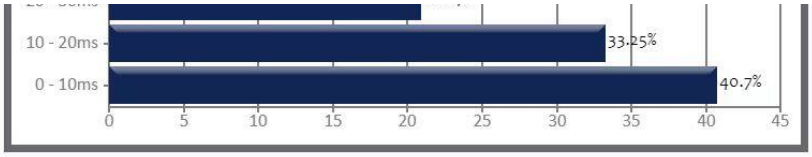


通过工具，我们可以看到吞吐量、停顿时间以及 GC 的频率，从而可以非常直观地了解到 GC 的性能情况。

这里我再推荐一个比较好用的 GC 日志分析工具，GCeasy是一款非常直观的 GC 日志分析工具，我们可以将日志文件压缩之后，上传到 GCeasy 官网即可看到非常清楚的 GC 日志分析结果：

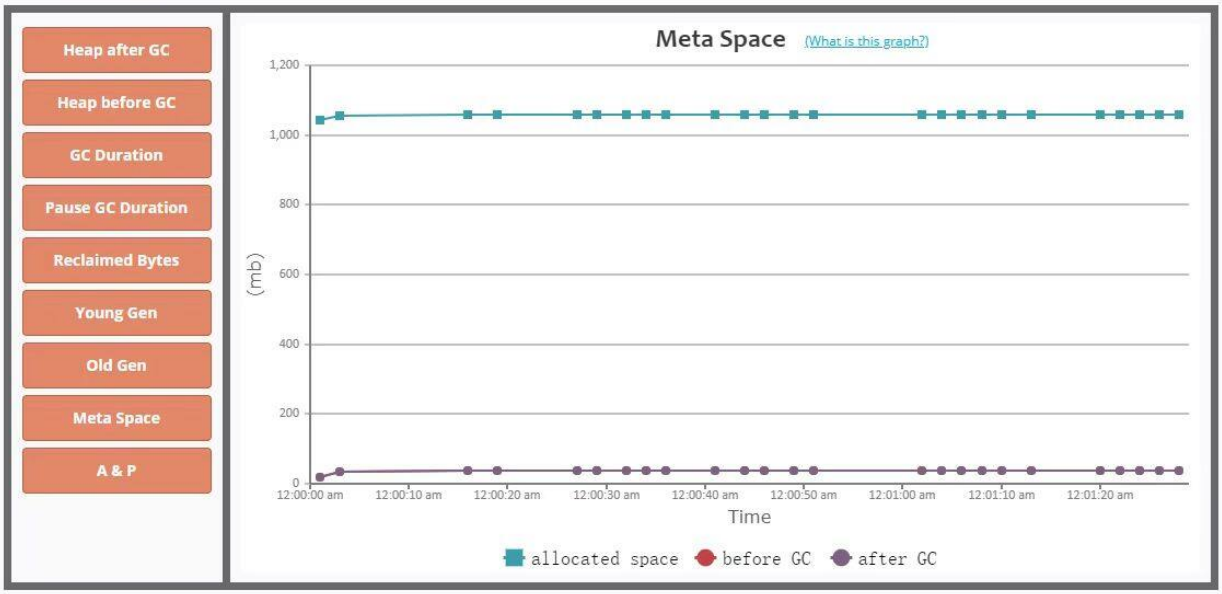


Duration (ms) 10 ms Change	No. of GCs	Percentage
0 - 10	864	40.7%
10 - 20	706	33.25%
20 - 30	443	20.87%
30 - 40	86	4.05%
50 - 60	19	0.89%
60 - 70	3	0.14%
80 - 90	1	0.05%
90 - 100	1	0.05%

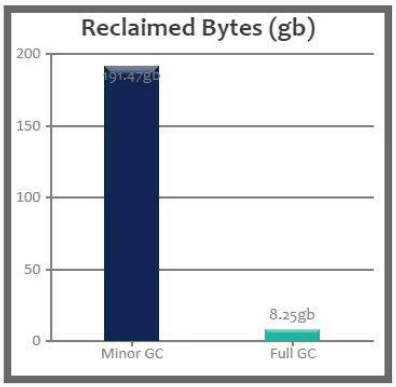


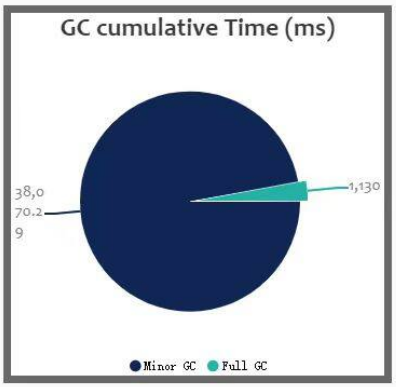
Interactive Graphs

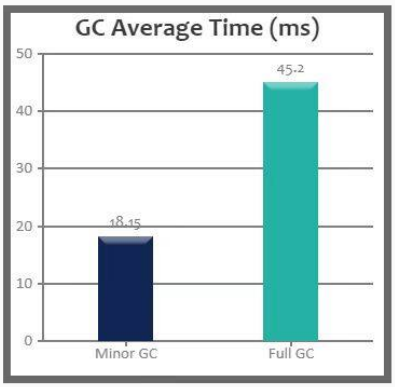
(All graphs are zoomable)



GC Statistics







Total GC stats

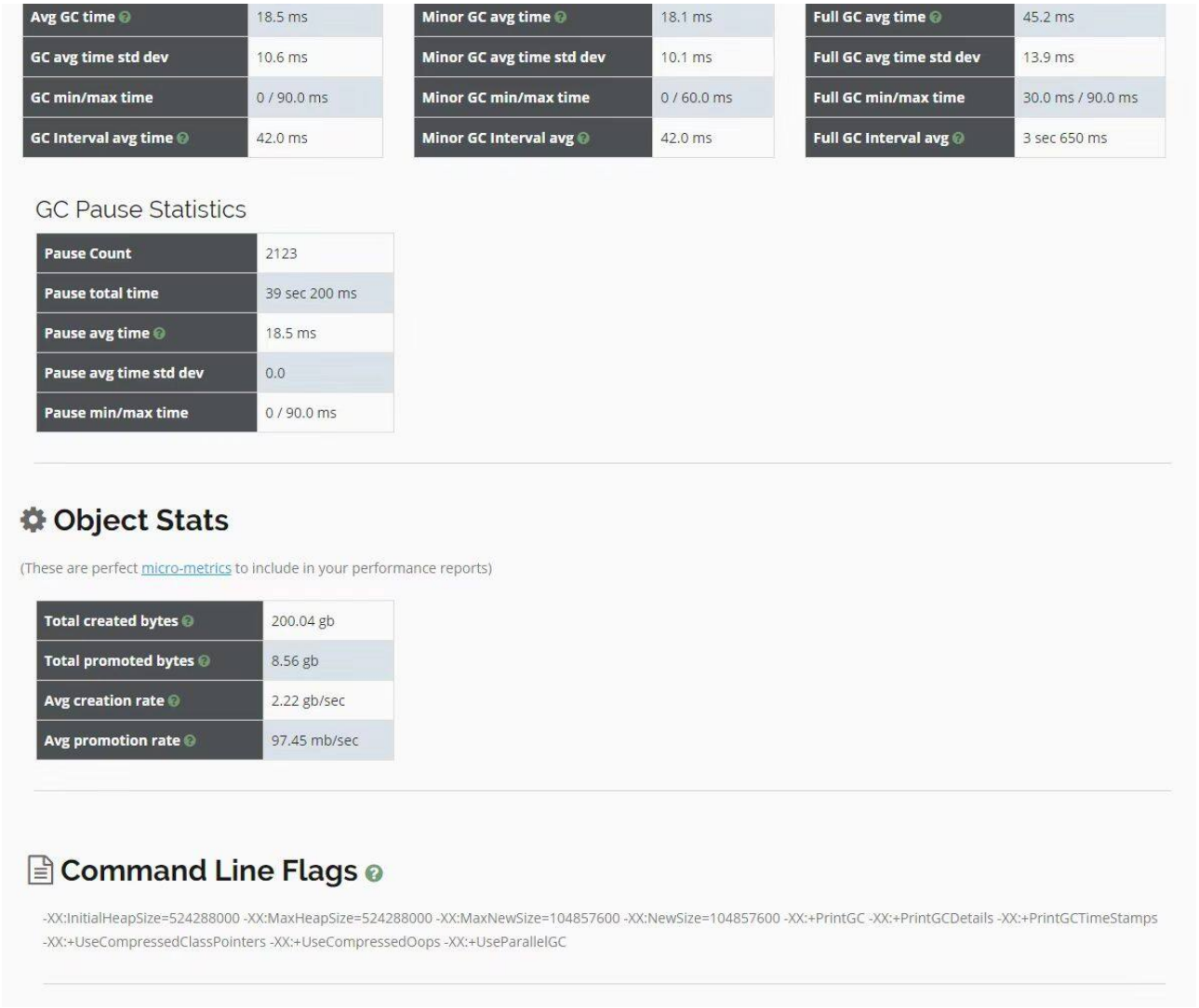
Total GC count	2123
Total reclaimed bytes	199.71 gb
Total GC time	39 sec 200 ms

Minor GC stats

Minor GC count	2098
Minor GC reclaimed	191.47 gb
Minor GC total time	38 sec 70 ms

Full GC stats

Full GC Count	25
Full GC reclaimed	8.25 gb
Full GC total time	1 sec 130 ms



Avg GC time	18.5 ms
GC avg time std dev	10.6 ms
GC min/max time	0 / 90.0 ms
GC Interval avg time	42.0 ms

Minor GC avg time	18.1 ms
Minor GC avg time std dev	10.1 ms
Minor GC min/max time	0 / 60.0 ms
Minor GC Interval avg	42.0 ms

Full GC avg time	45.2 ms
Full GC avg time std dev	13.9 ms
Full GC min/max time	30.0 ms / 90.0 ms
Full GC Interval avg	3 sec 650 ms

GC 调优策略

找出问题后，就可以进行调优了，下面介绍几种常用的 GC 调优策略。

1. 降低 Minor GC 频率

通常情况下，由于新生代空间较小，Eden 区很快被填满，就会导致频繁 Minor GC，因此我们可以通过增大新生代空间来降低 Minor GC 的频率。

可能你会有这样的疑问，扩容 Eden 区虽然可以减少 Minor GC 的次数，但不会增加单次 Minor GC 的时间吗？如果单次 Minor GC 的时间增加，那也很难达到我们期待的优化效果呀。

我们知道，单次 Minor GC 时间是由两部分组成：T1（扫描新生代）和 T2（复制存活对象）。假设一个对象在 Eden 区的存活时间为 500ms，Minor GC 的时间间隔是 300ms，那么正常情况下，Minor GC 的时间为 ： T1+T2。

当我们增大新生代空间，Minor GC 的时间间隔可能会扩大到 600ms，此时一个存活 500ms 的对象就会在 Eden 区中被回收掉，此时就不存在复制存活对象了，所以再发生 Minor GC 的时间为：两次扫描新生代，即 2T1。

可见，扩容后，Minor GC 时增加了 T1，但省去了 T2 的时间。通常在虚拟机中，复制对象的成本要远高于扫描成本。

如果在堆内存中存在较多的长期存活的对象，此时增加年轻代空间，反而会增加 Minor GC 的时间。如果堆中的短期对象很多，那么扩容新生代，单次 Minor GC 时间不会显著增加。因此，单次 Minor GC 时间更多取决于 GC 后存活对象的数量，而非 Eden 区的大小。

2. 降低 Full GC 的频率

通常情况下，由于堆内存空间不足或老年代对象太多，会触发 Full GC，频繁的 Full GC 会带来上下文切换，增加系统的性能开销。我们可以使用哪些方法来降低 Full GC 的频率呢?

减少创建大对象：在平常的业务场景中，我们习惯一次性从数据库中查询出一个大对象用于 web 端显示。例如，我之前碰到过一个一次性查询出 60 个字段的业务操作，这种大对象如果超过年轻代最大对象阈值，会被直接创建在老年代；即使被创建在了年轻代，由于年轻代的内存空间有限，通过 Minor GC 之后也会进入到老年代。这种大对象很容易产生较多的 Full GC。

我们可以将这种大对象拆解出来，首次只查询一些比较重要的字段，如果还需要其它字段辅助查看，再通过第二次查询显示剩余的字段。

增大堆内存空间：在堆内存不足的情况下，增大堆内存空间，且设置初始化堆内存为最大堆内存，也可以降低 Full GC 的频率。

选择合适的 GC 回收器

假设我们有这样一个需求，要求每次操作的响应时间必须在 500ms 以内。这个时候我们一般会选择响应速度较快的 GC 回收器，CMS（Concurrent Mark Sweep）回收器和 G1 回收器都是不错的选择。

而当我们的需求对系统吞吐量有要求时，就可以选择 Parallel Scavenge 回收器来提高系统的吞吐量。

总结

今天的内容比较多，最后再强调几个重点。

垃圾收集器的种类很多，我们可以将其分成两种类型，一种是响应速度快，一种是吞吐量高。通常情况下，CMS 和 G1 回收器的响应速度快，Parallel Scavenge 回收器的吞吐量高。

在 JDK1.8 环境下，默认使用的是 Parallel Scavenge（年轻代）+Serial Old（老年代）垃圾收集器，你可以通过文中介绍的查询 JVM 的 GC 默认配置方法进行查看。

通常情况，JVM 是默认垃圾回收优化的，在没有性能衡量标准的前提下，尽量避免修改 GC 的一些性能配置参数。如果一定要改，那就必须基于大量的测试结果或线上的具体性能来进行调整。

思考题

以上我们讲到了 CMS 和 G1 回收器，你知道 G1 是如何实现更好的 GC 性能的吗?

解答：

1 minor gc是否会导致stop the world？ 2 major gc什么时候会发生，它和full gc的区别是什么?

> 1、不管什么GC，都会发送stop the world，区别是发生的时间长短。而这个时间跟垃圾收集器又有关系，Serial、PartNew、Parallel Scavenge收集器无论是串行还是并行，都会挂起用户线程，而CMS和G1在并发标记时，是不会挂起用户线程，但其他时候一样会挂起用户线程，stop the world的时间相对来说小很多了。
>
> 2、major gc很多参考资料指的是等价于full gc，我们也可以发现很多性能监测工具中只有minor gc和full gc。一般情况下，一次full gc将会对年轻代、老年代以及元空间、堆外内存进行垃圾回收。而触发Full GC的原因有很多： a、当年轻代晋升到老年代的对象大小比目前老年代剩余的空间大小还要大时，此时会触发Full GC； b、当老年代的空间使用率超过某阈值时，此时会触发Full GC; c、当元空间不足时（JDK1.7永久代不足），也会触发Full GC; d、当调用System.gc()也会安排一次Full GC;

上一页 下一页