第三章 RDD 编程

本章介绍Spark对数据的核心抽象——弹性分布式数据集（Resilient Distributed Dataset，RDD），它具备像MapReduce等数据流模型的位置感知、容错性和负载均衡等特性。现有的基于数据集的方式不能复用曾经的计算结果，总是从物理存储上加载数据，然后操作数据，最后写入物理存储设备。而Spark中的RDD是基于工作集的工作模式，能将数据保存在内存中以提高性能，Spark的RDD之间具有依赖关系且高度抽象，编程模型更容易，容错更好。

本章会对RDD进行深入的分析，以增强读者的动手实践能力。在细致解析RDD的基础上，会动手实战RDD中的Transformation类型的操作和Action类型的操作，建议以循序渐进的动手实践方法来学习本章。

# 3.1 RDD基础

## 3.1.1 RDD简介

与许多其他大数据处理平台不同，Spark采用统一的抽象数据集RDD来存储正在被处理的大数据，这种建立在统一数据模型之上的计算模式，使得它可以以基本一致的方式应对不同的大数据处理场景。

RDD是Spark的核心数据结构，是一个容错的、并行的数据结构，是一种高度受限的共享内存模型，即RDD是只读的记录分区的集合，能横跨集群的所有节点进行并行计算，是一种基于工作集（即多个并行操作重用中间结果）的应用抽象。这个数据集的全部或者部分可以缓存在内存中，在多次计算间重用，所谓弹性就是指在内存不够时可以与磁盘进行交换。RDD可以让用户显示地将数据存储到磁盘和内存中，并能控制数据的分区，且提供了丰富的API来操作数据。逻辑上认为RDD是一个不可变的分布式对象集合，而集合中的每个元素可以是用户自定义的任意数据结构。RDD通过其依赖关系形成Spark的调度顺序，通过RDD的操作形成整个Spark程序。

## 3.1.2 RDD的基本特征

**（1）分区列表（A list of partitions）**

将RDD划分成很多的分区（partitions）分布到集群的节点上，分区的多少涉及到对RDD进行并行计算的粒度，每一个分区的数据能够进行并行计算，RDD的并行度默认从父RDD传给子RDD。RDD本质上是逻辑分区记录的集合，在集群中一个RDD可以包含多个分布在不同节点上的分区，每个分区是一个dataset片段。

在对RDD操作中，用户可以使用partitions.size的方法获取该RDD划分的分区数目；也可以通过手动设置分区数目，方法一在创建RDD调用textFile或者parallelize和makeRDD方法时手动指定分区个数即可，如果不指定分区数量，当RDD从集合创建时，则默认分区数量为该程序所分配到的资源的CPU核数，若是从HDFS文件创建，默认为文件的Block数；方法二是通过转换操作得到新的RDD时，直接调用repartition方法强制改变RDD的分区数量即可。语法格式如下：

语法格式为：sc.textFile( path , partitionNum)

语法格式为：sc.parallelize( seq , partitionNum)

语法格式为：sc.makeRDD( seq , partitionNum)

语法格式为：sc.repartition(partitionNum)

**（2）计算每个分区的函数（A function for computing each split）**

每一个分区都有一个计算函数，Spark的RDD的计算是以分片为基本单位的，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）依赖于其他RDD上的列表（A list of dependencies on other RDDs）**

由于RDD每次转换都会生成新的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，但宽依赖就不类似于流水线了，对于依赖关系为宽依赖的子RDD具体的数据分片会依赖于所有父RDD的所有数据分片，所以当有分区的数据丢失时，Spark会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。依赖具体分为宽依赖和窄依赖，但不是所有的RDD都有依赖，源RDD没有依赖。

**（4）键值对key-value数据类型的RDD的分区策略（A partitioner for key-value RDDs）**

一个Partitioner，即RDD的分片函数。每个key-value形式的RDD都会有partitioner属性，非key-value的RDD的partitioner的值是None。当前spark中实现了两种类型的分片函数，一个是基于哈希的HashPartitioner，另一个是基于范围的RangePartitioner。Partitioner函数不但决定了RDD本身的分片数量，也决定了parent RDD Shuffle输出时的分片数量。

**（5）计算每个分区的优先位置的列表（A list of preferred locations to compute each split）**

优先位置列表存储存取每个Partitioner的优先位置，对于一个HDFS文件而言，这个列表就是每个Partition所在的块的位置。依据大数据中数据不动代码动的原则，spark本身在进行任务调度的时候会尽可能地将任务分配到其所要处理数据块的存储位置。

## 3.1.3 依赖关系

窄依赖是指父RDD的每个分区最多只对应子RDD的一个分区，而子RDD的每一个分区可以依赖常数个父RDD的分区（与数据规模无关，O(1)）。例如map，filter等操作。

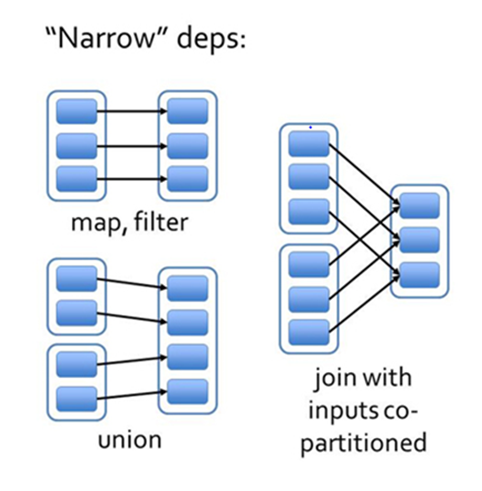


图3-1 窄依赖关系

对于窄依赖的实现如下：

abstract class NarrowDependency[T](\_rdd:RDD[T]) extends Dependency[T] {

//返回子RDD的partitionId所依赖的所有的parent RDD的Partition(s)

def getParents(partitionId:Int):Seq[Int]

override def rdd: RDD[T] = \_rdd

}

在Spark的源代码中，把窄依赖分为两类：一类是一对一的依赖关系（OneToOneDependency），如图3-2所示的map、filter和join with inputs co-partitioned。对于一对一的依赖的源代码如下：

class OneToOneDependency[T](rdd:RDD[T]) extends NarrowDependency[T](rdd){  
 override def getParents(partitionId:Int) = List(partitionId)

}

可以看到，子RDD在通过getParents方法时，查询的是相同partitionId的内容，也就是说子RDD仅仅依赖父RDD中相同partitionId的Partition。

第二类是范围的依赖（RangeDependency），它仅仅被org.apache.spark.rdd.UnionRDD使用。UnionRDD是将多个RDD拼接合成一个RDD，即每个parent RDD的Partition的相对顺序不变，只不过每个父RDD的分区在UnionRDD中的分区的起始位置不同。如图3-2所示的union。对于范围的依赖的源代码如下：

override def getParents(partitionId:Int):List[Int] = {  
 if (partitionId >= outStart && partitionId < outStart + length) {

//outStart是在UnionRDD 中的起始位置

//length是parent RDD中Partition的数量

List(partitionId – outStart + inStart)

//inStart是这个Partition在parent RDD中Partition的起始位置 }

else{ Nil }

}

可以看到，在RangeDependency中是将父RDD中的Partition根据partitionId的顺序依次插入到子RDD中。

宽依赖是指多个子RDD分区依赖一个父RDD分区，即父RDD的每个分区都有可能被子RDD的多个分区所使用（与数据规模有关，O(n)）。例如groupByKey等操作，如图3-3。

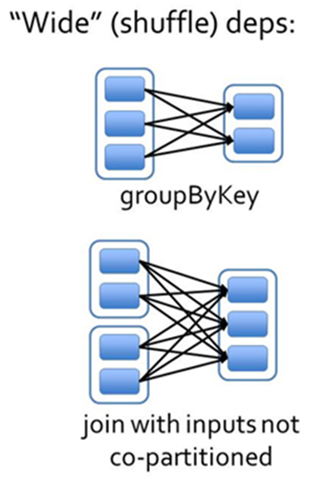


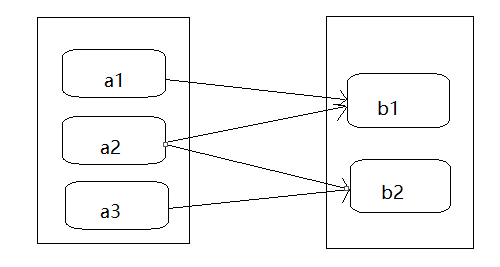
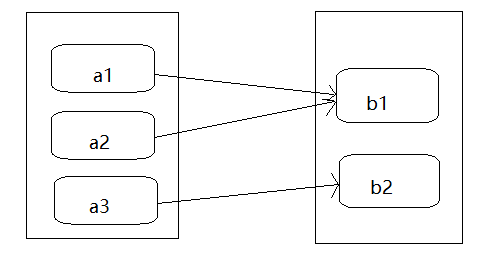
图3-2　宽依赖关系

Spark中RDD的高效与DAG图有着很大的关系，在DAG图的调度中需要对计算过程进行划分stage，而划分的依据就是RDD之间的依赖关系。总是将存在窄依赖关系的RDD划分在同一个stage，因为相对于宽依赖，窄依赖对优化更有利。原因如下：

（1）宽依赖需要在运行过程中将同一个父RDD的分区传入到不同的子RDD分区中，中间可能涉及多个节点之间的数据传输；而窄依赖的每个父RDD的分区只会传入到一个子RDD分区中，通常可以在一个节点内完成转换。

（2）当RDD的某个分区数据丢失时，由于窄依赖中父RDD的一个分区只对应子RDD的一个分区，所以在进行数据重算的时候只需要重算丢失RDD分区所对应的父RDD分区，重算效率100%；而宽依赖中父RDD的一个分区可能对应子RDD的多个分区，在数据重算的时候可能会出现冗余计算。

如下图3-4中（a）图，当b1出现数据丢失的时候，只需要重新计算a1和a2分区，并且不会出现冗余计算。（b）图中，当b1出现数据丢失，需要重新计算a1和a2分区，但是a2分区中部分数据对应b2分区，所以在重新计算a1,a2分区时出现了冗余计算（a2中对应b2的数据）。



（a）窄依赖 （b）宽依赖

图3-3 宽窄依赖对比

# 3.2 创建RDD

Spark的一切都是基于RDD的，创建RDD是很重要的一步，除了可以直接从父RDD转换，Spark还提供了两种创建RDD的方式：

1）并行化一个程序中已经存在的集合（例如：数组）；

2）引用一个外部文件存储系统中的数据集，包括本地的文件系统，还有所有Hadoop支持的数据集，比如HDFS、Cassandra、HBase、Tachyon等。

## 3.2.1 从已有集合（数组）创建RDD

（1）使用parallelize()方法创建RDD

parallelize方法的定义如下：

parallelize[T](seq:Seq[T],numSlices:Int = defaultParallelizelism):RDD[T]

其中第一个参数为一个已存在的对象集合，第二个参数为设定的分区数（第二个参数可不需要），返回指定对象类型的RDD。

例3.3 用parallelize方法创建RDD

scala> val data = Array(1,2,3,4,5)

data: Array[Int] = Array(1, 2, 3, 4, 5)

scala> val distData = sc.parallelize(data)

distData: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[12] at parallelize at <console>:26

scala> distData.partitions.size

res25: Int = 1

scala> val disData2 = sc.parallelize(data,4)

disData2: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[13] at parallelize at <console>:26

scala> disData2.partitions.size

res30: Int = 4

（2）使用makeRDD()创建RDD

makeRDD方法的定义如下：

makeRDD[T](seq:Seq[T],numSlices:Int = defaultParallelizelism):RDD[T]

例3.4 用makeRDD方法创建RDD

scala> val data1 = sc.makeRDD(Array("changsha","is a beautiful city","yes"))

data1: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[17] at makeRDD at <console>:24

scala> data1.collect

res31: Array[String] = Array(changsha, is a beautiful city, yes)

一旦创建完成，分布式集合就可以被并行操作，例如调用reduce((a,b) => a + b)将这个数组中的元素相加。并行集合有个很重要的参数，即切片数（slices），表示一个数据集的切片的份数。Spark会在集群的每一个分片上运行一个任务，集群中的每个CPU通常需要2~4个分区，可以通过parallelize和makeRDD中的第二个参数手动设置。

## 3.2.2 从外部存储创建RDD

RDD可以通过SparkContext的textFile方法创建，方法定义如下：

textFile(path: String, minPartitions: Int = defaultMinPartirions):RDD[String]

**textFile()函数参数分析：**

（1）path:String，path用来指定RDD外部数据源路径信息的URI（Uniform Resource Identifier，统一资源标识符）地址（本地文件路径，或者hdfs://、sdn://、kfs://……），并且以“行”的集合形式读取。

（2）minPartitions: Int = defaultMinPartirions，minPartitions参数用来指定生成的RDD的分区（partition）数，需要注意的是RDD的partition个数其实是在逻辑上将数据集进行划分，RDD各分区的实质是记录着数据源的各个文件块（block）在HDFS位置的信息集合，并不是数据源本身的集合，因此RDD partition数目也受HDFS的split size影响，HDFS默认文件块（block）大小为128MB，这就意味着当数据源文件小于128MB时，RDD分区数并不会按照minPartitions进行指定分区，而只有一个分区。

另外需要注意的一点是RDD的每个Partition对应着一个task（执行任务），如果partition的数量多，能起实例的资源也多，那自然并发度就多；如果partition数量少，资源很多，它也不会有很多并发。如果partition的数量很多，但是资源少，那么并发也不大，它会算完一批再继续下一批，所以根据集群资源合理地设置分区数，有利于提高并行度、充分利用资源。

有关Spark的textFile()读取文件的一些注意事项如下：

①如果需要从本地文件系统读取文件作为外部数据源，则文件必须确保集群上的所有工作节点可访问。可以将文件复制到所有工作节点或使用集群上的共享文件系统。

②Spark所有的基于文件的读取方法，包括textFile支持读取某个目录下多个指定文件，支持部分的压缩文件和通配符。具体实现如下所示：

val text1 = sc.textFile(“/my/directory/\*”)

//读取该目录下所有文件

val text2 = sc.textFile(“/my/directory/\*.txt”)

//采用通配符匹配同一类型的文件进行读取

val text3 = sc.textFile(“/my/directory/\*.gz”)

//也可以读取压缩文件

val text4 = sc.textFile(“/my/directory/test1.txt” , “/my/directory/test2.txt”)

//同时读取来自不同路径的多个文件

③该textFile方法还采用可选的第二个参数来控制文件的分区数。默认情况下，Spark为文件的每个块创建一个分区（HDFS中默认为128MB），但也可以通过传递更大的值来请求更高数量的分区。请注意：不能有比块少的分区。

例3.5 用textFile方法创建RDD

scala> val data = sc.textFile("/usr/spark/spark-2.3.0-bin-hadoop2.7/README.md",3)

data:org.apache.spark.rdd.RDD[String]=/usr/spark/spark-2.3.0-bin-hadoop2.7/README.md MapPartitionsRDD[7] at textFile at <console>:24

# 3.3 RDD算子

Spark在运行中通过算子对RDD进行计算算子是RDD中定义的函数，可以对RDD中的数据进行转换和操作。对于RDD可以有两种操作算子：Transformation和Action，在Transformation算子中再将数据类型维度细分为Value数据类型和Key-Value对数据类型的Transformation算子。Value型数据的算子封装在RDD类中可以直接使用，Key-Value对数据类型的算子封装在PairRDDFunctions类中，用户需要引用import.org.apache.spark.SparkContext.\_才能使用。进行这样的细分是由于不同的数据类型处理思想不太一样，同时有些算子是不同的。

## 3.3.1 Transformation算子

该操作是从已经存在的数据集上创建一个新的数据集，是数据集的逻辑操作，但不会触发一次真正的计算。Transformation都具有Lazy特性，不立即计算RDD的结果，仅仅记录转换操作应用到哪些RDD上，Transformation仅仅在执行Action时才进行计算（起作用），在Action之前不发生动作。

Action要数据的时候Transformation才会开始工作，Spark里任何一个正常的作业都是没有要计算的，最后一步要生成结果的时候，从后往前回溯父RDD有没有计算，以及父RDD的父RDD有没有计算，这就是Transformation的Lazy特性。这样的特性可以避免产生很多不必要的中间临时数据，这比较符合分布式并行计算的需求；另一个层面是调度层面，最后一步要计算时，可以看到前面的所有步骤，看见的步骤越多，进行优化的机会就越多，所以Spark是基于Lazy特性进行操作、基于Lineage来构建整个调度系统的，最终形成了DAG。目前常用的Transformation算子操作如下表。

表3-1RDD支持的Transformation

|  |  |
| --- | --- |
| **Transformation算子** | **含义** |
| map(func) | 数据集中的每条元素经过func函数转换后形成一个新的分布式数据集。 |
| filter(func) | 过滤函数，选取数据集中让函数func返回值为true的函数，形成一个新的数据集。 |
| flatMap(func) | 类似与map方法，但每一个输入项可以被映射为0个或者更多的输出项（所以func函数应该返回的是Seq而不是一个单独项）。 |
| union(otherDataset) | 返回一个由原数据集和参数数据集联合（求并集）而成的新的数据集。 |
| distinct([numPartitions]) | 返回一个数据集去重后得到的新的数据集。 |
| repartition(numPartitions) | 在RDD上随机重洗数据，从而创造出更多或者更少的分区以及它们之间的平衡。 |
| subtract(otherDataset) | 返回原数据集中存在的元素而参数数据集中不存在的数据的一个新的数据集。 |
| groupByKey([numTasks]) | 当在一个由键值对（K，V）组成的数据集上调用时，按照key进行分组，返回一个（K，Iterable<V>）键值对的数据集。 |
| reduceByKey(func,[numTasks]) | 当在一个键值对（K，V）数据集上调用，按照key将数据分组，使用给定的func聚合values值，返回一个键值对（K，V）数据集。 |
| sortByKey([ascending],[numTasks]) | 返回一个以key排序（升序或者降序）的（K，V）键值对组成的数据集，其中布尔代数ascending参数决定升序还是降序。 |
| join(otherDataset,[numTasks]) | 根据key连接两个数据集，将类型为（K，V）和（K，W）的数据集合合并成一个（K，（V，W））类型的数据集。 |

1. **Value型Transformation算子**
2. map(func)

map算子对RDD的转换如图3-4所示，图中RDD中的每个方框是RDD的一个分区，RDD\_1的分区经过用户自定义函数func:T->U映射为RDD\_2中新的分区。



图3-4 map算子对RDD转换

如以下例3.1，通过调用paralellize方法构建一个RDD，调用map(word=>(word,1))方法将数据集中每个元素转换成一个元组。

例3.6 map算子的实现

scala> val rdd = sc.parallelize(Array("hello","world","hello","changsha"))

rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[7] at parallelize at <console>:24filter(func)

scala> val line = rdd.map(word=>(word,1)) .foreach(println)

(hello,1)

(world,1)

(hello,1)

(changsha,1)

-

1. filter(func)

filter算子对RDD的转换如图3-5所示，图中RDD中的每个方框是RDD的一个分区，RDD\_1分区的元素经过用户自定义函数func:T->U筛选出func返回值为true的元素形成一个新的数据集。



图3-5 filter算子对RDD转换

如以下例3.2，对于例子3.1中的数据集，调用filter(word=>word.contains("hello"))方法将数据集中每个含有hello的元素形成新的数据集返回。

例3.7 filter算子的实现

scala> val line1 = rdd.filter(word=>word.contains("hello")).foreach(println)

hello

hello

1. flatMap(func)

flatMap算子对RDD的转换如图3-6所示，图中RDD中的每个方框是RDD的一个分区，RDD\_1的每个分区的元素经过用户自定义函数func:T->U转换为新的元素，并放入一个集合中，映射在RDD\_2中的分区中。



图3-6 flatMap算子对RDD转换

如以下例子3.3，创建一个RDD，调用flatMap(line=>line.split(" "))方法，将每一行用空格区分筛选得到新的数据集并返回。

例3.8 flatMap算子的实现

test的内容是：

hello spark

hello scala

hello python

hello bigdata

scla> val rdd2 = sc.textFile("/home/ubuntu01/test")

rdd2: org.apache.spark.rdd.RDD[String] = /home/ubuntu01/test MapPartitionsRDD[7] at textFile at <console>:24

scala> val line2 = rdd2.flatMap(line=>line.split(" ")).foreach(println)

hello

spark

hello

scala

hello

python

hello

bigdata

1. union(otherDataset)

union算子对RDD的转换如图3-7所示，图中RDD中的每个方框是RDD的一个分区。含有V1、V2…M2的RDD与含有V3、V6…M6的RDD合并所有元素形成一个RDD，V1、V2、V3、V3、V6形成一个分区，其他元素同理进行合并。



图3-7 union算子对RDD转换

如以下例子3.4，创建两个RDD，用其中一个RDD调用union方法，参数为另一个RDD，得到两个数据集的并集并作为结果返回。

例3.8 union算子的实现

scala> val rdd\_a = sc.parallelize(Array("apple","orange","pineapple","pineapple"))

rdd\_a: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[10] at parallelize at <console>:24

scala> val rdd\_b = sc.parallelize(Array("apple","orange","grape"))

rdd\_b: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[11] at parallelize at <console>:24

scala> val rdd\_union = rdd\_a.union(rdd\_b).foreach(println)

apple

orange

pineapple

pineapple

apple

orange

grape

1. distinct([numPartitions])

distinct算子对RDD的转换如图3-8所示，图中RDD中的方框是RDD的一个分区。旨在将RDD\_1中数据去重，剩下的元素组成一个新的RDD\_2。



图3-8 distinct算子对RDD转换

如以下例子3.5，用rdd\_a调用distinct方法将rdd\_a数据集中重复的元素去除得到新的数据集作为结果返回。

例3.9 distinct算子的实现

scala> val rdd\_distinct = rdd\_a.distinct().foreach(println)   
orange

pineapple

apple

1. repartition(numPartitions)

这个操作将重洗网络上所有的数据。如以下例子3.6，用rdd\_a调用partitions.size方法得到rdd\_a的分区数，再将其调用repartition方法强行改变分区数，则得到的结果便是改变后的分区数。

例3.10 repartition算子的实现

scala> rdd\_a.partitions.size

res4: Int = 1

scala> val rdd\_re = rdd\_a.repartition(4)

rdd\_re: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[29] at repartition at <console>:25

scala> rdd\_re.partitions.size

res7: Int = 4

1. subtract(otherDataset)

subtract算子对RDD的转换如图3-9所示，subtract相当于进行集合的差操作，RDD\_1去除RDD\_1和RDD\_2交集中的所有元素组成一个新的RDD为RDD\_3。



图3-9 subtract算子对RDD转换

如以下例子3.7，用rdd\_b调用subtract方法，参数数据集为rdd\_a，得到的结果是rdd\_b中含有的元素而rdd\_a中不含有的元素组成的新的数据集并返回。

例3.11 subtract算子的实现

scala> val rdd\_sub = rdd\_b.subtract(rdd\_a).foreach(println)

grape

1. **键值对转换操作**
2. groupByKey([numTasks])

当在一个由键值对（K，V）组成的数据集上调用时，按照key进行分组，返回一个（K，Iterable<V>）键值对的数据集。

注意：

◎如果是为了按照key值聚合数据（如进行求和，求平均值等操作）而进行分组，使用reduceByKey或者combineByKey方法会产生更好的性能。

◎默认情况下，输出的并行程度取决于父RDD的分区数目。可以通过传递一个可选的numTasks参数设置不同的并行任务数。

图3-10描述了RDD进行groupByKey时的内部RDD转换的实现逻辑图。



图3-10 groupByKey算子对RDD转换

如以下例子3.8，新建RDD后，对该RDD中的key值进行分组，key值相同的元素归为一组，例如元素（2，6）和（2，2）key值相同故归为一组。

例3.12 groupByKey算子的实现

scala> val rdd3 = sc.parallelize(Array((1,3),(2,6),(2,2),(3,6)))

rdd3: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[42] at parallelize at <console>:24

scala> val rdd\_gbk = rdd3.groupByKey().foreach(println)

(1,CompactBuffer(3))

(3,CompactBuffer(6))

(2,CompactBuffer(6, 2))

1. reduceByKey(func,[numTasks])

当在一个键值对（K，V）数据集上调用，按照key将数据分组，使用给定的func聚合values值，返回一个键值对（K，V）数据集，其中func函数的类型必须是（V，V）=>V。类似于groupByKey，Reduce并行任务数也可以通过可选的第二个参数进行配置。图3-11描述了RDD进行reduceByKey时的内部RDD转换的实现逻辑图。



图3-11 reduceByKey算子对RDD转换

如以下例子3.9，对于元素为键值对的数据集，定义func为两数相加，调用reduceByKey方法将key值相等的values值相加形成新的键值对，最后组成新的数据集并返回。

例3.13 reduceByKey算子的实现

scala> val rdd1 = sc.parallelize(Array((1,2),(2,3),(2,6),(3,8),(3,10)))

rdd1: org.apache.spark.rdd.RDD[(Int, Int)] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> val rdd2 = rdd1.reduceByKey((x,y)=>x+y).foreach(println)

(1,2)

(3,18)

(2,9)

1. sortByKey([ascending],[numTasks])

返回一个以key排序（升序或者降序）的（K，V）键值对组成的数据集，其中布尔代数ascending参数决定升序还是降序，若为true则升序，若为false则降序，默认为升序，而numTasks为并行任务数目。如以下例子3.10，调用sortByKey方法，将数据集中的每个键值对元素按照key值升序排序，形成新的数据集并返回。图3-12描述了RDD进行sortByKey时的内部RDD转换的实现逻辑图。



图3-12 sortByKey算子对RDD转换

例3.14 sortByKey算子的实现

scala> val rdd3 = rdd1.sortByKey(true).foreach(println)

(1,2)

(2,3)

(2,6)

(3,8)

(3,10)

1. join(otherDataset,[numTasks])

根据key连接两个数据集，将类型为（K，V）和（K，W）的数据集合合并成一个（K，（V，W））类型的数据集，其中numTasks为并行任务数目。图3-13描述了RDD进行join时的内部RDD转换的实现逻辑图。



图3-13 join算子对RDD转换

如以下例子3.11，新创建两个键值对类型的RDD，用其中一个RDD调用join方法，参数为另一个RDD，则两个RDD根据key值进行匹配，key值相等的将values值合并，组成（K，（V1，V2））类型的数据集并返回。

例3.15 join算子的实现

scala> val rdd1 = sc.parallelize(Array(("a",1),("b",1),("c",1)))

rdd1: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[7] at parallelize at <console>:24

scala> val rdd2 = sc.parallelize(Array(("a",2),("b",2),("c",2)))

rdd2: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[8] at parallelize at <console>:24

scala> val rdd3 = rdd1.join(rdd2). foreach(println)

(a,(1,2))

(b,(1,2))

(c,(1,2))

## 3.3.2 Action算子

Action是一种算法的描述，它通过SparkContext的runJob方法提交作业（Job），触发RDD DAG的执行并将数据输出到Spark系统。Action在RDD（数据集）上进行计算之后返回一个值到Driver，这样设计能让Spark运行得更加高效。目前常用的Action算子操作如下表。

表3-2 RDD支持的Action

|  |  |
| --- | --- |
| **Action算子** | **含义** |
| reduce(func) | 通过函数func聚集数据集中的所有元素，func函数接收两个参数，作用在RDD两个相同类型的元素上，返回一个值。 |
| fold(zeroValue)(func) | 和reduce(func)一样，并且需要提供初始值。 |
| aggregate(zeroValue)(seqOp,combOp) | aggregate函数首先用初始值（zeroValue）和seqOp操作，将每个分区里面的元素进行聚合，对聚合后的每个分区会返回一个类型为U的值，然后再用combOp函数将各个分区的返回值再次进行聚合。 |
| collect() | 以数组的形式返回数据集的所有元素。 |
| count() | 返回数据集的元素个数。 |
| first() | 返回数据集的第一个元素。 |
| take(n) | 以数组的形式，返回数据集上的前n个元素。 |
| foreach(func) | 在数据集的每个元素上都运行func函数。 |
| countByKey() | 只能运行在键值对类型（K，V）上，对每个key的个数进行计数。 |
| saveAsTextFile(path) | 将数据集的元素作为一个文本文件（或文本文件的集合）保存在本地文件系统中的给定目录、HDFS或任何其他Hadoop HDFS支持的文件系统。 |
| saveAsSequenceFile(path) | 将数据集的元素在本地文件系统中以sequencefile的格式保存至指定路径、Hadoop HDFS或Hadoop支持的任何文件系统。 |
| saveAsObjectFile(path) | 将数据集中的元素序列化成对象，存储到文件中，对于HDFS，默认采用以sequencefile的格式保存。 |

**1.常用行动操作**

（1）reduce(func : (T,T)=>T) : T

通过函数func聚集数据集中的所有元素，func函数接收两个参数，作用在RDD两个相同类型的元素上，返回一个值，这个函数必须满足交换律和结合律，以保证可以被正确地并发执行。可以实现RDD中累加、计数等聚集操作。由定义可知，归并RDD数据后得到的值的类型必须是和RDD元素的类型一致。图3-14描述了RDD进行reduce时的内部RDD操作的实现逻辑图。

图3-14 reduce算子对RDD转换

如以下例子，新创建一个RDD，数据集的每个元素都为int型，调用reduce方法，将每个元素进行累加，结果返回一个int类型的数据。

例3.16 reduce算子的实现

scala> val rdd4 = sc.parallelize(Array(3,4,5,6,7))

rdd4: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[12] at parallelize at <console>:24

scala> val rdd5 = rdd4.reduce((x,y)=>x+y)

rdd5: Int = 25

（2）fold(zeroValue : T)(func : (T,T) => T) : T

与reduce()类似，聚合每个分区的元素，然后使用具有关联性的操作，以及一个初始值，将每个分区聚合的结果进行归并，不同的是每次对分区内的value聚集时，分区内初始化的值为zeroValue。给定的func(t1,t2)操作运行修改第一个参数值，并返回其结果，这可以避免对结果值的内存分配，但不应该修改第二个参数值。由定义可知，fold操作之后，用于归并的初始值以及操作的返回值的类型都必须和RDD元素的类型一致。

图3-15描述了key-value类型的RDD进行fold时的内部RDD操作的实现逻辑图。

 图3-15 fold算子对key-value类型RDD转换

例3.17 fold算子对key-value类型RDD的实现

scala> val data = sc.parallelize(Array(("A",1),("B",2),("C",3)))

data: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[0] at parallelize at <console>:24

scala> data.fold(("V0",2))((A,B)=>(A.\_1 + "@" + B.\_1,A.\_2+B.\_2))

res5: (String, Int) = (V0@V0@A@B@C,10)

图3-16描述了value类型的RDD进行fold时的内部RDD操作的实现逻辑图。



图3-16 fold算子对value类型RDD转换

例3.18 fold算子对value类型RDD的实现

scala> val data1 = sc.parallelize(Array(1,2,3,4,5))

data1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at parallelize at <console>:24

scala> data1.fold(0)((A,B)=>A+B)

res7: Int = 15

（3）aggregate(zeroValue : U)(seqOp : (U,T) => U,combOp : (U,U) => U) : U

是一个聚合操作，允许用户对RDD使用两个不同的reduce函数，第一个reduce函数对各个分区内的数据聚集，每个分区得到一个结果。第二个reduce函数对每个分区的结果进行聚集，最终得到一个总的结果。Aggregate相当于对RDD内的元素数据归并聚集，且这种聚集是可以并行的。而fold和reduce的聚集是串行的。由定义可知，通过aggregate操作，最终得到类型为U的值，和初始值zeroValue的类型相同，但不需要和RDD中元素类型一致。图3-17描述了RDD进行aggregate时的内部RDD操作的实现逻辑图。

图3-17 aggregate算子对RDD转换

例3.19 aggregate算子的实现

scala> val data = sc.parallelize(Array(("A",1),("B",2),("C",3)))

data: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[2] at parallelize at <console>:24

scala> data.aggregate(("V0",2))((A,B)=>(A.\_1 + "@" + B.\_1,A.\_2+B.\_2),(A,B)=>(A.\_1 + "$" + B.\_1,A.\_2+B.\_2))

res10: (String, Int) = (V0$V0@A@B@C,10)

（2）collect()

在Driver程序中，以数组的形式返回数据集的所有元素到Driver程序，为防止Driver程序内存溢出，一般要控制返回的数据子集大小。图3-18描述了RDD进行collect时的内部RDD操作的实现逻辑图。



图3-18 collect算子对RDD转换

例3.20 collect算子的实现

scala> rdd4.collect

res1: Array[Int] = Array(3, 4, 5, 6, 7)

（3）count()

返回数据集的元素个数。如下图3-19所示，RDD中一个方框代表RDD中的一个分区。



图3-19 count算子对RDD转换

例3.21 count算子的实现

scala> rdd4.count

res2: Long = 5

（4）first()

返回数据集的第一个元素。



图3-20 first算子对RDD转换

例3.22 first算子的实现

scala> rdd4.first

res3: Int = 3

（5）take(n)

以数组的形式，返回数据集上的前n个元素。



图3-21 take算子对RDD转换

例3.23 take算子的实现

scala> rdd4.take(3)

res4: Array[Int] = Array(3, 4, 5)

（6）foreach(func)

在数据集的每个元素上都运行func函数。要注意如果对RDD执行foreach，只会在Executor端有效，而并不是Driver端，比如：rdd.foreach(println)，只会在Executor的stdout中打印出来，Driver端是看不到的。它也会有副作用，如会更新累加器变量或与外部存储系统相互作用。图3-22自定义的函数是println()，控制台打印所有的数据项。



图3-22 foreach算子对RDD转换

例3.24 foreach算子的实现

scala> rdd4.foreach(println)

3

4

5

6

7

（7）countByKey()

只能运行在键值对类型（K，V）上，对每个key的个数进行计数。

例3.25 countByKey算子的实现

scala> val rdd5 = sc.parallelize(Array(("a",1),("b",2),("a",5),("b",6)))

rdd5: org.apache.spark.rdd.RDD[(String, Int)] = ParallelCollectionRDD[13] at parallelize at <console>:24

scala> rdd5.countByKey

res10: scala.collection.Map[String,Long] = Map(a -> 2, b -> 2)

2.存储行动操作

（1）saveAsTextFile(path)

将数据集的元素作为一个文本文件（或文本文件的集合）保存在本地文件系统中的给定目录、HDFS或任何其他Hadoop HDFS支持的文件系统。Spark会对每个元素调用toString方法将其转换为一个文件中的文本行。图3-23中RDD\_1中的方框代表RDD分区，右侧方框代表HDFS的Block。通过函数将RDD的每个分区存储为HDFS中的一个Block。



图3-23 saveAsTextFile算子对RDD转换

例3.26 saveAsTextFile算子的实现

scala> val rdd6 = sc.parallelize(Array("a","b","c","d"))

rdd6: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[16] at parallelize at <console>:24

scala> rdd6.saveAsTextFile("/home/ubuntu01/TextFile1")

//查看TextFile1中文本文件Part-00000的内容为

a

b

c

d

（2）saveAsSequenceFile(path)

将数据集的元素在本地文件系统中以sequencefile的格式保存至指定路径、Hadoop HDFS或Hadoop支持的任何文件系统。该方法可作用于任意实现了Hadoop的读写接口的RDD键值对。

（3）saveAsObjectFile(path)

使用Java序列化将数据集的元素写入到一个简单的格式中，该格式的数据可以使用SparkContext的objectfile()函数加载。saveAsObjectFile将分区中的每个元素组成一个Array，然后将这个Array序列化，映射为（Null，BytesWritable(Y)）的元素，写入HDFS为SequenceFile的格式。图3-24中左侧方框代表RDD分区，右侧方框代表HDFS的Block。通过函数将RDD的每个分区存储为HDFS上的一个Block。



图3-24 saveAsObjectFile对RDD转换

## 3.3.3 不同类型RDD之间的转换

有些函数只能用于特定类型之间的RDD上，不如mean()、stdev()、和sum()这种数值计算的操作只能用在数值RDD上，而reduceByKey()、groupByKey()这种键值对操作只能用在键值对RDD上，在Scala和Java中，这些函数都没有定义在标准的RDD类中，所以要访问这些附加功能，必须要确保获得了正确的专用RDD类。

如下图3-25所示，是Spark支持的多类RDD（包括标准的RDD类，和一些专用的RDD类，例如：Pair RDD、jdbcRDD等）以及封装了针对特定RDD类的专用函数的函数类（http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.package）

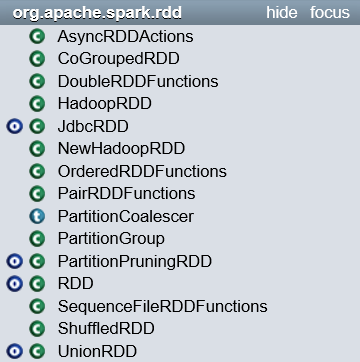


图3-25

在Scala中，将RDD转为有特定函数的RDD（比如在RDD[Double]上进行数值操作）是由隐式转换来自动处理的，我们需要加上import org.apache.spark.SparkContext.\_来使用这些隐式转换。可以在SparkContext对象的Scala文档中查看所列出的隐式转换，这些隐式转换可以隐式地将一个RDD转为各种封装类，这样我们就有了诸如mean()之类的额外的函数来提供相应的额外的功能。隐式转换后的类包括以下几种：

（1）PairRDDFunctions：该扩展类中的方法中输入的数据单元是一个包含两个元素的元组结构。Spark会把其中第一个元素当成Key，第二个当成Value。例如前面提到的reduceByKey()，grooupByKey()函数等。

（2）DoubleRDDFunctions：这个扩展类包含了很多数值的聚合方法。如果RDD的数值单元能够隐式变换成Scala的double数据类型，则这个方法会非常有用。一些常用的该扩展类的方法如下表所示。以data = sc.parallelize(List(1.1,1.2,2.6,2.3,6.5,6.6),2)为例。

表3-3 DoubleRDDFunctions扩展类中的方法

|  |  |  |  |
| --- | --- | --- | --- |
| 函数 | 功能描述 | 举例 | 结果 |
| mean() | 求RDD的平均值 | scala> data.mean | res12: Double = 3.383333333333334 |
| sampleStdev | 求RDD元素的样本标准偏差 | scala>data.sampleStdev | res13: Double = 2.5230272821883375 |
| sampleVariance | 计算RDD元素的样本偏差 | scala>data.sampleVariance | res14: Double = 6.365666666666668 |
| Stats | RDD元素的统计，包含平均值、样本偏差、最大值和最小值 | scala> data.stats | res15: org.apache.spark.util.StatCounter = (count: 6, mean: 3.383333, stdev: 2.303198, max: 6.600000, min: 1.100000) |
| Stdev | 计算RDD元素的标准偏差 | scala> data.stdev | res16: Double = 2.303198259425841 |
| Sum | RDD元素的求和 | scala> data.sum | res17: Double = 20.3 |
| variance | 求RDD元素的方差 | scala> data.variance | res18: Double = 5.304722222222223 |

（3）OrderedRDDFunctions：该扩展类的方法需要输入的数据是2元元组，并且key能够排序。例如前面提到的sortByKey()函数等。

（4）SequenceFileRDDFunctions：这个扩展类包含一些可以创建Hadoop sequence文件的方法。输入的数据必须是2元元组。但需要额外考虑到元组元素能够转换成可写类型。

隐式转化虽然强大，但是会让阅读代码的人感到困惑。如果你对RDD调用了像mean()这样的函数 ，可能会发现RDD类的Scala文档中根本没有该函数。调用之所以能够成功，是因为隐式转换可以把RDD[Double]转为DoubleRDDFunctions。当我们在Scala文档中查找函数时，不要忘记那些封装的专用类中的函数。

# 3.4 数据的读取与保存

Spark抽象出了RDD，在物理上RDD通常由多个Partition组成，一个partition对应一个block。在driver和每个executor端，都有一个Blockmanager。Blockmanager是spark在计算过程中对block进行读写的入口，它屏蔽了在读取数据时涉及到的内存分配，从其他executor端远程获取等具体细节。

Spark支持很多种输入输出源，一部分原因是Spark本身是基于Hadoop生态圈而构建，特别是Spark可以通过Hadoop MapReduce所使用的InputFormat和OutputFormat接口访问数据。对于存储在本地文件系统或者分布式文件系统中的数据，Spark可以访问很多种不同的文件格式，包括文本文件、JSON、SequenceFile以及protocol buffer。

接下来，将以读写block为主线，分析spark在计算过程中读写实际数据的流程。展示几种常见的格式如表3-4所示，以及不同文件格式的读写。

表3-4 Spark支持的一些常见格式

|  |  |  |
| --- | --- | --- |
| 格式名称 | 结构化 | 备注 |
| 文本文件 | 否 | 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，半结构化；大多数库都要求每行一条记录 |
| CSV | 是 | 非常常见的基于文本的格式，通常在电子表格应用中使用 |
| SequenceFiles | 是 | 一种用于键值对数据的常见Hadoop文件格式 |
| Protocol buffers | 是 | 一种快速、节约空间的跨语言格式 |
| 对象文件 | 是 | 用来将Spark作业中的数据存储下来以让共享的代码读取。改变类的时候它会失效，因为它依赖于Java序列化 |

## 3.4.1 数据读写流程分析

**一、读数据过程**

BlockManager的get方法是读数据的入口点，在读取时分为本地读取和远程节点读取两个步骤。

**1、内存读取**

本地读取使用getLocalValue方法，读取内存中的数据返回的是封装成BlockResult类型还是数据流，分别调用MemoryStore的getValues和getBytes两种方法，最终都是通过数据块编号获取内存中的数据。

**2、远程节点读取**

（1）Spark远程读取数据入口为getRemoteValues，然后调用getRemoteBytes方法，在该方法中调用getLocations方法先向BlockManagerMasterEndpoint终端点发送GetLocations消息，请求数据块所在的位置信息。当Driver的终端点接收到请求信息时，根据数据块的编号获取该数据块所在的位置信息，根据是否是本地节点数据对位置列表进行排序。获取数据块的位置列表之后，在BlockManager.getRemoteBytes方法中调用BlockTransferService提供的fetchBlockSync方法读取远程数据。

（2）调用远程数据传输服务BlockTransferService提供的fetchBlockSync方法后，在该方法中持续调用fetchBlocks方法。该方法是一个抽象方法，实际上调用的是Netty远程数据服务NettyBlockTransferService类中的fetchBlocks方法。在fetchBlocks方法中，根据远程节点的地址和端口创建通信客户端TransportClient，通过在OneForOneBlockFetche类中发送OpenBlocks消息向该RPC客户端向指定节点发送读取数据消息，

（3）当远程节点的RPC服务端接收到客户端发送信息时，在NettyBlockRpcServer类中对消息进行匹配。如果是请求读取消息时，则调用BlockManager的getBlockData方法读取该节点上的数据，读取的数据块封装为ManagedBuffer序列缓存在内存中，然后使用Netty提供的传输通道，把数据传递到请求节点上，完成远程传输任务。

**二、写数据过程**

BlockManager的doPutIterator方法是写数据的入口点，在该方法中，根据数据是否缓存到内存中进行分别处理。

**1、写入内存**

首先判断数据存储级别是否进行了反序列化。如果设置了反序列化，则说明获取的数据为值类型，调用putIteratorAsValues方法把数据存入内存；如果没有设置反序列化，则获取的数据为字节类型，调用putIteratorAsBytes方法把数据存入内存中。在把数据存入内存过程中，需要判断在内存中展开该数据大小是否足够，当足够时调用BlockManager的putArray方法写入内存，否则将数据写入磁盘。

在写入数据完成时，一方面把数据块的元数据发送给Driver端的BlockManagerMasterEndpoint终端点，请求其更新数据元数据，另一方面判断是否需要创新数据副本，如果需要则调用replicate方法，把数据写到远程节点上，类似于读取远程节点数据，Spark提供Netty方式写数据。

**2、写入磁盘**

Spark写入磁盘的方法调用了DiskStore的put方法，该方法提供了写入文件的回调方法writeFunc。在该方法中先获取写入文件句柄，然后把数据序列化为数据流，最后根据回调方法把数据写入文件中。

## 3.4.2 数据读写格式解析

Spark对很多种文件格式的读取和保存方式都很简单，从诸如文本文件的非结构化的文件，到诸如JSON格式的半结构化文件，再到诸如SequenceFile这样的结构化文件，Spark都支持。Spark会根据文件扩展名选择对应的处理方式，这一过程是封装好的，对用户透明。下表展示了各个格式的数据读写方式。

表3-5 数据读写格式解析表

|  |  |  |
| --- | --- | --- |
| 格式名称 | 读取数据 | 保存数据 |
| 文本文件 | 将一个文本文件读取为RDD 时，输入的每一行都会成为 RDD的一个元素，inputFile为文件目录。  **sc.textFile(inputfile)** | 保存文本文件用**saveAsTextFile(outputFile)**。  其中outputFile为保存文件目录，将RDD中的内容都输入到路径对应的文件中。Spark将传入的路径作为目录对待，会在那个目录下输出多个文件，这样，Spark就可以从多个节点上并行输出了，但在这种方法中，我们不能控制数据的哪一部分输出到哪个文件中，有些输出格式支持控制。 |
| 将多个完整的文本文件一次性读取为一个 pair RDD， 其中键是文件名，值是文件内容，inputFile为一个包含多个文件的目录。  sc.wholeTextFiles(inputFile) |
| JSON | 将数据当做普通文本文件读取， inputFile为文件目录。  **sc.textFile(inputFile)**  然后用JSON库scala.util.parsing.json.JSON对JSON数据进行解析， JSON.parseFull(jsonString:String)函数，以一个JSON字符串作为输入并进行解析，如果解析成功则返回一个Some(map: Map[String, Any])，如果解析失败则返回None。 | 保存JSON文件比读取要简单，因为不需要考虑格式错误的数据，并且知道需要写出数据的类型，可以使用之前将字符串RDD转为解析好的JSON数据的库，将由结构化数据组成的RDD转为字符串RDD（使用writeValueAsString将结果放入RDD），然后使用Spark的文本文件API写出。  **saveAsTextFile(outputFile)** |
| CSV文件（逗号分隔值文件） | 将数据当做普通文本文件读取，然后CSV库对CSV数据进行解析。inputFile为文件目录。  **sc.textFile(inputFile)** | 和JSON数据一样，使用CSV库输出到文件或者输出器，可以使用StringWriter或者StringIO来将结果放到RDD中。  用**saveAsTextFile(outputFile)**保存文件 |
| SequenceFile | 在SparkContext中，调用**sequenceFile(path , keyClass , valueClass , minPartitions)**读取SequenceFile文件 | 在SparkContext中，调用**saveAsSequenceFile(outputFile)**保存文件 |
| 对象文件 | 读取对象文件用SparkContext的**objectFile()**函数接受一个路径，返回对应的RDD | 保存对象文件调用 **saveAsObjectFile** |
| Hadoop输入输出格式 | Spark可以与任何Hadoop支持的格式交互。读取其他Hadoop输入格式，使用**newAPIHadoopFile**接收一个路径以及三个类，第一个类是格式类，代表输入格式，第二个类是键的类，最后一个类是值的类，如果需要设定额外的Hadoop配置属性，也可以传入一个conf对象。hadoopFile()函数用于使用旧的API实现的Hadoop输入格式。KeyValueTextInputFormat 是最简单的 Hadoop 输入格式之一，可以用于从文本文件中读取键值对数据。每一行都会被独立处理，键和值之间用制表符隔开。  例如：  Val data = sc.newAPIHadoopFile(  "E:\\share\\spark\\test.json" ,  classOf[KeyValueTextInputFormat],  classOf[Text],  classOf[Text],  job.getConfiguration) | 使用**saveAsNewAPIHadoopFile**函数存储。  例如：  data.saveAsNewAPIHadoopFile(  "E:\\share\\spark\\savehadoop",  classOf[Text],  classOf[Text],  classOf[TextOutputFormat[Text,Text]],  job.getConfiguration) |

# 3.5 深入理解RDD

## 3.5.1 RDD的缓存机制（持久化）

调用行动操作的时候，会触发一次从开始到结尾的运算，这对于迭代运算而言，代价是非常大的，迭代计算经常需要使用上一次运算的结果或者同一组数据。为了避免重复计算的开销，就涉及到持久化（缓存）机制的问题，即第一次行动操作得到的结果，如果能被第二次行动操作使用，则不需要从头开始的计算，直接使用持久化的结果。

可以通过使用persist()或者cache()方法标记持久化的RDD，使其被保存在内存的节点上，spark的cache()方法默认为将RDD缓存在内存中，实际上是调用persist(MEMORY\_ONLY)方法。Spark的缓存机制是容错的，如果RDD的任意分区丢失，它将会自动通过最初创建的转换操作重新计算，不需要全部重新计算，只需要计算丢失的部分。每一个需要持久化的RDD能使用不同的存储级别来存储，例如允许持久化数据集在磁盘或者内存中作为序列化的Java对象（节省空间），甚至跨节点复制。完整的存储级别如表3-6。

表3-6 持久化的存储级别

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Storage Level（存储级别） | Meaning（含义） | 使用的空间 | CPU时间 | 是否在内存上 | 是否在磁盘上 |
| MEMORY\_ONLY | 将RDD作为反序列化Java对象存储在JVM中，如果内存不足，RDD中的一些分区将不会被缓存，当它们每次被需要的时候将会被重复计算。这是默认的存储级别。 | 高 | 低 | 是 | 否 |
| MEMORY\_AND\_DISK | 将RDD作为反序列化Java对象存储在JVM中，如果内存不足，RDD中的超出的一些分区将会被存储在磁盘中，每当需要的时候会从磁盘中读取。 | 高 | 中等 | 部分 | 部分 |
| MEMORY\_ONLY\_SER（Java and Scala） | 将RDD作为序列化Java对象（每个分区占一个字节数组）存储，这比反序列化对象空间利用率更高，特别是当使用fast serializer时，但是在读取的时候更耗费CPU。 | 低 | 高 | 是 | 否 |
| MEMORY\_AND\_DISK\_SER（Java and Scala） | 类似于MEMORY\_ONLY\_SER，但是对于不能缓存而超出的分区将存储在磁盘中而不是每次需要的时候进行重新计算。 | 低 | 高 | 部分 | 部分 |
| DISK\_ONLY | 将RDD的分区仅存储在磁盘中。 | 低 | 高 | 否 | 是 |
| MEMORY\_ONLY\_2,MEMORY\_AND\_DISK\_2,etc. | 与以上存储级别一样，区别在于将每一个分区都复制到两个集群节点上。 |  |  |  |  |
| OFF\_HEAP(experimental) | 类似于MEMORY\_ONLY\_SER，但是将数据存储在off-heap memory中，这要求off-heap的内存不可丢弃，能被启用。 |  |  |  |  |

如果要缓存的数据太多，内存中放不下，Spark会自动利用最近最少使用（LRU）的缓存策略把最老的分区从内存中移除。对于仅把数据存放在内存中的缓存级别，下一次要用到已经被移除的分区时，这些分区就需要重新计算。但是对于使用内存和磁盘的缓存级别的分区来说，被移除的分区都会写入磁盘。不论哪一种情况，都不必担心你的作业会因为缓存了太多数据而被打断。不过，缓存不必要的数据会导致有用的数据被移除内存，带来更多重算的时间开销。

例3.1 未持久化的例子

scala> val list = List("Hadoop","Spark","Hive")

list: List[String] = List(Hadoop, Spark, Hive)

scala> val rdd = sc.parallelize(list)

rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[10] at parallelize at <console>:26

scala> println(rdd.count())

3

scala> println(rdd.collect().mkString(" , "))

Hadoop , Spark , Hive

rdd.count()是行动操作，触发一次真正的从头到尾的计算rdd.collect()是行动操作，触发一次真正的从头到尾的计算。

例3.2 持久化的例子

scala> val list = List("Hadoop","Spark","Hive")

list: List[String] = List(Hadoop, Spark, Hive)

scala> val rdd = sc.parallelize(list)

rdd: org.apache.spark.rdd.RDD[String] = ParallelCollectionRDD[11] at parallelize at <console>:26

scala> rdd.cache()

res22: rdd.type = ParallelCollectionRDD[11] at parallelize at <console>:26

scala> println(rdd.count())

3

scala> println(rdd.collect().mkString(" , "))

Hadoop , Spark , Hive

rdd.cache()会调用persist(MEMORY\_ONLY)，但是语句执行到这里，并不会缓存rdd，这时rdd还没有被计算生成。rdd.count()是第一次行动操作，触发一次真正的从头到尾的计算，也就是在这里只能在HDFS中读取，不是从内存中读取。这时才会执行上面的rdd.cache()，把这个rdd放到缓存中。rdd.collect()是第二次行动操作，不需要触发从头到尾的计算，只需要重复使用上面缓存中的rdd。

**存储级别的选择**

Spark的多个级别意味着在内存利用率和CPU利用率间的不同权衡。可通过下面的过程选择一个合适的存储级别。

①如果RDD适合默认的存储级别（MEMORY\_ONLY），就选择默认的存储级别。因为这是CPU利用率最高的选项，会使RDD上的操作尽可能的快。

②如果不适合用默认级别，选择MEMORY\_ONLY\_SER。选择一个更快的序列化库来提高对象的空间使用率，但是仍能够相当快的访问。

③除非算子计算RDD花费较大或者需要过滤大量的数据，否则不要将RDD存储到磁盘上，否则重复计算一个分区就会和磁盘上读取数据一样慢。

④如果希望更快地恢复错误，可以利用replicated存储级别，所有的存储级别都可以通过replicated计算丢失的数据来支持完整的容错，另外replicated的数据能在RDD上继续运行任务，而无须重复计算丢失的数据。

在拥有大量内存的环境中或者多应用程序的环境中，OFF\_HEAP将对象从堆中脱离出来序列化，然后存储在一大块内存中，这就像它存储在磁盘上一样，但它仍在RAM中。对象在这种状态下不能直接使用，它们必须首先反序列化，也不受垃圾收集机制影响。OFF\_HEAP具有以下优势：OFF\_HEAP运行多个执行者共享的Tachyon中相同的内存池；OFF\_HEAP显著地减少来自GC回收的划分；如果单个的Executor崩溃，缓存的数据不会丢失。

**移除数据**

RDD可以随意在RAM中进行缓存，因此它提供了更快速的数据访问。目前，缓存的粒度为RDD级别，只能缓存全部的RDD。

Spark自动监视每个节点上使用的缓存，在集群中没有足够的内存时，Spark会根据缓存的情况确定一个LRU的数据分区进行删除。如果想要手动删除RDD，而不想等待它从缓存中消失，可以使用RDD的unpersist()方法移除数据，unpersist()方法是立即生效的。

## 3.5.2 RDD 检查点

Spark中对于数据保存除了持久化操作之外还存在一种检查点（Checkpoint）方式，原因在于缓存的方式虽然也可以以文件形式保存在磁盘中，但是磁盘会出现损坏，文件也会出现丢失。Checkpoint的产生就是为了相对而言更加可靠的持久化数据，在Checkpoint可以指定把数据放在本地并且是多副本方式，但是在正常的生产环境下是放在 HDFS，这就天然借助了HDFS高容错性的高可靠的特性来完成了最大化的可靠的持久化数据的方式，从而降低数据被破坏或者丢失的风险，也减少了数据重新计算时的开销。

Checkpoint的运行原理是：首先需要sc.setCheckpointdir来设置一个检查点目录，再对需要缓存的RDD调用Checkpoint，当调用了Checkpoint方法要对RDD进行Checkpoint操作的话，此时框架会自动生成RDDCheckpointData，当RDD上运行过一个Job后就会立即触发RDDCheckpointData中的chckpoint方法，在其内部会调用doCkeckpoint，实际上在生产环境下会调用RDDCheckpointData的doCkeckpoint，在生产环境下会导致ReliableCheckpointRDD的writeRDDToCheckpointDirectory的调用，而在writeRDDToCheckpointDirectory方法内部会触发runJob来执行把当前的RDD中的数据写到Checkpoint的目录中，同时会产生ReliableCheckpointRDD实例。

注：在Spark中，某RDD进行Checkpoint操作后会将此RDD的依赖关系清空，该RDD的父RDD就是CheckpointRDD，故在后面的计算再使用该RDD时，若数据丢失，可以从Checkpoint中读取数据，不需要重新计算。

## 3.5.3 编程步骤

（1）RDD读入内、外部数据源进行创建。

（2）RDD经过一系列的转换操作（如map和filter）后，每一次会产生新的RDD，供下一个转换操作使用。

（3）对需要被重用的RDD手动执行persist或者cache操作。

（4）最后一个RDD经过动作操作（如collect和count）来触发一次并行计算，Spark会对记录下来的RDD转换过程进行优化后再执行计算，并输出到外部数据源。

这一系列处理称为一个Lineage（血缘关系），即DAG拓扑排序的结果。

# 3.6 综合实例

## 3.6.1 根据各班成绩查询全校前n位的情况

每个班级的成绩表为一个文件，若要统计整个全校排名前n的情况，需要对多个文件求前n个值。

对此模拟出3个班级的成绩表，分别存储，存储格式为：（序号，学号，分数，班级号）。

Class1.txt：

1,1001,50,2018001

2,1002,60,2018001

3,1003,70,2018001

4,1004,20,2018001

5,1005,80,2018001

6,1006,66,2018001

7,1007,99,2018001

Class2.txt:

1,2001,55,2018002

2,2002,56,2018002

3,2003,88,2018002

4,2004,60,2018002

5,2005,78,2018002

6,2006,62,2018002

Class3.txt:

1,3001,99,2018003

2,3002,84,2018003

3,3003,59,2018003

4,3004,71,2018003

5,3005,69,2018003

6,3006,100,2018003

基于以上数据，对其求整个数据集的排名前n位的信息。完整代码如下：

import org.apache.spark.{SparkConf,SparkContext}

object rank {

def main(args:Array[String]): Unit ={

val conf = new SparkConf().setAppName("rank").setMaster("local")

val sc = new SparkContext(conf)

sc.setLogLevel("ERROR")

//设置将错误信息记录于日志

val lines = sc.textFile("class1.txt,class2.txt,class3.txt")

//对多个文件进行读取

var num =0

val result = lines.filter(line => (line.trim().length > 0) && (line.split(",").length == 4))

.map(line => {

val fields = line.split(",")

val userid = fields(1)//每行数据的第二个属性值是学号userid

val core = fields(2).toInt//第三个属性值是分数core

val classs = fields(3)//第四个属性值是班级classs

//拼接数据

(core,(classs,userid))

})

println("rank" +"\t" + "class" +"\t" + "\t" + "userid" +"\t" + "core" +"\n")

//对数据进行排序，取前10名进行输出

val result1 = result.sortByKey(false).take(10).foreach(x => {

num = num +1

println(num + "\t\t" + x.\_2.\_1+ "\t\t" + x.\_2.\_2 +"\t" + x.\_1)

})

}

}

结果如下：

rank class userid core

1 2018003 3006 100

2 2018001 1007 99

3 2018003 3001 99

4 2018002 2003 88

5 2018003 3002 84

6 2018001 1005 80

7 2018002 2005 78

8 2018003 3004 71

9 2018001 1003 70

10 2018003 3005 69

## 3.6.2 通过基站信息追踪某个手机号码出现的位置及时长

根据手机信号可以来计算其所在的位置，手机一开机，就会和附近的基站建立连接，建立连接和断开连接都会被记录到服务器上的日志，所以即使手机没有开启网络或者GPS，也可以定位手机所在的位置。基站都有一定的辐射范围，并且根据信号强度有不同的信号级别，比如2G、3G和4G信号。 虽然不知道手机用户所在的具体位置，但是只要知道基站的位置，手机用户一旦进入基站的辐射范围，手机就会和基站之间建立连接。因此就可以计算用户大致的位置。于是就可以根据这些位置信息做一些推荐广告。比如附近的商家，手机用户可能喜欢的商品或者服务。

假如现在得到了一些位置数据，比如有手机号、建立连接的标记（比如1）、断开连接的标（比如0）、 建立连接的时间戳、断开连接的时间戳等字段。用断开连接的时间减去建立连接的时间就是用户在该基站下停留的时间。但是这种计算方式不是很好，因为在实际中用户可能会停留好几天的情况，或者说有建立连接但是没有断开连接的情况。所以这里面其实还会有一个会话的概念。其实基站不是一直保持连接的，它可能每隔一段时间他会自动断开一次。比如每隔一天就断开一次。 每个基站都有一个基站ID，这是一个UUID。所以可能还会有一个和基站相关的基站表，比如基站的id和经纬度等信息。于是应该将两个表进行join才能得到用户在基站下停留的时间等信息。这里先不考虑会话id的概念。这里只是求某个用户白天和晚上等某个时间段停留时间的从高到低进行排序。比如早晨8点到晚上6点之间停留时间最长的可以认为是用户的工作地点。相反，在晚上6点到第二天早上8点这段时间中停留时间最长的就认为是用户的住所。知道了用户的工作地点和住处，就可以做一些推荐了。但是存在两个问题是，其一是一个用户可能在一天中会经过几十甚至上百个基站。那怎么才能知道它在哪个基站下面停留的时间最长呢？其二是一个用户在同一个基站下路过还不止一次。比如某用户，在他公司和家之间有一个基站，他早上上班时路过某基站一次， 中午回家又路过一次，晚上下班又路过一次。这样，他就会在同一个基站中路过很多次。这样在基站的服务器日志中就会记录很多条数据。我们现在要计算用户在哪个基站下停留时间最长，其实就是简单的数据切分，然后进行求和，然后进行join。

为了便于理解，我们模拟了一些简单的日志数据存放在A.txt中，共4个字段：手机号码，时间戳，基站id，连接类型（1表示建立连接，0表示断开连接）：

18688888888,20160327082400,16030401EAFB68F1E3CDF819735E1C66,1

18611132889,20160327082500,16030401EAFB68F1E3CDF819735E1C66,1

18688888888,20160327170000,16030401EAFB68F1E3CDF819735E1C66,0

18611132889,20160327180000,16030401EAFB68F1E3CDF819735E1C66,0

18611132889,20160327075000,9F36407EAD0629FC166F14DDE7970F68,1

18688888888,20160327075100,9F36407EAD0629FC166F14DDE7970F68,1

18611132889,20160327081000,9F36407EAD0629FC166F14DDE7970F68,0

18688888888,20160327081300,9F36407EAD0629FC166F14DDE7970F68,0

18688888888,20160327175000,9F36407EAD0629FC166F14DDE7970F68,1

18611132889,20160327182000,9F36407EAD0629FC166F14DDE7970F68,1

18688888888,20160327220000,9F36407EAD0629FC166F14DDE7970F68,0

18611132889,20160327230000,9F36407EAD0629FC166F14DDE7970F68,0

18611132889,20160327081100,CC0710CC94ECC657A8561DE549D940E0,1

18688888888,20160327081200,CC0710CC94ECC657A8561DE549D940E0,1

18688888888,20160327081900,CC0710CC94ECC657A8561DE549D940E0,0

18611132889,20160327082000,CC0710CC94ECC657A8561DE549D940E0,0

18688888888,20160327171000,CC0710CC94ECC657A8561DE549D940E0,1

18688888888,20160327171600,CC0710CC94ECC657A8561DE549D940E0,0

18611132889,20160327180500,CC0710CC94ECC657A8561DE549D940E0,1

18611132889,20160327181500,CC0710CC94ECC657A8561DE549D940E0,0

下面是基站表的数据，共4个字段，分别代表基站id和经纬度以及信号的辐射类型（比如2G信号、3G信号和4G信号）：

9F36407EAD0629FC166F14DDE7970F68,116.304864,40.050645,6

CC0710CC94ECC657A8561DE549D940E0,116.303955,40.041935,6

16030401EAFB68F1E3CDF819735E1C66,116.296302,40.032296,6

基于以上基站的日志数据，要求计算某个手机号码在一天之内出现过的地点及所呆时长。

思路：求每个手机号码在哪些基站下面停留的时间最长，在计算的时候，用"手机号码+基站"才能定位在哪个基站下面停留的时间， 因为每个基站下面会有很多的用户的日志数据。

全国有很多的基站，每个电信分公司只负责计算自己的数据。数据存放在基站下面的机房的服务器上。一般是用过一些工具通过网络把这些数据搜集过来。搜集过来的数据量可能会很大，这些数据一般会存放到分布式的文件系统中，比如存放到HDFS中。我们可能会基于一周或者一个月的数据量来计算，时间跨度越大，计算出来的结构就越精确。

完整代码如下：

import org.apache.spark.rdd.RDD

import org.apache.spark.{SparkConf, SparkContext}

object mobineNum {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("mobineNum")

// AppName 参数是你的应用程序的名字，你可以在 Spark 内置 UI 上看到它。

conf.setMaster("local")

// Master 是 Spark、Mesos、或者 YARN 集群的 URL，或者使用一个专用的字符串“Local”设定其在本地模式下运行。

val sc = new SparkContext(conf)

//sc是SparkContext，指的是“上下文”，也就是运行的环境，需要把conf当参数传进去

val lines = sc.textFile("A.txt")

//通过sc获取一个文本文件，传入本地文本的路径，将输入文件转换成RDD，path应该是该文本文件在该项目所在文件中的路径。

//切分

val splited = lines.map(line => {

val fields = line.split(",")

//将每行记录以逗号进行分割

val mobile = fields(0)

//其中第一个属性值表示手机号

val lac = fields(2)

//第三个属性值为基站信息

val tp = fields(3)

//第四个属性值为连接状态

val time = if(tp == "1") -fields(1).toLong else fields(1).toLong

//第二个属性值为时间，将其转换为数据类型。

//拼接数据，将其拼接为以下格式组成新的RDD

((mobile, lac), time)

})

//分组聚合，将同一个基站中同一个手机号的时间进行相加

val reduced= splited.reduceByKey(\_+\_)

val lmt = reduced.map(x => {

(x.\_1.\_2, (x.\_1.\_1, x.\_2))

//x.\_1.\_2表示((mobile, lac), time)格式中lac，x.\_1.\_1表示mobine，x.\_2表示time，（基站id，（手机号， 时间））

})

val lacInfo = sc.textFile("B.txt")

//获取各个基站的信息

//整理基站数据

val splitedLacInfo = lacInfo.map(line => {

val fields = line.split(",")

val id = fields(0)

//基站信息中第一个属性值为基站的id

val x = fields(1)

//基站信息中第二个属性值为基站的经度

val y = fields(2)

//基站信息中第二个属性值为基站的纬度

(id, (x, y))

//拼接为（基站id，（经度，纬度））

})

//将两个RDD进行连接join操作

val joined = lmt.join(splitedLacInfo)

println(joined.collect().toBuffer)

sc.stop()

}

}

得到结果如下：

（基站id，（（手机号，所在基站的时长），（基站的经度，基站的纬度）））

ArrayBuffer(

(CC0710CC94ECC657A8561DE549D940E0,((18688888888,1300),(116.303955,40.041935))), (CC0710CC94ECC657A8561DE549D940E0,((18611132889,1900),(116.303955,40.041935))), (9F36407EAD0629FC166F14DDE7970F68,((18611132889,54000),(116.304864,40.050645))), (9F36407EAD0629FC166F14DDE7970F68,((18688888888,51200),(116.304864,40.050645))), (16030401EAFB68F1E3CDF819735E1C66,((18611132889,97500),(116.296302,40.032296))), (16030401EAFB68F1E3CDF819735E1C66,((18688888888,87600),(116.296302,40.032296))))

# 3.7 本章小结

本章重点讲解了Spark编程模型的两种抽象，一种是RDD，另一种是共享变量。第一节和第二节着重讲解了RDD的特征，RDD的依赖说明，以及RDD的缓存机制和检查点机制等；第三四节讲解了通过哪几种方式如何创建RDD以及RDD的基本算子（Transformation算子和Action算子），并给出了基于scala语言的一些实例用于加深读者对于RDD操作的了解以及在不同类型中RDD的转换的实质。最后的综合实例主要为了使读者更好的理解RDD的使用。