# **spark RDD常用函数/操作**

[](https://www.jianshu.com/u/b47e6f52116d)

[pcz](https://www.jianshu.com/u/b47e6f52116d" \t "/Users/liuqing/Documents\\x/_blank)关注

0.3932017.09.19 22:01:05字数 2,087阅读 1,668

# **spark RDD常用函数/操作**

文中的代码均可以在spark-shell中运行。

## **transformations**

### **map(func)**

集合内的每个元素通过function映射为新元素

val a = Array(1,2,3,4)val pa = sc.parallelize(a)val resultRdd = pa.map( \_ + 1)

注意对于所有transformation操作，生成的都是一个新的RDD（这里就是resultRdd），并不实际进行运算，只有对RDD进行action操作时才会实际计算并产生结果：

scala> resultRdd.collect

res3: Array[Int] = Array(2, 3, 4, 5)

以下的transformation操作同理。

### **filter(func)**

通过func过滤集合的元素，func的返回值必须是Boolean类型

val a = Array(1,2,3,4)val pa = sc.parallelize(a)val resultRdd = pa.filter( \_ > 1)

scala> resultRdd.collect

res4: Array[Int] = Array(2, 3, 4)

### **flatmap(func)**

通过func将集合内的每一元素，映射为一个序列(具体的是TraversableOnce[?]，这里可以不用管这个类型，spark会自己作隐式转换，一般的可以顺序迭代的序列都可以)。  
说起来可能不好理解，举个例子。还是[1,2,3,4]吧，假设func是这样的：x => Array(x+0.1, x+0.2)，也就是返回一个序对，flatMap流程可以看作是先对每个元素执行func，得到

[(1.1, 1.2), (2.1, 2.2), (3.1, 3.2), (4.1, 4.2)]

最后将所有的序列展平，就得到：

[1.1, 1.2, 2.1, 2.2, 3.1, 3.2, 4.1, 4.2]

代码形式：

val a = Array(1,2,3,4)val pa = sc.parallelize(a)val resultRdd = pa.flatMap( x => Array(x+0.1,x+0.2) )

scala> resultRdd.collect

res8: Array[Double] = Array(1.1, 1.2, 2.1, 2.2, 3.1, 3.2, 4.1, 4.2)

### **mapPartitions(func)**

和map类似，但它是在RDD的每个分区分别运行，可以理解成将一个分区内的元素映射成一个新分区，最后将所有新分区拼起来成为一个新RDD。

当对T类型的RDD使用此函数时，func的签名必须是Iterator<T> => Iterator<U>

val a = Array(1,2,3,4)val pa = sc.parallelize(a)val resultRdd = pa.mapPartitions(iter => iter.map(\_+1)) // 不要将这里的iter.map和RDD的map弄混了，这是scala内置的针对集合的操作

scala> resultRdd.collect

res11: Array[Int] = Array(2, 3, 4, 5)

### **mapPartitionsWithIndex(func)**

类似mapPartitions(func)，但是多提供一个分区的索引号信息  
所以对于元素为T类型的RDD，func 的类型签名应该是(Int, Iterator<T>) => Iterator<U>

val a = Array(1,2,3,4)val pa = sc.parallelize(a)val resultRdd = pa.mapPartitionsWithIndex( (index, iter) => iter.map( x => (x,s"分区$index")) )

scala> resultRdd.collect

res14: Array[(Int, String)] = Array((1,分区0), (2,分区1), (3,分区2), (4,分区3))

可以看到各个分区的情况，这里在集群结构不同的时候结果也会不同，这取决于RDD分布在哪些分区上。

### **sample(withReplacement, fraction, seed)**

采样函数，以一定的概率对数据进行采样

* withReplacement: 第一个参数决定在采样完成后是否将样本再放回去，类似于抽签完成后再把签放回去留给后面的人抽。
* fraction: 理解成概率比较好，也就是说每个元素以fraction的概率被抽到。[这个答案](https://link.jianshu.com/?t=https://stackoverflow.com/questions/32837530/how-to-get-a-sample-with-an-exact-sample-size-in-spark-rdd" \t "/Users/liuqing/Documents\\x/_blank)解释得比较好
* seed: 随机数生成器的种子

val a = Array(1,2,3,4)val pa = sc.parallelize(a)val resultRdd = pa.sample(true,0.5)

resultRdd.collect

多次运行的结果不同：

scala> pa.sample(false,0.5).collect

res52: Array[Int] = Array(1, 3)

scala> pa.sample(false,0.5).collect

res53: Array[Int] = Array(1, 2)

withReplacement设为true，也就是可以放回的情况，这可能会产生重复元素：

scala> pa.sample(true,0.5).collect

res60: Array[Int] = Array(1, 2, 2)

scala> pa.sample(true,0.5).collect

res61: Array[Int] = Array(3, 4)

### **union(otherDataset)**

求并集。

示例：

val pa = sc.parallelize( Array(1,2))val pb = sc.parallelize(Array(3,4))

pa.union(pb).collect

结果：

res63: Array[Int] = Array(1, 2, 3， 4）

### **intersection(otherDataset)**

求交集。

示例：

val pa = sc.parallelize( Array(1,2,3))val pb = sc.parallelize(Array(3,4,5))

pa.intersection(pb).collect

结果：

res66: Array[Int] = Array(3)

### **distinct([numTasks]))**

去重。numTasks是可选参数，表示分配成几个任务执行。

示例：

val pa = sc.parallelize(Array(0,1,1,2,2,3))

pa.distinct.collect

结果：

res92: Array[Int] = Array(0, 1, 2, 3)

### **groupByKey([numTasks])**

分组函数。

* 对类型 (K, V) 的数据集使用，返回(K, Iterable<V>)类型的数据集
* 如果想在分组后使用sum,average等聚合函数，最好使用 reduceByKey 或 aggregateByKey，这将获得更好的性能
* 默认的并行度依赖于父RDD，也可以传入可选参数numTasks指定并行任务数量。

示例：

val pa = sc.parallelize(Array( "a" -> 1,"a" ->2, "b" -> 3))

pa.groupByKey.collect

结果：

res99: Array[(String, Iterable[Int])] = Array((a,CompactBuffer(1, 2)), (b,CompactBuffer(3)))

### **reduceByKey(func, [numTasks])**

按照key分组然后聚集，类似于SQL中的groupby之后再使用聚集函数。

当一个 (K, V) 类型的数据集调用此函数， 返回一个同样是(K, V) 类型的数据集。

示例：

val pa = sc.parallelize(Array( "a" -> 1,"a" ->2,"a" ->3, "b" -> 4))

pa.reduceByKey( (x,y) => x+y).collect

结果：

res110: Array[(String, Int)] = Array((a,6), (b,4))

其实就是先分组一下，再对每一组内进行reduce。

### **aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])**

先分组，再在每个分组内调用aggregate进行折叠，与reduceByKey的区别在于折叠结果可以是不同的类型。

对(K, V) 的数据集调用，返回 (K, U)类型的数据集。

可以参考aggregate。

val pa = sc.parallelize(Array("a" -> 1,"a" ->2,"a" ->3, "b" -> 4))val r = pa.aggregateByKey("")(

(x:String,y:Int) => x+y.toString\*2,

(x:String,y:String) => x+y)

结果：

scala> r.collect

res135: Array[(String, String)] = Array((a,112233), (b,44))

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

排序

* ascending: 可选，是否升序排序
* numTasks: 可选，并发任务数量  
  对于 (K, V) 的数据集进行操作，返回同样是(K, V)类型的数据集，其中K实现了Orderedtrait，也就是可以排序。

val pa = sc.parallelize(Array("b" -> 1,"d" ->2,"a" ->3, "c" -> 4))

pa.sortByKey().collect

结果：

res139: Array[(String, Int)] = Array((a,3), (b,1), (c,4), (d,2))

### **join(otherDataset, [numTasks])**

两个集合的内积，对应数据库里的inner join。

对 一个(K, V)和 (K, W)类型的数据集操作，返回 (K, (V, W)) 类型的数据集。

另外还有对外积的支持：leftOuterJoin、rightOuterJoin、fullOuterJoin，与数据库里的相应概念相同

看一个例子：

val pa = sc.parallelize(

Array("a" -> 1,

"b" -> 2, "b" -> 3))val pb = sc.parallelize(

Array("b" -> 2, "b" -> 3,

"d" -> 4))

内积：

scala> pa.join(pb).collect

res140: Array[(String, (Int, Int))] = Array((b,(2,2)), (b,(2,3)), (b,(3,2)), (b,(3,3)))

左外积：

scala> pa.leftOuterJoin(pb).collect

res141: Array[(String, (Int, Option[Int]))] = Array((a,(1,None)), (b,(2,Some(2))), (b,(2,Some(3))), (b,(3,Some(2))), (b,(3,Some(3))))

右外积：

scala> pa.rightOuterJoin(pb).collect

res142: Array[(String, (Option[Int], Int))] = Array((d,(None,4)), (b,(Some(2),2)), (b,(Some(2),3)), (b,(Some(3),2)), (b,(Some(3),3)))

全外积：

scala> pa.fullOuterJoin(pb).collect

res143: Array[(String, (Option[Int], Option[Int]))] = Array((d,(None,Some(4))), (a,(Some(1),None)), (b,(Some(2),Some(2))), (b,(Some(2),Some(3))), (b,(Some(3),Some(2))), (b,(Some(3),Some(3))))

需要注意的是外积结果的元素变成了Option类型

### **cogroup(otherDataset, [numTasks])**

对两个数据集分别进行分组，然后把分组结果连接来作为元素。

对(K, V) 和 (K, W)类型操作，返回 (K, (Iterable<V>, Iterable<W>)) 类型，别名groupWith。

val pa = sc.parallelize(Array("a" -> 1, "b" -> 2, "b" -> 3))val pb = sc.parallelize(Array("a" -> "a", "b" -> "b", "b" -> "c"))

pa.cogroup(pb).collect

结果：

res152: Array[(String, (Iterable[Int], Iterable[String]))] = Array((a,(CompactBuffer(1),CompactBuffer(a))), (b,(CompactBuffer(2, 3),CompactBuffer(b, c))))

### **cartesian(otherDataset)**

对两个集合求笛卡尔积，对T 和 U类型操作，返回 (T, U)类型。

val pa = sc.parallelize(Array(1,2))val pb = sc.parallelize(Array(3,4))

pa.cartesian(pb).collect

结果：

res156: Array[(Int, Int)] = Array((1,3), (1,4), (2,3), (2,4))

### **pipe(command, [envVars])**

把每个分区输出到stdin，然后执行命令，最后读回stdout，以每行为元素，生成新的RDD。注意这里执行命令的单位是****分区****，不是元素。

在/home/zeta/目录新建一个脚本test.sh：

#!/bin/bash

while read LINE; do

echo e${LINE}

done

然后spark-shell里执行：

val pa = sc.parallelize(Array(1,2,3,4,5,6))

pa.pipe("/home/zeta/test.sh").collect

结果:

res181: Array[String] = Array(e1, e2, e3, e4, e5, e6)

### **coalesce(numPartitions)**

将RDD的分区数量减少到numPartitions个，在对一个大数据集进行filter操作之后，调用一下减少分区数量可以提高效率。

### **repartition(numPartitions)**

随机打乱RDD内全部分区的数据，并且平衡一下。

## **actions**

### **collect()**

以数组的形式返回集合内的所有元素

### **count()**

返回数据集内的元素个数

### **foreach(func)**

对数据集的每个元素执行func。

注意几点：

1. 副作用：分布式情况下，每个executor拥有自己的执行空间，所以变量不是全局共享的，对变量的副作用将导致未定义行为。这时候应该使用 [Accumulator](https://link.jianshu.com/?t=http://spark.apache.org/docs/latest/rdd-programming-guide.html" \l "accumulators" \t "/Users/liuqing/Documents\\x/_blank)。
2. 闭包：另外注意一些闭包引用的问题[Understanding closures](https://link.jianshu.com/?t=http://spark.apache.org/docs/latest/rdd-programming-guide.html" \l "understanding-closures-a-nameclosureslinka" \t "/Users/liuqing/Documents\\x/_blank)

### **first()**

返回数据集内的第一个元素

### **take(n)**

以数组的形式返回数据集内前n个元素

### **reduce(func)**

reduce操作，为了在并行计算下可以得到正确结果，这个函数要满足交换律和结合律，也就是数据集和这个运算必须构成一个交换群。

val pa = sc.parallelize(Array(1,2,3))

pa.reduce( (x,y) => x+y )

结果：

res184: Int = 6

### **aggregate(zeroValue)(seqOp, combOp)**

这个函数的操作流程可以看作两步

1. 在RDD的各个分区内调用seqOp操作进行折叠，它类似于fold
2. 调用combOp将各个分区的结果聚合起来

它的签名是这样的：

def aggregate[U](zeroValue: U)(

seqOp: (U, Int) => U,

combOp: (U, U) => U)

要解释太麻烦了。。。总之就是在各个分区内fold一下，再将结果聚合。

例子：

val pa = sc.parallelize(Array(1,2,3,4))def seqOp(x:String,y:Int) = x+y.toStringdef combOp(x:String,y:String) = x+y

pa.aggregate("")(seqOp,combOp)

由于聚合时分区的顺序是不一定的，所以上面代码的执行结果也是不确定的：

scala> pa.aggregate("")(seqOp,combOp)

res201: String = 2413

scala> pa.aggregate("")(seqOp,combOp)

res202: String = 1234

### **takeSample(withReplacement, num, [seed](https://link.jianshu.com/?t=%E5%8F%AF%E9%80%89%E5%8F%82%E6%95%B0%EF%BC%8C%E9%9A%8F%E6%9C%BA%E6%95%B0%E7%A7%8D%E5%AD%90" \t "/Users/liuqing/Documents\\x/_blank))**

随机取num个样本，

* withReplacement: 抽样后是否将元素放回
* num: 抽样个数

val pa = sc.parallelize(Array(1,2,3,4,5,6))

pa.takeSample(false,3)

随机结果：

res185: Array[Int] = Array(3, 2, 5)

### **takeOrdered[T](n:Int)(implicit ord: Ordering[T])**

返回排序后排前n的元素，第二个隐式参数ordering，编译器会自行寻找，也可由用户自定义。

val pa = sc.parallelize(Array(1,2,3,4,5,6))

pa.takeOrdered(3)

结果：

res193: Array[Int] = Array(1, 2, 3)

尝试自定义一个比较器：

object Ord extends Ordering[Int] {

override def compare(x:Int,y:Int):Int = {

if(x<y) 1 else -1;

}}val pa = sc.parallelize(Array(1,2,3,4,5,6))

pa.takeOrdered(3)(Ord)

这次结果变成了：

res195: Array[Int] = Array(6, 5, 4)

### **countByKey()**

对键值对类型RDD有效，统计每个键对应的元素个数。

### **saveAsTextFile(path)**

每个元素作为一行，写入一个文本文件(或一系列文本文件)。由参数path指定写入的目录，支持本地文件系统、HDFS以及其它任何Hadoop支持的文件系统。

### **saveAsSequenceFile(path)**

支持Java和Scala)，将所有元素写入一个 Hadoop SequenceFile， 支持 本地文件系统 、HDFS 和 Hadoop支持的任何文件系统。

只有实现 HadoopWritable 接口的键值对类型的RDD支持此操作。

在Scala里, 可以隐式转换到Writable的类型也支持这个操作， (Spark对基本类型Int, Double, String等都写好了隐式转换)。

### **saveAsObjectFile(path)**

使用Java的序列化格式序列化对象，支持Java 和 Scala，要加载回来的话使用 SparkContext.objectFile()。

5人点赞

[spark](https://www.jianshu.com/nb/16703189" \t "/Users/liuqing/Documents\\x/_blank)