第 3 章

Spark RDD弹性分布式数据集

学习目标：

- 了解 RDD，能够从不同方面介绍 RDD。
- 掌握 RDD 的创建，能够基于文件和数据集合创建 RDD。
- 掌握 RDD 的处理过程，能够使用转换算子和行动算子操作 RDD。
- 熟悉 RDD 的分区，能够指定 RDD 的分区数量。
- 熟悉 RDD 的依赖关系，能够区分 RDD 的窄依赖和宽依赖。
- 掌握 RDD 持久化机制，能够使用 persist()方法和 cache()方法持久化 RDD。
- 熟悉 RDD 容错机制，能够叙述 RDD 的故障恢复方式。
- 熟悉 DAG 的概念，能够叙述什么是 DAG。
- 掌握 RDD 在 Spark 中的运行流程，能够说出 RDD 被解析为 Task 执行的过程。

MapReduce 具有负载平衡、容错性高和可拓展性强的优点，但在进行迭代计算时要频繁进行磁盘读写操作，从而导致执行效率较低。相比之下，Spark 中的 RDD（Resilient Distributed Dataset，弹性分布式数据集）可以有效解决这一问题。RDD 是 Spark 提供的重要抽象概念，可将其理解为存储在 Spark 集群中的大型数据集。不同 RDD 之间可以通过转换操作建立依赖关系，并实现管道化的数据处理，避免中间结果的磁盘读写操作，从而提高了数据处理的速度和性能。接下来，本章针对 Spark RDD 进行详细讲解。

3.1 RDD 简介

RDD 是 Spark 中的基本数据处理模型，具有可容错性和并行的数据结构。RDD 不仅可以将数据存储到磁盘中，还可以将数据存储到内存中。对于迭代计算产生的中间结果，RDD 可以将其保存到内存中。如果后续计算需要使用这些中间结果，可以直接从内存中读取，提高数据计算的速度。

下面从 5 方面介绍 RDD。

1. 分区列表

每个 RDD 会被分为多个分区，这些分区分布在集群中的不同节点上，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任务的数量，因此分区数的合理设置对于并行计算性能至关重要。在创建 RDD 时，可以指定 RDD 分区的数量。如果没有指定分区数量，会根据不同的情况采用默认的分区策略。例如，根据数据集合创建 RDD 时，默认分区

数量为分配给程序的 CPU 核心数；而根据 HDFS 上的文件创建 RDD 时，默认分区数量为文件的分块数。

2. 计算函数

Spark 中的计算函数可以对 RDD 的每个分区进行迭代计算，用户可以根据具体需求自定义 RDD 中每个分区的数据处理逻辑。这种灵活性使得 Spark 能够适应各种数据处理场景。

3. 依赖关系

RDD 之间存在依赖关系，即每次对 RDD 进行转换操作都会生成一个新的 RDD。这种依赖关系在数据计算中发挥着重要作用。例如，如果某个分区的数据丢失，通过依赖关系，丢失的数据可以被重新计算和恢复，从而保证了数据计算的可靠性和容错性。

4. 分区器

当 Spark 读取的数据为键值对(key-value pair)类型的数据时，可以通过设置分区器来自定义数据的分区方式。Spark 提供了两种类型的分区器，一种是基于哈希值的分区器 HashPartitioner，另一种是基于范围的分区器 RangePartitioner。在读取的数据不是键值对类型的情况下，分区值为 None，这时 Spark 会采取默认的分区策略来处理这些非键值对数据。

5. 优先位置列表

优先位置列表通过存储每个分区中数据块的位置，帮助 Spark 优化数据处理性能。在进行数据计算时，Spark 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。这种做法遵循了“移动数据不如移动计算”的理念，即在可能的情况下，将计算任务移动到数据所在的位置，而不是将数据移动到计算任务所在的位置。通过这种方式，Spark 可以减少数据传输开销，从而提高整体计算效率。

3.2　RDD 的创建

Spark 提供了两种创建 RDD 的方式，分别是基于文件和基于数据集合。使用基于文件的方式创建 RDD 时，文件中的每行数据会被视为 RDD 的一个元素。使用基于数据集合的方式创建 RDD 时，数据集合中的每个元素会被视为 RDD 的一个元素。本节针对这两种创建 RDD 的方式进行详细讲解。

3.2.1　基于文件创建 RDD

Spark 提供了 textFile()方法，用于从文件系统中的文件读取数据并创建 RDD，包括本地文件系统、HDFS、Amazon S3 等，其语法格式如下。

```
sc.textFile(path)
```

上述语法格式中，sc 为 SparkContext 对象，path 用于指定文件的路径。

接下来，分别演示从本地文件系统和 HDFS 中的文件读取数据并创建 RDD。

1. 从本地文件系统中的文件读取数据并创建 RDD

在虚拟机 Hadoop1 的/export/data 目录执行 vi rdd.txt 命令创建文件 rdd.txt，具体内

容如文件 3-1 所示。

文件 3-1 rdd.txt

```
hadoop spark
itcast heima
scala spark
spark itcast
itcast hadoop
```

确保 Hadoop 集群已经成功启动后,进入虚拟机 Hadoop1 的目录/export/servers/sparkOnYarn/spark-3.3.0-bin-hadoop3/启动 Spark Shell,在 Spark Shell 中执行如下代码。

```
scala> val test = sc.textFile("file:///export/data/rdd.txt")
```

上述代码使用 SparkContext 对象 sc 的 textFile()方法,从本地文件系统中的文件 rdd.txt 读取数据创建 RDD,并将 RDD 保存到常量 test 中,该常量是一个 RDD 对象。

上述代码执行完成后,如图 3-1 所示。

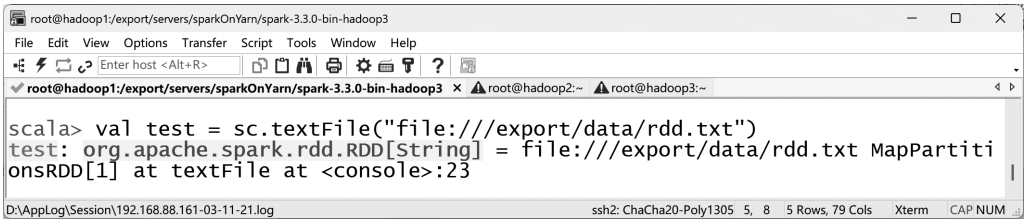

图 3-1 从本地文件系统中的文件读取数据并创建 RDD

从图 3-1 可以看出,Spark 从本地文件系统的目录/export/data 中读取文件 rdd.txt 的数据,并创建一个名为 test 的 RDD,其数据类型为 String。这意味着,文件 rdd.txt 中的每一行数据都会作为 String 类型的元素存储在 RDD 中。

2. 从 HDFS 中的文件读取数据并创建 RDD

将文件 rdd.txt 上传到 HDFS 的根目录。在虚拟机 Hadoop1 的/export/data 目录执行如下命令。

```
$ hdfs dfs -put rdd.txt /
```

从 HDFS 中的文件 rdd.txt 读取数据并创建 RDD,在 Spark Shell 中执行如下代码。

```
scala> val testRDD = sc.textFile("/rdd.txt")
```

上述代码执行完成后的效果如图 3-2 所示。

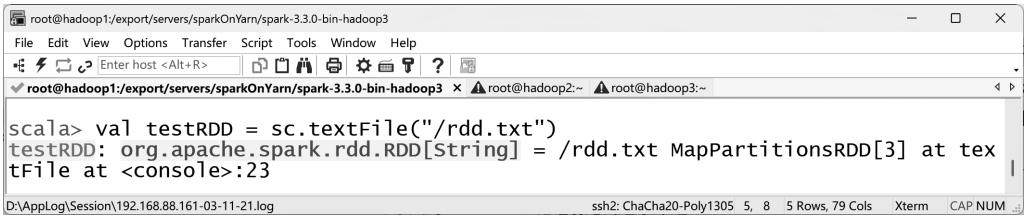

图 3-2 从 HDFS 中的文件读取数据并创建 RDD

从图 3-2 可以看出，Spark 从 HDFS 的根目录中读取文件 rdd.txt 的数据，并创建一个名为 testRDD 的 RDD，其数据类型为 String。

3.2.2　基于数据集合创建 RDD

Spark 提供了 parallelize()方法，用于从数据集合（数组、List 集合等）读取数据并创建 RDD，其语法格式如下。

```
sc.parallelize(seq, numSlices)
```

上述语法格式中，seq 用于指定数据集合。numSlices 为可选，用于指定创建 RDD 的分区数，该参数会在 3.4 节中讲解。

接下来，演示从数据集合读取数据并创建 RDD，在 Spark Shell 中执行如下代码。

```
scala> val numList = List[Int](1,2,3,4)
scala> val listRDD = sc.parallelize(numList)
```

上述代码中，首先创建一个数据类型为 Int 的 List 集合 numList，然后使用 SparkContext 对象 sc 的 parallelize()方法，从 List 集合 numList 中读取数据创建 RDD，并将 RDD 保存到常量 listRDD 中，该常量是一个 RDD 对象。

上述代码执行完成后，如图 3-3 所示。

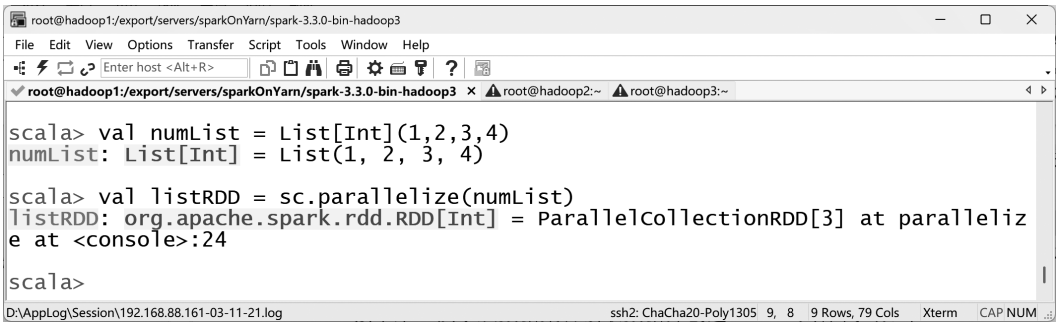

图 3-3　从 List 集合 numList 中读取数据创建 RDD

从图 3-3 可以看出，Spark 从 List 集合 numList 中读取数据，并创建一个名为 listRDD 的 RDD，其数据类型为 Int。

3.3　RDD 的处理过程

RDD 的处理过程主要包括转换和行动操作。下面通过图 3-4 来描述 RDD 的处理过程。

在图 3-4 中，RDD 经过一系列的转换操作，每一次转换操作都会生成一个新的 RDD，直到最后一个生成的 RDD 经过行动操作时，所有 RDD 才会触发实际计算，并将结果返回给驱动程序。如果某个 RDD 需要复用，则可以将其缓存到内存中。

Spark 针对转换操作和行动操作提供了对应的算子，即转换算子和行动算子。本节针对这两种算子进行详细讲解。

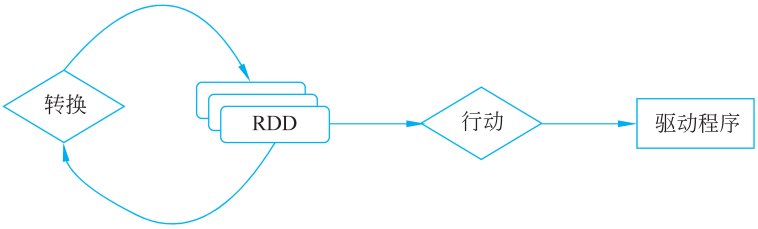

图 3-4 RDD 的处理过程

3.3.1 转换算子

转换算子用于将 RDD 转换为一个新的 RDD,但它们不会立即执行计算。相反,它们会构建一个执行计划,直到遇到行动算子时才会触发实际的计算。表 3-1 列举了一些常用的转换算子。

表 3-1 常用的转换算子

算 子	语 法 格 式	说 明
filter	RDD.filter(func)	根据给定的函数 func 筛选 RDD 中的元素
map	RDD.map(func)	对 RDD 中的每个元素应用函数 func,将其映射为一个新元素
flatMap	RDD.flatMap(func)	与 map 算子作用相似,但是每个输入的元素都可以映射为 0 或多个输出结果
groupByKey	RDD.groupByKey()	用于对键值对类型的 RDD 中具有相同键的元素进行分组
reduceByKey	RDD.reduceByKey(func)	用于对键值对类型的 RDD 中具有相同键的元素中的值应用函数 func 进行合并

下面针对表 3-1 列举的常用的转换算子进行详细讲解。

1. filter 算子

filter 算子通过对 RDD 中的每个元素应用一个函数来筛选数据,只留下满足指定条件的元素,而过滤掉不满足条件的元素。接下来,以文件 3-1 为例,通过一张图来描述如何通过 filter 算子筛选出文件 rdd.txt 中包含单词 spark 的元素,具体处理过程如图 3-5 所示。

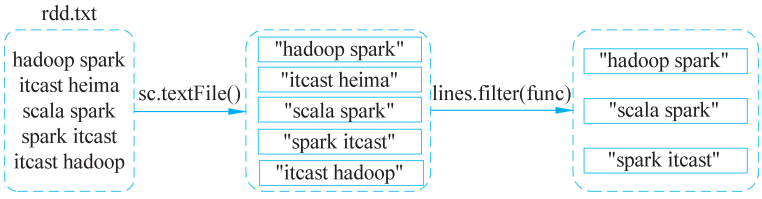

图 3-5 filter 算子处理过程

在图 3-5 中,通过从文件 rdd.txt 读取数据创建 RDD,然后通过 filter 算子将 RDD 的每个元素应用到函数 func 来筛选出包含单词 spark 的元素,并将其保留到新的 RDD 中。

接下来,以 IntelliJ IDEA 为例,演示如何使用 filter 算子,具体操作步骤如下。

(1) 在本地计算机的 D 盘根目录创建文件 rdd.txt,其内容与文件 3-1 一致。

(2) 在项目 Spark_Project 的目录/src/main/scala 下,新建一个名为 cn.itcast

.transformation 的包。在 cn.itcast.transformation 包中创建一个名为 FilterDemo 的 Scala 文件，用于筛选出文件 rdd.txt 中包含单词 spark 的行，具体代码如文件 3-2 所示。

文件 3-2　FilterDemo.scala

```
1  package cn.itcast.transformation
2  import org.apache.spark.{SparkConf, SparkContext}
3  object FilterDemo {
4    def main(args: Array[String]): Unit = {
5      //指定 Spark 程序的名称为 filter,并且可以使用本地计算机的线程数为 1
6      val conf:SparkConf = new SparkConf().setAppName("filter")
7        .setMaster("local[1]")
8      val sc:SparkContext = new SparkContext(conf)
9      val lines = sc.textFile("D:\\rdd.txt")
10     val result = lines.filter(x=>x.contains("spark"))
11     result.saveAsTextFile("D:\\Spark_out\\Filter_out")
12     //释放资源
13     sc.stop()
14   }
15 }
```

在文件 3-2 中，第 9 行代码用于从本地文件系统中读取文件 rdd.txt 的数据，并创建一个名为 lines 的 RDD。

第 10 行代码通过 filter 算子筛选出 lines 中包含 spark 的元素。在 filter 算子中，指定的函数是一个匿名函数，用于依次取出 lines 中的每个元素并赋值给变量 x，然后通过 contains()方法检查元素是否包含 spark。若包含，则将该元素存放到名为 result 的 RDD 中，否则，就过滤掉该元素。

第 11 行代码使用 saveAsTextFile()方法将 result 中的元素输出到指定目录的文件中，每个元素会占用文件的一行空间。

(3) 运行文件 3-2。当文件 3-2 运行完成后，查看本地计算机中目录 D:\\Spark_out\\Filter_out 的内容，如图 3-6 所示。

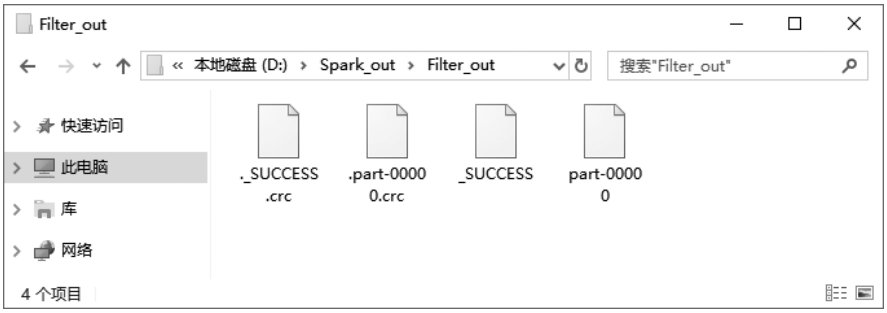

图 3-6　目录 D:\\Spark_out\\Filter_out 的内容

从图 3-6 可以看出，目录 D:\\Spark_out\\Filter_out 中包含 4 个文件，其中 part-00000 是存储 Spark 程序输出数据的文件。.part-00000.crc 是文件 part-00000 的校验和文件。_SUCCESS是标记 Spark 程序运行成功的文件，该文件的内容为空。_SUCCESS.crc 是文件_SUCCESS 的校验和文件。

(4) 通过记事本查看文件 part-00000 的内容,如图 3-7 所示。

图 3-7 查看文件 part-00000 的内容(1)

从图 3-7 可以看出,文件 part-00000 中的每一行数据都包含 spark。说明使用 filter 算子成功筛选出 RDD 中包含 spark 的元素。

【注意】 当 Spark 程序能够利用本地计算机的多个线程时,saveAsTextFile()方法可能会将 RDD 中的元素输出到指定目录的多个文件中。如果在允许 Spark 程序使用本地计算机的多个线程的情况下,希望 saveAsTextFile()方法将 RDD 中的元素输出到指定目录的单个文件中,可以通过将 RDD 的分区数指定为 1 来实现。关于这部分内容可以参考 3.4 节。

2. map 算子

map 算子可以将 RDD 中的每个元素通过一个函数映射为一个新元素。接下来,以文件 3-1 为例,通过一张图来描述如何通过 map 算子将文件 rdd.txt 中的每行数据拆分为单词后保存到数组中,具体处理过程如图 3-8 所示。

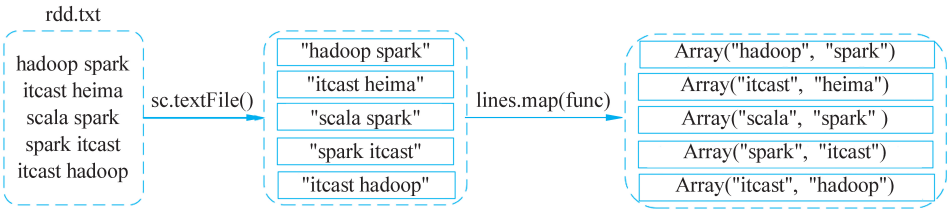

图 3-8 map 算子处理过程

在图 3-8 中,通过从文件 rdd.txt 读取数据创建 RDD,然后通过 map 算子将 RDD 的每个元素经函数 func 拆分为单词后保存到数组中,并将每个数组作为元素保留到新的 RDD 中。

接下来,以 IntelliJ IDEA 为例,演示如何使用 map 算子。在项目 Spark_Project 的 cn.itcast.transformation 包中创建一个名为 MapDemo 的 Scala 文件,用于将文件 rdd.txt 中的每行数据拆分为单词后保存到数组中,具体代码如文件 3-3 所示。

文件 3-3 MapDemo.scala

```
1  package cn.itcast.transformation
2  import org.apache.spark.{SparkConf, SparkContext}
3  object MapDemo {
4    def main(args: Array[String]): Unit = {
5      //指定 Spark 程序的名称为 map,并且可以使用本地计算机的线程数为 1
6      val conf:SparkConf = new SparkConf().setAppName("map")
7        .setMaster("local[1]")
8      val sc:SparkContext = new SparkContext(conf)
9      val lines = sc.textFile("D:\\rdd.txt")
10     val result = lines.map(x=>x.split(" ").toBuffer)
11     result.saveAsTextFile("D:\\Spark_out\\Map_out")
12     //释放资源
```

```
13      sc.stop()
14    }
15 }
```

在文件 3-3 中，第 9 行代码用于从本地文件系统中读取文件 rdd.txt 的数据，并创建一个名为 lines 的 RDD。

第 10 行代码通过 map 算子将 lines 中的每个元素映射为新的元素。在 map 算子中，指定的函数是一个匿名函数，用于依次取出 lines 中的每个元素并赋值给变量 x，然后通过 split()方法将每个元素按照分隔符“ ”(空格)拆分成单词，并将每个单词存放到数组中。为了便于在输出结果中查看数组的内容，这里通过 toBuffer()方法将数组转换为可变数组。这些可变数组最终将存放在名为 result 的 RDD 中。

第 11 行代码使用 saveAsTextFile()方法将 result 中的元素输出到指定目录的文件中，每个元素会占用文件的一行空间。

文件 3-3 运行完成后，在本地计算机的目录 D:\\Spark_out\\Map_out 中，通过记事本查看文件 part-00000 的内容，如图 3-9 所示。

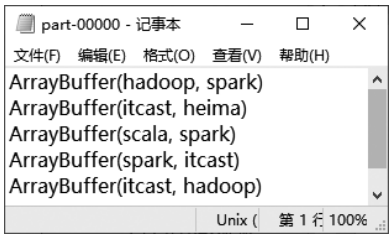

图 3-9　查看文件 part-00000 的内容(2)

从图 3-9 可以看出，文件 part-00000 中的每一行数据为可变数组的形式。可变数组中的每个元素为一个单词。说明使用 map 算子成功将 RDD 中的每个元素数据拆分为单词后保存到数组中。

3. flatMap 算子

flatMap 算子可以将 RDD 中的每个元素通过一个函数映射为一个或多个新元素。接下来，以文件 3-1 为例，通过一张图来描述如何通过 flatMap 算子将文件 rdd.txt 中的每行数据拆分为单词，具体处理过程如图 3-10 所示。

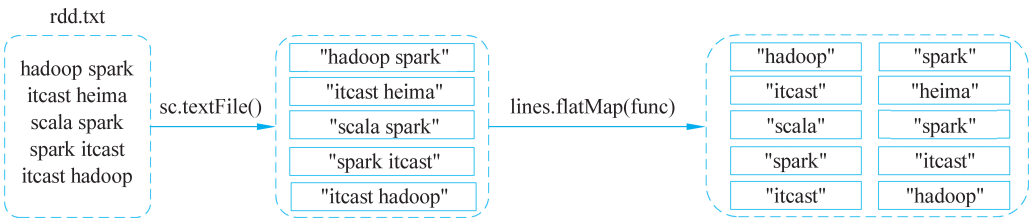

图 3-10　flatMap 算子的处理过程

在图 3-10 中，通过从文件 rdd.txt 读取数据创建 RDD，然后通过 flatMap 算子将 RDD 的每个元素通过函数 func 拆分为单词，并将每个单词作为元素保留到新的 RDD 中。

接下来，以 IntelliJ IDEA 为例，演示如何使用 flatMap 算子。在项目 Spark_Project 的 cn.itcast.transformation 包中创建一个名为 FlatMapDemo 的 Scala 文件，用于将文件 rdd.txt 中的每行数据拆分为单词，具体代码如文件 3-4 所示。

文件 3-4　FlatMapDemo.scala

```
1  package cn.itcast.transformation
2  import org.apache.spark.{SparkConf, SparkContext}
3  object FlatMapDemo {
```

```
4    def main(args: Array[String]): Unit = {
5      //指定 Spark 程序的名称为 flatMap,并且可以使用本地计算机的线程数为 1
6      val conf:SparkConf = new SparkConf().setAppName("flatMap")
7        .setMaster("local[1]")
8      val sc:SparkContext = new SparkContext(conf)
9      val lines = sc.textFile("D:\\rdd.txt")
10     val result = lines.flatMap(x=>x.split(" "))
11     result.saveAsTextFile("D:\\Spark_out\\FlatMap_out")
12     //释放资源
13     sc.stop()
14   }
15 }
```

在文件 3-4 中,第 9 行代码用于从本地文件系统中读取文件 rdd.txt 的数据,并创建一个名为 lines 的 RDD。

第 10 行代码通过 flatMap 算子将 lines 中的每个元素映射为多个新的元素。在 flatMap 算子中,指定的函数是一个匿名函数,用于依次取出 lines 中的每个元素并赋值给变量 x,然后通过 split()方法将每个元素按照分隔符“ ”拆分成单词,并将每个单词存放到数组中。flatMap 算子会对数组进行扁平化处理,将数组中的每个元素作为输出的一个元素存放在名为 result 的 RDD 中。

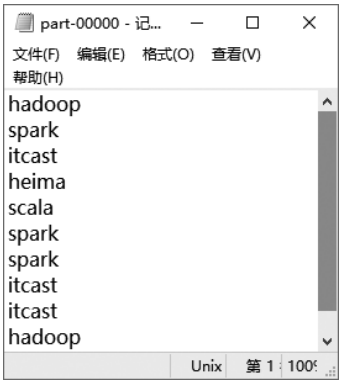

图 3-11 查看文件 part-00000 的内容(3)

第 11 行代码使用 saveAsTextFile()方法将 result 中的元素输出到指定目录的文件中,每个元素会占用文件的一行空间。

文件 3-4 运行完成后,在本地计算机的目录 D:\\Spark_out\\FlatMap_out 中,通过记事本查看文件 part-00000 的内容,如图 3-11 所示。

从图 3-11 可以看出,文件 part-00000 中的每一行数据为一个单词。说明使用 flatMap 算子成功将 RDD 中的每个元素数据拆分为单词。

4. groupByKey 算子

groupByKey 算子可以将 RDD 中具有相同键的元素划分到同一组中,返回一个新的 RDD。新的 RDD 中每个元素都是一个键值对,其中键是原始 RDD 中的键,而值则是一个迭代器,包含了原始 RDD 中具有相同键的所有值。接下来,以文件 3-1 为例,通过一张图来描述如何通过 groupByKey 算子将文件 rdd.txt 中的每个单词进行分组,具体处理过程如图 3-12 所示。

在图 3-12 中,首先,通过从文件 rdd.txt 读取数据创建 RDD。其次,通过 flatMap 算子将 RDD 的每个元素通过函数 func 拆分为单词。然后,通过 map 算子将 RDD 的每个元素通过函数 func 映射为键值对的形式,其中键为单词,值为 1 用于标识单词出现的次数。最后,通过 groupByKey 算子将相同键的元素划分到同一组中,返回一个新的 RDD。

接下来,以 IntelliJ IDEA 为例,演示如何使用 groupByKey 算子。在项目 Spark_Project 的 cn.itcast.transformation包中创建一个名为 GroupByKeyDemo 的 Scala 文件,用于将文

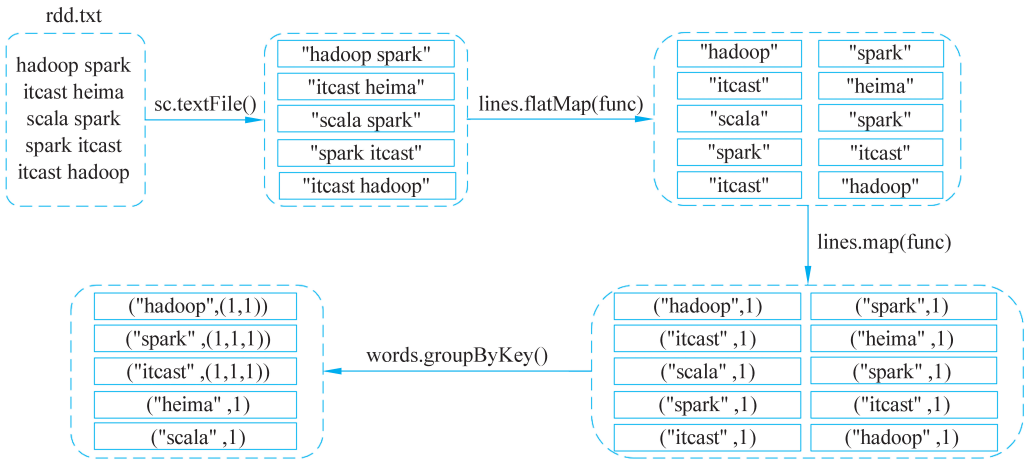

图 3-12　groupByKey 算子的处理过程

件 rdd.txt 中的每个单词进行分组，具体代码如文件 3-5 所示。

文件 3-5　GroupByKeyDemo.scala

```
1  package cn.itcast.transformation
2  import org.apache.spark.{SparkConf, SparkContext}
3  object GroupByKeyDemo {
4    def main(args: Array[String]): Unit = {
5      //指定 Spark 程序的名称为 groupByKey,并且可以使用本地计算机的线程数为 1
6      val conf:SparkConf = new SparkConf().setAppName("groupByKey")
7        .setMaster("local[1]")
8      val sc:SparkContext = new SparkContext(conf)
9      val lines = sc.textFile("D:\\rdd.txt")
10     val words = lines.flatMap(x=>x.split(" ")).map(y=>(y,1))
11     val result = words.groupByKey()
12     result.saveAsTextFile("D:\\Spark_out\\GroupByKey_out")
13     //释放资源
14     sc.stop()
15   }
16 }
```

在文件 3-5 中，第 9 行代码用于从本地文件系统中读取文件 rdd.txt 的数据，并创建一个名为 lines 的 RDD。

第 10 行代码，首先 flatMap 算子将 lines 的每个元素通过匿名函数拆分为单词。然后，map 算子将每个单词通过匿名函数映射为键值对的形式，将每个键值对作为输出的一个元素存放在名为 words 的 RDD 中。

第 11 行代码，通过 groupByKey 算子将 words 中相同键的元素划分到同一组，生成名为 result 的 RDD。

第 12 行代码使用 saveAsTextFile()方法将 result 中的元素输出到指定目录的文件中，每个元素会占用文件的一行空间。

文件 3-5 运行完成后，在本地计算机的目录 D:\\Spark_out\\GroupByKey_out 中，通过记事本查看文件 part-00000 的内容，如图 3-13 所示。

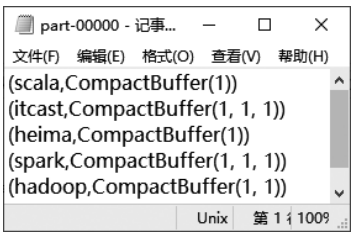

图 3-13 查看文件 part-00000 的内容(4)

从图 3-13 可以看出,文件 part-00000 中的每一行数据为键值对的形式,其中键为单词,值为迭代器,迭代器中 1 的数量,决定了相应单词出现的次数。说明使用 groupByKey 算子成功将 RDD 中相同键的元素划分到同一组。

5. reduceByKey 算子

reduceByKey 算子可以将 RDD 中具有相同键的元素中的值通过指定函数进行合并,返回一个新的 RDD。新的 RDD 中每个元素都是一个键值对,每个元素的键对应的值都是经过合并的结果。接下来,以文件 3-1 为例,通过一张图来描述如何通过 reduceByKey 算子统计文件 rdd.txt 中每个单词出现的次数,具体处理过程如图 3-14 所示。

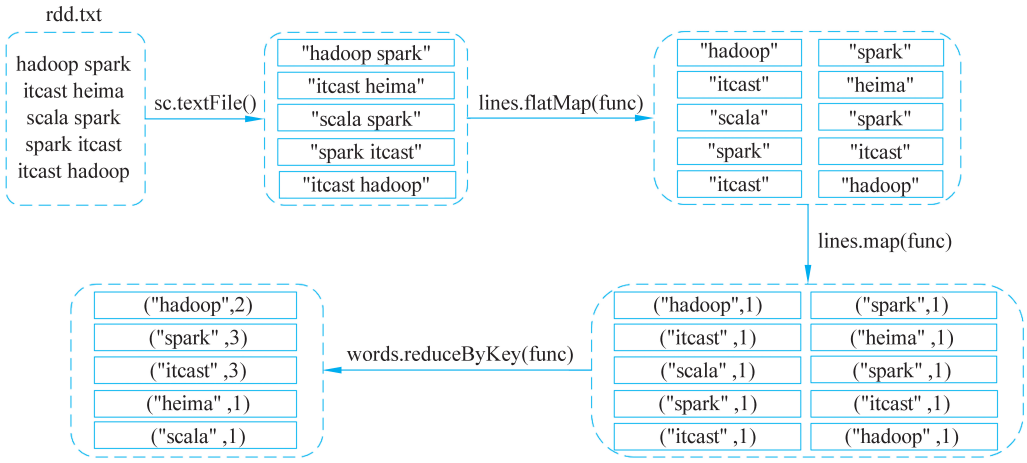

图 3-14 reduceByKey 算子的处理过程

在图 3-14 中,首先,通过从文件 rdd.txt 读取数据创建 RDD。其次,通过 flatMap 算子将 RDD 的每个元素通过函数 func 拆分为单词。然后,通过 map 算子将 RDD 的每个元素通过函数 func 映射为键值对的形式,其中键为单词,值为 1 用于标识单词出现的次数。最后,通过 reduceByKey 算子将相同键的元素中的值进行合并,返回一个新的 RDD。

接下来,以 IntelliJ IDEA 为例,演示如何使用 reduceByKey 算子。在项目 Spark_Project 的 cn.itcast.transformation 包中创建一个名为 ReduceByKeyDemo 的 Scala 文件,用于统计文件 rdd.txt 中每个单词出现的次数,具体代码如文件 3-6 所示。

文件 3-6 ReduceByKeyDemo.scala

```
1  package cn.itcast.transformation
2  import org.apache.spark.{SparkConf, SparkContext}
3  object ReduceByKeyDemo {
4    def main(args: Array[String]): Unit = {
5      //指定 Spark 程序的名称为 reduceByKey,并且可以使用本地计算机的线程数为 1
6      val conf:SparkConf = new SparkConf().setAppName("reduceByKey")
7        .setMaster("local[1]")
8      val sc:SparkContext = new SparkContext(conf)
```

```
9       val lines = sc.textFile("D:\\rdd.txt")
10      val words = lines.flatMap(x=>x.split(" ")).map(y=>(y,1))
11      val result = words.reduceByKey((a,b)=>a+b)
12      result.saveAsTextFile("D:\\Spark_out\\ReduceByKey_out")
13      //释放资源
14      sc.stop()
15    }
16 }
```

在文件 3-6 中，第 9 行代码用于从本地文件系统中读取文件 rdd.txt 的数据，并创建一个名为 lines 的 RDD。

第 10 行代码，首先 flatMap 算子将 lines 的每个元素通过匿名函数拆分为单词。然后，map 算子将每个单词通过匿名函数映射为键值对的形式，将每个键值对作为输出的一个元素存放在名为 words 的 RDD 中。

第 11 行代码，通过 reduceByKey 算子将 words 中具有相同键的元素中的值进行合并。在 reduceByKey 算子中，指定的函数是一个匿名函数，用于对相同键的元素中的值进行相加，返回一个名为 result 的 RDD。

第 12 行代码使用 saveAsTextFile() 方法将 result 中的元素输出到指定目录的文件中，每个元素会占用文件的一行空间。

文件 3-6 运行完成后，在本地计算机的目录 D:\\Spark_out\\ReduceByKey_out 中，通过记事本查看文件 part-00000 的内容，如图 3-15 所示。

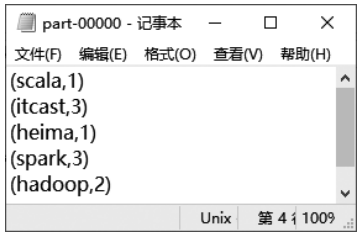

图 3-15　查看文件 part-00000 的内容(5)

从图 3-15 可以看出，文件 part-00000 中的每一行数据为键值对的形式，其中键为单词，值为单词出现的次数。说明使用 reduceByKey 算子成功将 RDD 中具有相同键的元素中的值进行合并。

3.3.2　行动算子

行动算子用于触发 RDD 的实际计算，并将计算结果返回给驱动器程序或者写入外部存储系统。与转换算子不同，行动算子并不会创建新的 RDD。接下来，通过表 3-2 来列举一些常用的行动算子。

表 3-2　常用的行动算子

算　子	语 法 格 式	说　明
count	RDD.count()	获取 RDD 中元素的数量
first	RDD.first()	获取 RDD 中的第一个元素
take	RDD.take(n)	以数组的形式返回 RDD 中的前 n 个元素
reduce	RDD.reduce(func)	使用指定的函数 func 对 RDD 中的元素进行聚合操作
collect	RDD.collect()	以数组的形式返回 RDD 中的所有元素
foreach	RDD.foreach(func)	对 RDD 中的每个元素应用指定的函数 func

下面演示如何使用表 3-2 中列举的常用行动算子。

1. count 算子

以 IntelliJ IDEA 为例,演示如何使用 count 算子,具体操作步骤如下。

(1) 在本地计算机 D 盘的根目录下创建文件 num.txt,具体内容如下。

```
1
2
3
4
5
```

(2) 在项目 Spark_Project 的目录/src/main/scala 下,新建一个名为 cn.itcast.action 的包。在 cn.itcast.action 包中创建一个名为 CountDemo 的 Scala 文件,用于通过 count 算子统计文件 num.txt 的行数,具体代码如文件 3-7 所示。

文件 3-7 CountDemo.scala

```
1  package cn.itcast.action
2  import org.apache.spark.{SparkConf, SparkContext}
3  object CountDemo {
4    def main(args: Array[String]): Unit = {
5      //指定 Spark 程序的名称为 count,并且可以使用本地计算机的所有线程
6      val conf:SparkConf = new SparkConf().setAppName("count")
7        .setMaster("local[*]")
8      val sc:SparkContext = new SparkContext(conf)
9      val lines = sc.textFile("D:\\num.txt")
10     println(lines.count())
11     //释放资源
12     sc.stop()
13   }
14 }
```

在文件 3-7 中,第 9 行代码用于从本地文件系统中读取文件 num.txt 的数据,并创建一个名为 lines 的 RDD。第 10 行代码通过 count 算子获取 lines 中元素的数量,并将其输出到控制台。

文件 3-7 的运行结果如图 3-16 所示。

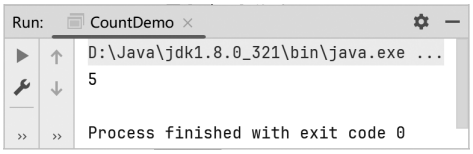

图 3-16 文件 3-7 的运行结果

从图 3-16 可以看出,lines 中元素的数量为 5。由于 lines 中元素的数量与文件 num.txt 的行数一致,因此可以推断出文件 num.txt 有 5 行数据。

2. first 算子

以 IntelliJ IDEA 为例,演示如何使用 first 算子。在项目 Spark_Project 的 cn.itcast.action 包中创建一个名为 FirstDemo 的 Scala 文件,用于通过 first 算子获取文件 num.txt 的第一行数据,具体代码如文件 3-8 所示。

文件 3-8　FirstDemo.scala

```
1  package cn.itcast.action
2  import org.apache.spark.{SparkConf, SparkContext}
3  object FirstDemo {
4    def main(args: Array[String]): Unit = {
5      //指定 Spark 程序的名称为 first,并且可以使用本地计算机的所有线程
6      val conf:SparkConf = new SparkConf().setAppName("first")
7        .setMaster("local[*]")
8      val sc:SparkContext = new SparkContext(conf)
9      val lines = sc.textFile("D:\\num.txt")
10     println(lines.first())
11     //释放资源
12     sc.stop()
13   }
14 }
```

在文件 3-8 中,第 9 行代码用于从本地文件系统中读取文件 num.txt 的数据,并创建一个名为 lines 的 RDD。第 10 行代码通过 first 算子获取 lines 的第一个元素,并将其输出到控制台。

文件 3-8 的运行结果如图 3-17 所示。

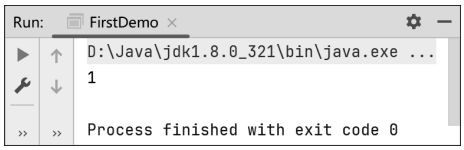

图 3-17　文件 3-8 的运行结果

从图 3-17 可以看出,lines 的第一个元素为 1。由于 lines 中的元素与文件 num.txt 中的数据一致,所以可以推断出文件 num.txt 的第一行数据为 1。

3. take 算子

以 IntelliJ IDEA 为例,演示如何使用 take 算子。在项目 Spark_Project 的 cn.itcast.action 包中创建一个名为 TakeDemo 的 Scala 文件,用于通过 take 算子获取文件 num.txt 的前 3 行数据,具体代码如文件 3-9 所示。

文件 3-9　TakeDemo.scala

```
1  package cn.itcast.action
2  import org.apache.spark.{SparkConf, SparkContext}
3  object TakeDemo {
4    def main(args: Array[String]): Unit = {
5      //指定 Spark 程序的名称为 take,并且可以使用本地计算机的所有线程
6      val conf:SparkConf = new SparkConf().setAppName("take")
7        .setMaster("local[*]")
8      val sc:SparkContext = new SparkContext(conf)
9      val lines = sc.textFile("D:\\num.txt")
10     println(lines.take(3).mkString("\n"))
11     //释放资源
12     sc.stop()
13   }
14 }
```

在文件 3-9 中,第 9 行代码用于从本地文件系统中读取文件 num.txt 的数据,并创建一个名为 lines 的 RDD。第 10 行代码通过 take 算子获取 lines 的前 3 个元素,并将其输出到控制台。由于 take 算子是以数组的形式返回 RDD 中的前 n 个元素,所以为了方便查看结果,使用 mkString()方法将数组中的元素通过指定分隔符组合成字符串。

文件 3-9 的运行结果如图 3-18 所示。

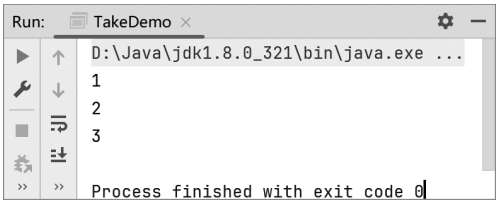

图 3-18 文件 3-9 的运行结果

从图 3-18 可以看出,lines 的前 3 个元素分别为 1、2 和 3。由于 lines 中的元素与文件 num.txt 中的数据一致,所以可以推断出文件 num.txt 前 3 行数据分别为 1、2 和 3。

4. reduce 算子

以 IntelliJ IDEA 为例,演示如何使用 reduce 算子。在项目 Spark_Project 的 cn.itcast.action 包中创建一个名为 ReduceDemo 的 Scala 文件,用于通过 reduce 算子对文件 num.txt 中的数据进行相加的聚合运算,具体代码如文件 3-10 所示。

文件 3-10 ReduceDemo.scala

```
1  package cn.itcast.action
2  import org.apache.spark.{SparkConf, SparkContext}
3  object ReduceDemo {
4    def main(args: Array[String]): Unit = {
5      //指定 Spark 程序的名称为 reduce,并且可以使用本地计算机的所有线程
6      val conf:SparkConf = new SparkConf().setAppName("reduce")
7        .setMaster("local[*]")
8      val sc:SparkContext = new SparkContext(conf)
9      val lines = sc.textFile("D:\\num.txt").map(x=>(x.toInt))
10     println(lines.reduce((a,b)=>a+b))
11     //释放资源
12     sc.stop()
13   }
14 }
```

在文件 3-10 中,第 9 行代码首先从本地文件系统中读取文件 num.txt 的数据,然后通过 map 算子将元素的数据类型转换为 Int,并创建一个名为 lines 的 RDD。第 10 行代码通过 reduce 算子对 lines 中的元素进行相加的聚合运算,并将运算结果输出到控制台。

文件 3-10 的运行结果如图 3-19 所示。

从图 3-19 可以看出,lines 中元素的相加结果为 15。由于 lines 中的元素与文件 num.txt 中的数据一致,所以可以推断出文件 num.txt 中数据的相加结果为 15。

5. collect 算子

以 IntelliJ IDEA 为例,演示如何使用 collect 算子。在项目 Spark_Project 的 cn.itcast.action包中创建一个名为 CollectDemo 的 Scala 文件,用于通过 collect 算子获取文件 num.txt

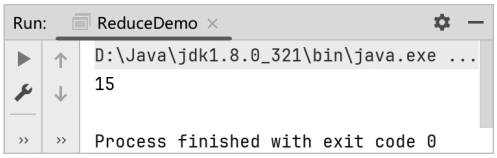

图 3-19　文件 3-10 的运行结果

中的所有数据，具体代码如文件 3-11 所示。

文件 3-11　CollectDemo.scala

```
1  package cn.itcast.action
2  import org.apache.spark.{SparkConf, SparkContext}
3  object CollectDemo {
4    def main(args: Array[String]): Unit = {
5      //指定 Spark 程序的名称为 collect,并且可以使用本地计算机的所有线程
6      val conf:SparkConf = new SparkConf().setAppName("collect")
7        .setMaster("local[*]")
8      val sc:SparkContext = new SparkContext(conf)
9      val lines = sc.textFile("D:\\num.txt")
10     println(lines.collect().mkString("\n"))
11     //释放资源
12     sc.stop()
13   }
14 }
```

在文件 3-11 中，第 9 行代码用于从本地文件系统中读取文件 num.txt 的数据，并创建一个名为 lines 的 RDD。第 10 行代码通过 collect 算子获取 lines 的所有元素，并将其输出到控制台。由于 collect 算子是以数组的形式返回 RDD 中的所有元素，所以为了方便查看结果，使用 mkString()方法将数组中的元素通过指定分隔符组合成字符串。

文件 3-11 的运行结果如图 3-20 所示。

图 3-20　文件 3-11 的运行结果

从图 3-20 可以看出，lines 的所有元素分别为 1、2、3、4 和 5。由于 lines 中的元素与文件 num.txt 中的数据一致，所以可以推断出文件 num.txt 的所有数据分别为 1、2、3、4 和 5。

6. foreach 算子

以 IntelliJ IDEA 为例，演示如何使用 foreach 算子。在项目 Spark_Project 的 cn.itcast.action 包中创建一个名为 ForeachDemo 的 Scala 文件，用于通过 foreach 算子获取文件 num.txt 中的所有数据，具体代码如文件 3-12 所示。

文件 3-12 ForeachDemo.scala

```
1  package cn.itcast.action
2  import org.apache.spark.{SparkConf, SparkContext}
3  object ForeachDemo {
4    def main(args: Array[String]): Unit = {
5      //指定 Spark 程序的名称为 foreach,并且可以使用本地计算机的线程数为 1
6      val conf:SparkConf = new SparkConf().setAppName("foreach")
7        .setMaster("local[1]")
8      val sc:SparkContext = new SparkContext(conf)
9      val lines = sc.textFile("D:\\num.txt")
10     lines.foreach(x=>println(x))
11     //释放资源
12     sc.stop()
13   }
14 }
```

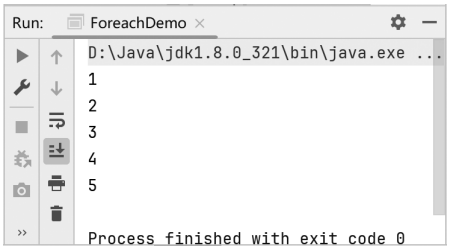

图 3-21 文件 3-12 的运行结果

在文件 3-12 中,第 9 行代码用于从本地文件系统中读取文件 num.txt 的数据,并创建一个名为 lines 的 RDD。第 10 行代码通过 foreach 算子将 lines 中的每个元素应用指定的匿名函数,该匿名函数用于依次取出 lines 中的每个元素并赋值给变量 x,然后通过 println()方法将元素输出到控制台。

文件 3-12 的运行结果如图 3-21 所示。

从图 3-21 可以看出,lines 的所有元素分别为 1、2、3、4 和 5。由于 lines 中的元素与文件 num.txt 中的数据一致,所以可以推断出文件 num.txt 所有数据分别为 1、2、3、4 和 5。

【注意】 当 Spark 程序能够利用本地计算机的多个线程时,foreach 算子会在每个分区上并行执行指定的函数,因此 RDD 中的每个元素不会按顺序依次应用函数。如果在允许 Spark 程序使用本地计算机的多个线程的情况下,希望让 RDD 中的每个元素按顺序依次应用 foreach 算子中指定的函数,可以通过指定 RDD 的分区数为 1 来实现。关于这部分内容可以参考 3.4 节。

3.4 RDD 的分区

在分布式计算中,网络通信开销是一个关键的性能瓶颈。因此,合理控制数据分布以减少网络传输对整体性能的影响至关重要。在编写 Spark 程序时,用户可以通过 parallelize()方法、repartition()方法和 coalesce()方法手动指定 RDD 的分区数量来精确地控制数据的分布。关于这 3 个方法的介绍如下。

- parallelize()方法用于在创建 RDD 时指定分区数量。
- repartition()方法用于增加或减少 RDD 的分区数量,它会触发重分区操作,从而生成新的 RDD。
- coalesce()方法通常用于减少 RDD 的分区数量,它也会触发重分区操作,从而生成新的 RDD。

repartition()方法和 coalesce()方法都用于减少 RDD 分区的数量，但它们的行为有所不同。repartition()方法会触发一个 Shuffle 过程，即数据会通过网络传输重新洗牌，以满足新的分区需求。与此不同，coalesce()方法不会触发 Shuffle 过程，它只是将原始分区中的数据合并到新的分区中，尽量保持数据的原始分布。在处理大规模数据集时，Shuffle 过程可能会消耗大量的网络带宽和计算资源。因此，使用 coalesce()方法减少 RDD 分区的数量时，性能开销相对较小。但在某些情况下，coalesce()方法可能会导致数据分布不均匀。

接下来，以 IntelliJ IDEA 为例，演示如何指定 RDD 的分区数量。在项目 Spark_Project 的目录/src/main/scala 下，新建一个名为 cn.itcast.partition 的包。在 cn.itcast.partition 包中创建一个名为 PartitionDemo 的 Scala 文件，具体代码如文件 3-13 所示。

文件 3-13　PartitionDemo.scala

```
1  package cn.itcast.partition
2  import org.apache.spark.{SparkConf, SparkContext}
3  object PartitionDemo {
4    def main(args: Array[String]): Unit = {
5      //指定 Spark 程序的名称为 partition,并且可以使用本地计算机的所有线程
6      val conf:SparkConf = new SparkConf().setAppName("partition")
7        .setMaster("local[*]")
8      val sc = new SparkContext(conf)
9      val arr = Array(1,2,3,4,5)
10     //创建名为 data1 的 RDD,并指定 RDD 的分区为 3
11     val data1 = sc.parallelize(arr,3)
12     println("data1 的分区数为:" + data1.getNumPartitions)
13     //将 data1 的分区数修改为 5,生成名为 data2 的 RDD
14     val data2 = data1.repartition(5)
15     println("data2 的分区数为:" + data2.getNumPartitions)
16     //将 data2 的分区数修改为 4,生成名为 data3 的 RDD
17     val data3 = data2.coalesce(4)
18     println("data3 的分区数为:" + data3.getNumPartitions)
19   }
20 }
```

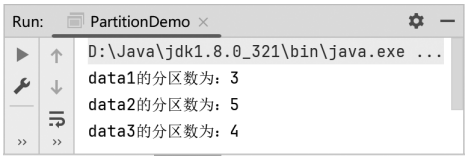

图 3-22　文件 3-13 的运行结果

在文件 3-13 中，getNumPartitions()方法用于获取 RDD 的分区数。

文件 3-13 的运行结果如图 3-22 所示。

从图 3-22 可以看出，data1、data2 和 data3 的分区数分别为 3、5 和 4。说明成功在创建 RDD 时指定分区数量，以及修改 RDD 的分区数量。

3.5　RDD 的依赖关系

在 Spark 中，不同的 RDD 之间可能存在依赖关系，这种依赖关系分为两种，分别是窄依赖(Narrow Dependency)和宽依赖(Wide Dependency)，具体介绍如下。

1. 窄依赖

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。在 Spark 中，父

RDD 指的是生成当前 RDD 的原始 RDD 或者转换操作之前的 RDD。而子 RDD 则是由当前 RDD 生成的 RDD 或者转换操作之后的 RDD。

窄依赖的表现形式通常分为两类：第一类表现为一个父 RDD 的分区对应一个子 RDD 的分区；第二类表现为多个父 RDD 的分区对应一个子 RDD 的分区。但是，一个父 RDD 的分区不会对应多个子 RDD 的分区。为了便于理解，通常把窄依赖形象地比喻为独生子女继承家产。接下来，通过图 3-23 来展示常见的窄依赖及其对应的操作。

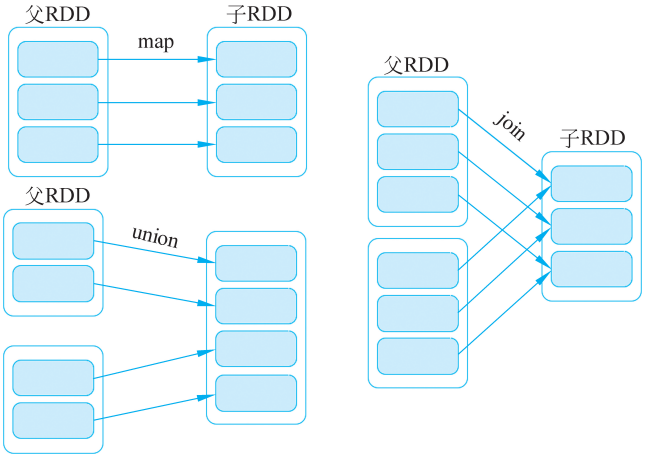

图 3-23 窄依赖

从图 3-23 可以看出，RDD 在进行 map 算子和 union 算子操作时，属于窄依赖的第一类表现；而 RDD 进行 join 算子操作时，属于窄依赖的第二类表现。

2. 宽依赖

宽依赖是指子 RDD 的每个分区都会使用父 RDD 的全部或多个分区。为了更直观理解这个概念，可以把宽依赖形象地比喻为兄弟姐妹共同继承家产。接下来，通过图 3-24 来展示常见的宽依赖及其对应的操作。

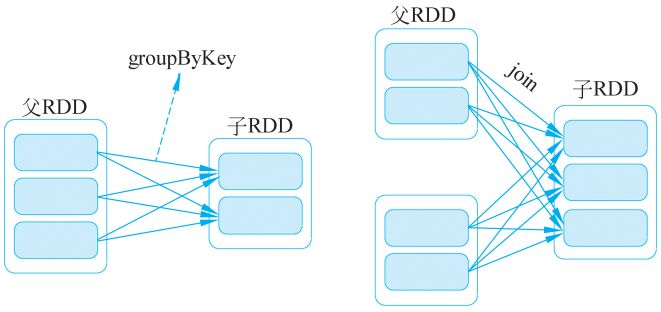

图 3-24 宽依赖

从图 3-24 可以看出，RDD 在进行 groupByKey 算子和 join 算子操作时为宽依赖。

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，如果子 RDD 的分区数与父 RDD 相同则为窄依赖；当 join 算子操作后，如果子 RDD 的分区数与父 RDD 不同则为宽依赖。

3.6　RDD 机制

Spark 为 RDD 提供了两个重要的机制，分别是持久化机制和容错机制。接下来，本节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

持久化机制，也称为缓存机制，用于将 RDD 缓存在内存或磁盘上，以备后续重用。在 Spark 中，由于 RDD 采用惰性求值的方式，意味着 RDD 的转换操作不会立即执行计算。只有在遇到行动操作时，Spark 才会根据 RDD 之间的依赖关系，触发转换操作执行计算。在存在多个行动算子的情况下，每个行动算子都可能导致转换操作的重复计算。为了避免这种资源开销，可通过持久化机制将重复使用的 RDD 缓存到内存或磁盘中，从而避免重复计算，提高计算效率。

通常情况下，一个 RDD 由多个分区组成，数据分布在多个节点中。因此，当对某个 RDD 进行持久化时，每个节点都会将其分区的计算结果缓存在内存或磁盘中。在对 RDD 或其衍生出的 RDD 执行行动操作时，无须重新计算，而是直接获取各个分区的计算结果，从而极大提高后续行动操作的速度。RDD 的持久化机制是 Spark 构建迭代式算法和快速交互式查询的关键，因为它可以避免多次使用的 RDD 导致转换操作的重复计算所产生的资源开销。

在编写 Spark 程序时，用户可以通过 cache()方法和 persist()方法持久化 RDD，其中 cache()方法使用 RDD 默认的持久化级别，将 RDD 缓存到内存中。而 persist()方法可以通过传递的参数指定持久化级别，将 RDD 缓存到内存或磁盘中。接下来，通过表 3-3 介绍 RDD 的持久化级别。

表 3-3　RDD 的持久化级别

持久化级别	说　明
MEMORY_ONLY	RDD 默认的持久化级别。将 RDD 缓存在 JVM 堆内存中。若 RDD 无法完全缓存在内存中，则某些分区将不会被缓存。可能会导致性能下降，因为需要重新计算丢失的分区
MEMORY_AND_DISK	将 RDD 缓存在 JVM 堆内存中。若 RDD 无法完全缓存在内存中，则某些分区将存储在磁盘上，并在需要时从磁盘读取
MEMORY_ONLY_SER	类似于 MEMORY_ONLY，但是使用序列化的方式将 RDD 缓存在 JVM 堆内存中。相比于 MEMORY_ONLY，使用序列化可以减少内存消耗，并提高缓存的效率
MEMORY_AND_DISK_SER	类似于 MEMORY_AND_DISK。但是使用序列化的方式将 RDD 缓存在 JVM 堆内存中
DISK_ONLY	仅将 RDD 缓存在磁盘中
MEMORY_ONLY_2、MEMORY_AND_DISK_2	分别与 MEMORY_ONLY 和 MEMORY_AND_DISK 类似，不同的是加上后缀_2，表示 RDD 的每个分区都会缓存 2 个副本
OFF_HEAP	将 RDD 缓存到堆外内存中