pyspark-RDD API - 风吴痕的博客 - CSDN博客

blog.csdn.net/wc781708249/article/details/78228117

2、上传文件

vim person.json

输入：

```
1. {"name":"Michael"}
2. {"name":"Andy", "age":30}
3. {"name":"Justin", "age":19}
```

如果安装了hadoop，(参考集群版安装)
执行：

hdfs dfs -mkdir /data

hdfs dfs -put person.json /data

读取hdfs上的数据

```
df = spark.read.json("hdfs://localhost:9000/data/person.json")
```

补充：数据读取

```
1. sc.pickleFile() # <class 'pyspark.rdd.RDD'>
2. sc.textFile() # <class 'pyspark.rdd.RDD'>
3. spark.read.json() # <class 'pyspark.sql.dataframe.DataFrame'>
4. spark.read.text() # <class 'pyspark.sql.dataframe.DataFrame'>
```

读取本地数据
sc.pickleFile("file:///home/mparsian/dna_seq.txt")

sc.pickleFile("home/mparsian/dna_seq.txt")

3、RDD API

map

```
# map
x = sc.parallelize([1,2,3]) # sc = spark context, parallelize creates an RDD from the passed object
y = x.map(lambda x: (x,x**2))
print(x.collect())  # collect copies RDD elements to a list on the driver
print(y.collect())

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

`map`(*f*, *preservesPartitioning=False*)
Return a new RDD by applying a function to each element of this RDD.

```
>>> rdd = sc.parallelize(["b", "a", "c"])
>>> sorted(rdd.map(lambda x: (x, 1)).collect())
[('a', 1), ('b', 1), ('c', 1)]
```

flatmap

```
# flatMap
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]
```

`flatMap`(*f, preservesPartitioning=False*)
Return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

```
>>> rdd = sc.parallelize([2, 3, 4])
>>> sorted(rdd.flatMap(lambda x: range(1, x)).collect())
[1, 1, 1, 2, 2, 3]
>>> sorted(rdd.flatMap(lambda x: [(x, x), (x, x)]).collect())
[(2, 2), (2, 2), (3, 3), (3, 3), (4, 4), (4, 4)]
```

mapPartitions

```
# mapPartitions
x = sc.parallelize([1,2,3], 2)
def f(iterator): yield sum(iterator)
y = x.mapPartitions(f)
print(x.glom().collect())  # glom() flattens elements on the same partition
print(y.glom().collect())
[[1], [2, 3]]
[[1], [5]]
```

`mapPartitions`(*f, preservesPartitioning=False*)
Return a new RDD by applying a function to each partition of this RDD.

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)
>>> def f(iterator): yield sum(iterator)
>>> rdd.mapPartitions(f).collect()
[3, 7]
```

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2) # [[1,2],[3,4]]
>>> def f(iterator): yield sum(iterator)
...
>>> rdd.mapPartitions(f).collect()
[3, 7]
>>> rdd.collect()
[1, 2, 3, 4]
>>> rdd = sc.parallelize([1, 2, 3, 4], 4) # [[1],[2],[3],[4]]
>>> rdd.mapPartitions(f).collect()
[1, 2, 3, 4]
```

mapPartitionsWithIndex

```
# mapPartitionsWithIndex
x = sc.parallelize([1,2,3], 2)
def f(partitionIndex, iterator): yield (partitionIndex,sum(iterator))
y = x.mapPartitionsWithIndex(f)
print(x.glom().collect())  # glom() flattens elements on the same partition
print(y.glom().collect())
[[1], [2, 3]]
[[(0, 1)], [(1, 5)]]
```

`mapPartitionsWithIndex`(*f, preservesPartitioning=False*)

Return a new RDD by applying a function to each partition of this RDD, while tracking the index of the original partition.

>>> rdd=sc.parallelize([1,2,3,4],4)>>>deff(splitIndex,iterator):yieldsplitIndex>>>rdd.mapPartitionsWithIndex(f).su

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 4)
>>> def f(splitIndex, iterator): yield splitIndex
...
>>> rdd.mapPartitionsWithIndex(f).sum()
6
>>> rdd.collect()
[1, 2, 3, 4]
>>>
```

getNumPartitions

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

`getNumPartitions`**()**
Returns the number of partitions in RDD

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)
>>> rdd.getNumPartitions()
2
```

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)
>>> rdd.getNumPartitions()
2
>>> rdd = sc.parallelize([1, 2, 3, 4], 3)
>>> rdd.getNumPartitions()
3
>>> rdd = sc.parallelize([1, 2, 3, 4], 4)
>>> rdd.getNumPartitions()
4
```

filter

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

`filter`**(*f*)**
Return a new RDD containing only the elements that satisfy a predicate.

```
>>> rdd = sc.parallelize([1, 2, 3, 4, 5])
>>> rdd.filter(lambda x: x % 2 == 0).collect()
[2, 4]
```

distinct

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

`distinct`(*numPartitions=None*)
Return a new RDD containing the distinct elements in this RDD.

```
>>> sorted(sc.parallelize([1, 1, 2, 3]).distinct().collect())
[1, 2, 3]
```

sample

```
# sample
x = sc.parallelize(range(7))
ylist = [x.sample(withReplacement=False, fraction=0.5) for i in range(5)] # call 'sample' 5 times
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 = [0, 6]
sample:1 y = [4]
sample:2 y = [1, 2, 3]
sample:3 y = [2, 3, 5, 6]
sample:4 y = [1, 2]
```

`sample`(*withReplacement*, *fraction*, *seed=None*)¶
Return a sampled subset of this RDD (relies on numpy and falls back on default random generator if numpy is unavailable).

takeSample

```
# takeSample
x = sc.parallelize(range(7))
ylist = [x.takeSample(withReplacement=False, num=3) for i in range(5)]  # call 'sample' 5 times
print('x = ' + str(x.collect()))
for cnt,y in zip(range(len(ylist)), ylist):
    print('sample:' + str(cnt) + ' y = ' +  str(y))  # no collect on y
x = [0, 1, 2, 3, 4, 5, 6]
sample:0 y = [5, 4, 3]
sample:1 y = [4, 0, 2]
sample:2 y = [1, 2, 4]
sample:3 y = [5, 6, 0]
sample:4 y = [3, 1, 6]
```

`takeSample`(*withReplacement*, *num*, *seed=None*)
Return a fixed-size sampled subset of this RDD (currently requires numpy).

```
>>> rdd = sc.parallelize(range(0, 10))
>>> len(rdd.takeSample(True, 20, 1))
20
>>> len(rdd.takeSample(False, 5, 2))
5
>>> len(rdd.takeSample(False, 15, 3))
10
```

union

```
# union
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']
```

`union`(*other*)
Return the union of this RDD and another one.

```
>>> rdd = sc.parallelize([1, 1, 2, 3])
>>> rdd.union(rdd).collect()
[1, 1, 2, 3, 1, 1, 2, 3]
```

intersection

```
# intersection
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']
```

`intersection`(*other*)
Return the intersection of this RDD and another one. The output will not contain any duplicate elements, even if the input RDDs did.

Note that this method performs a shuffle internally.

```
>>> rdd1 = sc.parallelize([1, 10, 2, 3, 4, 5])
>>> rdd2 = sc.parallelize([1, 6, 2, 3, 7, 8])
>>> rdd1.intersection(rdd2).collect()
[1, 2, 3]
```

sortByKey

```
# sortByKey
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)]
```

`sortByKey`(*ascending=True*, *numPartitions=None*, *keyfunc=<function <lambda> at 0x3d10398>*)
Sorts this RDD, which is assumed to consist of (key, value) pairs. # noqa

```
>>> tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
>>> sc.parallelize(tmp).sortByKey().first()
('1', 3)
>>> sc.parallelize(tmp).sortByKey(True, 1).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
>>> sc.parallelize(tmp).sortByKey(True, 2).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
>>> tmp2 = [('Mary', 1), ('had', 2), ('a', 3), ('little', 4), ('lamb', 5)]
>>> tmp2.extend([('whose', 6), ('fleece', 7), ('was', 8), ('white', 9)])
>>> sc.parallelize(tmp2).sortByKey(True, 3, keyfunc=lambda k: k.lower()).collect()
[('a', 3), ('fleece', 7), ('had', 2), ('lamb', 5),...('white', 9), ('whose', 6)]
```

sortBy

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

`sortBy`(*keyfunc*, *ascending=True*, *numPartitions=None*)
Sorts this RDD by the given keyfunc

```
>>> tmp = [('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
>>> sc.parallelize(tmp).sortBy(lambda x: x[0]).collect()
[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]
>>> sc.parallelize(tmp).sortBy(lambda x: x[1]).collect()
[('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
```

glom

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

`glom`**()**
Return an RDD created by coalescing all elements within each partition into a list.

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)
>>> sorted(rdd.glom().collect())
[[1, 2], [3, 4]]
```

```
>>> rdd = sc.parallelize([1, 2, 3, 4], 2)

>>> sorted(rdd.glom().collect())

[[1, 2], [3, 4]]

>>> rdd = sc.parallelize([1, 2, 3, 4], 3)

>>> sorted(rdd.glom().collect())

[[1], [2], [3, 4]]

>>> rdd = sc.parallelize([1, 2, 3, 4], 4)

>>> sorted(rdd.glom().collect())

[[1], [2], [3], [4]]

>>>
```

cartesian

```
# cartesian
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')]
```

`cartesian`**(*other*)**
Return the Cartesian product of this RDD and another one, that is, the RDD of all pairs of elements *(a, b)* where `a` is in `self` and `b` is in `other`.

```
>>> rdd = sc.parallelize([1, 2])
>>> sorted(rdd.cartesian(rdd).collect())
[(1, 1), (1, 2), (2, 1), (2, 2)]
```

groupBy

```
# groupBy
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()]) # y is nested, this iterates through it
[1, 2, 3]
[('A', [1, 3]), ('B', [2])]
```

`groupBy`**(*f*, *numPartitions=None*)**
Return an RDD of grouped items.

```
>>> rdd = sc.parallelize([1, 1, 2, 3, 5, 8])
>>> result = rdd.groupBy(lambda x: x % 2).collect()
>>> sorted([(x, sorted(y)) for (x, y) in result])
[(0, [2, 8]), (1, [1, 1, 3, 5])]
```

pipe

```
# pipe
x = sc.parallelize(['A', 'Ba', 'C', 'AD'])
y = x.pipe('grep -i "A"') # calls out to grep, may fail under Windows
print(x.collect())
print(y.collect())
['A', 'Ba', 'C', 'AD']
[u'A', u'Ba', u'AD']
```

`pipe`(*command*, *env={}*)
Return an RDD created by piping elements to a forked external process.

```
>>> sc.parallelize(['1', '2', '', '3']).pipe('cat').collect()
['1', '2', '', '3']
```

foreach

```
# foreach
from __future__ import print_function
x = sc.parallelize([1,2,3])
def f(el):
    '''side effect: append the current RDD elements to a file'''
    f1=open("./foreachExample.txt", 'a+')
    print(el,file=f1)

open('./foreachExample.txt', 'w').close()  # first clear the file contents

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

print(x.collect())
print(y) # foreach returns 'None'
# print the contents of foreachExample.txt
with open("./foreachExample.txt", "r") as foreachExample:
    print (foreachExample.read())
[1, 2, 3]
None
1
3
2
```

`foreach`(*f*)
Applies a function to all elements of this RDD.

```
>>> def f(x): print x
>>> sc.parallelize([1, 2, 3, 4, 5]).foreach(f)
```

foreachPartition

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

open('./foreachPartitionExample.txt', 'w').close()  # first clear the file contents

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("./foreachPartitionExample.txt", "r") as foreachExample:
    print (foreachExample.read())
[[], [1], [], [2], [3]]
None
[]
[1]
[]
[2]
[3]
```

`foreachPartition`(*f*)
Applies a function to each partition of this RDD.

```
>>> def f(iterator):
...      for x in iterator:
...           print x
>>> sc.parallelize([1, 2, 3, 4, 5]).foreachPartition(f)
```

collect

```
# collect
x = sc.parallelize([1,2,3])
y = x.collect()
print(x)  # distributed
print(y)  # not distributed
ParallelCollectionRDD[84] at parallelize at PythonRDD.scala:423
[1, 2, 3]
```

`collect`**()**¶
Return a list that contains all of the elements in this RDD.

reduce

```
# reduce
x = sc.parallelize([1,2,3])
y = x.reduce(lambda obj, accumulated: obj + accumulated)  # computes a cumulative sum
print(x.collect())
print(y)
[1, 2, 3]
6
```

`reduce`**(*f*)**
Reduces the elements of this RDD using the specified commutative and associative binary operator. Currently reduces partitions locally.

```
>>> from operator import add
>>> sc.parallelize([1, 2, 3, 4, 5]).reduce(add)
15
>>> sc.parallelize((2 for _ in range(10))).map(lambda x: 1).cache().reduce(add)
10
>>> sc.parallelize([]).reduce(add)
Traceback (most recent call last):
    ...
ValueError: Can not reduce() empty RDD
```

fold

```
# fold
x = sc.parallelize([1,2,3])
neutral_zero_value = 0  # 0 for sum, 1 for multiplication
y = x.fold(neutral_zero_value,lambda obj, accumulated: accumulated + obj) # computes cumulative sum
print(x.collect())
print(y)
[1, 2, 3]
6
```

`fold`**(*zeroValue*, *op*)**
Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function and a neutral "zero value."

The function *op(t1, t2)* is allowed to modify `t1` and return it as its result value to avoid object allocation; however, it should not modify `t2`.

```
>>> from operator import add
>>> sc.parallelize([1, 2, 3, 4, 5]).fold(0, add)
15
```

aggregate

```
# aggregate
x = sc.parallelize([2,3,4])
neutral_zero_value = (0,1) # sum: x+0 = x, product: 1*x = x
seqOp = (lambda aggregated, el: (aggregated[0] + el, aggregated[1] * el))
combOp = (lambda aggregated, el: (aggregated[0] + el[0], aggregated[1] * el[1]))
y = x.aggregate(neutral_zero_value,seqOp,combOp)  # computes (cumulative sum, cumulative product)
print(x.collect())
print(y)
[2, 3, 4]
(9, 24)
```

`aggregate`**(*zeroValue*, *seqOp*, *combOp*)**
Aggregate the elements of each partition, and then the results for all the partitions, using a given combine functions and a neutral "zero value."

The functions *op(t1, t2)* is allowed to modify `t1` and return it as its result value to avoid object allocation; however, it should not modify `t2`.

The first function (seqOp) can return a different result type, U, than the type of this RDD. Thus, we need one operation for merging a T into an U and one operation for merging two U

```
>>> seqOp = (lambda x, y: (x[0] + y, x[1] + 1))
>>> combOp = (lambda x, y: (x[0] + y[0], x[1] + y[1]))
>>> sc.parallelize([1, 2, 3, 4]).aggregate((0, 0), seqOp, combOp)
(10, 4)
>>> sc.parallelize([]).aggregate((0, 0), seqOp, combOp)
(0, 0)
```

max

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

`max`(*key=None*)
Find the maximum item in this RDD.

Parameters: **key** – A function used to generate key for comparing

```
>>> rdd = sc.parallelize([1.0, 5.0, 43.0, 10.0])
>>> rdd.max()
43.0
>>> rdd.max(key=str)
5.0
```

min

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

`min`(*key=None*)
Find the minimum item in this RDD.

Parameters: **key** – A function used to generate key for comparing

```
>>> rdd = sc.parallelize([2.0, 5.0, 43.0, 10.0])
>>> rdd.min()
2.0
>>> rdd.min(key=str)
10.0
```

sum

```
# sum
x = sc.parallelize([1,3,2])
y = x.sum()
print(x.collect())
print(y)
[1, 3, 2]
6
```

`sum`()
Add up the elements in this RDD.

```
>>> sc.parallelize([1.0, 2.0, 3.0]).sum()
6.0
```

count

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

`count`()
Return the number of elements in this RDD.

>>> sc.parallelize([2,3,4]).count()3

histogram

```
# histogram (example #1)
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])
```

```
# histogram (example #2)
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])
```

`histogram`(*buckets*)
Compute a histogram using the provided buckets. The buckets are all open to the right except for the last which is closed. e.g. [1,10,20,50] means the buckets are [1,10) [10,20) [20,50], which means 1<=x<10, 10<=x<20, 20<=x<=50. And on the input of 1 and 50 we would have a histogram of 1,0,1.

If your histogram is evenly spaced (e.g. [0, 10, 20, 30]), this can be switched from an O(log n) inseration to O(1) per element(where n = # buckets).

Buckets must be sorted and not contain any duplicates, must be at least two elements.

If *buckets* is a number, it will generates buckets which are evenly spaced between the minimum and maximum of the RDD. For example, if the min value is 0 and the max is 100, given buckets as 2, the resulting buckets will be [0,50) [50,100]. buckets must be at least 1 If the RDD contains infinity, NaN throws an exception If the elements in RDD do not vary (max == min) always returns a single bucket.

It will return an tuple of buckets and histogram.

```
>>> rdd = sc.parallelize(range(51))
>>> rdd.histogram(2)
([0, 25, 50], [25, 26])
>>> rdd.histogram([0, 5, 25, 50])
([0, 5, 25, 50], [5, 20, 26])
>>> rdd.histogram([0, 15, 30, 45, 60])  # evenly spaced buckets
([0, 15, 30, 45, 60], [15, 15, 15, 6])
>>> rdd = sc.parallelize(["ab", "ac", "b", "bd", "ef"])
>>> rdd.histogram(("a", "b", "c"))
(('a', 'b', 'c'), [2, 2])
```

mean

```
# mean
x = sc.parallelize([1,3,2])
y = x.mean()
print(x.collect())
print(y)
[1, 3, 2]
2.0
```

`mean`()
Compute the mean of this RDD’s elements.

```
>>> sc.parallelize([1, 2, 3]).mean()
2.0
```

variance

```
# variance
x = sc.parallelize([1,3,2])
y = x.variance()  # divides by N
print(x.collect())
print(y)
[1, 3, 2]
0.666666666667
```

`variance`**()**
Compute the variance of this RDD’s elements.

```
>>> sc.parallelize([1, 2, 3]).variance()
0.666...
```

stdev

```
# stdev
x = sc.parallelize([1,3,2])
y = x.stdev()  # divides by N
print(x.collect())
print(y)
[1, 3, 2]
0.816496580928
```

`stdev`**()**
Compute the standard deviation of this RDD’s elements.

```
>>> sc.parallelize([1, 2, 3]).stdev()
0.816...
```

sampleStdev

```
# sampleStdev
x = sc.parallelize([1,3,2])
y = x.sampleStdev() # divides by N-1
print(x.collect())
print(y)
[1, 3, 2]
1.0
```

`sampleStdev`**()**
Compute the sample standard deviation of this RDD’s elements (which corrects for bias in estimating the standard deviation by dividing by N-1 instead of N).

```
>>> sc.parallelize([1, 2, 3]).sampleStdev()
1.0
```

sampleVariance

```
# sampleVariance
x = sc.parallelize([1,3,2])
y = x.sampleVariance()  # divides by N-1
print(x.collect())
print(y)
[1, 3, 2]
1.0
```

`sampleVariance`**()**
Compute the sample variance of this RDD’s elements (which corrects for bias in estimating the variance by dividing by N-1 instead of N).

```
>>> sc.parallelize([1, 2, 3]).sampleVariance()
1.0
```

countByValue

```
# countByValue
x = sc.parallelize([1,3,1,2,3])
y = x.countByValue()
print(x.collect())
print(y)
[1, 3, 1, 2, 3]
defaultdict(<type 'int'>, {1: 2, 2: 1, 3: 2})
```

`countByValue`**()¶**
Return the count of each unique value in this RDD as a dictionary of (value, count) pairs.

top

```
# top
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]
```

`top`(*num*, *key=None*)
Get the top N elements from a RDD.

Note: It returns the list sorted in descending order.

```
>>> sc.parallelize([10, 4, 2, 12, 3]).top(1)
[12]
>>> sc.parallelize([2, 3, 4, 5, 6], 2).top(2)
[6, 5]
>>> sc.parallelize([10, 4, 2, 12, 3]).top(3, key=str)
[4, 3, 2]
```

takeOrdered

```
# takeOrdered
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]
```

`takeOrdered`(*num*, *key=None*)
Get the N elements from a RDD ordered in ascending order or as specified by the optional key function.

```
>>> sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7]).takeOrdered(6) # 升序排列 前6个
[1, 2, 3, 4, 5, 6]
>>> sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7], 2).takeOrdered(6, key=lambda x: -x)
[10, 9, 7, 6, 5, 4]
```

take

```
# take
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]
```

`take`(*num*)
Take the first num elements of the RDD.

It works by first scanning one partition, and use the results from that partition to estimate the number of additional partitions needed to satisfy the limit.

Translated from the Scala implementation in RDD#take().

```
>>> sc.parallelize([2, 3, 4, 5, 6]).cache().take(2)
[2, 3]
>>> sc.parallelize([2, 3, 4, 5, 6]).take(10)
[2, 3, 4, 5, 6]
>>> sc.parallelize(range(100), 100).filter(lambda x: x > 90).take(3)
[91, 92, 93]
```

first

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

`first`()
Return the first element in this RDD.

```
>>> sc.parallelize([2, 3, 4]).first()
2
>>> sc.parallelize([]).first()
Traceback (most recent call last):
    ...
ValueError: RDD is empty
```

collectAsMap

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

`collectAsMap`**()**
Return the key-value pairs in this RDD to the master as a dictionary.

```
>>> m = sc.parallelize([(1, 2), (3, 4)]).collectAsMap()
>>> m[1]
2
>>> m[3]
4
```

keys

```
# keys
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']
```

`keys`**()**
Return an RDD with the keys of each tuple.

```
>>> m = sc.parallelize([(1, 2), (3, 4)]).keys()
>>> m.collect()
[1, 3]
```

values

```
# values
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]
```

`values`**()**
Return an RDD with the values of each tuple.

```
>>> m = sc.parallelize([(1, 2), (3, 4)]).values()
>>> m.collect()
[2, 4]
```

reduceByKey

```
# reduceByKey
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)]
[('A', 12), ('B', 3)]
```

`reduceByKey`**(***func*, *numPartitions=None***)**
Merge the values for each key using an associative reduce function.

This will also perform the merging locally on each mapper before sending results to a reducer, similarly to a “combiner” in MapReduce.

Output will be hash-partitioned with `numPartitions` partitions, or the default parallelism level if `numPartitions` is not specified.

```
>>> from operator import add
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.reduceByKey(add).collect())
[('a', 2), ('b', 1)]
```

reduceByKeyLocally

```
# reduceByKeyLocally
x = sc.parallelize([('B',1),('B',2),('A',3),('A',4),('A',5)])
y = x.reduceByKeyLocally(lambda agg, obj: agg + obj)
print(x.collect())
print(y)
[('B', 1), ('B', 2), ('A', 3), ('A', 4), ('A', 5)]
{'A': 12, 'B': 3}
```

`reduceByKeyLocally`(*func*)
Merge the values for each key using an associative reduce function, but return the results immediately to the master as a dictionary.

This will also perform the merging locally on each mapper before sending results to a reducer, similarly to a “combiner” in MapReduce.

```
>>> from operator import add
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.reduceByKeyLocally(add).items())
[('a', 2), ('b', 1)]
```

countByKey

```
# countByKey
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(<type 'int'>, {'A': 3, 'B': 2})
```

`countByKey`()
Count the number of elements for each key, and return the result to the master as a dictionary.

```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> sorted(rdd.countByKey().items())
[('a', 2), ('b', 1)]
```

join

```
# join
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)]
[('A', (2, 8)), ('A', (2, 6)), ('A', (1, 8)), ('A', (1, 6)), ('B', (3, 7))]
```

`join`(*other*, *numPartitions=None*)
Return an RDD containing all pairs of elements with matching keys in `self` and `other`.

Each pair of elements will be returned as a (k, (v1, v2)) tuple, where (k, v1) is in `self` and (k, v2) is in `other`.

Performs a hash join across the cluster.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2), ("a", 3)])
>>> sorted(x.join(y).collect())
[('a', (1, 2)), ('a', (1, 3))]
```

leftOuterJoin

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

`leftOuterJoin`(*other*, *numPartitions=None*)
Perform a left outer join of `self` and `other`.

For each element (k, v) in `self`, the resulting RDD will either contain all pairs (k, (v, w)) for w in `other`, or the pair (k, (v, None)) if no elements in `other` have key k.

Hash-partitions the resulting RDD into the given number of partitions.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> sorted(x.leftOuterJoin(y).collect())
[('a', (1, 2)), ('b', (4, None))]
```

rightOuterJoin

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

`rightOuterJoin`(*other*, *numPartitions=None*)
Perform a right outer join of `self` and `other`.

For each element (k, w) in `other`, the resulting RDD will either contain all pairs (k, (v, w)) for v in this, or the pair (k, (None, w)) if no elements in `self` have key k.

Hash-partitions the resulting RDD into the given number of partitions.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> sorted(y.rightOuterJoin(x).collect())
[('a', (2, 1)), ('b', (None, 4))]
```

partitionBy

```
# partitionBy
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)]]
```

`partitionBy`(*numPartitions*, *partitionFunc=<function portable_hash at 0x3d132a8>*)
Return a copy of the RDD partitioned using the specified partitioner.

```
>>> pairs = sc.parallelize([1, 2, 3, 4, 2, 4, 1]).map(lambda x: (x, x))
>>> sets = pairs.partitionBy(2).glom().collect()
>>> set(sets[0]).intersection(set(sets[1]))
set([])
```

combineByKey

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

`combineByKey`(*createCombiner*, *mergeValue*, *mergeCombiners*, *numPartitions=None*)
Generic function to combine the elements for each key using a custom set of aggregation functions.

Turns an RDD[(K, V)] into a result of type RDD[(K, C)], for a “combined type” C. Note that V and C can be different – for example, one might group an RDD of type (Int, Int) into an RDD of type (Int, List[Int]).

Users provide three functions:

> - createCombiner, which turns a V into a C (e.g., creates a one-element list)
> - mergeValue, to merge a V into a C (e.g., adds it to the end of a list)
> - mergeCombiners, to combine two C’s into a single one.

In addition, users can control the partitioning of the output RDD.

```
>>> x = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> def f(x): return x
>>> def add(a, b): return a + str(b)
>>> sorted(x.combineByKey(str, add, add).collect())
[('a', '11'), ('b', '1')]
```

aggregateByKey

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

`aggregateByKey`(*zeroValue*, *seqFunc*, *combFunc*, *numPartitions=None*)
Aggregate the values of each key, using given combine functions and a neutral “zero value”. This function can return a different result type, U, than the type of the values in this RDD, V. Thus, we need one operation for merging a V into a U and one operation for merging two U’s, The former operation is used for merging values within a partition, and the latter is used for merging values between partitions. To avoid memory allocation, both of these functions are allowed to modify and return their first argument instead of creating a new U.

foldByKey

```
# foldByKey
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)]
[('A', 60), ('B', 2)]
```

`foldByKey`(*zeroValue*, *func*, *numPartitions=None*)
Merge the values for each key using an associative function “func” and a neutral “zeroValue” which may be added to the result an arbitrary number of times, and must not change the result (e.g., 0 for addition, or 1 for multiplication.).

```
>>> rdd = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> from operator import add
>>> rdd.foldByKey(0, add).collect()
[('a', 2), ('b', 1)]
```

groupByKey

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

`groupByKey`(*numPartitions=None*)
Group the values for each key in the RDD into a single sequence. Hash-partitions the resulting RDD with into numPartitions partitions.

Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey will provide much better performance.

```
>>> x = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
>>> map((lambda (x,y): (x, list(y))), sorted(x.groupByKey().collect()))
[('a', [1, 1]), ('b', [1])]
```

flatMapValues

```
# flatMapValues
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)]
```

`flatMapValues`(*f*)
Pass each value in the key-value pair RDD through a flatMap function without changing the keys; this also retains the original RDD's partitioning.

```
>>> x = sc.parallelize([("a", ["x", "y", "z"]), ("b", ["p", "r"])])
>>> def f(x): return x
>>> x.flatMapValues(f).collect()
[('a', 'x'), ('a', 'y'), ('a', 'z'), ('b', 'p'), ('b', 'r')]
```

mapValues

```
# mapValues
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])]
```

`mapValues`(*f*)
Pass each value in the key-value pair RDD through a map function without changing the keys; this also retains the original RDD's partitioning.

```
>>> x = sc.parallelize([("a", ["apple", "banana", "lemon"]), ("b", ["grapes"])])
>>> def f(x): return len(x)
>>> x.mapValues(f).collect()
[('a', 3), ('b', 1)]
```

groupWith

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

`groupWith`(*other*, **others*)
Alias for cogroup but with support for multiple RDDs.

```
>>> w = sc.parallelize([("a", 5), ("b", 6)])
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> z = sc.parallelize([("b", 42)])
>>> map((lambda (x,y): (x, (list(y[0]), list(y[1]), list(y[2]), list(y[3])))),                 sorted(list(w.groupWith(x, y, z).collect())))
[('a', ([5], [1], [2], [])), ('b', ([6], [4], [], [42]))]
```

cogroup

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

`cogroup`(*other*, *numPartitions=None*)
For each key k in `self` or `other`, return a resulting RDD that contains a tuple with the list of values for that key in `self` as well as `other`.

```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2)])
>>> map((lambda (x,y): (x, (list(y[0]), list(y[1])))), sorted(list(x.cogroup(y).collect())))
[('a', ([1], [2])), ('b', ([4], []))]
```

sampleByKey

```
# sampleByKey
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)]
```

`sampleByKey`(*withReplacement*, *fractions*, *seed=None*)
Return a subset of this RDD sampled by key (via stratified sampling). Create a sample of this RDD using variable sampling rates for different keys as specified by fractions, a key to sampling rate map.

```
>>> fractions = {"a": 0.2, "b": 0.1}
>>> rdd = sc.parallelize(fractions.keys()).cartesian(sc.parallelize(range(0, 1000)))
>>> sample = dict(rdd.sampleByKey(False, fractions, 2).groupByKey().collect())
>>> 100 < len(sample["a"]) < 300 and 50 < len(sample["b"]) < 150
True
>>> max(sample["a"]) <= 999 and min(sample["a"]) >= 0
True
>>> max(sample["b"]) <= 999 and min(sample["b"]) >= 0
True
```

subtractByKey

```
# subtractByKey
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)]
[('C', 1), ('B', 2)]
```

`subtractByKey`(*other*, *numPartitions=None*)
Return each (key, value) pair in `self` that has no pair with matching key in `other`.

```
>>> x = sc.parallelize([("a", 1), ("b", 4), ("b", 5), ("a", 2)])
>>> y = sc.parallelize([("a", 3), ("c", None)])
>>> sorted(x.subtractByKey(y).collect())
[('b', 4), ('b', 5)]
```

subtract

```
# subtract
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)]
[('A', 1), ('C', 4), ('B', 3)]
```

`subtract`(*other*, *numPartitions=None*)
Return each value in `self` that is not contained in `other`.

```
>>> x = sc.parallelize([("a", 1), ("b", 4), ("b", 5), ("a", 3)])
>>> y = sc.parallelize([("a", 3), ("c", None)])
>>> sorted(x.subtract(y).collect())
[('a', 1), ('b', 4), ('b', 5)]
```

keyBy

```
# keyBy
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)]
```

`keyBy`(*f*)
Creates tuples of the elements in this RDD by applying `f`.

```
>>> x = sc.parallelize(range(0,3)).keyBy(lambda x: x*x)
>>> y = sc.parallelize(zip(range(0,5), range(0,5)))
>>> map((lambda (x,y): (x, (list(y[0]), (list(y[1]))))), sorted(x.cogroup(y).collect()))
[(0, ([0], [0])), (1, ([1], [1])), (2, ([], [2])), (3, ([], [3])), (4, ([2], [4]))]
```

repartition

```
# repartition
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]]
```

`repartition`(*numPartitions*)
Return a new RDD that has exactly numPartitions partitions.

Can increase or decrease the level of parallelism in this RDD. Internally, this uses a shuffle to redistribute data. If you are decreasing the number of partitions in this RDD, consider using *coalesce*, which can avoid performing a shuffle.

```
>>> rdd = sc.parallelize([1,2,3,4,5,6,7], 4)
>>> sorted(rdd.glom().collect())
[[1], [2, 3], [4, 5], [6, 7]]
>>> len(rdd.repartition(2).glom().collect())
2
>>> len(rdd.repartition(10).glom().collect())
10
```

coalesce

```
# coalesce
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]]
```

`coalesce`(*numPartitions*, *shuffle=False*)
Return a new RDD that is reduced into *numPartitions* partitions.

```
>>> sc.parallelize([1, 2, 3, 4, 5], 3).glom().collect()
[[1], [2, 3], [4, 5]]
>>> sc.parallelize([1, 2, 3, 4, 5], 3).coalesce(1).glom().collect()
[[1, 2, 3, 4, 5]]
```

zip

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

`zip`(*other*)
Zips this RDD with another one, returning key-value pairs with the first element in each RDD second element in each RDD, etc. Assumes that the two RDDs have the same number of partitions and the same number of elements in each partition (e.g. one was made through a map on the other).

```
>>> x = sc.parallelize(range(0,5))
>>> y = sc.parallelize(range(1000, 1005))
>>> x.zip(y).collect()
[(0, 1000), (1, 1001), (2, 1002), (3, 1003), (4, 1004)]
```

zipWithIndex

```
# zipWithIndex
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)]
```

`zipWithIndex`()
Zips this RDD with its element indices.

The ordering is first based on the partition index and then the ordering of items within each partition. So the first item in the first partition gets index 0, and the last item in the last partition receives the largest index.

This method needs to trigger a spark job when this RDD contains more than one partitions.

```
>>> sc.parallelize(["a", "b", "c", "d"], 3).zipWithIndex().collect()
[('a', 0), ('b', 1), ('c', 2), ('d', 3)]
```

zipWithUniqueId

```
# zipWithUniqueId
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)]
```

`zipWithUniqueId`()
Zips this RDD with generated unique Long ids.

Items in the kth partition will get ids k, n+k, 2*n+k, ..., where n is the number of partitions. So there may exist gaps, but this method won't trigger a spark job, which is different from `zipWithIndex`

```
>>> sc.parallelize(["a", "b", "c", "d", "e"], 3).zipWithUniqueId().collect()
[('a', 0), ('b', 1), ('c', 4), ('d', 2), ('e', 5)]
```

使用py脚本

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

from pyspark.context import SparkContext
from pyspark.conf import SparkConf
#from pyspark.sql import DataFrame,SQLContext

sc = SparkContext(conf=SparkConf().setAppName("The first example"))

# map
x = sc.parallelize([1,2,3]) # sc = spark context, parallelize creates an RDD from the passed object
y = x.map(lambda x: (x,x**2))
print(x.collect())  # collect copies RDD elements to a list on the driver
print(y.collect())

执行：spark/bin/spark-submit test.py
```