RDD创建

1、从 Java 集合构造成 RDD

List data = Arrays.asList(1, 2, 3, 4, 5);

JavaRDD rdd = jsc.parallelize(data, 3) 3 是RDD分区个数

2. 从文件 / 目录构造 RDD

jsc.textFile(“/data”, 1) jsc.sequenceFile(“/data”, 1) jsc.wholeTextFiles(“/data”, 1)

Transformation操作

初始化Spark

In [2]:

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("My App")
sc = SparkContext(conf = conf)
```

map()操作

Return a new distributed dataset formed by passing each element of the source through a function func　接受一个函数，把这个函数作用于RDD中的每一个元素，将函数的返回结果作为结果RDD中对应元素的值

实例：求RDD中所有数的平方

In [101]:

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x : x * x).collect()
for num in squared:
    print  ("%i " % num)
sorted(squared)
```

```
1
4
9
16
```

Out[101]:

```
[1, 4, 9, 16]
```

注意：map()的返回值类型不需要和输入类型一样

flatMap()操作

Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item)　接受一个函数分别应用到输入RDD的每一个元素上，不过返回的不是一个元素，而是一个返回值序列的迭代器。

实例：把输入的字符串切分为单词

In [4]:

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line : line.split(" "))
words.first()
```

Out[4]:

```
'hello'
```

In [5]:

```
lines = sc.parallelize(["hello world", "hi"])
word = lines.map(lambda line : line.split(" "))
word.first()
```

Out[5]:

```
['hello', 'world']
```

flatMap相当于把返回的迭代器“拍扁”

glom()操作

将RDD中每个partition中元素转换为数组

In [6]:

```
nums = sc.parallelize([1, 2, 3, 4, 5, 6, 7])
result = nums.glom().collect()
print(result)
print("-------------")
print(nums)
```

```
[[1, 2, 3, 4, 5, 6, 7]]
-------------
ParallelCollectionRDD[6] at parallelize at PythonRDD.scala:475
```

filter()操作

Return a new dataset formed by selecting those elements of the source on which func returns true.接受一个函数，并将RDD中满足该函数的元素放入新的RDD中返回

In [7]:

```
nums = sc.parallelize([1, 2, 3, 4, 5, 6, 7])
even = nums.filter(lambda x : x % 2 == 0)
print(even.collect())
```

```
[2, 4, 6]
```

mapPartitions()操作

```
Similar to map, but runs separately on each partition (block) of the RDD, so func
must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.该函数
与map函数类似，只不过映射函数的参数由RDD中的每一个元素变为RDD中的每一个分区的迭代器。如果在映射
过程中需要频繁创建额外的对象，使用mapPartitions要比map更高效。比如，将RDD中的所有数据通过JDBC
连接写入数据库，如果使用map函数，可能要为每一个元素都要创建一个connection,这样开销很大，如果使
用mapPartions那么只需要为每一个分区创建一个connection
```

In [25]:

```
nums = sc.parallelize([1, 2, 3, 4, 5, 6, 7], 3)
print(nums.collect())
def f(iterator):
    while(iterator.hasNext)
numsPartitions = nums.mapPartitions(f).collect()
print(numsPartitions)
```

```
[1, 2, 3, 4, 5, 6, 7]
[3, 7, 18]
```

mapPartitionsWithIndex()操作

Similar to mapPartitions, but also provides func with an integer value representing the index of the partition so func must be of type (Int, Iterator) => Iterator when running on an RDD of type T.同mapPartitions， 不过提供了两个参数，第一个参数是分区的索引

In [28]:

```
nums = sc.parallelize([1, 2, 3, 4, 5, 6, 7], 3)
def f(splitIndex, iterator): yield splitIndex
nums.mapPartitionsWithIndex(f).collect()
```

Out[28]:

```
[0, 1, 2]
```

sample()操作

Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed 对RDD采样，以及是否替换。 :param withReplacement: can elements be sampled multiple times (replaced when sampled out) :param fraction: expected size of the sample as a fraction of this RDD's size without replacement: probability that each element is chosen; fraction must be [0, 1] with replacement: expected number of times each element is chosen; fraction must be >= 0 :param seed: seed for the random number generator

In [46]:

```
nums = sc.parallelize([1,2,3,4,5,6,7,8,9,0,10])
nums.sample(True, 0.6, 100).collect()
```

Out[46]:

```
[1, 2, 5, 5, 5, 6, 8, 10]
```

union()操作

Return a new dataset that contains the union of the elements in the source dataset and the argument

In [48]:

```
num1 = sc.parallelize([1,2,3,4,5,6,7,8,9,0,10])
num2 = sc.parallelize([11, 12])
num1.union(num2).collect()
```

Out[48]:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 10, 11, 12]
```

intersection()操作

Return a new RDD that contains the intersection of elements in the source dataset and the argument 返回一个新的RDD，其中包含源数据集中元素和参数的交集；操作的RDD是相同的数据类型。只返回两个RDD中都有的元素。在运行时也会去掉所有重复的元素。

In [50]:

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

Out[50]:

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

distinct()操作

Return a new dataset that contains the distinct elements of the source dataset.生成一个只包含不同元素的新RDD.distinct()操作的开销很大，因为它需要将所有的数据通过网络进行混洗，以确保每个元素只有一份。 操作的RDD是相同的数据类型。

In [52]:

```
rdd3 = sc.parallelize([1, 10, 2, 3, 4, 5, 5, 3, 2])
rdd3.distinct().collect()
```

Out[52]:

```
[1, 2, 3, 4, 5, 10]
```

groupByKey()操作

When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks."

In [57]:

```
rdd4 = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
rdd4.groupByKey().mapValues(list).collect()
```

Out[57]:

```
[('a', [1, 1]), ('b', [1])]
```

reduceByKey()操作

When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.

In [59]:

```
from operator import add
rdd5 = sc.parallelize([("a", 1), ("b", 1), ("a", 1)])
rdd5.reduceByKey(add).collect()
```

Out[59]:

```
[('a', 2), ('b', 1)]
```

aggregateByKey()操作

When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. 聚合操作,需要传入排序函数和聚合函数

In [68]:

```
rdd6 = sc.parallelize([("a", 1), ("b", 1), ("a", 1)],3)
def f(x, y) : return max(x,y)
def c(x,y) : return x+y
rdd6.aggregateByKey(0, f, c).collect()
```

Out[68]:

```
[('b', 1), ('a', 2)]
```

sortByKey()操作

When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument

In [71]:

```
rdd7 = sc.parallelize([(1, 1), (5, 1), (4, 1)],3)
rdd7.sortByKey(False).collect()
```

Out[71]:

```
[(5, 1), (4, 1), (1, 1)]
```

join()操作

When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin.

In [72]:

```
rdd8 = sc.parallelize([(1, 1), (5, 1), (4, 1)],3)
rdd9 = sc.parallelize([(1, 1), (5, 1), (4, 1)],2)
rdd8.join(rdd9).collect()
```

Out[72]:

```
[(5, (1, 1)), (1, (1, 1)), (4, (1, 1))]
```

cogroup()操作

When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable, Iterable)) tuples. This operation is also called groupWith.

In [73]:

```
x = sc.parallelize([("a", 1), ("b", 4)])
y = sc.parallelize([("a", 2)])
[(x, tuple(map(list, y))) for x, y in sorted(list(x.cogroup(y).collect()))]
```

Out[73]:

```
[('a', ([1], [2])), ('b', ([4], []))]
```

cartesian()操作

When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements). 求两个数据集的笛卡尔积。求大规模的笛卡尔积开销巨大。比如计算用户对各种产品的预期兴趣程度。

In [74]:

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

Out[74]:

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

pipe()操作

Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings. 通过管道的方式对RDD的每个分区使用shell命令进行操作，返回相应的结果

In [81]:

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

Out[81]:

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

coalesce()操作

Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset将RDD中的partition个数合并为numPartitions个.numPartitions必须比原始的分区数要小

In [90]:

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

Out[90]:

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

In [91]:

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

Out[91]:

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

repartition()操作

Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network. 将RDD中的 partition个数均匀合并为 numPartitions个；非常消耗资源需要shuffle

In [92]:

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

Out[92]:

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

In [94]:

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

Out[94]:

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

repartitionAndSortWithinPartitions()操作

Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery. 如果需要在repartition重分区之后，还要进行排序，建议直接使用repartitionAndSortWithinPartitions算子

In [95]:

```
rdd = sc.parallelize([(0, 5), (3, 8), (2, 6), (0, 8), (3, 8), (1, 3)])
rdd2 = rdd.repartitionAndSortWithinPartitions(2, lambda x: x % 2, 2)
rdd2.glom().collect()
```

Out[95]:

```
[[(0, 5), (0, 8), (2, 6)], [(1, 3), (3, 8), (3, 8)]]
```

subtract()操作

将原RDD里和参数RDD里相同的元素去掉Return each value in C{self} that is not contained in C{other}.

In [97]:

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

Out[97]:

```
[('a', 1), ('b', 4), ('b', 5)]
```

glom()操作

将RDD中每个partition中元素转换为数组 Return an RDD created by coalescing all elements within each partition into a list.

In [98]:

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

Out[98]:

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

zip()操作

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)

In [99]:

```
x = sc.parallelize(range(0,5))
y = sc.parallelize(range(1000, 1005))
x.zip(y).collect()
```

Out[99]:

```
[(0, 1000), (1, 1001), (2, 1002), (3, 1003), (4, 1004)]
```

Action操作

Reduce操作

reduce 它接受一个函数作为参数，这个函数要操作两个相同元素类型的RDD数据并返回一个同样类型的新元素。按照函数f对RDD中元素进行规约

In [49]:

```
sum = nums.reduce(lambda x , y : x + y)
print (sum)
```

```
55
```

Aggregate操作

需要我们提供需要返回的类型的初始值，然后再通过一个函数把RDD中的元素合并起来放入累加器。

In [102]:

```
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)
```

Out[102]:

```
(10, 4)
```

collect()操作

将RDD转化为数组(Driver端)，需要慎用，如果数据量大会把所有数据发送给driver

In [108]:

```
print(nums.collect())
```

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

count()操作

统计RDD中元素个数,并返回Long类型

In [109]:

```
print(nums.count())
```

```
4
```

take()操作

返回RDD中前k个元素,并保存成数组

In [110]:

```
print(nums.take(2))
```

```
[1, 2]
```

foreach()操作

对RDD中每个元素,调用函数f

In [111]:

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

first()操作

Return the first element of the dataset (similar to take(1)).

In [112]:

```
x = sc.parallelize(range(0,5))
x.first()
```

Out[112]:

```
0
```

takeSample()操作

Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed该方法仅在预期结果数组很小的情况下使用，因为所有数据都被加载到driver的内存中。

In [116]:

```
rdd = sc.parallelize(range(0, 10))
print(rdd.collect())
rdd.takeSample(True, 20, 1)
rdd.takeSample(False, 5, 2)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Out[116]:

```
[5, 9, 3, 4, 6]
```

takeOrdered()操作

Return the first n elements of the RDD using either their natural order or a custom comparator 该方法仅在预期结果数组很小的情况下使用，因为所有数据都被加载到driver的内存中。

In [117]:

```
sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7]).takeOrdered(6)
```

Out[117]:

```
[1, 2, 3, 4, 5, 6]
```

In [118]:

```
sc.parallelize([10, 1, 2, 9, 3, 4, 5, 6, 7], 2).takeOrdered(6, key=lambda x: -x)
```

Out[118]:

```
[10, 9, 7, 6, 5, 4]
```

saveAsTextFile()操作

Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file

saveAsSequenceFile()操作

Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).

saveAsObjectFile()操作

Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile()

countByKey()操作

Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.

In [119]:

```
x = sc.parallelize([("a", 1), ("b", 4), ("b", 5), ("a", 3)])
x.countByKey()
```

Out[119]:

```
defaultdict(int, {'a': 2, 'b': 2})
```

foreach()操作

"Run a function func on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. Note: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details"

In [121]:

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

In [ ]:

```

```