PySpark API

翻译自sark官方文档，作者：赵峰，2015年8月1日

1. public class:

SparkContext：Spark 主入口函数；
RDD：弹性分布数据集；

Broadcast：在任务中重复使用的broadcast变量；

Accumulator：An “add-only” shared variable that tasks can only add values to.

SparkConf：用于配置spark；

SparkFiles：Access files shipped with jobs.

StorageLevel：cache 存储级别的细分类。

2. class pyspark.SparkConf(loadDefaults=True, _jvm=None, _jconf=None)

spark应用的配置，用户设定spark变量为key-value形式。

大多数时候，你需要用sparkConf()创建一个sparkConf的对象，它也会从*spark.** Java加载一些变量的值，所有你在sparkConf中设定的变量

都要先于系统变量创建。

对于单元测试，你也可以用SparkConf(false)来跳过加载外部变量。

本类中的所有方法都支持链式调用，比如说，你可以conf.setMaster(“local”).setAppName(“My app”)。

一旦一个SparkConf 对象创建了传递到了spark，用户将不能再修改它。

contains(*key*)：给定的设置是否包含key；

get(*key*, *defaultValue=None*)：获取key的设定值，若没有则返回默认值；

set(*key*, *value*)：设定一个属性值；

setAll(*pairs*)：以key-value形式设定多个参数，pairs是一个列表；

setAppName(*value*)：设定应用的名字；

setExecutorEnv(*key=None*, *value=None*, *pairs=None*)：设定环境变量；

setIfMissing(*key*, *value*)：设定属性值，如果这一属性还没被设置；

setMaster(*value*)：设定Master的URL；

setSparkHome(*value*)：设定spark在工作节点上的路径；

toDebugString()：返回系统设置的一个可输出的版本，以key-value 一个list的形式，一行一个。

3. class pyspark.SparkContext(master=None, appName=None, sparkHome=None, pyFiles=None, environment=None, batchSize=0, serializer=PickleSerializer(), conf=None, gateway=None, jsc=None, profiler_cls=<class 'pyspark.profiler.BasicProfiler'>)

spark函数的入口，一个SparkContext 代表连接spark集群的连接，可用于创建RDD和Broadcast变量。

PACKAGE_EXTENSIONS = *('.zip', '.egg', '.jar')*；

用给定的值创建一个Accumulator，使用AccumulatorParam对象来定义怎样累加数据，对于整数型和浮点型提供默认的AccumulatorParam，其他类型则使用自定义的。

addFile(*path*)：每个节点上加载一个将要下载的文件，path既可以是本地文件，也可以是hdfs，http，https或ftp文件。

```
>>> from pyspark import SparkFiles
>>> path=os.path.join(tempdir,"test.txt")
>>> with open(path,"w") as testFile:    ... _=testFile.write("100")
>>> sc.addFile(path)
>>> def func(iterator):    ... with open(SparkFiles.get("test.txt")) as testFile:    ... fileVal=int(testFile.readline())    ... return [x*fileVal for x in iterator]
>>> sc.parallelize([1,2,3,4]).mapPartitions(func).collect()
 [100, 200, 300, 400]
```

addPyFile(*path*)：加载一个.py或.zip文件，path既可以是本地文件，也可以是hdfs，http，https或ftp文件。
binaryFiles(*path*, *minPartitions=None*)：从hdfs或Hadoop支持的其他文件系统上读取二进制文件的目录，输出成一个数组，每个文件被当成单一的文档读入，并返回一个
key-value对，key是文件路径，value是文件内容。最好读取小文件，大文件也支持，但效率可能不太好。
binaryRecords(*path*, *recordLength*)：从一个二进制文件读取数据，这里假定每一个文件是一个特定格式的数字的集合，文件中字节的长度也是固定的。
broadcast(*value*)：在集群中广播一个只读的变量，返回一个可以在分布式的函数中使用的
L{Broadcast<pyspark.broadcast.Broadcast>}对象，它只会发送给每个节点一次。
cancelAllJobs()：取消所有作业。
cancelJobGroup(*groupId*)：取消特定组的作业。
clearFiles()：清楚作业列表中使用addFile或addPyFile添加的作业，使他们不会发送到任何节点。
defaultMinPartitions：默认的Hadoop RDD最小的分区数。
defaultParallelism：默认的作业的并行数。
dump_profiles(*path*)：将profile状态导出到输出目录。
emptyRDD()：创建一个空的RDD。
getLocalProperty(*key*)：获取这一线程中的局部属性，如果没有则为null。

hadoopFile(*path*, *inputFormatClass*, *keyClass*, *valueClass*, *keyConverter=None*, *valueConverter=None*, *conf=None*, *batchSize=0*)：
hadoopRDD(*inputFormatClass*, *keyClass*, *valueClass*, *keyConverter=None*, *valueConverter=None*, *conf=None*, *batchSize=0*)：
newAPIHadoopFile(*path*, *inputFormatClass*, *keyClass*, *valueClass*, *keyConverter=None*, *valueConverter=None*, *conf=None*, *batchSize=0*)：
newAPIHadoopRDD(*inputFormatClass*, *keyClass*, *valueClass*, *keyConverter=None*, *valueConverter=None*, *conf=None*, *batchSize=0*)：

parallelize(*c*, *numSlices=None*)：将一个本地的python 集合转成一个RDD。

```
>>> sc.parallelize([0,2,3,4,6],5).glom().collect()
  [[0], [2], [3], [4], [6]]
>>> sc.parallelize(xrange(0,6,2),5).glom().collect()
  [[], [0], [], [2], [4]]
```

pickleFile(name, minPartitions=None)：加载一个之前保存的RDD，与RDD.saveAsPickleFile函数相对应。

```
>>> tmpFile=NamedTemporaryFile(delete=True)
>>> tmpFile.close()
>>> sc.parallelize(range(10)).saveAsPickleFile(tmpFile.name,5)
>>> sorted(sc.pickleFile(tmpFile.name,3).collect())     [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

range(*start*, *end=None*, *step=1*, *numSlices=None*)：创建一个包含int值的RDD，它的元素范围从start到end，步长为step，如果只用一个传入一个形参调用，则默认为start

为0，end为形参。**numSlices** 为RDD的分区数。

```
>>> sc.range(5).collect()       [0, 1, 2, 3, 4]
>>> sc.range(2,4).collect()     [2, 3]
>>> sc.range(1,7,2).collect()   [1, 3, 5]
```

runJob(*rdd*, *partitionFunc*, *partitions=None*, *allowLocal=False*)：在特定分区上执行给定函数*partitionFunc*，如果没指定分区就将子啊所有分区上执行。

```
>>> myRDD = sc.parallelize(range(6), 3)
>>> sc.runJob(myRDD, lambda part: [x * x for x in part])
[0, 1, 4, 9, 16, 25]
```

sequenceFile(path, keyClass=None, valueClass=None, keyConverter=None, valueConverter=None, minSplits=None, batchSize=0)：从HDFS或本地

文件系统等读取Hadoop序列文件。

setCheckpointDir(*dirName*)：设定RDD将要被标记成检查点的文件的目录，如果在集群上跑必须是HDFS上的目录。

setJobGroup(*groupId*, *description*, *interruptOnCancel=False*)：给这一线程启动的作业分配一个group id, 知道group id 被清楚或分配为另一个值。通常，一个应用上的执

行单元包含多个spark作业，用户可以通过这一函数将所有的这些工作分到一个组内，一旦设定，spark web UI会将这些作业联系到一起。

setLocalProperty(*key*, *value*)：设定一个可以影响作业从本线程提交的局部属性，比如spark的调度池。

setLogLevel(*logLevel*)：控制Log的级别，这重写了所有用户定义的log设置，有效的log级别包括：ALL, DEBUG, ERROR, FATAL, INFO, OFF, TRACE, WARN。

setSystemProperty(*key*, *value*)：设定一个Java系统属性，比如spark.executor.memory，必须在实例化SparkContext之前调用。

show_profiles()：打印出简介。

sparkUser()：获取spark的运行者。

startTime：返回spark开始运行的时间。

statusTracker()：返回一个StatusTracker对象。

stop()：停止SparkContext。

textFile(*name*, *minPartitions=None*, *use_unicode=True*)：从HDFS或其他文件系统上读取文件，如果use_unicode 为False，字符串将保存为utf-8编码格式。

```
>>> path = os.path.join(tempdir, "sample-text.txt")
>>> with open(path, "w") as testFile:
...     _ = testFile.write("Hello world!")
>>> textFile = sc.textFile(path)
>>> textFile.collect()
[u'Hello world!']
```

union(*rdds*)：连接两个RDD，支持不同序列化格式的RDD，这会迫使它们转化成默认的序列化格式。

version：返回正在运行的spark版本。

wholeTextFiles(*path*, *minPartitions=None*, *use_unicode=True*)：从HDFS或其他文件系统上读取文件，每一个文件作为一个单一的记录进行读取，并返回一个键值对，

其中key是文件路径，value是文件内容。如果use_unicode 为False，字符串将保存为utf-8编码格式。

```
>>> dirPath = os.path.join(tempdir, "files")
>>> os.mkdir(dirPath)
>>> with open(os.path.join(dirPath, "1.txt"), "w") as file1:
...     _ = file1.write("1")
>>> with open(os.path.join(dirPath, "2.txt"), "w") as file2:
...     _ = file2.write("2")
>>> textFiles = sc.wholeTextFiles(dirPath)
>>> sorted(textFiles.collect())
[(u'.../1.txt', u'1'), (u'.../2.txt', u'2')]
```

4. class pyspark.SparkFiles

SparkFiles:只有函数，用户不能创建实例。

get(*filename*)：获取由addFile 添加的文件的绝对路径。

getRootDirectory()：获取由addFile 添加的文件的根目录。

5. class pyspark.RDD(jrdd, ctx, jrdd_deserializer=AutoBatchedSerializer(PickleSerializer()))

RDD（弹性分布式数据集），表示一个不可改变的，可以并行的分区的元素的集合。

aggregate(*zeroValue*, *seqOp*, *combOp*) ：函数*op(t1, t2)* 用于修改t1，然后返回它的计算结果，但不能修改t2。seqOp可以返回一个不同类型的结果，因此，我们需要一个把T合并到U和合并两个U的操作。

```
>>> seqOp=(lambdax,y:(x[0]+y,x[1]+1))
>>> combOp=(lambdax,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)
```

aggregateByKey(zeroValue, seqFunc, combFunc, numPartitions=None)：合并每个key的值，使用给定的合并函数和初始值，这一函数可以返回不同类型的结果，因此我们

需要一个把U合并到V和合并两个U的函数，前一个函数是用于合并同一个分区的值，后一个是用于和合并不同分区的值。这两个函数都可以修改和返回他们的第一个形参而不是创建一个

新的U。

cache()：使用默认的存储方法保留此RDD。

cartesian(*other*)：返回此RDD和另一个的笛卡尔积。

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

checkpoint()：标记此RDD为检查点，它将被保存到checkpoint 目录下的一个文件中，所有其父RDD的引用都要被移除。在任何作业在此RDD上被执行前都要被调用，强烈建议此RDD保存在内存中，或需要重新计算的时候保存到一个文件。

coalesce(*numPartitions*, *shuffle=False*)：返回一个规约成*numPartitions* 个分区的RDD。

```
>>> 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]]
```

cogroup(*other*, *numPartitions=None*)：对每一个自身的和other的key，返回一个包含元组的RDD，元组中是自身和other中的key和他们的value组成的列表。

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

collect()：将RDD转化成列表。

collectAsMap()：将RDD转化成key-value对。

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

```
>>> m[3]
4
```

combineByKey(createCombiner, mergeValue, mergeCombiners, numPartitions=None)：对每个key使用一个聚合函数来结合起来的函数，将RDD[(K, V)] 转化成 RDD[(K, C)]形式，C是一种结合的形式，且V和C可以是不同的类别，比如我们可能将一个(int, int)形式的RDD转化成(int, List[int])形式。

createCombiner, 把V转化成C，如创建一个一个元素的的列表；

mergeValue, 把V合并成C，如将int 添加到list 的末尾；

mergeCombiners, 将两个C合并。

```
>>> 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')]
```

context：RDD创建的sparkContext。

count()：返回此RDD上的元素个数。

countApprox(timeout, confidence=0.95)：与count()相似，返回给定时间内的元素计数，即便有些元素还没被访问。

countApproxDistinct(relativeSD=0.05)：返回RDD上的唯一元素的近似值。

```
>>> n=sc.parallelize(range(1000)).map(str).countApproxDistinct()
>>> 900<n<1100
True
>>> n=sc.parallelize([i%20 fori in range(1000)]).countApproxDistinct()
>>> 16<n<24
True
```

countByKey()：对key计数，以字典的形式返回。

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

countByValue()：对value计数，以字典形式返回。

```
>>> sorted(sc.parallelize([1,2,1,2,2],2).countByValue().items())
[(1, 2), (2, 3)]
```

distinct(numPartitions=None)：返回唯一的元素。

filter(*f*)：返回符合要求的元素。

first()：返回RDD的第一个元素。

flatMap(*f*, *preservesPartitioning=False*)：将函数f 作用于一个RDD上的所有元素，但把结果压平。

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

flatMapValues(*f*)：对RDD上的所有键值对的value进行一个flatmap函数，key保持不变。

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

```
>>> x.flatMapValues(f).collect()
[('a', 'x'), ('a', 'y'), ('a', 'z'), ('b', 'p'), ('b', 'r')]
```

fold(*zeroValue*, *op*)：合并每个分区的元素，然后所有的元素一起合并，使用给定的计算公式和初始值。函数op(t1, t2)是用来修改t1并返回其值的，但不会修改t2的值。这一操作可能会先在各个分区内部进行操作，然后把结果合并到最后的结果中去，而不是对每个元素按顺序进行合并。

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

foldByKey(*zeroValue*, *func*, *numPartitions=None*)：对每个key使用函数“fun”和初始值zeroValues进行合并。

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

foreach(*f*)：将函数f作用于RDD内所有元素。

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

foreachPartition(*f*)：将函数f作用于RDD上每个分区。

fullOuterJoin(*other*, *numPartitions=None*)：将此RDD与other进行右外连接，对本身的每个(K,V)，若other中有key K，则连接后的RDD为k, (v, w),否则为 (k, (v, None))。

对other的每个键值对也有同样的结果。

```
>>> x=sc.parallelize([("a",1),("b",4)])
>>> y=sc.parallelize([("a",2),("c",8)])
>>> sorted(x.fullOuterJoin(y).collect())
[('a', (1, 2)), ('b', (4, None)), ('c', (None, 8))]
```

getCheckpointFile()：获取RDD是检查点的文件的名字。

getNumPartitions()：返回分区的个数。

getStorageLevel()：返回RDD现在的存储级别。

```
>>> rdd1=sc.parallelize([1,2])
>>> rdd1.getStorageLevel()
StorageLevel(False, False, False, False, 1)
>>> print(rdd1.getStorageLevel())
Serialized 1x Replicated
```

glom()：在各个分区里聚合元素成一个列表。

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

groupBy(*f*, *numPartitions=None*)：返回经过分组的RDD。

```
>>> 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])]
```

groupByKey(*numPartitions=None*)：通过RDD中的key进行分组。如果你想通过groupByKey进行聚合运算，使用aggregateByKey 或reduceByKey效果更好。

```
>>> rdd=sc.parallelize([("a",1),("b",1),("a",1)])
>>> sorted(rdd.groupByKey().mapValues(len).collect())
[('a', 2), ('b', 1)]
>>> sorted(rdd.groupByKey().mapValues(list).collect())
[('a', [1, 1]), ('b', [1])]
```

groupWith(*other*, **others*)：分组的别名，且支持多RDD。

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

histogram(*buckets*): 使用给定的桶计算直方图，这些桶都是左闭右开的，比如[1,10,20,50]表示这些桶是 [1,10) [10,20) [20,50]。桶必须是排序好的且不能有重复元素，且必须至少两个元素。

```
>>> 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])
```

id()：每个RDD特有的id。

intersection(*other*)：返回两个RDD之间的交集。

```
>>> 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]
```

isCheckpointed()：返回此RDD是否被检查过。

isEmpty()：返回此RDD是否为空。

join(*other*, *numPartitions=None*)：返回两个RDD有相同的键的键值对，假如自身RDD有(k, v1)，另一个RDD有(k, v2)，则返回(k, (v1, v2))。

```
>>> 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))]
```

keyBy(*f*)：对RDD上元素施加函数f。

```
>>> x=sc.parallelize(range(0,3)).keyBy(lambdax:x*x)

>>> y=sc.parallelize(zip(range(0,5),range(0,5)))

>>> [(x,list(map(list,y)))forx,yinsorted(x.cogroup(y).collect())]

[(0, [[0], [0]]), (1, [[1], [1]]), (2, [[], [2]]), (3, [[], [3]]), (4, [[2], [4]])]
```

keys()：返回RDD内元组的key。

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

leftOuterJoin(*other*, *numPartitions=None*)：将此RDD与另一个左连接，对于此RDD内的所有(k, v)，若(k, w)在另一个RDD中，则返回(k, (v, w))，否则返回(k, (v, None))

lookup(key)：返回RDD中key为*key*的值。

```
>>> l = range(1000)
>>> rdd = sc.parallelize(zip(l, l), 10)
>>> rdd.lookup(42)  # slow
[42]
>>> sorted = rdd.sortByKey()
>>> sorted.lookup(42)  # fast
[42]
>>> sorted.lookup(1024)
[]
```

map(*f*, *preservesPartitioning=False*)：对RDD内每个元素施加函数f。

```
>>> rdd=sc.parallelize(["b","a","c"])

>>> sorted(rdd.map(lambdax:(x,1)).collect())

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

mapPartitions(*f*, *preservesPartitioning=False*)：对RDD内每个分区的元素施加函数f。

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

>>> def f(iterator):yield sum(iterator)

>>> rdd.mapPartitions(f).collect()

[3, 7]
```

mapPartitionsWithIndex(*f*, *preservesPartitioning=False*)：在mapPartitions的基础上保留元素的index。

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

>>> def f(splitIndex,iterator):yield splitIndex

>>> rdd.mapPartitionsWithIndex(f).sum()

6
```

mapPartitionsWithSplit(*f*, *preservesPartitioning=False*)：被mapPartitionsWithIndex取代。

mapValues(*f*)：对RDD上每个value施加map函数f，而不改变其key。

```
>>> x=sc.parallelize([("a",["apple","banana","lemon"]),("b",["grapes"])])
```

```
>>> def f(x):return len(x)
>>> x.mapValues(f).collect()
[('a', 3), ('b', 1)]
```

max(*key=None*)：返回RDD中最大的元素。

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

mean()：计算RDD中元素的平均值。

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

meanApprox(*timeout*, *confidence=0.95*)：在一定timeoout和满足某些需要的情况下返回sum。

```
>>> rdd=sc.parallelize(range(1000),10)
>>> r=sum(range(1000))/1000.0
>>> abs(rdd.meanApprox(1000)-r)/r<0.05
True
```

min(*key=None*)：返回RDD中元素的最小值。

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

name()：返回RDD的名字。

partitionBy(*numPartitions*, *partitionFunc=<function portable_hash at 0x7f8d1be68a28>*)：使用特定的分区函数进行分区。

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

persist(*storageLevel=StorageLevel(False*, *True*, *False*, *False*, *1)*)：设定RDD的存储级别来保留它的值，当RDD还没有存储级别的时候可以用于分配存储级别。

```
>>> rdd=sc.parallelize(["b","a","c"])
>>> rdd.persist().is_cached
True
```

pipe(*command*, *env={}*)：

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

randomSplit(*weights*, *seed=None*)：使用给定的权重随机划分RDD。

```
>>> rdd=sc.parallelize(range(500),1)
>>> rdd1,rdd2=rdd.randomSplit([2,3],17)
>>> len(rdd1.collect()+rdd2.collect())
500
>>> 150<rdd1.count()<250
True
>>> 250<rdd2.count()<350
True
```

reduce(*f*)：使用给定函数对RDD内元素进行聚合。

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

reduceByKey(*func*, *numPartitions=None*)：使用给定函数f对RDD中的value进行聚合。

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

reduceByKeyLocally(*func*)：用给定的函数聚合每个key，这将先在每个分区进行然后再把结果发送到reducer，跟mapreduce的combiner类似。

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

repartition(*numPartitions*)：返回一个有*numPartitions*个分区的RDD，可以增加或降低RDD的并行级别。在内部使用洗牌来重新分布化数据，如果减少分区数，建议使用*coalesce*。

```
>>> 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
```

repartitionAndSortWithinPartitions(*numPartitions=None*, *partitionFunc=<function portable_hash at 0x7f8d1be68a28>*, *ascending=True*, *keyfunc=<function <lambda> at 0x7f8d1be6e230>*)：

通过给定的partitioner 函数对RDD重新分区，每个分区中按key排序。

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

rightOuterJoin(*other*, *numPartitions=None*)：又外连接，与左外连接相似。

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

sample(*withReplacement*, *fraction*, *seed=None*)：返回RDD的一个子集。

```
>>> rdd=sc.parallelize(range(100),4)
>>> 6<=rdd.sample(False,0.1,81).count()<=14
True
```

sampleByKey(*withReplacement*, *fractions*, *seed=None*)：返回RDD一个通过key采样的子集，

```
>>> 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"])<300and50<len(sample["b"])<150
True
>>> max(sample["a"])<=999andmin(sample["a"])>=0
True
>>> max(sample["b"])<=999andmin(sample["b"])>=0
True
```

sampleStdev()：RDD采样的子集的标准差。

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

sampleVariance()：RDD采样的子集的方差。

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

sampleVariance()：RDD采样自己的方差。

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

saveAsHadoopDataset(*conf*, *keyConverter=None*, *valueConverter=None*)：

saveAsHadoopFile(*path*, *outputFormatClass*, *keyClass=None*, *valueClass=None*, *keyConverter=None*, *valueConverter=None*, *conf=None*, *compressionCodecClass=None*)

saveAsNewAPIHadoopDataset(*conf*, *keyConverter=None*, *valueConverter=None*)

saveAsNewAPIHadoopFile(*path*, *outputFormatClass*, *keyClass=None*, *valueClass=None*, *keyConverter=None*, *valueConverter=None*, *conf=None*)

saveAsPickleFile(*path*, *batchSize=10*)：将此RDD的序列化对象保存为序列化文件。

```
>>> tmpFile = NamedTemporaryFile(delete=True)
>>> tmpFile.close()
```

```
>>> sc.parallelize([1, 2, 'spark', 'rdd']).saveAsPickleFile(tmpFile.name, 3)
>>> sorted(sc.pickleFile(tmpFile.name, 5).map(str).collect())
['1', '2', 'rdd', 'spark']
```

saveAsSequenceFile(*path*, *compressionCodecClass=None*)：将此RDD保存为序列化文件。

```
>>> tmpFile=NamedTemporaryFile(delete=True)

>>> tmpFile.close()

>>> sc.parallelize([1,2,'spark','rdd']).saveAsPickleFile(tmpFile.name,3)

>>> sorted(sc.pickleFile(tmpFile.name,5).map(str).collect())

['1', '2', 'rdd', 'spark']
```

saveAsTextFile(*path*, *compressionCodecClass=None*)：保存RDD，每个元素为一个字符串。

```
>>> tempFile=NamedTemporaryFile(delete=True)

>>> tempFile.close()

>>> sc.parallelize(range(10)).saveAsTextFile(tempFile.name)

>>> from fileinput import input

>>> from glob import glob

>>> ''.join(sorted(input(glob(tempFile.name+"/part-0000*"))))

'0\n1\n2\n3\n4\n5\n6\n7\n8\n9\n'
```

使用压缩形式保存：

```
>>> tempFile3=NamedTemporaryFile(delete=True)

>>> tempFile3.close()

>>> codec="org.apache.hadoop.io.compress.GzipCodec"

>>> sc.parallelize(['foo','bar']).saveAsTextFile(tempFile3.name,codec)

>>> from fileinput import input,hook_compressed

>>> result=sorted(input(glob(tempFile3.name+"/part*.gz"),openhook=hook_compressed))

>>> b''.join(result).decode('utf-8')

u'bar\nfoo\n'
```

setName(*name*)：给此RDD命名。

sortBy(*keyfunc*, *ascending=True*, *numPartitions=None*)：将RDD按给定的key排序。

```
>>> tmp=[('a',1),('b',2),('1',3),('d',4),('2',5)]

>>> sc.parallelize(tmp).sortBy(lambdax:x[0]).collect()

[('1', 3), ('2', 5), ('a', 1), ('b', 2), ('d', 4)]

>>> sc.parallelize(tmp).sortBy(lambdax:x[1]).collect()

[('a', 1), ('b', 2), ('1', 3), ('d', 4), ('2', 5)]
```

sortByKey(*ascending=True*, *numPartitions=None*, *keyfunc=<function <lambda> at 0x7f8d1be6e320>*)：将RDD按key排序。

```
>>> 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=lambdak:k.lower()).collect()
[('a', 3), ('fleece', 7), ('had', 2), ('lamb', 5),...('white', 9), ('whose', 6)]
```

stats()：返回一个**StatCounter** 对象保留了RDD元素的均值，方差和元素个数。

stdev()：计算RDD的标准差。

subtract(*other*, *numPartitions=None*)：将此RDD与other相减。

```
>>> 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)]
```

subtractByKey(*other*, *numPartitions=None*)：返回other中不包括的key的键值对。

```
>>> 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)]
```

sum()：返回RDD元素之和。

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

sumApprox(*timeout*, *confidence=0.95*)：在一定timeoout和满足某些需要的情况下返回sum。

```
>>> rdd = sc.parallelize(range(1000), 10)
>>> r = sum(range(1000))
>>> abs(rdd.sumApprox(1000) - r) / r < 0.05
True
```

take(*num*)：取RDD中前num个元素

```
>>> 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]
```

takeOrdered(*num*, *key=None*)：从RDD中取得递增或按给定key排序的前num个元素。

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

takeSample(*withReplacement*, *num*, *seed=None*)：返回RDD的定长的子集。

```
>>> 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
```

toDebugString()：RDD的描述和debug的一些依赖。

toLocalIterator()：返回RDD的一个迭代器。

```
>>> rdd = sc.parallelize(range(10))
>>> [x for x in rdd.toLocalIterator()]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

top(*num*, *key=None*)：返回RDD的前N个元素（以递减形式）。

```
>>> 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]
```

treeAggregate(*zeroValue*, *seqOp*, *combOp*, *depth=2*)：以多层树的形式聚合RDD中的元素。

```
>>> add = lambda x, y: x + y
>>> rdd = sc.parallelize([-5, -4, -3, -2, -1, 1, 2, 3, 4], 10)
>>> rdd.treeAggregate(0, add, add)
-5
>>> rdd.treeAggregate(0, add, add, 1)
-5
>>> rdd.treeAggregate(0, add, add, 2)
-5
>>> rdd.treeAggregate(0, add, add, 5)
-5
>>> rdd.treeAggregate(0, add, add, 10)
-5
```

treeReduce(*f*, *depth=2*)：以多层树的形式reduce RDD中的元素。

```
>>> add = lambda x, y: x + y
>>> rdd = sc.parallelize([-5, -4, -3, -2, -1, 1, 2, 3, 4], 10)
>>> rdd.treeReduce(add)
-5
>>> rdd.treeReduce(add, 1)
-5
>>> rdd.treeReduce(add, 2)
-5
>>> rdd.treeReduce(add, 5)
-5
>>> rdd.treeReduce(add, 10)
-5
```

union(*other*)：将此RDD与另一个联合。

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

unpersist()：将RDD标记成非持久的，并把所有模块从内存和磁盘上删除。

values()：返回RDD中每个元组的value。

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

variance()：计算RDD中元素的方差。

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

zip(*other*)：将RDD与另一个进行zip，返回以第一个RDD的值为key，第二个为value的RDD。

```
>>> 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()：将RDD与元素下标进行zip，这一方法在RDD在多个分区的时候会触发一个spark操作。

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

zipWithUniqueId()：将RDD与生成的唯一的id进行zip，第k个分区的id为：k, n+k, 2*n+k，n是分区的个数，这一方法可以避免触发spark操作。

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

6. class pyspark.StorageLevel(useDisk, useMemory, useOffHeap, deserialized, replication=1)

控制RDD存储的标志，每个存储级别控制是否存储，硬盘空间不足时是否删除RDD，是否以序列的方式将数据保存在RDD中，是否在多个节点上复制RDD分区，也包括一些常用的静态常量。

DISK_ONLY = *StorageLevel(True, False, False, False, 1)*

DISK_ONLY_2 = *StorageLevel(True, False, False, False, 2)*

MEMORY_AND_DISK = *StorageLevel(True, True, False, True, 1)*¶

MEMORY_AND_DISK_2 = *StorageLevel(True, True, False, True, 2)*

MEMORY_AND_DISK_SER = *StorageLevel(True, True, False, False, 1)*

MEMORY_AND_DISK_SER_2 = *StorageLevel(True, True, False, False, 2)*

MEMORY_ONLY = *StorageLevel(False, True, False, True, 1)*

MEMORY_ONLY_2 = *StorageLevel(False, True, False, True, 2)*

MEMORY_ONLY_SER = *StorageLevel(False, True, False, False, 1)*

MEMORY_ONLY_SER_2 = *StorageLevel(False, True, False, False, 2)*

OFF_HEAP = *StorageLevel(False, False, True, False, 1)*

7. class pyspark.Broadcast(sc=None, value=None, pickle_registry=None, path=None)

用SparkContext.broadcast()创建的广播变量，使用value访问其值。

```
>>> frompyspark.contextimportSparkContext>>> sc=SparkContext('local','test')>>> b=sc.broadcast([1,2,3,4,5
])>>> b.value[1, 2, 3, 4, 5]>>> sc.parallelize([0,0]).flatMap(lambdax:b.value).collect()[1, 2, 3, 4, 5, 1
, 2, 3, 4, 5]>>> b.unpersist()
```

dump(*value*, *f*)：

load(*path*)：

unpersist(*blocking=False*)：在运行的节点上删除缓存的广播变量的副本。

value：返回变量的值

#以下**class**及其少用

8. class pyspark.Accumulator(aid, value, accum_param)
9. class pyspark.AccumulatorParam
10. class pyspark.MarshalSerializer
11. class pyspark.PickleSerializer
12. class pyspark.StatusTracker(jtracker)
13. class pyspark.SparkJobInfo
14. class pyspark.SparkStageInfo¶
15. class pyspark.Profiler(ctx)
16. class pyspark.BasicProfiler(ctx)