

## 第 7 章 Spark Core 基础计算框架

### ★本章导读★

本章将正式进入 Spark 的篇章，对 Spark Core 的核心部分进行讲解，即 RDD，Spark 的运算和计算都是围绕 RDD 来实现的。本章以实战为主，尽量把最核心的概念介绍给读者，并通过完整的代码实战方式使读者更好的理解 Spark RDD，同时也为后续的相关编程操作奠定基础。

### ★知识要点★

通过本章内容的学习，读者将掌握以下知识：

- RDD 的原理、工作机制和特点
- 不同类型 RDD 常用的 Transform 方法和 Action 方法
- RDD 持久化原理

## 7.1 RDD 介绍

### 7.1.1 什么是 RDD

RDD (Resilient Distributed Dataset) 叫做弹性分布式数据集，是 Spark 中最基本的数据抽象。RDD 是一个抽象类，它代表一个弹性的、不可变、可分区、里面的元素可并行计算的集合。RDD 有 3 种类型的运算，分别是 Transform 运算、Action 运算和持久化。RDD 也是 Spark 最重要的核心部分，整个 Spark 的运算和计算都是围绕 RDD 实现的。

RDD 可以简单的看成是一个“数组”，对 RDD 的操作也只需要调用相应的方法即可实现，它与一般意义上的数组的区别在于，RDD 是分布式存储的，可以更好地利用大数据集群的资源，并且 RDD 是运行在内存中的。分布式存储的最大好处就是可以让数据在不同的工作节点上并行存储，使得我们在需要数据的时候并行运行，从而提高计算效率。

此外，RDD 是“弹性”的，这是指 RDD 的数据存储方式，即数据在节点中进行存储的时候，既可以使用内存也可以使用磁盘，这给开发者提供了非常大的自由度，对于需要反复计算的数据可以存储在内存中，便于提高计算效率，对于使用较少的数据可以存储在硬盘中，当需要使用的时候再读取计算。除此之外，“弹性”还有另一层含义，即容错性，RDD 的容错性非常强，这里的容错性是指 Spark 在计算过程中不会因为某个子节点的计算错误而使得整个任务计算失败而从头开始计算。如果某个节点发生错误时，RDD 会自动将该节点上的计算重新运行。

### 7.1.2 RDD 特点与特性

RDD 是一种弹性分布式数据集，具有如下特点。

1. 存储的弹性：内存与磁盘的自动切换；
2. 容错的弹性：数据丢失可以自动恢复；
3. 计算的弹性：计算出错重试机制；
4. 分片的弹性：可根据需要重新分片。

同样，RDD 具有五大特性。

1. **A list of partitions**

RDD 逻辑上是分区的，每个分区的数据是抽象存在的，计算的时候会通过一个 `compute` 函数得到每个分区的数据。RDD 由很多 `partition` 构成，在 Spark 中，RDD 有多少 `partition` 就对应有多少个 `task` 来执行。

2. **A function for computing each split**

一个函数计算每一个分片，RDD 的每个 `partition` 上面都会有 `function`，也就是函数应用，其作用是实现 RDD 之间 `partition` 的转换。Spark 中的 RDD 的计算是以分片为单位的，每个 RDD 都会实现 `compute` 函数以达到这个目的。`compute` 函数会对迭代器进行复合，不需要保存每次计算的结果。

3. **A list of dependencies on other RDDs**

RDD 的每次转换都会生成一个新的 RDD，所以 RDD 之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark 可以通过这个依赖关系重新计算丢失的分区数据，而不是对 RDD 的所有分区进行重新计算。

4. **Optionally, a Partitioner for Key-value RDDs**

对于 Key-Value 型 RDD，可以指定一个 `partition`，告诉它如何进行分片。也就是说，如果 RDD 里面存的数据是 Key-Value 形式，则可以传递一个自定义的 `Partitioner` 进行重新分区，例如这里自定义的 `Partitioner` 是基于 `key` 进行分区，那则会不同 RDD 里面的相同 `key` 的数据放到同一个 `partition` 里面。

5. **Optionally, a list of preferred locations to compute each split on**

最优的位置去计算，比如 HDFS 的 `block` 的所在位置应该是优先计算的位置。存储存取每个 `Partition` 的优先位置（`preferred location`）。对于一个 HDFS 文件来说，这个列表保存的就是每个 `Partition` 所在的块的位置。按照“移动数据不如移动计算”的理念，Spark 在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。

批注 [雷1]: 特性需要翻译成中文

批注 [雷2]: 翻译成中文

批注 [雷3]: 翻译成中文

批注 [雷4]: 翻译成中文

批注 [雷5]: 翻译成中文

## 7.2 RDD 工作原理

### 7.2.1 RDD 工作机制原理

Spark 的计算模式不同于 Hadoop 中的 MapReduce，RDD 的计算是分布在不同节点中的，数据是以数据块的形式存储在各个节点中，数据块也就是前面章节提到的 `Block`。

RDD 是由不同的分区组成的，所进行的转换和执行操作都是在每一块独立的分区上各自进行的。而在存储管理模块内部，RDD 又由不同的数据块组成，对于 RDD 的存取是以数据块为单位的，本质上分区（`partition`）和数据块（`Block`）是等价的。同时，在 Spark 存储管理模块中存取数据的最小单位是数据块，所有的操作都是以数据块为单位的。数据计算存储如下图 7-1 所示。

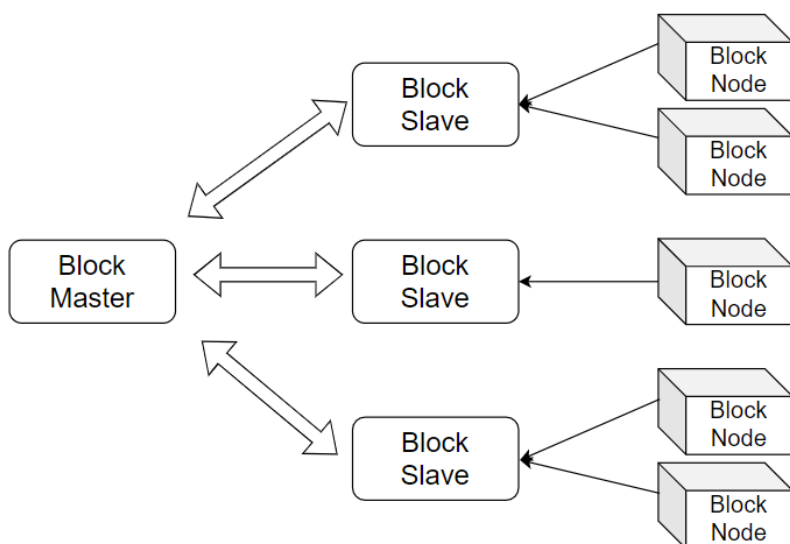


图 7-1 数据块存储方式

### 7.2.2 RDD 依赖关系

Transform 在生成 RDD 的时候，生成的是整个 RDD，这里的 RDD 的生成方式并不是一次性生成整个，而是由上一级的 RDD 依次往下生成，我们将其称为依赖。

RDD 生成的方式也不尽相同，在实际工作中，RDD 一般由两种方式生成：宽依赖 (wide dependency) 和窄依赖 (narrow dependency)。具体来说，窄依赖指的是每一个父 RDD 的 Partition 最多被子 RDD 的一个 Partition 使用，比如我们生活中的独生子女，如图 7-2 所示；宽依赖指的是多个子 RDD 的 Partition 会依赖同一个父 RDD 的 Partition，比如我们生活中的非独生子女，如图 7-3 所示。

Wide Dependencies:

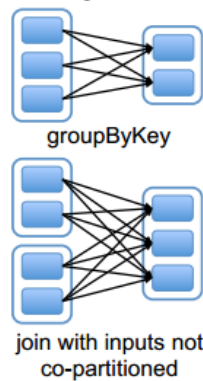


图 7-2 宽依赖

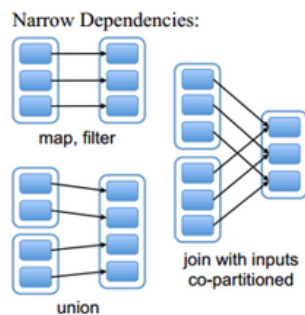


图 7-3 窄依赖

那么，我们需要一种机制来记录父 RDD 与子 RDD 之间的关系，这种关系就叫 Lineage（血统）。

把创建 RDD 的一系列 Lineage（即血统）记录下来，以便恢复丢失的分区。RDD 的 Lineage 会记录 RDD 的元数据信息和转换行为，当该 RDD 的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。

宽依赖和窄依赖在实际应用中有着不同的作用。窄依赖便于在单一节点上按次序执行任务，使任务可控。而宽依赖便于的是考虑任务的交互和容错性。这里没有好坏之分，具体选择哪种方式需要根据具体情况处理。

## 7.3 RDD API 应用

本节将进行 RDD API 的实战讲解，首先我们需要启动集群和 Pyspark Notebook，然后为本章节创建 Notebook 文件。

步骤 1：启动 Hadoop 集群，代码如下所示。

```
cd /usr/local/src/ hadoop-2.6.5/sbin
./start-all.sh
```

步骤 2: 启动 Pyspark Notebook, 代码如下所示。

```
PYSPARK_DRIVER_PYTHON=ipython
PYSPARK_DRIVER_PYTHON_OPTS="notebook" pyspark
```

步骤 3: 新建 Notebook 文件

在启动并打开 Jupyter 之后, 在页面右上角点击【New】下拉菜单, 再弹出的下拉菜单中单击选择【Python 3】, 即可创建 Notebook 文件, 详情如下图 7-4 所示

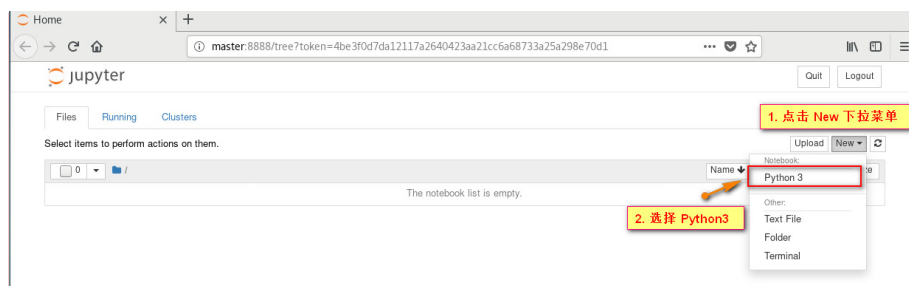


图 7-4 创建 Notebook 文件

步骤 4: 重命名 Notebook

接下来, 为 Notebook 文件进行重命名, 单击页面左上角的“Untitled”, 系统会自动弹出重命名对话框, 在弹出的对话框中输入要重新命名的名称, 如“Chapter07”, 最后单击【Rename】即可, 步骤详情如图 7-5 所示。



图 7-5 重命名 Notebook

### 7.3.1 RDD 的创建

我们使用最简单的创建 RDD 的方法, 即 SparkContext 的 parallelize 方法。

步骤 1: 创建 Int 类型 RDD, 代码如下所示。

```
int_data = [1,2,3,4,5,5]
int_rdd = sc.parallelize(int_data)
```

上面代码通过两个步骤创建了 Int 类型 RDD。首先定义了一个 List 变量 int\_data, 其中每个元素均为整数类型, 然后, 通过 parallelize 方法将 List 变量创建 RDD 变量。正如前面小节提到的, 由于 RDD 的特性, 执行完上述代码之后, 并不会马上出结果, 这里只

批注 [雷6]: 图上步骤文字需要在正文中表述, 只是图上标注是不行, 图上可以标重点

批注 [雷7]: 图上步骤文字需要在正文中表述, 只是图上标注是不行, 图上可以标重点

是相当于 Transform 方法，必须通过 Action 方法才能真正的执行。这里，我们使用最简单的 collect 方法来使代码执行，打印 RDD 变量内容，执行结果如下图 7-6 所示。

```
In [1]: int_data = [1,2,3,4,5,5]
In [2]: int_rdd = sc.parallelize(int_data)
In [3]: int_rdd.collect()
Out[3]: [1, 2, 3, 4, 5, 5]
```

图 7-6 查看 Int 类型 RDD

#### 步骤 2: 创建 String 类型 RDD

同样的方法，改变 List 中元素变量的类型，我们可以创建 String 类型 RDD，代码如下。

```
str_data = ["one", "two", "three", "four", "five", "five"]
str_rdd = sc.parallelize(str_data)
```

使用 collect 方法进行打印，如图 7-7 所示。

```
In [5]: str_rdd.collect()
Out[5]: ['one', 'two', 'three', 'four', 'five', 'five']
```

图 7-7 查看 String 类型 RDD

#### 步骤 3: 创建 Key-Value 类型 RDD

在 Python 中，有字典类型的变量，即 Key-Value 形式。同样，Spark 也可以创建 Key-Value 类型 RDD，创建并展示 Key-Value 类型 RDD 代码如下图 7-8 所示。

```
In [6]: kv_data = [('a', 1), ('b', 6), ('a', 3), ('c', 3)]
kv_rdd = sc.parallelize(kv_data)
In [7]: kv_rdd.collect()
Out[7]: [('a', 1), ('b', 6), ('a', 3), ('c', 3)]
```

图 7-8 创建 Key-Value 类型 RDD

#### 步骤 4: 混合类型 RDD

同样，当 List 中元素类型不一致时，也可以进行 RDD 的创建，代码如下图 7-9 所示。

```
In [8]: mutil_data = [1, "two", 3.3]
In [9]: mutil_rdd = sc.parallelize(mutil_data)
mutil_rdd.collect()
Out[9]: [1, 'two', 3.3]
```

图 7-9 创建混合类型 RDD

### 7.3.2 RDD 基本信息

上一小节我们介绍了不同类型 RDD 的创建，对于 RDD 变量，Spark 自带一些方法可以查看 RDD 的基本信息。这里展示的 RDD 基本信息包括系统信息和统计信息，其中统计信息包含了基本的统计计算值，如最大值、最小值、平均数、描述统计等，详情如下所示。

RDD 是否为空，使用 isEmpty()方法，如下图 7-10 所示。

```
In [10]: int_rdd.isEmpty()
Out[10]: False
```

图 7-10 判断 RDD 是否为空

获取 RDD 的分区数，使用 getNumPartition 方法，如下图 7-11 所示。

```
In [11]: int_rdd.getNumPartitions()
Out[11]: 2
```

图 7-11 获取 RDD 的分区数

RDD 数理数据，计算 RDD 最小值、最大值、标准差分别使用 min()、max()、stdev()方法，如下图 7-12 所示。

```
In [13]: int_rdd.min()
Out[13]: 1
```

1. 最小值

```
In [14]: int_rdd.max()
Out[14]: 5
```

2. 最大值

```
In [15]: int_rdd.stdev()
Out[15]: 1.4907119849998596
```

3. 标准差

图 7-12 查看 RDD 数理数据

RDD 元素个数，使用 count()方法，如下图 7-13 所示。

```
In [16]: int_rdd.count()
Out[16]: 6
```

图 7-13 查看 RDD 元素个数

RDD 统计信息，使用 stats()方法，如下图 7-14 所示。

```
In [12]: int_rdd.stats()
Out[12]: (count: 6, mean: 3.3333333333333335, stdev: 1.4907119849998596, max: 5.0, min: 1.0)
```

图 7-14 查看 RDD 统计信息

### 7.3.3 RDD Transform 基本运算

在 7.3.1 小节中，我们认识了最简单的 Transform 运算和 Action 运算，即 parallelize 方法和 collect 方法，本小节将对 RDD 的常见 Transform 运算进行详细介绍。

## 1. map 方法

map 方法的作用是返回一个新的 RDD，该 RDD 是由原 RDD 的每个元素经过函数转换后的值而组成，就是对 RDD 中的数据做转换。在 Spark 中，map 方法可以传入两种语句：具名函数和匿名函数。

下面分别编写具名函数和匿名函数传入 map 的方法，实现原始 RDD 中每个元素乘以 2 的操作。

➤ map 使用具名函数的方法，如下图 7-5 所示。

```
In [20]: def doubleFunc(x):  
         return x * 2  
  
         rdd_out = int_rdd.map(doubleFunc)  
         rdd_out.collect()
```

```
Out[20]: [2, 4, 6, 8, 10, 10]
```

图 7-15 map 使用具名函数

➤ map 使用匿名函数的方法，如下图 7-6 所示。

```
In [19]: rdd_out = int_rdd.map(lambda x : x * 2)  
         rdd_out.collect()
```

```
Out[19]: [2, 4, 6, 8, 10, 10]
```

图 7-16 map 使用匿名函数

可以看出，无论使用具名函数还是匿名函数都可以实现原始 RDD 中每个元素乘以 2 的操作。

当然，map 方法也可以用于其他类型 RDD 的转换操作，如 String 类型 RDD，示例如下图 7-17 所示。

```
In [21]: rdd_out = str_rdd.map(lambda x: "English num is :"+x)  
         rdd_out.collect()
```

```
Out[21]: ['English num is :one',  
          'English num is :two',  
          'English num is :three',  
          'English num is :four',  
          'English num is :five',  
          'English num is :five']
```

图 7-17 map 处理 String 类型 RDD

## 2. filter 方法

filter 方法用于实现 RDD 的过滤操作，返回一个新 RDD，内容是由满足过滤条件的元素组成。

接下来，我们使用 filter 方法实现不同条件的过滤操作。

➤ 过滤大于等于 2 的元素，详情如下图 7-18 所示。

```
In [22]: rdd_out = int_rdd.filter(lambda x : x >= 2)  
         rdd_out.collect()
```

```
Out[22]: [2, 3, 4, 5, 5]
```

图 7-18 大于等于 2 的元素

➤ 使用 and 条件过滤，详情如下图 7-19 所示。



```
In [23]: rdd_out = int_rdd.filter(lambda x : x >=1 and x <=4)
rdd_out.collect()

Out[23]: [1, 2, 3, 4]
```

图 7-19 and 条件过滤

➤ 使用 or 条件过滤，详情如下图 7-20 所示。

```
In [24]: rdd_out = int_rdd.filter(lambda x : x >4 or x<=2 )
rdd_out.collect()

Out[24]: [1, 2, 5, 5]
```

图 7-20 or 条件过滤

➤ 对字符串进行过滤，详情如下图 7-21 所示。

```
In [25]: rdd_out = str_rdd.filter(lambda x : x.endswith("e"))
rdd_out.collect()

Out[25]: ['one', 'three', 'five', 'five']
```

图 7-21 endswith 函数过滤字符串

### 3. distinct 方法

distinct 方法用于对 RDD 进行去重，删除 RDD 中重复的元素，这里我们分别对 int 类型 RDD 和 string 类型的 RDD 进行去重，具体代码如下图 7-22 所示。。

```
In [26]: rdd_out = int_rdd.distinct()
rdd_out.collect()

Out[26]: [2, 4, 1, 3, 5]

In [27]: rdd_out = str_rdd.distinct()
rdd_out.collect()

Out[27]: ['two', 'three', 'four', 'one', 'five']
```

Int类型RDD去重

String类型RDD去重

图 7-22 RDD 去重

### 4. groupBy 方法

groupBy 方法可以按照指定的函数方法规则对 RDD 中的元素进行分组运算。例如，我们可以将 Int 类型 RDD 中的元素分为奇数和偶数，详情如下图 7-22 所示。。

```
In [28]: grdd = int_rdd.groupBy(lambda x: "odd" if (x % 2 != 0) else "even")
grdd_out = grdd.collect()

In [29]: grdd_out[0][0], sorted(grdd_out[0][1])

Out[29]: ('even', [2, 4])

In [30]: grdd_out[1][0], sorted(grdd_out[1][1])

Out[30]: ('odd', [1, 3, 5, 5])
```

图 7-23 groupBy 拆分奇数和偶数

不能直接对分组后的变量进行打印，不会展示结果，如下图 7-24 所示。

批注 [雷8]: 图上步骤文字需要在正文中表述，只是图上标注是不行

批注 [雷9]: 图上步骤文字需要在正文中表述，只是图上标注是不行，图上可以标重点

```
In [15]: grdd_out[0][1]
Out[15]: <pyspark.resultiterable.ResultIterable at 0x7f5d717f31d0>
```

图 7-24 报错提示

#### 5. flatMap 方法

flatMap 方法用于将 RDD 内容扁平化后进行返回新的 RDD。类似于 map，但是每一个输入元素可以被映射为 0 或多个输出元素（所以 flatMap 传入的函数应该返回一个序列，而不是单一元素）如下图 7-25 所示。

```
In [32]: int_rdd.flatMap(lambda x : range(1,x)).collect()
Out[32]: [1, 1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4]
```

图 7-25 flatMap 扁平化处理

#### 6. repartition 方法

repartition 方法用于改变 RDD 的分区情况，根据新的分区数，重新 shuffle 打乱所有的数据，按新指定的分区个数进行分区，在图 7-11 中我们知道 int\_rdd 的分区数为 2，这里我们将它的分区数据设置为 3，具体代码如下图 7-26 所示。

```
In [33]: int_rdd.repartition(3).getNumPartitions()
Out[33]: 3
```

图 7-26 repartition 分区处理

#### 7. randomSplit 方法

randomSplit 方法以传入的随机数的方式对 RDD 中的数据进行按比例分割成两个新的 RDD，如下图 7-27 所示。

```
In [23]: rp1, rp2 = int_rdd.randomSplit([0.3, 0.7])
         print("rp1:",rp1.collect(), "rp2:",rp2.collect())
rp1: [3, 4] rp2: [1, 2, 5, 5]
```

图 7-27 randomSplit 切分 RDD

### 7.3.4 RDD Action 基本运算

本小节将对 RDD 的常见 Action 运算方法进行详细介绍。

#### 1. collect 方法

collect 方法是 RDD 最基本的 Action 方法之一，该方法可以以数组的形式返回数据集 RDD 的所有元素，在前面章节我们已经使用过该方法进行 RDD 的数据打印，这里我们再一次进行介绍，如下图 7-28 所示。

```
In [34]: int_rdd.collect()
Out[34]: [1, 2, 3, 4, 5, 5]
```

图 7-28 collect 运算

#### 2. reduce 方法

reduce 方法用于通过传入的函数聚集 RDD 中的所有元素，先聚合分区内数据，再聚合分区间数据，如下图 7-29 所示。

批注 [HZ10]: 这里是对 collect() 这个方法作用的介绍。

批注 [雷11]: 图片中的代码，与实现步骤，需在正文中描写出来，下面相同问题同样处理

批注 [HZ12R11]: 这一章节介绍的都是函数方法，可以看到都是一行代码完成的，然后正文第一句话都介绍了对应函数方法的作用、功能。

```
In [35]: rdd_out = int_rdd.reduce(lambda x,y : x+y)
```

```
In [36]: rdd_out
```

```
Out[36]: 20
```

图 7-29 reduce 运算

### 3. count 方法

count 方法可以实现统计 RDD 内元素的个数，如下图 7-30 所示。

```
In [37]: int_rdd.count()
```

```
Out[37]: 6
```

图 7-30 count 运算

### 4. first 方法

collect 方法可以获取 RDD 中所有元素，但有时候我们只需要查看 RDD 的第一个元素，这时我们可以使用 first 方法，如下图 7-31 所示。

```
In [24]: str_rdd.first()
```

```
Out[24]: 'one'
```

图 7-31 first 运算

### 5. take 方法

take 方法类似于 first 方法，take 方法可实现读取 RDD 内指定元素个数，如下图 7-32 所示。

```
In [39]: str_rdd.take(2)
```

```
Out[39]: ['one', 'two']
```

图 7-32 take 运算

### 6. top 方法

top 方法同样可以读取 RDD 内指定元素个数，但是 top 方法会对 RDD 的元素按降序进行排列后取指定个数元素，如下图 7-33 所示。

```
In [40]: int_rdd.top(3)
```

```
Out[40]: [5, 5, 4]
```

图 7-33 top 运算

### 7. takeOrdered 方法

takeOrdered 相当于 top 方法的升级版，可以按照指定的方法对 RDD 内元素进行排序后取出指定个数元素，该方法可以传入两个参数，第一个参数用于指定排序后输出的元素个数，第二个参数 key 用于指定排序的方法，参数 key 可以使用 lambda 表达式，如下图 7-34 所示。

```
In [41]: int_rdd.takeOrdered(3)
```

```
Out[41]: [1, 2, 3]
```

默认升序排列，参数3指输出元素个数

```
In [42]: int_rdd.takeOrdered(3, key=lambda x: -x)
```

```
Out[42]: [5, 5, 4]
```

参数key指排序的方法，这里进行降序排序

图 7-34 takeOrdered 运算

批注 [雷13]: 图上步骤文字需要在正文中表述，只是图上标注是不行，图上可以标重点

8. takeSample 方法

takeSample 方法将返回原 RDD 的固定大小的采样子集，如下图 7-35 所示。

```
In [43]: int_rdd.takeSample(True, 2, 3)
Out[43]: [2, 5]
```

图 7-35 takeSample 运算

其中 takeSample 有三个参数，第一个参数为：withReplacement，代表是否有回放；第二个参数为：num，代表采样个数；第三个参数 seed，代表采用的随机种子。

9. foreach 方法

foreach 方法对所有元素应用所传入的函数，即针对 RDD 中的每个元素都执行一次传入的函数。需要注意的是，每个函数是在 Executor 上执行的，不是在 driver 端执行的，所以结果也不会 Notebook 上直接打印出来，打印结果需要在终端日志中查看，如下图 7-36 所示。

```
In [44]: def f(x):
         print(x)
         int_rdd.foreach(f)

[I 20:35:41.434 NotebookApp] Adapting to protocol v5.1 for kernel 43fd7f05-f5c0-4e76-a3e2-24882147650c
1
2
3
4
5
6
```

打印结果要在终端日志中查看

图 7-36 foreach 运算

10. saveAsTextFile 方法

saveAsTextFile 用于将 RDD 进行保存，可以选择保存本地或者 HDFS，如下图 7-37 所示。

```
In [46]: !pwd
/root/pyspark-book

In [47]: int_rdd.saveAsTextFile('file:///root/pyspark-book/rdd.txt')
```

	Name	Last Modified	File size
	..	几秒钟前	
	._SUCCESS	4 分钟前	0 B
	part-00000	4 分钟前	6 B
	part-00001	4 分钟前	6 B

图 7-37 saveAsTextFile 保存运算

11. union 方法

接下来的运算方法均作用于多个 RDD 的操作。union 运算用于将两个 RDD 进行合并处理，返回一个新的 RDD，如下图 7-38 所示。

```

In [48]: int_rdd1 = sc.parallelize([3, 1, 2, 5, 5])
         int_rdd2 = sc.parallelize([5, 6])
         int_rdd3 = sc.parallelize([2, 7])

In [49]: int_rdd1.collect()
Out[49]: [3, 1, 2, 5, 5]

In [50]: int_rdd2.collect()
Out[50]: [5, 6]

In [51]: int_rdd3.collect()
Out[51]: [2, 7]

In [52]: int_rdd1.union(int_rdd2).union(int_rdd3).collect()
Out[52]: [3, 1, 2, 5, 5, 5, 6, 2, 7]

```

图 7-38 union 运算

## 12. intersection 方法

**intersection** 方法用于对两个 RDD 求交集后返回一个新的 RDD，如下图 7-39 所示。

```

In [53]: int_rdd1.intersection(int_rdd2).collect()
Out[53]: [5]

```

图 7-39 intersection 交集运算

## 13. subtract 方法

**subtract** 方法用于对两个 RDD 求差集后返回一个新的 RDD，也就是去除两个 RDD 中的共同的部分，如下图 7-40 所示。

```

In [54]: int_rdd1.subtract(int_rdd2).collect()
Out[54]: [1, 2, 3]

```

图 7-40 subtract 差集运算

## 14. cartesian 方法

**cartesian** 方法用于计算 2 个 RDD 的笛卡尔积，尽量避免使用，笛卡尔积的运算结果的数量会随 RDD 中元素个数的量而剧增，如下图 7-41 所示。

```

In [55]: int_rdd1.cartesian(int_rdd2).collect()
Out[55]: [(3, 5),
          (1, 5),
          (3, 6),
          (1, 6),
          (2, 5),
          (5, 5),
          (5, 5),
          (2, 6),
          (5, 6),
          (5, 6)]

```

图 7-41 cartesian 笛卡尔积运算

### 7.3.5 RDD Key-Value Transform 基本运算

本小节将对 Key-Value 类型 RDD 的常见 Transform 运算方法进行详细介绍。

#### 1. keys 方法

keys 方法可以获取 RDD 中所有元素的 key 值，如下图 7-42 所示。

```
In [56]: kv_rdd.keys().collect()
Out[56]: ['a', 'b', 'a', 'c']
```

图 7-42 keys 运算

#### 2. values 方法

同样的，values 方法与 keys 方法相当，可以获取 RDD 中所有元素的值，如下图 7-43 所示。

```
In [57]: kv_rdd.values().collect()
Out[57]: [1, 6, 3, 3]
```

图 7-43 values 运算

#### 3. filter 方法

filter 方法可以实现对 RDD 按照规则、方法对 Key-Value 类型 RDD 进行过滤操作，可以按照 key 进行过滤，也可以按照 value 进行过滤，支持 Python 中的 lambda 表达式，如下图 7-44 所示。

```
In [58]: kv_rdd.filter(lambda kv: kv[0] != 'a').collect()
Out[58]: [('b', 6), ('c', 3)]
```

图 7-44 针对 RDD 元素的 key 进行过滤

如上图代码所示，lambda 表达式中变量 kv 即为 RDD 中的 Key-Value 元素，kv[0]即为 Key 值，同理，kv[1]代表 Value 值。

```
In [59]: kv_rdd.filter(lambda kv: kv[1] > 2).collect()
Out[59]: [('b', 6), ('a', 3), ('c', 3)]
```

图 7-45 针对 RDD 元素的 value 进行过滤

#### 4. mapValues 方法

mapValues 方法针对 RDD 中每个 Key-Value 元素的 value 进行操作，如下图 7-46 所示。

```
In [60]: def f(x):
          return 2 * x
          kv_rdd.mapValues(f).collect()
Out[60]: [('a', 2), ('b', 12), ('a', 6), ('c', 6)]
```

图 7-46 mapValues 运算

#### 5. sortByKey 方法

sortByKey 方法对 RDD 中每个 Key-Value 元素的 key 进行排序，默认是升序排序，可以通过参数 ascending 进行控制，其值为 False 或者 True，如下图 7-47 所示。

```
In [61]: kv_rdd.sortByKey().collect()
Out[61]: [('a', 1), ('a', 3), ('b', 6), ('c', 3)]

In [62]: kv_rdd.sortByKey(ascending=False).collect()
Out[62]: [('c', 3), ('b', 6), ('a', 1), ('a', 3)]
```

默认升序排序

降序排序

批注 [雷14]: 图片内容需在正文中表达出来，下面同样问题相同处理

图 7-47 sortByKey 运算

#### 6. sortBy 方法

sortBy 方法对 RDD 中每个 Key-Value 元素按照指定的函数方法进行排序，同样，可以通过参数 ascending 进行控制排序方法为升序还是降序，如下图 7-48 所示。

```
In [63]: kv_rdd.sortBy(lambda x: x[1], ascending=False).collect()
Out[63]: [('b', 6), ('a', 3), ('c', 3), ('a', 1)]
```

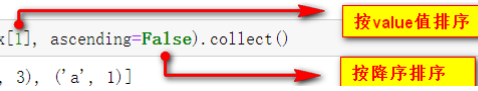


图 7-48 sortBy 运算

#### 7. reduceByKey 方法

reduceByKey 方法根据 key 对 value 进行合并聚合操作，例如对 RDD 中元素按相同的 key 进行相加求和运算，如下图 7-49 所示。

```
In [64]: kv_rdd.reduceByKey(lambda x, y: x+y).collect()
Out[64]: [('b', 6), ('c', 3), ('a', 4)]
```

图 7-49 reduceByKey 运算

#### 8. flatMapValues 方法

flatMapValues 方法和 flatMap 方法类似，是针对 Key-Value 型元素 RDD 的操作，如下图 7-50 所示。

```
In [65]: kv_rdd2 = sc.parallelize([("a", ["x", "y", "z"]), ("b", ["p", "r"])]
kv_rdd2.flatMapValues(lambda x: x).collect()
Out[65]: [('a', 'x'), ('a', 'y'), ('a', 'z'), ('b', 'p'), ('b', 'r')]
```

图 7-50 flatMapValues 运算

#### 9. join 方法

下面的 Transform 方法为针对多个 Key-Value 型 RDD 的操作。join 方法可以按照相同的 key 值连接两个 RDD，如下图 7-51 所示。

```
In [66]: kv_rdd3 = sc.parallelize([('a', 1), ('c', 6)])
kv_rdd3.collect()
Out[66]: [('a', 1), ('c', 6)]

In [67]: kv_rdd.join(kv_rdd3).collect()
Out[67]: [('c', (3, 6)), ('a', (1, 1)), ('a', (3, 1))]
```

图 7-51 join 操作

上面代码 join 过程图解如下。

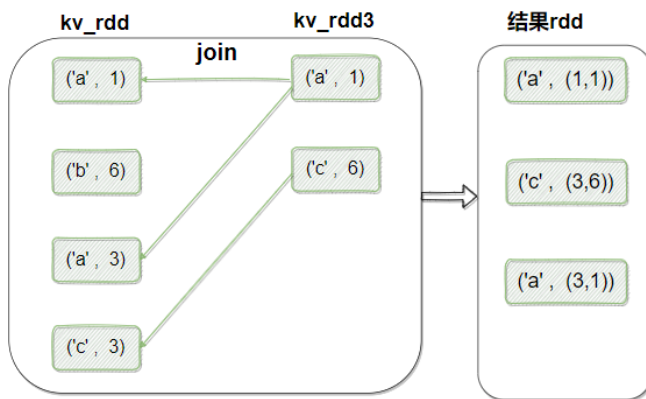


图 7-52 join 过程

从上图 7-52 可以看出 join 的过程，即把 `kv_rdd` 和 `kv_rdd3` 中相同 key 值的元素的 value 值合并到一起，从图中可以看出，`kv_rdd` 和 `kv_rdd3` 两个 RDD 中相同的 key 值分别为 'a' 和 'c'。

批注 [雷15]: 语意不明, 需审核修改

#### 10. leftOuterJoin 方法

`leftOuterJoin` 和 `rightOuterJoin` 均为 join 的一种方式，即以左边的 RDD 中元素的 key 为准，即当右边的 RDD 中没有与左边 RDD 进行连接的 key 时，这些左边的 RDD 中的 key 对应的 value 会以 None 进行补充，如下图 7-53 所示。

```
In [68]: kv_rdd.leftOuterJoin(kv_rdd3).collect()
Out[68]: [('b', (6, None)), ('c', (3, 6)), ('a', (1, 1)), ('a', (3, 1))]
```

图 7-53 leftOuterJoin 操作

`leftOuterJoin` 图解如下图 7-54 所示。

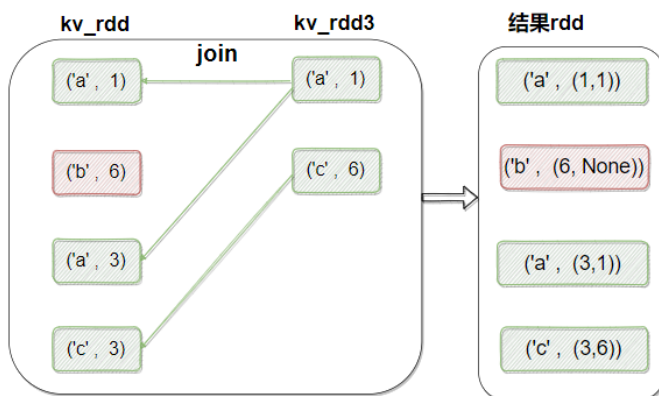


图 7-54 leftOuterJoin 过程

#### 11. rightOuterJoin 方法



rightOuterJoin 与 leftOuterJoin 原理相同，区别在于以右边的 RDD 中元素的 key 为准，如下图 7-55 所示。

```
In [69]: kv_rdd.rightOuterJoin(kv_rdd3).collect()
Out[69]: [('c', (3, 6)), ('a', (1, 1)), ('a', (3, 1))]
```

图 7-55 rightOuterJoin 操作

rightOuterJoin 图解如下图 7-56 所示。

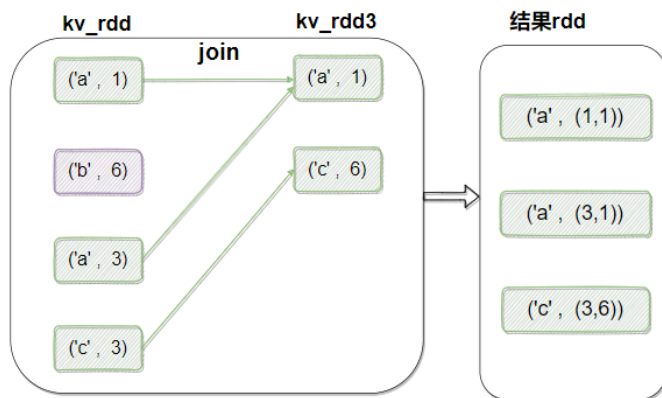


图 7-56 rightOuterJoin 过程

由上图可以看出，该结果与 join 结果相同。这是因为 kv\_rdd3 中的 key 在 kv\_rdd 中完全存在，若 kv\_rdd3 中存在有 kv\_rdd 没有的 key 时，则结果中存在形如 leftOuterJoin 的结果，即对匹配不到的 key 以 None 进行补充。

#### 12. intersection 方法

intersection 方法同样可以求两个 Key-Value 类型 RDD 的交集，如下图 7-57 所示。

```
In [70]: kv_rdd.intersection(kv_rdd3).collect()
Out[70]: [('a', 1)]
```

图 7-57 intersection 交集运算

#### 13. union 方法

union 可以求两个 Key-Value 类型 RDD 的并集(包含重复元素)，如下图 7-58 所示。

```
In [71]: kv_rdd.union(kv_rdd3).collect()
Out[71]: [('a', 1), ('b', 6), ('a', 3), ('c', 3), ('a', 1), ('c', 6)]
```

图 7-58 union 交集运算

#### 14. subtract 方法

subtract 方法求两个 RDD 的差集，例如 RDD1 对 RDD2 的差集，如下图 7-59 所示。

```
In [72]: kv_rdd.subtract(kv_rdd3).collect()
Out[72]: [('b', 6), ('a', 3), ('c', 3)]
```

图 7-59 subtract 差集运算

#### 15. subtractByKey 方法

subtractByKey 方法求两个 RDD 根据 key 求差集，如下图 7-60 所示。

```
In [73]: kv_rdd.subtractByKey(kv_rdd3).collect()
Out[73]: [('b', 6)]
```

图 7-60 subtractByKey 差集运算

#### 16. cartesian 方法

cartesian 方法计算两个 RDD 笛卡尔积，对于笛卡尔积，我们可以举一个通俗的例子，如果 A 表示某学校学生的集合，B 表示该学校所有课程的集合，则 A 与 B 的笛卡尔积表示所有可能的选课情况。A 表示所有声母的集合，B 表示所有韵母的集合，那么 A 和 B 的笛卡尔积就为所有可能的汉字全拼。笛卡尔积运行结果如下图 7-61 所示。

```
In [74]: kv_rdd.cartesian(kv_rdd3).collect()
Out[74]: [('a', 1), ('a', 1)],
          [('b', 6), ('a', 1)],
          [('a', 1), ('c', 6)],
          [('b', 6), ('c', 6)],
          [('a', 3), ('a', 1)],
          [('c', 3), ('a', 1)],
          [('a', 3), ('c', 6)],
          [('c', 3), ('c', 6)]
```

图 7-61 cartesian 笛卡尔积运算

### 7.3.6 RDD Key-Value Action 基本运算

同样的，Key-Value 类型 RDD 也有 Action 运算操作，部分函数方法和前面小节介绍的单一类型 RDD 的方法相同，详情如下所示。

#### 1. reduceByKey 方法

reduceByKey 方法它会把所有 key 相同的 value 值处理并且进行归并，其中归并的方法可以自己定义，如下图 7-62 所示。

```
In [26]: kv_rdd.reduceByKey(lambda x,y:x+y).collect()
Out[26]: [('b', 6), ('c', 3), ('a', 4)]
```

图 7-62 reduceByKey 运算

#### 2. first 方法

first 方法作用和 7.3.4 小节中 RDD Action 中的 first 方法作用完全相同。

```
In [76]: kv_rdd.first()
Out[76]: ('a', 1)
```

图 7-63 first 运算

#### 3. take 方法

take 方法作用和 7.3.4 小节中 RDD Action 中的 take 方法作用完全相同。

```
In [77]: kv_rdd.take(2)
Out[77]: [('a', 1), ('b', 6)]
```

图 7-64 take 运算

#### 4. countByKey 方法

countByKey 用于按 key 统计 RDD 里的元素个数。注意，需使用 items 方法返回一个字典类型结果，

批注 [雷16]: 方法详情介绍一下

如下图 7-65 所示。

```
In [78]: kv_rdd.countByKey().items()
Out[78]: dict_items([('a', 2), ('b', 1), ('c', 1)])
```

图 7-65 countByKey 运算

#### 5. countByValue 方法

countByValue 用于将 RDD 中每个唯一值的计数作为(值、计数)对的字典返回，如下图 7-66 所示。

```
In [79]: kv_rdd.countByValue().items()
Out[79]: dict_items([(('a', 1), 1), (('b', 6), 1), (('a', 3), 1), (('c', 3), 1)])
```

图 7-66 countByValue 运算

#### 6. groupByKey 方法

groupByKey 方法用于对 Key-Value 类型 RDD 进行分组操作，如下图 7-67 所示。

```
In [80]: kv_rdd.collectAsMap()
Out[80]: {'a': 3, 'b': 6, 'c': 3}
```

图 7-67 groupByKey 运算

温馨提示：

reduceByKey 与 groupByKey 区别：

- (1) reduceByKey: 按照 key 进行聚合，在 shuffle 之前有 combine（预聚合）操作，返回结果是 Key-Value 类型的 RDD。
- (2) groupByKey: 按照 key 进行分组，直接进行 shuffle。
- (3) reduceByKey 比 groupByKey 性能更好，建议使用，但是需要注意是否会影响业务逻辑。

#### 7. lookup 方法

使用 lookup 方法可以通过 key 来查找对应的 value 值，如下图 7-68 所示。

```
In [81]: kv_rdd.lookup('a')
Out[81]: [1, 3]
```

图 7-68 lookup 运算

### 7.3.7 共享变量

下面，我们将对 Spark 中共享变量进行介绍。合理使用共享变量可以节省内存和运行时间。正常情况下，传递给 Spark 算子(比如: map, reduce 等)的函数都是在远程的集群节点上执行，函数中用到的所有变量都是独立拷贝的。这些变量被拷贝到集群上的每个节点上，这些变量的更改不会传递回 Driver 端，跨 task 之间共享变量通常是低效的，所以，Spark 对共享变量也提供了两种支持，分别是 Broadcast 广播变量和 accumulator 累加器。

#### 1. Broadcast 广播变量

广播变量在每个节点上保存一个只读的变量的缓存，而不用给每个 task 来传送一个 copy。例如，给每个节点一个比较大的输入数据集是一个比较高效的方法。Spark 也会用该对象的广播逻辑去分发广播变量来降低通讯的成本。

下面我们通过简单的例子来进行介绍，我们创建 Key-Value 型 RDD，key 为编号，value 为动物名称，我们将编号与动物名称进行对应起来。

#### ➤ 不使用广播变量

首先我们创建一个 animal\_id 的整型 RDD，代表动物名称的编号 id；然后再创建一个 Key\_value 类型的 RDD，其中每个元素的 Key 代表动物名称的编号 id，Value 代表动物名称。代码如图 7-69 所示，这里使用了 collectAsMap() 函数，collectAsMap 函数类似于 collect 函数，它针对 Key-Value 格式的 RDD 进行操作，将 RDD 回收到 Driver 端形成一个 Map，返回所有元素集合，不过该集合是去掉的重复的 key 的集合，如果元素重复集合中保留的元素是位置最后的一组。



批注 [雷17]: 图片内步骤需写在正文中

图 7-69 未使用广播变量示例

#### ➤ 使用广播变量

首先，我们将创建的 animal\_map 通过 broadcast() 方法创建广播变量 bst\_map；然后我们可以通过对 animal\_id 中每个动物名称的编号 id 进行映射，将 id 编号映射为动物名称，代码如下图 7-70 所示。



图 7-70 广播变量示例

## 2. accumulator 累加器

累加器用来对信息进行聚合，通常在向 Spark 传递函数时，比如使用 map 函数传条件时，可以使用 Driver 程序中定义的变量，但是集群中运行的每个任务都会得到这些变量的一份新的副本，所以更新这些副本的值不会影响驱动器中的对应变量。

下面我们通过简单的例子介绍累加器，通过 foreach 函数实现对累加器的求和，需要注意的是，foreach 循环是在各个节点运行的，在 task 中不能读取累加器的值，需要使用 .value 来获取累加器的值，如下图 7-71 所示。。

```

In [86]: accum = sc.accumulator(0)
         acc_rdd = sc.parallelize([1,2,3,4])

In [87]: def g(x):
         accum.add(x)

In [88]: acc_rdd.foreach(g)
         final = accum.value
         print("final=", final)

final= 10

```




图 7-71 累加器示例

### 7.3.8 RDD 持久化

由于每个 Action 就会产生一个 job，每个 job 开始计算的时候总是从这个 job 最开始的 RDD 开始计算。所以，为了提升运算效率，Spark 通过 RDD 持久化机制，将需要重复运算的 RDD 存储在内存中。

我们可以使用方法 `persist` 或者 `cache` 来持久化一个 RDD。第一个 Action 会计算这个 RDD，然后把结果存储到他的节点的内存中。Spark 的 Cache 也是容错，如果 RDD 的任何一个分区的数据丢失了，Spark 会自动的重新计算，又因为 RDD 的各个 Partition 是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部 Partition。

另外，允许我们对持久化的 RDD 使用不同的存储级别，如下表 7-1 所示。

表 7-1 RDD 持久化存储级别

存储级别	含义
MEMORY_ONLY	默认选项，存储 RDD 的方式是以 Java 反串行化在 JVM 内存中，当 RDD 太大时，则无法完全存储在内存中，多余的 RDD partitions 在需要时会被重新计算。
MEMORY_AND_DISK	存储 RDD 的方式是以 Java 反串行化在 JVM 内存中，当 RDD 太大时，则无法完全存储在内存中，多余的 RDD partitions 会存储在硬盘中，需要时会从硬盘中读取。
MEMORY_ONLY_SER (Java and Scala)	与 MEMORY_ONLY 类似，存储 RDD 的形式为 Java 对象串行化，需要进行反串行化才可以使用。
MEMORY_AND_DISK_SER (Java and Scala)	原理与 MEMORY_AND_DISK_SER 类似。
DISK_ONLY	存储 RDD 在硬盘上。
MEMORY_ONLY_2, MEMORY_AND_DISK_2	与上列相对应的存储选项一样，但是每一个 RDD partitions 都需要复制到两个节点上。

下面通过使用 `persist` 方法实现对 RDD 变量的持久化，如下图 7-72 所示。

```

In [89]: int_rdd.persist()
int_rdd.is_cached
Out[89]: True

In [90]: int_rdd.unpersist()
int_rdd.is_cached
Out[90]: False

In [91]: from pyspark import StorageLevel
int_rdd.persist(StorageLevel.MEMORY_AND_DISK)
int_rdd.is_cached
Out[91]: True

```

图 7-72 RDD 持久化示例

### 7.3.9 WordCount 代码详解

最后，本章详细介绍了 RDD 的 Transform 算子和 Action 算子的作用和使用方法。接下来，我们将对上一章节中基于 Pyspark 的 WordCount 代码部分进行详细讲解和分析。

步骤 1：上传数据文件，代码如下所示。

```

# HDFS 路径
hdfs dfs -mkdir /data
hdfs dfs -put The_DaVinci_Code.txt /data/

```

步骤 2：读取文件

这里使用 `sc.textFile` 方法读取 HDFS 上的数据文件，代码如下所示。

```

word_data = sc.textFile("/data/The_DaVinci_Code.txt")
word_data.take(3)

```

使用 `take` 方法查看 RDD 数据中的 3 个元素，如下图 7-73 所示。

```

Out[97]: ['TheDaVinciCode - FOR BLYTHE... AGAIN. MORE THAN EVER',
          'FOR BLYTHE... AGAIN. MORE THAN EVER.']

```

图 7-73 读取并查看数据

步骤 3：取出每一个单词

我们使用 `flatMap` 方法取出每一个单词，对每行内容按空格进行分割，同样使用 `take` 方法进行查看，代码如下所示。

```

word_rdd = word_data.flatMap(lambda line : line.split(" "))
word_rdd.take(5)

```

结果如图 7-74 所示。

```

Out[100]: ['TheDaVinciCode', '-', 'FOR', 'BLYTHE...', 'AGAIN.']

```

图 7-74 取出每一个单词数据

步骤 4：映射每个单词

使用 `map` 方法将每个单词的次数赋值为 1，用于后续的统计工作，代码如下所示。

```

word_cnt = word_rdd.map(lambda word : (word, 1))
word_cnt.take(5)

```

结果如图 7-75 所示。

```
Out[101]: [('TheDaVinciCode', 1), ('-', 1), ('FOR', 1), ('BLYTHE...', 1), ('AGAIN.', 1)]
```

图 7-75 map 映射转换单词

步骤 5: 统计每个单词出现次数

使用 `reduceByKey` 的方法按单词进行统计, 代码如下所示。

```
word_count = word_cnt.reduceByKey(lambda x,y : x+y)
word_count.take(5)
```

结果如图 7-76 所示。

```
Out[102]: [('TheDaVinciCode', 109),
            ('FOR', 4),
            ('BLYTHE...', 2),
            ('AGAIN.', 2),
            ('MORE', 2)]
```

图 7-76 reduceByKey 统计单词个数

步骤 6: 保存结果数据

最后, 我们使用 `saveAsTextFile` 方法将计算结果保存到 HDFS 上, 代码如下所示。

```
word_count.saveAsTextFile("/data/wordcount_output")
```

我们可以使用 HDFS 命令查看我们存储的数据目录和结果, 如图 7-77 所示。

```
In [110]: !hdfs dfs -ls /data/wordcount_output
```

```
Found 3 items
-rw-r--r--  2 root supergroup          0 2022-02-05 20:52 /data/wordcount_output/_SUCCESS
-rw-r--r--  2 root supergroup 169265 2022-02-05 20:52 /data/wordcount_output/part-00000
-rw-r--r--  2 root supergroup 167253 2022-02-05 20:52 /data/wordcount_output/part-00001
```

```
In [113]: !hdfs dfs -cat /data/wordcount_output/part-00000
```

```
('TheDaVinciCode', 109)
('FOR', 4)
('BLYTHE...', 2)
('AGAIN.', 2)
('MORE', 2)
('THAN', 2)
('EVER', 1)
```

图 7-77 查看存储结果

## ★回顾思考★

### 01 RDD 有哪些特点？

答：包括如下：

- （1）存储的弹性：内存与磁盘的自动切换；
- （2）容错的弹性：数据丢失可以自动恢复；
- （3）计算的弹性：计算出错重试机制；
- （4）分片的弹性：可根据需要重新分片。

### 02 RDD 依赖关系有那两种？

答：包括如下：

宽依赖和窄依赖。

### 03 reduceByKey 和 groupByKey 的区别？

答：reduceByKey 与 groupByKey 区别：

- （1）reduceByKey：按照 key 进行聚合，在 shuffle 之前有 combine（预聚合）操作，返回结果是 Key-Value 类型 RDD；
- （2）groupByKey：按照 key 进行分组，直接进行 shuffle；
- （3）reduceByKey 比 groupByKey 性能更好。

## ★练习★

### 一、选择题

1. 以下哪个不是 RDD 的特点（ ）
  - A. 存储的弹性
  - B. 类型的弹性
  - C. 容错的弹性
  - D. 计算的弹性
2. 以下哪项是 RDD 的依赖关系（ ）
  - A. 窄依赖
  - B. 宽依赖
  - C. 宽依赖和窄依赖
  - D. 子依赖和父依赖
3. 能够实现对 Int 型 RDD 过滤偶数的代码是（ ）
  - A. `rdd.filter(lambda x : x == 2)`
  - B. `rdd.filter(lambda x : x %2 == 0)`
  - C. `rdd.filter(lambda x : x %2 != 0)`



D. `rdd.filter(lambda x : x != 0)`

## 二、填空题

1. `sortByKey` 方法可以实现 Key-Value 型 RDD 按 value 排序

(×)

2. `MEMORY_ONLY` 是 RDD 持久化默认的方式。

(√)

## 三、实战练习

**任务 1：**完整实现 RDD 的 Transform 和 Action 操作代码实战。

请参考本书配套第 7 章 Notebook 代码文件

**任务 2：**完整实现 RDD 持久化和共享变量的代码实战。

请参考本书配套第 7 章 Notebook 代码文件

**任务 3：**尝试使用 `zip` 方法压缩 RDD

示例：

```
rdd1 = sc.parallelize([1,2,3,4,5,6])
rdd2 = sc.parallelize(['a', 'b', 'c', 'd', 'e', 'f'])
rdd1.zip(rdd2)
```

**任务 3：**再次实现 Pyspark WordCount 代码实战，并满足以下要求：

(1) 过滤逗号、冒号等标点符号；

提示：`rdd.filter(',').filter(':')`

(2) 存储计算结果时改变 RDD 区分数，并进行对比。

提示：`rdd.repartition(2)`

完整代码请参考本书配套第 7 章 Notebook 代码文件

批注 [雷18]: 需要答案补充上

## 本章小结

本章对 Spark Core 进行了详细介绍，首先对 RDD 的基本概念和原理进行了简单介绍。RDD 是整个 Spark 框架的核心和基础。

从 7.3 节开始，本章详细展开了 RDD 的代码实战，对 RDD 的 Transform、Action 以及持久化最基本、最常用的 API 方法进行了详细介绍，限于篇幅，本章只对常用的函数方法进行了介绍，读者可以通过 Spark 官网了解更多 API 方法。

本章前两节是关于 RDD 内容介绍，可能有些晦涩不易理解，读者可以通过代码实战以及后续章节的学习之后再对 RDD 原理进行阅读，相信读者们会有更深一步的理解。