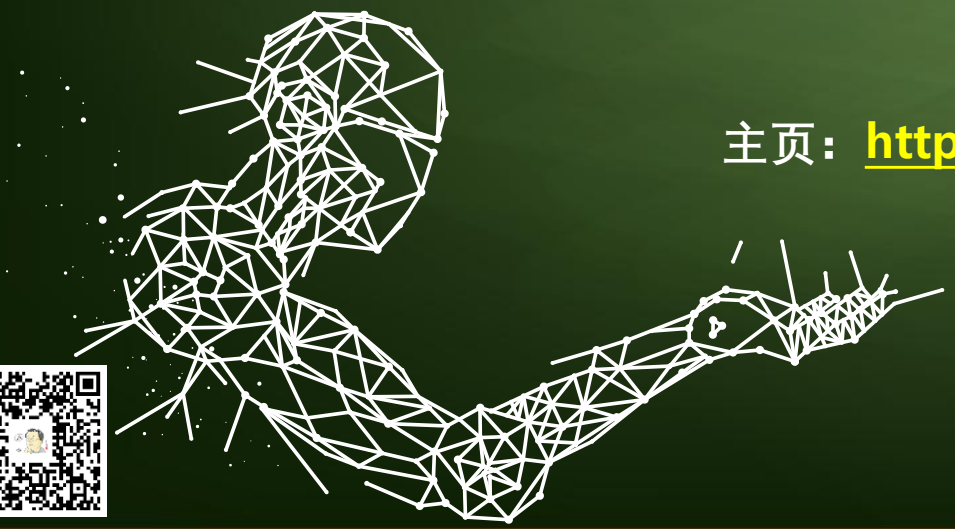
05

第5章 RDD编程
(RDD Programming)

授课教师：[李阳源]

ywlee1212@gmail.com

主页：https://sites.google.com/view/dalguri



SL-01:提纲

5.1 RDD编程基础(RDD Programming Basics)

5.2 键值对RDD(Key-Value Pair RDD)

5.3 数据读写(Data Read/Write)

5.4 综合案例(Comprehensive Case Study)

SL-02: RDD编程基础

5.1.1 RDD创建(RDD Creation)
5.1.2 RDD操作(RDD Operations)
5.1.3 持久化(Persistence)
5.1.4 分区(Partitioning)
5.1.5 一个综合实例(A Comprehensive Example)

SL-03: 5.1.1 RDD创建

- 从文件系统中加载数据创建RDD **(Creating RDD by loading data from file system)**
- 通过并行集合(列表)创建RDD **(Creating RDD through parallel collections)**

SL-04: 5.1.1 RDD创建



1. 从文件系统中加载数据创建RDD

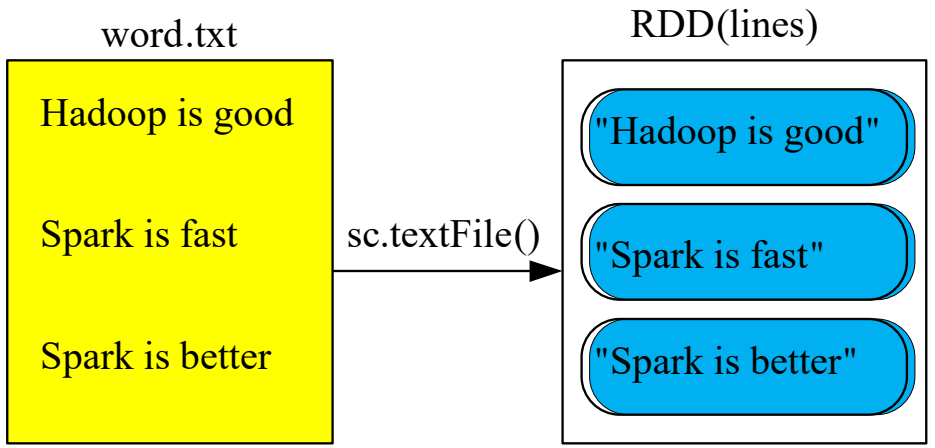
- Spark采用textFile()方法来从文件系统中加载数据创建RDD
- 该方法把文件的URI作为参数，
  这个URI可以是：
  - 本地文件系统的地址**(Local file system address)**
  - 或者是分布式文件系统HDFS的地址**(HDFS distributed file syste address)**
  - 或者是Amazon S3的地址**(Amazon S3 address)**等等



SL-05: 5.1.1 RDD创建

1) 从本地文件系统中加载数据创建RDD

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



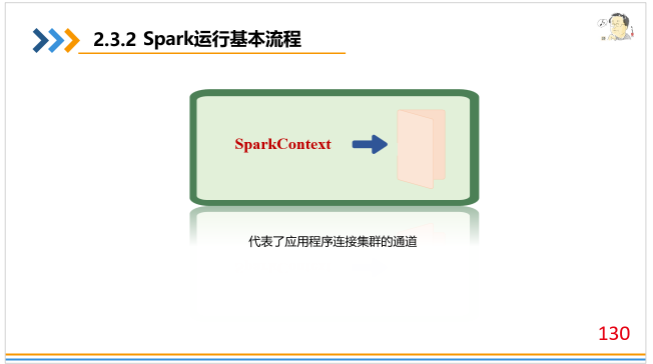


图5-1 从文件中加载数据生成RDD



表示一个RDD元素

SL06: 5.1.1 RDD创建



- 在上述语句中，使用了Spark提供的SparkContext对象，名称为sc，这是pyspark启动的时候自动创建的，在交互式编程环境中可以直接使用。

- 可是我们在独立应用程序，则可以通过如下语句生成SparkContext对象(ex:Pycharm)

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

SL07: 5.1.1 RDD创建



eSoftner

如果要在PyCharm中调试程序，可以编写如下代码：

```
# coding:utf8
from pyspark import SparkConf, SparkContext
if __name__ == '__main__':
    conf = SparkConf().setMaster("local[*]").setAppName("My App")
    sc = SparkContext(conf = conf)
    lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
    lines.foreach(print)
    sc.stop()
```

SL-08: 5.1.1 RDD创建

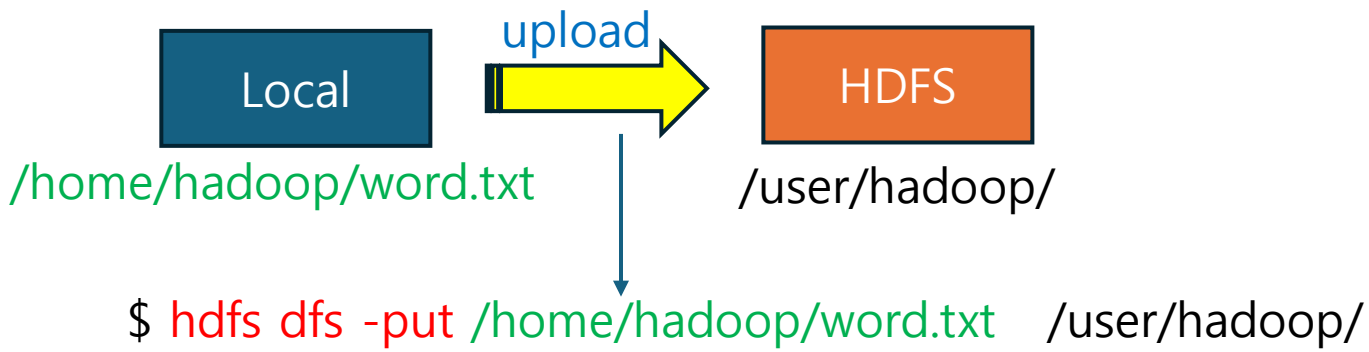


2) 从分布式文件系统HDFS中加载数据

```
>>> lines = sc.textFile("hdfs://hadoop01:9000/user/hadoop/word.txt")
>>> lines = sc.textFile("/user/hadoop/word.txt")
>>> lines = sc.textFile("word.txt")
```

三条语句是完全等价的，可以使用其中任意一种方式

```
>>> lines.foreach(print)
Hadoop is good
Spark is fast
Spark is better
```



SL-09: 5.1.1 RDD创建



2. 通过并行集合（列表）创建RDD

可以调用SparkContext的parallelize方法，在Driver中一个已经存在的集合（列表）上创建。

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

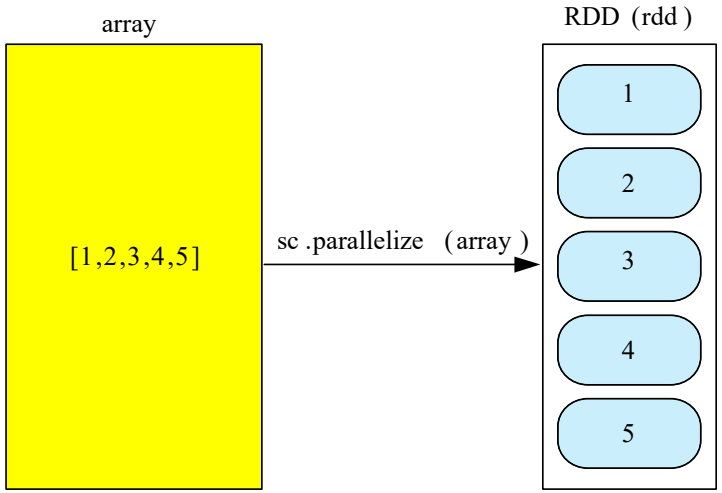


图5-2 从数组创建RDD示意图

SL-10: 5.1.2 RDD操作

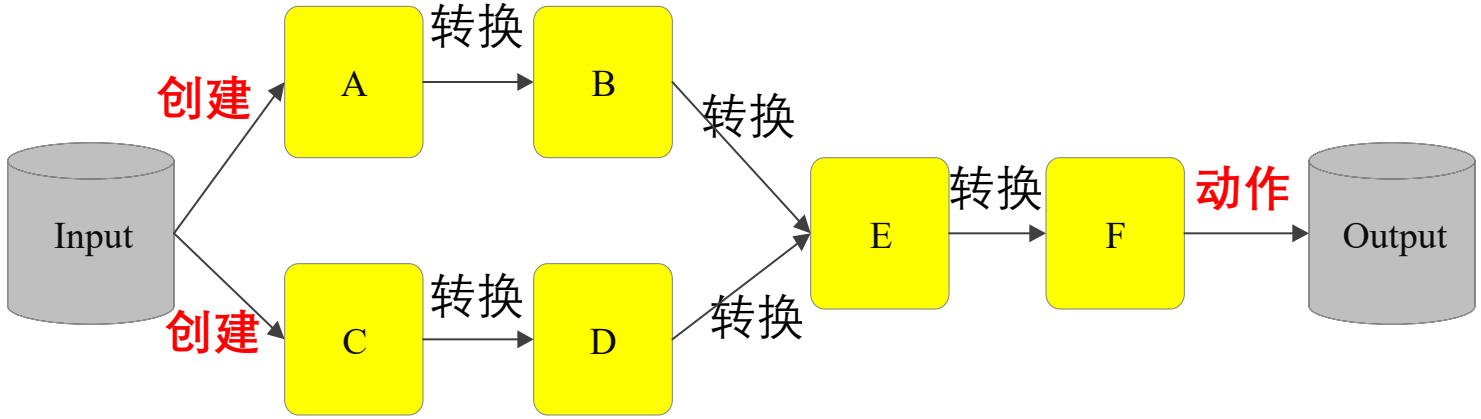
- 转换操作(transformation)
- 行动操作(action)
- 惰性机制(lazy evaluation)

SL-11: 5.1.2 RDD操作



1. 转换操作

- 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
- 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



SL-12: 5.1.2 RDD操作



1. 转换操作(1)

表5-1 常用的RDD转换操作API

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
mapPartitions(func)	对RDD中的每个分区的迭代器对象进行操作，将每个迭代器对象传递到函数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
groupBy(func)	对数据集按照指定的规则进行分组
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(func)	筛选出满足函数func的元素，并返回一个新的数据集

SL-13: 5.1.2 RDD操作



1. 转换操作(2)

表5-1 常用的RDD转换操作API （续表）

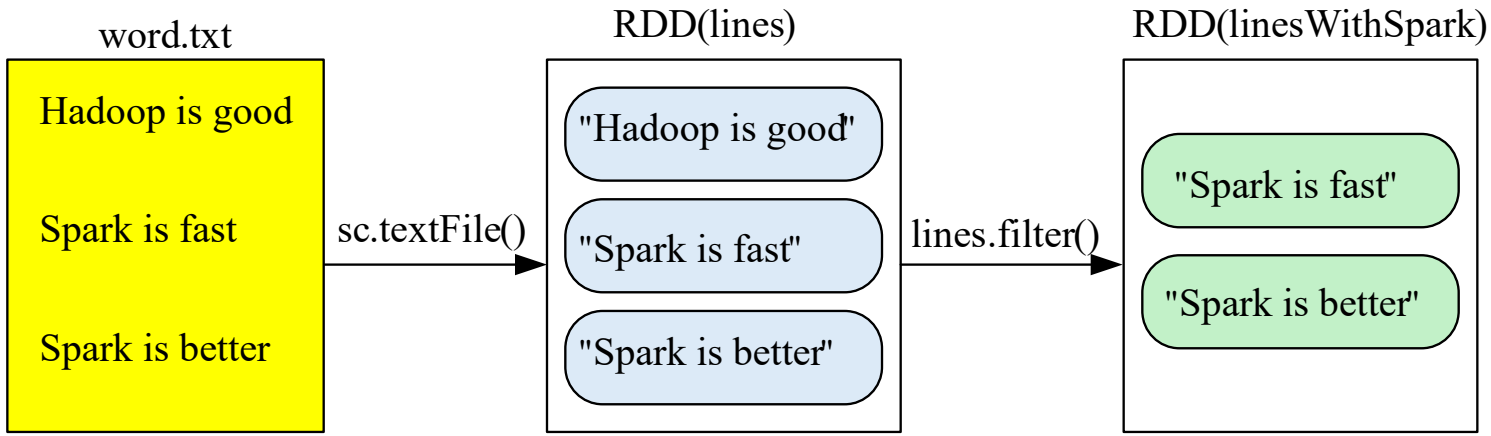
操作	含义
sortBy(func)	根据一定的规则对数据进行排序
distinct	对RDD内部的元素进行去重，并把去重后的元素放到新的RDD中
union	对两个RDD进行并集运算，并返回新的RDD
intersection	对两个RDD进行交集运算，并返回新的RDD
subtract	对两个RDD进行差集运算，并返回新的RDD
zip	把两个RDD中的元素以键值对的形式进行合并

SL-14: 5.1.2 RDD操作



- filter(func)：筛选出满足函数func的元素，并返回一个新的数据集

```
>>> 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
```



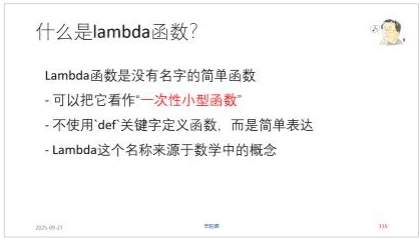


图5-3 filter()操作实例执行过程示意图

SL-15: 5.1.2 RDD操作



- map(func)： map(func)操作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

```
>>> 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
```

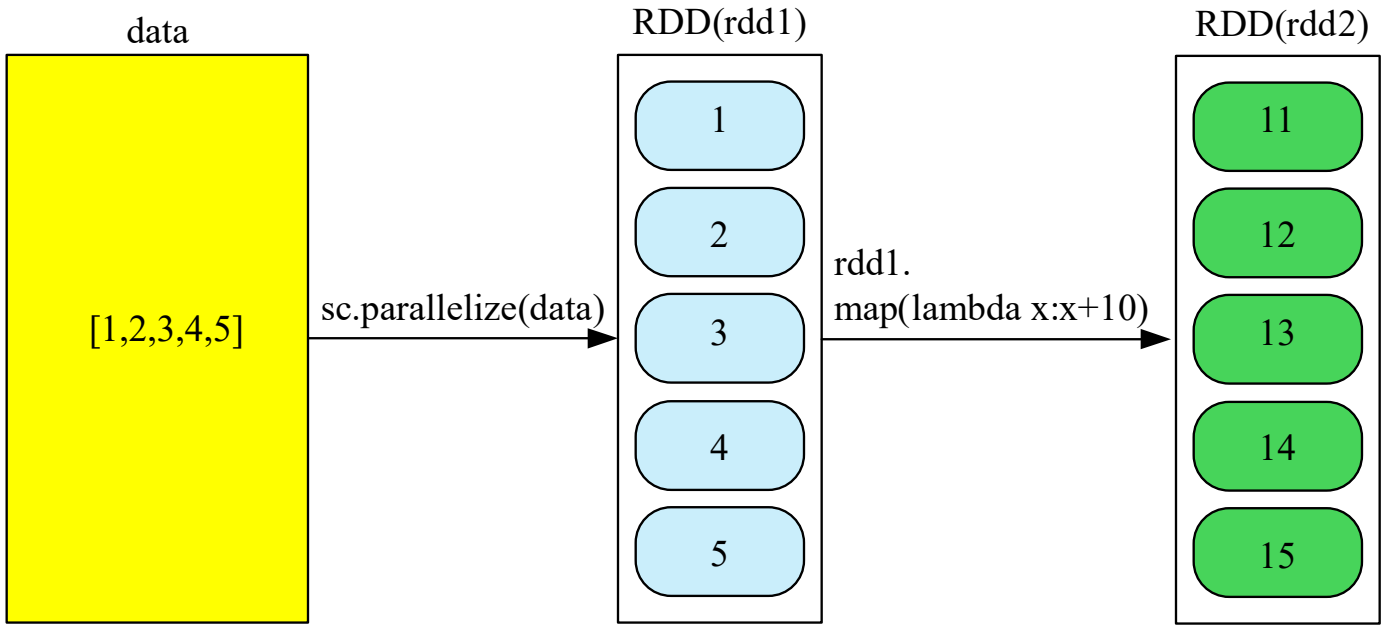


图5-4 map()操作实例执行过程示意图

SL-16: 5.1.2 RDD操作



- map(func) :另外一个实例

```
>>> lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> words = lines.map(lambda line:line.split(" "))
>>> words.foreach(print)
['Hadoop', 'is', 'good']
['Spark', 'is', 'fast']
['Spark', 'is', 'better']
```

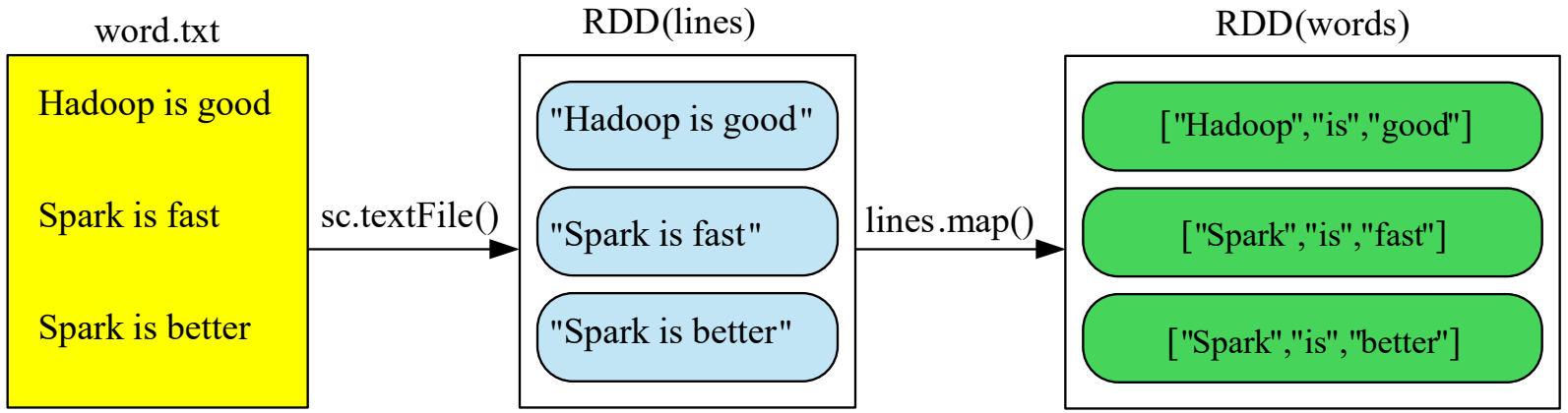


图5-5 map()操作实例执行过程示意图

SL-17: 5.1.2 RDD操作



在上面的例子中，我们传递给map()算子的是一个Labmda表达式，
实际上，也可以单独定义一个方法传递给map()算子，
实现代码如下：

```
# coding:utf8
from pyspark import SparkConf, SparkContext
if __name__ == '__main__':
    conf = SparkConf().setMaster("local[*]").setAppName("My App")
    sc = SparkContext(conf = conf)
    data = [1,2,3,4,5]
    rdd1 = sc.parallelize(data)

    def add(data):
        return data + 10

    print(rdd1.map(add).collect())
    sc.stop()
```

SL-18: 5.1.2 RDD操作



- mapPartitions(func)：

map算子是对RDD中的每一个元素进行操作，

mapPartitions算子则是对RDD中的每个分区的迭代器进行操作。

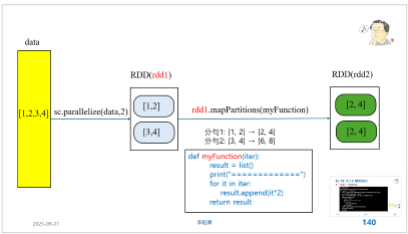
- ✓ mapPartitions的优势是可以提高性能，比如，对一个含有100条日志数据的分区进行操作，使用map时，函数要执行100次计算。
- ✓ 使用mapPartitions操作之后，一个任务会一次性接收一个分区内的所有数据，统一执行一次计算。
- ✓ 如果map执行的过程中还需要创建对象，比如创建Redis连接或者JDBC连接等，那么，map需要为每个元素创建一个连接，而mapPartitions只需要为每个分区创建一个连接，这样就大大降低了数据库连接的开销。
- ✓ mapPartitons的缺点是，如果一个分区有很多数据，一次函数处理可能会导致内存溢出，而普通的map一般不会导致内存溢出。

SL-19: 5.1.2 RDD操作



- 下面是一个具体实例:

```
# coding:utf8
from pyspark import SparkConf, SparkContext
if __name__ == '__main__':
    conf = SparkConf().setMaster("local[*]").setAppName("My App")
    sc = SparkContext(conf = conf)
    rdd1 = sc.parallelize([1, 2, 3, 4], 2)  # 设置两个分区
    def myFunction(iter):
        result = list()
        print("==============")
        for it in iter:
            result.append(it*2)
        return result
    rdd2 = rdd1.mapPartitions(myFunction)
    print(rdd2.collect())
    sc.stop()
```



SL-20: 5.1.2 RDD操作



- 程序的执行结果如下：

```
==============
==============
[2, 4, 6, 8]
```



- 为什么这里会出现两次 "==========" 呢？

从程序执行结果可以看出，由于在程序中设置了两个分区，所以，myFunction被调用了2次，而不是4次。

SL-21: 5.1.2 RDD操作



- flatMap(func)

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

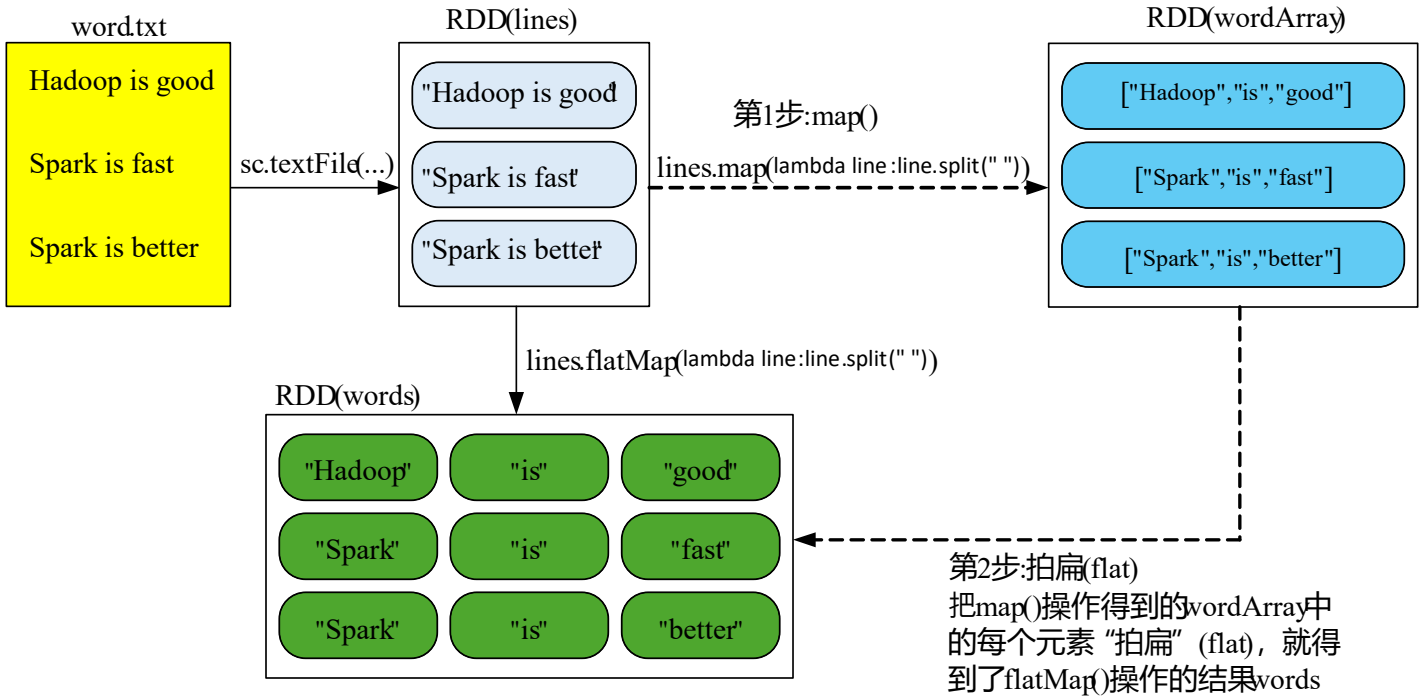


图5-6 flatMap()操作实例执行过程示意图

SL-22: 5.1.2 RDD操作



- groupBy()：groupBy()用于对数据集按照指定规则进行分组。下面给出一个简单实例，代码如下

```
>>> rdd1 = sc.parallelize(["Hadoop", "Spark", "Storm", "Flink", "Flume"])
>>> rdd2 = rdd1.groupBy(lambda x:x[0])
>>> rdd2.collect()
[('S', <pyspark.resultiterable.ResultIterable object at 0x7fda9f0a6b20>), ('H', <pyspark.resultiterable.ResultIterable object at 0x7fda9f08b310>), ('F', <pyspark.resultiterable.ResultIterable object at 0x7fda9f08b4c0>)]
```

- 上面代码的含义是，根据单词的首字母进行分组，首字母相同的单词会被分到同一个组中。

SL-23: 5.1.2 RDD操作



- groupByKey()： 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

```
>>> 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>)
```

SL-24: 5.1.2 RDD操作



- groupByKey()

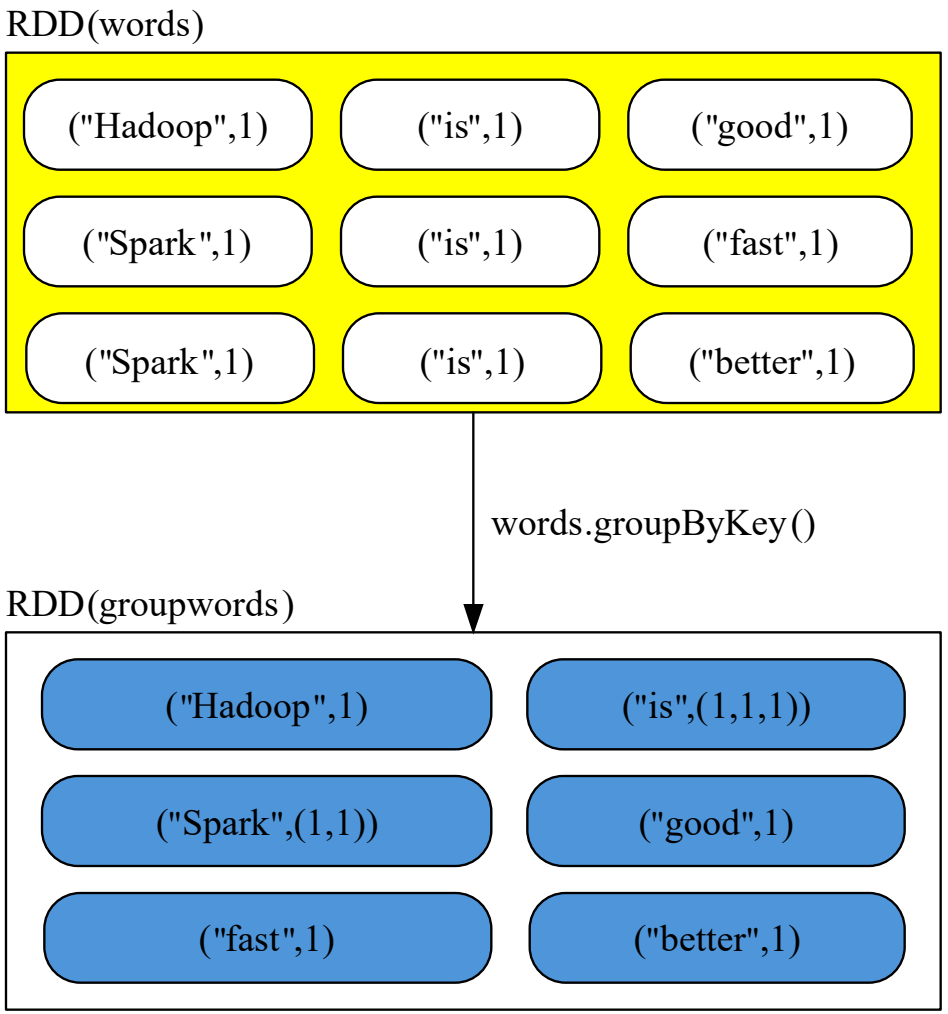


图5-7　groupByKey()操作实例执行过程示意图

SL-25: 5.1.2 RDD操作



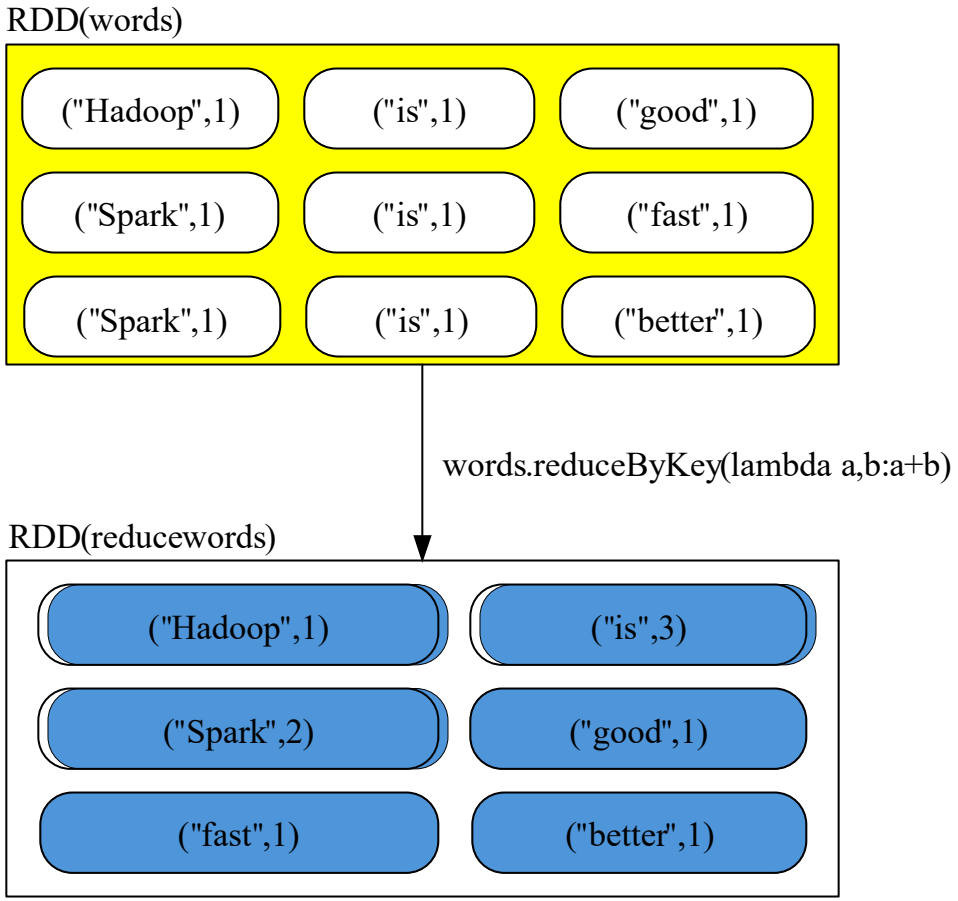
- reduceByKey(func): reduceByKey(func)应用于(K,V)键值对的数据集时，返回一个新的(K, V)形式的数据集，其中的每个值是将每个key传递到函数func中进行聚合后得到的结果

```
>>> 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)
```

SL-26: 5.1.2 RDD操作



- reduceByKey(func)



由于先在本地合并，因此减少了网络传输量
("is", 1) + ("is", 1) = ("is", 2) # 分区内处理
("is", 2) + ("is", 1) = ("is", 3) # 最终合计

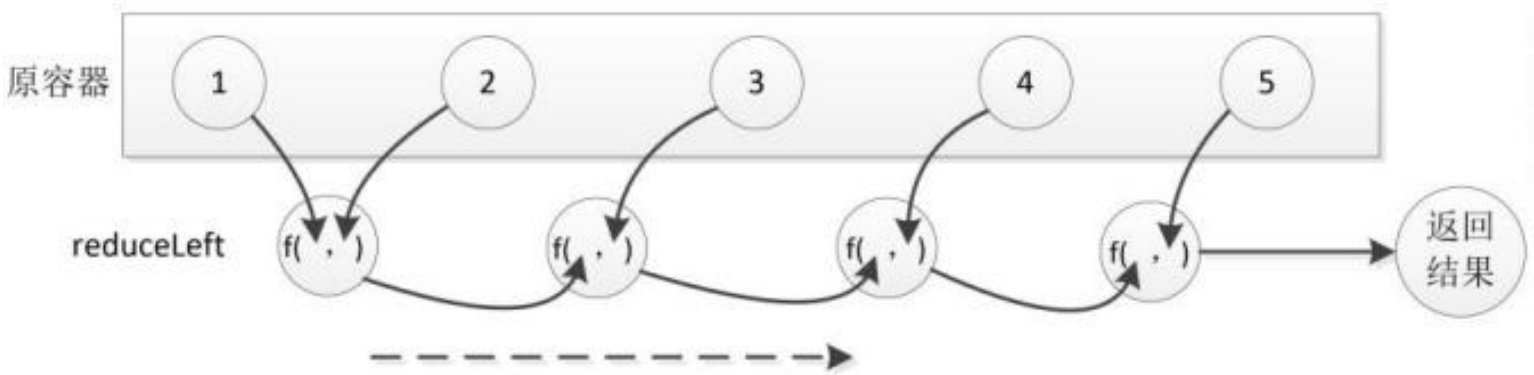
图5-8 reduceByKey()操作实例执行过程示意图

SL-27: 5.1.2 RDD操作

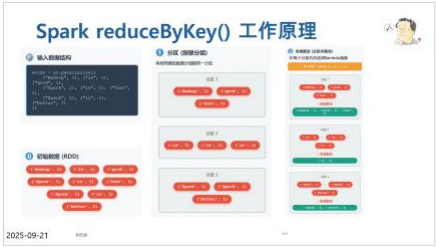


- reduceByKey(func)

rdd.reduceByKey(lambda a,b:a+b)



<“spark”,<1,1,1>>



SL-28: 5.1.2 RDD操作



- sortBy(func)

sortBy(func)会根据一定的规则对数据进行排序。下面是一个具体实例：

```
>>> rdd1 = sc.parallelize([3, 1, 2, 4],1) # 设置1个分区，保证全局有序
>>> rdd2 = rdd1.sortBy(lambda x:x)
>>> rdd2.foreach(print)
1
2
3
4
>>> rdd2 = rdd1.sortBy(lambda x:x,False) #使用参数False，实现降序排序
>>> rdd2.foreach(print)
4
3
2
1
```



SL-29: 5.1.2 RDD操作

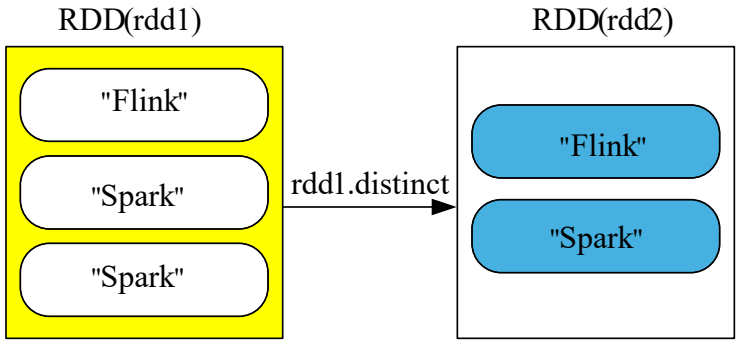


- distinct

distinct操作会对RDD内部的元素进行去重，并把去重后的元素放到新的RDD中。该方法实际是对map方法及reduceByKey方法的封装。这里给出一个简单实例，代码如下：

```
>>> rdd1 = sc.parallelize(["Flink","Spark","Spark"])
>>> rdd2 = rdd1.distinct()
>>> rdd2.foreach(print)
Flink
Spark
```

如图5-9所示，rdd1中有三个元素，即"Flink"、"Spark"和"Spark"，执行rdd1.distinct语句以后，两个重复的元素"Spark"就会被去掉一个，只保留一个，得到的结果rdd2中只会包含两个元素，即"Flink"和"Spark"。



SL-30: 5.1.2 RDD操作



- union

union操作会对两个RDD进行并集运算，并返回新的RDD，整个过程不会对元素进行去重。下面是一个具体实例。

```
>>> rdd1 = sc.parallelize([1,2,3])
>>> rdd2 = sc.parallelize([3,4,5])
>>> rdd3 = rdd1.union(rdd2)
>>> rdd3.collect()
[1, 2, 3, 3, 4, 5]
```

如图5-10所示，rdd1中有三个元素，即1、2和3，rdd2中有三个元素，即3、4和5，执行rdd1.union(rdd2)以后得到的结果rdd3中，包含了6个元素，即1、2、3、3、4和5，可以看出，整个过程没有对元素进行去重。

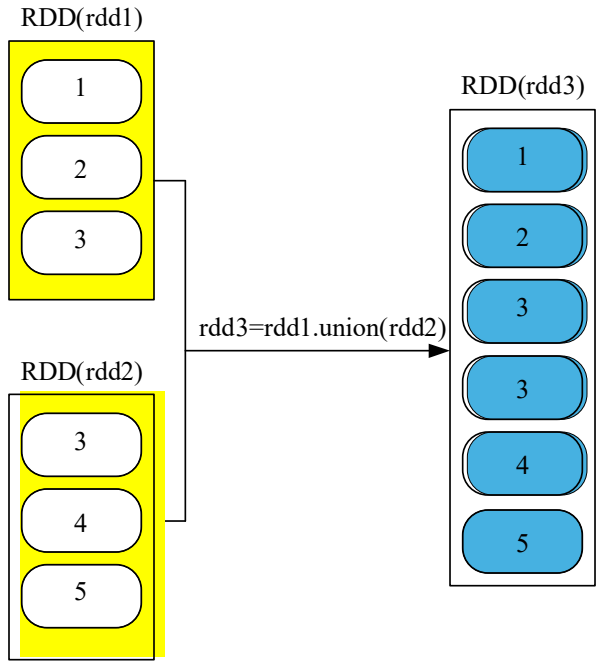


图5-10 union() 操作例执行过程示意图

SL-31: 5.1.2 RDD操作



- intersection

intersection操作会对两个RDD进行交集运算，并返回新的RDD。下面是一个具体实例。

```
>>> rdd1 = sc.parallelize([1,2,3])
>>> rdd2 = sc.parallelize([3,4,5])
>>> rdd3 = rdd1.intersection (rdd2)
>>> rdd3.collect()
[3]
```

如图5-11所示，rdd1中有三个元素，即1、2和3，rdd2中有三个元素，即3、4和5，执行rdd1.intersection (rdd2)以后得到的结果rdd3中，包含了1个元素，即3。

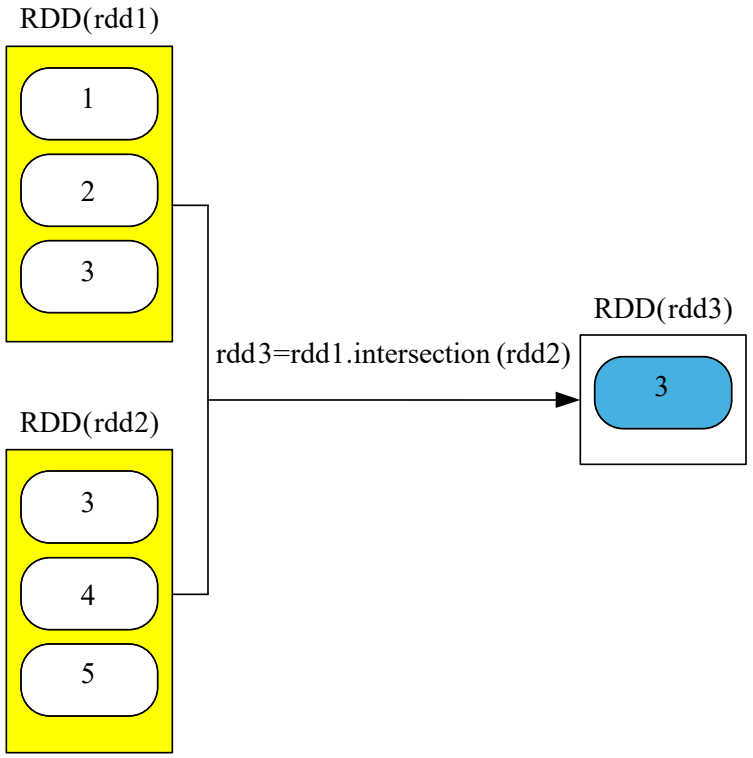


图5-11 intersection() 操作实例执行过程示意图

SL-32: 5.1.2 RDD操作



- subtract

subtract操作会对两个RDD进行差集运算，并返回新的RDD，整个过程不会对元素去重。下面是一个具体实例。

```
>>> rdd1 = sc.parallelize([1,2,3])
>>> rdd2 = sc.parallelize([3,4,5])
>>> rdd3 = rdd1.subtract (rdd2)
>>> rdd3.collect()
[1, 2]
```

如图5-12所示，rdd1中有三个元素，即1、2和3，rdd2中有三个元素，即3、4和5，执行rdd1.subtract(rdd2)以后得到的结果rdd3中，包含了2个元素，即1和2，也就是说，最终返回的是在rdd1中存在但不在rdd2中的元素。

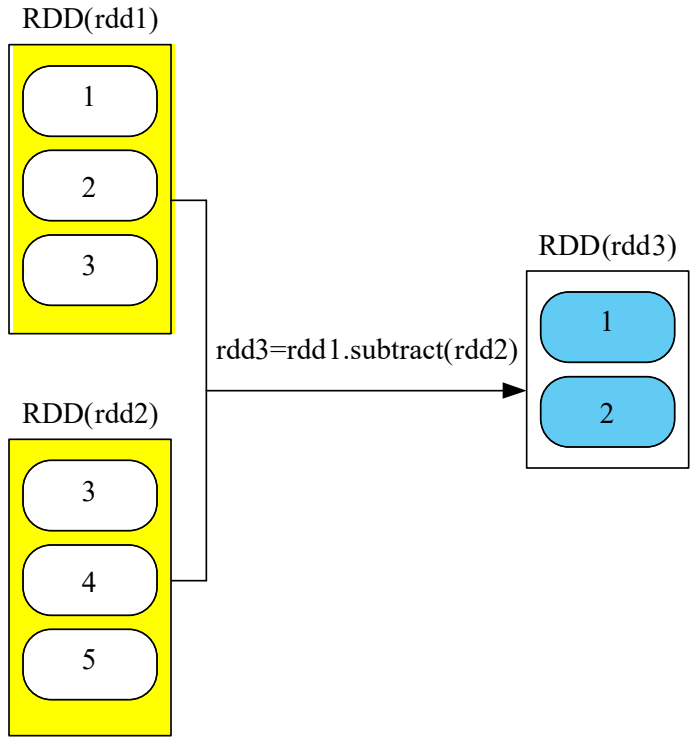


图5-12 substract() 操作实例执行过程示意图

SL-33: 5.1.2 RDD操作



- zip

zip操作会把两个RDD中的元素以键值对的形式进行合并。需要注意的是，在使用zip操作时，需要确保两个RDD中的元素个数是相同的。下面是一个具体实例。

```
>>> rdd1 = sc.parallelize([1,2,3])
>>> rdd2 = sc.parallelize(["Hadoop","Spark","Flink"])
>>> rdd3 = rdd1.zip (rdd2)
>>> rdd3.collect()
[(1, 'Hadoop'), (2, 'Spark'), (3, 'Flink')]
```

如图5-13所示，rdd1中有三个元素，即1、2和3，rdd2中有三个元素，即"Hadoop"、"Spark"和"Flink"，执行rdd1.zip(rdd2)以后得到的结果rdd3中，包含了3个元素，即(1,"Hadoop")、(2,"Spark")和(3,"Flink")。

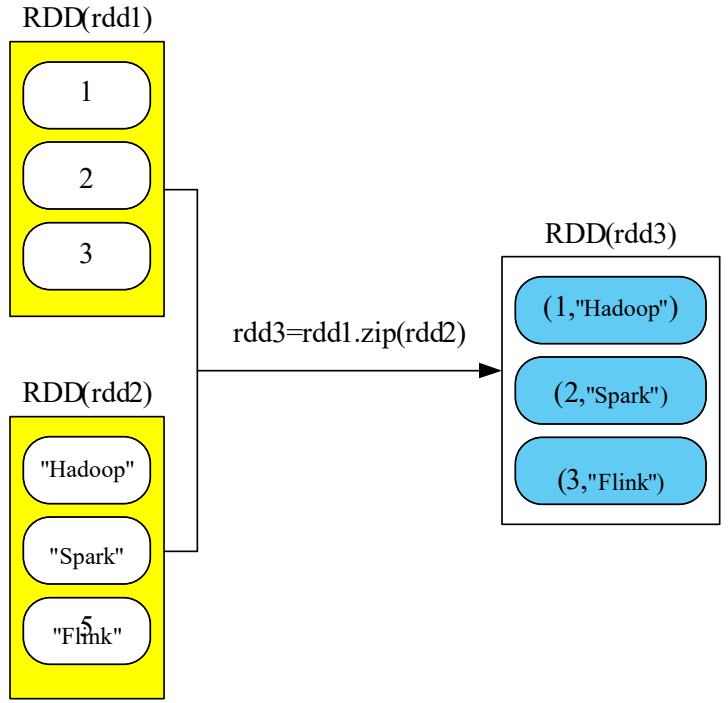


图5-13 zip() 操作实例执行过程示意图

SL-34: 5.1.2 RDD操作



2. 行动操作(action operation)

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。

表5-2 常用的RDD行动操作API

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
countByKey()	对于键值对类型的RDD进行计算，统计出每个键出现的次数
aggregate(zeroValue, seqOp, combOp)	首先使用函数seqOp把每个分区的元素进行聚合，然后使用函数combOp对所有分区进行聚合，在聚合时可以使用初始值zeroValue

SL-35: 5.1.2 RDD操作



2. 行动操作

```
>>> rdd = sc.parallelize([1,2,3,4,5])
>>> rdd.count()
5
>>> rdd.first()
1
>>> rdd.take(3)
[1, 2, 3]
>>> rdd.reduce(lambda a,b:a+b)
15
>>> rdd.collect()
[1, 2, 3, 4, 5]
>>> rdd.foreach(lambda elem:print(elem))
1
2
3
4
5
```

SL-36: 5.1.2 RDD操作



- 下面再给与一个关于countByKey算子的实例：

```
>> rdd = sc.parallelize([("hadoop",1),("spark",1),("spark",1)])
>>> result = rdd.countByKey()
>>> print(result)
defaultdict(<class 'int'>, {'hadoop': 1, 'spark': 2})
```

- 最后，给出一个关于aggregate算子的实例：

```
>>> rdd1 = sc.parallelize([1, 2, 3, 4], 2)
>>> result = rdd1.aggregate(0,lambda a,b:a+b,lambda a,b:a+b)
>>> print(result)
10
```

SL-37: 5.1.2 RDD操作



3. 惰性机制(Lazy Mechanism)

- 所谓的“惰性机制”是指，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，才会触发“从头到尾”的真正的计算
- 这里给出一段简单的语句来解释Spark的惰性机制

```
>>> lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> lineLengths = lines.map(lambda s:len(s))
>>> totalLength = lineLengths.reduce(lambda a,b:a+b)
>>> print(totalLength)
```

SL-38: 5.1.3 持久化(persistence)



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

下面就是多次计算同一个RDD的例子：

```
>>> list = ["Hadoop","Spark","Hive"]
>>> rdd = sc.parallelize(list)
>>> print(rdd.count()) //行动操作，触发一次真正从头到尾的计算
3
>>> print(','.join(rdd.collect())) //行动操作，触发一次真正从头到尾的计算
Hadoop,Spark,Hive
```

SL-39: 5.1.3 持久化



- 可以通过持久化（缓存）机制避免这种重复计算的开销
- 可以使用persist()方法对一个RDD标记为持久化
- 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
- 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用

SL-40: 5.1.3 持久化



1. persist()的圆括号中包含的是持久化级别参数：

- 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
- 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
- 一般而言，使用cache()方法时，会调用persist(MEMORY_ONLY)
- 可以使用unpersist()方法手动地把持久化的RDD从缓存中移除

SL-41: 5.1.3 持久化



1. 针对上面的实例，增加持久化语句以后的执行过程如下：

```
>>> list = ["Hadoop","Spark","Hive"]
>>> rdd = sc.parallelize(list)
>>> rdd.cache()  #会调用persist(MEMORY_ONLY)，但是，语句执行到这里，并不会缓存rdd，因为这时rdd还没有被计算生成
>>> print(rdd.count()) #第一次行动操作，触发一次真正从头到尾的计算，这时上面的rdd.cache()才会被执行，把这个rdd放到缓存中
3
>>> print(','.join(rdd.collect())) #第二次行动操作，不需要触发从头到尾的计算，只需要重复使用上面缓存中的rdd
Hadoop,Spark,Hive
```

SL-42: 5.1.4 分区(partitions)



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

1. 分区的作用
   1) 增加并行度

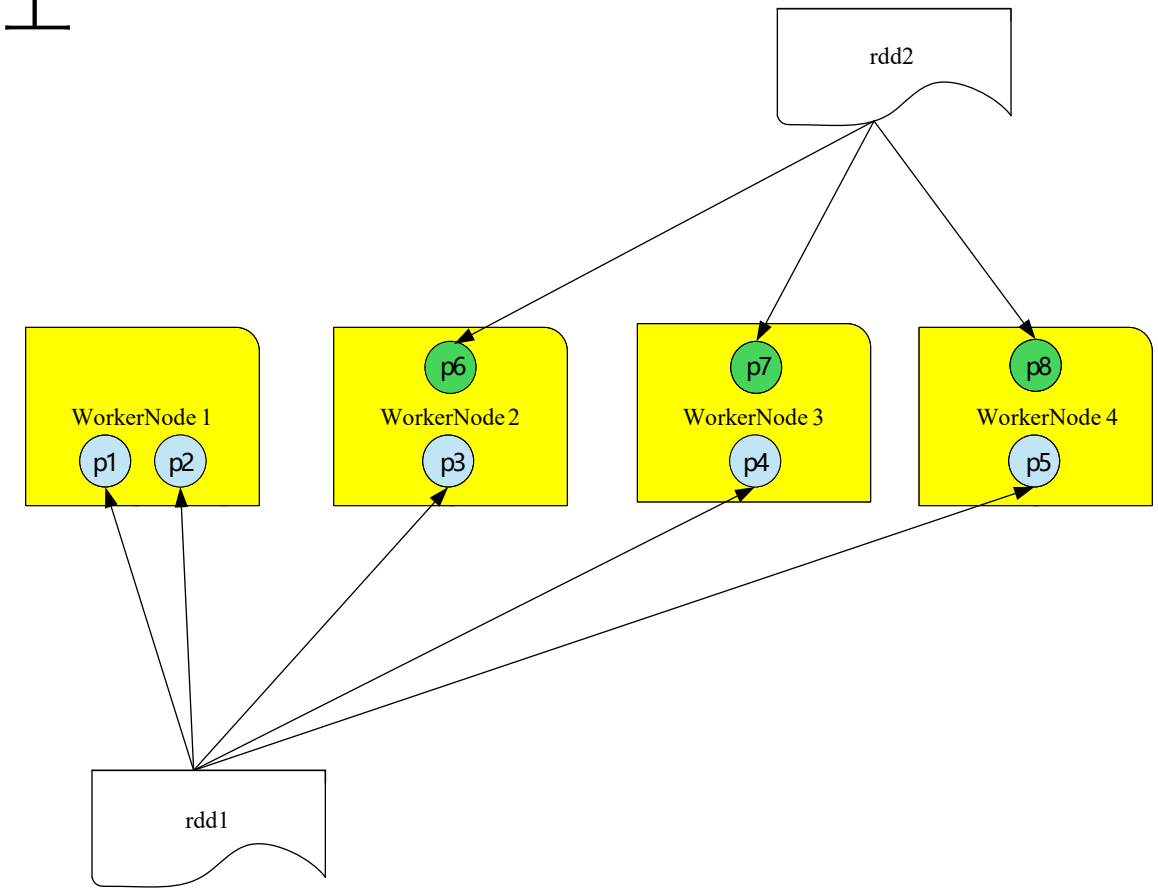


图5-14 RDD分区被保存到不同节点上

SL-43: 5.1.4 分区(partitions)



2) 减少通信开销

UserData（UserId，UserInfo）
Events（UserID，LinkInfo）
UserData 和Events 表进行连接操作，获得（UserID，UserInfo，LinkInfo）

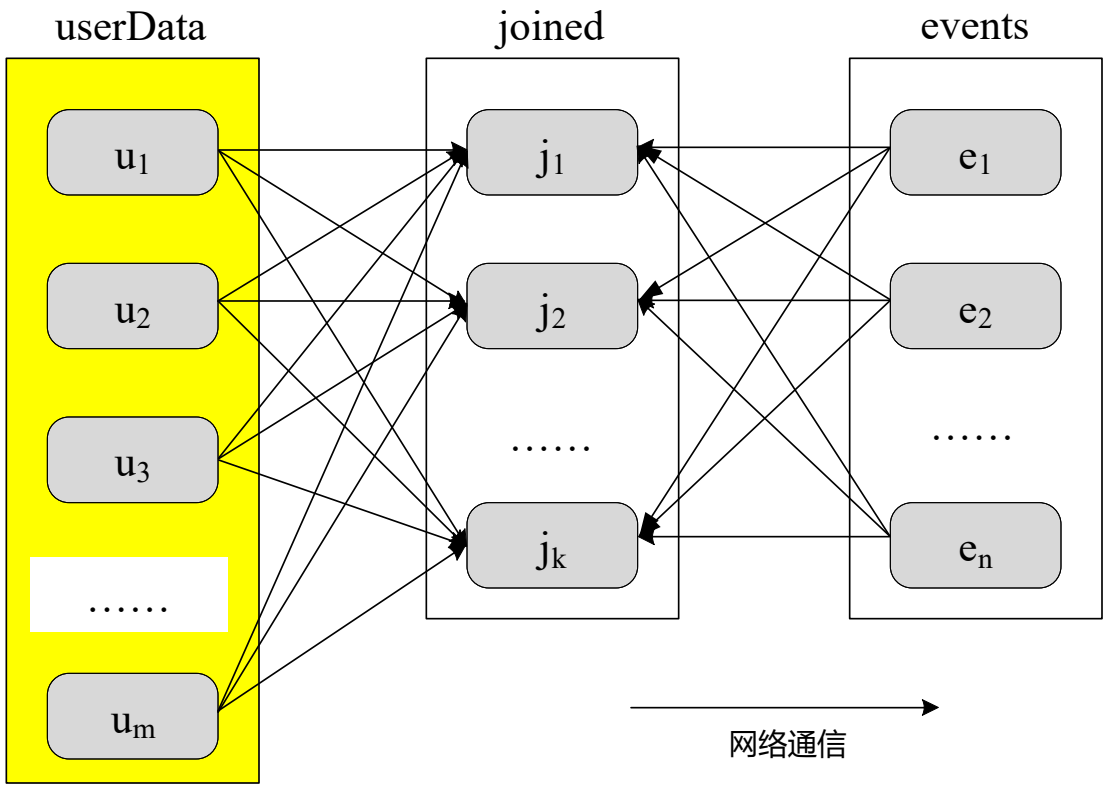


图5-15 未分区时对UserData和Events两个表进行连接操作

SL-44: 5.1.4 分区

UserData（UserId，UserInfo）
Events（UserID，LinkInfo）
UserData 和 Events 表进行连接操作，获得（UserID，UserInfo，LinkInfo）



2) 减少通信开销

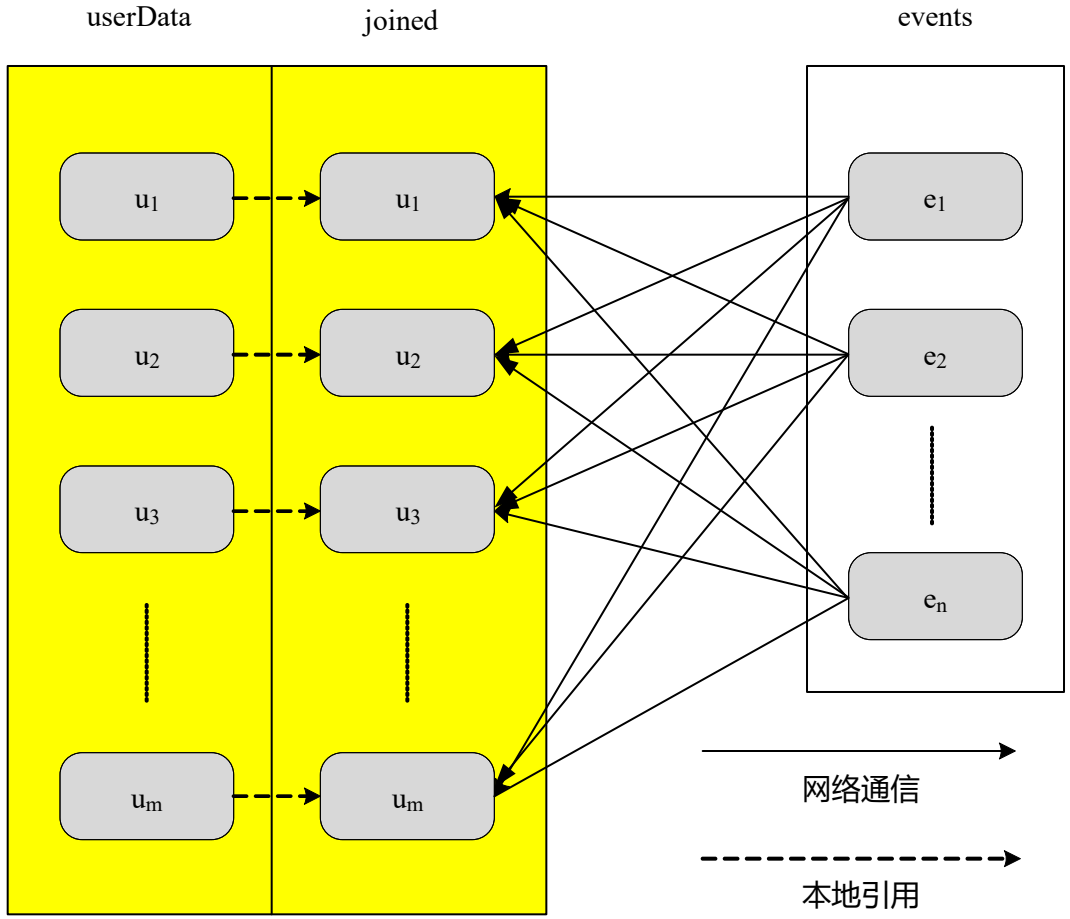


图5-16 采用分区以后对UserData和Events两个表进行连接操作

SL-45: 5.1.4 分区(partitions)



2. RDD分区原则

- RDD分区的一个原则是使得分区的个数尽量等于集群中的CPU核心（core）数目
- 对于不同的Spark部署模式而言（本地模式、Standalone模式、YARN模式、Mesos模式），都可以通过设置spark.default.parallelism这个参数的值，来配置默认的分区数目，
- 一般而言：
  - 本地模式：默认为本地机器的CPU数目，若设置了local[N],则默认为N
  - Apache Mesos：默认的分区数为8
  - Standalone或YARN：在“集群中所有CPU核心数目总和”和“2”二者中取较大值作为默认值

SL-46: 5.1.4 分区



3. 设置分区的个数

1)创建RDD时手动指定分区个数

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
```

SL-47: 5.1.4 分区



3. 设置分区的个数

2) 使用repartition()方法重新设置分区个数

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
```

SL-48: 5.1.4 分区



4. 自定义分区方法(Custom Partitioning Method)

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，
即通过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(**Communication Overhead)**

SL-49: 5.1.4 分区



实例：根据key值的最后一位数字，写到不同的文件

例如：
10写入到part-00000
11写入到part-00001
.
.
.
19写入到part-00009

SL-50: 5.1.4 分区



```
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()
```

SL-51: 5.1.4 分区



使用如下命令运行TestPartitioner.py：

```
$ cd /usr/local/spark/mycode/r
dd
```

或者，使用如下命令运行TestPartitioner.py：

```
$ cd /usr/local/spark/mycode/rdd
$ /usr/local/spark/bin/spark-submit TestPartitioner.
```

程序运行结果会返回如下信息：

```
The main function is running
MyPartitioner is running
The key is 0
MyPartitioner is running
The key is 1
……
MyPartitioner is running
The key is 9
```

SL-52 5.1.5 一个综合实例

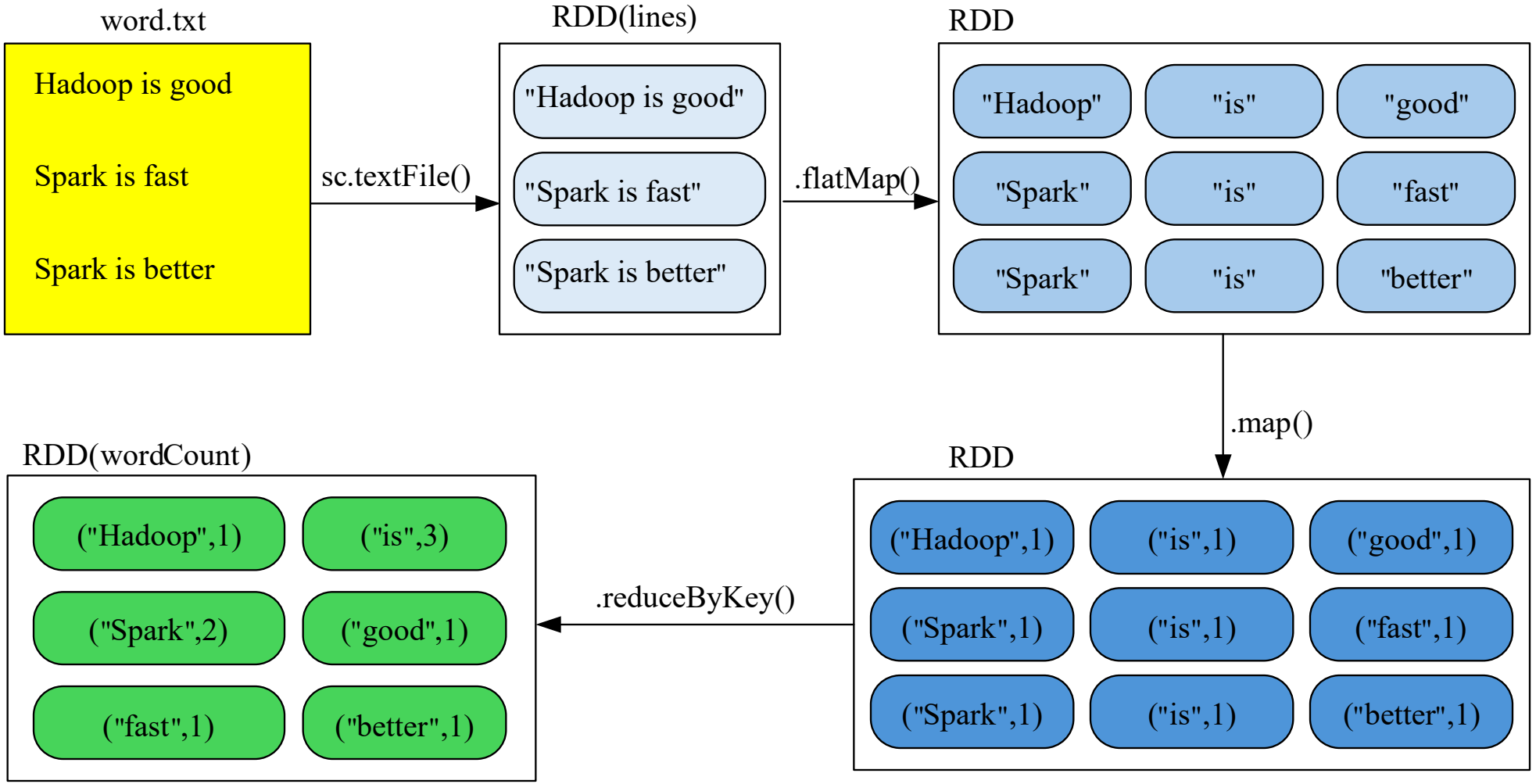


- 假设：有一个本地文件word.txt，里面包含了很多行文本，
  每行文本由多个单词构成，单词之间用空格分隔。
  可以使用如下语句进行词频统计（即统计每个单词出现的次数）：

```
>>> lines = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> wordCount = lines.flatMap(lambda line:line.split(" ")). ₩
... map(lambda word:(word,1)).reduceByKey(lambda a,b:a+b)
>>> print(wordCount.collect())
[('good', 1), ('Spark', 2), ('is', 3), ('better', 1), ('Hadoop', 1), ('fast', 1)]
```

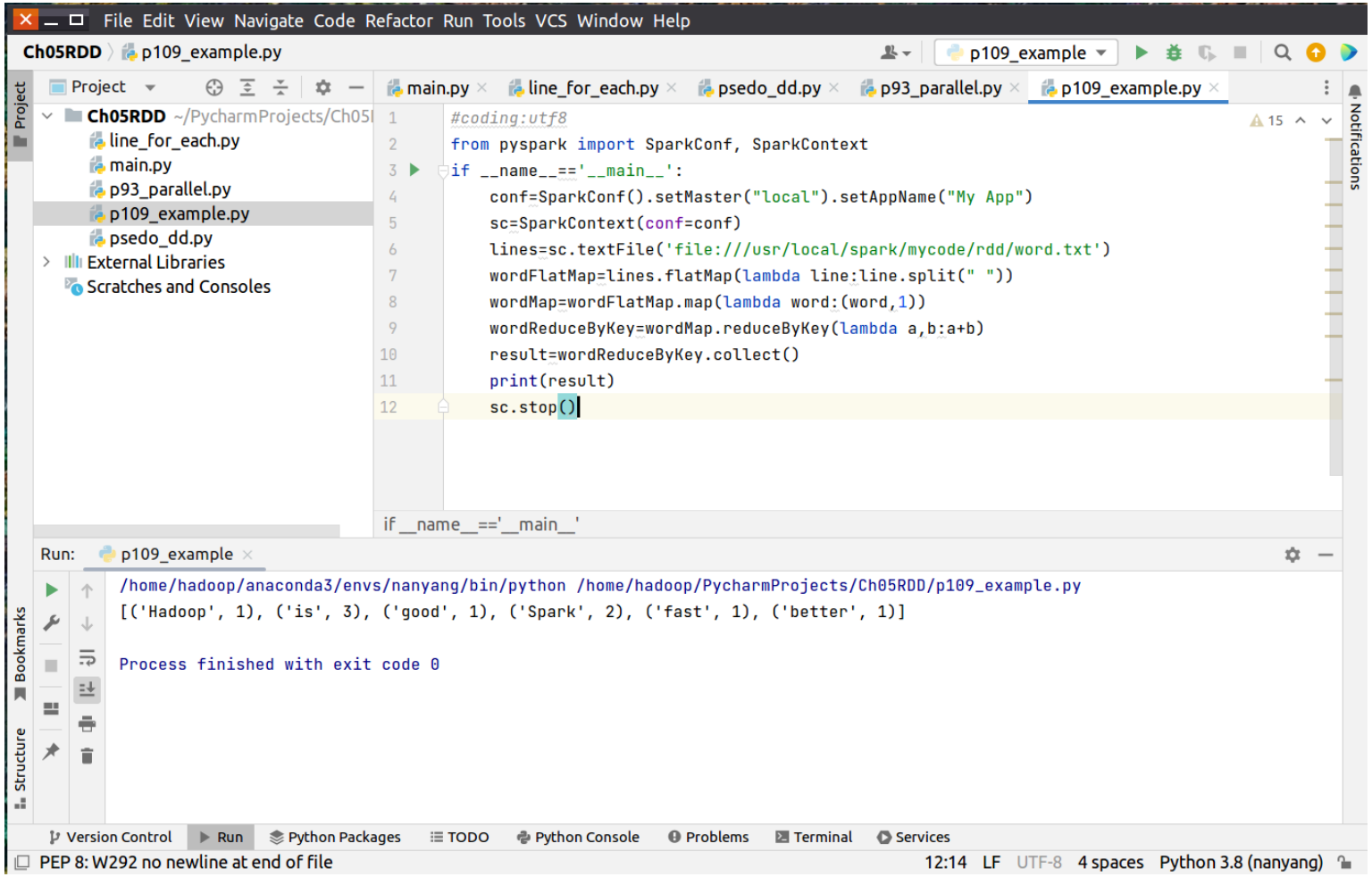
SL-53 5.1.5 一个综合实例





在Pycharm 运行




File Edit View Navigate Code Refactor Run Tools VCS Window Help
Ch05RDD 〉 p109_example.py
p109_example
Project
Ch05RDD ~/PycharmProjects/Ch05
line_for_each.py
main.py
p93_parallel.py
p109_example.py
psedo_dd.py
External Libraries
Scratches and Consoles
main.py
line_for_each.py
psedo_dd.py
p93_parallel.py
p109_example.py
#coding:utf8
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
conf=SparkConf().setMaster("local").setAppName("My App")
sc=SparkContext(conf=conf)
lines=sc.textFile('file:///usr/local/spark/mycode/rdd/word.txt')
wordFlatMap=lines.flatMap(lambda line:line.split(" "))
wordMap=wordFlatMap.map(lambda word:(word,1))
wordReduceByKey=wordMap.reduceByKey(lambda a,b:a+b)
result=wordReduceByKey.collect()
print(result)
sc.stop()
if __name__=='__main__'
Run:
p109_example
/home/hadoop/anaconda3/envs/nanyang/bin/python /home/hadoop/PycharmProjects/Ch05RDD/p109_example.py
[('Hadoop', 1), ('is', 3), ('good', 1), ('Spark', 2), ('fast', 1), ('better', 1)]
Process finished with exit code 0
Version Control
Run
Python Packages
TODO
Python Console
Problems
Terminal
Services
PEP 8: W292 no newline at end of file
12:14 LF UTF-8 4 spaces Python 3.8 (nanyang)

SL-54 5.1.5 一个综合实例



在实际应用中，单词文件可能非常大，会被保存到分布式文件系统HDFS中，Spark和Hadoop会统一部署在一个集群上

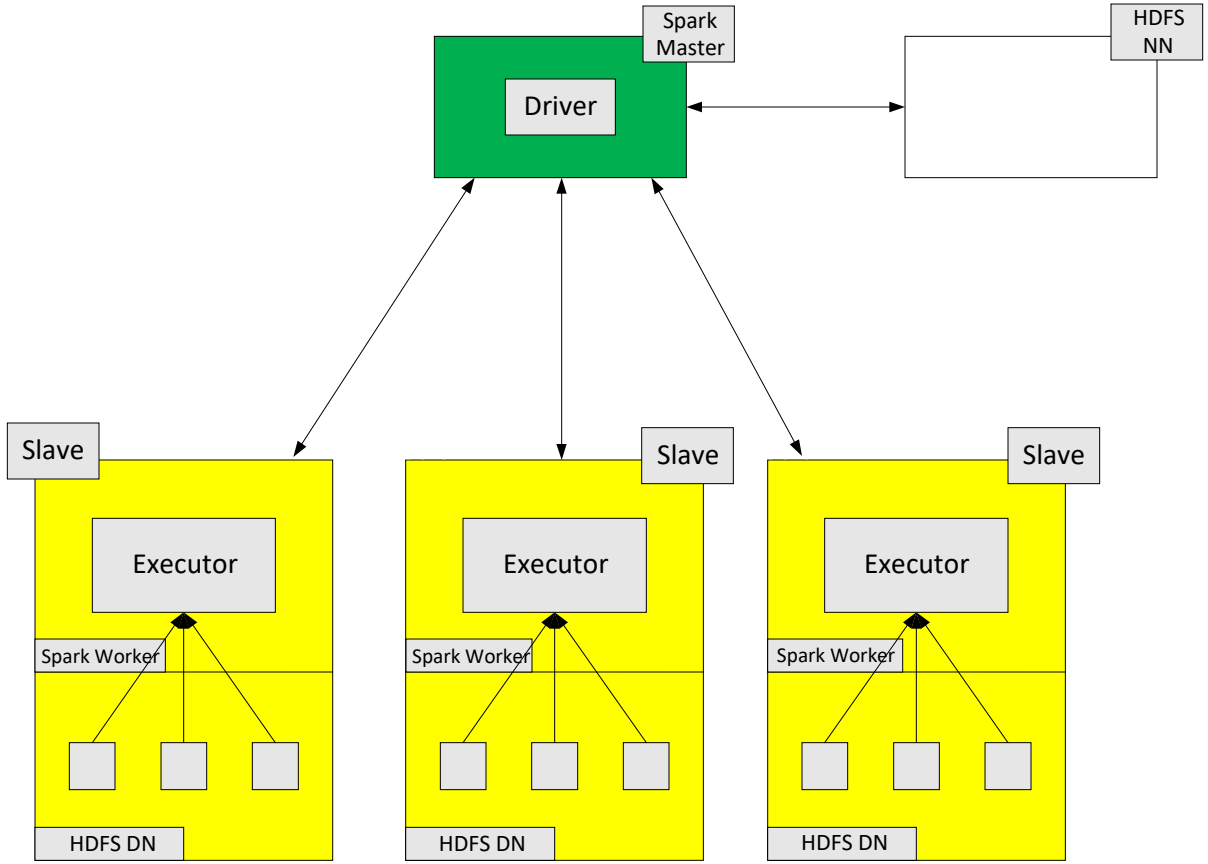


图　在一个集群中同时部署Hadoop和Spark

SL-55 5.1.5 一个综合实例



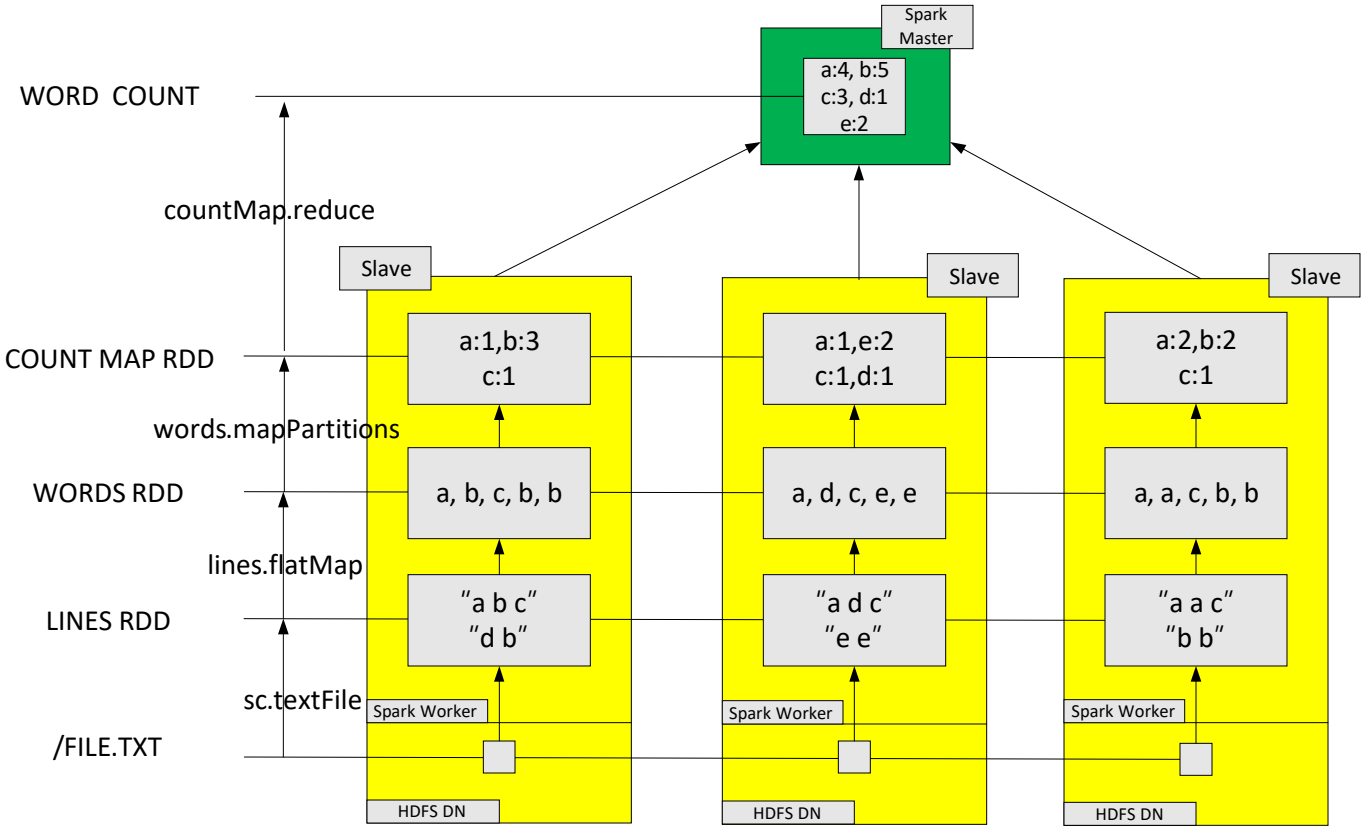


图 在集群中执行词频统计过程示意图

SL-56 5.2 键值对RDD



5.2.1 键值对RDD的创建

5.2.2 常用的键值对RDD转换操作

5.2.3 一个综合实例

SL-57 5.2.1 键值对RDD的创建



1. 第一种创建方式：从文件中加载

可以采用多种方式创建键值对RDD，其中一种主要方式是使用map()函数来实现

```
>>> 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)
……
```

SL-58 5.2.1 键值对RDD的创建



2. 第二种创建方式：通过并行集合（列表）创建RDD

```
>>> 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)
```

SL-59 5.2.2 常用的键值对RDD转换操作

1. reduceByKey(func)
2. groupByKey()
3. keys()
4. values()
5. sortByKey()
6. mapValues(func)
7. join()
8. combineByKey()

SL-60 5.2.2 常用的键值对RDD转换操作



1. reduceByKey(func)

reduceByKey(func)的功能是，使用func函数合并具有相同键的值

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

```
>>> 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)
```

SL-61 5.2.2 常用的键值对RDD转换操作



2. groupByKey()

groupByKey()的功能是，对具有相同键的值进行分组

比如，对四个键值对("spark",1)、("spark",2)、("hadoop",3)和("hadoop",5)，采用groupByKey()后得到的结果是：("spark",(1,2))和("hadoop",(3,5))

```
>>> 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>)
```

SL-62 5.2.2 常用的键值对RDD转换操作



- **reduceByKey和groupByKey的区别**
  - 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
  - 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

SL-63 5.2.2 常用的键值对RDD转换操作



reduceByKey和groupByKey的区别

```
>>> words = ["one", "two", "two", "three", "three", "three"]
>>> wordPairsRDD = sc.parallelize(words).map(lambda word:(word, 1))
>>> wordCountsWithReduce = wordPairsRDD.reduceByKey(lambda a,b:a+b)
>>> wordCountsWithReduce.foreach(print)
('one', 1)
('two', 2)
('three', 3)
>>> wordCountsWithGroup = wordPairsRDD.groupByKey(). ₩
... map(lambda t:(t[0],sum(t[1])))
>>> wordCountsWithGroup.foreach(print)
('two', 2)
('three', 3)
('one', 1)
```

上面得到的wordCountsWithReduce和wordCountsWithGroup是完全一样的，但是，它们的内部运算过程是不同的

SL-64 5.2.2 常用的键值对RDD转换操作



3. keys()

keys只会把Pair RDD中的key返回形成一个新的RDD

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

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

SL-65 5.2.2 常用的键值对RDD转换操作



4. values()

values只会把Pair RDD中的value返回形成一个新的RDD。

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

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

SL-66 5.2.2 常用的键值对RDD转换操作



5. sortByKey()

sortByKey()的功能是返回一个根据键排序的RDD

(Hadoop,1)　(Spark,1)　(Hive,1)　(Spark,1)

```
>>> 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)
```

SL-67 5.2.2 常用的键值对RDD转换操作



😊 sortByKey()和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)])
>>> print(d1.reduceByKey(lambda a,b:a+b).sortByKey(False).collect())
[('g', 21), ('f', 29), ('e', 17), ('d', 9), ('c', 27), ('b', 38), ('a', 42)]
```

```
>>> 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)]
```

SL-68 5.2.2 常用的键值对RDD转换操作



6. mapValues(func)

对键值对RDD中的每个value都应用一个函数，但是，key不会发生变化

(Hadoop,1)
(Spark,1)
(Hive,1)
(Spark,1)

```
>>> 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)
```

SL-69 5.2.2 常用的键值对RDD转换操作



7. join()

join就表示内连接。对于内连接，对于给定的两个输入数据集(K,V1)和(K,V2)，只有在两个数据集中都存在的key才会被输出，最终得到一个(K,(V1,V2))类型的数据集。

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

SL-70 5.2.2 常用的键值对RDD转换操作



8. combineByKey

combineByKey(createCombiner,mergeValue,mergeCombiners,partitioner,mapSideCombine)中的各个参数的含义如下。

（1）createCombiner：在第一次遇到key时创建组合器函数，将RDD数据集中的V类型值转换成C类型值（V => C）；

（2）mergeValue：合并值函数，表达的是同一个分区内的计算规则，再次遇到相同的key时，将createCombiner的C类型值与这次传入的V类型值合并成一个C类型值（C,V）=>C；

（3）mergeCombiners：合并组合器函数，表达的是不同分区之间的计算规则，将C类型值两两合并成一个C类型值；

（4）partitioner：使用已有的或自定义的分区函数，默认是HashPartitioner；

（5）mapSideCombine：是否在map端进行Combine操作，默认为true。

SL-71 5.2.2 常用的键值对RDD转换操作



下面通过一个实例来解释如何使用combineByKey操作。假设有一些销售数据，数据采用键值对的形式，即<公司,当月收入>，要求使用combineByKey操作求出每个公司的总收入和每月平均收入，并保存在本地文件中。

```
# coding:utf8
from pyspark import SparkConf, SparkContext
if __name__ == '__main__':
    conf = SparkConf().setMaster("local[*]").setAppName("My App")
    sc = SparkContext(conf = conf)
    data = sc.parallelize([("company-1", 88), ("company-1", 96), ("company-1", 85),
₩
                           ("company-2", 94), ("company-2", 86), ("company-2", 74), ₩
                           ("company-3", 86), ("company-3", 88), ("company-3", 92)], 3)
    res = data.combineByKey( ₩
        lambda income: (income, 1), ₩
        lambda acc, income: (acc[0] + income, acc[1] + 1), ₩
        lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1])). ₩
        map(lambda x: (x[0], x[1][0], x[1][0] / float(x[1][1])))
    res.repartition(1) ₩
        .saveAsTextFile("file:///usr/local/spark/mycode/rdd/combineresult")
    sc.stop()
```

SL-72 5.2.2 常用的键值对RDD转换操作



下面解释代码的执行过程。data = sc.parallelize()用来创建一个RDD，即data，data中的每个元素都是(key,value)键值对的形式，比如("company-1",88)和("company-1",96)。res = data.combineByKey()语句用来计算得到每个公司的总收入和平均收入，combineByKey()函数中使用了3个参数，即createCombiner、mergeValue和mergeCombiners，另外两个参数（partitioner和mapSideCombine）都采用默认值。为了让代码中combineByKey()的参数值和参数名称之间的对应关系更加清晰，表5-3列出了二者的对应关系。

参数名称	参数值
createCombiner	lambda income:(income,1)
mergeValue	lambdaacc,income:(acc[0]+income, acc[1]+1)
mergeCombiners	lambda x:(x[0],x[1][0],x[1][0]/float(x[1][1]))

SL-73 5.2.2 常用的键值对RDD转换操作



在执行data.combineByKey()时，
首先，系统取出data中的第一个RDD元素，即("company-1",88)，key是"company-1"，这个key是第一次遇到，因此Spark会为这个key创建一个组合器函数createCombiner，负责把value从V类型值转换成C类型值。这里createCombiner的值是一个Lambda表达式（或者称为匿名函数），即lambda income:(income,1)，系统会把"company-1"这个key对应的value赋值给income，也就是把88赋值给income，然后执行函数体部分，把income转换成一个元组(income,1)。因此，88会被转换成(88,1)，从V类型值变成C类型值。
然后，系统取出data中的第2个RDD元素，即("company-1",96)，key是"company-1"，这是第2次遇到相同的key，因此，系统会使用mergeValue所提供的合并值函数，将createCombiner的C类型值与这次传入的V类型值合并成一个C类型值。

SL-74 5.2.2 常用的键值对RDD转换操作



这里mergeValue参数所对应的值是一个Lambda表达式，即lambda acc,income: (acc[0]+income, acc[1]+1)，也就是使用这个Lambda表达式作为合并值函数，系统会把("company-1",96)中的96这个V类型值赋值给income，把之前已经得到的(88,1)这个C类型值赋值给acc，然后执行函数体部分，其中，acc[0]+income语句会把88和96相加，acc[1]+1语句会把(88,1)中的1增加1，得到新的C类型值(184,2)。实际上，C类型值(184,2)中，184就是company-1这个公司两个月的收入总和，2就是表示两个月。通过这种方式，下次再扫描到一个key为"company-1"的键值对时，又会把该公司的收入累加进来，最终得到"company-1"对应的C类型值(*m*,*n*)，其中，*m*表示总收入，*n*表示月份总数，用*m*除以*n*就可以得出该公司的每月平均收入。同理，当扫描到的RDD元素的key是"company-2"或者key是"company-3"时，系统也会执行类似上述的过程。这样，就可以得到每个公司对应的C类型值(*m*,*n*)。

SL-75 5.2.2 常用的键值对RDD转换操作



由于RDD元素被分成了多个分区，在实际应用中，多个分区可能位于不同的机器上，因此，需要根据mergeCombiners提供的函数，对不同分区的统计结果进行汇总。这里mergeCombiners 的值是一个Lambda表达式，即lambda acc1,acc2:(acc1[0]+acc2[0],acc1[1]+acc2[1])，该函数的功能是把两个C类型值进行合并，得到一个C类型值。比如，假设在一个分区里，key为"company-1"对应的统计结果是一个C类型值(*m*1,*n*1)，在另一个分区里key为"company-1"对应的统计结果是一个C类型值(*m*2,*n*2)，则acc1取值为(*m*1,*n*1)，acc2取值为(*m*2,*n*2)，acc1[0]+acc2[0]就是*m*1+*m*2，acc1[1]+acc2[1]就是*n*1+*n*2，最终得到一个合并后的C类型值(*m*1+*m*2,*n*1+*n*2)。

SL-76 5.2.2 常用的键值对RDD转换操作



map(lambda x:(x[0],x[1][0],x[1][0]/float(x[1][1])))语句用来求出每个公司的总收入和平均收入。输入到map()的每个RDD元素是类似于("company-1",(432,5))的形式，这时，x[0]就是"company-1"，x[1]就是(432,5)，x[1][0]就是432，x[1][1]就是5，因此，(x[0],x[1][0],x[1][0]/float(x[1][1]))的结果就是类似("company-1", 432, 86.4)这种形式。最后，res.repartition(1)语句用来把RDD从3个分区变成一个分区，这样就可以保证所有生成的结果都保存到同一个文件中（即part-00000）。

SL-77 5.2.2 常用的键值对RDD转换操作



执行如下命令运行该程序:

```
$cd /usr/local/spark/mycode/rdd
$ /usr/local/spark/bin/spark-submit Combine.py
```

执行后，在“file:///usr/local/spark/mycode/rdd/combineresult”目录下可以看到part-00000文件和_SUCCESS文件（该文件可以不用考虑），part-00000文件里面包含的结果如下:

```
('company-3', 266, 88.66666666666667)
('company-1', 269, 89.66666666666667)
('company-2', 254, 84.66666666666667)
```

在Combine.py中，如果没有使用res.repartition(1)把RDD从3个分区变成1个分区，则res这个RDD还是3个分区，那么执行后在“file:///usr/local/spark/mycode/rdd/combineresult”目录下会看到part-00000、part-00001、part-00002这3个文件和_SUCCESS文件，其中，part-00000文件中包含("company-3",266,88.666667)，part-00001文件中包含("company-1",269,89.666667)，part-00002文件中包含("company-2",254,84.666667)。

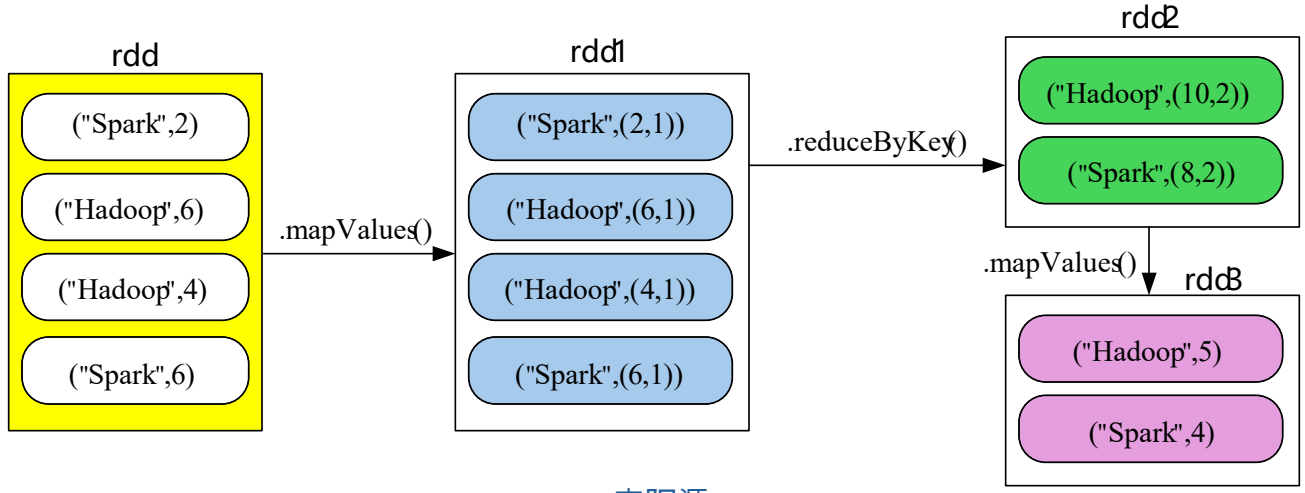
SL-78 5.2.3 一个综合实例



一个综合实例

题目：给定一组键值对("spark",2),("hadoop",6),("hadoop",4),("spark",6)，键值对的key表示图书名称，value表示某天图书销量，请计算每个键对应的平均值，也就是计算每种图书的每天平均销量。

```
>>> rdd = sc.parallelize([("spark",2),("hadoop",6),("hadoop",4),("spark",6)])
>>> rdd.mapValues(lambda x:(x,1)).⩩
... reduceByKey(lambda x,y:(x[0]+y[0],x[1]+y[1])).⩩
... mapValues(lambda x:x[0]/x[1]).collect()
[('hadoop', 5.0), ('spark', 4.0)]
```



SL-79 5.2.3 一个综合实例



如图5-20所示，rdd = sc.parallelize()执行后，生成一个名称为rdd的RDD，里面包含了4个RDD元素，即("Spark",2)、("Hadoop",6)、("Hadoop",4)、("Spark",6)。

rdd.mapValues(lambda x:(x,1))会把rdd中的每个元素依次取出来，并对该元素的value使用lambda x:(x,1)这个匿名函数进行转换。

比如，扫描到("Spark",2)这个元素时，就会把该元素的value（也就是2），转换成一个元组(2,1)，转换后得到的("Spark",(2,1))放入新的RDD（假设为rdd1）。

同理，("Hadoop",6)、("Hadoop",4)、("Spark",6)也会被分别转换成("Hadoop",(6,1))、("Hadoop",(4,1))、("Spark",(6,1))，放入到rdd1中。

SL-80 5.2.3 一个综合实例



reduceByKey(lambda x,y:(x[0]+y[0],x[1]+y[1]))会对rdd1中相同的key所对应的所有value进行聚合运算。比如，("Hadoop",(6,1))和("Hadoop",(4,1))，这两个键值对具有相同的key，
因此，reduceByKey()操作首先得到"Hadoop"这个key对应的value-list，即((6,1),(4,1))；然后，使用匿名函数lambda x,y:(x[0]+y[0],x[1]+y[1])对这个value-list进行聚合运算；
这时，Spark会把(6,1)赋值给参数x，把(4,1)赋值给参数y，因此，x._1和y._1分别是6和4，x._2和y._2都是1，(x[0]+y[0],x[1]+y[1])就是(10,2)。
如果value-list还有更多的元素，比如，假设value-list是((6,1),(4,1),(3,1))，那么，刚才计算得到的(10,2)就会作为新的x，(3,1)作为新的y，继续执行计算。
reduceByKey()操作结束后得到的新的RDD（假设为rdd2）中包含了两个元素，分别是("Hadoop",(10,2))和("Spark",(8,2))。

SL-81 5.2.3 一个综合实例



mapValues(lambda x:x[0]/x[1])操作会对rdd2中的每个元素的value执行变换，比如，当扫描到RDD中的第1个元素("Hadoop",(10,2))时，会对该元素的value（即(10,2)）进行变换。
这时，x[0]是10，x[1]是2，x[0]/x[1]就是5。
因此，经过变换后得到的结果("Hadoop",5)就会被放入新的RDD（假设为rdd3）。
最终，执行结果的RDD中就包含了("Hadoop",5)和("Spark",4)。



File Edit View Navigate Code Refactor Run Tools VCS Window Help

Ch05RDD 〉 p116_key_value_pair.py

p116_key_value_pair

Project

Ch05RDD ~/PycharmProjects/Ch05RDD

line_for_each.py
main.py
p93_parallel.py
p109_example.py
p112_differece_groupBy_reduceBy.
p113_sortBy.py
p116_key_value_pair.py
psedo_dd.py
External Libraries
Scratches and Consoles

p113_sortBy.py × p116_key_value_pair.py ×

```
1  from pyspark import SparkConf, SparkContext
2  if __name__=='__main__':
3      conf=SparkConf().setMaster("local").setAppName("My App")
4      sc=SparkContext(conf=conf)
5      list=[('Spark',2),('Hadoop',6),('Hadoop',4),('Spark',6)]
6      rdd=sc.parallelize(list)
7      rddmV=rdd.mapValues(lambda x:(x,1))
8      rddmV.foreach(print)
9      rddrBK=rddmV.reduceByKey(lambda x,y:(x[0]+y[0],x[1]+y[1]))
10     rddmV2=rddrBK.mapValues(lambda x:x[0]/x[1]).collect()
11     print(rddmV2)
12     sc.stop()
```

if __name__=='__main__'

Run: p116_key_value_pair

```
/home/hadoop/anaconda3/envs/nanyang/bin/python /home/hadoop/PycharmProjects/Ch05RDD/p116_key_value_pai
('Spark', (2, 1))
('Hadoop', (6, 1))
('Hadoop', (4, 1))
('Spark', (6, 1))
[('Spark', 4.0), ('Hadoop', 5.0)]
```

Version Control | Run | Python Packages | TODO | Python Console | Problems | Terminal | Services

PEP 8: W292 no newline at end of file 12:14 LF UTF-8 4 spaces

5.3 数据读写

5.3.1 本地文件系统的数据读写

5.3.2分布式文件系统HDFS的数据读写

5.3.3 读写MySQL数据库

5.3.1 本地文件系统的数据读写



1.从文件中读取数据创建RDD

```
>>> textFile = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> textFile.first()
'Hadoop is good'
```

因为Spark采用了惰性机制，在执行转换操作的时候，即使输入了错误的语句，spark-shell也不会马上报错（假设word123.txt不存在）

```
>>> textFile = sc.textFile("file:///usr/local/spark/mycode/wordcount/word123.txt")
```

5.3.1 本地文件系统的数据读写



2.把RDD写入到文本文件中

可以使用saveAsTextFile()方法把RDD中的数据保存到文本文件中。需要注意的是，saveAsTextFile()中提供的参数，不是文件名称，而是一个目录名称。因为Spark通常是在分布式环境下执行，RDD会存在多个分区，由多个任务对这些分区进行并行计算，每个分区的计算结果都会保存到一个单独的文件中。比如，如果RDD有3个分区，saveAsTextFile()方法就会产生part-00000、part-00001和part-00002，以及一个_SUCCESS文件，其中，part-00000、part-00001和part-00002包含了RDD中的数据，_SUCCESS文件只是用来表示写入操作已经成功执行，该文件里面是空的，可以忽略该文件。

下面把textFile变量中的内容再次写回到另外一个目录writeback中，命令如下：

```
>>> textFile = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> textFile.saveAsTextFile("file:///usr/local/spark/mycode/rdd/writeback")
```

5.3.1 本地文件系统的数据读写



上面语句执行后，请打开一个新的Linux终端，进入到“/usr/local/spark/mycode/rdd/”目录，可以看到这个目录下新增了一个名称为“writeback”的子目录。进入writeback子目录以后，可以看到该目录中生成了3个文件—part-00000、part-00001和_SUCCESS（可以忽略），part-00000和part-00001文件包含了刚才写入的数据。之所以writeback目录下包含两个part文件（即part-00000和part-00001），而不是一个part文件，是因为我们在启动进入pyspark环境时，使用了如下命令：

```
$ cd /usr/local/spark
$ ./bin/pyspark
```

5.3.1 本地文件系统的数据读写



在上述启动命令中，pyspark命令后面没有带上任何参数，则系统默认采用local[*]模式启动pyspark，当本地计算机的CPU有两个核时，系统就会使用两个Worker线程本地化运行Spark。而且，在读取文件时，sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")语句的圆括号中的参数只有文件地址，并没有包含分区数量，这时，系统会自动为textFile这个RDD生成两个分区，因此saveAsTextFile()最终生成的文件就有两个，即part-00000和part-00001。作为对比，下面在启动pyspark时设置masterURL为“local[1]”，具体如下：

```
$ cd /usr/local/spark
$ ./bin/pyspark  --master local[1]
```

首先删除之前生成的writeback目录，然后再次执行如下语句：

```
>>>  textFile = sc.textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>>  textFile.saveAsTextFile("file:///usr/local/spark/mycode/rdd/writeback")
```

上述语句执行后，可以发现，在writeback子目录下只有1个part文件，即part-00000。

5.3.2 分布式文件系统HDFS的数据读写



从分布式文件系统HDFS中读取数据，也是采用textFile()方法，可以为textFile()方法提供一个HDFS文件或目录地址，如果是一个文件地址，它会加载该文件，如果是一个目录地址，它会加载该目录下的所有文件的数据

```
>>> textFile = sc.textFile("hdfs://hadoop01:9000/user/hadoop/word.txt")
>>> textFile.first()
```

如下三条语句都是等价的：

```
>>> textFile = sc.textFile("hdfs://hadoop01:9000/user/hadoop/word.txt")
>>> textFile = sc.textFile("/user/hadoop/word.txt")
>>> textFile = sc.textFile("word.txt")
```

同样，可以使用saveAsTextFile()方法把RDD中的数据保存到HDFS文件中，命令如下：

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

这时，在HDFS中就会生成一个目录“hdfs://hadoop01:9000/user/hadoop/writeback”。

Hadoop HDFS常用操作命令表格



分类	命令	说明
目录查看	`hdfs dfs -ls /路径`	查看目录内容
	`hdfs dfs -ls -R /路径`	递归查看子目录（包含所有子目录和文件）
	`hdfs dfs -ls -h /路径`	以人性化格式显示文件大小（KB/MB/GB）
目录删除	`hdfs dfs -rm -r /路径`	安全删除（文件会进入回收站）
	`hdfs dfs -rm -r -skipTrash /路径`	永久删除（绕过回收站直接删除）
	`hdfs dfs -rmdir /空目录路径`	仅删除空目录

5.3.3 读写MySQL数据库



1.准备工作

在Linux中启动MySQL数据库，命令如下：

```
$ service mysql start
$ mysql -u root -p  #屏幕会提示输入密码
```

在MySQL Shell环境中，输入下面SQL语句完成数据库和表的创建：

```
mysql> create database spark;
mysql> use spark;
mysql> create table student (id int(4), name char(20), gender char(4), age int(4));
mysql> insert into student values(1,'Xueqian','F',23);
mysql> insert into student values(2,'Weiliang','M',24);
mysql> select * from student;
```

5.3.3 读写MySQL数据库



要想顺利连接MySQL数据库，还需要使用MySQL数据库驱动程序。请到MySQL官网下载MySQL的JDBC驱动程序，或者直接到本教材官网的“下载专区”的“软件”目录中下载驱动程序文件mysql-connector-java-5.1.40.tar.gz，对该文件进行解压，找到文件mysql-connector-java-5.1.40-bin.jar，然后，把该驱动程序JAR包复制到Spark的安装目录的jars子目录中（即“/usr/local/spark/jars”）。

利用pyspark类库编写程序访问MySQL数据库时，还需要在Python环境中安装pymysql模块，需要在hadoop01节点的Linux终端中执行如下命令：

```
$ conda activate nanyang
$ pip install pymysql
```

```
(base) hadoop@hadoop01:~$ conda activate nanyang
(nanyang) hadoop@hadoop01:~$ pip install pymysql
Collecting pymysql
  Downloading PyMySQL-1.1.1-py3-none-any.whl.metadata (4.4 kB)
Downloading PyMySQL-1.1.1-py3-none-any.whl (44 kB)
Installing collected packages: pymysql
Successfully installed pymysql-1.1.1
(nanyang) hadoop@hadoop01:~$
```

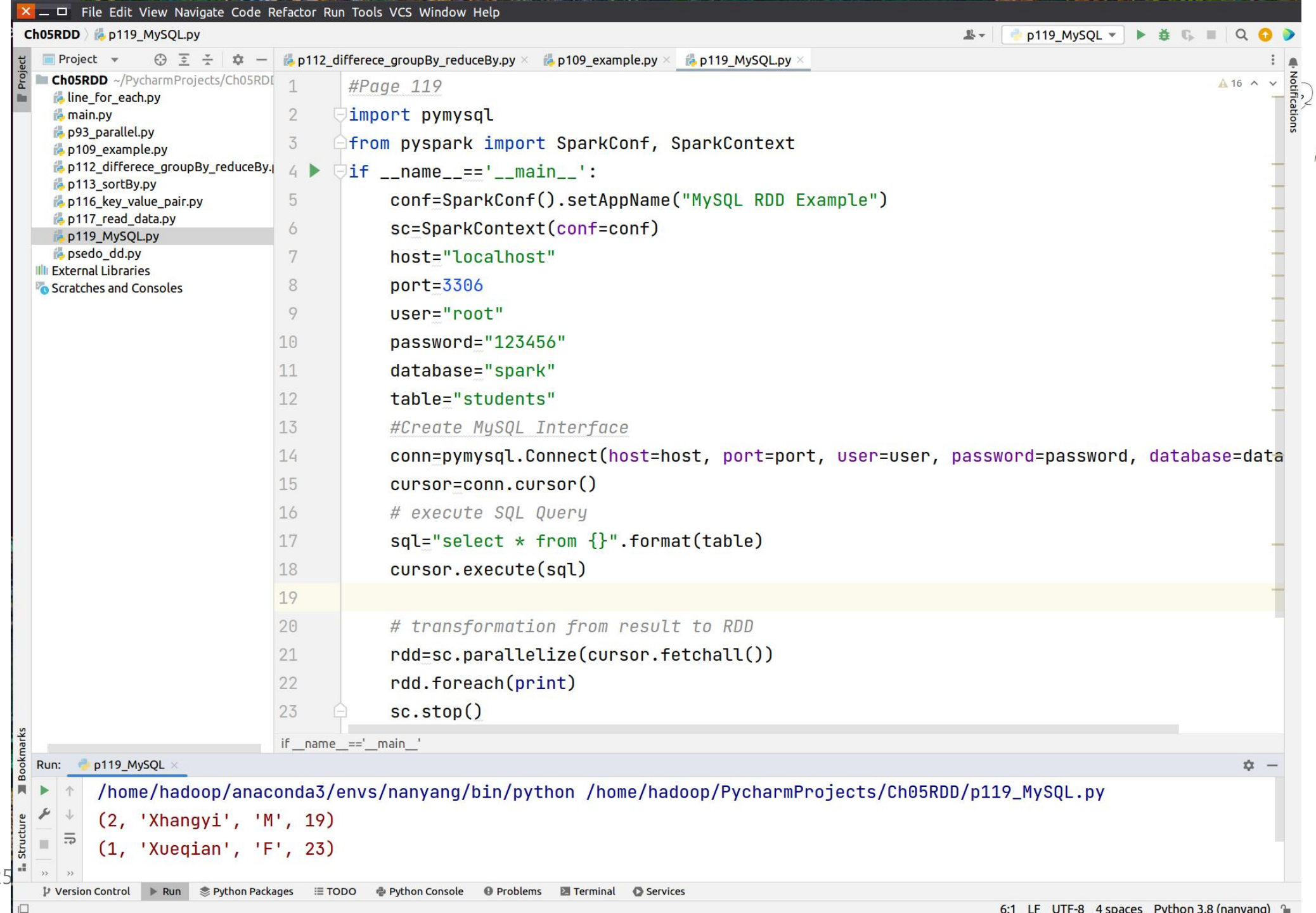
5.3.3 读写MySQL数据库



2.读取MySQL数据库中的数据

新建一个代码文件SparkReadMySQL.py，内容如下：

```
from pyspark import SparkConf, SparkContext
import pymysql
if __name__ == '__main__':
    conf = SparkConf().setAppName("MySQL RDD Example")
    sc = SparkContext(conf=conf)
    host = "localhost"
    port = 3306
    user = "root"
    password = "123456"
    database = "spark"
    table = "student"
    # 创建MySQL连接
    conn = pymysql.connect(host=host, port=port, user=user, password=password, database=database)
    cursor = conn.cursor()
    # 执行SQL查询
    sql = "SELECT * FROM {}".format(table)
    cursor.execute(sql)
    # 将结果转换为RDD
    rdd = sc.parallelize(cursor.fetchall())
    rdd.foreach(print)
    sc.stop()
```


File Edit View Navigate Code Refactor Run Tools VCS Window Help
Ch05RDD 〉 p119_MySQL.py
p119_MySQL
Project
p112_differece_groupBy_reduceBy.py
p109_example.py
p119_MySQL.py
Ch05RDD ~/PycharmProjects/Ch05RDD
line_for_each.py
main.py
p93_parallel.py
p109_example.py
p112_differece_groupBy_reduceBy.
p113_sortBy.py
p116_key_value_pair.py
p117_read_data.py
p119_MySQL.py
psedo_dd.py
External Libraries
Scratches and Consoles
#Page 119
import pymysql
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
conf=SparkConf().setAppName("MySQL RDD Example")
sc=SparkContext(conf=conf)
host="localhost"
port=3306
user="root"
password="123456"
database="spark"
table="students"
#Create MySQL Interface
conn=pymysql.Connect(host=host, port=port, user=user, password=password, database=data
cursor=conn.cursor()
# execute SQL Query
sql="select * from {}".format(table)
cursor.execute(sql)
# transformation from result to RDD
rdd=sc.parallelize(cursor.fetchall())
rdd.foreach(print)
sc.stop()
if __name__=='__main__'
Run: p119_MySQL
/home/hadoop/anaconda3/envs/nanyang/bin/python /home/hadoop/PycharmProjects/Ch05RDD/p119_MySQL.py
(2, 'Xhangyi', 'M', 19)
(1, 'Xueqian', 'F', 23)
Version Control
Run
Python Packages
TODO
Python Console
Problems
Terminal
Services
6:1 LF UTF-8 4 spaces Python 3.8 (nanyang)

Ch05RDD > p120_SparkWriteMySQL.py

p112_differece_groupBy_reduceBy.py | p109_example.py | p119_MySQL.py | p120_SparkWriteMySQL.py

```
10          host="localhost"
11          port=3306
12          user="root"
13          password="123456"
14          database="spark"
15          table="students"
16          #Create MySQL Interface
17          conn=pymysql.Connect(host=host, port=port, user=user, password=password, database=database)
18          cursor=conn.cursor()
19          # execute SQL Query
20          sql="INSERT INTO {} (id, name, gender, age) VALUES (%d, '%s','%s',%d)".format(table)
21          value=(tuple[0], tuple[1],tuple[2],tu
22          cursor.execute(sql % value)
23          conn.commit()
24          cursor.close()
25          conn.close()
26          # transformation from result to RDD
27      rddData.foreach(insertIntoMySQL)
28      sc.stop()
```

if __name__=='__main__' > insertIntoMySQL()

Run: p120_SparkWriteMySQL

```
/home/hadoop/anaconda3/envs/nanyang/bin/python

Process finished with exit code 0
```

Version Control | Run | Python Packages | TODO | Python Console | Problems

Terminal File Edit View Search Terminal Help

```
mysql> select * from student;
ERROR 1146 (42S02): Table 'spark.student' doesn't exist
mysql> select * from students;
+------+---------+--------+------+
| id   | name    | gender | age  |
+------+---------+--------+------+
|    1 | Xueqian | F      |   23 |
|    2 | Xhangyi | M      |   19 |
+------+---------+--------+------+
2 rows in set (0.00 sec)

mysql> select * from students;
+------+------------+--------+------+
| id   | name       | gender | age  |
+------+------------+--------+------+
|    1 | Xueqian    | F      |   23 |
|    2 | Xhangyi    | M      |   19 |
|    3 | ywlee      | M      |   66 |
|    4 | YoonSangOk | M      |   56 |
+------+------------+--------+------+
4 rows in set (0.00 sec)

mysql>
```



5.3.3 读写MySQL数据库



程序的执行结果如下：

```
(1, 'Xueqian', 'F', 23)
(2, 'Weiliang', 'M', 24)
```

5.3.3 读写MySQL数据库



3.向MySQL数据库中写入数据

```
from pyspark import SparkConf, SparkContext
import pymysql
if __name__ == '__main__':
    conf = SparkConf().setAppName("MySQL RDD Example")
    sc = SparkContext(conf=conf)
    rddData = sc.parallelize([(3, "Rongcheng", "M", 26), (4, "Guanhua", "M", 27)])
    def insertIntoMySQL(tuple):
        host = "localhost"
        port = 3306
        user = "root"
        password = "123456"
        database = "spark"
        table = "student"
        conn = pymysql.connect(host=host, port=port, user=user, password=password, database=datab
ase)
        cursor = conn.cursor()
        sql = "INSERT INTO {} (id, name, gender, age) VALUES (%d, '%s', '%s', %d)".format(table)
        value = (tuple[0], tuple[1], tuple[2], tuple[3])
        cursor.execute(sql % value)
        conn.commit()
        cursor.close()
        conn.close()
    rddData.foreach(insertIntoMySQL)
    sc.stop()
```

执行该程序以后，到MySQL中就可以看到新插入了两条数据。

5.4 综合案例

5.4.1 案例1：求TOP值

5.4.2 案例2：文件排序

5.4.3 案例3：二次排序

5.4.1 案例1：求TOP值



1. 实例1

任务描述：

orderid,userid,payment,productid

file1.txt

```
1,1768,50,155
2,1218, 600,211
3,2239,788,242
4,3101,28,599
5,4899,290,129
6,3110,54,1201
7,4436,259,877
8,2369,7890,27
```

file2.txt

```
100,4287,226,233
101,6562,489,124
102,1124,33,17
103,3267,159,179
104,4569,57,125
105,1438,37,116
```

求Top N个payment值

5.4.1 案例1：求TOP值



TopN.py

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

from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("ReadHBase")
sc = SparkContext(conf = conf)
lines = sc.textFile("file:///usr/local/spark/mycode/rdd/file")
result1 = lines.filter(lambda line:(len(line.strip()) > 0) and (len(line.split(","))== 4))
result2 = result1.map(lambda x:x.split(",")[2])
result3 = result2.map(lambda x:(int(x),""))
result4 = result3.repartition(1)
result5 = result4.sortByKey(False)
result6 = result5.map(lambda x:x[0])
result7 = result6.take(5)
for a in result7:
    print(a)
```

Ch05RDD 〉 p121_Top.py

p121_Top

Project

Ch05RDD ~/PycharmProjects/Ch05

- line_for_each.py
- main.py
- p93_parallel.py
- p109_example.py
- p112_differece_groupBy_reduce
- p113_sortBy.py
- p116_key_value_pair.py
- p117_read_data.py
- p119_MySQL.py
- p120_SparkWriteMySQL.py
- p121_Top.py
- psedo_dd.py

External Libraries

Scratches and Consoles

p112_differece_groupBy_reduceBy.py | p109_example.py | p119_MySQL.py | p120_SparkWriteMySQL.py | p121_Top.py

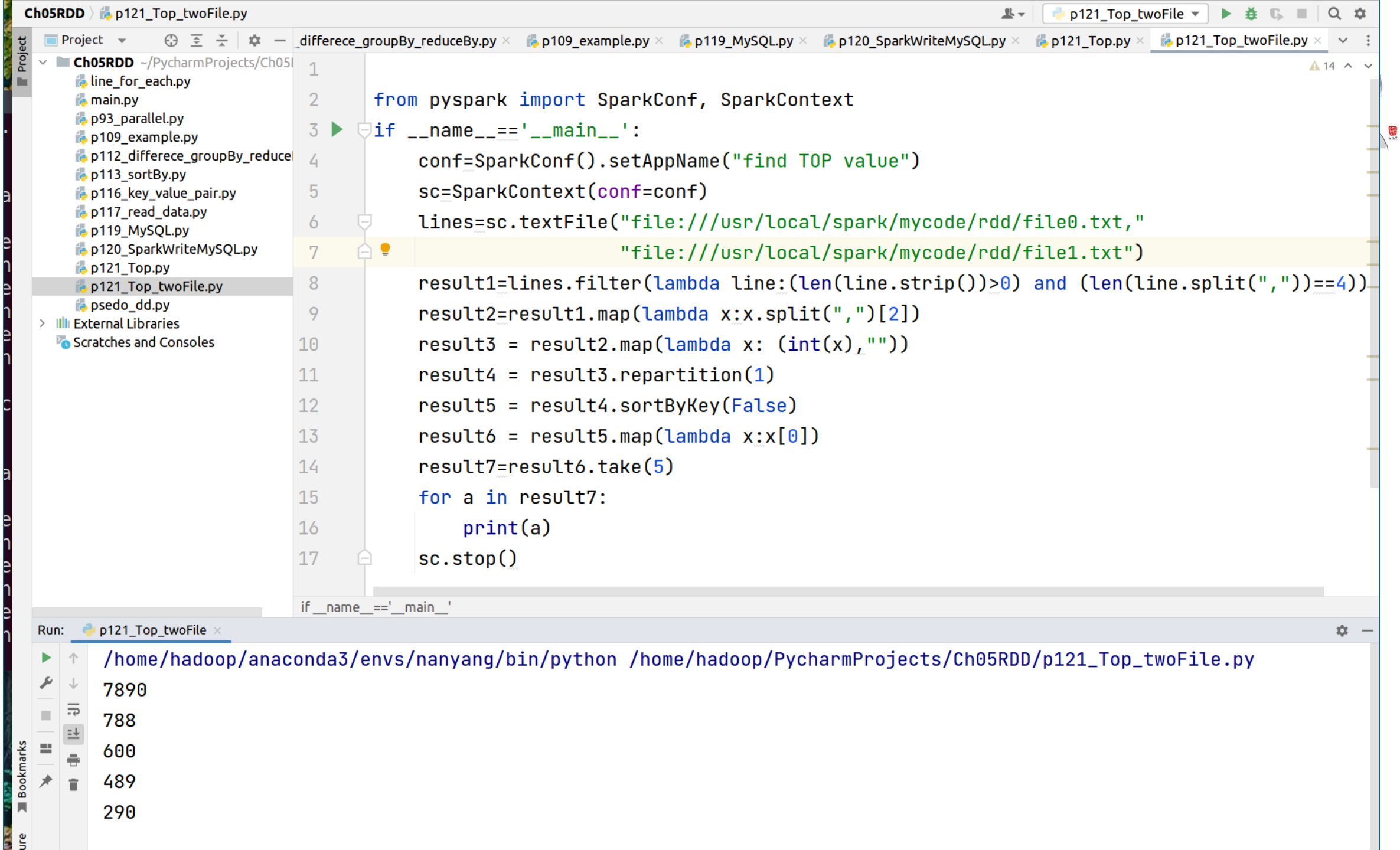
```
1  from pyspark import SparkConf, SparkContext
2  if __name__=='__main__':
3      conf=SparkConf().setAppName("find TOP value")
4      sc=SparkContext(conf=conf)
5      lines=sc.textFile("file:///usr/local/spark/mycode/rdd/file0.txt")
6      result1=lines.filter(lambda line:(len(line.strip())>0) and (len(line.split(","))==4))
7      result2=result1.map(lambda x:x.split(",")[2])
8      result3 = result2.map(lambda x: (int(x),""))
9      result4 = result3.repartition((1))
10     result5 = result4.sortByKey(False)
11     result6 = result5.map(lambda x:x[0])
12     result7=result6.take(5)
13     for a in result7:
14         print(a)
15     sc.stop()
```

Run: p121_Top

```
/home/hadoop/anaconda3/envs/nanyang/bin/python /home/hadoop/PycharmProjects/Ch05RDD/p121_Top.py
7890
788
600
290
259

Process finished with exit code 0
```

Bookmarks

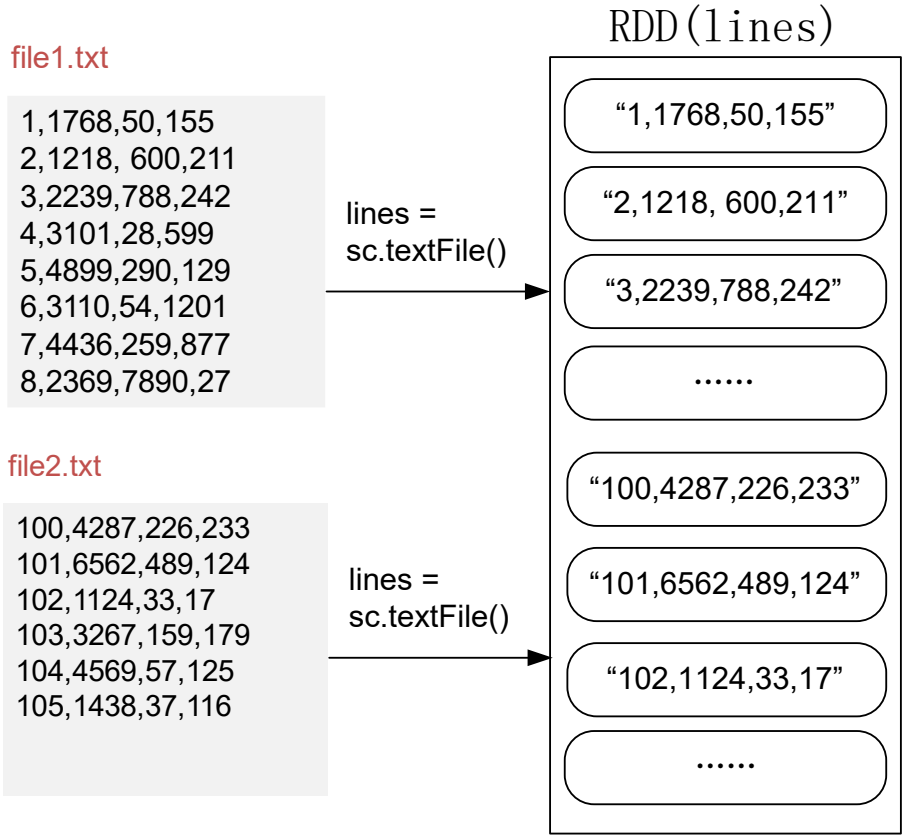

Ch05RDD
p121_Top_twoFile.py
Project
Ch05RDD ~/PycharmProjects/Ch05
line_for_each.py
main.py
p93_parallel.py
p109_example.py
p112_differece_groupBy_reduce
p113_sortBy.py
p116_key_value_pair.py
p117_read_data.py
p119_MySQL.py
p120_SparkWriteMySQL.py
p121_Top.py
p121_Top_twoFile.py
psedo_dd.py
External Libraries
Scratches and Consoles
differece_groupBy_reduceBy.py
p109_example.py
p119_MySQL.py
p120_SparkWriteMySQL.py
p121_Top.py
p121_Top_twoFile.py
14
from pyspark import SparkConf, SparkContext
if __name__=='__main__':
conf=SparkConf().setAppName("find TOP value")
sc=SparkContext(conf=conf)
lines=sc.textFile("file:///usr/local/spark/mycode/rdd/file0.txt,"
"file:///usr/local/spark/mycode/rdd/file1.txt")
result1=lines.filter(lambda line:(len(line.strip())>0) and (len(line.split(","))==4))
result2=result1.map(lambda x:x.split(",")[2])
result3 = result2.map(lambda x: (int(x),""))
result4 = result3.repartition(1)
result5 = result4.sortByKey(False)
result6 = result5.map(lambda x:x[0])
result7=result6.take(5)
for a in result7:
print(a)
sc.stop()
if __name__=='__main__'
Run:
p121_Top_twoFile
/home/hadoop/anaconda3/envs/nanyang/bin/python /home/hadoop/PycharmProjects/Ch05RDD/p121_Top_twoFile.py
7890
788
600
489
290
Bookmarks

5.4.1 案例1：求TOP值



```
lines = sc.textFile("file:///usr/local/spark/mycode/rdd/file")
```

该语句从文件中读取数据生成RDD（名称为lines），执行后的效果如下：

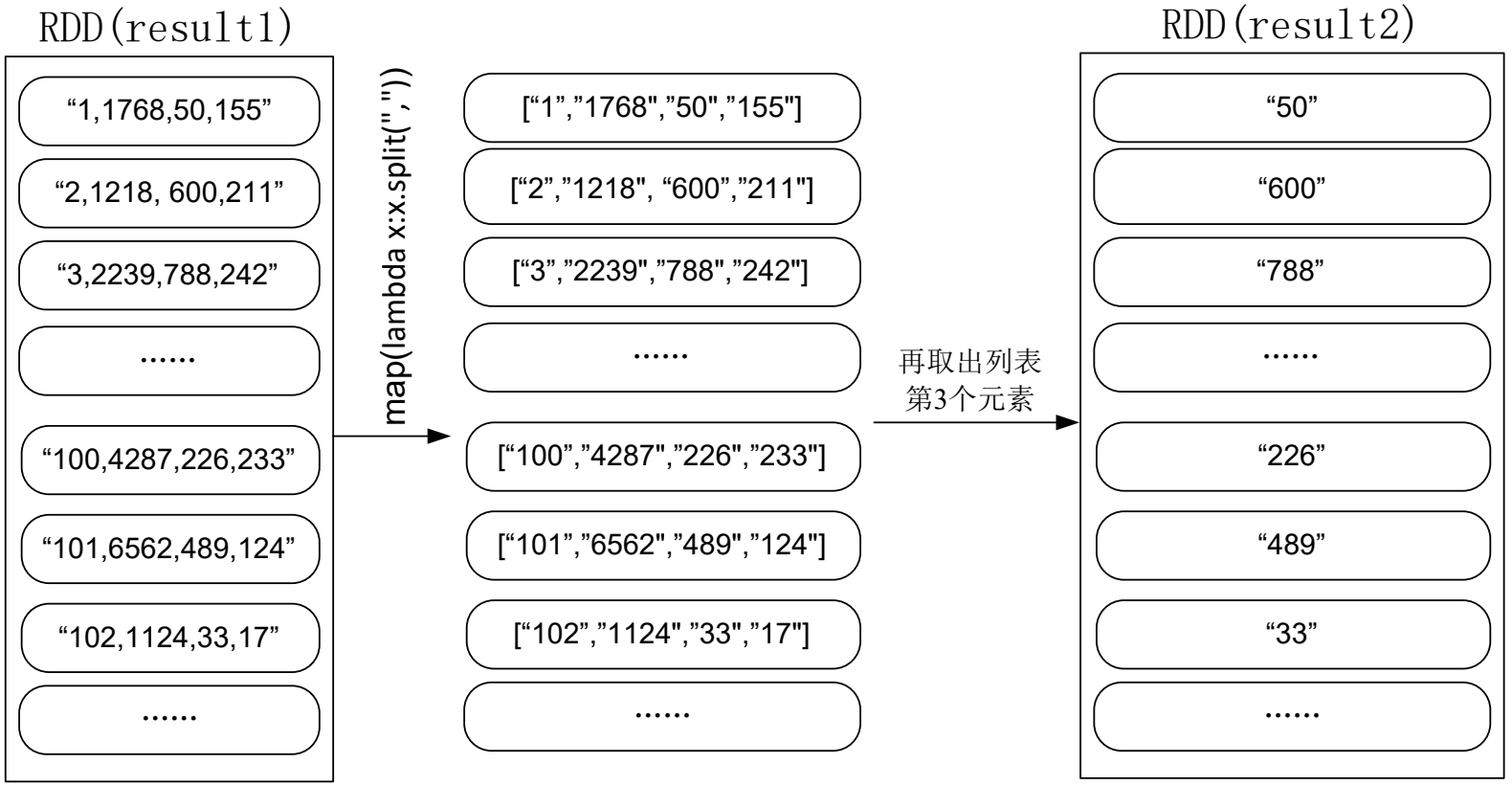


5.4.1 案例1：求TOP值



```
result1 = lines.filter(lambda line:(len(line.strip()) > 0) and (len(line.split(","))== 4))
result2 = result1.map(lambda x:x.split(",")[2])
```

该语句执行效果如下：

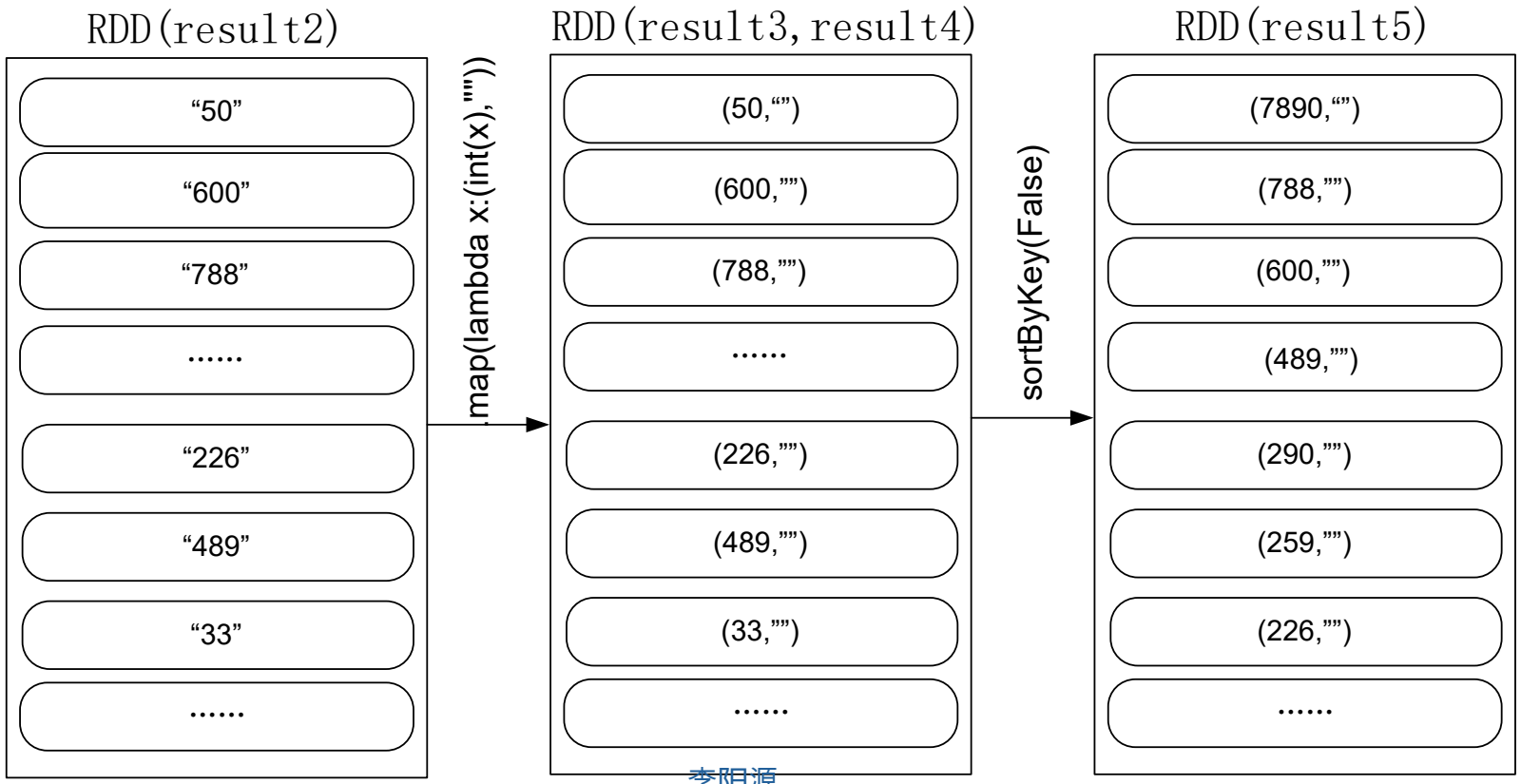


5.4.1 案例1：求TOP值



```
result3 = result2.map(lambda x:(int(x),""))
result4 = result3.repartition(1)
result5 = result4.sortByKey(False)
```

该语句执行效果如下：

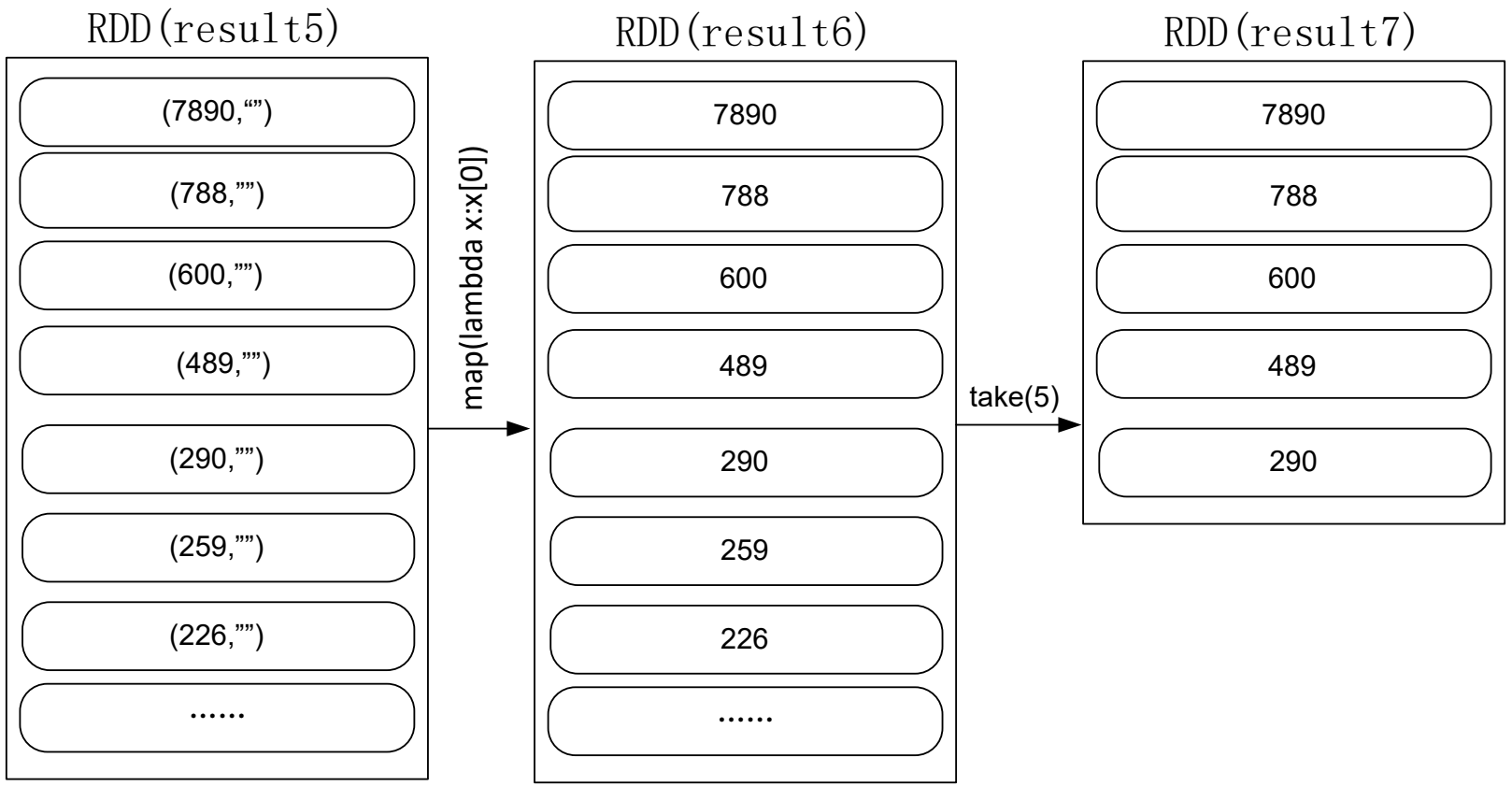


5.4.1 案例1：求TOP值



```
result6 = result5.map(lambda x:x[0])
result7 = result6.take(5)
```

该语句执行效果如下：



5.4.1 案例1：求TOP值



2. 实例2

有一个商品销售数据集sales.txt，里面的每行记录包含5个字段：时间戳、省份、城市、用户、购买的商品，字段之间用英文逗号隔开，下面是某条数据的样例：

```
1602031567000,21,1005,95001,20001
```

在这条数据中，
“1602031567000”表示时间戳，
“21”表示福建省，
“1005”表示厦门市，
“95001”表示顾客王明，
“20001”表示华为Mate60手机。

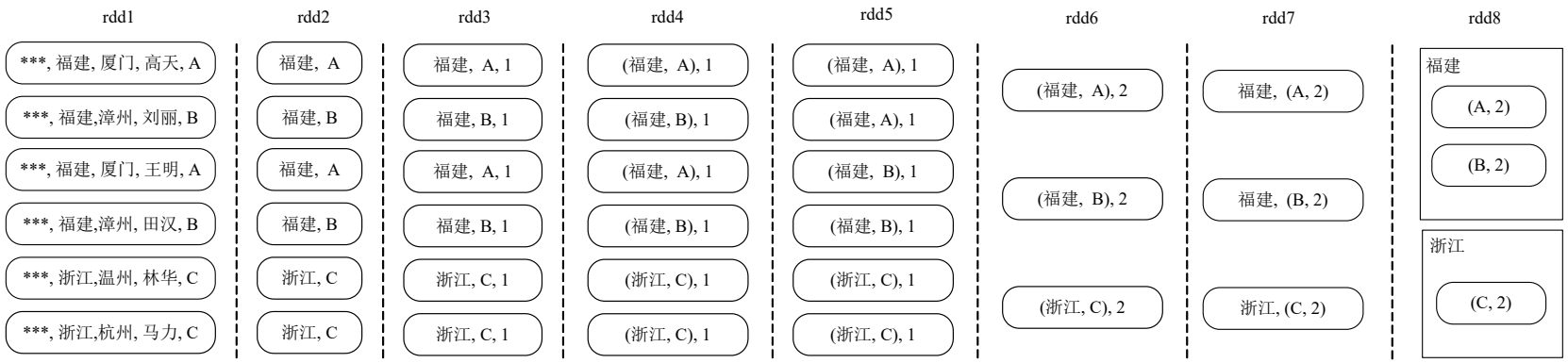
要求求出每个省份销量前3名的商品。

5.4.1 案例1：求TOP值



- 图5-25给出了结题思路，为了方便理解，这里直接使用了省份名字、城市名字、用户名字和商品名称，而不是使用编号，图中的“***”表示时间戳。
- 首先从sales.txt中加载数据生成rdd1；然后，过滤出需要的字段，即省份和商品，得到rdd2；
- 再把rdd2中的每个元素从(省份,商品)转换成(省份,商品,1)，得到rdd3；再把每个元素中的省份和商品名称组合在一起，得到rdd4；
- 再根据(省份,商品)进行分组，得到rdd5；
- 再对每个分组进行汇总，得到rdd6；再把每个元素从((省份,商品),销量))转换成(省份,(商品,销量))，得到rdd7；
- 再对元素按照省份进行分组，得到rdd8；最后，对元素进行排序，求出每个省份销量前3名的商品。

实际上，在编写程序时，可以更加精炼，也就是，可以直接从rdd1转换得到rdd4。



5.4.1 案例1：求TOP值



```
# coding:utf8
from pyspark import SparkConf, SparkContext
if __name__ == '__main__':
    conf = SparkConf().setMaster("local[*]").setAppName("My App")
    sc = SparkContext(conf = conf)
    # 1.获取原始数据：时间戳，省份，城市，用户，商品
    sourceRDD = sc.textFile("file:///home/hadoop/sales.txt")
    dataRDD = sourceRDD.filter(lambda line : len(line.split(",")) == 5)
```

5.4.1 案例1：求TOP值



```
# 2.将原始数据进行结构转换
    # 把数据从(时间戳, 省份, 城市, 用户, 商品)
    # 转变成((省份, 商品), 1)
    def myMapFunction1(line):
        datas = line.split(",")
        return ((datas[1], datas[4]), 1)
    mapRDD = dataRDD.map(myMapFunction1)
    # 3.将数据进行分组聚合
    # 得到((省份, 商品), sum)形式的结果，sum是汇总结果
    reduceRDD = mapRDD.reduceByKey(lambda a,b:a+b)
```

5.4.1 案例1：求TOP值



```
# 4.将聚合的结果进行数据结构转换
 # 从((省份, 商品), sum)转变成(省份, (商品, sum))
 def myMapFunction2(tuple):
     x = tuple[0] # 值是(省份, 商品)
     y = tuple[1] # 值是sum
     return (x[0], (x[1],y))
 newMapRDD = reduceRDD.map(myMapFunction2)
 # 5.根据省份进行分组
 # 得到的数据形式是(省份, ((商品1, sum1), (商品2, sum2), ...))
 groupRDD = newMapRDD.groupByKey()
```

5.4.1 案例1：求TOP值



```
    # 6.将分组后的数据进行组内排序（降序），取前3名
    resultRDD = groupRDD.mapValues(lambda iter: sorted(list(iter), ke
y=lambda tup: tup[1],reverse=True)[:3])
    # 7.把数据打印出到控制台
    resultRDD.foreach(print)
    sc.stop()
```

程序的运行结果如下：

```
('21', [('20002', 3), ('20001', 2), ('20003', 1)])
('20', [('20002', 4), ('20001', 2), ('20003', 2)])
```

5.4.2 案例2：文件排序



任务描述：
有多个输入文件，每个文件中的每一行内容均为一个整数。要求读取所有文件中的整数，进行排序后，输出到一个新的文件中，输出的内容个数为每行两个整数，第一个整数为第二个整数的排序位次，第二个整数为原待排序的整数

输入文件

file1.txt	file2.txt	file3.txt
33	4	1
37	16	45
12	39	25
40	5	

输出文件

1	1
2	4
3	5
4	12
5	16
6	25
7	33
8	37
9	39
10	40
11	45

5.4.2 案例2：文件排序



FileSort.py

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

from pyspark import SparkConf, SparkContext

index = 0

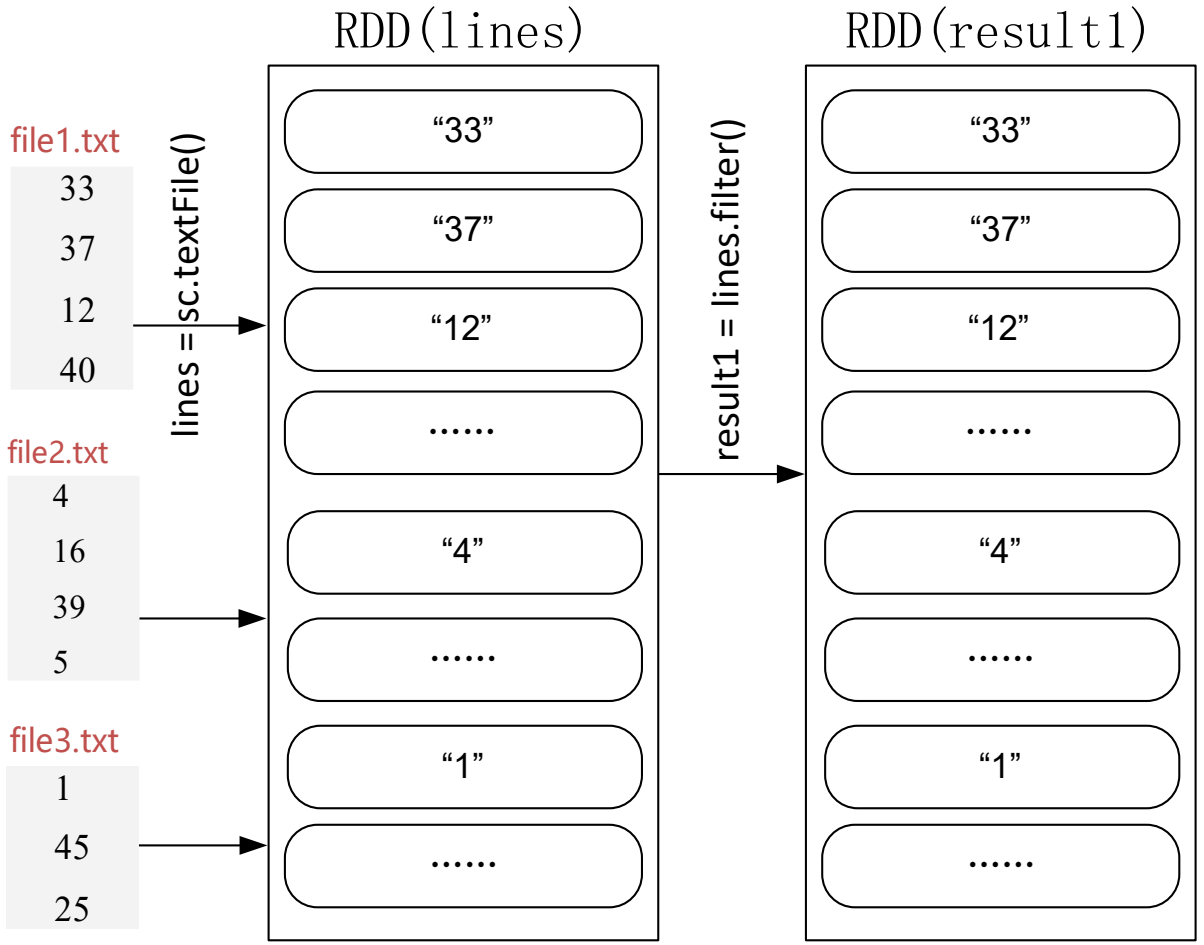
def getindex():
    global index
    index+=1
    return index

def main():
    conf = SparkConf().setMaster("local[1]").setAppName("FileSort")
    sc = SparkContext(conf = conf)
    lines = sc.textFile("file:///usr/local/spark/mycode/rdd/filesort/file*.txt")
    index = 0
    result1 = lines.filter(lambda line:(len(line.strip()) > 0))
    result2 = result1.map(lambda x:(int(x.strip()),""))
    result3 = result2.repartition(1)
    result4 = result3.sortByKey(True)
    result5 = result4.map(lambda x:x[0])
    result6 = result5.map(lambda x:(getindex(),x))
    result6.foreach(print)
    result6.saveAsTextFile("file:///usr/local/spark/mycode/rdd/filesort/sortresult")
if __name__ == '__main__':
    main()
```

5.4.2 案例2：文件排序



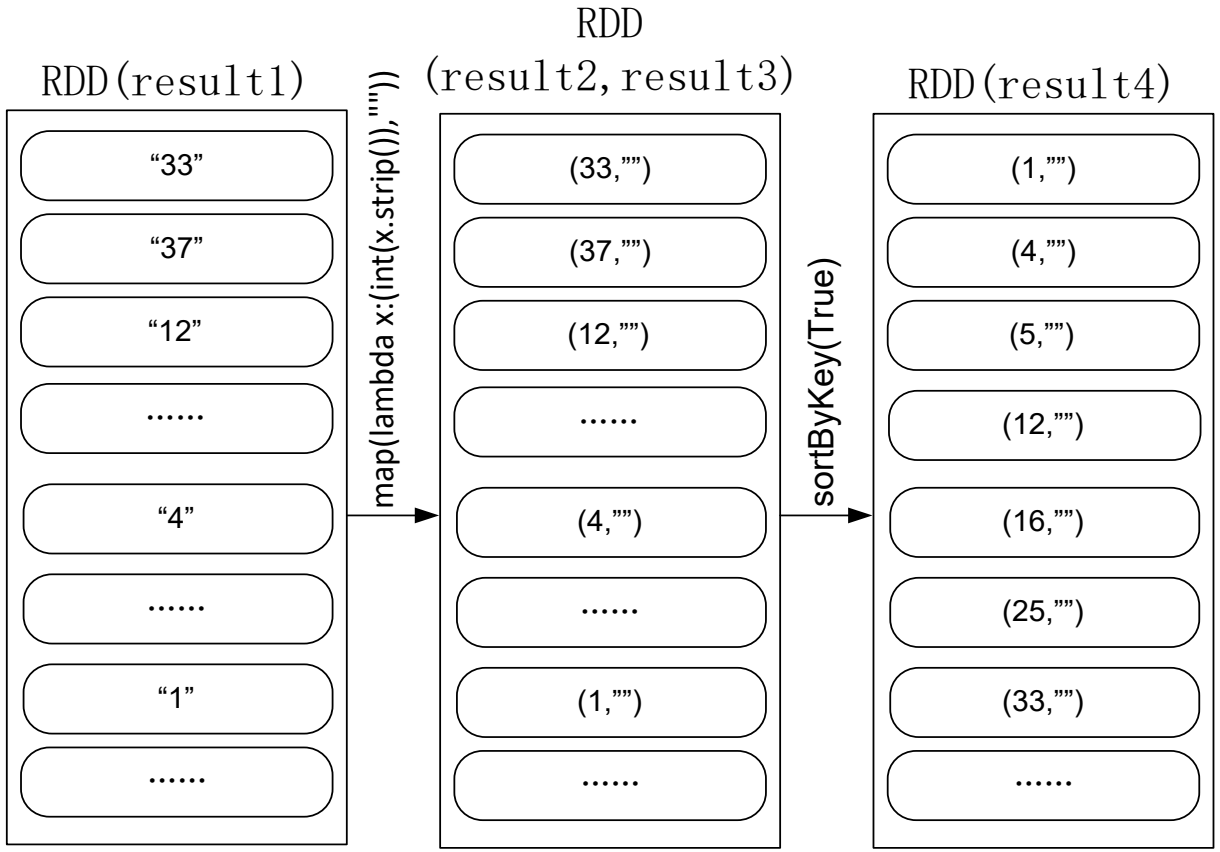
```
lines = sc.textFile("file:///usr/local/spark/mycode/rdd/filesort/file*.txt")
result1 = lines.filter(lambda line:(len(line.strip()) > 0))
```



5.4.2 案例2：文件排序



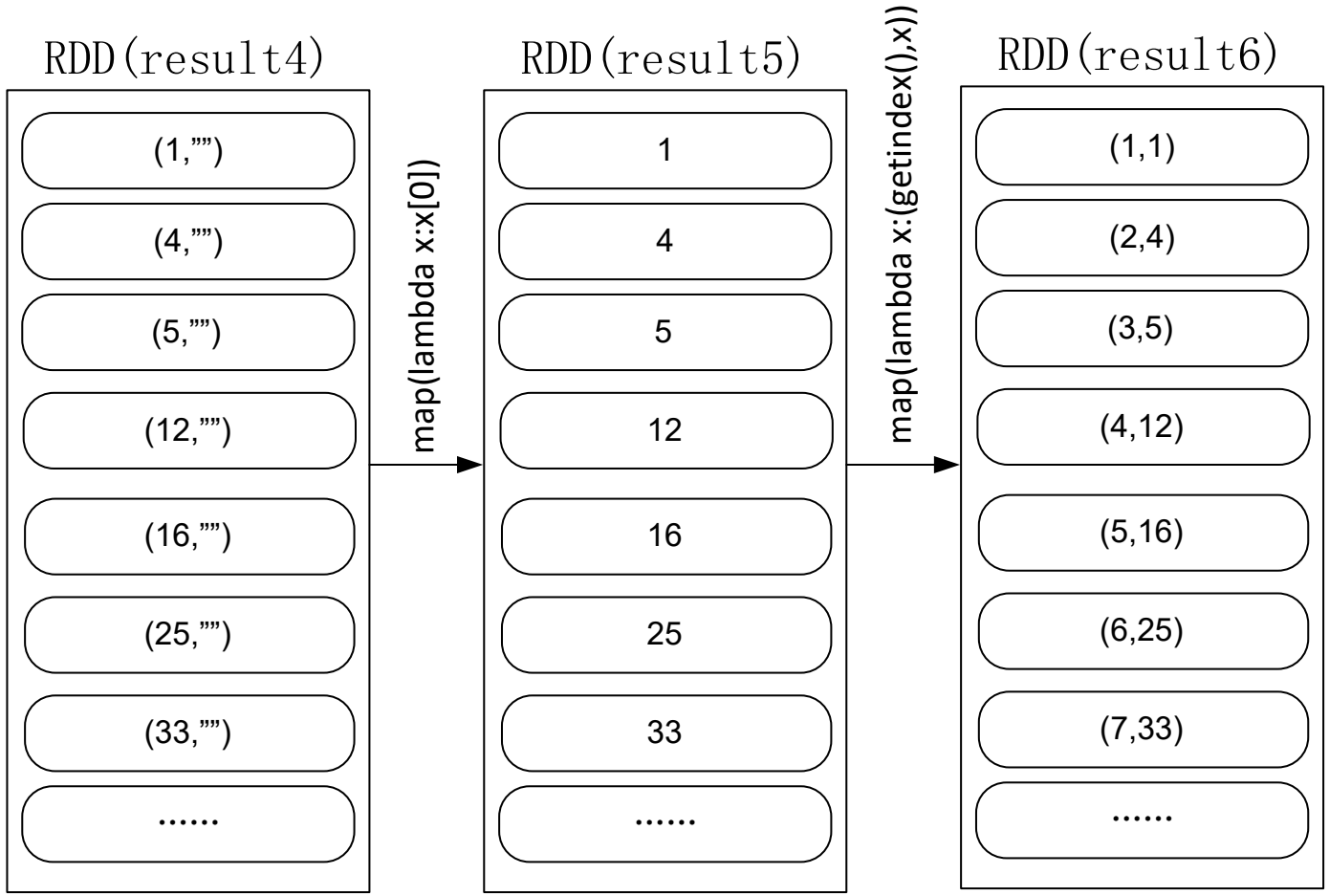
```
result2 = result1.map(lambda x:(int(x.strip()),""))
result3 = result2.repartition(1)
result4 = result3.sortByKey(True)
```



5.4.2 案例2：文件排序



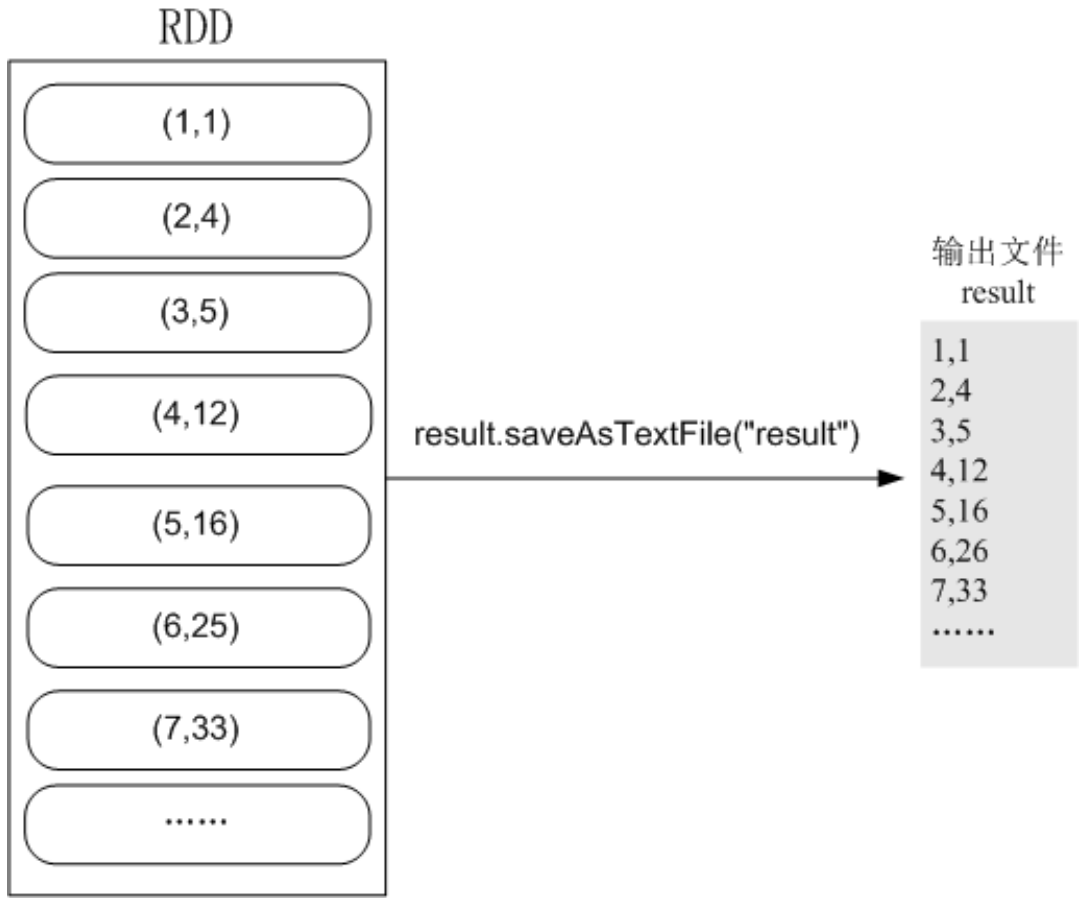
```
result5 = result4.map(lambda x:x[0])
result6 = result5.map(lambda x:(getindex(),x))
```



5.4.2 案例2：文件排序



```
result6.saveAsTextFile("file:///usr/local/spark/mycode/rdd/filesort/sortresult")
```



5.4.3 案例3：二次排序



任务要求：
对于一个给定的文件（数据如file1.txt所示），请对数据进行排序，首先根据第1列数据降序排序，如果第1列数据相等，则根据第2列数据降序排序

输入文件file1.txt

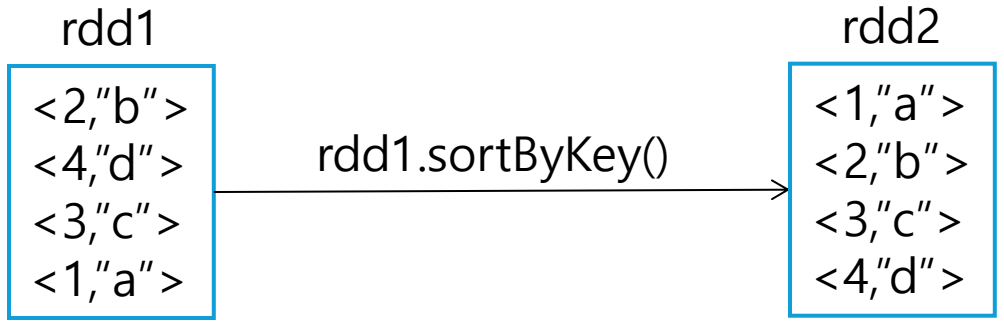
5	3
1	6
4	9
8	3
4	7
5	6
3	2

输出结果

8	3
5	6
5	3
4	9
4	7
3	2
1	6

5.4.3 案例3：二次排序





用于排序的key必须是可比较的对象

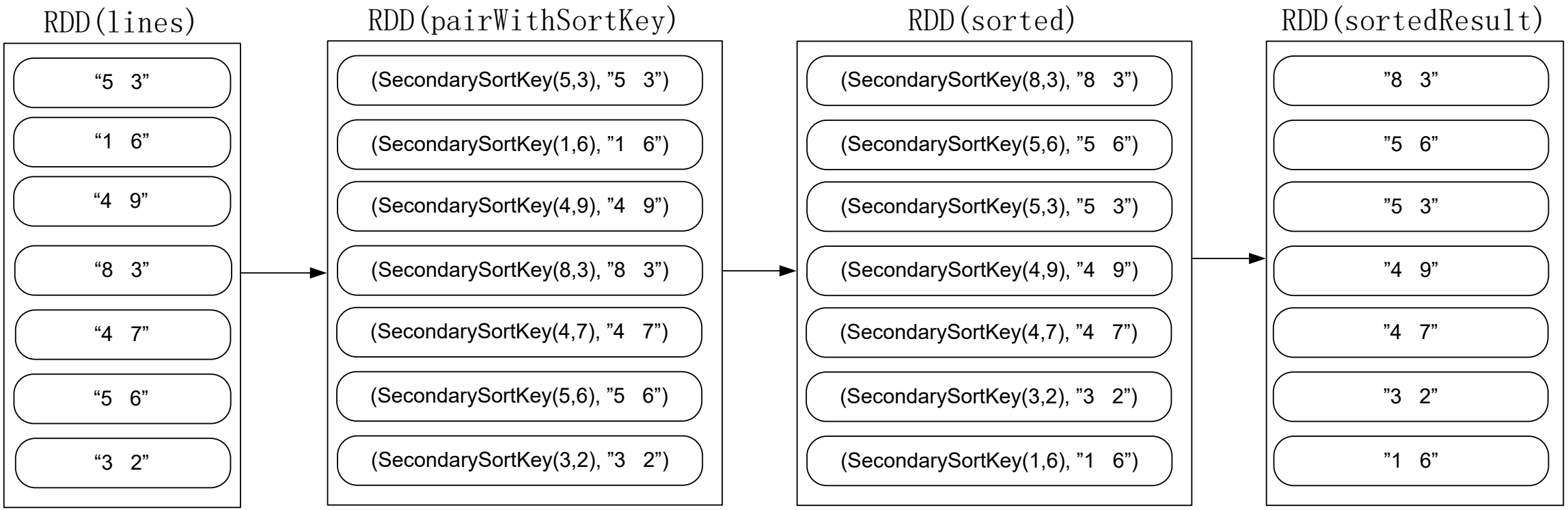
“5 3”
“1 6”
“4 9”
“8 3”
“4 7”
“5 6”
“3 2”

如何比较？

5.4.3 案例3：二次排序



解题思路：



5.4.3 案例3：二次排序



二次排序，具体的实现步骤如下

01 第一步：定义一个类SecondarySortKey，实现自定义的用于排序的key

02 第二步：将要进行二次排序的文件加载进来生成(key,value)类型的RDD

03 第三步：使用sortByKey()基于自定义的key进行二次排序

04 第四步：去除掉排序的key，只保留排序的结果

5.4.3 案例3：二次排序



SecondarySortKey.py代码如下：

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

from operator import gt
from pyspark import SparkContext, SparkConf

class SecondarySortKey():
    def __init__(self, k):
        self.column1 = k[0]
        self.column2 = k[1]

    def __gt__(self, other):
        if other.column1 == self.column1:
            return gt(self.column2,other.column2)
        else:
            return gt(self.column1, other.column1)
```

剩余代码见下一页

5.4.3 案例3：二次排序



SecondarySortApp.py代码如下：

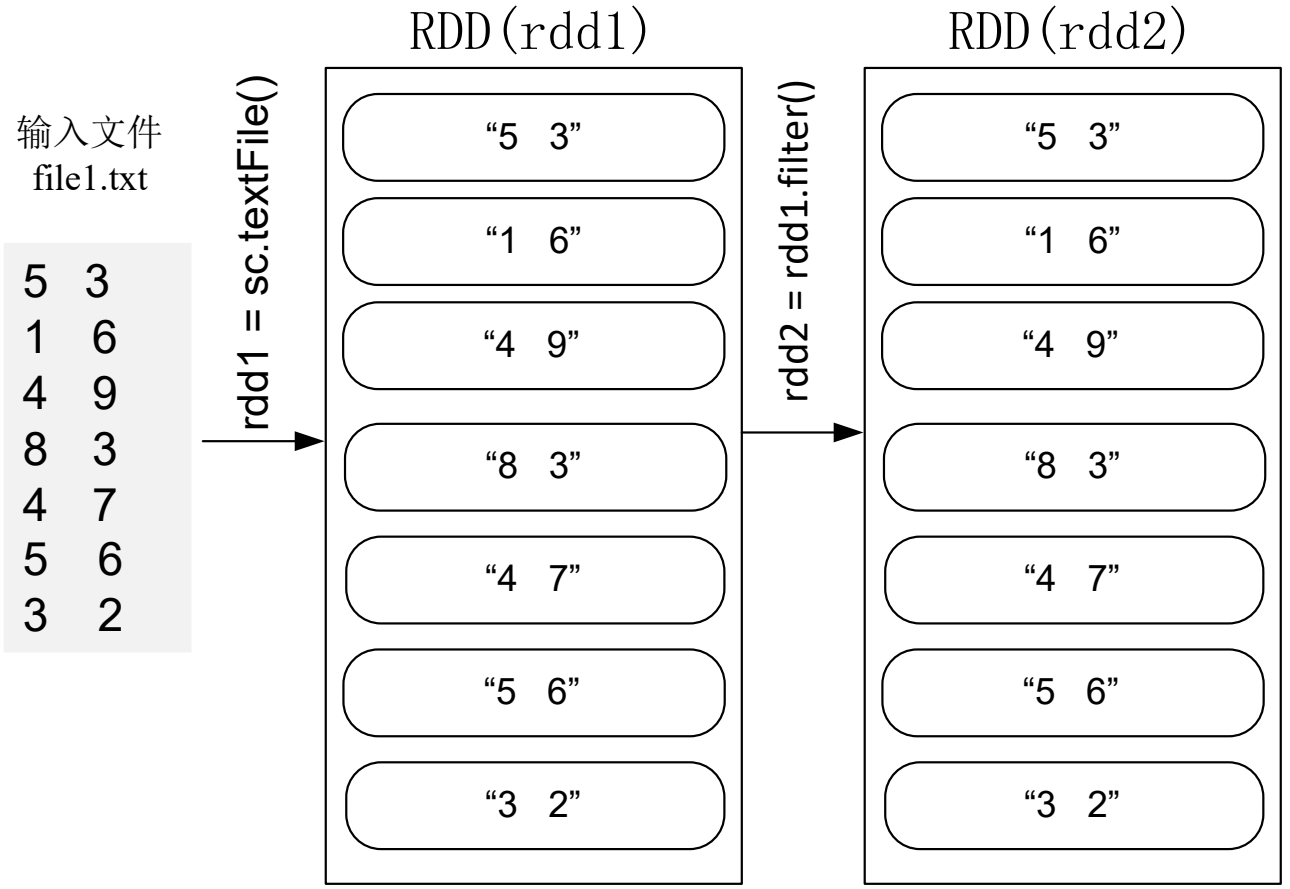
```
def main():
    conf = SparkConf().setAppName('spark_sort').setMaster('local[1]')
    sc = SparkContext(conf=conf)
    file="file:///usr/local/spark/mycode/rdd/secondarysort/file4.txt"
    rdd1 = sc.textFile(file)
    rdd2 = rdd1.filter(lambda x:(len(x.strip()) > 0))
    rdd3 = rdd2.map(lambda x:((int(x.split(" ")[0]),int(x.split(" ")[1])),x))
    rdd4 = rdd3.map(lambda x: (SecondarySortKey(x[0]),x[1]))
    rdd5 = rdd4.sortByKey(False)
    rdd6 = rdd5.map(lambda x:x[1])
    rdd6.foreach(print)

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

5.4.3 案例3：二次排序



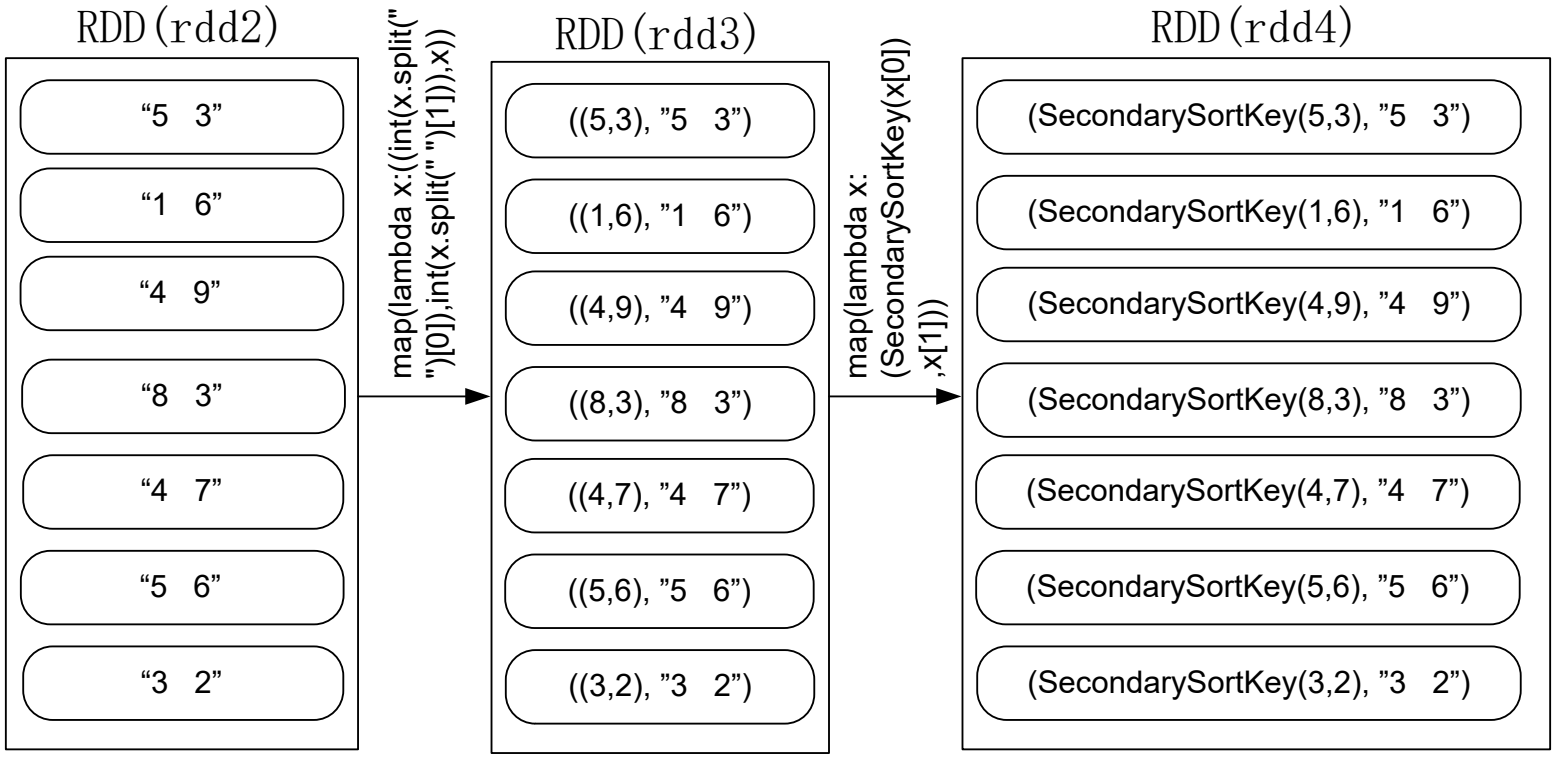
```
rdd1 = sc.textFile(file)
rdd2 = rdd1.filter(lambda x:(len(x.strip()) > 0))
```



5.4.3 案例3：二次排序



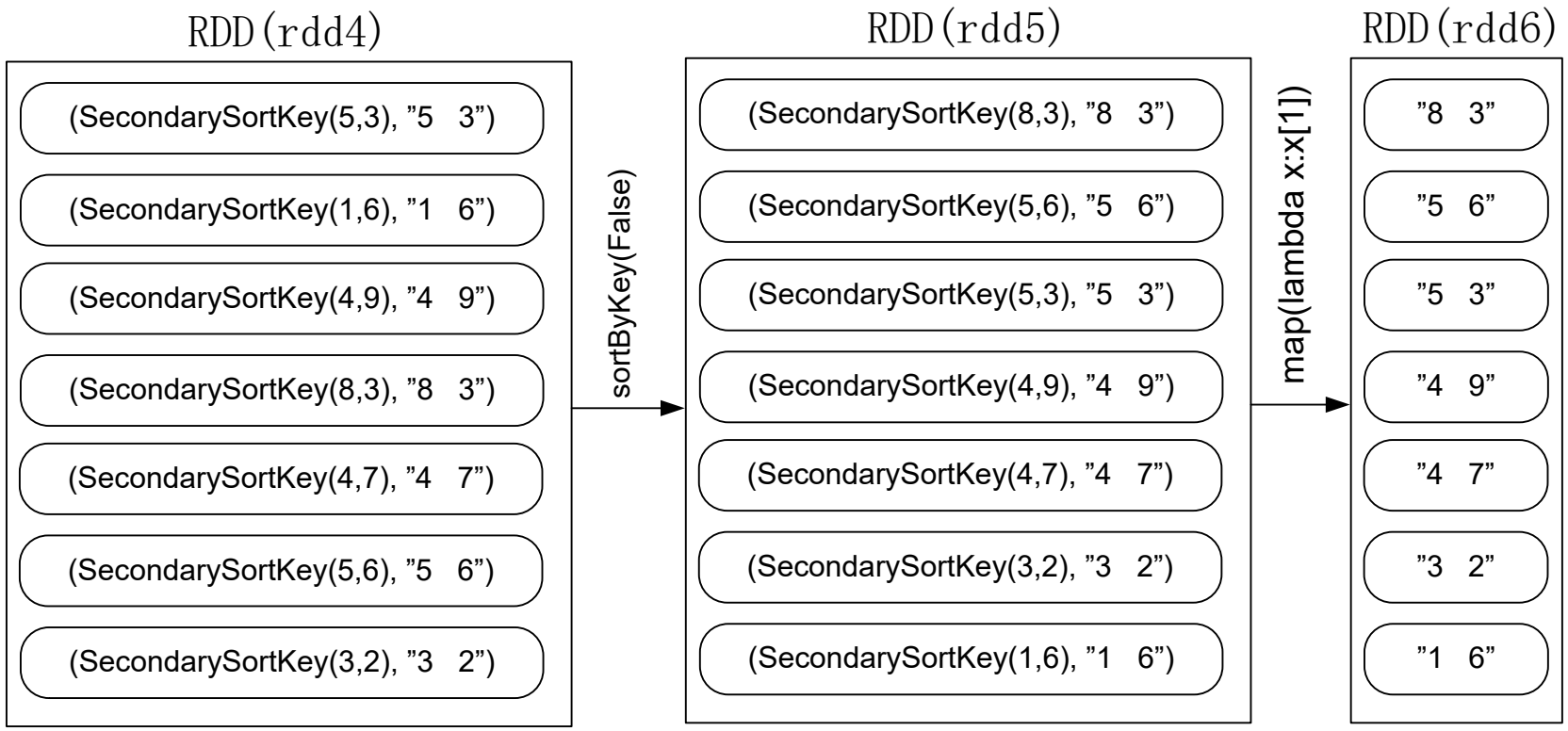
```
rdd3 = rdd2.map(lambda x:((int(x.split(" ")[0]),int(x.split(" ")[1])),x))
rdd4 = rdd3.map(lambda x: (SecondarySortKey(x[0]),x[1]))
```



5.4.3 案例3：二次排序



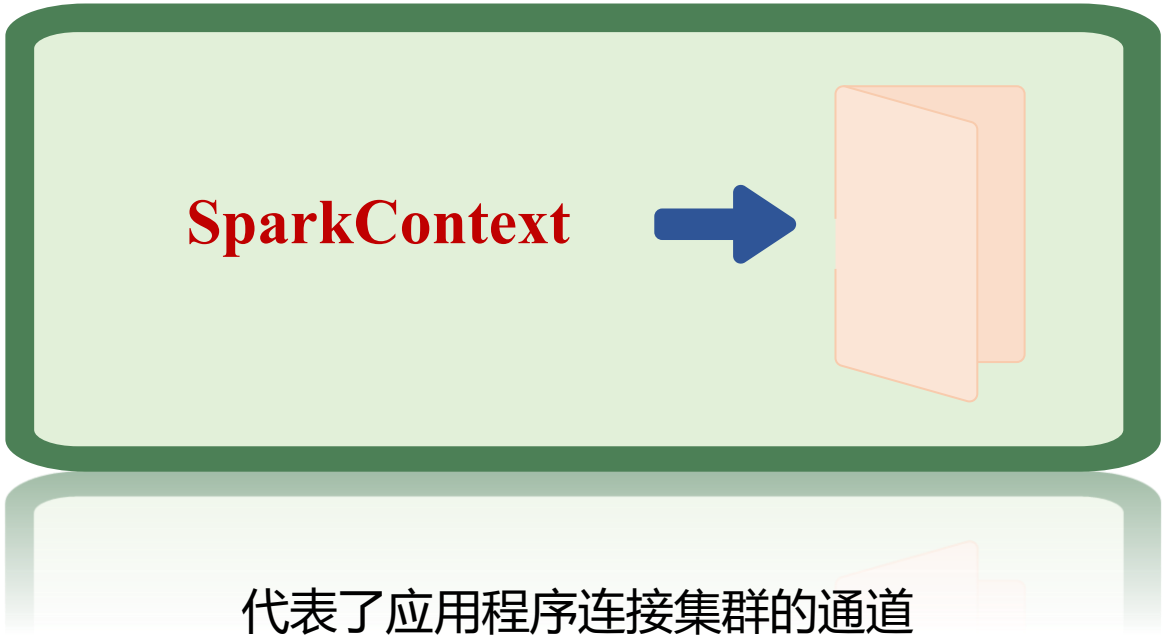
```
rdd5 = rdd4.sortByKey(False)
rdd6 = rdd5.map(lambda x:x[1])
```



5.5 本章小结

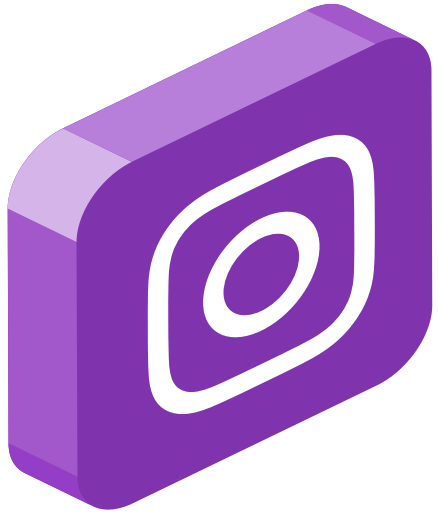
- 本章介绍了RDD编程基础知识，主要是RDD各种操作API的使用，无论多复杂的Spark应用程序，最终都是借助于这些RDD操作来实现的。
- RDD编程都是从创建RDD开始的，我们可以通过多种方式创建得到RDD，比如，从本地文件或者分布式文件系统HDFS中读取数据创建RDD，或者使用parallelize()方法从一个集合中创建得到RDD。
- 创建得到RDD以后，就可以对RDD执行各种操作，包括转换操作和行动操作，本章通过多个实例详细介绍了每个操作API的使用方法。另外，通过持久化，可以把RDD保存在内存或者磁盘中，避免多次重复计算。通过对RDD进行分区，不仅可以增加程序并行度，而且在一些应用场景中可以降低网络通信开销。
- 键值对RDD是一种常见的RDD类型，在Spark编程中经常被使用，本章介绍了键值对RDD的各种操作，并给出了一个综合实例。
- 此外，本章还介绍了文件数据读写和MySQL数据库读写的方法，最后，给出了3个综合实例。

2.3.2 Spark运行基本流程



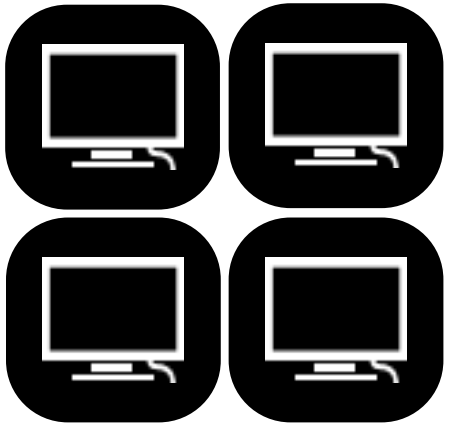
代表了应用程序连接集群的通道

2.3.2 Spark运行基本流程



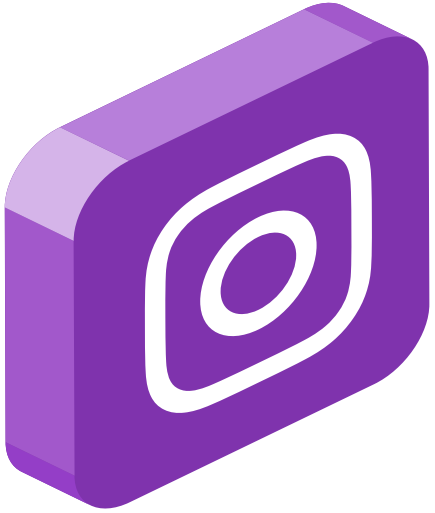
应用



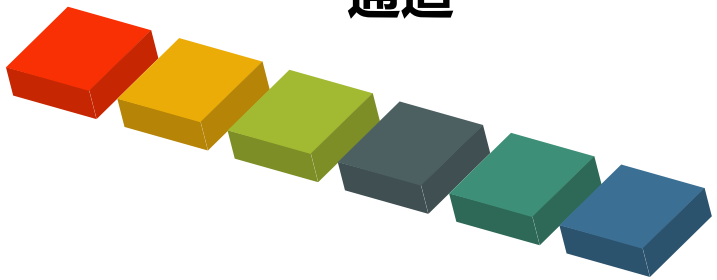


集群

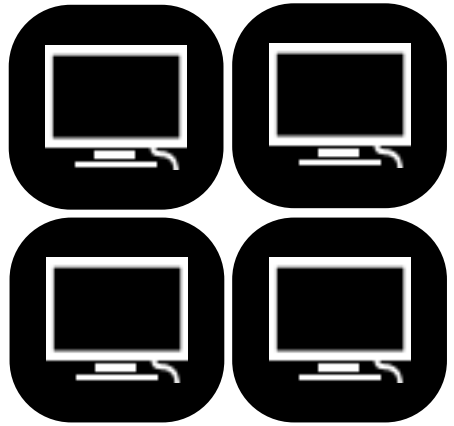
2.3.2 Spark运行基本流程



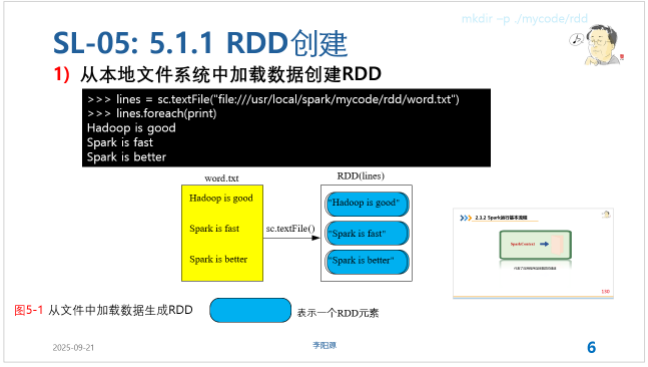
应用



通道



集群



2025-11-11

실험시 에러 발생하면

1. SSH 호스트 키 재생성

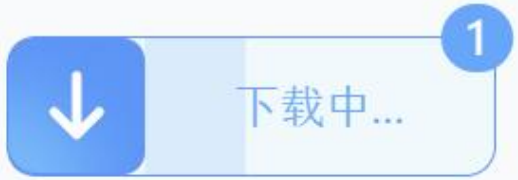
bash

Copy Download

```bash
# 기존 SSH 호스트 키 백업 및 삭제
sudo mv /etc/ssh/ssh_host_* /tmp/

# 새로운 SSH 호스트 키 생성
sudo dpkg-reconfigure openssh-server

# 또는 수동으로 생성
sudo ssh-keygen -A
```



실험시 에러 발생하면

1. 이벤트 로그 디렉토리 생성 (가장 빠른 해결책)

bash Copy Download

```
sudo mkdir -p /tmp/spark-events
sudo chmod 777 /tmp/spark-events  # 모든 사용자에게 권한 부여
```

2. Spark 설정 변경 (영구적 해결)

bash Copy Download

```
# Spark 기본 설정 파일 수정
sudo nano /usr/local/spark/conf/spark-defaults.conf
```

다음 내용 추가:

properties Copy Download

```
spark.eventLog.enabled false
spark.history.fs.logDirectory /tmp/spark-events  # (옵션) 다른 경로 지정 시
```

什么是lambda函数?



Lambda函数是没有名字的简单函数

- 可以把它看作“一次性小型函数”
- 不使用`def`关键字定义函数，而是简单表达
- Lambda这个名称来源于数学中的概念

普通函数 vs Lambda函数对比



定义普通函数

```
def add(x, y):
    return x + y

result = add(3, 5)
print(result)  # 输出: 8
```

普通函数 vs Lambda函数对比



使用Lambda函数定义

```
add = lambda x, y: x + y
result = add(3, 5)
print(result)  # 输出: 8
```

Lambda函数基本结构



lambda 参数1, 参数2, ... ：表达式

Lambda函数的特点

1.**没有名称**（匿名函数）

2.**可以一行编写**

3.**即使没有return语句**也会返回结果

4.只用于**简单功能**

Lambda函数的真正威力：与其他函数配合使用



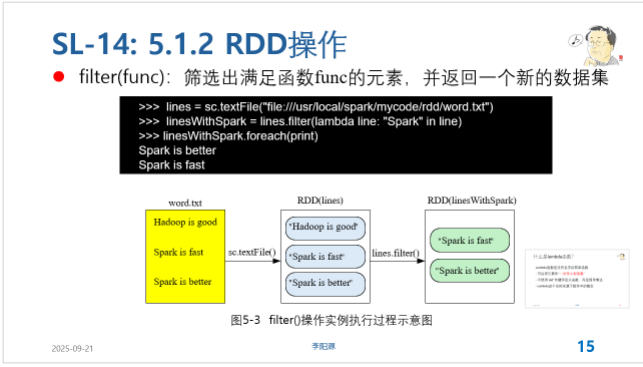
Lambda函数与map()、filter()、sorted()等函数一起使用时能发挥真正价值

与map()函数一起使用

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

squared = list(map(lambda x: x**2, numbers))

print(squared)  # 输出: [1, 4, 9, 16, 25]
```





data

[1,2,3,4]

sc.parallelize(data,2)

RDD(rdd1)

[1,2]

[3,4]

rdd1.mapPartitions(myFunction)

分句1: [1, 2] → [2, 4]
分句2: [3, 4] → [6, 8]

RDD(rdd2)

[2, 4]

[2, 4]

```
def myFunction(iter):
        result = list()
        print("==============")
        for it in iter:
            result.append(it*2)
        return result
```



Spark reduceByKey() 工作原理



输入数据结构

```
words = sc.parallelize([
    ("Hadoop", 1), ("is", 1),
("good", 1),
    ("Spark", 1), ("is", 1), ("fast",
1),
    ("Spark", 1), ("is", 1),
("better", 1)
])
```

0 初始数据 (RDD)

('Hadoop', 1) ('is', 1) ('good', 1)
('Spark', 1) ('is', 1) ('fast', 1)
('Spark', 1) ('is', 1)
('better', 1)

1 分区 (按键分组)

将相同键的数据分组到同一分区

分区 1
('Hadoop', 1) ('good', 1)
('fast', 1)

分区 2
('is', 1) ('is', 1) ('is', 1)

分区 3
('Spark', 1) ('Spark', 1)
('better', 1)

2 本地聚合 (分区内聚合)

在每个分区内先应用lambda函数

聚合函数: lambda a, b: a + b

分区 1
('Hadoop', 1) ('good', 1)
('fast', 1)
↓ 本地聚合
('Hadoop', 1), ('good', 1), ('fast', 1)

分区 2
('is', 1) + ('is', 1) +
('is', 1)
↓ 本地聚合
('is', 3)

分区 3
('Spark', 1) + ('Spark', 1)
('better', 1)
↓ 本地聚合
('Spark', 2), ('better', 1)



3 最终聚合 (全局合并)

将所有分区的结果合并计算最终值

各分区本地聚合结果

分区1: ('Hadoop',1), ('good',1), ('fast',1)

分区2: ('is', 3)

分区3: ('Spark',2), ('better',1)

⬇️

最终合并结果

('Hadoop', 1) ('is', 3)

('good', 1) ('Spark', 2)

('fast', 1) ('better', 1)

lambda函数示例说明

示例1: ('is', 1) + ('is', 1) + ('is', 1) → ('is', 3)
示例2: ('Spark', 1) + ('Spark', 1) → ('Spark', 2)

工作原理： lambda函数 `lambda a, b: a + b` 将两个值相加来计算累积和。这个过程对所有相同键的数据重复执行，最终产生聚合结果。

性能优势：

- 网络传输优化：先进行本地聚合，减少数据传输量
- 并行处理：各分区同时进行聚合操作
- 内存高效：最小化中间结果存储



PySpark sortBy() 实际示例分析



初始数据 / Initial Data

myList=[3,1,2,4] →
rdd1=sc.parallelize(myList,1)

3 1 2 4

分区数 / Partitions: 1 (所有数据在同一分区 / All data in one partition)

Lambda函数分析 / Lambda Function Analysis

lambda x: x

中文: 恒等函数，返回元素本身作为排序键
English: Identity function, returns the element itself as sort key

键提取过程 / Key Extraction Process

对每个元素应用 lambda x: x / Apply lambda x: x to each element

3 → 3　1 → 1　2 → 2　4 → 4

两种排序方式比较 / Comparison of Two Sorting Methods

rdd2 = rdd1.sortBy(lambda x:x)

升序排序 / Ascending Sort
(默认 ascending=True / Default ascending=True)

原始数据 / Original Data

3 1 2 4

↓ 升序排序 / Ascending Sort

排序后 / After Sorting

1 2 3 4

```
输出结果 / Output:
1
2
3
4
```

rdd3 = rdd1.sortBy(lambda x:x, False)

降序排序 / Descending Sort
(明确指定 ascending=False / Explicitly set ascending=False)

原始数据 / Original Data

3 1 2 4

↓ 降序排序 / Descending Sort

排序后 / After Sorting

4 3 2 1

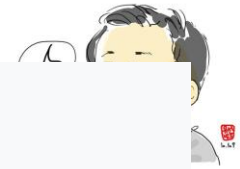
```
输出结果 / Output:
4
3
2
1
```



执行流程分析 / Execution Flow Analysis

代码执行顺序 / Code Execution Order:

1. 创建RDD: rdd1 = sc.parallelize([3,1,2,4], 1)
   → 在单个分区中创建包含4个元素的RDD
   → Create RDD with 4 elements in a single partition
2. 升序排序: rdd2 = rdd1.sortBy(lambda x:x)
   → 使用默认升序排序 / Use default ascending order
3. 输出升序结果: rdd2.foreach(print)
   → 打印: 1, 2, 3, 4
4. 降序排序: rdd3 = rdd1.sortBy(lambda x:x, False)
   → 明确指定降序排序 / Explicitly specify descending order
5. 输出降序结果: rdd3.foreach(print)
   → 打印: 4, 3, 2, 1



关键要点总结 / Key Points Summary

🎯 sortBy() 参数说明 / sortBy() Parameters:

- **keyfunc**: lambda x: x (提取排序键的函数 / Function to extract sort key)
- **ascending**: True(默认升序) / False(降序) / True(default ascending) / False(descending)

☑ 代码特点 / Code Characteristics:

- 使用单分区处理，避免了跨分区洗牌的复杂性
- Single partition processing, avoiding cross-partition shuffle complexity
- lambda x: x 是最简单的恒等函数
- lambda x: x is the simplest identity function
- foreach(print) 直接输出每个元素
- foreach(print) directly outputs each element

🔧 实际应用场景 / Practical Use Cases:

- 数据预处理中的排序需求 / Sorting requirements in data preprocessing
- 生成有序的输出结果 / Generate ordered output results
- 为后续分析准备有序数据 / Prepare ordered data for subsequent analysis

