数据本地性对 Spark 生产作业容错能力的负面影响

Kent_Yao 浪尖聊大数据 3/25

收录于话题
#spark 10 #大数据 23 #性能调优 13 #java 11 #面试 15

作者：Kent_Yao
链接：https://www.jianshu.com/p/72ffaa10220

数据本地性是 Spark 等计算引擎从计算性能方面去考量的一个重要指标，对于某个数据分片的运算，Spark 在调度侧会做数据本地性的预测，然后尽可能的将这个运算对应的Task调度到靠近这个数据分片的Executor上。

Spark 计算作业依赖于整个物理计算集群的稳定性，抛开软件层，如资源管理层（YARN，Kubernetes），存储层（HDFS）本身的稳定性不说，Spark 依赖于物理机器上的 CPU、 内存、 磁盘和网络进行真正的计算作业。单个物理机的硬件故障是一个小概率的事件，但当集群的规模到达成百上千甚至过万台，那以集群为维度，大大小小的硬件故障将成为一个常态。

关键字：TaskLocality, 容错, 已经故障

1. Spark TaskLocality

在 Spark 中数据本地性通过 TaskLocality 来表示，有如下几个级别，

- PROCESS_LOCAL
- NODE_LOCAL
- NO_PREF
- RACK_LOCAL
- ANY

从上到下数据本地性依次递减。

Spark 在执行前通过数据的分区信息进行计算 Task 的 Locality，Task 总是会被优先分配到它要计算的数据所在节点以尽可能地减少网络 IO。这个计算的过程通过 `spark.locality.wait` 默认为3s，控制这个计算的过程。

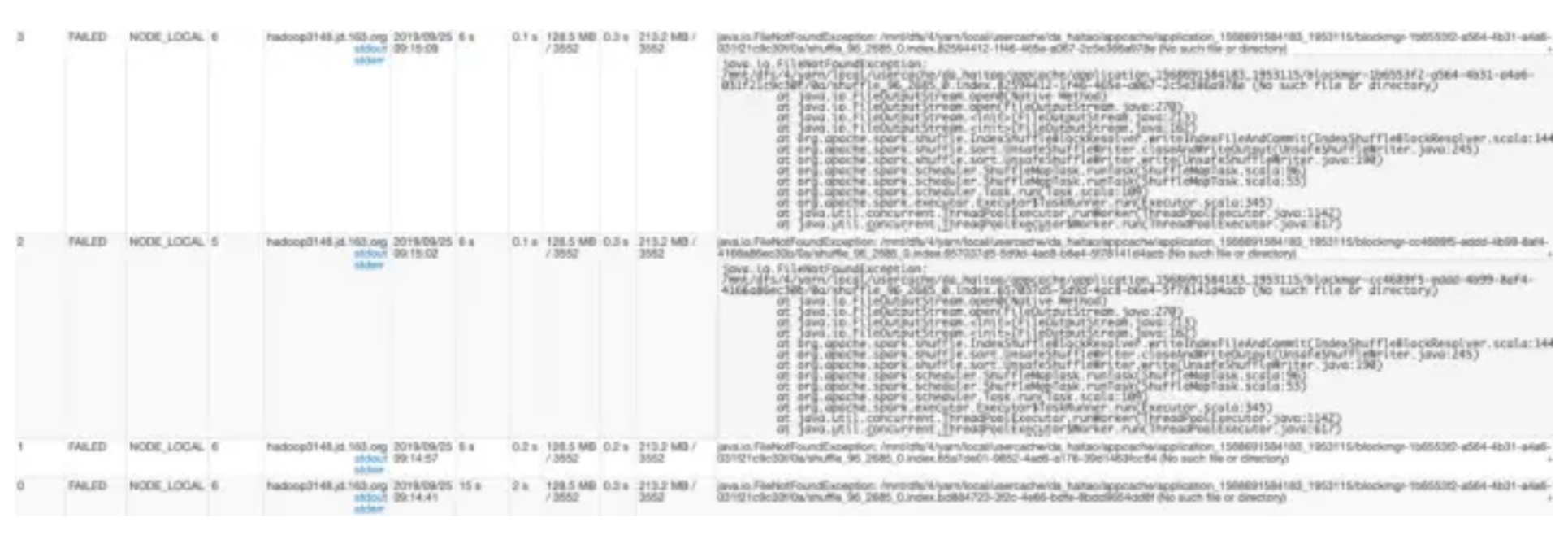
2. Spark 内部容错

原理这里不细讲，简而言之就是重试。Spark 规定了同一个 Job 中同一个 Stage 连续失败重试的上限（`spark.stage.maxConsecutiveAttempts`），默认为4，也规定了一个 Stage 中 同一个 Task 可以失败重试的次数（`spark.task.maxFailures`），默认为4。当其中任何一个阈值达到上限，Spark 都会使整个 Job 失败，停止可能的“无意义”的重试。

3. 数据本地性和容错的冲突

我们首先来看一个例子，如图所示，图为 Spark Stage 页面下 Task Page 的详细视图。

- 第一列表示该 Task 进行了4次重试，所以这个 Task 对应的 Job 也因此失败了。
- 第三列表示该 Task 的数据本地性，都是 NODE_LOCAL 级别，对于一个从HDFS读取数据的任务，显然获得了最优的数据本地性
- 第四列表示的是 Executor ID，我们可以看到我们任务的重试被分配到ID 为5和6两个 Executor 上
- 第五列表示我们运行这些重试的 Task 所在的 Executor 所在的物理机地址，我们可以看到他们都被调度到了同一个
- 最后列表示每次重试失败的错误栈



Spark Stage 页面下 Task Page 的详细视图

3.1 问题一：单个 Task 重试为什么失败？

结合硬件层面的排查，发现是 NodeManager 物理节点上挂在的 /mnt/dfs/4，出现硬件故障导致盘只读，ShuffleMapTask 在即将完成时，将index文件和data文件commit时，获取index的临时文件时候发生`FileNotFoundException`。

```
java.io.FileNotFoundException: /mnt/dfs/4/yarn/local/usercache/da_haita
    at java.io.FileOutputStream.open0(Native Method)
    at java.io.FileOutputStream.open(FileOutputStream.java:270)
    at java.io.FileOutputStream.<init>(FileOutputStream.java:213)
    at java.io.FileOutputStream.<init>(FileOutputStream.java:162)
    at org.apache.spark.shuffle.IndexShuffleBlockResolver.writeIndexFil
    at org.apache.spark.shuffle.sort.UnsafeShuffleWriter.closeAndWriteO
    at org.apache.spark.shuffle.sort.UnsafeShuffleWriter.write(UnsafeSh
    at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask
    at org.apache.spark.scheduler.ShuffleMapTask.runTask(ShuffleMapTask
    at org.apache.spark.scheduler.Task.run(Task.scala:109)
```

`3.2` 问题二：为什么该 `Task` 的4次重试都在同一个物理节点？

这是由于 Driver 在调度该 Task 的时候进行了数据本地性的运算，而且在`spark.locality.wait` 默认为3s的时间约束内成功获得了NODE_LOCAL级别的数据本地性，故而都调度到了同一个 `NodeManger` 物理节点。

3.3 问题三：为什么总是“本地重试”，不是“异地重试”？

这个过程从逻辑上讲，其实已经不是“本地重试”，而恰恰是“异地重试”了。这我们可以从4次的重试的 Executor ID 上进行判断，第0、1和3次是在 ID 6上进行的，而第2次是在 ID 5上发生的。但由于ID 5和6都在同一个 NodeManger 节点，所以我们看起来像是“本地重试”。另一个原因就是上面所说的数据本地性的成功解析，所以这些 Task 的每次重试都高概率的来到这个节点。
所有 Spark Task 级别的重试从逻辑上都应该属于“异地重试”，他们都需要通过 Driver 重新调度到新的 Executor 进行重试。我们所观测到的“本地”和“异地”是属于“现象”而非“本质”，影响这种现象的条件有比如下面几个（不一定全面）：1. 数据本地性 2. Executor 由于 NodeLabel 限制，只在若干有限的物理机上分配 3. ResourceManager 调度时刚好把所有的 Executor 都分配到某个节点上。

3.4 问题5：为什么4次失败都操作同一个坏的盘？

该 NodeManger 实际上有/mnt/dfs/{0-11}， 一共12块盘，从物理检查上看，整个过程中也只有/mnt/dfs/4有异常告警，那为啥 Spark 这么傻？这么多好盘不用，专挑一块坏的盘死磕？

我们可以先看下出错的文件，我们包这个文件分成5个部分来看，

```
1. /mnt/dfs/4/yarn/local/
2. usercache/da_haitao/appcache/application_1568691584183_1953115/ block
3. 0a/
4. shuffle_96_2685_0.index
5. .82594412-1f46-465e-a067-2c5e386a978e
```

- 第一行，是 Yarn NodeManger 所配置的LOCAL_DIR的一部分，完整的应该包括12块盘
- 第二行，是 Spark 生成的 BlockManger 的根目录之一，其他盘符下也有类似的一个目录
- 第三行，是一个根目录下的一级子目录，数量由`spark.diskStore.subDirectories` 默认为64控制
- 第四行，Spark Shuffle 过程产生的两个重要的文件之一，一个是数据文件 `.data` 结尾，另一个就是这个与之对应的 `.index` 文件。96是 ShuffleID 表标识是哪个 Shuffle 过程，2685是 MapID 对应的是 一个RDD 所以有分区中其中一个的顺序号， 而0是一个固定值，原本表示是ReduceID，Spark Sort Based Shuffle 的实现不需要依赖这个值，所以被固定为了0。通过Shuffle ID和 MapId，Shufle Write 阶段就可以生成类似shuffle_96_2685_0.index这样的文件，而Shuffle Read 阶段也可以通过两个ID 定位到这个文件。
- 第五行， 是Index文件的对应临时文件的UUID标识。

基于这样的逻辑，对于某次Shuffle 过程的某个分区（Partition）的最终输出文件名其实是可以预测的也是固定的，比如我们这个 case 中，第96次shuffle的第2685分区的index 文件的文件名即为shuffle_96_2685_0.index。

Spark 在写和读这个文件的时候，基于相同的定位逻辑（算法）来保证依赖关系，
第一步确定根目录，Spark 通过文件名的hash绝对值与盘符数的模，作为索引却确定根目录

```
scala> math.abs("shuffle_96_2685_0.index".hashCode) % 12
res0: Int = 6
```

而根目录的数组对于一个 Executor 的这个生命周期内而言是确定的，它是一个由简单随机算法将所有路径打散的一个固定数组。所以一旦文件名称确定，Executor 不换的话，根目录一定是确定的。所以都固定的去访问/mnt/dfs/4这个坏盘。
但这只解释了一个 Executor 所被分配 Task 失败的原因，我们的 Task 还在不同的 executor 上进行过尝试。

3.5 问题5：为什么两个 Executor 上的重试都失败了？

其实这个问题只是概率的问题, Spark 用类似下面算法打乱所有LOCAL_DIRS的配置，如下面的的简单测试，这种碰撞的概率还是极高的，我们ID 5，6，的 Executor 下 DiskBlockManager 包含的 localDirs(6)应该都对应于 /mnt/dfs/4 这个坏盘。

```
scala> def randomizeInPlace[T](arr: Array[Int], rand: java.util.Random
     |     for (i <- (arr.length - 1) to 1 by -1) {
     |       val j = rand.nextInt(i + 1)
     |       val tmp = arr(j)
     |       arr(j) = arr(i)
     |       arr(i) = tmp
     |     }
     |     arr
     |   }
randomizeInPlace: [T](arr: Array[Int], rand: java.util.Random)Array[Int
scala> randomizeInPlace(res11)
res23: Array[Int] = Array(3, 2, 4, 1)

scala> randomizeInPlace(res11)
res24: Array[Int] = Array(2, 3, 4, 1)

scala> randomizeInPlace(res11)
res25: Array[Int] = Array(2, 1, 3, 4)

scala> randomizeInPlace(res11)
res26: Array[Int] = Array(4, 2, 1, 3)

scala> randomizeInPlace(res11)
res27: Array[Int] = Array(2, 3, 4, 1)
```

4. 总结

4.1 问题原因

集群某个或某几个物理机上某块或某几块盘出现磁盘问题时，Spark 由于数据本地性原因反复把 Task 调度到这个节点的某个 Executor，或这个节点的其他 Executor 上， 前者必然失败，后者有概率失败。
当然忽略数据本地性进行随机调度，也有一定的概率出现“现象”为“本地重试”的这种失败场景，但数据本地性的策略会极大的放大这个概率。

4.2 规避方案

- 设置 `spark.locality.wait=0s`，让 Task 有更大的概率调度到别的节点，当然可能会影响一定的性能
- 设置 `spark.blacklist.enabled=true`, 开启黑名单，把问题节点加到黑名单中，暂不参与 Task 的分配。当然使用黑名单的话，不注意也很容易踩坑。

4.3 解决方案

说来也巧，在我刚去社区提https://issues.apache.org/jira/browse/SPARK-29257这个JIRA，并沟通初步方案时，发现社区在两天之前刚将 https://github.com/apache/spark/pull/25620 这个Pull request合入了，虽然这个PR不是专门解决我所提到的这个问题的，但它确产生了一个副作用，刚好解决了这个问题。
本质的想法就是构建`shuffle_${shuffleId}_${mapId}_0.index` 这类Shuffle 文件时，可以让每次重试都可以生成 Unique 的文件名，这样就可以生成不同的 hash 值并挑选别的盘作为根目录了，这样就不会一直在一块坏盘上吊死。这个PR中已经将mapId换成了每个 task 的 taskAttemtId，而这个值就是unique的，所以天然就解决了这个问题。
对于2.x的 Spark 版本，大家可以尝试合入这个PR.

5. 参考文献

https://issues.apache.org/jira/browse/SPARK-29257
https://github.com/apache/spark/pull/25620

收录于话题 #spark·10个 >

上一篇
列一些常见的Spark面试题

下一篇
面试｜spark刷爆磁盘与java弱引用的关系