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。

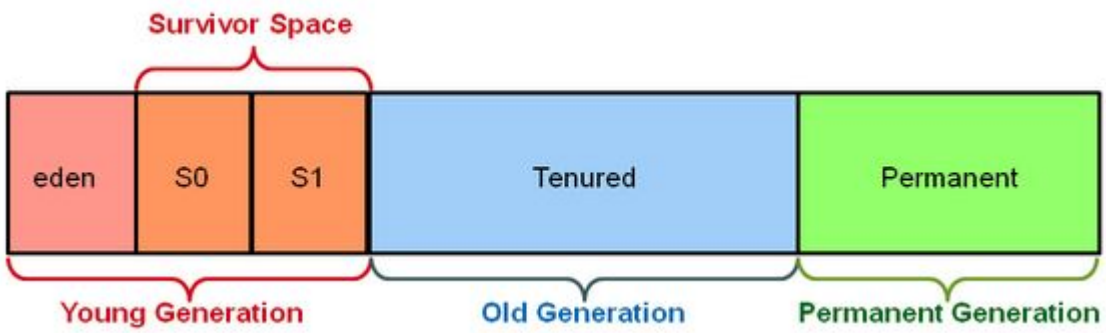


图 1 分年代的 Heap 结构

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域块（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。

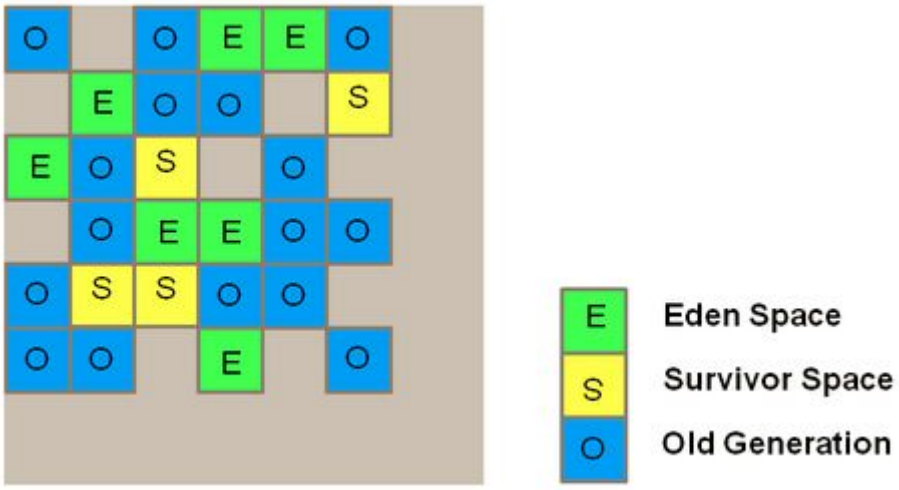


图 2 G1 Heap 结构示意

由于 G1 取消了对于 heap 空间不同新旧对象固定分区的概念，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将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 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%。

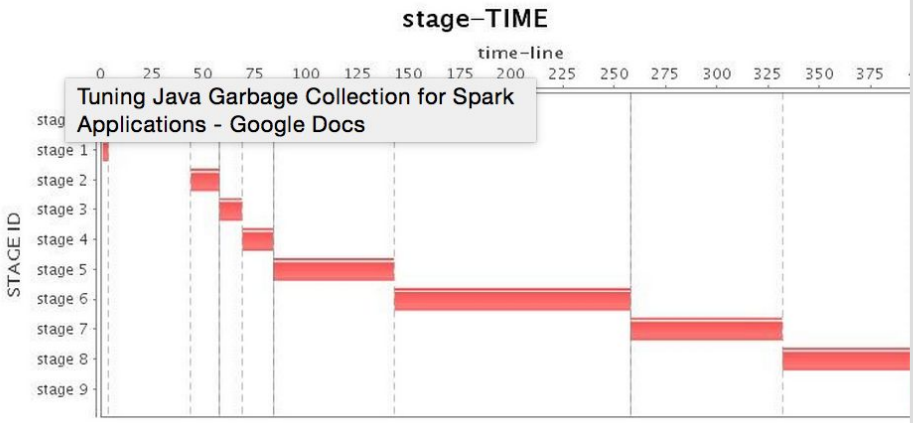
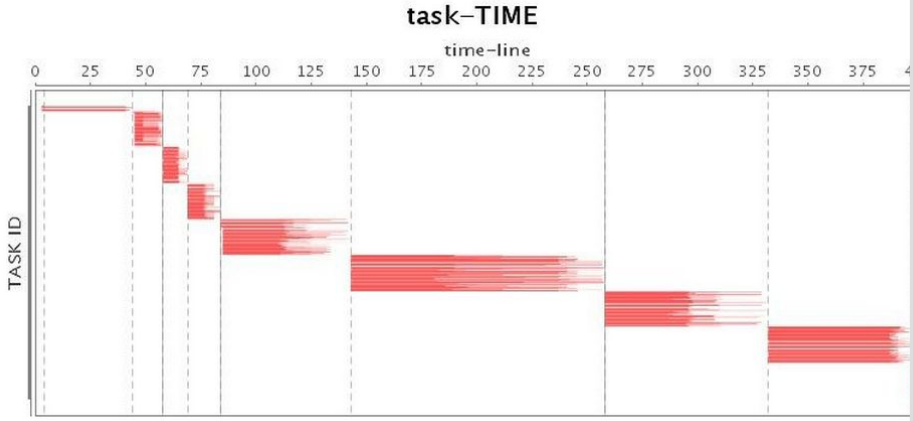
迭代轮数	单次迭代缓存大小	总缓存大小（优化前）	总缓存大小（优化后）
初始化	4.3GB	4.3GB	4.3GB
1	8.2GB	12.5GB	8.2GB
2	98.8GB	111.3GB	98.8GB
3	90.8GB	202.1GB	90.8GB

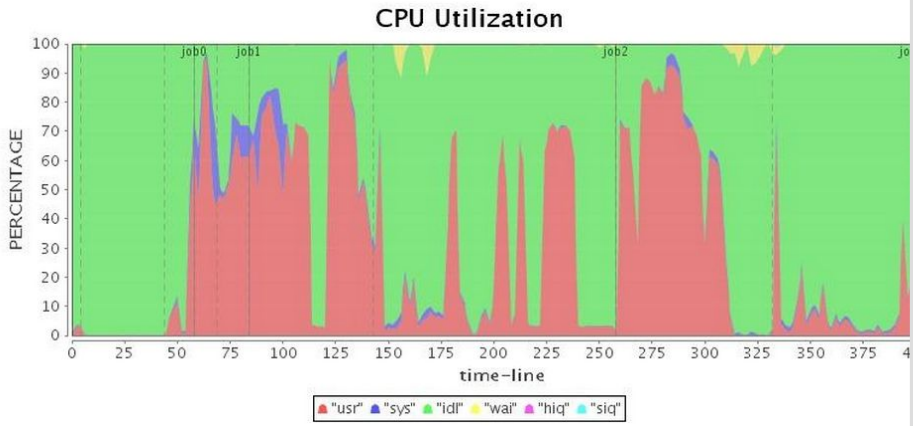
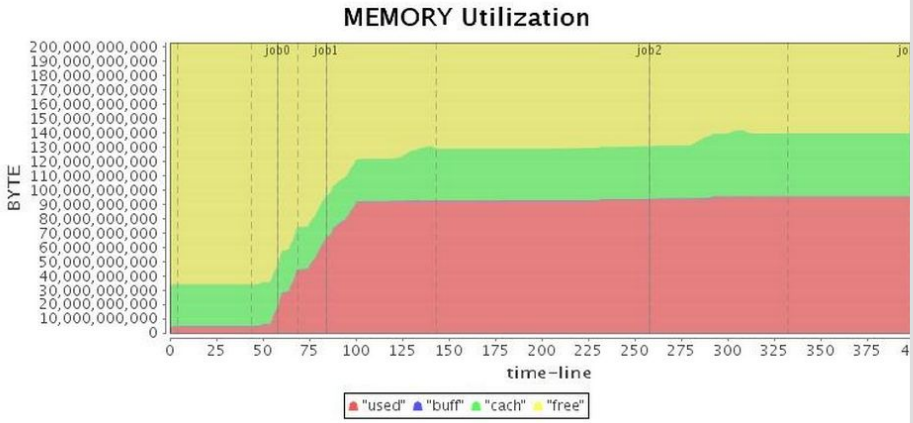
小结：当观察到 GC 频繁或者延时长的情况，也可能是 Spark 进程或者应用中内存空间没有有效利用。所以可以尝试检查是否存在 RDD 持久化后未得到及时释放等情况。

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*	
Task*	

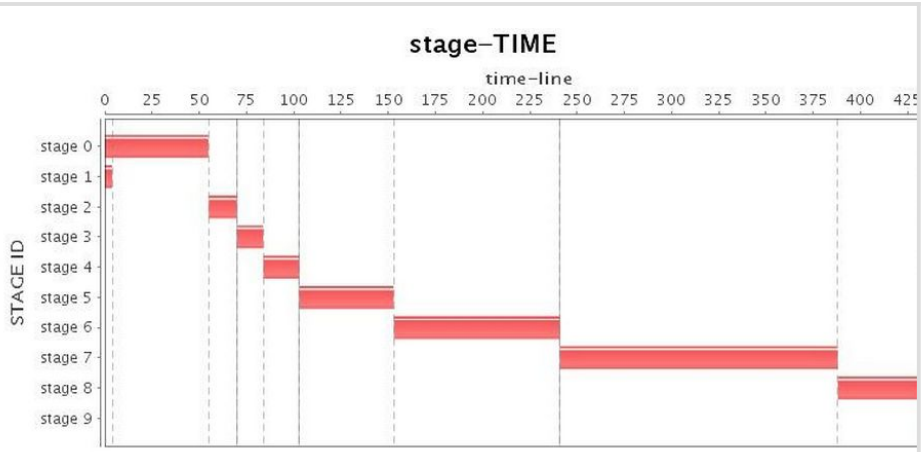
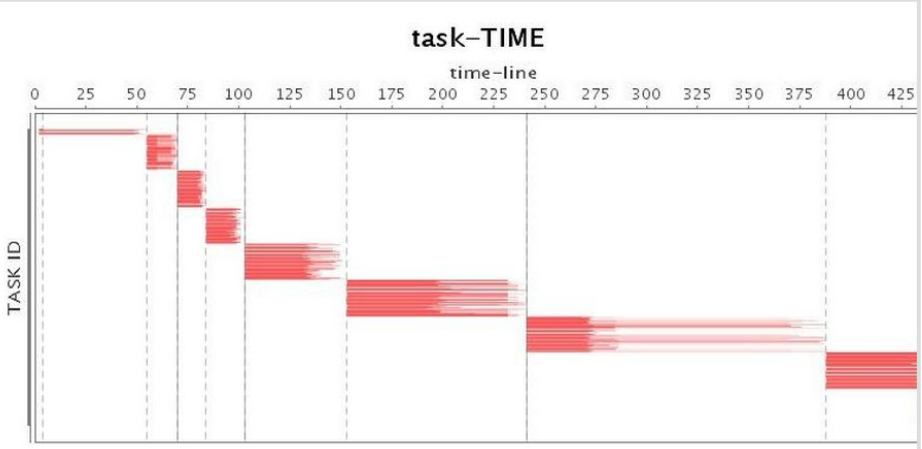
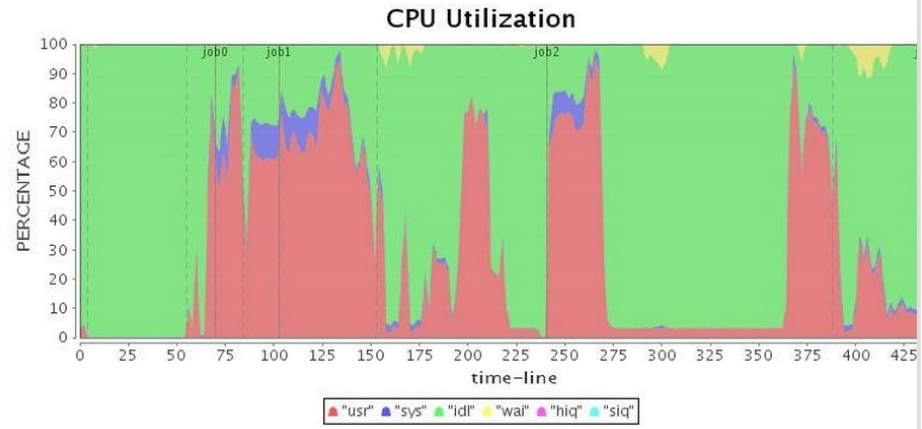
CPU*	
Mem*	

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。

Configuration Options	-XX:+UseG1GC -XX:+PrintFlagsFinal -XX:+PrintReferenceGC -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintAdaptiveSizePolicy -XX:+UnlockDiagnosticVMOptions -XX:+G1SummarizeConcMark -Xms88g -Xmx88g

Stage*	
Task*	
CPU*	

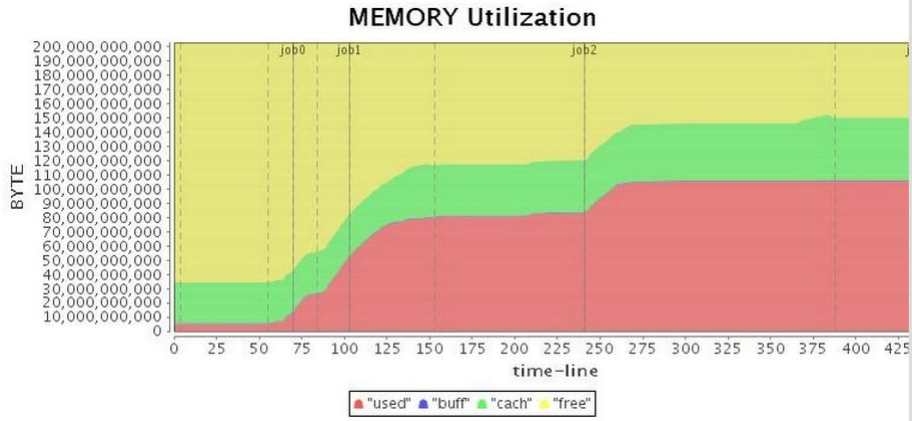
Mem*	

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，我们可以分析一下单次 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

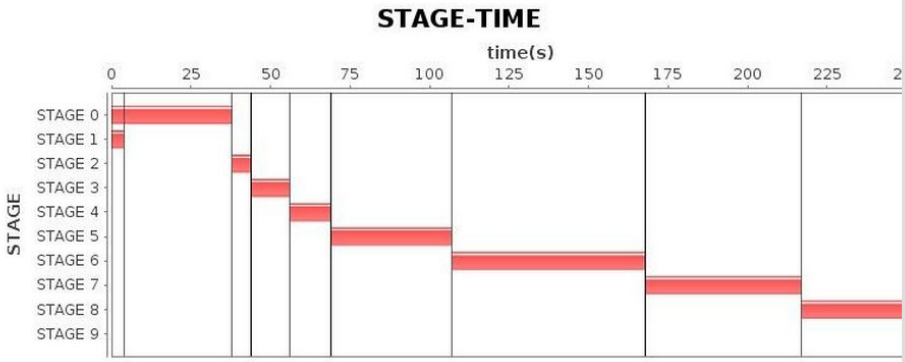
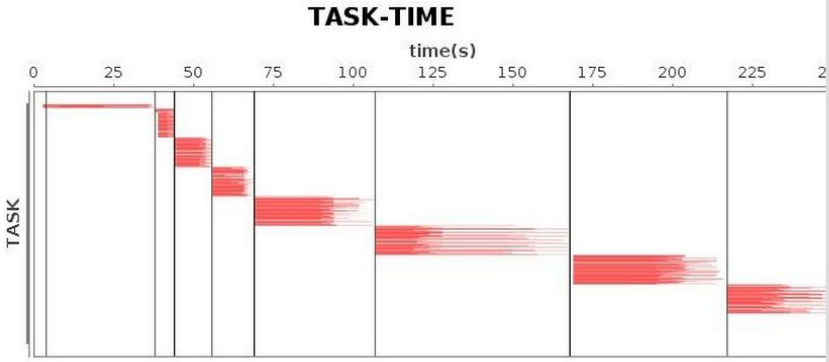
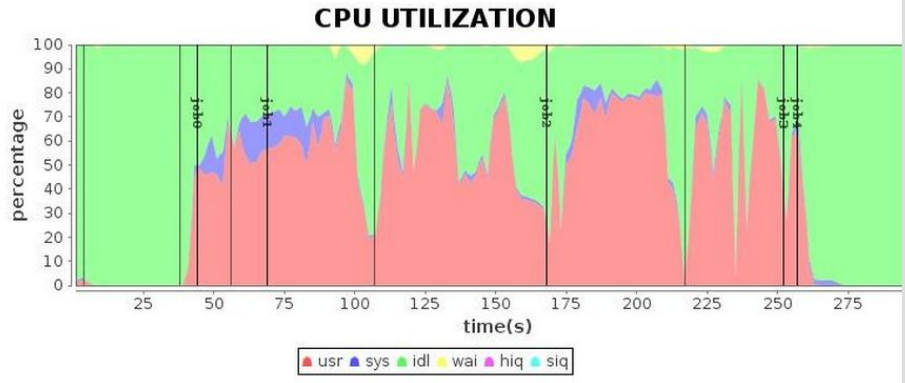
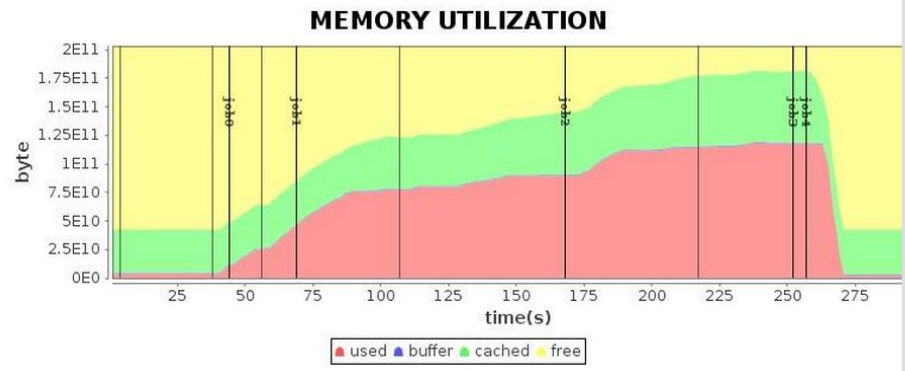
Stage*	
Task*	
CPU*	
Mem*	

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。