Pyspark RDD

2020 年 11 月 24 日

1 查看 pyspark 的版本号

[1]:

```
print("pyspark version:" + str(sc.version))
```

```
pyspark version:2.4.3
```

2 使用 parallelize 创建 RDD

[2]:

```
x = sc.parallelize([1,2,3])
print(x.collect())
```

```
[1, 2, 3]
```

[3]:

```
x = sc.parallelize(["apple","orange","banana"])
print(x.collect())
```

```
['apple', 'orange', 'banana']
```

3 map

map(f, preservesPartitioning=False)：对 RDD 中每个元素进行 f 函数里面的操作，返回一个新 RDD

preservesPartitioning 表示是否保留父 RDD 的分区信息

[4]:

```
x=sc.parallelize([1,2,3])
y = x.map(lambda x:(x,x**2))
print(x.collect())
print(y.collect())
```

```
[1, 2, 3]
[(1, 1), (2, 4), (3, 9)]
```

4 flatMap

flatMap(f, preservesPartitioning=False)：对 RDD 中每个元素进行 f 函数里面的操作，返回一个扁平化结果的新 RDD

[5]:

```
x = sc.parallelize([1,2,3])
y = x.flatMap(lambda x : (x, 100*x, x**2))
print(x.collect())
print(y.collect())
```

```
[1, 2, 3]
[1, 100, 1, 2, 200, 4, 3, 300, 9]
```

5 mapPartitions

mapPartitions(f, preservesPartitioning=False)：对 RDD 中每个分区里面的全部元素进行自定义 f 函数操作，返回一个新 RDD Section **??**

[6]:

```
x = sc.parallelize([1,2,3],2)
def f(iterator): yield sum(iterator)

y = x.mapPartitions(f)
print(x.glom().collect())
print(y.glom().collect())
```

```
[[1], [2, 3]]
[[1], [5]]
```

6 mapPartitionsWithIndex

mapPartitionsWithIndex(f, preservesPartitioning=False)：对 RDD 中每个分区里面的全部元素进行自定义 f 函数操作，并跟踪每个分区索引

[7]:

```
x = sc.parallelize([1,2,3],2)
def f(partitionIndex, iterator): yield (partitionIndex,sum(iterator))

y = x.mapPartitionsWithIndex(f)
print(x.glom().collect())
print(y.glom().collect())
```

```
[[1], [2, 3]]
[[(0, 1)], [(1, 5)]]
```

7 分区数量

getNumPartitions()：返回分区的数量

[8]:

```
x = sc.parallelize([1,2,3],2)
y = x.getNumPartitions()
print(x.glom().collect())
print(y)
```

```
[[1], [2, 3]]
2
```

8 filter

filter(f)：对 RDD 中的元素进行过滤，返回一个满足过滤条件的新 RDD

[9]:

```
x = sc.parallelize([1,2,3])
y = x.filter(lambda x: x%2==1)
print(x.collect())
print(y.collect())
```

```
[1, 2, 3]
[1, 3]
```

9 distinct

distinct(numPartitions=None)：对 RDD 中的元素进行去重，返回去重后的新 RDD

[10]:

```
x = sc.parallelize(["A","B","A","C"])
y = x.distinct()
print(x.collect())
print(y.collect())
```

```
['A', 'B', 'A', 'C']
['C', 'A', 'B']
```

10 sample

sample(withReplacement, fraction, seed=None)：对 RDD 进行抽样操作。

- withReplacement：是否有放回
- fraction：抽取的比率
- seed：随机生成的种子

[11]:

```
x = sc.parallelize(range(7))
# call 'sample' 5 times
ylist = [x.sample(withReplacement=False, fraction=0.5) for i in range(5)]
print('x = ' + str(x.collect()))
for cnt,y in zip(range(len(ylist)), ylist):
    print('sample:' + str(cnt) + ' y = ' +  str(y.collect()))
```

```
x = [0, 1, 2, 3, 4, 5, 6]
sample:0 y = [1, 3, 4]
sample:1 y = [0, 1, 2, 5, 6]
sample:2 y = [0, 2, 3, 5, 6]
sample:3 y = [0, 1, 3, 4]
sample:4 y = [0, 1, 2, 4]
```

11 takeSample

takeSample(withReplacement, num, seed=None)：对 RDD 中元素进行抽样，返回抽样后 num 个元素。- withReplacement：是否有放回 - seed：随机种子数

[12]:

```
x = sc.parallelize(range(7))
# call 'sample' 5 times
ylist = [x.takeSample(withReplacement=False, num=3) for i in range(5)]
```

```
print('x = ' + str(x.collect()))
for cnt,y in zip(range(len(ylist)), ylist):
    print('sample:' + str(cnt) + ' y = ' +  str(y))
```

```
x = [0, 1, 2, 3, 4, 5, 6]
sample:0 y = [5, 4, 2]
sample:1 y = [5, 3, 2]
sample:2 y = [4, 6, 0]
sample:3 y = [2, 3, 4]
sample:4 y = [0, 5, 4]
```

12 union

union(other)：将自身 RDD 与其它 RDD 进行合并操作，返回一个新的 RDD

[13]:

```
x = sc.parallelize(['A','A','B'])
y = sc.parallelize(['D','C','A'])
z = x.union(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
['A', 'A', 'B']
['D', 'C', 'A']
['A', 'A', 'B', 'D', 'C', 'A']
```

13 intersection

intersection：对自身 RDD 与其它 RDD 取交集

[14]:

```
x = sc.parallelize(['A','A','B'])
y = sc.parallelize(['A','C','D'])
z = x.intersection(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
['A', 'A', 'B']
['A', 'C', 'D']
['A']
```

14 sortByKey

sortByKey(ascending=True, numPartitions=None, keyfunc)：对 RDD 按 key 值或对 key 操作的自定义 keyfunc 函数进行排序，默认为升序，numPartitions：分区的数目。

[15]:

```
x = sc.parallelize([('B',1),('A',2),('C',3)])
y = x.sortByKey()
print(x.collect())
print(y.collect())
```

```
[('B', 1), ('A', 2), ('C', 3)]
[('A', 2), ('B', 1), ('C', 3)]
```

15 sortBy

sortBy(keyfunc, ascending=True, numPartitions=None)：按自定义的 keyfunc 函数对 RDD 中元素进行排序，默认为升序。

[16]:

```
x = sc.parallelize(['Cat','Apple','Bat'])
def keyGen(val): return val[0]

y = x.sortBy(keyGen)
print(y.collect())
```

```
['Apple', 'Bat', 'Cat']
```

16 glom

glom()：创建一个新的 RDD，通过合并每个分区里面的全部元素到一个列表中。

[17]:

```
x = sc.parallelize(['C','B','A'], 2)
y = x.glom()
print(x.collect())
print(y.collect())
```

```
['C', 'B', 'A']
[['C'], ['B', 'A']]
```

17 cartesian

cartesian(other)：将 RDD 与其它 RDD 进行笛卡尔积，返回 <key,value> 类型的 RDD，其中 key 为自身的元素，value 为其它 RDD 的元素。

[18]:
```
x = sc.parallelize(['A','B'])
y = sc.parallelize(['C','D'])
z = x.cartesian(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
['A', 'B']
['C', 'D']
[('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D')]
```

18 groupBy

groupBy(f, numPartitions=None, partitionFunc)：对 RDD 中每个元素按照满足自定义的 f 函数为条件进行分组，返回一个新的 <key,value&gt 类型的 RDD，其中 key 为 f 函数的返回值。

[19]:
```
x = sc.parallelize([1,2,3])
y = x.groupBy(lambda x: 'A' if (x%2 == 1) else 'B' )
print(x.collect())
print([(j[0],[i for i in j[1]]) for j in y.collect()])
```

```
[1, 2, 3]
[('A', [1, 3]), ('B', [2])]
```

19 pipe

pipe(command, env=None, checkCode=False)：对 RDD 元素进行管道操作，将返回 shell 命令的处理结果，形成一个新的 RDD。

[20]:
```python
x = sc.parallelize(['A', 'Ba', 'C', 'AD'])
y = x.pipe('grep -i "A"')
print(x.collect())
print(y.collect())
```

```
['A', 'Ba', 'C', 'AD']
['A', 'Ba', 'AD']
```

20 foreach

foreach(f)：对 RDD 中每个元素进行自定义 f 函数输出操作

[21]:
```python
x = sc.parallelize([1,2,3])
def f(el):
    '''side effect: append the current RDD elements to a file'''
    f1 = open("/home/lei/foreachExample.txt", 'a+')
    print(el,file=f1)
```

[22]:
```python
# first clear the file contents
open('/home/lei/foreachExample.txt', 'w').close()

y = x.foreach(f) # writes into foreachExample.txt
print(x.collect())
print(y) # foreach returns 'None'
# print the contents of foreachExample.txt
with open("/home/lei/foreachExample.txt", "r") as foreachExample:
    print (foreachExample.read())
```

```
[1, 2, 3]
None
1
2
3
```

21 foreachPartition

foreachPartition(f)：对 RDD 每个分区中元素进行自定义 f 函数操作。

[23]:

```
x = sc.parallelize([1,2,3,4,5,6],5)
def f(parition):
    '''side effect: append the current RDD partition contents to a file'''
    f1=open("/home/lei/foreachPartitionExample.txt", 'a+')
    print([el for el in parition],file=f1)
```

[24]:

```
open('/home/lei/foreachPartitionExample.txt', 'w').close()

y = x.foreachPartition(f) # writes into foreachExample.txt

print(x.glom().collect())
print(y)  # foreach returns 'None'
# print the contents of foreachExample.txt
with open("/home/lei/foreachPartitionExample.txt", "r") as foreachExample:
    print (foreachExample.read())
```

```
[[1], [2], [3], [4], [5, 6]]
None
[1]
[2]
[3]
[4]
[5, 6]
```

22 reduce

reduce(f)：使用指定的二元运算符，对 RDD 中每个元素进行 reduce 操作。

[25]:

```
x = sc.parallelize([1,2,3])
y = x.reduce(lambda x, y: x + y)  # computes a cumulative sum
```

```
print(x.collect())
print(y)
```

```
[1, 2, 3]
6
```

23 fold

fold(zeroValue, op)：对 RDD 中每个元素进行聚合操作。使用一个函数和零值，先对每个分区的元素进行聚合，然后对全部分区进行聚合。

[26]:

```
x = sc.parallelize([1,2,3],2)
neutral_zero_value = 0  # 0 for sum, 1 for multiplication
y = x.fold(neutral_zero_value,lambda x, y: x + y) # computes cumulative sum
print(x.glom().collect())
print(y)
```

```
[[1], [2, 3]]
6
```

24 aggregate

aggregate(zeroValue, seqOp, combOp)：使用一个合并函数和一个零值，先对每个分区按合并函数进行聚合，然后将全部分区进行聚合。- seqOp: 每个分区执行的聚合函数, 对 rdd 中按分区每个元素 y 执行此函数, x 为上一次的执行结果, 首次计算时使用默认值 zeroValue

- comOp: 对每个分区的结果执行的聚合函数, 执行此函数时, 每个分区的计算结果 y 执行此函数, x 为上一次的执行结果, 首次计算时使用默认值 zeroValue

[27]:

```
x = sc.parallelize([2,3,4])
neutral_zero_value = (0,1) # sum: x = x+0, product: x = 1*x
seqOp = (lambda x, y: (x[0] + y, x[1] * y))
combOp = (lambda x, y: (x[0] + y[0], x[1] * y[1]))
y = x.aggregate(neutral_zero_value,seqOp,combOp)  # computes (cumulative sum,␣
 ↪cumulative product)
print(x.collect())
print(y)
```

```
[2, 3, 4]
(9, 24)
```

25 max

max(key=None)：找寻 RDD 中最大的一项，参数 key: 一个函数用于生成比较的关键条件

[28]:

```
x = sc.parallelize([1,3,2])
y = x.max()
print(x.collect())
print(y)
```

```
[1, 3, 2]
3
```

26 min

min(key=None)：找寻 RDD 中最小的一项，参数 key: 一个函数用于生成比较的关键条件

[29]:

```

x = sc.parallelize([1,3,2])
y = x.min()
print(x.collect())
print(y)
```

```
[1, 3, 2]
1
```

27 sum

sum()：对 RDD 中所有元素进行累加求和

[30]:

```
x = sc.parallelize([1,3,2])
y = x.sum()
print(x.collect())
print(y)
```

```
[1, 3, 2]
6
```

28 count

count()：计算 RDD 中元素的个数

[31]:

```
x = sc.parallelize([1,3,2])
y = x.count()
print(x.collect())
print(y)
```

```
[1, 3, 2]
3
```

29 histogram

histogram(buckets)：使用提供的桶计算直方图。例如 [1,10,20,50] 意思是桶 [1,10) [10,20) [20,50]

[32]:

```
x = sc.parallelize([1,3,1,2,3])
y = x.histogram(buckets = 2)
print(x.collect())
print(y)
```

```
[1, 3, 1, 2, 3]
([1, 2, 3], [2, 3])
```

[33]:

```
x = sc.parallelize([1,3,1,2,3])
y = x.histogram([0,0.5,1,1.5,2,2.5,3,3.5])
print(x.collect())
print(y)
```

```
[1, 3, 1, 2, 3]
([0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5], [0, 0, 2, 0, 1, 0, 2])
```

30 mean

mean()：计算 RDD 中元素的平均值

[34]:

```
x = sc.parallelize([1,3,2])
y = x.mean()
print(x.collect())
print(y)
```

```
[1, 3, 2]
2.0
```

31 variance

variance()：计算 RDD 中所有元素的方差

[35]:

```
x = sc.parallelize([1,3,2])
y = x.variance()  # divides by N
print(x.collect())
print(y)
```

```
[1, 3, 2]
0.6666666666666666
```

32 stdev

stdev()：计算 RDD 中所有元素的标准差

[36]:

```
x = sc.parallelize([1,3,2])
y = x.stdev()  # divides by N
print(x.collect())
print(y)
```

```
[1, 3, 2]
0.816496580927726
```

33 sampleStdev

sampleStdev()：计算 RDD 中所有元素的样本标准差

[37]:

```
x = sc.parallelize([1,3,2])
y = x.sampleStdev() # divides by N-1
print(x.collect())
print(y)
```

```
[1, 3, 2]
1.0
```

34 sampleVariance

sampleVariance()：计算 RDD 中所有元素的样本方差

[38]:

```
x = sc.parallelize([1,3,2])
y = x.sampleVariance()  # divides by N-1
print(x.collect())
print(y)
```

```
[1, 3, 2]
1.0
```

35 countByValue

countByValue()：对 RDD 中每个元素进行计数

[39]:

```
x = sc.parallelize(["A","C","A","B","C"])
y = x.countByValue()
print(x.collect())
print(y)
```

```
['A', 'C', 'A', 'B', 'C']
defaultdict(<class 'int'>, {'A': 2, 'C': 2, 'B': 1})
```

36 top

top(num, key=None)：对 RDD 中元素按降序或自定义 key 函数进行排序，输出排序后的前 num 个元素

[40]:

```
x = sc.parallelize([1,3,1,2,3])
y = x.top(num = 3)
print(x.collect())
print(y)
```

```
[1, 3, 1, 2, 3]
[3, 3, 2]
```

37 takeOrdered

takeOrdered(num, key=None)：对 RDD 中元素按升序或自定义 key 函数进行排序，输出排序后的前 num 个元素。

[41]:

```
x = sc.parallelize([1,3,1,2,3])
y = x.takeOrdered(num = 3)
print(x.collect())
print(y)
```

```
[1, 3, 1, 2, 3]
[1, 1, 2]
```

38 take

take(num)：输出 RDD 中前 num 个元素

[42]:

```
x = sc.parallelize([1,3,1,2,3])
y = x.take(num = 3)
print(x.collect())
print(y)
```

```
[1, 3, 1, 2, 3]
[1, 3, 1]
```

39 first

first()：输出 RDD 中第一个元素

[43]:

```
x = sc.parallelize([1,3,1,2,3])
y = x.first()
print(x.collect())
print(y)
```

```
[1, 3, 1, 2, 3]
1
```

40 collectAsMap

collectAsMap()：对 RDD 的每个元素进行遍历，返回一个键值对类型的字典。

[44]:

```
x = sc.parallelize([('C',3),('A',1),('B',2)])
y = x.collectAsMap()
print(x.collect())
print(y)
```

```
[('C', 3), ('A', 1), ('B', 2)]
{'C': 3, 'A': 1, 'B': 2}
```

41 keys

keys()：对 <key,value> 类型 RDD 进行操作，返回 RDD 每个元素的 key 值。

[45]:

```
x = sc.parallelize([('C',3),('A',1),('B',2)])
y = x.keys()
print(x.collect())
print(y.collect())
```

```
[('C', 3), ('A', 1), ('B', 2)]
['C', 'A', 'B']
```

42 values

values()：对 <key,value> 类型的 RDD 进行操作，返回 RDD 每个元素的 value 值。

[46]:

```
x = sc.parallelize([('C',3),('A',1),('B',2)])
y = x.values()
print(x.collect())
print(y.collect())
```

```
[('C', 3), ('A', 1), ('B', 2)]
[3, 1, 2]
```

43 reduceByKey

reduceByKey(func, numPartitions=None, partitionFunc)：对 pairRDD 中的 key 先进行 group by 操作，然后对聚合后的 value 数据进行自定义 f 函数操作，返回一个新的 RDD

[47]:

```
x = sc.parallelize([('B',1),('B',2),('A',3),('A',4),('A',5)])
y = x.reduceByKey(lambda agg, obj: agg + obj)
print(x.collect())
print(y.collect())
```

```
[('B', 1), ('B', 2), ('A', 3), ('A', 4), ('A', 5)]
[('B', 3), ('A', 12)]
```

44 countByKey

countByKey()：对 key 相同的所有元素进行计数，返回值为一个字典

[48]:

```
x = sc.parallelize([('B',1),('B',2),('A',3),('A',4),('A',5)])
y = x.countByKey()
print(x.collect())
print(y)
```

```
[('B', 1), ('B', 2), ('A', 3), ('A', 4), ('A', 5)]
defaultdict(<class 'int'>, {'B': 2, 'A': 3})
```

45 join

join(other, numPartitions=None)：对 RDD 上的每个元素与其它 RDD 进行 join 操作，返回一个 (k, (v1, v2)) 类型的新 RDD，其中 (k, v1) 在自身 RDD，(k, v2) 在其它 RDD。

[49]:

```
x = sc.parallelize([('C',4),('B',3),('A',2),('A',1)])
y = sc.parallelize([('A',8),('B',7),('A',6),('D',5)])
z = x.join(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
[('C', 4), ('B', 3), ('A', 2), ('A', 1)]
[('A', 8), ('B', 7), ('A', 6), ('D', 5)]
[('B', (3, 7)), ('A', (2, 8)), ('A', (2, 6)), ('A', (1, 8)), ('A', (1, 6))]
```

46 leftOuterJoin

leftOuterJoin(other, numPartitions=None)：执行自身 RDD 与其他 RDD 的 left outer join 操作，例如自身 RDD 每个元素为 <k,v>，其他 RDD 每个元素为 <k,w>，返回新的 RDD 中包含全部的 pairs(k, (v, w)) 或者 pair(k, (v, None))。numPartitions：进行 Hash 分区的数量

[50]:

```
x = sc.parallelize([('C',4),('B',3),('A',2),('A',1)])
y = sc.parallelize([('A',8),('B',7),('A',6),('D',5)])
z = x.leftOuterJoin(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
[('C', 4), ('B', 3), ('A', 2), ('A', 1)]
[('A', 8), ('B', 7), ('A', 6), ('D', 5)]
[('B', (3, 7)), ('A', (2, 8)), ('A', (2, 6)), ('A', (1, 8)), ('A', (1, 6)),
('C', (4, None))]
```

47 rightOuterJoin

rightOuterJoin：执行自身 RDD 与其他 RDD 的 right outer join 操作，例如自身 RDD 每个元素为 <k,v>，其他 RDD 每个元素为 <k,w>，返回新的 RDD 中包含全部的 pairs(k, (v, w)) 或者 pair(k, (None, w))。numPartitions：进行 Hash 分区的数量

[51]:

```
x = sc.parallelize([('C',4),('B',3),('A',2),('A',1)])
y = sc.parallelize([('A',8),('B',7),('A',6),('D',5)])
```

```
z = x.rightOuterJoin(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
[('C', 4), ('B', 3), ('A', 2), ('A', 1)]
[('A', 8), ('B', 7), ('A', 6), ('D', 5)]
[('B', (3, 7)), ('A', (2, 8)), ('A', (2, 6)), ('A', (1, 8)), ('A', (1, 6)),
('D', (None, 5))]
```

48 partitionBy

partitionBy(numPartitions, partitionFunc)：对 RDD 进行分区。numPartitions：分区的数目，partitionFunc：自定义分区函数，partitionFunc(k) % numPartitions 的值为新分区的索引 index

[52]:

```
# partitionBy Example1
x = sc.parallelize([(0,1),(1,2),(2,3)],2)
y = x.partitionBy(numPartitions = 3, partitionFunc = lambda x: x)  # only key␣
 ↪is passed to paritionFunc
print(x.glom().collect())
print(y.glom().collect())
```

```
[[(0, 1)], [(1, 2), (2, 3)]]
[[(0, 1)], [(1, 2)], [(2, 3)]]
```

[53]:

```
# partitionBy Example2
x = sc.parallelize([("hadoop",1),("spark",2),("python",3),("C",4)],2)
y = x.partitionBy(numPartitions = 3, partitionFunc = lambda x: len(x))  # only␣
 ↪key is passed to paritionFunc
print(x.glom().collect())
print(y.glom().collect())
```

```
[[('hadoop', 1), ('spark', 2)], [('python', 3), ('C', 4)]]
[[('hadoop', 1), ('python', 3)], [('C', 4)], [('spark', 2)]]
```

49 combineByKey

combineByKey(createCombiner, mergeValue, mergeCombiners, numPartitions=None, partitionFunc)：泛型函数使用一个自定义的聚合函数，去合并 RDD 中每个 key 相同的元素，具体为转换 RDD[(K, V)] 形成一个新的 RDD[(K, C)]，其中 C 是一个合并类型。createCombiner：创建一个 V 到 C 的函数，mergeValue：将一个 V 形成一个 C，mergeCombiners：将 C 的集合进行合并。

[54]:

```
x = sc.parallelize([('B',1),('B',2),('A',3),('A',4),('A',5)])
createCombiner = (lambda el: [(el,el**2)])
mergeVal = (lambda aggregated, el: aggregated + [(el,el**2)]) # append to␣
 ↪aggregated
mergeComb = (lambda agg1,agg2: agg1 + agg2 )  # append agg1 with agg2
y = x.combineByKey(createCombiner,mergeVal,mergeComb)
print(x.collect())
print(y.collect())
```

```
[('B', 1), ('B', 2), ('A', 3), ('A', 4), ('A', 5)]
[('B', [(1, 1), (2, 4)]), ('A', [(3, 9), (4, 16), (5, 25)])]
```

50 aggregateByKey

aggregateByKey(zeroValue, seqFunc, combFunc, numPartitions=None, partitionFunc)：聚合每个键的值, 使用组合函数和一个零值，函数返回一个不同类型的 rdd。seqFunc：是对一个分区里每个键的值聚合，combFunc：是对分区间每个键的聚合结果进行聚合。

[55]:

```
x = sc.parallelize([('B',1),('B',2),('A',3),('A',4),('A',5)])
zeroValue = [] # empty list is 'zero value' for append operation
mergeVal = (lambda aggregated, el: aggregated + [(el,el**2)])
mergeComb = (lambda agg1,agg2: agg1 + agg2 )
y = x.aggregateByKey(zeroValue,mergeVal,mergeComb)
print(x.collect())
print(y.collect())
```

```
[('B', 1), ('B', 2), ('A', 3), ('A', 4), ('A', 5)]
[('B', [(1, 1), (2, 4)]), ('A', [(3, 9), (4, 16), (5, 25)])]
```

51 foldByKey

foldByKey(zeroValue, func, numPartitions=None, partitionFunc)：使用一个组合函数 func 与一个零值，对 key 相同的 value 值进行聚合。

[56]:
```
x = sc.parallelize([('B',1),('B',2),('A',3),('A',4),('A',5)])
zeroValue = 1 # one is 'zero value' for multiplication
y = x.foldByKey(zeroValue,lambda agg,x: agg*x )  # computes cumulative product␣
 ↪within each key
print(x.collect())
print(y.collect())
```

```
[('B', 1), ('B', 2), ('A', 3), ('A', 4), ('A', 5)]
[('B', 2), ('A', 60)]
```

52 groupByKey

groupByKey(numPartitions=None, partitionFunc)：对 RDD 里 key 相同的元素进行分组，分组结果形成一个序列，最后返回一个新的 <key,value> 类型的 RDD。numPartitions：进行 Hash 分区的分区数

[57]:
```
x = sc.parallelize([('B',5),('B',4),('A',3),('A',2),('A',1)])
y = x.groupByKey()
print(x.collect())
print([(j[0],[i for i in j[1]]) for j in y.collect()])
```

```
[('B', 5), ('B', 4), ('A', 3), ('A', 2), ('A', 1)]
[('B', [5, 4]), ('A', [3, 2, 1])]
```

53 flatMapValues

flatMapValues(f)：使用一个 flatMap 函数，对类型 RDD 中 key 相同的 value 值进行操作，返回一个新的 RDD。新 RDD 的 key 值不变，只改变了 value 值，还保留了原始 RDD 的分区。

[58]:
```
x = sc.parallelize([('A',(1,2,3)),('B',(4,5))])
y = x.flatMapValues(lambda x: [i**2 for i in x]) # function is applied to␣
 ↪entire value, then result is flattened
print(x.collect())
print(y.collect())
```

```
[('A', (1, 2, 3)), ('B', (4, 5))]
[('A', 1), ('A', 4), ('A', 9), ('B', 16), ('B', 25)]
```

54 mapValues

mapValues：对键值对 <key,value> 中的 value 部分执行函数里面的操作，返回 <key,value> 键值对形式的新 RDD

[59]:

```
x = sc.parallelize([('A',(1,2,3)),('B',(4,5))])
y = x.mapValues(lambda x: [i**2 for i in x]) # function is applied to entire␣
 ↪value
print(x.collect())
print(y.collect())
```

```
[('A', (1, 2, 3)), ('B', (4, 5))]
[('A', [1, 4, 9]), ('B', [16, 25])]
```

55 cogroup

cogroup(other)：对两个 RDD 数据集按 key 相同的数据进行 group by，并对 key 值相同的数据中每个 RDD 的 value 进行单独 group by

[60]:

```
x = sc.parallelize([('C',4),('B',(3,3)),('A',2),('A',(1,1))])
y = sc.parallelize([('A',8),('B',7),('A',6),('D',(5,5))])
z = x.cogroup(y)
print(x.collect())
print(y.collect())
for key,val in list(z.collect()):
    print(key, [list(i) for i in val])
```

```
[('C', 4), ('B', (3, 3)), ('A', 2), ('A', (1, 1))]
[('A', 8), ('B', 7), ('A', 6), ('D', (5, 5))]
B [[(3, 3)], [7]]
A [[2, (1, 1)], [8, 6]]
C [[4], []]
```

```
D [[], [(5, 5)]]
```

56 groupWith

groupWith(other, *others)：类似于 cogroup 操作，但支持多个 RDD。返回类型为 RDD。

[61]:

```
x = sc.parallelize([('C',4),('B',(3,3)),('A',2),('A',(1,1))])
y = sc.parallelize([('B',(7,7)),('A',6),('D',(5,5))])
z = sc.parallelize([('D',9),('B',(8,8))])
a = x.groupWith(y,z)
print(x.collect())
print(y.collect())
print(z.collect())
print("Result:")
for key,val in list(a.collect()):
    print(key, [list(i) for i in val])
```

```
[('C', 4), ('B', (3, 3)), ('A', 2), ('A', (1, 1))]
[('B', (7, 7)), ('A', 6), ('D', (5, 5))]
[('D', 9), ('B', (8, 8))]
Result:
C [[4], [], []]
A [[2, (1, 1)], [6], []]
B [[(3, 3)], [(7, 7)], [(8, 8)]]
D [[], [(5, 5)], [9]]
```

57 sampleByKey

sampleByKey(withReplacement, fractions, seed=None)：以 key 值对元素进行抽样，返回一个新 RDD，withReplacement：表示是否有放回，True 表示有放回，fractions：key 值得抽样率，seed：随机种子

[62]:

```
x = sc.parallelize([('A',1),('B',2),('C',3),('B',4),('A',5)])
y = x.sampleByKey(withReplacement=False, fractions={'A':0.5, 'B':1, 'C':0.2})
print(x.collect())
print(y.collect())
```

```
[('A', 1), ('B', 2), ('C', 3), ('B', 4), ('A', 5)]
[('A', 1), ('B', 2), ('B', 4), ('A', 5)]
```

58 subtractByKey

subtractByKey(other, numPartitions=None)：按 key 值对 RDD 进行扣除操作，返回自身 <key,value> 类型 RDD 不匹配其他 RDD 中 key 的部分

[63]:

```
x = sc.parallelize([('C',1),('B',2),('A',3),('A',4)])
y = sc.parallelize([('A',5),('D',6),('A',7),('D',8)])
z = x.subtractByKey(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
[('C', 1), ('B', 2), ('A', 3), ('A', 4)]
[('A', 5), ('D', 6), ('A', 7), ('D', 8)]
[('B', 2), ('C', 1)]
```

59 subtract

subtract(other, numPartitions=None)：返回自身 RDD 中不匹配其他 RDD 的部分

[64]:

```
x = sc.parallelize([('C',4),('B',3),('A',2),('A',1)])
y = sc.parallelize([('C',8),('A',2),('D',1)])
z = x.subtract(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
[('C', 4), ('B', 3), ('A', 2), ('A', 1)]
[('C', 8), ('A', 2), ('D', 1)]
[('C', 4), ('A', 1), ('B', 3)]
```

60 keyBy

keyBy(f)：使用自定义函数 f，创建每个元素为元组类型的新 RDD，f 函数的返回值作为 key，RDD 的元素值作为 value。

[65]:

```python
x = sc.parallelize([1,2,3])
y = x.keyBy(lambda x: x**2)
print(x.collect())
print(y.collect())
```

```
[1, 2, 3]
[(1, 1), (4, 2), (9, 3)]
```

61 repartition

repartition(numPartitions)：对 RDD 进行分区。numPartitions：为分区数

[66]:

```python
x = sc.parallelize([1,2,3,4,5],2)
y = x.repartition(numPartitions=3)
print(x.glom().collect())
print(y.glom().collect())
```

```
[[1, 2], [3, 4, 5]]
[[], [1, 2], [3, 4, 5]]
```

62 coalesce

coalesce(numPartitions, shuffle=False)：对 RDD 进行分区，将分区数减少到 numPartitions。

[67]:

```python
x = sc.parallelize([1,2,3,4,5],2)
y = x.coalesce(numPartitions=1)
print(x.glom().collect())
print(y.glom().collect())
```

```
[[1, 2], [3, 4, 5]]
[[1, 2, 3, 4, 5]]
```

63 zip

zip(other)：将 RDD 与其它 RDD 进行 zip 操作，返回 <key,value> 类型的新 RDD，其中 key 为自身 RDD 的元素值，value 为其它 RDD 的元素值

[68]:
```
x = sc.parallelize(['B','A','A'])
# zip expects x and y to have same #partitions and #elements/partition
y = x.map(lambda x: ord(x))
z = x.zip(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
['B', 'A', 'A']
[66, 65, 65]
[('B', 66), ('A', 65), ('A', 65)]
```

64 zipWithIndex

zipWithIndex()：对 RDD 进行 zip 操作，返回新的 RDD 中，每个元素包含原 RDD 的元素值还有对应的索引。

[69]:
```
x = sc.parallelize(['B','A','A'],2)
y = x.zipWithIndex()
print(x.glom().collect())
print(y.collect())
```

```
[['B'], ['A', 'A']]
[('B', 0), ('A', 1), ('A', 2)]
```

65 zipWithUniqueId

zipWithUniqueId()：对 RDD 进行 zip 操作，返回 <key,value> 类型新 RDD，key 为原 RDD 的元素值，value 为从 0 开始的值得 id

[70]:
```
x = sc.parallelize(['B','A','A'],2)
y = x.zipWithUniqueId()
print(x.glom().collect())
print(y.collect())
```

```
[['B'], ['A', 'A']]
[('B', 0), ('A', 1), ('A', 3)]
```

66 WordCount

[71]:

```
textFile = sc.textFile("hdfs://localhost:9000/input/wordcount/testfile")
```

[72]:

```
stringRDD = textFile.flatMap(lambda line:line.split(" "))
```

[74]:

```
countsRDD = stringRDD.map(lambda word:(word,1)).reduceByKey(lambda x,y:x+y)
```

[75]:

```
countsRDD.collect()
```

[75]:

```
[('world', 1), ('hdfs', 1), ('hello', 3), ('big', 1), ('data', 1)]
```

问题：解释上面 WordCount 中每一条语句实现的功能。