第3章

Spark RDD 编程

RDD 是 Spark 的核心概念。Spark 基于 Python 语言提供了对 RDD 的转换操作和行动操作，通过这些操作可实现复杂的应用。本章主要介绍 RDD 创建的方式、RDD 转换操作、RDD 行动操作、RDD 之间的依赖关系和 RDD 的持久化，最后给出案例实战——利用 Spark RDD 实现词频统计。

3.1 RDD 的创建方式


RDD 的创建方式

传统的 MapReduce 虽然具有自动容错、平衡负载和可拓展性的优点，但是其最大的缺点是在迭代计算式的时候要进行大量的磁盘 I/O 操作，而 RDD 正是为解决这一缺点而出现的。

Spark 数据处理引擎 Spark Core 是建立在统一的抽象弹性分布式数据集(RDD)之上的，这使得 Spark 的 Spark Streaming、Spark SQL、Spark MLlib、Spark GraphX 等应用组件可以无缝地进行集成，能够在同一个应用程序中完成大数据处理。RDD 是 Spark 对具体数据对象的一种抽象(封装)，本质上是一个只读的分区(partition)记录集合，每个分区就是一个数据集片段，每个分区对应一个任务。一个 RDD 的不同分区可以保存到集群中的不同节点上，对 RDD 进行操作，相当于对 RDD 的每个分区进行操作。RDD 中的数据对象可以是 Python、Java、Scala 中任意类型的对象，甚至是用户自定义的对象。Spark 中的所有操作都是基于 RDD 进行的，一个 Spark 应用可以看作一个由 RDD 的创建到一系列 RDD 转化操作再到 RDD 存储的过程。图 3-1 展示了 RDD 的分区及分区与工作节点(worker node)的分布关系，其中的 RDD 被切分成 4 个分区。

RDD 最重要的特性是容错性。如果 RDD 某个节点上的分区因为节点故障导致数据丢了，那么 RDD 会自动通过自己的数据来源重新计算得到该分区，这一切对用户是透明的。

创建 RDD 有两种方式：通过 Spark 应用程序中的数据集创建；使用本地及 HDFS、HBase 等外部存储系统上的文件创建。

下面讲解创建 RDD 的常用方式。

3.1.1 使用程序中的数据集创建 RDD

可通过调用 SparkContext 对象的 parallelize()方法并行化程序中的数据集

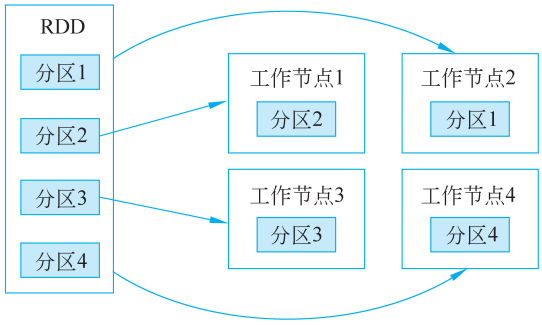

图 3-1 RDD 的分区及分区与工作节点的分布关系

合以创建 RDD。可以序列化 Python 对象得到 RDD。例如:

```
>>> arr = [1, 2, 3, 4, 5, 6]
>>> rdd = sc.parallelize(arr)          #把 arr 这个数据集并行化到节点上以创建 RDD
>>> rdd1 = sc.parallelize([('a', 7), ('a', 2), ('b', 2)])
>>> rdd2 = sc.parallelize(range(100))
>>> rdd3 = sc.parallelize([('a', [1, 2, 3]), ('b', [4, 5, 6])])
>>> rdd.collect()                      #以列表形式返回 RDD 中的所有元素
[1, 2, 3, 4, 5, 6]
>>> rdd3.collect()
[('a', [1, 2, 3]), ('b', [4, 5, 6])]
```

在上述语句中,使用了 Spark 提供的 SparkContext 对象,名称为 sc,这是 PySpark 启动的时候自动创建的,在交互式编程环境中可以直接使用。如果编写脚本程序文件,则在程序文件中通过如下语句创建 sc:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf( ).setAppName("Spark Demo").setMaster("local")
sc = SparkContext(conf = conf)
```

任何 Spark 程序都是从 SparkContext 开始的,SparkContext 的初始化需要一个 SparkConf 对象,SparkConf 包含了 Spark 集群配置的各种参数。创建 SparkContext 对象后,就可以使用 SparkContext 对象所包含的各种方法创建和操作 RDD。

实际上,RDD 也是一个数据集合。与 Python 的 list(列表)对象不同的是,RDD 的数据可能分布于多台计算机上。

在调用 parallelize()方法时,可以设置一个参数指定将一个数据集合切分成多少个分区,例如,parallelize(arr, 3)指定 RDD 的分区数是 3。Spark 会为每一个分区运行一个任务,对其进行处理。Spark 默认会根据集群的情况设置分区的数量。当调用 parallelize()方法时,若不指定分区数,则使用系统给出的分区数。例如:

```
>>> rdd4 = sc.parallelize([1, 2, 3, 4, 5, 6], 3)
```

```
>>> rdd4.getNumPartitions()                    #获取 rdd4 的分区数
3
```

RDD 对象的 glom()方法分别将 RDD 对象的每个分区上的元素分别放入一个列表中，返回一个由这些列表组成的新 RDD。例如：

```
>>> rdd4.glom().collect()
[[1, 2], [3, 4], [5, 6]]
```

3.1.2　使用文本文件创建 RDD

Spark 可以使用任何 Hadoop 支持的存储系统上的文件（如 HDFS、HBase 以及本地文件）创建 RDD。调用 SparkContext 对象的 textFile()方法读取文件的位置，即可创建 RDD。textFile()方法支持针对目录、文本文件、压缩文件以及通配符匹配的文件进行 RDD 的创建。

Spark 支持的常见文件格式如表 3-1 所示。

表 3-1　Spark 支持的常见文件格式

文 件 格 式	数 据 类 型	描　　述
文本文件	非结构化	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
CSV	结构化	常见的基于文本的格式，通常应用在电子表格中
SequenceFile	结构化	用于键值对数据的常见 Hadoop 文件格式
对象文件	结构化	用来存储 Spark 作业中的数据，给共享的代码读取

1. 读取 HDFS 中的文本文件创建 RDD

在 HDFS 中有一个文件名为/user/hadoop/input/data.txt，其内容如下：

```
Business before pleasure.
Nothing is impossible to a willing heart.
I feel strongly that I can make it.
```

在读取该文件创建 RDD 之前，需要先启动 Hadoop 系统，命令如下：

```
$ cd /usr/local/hadoop
$ ./sbin/start-dfs.sh                          #启动 Hadoop
#读取 HDFS 上的文件创建 RDD
>>> rdd = sc.textFile("/user/hadoop/input/data.txt")
>>> rdd.foreach(print)                         #输出 rdd 中的每个元素
Business before pleasure.
```

```
Nothing is impossible to a willing heart.
I feel strongly that I can make it.
>>> rdd.keys().collect()                                    #获取 rdd 的 key
['B', 'N', 'I']
```

执行 rdd = sc.textFile("/user/hadoop/input/data.txt")语句后,Spark 从 data.txt 文件中加载数据到内存中,在内存中生成一个 RDD 对象 rdd。这个 rdd 里面包含了若干元素,元素的类型是字符串,从 data.txt 文件中读取的每一行文本内容都成为 rdd 中的一个元素。

使用 textFile()方法读取文件创建 RDD 时,可指定分区的个数。例如:

```
>>> rdd = sc.textFile("/user/hadoop/input/data.txt", 3)
                                                    #创建包含 3 个分区的 RDD 对象
```

2. 读取本地的文本文件创建 RDD

读取 Linux 本地文件也是通过 sc.textFile("路径")方法实现的,但需要在路径前面加上“file:”以表示从 Linux 本地文件系统读取。在 Linux 本地文件系统上存在一个文件/home/hadoop/data.txt,其内容和上面的 HDFS 中的文件/user/hadoop/input/data.txt 完全一样。

下面给出读取 Linux 本地的/home/hadoop/data.txt 文件创建一个 RDD 的例子:

```
>>> rdd1 = sc.textFile("file:/home/hadoop/data.txt")     #读取本地文件
>>> rdd1.foreach(print)                                  #输出 rdd1 中的每个元素
Business before pleasure.
Nothing is impossible to a willing heart.
I feel strongly that I can make it.
```

3. 读取目录创建 RDD

textFile()方法也可以读取目录。将目录作为参数,会将目录中的各个文件中的数据都读入 RDD 中。/home/hadoop/input 目录中有文件 text1.txt 和 text2.txt,text1.txt 中的内容为“Hello Spark”,text2.txt 中的内容为“Hello Python”。

```
>>> rddw1 = sc.textFile("file:/home/hadoop/input")   #读取本地文件夹
>>> rddw1.collect()
['Hello Python', 'Hello Spark']
```

4. 使用 wholeTextFiles()方法读取目录创建 RDD

SparkContext 对象的 wholeTextFiles()方法也可用来读取给定目录中的所有文件,可在输入路径时使用通配符(如 part-*.txt)。wholeTextFiles()方法会返回若干键值对

组成的 RDD，每个键值对的键是目录中一个文件的文件名，值是该文件名所表示的文件的内容。

```
>>> rddw2 = sc.wholeTextFiles ("file:/home/hadoop/input")        #读取本地目录
>>> rddw2.collect()
[('file:/home/hadoop/input/text2.txt', 'Hello Python\n'),
 ('file:/home/hadoop/input/text1.txt', 'Hello Spark\n')]
```

3.1.3　使用 JSON 文件创建 RDD

JSON(JavaScript Object Notation，JavaScript 对象标记)是一种轻量级的数据交换格式，JSON 文件在许多编程 API 中都得到支持。简单地说，JSON 可以将 JavaScript 对象表示的一组数据转换为字符串，然后就可以在网络或者程序之间轻松地传递这个字符串，并在需要的时候将它还原为各编程语言所支持的数据格式，是互联网上最受欢迎的数据交换格式。

在 JSON 语言中，一切皆对象。任何支持的类型都可以通过 JSON 表示，例如字符串、数字、对象、数组等。但是对象和数组是比较特殊且常用的两种类型。

对象在 JSON 中是用“{}”括起来的内容，采用{key1：value1，key2：value2，…}这样的键值对结构。在面向对象的语言中，key 为对象的属性，value 为对应的值。键名可以用整数和字符串表示，值可以是任意类型。

数组在 JSON 中是用“[]”括起来的内容，例如["Java"，"Python"，"VB"，…]。数组是一种比较特殊的数据类型，数组内也可以像对象那样使用键值对。

JSON 格式的 5 条规则如下：

(1) 并列的数据之间用“，”分隔。

(2) 映射(键值对)用“：”表示。

(3) 并列数据的集合(数组)用“[]”表示。

(4) 映射(键值对)的集合(对象)用“{}”表示。

(5) 元素值可具有的类型为 string、number、object(对象)、array(数组)，元素值也可以是 true、false、null。

在 Windows 系统中，可以使用记事本或其他类型的文本编辑器打开 JSON 文件以查看内容；在 Linux 系统中，可以使用 vim 编辑器打开和查看 JSON 文件。

例如，表示中国部分省市的 JSON 数据如下：

```
{
    "name": "中国",
    "province": [{
        "name": "河南",
        "cities": {
            "city": ["郑州", "洛阳"]
        }
```

```
        }, {
            "name": "广东",
            "cities": {
                "city": ["广州", "深圳"]
            }
        }, {
            "name": "陕西",
            "cities": {
                "city": ["西安", "咸阳"]
            }
        }]
    }
```

下面再给出一个 JSON 文件示例数据:

```
{
    "code": 0,
    "msg": "",
    "count": 2,
    "data": [
    {
        "id": "101",
        "username": "ZhangSan",
        "city":"XiaMen",
    }, {
        "id": "102",
        "username": "LiMing",
        "city": "ZhengZhou",
    }]
}
```

创建 JSON 文件的一种方法是:新建一个扩展名为.txt 的文本文件,在文件中写入 JSON 数据,保存该文件,将扩展名修改成.json,就成为 JSON 文件了。

在本地文件系统/home/hadoop/目录下有一个 student.json 文件,内容如下:

```
{"学号":"106","姓名":"李明","数据结构":"92"}
{"学号":"242","姓名":"李乐","数据结构":"96"}
{"学号":"107","姓名":"冯涛","数据结构":"84"}
```

从文件内容可看到每个"{…}"中为一个 JSON 格式的数据,一个 JSON 文件包含若干 JSON 格式的数据。

读取 JSON 文件创建 RDD 最简单的方法是将 JSON 文件作为文本文件读取。例如:

```
>>> jsonStr = sc.textFile("file:/home/hadoop/student.json")
>>> jsonStr.collect()
['{"学号":"106","姓名":"李明","数据结构":"92"}', '{"学号":"242","姓名":"李乐",
"数据结构":"96"}', '{"学号":"107","姓名":"冯涛","数据结构":"84"}']
```

3.1.4　使用 CSV 文件创建 RDD

CSV(Comma Separated Values,逗号分隔值)文件是一种用来存储表格数据(数字和文本)的纯文本格式文件。CSV 文件的内容由以“,”分隔的一列列数据构成,它可以被导入各种电子表格和数据库中。纯文本意味着该文件是一个字符序列。在 CSV 文件中,列之间以逗号分隔。CSV 文件由任意数目的记录组成,记录间以某种换行符分隔,一行为一条记录。可使用 Word、Excel、记事本等方式打开 CSV 文件。

创建 CSV 文件的方法有很多,最常用的方法是用电子表格创建。例如,在 Excel 中,选择“文件”→“另存为”命令,然后在“文件类型”下拉列表框中选择“CSV (逗号分隔)(*.csv)”,最后单击“保存”按钮,即创建了一个 CSV 文件。

如果 CSV 文件的所有数据字段均不包含换行符,可以使用 textFile()方法读取并解析数据。

例如,/home/hadoop 目录下保存了一个名为 grade.csv 的 CSV 文件,文件内容如下:

```
101,LiNing,95
102,LiuTao,90
103,WangFei,96
```

使用 textFile()方法读取 grade.csv 文件,创建 RDD:

```
>>> gradeRDD = sc.textFile("file:/home/hadoop/grade.csv")          #创建 RDD
>>> gradeRDD.collect()
['101,LiNing,95', '102,LiuTao,90', '103,WangFei,96']
```

3.2　RDD 转换操作

从相关数据源获取数据形成初始 RDD 后,根据应用需求,调用 RDD 对象的转换操作(算子)方法对得到的初始 RDD 进行操作,生成一个新的 RDD。对 RDD 的操作分为两大类型:转换操作和行动操作。Spark 里的计算就是操作 RDD。

转换操作负责对 RDD 中的数据进行计算并转换为新的 RDD。RDD 转换操作是惰性求值的,只记录转换的轨迹,而不会立即转换,直到遇到行动操作时才会与行动操作一起执行。

下面给出 RDD 对象的常用转换操作方法。


映射操作

3.2.1 映射操作

映射操作方法主要有 map()、flatMap()、mapValues()、flatMapValues()和 mapPartitions()。

1. map()

map(func)对一个 RDD 中的每个元素执行 func 函数,通过计算得到新元素,这些新元素组成的 RDD 作为 map(func)的返回结果。例如:

```
>>> rdd1 = sc.parallelize([1, 2, 3, 4])
>>> result=rdd1.map(lambda x:x * 2)          #用 map()对 rdd1 中的每个数进行乘 2 操作
>>> result.collect()                         #以列表形式返回 RDD 中的所有元素
[2, 4, 6, 8]
```

上述代码中,向 map()操作传入了一个匿名函数 lambda x:x * 2。其中,x 为函数的参数名称,也可以使用其他字符,如 y:x * 2 为函数解析式,用来实现函数的运算。Spark 会将 RDD 中的每个元素依次传入该函数的参数中,返回一个由所有函数值组成的新 RDD。

collect()为行动操作,将生成的 RDD 对象 result 转化为 list 类型,同时可实现查看 RDD 中数据的效果。

map(func)可用来将一个普通的 RDD 转换为一个键值对形式的 RDD,供只能操作键值对类型的 RDD 使用。

例如,对一个由英语单词组成的文本行,提取其中的第一个单词作为 key,将整个句子作为 value,建立键值对 RDD,具体实现如下:

```
>>> wordsRDD = sc.parallelize(["Who is that", "What are you doing", "Here you
are"])
>>> PairRDD = wordsRDD.map(lambda x: (x.split(" ")[0], x))
>>> PairRDD.collect()
[('Who', 'Who is that'), ('What', 'What are you doing'), ('Here', 'Here you are')]
```

2. flatMap()

flatMap(func)类似于 map(func),但又有所不同。flatMap(func)中的 func 函数会返回 0 个或多个元素,flatMap(func)将 func 函数返回的元素合并成一个 RDD,作为本操作的返回值。例如:

```
>>> wordsRDD = sc.parallelize(["Who is that", "What are you doing", "Here you
are"])
>>> FlatRDD = wordsRDD.flatMap(lambda x: x.split(" "))
>>> FlatRDD.collect()
['Who', 'is', 'that', 'What', 'are', 'you', 'doing', 'Here', 'you', 'are']
```

flatMap()的一个简单用途是把输入的字符串切分为单词。例如：

```
#定义函数
>>> def tokenize(ws):
        return ws.split(" ")
>>> lines = sc.parallelize(["One today is worth two tomorrows","Better late
than never","Nothing is impossible for a willing heart"])
>>> lines.map(tokenize).foreach(print)
['One', 'today', 'is', 'worth', 'two', 'tomorrows']
['Better', 'late', 'than', 'never']
['Nothing', 'is', 'impossible', 'for', 'a', 'willing', 'heart']
>>> lines.flatMap(tokenize).collect()
['One', 'today', 'is', 'worth', 'two', 'tomorrows', 'Better', 'late', 'than',
'never', 'Nothing', 'is', 'impossible', 'for', 'a', 'willing', 'heart']
```

3. mapValues()

mapValues(func)对键值对组成的 RDD 对象中的每个 value 都执行函数 func()，返回由键值对(key，func(value))组成的新 RDD，但是，key 不会发生变化。键值对 RDD 是指 RDD 中的每个元素都是(key，value)二元组，key 为键，value 为值。例如：

```
>>> rdd = sc.parallelize(["Hadoop","Spark","Hive","HBase"])
>>> pairRdd = rdd.map(lambda x: (x,1))                    #转换为键值对 RDD
>>> pairRdd.collect()
[('Hadoop', 1), ('Spark', 1), ('Hive', 1), ('HBase', 1)]
>>> pairRdd.mapValues(lambda x: x+1).foreach(print)     #对每个值加 1
('Hadoop', 2)
('Spark', 2)
('Hive', 2)
('HBase', 2)
```

再给出一个 mapValues()应用示例：

```
>>> rdd1 = sc.parallelize(list(range(1,9)))
>>> rdd1.collect()
[1, 2, 3, 4, 5, 6, 7, 8]
>>> result = rdd1.map(lambda x: (x % 4, x)).mapValues(lambda v: v + 10)
>>> result.collect()
[(1, 11), (2, 12), (3, 13), (0, 14), (1, 15), (2, 16), (3, 17), (0, 18)]
```

4. flatMapValues()

flatMapValues(func)转换操作把键值对 RDD 中的每个键值对的值都传给一个函数处理，对于每个值，该函数返回 0 个或多个输出值，键和每个输出值构成一个二元组，作为

flatMapValues(func)函数返回的新 RDD 中的一个元素。使用 flatMapValues(func)会保留原 RDD 的分区情况。

```
>>> stuRDD = sc.parallelize(['Wang,81|82|83','Li,76|82|80|','Liu,90|88|91'])
>>> kvRDD = stuRDD.map(lambda x: x.split(','))
>>> print('kvRDD: ',kvRDD.take(2))
kvRDD:  [['Wang', '81|82|83'], ['Li', '76|82|80|']]
>>> RDD = kvRDD.flatMapValues(lambda x: x.split('|')).map(lambda x:(x[0],int
(x[1])))
>>> print('RDD: ', RDD.take(6))
RDD:  [('Wang', 81), ('Wang', 82), ('Wang', 83), ('Li', 76), ('Li', 82), ('Li',
80)]
```

5. mapPartitions()

mapPartitions(func)对每个分区数据执行指定函数。

```
>>> rdd = sc.parallelize([1, 2, 3, 4],2)
>>> rdd.glom().collect()                    #查看每个分区中的数据
[[1, 2], [3, 4]]
>>> def f(x):
        yield sum(x)
>>> rdd.mapPartitions(f).collect()          #对每个分区中的数据执行 f 函数操作
[3, 7]
```

3.2.2 去重操作

去重操作包括 filter()和 distinct()。

1. filter()

filter(func)使用过滤函数 func 过滤 RDD 中的元素,func 函数的返回值为 Boolean 类型,filter(func)执行 func 函数后返回值为 true 的元素,组成新的 RDD。例如:

```
>>> rdd4=sc.parallelize([1,2,2,3,4,3,5,7,9])
>>> rdd4.filter(lambda x:x>4).collect()     #对 rdd4 进行过滤,得到大于 4 的数据
[5, 7, 9]
```

创建 4 名学生考试数据信息的 RDD,学生考试数据信息包括姓名、考试科目、考试成绩,各项之间用空格分隔。下面给出找出成绩为 100 的学生姓名和考试科目的具体命令语句。

(1) 创建学生考试数据信息的 RDD:

```
>>> students = sc.parallelize(["XiaoHua Scala 85","LiTao Scala 100","LiMing
Python 95","WangFei Java 100"])
```

(2) 将 students 的数据存储为 3 元组：

```
>>> studentsTup = students.map(lambda x : (x.split(" ")[0], x.split(" ")[1],
int(x.split(" ")[2])))
>>> studentsTup.collect()
[('XiaoHua', 'Scala', 85), ('LiTao', 'Scala', 100), ('LiMing', 'Python', 95),
('WangFei', 'Java', 100)]
```

(3) 过滤出成绩为 100 的学生的姓名和考试科目：

```
>>> studentsTup.filter(lambda x: x[2]==100).map(lambda x:(x[0], x[1])).
foreach(print)
('LiTao', 'Scala')
('WangFei', 'Java')
```

2. distinct()

distinct([numPartitions])对 RDD 中的数据进行去重操作，返回一个新的 RDD。其中，可选参数 numPartitions 用来设置操作的并行任务个数。例如：

```
>>> Rdd = sc.parallelize([1,2,1,5,3,5,4,8,6,4])
>>> distinctRdd = Rdd.distinct()
>>> distinctRdd.collect()
[1, 2, 5, 3, 4, 8, 6]
```

从返回结果[1，2，5，3，4，8，6]中可以看出，数据已经去重。

3.2.3　排序操作

排序操作包括 sortByKey()和 sortBy()。

1. sortByKey()

sortByKey(ascending，[numPartitions])对 RDD 中的数据集进行排序操作，对键值对类型的数据按照键进行排序，返回一个排序后的键值对类型的 RDD。参数 ascending 用来指定是升序还是降序，默认值是 True，按升序排序。可选参数 numPartitions 用来指定排序分区的并行任务个数。

```
>>> rdd = sc.parallelize([("WangLi", 1), ("LiHua", 3), ("LiuFei", 2),
("XuFeng", 1)])
>>> rdd.collect()
[('WangLi', 1), ('LiHua', 3), ('LiuFei', 2), ('XuFeng', 1)]
>>> rdd1 = rdd.sortByKey(False)                    #False 表示降序
>>> rdd1.collect()
[('XuFeng', 1), ('WangLi', 1), ('LiuFei', 2), ('LiHua', 3)]
```

2. sortBy()

sortBy(keyfunc,[ascending],[numPartitions])使用 keyfunc 函数先对数据进行处理,按照处理后的数据排序,默认为升序。sortBy()可以指定按键还是按值进行排序。

第一个参数 keyfunc 是一个函数,sortBy()按 keyfunc 对 RDD 中的每个元素计算的结果对 RDD 中的元素进行排序。

第二个参数是 ascending,决定排序后 RDD 中的元素是升序还是降序。默认是 True,按升序排序。

第三个参数是 numPartitions,该参数决定排序后的 RDD 的分区个数。默认排序后的分区个数和排序之前相等。

例如,创建 4 种商品数据信息的 RDD,商品数据信息包括名称、单价、数量,各项之间用空格分隔。命令如下:

```
>>> goods = sc.parallelize(["radio 30 50","soap 3 60","cup 6 50","bowl 4 80"])
```

(1) 按键进行排序,等同于 sortByKey()。

首先将 goods 的数据存储为 3 元组:

```
>>> goodsTup = goods.map(lambda x: (x.split(" ")[0], int(x.split(" ")[1]),
int(x.split(" ")[2])))
```

然后按商品名称进行排序:

```
>>> goodsTup.sortBy(lambda x:x[0]).foreach(print)
('bowl', 4, 80)
('cup', 6, 50)
('radio', 30, 50)
('soap', 3, 60)
```

(2) 按值进行排序。

按照商品单价降序排序:

```
>>> goodsTup.sortBy(lambda x:x[1], False).foreach(print)
('radio', 30, 50)
('cup', 6, 50)
('bowl', 4, 80)
('soap', 3, 60)
```

按照商品数量升序排序:

```
>>> goodsTup.sortBy(lambda x:x[2]).foreach(print)
('radio', 30, 50)
('cup', 6, 50)
```

```
('soap', 3, 60)
('bowl', 4, 80)
```

按照商品数量与 7 相除的余数升序排序：

```
>>> goodsTup.sortBy(lambda x:x[2]%7).foreach(print)
('radio', 30, 50)
('cup', 6, 50)
('bowl', 4, 80)
('soap', 3, 60)
```

（3）通过 Tuple 方式，按照数组的元素进行排序：

```
>>> goodsTup.sortBy(lambda x: (-x[1], -x[2])).foreach(print)
('radio', 30, 50)
('cup', 6, 50)
('bowl', 4, 80)
('soap', 3, 60)
```

3.2.4　分组聚合操作

分组聚合操作包括 groupBy()、groupByKey()、groupWith()、reduceByKey()和 combineByKey()。

1. groupBy()

groupBy(func)返回一个按指定条件(用函数 func 表示)对元素进行分组的 RDD。参数 func 可以是有名称的函数，也可以是匿名函数，用来指定对所有元素进行分组的键，或者指定对元素进行求值以确定其所属分组的表达式。注意，groupBy()返回的是一个可迭代对象，称为迭代器。例如：

```
>>> rdd=sc.parallelize([1,2,3,4,5, 6, 7, 8])
>>> res=rdd.groupBy(lambda x:x%2).collect()
>>> for x,y in res:                              #输出迭代器的具体值
        print(x)
        print(y)
        print(sorted(y))
        print("*"*44)
1
<pyspark.resultiterable.ResultIterable object at 0x7fe71012ea60>
[1, 3, 5, 7]
********************************************
0
<pyspark.resultiterable.ResultIterable object at 0x7fe70de43bb0>
```

```
[2, 4, 6, 8]
*****************************************
```

2. groupByKey()

groupByKey()对一个由键值对(K,V)组成的 RDD 进行分组聚合操作,返回由键值对(K,Seq[V])组成的新 RDD,Seq[V]表示由键相同的值所组成的序列。

```
>>> rdd=sc. parallelize([("Spark",1),("Spark",1),("Hadoop",1),("Hadoop",1)])
>>> rdd. groupByKey().map(lambda x : (x[0], list(x[1]))).collect()
[('Spark', [1, 1]), ('Hadoop', [1, 1])]
>>> rdd. groupByKey().map(lambda x : (x[0], len(list(x[1])))).collect()
[('Spark', 2), ('Hadoop', 2)]
```

3. groupWith()

groupWith(otherRDD1, otherRDD2,…)把多个 RDD 按键进行分组,输出(键,迭代器)形式的数据。分组后的数据是有顺序的,每个键对应的值是按列出 RDD 的顺序排序的。如果 RDD 没有键,则对应位置取空值。例如:

```
>>> w = sc.parallelize([("a", "w"), ("b", "w")])
>>> x = sc.parallelize([("a", "x"), ("b", "x")])
>>> y = sc.parallelize([("a", "y")])
>>> z = sc.parallelize([("b", "z")])
>>> w.groupWith(x, y, z).collect()
[('b', (<pyspark.resultiterable.ResultIterable object at 0x7fe70de3abb0>,
        <pyspark.resultiterable.ResultIterable object at 0x7fe70ddea2b0>,
        <pyspark.resultiterable.ResultIterable object at 0x7fe70ddea310>,
        <pyspark.resultiterable.ResultIterable object at 0x7fe70ddea370>)),
 ('a', (<pyspark.resultiterable.ResultIterable object at 0x7fe70ddea3d0>,
        <pyspark.resultiterable.ResultIterable object at 0x7fe70ddea430>,
        <pyspark.resultiterable.ResultIterable object at 0x7fe70ddea490>,
        <pyspark.resultiterable.ResultIterable object at 0x7fe70ddea4f0>))]
```

迭代输出每个分组:

```
>>> [(x, tuple(map(list, y))) for x, y in list(w.groupWith(x, y, z).collect())]
[('b', (['w'], ['x'], [], ['z'])), ('a', (['w'], ['x'], ['y'], []))]
```

4. reduceByKey()

reduceByKey(func)对一个由键值对组成的 RDD 进行聚合操作,对键相同的值,使用指定的 func 函数将它们聚合到一起。例如:

```
>>> rdd=sc. parallelize([("Spark",1),("Spark",2),("Hadoop",1),("Hadoop",5)])
>>> rdd.reduceByKey(lambda x, y: x+ y).collect()
[('Spark', 3), ('Hadoop', 6)]
```

下面给出一个统计词频的例子：

```
>>> wordsRDD = sc.parallelize(["HewhodoesnotreachtheGreatWallisnotatrueman", "
He who has never been to the Great Wall is not a true man"])  #创建 RDD
>>> FlatRDD = wordsRDD.flatMap(lambda x: x.split(" "))
>>> FlatRDD.collect()
['He', 'who', 'does', 'not', 'reach', 'the', 'Great', 'Wall', 'is', 'not', 'a',
'true', 'man', ' ', 'He', 'who', 'has', 'never', 'been', 'to', 'the', 'Great', '
Wall', 'is', 'not', 'a', 'true', 'man']
>>> KVRdd = FlatRDD.map(lambda x:(x,1))                    #创建键值对 RDD
>>> KVRdd.collect()
[('He', 1), ('who', 1), ('does', 1), ('not', 1), ('reach', 1), ('the', 1), ('
Great', 1), ('Wall', 1), ('is', 1), ('not', 1), ('a', 1), ('true', 1), ('man', 1),
('', 1), ('He', 1), ('who', 1), ('has', 1), ('never', 1), ('been', 1), ('to', 1),
('the', 1), ('Great', 1), ('Wall', 1), ('is', 1), ('not', 1), ('a', 1), ('true',
1), ('man', 1)]
>>> KVRdd.reduceByKey(lambda x, y: x+ y).collect()       #统计词频
[('He', 2), ('who', 2), ('does', 1), ('not', 3), ('reach', 1), ('the', 2), ('
Great', 2), ('Wall', 2), ('is', 2), ('a', 2), ('true', 2), ('man', 2), (' ', 1), ('
has', 1), ('never', 1), ('been', 1), ('to', 1)]
```

5. combineByKey()

combineByKey(createCombiner,mergeValue,mergeCombiners)是对键值对 RDD 中的每个键值对按照键进行聚合操作,即合并相同键的值。聚合操作的逻辑是通过自定义函数提供给 combineByKey()方法的,把键值对(K,V)类型的 RDD 转换为键值对(K,C)类型的 RDD,其中 C 表示聚合对象类型。

3 个参数含义如下：

(1) createCombiner 是函数。在遍历(K,V)时,若 combineByKey()是第一次遇到键为 K 的键值对,则对该键值对调用 createCombiner 函数将 V 转换为 C,C 会作为 K 的累加器的初始值。

(2) mergeValue 是函数。在遍历(K,V)时,若 comineByKey()不是第一次遇到键为 K 的键值对,则对该键值对调用 mergeValue 函数将 V 累加到 C 中。

(3) mergeCombiners 是函数。combineByKey()是在分布式环境中执行的,RDD 的每个分区单独进行 combineBykey()操作,最后需要利用 mergeCombiners 函数对各个分区进行最后的聚合。

下面给出一个例子。

(1) 定义 createCombiner 函数:

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

(2) 定义 mergeValue 函数:

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

(3) 定义 mergeCombiners 函数:

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

(4) 创建考试成绩 RDD 对象:

```
>>> Rdd = sc.parallelize([('ID1', 80),('ID2', 85),('ID1', 90),('ID2', 95),
('ID3', 99)], 2)
> > > combineByKeyRdd = Rdd. combineByKey (createCombiner, mergeValue,
mergeCombiners)
>>> combineByKeyRdd.collect()
[('ID1', (170, 2)), ('ID2', (180, 2)), ('ID3', (99, 1))]
```

(5) 求平均成绩:

```
>>> avgRdd = combineByKeyRdd.map(lambda x:(x[0],float(x[1][0])/x[1][1]))
>>> avgRdd.collect()
[('ID1', 85.0), ('ID2', 90.0), ('ID3', 99.0)]
```

3.2.5 集合操作

集合操作包括 union()、intersection()、subtract()和 cartesian()。

1. union()

union(otherRDD)对源 RDD 和参数 otherRDD 指定的 RDD 求并集后返回一个新的 RDD,不进行去重操作。例如:

```
>>> rdd1 = sc.parallelize(list(range(1,5)))
>>> rdd2 = sc.parallelize(list(range(3,7)))
>>> rdd1.union(rdd2).collect()
[1, 2, 3, 4, 3, 4, 5, 6]
```

2. intersection()

intersection(otherRDD)对源 RDD 和参数 otherRDD 指定的 RDD 求交集后返回一个新的 RDD,且进行去重操作。例如:

```
>>> rdd1.intersection(rdd2).collect()
[4, 3]
```

3. subtract()

subtract(otherRDD)相当于进行集合的差集操作,即从源 RDD 中去除与参数 otherRDD 指定的 RDD 中相同的元素。例如:

```
>>> rdd1.subtract(rdd2).collect()
[2, 1]
```

4. cartesian()

cartesian(otherRDD)对源 RDD 和参数 otherRDD 指定的 RDD 进行笛卡儿积操作。例如:

```
>>> rdd1.cartesian(rdd2).collect()
[(1, 3), (1, 4), (1, 5), (1, 6), (2, 3), (2, 4), (2, 5), (2, 6), (3, 3), (3, 4), (3,
5), (3, 6), (4, 3), (4, 4), (4, 5), (4, 6)]
```

3.2.6　抽样操作

抽样操作包括 sample()和 sampleByKey()。

1. sample()

sample(withReplacement, fraction, seed)操作以指定的抽样种子 seed 从 RDD 的数据中抽取比例为 fraction 的数据。withReplacement 表示抽出的数据是否放回,True 为有放回的抽样,False 为无放回的抽样。相同的 seed 得到的随机序列一样。

```
>>> SampleRDD=sc.parallelize(list(range(1,1000)))
>>> SampleRDD.sample(False,0.01,1).collect()      #输出取样
[14, 100, 320, 655, 777, 847, 858, 884, 895, 935]
```

2. sampleByKey()

sampleByKey(withReplacement, fractions, seed)按键的比例抽样,withReplacement 表示是否有放回,fractions 表示抽样比例,seed 表示抽样种子。例如:

```
>>> fractions = {"a":0.5, "b":0.1}
>>> rdd = sc.parallelize(fractions.keys(), 3).cartesian(sc.parallelize
(range(0,10),2))
>>> sample = dict(rdd.sampleByKey(False,fractions,2).groupByKey(3).collect())
>>> [(iter[0],list(iter[1])) for iter in sample.items()]
[('b', [5, 9]), ('a', [1, 4, 5, 7])]
```

3.2.7 连接操作

连接操作包括 join()、leftOuterJoin()、rightOuterJoin()和 fullOuterJoin()。

1. join()

join(otherRDD, [numPartitions])对两个键值对 RDD 进行内连接,将两个 RDD 中键相同的(K, V)和(K, W)进行连接,返回键值对(K, (V, W))。其中,V 表示源 RDD 的值,W 表示参数 otherRDD 指定的 RDD 的值。例如:

```
>>> pairRDD1 = sc.parallelize([("Scala",2), ("Scala", 3), ("Java", 4),
("Python", 8)])
>>> pairRDD2 = sc.parallelize([ ("Scala",3), ("Java", 5), ("HBase", 4),
( "Java", 10)])
>>> pairRDD3 = pairRDD1.join(pairRDD2)
>>> pairRDD3.collect()
[('Java', (4, 5)), ('Java', (4, 10)), ('Scala', (2, 3)), ('Scala', (3, 3))]
```

2. leftOuterJoin()

leftOuterJoin()可用来对两个键值对 RDD 进行左外连接操作,保留第一个 RDD 的所有键。在左外连接中,如果第二个 RDD 中有对应的键,则连接结果中显示为 Some 类型,表示有值可以引用;如果没有,则为 None 值。例如:

```
>>> left_Join = pairRDD1.leftOuterJoin(pairRDD2)
>>> left_Join.collect()
[('Java', (4, 5)), ('Java', (4, 10)), ('Python', (8, None)), ('Scala', (2, 3)),
('Scala', (3, 3))]
```

3. rightOuterJoin()

rightOuterJoin()可用来对两个键值对 RDD 进行右外连接操作,确保第二个 RDD 的键必须存在,即保留第二个 RDD 的所有键。

4. fullOuterJoin()

fullOuterJoin()是全外连接操作,会保留两个 RDD 中所有键的连接结果。例如:

```
>>> full_Join = pairRDD1.fullOuterJoin (pairRDD2)
>>> full_Join.collect()
[('Java', (4, 5)), ('Java', (4, 10)), ('Python', (8, None)), ('Scala', (2, 3)),
('Scala', (3, 3)), ('HBase', (None, 4))]
```

3.2.8　打包操作

zip(otherRDD)将两个 RDD 打包成键值对形式的 RDD，要求两个 RDD 的分区数量以及每个分区中元素的数量都相同。例如：

```
>>> rdd1=sc.parallelize([1, 2, 3], 3)
>>> rdd2=sc.parallelize(["a","b","c"], 3)
>>> zipRDD=rdd1.zip(rdd2)
>>> zipRDD.collect()
[(1, 'a'), (2, 'b'), (3, 'c')]
```

3.2.9　获取键值对 RDD 的键和值集合

对一个键值对 RDD，调用 keys()返回一个仅包含键的 RDD，调用 values()返回一个仅包含值的 RDD。

```
>>> zipRDD.keys().collect()
[1, 2, 3]
>>> zipRDD.values().collect()
['a', 'b', 'c']
```

3.2.10　重新分区操作

重新分区操作包括 coalesce()和 repartition()。

1. coalesce()

在分布式集群里，网络通信的代价很大，减少网络传输可以极大地提升性能。MapReduce 框架的性能开销主要在 I/O 和网络传输两方面。I/O 因为要大量读写文件，性能开销是不可避免的；但可以通过优化方法降低网络传输的性能开销，例如把大文件压缩为小文件可减少网络传输的开销。

I/O 在 Spark 中也是不可避免的，但 Spark 对网络传输进行了优化。Spark 对 RDD 进行分区(分片)，把这些分区放在集群的多个计算节点上并行处理。例如，把 RDD 分成 100 个分区，平均分布到 10 个节点上，一个节点上有 10 个分区。当进行求和型计算的时候，先进行每个分区的求和，然后把分区求和得到的结果传输到主程序进行全局求和，这样就可以降低求和计算时网络传输的开销。

coalesce(numPartitions, shuffle)的作用是：默认使用 HashPartitioner(哈希分区方

式)对 RDD 进行重新分区,返回一个新的 RDD,且该 RDD 的分区个数等于 numPartitions。

参数说明如下:

(1) numPartitions:要生成的新 RDD 的分区个数。

(2) shuffle:指定是否进行洗牌。默认为 False,重设的分区个数只能比 RDD 原有分区数小;如果 shuffle 为 True,重设的分区个数不受原有 RDD 分区个数的限制。

下面给出一个例子:

```
>>> rdd =sc.parallelize(range(1,17), 4)          #创建 RDD,分区个数为 4
>>> rdd.getNumPartitions()                       #查看 RDD 分区个数
4
>>> coalRDD=rdd.coalesce(5)                      #重新分区,分区个数为 5
>>> coalRDD.getNumPartitions()
4
>>> coalRDD1 =rdd.coalesce(5, True)              #重新分区,shuffle 为 True
>>> coalRDD1.getNumPartitions()                  #查看 coalRDD1 分区个数
5
```

2. repartition()

repartition(numPartitions)其实就是 coalesce()方法的第二个参数 shuffle 为 True 的简单实现。例如:

```
>>> coalRDD2 = coalRDD1.repartition(2)           #转换成两个分区的 RDD
>>> coalRDD2.getNumPartitions()                  #查看 coalRDD2 分区个数
2
```

Spark 支持自定义分区方式,即通过一个自定义的分区函数对 RDD 进行分区。需要注意,Spark 的分区函数针对的是键值对类型的 RDD,分区函数根据键对 RDD 的元素进行分区。因此,当需要对一些非键值对类型的 RDD 进行自定义分区时,需要先把该 RDD 转换成键值对类型的 RDD,然后再使用分区函数。

下面给出一个自定义分区的实例,要求根据键的最后一位数字将键值对写入不同的分区中。打开一个 Linux 终端,使用 gedit 编辑器创建一个代码文件,将其命名为/usr/local/spark/myproject/rdd/partitionerTest.py,输入以下代码:

```
from pyspark import SparkConf,SparkContext
def SelfPartitioner(key):                        #自定义分区函数
    print('Self Defined Partitioner is running')
    print('The key is %d'%key)
    return key%5                                 #设定分区方式
def main():
    print('The main function is running')
```