
# 大数据应用平台实验指导书

梅海彬　编著

上海海洋大学海洋智能信息实验教学示范中心

# 目录

# 实验一 Linux 基本操作

## 一、实验目的

1.学会 Linux 系统的基本命令；

2.学会 Linux 系统的目录结构；

3.学会 Linux 系统的文件权限管理；

4.学会 Linux 系统的软件安装方式。

## 二、实验原理

Linux 内核最初只是由芬兰人李纳斯·托瓦兹（Linus Torvalds）在赫尔辛基大学上学时出于个人爱好而编写的。Linux 是一套免费使用和自由传播的类 Unix 操作系统，是一个基于 POSIX 和 UNIX 的多用户、多任务、支持多线程和多 CPU 的操作系统。

Linux 能运行主要的 UNIX 工具软件、应用程序和网络协议。它支持 32 位和 64 位硬件。Linux 继承了 Unix 以网络为核心的设计思想，是一个性能稳定的多用户网络操作系统。

市面上较知名的发行版有：Ubuntu、RedHat、CentOS、Debian、Fedora、SuSE、OpenSUSE、Arch Linux、SolusOS 等。

## 三、实验环境

1.Windows 操作系统

2.虚拟机软件 VirtualBox

3.Linux ubuntu 16.04

## 四、实验内容

1.学习 Linux 的基础命令、管理与软件安装。

2.学习 Linux 的文件系统目录结构与文件权限。

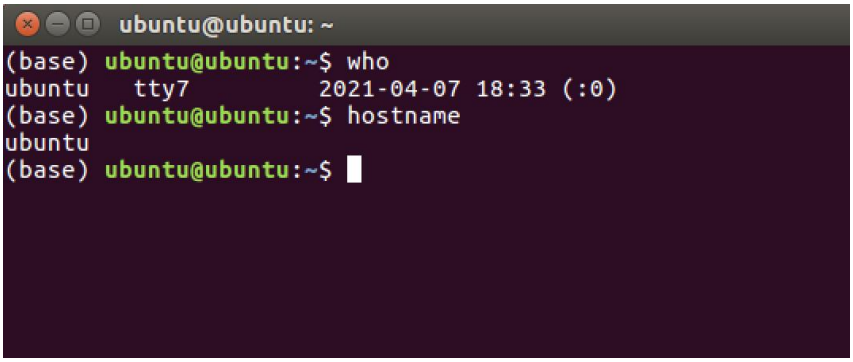
## 五、实验步骤

注：如果提示权限不够，就在命令前加上 sudo，临时提升用户权限，

虚拟机安装详见：VirtualBox 安装 ubuntu16 LTS 及其配置

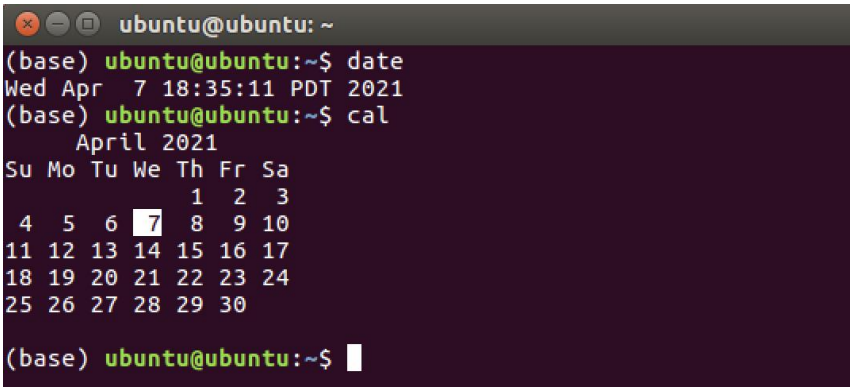
1. 查看用户和主机名

- who
- hostname




2. 查看日期

- date
- cal




3. 目录及文件基本操作

- cd 和 pwd 命令

（1）切换到目录/usr/local

（2）去到目前的上层目录

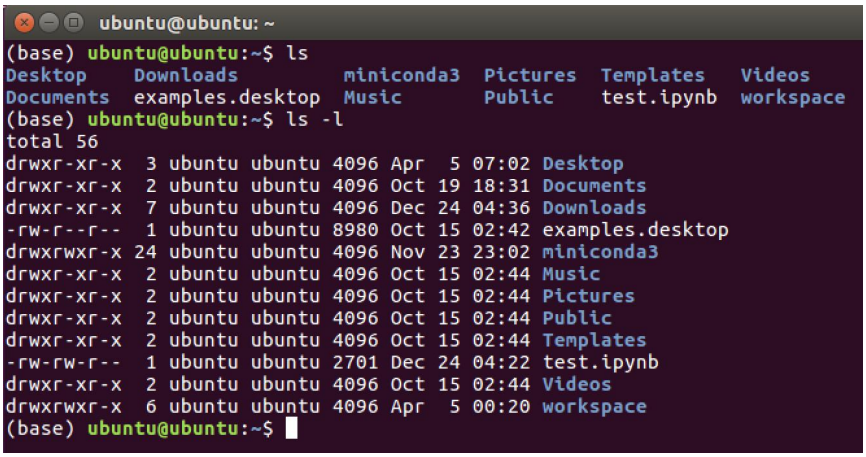
（3）回到自己的主文件夹

（4）显示当前所在工作目录的完整路径

```
(base) ubuntu@ubuntu:~$ cd /usr/local
(base) ubuntu@ubuntu:/usr/local$ cd ..
(base) ubuntu@ubuntu:/usr$ cd ~
(base) ubuntu@ubuntu:~$ pwd
/home/ubuntu
(base) ubuntu@ubuntu:~$
```

- ls

（1）查看目录/usr 下所有的文件




- mkdir

（1）进入/tmp 目录，创建一个名为 a 的目录

（2）创建目录 a1/a2/a3/a4

- rmdir

（1）将上例创建的目录 a（/tmp 下面）删除

（2）删除目录 a1/a2/a3/a4

- touch

（1）在/tmp 下创建一个空文件 hello.txt

- cp

（1）将主文件夹下的.bashrc 复制到/usr 下，命名为 bashrc1

（2）在/tmp 下新建目录 test，再复制这个目录到/usr 目录下

- mv

（1）将上例文件 bashrc1 移动到目录/usr/test

（2）将上例 test 目录重命名为 test2

- rm

（1）将上例复制的 bashrc1 文件删除

（2）将上例的 test2 目录删除

- cat

（1）查看主文件夹下的.bashrc 文件内容

- more

（1）翻页查看主文件夹下.bashrc 文件内容

- man

（1）查看 head 命令的使用方法

- head

（1）查看主文件夹下.bashrc 文件内容前 20 行

tail

（1）查看主文件夹下.bashrc 文件内容最后 20 行

grep

（1）从～/.bashrc 文件中查找字符串'export'

**4. 文件压缩与解压缩**

- tar

（1）在/目录下新建文件夹 test,然后在/目录下将其打包成 test.tar 文件

（2）解压缩到/tmp 目录

**5. 用户与文件权限管理**

- useradd

（1）添加一个名为 student 的用户

- passwd

（1）为 student 用户设置密码 123456

- userdel

（1）删除 student 用户

- chown

（1）在/tmp 下创建一个空文件 hello.txt，并将 hello.txt 文件所有者改为 root 帐号，并查看属性

- chmod

（1）将上例中的文件 hello.txt 设为所有人皆可读取

### 6. 网络配置查看

- ifconfig

（1）查看当前网卡的 IP 地址

### 7. 编辑文件与软件安装

- vim

（1）在主目录下用 vim 编辑 hello.cpp 文件。在文件中输入

```
#include <stdio.h>
int main()
{
  printf("hello Linux!\n");
  return 0;
}
```

保存并退出。

- apt-get

（1）安装 unzip 软件

（2）卸载 unzip 软件

## 六、实验报告要求

请见“实验报告”电子版附件，并按格式要求撰写实验报告并提交。实验报告参考模板如下：


# 实验报告

题目：_____

学 院：信息学院

专 业：

班 级：

学 号：

姓 名：

年　　月　　日

## 一、实验目的（宋体四号加粗）

正文（正文 宋体小四，1.5 倍行距）

## 二、实验环境（宋体四号加粗）

## 三、实验内容（宋体四号加粗）

## 四、实验步骤（图文方式叙述）（宋体四号加粗）

## 五、实验结果及分析（遇到的问题与解决）（宋体四号加粗）

## 六、实验体会（宋体四号加粗）

# 实验二 Hadoop 伪分布模式安装与基本测试

## 一、实验目的

1、了解 Hadoop 的 3 种运行模式

2、熟练掌握 Hadoop 伪分布模式安装流程

3、培养独立完成 Hadoop 伪分布安装的能力

## 二、实验原理

Hadoop 由 Apache 基金会开发的分布式系统基础架构，是利用集群对大量数据进行分布式处理和存储的软件框架。用户可以轻松地在 Hadoop 集群上开发和运行处理海量数据的应用程序。Hadoop 有高可靠，高扩展，高效性，高容错等优点。Hadoop 框架最核心的设计就是 HDFS 和 MapReduce。HDFS 为海量的数据提供了存储，MapReduce 为海量的数据提供了计算。此外，Hadoop 生态还包括了 Hive，Hbase，ZooKeeper，Pig，Avro，Sqoop，Flume，Mahout 等项目。

Hadoop 的运行模式分为 3 种：本地运行模式，伪分布运行模式，完全分布运行模式。

（1）本地模式（local mode）

这种运行模式在一台单机上运行，没有 HDFS 分布式文件系统，而是直接读写本地操作系统中的文件系统。在本地运行模式（local mode）中不存在守护进程，**所有进程都运行在一个 JVM 上**。单机模式适用于开发阶段运行 MapReduce 程序，这也是**最少使用的一个模式**。

（2）伪分布模式

这种运行模式是在**单台服务器**上模拟 Hadoop 的完全分布模式，单机上的分布式并不是真正的分布式，而是**使用线程模拟的分布式**。在这个模式中，所有守护进程(NameNode，DataNode，ResourceManager，NodeManager，SecondaryNameNode)都在同一台机器上运行。因为伪分布运行模式的 Hadoop 集群只有一个节点,所以 HDFS 中的块复制将限制为单个副本,其 secondary-master 和 slave 也都将运行于本地主机。此种模式除了并非真正意义的分布式之外，其

程序执行逻辑完全类似于完全分布式，因此，**常用于开发人员测试程序的执行**。本次实验就是在一台服务器上进行伪分布运行模式的搭建。

（3）完全分布模式

这种模式通常被用于**生产环境**，使用 N 台主机组成一个 Hadoop 集群，Hadoop 守护进程运行在每台主机之上。这里会存在 Namenode 运行的主机，Datanode 运行的主机，以及 SecondaryNameNode 运行的主机。在完全分布式环境下，主节点和从节点会分开。

## 三、实验环境

Linux Ubuntu 16.04

## 四、实验内容

在一台安装 Linux 系统的计算机上，安装 Hadoop 伪分布模式。

## 五、实验步骤

### 1.创建一个专门用户(**此步为可选项**,单机安装不建议使用)

**（注意：如果是真正集群安装，则需要每台机器上有一个相同的账户）**

建议用户创建一个新用户及用户组，后续的操作基本都是在此用户下来操作。但是用户亦可在自己当前非 root 用户下进行操作。 创建一个用户，名为 ubuntu（也可以使用其他名字，比方用 ubuntu 等），并为此用户创建 home 目录，此时会默认创建一个与 ubuntu 同名的用户组。

```
sudo useradd -d /home/ubuntu -m ubuntu
```

为 ubuntu 用户设置密码，执行下面的语句

```
sudo passwd ubuntu
```

按提示消息，输入密码以及确认密码即可，此处密码设置为 ubuntu 将 ubuntu 用户的权限，提升到 sudo 超级用户级别

```
sudo usermod -G sudo ubuntu
```

后续操作，我们需要切换到 ubuntu 用户下来进行操作(**建议注销当前用户后，再以 ubuntu 用户登录，如果用以下命令不是很方便！**)。

```
su - ubuntu
```

**如果上述命令** sudo usermod -G sudo ubuntu **不行（有的系统），则可以采用如下方式**：

需要修改/etc/sudoers 文件:

输入 sudo vim /etc/sudoers 也可以

在文件中的 root ALL=（ALL：ALL）ALL 下添加一样格式的，如下：

ubuntu ALL=（ALL：ALL）ALL

## 2. 首先来配置 SSH 免密码登陆

SSH 免密码登陆需要在服务器执行以下命令，生成公钥和私钥对

```
ssh-keygen -t rsa
```

此时会有多处提醒输入在冒号后输入文本，这里主要是要求输入 ssh 密码以及密码的放置位置。在这里，只需要使用默认值，按回车即可。

```
ubuntu@ubuntu1604:/apps$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ubuntu/.ssh/id_rsa):
Created directory '/home/ubuntu/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ubuntu/.ssh/id_rsa.
```

```
Your public key has been saved in /home/ubuntu/.ssh/id_rsa.pub.
The key fingerprint is:
b3:00:c6:75:86:d6:8b:17:45:c6:7d:a1:74:aa:16:a7 ubuntu@ubuntu1604
The key's randomart image is:
+--[ RSA 2048]----+
|      .oo++.. o. |
|   . .ooo....+.  |
|    +. . o. +.   |
|   . .. o  =     |
|      ..S E      |
|       . +       |
|        .        |
|                 |
|                 |
+-----------------+
ubuntu@ubuntu1604:/apps$
```

此时 ssh 公钥和私钥已经生成完毕,且放置在~/.ssh 目录下。切换到 ~/.ssh 目录下

```
cd ~/.ssh
```

可以看到~/.ssh 目录下的文件

```
ubuntu@ubuntu1604:~/.ssh$ ll
总用量 16
drwx------  2 ubuntu ubuntu 4096 11 月  1 06:37 ./
drwxr-xr-x 51 ubuntu ubuntu 4096 11 月  1 06:37 ../
-rw-------  1 ubuntu ubuntu 1675 11 月  1 06:37 id_rsa
-rw-r--r--  1 ubuntu ubuntu  402 11 月  1 06:37 id_rsa.pub
ubuntu@ubuntu1604:~/.ssh$
```

下面在~/.ssh 目录下，创建一个空文本，名为 authorized_keys

```
touch ~/.ssh/authorized_keys
```

将存储公钥文件的 id_rsa.pub 里的内容，追加到 authorized_keys 中

```
cat  ~/.ssh/id_rsa.pub  >>  ~/.ssh/authorized_keys
```

下面执行 ssh localhost 测试 ssh 配置是否正确

```
ssh localhost
```

第一次使用 ssh 访问，会提醒是否继续连接，输入“yes”继续进行，执行完以后退出。

```
ubuntu@ubuntu1604:~/.ssh$  ssh localhost
The authenticity of host 'localhost (127.0.0.1)' can't be established.
ECDSA key fingerprint is 72:63:26:51:c7:2a:9e:81:24:55:5c:43:b6:7c:14:10.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'localhost' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 14.04.2 LTS (GNU/Linux 3.16.0-23-generic x86_64)
 * Documentation:  https://help.ubuntu.com/
Last login: Tue Nov  1 06:04:05 2016 from 192.168.1.179
ubuntu@ubuntu1604:~$
ubuntu@ubuntu1604:~$ exit
```

注销

```
Connection to localhost closed.
ubuntu@ubuntu1604:~/.ssh$
```

后续再执行 ssh localhost 时，就不用输入密码了。

**注意：**如果 ssh localhost 时提示如下信息，说明 sshd 服务没有。

ssh: connect to host localhost port 22: Connection refused

则需要做如下处理，使用命令安装 sshd server：

```
sudo  apt-get install  openssh-server
```

当出现 Do you want to continue? [Y/n] 时，输入 Y 后回车，来安装此软件。

## 3. 下面首先来创建两个目录，用于存放安装的程序及数据

**（注意：如果数据和安装的程序，放在用户的主目录或其子目录中则不需要改变文件夹的权限和所属用户组）**

```
sudo mkdir /apps

sudo mkdir /data
```

并为/apps 和/data 目录切换所属的用户为 ubuntu 及用户组为 ubuntu （这个需要根据具体的用户来修改用户名）

```
sudo chown -R ubuntu:ubuntu /apps

sudo chown -R ubuntu:ubuntu /data
```

两个目录的作用分别为：/apps 目录用来存放安装的框架，/data 目录用来存放临时数据、HDFS 数据、程序代码或脚本。

切换到根目录下，执行 ls -l 命令

```
cd /

ls -l
```

可以看到根目录下/apps 和/data 目录所属用户及用户组已切换为 ubuntu:ubuntu

```
drwxr-xr-x 171 root    root       4096 11 月  2 01:56 ./

drwxr-xr-x 171 root    root       4096 11 月  2 01:56 ../

drwxr-xr-x   4 ubuntu ubuntu    4096 11 月  1 02:39 apps/
```

```
drwxr-xr-x   2 root   root      4096 11月  1 02:56 bin/
drwxr-xr-x   2 root   root      4096  4月 10  2014 boot/
drwxr-xr-x   2 ubuntu ubuntu    4096 11月  2 01:56 data/
```

## 4. 下载与安装 JDK

创建/data/softwares 目录，用来存放相关安装软件，如 jdk 安装包 jdk-8u201-linux-x64.tar.gz 及 hadoop 安装包 hadoop-2.7.3.tar.gz 等。

```
mkdir -p /data/softwares
```

### 1) 下载或拷贝 JDK 安装包

下载所需的安装包 jdk-8u201-linux-x64.tar.gz。

```
cd /data/softwares

wget
https://download.oracle.com/otn/java/jdk/8u201-b09/42970487e3af4f5aa5bca3f542482c60/jdk-8u201-linux-x64.tar.gz # 这个与具体文件下载地址有关,或直接拷贝下载好的 jdk-8u201-linux-x64.tar.gz 文件到此目录下
```

（注意：也可以将下载好的安装文件，拷贝到这个/data/softwares 目录下。）

例如已经下载到了主目录下的 Downloads 目录下，则可以使用如下命令来拷贝：sudo cp ~/Downloads/*.* /data/softwares/

### 2) 安装 JDK

将/data/software 目录下 jdk-8u201-linux-x64.tar.gz 解压缩到/apps 目录下。

```
tar  -xzvf  /data/softwares/jdk-8u201-linux-x64.tar.gz  -C
/apps/
```

其中，tar -xzvf 对文件进行解压缩，-C 指定解压后，将文件放到/apps 目录下。

切换到/apps 目录下，可以看到目录下内容如下：

```
cd  /apps/

ls  -l
```

下面将 jdk1.8.0_201 目录重命名为 java，执行：

```
mv /apps/jdk1.8.0_201/  /apps/java
```

### 3）为 JDK 配置环境变量

修改主目录下的配置文件.bashrc，添加用户环境变量。

```
sudo  vim  ~/.bashrc
```

输入上面的命令，打开存储环境变量的配置文件。将 java 的环境变量，追加进用户环境变量中。

```
# java

export JAVA_HOME=/apps/java

export PATH=$JAVA_HOME/bin:$PATH
```

输入 Esc，进入 vim 命令模式，输入 :wq 进行保存。

让环境变量生效。

```
source  ~/.bashrc
```

执行 source 命令，让 java 环境变量生效。执行完毕后，可以输入 java，来测试环境变量是否配置正确。如果出现下面界面，则正常运行。

```
java
```

```
ubuntu@ ubuntu1604:~$ java
```

```
用法: java [-options] class [args...]  #注意操作系统是英文则显示为英文
           (执行类)
   或  java [-options] -jar jarfile [args...]
           (执行 jar 文件)
其中选项包括:
    -d32      使用 32 位数据模型 (如果可用)
    -d64      使用 64 位数据模型 (如果可用)
    -server   选择 "server" VM
                  默认 VM 是 server,
                  因为您是在服务器类计算机上运行。
    -cp <目录和 zip/jar 文件的类搜索路径>
    -classpath <目录和 zip/jar 文件的类搜索路径>
                  用 : 分隔的目录, JAR 档案
                  和 ZIP 档案列表, 用于搜索类文件。
    -D<名称>=<值>
    设置系统属性
    -verbose:[class|gc|jni]
    启用详细输出
    -version      输出产品版本并退出
    -version:<值>
    需要指定的版本才能运行
    -showversion  输出产品版本并继续
    -jre-restrict-search | -no-jre-restrict-search
    在版本搜索中包括/排除用户专用 JRE
    -? -help      输出此帮助消息
    -X            输出非标准选项的帮助
    -ea[:<packagename>...|:<classname>]
    -enableassertions[:<packagename>...|:<classname>]
    按指定的粒度启用断言
    -da[:<packagename>...|:<classname>]
    -disableassertions[:<packagename>...|:<classname>]
    禁用具有指定粒度的断言
    -esa | -enablesystemassertions
    启用系统断言
    -dsa | -disablesystemassertions
    禁用系统断言
    -agentlib:<libname>[=<选项>]
    加载本机代理库 <libname>, 例如 -agentlib:hprof
    另请参阅 -agentlib:jdwp=help 和 -agentlib:hprof=help
```

```
    -agentpath:<pathname>[=<选项>]
    按完整路径名加载本机代理库
    -javaagent:<jarpath>[=<选项>]
    加载 Java 编程语言代理, 请参阅 java.lang.instrument
    -splash:<imagepath>
    使用指定的图像显示启动屏幕
    有关详细信息, 请参阅
http://www.oracle.com/technetwork/java/javase/documentation/index.html。
    ubuntu@ ubuntu1604:~$
```

## 5. 下载与安装 Hadoop

### 1) 下载 Hadoop

切换目录到/data/softwares 目录，使用 wget 命令，下载所需的 hadoop 安装包 hadoop-2.7.3.tar.gz 。 **也可以将下载好的安装文件 hadoop-2.7.3.tar.gz，拷贝到这个/data/softwares 目录下。**

```
cd /data/softwares

wget
http://archive.apache.org/dist/hadoop/core/hadoop-2.7.3/ha
doop-2.7.3.tar.gz
```

### 2) 安装 Hadoop

切换到/data/hadoop1 目录下，将 hadoop-2.7.3.tar.gz 解压缩到/apps 目录下。

```
cd /data/softwares

tar -xzvf /data/softwares/hadoop-2.7.3.tar.gz -C /apps/
```

为了便于操作，将/hadoop-2.7.3 重命名为 hadoop。

```
mv /apps//hadoop-2.7.3/  /apps/hadoop
```

### 3）修改用户环境变量

将 hadoop 的路径添加到 path 中。先打开用户环境变量文件。

```
sudo vim ~/.bashrc
```

将以下内容追加到环境变量~/.bashrc 文件中。

```
#hadoop

export HADOOP_HOME=/apps/hadoop

export PATH=$HADOOP_HOME/bin:$PATH
```

让环境变量生效。

```
source ~/.bashrc
```

验证 hadoop 环境变量配置是否正常。

```
hadoop version
```

会看到类似如下内容

```
Hadoop 2.7.3
Subversion ssh://git.corp.linkedin.com:29418/hadoop/hadoop.git -r
e2f1f118e465e787d8567dfa6e2f3b72a0eb9194
Compiled by jhung on 2019-10-22T19:10Z
Compiled with protoc 2.5.0
From source with checksum 7b2d8877c5ce8c9a2cca5c7e81aa4026
This command was run using
/Users/Spark/Cloud/hadoop/share/hadoop/common/hadoop-common-2.7.3.jar
```

## 6.修改 hadoop 相关的配置文件

首先切换到 hadoop 配置目录下。

```
cd /apps/hadoop/etc/hadoop
```

1）编辑 hadoop-env.sh 配置文件

输入 vim /apps/hadoop/etc/hadoop/hadoop-env.sh，打开 hadoop-env.sh 配置文件。

```
vim /apps/hadoop/etc/hadoop/hadoop-env.sh
```

将下面 JAVA_HOME 追加到 hadoop-env.sh 文件中。

```
export JAVA_HOME=/apps/java
```

2）编辑 core-site.xml 文件

输入 vim /apps/hadoop/etc/hadoop/core-site.xml，打开 core-site.xml 配置文件。

```
vim /apps/hadoop/etc/hadoop/core-site.xml
```

添加下面配置到<configuration>与</configuration>标签之间。

```
<property>
    <name>hadoop.tmp.dir</name>
    <value>/data/tmp/hadoop/tmp</value>
</property>
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:9000</value>
</property>
```

这里有两项配置：

一项是 hadoop.tmp.dir，配置 hadoop 处理过程中，临时文件的存储位置。这里的目录/data/tmp/hadoop/tmp 需要提前创建(如下命令)。(注意命令前面一定不要使用 sudo)

```
mkdir -p /data/tmp/hadoop/tmp
```

另一项是 fs.defaultFS，是 hadoop HDFS 文件系统的网络访问地址及端口号。

3）编辑 hdfs-site.xml 文件

输入 vim /apps/hadoop/etc/hadoop/hdfs-site.xml，打开 hdfs-site.xml 配置文件。

```
vim /apps/hadoop/etc/hadoop/hdfs-site.xml
```

添加下面配置到<configuration>与</configuration>标签之间。

```
<property>
    <name>dfs.namenode.name.dir</name>
    <value>/data/tmp/hadoop/hdfs/name</value>
</property>
 <property>
     <name>dfs.datanode.data.dir</name>
     <value>/data/tmp/hadoop/hdfs/data</value>
 </property>
 <property>
     <name>dfs.replication</name>
     <value>1</value>
 </property>
 <property>
     <name>dfs.permissions.enabled</name>
     <value>false</value>
 </property>
```

配置项说明：

dfs.namenode.name.dir，配置元数据信息存储位置；

dfs.datanode.data.dir，配置具体数据存储位置；

dfs.replication，配置每个文件的备份数，由于目前只使用 1 台计算机节点，所以，设置为 1，如果设置为多个的话，运行会报错。

dfs.permissions.enabled，配置 hdfs 是否启用权限认证

另外/data/tmp/hadoop/hdfs 路径，需要提前创建(如下命令)，所以需要执行(注意命令前面一定不要使用 sudo)

```
mkdir -p /data/tmp/hadoop/hdfs
```

**4）编辑 slaves 文件（里面一般已经有内容了，所以可以省略该步）**

输入 vim /apps/hadoop/etc/hadoop/slaves，打开 slaves 配置文件。

```
vim  /apps/hadoop/etc/hadoop/slaves
```

将集群中 slave 角色的节点的主机名，添加进 slaves 文件中。目前只有一台节点，所以 slaves 文件内容为：

```
localhost
```

## 7.格式化 HDFS 文件系统

执行：

```
hadoop  namenode  -format
```

## 8.启动 hadoop 的 hdfs 相关进程

切换目录到/apps/hadoop/sbin 目录下。

```
cd  /apps/hadoop/sbin/
```

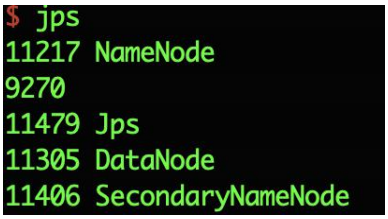
启动 hadoop 的 hdfs 相关进程。

```
./start-dfs.sh
```

这里只会启动 HDFS 相关进程。

输入 jps 查看 HDFS 相关进程是否已经启动。

```
jps
```




我们可以看到相关进程，都已经启动。

## 9.验证 HDFS 运行状态

先在 HDFS 上创建一个目录。

```
hadoop fs -mkdir /myhadoop1
```

执行下面命令，查看目录是否创建成功。

```
hadoop fs -ls -R /
```

```
drwxr-xr-x   - Spark supergroup          0 2021-03-20 22:02 /myhadoop1
```

以上，便是 HDFS 安装过程。

## 10.配置 MapReduce 相关配置

再次切换到 hadoop 配置文件目录

```
cd /apps/hadoop/etc/hadoop
```

### 1）编辑 mapred-site.xml 文件

下面将 mapreduce 的配置文件 mapred-site.xml.template，重命名为 mapred-site.xml。

```
mv /apps/hadoop/etc/hadoop/mapred-site.xml.template
/apps/hadoop/etc/hadoop/mapred-site.xml
```

输入 vim /apps/hadoop/etc/hadoop/mapred-site.xml，打开 mapred-site.xml 配置文件。

```
vim /apps/hadoop/etc/hadoop/mapred-site.xml
```

将 mapreduce 相关配置，添加到<configuration>标签之间。

```
<property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
</property>
```

这里指定 mapreduce 任务处理所使用的框架，为 yarn。

### 2）编辑 yarn-site.xml 文件

输入 vim /apps/hadoop/etc/hadoop/yarn-site.xml，打开 yarn-site.xml 配置文件。

```
vim /apps/hadoop/etc/hadoop/yarn-site.xml
```

将 yarn 相关配置，添加到<configuration>标签之间。

```
<property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
</property>
```

这里的配置是指定所用服务，默认为空。

## 11.启动计算层面相关进程

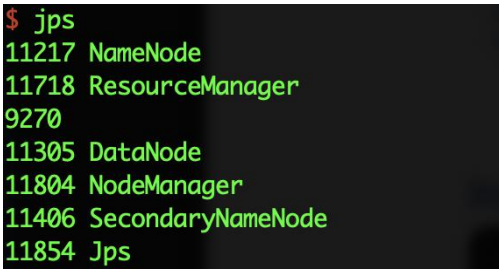
切换到 hadoop 启动目录。

```
cd /apps/hadoop/sbin/
```

执行命令，启动 yarn。

```
./start-yarn.sh
```

输入 jps，查看当前运行的进程。




## 12.执行分布式计算测试

切换到/apps/hadoop/share/hadoop/mapreduce 目录下。

```
cd /apps/hadoop/share/hadoop/mapreduce
```

然后，在该目录下跑一个 mapreduce 程序，来检测一下 hadoop 是否能正常运行。

```
hadoop  jar  hadoop-mapreduce-examples-2.10.0.jar pi  3  3
```

这个程序是计算数学中的 pi 值。当然暂时先不用考虑数据的准确性。当你看到下面流程的时候，表示程序已正常运行，hadoop 环境也是没问题的。

```
17/01/11 04:02:35 INFO mapreduce.Job: Running job: job_1484107180814_0001
17/01/11 04:02:42 INFO mapreduce.Job: Job job_1484107180814_0001 running in uber mode : false
17/01/11 04:02:42 INFO mapreduce.Job:  map 0% reduce 0%
17/01/11 04:02:48 INFO mapreduce.Job:  map 67% reduce 0%
17/01/11 04:02:49 INFO mapreduce.Job:  map 100% reduce 0%
17/01/11 04:02:54 INFO mapreduce.Job:  map 100% reduce 100%
17/01/11 04:02:54 INFO mapreduce.Job: Job job_1484107180814_0001 completed successfully
```

至此，Hadoop 伪分布模式已经安装完成!

## 六、实验报告要求

1.要求完成免密码登陆配置完成后运行

```
ssh localhost
```

命令后的截图。

2.要求完成 JDK 完成后运行

```
cd ~
java -version
```

命令后的截图。

3.要求完成第 5 步“下载与安装 Hadoop”后运行

```
cd ~
hadoop  version
```

命令后的截图。

4.要求完成第 6 步“修改 hadoop 相关的配置文件” 后运行

```
cd ~
cat /apps/hadoop/etc/hadoop/hdfs-site.xml | more
```

命令后的截图。

5.要求完成第 8 步“启动 hadoop 的 hdfs 相关进程”后运行

```
cd /apps/hadoop/sbin/
./start-dfs.sh
jps
```

命令后的截图。

6.要求完成第 10 步“配置 MapReduce 相关配置”后运行

```
cd ~
vim /apps/hadoop/etc/hadoop/mapred-site.xml
```

命令后的截图。

7.要求完成第 11 步“启动计算层面相关进程”后运行

```
cd /apps/hadoop/sbin/
./start-all.sh
```

命令后的截图。

8.要求完成第 12 步“执行分布式计算测试”后的截图。

# 实验三 Hadoop Shell 基本操作

## 一、实验目的

1.熟练掌握常用的 Hadoop shell 命令

## 二、实验原理

使用 HDFS 文件系统的 Shell 命令可使用（也可以参考 PPT 讲义上的内容）

1. hadoop fs [genericOptions] [commandOptions]
2. hadoop dfs [genericOptions] [commandOptions]
3. hdfs dfs [genericOptions] [commandOptions]

三种形式的命令格式，其中：

1. hadoop fs 适用于任何不同的文件系统，比如本地文件系统和 HDFS 文件系统。
2. hadoop dfs 只能适用于 HDFS 文件系统。
3. hdfs dfs 跟 hadoop dfs 的命令作用一样，也只能适用于 HDFS 文件系统。建议使用 hdfs dfs 命令格式。

以上所有 shell 命令在对文件系统进行操作时，可使用 URI 路径作为参数。URI 格式是 scheme://authority/path。

对 HDFS 文件系统，scheme 是 hdfs，对本地文件系统，scheme 是 file。其中 scheme 和 authority 参数都是可选的，如果未加指定，就会使用配置中指定的默认 scheme。一个 HDFS 文件或目录比如/parent/child 可以表示成 hdfs://namenode:namenodeport/parent/child，或者更简单的/parent/child（假设你配置文件中的默认值是 namenode:namenodeport）。大多数 Shell 命令的行为和对应的 Unix Shell 命令类似，出错信息会输出到 stderr，其他信息输出到 stdout。

## 三、实验环境

Linux ubuntu 16.04

hadoop-2.7.3

## 四、实验内容

1. 学习在开启、关闭 Hadoop

2. 学习在 Hadoop 中创建、修改、查看、删除文件夹及文件

3. 学习改变文件的权限及文件的拥有者

4. 学习使用 shell 命令提交 job 任务

5. Hadoop 安全模式的进入与退出

附：在**安全模式**下，不能进行修改文件系统的操作，但可以浏览目录结构、查看文件内容。

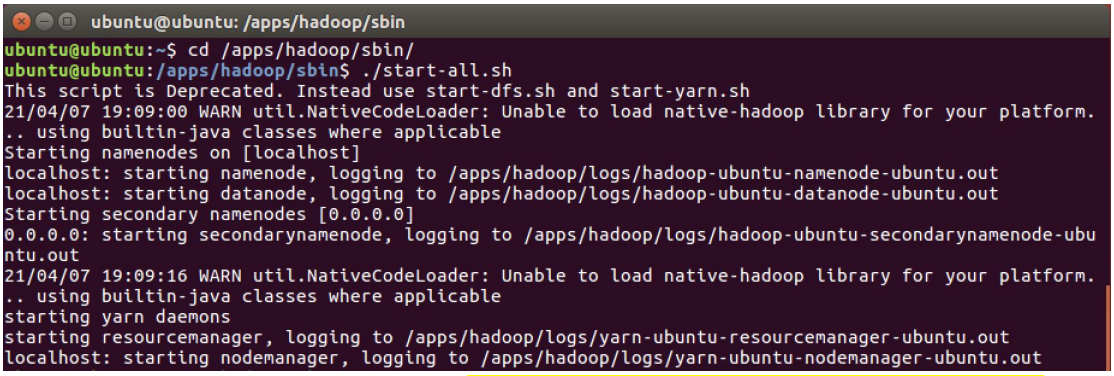
## 五、实验步骤

1. 启动 Hadoop 系统

打开终端模拟器，切换到 Hadoop 的安装目录下 sbin 目录中，然后使用 start-all.s 命令启动 Hadoop 系统的 HDFS 和 Yarn。例如在/apps/hadoop/sbin 目录下，启动 Hadoop

```
cd  /apps/hadoop/sbin
./start-all.sh
```

输入大致如下：




除了直接执行 start-all.sh 外，还可以分步启动 HDFS 用 start-dfs.sh 和启动 Yarn 用 start-yarn.sh。

2. 执行 jps，检查一下 Hadoop 相关进程是否启动

```
jps
```

应该能看到 ResurceManager、DataNode、NameNode、SecondaryNameNode 和 NodeManager。

```
ubuntu@ubuntu:/apps/hadoop/sbin$ jps
3936 Jps
3156 DataNode
3783 NodeManager
3337 SecondaryNameNode
3006 NameNode
3486 ResourceManager
ubuntu@ubuntu:/apps/hadoop/sbin$
```

3. 在/目录下创建一个 test1 文件夹

```
hadoop  fs  -mkdir  /test01
```

或

```
hdfs  dfs  -mkdir  /test01
```

4. 在 Hadoop 中的 test1 文件夹中创建一个 file.txt 文件

```
hadoop  fs  -touchz  /test01/file.txt
```

或

```
hdfs  dfs  -touchz  /test01/file.txt
```

注意：后面的命令为了简化只写 hadoop fs 命令，同理也可以换为 hdfs dfs 命令。

5. 查看根目录下所有文件

```
hadoop  fs  -ls  /
```

```
ubuntu@ubuntu:/apps/hadoop/sbin$ hadoop fs -ls  /
21/04/07 19:14:43 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Found 1 items
drwxr-xr-x   - ubuntu supergroup          0 2020-10-26 05:17 /test01
ubuntu@ubuntu:/apps/hadoop/sbin$
```

6. 还可以使用 ls -R 的方式递归查看根下所有文件

```
hadoop  fs  -ls  -R  /
```

```
ubuntu@ubuntu: /apps/hadoop/sbin
ubuntu@ubuntu:/apps/hadoop/sbin$ hadoop fs -ls -R /
21/04/07 19:15:38 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
drwxr-xr-x   - ubuntu supergroup          0 2020-10-26 05:17 /test01
-rw-r--r--   1 ubuntu supergroup          0 2020-10-26 05:17 /test01/file.txt
ubuntu@ubuntu:/apps/hadoop/sbin$
```

7. 将 Hadoop 根下 test01 目录中的 file.txt 文件，移动到根下并重命名为 file2.txt

```
hadoop  fs  -mv /test01/file.txt   /file2.txt
```

Hadoop 中的 mv 用法同 Linux 中的一样，都可以起到移动文件和重命名的作用。

8. 将 Hadoop 根下的 file2.txt 文件复制到 test1 目录下

```
hadoop  fs  -cp  /file2.txt /test01
```

9. 在 Linux 本地/data 目录下，创建一个 data.txt 文件，并向其中写入 hello hadoop!

```
cd  /data
touch  data.txt
echo  hello  hadoop!  >>  data.txt
```

10. 将 Linux 本地/data 目录下的 data.txt 文件，上传到 HDFS 中的/test01 目录下

```
hadoop  fs  -put  /data/data.txt  /test01
```

11. 查看 Hadoop 中/test01 目录下的 data.txt 文件

```
hadoop  fs  -cat  /test01/data.txt
```

```
ubuntu@ubuntu: /data
ubuntu@ubuntu:/data$ hdfs dfs -ls /
21/04/07 19:22:02 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... us
ing builtin-java classes where applicable
Found 1 items
drwxr-xr-x   - ubuntu supergroup          0 2021-04-07 19:21 /test01
ubuntu@ubuntu:/data$ hdfs dfs -cat  /test01/data.txt
21/04/07 19:22:09 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... us
ing builtin-java classes where applicable
hello hadoop!
ubuntu@ubuntu:/data$
```

12. 除此之外还可以使用 tail 方法

```
hadoop  fs  -tail  /test1/data.txt
```

```
ubuntu@ubuntu:/data$ hdfs dfs -tail  /test01/data.txt
21/04/07 19:22:35 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... us
ing builtin-java classes where applicable
hello hadoop!
ubuntu@ubuntu:/data$
```

tail 方法是将文件尾部 1K 字节的内容输出。支持-f 选项，行为和 Unix 中一致。

13. 查看 Hadoop 中/test1 目录下的 data.txt 文件大小

```
hadoop  fs  -du  -s  /test01/data.txt
```

```
ubuntu@ubuntu:/data$ hadoop  fs  -du  -s  /test01/data.txt
21/04/07 19:23:23 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... us
ing builtin-java classes where applicable
14  14  /test01/data.txt
ubuntu@ubuntu:/data$
```

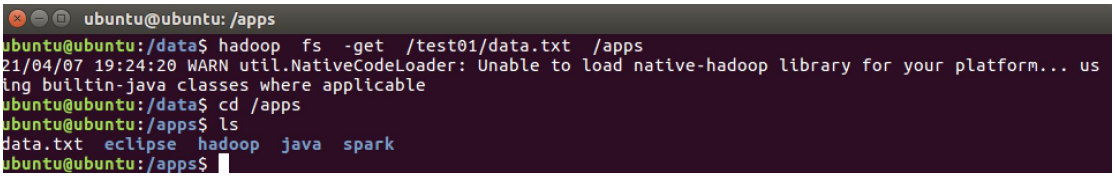
-du 后面可以不加-s，直接写目录表示查看该目录下所有文件大小

14. 将 Hadoop 中/test1 目录下的 data.txt 文件，下载到 Linux 本地/apps 目录中

```
hadoop  fs  -get  /test01/data.txt  /apps
```

15. 查看一下/apps 目录下是否存在 data.txt 文件

```
ls  /apps
```




16. 使用 chown 方法，改变 Hadoop 中/test1 目录中的 data.txt 文件拥有者为 root，使用-R 将使改变在目录结构下递归进行。

```
hadoop  fs  -chown  root  /test01/data.txt
```

17. 使用 chmod 方法，赋予 Hadoop 中/test01 目录中的 data.txt 文件 777 权限

```
hadoop  fs  -chmod 777  /test01/data.txt
```

18. 删除 Hadoop 根下的 file2.txt 文件

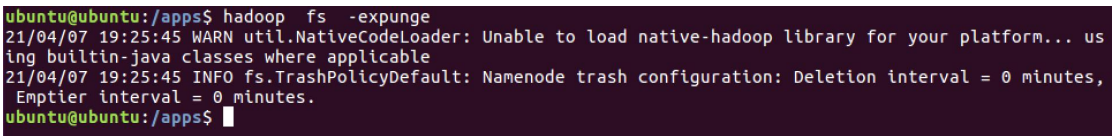
```
hadoop  fs  -rm  /file2.txt
```

19. 删除 Hadoop 根下的 test01 目录

```
hadoop  fs  -rm  -r  /test01
```

20. 当在 Hadoop 中设置了回收站功能时，删除的文件会保留在回收站中，可以使用 expunge 方法清空回收站。

```
hadoop  fs  -expunge
```




在分布式文件系统启动的时候，开始的时候会有安全模式，当分布式文件系统处于安全模式的情况下，文件系统中的内容不允许修改也不允许删除，直到安全模式结束。安全模式主要是为了系统启动的时候检查各个 DataNode 上数据块的有效性，同时根据策略必要的复制或者删除部分数据块。运行期通过命令也可

以进入安全模式。在实践过程中，系统启动的时候去修改和删除文件也会有安全模式不允许修改的出错提示，只需要等待一会儿即可。

21.使用 Shell 命令执行 Hadoop 自带的 WordCount

首先切换到/data 目录下，使用 vim 编辑一个 data.txt 文件，内容为: hello world hello hadoop hello ipieuvre

```
cd  /data
vim  data.txt
```

在 HDFS 的根下创建 in 目录，并将/data 下的 data.txt 文件上传到 HDFS 中的 in 目录

```
hdfs  dfs  -mkdir /in
hadoop  fs  -put  /data/data.txt  /in
```

执行 hadoop jar 命令(注意这里只能是 hadoop 命令)，在 hadoop 的 /apps/hadoop/share/hadoop/mapreduce 路径下存在 hadoop-mapreduce-examples-2.6.0-cdh5.4.5.jar(注意这个文件名与具体安装的 hadoop 版本有关)，执行其中的 worldcount 类，数据来源为 HDFS 的/in 目录，数据输出到 HDFS 的/out 目录

```
hadoop  jar
/apps/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-exampl
es-2.6.0-cdh5.4.5.jar  wordcount  /in  /out
```

查看 HDFS 中的/out 目录

```
Hadoop  fs  -ls  /out
hadoop  fs  -cat  /out/*
```

```
zhangyu@e2adddc16fd8:/data$ hadoop fs -ls /out

Found 2 items
-rw-r--r--   1 zhangyu supergroup          0 2017-08-11 06:49 /out/_SUCCESS
-rw-r--r--   1 zhangyu supergroup         36 2017-08-11 06:49 /out/part-r-00000
zhangyu@e2adddc16fd8:/data$ hadoop fs -cat /out/*
hadoop  1
hello   3
ipieuvre        1
world   1
zhangyu@e2adddc16fd8:/data$
```

22.进入 Hadoop 安全模式

```
hdfs  dfsadmin  -safemode  enter
```

```
zhangyu@a81ff3854876:/data$ hdfs dfsadmin -safemode enter
Safe mode is ON
```

23. 退出 hadoop 安全模式

```
hdfs  dfsadmin  -safemode  leave
```

```
zhangyu@a81ff3854876:/data$ hdfs dfsadmin -safemode leave
Safe mode is OFF
```

24. 切换到/apps/hadoop/sbin 目录下，关闭 Hadoop

```
cd /apps/hadoop/sbin
./stop-all.sh
```

```
zhangyu@a81ff3854876:/data$ cd /apps/hadoop/sbin
zhangyu@a81ff3854876:/apps/hadoop/sbin$ ./stop-all.sh
This script is Deprecated. Instead use stop-dfs.sh and stop-yarn.sh
Stopping namenodes on [0.0.0.0]
0.0.0.0: stopping namenode
localhost: stopping datanode
Stopping secondary namenodes [0.0.0.0]
0.0.0.0: stopping secondarynamenode
stopping yarn daemons
stopping resourcemanager
localhost: stopping nodemanager
no proxyserver to stop
```

25. 将讲义 PPT 中的第 3.6 节的在 HDFS 系统上创建文件夹 hdfstest 的例子代码，在 Eclipse 环境中调试运行。（选做）

## 六、实验报告要求

根据实验后，要求完成如下内容，完成每步的截图。

（1） 在本地～/Download 文件夹下创建文件 hdfs03.txt。

（2） 在 hdfs03.txt 文件里面输入内容 hello,hdfs!。

（3） 在 hdfs 分布式文件系统中创建文件夹/hdfs/test03。

（4） 将 hdfs03.txt 文件上传到 hdfs 分布式文件系统的文件夹/hdfs/test03 中。

（5） 使用命令查看/hdfs/test03 文件夹中的内容。

（6） 使用命令查看/hdfs/test03/hdfs03.txt 文件中的内容。

（7） 将（4）步中上传的文件下载到本地文件夹～/Documents 中，并利用 ls 命令查看是否下载成功。

（8） 使用命令删除文件夹/hdfs/test03。

（9） 使用命令查看删除后文件夹/hdfs 中的内容。

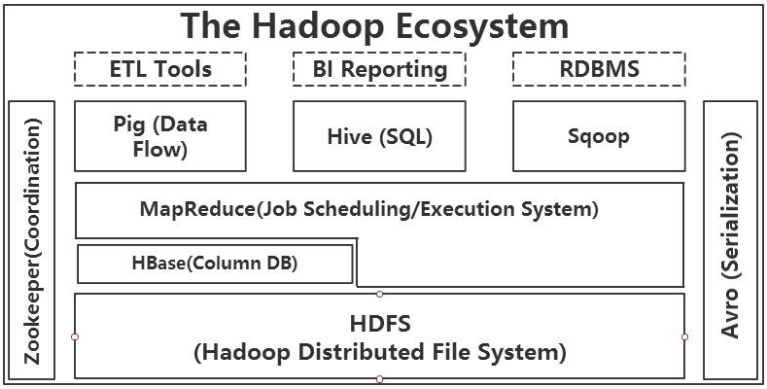
# 实验四 HBase 安装与基本操作

## 一、实验目的

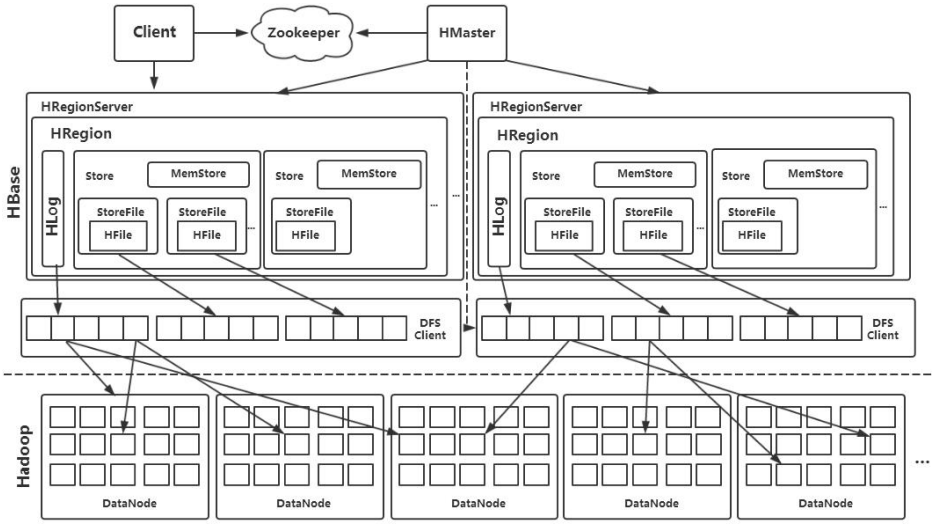
1. 了解 HBase 的基本工作原理与安装流程
2. 学习 HBase 的基本操作 shell 命令

## 二、实验原理

HBase 是一个分布式的，面向列的开源数据库，该技术来源于 Fay Chang 所撰写的 Google 论文”Bigtable：一个结构化数据的分布式存储系统“。HBase 不同于一般的关系数据库，它是一个适合于非结构化数据存储的数据库。另一个不同的是 HBase 基于列而不是基于行模式。在需要实时读写、随机访问超大规模数据集时，可以使用 HBase。




上图描述 Hadoop Ecosystem（生态系统）中的各层结构。其中，HBase 位于 HDFS 文件存储层之上，Hadoop HDFS 为 HBase 提供了高可靠性的底层存储支持。Hadoop MapReduce 为 HBase 提供了高性能的计算能力，ZooKeeper 为 HBase 提供了稳定服务和故障切换(failover)机制。此外，Pig 和 Hive 还为 HBase 提供了高层语言（类似 SQL）支持，使得在 HBase 上进行数据统计处理变的非常简单。Sqoop 则为 HBase 提供了方便的 RDBMS（关系数据库管理系统）数据导入功能，使得传统数据库数据向 HBase 中迁移变的非常方便。




HBase 架构图

HBase Master 负责管理所有的 HRegion，HBase Master 并不存储 HBase 服务器的任何数据，HBase 逻辑上的表可能会划分为多个 HRegion，然后存储在 HRegion Server 群中,HBase Master Server 中存储的是从数据到 HRegion Server 的映射。

一台机器只能运行一个 HRegion 服务器，数据的操作会记录在 Hlog 中，在读取数据时候，HRegion 会先访问缓存（MemStore），如果缓存中没有数据才回到 HStore 中上找，每一个列都会有一个 HStore 集合，每个 HStore 集合包含了很多具体的 StoreFile 文件（StoreFile 底层是以 HFile 的格式文件存储的），这些文件是 B 树结构的，方便快速读取。

## 三、实验环境

Linux Ubuntu 16.04
jdk-7u75-linux-x64
hadoop-2.6.0-cdh5.4.5

## 四、实验内容

在已安装好的 Hadoop 环境基础上，安装并配置 HBase。

## 五、实验步骤

1. 首先在 Linux 本地，新建/data/hbase1 目录，用于存放所需文件。

```
mkdir  -p  /data/hbase1
```

切换目录到/data/hbase1 下，使用 wget 命令，下载 HBase 所需安装包 hbase-1.0.0-cdh5.4.5.tar.gz。（或将已经下载到安装包直接拷贝到该目录下）

```
cd  /data/hbase1
wget
http://192.168.1.100:60000/allfiles/hbase1/hbase-1.0.0-cdh5.4.5.tar.gz
```

2.将/data/hbase1 目录下,HBase 的安装包 hbase-1.0.0-cdh5.4.5.tar.gz，解压缩到/apps 目录下。

```
tar  -xzvf  /data/hbase1/hbase-1.0.0-cdh5.4.5.tar.gz  -C
/apps
```

再切换到/apps 目录下,将/apps/hbase-1.0.0-cdh5.4.5/,重命名为 hbase。

```
cd /apps
mv  /apps/hbase-1.0.0-cdh5.4.5/  /apps/hbase
```

3.添加 HBase 的环境变量。首先使用 vim 打开用户环境变量文件。

```
sudo  vim  ~/.bashrc
```

在环境变量文件末尾位置，追加 HBase 的 bin 目录路径相关配置，并保存退出。即下列内容：

```
#hbase
export HBASE_HOME=/apps/hbase
export PATH=$HBASE_HOME/bin:$PATH
```

执行 source（或直接 bash 命令）命令，使环境变量生效。

```
source  ~/.bashrc
```

此时就可以调用 HBase 的 bin 目录下的脚本了。先来查看一下 HBase 的版本信息。

```
hbase  version
```

4.下面开始配置 HBase。切换目录到/apps/hbase/conf 目录下，并使用 vim 编辑 hbase-env.sh 文件。

```
cd  /apps/hbase/conf
vim  hbase-env.sh
```

追加配置内容到 hbase-env.sh 中，并保存退出。

```
export  JAVA_HOME=/apps/java
export  HBASE_MANAGES_ZK=true
export  HBASE_CLASSPATH=/apps/hbase/conf
```

很明显：

JAVA_HOME 为 java 程序所在位置；

HBASE_MANAGES_ZK 表示是否使用 HBase 自带的 zookeeper 环境；

HBASE_CLASSPATH 指向 hbase 配置文件的路径。

5.下面使用 vim 打开 hbase-site.xml 文件。

```
vim  hbase-site.xml
```

在两个<configuration>之间添加如下内容，并保存退出。

```
<property>
    <name>hbase.master</name>
    <value>localhost</value>
</property>
<property>
    <name>hbase.rootdir</name>
    <value>hdfs://localhost:9000/hbase</value>
</property>
<property>
    <name>hbase.cluster.distributed</name>
    <value>true</value>
</property>
<property>
    <name>hbase.zookeeper.quorum</name>
    <value>localhost</value>
</property>
<property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/data/tmp/zookeeper-hbase</value>
</property>
```

配置项说明：

hbase.master：HBase 主节点地址。

hbase.rootdir：HBase 文件所存储的位置。

hbase.cluster.distributed：HBase 是否为分布式模式。

hbase.zookeeper.quorum：这里是配置 ZooKeeper 的服务器的地方。

hbase.zookeeper.property.dataDir：这里表示 HBase 在 ZooKeeper 上存储数据的位置。

注意：这里 hbase.zookeeper.property.dataDir 目录，需要提前创建。

```
sudo  mkdir  -p  /data/tmp/zookeeper-hbase
```

将/data/tmp/zookeeper-hbase 目录切换所属用户为 ubuntu 及用户组为 ubuntu。

```
sudo  chown  -R  ubuntu:ubuntu  /data/tmp/zookeeper-hbase
```

6.(改步默认已经有内容了，可以不进行配置，可以打开改文件核实一下是否已经有内容了)使用 vim 编辑/apps/hbase/conf/regionservers 文件，此文件存储了 HBase 集群节点的 ip 地址，目前只有一台节点，所以只需要填写 localhost 即可。

```
    vim /apps/hbase/conf/regionservers
```

修改文件内容为：

```
    localhost
```

7.下面输入 jps，查看当前进程，Hadoop 进程是否已经启动。

```
    jps
```

若未启动，则切换到/apps/hadoop/sbin 目录下，启动 Hadoop。

```
    cd  /apps/hadoop/sbin
    ./start-all.sh
```

当 Hadoop 相关进程启动后，进入 HBase 的 bin 目录下，启动 HBase 服务。

```
    cd  /apps/hbase/bin/
```

```
./start-hbase.sh
```

8.输入 jps，查看 HBase 相关进程是否存在。

```
jps
```

输出结果为：

```
zhangyu@3e23b66754d6:/apps/hbase/bin$ jps
3556 HMaster
3472 HQuorumPeer
3705 HRegionServer
4086 Jps
705 SecondaryNameNode
407 NameNode
536 DataNode
878 ResourceManager
981 NodeManager
```

可以看到 HMaster、HRegionServer、HQuorumPeer 进程都已启动。

为了进一步测试 HBase 安装，是否正常，进入 HBase Shell 接口。

```
hbase  shell
```

（注意：如果我们使用 SecureCRT 这样的远程访问工具，输错命令时，直接按 Backspace 键，是不能删除掉前面的文字的。在这里可以使用 Ctrl 键+Backspace 键来删除前面输错的文字。）

输入 list 的命令，查看当前有哪些 HTable 表。

```
list
```

创建一张表 tb，表中含有一个列簇 mycf。

```
create 'tb','mycf'
```

再次输入 list，列出 HBase 中的表。

```
hbase(main):010:0> create 'tb','mycf'
0 row(s) in 0.4440 seconds

=> Hbase::Table - tb
hbase(main):011:0> list
TABLE
tb
1 row(s) in 0.0100 seconds

=> ["tb"]
hbase(main):012:0>
```

到此，hbase 的安装测试都已完毕！

==========

以下为 HBase 基本操作部分

9.用 create 命令创建一张表，表的参数如下：

表名为 table_name,列族名为 f1(语法:create <table>, {NAME => <family>, VERSIONS => <VERSIONS>})

```
create  'table_name','f1'
```

也可以指定数据保存的版本数，如：create 'table_name2',{NAME => 'f1', VERSIONS => 2}

10.使用 exists 命令查看 table_name 表是否存在

```
exists  'table_name'
```

```
hbase(main):004:0> exists 'table_name'
Table table_name does exist
0 row(s) in 0.0130 seconds
```

11.使用 desc 命令来查看一下 table_name 表结构(语法:describe <table>)

```
desc  'table_name'
```

```
hbase(main):016:0> desc 'table_name'
Table table_name is ENABLED
table_name
COLUMN FAMILIES DESCRIPTION
{NAME => 'f1', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSI
ON => 'NONE', MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '65536', IN_MEMORY => '
false', BLOCKCACHE => 'true'}
1 row(s) in 0.0320 seconds

hbase(main):017:0>
```

12.修改 table_name 的表结构，将 TTL（生存周期，TTL 的存活时间是针对于每一个版本来说的，因为默认每一个版本都是以当前的时间戳作为版本号，也就是说当某个版本插入时，从它的版本号开始算起，在 TTL 的过期时间到达时，这个版本就会被删除。）改为 30 天，这里要注意，修改表结构前必须先 disable 使表失效，修改完成后再使用 enable 命令，使表重新生效（可用 is_enabled 'table_name'或 is_disabled 'table_name'判断表的状态）

```
disable  'table_name'
alter  'table_name',{NAME=>'f1',TTL=>'2592000'}
enable  'table_name'
```

这里 2592000 为 30 天的秒数，再次使用 desc 命令会发现表的 TTL 已经改为了 2592000

```
hbase(main):031:0> desc 'table_name'
Table table_name is ENABLED
table_name
COLUMN FAMILIES DESCRIPTION
{NAME => 'f1', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSI
ON => 'NONE', MIN_VERSIONS => '0', TTL => '2592000 SECONDS (30 DAYS)', KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '6553
6', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
1 row(s) in 0.0220 seconds

hbase(main):032:0>
```

13.现在使用 put 命令向 table_name 表中插入一行数据

（语法:put  <table>,<rowkey>,<family:column>,<value>,<timestamp>）

```
put  'table_name','rowkey001','f1:col1','value1'
put  'table_name','rowkey001','f1:col2','value2'
put  'table_name','rowkey002','f1:col1','value1'
```

```
hbase(main):012:0> put 'table_name','rowkey001','f1:col1','value1'
0 row(s) in 0.1180 seconds

hbase(main):013:0> put 'table_name','rowkey001','f1:col2','value2'
0 row(s) in 0.0080 seconds

hbase(main):014:0> put 'table_name','rowkey002','f1:col1','value1'
0 row(s) in 0.0090 seconds

hbase(main):015:0>
```

这其中，'table_name'为表名，'rowkey001'为 rowkey，'f1:col1' f1 为列族，col1 为列，'value1'为值，同一个列族下可以有多个列，同一个 rowkey 视为同一行。

14.使用 get 命令来查询一下 table_name 表，rowkey001 中的 f1 下的 col1 的值

（语法：get <table>,<rowkey>,[<family:column>,....]）

```
get 'table_name','rowkey001', 'f1:col1'
```

另一种用法：

```
get 'table_name','rowkey001', {COLUMN=>'f1:col1'}
```

```
hbase(main):017:0> get 'table_name','rowkey001', 'f1:col1'
COLUMN                          CELL
 f1:col1                        timestamp=1506045223485, value=value1
1 row(s) in 0.0240 seconds

hbase(main):018:0> get 'table_name','rowkey001', {COLUMN=>'f1:col1'}
COLUMN                          CELL
 f1:col1                        timestamp=1506045223485, value=value1
1 row(s) in 0.0120 seconds

hbase(main):019:0>
```

15.查询表 table_name，rowkey001 中的 f1 下的所有列值

```
get 'table_name','rowkey001'
```

```
hbase(main):019:0> get 'table_name','rowkey001'
COLUMN                          CELL
 f1:col1                        timestamp=1506045223485, value=value1
 f1:col2                        timestamp=1506045223539, value=value2
2 row(s) in 0.0140 seconds

hbase(main):020:0>
```

16.使用 scan 命令扫描全表（语法：scan <table>, {COLUMNS => [ <family:column>,.... ], LIMIT => num}）

```
scan 'table_name'
```

也可以限定扫描表的前几行数据，例如扫描前 1 行数据：

```
scan 'table_name',{LIMIT=>1}
```

```
hbase(main):023:0> scan 'table_name'
ROW                             COLUMN+CELL
 rowkey001                      column=f1:col1, timestamp=1506045223485, value=value1
 rowkey001                      column=f1:col2, timestamp=1506045223539, value=value2
 rowkey002                      column=f1:col1, timestamp=1506045224188, value=value1
2 row(s) in 0.0110 seconds

hbase(main):024:0> scan 'table_name',{LIMIT=>1}
ROW                             COLUMN+CELL
 rowkey001                      column=f1:col1, timestamp=1506045223485, value=value1
 rowkey001                      column=f1:col2, timestamp=1506045223539, value=value2
1 row(s) in 0.0160 seconds

hbase(main):025:0>
```

由此也可以看出，rowkey 相同的数据视为一行数据。

17.使用 count 命令，查看 table_name 表中的数据行数

（语法：count <table>, {INTERVAL => intervalNum, CACHE => cacheNum}）

INTERVAL 设置多少行显示一次及对应的 rowkey，默认 1000；CACHE 每次去取的缓存区大小，默认是 10，调整该参数可提高查询速度

查询表 table_name 中的数据行数，每 10 条显示一次，缓存区为 200

```
count 'table_name', {INTERVAL => 10, CACHE => 200}
```

```
hbase(main):025:0> count 'table_name', {INTERVAL => 10, CACHE => 200}
2 row(s) in 0.0310 seconds

=> 2
hbase(main):026:0>
```

由于 table_name 表的数据只有 2 行，所以查询结果为 2。

18. 使用 delete 命令删除 table_name 表中，rowkey001 中的 f1：col2 的数据

（语法：delete <table>, <rowkey>, <family:column> , <timestamp>, 必须指定列名）

```
delete 'table_name','rowkey001','f1:col2'
```

这里需要注意，如果该列保存有多个版本的数据，将一并被删除

19. 使用 deleteall 命令，删除 table_name 表中 rowkey002 这行数据

（语法：deleteall <table>, <rowkey>, <family:column> , <timestamp>, 可以不指定列名，删除整行数据）

```
deleteall 'table_name','rowkey002'
```

20. 使用 truncate 命令，删除 table_name 表中的所有数据

（语法： truncate <table> 其具体过程是：disable table -> drop table -> create table）

```
truncate 'table_name'
```

```
hbase(main):042:0> truncate 'table_name'
Truncating 'table_name' table (it may take a while):
 - Disabling table...
 - Truncating table...
0 row(s) in 1.6730 seconds

hbase(main):043:0>
```

## 六、实验报告要求

要求完成如下内容，完成每步的截图。

（1） HBase 安装完成后，依次启动 Hadoop 系统和 HBase 系统。

（2） 用 jps 命令查看系统启动后的 java 进程，看系统是否启动成功。

（3） 进入到 HBase 的 shell 命令环境，并用 list 命令查看 HBase 数据中已有的表。

（4） 用 create 命令创建一个学生信息表（表名为 student）。

用来存储学生姓名（姓名作为行键，并且假设姓名不会重复）以及考试成绩，其中，考试成绩 score 是一个列族，分别存储了各个科目的考试成绩（为列限定符）。如下所示：

student 表结构

| name | score | | |
|---|---|---|---|
| | english | math | computer |

（5） 用 put 命令向 student 表中添加的两条数据。

两条数据的内容如下：

| | | | |
|---|---|---|---|
| zhangsan | 69 | 86 | 77 |
| lisi | 55 | 100 | 88 |

（6） 用 get 命令查看表中 zhangsan 的所有分数。
（7） 用 scan 命令查看 student 表中所有数据。
（8） 依次关闭 HBase 系统和 Hadoop 系统，并用 jps 查看是否成功。

# 实验五 Mapreduce 实例——统计商品的销售数

（类似 WordCount 例子）

## 一、实验目的

1. 准确理解 Mapreduce 的设计原理
2. 熟练掌握类 WordCount 程序代码编写
3. 学会自己编写类 WordCount 程序进行基本的统计

## 二、实验原理

MapReduce 采用的是“分而治之”的思想，把对大规模数据集的操作，分发给一个主节点管理下的各个从节点共同完成，然后通过整合各个节点的中间结果，得到最终结果。简单来说，MapReduce 就是“任务的分解与结果的汇总”。

### 1. MapReduce 的工作原理

在分布式计算中，MapReduce 框架负责处理了并行编程里分布式存储、工作调度，负载均衡、容错处理以及网络通信等复杂问题，现在我们把处理过程高度抽象为 Map 与 Reduce 两个部分来进行阐述，其中 Map 部分负责把任务分解成多个子任务，Reduce 部分负责把分解后多个子任务的处理结果汇总起来，具体设计思路如下。

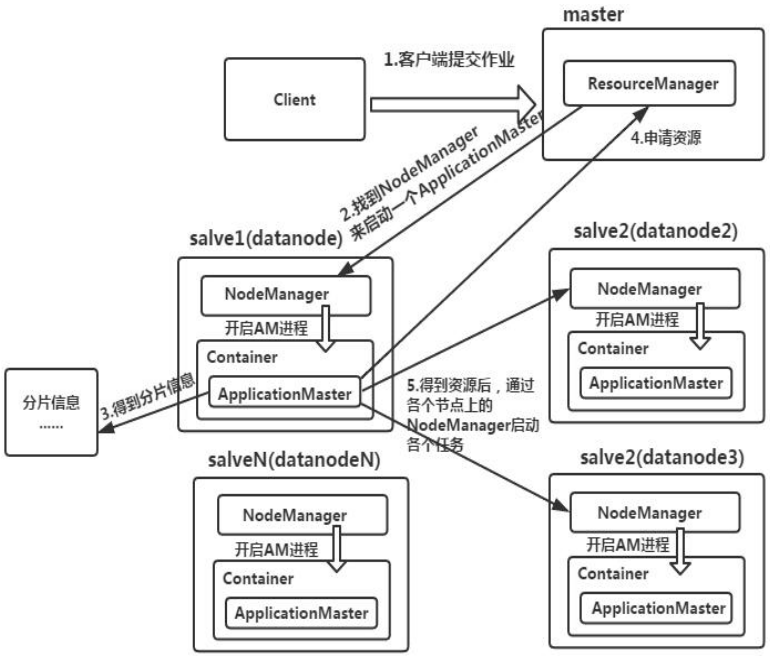
（1）Map 过程需要继承 org.apache.hadoop.mapreduce 包中 Mapper 类，并重写其 map 方法。通过在 map 方法中添加把 key 值和 value 值输出到控制台的代码，可以发现 map 方法中输入的 value 值存储的是文本文件中的一行（以回车符为行结束标记），而输入的 key 值存储的是该行的首字母相对于文本文件的首地址的偏移量。然后用 StringTokenizer 类将每一行拆分成为一个个的字段，把截取出需要的字段（本实验为买家 id 字段）设置为 key，将 value 设置为 1，并将其作为 map 方法的结果输出。

（2）Reduce 过程需要继承 org.apache.hadoop.mapreduce 包中 Reducer 类，并重写其 reduce 方法。Map 过程输出的<key,value>键值对先经过 shuffle 过程把 key 值相同的所有 value 值聚集起来形成 values，此时 values 是对应 key 字段的计数值所组成的列表，然后将<key,values>输入到 reduce 方法中，reduce 方法只要遍历 values 并求和，即可得到某个单词的总次数。

在 main()主函数中新建一个 Job 对象，由 Job 对象负责管理和运行 MapReduce 的一个计算任务，并通过 Job 的一些方法对任务的参数进行相关的设置。

本实验是设置使用将继承 Mapper 的 doMapper 类完成 Map 过程中的处理和使用 doReducer 类完成 Reduce 过程中的处理。还设置了 Map 过程和 Reduce 过程的输出类型：key 的类型为 Text，value 的类型为 IntWritable。任务的输出和输入路径则由字符串指定，并由 FileInputFormat 和 FileOutputFormat 分别设定。完成相应任务的参数设定后，即可调用 job.waitForCompletion()方法执行任务，其余的工作都交由 MapReduce 框架处理。

2.MapReduce 框架的作业运行流程




（1）ResourceManager：是 YARN 资源控制框架的中心模块，负责集群中所有资源的统一管理和分配。它接收来自 NM(NodeManager)的汇报，建立 AM，并将资源派送给 AM(ApplicationMaster)。

（2）NodeManager：简称 NM，NodeManager 是 ResourceManager 在每台机器上的代理，负责容器管理，并监控他们的资源使用情况（cpu、内存、磁盘及网络等），以及向 ResourceManager 提供这些资源使用报告。

（3）ApplicationMaster：以下简称 AM。YARN 中每个应用都会启动一个 AM，负责向 RM 申请资源，请求 NM 启动 Container，并告诉 Container 做什么事情。

（4）Container：资源容器。YARN 中所有的应用都是在 Container 之上运行的。AM 也是在 Container 上运行的，不过 AM 的 Container 是 RM 申请的。Container 是 YARN 中资源的抽象，它封装了某个节点上一定量的资源（CPU 和内存两类资源）。Container 由 ApplicationMaster 向 ResourceManager 申请的，由 ResouceManager 中的资源调度器异步分配给 ApplicationMaster。Container 的运行是由 ApplicationMaster 向资源所在的 NodeManager 发起的，Container 运行时需提供内部执行的任务命令（可以是任何命令，比如 java、Python、C++进程启动命令均可）以及该命令执行所需的环境变量和外部资源（比如词典文件、可执行文件、jar 包等）。

另外，一个应用程序所需的 Container 分为两大类，如下：

①运行 ApplicationMaster 的 Container：这是由 ResourceManager（向内部的资源调度器）申请和启动的，用户提交应用程序时，可指定唯一的 ApplicationMaster 所需的资源。

②运行各类任务的 Container：这是由 ApplicationMaster 向 ResourceManager 申请的，并为了 ApplicationMaster 与 NodeManager 通信以启动的。

以上两类 Container 可能在任意节点上，它们的位置通常而言是随机的，即 ApplicationMaster 可能与它管理的任务运行在一个节点上。

## 三、实验环境

Linux Ubuntu 16.0
jdk-7u75-linux-x64
hadoop-2.6.0-cdh5.4.5
hadoop-2.6.0-eclipse-cdh5.4.5.jar
eclipse-java-juno-SR2-linux-gtk-x86_64

## 四、实验内容

现有某电商网站用户对商品的收藏数据，记录了用户收藏的商品 id 以及收藏日期，名为 buyer_favorite1.txt。

buyer_favorite1.txt 包含：买家 id，商品 id，收藏日期这三个字段，数据以“\t”分割，样本数据及格式如下：

```
买家 id    商品 id     收藏日期
```

```
10181	1000481	2010-04-04 16:54:31
20001	1001597	2010-04-07 15:07:52
20001	1001560	2010-04-07 15:08:27
20042	1001368	2010-04-08 08:20:30
20067	1002061	2010-04-08 16:45:33
20056	1003289	2010-04-12 10:50:55
20056	1003290	2010-04-12 11:57:35
20056	1003292	2010-04-12 12:05:29
20054	1002420	2010-04-14 15:24:12
20055	1001679	2010-04-14 19:46:04
20054	1010675	2010-04-14 15:23:53
20054	1002429	2010-04-14 17:52:45
20076	1002427	2010-04-14 19:35:39
20054	1003326	2010-04-20 12:54:44
20056	1002420	2010-04-15 11:24:49
20064	1002422	2010-04-15 11:35:54
20056	1003066	2010-04-15 11:43:01
20056	1003055	2010-04-15 11:43:06
20056	1010183	2010-04-15 11:45:24
20056	1002422	2010-04-15 11:45:49
20056	1003100	2010-04-15 11:45:54
20056	1003094	2010-04-15 11:45:57
20056	1003064	2010-04-15 11:46:04
20056	1010178	2010-04-15 16:15:20
20076	1003101	2010-04-15 16:37:27
20076	1003103	2010-04-15 16:37:05
20076	1003100	2010-04-15 16:37:18
20076	1003066	2010-04-15 16:37:31
20054	1003103	2010-04-15 16:40:14
20054	1003100	2010-04-15 16:40:16
```

要求编写 MapReduce 程序，统计每个买家收藏商品数量。统计结果数据如下(注意：这个数据不正确，因为原始文件数据已修改，仅供参考)：

```
买家 id 商品数量
10181	1
20001	2
20042	1
20054	6
20055	1
20056	12
20064	1
20067	1
20076	5
```

## 五、实验步骤

1.切换目录到/apps/hadoop/sbin 下，启动 hadoop。

```
cd  /apps/hadoop/sbin
./start-all.sh
```

2.在 linux 上，创建一个目录/data/mapreduce1。

```
mkdir  -p  /data/mapreduce1
```

3. 切换到/data/mapreduce1 目录下，使用 wget 命令从网址 http://192.168.1.100:60000/allfiles/mapreduce1/buyer_favorite1.txt，下载文本文件 buyer_favorite1.txt。（注意：该命令不用执行，直接将 buyer_favorite1.txt 文件拷贝到/data/mapreduce1 目录下即可。）

```
cd  /data/mapreduce1
wget  http://192.168.1.100:60000/allfiles/mapreduce1/buyer_favorite1.txt
```

依然在/data/mapreduce1 目录下，使用 wget 命令，从 http://192.168.1.100:60000/allfiles/mapreduce1/hadoop2lib.tar.gz，下载项目用到的依赖包（注意：该命令不用执行，直接将文件 hadoop2lib.tar.gz 拷贝到/data/mapreduce1 目录下即可）。

```
wget  http://192.168.1.100:60000/allfiles/mapreduce1/hadoop2lib.tar.gz
```

将 hadoop2lib.tar.gz 解压到当前目录下。

```
tar  -xzvf  hadoop2lib.tar.gz
```

4.将 linux 本地/data/mapreduce1/buyer_favorite1.txt，上传到 HDFS 上的/mymapreduce1/in 目录下。若 HDFS 系统中/mymapreduce1/in 目录不存在，需提前创建。

创建命令如下：

```
hadoop  fs  -mkdir  -p  /mymapreduce1/in
```

上传命令如下：

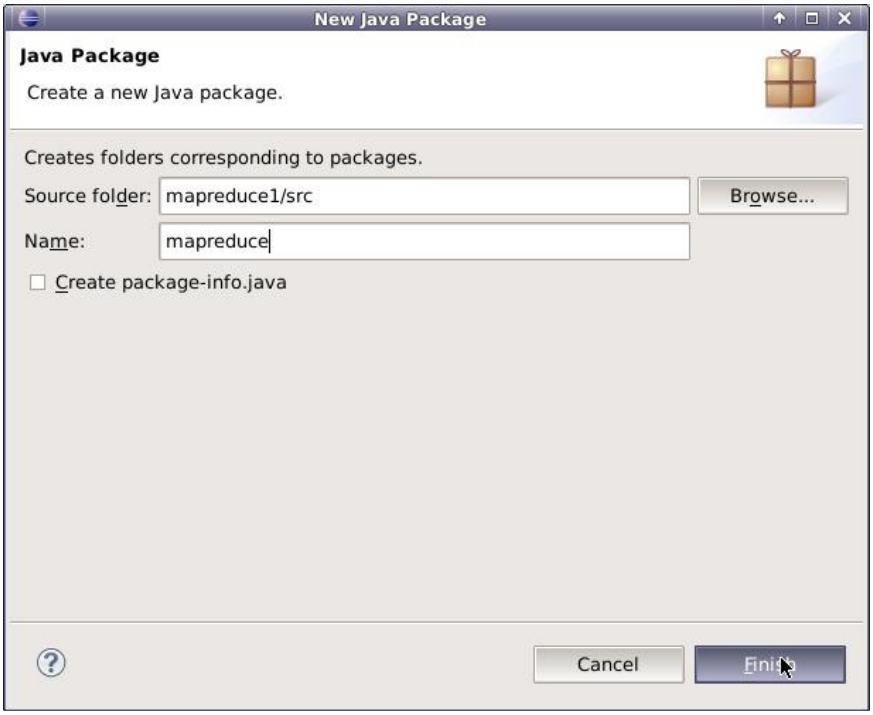
```
hadoop  fs  -put  /data/mapreduce1/buyer_favorite1.txt  /mymapreduce1/in
```

5.打开 Eclipse，新建 Java Project 项目。

并将项目名设置为 mapreduce1。

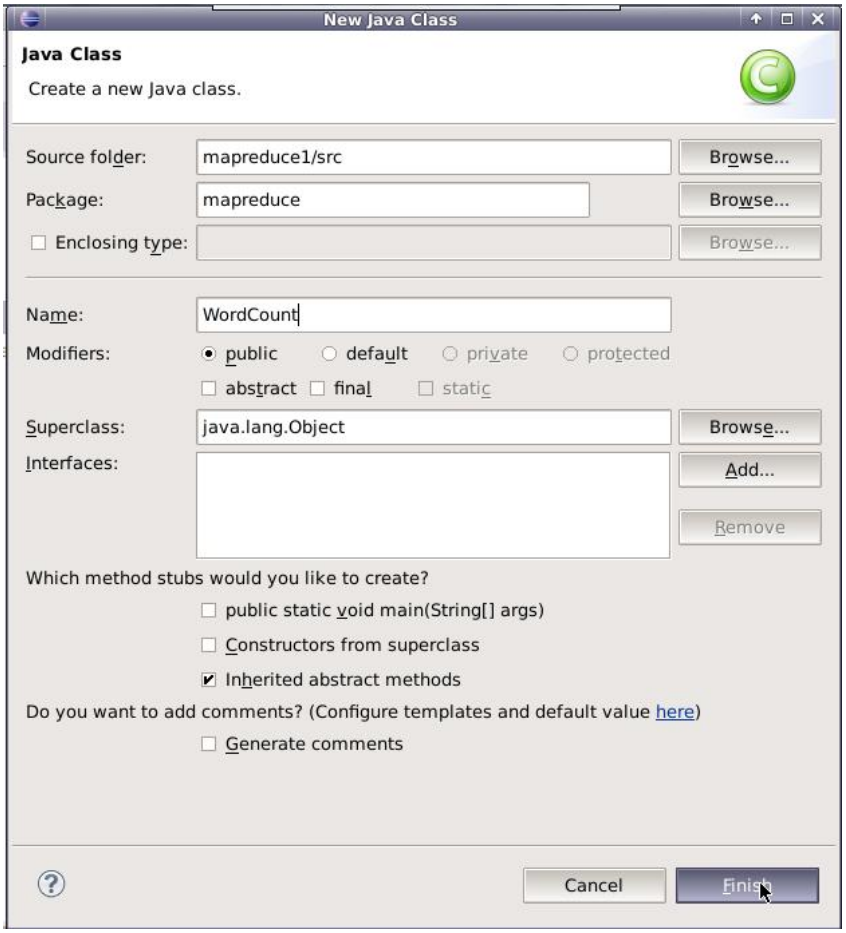

6.在项目名 mapreduce1 下，新建 package 包。


并将包命名为 mapreduce 。

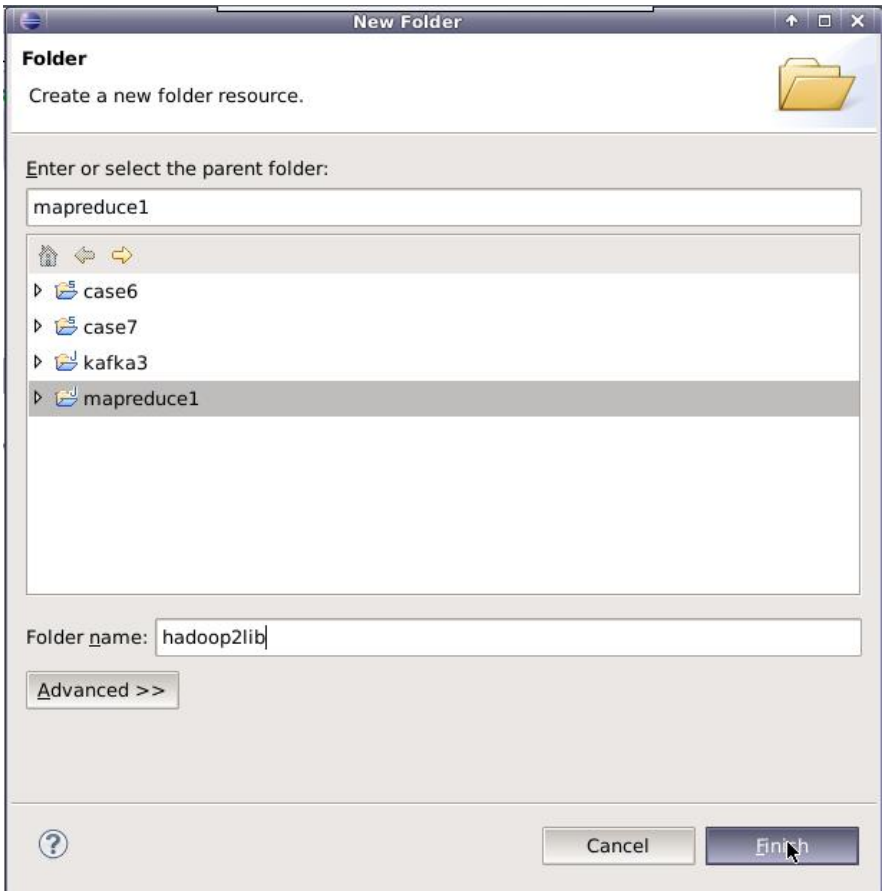



7.在创建的包 mapreduce 下，新建类。

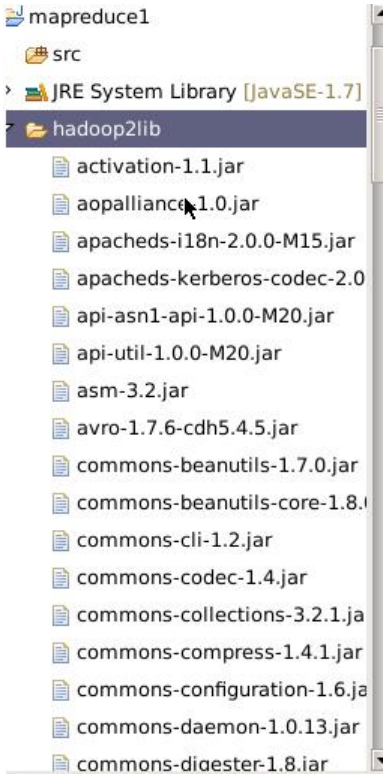

并将类命名为 WordCount。




8.添加项目所需依赖的 jar 包，右键单击项目名，新建一个目录 hadoop2lib，用于存放项目所需的 jar 包。



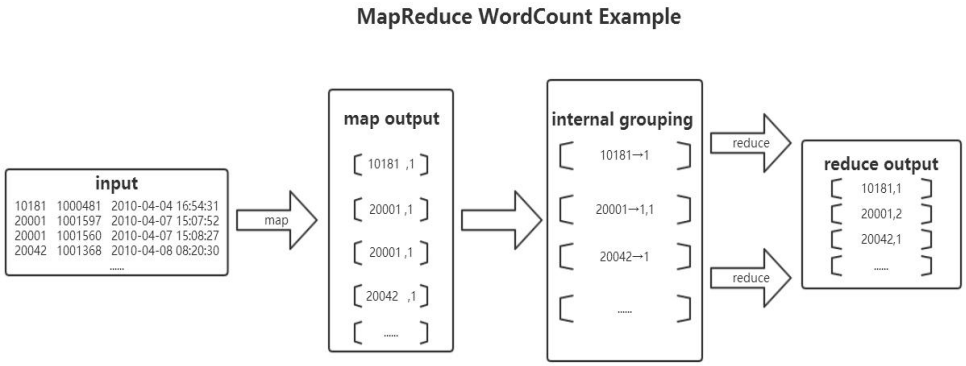

将 linux 上/data/mapreduce1 目录下，hadoop2lib 目录中的 jar 包，全部拷贝到 eclipse 中，mapreduce1 项目的 hadoop2lib 目录下。



选中 hadoop2lib 目录下所有的 jar 包，单击右键，选择 Build Path=>Add to Build Path


## 9. 编写 Java 代码，并描述其设计思路。

下图描述了该 mapreduce 的执行过程




大致思路是将 hdfs 上的文本作为输入，MapReduce 通过 InputFormat 会将文本进行切片处理，并将每行的首字母相对于文本文件的首地址的偏移量作为输入键值对的 key，文本内容作为输入键值对的 value，经过在 map 函数处理，输出中间结果<word,1>的形式，并在 reduce 函数中完成对每个单词的词频统计。整个程序代码主要包括两部分：Mapper 部分和 Reducer 部分。

Mapper 代码

```
public static class doMapper extends Mapper<Object, Text, Text,
IntWritable>{

//第一个 Object 表示输入 key 的类型；第二个 Text 表示输入 value 的类型；第三个 Text 表示输出键的类型；第四个 IntWritable 表示输出值的类型

public static final IntWritable one = new IntWritable(1);
        public static Text word = new Text();
        @Override
        protected void map(Object key, Text value, Context context)
                throws IOException, InterruptedException //抛出异常
{
```

```
            StringTokenizer tokenizer = new
StringTokenizer(value.toString(),"\t"); //StringTokenizer 是 Java 工具
包中的一个类，用于将字符串进行拆分
            word.set(tokenizer.nextToken()); //返回当前位置到下一个分
隔符之间的字符串
            context.write(word, one); //将 word 存到容器中，记一个数
        }
```

在 map 函数里有三个参数，前面两个 Object key,Text value 就是输入的 key 和 value，第三个参数 Context context 是可以记录输入的 key 和 value。例如 context.write(word,one)；此外 context 还会记录 map 运算的状态。map 阶段采用 Hadoop 的默认的作业输入方式,把输入的 value 用 StringTokenizer()方法截取出的买家 id 字段设置为 key，设置 value 为 1，然后直接输出<key,value>。

Reducer 代码

```
public static class doReducer extends Reducer<Text, IntWritable, Text,
IntWritable>{
// 参数同 Map 一样，依次表示是输入键类型，输入值类型，输出键类型，输出
值类型
private IntWritable result = new IntWritable();
        @Override
        protected void reduce(Text key, Iterable<IntWritable> values,
Context context)
    throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable value : values) {
            sum += value.get();
      }
      // for 循环遍历，将得到的 values 值累加
       result.set(sum); // 设置 result 的值
       context.write(key, result);
    }
}
```

map 输出的<key,value>先要经过 shuffle 过程把相同 key 值的所有 value 聚集起来形成<key,values>后交给 reduce 端。reduce 端接收到<key,values>之后，将输入的 key 直接复制给输出的 key,用 for 循环遍历 values 并求和，求和结果就是 key 值代表的单词出现的总次，将其设置为 value，直接输出<key,value>。

完整代码（需要自己补充完整）：

```
package mapreduce;
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
```

```
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
public class WordCount {
    public static void main(String[] args) throws IOException,
ClassNotFoundException, InterruptedException {
        Job job = Job.getInstance();
        job.setJobName("WordCount");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(doMapper.class);
        job.setReducerClass(doReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        Path in = new
Path("hdfs://localhost:9000/mymapreduce1/in/buyer_favorite1.txt");
        Path out = new Path("hdfs://localhost:9000/mymapreduce1/out");
        FileInputFormat.addInputPath(job, in);
        FileOutputFormat.setOutputPath(job, out);
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
    public static class doMapper extends Mapper<Object, Text, Text,
IntWritable>{
        // 请参考以上函数代码，完善此处代码
    }
    public static class doReducer extends Reducer<Text, IntWritable, Text,
IntWritable>{
       // 请参考以上函数代码，完善此处代码
    }
}
```

10. 在 WordCount 类文件中，单击右键=>Run As=>Run on Hadoop（如果没有 Run on Hadoop，则运行 Java Application）选项，将 MapReduce 任务提交到 Hadoop 中。


11. 待执行完毕后，打开终端或使用 hadoop eclipse 插件，查看 hdfs 上，程序输出的实验结果。

```
hadoop  fs  -ls   /mymapreduce1/out
hadoop  fs  -cat  /mymapreduce1/out/part-r-00000
```

结果大致如下：

```
zhangyu@a6a57a8dc3bf:/data/mapreduce1$ hadoop fs -ls /mymapreduce1/out
Found 2 items
-rw-r--r--   3 zhangyu supergroup          0 2017-01-05 07:35 /mymapreduce1/out/_SUCCESS
-rw-r--r--   3 zhangyu supergroup         73 2017-01-05 07:35 /mymapreduce1/out/part-r-00000
zhangyu@a6a57a8dc3bf:/data/mapreduce1$ hadoop fs -cat /mymapreduce1/out/part-r-00000
10181   1
20001   2
20042   1
20054   6
20055   1
20056   12
20064   1
20067   1
20076   5
```

将 hdfs 上的实验结果下载到本地/data/mapreduce1/目录下

```
hadoop fs -get /mymapreduce1/out/part-r-00000 /data/mapreduce1/
```

## 六、实验报告要求

要求完成如下内容。

（1） 请按实验指导文档，执行完**实验步骤**中的第 3 步后，运行：ls -l /data 命令后的结果截图。

（2） 请按实验指导文档,执行完**实验步骤**中的第 4 步后,运行:hdfs fs -ls -R /mymapreduce1/ 命令后的结果截图。

（3） 请按实验指导文档，执行完**实验步骤**中的第 9 步后，将自己补充代码后的完整代码粘贴下面。

（4） 请按实验指导文档，执行完**实验步骤**中的第 11 步后，执行 hadoop fs -cat /mymapreduce1/out/part-r-00000 后结果的截图。

（5） 请将给你们的 readme.txt 文件上传到 hdfs 系统的/mymapreduce1/in 文件夹下。写出上传命令。

（6） 请参考教学 PPT 中 wordcount 例子的源代码,以及本次实验中的源代码，编写统计 readme.txt 文件中每个单词出现的次数，并将结果存放在 hdfs 系统中的/mymapreduce1/out 目录中。

a) 粘贴源代码
b) 给出运行结束后，使用 hdfs 命令查看结果的贴图
c) 使用命令将结果下载到本地，并用 vim 编辑器打开下载的结果文件的贴图。

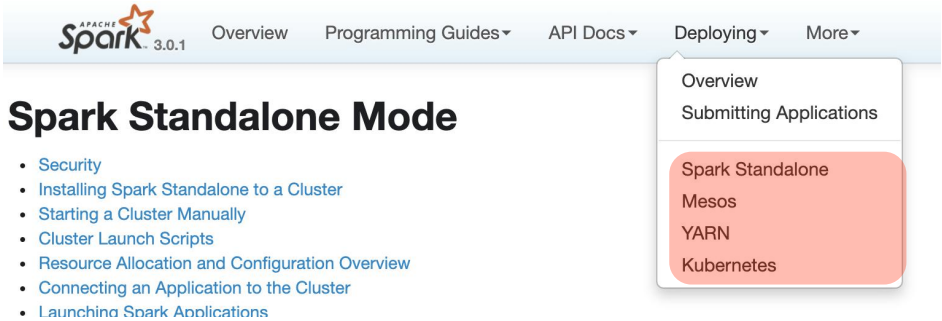
# 实验六 Spark Local 模式安装与 Anaconda 安装

## 一、实验目的

1. 了解 Spark 的六种运行模式
2. 准确理解 Spark Local 模式运行原理
3. 熟练掌握 Spark Local 模式的安装流程
4. 熟练掌握 Anaconda(或 miniconda)的安装流程

## 二、实验原理

目前 Apache Spark 主要支持四种分布式部署方式：分别是 Standalone、Spark on mesos、Spark on YARN 和 Spark on Kubernetes（图中红色框住部分）。




其中，第一种是 Spark 内部实现了容错性和资源管理，后几种则借助外部软件平台，容错性和资源管理交由统一的资源管理系统完成，即让 Spark 运行在一个通用的资源管理系统之上，这样可以与其他计算框架，比如 MapReduce 共用一个集群资源，最大的好处是降低运维成本和提高资源利用率（资源按需分配）。

### 1. Spark 运行模式概述

在实际应用中，Spark 应用程序的运行模式取决于传递给 SparkContext 的 Master 环境变量的值，个别模式还需要依赖辅助的程序接口来配合使用，目前所支持的 Master 环境变量由特定的字符串或 URL 所组成，如下所示。

（1）Local[N]：本地模式，使用多个线程，N 为线程数，如果是*号则表示使用本机器最大的线程数。

（2）Spark://hostname:port：Standalone 模式，需要部署 Spark 到相关节点，URL 为 Spark Master 主机地址和端口。

（3）Mesos://hostname:port：Mesos 模式，需要部署 Spark 和 Mesos 到相关节点，URL 为 Mesos 主机地址和端口。

（4）Yarn cluster：YARN 模式一，主程序逻辑和任务都运行在 YARN 集群中。

（5）YARN client：YARN 模式二，主程序逻辑运行在本地，具体任务运行在 YARN 集群中。

**2.Local 模式部署及程序运行**

Local 模式，顾名思义就是在本地运行，如果不加任何配置，Spark 默认设置为 Local 模式。以 SparkPi 为例，Local 模式下的应用程序的启动命令如下：

./bin/run-example org.apache.spark.examples.SparkPi local[*]

在 SparkPi 代码的具体实现中，是根据用户传入的参数来选择运行模式的，如果需要自己在代码中指定运行模式，可以通过在代码中配置 Master 为 Local 来实现，如以下程序所示。

```
from pyspark import SparkContext,SparkConf
from pyspark.sql import SparkSession
conf = SparkConf()
conf.setMaster('local[2]')# 使用本地启动，用 2 个线程
conf.setAppName(' My application ')
conf.set('spark.executor.memory', '1g')# 每个执行器分配 1G 内存
sc = SparkContext(conf)
```

当然，为了使应用程序能够更灵活地在各种部署环境下使用，不建议把与运行环境相关的设置直接在代码中写死。

## 三、实验环境

Linux Ubuntu 16.04 及以上

jdk-8u201-linux-x64.tar.gz

spark-2.4.7-bin-hadoop2.6.tgz

## 四、实验内容

在已经安装有 jdk1.8 版本以上的 Linux 系统的环境下，进行 Spark local 模式安装。

## 五、实验步骤

1.核实 java 版本

Spark 的运行依赖 jvm，先在命令行环境下，运行 java -version 命令。如果版本为 jdk1.7 版本，则可以到/apps/目录运行命令 mv java java1.7，将以前的 java 版本进行备份。然后下载 jdk-8u201-linux-x64.tar.gz 文件（在泛雅平台本课的资料中下载）。具体安装 jdk 的方法请参考以前的实验。

2.创建文件夹并准备好需要安装的 spark 软件。

首先在 linux 上创建目录/data/spark1，用于存储所需文件。

```
mkdir -p /data/spark1
```

切换目录到/data/spark1 目录，使用 wget 命令，下载所需的 Spark 安装包 spark-1.6.0-bin-hadoop2.6.tgz（假定 python 环境已经安装，如果没安装请参考以前的 Anaconda（或 miniconda）安装步骤）。

```
cd /data/spark1/
```

可以用 wget 下载 spark-2.4.7-bin-hadoop2.6.tgz 文件到该目录

关于 Spark 版本,没有严格要求,这里使用 Spark2.4.7 版本,可以下载 Spark 2.x 其他版本,但为了后续实验不要安装 spark 2.2 以前到版本。因为从 Spark 2.2 版本开始，不支持 JDK 1.7 版本，只支持 JDK 1.8 后的版本。

3，安装 Spark 软件。

切换目录到/data/spark1 目录下，将 spark 的安装包 spark-2.4.7-bin-hadoop2.6.tgz，解压缩到/apps 目录下，并将解压后的目录名重命名为 spark。

```
cd /data/spark1
tar -xzvf /data/spark1/spark-2.4.7-bin-hadoop2.6.tgz -C /apps/
cd /apps/
mv /apps/spark-2.4.7-bin-hadoop2.6/ /apps/spark
```

使用 vim 打开用户环境变量~/.bashrc。

```
vim ~/.bashrc
```

将 Spark 的配置信息追加到用户环境变量中，然后保存退出。

```
#spark
```

```
export SPARK_HOME=/apps/spark
export PATH=$SPARK_HOME/bin:$PATH
```

执行 source 命令，使用户环境变量生效。

```
source  ~/.bashrc
```

4，spark 的配置与启动 pyspark 命令环境。

下面不需要对 spark 进行任何配置，就可以启动 pyspark 进行任务处理了。切换目录到/apps/spark/bin 目录下，启动 pyspark，验证安装完的 spark 是否可用。执行如下命令

```
cd /apps/spark/bin/
pyspark
```

可以启动本地模式。运行结果类似下图：

```
(base) ubuntu@ubuntu:~$ pyspark
Python 3.7.9 (default, Aug 31 2020, 12:42:55)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
20/11/24 06:03:32 WARN Utils: Your hostname, ubuntu resolves to a loopback addre
ss: 127.0.1.1; using 172.16.7.2 instead (on interface ens33)
20/11/24 06:03:32 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
address
20/11/24 06:03:33 WARN NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... 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 setLogLeve
l(newLevel).
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.4.7
      /_/

Using Python version 3.7.9 (default, Aug 31 2020 12:42:55)
SparkSession available as 'spark'.
>>>
```

或指定本地运行，且使用机器上的所有线程

```
pyspark  --master local[*]
```

5，执行测试。

在 pyspark 中，加载 Spark 安装目录下，文件 README.md，并转变为 spark 的封装的数据集 rdd。

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

对 rdd 进行算子操作，统计文件的行数。

```
rdd.count()
```

可以看到输出的结果类似为：

```
>>> rdd = sc.textFile("/apps/spark/README.md")
>>> rdd.count()
104
>>>
```

表明安装正确。

在>>>后输入 quit() 退出 pyspark shell 环境。到此 Spark Local 模式已经安装完成！

**6. 下载与安装 Anaconda（miniconda）。**

Anaconda 是一个用于科学计算的 Python 发行版，支持 Linux, Mac, Windows, 包含了众多流行的科学计算、数据分析的 Python 包。

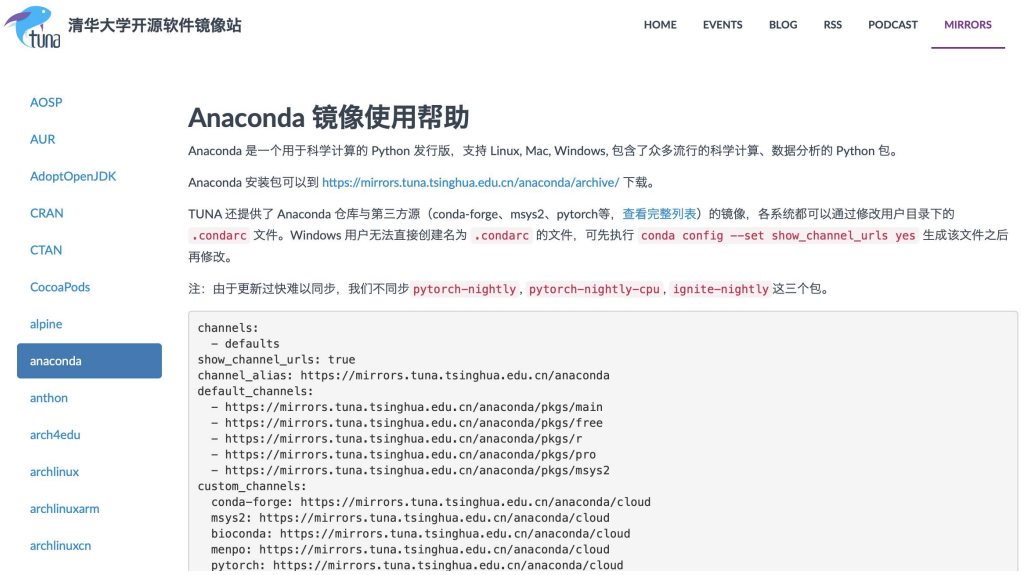
Miniconda 是一个 Anaconda 的轻量级替代，默认只包含了 Python 和 conda，但是可以通过 pip 和 conda 来安装所需要的包。Miniconda 安装包可以到 https://mirrors.tuna.tsinghua.edu.cn/anaconda/miniconda/下载。

**6.1. 下载 Anaconda（miniconda）**

下载 Anaconda3-5.2.0-Linux-x86_64.sh（或最新版，Anaconda 版本，可以自己根据需要下载，需要注意与后续软件版本的兼容性）：

方法 1：在 Anaconda 官网下载，网速慢，比较费时，如下图所示,不建议安装最新的 Python 3.8 版本，可能与 Spark 兼容不太好（下载老版本的地方在 archive）。不过下载了最新的安装版本也可以后续重新安装其中 Python 的低版本。


方法 2：在清华大学开源软件镜像站下载，如图所示。建议方法 2.




## 6.2 安装 Anaconda

1）打开 terminal。

2）打开下载文件的位置

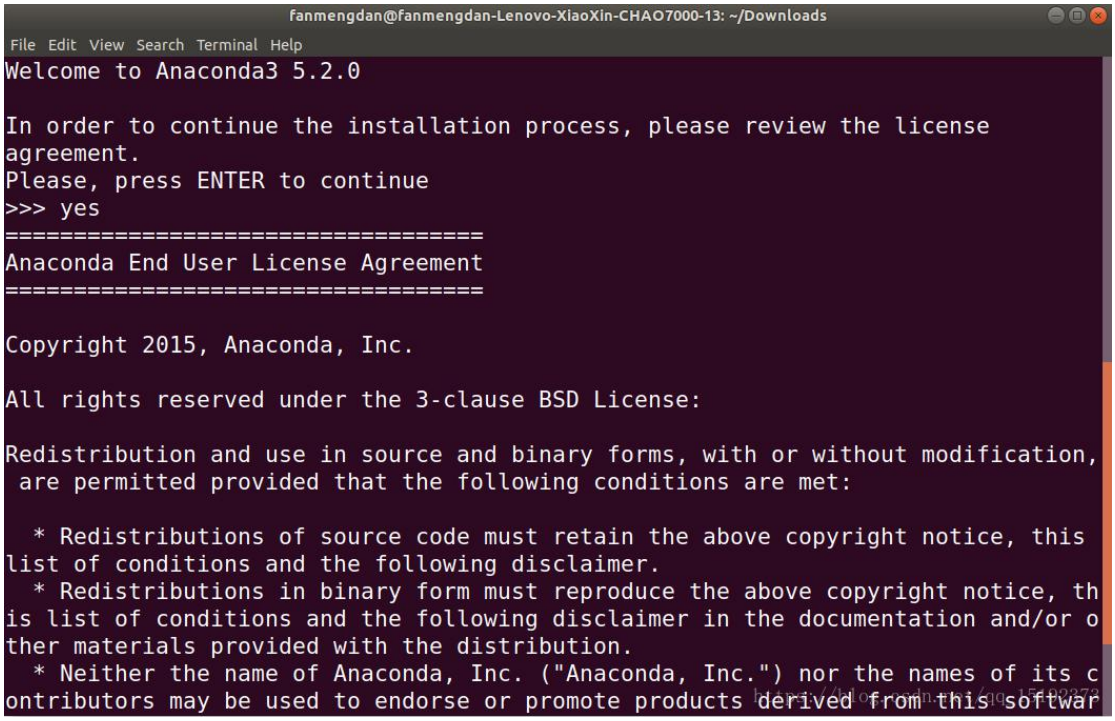
（这里假定下载文件在`~/Downloads/`文件夹下）

```
cd ~/Downloads/
```

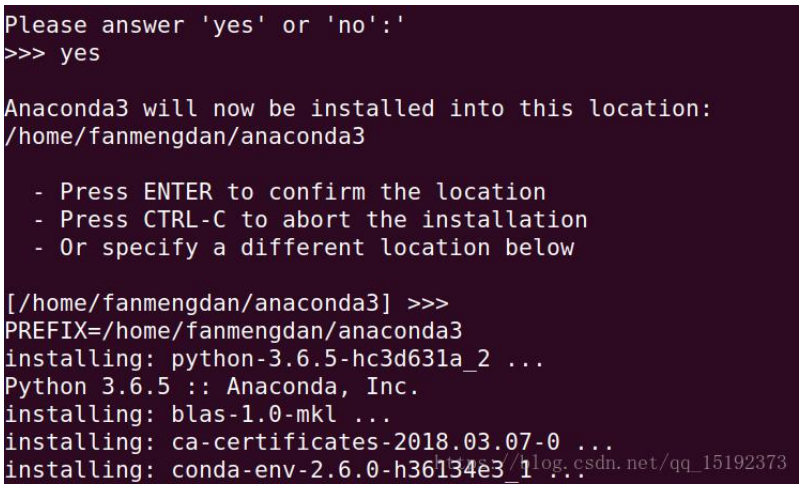
3）利用 bash 命令运行 .sh 文件。

```
bash Anaconda3-5.2.0-Linux-x86_64.sh
```

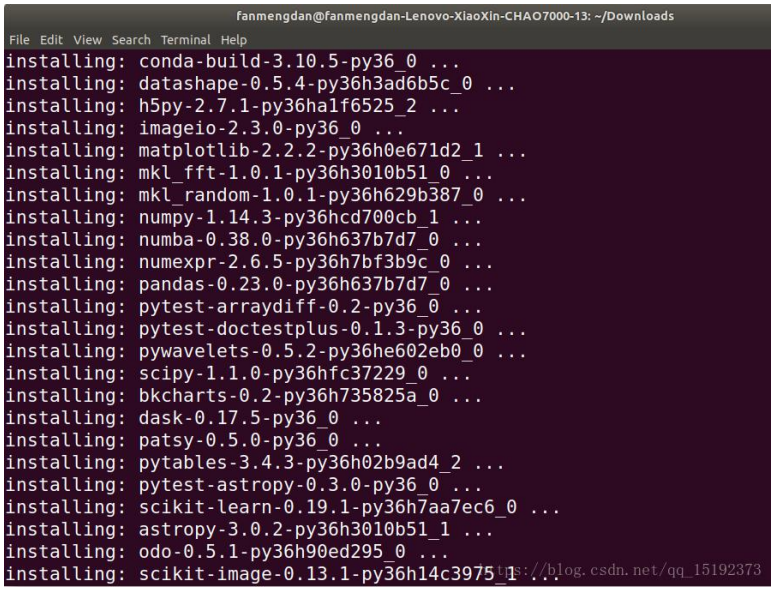
4）进入注册信息页面，输入 yes。



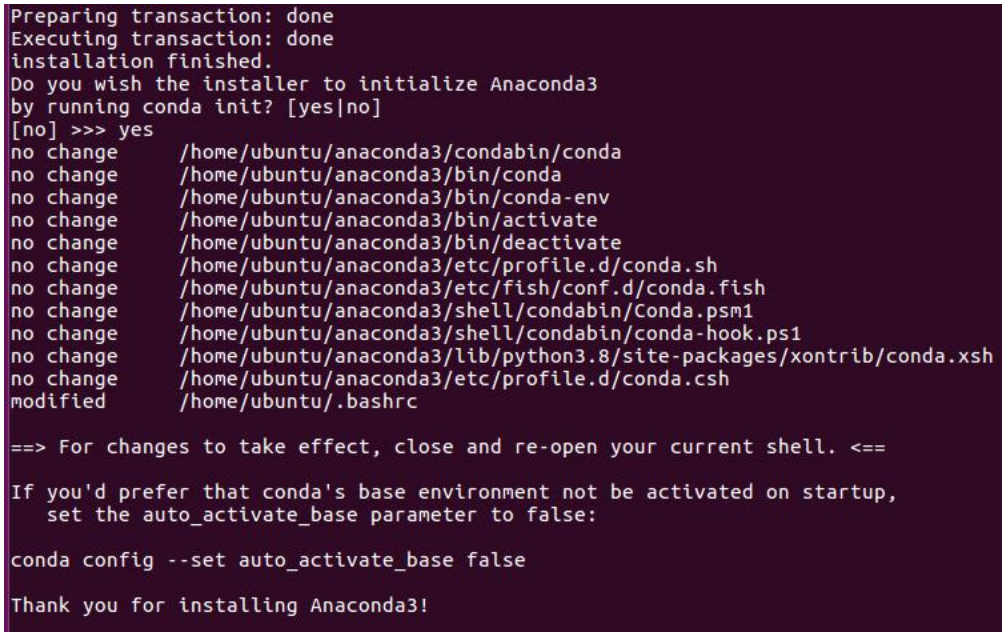

5）阅读注册信息，然后输入 yes；查看文件即将安装的位置，按 Enter，即可安装，如图所示。



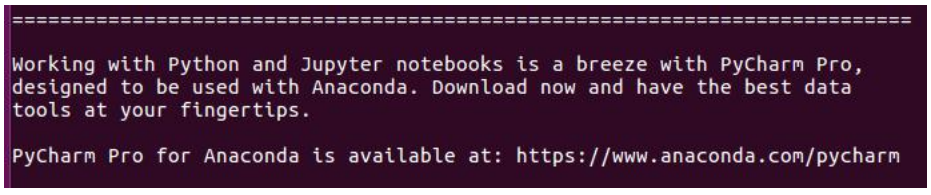

6）进入安装过程，如图所示。



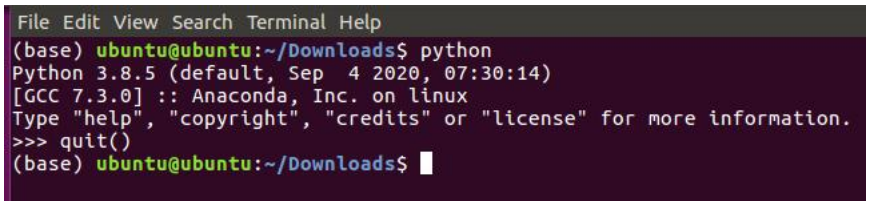

7）安装完成后，收到初始化 Anaconda 的提示信息，输入 yes。看到感谢你安装的信息，说明已经安装完成，如图




8）下面会推荐安装 pycharm 的 python 开发集成环境 IDE，可后续安装。



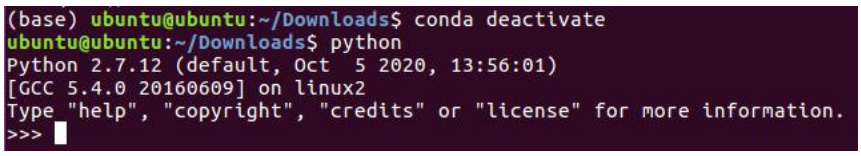

9）重启终端或在命令窗口下运行 bash 命令，即可使用 Anaconda3。




10）若在终端运行 ubuntu 自带的 python 版本，可执行：

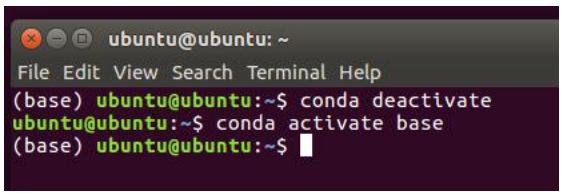
```
conda deactivate
```

```
pyhton
```




重新返回到 Anaconda 的 base 环境则可以运行以下命令：

```
conda activate base
```




11）为 Anaconda 建立国内镜像源，提高包软件的下载速度。

Anaconda 安装成功之后，再添加一下镜像，方便以后下载软件。逐行运行下面命令。

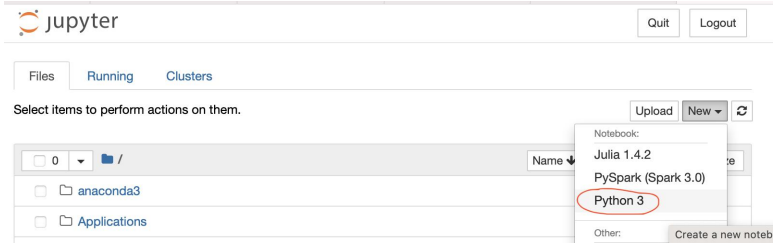
```
conda config --add channels
https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free
conda config --add channels
https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/conda-forge
conda config --add channels
https://mirrors.tuna.tsinghua.edu.cn/anaconda/cloud/bioconda
conda config --set show_channel_urls yes
```

12）卸载 anaconda（这步不要做只是大致知道如何卸载即可）

首先，在主目录下有一个 anaconda3 的文件夹，使用 rm 命令将它删除，然后，用 vim 命令进入.bashrc 文件，将与 conda 有关的语句用#注释掉

13）在命令行下启动 jupyter notebook


会自动打开览器，开启 jupyter notebook 的界面。



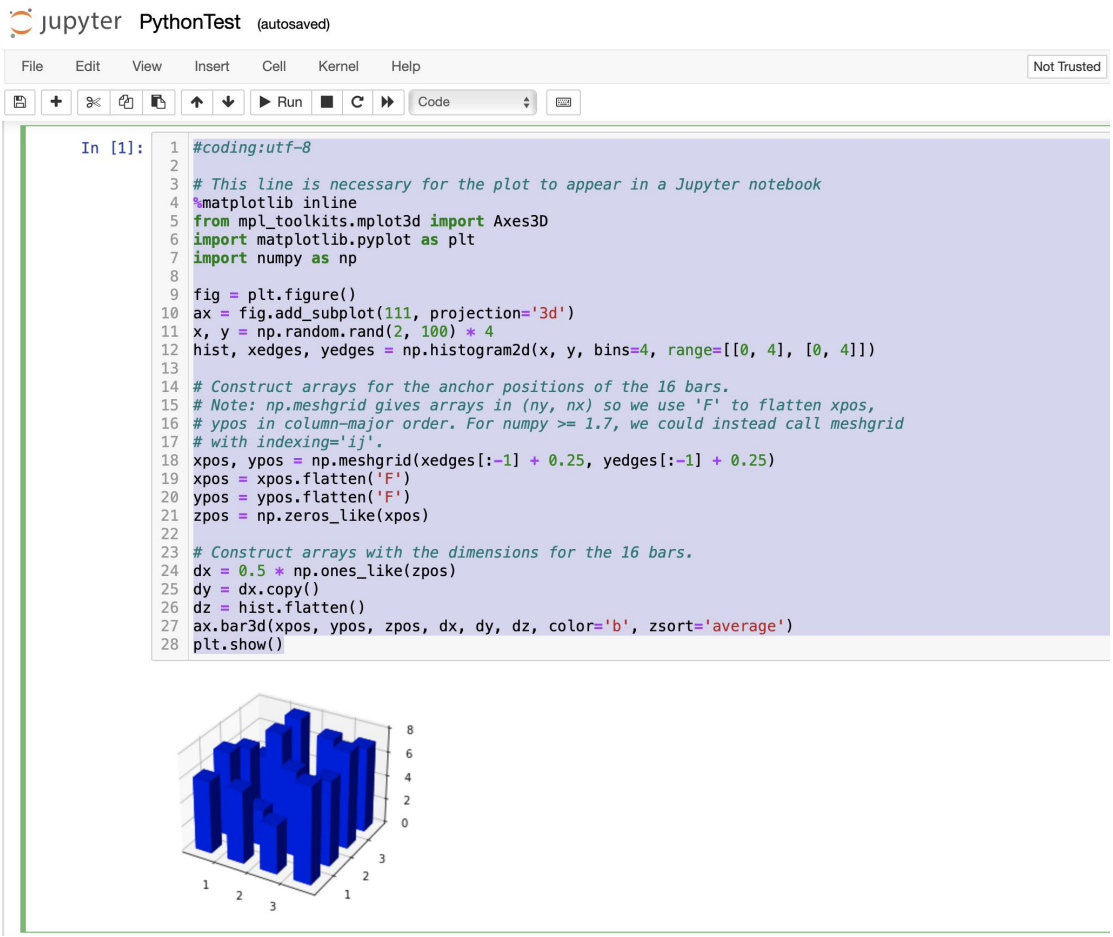

在界面中创建一个 Python3 类型的新 notebook 文件。

14）在建好的文件中输入以下代码，并运行。

```
#coding:utf-8

# This line is necessary for the plot to appear in a Jupyter notebook
%matplotlib inline
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
x, y = np.random.rand(2, 100) * 4
hist, xedges, yedges = np.histogram2d(x, y, bins=4, range=[[0, 4], [0,
4]])

# Construct arrays for the anchor positions of the 16 bars.
# Note: np.meshgrid gives arrays in (ny, nx) so we use 'F' to flatten xpos,
# ypos in column-major order. For numpy >= 1.7, we could instead call
meshgrid
# with indexing='ij'.
```

```
xpos, ypos = np.meshgrid(xedges[:-1] + 0.25, yedges[:-1] + 0.25)
xpos = xpos.flatten('F')
ypos = ypos.flatten('F')
zpos = np.zeros_like(xpos)

# Construct arrays with the dimensions for the 16 bars.
dx = 0.5 * np.ones_like(zpos)
dy = dx.copy()
dz = hist.flatten()
ax.bar3d(xpos, ypos, zpos, dx, dy, dz, color='b', zsort='average')
plt.show()
```

运行结果，如下图所示：




（注意：如果提示少库包，则可以在命令窗口下运行 conda install 安装相应

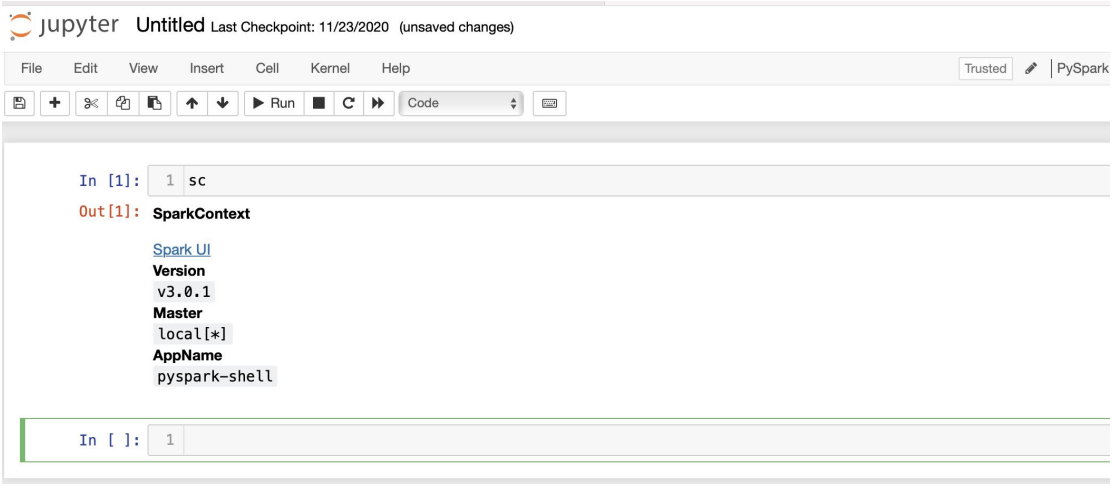
的库包。）

**退出 jupyter notebook 的方法：**

在运行 jupyter notebook 的命令窗口下，按 ctrl+c 键中断，jupyter notebook 的后台服务进程。

15）为 jupyter notebook 创建 pyspark 核（kernel）

具体创建方法，请参见资料中的“为 Jupyter 创建 pyspark kernel 的方法.pdf”文件。配置成功后，启动 jupyter notebook，创建一个核为 PySpark 类型的文件，在一个 cell 中输入 sc 并运行，看运行结果，大致应该如下图所示：




## 六、实验报告要求

要求完成如下内容。

（1） 请按实验指导文档，执行完**实验步骤**中的第 3 步后，运行：

```
cd ~
pyspark
```

命令后的结果截图。

（2） 请按实验指导文档，执行完**实验步骤**中的第 5 步后的结果截图。

（3） 请按实验指导文档，执行完**实验步骤**中的第 6.2 中 11）步后，运行以下命令后的结果图。

```
cd ~
vim .condarc
```

（4） 请按实验指导文档，执行完**实验步骤**中的第 6.2 中 14）步后结果的截图。

（5） 请按实验指导文档，执行完**实验步骤**中的第 6.2 中 15）步后，运行如下命令后的结果截图。

```
cd ~
jupyter kernelspec list
```

# 实验七 Spark RDD 基本操作

## 一、实验目的

1、了解 RDD 的特性

2、熟练掌握 RDD 的二种基本运算

3、了解 Shared variable 共享变量

4、了解 RDD Persistence 持久化

## 二、实验原理

Spark 的核心是 RDD（Resilient Distributed Dataset），即弹性分布式数据集，是由 AMPLab 实验室提出的概念，属于一种分布式的内存系统数据集应用。Spark 的主要优势来自 RDD 本身的特性，RDD 能与其他系统兼容，可以导入外部存储系统的数据集，例如 HDFS、HBase 或其他 Hadoop 数据源。

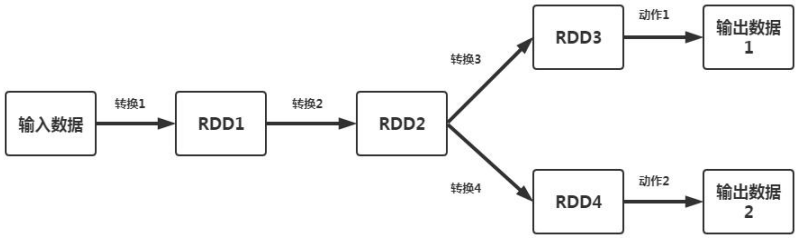
| RDD运算类型 | 说明 |
| --- | --- |
| "转换" 运算<br>Transformation | RDD执行"转换"运算的结果，会产生另外一个RDD<br>RDD具有lazy特性，所以"转换"运算并不会立刻执行，等到执行"动作"运算才会实际执行 |
| "动作" 运算<br>Action | RDD执行"动作"运算后不会产生另一个RDD，而是会产生数值、数组或写入文件系统<br>RDD执行"动作"运算时会立刻执行，并且连同之前的"转换"运算一起执行 |
| "持久化"<br>Persistence | 对于那些会重复使用的RDD，可以将RDD"持久化"在内存中作为后续使用，以提高执行性能 |

RDD 通过“转换”运算可以得出新的 RDD，但 Spark 会延迟这个“转换”动作的发生时间点(即所谓的“惰性 lazy”)。它并不会马上执行，而是等到执行了 Action 之后才会基于所有的 RDD 关系来执行转换。

执行流程如下：

- 输入数据，执行“转换 1”运算产生 RDD1，此时不会执行，只记录操作命令。
- RDD1 执行“转换 2”运算产生 RDD2，此时不会实际执行，只记录操作命令。
- RDD2 执行“转换 3”运算产生 RDD3，此时不会实际执行，只记录操作命令。

- RDD2 执行“转换 4”运算产生 RDD4，此时不会实际执行，只记录操作命令。
- RDD3 执行“动作 1”运算，此时会实际执行：“转换 1”+“转换 2”+“转换 3”+“动作 1”，产生输出数据 1。
- RDD4 执行“动作 2”运算，此时会实际执行：“转换 1” + “转换 2” + “转换 4”+“动作 2”，产生输出数据 2。
- 如果之前已经先执行了“动作 1“运算，那么”转换 1” + ”转换 2“ 已经实际执行完成，在此只会实际执行”转换 4“ + ”动作 2“，以节省运行时间。




## 三、实验环境

- Linux Ubuntu 16.04
- Python 3.x
- Anaconda
- Spark 2.4
- Jupyter Notebook

## 四、实验内容

本实验包含基本 RDD“转换”运算、多个 RDD“转换运算”、基本”动作“运算、RDD Key-Value 基本“转换”运算、多个 RDD Key-Value“转换”运算、RDD Key-Value“动作”运算、Broadcast 广播变量、accumulator 累加器、RDD Persistence 持久化等操作。

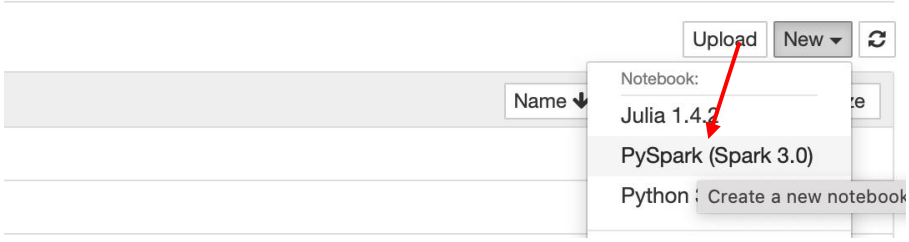
## 五、实验步骤

1.切换到/data 目录下，启动 Jupyter Notebook 界面运行 PySpark

```
cd /data
jupyter notebook
```

2. 创建一个 pyspark 的 notebook

在浏览器中点击右侧 New 按钮，选择 PySpark




a) 基本 RDD“Transformation,转换”运算

3. 创建名为 intRDD 的 RDD。

使用 SparkContext 的 parallelize 方法是最简单的创建 RDD 方式，该方法输入一个列表以创建 intRDD，但这是一个“转换”运算，并不会马上执行

```
intRDD = sc.parallelize([3,1,2,5,5])
```

4. 使用 collect()方法将 intRDD 转换为本地数据集 List。

这是一个“动作”运算，将结果从 RDD 中取回到本地，会立刻执行。

```
intRDD.collect()
```

```
In [1]: intRDD = sc.parallelize([3,1,2,5,5])

In [2]: intRDD.collect()
Out[2]: [3, 1, 2, 5, 5]
```

5. 创建一个 stringRDD

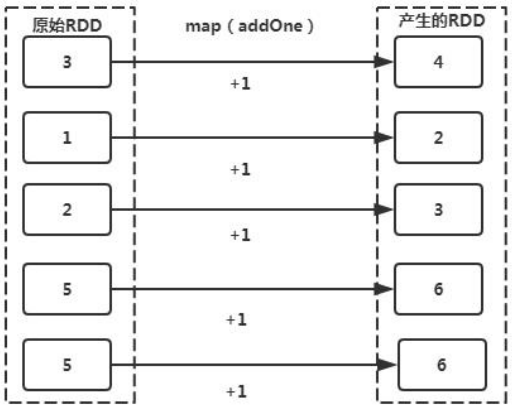
同样的，可以使用以上方法创建一个 stringRDD。

```
stringRDD =
sc.parallelize(["hadoop","spark","flink","hive","hbase"])
stringRDD.collect()
```

```
In [3]: stringRDD = sc.parallelize(["hadoop","spark","flink","hive","hbase"])
        stringRDD.collect()
Out[3]: ['hadoop', 'spark', 'flink', 'hive', 'hbase']
```

map 运算介绍

map 运算可以通过传入的函数将每一个元素经过函数运算产生另外一个 RDD。如下图，RDD 通过传入的函数 addOne 将每一个元素加 1，从而产生另外一个 RDD。




Spark 的 map 运算中，可以使用两种语句，具名函数和匿名函数。

map 运算使用具名函数

6. 定义 addOne 函数

功能为将传入的参数 x，加 1 后返回：

```
def addOne(x):
    return (x+1)
```

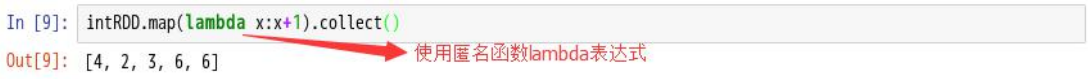
7. 将函数名称 addOne 作为参数输入 map 函数

```
intRDD.map(addOne).collect()
```


map 运算使用匿名函数

8. 使用 lambda 表达式，实现将列表中的每一个元素加 1 并返回。

```
intRDD.map(lambda x:x+1).collect()
```




具名函数和匿名函数的使用依据

- 简单的功能使用匿名函数，复杂的功能使用具名函数。使用匿名函数会让程序简洁许多，使代码更易读。复杂的功能使用具名函数，可以使程序表达的更加清楚。
- 重复使用的功能使用具名函数，当需要修改函数功能时，只需修改函数定义即可，而匿名函数则更加麻烦。

map 对字符串类型的 RDD 运算

9.针对 stringRDD 执行 map 运算。

功能为将列表中的每一个元素前加上 bigdata:

```
stringRDD.map(lambda x:"bigdata:"+x).collect()
```

```
In [11]: stringRDD.map(lambda x:"bigdata:"+x).collect()
Out[11]: ['bigdata:hadoop',
          'bigdata:spark',
          'bigdata:flink',
          'bigdata:hive',
          'bigdata:hbase']
```

将stringRDD所有字符串元素前面加上bigdata：来产生另一个RDD

## filter 数字运算

filter 可以用于对 RDD 内每一个元素进行筛选，并产生另外的 RDD。

10.让 intRDD 筛选出数字小于 3 的元素

```
intRDD.filter(lambda x:x<3).collect()
```

```
In [12]: intRDD.filter(lambda x:x<3).collect()
Out[12]: [1, 2]

In [ ]:
```

11.让 intRDD 筛选出数字等于 3 的元素

```
intRDD.filter(lambda x:x ==3).collect()
```

```
In [15]: intRDD.filter(lambda x:x ==3).collect()
Out[15]: [3]
```

12.让 intRDD 筛选出数字介于 1~5 之间的元素

```
intRDD.filter(lambda x:1<x and x<5).collect()
```

```
In [16]: intRDD.filter(lambda x:1<x and x<5).collect()
Out[16]: [3, 2]
```

13.让 intRDD 筛选出数字大于等于 5 或小于 3 的元素

```
intRDD.filter(lambda x:x>=5 or x<3).collect()
```

```
In [18]: intRDD.filter(lambda x:x>=5 or x<3).collect()
Out[18]: [1, 2, 5, 5]
```

## filter 字符串运算

14.筛选内含 h 的字符串

```
stringRDD.filter(lambda x:"h" in x).collect()
```

```
In [20]: stringRDD.filter(lambda x:"h" in x).collect()
Out[20]: ['hadoop', 'hive', 'hbase']
```

## distinct 运算

distinct 运算会删除重复的元素

15. 删除 intRDD 中的重复元素

```
intRDD.distinct().collect()
```

```
In [26]: intRDD.distinct().collect()
Out[26]: [1, 2, 3, 5]
```

16. 删除 stringRDD 中的重复元素

```
stringRDD.distinct().collect()
```

```
In [27]: stringRDD.distinct().collect()
Out[27]: ['spark', 'hive', 'hbase', 'hadoop', 'flink']
```

## randomSplit 运算

randomSplit 可以将整个集合元素以随机数的方式按照比例分为多个 RDD

17. 使用 randomSplit 将整个集合按照 4:6 的比例分为两个 RDD，这里注意权重总和加起来应为 1

```
sRDD=intRDD.randomSplit([0.4,0.6])
```

18. 第一个 RDD

```
sRDD[0].collect()
```

```
In [40]: sRDD[0].collect()
Out[40]: [5]
```

19. 第二个 RDD

```
sRDD[1].collect()
```

```
In [41]: sRDD[1].collect()
Out[41]: [3, 1, 2, 5]
```

## groupBy 运算

groupBy 可以按照传入的匿名函数规则将数据分为多个 List

20. 使用 groupBy 运算将整个集合分成奇数与偶数

```
gRDD=intRDD.groupBy(lambda x:"even" if (x%2 == 0) else
"odd").collect()
```

使用 groupBy 运算时，传入的匿名函数为(lambda x:"even" if (x%2 == 0) else "odd")，以 if(x%2 == 0)判断传入的数据除以 2 的余数是否为 0，如果是就是偶数，否则为奇数。完成的功能就是将数据分为两组，奇数组和偶数组，数据的组织形式为[('even',偶数列表),('odd',奇数列表)]。

21. 查看偶数 List

```
print(gRDD[0][0],sorted(gRDD[0][1]))
```

```
In [3]: 1 print(gRDD[0][0],sorted(gRDD[0][1]))
        even [2]
```

22.查看奇数 List

```
print(gRDD[1][0],sorted(gRDD[1][1]))
```

```
In [4]: 1 print(gRDD[1][0],sorted(gRDD[1][1]))
        odd [1, 3, 5, 5]
```

**多个 RDD“转换运算”**

RDD 也支持执行多个 RDD 的运算

23.创建 3 个 intRDD

```
intRDD1 = sc.parallelize([3,1,2,5,5])
intRDD2 = sc.parallelize([5,6])
intRDD3 = sc.parallelize([2,7])
```

24.使用 union 函数进行并集运算

```
intRDD1.union(intRDD2).union(intRDD3).collect()
```

```
In [49]: intRDD1.union(intRDD2).union(intRDD3).collect()
Out[49]: [3, 1, 2, 5, 5, 5, 6, 2, 7]
```

使用 intersection 交集运算

25.将 intRDD1、intRDD2 进行交集运算

```
intRDD1.intersection(intRDD2).collect()
```

```
In [50]: intRDD1.intersection(intRDD2).collect()
Out[50]: [5]
```

使用 subtract 差集运算

26.将 intRDD1、intRDD2 进行差集运算

```
intRDD1.subtract(intRDD2).collect()
```

```
In [51]: intRDD1.subtract(intRDD2).collect()
Out[51]: [1, 2, 3]
```

使用 cartesian 笛卡尔积运算

27.将 intRDD1、intRDD2 进行笛卡尔积运算

```
print(intRDD1.cartesian(intRDD2).collect())
```

```
In [52]: print(intRDD1.cartesian(intRDD2).collect())
         [(3, 5), (3, 6), (1, 5), (1, 6), (2, 5), (2, 6), (5, 5), (5, 6), (5, 5), (5, 6)]
```

**b) RDD 的基本"Action,动作"运算**

可以使用下列命令读取 RDD 内的元素，这是 Actions 运算，立刻执行且不会产生新的 RDD。

28.取出 intRDD 中的第一项元素

```
intRDD.first()
```

```
In [53]: intRDD.first()
Out[53]: 3
```

29.取出 intRDD 中的前二项元素

```
intRDD.take(2)
```

```
In [56]: intRDD.take(2)
Out[56]: [3, 1]
```

30.从小到大排序，取出 intRDD 中的前三项

```
intRDD.takeOrdered(3)
```

```
In [57]: intRDD.takeOrdered(3)
Out[57]: [1, 2, 3]
```

31.从到到小排序，取出 intRDD 中的前三项

```
intRDD.takeOrdered(3,key=lambda x:-x)
```

```
In [58]: intRDD.takeOrdered(3,key=lambda x:-x)
Out[58]: [5, 5, 3]
```

32.还可以将 RDD 内的元素进行统计运算

```
intRDD.stats()  #统计信息 （注意：只能对 RDD 中元素类型为数字型才可以）
```

```
In [60]: intRDD.stats()
Out[60]: (count: 5, mean: 3.2, stdev: 1.6, max: 5.0, min: 1.0)
```

```
intRDD.min()    #求最小值
```

```
In [61]: intRDD.min()
Out[61]: 1
```

```
intRDD.max()    #求最大值
```

```
In [62]: intRDD.max()
Out[62]: 5
```

```
intRDD.stdev()  #求标准差
```

```
In [63]: intRDD.stdev()
Out[63]: 1.6
```

```
intRDD.count()  #统计 RDD 中元素个数，即计数
```

```
In [64]: intRDD.count()
Out[64]: 5
```

```
intRDD.sum()      #求总和
```

```
In [65]: intRDD.sum()
Out[65]: 16
```

```
intRDD.mean()     #求平均值
```

```
In [66]: intRDD.mean()
Out[66]: 3.2
```

## c) Key-Value RDD（也称 pairRDD）的基本“转换”运算

Spark RDD 支持键值（Key-Value）运算，Key-Value 运算也是 Map/Reduce 的基础。

### 33. 创建 kvRDD1

```
kvRDD1 = sc.parallelize([(3,4),(3,6),(5,6),(1,2)])
kvRDD1.collect()
```

```
In [67]: kvRDD1 = sc.parallelize([(3,4),(3,6),(5,6),(1,2)])
         kvRDD1.collect()
Out[67]: [(3, 4), (3, 6), (5, 6), (1, 2)]
```

第一个字段是 Key，第二个字段是 Value，例如第一项数据（3，4），3 是 key 值，4 是 value 值。

### 34. 列出全部的 Key 值

```
kvRDD1.keys().collect()
```

```
In [69]: kvRDD1.keys().collect()
Out[69]: [3, 3, 5, 1]
```

### 35. 列出全部的 Values 值

```
kvRDD1.values().collect()
```

```
In [70]: kvRDD1.values().collect()
Out[70]: [4, 6, 6, 2]
```

### 36. 使用 filter 筛选出 key 小于 5 的数据

```
kvRDD1.filter(lambda keyValue:keyValue[0]<5).collect()
```

```
In [71]: kvRDD1.filter(lambda keyValue:keyValue[0]<5).collect()
Out[71]: [(3, 4), (3, 6), (1, 2)]
```

### 37. 使用 filter 筛选出 value 小于 5 的数据

```
kvRDD1.filter(lambda keyValue:keyValue[1]<5).collect()
```

```
In [72]: kvRDD1.filter(lambda keyValue:keyValue[1]<5).collect()
Out[72]: [(3, 4), (1, 2)]
```

mapValues 运算可以针对 RDD 内每一组（key,value）进行运算，并且产生另外一个 RDD。

38. 将 Values 中的每一个值进行平方运算

```
kvRDD1.mapValues(lambda x:x*x).collect()
```

```
In [73]: kvRDD1.mapValues(lambda x:x*x).collect()
Out[73]: [(3, 16), (3, 36), (5, 36), (1, 4)]
```

39. 使用 sortByKey()按照 key 值从小到大排序，传入参数默认值是 true，也就是从小到大排序

```
kvRDD1.sortByKey(ascending=True).collect()
```

```
In [74]: kvRDD1.sortByKey(ascending=True).collect()
Out[74]: [(1, 2), (3, 4), (3, 6), (5, 6)]
```

40. 由于默认是 true，所以可以不传入参数

```
kvRDD1.sortByKey().collect()
```

```
In [75]: kvRDD1.sortByKey().collect()
Out[75]: [(1, 2), (3, 4), (3, 6), (5, 6)]
```

41. 使用 sortByKey()按照 key 值从大到小排序，只需将参数改为 False 即可。

```
kvRDD1.sortByKey(ascending=False).collect()
```

```
In [76]: kvRDD1.sortByKey(ascending=False).collect()
Out[76]: [(5, 6), (3, 4), (3, 6), (1, 2)]
```

42. reduceByKey 会将具有相同 Key 值的数据合并

```
kvRDD1.reduceByKey(lambda x,y:x+y).collect()
```

```
In [78]: kvRDD1.reduceByKey(lambda x,y:x+y).collect()
Out[78]: [(1, 2), (3, 10), (5, 6)]
```

d) 多个 RDD Key-Value “转换”运算

43. 创建 kvRDD1 和 kvRDD2

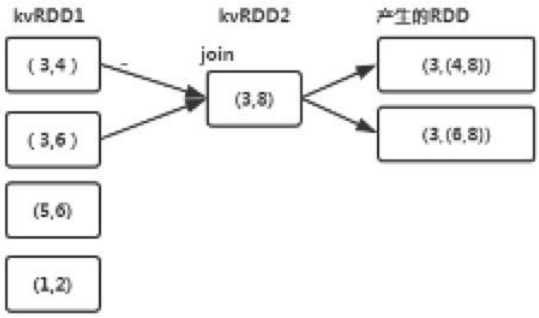
```
kvRDD1 = sc.parallelize([(3,4),(3,6),(5,6),(1,2)])
kvRDD2 = sc.parallelize([(3,8)])
```

Key-Value RDD join 运算

44. 将 kvRDD1 和 kvRDD2 进行 join 运算

```
kvRDD1.join(kvRDD2).collect()
```

```
In [83]: kvRDD1.join(kvRDD2).collect()
Out[83]: [(3, (4, 8)), (3, (6, 8))]
```



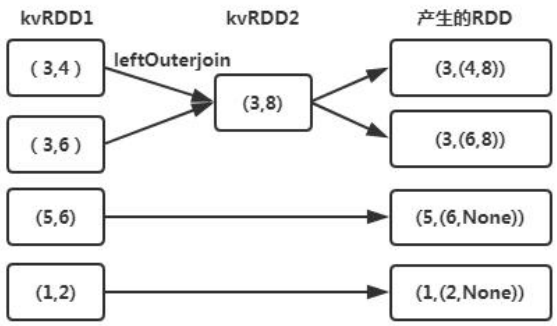

Key-Value RDD leftOuterJoin 运算

45.将 kvRDD1 和 kvRDD2 进行 Left Join 运算

```
kvRDD1.leftOuterJoin(kvRDD2).collect()
```

```
In [84]: kvRDD1.leftOuterJoin(kvRDD2).collect()
Out[84]: [(1, (2, None)), (3, (4, 8)), (3, (6, 8)), (5, (6, None))]
```

leftOuterJoin 会从左边的集合（kvRDD1）对应到右边的集合（kvRDD2），并显示所有左边的集合（kvRDD1）中的所有元素。



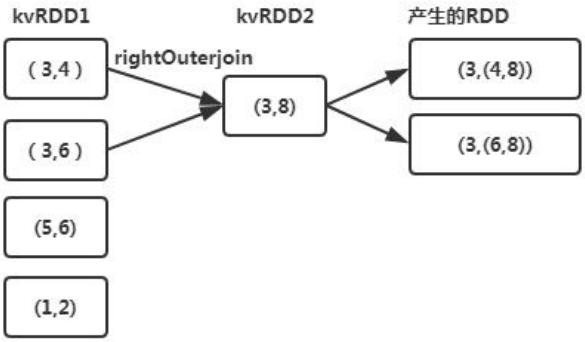

Key-Value RDD rightOuterJoin 运算

46.将 kvRDD1 和 kvRDD2 进行 Right Join 运算

```
kvRDD1.rightOuterJoin(kvRDD2).collect()
```

```
In [85]: kvRDD1.rightOuterJoin(kvRDD2).collect()
Out[85]: [(3, (4, 8)), (3, (6, 8))]
```

rightOuterJoin 会从右边的集合（kvRDD2）对应到左边的集合（kvRDD1），并显示所有右边的集合（kvRDD2）中的所有元素。




## d) RDD Key-Value“动作”运算

47.获取 kvRDD1 中的第一项数据

```
kvRDD1.first()
```

```
In [87]: kvRDD1.first()
Out[87]: (3, 4)
```

48.获取 kvRDD1 中的前两项数据

```
kvRDD1.take(2)
```

```
In [88]: kvRDD1.take(2)
Out[88]: [(3, 4), (3, 6)]
```

读取第一项数据的元素

49.使用 kvRDD1.first()获取第一项数据并存储在 kvFirst 变量中

在 python 中可以使用 kvFirst[0]获取第 1 个元素、kvFirst[1]取得第 2 个元素，以此类推。

```
kvFirst=kvRDD1.first()
kvFirst
```

```
In [90]: kvFirst=kvRDD1.first()
         kvFirst
Out[90]: (3, 4)
```

50.读取第一项数据中的第一个元素，也就是 Key 值

```
kvFirst[0]
```

```
In [91]: kvFirst[0]
Out[91]: 3
```

51.读取第一项数据中的第二个元素，也就是 Value 值

```
kvFirst[1]
```

```
In [92]: kvFirst[1]
Out[92]: 4
```

计算 RDD 中相同 Key 值的个数

52.例如 kvRDD1 数据(3,4),(3,6),(5,6),(1,2),其中 key 值为 3 的有两项数据，其余 key 值（1 和 5）都只有一项数据，因此结果为：1→1，3→2,5→1。

```
kvRDD1.countByKey()
```

```
In [93]: kvRDD1.countByKey()
Out[93]: defaultdict(int, {1: 1, 3: 2, 5: 1})
```

collectAsMap 创建 Key-Value 的字典

53.Python 字典数据类型就好像真实的字典，可以用“字”查询”字的解说“。其中“字”就是 key，“字的解说”就是 value。

可以使用 collectAsMap 创建 Key-Value 的字典。例如 kvRDD1 数据(3,4),(3,6),(5,6),(1,2)能产生字典 dict(1:2,3:6,5:6)，不过 Key=3 有两个值 4 和 6，系统只能自动对应到其中的值 6。

```
KV=kvRDD1.collectAsMap()
KV
```

```
In [94]: KV=kvRDD1.collectAsMap()
         KV
Out[94]: {1: 2, 3: 6, 5: 6}
```

54.查看 KV 的数据类型

```
type(KV)
```

```
In [95]: type(KV)
Out[95]: dict
```

使用对照表转换数据

55.字典建立后，使用字典转换数据，向 KV 字典传入参数 3

```
KV[3]
```

```
In [96]: KV[3]
Out[96]: 6
```

Key-Value lookup 运算

可以使用 lookup 运算来根据 key 值查找 value 的值

56.查找 key 为 3 的 value 值

```
kvRDD1.lookup(3)
```

```
In [97]: kvRDD1.lookup(3)
Out[97]: [4, 6]
```

e) Broadcast 广播变量

共享变量（Shared variable）可用于节省内存与运行时间，提高并行处理的执行效率，共享变量包括 Broadcast 和 accumulator

- Broadcast--广播变量
- accumulator--累加器

## 57.不使用广播变量的例子

创建水果编号和与名称对照表，然后使用此对照表将水果编号转换为水果名称

1） 创建水果编号与名称的 Key-Value RDD，名字为 kvFruit

```
kvFruit =
sc.parallelize([(1,"apple"),(2,"orange"),(3,"banana"),(4,"grape")])
```

2） 创建 fruitMap 字典：使用 collectAsMap 创建 fruitMap 字典（水果编号与名称对照表）

```
fruitMap=kvFruit.collectAsMap()
print("dict:"+str(fruitMap))
```

```
In [108]: fruitMap=kvFruit.collectAsMap()
          print("dict:"+str(fruitMap))

          dict:{1: 'apple', 2: 'orange', 3: 'banana', 4: 'grape'}
```

3） 创建 fruitIds RDD

```
fruitIds=sc.parallelize([2,4,1,3])
print("fruitIds:"+str(fruitIds.collect()))
```

```
fruitIds=sc.parallelize([2,4,1,3])
print("fruitIds:"+str(fruitIds.collect()))

fruitIds:[2, 4, 1, 3]
```

4） 使用 fruitMap 字典进行转换

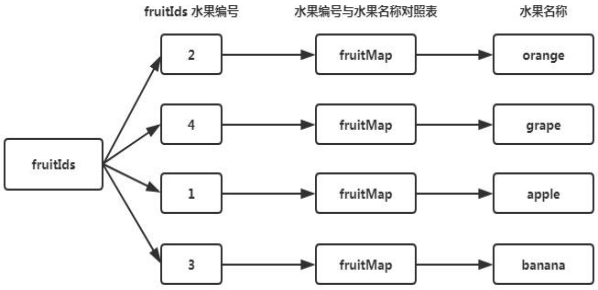
即根据给定的 fruitIds 数据来找到对应的水果名字。

```
fruitNames=fruitIds.map(lambda x:fruitMap[x]).collect()
print("fruitNames:"+str(fruitNames))
```

```
In [104]: fruitNames=fruitIds.map(lambda x:fruitMap[x]).collect()
          print("fruitNames:"+str(fruitNames))

          fruitNames:['orange', 'grape', 'apple', 'banana']
```

以上的例子执行起来虽然没问题，但是在并行处理中每执行一次转换（即一个转换 Task）都必须将 fruitMap 传到 Worker 节点，才能够执行转换。如果字

典 fruitMap（对照表）很大，而且需要转换的 fruitIds 水果编号 RDD 也很大，就会消耗很多内存与时间。




为了解决这个问题，可以使用 Broadcast 广播变量。

Broadcast 广播变量的使用规则如下：

- 可以使用 SparkContext.broadcast([初始值])创建。
- 使用.value 的方法来读取广播变量的值
- Broadcast 广播变量被创建后不能修改

下列使用 Broadcast 广播变量的例子与之前的例子类似，不同之处是使用 sc.broadcast 传入 fruitMap 作为参数，创建 bcFruitMap 广播变量，使用 bcFruitMap.value(x)广播变量转换为 fruitNames 水果名称。

## 58.使用广播变量的例子

**1） 创建 kvFruit，这是水果编号与名称的 Key-Value RDD**

```
kvFruit =
sc.parallelize([(1,"apple"),(2,"orange"),(3,"banana"),(4,"grape")])
```

**2） 使用 collectAsMap 创建 fruitMap 字典（水果编号与名称对照表）**

```
fruitMap=kvFruit.collectAsMap()
print("dict:"+str(fruitMap))
```

```
In [110]: fruitMap=kvFruit.collectAsMap()
          print("dict:"+str(fruitMap))

dict:{1: 'apple', 2: 'orange', 3: 'banana', 4: 'grape'}
```

以上两步与不使用广播变量一样。

**3） 将 fruitMap 字典转换为 bcFruitMap 广播变量**

使用 sc.broadcast 传入 fruitMap 参数，创建 bcFruitMap 广播变量

```
fruitMap=kvFruit.collectAsMap()  #这个可以不要，上面已经执行了。
bcFruitMap=sc.broadcast(fruitMap)
```

```
print("dict;"+str(fruitMap))
```

```
In [111]: fruitMap=kvFruit.collectAsMap()
          bcFruitMap=sc.broadcast(fruitMap)
          print("dict;"+str(fruitMap))

          dict;{1: 'apple', 2: 'orange', 3: 'banana', 4: 'grape'}
```

4) 创建 fruitIds RDD

```
fruitIds=sc.parallelize([2,4,1,3])
print("fruitIds:"+str(fruitIds.collect()))
```

```
fruitIds=sc.parallelize([2,4,1,3])
print("fruitIds:"+str(fruitIds.collect()))

fruitIds:[2, 4, 1, 3]
```

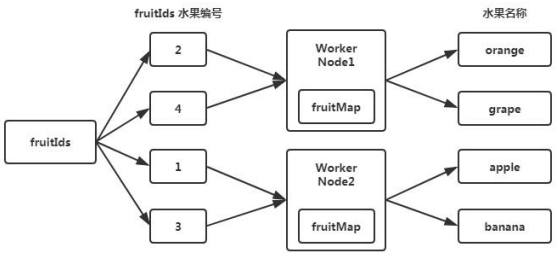
5) 使用广播变量

使用 bcFruitMap.value[x]广播变量将 fruitIds 转换为 fruitNames

```
fruitNames=fruitIds.map(lambda x:bcFruitMap.value[x]).collect()
print("fruitNames:"+str(fruitNames))
```

```
In [113]: fruitNames=fruitIds.map(lambda x:bcFruitMap.value[x]).collect()
          print("fruitNames:"+str(fruitNames))

          fruitNames:['orange', 'grape', 'apple', 'banana']
```

执行结果与之前的例子相同，在并行处理中 bcFruitMap 广播变量会传送到 Worker Node 机器，并且存储在内存中，后续在此 Worker Node 上的转换 Task 都可以使用这个 bcFruitMap 广播变量执行转换，这样就可以节省很多内存与传送时间。




## f) accumulator 累加器

计算总和是 MapReduce 常用的运算。为了方便并行处理，Spark 特别提供了 accumulator 累加器共享变量（Shared variable），使用规则如下：

- accumulator 累加器可以使用 SparkContext.accumulator([初始值])来创建。
- 使用.add()进行累加
- 在 task 中，例如 foreach 循环中，不能读取累加器的值

- 只有驱动程序，也就是循环外，才可以使用.value 来读取累加器的值

## 59.使用累加器的例子

1） 创建名为 intRDD 的 RDD

```
intRDD = sc.parallelize([3,1,2,5,5])
```

2） 创建 total 累加器，初始值使用 0.0，所以是 Double 的类型

```
total = sc.accumulator(0.0)
```

3） 创建 num 累加器，初始值使用 0，所以是 Int 类型

```
num = sc.accumulator(0)
```

4） 使用 foreach 传入参数 i，针对每一项数据执行，total 累加 intRDD 元素的值、num 累加 intRDD 元素的数量

```
intRDD.foreach(lambda i :[total.add(i),num.add(1)])
```

5） 计算平均=求和/计数，并显示总和、数量

```
avg = total.value/num.value
print("total="+str(total.value)+"num="+str(num.value)+"avg="+str(avg))
```

```
In [119]: avg = total.value/num.value
          print("total="+str(total.value)+"num="+str(num.value)+"avg="+str(avg))

          total=16.0num=5avg=3.2
```

## g） RDD Persistence 持久化

Spark RDD 持久化机制可以用于将需要重复运算的 RDD 存储在内存中，以便大幅提升运算效率。

Spark RDD 持久化使用方法如下：

- RDD.persist(存储的等级)——可以指定存储等级，默认是 MEMORY_ONLY，也就是存储在内存中
- RDD.unpersist()——取消持久化

## 60.使用持久化的例子

1） 创建名为 intRddMemory 的 RDD

```
intRddMemory = sc.parallelize([3,1,2,5,5])
```

2） 使用 RDD.persist()将 intRddMemory 进行持久化

```
intRddMemory.persist()  # 等价 cache()方法
```

```
In [122]: intRddMemory.persist()
Out[122]: ParallelCollectionRDD[220] at parallelize at PythonRDD.scala:423
```

3） 查看是否已经缓存

```
intRddMemory.is_cached
```

```
In [123]: intRddMemory.is_cached
Out[123]: True
```

4） 取消持久化

```
intRddMemory.unpersist()
```

```
In [38]: intRddMemory.unpersist()
Out[38]: ParallelCollectionRDD[55] at parallelize at PythonRDD.scala:423
```

本实验素材来自章鱼公司，在此表示感谢！

## 六、实验报告要求

请在 jupyter nootbook 编程环境下完成实验指导文档中的测试代码，仔细观察结果并理解掌握基本的 RDD 操作，并最终将编辑生成的 jupyter notebook 文件（后缀为.ipynb 的文件）以附件的形式提交。

# 实验八 Spark SQL 基本操作

## 一、实验目的

1.掌握 Spark SQL 的基本操作

2.学习使用 DataFrame 创建表及查询数据

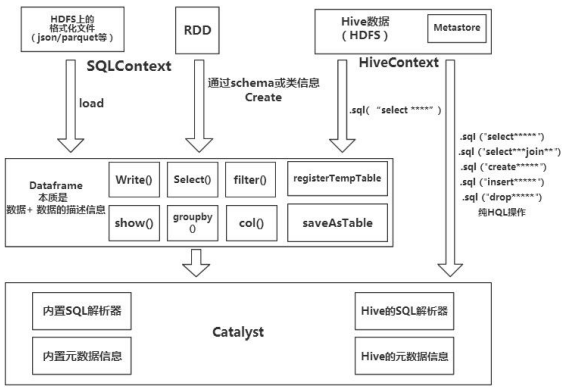
3.了解 Spark SQL 对文件的存储

## 二、实验原理

Spark SQL 的前身是 Shark，Shark 是伯克利实验室 Spark 生态环境的组件之一，它能运行在 Spark 引擎上，从而使得 SQL 查询的速度得到 10-100 倍的提升，但是，随着 Spark 的发展，由于 Shark 对于 Hive 的太多依赖（如采用 Hive 的语法解析器、查询优化器等），制约了 Spark 的 One Stack Rule Them All 的既定方针，制约了 Spark 各个组件的相互集成，所以提出了 SparkSQL 项目。SparkSQL 抛弃了原有 Shark 的代码，汲取了 Shark 的一些优点，如内存列存储（In-MemoryColumnarStorage）、Hive 兼容性等，重新开发了 SparkSQL 代码；由于摆脱了对 Hive 的依赖性，SparkSQL 无论在数据兼容、性能优化、组件扩展方面都得到了极大的方便。

Spark SQL 重要的是操作 DataFrame，DataFrame 本质是数据 + 数据的描述信息（结构元信息）。DataFrame 可以从本地数据集来创建，Spark SQL 本身提供了对文件进行 Load、Read 和 Save 的操作，Read 和 Load 可以通过读文件创建 DataFrame。Save 可以把 DataFrame 中的数据保存到文件或者说用具体的格式来指明我们要读取的文件类型，以及用具体的格式来指出要输出的文件是什么类型。此外，DatFrame 还可以通过 RDD 来创建。

Spark SQL 执行基本操作时，内部结构流程图如下：



所有的上述 SQL 及 DataFrame 操作最终都通过 Catalyst 翻译成 Spark 程序 RDD 操作代码。

RDD、DataFrame、Spark SQL 三者的主要差异在于是否定义 Schema。

- RDD 的数据未定义 Schema（也就是未定义字段名及数据类型）。使用上必须有 Map/Reduce 的概念，需要高级别的程序设计能力。但是功能也最强，能完成所有 Spark 功能。
- Spark DataFrame 建立时必须定义 Schema（定义每一个字段名与数据类型）
- Spark SQL 是由 DataFrame 衍生出来的，必须先建立 DataFrame，然后通过登录 Spark SQL temp table，就可以使用 Spark SQL 语法了。

易使用度：Spark SQL>DataFrame>RDD。

## 三、实验环境

Linux Ubuntu 16.04

jdk-1u84-linux-x64

spark-2.4.7-bin-hadoop2.6

python 3.0

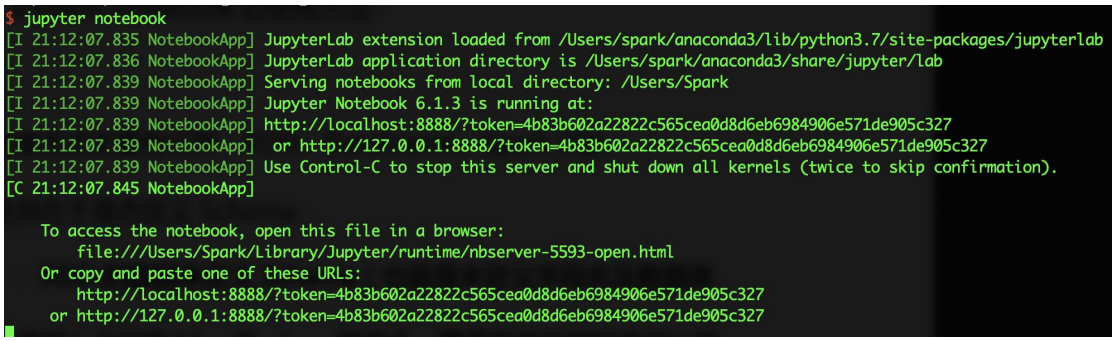
Anaconda 3.0

## 四、实验内容

使用 Spark SQL 产生 DataFrame，通过 DataFrame 创建临时表，通过 SQL 语句对表进行查询数据的操作。

## 五、实验步骤

1. 进入安装配置好的 python 环境，在命令行模式启动 jupyter notebook。




2. 在开启的网页中，新建一个 notebook。


3. 查看内建的 SparkContext 对象 sc 和 SparkSession 对像 spark 是否正常。

```
In [11]: 1 sc
Out[11]: SparkContext

         Spark UI
         Version
         v3.0.1
         Master
         local[*]
         AppName
         pyspark-shell

In [12]: 1 spark
Out[12]: SparkSession - hive
         SparkContext

         Spark UI
         Version
         v3.0.1
         Master
         local[*]
         AppName
         pyspark-shell
```

附：版本号不一定要一样，这里的 spark 版本是 3.0.1

4. 从本地数据集合创建 DataFrame

```
list1=[("Alice",18),("Jom",19)]
df1=spark.createDataFrame(list1)
df2=spark.createDataFrame(list1,['name','age'])
df1.show()
df1.show()
```

```
1 list1=[("Alice",18),("Jom",19)]
2 df1=spark.createDataFrame(list1)
3 df2=spark.createDataFrame(list1,['name','age'])
4 df1.show()
5 df2.show()
```

```
+-----+---+
|   _1| _2|
+-----+---+
|Alice| 18|
|  Jom| 19|
+-----+---+

+-----+---+
| name|age|
+-----+---+
|Alice| 18|
|  Jom| 19|
+-----+---+
```

5. 通过读文件创建 DataFrame

1) 下载或拷贝数据文件“goods_visit.json”到/data 目录下。

2) 读该文件生成 DataFrame。

```
df=spark.read.json("/data/goods_visit.json")
```

3) 显示 goods_visit.json 中的所有数据。

df.show()

```
In [14]: 1 df=spark.read.json("/data/goods_visit.json")

In [15]: 1 df.show()
```

```
+---------+--------+
|click_num|goods_id|
+---------+--------+
|      118| 1010090|
|     2387| 1010091|
|        0| 1010092|
|       87| 1010093|
|        0| 1010094|
|        0| 1010095|
|      126| 1010096|
|     1383| 1010097|
|        0| 1010098|
|      161| 1010099|
|      438| 1010100|
+---------+--------+
```

4) 查看 goods_visit.json 的表结构。

df.printSchema()

```
In [16]: 1 df.printSchema()
```

```
root
 |-- click_num: long (nullable = true)
 |-- goods_id: string (nullable = true)
```

5) 只查看商品 ID(goods_id)。

```
In [17]: 1 df.select("goods_id").show()
```

```
+--------+
|goods_id|
+--------+
| 1010090|
| 1010091|
| 1010092|
| 1010093|
| 1010094|
| 1010095|
| 1010096|
| 1010097|
| 1010098|
| 1010099|
| 1010100|
+--------+
```

6) 统计文件行数。

```
In [18]: 1 df.count()
Out[18]: 11
```

7) 条件查询,查询点击次数超过 500 商品。(show 是返回字段和表数据,collect 是返回集合)

```
1 df.filter(df.click_num > 500).show()
```

```
+---------+--------+
|click_num|goods_id|
+---------+--------+
|     2387| 1010091|
|     1383| 1010097|
+---------+--------+
```

8) 统计点击次数的最值、总和及平均数。

```
from pyspark.sql import functions as F
df1=df.agg(F.max(df["click_num"]),F.sum(df["click_num"]),F.min(df["click_num"]),F.avg(df["click_num"]))
df1.show()
```

```
+--------------+--------------+--------------+------------------+
|max(click_num)|sum(click_num)|min(click_num)|    avg(click_num)|
+--------------+--------------+--------------+------------------+
|          2387|          4700|             0|427.27272727272725|
+--------------+--------------+--------------+------------------+
```

9) 按点击次数进行分组统计。

```
df.groupBy(df.click_num).count().show()
```

```
+---------+-----+
|click_num|count|
+---------+-----+
|        0|    4|
|      126|    1|
|      161|    1|
|       87|    1|
|      118|    1|
|      438|    1|
|     1383|    1|
|     2387|    1|
+---------+-----+
```

10) 按点击次数进行描述性统计。

```
df.describe('click_num').show()
```

```
+-------+------------------+
|summary|         click_num|
+-------+------------------+
|  count|                11|
|   mean|427.27272727272725|
| stddev| 765.3371924725848|
|    min|                 0|
|    max|              2387|
+-------+------------------+
```

11) 按点击次数进行降序排序。

```
df.sort("click_num",ascending=False).show()
```

```
+---------+--------+
|click_num|goods_id|
+---------+--------+
|     2387| 1010091|
|     1383| 1010097|
|      438| 1010100|
|      161| 1010099|
|      126| 1010096|
|      118| 1010090|
|       87| 1010093|
|        0| 1010092|
|        0| 1010094|
|        0| 1010095|
|        0| 1010098|
+---------+--------+
```

12) 将点击次数都加 10，然后输出点击次数与商品 id 的前 10 行数据。

```
df1=df.select(df.goods_id,(df.click_num+10).alias("click_num1"))
df1.show(10)
```

```
+--------+----------+
|goods_id|click_num1|
+--------+----------+
| 1010090|       128|
| 1010091|      2397|
| 1010092|        10|
| 1010093|        97|
| 1010094|        10|
| 1010095|        10|
| 1010096|       136|
| 1010097|      1393|
| 1010098|        10|
| 1010099|       171|
+--------+----------+
only showing top 10 rows
```

6. 用 load 方法读取 goods_visit.json 文件，显示信息，并保存为 parquet 格式。

```
df = spark.read.format("json").load("goods_visit.json")
df.show()
df.select("goods_id",
"click_num").write.format("parquet").save("goods_visit.parquet")
```

```
df = spark.read.format("json").load("/data/goods_visit.json")
df.show()
df.select("goods_id", "click_num").write.format("parquet").save("goods_visit.parquet")
```

```
+---------+--------+
|click_num|goods_id|
+---------+--------+
|      118| 1010090|
|     2387| 1010091|
|        0| 1010092|
|       87| 1010093|
|        0| 1010094|
|        0| 1010095|
|      126| 1010096|
|     1383| 1010097|
|        0| 1010098|
|      161| 1010099|
|      438| 1010100|
+---------+--------+
```

7. 从 RDD 创建 DataFrame 方法

1）

```
li=[("Alice",18),("Jom",19)]
rdd=sc.parallelize(li)
df1 = spark.createDataFrame(rdd) # 不给字段名
df1 = spark.createDataFrame(rdd,['name','age']) # 给字段名
df1.show()
```

```
li=[("Alice",18),("Jom",19)]
rdd=sc.parallelize(li)
df1 = spark.createDataFrame(rdd) # 不给字段名
df1 = spark.createDataFrame(rdd,['name','age']) # 给字段名
df1.show()
```

```
+-----+---+
| name|age|
+-----+---+
|Alice| 18|
|  Jom| 19|
+-----+---+
```

2)

```
from pyspark.sql.types import *
schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)])
df3 = spark.createDataFrame(rdd, schema)
df3.show()
```

```
from pyspark.sql.types import *
schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)])
df3 = spark.createDataFrame(rdd, schema)
df3.show()
```

```
+-----+---+
| name|age|
+-----+---+
|Alice| 18|
|  Jom| 19|
+-----+---+
```

8. 将 DataFrame 转换为 view 或 table。

   createOrReplaceTempView(tableName)方法或

   registerTempTable(tableName)方法。

```
df3.createOrReplaceTempView("table1")#或
df3.registerTempTable("table1")
```

9. 使用 sql 查看表的全部信息，查看表的行数。

```
df=spark.sql("select * from table1")# 可以使用 SQL 语句对表进行操作
cn=spark.sql("select count(*) as cn from table1").collect()
df.show()
print(cn[0].cn)
```

```
df=spark.sql("select * from table1")
cn=spark.sql("select count(*) as cn from table1").collect()
df.show()
print(cn[0].cn)
```

```
+-----+---+
| name|age|
+-----+---+
|Alice| 18|
|  Jom| 19|
+-----+---+

2
```

10. 删除表 dropTempTable(tableName)

```
sqlContext.dropTempTable ("table1") #或
spark.catalog.dropTempView("table1")
```

## 六、实验报告要求

（1）请在 jupyter nootbook 编程环境下完成实验指导文档中的测试代码，仔细观察结果并理解掌握基本的 DataFrame 操作，并最终将编辑生成的 jupyter notebook 文件（后缀为.ipynb 的文件）以附件的形式提交。

（2）请回答问题：Spark SQL 是什么？为什么需要 Spark SQL？Spark 创建 DataFrame 常见的方式有哪些？