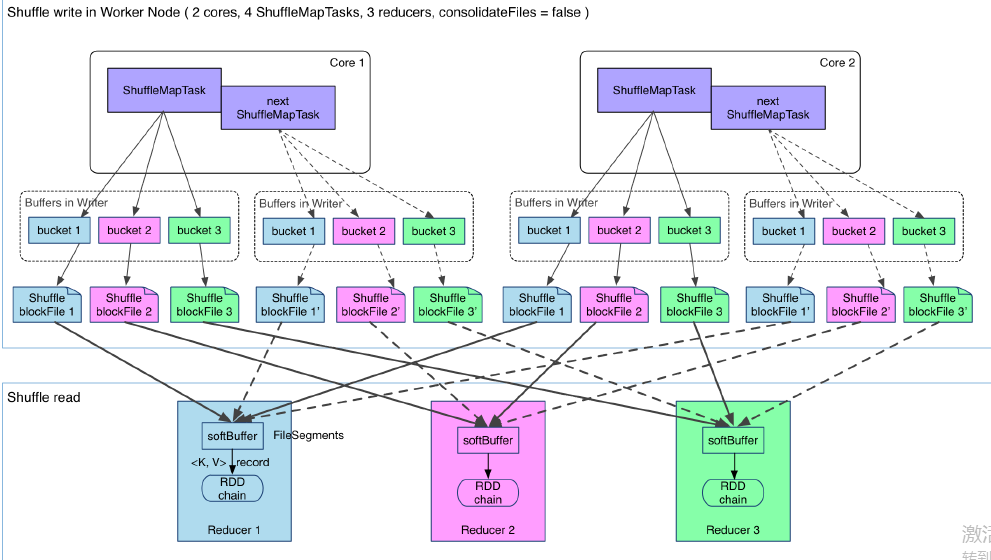
Shuffle Read: 将reducer读入数据，aggregate数据的过程。

Shuffle Write: Map端

划分数据和持久化数据

将数据partition好，并持久化。

持久化好处：一方面是要减少内存存储空间压力，另一个方面是为了fault-tolerance。



实现过程：

1. 将Shuffle Write 的处理逻辑加入到ShuffleMapStage(ShuffeMapTask所在的Stage中)，最后该Stage的final RDD没输出一个record就将其partition持久化。

**ShuffleMapTask执行过程：**

先利用pipeline计算得到finalRDD中对应partition的records。每得到一个record就将其送到对应的bucket里，具体是哪个bucket由partitioner.partition(record.getKey()))决定。每个bucket里面的数据会不断被写入到本地磁盘上，形成一个ShuffleBlockFile.或者简称FileSegment. 之后的 reducer 会去 fetch 属于自己的 FileSegment，进入 shuffle read 阶段。

三：默认Sort-ased Shuffle的几个缺陷：

1. 如果Mapper中Task的数量过大，依旧会产生很多小文件，此时在Shuffle传递数据的过程中到Reducer端，reduce会需要同时打开大量的记录来进行反序列化，导致大量的内存消耗和GC的巨大负担，造成系统缓慢甚至崩溃！

**Shuffle默认将数据持久化到磁盘中。**

**Spark中**Hash Based Shuffle Write实现：

每个Shuffle Map Task根据Key的哈希值，计算出每个Key需要写入的Partition然后将数据单独写入一个文件，这个Partition实际上就对应了下游的一个Shuffle Map Task或者Result Task。因此下游的Task在计算的时候会通过网络(如果该Task与上游的ShuffleMapTask运行在一个节点上，那么此时就是一个本地的硬盘读写。)

Hash Based Shuffle Write

1. Basic Shuffle Writer

首先，获得ShuffleManager，为了获得Shuffle方式是Hash Based Shuffle.

其次，获得HashShuffleWriter。

**val** manager = SparkEnv.*get*.shuffleManager  
writer = manager.getWriter[Any, Any](dep.*shuffleHandle*, partitionId, context)  
writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[\_ <: Product2[Any, Any]]])  
writer.stop(success = **true**).get

调用HashShuffleWriter.write方法

*/\*\** *Write a bunch of records to this task's output \*/***override def** write(records: Iterator[Product2[K, V]]): Unit = {

//判断定义了aggregator   
 **val** iter = **if** (*dep*.aggregator.isDefined) {

**if** (*dep*.mapSideCombine) {//是否需要聚合，如果需要，将聚合到records  
 *dep*.aggregator.get.combineValuesByKey(records, context)  
 } **else** {  
 records  
 }  
 } **else** {  
 *require*(!*dep*.mapSideCombine, "Map-side combine without Aggregator specified!")  
 records  
 }  
  
 **for** (elem <- iter) {  
 **val** bucketId = *dep*.partitioner.getPartition(elem.\_1)//获得element需要写入的partition

/\*

实际调用的是shuffle = shuffleBlockResolver.forMapTask(*dep*.*shuffleId*, mapId, *numOutputSplits*, *ser*,  
 *writeMetrics*)

\*/  
 *shuffle*.writers(bucketId).write(elem.\_1, elem.\_2)  
 }  
}

调用RDD开始运算 -> 通过Writer进行持久化 -> 然后将原始结果或者聚合后的结果通过FileShuffleBlockResolver#forMapTask的方法写入。

*blockManager*.getDiskWriter(blockId, tmp, *serializerInstance*, *bufferSize*, writeMetrics)

写入后 -> 将元素数据信息写入MapStatus，下游的Task就是通过这个MapStatus取得需要处理的数据。

blockManager.getDiskWriter为每个文件创建一个DiskBlockObjectWriter，DiskBlockObjectWriter可以直接向一个文件写入数据。如果文件存在，那么会以追加的方式写入。

**def** getDiskWriter(  
 blockId: BlockId,  
 file: File,  
 serializerInstance: SerializerInstance,  
 bufferSize: Int,  
 writeMetrics: ShuffleWriteMetrics): DiskBlockObjectWriter = {  
 **val** compressStream: OutputStream => OutputStream = wrapForCompression(blockId, \_)  
 **val** syncWrites = conf.getBoolean("spark.shuffle.sync", **false**)  
 **new** DiskBlockObjectWriter(file, serializerInstance, bufferSize, compressStream,  
 syncWrites, writeMetrics, blockId)

ShuffleWriterGroup：一个ShuffleMapTask对应一组writers,一个writer对应一个reducer.

*/\*\* A group of writers for a ShuffleMapTask, one writer per reducer. \*/***private**[spark] **trait** ShuffleWriterGroup {  
 **val** writers: Array[DiskBlockObjectWriter]