Yarn日志聚合

<property>

<name>yarn.log-aggregation-enable</name>

<value>true</value>

</property>

<property>

<name>yarn.log.server.url</name>

<value>http://mu4:19888/jobhistory/job</value>

</property>

<property>

<name>yarn.nodemanager.log-aggregation.roll-monitoring-interval-seconds</name>

<value>3600</value>

</property>

<property>

<name>yarn.nodemanager.remote-app-log-dir</name>

<value>/tmp/logs</value>

</property>

<property>

    <name>yarn.app.mapreduce.am.staging-dir</name>

    <value>/tmp/hadoop-yarn/staging</value>

    <description>这个配置项是放在yarn-site.xml里面的。该参数指定的目录在hdfs上，主要用来存放application的说明信息，包括依赖的jar，输入数据的划分信息以及作业的执行计划等</descrption>

</property>

<property>

    <name>mapreduce.jobhistory.address</name>

    <value>0.0.0.0:10020</value>

</property>

<property>

    <name>mapreduce.jobhistory.webapp.address</name>

    <value>0.0.0.0:19888</value>

</property>

<property>

    <name>mapreduce.jobhistory.done-dir</name>

    <value>${yarn.app.mapreduce.am.staging-dir}/history/done</value>

</property>

<property>

    <name>mapreduce.jobhistory.intermediate-done-dir</name>

    <value>${yarn.app.mapreduce.am.staging-dir}

                         /history/done\_intermediate</value>

</property>

修改spark-defaults.conf

spark.eventLog.enabled=true

spark.eventLog.dir=hdfs://cluster/tmp/spark\_historyserver\_event

spark.yarn.historyServer.address=http://mu4:18080

# spark.eventLog.dir hdfs://namenode:8021/directory

# spark.serializer org.apache.spark.serializer.KryoSerializer

# spark.driver.memory 5g

# spark.executor.extraJavaOptions -XX:+PrintGCDetails -Dkey=value -Dnumbers="one two three"

spark.yarn.jars=hdfs://cluster/lib/spark/\*

spark.hadoop.yarn.timeline-service.enabled=false

1. MR的job History服务启动(不需要启动)  
　　　　mr-jobhistory-daemon.sh start historyserver  
　　　　http://hadoop-senior01:19888/jobhistory  
　　2. Spark的Job History服务启动  
　　　　sbin/start-history-server.sh hdfs://cluster/tmp/spark\_historyserver\_event

Unlike Spark standalone and Mesos modes, in which the master’s address is specified in the --master parameter,  
in YARN mode the ResourceManager’s address is picked up from the Hadoop configuration. Thus,  
the --master parameter is yarn.

提交任务命令:

$ ./bin/spark-submit --class org.apache.spark.examples.SparkPi \

--master yarn \

--deploy-mode cluster \

--driver-memory 4g \

--executor-memory 2g \

--executor-cores 1 \

--queue thequeue \

lib/spark-examples\*.jar \

--conf spark.default.parallelism=300

--jars a.jar,b.jar

10

nohup spark-submit --master spark://hxf:7077

--executor-memory 30G 每一个

--executor-cores 2 每一个executor的核数、yarn有效、默认1

--num-executors 12

--conf spark.default.parallelism=300

--class com.analysis.main.GenAppInfo

/home/hadoop/jar/analysis.jar $dst\_date > /home/hadoop/logs/genAppInfo.log &

**–total-executor-cores NUM**：设置任务占用的总CPU核数（即任务的并发量），由主节点指定各个工作节点CPU的使用数。   
**注意：**该参数选项只在Spark standalone and Mesos 模式下有效

yarn logs -applicationId application\_1511334364226\_0440

yarn application -kill appid//appid

优雅的停止sparkstreaming

Sys.addshutdownhook{

Stringcontext.stop(boolean stopsparkcontext,boolean stopgraceful)

}

或者配置spark.streaming.stopGracefullyOnShutdown=true

通过 yarn application -kill appid或者kill -15杀死（正常退出，执行钩子）

Spark master:master挂了不能提交作业，不影响正在运行的作业

slave

在基于standalone的Spark集群，Cluster Manger就是Master。   
Master负责分配资源，在集群启动时，Driver向Master申请资源，Worker负责监控自己节点的内存和CPU等状况，并向Master汇报。

从资源方面，可以分为两个层面：   
**1）资源的管理和分配**   
资源的管理和分配，由Master和Worker来完成。Master给Worker分配资源，   
Master时刻知道Worker的资源状况。   
客户端向服务器提交作业，实际是提交给Master。

**2）资源的使用**   
资源的使用，由Driver和Executor。程序运行时候，向Master请求资源。

**参数:**

**优化:**

**batchDuration** : streamingcontext初始化时指定，过小会频繁提交作业，对整个streaming增加额外负担，并且计算不完的话会造成数据堆积

**spark.streaming.kafka.maxRatePerPartition ：kafka拉去量，默认无上限。应该结合batchDuration设置合理的数据量。**

**合理Gc参数:**--conf "spark.executor.extraJavaOptions=-XX:+UseConcMarkSweepGC"

缓存反复使用的RDD

spark.default.parallelizm：并行度问题，如果不设置这个参数，Spark 会跟据 HDFS 中 Block 的个数去设置这一个数量，原理是默应每个 Block 会对应一个 Task，默应情况下，如果数据量不是太多就不可以充份利用 executor 设置的资源，就会浪费了资源。建义设置为 100个，最好 700个左右。Spark官方的建义是每一个 Core 负责 2-3 个 Task。

spark.storage.memoryFraction：默应占用 60％，如果计算比较依赖于历史数据则可以调高该参数，当如果计算比较依赖 Shuffle 的话则需要降低该比例。

spark.shuffle.memoryFraction：默应占用 20％，如果计算比较依赖 Shuffle 的话则需要调高该比例。

spark.shuffle.safetyFraction=0.8

**Spark.shuffle.manager**=hash,sort,tungsten-sort

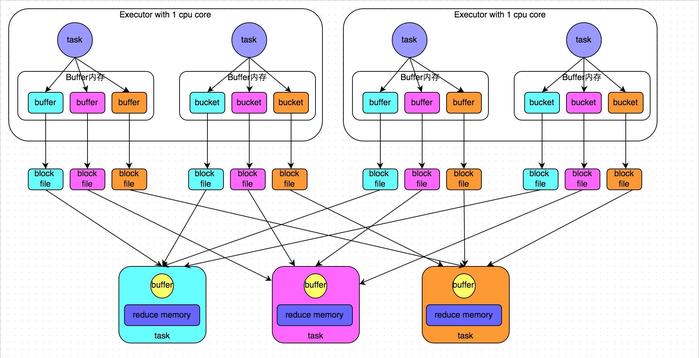
**HashShuffleManager**：shuffle write阶段，默认Mapper阶段会为Reducer阶段的每一个Task单独创建一个文件来保存该Task中要使用的数据。

优点：就是操作数据简单。

缺点：大量中间文件，大量io，容易OOM

优化: spark.shuffle.consolidateFiles=true，优化之后每个exucutor产生reduce task数量的文件

CPU core的数量 \* 下一个stage的task数量



**SortShuffleManager**

在Mapper中的每一个ShuffleMapTask中产生两个文件：Data文件和Index文件，产生任务数 2 \* shufflemaptask

Sorted-Based Shuffle 在 **Mapper 端是排序的**，包括 partition 的排序和每个 partition 内部元素的排序！但在 **Reducer 端是没有进行排序**，所以 Job 的结果默认不是排序的。Sorted-Based Shuffle 采用了 **Tim-Sort 排序算法**，好处是可以极为高效的使用 Mapper 端的排序成果全局排序Sorted-Based Shuffle 的核心是借助于 ExternalSorter 把每个 ShuffleMapTask 的输出排序到一个文件中(FileSegmentGroup)

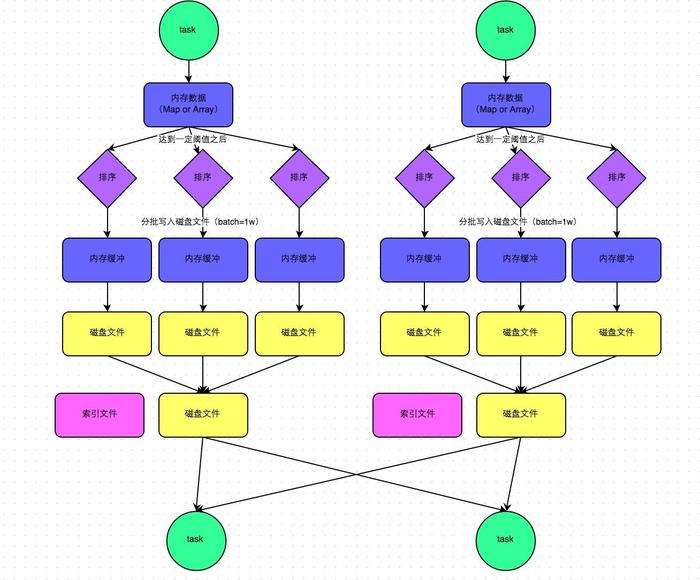
Bypass机制:触发条件1：shuffle map task数量小于spark.shuffle.sort.bypassMergeThreshold参数的值,默认200,此时会使用类似hashshuffle将分区文件写为单独文件，最后进行合并。这样通过去除Sort步骤来加快处理速度，代价是需要并发打开多个文件，所以内存消耗量增加。触发条件2：不是聚合类的shuffle算子（比如reduceByKey）。

优点:中间文件数量减少，节省io,内存

缺点:shufflemaptask如果数量很多，依旧会产生很多小文件

若需要分片内排序，需进行mapper端和reducer端两次排序，默认reduce端不排序。

不需要排序的操作也会执行map端排序，浪费计算资源



**Tungston-sort**：机制与sort类似，但是用到了对外内存，增加了对内存和cpu的优化

**Catalyst：是一种解析引擎，不仅是sql解析引擎。采用数据结构，使用解析规则解析为语法树。**

**Shuffle相关参数:**

[**spark.shuffle.file.buffer**](https://www.jianshu.com/p/98a1d67bc226)默认值：32k

参数说明：该参数用于设置shuffle write task的BufferedOutputStream的buffer缓冲大小。将数据写到磁盘文件之前，会先写入buffer缓冲中，待缓冲写满之后，才会溢写到磁盘。

调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如64k），从而减少shuffle write过程中溢写磁盘文件的次数，也就可以减少磁盘IO次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

**spark.reducer.maxSizeInFlight** 默认值：48m

参数说明：该参数用于设置shuffle read task的buffer缓冲大小，而这个buffer缓冲决定了每次能够拉取多少数据。

调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如96m），从而减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

**spark.shuffle.io.maxRetries** 默认值：3

参数说明：shuffle read task从shuffle write task所在节点拉取属于自己的数据时，如果因为网络异常导致拉取失败，是会自动进行重试的。该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败。

调优建议：对于那些包含了特别耗时的shuffle操作的作业，建议增加重试最大次数（比如60次），以避免由于JVM的full gc或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~上百亿）的shuffle过程，调节该参数可以大幅度提升稳定性。

[**spark.shuffle.io.retryWait**](https://www.jianshu.com/p/98a1d67bc226) 默认值：5s

参数说明：具体解释同上，该参数代表了每次重试拉取数据的等待间隔，默认是5s。

调优建议：建议加大间隔时长（比如60s），以增加shuffle操作的稳定性。

**spark.shuffle.consolidateFiles** 默认值：false

参数说明：如果使用HashShuffleManager，该参数有效。如果设置为true，那么就会开启consolidate机制，会大幅度合并shuffle write的输出文件，对于shuffle read task数量特别多的情况下，这种方法可以极大地减少磁盘IO开销，提升性能。

调优建议：如果的确不需要SortShuffleManager的排序机制，那么除了使用bypass机制，还可以尝试将spark.shffle.manager参数手动指定为hash，使用HashShuffleManager，同时开启consolidate机制。在实践中尝试过，发现其性能比开启了bypass机制的SortShuffleManager要高出10%~30%

**spark.ui.killEnabled:true**

运行在web UI中杀死stage和相应的job

**spark.ui.port:4040**

你的应用程序dashboard的端口。显示内存和工作量数据

**spark.rdd.compress:true**

是否压缩序列化的RDD分区。在花费一些额外的CPU时间的同时节省大量的空

**spark.broadcast.compress:true**

在发送广播变量之前是否压缩

**spark.io.compression.codec:snappy**

压缩诸如RDD分区、广播变量、shuffle输出等内部数据的编码解码器。默认情况下，Spark提供了三种选择：lz4、lzf和snappy，你也可以用完整的类名来制定

**spark.kryo.classesToRegister:(none)**

如果你用Kryo序列化，给定的用逗号分隔的自定义类名列表表示要注册的类

**spark.kryoserializer.buffer.max.mb:64**

Kryo序列化缓存允许的最大值。这个值必须大于你尝试序列化的对象

**spark.serializer:org.apache.spark.serializer.JavaSerializer**

序列化对象使用的类。默认的Java序列化类可以序列化任何可序列化的java对象但是它很慢。所有我们建议用org.apache.spark.serializer.KryoSerializer

**spark.cleaner.ttl:(infinite)**

spark记录任何元数据（stages生成、task生成等）的持续时间。定期清理可以确保将超期的元数据丢弃，这在运行长时间任务是很有用的，如运行7\*24的sparkstreaming任务。RDD持久化在内存中的超期数据也会被清理

**spark.default.parallelism**

本地模式：机器核数；Mesos：8；其他：max(executor的core，2)

如果用户不设置，系统使用集群中运行shuffle操作的默认任务数（groupByKey、 reduceByKey等）

**spark.files.fetchTimeout:60**

driver 程序获取通过SparkContext.addFile()添加的文件时的超时时间，单位秒

**spark.storage.unrollFraction:0.2**

Fraction of spark.storage.memoryFraction to use for unrolling blocks in memory.

**spark.tachyonStore.baseDir:System.getProperty(“java.io.tmpdir”)**

Tachyon File System临时目录

**spark.tachyonStore.url:tachyon://localhost:19998**

Tachyon File System URL

**spark.speculation.multiplier:1.5**

比已完成Task的运行速度中位数慢多少倍才启用推测

**spark.locality.wait:3000**

以下几个参数是关于Spark数据本地性的。本参数是以毫秒为单位启动本地数据task的等待时间，如果超出就启动下一本地优先级别 的task。该设置同样可以应用到各优先级别的本地性之间（本地进程 -> 本地节点 -> 本地机架 -> 任意节点 ），当然，也可以通过spark.locality.wait.node等参数设置不同优先级别的本地性

**当你运行Spark Standalone Mode或者Spark on Mesos模式时，你可以通过Spark History Server来查看job运行情况。**

**Kryo序列化**

1. val sparkConf = **new** SparkConf()
2. /\*.setMaster("local")\*/
3. .setAppName("Test")
4. .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
5. /\*.set("spark.kryo.registrationRequired", "true")\*/
6. .registerKryoClasses(Array(
7. classOf[Array[String]],
8. classOf[util.HashMap[String, String]],
9. classOf[util.ArrayList[String]],
10. classOf[MyTest]
11. ))

**自定义累加器：累加器要在action算子后才能取到值，因为是lazy计算模式。下列计算结果 accum1:9 accum2:18，因为两个action算子，map任务执行了两遍，对应的累加器也累加了两次，可通过缓存来避免。**

1. val accum = sc.longAccumulator("longAccum")
2. val numberRDD = sc.parallelize(Array(1,2,3,4,5,6,7,8,9),2).map(n=>{
3. accum.add(1L)
4. n+1
5. })
6. numberRDD.count
7. println("accum1:"+accum.value)
8. numberRDD.reduce(\_+\_)
9. println("accum2: "+accum.value)
10. //修改之后
11. numberRDD.cache().count
12. println("accum1:"+accum.value)
13. numberRDD.reduce(\_+\_)
14. println("accum2: "+accum.value)

**2.X之前implements AccumulatorParam<String>**

**2.X之后extends AccumulatorV2[String, String]**

### value方法：获取累加器中的值

### merge方法：该方法特别重要，一定要写对，这个方法是各个task的累加器进行合并的方法（下面介绍执行流程中将要用到）

### iszero方法：判断是否为初始值

### reset方法：重置累加器中的值

### copy方法：拷贝累加器

### spark中累加器的执行流程：

### 首先有几个task，spark engine就调用copy方法拷贝几个累加器（不注册的），然后在各个task中进行累加（注意在此过程中，被最初注册的累加器的值是不变的），执行最后将调用merge方法和各个task的结果累计器进行合并（此时被注册的累加器是初始值）

**Spark 2.1.0 新型 JVM Heap 分成三个部份：Reserved Memory、User Memory 和 Spark Memory。**

1. Reserved Memory：默认都是300MB
2. User Memory：写 Spark 程序中产生的临时数据或者是自己维护的一些数据结构也需要给予它一部份的存储空间，它占用的空间是 (Java Heap - Reserved Memory) x 25%(默认是25％，可以有参数供调优)
3. Spark Memeory：系统框架运行时需要使用的空间，这是从两部份构成的，分别是 Storage Memeory 和 Execution Memory。现在 Storage 和 Execution (Shuffle) 采用了 Unified 的方式共同使用了 (Heap Size - 300MB) x 75%，默认情况下 Storage 和 Execution 各占该空间的 50%。

Storage Memeory：相当于旧版本的 Storage 空间，在旧版本中 Storage 占了 54% 的 Heap 空间，这个空间会负责存储 Persist、Unroll 以及 Broadcast 的数据。

Execution Memeory：相当于旧版本的 Shuffle 空间，这个空间会负责存储 ShuffleMapTask 的数据。比如说从上一个 Stage 抓取数据和一些聚合的操作、等等。

**Spark数据倾斜**

#### 1、并行度分散同一个Task的不同Key

##### 原理

Spark在做Shuffle时，默认使用HashPartitioner（非Hash Shuffle）对数据进行分区。如果并行度设置的不合适，可能造成大量不相同的Key对应的数据被分配到了同一个Task上，造成该Task所处理的数据远大于其它Task，从而造成数据倾斜。

如果调整Shuffle时的并行度，使得原本被分配到同一Task的不同Key发配到不同Task上处理，则可降低原Task所需处理的数据量，从而缓解数据倾斜问题造成的短板效应。

适用场景  
大量不同的Key被分配到了相同的Task造成该Task数据量过大。

解决方案  
调整并行度。一般是增大并行度，但有时如本例减小并行度也可达到效果。

优势  
实现简单，可在需要Shuffle的操作算子上直接设置并行度或者使用spark.default.parallelism设置。如果是Spark SQL，还可通过SET spark.sql.shuffle.partitions=[num\_tasks]设置并行度。可用最小的代价解决问题。一般如果出现数据倾斜，都可以通过这种方法先试验几次，如果问题未解决，再尝试其它方法。

劣势  
适用场景少，只能将分配到同一Task的不同Key分散开，但对于同一Key倾斜严重的情况该方法并不适用。并且该方法一般只能缓解数据倾斜，没有彻底消除问题。从实践经验来看，其效果一般。

2、**自定义Partitioner**

原理

使用自定义的Partitioner（默认为HashPartitioner），将原本被分配到同一个Task的不同Key分配到不同Task。

总结

适用场景  
大量不同的Key被分配到了相同的Task造成该Task数据量过大。

解决方案  
使用自定义的Partitioner实现类代替默认的HashPartitioner，尽量将所有不同的Key均匀分配到不同的Task中。

优势  
不影响原有的并行度设计。如果改变并行度，后续Stage的并行度也会默认改变，可能会影响后续Stage。

劣势  
适用场景有限，只能将不同Key分散开，对于同一Key对应数据集非常大的场景不适用。效果与调整并行度类似，只能缓解数据倾斜而不能完全消除数据倾斜。而且需要根据数据特点自定义专用的Partitioner，不够灵活

**3、将Reduce side Join转变为Map side Join**

原理

通过Spark的Broadcast机制，将Reduce侧Join转化为Map侧Join，避免Shuffle从而完全消除Shuffle带来的数据倾斜。

正确的使用Broadcast实现Map侧Join的方式是，

通过SET spark.sql.autoBroadcastJoinThreshold=104857600;

将Broadcast的阈值设置得足够大。

SET spark.sql.autoBroadcastJoinThreshold=104857600;

适用场景  
参与Join的一边数据集足够小，可被加载进Driver并通过Broadcast方法广播到各个Executor中。

解决方案  
在Java/Scala代码中将小数据集数据拉取到Driver，然后通过broadcast方案将小数据集的数据广播到各Executor。或者在使用SQL前，将broadcast的阈值调整得足够多，从而使用broadcast生效。进而将Reduce侧Join替换为Map侧Join。

优势  
避免了Shuffle，彻底消除了数据倾斜产生的条件，可极大提升性能。

劣势  
要求参与Join的一侧数据集足够小，并且主要适用于Join的场景，不适合聚合的场景，适用条件有限。

**4、为skew的key增加随机前/后缀**

原理

为数据量特别大的Key增加随机前/后缀，使得原来Key相同的数据变为Key不相同的数据，从而使倾斜的数据集分散到不同的Task中，彻底解决数据倾斜问题。Join另一则的数据中，与倾斜Key对应的部分数据，与随机前缀集作笛卡尔乘积，从而保证无论数据倾斜侧倾斜Key如何加前缀，都能与之正常Join。

适用场景  
两张表都比较大，无法使用Map则Join。其中一个RDD有少数几个Key的数据量过大，另外一个RDD的Key分布较为均匀。

解决方案  
将有数据倾斜的RDD中倾斜Key对应的数据集单独抽取出来加上随机前缀，另外一个RDD每条数据分别与随机前缀结合形成新的RDD（相当于将其数据增到到原来的N倍，N即为随机前缀的总个数），然后将二者Join并去掉前缀。然后将不包含倾斜Key的剩余数据进行Join。最后将两次Join的结果集通过union合并，即可得到全部Join结果。

优势  
相对于Map侧Join，更能适应大数据集的Join。如果资源充足，倾斜部分数据集与非倾斜部分数据集可并行进行，效率提升明显。且只针对倾斜部分的数据做数据扩展，增加的资源消耗有限。

劣势  
如果倾斜Key非常多，则另一侧数据膨胀非常大，此方案不适用。而且此时对倾斜Key与非倾斜Key分开处理，需要扫描数据集两遍，增加了开销。

**5、大表随机添加N种随机前缀，小表扩大N倍**

原理

如果出现数据倾斜的Key比较多，上一种方法将这些大量的倾斜Key分拆出来，意义不大。此时更适合直接对存在数据倾斜的数据集全部加上随机前缀，然后对另外一个不存在严重数据倾斜的数据集整体与随机前缀集作笛卡尔乘积（即将数据量扩大N倍）。

适用场景  
一个数据集存在的倾斜Key比较多，另外一个数据集数据分布比较均匀。

优势  
对大部分场景都适用，效果不错。

劣势  
需要将一个数据集整体扩大N倍，会增加资源消耗。

总结

对于数据倾斜，并无一个统一的一劳永逸的方法。更多的时候，是结合数据特点（数据集大小，倾斜Key的多少等）

**Spark on yarn**

**1.** 由client向ResourceManager提交请求，并上传jar到HDFS上  
这期间包括四个步骤：  
**a). 连接到RM  
b). 从RM ASM（ApplicationsManager ）中获得metric、queue和resource等信息。  
c). upload app jar and spark-assembly jar  
d). 设置运行环境和container上下文（launch-container.sh等脚本)**

**2.**ResouceManager向NodeManager申请资源，创建Spark ApplicationMaster（每个SparkContext都有一个ApplicationMaster）  
**3.**NodeManager启动Spark App Master，并向ResourceManager AsM注册  
**4.** Spark ApplicationMaster从HDFS中找到jar文件，启动DAGscheduler和YARN Cluster Scheduler  
**5.** ResourceManager向ResourceManager AsM注册申请container资源（INFO YarnClientImpl: Submitted application）  
**6.**ResourceManager通知NodeManager分配Container，这时可以收到来自ASM关于container的报告。（每个container的对应一个executor）  
**7.**Spark ApplicationMaster直接和container（executor）进行交互，完成这个分布式任务。

出现问题：

**1. Master挂掉,standby重启也失效**

Master默认使用512M内存，当集群中运行的任务特别多时，就会挂掉，原因是master会读取每个task的event log日志去生成spark ui，内存不足自然会OOM，可以在master的运行日志中看到，通过HA启动的master自然也会因为这个原因失败。

**解决：**

增加Master的内存占用，在Master节点spark-env.sh 中设置：

export SPARK\_DAEMON\_MEMORY 10g # 根据你的实际情况

减少保存在Master内存中的作业信息

spark.ui.retainedJobs 500 # 默认都是1000

spark.ui.retainedStages 500

**2. worker挂掉或假死**

有时候我们还会在web ui中看到worker节点消失或处于dead状态，在该节点运行的任务则会报各种 lost worker 的错误，引发原因和上述大体相同，worker内存中保存了大量的ui信息导致gc时失去和master之间的心跳。

**解决：**

增加Master的内存占用，在Worker节点spark-env.sh 中设置：

export SPARK\_DAEMON\_MEMORY 2g # 根据你的实际情况

减少保存在Worker内存中的Driver,Executor信息

spark.worker.ui.retainedExecutors 200 # 默认都是1000

spark.worker.ui.retainedDrivers 200

**运行时错误**

**1.shuffle FetchFailedException**

错误提示

1、missing output location

org.apache.spark.shuffle.MetadataFetchFailedException:

Missing an output location for shuffle 0

2、shuffle fetch faild

org.apache.spark.shuffle.FetchFailedException:

Failed to connect to spark047215/192.168.47.215:50268

当前的配置为每个executor使用1core,5GRAM,启动了20个executor

**解决：**

这种问题一般发生在有大量shuffle操作的时候,task不断的failed,然后又重执行，一直循环下去，直到application失败

一般遇到这种问题提高executor内存即可,同时增加每个executor的cpu,这样不会减少task并行度。

spark.executor.memory 15G

spark.executor.cores 3

spark.cores.max 21

启动的execuote数量为:7个

execuoterNum = spark.cores.max/spark.executor.cores

每个executor的配置：3core,15G RAM

消耗的内存资源为:105G RAM

15G\*7=105G

可以发现使用的资源并没有提升，但是同样的任务原来的配置跑几个小时还在卡着，改了配置后几分钟就能完成。

**2.Executor&Task Lost**

错误提示

1、executor lost

WARN TaskSetManager: Lost task 1.0 in stage 0.0 (TID 1, aa.local):

ExecutorLostFailure (executor lost)

2、task lost

WARN TaskSetManager: Lost task 69.2 in stage 7.0 (TID 1145, 192.168.47.217):

java.io.IOException: Connection from /192.168.47.217:55483 closed

3、各种timeout

java.util.concurrent.TimeoutException: Futures timed out after [120 second]

ERROR TransportChannelHandler: Connection to /192.168.47.212:35409

has been quiet for 120000 ms while there are outstanding requests.

Assuming connection is dead; please adjust spark.network.

timeout if this is wrong

**解决**

**由网络或者gc引起,worker或executor没有接收到executor或task的心跳反馈。**

提高 spark.network.timeout 的值，根据情况改成300(5min)或更高。

默认为 120(120s),配置所有网络传输的延时，如果没有主动设置以下参数，默认覆盖其属性

spark.core.connection.ack.wait.timeout

spark.akka.timeout

spark.storage.blockManagerSlaveTimeoutMs

spark.shuffle.io.connectionTimeout

spark.rpc.askTimeout or spark.rpc.lookupTimeout

**3.倾斜**

1、数据倾斜

查看监控页面，发现个别作业执行缓慢

分为数据倾斜和任务倾斜

针对数据倾斜，大多数情况由无效数据引起，先过滤再计算

任务倾斜，原因较多，大多由io,cpu,mem引起运行缓慢，看性能监控分析原因，或者开启推测执行机制。

spark.speculation true

spark.speculation.interval 100 - 检测周期，单位毫秒；

spark.speculation.quantile 0.75 - 完成task的百分比时启动推测

spark.speculation.multiplier 1.5 - 比其他的慢多少倍时启动推测。

**4.OOM**

错误提示

堆内存溢出

java.lang.OutOfMemoryError: Java heap space

**解决**

内存不够，数据太多就会抛出OOM的Exeception，主要有driver OOM和executor OOM两种

driver OOM

一般是使用了collect操作将所有executor的数据聚合到driver导致。尽量不要使用collect操作即可。

executor OOM

可以按下面的内存优化的方法增加code使用内存空间

1增加executor内存总量,也就是说增加spark.executor.memory的值

2增加任务并行度（大任务就被分成小任务了)，参考下面优化并行度的方法

**5.task not serializable**

错误提示

org.apache.spark.SparkException: Job aborted due to stage failure:

Task not serializable: java.io.NotSerializableException: ...

解决

如果你在worker中调用了driver中定义的一些变量，Spark就会将这些变量传递给Worker，这些变量并没有被序列化，所以就会看到如上提示的错误了。

val x = new X() //在driver中定义的变量

dd.map{r => x.doSomething(r) }.collect //map中的代码在worker(executor)中执行

除了上文的map,还有filter,foreach,foreachPartition等操作，还有一个典型例子就是在foreachPartition中使用数据库创建连接方法。这些变量没有序列化导致的任务报错。

下面提供三种解决方法：

1、将所有调用到的外部变量直接放入到以上所说的这些算子中，这种情况最好使用foreachPartition减少创建变量的消耗。

2、将需要使用的外部变量包括sparkConf,SparkContext,都用 @transent进行注解，表示这些变量不需要被序列化

3、将外部变量放到某个class中对类进行序列化。

**6.driver.maxResultSize太小**

错误提示

Caused by: org.apache.spark.SparkException:

Job aborted due to stage failure: Total size of serialized

results of 374 tasks (1026.0 MB) is bigger than

spark.driver.maxResultSize (1024.0 MB)

解决

spark.driver.maxResultSize默认大小为1G 每个Spark action(如collect)所有分区的序列化结果的总大小限制，简而言之就是executor给driver返回的结果过大，报这个错说明需要提高这个值或者避免使用类似的方法，比如countByValue，countByKey等。

将值调大即可

spark.driver.maxResultSize 2g

1. taskSet too large

WARN TaskSetManager: Stage 198 contains a task of very large size (5953 KB). The maximum recommended task size is 100 KB.

Caused by: java.lang.RuntimeException: Failed to commit task

Caused by: org.apache.spark.executor.CommitDeniedException: attempt\_201603251514\_0218\_m\_000245\_0: Not committed because the driver did not authorize commit

解决

如果你比较了解spark中的stage是如何划分的，这个问题就比较简单了。

一个Stage中包含的task过大，一般由于你的transform过程太长，因此driver给executor分发的task就会变的很大。

所以解决这个问题我们可以通过拆分stage解决。也就是在执行过程中调用cache.count缓存一些中间数据从而切断过长的stage。