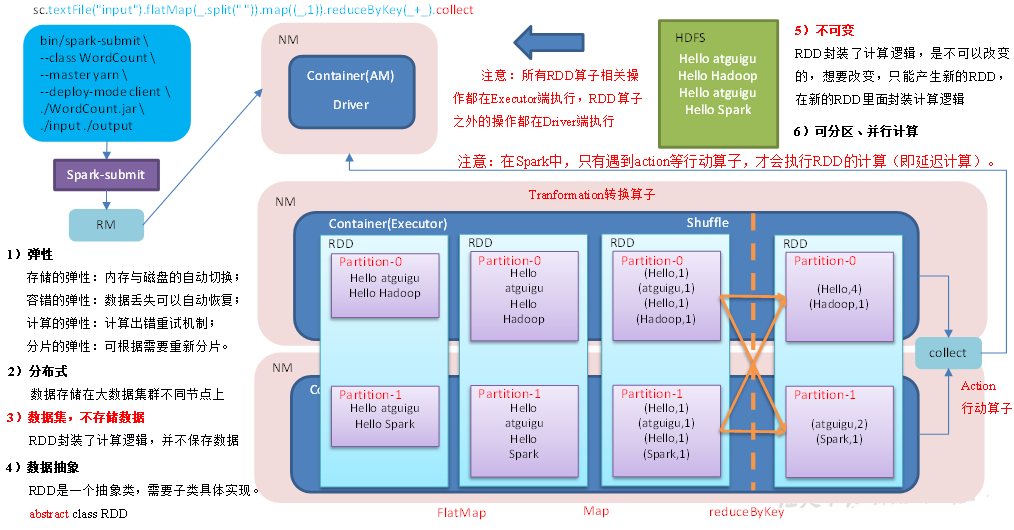
# 第1章 RDD概述

## 1.1 什么是RDD

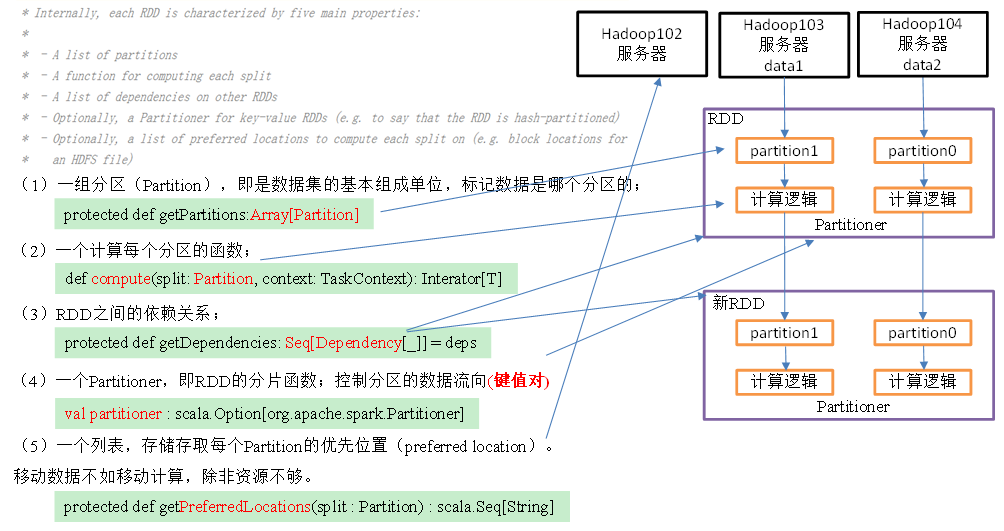
RDD（Resilient Distributed Dataset）叫做弹性分布式数据集，是Spark中最基本的数据抽象。

代码中是一个抽象类，它代表一个弹性的、不可变、可分区、里面的元素可并行计算的集合。

### 1.1.2 WordCount工作流程



## 1.2 RDD五大特性



# 第2章 RDD编程

## 2.1 RDD的创建

在Spark中创建RDD的创建方式可以分为三种：从集合中创建RDD、从外部存储创建RDD、从其他RDD创建。

### 2.1.1 IDEA环境准备

1）创建一个maven工程，工程名称叫SparkCoreTest

2）添加scala框架支持

3）创建一个scala文件夹，并把它修改为Source Root

4）创建包名：com.user1.createrdd

5）在pom文件中添加

<dependencies>  
 <dependency>  
 <groupId>org.apache.spark</groupId>  
 <artifactId>spark-core\_2.12</artifactId>  
 <version>3.0.0</version>  
 </dependency>  
</dependencies>  
<build>  
 <finalName>SparkCoreTest</finalName>  
 <plugins>  
 <plugin>  
 <groupId>net.alchim31.maven</groupId>  
 <artifactId>scala-maven-plugin</artifactId>  
 <version>3.4.6</version>  
 <executions>  
 <execution>  
 <goals>  
 <goal>compile</goal>  
 <goal>testCompile</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>

### 2.1.2 从集合中创建

1）从集合中创建RDD，[Spark](https://www.iteblog.com/archives/tag/spark/" \t "_blank)主要提供了两种函数：parallelize和makeRDD

package com.user1.createrdd  
import org.apache.spark.rdd.RDD  
import org.apache.spark.{SparkConf, SparkContext}  
  
object createrdd01\_array {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3.使用parallelize()创建rdd  
 val rdd: RDD[Int] = sc.parallelize(Array(1, 2, 3, 4, 5, 6, 7, 8))  
  
 rdd.collect().foreach(println)  
  
 //4.使用makeRDD()创建rdd  
 val rdd1: RDD[Int] = sc.makeRDD(Array(1, 2, 3, 4, 5, 6, 7, 8))  
  
 rdd1.collect().foreach(println)  
  
 sc.stop()  
 }  
}

注意：makeRDD有两种重构方法，重构方法一如下，makeRDD和parallelize功能一样。

def makeRDD[T: ClassTag](  
 seq: Seq[T],  
 numSlices: Int = defaultParallelism): RDD[T] = withScope {  
 parallelize(seq, numSlices)  
}

2）makeRDD的重构方法二，增加了位置信息

注意：只需要知道makeRDD不完全等于parallelize即可。

def makeRDD[T: ClassTag](seq: Seq[(T, Seq[String])]): RDD[T] = withScope {  
 assertNotStopped()  
 val indexToPrefs = seq.zipWithIndex.map(t => (t.\_2, t.\_1.\_2)).toMap  
 new ParallelCollectionRDD[T](this, seq.map(\_.\_1), math.max(seq.size, 1), indexToPrefs)  
}

### 2.1.3 从外部存储系统的数据集创建

由外部存储系统的数据集创建RDD包括：本地的文件系统，还有所有Hadoop支持的数据集，比如HDFS、HBase等。

1）数据准备

在新建的SparkCoreTest项目名称上右键=》新建input文件夹=》在input文件夹上右键=》分别新建1.txt和2.txt。每个文件里面准备一些word单词。

2）创建RDD

package com.user1.createrdd

import org.apache.spark.rdd.RDD

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

object createrdd02\_file {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.读取文件。如果是集群路径：hdfs://hadoop102:9000/input

val lineWordRdd: RDD[String] = sc.textFile("input")

//4.打印

lineWordRdd.foreach(println)

//5.关闭

sc.stop()

}

}

### 2.1.4 从其他RDD创建

主要是通过一个RDD运算完后，再产生新的RDD。

## 2.2 分区规则

### 2.2.1 默认分区源码（RDD数据从集合中创建）

1）默认分区数源码解读



2）创建一个包名：com.user1.partition

3）代码验证

package com.user1.partition

import org.apache.spark.rdd.RDD

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

object partition01\_default {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

val rdd: RDD[Int] = sc.makeRDD(Array(1,2,3,4))

//3. 输出数据，产生了8个分区

rdd.saveAsTextFile("output")

//4.关闭连接

sc.stop()

}

}

### 2.2.2 分区源码（RDD数据从集合中创建）

1）分区测试（RDD数据从集合中创建）

object partition02\_Array {

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

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkCoreTest")

val sc: SparkContext = new SparkContext(conf)

//1）4个数据，设置4个分区，输出：0分区->1，1分区->2，2分区->3，3分区->4

//val rdd: RDD[Int] = sc.makeRDD(Array(1, 2, 3, 4), 4)

//2）4个数据，设置3个分区，输出：0分区->1，1分区->2，2分区->3,4

//val rdd: RDD[Int] = sc.makeRDD(Array(1, 2, 3, 4), 3)

//3）5个数据，设置3个分区，输出：0分区->1，1分区->2、3，2分区->4、5

val rdd: RDD[Int] = sc.makeRDD(Array(1, 2, 3, 4, 5), 3)

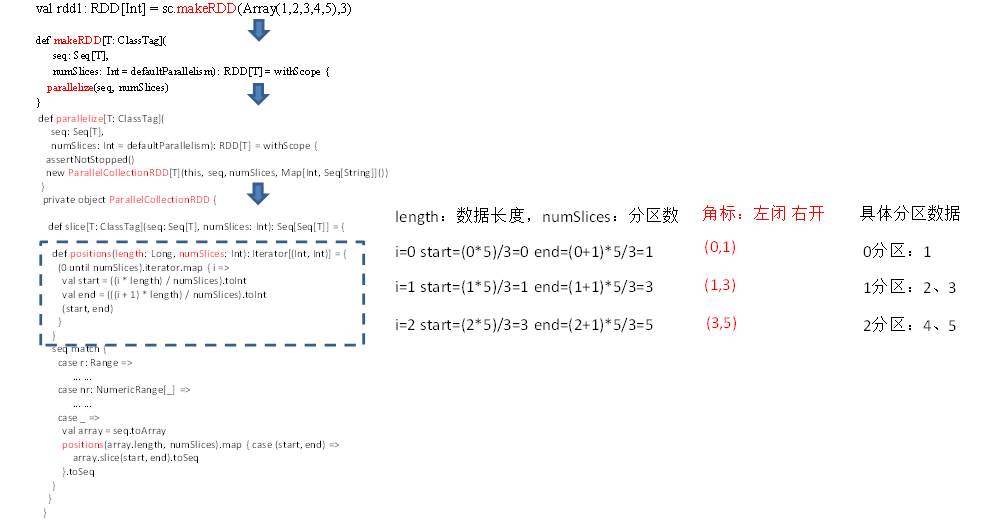
rdd.saveAsTextFile("output")

sc.stop()

}

}

2）分区源码



分区的开始位置 = 分区号 \* 数据总长度/分区总数

分区的结束位置 =（分区号 + 1）\* 数据总长度/分区总数

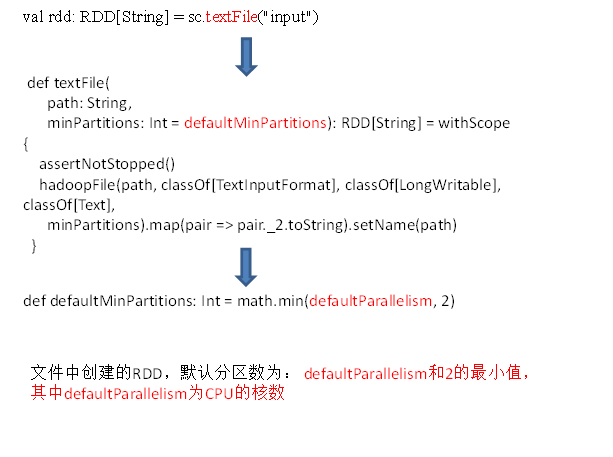
### 2.2.3 默认分区源码（RDD数据从文件中读取后创建）

1）分区测试

object partition03\_file\_default {  
  
 def main(args: Array[String]): Unit = {

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkCoreTest")  
 val sc: SparkContext = new SparkContext(conf)  
  
 //1）默认分区的数量：默认取值为当前核数和2的最小值  
 //val rdd: RDD[String] = sc.textFile("input")  
  
 rdd.saveAsTextFile("output")  
  
 sc.stop()  
 }  
}

2）分区源码



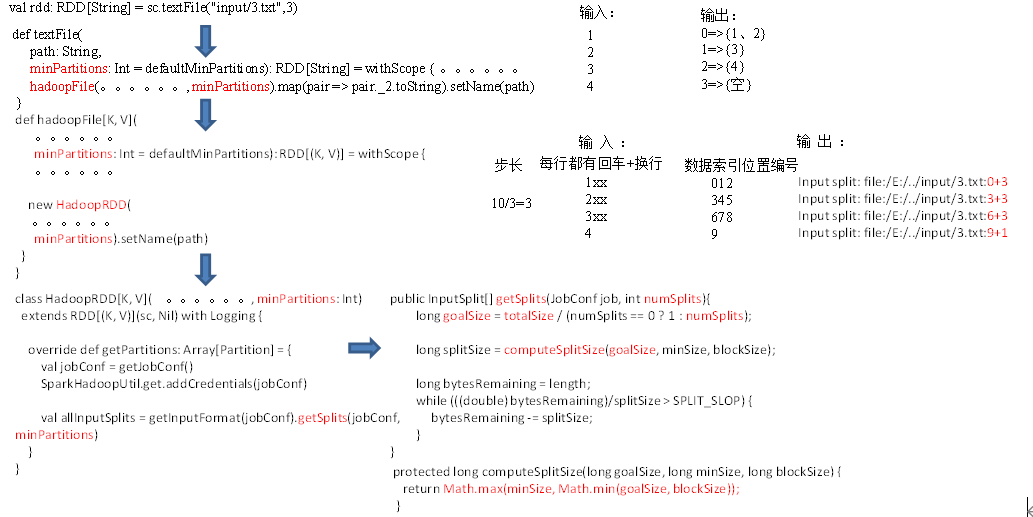
### 2.2.4 分区源码（RDD数据从文件中读取后创建）

1）分区测试

object partition04\_file {  
  
 def main(args: Array[String]): Unit = {

val conf: SparkConf = new SparkConf().setMaster("local[\*]").setAppName("SparkCoreTest")  
 val sc: SparkContext = new SparkContext(conf)  
  
 //1）输入数据1-4，每行一个数字；输出：0=>{1、2} 1=>{3} 2=>{4} 3=>{空}  
 //val rdd: RDD[String] = sc.textFile("input/3.txt", 3)  
  
 rdd.saveAsTextFile("output")  
  
 sc.stop()  
 }  
}

2）源码解析



注意：getSplits文件返回的是切片规划，真正读取是在compute方法中创建LineRecordReader读取的，有两个关键变量： start = split.getStart() end = start + split.getLength

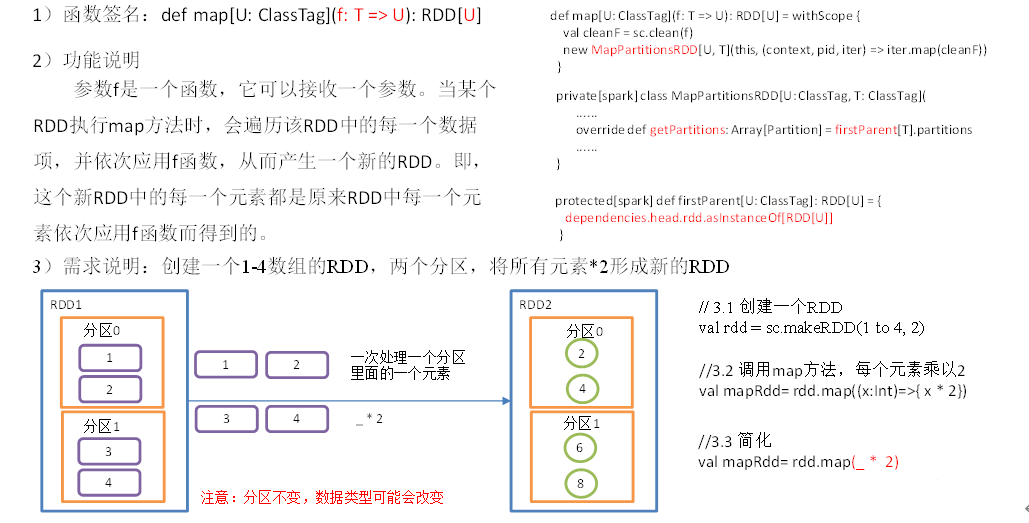
## 2.3 Transformation转换算子（面试开发重点）

RDD整体上分为Value类型、双Value类型和Key-Value类型

### 2.3.1 Value类型

1）创建包名：com.user1.value

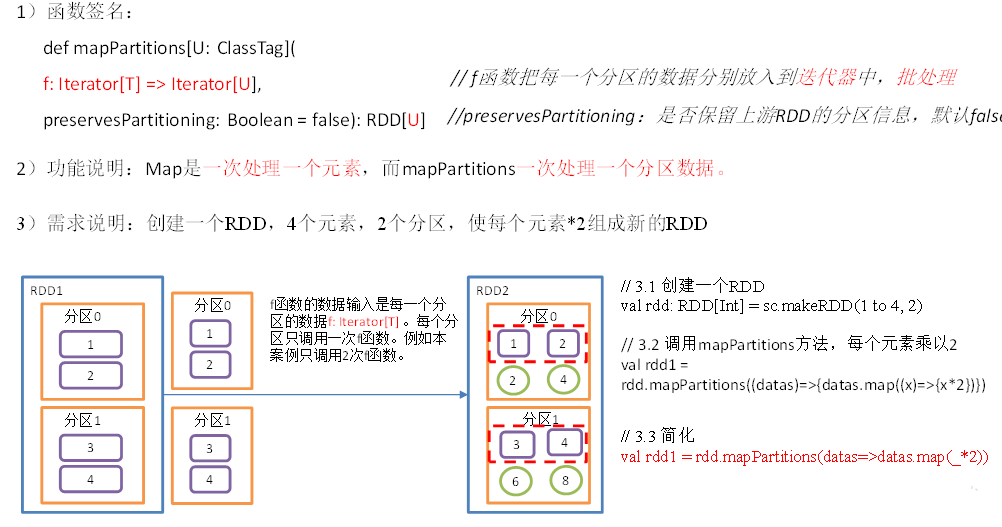
#### 2.3.1.1 map()映射



4）具体实现

object value01\_map {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc = new SparkContext(conf)  
  
 //3具体业务逻辑  
 // 3.1 创建一个RDD  
 val rdd: RDD[Int] = sc.makeRDD(1 to 4, 2)  
  
 // 3.2 调用map方法，每个元素乘以2  
 val mapRdd: RDD[Int] = rdd.map(\_ \* 2)  
  
 // 3.3 打印修改后的RDD中数据  
 mapRdd.collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

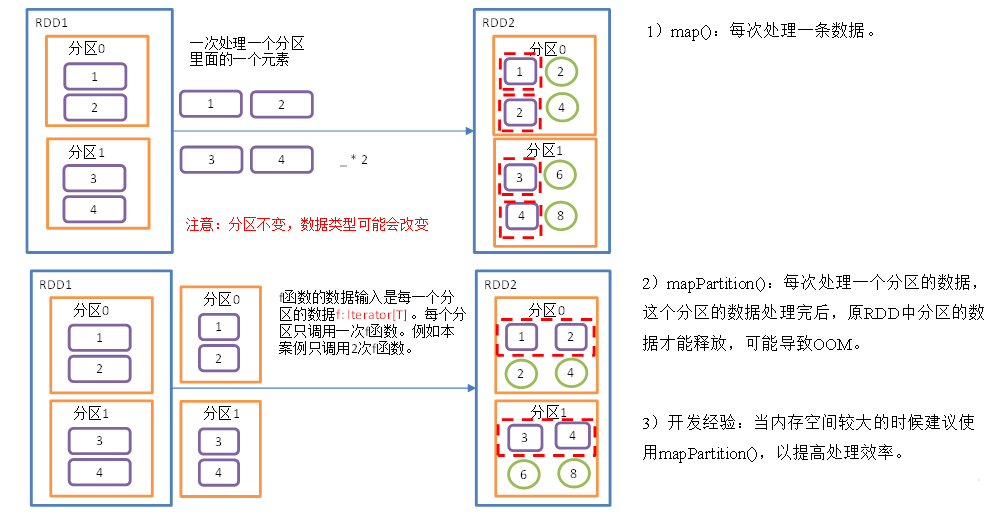
#### 2.3.1.2 mapPartitions()以分区为单位执行Map



4）具体实现

object value02\_mapPartitions {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc = new SparkContext(conf)  
  
 //3具体业务逻辑  
 // 3.1 创建一个RDD  
 val rdd: RDD[Int] = sc.makeRDD(1 to 4, 2)  
  
 // 3.2 调用mapPartitions方法，每个元素乘以2  
 val rdd1 = rdd.mapPartitions(x=>x.map(\_\*2))  
  
 // 3.3 打印修改后的RDD中数据  
 rdd1.collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

#### 2.3.1.3 map()和mapPartitions()区别



#### 2.3.1.4 mapPartitionsWithIndex()带分区号

1）函数签名：

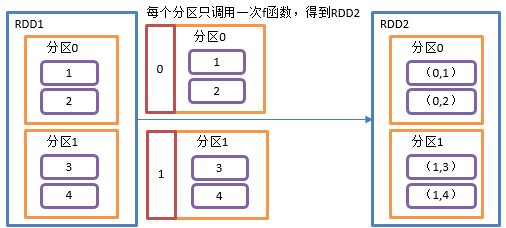
def mapPartitionsWithIndex[U: ClassTag](

f: (Int, Iterator[T]) => Iterator[U], // Int表示分区编号

preservesPartitioning: Boolean = false): RDD[U]

2）功能说明：类似于mapPartitions，比mapPartitions多一个整数参数表示分区号

3）需求说明：创建一个RDD，使每个元素跟所在分区号形成一个元组，组成一个新的RDD



4）具体实现

object value03\_mapPartitionsWithIndex {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc = new SparkContext(conf)  
  
 //3具体业务逻辑  
 // 3.1 创建一个RDD  
 val rdd: RDD[Int] = sc.makeRDD(1 to 4, 2)  
  
 // 3.2 创建一个RDD，使每个元素跟所在分区号形成一个元组，组成一个新的RDD  
 val indexRdd = rdd.mapPartitionsWithIndex( (index,items)=>{items.map( (index,\_) )} )  
  
 // 3.3 打印修改后的RDD中数据  
 indexRdd.collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

#### 2.3.1.5 flatMap()扁平化

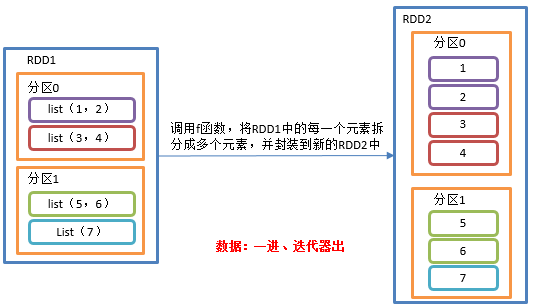
1）函数签名：def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U]

2）功能说明

与map操作类似，将RDD中的每一个元素通过应用f函数依次转换为新的元素，并封装到RDD中。

区别：在flatMap操作中，f函数的返回值是一个集合，并且会将每一个该集合中的元素拆分出来放到新的RDD中。

3）需求说明：创建一个集合，集合里面存储的还是子集合，把所有子集合中数据取出放入到一个大的集合中。



4）具体实现：

object value04\_flatMap {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc = new SparkContext(conf)  
  
 //3具体业务逻辑  
 // 3.1 创建一个RDD  
 val listRDD=sc.makeRDD(List(List(1,2),List(3,4),List(5,6),List(7)), 2)  
  
 // 3.2 把所有子集合中数据取出放入到一个大的集合中  
 listRDD.flatMap(list=>list).collect.foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

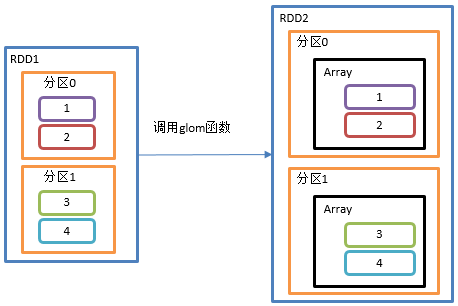
#### 2.3.1.6 glom()分区转换数组

1）函数签名：def glom(): RDD[Array[T]]

2）功能说明

该操作将RDD中每一个分区变成一个数组，并放置在新的RDD中，数组中元素的类型与原分区中元素类型一致

3）需求说明：创建一个2个分区的RDD，并将每个分区的数据放到一个数组，求出每个分区的最大值



4）具体实现

object value05\_glom {

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

//1.创建SparkConf并设置App名称

val conf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc = new SparkContext(conf)

//3具体业务逻辑

// 3.1 创建一个RDD

val rdd = sc.makeRDD(1 to 4, 2)

// 3.2 求出每个分区的最大值 0->1,2 1->3,4

val maxRdd: RDD[Int] = rdd.glom().map(\_.max)

// 3.3 求出所有分区的最大值的和 2 + 4

println(maxRdd.collect().sum)

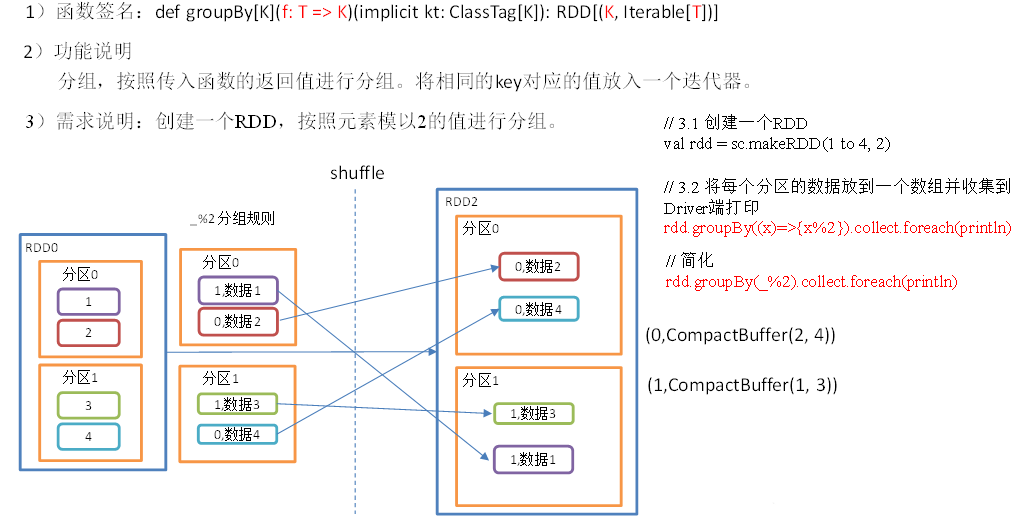
//4.关闭连接

sc.stop()

}

}

#### 2.3.1.7 groupBy()分组



4）具体实现

object value06\_groupby {

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

//1.创建SparkConf并设置App名称

val conf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc = new SparkContext(conf)

//3具体业务逻辑

// 3.1 创建一个RDD

val rdd = sc.makeRDD(1 to 4, 2)

// 3.2 将每个分区的数据放到一个数组并收集到Driver端打印

rdd.groupBy(\_ % 2).collect().foreach(println)

// 3.3 创建一个RDD

val rdd1: RDD[String] = sc.makeRDD(List("hello","hive","hadoop","spark","scala"))

// 3.4 按照首字母第一个单词相同分组

rdd1.groupBy(str=>str.substring(0,1)).collect().foreach(println)

sc.stop()

}

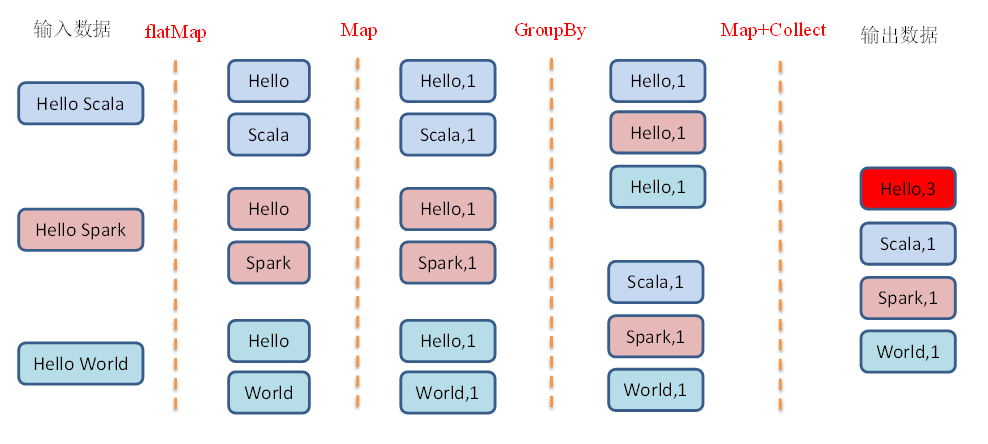
}

groupBy会存在shuffle过程

shuffle：将不同的分区数据进行打乱重组的过程

shuffle一定会落盘。可以在local模式下执行程序，通过4040看效果。

#### 2.3.1.8 GroupBy之WordCount



object value07\_groupby\_wordcount {

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

//1.创建SparkConf并设置App名称

val conf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc = new SparkContext(conf)

//3具体业务逻辑

// 3.1 创建一个RDD

val strList: List[String] = List("Hello Scala", "Hello Spark", "Hello World")

val rdd = sc.makeRDD(strList)

// 3.2 将字符串拆分成一个一个的单词

val wordRdd: RDD[String] = rdd.flatMap(str => str.split(" "))

// 3.3 将单词结果进行转换：word=>(word,1)

val wordToOneRdd: RDD[(String, Int)] = wordRdd.map(word => (word, 1))

// 3.4 将转换结构后的数据分组

val groupRdd: RDD[(String, Iterable[(String, Int)])] = wordToOneRdd.groupBy(t => t.\_1)

// 3.5 将分组后的数据进行结构的转换

// val wordToSum: RDD[(String, Int)] = groupRdd.map(

// t => (t.\_1, t.\_2.toList.size)

// )

// val wordToSum: RDD[(String, Int)] = groupRdd.map {

// x =>

// x match {

// case (word, list) => {

// (word, list.size)

// }

// }

// }

val wordToSum: RDD[(String, Int)] = groupRdd.map {

case (word, list) => {

(word, list.size)

}

}

// 3.6 打印输出

wordToSum.collect().foreach(println)

// 4 关闭资源

sc.stop()

}

}

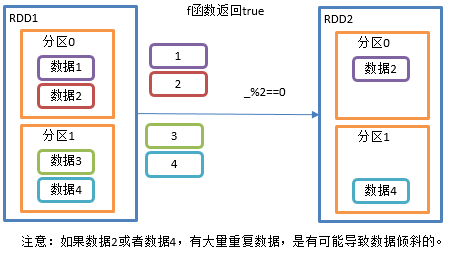
#### 2.3.1.9 filter()过滤

1）函数签名： def filter(f: T => Boolean): RDD[T]

2）功能说明

接收一个返回值为布尔类型的函数作为参数。当某个RDD调用filter方法时，会对该RDD中每一个元素应用f函数，如果返回值类型为true，则该元素会被添加到新的RDD中。

3）需求说明：创建一个RDD，过滤出对2取余等于0的数据



4）代码实现

object value08\_filter {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.创建一个RDD

val rdd: RDD[Int] = sc.makeRDD(Array(1, 2, 3, 4), 2)

//3.1 过滤出符合条件的数据

val filterRdd: RDD[Int] = rdd.filter(\_ % 2 == 0)

//3.2 收集并打印数据

filterRdd.collect().foreach(println)

//4 关闭连接

sc.stop()

}

}

#### 2.3.1.10 sample()采样

1）函数签名：

def sample(

withReplacement: Boolean,

fraction: Double,

seed: Long = Utils.random.nextLong): RDD[T]

// withReplacement： true为有放回的抽样，false为无放回的抽样；

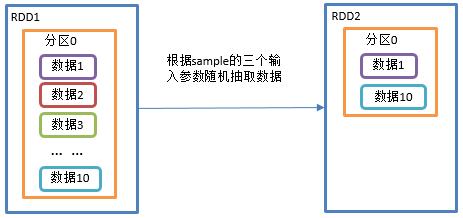
// fraction表示：以指定的随机种子随机抽样出数量为fraction的数据；

// seed表示：指定随机数生成器种子。

2）功能说明

从大量的数据中采样

3）需求说明：创建一个RDD（1-10），从中选择放回和不放回抽样



4）代码实现

object value09\_sample {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.1 创建一个RDD

val dataRDD: RDD[Int] = sc.makeRDD(List(1,2,3,4,5,6))

// 抽取数据不放回（伯努利算法）

// 伯努利算法：又叫0、1分布。例如扔硬币，要么正面，要么反面。

// 具体实现：根据种子和随机算法算出一个数和第二个参数设置几率比较，小于第二个参数要，大于不要

// 第一个参数：抽取的数据是否放回，false：不放回

// 第二个参数：抽取的几率，范围在[0,1]之间,0：全不取；1：全取；

// 第三个参数：随机数种子

val sampleRDD: RDD[Int] = dataRDD.sample(false, 0.5)

sampleRDD.collect().foreach(println)

println("----------------------")

// 抽取数据放回（泊松算法）

// 第一个参数：抽取的数据是否放回，true：放回；false：不放回

// 第二个参数：重复数据的几率，范围大于等于0.表示每一个元素被期望抽取到的次数

// 第三个参数：随机数种子

val sampleRDD1: RDD[Int] = dataRDD.sample(true, 2)

sampleRDD1.collect().foreach(println)

//4.关闭连接

sc.stop()

}

}

5）随机数测试

public class TestRandom {

public static void main(String[] args) {

// 随机算法相同，种子相同，那么随机数就相同

//Random r1 = new Random(100);

// 不输入参数，种子取的当前时间的纳秒值，所以随机结果就不相同了

Random r1 = new Random();

for (int i = 0; i < 5; i++) {

System.out.println(r1.nextInt(10));

}

System.out.println("--------------");

//Random r2 = new Random(100);

Random r2 = new Random();

for (int i = 0; i < 5; i++) {

System.out.println(r2.nextInt(10));

}

}

}

种子相同时的输出结果：

5

0

4

8

1

--------------

5

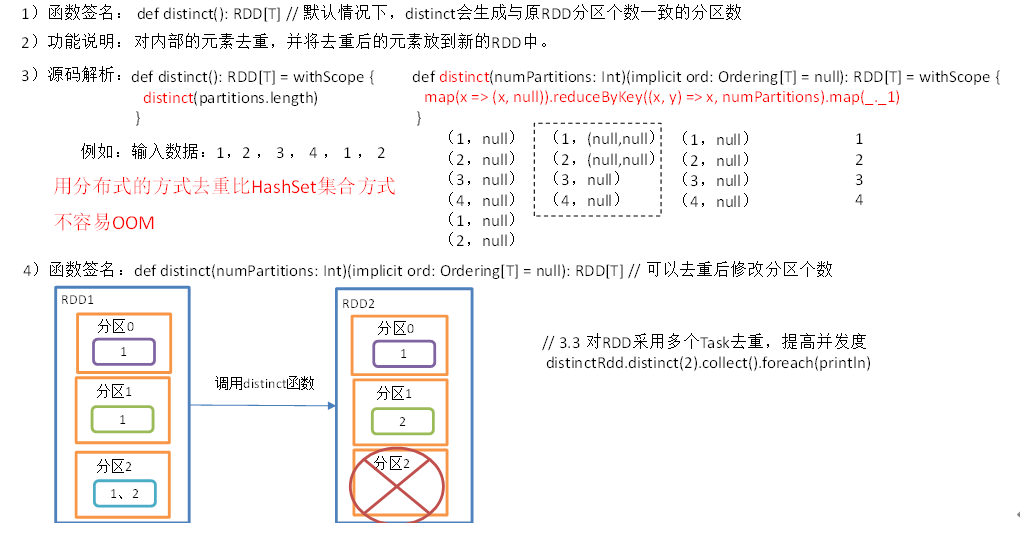
0

4

8

1

#### 2.3.1.11 distinct()去重



4）代码实现

object value10\_distinct {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3具体业务逻辑  
 // 3.1 创建一个RDD  
 val distinctRdd: RDD[Int] = sc.makeRDD(List(1,2,1,5,2,9,6,1))  
  
 // 3.2 打印去重后生成的新RDD  
 distinctRdd.distinct().collect().foreach(println)  
  
 // 3.3 对RDD采用多个Task去重，提高并发度  
 distinctRdd.distinct(2).collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

#### 2.3.1.12 coalesce()合并分区

Coalesce算子包括：配置执行Shuffle和配置不执行Shuffle两种方式。

**1、不执行Shuffle方式**

1）函数签名：

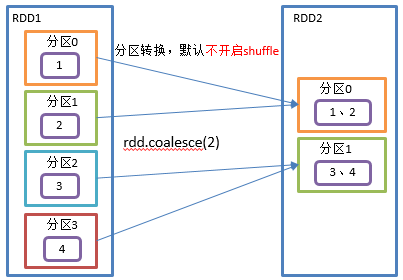
def coalesce(numPartitions: Int, shuffle: Boolean = false, //默认false不执行shuffle

partitionCoalescer: Option[PartitionCoalescer] = Option.empty)

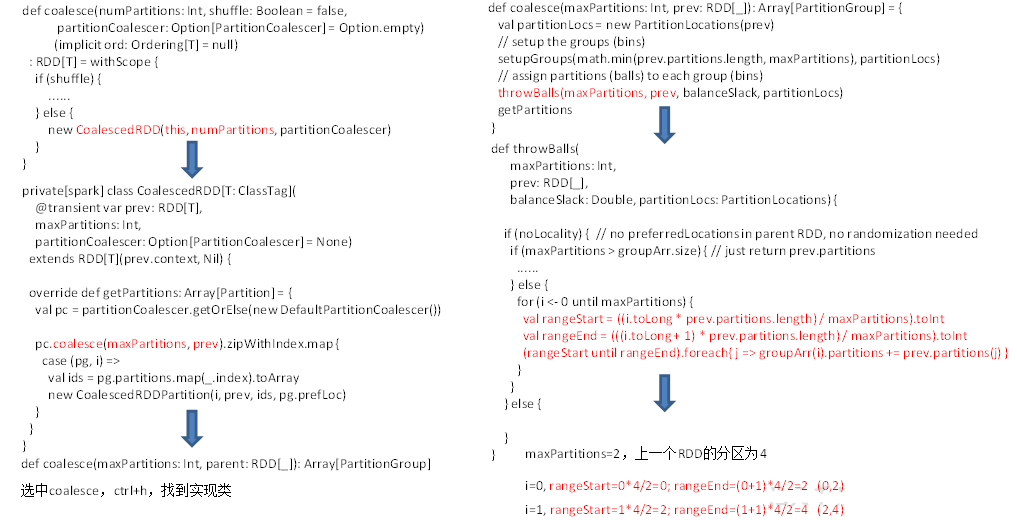
(implicit ord: Ordering[T] = null) : RDD[T]

2）功能说明：缩减分区数，用于大数据集过滤后，提高小数据集的执行效率。

3）需求：4个分区合并为2个分区



4）分区源码



5）代码实现

object value11\_coalesce {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.创建一个RDD

//val rdd: RDD[Int] = sc.makeRDD(Array(1, 2, 3, 4), 4)

//3.1 缩减分区

//val coalesceRdd: RDD[Int] = rdd.coalesce(2)

//4. 创建一个RDD

val rdd: RDD[Int] = sc.makeRDD(Array(1, 2, 3, 4, 5, 6), 3)

//4.1 缩减分区

val coalesceRDD: RDD[Int] = rdd.coalesce(2)

//5 查看对应分区数据

val indexRDD: RDD[(Int, Int)] = coalesceRDD.mapPartitionsWithIndex(

(index, datas) => {

datas.map((index, \_))

}

)

//6 打印数据

indexRDD.collect().foreach(println)

//8 延迟一段时间，观察<http://localhost:4040>页面，查看Shuffle读写时间

Thread.sleep(100000)

//7.关闭连接

sc.stop()

}

}

**2、执行Shuffle方式**

//3. 创建一个RDD  
val rdd: RDD[Int] = sc.makeRDD(Array(1, 2, 3, 4, 5, 6), 3)  
//3.1 执行shuffle  
val coalesceRdd: RDD[Int] = rdd.coalesce(2, true)

输出结果：

(0,1)

(0,4)

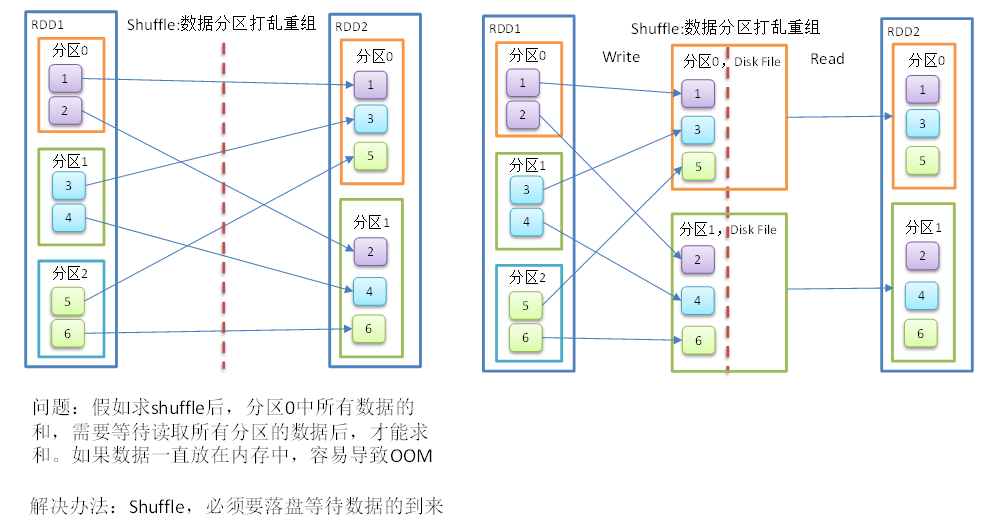
(0,5)

(1,2)

(1,3)

(1,6)

**3、Shuffle原理**



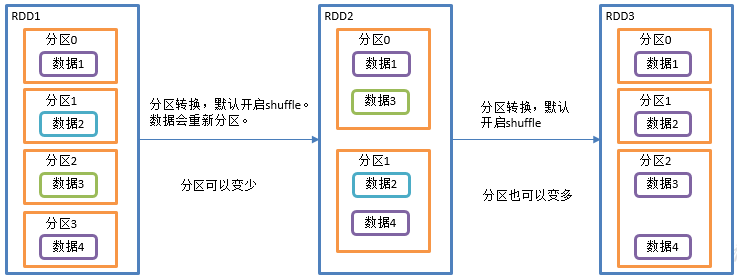
#### 2.3.1.13 repartition()重新分区（执行Shuffle）

1）函数签名： def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T]

2）功能说明

该操作内部其实执行的是coalesce操作，参数shuffle的默认值为true。无论是将分区数多的RDD转换为分区数少的RDD，还是将分区数少的RDD转换为分区数多的RDD，repartition操作都可以完成，因为无论如何都会经shuffle过程。

3）需求说明：创建一个4个分区的RDD，对其重新分区。



4）代码实现

object value12\_repartition {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3. 创建一个RDD

val rdd: RDD[Int] = sc.makeRDD(Array(1, 2, 3, 4, 5, 6), 3)

//3.1 缩减分区

//val coalesceRdd: RDD[Int] = rdd.coalesce(2, true)

//3.2 重新分区

val repartitionRdd: RDD[Int] = rdd.repartition(2)

//4 打印查看对应分区数据

val indexRdd: RDD[(Int, Int)] = repartitionRdd.mapPartitionsWithIndex(

(index, datas) => {

datas.map((index, \_))

}

)

//5 打印

indexRdd.collect().foreach(println)

//6. 关闭连接

sc.stop()

}

}

#### 2.3.1.14 coalesce和repartition区别

1）coalesce重新分区，可以选择是否进行shuffle过程。由参数shuffle: Boolean = false/true决定。

2）repartition实际上是调用的coalesce，进行shuffle。源码如下：

def repartition(numPartitions: Int)(implicit ord: Ordering[T] = null): RDD[T] = withScope {  
 coalesce(numPartitions, shuffle = true)  
}

3）coalesce一般为缩减分区，如果扩大分区，不使用shuffle是没有意义的，repartition扩大分区执行shuffle。

#### 2.3.1.15 sortBy()排序

1）函数签名：

def sortBy[K]( f: (T) => K,

ascending: Boolean = true, // 默认为正序排列

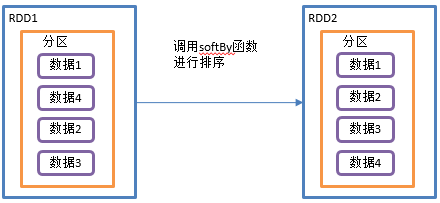
numPartitions: Int = this.partitions.length)

(implicit ord: Ordering[K], ctag: ClassTag[K]): RDD[T]

2）功能说明

该操作用于排序数据。在排序之前，可以将数据通过f函数进行处理，之后按照f函数处理的结果进行排序，默认为正序排列。排序后新产生的RDD的分区数与原RDD的分区数一致。

3）需求说明：创建一个RDD，按照数字大小分别实现正序和倒序排序



4）代码实现：

object value13\_sortBy {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

// 3.1 创建一个RDD

val rdd: RDD[Int] = sc.makeRDD(List(2, 1, 3, 4, 6, 5))

// 3.2 默认是升序排

val sortRdd: RDD[Int] = rdd.sortBy(num => num)

sortRdd.collect().foreach(println)

// 3.3 配置为倒序排

val sortRdd2: RDD[Int] = rdd.sortBy(num => num, false)

sortRdd2.collect().foreach(println)

// 3.4 创建一个RDD

val strRdd: RDD[String] = sc.makeRDD(List("1", "22", "12", "2", "3"))

// 3.5 按照字符的int值排序

strRdd.sortBy(num => num.toInt).collect().foreach(println)

// 3.5 创建一个RDD

val rdd3: RDD[(Int, Int)] = sc.makeRDD(List((2, 1), (1, 2), (1, 1), (2, 2)))

// 3.6 先按照tuple的第一个值排序，相等再按照第2个值排

rdd3.sortBy(t=>t).collect().foreach(println)

//4.关闭连接

sc.stop()

}

}

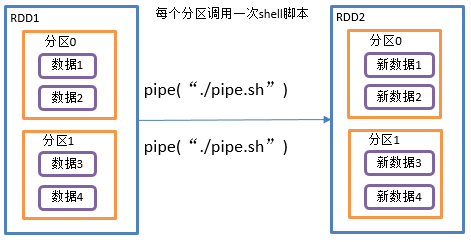
#### 2.3.1.16 pipe()调用脚本

1）函数签名： def pipe(command: String): RDD[String]

2）功能说明

管道，针对每个分区，都调用一次shell脚本，返回输出的RDD。

注意：在Worker节点可以访问到的位置脚本需要放



3）需求说明：编写一个脚本，使用管道将脚本作用于RDD上。

（1）编写一个脚本，并增加执行权限

[user1@hadoop102 spark-local]$ vim pipe.sh

#!/bin/bash

echo "Start"

while read LINE; do

echo ">>>"${LINE}

done

[user1@hadoop102 spark-local]$ chmod 777 pipe.sh

（2）创建一个只有一个分区的RDD

[user1@hadoop102 spark-local]$ bin/spark-shell

scala> val rdd = sc.makeRDD (List("hi","Hello","how","are","you"), 1)

（3）将脚本作用该RDD并打印

scala> rdd.pipe("/opt/module/spark-local/pipe.sh").collect()

res18: Array[String] = Array(Start, >>>hi, >>>Hello, >>>how, >>>are, >>>you)

（4）创建一个有两个分区的RDD

scala> val rdd = sc.makeRDD(List("hi","Hello","how","are","you"), 2)

（5）将脚本作用该RDD并打印

scala> rdd.pipe("/opt/module/spark-local/pipe.sh").collect()

res19: Array[String] = Array(Start, >>>hi, >>>Hello, Start, >>>how, >>>are, >>>you)

说明：一个分区调用一次脚本。

### 2.3.2 双Value类型交互

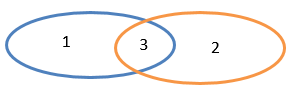
1）创建包名：com.user1.doublevalue

#### 2.3.2.1 intersection()交集

1）函数签名：def intersection(other: RDD[T]): RDD[T]

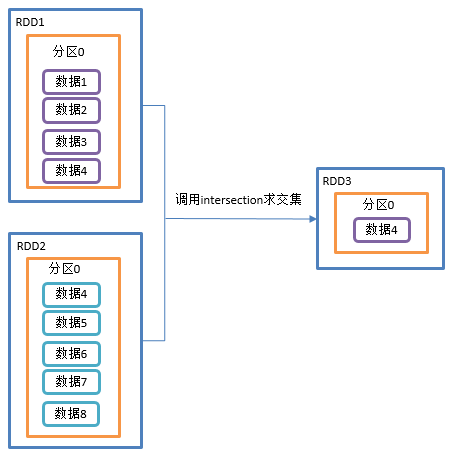
2）功能说明

对源RDD和参数RDD求交集后返回一个新的RDD



交集：只有3

3）需求说明：创建两个RDD，求两个RDD的交集



4）代码实现：

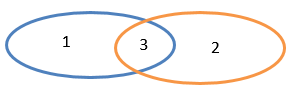
object DoubleValue01\_intersection {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3具体业务逻辑  
 //3.1 创建第一个RDD  
 val rdd1: RDD[Int] = sc.makeRDD(1 to 4)  
  
 //3.2 创建第二个RDD  
 val rdd2: RDD[Int] = sc.makeRDD(4 to 8)  
  
 //3.3 计算第一个RDD与第二个RDD的交集并打印  
 rdd1.intersection(rdd2).collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

#### 2.3.2.2 union()并集

1）函数签名：def union(other: RDD[T]): RDD[T]

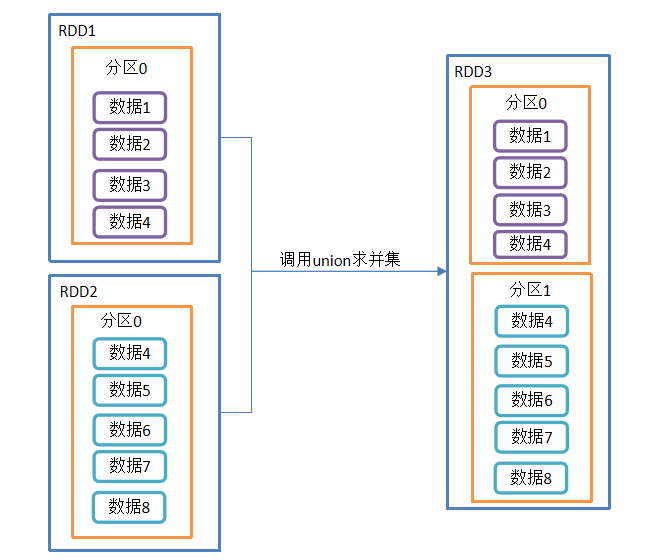
2）功能说明

对源RDD和参数RDD求并集后返回一个新的RDD



并集：1、2、3全包括

3）需求说明：创建两个RDD，求并集



4）代码实现：

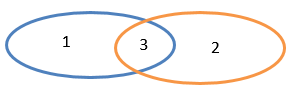
object DoubleValue02\_union {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3具体业务逻辑  
 //3.1 创建第一个RDD  
 val rdd1: RDD[Int] = sc.makeRDD(1 to 4)  
  
 //3.2 创建第二个RDD  
 val rdd2: RDD[Int] = sc.makeRDD(4 to 8)  
  
 //3.3 计算两个RDD的并集  
 rdd1.union(rdd2).collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

#### 2.3.2.3 subtract()差集

1）函数签名：def subtract(other: RDD[T]): RDD[T]

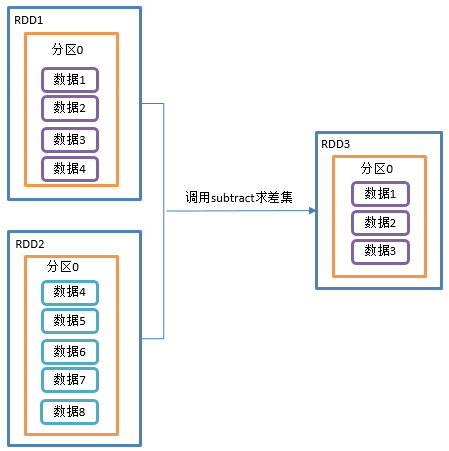
2）功能说明

计算差的一种函数，去除两个RDD中相同元素，不同的RDD将保留下来



差集：只有1

3）需求说明：创建两个RDD，求第一个RDD与第二个RDD的差集



4）代码实现：

object DoubleValue03\_subtract {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3具体业务逻辑  
 //3.1 创建第一个RDD  
 val rdd: RDD[Int] = sc.makeRDD(1 to 4)  
  
 //3.2 创建第二个RDD  
 val rdd1: RDD[Int] = sc.makeRDD(4 to 8)  
  
 //3.3 计算第一个RDD与第二个RDD的差集并打印  
 rdd.subtract(rdd1).collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

#### 2.3.2.4 zip()拉链

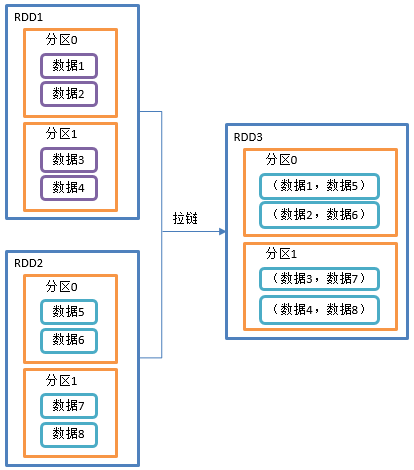
1）函数签名：def zip[U: ClassTag](other: RDD[U]): RDD[(T, U)]

2）功能说明

该操作可以将两个RDD中的元素，以键值对的形式进行合并。其中，键值对中的Key为第1个RDD中的元素，Value为第2个RDD中的元素。

将两个RDD组合成Key/Value形式的RDD，这里默认两个RDD的partition数量以及元素数量都相同，否则会抛出异常。

3）需求说明：创建两个RDD，并将两个RDD组合到一起形成一个(k,v)RDD



4）代码实现：

object DoubleValue04\_zip {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd1: RDD[Int] = sc.makeRDD(Array(1,2,3),3)

//3.2 创建第二个RDD

val rdd2: RDD[String] = sc.makeRDD(Array("a","b","c"),3)

//3.3 第一个RDD组合第二个RDD并打印

rdd1.zip(rdd2).collect().foreach(println)

//3.4 第二个RDD组合第一个RDD并打印

rdd2.zip(rdd1).collect().foreach(println)

//3.5 创建第三个RDD（与1，2分区数不同）

val rdd3: RDD[String] = sc.makeRDD(Array("a","b"), 3)

//3.6 元素个数不同，不能拉链

// Can only zip RDDs with same number of elements in each partition

rdd1.zip(rdd3).collect().foreach(println)

//3.7 创建第四个RDD（与1，2分区数不同）

val rdd4: RDD[String] = sc.makeRDD(Array("a","b","c"), 2)

//3.8 分区数不同，不能拉链

// Can't zip RDDs with unequal numbers of partitions: List(3, 2)

rdd1.zip(rdd4).collect().foreach(println)

//4.关闭连接

sc.stop()

}

}

### 2.3.3 Key-Value类型

1）创建包名：com.user1.keyvalue

#### 2.3.3.1 partitionBy()按照K重新分区

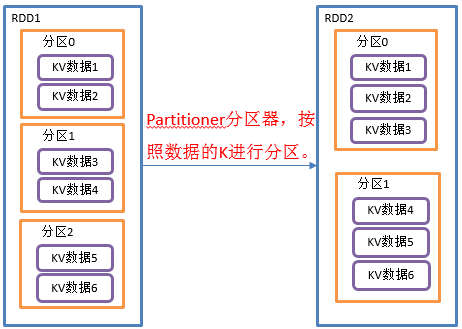
1）函数签名：def partitionBy(partitioner: Partitioner): RDD[(K, V)]

2）功能说明

将RDD[K,V]中的K按照指定Partitioner重新进行分区；

如果原有的RDD和新的RDD是一致的话就不进行分区，否则会产生Shuffle过程。

3）需求说明：创建一个3个分区的RDD，对其重新分区



4）代码实现：

object KeyValue01\_partitionBy {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3具体业务逻辑  
 //3.1 创建第一个RDD  
 val rdd: RDD[(Int, String)] = sc.makeRDD(Array((1,"aaa"),(2,"bbb"),(3,"ccc")),3)  
  
 //3.2 对RDD重新分区  
 val rdd2: RDD[(Int, String)] = rdd.partitionBy(new org.apache.spark.HashPartitioner(2))  
  
 //3.3 打印查看对应分区数据 (0,(2,bbb)) (1,(1,aaa)) (1,(3,ccc))

val indexRdd = rdd2.mapPartitionsWithIndex(

(index, datas) => datas.map((index,\_))

)

indexRdd.collect().foreach(println)

//4.关闭连接  
 sc.stop()  
 }  
}

#### 2.3.3.2 自定义分区

1）HashPartitioner源码解读

class HashPartitioner(partitions: Int) extends Partitioner {

require(partitions >= 0, s"Number of partitions ($partitions) cannot be negative.")

def numPartitions: Int = partitions

def getPartition(key: Any): Int = key match {

case null => 0

case \_ => Utils.nonNegativeMod(key.hashCode, numPartitions)

}

override def equals(other: Any): Boolean = other match {

case h: HashPartitioner =>

h.numPartitions == numPartitions

case \_ =>

false

}

override def hashCode: Int = numPartitions

}

2）自定义分区器

要实现自定义分区器，需要继承org.apache.spark.Partitioner类，并实现下面三个方法。

（1）numPartitions: Int:返回创建出来的分区数。

（2）getPartition(key: Any): Int:返回给定键的分区编号（0到numPartitions-1）。

（3）equals():Java 判断相等性的标准方法。这个方法的实现非常重要，Spark需要用这个方法来检查你的分区器对象是否和其他分区器实例相同，这样Spark才可以判断两个RDD的分区方式是否相同

object KeyValue01\_partitionBy {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd: RDD[(Int, String)] = sc.makeRDD(Array((1, "aaa"), (2, "bbb"), (3, "ccc")), 3)

//3.2 自定义分区

val rdd3: RDD[(Int, String)] = rdd.partitionBy(new MyPartitioner(2))

//4 打印查看对应分区数据

val indexRdd = rdd3.mapPartitionsWithIndex(

(index, datas) => datas.map((index,\_))

)

indexRdd.collect()

//5.关闭连接

sc.stop()

}

}

// 自定义分区

class MyPartitioner(num: Int) extends Partitioner {

// 设置的分区数

override def numPartitions: Int = num

// 具体分区逻辑

override def getPartition(key: Any): Int = {

if (key.isInstanceOf[Int]) {

val keyInt: Int = key.asInstanceOf[Int]

if (keyInt % 2 == 0)

0

else

1

}else{

0

}

}

}

#### 2.3.3.3 reduceByKey()按照K聚合V

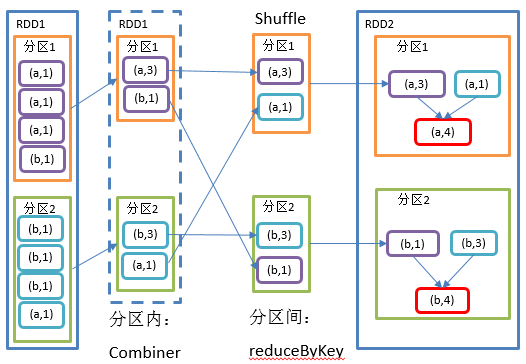
1）函数签名：

def reduceByKey(func: (V, V) => V): RDD[(K, V)]

def reduceByKey(func: (V, V) => V, numPartitions: Int): RDD[(K, V)]

2）功能说明：该操作可以将RDD[K,V]中的元素按照相同的K对V进行聚合。其存在多种重载形式，还可以设置新RDD的分区数。

3）需求说明：统计单词出现次数



4）代码实现：

object KeyValue02\_reduceByKey {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3具体业务逻辑  
 //3.1 创建第一个RDD  
 val rdd = sc.makeRDD(List(("a",1),("b",5),("a",5),("b",2)))  
  
 //3.2 计算相同key对应值的相加结果  
 val reduce: RDD[(String, Int)] = rdd.reduceByKey((v1,v2) => v1+v2)  
  
 //3.3 打印结果  
 reduce.collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

#### 2.3.3.4 groupByKey()按照K重新分组

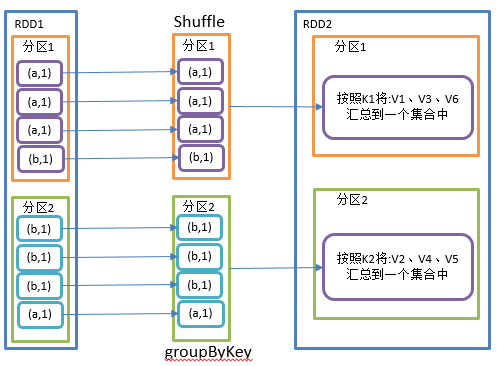
1）函数签名：def groupByKey(): RDD[(K, Iterable[V])]

2）功能说明

groupByKey对每个key进行操作，但只生成一个seq，并不进行聚合。

该操作可以指定分区器或者分区数（默认使用HashPartitioner）

3）需求说明：统计单词出现次数（重画一下图）



4）代码实现：

object KeyValue03\_groupByKey {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd = sc.makeRDD(List(("a",1),("b",5),("a",5),("b",2)))

//3.2 将相同key对应值聚合到一个Seq中

val group: RDD[(String, Iterable[Int])] = rdd.groupByKey()

//3.3 打印结果

group.collect().foreach(println)

//3.4 计算相同key对应值的相加结果

group.map(t=>(t.\_1,t.\_2.sum)).collect().foreach(println)

//4.关闭连接

sc.stop()

}

}

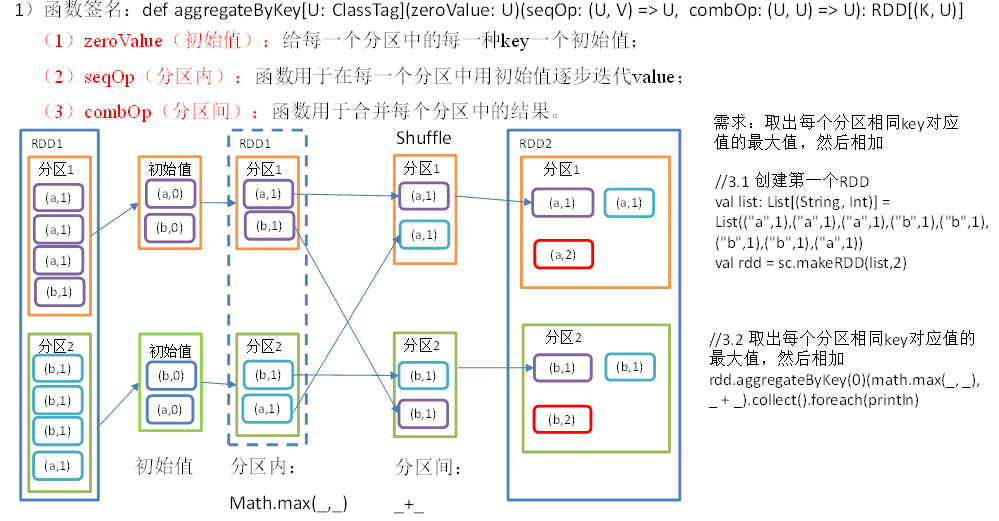
#### 2.3.3.5 reduceByKey和groupByKey区别

1）reduceByKey：按照key进行聚合，在shuffle之前有combine（预聚合）操作，返回结果是RDD[K,V]。

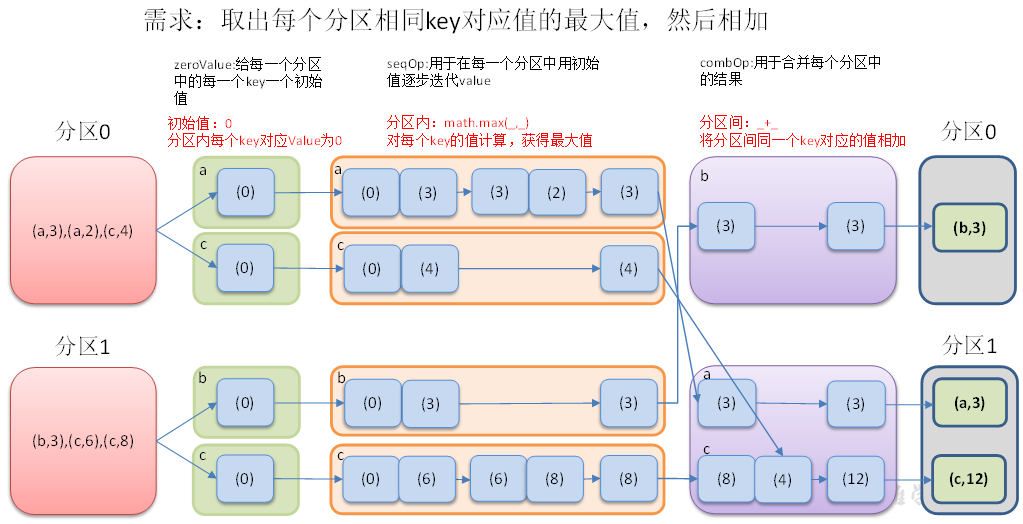
2）groupByKey：按照key进行分组，直接进行shuffle。

3）开发指导：在不影响业务逻辑的前提下，优先选用reduceByKey。求和操作不影响业务逻辑，求平均值影响业务逻辑。

#### 2.3.3.6 aggregateByKey()按照K处理分区内和分区间逻辑



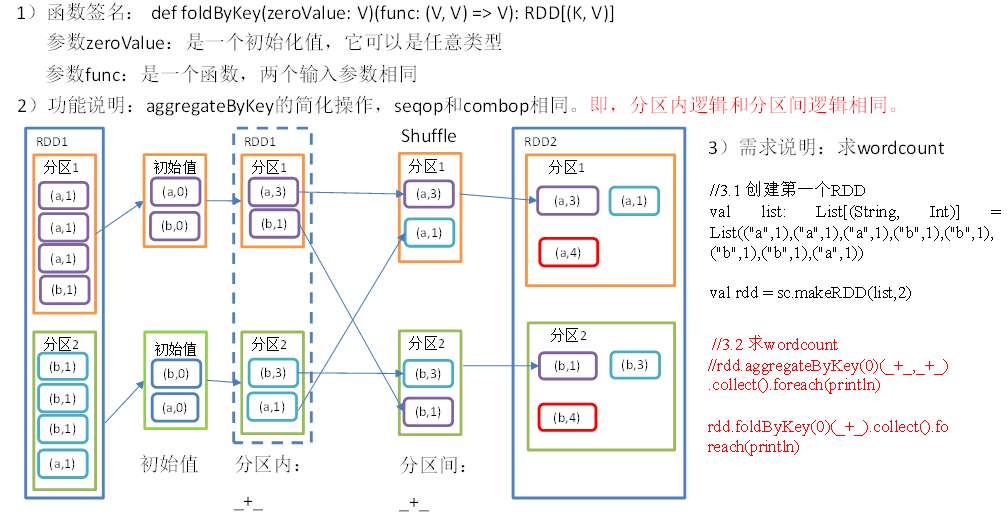
2）需求分析



3）代码实现：

object KeyValue04\_aggregateByKey {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3具体业务逻辑  
 //3.1 创建第一个RDD  
 val rdd: RDD[(String, Int)] = sc.makeRDD(List(("a", 3), ("a", 2), ("c", 4), ("b", 3), ("c", 6), ("c", 8)), 2)  
  
 //3.2 取出每个分区相同key对应值的最大值，然后相加  
 rdd.aggregateByKey(0)(math.max(\_, \_), \_ + \_).collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

#### 2.3.3.7 foldByKey()分区内和分区间相同的aggregateByKey()



4）代码实现：

object KeyValue05\_foldByKey {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val list: List[(String, Int)] = List(("a",1),("a",1),("a",1),("b",1),("b",1),("b",1),("b",1),("a",1))

val rdd = sc.makeRDD(list,2)

//3.2 求wordcount

//rdd.aggregateByKey(0)(\_+\_,\_+\_).collect().foreach(println)

rdd.foldByKey(0)(\_+\_).collect().foreach(println)

//4.关闭连接

sc.stop()

}

}

#### 2.3.3.8 combineByKey()转换结构后分区内和分区间操作

1）函数签名：

def combineByKey[C](

createCombiner: V => C,

mergeValue: (C, V) => C,

mergeCombiners: (C, C) => C): RDD[(K, C)]

（1）createCombiner（转换数据的结构）: combineByKey() 会遍历分区中的所有元素，因此每个元素的键要么还没有遇到过，要么就和之前的某个元素的键相同。如果这是一个新的元素，combineByKey()会使用一个叫作createCombiner()的函数来创建那个键对应的累加器的初始值

（2）mergeValue（分区内）: 如果这是一个在处理当前分区之前已经遇到的键，它会使用mergeValue()方法将该键的累加器对应的当前值与这个新的值进行合并

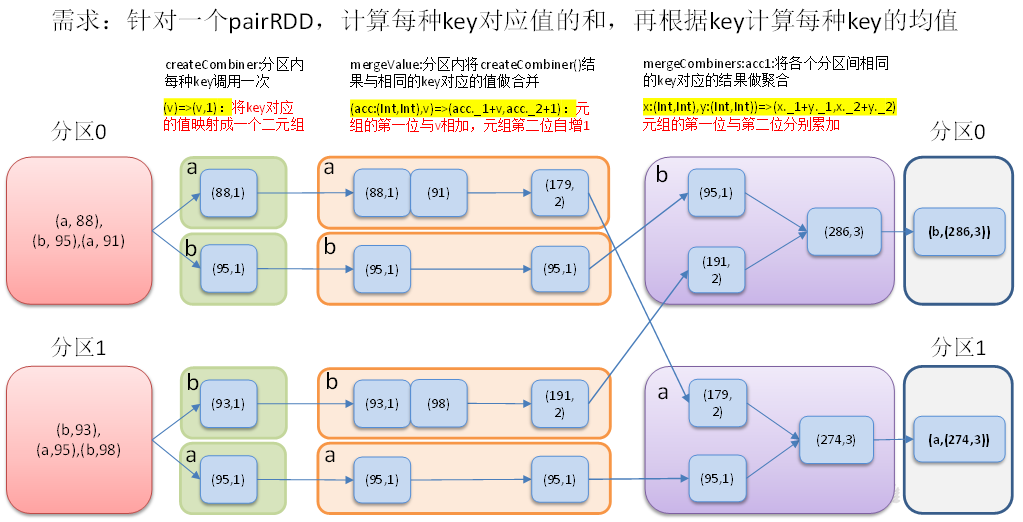
（3）mergeCombiners（分区间）: 由于每个分区都是独立处理的，因此对于同一个键可以有多个累加器。如果有两个或者更多的分区都有对应同一个键的累加器，就需要使用用户提供的mergeCombiners()方法将各个分区的结果进行合并。

2）功能说明

针对相同K，将V合并成一个集合。

3）需求说明：创建一个pairRDD，根据key计算每种key的均值。（先计算每个key出现的次数以及可以对应值的总和，再相除得到结果）

4）需求分析：



5）代码实现

object KeyValue06\_combineByKey {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.1 创建第一个RDD

val list: List[(String, Int)] = List(("a", 88), ("b", 95), ("a", 91), ("b", 93), ("a", 95), ("b", 98))

val input: RDD[(String, Int)] = sc.makeRDD(list, 2)

//3.2 将相同key对应的值相加，同时记录该key出现的次数，放入一个二元组

val combineRdd: RDD[(String, (Int, Int))] = input.combineByKey(

(\_, 1),

(acc: (Int, Int), v) => (acc.\_1 + v, acc.\_2 + 1),

(acc1: (Int, Int), acc2: (Int, Int)) => (acc1.\_1 + acc2.\_1, acc1.\_2 + acc2.\_2)

)

//3.3 打印合并后的结果

combineRdd.collect().foreach(println)

//3.4 计算平均值

combineRdd.map {

case (key, value) => {

(key, value.\_1 / value.\_2.toDouble)

}

}.collect().foreach(println)

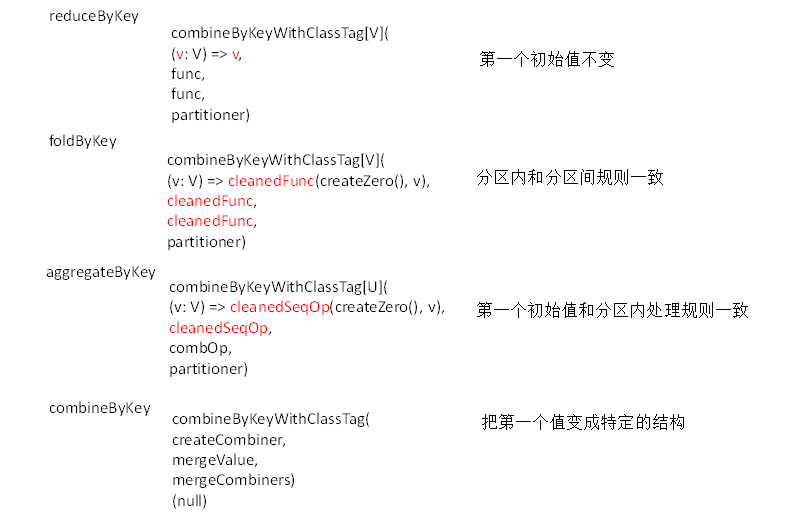
//4.关闭连接

sc.stop()

}

}

#### 2.3.3.9 reduceByKey、foldByKey、aggregateByKey、combineByKey



#### 2.3.3.10 sortByKey()按照K进行排序

1）函数签名：

def sortByKey(

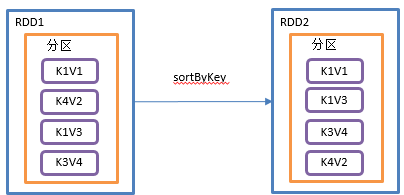
ascending: Boolean = true, // 默认，升序

numPartitions: Int = self.partitions.length) : RDD[(K, V)]

2）功能说明

在一个(K,V)的RDD上调用，K必须实现Ordered接口，返回一个按照key进行排序的(K,V)的RDD

3）需求说明：创建一个pairRDD，按照key的正序和倒序进行排序



4）代码实现：

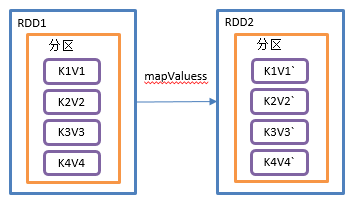
object KeyValue07\_sortByKey {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3具体业务逻辑  
 //3.1 创建第一个RDD  
 val rdd: RDD[(Int, String)] = sc.makeRDD(Array((3,"aa"),(6,"cc"),(2,"bb"),(1,"dd")))  
  
 //3.2 按照key的正序（默认顺序）  
 rdd.sortByKey(true).collect().foreach(println)  
  
 //3.3 按照key的倒序  
 rdd.sortByKey(false).collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

#### 2.3.3.11 mapValues()只对V进行操作

1）函数签名：def mapValues[U](f: V => U): RDD[(K, U)]

2）功能说明：针对于(K,V)形式的类型只对V进行操作

3）需求说明：创建一个pairRDD，并将value添加字符串"|||"



4）代码实现：

object KeyValue08\_mapValues {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3具体业务逻辑  
 //3.1 创建第一个RDD  
 val rdd: RDD[(Int, String)] = sc.makeRDD(Array((1, "a"), (1, "d"), (2, "b"), (3, "c")))  
  
 //3.2 对value添加字符串"|||"  
 rdd.mapValues(\_ + "|||").collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

#### 2.3.3.12 join()连接

1）函数签名：

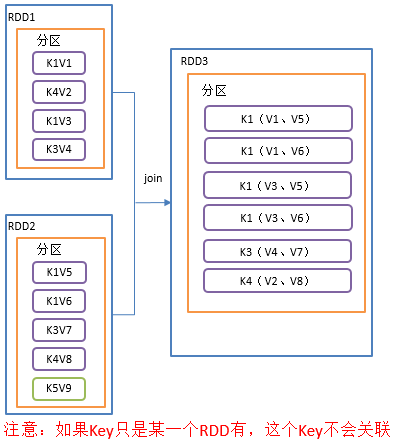
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]

def join[W](other: RDD[(K, W)], numPartitions: Int): RDD[(K, (V, W))]

2）功能说明

在类型为(K,V)和(K,W)的RDD上调用，返回一个相同key对应的所有元素对在一起的(K,(V,W))的RDD

3）需求说明：创建两个pairRDD，并将key相同的数据聚合到一个元组。



4）代码实现：

object KeyValue09\_join {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3具体业务逻辑  
 //3.1 创建第一个RDD  
 val rdd: RDD[(Int, String)] = sc.makeRDD(Array((1, "a"), (2, "b"), (3, "c")))  
  
 //3.2 创建第二个pairRDD  
 val rdd1: RDD[(Int, Int)] = sc.makeRDD(Array((1, 4), (2, 5), (4, 6)))  
  
 //3.3 join操作并打印结果  
 rdd.join(rdd1).collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

#### 2.3.3.13 cogroup()类似全连接，但是在同一个RDD中对key聚合

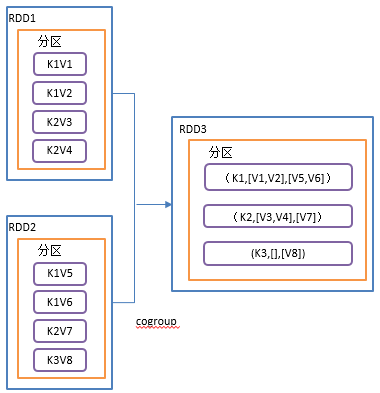
1）函数签名：def cogroup[W](other: RDD[(K, W)]): RDD[(K, (Iterable[V], Iterable[W]))]

2）功能说明

在类型为(K,V)和(K,W)的RDD上调用，返回一个(K,(Iterable<V>,Iterable<W>))类型的RDD

操作两个RDD中的KV元素，每个RDD中相同key中的元素分别聚合成一个集合。

3）需求说明：创建两个pairRDD，并将key相同的数据聚合到一个迭代器。



4）代码实现：

object KeyValue10\_cogroup {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3具体业务逻辑  
 //3.1 创建第一个RDD  
 val rdd: RDD[(Int, String)] = sc.makeRDD(Array((1,"a"),(2,"b"),(3,"c")))  
  
 //3.2 创建第二个RDD  
 val rdd1: RDD[(Int, Int)] = sc.makeRDD(Array((1,4),(2,5),(4,6)))  
  
 //3.3 cogroup两个RDD并打印结果

// (1,(CompactBuffer(a),CompactBuffer(4)))

// (2,(CompactBuffer(b),CompactBuffer(5)))

// (3,(CompactBuffer(c),CompactBuffer()))

// (4,(CompactBuffer(),CompactBuffer(6)))  
 rdd.cogroup(rdd1).collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

## 2.4 Action行动算子

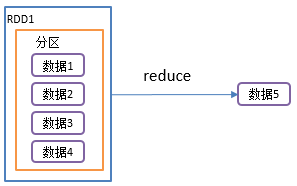
行动算子是触发了整个作业的执行。因为转换算子都是懒加载，并不会立即执行。

1）创建包名：com.user1.action

### 2.4.1 reduce()聚合

1）函数签名：def reduce(f: (T, T) => T): T

2）功能说明：f函数聚集RDD中的所有元素，先聚合分区内数据，再聚合分区间数据。



3）需求说明：创建一个RDD，将所有元素聚合得到结果

object action01\_reduce {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

//3.2 聚合数据

val reduceResult: Int = rdd.reduce(\_+\_)

println(reduceResult)

//4.关闭连接

sc.stop()

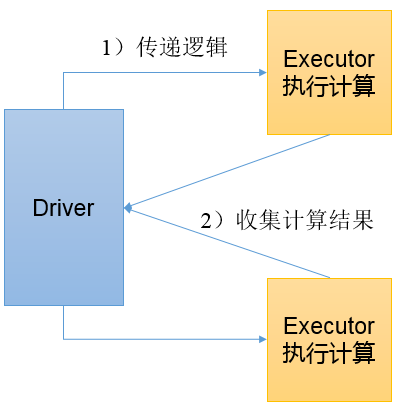
}

}

### 2.4.2 collect()以数组的形式返回数据集

1）函数签名：def collect(): Array[T]

2）功能说明：在驱动程序中，以数组Array的形式返回数据集的所有元素。



注意：所有的数据都会被拉取到Driver端，慎用

3）需求说明：创建一个RDD，并将RDD内容收集到Driver端打印

object action02\_collect {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

//3.2 收集数据到Driver

rdd.collect().foreach(println)

//4.关闭连接

sc.stop()

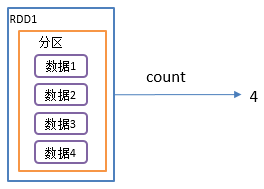
}

}

### 2.4.3 count()返回RDD中元素个数

1）函数签名：def count(): Long

2）功能说明：返回RDD中元素的个数



3）需求说明：创建一个RDD，统计该RDD的条数

object action03\_count {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

//3.2 返回RDD中元素的个数

val countResult: Long = rdd.count()

println(countResult)

//4.关闭连接

sc.stop()

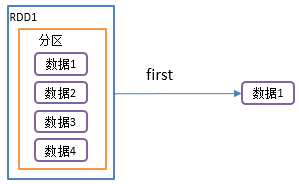
}

}

### 2.4.4 first()返回RDD中的第一个元素

1）函数签名：def first(): T

2）功能说明：返回RDD中的第一个元素



3）需求说明：创建一个RDD，返回该RDD中的第一个元素

object action04\_first {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

//3.2 返回RDD中元素的个数

val firstResult: Int = rdd.first()

println(firstResult)

//4.关闭连接

sc.stop()

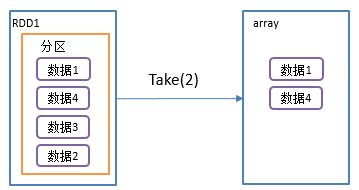
}

}

### 2.4.5 take()返回由RDD前n个元素组成的数组

1）函数签名：def take(num: Int): Array[T]

2）功能说明：返回一个由RDD的前n个元素组成的数组



3）需求说明：创建一个RDD，统计该RDD的条数

object action05\_take {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

//3.2 返回RDD中前2个元素

val takeResult: Array[Int] = rdd.take(2)

println(takeResult.mkString(","))

//4.关闭连接

sc.stop()

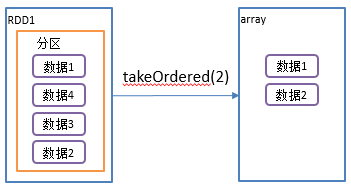
}

}

### 2.4.6 takeOrdered()返回该RDD排序后前n个元素组成的数组

1）函数签名：def takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T]

2）功能说明：返回该RDD排序后的前n个元素组成的数组



def takeOrdered(num: Int)(implicit ord: Ordering[T]): Array[T] = withScope {

......

if (mapRDDs.partitions.length == 0) {

Array.empty

} else {

mapRDDs.reduce { (queue1, queue2) =>

queue1 ++= queue2

queue1

}.toArray.sorted(ord)

}

}

3）需求说明：创建一个RDD，获取该RDD排序后的前2个元素

object action06\_takeOrdered{

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd: RDD[Int] = sc.makeRDD(List(1,3,2,4))

//3.2 返回RDD中排完序后的前两个元素

val result: Array[Int] = rdd.takeOrdered(2)

println(result.mkString(","))

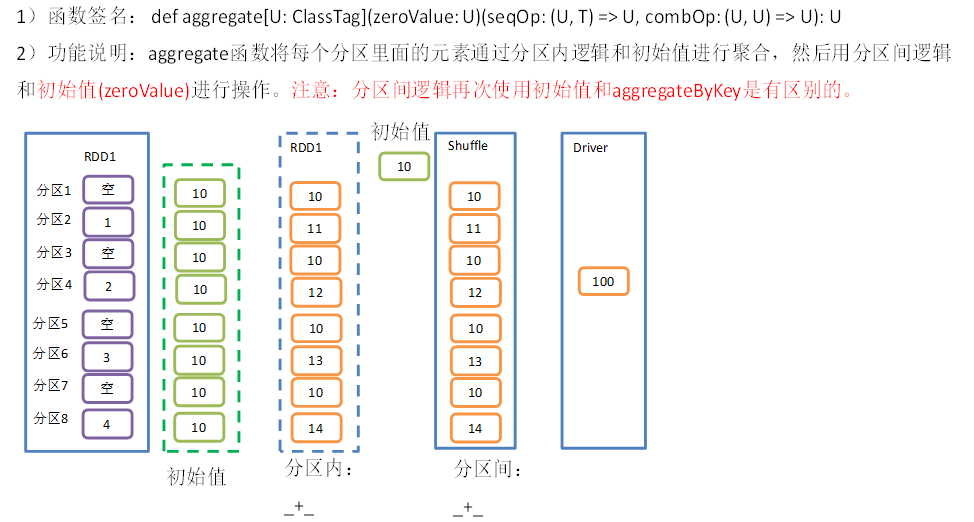
//4.关闭连接

sc.stop()

}

}

### 2.4.7 aggregate()案例



3）需求说明：创建一个RDD，将所有元素相加得到结果

object action07\_aggregate {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd: RDD[Int] = sc.makeRDD(List(1, 2, 3, 4), 8)

//3.2 将该RDD所有元素相加得到结果

//val result: Int = rdd.aggregate(0)(\_ + \_, \_ + \_)

val result: Int = rdd.aggregate(10)(\_ + \_, \_ + \_)

println(result)

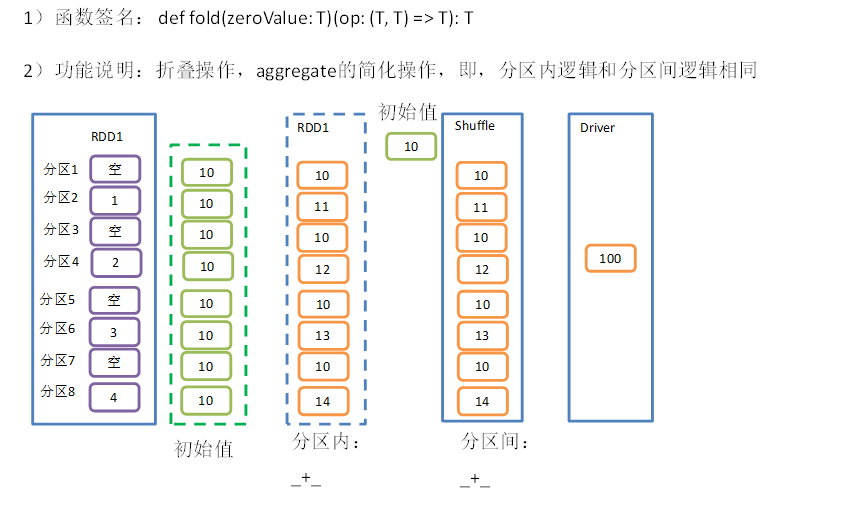
//4.关闭连接

sc.stop()

}

}

### 2.4.8 fold()案例



3）需求说明：创建一个RDD，将所有元素相加得到结果

object action08\_fold {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd: RDD[Int] = sc.makeRDD(List(1, 2, 3, 4))

//3.2 将该RDD所有元素相加得到结果

val foldResult: Int = rdd.fold(0)(\_+\_)

println(foldResult)

//4.关闭连接

sc.stop()

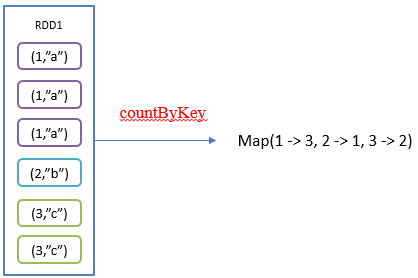
}

}

### 2.4.9 countByKey()统计每种key的个数

1）函数签名：def countByKey(): Map[K, Long]

2）功能说明：统计每种key的个数



3）需求说明：创建一个PairRDD，统计每种key的个数

object action09\_countByKey {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd: RDD[(Int, String)] = sc.makeRDD(List((1, "a"), (1, "a"), (1, "a"), (2, "b"), (3, "c"), (3, "c")))

//3.2 统计每种key的个数

val result: collection.Map[Int, Long] = rdd.countByKey()

println(result)

//4.关闭连接

sc.stop()

}

}

### 2.4.10 save相关算子

**1）saveAsTextFile(path)保存成Text文件**

（1）函数签名

（2）功能说明：将数据集的元素以textfile的形式保存到HDFS文件系统或者其他支持的文件系统，对于每个元素，Spark将会调用toString方法，将它装换为文件中的文本

**2）saveAsSequenceFile(path) 保存成Sequencefile文件**

（1）函数签名

（2）功能说明：将数据集中的元素以Hadoop Sequencefile的格式保存到指定的目录下，可以使HDFS或者其他Hadoop支持的文件系统。

注意：只有kv类型RDD有该操作，单值的没有

**3）saveAsObjectFile(path) 序列化成对象保存到文件**

（1）函数签名

（2）功能说明：用于将RDD中的元素序列化成对象，存储到文件中。

4）代码实现

object action10\_save {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4), 2)

//3.2 保存成Text文件

rdd.saveAsTextFile("output")

//3.3 序列化成对象保存到文件

rdd.saveAsObjectFile("output1")

//3.4 保存成Sequencefile文件

rdd.map((\_,1)).saveAsSequenceFile("output2")

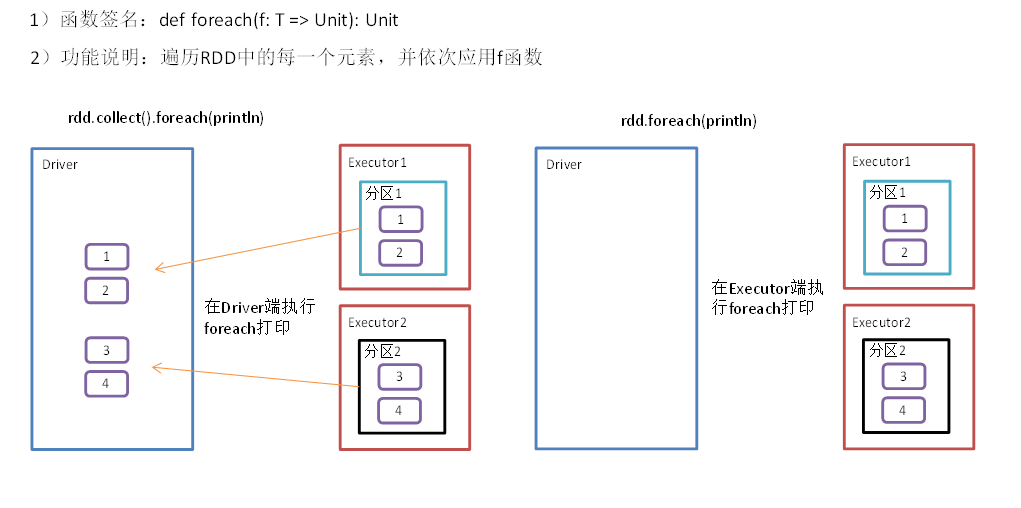
//4.关闭连接

sc.stop()

}

}

### 2.4.11 foreach(f)遍历RDD中每一个元素



3）需求说明：创建一个RDD，对每个元素进行打印

object action11\_foreach {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3具体业务逻辑

//3.1 创建第一个RDD

// val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4),2)

val rdd: RDD[Int] = sc.makeRDD(List(1,2,3,4))

//3.2 收集后打印

rdd.collect().foreach(println)

println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

//3.3 分布式打印

rdd.foreach(println)

//4.关闭连接

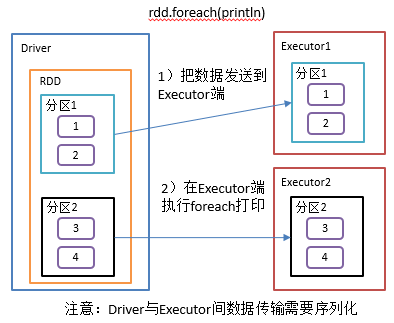
sc.stop()

}

}

## 2.5 RDD序列化

在实际开发中我们往往需要自己定义一些对于RDD的操作，那么此时需要注意的是，初始化工作是在Driver端进行的，而实际运行程序是在Executor端进行的，这就涉及到了跨进程通信，是需要序列化的。下面我们看几个例子：



### 2.5.1 闭包检查

0）创建包名：com.user1.serializable

1）闭包引入（有闭包就需要进行序列化）

object serializable01\_object {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.创建两个对象

val user1 = new User()

user1.name = "zhangsan"

val user2 = new User()

user2.name = "lisi"

val userRDD1: RDD[User] = sc.makeRDD(List(user1, user2))

//3.1 打印，ERROR报java.io.NotSerializableException

//userRDD1.foreach(user => println(user.name))

//3.2 打印，RIGHT （因为没有传对象到Executor端）

val userRDD2: RDD[User] = sc.makeRDD(List())

//userRDD2.foreach(user => println(user.name))

//3.3 打印，ERROR Task not serializable 注意：没执行就报错了

userRDD2.foreach(user => println(user1.name))

//4.关闭连接

sc.stop()

}

}

//class User {

// var name: String = \_

//}

class User extends Serializable {

var name: String = \_

}

### 2.5.2 序列化方法和属性

1）说明

Driver：算子以外的代码都是在Driver端执行

Executor：算子里面的代码都是在Executor端执行

2）代码实现

object serializable02\_function {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.创建一个RDD

val rdd: RDD[String] = sc.makeRDD(Array("hello world", "hello spark", "hive", "user1"))

//3.1创建一个Search对象

val search = new Search("hello")

// Driver：算子以外的代码都是在Driver端执行

// Executor：算子里面的代码都是在Executor端执行

//3.2 函数传递，打印：ERROR Task not serializable

search.getMatch1(rdd).collect().foreach(println)

//3.3 属性传递，打印：ERROR Task not serializable

search.getMatche2(rdd).collect().foreach(println)

//4.关闭连接

sc.stop()

}

}

class Search(query:String) extends Serializable {

def isMatch(s: String): Boolean = {

s.contains(query)

}

// 函数序列化案例

def getMatch1 (rdd: RDD[String]): RDD[String] = {

//rdd.filter(this.isMatch)

rdd.filter(isMatch)

}

// 属性序列化案例

def getMatche2(rdd: RDD[String]): RDD[String] = {

//rdd.filter(x => x.contains(this.query))

rdd.filter(x => x.contains(query))

//val q = query

//rdd.filter(x => x.contains(q))

}

}

3）问题一说明

//过滤出包含字符串的RDD

def getMatch1 (rdd: RDD[String]): RDD[String] = {

rdd.filter(isMatch)

}

（1）在这个方法中所调用的方法isMatch()是定义在Search这个类中的，实际上调用的是this. isMatch()，this表示Search这个类的对象，程序在运行过程中需要将Search对象序列化以后传递到Executor端。

（2）解决方案

类继承scala.Serializable即可。

class Search() extends Serializable{...}

4）问题二说明

//过滤出包含字符串的RDD

def getMatche2(rdd: RDD[String]): RDD[String] = {

rdd.filter(x => x.contains(query))

}

（1）在这个方法中所调用的方法query是定义在Search这个类中的字段，实际上调用的是this. query，this表示Search这个类的对象，程序在运行过程中需要将Search对象序列化以后传递到Executor端。

（2）解决方案一

（a）类继承scala.Serializable即可。

class Search() extends Serializable{...}

（b）将类变量query赋值给局部变量

修改getMatche2为

//过滤出包含字符串的RDD

def getMatche2(rdd: RDD[String]): RDD[String] = {

val q = this.query//将类变量赋值给局部变量

rdd.filter(x => x.contains(q))

}

（3）解决方案二

把Search类变成样例类，样例类默认是序列化的。

case class Search(query:String) {...}

### 2.5.3 Kryo序列化框架

参考地址: https://github.com/EsotericSoftware/kryo

Java的序列化能够序列化任何的类。但是比较重，序列化后对象的体积也比较大。

Spark出于性能的考虑，Spark2.0开始支持另外一种Kryo序列化机制。Kryo速度是Serializable的10倍。当RDD在Shuffle数据的时候，简单数据类型、数组和字符串类型已经在Spark内部使用Kryo来序列化。

注意：即使使用Kryo序列化，也要继承Serializable接口。

object serializable03\_Kryo {

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

val conf: SparkConf = new SparkConf()

.setAppName("SerDemo")

.setMaster("local[\*]")

// 替换默认的序列化机制

.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

// 注册需要使用kryo序列化的自定义类

.registerKryoClasses(Array(classOf[Searche]))

val sc = new SparkContext(conf)

val rdd: RDD[String] = sc.makeRDD(Array("hello world", "hello user1", "user1", "hahah"), 2)

val searche = new Searche("hello")

val result: RDD[String] = searche.getMatchedRDD1(rdd)

result.collect.foreach(println)

}

}

case class Searche(val query: String) {

def isMatch(s: String) = {

s.contains(query)

}

def getMatchedRDD1(rdd: RDD[String]) = {

rdd.filter(isMatch)

}

def getMatchedRDD2(rdd: RDD[String]) = {

val q = query

rdd.filter(\_.contains(q))

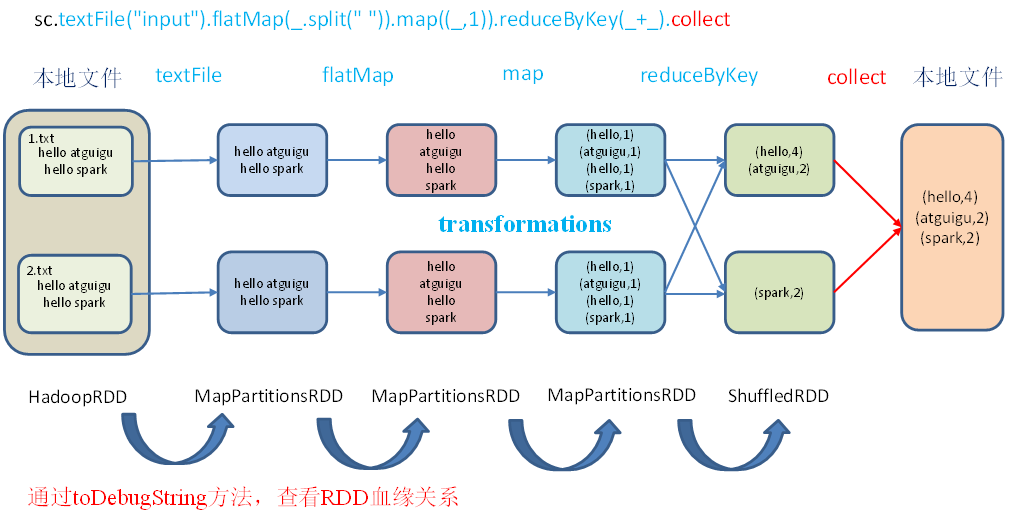
}

}

## 2.6 RDD依赖关系

### 2.6.1 查看血缘关系

RDD只支持粗粒度转换，即在大量记录上执行的单个操作。将创建RDD的一系列Lineage（血统）记录下来，以便恢复丢失的分区。RDD的Lineage会记录RDD的元数据信息和转换行为，当该RDD的部分分区数据丢失时，它可以根据这些信息来重新运算和恢复丢失的数据分区。



0）创建包名：com.user1.dependency

1）代码实现

object Lineage01 {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

val fileRDD: RDD[String] = sc.textFile("input/1.txt")

println(fileRDD.toDebugString)

println("----------------------")

val wordRDD: RDD[String] = fileRDD.flatMap(\_.split(" "))

println(wordRDD.toDebugString)

println("----------------------")

val mapRDD: RDD[(String, Int)] = wordRDD.map((\_,1))

println(mapRDD.toDebugString)

println("----------------------")

val resultRDD: RDD[(String, Int)] = mapRDD.reduceByKey(\_+\_)

println(resultRDD.toDebugString)

resultRDD.collect()

//4.关闭连接

sc.stop()

}

}

2）打印结果

(2) input/1.txt MapPartitionsRDD[1] at textFile at Lineage01.scala:15 []

| input/1.txt HadoopRDD[0] at textFile at Lineage01.scala:15 []

----------------------

(2) MapPartitionsRDD[2] at flatMap at Lineage01.scala:19 []

| input/1.txt MapPartitionsRDD[1] at textFile at Lineage01.scala:15 []

| input/1.txt HadoopRDD[0] at textFile at Lineage01.scala:15 []

----------------------

(2) MapPartitionsRDD[3] at map at Lineage01.scala:23 []

| MapPartitionsRDD[2] at flatMap at Lineage01.scala:19 []

| input/1.txt MapPartitionsRDD[1] at textFile at Lineage01.scala:15 []

| input/1.txt HadoopRDD[0] at textFile at Lineage01.scala:15 []

----------------------

(2) ShuffledRDD[4] at reduceByKey at Lineage01.scala:27 []

+-(2) MapPartitionsRDD[3] at map at Lineage01.scala:23 []

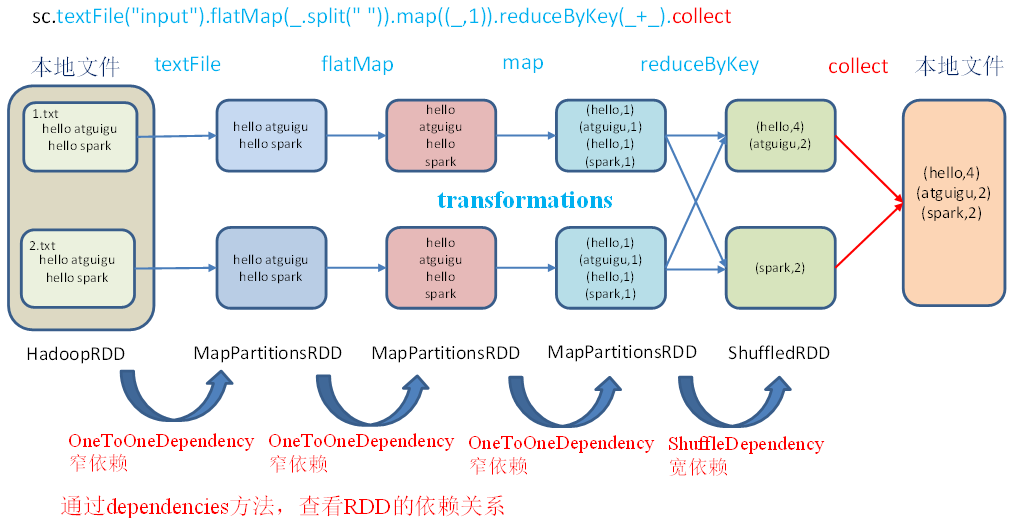
| MapPartitionsRDD[2] at flatMap at Lineage01.scala:19 []

| input/1.txt MapPartitionsRDD[1] at textFile at Lineage01.scala:15 []

| input/1.txt HadoopRDD[0] at textFile at Lineage01.scala:15 []

注意：圆括号中的数字表示RDD的并行度，也就是有几个分区

### 2.6.2 查看依赖关系



1）代码实现

object Lineage02 {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

val fileRDD: RDD[String] = sc.textFile("input/1.txt")

println(fileRDD.dependencies)

println("----------------------")

val wordRDD: RDD[String] = fileRDD.flatMap(\_.split(" "))

println(wordRDD.dependencies)

println("----------------------")

val mapRDD: RDD[(String, Int)] = wordRDD.map((\_,1))

println(mapRDD.dependencies)

println("----------------------")

val resultRDD: RDD[(String, Int)] = mapRDD.reduceByKey(\_+\_)

println(resultRDD.dependencies)

resultRDD.collect()

// 查看localhost:4040页面，观察DAG图

Thread.sleep(10000000)

//4.关闭连接

sc.stop()

}

}

2）打印结果

List(org.apache.spark.OneToOneDependency@f2ce6b)

----------------------

List(org.apache.spark.OneToOneDependency@692fd26)

----------------------

List(org.apache.spark.OneToOneDependency@627d8516)

----------------------

List(org.apache.spark.ShuffleDependency@a518813)

3）全局搜索（ctrl+n）org.apache.spark.OneToOneDependency

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

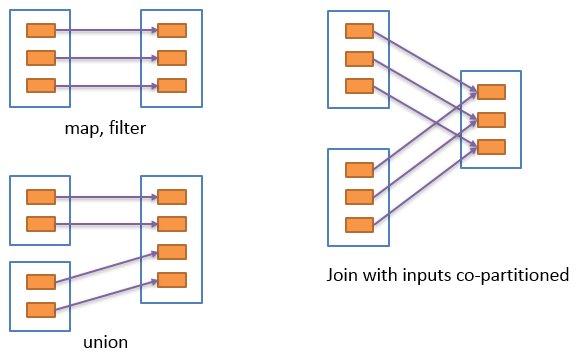
注意：要想理解RDDS是如何工作的，最重要的就是理解Transformations。

RDD之间的关系可以从两个维度来理解：一个是RDD是从哪些RDD转换而来，也就是 RDD的parent RDD(s)是什么; 另一个就是RDD依赖于parent RDD(s)的哪些Partition(s)，这种关系就是RDD之间的依赖。

RDD和它依赖的父RDD（s）的关系有两种不同的类型，即窄依赖（narrow dependency）和宽依赖（wide dependency）。

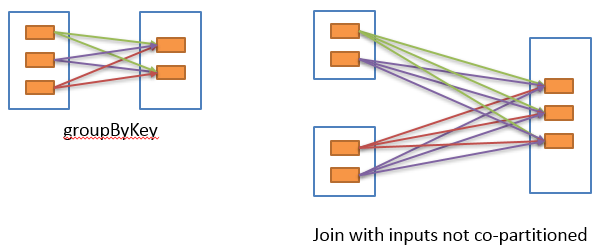
### 2.6.3 窄依赖

窄依赖表示每一个父RDD的Partition最多被子RDD的一个Partition使用，窄依赖我们形象的比喻为独生子女。



### 2.6.4 宽依赖

宽依赖表示同一个父RDD的Partition被多个子RDD的Partition依赖，会引起Shuffle，总结：宽依赖我们形象的比喻为超生。



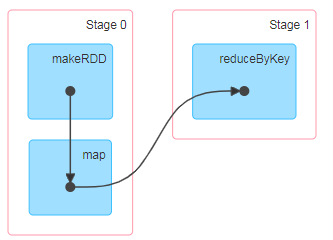
具有宽依赖的*transformations*包括：*sort*、*reduceByKey*、*groupByKey*、*join*和调用*rePartition*函数的任何操作。

宽依赖对Spark去评估一个transformations有更加重要的影响，比如对性能的影响。

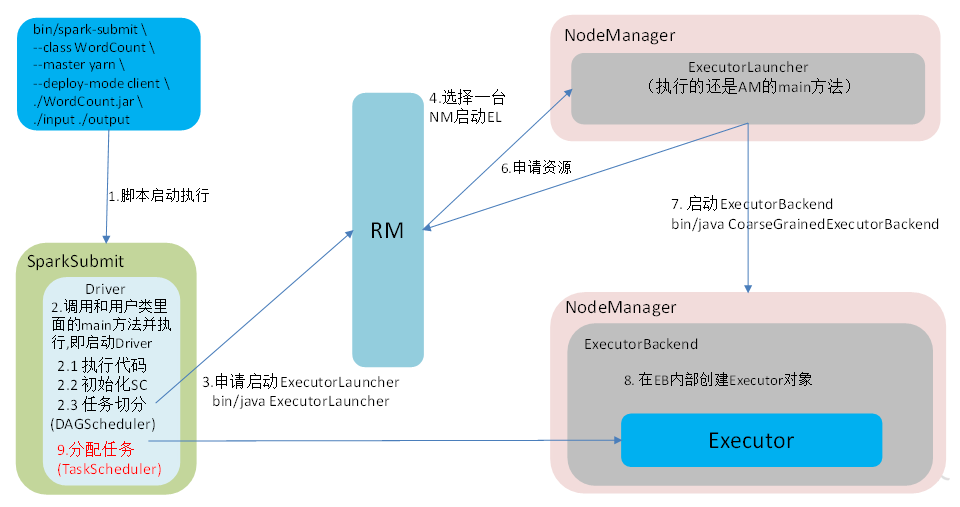
### 2.6.5 Stage任务划分

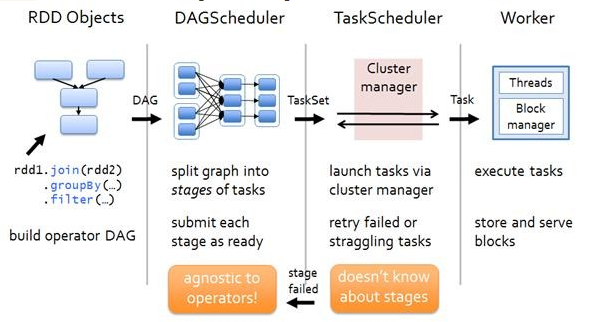
1）DAG有向无环图

DAG（Directed Acyclic Graph）有向无环图是由点和线组成的拓扑图形，该图形具有方向，不会闭环。例如，DAG记录了RDD的转换过程和任务的阶段。



2）任务运行的整体流程





3）RDD任务切分中间分为：Application、Job、Stage和Task

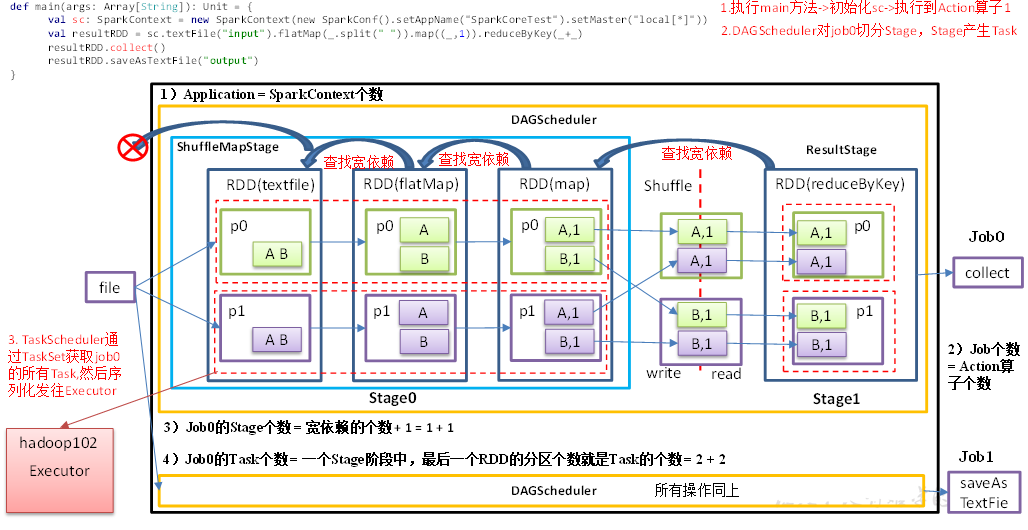
（1）Application：初始化一个SparkContext即生成一个Application；

（2）Job：一个Action算子就会生成一个Job；

（3）Stage：Stage等于宽依赖的个数加1；

（4）Task：一个Stage阶段中，最后一个RDD的分区个数就是Task的个数。

注意：Application->Job->Stage->Task每一层都是1对n的关系。



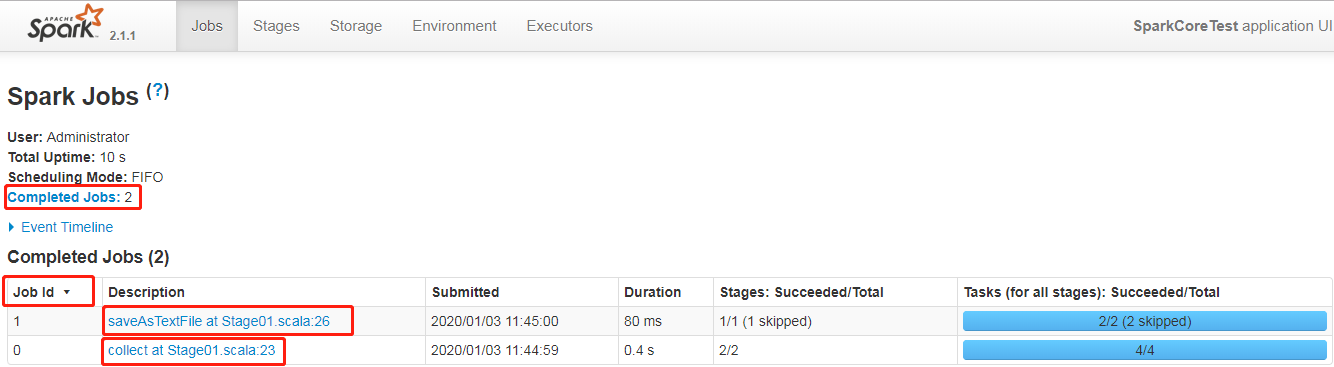
4）代码实现

object Stage01 {

def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2. Application：初始化一个SparkContext即生成一个Application；  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3. 创建RDD  
 val dataRDD: RDD[Int] = sc.makeRDD(List(1,2,3,4,1,2),2)  
  
 //3.1 聚合  
 val resultRDD: RDD[(Int, Int)] = dataRDD.map((\_,1)).reduceByKey(\_+\_)  
  
 // Job：一个Action算子就会生成一个Job；  
 //3.2 job1打印到控制台  
 resultRDD.collect().foreach(println)  
  
 //3.3 job2输出到磁盘  
 resultRDD.saveAsTextFile("output")  
  
 Thread.sleep(1000000)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

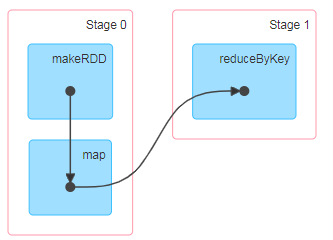
5）查看Job个数

查看<http://localhost:4040/jobs/>，发现Job有两个。

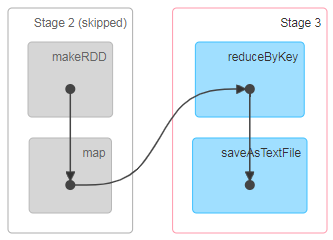


6）查看Stage个数

查看Job0的Stage。由于只有1个Shuffle阶段，所以Stage个数为2。

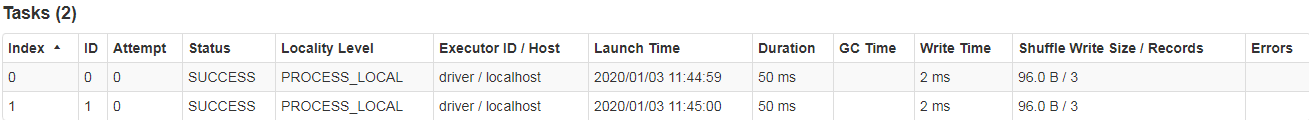


查看Job1的Stage。由于只有1个Shuffle阶段，所以Stage个数为2。

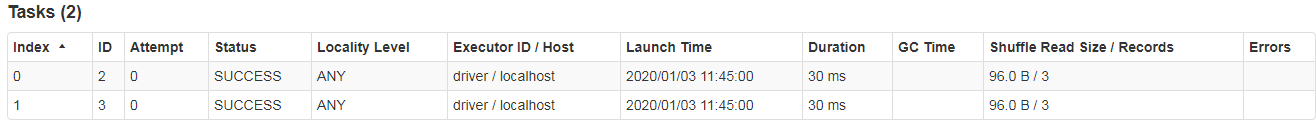


7）Task个数

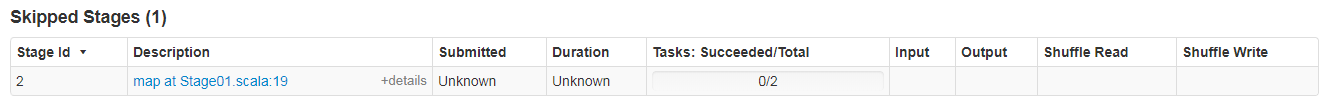
查看Job0的Stage0的Task个数



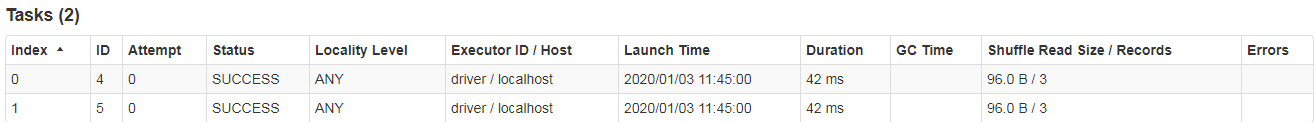
查看Job0的Stage1的Task个数



查看Job1的Stage2的Task个数



查看Job1的Stage3的Task个数

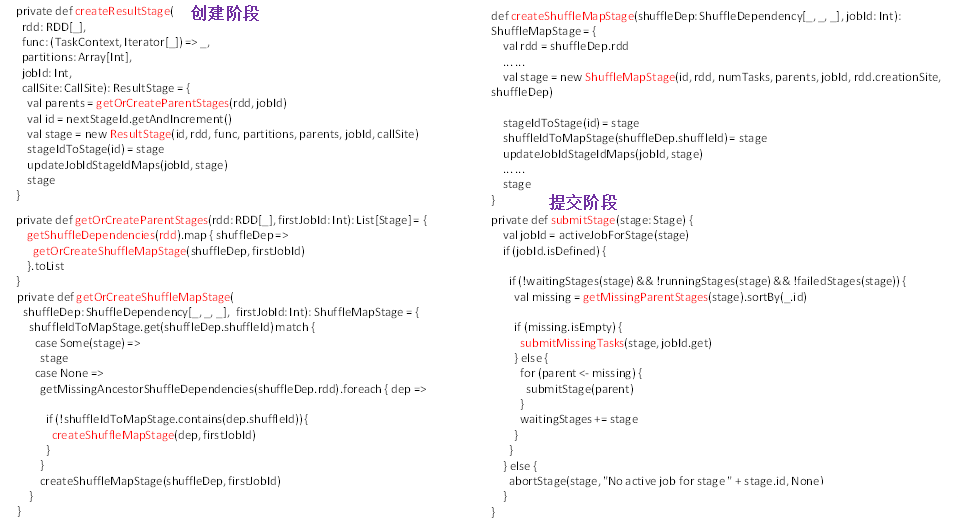


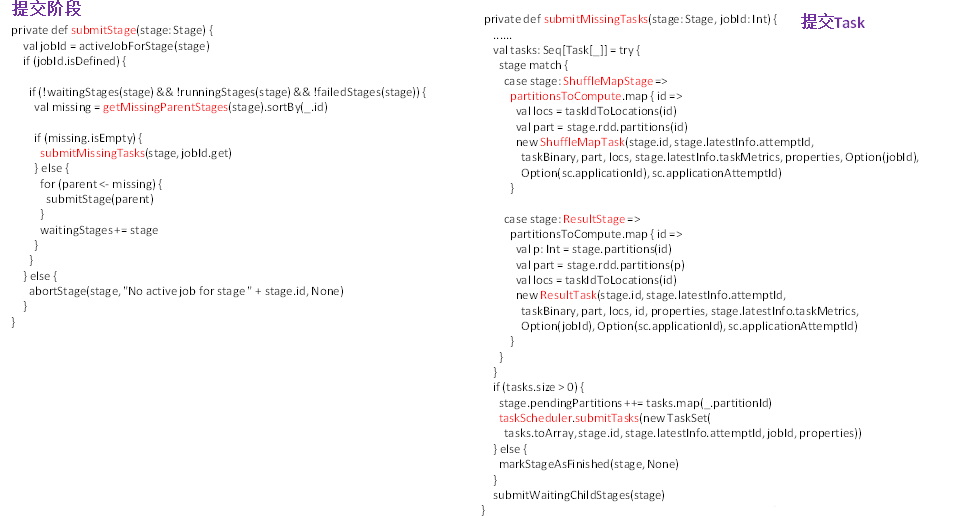
注意：如果存在shuffle过程，系统会自动进行缓存，UI界面显示skipped的部分

### 2.6.6 Stage任务划分源码分析





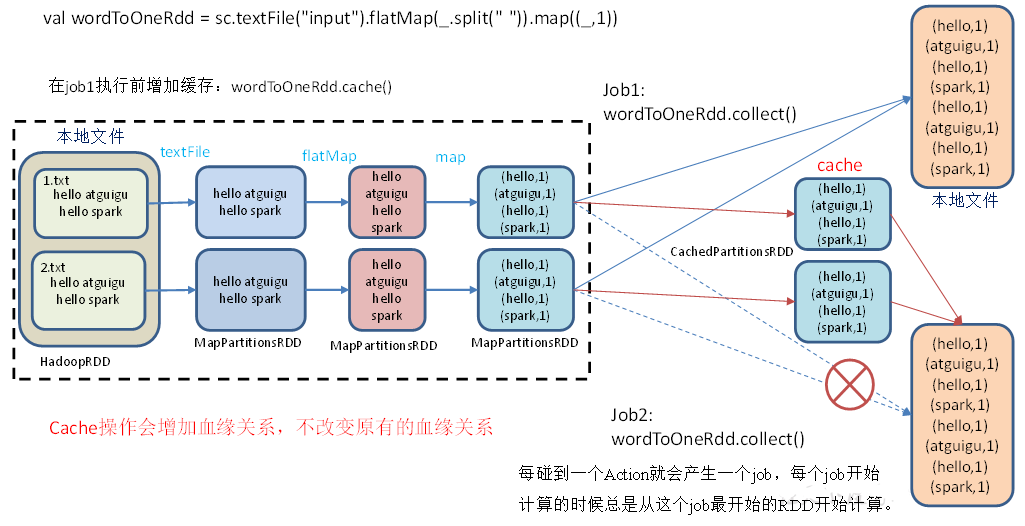




## 2.7 RDD持久化

### 2.7.1 RDD Cache缓存

RDD通过Cache或者Persist方法将前面的计算结果缓存，默认情况下会把数据以序列化的形式缓存在JVM的堆内存中。但是并不是这两个方法被调用时立即缓存，而是触发后面的action时，该RDD将会被缓存在计算节点的内存中，并供后面重用。



0）创建包名：com.user1.cache

1）代码实现

object cache01 {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3. 创建一个RDD，读取指定位置文件:hello user1 user1

val lineRdd: RDD[String] = sc.textFile("input1")

//3.1.业务逻辑

val wordRdd: RDD[String] = lineRdd.flatMap(line => line.split(" "))

val wordToOneRdd: RDD[(String, Int)] = wordRdd.map {

word => {

println("\*\*\*\*\*\*\*\*\*\*\*\*")

(word, 1)

}

}

//3.5 cache操作会增加血缘关系，不改变原有的血缘关系

println(wordToOneRdd.toDebugString)

//3.4 数据缓存。

wordToOneRdd.cache()

//3.6 可以更改存储级别

// wordToOneRdd.persist(StorageLevel.MEMORY\_AND\_DISK\_2)

//3.2 触发执行逻辑

wordToOneRdd.collect()

println("-----------------")

println(wordToOneRdd.toDebugString)

//3.3 再次触发执行逻辑

wordToOneRdd.collect()

Thread.sleep(1000000)

//4.关闭连接

sc.stop()

}

}

2）源码解析

mapRdd.cache()

def cache(): this.type = persist()

def persist(): this.type = persist(StorageLevel.MEMORY\_ONLY)

object StorageLevel {

val NONE = new StorageLevel(false, false, false, false)

val DISK\_ONLY = new StorageLevel(true, false, false, false)

val DISK\_ONLY\_2 = new StorageLevel(true, false, false, false, 2)

val MEMORY\_ONLY = new StorageLevel(false, true, false, true)

val MEMORY\_ONLY\_2 = new StorageLevel(false, true, false, true, 2)

val MEMORY\_ONLY\_SER = new StorageLevel(false, true, false, false)

val MEMORY\_ONLY\_SER\_2 = new StorageLevel(false, true, false, false, 2)

val MEMORY\_AND\_DISK = new StorageLevel(true, true, false, true)

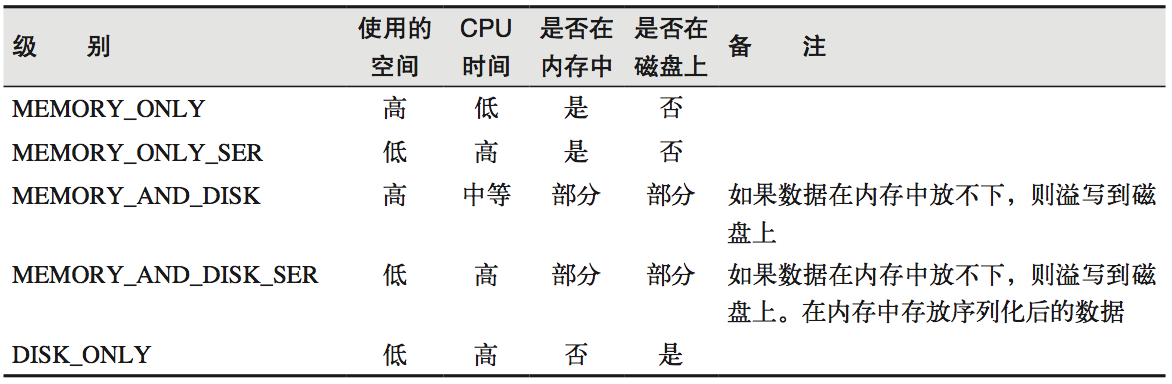
val MEMORY\_AND\_DISK\_2 = new StorageLevel(true, true, false, true, 2)

val MEMORY\_AND\_DISK\_SER = new StorageLevel(true, true, false, false)

val MEMORY\_AND\_DISK\_SER\_2 = new StorageLevel(true, true, false, false, 2)

val OFF\_HEAP = new StorageLevel(true, true, true, false, 1)

注意：默认的存储级别都是仅在内存存储一份。在存储级别的末尾加上“\_2”表示持久化的数据存为两份。SER：表示序列化。



缓存有可能丢失，或者存储于内存的数据由于内存不足而被删除，RDD的缓存容错机制保证了即使缓存丢失也能保证计算的正确执行。通过基于RDD的一系列转换，丢失的数据会被重算，由于RDD的各个Partition是相对独立的，因此只需要计算丢失的部分即可，并不需要重算全部Partition。

3）自带缓存算子

Spark会自动对一些Shuffle操作的中间数据做持久化操作（比如：reduceByKey）。这样做的目的是为了当一个节点Shuffle失败了避免重新计算整个输入。但是，在实际使用的时候，如果想重用数据，仍然建议调用persist或cache。

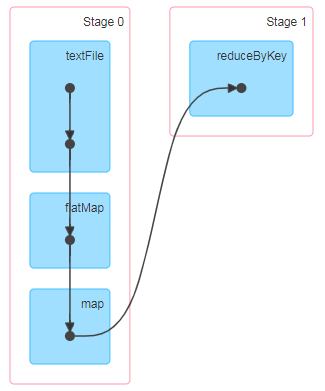
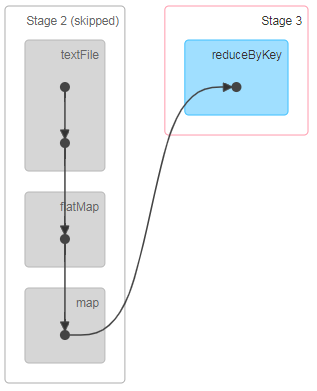
object cache02 {  
  
 def main(args: Array[String]): Unit = {

//1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3. 创建一个RDD，读取指定位置文件:hello user1 user1  
 val lineRdd: RDD[String] = sc.textFile("input1")  
  
 //3.1.业务逻辑  
 val wordRdd: RDD[String] = lineRdd.flatMap(line => line.split(" "))  
  
 val wordToOneRdd: RDD[(String, Int)] = wordRdd.map {  
 word => {  
 println("\*\*\*\*\*\*\*\*\*\*\*\*")  
 (word, 1)  
 }  
 }

// 采用reduceByKey，自带缓存  
 val wordByKeyRDD: RDD[(String, Int)] = wordToOneRdd.reduceByKey(\_+\_)  
  
 //3.5 cache操作会增加血缘关系，不改变原有的血缘关系  
 println(wordByKeyRDD.toDebugString)  
  
 //3.4 数据缓存。  
 //wordByKeyRDD.cache()  
  
 //3.2 触发执行逻辑  
 wordByKeyRDD.collect()  
  
 println("-----------------")  
 println(wordByKeyRDD.toDebugString)  
  
 //3.3 再次触发执行逻辑  
 wordByKeyRDD.collect()

Thread.sleep(1000000)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

访问<http://localhost:4040/jobs/>页面，查看第一个和第二个job的DAG图。说明：增加缓存后血缘依赖关系仍然有，但是，第二个job取的数据是从缓存中取的。

### 2.7.2 RDD CheckPoint检查点

1）检查点：是通过将RDD中间结果写入磁盘。

2）为什么要做检查点？

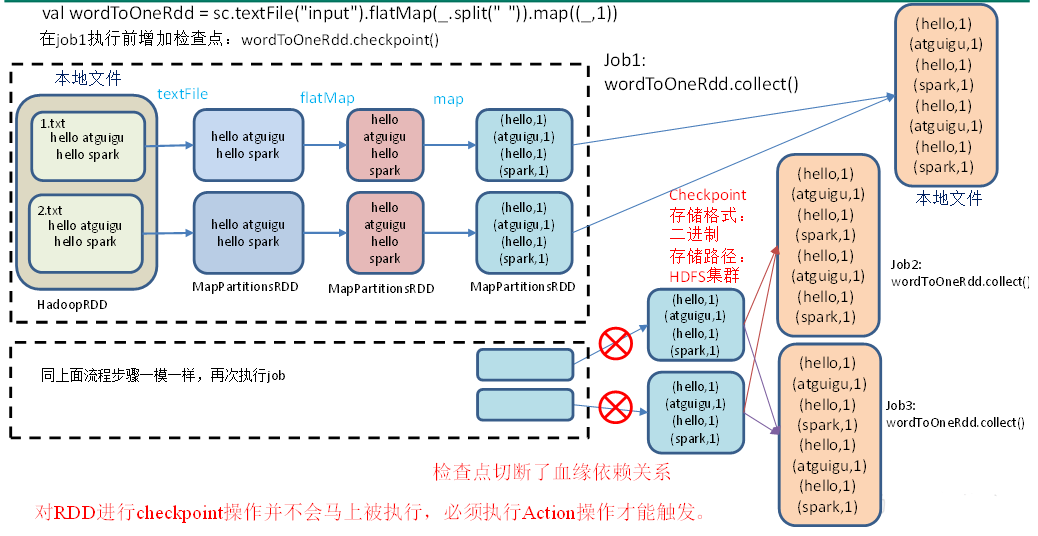
由于血缘依赖过长会造成容错成本过高，这样就不如在中间阶段做检查点容错，如果检查点之后有节点出现问题，可以从检查点开始重做血缘，减少了开销。

3）检查点存储路径：Checkpoint的数据通常是存储在HDFS等容错、高可用的文件系统

4）检查点数据存储格式为：二进制的文件

5）检查点切断血缘：在Checkpoint的过程中，该RDD的所有依赖于父RDD中的信息将全部被移除。

6）检查点触发时间：对RDD进行Checkpoint操作并不会马上被执行，必须执行Action操作才能触发。但是检查点为了数据安全，会从血缘关系的最开始执行一遍。



7）设置检查点步骤

（1）设置检查点数据存储路径：sc.setCheckpointDir("./checkpoint1")

（2）调用检查点方法：wordToOneRdd.checkpoint()

8）代码实现

object checkpoint01 {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

// 需要设置路径，否则抛异常：Checkpoint directory has not been set in the SparkContext

sc.setCheckpointDir("./checkpoint1")

//3. 创建一个RDD，读取指定位置文件:hello user1 user1

val lineRdd: RDD[String] = sc.textFile("input1")

//3.1.业务逻辑

val wordRdd: RDD[String] = lineRdd.flatMap(line => line.split(" "))

val wordToOneRdd: RDD[(String, Long)] = wordRdd.map {

word => {

(word, System.currentTimeMillis())

}

}

//3.5 增加缓存，避免再重新跑一个job做checkpoint

// wordToOneRdd.cache()

//3.4 数据检查点：针对wordToOneRdd做检查点计算

wordToOneRdd.checkpoint()

//3.2 触发执行逻辑

wordToOneRdd.collect().foreach(println)

// 会立即启动一个新的job来专门的做checkpoint运算

//3.3 再次触发执行逻辑

wordToOneRdd.collect().foreach(println)

wordToOneRdd.collect().foreach(println)

Thread.sleep(10000000)

//4.关闭连接

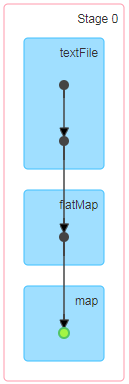
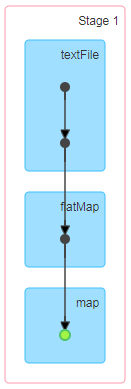
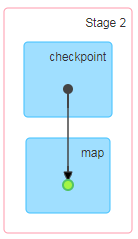
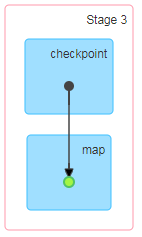
sc.stop()

}

}

9）执行结果

访问<http://localhost:4040/jobs/页面，查看4>个job的DAG图。其中第2个图是checkpoint的job运行DAG图。第3、4张图说明，检查点切断了血缘依赖关系。

（1）只增加checkpoint，没有增加Cache缓存打印

第1个job执行完，触发了checkpoint，第2个job运行checkpoint，并把数据存储在检查点上。第3、4个job，数据从检查点上直接读取。

(hadoop,1577960215526)

。。。。。。

(hello,1577960215526)

(hadoop,1577960215609)

。。。。。。

(hello,1577960215609)

(hadoop,1577960215609)

。。。。。。

(hello,1577960215609)

（2）增加checkpoint，也增加Cache缓存打印

第1个job执行完，数据就保存到Cache里面了，第2个job运行checkpoint，直接读取Cache里面的数据，并把数据存储在检查点上。第3、4个job，数据从检查点上直接读取。

(hadoop,1577960642223)

。。。。。。

(hello,1577960642225)

(hadoop,1577960642223)

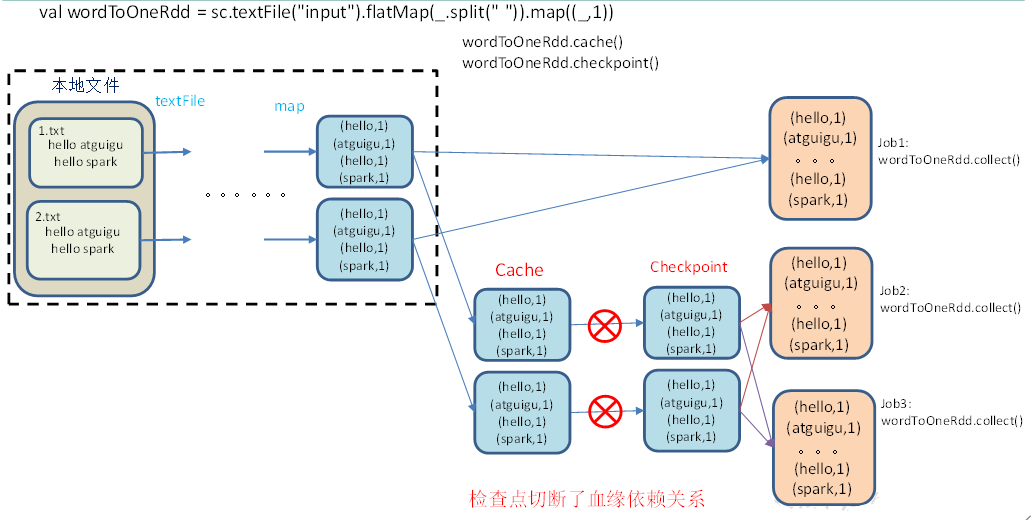
。。。。。。

(hello,1577960642225)

(hadoop,1577960642223)

。。。。。。

(hello,1577960642225)



### 2.7.3 缓存和检查点区别

1）Cache缓存只是将数据保存起来，不切断血缘依赖。Checkpoint检查点切断血缘依赖。

2）Cache缓存的数据通常存储在磁盘、内存等地方，可靠性低。Checkpoint的数据通常存储在HDFS等容错、高可用的文件系统，可靠性高。

3）建议对checkpoint()的RDD使用Cache缓存，这样checkpoint的job只需从Cache缓存中读取数据即可，否则需要再从头计算一次RDD。

4）如果使用完了缓存，可以通过unpersist()方法释放缓存

### 2.7.4 检查点存储到HDFS集群

如果检查点数据存储到HDFS集群，要注意配置访问集群的用户名。否则会报访问权限异常。

object checkpoint02 {  
  
 def main(args: Array[String]): Unit = {  
  
 // 设置访问HDFS集群的用户名  
 System.setProperty("HADOOP\_USER\_NAME","user1")  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 // 需要设置路径.需要提前在HDFS集群上创建/checkpoint路径  
 sc.setCheckpointDir("hdfs://hadoop102:8020/checkpoint")  
  
 //3. 创建一个RDD，读取指定位置文件:hello user1 user1  
 val lineRdd: RDD[String] = sc.textFile("input1")  
  
 //3.1.业务逻辑  
 val wordRdd: RDD[String] = lineRdd.flatMap(line => line.split(" "))  
  
 val wordToOneRdd: RDD[(String, Long)] = wordRdd.map {  
 word => {  
 (word, System.currentTimeMillis())  
 }  
 }  
  
 //3.4 增加缓存，避免再重新跑一个job做checkpoint  
 wordToOneRdd.cache()  
  
 //3.3 数据检查点：针对wordToOneRdd做检查点计算  
 wordToOneRdd.checkpoint()  
  
 //3.2 触发执行逻辑  
 wordToOneRdd.collect().foreach(println)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

## 2.8 键值对RDD数据分区

Spark目前支持Hash分区、Range分区和用户自定义分区。Hash分区为当前的默认分区。分区器直接决定了RDD中分区的个数、RDD中每条数据经过Shuffle后进入哪个分区和Reduce的个数。

1）注意：

（1）只有Key-Value类型的RDD才有分区器，非Key-Value类型的RDD分区的值是None

（2）每个RDD的分区ID范围：0~numPartitions-1，决定这个值是属于那个分区的。

2）获取RDD分区

（1）创建包名：com.user1.partitioner

（2）代码实现

object partitioner01\_get {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3 创建RDD

val pairRDD: RDD[(Int, Int)] = sc.makeRDD(List((1,1),(2,2),(3,3)))

//3.1 打印分区器

println(pairRDD.partitioner)

//3.2 使用HashPartitioner对RDD进行重新分区

val partitionRDD: RDD[(Int, Int)] = pairRDD.partitionBy(new HashPartitioner(2))

//3.3 打印分区器

println(partitionRDD.partitioner)

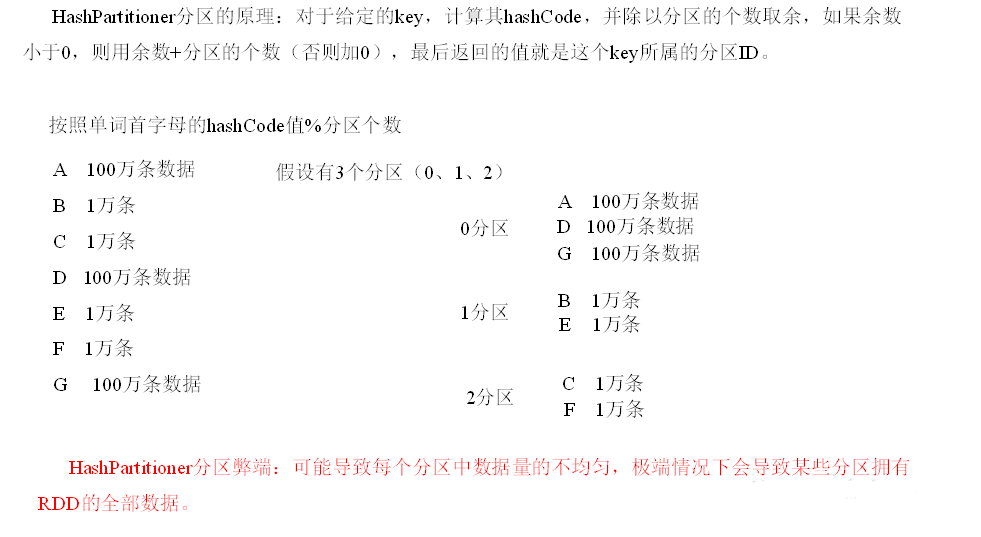
//4.关闭连接

sc.stop()

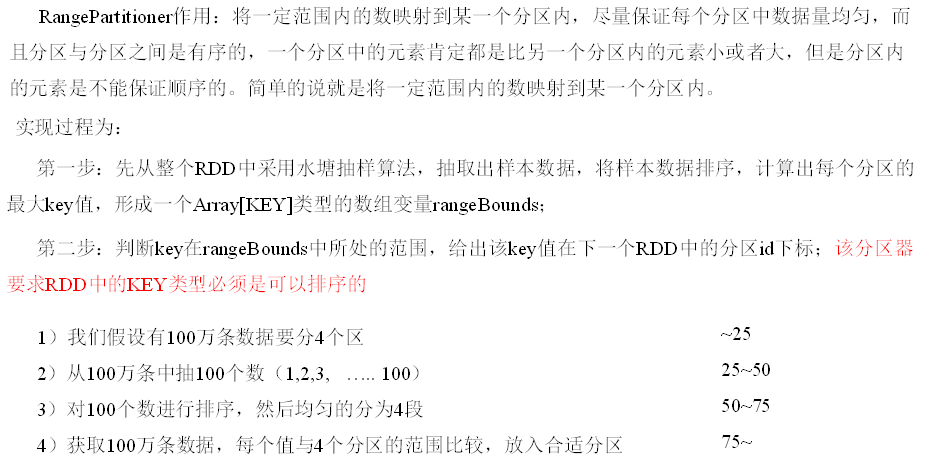
}

}

### 2.8.1 Hash分区



### 2.8.2 Ranger分区



# 第3章 数据读取与保存

Spark的数据读取及数据保存可以从两个维度来作区分：文件格式以及文件系统。

文件格式分为：Text文件、Json文件、Csv文件、Sequence文件以及Object文件；

文件系统分为：本地文件系统、HDFS以及数据库。

## 3.1 文件类数据读取与保存

1）创建包名：com.user1.readAndSave

### 3.1.1 Text文件

1）基本语法

（1）数据读取：textFile(String)

（2）数据保存：saveAsTextFile(String)

2）代码实现

object Operate\_Text {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[1]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.1 读取输入文件

val inputRDD: RDD[String] = sc.textFile("input/1.txt")

//3.2 保存数据

inputRDD.saveAsTextFile("output")

//4.关闭连接

sc.stop()

}

}

4）注意：如果是集群路径：hdfs://hadoop102:8020/input/1.txt

### 3.1.2 Sequence文件

SequenceFile文件是Hadoop用来存储二进制形式的key-value对而设计的一种平面文件(Flat File)。在SparkContext中，可以调用sequenceFile[keyClass, valueClass](path)。

1）代码实现

object Operate\_Sequence {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[1]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.1 创建rdd

val dataRDD: RDD[(Int, Int)] = sc.makeRDD(Array((1,2),(3,4),(5,6)))

//3.2 保存数据为SequenceFile

dataRDD.saveAsSequenceFile("output")

//3.3 读取SequenceFile文件

sc.sequenceFile[Int,Int]("output").collect().foreach(println)

//4.关闭连接

sc.stop()

}

}

2）注意：SequenceFile文件只针对PairRDD

### 3.1.3 Object对象文件

对象文件是将对象序列化后保存的文件，采用Java的序列化机制。可以通过objectFile[k,v](path)函数接收一个路径，读取对象文件，返回对应的RDD，也可以通过调用saveAsObjectFile()实现对对象文件的输出。因为是序列化所以要指定类型。

1）代码实现

object Operate\_Object {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[1]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.1 创建RDD

val dataRDD: RDD[Int] = sc.makeRDD(Array(1,2,3,4))

//3.2 保存数据

dataRDD.saveAsObjectFile("output")

//3.3 读取数据

sc.objectFile[(Int)]("output").collect().foreach(println)

//4.关闭连接

sc.stop()

}

}

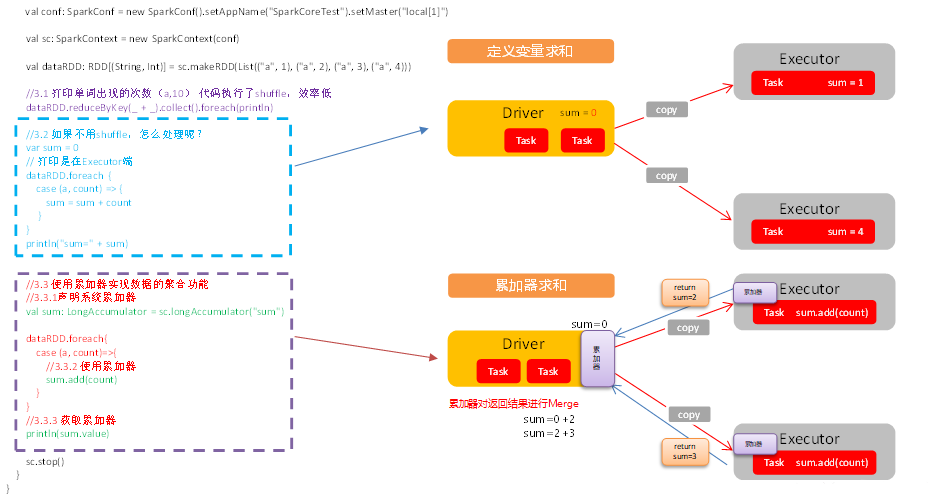
## 3.2 文件系统类数据读取与保存

Spark的整个生态系统与Hadoop是完全兼容的，所以对于Hadoop所支持的文件类型或者数据库类型，Spark也同样支持。另外，由于Hadoop的API有新旧两个版本，所以Spark为了能够兼容Hadoop所有的版本，也提供了两套创建操作接口。如TextInputFormat，新旧两个版本所引用分别是org.apache.hadoop.mapred.InputFormat、org.apache.hadoop.mapreduce.InputFormat(NewInputFormat)

# 第4章 累加器

累加器：分布式共享只写变量。（Executor和Executor之间不能读数据）

累加器用来把Executor端变量信息聚合到Driver端。在Driver程序中定义的变量，在Executor端的每个task都会得到这个变量的一份新的副本，每个task更新这些副本的值后，传回Driver端进行merge。



## 4.1 系统累加器

1）累加器使用

（1）累加器定义（SparkContext.accumulator(initialValue)方法）

val sum: LongAccumulator = sc.longAccumulator("sum")

（2）累加器添加数据（累加器.add方法）

sum.add(count)

（3）累加器获取数据（累加器.value）

sum.value

2）创建包名：com.user1.accumulator

3）代码实现

object accumulator01\_system {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[1]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.创建RDD

val dataRDD: RDD[(String, Int)] = sc.makeRDD(List(("a", 1), ("a", 2), ("a", 3), ("a", 4)))

//3.1 打印单词出现的次数（a,10） 代码执行了shuffle，效率比较低

dataRDD.reduceByKey(\_ + \_).collect().foreach(println)

//3.2 如果不用shuffle，怎么处理呢？

var sum = 0

// 打印是在Executor端

dataRDD.foreach {

case (a, count) => {

sum = sum + count

println("sum=" + sum)

}

}

// 打印是在Driver端

println(("a", sum))

//3.3 使用累加器实现数据的聚合功能

// Spark自带常用的累加器

//3.3.1 声明累加器

val sum1: LongAccumulator = sc.longAccumulator("sum1")

dataRDD.foreach{

case (a, count)=>{

//3.3.2 使用累加器

sum1.add(count)

//3.3.3 不在Executor端读取累加器的值

//println(sum1.value)

}

}

//3.3.4 获取累加器

println(sum1.value)

//4.关闭连接

sc.stop()

}

}

注意：Executor端的任务不能读取累加器的值（例如：在Executor端调用sum.value，获取的值不是累加器最终的值）。从这些任务的角度来看，累加器是一个只写变量。

3）累加器放在行动算子中

对于要在行动操作中使用的累加器，Spark只会把每个任务对各累加器的修改应用一次。因此，如果想要一个无论在失败还是重复计算时都绝对可靠的累加器，我们必须把它放在foreach()这样的行动操作中。转化操作中累加器可能会发生不止一次更新。

object accumulator02\_updateCount {  
  
 def main(args: Array[String]): Unit = {  
  
 //1.创建SparkConf并设置App名称  
 val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")  
  
 //2.创建SparkContext，该对象是提交Spark App的入口  
 val sc: SparkContext = new SparkContext(conf)  
  
 //3.创建RDD  
 val dataRDD: RDD[(String, Int)] = sc.makeRDD(List(("a", 1), ("a", 2), ("a", 3), ("a", 4)))

//3.1 定义累加器  
 val sum: LongAccumulator = sc.longAccumulator("sum")

val value: RDD[(String, Int)] = dataRDD.map(t => {  
 //3.2 累加器添加数据  
 sum.add(1)  
 t  
 })  
  
 //3.3 调用两次行动算子，map执行两次，导致最终累加器的值翻倍  
 value.foreach(println)  
 value.collect()  
  
 //3.4 获取累加器的值  
 println("a:"+sum.value)  
  
 //4.关闭连接  
 sc.stop()  
 }  
}

## 4.2 自定义累加器

自定义累加器类型的功能在1.X版本中就已经提供了，但是使用起来比较麻烦，在2.0版本后，累加器的易用性有了较大的改进，而且官方还提供了一个新的抽象类：AccumulatorV2来提供更加友好的自定义类型累加器的实现方式。

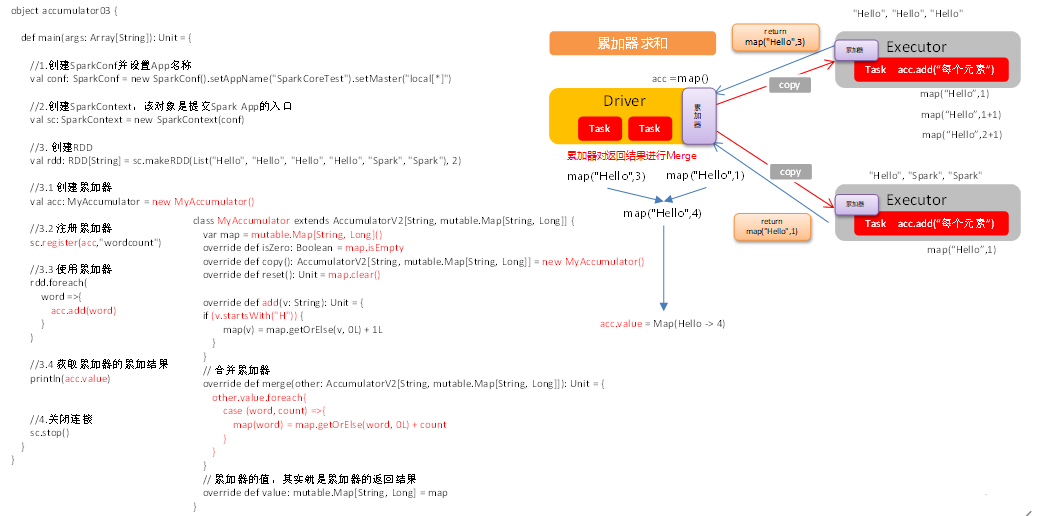
1）自定义累加器步骤

（1）继承AccumulatorV2，设定输入、输出泛型

（2）重写方法

2）需求：自定义累加器，统计RDD中首字母为“H”的单词以及出现的次数。

List("Hello", "Hello", "Hello", "Hello", "Hello", "Spark", "Spark")



3）代码实现

package com.user1.accumulator

import org.apache.spark.rdd.RDD

import org.apache.spark.util.AccumulatorV2

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

import scala.collection.mutable

object accumulator03\_define {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3. 创建RDD

val rdd: RDD[String] = sc.makeRDD(List("Hello", "Hello", "Hello", "Hello", "Spark", "Spark"), 2)

//3.1 创建累加器

val acc: MyAccumulator = new MyAccumulator()

//3.2 注册累加器

sc.register(acc,"wordcount")

//3.3 使用累加器

rdd.foreach(

word =>{

acc.add(word)

}

)

//3.4 获取累加器的累加结果

println(acc.value)

//4.关闭连接

sc.stop()

}

}

// 声明累加器

// 1.继承AccumulatorV2,设定输入、输出泛型

// 2.重新方法

class MyAccumulator extends AccumulatorV2[String, mutable.Map[String, Long]] {

// 定义输出数据集合

var map = mutable.Map[String, Long]()

// 是否为初始化状态，如果集合数据为空，即为初始化状态

override def isZero: Boolean = map.isEmpty

// 复制累加器

override def copy(): AccumulatorV2[String, mutable.Map[String, Long]] = {

new MyAccumulator()

}

// 重置累加器

override def reset(): Unit = map.clear()

// 增加数据

override def add(v: String): Unit = {

// 业务逻辑

if (v.startsWith("H")) {

map(v) = map.getOrElse(v, 0L) + 1L

}

}

// 合并累加器

override def merge(other: AccumulatorV2[String, mutable.Map[String, Long]]): Unit = {

other.value.foreach{

case (word, count) =>{

map(word) = map.getOrElse(word, 0L) + count

}

}

}

// 累加器的值，其实就是累加器的返回结果

override def value: mutable.Map[String, Long] = map

}

# 第5章 广播变量

广播变量：分布式共享只读变量。

广播变量用来高效分发较大的对象。向所有工作节点发送一个较大的只读值，以供一个或多个Spark操作使用。比如，如果你的应用需要向所有节点发送一个较大的只读查询表，广播变量用起来都很顺手。在多个并行操作中使用同一个变量，但是 Spark会为每个任务分别发送。

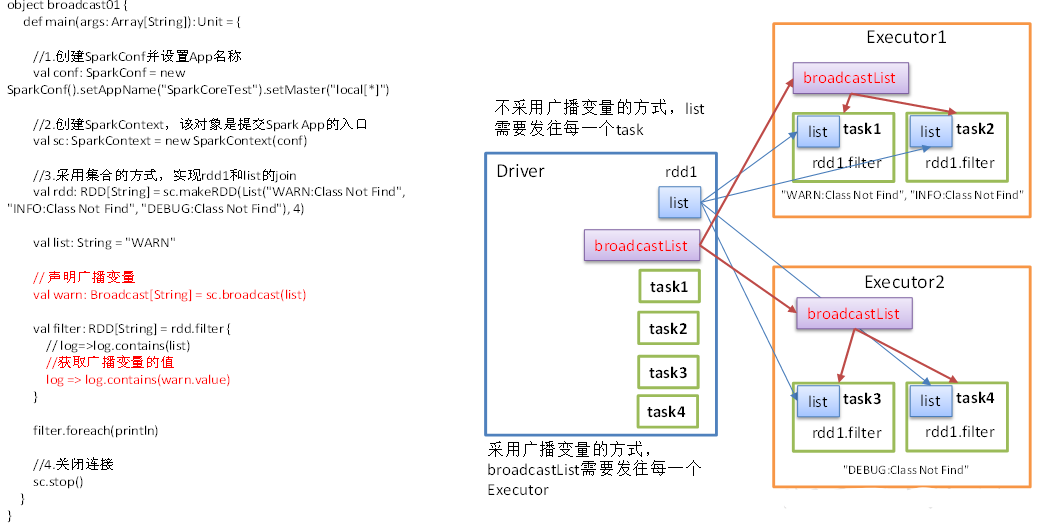
1）使用广播变量步骤：

（1）调用SparkContext.broadcast（广播变量）创建出一个广播对象，任何可序列化的类型都可以这么实现。

（2）通过广播变量.value，访问该对象的值。

（3）变量只会被发到各个节点一次，作为只读值处理（修改这个值不会影响到别的节点）。

2）原理说明

3）创建包名：com.user1.broadcast

4）代码实现

object broadcast01 {

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

//1.创建SparkConf并设置App名称

val conf: SparkConf = new SparkConf().setAppName("SparkCoreTest").setMaster("local[\*]")

//2.创建SparkContext，该对象是提交Spark App的入口

val sc: SparkContext = new SparkContext(conf)

//3.采用集合的方式，实现rdd1和list的join

val rdd: RDD[String] = sc.makeRDD(List("WARN:Class Not Find", "INFO:Class Not Find", "DEBUG:Class Not Find"), 4)

val list: String = "WARN"

// 声明广播变量

val warn: Broadcast[String] = sc.broadcast(list)

val filter: RDD[String] = rdd.filter {

// log=>log.contains(list)

log => log.contains(warn.value)

}

filter.foreach(println)

//4.关闭连接

sc.stop()

}

}