第3章 Spark RDD弹性分布式数据集

学习目标

- 理解 RDD 的五大特征。
- 掌握 RDD 的创建方法。
- 掌握 RDD 的转换算子和行动算子的操作方法。
- 了解 RDD 之间的依赖关系。
- 了解 RDD 的持久化和容错机制。
- 理解 Spark 的任务调度。

传统的 MapReduce 虽然具有自动容错、平衡负载和可拓展性强的优点，但是其最大缺点是采用非循环式的数据流模型，使得在迭代计算时要进行大量的磁盘 I/O 操作。Spark 中的 RDD 可以很好地解决这一缺点。RDD 是 Spark 提供的最重要的抽象概念，可以将 RDD 理解为一个分布式存储在集群中的大型数据集合，不同 RDD 之间可以通过转换操作形成依赖关系实现管道化，从而避免了中间结果的 I/O 操作，提高数据处理的速度和性能。接下来，本章将针对 RDD 进行详细讲解。

3.1 RDD 简介

RDD(Resilient Distributed Dataset，弹性分布式数据集)，是一个容错的、并行的数据结构，可以让用户显式地将数据存储到磁盘和内存中，并且还能控制数据的分区。对于迭代式计算和交互式数据挖掘，RDD 可以将中间计算的数据结果保存在内存中，若是后面需要中间结果参与计算时，则可以直接从内存中读取，从而可以极大地提高计算速度。

每个 RDD 都具有五大特征，具体如下。

1. 分区列表(a list of partitions)

每个 RDD 被分为多个分区(Partitions)，这些分区运行在集群中的不同节点，每个分区都会被一个计算任务处理，分区数决定了并行计算的数量，创建 RDD 时可以指定 RDD 分区的个数。如果不指定分区数量，当 RDD 从集合创建时，默认分区数量为该程序所分配到的资源的 CPU 核数(每个 Core 可以承载 2～4 个 Partition)，如果是从 HDFS 文件创建，默认为文件的 Block 数。

2．每个分区都有一个计算函数(a function for computing each split)

Spark 的 RDD 的计算函数是以分片为基本单位的，每个 RDD 都会实现 compute 函数，对具体的分片进行计算。

3．依赖于其他 RDD(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．(Key，Value)数据类型的 RDD 分区器(a Partitioner for Key-Value RDDS)

当前 Spark 中实现了两种类型的分区函数，一个是基于哈希的 HashPartitioner，另外一个是基于范围的 RangePartitioner。只有对于(Key，Value)的 RDD，才会有 Partitioner(分区)，非(Key，Value)的 RDD 的 Parititioner 的值是 None。Partitioner 函数不但决定了 RDD 本身的分区数量，也决定了 parent RDD Shuffle 输出时的分区数量。

5．每个分区都有一个优先位置列表(a list of preferred locations to compute each split on)

优先位置列表会存储每个 Partition 的优先位置，对于一个 HDFS 文件来说，就是每个 Partition 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。

3.2　RDD 的创建方式

Spark 提供了两种创建 RDD 的方式，分别是从文件系统(本地和 HDFS)中加载数据创建 RDD 和通过并行集合创建 RDD。接下来，本节将讲解 RDD 的两种创建方式。

3.2.1　从文件系统加载数据创建 RDD

Spark 可以从 Hadoop 支持的任何存储源中加载数据去创建 RDD，包括本地文件系统和 HDFS 等文件系统。

接下来，通过 Spark 中的 SparkContext 对象调用 textFile()方法加载数据创建 RDD。这里以 Linux 本地系统和 HDFS 为例，讲解如何创建 RDD。

1．从 Linux 本地文件系统加载数据创建 RDD

在 Linux 本地文件系统中有一个名为 test.txt 的文件，具体内容如文件 3-1 所示。

文件 3-1　test.txt

```
1  hadoop spark
2  itcast heima
3  scala spark
```

```
4  spark itcast
5  itcast hadoop
```

在 Linux 本地系统读取 test.txt 文件数据创建 RDD，具体代码如下：

```
scala>val test=sc.textFile("file:///export/data/test.txt")
test: org.apache.spark.rdd.RDD[String]=file:///export/data/test.txt
                              MapPartitionsRDD[1] at textFile at <console>:24
```

上述的代码中，文件路径中的 file://表示从本地 Linux 文件系统中读取文件。test：org.apache.spark.rdd.RDD[String]…是命令执行后返回的信息，而 test 则是一个创建好的 RDD。当执行 textFile()方法后，Spark 会从 Linux 本地文件 test.txt 中加载数据到内存中，在内存中生成了一个 RDD 对象（即 test），并且这个 RDD 里面包含若干个 String 类型的元素，也就是说，从 test.txt 文件中读取出来的每一行文本内容，都是 RDD 中的一个元素。

2. 从 HDFS 中加载数据创建 RDD

假设，在 HDFS 上的/data 目录下有一个名为 test.txt 的文件，该文件内容与文件 3-1 相同。接下来，通过加载 HDFS 中的数据创建 RDD，具体代码如下：

```
scala>val testRDD=sc.textFile("/data/test.txt")
testRDD:org.apache.spark.rdd.RDD[String]=/data/test.txt MapPartitionsRDD[1]
                                                 at textFile at <console>:24
```

执行上述代码后，从返回结果 testRDD 的属性中看出 RDD 创建完成。在上述代码中，通过 textFile("/data/test.txt")方法来读取 HDFS 上的文件，其中方法 testFile()中的参数为/data/test.txt 文件路径，传入的参数也可以为 hdfs://localhost:9000/data/test.txt 和/test.txt 路径，最终所达到的效果是一致的。

3.2.2 通过并行集合创建 RDD

Spark 可以通过并行集合创建 RDD。即从一个已经存在的集合、数组上，通过 SparkContext 对象调用 parallelize()方法创建 RDD。

若要创建 RDD，则需要先创建一个数组，再通过执行 parallelize()方法实现，具体代码如下：

```
scala>val array=Array(1,2,3,4,5)
array: Array[Int]=Array(1,2,3,4,5)
scala>val arrRDD=sc.parallelize(array)
arrRDD: org.apache.spark.rdd.RDD[Int]=ParallelcollectionRDD[6] at parallelize
                                                               at <console>:26
```

执行上述代码后，从返回结果 arrRDD 的属性中看出 RDD 创建完成。

3.3 RDD 的处理过程

Spark 用 Scala 语言实现了 RDD 的 API,程序开发者可以通过调用 API 对 RDD 进行操作处理。下面,通过图 3-1 来描述 RDD 的处理过程。

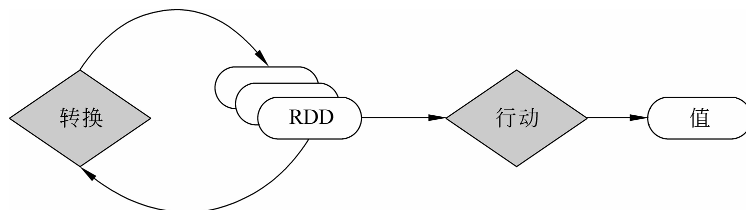

图 3-1 RDD 的处理过程

在图 3-1 中,RDD 经过一系列的“转换”操作,每一次转换都会产生不同的 RDD,以供给下一次“转换”操作使用,直到最后一个 RDD 经过“行动”操作才会被真正计算处理,并输出到外部数据源中,若是中间的数据结果需要复用,则可以进行缓存处理,将数据缓存到内存中。

需要注意的是,RDD 采用了惰性调用,即在 RDD 的处理过程中,真正的计算发生在 RDD 的“行动”操作,对于“行动”之前的所有“转换”操作,Spark 只是记录下“转换”操作应用的一些基础数据集以及 RDD 生成的轨迹(即 RDD 相互之间的依赖关系),而不会触发真正的计算处理。

接下来,将针对 RDD 处理过程中的“转换”操作和“行动”操作进行详细的讲解。

3.3.1 转换算子

RDD 处理过程中的“转换”操作主要用于根据已有 RDD 创建新的 RDD,每一次通过 Transformation 算子计算后都会返回一个新 RDD,供给下一个转换算子使用。下面,通过表 3-1 来列举一些常用的转换算子操作的 API。

表 3-1 常用的转换算子 API

转换算子	相关说明
filter(func)	筛选出满足函数 func 的元素,并返回一个新的数据集
map(func)	将每个元素传递到函数 func 中,返回的结果是一个新的数据集
flatMap(func)	与 map()相似,但是每个输入的元素都可以映射到 0 或者多个输出结果
groupByKey()	应用于(Key,Value)键值对的数据集时,返回一个新的(Key,Iterable <Value>)形式的数据集
reduceByKey(func)	应用于(Key,Value)键值对的数据集时,返回一个新的(Key,Value)形式的数据集。其中,每个 Value 值是将每个 Key 键传递到函数 func 中进行聚合后的结果

下面,结合具体的示例对这些转换算子 API 进行详细讲解。

1. filter(func)

filter(func)操作会筛选出满足函数 func 的元素，并返回一个新的数据集。假设，有一个文件 test.txt(内容如文件 3-1 所示)，接下来，通过一张图来描述如何通过 filter 算子操作筛选出包含单词 spark 的元素，具体过程如图 3-2 所示。

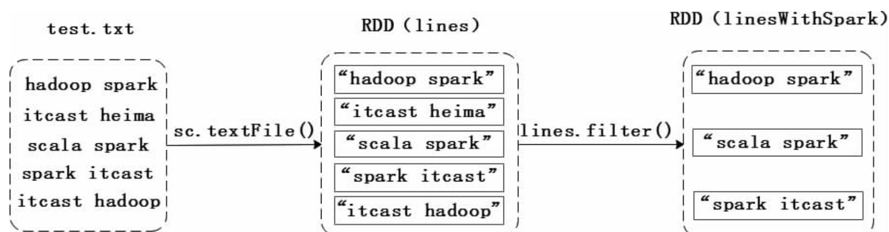

图 3-2　filter 算子操作

在图 3-2 中，通过从 test.txt 文件中加载数据的方式创建 RDD，然后通过 filter 操作筛选出满足条件的元素，这些元素组成的集合是一个新的 RDD。接下来，通过代码来进行演示，具体代码如下：

```
scala>val lines =sc.textFile("file:///export/data/test.txt")
lines: org.apache.spark.rdd.RDD[String] =file:///export/data/test.txt
                        MapPartitionsRDD[1] at textFile at <console>:24
scala>val linesWithSpark =lines.filter(line =>line.contains("spark"))
linesWithSpark: org.apache.spark.rdd.RDD[String] =MapPartitionsRDD[2] at
                                                      filter at <console>:25
```

在上述代码中，filter()输入的参数 line => line.contains("spark")是一个匿名函数，其含义是依次取出 lines 这个 RDD 中的每一个元素，对于当前取到的元素，把它赋值给匿名函数中的 line 变量。若 line 中包含 spark 单词，就把这个元素加入到 RDD(即 linesWithSpark)中，否则就丢弃该元素。

2. map(func)

map(func)操作将每个元素传递到函数 func 中，并将结果返回为一个新的数据集。假设，有一个文件 test.txt(内容如文件 3-1 所示)，接下来，通过一张图来描述如何通过 map 算子操作把文件内容拆分成一个个的单词并封装在数组对象中，具体过程如图 3-3 所示。

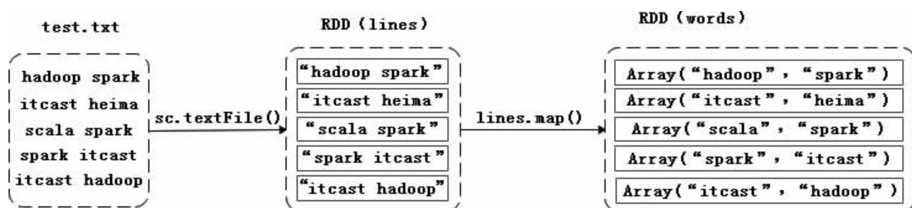

图 3-3　map 算子操作

在图 3-3 中，通过从 test.txt 文件中加载数据的方式创建 RDD，然后通过 map 操作将

文件的每一行内容都拆分成一个个的单词元素,这些元素组成的集合是一个新的 RDD。接下来,通过代码来进行演示,具体代码如下:

```
scala>val lines =sc.textFile("file:///export/data/test.txt")
lines: org.apache.spark .rdd.RDD[String] =file:///export/data/test.txt
                         MapPartitionsRDD[4] at textFile at <console>:24
scala>val words =lines.map(line =>line.split(" "))
words: org.apache.spark.rdd.RDD[Array[String]] =MapPartitionsRDD[13] at
                                                      map at <console>:25
```

上述代码中,lines.map(line => line.split(" "))含义是依次取出 lines 这个 RDD 中的每个元素,对于当前取到的元素,把它赋值给匿名函数中的 line 变量。由于 line 是一行文本,如 hadoop spark,一行文本中包含多个单词,且用空格进行分隔,通过 line.split(" ")匿名函数,将文本分成一个个的单词,拆分后得到的单词都被封装到一个数组对象中,成为新的 RDD(即 words)的一个元素。

3. flatMap(func)

flatMap(func)与 map(func)相似,但是每个输入的元素都可以映射到 0 或者多个输出的结果。有一个文件 test.txt(内容如文件 3-1 所示),接下来,通过一张图来描述如何通过 flatMap 算子操作把文件内容拆分成一个个的单词,具体过程如图 3-4 所示。

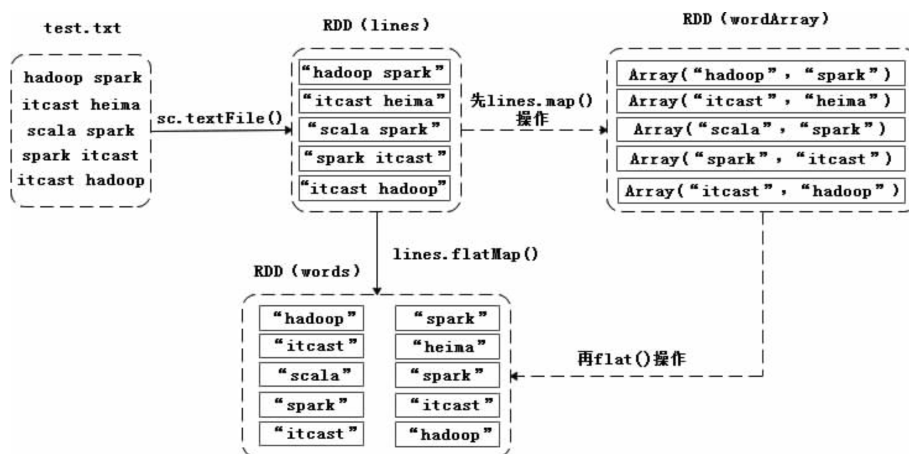

图 3-4 flatMap 算子操作

在图 3-4 中,通过从 test.txt 文件中加载数据的方式创建 RDD,然后通过 flatMap 操作将文件的每一行内容都拆分成一个个的单词元素,这些元素组成的集合是一个新的 RDD。接下来,通过代码来进行演示,具体代码如下:

```
scala>val lines =sc.textFile("file:///export/data/test.txt")
lines: org.apache.spark .rdd.RDD[String] =file:///export/data/test.txt
                         MapPartitionsRDD[5] at textFile at <console>:24
scala>val words =lines.flatMap(line =>line.split(" "))
```

```
words: org.apache.spark.rdd.RDD[Array[String]] =MapPartitionsRDD[14] at
                                                  map at <console>:25
```

在上述代码中,lines. flatMap(line => line. split(" "))等价于先执行 lines. map(line => line. split(" "))操作(请参考 map(func)操作),再执行 flat()操作(即扁平化操作),把 wordArray 中的每个 RDD 都扁平成多个元素,被扁平后得到的元素构成一个新的 RDD(即 words)。

4. groupByKey()

groupByKey()主要用于(Key,Value)键值对的数据集,将具有相同 Key 的 Value 进行分组,会返回一个新的(Key,Iterable)形式的数据集。同样以文件 test. txt(内容如文件 3-1 所示)为例,接下来,通过一张图来描述如何通过 groupByKey 算子操作将文件内容中的所有单词进行分组,具体过程如图 3-5 所示。

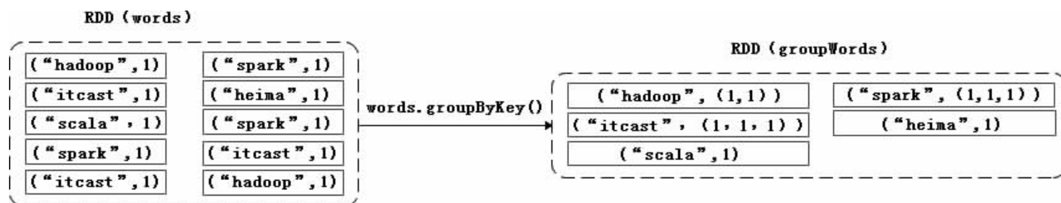

图 3-5 groupByKey 算子操作

在图 3-5 中,通过 groupByKey 操作把(Key,Value)键值对类型的 RDD 按单词出现的次数进行分组,这些元素组成的集合是一个新的 RDD。接下来,通过代码来进行演示,具体代码如下:

```
scala>val lines =sc.textFile("file:///export/data/test.txt")
lines: org.apache.spark.rdd.RDD[String] =file:///export/data/test.txt
                    MapPartitionsRDD[6] at textFile at <console>:24
scala>val words=lines.flatMap(line=>line.split(" ")).map(word=>(word,1))
words: org.apache.spark.rdd.RDD[(String, Int)] =MapPartitionsRDD[15] at
                                                  map at <console>:25
scala>val groupWords=words.groupByKey()
groupWords: org.apache.spark.rdd.RDD[(String,Iterable[Int])]=ShuffledRDD[16]
                                              at groupByKey at <console>:25
```

上述代码中,words. groupByKey()操作执行后,RDD 中所有的 Key 相同的 Value 都被合并到一起。例如,("spark",1)、("spark",1)、("spark",1)这 3 个键值对的 Key 都是 spark,合并后得到新的键值对("spark",(1,1,1))。

5. reduceByKey(func)

reduceByKey()主要用于(Key,Value)键值对的数据集,返回的是一个新的(Key,Value)形式的数据集,该数据集是每个 Key 传递给函数 func 进行聚合运算后得到的结果。同样以文件 test. txt(内容如文件 3-1 所示)为例,接下来,通过一张图来描述如何通过

reduceByKey 算子操作统计单词出现的次数，具体操作如图 3-6 所示。

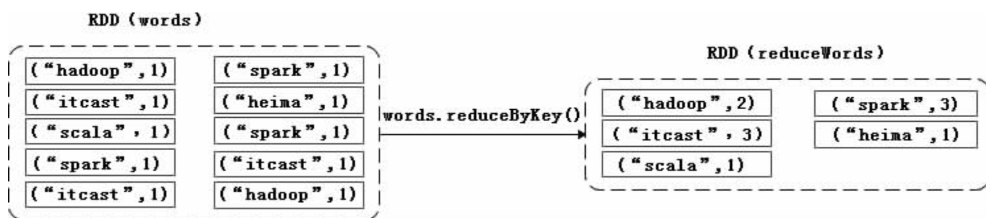

图 3-6　reduceByKey()算子操作

在图 3-6 中，通过 reduceByKey 操作把(Key，Value)键值对类型的 RDD，按单词 Key 出现的次数 Value 进行聚合，这些元素组成的集合是一个新的 RDD。接下来，通过代码来进行演示，具体代码如下：

```
scala>val lines =sc.textFile("file:///export/data/test.txt")
lines: org.apache.spark.rdd.RDD[String] =file:///export/data/test.txt
                        MapPartitionsRDD[7] at textFile at <console>:24
scala>val words=lines.flatMap(line=>line.split(" ")).map(word=>(word,1))
words: org.apache.spark.rdd.RDD[(String, Int)] =MapPartitionsRDD[16] at
                                                    map at <console>:25
scala>val reduceWords=words.reduceByKey((a,b)=>a+b)
reduceWords: org.apache.spark.rdd.RDD[(String, Int)] =ShuffledRDD[17] at
                                             reduceByKey at <console>:25
```

上述代码中，执行 words.reduceByKey((a,b) => a + b)操作，共分为两个步骤，分别是先执行 reduceByKey()操作，将所有 Key 相同的 Value 值合并到一起，生成一个新的键值对，如("spark",(1,1,1))；然后执行函数 func 的操作，即使用(a,b)=> a + b 函数把(1,1,1)进行聚合求和，得到最终的结果，即("spark",3)。

3.3.2　行动算子

行动算子主要是将在数据集上运行计算后的数值返回到驱动程序，从而触发真正的计算。下面列举一些常用的行动算子 API，如表 3-2 所示。

表 3-2　常用的行动算子 API

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

下面，结合具体的示例对这些行动算子 API 进行详细讲解。

1. count()

count()主要用于返回数据集中的元素个数。假设,现有一个 arrRdd,如果要统计 arrRdd 元素的个数,示例代码如下:

```
scala>val arrRdd=sc.parallelize(Array(1,2,3,4,5))
arrRdd: org.apache.spark.rdd.RDD[Int]=
                 ParallelcollectionRDD[0] at parallelize at <console>:24
scala>arrRdd.count()
res0: Long =5
```

上述代码中,第 1 行代码创建了一个 RDD 对象,当 arrRdd 调用 count()操作后,返回的结果是 5,说明成功获取到了 RDD 数据集的元素个数。值得一提的是,可以将第一行代码分解成下面两行代码,具体如下:

```
val arr =Array(1,2,3,4,5)
val arrRdd =sc.parallelize(arr)
```

2. first()

first()主要用于返回数组的第一个元素。现有一个 arrRdd,如果要获取 arrRdd 中第一个元素,示例代码如下:

```
scala>val arrRdd=sc.parallelize(Array(1,2,3,4,5))
arrRdd: org.apache.spark.rdd.RDD[Int]=
                 ParallelcollectionRDD[0] at parallelize at <console>:24
scala>arrRdd.first()
res1: Int =1
```

从上述结果可以看出,当执行 arrRdd. first()操作后返回的结果是 1,说明成功获取到了 RDD 数据集的第 1 个元素。

3. take(n)

take()主要用于以数组的形式返回数组集中的前 n 个元素。现有一个 arrRdd,如果要获取 arrRdd 中的前 3 个元素,示例代码如下:

```
scala>val arrRdd =sc.parallelize(Array(1,2,3,4,5))
arrRdd: org.apache.spark.rdd.RDD[Int]=
                 ParallelcollectionRDD[0] at parallelizeat <console>:24
scala>arrRdd.take(3)
res2: Array[Int]=Array(1,2,3)
```

从上述代码可以看出,执行 arrRdd. take(3)操作后返回的结果是 Array(1,2,3),说明成功获取到了 RDD 数据集的前 3 个元素。

4．reduce(func)

reduce()主要用于通过函数 func(输入两个参数并返回一个值)聚合数据集中的元素。现有一个 arrRdd，如果要对 arrRdd 中的元素进行聚合，示例代码如下：

```
scala>val arrRdd =sc.parallelize(Array(1,2,3,4,5))
arrRdd: org.apache.spark.rdd.RDD[Int]=
                   ParallelcollectionRDD[0] at parallelize at <console>:24
scala>arrRdd.reduce((a,b)=>a+b)
res3: Int =15
```

在上述代码中，执行 arrRdd.reduce((a,b)=>a+b)操作后返回的结果是 15，说明成功的将 RDD 数据集中的所有元素进行求和，结果为 15。

5．collect()

collect()主要用于以数组的形式返回数据集中的所有元素。现有一个 arrRdd，如果希望 arrRdd 中的元素以数组的形式输出，示例代码如下：

```
scala>val arrRdd =sc.parallelize(Array(1,2,3,4,5))
arrRdd: org.apache.spark.rdd.RDD[Int]=
                   ParallelcollectionRDD[0] at parallelize at <console>:24
scala>arrRdd.collect()
res4: Array[Int] =Array(1,2,3,4,5)
```

在上述代码中，执行 arrRdd.collect()操作后返回的结果是 Array(1,2,3,4,5)，说明成功地将 RDD 数据集中的元素以数组的形式输出。

6．foreach(func)

foreach()主要用于将数据集中的每个元素传递到函数 func 中运行。现有一个 arrRdd，如果希望遍历输出 arrRdd 中的元素，示例代码如下：

```
scala>val arrRdd =sc.parallelize(Array(1,2,3,4,5))
arrRdd: org.apache.spark.rdd.RDD[Int]=
                   ParallelcollectionRDD[0] at parallelize at <console>:24
scala>arrRdd.foreach(x =>println(x))
1
2
3
4
5
```

在上述代码中，foreach(x => println(x))的含义是依次遍历 arrRdd 中的每一个元素，把当前遍历的元素赋值给变量 x，并且通过 println(x)打印出 x 的值。执行 arrRdd.foreach()操作后，arrRdd 中的元素被依次输出了(即 RDD 数据集中所有的元素被遍历输出)。这里的 arrRdd.foreach(x => println(x))可以简写为 arrRdd.foreach(println)。

3.3.3　编写 WordCount 词频统计案例

在 Linux 本地系统的/export/data 目录下，有一个 test. txt 文件，文件里面有多行文本，每行文本都是由 2 个单词构成，并且单词之间都是用空格分隔。接下来需要通过 RDD 统计每个单词出现的次数(即词频)，具体操作过程如图 3-7 所示。

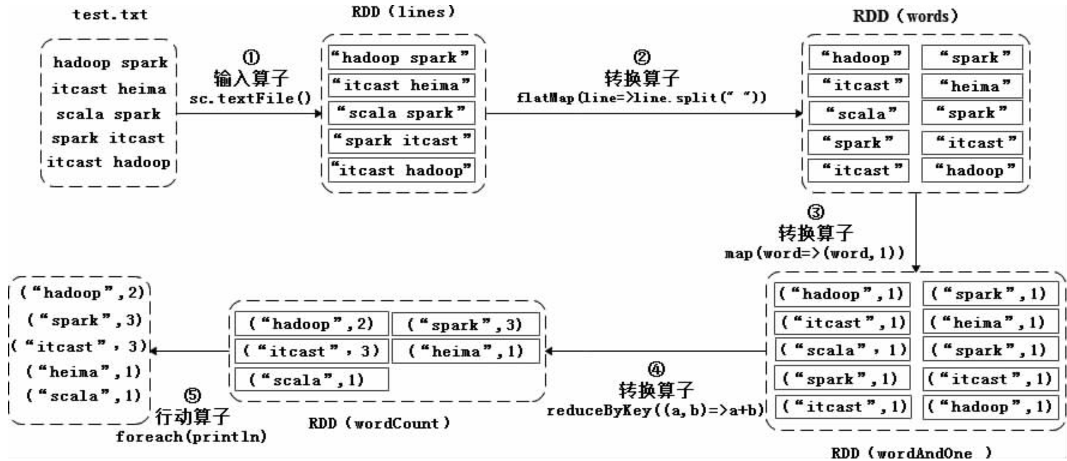

图 3-7　词频统计的操作

在图 3-7 中，Spark 通过输入算子的操作读取文件来创建 RDD，然后通过转换算子和行动算子操作将文件中的所有单词进行了词频统计。接下来，通过代码来进行演示，具体代码如下：

```
scala>val lines =sc.textFile("file:///export/data/test.txt")
lines: org.apache.spark.rdd.RDD[String] =file:///export/data/test.txt
                  MapPartitionsRDD[8] at textFile at <console>:24
scala>val words=lines.flatMap(line=>line.split(" "))
words: org.apache.spark.rdd.RDD[String] =MapPartitionsRDD[20] at flatMap
                                                       at <console>:25
scala>val wordAndOne =words.map(word=>(word,1))
wordAndOne: org.apache.spark.rdd.RDD[(String, Int)] =MapPartitionsRDD[21]
                                                  at map at <console>:25
scala>val wordCount =wordAndOne.reduceByKey((a,b)=>a+b)
wordCount: org.apache.spark.rdd.RDD[(String, Int)] =ShuffledRDD[22] at
                                            reduceByKey at <console>:25
scala>wordCount.foreach(println)
(spark,3)
(hadoop,2)
(scala,1)
(itcast,3)
(heima,1)
```

上述代码中，执行 wordCount. foreach(println)操作后返回的结果是(spark，3)、(hadoop，2)、(scala，1)、(itcast，3)、(heima，1)，说明已经实现了对文件 test. txt 的词频统计

操作。

3.4 RDD 的分区

在分布式程序中，网络通信的开销是很大的，因此控制数据分布以获得最少的网络传输开销可以极大地提升整体性能，Spark 程序可以通过控制 RDD 分区方式来减少通信开销。Spark 中所有的 RDD 都可以进行分区，系统会根据一个针对键的函数对元素进行分区。虽然 Spark 不能控制每个键具体划分到哪个节点上，但是可以确保相同的键出现在同一个分区上。

RDD 的分区原则是分区的个数尽量等于集群中的 CPU 核心（Core）数目。对于不同的 Spark 部署模式而言，都可以通过设置 spark.default.parallelism 这个参数值来配置默认的分区数目。一般而言，各种模式下的默认分区数目如下。

（1）Local 模式：默认为本地机器的 CPU 数目，若设置了 local[*N*]，则默认为 *N*。

（2）Standalone 或者 Yarn 模式：在“集群中所有 CPU 核数总和”和“2”这两者中取较大值作为默认值。

（3）Mesos 模式：默认的分区数是 8。

Spark 框架为 RDD 提供了两种分区方式，分别是哈希分区（HashPartitioner）和范围分区（RangePartitioner）。其中，哈希分区是根据哈希值进行分区；范围分区是将一定范围的数据映射到一个分区中。这两种分区方式已经可以满足大多数应用场景的需求。与此同时，Spark 也支持自定义分区方式，即通过一个自定义的 Partitioner 对象来控制 RDD 的分区，从而进一步减少通信开销。需要注意的是，RDD 的分区函数是针对（Key，Value）类型的 RDD，分区函数根据 Key 对 RDD 元素进行分区。因此，当需要对一些非（Key，Value）类型的 RDD 进行自定义分区时，需要先把 RDD 元素转换为（Key，Value）类型，再通过分区函数进行分区操作。

如果想要实现自定义分区，就需要定义一个类，使得这个自定义的类继承 org.apache.spark.Partitioner 类，并实现其中的 3 个方法，具体如下。

（1）def numPartitions:Int：用于返回创建的分区个数。

（2）def getPartition(Key:Any)：用于对输入的 Key 做处理，并返回该 Key 的分区 ID，分区 ID 的范围是 0～numPartitions－1。

（3）equals(other:Any)：用于 Spark 判断自定义的 Partitioner 对象和其他的 Partitioner 对象是否相同，从而判断两个 RDD 的分区方式是否相同。其中，equals()方法中的参数 other 表示其他的 Partitioner 对象，该方法的返回值是一个 Boolean 类型，当返回值为 true 时表示自定义的 Partitioner 对象和其他 Partitioner 对象相同，则两个 RDD 的分区方式也是相同的；反之，自定义的 Partitioner 对象和其他 Partitioner 对象不相同，则两个 RDD 的分区方式也不相同。

3.5 RDD 的依赖关系

在 Spark 中，不同的 RDD 之间具有依赖的关系。RDD 与它所依赖的 RDD 的依赖关系有两种类型，分别是窄依赖（narrow dependency）和宽依赖（wide dependency）。

窄依赖是指父 RDD 的每一个分区最多被一个子 RDD 的分区使用，即 OneToOneDependencies。窄依赖的表现一般分为两类：第一类表现为一个父 RDD 的分区对应于一个子 RDD 的分区；第二类表现为多个父 RDD 的分区对应于一个子 RDD 的分区。也就是说，一个父 RDD 的一个分区不可能对应一个子 RDD 的多个分区。为了便于理解，通常把窄依赖形象地比喻为独生子女。当 RDD 执行 map、filter、union 和 join 操作时，都会产生窄依赖，如图 3-8 所示。

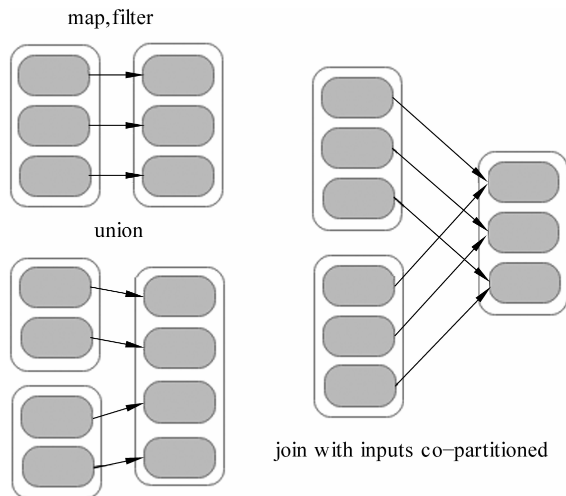

图 3-8 窄依赖

从图 3-8 可以看出，RDD 进行 map、filter 和 union 算子操作时，是属于窄依赖的第一类表现；而 RDD 进行 join 算子操作(对输入进行协同划分)时，是属于窄依赖表现的第二类。这里的输入协同划分是指多个父 RDD 的某一个分区的所有 Key，被划分到子 RDD 的同一分区，而不是指同一个父 RDD 的某一个分区，被划分到子 RDD 的两个分区中。当子 RDD 进行算子操作，因为某个分区操作失败导致数据丢失时，只需要重新对父 RDD 中对应的分区(与子 RDD 相对应的分区)进行算子操作即可恢复数据。

宽依赖是指子 RDD 的每一个分区都会使用所有父 RDD 的所有分区或多个分区，即 OneToManyDependecies。为了便于理解，通常把宽依赖形象地比喻为超生。当 RDD 进行 groupByKey 和 join 操作时，会产生宽依赖，如图 3-9 所示。

从图 3-9 可以看出，父 RDD 进行 groupByKey 和 join(输入未协同划分)算子操作时，子 RDD 的每一个分区都会依赖于所有父 RDD 的所有分区。当子 RDD 进行算子操作，因为某个分区操作失败导致数据丢失时，则需要重新对父 RDD 中的所有分区进行算子操作才能恢复数据。

需要注意的是，join 算子操作既可以属于窄依赖，也可以属于宽依赖。当 join 算子操作后，分区数量没有变化则为窄依赖(如 join with inputs co-partitioned，输入协同划分)；当 join 算子操作后，分区数量发生变化则为宽依赖(如 join with inputs not co-partitioned，输入非协同划分)。

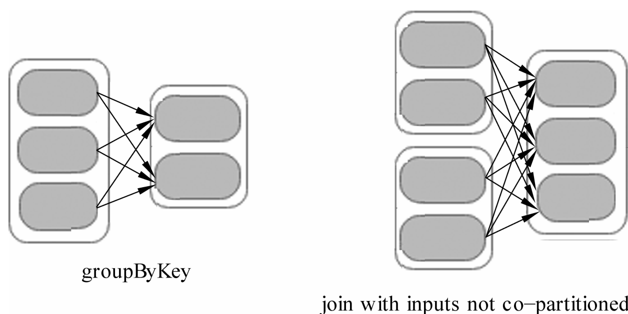

图 3-9　宽依赖

3.6　RDD 机制

Spark 为 RDD 提供了两个重要的机制，分别是持久化机制（即缓存机制）和容错机制。接下来，本节将针对持久化机制和容错机制进行详细介绍。

3.6.1　持久化机制

在 Spark 中，RDD 是采用惰性求值，即每次调用行动算子操作，都会从头开始计算。然而，每次调用行动算子操作，都会触发一次从头开始的计算，这对于迭代计算来说，代价是很大的，因为迭代计算经常需要多次重复地使用同一组数据集，所以，为了避免重复计算的开销，可以让 Spark 对数据集进行持久化。

通常情况下，一个 RDD 是由多个分区组成的，RDD 中的数据分布在多个节点中，因此，当持久化某个 RDD 时，每一个节点都将把计算分区的结果保存在内存中，若对该 RDD 或衍生出的 RDD 进行其他行动算子操作时，则不需要重新计算，直接去取各个分区保存的数据即可，这使得后续的行动算子操作速度更快（通常超过 10 倍），并且缓存是 Spark 构建迭代式算法和快速交互式查询的关键。

RDD 的持久化操作有两种方法，分别是 cache()方法和 persist()方法。每一个持久化的 RDD 都可以使用不同的存储级别存储，从而允许持久化数据集在硬盘或者内存中作为序列化的 Java 对象存储，甚至可以跨节点复制。

persist()方法的存储级别是通过 StorageLevel 对象（Scala、Java、Python）设置的。

cache()方法的存储级别是使用默认的存储级别（即 StorageLevel.MEMORY_ONLY（将反序列化的对象存入内存））。接下来，通过表 3-3 介绍持久化 RDD 的存储级别。

表 3-3　持久化 RDD 的存储级别

存储级别	相关说明
MEMORY_ONLY	默认存储级别。将 RDD 作为反序列化的 Java 对象，缓存到 JVM 中，若内存放不下（内存已满情况），则某些分区将不会被缓存，并且每次需要时都会重新计算
MEMORY_AND_DISK	将 RDD 作为反序列化的 Java 对象，缓存到 JVM 中，若内存放不下（内存已满情况），则将剩余分区存储到磁盘上，并在需要时从磁盘读取

续表

存储级别	相关说明
MEMORY_ONLY_SER	将 RDD 作为序列化的 Java 对象(每个分区序列化为一个字节数组),比反序列化的 Java 对象节省空间,但读取时,更占 CPU
MEMORY_AND_DISK_SER	与 MEMORY_ONLY_SER 类似,但是当内存放不下时则溢出到磁盘,而不是每次需要时重新计算它们
DISK_ONLY	仅将 RDD 分区全部存储到磁盘上
MEMORY_ONLY_2 MEMORY_AND_DISK_2	与上面的级别相同。若加上后缀_2,代表的是将每个持久化的数据都复制一份副本,并将副本保存到其他节点上
OFF_HEAP(实验性)	与 MEMORY_ONLY_SER 类似,但将数据存储在堆外内存中(这需要启用堆外内存)

在表 3-3 中,列举了持久化 RDD 的存储级别,可以在 RDD 进行第一次算子操作时,根据自己的需求选择对应的存储级别。

为了大家更好地理解,接下来,通过代码演示如何使用 persist()方法和 cache()方法对 RDD 进行持久化。

1. 使用 persist()方法对 RDD 进行持久化

定义一个列表 list,通过该列表创建一个 RDD,然后通过 persist 持久化操作和算子操作统计 RDD 中的元素个数以及打印输出 RDD 中的所有元素。具体代码如下:

```
1   scala>import org.apache.spark.storage.StorageLevel
2   import org.apache.spark.storage.StorageLevel
3   scala>val list =List("hadoop","spark","hive")
4   list: List[String] =List(hadoop, spark, hive)
5   scala>val listRDD =sc.parallelize(list)
6   listRDD: org.apache.spark.rdd.RDD[String] =ParallelCollectionRDD[0] at
7                                                parallelize at <console>:27
8   scala>listRDD.persist(StorageLevel.DISK_ONLY)
9   res1: listRDD.type =ParallelCollectionRDD[0] at parallelize at <console>:27
10  scala>println(listRDD.count())
11  3
12  scala>println(listRDD.collect().mkString(","))
13  hadoop,spark,hive
```

上述代码中,第 1 行代码导入 StorageLevel 对象的包;第 3 行代码定义了一个列表 list;第 5 行代码执行 sc. parallelize(list)操作,创建了一个 RDD,即 listRDD;第 8 行代码添加了 persist()方法,用于持久化 RDD,减少 I/O 操作,提高计算效率;第 10 行代码执行 listRDD. count()行动算子操作,将统计 listRDD 中元素的个数;第 12 行代码执行 listRDD. collect()行动算子操作和 mkString(",")操作,将 listRDD 中的所有元素进行打印输出,并且以逗号为分隔符。

需要注意的是,当程序执行到第 8 行代码时,并不会持久化 listRDD,因为 listRDD 还没有被真正计算;当执行第 10 行代码时,listRDD 才会进行第一次行动算子操作,触发真正的从头到尾的计算,这时 listRDD. persist()方法才会被真正执行,把 listRDD 持久化到磁盘

中；当执行到第 12 行代码时，进行第二次行动算子操作，但不触发从头到尾的计算，只需使用已经进行持久化的 listRDD 来进行计算。

2. 使用 cache()方法对 RDD 进行持久化

定义一个列表 list，通过该列表创建一个 RDD，然后通过 cache 持久化操作和算子操作统计 RDD 中的元素个数以及打印输出 RDD 中的所有元素。具体代码如下：

```
1   scala>val list=List("hadoop","spark","hive")
2   list: List[String] =List(hadoop, spark, hive)
3   scala>val listRDD=sc.parallelize(list)
4   listRDD: org.apache.spark.rdd.RDD[String] =ParallelCollectionRDD[0] at
5                                              parallelize at <console>:26
6   scala>listRDD.cache()
7   res2: listRDD.type =ParallelCollectionRDD[1] at parallelize at <console>:26
8   scala>println(listRDD.count())
9   3
10  scala>println(listRDD.collect().mkString(","))
11  hadoop,spark,hive
```

上述代码中，第 6 行代码对 listRDD 进行持久化操作，即添加 cache()方法，用于持久化 RDD，减少 I/O 操作，提高计算效率。然而，使用 cache()方法进行持久化操作，底层是调用了 persist(MEMORY_ONLY)方法，用来对 RDD 进行持久化。当程序执行到第 6 行代码时，并不会持久化 listRDD，因为 listRDD 还没有被真正计算；当程序执行第 8 行代码时，listRDD 才会进行第一次行动算子操作，触发真正的从头到尾的计算，这时 listRDD. cache()方法才会被真正执行，把 listRDD 持久化到内存中；当程序执行到第 10 行代码时，进行第二次行动算子操作，但不触发从头到尾的计算，只需使用已经持久化的 listRDD 来进行计算。

3.6.2　容错机制

当 Spark 集群中的某一个节点由于宕机导致数据丢失，可以通过 Spark 中的 RDD 容错机制恢复已经丢失的数据。RDD 提供了两种故障恢复的方式，分别是血统(lineage)方式和设置检查点(checkpoint)方式。下面就来介绍这两种方式。

血统方式，主要是根据 RDD 之间的依赖关系对丢失数据的 RDD 进行数据恢复。如果丢失数据的子 RDD 在进行窄依赖运算，则只需要把丢失数据的父 RDD 的对应分区进行重新计算即可，不需要依赖其他的节点，并且在计算过程中不会存在冗余计算；若丢失数据的子 RDD 进行宽依赖运算，则需要父 RDD 的所有分区都要进行从头到尾的计算，在计算过程中会存在冗余计算。为了解决宽依赖运算中出现的计算冗余问题，Spark 又提供了另一种方式进行数据容错，即设置检查点方式。

设置检查点方式，本质上是将 RDD 写入磁盘进行存储。当 RDD 在进行宽依赖运算时，只需要在中间阶段设置一个检查点进行容错，即通过 Spark 中的 sparkContext 对象调用 setCheckpoint()方法，设置一个容错文件系统目录(如 HDFS)作为检查点 checkpoint，将 checkpoint 的数据写入之前设置的容错文件系统中进行高可用的持久化存储，若是后面有节点出现宕机导致分区数据丢失，则可以从作为检查点的 RDD 开始重新计算，不需要进行

从头到尾的计算,这样就会减少开销。

3.7 Spark 的任务调度

3.7.1 DAG 的概念

DAG(Directed Acyclic Graph,有向无环图),Spark 中的 RDD 通过一系列的转换算子操作和行动算子操作形成了一个 DAG。DAG 是一种非常重要的图论数据结构。如果一个有向图无法从任意顶点出发经过若干条边回到该点,则这个图就是有向无环图,具体如图 3-10 所示。

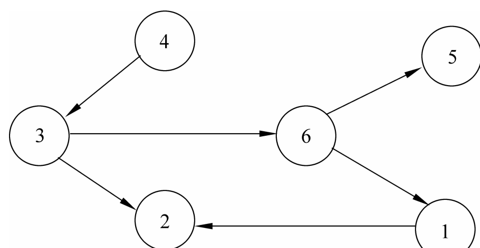

图 3-10 DAG 有向无环图

从图 3-10 可以看出,4→6→1→2 是一条路径,4→6→5 也是一条路径,并且图中不存在从顶点经过若干条边后能回到该点的路径。在 Spark 中,有向无环图的连贯关系被用来表达 RDD 之间的依赖关系。其中,顶点表示 RDD 及产生该 RDD 的操作算子,有方向的边表示算子之间的相互转化。

根据 RDD 之间依赖关系的不同可以将 DAG 划分成不同的 Stage(调度阶段)。对于窄依赖来说,RDD 分区的转换处理是在一个线程里完成的,所以窄依赖会被 Spark 划分到同一个 Stage 中;而对于宽依赖来说,由于有 Shuffle 的存在,所以只能在父 RDD 处理完成后,下一个 Stage 才能开始接下来的计算,因此宽依赖是划分 Stage 的依据,当 RDD 进行转换操作,遇到宽依赖类型的转换操作时,就划为一个 Stage。Stage 的具体划分如图 3-11 所示。

在图 3-11 中,创建了 3 个 RDD 的实例 A、C 以及 E。当 RDD 的实例 A 进行 groupByKey 转换操作生成 B 时,由于 groupByKey 转换操作属于宽依赖类型,所以就把实例 A 划分为一个 Stage,如 Stage1;当实例 C 进行 map 转换操作生成 D, D 与实例 E 进行 union 转换操作生成 F 时,由于 map 和 union 转换操作都属于窄依赖类型,因此不进行 Stage 的划分,而是将 C、D、E、F 加入到同一个 Stage 中;当 F 与 B 进行 join 转换操作时,由于这时的 join 操作是非协同划分,所以属于宽依赖,因此会划分为一个 Stage,如 Stage2;剩下的 B 和 G 被划分为一个 Stage,如 Stage3。

3.7.2 RDD 在 Spark 中的运行流程

下面,通过图 3-12 来学习 RDD 在 Spark 中的运行流程。

在图 3-12 中,Spark 的任务调度流程分为 RDD Objects、DAGScheduler、TaskScheduler 以及 Worker 4 个部分。关于这 4 个部分的介绍具体如下。

(1) RDD Objects:当 RDD 对象创建后,SparkContext 会根据 RDD 对象构建 DAG 有向无环图,然后将 Task 提交给 DAGScheduler。

(2) DAGScheduler:将作业的 DAG 划分成不同的 Stage,每个 Stage 都是 TaskSet 任务集合,并以 TaskSet 为单位提交给 TaskScheduler。

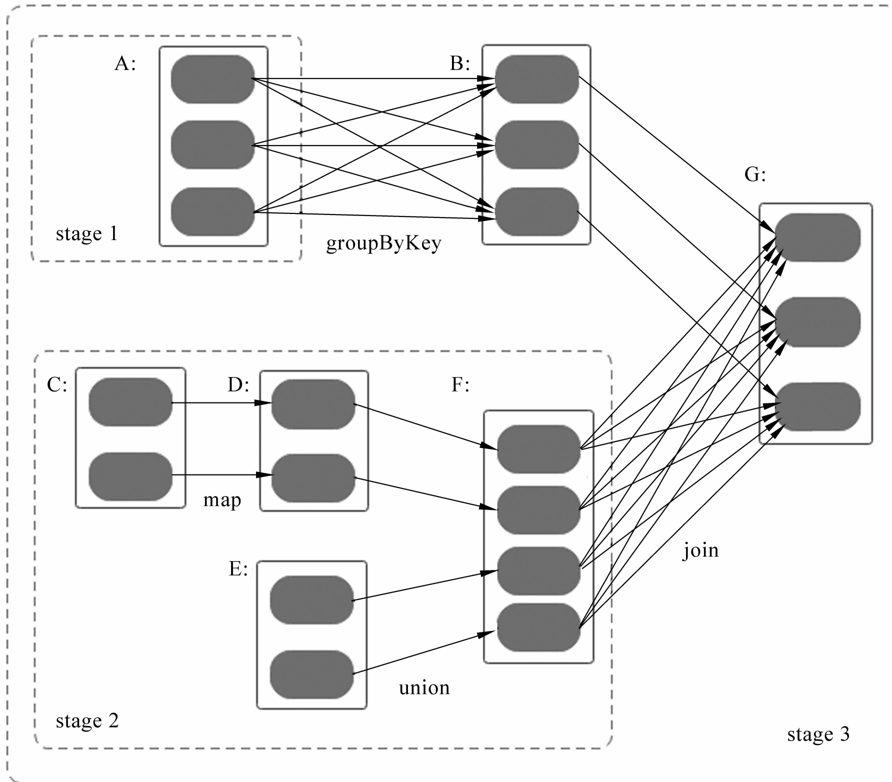

图 3-11　Stage 的划分

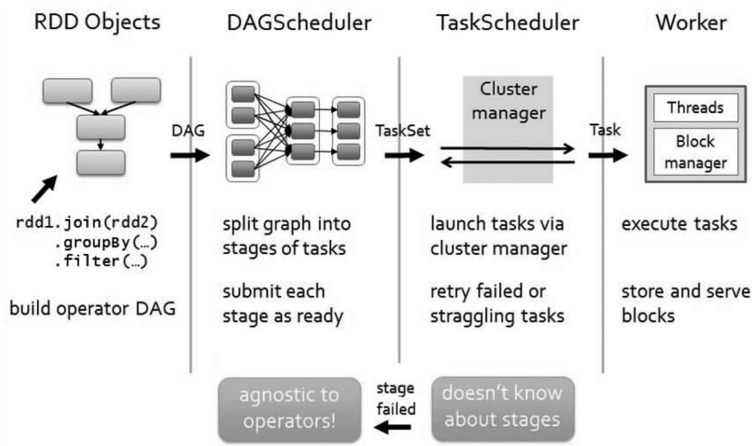

图 3-12　RDD 在 Spark 中的运行流程

(3) TaskScheduler：通过 TaskSetManager 管理 Task，并通过集群中的资源管理器(Standalone 模式下是 Master，Yarn 模式下是 ResourceManager)把 Task 发给集群中 Worker 的 Executor。若期间有某个 Task 失败，则 TaskScheduler 会重试；若 TaskScheduler 发现某个 Task 一直没有运行完成，则有可能在空闲的机器上启动同一个 Task，哪个 Task 先完成就用哪个 Task 的结果。但是，无论 Task 是否成功，TaskScheduler 都会向 DAGScheduler 汇报当前的

状态，若某个 Stage 运行失败，则 TaskScheduler 会通知 DAGScheduler 重新提交 Task。需要注意的是，一个 TaskScheduler 只能服务一个 SparkContext 对象。

（4）Worker：Spark 集群中的 Worker 接收到 Task 后，把 Task 运行在 Executor 进程中，这个 Task 就相当于 Executor 进程中的一个线程。一个进程中可以有多个线程在工作，从而可以处理多个数据分区（如运行任务、读取或者存储数据）。

3.8 本章小结

本章主要介绍 RDD 及 RDD 编程的相关知识，包括 RDD 创建、RDD 的处理、RDD 的分区、RDD 的依赖关系、RDD 的容错机制以及 Spark 的任务调度。希望读者通过本章的学习，可以掌握 RDD 编程，因为掌握了 RDD，可以帮助读者更好地使用 Spark 框架解决实际应用中的数据分析问题。

3.9 课后习题

一、填空题

1. RDD 是________的一个抽象概念，也是一个________、并行的数据结构。
2. RDD 的操作主要分为 ________和________。
3. RDD 的依赖关系有 ________和________。
4. RDD 的分区方式有 ________________和________。
5. RDD 的容错方式有________和________。

二、判断题

1. RDD 是一个可变、不可分区、里面的元素是可并行计算的集合。（　　）
2. RDD 采用了惰性调用，即在 RDD 的处理过程中，真正的计算发生在 RDD 的“行动”操作。（　　）
3. 宽依赖是指每一个父 RDD 的 Partition（分区）最多被子 RDD 的一个 Partition 使用。（　　）
4. 如果一个有向图可以从任意顶点出发经过若干条边回到该点，则这个图就是有向无环图。（　　）
5. 窄依赖是划分 Stage 的依据。（　　）

三、选择题

1. 下列方法中，用于创建 RDD 的方法是（　　）。

   A. makeRDD()　　B. parallelize()　　C. textFile()　　D. testFile()

2. 下列选项中，哪个不属于转换算子操作？（　　）

   A. filter(func)　　B. map(func)

   C. reduce(func)　　D. reduceByKey(func)

3. 下列选项中,能使 RDD 产生宽依赖的是(　　)。
   A. map(func)　　B. filter(func)　　C. union　　D. groupByKey()

四、简答题

1. 简述 RDD 提供的两种故障恢复方法。
2. 简述如何在 Spark 中划分 Stage。

五、编程题

通过 Spark 的 RDD 编程,实现词频统计的功能。
提示:对文件 test.txt(内容如文件 3-1 所示)进行词频统计。