

业界

移动开发

云计算

软件研发

程序员

极客头条

专题

大数据

数据中心

服务器

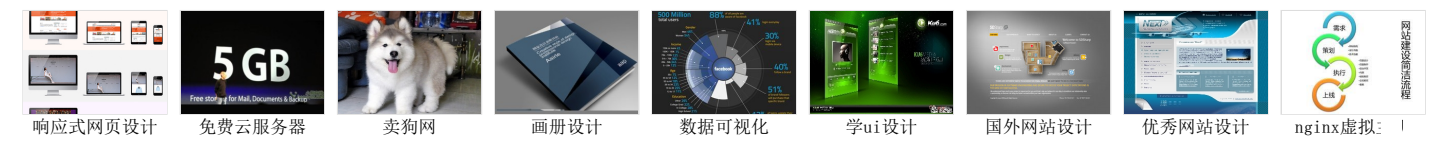
存储

虚拟化

NoSQL

安全

云先锋



CSDN首页 > 云计算

订阅云计算RSS

GC调优在Spark应用中的实践

发表于 2015-06-02 13:34 | 8227次阅读 | 来源 《程序员》电子刊5月B | 7条评论 | 作者 王道远, 黄洁

Spark

大数据

开源

Intel

摘要: Spark立足内存计算, 常常需要在内存中存放大量数据, 因此也更依赖JVM的垃圾回收机制。与此同时, 它也兼容批处理和流式处理, 对于程序吞吐量和延迟都有较高要求, 因此GC参数的调优在Spark应用实践中显得尤为重要。

Spark是时下非常热门的大数据计算框架, 以其卓越的性能优势、独特的架构、易用的用户接口和丰富的分析计算库, 正在工业界获得越来越广泛的应用。与Hadoop、HBase生态圈的众多项目一样, Spark的运行离不开JVM的支持。由于Spark立足于内存计算, 常常需要在内存中存放大量数据, 因此也更依赖JVM的垃圾回收机制(GC)。并且同时, 它也支持兼容批处理和流式处理, 对于程序吞吐量和延迟都有较高要求, 因此GC参数的调优在Spark应用实践中显得尤为重要。本文主要讲述如何针对Spark应用程序配置JVM的垃圾回收器, 并从实际案例出发, 剖析如何进行GC调优, 进一步提升Spark应用的性能。

问题介绍

随着Spark在工业界得到广泛使用, Spark应用稳定性以及性能调优问题不可避免地引起了用户的关注。由于Spark的特色在于内存计算, 我们在部署Spark集群时, 动辄使用超过100GB的内存作为Heap空间, 这在传统的Java应用中是比较少见的。在广泛的合作过程中, 确实有很多用户向我们抱怨运行Spark应用时GC所带来的各种问题。例如垃圾回收时间久、程序长时间无响应, 甚至造成程序崩溃或者作业失败。对此, 我们该怎样调试Spark应用的垃圾收集器呢? 在本文中, 我们从应用实例出发, 结合具体问题场景, 探讨了Spark应用的GC调优方法。

按照经验来说, 当我们配置垃圾收集器时, 主要有两种策略——Parallel GC和CMS GC。前者注重更高的吞吐量, 而后者则注重更低的延迟。两者似乎是鱼和熊掌, 不能兼得。在实际应用中, 我们只能根据应用对性能瓶颈的侧重性, 来选取合适的垃圾收集器。例如, 当我们运行需要有实时响应的场景的应用时, 我们一般选用CMS GC, 而运行一些离线分析程序时, 则选用Parallel GC。那么对于Spark这种既支持流式计算, 又支持传统的批处理运算的计算框架来说, 是否存在一组通用的配置选项呢?

通常CMS GC是企业比较常用的GC配置方案, 并在长期实践中取得了比较好的效果。例如对于进程中若存在大量寿命较长的对象, Parallel GC经常带来较大的性能下降。因此, 即使是批处理的程序也能从CMS GC中获益。不过, 在从1.6开始的HOTSPOT JVM中, 我们发现了一个新的GC设置项: Garbage-First GC(G1 GC)。Oracle将其定位为CMS GC的长期演进, 这让我们重燃了鱼与熊掌兼得的希望! 那么, 我们首先了解一下GC的一些相关原理吧。

GC算法原理

在传统JVM内存管理中, 我们把Heap空间分为Young/Old两个分区, Young分区又包括一个Eden和两个Survivor分区, 如图1所示。新产生的对象首先会被存放在Eden区, 而每次minor GC发生时, JVM一方面将Eden分区内存活的对象拷贝到一个空的Survivor分区, 另一方面将另一个正在被使用的Survivor分区中的存活对象也拷贝到空的Survivor分区内。在此过程中, JVM始终保持一个Survivor分区处于全空的状态。一个对象在两个Survivor之间的拷贝到一定次数后, 如果还是存活的, 就将其拷入Old分区。当Old分区没有足够空间时, GC会停下所有程序线程, 进行Full GC, 即对Old区中的对



CSDN官方微信

扫描二维码,向CSDN吐槽

微信号: CSDNnews

程序员移动端订阅下载

每日资讯快速浏览

微博关注

CSDN云计算

北京 朝阳区

加关注

- 给运维带来的变革与机会。 <http://t.cn/8s8szEZ>
- 10月16日 10:47 转发(1) | 评论
- 2016中国开源年会, 微软工作坊来讲, 带大家了解 Azure, 并现场进行开源LAMP建站实践。 <http://t.cn/8s8szEZ>
- 10月16日 09:45 转发(2) | 评论

象进行整理。这个所有线程都暂停的阶段被称为Stop-The-World(STW)，也是大多数GC算法中对性能影响最大的部分。

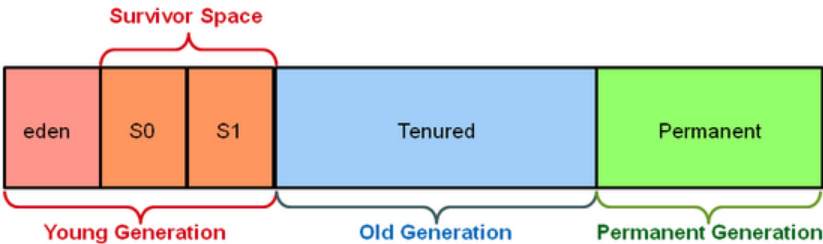


图 1 分年代的Heap结构

而G1 GC则完全改变了这一传统思路。它将整个Heap分为若干个预先设定的小区块（如图2），每个区域块内部不再进行新旧分区，而是将整个区域块标记为Eden/Survivor/Old。当创建新对象时，它首先被存放到某一个可用区块（Region）中。当该区块满了，JVM就会创建新的区块存放对象。当发生minor GC时，JVM将一个或几个区块中存活的对象拷贝到一个新的区块中，并在空余的空间中选择几个全新区块作为新的Eden分区。当所有区域中都有存活对象，找不到全空区块时，才发生Full GC。而在标记存活对象时，G1使用RememberSet的概念，将每个分区外指向分区内的引用记录在该分区的RememberSet中，避免了对整个Heap的扫描，使得各个分区的GC更加独立。在这样的背景下，我们可以看出G1 GC大大提高了触发Full GC时的Heap占用率，同时也使得Minor GC的暂停时间更加可控，对于内存较大的环境非常友好。这些颠覆性的改变，将给GC性能带来怎样的变化呢？最简单的方式，我们可以将老的GC设置直接迁移为G1 GC，然后观察性能变化。

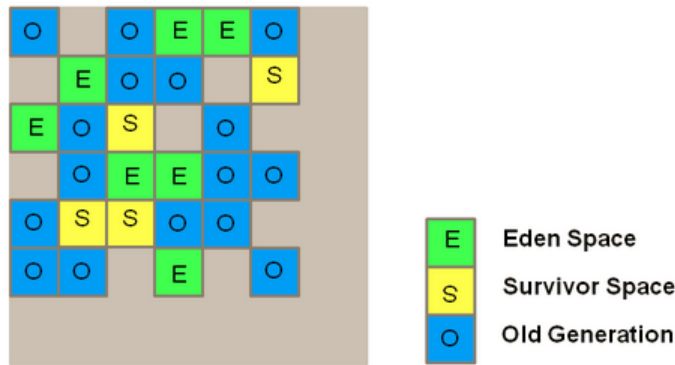


图 2 G1 Heap结构示意图

由于G1取消了对heap空间不同新旧对象固定分区概念，所以我们需要在GC配置选项上作相应的调整，使得应用能够合理地运行在G1 GC收集器上。一般来说，对于原运行在Parallel GC上的应用，需要去除的参数包括-Xmn, -XX:-UseAdaptiveSizePolicy, -XX:SurvivorRatio=n等；而对于原来使用CMS GC的应用，我们需要去掉-Xmn -XX:InitialSurvivorRatio -XX:SurvivorRatio -XX:InitialTenuringThreshold -XX:MaxTenuringThreshold等参数。另外在CMS中已经调优过的-XX:ParallelGCThreads -XX:ConcGCThreads参数最好也移除掉，因为对于CMS来说性能最好的不一定是对于G1性能最好的选择。我们先统一置为默认值，方便后期调优。此外，当应用开启的线程较多时，最好使用-XX:-ResizePLAB来关闭PLAB()的大小调整，以避免大量的线程通信所导致的性能下降。

关于Hotspot JVM所支持的完整的GC参数列表，可以使用参数-XX:+PrintFlagsFinal打印出来，也可以参见Oracle官方的文档中对部分参数的解释。

Spark的内存管理

Spark的核心概念是RDD，实际运行中内存消耗都与RDD密切相关。Spark允许用户将应用中重复使用的RDD数据持久化缓存起来，从而避免反复计算的开销，而RDD的持久化形态之一就是全部或者部分数据缓存在JVM的Heap中。Spark Executor会将JVM的heap空间大致分为两个部分，一部分用来存放Spark应用中持久化到内存中的RDD数据，剩下的部分则用来作为JVM运行时的堆空间，负责RDD转化等过程中的内存消耗。我们可以通过spark.storage.memoryFraction参数调节这两块内存的比例，Spark会控制缓存RDD总大小不超过heap空间体积乘以这个参数所设置的值，而这块缓存RDD



5 GB
Free storage for Mail, Documents & Backup



星河湾二手房



免费云服务器



卖狗网



云服务器搭建

相关热门文章

热门标签

Hadoop	AWS	移动游戏
Java	Android	iOS
Swift	智能硬件	Docker
OpenStack	VPN	Spark
ERP	IE10	Eclipse
CRM	JavaScript	数据库
Ubuntu	NFC	...

下载专辑



CSDN主题月直播技术专辑



servlet和jsp



mysql



proxool数据库连接池相关jar包



算法与数据结构汇总

的空间中没有使用的部分也可以为JVM运行时所用。因此，分析Spark应用GC问题时应当分别分析两部分内存的使用情况。

而当我们观察到GC延迟影响效率时，应当先检查Spark应用本身是否有效利用有限的内存空间。RDD占用的内存空间比较少的话，程序运行的heap空间也会比较宽松，GC效率也会相应提高；而RDD如果占用大量空间的话，则会带来巨大的性能损失。下面我们从一个用户案例展开：

该应用是利用Spark的组件Bagel来实现的，其本质就是一个简单的迭代计算。而每次迭代计算依赖于上一次的迭代结果，因此每次迭代结果都会被主动持续化到内存空间中。当运行用户程序时，我们观察到随着迭代次数的增加，进程占用的内存空间不断快速增长，GC问题越来越突出。但是，仔细分析Bagel实现机制，我们很快发现Bagel将每次迭代产生的RDD都持久化下来了，而没有及时释放掉不再使用的RDD，从而造成了内存空间不断增长，触发了更多GC执行。经过简单的修改，我们修复了这个问题（SPARK-2661）。应用的内存空间得到了有效的控制后，迭代次数三次以后RDD大小趋于稳定，缓存空间得到有效控制（如表1所示），GC效率得以大大提高，程序总的运行时间缩短了10%~20%。

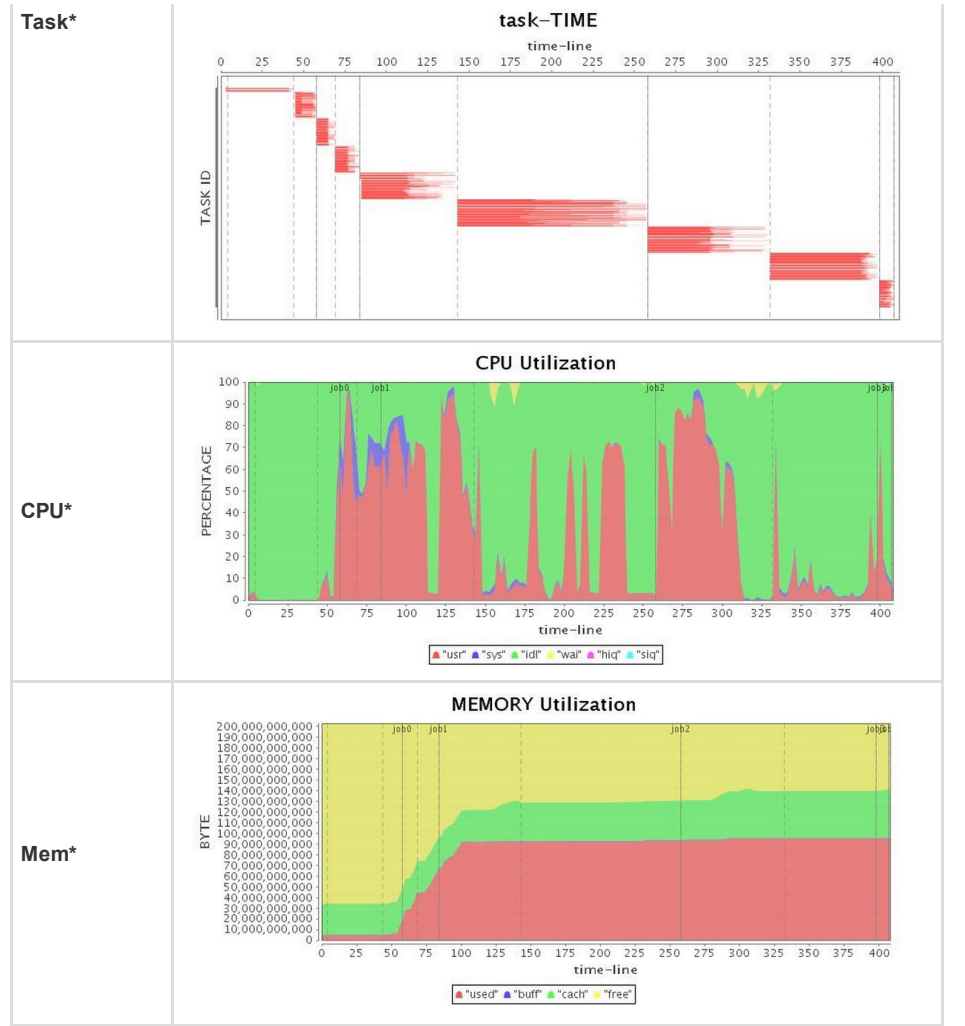
迭代轮数	单次迭代缓存大小	总缓存大小(优化前)	总缓存大小(优化后)
初始化	4.3GB	4.3GB	4.3GB
1	8.2GB	12.5GB	8.2GB
2	98.8GB	111.3GB	98.8GB
3	90.8GB	202.1GB	90.8GB

小结：当观察到GC频繁或者延时的情况，也可能是Spark进程或者应用中内存空间没有有效利用。所以可以尝试检查是否存在RDD持久化后未得到及时释放等情况。

选择垃圾收集器

在解决了应用本身的问题之后，我们就要开始针对Spark应用的GC调优了。基于修复了SPARK-2661的Spark版本，我们搭建了一个4个节点的集群，给每个Executor分配88G的Heap，在Spark的Standalone模式下来进行我们的实验。在使用默认的Parallel GC运行我们的Spark应用时，我们发现，由于Spark应用对于内存的开销比较大，而且大部分对象并不能在一个较短的生命周期中被回收，Parallel GC也常常受困于Full GC，而每次Full GC都给性能带来了较大的下降。而Parallel GC可以进行参数调优的空间也非常有限，我们只能通过调节一些基本参数来提高性能，如各年代分区大小比例、进入老年代前的拷贝次数等。而且这些调优策略只能推迟Full GC的到来，如果是长期运行的应用，Parallel GC调优的意义就非常有限了。因此，本文中不会再对Parallel GC进行调优。表2列出了Parallel GC的运行情况，其中CPU利用率较低的部分正是发生Full GC的时候。

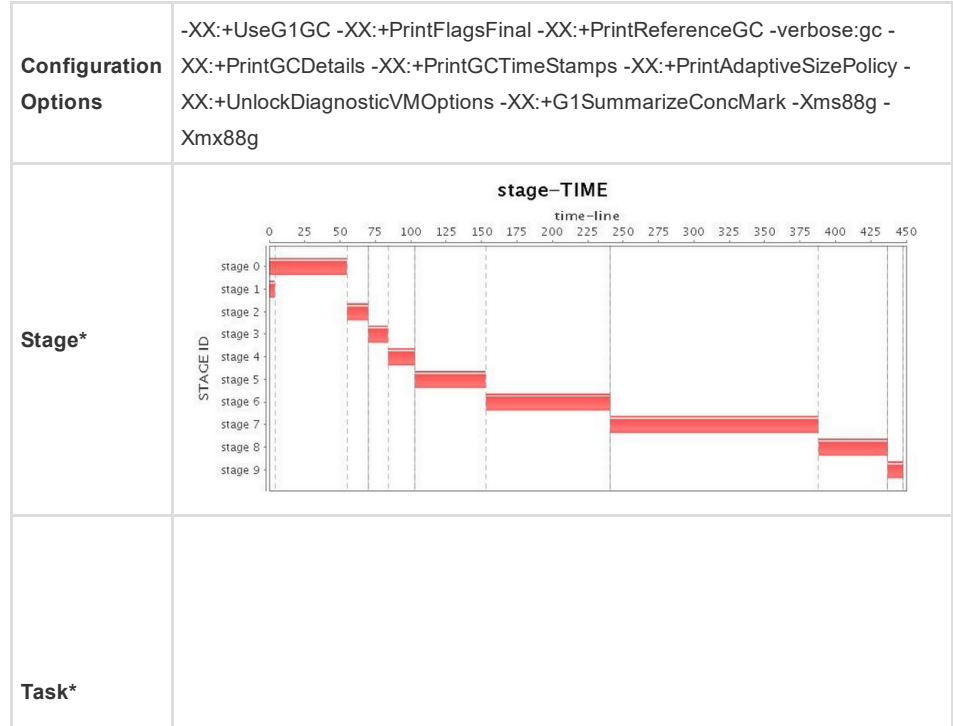
Configuration Options	-XX:+UseParallelGC -XX:+UseParallelOldGC -XX:+PrintFlagsFinal -XX:+PrintReferenceGC -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintAdaptiveSizePolicy -Xms88g -Xmx88g
Stage*	<div><div>stage-TIME</div><div>time-line</div><div><div>Tuning Java Garbage Collection for Spark Applications - Google Docs</div></div></div>



Parallel GC运行情况(未调优)

至于CMS GC，也没有办法消除这个Spark应用中的Full GC，而且CMS的Full GC的暂停时间远远超过了Parallel GC，大大拖累了该应用的吞吐量。

接下来，我们就使用最基本的G1 GC配置来运行我们的应用。实验结果发现，G1 GC竟然也出现了不可忍受的Full GC（表3的CPU利用率图中，可以明显发现Job 3中出现了将近100秒的暂停），超长的暂停时间大大拖累了整个应用的运行。如表4所示，虽然总的运行时间比Parallel GC略长，不过G1 GC表现略好于CMS GC。



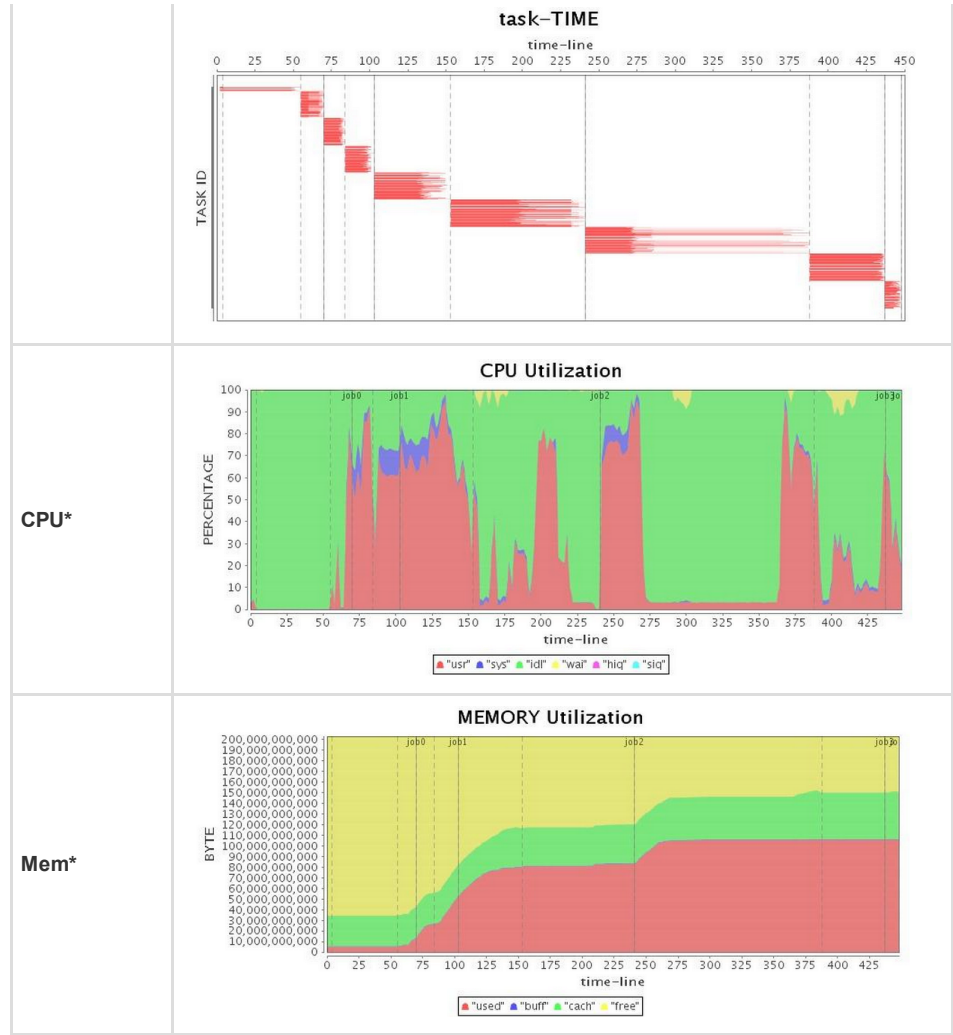


表 3 G1 GC运行情况(未调优)

垃圾收集器	88GB heap 运行时间
Parallel GC	6.5min
CMS GC	9min
G1 GC	7.6min

表 4 三种垃圾收集器对应的程序运行时间比较（88GB heap未调优）

根据日志进一步调优

在让G1 GC跑起来之后，我们下一步就是需要根据GC log，来进一步进行性能调优。首先，我们要让JVM记录比较详细的GC日志。对于Spark而言，我们需要在SPARK_JAVA_OPTS中设置参数使得Spark保留下我们需要用到的日志。一般而言，我们需要设置这样一串参数：

```
-XX:+PrintFlagsFinal -XX:+PrintReferenceGC -verbose:gc -XX:+PrintGCDetails -
XX:+PrintGCTimeStamps -XX:+PrintAdaptiveSizePolicy -
XX:+UnlockDiagnosticVMOptions -XX:+G1SummarizeConcMark
```

有了这些参数，我们就可以在SPARK的EXECUTOR日志中（默认输出到各worker节点的\$SPARK_HOME/work/\$app_id/\$executor_id/stdout中）读到详尽的GC日志以及生效的GC参数了。接下来，我们就可以根据GC日志来分析问题，使程序获得更优性能。我们先来了解一下G1中一次GC的日志结构。

```
251.354: [G1Ergonomics (Mixed GCs) continue mixed GCs, reason: candidate old regions
available, candidate old regions: 363 regions, reclaimable: 9830652576 bytes (10.40
```

```
%), threshold: 10.00 %]

[Parallel Time: 145.1 ms, GC Workers: 23]

[GC Worker Start (ms): Min: 251176.0, Avg: 251176.4, Max: 251176.7, Diff: 0.7]

[Ext Root Scanning (ms): Min: 0.8, Avg: 1.2, Max: 1.7, Diff: 0.9, Sum: 28.1]

[Update RS (ms): Min: 0.0, Avg: 0.3, Max: 0.6, Diff: 0.6, Sum: 5.8]

[Processed Buffers: Min: 0, Avg: 1.6, Max: 9, Diff: 9, Sum: 37]

[Scan RS (ms): Min: 6.0, Avg: 6.2, Max: 6.3, Diff: 0.3, Sum: 143.0]

[Object Copy (ms): Min: 136.2, Avg: 136.3, Max: 136.4, Diff: 0.3, Sum: 3133.9]

[Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.3]

[GC Worker Other (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.2, Sum: 1.9]

[GC Worker Total (ms): Min: 143.7, Avg: 144.0, Max: 144.5, Diff: 0.8, Sum: 3313.0]

[GC Worker End (ms): Min: 251320.4, Avg: 251320.5, Max: 251320.6, Diff: 0.2]

[Code Root Fixup: 0.0 ms]

[Clear CT: 6.6 ms]

[Other: 26.8 ms]

[Choose CSet: 0.2 ms]

[Ref Proc: 16.6 ms]

[Ref Enq: 0.9 ms]

[Free CSet: 2.0 ms]

[Eden: 3904.0M(3904.0M)->0.0B(4448.0M) Survivors: 576.0M->32.0M Heap: 63.7G(88.0G)->58.3G(88.0G)]

[Times: user=3.43 sys=0.01, real=0.18 secs]
```

以G1 GC的一次mixed GC为例，从这段日志中，我们可以看到G1 GC日志的层次是非常清晰的。日志列出了这次暂停发生的时间、原因，并分级各种线程所消耗的时长以及CPU时间的均值和最值。最后，G1 GC列出了本次暂停的清理结果，以及总共消耗的时间。

而在我们现在的G1 GC运行日志中，我们明显发现这样一段特殊的日志：

```
(to-space exhausted), 1.0552680 secs]

[Parallel Time: 958.8 ms, GC Workers: 23]

[GC Worker Start (ms): Min: 759925.0, Avg: 759925.1, Max: 759925.3, Diff: 0.3]

[Ext Root Scanning (ms): Min: 1.1, Avg: 1.4, Max: 1.8, Diff: 0.6, Sum: 33.0]

[SATB Filtering (ms): Min: 0.0, Avg: 0.0, Max: 0.3, Diff: 0.3, Sum: 0.3]
```

```
[Update RS (ms): Min: 0.0, Avg: 1.2, Max: 2.1, Diff: 2.1, Sum: 26.9]

[Processed Buffers: Min: 0, Avg: 2.8, Max: 11, Diff: 11, Sum: 65]

[Scan RS (ms): Min: 1.6, Avg: 2.5, Max: 3.0, Diff: 1.4, Sum: 58.0]

[Object Copy (ms): Min: 952.5, Avg: 953.0, Max: 954.3, Diff: 1.7, Sum: 21919.4]

[Termination (ms): Min: 0.0, Avg: 0.1, Max: 0.2, Diff: 0.2, Sum: 2.2]

[GC Worker Other (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.6]

[GC Worker Total (ms): Min: 958.1, Avg: 958.3, Max: 958.4, Diff: 0.3, Sum: 22040.4]

[GC Worker End (ms): Min: 760883.4, Avg: 760883.4, Max: 760883.4, Diff: 0.0]

[Code Root Fixup: 0.0 ms]

[Clear CT: 0.4 ms]

[Other: 96.0 ms]

[Choose CSet: 0.0 ms]

[Ref Proc: 0.4 ms]

[Ref Enq: 0.0 ms]

[Free CSet: 0.1 ms]

[Eden: 160.0M(3904.0M)->0.0B(4480.0M) Survivors: 576.0M->0.0B Heap: 87.7G(88.0G)->87.7G(88.0G)]

[Times: user=1.69 sys=0.24, real=1.05 secs]

760.981: [G1Ergonomics (Heap Sizing) attempt heap expansion, reason: allocation
request failed, allocation request: 90128 bytes]

760.981: [G1Ergonomics (Heap Sizing) expand the heap, requested expansion amount:
33554432 bytes, attempted expansion amount: 33554432 bytes]

760.981: [G1Ergonomics (Heap Sizing) did not expand the heap, reason: heap expansion
operation failed]

760.981: [Full GC 87G->36G(88G), 67.4381220 secs]
```

显然最大的性能下降是这样的**Full GC**导致的，我们可以在日志中看到类似**To-space Exhausted**或者**To-space Overflow**这样的输出（取决于不同版本的JVM，输出略有不同）。这是**G1 GC**收集器在将某个需要垃圾回收的分区进行回收时，无法找到一个能将其中存活对象拷贝过去的空闲分区。这种情况被称为**Evacuation Failure**，常常会引发**Full GC**。而且很显然，**G1 GC**的**Full GC**效率相对于**Parallel GC**实在是相差太远，我们想要获得比**Parallel GC**更好的表现，一定要尽力规避**Full GC**的出现。对于这种情况，我们常见的处理办法有两种：

1. 将**InitiatingHeapOccupancyPercent**参数调低（默认值是**45**），可以使**G1 GC**收集器更早开始**Mixed GC**；但另一方面，会增加GC发生频率。
2. 提高**ConcGCThreads**的值，在**Mixed GC**阶段投入更多的并发线程，争取提高每次暂停的效率。但是此参数会占用一定的有效工作线程资源。

调试这两个参数可以有效降低Full GC出现的概率。Full GC被消除之后，最终的性能获得了大幅提升。但是我们发现，仍然有一些地方GC产生了大量的暂停时间。比如，我们在日志中读到很多类似这样的片断：

```
280.008: [G1Ergonomics (Concurrent Cycles) request concurrent cycle initiation,
reason: occupancy higher than threshold, occupancy: 62344134656 bytes, allocation
request: 46137368 bytes, threshold: 42520176225 bytes (45.00 %), source: concurrent
humongous allocation]
```

这里就是Humongous object，一些比G1的一个分区的一半更大的对象。对于这些对象，G1会专门在Heap上开出一个Humongous Area来存放，每个分区只放一个对象。但是申请这么大的空间是比较耗时的，而且这些区域也仅当Full GC时才进行处理，所以我们要尽量减少这样的对象产生。或者提高G1HeapRegionSize的值减少HumongousArea的创建。不过在内存比较大的时，JVM默认把这个值设到了最大(32M)，此时我们只能通过分析程序本身找到这些对象并且尽量减少这样的对象产生。当然，相信随着G1 GC的发展，在后期的版本中相信这个最大值也会越来越大，毕竟G1号称是在1024~2048个Region时能够获得最佳性能。

接下来，我们可以分析下单次cycle start到Mixed GC为止的时间间隔。如果这一时间过长，可以考虑进一步提升ConcGCThreads，需要注意的是，这会进一步占用一定CPU资源。

对于追求更短暂停时间的在线应用，如果观测到较长的Mixed GC pause，我们还要把G1RSetUpdatingPauseTimePercent调低，把G1ConcRefinementThreads调高。前文提到G1 GC通过为每个分区维护RememberSet来记录分区外对分区内的引用，G1RSetUpdatingPauseTimePercent则正是在STW阶段为G1收集器指定更新RememberSet的时间占总STW时间的期望比例，默认为10。而G1ConcRefinementThreads则是在程序运行时维护RememberSet的线程数目。通过对这两个值的对应调整，我们可以把STW阶段的RememberSet更新工作压力更多地移到Concurrent阶段。

另外，对于需要长时间运行的应用，我们不妨加上AlwaysPreTouch参数，这样JVM会在启动时就向OS申请所有需要使用的内存，避免动态申请，也可以提高运行时性能。但是该参数也会大大延长启动时间。

最终，经过几轮GC参数调试，其结果如下表5所示。较之先前的结果，我们最终还是获得了较满意的运行效率。

Configuration Options	-XX:+UseG1GC -XX:+PrintFlagsFinal -XX:+PrintReferenceGC -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintAdaptiveSizePolicy -XX:+UnlockDiagnosticVMOptions -XX:+G1SummarizeConcMark -Xms88g -Xmx88g -XX:InitiatingHeapOccupancyPercent=35 -XX:ConcGCThread=20
Stage*	<div>STAGE-TIME</div>
Task*	<div>TASK-TIME</div>

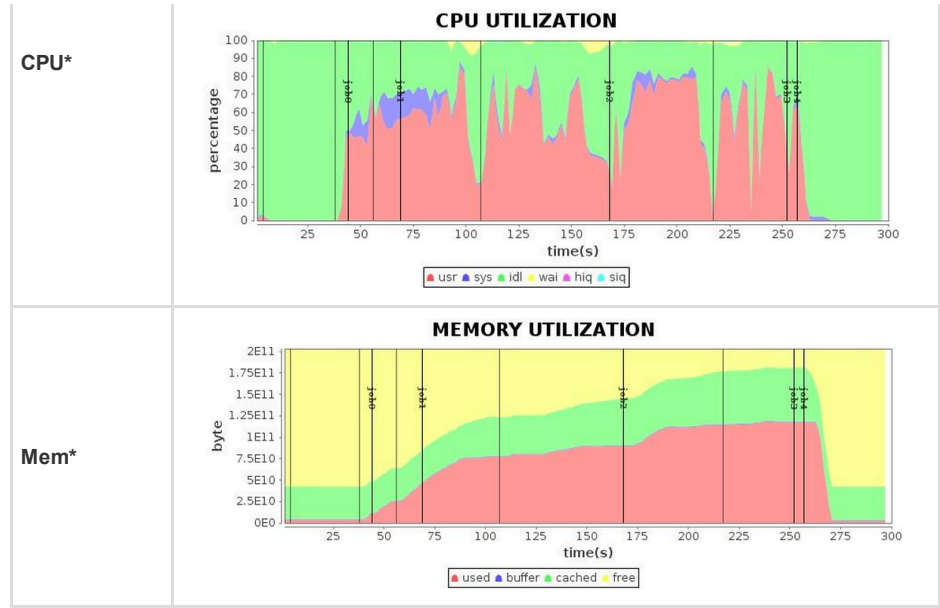


表 5 使用G1 GC调优完成后的表现

小结：综合考虑G1 GC是较为推崇的默认Spark GC机制。进一步的GC日志分析，可以收获更多的GC优化。经过上面的调优过程，我们将该应用的运行时间缩短到了4.3分钟，相比调优之前，我们获得了1.7倍左右的性能提升，而相比Parallel GC也获得了1.5倍左右的性能提升。

总结

对于大量依赖于内存计算的Spark应用，GC调优显得尤为重要。在发现GC问题的时候，不要着急调试GC。而是先考虑是否存在Spark进程内存管理的效率问题，例如RDD缓存的持久化和释放。至于GC参数的调试，首先我们比较推荐使用G1 GC来运行Spark应用。相较于传统的垃圾收集器，随着G1的不断成熟，需要配置的选项会更少，能同时满足高吞吐量和低延迟的寻求。当然，GC的调优不是绝对的，不同的应用会有不同应用的特性，掌握根据GC日志进行调优的方法，才能以不变应万变。最后，也不能忘了先对程序本身的逻辑和代码编写进行考量，例如减少中间变量的创建或者复制，控制大对象的创建，将长期存活对象放在Off-heap中等等。

6月3-5日，北京国家会议中心，[第七届中国云计算大会](#)，3天主会，17场分论坛，3场实战培训，160+位讲师，[议题全公开](#)！

顶
11

踩
0


网页设计


自己建网站


免费ftp服务器


免费云服务器


免流服务器


云服务器

推荐阅读相关主题：应用 云计算大会 内存管理 应用程序 流式计算 批处理

相关文章 | 最新报道

基于PredictionIO的推荐引擎打造，及大规模多标签...

CausalImpact，谷歌开源的R时域因果关系分析工具

与Hadoop之间的PK Spark胜算几何？

GC调优在Spark应用中的实践

IBM陈冠诚：如何使用OpenStack、Docker和Spark...

盘点Hadoop生态圈：13个让大象飞起来的开源工具

已有7条评论

还可以再输入500个字




有什么感想，你也来说说吧！

yxydde 欢迎您！

发表评论

最新评论

最热评论

 sparkjvm 2015-06-04 15:22

-XX:ConcGCThreads=n Number of threads concurrent garbage collectors will use. The default value varies with the platform on which the JVM is running.少了个s了调优参数

1票 回复

 weian404 2015-06-04 11:00

上边那任务图表用的是啥做的

回复

 快乐程序员 2015-06-03 22:30

GC1成熟能用了？

回复

 快乐程序员 2015-06-03 22:11

赞，就是太长了。

回复

 challengezhou 2015-06-03 15:04

先顶再看

回复

 keyantouru 2015-06-03 11:08

好东西必须顶。

回复

 yangfei8603 2015-06-02 22:52

说的不错 好

回复

请您注意

- 自觉遵守：爱国、守法、自律、真实、文明的原则
- 尊重网上道德，遵守《全国人大常委会关于维护互联网安全的决定》及中华人民共和国其他各项有关法律法规
- 严禁发表危害国家安全，破坏民族团结、国家宗教政策和社会稳定，含侮辱、诽谤、教唆、淫秽等内容的作品
- 承担一切因您的行为而直接或间接导致的民事或刑事责任
- 您在CSDN新闻评论发表的作品，CSDN有权在网站内保留、转载、引用或者删除
- 参与本评论即表明您已经阅读并接受上述条款



尚学堂
www.bjsxxt.com
400-009-1906

java android开发培训 赠送价值1800元基础班 月薪10000+



马上报名>>>

