

# Spark高级编程（I）



# 摘要

- 基本**RDD**操作
- 键值对操作
- 数据读取与保存
- 共享变量



# 摘要

- 基本**RDD**操作
- 键值对操作
- 数据读取与保存
- 共享变量



# 基本RDD操作（转换操作）

4

rdd: {1,2,3,3}

函数名	目的	示例	结果
map()	将函数应用于RDD中的每个元素，将返回值构成新的RDD	<code>rdd.map(x =&gt; x+1)</code>	{2,3,4,4}
flatMap()	将函数应用于RDD中的每个元素，将返回的迭代器的所有内容构成新的RDD，通常用来切分单词	<code>rdd.flatMap(x =&gt; x.to(3))</code>	{1,2,3,2,3,3,3}
filter()	返回一个由通过传给filter()的函数的元素组成的RDD	<code>rdd.filter(x =&gt; x!=1)</code>	{2,3,3}
distinct()	去重	<code>rdd.distinct()</code>	{1,2,3}
sample(withReplacement, fraction, [seed])	对RDD采样，以及是否替换	<code>rdd.sample(false, 0.5)</code>	非确定的



# 基本RDD操作（转换操作）

5

rdd: {1,2,3} 和 other: {3,4,5}

函数名	目的	示例	结果
union()	生成一个包含两个RDD中所有元素的RDD	rdd.union(other)	{1,2,3,3,4,5}
intersection()	求两个RDD共同的元素的RDD	rdd.intersection(other)	{3}
subtract()	移除一个RDD中的内容（例如移除训练数据）	rdd.subtract(other)	{1,2}
cartesian()	与另一个RDD的笛卡尔积	rdd.cartesian(other)	{{1,3},{1,4},{1,5},...{3,5}}



# 基本RDD操作（行动操作）

6

rdd: {1,2,3,3}

函数名	目的	示例	结果
collect()	返回RDD中的所有元素	rdd.collect()	{1,2,3,3}
count()	RDD中的元素个数	rdd.count()	4
countByValue()	各元素在RDD中出现的次数	rdd.countByValue()	{(1,1),(2,1),(3,2)}
take(num)	从RDD中返回前num个元素	rdd.take(2)	{1,2}
top(num)	从RDD中返回最“前面”的num个元素（默认降序）	rdd.top(2)	{3,3}
takeOrder(num)(or dering)	从RDD中按照提供的顺序返回最前面的num个元素	rdd.takeOrdered(2)(myOrdering)	{3,3}



# 基本RDD操作（行动操作）

7

rdd: {1,2,3,3}

函数名	目的	示例	结果
reduce(func)	并行整合RDD中的所有数据（例如sum）	<code>rdd.reduce((x,y) =&gt; x+y)</code>	9
fold(zero)(func)	和reduce一样，但是需要提供初始值	<code>rdd.fold(0)((x,y) =&gt; x+y)</code>	9
aggregate(zeroValue)(seqOp, combOp)	和reduce一样，但是通常返回不同类型的函数	<code>rdd.aggregate((0, 0))((x,y)=&gt;(x._1 + y, x._2+1), (x,y)=&gt; (x._1+y._1, x._2+y._2))</code>	(9,4)
foreach(func)	对RDD中的每个元素使用给定的函数	<code>rdd.foreach(func)</code>	无



# 摘要

- 基本**RDD**操作
- 键值对操作
- 数据读取与保存
- 共享变量





# 键值对操作

9

- 键值对RDD (Pair RDD) 通常用来进行聚合计算
- 键值对RDD的操作接口
- 键值对RDD分区



# 创建Pair RDD

10

## □ 调用map()函数实现

### ▣ Scala

```
val pairs = lines.map(x=>(x.split(" ")(0),x))
```

### ▣ Java

```
PairFunction<String, String, String> keyData = new PairFunction<String, String,  
String>() {  
    public Tuple2<String, String> call(String x){  
        return new Tuple2(x.split(" ")[0], x);  
    }  
}  
  
JavaPairRDD<String, String> pairs = lines.mapToPair(keyData)
```



# Pair RDD的转化操作

11

Pair RDD: {(1,2),(3,4),(3,6)}

函数名	目的	示例	结果
reduceByKey(func)	合并具有相同键的值	<code>rdd.reduceByKey((x,y) =&gt; x + y)</code>	{(1,2),(3,10)}
groupByKey()	对具有相同键的值进行分组	<code>rdd.groupByKey()</code>	{(1,[2]),(3,[4,6])}
combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)	使用不同的返回类型合并具有相同键的值		
mapValues(func)	对pair RDD中的每个值应用一个函数而不改变键	<code>rdd.mapValues(x=&gt;x+1)</code>	{(1,3),(3,5),(3,7)}
flatMapValues(func)	对pair RDD中的每个值应用一个返回迭代器的函数，然后对返回的每个元素都生成一个对应原键的键值对记录	<code>rdd.flatMapValues(x=&gt;(x to 5))</code>	{(1,2),(1,3),(1,4),(1,5),(3,4),(3,5)}
keys()	返回一个仅包含键的RDD	<code>rdd.keys()</code>	{1,3,3}
values()	返回一个仅包含值的RDD	<code>rdd.values()</code>	{2,4,6}
sortByKey()	返回一个根据键排序的RDD	<code>rdd.sortByKey()</code>	{(1,2),(3,4),(3,6)}



# Pair RDD的转化操作

12

`rdd = {(1,2),(3,4),(3,6)}` `other = {(3,9)}`

函数名	目的	示例	结果
<code>subtractByKey(func)</code>	删掉rdd中键与other中的键相同的元素	<code>rdd.subtractByKey(other)</code>	<code>{(1,2)}</code>
<code>join</code>	对两个rdd进行内连接	<code>rdd.join()</code>	<code>{(3,(4,9)),(3,(6,9))}</code>
<code>rightOuterJoin</code>	对两个rdd进行连接操作，确保右边rdd的键必须存在（右外连接）	<code>rdd.rightOuterJoin(other)</code>	<code>{(3,(Some(4),9)),(3,(Some(6),9))}</code>
<code>leftOuterJoin</code>	对两个rdd进行连接操作，确保左边rdd的键必须存在（左外连接）	<code>rdd.leftOuterJoin(other)</code>	<code>{(1,(2,none)),(3,(4,Some(9))), (3,(6,Some(9)))}</code>
<code>cogroup</code>	将两个rdd中拥有相同键的数据分组到一起	<code>rdd.cogroup(other)</code>	<code>{(1,([2],[ ])), (3,([4,6],[9]))}</code>



# Pair RDD

13

- Pair RDD也还是RDD（元素为Java或Scala中的Tuple2对象或者Python中的元组）。Pair RDD支持RDD所支持的函数。例如：

- Scala

```
result = paris.filter{case (key, value) => value.length < 20}
```

- Java

```
Function<Tuple2<String, String>, Boolean> longWordFilter =  
    new Function<Tuple2<String, String>, Boolean>() {  
        public Boolean call(Tuple2<String, String> keyValue){  
            return (keyValue._2().length() < 20);  
        }  
    }  
  
JavaPairRDD<String, String> result = pairs.filter(longWordFilter);
```

key	value
holden	Likes coffee
panda	Likes long strings and coffee

filter

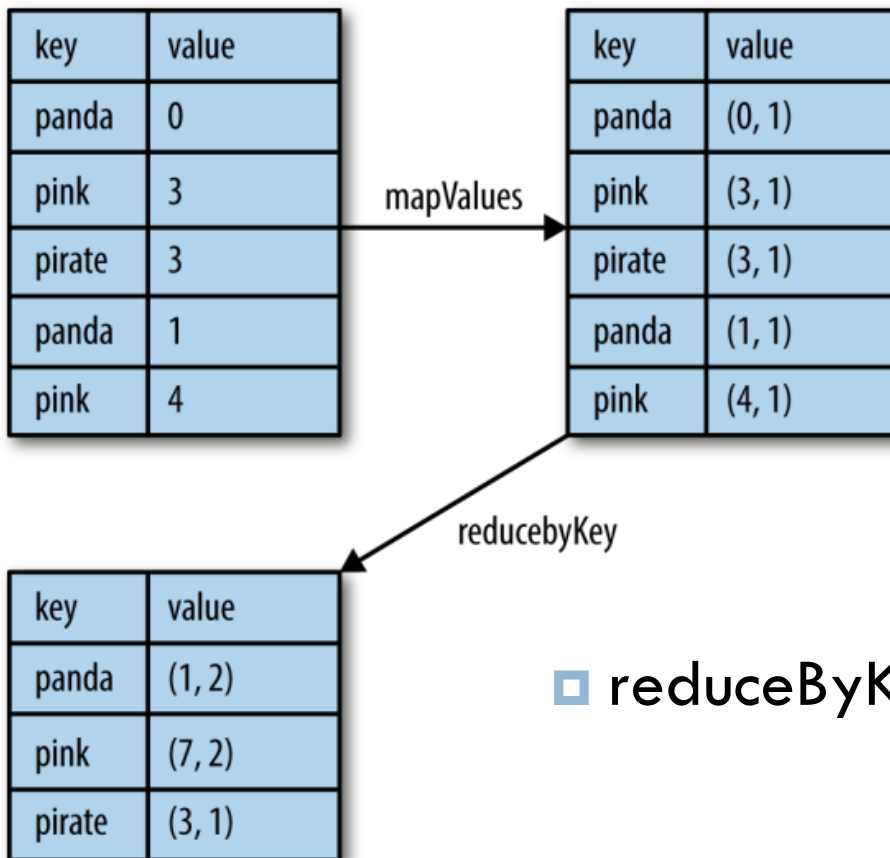
key	value
holden	Likes coffee



# 聚合操作

14

□ **reduceByKey()** `rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => x._1 + y._1, x._2 + y._2))`



□ **reduceByKey()** vs. **reduce()**

求每个键平均值的数据流



# 聚合操作

15

## □ 单词计数

```
val input = sc.textFile("s3://...")  
val words = input.flatMap(x => x.split(" "))  
val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)
```

## □ 或者

```
var result = input.flatMap(x=>x.split(" ")).countByValue()
```



# 聚合操作

16

## □ combineByKey(): 基于键进行聚合的函数

Partition 1

coffee	1
coffee	2
panda	3

Partition 1 trace:

(coffee, 1) -> new key

accumulators[coffee] = createCombiner(1)

(coffee, 2) -> existing key

accumulators[coffee] = merge Value(accumulators[coffee], 2)

(panda, 3) -> new key

accumulators[panda] = createCombiner(3)

如果是新元素，创建对应累加器的初始值

如果之前遇到过，将该键的累加器对应的当前值与这个新的值进行合并

Partition 2

coffee	9
--------	---

Partition 2 trace:

(coffee, 9) -> new key

accumulators[coffee] = createCombiner(9)

Merge Partitions:

mergeCombiners(partition1.accumulators[coffee],  
partition2.accumulators[coffee])

如果多个分区都有对应于同一个键的累加器，则需要将各个分区的结果进行合并

```
def createCombiner(value):  
  (value, 1)
```

```
def mergeValue(acc, value):  
  (acc[0] + value, acc[1] + 1)
```

```
def mergeCombiners(acc1, acc2):  
  (acc1[0] + acc2[0], acc1[1] + acc2[1])
```

## □ 求每个键对应的平均值

```
val result =  
  input.combineByKey( (v) =>  
    (v, 1),  
    (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),  
    (acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + acc2._2)  
  ).map{ case (key, value) => (key, value._1 / value._2.toFloat) }  
  result.collectAsMap().map(println(_))
```





# 分组

17

- **groupByKey():** 对数据进行分组
  - ▣ **RDD:  $[K, V] \rightarrow [K, \text{Iterable}[V]]$** 
    - `rdd.reduceByKey(func)`与`rdd.groupByKey().mapValues(value => value.reduce(func))`等价，但前者更为高效。
- **cogroup():** 对多个共享同一个键的**RDD**进行分组
  - ▣ **RDD:  $[K, V] \& [K, W] \rightarrow [(K, (\text{Iterable}[V], \text{Iterable}[W]))]$**
  - ▣ 不仅可以用于实现连接操作，还可以用来求键的交集。除此之外，还能同时应用于三个及以上的**RDD**。



# 连接

18

- 将有键的数据与另一组有键的数据一起使用。
- 连接方式：右外连接，左外连接，交叉连接，内连接。
  - ▣ 内连接(**join**)：只有两个**pari RDD**都存在的键才输出。
  - ▣ 左外连接(**leftOuterJoin**)：源**RDD**的每个键都有对应的记录。每个键相应的值是由一个源**RDD**中的值与一个包含第二个**RDD**的值的**Option**对象组成的二元组。
  - ▣ 右外连接(**rightOuterJoin**)：预期结果中的键必须出现在第二个**RDD**中，二元组中可缺失的部分则来自于源**RDD**而非第二个**RDD**。



# 排序

19

- **sortByKey():** 默认升序
- 自定义排序

## ▣ Scala

```
val input: RDD[(Int, Venue)] = ...
```

```
implicit val sortIntegersByString = new Ordering[Int]{
```

```
  override def compare(a: Int, b: Int) = a.toString.compare(b.toString)}
```

## ▣ Java

```
class IntegerComparator implements Comparator<Integer>{  
  public int compare(Integer a, Integer b) {  
    return String.valueOf(a).compareTo(String.valueOf(b));  
  }  
}
```



# Pair RDD的行动操作

20

□ RDD = {(1,2),(3,4),(3,6)}

函数名	目的	示例	结果
countByKey()	对每个键对应的元素进行计数	rdd.countByKey()	{(1,1),(3,2)}
collectAsMap()	将结果以映射表的形式返回，以便查询	rdd.collectAsMap()	Map{(1,2),(3,6)}
lookup(key)	返回给定键对应的所有值	rdd.lookup(3)	[4,6]

`collectAsMap()` 作用于K-V类型的RDD上，作用与`collect`不同的是`collectAsMap`函数不包含重复的key，对于重复的key，后面的元素覆盖前面的元素。



# 数据分区

21

- 合理分布数据能减少网络通信，从而大大提高性能。
- **Spark**可以选择自己的**RDD**分区分布来降低通信，但只有当一个数据集重复多次使用键值操作才起作用。
- **Spark**的分区操作作用于**key/value**型**RDD**上，它会让系统根据键值函数来分组元素。
  - ▣ 比如限定一组键值出现在指定节点上



# 数据分区

22

- 用户信息的大表(UserID, UserInfo)
- 过去五分钟的网站链接点击信息(UserID, LinkInfo)
- 计算多少用户访问了不在订阅列表的链接：用Spark的join()来分组UserInfo和LinkInfo，基于每个UserID。

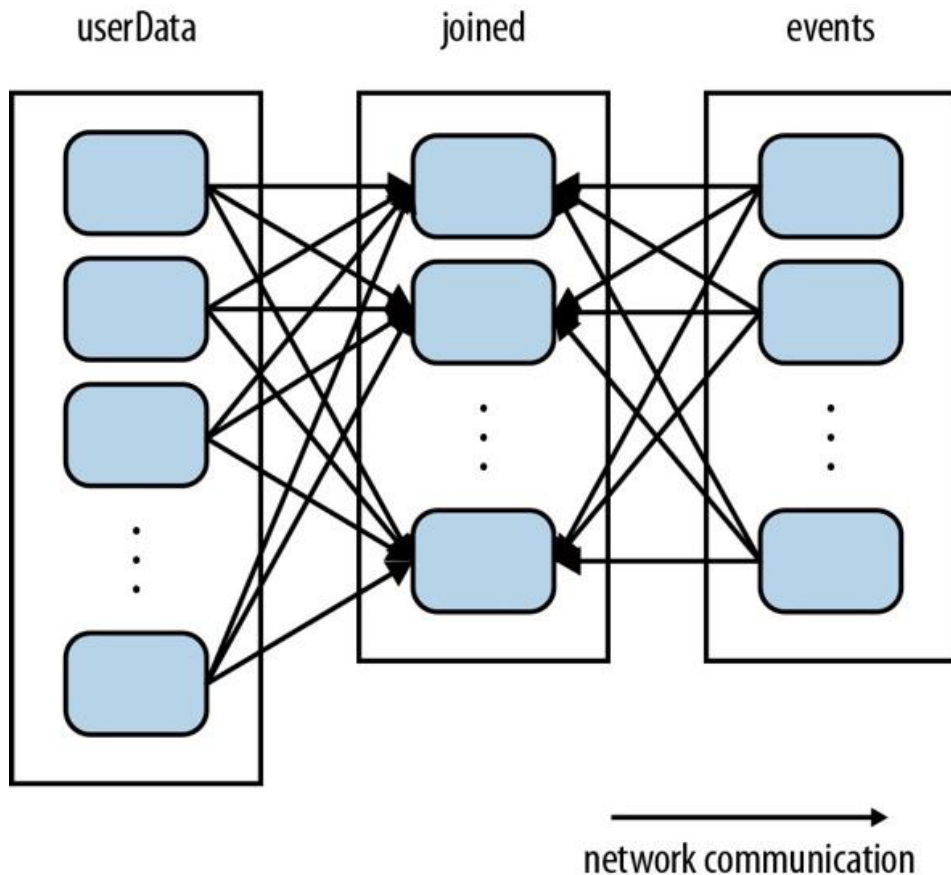
```
// Initialization code; we load the user info from a Hadoop SequenceFile on HDFS.
// This distributes elements of userData by the HDFS block where they are found,
// and doesn't provide Spark with any way of knowing in which partition a
// particular UserID is located.
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

// Function called periodically to process a logfile of events in the past 5 minutes;
// we assume that this is a SequenceFile containing (UserID, LinkInfo) pairs.
def processNewLogs(logFileName: String) {
  val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
  val joined = userData.join(events) // RDD of (UserID, (UserInfo, LinkInfo)) pairs
  val offTopicVisits = joined.filter {
    case (userId, (userInfo, linkInfo)) => // Expand the tuple into its components
      !userInfo.topics.contains(linkInfo.topic)
  }.count()
  println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```



# 数据分区

23



- 每次调用 `processNewLogs()` 就会调用 `join()`，在不知道该键值数据分区的情况下。默认情况下，会先哈希两个数据集的所有键，然后通过网络发送相同哈希值的元素到同一台机器，然后再用 `join` 连接相同机器上相同键值的元素
- **userData** 表比五分钟日志事件文件大得多，这会浪费大量的计算：在每次调用时，**userData** 表都会被散列并通过网络传输，即使它并没有改变。

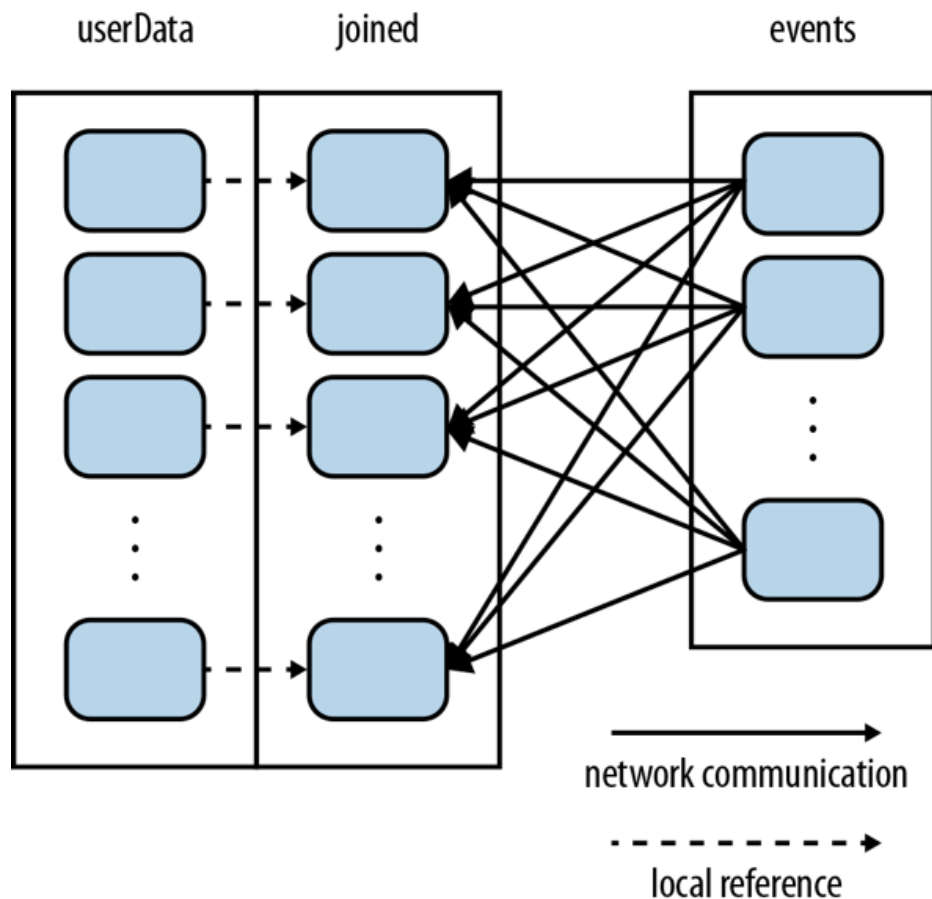


# 数据分区

24

```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
    .partitionBy(new HashPartitioner(100)) // Create 100 partitions
    .persist()
```

对userData表使用partitionBy(), 将其转为哈希分区



- 调用partitionBy()建立用户数据的时候，Spark知道这是哈希分区，调用join()方法时候利用这个优势。特别是，当我们调用userData.join(events)，Spark将重排事件RDD，把每个特定UserID发送到包含userData的散列分区，其结果是，减少了许多网络通信，让程序更快地运行。





# 数据分区

25

- `partitionBy()`是一个变换，所以它总是返回一个新的RDD。它不改变原来的RDD。RDD创建后无法修改。因此保存`partitionBy()`的结果`userData`非常重要的，而不是原来`sequenceFile()`。
- 在RDD传递给`partitionBy()`之后如果没有保存RDD，将导致RDD的后续使用重做数据分区操作。如果没有持久保存数据，分区的RDD会导致RDD完整谱系（`lineage`）的重新计算。那会与`partitionBy()`的优点矛盾，导致重复分区，并且通过网络重混数据（`shuffling`），相当于没有指定分区的情况。
- Spark的其它操作也常常自动生成已知划分信息的RDD，除了`join()`其他许多操作都会利用这一信息。例如，`sortByKey()`和`groupByKey()`会生成范围分区和哈希散列分区的RDD。而另一方面，像`map()`的操作会生成新的RDD并丢失父分区的信息，因为这个操作理论上可以修改每个记录。



# 数据分区

26

```
scala> val pairs = sc.parallelize(List((1, 1), (2, 2), (3, 3)))
pairs: spark.RDD[(Int, Int)] = ParallelCollectionRDD[0] at parallelize at <console>:12

scala> pairs.partitioner
res0: Option[spark.Partitioner] = None

scala> val partitioned = pairs.partitionBy(new spark.HashPartitioner(2))
partitioned: spark.RDD[(Int, Int)] = ShuffledRDD[1] at partitionBy at
<console>:14

scala> partitioned.partitioner
res1: Option[spark.Partitioner] = Some(spark.HashPartitioner@5147788d)
```

如果想在进一步的操作中使用分区，应该在输入第三行加上`persist()`，并在其中定义分区。如果没有`persist()`，随后RDD将重新评估分区的整个属性，这将导致数据对一遍又一遍地作哈希分区操作。



# 数据分区

27

- 以下操作会使输出RDD设置好分区器：`cogroup()`, `groupWith()`, `join()`, `leftOuterJoin()`, `rightOuterJoin()`, `groupByKey()`, `reduceByKey()`, `combineByKey()`, `partitionBy()`, `sort()`, `mapValues()`（如果父RDD具有分区器），`flatMapValues()`（如果父RDD具有分区器），和`filter()`（如果父RDD具有分区器）。其他所有操作将产生的结果都没有分区器。
- 对于多RDD操作，其分区器被设置的规则取决于父RDDs的分区器。默认情况下是散列分区器，分区数目与并发级别保持一致。然而，如果其中一个父RDD含有分区器，结果便会采用这个分区方式；如果多个父RDD都含有分区器，结果将取第一个父分区的分区器。



# Example: PageRank

28

- 数据集: (pageID, linkList)
- 输出: (pageID, rank)
- 算法:
  - 将每个页面的排序值初始化为1.0;
  - 在每次迭代中, 对页面 $p$ , 向其每个相邻页面 (有直接链接的页面) 发送一个值为  $\text{rank}(p)/\text{numNeighbours}(p)$  的贡献值;
  - 将每个页面的排序值设为  $0.15 + 0.85 * \text{contributionsReceived}$
  - 重复后两步, 算法逐渐收敛。通常需要10轮迭代。



# Example: PageRank

29

```
// Assume that our neighbor list was saved as a Spark objectFile
val links = sc.objectFile[(String, Seq[String])]("links")
    .partitionBy(new HashPartitioner(100))
    .persist()

// Initialize each page's rank to 1.0; since we use mapValues, the resulting RDD
// will have the same partitioner as links
var ranks = links.mapValues(v => 1.0)

// Run 10 iterations of PageRank
for (i <- 0 until 10) {
    val contributions = links.join(ranks).flatMap {
        case (pageId, (links, rank)) =>
            links.map(dest => (dest, rank / links.size))
    }
    ranks = contributions.reduceByKey((x, y) => x + y).mapValues(v => 0.15 + 0.85*v)
}

// Write out the final ranks
ranks.saveAsTextFile("ranks")
```

为了最大化分区相关优化的潜在作用，应该在无需改变元素的键时尽量使用`mapValues()`或`flatMapValues()`



# 自定义分区

30

- HashPartitioner和RangePartitioner
- 继承org.apache.spark.Partitioner
  - ▣ 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
  - ▣ equals(): Boolean 检查分区器对象是否和其它分区器实例相同。



# 自定义分区

31

```
class DomainNamePartitioner(numParts: Int) extends Partitioner{  
  override def numPartitions: Int = numParts  
  override def getPartition(key: Any): Int = {  
    val domain = new Java.net.URL(key.toString).getHost()  
    val code = (domain.hashCode % numPartitions)  
    if (code<0){ code + numPartitions } else { code }  
  }  
  override def equals(other:Any): Boolean = other match {  
    case dnp: DomainNamePartitioner =>  
      dnp.numPartitions == numPartitions  
    case _ => false  
  }  
}
```



# 摘要

- 基本**RDD**操作
- 键值对操作
- 数据读取与保存
- 共享变量





# 数据源

33

- 文件格式与文件系统
- **Spark SQL**中的结构化数据源
- 数据库与键值存储



# 文件格式

34

格式名称	结构化	备注
文本文件	否	普通的文本文件，每行一条记录
JSON	半结构化	常见的基于文本的格式，半结构化；大多数库都要求每行一条记录
CSV	是	非常常见的基于文本的格式，通常在电子表格应用中使用
SequenceFiles	是	一种用于键值对数据的常见Hadoop文件格式
Protocol buffers	是	一种快速、节约空间的跨语言格式
对象文件	是	用来将Spark作业中的数据存储下来以让共享的代码读取。改变类的时候它会失效，因为它依赖于Java序列化。



# 文件格式

35

- 文本文件：输入的每一行都会成为**RDD**的一个元素；也可以将多个完整的文本文件一次性读取为一个**Pair RDD**，键是文件名，值是文件内容。

```
val input = sc.textFile("file:///home/spark/README.MD")
```

```
input.saveAsTextFile(outputFile)
```

scala

```
val input = sc.wholeTextFiles("file:///home/spark/salefiles")
```

- **JSON**：将数据作为文本文件读取，然后使用**JSON**解析器来对**RDD**的值进行映射操作。

```
import json
```

```
data = input.map(lambda x: json.loads(x))
```

```
(data.filter(lambda x: x["lovesPandas"])).map(lambda x:  
json.dumps(x)).saveAsTextFile(outputFile)
```

python



# 文件格式

36

- **CSV**: 当作普通文本文件读取，再对数据进行处理。每条记录都没有相关联的字段名，只能得到对应的序号。常规做法是使用第一行中每列的值作为字段名。

```
import Java.io.StringReader
```

```
Import au.com.bytecode.opencsv.CSVReader
```

```
...
```

```
val input = sc.textFile(inputFile)
```

```
val result = input.map{ line =>
```

```
    val reader = new CSVReader(new StringReader(line));
```

```
    reader.readNext();
```

```
}
```

scala



# 文件格式

37

- **SequenceFile**: 由没有相对关系结构的键值对文件组成的常用**Hadoop**格式。有同步标记, **Spark**可以用它来定位到文件中的某个点, 然后再与记录的边界对齐。由实现**Hadoop**的**Writable**接口的元素组成。

```
val data = sc.sequenceFile(inFile, classOf[Text],  
  classOf[IntWritable]).map{case(x,y) => (x.toString, y.get())}  
val data = sc.parallelize(list(("Panda", 3), ("Kay", 6), ("Snail", 2)))  
data.saveAsSequenceFile(outputFile)
```

scala



# 文件系统

38

- 本地“常规”文件系统
  - ▣ `file:///home/holden/happypandas.gz`
- Amazon S3
  - ▣ `s3n://bucket/path-within-bucket`
- HDFS
  - ▣ `hdfs://master:port/path`



# Spark SQL中的结构化数据

39

- **Hive**: Spark SQL可以读取**Hive**支持的任何表

```
import org.apache.spark.sql.hive.HiveContext  
val hiveCtx = new org.apache.spark.sql.hive.HiveContext(sc)  
val rows = hiveCtx.sql("SELECT name, age FROM users")  
val firstRow = rows.first()  
println(firstRow.getString(0))
```



# Spark SQL中的结构化数据

40

- JSON: Spark SQL可以自动推断出JSON数据的结构信息

```
import org.apache.spark.sql.hive.HiveContext
val hiveCtx = new org.apache.spark.sql.hive.HiveContext(sc)
val tweets = hiveCtx.jsonFile("tweets.json")
tweets.registerTempTable("tweets")
val results = hiveCtx.sql("SELECT user.name, text FROM tweets")
```





# 数据库

41

## □ Java 数据库连接: `org.apache.spark.rdd.JdbcRDD`

```
def createConnection() = {  
    Class.forName("com.mysql.jdbc.Driver").newInstance();  
    DriverManager.getConnection("jdbc:mysql://localhost/test?user=holden");  
}  
  
def extractValues(r: ResultSet) = {  
    (r.getInt(1), r.getString(2))  
}  
  
val data = new JdbcRDD(sc, createConnection, "SELECT * FROM panda  
    WHERE ? <= id and id <= ?", lowerBound = 1, upperBound = 3,  
    numPartitions = 2, mapRow = extractValues)  
println(data.collect().toList)
```



# 数据库

42

- **Spark**可以通过**Hadoop**输入格式访问**HBase**，这个输入格式会返回键值对数据，其中键的类型为 `org.apache.hadoop.hbase.io.ImmutableBytesWritable`，值的类型为 `org.apache.hadoop.hbase.client.Result`.



# 摘要

- 基本**RDD**操作
- 键值对操作
- 数据读取与保存
- 共享变量



# 共享变量

44

- 在默认情况下，当**Spark**在集群的多个不同节点的多个任务上并行运行一个函数时，它会把函数中涉及到的每个变量，在每个任务上都生成一个副本。但是，有时候，需要在多个任务之间共享变量，或者在任务（**Task**）和任务控制节点（**Driver Program**）之间共享变量。为了满足这种需求，**Spark**提供了两种类型的变量：广播变量（**broadcast variables**）和累加器（**accumulators**）。



# 共享变量

45

## □ 广播变量

- ▣ 让程序高效地向所有工作节点发送一个较大的只读值，以供一个或多个**Spark**操作使用。

## □ 累加器

- ▣ 支持在所有不同节点之间进行累加计算（比如计数或者求和）



# 广播变量

46

- 可以通过调用 **SparkContext.broadcast(v)** 来从一个普通变量 **v** 中创建一个广播变量。这个广播变量就是对普通变量 **v** 的一个包装器，通过调用 **value** 方法就可以获得这个广播变量的值，例如：

```
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)
```

这个广播变量被创建以后，那么在集群中的任何函数中，都应该使用广播变量 **broadcastVar** 的值，而不是使用 **v** 的值，这样就不会把 **v** 重复分发到这些节点上。此外，一旦广播变量创建后，普通变量 **v** 的值就不能再发生修改，从而确保所有节点都获得这个广播变量的相同的值。



# 累加器

47

- 累加器是仅仅被相关操作累加的变量，通常可以被用来实现计数器（**counter**）和求和（**sum**）。**Spark**原生地支持数值型（**numeric**）的累加器，程序开发人员可以编写对新类型的支持。
- 一个数值型的累加器，可以通过调用 `SparkContext.longAccumulator()` 或者 `SparkContext.doubleAccumulator()` 来创建。运行在集群中的任务，就可以使用 `add` 方法来把数值累加到累加器上，但是，这些任务只能做累加操作，不能读取累加器的值，只有任务控制节点（**Driver Program**）可以使用 `value` 方法来读取累加器的值。



# 累加器

48

□ 例如：

```
scala> val accum = sc.longAccumulator("My Accumulator")
```

```
accum: org.apache.spark.util.LongAccumulator = LongAccumulator(id: 0,  
name: Some(My Accumulator), value: 0)
```

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

```
scala> accum.value
```

```
res1: Long = 10
```