第46课：性能优化

本讲内容主要通过3个方面来讲解Spark性能优化：

1. Task性能优化
2. 慢任务情况：

Spark应用程序计算时候是通过Task进行计算，同时，Spark内部由于划分了Stage,如果本Stage中的某个Task计算速度特别慢，会拖累整个Stage的运算速度。这种情况我们通常称为慢任务。

Spark慢任务出现的情况主要有两种，一种是硬件故障，这种事情的解决方案只能换换换；第二种是由于任务倾斜导致的，由于Spark集群中存在多任务调度模式，导致不同任务可能获得到大小不等的资源，导致部分task计算资源不够降低计算速度。

解决方案：

1. 由于上诉所讲的情况有计算资源不够的情况，此时可以采用减少每个partition处理的数据量，防止因为内存不够，数据量过大导致的溢写到磁盘降低效率；
2. 建议开启Spark的任务推测模式(spark.speculation),这个模式中Driver会定时扫描是否有慢任务，如果出现慢任务，会开启多个任务处理同一个任务，优先选择最先完成的结果。

当然，这个方式的缺点也是很明显的，因为Driver端需要时不时扫描，会对资源产生一定的浪费。

1. Shuffle(Task阶段)
2. 在算法编写过程中，尽量减少groupByKey(),因为在计算过程中，groupByKey()方法需要通过网络拷贝一份数据；  
   更多情况推荐使用reduceByKey()因为reduceByKey()方法会对处理的数据先进性Local级别的Reduce从而减小Shuffle的数据量！
3. 在算法编写的过程中，使用Join操作时候尽量选择相同Key的数据进行join,原理有点类似Local级别的reduce，会在同一个节点先对key相同的先进行合并，但是如果key不相同，就需要进行全部数据的shuffle。
4. Repartition(重设定数据分片数)

在Spark分布式框架中RDD几乎可以说是数据计算的关键性的承载点，而RDD中Partition便是真正承载数据的数据分片，但是在Spark计算中会出现很多种数据分片数量不适应整个框架的情况：

1. 数据分片太大

单一的数据分片太大，会导致计算一个数据分片需要花费较多的时间，而且也容易导致内存管理的不充分，例如闲置或者溢出。

对于这种情况，通常采用通过repartition的方法将数据分片数增加，从而让每一个数据分片中的数据量减小，已达到更加充分的使用计算资源的目的。

1. 碎片文件太多(partition太多)

partiton数据太多，导致task需要频繁地通过复用线程的方式去计算很小片的数据，在线程调用过程中产生时间和资源的浪费。

对于这种情况通常会通过Repartition方法的Coalesce方法，根据你设定的分片数目返回一个新的RDD。这个过程中分片内容设置是通过HashPartitioner()进行打乱分配的，对于特殊的业务需求，可以通过重写HashPartitioner()来达到更加优化的Repartition效果。

1. Spark中的数据倾斜
2. 机器集中处理

由于Spark是分布式计算框架，框架本身会对资源进行调度和分配，但是由于Spark中的资源调度的数据本地性的计算优先级问题，会导致某些机器会集中处理大量数据，而其他机器可能只处理极少数数据。这部分的详细优化后面会有专门一期讲解，这边就暂时跳过。

1. 大型partition

Spark计算过程中，基于计算本身可能会导致某几个partition特别大，或者数据大量集中到某几个partition中，而由于Spark的调度机制，Task不执行完毕，本stage无法继续进行，这样便会导致程序卡顿在这个状态。

1. 对于这类的解决方案可以选择在算法中通过自定义分片算法，将数据产生的partition变小
2. 通过对于大型partition的数据采用一边写入一边读取处理的方式，加快Partition的处理速度: 可以考虑使用ByteBuffer来存储block数据，当然需要注意的这样的方式最大存储的数据为2G，关于ByteBuffer的更多内容，在下文附录提供。
3. 集群中的网络制约

分布式集群的最大的制约便是网络，也就是shuffle阶段，因为不管是数据写到磁盘还是内存，都远远快于通过网络传输(即使是内网传输)，目前Spark中可以采用以下几种方式来优化网络传输：

1. 采用tachyon

通过tachyon将数据提前从外部外部数据源加载到Tachyon中，以求获得更好的数据本地性(尽可能将数据发布到每个机器所在的节点)

1. Netty方式进行网络通信

Netty方式的网络通信是基于BIO和NIO的UDP传输，UDP的传输有点类似于现在的网络直播，你只需要将一份信息发布到网络，所有人都可以同时获得这份信息，而不是以往的TCP/IP方式需要建立通道，从而大大节约了网络带宽的压力。

1. 通过广播

通过广播的方式，提前将小批量的数据放置到需要Join的大批量数据的机器内存中，将数据通过本地级别的local,减少网络传输

1. mapPartitions

mapPartitions函数不同于map操作，对于每个partition中的每个数据进行操作，mapPartition中的函数会直接作用于整个Partition(一次)，例如在数据库连接中，应当使用mapPartition。

1. Process\_Local

Task计算的优先级源于，本进程内有一个task在排队等待计算，结果等待一定时间后(默认3秒)，此task还是没有被计算，此时会降低次task的计算优先级，可以被分配到下一个优先级的计算节点，例如同一个node的例外一个Executor。

Task的locality的优先级如下(优先级依次降低)

PROCESS\_LOCAL(同进程) NODE\_LOCAL(同node) NO\_PREF(同一个机器上) RACK\_LOCAL(同机架) ANY(任意情况)

1. 访问HBase,Canssandra

在访问这两个外部数据源的时候需要确保数据处理发送在数据所在的机器上，以减少数据在不同机器的传输的情况。例如华为对于Spark运行于HBase上开源代码。

**byteBuffer附录**

**概述**

ByteBuffer是NIO里用得最多的Buffer，它包含两个实现方式：HeapByteBuffer是基于Java堆的实现，而DirectByteBuffer则使用了unsafe的API进行了堆外的实现。这里只说HeapByteBuffer。

**ByteBuffer内部字段**

**byte[] buff**

buff即内部用于缓存的数组。

**position**

当前读取的位置。

**mark**

为某一读过的位置做标记，便于某些时候回退到该位置。

**capacity**

初始化时候的容量。

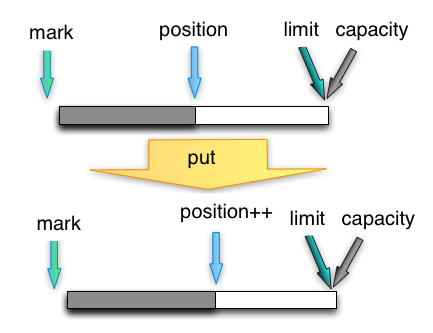
**limit**

读写的上限，limit<=capacity。

**以下是读写的大致实现流程**

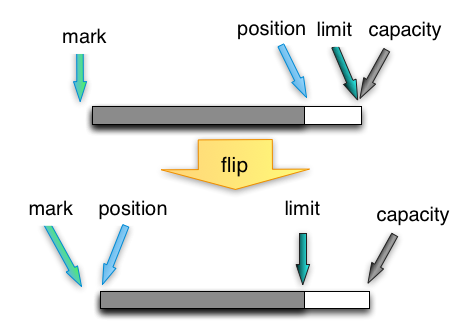
**put**

写模式下，往buffer里写一个字节，并把postion移动一位。写模式下，一般limit与capacity相等。

[](http://static.oschina.net/uploads/space/2013/0906/145012_wNAE_190591.png)

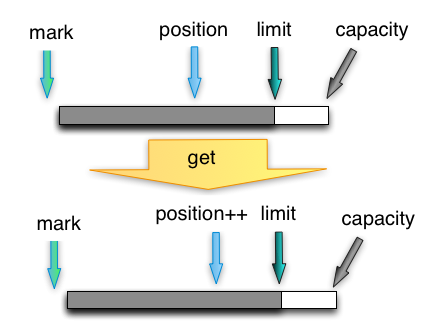
**flip**

写完数据，需要开始读的时候，将postion复位到0，并将limit设为当前postion。

[](http://static.oschina.net/uploads/space/2013/0906/145641_qTuf_190591.png)

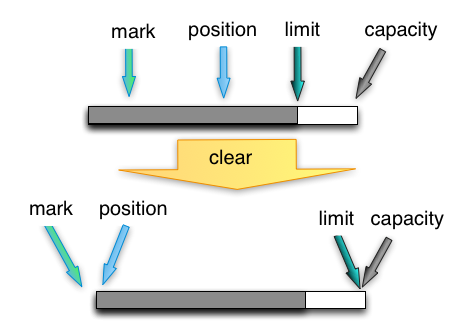
**get**

从buffer里读一个字节，并把postion移动一位。上限是limit，即写入数据的最后位置。

[](http://static.oschina.net/uploads/space/2013/0906/145049_jHFW_190591.png)

**clear**

将position置为0，并不清除buffer内容。

[](http://static.oschina.net/uploads/space/2013/0906/145130_5UA5_190591.png)

mark相关的方法主要是mark()(标记)和reset()(回到标记)，比较简单，就不画图了。