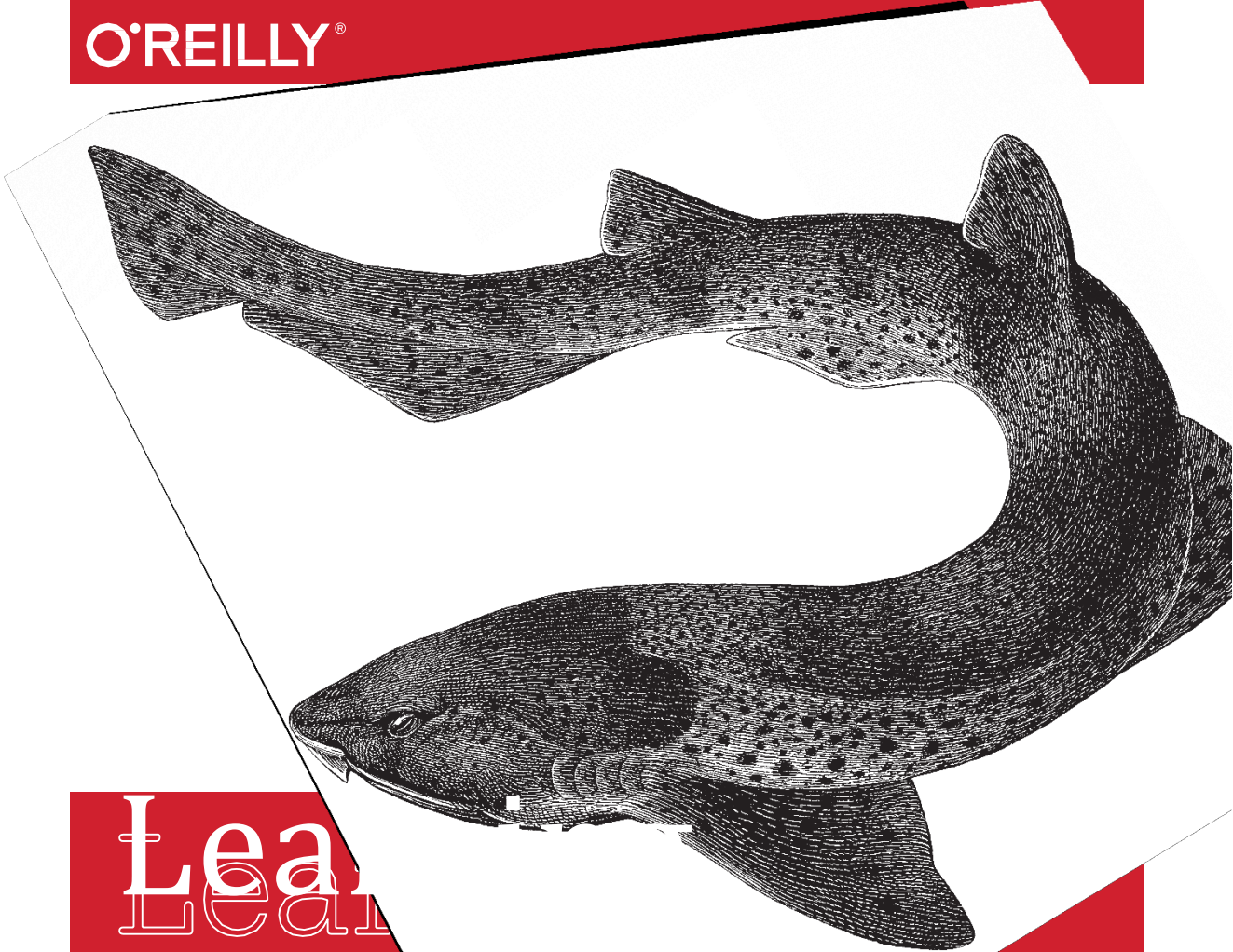
O'REILLY®



Lea Spark

LIGHTNING-FAST DATA ANALYSIS

Holden Karau, Andy Konwinski,
Patrick Wendell & Matei Zaharia

CHAPTER 4

Working with Key/Value Pairs

This chapter covers how to work with RDDs of key/value pairs, which are a common data type required for many operations in Spark. Key/value RDDs are commonly used to perform aggregations, and often we will do some initial ETL (extract, transform, and load) to get our data into a key/value format. Key/value RDDs expose new operations (e.g., counting up reviews for each product, grouping together data with the same key, and grouping together two different RDDs).

本章介绍了如何使用 key/value RDD，这是 Spark 最常见的数据类型。key/valueRDD 常用于进行数据聚合，而且往往我们会做一些初步的 ETL（提取，传输，加载），以把我们的数据转成 key/value 格式。key/value RDD 提供更多操作（例如，统计每个产品的评论数，统计相同键的数据，把两个不同的 RDD 组合在一起）。

We also discuss an advanced feature that lets users control the layout of pair RDDs across nodes: *partitioning*. Using controllable partitioning, applications can sometimes greatly reduce communication costs by ensuring that data will be accessed together and will be on the same node. This can provide significant speedups. We illustrate partitioning using the PageRank algorithm as an example. Choosing the right partitioning for a distributed dataset is similar to choosing the right data structure for a local one—in both cases, data layout can greatly affect performance.

我们会进一步讨论让用户控制各节点之间 RDD 的布局，即分区策略。使用可控的分区，把常常同时访问的数据放在同一节点，将大大降低通信费用，提升访问速率。我们将以 PageRank 算法为例介绍分区策略。为分布式数据集选择正确的分区策略的重要性类似于为本地数据选择正确的数据结构。在这两种环境下，分区策略和数据结构都会极大地影响系统性能。

Motivation

Spark provides special operations on RDDs containing key/value pairs. These RDDs are called pair RDDs. Pair RDDs are a useful building block in many programs, as

they expose operations that allow you to act on each key in parallel or regroup data across the network. For example, pair RDDs have a `reduceByKey()` method that can aggregate data separately for each key, and a `join()` method that can merge two RDDs together by grouping elements with the same key. It is common to extract fields from an RDD (representing, for instance, an event time, customer ID, or other identifier) and use those fields as keys in pair RDD operations.

> Spark 对 key/value RDD 提供特殊操作，这些 key/value RDD 称为 pair RDD。pair RDD 在许多程序的基础，因为它们可以操作平行的每个键或者通过网络重新组合数据。例如，pair RDD 有 reduceByKey()方法，该方法可以单独汇总数据，为每个键，和一个 join()方法，可以通过组合使用相同的键值元素合并两个 RDD 在一起。又如从 RDD 提取字段（例如，事件时间，客户 ID，或其它标识符），并使用这些字段在 pair RDD 作为操作的键。

Creating Pair RDDs

There are a number of ways to get pair RDDs in Spark. Many formats we explore loading from in Chapter 5 will directly return pair RDDs for their key/value data. In other cases we have a regular RDD that we want to turn into a pair RDD. We can do this by running a `map()` function that returns key/value pairs. To illustrate, we show code that starts with an RDD of lines of text and keys the data by the first word in each line.

> 在 Spark 中，可通过多种方式建立 pair RDD。第 5 章会探讨加载多种格式的 key/value 数据到 pair RDD 的方法。在其他情况下，要把 regular RDD 变成 pair RDD，可以通过运行 map()函数，返回 key/value。下面展示建立 RDD 的方法，此 RDD 包含若干行文字，并以第一个单词作为每行的 key。

The way to build key-value RDDs differs by language. In Python, for the functions on keyed data to work we need to return an RDD composed of tuples (see Example 4-1).

> 建立 key-value RDD 的方式因编程语言而异。在 Python 中，为了生成 key-value 数据，map 的参数为一个需要返回元组的函数（见例 4-1）。

Example 4-1. Creating a pair RDD using the first word as the key in Python

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```

In Scala, for the functions on keyed data to be available, we also need to return tuples (see Example 4-2). An implicit conversion on RDDs of tuples exists to provide the additional key/value functions.

在 Scala 中，为了生成 key-value 数据，我们需要返回元组（见例 4-2）。对元组的 RDD 存在隐式转换，以提供额外的 key/value 的函数。

Example 4-2. Creating a pair RDD using the first word as the key in Scala

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

Java doesn't have a built-in tuple type, so Spark's Java API has users create tuples using the `scala.Tuple2` class. This class is very simple: Java users can construct a new tuple by writing `new Tuple2(elem1, elem2)` and can then access its elements with the `._1()` and `._2()` methods.

Java 没有内置的元组类型，所以 Spark 的 Java API 有用户创建一个使用 scala.Tuple2 类的元组。这个类是非常简单的：Java 用户可以通过写新 Tuple2 构造一个新的元组（elem1，elem2），然后可以访问它的元素与._1()和._2()方法。

Java users also need to call special versions of Spark's functions when creating pair RDDs. For instance, the `mapToPair()` function should be used in place of the basic `map()` function. This is discussed in more detail in "Java" on page 43, but let's look at a simple case in Example 4-3.

Java 创建 pair RDD 时要调用 Spark 函数的特殊版本。例如，mapToPair()函数应代替基本的 map()函数使用。这会在 43 页上的"Java"部分更详细地讨论，但让我们先看看例 4-3 这个简单的例子。

Example 4-3. Creating a pair RDD using the first word as the key in 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);
```

When creating a pair RDD from an in-memory collection in Scala and Python, we only need to call `SparkContext.parallelize()` on a collection of pairs. To create a pair RDD in Java from an in-memory collection, we instead use `SparkContext.parallelizePairs()`.

对于 Scala 和 Python，从内存中集合创建一个 pair RDD，我们只需要调用 SparkContext.parallelize()。对于 Java，要从内存中集合创建 pair RDD，我们改用

SparkContext.parallelizePairs()。

Transformations on Pair RDDs

Pair RDDs are allowed to use all the transformations available to standard RDDs. The same rules apply from "Passing Functions to Spark" on page 30. Since pair RDDs contain tuples, we need to pass functions that operate on tuples rather than on individual elements. Tables 4-1 and 4-2 summarize transformations on pair RDDs, and we will dive into the transformations in detail later in the chapter.

RDD 可使用所有标准的 RDD 转换。同样的规则也适用于 30 页上“传递函数给 Spark”的部分。由于 RDD 包含元组，我们需要处理元组的函数，而不是对的单个元素进行操作的函数。表 4-1 和表 4-2 中总结了 pair RDD 的转换，我们将在此章详细讨论。

Table 4-1. Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)})

Function name	Purpose	Example	Result
`reduceByKey(func)`	Combine values with the same key.	`rdd.reduceByKey((x, y) => x + y)`	`{(1, 2), (3, 10)}`
`groupByKey()`	Group values with the same key.	`rdd.groupByKey()`	`{(1, [2]), (3, [4, 6])}`
`combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)`	Combine values with the same key using a different result type.	See Examples 4-12 through 4-14.	
`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key.	`rdd.mapValues(x => x+1)`	`{(1, 3), (3, 5), (3, 7)}`
`flatMapValues(func)`	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value		entry with the old key. Often used for tokenization.

`rdd.flatMapValues(x => (x to 5)`			`{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}`
`keys()`	Return an RDD of just the keys.	`rdd.keys()`	`{1, 3, 3}`

Function name	Purpose	Example	Result
values()	Return an RDD of just the values.	rdd.values()	{2, 4, 6}
sortByKey()	Return an RDD sorted by the key.	rdd.sortByKey()	{(1, 2), (3, 4), (3, 6)}

Table 4-2. Transformations on two pair RDDs (rdd = {(1, 2), (3, 4), (3, 6)} other = {(3, 9)})

Function name	Purpose	Example	Result
subtractByKey	Remove elements with a key present in the other RDD.	rdd.subtractByKey(other)	{(1, 2)}
join	Perform an inner join between two RDDs.	rdd.join(other)	{(3, (4, 9)), (3, (6, 9))}
rightOuterJoin	Perform a join between two RDDs where the key must be present in the first RDD.	rdd.rightOuterJoin(other)	{(3,(Some(4),9)), (3,(Some(6),9))}
leftOuterJoin	Perform a join between two RDDs where the key must be present in the other RDD.	rdd.leftOuterJoin(other)	{(1,(2,None)), (3,(4,Some(9))), (3,(6,Some(9)))}
cogroup	Group data from both RDDs sharing the same key.	rdd.cogroup(other)	{(1,([2],[])), (3,([4, 6],[9]))}

We discuss each of these families of pair RDD functions in more detail in the upcoming sections.

> 接下来我们将详细讨论每个 pair RDD 函数的细节。

Pair RDDs are also still RDDs (of Tuple2 objects in Java/Scala or of Python tuples), and thus support the same functions as RDDs. For instance, we can take our pair RDD from the previous section and filter out lines longer than 20 characters, as shown in Examples 4-4 through 4-6 and Figure 4-1.

pair RDD 也仍然是 RDD（包含 Scala 或 Python 的元组、Java 的 Tuple2 对象），自然也支持 RDD 上的函数。例如，我们可以利用我们 pair RDD 从上一节并过滤掉行超过 20 个字符，如下例 4-4 至 4-6 和图 4-1。

Example 4-4. Simple filter on second element in Python

```
result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)
```

Example 4-5. Simple filter on second element in Scala

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

Example 4-6. Simple filter on second element in 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);
```

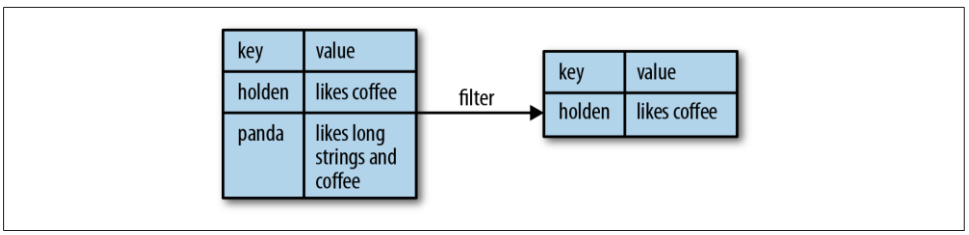


Figure 4-1. Filter on value

Sometimes working with pairs can be awkward if we want to access only the value part of our pair RDD. Since this is a common pattern, Spark provides the `mapValues(func)` function, which is the same as `map{case (x, y): (x, func(y))}`. We will use this function in many of our examples.

> 有时我们仅访问 pair RDD 的 value 部分。由于这是一个常见的模式，Spark 提供 mapValues(func)函数，效果与 map{case (x, y): (x, func(y))}相同。我们将在许多例子中使用这个函数。

We now discuss each of the families of pair RDD functions, starting with aggregations.

> 我们现在开始讨论的每一个 pair RDD 函数家族，从聚合函数开始。

Aggregations

When datasets are described in terms of key/value pairs, it is common to want to aggregate statistics across all elements with the same key. We have looked at the `fold()`, `combine()`, and `reduce()` actions on basic RDDs, and similar per-key transformations exist on pair RDDs. Spark has a similar set of operations that combines values that have the same key. These operations return RDDs and thus are transformations rather than actions.

> 当数据集用 key/value 表示时，我们往往需要聚合相同键值的统计信息。我们已经看到 fold()，combine()，reduce()等基本的 RDD 操作，以及 pair RDD 每个键的转换。Spark 也有类似的一套操作操作具有相同键值的数据。这些操作返回 RDD，因此是转

换（transformation），而不是行动（action）。

reduceByKey() is quite similar to reduce(); both take a function and use it to combine values. reduceByKey() runs several parallel reduce operations, one for each key in the dataset, where each operation combines values that have the same key. Because datasets can have very large numbers of keys, reduceByKey() is not implemented as an action that returns a value to the user program. Instead, it returns a new RDD consisting of each key and the reduced value for that key.

reduceByKey()类似于 reduce()；即以一个函数作为输入，并使用此函数来合并数据。reduceByKey()运行多个并行的 reduce 操作，一个用于在数据集中的每个键，其中每个操作结合具有相同的键值。由于数据集可以有非常多的键值，reduceByKey()未实现为返回一个值给程序；相反，它返回一个新的 RDD，其中包含每个键，以及各个键的 reduce 值。

foldByKey() is quite similar to fold(); both use a zero value of the same type of the data in our RDD and combination function. As with fold(), the provided zero value for foldByKey() should have no impact when added with your combination function to another element.

foldByKey()非常类似于 fold()，它们都使用 RDD 中相同类型的零值以及组合函数。类似于 fold()， 对 foldByKey()提供的零值应该对组合数据相加没有影响。

As Examples 4-7 and 4-8 demonstrate, we can use reduceByKey() along with mapValues() to compute the per-key average in a very similar manner to how fold() and map() can be used to compute the entire RDD average (see Figure 4-2). As with averaging, we can achieve the same result using a more specialized function, which we will cover next.

如示例 4-7 和 4-8 所示，我们用类似于 fold()和 map()计算整个 RDD 平均数的方法，通过 reduceByKey()和 mapValues()来计算到如何折叠每个键对应的平均数（见图 4-2）。当然实现平均数可以使用一个更专业的函数，我们将在下面讨论。

Example 4-7. Per-key average with reduceByKey() and mapValues() in Python

```
rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1]))
```

Example 4-8. Per-key average with reduceByKey() and mapValues() in Scala

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

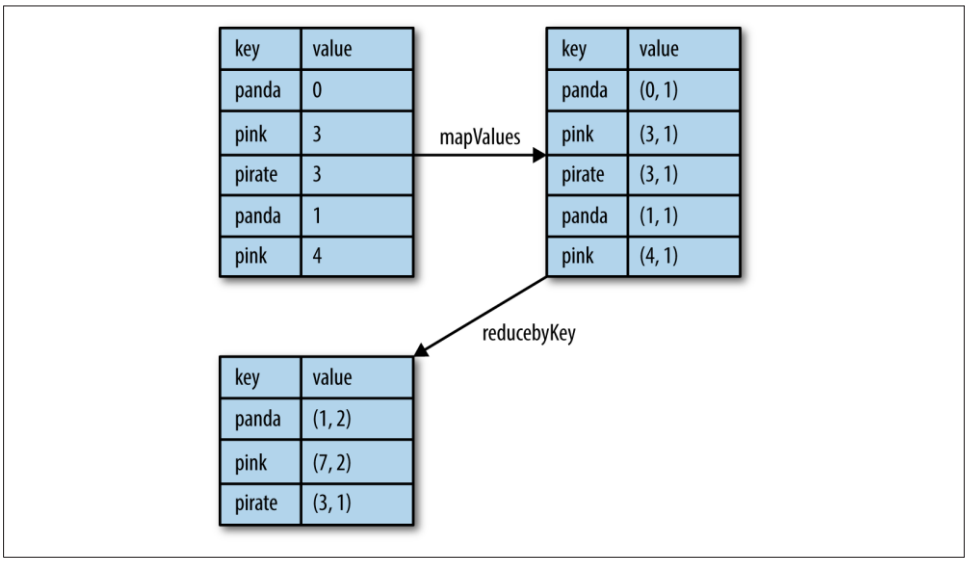


Figure 4-2. Per-key average data flow

Those familiar with the combiner concept from MapReduce should note that calling `reduceByKey()` and `foldByKey()` will automatically perform combining locally on each machine before computing global totals for each key. The user does not need to specify a combiner. The more general `combineByKey()` interface allows you to customize combining behavior.

熟悉 MapReduce 的 combiner 概念的用户应该会注意到，在调用 reduceByKey()和 foldByKey()时会自动在每台机器上本地计算局部合并值，然后才计算全局总量；用户不需要指定一个 combiner。而更通用的 combineByKey()接口允许用户自定义组合的行为。

We can use a similar approach in Examples 4-9 through 4-11 to also implement the classic distributed word count problem. We will use `flatMap()` from the previous chapter so that we can produce a pair RDD of words and the number 1 and then sum together all of the words using `reduceByKey()` as in Examples 4-7 and 4-8.

> 我们可以使用实例 4-9 类似的方法，通过 4-11 也实现了经典的分布式的字数统计（word count）问题。我们将使用上一章的 flatMap()，使我们可以生产 pair RDD 和数字 1，然后使用 reduceByKey()把字数加在一起，如示例 4-7 和 4-8。

Example 4-9. Word count in Python

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

Example 4-10. Word count in Scala

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

Example 4-11. Word count in Java

```
JavaRDD<String> input = sc.textFile("s3://...")
JavaRDD<String> words = rdd.flatMap(new FlatMapFunction<String, String>() {
  public Iterable<String> call(String x) { return Arrays.asList(x.split(" ")); }
});
JavaPairRDD<String, Integer> result = words.mapToPair(
  new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String x) { return new Tuple2(x, 1); }
}).reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer a, Integer b) { return a + b; }
});
```

We can actually implement word count even faster by using the `countByValue()` function on the first RDD: `input.flatMap(x => x.split(" ")).countByValue()`.

事实上，我们可以利用 countByValue()函数在第一个 RDD 上实现更快的字数统计：input.flatMap(x => x.split("")).countByValue()。

`combineByKey()` is the most general of the per-key aggregation functions. Most of the other per-key combiners are implemented using it. Like `aggregate()`, `combineByKey()` allows the user to return values that are not the same type as our input data.

combineByKey()是最通用的键值聚集函数。大多数其他每个键组合都在使用它来实现。像 aggregate()，combineByKey()允许返回值类型与输入值类型不同。

To understand `combineByKey()`, it's useful to think of how it handles each element it processes. As `combineByKey()` goes through the elements in a partition, each element either has a key it hasn't seen before or has the same key as a previous element.

想要理解 combineByKey()，得先明白它是如何作用于处理的每个元素的。由于函数 combineByKey()依次访问分区中的元素，访问的时候遇到的键值要么已经开始统计，要么就是新的键值。

If it's a new element, `combineByKey()` uses a function we provide, called `createCombiner()`, to create the initial value for the accumulator on that key. It's important to note that this happens the first time a key is found in each partition, rather than only the first time the key is found in the RDD.

当它是一个新的键值，combineByKey()使用提供的函数，称为创建 Combiner()，来创建该键值统计的初始值。值得注意的是，这发生在第一次键值单个分区中被发现，而不仅仅是在第一次键值在 RDD 中被找到。

If it is a value we have seen before while processing that partition, it will instead use the provided function, `mergeValue()`, with the current value for the accumulator for that key and the new value.

如果是已经见过该键值，我们会使用另外一个函数，mergeValue()，对当前键值的统计值和新值进行合并。

Since each partition is processed independently, we can have multiple accumulators for the same key. When we are merging the results from each partition, if two or more partitions have an accumulator for the same key we merge the accumulators using the user-supplied `mergeCombiners()` function.

由于每个分区被独立地处理，我们对相同的键值有多个累加器。当我们合并来自每

个分区的结果，如果两个或多个分区有相同键值的累加器，我们就使用用户提供的mergeCombiners()函数中来合并。

We can disable map-side aggregation in `combineByKey()` if we know that our data won't benefit from it. For example, `groupByKey()` disables map-side aggregation as the aggregation function (appending to a list) does not save any space. If we want to disable map-side combines, we need to specify the partitioner; for now you can just use the partitioner on the source RDD by passing `rdd.partitioner`.

我们可以在 combineByKey()禁用 map 端的聚合，如果我们知道数据在 map 端聚合没有好处。例如，groupByKey()禁用 map 端聚合为“聚合函数（追加到列表中）不保存任何空间”。如果我们要禁用 map 端 combine，我们需要指定具体分区器；然后可以通过传递 rdd.partitioner 来表示源端的 RDD 分区。

Since `combineByKey()` has a lot of different parameters it is a great candidate for an explanatory example. To better illustrate how `combineByKey()` works, we will look at computing the average value for each key, as shown in Examples 4-12 through 4-14 and illustrated in Figure 4-3.

由于 combineByKey()有很多不同的参数，能用它阐述 Spark 的编程方式。为了更好地说明 combineByKey()是如何工作的，我们用它来计算每个键的平均值，如示例 4-12 至 4-14 和图 4-3 所示。

Example 4-12. Per-key average using combineByKey() in Python

```
sumCount = nums.combineByKey((lambda x: (x,1)),
                             (lambda x, y: (x[0] + y, x[1] + 1)),
                             (lambda x, y: (x[0] + y[0], x[1] + y[1])))
sumCount.map(lambda key, xy: (key, xy[0]/xy[1])).collectAsMap()
```

Example 4-13. Per-key average using combineByKey() in Scala

```
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(_))
```

Example 4-14. Per-key average using combineByKey() in Java

```
public static class AvgCount implements Serializable {
  public AvgCount(int total, int num) {   total_ = total;   num_ = num; }
  public int total_;
  public int num_;
  public float avg() {   return total_ / (float) num_; }
}

Function<Integer, AvgCount> createAcc = new Function<Integer, AvgCount>() {
  public AvgCount call(Integer x) {
    return new AvgCount(x, 1);
  }
};
Function2<AvgCount, Integer, AvgCount> addAndCount =
  new Function2<AvgCount, Integer, AvgCount>()
  { public AvgCount call(AvgCount a, Integer x) {
    a.total_ += x;
    a.num_ += 1;
    return a;
  }
};
Function2<AvgCount, AvgCount, AvgCount> combine =
  new Function2<AvgCount, AvgCount, AvgCount>()
  { public AvgCount call(AvgCount a, AvgCount b)
  {
    a.total_ += b.total_;
    a.num_ += b.num_;
    return a;
  }
};
AvgCount initial = new AvgCount(0,0);
JavaPairRDD<String, AvgCount> avgCounts =
  nums.combineByKey(createAcc, addAndCount, combine);
Map<String, AvgCount> countMap = avgCounts.collectAsMap();
for (Entry<String, AvgCount> entry : countMap.entrySet()) {
  System.out.println(entry.getKey() + ":" + entry.getValue().avg());
}
```

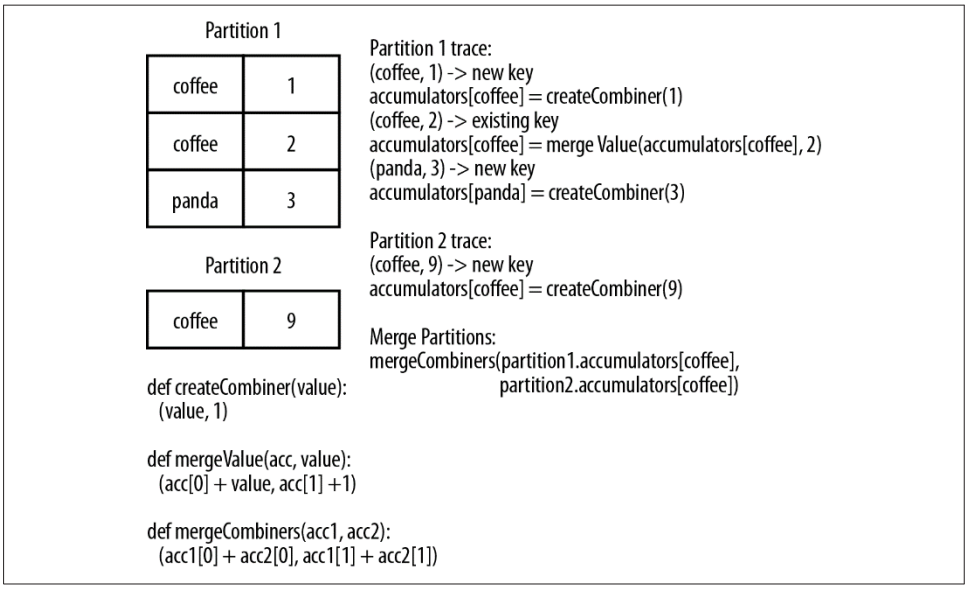


Figure 4-3. combineByKey() sample data flow

There are many options for combining our data by key. Most of them are implemented on top of `combineByKey()` but provide a simpler interface. In any case, using one of the specialized aggregation functions in Spark can be much faster than the naive approach of grouping our data and then reducing it.

> 按键值合并数据有着非常多的选项。它们大部分是以 combineByKey()为基础实现的，以提供更简单的接口。在任何情况下，使用在 Spark 内置的专门聚合函数会比使用自制的分组统计（group/reduce）来得更快。

Tuning the level of parallelism

So far we have talked about how all of our transformations are distributed, but we have not really looked at how Spark decides how to split up the work. Every RDD has a fixed number of *partitions* that determine the degree of parallelism to use when executing operations on the RDD.

> 到目前为止，我们已经提到了的数据处理都是分布式的，但我们还没有真正了解 Spark 如何决定分区并分派工作的。实际上，每个 RDD 具有确定数目的分区，分区数目决定了在 RDD 上执行操作的并发量。

When performing aggregations or grouping operations, we can ask Spark to use a specific number of partitions. Spark will always try to infer a sensible default value based on the size of your cluster, but in some cases you will want to tune the level of parallelism for better performance.

> 当进行聚合或分组操作，我们可以要求 Spark 使用分区的具体数量。Spark 总是会尝试推断基于集群的大小合理的默认值，但在某些情况下，手动调整的并发数可以获得更好的性能。

Most of the operators discussed in this chapter accept a second parameter giving the number of partitions to use when creating the grouped or aggregated RDD, as shown in Examples 4-15 and 4-16.

> 在创建分组和聚合的 RDD 时，本章提及的大多数操作函数接受第二个参数来指定分区数目，如示例 4-15 和 4-16。

Example 4-15. reduceByKey() with custom parallelism in Python

```
data = [("a", 3), ("b", 4), ("a", 1)]
sc.parallelize(data).reduceByKey(lambda x, y: x + y)     # Default parallelism
sc.parallelize(data).reduceByKey(lambda x, y: x + y, 10) # Custom parallelism
```

Example 4-16. reduceByKey() with custom parallelism in Scala

```
val data = Seq(("a", 3), ("b", 4), ("a", 1))
sc.parallelize(data).reduceByKey((x, y) => x + y)     // Default parallelism
sc.parallelize(data).reduceByKey((x, y) => x + y)     // Custom parallelism
```

Sometimes, we want to change the partitioning of an RDD outside the context of grouping and aggregation operations. For those cases, Spark provides the `repartition()` function, which shuffles the data across the network to create a new set of partitions. Keep in mind that repartitioning your data is a fairly expensive operation.Spark also has an optimized version of `repartition()` called `coalesce()` that allows avoiding data movement, but only if you are decreasing the number of RDD parti-tions. To know whether you can safely call `coalesce()`, you can check the size of the RDD using `rdd.partitions.size()` in Java/Scala and `rdd.getNumPartitions()` in Python and make sure that you are coalescing it to fewer partitions than it currentlyhas.

> 有时候，我们想在分组和聚合操作的范围之外改变一个 RDD 的分区。对于这些情况下，Spark 提供 repartition()重分区函数，用来调整（shuffle）整个网络的数据，以创建一组新的分区。请记住，repartition()是一个相当昂贵的操作。Spark 也有 repartition()的优化版本 coalesce()，coalesce()允许避免数据移动，但只有当你正在降低 RDD 分区数量才会起作用。想知道是否可以安全地调用 coalesce()，可以在 Python 中使用 rdd.getNumPartitions()，或者在 Java/Scala 中使用 rdd.partitions.size()检查 RDD 的分区数目，以确保你合并到比它目前拥有更少的分区中去。

Grouping Data

With keyed data a common use case is grouping our data by key—for example, view-

ing all of a customer's orders together.

对于带键值的数据，常见的需求是按值对数据进行分组，例如，显示一个客户的所有订单。

If our data is already keyed in the way we want, `groupByKey()` will group our data using the key in our RDD. On an RDD consisting of keys of type `K` and values of type `V`, we get back an RDD of type `[K, Iterable[V]]`.

如果数据已经按我们想要的方式带着键值，groupByKey()会根据 RDD 中的键值将数据分组。在由 K 型的键值和 V 型的内容的 RDD 上，groupByKey()会返回一个类型为[K, Iterable[V]]的 RDD。

`groupBy()` works on unpaired data or data where we want to use a different condition besides equality on the current key. It takes a function that it applies to every element in the source RDD and uses the result to determine the key.

groupBy()作用于非成对数据或不想利用键值的数据。它需要一个适用于源 RDD 中每一个元素的函数作为参数，这个函数作用于 RDD 中元素的结果即是键值 。

If you find yourself writing code where you `groupByKey()` and then use a `reduce()` or `fold()` on the values, you can probably achieve the same result more efficiently by using one of the per-key aggregation functions. Rather than reducing the RDD to an in-memory value, we reduce the data per key and get back an RDD with the reduced values corresponding to each key. For example, `rdd.reduceByKey(func)` produces the same RDD as `rdd.groupByKey().mapValues(value => value.reduce(func))` but is more efficient as it avoids the step of creating a list of values for each key.

如果你的代码首先进行 groupByKey()，然后再对数据作 reduce()或 fold()处理，那么也许可以更有效地使用“每键值（per-key）”聚集函数达到同样的效果。这不是 reduce 处理 RDD 放到内存中，而是 reduce 每个键值的数据并取回 RDD 中每个键值对应的 reduce 结果 。例如，rdd.reduceByKey(func)来产生的 RDD 与下面的操作相同：rdd.groupByKey().mapValues(value => value.reduce(func))，但是前者更为有效的，因为它避免了为每一个键值产生一个列表的步骤。

In addition to grouping data from a single RDD, we can group data sharing the same key from multiple RDDs using a function called `cogroup()`. `cogroup()` over two RDDs sharing the same key type, `K`, with the respective value types `V` and `W` gives us back `RDD[(K, (Iterable[V], Iterable[W]))]`. If one of the RDDs doesn't have elements for a given key that is present in the other RDD, the corresponding `Iterable` is simply empty. `cogroup()` gives us the power to group data from multiple RDDs.

除了从单个 RDD 对数据进行分组，也可使用 cogroup()在若干个 RDD 间收集相同键值对数据进行分组。cogroup()在两个 RDD 共享同一键型 K，以及对两个分区内容类型的 V 和 W，返回 RDD[(K, (Iterable[V], Iterable[W]))]。如果其中一个 RDD 不具有其他 RDD 中指定键值 K 的元素，对应的 Iterable 为空。cogroup()让我们能对多个 RDD 的数据进行分组。

`cogroup()` is used as a building block for the joins we discuss in the next section.

cogroup()用来作数据连接（join）操作的方法我们会在下一节讨论。

`cogroup()` can be used for much more than just implementing joins. We can also use it to implement intersect by key. Additionally, `cogroup()` can work on three or more RDDs at once.

cogroup()不仅仅能用于连接。也可以用来实现交集（interesect）。同时，cogroup()更可以应用在多于三个 RDD 上。

Joins

Some of the most useful operations we get with keyed data comes from using it together with other keyed data. Joining data together is probably one of the most common operations on a pair RDD, and we have a full range of options including right and left outer joins, cross joins, and inner joins.

我们常用的操作是从一组键值数据中查询另外一组键值数据的信息。联合数据可能就是 pair RDD 上最常见的操作，在 Spark 中我们有完善的操作函数，包括了左右外连接，交叉连接和内部连接。

The simple `join` operator is an inner join.[1] Only keys that are present in both pair RDDs are output. When there are multiple values for the same key in one of the inputs, the resulting pair RDD will have an entry for every possible pair of values with that key from the two input RDDs. A simple way to understand this is by looking at Example 4-17.

简单的连接运算符是内部连接。内部连接只输出同时存在于两个 pair RDD 的键值。当有一个键值在其中一个输入中含有多个值，所得到的 pair RDD 将返回每一个符合要求的数值。要了解具体请看实例 4-17。

Example 4-17. Scala shell inner join

```
storeAddress = {
  (Store("Ritual"), "1026 Valencia St"), (Store("Philz"), "748 Van Ness Ave"),
  (Store("Philz"), "3101 24th St"), (Store("Starbucks"), "Seattle")}

storeRating = {
  (Store("Ritual"), 4.9), (Store("Philz"), 4.8))}

storeAddress.join(storeRating) ==
  { (Store("Ritual"), ("1026 Valencia St",
  4.9)),
  (Store("Philz"), ("748 Van Ness Ave", 4.8)),
  (Store("Philz"), ("3101 24th St", 4.8))}
```

1 "Join" is a database term for combining fields from two tables using common values.

Sometimes we don't need the key to be present in both RDDs to want it in our result. For example, if we were joining customer information with recommendations we might not want to drop customers if there were not any recommendations yet. `leftOuterJoin(other)` and `rightOuterJoin(other)` both join pair RDDs together by key, where one of the pair RDDs can be missing the key.

有时候，我们不仅仅需要同时出现在两个 RDD 的键值对应项。例如，如果我们连接推荐的客户信息，我们可能不想丢失没有任何推荐的客户 。leftOuterJoin(other)和 rightOuterJoin(other)两者都会连接 pair RDD，其中的一个 pair RDD 允许缺少键值。

With `leftOuterJoin()` the resulting pair RDD has entries for each key in the source RDD. The value associated with each key in the result is a tuple of the value from the source RDD and an `Option` (or `Optional` in Java) for the value from the other pair RDD. In Python, if a value isn't present `None` is used; and if the value is present the regular value, without any wrapper, is used. As with `join()`, we can have multiple entries for each key; when this occurs, we get the Cartesian product between the two lists of values.

leftOuterJoin()得到的 pair RDD 具有源 RDD 中的每个键的条目。结果中的每个键值关联的值是从源 RDD 的值和一个 Option（或 Java 中的 Optional），Option 用来表示从另一个 pair RDD 得到的值。在 Python 中，如果某个值不存在则用 None 表示；如果该值存在，则用常规值表示。对于 join()方法，每个键值可能有多个条目；此时，我们得到两个列表的笛卡尔积。

`Optional` is part of Google's Guava library and represents a possibly missing value. We can check `isPresent()` to see if it's set, and `get()` will return the contained instance provided data is present.

Optional 是 Google 的 Guava 库的一部分，表示可能缺失值。Optional 的 isPresent()可以确认 Optional 值是否存在，而 get()将返回所包含的实例 。

`rightOuterJoin()` is almost identical to `leftOuterJoin()` except the key must be present in the other RDD and the tuple has an option for the source rather than the other RDD.

rightOuterJoin()几乎与 leftOuterJoin()相同，只是右边 RDD 键值必须存在，而左边 RDD 的键值为可选 。

We can revisit Example 4-17 and do a `leftOuterJoin()` and a `rightOuterJoin()` between the two pair RDDs we used to illustrate `join()` in Example 4-18.

重温示例 4-17，再看示例 4-18 中 leftOuterJoin()和 rightOuterJoin()的用法让我们更好的理解 join()函数。

Example 4-18. leftOuterJoin() and rightOuterJoin()

```
storeAddress.leftOuterJoin(storeRating) ==
{(Store("Ritual"),("1026 Valencia  St",Some(4.9))),
  (Store("Starbucks"),("Seattle",None)),
  (Store("Philz"),("748 Van Ness Ave",Some(4.8))),
  (Store("Philz"),("3101 24th  St",Some(4.8)))}

storeAddress.rightOuterJoin(storeRating) ==
{(Store("Ritual"),(Some("1026 Valencia  St"),4.9)),
  (Store("Philz"),(Some("748 Van Ness Ave"),4.8)),
  (Store("Philz"), (Some("3101 24th St"),4.8))}
```

Sorting Data

Having sorted data is quite useful in many cases, especially when you're producing downstream output. We can sort an RDD with key/value pairs provided that there is an ordering defined on the key. Once we have sorted our data, any subsequent call on the sorted data to `collect()` or `save()` will result in ordered data.

有序数据在许多情况下非常有用，特别是当生成流型输出时。在 key/value 的 RDD 上排序需要在键值上指定顺序 。一旦我们已经排序了数据， 任何后续调用 collect()或 save()都将产出有序的数据。

Since we often want our RDDs in the reverse order, the `sortByKey()` function takes a parameter called `ascending` indicating whether we want it in ascending order (it defaults to `true`). Sometimes we want a different sort order entirely, and to support this we can provide our own comparison function. In Examples 4-19 through 4-21, we will sort our RDD by converting the integers to strings and using the string comparison functions.

因为我们常常也想要得到相序的 RDD，该 sortByKey()函数接受一个名为 ascending 的参数，表示希望它以升序排列输出（默认为真）。有时我们需要一个自定义的排序顺序，该 sortByKey()函数可以指定自定义的比较函数。在示例 4-19 和 4-21 中，我们将通过转换整数为字符串，并且使用字符串的比较函数来排序我们 RDD。

Example 4-19. Custom sort order in Python, sorting integers as if strings

```
rdd.sortByKey(ascending=True, numPartitions=None, keyfunc = lambda x: str(x))
```

Example 4-20. Custom sort order in Scala, sorting integers as if strings

```
val input: RDD[(Int, Venue)] = ...
implicit val sortIntegersByString = new Ordering[Int] {
  override def compare(a: Int, b: Int) = a.toString.compare(b.toString)
}
rdd.sortByKey()
```

Example 4-21. Custom sort order in Java, sorting integers as if strings

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

Actions Available on Pair RDDs

As with the transformations, all of the traditional actions available on the base RDD are also available on pair RDDs. Some additional actions are available on pair RDDs to take advantage of the key/value nature of the data; these are listed in Table 4-3.

> 对于转换操作，基础 RDD 的操作在 pair RDD 上也有效。而 pair RDD 利用数据的 key/value 的性质支持更多的操作；这些操作列在表 4-3 中。

Table 4-3. Actions on pair RDDs (example ({(1, 2), (3, 4), (3, 6)}))

Function	Description	Example	Result
countByKey()	Count the number of elements for each key.	rdd.countByKey()	{(1, 1), (3, 2)}
collectAsMap()	Collect the result as a map to provide easy lookup.	rdd.collectAsMap()	Map{(1, 2), (3, 4), (3, 6)}
lookup(key)	Return all values associated with the provided key.	rdd.lookup(3)	[4, 6]

There are also multiple other actions on pair RDDs that save the RDD, which we will describe in Chapter 5.

pair RDD 有更多用来保存 RDD 的操作，我们在第 5 章里介绍。

• Data Partitioning (Advanced)

The final Spark feature we will discuss in this chapter is how to control datasets' partitioning across nodes. In a distributed program, communication is very expensive, so laying out data to minimize network traffic can greatly improve performance. Much like how a single-node program needs to choose the right data structure for a collection of records, Spark programs can choose to control their RDDs' partitioning to reduce communication. Partitioning will not be helpful in all applications—for example, if a given RDD is scanned only once, there is no point in partitioning it in advance. It is useful only when a dataset is reused *multiple times* in key-oriented operations such as joins. We will give some examples shortly.

本章讨论的最后一个 Spark 的特性， 是如何控制数据集的跨节点分区。在一个分布式系统中，通信是非常昂贵的。因此合理地分布数据能减少网络通信，从而大大提高性能。就像单个节点的程序如何选择合适的数据结构，Spark 程序可以选择自己的 RDD 分区分布来降低通信。指定分区并不总是有用，例如，如果对给定的 RDD 扫描一次，指定分区是没有意义的。只有当一个数据集重复多次使用键值操作才起作用，例如连接操作。我们接下来会举一些例子。

Spark's partitioning is available on all RDDs of key/value pairs, and causes the system to group elements based on a function of each key. Although Spark does not give explicit control of which worker node each key goes to (partly because the system is designed to work even if specific nodes fail), it lets the program ensure that a *set* of keys will appear together on *some* node. For example, you might choose to hash-partition an RDD into 100 partitions so that keys that have the same hash value modulo 100 appear on the same node. Or you might range-partition the RDD into sorted ranges of keys so that elements with keys in the same range appear on the same node.

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。尽管 Spark 中不能显式地控制对应键值的 worker node 去哪个分区（部分是因为 node 的容错机制），Spark 允许程序限定一组键值出现在指定的节点上。例如，可以指定一个哈希函数把 RDD 分割到 100 个分区，使得具有相同的散列值模 100 的数据出现在同一个节点上。或者指定范围让 RDD 根据键值排序进行区间，这样相邻范围的元素会出现在同一节点上。

As a simple example, consider an application that keeps a large table of user information in memory—say, an RDD of (`UserID`, `UserInfo`) pairs, where `UserInfo` contains a list of topics the user is subscribed to. The application periodically combines

this table with a smaller file representing events that happened in the past five minutes—say, a table of (UserID, LinkInfo) pairs for users who have clicked a link on a website in those five minutes. For example, we may wish to count how many users visited a link that was *not* to one of their subscribed topics. We can perform this combination with Spark's join() operation, which can be used to group the UserInfo and LinkInfo pairs for each UserID by key. Our application would look like Example 4-22.

一个简单的例子，考虑一个程序，在内存中维护一张用户信息的大表，（UserID，UserInfo）的 RDD，这里的 UserInfo 包含了用户的订阅列表。该程序定期根据一个记录过去 5 分钟发生时间的文件来更新此表，这个文件保存了这五分钟的网站链接点击信息：（用户 ID，LinkInfo）。例如，我们希望来算算多少用户访问了不在订阅列表的链接。我们可以用 Spark 的 join()函数来分组 UserInfo 和 LinkInfo，基于每个 UserID。具体程序如示例 4-22。

Example 4-22. Scala simple application

```
// 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)
}
```

This code will run fine as is, but it will be inefficient. This is because the join() operation, called each time processNewLogs() is invoked, does not know anything about how the keys are partitioned in the datasets. By default, this operation will hash all the keys of *both* datasets, sending elements with the same key hash across the network to the same machine, and then join together the elements with the same key on that machine (see Figure 4-4). Because we expect the userData table to be much larger than the small log of events seen every five minutes, this wastes a lot of work: the userData table is hashed and shuffled across the network on every call, even though it doesn'tchange.

这个代码倒是能工作，但是效率不足。这是因为每次调用 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（见图 4-4 ）。我们发现 userData 表为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。

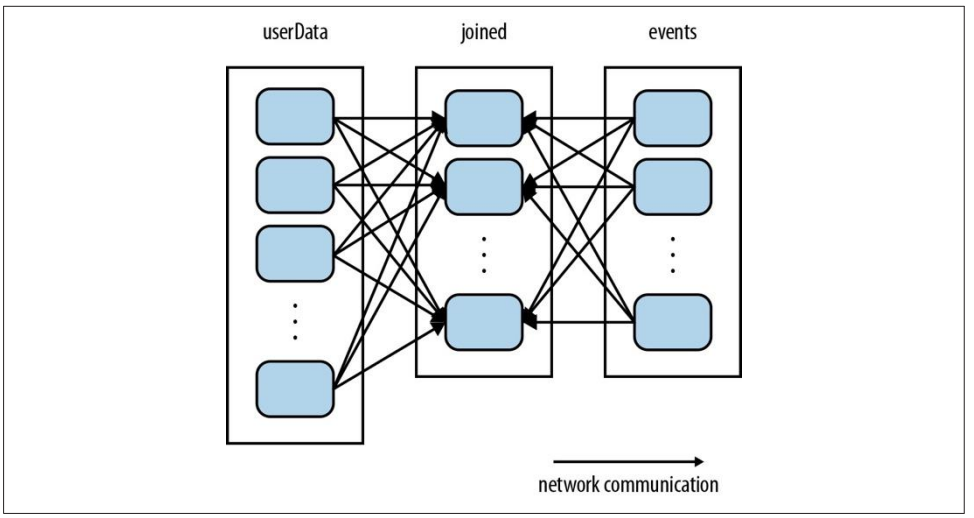


Figure 4-4. Each join of userData and events without using partitionBy()

Fixing this is simple: just use the `partitionBy()` transformation on `userData` to hash-partition it at the start of the program. We do this by passing a `spark.HashPartitioner` object to `partitionBy`, as shown in Example 4-23.

改进这个算法很简单：只需在程序开始时，使用 partitionBy()转换 userData 的散列分区。我们通过传递 spark.HashPartitioner 对象给 partitionBy()，如例 4-23。

Example 4-23. Scala custom partitioner

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

The `processNewLogs()` method can remain unchanged: the `events` RDD is local to `processNewLogs()`, and is used only once within this method, so there is no advantage in specifying a partitioner for `events`. Because we called `partitionBy()` when building `userData`, Spark will now know that it is hash-partitioned, and calls to `join()` on it will take advantage of this information. In particular, when we call `userData.join(events)`, Spark will shuffle only the `events` RDD, sending events with each particular `UserID` to the machine that contains the corresponding hash partition of `userData` (see Figure 4-5). The result is that a lot less data is communicated over the network, and the program runs significantly faster.

> processNewLogs()方法可以保持不变：事件 RDD 对于 processNewLogs()是本地操作，在其中只使用一次，为事件指定分区不会带来提升。因为我们调用 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（见图 4-5）。其结果是，减少了许多网络通信，让程序更快地运行。

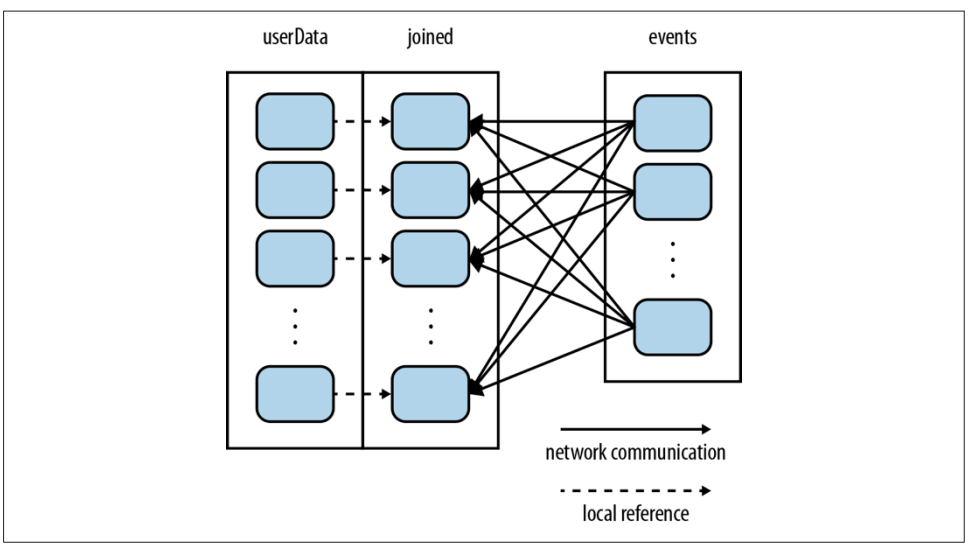


Figure 4-5. Each join of userData and events using partitionBy()

Note that `partitionBy()` is a transformation, so it always returns a *new* RDD—it does not change the original RDD in place. RDDs can never be modified once created. Therefore it is important to persist and save as `userData` the result of `parti`

tionBy(), not the original sequenceFile(). Also, the 100 passed to partitionBy() represents the number of partitions, which will control how many parallel tasks perform further operations on the RDD (e.g., joins); in general, make this at least as large as the number of cores in your cluster.

需要注意的是 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 partionBy()的结果 userData 非常重要的，而不是原来 sequenceFile()。而且，传递给 partitionBy()的 100 表示分区的数目，这会控制 RDD 上的并行任务的数目（例如 join）；通常该值应该大于 CPU 核心的数目。

Failure to persist an RDD after it has been transformed with partitionBy() will cause subsequent uses of the RDD to repeat the partitioning of the data. Without persistence, use of the partitioned RDD will cause reevaluation of the RDDs complete lineage. That would negate the advantage of partitionBy(), resulting in repeated partitioning and shuffling of data across the network, similar to what occurs without any specified partitioner.

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。

In fact, many other Spark operations automatically result in an RDD with known partitioning information, and many operations other than join() will take advantage of this information. For example, sortByKey() and groupByKey() will result in range-partitioned and hash-partitioned RDDs, respectively. On the other hand, operations like map() cause the new RDD to *forget* the parent's partitioning information, because such operations could theoretically modify the key of each record. The next few sections describe how to determine how an RDD is partitioned, and exactly how partitioning affects the various Spark operations.

事实上， 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。接下来的几节将介绍如何确定的 RDD 的分区，以及如何分区会影响到不同 Spark 操作。

Partitioning in Java and Python

Spark's Java and Python APIs benefit from partitioning in the same way as the Scala API. However, in Python, you cannot pass a HashPartitioner object to partitionBy; instead, you just pass the number

of partitions desired (e.g., `rdd.partitionBy(100)`).

Spark 的 Java 和 Python API 的分区策略与 Scala API 类似。不过在 Python 中，你不能传递一个哈希分区对象给 partitionBy；你只需传递所需的分区数目（例如，rdd.partitionBy(100)）。

Determining an RDD's Partitioner

In Scala and Java, you can determine how an RDD is partitioned using its `partitioner` property (or `partitioner()` method in Java).[2] This returns a `scala.Option` object, which is a Scala class for a container that may or may not contain one item. You can call `isDefined()` on the `Option` to check whether it has a value, and `get()` to get this value. If present, the value will be a `spark.Partitioner` object. This is essentially a function telling the RDD which partition each key goes into; we'll talk more about this later.

在 Scala 和 Java 中，使用 RDD 的 partitioner 属性（Java 中使用 partitioner()方法）来决定 RDD 的分区策略。此调用会返回一个 scala.Option 对象，它是 Scala 的容器类，可能包含一个元素。您可以在 Option 上调用 isDefined()以检查它是否有内容，并使用 get()来得到它的值。如果存在的话，该值将是一个 spark.Partitioner 对象。这是本质上是一个判断每个键进入哪个 RDD 分区的函数；我们将在后面详细讨论这一点。

The `partitioner` property is a great way to test in the Spark shell how different Spark operations affect partitioning, and to check that the operations you want to do in your program will yield the right result (see Example 4-24).

该 partitioner 属性是用来在 Spark shell 中测试 不同的 Spark 操作如何影响分区的好方法，亦可以用来测试你计划用在程序中的操作函数（见例 4-24）。

2 The Python API does not yet offer a way to query partitioners, though it still uses them internally.

Example 4-24. Determining partitioner of an RDD

```
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)
```

In this short session, we created an RDD of `(Int, Int)` pairs, which initially have no partitioning information (an `Option` with value `None`). We then created a second RDD by hash-partitioning the first. If we actually wanted to use `partitioned` in further operations, then we should have appended `persist()` to the third line of input, in which `partitioned` is defined. This is for the same reason that we needed `persist()` for `userData` in the previous example: without `persist()`, subsequent RDD actions will evaluate the entire lineage of `partitioned`, which will cause `pairs` to be hash-partitioned over and over.

> 在这一段代码中，我们创建了(Int, Int)型的 RDD，最初没有分区信息（带有 None 的 Option）。然后，我们创建了第二个 RDD：即在第一个 RDD 被哈希分区的版本。如果我们确实想在进一步的操作中使用分区，我们应该有在输入第三行加上 persist()，并在其中定义分区。这就是需要在 userData 用 persist()的原因：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。

Operations That Benefit from Partitioning

Many of Spark's operations involve shuffling data by key across the network. All of these will benefit from partitioning. As of Spark 1.0, the operations that benefit from partitioning are `cogroup()`, `groupWith()`, `join()`, `leftOuterJoin()`, `rightOuterJoin()`, `groupByKey()`, `reduceByKey()`, `combineByKey()`, and `lookup()`.

> Spark 的许多操作都会导致数据在网络中传输并重新布局。操作能受益于已有的分区策略。在 Spark 1.0 中，能从分区策略中获益的操作有：cogroup()，groupWith()，join()，leftOuterJoin()，rightOuterJoin()，groupByKey()，reduceByKey()，combineByKey()，和 lookup()。

For operations that act on a single RDD, such as `reduceByKey()`, running on a pre-partitioned RDD will cause all the values for each key to be computed *locally* on a single machine, requiring only the final, locally reduced value to be sent from

each worker node back to the master. For binary operations, such as `cogroup()` and `join()`, pre-partitioning will cause at least one of the RDDs (the one with the known partitioner) to not be shuffled. If both RDDs have the *same* partitioner, and if they are cached on the same machines (e.g., one was created using `mapValues()` on the other, which preserves keys and partitioning) or if one of them has not yet been com- puted, then no shuffling across the network will occur.

> 在单一 RDD 作用的操作，如 reduceByKey()，如果 RDD 已经定好分区策略，其上每个键值的对应结果都在本地单机计算，仅需要在最后，在本地作 reduce 并把结果从每个 worker 传回 master。对于多个 RDD 的操作，如 cogroup()和 join()，预分区将让至少一个（带有分区策略的）RDD 不再在网络内重排。如果两个 RDD 具有相同的分区策略，并且它们缓存在相同机器（例如，一个 RDD 在另外一个 RDD 基础上用 mapValues()创建，继承了键值和分区策略），或者如果他们中的一个尚未被计算，都不会使它们在网络内重排。

Operations That Affect Partitioning

Spark knows internally how each of its operations affects partitioning, and automatically sets the `partitioner` on RDDs created by operations that partition the data. For example, suppose you called `join()` to join two RDDs; because the elements with the same key have been hashed to the same machine, Spark knows that the result is hash-partitioned, and operations like `reduceByKey()` on the join result are going to be significantly faster.

> Spark 内部控制每一个操作如何影响分区，并自动根据操作该分区中的数据设置创建 RDDS 的分区。例如，调用 join()方法来连接两个 RDD；因为具有相同键的元素已被散列到相同的机器，Spark 知道结果是散列分区，还有类似 reduceByKey()上的连接操作结果也将会更快。

The flipside, however, is that for transformations that *cannot* be guaranteed to produce a known partitioning, the output RDD will not have a `partitioner` set. For example, if you call `map()` on a hash-partitioned RDD of key/value pairs, the function passed to `map()` can in theory change the key of each element, so the result will not have a `partitioner`. Spark does not analyze your functions to check whether they retain the key. Instead, it provides two other operations, `mapValues()` and `flatMapValues()`, which guarantee that each tuple's key remains the same.

> 另一方面，对于那些不能保证输出到已知分区的转化操作，输出 RDD 将不具有分区器集。例如，如果在散列分区 key/value RDD 调用 map()，传递给 map()的函数理论上会改变每个元素的键，所以结果将不具有分区器。Spark 不分析你的函数并检查它们是否改变键值。相反，它提供了两个操作，mapValues()和 flatMapValue()，保证每个键值保持不变。

All that said, here are all the operations that result in a partitioner being set on the

output RDD: `cogroup()`, `groupWith()`, `join()`, `leftOuterJoin()`, `rightOuterJoin()`, `groupByKey()`, `reduceByKey()`, `combineByKey()`, `partitionBy()`, `sort()`, `mapValues()` (if the parent RDD has a partitioner), `flatMapValues()` (if parent has a partitioner), and `filter()` (if parent has a partitioner). All *other* operations will produce a result with no partitioner.

> 综上所述，以下操作会使输出 RDD 设置上一个分区器：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。

Finally, for binary operations, *which* partitioner is set on the output depends on the parent RDDs' partitioners. By default, it is a hash partitioner, with the number of partitions set to the level of parallelism of the operation. However, if one of the parents has a `partitioner` set, it will be that partitioner; and if both parents have a `partitioner` set, it will be the partitioner of the first parent.

> 最后，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割器，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，结果便会设置上这个分区器；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

As an example of a more involved algorithm that can benefit from RDD partitioning, we consider PageRank. The PageRank algorithm, named after Google's Larry Page, aims to assign a measure of importance (a "rank") to each document in a set based on how many documents have links to it. It can be used to rank web pages, of course, but also scientific articles, or influential users in a social network.

> 本节以更复杂的 PageRank 算法为例来解释如何从 RDD 分区策略获益。PageRank 算法，以 Google 的 Larry Page 的名字命名，它基于被链接的次数来计算页面的重要性（rank），进一步用来排序网页、科学论文、或者用户影响力。

PageRank is an iterative algorithm that performs many joins, so it is a good use case for RDD partitioning. The algorithm maintains two datasets: one of `(pageID, linkList)` elements containing the list of neighbors of each page, and one of `(pageID, rank)` elements containing the current rank for each page. It proceeds as follows:

> PageRank 是一种进行多次连接操作的迭代算法，它能够很好地展示 RDD 分区策略。该算法维护两个数据集：1）[pageID，链接列表]，包含每个页面的相邻页面；2）

[pageID，rank]，包含每个页面的 rank。算法执行过程如下：

1. Initialize each page's rank to 1.0.
2. On each iteration, have page p send a contribution of `rank(p)/numNeighbors(p)` to its neighbors (the pages it has links to).
3. Set each page's rank to `0.15 + 0.85 *contributionsReceived`.

1. 初始化每个页面的 rank 为 1.0。

2. 在每次迭代中，页面 p 为它的邻居（它有链接指向的页面）的 rank 值贡献 rank(P)/ numNeighbors(P)。

3. 将每个页面的 rank 设为 0.15+0.85* contributionsReceived。

The last two steps repeat for several iterations, during which the algorithm will converge to the correct PageRank value for each page. In practice, it's typical to run about 10 iterations.

后两个步骤多次迭代，在此期间，该算法将收敛到到每个页面的 PageRank 值。实际情况中通常需要 10 轮迭代。

Example 4-25 gives the code to implement PageRank in Spark.

Example 4-25. Scala PageRank

```
// 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")
```

That's it! The algorithm starts with a `ranks` RDD initialized at 1.0 for each element,and keeps updating the `ranks` variable on each iteration. The body of PageRank is pretty simple to express in Spark: it first does a `join()` between the current `ranks`RDD and the static `links` one, in order to obtain the link list and rank for each pageID together, then uses this in a `flatMap` to create "contribution" values to send to each of the page's neighbors. We then add up these values by page ID (i.e., by thepage receiving the contribution) and set that page's rank to `0.15 + 0.85 * contribu tionsReceived`.

该算法开始于一个所有 rank 初始化为 1.0 的 RDD，每次迭代更新 rank 的值。PageRank 算法在 Spark 中编写非常简单：首先在 ranks 和 links 做一个 join()，以获得每个 pageID 的链接列表和 rank，然后在 flatMap 中使用刚才的结果计算发送给每个邻居的“贡献”。然后，把这些值按 pageID 加起来（即由页面接收的贡献）得到 contributionsReceived，并最后令该网页的 rank 为 0.15+0.85* contributionsReceived。

Although the code itself is simple, the example does several things to ensure that the

RDDs are partitioned in an efficient way, and to minimize communication:

尽管代码本身很简单，这个实例确实通过配置让 RDDS 以有效的方式进行分区，以减少网络通信：

1. Notice that the `links` RDD is joined against `ranks` on each iteration. Since `links` is a static dataset, we partition it at the start with `partitionBy()`, so that it does not need to be shuffled across the network. In practice, the `links` RDD is also likely to be much larger in terms of bytes than `ranks`, since it contains a list of neighbors for each page ID instead of just a `Double`, so this optimization saves considerable network traffic over a simple implementation of PageRank (e.g., in plain MapReduce).

1. 注意到 links 的 RDD 已在每次迭代中 join 了 ranks。因为 links 是静态数据集，在开始时用 partitionBy()对它进行分区，因此它不需要通过网络重排。实际上，links RDD 可能比 ranks RDD 大很多，因为它包含每个页面的邻居列表，而不是仅仅是二元组的列表，所以这种优化节省相当大的网络通信量并简单实现了 PageRank。（例如，使用 MapReduce）。

2. For the same reason, we call `persist()` on `links` to keep it in RAM across iterations.

2. 出于同样的原因在 links 上调用 persist()，以保证它在迭代过程中维持在内存里。

3. When we first create `ranks`, we use `mapValues()` instead of `map()` to preserve the partitioning of the parent RDD (`links`), so that our first join against it is cheap.

3. 当我们第一次创建 ranks，使用了 mapValues()来保存父 RDD（即 links）的分区，而没有用 map()，这使得第一个 join 的开销减少很多。

4. In the loop body, we follow our `reduceByKey()` with `mapValues()`; because the result of `reduceByKey()` is already hash-partitioned, this will make it more efficient to join the mapped result against `links` on the next iteration.

4. 在循环体中，我们使用 reduceByKey()与 mapValues()；因为 reduceByKey()的结果已经散列分区了，这将使下一次迭代 map 的结果 join 到 links 更加高效

To maximize the potential for partitioning-related optimizations, you should use `mapValues()` or `flatMapValues()` whenever you

are not changing an element's key.

为了最大限度地利用分区相关的优化，建议使用 mapValues()或 flatMapValues()，只要你不更改元素的键值。

Custom Partitioners

While Spark's HashPartitioner and RangePartitioner are well suited to many use cases, Spark also allows you to tune how an RDD is partitioned by providing a custom Partitioner object. This can help you further reduce communication by taking advantage of domain-specific knowledge.

虽然 Spark 的 HashPartitioner 和 RangePartitioner 适合许多场景，Spark 还允许自定义分区策略，可以进一步根据领域特性来减少网络通信开销。

For example, suppose we wanted to run the PageRank algorithm in the previous section on a set of web pages. Here each page's ID (the key in our RDD) will be its URL. Using a simple hash function to do the partitioning, pages with similar URLs (e.g., *http://www.cnn.com/WORLD* and *http://www.cnn.com/US*) might be hashed to completely different nodes. However, we know that web pages within the same domain tend to link to each other a lot. Because PageRank needs to send a message from each page to each of its neighbors on each iteration, it helps to group these pages into the same partition. We can do this with a custom Partitioner that looks at just the domain name instead of the whole URL.

假设我们想在网页数据上运行 PageRank 算法。每个页面的 ID（RDD 中的键值）将是它的 URL。使用一个简单的散列函数作分区划分，类似的网址页面（例如，http://www.cnn.com/WORLD 和 http://www.cnn.com/US）可能会被散列到完全不同的节点。但是，我们知道，相同域中的网页往往会相互链接。由于 PageRank 需要在每次迭代中发送消息给相邻页面，最好把这些页面划分在同一个分区。这里可以用自定义的分区策略：仅仅使用域名，而不是整个 URL。

To implement a custom partitioner, you need to subclass the org.apache.spark.Partitioner class and implement three methods:

要实现自定义分区，需要继承 org.apache.spark.Partitioner 类，并实现三个方法：

- numPartitions: Int, which returns the number of partitions you will create.
- getPartition(key: Any): Int, which returns the partition ID (0 to numPartitions-1) for a given key.

- `equals()`, the standard Java equality method. This is important to implement because Spark will need to test your `Partitioner` object against other instances of itself when it decides whether two of your RDDs are partitioned the same way!

- numPartitions: Int，返回创建的分区数目。
- getPartition(key:Any):Int，返回指定键值的分区 ID（0 至 numPartitions-1）。
- equals()，标准的 Java 相等方法。其实现非常重要，因为 Spark 需要检查不同 Spark 实例的分区器是否相等，并证明两个 RDDs 是用相同策略分区的。

One gotcha is that if you rely on Java's `hashCode()` method in your algorithm, it can return negative numbers. You need to be careful to ensure that `getPartition()` always returns a nonnegative result.

有一个问题是，如果你依赖于 Java 的 hashCode()方法，它可能返回负数。你必须要小心确保 getPartition()总是返回一个非负的结果。

Example 4-26 shows how we would write the domain-name-based partitioner sketched previously, which hashes only the domain name of each URL.

例 4-26 显示了如何编写领域特定的分区器，为每个 URL 的仅域名部分做哈希。

Example 4-26. Scala custom partitioner

```
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 //Make it non-negative
    } else
      { code
    }
  }
  // Java equals method to let Spark compare our Partitioner objects
  override def equals(other: Any): Boolean = other match {
    case dnp: DomainNamePartitioner =>
      dnp.numPartitions == numPartitions
    case _ =>
      false
  }
}
```

Note that in the `equals()` method, we used Scala's pattern matching operator (`match`)

to test whether `other` is a `DomainNamePartitioner`, and cast it if so; this is the same as using `instanceof()` in Java.

需要注意的是，在 equals()方法中，我们使用 Scala 的模式匹配运算符（match），以测试另一个是否为 DomainNamePartitioner；这与 Java 中的 instanceof()一致。

Using a custom `Partitioner` is easy: just pass it to the `partitionBy()` method. Many of the shuffle-based methods in Spark, such as `join()` and `groupByKey()`, can also take an optional `Partitioner` object to control the partitioning of the output.

使用自定义分区程序很简单：只需将分区器传递给 partitionBy()方法。Spark 中有许多基于分区间重排的方法，如 join()和 groupByKey()，这些方法也可以使用一个可选的分区器对象来控制输出的分区策略。

Creating a custom `Partitioner` in Java is very similar to Scala: just extend the `spark.Partitioner` class and implement the required methods.

> 创建 Java 中的自定义分区程序是非常相似的斯卡拉：刚刚扩展了 spark.Partitioner 类，并实现所需的方法。

In Python, you do not extend a `Partitioner` class, but instead pass a hash function as an additional argument to `RDD.partitionBy()`. Example 4-27 demonstrates.

> 在 Python 中，你不用扩展分区器程序类，而是需要传递一个哈希函数作为参数给 RDD.partitionBy()。如实例 4-27 所示。

Example 4-27. Python custom partitioner

```
import urlparse

def hash_domain(url):
  return  hash(urlparse.urlparse(url).netloc)

rdd.partitionBy(20, hash_domain)  # Create 20 partitions
```

Note that the hash function you pass will be compared *by identity* to that of other RDDs. If you want to partition multiple RDDs with the same partitioner, pass the same function object (e.g., a global function) instead of creating a new `lambda` for each one!

> 请注意，你传递的哈希函数也将与其他 RDD 的进行比较。如果你要用相同的分区策略来对多个 RDDS 作分区，通过传递相同的函数对象（例如，一个全局函数），而不是每次创建一个新的 lambda！

Conclusion

In this chapter, we have seen how to work with key/value data using the specialized functions available in Spark. The techniques from Chapter 3 also still work on our pair RDDs. In the next chapter, we will look at how to load and save data.

> 在本章中，我们看到如何使用 Spark 中的函数来处理 key/value 数据，即 pair RDD。第 3 章中的技术也仍适用于我们 pair RDD。在下一章中，我们将着眼于如何加载和保存数据。