**Hadoop性能调优小结（一）**

www.MyException.Cn  网友分享于：2015-05-26  浏览：0次

Hadoop性能调优总结（一）

**目的**

随着企业要处理的数据量越来越大，Hadoop运行在越来越多的集群上，同时MapReduce由于具有高可扩展性和容错性，已经逐步广泛使用开来。因此也产生很多问题，尤其是性能方面的问题。这里从管理员角度和用户角度分别介绍Hadoop性能优化的一些体会。

本文是基于Hadoop 0.20.x(包括1x),cdh 3及以上版本做介绍。（Hadoop的版本比较杂乱，具体可以看参考部分链接介绍）。

**管理员角度**

1.    硬件选择： Master机器配置的选择要高于slave机器配置的选择。

2.    磁盘I/O：由于磁盘I/O的速度是比较慢的，如果一个进程的内存空间不足，它会将内存中的部分数据暂时写到磁盘，当需要的时候，再把磁盘上面的数据写到内存上面。因此设置和的预读缓冲区大小来提高hadoop里面大文件顺序读的性能。以此来提高I/O性能。

**用户角度**

**Hadoop参数调优:**

通过修改hadoop三个配置文件的参数来提高性能。主要有三个文件core-site.xml、hdfs-site.xml、mapred-site.xml。下面分别介绍这三个文件常用的参数配置。我们的环境上面的路径是: /usr/lib/hadoop/etc/hadoop

三个配置文件介绍：

**core-site.xml**

该文件中是集群的一些基本参数，与hadoop部署密切相关，但是对于性能优化作用不是特别明显。这里就简单介绍几个常用的配置参数。

|  |  |
| --- | --- |
| fs.default.name | 主节点地址。 |
| hadoop.tmp.dir | 集群的临时文件存放目录。 |
| io.file.buffer.size | 系统I/O的属性，读写缓冲区的大小。 |
| io.seqfile.compress.blocksize | 块压缩时块的最小块大小。 |
| io.seqfile.lazydecompress | 压缩块解压的相关参数。 |

**hdfs-site.xml**

该文件与HDFS子项目密切相关，其参数对集群性能调整具有很大影响。

|  |  |
| --- | --- |
| dfs.name.dir | 指定name镜像文件存放目录，如不指定则默认为core-site中配置的tmp目录。 |
| dfs.data.dir | 真正的datanode数据保存路径，可以写多块硬盘，中间以逗号分隔。 |
| dfs.replication | hdfs数据块的复制份数，默认3，理论上份数越多跑数速度越快，但是需要的存储空间也更多。 |
| dfs.permissions | 是否需要角色权限验证，上传文件时会用到。一般设置false，设置为true有时候会遇到数据因为权限访问不了。 |
| dfs.block.size | 每个文件块的大小，默认是64M，对于大型文件可以设置为128M。 |
| dfs.namenode.handler.count | NameNode 节点上面为处理datanode节点来来气的远程调用的服务线程数量。 |
| dfs.datanode.max.xcievers | 相当于linux下的打开文件最大数量，文档中无此参数，当出现DataXceiver报错的时候，需要调大。默认256 |
| dfs.datanode.handler.count | datanode节点上为处理datanode节点的远程调用开启的服务线程数量，默认为3。当有很多HDFS客户端时可以设置更大。 |

**mapred-site.xml**

该文件与mapreduce计算模型密切相关，其中的参数对集群的性能影响很大。

|  |  |
| --- | --- |
| mapred.job.tracker | Job tracker地址 |
| mapred.job.tracker.handler.count | jobtracker服务的线程数，一般默认为15. |
| mapred.map.tasks | 默认每个job所使用的map数，意思是假设设置dfs块大小为64M，需要排序一个60M的文件，也会开启2个map线程，当jobtracker设置为本地是不起作用。 |
| mapred.reduce.tasks | 每个job的reduce任务数量，经常设置成与集群中存在的主机数量很接近的一个数值。 |
| mapred.tasktracker.map.tasks.maximum | 一个task tracker上可以同时运行的map任务的最大数量。 |
| mapred.tasktracker.reduce.tasks.maximum | 一个task tracker上可以同时运行的reduce任务的最大数量。 |
| io.sort.mb | 排序所使用的内存数量。默认值：100M，需要与mapred.child.java.opts相配 默认：-Xmx200m。不能超过mapred.child.java.opt设置，否则会OOM。 |
| io.sort.factor | 处理流merge时的文件数， 默认：10  ,建议调大到100. |

**参数设置调优：**

对于这些常用的参数设置，需要整体把握的一些主体思想：

（1）       Reduce个数设置：

A． 如果reduce个数设置太小，单个reducetask执行速度很慢，这样出错重新调试的时间花销就比较多。

B． 如果怕Reduce个数设置太大，Shuffle开销及调度开销很大，job输出大量文件，影响后续Job的执行。

C． 推荐的reduce的个数。单个reducetask处理数据量介于1~10G之间，reduce的个数要少于map的个数。

（2）       压缩中间数据，用CPU换磁盘和网络，设置mapred.compress.map.output设为true

A． 减少磁盘操作

B． 减少网络传输数据量

**实验数据：**

  该实验数据源自网络，详见参考链接。我们这里主要是举例看一下Reduce任务数对性能的影响。

1.      Reduce Task设置（数据量为1GB）。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Map task = 16 | | | | | | | | | | |
| Reduce Task | 1 | 5 | 10 | 15 | 16 | 20 | 25 | 30 | 45 | 60 |
| Total Time | 892 | 146 | 110 | 92 | 88 | 100 | 128 | 101 | 145 | 104 |
| Map Time | 24 | 21 | 25 | 50 | 21 | 40 | 24 | 48 | 109 | 25 |
| Reduce Time | 875 | 125 | 88 | 71 | 67 | 76 | 102 | 80 | 98 | 83 |
| Killed map/reduce Task Attempts | 0/0 | 0/2 | 0/2 | 0/5 | 0/4 | 0/9 | 0/9 | 0/8 | 1/7 | 0/17 |

a)        当reduce task<15时，Total Time和Reduce Time都与Reduce task数量成反比关系。当reduce task>15时，TotalTime和ReduceTime基本保持恒定。Reducetask的数量应该设置为接近slave节点数量，或者适当大于节点数，不宜设置为比节点数量小太多。

b)        Map时间与Reduce task之间没有明显的关系。

c)        Killed map Task Attempts的值对Map的时间影响很大，表1中当reduce task = 45时，Killedmap Task Attempts的值为1，此时Map的时间很长，从图1可看出，map的时间主要集中在map99%的最后阶段。

d)        job运行过程中产生Killed Task Attempts的原因： hadoop里面对task的speculative机制。简单来说就是hadoop觉得有些task运行过慢，所以它在其它tasktracker上同时再运行同样的任务，当其中一个完成后，其余同样的任务就会被kill掉。这就造成有多个被kill的taskattempt。可以通过设置mapred.map.tasks.speculative.execution为false来禁止hadoop的这种行为，这样可以提高效率，因为每个speculative都是占用task的slot的。

**Hadoop作业调度调优：**

虽然这些hadoop参数的配置可以很好的提高性能，但是，这些方式只是静态的对集群性能做优化，在job运行的时候无法动态的修改配置文件并使加载生效。因此我们需要考虑动态的性能调优，首先在作用调度方面着手。

**Hadoop作业流程介绍：**

—  A. JobClient向master节点的JobTracker提交一个mapreduce作业,JobTracker接到JobClient的请求后把其加入作业队列中。

—  B. JobTracker一直在等待JobClient通过RPC提交作业.

—  C. TaskTracker一直通过RPC向 JobTracker发送heartbeat询问有没有任务可做，如果有，让其派发任务给它执行。如果JobTracker的作业队列不为空,则TaskTracker发送的心跳将会获得JobTracker给它派发的任务。

—  D. slave节点的TaskTracker接到任务后在其本地发起Task,执行任务。

从流程里面可以看出，优化hadoop作业调度可以很好的提高性能。下面介绍三种Hadoop作业调度算法：默认调度算法FIFO、公平调度算法默认调度算法FairScheduler和计算能力调度算法CapacityScheduler。

**调度算法介绍：**

**默认调度算法FIFO**

FIFO, 它先按照作业的优先级高低，再按照到达时间的先后选择被执行的作业。

Ø 优点 : 简单、易于实现，同时也减轻了jobtracker的负担。

Ø 缺点 : 对所有的作业都一视同仁，没有考虑到作业的紧迫程度。例如如果类似对海量数据进行统计分析的作业长期占据计算资源，那么在其后提交的交互型作业有可能迟迟得不到处理，从而影响到用户的体验。

当JobTracker给某个TaskTracker分配任务时，它就会调用TaskScheduler的assignTasks(TaskTrackerStatus)方法，让TaskScheduler给该TaskTracker分配任务。那么，究竟TaskScheduler是如何给TaskTracker任务分配任务的，这就得看TaskScheduler的具体实现了。

FIFO算法在hadoop源码里面是JobQueueTaskScheduler这个调度器，里面主要通过如下这个方法分派任务：

**[java]** view plaincopy

1. **public** **synchronized** List<Task> assignTasks(TaskTracker taskTracker)
2. **throws** IOException
3. {}

当JobQueueTaskScheduler调用assignTasks来分配任务的时候，如果发现当前的TaskTracker有空闲的槽，就会为其寻找合适的任务，当分配好一个任务后，JobQueueTaskScheduler就会调用exceededPadding函数去判断一下，是否超过保留一定槽的限制，如果是，则只分配一个这一类型的任务就跳出。

以上这段逻辑在exceededPadding方法里面实现。绿色文字注释部分是简单的步骤。

**[java]** view plaincopy

1. **private** **boolean** exceededPadding(**boolean** isMapTask, ClusterStatus clusterStatus, **int** maxTask-TrackerSlots)
2. {
3. //1.获取集群的taskTracker的数目
4. **int** numTaskTrackers = clusterStatus.getTaskTrackers();
5. //2. 得到集群的map或者reduce任务数
6. **int** totalTasks = isMapTask ? clusterStatus.getMapTasks() : clusterStatus.getReduceTasks();
7. //3. 得到集群最大的map或reduce资源槽数
8. **int** totalTaskCapacity = isMapTask ? clusterStatus.getMaxMapTasks() : cluster-Status.getMaxReduceTasks();
9. Collection jobQueue = **this**.jobQueueJobInProgressListener.getJobQueue();
10. **boolean** exceededPadding = **false**;
11. **int** totalNeededTasks;
12. **synchronized** (jobQueue) {
13. totalNeededTasks = 0;
14. **for** (JobInProgress job : jobQueue) {
15. **if** ((job.getStatus().getRunState() != 1) || (job.numReduceTasks == 0))
16. {
17. **continue**;
18. }
19. // 4.计算所有job所需要的map或reduce任务数
20. totalNeededTasks += (isMapTask ? job.desiredMaps() : job.desiredReduces());
21. **int** padding = 0;
22. **if** (numTaskTrackers > 3) {
23. // 5.如果是map任务，那maxTaskTrackerSlots就是tracker上最大的map槽数，reduce的话同理。
24. padding = Math.min(maxTaskTrackerSlots, (**int**)(totalNeededTasks \* **this**.padFraction));
25. }
26. // 6.如果集群已占用的某种任务槽数加上当前TaskTracker应当保留的槽数大于集群某种任务的最大槽数
27. **if** (totalTasks + padding >= totalTaskCapacity) {
28. exceededPadding = **true**;
29. **break**;
30. }
31. }
32. }
34. **return** exceededPadding;
35. }

通过以上的介绍，对于JobQueueTaskScheduler的任务调度实现原则可以按照下面的方式来调优性能。

A． 先调度优先级高的作业，统一优先级的作业则先进先出

B． 尽量使集群中每一个TaskTracker达到负载均衡(这个均衡是task数量上的而不是实际的工作强度)

C． 尽量分配作业的本地任务给TaskTracker，但不是尽快分配作业的本地任务给TaskTracker，最多分配一个非本地任务给TaskTracker(一是保证任务的并发性，二是避免有些TaskTracker的本地任务被偷走)，最多分配一个reduce任务

D． 为优先级或者紧急的Task预留一定的slot；

**公平调度算法 Fair Scheduler**

背景：

由于不同用户提交的作业在计算时间、存储空间、数据流量和响应时间上都有不同需求。为使hadoopmapreduce框架能够应对多种类型作业并行执行，使得用户具有良好的体验，Facebook公司提出该算法。

介绍：

公平调度器按资源池(pool)来组织作业，并把资源公平的分到这些资源池里面。

—  用户提交的作业将会放进一个能够公平共享资源的pool(池)中。

—  每个作业池设定了一个最低资源保障(aguaranteed minimum share)，当一个池中包含job时，它至少可以获得minimumshare的资源——最低保障资源份额机制。

—  池中的作业获得一定份额的资源。

—  可以通过配置文件限制每个池中的作业数量。

—  缺省情况下，每个作业池中选择将要执行的作业的策略是FIFO策略，先按照优先级高低排序，然后再按照提交时间排序。

操作：

A．安装公平调度器

—  将FairScheduler的jar文件复制到lib目录下。然后更改mapred-site.xml配置文件使hadoop使用此调度器。

—  重启hadoop，可以在http://<jobtrackerurl>/scheduler下查看相应的web页面jobscheduler administration。

B. 公平调度器的池配置

—  在mapred-site.xml文件中可以通过mapred.fairscheduler.allocation.file参数设定FairScheduler的池配置文件的路径。

—

—  FairScheduler中池和作业的相关参数可以在配置文件pools.xml中设定。

—  调度器会每隔10-15秒检查一次pools.xml，如果发现有更新将重新加载它使之生效。

—  在作业运行时我们可以修改pools.xml中的相关参数，从而实现动态调度。

**计算能力调度CapacityScheduler**

背景：

Capacity Scheduler是由雅虎提出的作业调度算法，它提供了类似于Fair Scheduler算法的功能。

介绍：

—  计算能力保证。支持多个队列，某个作业可被提交到某一个队列中。每个队列会配置一定比例的计算资源，且所有提交到队列中的作业共享该队列中的资源。

—  灵活性。空闲资源会被分配给那些未达到资源使用上限的队列，当某个未达到资源的队列需要资源时，一旦出现空闲资源，便会分配给他们。

—  支持优先级。队列支持作业优先级调度（默认是FIFO）

—  多重租赁。综合考虑多种约束防止单个作业、用户或者队列独占队列或者集群中的资源。

—  基于资源的调度。 支持资源密集型作业，允许作业使用的资源量高于默认值，进而可容纳不同资源需求的作业。不过，当前仅支持内存资源的调度。

 安装 ：

—  CapacityScheduler的jar文件复制到lib目录下，然后更改mapred-site.xml配置文件使hadoop使用此调度器。

—  在配置文件mapred-site.xml中定义队列。

—  可以在CapacityScheduler配置文件(conf/capacity-scheduler.xml)中设置每个队列的一系列属性以控制调度。

**调度算法总结:**

—  Hadoop集群作业调度算法也是当前研究的热门，当前大量的设计与实现围绕着作业调度展开，以求优化集群性能。

—  然而，对于只有一个jobtracker的hadoop框架来说，经常会出现大规模的作业提交和运行，在其上运行的调度算法一定不能过于复杂，否则将会给jobtracker带来繁重的工作压力，一旦jobtracker宕机，后果将不堪设想，这也是FIFO调度算法始终还没有被遗弃的原因之一。

—  究竟该如何调度作业必须视具体情况而定。

**Hadoop（Map/Reduce）程序编写调优：**

下面这些建议主要是**Todd Lipcon(@tlipcon)**做的关于如何从map/reduce程序编写方面调优Hadoop的性能。

**正确的配置集群(Configure your cluster correctly)**

      如果是一大批MR程序，如果可以设置一个Combiner，Combiner可减少MapTask中间输出结果，从而减少各个ReduceTask的远程拷贝数据量，最终表现为MapTask和ReduceTask执行时间缩短。

**使用LZO压缩（Use LZO Compression）**

      当一个job需要输出大量数据时，应用LZO压缩可以提高输出端的输出性能。这是因为默认情况下每个文件的输出都会保存3个幅本，1GB的输出文件你将要保存3GB的磁盘数据，当采用压缩后当然更能节省空间并提高性能。   
为了使LZO压缩有效，请设置参数mapred.compress.map.output值为true。

**设置合理的map和reduce数量（Tune the number ofmap and reduce tasks appropriately）**

      调整job中map和reducetask的数量是一件很重要且常常被忽略的事情。如果一个job的输入数据大于1TB，我们就增加blocksize到256或者512，这样可以减少task的数量。

你可以使用这个命令去修改已存在文件的blocksize: hadoop distcp -Ddfs.block.size=$[256\*1024\*1024]/path/to/inputdata  /path/to/inputdata-with/largeblocks。在执行完这个命令后，你就可以删除原始的输入文件了(/path/to/inputdata)。

     只要每个task运行至少30到40秒，那么就增加maptask的数量，增加到整个cluster上mapslot总数的几倍。如果你的cluster中有100个mapslot，那就避免运行一个有101个maptask的job— 如果运行的话，前100个map同时执行，第101个task会在reduce执行之前单独运行。这个建议对于小型cluste和小型job是很重要的。

不要调度太多的reduce task — 对于大多数job来说，我们推荐reduce task的数量应当等于或是略小于cluster中reduceslot的数量。

**设置combiner(Write a combiner)**

     Mapreduce中的Combiner就是为了避免map任务和reduce任务之间的数据传输而设置的，Hadoop允许用户针对maptask的输出指定一个合并函数。即为了减少传输到Reduce中的数据量。它主要是为了削减Mapper的输出从而减少网络带宽和Reducer之上的负载。

      Combiner，它在Mapper之后Reducer之前运行。Combiner是可选的，如果这个过程适合于你的作业，Combiner实例会在每一个运行map任务的节点上运行。Combiner会接收特定节点上的Mapper实例的输出作为输入，接着Combiner的输出会被发送到Reducer那里，而不是发送Mapper的输出。Combiner是一个“迷你reduce”过程，它只处理单台机器生成的数据。

     词频统计是一个可以展示Combiner的用处的基础例子，上面的词频统计程序为每一个它看到的词生成了一个（word，1）键值对。所以如果在同一个文档内“cat”出现了3次，（”cat”，1）键值对会被生成3次，这些键值对会被送到Reducer那里。通过使用Combiner，这些键值对可以被压缩为一个送往Reducer的键值对（”cat”，3）。现在每一个节点针对每一个词只会发送一个值到reducer，大大减少了shuffle过程所需要的带宽并加速了作业的执行。

测试结果:

   删去Wordcount例子中对setCombinerClass方法的调用。仅这个修改就让maptask的平均运行时间由33秒增长到48秒，shuffle的数据量也从1GB提高到1.4GB。整个job的运行时间由原来的8分30秒变成15分42秒，差不多慢了两倍。这次测试过程中开启了map输出结果的压缩功能，如果没有开启这个压缩功能的话，那么Combiner的影响就会变得更加明显。

**选择合理的Writable类型（Use the mostappropriate and compact Writable type for your data）**

      开发者们经常在不必要的时候使用Text 对象。尽管Text对象使用起来很方便，但它在由数值转换到文本或是由UTF8字符串转换到文本时都是低效的，且会消耗大量的CPU时间。当处理那些非文本的数据时，可以使用二进制的Writable类型，如IntWritable，FloatWritable等。

     除了避免文件转换的消耗外，二进制Writable类型作为中间结果时会占用更少的空间。当磁盘IO和网络传输成为大型job所遇到的瓶颈时，减少些中间结果的大小可以获得更好的性能。在处理整形数值时，有时使用VIntWritable或VLongWritable类型可能会更快些—这些实现了变长整形编码的类型在序列化小数值时会更节省空间。例如，整数4会被序列化成单字节，而整数10000会被序列化成两个字节。这些变长类型用在统计等任务时更加有效，在这些任务中我们只要确保大部分的记录都是一个很小的值，这样值就可以匹配一或两个字节。

    如果Hadoop自带的Writable类型不能满足你的需求，你可以开发自己的Writable类型。如果你编写了自己的Writable类型，请务必提供一个RawComparator类—你可以以内置的Writable类型做为例子。

**重用Writables（Reuse Writables）**

     在你的代码中搜索"new Text" 或"newIntWritable"。如果它们出现在一个内部循环或是map/reduce方法的内部时，这条建议可能会很有用。

     很多MapReduce用户常犯的一个错误是，在一个map/reduce方法中为每个输出都创建Writable对象。例如，你的Wordcoutmapper方法可能这样写：

**Java代码**

1. **public void** map(...) {

2.       …

3.       **for** (String word : words) {

4.               output.collect(**new** Text(word), **new** IntWritable(1));

5.       }

6. }

     这样会导致程序分配出成千上万个短周期的对象。Java垃圾收集器就要为此做很多的工作。更有效的写法是：

**Java代码**

1. **class** MyMapper … {

2.    Text wordText = **new** Text();

3.    IntWritable one = **new** IntWritable(1);

4.    **public void** map(...) {

5.          **for** (String word: words) {

6.                wordText.set(word);

7.                output.collect(wordText, one);

8.           }

9.        }

10.}

参考：

1.        http://blog.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/

2.        http://blog.csdn.net/xhh198781/article/details/7047354

3.        http://dongxicheng.org/mapreduce-nextgen/how-to-select-hadoop-versions/

4.        http://blog.csdn.net/xiejava/article/details/6432095

# 问题解决

## HDFS很慢

**日志如下：**

15/01/14 22:13:07 WARN datanode.DataNode: Slow BlockReceiver write packet to mirror took 6339ms (threshold=300ms)

15/01/14 22:13:26 INFO DataNode.clienttrace: src: /10.10.5.7:17276, dest: /10.10.5.4:50010, bytes: 176285, op: HDFS\_WRITE, cliID: DFSClient\_NONMAPREDUCE\_-832581408\_1, offset: 0, srvID: af886556-96db-4b03-9b5b-cd20c3d66f5a, blockid: BP-784291941-127.0.1.1-1420922413498:blk\_1073742333\_1531, duration: 19383299287

**解决：**

This was due to our network MTU, nothing to do with the FS. Our experimental MTU was far too-large, and was causing such behaviors. Getting back to the classical values bring things back to normal.