

Python并发编程以及系统常用模块

七月在线 王博士

2016年11月19日

主要内容

- ❑ Python多进程与多线程
- ❑ Python使用Hadoop分布式计算库mrjob
- ❑ Python使用Spark分布式计算库PySpark
- ❑ 例子：分别使用MapReduce和Spark实现wordcount
- ❑ 正则表达式简介
- ❑ 日期和时间
- ❑ 常用内建模块：collections；itertools

进程与线程

- 进程：程序的一次执行（程序装载入内存，系统分配资源运行）。
 - 每个进程有自己的内存空间、数据栈等，只能使用进程间通讯，而不能直接共享信息。
- 线程：所有线程运行在同一个进程中，共享相同的运行环境。
 - 每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。
 - 线程的运行可以被抢占（中断），或暂时被挂起（睡眠），让其他线程运行（让步）。
 - 一个进程中的各个线程间共享同一片数据空间。

全局解释器锁GIL

- ❑ GIL全称全局解释器锁Global Interpreter Lock，GIL并不是Python的特性，它是在实现Python解析器(CPython)时所引入的一个概念。
- ❑ GIL是一把全局排他锁，同一时刻只有一个线程在运行。
 - 毫无疑问全局锁的存在会对多线程的效率有不小影响。甚至就几乎等于Python是个单线程的程序。
 - multiprocessing库的出现很大程度上是为了弥补thread库因为GIL而低效的缺陷。它完整的复制了一套thread所提供的接口方便迁移。唯一的不同就是它使用了多进程而不是多线程。每个进程有自己的独立的GIL，因此也不会出现进程之间的GIL争抢。

顺序执行单线程与同时执行两个并发线程

```
from threading import Thread
import time

def my_counter():
    i = 0
    for _ in range(100000000):
        i = i + 1
    return True

def main():
    thread_array = {}
    start_time = time.time()
    for tid in range(2):
        t = Thread(target=my_counter)
        t.start()
        t.join()
    end_time = time.time()
    print("Total time: {}".format(end_time - start_time))

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

Total time: 36.1931009293

```
from threading import Thread
import time

def my_counter():
    i = 0
    for _ in range(100000000):
        i = i + 1
    return True

def main():
    thread_array = {}
    start_time = time.time()
    for tid in range(2):
        t = Thread(target=my_counter)
        t.start()
        thread_array[tid] = t
    for i in range(2):
        thread_array[i].join()
    end_time = time.time()
    print("Total time: {}".format(end_time - start_time))

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

Total time: 45.5059299469

❑ join阻塞进程直到线程执行完毕

Python 多进程（multiprocessing）

❑ fork操作：

- 调用一次，返回两次。因为操作系统自动把当前进程（称为父进程）复制了一份（称为子进程），然后分别在父进程和子进程内返回。子进程永远返回0，而父进程返回子进程的ID。子进程只需要调用getppid()就可以拿到父进程的ID。

```
multiprocessing.py ×
1 import os
2
3 print 'Process (%s) start...' % os.getpid()
4 pid = os.fork()
5 if pid==0:
6     print 'I am child process (%s) and my parent is %s.' % (os.getpid(), os.getppid())
7 else:
8     print 'I (%s) just created a child process (%s).' % (os.getpid(), pid)
```

```
localhost:Python分布式计算 ting$ python multiprocessing.py
Process (25808) start...
I (25808) just created a child process (25809).
I am child process (25809) and my parent is 25808.
```

由于Windows没有fork调用，上面的代码在Windows上无法运行。

multiprocessing

- ❑ multiprocessing是跨平台版本的多进程模块，它提供了一个Process类来代表一个进程对象，下面是示例代码：

```
from multiprocessing import Process
import time

def f(n):
    time.sleep(1)
    print n*n

if __name__ == '__main__':
    for i in range(10):
        p = Process(target=f, args=[i,])
        p.start()
```

0
1
4
9
16
25
36
49
64
81

- ❑ 这个程序如果用单进程写则需要执行10秒以上的时间，而用多进程则启动10个进程并行执行，只需要用1秒多的时间。

进程间通信Queue

- Queue是多进程安全的队列，可以使用Queue实现多进程之间的数据传递。

```
from multiprocessing import Process, Queue
import time

def write(q):
    for i in ['A', 'B', 'C', 'D', 'E']:
        print('Put %s to queue' % i)
        q.put(i)
        time.sleep(0.5)

def read(q):
    while True:
        v = q.get(True)
        print('get %s from queue' % v)

if __name__ == '__main__':
    q = Queue()
    pw = Process(target=write, args=(q,))
    pr = Process(target=read, args=(q,))
    pw.start()
    pr.start()
    pr.join()
    pr.terminate()
```

```
Put A to queue
get A from queue
Put B to queue
get B from queue
Put C to queue
get C from queue
Put D to queue
get D from queue
Put E to queue
get E from queue
```


进程池Pool

□ 用于批量创建子进程，可以灵活控制子进程的数量

```
from multiprocessing import Pool
import time
```

```
def f(x):
    print x*x
    time.sleep(2)
    return x*x

if __name__ == '__main__':
    '''定义启动的进程数量'''
    pool = Pool(processes=5)
    res_list = []

    for i in range(10):
        '''以异步并行的方式启动进程，如果要同步等待的方式，可以在每次启动进程之后调用res.get()方法，也可以使用Pool.apply'''
        res = pool.apply_async(f,[i,])
        print('-----:',i)
        res_list.append(res)

    pool.close()
    pool.join()
    for r in res_list:
        print "result", (r.get(timeout=5))
```

```
1
16
4
0
9
('-----:', 0)
('-----:', 1)
('-----:', 2)
('-----:', 3)
('-----:', 4)
('-----:', 5)
('-----:', 6)
('-----:', 7)
('-----:', 8)
('-----:', 9)
25
49
36
64
81
result 0
result 1
result 4
result 9
result 16
result 25
result 36
result 49
result 64
result 81
```



多进程与多线程对比

- ❑ 在一般情况下多个进程的内存资源是相互独立的，而多线程可以共享同一个进程中的内存资源

```
from multiprocessing import Process
import threading
import time
lock = threading.Lock()

def run(info_list,n):
    lock.acquire()
    info_list.append(n)
    lock.release()
    print('%s\n' % info_list)

if __name__ == '__main__':
    info = []
    for i in range(10):
        #target为子进程执行的函数, args为需要给函数传递的参数
        p = Process(target=run,args=[info,i])
        p.start()
        p.join()
    time.sleep(1) #这里是为了输出整齐让主进程的执行等一下子进程
    print('-----threading-----')
    for i in range(10):
        p = threading.Thread(target=run,args=[info,i])
        p.start()
        p.join()
```

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

函数式编程

□ 三大特性：

- immutable data 不可变数据
- first class functions: 函数像变量一样使用
- 尾递归优化: 每次递归都重用stack

□ 好处：

- parallelization 并行
- lazy evaluation 惰性求值
- determinism 确定性

□ 函数式编程 <http://coolshell.cn/articles/10822.html>

函数式编程技术

□ 技术:

- map & reduce
- pipeline
- recursing 递归
- currying
- higher order function 高阶函数

```
def inc(x):  
    def incx(y):  
        return x+y  
    return incx  
  
inc2 = inc(2)  
inc5 = inc(5)  
  
print inc2(5) # 输出 7  
print inc5(5) # 输出 10
```

7
10

Python中的lambda和map、filter、reduce

□ **lambda**:快速定义单行的最小函数，inline的匿名函数

```
g = lambda x: x * 2
print g(3)
print (lambda x: x * 2)(4)
```

6

8

Python中的lambda和map、filter、reduce

□ **map(function, sequence)** : 对sequence中的item依次执行function(item), 执行结果组成一个List返回

```
for n in ["qi", "yue", "July"]:
    print len(n)
```

```
2
3
4
```

```
name_len = map(len, ["qi", "yue", "July"])
print name_len
```

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

```
def toUpper(item):
    return item.upper()
```

```
upper_name = map(toUpper, ["qi", "yue", "July"])
print upper_name
```

```
['QI', 'YUE', 'JULY']
```

```
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
    squared.append(i**2)
print squared
```

```
[1, 4, 9, 16, 25]
```

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))
print squared
```

```
[1, 4, 9, 16, 25]
```

Python中的lambda和map、filter、reduce

□ **filter(function, sequence)**: 对sequence中的item依次执行function(item), 将执行结果为True的item组成一个List/String/Tuple (取决于sequence的类型) 返回

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print(less_than_zero)
```

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

Python中的lambda和map、filter、reduce

- ❑ **reduce(function, sequence, starting_value):**
对sequence中的item顺序迭代调用function，
如果有starting_value，还可以作为初始值调用

```
def add(x,y): return x + y
print reduce(add, range(1, 5))
print reduce(add, range(1, 5), 10)
```

```
10
20
```


例子：计算数组中的平均数

□ 正常写法：

```
1 num = [2, -5, 9, 7, -2, 5, 3, 1, 0, -3, 8]
2 positive_num_cnt = 0
3 positive_num_sum = 0
4 for i in range(len(num)):
5     if num[i] > 0:
6         positive_num_cnt += 1
7         positive_num_sum += num[i]
8
9 if positive_num_cnt > 0:
10     average = positive_num_sum / positive_num_cnt
11
12 print average
13 # 输出 5
```

例子：计算数组中的平均数

□ 函数式编程：

■ 这样的代码是在描述要干什么，而不是怎么干

```
1 | positive_num = filter(lambda x: x>0, num)
2 | average = reduce(lambda x,y: x+y, positive_num) / len( positive_num )
```

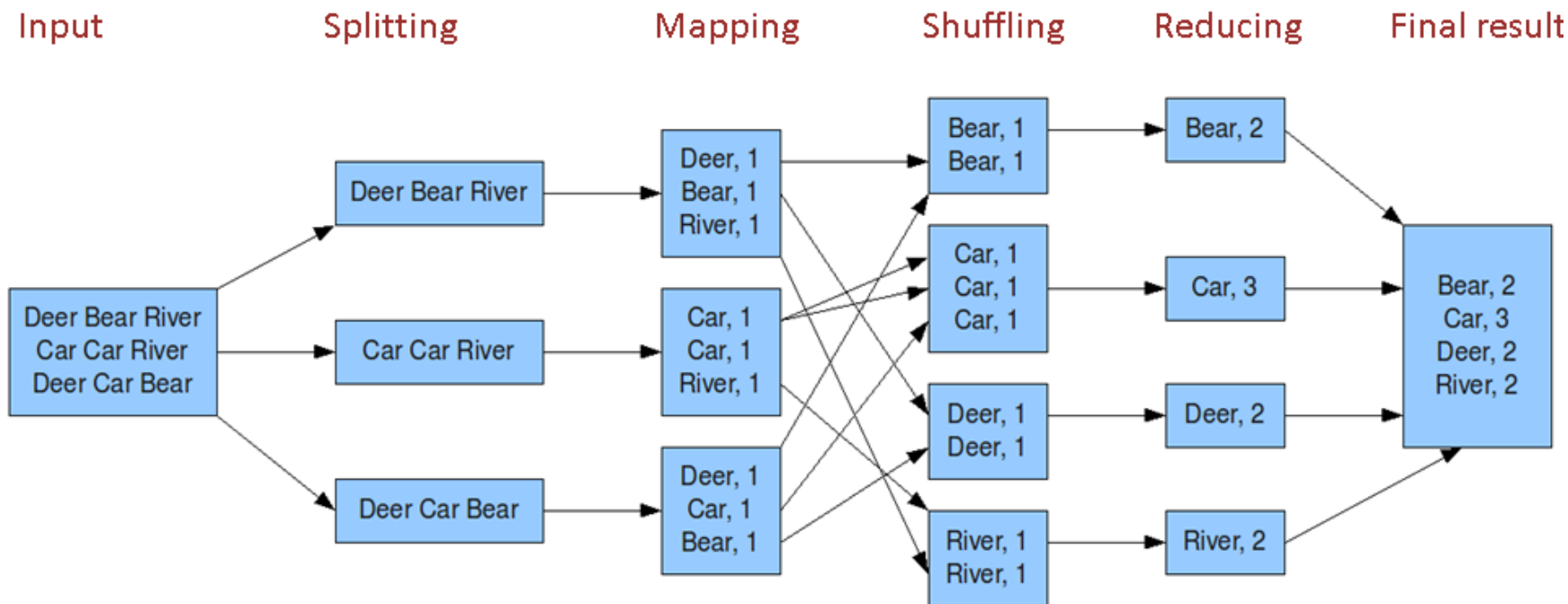
Hadoop

- Hadoop是Apache开源组织的一个分布式计算开源框架。
- 核心的设计就是：MapReduce和HDFS（Hadoop Distributed File System）



MapReducer

□ 思想：任务的分解与结果的汇总



基于Linux管道的MapReducer

□ mapper.py

```
mapper.py
1  import sys
2  for line in sys.stdin:
3      ls = line.split()
4      for word in ls:
5          if len(word.strip()) != 0:
6              print word + ',' + str(1)
```

基于Linux管道的MapReducer

□ reducer.py

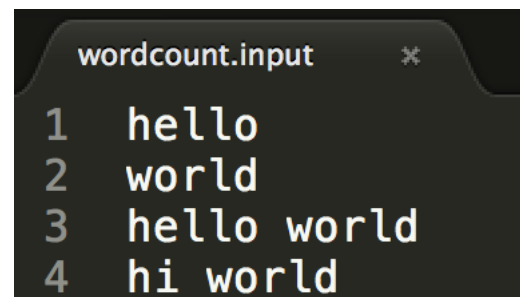
```
reducer.py  ×
1  import sys
2  word_dict = {}
3  for line in sys.stdin:
4      ls = line.split(',')
5      word_dict.setdefault(ls[0], 0)
6      word_dict[ls[0]] += int(ls[1])
7
8  for word in word_dict:
9      print word, word_dict[word]
```

基于Linux管道的MapReducer

❑ `$ cat wordcount.input | python mapper.py | python reducer.py | sort -k 2r`

❑ Output:

- world 3
- hello 2
- hi 1



A screenshot of a terminal window with a dark background. The title bar of the window is labeled 'wordcount.input'. The terminal displays four lines of text, each preceded by a line number: '1 hello', '2 world', '3 hello world', and '4 hi world'.

```
localhost:Python分布式计算 ting$ cat wordcount.input | python mapper.py | python reducer.py | sort -k 2r
world 3
hello 2
hi 1
```

Hadoop Streaming & mrjob

- Hadoop有Java和Streaming两种方式来编写MapReduce任务。
 - Java的优点是计算效率高，并且部署方便，直接打包成一个jar文件就行了。
 - Hadoop Streaming是Hadoop提供的一个编程工具，它允许用户使用任何可执行文件或者脚本文件作为Mapper和Reducer。

Hadoop Streaming & mrjob

```
$ HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/contrib/streaming/hadoop-*-streaming.jar \  
-input myInputDirs \  
-output myOutputDir \  
-mapper Mapper.py\  
-reducer Reducerr.py\  
-file Mapper.py \  
-file Reducer.py
```

□ Streaming 单机测试:

■ `cat input | mapper | sort | reducer > output`

□ mrjob 实质上就是在Hadoop Streaming的命令行上包了一层，有了统一的Python界面，无需你再去直接调用Hadoop Streaming命令。

Mrjob实现wordcount

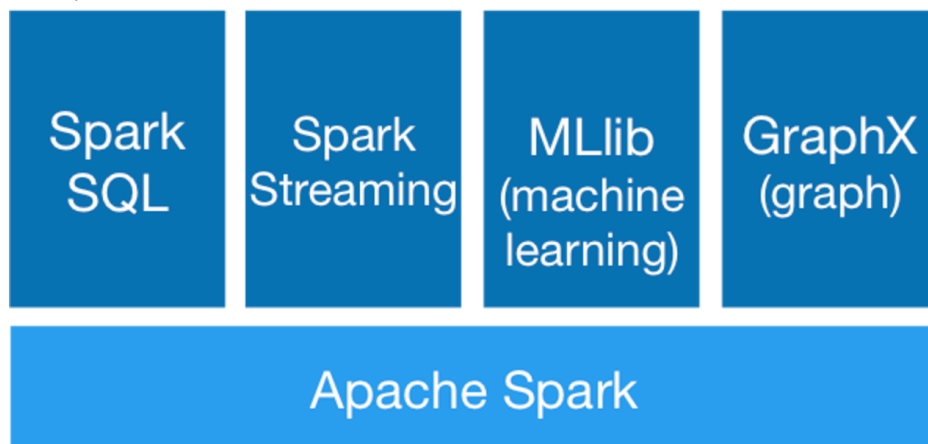
```
word_count.py *
1  from mrjob.job import MRJob
2  class MRWordFrequencyCount(MRJob):
3      def mapper(self, _, line):
4          yield "chars", len(line)
5          yield "words", len(line.split())
6          yield "lines", 1
7      def reducer(self, key, values):
8          yield key, sum(values)
9  if __name__ == '__main__':
10     MRWordFrequencyCount.run()
```

```
localhost:Python分布式计算 ting$ python word_count.py wordcount.input
No configs found; falling back on auto-configuration
Creating temp directory /var/folders/20/qraqw_4z511g0nfybfqwnb_400000gn/T/word_count.ting.20161022.152847.695207
Running step 1 of 1...
Streaming final output from /var/folders/20/qraqw_4z511g0nfybfqwnb_400000gn/T/word_count.ting.20161022.152847.695207/output...
"chars" 29
"lines" 4
"words" 6
Removing temp directory /var/folders/20/qraqw_4z511g0nfybfqwnb_400000gn/T/word_count.ting.20161022.152847.695207...
```

Spark

□ Spark是基于map reduce算法实现的分布式计算框架：

- Spark的中间输出和结果输出可以保存在内存中，从而不再需要读写HDFS。
- Spark能更好地用于数据挖掘与机器学习等需要迭代的map reduce的算法中。



Spark与Hadoop结合

□ Spark可以直接对HDFS进行数据的读写，同样支持Spark on YARN。Spark可以与MapReduce运行于同集群中，共享存储资源与计算。

- 本地模式
- Standalone模式
- Mesos模式
- yarn模式



RDD

□ 弹性分布式数据集 Resilient Distributed Datasets:

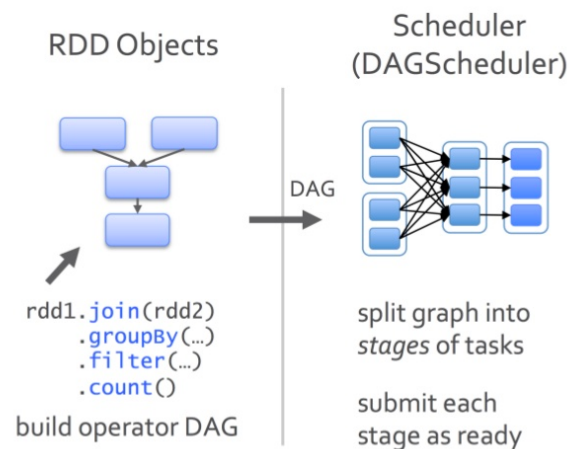
- 集群节点上不可变、已分区对象
- 可序列化
- 可以控制存储级别（内存、磁盘等）来进行重用。

□ 计算特性:

- 血统lineage
- 惰性计算lazy evaluation

□ 生成方式:

- 文件读取
- 来自父RDD



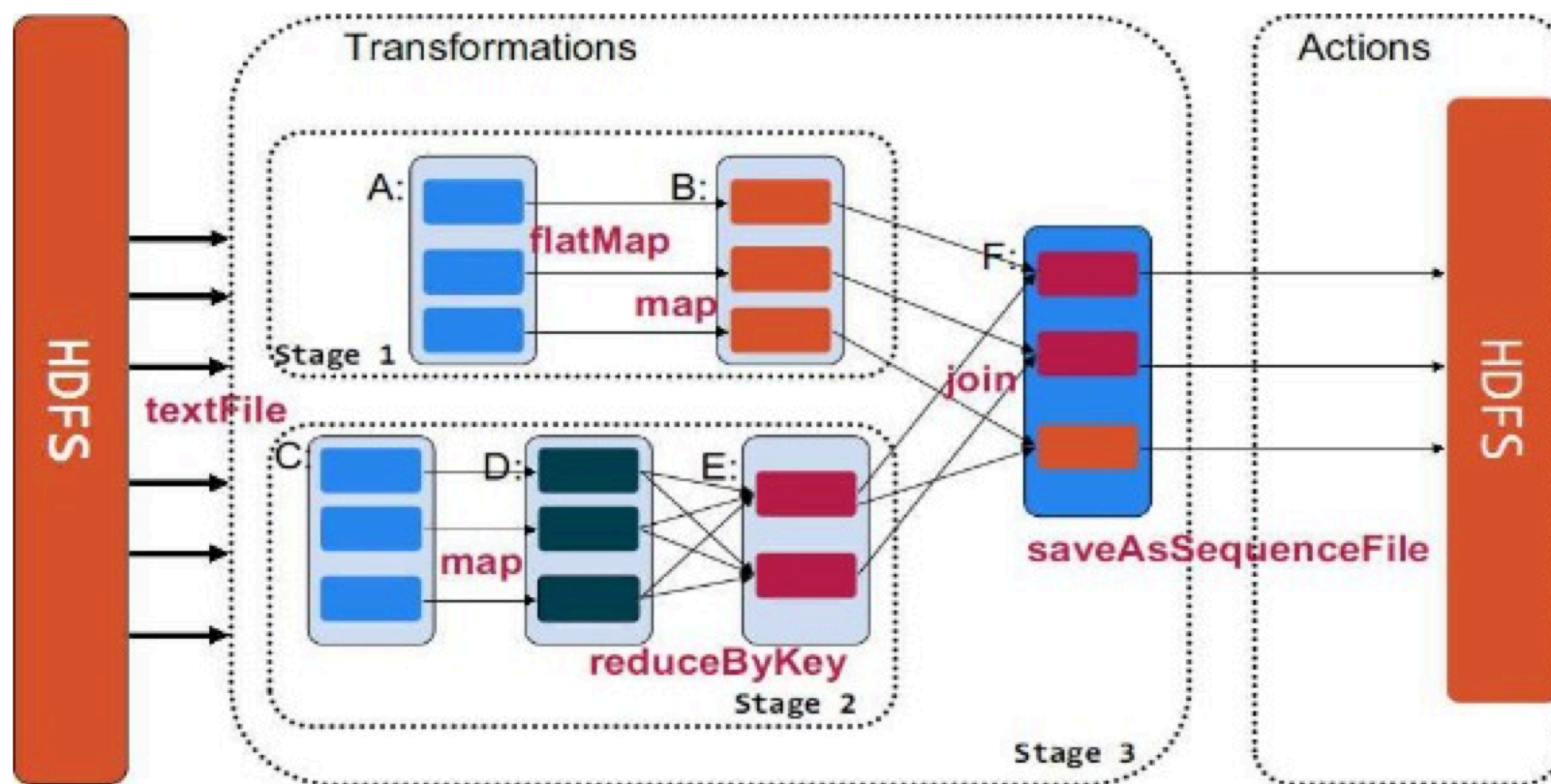
算子： Transformations & Actions

Transformations	$\text{map}(f : T \Rightarrow U) : \text{RDD}[T] \Rightarrow \text{RDD}[U]$ $\text{filter}(f : T \Rightarrow \text{Bool}) : \text{RDD}[T] \Rightarrow \text{RDD}[T]$ $\text{flatMap}(f : T \Rightarrow \text{Seq}[U]) : \text{RDD}[T] \Rightarrow \text{RDD}[U]$ $\text{sample}(\text{fraction} : \text{Float}) : \text{RDD}[T] \Rightarrow \text{RDD}[T]$ (Deterministic sampling) $\text{groupByKey}() : \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, \text{Seq}[V])]$ $\text{reduceByKey}(f : (V, V) \Rightarrow V) : \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$ $\text{union}() : (\text{RDD}[T], \text{RDD}[T]) \Rightarrow \text{RDD}[T]$ $\text{join}() : (\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \Rightarrow \text{RDD}[(K, (V, W))]$ $\text{cogroup}() : (\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \Rightarrow \text{RDD}[(K, (\text{Seq}[V], \text{Seq}[W]))]$ $\text{crossProduct}() : (\text{RDD}[T], \text{RDD}[U]) \Rightarrow \text{RDD}[(T, U)]$ $\text{mapValues}(f : V \Rightarrow W) : \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, W)]$ (Preserves partitioning) $\text{sort}(c : \text{Comparator}[K]) : \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$ $\text{partitionBy}(p : \text{Partitioner}[K]) : \text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$
Actions	$\text{count}() : \text{RDD}[T] \Rightarrow \text{Long}$ $\text{collect}() : \text{RDD}[T] \Rightarrow \text{Seq}[T]$ $\text{reduce}(f : (T, T) \Rightarrow T) : \text{RDD}[T] \Rightarrow T$ $\text{lookup}(k : K) : \text{RDD}[(K, V)] \Rightarrow \text{Seq}[V]$ (On hash/range partitioned RDDs) $\text{save}(\text{path} : \text{String}) : \text{Outputs RDD to a storage system, e.g., HDFS}$

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.



Spark运算逻辑



PySpark实现WordCount

```
import sys
from operator import add
from pyspark import SparkContext
sc = SparkContext()
```

```
lines = sc.textFile("stormofswords.csv")
counts = lines.flatMap(lambda x: x.split(',')) \
               .map(lambda x: (x, 1)) \
               .reduceByKey(add)
output = counts.collect()
output = filter(lambda x: not x[0].isnumeric(), sorted(output, key=lambda x: x[1], reverse = True))
for (word, count) in output[:10]:
    print "%s: %i" % (word, count)

sc.stop()
```

```
Tyrion: 36
Jon: 26
Sansa: 26
Robb: 25
Jaime: 24
Tywin: 22
Cersei: 20
Arya: 19
Robert: 18
Joffrey: 18
```


正则表达式

□ 两种模式匹配：搜索search()和匹配match()

```
import re
m = re.match(r'dog', 'dog cat dog')
print m.group()
print re.match(r'cat', 'dog cat dog')
s = re.search(r'cat', 'dog cat dog')
print s.group()
print re.findall(r'dog', 'dog cat dog')
```

```
dog
None
cat
['dog', 'dog']
```

```
# group()分组
contactInfo = 'Doe, John: 555-1212'
m = re.search(r'(\w+), (\w+): (\S+)', contactInfo)
print m.group(1)
print m.group(2)
print m.group(3)
print m.group(0)
```

```
Doe
John
555-1212
Doe, John: 555-1212
```

正则表达式

❑ 判断一个字符串是否是合法的Email地址

```
# email example
str = 'purple alice-b@google.com monkey dishwasher'
match = re.search(r'\w+@\w+', str)
if match:
    print match.group()  ## 'b@google', 因为\w不能匹配到地址中的 '-' 和 '.'
```

b@google

```
match = re.search(r'[\w.-]+@[ \w.-]+', str)
if match:
    print match.group()  ## 'alice-b@google.com'
```

alice-b@google.com

❑ 作业1：电话号码正则匹配

正则表达式

□ 作业1：电话号码正则匹配

■ 例子：

- +008613112345678
- +861795101023231212
- +8608715432231
- 01023459764
- 06346046499
- 010120

时间和日期

□ time模块和datetime模块

```
import time
print time.time()
print time.localtime()
for i in range(3):
    time.sleep(0.5)
    print "Tick!"
```

```
1479398434.32
time.struct_time(tm_year=2016, tm_mon=11, tm_mday=18,
tm_hour=0, tm_min=0, tm_sec=34, tm_wday=4,
tm_yday=323, tm_isdst=0)
Tick!
Tick!
Tick!
```

```
import datetime
print "today is: ", datetime.date.today()
print "now is: ", datetime.datetime.now()
print datetime.date(2016,6,4)
print datetime.time(14,00)
```

```
today is: 2016-11-18
now is: 2016-11-18 00:11:53.873161
2016-06-04
14:00:00
```

时间和日期

```
# 计算昨天和明天的日期  
import datetime  
today = datetime.date.today()  
yesterday = today - datetime.timedelta(days=1)  
tomorrow = today + datetime.timedelta(days=1)  
print yesterday, today, tomorrow
```

2016-11-17 2016-11-18 2016-11-19

 作业2：计算日期之间的工作日

有用的内建函数

□ enumerate 函数

对一个列表或数组既要遍历索引又要遍历元素时

```
l = [1,2,3]
for i in range (len(l)):
    print i ,l[i]
```

```
0 1
1 2
2 3
```

enumerate会将数组或列表组成一个索引序列。使我们再获取索引和索引内容的时候更加方便如下:

```
for index,text in enumerate(l):
    print index ,text
```

```
0 1
1 2
2 3
```

集合模块collections

- ❑ collections是Python内建的一个集合模块，提供了许多有用的集合类。
- ❑ deque是为了高效实现插入和删除操作的双向列表，适合用于队列和栈。
- ❑ OrderedDict的Key会按照插入的顺序排列。
- ❑ Counter是一个简单的计数器，也是dict的一个子类。

迭代器itertools

❑ 为类序列对象提供了一个类序列接口

❑ 无限迭代器:

Iterator	Arguments	Results	Example
<code>count()</code>	start, [step]	start, start+step, start+2*step, ...	<code>count(10)</code> --> 10 11 12 13 14 ...
<code>cycle()</code>	p	p0, p1, ... plast, p0, p1, ...	<code>cycle('ABCD')</code> --> A B C D A B C D ...
<code>repeat()</code>	elem [,n]	elem, elem, elem, ... endlessly or up to n times	<code>repeat(10, 3)</code> --> 10 10 10

❑ 在最短输入序列终止的迭代器:

Iterator	Arguments	Results	Example
<code>chain()</code>	p, q, ...	p0, p1, ... plast, q0, q1, ...	<code>chain('ABC', 'DEF')</code> --> A B C D E F
<code>compress()</code>	data, selectors	(d[0] if s[0]), (d[1] if s[1]), ...	<code>compress('ABCDEF', [1,0,1,0,1,1])</code> --> A C E F
<code>dropwhile()</code>	pred, seq	seq[n], seq[n+1], starting when pred fails	<code>dropwhile(lambda x: x<5, [1,4,6,4,1])</code> --> 6 4 1
<code>groupby()</code>	iterable[, keyfunc]	sub-iterators grouped by value of keyfunc(v)	
<code>ifilter()</code>	pred, seq	elements of seq where pred(elem) is true	<code>ifilter(lambda x: x%2, range(10))</code> --> 1 3 5 7 9
<code>ifilterfalse()</code>	pred, seq	elements of seq where pred(elem) is false	<code>ifilterfalse(lambda x: x%2, range(10))</code> --> 0 2 4 6 8
<code>islice()</code>	seq, [start,] stop [, step]	elements from seq[start:stop:step]	<code>islice('ABCDEFGH', 2, None)</code> --> C D E F G
<code>imap()</code>	func, p, q, ...	func(p0, q0), func(p1, q1), ...	<code>imap(pow, (2,3,10), (5,2,3))</code> --> 32 9 1000
<code>starmap()</code>	func, seq	func(*seq[0]), func(*seq[1]), ...	<code>starmap(pow, [(2,5), (3,2), (10,3)])</code> --> 32 9 1000
<code>tee()</code>	it, n	it1, it2, ... itn splits one iterator into n	
<code>takewhile()</code>	pred, seq	seq[0], seq[1], until pred fails	<code>takewhile(lambda x: x<5, [1,4,6,4,1])</code> --> 1 4
<code>izip()</code>	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	<code>izip('ABCD', 'xy')</code> --> Ax By
<code>izip_longest()</code>	p, q, ...	(p[0], q[0]), (p[1], q[1]), ...	<code>izip_longest('ABCD', 'xy', fillvalue='-')</code> --> Ax By C- D-

迭代器itertools

□ 组合生成器：

Iterator	Arguments	Results
<code>product()</code>	<code>p, q, ... [repeat=1]</code>	cartesian product, equivalent to a nested for-loop
<code>permutations()</code>	<code>p[, r]</code>	r-length tuples, all possible orderings, no repeated elements
<code>combinations()</code>	<code>p, r</code>	r-length tuples, in sorted order, no repeated elements
<code>combinations_with_replacement()</code>	<code>p, r</code>	r-length tuples, in sorted order, with repeated elements
<code>product('ABCD', repeat=2)</code>		AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD
<code>permutations('ABCD', 2)</code>		AB AC AD BA BC BD CA CB CD DA DB DC
<code>combinations('ABCD', 2)</code>		AB AC AD BC BD CD
<code>combinations_with_replacement('ABCD', 2)</code>		AA AB AC AD BB BC BD CC CD DD

□ 参考：http://python.usyiyi.cn/python_278/library/itertools.html



感谢大家！

恳请大家批评指正！