29 GC 疑难情况问题排查与分析（上篇）

本章介绍导致 GC 性能问题的典型情况。相关示例都来源于生产环境，为演示需要做了一定程度的精简。

> **名词说明**：Allocation Rate，翻译为“分配速率”，而不是分配率。因为不是百分比，而是单位时间内分配的量。同理，Promotion Rate 翻译为“提升速率”。

高分配速率（High Allocation Rate）

分配速率（Allocation Rate）表示单位时间内分配的内存量。通常使用 MB/sec 作为单位，也可以使用 PB/year 等。分配速率过高就会严重影响程序的性能，在 JVM 中可能会导致巨大的 GC 开销。

如何测量分配速率?

通过指定 JVM 参数： `-XX:+PrintGCDetails -XX:+PrintGCTimeStamps` ，通过 GC 日志来计算分配速率。GC 日志如下所示：

```
0.291: [GC (Allocation Failure)
        [PSYoungGen: 33280K->5088K(38400K)]
        33280K->24360K(125952K), 0.0365286 secs]
    [Times: user=0.11 sys=0.02, real=0.04 secs]
0.446: [GC (Allocation Failure)
        [PSYoungGen: 38368K->5120K(71680K)]
        57640K->46240K(159232K), 0.0456796 secs]
    [Times: user=0.15 sys=0.02, real=0.04 secs]
0.829: [GC (Allocation Failure)
        [PSYoungGen: 71680K->5120K(71680K)]
        112800K->81912K(159232K), 0.0861795 secs]
    [Times: user=0.23 sys=0.03, real=0.09 secs]
```

具体就是计算上一次垃圾收集之后，与下一次 GC 开始之前的年轻代使用量，两者的差值除以时间，就是分配速率。通过上面的日志，可以计算出以下信息：

- JVM 启动之后 291ms，共创建了 33280KB 的对象。第一次 Minor GC（小型 GC）完成后，年轻代中还有 5088KB 的对象存活。
- 在启动之后 446ms，年轻代的使用量增加到 38368KB，触发第二次 GC，完成后年轻代的使用量减少到 5120KB。
- 在启动之后 829ms，年轻代的使用量为 71680KB，GC 后变为 5120KB。

可以通过年轻代的使用量来计算分配速率，如下表所示：

Event	Time	Young before	Young after	Allocated during	Allocation rate
1st GC	291ms	33,280KB	5,088KB	33,280KB	114MB/sec
2nd GC	446ms	38,368KB	5,120KB	33,280KB	215MB/sec
3rd GC	829ms	71,680KB	5,120KB	66,560KB	174MB/sec
Total	829ms	N/A	N/A	133,120KB	161MB/sec

通过这些信息可以知道，在此期间，该程序的内存分配速率为 16MB/sec。

分配速率的意义

分配速率的变化，会增加或降低 GC 暂停的频率，从而影响吞吐量。但只有年轻代的 Minor GC 受分配速率的影响，老年代 GC 的频率和持续时间一般不受 **分配速率**（Allocation Rate）的直接影响（想想为什么？），而是受到 **提升速率**（Promotion Rate）的影响，请参见下文。

现在我们只关心 Minor GC 暂停，查看年轻代的 3 个内存池。因为对象在 Eden 区分配，所以我们一起来看 Eden 区的大小和分配速率的关系。看看增加 Eden 区的容量，能不能减少 Minor GC 暂停次数，从而使程序能够维持更高的分配速率。

经过我们的实验，通过参数 `-XX:NewSize`、`-XX:MaxNewSize` 以及 `-XX:SurvivorRatio` 设置不同的 Eden 空间，运行同一程序时，可以发现：

- Eden 空间为 100MB 时，分配速率低于 100MB/秒。
- 将 Eden 区增大为 1GB，分配速率也随之增长，大约等于 200MB/秒。

为什么会这样?

因为减少 GC 暂停，就等价于减少了任务线程的停顿，就可以做更多工作，也就创建了更多对象，所以对同一应用来说，分配速率越高越好。

在得出“Eden 区越大越好”这个结论前，我们注意到：分配速率可能会、也可能不会影响程序的实际吞吐量。

总而言之，吞吐量和分配速率有一定关系，因为分配速率会影响 Minor GC 暂停，但对于总体吞吐量的影响，还要考虑 Major GC 暂停等。

示例

参考 Demo 程序。假设系统连接了一个外部的数字传感器。应用通过专有线程，不断地获取传感器的值（此处使用随机数模拟），其他线程会调用 processSensorValue() 方法，传入传感器的值来执行某些操作。

```
public class BoxingFailure {
  private static volatile Double sensorValue;

  private static void readSensor() {
    while(true) sensorValue = Math.random();
  }

  private static void processSensorValue(Double value) {
    if(value != null) {
      //...
    }
  }
}
```

如同类名所示，这个 Demo 是模拟 boxing 的。为了 null 值判断，使用的是包装类型 Double。程序基于传感器的最新值进行计算，但从传感器取值是一个耗时的操作，所以采用了异步方式：一个线程不断获取新值，计算线程则直接使用暂存的最新值，从而避免同步等待。

Demo 程序在运行的过程中，由于分配速率太大而受到 GC 的影响。下面将确认问题，并给出解决办法。

高分配速率对 JVM 的影响

首先，我们应该检查程序的吞吐量是否降低。如果创建了过多的临时对象，Minor GC 的次数就会增加。如果并发较大，则 GC 可能会严重影响吞吐量。

遇到这种情况时，GC 日志将会像下面这样，当然这是上面的示例程序 产生的 GC 日志。

JVM 启动参数为： `-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -Xmx32m` 。

```
2.808: [GC (Allocation Failure)
        [PSYoungGen: 9760K->32K(10240K)], 0.0003076 secs]
2.819: [GC (Allocation Failure)
        [PSYoungGen: 9760K->32K(10240K)], 0.0003079 secs]
2.830: [GC (Allocation Failure)
        [PSYoungGen: 9760K->32K(10240K)], 0.0002968 secs]
2.842: [GC (Allocation Failure)
        [PSYoungGen: 9760K->32K(10240K)], 0.0003374 secs]
2.853: [GC (Allocation Failure)
        [PSYoungGen: 9760K->32K(10240K)], 0.0004672 secs]
2.864: [GC (Allocation Failure)
        [PSYoungGen: 9760K->32K(10240K)], 0.0003371 secs]
2.875: [GC (Allocation Failure)
        [PSYoungGen: 9760K->32K(10240K)], 0.0003214 secs]
2.886: [GC (Allocation Failure)
        [PSYoungGen: 9760K->32K(10240K)], 0.0003374 secs]
2.896: [GC (Allocation Failure)
        [PSYoungGen: 9760K->32K(10240K)], 0.0003588 secs]
```

很显然 Minor GC 的频率太高了。这说明创建了大量的对象。另外，年轻代在 GC 之后的使用量又很低，也没有 Full GC 发生。种种迹象表明，GC 对吞吐量造成了严重的影响。

解决方案

在某些情况下，只要增加年轻代的大小，即可降低分配速率过高所造成的影响。增加年轻代空间并不会降低分配速率，但是会减少 GC 的频率。如果每次 GC 后只有少量对象存活，Minor GC 的暂停时间就不会明显增加。

运行 示例程序 时，增加堆内存大小（同时也就增大了年轻代的大小），使用的 JVM 参数为： `-Xmx64m` 。

```
2.808: [GC (Allocation Failure)
        [PSYoungGen: 20512K->32K(20992K)], 0.0003748 secs]
2.831: [GC (Allocation Failure)
        [PSYoungGen: 20512K->32K(20992K)], 0.0004538 secs]
2.855: [GC (Allocation Failure)
        [PSYoungGen: 20512K->32K(20992K)], 0.0003355 secs]
```

```
2.879: [GC (Allocation Failure)
        [PSYoungGen: 20512K->32K(20992K)], 0.0005592 secs]
```

但有时候增加堆内存的大小，并不能解决问题。

通过前面学到的知识，我们可以通过分配分析器找出大部分垃圾产生的位置。实际上，在此示例中 99% 的对象属于 Double 包装类，在readSensor 方法中创建。

最简单的优化，将创建的 Double 对象替换为原生类型 double，而针对 null 值的检测，可以使用 Double.NaN 来进行。

由于原生类型不算是对象，也就不会产生垃圾，导致 GC 事件。

优化之后，不在堆中分配新对象，而是直接覆盖一个属性域即可。对示例程序进行简单的改造（查看 diff）后，GC 暂停基本上完全消除。

有时候 JVM 也很智能，会使用逃逸分析技术（Escape Analysis Technique）来避免过度分配。

简单来说，JIT 编译器可以通过分析得知，方法创建的某些对象永远都不会“逃出”此方法的作用域。这时候就不需要在堆上分配这些对象，也就不会产生垃圾，所以 JIT 编译器的一种优化手段就是：消除堆上内存分配（请参考基准测试）。

过早提升（Premature Promotion）

提升速率（Promotion Rate）用于衡量单位时间内从年轻代提升到老年代的数据量。一般使用 MB/sec 作为单位，和“分配速率”类似。

JVM 会将长时间存活的对象从年轻代提升到老年代。根据分代假设，可能存在一种情况，老年代中不仅有存活时间长的对象，也可能有存活时间短的对象。这就是过早提升：对象存活时间还不够长的时候就被提升到了老年代。

Major GC 不是为频繁回收而设计的，但 Major GC 现在也要清理这些生命短暂的对象，就会导致 GC 暂停时间过长。这会严重影响系统的吞吐量。

如何测量提升速率

可以指定 JVM 参数 `-XX:+PrintGCDetails -XX:+PrintGCTimeStamps` ，通过 GC 日志来测量提升速率。JVM 记录的 GC 暂停信息如下所示：

```
0.291: [GC (Allocation Failure)
        [PSYoungGen: 33280K->5088K(38400K)]
        33280K->24360K(125952K), 0.0365286 secs]
    [Times: user=0.11 sys=0.02, real=0.04 secs]
0.446: [GC (Allocation Failure)
        [PSYoungGen: 38368K->5120K(71680K)]
        57640K->46240K(159232K), 0.0456796 secs]
    [Times: user=0.15 sys=0.02, real=0.04 secs]
0.829: [GC (Allocation Failure)
        [PSYoungGen: 71680K->5120K(71680K)]
        112800K->81912K(159232K), 0.0861795 secs]
    [Times: user=0.23 sys=0.03, real=0.09 secs]
```

从上面的日志可以得知：GC 之前和之后的年轻代使用量以及堆内存使用量。这样就可以通过差值算出老年代的使用量。GC 日志中的信息可以表述为：

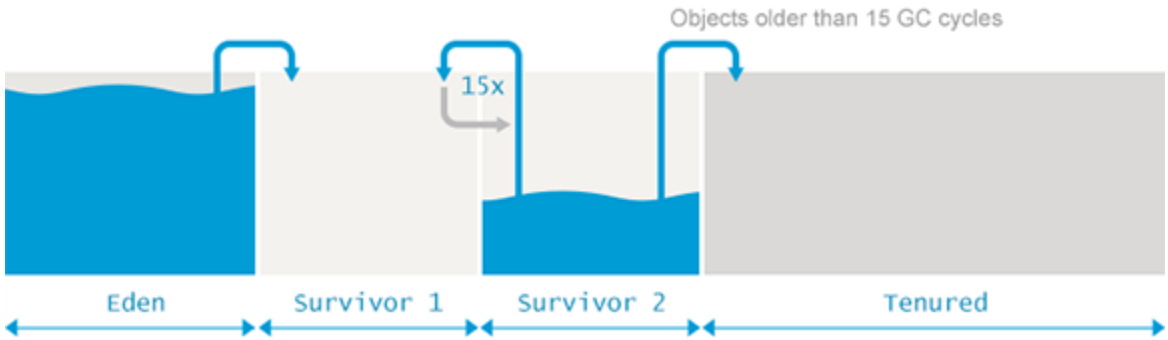
Event	Time	Young decreased	Total decreased	Promoted	Promotion rate
（事件）	（耗时）	（年轻代减少）	（整个堆内存减少）	（提升量）	（提升速率）
1st GC	291ms	28192K	8920K	19272K	66.2 MB/sec
2nd GC	446ms	33248K	11400K	21848K	140.95 MB/sec
3rd GC	829ms	66560K	30888K	35672K	93.14 MB/sec
Total	829ms			76792K	92.63 MB/sec

根据这些信息，就可以计算出观测周期内的提升速率：平均提升速率为 92MB/秒，峰值为 140.95MB/秒。

请注意，**只能根据 Minor GC 计算提升速率**。Full GC 的日志不能用于计算提升速率，因为 Major GC 会清理掉老年代中的一部分对象。

提升速率的意义

和分配速率一样，提升速率也会影响 GC 暂停的频率。但分配速率主要影响 minor GC，而提升速率则影响 major GC 的频率。有大量的对象提升，自然很快将老年代填满。老年代填充的越快，则 Major GC 事件的频率就会越高。



前面章节提到过，Full GC 通常需要更多的时间，因为需要处理更多的对象，还要执行碎片整理等额外的复杂过程。

示例

让我们看一个过早提升的示例。这个程序创建/获取大量的对象/数据，并暂存到集合之中，达到一定数量后进行批处理：

```
public class PrematurePromotion {

   private static final Collection<byte[]> accumulatedChunks
               = new ArrayList<>();

   private static void onNewChunk(byte[] bytes) {
       accumulatedChunks.add(bytes);

       if(accumulatedChunks.size() > MAX_CHUNKS) {
           processBatch(accumulatedChunks);
           accumulatedChunks.clear();
       }
   }
}
```

此 Demo 程序 受到过早提升的影响。下面将进行验证并给出解决办法。

过早提升的影响

一般来说过早提升的症状表现为以下形式：

- 短时间内频繁地执行 Full GC
- 每次 Full GC 后老年代的使用率都很低，在 10~20% 或以下
- 提升速率接近于分配速率

要演示这种情况稍微有点麻烦，所以我们使用特殊手段，让对象提升到老年代的年龄比默认情况小很多。指定 GC 参数 `-Xmx24m -XX:NewSize=16m -XX:MaxTenuringThreshold=1` ，运行程序之后，可以看到下面的 GC 日志：

```
2.176: [Full GC (Ergonomics)
        [PSYoungGen: 9216K->0K(10752K)]
        [ParOldGen: 10020K->9042K(12288K)]
        19236K->9042K(23040K), 0.0036840 secs]
2.394: [Full GC (Ergonomics)
        [PSYoungGen: 9216K->0K(10752K)]
        [ParOldGen: 9042K->8064K(12288K)]
        18258K->8064K(23040K), 0.0032855 secs]
2.611: [Full GC (Ergonomics)
        [PSYoungGen: 9216K->0K(10752K)]
        [ParOldGen: 8064K->7085K(12288K)]
        17280K->7085K(23040K), 0.0031675 secs]
2.817: [Full GC (Ergonomics)
        [PSYoungGen: 9216K->0K(10752K)]
        [ParOldGen: 7085K->6107K(12288K)]
        16301K->6107K(23040K), 0.0030652 secs]
```

乍一看似乎不是过早提升的问题，每次 GC 之后老年代的使用率似乎在减少。但反过来想，要是没有对象提升或者提升率很小，也就不会看到这么多的 Full GC 了。

简单解释一下这里的 GC 行为：有很多对象提升到老年代，同时老年代中也有很多对象被回收了，这就造成了老年代使用量减少的假象。但事实是大量的对象不断地被提升到老年代，并触发 Full GC。

解决方案

简单来说，要解决这类问题，需要让年轻代存放得下暂存的数据。有两种简单的方法：

一是增加年轻代的大小，设置 JVM 启动参数，类似这样： `-Xmx64m -XX:NewSize=32m` ，程序在执行时，Full GC 的次数自然会减少很多，只会对 Minor GC 的持续时间产生影响：

```
2.251: [GC (Allocation Failure)
        [PSYoungGen: 28672K->3872K(28672K)]
        37126K->12358K(61440K), 0.0008543 secs]
2.776: [GC (Allocation Failure)
        [PSYoungGen: 28448K->4096K(28672K)]
```

```
36934K->16974K(61440K), 0.0033022 secs]
```

二是减少每次批处理的数量，也能得到类似的结果。

至于选用哪个方案，要根据业务需求决定。

在某些情况下，业务逻辑不允许减少批处理的数量，那就只能增加堆内存，或者重新指定年轻代的大小。

如果都不可行，就只能优化数据结构，减少内存消耗。但总体目标依然是一致的——让临时数据能够在年轻代存放得下。

上一页 下一页