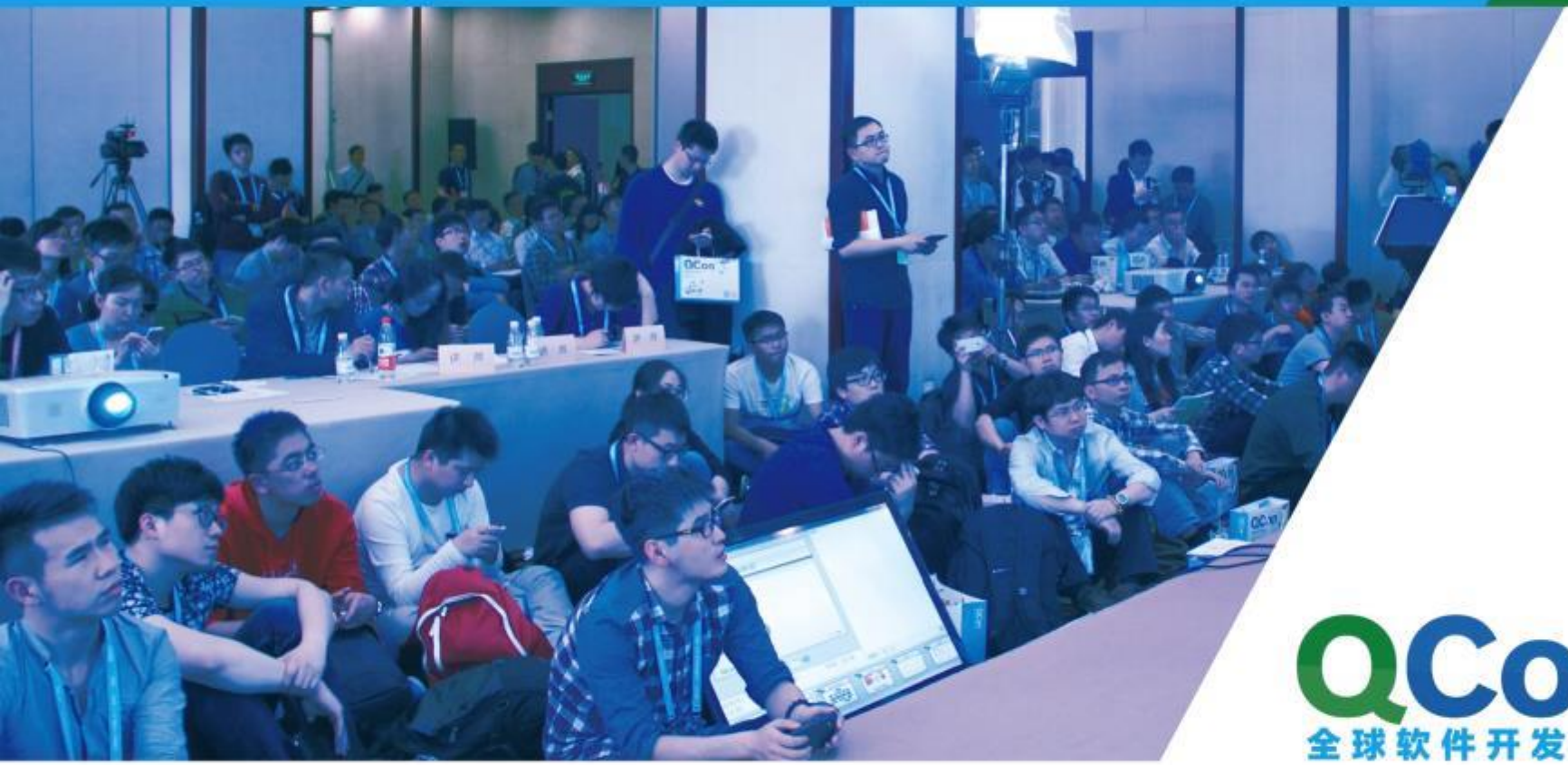
QCon全球软件开发大会

International Software Development Conference



QCon
全球软件开发大会

Spark应用GC调优

王道远 @ Intel

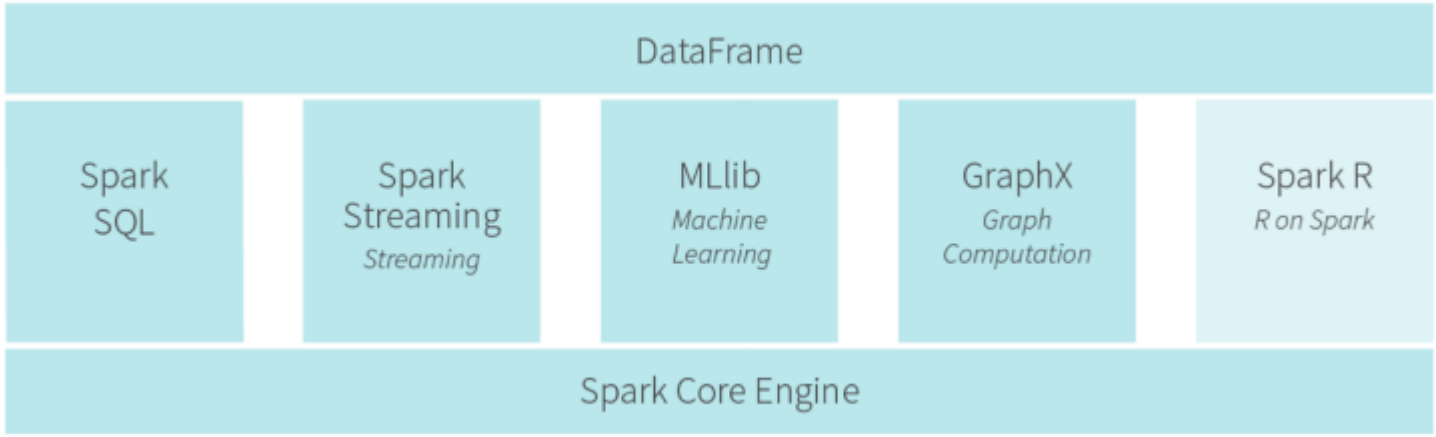
AGENDA

- 动机
- Spark内存模型
- GC算法比较
- GC调优实践
- 总结

动机

- Spark是一个开源的大数据通用计算框架，应用领域十分广泛
- Spark运行在JVM上
- Spark将许多数据放在内存中，需要管理比较大的heap空间
- Spark作业运行时间较长，过程中常常观测到很多GC pause

Spark


DataFrame
Spark SQL
Spark Streaming
Streaming
MLlib
Machine Learning
GraphX
Graph Computation
Spark R
R on Spark
Spark Core Engine

Spark内存模型

- RDD 弹性分布式数据集
  - 放在内存中，有助于迭代计算
  - Lazy计算，充分优化DAG
- Partitions
  - RDD按partition存放
- Shuffle
  - RDD有时需要通过网络进行混洗

内存中的对象都是些什么

- RDD缓存
  - 一块专用区域
  - spark.storage.memoryFraction
- 计算中RDD的时候使用的内存
  - 主要的内存消耗
- Shuffle过程中用到的Serializer/Deserializer
  - 很影响性能

首先检查Spark应用本身

- 是不是cache了没有必要cache的RDD
- 迭代计算，及时清除cache

iteration	iteration	Total cache size	Total cache size
0	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

GC基本概念 - Tracing

- GC Root Tracing
- 大多数支持GC的主流语言使用，包括Java/C#/Lisp
- GC Roots -> reference chain -> unreachable
- GC Roots: references in stack/JNI/method

GC基本概念 - Sweep

- 从堆中移除
- 导致内存碎片化
- 大对象回收可能会导致full GC

GC基本概念 - Copy

- 基本概念: (From space) -> (To space)
- 把内存分为相等大小的两块，每次只使用一块。
- 将存活对象拷贝到另一块后，使用新的这块。

GC基本概念 – Compact

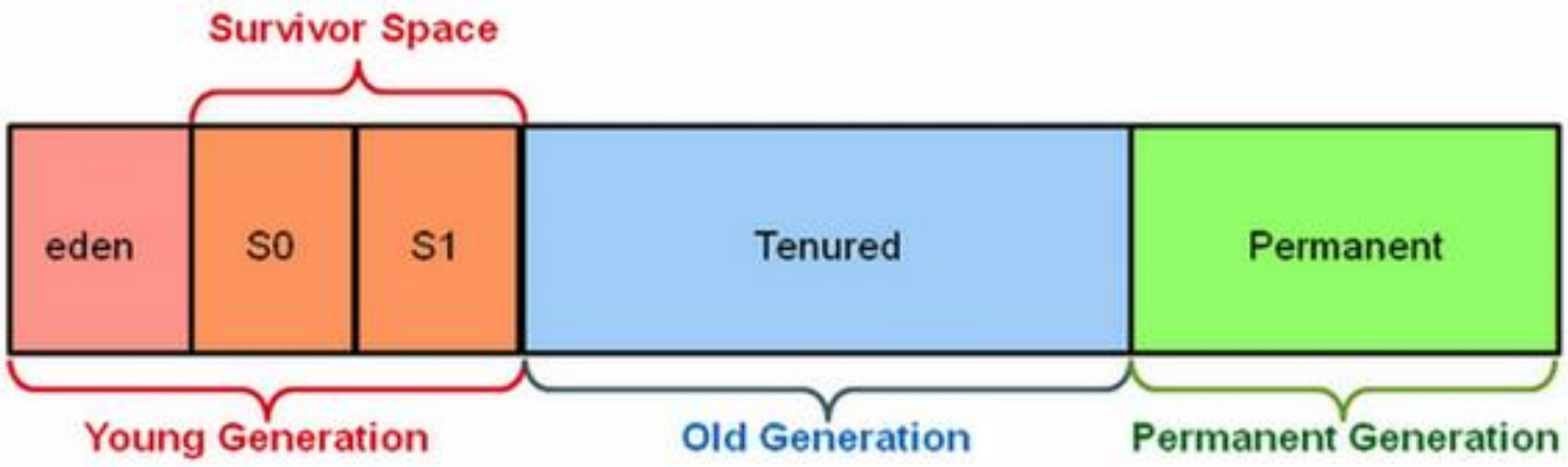
- 将存活对象移动到一起，清理剩下的空间
- 这样减少了内存的浪费.
- 通常使用一些外部的数据结构来记录对象存活情况

GC基本概念 - Generation

- 大多数对象生命周期是很短暂的.
- Young/Old
- 对于生命周期较短的对象应当更加频繁地进行GC
- Use copy in young because a lot of them has been dead, and use Mark-Sweep or Mark-Compact in old gen

GC基本概念 – Generation**

- Hotspot JVM采用的实现



GC收集器比较

- Incremental (已放弃)
- Serial
- Parallel
- Concurrent-Mark-Sweep (CMS)
- G1

GC收集器 - Parallel

- Only In Young
- Copy, multi-thread
- STW
- 1.6以后默认打开，以前使用的是SerialOld
- -XX:+UseParallelGC
- Throughput最大化

GC收集器 - ParallelOld

- Only work in old space, 与Parallel共同工作
- -XX:+UseParallelOldGC
- 最大化Throughput
- Start from Java 1.6.
- STW

GC收集器 - ParNew

- Only work in young generation
- 和Parallel相似，但是和CMS共同工作
- Throughput略低于Parallel，暂停时间更短

GC收集器 - CMS

- Only work in old generation
- 目标是实现更短的暂停，而不是更高的throughput
- -XX:+UseConcMarkSweepGC

GC收集器 - CMS

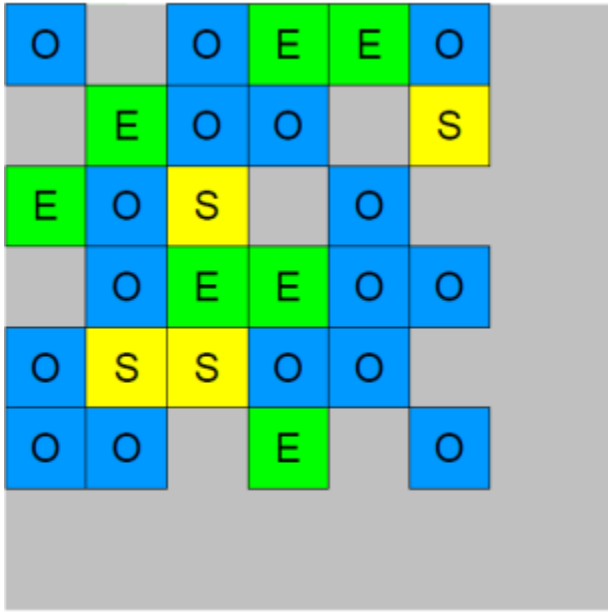
- Initial mark(STW, single-thread)
- Concurrent mark
- Concurrent pre-clean[part of remark work]
- Remark(STW, multi-thread)
  - revisiting objects modified during marking
- Concurrent sweep
- Concurrent reset

CMS GC缺点

- 碎片很多
  - 需要更多的heap空间
- 分配内存的代价比较大
  - 没有连续的空闲空间
  - 必须维护空闲空间列表
  - 在从young空间升级时，需要动态分配出合适的空间
- 升级失败时采用类似ParallelOld的策略

GC收集器 – G1**

- 把堆空间分为许多固定大小的区域，动态分配各年代的空间大小 (Eden, survivor, old gen, humongous, unused)



GC收集器 – G1

- CSet
  - 需要回收的对象
- RSet
  - 从别的区域进入当前区域的引用，用来隔离各个区域
- 每个区域维护Rset和Cset各一个

GC收集器 – G1

- 永远优先收集垃圾最多的区域
- 通过并发循环标记
- G1 Young GC/G1 Mixed GC

GC收集器 – G1

- Initial Mark(STW)
  - mark roots
- Root Region Scan
  - Must complete before the next young GC can happen
- Concurrent mark
  - can be interrupted by young GCs

GC收集器 – G1

- Remark(STW)
  - Completes marking (SnapshotAtTheBeginning[SATB] buffer (other GC use dirty card algorithm))
  - Reference processing
- Cleanup (partial STW, namely mixed GC)
  - Liveness accounting and identify completely free regions (STW)
  - Remembered Set scrubbing (STW)
  - Reset the empty regions and return them to the free list (Concurrent)

GC收集器 – G1

- Copying (Young GCs and Mixed GCs)
  - STW
  - If it's a young GC, copy all live objects in young regions into new regions (young, old)
  - If it's a mixed GC, copy all live objects in young regions into new regions (young, old) and copy some old regions into new regions (old)

JVM GC Compatibility

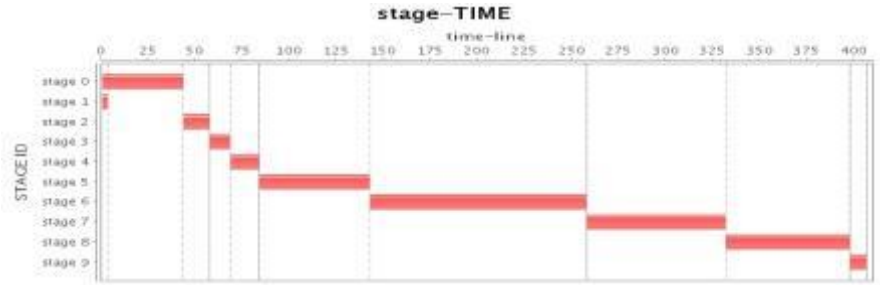
GC Collector	Serial	Parallel	ParallelOld	ParNew	ConcMark Sweep	G1
Serial		×	×	×	×	×
Parallel	×		✓	×	×	×
ParallelOld	×	✓		×	×	×
ParNew	×	×	×		✓	×
ConcMark Sweep	×	×	×	✓		×
G1	×	×	×	×	×	

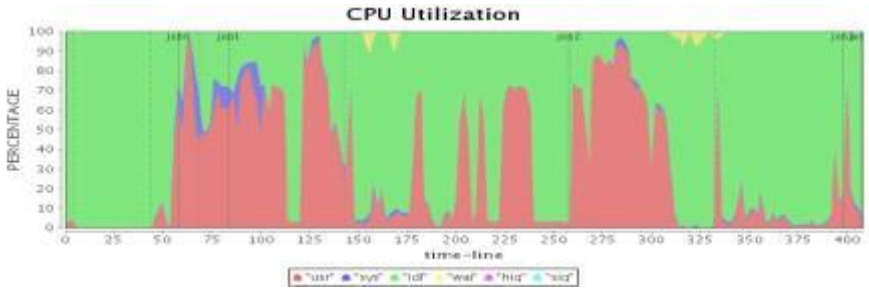
GC方案分析

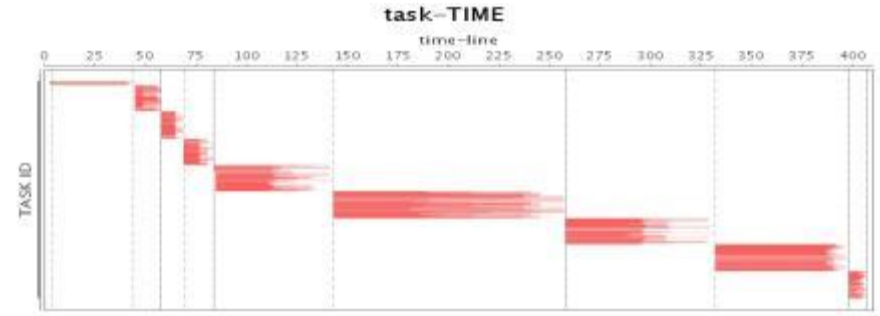
- 我们在没有进行调优的情况下，比较了三种GC收集器的表现

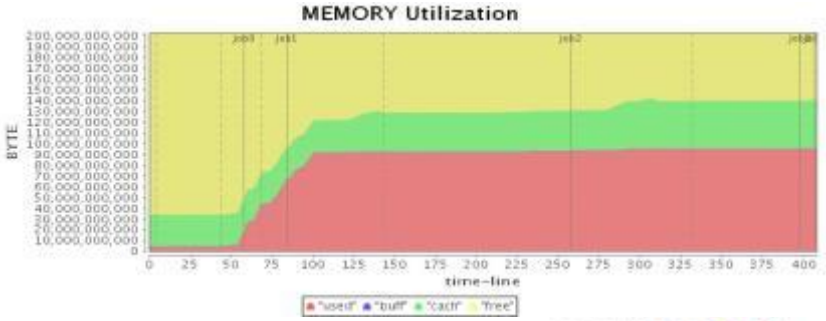
Garbage Collector	Run time
Parallel GC	6.5min
CMS GC	9min
G1 GC	7.6min

GC方案分析 – Parallel*





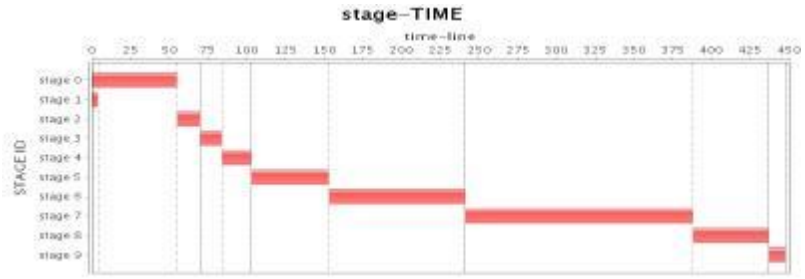


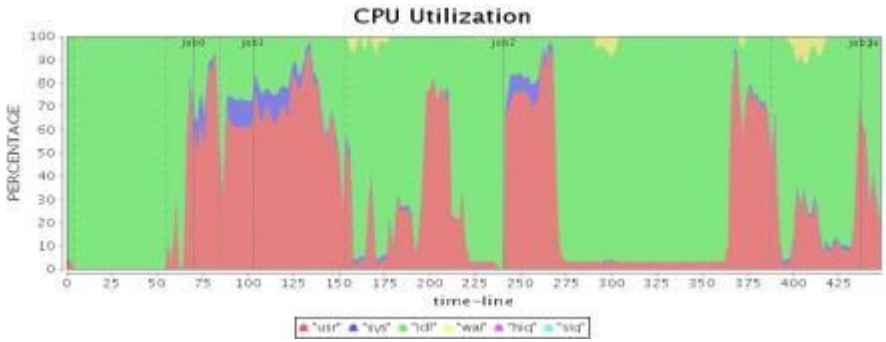


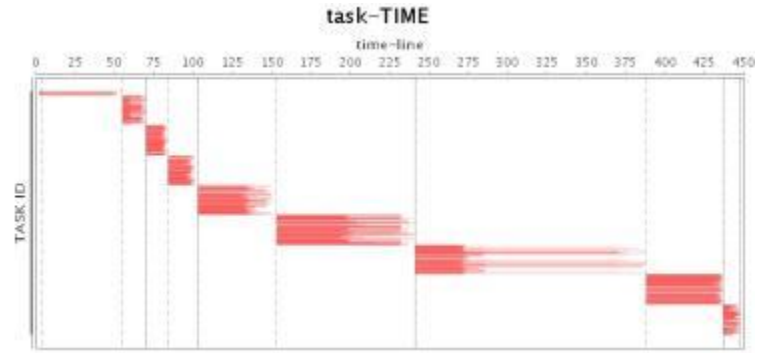
GC方案分析 - Parallel

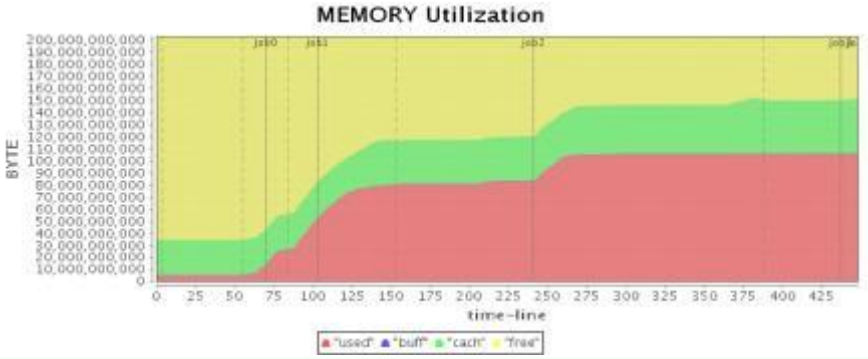
- -XX:+UseParallelGC -XX:+UseParallelOldGC -XX:+PrintFlagsFinal -XX:+PrintReferenceGC -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintAdaptiveSizePolicy -Xms88g -Xmx88g

GC方案分析 – CMS*


stage–TIME
time–line
STAGE ID


CPU Utilization
PERCENTAGE
time–line


task–TIME
time–line
TASK ID


MEMORY Utilization
BYTE
time–line

GC方案分析 - CMS

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

GC方案分析 – GC调优的考量

- Footprint
  - Application(soft/weak/phantom refs, ….)
- Throughput
- Latency
  - 把暂停时的工作尽量往并行阶段转移

GC方案分析

- 如果不在意延时，可以先用Parallel GC试试，如果没有发生full GC，可以使用Parallel GC，否则使用.
- 即使在意延时，不妨也先用Parallel GC试试，延时不能达到要求再尝试别的方案
- 关掉PLAB “-XX:-ResizePLAB”
- 如果要使用G1，确保使用最新的JVM

GC调优-收集GC日志

-XX:+PrintFlagsFinal -XX:+PrintReferenceGC

-verbose:gc -XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-XX:+PrintAdaptiveSizePolicy

-XX:+UnlockDiagnosticVMOptions

-XX:+G1SummarizeConcMark

G1 GC调优

- G1调优的两大目标
- 1. 避免 full GC
- 2. 减少暂停时间
- 2.1 减少总的暂停时间
- 2.2 减少最大暂停时间

从Parallel迁移到G1

- Don’t use –Xmn, -XX:-UseAdaptiveSizePolicy, -XX:SurvivorRatio=n
- 保留其他配置

Evacuation Failure

- To-space exhaused/overflow
- 一般都会导致一次Full GC

Evacuation Failure

- Decrease InitiatingHeapOccupancyPercent, -XX:InitiatingHeapOccupancyPercent=nn(default to 45)
- Increase ConcGCThreads, -XX:ConcGCThread=nn (default to ¼ of ParallelGCThreads)
- 牺牲程序运行时的CPU时间，换取更少出现to-space overflow

Humongous object

- 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]
- humongous allocation 是指单个对象超过G1 region size的50%
- Increase that by –XX:G1HeapRegionSize (?)

从CMS迁移到G1

- 移除–Xmn –XX:InitialSurvivorRatio –XX:SurvivorRatio –XX:InitialTenuringThreshold –XX:MaxTenuringThreshold –XX:ParallelGCThreads –XX:ConcGCThreads

较长的RSet Update时间

- [Update RS (ms): Min: 1.2, Avg: 1.9, Max: 3.3, Diff: 2.1, Sum: 43.4]
- -XX:G1RSetUpdatingPauseTimePercent=10
- Higher -> 把更多工作放到STW阶段完成
- Lower -> 把更多工作放到G1 Concurrent Refinement Threads中完成
- -XX: G1ConcRefinementThreads(default to ParallelGCThreads)

其他潜在提升

- Cycle开始到mixed GC发生的间隔过长
  - Increase ConcGCThreads
  - 会提高GC的CPU占用率
- Marking Cycle过于频繁
  - Increase –XX:InitiatingHeapOccupancyPercent
  - 会降低GC的CPU占用率

并行引用处理

- -XX:+ParallelRefProcEnabled

```
   [Other: 25.5 ms]
      [Choose CSet: 0.0 ms]
      [Ref Proc: 5.7 ms]
      [Ref Enq: 0.5 ms]
      [Free CSet: 1.0 ms]
   [Eden: 3904.0M(3904.0M)->0.0B(3904.0M) Su
 Times: user=9.44 sys=6.55, real=0.73 secs]


644.403: [GC remark 644.404: [GC ref-proc644.
 [Times: user=0.90 sys=0.00, real=0.05 secs]
```

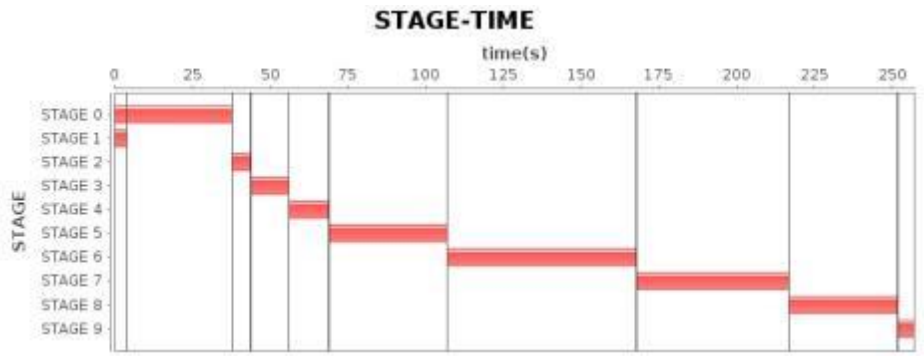
一次典型的GC暂停日志

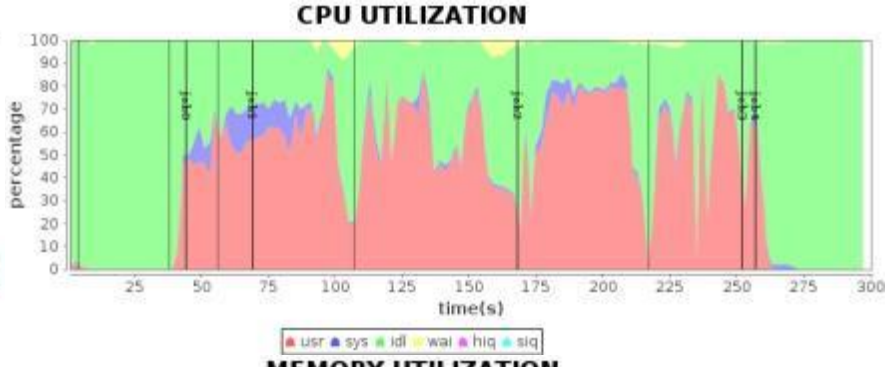
```
 251.354: [G1Ergonomics (Mixed GCs) continue mixed GCs, reason: candidate old regions available, candidate old regi
, 0.1785630 secs]
   [Parallel Time: 145.1 ms, GC Workers: 23]
      [GC Worker Start (ms): Min: 251176.0, Avg: 251176.4, Max: 251176.7, Diff: 0.7]
      [Ext Root Scanning (ms): Min: 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]
```

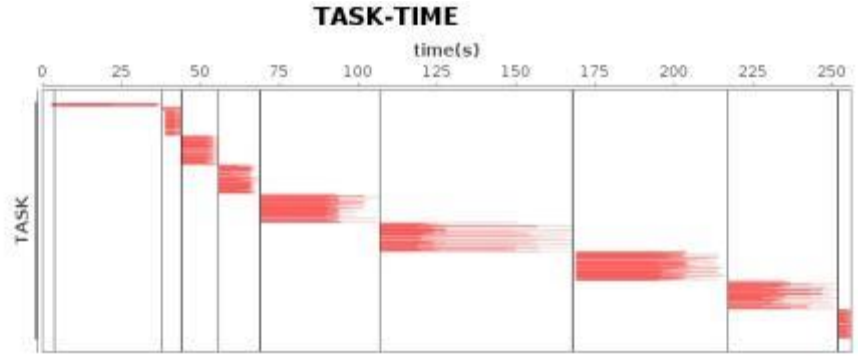
结果

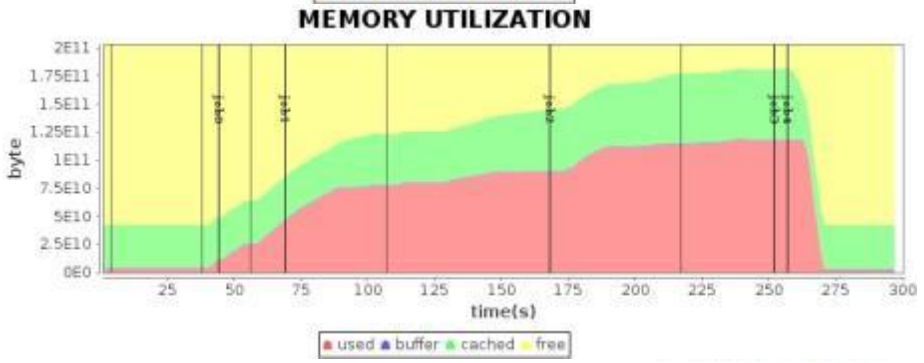
- -XX:+UseG1GC -XX:+PrintFlagsFinal -XX:+PrintReferenceGC -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintAdaptiveSizePolicy -XX:+UnlockDiagnosticVMOptions -XX:+G1SummarizeConcMark -Xms88g -Xmx88g
- 通过调优，总运行时间为4.3min

结果*


STAGE-TIME
time(s)
0 25 50 75 100 125 150 175 200 225 250
STAGE
STAGE 0
STAGE 1
STAGE 2
STAGE 3
STAGE 4
STAGE 5
STAGE 6
STAGE 7
STAGE 8
STAGE 9


CPU UTILIZATION
percentage
100 90 80 70 60 50 40 30 20 10 0
25 50 75 100 125 150 175 200 225 250 275 300
time(s)
usr sys idl wai hiq siq


TASK-TIME
time(s)
0 25 50 75 100 125 150 175 200 225 250
TASK


MEMORY UTILIZATION
byte
2E11
1.75E11
1.5E11
1.25E11
1E11
7.5E10
5E10
2.5E10
0E0
25 50 75 100 125 150 175 200 225 250 275 300
time(s)
used buffer cached free

小结

- 对于不追求响应时间的应用，不妨先尝试Parallel GC.
- 如果遇到Full GC，可以使用“-XX:+UseG1GC –XX:-ResizePLAB”
- 如果还是有Full GC，再进行进一步调优☺

还可以做得更好☺

- GC调优没有万能的策略
- GC调优会越来越容易
- Spark为内存管理也在不断进行优化
  - Tachyon, Project Tungsten
- 可以尝试使用更多的executor

总结

- 首先对应用本身进行优化
- G1是一个很有潜力的垃圾回收器，对于大heap管理，我们在Spark应用中使用G1得到了令人满意的结果
- Spark新的堆外存储等技术也是非常值得关注

Acknowledgement

- 在调优过程中，我们得到了Intel Java Runtime团队资深专家Yanping Wang的指导和帮助。

- *** 表示图片来自Intel大数据团队的内部Perf工具.**
- **** 标示图片来自Oracle官方文档。详情参见参考 [2] [3]**

参考

- [1] https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html#wp1086917
- [2] http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html
- [3]http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/G1GettingStarted/index.html
- [4] http://www.infoq.com/articles/tuning-tips-G1-GC
- [5] https://blogs.oracle.com/poonam/entry/understanding_g1_gc_logs

THANKS