# 一、Spark RDD 概述

**Resilient Distributed Dataset**(http://spark.apache.org/docs/latest/rdd-programming-guide.html)

回顾Spark 程序，一般都包含一个Driver Program用于运行main函数，在该函数中执行着各种各样的并行操作。其中在Spark中有重要的概念RDD。该RDD是一个带有分区的分布式数据集，将数据分布存储在Spark集群的各个节点。当对RDD做任何操作，该操作都是并行的。

## RDD特点

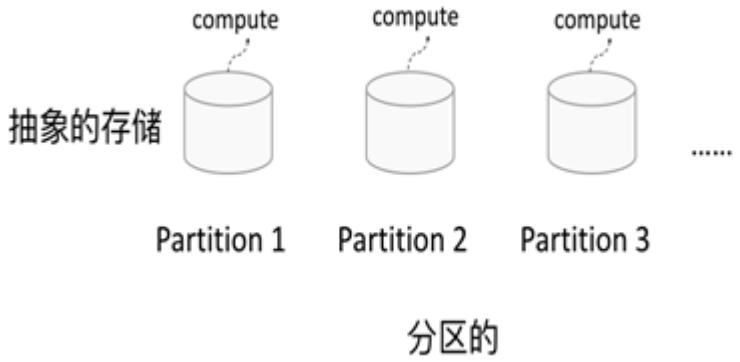
> RDD Represents an immutable, partitioned collection of elements that can be operated on in parallel.
> RDD 代表者一个不可变、带有分区的集合，可以被并行操作。

五大特性

```
- A list of partitions (带有分区)
- A function for computing each split (每个分区都是独立运行function操作的，继而实现并行)
- A list of dependencies on other RDDs (因为RDD是不可变的，因此RDD存在一种转换依赖关系，将这种转换依赖关系成为RDD的血统-lineage，可以实现RDD的故障恢复)
- Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)(可以对Key-Value类型的数据，指定Partitioner策略，默认系统使用Hash-Partitioned)
- Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file) (Spark在计算HDFS的时候，可以考虑最优计算策略。)
```

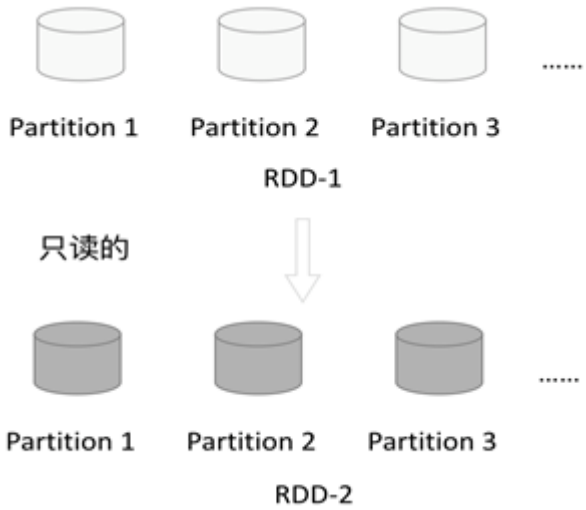
## 1.1 分区

RDD逻辑上是分区的，每个分区的数据是抽象存在的，计算的时候会通过一个compute函数得到每个分区的数据。如果RDD是通过已有的文件系统构建，则compute函数是读取指定文件系统中的数据，如果RDD是通过其他RDD转换而来，则compute函数是执行转换逻辑将其他RDD的数据进行转换。

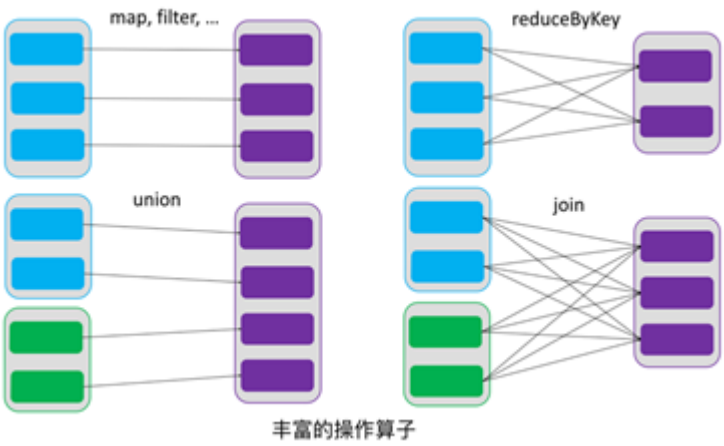



## 1.2 只读

如下图所示，RDD是只读的，要想改变RDD中的数据，只能在现有的RDD基础上创建新的RDD。



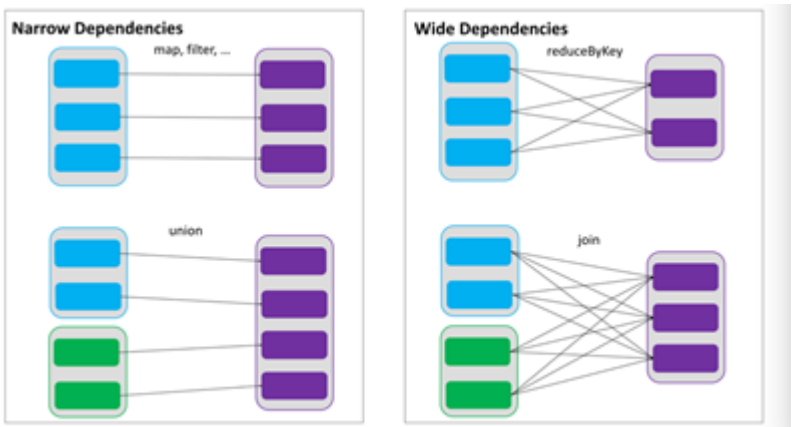

由一个RDD转换到另一个RDD，可以通过丰富的操作算子实现，不再像MapReduce那样只能写map和reduce了，如下图所示。



RDD的操作算子包括两类，一类叫做transformations，它是用来将RDD进行转化，构建RDD的血缘关系；另一类叫做actions，它是用来触发RDD的计算，得到RDD的相关计算结果或者将RDD保存的文件系统中。下图是RDD所支持的操作算子列表。

## 1.3 依赖

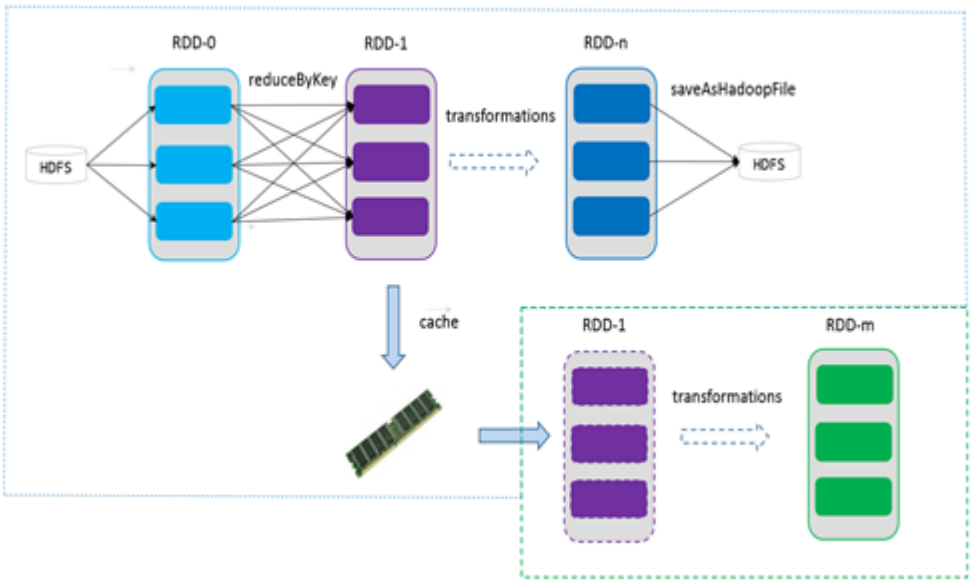
RDDs通过操作算子进行转换，转换得到的新RDD包含了从其他RDDs衍生所必需的信息，RDDs之间维护着这种血缘关系，也称之为依赖。如下图所示，依赖包括两种，一种是窄依赖，RDDs之间分区是一一对应的，另一种是宽依赖，下游RDD的每个分区与上游RDD(也称之为父RDD)的每个分区都有关，是多对多的关系。



RDD之间依赖关系

## 1.4 缓存

如果在应用程序中多次使用同一个RDD，可以将该RDD缓存起来，该RDD只有在第一次计算的时候会根据血缘关系得到分区的数据，在后续其他地方用到该RDD的时候，会直接从缓存处取而不用再根据血缘关系计算，这样就加速后期的重用。如下图所示，RDD-1经过一系列的转换后得到RDD-n并保存到hdfs，RDD-1在这一过程中会有个中间结果，如果将其缓存到内存，那么在随后的RDD-1转换到RDD-m这一过程中，就不会计算其之前的RDD-0了。




## 1.5 CheckPoint

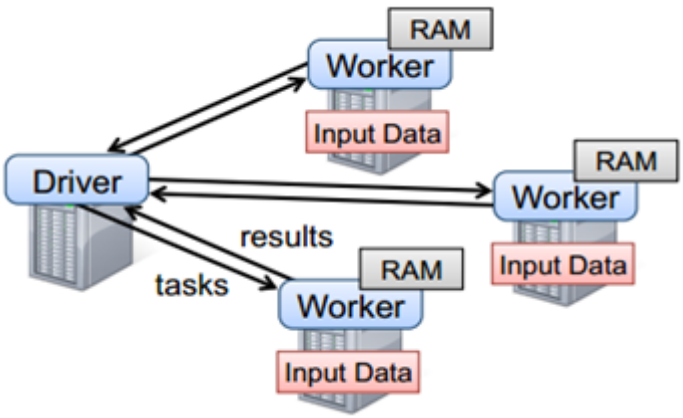
虽然RDD的血缘关系天然地可以实现容错，当RDD的某个分区数据失败或丢失，可以通过血缘关系重建。但是对于长时间迭代型应用来说，随着迭代的进行，RDDs之间的血缘关系会越来越长，一旦在后续迭代过程中出错，则需要通过非常长的血缘关系去重建，势必影响性能。为此，RDD支持checkpoint将数据保存到持久化的存储中，这样就可以切断之前的血缘关系，因为checkpoint后的RDD不需要知道它的父RDDs了，它可以从checkpoint处拿到数据。

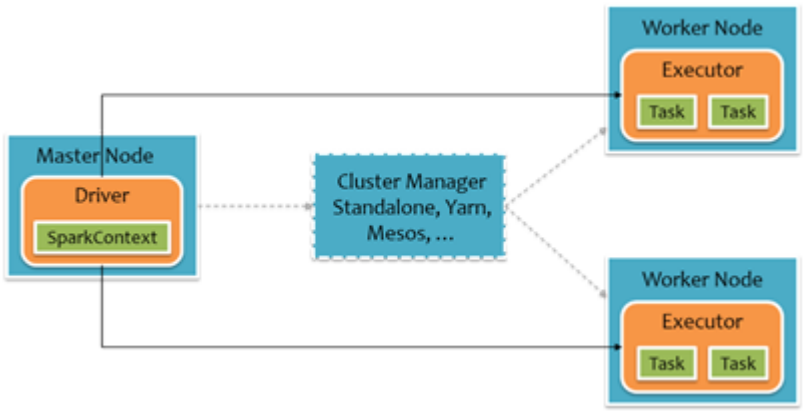
# 二、RDD编程

## 1 编程模型

在Spark中，RDD被表示为对象，通过对象上的方法调用来对RDD进行转换。经过一系列的transformations定义RDD之后，就可以调用actions触发RDD的计算，action可以是向应用程序返回结果(count, collect等)，或者是向存储系统保存数据(saveAsTextFile等)。在Spark中，只有遇到action，才会执行RDD的计算(即延迟计算)，这样在运行时可以通过管道的方式传输多个转换。

要使用Spark，开发者需要编写一个Driver程序，它被提交到集群以调度运行Worker，如下图所示。Driver中定义了一个或多个RDD，并调用RDD上的action，Worker则执行RDD分区计算任务。



## 2.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。

① 从集合中创建RDD，Spark主要提供了两种函数：parallelize和makeRDD

1）使用parallelize()从集合创建

```
scala> val rdd = sc.parallelize(Array(1,2,3,4,5,6,7,8))
rdd: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:24
```

2）使用makeRDD()从集合创建

```
scala> val rdd1 = sc.makeRDD(Array(1,2,3,4,5,6,7,8))
rdd1: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[1] at makeRDD at <console>:24
```

② 由外部存储系统的数据集创建

```
scala> val rdd2= sc.textFile("hdfs://spark1:9000/a.txt")
rdd2: org.apache.spark.rdd.RDD[String] = hdfs://spark1:9000/a.txt MapPartitionsRDD[1] at
textFile at <console>:24
```

③ 从其他RDD创建

见后续转换算子章节

## 3.RDD的转换(Transformations)算子

**RDD整体上分为Value类型和Key-Value类型**

### Value类型

**map(func)案例 [重要]**

作用：返回一个新的RDD，该RDD由每一个输入元素经过func函数转换后组成

需求：创建一个1-10数组的RDD，将所有元素*2形成新的RDD

```scala
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(1 to 10)
//2.将所有元素*2
val rdd2: RDD[Int] = rdd1.map(_*2)
//3.在驱动程序中，以数组的形式返回数据集的所有元素
val arr: Array[Int] = rdd2.collect()
//4.输出数组中的元素
arr.foreach(println)
```

**mapPartitions(func)案例 [重要]**

作用：类似于map，但独立地在RDD的每一个分区上运行，因此在类型为T的RDD上运行时，func的函数类型必须是Iterator[T] => Iterator[U]。假设有N个元素，有M个分区，那么map的函数的将被调用N次,而mapPartitions被调用M次,一个函数一次处理该分区。

需求：创建一个RDD，使每个元素*2组成新的RDD

```scala
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(1 to 10)
//2.将所有元素*2
val rdd2: RDD[Int] = rdd1.mapPartitions(iter => {
   iter.map(_ * 2)
})
//3.打印结果
rdd2.collect().foreach(println)
```

> map和mapPartition的区别
>
> map()：每次处理一条数据。
>
> mapPartition()：每次处理一个分区的数据

**mapPartitionsWithIndex(func)案例**

作用：类似于mapPartitions，但func带有一个整数参数表示分区的索引值，因此在类型为T的RDD上运 行时，func的函数类型必须是(Int, Interator[T]) => Iterator[U]；

需求：创建一个RDD，使每个元素跟所在分区形成一个元组组成一个新的RDD

```scala
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(1 to 10)
//2.使每个元素跟所在分区形成一个元组组成一个新的RDD
val rdd2: RDD[(Int, Int)] = rdd1.mapPartitionsWithIndex((index, iter) => {
    iter.map((index, _))
})
//3.打印新的RDD
rdd2.collect().foreach(println)
```

**flatMap(func)案例 [重要]**

作用：类似于map，但是每一个输入元素可以被映射为0或多个输出元素（所以func应该返回一个序 列，而不是单一元素）

需求：创建一个元素为1-5的RDD，运用flatMap创建一个新的RDD，新的RDD为（1,1, 2, 1, 2, 3, 1, 2, 3, 4, 1, 2, 3, 4, 5）

```
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(1 to 5)
//2.根据原RDD创建新RDD
val rdd2 = rdd1.flatMap(v => 1 to v)
//3.打印新的RDD
rdd2.collect().foreach(v=>{
    print(v + " ")
})
```

**glom案例**

作用：将每一个分区形成一个数组，形成新的RDD类型是RDD[Array[T]]

需求：创建一个4个分区的RDD，并将每个分区的数据放到一个数组

```
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(1 to 16,4)
//2.根据原RDD创建新RDD
val rdd2: RDD[Array[Int]] = rdd1.glom()
//3.打印
rdd2.collect().foreach(v=>{
    println(v.mkString(","))
})
```

应用场景：计算RDD中的最大值

**groupBy(func)案例 [重要]**

作用：分组，按照传入函数的返回值进行分组。将相同的key对应的值放入一个迭代器。

需求：创建一个RDD，按照元素模以2的值进行分组。

```
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(1 to 10)
//2.根据原RDD创建新RDD
val rdd2: RDD[(Int, Iterable[Int])] = rdd1.groupBy( v=> v%2)
//3.打印
rdd2.collect().foreach(println)
```

**filter(func)案例 [重要]**

作用：过滤。返回一个新的RDD，该RDD由经过func函数计算后返回值为true的输入元素组成。

需求：创建一个RDD（由字符串组成），过滤出一个新RDD（包含”xiao”子串）

```
//1.创建一个RDD
val rdd1: RDD[String] = sc.makeRDD(List("xiaoming","zhangsan","xiaohong","lisi","wangxiao"))
//2.根据原RDD创建新RDD
val rdd2: RDD[String] = rdd1.filter(v=>v.contains("xiao"))
//3.打印
rdd2.collect().foreach(println)
```

**distinct案例 [重要]**

作用：对源RDD进行去重后返回一个新的RDD

需求：创建一个RDD，使用distinct()对其去重。

```
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(List(1,2,3,3,5,6,7,1,2,3))
//2.根据原有RDD去重，产生新的RDD
val rdd2: RDD[Int] = rdd1.distinct()
//3.打印
rdd2.collect().foreach(println)
```

**coalesce(numPartitions)案例**

作用：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。

需求：创建一个4个分区的RDD，对其缩减分区不需要对其进行洗牌操作，增大分区时需要对其进行洗牌操作

```
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(1 to 10,4)
//2.对RDD重新分区
val rdd2: RDD[Int] = rdd1.coalesce(2)
//3.打印
println("分区数："+rdd2.getNumPartitions)

rdd2.collect()
```

**repartition(numPartitions) 案例**

作用：根据分区数，重新通过网络随机洗牌(shuffle)所有数据。无论减少或者增大分区都要进行洗牌操作

需求：创建一个4个分区的RDD，对其重新分区

```
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(1 to 10,4)
//2.重新分区
 val rdd2: RDD[Int] = rdd1.repartition(6)

 //3.打印
println("分区数："+rdd2.getNumPartitions)

rdd2.collect()
```

> coalesce和repartition的区别
>
> 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。
>
> repartition实际上是调用的coalesce，默认是进行shuffle的
>
> 注意：减少分区允许不进行shuffle过程，但是增大分区需要
>
> 所以coalesce可以在不进行shuffle的情况下减少分区，增大分区需要指定第二个参数为true
>
> 减少分区的应用场景：例如通过filter之后，有些分区数据量比较少，通过减少分区，防止数据倾斜
>
> 增大分区的应用场景：分区内数据量太大，通过增加分区提高并行度[提高执行效率]

**sortBy(func,[ascending],[numTasks]) 案例 [重要]**

作用；使用func先对数据进行处理，按照处理后的数据比较结果排序，默认为正序。

需求：创建一个RDD，按照不同的规则进行排序

```
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(List(1,12,4,5,2))
//按照自身大小排序
val rdd2: RDD[Int] = rdd1.sortBy(v=>v)
rdd2.collect().foreach(println)
//按照与3余数的大小排序
val rdd3: RDD[Int] = rdd1.sortBy(v=>v%3)
rdd3.collect().foreach(println)
```

**union(otherDataset)案例**

作用：对源RDD和参数RDD求并集后返回一个新的RDD

需求：创建两个RDD，求并集

```scala
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(1 to 5)
val rdd2: RDD[Int] = sc.makeRDD(4 to 8)
//2.计算两个RDD的并集
val rdd3: RDD[Int] = rdd1.union(rdd2)
//3.打印
rdd3.collect().foreach(println)
```

**subtract (otherDataset) 案例**

作用：计算差的一种函数，从第一个RDD减去第二个RDD的交集部分

需求：创建两个RDD，求第一个RDD与第二个RDD的差集

```scala
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(1 to 5)
val rdd2: RDD[Int] = sc.makeRDD(4 to 8)
//2.计算两个RDD的差集
val rdd3: RDD[Int] = rdd1.subtract(rdd2)
//3.打印
rdd3.collect().foreach(println)
//结果：1 2 3
```

**intersection(otherDataset)案例**

作用：对源RDD和参数RDD求交集后返回一个新的RDD

需求：创建两个RDD，求两个RDD的交集

```scala
//1.创建一个RDD
val rdd1: RDD[Int] = sc.makeRDD(1 to 5)
val rdd2: RDD[Int] = sc.makeRDD(4 to 8)
//2.计算两个RDD的交集
val rdd3: RDD[Int] = rdd1.intersection(rdd2)
//3.打印
rdd3.collect().foreach(println)
//结果：4 5
```

## Key-Value pair类型

**groupByKey案例**

作用：groupByKey也是对每个key进行操作，但只生成一个sequence。

需求：创建一个pairRDD，将相同key对应值聚合到一个sequence中，

并计算相同key对应值的相加结果。

```
//1.创建一个RDD
val rdd1: RDD[String] = sc.makeRDD(List("one", "two", "two", "three", "three", "three"))
//2.相同key对应值的相加结果
val rdd2: RDD[(String, Int)] = rdd1.map((_,1)).groupByKey().map(v=>(v._1,v._2.size))
//3.打印
rdd2.collect().foreach(println)

结果(three,3)  (two,2)  (one,1)
```

**reduceByKey(func, [numTasks])案例**

在一个(K,V)的RDD上调用，返回一个(K,V)的RDD，使用指定的reduce函数，将相同key的值聚合到一起，reduce任务的个数可以通过第二个可选的参数来设置。

需求：创建一个pairRDD，计算相同key对应值的相加结果

```
//1.创建一个RDD
val rdd1: RDD[(String,Int)] = sc.makeRDD(List(("female",1),("male",5),("female",5),("male",2)))
//2.相同key对应值的相加结果
val rdd2: RDD[(String, Int)] = rdd1.reduceByKey(_+_)
//3.打印
rdd2.collect().foreach(println)

结果：(male,7)  (female,6)
```

> reduceByKey和groupByKey的区别
>
> 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].
>
> groupByKey：按照key进行分组，直接进行shuffle。
>
> 开发指导：reduceByKey比groupByKey，建议使用。但是需要注意是否会影响业务逻辑。

**aggregateByKey案例**

参数：(zeroValue:U,[partitioner: Partitioner]) (seqOp: (U, V) => U,combOp: (U, U) => U)

作用：在kv对的RDD中，，按key将value进行分组合并，合并时，将每个value和初始值作为seq函数的参数，进行计算，返回的结果作为一个新的kv对，然后再将结果按照key进行合并，最后将每个分组的value传递给combine函数进行计算（先将前两个value进行计算，将返回结果和下一个value传给combine函数，以此类推），将key与计算结果作为一个新的kv对输出。

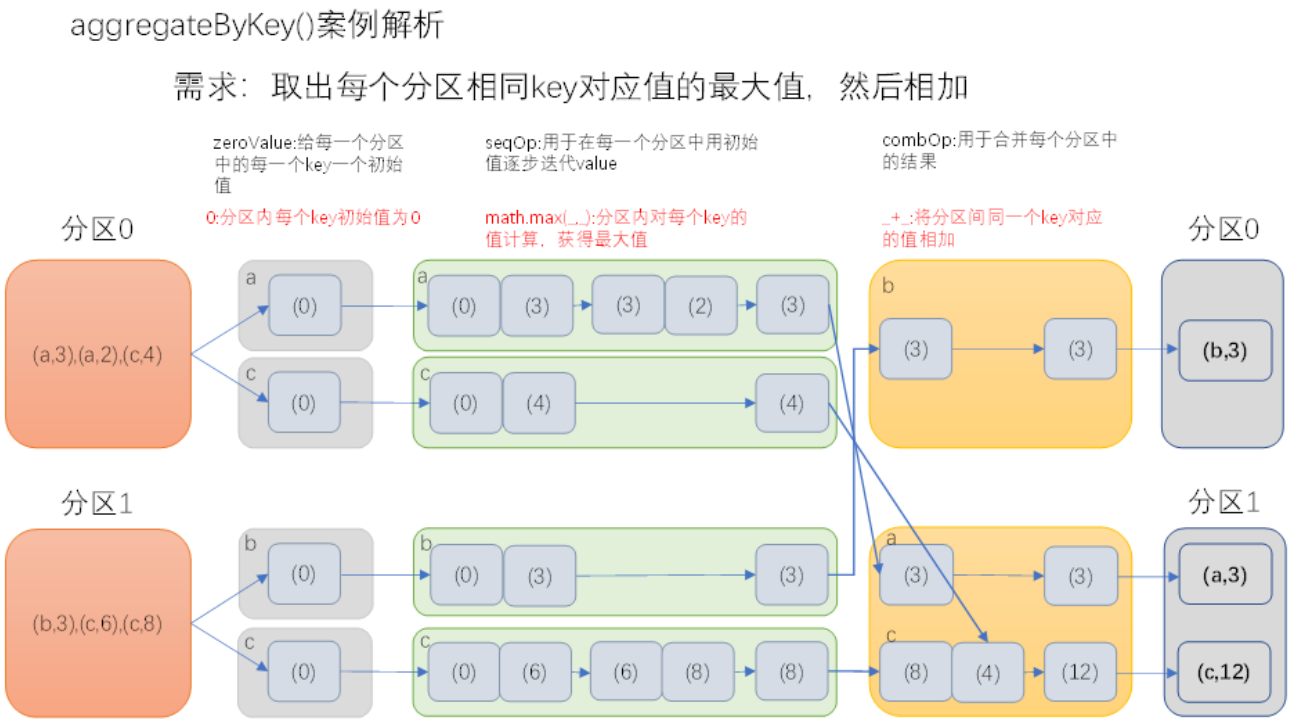
参数描述：

（1）zeroValue：给每一个分区中的每一个key一个初始值；

（2）seqOp：函数用于在每一个分区中用初始值逐步迭代value；

（3）combOp：函数用于合并每个分区中的结果。

需求：创建一个pairRDD，取出每个分区相同key对应值的最大值，然后相加




```
//1.创建一个RDD
val rdd1: RDD[(String,Int)] = sc.makeRDD(List(("a",3),("a",2),("c",4),("b",3),("c",6),
("c",8)),2)
//2.取出每个分区相同key对应值的最大值
val rdd2: RDD[(String, Int)] = rdd1.aggregateByKey(0)((v1,v2)=> if(v1 > v2) v1 else v2 ,_+_)
//3.打印
rdd2.collect().foreach(println)

结果：(b,3)  (a,3)  (c,12)
```

**sortByKey([ascending],[numTasks]) 案例**

作用：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

需求：创建一个pairRDD，按照key的正序和倒序进行排序

```
//1.创建一个RDD
val rdd1: RDD[(Int,String)] = sc.makeRDD(List((3,"aa"),(6,"cc"),(2,"bb"),(1,"dd")))
//2.按照key的正序
rdd1.sortByKey(true).collect().foreach(print)    //(1,dd)(2,bb)(3,aa)(6,cc)
//3.按照key的降序
rdd1.sortByKey(false).collect().foreach(print)  //(6,cc)(3,aa)(2,bb)(1,dd)
```

**mapValues案例**

针对于(K,V)形式的类型只对V进行操作

需求：创建一个pairRDD，并将value添加字符串"_x"

```
//1.创建一个RDD
val rdd1: RDD[(Int,String)] = sc.makeRDD(List((1,"a"),(1,"d"),(2,"b"),(3,"c")))
//2.给value增加一个_x
val rdd2: RDD[(Int, String)] = rdd1.mapValues(v=>v+"_x")
//3.打印
rdd2.collect().foreach(print)

结果 (1,a_x)(1,d_x)(2,b_x)(3,c_x)
```

**join(otherDataset,[numTasks]) 案例**

作用：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

需求：创建两个pairRDD，并将key相同的数据聚合到一个元组。

```
//1.创建一个RDD
val rdd1: RDD[(Int,String)] = sc.makeRDD(List((1,"a"),(2,"b"),(3,"c")))
val rdd2: RDD[(Int,Int)] = sc.makeRDD(List((1,4),(2,5),(3,6)))
//2.join操作
val rdd3: RDD[(Int, (String, Int))] = rdd1.join(rdd2)
//3.打印
rdd3.collect().foreach(println)

结果：(1,(a,4))
     (2,(b,5))
     (3,(c,6))
```

# 4.RDD的行动(Action)算子

> spark调用rdd的转换算子，默认都是延迟执行的，只有调用行动算子才会触发之前的转换算子的调用
>
> rdd调用转换算子返回的还是rdd对象，如果调用行动算子返回的是scala对象
>
> 只有rdd才可以调用spark中的转换或者行动算子

## reduce(func)案例

作用：通过func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。

需求：创建一个RDD，将所有元素聚合得到结果。

```
val rdd1 = sc.makeRDD(1 to 10)
println(rdd1.reduce(_ + _))     //输出 55

val rdd2 = sc.makeRDD(Array(("a",1),("a",3),("c",3),("d",5)))
println(rdd2.reduce((v1, v2) => (v1._1 + v2._1, v1._2 + v2._2)))  //输出 (aacd,12)
```

## collect()案例

作用：在驱动程序中，以数组的形式返回数据集的所有元素。

需求：创建一个RDD，并将RDD内容收集到Driver端打印

## count()案例

作用：返回RDD中元素的个数

需求：创建一个RDD，统计该RDD的条数

```
val rdd1 = sc.makeRDD(1 to 10)
println(rdd1.count())           //输出 10
```

## first()案例

作用：返回RDD中的第一个元素

需求：创建一个RDD，返回该RDD中的第一个元素

```
val rdd1 = sc.makeRDD(1 to 10)
println(rdd1.first())
```

## take(n)案例

作用：返回一个由RDD的前n个元素组成的数组

需求：创建一个RDD，统计该RDD的条数

```
val rdd1 = sc.makeRDD(1 to 10)
println(rdd1.take(5).mkString(","))      //输出：1,2,3,4,5
```

## takeOrdered(n)案例

作用：返回该RDD排序后的前n个元素组成的数组

需求：创建一个RDD，统计该RDD的条数

```
val rdd = sc.parallelize(Array(2,5,4,6,8,3))
println(rdd.takeOrdered(3).mkString(","))    //输出：2,3,4
```

## aggregate案例

参数：(zeroValue: U)(seqOp: (U, T) ⇒ U, combOp: (U, U) ⇒ U)

作用：aggregate函数将每个分区里面的元素通过seqOp和初始值进行聚合，然后用combine函数将每个分区的结果和初始值(zeroValue)进行combine操作。这个函数最终返回的类型不需要和RDD中元素类型一致。

## fold(num)(func)案例

作用：折叠操作，aggregate的简化操作，seqop和combop一样。

需求：创建一个RDD，将所有元素相加得到结果

## saveAsTextFile(path)

作用：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

```
对应读取文件的方法：sc.textFile("")
```

## saveAsSequenceFile(path)

作用：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。

```
针对rdd中的元素是(k,v)格式的数据进行保存
sc.sequenceFile()可以读取rdd.saveAsSequenceFile()保存的数据
```

## saveAsObjectFile(path)

作用：用于将RDD中的元素序列化成对象，存储到文件中。

```
对保存的数据进行序列化
sc.objectFile("")
```

## countByKey()案例

作用：针对(K,V)类型的RDD，返回一个(K,Int)的map，表示每一个key对应的元素个数。

需求：创建一个PairRDD，统计每种key的个数

```
val rdd = sc.parallelize(List((1,3),(1,2),(1,4),(2,3),(3,6),(3,8)))
val map: collection.Map[Int, Long] = rdd.countByKey()
println(map)

输出：Map(1 -> 3, 2 -> 1, 3 -> 2)
```

### foreach(func)案例

作用：在数据集的每一个元素上，运行函数func进行更新。

需求：创建一个RDD，对每个元素进行打印

# 三、RDD相关概念

## RDD的依赖关系

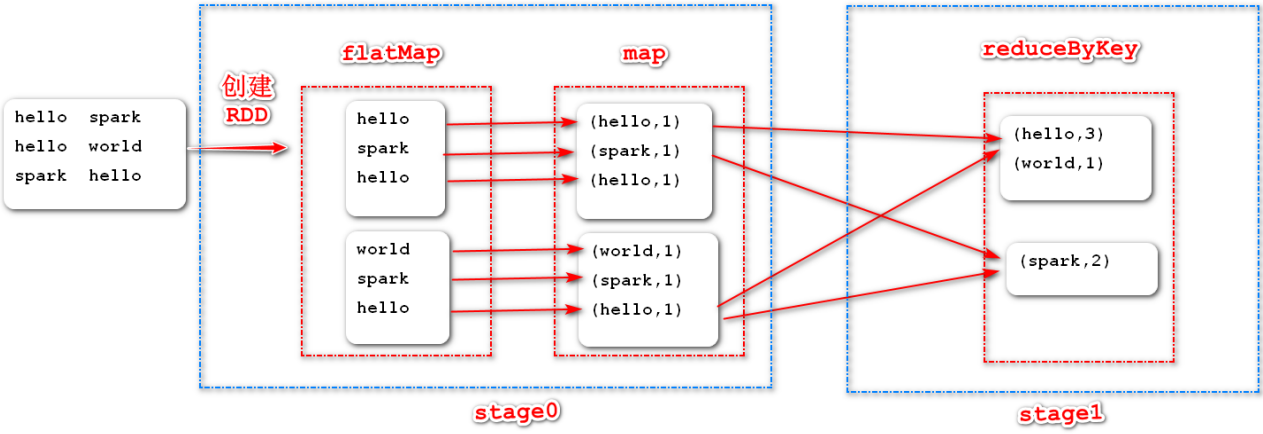
RDD依赖关系也称为RDD的血统，描述了RDD间的转换关系 。Spark将RDD间依赖关系分为了`宽依赖|ShuffleDependency`、`窄依赖|NarrowDependency`，Spark在提交任务的时候会根据转换算子逆向推导出所有的Stage。然后计算推导的stage的分区用于表示该Stage执行的`并行度`。

### 窄依赖：

- 父RDD和子RDD partition之间的关系是一对一的。或者父RDD一个partition只对应一个子RDD的partition情况下的父RDD和子RDD partition关系是多对一的。不会有shuffle的产生。父RDD的一个分区去到子RDD的一个分区。
- 可以理解为独生子女

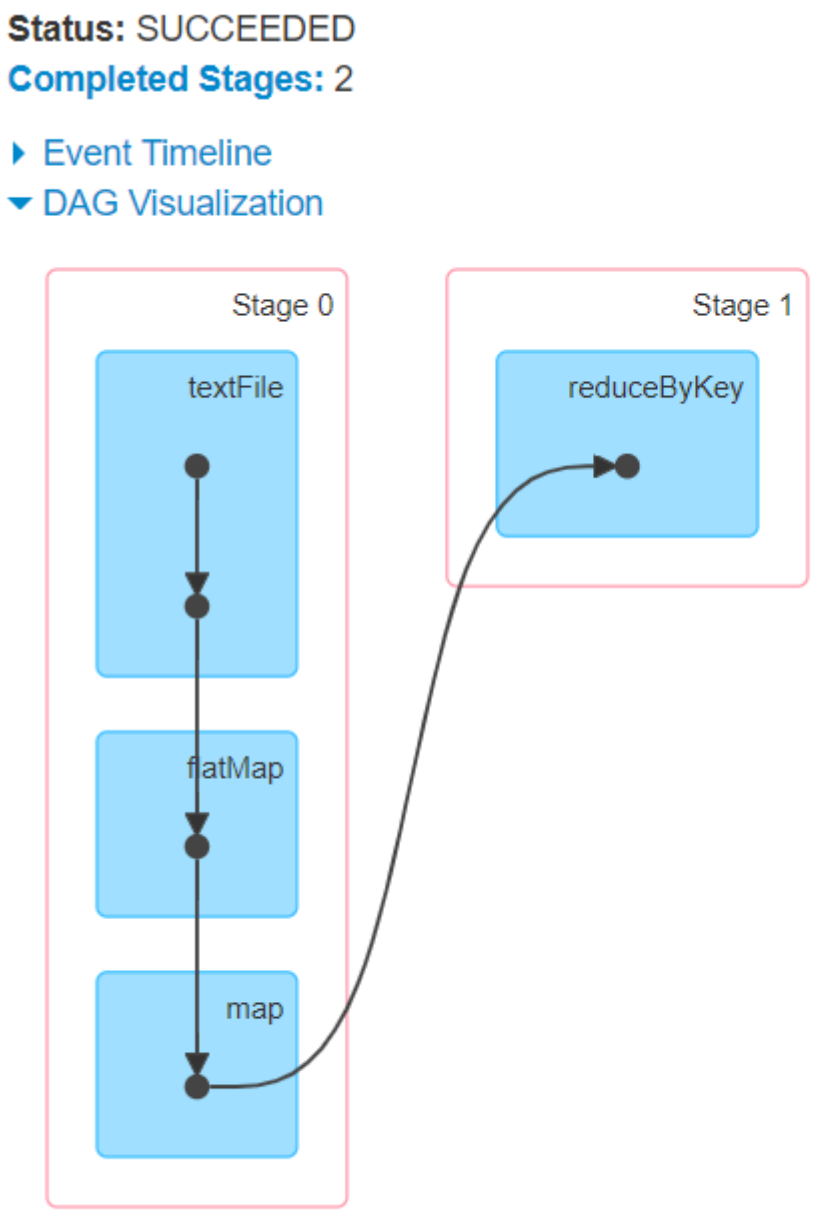
### 宽依赖：

- 父RDD与子RDD partition之间的关系是一对多。会有shuffle的产生。父RDD的一个分区的数据去到子RDD的不同分区里面。
- 可以理解为超生




## DAG

DAG(Directed Acyclic Graph)叫做有向无环图，原始的RDD通过一系列的转换就就形成了DAG，根据RDD之间的依赖关系的不同将DAG划分成不同的Stage，对于窄依赖，partition的转换处理在Stage中完成计算。对于宽依赖，由于有Shuffle的存在，只能在parent RDD处理完成后，才能开始接下来的计算，因此宽依赖是划分stage的依据。




**任务划分**

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：根据RDD之间的依赖关系的不同将Job划分成不同的Stage，遇到一个宽依赖则划分一个Stage。

4）Task：Stage是一个TaskSet，将Stage划分的结果发送到不同的Executor执行即为一个Task。

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

## RDD的缓存

RDD通过persist方法或cache方法可以将前面的计算结果缓存，默认情况下 persist() 会把数据缓存在 JVM 的堆空间中。

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。

```
/** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
def persist(): this.type = persist(StorageLevel.MEMORY_ONLY)

/** Persist this RDD with the default storage level (`MEMORY_ONLY`). */
def cache(): this.type = persist()
```

通过查看源码发现cache最终也是调用了persist方法，默认的存储级别都是仅在内存存储一份，Spark的存储级别还有好多种，存储级别在object StorageLevel中定义的。

```
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(false, false, true, false)
```

在存储级别的末尾加上“_2”来把持久化数据存为两份(备份)

| 级　别 | 使用的空间 | CPU时间 | 是否在内存中 | 是否在磁盘上 | 备　注 |
|---|---|---|---|---|---|
| MEMORY_ONLY | 高 | 低 | 是 | 否 | |
| MEMORY_ONLY_SER | 低 | 高 | 是 | 否 | |
| MEMORY_AND_DISK | 高 | 中等 | 部分 | 部分 | 如果数据在内存中放不下，则溢写到磁盘上 |
| MEMORY_AND_DISK_SER | 低 | 高 | 部分 | 部分 | 如果数据在内存中放不下，则溢写到磁盘上。在内存中存放序列化后的数据 |
| DISK_ONLY | 低 | 高 | 否 | 是 | |

```
val rdd1: RDD[String] = sc.makeRDD(List("zhangsan"))
val rdd2: RDD[String] = rdd1.map(_ + System.currentTimeMillis()).cache()
rdd2.foreach(println)
rdd2.foreach(println)     //两次打印结果相同
```

## RDD的checkpoint

Spark中对于数据的保存除了持久化操作之外，还提供了一种检查点的机制，检查点（本质是通过将RDD写入Disk做检查点）是为了通过lineage做容错的辅助，lineage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，如果之后有节点出现问题而丢失分区，从做检查点的RDD开始重做Lineage，就会减少开销。检查点通过将数据写入到HDFS文件系统实现了RDD的检查点功能。

为当前RDD设置检查点。该函数将会创建一个二进制的文件，并存储到checkpoint目录中，该目录是用SparkContext.setCheckpointDir()设置的。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。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。

```
sc.setCheckpointDir("hdfs://spark1:9000/rdd-checkpoint")

val rdd1: RDD[String] = sc.makeRDD(List("zhangsan"))
    val rdd2: RDD[String] = rdd1.map(_ + System.currentTimeMillis())

rdd2.checkpoint()   //此处可以通过查看checkpoint方法的注释了解它的运行机制

rdd2.collect().foreach(println)     //zhangsan1588656910532
rdd2.collect().foreach(println)     //zhangsan1588656910641
rdd2.collect().foreach(println)     //zhangsan1588656910641
```

# 四、Spark对外部数据库的写入

### MySQL数据库

① 添加依赖

```
<dependency>
    <groupId>mysql</groupId>
     <artifactId>mysql-connector-java</artifactId>
      <version>5.1.38</version>
</dependency>
```

② 将spark计算的数据写入mysql

```
val data = sc.parallelize(List("zhangsan", "lisi","wangwu"),2)

data.foreachPartition(iter=>{

   val conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/test1","root","123456")
   iter.foreach(data=>{
        val ps = conn.prepareStatement("insert into spark_user(name) values (?)")
        ps.setString(1, data)
        ps.executeUpdate()
   })
   conn.close()

})
```

```
create table spark_user(
    id  int primary key auto_increment,
    name varchar(50)
)
```

## HBase数据库

①. 添加依赖

```
<dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>2.9.0</version>
</dependency>
<dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-auth</artifactId>
        <version>2.9.0</version>
</dependency>

<dependency>
        <groupId>org.apache.hbase</groupId>
        <artifactId>hbase-server</artifactId>
        <version>1.5.0</version>
</dependency>

<dependency>
        <groupId>org.apache.hbase</groupId>
        <artifactId>hbase-client</artifactId>
        <version>1.5.0</version>
</dependency>
```

②. 将spark计算结果添加到hbase中

> create_namespace "baizhi" 在hbase中提前创建namespace

> create "baizhi:spark_fruit","info" 在namespace下创建表

```
val rdd = sc.parallelize(List((1,"苹果",11), (2,"香蕉",12), (3,"梨",13)))

 // 批量插入
rdd.foreachPartition(x => {
        val conf = HBaseConfiguration.create();
        conf.set("hbase.zookeeper.quorum","spark1");
        val conn = ConnectionFactory.createConnection(conf);
        val table = conn.getTable(TableName.valueOf("baizhi:spark_fruit"))
        val puts = new java.util.ArrayList[Put]()
        x.foreach(y => {
          // 将数组插入hbase
          val wordPut = new Put(Bytes.toBytes(y._1))
          wordPut.addColumn(Bytes.toBytes("info"), Bytes.toBytes("name"), Bytes.toBytes(y._2))
          wordPut.addColumn(Bytes.toBytes("info"), Bytes.toBytes("price"), Bytes.toBytes(y._3))
          puts.add(wordPut)
        })
        table.put(puts)
})
```

③. 读取hbase中的数据

```
val hadoopConfig = new Configuration()
hadoopConfig.set(HConstants.ZOOKEEPER_QUORUM,"spark1")//配置Hbase连接参数
hadoopConfig.set(TableInputFormat.INPUT_TABLE,"baizhi:spark_fruit") //配置扫描的表

sc.newAPIHadoopRDD(hadoopConfig,classOf[TableInputFormat],classOf[ImmutableBytesWritable],classO
f[Result])
      .map(t=>{
        var rowKey=Bytes.toInt(t._1.get())
        var name=Bytes.toString(t._2.getValue("info".getBytes(),"name".getBytes()))
        var price=Bytes.toInt(t._2.getValue("info".getBytes(),"price".getBytes()))
        (rowKey,name,price)
      }).collect()
      .foreach(t=>{
        println(t)
})
```

# 五、RDD编程进阶

## 1.累加器 accumulators

如果在Driver定义了变量，下游的算子在使用Driver变量的时候会通过网络下载Driver的变量。因此定义在Driver的变量一般的是只读的，并且支持序列化。

```
//1.创建SparkContext
val sparkConf = new SparkConf()
.setMaster("local[*]")
.setAppName("wordcount")
val sc=new SparkContext(sparkConf)

var count=0
sc.makeRDD(List(1,2,3,4,5))
.foreach(item=> count += item)//并行执行 远程复制count变量，只是修改远程变量

println(s"count:${count}") //0

//4.关闭SparkContext
sc.stop();
```

以上因为count变量定义在Driver中，foreach算子是并行执行的算子，远程的Task会向Driver下载变量count，所有下游的Task都存储了count变量的副本，因此拿到都是0.当Task结束后远程count变量的值并不会传递给Driver。因此count最后依然是0.如果定义如下：

```
var count=0
sc.makeRDD(List(1,2,3,4,5))
.collect()//把RDD拿到Driver端，后续的foreach并不是并行执行
.foreach(item=> count += item) //count = 15
```

Spark提供了`accumulators`(累加器)专门用于Driver端和Task端传递计数变量。

```
var count=sc.longAccumulator("a1")//long 类型的累加器
sc.makeRDD(List(1,2,3,4,5)).foreach(item=> count.add(item))
println(count.value) //15
```

## 2.广播变量

如果你的算子函数中，使用到了特别大的数据，那么，这个时候，推荐将该数据进行广播。这样的话，就不至于将一个大数据拷贝到每一个task上去。而是给每个节点拷贝一份，然后节点上的task共享该数据。这样的话，就可以减少大数据在节点上的内存消耗。并且可以减少数据到节点的网络传输消耗。

需求：读取文件中的单词，然后进行筛选，最终保留的数据为Set集合中规范的

```
//定义一个set集合
 val set = Set("hello","hehe")//数据量比较大
//从hfds读取文件，创建RDD
val rdd1: RDD[String] = sc.textFile("hdfs://spark1:9000/a.txt")
//对读取的单词执行flatMap操作，并且进一步筛选，最终保留set集合中规范的数据
rdd1.flatMap(_.split(" ")).filter(v=>{
        set.contains(v)
}).collect().foreach(println)
```

使用broadcast改造上述代码

```
//定义一个set集合
val set = Set("hello","hehe")
//通过set集合 创建广播变量
val broadCast: Broadcast[Set[String]] = sc.broadcast(set)

//从hfds读取文件，创建RDD
val rdd1: RDD[String] = sc.textFile("hdfs://spark1:9000/a.txt")
//对读取的单词执行flatMap操作，并且进一步筛选，最终保留set集合中规范的数据
rdd1.flatMap(_.split(" ")).filter(v=>{
        //通过广播变量获取set集合
     broadCast.value.contains(v)
}).collect().foreach(println)
```