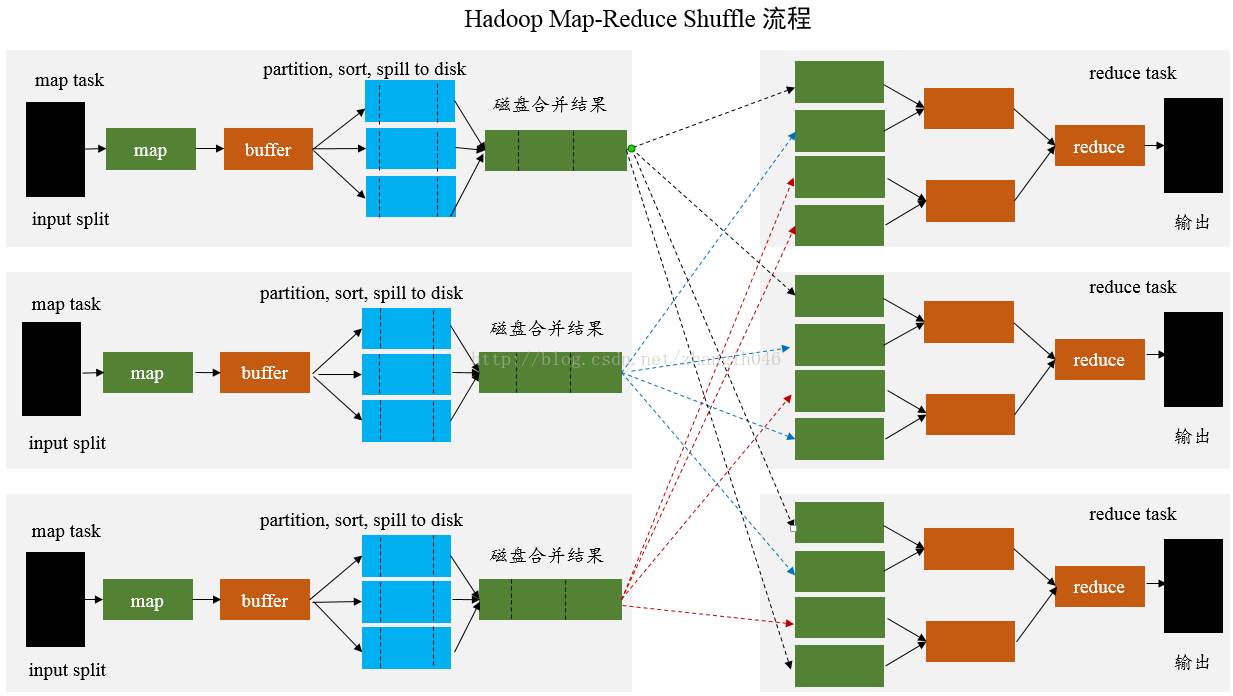
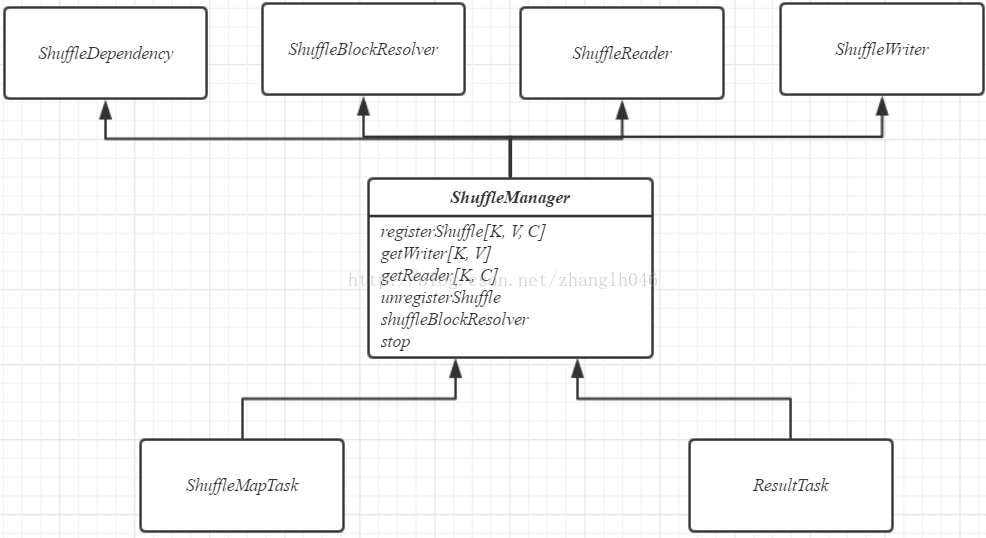
## 一 概述

Shuffle就是对数据进行重组，是把一组无规则的数据尽量转换成一组具有一定规则的数据。由于分布式计算的特性和要求，在实现细节上更加繁琐和复杂。

在MapReduce框架，Shuffle是连接Map和Reduce之间的桥梁，Map阶段通过shuffle读取数据并输出到对应的Reduce。而Reduce阶段负责从Map端拉取数据并进行计算。在整个shuffle过程中，往往伴随着大量的磁盘和网络I/O。所以shuffle性能的高低也直接决定了整个程序的性能高低。下图为Hadoop Shuffle过程。





Spark也有自己的shuffle实现过程。在DAG调度的过程中，Stage阶段的划分是根据是否有shuffle过程，也就是存在ShuffleDependency宽依赖的时候，需要进行shuffle。这时候会将作业job划分成多个Stage。并且在划分Stage并构建ShuffleDependency的时候进行shuffle注册，获取后续数据读取所需要的ShuffleHandle,最终每一个job提交后都会生成一个ResultStage和若干个ShuffleMapStage，其中ResultStage表示生成作业的最终结果所在的Stage. ResultStage与ShuffleMapStage中的task分别对应着ResultTask与ShuffleMapTask。一个作业，除了最终的ResultStage外，其他若干ShuffleMapStage中各个ShuffleMapTask都需要将最终的数据根据相应的Partitioner对数据进行分组，然后持久化分区的数据。

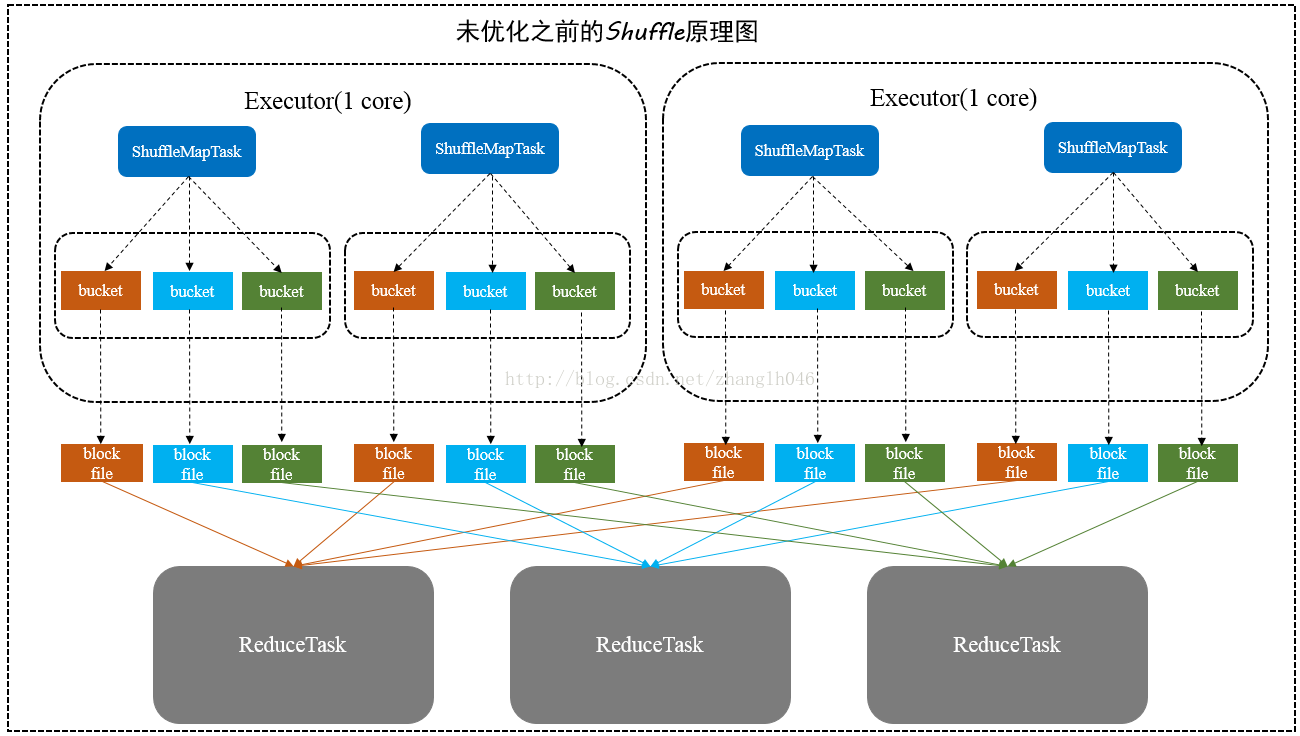
## 二 HashShuffle机制

### 2.1 HashShuffle概述

在spark-1.6版本之前，一直使用HashShuffle，在spark-1.6版本之后使用Sort-Base Shuffle，因为HashShuffle存在的不足所以就替换了HashShuffle。

我们知道，Spark的运行主要分为2部分：一部分是驱动程序，其核心是SparkContext。另一部分是Worker节点上Task,它是运行实际任务的。程序运行的时候，Driver和Executor进程相互交互，Driver会分配Task到Executor，也就是Driver 跟 Executor 会进行网络传输。另外，当前Task要抓取其他上游的 Task 的数据结果，所以这个过程中就不断的产生网络结果。其中下一个Stage向上一个Stage要数据这个过程，我们就称之为 Shuffle。

### 2.2 没有优化之前的HashShuffle机制



在HashShuffle没有优化之前，每一个ShufflleMapTask会为每一个ReduceTask创建一个bucket缓存，并且会为每一个bucket创建一个文件。这个bucket存放的数据就是经过Partitioner操作(默认是HashPartitioner)之后找到对应的bucket然后放进去，最后将数据刷新bucket缓存的数据到磁盘上，即对应的block file。

然后ShuffleMapTask将输出作为MapStatus发送到DAGScheduler的MapOutputTrackerMaster，每一个MapStatus包含了每一个ResultTask要拉取的数据的位置和大小。

接下来ResultTask去利用BlockStoreShuffleFetcher向MapOutputTrackerMaster获取MapStatus，看哪一份数据是属于自己的，然后底层通过BlockManager将数据拉取过来。

拉取过来的数据会组成一个内部的ShuffleRDD，优先放入内存，内存不够用则放入磁盘，然后ResulTask开始进行聚合，最后生成我们希望获取的那个MapPartitionRDD。

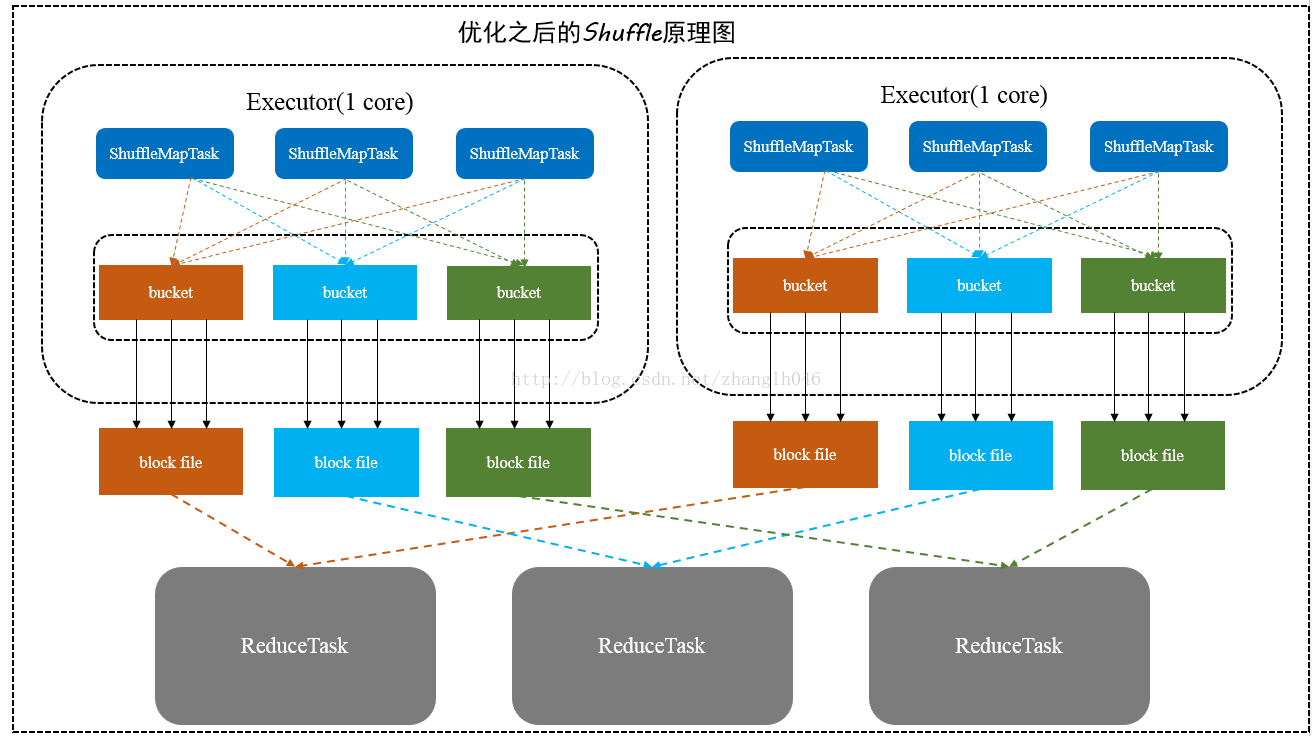
缺点：

如上图所示：在这里有1个worker，2个executor，每一个executor运行2个ShuffleMapTask，有三个ReduceTask，计算方式为：executor数量\*每个executor的ShuffleMapTask数量\*ReduceTask数量。所以总共就有2\*2\*3=12个bucket以及对应12个block file(分区文件)。

# 如果数据量较大，将会生成M\*R个小文件，比如ShuffleMapTask有100个，ResultTask有100个，这就会产生100\*100=10000个小文件

# bucket缓存很重要，需要将ShuffleMapTask所有数据都写入bucket，然后再刷到磁盘。那么如果Map端数据过多，这就很容易造成内存溢出。尽管后面有优化，bucket写入的数据达到刷新到磁盘的阀值之后，就会将数据一点一点的刷新到磁盘，但是这样磁盘I/O就多了。

### 2.3 优化后的HashShuffle



每一个Executor进程根据核数，决定Task的并发数量，比如executor核数是2，那就可以并发运行两个task，如果是一个则只能运行一个task。

假设executor核数是1，ShuffleMapTask数量是M,那么它依然会根据ResultTask的数量R，创建R个bucket缓存，然后对key进行hash，数据进入不同的bucket中，每一个bucket对应着一个block file,用于刷新bucket缓存里的数据。

然后下一个task运行的时候，就不会再创建新的bucket和block file，而是复用之前的task已经创建好的bucket和block file。即所谓同一个Executor进程里所有Task都会把相同的key放入相同的bucket缓冲区中。

这样的话，生成文件的数量就是(本地worker的所有executor对应的cores的总数\*ResultTask数量)如上图所示，即2 \* 3 = 6个文件，每一个Executor的shuffleMapTask数量100,ReduceTask数量即为100。

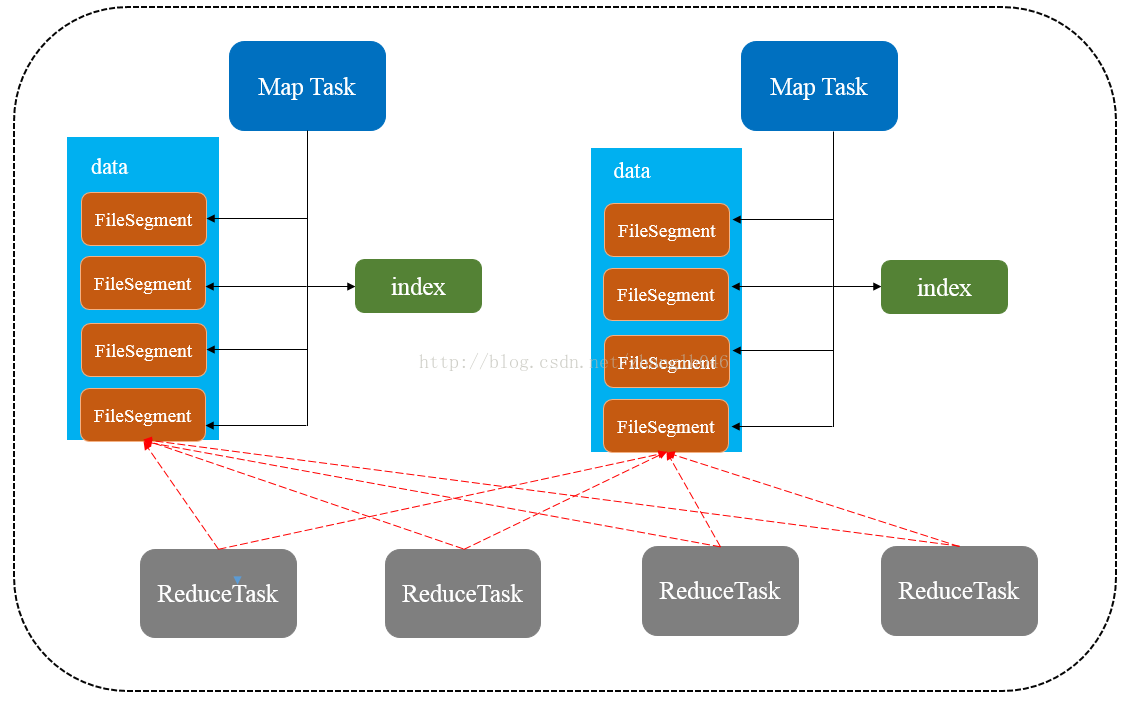
接下来举例比较一下，未优化的HashShuffle的文件数是2\*100\*100 =20000，优化之后的数量是2\*100 = 200文件，相当于少了100倍。

缺点：如果 Reducer 端的并行任务或者是数据分片过多的话则 Core \* Reducer Task 依旧过大，也会产生很多小文件。

## 三 Sort-Based Shuffle

### 3.1 Sorted-Based Shuffle介绍

为了缓解Shuffle过程产生文件数过多和Writer缓存开销过大的问题，spark引入了类似于hadoop Map-Reduce的shuffle机制。该机制每一个ShuffleMapTask不会为后续的任务创建单独的文件，而是会将所有的Task结果写入同一个文件，并且对应生成一个索引文件。以前的数据是放在内存缓存中，等到缓存读取完数据后再刷到磁盘，现在为了减少内存的使用，在内存不够用的时候，可以将输出溢写到磁盘。结束的时候，再将这些不同的文件联合内存（缓存）的数据一起进行归并，从而减少内存的使用量。一方面文件数量显著减少，另一方面减少Writer缓存所占用的内存大小，而且同时避免GC的风险和频率。



Sort-Based Shuffle有几种不同的策略：BypassMergeSortShuffleWriter（Bypass机制）、SortShuffleWriter（普通机制）和UnsafeShuffleWriter。

对于BypassMergeSortShuffleWriter，使用这个模式的特点为：

# 主要用于处理不需要排序和聚合的Shuffle操作，所以数据是直接写入文件，数据量较大的时候，网络I/O和内存负担较重。

# 主要适合处理Reducer任务数量比较少的情况。

# 将每一个分区写入一个单独的文件，最后将这些文件合并,减少文件数量。但是这种方式需要并发打开多个文件，对内存消耗比较大。

因为BypassMergeSortShuffleWriter这种方式比SortShuffleWriter更快，所以如果在Reducer数量不大，又不需要在map端聚合和排序，而且Reducer的数目小于spark.shuffle.sort.bypassMergeThreshold指定的阀值（默认200）时，就是用的是这种方式（即启用条件）。

对于SortShuffleWriter(普通机制),使用这个模式的特点为：

# 比较适合数据量很大的场景或者集群规模很大。

# 引入了外部排序器，可以支持在Map端进行本地聚合或者不聚合。

# 如果外部排序器enable了spill功能，如果内存不够，可以先将输出溢写到本地磁盘，最后将内存结果和本地磁盘的溢写文件进行合并。

另外，这个Sort-Based Shuffle跟Executor核数没有关系，即跟并发度没有关系，它是每一个ShuffleMapTask都会产生一个data文件和index文件，所谓合并也只是将该ShuffleMapTask的各个partition对应的分区文件合并到data文件而已。所以这个就需要和Hash-BasedShuffle的consolidation机制区别开来。

对于UnsafeShuffleWriter由于需要谨慎使用，我们暂不做分析。