Spark

实验目的

安装spark并熟悉pyspark 的RDD算子操作

实验过程

一、Spark local模式安装

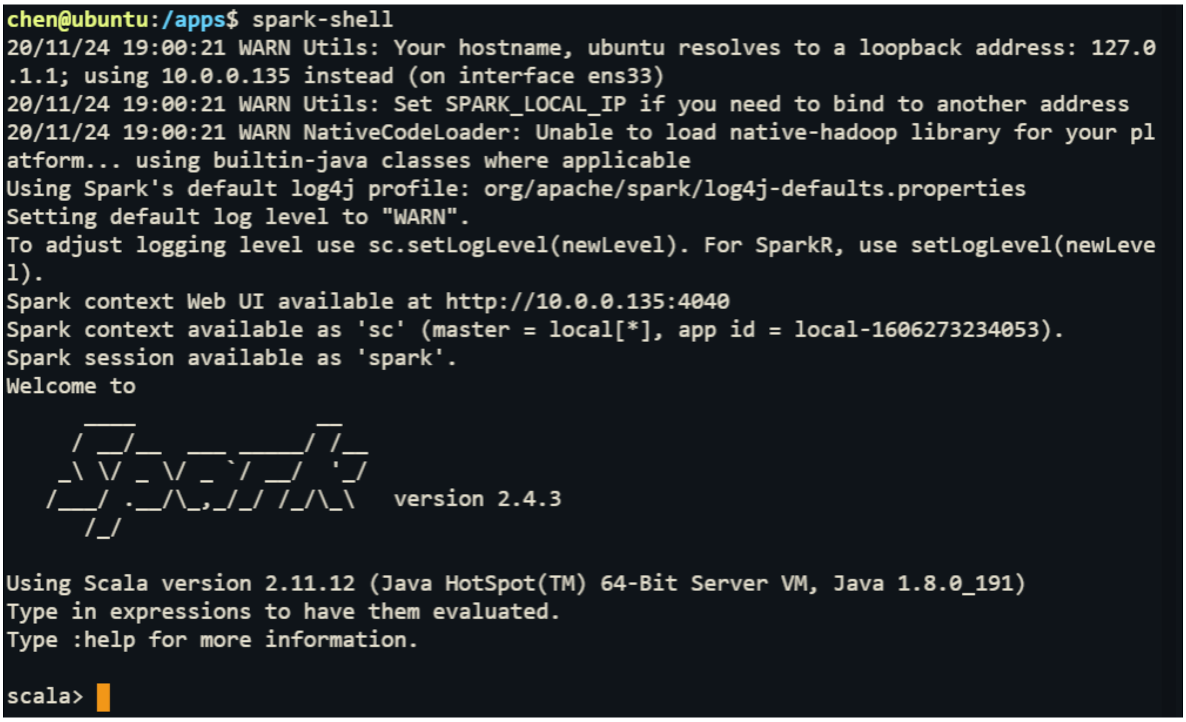
1.1安装scala

```
cp ~/big_data_tools/scala-2.12.8.tgz /apps/
tar zxvf /apps/scala-2.12.8.tgz -C /apps/
mv /apps/scala-2.12.8/ /apps/scala
# 删除压缩包
rm /apps/scala-2.12.8.tgz
# 添加环境变量
vim ~/.bashrc
# Scala
export SCALA_HOME=/apps/scala
export PATH=$SCALA_HOME/bin:$PATH
# 使配置生效
source ~/.bashrc
```

1.2安装spark

```
cp ~/big_data_tools/spark-2.4.3-bin-hadoop2.7.tgz /apps
tar zxvf /apps/spark-2.4.3-bin-hadoop2.7.tgz -C /apps/
mv /apps/spark-2.4.3-bin-hadoop2.7/ /apps/spark
# 删除压缩包
rm /apps/spark-2.4.3-bin-hadoop2.7.tgz
# 添加环境变量
vim ~/.bashrc
# Spark
export SPARK_HOME=/apps/spark
export PATH=$SPARK_HOME/bin:$PATH
# 使配置生效
source ~/.bashrc
```

不需要对 spark 进行任何配置，就可以启动 spark-shell 进行任务处理了。 在终端中执行



```
chen@ubuntu:/apps$ spark-shell
20/11/24 19:00:21 WARN Utils: Your hostname, ubuntu resolves to a loopback address: 127.0
.1.1; using 10.0.0.135 instead (on interface ens33)
20/11/24 19:00:21 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
20/11/24 19:00:21 WARN NativeCodeLoader: Unable to load native-hadoop library for your pl
atform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLeve
l).
Spark context Web UI available at http://10.0.0.135:4040
Spark context available as 'sc' (master = local[*], app id = local-1606273234053).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.3
      /_/

Using Scala version 2.11.12 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_191)
Type in expressions to have them evaluated.
Type :help for more information.

scala> 
```

产查看当前运行模式

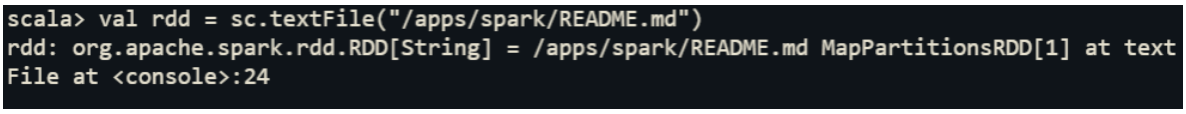
```
scala> sc.master
res1: String = local[*]
```

用单机的多个线程来模拟Spark分布式计算。

1.3执行测试

在 Spark Shell 中，使用 Scala 加载 Spark 安装目录下文件 README.md 并转变为 RDD。

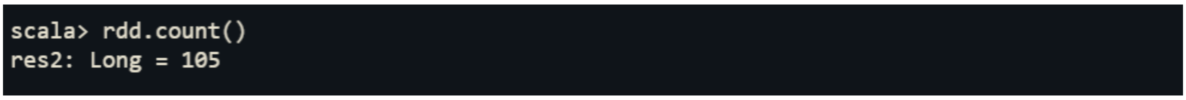
```
val rdd = sc.textFile("/apps/spark/README.md")
```



```
scala> val rdd = sc.textFile("/apps/spark/README.md")
rdd: org.apache.spark.rdd.RDD[String] = /apps/spark/README.md MapPartitionsRDD[1] at text
File at <console>:24
```

对 RDD 进行算子操作，统计文件的行数。

```
rdd.count()
```



```
scala> rdd.count()
res2: Long = 105
```

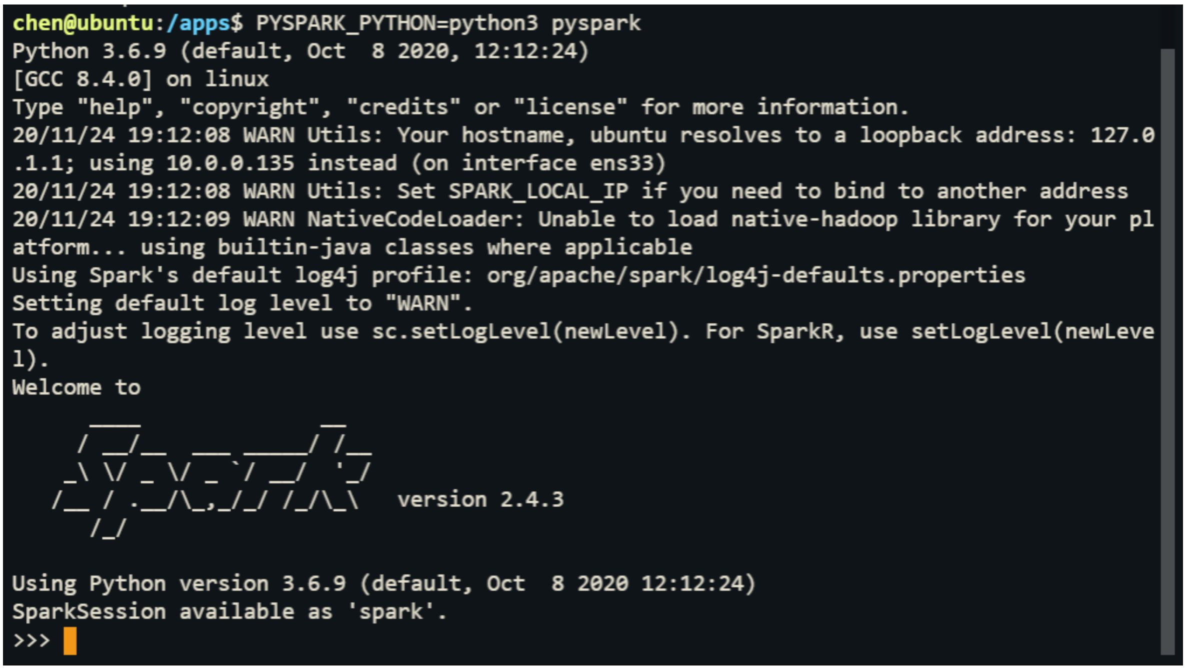
退出

```
:quit
```

1.4启动pyspark

在终端中执行 ,指定运行spark的python版本

```
PYSPARK_PYTHON=python3 pyspark
```

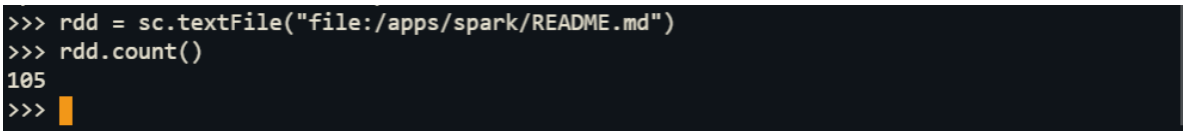


```
chen@ubuntu:/apps$ PYSPARK_PYTHON=python3 pyspark
Python 3.6.9 (default, Oct  8 2020, 12:12:24)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
20/11/24 19:12:08 WARN Utils: Your hostname, ubuntu resolves to a loopback address: 127.0
.1.1; using 10.0.0.135 instead (on interface ens33)
20/11/24 19:12:08 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
20/11/24 19:12:09 WARN NativeCodeLoader: Unable to load native-hadoop library for your pl
atform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLeve
l).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.4.3
      /_/

Using Python version 3.6.9 (default, Oct  8 2020 12:12:24)
SparkSession available as 'spark'.
>>> 
```

在 Spark Shell 中，使用 Python 加载 Spark 安装目录下文件 README.md 并转变为 RDD

```
>>> rdd = sc.textFile("file:/apps/spark/README.md")
>>> rdd.count()
```



```
>>> rdd = sc.textFile("file:/apps/spark/README.md")
>>> rdd.count()
105
>>> 
```

到此 Spark Local 模式已经安装完成.

二、伪分布式安装

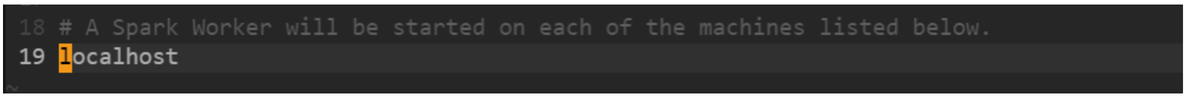
安装伪分布式，还需要对配置文件做一些修改。进入配置文件目录/apps/spark/conf

```
cd /apps/spark/conf
```

将 slaves.template 重命名为 slaves

```
mv slaves.template slaves
```

之前只有一个节点，保持原样就可以了



```
# A Spark Worker will be started on each of the machines listed below.
localhost
```

将 spark-env.sh.template 重命名 spark-env.sh

```
 mv spark-env.sh.template spark-env.sh
```

在 spark-env.sh 中添加如下内容

```
HADOOP_CONF_DIR=/apps/hadoop/etc/hadoop
JAVA_HOME=/apps/java
SPARK_MASTER_IP=ubuntu
SPARK_MASTER_PORT=7077
SPARK_MASTER_WEBUI_PORT=8080
SPARK_WORKER_CORES=1
SPARK_WORKER_MEMORY=1g
SPARK_WORKER_PORT=7078
SPARK_WORKER_WEBUI_PORT=8081
SPARK_EXECUTOR_INSTANCES=1
```

说明：需要配置 JAVA_HOME 以及 HADOOP 配置文件所在的目录 HADOOP_CONF_DIR。SPARK_MASTER_IP、SPARK_MASTER_PORT、 SPARK_MASTER_WEBUI_PORT，分别指 spark 集群中，master 节点的 ip 地址、端口 号、提供的 web 接口的端口。SPARK_WORKER_CORES、SPARK_WORKER_MEMORY 分布为 worker 节点的内核数、内存大小。此处根据自己机器情况调整配置项参数，比如 ip 地址改为自己的主机名。

配置传递给 spark 应用程序的默认属性 将 spark-defaults.conf.template 重命名 spark-defaults.conf

```
mv spark-defaults.conf.template spark-defaults.conf
```

在其中添加如下内容

```
spark.master                    spark://ubuntu:7077
spark.eventLog.enabled          true
spark.eventLog.dir              hdfs://localhost:9000/spark/eventLog
spark.serializer                org.apache.spark.serializer.KryoSerializer
spark.driver.memory             1g
spark.jars.package              Azure:mmlspark:0.12
```

MMLSpark 是微软开源的用于 Spark 的深度学习库，为 Apache Spark 提供了大量深度 学习和数据科学工具，包括将 Spark Machine Learning 管道与 Microsoft Cognitive Toolkit(CNTK)和 OpenCV 进行无缝集成，使您能够快速创建功能强大，高度可扩展的大 型图像和文本数据集分析预测模型。

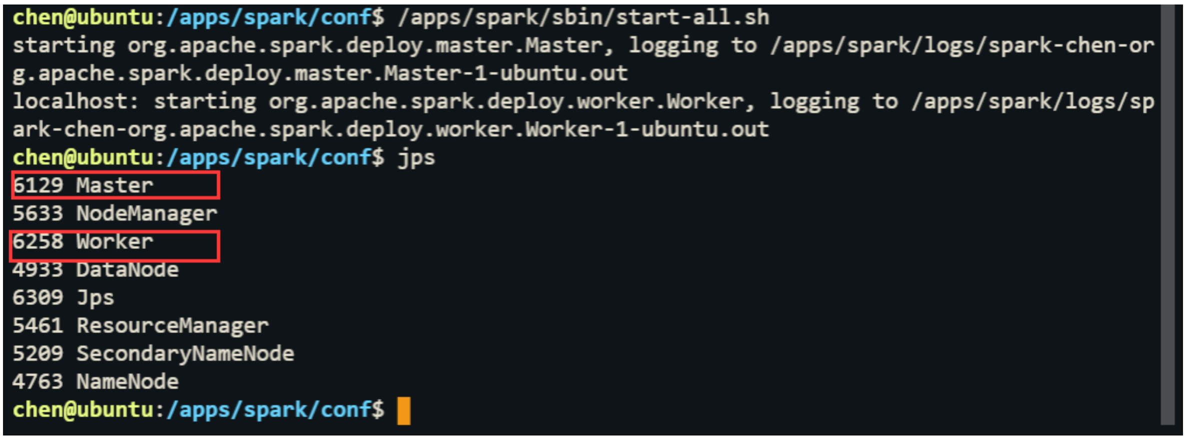
eventLog 用来存放日志，需要手动创建

```
start-all.sh
hadoop fs -mkdir -p /spark/eventLog
```

spark-defaults.conf 文件不配置的话，运行演示示例的任务不会显示在 web 界面中。

2.1启动spark

```
/apps/spark/sbin/start-all.sh
```

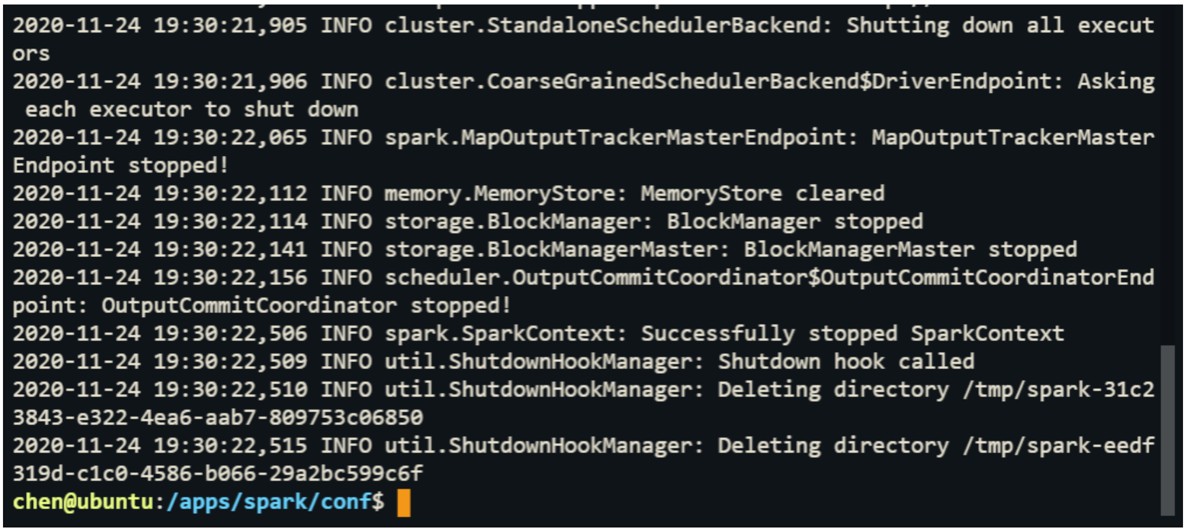


```
chen@ubuntu:/apps/spark/conf$ /apps/spark/sbin/start-all.sh
starting org.apache.spark.deploy.master.Master, logging to /apps/spark/logs/spark-chen-or
g.apache.spark.deploy.master.Master-1-ubuntu.out
localhost: starting org.apache.spark.deploy.worker.Worker, logging to /apps/spark/logs/sp
ark-chen-org.apache.spark.deploy.worker.Worker-1-ubuntu.out
chen@ubuntu:/apps/spark/conf$ jps
6129 Master
5633 NodeManager
6258 Worker
4933 DataNode
6309 Jps
5461 ResourceManager
5209 SecondaryNameNode
4763 NameNode
chen@ubuntu:/apps/spark/conf$
```

可以看到 Spark 创建了 Master 和 Worker 两个进程

运行演示实例

```
/apps/spark/bin/run-example SparkPi
```



```
2020-11-24 19:30:21,905 INFO cluster.StandaloneSchedulerBackend: Shutting down all execut
ors
2020-11-24 19:30:21,906 INFO cluster.CoarseGrainedSchedulerBackend$DriverEndpoint: Asking
 each executor to shut down
2020-11-24 19:30:22,065 INFO spark.MapOutputTrackerMasterEndpoint: MapOutputTrackerMaster
Endpoint stopped!
2020-11-24 19:30:22,112 INFO memory.MemoryStore: MemoryStore cleared
2020-11-24 19:30:22,114 INFO storage.BlockManager: BlockManager stopped
2020-11-24 19:30:22,141 INFO storage.BlockManagerMaster: BlockManagerMaster stopped
2020-11-24 19:30:22,156 INFO scheduler.OutputCommitCoordinator$OutputCommitCoordinatorEnd
point: OutputCommitCoordinator stopped!
2020-11-24 19:30:22,506 INFO spark.SparkContext: Successfully stopped SparkContext
2020-11-24 19:30:22,509 INFO util.ShutdownHookManager: Shutdown hook called
2020-11-24 19:30:22,510 INFO util.ShutdownHookManager: Deleting directory /tmp/spark-31c2
3843-e322-4ea6-aab7-809753c06850
2020-11-24 19:30:22,515 INFO util.ShutdownHookManager: Deleting directory /tmp/spark-eedf
319d-c1c0-4586-b066-29a2bc599c6f
chen@ubuntu:/apps/spark/conf$
```

日志信息很多，很难找到输出结果，下面对日志进行设置。

2.2设置日志

上面运行过程中，由于 Log4j 的日志输出级别为 INFO 级别，所以会在屏幕上输出很多的 日志信息，造成很难定位程序的输出结果。可以通过修改日志级别进行解决。

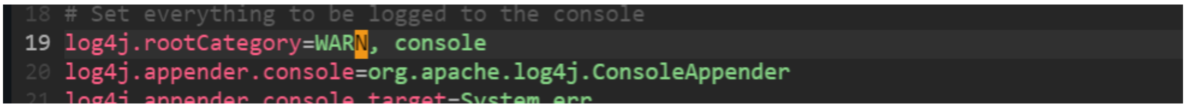
切换目录到/apps/spark/sbin 目录下，停止 Spark。

```
/apps/spark/sbin/stop-all.sh
```

再切换目录到/apps/spark/conf 目录下，将目录下 log4j.properties.template 重命名为

```
mv log4j.properties.template log4j.properties
vim log4j.properties
```

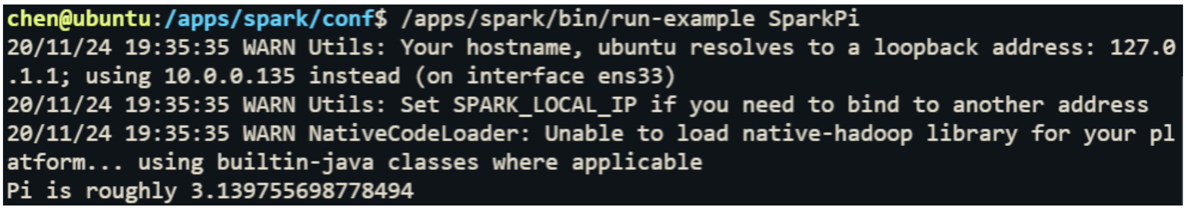
第 19 行修改 log4j.rootCategory 的值为 WARN



```
18 # Set everything to be logged to the console
19 log4j.rootCategory=WARN, console
20 log4j.appender.console=org.apache.log4j.ConsoleAppender
21 log4j.appender.console.target=System.err
```

启动 Spark，再次运行演示实例，可以很容易找到结果。

```
/apps/spark/sbin/start-all.sh
/apps/spark/bin/run-example SparkPi
```

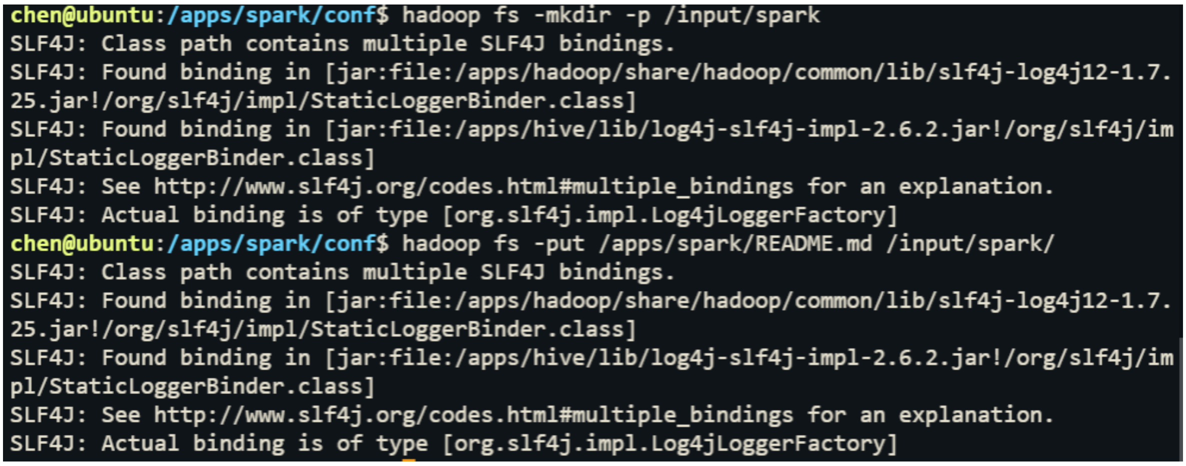


```
chen@ubuntu:/apps/spark/conf$ /apps/spark/bin/run-example SparkPi
20/11/24 19:35:35 WARN Utils: Your hostname, ubuntu resolves to a loopback address: 127.0
.1.1; using 10.0.0.135 instead (on interface ens33)
20/11/24 19:35:35 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
20/11/24 19:35:35 WARN NativeCodeLoader: Unable to load native-hadoop library for your pl
atform... using builtin-java classes where applicable
Pi is roughly 3.139755698778494
```

使用pyspark统计HDFS上的文件的行数

在 HDFS 上新建目录/input/spark 并上传文件 README.md 到该目录

```
hadoop fs -mkdir /input/spark/
hadoop fs -put /apps/spark/README.md /input/spark/
```



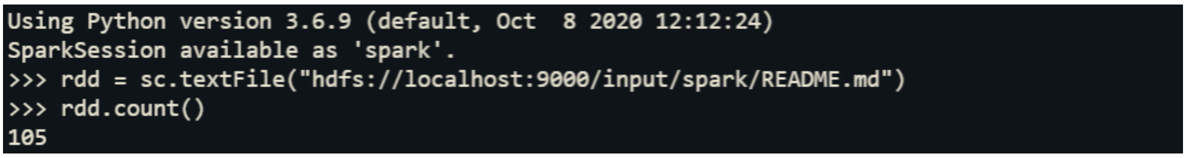
```
chen@ubuntu:/apps/spark/conf$ hadoop fs -mkdir -p /input/spark
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/apps/hadoop/share/hadoop/common/lib/slf4j-log4j12-1.7.
25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/apps/hive/lib/log4j-slf4j-impl-2.6.2.jar!/org/slf4j/im
pl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
chen@ubuntu:/apps/spark/conf$ hadoop fs -put /apps/spark/README.md /input/spark/
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/apps/hadoop/share/hadoop/common/lib/slf4j-log4j12-1.7.
25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/apps/hive/lib/log4j-slf4j-impl-2.6.2.jar!/org/slf4j/im
pl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
```

启动 pyspark

```
PYSPARK_PYTHON=python3 pyspark
```

使用 python 加载 HDFS 上的 README.md 文件，并转变为 RDD

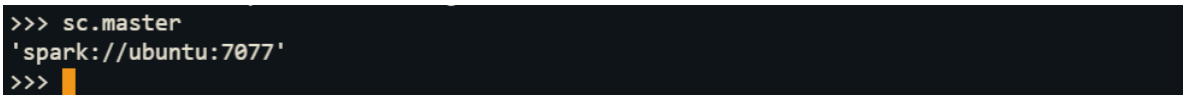
```
rdd = sc.textFile("hdfs://localhost:9000/input/spark/README.md")
rdd.count()
```



```
Using Python version 3.6.9 (default, Oct  8 2020 12:12:24)
SparkSession available as 'spark'.
>>> rdd = sc.textFile("hdfs://localhost:9000/input/spark/README.md")
>>> rdd.count()
105
```

查看当前运行模式

```
sc.master
```

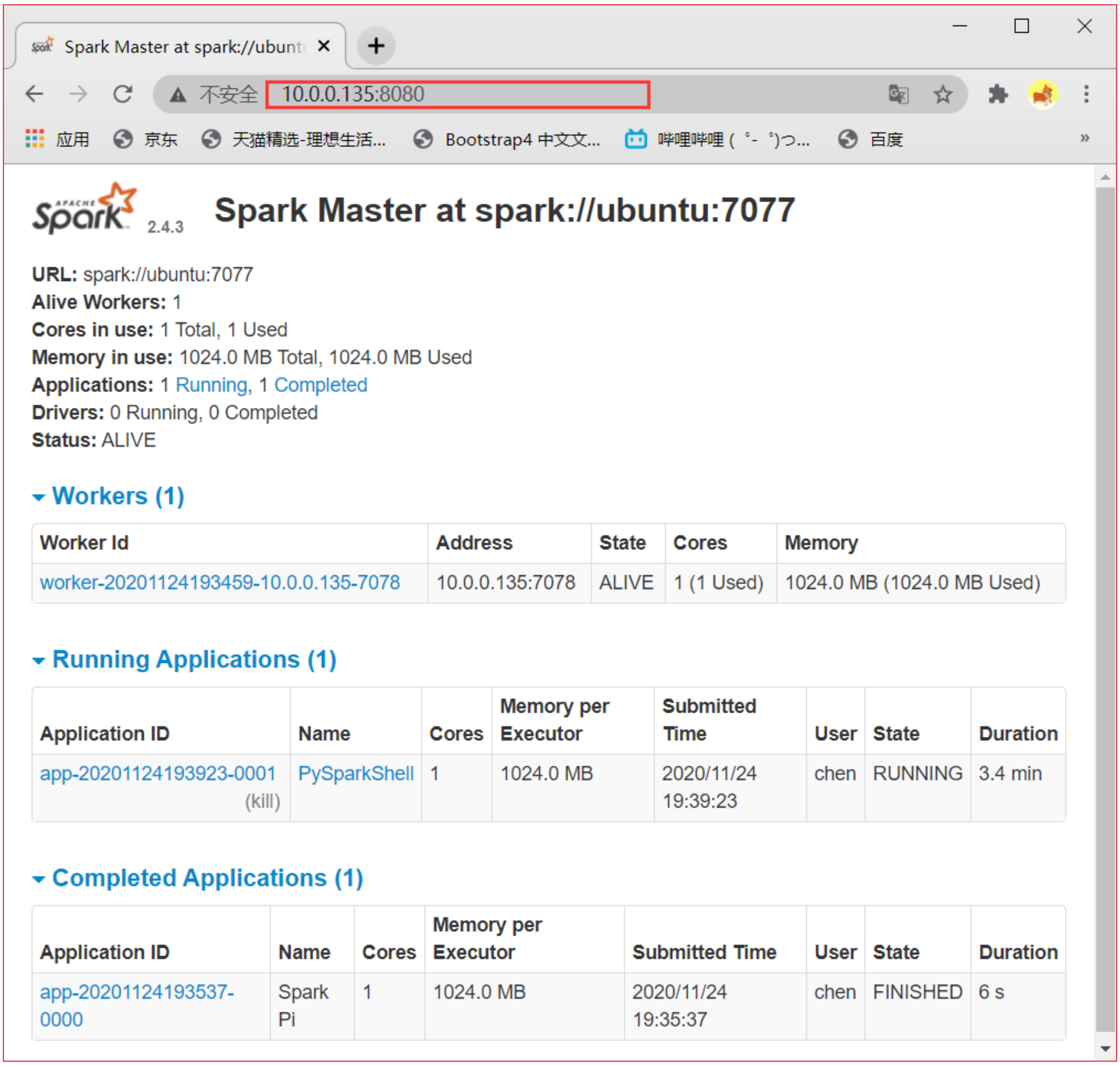


```
>>> sc.master
'spark://ubuntu:7077'
>>> 
```

因为我们在 spark-defaults.conf 中对主节点进行了设置，所以这里显示的运行模式不再 是 local。

三、Web界面

可以看到只有一个 worker，我们运行的例子显示的已完成的列表里



Spark Master at spark://ubuntu:7077

URL: spark://ubuntu:7077
Alive Workers: 1
Cores in use: 1 Total, 1 Used
Memory in use: 1024.0 MB Total, 1024.0 MB Used
Applications: 1 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (1)

Worker Id	Address	State	Cores	Memory
worker-20201124193459-10.0.0.135-7078	10.0.0.135:7078	ALIVE	1 (1 Used)	1024.0 MB (1024.0 MB Used)

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20201124193923-0001 (kill)	PySparkShell	1	1024.0 MB	2020/11/24 19:39:23	chen	RUNNING	3.4 min

Completed Applications (1)

Application ID	Name	Cores	Memory per Executor	Submitted Time	User	State	Duration
app-20201124193537-0000	Spark Pi	1	1024.0 MB	2020/11/24 19:35:37	chen	FINISHED	6 s

四、Jupyter notebook环境搭建

安装jupyter notebook

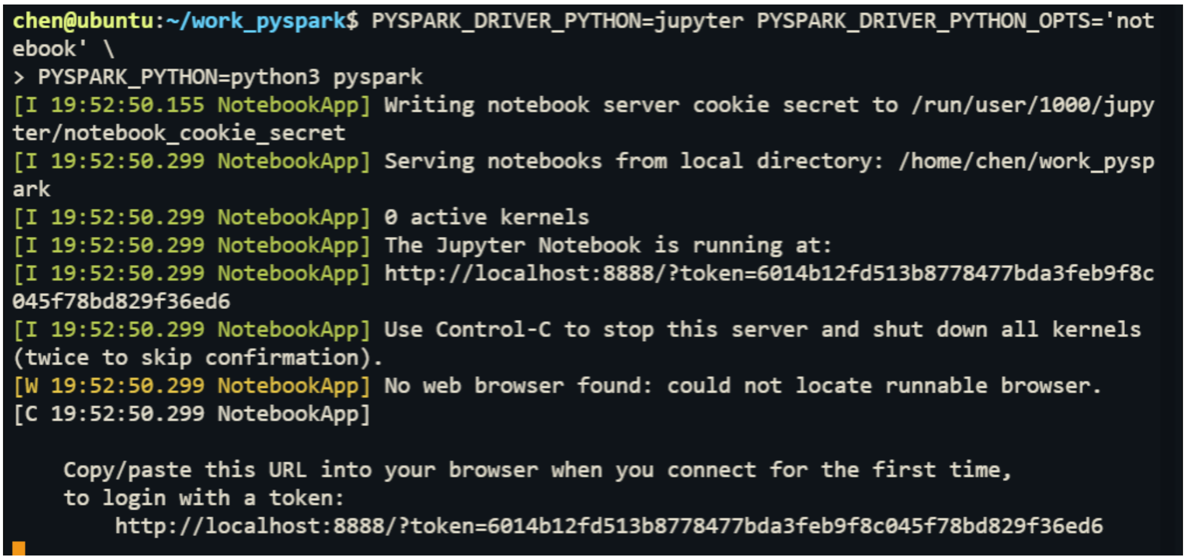
```
sudo apt install jupyter-notebook
```

新建工作目录~/work_pyspark

```
mkdir ~/work_pyspark
```

进入目录，执行以下命令在 jupyter notebook 中运行 spark

```
PYSPARK_DRIVER_PYTHON=jupyter PYSPARK_DRIVER_PYTHON_OPTS='notebook' \
PYSPARK_PYTHON=python3 pyspark
```



```
chen@ubuntu:~/work_pyspark$ PYSPARK_DRIVER_PYTHON=jupyter PYSPARK_DRIVER_PYTHON_OPTS='not
ebook' \
> PYSPARK_PYTHON=python3 pyspark
[I 19:52:50.155 NotebookApp] Writing notebook server cookie secret to /run/user/1000/jupy
ter/notebook_cookie_secret
[I 19:52:50.299 NotebookApp] Serving notebooks from local directory: /home/chen/work_pysp
ark
[I 19:52:50.299 NotebookApp] 0 active kernels
[I 19:52:50.299 NotebookApp] The Jupyter Notebook is running at:
[I 19:52:50.299 NotebookApp] http://localhost:8888/?token=6014b12fd513b8778477bda3feb9f8c
045f78bd829f36ed6
[I 19:52:50.299 NotebookApp] Use Control-C to stop this server and shut down all kernels
(twice to skip confirmation).
[W 19:52:50.299 NotebookApp] No web browser found: could not locate runnable browser.
[C 19:52:50.299 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://localhost:8888/?token=6014b12fd513b8778477bda3feb9f8c045f78bd829f36ed6
```

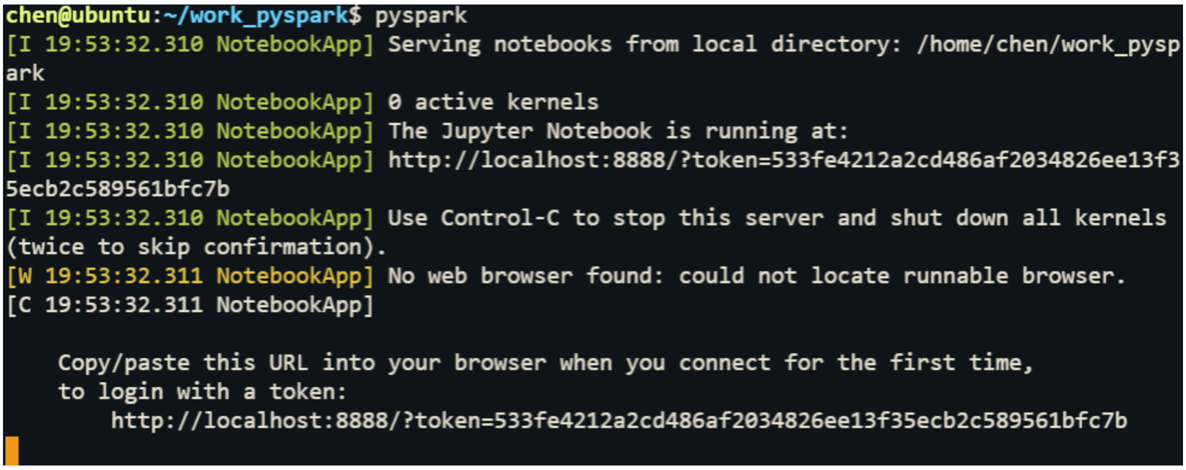
为方便起见，可以将下面的环境变量添加到~/.bashrc 中

```
export PYSPARK_DRIVER_PYTHON=jupyter
export PYSPARK_DRIVER_PYTHON_OPTS='notebook'
export PYSPARK_PYTHON=python3
```

使配置生效

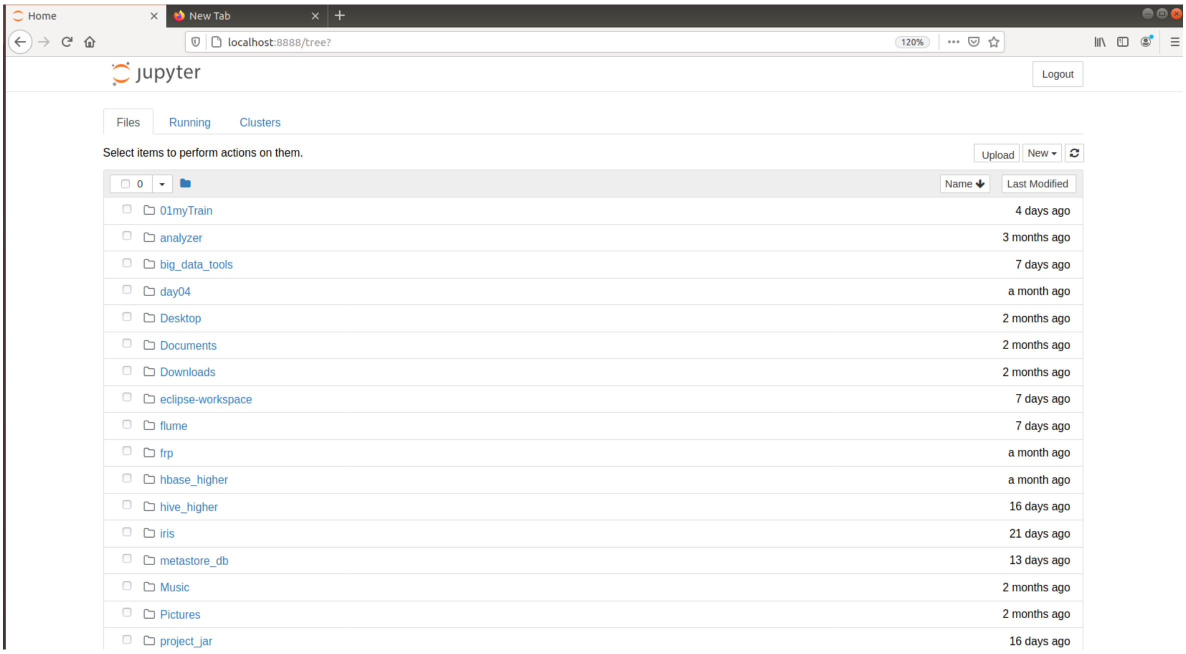
```
source ~/.bashrc
```

这样在终端中执行 pyspark，就默认在 jupyter notebook 中运行 spark。



```
chen@ubuntu:~/work_pyspark$ pyspark
[I 19:53:32.310 NotebookApp] Serving notebooks from local directory: /home/chen/work_pysp
ark
[I 19:53:32.310 NotebookApp] 0 active kernels
[I 19:53:32.310 NotebookApp] The Jupyter Notebook is running at:
[I 19:53:32.310 NotebookApp] http://localhost:8888/?token=533fe4212a2cd486af2034826ee13f3
5ecb2c589561bfc7b
[I 19:53:32.310 NotebookApp] Use Control-C to stop this server and shut down all kernels
(twice to skip confirmation).
[W 19:53:32.311 NotebookApp] No web browser found: could not locate runnable browser.
[C 19:53:32.311 NotebookApp]

    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://localhost:8888/?token=533fe4212a2cd486af2034826ee13f35ecb2c589561bfc7b
```



Select items to perform actions on them.

Name	Last Modified
01myTrain	4 days ago
analyzer	3 months ago
big_data_tools	7 days ago
day04	a month ago
Desktop	2 months ago
Documents	2 months ago
Downloads	2 months ago
eclipse-workspace	7 days ago
flume	7 days ago
frp	a month ago
hbase_higher	a month ago
hive_higher	16 days ago
iris	21 days ago
metastore_db	13 days ago
Music	2 months ago
Pictures	2 months ago
project_jar	16 days ago

新建一个工作目录

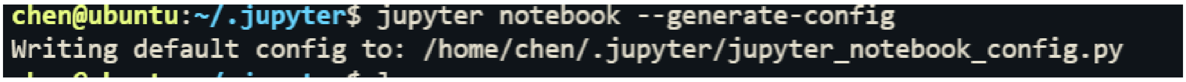
```
mkdir ~/pyspark-workspace
```

五、配置jupyter notebook

由于虚拟机内部使用图形界面体验感不够好,所以配置以下jupyter服务器,使宿主机可以直接访问notebook

- 生成配置文件

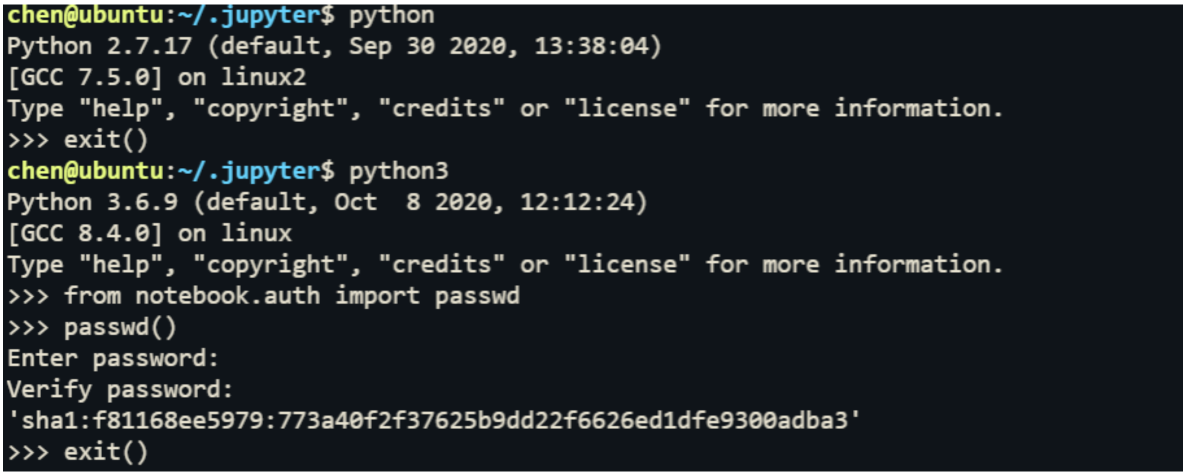
```
jupyter notebook --generate-config
```



```
chen@ubuntu:~/.jupyter$ jupyter notebook --generate-config
Writing default config to: /home/chen/.jupyter/jupyter_notebook_config.py
```

- 启动python3 shell,设置密码,并拷贝输出的sha1一行

```
python3
>>> from notebook.auth import passwd
>>> passwd()
Enter password:
Verify password:
'sha1:f81168ee5979:773a40f2f37625b9dd22f6626ed1dfe9300adba3'
>>> exit()
```



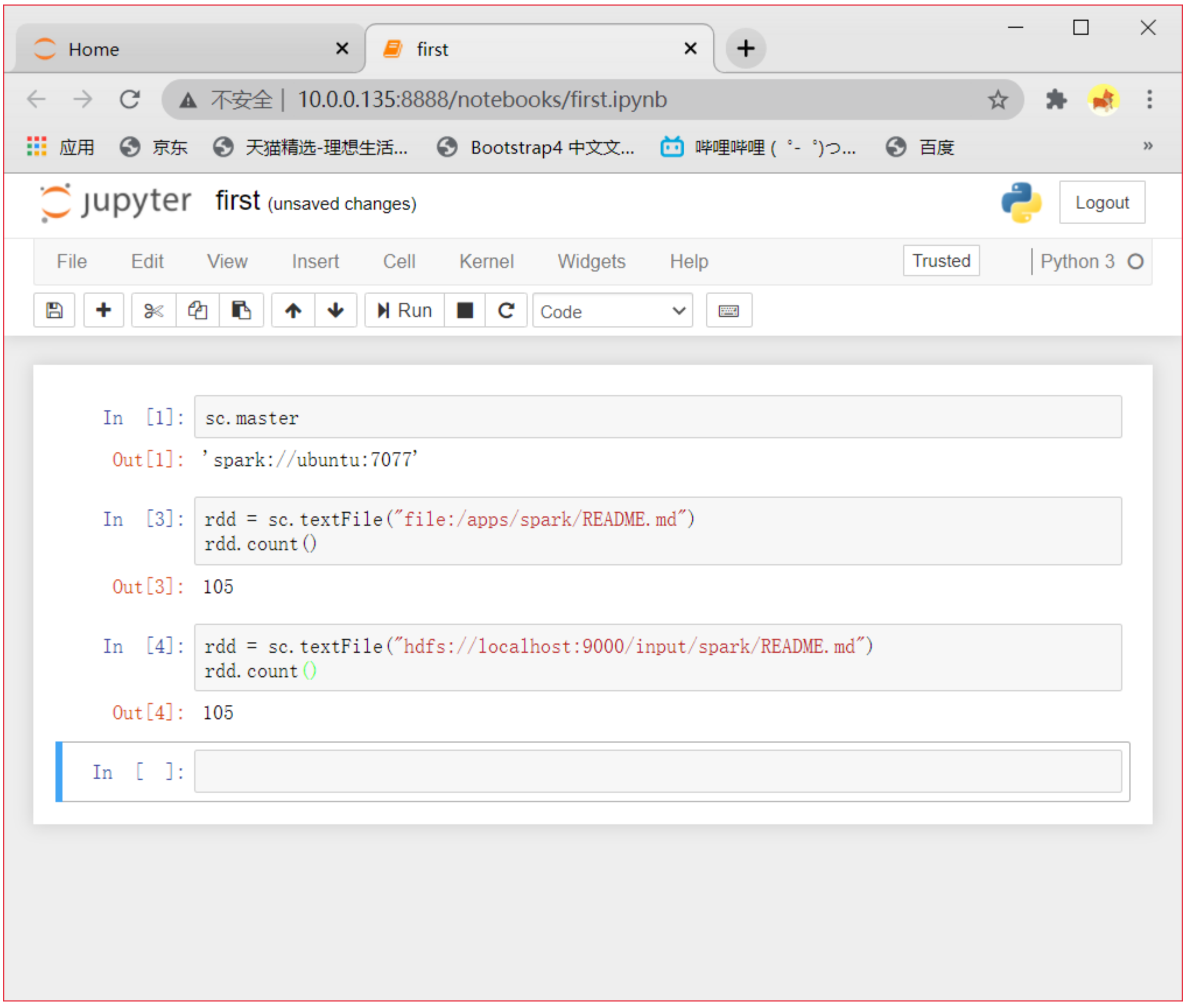
```
chen@ubuntu:~/.jupyter$ python
Python 2.7.17 (default, Sep 30 2020, 13:38:04)
[GCC 7.5.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
chen@ubuntu:~/.jupyter$ python3
Python 3.6.9 (default, Oct  8 2020, 12:12:24)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from notebook.auth import passwd
>>> passwd()
Enter password:
Verify password:
'sha1:f81168ee5979:773a40f2f37625b9dd22f6626ed1dfe9300adba3'
>>> exit()
```

- 编辑 /home/chen/.jupyter/jupyter_notebook_config.py,添加如下

```
c.NotebookApp.ip = "10.0.0.135" # 不知道为什么这里填127.0.0.1不可以,10.0.0.135是NAT模式的IP
c.NotebookApp.password=u"sha1:f81168ee5979:773a40f2f37625b9dd22f6626ed1dfe9300adba3"
c.NotebookApp.open_browser = False
c.NotebookApp.port=8888
```

- 在工作空间启动pyspark

```
cd ~/pyspark-workspace
pyspark
```



```
In [1]: sc.master
Out[1]: 'spark://ubuntu:7077'

In [3]: rdd = sc.textFile("file:/apps/spark/README.md")
        rdd.count()
Out[3]: 105

In [4]: rdd = sc.textFile("hdfs://localhost:9000/input/spark/README.md")
        rdd.count()
Out[4]: 105

In [ ]:
```

```
chen@ubuntu:~/pyspark-workspace$ ls
first.ipynb
```

六、pyspark RDD

RDD算子

- Transformation
  - Transformation是通过转换从一个或多个RDD生成新的RDD，该操作是lazy的，当调用action算则，才会发起job
  - 典型算子：map . flatMap .filter . reduceByKey 等
- Action
  - 当代码调用该类型算子时，立即启动job。
  - 典型算子：take count saveAsTextFile等

Transformation

Transformation API	**说明**
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，其中的每个值是将每个key传递到函数func中进行聚合。

Action

Action API	**说明**
count()	返回数据集中的元素的个数
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

七、Pyspark RDD

1.查看pyspark的版本号

```
print("pyspark version:"+str(sc.version))
```

```
pyspark version:2.4.3
```

2.使用parallelize创建RDD

```
x = sc.parallelize([1,2,3])
print(x.collect())
```

```
[1, 2, 3]
```

```
x = sc.parallelize(["apple","orange","banana"])
print(x.collect())
```

```
['apple', 'orange', 'banana']
```

3. map

map(f,prescrcesPartitioning=False):对RDD中每个元素进行f函数里面的操作，返回一个新的RDD。preservePartitioning表示是否保留父RDD的分区信息

```
x = sc.parallelize([1,2,3])
y = x.map(lambda x:(x,x**2))
print(x.collect())
print(y.collect())
```

```
[1, 2, 3]
[(1, 1), (2, 4), (3, 9)]
```

4.flatMap

flatMap(f, preservesPartitioning=False)：对 RDD 中每个元素进行 f 函数里面的操作，返回一个扁平化结果的新 RDD

```
x = sc.parallelize([1,3,3])
y = x.flatMap(lambda x:(x,x*100,x**2))
print(x.collect())
print(y.collect())
```

```
[1, 3, 3]
[1, 100, 1, 3, 300, 9, 3, 300, 9]
```

5.mapPatitions

mapPartitions(f, preservesPartitioning=False)：对 RDD 中每个分区里面的全部元素进行自定义f 函数操作，返回一个新 RDD Section ??

```
x = sc.parallelize([1,2,3],2)
def f(iterator):yield sum(iterator)

y = x.mapPartitions(f)
print(x.glom().collect())
print(y.glom().collect())
```

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

6.mapPartitionsWithIndex

mapPartitionsWithIndex(f, preservesPartitioning=False)：对 RDD 中每个分区里面的全部元素进行自定义 f 函数操作，并跟踪每个分区索引

```
x = sc.parallelize([1,2,3],2)
def f(partitionIndex,iterator):yield (partitionIndex,sum(iterator))

y = x.mapPartitionsWithIndex(f)
print(x.glom().collect())
print(y.glom().collect())
```

```
[[1], [2, 3]]
[[(0, 1)], [(1, 5)]]
```

7.分区数量

getNumPartition():返回分区的数量

```
x = sc.parallelize([1,2,3],2)
y = x.getNumPartitions()
print(x.glom().collect())
print(y)
```

```
[[1], [2, 3]]
2
```

8.filter

filter(f)：对 RDD 中的元素进行过滤，返回一个满足过滤条件的新 RDD

```
x = sc.parallelize([1,2,3])
y = x.filter(lambda x:x%2==1)
print(x.collect())
print(y.collect())
```

```
[1, 2, 3]
[1, 3]
```

9.distinct

distinct(numPartitions=None)：对 RDD 中的元素进行去重，返回去重后的新 RDD

```
x = sc.parallelize(["A","B","A","C"])
y = x.distinct()
print(x.collect())
print(y.collect())
```

```
['A', 'B', 'A', 'C']
['C', 'A', 'B']
```

10.sample

sample(withReplacement, fraction, seed=None)：对 RDD 进行抽样操作。

- withReplacement：是否有放回
- fraction：抽取的比率
- seed：随机生成的种子

```
x = sc.parallelize(range(7))
# call 'sample' 5 times
ylist = [x.sample(withReplacement=False, fraction=0.5) for i in range(5)]
print('x = ' + str(x.collect()))
for cnt,y in zip(range(len(ylist)), ylist):
    print('sample:' + str(cnt) + ' y = ' + str(y.collect()))
```

```
x = [0, 1, 2, 3, 4, 5, 6]
sample:0 y = [1, 2, 5, 6]
sample:1 y = [0, 1, 2, 6]
sample:2 y = [1, 2, 3, 4]
sample:3 y = [2, 4, 6]
sample:4 y = [0, 1, 3]
```

11.takeSample

takeSample(withReplacement, num, seed=None)：对 RDD 中元素进行抽样，返回抽样后 num 个元素。- withReplacement：是否有放回 - seed：随机种子数

```
x = sc.parallelize(range(7))
# call 'sample' 5 times
ylist = [x.takeSample(withReplacement=False, num=3) for i in range(5)]
print('x = ' + str(x.collect()))
for cnt,y in zip(range(len(ylist)), ylist):
    print('sample:' + str(cnt) + ' y = ' + str(y))
```

```
x = [0, 1, 2, 3, 4, 5, 6]
sample:0 y = [6, 2, 3]
sample:1 y = [3, 1, 2]
sample:2 y = [1, 4, 3]
sample:3 y = [0, 2, 4]
sample:4 y = [2, 5, 0]
```

12.union

union(other)：将自身 RDD 与其它 RDD 进行合并操作，返回一个新的 RDD

```
x = sc.parallelize(['A','A','B'])
y = sc.parallelize(['D','C','A'])
z = x.union(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
['A', 'A', 'B']
['D', 'C', 'A']
['A', 'A', 'B', 'D', 'C', 'A']
```

13.intersection

intersection：对自身 RDD 与其它 RDD 取交集

```
x = sc.parallelize(['A','A','B'])
y = sc.parallelize(['A','C','D'])
z = x.intersection(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
['A', 'A', 'B']
['A', 'C', 'D']
['A']
```

14.sortByKey

sortByKey(ascending=True, numPartitions=None, keyfunc)：对 RDD 按 key 值或对 key 操作的自定义 keyfunc 函数进行排序，默认为升序，numPartitions：分区的数目。

```
x = sc.parallelize([('B',1),('A',2),('C',3)])
y = x.sortByKey()
print(x.collect())
print(y.collect())
```

```
[('B', 1), ('A', 2), ('C', 3)]
[('A', 2), ('B', 1), ('C', 3)]
```

15.sortBy

sortBy(keyfunc, ascending=True, numPartitions=None)：按自定义的 keyfunc 函数对 RDD 中元素进行排序，默认为升序

```
x = sc.parallelize(['Cat','Apple','Bat'])
def keyGen(val): return val[0]
y = x.sortBy(keyGen)
print(y.collect())
```

```
['Apple', 'Bat', 'Cat']
```

16.glom

glom()：创建一个新的 RDD，通过合并每个分区里面的全部元素到一个列表中

```
x = sc.parallelize(['C','B','A'], 2)
y = x.glom()
print(x.collect())
print(y.collect())
```

```
['C', 'B', 'A']
[['C'], ['B', 'A']]
```

17.cartesian

cartesian(other)：将 RDD 与其它 RDD 进行笛卡尔积，返回 <key,value> 类型的 RDD，其中 key为自身的元素，value 为其它 RDD 的元素。

```
x = sc.parallelize(['A','B'])
y = sc.parallelize(['C','D'])
z = x.cartesian(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
['A', 'B']
['C', 'D']
[('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D')]
```

18.groupBy

groupBy(f, numPartitions=None, partitionFunc)：对 RDD 中每个元素按照满足自定义的 f 函数为条件进行分组，返回一个新的 <key,value&gt 类型的 RDD，其中 key 为 f 函数的返回值。

```
x = sc.parallelize([1,2,3])
y = x.groupBy(lambda x: 'A' if (x%2 == 1) else 'B' )
print(x.collect())
print(y.collect())
print([(j[0],[i for i in j[1]]) for j in y.collect()])
```

```
[1, 2, 3]
[('A', <pyspark.resultiterable.ResultIterable object at 0x7f380801f2e8>), ('B',
<pyspark.resultiterable.ResultIterable object at 0x7f380801ff98>)]
[('A', [1, 3]), ('B', [2])]
```

19.pipe

pipe(command, env=None, checkCode=False)：对 RDD 元素进行管道操作，将返回 shell 命令的处理结果，形成一个新的 RDD

```
x = sc.parallelize(['A', 'Ba', 'C', 'AD'])
y = x.pipe('grep "A"')
y1 = x.pipe('grep -i "a"')
print(x.collect())
print(y.collect())
print(y1.collect())
```

```
['A', 'Ba', 'C', 'AD']
['A', 'AD']
['A', 'Ba', 'AD']
```

20.foreach

foreach(f)：对 RDD 中每个元素进行自定义 f 函数输出操作

```
x = sc.parallelize([1,2,3])
def f(el):
    '''side effect: append the current RDD elements to a file'''
    f1 = open("/home/chen/foreachExample.txt", 'a+')
    print(el,file=f1)
```

```
# first clear the file contents
open('/home/chen/foreachExample.txt', 'w').close()
y = x.foreach(f) # writes into foreachExample.txt
print(x.collect())
print(y) # foreach returns 'None'
# print the contents of foreachExample.txt
with open("/home/chen/foreachExample.txt", "r") as foreachExample:
    print (foreachExample.read())
```

```
[1, 2, 3]
None
1
2
3
```

21.foreachPartition

foreachPartition(f)：对 RDD 每个分区中元素进行自定义 f 函数操作。

```
x = sc.parallelize([1,2,3,4,5,6],5)
def f(parition):
    '''side effect: append the current RDD partition contents to a file'''
    f1=open("/home/chen/foreachPartitionExample.txt", 'a+')
    print([el for el in parition],file=f1)
```

```
open('/home/chen/foreachPartitionExample.txt', 'w').close()
y = x.foreachPartition(f) # writes into foreachExample.txt
print(x.glom().collect())
print(y) # foreach returns 'None'
# print the contents of foreachExample.txt
with open("/home/chen/foreachPartitionExample.txt", "r") as foreachExample:
    print (foreachExample.read())
```

```
[[1], [2], [3], [4], [5, 6]]
None
[1]
[2]
[3]
[4]
[5, 6]
```

22.reduce

reduce(f)：使用指定的二元运算符，对 RDD 中每个元素进行 reduce 操作

```
x = sc.parallelize([1,2,3])
y = x.reduce(lambda x, y: x + y) # computes a cumulative sum
print(x.collect())
print(y)
```

```
[1, 2, 3]
6
```

23.fold

fold(zeroValue, op)：对 RDD 中每个元素进行聚合操作。使用一个函数和零值，先对每个分区的元素进行聚合，然后对全部分区进行聚合。

```
x = sc.parallelize([1,2,3],2)
neutral_zero_value = 0 # 0 for sum, 1 for multiplication
y = x.fold(neutral_zero_value,lambda x, y: x + y) # computes cumulative sum
print(x.glom().collect())
print(y)
```

```
[[1], [2, 3]]
6
```

24.aggregate

aggregate(zeroValue, seqOp, combOp)：使用一个合并函数和一个零值，先对每个分区按合并函数进行聚合，然后将全部分区进行聚合。- seqOp: 每个分区执行的聚合函数, 对 rdd 中按分区每个元素 y 执行此函数, x 为上一次的执行结果, 首次计算时使用默认值 zeroValue

- comOp: 对每个分区的结果执行的聚合函数, 执行此函数时, 每个分区的计算结果 y 执行此函数, x 为上一次的执行结果, 首次计算时使用默认值 zeroValue

```
x = sc.parallelize([2,3,4])
neutral_zero_value = (0,1) # sum: x = x+0, product: x = 1*x
seqOp = (lambda x, y: (x[0] + y, x[1] * y))
combOp = (lambda x, y: (x[0] + y[0], x[1] * y[1]))
y = x.aggregate(neutral_zero_value,seqOp,combOp) # computes (cumulative sum,cumulative product)
print(x.collect())
print(y)
```

```
[2, 3, 4]
(9, 24)
```

25.max

max(key=None)：找寻 RDD 中最大的一项，参数 key: 一个函数用于生成比较的关键条件

```
x = sc.parallelize([1,3,2])
y = x.max()
print(x.collect())
print(y)
```

```
[1, 3, 2]
3
```

26.min

min(key=None)：找寻 RDD 中最小的一项，参数 key: 一个函数用于生成比较的关键条件

```
x = sc.parallelize([1,3,2])
y = x.min()
print(x.collect())
print(y)
```

```
[1, 3, 2]
1
```

27.sum

sum()：对 RDD 中所有元素进行累加求和

```
x = sc.parallelize([1,3,2])
y = x.sum()
print(x.collect())
print(y)
```

```
[1, 3, 2]
6
```

28.count

count()：计算 RDD 中元素的个数

```
x = sc.parallelize([1,3,2])
y = x.count()
print(x.collect())
print(y)
```

```
[1, 3, 2]
3
```

29.histogram

histogram(buckets)：使用提供的桶计算直方图。例如 [1,10,20,50] 意思是桶 [1,10) [10,20) [20,50]

```
x = sc.parallelize([1,3,1,2,3])
y = x.histogram(buckets = 2)
print(x.collect())
print(y)
```

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

```
x = sc.parallelize([1,3,1,2,3])
y = x.histogram([0,0.5,1,1.5,2,2.5,3,3.5])
print(x.collect())
print(y)
```

```
[1, 3, 1, 2, 3]
([0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5], [0, 0, 2, 0, 1, 0, 2])
```

30.mean

mean()：计算 RDD 中元素的平均值

```
x = sc.parallelize([1,3,2])
y = x.mean()
print(x.collect())
print(y)
```

```
[1, 3, 2]
2.0
```

31.variance

variance()：计算 RDD 中所有元素的方差

```
x = sc.parallelize([1,3,2])
y = x.variance() # divides by N
print(x.collect())
print(y)
```

```
[1, 3, 2]
0.6666666666666666
```

32.stdev

stdev()：计算 RDD 中所有元素的标准差

```
x = sc.parallelize([1,3,2])
y = x.stdev() # divides by N
print(x.collect())
print(y)
```

```
[1, 3, 2]
0.816496580927726
```

33.sampleStdev

sampleStdev()：计算 RDD 中所有元素的样本标准差

```
x = sc.parallelize([1,3,2])
y = x.sampleStdev() # divides by N-1
print(x.collect())
print(y)
```

```
[1, 3, 2]
1.0
```

34.sampleVariance

sampleVariance()：计算 RDD 中所有元素的样本方差

```
x = sc.parallelize([1,3,2])
y = x.sampleVariance() # divides by N-1
print(x.collect())
print(y)
```

```
[1, 3, 2]
1.0
```

35.countByValue

countByValue()：对 RDD 中每个元素进行计数

```
x = sc.parallelize(["A","C","A","B","C"])
y = x.countByValue()
print(x.collect())
print(y)
```

```
['A', 'C', 'A', 'B', 'C']
defaultdict(<class 'int'>, {'A': 2, 'C': 2, 'B': 1})
```

36.top

top(num, key=None)：对 RDD 中元素按降序或自定义 key 函数进行排序，输出排序后的前 num 个元素

```
x = sc.parallelize([1,3,1,2,3])
y = x.top(num = 3)
print(x.collect())
print(y)
```

```
[1, 3, 1, 2, 3]
[3, 3, 2]
```

37.takeOrdered

takeOrdered(num, key=None)：对 RDD 中元素按升序或自定义 key 函数进行排序，输出排序后的前 num 个元素。

```
x = sc.parallelize([1,3,1,2,3])
y = x.takeOrdered(num = 3)
print(x.collect())
print(y)
```

```
[1, 3, 1, 2, 3]
[1, 1, 2]
```

38.take

take(num)：输出 RDD 中前 num 个元素

```
x = sc.parallelize([1,3,1,2,3])
y = x.take(num = 3)
print(x.collect())
print(y)
```

```
[1, 3, 1, 2, 3]
[1, 3, 1]
```

39.first

first()：输出 RDD 中第一个元素

```
x = sc.parallelize([1,3,1,2,3])
y = x.first()
print(x.collect())
print(y)
```

```
[1, 3, 1, 2, 3]
1
```

40.collectAsMap

collectAsMap()：对 RDD 的每个元素进行遍历，返回一个键值对类型的字典。

```
x = sc.parallelize([('C',3),('A',1),('B',2)])
y = x.collectAsMap()
print(x.collect())
print(y)
```

```
[('C', 3), ('A', 1), ('B', 2)]
{'C': 3, 'A': 1, 'B': 2}
```

41.keys

keys()：对 <key,value> 类型 RDD 进行操作，返回 RDD 每个元素的 key 值。

```
x = sc.parallelize([('C',3),('A',1),('B',2)])
y = x.keys()
print(x.collect())
print(y.collect())
```

```
[('C', 3), ('A', 1), ('B', 2)]
['C', 'A', 'B']
```

42.values

values()：对 <key,value> 类型的 RDD 进行操作，返回 RDD 每个元素的 value 值。

```
x = sc.parallelize([('C',3),('A',1),('B',2)])
y = x.values()
print(x.collect())
print(y.collect())
```

```
[('C', 3), ('A', 1), ('B', 2)]
[3, 1, 2]
```

43.reduceByKey

reduceByKey(func, numPartitions=None, partitionFunc)：对 pairRDD 中的 key 先进行 group by 操作，然后对聚合后的 value 数据进行自定义 f 函数操作，返回一个新的 RDD

```
x = sc.parallelize([('B',1),('B',2),('A',3),('A',4),('A',5)])
y = x.reduceByKey(lambda agg, obj: agg + obj)
print(x.collect())
print(y.collect())
```

```
[('B', 1), ('B', 2), ('A', 3), ('A', 4), ('A', 5)]
[('B', 3), ('A', 12)]
```

44.countByKey

countByKey()：对 key 相同的所有元素进行计数，返回值为一个字典

```
x = sc.parallelize([('B',1),('B',2),('A',3),('A',4),('A',5)])
y = x.countByKey()
print(x.collect())
print(y)
```

```
[('B', 1), ('B', 2), ('A', 3), ('A', 4), ('A', 5)]
defaultdict(<class 'int'>, {'B': 2, 'A': 3})
```

45.join

join(other, numPartitions=None)：对 RDD 上的每个元素与其它 RDD 进行 join 操作，返回一个 (k, (v1, v2)) 类型的新 RDD，其中 (k, v1) 在自身 RDD，(k, v2) 在其它 RDD。

```
x = sc.parallelize([('C',4),('B',3),('A',2),('A',1)])
y = sc.parallelize([('A',8),('B',7),('A',6),('D',5)])
z = x.join(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
[('C', 4), ('B', 3), ('A', 2), ('A', 1)]
[('A', 8), ('B', 7), ('A', 6), ('D', 5)]
[('B', (3, 7)), ('A', (2, 8)), ('A', (2, 6)), ('A', (1, 8)), ('A', (1, 6))]
```

46.leftOuterJoin

leftOuterJoin(other, numPartitions=None)：执行自身 RDD 与其他 RDD 的 left outer join 操作，例如自身 RDD 每个元素为 <k,v>，其他 RDD 每个元素为 <k,w>，返回新的 RDD 中包含全部的 pairs(k, (v, w)) 或者 pair(k, (v, None))。numPartitions：进行 Hash 分区的数量

```
x = sc.parallelize([('C',4),('B',3),('A',2),('A',1)])
y = sc.parallelize([('A',8),('B',7),('A',6),('D',5)])
z = x.leftOuterJoin(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
[('C', 4), ('B', 3), ('A', 2), ('A', 1)]
[('A', 8), ('B', 7), ('A', 6), ('D', 5)]
[('B', (3, 7)), ('A', (2, 8)), ('A', (2, 6)), ('A', (1, 8)), ('A', (1, 6)),
('C', (4, None))]
```

47.rightOuterJoin

rightOuterJoin：执行自身 RDD 与其他 RDD 的 right outer join 操作，例如自身 RDD 每个元素为 <k,v>，其他 RDD 每个元素为 <k,w>，返回新的 RDD 中包含全部的 pairs(k, (v, w)) 或者 pair(k, (None, w))。numPartitions：进行 Hash 分区的数量

```
x = sc.parallelize([('C',4),('B',3),('A',2),('A',1)])
y = sc.parallelize([('A',8),('B',7),('A',6),('D',5)])
z = x.rightOuterJoin(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
[('C', 4), ('B', 3), ('A', 2), ('A', 1)]
[('A', 8), ('B', 7), ('A', 6), ('D', 5)]
[('B', (3, 7)), ('A', (2, 8)), ('A', (2, 6)), ('A', (1, 8)), ('A', (1, 6)),
('D', (None, 5))]
```

48.partitionBy

partitionBy(numPartitions, partitionFunc)：对 RDD 进行分区。numPartitions：分区的数目，partitionFunc：自定义分区函数，partitionFunc(k) % numPartitions 的值为新分区的索引 index

```
# partitionBy Example1
x = sc.parallelize([(0,1),(1,2),(2,3)],2)
y = x.partitionBy(numPartitions = 3, partitionFunc = lambda x: x) # only key␣is
passed to paritionFunc
print(x.glom().collect())
print(y.glom().collect())
```

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

```
# partitionBy Example2
x = sc.parallelize([("hadoop",1),("spark",2),("python",3),("C",4)],2)
y = x.partitionBy(numPartitions = 3, partitionFunc = lambda x: len(x)) #
only key is passed to paritionFunc
print(x.glom().collect())
print(y.glom().collect())
```

```
[[('hadoop', 1), ('spark', 2)], [('python', 3), ('C', 4)]]
[[('hadoop', 1), ('python', 3)], [('C', 4)], [('spark', 2)]]
```

49.combineByKey

combineByKey(createCombiner, mergeValue, mergeCombiners, numPartitions=None, partitionFunc)：泛型函数使用一个自定义的聚合函数，去合并 RDD 中每个 key 相同的元素，具体为转换 RDD[(K, V)] 形成一个新的 RDD[(K, C)]，其中 C 是一个合并类型。createCombiner：创建一个 V 到 C 的函数，mergeValue：将一个 V 形成一个 C，mergeCombiners：将 C 的集合进行合并。

```
x = sc.parallelize([('B',1),('B',2),('A',3),('A',4),('A',5)])
createCombiner = (lambda el: [(el,el**2)])
mergeVal = (lambda aggregated, el: aggregated + [(el,el**2)]) # append
to aggregated
mergeComb = (lambda agg1,agg2: agg1 + agg2 ) # append agg1 with agg2
y = x.combineByKey(createCombiner,mergeVal,mergeComb)
print(x.collect())
print(y.collect())
```

```
[('B', 1), ('B', 2), ('A', 3), ('A', 4), ('A', 5)]
[('B', [(1, 1), (2, 4)]), ('A', [(3, 9), (4, 16), (5, 25)])]
```

50.aggregateByKey

aggregateByKey(zeroValue, seqFunc, combFunc, numPartitions=None, partitionFunc)：聚合每个键的值, 使用组合函数和一个零值，函数返回一个不同类型的 rdd。seqFunc：是对一个分区里每个键的值聚合，combFunc：是对分区间每个键的聚合结果进行聚合。

```
x = sc.parallelize([('B',1),('B',2),('A',3),('A',4),('A',5)])
zeroValue = [] # empty list is 'zero value' for append operation
mergeVal = (lambda aggregated, el: aggregated + [(el,el**2)])
mergeComb = (lambda agg1,agg2: agg1 + agg2 )
y = x.aggregateByKey(zeroValue,mergeVal,mergeComb)
print(x.collect())
print(y.collect())
```

```
[('B', 1), ('B', 2), ('A', 3), ('A', 4), ('A', 5)]
[('B', [(1, 1), (2, 4)]), ('A', [(3, 9), (4, 16), (5, 25)])]
```

51.foldByKey

foldByKey(zeroValue, func, numPartitions=None, partitionFunc)：使用一个组合函数 func 与一个零值，对 key 相同的 value 值进行聚合

```
x = sc.parallelize([('B',1),('B',2),('A',3),('A',4),('A',5)])
zeroValue = 1 # one is 'zero value' for multiplication
y = x.foldByKey(zeroValue,lambda agg,x: agg*x ) # computes cumulative
product␣,→within each key
print(x.collect())
print(y.collect())
```

```
[('B', 1), ('B', 2), ('A', 3), ('A', 4), ('A', 5)]
[('B', 2), ('A', 60)]
```

52.groupByKey

groupByKey(numPartitions=None, partitionFunc)：对 RDD 里 key 相同的元素进行分组，分组结果形成一个序列，最后返回一个新的 <key,value> 类型的 RDD。numPartitions：进行 Hash 分区的分区数

```
x = sc.parallelize([('B',5),('B',4),('A',3),('A',2),('A',1)])
y = x.groupByKey()
print(x.collect())
print([(j[0],[i for i in j[1]]) for j in y.collect()])
```

```
[('B', 5), ('B', 4), ('A', 3), ('A', 2), ('A', 1)]
[('B', [5, 4]), ('A', [3, 2, 1])]
```

53.flatMapValues

flatMapValues(f)：使用一个 flatMap 函数，对类型 RDD 中 key 相同的 value 值进行操作，返回一个新的 RDD。新 RDD 的 key 值不变，只改变了 value 值，还保留了原始 RDD 的分区。

```
x = sc.parallelize([('A',(1,2,3)),('B',(4,5))])
y = x.flatMapValues(lambda x: [i**2 for i in x]) # function is applied
to␣,→entire value, then result is flattened
print(x.collect())
print(y.collect())
```

```
[('A', (1, 2, 3)), ('B', (4, 5))]
[('A', 1), ('A', 4), ('A', 9), ('B', 16), ('B', 25)]
```

54.mapValues

mapValues：对键值对 <key,value> 中的 value 部分执行函数里面的操作，返回 <key,value> 键值对形式的新 RDD

```
x = sc.parallelize([('A',(1,2,3)),('B',(4,5))])
y = x.mapValues(lambda x: [i**2 for i in x]) # function is applied to
entire␣,→value
print(x.collect())
print(y.collect())
```

```
[('A', (1, 2, 3)), ('B', (4, 5))]
[('A', [1, 4, 9]), ('B', [16, 25])]
```

55.cogroup

cogroup(other)：对两个 RDD 数据集按 key 相同的数据进行 group by，并对 key 值相同的数据中每个 RDD 的 value 进行单独 group by

```
x = sc.parallelize([('C',4),('B',(3,3)),('A',2),('A',(1,1))])
y = sc.parallelize([('A',8),('B',7),('A',6),('D',(5,5))])
z = x.cogroup(y)
print(x.collect())
print(y.collect())
for key,val in list(z.collect()):
    print(key, [list(i) for i in val])
```

```
[('C', 4), ('B', (3, 3)), ('A', 2), ('A', (1, 1))]
[('A', 8), ('B', 7), ('A', 6), ('D', (5, 5))]
B [[(3, 3)], [7]]
A [[2, (1, 1)], [8, 6]]
C [[4], []]
D [[], [(5, 5)]]
```

56.groupWith

groupWith(other, *others)：类似于 cogroup 操作，但支持多个 RDD。返回类型为 RDD。

```
x = sc.parallelize([('C',4),('B',(3,3)),('A',2),('A',(1,1))])
y = sc.parallelize([('B',(7,7)),('A',6),('D',(5,5))])
z = sc.parallelize([('D',9),('B',(8,8))])
a = x.groupWith(y,z)
print(x.collect())
print(y.collect())
print(z.collect())
print("Result:")
for key,val in list(a.collect()):
    print(key, [list(i) for i in val])
```

```
[('C', 4), ('B', (3, 3)), ('A', 2), ('A', (1, 1))]
[('B', (7, 7)), ('A', 6), ('D', (5, 5))]
[('D', 9), ('B', (8, 8))]
Result:
C [[4], [], []]
A [[2, (1, 1)], [6], []]
B [[(3, 3)], [(7, 7)], [(8, 8)]]
D [[], [(5, 5)], [9]]
```

57.sampleByKey

sampleByKey(withReplacement, fractions, seed=None)：以 key 值对元素进行抽样，返回一个新 RDD，withReplacement：表示是否有放回，True 表示有放回，fractions：key 值得抽样率，seed：随机种子

```
x = sc.parallelize([('A',1),('B',2),('C',3),('B',4),('A',5)])
y = x.sampleByKey(withReplacement=False, fractions={'A':0.5, 'B':1, 'C':0.2})
print(x.collect())
print(y.collect())
```

```
[('A', 1), ('B', 2), ('C', 3), ('B', 4), ('A', 5)]
[('B', 2), ('C', 3), ('B', 4), ('A', 5)]
```

58.subtractByKey

subtractByKey(other, numPartitions=None)：按 key 值对 RDD 进行扣除操作，返回自身 <key,value> 类型 RDD 不匹配其他 RDD 中 key 的部分

```
x = sc.parallelize([('C',1),('B',2),('A',3),('A',4)])
y = sc.parallelize([('A',5),('D',6),('A',7),('D',8)])
z = x.subtractByKey(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
[('C', 1), ('B', 2), ('A', 3), ('A', 4)]
[('A', 5), ('D', 6), ('A', 7), ('D', 8)]
[('B', 2), ('C', 1)]
```

59.subtract

subtract(other, numPartitions=None)：返回自身 RDD 中不匹配其他 RDD 的部分

```
x = sc.parallelize([('C',4),('B',3),('A',2),('A',1)])
y = sc.parallelize([('C',8),('A',2),('D',1)])
z = x.subtract(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
[('C', 4), ('B', 3), ('A', 2), ('A', 1)]
[('C', 8), ('A', 2), ('D', 1)]
[('C', 4), ('A', 1), ('B', 3)]
```

60.keyBy

keyBy(f)：使用自定义函数 f，创建每个元素为元组类型的新 RDD，f 函数的返回值作为 key，RDD 的元素值作为 value。

```
x = sc.parallelize([1,2,3])
y = x.keyBy(lambda x: x**2)
print(x.collect())
print(y.collect())
```

```
[1, 2, 3]
[(1, 1), (4, 2), (9, 3)]
```

61.repartition

repartition(numPartitions)：对 RDD 进行分区。numPartitions：为分区数

```
x = sc.parallelize([1,2,3,4,5],2)
y = x.repartition(numPartitions=3)
print(x.glom().collect())
print(y.glom().collect())
```

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

62.coalesce

coalesce(numPartitions, shuffle=False)：对 RDD 进行分区，将分区数减少到 numPartitions。

```
x = sc.parallelize([1,2,3,4,5],2)
y = x.coalesce(numPartitions=1)
print(x.glom().collect())
print(y.glom().collect())
```

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

63.zip

zip(other)：将 RDD 与其它 RDD 进行 zip 操作，返回 <key,value> 类型的新 RDD，其中 key 为自身 RDD 的元素值，value 为其它 RDD 的元素值

```
x = sc.parallelize(['B','A','A'])
# zip expects x and y to have same #partitions and #elements/partition
y = x.map(lambda x: ord(x))
z = x.zip(y)
print(x.collect())
print(y.collect())
print(z.collect())
```

```
['B', 'A', 'A']
[66, 65, 65]
[('B', 66), ('A', 65), ('A', 65)]
```

64.zipWithIndex

zipWithIndex()：对 RDD 进行 zip 操作，返回新的 RDD 中，每个元素包含原 RDD 的元素值还有对应的索引。

```
x = sc.parallelize(['B','A','A'],2)
y = x.zipWithIndex()
print(x.glom().collect())
print(y.collect())
```

```
[['B'], ['A', 'A']]
[('B', 0), ('A', 1), ('A', 2)]
```

65. zipWithUniqueId

zipWithUniqueId()：对 RDD 进行 zip 操作，返回 <key,value> 类型新 RDD，key 为原 RDD 的元素值，value 为从 0 开始的值得 id
[70]: x = sc.parallelize(['B','A','A']

```
x = sc.parallelize(['B','A','A'],2)
y = x.zipWithUniqueId()
print(x.glom().collect())
print(y.collect())

```

```
[['B'], ['A', 'A']]
[('B', 0), ('A', 1), ('A', 3)]
```

66.WordCount

```
## 从dhfs加载文件
textFile = sc.textFile("hdfs://localhost:9000/input/wordcount/testfile")
```

```
## 以空格为分界符分词
stringRDD = textFile.flatMap(lambda line:line.split(" "))
```

```
## map执行的是将每个词都变为（词，1）这样的二元组，
## reduceByKey执行的是将key相同的二元组的value相加
countsRDD = stringRDD.map(lambda word:(word,1)).reduceByKey(lambda x,y:x+y)
```

```
countsRDD.collect()
```

```
[('hadoop', 1), ('world', 1), ('python', 1), ('hello', 4), ('spark', 1)]
```