《Spark编程基础（Scala版）》

教材官网： http://dblab.xmu.edu.cn/post/spark/

温馨提示：编辑幻灯片母版，可以修改每页PPT的厦大校徽和底部文字

第5章 RDD编程

（PPT版本号： 2018年7月版本）

扫一扫访问教材官网

林子雨

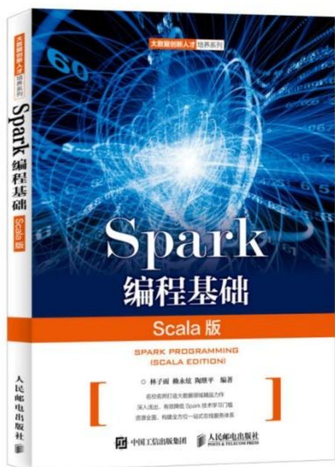
厦门大学计算机科学系

E-mail: ziyulin@xmu.edu.cn

主页：**http://www.cs.xmu.edu.cn/linziyu**

课程教材

本套讲义PPT属于以下教材的配套材料



《Spark编程基础（Scala版）》

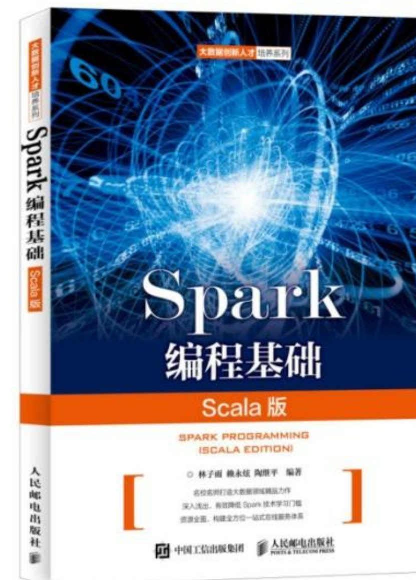
厦门大学 林子雨，赖永炫，陶继平 编著

披荆斩棘，在大数据丛林中开辟学习捷径
填沟削坎，为快速学习Spark技术铺平道路
深入浅出，有效降低Spark技术学习门槛
资源全面，构建全方位一站式在线服务体系

人民邮电出版社出版发行，ISBN:978-7-115-48816-9
教材官网：http://dblab.xmu.edu.cn/post/spark/

本书以Scala作为开发Spark应用程序的编程语言，系统介绍了Spark编程的基础知识。全书共8章，内容包括大数据技术概述、Scala语言基础、Spark的设计与运行原理、Spark环境搭建和使用方法、RDD编程、Spark SQL、Spark Streaming、Spark MLlib等。本书每个章节都安排了入门级的编程实践操作，以便读者更好地学习和掌握Spark编程方法。本书官网免费提供了全套的在线教学资源，包括讲义PPT、习题、源代码、软件、数据集、授课视频、上机实验指南等。

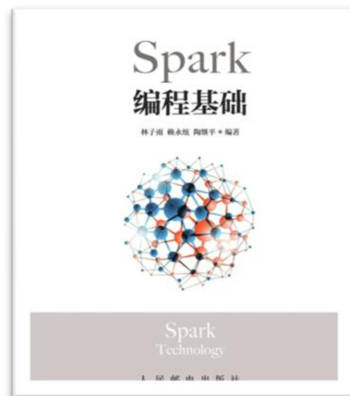
课程配套授课视频



课程在线视频地址：http://dblab.xmu.edu.cn/post/10482/

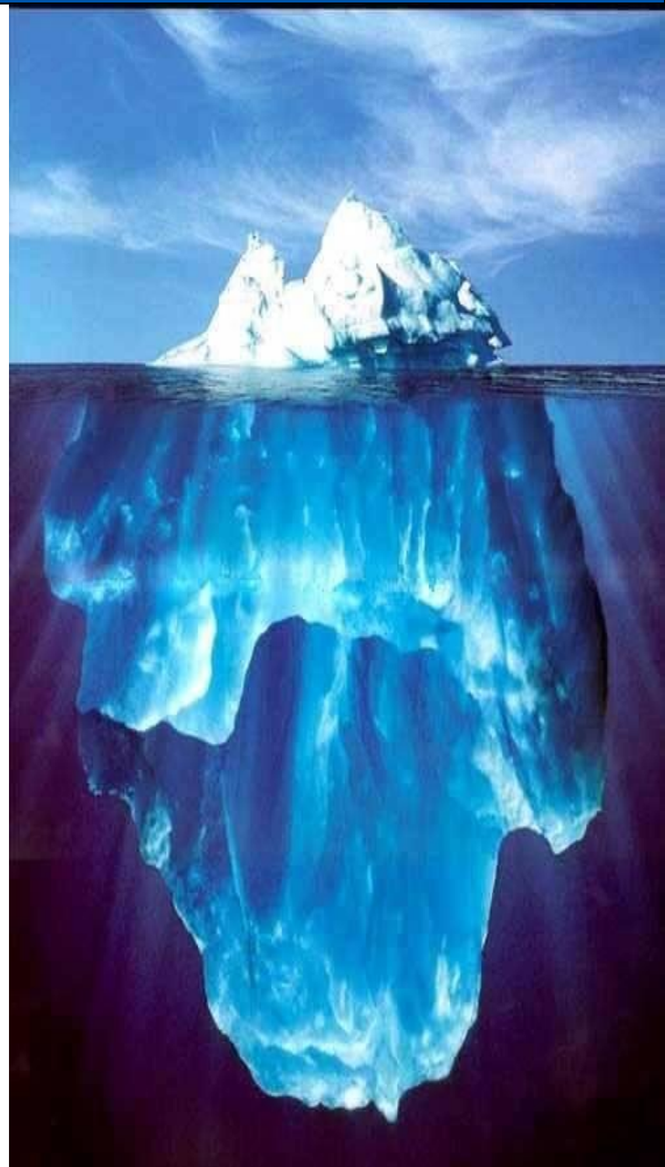
提纲

5.1 RDD编程基础
5.2 键值对RDD
5.3 数据读写
5.4 综合案例



高校大数据课程
公 共 服 务 平 台

百度搜索厦门大学数据库实验室网站访问平台



5.1 RDD编程基础

5.1.1 RDD创建
5.1.2 RDD操作
5.1.3 持久化
5.1.4 分区
5.1.5 一个综合实例

5.1.1 RDD创建

1. 从文件系统中加载数据创建RDD
2. 通过并行集合（数组）创建RDD

5.1.1 RDD创建

1. 从文件系统中加载数据创建RDD

- Spark采用textFile()方法来从文件系统中加载数据创建RDD
- 该方法把文件的URI作为参数，这个URI可以是：
  - 本地文件系统的地址
  - 或者是分布式文件系统HDFS的地址
  - 或者是Amazon S3的地址等等

5.1.1 RDD创建

（1）从本地文件系统中加载数据创建RDD

```
scala> val lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
lines: org.apache.spark.rdd.RDD[String] =
file:///usr/local/spark/mycode/rdd/word.txt MapPartitionsRDD[12] at textFile
at <console>:27
```

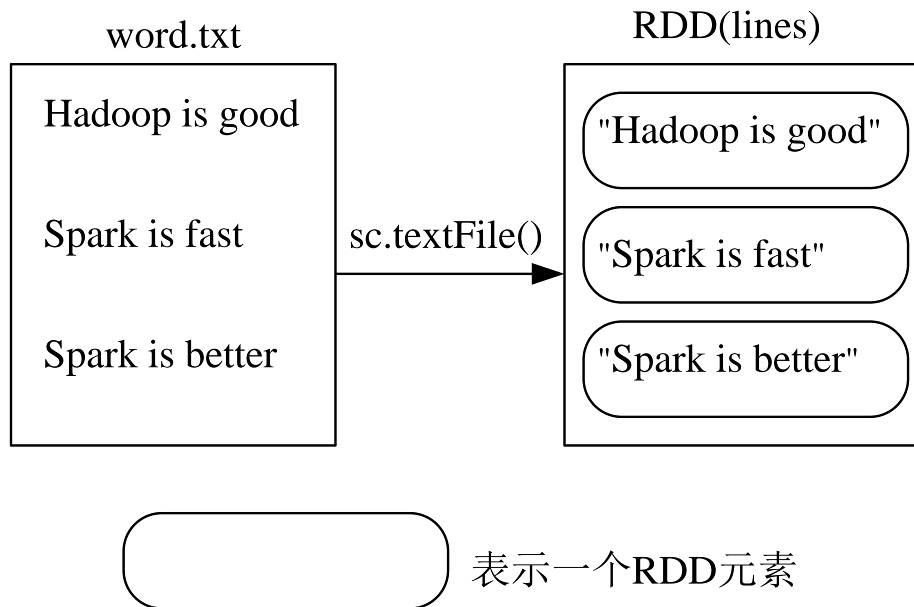


图 从文件中加载数据生成RDD

5.1.1 RDD创建

（2）从分布式文件系统HDFS中加载数据

```
scala> val lines = sc.textFile("hdfs://localhost:9000/user/hadoop/word.txt")
scala> val lines = sc.textFile("/user/hadoop/word.txt")
scala> val lines = sc.textFile("word.txt")
```

三条语句是完全等价的，可以使用其中任意一种方式

5.1.1 RDD创建

2. 通过并行集合（数组）创建RDD

可以调用SparkContext的parallelize方法，在Driver中一个已经存在的集合（数组）上创建。

```
scala>val array = Array(1,2,3,4,5)
array: Array[Int] = Array(1, 2, 3, 4, 5)
scala>val rdd = sc.parallelize(array)
rdd: org.apache.spark.rdd.RDD[Int] =
ParallelCollectionRDD[13] at
parallelize at <console>:29
```

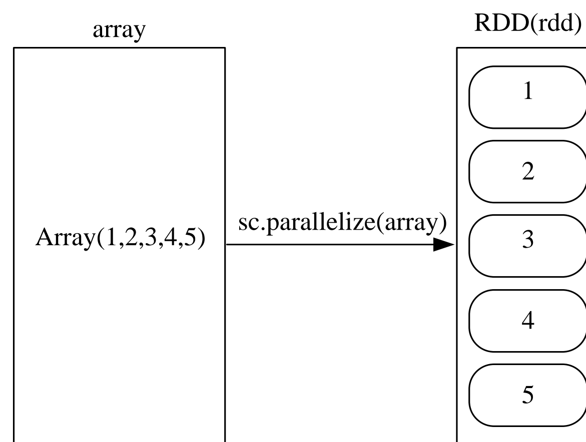


图 从数组创建RDD示意图

或者，也可以从列表中创建：

```
scala>val list = List(1,2,3,4,5)
list: List[Int] = List(1, 2, 3, 4, 5)
scala>val rdd = sc.parallelize(list)
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[14] at
parallelize at <console>:29
```

5.1.2 RDD操作

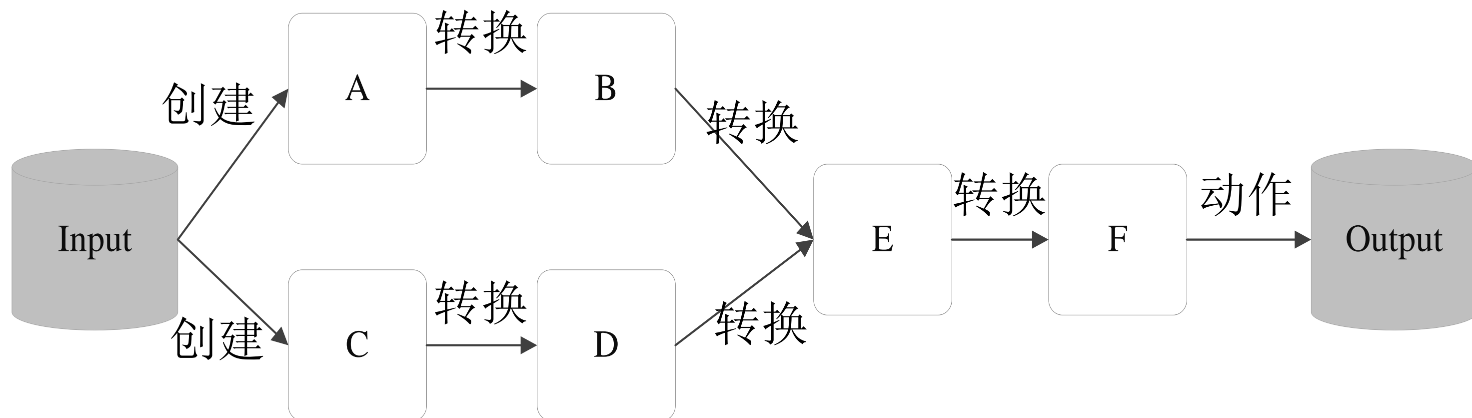
1. 转换操作
2. 行动操作
3. 惰性机制

5.1.2 RDD操作

1. 转换操作

•对于RDD而言，每一次转换操作都会产生不同的RDD，供给下一个“转换”使用

•转换得到的RDD是惰性求值的，也就是说，整个转换过程只是记录了转换的轨迹，并不会发生真正的计算，只有遇到行动操作时，才会发生真正的计算，开始从血缘关系源头开始，进行物理的转换操作



5.1.2 RDD操作

1. 转换操作

表 常用的RDD转换操作API

操作	含义
filter(func)	筛选出满足函数func的元素，并返回一个新的数据集
map(func)	将每个元素传递到函数func中，并将结果返回为一个新的数据集
flatMap(func)	与map()相似，但每个输入元素都可以映射到0或多个输出结果
groupByKey()	应用于(K,V)键值对的数据集时，返回一个新的(K, Iterable)形式的数据集
reduceByKey(func)	应用于(K,V)键值对的数据集时，返回一个新的(K, V)形式的数据集，其中每个值是将每个key传递到函数func中进行聚合后的结果

5.1.2 RDD操作

1. 转换操作

•filter(func)

```
scala>  val  lines =sc.textFile(file:///usr/local/spark/mycode/rdd/word.txt)
scala>  val  linesWithSpark=lines.filter(line => line.contains("Spark"))
```

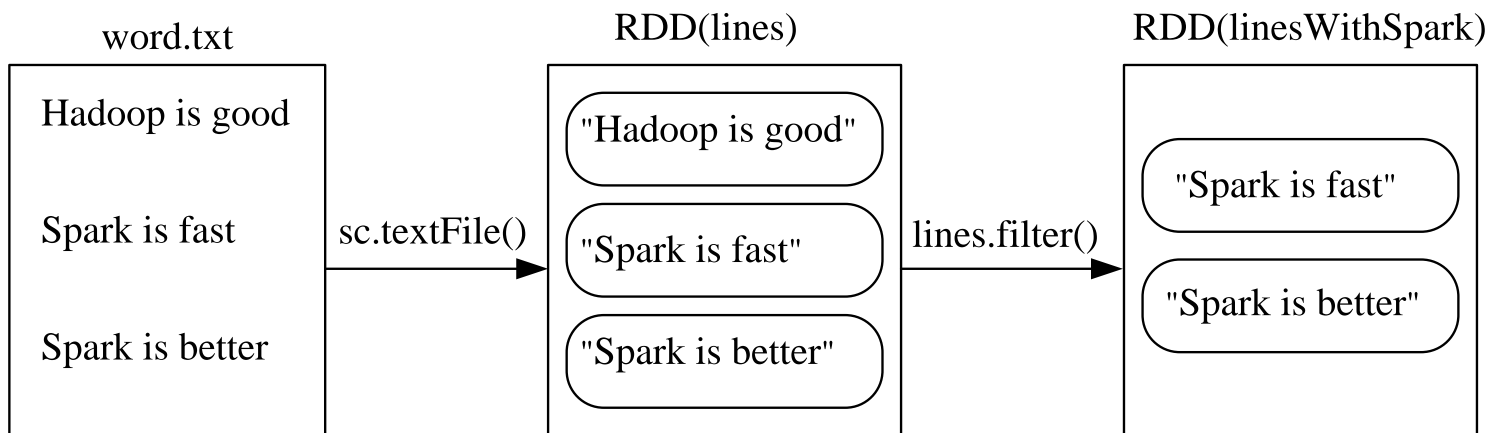


图 filter()操作实例执行过程示意图

5.1.2 RDD操作

1. 转换操作

•map(func)

map(func)操作将每个元素传递到函数func中，并将结果返回为一个新的数据集

```
scala> data=Array(1,2,3,4,5)
scala> val  rdd1= sc.parallelize(data)
scala> val  rdd2=rdd1.map(x=>x+10)
```

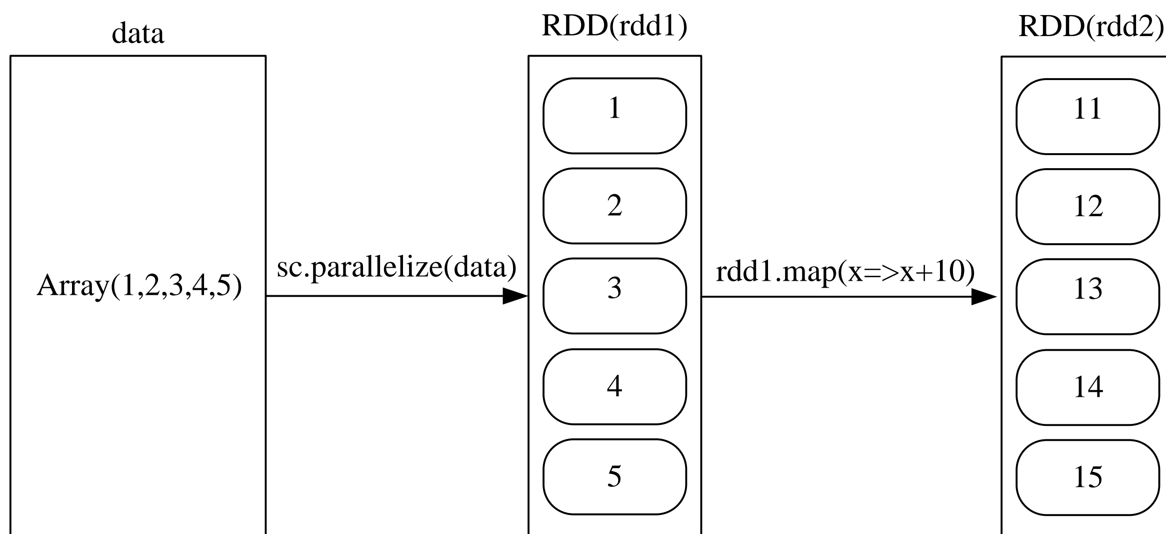


图 map()操作实例执行过程示意图

5.1.2 RDD操作

1. 转换操作

- map(func)

另外一个实例

```
scala> val  lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
scala> val  words=lines.map(line => line.split(" "))
```

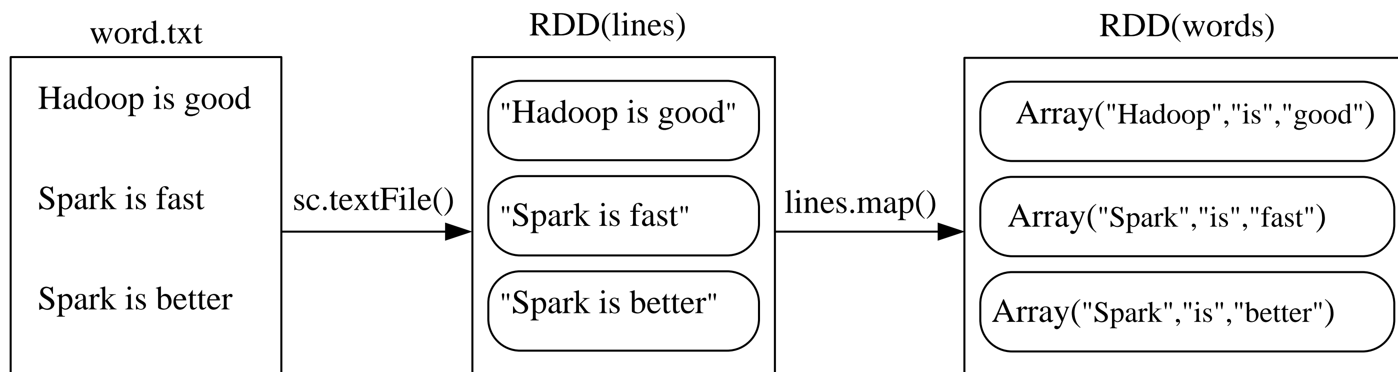


图 map()操作实例执行过程示意图

5.1.2 RDD操作

1. 转换操作

•flatMap(func)

```
scala> val  lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
scala> val  words=lines.flatMap(line => line.split(" "))
```

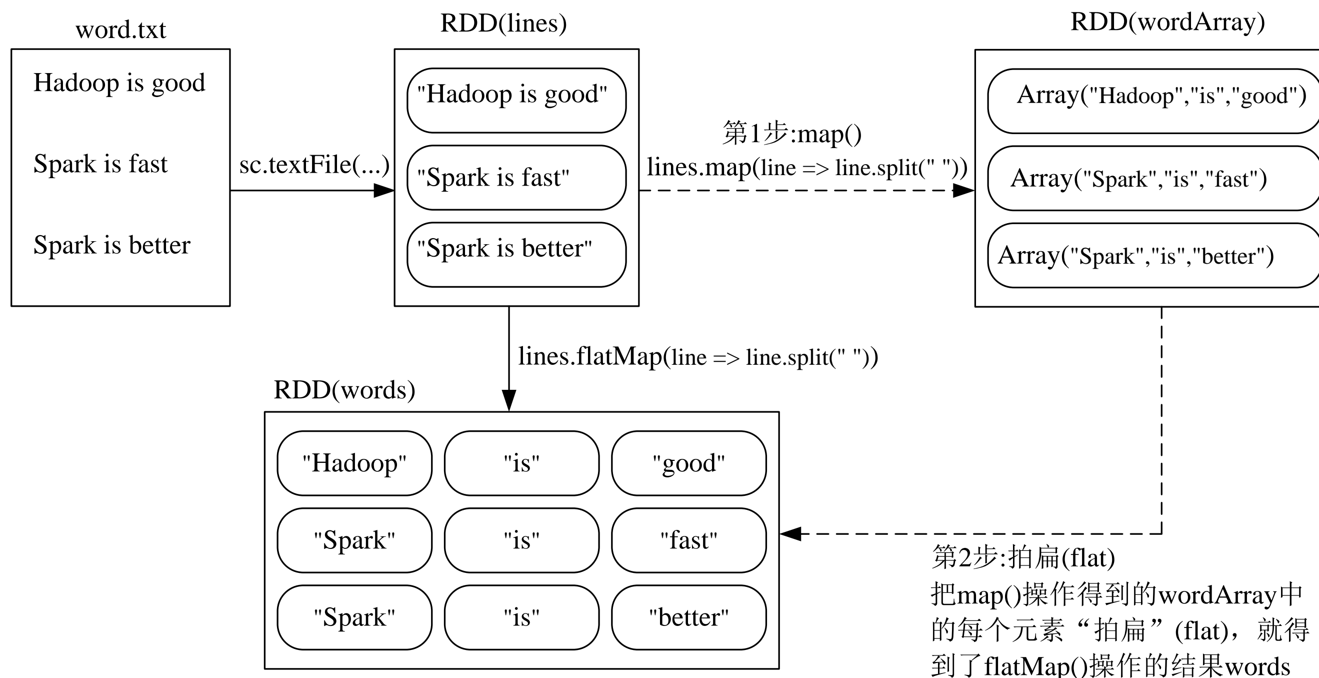


图 flatMap()操作实例执行过程示意图

5.1.2 RDD操作

1. 转换操作

•groupByKey()

groupByKey()应用于(K,V)键值对的数据集时，返回一个新的(K, Iterable)形式的数据集

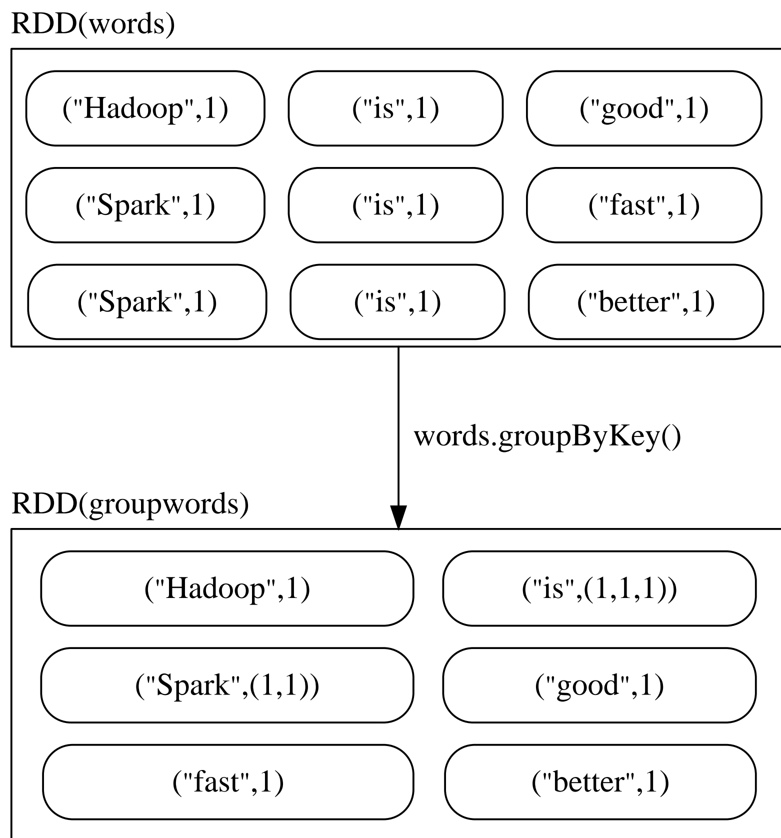


图　groupByKey()操作实例执行过程示意图

5.1.2 RDD操作

1. 转换操作

•reduceByKey(func)

reduceByKey(func)应用于(K,V)键值对的数据集时，返回一个新的(K, V)形式的数据集，其中的每个值是将每个key传递到函数func中进行聚合后得到的结果

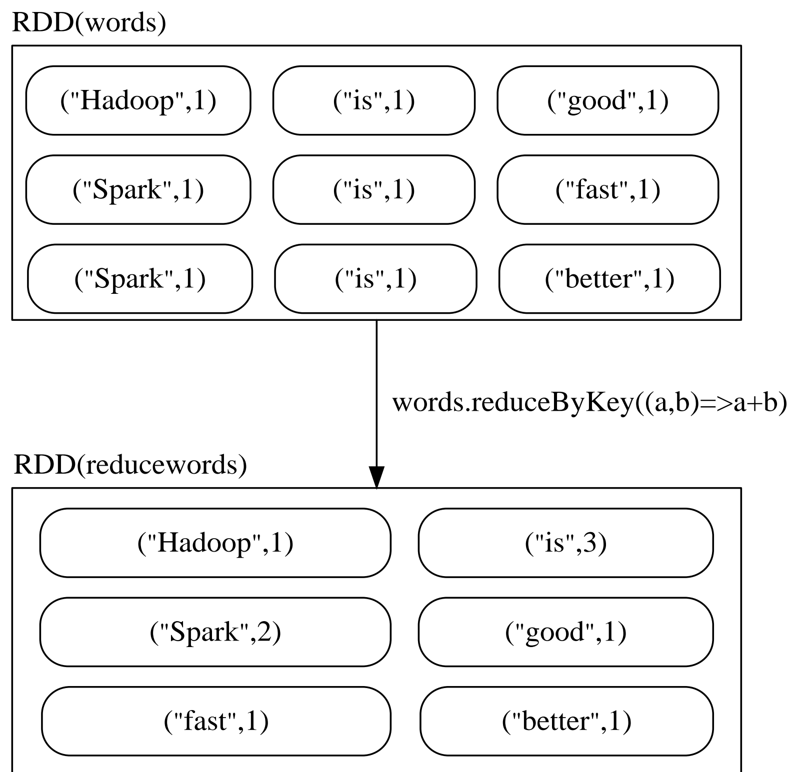


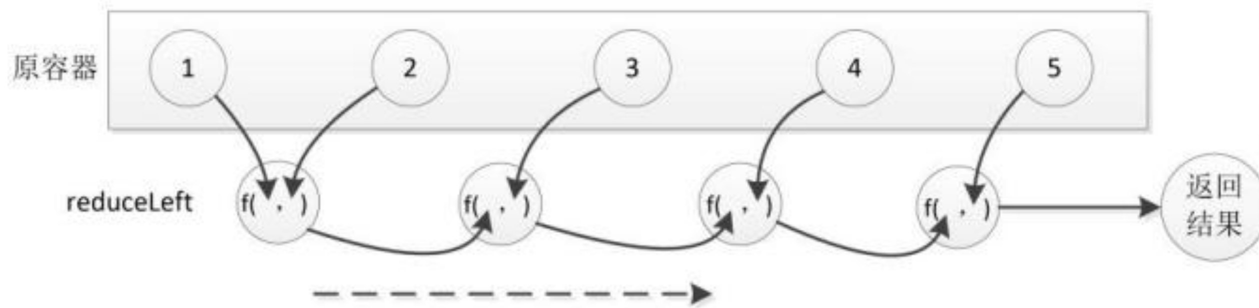
图 reduceByKey()操作实例执行过程示意图

5.1.2 RDD操作

1. 转换操作

•reduceByKey(func)

rdd.reduceByKey((a,b)=>a+b)



<“spark”,<1,1,1>>

5.1.2 RDD操作

2. 行动操作

行动操作是真正触发计算的地方。Spark程序执行到行动操作时，才会执行真正的计算，从文件中加载数据，完成一次又一次转换操作，最终，完成行动操作得到结果。

表 常用的RDD行动操作API

操作	含义
count()	返回数据集中的元素个数
collect()	以数组的形式返回数据集中的所有元素
first()	返回数据集中的第一个元素
take(n)	以数组的形式返回数据集中的前n个元素
reduce(func)	通过函数func（输入两个参数并返回一个值）聚合数据集中的元素
foreach(func)	将数据集中的每个元素传递到函数func中运行

5.1.2 RDD操作

```
scala> val  rdd=sc.parallelize(Array(1,2,3,4,5))
rdd: org.apache.spark.rdd.RDD[Int]=ParallelCollectionRDD[1] at parallelize at <console>:24
scala> rdd.count()
res0: Long = 5
scala> rdd.first()
res1: Int = 1
scala> rdd.take(3)
res2: Array[Int] = Array(1,2,3)
scala> rdd.reduce((a,b)=>a+b)
res3: Int = 15
scala> rdd.collect()
res4: Array[Int] = Array(1,2,3,4,5)
scala> rdd.foreach(elem=>println(elem))
1
2
3
4
5
```

5.1.3 惰性机制

所谓的“惰性机制”是指，整个转换过程只是记录了转换的轨迹，并不会发生真正的计算，只有遇到行动操作时，才会触发“从头到尾”的真正的计算

这里给出一段简单的语句来解释Spark的惰性机制

```
scala> val  lines = sc.textFile("data.txt")
scala> val  lineLengths = lines.map(s => s.length)
scala> val  totalLength = lineLengths.reduce((a, b) => a + b)
```

5.1.4 持久化

在Spark中，RDD采用惰性求值的机制，每次遇到行动操作，都会从头开始执行计算。每次调用行动操作，都会触发一次从头开始的计算。这对于迭代计算而言，代价是很大的，迭代计算经常需要多次重复使用同一组数据

下面就是多次计算同一个RDD的例子：

```
scala> val  list = List("Hadoop","Spark","Hive")
list: List[String] = List(Hadoop, Spark, Hive)
scala> val  rdd = sc.parallelize(list)
rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[22] at parallelize at <console>:29
scala> println(rdd.count())  //行动操作，触发一次真正从头到尾的计算
3
scala> println(rdd.collect().mkString(","))  //行动操作，触发一次真正从头到尾的计算
Hadoop,Spark,Hive
```

5.1.3 持久化

- 可以通过持久化（缓存）机制避免这种重复计算的开销
- 可以使用persist()方法对一个RDD标记为持久化
- 之所以说“标记为持久化”，是因为出现persist()语句的地方，并不会马上计算生成RDD并把它持久化，而是要等到遇到第一个行动操作触发真正计算以后，才会把计算结果进行持久化
- 持久化后的RDD将会被保留在计算节点的内存中被后面的行动操作重复使用

5.1.3 持久化

persist()的圆括号中包含的是持久化级别参数：

- persist(MEMORY_ONLY)：表示将RDD作为反序列化的对象存储于JVM中，如果内存不足，就要按照LRU原则替换缓存中的内容
- persist(MEMORY_AND_DISK)表示将RDD作为反序列化的对象存储在JVM中，如果内存不足，超出的分区将会被存放在硬盘上
- 一般而言，使用cache()方法时，会调用persist(MEMORY_ONLY)
- 可以使用unpersist()方法手动地把持久化的RDD从缓存中移除

5.1.3 持久化

针对上面的实例，增加持久化语句以后的执行过程如下：

```
scala> val  list = List("Hadoop","Spark","Hive")
list: List[String] = List(Hadoop, Spark, Hive)
scala> val  rdd = sc.parallelize(list)
rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[22] at
parallelize at <console>:29
scala> rdd.cache()  //会调用persist(MEMORY_ONLY)，但是，语句执行到这
里，并不会缓存rdd，因为这时rdd还没有被计算生成
scala> println(rdd.count()) //第一次行动操作，触发一次真正从头到尾的计算，
这时上面的rdd.cache()才会被执行，把这个rdd放到缓存中
3
scala> println(rdd.collect().mkString(",")) //第二次行动操作，不需要触发从头
到尾的计算，只需要重复使用上面缓存中的rdd
Hadoop,Spark,Hive
```

5.1.4 分区

RDD是弹性分布式数据集，通常RDD很大，会被分成很多个分区，分别保存在不同的节点上

1.分区的作用

（1）增加并行度

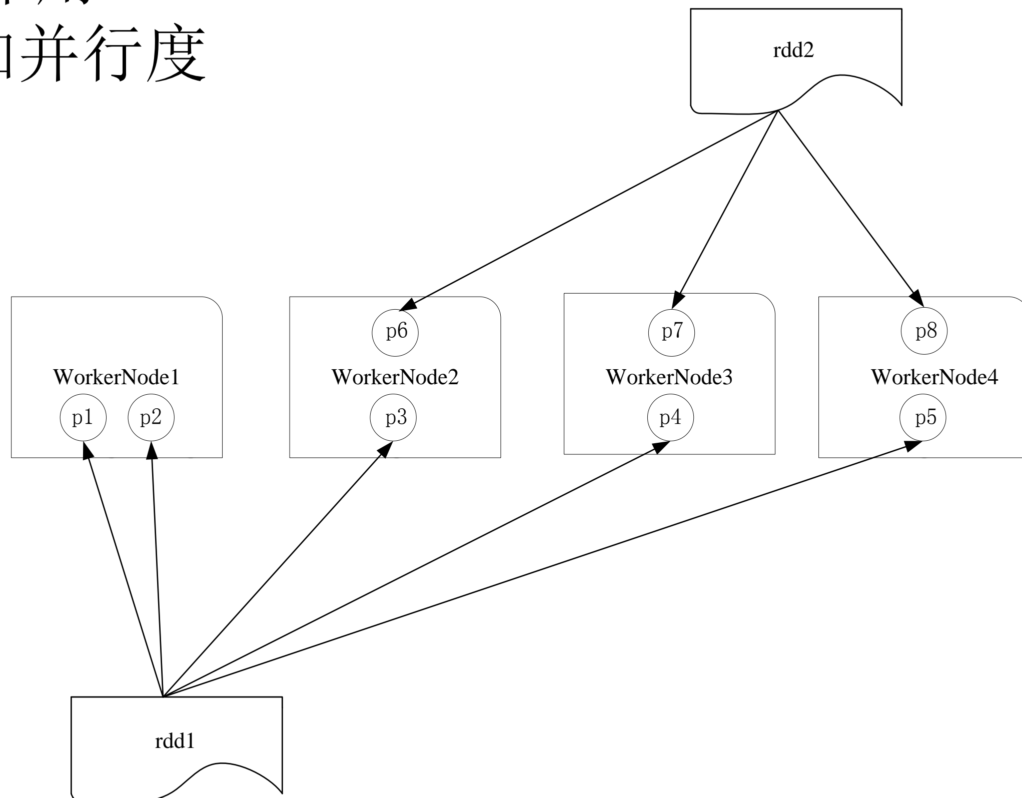


图 RDD分区被保存到不同节点上

5.1.4 分区

1.分区的作用

（2）减少通信开销

UserData（UserId，UserInfo）
Events（UserID，LinkInfo）
UserData 和Events 表进行连接操作，获得（UserID，UserInfo，LinkInfo）

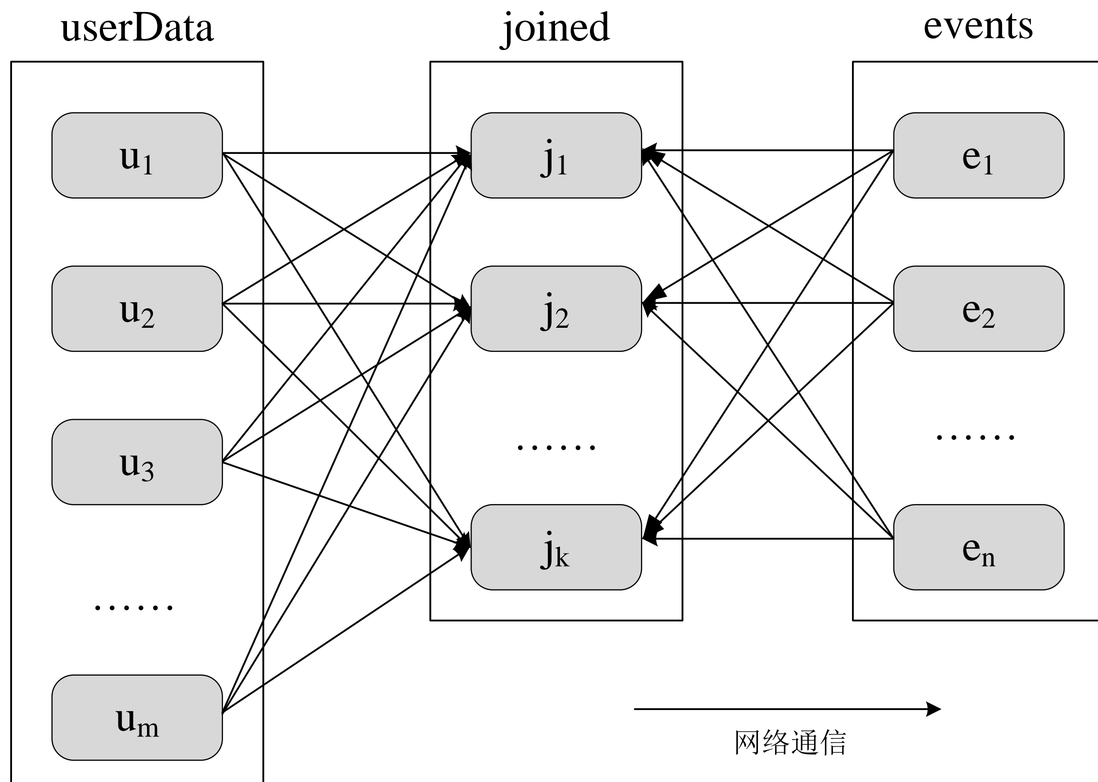


图 未分区时对UserData和Events两个表进行连接操作

5.1.4 分区

1.分区的作用

（2）减少通信开销

UserData（UserId，UserInfo）
Events（UserID，LinkInfo）
UserData 和Events 表进行连接操作，获得（UserID，UserInfo，LinkInfo）

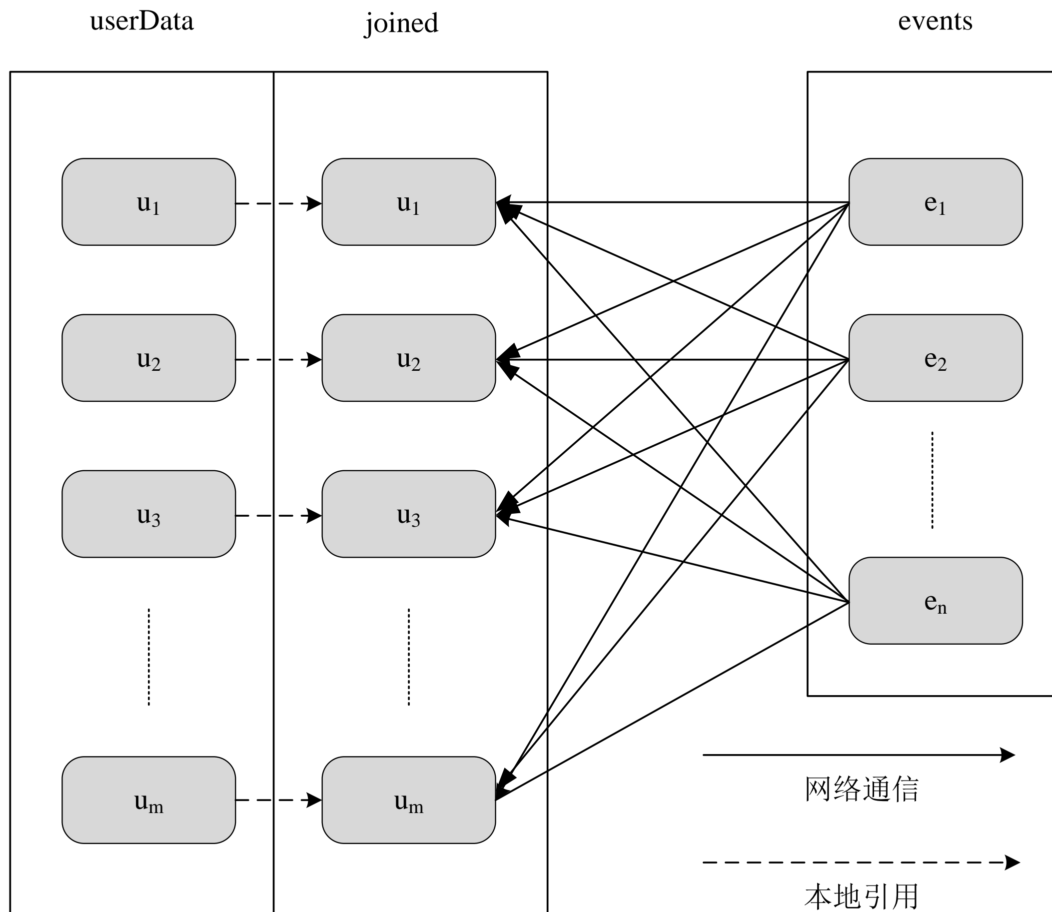


图 采用分区以后对UserData和Events两个表进行连接操作

5.1.4 分区

2.RDD分区原则

RDD分区的一个原则是使得分区的个数尽量等于集群中的CPU核心（core）数目

对于不同的Spark部署模式而言（本地模式、Standalone模式、YARN模式、Mesos模式），都可以通过设置spark.default.parallelism这个参数的值，来配置默认的分区数目，一般而言：
*本地模式：默认为本地机器的CPU数目，若设置了local[N],则默认为N
*Apache Mesos：默认的分区数为8
*Standalone或YARN：在“集群中所有CPU核心数目总和”和“2”二者中取较大值作为默认值

5.1.4 分区

3.设置分区的个数

（1）创建RDD时手动指定分区个数

在调用textFile()和parallelize()方法的时候手动指定分区个数即可，语法格式如下：

sc.textFile(path, partitionNum)

其中，path参数用于指定要加载的文件的地址，partitionNum参数用于指定分区个数。

```
scala> val  array = Array(1,2,3,4,5)
scala> val  rdd = sc.parallelize(array,2)  //设置两个分区
```

5.1.4 分区

3.设置分区的个数

（2）使用reparititon方法重新设置分区个数

通过转换操作得到新 RDD 时，直接调用 repartition 方法即可。例如：

```
scala> val  data = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt",2)
data: org.apache.spark.rdd.RDD[String] =
file:///usr/local/spark/mycode/rdd/word.txt MapPartitionsRDD[12] at textFile at
<console>:24
scala> data.partitions.size  //显示data这个RDD的分区数量
res2: Int=2
scala> val  rdd = data.repartition(1)  //对data这个RDD进行重新分区
rdd: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[11] at repartition
at :26
scala> rdd.partitions.size
res4: Int = 1
```

5.1.4 分区

4.自定义分区方法

Spark提供了自带的HashPartitioner（哈希分区）与RangePartitioner（区域分区），能够满足大多数应用场景的需求。与此同时，Spark也支持自定义分区方式，即通过提供一个自定义的Partitioner对象来控制RDD的分区方式，从而利用领域知识进一步减少通信开销

要实现自定义分区，需要定义一个类，这个自定义类需要继承org.apache.spark.Partitioner类，并实现下面三个方法：

- numPartitions: Int 返回创建出来的分区数
- getPartition(key: Any): Int 返回给定键的分区编号（0到numPartitions-1）
- equals() Java判断相等性的标准方法

5.1.4 分区

实例：根据key值的最后一位数字，写到不同的文件

例如：
10写入到part-00000
11写入到part-00001
.
.
.
19写入到part-00009

5.1.4 分区

```
import org.apache.spark.{Partitioner, SparkContext, SparkConf}
//自定义分区类，需要继承org.apache.spark.Partitioner类
class MyPartitioner(numParts:Int) extends Partitioner{
  //覆盖分区数
  override def numPartitions: Int = numParts
  //覆盖分区号获取函数
  override def getPartition(key: Any): Int = {
    key.toString.toInt%10
  }
}
object TestPartitioner {
  def main(args: Array[String]) {
    val conf=new SparkConf()
    val sc=new SparkContext(conf)
    //模拟5个分区的数据
    val data=sc.parallelize(1 to 10,5)
    //根据尾号转变为10个分区，分别写到10个文件
    data.map((_,1)).partitionBy(new
MyPartitioner(10)).map(_._1).saveAsTextFile("file:///usr/local/spark/mycode/rdd/partitioner")
  }
}
```

5.1.5 一个综合实例

假设有一个本地文件word.txt，里面包含了很多行文本，每行文本由多个单词构成，单词之间用空格分隔。可以使用如下语句进行词频统计（即统计每个单词出现的次数）：

```
scala> val  lines = sc.  //代码一行放不下，可以在圆点后回车，在下行继续输入
|  textFile("file:///usr/local/spark/mycode/wordcount/word.txt")
scala> val wordCount = lines.flatMap(line => line.split(" ")).
|  map(word => (word, 1)).reduceByKey((a, b) => a + b)
scala> wordCount.collect()
scala> wordCount.foreach(println)
```

5.1.5 一个综合实例

```
scala> val  lines = sc.  //代码一行放不下，可以在圆点后回车，在下行继续输入
|  textFile("file:///usr/local/spark/mycode/wordcount/word.txt")
scala> val wordCount = lines.flatMap(line => line.split(" ")).
|  map(word => (word, 1)).reduceByKey((a, b) => a + b)
scala> wordCount.collect()
scala> wordCount.foreach(println)
```

5.1.5 一个综合实例

在实际应用中，单词文件可能非常大，会被保存到分布式文件系统HDFS中，Spark和Hadoop会统一部署在一个集群上

图 在一个集群中同时部署Hadoop和Spark

5.1.5 一个综合实例

Spark Master

a:4, b:5
c:3, d:1
e:2

WORD COUNT

countMap.reduce

Slave

Slave

Slave

COUNT MAP RDD

a:1,b:3
c:1

a:1,e:2
c:1,d:1

a:2,b:2
c:1

words.mapPartitions

WORDS RDD

a, b, c, b, b

a, d, c, e, e

a, a, c, b, b

lines.flatMap

LINES RDD

“a b c”
“d b”

“a d c”
“e e”

“a a c”
“b b”

sc.textFile

Spark Worker

Spark Worker

Spark Worker

/FILE.TXT

HDFS DN

HDFS DN

HDFS DN

图 在集群中执行词频统计过程示意图

5.2 键值对RDD

5.2.1 键值对RDD的创建
5.2.2 常用的键值对RDD转换操作
5.2.3 一个综合实例

5.2.1 键值对RDD的创建

（1）第一种创建方式：从文件中加载

可以采用多种方式创建Pair RDD，其中一种主要方式是使用map()函数来实现

```
scala> val lines = sc.textFile("file:///usr/local/spark/mycode/pairrdd/word.txt")
lines: org.apache.spark.rdd.RDD[String] =
file:///usr/local/spark/mycode/pairrdd/word.txt MapPartitionsRDD[1] at
textFile at <console>:27
scala> val pairRDD = lines.flatMap(line => line.split(" ")).map(word =>
(word,1))
pairRDD: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at
map at <console>:29
scala> pairRDD.foreach(println)
(i,1)
(love,1)
(hadoop,1)
……
```

5.2.1键值对RDD的创建

（2）第二种创建方式：通过并行集合（数组）创建**RDD**

```
scala> val list = List("Hadoop","Spark","Hive","Spark")
list: List[String] = List(Hadoop, Spark, Hive, Spark)

scala> val rdd = sc.parallelize(list)
rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[11] at
parallelize at <console>:29

scala> val pairRDD = rdd.map(word => (word,1))
pairRDD: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[12] at
map at <console>:31

scala> pairRDD.foreach(println)
(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)
```

5.2.2 常用的键值对RDD转换操作

- **reduceByKey(func)**
- **groupByKey()**
- **keys**
- **values**
- **sortByKey()**
- **mapValues(func)**
- **join**
- **combineByKey**

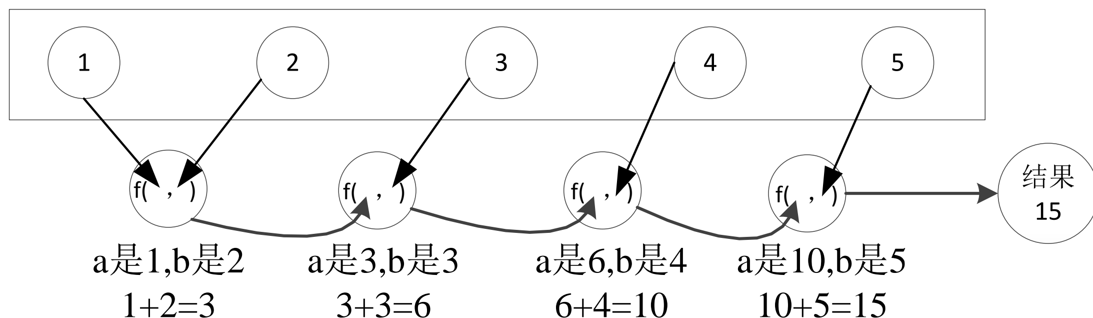
5.2.2 常用的键值对RDD转换操作

•reduceByKey(func)

reduceByKey(func)的功能是，使用func函数合并具有相同键的值

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

```
scala> pairRDD.reduceByKey((a,b)=>a+b).foreach(println)
(Spark,2)
(Hive,1)
(Hadoop,1)
```



5.2.2 常用的键值对RDD转换操作

•groupByKey()

groupByKey()的功能是，对具有相同键的值进行分组

比如，对四个键值对("spark",1)、("spark",2)、("hadoop",3)和("hadoop",5)，采用groupByKey()后得到的结果是：("spark",(1,2))和("hadoop",(3,5))

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

```
scala> pairRDD.groupByKey()
res15: org.apache.spark.rdd.RDD[(String, Iterable[Int])] = ShuffledRDD[15]
at groupByKey at <console>:34
```

5.2.2 常用的键值对RDD转换操作

reduceByKey和groupByKey的区别

- reduceByKey用于对每个key对应的多个value进行merge操作，最重要的是它能够在本地先进行merge操作，并且merge操作可以通过函数自定义

- groupByKey也是对每个key进行操作，但只生成一个sequence，groupByKey本身不能自定义函数，需要先用groupByKey生成RDD，然后才能对此RDD通过map进行自定义函数操作

5.2.2 常用的键值对RDD转换操作

reduceByKey和groupByKey的区别

```
scala>  val words = Array("one", "two", "two", "three", "three", "three")

scala>  val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))

scala>  val wordCountsWithReduce = wordPairsRDD.reduceByKey(_ + _)

scala>
val wordCountsWithGroup = wordPairsRDD.groupByKey().map(t => (t._1, t._2.sum))
```

上面得到的wordCountsWithReduce和wordCountsWithGroup是完全一样的，但是，它们的内部运算过程是不同的

5.2.2 常用的键值对RDD转换操作

•keys

keys只会把Pair RDD中的key返回形成一个新的RDD

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

```
scala> pairRDD.keys
res17: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[17] at keys
at <console>:34
scala> pairRDD.keys.foreach(println)
Hadoop
Spark
Hive
Spark
```

5.2.2 常用的键值对RDD转换操作

- **values**

values只会把Pair RDD中的value返回形成一个新的RDD。

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

```
scala> pairRDD.values
res0: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[2] at values at <console>:34
scala> pairRDD.values.foreach(println)
1
1
1
1
```

5.2.2 常用的键值对RDD转换操作

- **sortByKey()**

sortByKey()的功能是返回一个根据键排序的RDD

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

```
scala> pairRDD.sortByKey()
res0: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[2] at
sortByKey at <console>:34
scala> pairRDD.sortByKey().foreach(println)
(Hadoop,1)
(Hive,1)
(Spark,1)
(Spark,1)
```

5.2.2 常用的键值对RDD转换操作

- **sortByKey()和sortBy()**

```
scala> val d1 =
sc.parallelize(Array((“c",8),(“b“,25),(“c“,17),(“a“,42),(“b“,4),(“d“,9),(“e“,17),(“c“
,2),(“f“,29),(“g“,21),(“b“,9)))
scala> d1.reduceByKey(_+_).sortByKey(false).collect
res2: Array[(String, Int)] = Array((g,21),(f,29),(e,17),(d,9),(c,27),(b,38),(a,42))
```

```
scala> val d2 =
sc.parallelize(Array((“c",8),(“b“,25),(“c“,17),(“a“,42),(“b“,4),(“d“,9),(“e“,17),(“c“
,2),(“f“,29),(“g“,21),(“b“,9)))
scala> d2.reduceByKey(_+_).sortBy(_._2,false).collect
res4: Array[(String, Int)] = Array((a,42),(b,38),(f,29),(c,27),(g,21),(e,17),(d,9))
```

5.2.2 常用的键值对RDD转换操作

- **mapValues(func)**

对键值对RDD中的每个value都应用一个函数，但是，key不会发生变化

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

```
scala> pairRDD.mapValues(x => x+1)
res2: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[4] at
mapValues at <console>:34
scala> pairRDD.mapValues(x => x+1).foreach(println)
(Hadoop,2)
(Spark,2)
(Hive,2)
(Spark,2)
```

5.2.2 常用的键值对RDD转换操作

•**join**

join就表示内连接。对于内连接，对于给定的两个输入数据集(K,V1)和(K,V2)，只有在两个数据集中都存在的key才会被输出，最终得到一个(K,(V1,V2))类型的数据集。

```
scala> val pairRDD1 = sc.parallelize(Array(("spark",1),("spark",2),("hadoop",3),("hadoop",5)))
pairRDD1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[24] at parallelize at
<console>:27

scala> val pairRDD2 = sc.parallelize(Array(("spark","fast")))
pairRDD2: org.apache.spark.rdd.RDD[(String, String)] = ParallelCollectionRDD[25] at parallelize at
<console>:27

scala> pairRDD1.join(pairRDD2)
res9: org.apache.spark.rdd.RDD[(String, (Int, String))] = MapPartitionsRDD[28] at join at <console>:32

scala> pairRDD1.join(pairRDD2).foreach(println)
(spark,(1,fast))
(spark,(2,fast))
```

5.2.3 一个综合实例

一个综合实例

题目：给定一组键值对("spark",2),("hadoop",6),("hadoop",4),("spark",6)，键值对的key表示图书名称，value表示某天图书销量，请计算每个键对应的平均值，也就是计算每种图书的每天平均销量。

```
scala> val rdd =
sc.parallelize(Array(("spark",2),("hadoop",6),("hadoop",4),("spark",6)))
rdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[38] at
parallelize at <console>:27

scala> rdd.mapValues(x => (x,1)).reduceByKey((x,y) => (x._1+y._1,x._2 +
y._2)).mapValues(x => (x._1 / x._2)).collect()
res22: Array[(String, Int)] = Array((spark,4), (hadoop,5))
```

5.2.3 一个综合实例

```
scala> val rdd =
sc.parallelize(Array(("spark",2),("hadoop",6),("hadoop",4),("spark",6)))
rdd: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[38] at
parallelize at <console>:27

scala> rdd.mapValues(x => (x,1)).reduceByKey((x,y) => (x._1+y._1,x._2 +
y._2)).mapValues(x => (x._1 / x._2)).collect()
res22: Array[(String, Int)] = Array((spark,4), (hadoop,5))
```

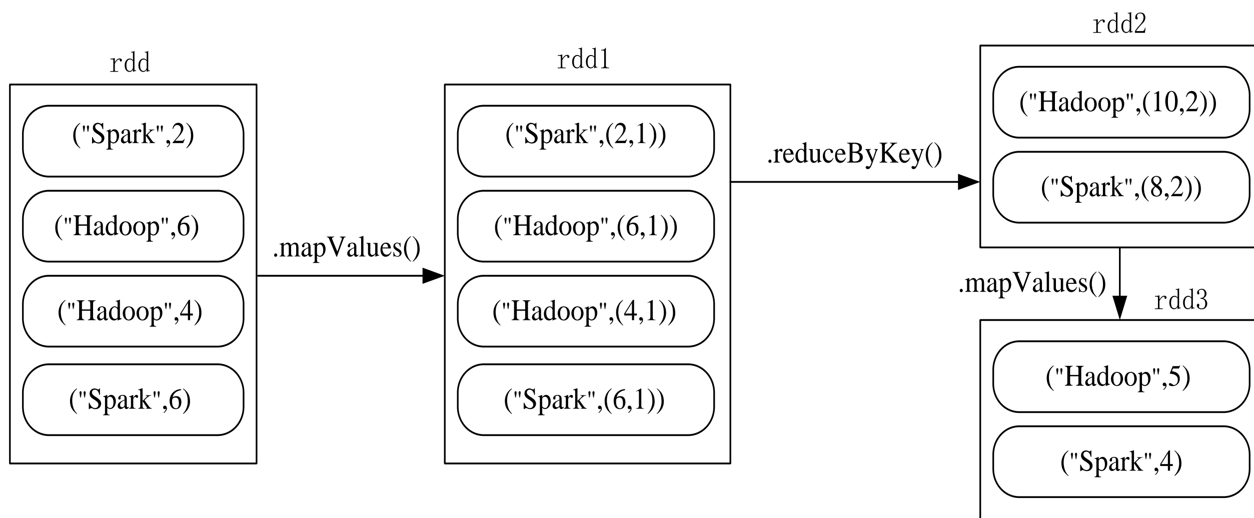


图 计算图书平均销量过程示意图

5.3 数据读写

5.3.1 文件数据读写
5.3.2 读写HBase数据

5.3.1 文件数据读写

1. 本地文件系统的数据读写
2. 分布式文件系统HDFS的数据读写
3. JSON文件的数据读写

5.3.1 文件数据读写

1.本地文件系统的数据读写

（1）从文件中读取数据创建RDD

```
scala> val  textFile = sc.
|  textFile("file:///usr/local/spark/mycode/wordcount/word.txt")
```

因为Spark采用了惰性机制，在执行转换操作的时候，即使输入了错误的语句，spark-shell也不会马上报错（假设word123.txt不存在）

```
scala> val  textFile = sc.
|  textFile("file:///usr/local/spark/mycode/wordcount/word123.txt")
```

5.3.1 文件数据读写

1.本地文件系统的数据读写

（2）把RDD写入到文本文件中

```
scala> val  textFile = sc.
|  textFile("file:///usr/local/spark/mycode/wordcount/word.txt")
scala> textFile.
|  saveAsTextFile("file:///usr/local/spark/mycode/wordcount/writeback")
```

```
$ cd /usr/local/spark/mycode/wordcount/writeback/
$ ls
```

```
part-00000
_SUCCESS
```

如果想再次把数据加载在RDD中，只要使用writeback这个目录即可，如下：

```
scala> val textFile =
sc.textFile("file:///usr/local/spark/mycode/wordcount/writeback")
```

5.3.1 文件数据读写

2.分布式文件系统HDFS的数据读写

从分布式文件系统HDFS中读取数据，也是采用textFile()方法，可以为textFile()方法提供一个HDFS文件或目录地址，如果是一个文件地址，它会加载该文件，如果是一个目录地址，它会加载该目录下的所有文件的数据

```
scala> val  textFile = sc.textFile("hdfs://localhost:9000/user/hadoop/word.txt")
scala> textFile.first()
```

如下三条语句都是等价的：

```
scala> val textFile = sc.textFile("hdfs://localhost:9000/user/hadoop/word.txt")
scala> val textFile = sc.textFile("/user/hadoop/word.txt")
scala> val textFile = sc.textFile("word.txt")
```

同样，可以使用saveAsTextFile()方法把RDD中的数据保存到HDFS文件中，命令如下：

```
scala> textFile.saveAsTextFile("writeback")
```

5.3.1 文件数据读写

3. JSON文件的读取

- JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式
- Spark提供了一个JSON样例数据文件，存放在“/usr/local/spark/examples/src/main/resources/people.json”中

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

5.3.1 文件数据读写

3. JSON文件的读取

把本地文件系统中的people.json文件加载到RDD中：

```
scala> val  jsonStr = sc.
|  textFile("file:///usr/local/spark/examples/src/main/resources/people.json")
scala> jsonStr.foreach(println)
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

5.3.1 文件数据读写

任务：编写程序完成对JSON数据的解析工作

- Scala中有一个自带的JSON库——scala.util.parsing.json.JSON，可以实现对JSON数据的解析

- JSON.parseFull(jsonString:String)函数，以一个JSON字符串作为输入并进行解析，如果解析成功则返回一个Some(map: Map[String, Any])，如果解析失败则返回None

5.3.1 文件数据读写

在JSONRead.scala代码文件中输入以下内容：

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import scala.util.parsing.json.JSON
object JSONRead {
    def main(args: Array[String]) {
        val inputFile = "file:///usr/local/spark/examples/src/main/resources/people.json"
        val conf = new SparkConf().setAppName("JSONRead")
        val sc = new SparkContext(conf)
        val jsonStrs = sc.textFile(inputFile)
        val result = jsonStrs.map(s => JSON.parseFull(s))
        result.foreach( {r => r match {
                        case Some(map: Map[String, Any]) => println(map)
                        case None => println("Parsing failed")
                        case other => println("Unknown data structure: " + other)
                }
        }
        )
    }
}
```

5.3.1 文件数据读写

•将整个应用程序打包成 JAR包
•通过 spark-submit 运行程序

```
$ /usr/local/spark/bin/spark-submit   \
> --class "JSONRead”   \
> /usr/local/spark/mycode/json/target/scala-2.11/json-project_2.11-1.0.jar
```

执行后可以在屏幕上的大量输出信息中找到如下结果：

```
Map(name -> Michael)
Map(name -> Andy, age -> 30.0)
Map(name -> Justin, age -> 19.0)
```

5.3.2 读写HBase数据

0. HBase简介
1. 创建一个HBase表
2. 配置Spark
3. 编写程序读取HBase数据
4. 编写程序向HBase写入数据

5.3.2 读写HBase数据

0. HBase简介

- HBase是Google BigTable的开源实现

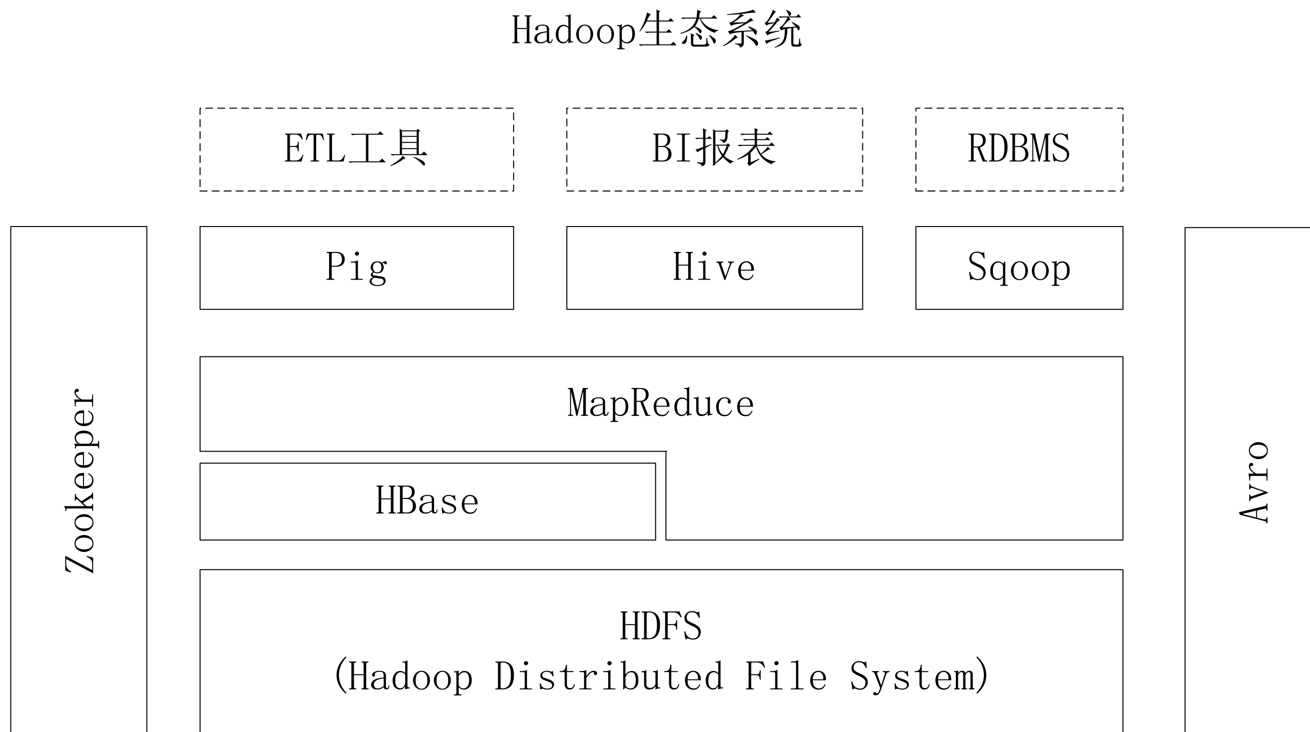


图 Hadoop生态系统中HBase与其他部分的关系

5.3.2 读写HBase数据

- HBase是一个稀疏、多维度、排序的映射表，这张表的索引是行键、列族、列限定符和时间戳
- 每个值是一个未经解释的字符串，没有数据类型
- 用户在表中存储数据，每一行都有一个可排序的行键和任意多的列
- 表在水平方向由一个或者多个列族组成，一个列族中可以包含任意多个列，同一个列族里面的数据存储在一起
- 列族支持动态扩展，可以很轻松地添加一个列族或列，无需预先定义列的数量以及类型，所有列均以字符串形式存储，用户需要自行进行数据类型转换
- HBase中执行更新操作时，并不会删除数据旧的版本，而是生成一个新的版本，旧有的版本仍然保留（这是和HDFS只允许追加不允许修改的特性相关的）

5.3.2 读写HBase数据

- 表：HBase采用表来组织数据，表由行和列组成，列划分为若干个列族
- 行：每个HBase表都由若干行组成，每个行由行键（row key）来标识。
- 列族：一个HBase表被分组成许多“列族”（Column Family）的集合，它是基本的访问控制单元
- 列限定符：列族里的数据通过列限定符（或列）来定位
- 单元格：在HBase表中，通过行、列族和列限定符确定一个“单元格”（cell），单元格中存储的数据没有数据类型，总被视为字节数组byte[]
- 时间戳：每个单元格都保存着同一份数据的多个版本，这些版本采用时间戳进行索引

列限定符　列族

	Info		
	name	major	email
201505001	Luo Min	Math	luo@qq.com
201505002	Liu Jun	Math	liu@qq.com
201505003	Xie You	Math	xie@qq.com you@163.com

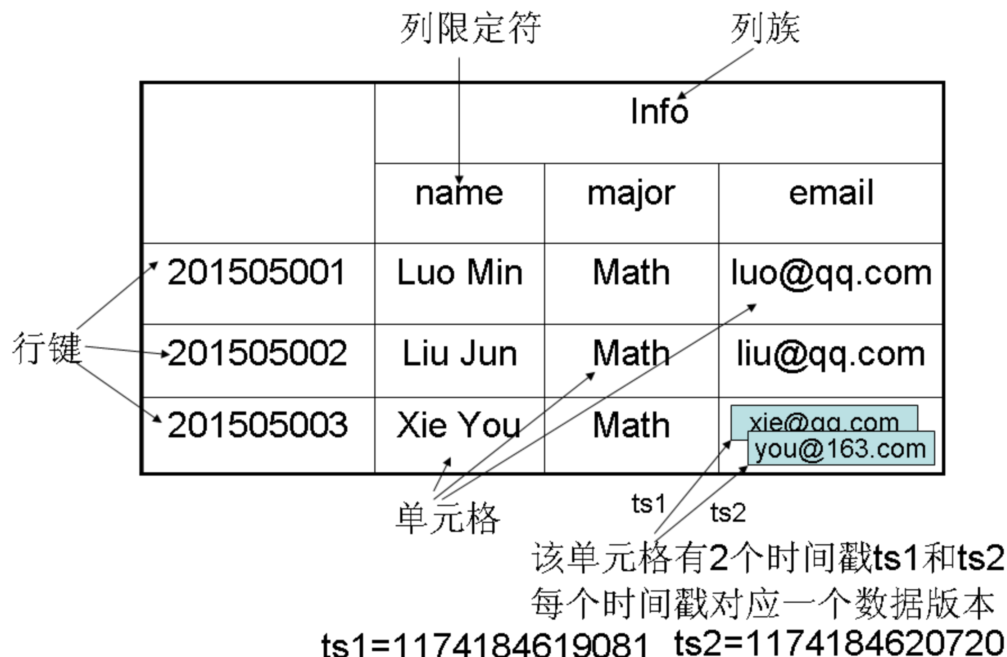
行键　单元格　ts1　ts2

该单元格有2个时间戳ts1和ts2
每个时间戳对应一个数据版本
ts1=1174184619081 ts2=1174184620720

5.3.2 读写HBase数据

- HBase中需要根据行键、列族、列限定符和时间戳来确定一个单元格，因此，可以视为一个“四维坐标”，即[行键, 列族, 列限定符, 时间戳]

键	值
[“201505003”, “Info”, “email”, 1174184619081]	“xie@qq.com”
[“201505003”, “Info”, “email”, 1174184620720]	“you@163.com”



5.3.2 读写HBase数据

表　HBase数据的概念视图

行键	时间戳	列族contents	列族anchor
"com.cnn.www"	t5		anchor:cnnsi.com=”CNN”
	t4		anchor:my.look.ca="CNN.com"
	t3	contents:html="<html>..."	
	t2	contents:html="<html>..."	
	t1	contents:html="<html>..."	

5.3.2 读写HBase数据

表　HBase数据的物理视图

列族contents

行键	时间戳	列族contents
"com.cnn.www"	t3	contents:html="<html>..."
	t2	contents:html="<html>..."
	t1	contents:html="<html>..."

列族anchor

行键	时间戳	列族anchor
"com.cnn.www"	t5	anchor:cnnsi.com=”CNN”
	t4	anchor:my.look.ca="CNN.com"

5.3.2 读写HBase数据

1. 创建一个HBase表

首先，请参照厦门大学数据库实验室博客完成HBase的安装（伪分布式模式）：

http://dblab.xmu.edu.cn/blog/install-hbase/

因为HBase是伪分布式模式，需要调用HDFS，所以，请首先在终端中输入下面命令启动Hadoop：

```
$ cd /usr/local/hadoop
$ ./sbin/start-all.sh
```

下面就可以启动HBase，命令如下：

```
$ cd /usr/local/hbase
$ ./bin/start-hbase.sh //启动HBase
$ ./bin/hbase shell  //启动hbase shell
```

如果里面已经有一个名称为student的表，请使用如下命令删除：

```
hbase> disable 'student'
hbase> drop 'student'
```

5.3.2 读写HBase数据

1. 创建一个HBase表

下面创建一个student表，要在这个表中录入如下数据：

```
+------+----------+--------+------+
| id   | name     | gender | age  |
+------+----------+--------+------+
|    1 | Xueqian  | F      |   23 |
|    2 | Weiliang | M      |   24 |
+------+----------+--------+------+
```

```
hbase>  create 'student','info'
```

```
//首先录入student表的第一个学生记录
hbase> put 'student','1','info:name','Xueqian'
hbase> put 'student','1','info:gender','F'
hbase> put 'student','1','info:age','23'
//然后录入student表的第二个学生记录
hbase> put 'student','2','info:name','Weiliang'
hbase> put 'student','2','info:gender','M'
hbase> put 'student','2','info:age','24'
```

5.3.2 读写HBase数据

2. 配置Spark

把HBase的lib目录下的一些jar文件拷贝到Spark中，这些都是编程时需要引入的jar包，需要拷贝的jar文件包括：所有hbase开头的jar文件、guava-12.0.1.jar、htrace-core-3.1.0-incubating.jar和protobuf-java-2.5.0.jar

执行如下命令：

```
$ cd /usr/local/spark/jars
$ mkdir hbase
$ cd hbase
$ cp /usr/local/hbase/lib/hbase*.jar ./
$ cp /usr/local/hbase/lib/guava-12.0.1.jar ./
$ cp /usr/local/hbase/lib/htrace-core-3.1.0-incubating.jar ./
$ cp /usr/local/hbase/lib/protobuf-java-2.5.0.jar ./
```

5.3.2 读写HBase数据

3. 编写程序读取HBase数据

如果要让Spark读取HBase，就需要使用SparkContext提供的newAPIHadoopRDD这个API将表的内容以RDD的形式加载到Spark中。

```
import org.apache.hadoop.conf.Configuration
import org.apache.hadoop.hbase._
import org.apache.hadoop.hbase.client._
import org.apache.hadoop.hbase.mapreduce.TableInputFormat
import org.apache.hadoop.hbase.util.Bytes
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

//剩余代码见下一页
```

5.3.2 读写HBase数据

在SparkOperateHBase.scala文件中输入以下代码：

```
object SparkOperateHBase {
def main(args: Array[String]) {
    val conf = HBaseConfiguration.create()
    val sc = new SparkContext(new SparkConf())
    //设置查询的表名
    conf.set(TableInputFormat.INPUT_TABLE, "student")
    val stuRDD = sc.newAPIHadoopRDD(conf, classOf[TableInputFormat],
  classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable],
  classOf[org.apache.hadoop.hbase.client.Result])
    val count = stuRDD.count()
    println("Students RDD Count:" + count)
    stuRDD.cache()
    //遍历输出
    stuRDD.foreach({ case (_,result) =>
        val key = Bytes.toString(result.getRow)
        val name = Bytes.toString(result.getValue("info".getBytes,"name".getBytes))
        val gender = Bytes.toString(result.getValue("info".getBytes,"gender".getBytes))
        val age = Bytes.toString(result.getValue("info".getBytes,"age".getBytes))
        println("Row key:"+key+" Name:"+name+" Gender:"+gender+" Age:"+age)
    })
}
}
```

5.3.2 读写HBase数据

在simple.sbt中录入下面内容：

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.11.8"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.1.0"
libraryDependencies += "org.apache.hbase" % "hbase-client" % "1.1.5"
libraryDependencies += "org.apache.hbase" % "hbase-common" % "1.1.5"
libraryDependencies += "org.apache.hbase" % "hbase-server" % "1.1.5"
```

采用sbt打包，通过 spark-submit 运行程序

```
$ /usr/local/spark/bin/spark-submit  \
>--driver-class-path /usr/local/spark/jars/hbase/*:/usr/local/hbase/conf  \
>--class "SparkOperateHBase"  \
>/usr/local/spark/mycode/hbase/target/scala-2.11/simple-project_2.11-1.0.jar
```

必须使用“--driver-class-path”参数指定依赖JAR包的路径，而且必须把“/usr/local/hbase/conf”也加到路径中

5.3.2 读写HBase数据

执行后得到如下结果：

```
Students RDD Count:2
Row key:1 Name:Xueqian Gender:F Age:23
Row key:2 Name:Weiliang Gender:M Age:24
```

5.3.2 读写HBase数据

4. 编写程序向HBase写入数据

下面编写应用程序把表中的两个学生信息插入到HBase的student表中

表　向student表中插入的新数据

id	info		
	name	gender	age
3	Rongcheng	M	26
4	Guanhua	M	27

5.3.2 读写HBase数据

在SparkWriteHBase.scala文件中输入下面代码：

```
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.mapreduce.TableOutputFormat
import org.apache.spark._
import org.apache.hadoop.mapreduce.Job
import org.apache.hadoop.hbase.io.ImmutableBytesWritable
import org.apache.hadoop.hbase.client.Result
import org.apache.hadoop.hbase.client.Put
import org.apache.hadoop.hbase.util.Bytes

//剩余代码见下一页
```

5.3.2 读写HBase数据

在SparkWriteHBase.scala文件中输入下面代码：

```
object SparkWriteHBase {
  def main(args: Array[String]): Unit = {
    val sparkConf = new SparkConf().setAppName("SparkWriteHBase").setMaster("local")
    val sc = new SparkContext(sparkConf)
    val tablename = "student"
    sc.hadoopConfiguration.set(TableOutputFormat.OUTPUT_TABLE, tablename)
    val job = new Job(sc.hadoopConfiguration)
    job.setOutputKeyClass(classOf[ImmutableBytesWritable])
    job.setOutputValueClass(classOf[Result])
    job.setOutputFormatClass(classOf[TableOutputFormat[ImmutableBytesWritable]])
    val indataRDD = sc.makeRDD(Array("3,Rongcheng,M,26","4,Guanhua,M,27")) //构建两行记录
    val rdd = indataRDD.map(_.split(',')).map{arr=>{
      val put = new Put(Bytes.toBytes(arr(0))) //行健的值
      put.add(Bytes.toBytes("info"),Bytes.toBytes("name"),Bytes.toBytes(arr(1)))  //info:name列的值
      put.add(Bytes.toBytes("info"),Bytes.toBytes("gender"),Bytes.toBytes(arr(2)))  //info:gender列的值
      put.add(Bytes.toBytes("info"),Bytes.toBytes("age"),Bytes.toBytes(arr(3).toInt))  //info:age列的值
      (new ImmutableBytesWritable, put)
    }}
    rdd.saveAsNewAPIHadoopDataset(job.getConfiguration())
  }
}
```

5.3.2 读写HBase数据

```
$ /usr/local/spark/bin/spark-submit   \
>--driver-class-path /usr/local/spark/jars/hbase/*:/usr/local/hbase/conf   \
>--class "SparkWriteHBase"   \
>/usr/local/spark/mycode/hbase/target/scala-2.11/simple-project_2.11-1.0.jar
```

切换到HBase Shell中，执行如下命令查看student表

```
hbase> scan 'student'
```

```
ROW                  COLUMN+CELL
 1                   column=info:age, timestamp=1479640712163, value=23
 1                   column=info:gender, timestamp=1479640704522, value=F
 1                   column=info:name, timestamp=1479640696132, value=Xueqian
 2                   column=info:age, timestamp=1479640752474, value=24
 2                   column=info:gender, timestamp=1479640745276, value=M
 2                   column=info:name, timestamp=1479640732763, value=Weiliang
 3                   column=info:age, timestamp=1479643273142, value=\x00\x00\x00\x1A
 3                   column=info:gender, timestamp=1479643273142, value=M
 3                   column=info:name, timestamp=1479643273142, value=Rongcheng
 4                   column=info:age, timestamp=1479643273142, value=\x00\x00\x00\x1B
 4                   column=info:gender, timestamp=1479643273142, value=M
 4                   column=info:name, timestamp=1479643273142, value=Guanhua
4 row(s) in 0.3240 seconds
```

5.4 综合案例

5.4.1 案例1：求TOP值
5.4.2 案例2：求最大最小值
5.4.3 案例3：文件排序
5.4.4 案例4：二次排序
5.4.5 案例5：连接操作

5.4.1 案例1：求TOP值

任务描述：

orderid,userid,payment,productid

file1.txt

```
1,1768,50,155
2,1218, 600,211
3,2239,788,242
4,3101,28,599
5,4899,290,129
6,3110,54,1201
7,4436,259,877
8,2369,7890,27
```

file2.txt

```
100,4287,226,233
101,6562,489,124
102,1124,33,17
103,3267,159,179
104,4569,57,125
105,1438,37,116
```

求Top N个payment值

5.4.1 案例1：求TOP值

```
import org.apache.spark.{SparkConf, SparkContext}
object TopN {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("TopN").setMaster("local")
    val sc = new SparkContext(conf)
    sc.setLogLevel("ERROR")
    val lines =
sc.textFile("hdfs://localhost:9000/user/hadoop/spark/mycode/rdd/examples",2)
    var num = 0;
    val result = lines.filter(line => (line.trim().length > 0) && (line.split(",").length == 4))
      .map(_.split(",")(2))
      .map(x => (x.toInt,""))
      .sortByKey(false)
      .map(x => x._1).take(5)
      .foreach(x => {
        num = num + 1
        println(num + "\t" + x)
      })
  }
}
```

5.4.1 案例1：求TOP值

```
val lines = sc.textFile("hdfs://localhost:9000/user/hadoop/spark/chapter5”,2)
```

该语句从文件中读取数据生成RDD（名称为lines），执行后的效果如下：

file1.txt

```
1,1768,50,155
2,1218, 600,211
3,2239,788,242
4,3101,28,599
5,4899,290,129
6,3110,54,1201
7,4436,259,877
8,2369,7890,27
```

file2.txt

```
100,4287,226,233
101,6562,489,124
102,1124,33,17
103,3267,159,179
104,4569,57,125
105,1438,37,116
```

val lines = sc.textFile()

RDD(lines)

- “1,1768,50,155”
- “2,1218, 600,211”
- “3,2239,788,242”
- ……
- “100,4287,226,233”
- “101,6562,489,124”
- “102,1124,33,17”
- ……

5.4.1 案例1：求TOP值

```
val result = lines.filter(line => (line.trim().length > 0) && (line.split(",").length == 4))
                    .map(_.split(",")(2))
```

该语句执行效果如下：

5.4.1 案例1：求TOP值

```
.map(x => (x.toInt,""))
```

该语句执行效果如下：

5.4.1 案例1：求TOP值

```
.sortByKey(false)
```

该语句会按照key的降序对RDD元素进行排序，执行效果如下：

RDD	RDD
(50,“”)	(7890,“”)
(600,””)	(788,””)
(788,””)	(600,””)
……	(489,””)
(226,””)	(290,””)
(489,””)	(259,””)
(33,””)	(226,””)
……	……

.sortByKey(false)

5.4.1 案例1：求TOP值

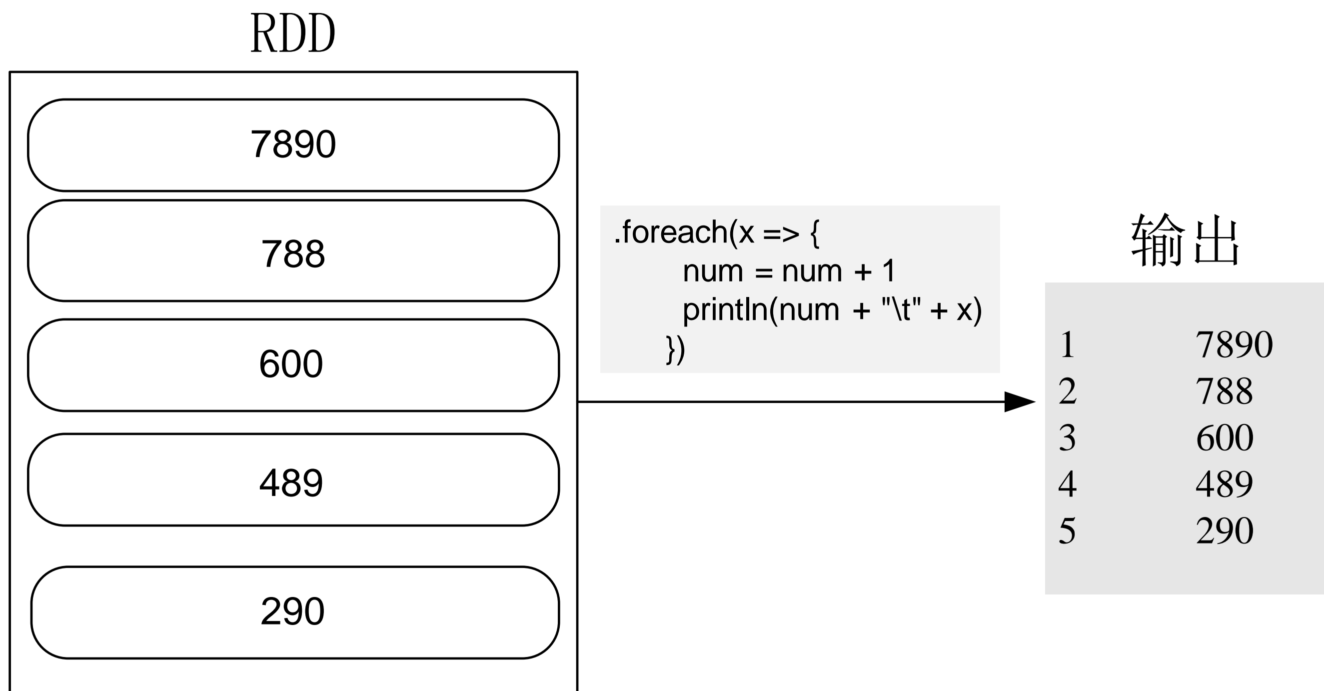
```
.map(x => x._1). take(5)
```

该语句执行效果如下：

5.4.1 案例1：求TOP值

```
.foreach(x => {
        num = num + 1
        println(num + "\t" + x)
    })
```

该语句执行效果如下：



5.4.2 案例2：求最大最小值

任务描述：求出多个文件中数值的最大、最小值

file1.txt

```
129
54
167
324
111
54
26
697
4856
3418
```

file2.txt

```
5
329
14
4567
2186
457
35
267
```

5.4.2 案例2：求最大最小值

```
import org.apache.spark.{SparkConf, SparkContext}
object MaxAndMin {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf().setAppName("MaxAndMin").setMaster("local")
    val sc = new SparkContext(conf)
    sc.setLogLevel("ERROR")
    val lines = sc.textFile("hdfs://localhost:9000/user/hadoop/spark/chapter5", 2)
```

剩余代码见下一页

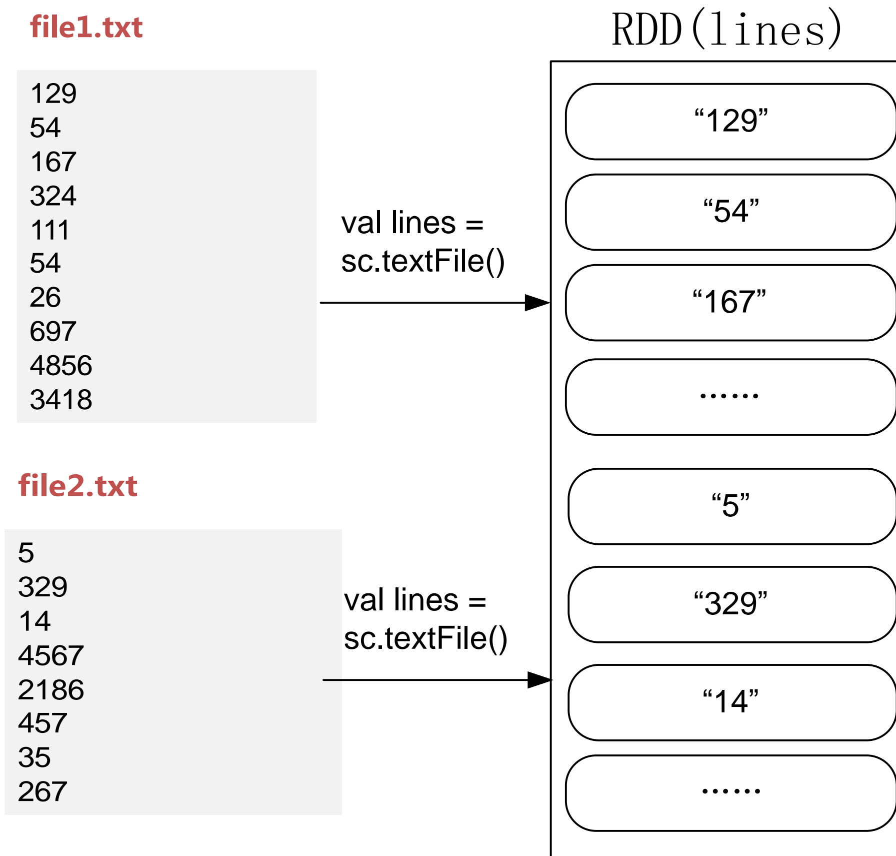
5.4.2 案例2：求最大最小值

```
 val result = lines.filter(_.trim().length>0).map(line =>
("key",line.trim.toInt)).groupByKey().map(x => {
      var min = Integer.MAX_VALUE
      var max = Integer.MIN_VALUE
      for(num <- x._2){
        if(num>max){
          max = num
        }
        if(num<min){
          min = num
        }
      }
      (max,min)
    }).collect.foreach(x => {
      println("max\t"+x._1)
      println("min\t"+x._2)
    })
  }
}
```

5.4.2 案例2：求最大最小值

```
val lines = sc.textFile("hdfs://localhost:9000/user/hadoop/spark/chapter5", 2)
```

该代码执行效果如下：



5.4.2 案例2：求最大最小值

```
val result = lines.filter(_.trim().length>0).map(line => ("key",line.trim.toInt))
```

该代码执行效果如下：

RDD(lines)

"129"
"54"
"167"
……
"5"
"329"
"14"
……

.map(line => ("key",line.trim.toInt))

RDD(lines)

("key",129)
("key",54)
("key",167)
……
("key",5)
("key",329)
("key",14)
……

5.4.2 案例2：求最大最小值

```
.groupByKey()
```

该代码执行效果如下：

RDD(lines)

("key",129)

("key",54)

("key",167)

……

("key",5)

("key",329)

("key",14)

……

.groupByKey()

RDD(lines)

("key",<129,54,167,…,5,329,14,...>)

5.4.2 案例2：求最大最小值

```
 val result = lines.filter(_.trim().length>0).map(line =>
("key",line.trim.toInt)).groupByKey().map(x => {
      var min = Integer.MAX_VALUE
      var max = Integer.MIN_VALUE
      for(num <- x._2){
        if(num>max){
          max = num
        }
        if(num<min){
          min = num
        }
      }
      (max,min)
    }).collect.foreach(x => {
      println("max\t"+x._1)
      println("min\t"+x._2)
    })
  }
}
```

for(num <- x._2){……}

for语句就是对x._2进行遍历
x._2就是键值对(key,value-list)中的value-list
这里value-list就是
<129,54,167,…,5,329,14,...>

5.4.3 案例3：文件排序

任务描述：

有多个输入文件，每个文件中的每一行内容均为一个整数。要求读取所有文件中的整数，进行排序后，输出到一个新的文件中，输出的内容个数为每行两个整数，第一个整数为第二个整数的排序位次，第二个整数为原待排序的整数

输入文件

file1.txt

33
37
12
40

file2.txt

4
16
39
5

file3.txt

1
45
25

输出文件

1	1
2	4
3	5
4	12
5	16
6	25
7	33
8	37
9	39
10	40
11	45

5.4.3 案例3：文件排序

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
import org.apache.spark.HashPartitioner
object FileSort {
    def main(args: Array[String]) {
        val conf = new SparkConf().setAppName("FileSort")
        val sc = new SparkContext(conf)
        val dataFile = "file:///usr/local/spark/mycode/rdd/data"
        val lines = sc.textFile(dataFile,3)
        var index = 0
        val result =
lines.filter(_.trim().length>0).map(n=>(n.trim.toInt,"")).partitionBy(new
HashPartitioner(1)).sortByKey().map(t => {
                     index += 1
            (index,t._1)
        })

result.saveAsTextFile("file:///usrl/local/spark/mycode/rdd/examples/result")
    }
}
```

5.4.4 案例4：二次排序

任务要求：

对于一个给定的文件（数据如file1.txt所示），请对数据进行排序，首先根据第1列数据降序排序，如果第1列数据相等，则根据第2列数据降序排序

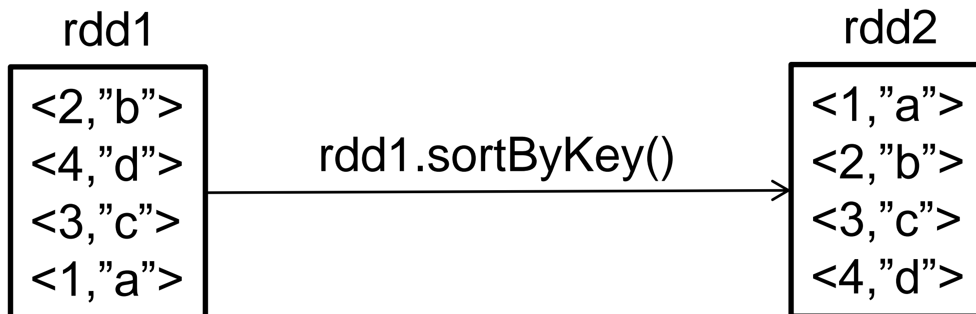
输入文件file1.txt

```
5 3
1 6
4 9
8 3
4 7
5 6
3 2
```

输出结果

```
8 3
5 6
5 3
4 9
4 7
3 2
1 6
```

5.4.4 案例4：二次排序



用于排序的key必须是可比较的对象

"5 3"
"1 6"
"4 9"
"8 3"
"4 7"
"5 6"
"3 2"

如何比较？

5.4.4 案例4：二次排序

解题思路：

RDD(lines)	RDD(pairWithSortKey)	RDD(sorted)	RDD(sortedResult)
“5 3”	(SecondarySortKey(5,3), ”5 3”)	(SecondarySortKey(8,3), ”8 3”)	”8 3”
“1 6”	(SecondarySortKey(1,6), ”1 6”)	(SecondarySortKey(5,6), ”5 6”)	”5 6”
“4 9”	(SecondarySortKey(4,9), ”4 9”)	(SecondarySortKey(5,3), ”5 3”)	”5 3”
“8 3”	(SecondarySortKey(8,3), ”8 3”)	(SecondarySortKey(4,9), ”4 9”)	”4 9”
“4 7”	(SecondarySortKey(4,7), ”4 7”)	(SecondarySortKey(4,7), ”4 7”)	”4 7”
“5 6”	(SecondarySortKey(5,6), ”5 6”)	(SecondarySortKey(3,2), ”3 2”)	”3 2”
“3 2”	(SecondarySortKey(3,2), ”3 2”)	(SecondarySortKey(1,6), ”1 6”)	”1 6”

5.4.4 案例4：二次排序

二次排序，具体的实现步骤如下

01 第一步：按照Ordered和Serializable接口实现自定义排序的key

02 第二步：将要进行二次排序的文件加载进来生成<key,value>类型的RDD

03 第三步：使用sortByKey基于自定义的Key进行二次排序

04 第四步：去除掉排序的Key,只保留排序的结果

5.4.4 案例4：二次排序

SecondarySortKey.scala代码如下：

```
package cn.edu.xmu.spark
class SecondarySortKey(val first:Int,val second:Int) extends Ordered
[SecondarySortKey] with Serializable {
def compare(other:SecondarySortKey):Int = {
    if (this.first - other.first !=0) {
         this.first - other.first
    } else {
      this.second - other.second
    }
  }
}
```

剩余代码见下一页

5.4.4 案例4：二次排序

SecondarySortApp.scala代码如下：

```
package cn.edu.xmu.spark
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
object SecondarySortApp {
  def main(args:Array[String]){
     val conf = new SparkConf().setAppName("SecondarySortApp").setMaster("local")
       val sc = new SparkContext(conf)
       val lines = sc.textFile("file:///usr/local/spark/mycode/rdd/examples/file1.txt", 1)
       val pairWithSortKey = lines.map(line=>(new SecondarySortKey(line.split("
")(0).toInt, line.split(" ")(1).toInt),line))
       val sorted = pairWithSortKey.sortByKey(false)
       val sortedResult = sorted.map(sortedLine =>sortedLine._2)
       sortedResult.collect().foreach (println)
  }
}
```

5.4.4 案例4：二次排序

```
val lines = sc.textFile("file:///usr/local/spark/mycode/rdd/file1.txt", 1)
```

该代码执行后的效果如下：

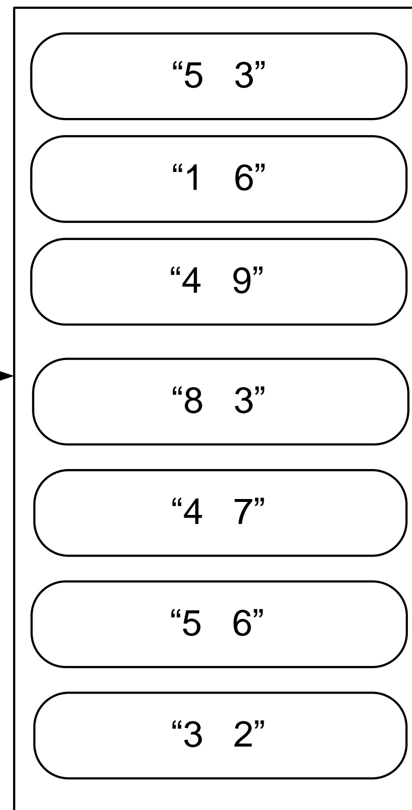
输入文件

file1.txt

```
5   3
1   6
4   9
8   3
4   7
5   6
3   2
```

val lines = sc.textFile(...)

RDD(lines)



5.4.4 案例4：二次排序

```
val pairWithSortKey =
 lines.map(line=>(new SecondarySortKey(line.split(" ")(0).toInt, line.split(" ")(1).toInt),line))
```

该代码执行后的效果如下：

RDD(lines)

- “5 3”
- “1 6”
- “4 9”
- “8 3”
- “4 7”
- “5 6”
- “3 2”

val pairWithSortKey = lines.map(line=>(new SecondarySortKey(line.split(" ")(0).toInt, line.split(" ")(1).toInt),line)) →

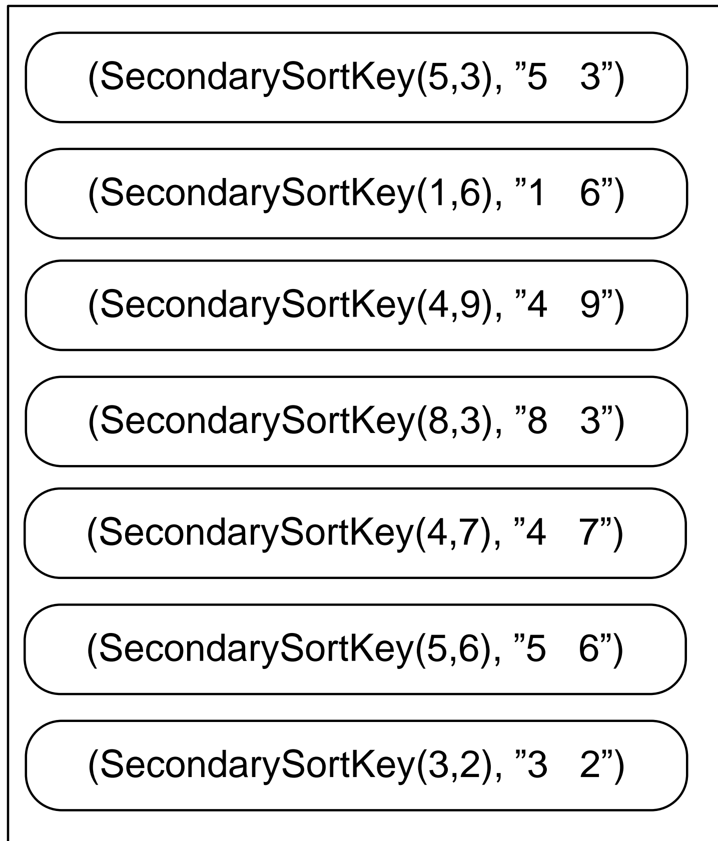
RDD(pairWithSortKey)

- (SecondarySortKey(5,3), ”5 3”)
- (SecondarySortKey(1,6), ”1 6”)
- (SecondarySortKey(4,9), ”4 9”)
- (SecondarySortKey(8,3), ”8 3”)
- (SecondarySortKey(4,7), ”4 7”)
- (SecondarySortKey(5,6), ”5 6”)
- (SecondarySortKey(3,2), ”3 2”)

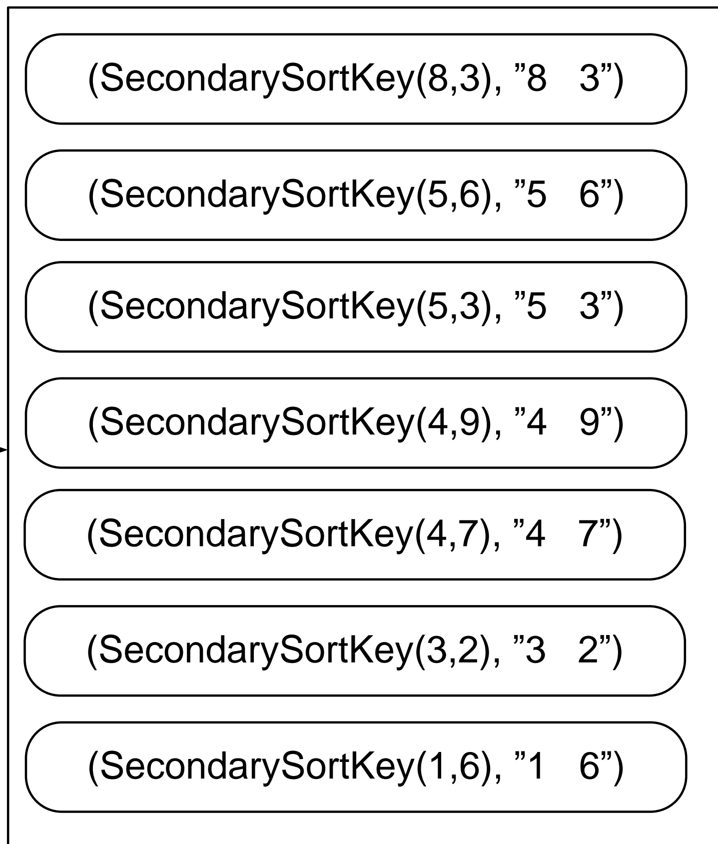
5.4.4 案例4：二次排序

```
val sorted = pairWithSortKey.sortByKey(false)
```

该代码会按照SecondarySortKey对象的降序排序，执行后的效果如下：



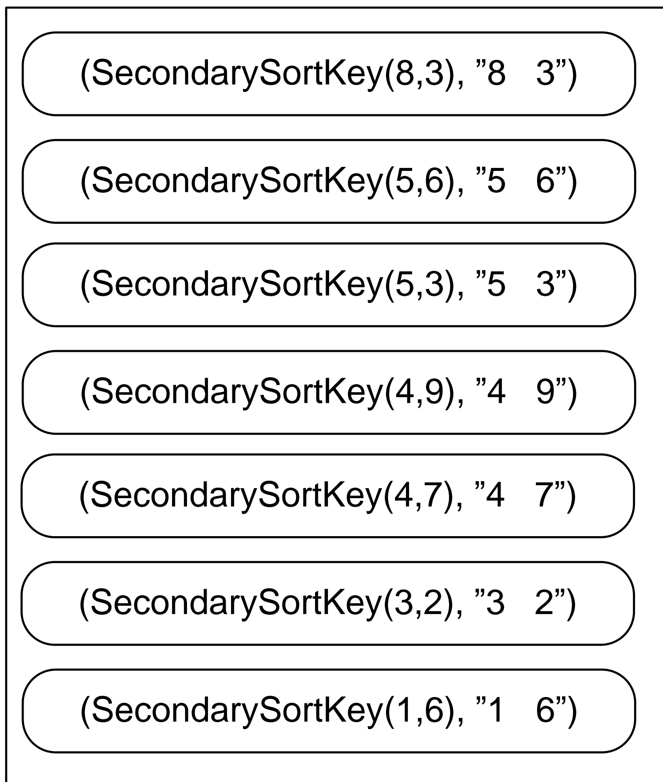
val sorted =
pairWithSortKey.
sortByKey(false)



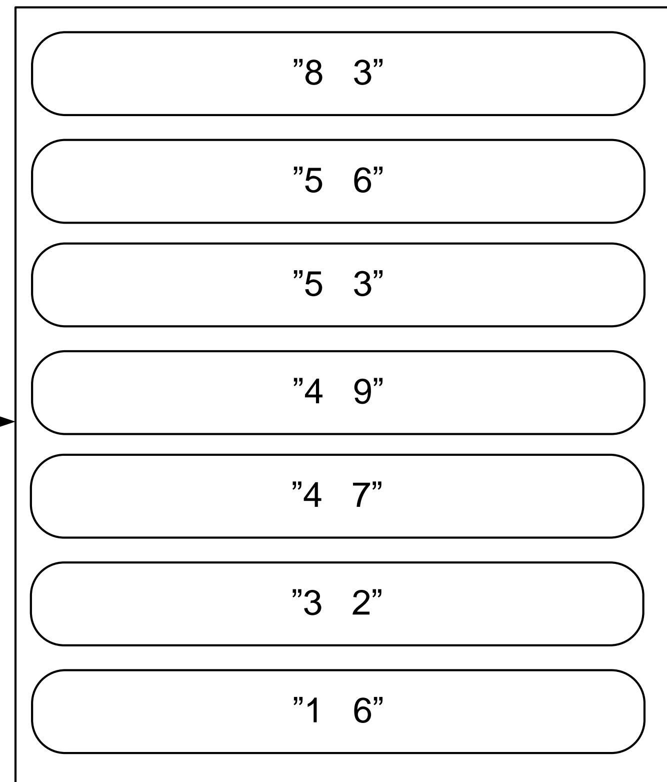
5.4.4 案例4：二次排序

```
val sortedResult = sorted.map(sortedLine =>sortedLine._2)
```

该代码执行后的效果如下：



val sortedResult = sorted.map(sortedLine =>sortedLine._2)



5.4.5 案例5：连接操作

任务描述：在推荐领域有一个著名的开放测试集，下载链接是：http://grouplens.org/datasets/movielens/，该测试集包含三个文件，分别是ratings.dat、sers.dat、movies.dat，具体介绍可阅读：README.txt。请编程实现：通过连接ratings.dat和movies.dat两个文件得到平均得分超过4.0的电影列表，采用的数据集是：ml-1m

5.4.5 案例5：连接操作

movies.dat

MovieID::Title::Genres

1::Toy Story (1995)::Animation|Children's|Comedy
2::Jumanji (1995)::Adventure|Children's|Fantasy
3::Grumpier Old Men (1995)::Comedy|Romance
4::Waiting to Exhale (1995)::Comedy|Drama
5::Father of the Bride Part II (1995)::Comedy
6::Heat (1995)::Action|Crime|Thriller
7::Sabrina (1995)::Comedy|Romance
8::Tom and Huck (1995)::Adventure|Children's
9::Sudden Death (1995)::Action
10::GoldenEye (1995)::Action|Adventure|Thriller

ratings.dat

UserID::MovieID::Rating::Timestamp

1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
1::2355::5::978824291
1::1197::3::978302268
1::1287::5::978302039
1::2804::5::978300719
1::594::4::978302268
1::919::4::978301368
1::595::5::978824268
1::938::4::978301752
1::2398::4::978302281
1::2918::4::978302124
1::1035::5::978301753
1::2791::4::978302188
1::2687::3::978824268

5.4.5 案例5：连接操作

```
import org.apache.spark._
import SparkContext._
object SparkJoin {
  def main(args: Array[String]) {
    if (args.length != 3 ){
      println("usage is WordCount <rating> <movie> <output>")
      return
    }
   val conf = new SparkConf().setAppName("SparkJoin").setMaster("local")
   val sc = new SparkContext(conf)
   // Read rating from HDFS file
   val textFile = sc.textFile(args(0))
```

//后面还有剩余代码

5.4.5 案例5：连接操作

```
//extract (movieid, rating)
   val rating = textFile.map(line => {
      val fileds = line.split("::")
      (fileds(1).toInt, fileds(2).toDouble)
     })
 //get (movieid,ave_rating)
   val movieScores = rating
     .groupByKey()
     .map(data => {
       val avg = data._2.sum / data._2.size
       (data._1, avg)
     })
```

//后面还有剩余代码

5.4.5 案例5：连接操作

```
// Read movie from HDFS file
     val movies = sc.textFile(args(1))
     val movieskey = movies.map(line => {
       val fileds = line.split("::")
        (fileds(0).toInt, fileds(1))   //(MovieID,MovieName)
     }).keyBy(tup => tup._1)

     // by join, we get <movie, averageRating, movieName>
     val result = movieScores
       .keyBy(tup => tup._1)
       .join(movieskey)
       .filter(f => f._2._1._2 > 4.0)
       .map(f => (f._1, f._2._1._2, f._2._2._2))

    result.saveAsTextFile(args(2))
  }
}
```

5.4.5 案例5：连接操作

```
// Read rating from HDFS file
val textFile = sc.textFile(args(0))
```

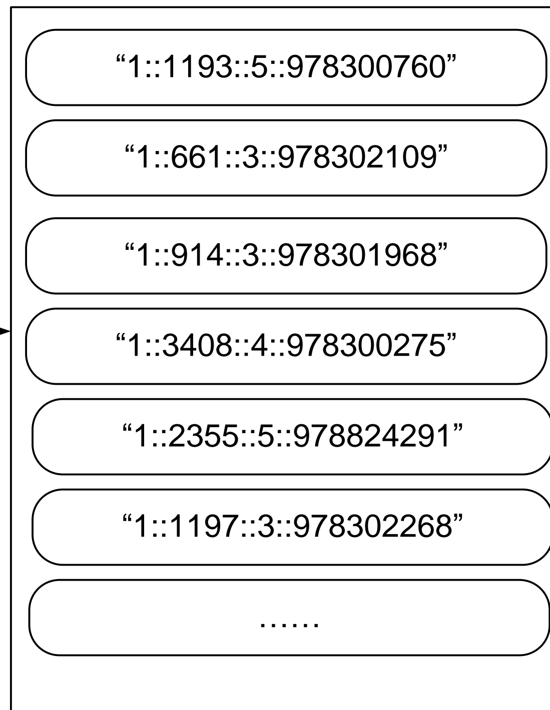
该代码执行效果如下：

ratings.dat

UserID::MovieID::Rating::Timestamp

```
1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
1::2355::5::978824291
1::1197::3::978302268
1::1287::5::978302039
1::2804::5::978300719
1::594::4::978302268
1::919::4::978301368
1::595::5::978824268
1::938::4::978301752
1::2398::4::978302281
1::2918::4::978302124
1::1035::5::978301753
1::2791::4::978302188
1::2687::3::978824268
```

val textFile = sc.textFile(args(0))



5.4.5 案例5：连接操作

连接操作

UserID::MovieID::Rating::Timestamp

```
//extract (movieid, rating)
    val rating = textFile.map(line => {
        val fileds = line.split("::")
        (fileds(1).toInt, fileds(2).toDouble)
       })
 //get (movieid,ave_rating)
    val movieScores = rating
       .groupByKey()
       .map(data => {
         val avg = data._2.sum / data._2.size
         (data._1, avg)
       })
```

```
1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
1::2355::5::978824291
1::1197::3::978302268
1::1287::5::978302039
1::2804::5::978300719
1::594::4::978302268
1::919::4::978301368
1::595::5::978824268
1::938::4::978301752
1::2398::4::978302281
1::2918::4::978302124
1::1035::5::978301753
1::2791::4::978302188
1::2687::3::978824268
```

5.4.5 案例5：连接操作

```
//extract (movieid, rating)
    val rating = textFile.map(line => {
        val fileds = line.split("::")
        (fileds(1).toInt, fileds(2).toDouble)
      })
```

该代码执行效果如下：

5.4.5 案例5：连接操作

```
rating.groupByKey()
```

该代码执行效果如下：

RDD(rating)

(1193,5)
(661,3)
(914,3)
(3408,4)
(2355,5)
(1197,3)
……

rating.groupByKey() →

RDD

(1193,<5,4,2,5,...>)
(661,<3,2,3,4,...>)
(914,<3,2,…>)
(3408,<4,2,5,…>)
(2355,<5,4,5,...>)
(1197,<3,2,4,...>)
……

5.4.5 案例5：连接操作

```
.map(data => {
        val avg = data._2.sum / data._2.size
        (data._1, avg)
      })
```

该代码执行效果如下：

5.4.5 案例5：连接操作

连接操作

```
// Read movie from HDFS file
     val movies = sc.textFile(args(1))
     val movieskey = movies.map(line => {
       val fileds = line.split("::")
        (fileds(0).toInt, fileds(1))   //(MovieID,MovieName)
     }).keyBy(tup => tup._1)
     // by join, we get <movie, averageRating, movieName>
     val result = movieScores
       .keyBy(tup => tup._1)
       .join(movieskey)
       .filter(f => f._2._1._2 > 4.0)
       .map(f => (f._1, f._2._1._2, f._2._2._2))

    result.saveAsTextFile(args(2))
  }
}
```

MovieID::Title::Genres

```
1::Toy Story (1995)::Animation|Children's|Comedy
2::Jumanji (1995)::Adventure|Children's|Fantasy
3::Grumpier Old Men (1995)::Comedy|Romance
4::Waiting to Exhale (1995)::Comedy|Drama
5::Father of the Bride Part II (1995)::Comedy
6::Heat (1995)::Action|Crime|Thriller
7::Sabrina (1995)::Comedy|Romance
8::Tom and Huck (1995)::Adventure|Children's
9::Sudden Death (1995)::Action
10::GoldenEye (1995)::Action|Adventure|Thriller
```

5.4.5 案例5：连接操作

```
// Read movie from HDFS file
    val movies = sc.textFile(args(1))
```

该代码执行效果如下：

5.4.5 案例5：连接操作

```
movies.map(line => {
      val fileds = line.split("::")
       (fileds(0).toInt, fileds(1))   //(MovieID,MovieName)
   })
```

该代码执行效果如下：

RDD(movies)

"1::Toy Story (1995)::Animation|Children's|Comedy"
"2::Jumanji (1995)::Adventure|Children's|Fantasy"
"3::Grumpier Old Men (1995)::Comedy|Romance"
"4::Waiting to Exhale (1995)::Comedy|Drama"
"5::Father of the Bride Part II (1995)::Comedy"
"6::Heat (1995)::Action|Crime|Thriller"
……

```
movies.map(line => {
      val fileds = line.split("::")
       (fileds(0).toInt, fileds(1))
   })
```

RDD

(1,"Toy Story (1995)")
(2,"Jumanji (1995)")
(3,"Grumpier Old Men (1995)")
(4,"Waiting to Exhale (1995)")
(5,"Father of the Bride Part II (1995)")
(6,"Heat (1995)")
……

5.4.5 案例5：连接操作

```
.keyBy(tup => tup._1)
```

该代码执行效果如下：

RDD

(1,"Toy Story (1995)")
(2,"Jumanji (1995)")
(3,"Grumpier Old Men (1995)")
(4,"Waiting to Exhale (1995)")
(5,"Father of the Bride Part II (1995)")
(6,"Heat (1995)")
……

.keyBy(tup => tup._1)

RDD(movieskey)

(1,(1,"Toy Story (1995)"))
(2,(2,"Jumanji (1995)"))
(3,(3,"Grumpier Old Men (1995)"))
(4,(4,"Waiting to Exhale (1995)"))
(5,(5,"Father of the Bride Part II (1995)"))
(6,(6,"Heat (1995)"))
……

5.4.5 案例5：连接操作

```
movieScores.keyBy(tup => tup._1)
```

该代码执行效果如下：

RDD

(1193,4.88)
(661,3.26)
(914,2.83)
(3408,4.66)
(2355,4.79)
(1197,3.17)
……

movieScores.keyBy(tup => tup._1)

RDD

(1193,(1193,4.88))
(661,(661,3.26))
(914,(914,2.83))
(3408,(3408,4.66))
(2355,(2355,4.79))
(1197,(1197,3.17))
……

5.4.5 案例5：连接操作

```
.join(movieskey)
```

执行join时，参与连接的两个RDD分别如下：

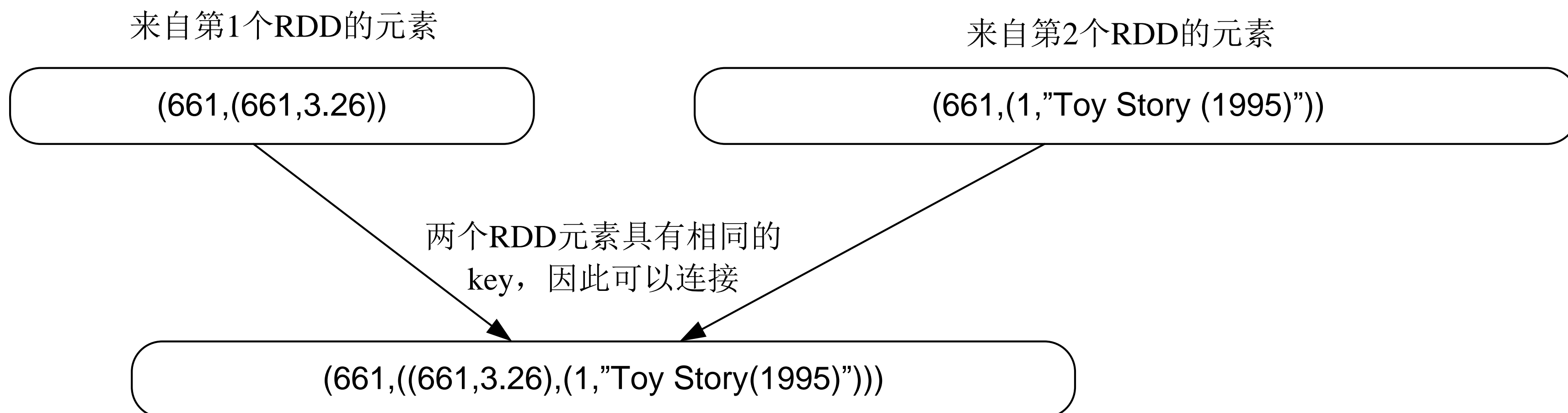
RDD	RDD(movieskey)
(1193,(1193,4.88))	(1,(1,"Toy Story (1995)"))
(661,(661,3.26))	(2,(2,"Jumanji (1995)"))
(914,(914,2.83))	(3,(3,"Grumpier Old Men (1995)"))
(3408,(3408,4.66))	(4,(4,"Waiting to Exhale (1995)"))
(2355,(2355,4.79))	(5,(5,"Father of the Bride Part II (1995)"))
(1197,(1197,3.17))	(6,(6,"Heat (1995)"))
……	……

两个RDD
执行join操作

5.4.5 案例5：连接操作

```
.join(movieskey) . filter(f => f._2._1._2 > 4.0) . map(f => (f._1, f._2._1._2, f._2._2._2))
```

执行join时，key相同的来自两个RDD的元素可以进行连接：



f._1是661
f._2是((661,3.26),(1,"Toy Story(1995)"))
f._2._1是(661,3.26)
f._2._1._2是3.26
f._2._2是(1,"Toy Story(1995)")
f._2._2._2是"Toy Story(1995)"

附录A：主讲教师林子雨简介

主讲教师：林子雨

单位：厦门大学计算机科学系
E-mail: ziyulin@xmu.edu.cn
个人网页：http://www.cs.xmu.edu.cn/linziyu
数据库实验室网站：http://dblab.xmu.edu.cn

扫一扫访问个人主页

林子雨，男，1978年出生，博士（毕业于北京大学），现为厦门大学计算机科学系助理教授（讲师），曾任厦门大学信息科学与技术学院院长助理、晋江市发展和改革局副局长。中国计算机学会数据库专业委员会委员，中国计算机学会信息系统专业委员会委员。国内高校首个“数字教师”提出者和建设者，厦门大学数据库实验室负责人，厦门大学云计算与大数据研究中心主要建设者和骨干成员，2013年度和2017年度厦门大学教学类奖教金获得者，荣获2017年福建省精品在线开放课程和2017年厦门大学高等教育成果二等奖。主要研究方向为数据库、数据仓库、数据挖掘、大数据、云计算和物联网，并以第一作者身份在《软件学报》《计算机学报》和《计算机研究与发展》等国家重点期刊以及国际学术会议上发表多篇学术论文。作为项目负责人主持的科研项目包括1项国家自然科学青年基金项目(No.61303004)、1项福建省自然科学青年基金项目(No.2013J05099)和1项中央高校基本科研业务费项目(No.2011121049)，主持的教改课题包括1项2016年福建省教改课题和1项2016年教育部产学协作育人项目，同时，作为课题负责人完成了国家发改委城市信息化重大课题、国家物联网重大应用示范工程区域试点泉州市工作方案、2015泉州市互联网经济调研等课题。中国高校首个“数字教师”提出者和建设者，2009年至今，“数字教师”大平台累计向网络免费发布超过500万字高价值的研究和教学资料，累计网络访问量超过500万次。打造了中国高校大数据教学知名品牌，编著出版了中国高校第一本系统介绍大数据知识的专业教材《大数据技术原理与应用》，并成为京东、当当网等网店畅销书籍；建设了国内高校首个大数据课程公共服务平台，为教师教学和学生学习大数据课程提供全方位、一站式服务，年访问量超过100万次。

附录B：大数据学习路线图

大数据学习路线图访问地址：http://dblab.xmu.edu.cn/post/10164/

附录C：《大数据技术原理与应用》教材

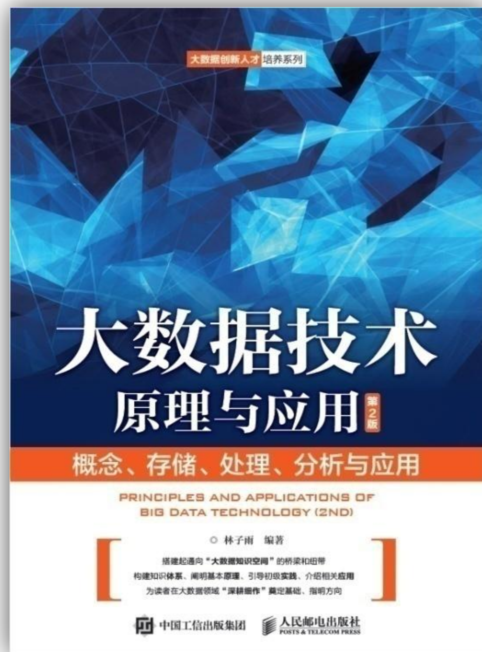
《大数据技术原理与应用——概念、存储、处理、分析与应用（第2版）》，由厦门大学计算机科学系林子雨博士编著，是国内高校第一本系统介绍大数据知识的专业教材。人民邮电出版社 ISBN:978-7-115-44330-4 定价：49.80元

扫一扫访问教材官网

全书共有15章，系统地论述了大数据的基本概念、大数据处理架构Hadoop、分布式文件系统HDFS、分布式数据 库HBase、NoSQL数据库、云数据库、分布式并行编程模型MapReduce、Spark、流计算、图计算、数据可视化以及大数据在互联网、生物医学和物流等各个领域的应用。在Hadoop、HDFS、HBase和MapReduce等重要章节，安排了入门级的实践操作，让读者更好地学习和掌握大数据关键技术。

本书可以作为高等院校计算机专业、信息管理等相关专业的大数据课程教材，也可供相关技术人员参考、学习、培训之用。

欢迎访问《大数据技术原理与应用——概念、存储、处理、分析与应用》教材官方网站：
http://dblab.xmu.edu.cn/post/bigdata



附录D：《大数据基础编程、实验和案例教程》

本书是与《大数据技术原理与应用（第2版）》教材配套的唯一指定实验指导书

大数据教材

1+1黄金组合

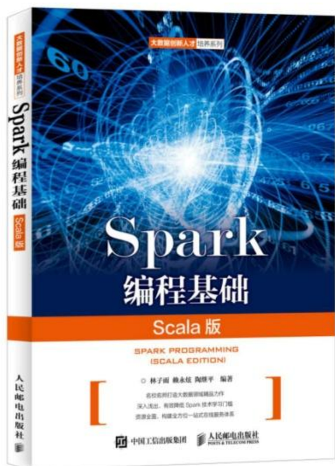
厦门大学林子雨编著

配套实验指导书

清华大学出版社 ISBN:978-7-302-47209-4 定价：59元

- 步步引导，循序渐进，详尽的安装指南为顺利搭建大数据实验环境铺平道路
- 深入浅出，去粗取精，丰富的代码实例帮助快速掌握大数据基础编程方法
- 精心设计，巧妙融合，五套大数据实验题目促进理论与编程知识的消化和吸收
- 结合理论，联系实际，大数据课程综合实验案例精彩呈现大数据分析全流程

附录E：《Spark编程基础（Scala版）》



《Spark编程基础（Scala版）》

厦门大学 林子雨，赖永炫，陶继平 编著

披荆斩棘，在大数据丛林中开辟学习捷径
填沟削坎，为快速学习Spark技术铺平道路
深入浅出，有效降低Spark技术学习门槛
资源全面，构建全方位一站式在线服务体系

人民邮电出版社出版发行，ISBN:978-7-115-48816-9
教材官网：http://dblab.xmu.edu.cn/post/spark/

本书以Scala作为开发Spark应用程序的编程语言，系统介绍了Spark编程的基础知识。全书共8章，内容包括大数据技术概述、Scala语言基础、Spark的设计与运行原理、Spark环境搭建和使用方法、RDD编程、Spark SQL、Spark Streaming、Spark MLlib等。本书每个章节都安排了入门级的编程实践操作，以便读者更好地学习和掌握Spark编程方法。本书官网免费提供了全套的在线教学资源，包括讲义PPT、习题、源代码、软件、数据集、授课视频、上机实验指南等。

附录F：高校大数据课程公共服务平台

高校大数据课程

公 共 服 务 平 台

http://dblab.xmu.edu.cn/post/bigdata-teaching-platform/

扫一扫访问平台主页

扫一扫观看3分钟FLASH动画宣传片

附录G：高校大数据实训课程系列案例教材

为了更好满足高校开设大数据实训课程的教材需求，厦门大学数据库实验室林子雨老师团队联合企业共同开发了《高校大数据实训课程系列案例》，目前已经完成开发的系列案例包括：

《基于协同过滤算法的电影推荐》

《电信用户行为分析》

《实时日志流处理分析》

《微博用户情感分析》

《互联网广告预测分析》

《网站日志处理分析》

部分教材书稿已经完成写作，将于2019年陆续出版发行，教材相关信息，敬请关注网页后续更新！http://dblab.xmu.edu.cn/post/shixunkecheng/

扫一扫访问大数据实训课程系列案例教材主页

Thank You!