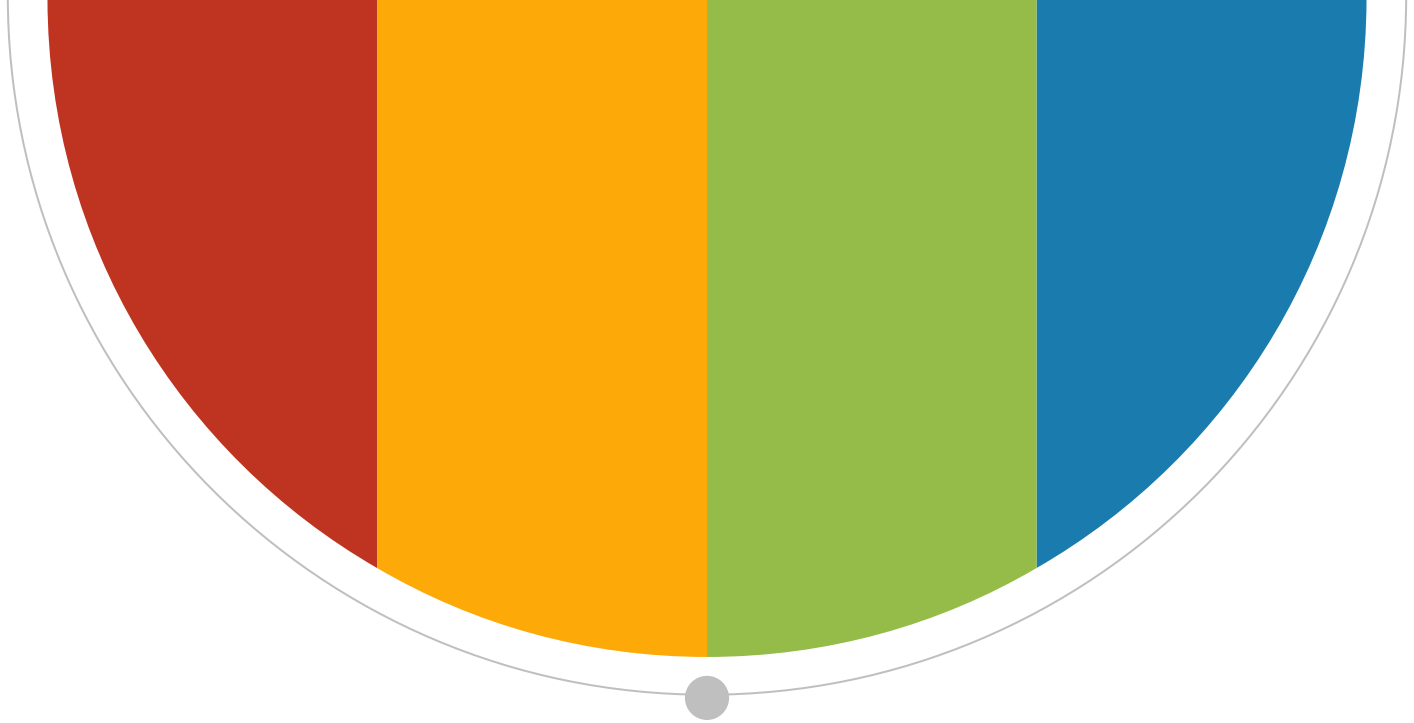


大数据处理与分析：Spark编程实践

王桂玲

北方工业大学信息学院数据工程研究院

大规模流数据集成与分析技术北京市重点实验室

wangguiling@ncut.edu.cn

1 5.1 RDD编程基础

2 5.2 键值对RDD

3 5.3 数据读写

4 5.4 综合案例

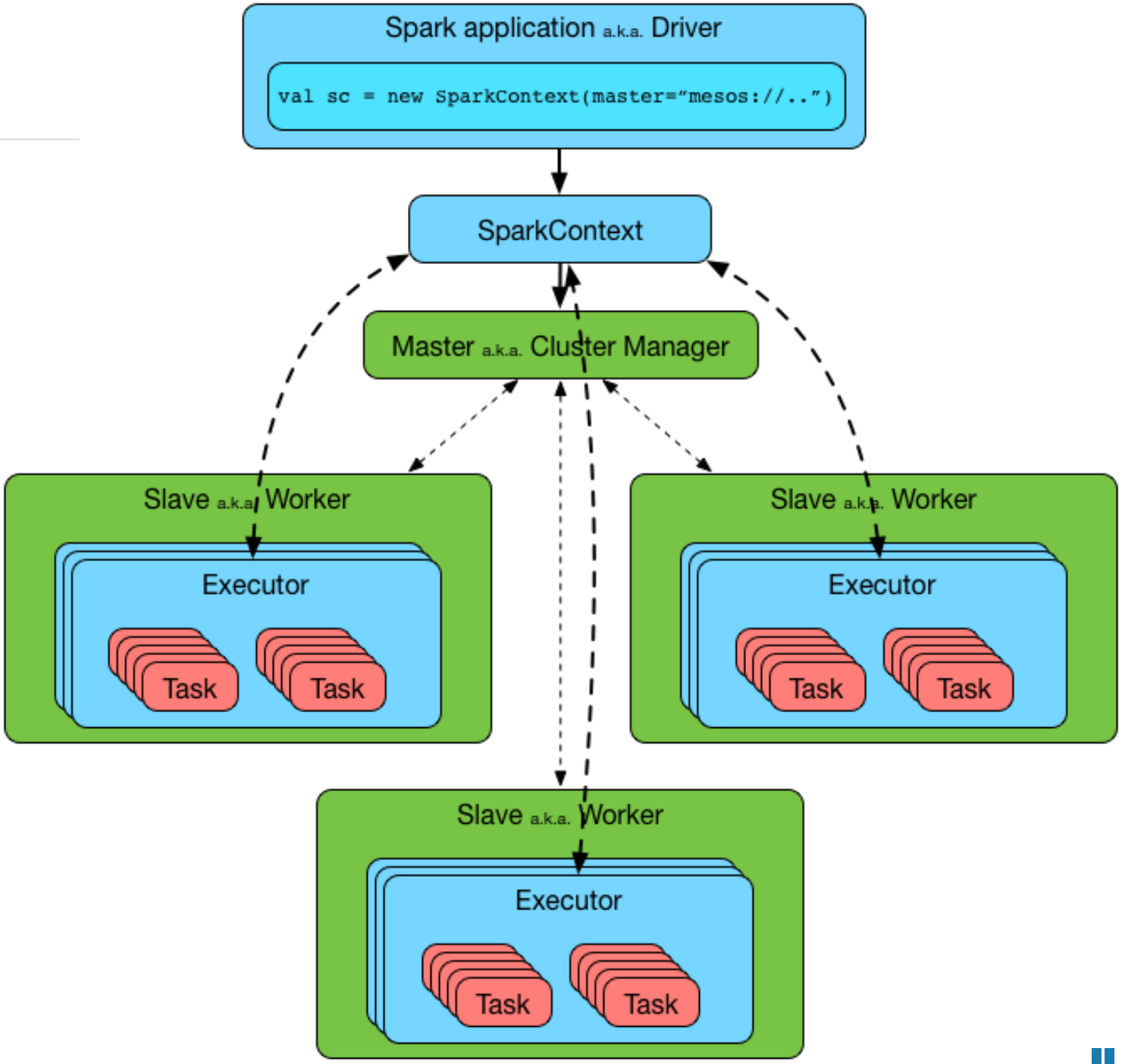
第五章 RDD编程

课程回顾

- 第0章 课程简介
- 第1章 大数据处理与分析技术概述
- 第2章 Python语言基础
- 第3章 Spark的设计与运行原理
- 第4章 Spark环境搭建和使用方法
- 第5章 RDD编程

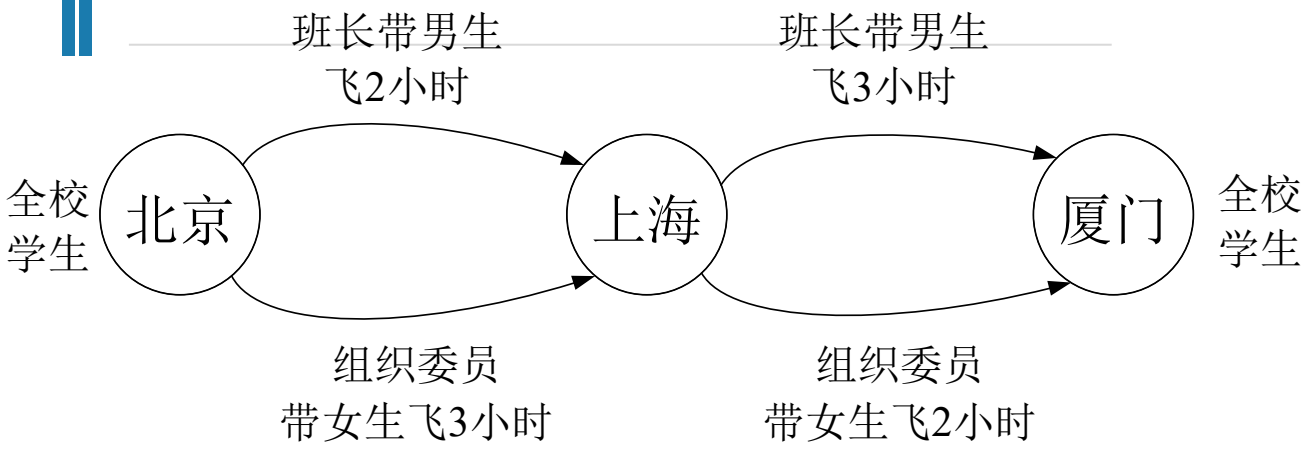
课程回顾

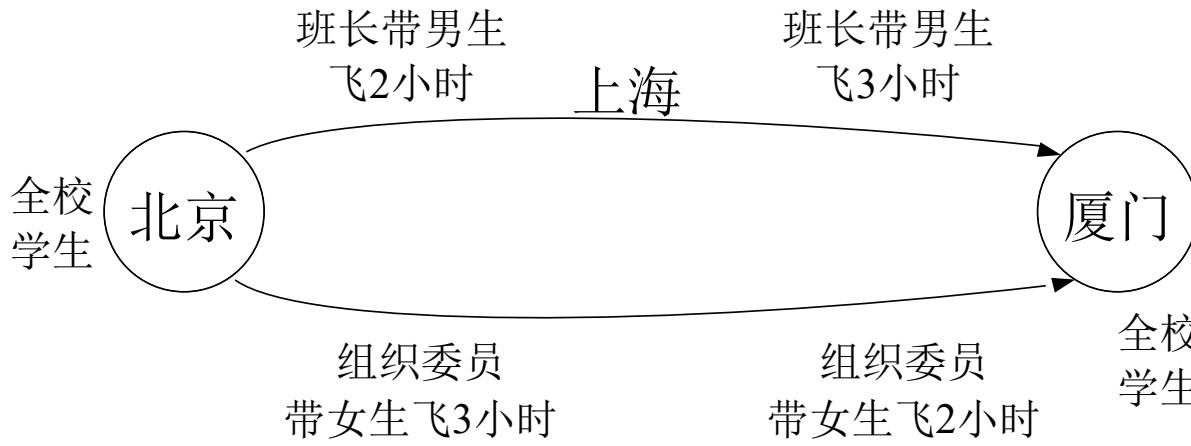
- Spark运行架构包括集群资源管理器（Cluster Manager）、运行作业任务的工作节点（Worker Node）、每个应用的**任务控制节点（Driver）**和每个工作节点上负责具体任务的执行进程（Executor）
- Executor：是运行在工作节点（WorkerNode）的一个进程，负责运行Task
- 应用（Application）：用户编写的Spark应用程序
- 任务（Task）：运行在Executor上的工作单元
- 作业（Job）：一个作业包含多个RDD及作用于相应RDD上的各种操作
- 阶段（Stage）：是作业的基本调度单位，一个作业会分为多组任务，每组任务被称为阶段，或者也被称为任务集合



- Spark有不同的部署模式，下列说法正确的是：
  - Spark的Local部署模式是指单机部署模式，client/driver/worker都在本地执行
  - Spark的Standalone模式使用Spark自带的简单集群管理器
  - YARN模式使用YARN作为集群管理器
  - Mesos模式使用Mesos作为集群管理器

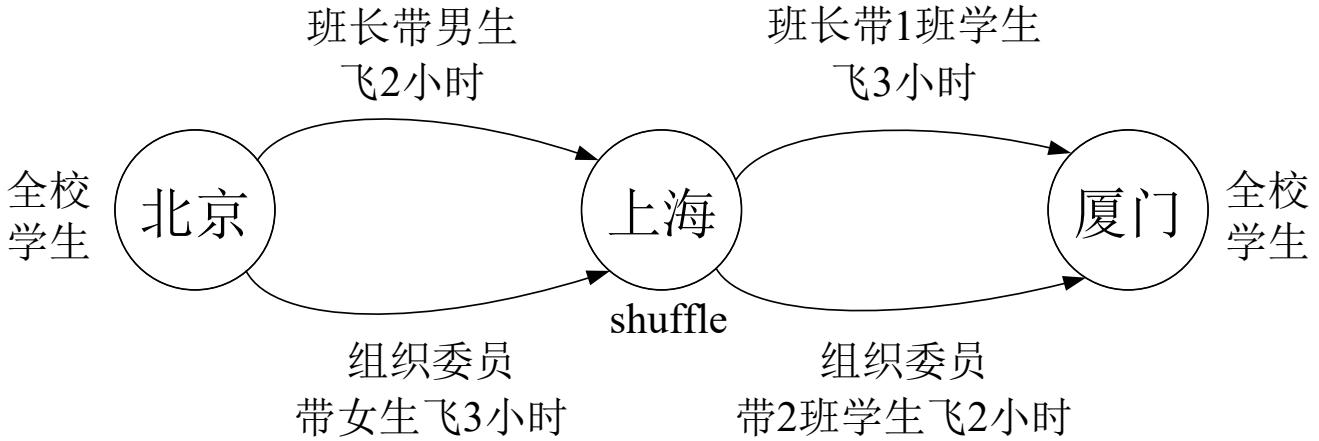
- Spark 根据DAG 图中的RDD 依赖关系，把一个作业分成多个阶段。其中，窄依赖表现为一个父RDD的分区对应于一个子RDD的分区或多个父RDD的分区对应于一个子RDD的分区；宽依赖则表现为存在一个父RDD的一个分区对应一个子RDD的多个分区。下面阶段划分的依据说法正确的是：
  - A. 阶段划分的依据是窄依赖和宽依赖，遇到宽依赖就断开，遇到窄依赖就把当前的RDD加入到Stage中
  - B. 阶段划分的依据是RDD的操作类型（Action行动和Transformation转换），行动和转换在不同的阶段
  - C. 在DAG中将窄依赖尽量划分在同一个Stage中，可以实现流水线计算
  - D. 每个阶段的任务集是由相互之间没有Shuffle依赖关系的任务组成的
  - E. 窄依赖无法实现“流水线”优化，宽依赖可以实现“流水线”优化





（a）窄依赖

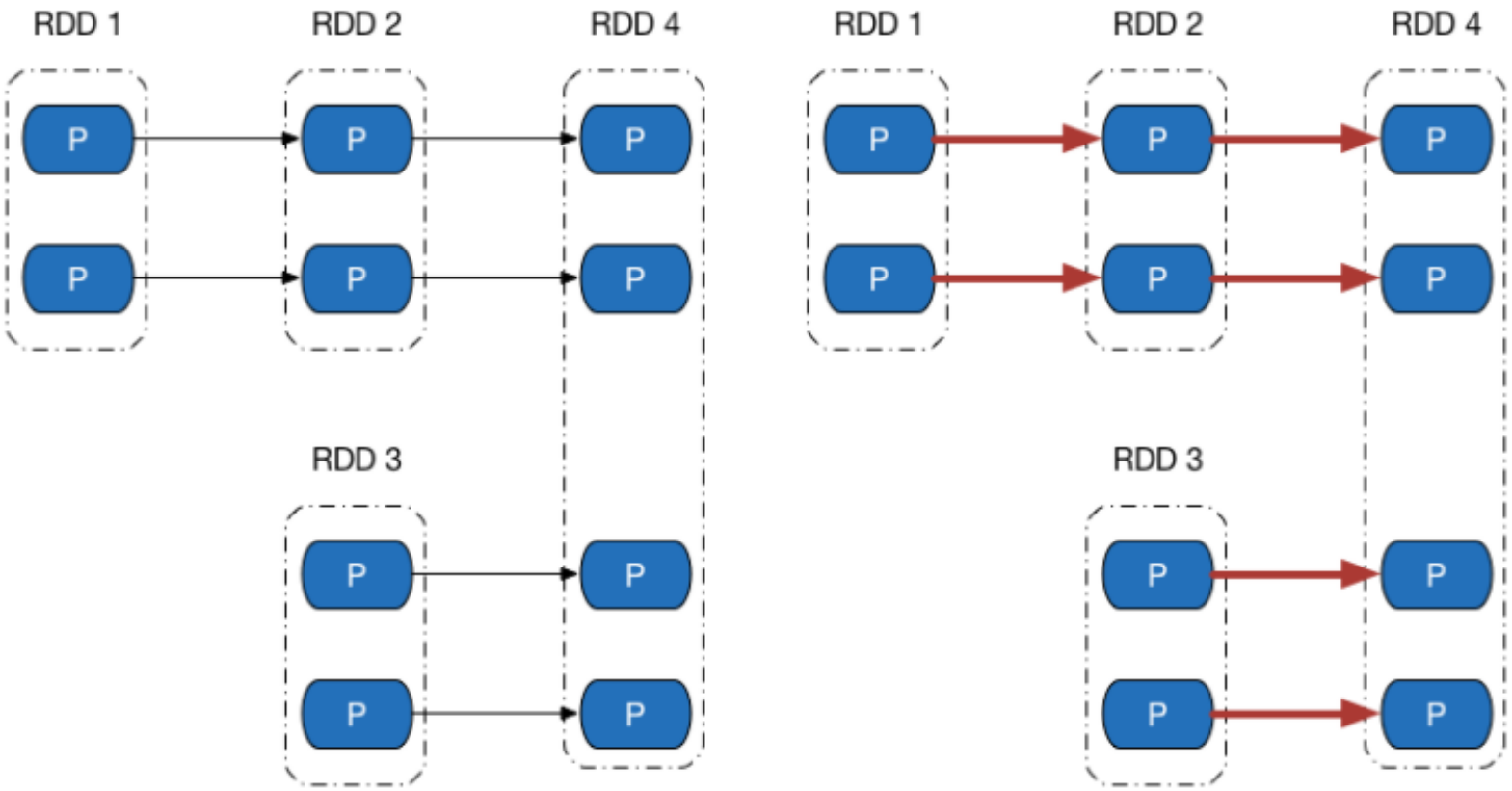
- a班长带男生从北京-上海和上海-厦门可以放到同一task中，和组织委员带女生从北京-上海-厦门可以放到两个task中并行进行，两个任务都是流水线作业
- b班长带1班学生必须等待男生北京-上海和女生北京-上海完成后再进行，**人员在上海要进行不同队伍中交换（shuffle），**无法作为两个并行任务流水线作业
- 父RDD：学生（北京-上海）；子RDD：学生（上海-厦门）
- 将男生/女生看作不同分区；1班/2班看作不同分区
- (a) 男生（上海-厦门）对应男生（北京-上海），也即父RDD的一个分区对应子RDD的一个分区=》窄依赖
- (b) 1班 (上海-厦门)对应男生（北京-上海）和女生（北京-上海），也即父RDD的一个分区对应子RDD的多个分区=》宽依赖



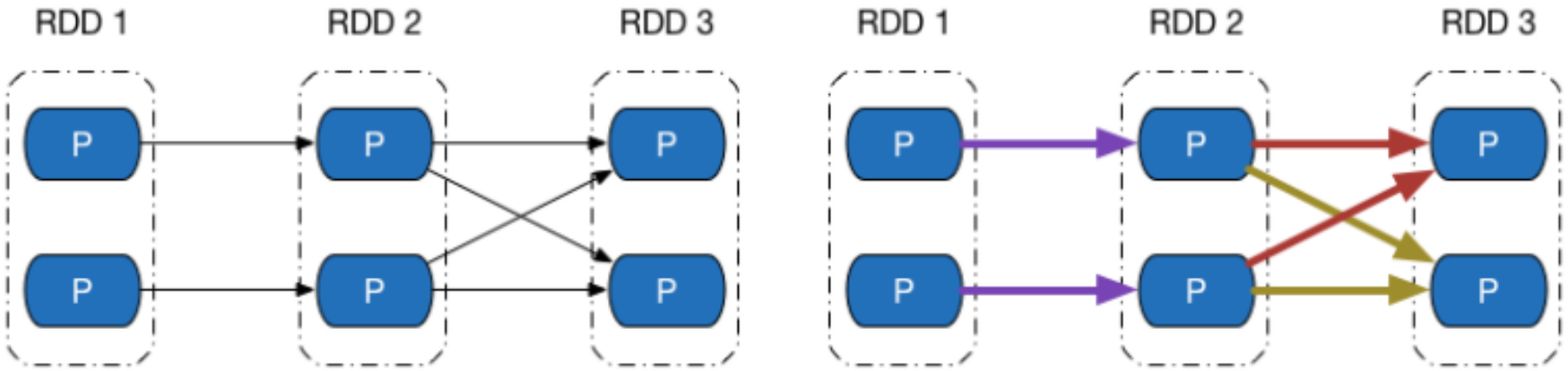
（a）宽依赖

窄依赖的流水线操作

- 右图不同分区上都是一对一依赖，其上计算任务独立互不干扰
- 可以把 RDD 每个分区内数据的计算合并为一个并行任务交付给一个节点上的一个 CPU 核心去执行



宽依赖无法实现流水线操作



- 父RDD2一个分区的数据要分割给子RDD3的两个分区
- 当我们想计算末 RDD 中一个分区的数据时，需要把父 RDD 所有分区的数据计算出来，而这些数据，在计算末 RDD 另外一个分区的数据时候，同样会被用到
- 若想并行，要么父 RDD 的数据在每次需要使用的时候都重复计算一遍；要么想办法把父 RDD 数据保存起来，提供给其余分区的数据计算使用
- Spark采用后面一种方法，**Spark 把整个计算作业从宽依赖处断开**，划分成不同的**阶段（Stage）**
- 阶段之内每个分区的计算可以合并在一个节点上并行执行；阶段之间存在宽依赖，由于上一阶段的数据要给下一阶段的多个分区使用，因此存在shuffle操作

PART ONE

5.1 RDD编程基础

One can never be overeducated.

5.1 RDD编程基础

- 5.1.1 RDD创建
- 5.1.2 RDD操作
- 5.1.3 持久化
- 5.1.4 分区
- 5.1.5 一个综合实例

There's no shame in not knowing things! The only shame is to pretend that we know everything.

Richard Feynman

RDD的基本概念

- RDD（Resilient Distributed Dataset）叫做弹性分布式数据集。是Spark中最基本也是最根本的数据抽象。
  - a. 它是分布式的，可以分布在多台机器上，进行并行的计算。
  - b. 它是弹性的，计算过程中内存不够时它会和磁盘进行数据交换。
- RDD支持常见的MapReduce操作，同时提供了比MR更丰富的操作API
- RDD编程的思想以MapReduce编程模型为基础，可看做MapReduce编程模型的一种扩展

*RDD：是Resillient Distributed Dataset（弹性分布式数据集）的简称，RDD是一种有容错机制的特殊集合，可以**分布**在集群的节点上，以函数式编操作集合的方式，进行各种**并行操作**。可以将RDD理解为一个具有**容错机制**的特殊集合，它提供了一种只读、只能由已存在的RDD变换而来的共享内存，然后将所有数据都**加载到内存**中，方便进行多次重用。*

5.1.1 RDD创建

- 1. 从文件系统中加载数据创建RDD
- 2. 通过并行集合（列表）创建RDD

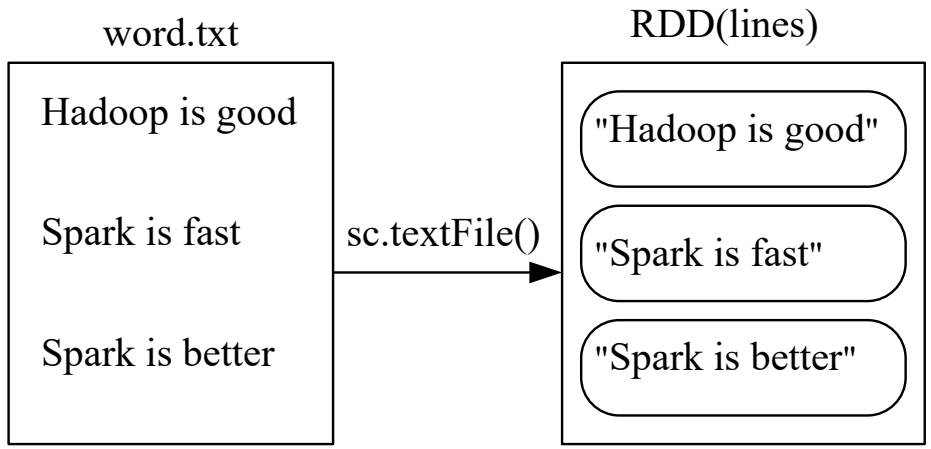
5.1.1 RDD创建

- **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
  - 或者是分布式文件系统HDFS的地址
  - 或者是Amazon S3的地址等等

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
```



一个RDD元素是文件中的一行内容

从文件中加载数据生成RDD

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

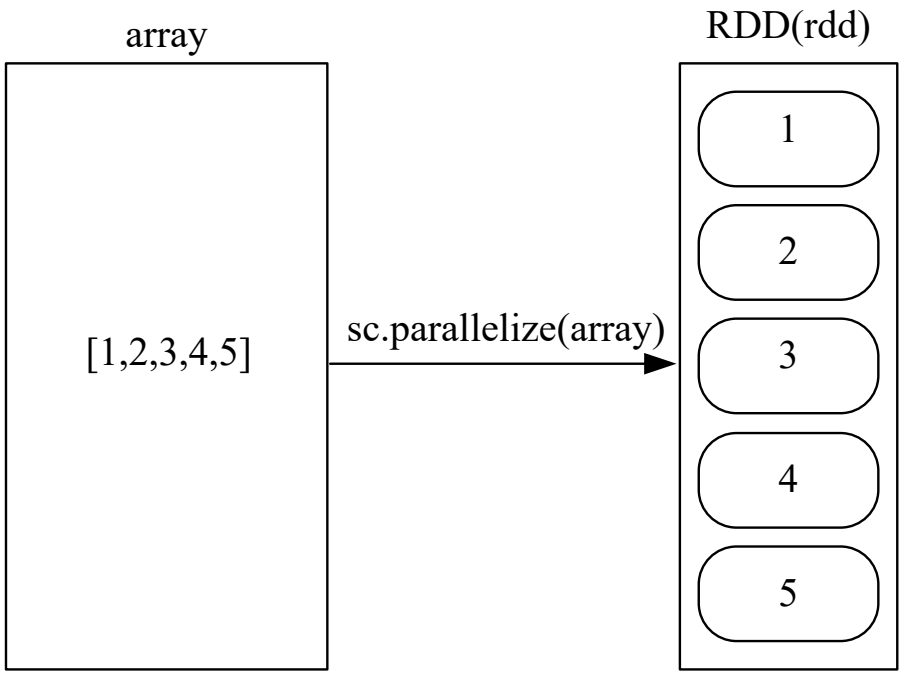
三条语句是完全等价的，可以使用其中任意一种方式

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
```



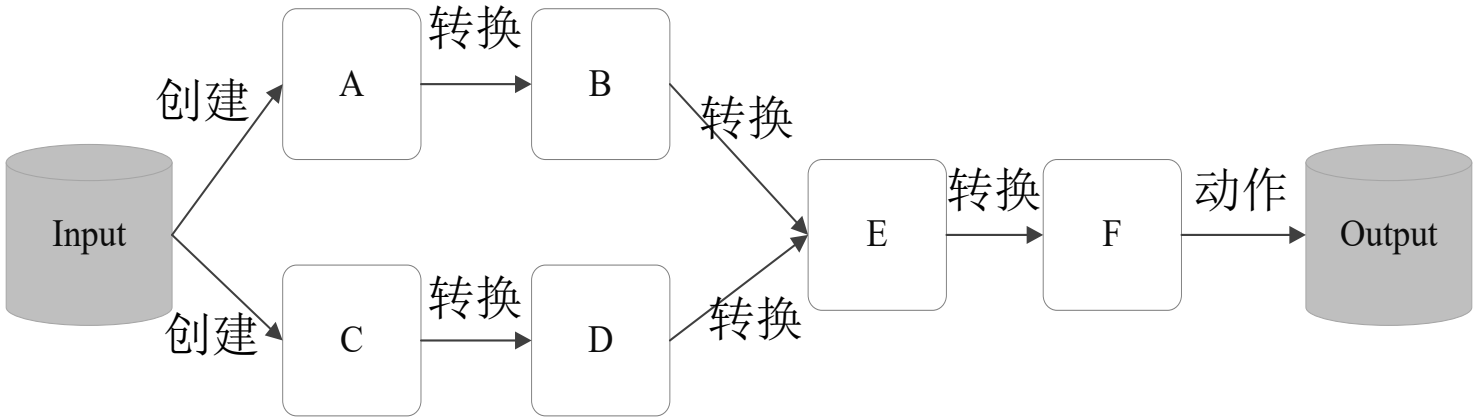
从列表创建RDD示意图

5.1.2 RDD操作

- 1. 转换操作
- 2. 行动操作
- 3. 惰性机制

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



RDD惰性求值的好处

- 避免不必要的计算：如果一个程序没有action呢?
- 节省存储空间
  - 避免存储转换操作的中间结果，例如，筛选出某个文件中包含“spark”的行，最后只输出第一个符合条件的行。我们先读取文件textFile()生成RDD1，然后使用filter()方法生成RDD2，最后是行动操作first()。试想，如果读取文件的时候就把所有的行都存储起来，而事实上只需要存储第一行，就太浪费存储空间了
- 将转换操作合并到一起来减少计算数据的步骤，提高执行效率
  - 从DAG图整体进行分析，进行阶段划分、计算优化
  - Result 产生的地方的就是要计算的位置，要确定 “需要计算的数据”，我们可以从后往前推，需要哪个 partition 就计算哪个 partition，如果 partition 里面没有数据，就继续向前推，形成 computing chain（或看为流水线）。

在类似 Hadoop MapReduce 的系统中，开发者常常花费大量时间考虑如何把操作组合到一起，以减少 MapReduce 的周期数。而在 Spark 中，写出一个非常复杂的映射并不见得能比使用很多简单的连续操作获得好很多的性能。因此，用户可以用更小的操作来组织他们的程序，这样也使这些操作更容易管理。

只低头做事，不抬头看路 vs.
低头走路，脚踏实地，抬头看路，关注全局

5.1.2 RDD操作

- 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
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

5.1.2 RDD操作

- **1. 转换操作**
- 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
```

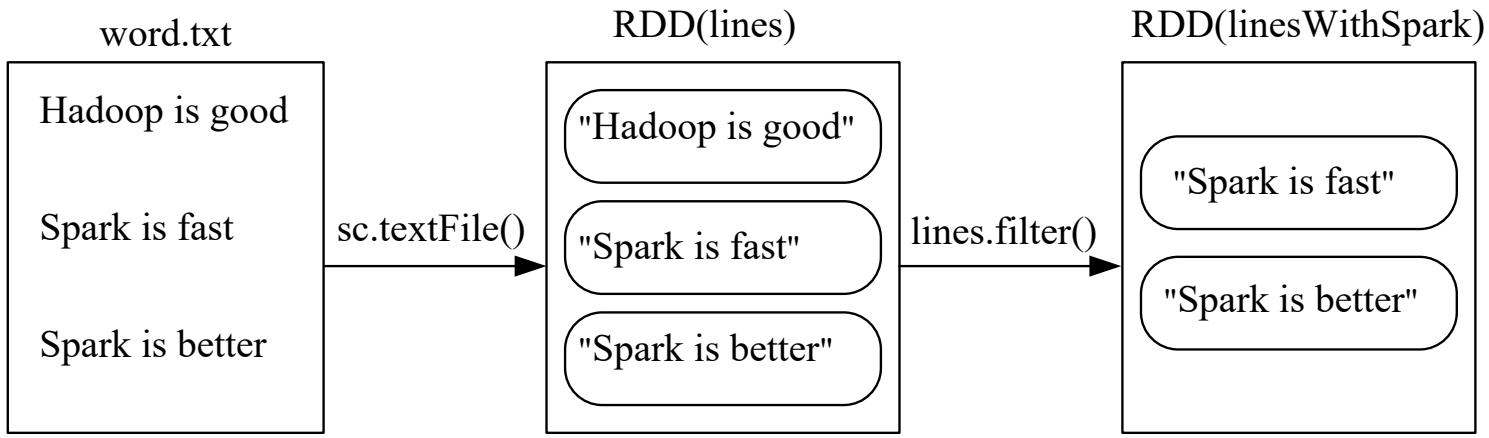


图 filter()操作实例执行过程示意图

5.1.2 RDD操作

- **1. 转换操作**
- 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
```

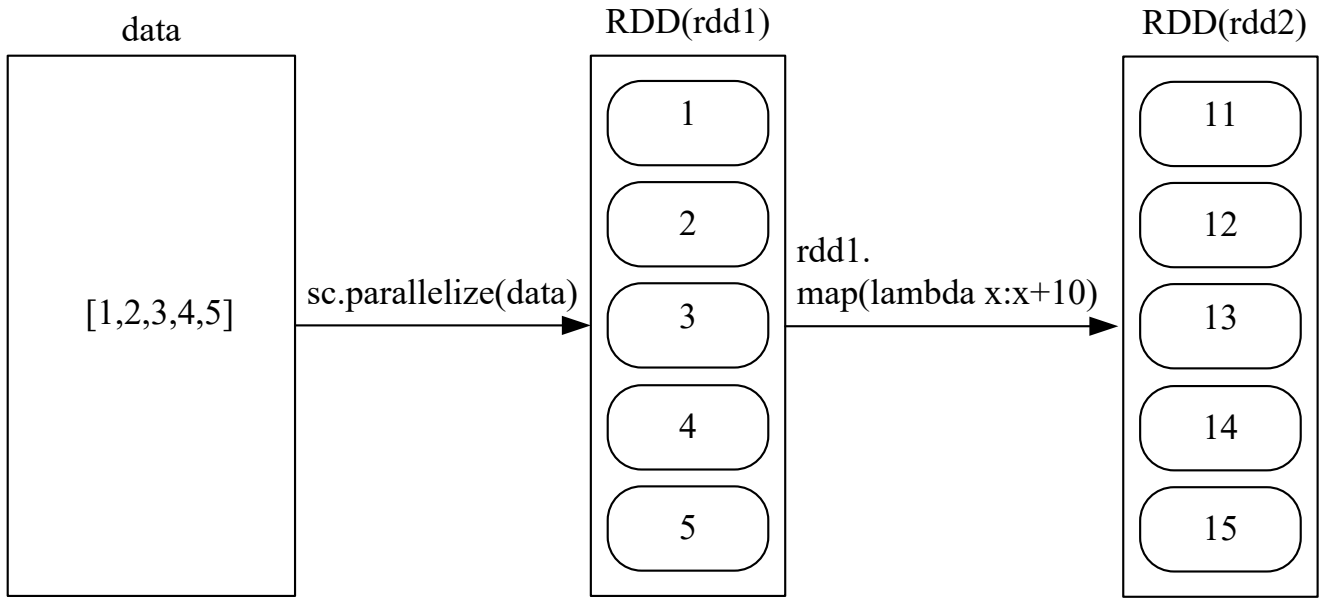


图 map()操作实例执行过程示意图

5.1.2 RDD操作

- **1. 转换操作**
- 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']
```

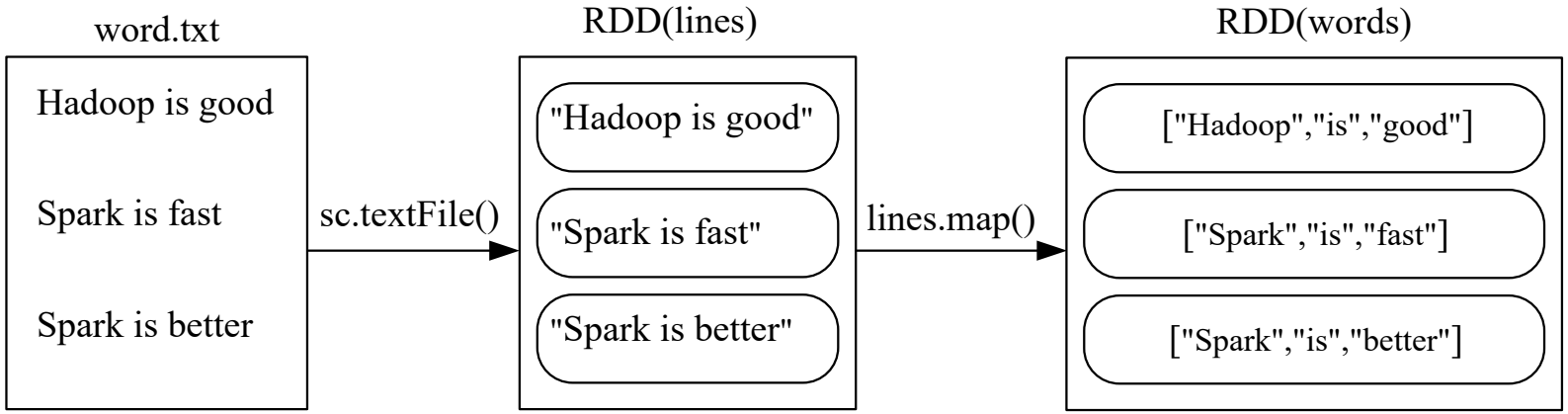
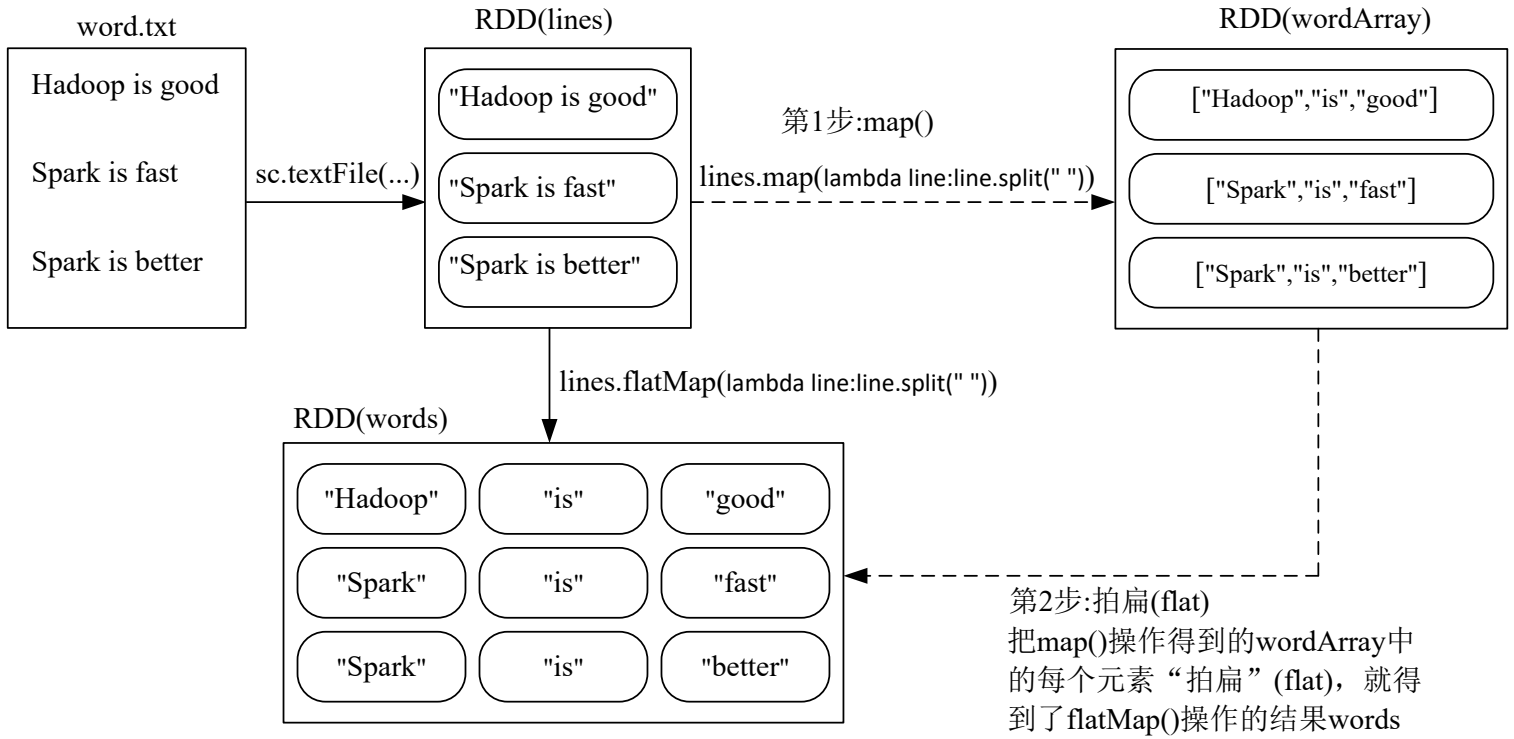


图 map()操作实例执行过程示意图

5.1.2 RDD操作

- **转换操作**
- flatMap(func)

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



理解map和flatmap

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.Map(lambda x: [x, x+5])
RDD: [1, 2, 3] → [[1, 6], [2, 7], [3, 8]]

>>> rdd.flatMap(lambda.x: [x, x+5])
   RDD: [1, 2, 3] → [1, 6, 2, 7, 3, 8]
```

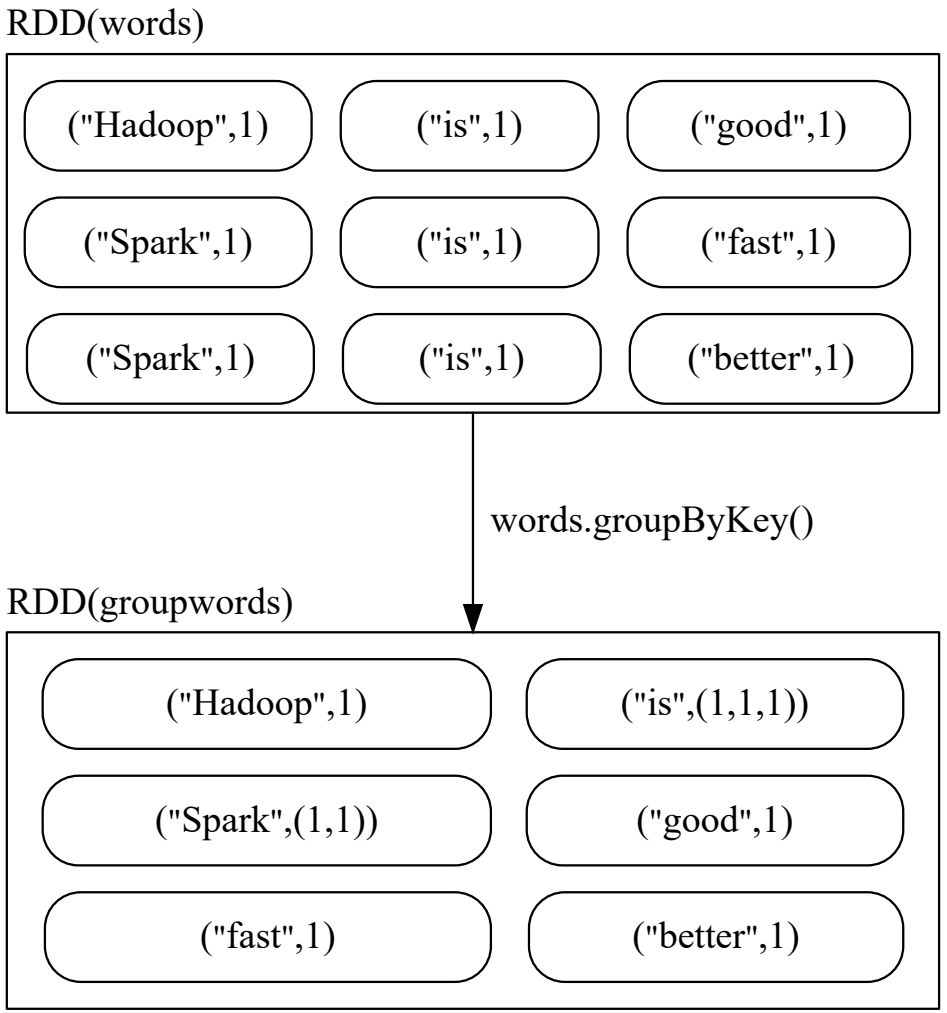
5.1.2 RDD操作

- 转换操作
- groupByKey()
- groupByKey()应用于(K,V)键值对的数据集时，返回一个新的[key,Iterable[value]]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>)
```

5.1.2 RDD操作

- 转换操作
- groupByKey()



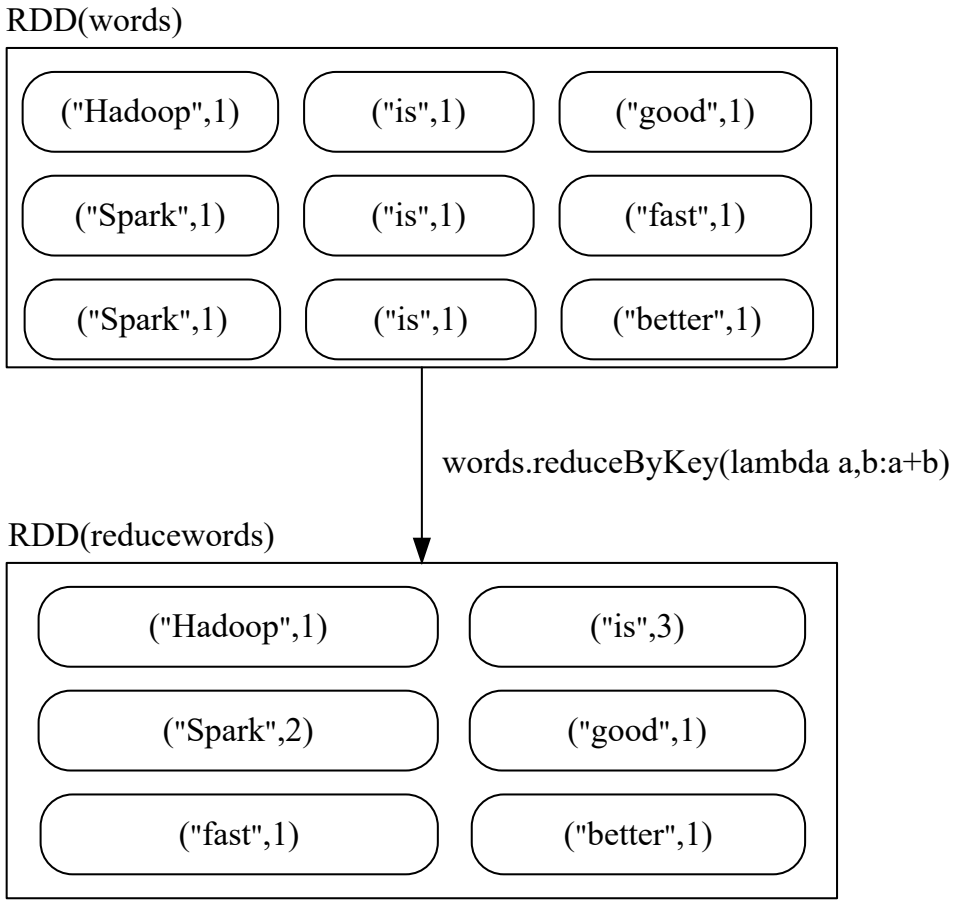
5.1.2 RDD操作

- 转换操作
- 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)
```

5.1.2 RDD操作

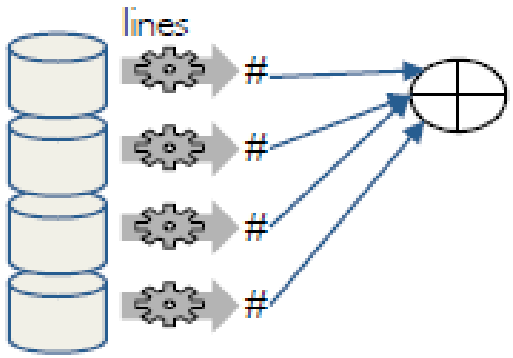
- 转换操作
- reduceByKey(func)



5.1.2 RDD操作

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
```



Count()执行：

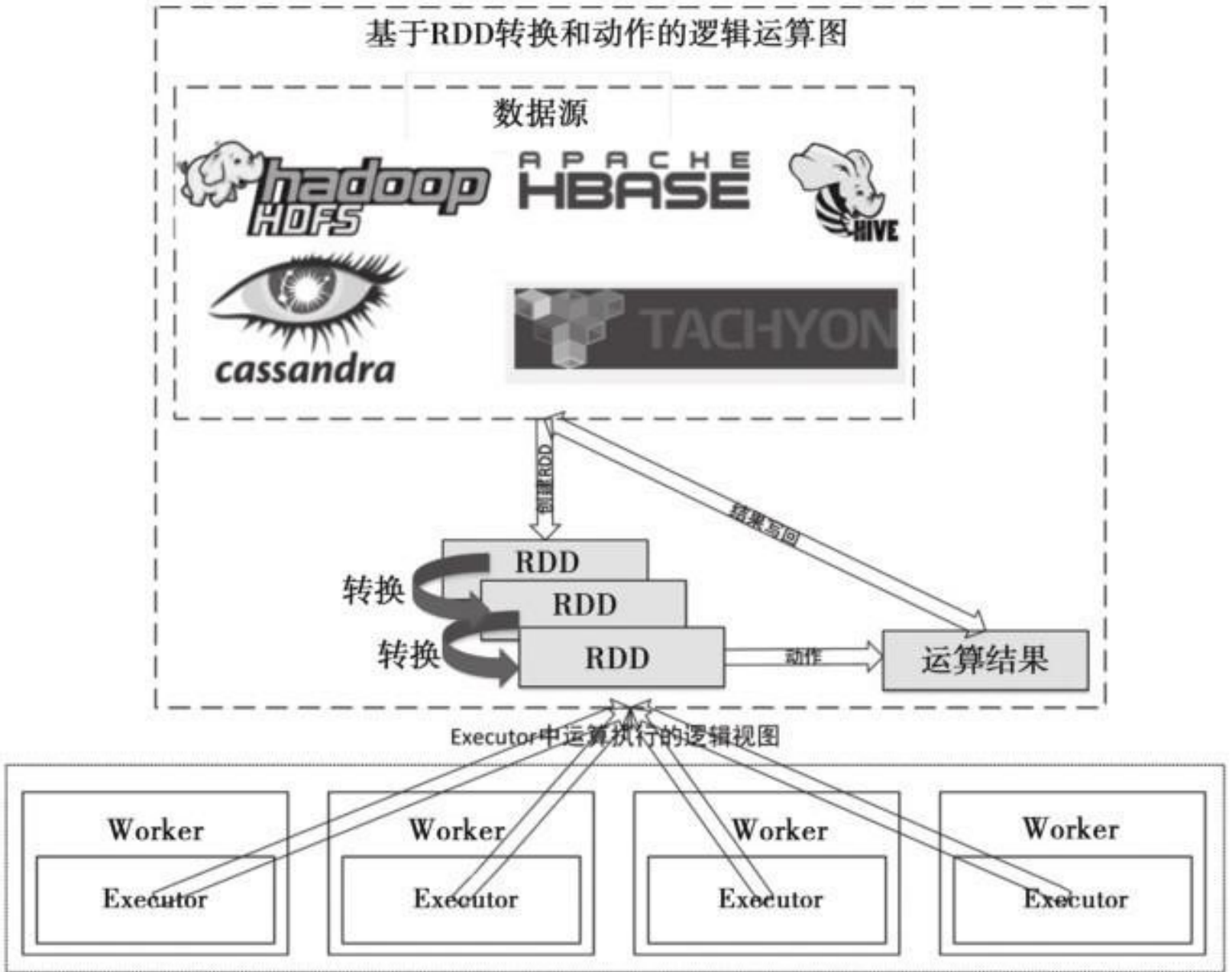
读数据

在分区内进行数目的累加

在driver中将累加数再关联到一起

```
>>> rdd = sc.parallelize([5, 3, 1, 2])
>>> rdd.takeOrdered(3, lambda s: -1 * s)
Value: [5, 3, 2]  # as list
```

5.1.2 RDD转换和动作的逻辑运算



5.1.3 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
- 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
```

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

5.1.3 持久化

- 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

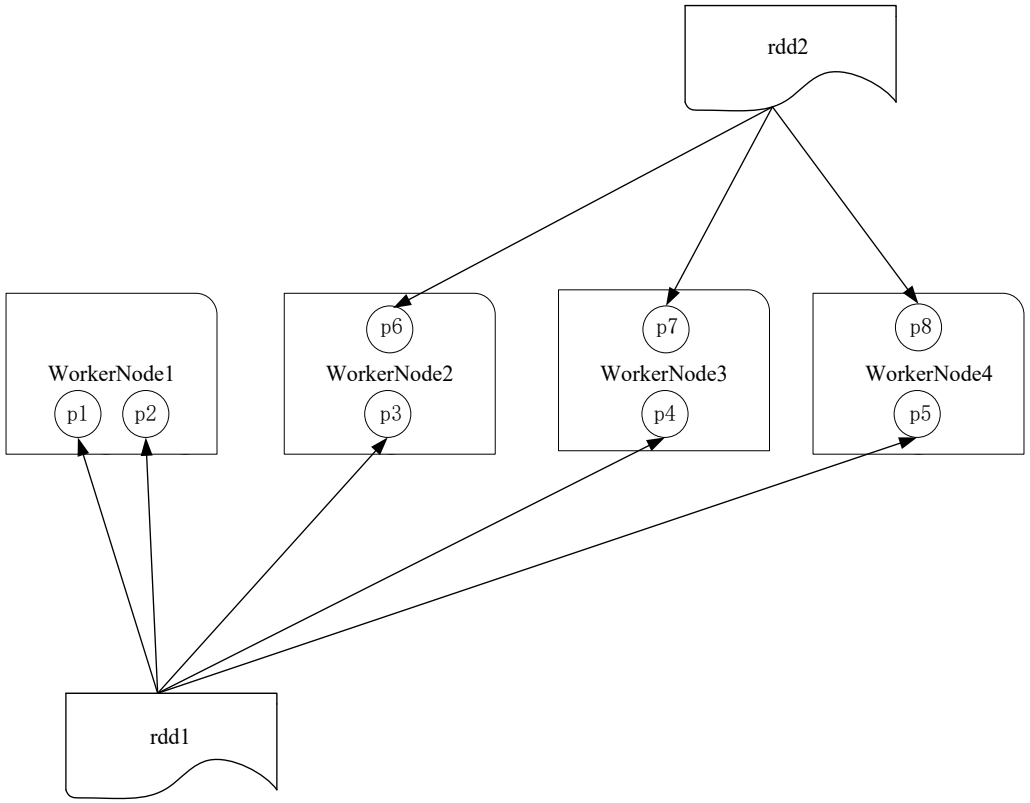
5.1.3 持久化

- 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
```

5.1.4 分区（partition）

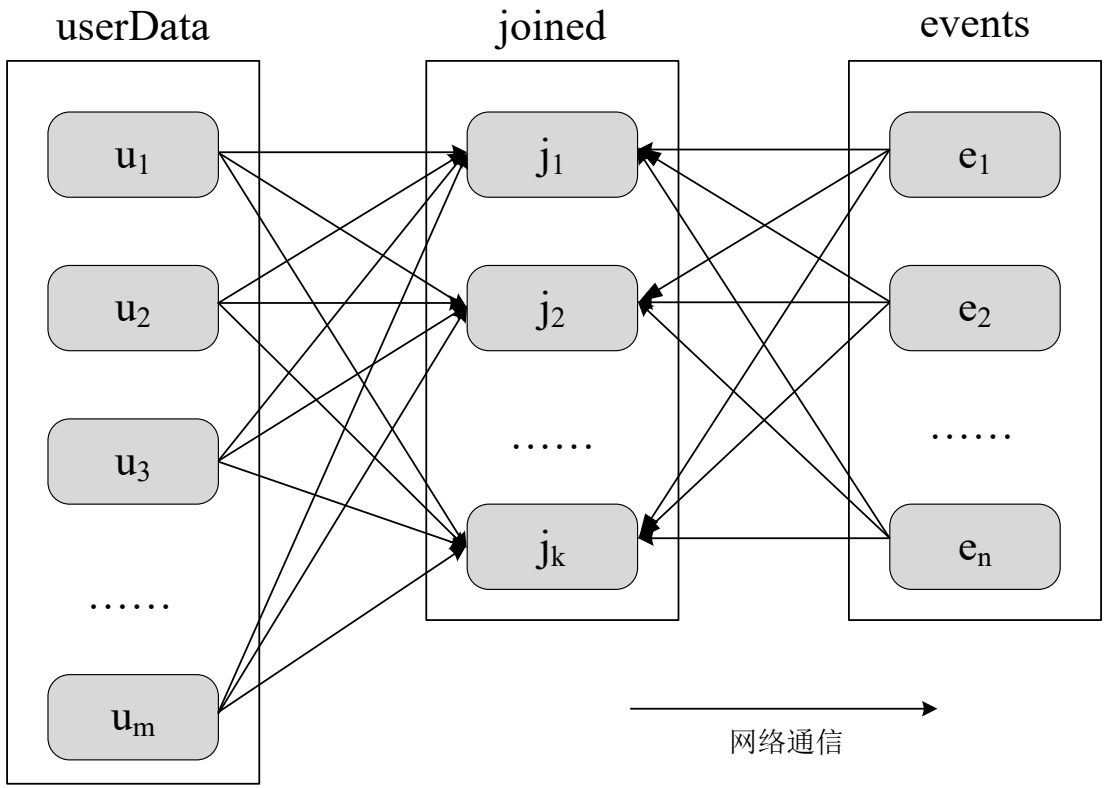
- RDD是弹性分布式数据集，通常RDD很大，会被分成很多个分区，分别保存在不同的节点上
- 分区的作用
- 1. 增加并行度



5.1.4 分区

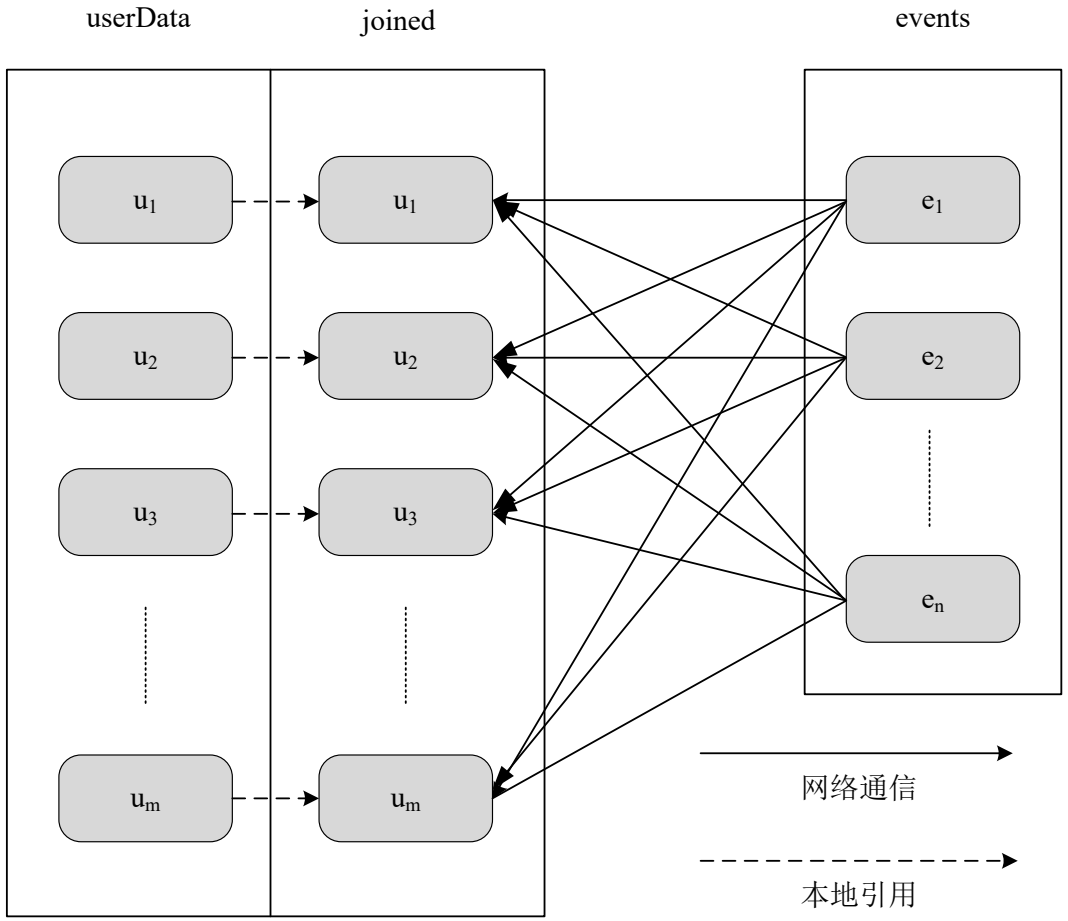
- 分区的作用
- 2. 减少通信开销

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



未分区时对UserData和Events两个表进行连接操作

分区后



5.1.4 分区

- **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

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
```

5.1.4 分区

- **3.设置分区的个数**
- **（2）使用repariti ton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
```

5.1.4 分区

- **4.自定义分区方法**

- 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

5.1.4 分区

4.自定义分区方法

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

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()
```

5.1.4 分区

使用如下命令运行TestPartitioner.py：

```
$ cd /usr/local/spark/mycode/rdd
$ python3 TestPartitioner.py
```

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.py
```

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
```

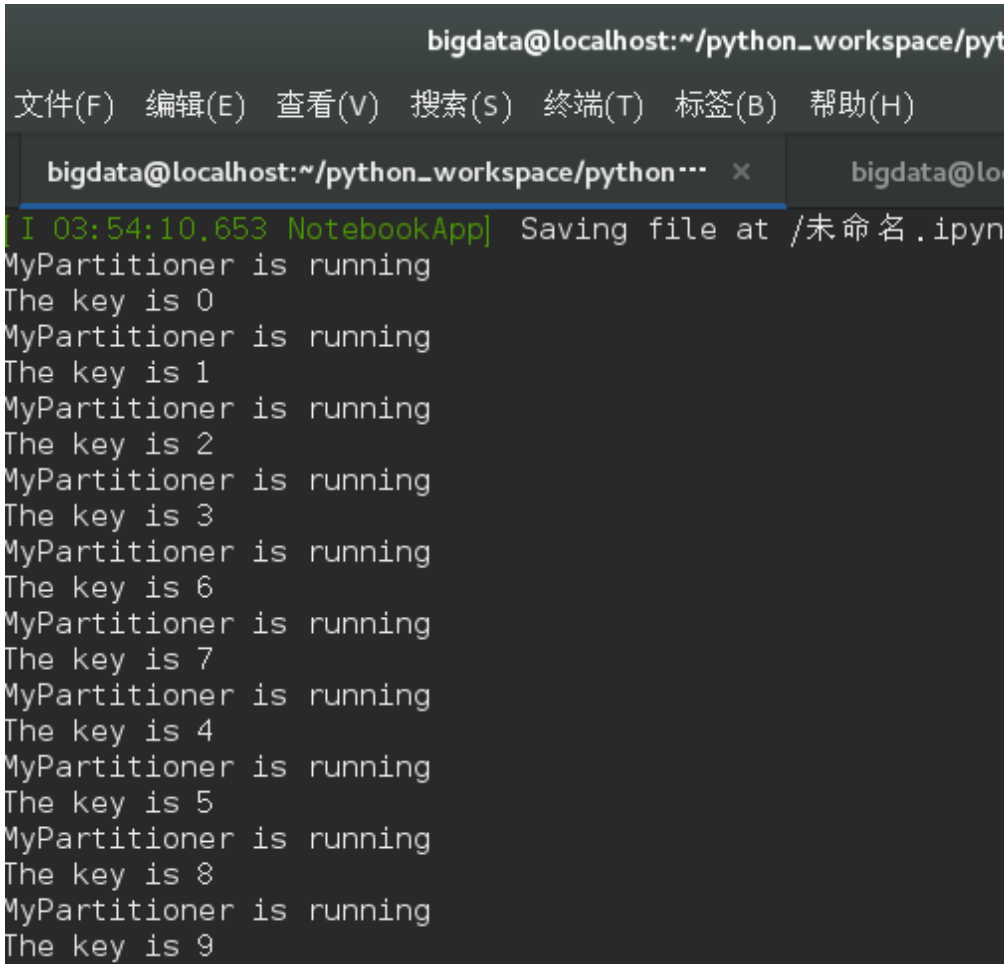
或者，在Jupyter notebook中运行上述代码段：

In [1]:
```
from pyspark import SparkConf, SparkContext
```

In [2]:
```
def MyPartitioner(key):
    print("MyPartitioner is running")
    print('The key is %d' % key)
    return key%10
```

```
print("The main function is running")
conf = SparkConf().setMaster("local").setAppName("MyApp")
sc = SparkContext.getOrCreate(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")
```

```
The main function is running
```



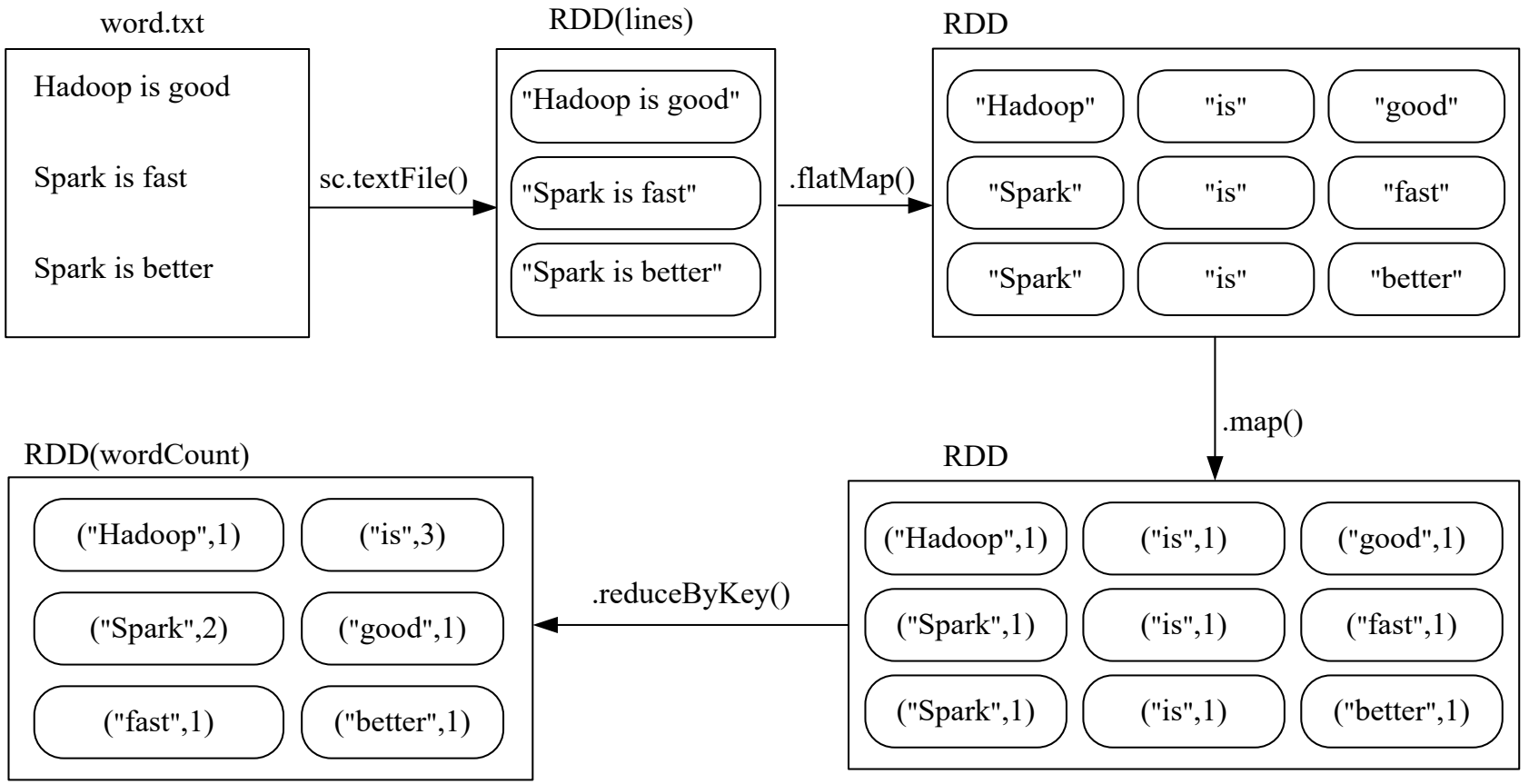
5.1.5 一个综合实例：单词计数

- 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)]
```

5.1.5 一个综合实例：单词计数

```
>>> 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)]
```

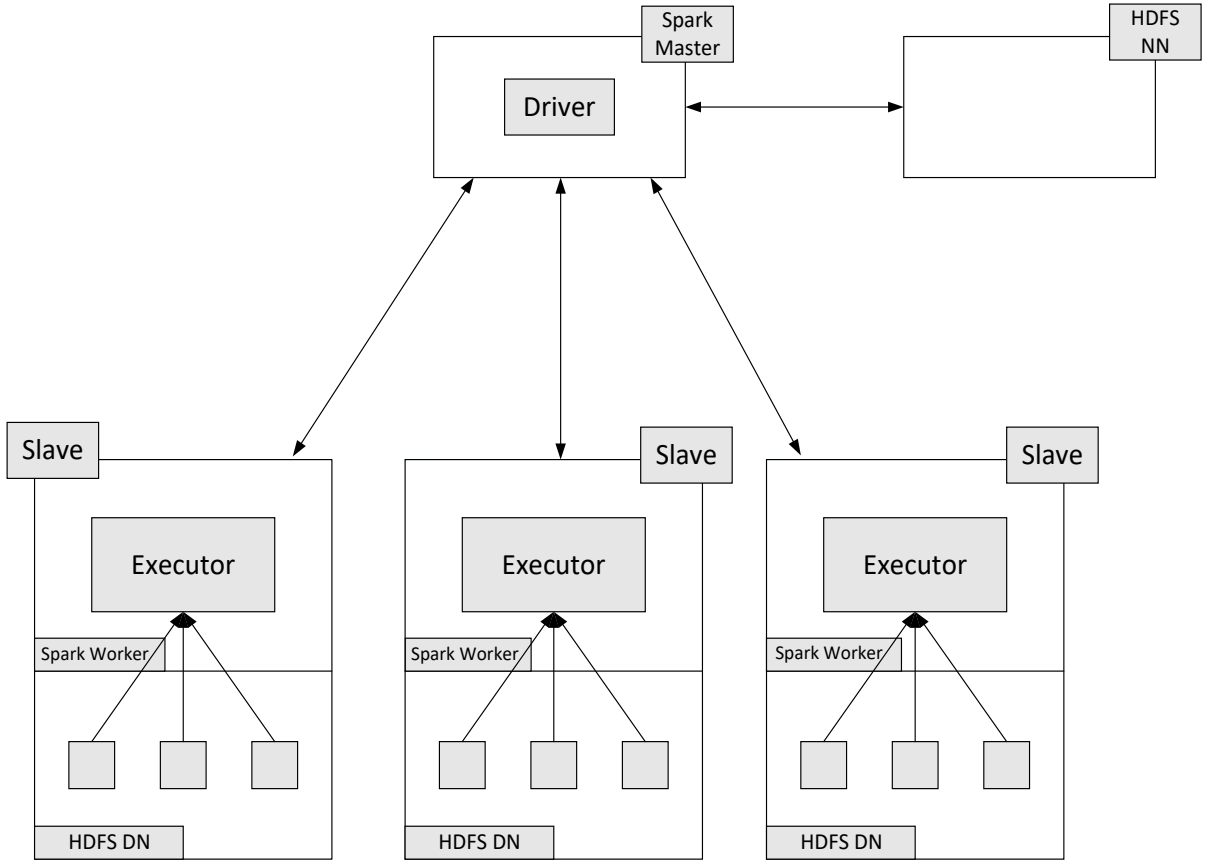


```
>>> lines = sc. \
... textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> wordCount = lines.flatMap(lambda line:line.split(" ")). \
... map(lambda word:(word,1)).reduceByKey(lambda a,b:a+b)
>>> print(wordCount.collect())
[('good', 1), ('Spark', 2), ('is', 3), ('better', 1), ('Hadoop', 1),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

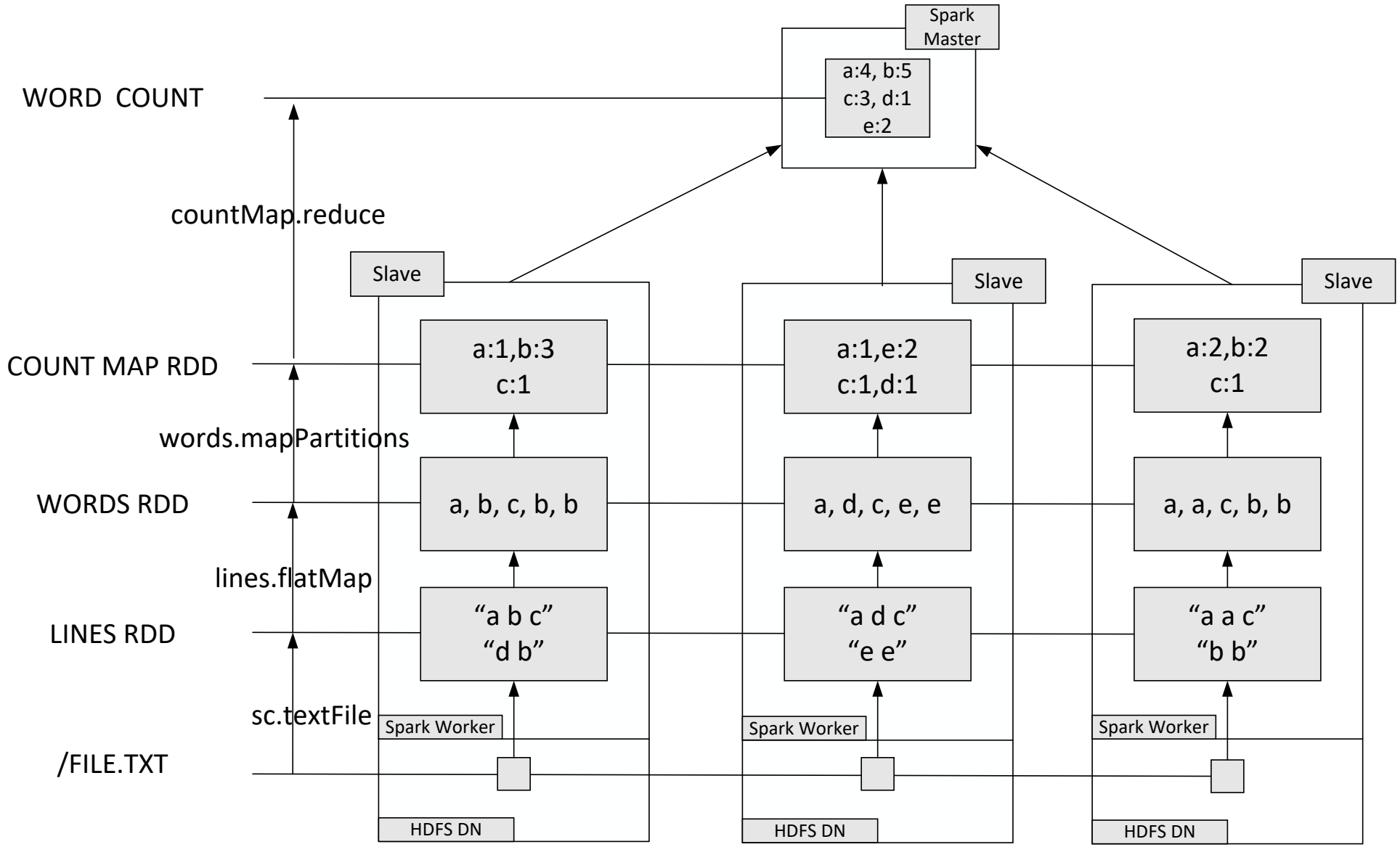
```
>>> from operator import add
>>> lines = sc. \
... textFile("file:///usr/local/spark/mycode/rdd/word.txt")
>>> wordCount = lines.flatMap(lambda line:line.split(" ")). \
... map(lambda word:(word,1)).reduceByKey(add)
>>> print(wordCount.collect())
[('good', 1), ('Spark', 2), ('is', 3), ('better',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

5.1.5 一个综合实例：单词计数

- 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



5.1.5 一个综合实例：单词计数



2

PART TWO

5.2 键值对RDD

5.2 键值对RDD（pair 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

- 键值对RDD（pair RDD）的每个元素是一个键值对元组（pair tuple）
- `rdd = sc.parallelize([(1, 2), (3, 4)])`
- `RDD: [(1, 2), (3, 4)]`

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)
……
```

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)
```

5.2.2常用的键值对RDD转换操作

- reduceByKey(func)
- groupByKey()
- keys
- values
- sortByKey()
- mapValues(func)
- join
- combineByKey

5.2.2常用的键值对RDD转换操作

- **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)
```

5.2.2 常用的键值对RDD转换操作

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

```
>>> 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>)
```

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

分析groupByKey与reduceByKey

- groupByKey() 转换将RDD中所有key相同的元素分成同一组，形成RDD[key,Iterable[value]]的形式，为一个分区中的一个列表。
- groupByKey的问题
  - 此操作需要进行大量的数据移动操作，将所有的值移动到合适的分区中.
  - 生成的列表可能非常庞大，可能会耗尽worker节点的内存.
- reduceByKey() 转换将所有key相同的键值对聚集在一起，将一个函数一次作用到两个值上，并迭代的进行此操作直到生成一个最后的值。
- reduceByKey() 的执行首先在每个分区内进行，然后跨不同分区进行，因此可以有效地扩展到大数据集上.

5.2.2 常用的键值对RDD转换操作

- **reduceByKey和groupByKey的区别**
- 下面得到的wordCountsWithReduce和wordCountsWithGroup是完全一样的，但是，它们的内部运算过程是不同的

```
>>> 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)
```

5.2.2 常用的键值对RDD转换操作

- 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
```

5.2.2 常用的键值对RDD转换操作

- 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
```

5.2.2 常用的键值对RDD转换操作

- 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)
```

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)])
>>> 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)]
```

5.2.2 常用的键值对RDD转换操作

- **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)
```

5.2.2 常用的键值对RDD转换操作

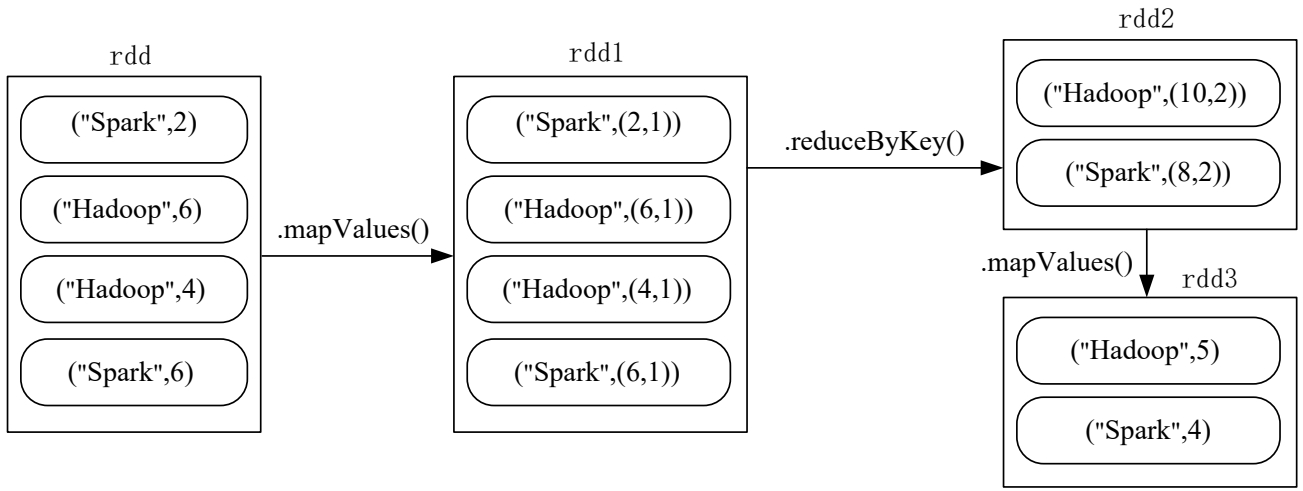
- join就表示内连接。对于内连接，对于给定的两个输入数据集(K,V1)和(K,V2)，只有在两个数据集中都存在的key才会被输出，最终得到一个(K,(V1,V2))类型的数据集。

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

5.2.3 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)]
```



课堂提问

- 下面哪个操作肯定是宽依赖:
  A map B flatMap
  C reduceByKey D sample
- 下列哪个不是 RDD 的缓存方法 :
  A Persist() B Cache()
  C Memory()

课堂提问

- 关于Spark常用算子reduceByKey与groupByKey的区别以及哪种根据优势的说法是正确的是：
  - A. reduceByKey能够在本地先进行merge（聚合)操作，然后跨不同分区进行，因此可以有效地扩展到大数据集上
  - B. reduceByKey比groupByKey更适合针对大数据集进行处理
  - C. groupByKey() 转换将RDD中所有key相同的元素分成同一组，形成RDD[key,Iterable[value]]的形式，为一个分区中的一个列表
  - D. groupByKey：按照key进行分组，直接进行shuffle; reduceByKey按照key进行merge（聚合)，在shuffle之前在本地进行聚合操作

3

PART THREE

5.3 数据读写

5.3 数据读写

- 5.3.1 文件数据读写
- 5.3.2 读写HBase数据

5.3.1 文件数据读写

1. 本地文件系统的数据读写
2. 分布式文件系统HDFS的数据读写

5.3.1 文件数据读写

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 文件数据读写

- **1.本地文件系统的数据读写**
- **（2）把RDD写入到文本文件中**

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

```
$ cd /usr/local/spark/mycode/wordcount/writeback/
$ ls
```

```
part-00000
_SUCCESS
```

- 如果想再次把数据加载在RDD中，只要使用writeback这个目录即可，如下：

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

5.3.1 文件数据读写

- **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://localhost:9000/user/hadoop/word.txt")
>>> textFile.first()
```

- 如下三条语句都是等价的：

```
>>> textFile = sc.textFile("hdfs://localhost: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")
```

HDFS的安装

- 要想从分布式文件系统HDFS中读取数据，首先需要安装Hadoop环境
- Hadoop的三种部署模式
  - 本地（单机）模式
    - 完全运行在本地
    - 不使用HDFS，不加载任何Hadoop守护进程，加载HMaster进程
    - 用于Hadoop程序的开发调试
  - 单机伪分布式模式
    - 所有守护进程运行在同一台机器上
    - 在单机模式基础上增加了代码调试功能：检查HDFS输入输出、与守护进程的交互等
  - 全分布模式
    - 全配置的Hadoop集群
- 单机伪分布式Hadoop环境即能够练习使用HDFS的读写了
- 详情：“CentOS下搭建单机伪分布式HDFS文件系统”

- 实际例子中：
  - 在进行计数时应该忽略其大小写 (e.g., Spark 和 spark 应该当作同一单词进行处理)
  - 去掉所有的标点符号
  - 一行中开头和结尾的空格应该去掉
- 使用Python正则表达式 re 模块来去掉非字母、数字和空格的字符，并转换为小写
- 对单词计数的结果进行排序，取最常用的15个单词及其计数
- 上机作业

基于Jupyter Notebook进行单词计数

- 熟悉基本操作
- 创建RDD及pair RDD
- 将某函数传递给map
  - 把一个基本的RDD中的每个item传递给一个 map() 转换（Transformation），将某函数作用到每个元素上，然后，调用 collect() 动作（Action）来查看转换后的RDD结果
- 将 lambda函数传递给 map

- 熟悉从列表创建RDD

```
wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']
wordsRDD = sc.parallelize(wordsList, 4)
# Print out the type of wordsRDD
print(type(wordsRDD))
```

- 提问：上面的“4”表示什么？

- 将函数 makePlural() 作用到每个元素上. 然后，调用 collect() 行动（Action）来查看转换后的RDD结果.

```
wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']
wordsRDD = sc.parallelize(wordsList, 4)
# Print out the type of wordsRDD
print(type(wordsRDD))
```

```
def makePlural(word):
    return word + 's'

print(makePlural('cat'))
```

```
cats
```

```
pluralRDD = wordsRDD.map(makePlural)
print(pluralRDD.collect())
```

```
['cats', 'elephants', 'rats', 'rats', 'cats']
```

- 将 lambda函数传递给 map，取得同样的结果

```
pluralLambdaRDD = wordsRDD.map(lambda v: v + 's')
print(pluralLambdaRDD.collect())
```

```
['cats', 'elephants', 'rats', 'rats', 'cats']
```

- 使用map和lambda函数取得每个单词的长度

```
pluralLengths = (pluralRDD
                 .map(lambda word: len(word))
                 .collect())
print(pluralLengths)
```

```
[4, 9, 4, 4, 4]
```

创建pair RDD

```
wordPairs = wordsRDD.map(lambda word: (word, 1))
print(wordPairs.collect())
```

```
[('cat', 1), ('elephant', 1), ('rat', 1), ('rat', 1), ('cat', 1)]
```

- 使用一行代码在pair RDD上执行map和reduceByKey、以及collect，进行单词计数

```
from operator import add
wordCountsCollected = (wordsRDD
                       .map(lambda word: (word, 1))
                       .reduceByKey(add)
                       .collect())
print(wordCountsCollected)
```

```
[('cat', 2), ('elephant', 1), ('rat', 2)]
```

计算单词数量以及平均单词个数

- 计算 wordsRDD 中唯一单词的数量

```
uniqueWords = wordCounts.count()
print(uniqueWords)
```

```
3
```

```
from operator import add
wordCounts = wordPairs.reduceByKey(add)
print(wordCounts.collect())
```

```
[('cat', 2), ('elephant', 1), ('rat', 2)]
```

- 计算平均有多少唯一单词。
  - 使用 reduce() 活动计算 wordCounts 中的总数，并除以唯一单词的个数
  - 首先将包含 (key, value)键值对的wordCounts 使用 map() 转换到表示某单词个数的RDD上.

```
from operator import add
totalCount = (wordCounts
              .map(lambda word: word[1])
              .reduce(add))
average = totalCount / float(uniqueWords)
print(totalCount)
print(round(average, 2))
```

```
5
1.67
```

获取最常用的2个单词及它们的计数

top2WordsAndCounts = wordCounts.takeOrdered(2, key=lambda x: -x[1])
print('\n'.join(map(lambda p: '{0}: {1}'.format(p[0], p[1]), top2WordsAndCounts)))

In [1]:
```
wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']
wordsRDD = sc.parallelize(wordsList, 4)
wordPairs = wordsRDD.map(lambda word : (word, 1))
```

In [2]:
```
from operator import add

wordCounts = wordPairs.reduceByKey(add)
top2WordsAndCounts = wordCounts.takeOrdered(2, key = lambda x : -x[1])
print('\n'.join(map(lambda p : '{0} : {1}'.format(p[0], p[1]), top2WordsAndCounts)))
```

```
cat : 2
rat : 2
```

Thank you