

法律声明

□ 本课件包括：演示文稿，示例，代码，题库，视频和声音等，小象学院拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意，我们将保留一切通过法律手段追究违反者的权利。

□ 课程详情请咨询

■ 微信公众号：小象学院

■ 新浪微博：小象AI学院



大纲

- 上节课内容回顾
- Spark Shuffle
- Spark 程序设计

大纲

上节课内容回顾

Spark计算引擎回顾-问题解决

问题： Spark使用textFile读取HDFS只有一个block的文件时，会产生2个task。

分析： 默认读取HDFS文件，第一个stage的task数与处理的RDD分区数相同，RDD的分区与HDFS存在的文件block一一对应，所以按照正常流程读取只有一个block的文件应该只有一个task，猜测问题应该出现在textFile方法上。

验证： 经过查看SparkContext源码，发现textFile方法在参数中定义了最小分区数minPartitions初始值defaultMinPartitions，如果用户没有指定最小分区数，则使用默认的最小分区数defaultMinPartitions的值2。所以当使用textFile读取HDFS文件的时候，用户不设置最小分区数，即使hdfs文件只有一个block块，产生的RDD默认最小也会有2个分区，从而产生两个task。

```
scala> val rdd = sc.textFile("hdfs://192.168.183.100:9000/data/wctest/in")
rdd: org.apache.spark.rdd.RDD[String] = hdfs://192.168.183.100:9000/data/wctest/in MapPartitionsRDD[29]
] at textFile at <console>:24

scala> rdd.partitions.size
res14: Int = 2
```

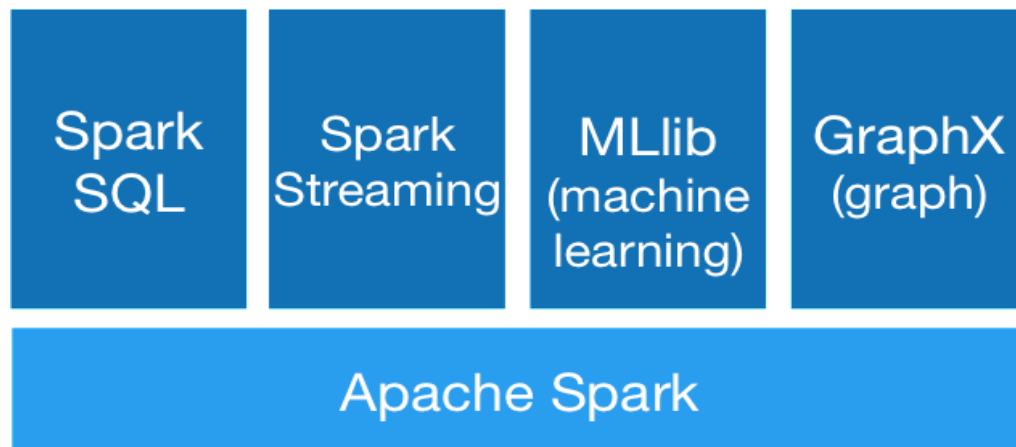
解决： 如果Spark通过textFile方法读取HDFS只有一个block的文件，在调用textFile的时候设置最小分区数值为1

```
scala> val rdd = sc.textFile("hdfs://192.168.183.100:9000/data/wctest/in",1)
rdd: org.apache.spark.rdd.RDD[String] = hdfs://192.168.183.100:9000/data/wctest/in MapPartitionsRDD[31]
] at textFile at <console>:24

scala> rdd.partitions.size
res15: Int = 1
```

Spark计算引擎回顾

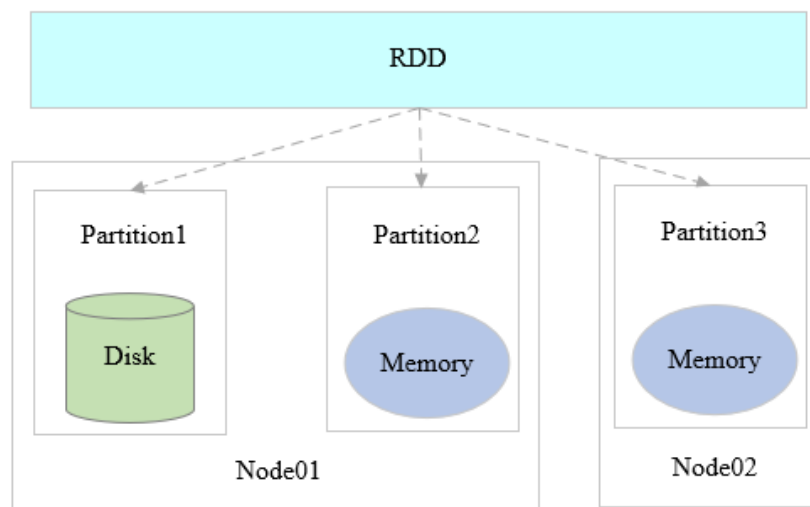
- Spark是一个规模分布式通用计算引擎
 - 具有高吞吐、低延时、通用易扩展、高容错等特点
 - Scala语言开发，提供了丰富的开发API
 - 提供了多种运行模式：local、standalone、yarn-client、yarn-cluster等
 - 集成了SparkSQL、SparkStreaming、Mllib、GraphX



Spark计算引擎回顾-RDD

➤ RDD弹性分布式数据集

- 多分区只读对象集合
- 多种缓存策略，可以存储在磁盘和内存中
- 两种创建方式
 - ✓ 从文件系统输入创建
 - ✓ 从已有的RDD转换创建
- 失效后自动构建



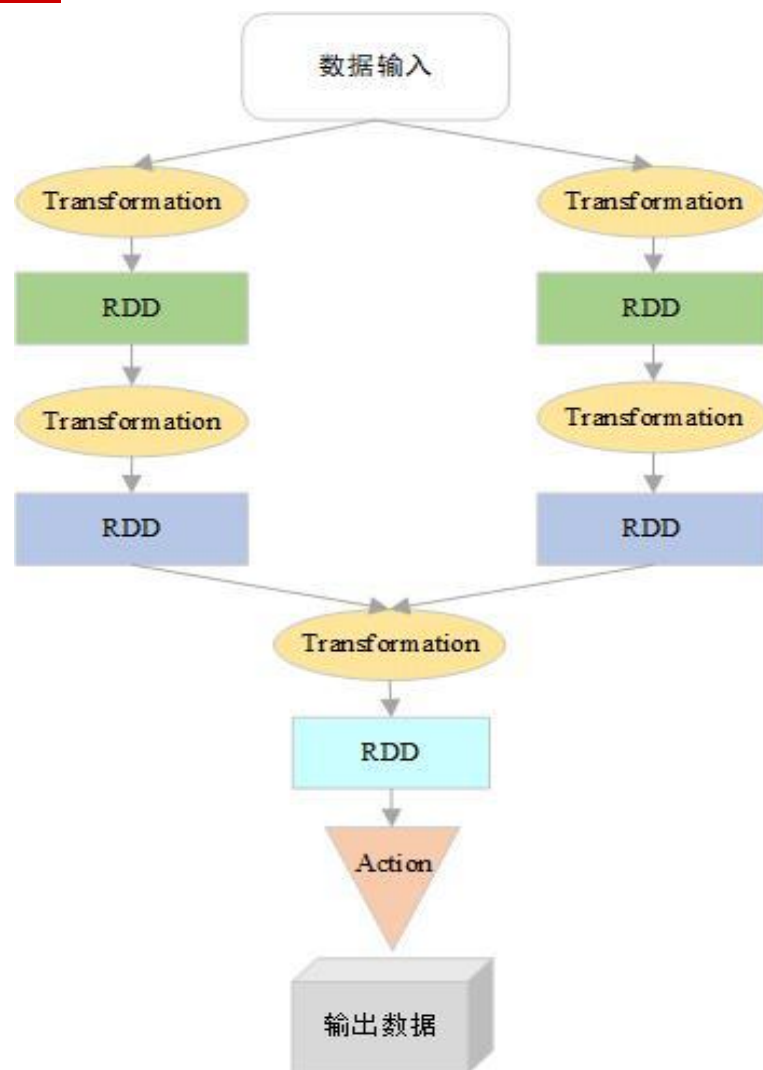
Spark计算引擎回顾-RDD操作

➤ Transformation

- 转换生成新的RDD
- 惰性执行，只记录RDD转化关系，不触发计算执行

➤ Action

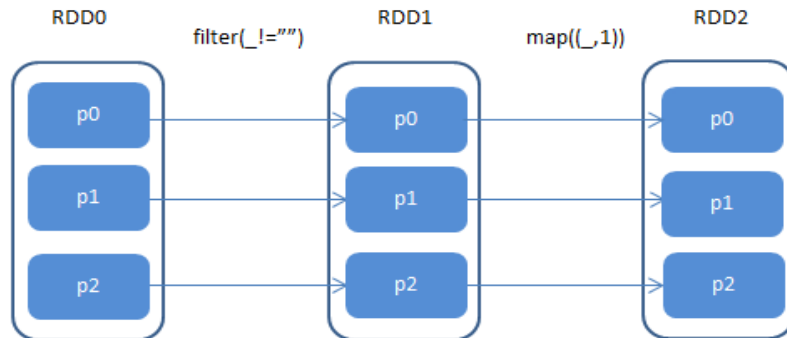
- 真正触发计算执行
- 通过RDD计算得到一个或者一组值



Spark计算引擎回顾-RDD依赖

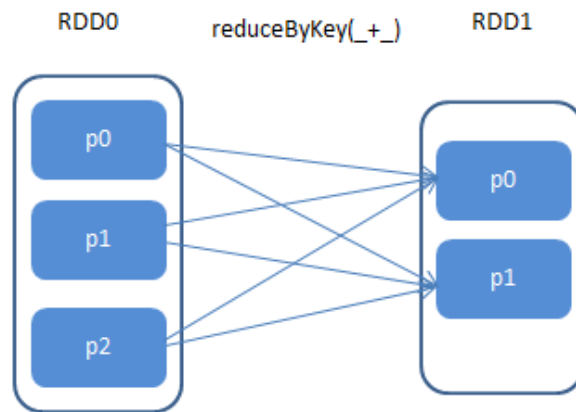
Narrow Dependency窄依赖

- 父RDD中的分区最多只能被一个子RDD的一个分区使用
- 子RDD如果只有部分分区数据丢失或者损坏只需要从对应的父RDD重新计算恢复



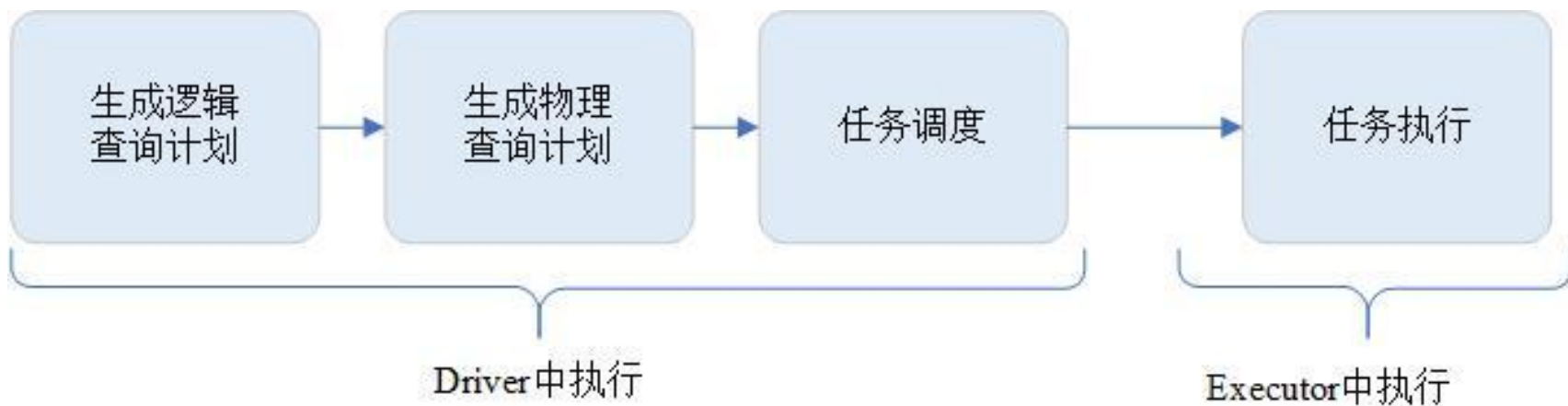
Shuffle Dependency宽依赖

- 子RDD分区依赖父RDD所有分区
- 子RDD如果部分分区或者全部分区数据丢失或者损坏需要从所有父RDD重新计算，相对窄依赖付出的代价更高，尽量避免宽依赖的使用



Spark计算引擎回顾-内部执行流程

- DAGScheduler根据计算任务的依赖关系建立DAG（逻辑查询计划）；根据依赖关系是否是宽依赖，将DAG划分为不同的Stage；将各个Stage中的Task组成的TaskSet提交到TaskScheduler（物理查询计划）；TaskScheduler调度Task给Executor执行（任务调度），Executor进程中采用多线程方式并行执行Task(任务执行)
- 失败重试
- 推测执行

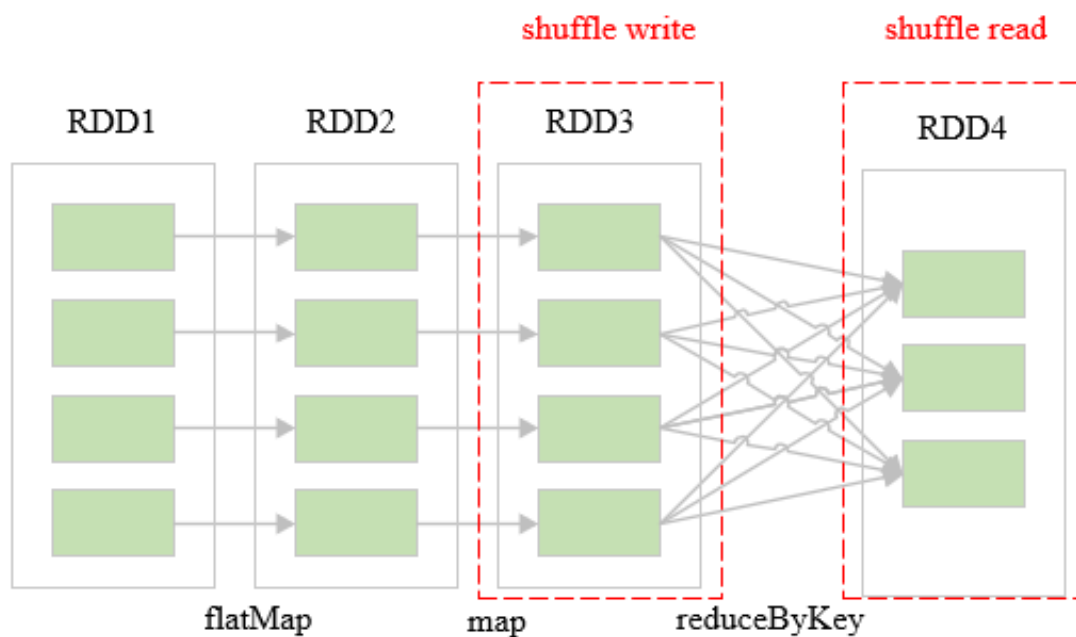


大纲

Spark Shuffle

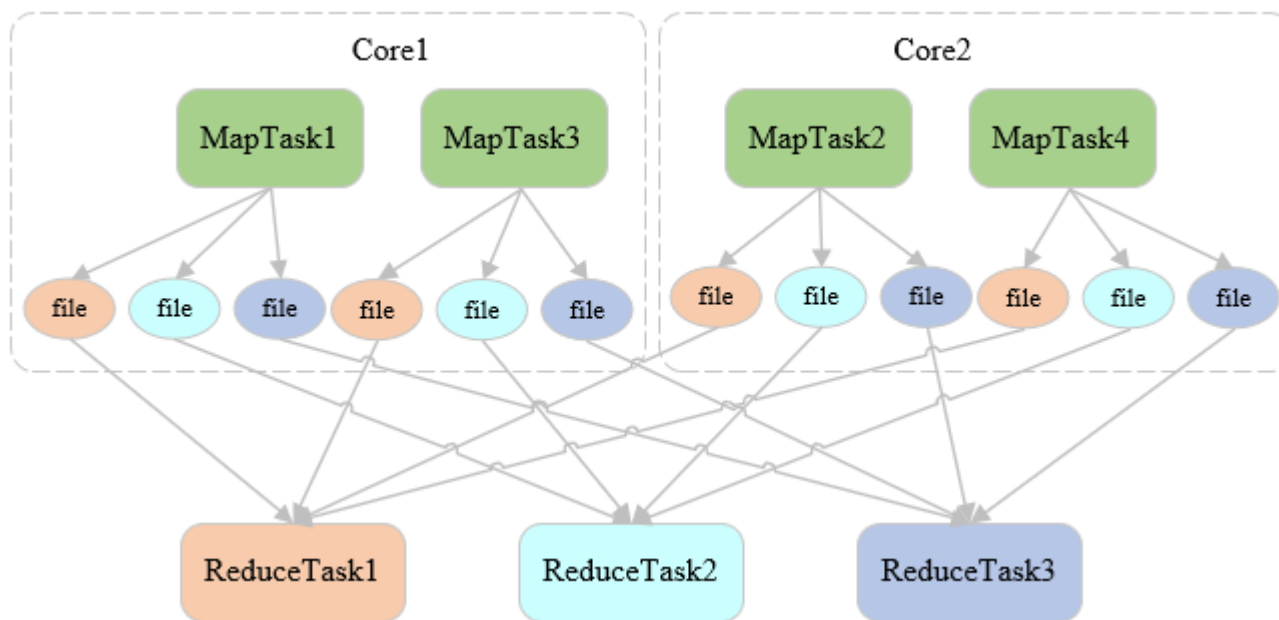
Spark Shuffle

- Shuffle Write: 将Task中间结果数据写入到本地磁盘
- Shuffle Read: 从Shuffle Write阶段拉取数据到内存中并行计算



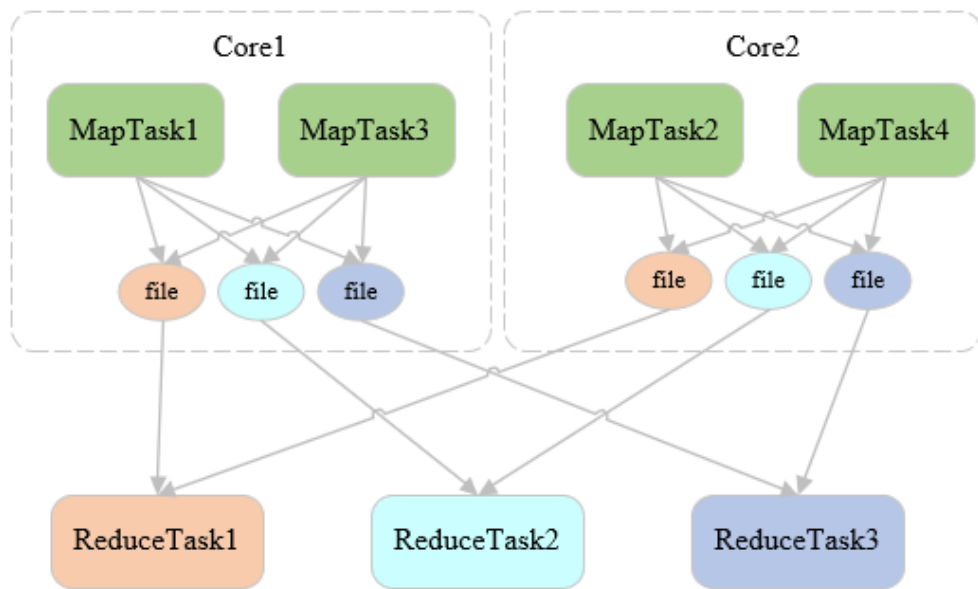
Shuffle Write (hash-based)

- Shuffle Write阶段产生的总文件数=MapTaskNum * ReduceTaskNum
- TotalBufferSize=CoreNum * ReducceTaskNum*FileBufferSize
- 产生大量小文件，占用更多的内存缓冲区，造成不必要的内存开销，增加了磁盘IO和网络开销



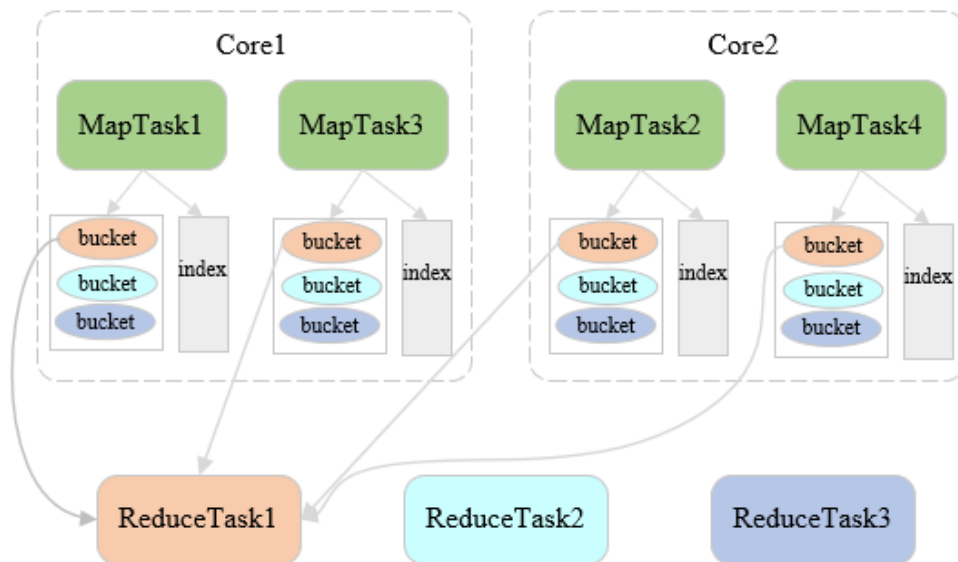
Shuffle Write (hash-based优化)

- Shuffle Write阶段产生的总文件数=CoreNum * ReduceTaskNum
- TotalBufferSize=CoreNum * ReduceTaskNum*FileBufferSize
- 减少了小文件产生的个数，但是占用内存缓冲区的大小没变
- 设置方法
 - `conf.set("spark.shuffle.manager", "hash")`
 - 在`conf/spark-default.conf` 配置文件中添加`spark.shuffle.manager=hash`



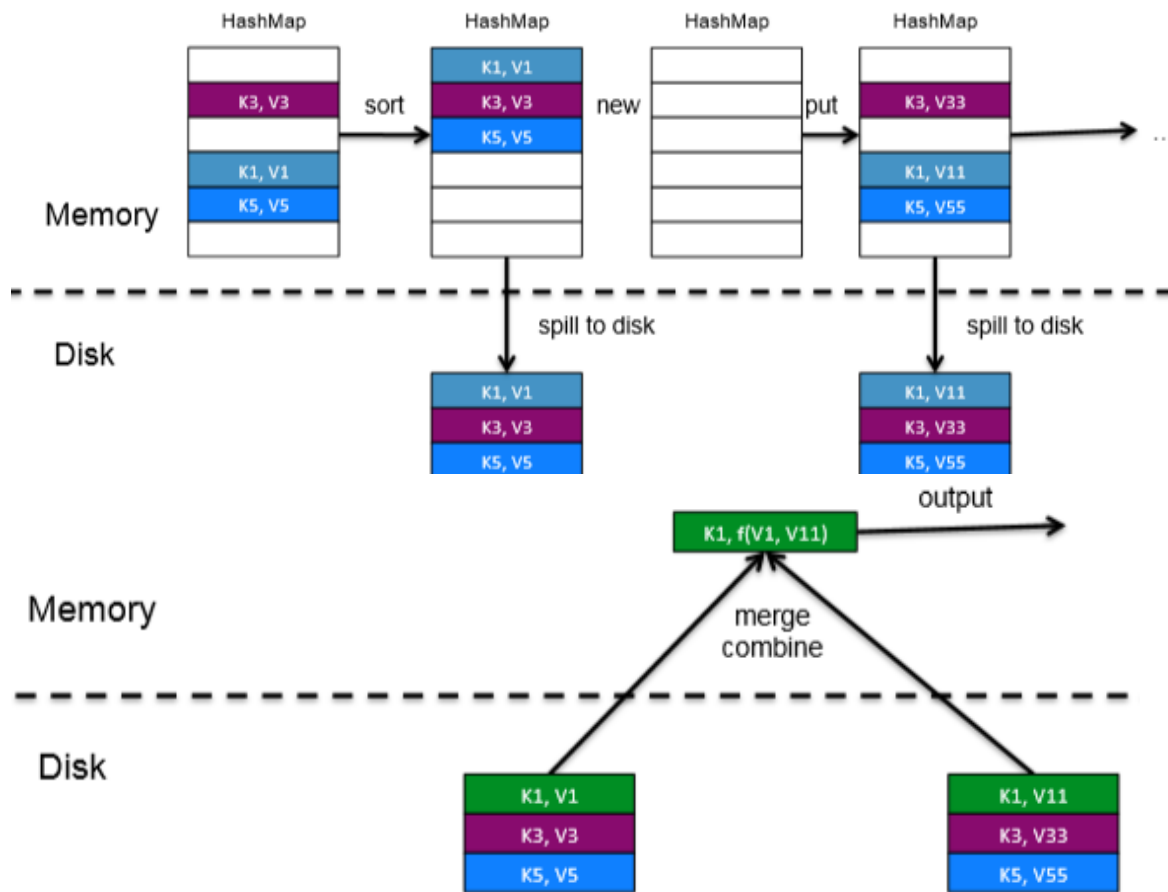
Shuffle Write (sort-based)

- Shuffle Write阶段产生的总文件数= MapTaskNum * 2
- 优点： 顺序读写能够大幅提高磁盘IO性能，不会产生过多小文件，降低文件缓存占用内存空间大小，提高内存使用率。
- 缺点： 多了一次粗粒度的排序。
- 设置方法
 - 代码中设置： `conf.set("spark.shuffle.manager", "sort")`
 - 在conf/spark-default.conf 配置文件中添加`spark.shuffle.manager=sort`



Shuffle Read

- hase-based和sort-based使用相同的shuffle read实现



大纲

Spark程序设计

Spark开发环境搭建

- 下载安装JDK（建议JDK1.8版本）

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

- 下载安装Scala

<http://spark.apache.org/downloads.html>

- 下载安装Maven

<https://maven.apache.org/download.cgi>

- 下载安装IDEA

<https://www.jetbrains.com/idea/download/previous.html>

以上步骤参考课程资料开发环境配置文档

- 演示在IDEA中创建Spark Maven项目

Spark History Server配置

- spark history server查看运行完成的作业信息和日志
- 配置Hadoop的yarn-site.xml文件，所有节点配置文件同步，重启yarn

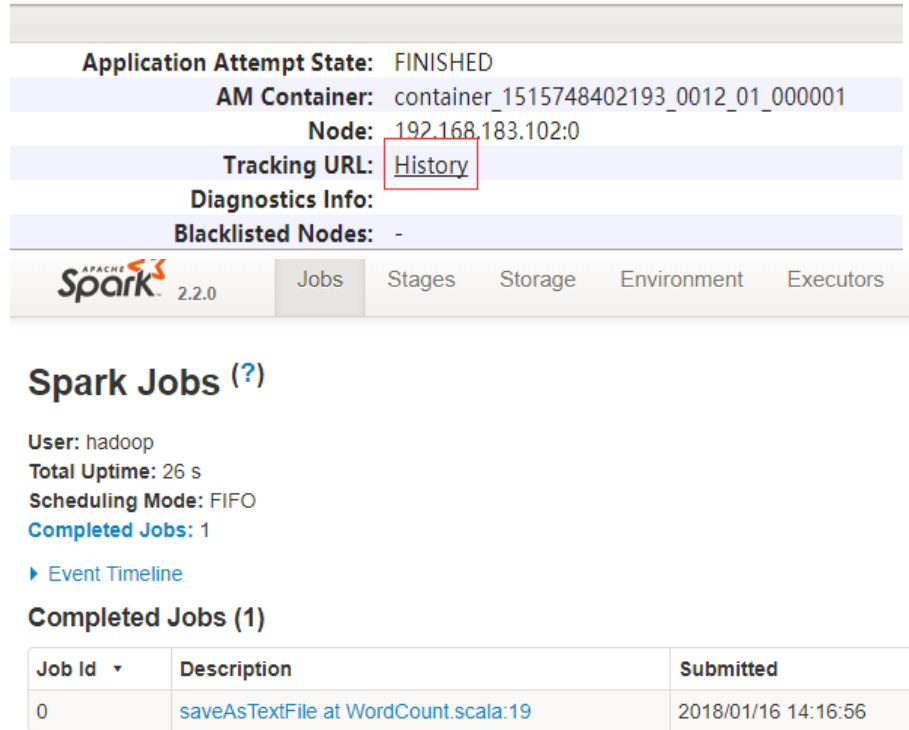
```
<property>  
  <name>yarn.log.server.url</name>  
  <value>http://node02:19888/jobhistory/logs</value>  
  <description> Yarn JobHistoryServer访问地址 </description>  
</property>
```

- 修改spark安装包conf目录下的spark-defaults.conf（如果没有该文件，通过spark-defaults.conf.template模板复制一个），spark history server在192.168.183.100节点启动，spark_logs这个目录需要在HDFS上提前创建

```
spark.yarn.historyServer.address=192.168.183.100:18080  
spark.history.ui.port=18080  
spark.eventLog.enabled=true  
spark.eventLog.dir=hdfs:///spark_logs  
spark.history.fs.logDirectory=hdfs:///spark_logs
```

Spark History Server启动

- 启动Spark History Server
 - `sbin/start-history-server.sh`
- Spark History Server访问地址
 - `httpL://192.168.183.100:18080`
- Spark History Server使用



The screenshot displays the Spark History Server web interface. At the top, it shows the 'Application Attempt State' as 'FINISHED'. Below this, it lists the 'AM Container' as 'container_1515748402193_0012_01_000001' and the 'Node' as '192.168.183.102:0'. The 'Tracking URL' is highlighted with a red box and contains the word 'History'. Below this, it shows 'Diagnostics Info' and 'Blacklisted Nodes' as '-'. A navigation bar at the bottom of the top section includes the 'Spark 2.2.0' logo and tabs for 'Jobs', 'Stages', 'Storage', 'Environment', and 'Executors'. The 'Jobs' tab is selected, showing 'Spark Jobs (?)'. Below this, it displays 'User: hadoop', 'Total Uptime: 26 s', 'Scheduling Mode: FIFO', and 'Completed Jobs: 1'. A link for 'Event Timeline' is also present. The 'Completed Jobs (1)' section contains a table with one job entry.

Job Id	Description	Submitted
0	saveAsTextFile at WordCount.scala:19	2018/01/16 14:16:56

Spark运行环境优化

- 将spark系统jar包上传到HDFS上，直接使用HDFS上的文件
 - 在spark安装目录下运行：`jar cv0f spark-libs.jar -C jars/ .`
 - 将spark安装目录下生成的spark-libs.jar上传到HDFS上的
`/system/spark`（需要手动创建）目录下
`hadoop fs -put spark-libs.jar /system/spark`
 - 修改spark安装包conf目录下spark-defaults.conf配置文件添加spark-libs.jar在HDFS上的路径
`spark.yarn.archive=hdfs:///system/spark/spark-libs.jar`

Spark编程模型

➤ 创建SparkContext

- 封装了spark执行环境信息

➤ 创建RDD

- 可以用scala集合或hadoop数据文件创建

➤ 在RDD上进行transformation和action

- spark提供了丰富的transformation和action算子

➤ 返回结果

- 保存到hdfs、其他外部存储、直接打印

Spark编程模型-WordCount

```
import org.apache.spark.{SparkContext, SparkConf}
/**
 * Created by xiaoguan yu on 2018/1/15.
 */
object WordCount {
  def main(args: Array[String]) {
    val inPath = args(0)
    val outputPath = args(1)
    val sparkConf = new SparkConf().setAppName("SparkWordCount")
    val sc = new SparkContext(sparkConf)
    val rowRdd = sc.textFile(inPath)
    val resultRdd = rowRdd.flatMap(_.split("\t")).map((_, 1)).reduceByKey(_ + _)
    resultRdd.saveAsTextFile(outputPath)
    sc.stop()
  }
}
```

提交Spark程序到Yarn上

bin/spark-submit \

--class bigadta.spark.WordCount \

应用程序主类

--master yarn \

运行模式, local、yarn

--deploy-mode cluster \

yarn运行模式, client、cluster

--driver-cores 1 \

Driver需要的CPU Core数

--driver-memory 1g \

Driver需要的内存大小

--num-executors 3 \

需要启动的Executor总数

--executor-cores 2 \

每个Executor启动的线程数

--executor-memory 3g \

每个Executor需要的内存大小

./FirstSparkPro-1.0-SNAPSHOT.jar /data/wc/in /data/wc/out10

应用程序jar包路径

输入参数

Scala基础补充-方法

➤ 方法定义

- 使用def关键字定义方法
- 不需要使用return关键词显示返回，最后一行计算结果作为返回值

```
def methonName(param1 : Int, param2 : Int) : Int = {  
    param1 + param2  
}
```

方法名称 参数名称 参数类型 返回值类型

最后一行计算结果最为返回值

- 返回值类型可以根据返回结果自动推断
- 如果方法体只有一行，可以把“{}”去掉

```
def methonName(param1 : Int, param2 : Int) = param1 + param2
```


Scala基础补充-函数

➤ 值函数

- 定义函数不需要使用def关键字，定义一个变量作为值函数名称
- 参数列表和方法体之间使用“=>”标识，而不是使用“=”，返回值类型自动推断
- 调用值函数的时候直接使用变量名称

```
值函数名称      参数列表
val f1 = (x : Int, y : Int) => {
  x + y
}
```

方法体，返回值自动推断

Scala基础补充-函数

➤ 匿名函数

- 没有函数名称的函数
- 常用于在方法中作为参数传入

```
scala> (x : Int, y : Int) => x + y  
res0: (Int, Int) => Int = <function2>  
  
scala> (x : Int, y : Int) => {x + y}  
res1: (Int, Int) => Int = <function2>
```

- 如果匿名函数只有一个参数，类型可以自动推断，并且可以省略“=>”和左边的参数列表，使用“_”占位符表示参数

```
scala> 1 to 10  
res8: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
  
scala> res8.map((x : Int) => x + 10)  
res9: scala.collection.immutable.IndexedSeq[Int] = Vector(11, 12, 13, 14, 15, 16, 17, 18, 19, 20)  
  
scala> res8.map(x => x + 10)  
res10: scala.collection.immutable.IndexedSeq[Int] = Vector(11, 12, 13, 14, 15, 16, 17, 18, 19, 20)  
  
scala> res8.map(_ + 10)  
res11: scala.collection.immutable.IndexedSeq[Int] = Vector(11, 12, 13, 14, 15, 16, 17, 18, 19, 20)
```

Scala基础补充-方法与函数

➤ 方法与函数的区别

- 在Scala中，函数是“头等公民”，它可以像任何数据类型一样被传递和操作

➤ 定义一个方法，将函数作为参数传入

```
def m1(f : Int => Int): Int = {  
    //在方法体内调用函数f  
    f(10)  
}
```

Scala基础补充-元组

- 元组是用“()”定义的一种集合类型
- 元组中的元素类型可以不同
- 元组是有序的集合
- 获取元组中的元素可以使用下划线加脚标获取，注意元组中的元素脚标是从1开始的
- 如果一个方法想返回多个值可以使用元组

```
scala> val t1 = ("scala", 123, true)
t1: (String, Int, Boolean) = (scala, 123, true)

scala> t1._1
res0: String = scala

scala> t1._2
res1: Int = 123

scala> t1._3
res2: Boolean = true
```

Scala基础补充-伴生对象

- 在Scala中没有静态方法和静态字段，但是可以使用object语法结构来达到相同目的
- 在Scala的类中，由object修饰，并且与类名相同的对象叫做伴生对象，类和伴生对象之间可以相互访问私有的方法和属性
- 类的main方法在该类的伴生对象中定义

```
class People {  
    private var name = "scala"  
    def getAge():Unit = {  
        println("age ->" + People.age)  
    }  
}  
  
object People{  
    private val age = 20  
    def main(args: Array[String]) {  
        val p = new People()  
        println("name->" + p.name)  
        p.name = "spark"  
        println("newName ->" + p.name)  
        p.getAge()  
    }  
}
```

Scala基础补充-map,flatten和flatMap

- **map**: 遍历集合，根据已有的集合创建一个新的集合，两个集合元素一一对应

```
scala> val list = List(1,2,3,4)
list: List[Int] = List(1, 2, 3, 4)

scala> list.map(_ * 10)
res4: List[Int] = List(10, 20, 30, 40)
```

- **flatten**: 把嵌套的结构展开

```
scala> val arr = Array(List(1,2,3),List(4,5,6))
arr: Array[List[Int]] = Array(List(1, 2, 3), List(4, 5, 6))

scala> arr.flatten
res3: Array[Int] = Array(1, 2, 3, 4, 5, 6)
```

- **flatMap**: 结合了map和flatten的功能。flatMap接收一个可以处理嵌套列表的函数，然后把返回结果连接起来。

```
scala> arr.flatMap(x => x.map(_ * 10))
res7: Array[Int] = Array(10, 20, 30, 40, 50, 60)
```

Spark RDD算子分类

- Transformation转换操作，惰性执行，不触发app执行
 - 针对Value数据类型，如map、filter
 - 针对Key-Value数据类型，如groupByKey、reduceByKey
- Action执行操作，触发app执行

创建RDD

➤ parallelize从集合创建RDD

- 参数1: Seq集合, 必须
- 参数2: 分区数
- 创建RDD: `val rdd = sc.parallelize(List(1,2,3,4,5,6,7),3)`
- 查看RDD分区数: `rdd.partitions.size`

➤ textFile从外部数据源（本地文件或者HDFS数据集）创建RDD

- 参数1: 外部数据源路径, 必须
- 参数2: 最小分区数
- 从本地文件创建RDD: `val rdd = sc.textFile("file:///home/hadoop/apps/in")`
- 从HDFS数据集创建RDD: `val rdd = sc.textFile("hdfs:///data/wc/in",1)`

Value数据类型Transformation

- map
 - 输入是一个RDD，将一个RDD中的每个数据项，通过map中的函数映射输出一个新的RDD，输入分区与输出分区一一对应
- flatMap
 - 与map算子功能类似，可以将嵌套类型数据拆开展平
- distinct
 - 对RDD元素进行去重
- coalesce
 - 对RDD进行重分区
 - 第一个参数为重分区的数目
 - 第二个为是否进行shuffle，默认为false，如果重分区之后分区数目大于原RDD的分区数，则必须设置为true
- repartition
 - 对RDD进行重分区， 等价于coalesce第二个参数设置为true

Value数据类型Transformation

- union
 - 将两个RDD进行合并，不去重
- mapPartitions
 - 针对RDD的每个分区进行操作，接收一个能够处理迭代器的函数作为参数
 - 如果RDD处理的过程中，需要频繁的创建额外对象，使用mapPartitions要比使用map的性能高很多，如：创建数据库连接
- mapPartitionsWithIndex
 - 与mapPartitions功能类似，接收一个第一个参数是分区索引，第二个参数是分区迭代器的函数
- zip
 - 拉链操作，将两个RDD组合成Key-Value形式的RDD，保证两个RDD的partition数量和元素个数要相同，否则会抛出异常

Key-Value数据类型Transformation

➤ mapValues

- 针对[K,V]中的V值进行map操作

➤ groupByKey

- 将RDD[K,V]中每个K对应的V值，合并到一个集合Iterable[V]中

➤ reduceByKey

- 将RDD[K,V]中每个K对应的V值根据传入的映射函数计算

➤ join

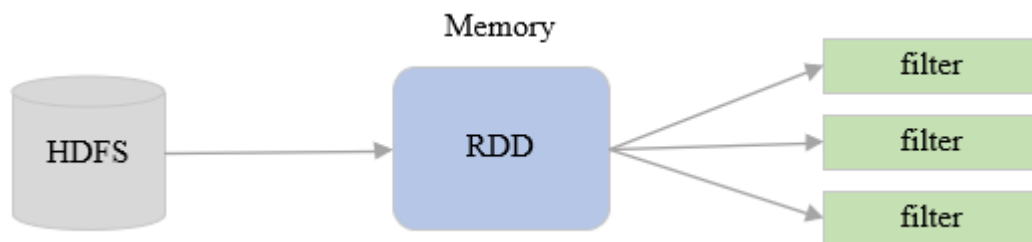
- 返回两个RDD根据K可以关联上的结果，join只能用于两个RDD之间的关联，如果要多个RDD关联，需要关联多次

RDD Action

- collect
 - 将一个RDD转换成数组，常用于调试
- saveAsTextFile
 - 用于将RDD以文本文件的格式存储到文件系统中
- take
 - 根据传入参数返回RDD的指定个数元素
- count
 - 返回RDD中元素数量

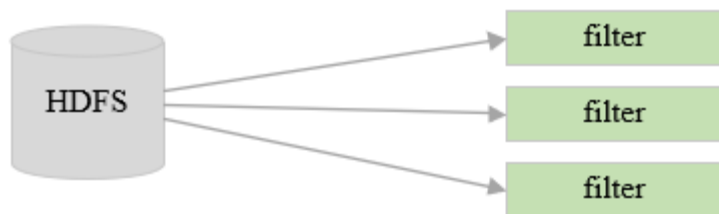
Spark优化-Cache应用

```
val rdd = sc.textFile("/data/in")  
rdd.cache()  
rdd.filter(_.startsWith("error")).count()  
rdd.filter(_.startsWith("spark")).count()  
rdd.filter(_.startsWith("hadoop")).count()
```



第一次action触发执行的时候从HDFS读取一次，后边的每次action都会从缓存中读

```
val rdd = sc.textFile("/data/in")  
rdd.cache()  
rdd.filter(_.startsWith("error")).count()  
rdd.filter(_.startsWith("spark")).count()  
rdd.filter(_.startsWith("hadoop")).count()
```



每次action都会从HDFS中读

Accumulator计数器

➤ accumulator累加器，计数器

- 通常用于监控，调试，记录关键数据处理的数目等
- 分布式计数器，在Driver端汇总

```
val total_counter = sc.accumulator(0L,"total_counter")
val resultRdd = rowRdd.flatMap(_.split("\t")).map(x=>{
    total_counter += 1
    (x,1)
}).reduceByKey(_ + _)
```

- 通过Spark Web UI查看

Accumulators

Accumulable	Value
total_counter	6

疑问

□ 小象问答官网

■ <http://wenda.chinahadoop.cn>

联系我们

小象学院：互联网新技术在线教育领航者

- 微信公众号：小象学院
- 新浪微博：小象AI学院

