本期内容：

1. 从数据流动视角解密WordCount，即用Spark作单词计数统计，数据到底是怎么流动的。

2. 从RDD依赖关系的视角解密WordCount。Spark中的一切操作皆RDD，后面的RDD对前面的RDD有依赖关系。

3. DAG与Lineage的思考。依赖关系会形成DAG。

1. 从数据流动视角解密WordCount

（1）在IntelliJ IDEA中编写下面代码：  
  
package com.dt.spark  
/\*\*  
 \* 使用Java的方式开发进行本地测试Spark的WordCount程序  
 \* @author DT大数据梦工厂  
 \* http://weibo.com/ilovepains  
 \*/  
import org.apache.spark.SparkConf  
import org.apache.spark.SparkContext  
object WordCount {  
  def main(args: Array[String]){  
  
  
    val conf = new SparkConf()  
    conf.setAppName("Wow, My First Spark App!")  
    conf.setMaster("local")  
    val sc = new SparkContext(conf)  
    val lines = sc.textFile("D://tmp//helloSpark.txt", 1)  
    val words = lines.flatMap { line => line.split(" ") }  
    val pairs = words.map { word => (word,1) }  
    val wordCounts = pairs.reduceByKey(\_+\_)  
    wordCounts.foreach(wordNumberPair => println(wordNumberPair.\_1 + " : " + wordNumberPair.\_2))  
    sc.stop()  
  }  
}

（2）在D盘下地tmp文件夹下新建helloSpark.txt文件，内容如下：

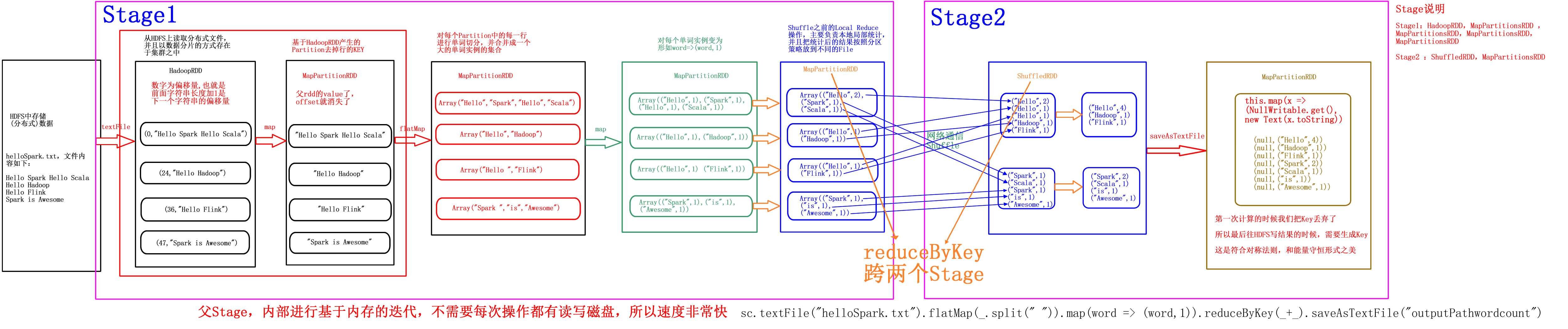
Hello Spark Hello Scala  
Hello Hadoop  
Hello Flink  
Spark is awesome

（3） 在WordCount代码区域点击右键选择Run 'WordCount'。可以得到如下运行结果：

Flink : 1  
Spark : 2  
is : 1  
Hello : 4  
awesome : 1  
Hadoop : 1

Scala : 1

下面从数据流动的视角分析数据到底是怎么被处理的。



说明：

Spark有三大特点：

1. 分布式。无论数据还是计算都是分布式的。默认分片策略：Block多大，分片就多大。但这种说法不完全准确，因为分片切分时有的记录可能跨两个Block，所以一个分片不会严格地等于Block的大小，例如HDFS的Block大小是128MB的话，分片可能多几个字节或少几个字节。一般情况下，分片都不会完全与Block大小相等。

分片不一定小于Block大小，因为如果最后一条记录跨两个Block的话，分片会把最后一条记录放在前一个分片中。

2. 基于内存（部分基于磁盘）

3. 迭代

textFile源码（SparkContext中）;

**def** textFile(

path: String,

minPartitions: Int = defaultMinPartitions): RDD[String] = withScope {

assertNotStopped()

hadoopFile(path, *classOf*[TextInputFormat], *classOf*[LongWritable], *classOf*[Text],

minPartitions).map(pair => pair.\_2.toString)

}

可以看出在进行了hadoopFile之后又进行了map操作。

HadoopRDD从HDFS上读取分布式文件，并且以数据分片的方式存在于集群之中。

map的源码（RDD.scala中）

**def** map[U: ClassTag](f: T => U): RDD[U] = withScope {

**val** cleanF = sc.clean(f)

**new** MapPartitionsRDD[U, T](**this**, (context, pid, iter) => iter.map(cleanF))

}

读取到的一行数据（key,value的方式），对行的索引位置不感兴趣，只对其value事情兴趣。pair时有个匿名函数，是个tuple，取第二个元素。

此处又产生了MapPartitionsRDD。MapPartitionsRDD基于hadoopRDD产生的Parition去掉行的KEY。

注：可以看出一个操作可能产生一个RDD也可能产生多个RDD。如sc.textFile就产生了两个RDD：hadoopRDD和MapParititionsRDD。

下一步：val words = lines.flatMap { line => line.split(" ") }

对每个Partition中的每行进行单词切分，并合并成一个大的单词实例的集合。

FlatMap做的一件事就是对RDD中的每个Partition中的每一行的内容进行单词切分。

这边有4个Partition，对单词切分就变成了一个一个单词，

下面是FlatMap的源码（RDD.scala中）

**def** flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U] = withScope {

**val** cleanF = sc.clean(f)

**new** MapPartitionsRDD[U, T](**this**, (context, pid, iter) => iter.flatMap(cleanF))

}

可以看出flatMap又产生了一个MapPartitionsRDD,

此时的各个Partition都是拆分后的单词。

下一步：    val pairs = words.map { word => (word,1) }

将每个单词实例变为形如word=>(word,1)

map操作就是把切分后的每个单词计数为1。

根据源码可知，map操作又会产生一个MapPartitonsRDD。此时的MapPartitionsRDD是把每个单词变成Array(""Hello",1),("Spark",1)等这样的形式。

下一步：val wordCounts = pairs.reduceByKey(\_+\_)

reduceByKey是进行全局单词计数统计，对相同的key的value相加，包括local和reducer同时进行reduce。所以在map之后，本地又进行了一次统计，即local级别的reduce。

shuffle前的Local Reduce操作，主要负责本地局部统计，并且把统计后的结果按照分区策略放到不同的File。

下一Stage就叫Reducer了，下一阶段假设有3个并行度的话，每个Partition进行Local Reduce后都会把数据分成三种类型。最简单的方式就是用HashCode对其取模。

至此都是stage1。

Stage内部完全基于内存迭代，不需要每次操作都有读写磁盘，所以速度非常快。

reduceByKey的源码：

**def** reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)] = self.withScope {

combineByKeyWithClassTag[V]((v: V) => v, func, func, partitioner)

}

*/\*\**

*\* Merge the values for each key using an associative reduce function. This will also perform*

*\* the merging locally on each mapper before sending results to a reducer, similarly to a*

*\* "combiner" in MapReduce. Output will be hash-partitioned with numPartitions partitions.*

*\*/*

**def** reduceByKey(func: (V, V) => V, numPartitions: Int): RDD[(K, V)] = self.withScope {

reduceByKey(**new** HashPartitioner(numPartitions), func)

}

可以看到reduceByKey内部有combineByKeyWithClassTag。combineByKeyWithClassTag的源码如下：

**def** combineByKeyWithClassTag[C](

createCombiner: V => C,

mergeValue: (C, V) => C,

mergeCombiners: (C, C) => C,

partitioner: Partitioner,

mapSideCombine: Boolean = **true**,

serializer: Serializer = **null**)(**implicit** ct: ClassTag[C]): RDD[(K, C)] = self.withScope {

*require*(mergeCombiners != **null**, "mergeCombiners must be defined") // required as of Spark 0.9.0

**if** (keyClass.isArray) {

**if** (mapSideCombine) {

**throw new** SparkException("Cannot use map-side combining with array keys.")

}

**if** (partitioner.isInstanceOf[HashPartitioner]) {

**throw new** SparkException("Default partitioner cannot partition array keys.")

}

}

**val** aggregator = **new** Aggregator[K, V, C](

self.context.clean(createCombiner),

self.context.clean(mergeValue),

self.context.clean(mergeCombiners))

**if** (self.*partitioner* == *Some*(partitioner)) {

self.mapPartitions(iter => {

**val** context = TaskContext.*get*()

**new** InterruptibleIterator(context, aggregator.combineValuesByKey(iter, context))

}, preservesPartitioning = **true**)

} **else** {

**new** ShuffledRDD[K, V, C](self, partitioner)

.setSerializer(serializer)

.setAggregator(aggregator)

.setMapSideCombine(mapSideCombine)

}

}

可以看出在combineByKeyWithClassTag内又new 了一个ShuffledRDD。

ReduceByKey有两个作用：

1. 进行Local级别的Reduce，减少网络传输。

2. 把当前阶段的内容放到本地磁盘上供shuffle使用。

下一步是shuffledRDD,

产生Shuffle数据就需要进行分类，MapPartitionsRDD时其实已经分好类了，最简单的分类策略就是Hash分类。

ShuffledRDD需要从每台机上抓取同一单词。

reduceByKey发生在哪里？

Stage2全部都是reduceByKey

最后一步：保存数据到HDFS（MapPartitionsRDD）

统计完的结果：（“Hello”，4）只是一个Value，而不是Key:"Hello",value:4。但输出到文件系统时需要KV的格式，现在只有Value，所以需要造个KEY。

saveAsTextFile的源码：

def saveAsTextFile(path: String){

this.map(x => (NullWritable.get())),new Text(x.toStirng))

.saveAsHadoopFile[TextOutputFormat[NullWritable,Text]](path)

}

this.map把当前的值（x）变成tuple。tuple的Key是Null，Value是（“Hello”，4）。

为什么要为样？因为saveAsHadoopFile时要求以这样的格式输出。Hadoop需要KV的格式！！

map操作时把key舍去了，输出时就需要通过生成Key。

第一个Stage有哪些RDD？HadoopRDD、MapPartitionsRDD、MapPartitionsRDD、MapPartitionsRDD、MapPartitionsRDD

第二个Stage有哪些RDD？ShuffledRDD、MapPartitionsRDD

只有Collect 或saveAsTextFile会触发作业，其他的时候都没有触发作业（Lazy）