RDD创建

1.从本地创建

```
>>> lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> lines.foreach(print)
Hadoop is good
Spark is fast
Spark is better
```

2.从本地文件系统创建

```
>>> lines = sc.textFile("hdfs://localhost:9000/user/hadoop/word.txt")
>>> lines = sc.textFile("/user/hadoop/word.txt")
>>> lines = sc.textFile("word.txt")
```

3.通过集合创建

```
>>> array = [1,2,3,4,5]
>>> rdd = sc.parallelize(array)
>>> rdd.foreach(print)
1
2
3
4
5
```

RDD操作

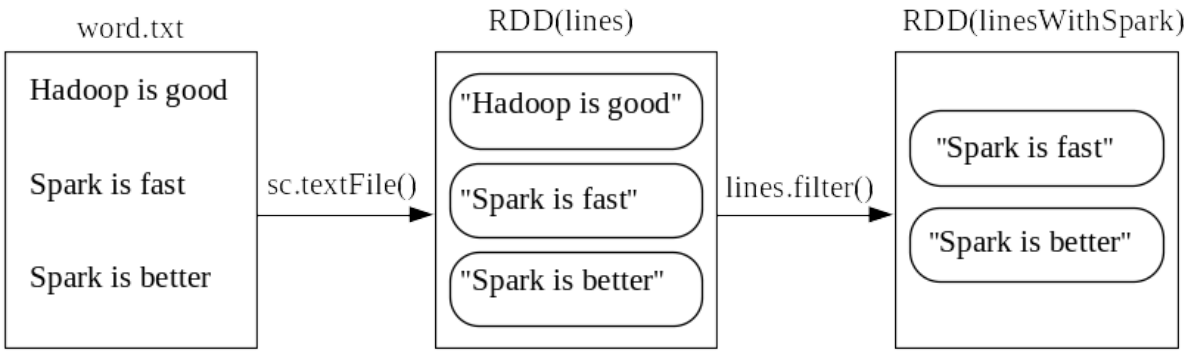
1.转换操作

（1）对于RDD而言，每一次转换操作都会产生不同的RDD，供给下一个“转换”使用
（2）转换得到的RDD是惰性求值的，也就是说，整个转换过程只是记录了转换的轨迹，并不会发生真正的计算，只有遇到行动操作时，才会发生真正的计算，开始从血缘关系源头开始，进行物理的转换操作

操作	含义
filter(func)	筛选出满足函数 func 的元素，并返回一个新的数据集
map(func)	将每个元素传递到函数 func 中，并将结果返回为一个新的数据集
flatMap(func)	与 map() 相似，但每个输入元素都可以映射到 0 或多个输出结果
groupByKey()	应用于 (K,V) 键值对的数据集时，返回一个新的 (K, Iterable) 形式的数据集
reduceByKey(func)	应用于 (K,V) 键值对的数据集时，返回一个新的 (K, V) 形式的数据集，其中每个值是将每个 key 传递到函数 func 中进行聚合后的结果

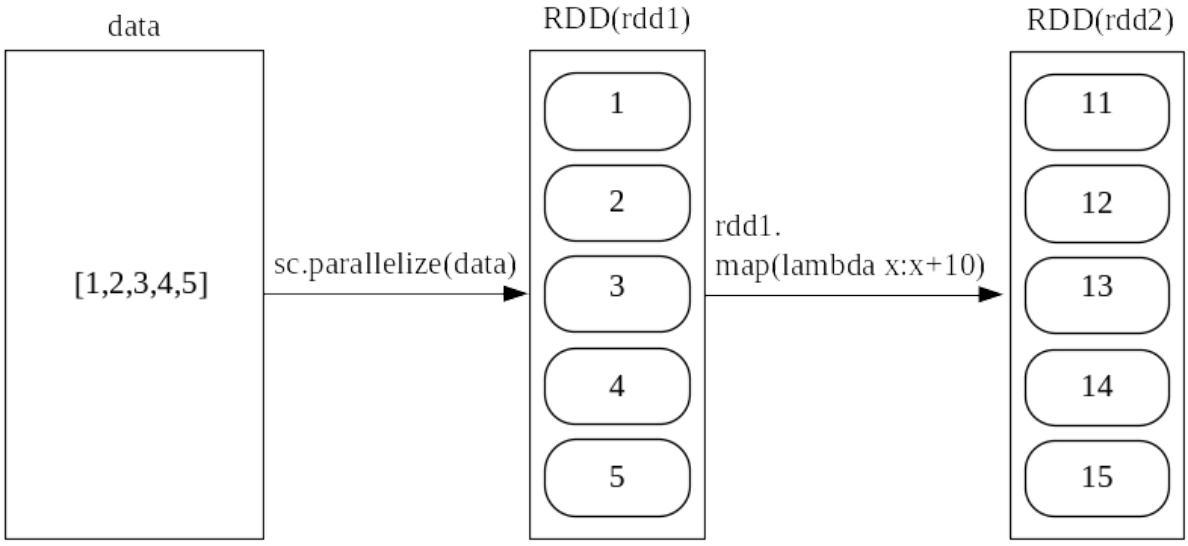
filter

```
>>>  lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>>  linesWithSpark = lines.filter(lambda line: "Spark" in line)
>>> linesWithSpark.foreach(print)
Spark is better
Spark is fast
```



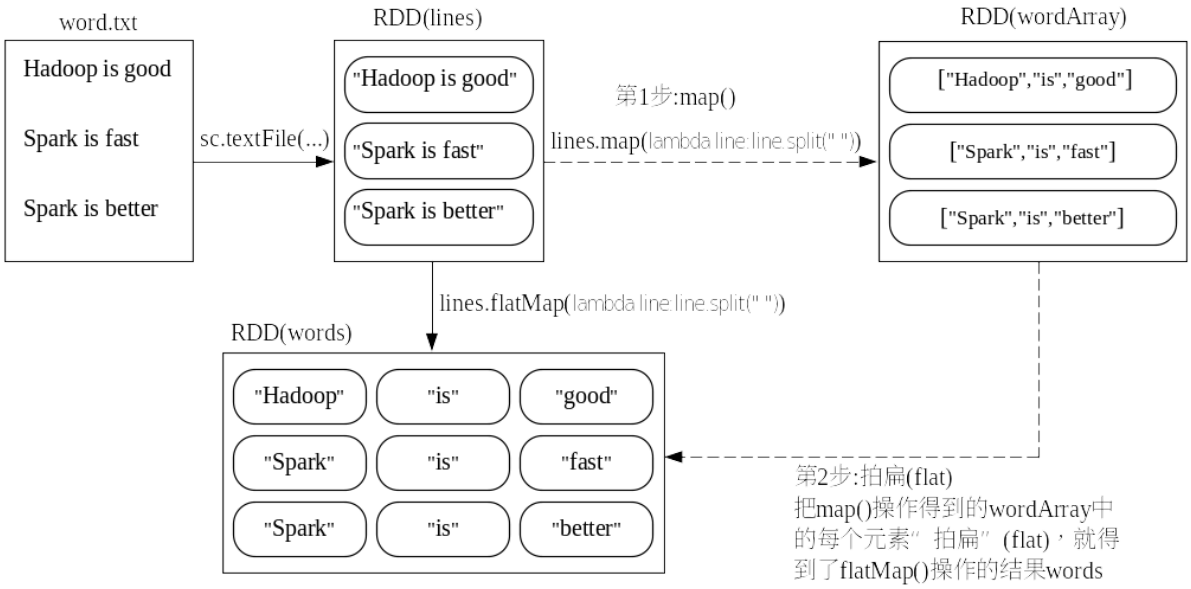
map

```
>>> data = [1,2,3,4,5]
>>> rdd1 = sc.parallelize(data)
>>> rdd2 = rdd1.map(lambda x:x+10)
>>> rdd2.foreach(print)
11
13
12
14
15
```



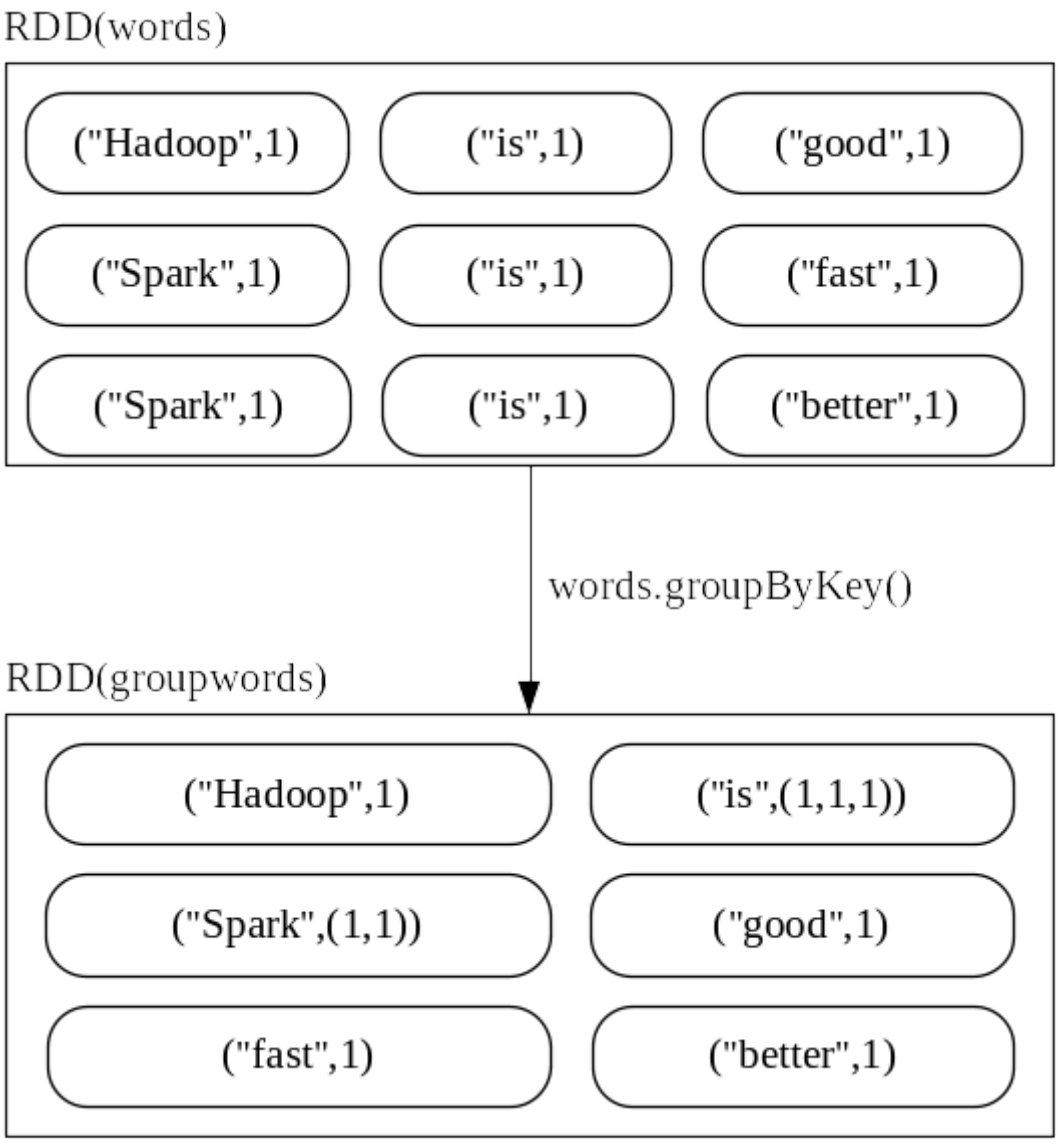
flatMap

```
>>> lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> words = lines.flatMap(lambda line:line.split(" "))
```



groupByKey

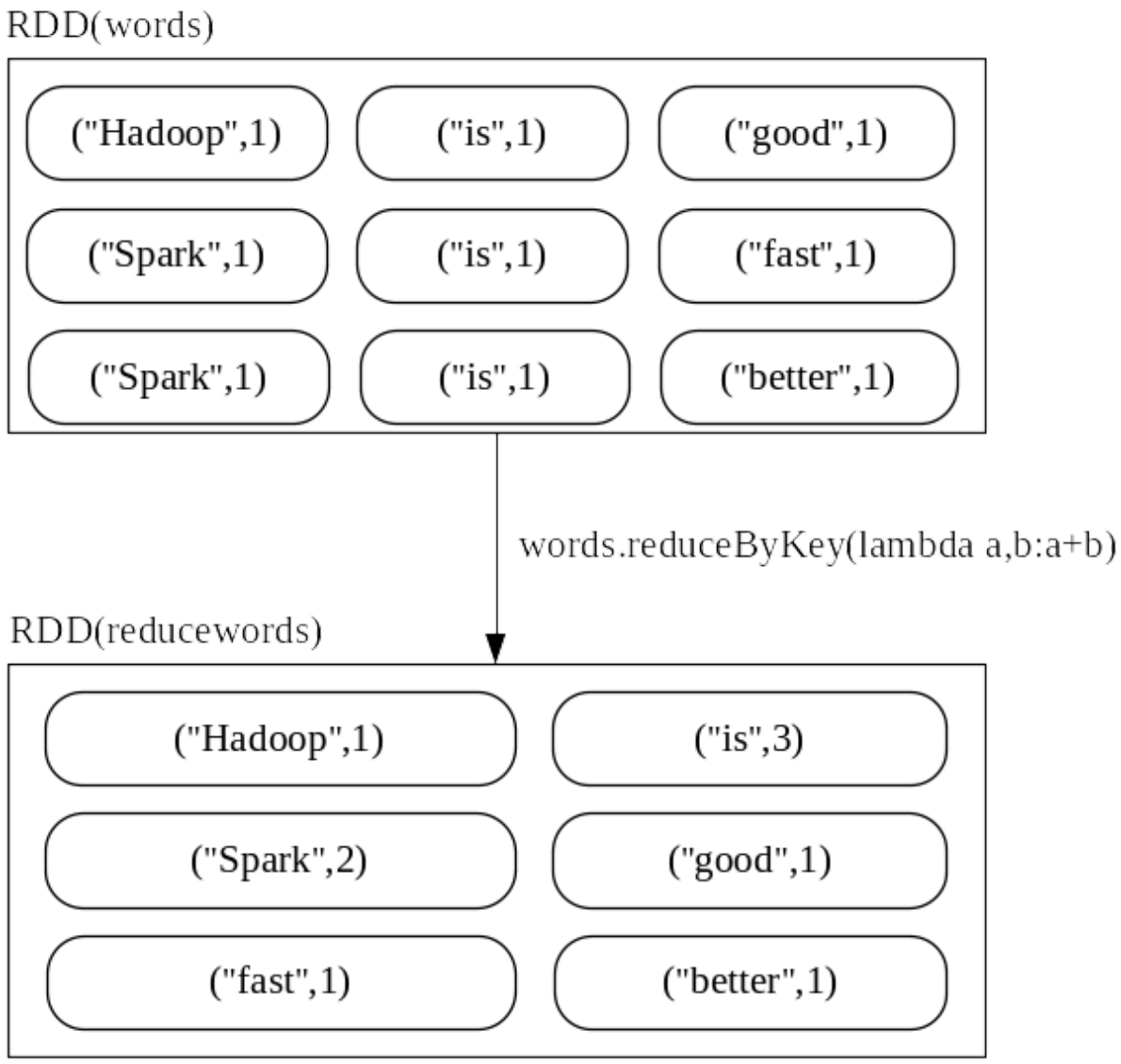
```
>>> words = sc.parallelize([("Hadoop",1),("is",1),("good",1), \
... ("Spark",1),("is",1),("fast",1),("Spark",1),("is",1),("better",1)])
>>> words1 = words.groupByKey()
>>> words1.foreach(print)
('Hadoop', <pyspark.resultiterable.ResultIterable object at 0x7fb210552c88>)
('better', <pyspark.resultiterable.ResultIterable object at 0x7fb210552e80>)
('fast', <pyspark.resultiterable.ResultIterable object at 0x7fb210552c88>)
('good', <pyspark.resultiterable.ResultIterable object at 0x7fb210552c88>)
('Spark', <pyspark.resultiterable.ResultIterable object at 0x7fb210552f98>)
('is', <pyspark.resultiterable.ResultIterable object at 0x7fb210552e10>)
```



reduceByKey

将RDD中元素两两传递给输入函数同时产生一个新的值，新产生的值与RDD中下一个元素再被传递给输入函数直到最后只有一个值为止。

```
>>> words = sc.parallelize([("Hadoop",1),("is",1),("good",1),("Spark",1), \
... ("is",1),("fast",1),("Spark",1),("is",1),("better",1)])
>>> words1 = words.reduceByKey(lambda a,b:a+b)
>>> words1.foreach(print)
 ('good', 1)
('Hadoop', 1)
('better', 1)
('Spark', 2)
('fast', 1)
('is', 3)
```



2.行动操作

行动操作是真正触发计算的地方。Spark程序执行到行动操作时，才会执行真正的计算，从文件中加载数据，完成一次又一次转换操作，最终，完成行动操作得到结果。

操作	含义
count()	返回数据集中的元素个数
collect()	以数组的形式返回数据集中的所有元素
first()	返回数据集中的第一个元素
take(n)	以数组的形式返回数据集中的前 n 个元素
reduce(func)	通过函数 func （输入两个参数并返回一个值）聚合数据集中的元素
foreach(func)	将数据集中的每个元素传递到函数 func 中运行

count

```
>>> rdd = sc.parallelize([1,2,3,4,5])
>>> rdd.count()
5
```

collect

```
>>> rdd.collect()
[1, 2, 3, 4, 5]
```

first

```
>>> rdd.first()
1
```

take

```
>>> rdd.take(3)
[1, 2, 3]
```

reduce

reduce将**RDD**中元素两两传递给输入函数，同时产生一个新的值，新产生的值与RDD中下一个元素再被传递给输入函数直到最后只有一个值为止。

```
>>> rdd.reduce(lambda a,b:a+b)
15
```

foreach

```
>>> rdd.foreach(lambda elem:print(elem))
1
2
3
4
5
```

RDD持久化

在Spark中，RDD采用惰性求值的机制，每次遇到行动操作，都会从头开始执行计算。每次调用行动操作，都会触发一次从头开始的计算。这对于迭代计算而言，代价是很大的，迭代计算经常需要多次重复使用同一组数据。

（1）可以通过持久化（缓存）机制避免这种重复计算的开销
（2）可以使用persist()方法对一个RDD标记为持久化
（3）之所以说“标记为持久化”，是因为出现persist()语句的地方，并不会马上计算生成RDD并把它持久化，而是要等到遇到第一个行动操作触发真正计算以后，才会把计算结果进行持久化.
（4）持久化后的RDD将会被保留在计算节点的内存中被后面的行动操作重复使。

Persist()

（1）persist(MEMORY_ONLY)：表示将RDD作为反序列化的对象存储于JVM中，如果内存不足，就要按照LRU原则替换缓存中的内容。
（2）persist(MEMORY_AND_DISK)表示将RDD作为反序列化的对象存储在JVM中，如果内存不足，超出的分区将会被存放在硬盘上。
（3）一般而言，使用cache()方法时，会调用persist(MEMORY_ONLY)
（4）可以使用unpersist()方法手动地把持久化的RDD从缓存中移除

RDD分区

RDD是弹性分布式数据集，通常RDD很大，会被分成很多个分区，分别保存在不同的节点上

分区的作用

（1）增加并行度

（2）减少通信开销

分区个数

在调用textFile()和parallelize()方法的时候手动指定分区个数即可，语法格式如下：
sc.textFile(path, partitionNum)
其中，path参数用于指定要加载的文件的地址，partitionNum参数用于指定分区个数。

```
>>> list = [1,2,3,4,5]
>>> rdd = sc.parallelize(list,2)  //设置两个分区
```

```
>>> rdd = sc.textFile(list,2)  //设置两个分区
```

重新分区

通过转换操作得到新 RDD 时，直接调用 repartition 方法即可。例如：

```
>>> data = sc.parallelize([1,2,3,4,5],2)
>>> len(data.glom().collect())  #显示data这个RDD的分区数量
2
>>> rdd = data.repartition(1)   #对data这个RDD进行重新分区
>>> len(rdd.glom().collect())   #显示rdd这个RDD的分区数量
1
```

分区方式

Spark提供了自带的HashPartitioner（哈希分区）与RangePartitioner（区域分区），能够满足大多数应用场景的需求。与此同时，Spark也支持自定义分区方式，即提供一个自定义的分区函数来控制RDD的分区方式，从而利用领域知识进一步减少通信开销。

partitionBy

```
from pyspark import SparkConf, SparkContext

def MyPartitioner(key):
    print("MyPartitioner is running")
    print('The key is %d' % key)
```

```
    return key%10

def main():
    print("The main function is running")
    conf  =  SparkConf().setMaster("local").setAppName("MyApp")
    sc  =  SparkContext(conf = conf)
    data = sc.parallelize(range(10),5)
    data.map(lambda x:(x,1)) \
           .partitionBy(10,MyPartitioner) \
           .map(lambda x:x[0]) \
           .saveAsTextFile("file:///usr/local/spark/mycode/rdd/partitioner")

if __name__ == '__main__':
       main()
```

键值对RDD

创建

通过map操作

```
>>> lines = sc.textFile("file:///usr/local/spark/mycode/pairrdd/word.txt")
>>> pairRDD = lines.flatMap(lambda line:line.split(" ")).map(lambda word:
(word,1))
>>> pairRDD.foreach(print)
('I', 1)
('love', 1)
('Hadoop', 1)
……
```

```
>>> list = ["Hadoop","Spark","Hive","Spark"]
>>> rdd = sc.parallelize(list)
>>> pairRDD = rdd.map(lambda word:(word,1))
>>> pairRDD.foreach(print)
(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)
```

键值对RDD转换操作

（1）reduceByKey(func)

```
>>> pairRDD = sc.parallelize([("Hadoop",1),("Spark",1),("Hive",1),("Spark",1)])
>>> pairRDD.reduceByKey(lambda a,b:a+b).foreach(print)
('Spark', 2)
('Hive', 1)
('Hadoop', 1)
```

reduceByKey用于对每个key对应的多个value进行merge操作，最重要的是它能够在本地先进行merge操作，并且merge操作可以通过函数自定义。

（2）groupByKey()

```
>>> list = [("spark",1),("spark",2),("hadoop",3),("hadoop",5)]
>>> pairRDD = sc.parallelize(list)
>>> pairRDD.groupByKey()
PythonRDD[27] at RDD at PythonRDD.scala:48
>>> pairRDD.groupByKey().foreach(print)
('hadoop', <pyspark.resultiterable.ResultIterable object at 0x7f2c1093ecf8>)
('spark', <pyspark.resultiterable.ResultIterable object at 0x7f2c1093ecf8>)
```

groupByKey也是对每个key进行操作，但只生成一个sequence，groupByKey本身不能自定义函数，需要先用groupByKey生成RDD，然后才能对此RDD通过map进行自定义函数操作。

（3）keys

```
>>> list = [("Hadoop",1),("Spark",1),("Hive",1),("Spark",1)]
>>> pairRDD = sc.parallelize(list)
>>> pairRDD.keys().foreach(print)
Hadoop
Spark
Hive
Spark
```

（4）values

```
>>> list = [("Hadoop",1),("Spark",1),("Hive",1),("Spark",1)]
>>> pairRDD = sc.parallelize(list)
>>> pairRDD.values().foreach(print)
1
1
1
1
```

（5）sortByKey()

```
>>> list = [("Hadoop",1),("Spark",1),("Hive",1),("Spark",1)]
>>> pairRDD = sc.parallelize(list)
>>> pairRDD.foreach(print)
('Hadoop', 1)
('Spark', 1)
('Hive', 1)
('Spark', 1)
>>> pairRDD.sortByKey().foreach(print)
('Hadoop', 1)
('Hive', 1)
('Spark', 1)
('Spark', 1)
```

(6)sortBy

```
>>> d1 = sc.parallelize([("c",8),("b",25),("c",17),("a",42), \
... ("b",4),("d",9),("e",17),("c",2),("f",29),("g",21),("b",9)])
>>> d1.reduceByKey(lambda a,b:a+b).sortBy(lambda x:x,False).collect()
[('g', 21), ('f', 29), ('e', 17), ('d', 9), ('c', 27), ('b', 38), ('a', 42)]
>>> d1.reduceByKey(lambda a,b:a+b).sortBy(lambda x:x[0],False).collect()
[('g', 21), ('f', 29), ('e', 17), ('d', 9), ('c', 27), ('b', 38), ('a', 42)]
>>> d1.reduceByKey(lambda a,b:a+b).sortBy(lambda x:x[1],False).collect()
[('a', 42), ('b', 38), ('f', 29), ('c', 27), ('g', 21), ('e', 17), ('d', 9)]
```

(7) mapValues(func)

```
>>> list = [("Hadoop",1),("Spark",1),("Hive",1),("Spark",1)]
>>> pairRDD = sc.parallelize(list)
>>> pairRDD1 = pairRDD.mapValues(lambda x:x+1)
>>> pairRDD1.foreach(print)
('Hadoop', 2)
('Spark', 2)
('Hive', 2)
('Spark', 2)
```

(8) join

```
>>> pairRDD1 = sc. \
... parallelize([("spark",1),("spark",2),("hadoop",3),("hadoop",5)])
>>> pairRDD2 = sc.parallelize([("spark","fast")])
>>> pairRDD3 = pairRDD1.join(pairRDD2)
>>> pairRDD3.foreach(print)
('spark', (1, 'fast'))
('spark', (2, 'fast'))
```

(9) combineByKey

todo

RDD写入

```
>>> textFile = sc.textFile("word.txt")
>>> textFile.saveAsTextFile("writeback")
```

RDD读写HBase

读取

```
#!/usr/bin/env python3
from pyspark import SparkConf, SparkContext
conf = SparkConf().setMaster("local").setAppName("ReadHBase")
sc = SparkContext(conf = conf)
host = 'localhost'
table = 'student'
conf = {"hbase.zookeeper.quorum": host, "hbase.mapreduce.inputtable": table}
keyConv =
"org.apache.spark.examples.pythonconverters.ImmutableBytesWritableToStringConvert
er"
```

```
valueConv =
"org.apache.spark.examples.pythonconverters.HBaseResultToStringConverter"

hbase_rdd =
sc.newAPIHadoopRDD("org.apache.hadoop.hbase.mapreduce.TableInputFormat","org.apac
he.hadoop.hbase.io.ImmutableBytesWritable","org.apache.hadoop.hbase.client.Result
",keyConverter=keyConv,valueConverter=valueConv,conf=conf)

count = hbase_rdd.count()

hbase_rdd.cache()

output = hbase_rdd.collect()

for (k, v) in output:
        print (k, v)
```

写入

```
#!/usr/bin/env python3

from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("ReadHBase")
sc = SparkContext(conf = conf)
host = 'localhost'
table = 'student'
keyConv =
"org.apache.spark.examples.pythonconverters.StringToImmutableBytesWritableConvert
er"
valueConv = "org.apache.spark.examples.pythonconverters.StringListToPutConverter"
conf = {"hbase.zookeeper.quorum": host,"hbase.mapred.outputtable":
table,"mapreduce.outputformat.class":
"org.apache.hadoop.hbase.mapreduce.TableOutputFormat","mapreduce.job.output.key.c
lass":
"org.apache.hadoop.hbase.io.ImmutableBytesWritable","mapreduce.job.output.value.c
lass": "org.apache.hadoop.io.Writable"}
rawData =
['3,info,name,Rongcheng','3,info,gender,M','3,info,age,26','4,info,name,Guanhua',
'4,info,gender,M','4,info,age,27']
sc.parallelize(rawData).map(lambda x:
(x[0],x.split(','))).saveAsNewAPIHadoopDataset(conf=conf,keyConverter=keyConv,val
ueConverter=valueConv)
```

SparkSQL

1.DataFrame创建

```
from pyspark import SparkContext,SparkConf
from pyspark.sql import SparkSession
spark = SparkSession.builder.config(conf = SparkConf()).getOrCreate()
df = spark.read.text("people.txt")
df = spark.read.json("people.json")
df = spark.read.parquet(“people.parquet”)
```

```
spark.read.format("text").load("people.txt")
spark.read.format("json").load("people.json")
spark.read.format("parquet").load("people.parquet")
```

2.DataFrame保存

```
#可以使用spark.write操作，把一个DataFrame保存成不同格式的文件，例如，把一个名称为df的
DataFrame保存到不同格式文件中，方法如下：
df.write.text("people.txt")
df.write.json("people.json“)
df.write.parquet("people.parquet“)
```

```
#或者也可以使用如下格式的语句：
df.write.format("text").save("people.txt")
df.write.format("json").save("people.json")
df.write.format ("parquet").save("people.parquet")
```

例如：下面从示例文件people.json中创建一个DataFrame，名称为peopleDF，把peopleDF保存到另外一个JSON文件中，然后，再从peopleDF中选取一个列（即name列），把该列数据保存到一个文本文件中。

```
>>> peopleDF = spark.read.format("json").\
... load("file:///usr/local/spark/examples/src/main/resources/people.json")
>>> peopleDF.select("name", "age").write.format("json").\
... save("file:///usr/local/spark/mycode/sparksql/newpeople.json")
>>> peopleDF.select("name").write.format("text").\
... save("file:///usr/local/spark/mycode/sparksql/newpeople.txt")
```

3.DataFrame操作

```
>>> df=spark.read.json(“people.json”)
>>> df.printSchema()
root
  |-- age:long(nullable=true)
  |-- name:string (nullable = true)


>>>df.select(df["name"],df["age"]+1).show()
>>>+-------+---------+
   |   name|(age + 1)|
   +-------+---------+
   |Micheal|     null|
   |   Andy|       31|
   | Justin|       20|
   +-------+---------+


>>> df.filter(df["age"]>20).show()
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+
```

```
>>> df.groupBy("age").count().show()
+----+-----+
| age|count|
+----|-----+
|  19|    1|
|null|    1|
|  30|    1|
+----+-----+

>>>df.sort(df["age"].desc(),df["name"].asc()).show()
+----+--------+
| age|    name|
+----+--------+
|  30|    Andy|
|  19|  Justin|
|null| Micheal|
+----+--------+
```

4.RDD与DataFrame转换

1.反射推断机制

```
>>> from pyspark.sql import Row
>>> people = spark.sparkContext.\
... textFile("file:///usr/local/spark/examples/src/main/resources/people.txt").\
... map(lambda line: line.split(",")).\
... map(lambda p: Row(name=p[0], age=int(p[1])))
>>> schemaPeople = spark.createDataFrame(people)
#必须注册为临时表才能供下面的查询使用
>>> schemaPeople.createOrReplaceTempView("people")
>>> personsDF = spark.sql("select name,age from people where age > 20")
#DataFrame中的每个元素都是一行记录，包含name和age两个字段，分别用p.name和p.age来获取值
>>> personsRDD=personsDF.rdd.map(lambda p:"Name: "+p.name+ ","+"Age:
"+str(p.age))
>>> personsRDD.foreach(print)
Name: Michael,Age: 29
Name: Andy,Age: 30
```

2.编程方式

```
>>> from pyspark.sql.types import *
>>> from pyspark.sql import Row
#下面生成“表头”
>>> schemaString = "name age"
>>> fields = [StructField(field_name, StringType(), True) for field_name in
schemaString.split(" ")]
>>> schema = StructType(fields)
#下面生成“表中的记录”
>>> lines = spark.sparkContext.\
... textFile("file:///usr/local/spark/examples/src/main/resources/people.txt")
>>> parts = lines.map(lambda x: x.split(","))
>>> people = parts.map(lambda p: Row(p[0], p[1].strip()))
#下面把“表头”和“表中的记录”拼装在一起
>>> schemaPeople = spark.createDataFrame(people, schema)
```

5.Spark操作Mysql数据库

读取

```
>>>    jdbcDF = spark.read \
        .format("jdbc") \
        .option("driver","com.mysql.jdbc.Driver") \
        .option("url", "jdbc:mysql://localhost:3306/spark") \
        .option("dbtable", "student") \
        .option("user", "root") \
        .option("password", "123456") \
        .load()
>>>  jdbcDF.show()
+---+--------+------+---+
| id|    name|gender|age|
+---+--------+------+---+
|  1| Xueqian|     F| 23|
|  2|Weiliang|     M| 24|
+---+--------+------+---+
```

插入

```
#!/usr/bin/env python3

from pyspark.sql import Row
from pyspark.sql.types import *
from pyspark import SparkContext,SparkConf
from pyspark.sql import SparkSession

spark = SparkSession.builder.config(conf = SparkConf()).getOrCreate()

#下面设置模式信息
schema = StructType([StructField("id", IntegerType(), True), \
StructField("name", StringType(), True), \
StructField("gender", StringType(), True), \
StructField("age", IntegerType(), True)])
#下面设置两条数据，表示两个学生的信息
studentRDD = spark \
.sparkContext \
.parallelize(["3 Rongcheng M 26","4 Guanhua M 27"]) \
.map(lambda x:x.split(" "))

#下面创建Row对象，每个Row对象都是rowRDD中的一行
rowRDD = studentRDD.map(lambda p:Row(int(p[0].strip()), p[1].strip(),
p[2].strip(), int(p[3].strip())))

#建立起Row对象和模式之间的对应关系，也就是把数据和模式对应起来
studentDF = spark.createDataFrame(rowRDD, schema)

#写入数据库
prop = {}
prop['user'] = 'root'
prop['password'] = '123456'
prop['driver'] = "com.mysql.jdbc.Driver"
studentDF.write.jdbc("jdbc:mysql://localhost:3306/spark",'student','append',
prop)
```

SparkStreaming

1.通过创建输入DStream来定义输入源
2.通过对DStream应用转换操作和输出操作来定义流计算
3.用streamingContext.start()来开始接收数据和处理流程
4.通过streamingContext.awaitTermination()方法来等待处理结束（手动结束或因为错误而结束）
5.可以通过streamingContext.stop()来手动结束流计算进程

（1）如果要运行一个Spark Streaming程序，就需要首先生成一个StreamingContext对象，它是Spark Streaming程序的主入口。
（2）可以从一个SparkConf对象创建一个StreamingContext对象。
（3）在pyspark中的创建方法：进入pyspark以后，就已经获得了一个默认的SparkConext对象，也就是sc。因此，可以采用如下方式来创建StreamingContext对象：

```
>>> from pyspark.streaming import StreamingContext
>>> ssc = StreamingContext(sc, 1)
```

（4）如果是编写一个独立的Spark Streaming程序，而不是在pyspark中运行，则需要通过如下方式创建StreamingContext对象：

```
from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext
conf = SparkConf()
conf.setAppName('TestDStream')
conf.setMaster('local[2]')
sc = SparkContext(conf = conf)
ssc = StreamingContext(sc, 1)
```

1.文件流

```
#!/usr/bin/env python3

from pyspark import SparkContext, SparkConf
from pyspark.streaming import StreamingContext

conf = SparkConf()
conf.setAppName('TestDStream')
conf.setMaster('local[2]')
sc = SparkContext(conf = conf)
ssc = StreamingContext(sc, 10)
lines = ssc.textFileStream('file:///usr/local/spark/mycode/streaming/logfile')
words = lines.flatMap(lambda line: line.split(' '))
wordCounts = words.map(lambda x : (x,1)).reduceByKey(lambda a,b:a+b)
wordCounts.pprint()
ssc.start()
ssc.awaitTermination()
```

2.套接字流

```
#!/usr/bin/env python3
```

```
from __future__ import print_function
import sys
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: NetworkWordCount.py <hostname> <port>", file=sys.stderr)
        exit(-1)
    sc = SparkContext(appName="PythonStreamingNetworkWordCount")
    ssc = StreamingContext(sc, 1)
    lines = ssc.socketTextStream(sys.argv[1], int(sys.argv[2]))
    counts = lines.flatMap(lambda line: line.split(" ")) \
                  .map(lambda word: (word, 1)) \
                  .reduceByKey(lambda a, b: a+b)
    counts.pprint()
    ssc.start()
    ssc.awaitTermination()
```

新打开一个窗口作为nc窗口，启动nc程序：

```
nc -lk 9999
```

打开流计算程序

```
spark-submit NetworkWordCount.py localhost 9999
```

会显示：

```
-------------------------------------------
Time: 2018-12-24 11:30:26
-------------------------------------------
('Spark', 1)
('love', 1)
('I', 1)
(spark,1)
```

nc是网络接收发包程序，可以自定义服务端

```
#!/usr/bin/env python3
import socket
# 生成socket对象
server = socket.socket()
# 绑定ip和端口
server.bind(('localhost', 9999))
# 监听绑定的端口
server.listen(1)
while 1:
    # 为了方便识别，打印一个“我在等待”
    print("I'm waiting the connect...")
    # 这里用两个值接受，因为连接上之后使用的是客户端发来请求的这个实例
    # 所以下面的传输要使用conn实例操作
    conn,addr = server.accept()
    # 打印连接成功
```

```
    print("Connect success! Connection is from %s " % addr[0])
    # 打印正在发送数据
    print('Sending data...')
    conn.send('I love hadoop I love spark hadoop is good spark is fast'.encode())
    conn.close()
    print('Connection is broken.‘)
```

3.RDD队列流

TODO

4.输出DStream到mysql

创建表

```
mysql> create table wordcount (word char(20), count int(4));
```

写入

```
#!/usr/bin/env python3

from __future__ import print_function
import sys
import pymysql
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: NetworkWordCountStateful <hostname> <port>",
file=sys.stderr)
        exit(-1)
    sc = SparkContext(appName="PythonStreamingStatefulNetworkWordCount")
    ssc = StreamingContext(sc, 1)
    ssc.checkpoint("file:///usr/local/spark/mycode/streaming/stateful")
    # RDD with initial state (key, value) pairs
    initialStateRDD = sc.parallelize([(u'hello', 1), (u'world', 1)])

    def updateFunc(new_values, last_sum):
        return sum(new_values) + (last_sum or 0)

    lines = ssc.socketTextStream(sys.argv[1], int(sys.argv[2]))
    running_counts = lines.flatMap(lambda line: line.split(" "))\
                          .map(lambda word: (word, 1))\
                          .updateStateByKey(updateFunc,
initialRDD=initialStateRDD)
    running_counts.pprint()

    def dbfunc(records):
        db = pymysql.connect("localhost","root","123456","spark")
        cursor = db.cursor()
        def doinsert(p):
            sql = "insert into wordcount(word,count) values ('%s', '%s')" %
(str(p[0]), str(p[1]))
            try:
                cursor.execute(sql)
```

```
                db.commit()
            except:
                db.rollback()
        for item in records:
            doinsert(item)
def func(rdd):
        repartitionedRDD = rdd.repartition(3)
        repartitionedRDD.foreachPartition(dbfunc)

    running_counts.foreachRDD(func)
    ssc.start()
    ssc.awaitTermination()
```