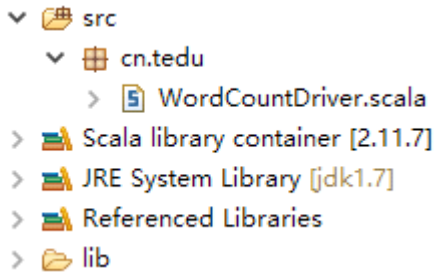
案例—WordCount

2018年3月1日 21:15

实现步骤

1）创建spark的项目

在scala中创建项目 导入spark相关的jar包



2）开发spark相关代码

代码示例：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object WordCountDriver {

  def main(args: Array[String]): Unit = {

    val conf=new SparkConf().setMaster("spark://hadoop01:7077").setAppName("wordcount")
    val sc=new SparkContext(conf)

    val data=sc.textFile("hdfs://hadoop01:9000/words.txt", 2)
    val result=data.flatMap { x => x.split(" ") }.map { x => (x,1) }.reduceByKey(_+_)

    result.saveAsTextFile("hdfs://hadoop01:9000/wcresult")
  }
}
```

3）将写好的项目打成jar，上传到服务器，进入bin目录

执行：spark-submit --class cn.tedu.WordCountDriver /home/software/spark/conf/wc.jar

案例—求平均值

2018年3月2日 17:43

案例文件：

1 16

2 74

3 51

4 35

5 44

6 95

7 5

8 29

10 60

11 13

12 99

13 7

14 26

正确答案：42
①只拿第二列，形成RDD
②类型转换->String->Int
③和/个数

代码示例一：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object AverageDriver {

  def main(args: Array[String]): Unit = {
    val conf=new SparkConf().setMaster("local").setAppName("AverageDriver")

    val sc=new SparkContext(conf)
```

```
    val data=sc.textFile("d://average.txt")

    val ageData=data.map { line=>{line.split(" ")(1).toInt}}

    val ageSum=ageData.reduce(_+_)

    val pepopleCount=data.count()

    val average=ageSum/pepopleCount

    println(average)
  }
}
```

代码示例二：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object AverageDriver {

  def main(args: Array[String]): Unit = {
    val conf=new SparkConf().setMaster("local").setAppName("AverageDriver")

    val sc=new SparkContext(conf)

    val data=sc.textFile("d://average.txt",3)

    val ageData=data.map { line=>{line.split(" ")(1).toInt}}


    val ageSum=ageData.mapPartitions{it=>{
      val result=List[Int]()
      var i=0
      while(it.hasNext){
        i+=it.next()
      }
      result.::(i).iterator
```

```
    }}.reduce(_+_)


    val pepopleCount=data.count()

    val average=ageSum/pepopleCount

    println(average)
  }
}
```

案例—求最大值和最小值

2018年3月2日 17:43

案例文件：

1 M 174

2 F 165

3 M 172

4 M 180

5 F 160

6 F 162

7 M 172

8 M 191

9 F 175

10 F 167

代码示例一：

```
package cn.tedu

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object MaxMinDriver {

 def main(args: Array[String]): Unit = {

   val conf=new SparkConf().setMaster("local").setAppName("MaxMin")
   val sc=new SparkContext(conf)

   val data=sc.textFile("d://MaxMin.txt")


   val manData=data.filter { x => x.contains("M") }.map {  x => x.split(" ")(2).toInt}
   val girlData=data.filter { x => x.contains("F") }.map {  x => x.split(" ")(2).toInt}

   println("Man Max is:"+manData.max()+"Man min is:"+manData.min())
```

```
 }
}
```

代码示例二：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object MaxMinDriver {

 def main(args: Array[String]): Unit = {

  val conf=new SparkConf().setMaster("local").setAppName("MaxMin")
  val sc=new SparkContext(conf)

  val data=sc.textFile("d://MaxMin.txt")

  val manMax=data.filter { line => line.contains("M") }.
   sortBy({line=>line.split(" ")(2)},false).first().mkString

   val manMin=data.filter { line => line.contains("M") }.
   sortBy({line=>line.split(" ")(2)},true).first.mkString

  println(manMax+"\n"+manMin)

 }
}
```

代码示例三：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object MaxMinDriver {

 def main(args: Array[String]): Unit = {

  val conf=new SparkConf().setMaster("spark://hadoop01:7077").setAppName("MaxMin")
  val sc=new SparkContext(conf)
```

```
    val data=sc.textFile("hdfs://hadoop01:9000/MaxMin.txt",3)

    val manMax=data.filter { line => line.contains("M") }.
      sortBy({line=>line.split(" ")(2)},false).first.mkString

    val manMin=data.filter { line => line.contains("M") }.
      sortBy({line=>line.split(" ")(2)},true).first.mkString

    val result=sc.makeRDD(Array(manMax,manMin))

    //--spark输出文件时，默认是有几个Task,就会生成几个结果文件，
    //--所以如果想控制文件个数，控制分区数(task)即可
    result.coalesce(1,true).saveAsTextFile("hdfs://hadoop01:9000/MaxMinResult")


  }
}
```

案例—TopK

2018年3月1日 21:15

案例说明

Top K算法有两步，一是统计词频，二是找出词频最高的前K个词。

文件数据 topk.txt:

hello world bye world

hello hadoop bye hadoop

hello world java web

hadoop scala java hive

hadoop hive redis hbase

hello hbase java redis

代码示例:

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object TopkDriver {

 def main(args: Array[String]): Unit = {
    val conf =new SparkConf().setMaster("local").setAppName("topk")
    val sc=new SparkContext(conf)

    val data=sc.textFile("e://topk.txt",2)

    val count=data.flatMap { x => x.split(" ") }
     .map { x => (x,1) }.reduceByKey(_+_)

    val orderingDesc = Ordering.by [(String, Int), Int](_._2)

    val topk=count.top(3)(orderingDesc)

    //val topk=count.top(3)(Ordering.by{case (word,count)=>count})

    topk.foreach{println}
```

```
 }
}
```

应用场景

Top K的示例模型可以应用在求过去一段时间消费次数最多的消费者、访问最频繁的IP地址和最近、更新、最频繁的微博等应用场景。

案例—求中位数

2018年3月1日 23:07

文件数据 median.txt:

1 20 8 2 5 11 29 10

7 4 45 6 23 17 19

一共是15个数，正确答案是10

代码示例：

```
object Driver {

  def main(args: Array[String]): Unit = {
    val conf=new SparkConf().setMaster("local").setAppName("median")
    val sc=new SparkContext(conf)
    val data=sc.textFile("d://data/median.txt")

    val count=data.flatMap{_.split(" ")}.count()
    val medianIndex=(count+1)/2

    val median=data.flatMap { _.split(" ") }
      .map{x=>x.toInt}.sortBy(x=>x).take(medianIndex.toInt).last

    println(median)
  }
}
```

代码示例：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext


object MedianDriver {


  def main(args: Array[String]): Unit = {
    val conf=new SparkConf().setMaster("local").setAppName("Median")
    val sc=new SparkContext(conf)
    val data=sc.textFile("e://median.txt",2)


    val nums=data.flatMap { x => x.split(" ") }


    val sortResult=nums.map { x => (x.toInt,1) }.sortByKey()
```

```
    val totalNum=sortResult.mapPartitions{x=>

      val list=List[Int]()
      list.::(x.size).iterator
    }.sum().toInt

    val medianIndex=(totalNum+1)/(2)

    val medianResult=sortResult.takeOrdered(medianIndex).last._1
    //val medianResult=sortResult.top(medianIndex)(Ordering.by [(Int, Int), Int](-_._1)).last._1

    println(medianResult)



  }
}
```

案例—二次排序

2018年3月2日 19:06

文件数据：

aa 12

bb 32

aa 3

cc 43

dd 23

cc 5

cc 8

bb 33

bb 12

要求：先按第一例升序排序，再按第二列降序排序

自定义排序类代码：

```
class SecondarySortKey(val first:String,val second:Int) extends Ordered[SecondarySortKey] with Serializable {

 def compare(other:SecondarySortKey):Int={
  var comp=this.first.compareTo(other.first)
  if(comp==0){
   other.second.compareTo(this.second)
  }else{
   comp
  }
 }
}
```

Driver代码：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
```

```
object SsortDriver {

  def main(args: Array[String]): Unit = {

    val conf=new SparkConf().setMaster("local").setAppName("ssort")
    val sc=new SparkContext(conf)
    val data=sc.textFile("d://ssort.txt",3)

    val ssortData=data.map { line =>{
    (new SecondarySortKey(line.split(" ")(0),line.split(" ")(1).toInt),line)
    }
    }

    val result=ssortData.sortByKey(true)

    result.foreach(println)


  }
}
```

案例—倒排索引

2018年3月2日 20:55

文件代码：

doc1.txt:

hello spark

hello hadoop

doc2.txt:

hello hive

hello hbase

hello spark

doc3.txt:

hadoop hbase

hive scala

最后的结果形式为：

```
(scala,doc3)
(spark,doc1,doc2)
(hive,doc2,doc3)
(hadoop,doc1,doc3)
(hello,doc1,doc2)
(hbase,doc2,doc3)
```

代码：

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext

object Driver {

 def main(args: Array[String]): Unit = {
   val conf=new SparkConf().setMaster("local").setAppName("inverted")
   val sc=new SparkContext(conf)

   //--读取指定目录下所有的文件，并返回到一个RDD中
   //--(filepath,filetext)
   val data=sc.wholeTextFiles("d://data/inverted/*")
```

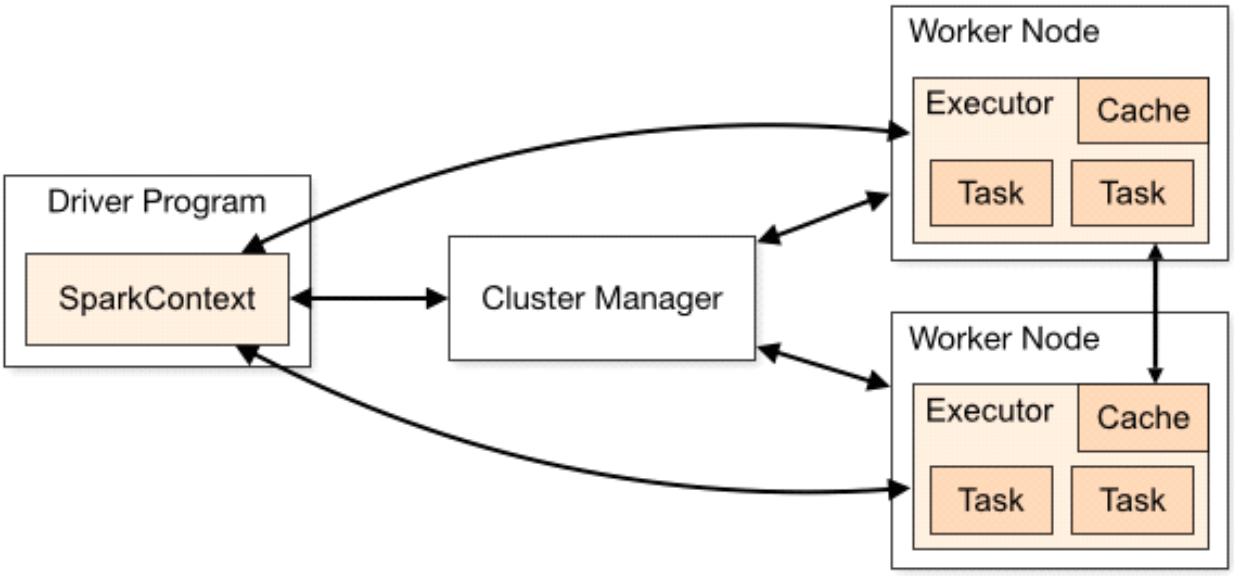
```
    val clearData=data.map{case(filepath,filetext)=>
      val filename=filepath.split("/").last.dropRight(4)
      (filename,filetext)
    }

   //--(hello,doc1) (hello,doc2) (hadoop,doc1)……
   //--(hello,List(doc1,doc2))

   val resultData=clearData.flatMap{case(filename,filetext)=>
     filetext.split("\r\n").flatMap { x => x.split(" ") }.map { x =>(x,filename) }
   }

   val result=resultData.groupByKey.map{case(word,buffer)=>
(word,buffer.toList.distinct.mkString(","))}

   result.foreach(println)
  }
}
```

Spark架构

2018年2月13日 13:16

概述

为了更好地理解调度，我们先来鸟瞰一下集群模式下的Spark程序运行架构图。



1. Driver Program

用户编写的Spark程序称为Driver Program。每个Driver程序包含一个代表集群环境的SparkContext对象，程序的执行从Driver程序开始，所有操作执行结束后回到Driver程序中，在Driver程序中结束。如果你是用spark shell，那么当你启动 Spark shell的时候，系统后台自启了一个 Spark 驱动器程序，就是在Spark shell 中预加载的一个叫作 sc 的SparkContext 对象。如果驱动器程序终止，那么Spark 应用也就结束了。

2. SparkContext对象

每个Driver Program里都有一个SparkContext对象，职责如下：

1）SparkContext对象联系 cluster manager（集群管理器），让 cluster manager 为Worker Node分配CPU、内存等资源。此外， cluster manager会在 Worker Node 上启动

一个执行器（专属于本驱动程序）。

2）和Executor进程交互，负责任务的调度分配。

3. cluster manager 集群管理器

它对应的是Master进程。集群管理器负责集群的**资源调度**，比如为Worker Node分配CPU、内存等资源。并实时监控Worker的资源使用情况。一个Worker Node默认情况下分配一个Executor（进程）。

从图中可以看到sc和Executor之间画了一根线条，这表明：程序运行时，sc是直接与Executor进行交互的。

所以，cluster manager **只是负责资源的管理调度，而任务的分配和结果处理它不管。**

4.Worker Node

Worker节点。集群上的计算节点，对应一台物理机器

5.Worker进程

它对应Worder进程，用于和Master进程交互，向Master注册和汇报自身节点的资源使用情况，并管理和启动Executor进程

6.Executor

负责运行Task计算任务，并将计算结果回传到Driver中。

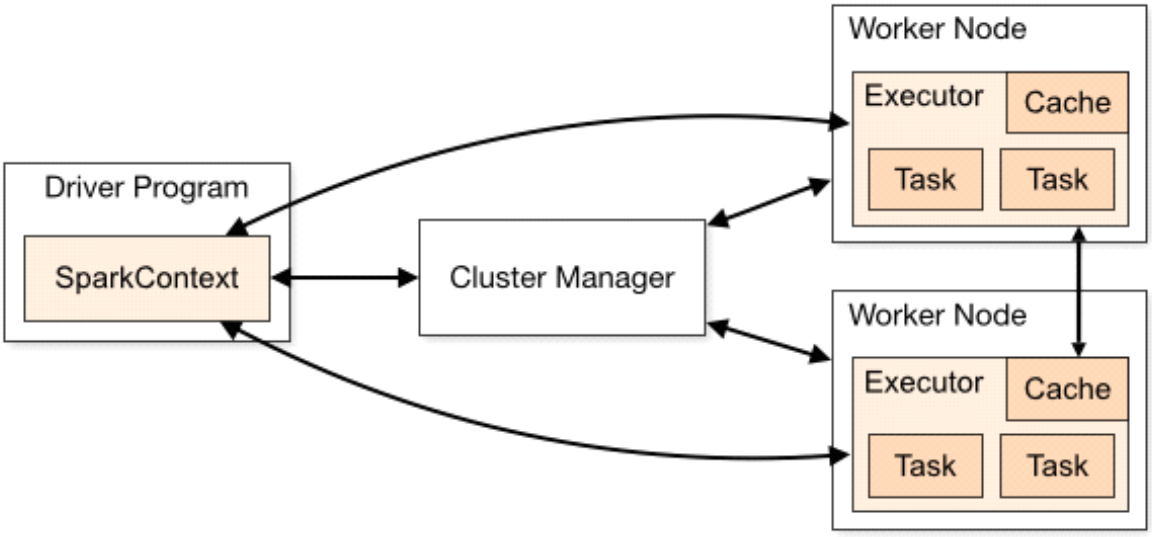
7.Task

在执行器上执行的最小单元。比如RDD Transformation操作时对RDD内每个分区的计算都会对应一个Task。

Spark调度模块

2018年3月1日 12:46

概述

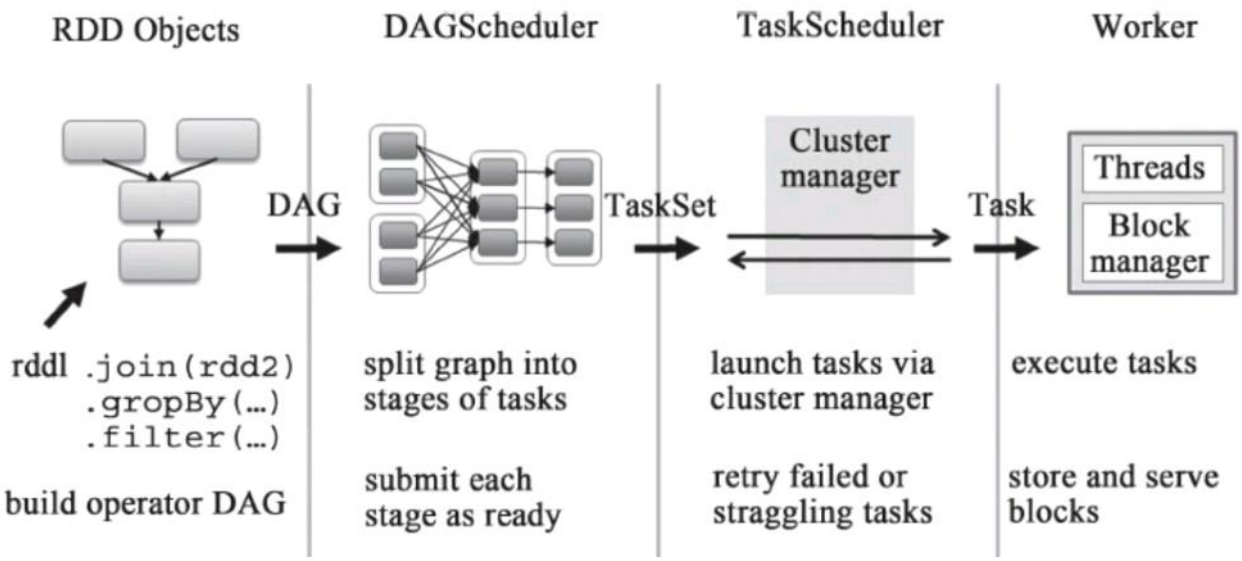


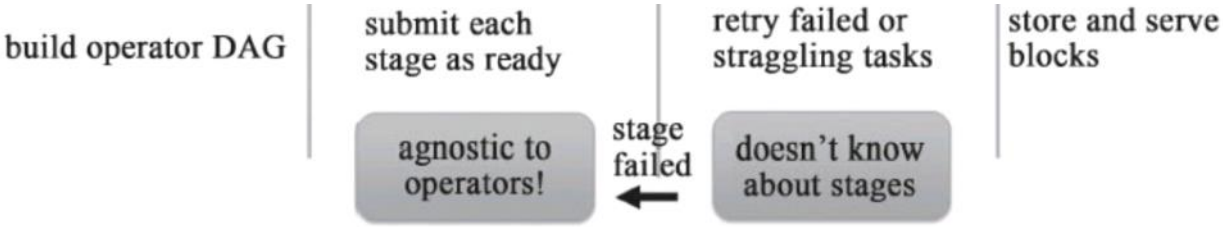
之前我们提到：Driver 的sc负责和Executor交互，完成任务的分配和调度，在底层，任务调度模块主要包含两大部分：

1）DAGScheduler

2）TaskScheduler

它们负责将用户提交的计算任务按照DAG划分为不同的阶段并且将不同阶段的计算任务提交到集群进行最终的计算。整个过程可以使用下图表示



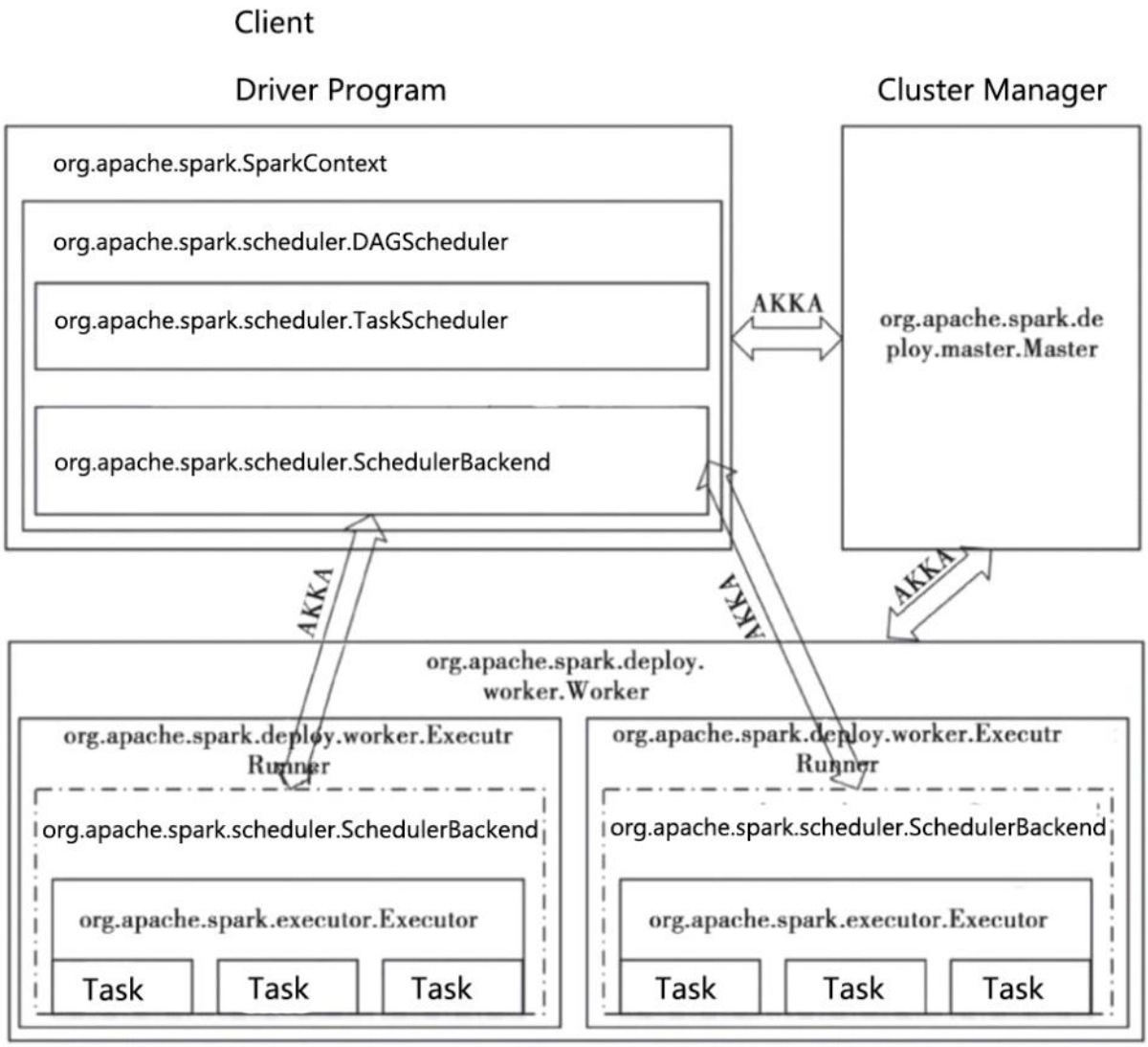


RDD Objects可以理解为用户实际代码中创建的RDD，这些代码逻辑上组成了一个DAG。

DAGScheduler主要负责分析依赖关系，然后将DAG划分为不同的Stage（阶段），其中每个Stage由可以并发执行的一组Task构成，这些Task的执行逻辑完全相同，只是作用于不同的数据。

在DAGScheduler将这组Task划分完成后，会将这组Task提交到TaskScheduler。TaskScheduler通过Cluster Manager 申请计算资源，比如在集群中的某个Worker Node上启动专属的Executor，并分配CPU、内存等资源。接下来，就是在Executor中运行Task任务，如果缓存中没有计算结果，那么就需要开始计算，同时，计算的结果会回传到Driver或者保存在本地。

Scheduler的实现概述



任务调度模块涉及的最重要的三个类是：

1）org.apache.spark.scheduler.DAGScheduler 前面提到的DAGScheduler的实现。

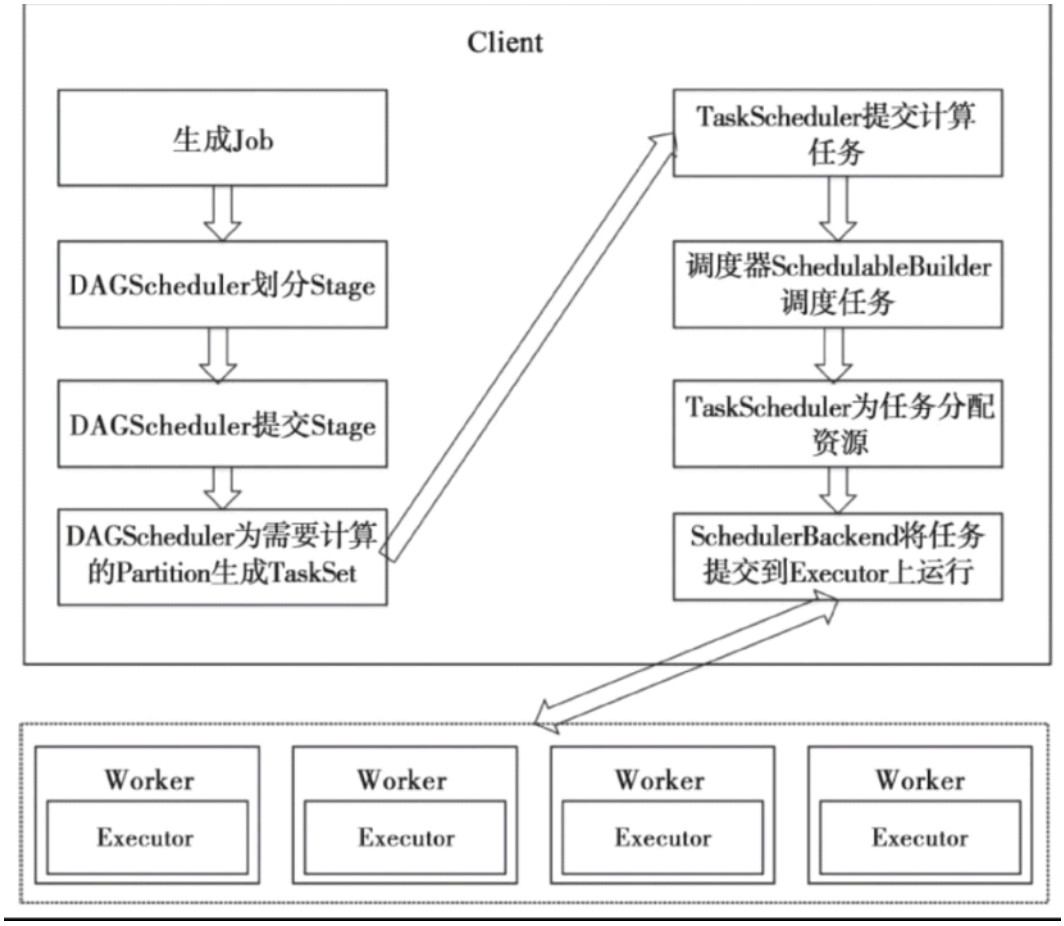
2）org.apache.spark.scheduler.TaskScheduler

它的作用是为创建它的SparkContext调度任务，即从DAGScheduler接收不同Stage的任务，并且向集群提交这些任务，并为执行特别慢的任务启动备份任务。

3）org.apache.spark.scheduler.SchedulerBackend

是一个trait，作用是分配当前可用的资源，具体就是向当前等待分配计算资源的Task分配计算资源（即Executor），并且在分配的Executor上启动Task，完成计算的调度过程。

任务调度流程图


Client
生成Job
DAGScheduler划分Stage
DAGScheduler提交Stage
DAGScheduler为需要计算的Partition生成TaskSet
TaskScheduler提交计算任务
调度器SchedulableBuilder调度任务
TaskScheduler为任务分配资源
SchedulerBackend将任务提交到Executor上运行
Worker
Executor
Worker
Executor
Worker
Executor
Worker
Executor

Spark共享变量

2018年2月13日 13:22

概述

Spark程序的大部分操作都是RDD操作，通过传入函数给RDD操作函数来计算。这些函数在不同的节点上并发执行，但每个内部的变量有不同的作用域，不能相互访问，所以有时会不太方便，Spark提供了两类共享变量供编程使用——广播变量和计数器。

1. 广播变量

这是一个只读对象，在所有节点上都有一份缓存，创建方法是SparkContext.broadcast()，比如：

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
scala> broadcastVar.value
res0: Array[Int] = Array(1, 2, 3)
```

注意，广播变量是只读的，所以创建之后再更新它的值是没有意义的，一般用val修饰符来定义广播变量。

2. 计数器

计数器只能增加，是共享变量，用于计数或求和。

计数器变量的创建方法是SparkContext.accumulator(v, name)，其中v是初始值，name是名称。

示例如下：

```
scala> val accum = sc.accumulator(0, "My Accumulator")
accum: org.apache.spark.Accumulator[Int] = 0
```

```
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)

scala> accum.value

res1: Int = 10
```

RDD容错机制

2018年2月13日 13:23

概述

分布式系统通常在一个机器集群上运行，同时运行的几百台机器中某些出问题的概率大大增加，所以容错设计是分布式系统的一个重要能力。

Spark以前的集群容错处理模型，像MapReduce，将计算转换为一个有向无环图（DAG）的任务集合，这样可以通过重复执行DAG里的一部分任务来完成容错恢复。但是由于主要的数据存储在分布式文件系统中，没有提供其他存储的概念，容错过程需要在网络上进行数据复制，从而增加了大量的消耗。所以，分布式编程中经常需要做检查点，即将某个时机的中间数据写到存储（通常是分布式文件系统）中。

RDD也是一个DAG，每一个RDD都会记住创建该数据集需要哪些操作，跟踪记录RDD的继承关系，这个关系在Spark里面叫**lineage（血缘关系）**。当一个RDD的某个分区丢失时，RDD是有足够的信息记录其如何通过其他RDD进行计算，且只需重新计算该分区，这是Spark的一个创新。

RDD的缓存

2018年2月22日 13:11

概述

相比Hadoop MapReduce来说，Spark计算具有巨大的性能优势，其中很大一部分原因是Spark对于内存的充分利用，以及提供的缓存机制。

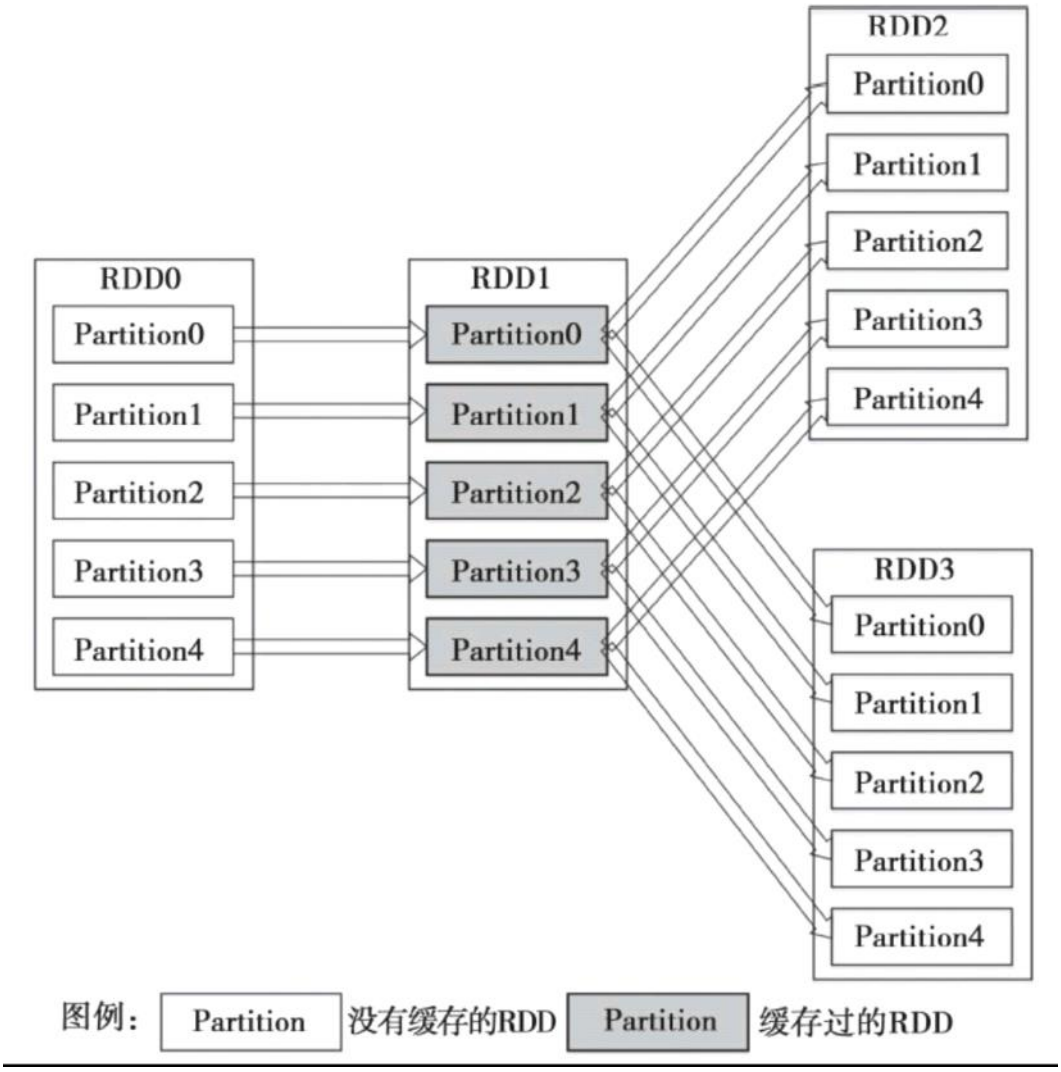
RDD持久化（缓存）

持久化在早期被称作缓存（cache），但缓存一般指将内容放在内存中。虽然持久化操作在绝大部分情况下都是将RDD缓存在内存中，但一般都会在内存不够时用磁盘顶上去（比操作系统默认的磁盘交换性能高很多）。当然，也可以选择不使用内存，而是仅仅保存到磁盘中。所以，现在Spark使用持久化（persistence）这一更广泛的名称。

如果一个RDD不止一次被用到，那么就可以持久化它，这样可以大幅提升程序的性能，甚至达10倍以上。

默认情况下，RDD只使用一次，用完即扔，再次使用时需要重新计算得到，而持久化操作**避免了这里的重复计算**，实际测试也显示持久化对性能提升明显，**这也是Spark刚出现时被人称为内存计算框架的原因。**

假设首先进行了RDD0→RDD1→RDD2的计算作业，那么计算结束时，RDD1就已经缓存在系统中了。在进行RDD0→RDD1→RDD3的计算作业时，由于RDD1已经缓存在系统中，因此RDD0→RDD1的转换不会重复进行，计算作业只须进行RDD1→RDD3的计算就可以了，因此计算速度可以得到很大提升。



持久化的方法是调用persist()函数，除了持久化至内存中，还可以在persist()中指定

storage level参数使用其他的类型，具体如下：

1) MEMORY_ONLY ： 将 RDD 以反序列化的 Java 对象的形式存储在 JVM 中. 如果内存空间不够，部分数据分区将不会被缓存，在每次需要用到这些数据时重新进行计算. 这是默认的级别。

cache()方法对应的级别就是**MEMORY_ONLY级别**

2) MEMORY_AND_DISK：将 RDD 以反序列化的 Java 对象的形式存储在 JVM 中。如果内存空间不够，将未缓存的数据分区存储到磁盘，在需要使用这些分区时从磁盘读取。

3) MEMORY_ONLY_SER ：将 RDD 以序列化的 Java 对象的形式进行存储（每个分区为一个 byte 数组）。这种方式会比反序列化对象的方式节省很多空间，尤其是在使用 fast

serialize时会节省更多的空间，但是在读取时会使得 CPU 的 read 变得更加密集。如果内存空间不够，部分数据分区将不会被缓存，在每次需要用到这些数据时重新进行计算。

4) MEMORY_AND_DISK_SER ：类似于 MEMORY_ONLY_SER ，但是溢出的分区会存储到磁盘，而不是在用到它们时重新计算。如果内存空间不够，将未缓存的数据分区存储到磁盘，在需要使用这些分区时从磁盘读取。

5) DISK_ONLY：只在磁盘上缓存 RDD。

6) MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc. ：与上面的级别功能相同，只不过每个分区在集群中两个节点上建立副本。

7 ）**OFF_HEAP** 将数据存储在 off-heap memory 中。使用堆外内存，这是Java虚拟机里面的概念，堆外内存意味着把内存对象分配在Java虚拟机的堆以外的内存，这些内存直接受操作系统管理（而不是虚拟机）。注意，可能带来一些GC回收问题。

Spark 也会自动持久化一些在 shuffle 操作过程中产生的临时数据（比如 reduceByKey），即便是用户并没有调用持久化的方法。这样做可以避免当 shuffle 阶段时如果一个节点挂掉了就得重新计算整个数据的问题。如果用户打算多次重复使用这些数据，我们仍然建议用户自己调用持久化方法对数据进行持久化。

使用缓存

```
scala> import org.apache.spark.storage._

scala> val rdd1=sc.makeRDD(1 to 5)

scala> rdd1.cache  //cache只有一种默认的缓存级别，即MEMORY_ONLY

scala> rdd1.persist(StorageLevel.MEMORY_ONLY)
```

缓存数据的清除

Spark 会自动监控每个节点上的缓存数据，然后使用 least-recently-used (LRU) 机制来处理旧的缓存数据。如果你想手动清理这些缓存的 RDD 数据而不是去等待它们被自动清理掉，可以使用 RDD.unpersist( ) 方法。

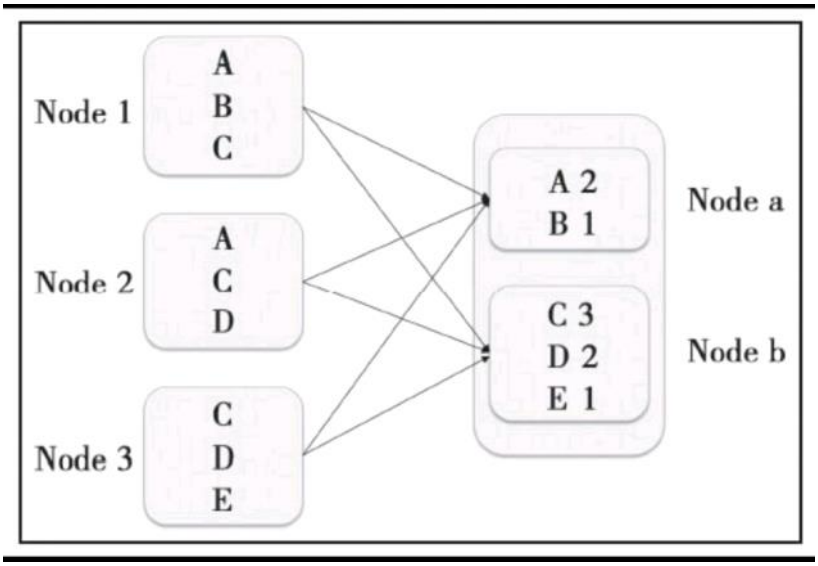
Spark Shuffle详解

2018年3月1日 14:36

概述

Shuffle，翻译成中文就是洗牌。之所以需要Shuffle，还是**因为具有某种共同特征的一类数据需要最终汇聚（aggregate）到一个计算节点上进行计算。**这些数据分布在各个存储节点上并且由不同节点的计算单元处理。以最简单的Word Count为例，其中数据保存在Node1、Node2和Node3；

经过处理后，这些数据最终会汇聚到Nodea、Nodeb处理，如下图所示。



这个数据重新打乱然后汇聚到不同节点的过程就是Shuffle。但是实际上，Shuffle过程可能会非常复杂：

1）数据量会很大，比如单位为TB或PB的数据分散到几百甚至数千、数万台机器上。

2）为了将这个数据汇聚到正确的节点，需要将这些数据放入正确的Partition，因为数据大小已经大于节点的内存，因此这个过程中可能会发生多次硬盘续写。

3）为了节省带宽，这个数据可能需要压缩，如何在压缩率和压缩解压时间中间

做一个比较好的选择？

4）数据需要通过网络传输，因此数据的序列化和反序列化也变得相对复杂。

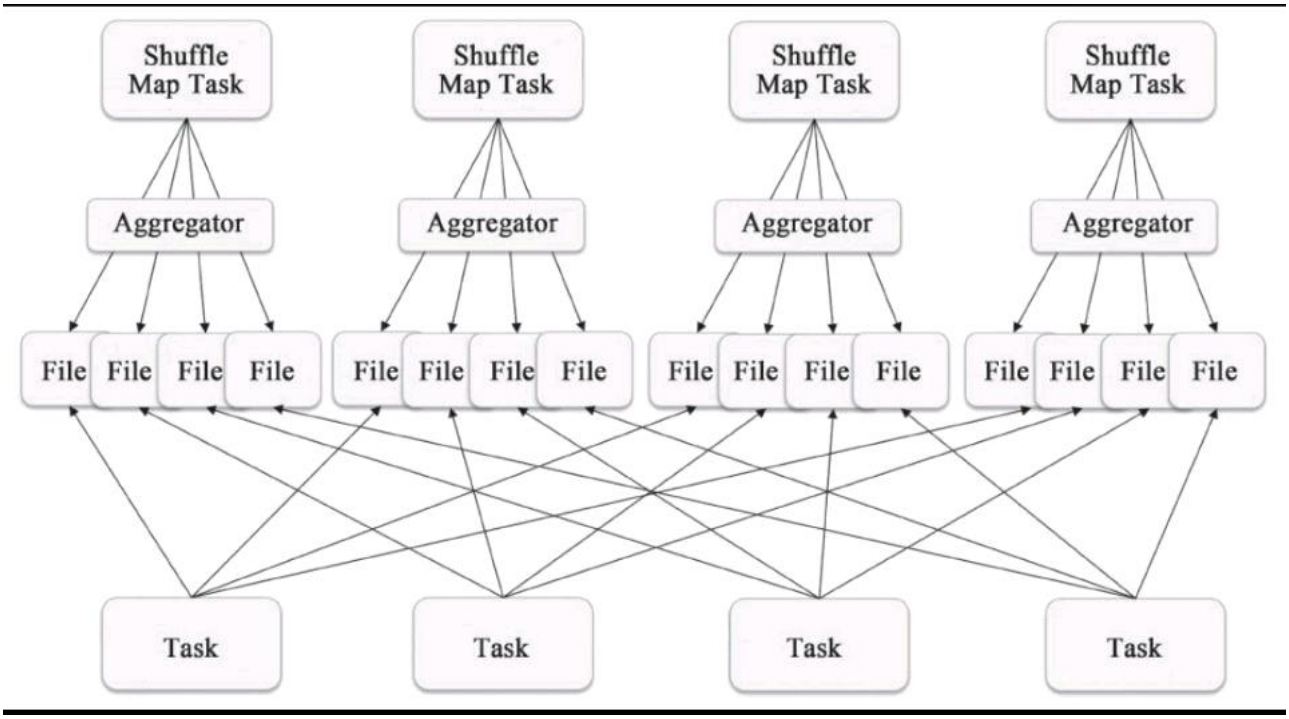
一般来说，每个Task处理的数据可以完全载入内存（如果不能，可以减小每个Partition的大小），因此Task可以做到在内存中计算。**但是对于Shuffle来说，如果不持久化这个中间结果，一旦数据丢失，就需要重新计算依赖的全部RDD**，因此有必要持久化这个中间结果。所以这就是为什么Shuffle过程会产生文件的原因。

Shuffle Write

Shuffle Write，即数据是如何持久化到文件中，以使得下游的Task可以获取到其需要处理的数据的（即Shuffle Read）。在Spark 0.8之前，Shuffle Write是持久化到缓存的，但后来发现实际应用中，shuffle过程带来的数据通常是巨量的，所以经常会发生内存溢出的情况，所以在Spark 0.8以后，Shuffle Write会将数据持久化到硬盘，再之后Shuffle Write不断进行演进优化，但是数据落地到本地文件系统的实现并没有改变。

1）Hash Based Shuffle Write

在Spark 1.0以前，Spark只支持Hash Based Shuffle。因为在很多运算场景中并不需要排序，因此多余的排序只能使性能变差，比如Hadoop的Map Reduce就是这么实现的，也就是Reducer拿到的数据都是已经排好序的。实际上Spark的实现很简单：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在计算时会通过网络（如果该Task与上游的Shuffle Map Task运行在同一个节点上，那么此时就是一个本地的硬盘读写）读取这个文件并进行计算。



Hash Based Shuffle Write存在的问题

由于每个Shuffle Map Task需要为每个下游的Task创建一个单独的文件，因此文件的数量就是：

number（shuffle_map_task）*number（following_task）。

如果Shuffle Map Task是1000，下游的Task是500，那么理论上会产生500000个文件（对于size为0的文件Spark有特殊的处理）。生产环境中Task的数量实际上会更多，因此这个简单的实现会带来以下问题：

1）每个节点可能会同时打开多个文件，每次打开文件都会占用一定内存。假设每个Write Handler的默认需要100KB的内存，那么同时打开这些文件需要50GB的内存，对于一个集群来说，还是有一定的压力的。尤其是如果Shuffle Map Task和下游的Task同时增大10倍，那么整体的内存就增长到5TB。

2）从整体的角度来看，打开多个文件对于系统来说意味着随机读，尤其是每个文件比较小但是数量非常多的情况。而现在机械硬盘在随机读方面的性能特别差，非常容易成为性能的瓶颈。如果集群依赖的是固态硬盘，也许情况会改善很多，但是随机写的性能肯定不如顺序写的。

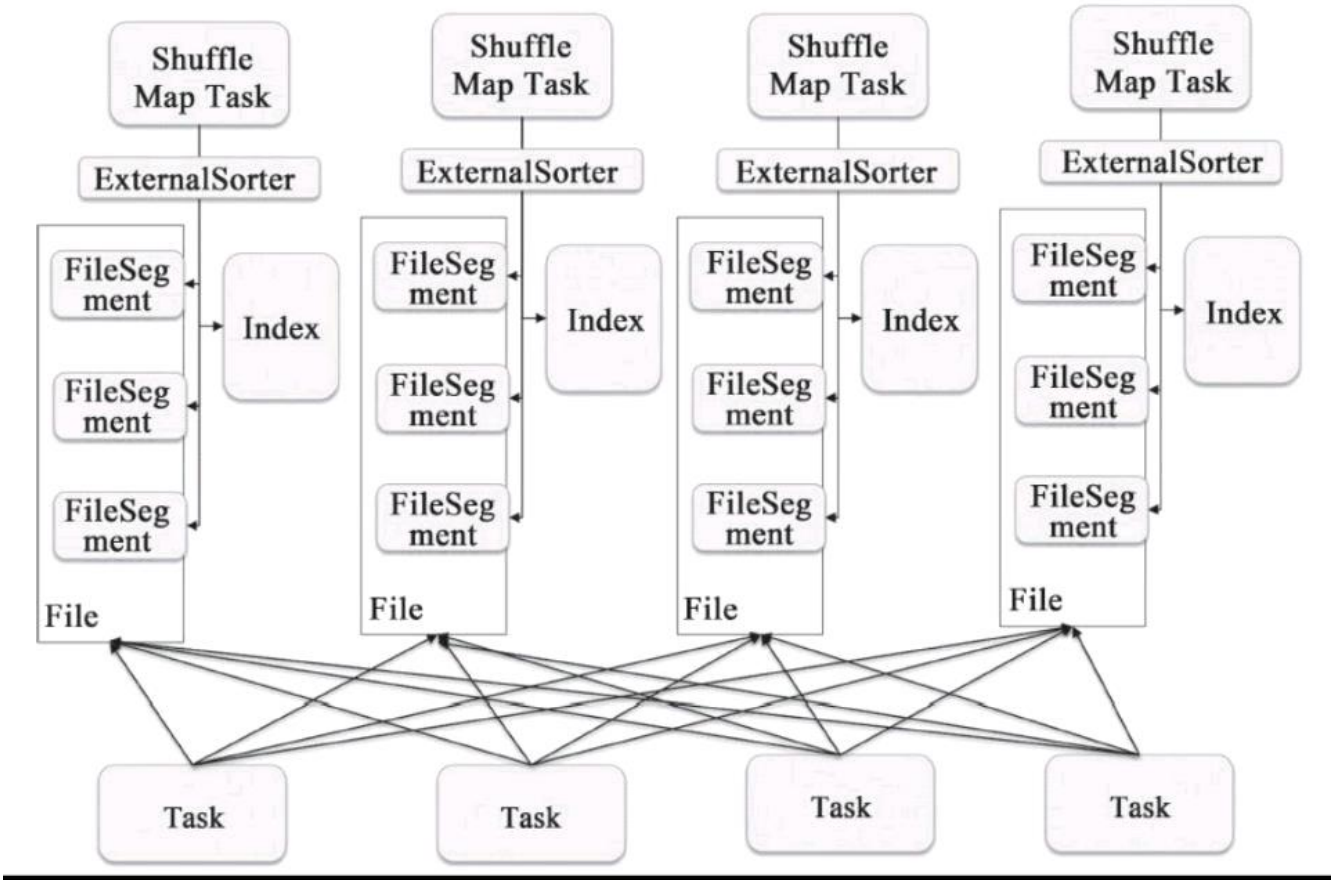
2）Sort Based Shuffle Write

在Spark 1.2.0中，Spark Core的一个重要的升级就是将默认的Hash Based Shuffle换成了Sort Based Shuffle，即spark.shuffle.manager从Hash换成了Sort，对应的实现类分别是org.apache.spark.shuffle.hash.HashShuffleManager和org.apache.spark.shuffle.sort.SortShuffleManager。

那么Sort Based Shuffle“取代”Hash Based Shuffle作为默认选项的原因是什么？

正如前面提到的，Hash Based Shuffle的每个Mapper都需要为每个Reducer写一个文件，供Reducer读取，即需要产生M*R个数量的文件，如果Mapper和Reducer的数量比较大，产生的文件数会非常多。

而Sort Based Shuffle的模式是：每个Shuffle Map Task不会为每个Reducer生成一个单独的文件；相反，它会将所有的结果写到一个文件里，同时会生成一个Index文件，



Reducer可以通过这个Index文件取得它需要处理的数据。避免产生大量文件的直接收益就是节省了内存的使用和顺序Disk IO带来的低延时。节省内存的使用可以减少GC的风险和频率。而减少文件的数量可以避免同时写多个文件给系统带来的压力。

Sort Based Write实现详解

Shuffle Map Task会按照key相对应的Partition ID进行Sort，其中属于同一个Partition的key不会Sort。因为对于不需要Sort的操作来说，这个Sort是负收益的；要知道之前Spark刚开始使用Hash Based的Shuffle而不是Sort Based就是为了避免Hadoop Map Reduce对于所有计算都会Sort的性能损耗。对于那些需要Sort的运算，比如sortByKey，这个Sort在Spark 1.2.0里还是由Reducer完成的。

①答出shuffle的定义

②spark shuffle的特点

③spark shuffle的目的

④spark shuffel的实现类，即对应优缺点