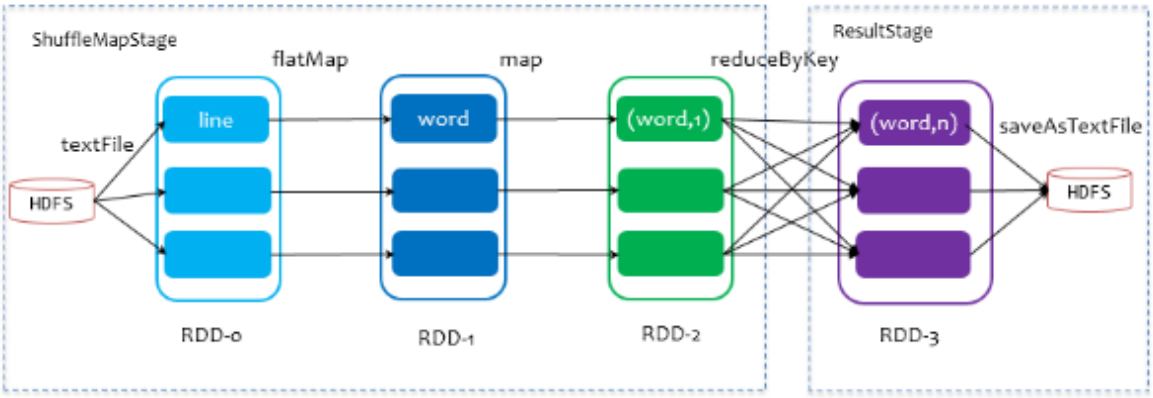
> 你要克服的是你的虚荣心，是你的炫耀欲，你要对付的是你时刻想要出风头的小聪明

SparkShuffle解析

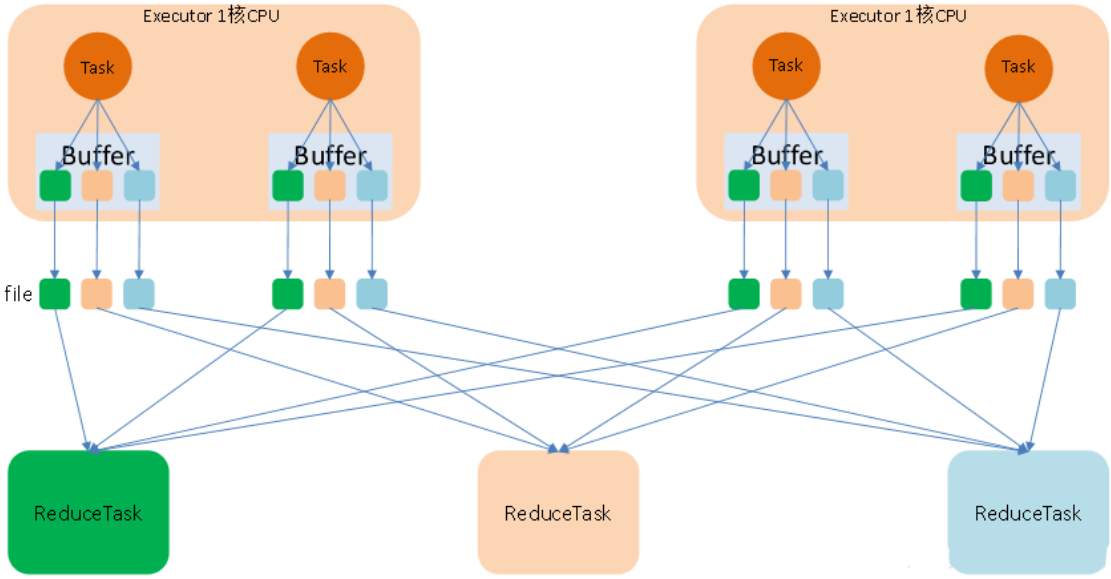
ShuffleMapStage与ResultStage



> 在划分stage时，最后一个stage称为finalStage，它本质上是一个**ResultStage对象**，前面的所有stage被称为ShuffleMapStage。ShuffleMapStage的结束伴随着shuffle文件的写磁盘。ResultStage基本上对应代码中的action算子，即将一个函数应用在RDD的各个partition的数据集上，意味着一个job的运行结束。

HashShuffle解析

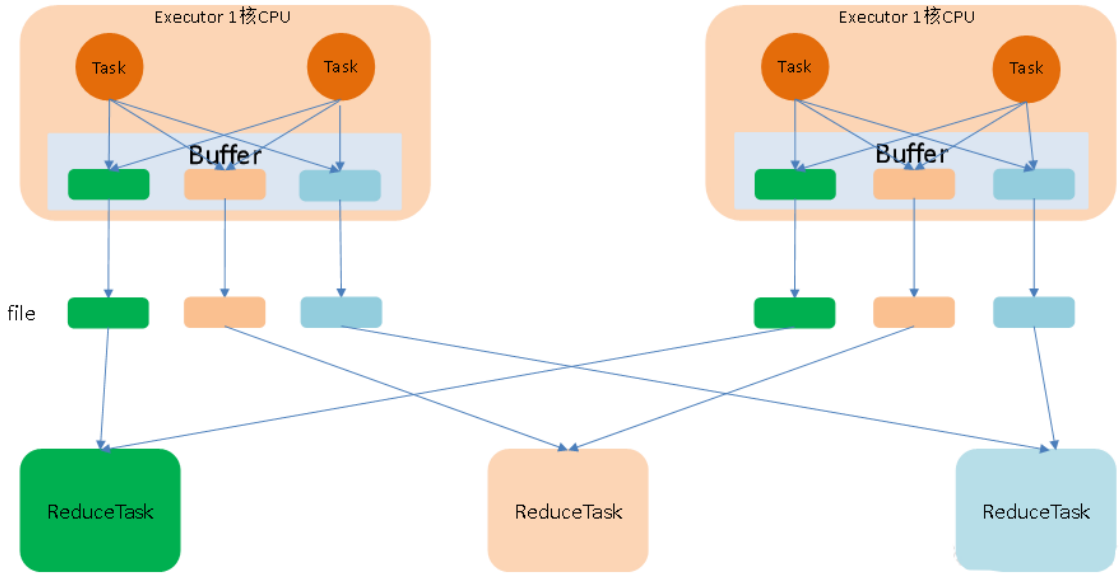
> 这里我们先明确一个假设前提：每个Executor只有1个CPU core，也就是说，无论这个Executor上分配多少个task线程，同一时间都只能执行一个task线程。
>
> 如下图中有3个 Reducer，从Task 开始那边各自把自己进行 Hash 计算(分区器：hash/numreduce取模)，分类出3个不同的类别，每个 Task 都分成3种类别的数据，想把不同的数据汇聚然后计算出最终的结果，所以Reducer 会在每个 Task 中把属于自己类别的数据收集过来，汇聚成一个同类别的大集合，每1个 Task 输出3份本地文件，这里有4个 Mapper Tasks，所以总共输出了4个 Tasks x 3个分类文件 = 12个本地小文件。



优化后的HashShuffle

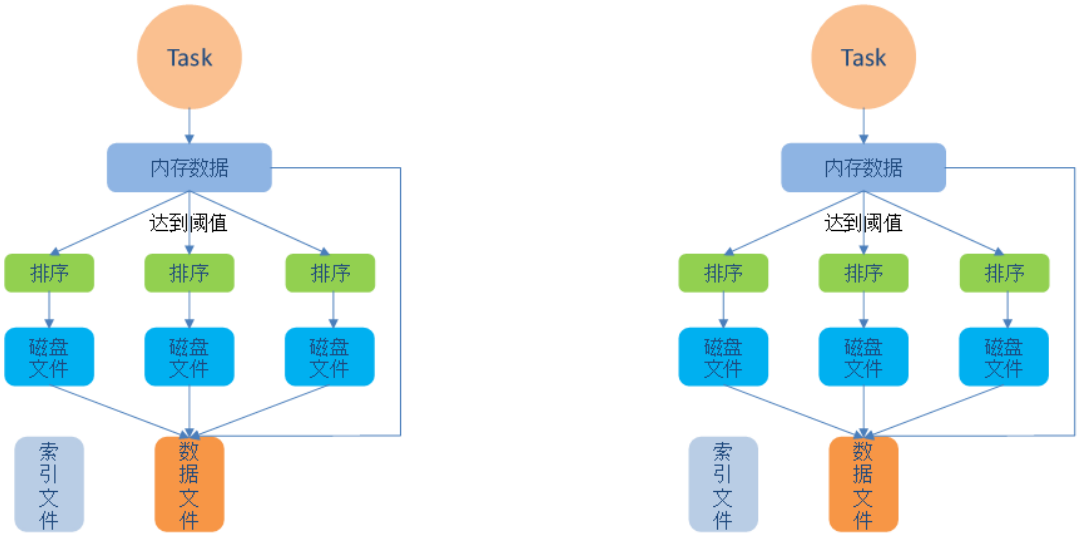
优化的HashShuffle过程就是启用合并机制，合并机制就是复用buffer，开启合并机制的配置是spark.shuffle.consolidateFiles。该参数默认值为false，将其设置为true即可开启优化机制。通常来说，如果我们使用HashShuffleManager，那么都建议开启这个选项。

这里还是有4个Tasks，数据类别还是分成3种类型，因为Hash算法会根据你的 Key 进行分类，在同一个进程中，无论是有多少过Task，都会把同样的Key放在同一个Buffer里，然后把Buffer中的数据写入以Core数量为单位的本地文件中，(一个Core只有一种类型的Key的数据)，每1个Task所在的进程中，分别写入共同进程中的3份本地文件，这里有4个Mapper Tasks，所以总共输出是 2个Cores x 3个分类文件 = 6个本地小文件。



普通SortShuffle

> 在该模式下，数据会先写入一个数据结构，reduceByKey写入Map，一边通过Map局部聚合，一遍写入内存。Join算子写入ArrayList直接写入内存中。然后需要判断是否达到阈值，如果达到就会将内存数据结构的数据写入到磁盘，清空内存数据结构。



在溢写磁盘前，先根据key进行排序，排序过后的数据，会分批写入到磁盘文件中。默认批次为10000条，数据会以每批一万条写入到磁盘文件。写入磁盘文件通过缓冲区溢写的方式，每次溢写都会产生一个磁盘文件，也就是说一个Task过程会产生多个临时文件。

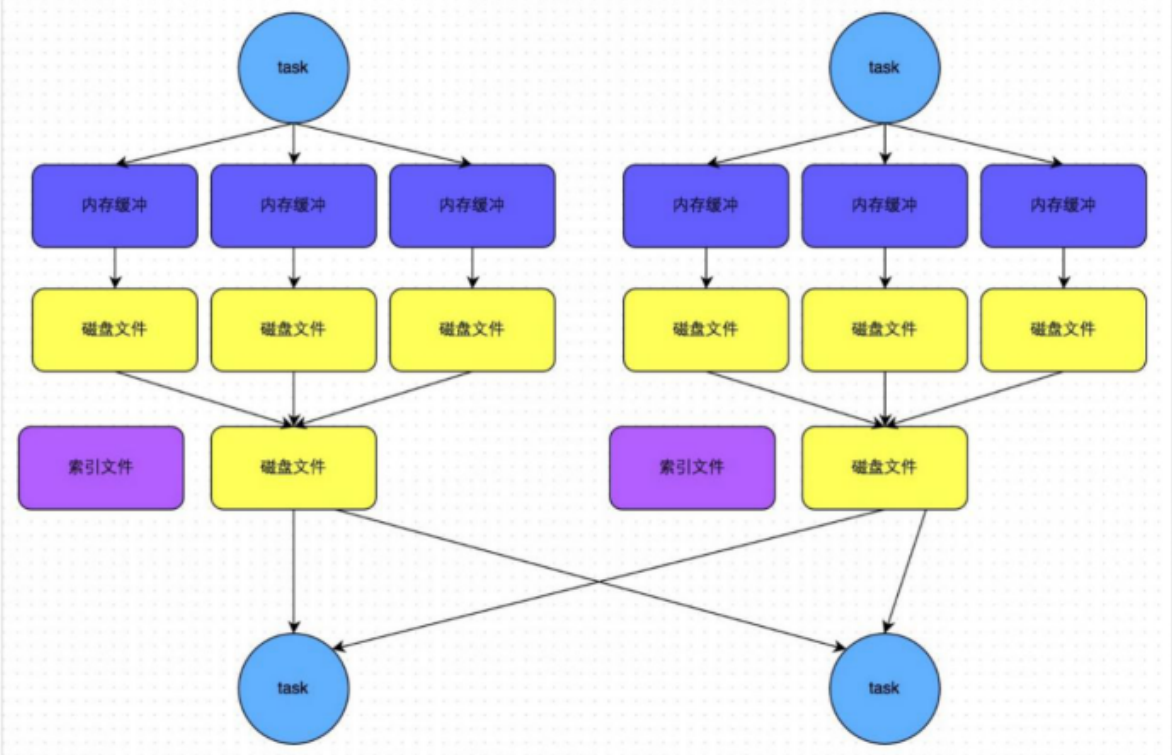
最后在每个Task中，将所有的临时文件合并，这就是merge过程，此过程将所有临时文件读取出来，一次写入到最终文件。意味着一个Task的所有数据都在这一个文件中。同时单独写一份索引文件，标识下游各个Task的数据在文件中的索引，start offset和end offset。

bypass SortShuffle

> bypass运行机制的触发条件如下：

1）shuffle map task数量小于spark.shuffle.sort.bypassMergeThreshold参数的值，默认为200。

2）不是聚合类的shuffle算子（比如reduceByKey）。



此时task会为每个reduce端的task都创建一个临时磁盘文件，并将数据按key进行hash然后根据key的hash值，将key写入对应的磁盘文件之中。当然，写入磁盘文件时也是先写入内存缓冲，缓冲写满之后再溢写到磁盘文件的。最后，同样会将所有临时磁盘文件都合并成一个磁盘文件，并创建一个单独的索引文件。

该过程的磁盘写机制其实跟未经优化的HashShuffleManager是一模一样的，因为都要创建数量惊人的磁盘文件，只是在最后会做一个磁盘文件的合并而已。因此少量的最终磁盘文件，也让该机制相对未经优化的HashShuffleManager来说，shuffle read的性能会更好。而该机制与普通SortShuffleManager运行机制的不同在于：不会进行排序。也就是说，启用该机制的最大好处在于，shuffle write过程中，不需要进行数据的排序操作，也就节省掉了这部分的性能开销。