

第一、二次上机实验内容

本次实验需要完成虚拟 linux 系统的安装，并完成基本实验环境的配置。

1、安装虚拟机：

下载 VMWare，安装 ubuntu16.04，完成虚拟系统基本设置，学习本地与虚拟机之间文件互传的方式。此部分内容自行参照网上教程。

2、创建用户：

如果你安装 Ubuntu 的时候不是用的 "hadoop" 用户，那么需要增加一个名为 hadoop 的用户：

首先按 **ctrl+alt+t** 打开终端窗口，输入如下命令创建新用户：

```
1. sudo useradd -m hadoop -s /bin/bash
```

这条命令创建了可以登陆的 hadoop 用户，并使用 /bin/bash 作为 shell。

sudo 命令：本文中会大量使用到 sudo 命令。sudo 是 ubuntu 中一种权限管理机制，管理员可以授权给一些普通用户去执行一些需

要 `root` 权限执行的操作。当使用 `sudo` 命令时，就需要输入您当前用户的密码。

密码:在 Linux 的终端中输入密码，终端是不会显示任何你当前输入的密码，也不会提示你已经输入了多少字符密码。而在 windows 系统中,输入密码一般都会以“`*`”表示你输入的密码字符。

接着使用如下命令设置密码，可简单设置为 `hadoop`，按提示输入两次密码：

```
1. sudo passwd hadoop
```

可为 `hadoop` 用户增加管理员权限，方便部署，避免一些对新手来说比较棘手的权限问题：

```
1. sudo adduser hadoop sudo
```

最后注销当前用户（点击屏幕右上角的齿轮，选择注销），返回登陆界面。在登陆界面中选择刚创建的 `hadoop` 用户进行登陆。

3、更新 apt

用 `hadoop` 用户登录后，我们先更新一下 `apt`，后续我们使用 `apt` 安装软件，如果没更新可能有一些软件安装不了。按 `ctrl+alt+t` 打开终端窗口，执行如下命令：

1. sudo apt-get update

若出现如下 "Hash 校验和不符" 的提示，可通过更改软件源来解决。若没有该问题，则不需要更改。从软件源下载某些软件的过程中，可能由于网络方面的原因出现没法下载的情况，那么建议更改软件源。在学习 Hadoop 过程中，即使出现"Hash 校验和不符"的提示，也不会影响 Hadoop 的安装。

```
W: 无法下载 bzip2:/var/lib/apt/lists/partial/cn.archive.ubuntu.com_ubuntu_dists_trusty-updates_universe_l18n_Translation-en Hash 校验和不符  
W: 无法下载 bzip2:/var/lib/apt/lists/partial/cn.archive.ubuntu.com_ubuntu_dists_trusty-backports_universe_binary-amd64_Packages Hash 校验和不符
```

首先点击左侧任务栏的【系统设置】（齿轮图标），选择【软件和更新】



点击“下载自”右侧的方框，选择【其他节点】



在列表中选中【mirrors.aliyun.com】，并点击右下角的【选择服务器】，会要求输入用户密码，输入即可。



接着点击关闭。



此时会提示列表信息过时，点击【重新载入】，



最后耐心等待更新缓存即可。更新完成会自动关闭【软件和更新】这个窗口。如果还是提示错误，请选择其他服务器节点如

`mirrors.163.com` 再次进行尝试。更新成功后，再次执行 `sudo apt-get update` 就正常了。

后续需要更改一些配置文件，我比较喜欢用的是 `vim`（`vi` 增强版，基本用法相同），建议安装一下（如果你实在还不会用 `vi/vim` 的，请将后面用到 `vim` 的地方改为 `gedit`，这样可以使用文本编辑器进行修改，并且每次文件更改完成后请关闭整个 `gedit` 程序，否则会占用终端）：

```
1. sudo apt-get install vim
```

安装软件时若需要确认，在提示处输入 `y` 即可。

`vim` 的常用模式有分为命令模式，插入模式，可视模式，正常模式。本教程中，只需要用到正常模式和插入模式。二者间的切换即可以帮助你完成本指南的学习。

1. 正常模式

正常模式主要用来浏览文本内容。一开始打开 `vim` 都是正常模式。在任何模式下按下 `Esc` 键就可以返回正常模式

2. 插入编辑模式

插入编辑模式则用来向文本中添加内容的。在正常模式下，输入 `i` 键即可进入插入编辑模式

3. 退出 `vim`

如果有利用 `vim` 修改任何的文本，一定要记得保存。`Esc` 键退回到正常模式中，然后输入`:wq` 即可保存文本并退出 `vim`

4、安装 SSH、配置 SSH 无密码登陆：

集群、单节点模式都需要用到 **SSH** 登陆（类似于远程登陆，你可以登录某台 **Linux** 主机，并且在上面运行命令），**Ubuntu** 默认已安装了 **SSH client**，此外还需要安装 **SSH server**：

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

安装后，可以使用如下命令登陆本机：

1. ssh localhost

此时会有如下提示(SSH 首次登陆提示), 输入 yes 。然后按提示输入密码 hadoop, 这样就登陆到本机了。

```
hadoop@DBLab-XMU:~$ ssh localhost
The authenticity of host 'localhost (127.0.0.1)' can't be established.
ECDSA key fingerprint is a9:28:e0:4e:89:40:a4:cd:75:8f:0b:8b:57:79:67:86.
Are you sure you want to continue connecting (yes/no)? yes
```

但这样登陆是需要每次输入密码的，我们需要配置成 SSH 无密码登陆比较方便。

首先退出刚才的 ssh，就回到了我们原先的终端窗口，然后利用 ssh-keygen 生成密钥，并将密钥加入到授权中：

```
3. ssh-keygen -t rsa          # 会有提示，都按回车  
就可以  
4. cat ./id_rsa.pub >> ./authorized_keys # 加入授权
```

~的含义：在 Linux 系统中，~ 代表的是用户的主文件夹，即 "/home/用户名" 这个目录，如你的用户名为 hadoop，则 ~ 就代表 "/home/hadoop/"。

此时再用 `ssh localhost` 命令，无需输入密码就可以直接登陆了，如下图。

5、安装 Java 环境：

按照下面步骤来自己手动安装 JDK1.8。安装包：`jdk-8u162-linux-x64.tar.gz`

在 Linux 命令行界面中，执行如下 Shell 命令（注意：当前登录用户名是 hadoop）：

```
1. cd /usr/lib  
2. sudo mkdir jvm # 创建 /usr/lib/jvm 目录用来存放 JDK 文  
件  
3. cd ~ # 进入 hadoop 用户的主目录
```

```
4. cd Downloads #注意区分大小写字母，刚才已经通过FTP  
软件把JDK安装包 jdk-8u162-linux-x64.tar.gz 上传到该  
目录下  
5. sudo tar -zxvf ./jdk-8u162-linux-x64.tar.gz -C /u  
sr/lib/jvm #把JDK文件解压到/usr/lib/jvm目录下
```

JDK文件解压缩以后，可以执行如下命令到/usr/lib/jvm目录查看一下：

```
1. cd /usr/lib/jvm  
2. ls
```

可以看到，在/usr/lib/jvm目录下有个 jdk1.8.0_162 目录。

下面继续执行如下命令，设置环境变量：

```
1. cd ~  
2. vim ~/.bashrc
```

上面命令使用 vim 编辑器打开了 hadoop 这个用户的环境变量配置文件，请在这个文件的开头位置，添加如下几行内容：

```
export JAVA_HOME=/usr/lib/jvm/jdk1.8.0_162  
export JRE_HOME=${JAVA_HOME}/jre
```

```
export CLASSPATH=.:${JAVA_HOME}/lib:${JRE_HOME}/lib  
  
export PATH=${JAVA_HOME}/bin:$PATH
```

保存.bashrc 文件并退出 vim 编辑器。然后，继续执行如下命令让.bashrc 文件的配置立即生效：

```
1. source ~/.bashrc
```

这时，可以使用如下命令查看是否安装成功：

```
1. java -version
```

如果能够在屏幕上返回如下信息，则说明安装成功：

```
hadoop@ubuntu:~$ java -version  
  
java version "1.8.0_162"  
  
Java(TM) SE Runtime Environment (build 1.8.0_162-b12)  
  
Java HotSpot(TM) 64-Bit Server VM (build 25.162-b12, mixed mode)
```

至此，就成功安装了 Java 环境。

第三次上机实验内容

本次实验将完成 Hadoop 和 spark 的安装，并运行一个简单的 spark 程序。

1、安装 Hadoop 2

Hadoop 安装文件 hadoop-2.7.1.tar.gz

我们选择将 Hadoop 安装至 /usr/local/ 中：

```
1. sudo tar -zxf ~/下载/hadoop-2.7.1.tar.gz -C /usr/local      # 解压到/usr/local 中  
2. cd /usr/local/  
3. sudo mv ./hadoop-2.7.1/ ./hadoop                         # 将文件夹名改为hadoop  
4. sudo chown -R hadoop ./hadoop                            # 修改文件权限
```

Hadoop 解压后即可使用。输入如下命令来检查 Hadoop 是否可用，成功则会显示 Hadoop 版本信息：

```
1. cd /usr/local/hadoop  
2. ./bin/hadoop version
```

2、测试 Hadoop

现在我们可以执行例子来感受下 Hadoop 的运行。Hadoop 附带了丰富的例子（运行 `./bin/hadoop jar ./share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.1.jar` 可以看到所有例子），包括 `wordcount`、`terasort`、`join`、`grep` 等。

在此我们选择运行 `grep` 例子，我们将 `input` 文件夹中的所有文件作为输入，筛选当中符合正则表达式 `dfs[a-z.]+` 的单词并统计出现的次数，最后输出结果到 `output` 文件夹中。

```
1. cd /usr/local/hadoop  
2. mkdir ./input  
3. cp ./etc/hadoop/*.xml ./input    # 将配置文件作为输入文件  
4. ./bin/hadoop jar ./share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar grep ./input ./output 'dfs[a-z.]+'  
5. cat ./output/*      # 查看运行结果
```

执行成功后如下所示，输出了作业的相关信息，输出的结果是符合正则的单词 `dfsadmin` 出现了 1 次

IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
File Input Format Counters
Bytes Read=123
File Output Format Counters
Bytes Written=23

程序执行成功的输出信息

hadoop@DBLab-XMU:/usr/local/hadoop\$ cat ./output/*
1 dfsadmin

程序的执行结果

hadoop@DBLab-XMU:/usr/local/hadoop\$

注意，Hadoop 默认不会覆盖结果文件，因此再次运行上面实例会提示出错，需要先将 `./output` 删除。

```
1. rm -r ./output
```

3、安装 Spark

Spark 可以独立安装使用，也可以和 Hadoop 一起安装使用。本教程中，我们采用和 Hadoop 一起安装使用，这样，就可以让 Spark 使用 HDFS 存取数据。

我们选择 spark-2.4.7-bin-without-hadoop.tgz 版本，并且假设当前使用用户名 hadoop 登录了 Linux 操作系统。

```
1. sudo tar -zxf ~/下载/spark-2.4.7-bin-without-hadoop.tgz  
-C /usr/local/  
2. cd /usr/local  
3. sudo mv ./spark-2.1.0-bin-without-hadoop/ ./spark
```

```
4. sudo chown -R hadoop:hadoop ./spark          # 此  
处的 hadoop 为你的用户名
```

安装后，还需要修改 Spark 的配置文件 spark-env.sh

```
1. cd /usr/local/spark  
2. cp ./conf/spark-env.sh.template ./conf/spark-env.  
sh
```

编辑 spark-env.sh 文件(vim ./conf/spark-env.sh)，在第一行添加以下配置信息：

```
export SPARK_DIST_CLASSPATH=$( /usr/local/hadoop/bin/ha  
doop classpath )
```

有了上面的配置信息以后，Spark 就可以把数据存储到 Hadoop 分布式文件系统 HDFS 中，也可以从 HDFS 中读取数据。如果没有配置上面信息，Spark 就只能读写本地数据，无法读写 HDFS 数据。

然后通过如下命令，修改环境变量

```
1. vim ~/.bashrc
```

在.bashrc 文件中添加如下内容

```
export JAVA_HOME=/usr/lib/jvm/default-java

export HADOOP_HOME=/usr/local/hadoop

export SPARK_HOME=/usr/local/spark

export PYTHONPATH=$SPARK_HOME/python:$SPARK_HOME/python/lib/py4j-0.10.4-src.zip:$PYTHONPATH

export PYSPARK_PYTHON=python3

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

PYTHONPATH 环境变量主要是为了在 Python3 中引入 pyspark 库，PYSPARK_PYTHON 变量主要是设置 pyspark 运行的 python 版本。

.bashrc 中必须包含：JAVA_HOME, HADOOP_HOME, SPARK_HOME, PYTHONPATH, PYSPARK_PYTHON, PATH 这些环境变量。如果已经设置了这些变量则不需要重新添加设置。另外需要注意，上面的配置项中，PYTHONPATH 这一行有个 py4j-0.10.4-src.zip，这个 zip 文件的版本号一定要和“/usr/local/spark/python/lib”目录下的 py4j-0.10.4-src.zip 文件保持版本一致。比如，如果“/usr/local/spark/python/lib”目录下是 py4j-0.10.7-src.zip，那

么，`PYTHONPATH` 这一行后面也要写 `py4j-0.10.7-src.zip`，从而使二者版本一致。

接着还需要让该环境变量生效，执行如下代码：

```
1. source ~/.bashrc
```

配置完成后就可以直接使用，不需要像 Hadoop 运行启动命令。

通过运行 Spark 自带的示例，验证 Spark 是否安装成功。

```
1. cd /usr/local/spark  
2. bin/run-example SparkPi
```

执行时会输出非常多的运行信息，输出结果不容易找到，可以通过 `grep` 命令进行过滤：

```
1. bin/run-example SparkPi 2>&1 | grep "Pi is"
```

过滤后的运行结果如下图示，可以得到 π 的 5 位小数近似值：



```
[hadoop@dblab spark]$ ./bin/run-example SparkPi 2>&1 | grep "Pi is roughly"  
Pi is roughly 3.14588  
[hadoop@dblab spark]$
```

4、在 `pyspark` 中运行代码

学习 Spark 程序开发，建议首先通过 `pyspark` 交互式学习，加深 Spark 程序开发的理解。

这里介绍 `pyspark` 的基本使用。`pyspark` 提供了简单的方式来学习 API，并且提供了交互的方式来分析数据。你可以输入一条语句，`pyspark` 会立即执行语句并返回结果，这就是我们所说的 REPL（Read-Eval-Print Loop，交互式解释器），为我们提供了交互式执行环境，表达式计算完成就会输出结果，而不必等到整个程序运行完毕，因此可即时查看中间结果，并对程序进行修改，这样可以在很大程度上提升开发效率。

前面已经安装了 Hadoop 和 Spark，如果 Spark 不使用 HDFS 和 YARN，那么就不用启动 Hadoop 也可以正常使用 Spark。如果在使用 Spark 的过程中需要用到 HDFS，就要首先启动 Hadoop（启动 Hadoop 的方法可以参考上面给出的 Hadoop 安装教程）。这里假设不需要用到 HDFS，因此，就没有启动 Hadoop。现在我们直接开始使用 Spark。

注意：如果按照上面的安装步骤，已经设置了 `PYSPARK_PYTHON` 环境变量，那么你直接使用如下命令启动 `pyspark` 即可。

```
1. bin/pyspark
```

如果没有设置 `PYSPARK_PYTHON` 环境变量，则使用如下命令启动 `pyspark`

```
1. PYSPARK_PYTHON=python3
```

```
2. ./bin/pyspark
```

pyspark 命令及其常用的参数如下：

```
./bin/pyspark --master <master-url>
```

Spark 的运行模式取决于传递给 `SparkContext` 的 Master URL 的值。Master URL 可以是以下任一种形式：

- * local 使用一个 Worker 线程本地化运行 SPARK(完全不并行)
- * local[*] 使用逻辑 CPU 个数数量的线程来本地化运行 Spark
- * local[K] 使用 K 个 Worker 线程本地化运行 Spark (理想情况下，K 应该根据运行机器的 CPU 核数设定)
- * spark://HOST:PORT 连接到指定的 Spark standalone master。默认端口是 7077.
- * yarn-client 以客户端模式连接 YARN 集群。集群的位置可以在 HADOOP_CONF_DIR 环境变量中找到。
- * yarn-cluster 以集群模式连接 YARN 集群。集群的位置可以在 HADOOP_CONF_DIR 环境变量中找到。
- * mesos://HOST:PORT 连接到指定的 Mesos 集群。默认接口是 5050。

需要强调的是，这里我们采用“本地模式”（local）运行 Spark。

在 Spark 中采用本地模式启动 `pyspark` 的命令主要包含以下参数：

`--master`: 这个参数表示当前的 `pyspark` 要连接到哪个 master, 如果是 `local[*]`, 就是使用本地模式启动 `pyspark`, 其中, 中括号内的星号表示需要使用几个 CPU 核心(core);

`--jars`: 这个参数用于把相关的 JAR 包添加到 CLASSPATH 中; 如果有多个 jar 包, 可以使用逗号分隔符连接它们;

比如, 要采用本地模式, 在 4 个 CPU 核心上运行 `pyspark`:

```
1. cd /usr/local/spark  
2. ./bin/pyspark --master local[4]
```

或者, 可以在 CLASSPATH 中添加 code.jar, 命令如下:

```
cd /usr/local/spark  
  
../bin/pyspark --master local[4] --jars code.jar
```

可以执行“`pyspark --help`”命令, 获取完整的选项列表, 具体如下:

```
cd /usr/local/spark  
  
../bin/pyspark --help
```

上面是命令使用方法介绍，下面正式使用命令进入 pyspark 环境，可以通过下面命令启动 pyspark 环境：

```
bin/pyspark
```

该命令省略了参数，这时，系统默认是“`bin/pyspark--master local[*]`”，也就是说，是采用本地模式运行，并且使用本地所有的 CPU 核心。

启动 `pyspark` 后，就会进入“`>>>`”命令提示符状态，如下图所示：



```
Python 3.5.2 (default, Nov 17 2016 17:05:23)
[Pytho
Using Python version 3.5.2 (default, Nov 17 2016 17:05:23)
SparkSession available as 'spark'.
>>> 
```

现在，你就可以在里面输入 `python` 代码进行调试了。

比如，下面在命令提示符后面输入一个表达式“`8 * 2 + 5`”，然后回车，就会立即得到结果：

```
1. >>> 8 * 2 + 5
```

最后，可以使用命令“`exit()`”退出 `pyspark`，如下所示：

```
1. >>> exit()
```

或者，也可以直接使用“**Ctrl+D**”组合键，退出 pyspark。

第四次上机实验内容

本次实验我们将通过实验深入了解 HDFS 文件存储系统的结构

并通过一个词频统计的程序了解 spark 读取 HDFS 文件的方式。

1、Spark 独立应用程序编程

接着我们通过一个简单的应用程序来演示如何通过 Spark API 编写一个独立应用程序。

在进行 Python 编程前, 请先确定是否已经.bashrc 中添加 PYTHONPATH 环境变量。接下来即可进行 Python 编程.

这里在新建一个 test.py 文件,并在 test.py 添加代码

```
1. cd ~  
2. vim test.py
```

在 test.py 中添加如下代码,:;

```
1. from pyspark import SparkContext  
2. sc = SparkContext( 'local' , 'test' )  
3. logFile = "file:///usr/local/spark/README.md"  
4. logData = sc.textFile(logFile, 2).cache()  
5. numAs = logData.filter(lambda line: 'a' in line).  
count()
```

```
6. numBs = logData.filter(lambda line: 'b' in line).  
    count()  
  
7. print('Lines with a: %s, Lines with b: %s' % (numA  
    s, numBs))
```

保存代码后，通过如下命令执行：

```
1. python3 ~/test.py
```

执行结果如下图：

```
dblab@dblabc-Lenovo:/usr/local/spark/bin$ touch ~/test.py  
dblab@dblabc-Lenovo:/usr/local/spark/bin$ vim ~/test.py  
dblab@dblabc-Lenovo:/usr/local/spark/bin$ python3 ~/test.py  
Setting default log level to "WARN".  
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setL  
17/12/03 21:06:01 WARN util.NativeCodeLoader: Unable to load native-hadoop  
sing builtin-java classes where applicable  
17/12/03 21:06:01 WARN util.Utils: Your hostname, dblab-Lenovo resolves to  
using 192.168.1.121 instead (on interface en0)  
17/12/03 21:06:01 WARN util.Utils: Set SPARK_LOCAL_IP if you need to bind  
17/12/03 21:06:02 WARN util.Utils: Service 'SparkUI' could not bind on port  
Lines with a: 61, Lines with b: 30
```

最终得到的结果如下：

```
Lines with a: 62, Lines with b: 30
```

自此，你就完成了你的第一个 Spark 程序了。

2、第一个 Spark 应用程序：WordCount:

任务：编写 Spark 应用程序，对某文件中的单词进行词频统计。

(1) 准备工作:

请进入 Linux 系统，打开“终端”，进入 Shell 命令提示符状态，然后，执行如下命令新建目录：

```
1. cd /usr/local/spark  
2. mkdir mycode  
3. cd mycode  
4. mkdir wordcount  
5. cd wordcount
```

然后，在“/usr/local/spark/mycode/wordcount”目录下新建一个包含了一些语句的文本文件 word.txt，命令如下：

```
1. vim word.txt
```

你可以在文本文件中随意输入一些单词，用空格隔开，我们会编写 Spark 程序对该文件进行单词词频统计。然后，按键盘 Esc 键退出 vim 编辑状态，输入“:wq”保存文件并退出 vim 编辑器。

(2) 启动 pyspark

首先，请登录 Linux 系统(要注意记住登录采用的用户名，本教程统一采用 hadoop 用户名进行登录)，打开“终端”进入 shell 命令提示符状态，然后执行以下命令进入 pyspark：

```
1. cd /usr/local/spark  
2. ./bin/pyspark  
3. ....//这里省略启动过程显示的一大堆信息  
4. >>>
```

启动进入 `pyspark` 需要一点时间，在进入 `pyspark` 后，我们可能还需要到 `Linux` 文件系统中对相关目录下的文件进行编辑和操作（比如要查看 `spark` 程序执行过程生成的文件），这个无法在 `pyspark` 中完成，因此，这里再打开第二个终端，用来在 `Linux` 系统的 `Shell` 命令提示符下操作。

(3) 加载本地文件

在开始具体词频统计代码之前，需要解决一个问题，就是如何加载文件？

要注意，文件可能位于本地文件系统中，也有可能存放在分布式文件系统 `HDFS` 中，所以，下面我们分别介绍如何加载本地文件，以及如何加载 `HDFS` 中的文件。

首先，请在第二个终端窗口下操作，用下面命令到达 “`/usr/local/spark/mycode/wordcount`” 目录，查看一下上面已经建好的 `word.txt` 的内容：

```
1. cd /usr/local/spark/mycode/wordcount
```

```
2. cat word.txt
```

cat 命令会把 word.txt 文件的内容全部显示到屏幕上。

现在让我们切换回到第一个终端，也就是 pyspark，然后输入下面命令：

```
1. >>> textFile = sc.textFile('file:///usr/local/spark/mycode/wordcount/word.txt')
```

上面代码中，`sc.textFile()`中的这个 `textFile` 是 `sc` 的一个方法名称，这个方法用来加载文件数据。这两个 `textFile` 不是一个东西，不要混淆。实际上，前面的变量 `textFile`，你完全可以换个变量名称，比如，`lines = sc.textFile("file:///usr/local/spark/mycode/wordcount/word.txt")`。这里使用相同名称，就是有意强调二者的区别。

注意，要加载本地文件，必须采用“`file:///`”开头的这种格式。执行上面这条命令以后，并不会马上显示结果，因为，Spark 采用惰性机制，只有遇到“行动”类型的操作，才会从头到尾执行所有操作。所以，下面我们执行一条“行动”类型的语句，就可以看到结果：

```
1. >>> textFile.first()
```

`first()`是一个“行动”（Action）类型的操作，会启动真正的计算过程，从文件中加载数据到变量 `textFile` 中，并取出第一行文本。屏幕上会显示很多反馈信息，这里不再给出，你可以从这些结果信息中，找到 `word.txt` 文件中的第一行的内容。

正因为 Spark 采用了惰性机制，在执行转换操作的时候，即使我们输入了错误的语句，`pyspark` 也不会马上报错，而是等到执行“行动”类型的语句时启动真正的计算，那个时候“转换”操作语句中的错误就会显示出来，比如：

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

上面我们使用了一个根本就不存在的 `word123.txt`，执行上面语句时，`pyspark` 根本不会报错，因为，没有遇到“行动”类型的 `first()` 操作之前，这个加载操作时不会真正执行的。然后，我们执行一个“行动”类型的操作 `first()`，如下：

```
1. >>> textFile.first()
```

执行上面语句后，你会发现，会返回错误信息，显示“拒绝连接”。因为，这个 `word123.txt` 文件根本就不存在。

好了，现在我们可以练习一下如何把 `textFile` 变量中的内容再次写回到另外一个目录 `wordback` 中：

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

上面的 `saveAsTextFile()` 括号里面的参数是保存文件的路径，不是文件名。`saveAsTextFile()` 是一个“行动”（Action）类型的操作，所以，马上会执行真正的计算过程，从 `word.txt` 中加载数据到变量 `textFile` 中，然后，又把 `textFile` 中的数据写回到本地文件目录 “`/usr/local/spark/mycode/wordcount/writeback/`”下面，现在让我们切换到 Linux Shell 命令提示符窗口中，执行下面命令：

```
1. cd /usr/local/spark/mycode/wordcount/writeback/  
2. ls
```

执行结果类似下面：

```
part-00000 _SUCCESS
```

也就是说，该目录下包含两个文件，我们可以使用 `cat` 命令查看一下 `part-00000` 文件（注意：`part-`后面是五个零）

```
1. cat part-00000
```

显示结果，是和上面 word.txt 中的内容一样的。

(4) 加载 HDFS 文件

为了能够读取 HDFS 中的文件，请首先启动 Hadoop 中的 HDFS 组件。注意，之前我们在“Spark 安装”这章内容已经介绍了如何安装 Hadoop 和 Spark，所以，这里我们可以使用以下命令直接启动 Hadoop 中的 HDFS 组件（由于用不到 MapReduce 组件，所以，不需要启动 MapReduce 或者 YARN）。请到第二个终端窗口，使用 Linux Shell 命令提示符状态，然后输入下面命令：

```
1. cd /usr/local/hadoop  
2. ./sbin/start-dfs.sh
```

启动结束后，HDFS 开始进入可用状态。如果你在 HDFS 文件系统中，还没有为当前 Linux 登录用户创建目录(本教程统一使用用户名 hadoop 登录 Linux 系统)，请使用下面命令创建：

```
1. ./bin/hdfs dfs -mkdir -p /user/hadoop
```

也就是说，HDFS 文件系统为 Linux 登录用户开辟的默认目录是“/user/用户名”（注意：是 user，不是 usr），本教程统一使用用户名 hadoop 登录 Linux 系统，所以，上面创建了“/user/hadoop”目录，再次强调，这个目录是在 HDFS 文件系统中，不在本地文件系

统中。创建好以后，下面我们使用命令查看一下 HDFS 文件系统中的目录和文件：

```
1. ./bin/hdfs dfs -ls .
```

上面命令中，最后一个点号“.”，表示要查看 Linux 当前登录用户 hadoop 在 HDFS 文件系统中与 hadoop 对应的目录下的文件，也就是查看 HDFS 文件系统中“/user/hadoop/”目录下的文件，所以，下面两条命令是等价的：

```
1. ./bin/hdfs dfs -ls .
2. ./bin/hdfs dfs -ls /user/hadoop
```

如果要查看 HDFS 文件系统根目录下的内容，需要使用下面命令：

```
1. ./bin/hdfs dfs -ls /
```

下面，我们把本地文件系统中的“/usr/local/spark/mycode/wordcount/word.txt”上传到分布式文件系统 HDFS 中（放到 hadoop 用户目录下）：

```
1. ./bin/hdfs dfs -put /usr/local/spark/mycode/wordcount/word.txt .
```

然后，用命令查看一下 HDFS 的 hadoop 用户目录下是否多了 word.txt 文件，可以使用下面命令列出 hadoop 目录下的内容：

```
1. ./bin/hdfs dfs -ls .
```

可以看到，确实多了一个 word.txt 文件，我们使用 cat 命令查看一个 HDFS 中的 word.txt 文件的内容，命令如下：

```
1. ./bin/hdfs dfs -cat ./word.txt
```

上面命令执行后，就会看到 HDFS 中 word.txt 的内容了。

现在，让我们切换回到 pyspark 窗口，编写语句从 HDFS 中加载 word.txt 文件，并显示第一行文本内容：

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

执行上面语句后，就可以看到 HDFS 文件系统中（不是本地文件系统）的 word.txt 的第一行内容了。

需要注意的是，

sc.textFile("hdfs://localhost:9000/user/hadoop/word.txt")中，“hdfs://localhost:9000/”是前面介绍 Hadoop 安装内容时确定下来的

端口地址 9000。实际上，也可以省略不写，如下三条语句都是等价的：

```
1. >>> textFile = sc.textFile("hdfs://localhost:9000  
/user/hadoop/word.txt")  
  
2. >>> textFile = sc.textFile("/user/hadoop/word.txt  
")  
  
3. >>> textFile = sc.textFile("word.txt")
```

下面，我们再把 `textFile` 的内容写回到 HDFS 文件系统中（写到 `hadoop` 用户目录下）：

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

执行上面命令后，文本内容会被写入到 HDFS 文件系统的 “/user/hadoop/writeback” 目录下，我们可以切换到 Linux Shell 命令提示符窗口查看一下：

```
1. ./bin/hdfs dfs -ls .
```

执行上述命令后，在执行结果中，可以看到有个 `writeback` 目录，下面我们查看该目录下有什么文件：

```
1 ./bin/hdfs dfs -ls ./writeback
```

执行结果中，可以看到存在两个文件：part-00000 和 _SUCCESS。我们使用下面命令输出 part-00000 文件的内容（注意：part-00000 里面有五个零）：

```
1 ./bin/hdfs dfs -cat ./writeback/part-00000
```

执行结果中，就可以看到和 word.txt 文件中一样的文本内容。

(5) 词频统计

有了前面的铺垫性介绍，下面我们就开始第一个 Spark 应用程序：WordCount。请切换到 pyspark 窗口：

```
1. >>> textFile = sc.textFile("file:///usr/local/spark/mycode/wordcount/word.txt")
2. >>> wordCount = textFile.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).reduceByKey(lambda a, b : a + b)
3. >>> wordCount.collect()
```

上面只给出了代码，省略了执行过程中返回的结果信息，因为返回信息很多。

下面简单解释一下上面的语句：

`textFile` 包含了多行文本内容，`textFile.flatMap(lambda line : line.split(" "))`会遍历 `textFile` 中的每行文本内容，当遍历到其中一行文本内容时，会把文本内容赋值给变量 `line`，并执行 Lamda 表达式 `line : line.split(" ")`。

`line : line.split(" ")`是一个 Lamda 表达式，左边表示输入参数，右边表示函数里面执行的处理逻辑，这里执行 `line.split(" ")`，也就是针对 `line` 中的一行文本内容，采用空格作为分隔符进行单词切分，从一行文本切分得到很多个单词构成的单词集合。

这样，对于 `textFile` 中的每行文本，都会使用 Lamda 表达式得到一个单词集合，最终，多行文本，就得到多个单词集合。
`textFile.flatMap()`操作就把这多个单词集合“拍扁”得到一个大的单词集合。

然后，针对这个大的单词集合，执行 `map()`操作，也就是 `map(lambda word : (word, 1))`，这个 `map` 操作会遍历这个集合中的每个单词，当遍历到其中一个单词时，就把当前这个单词赋值给变量 `word`，并执行 Lamda 表达式 `word : (word, 1)`，这个 Lamda 表达式的含义是，`word` 作为函数的输入参数，然后，执行函数处理逻辑，这里会执行(`word, 1`)，也就是针对输入的 `word`，构建得到一个 `tuple`，形式为(`word,1`)，`key` 是 `word`，`value` 是 `1`（表示该单词出现 1 次）。

程序执行到这里，已经得到一个 RDD，这个 RDD 的每个元素是 (key,value) 形式的 tuple。最后，针对这个 RDD，执行 reduceByKey (lambda a, b : a + b) 操作，这个操作会把所有 RDD 元素按照 key 进行分组，然后使用给定的函数（这里就是 Lambda 表达式： a, b : a + b），对具有相同的 key 的多个 value 进行 reduce 操作，返回 reduce 后的 (key,value)，比如 ("hadoop",1) 和 ("hadoop",1)，具有相同的 key，进行 reduce 以后就得到 ("hadoop",2)，这样就计算得到了这个单词的词频。

3、编写独立应用程序执行词频统计

下面我们编写一个 Scala 应用程序来实现词频统计。

请登录 Linux 系统（本教程统一采用用户名 hadoop 进行登录），进入 Shell 命令提示符状态，然后，执行下面命令：

```
1. cd /usr/local/spark/mycode/wordcount/
```

请在“/usr/local/spark/mycode/wordcount/”目录下新建一个 test.py 文件，里面包含如下代码：

```
from pyspark import SparkContext  
  
sc = SparkContext( 'local', 'test' )
```

```
textFile = sc.textFile("file:///usr/local/spark/mycode  
/wordcount/word.txt")  
  
wordCount = textFile.flatMap(lambda line: line.split(" ")).map(lambda word: (word,1)).reduceByKey(lambda a,  
b : a + b)  
  
wordCount.foreach(print)
```

然后执行如下命令：

```
1. python3 ./test.py
```

即可得出结果

下面是笔者的 word.txt 进行词频统计后的结果（你的结果应该和这个类似）：

(Spark,1)

(is,1)

(hadoop,2)

(Spark,1)

第五次上机实验内容

本次实验我们将深入了解 spark 的核心机制—RDD 数据结构。

通过前面几章的介绍，我们已经了解了 Spark 的运行架构和 RDD 设计与运行原理，并介绍了 RDD 操作的两种类型：转换操作和行动操作。

同时，我们前面通过一个简单的 WordCount 实例，也大概介绍了 RDD 的几种简单操作。现在我们介绍更多关于 RDD 编程的内容。

Spark 中针对 RDD 的操作包括创建 RDD、RDD 转换操作和 RDD 行动操作。

1、RDD 创建

RDD 可以通过两种方式创建：

- * 第一种：读取一个外部数据集。比如，从本地文件加载数据集，或者从 HDFS 文件系统、HBase、Cassandra、Amazon S3 等外部数据源中加载数据集。Spark 可以支持文本文件、SequenceFile 文件（Hadoop 提供的 SequenceFile 是一个由二进制序列化过的 key/value 的字节流组成的文本存储文件）和其他符合 Hadoop InputFormat 格式的文件。

* 第二种：调用 `SparkContext` 的 `parallelize` 方法，在 Driver 中一个已经存在的集合（数组）上创建。

创建 RDD 之前的准备工作

在即将进行相关的实践操作之前，我们首先要登录 Linux 系统（本教程统一采用 `hadoop` 用户登录），然后，打开命令行“终端”，请按照下面的命令启动 Hadoop 中的 HDFS 组件：

```
1. cd /usr/local/hadoop  
2. ./sbin/start-dfs.sh
```

然后，我们按照下面命令启动 `pyspark`：

```
1. cd /usr/local/spark  
2. ./bin/pyspark
```

然后，新建第二个“终端”，方法是，在前面已经建设的第一个终端窗口的左上方，点击“终端”菜单，在弹出的子菜单中选择“新建终端”，就可以打开第二个终端窗口，现在，我们切换到第二个终端窗口，在第二个终端窗口中，执行以下命令，进入之前已经创建好的 “`/usr/local/spark/mycode/`” 目录，在这个目录下新建 `rdd` 子目录，用来存放本章的代码和相关文件：

```
1. cd usr/local/spark/mycode/
```

```
2. mkdir rdd
```

然后，使用 `vim` 编辑器，在 `rdd` 目录下新建一个 `word.txt` 文件，
你可以在文件里面随便输入几行英文语句用来测试。

经过上面的准备工作以后，我们就可以开始创建 `RDD` 了。

从文件系统中加载数据创建 `RDD`

`Spark` 采用 `textFile()` 方法来从文件系统中加载数据创建 `RDD`，
该方法把文件的 `URI` 作为参数，这个 `URI` 可以是本地文件系统的地
址，或者是分布式文件系统 `HDFS` 的地址，或者是 `Amazon S3` 的
地址等等。

下面请切换回 `pyspark` 窗口，看一下如何从本地文件系统中加载
数据：

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

下面看一下如何从 `HDFS` 文件系统中加载数据，这个在前面的第一
一个 `Spark` 应用程序：`WordCount` 实例中已经讲过，这里再简单复
习一下。

请根据前面的第一个 `Spark` 应用程序：`WordCount` 实例中的内
容介绍，把刚才在本地文件系统中的“`/usr/local/spark/mycode
/rdd/word.txt`”上传到 `HDFS` 文件系统的 `hadoop` 用户目录下（注

意：本教程统一使用 `hadoop` 用户登录 `Linux` 系统）。然后，在 `pyspark` 窗口中，就可以使用下面任意一条命令完成从 `HDFS` 文件系统中加载数据：

```
1.>>> lines = sc.textFile("hdfs://localhost:9000/us  
er/hadoop/word.txt")  
2.>>> lines = sc.textFile("/user/hadoop/word.txt")  
3.>>> lines = sc.textFile("word.txt")
```

注意，上面三条命令是完全等价的命令，只不过使用了不同的目录形式，你可以使用其中任意一条命令完成数据加载操作。

在使用 `Spark` 读取文件时，需要说明以下几点：

(1) 如果使用了本地文件系统的路径，那么，必须要保证在所有的 `worker` 节点上，也都能够采用相同的路径访问到该文件，比如，可以把该文件拷贝到每个 `worker` 节点上，或者也可以使用网络挂载共享文件系统。

(2) `textFile()`方法的输入参数，可以是文件名，也可以是目录，也可以是压缩文件等。比如，`textFile("/my/directory")`,
`textFile("/my/directory/.txt")`, and `textFile("/my/directory/.gz")`.

(3) `textFile()`方法也可以接受第 2 个输入参数（可选），用来指定分区的数目。默认情况下，`Spark` 会为 `HDFS` 的每个 `block` 创建一个分区（`HDFS` 中每个 `block` 默认是 128MB）。你也可以提供一个

比 `block` 数量更大的值作为分区数目，但是，你不能提供一个小于 `block` 数量的值作为分区数目。

通过并行集合（数组）创建 RDD

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

下面请在 `pyspark` 中操作：

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

上面使用列表来创建。在 `Python` 中并没有数组这个基本数据类型，为了便于理解，你可以把列表当成其他语言的数组。

RDD 操作

RDD 被创建好以后，在后续使用过程中一般会发生两种操作：

* λ 转换（Transformation）： 基于现有的数据集创建一个新的数据集。

* λ 行动（Action）： 在数据集上进行运算，返回计算值。

2、转换操作

对于 `RDD` 而言，每一次转换操作都会产生不同的 `RDD`，供给下一个“转换”使用。转换得到的 `RDD` 是惰性求值的，也就是说，整个转换过程只是记录了转换的轨迹，并不会发生真正的计算，只有遇

到行动操作时，才会发生真正的计算，开始从血缘关系源头开始，进行物理的转换操作。

下面列出一些常见的转换操作（Transformation API）：

- * **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

3、行动操作

行动操作是真正触发计算的地方。Spark 程序执行到行动操作时，才会执行真正的计算，从文件中加载数据，完成一次又一次转换操作，最终，完成行动操作得到结果。

下面列出一些常见的行动操作（Action API）：

- * **count()** 返回数据集中的元素个数
- * **collect()** 以数组的形式返回数据集中的所有元素
- * **first()** 返回数据集中的第一个元素

- * `take(n)` 以数组的形式返回数据集中的前 n 个元素
- * `reduce(func)` 通过函数 `func`（输入两个参数并返回一个值）聚合数据集中的元素
- * `foreach(func)` 将数据集中的每个元素传递到函数 `func` 中运行*

4、惰性机制

这里给出一段简单的代码来解释 Spark 的惰性机制。

```
1. >>> lines = sc.textFile("data.txt")
2. >>> lineLengths = lines.map(lambda s : len(s))
3. >>> totalLength = lineLengths.reduce( lambda a, b
   : a + b)
```

上面第一行首先从外部文件 `data.txt` 中构建得到一个 RDD，名称为 `lines`，但是，由于 `textFile()`方法只是一个转换操作，因此，这行代码执行后，不会立即把 `data.txt` 文件加载到内存中，这时的 `lines` 只是一个指向这个文件的指针。

第二行代码用来计算每行的长度（即每行包含多少个单词），同样，由于 `map()`方法只是一个转换操作，这行代码执行后，不会立即计算每行的长度。

第三行代码的 `reduce()`方法是一个“动作”类型的操作，这时，就会触发真正的计算。这时，Spark 会把计算分解成多个任务在不同

的机器上执行，每台机器运行位于属于它自己的 map 和 reduce，最后把结果返回给 Driver Program。

5、实例

下面我们举几个实例加深了解。请在 pyspark 下执行下面操作。

下面是一个关于 filter() 操作的实例。

```
1. >>> lines = sc.textFile("file:///usr/local/spark/  
mycode/rdd/word.txt")  
2. >>> lines.filter(lambda line : "Spark" in line).c  
ount()  
3. 2
```

上面的代码中，`lines` 就是一个 RDD。`lines.filter()` 会遍历 `lines` 中的每行文本，并对每行文本执行括号中的匿名函数，也就是执行 Lamda 表达式：`line : "Spark" in line`，在执行 Lamda 表达式时，会把当前遍历到的这行文本内容赋值给参数 `line`，然后，执行处理逻辑 "`Spark`" in `line`，也就是只有当改行文本包含 "Spark" 才满足条件，才会被放入到结果集中。最后，等到 `lines` 集合遍历结束后，就会得到一个结果集，这个结果集中包含了所有包含 "Spark" 的行。最后，对这个结果集调用 `count()`，这是一个行动操作，会计算出结果集中的元素个数。

这里再给出另外一个实例，我们要找出文本文件中单行文本所包含的单词数量的最大值，代码如下：

```
1. >>> lines = sc.textFile("file:///usr/local/spark/  
mycode/rdd/word.txt")  
2. >>> lines.map(lambda line : len(line.split(" "))).reduce(lambda a,b : (a > b and a or b))  
3. 5
```

上面代码中，`lines` 是一个 `RDD`，是 `String` 类型的 `RDD`，因为这个 `RDD` 里面包含了很多行文本。

`lines.map()`，是一个转换操作，之前说过，`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，所以，`lines.map(lambda line : len(line.split(" ")))`会把每行文本都传递给匿名函数，也就是传递给 `Lambda` 表达式 `line : len(line.split(" "))`中的 `line`，然后执行处理逻辑 `len(line.split(" "))`。

`len(line.split(" "))`这个处理逻辑的功能是，对 `line` 文本内容进行单词切分，得到很多个单词构成的集合，然后，计算出这个集合中的单词的个数。

因此，最终 `lines.map(lambda line : len(line.split(" ")))`转换操作得到的 `RDD`，是一个整型 `RDD`，里面每个元素都是整数值（也就是单词的个数）。

最后，针对这个 `RDD[Int]`，调用 `reduce()` 行动操作，完成计算。
`reduce()` 操作每次接收两个参数，取出较大者留下，然后再继续比较，例如，`RDD[Int]` 中包含了 `1,2,3,4,5`，那么，执行 `reduce` 操作时，首先取出 `1` 和 `2`，把 `a` 赋值为 `1`，把 `b` 赋值为 `2`，然后，执行大小判断，保留 `2`。下一次，让保留下来的 `2` 赋值给 `a`，再从 `RDD` 中取出下一个元素 `3`，把 `3` 赋值给 `b`，然后，对 `a` 和 `b` 执行大小判断，保留较大者 `3`。依此类推。最终，`reduce()` 操作会得到最大值是 `5`。

实际上，如果我们把上面的 `lines.map(lambda line : len(line.split(" ")))`.`reduce(lambda a,b : (a > b and a or b))` 分开逐步执行，你就可以更加清晰地发现每个步骤生成的 `RDD` 的类型。

```
1. >>> lines = sc.textFile("file:///usr/local/spark/  
mycode/rdd/word.txt")  
2. >>> lines.map(lambda line : line.split(" "))  
3. //从上面执行结果可以发现，lines.map(line => line.sp  
lit(" ")) 返回的结果是分割后字符串列表 List  
4. >>> lines.map(line => len(line.split(" ")))  
5. // 这个 RDD 中的每个元素都是一个整数值（也就是一行文本  
包含的单词数）  
6. >>> lines.map(lambda line : len(line.split(" "))).reduce(lambda a,b : (a > b and a or b))
```

7.5

6、持久化

前面我们已经说过，在 Spark 中，RDD 采用惰性求值的机制，每次遇到行动操作，都会从头开始执行计算。如果整个 Spark 程序中只有一次行动操作，这当然不会有什么问题。但是，在一些情形下，我们需要多次调用不同的行动操作，这就意味着，每次调用行动操作，都会触发一次从头开始的计算。这对于迭代计算而言，代价是很大的，迭代计算经常需要多次重复使用同一组数据。

比如，下面就是多次计算同一个 DD 的例子：

```
1. >>> list = ["Hadoop", "Spark", "Hive"]
2. >>> rdd = sc.parallelize(list)
3. >>> print(rdd.count()) //行动操作，触发一次真正从头
                           到尾的计算
4. 3
5. >>> print(','.join(rdd.collect())) //行动操作，触
                           发一次真正从头到尾的计算
6. Hadoop,Spark,Hive
```

上面代码执行过程中，前后共触发了两次从头到尾的计算。

实际上，可以通过持久化（缓存）机制避免这种重复计算的开销。可以使用 `persist()` 方法对一个 RDD 标记为持久化，之所以说“标记为持久化”，是因为出现 `persist()` 语句的地方，并不会马上计算生成 RDD 并把它持久化，而是要等到遇到第一个行动操作触发真正计算以后，才会把计算结果进行持久化，持久化后的 RDD 将会被保留在计算节点的内存中被后面的行动操作重复使用。

`persist()` 的圆括号中包含的是持久化级别参数，比如，`persist(MEMORY_ONLY)` 表示将 RDD 作为反序列化的对象存储于 JVM 中，如果内存不足，就要按照 LRU 原则替换缓存中的内容。

`persist(MEMORY_AND_DISK)` 表示将 RDD 作为反序列化的对象存储在 JVM 中，如果内存不足，超出的分区将会被存放在硬盘上。一般而言，使用 `cache()` 方法时，会调用 `persist(MEMORY_only)`。

例子如下：

```
1. >>> list = ["Hadoop", "Spark", "Hive"]
2. >>> rdd = sc.parallelize(list)
3. >>> rdd.cache() //会调用 persist(MEMORY_ONLY)，但是，语句执行到这里，并不会缓存 rdd，这是 rdd 还没有被计算生成
```

```
4. >>> print(rdd.count()) //第一次行动操作，触发一次真正从头到尾的计算，这时才会执行上面的 rdd.cache()，把这个 rdd 放到缓存中
```

5.3

```
6. >>> print(','.join(rdd.collect())) //第二次行动操作，不需要触发从头到尾的计算，只需要重复使用上面缓存中的 rdd
```

7. Hadoop, Spark, Hive

最后，可以使用 `unpersist()`方法手动地把持久化的 RDD 从缓存中移除。

7、分区

RDD 是弹性分布式数据集，通常 RDD 很大，会被分成很多个分区，分别保存在不同的节点上。RDD 分区的一个分区原则是使得分区的个数尽量等于集群中的 CPU 核心（core）数目。

对于不同的 Spark 部署模式而言（本地模式、Standalone 模式、YARN 模式、Mesos 模式），都可以通过设置 `spark.default.parallelism` 这个参数的值，来配置默认的分区数目，一般而言：

***本地模式**：默认为本地机器的 CPU 数目，若设置了 `local[N]`，则默认为 N；

***Apache Mesos**：默认的分区数为 8；

***Standalone** 或 **YARN**: 在“集群中所有 CPU 核心数目总和”和“2”二者中取较大值作为默认值;

因此，对于 **parallelize** 而言，如果没有在方法中指定分区数，则默认为 **spark.default.parallelism**，比如：

```
1. >>> array = [1,2,3,4,5]
2. >>> rdd = sc.parallelize(array,2) #设置两个分区
```

对于 **textFile** 而言，如果没有在方法中指定分区数，则默认为 **min(defaultParallelism,2)**，其中，**defaultParallelism** 对应的就是 **spark.default.parallelism**。

如果是从 **HDFS** 中读取文件，则分区数为文件分片数(比如，128MB/片)。

第六次上机实验内容

本次上机实验将继续深入了解键值对 RDD 的运算处理机制。

虽然 RDD 中可以包含任何类型的对象，但是“键值对”是一种比较常见的 RDD 元素类型，分组和聚合操作中经常会用到。

Spark 操作中经常会用到“键值对 RDD”（Pair RDD），用于完成聚合计算。普通 RDD 里面存储的数据类型是 Int、String 等，而“键值对 RDD”里面存储的数据类型是“键值对”。

1、创建 RDD 之前的准备工作

在即将进行相关的实践操作之前，我们首先要登录 Linux 系统（本教程统一采用 hadoop 用户登录），然后，打开命令行“终端”。

请按照下面的命令启动 Hadoop 中的 HDFS 组件：

```
1. cd /usr/local/hadoop  
2. ./sbin/start-dfs.sh
```

然后，我们按照下面命令启动 pyspark：

```
1. cd /usr/local/spark  
2. ./bin/pyspark
```

然后，新建第二个“终端”，方法是，在前面已经建设的第一个终端窗口的左上方，点击“终端”菜单，在弹出的子菜单中选择“新建终端”，就可以打开第二个终端窗口，现在，我们切换到第二个终端窗口，在第二个终端窗口中，执行以下命令，进入之前已经创建好的“/usr/local/spark/mycode/”目录，在这个目录下新建 pairrdd 子目录，用来存放本章的代码和相关文件：

```
1. cd /usr/local/spark/mycode/  
2. mkdir pairrdd
```

然后，使用 vim 编辑器，在 pairrdd 目录下新建一个 word.txt 文件，你可以在文件里面随便输入几行英文语句用来测试。

经过上面的准备工作以后，我们就可以开始创建 RDD 了。

键值对 RDD 的创建

2、第一种创建方式：从文件中加载

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

```
1. >>> lines = sc.textFile("file:///usr/local/spark  
/mycode/pairrdd/word.txt")  
2. >>> pairRDD = lines.flatMap(lambda line : line.sp  
lit(" ")).map(lambda word : (word,1))
```

```
3. >>> pairRDD.foreach(print)
4. (i,1)
5. (love,1)
6. (hadoop,1)
7. (i,1)
8. (love,1)
9. (Spark,1)
10. (Spark,1)
11. (is,1)
12. (fast,1)
13. (than,1)
14. (hadoop,1)
```

我们之前在“第一个 Spark 应用程序:WordCount”章节已经详细解释过类似代码，所以，上面代码不再做细节分析。从代码执行返回信息：pairRDD: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at map at :29，可以看出，返回的结果是键值对类型的 RDD，即 RDD[(String, Int)]。从 pairRDD.foreach(**println**) 执行的打印输出结果也可以看到，都是由(单词,1)这种形式的键值对。

3、第二种创建方式：通过并行集合（列表）创建 RDD

```
1. >>> list = ["Hadoop","Spark","Hive","Spark"]
```

```
2. >>> rdd = sc.parallelize(list)
3. >>> pairRDD = rdd.map(lambda word : (word,1))
4. >>> pairRDD.foreach(print)
5. (Hadoop,1)
6. (Spark,1)
7. (Hive,1)
8. (Spark,1)
```

我们下面实例都是采用这种方式得到的 pairRDD 作为基础。

4、常用的键值对转换操作

常用的键值对转换操作包括 `reduceByKey()`、`groupByKey()`、`sortByKey()`、`join()`、`cogroup()` 等，下面我们通过实例来介绍。

`reduceByKey(func)`

`reduceByKey(func)` 的功能是，使用 `func` 函数合并具有相同键的值。比如，`reduceByKey((a,b) => a+b)`，有四个键值对 ("spark",1)、("spark",2)、("hadoop",3) 和 ("hadoop",5)，对具有相同 `key` 的键值对进行合并后的结果就是： ("spark",3)、("hadoop",8)。可以看出，`(a,b) => a+b` 这个 `Lamda` 表达式中，`a` 和 `b` 都是指 `value`，比如，对于两个具有相同 `key` 的键值对 ("spark",1)、("spark",2)，`a` 就是 1，`b` 就是 2。

我们对上面第二种方式创建得到的 pairRDD 进行 reduceByKey() 操作，代码如下：

```
1. >>> pairRDD.reduceByKey(lambda a,b : a+b).foreach  
(print)  
2. (Spark,2)  
3. (Hive,1)  
4. (Hadoop,1)
```

groupByKey()

groupByKey() 的功能是，对具有相同键的值进行分组。比如，对四个键值对("spark",1)、("spark",2)、("hadoop",3)和("hadoop",5)，采用 groupByKey() 后得到的结果是： ("spark",(1,2)) 和 ("hadoop",(3,5))。

我们对上面第二种方式创建得到的 pairRDD 进行 groupByKey() 操作，代码如下：

```
1. >>> pairRDD.groupByKey()  
2. PythonRDD[11] at RDD at PythonRDD.scala:48  
3. >>> pairRDD.groupByKey().foreach(print)  
4. ('spark', <pyspark.resultiterable.ResultIterable  
object at 0x7f1869f81f60>)
```

```
5. ('hadoop', <pyspark.resultiterable.ResultIterable  
      object at 0x7f1869f81f60>)  
6. ('hive', <pyspark.resultiterable.ResultIterable o  
      bject at 0x7f1869f81f60>)
```

keys()

keys()只会把键值对 RDD 中的 key 返回形成一个新的 RDD。比如，对四个键值对("spark",1)、("spark",2)、("hadoop",3)和 ("hadoop",5)构成的 RDD，采用 keys()后得到的结果是一个 RDD[Int]，内容是{"spark","spark","hadoop","hadoop"}。

我们对上面第二种方式创建得到的 pairRDD 进行 keys 操作，代码如下：

```
1. >>> pairRDD.keys()  
2. PythonRDD[20] at RDD at PythonRDD.scala:48  
3. >>> pairRDD.keys().foreach(print)  
4. Hadoop  
5. Spark  
6. Hive  
7. Spark
```

values()

`values()`只会把键值对 RDD 中的 `value` 返回形成一个新的 RDD。比如，对四个键值对("spark",1)、("spark",2)、("hadoop",3) 和("hadoop",5)构成的 RDD，采用 `values()`后得到的结果是一个 RDD[Int]，内容是{1,2,3,5}。

我们对上面第二种方式创建得到的 pairRDD 进行 `values()`操作，代码如下：

```
1. scala> pairRDD.values()
2. PythonRDD[22] at RDD at PythonRDD.scala:48
3. scala> pairRDD.values().foreach(println)
4. 1
5. 1
6. 1
7. 1
```

sortByKey()

`sortByKey()`的功能是返回一个根据键排序的 RDD。

我们对上面第二种方式创建得到的 pairRDD 进行 `keys` 操作，代码如下：

```
1. >>> pairRDD.sortByKey()
2. PythonRDD[30] at RDD at PythonRDD.scala:48
```

```
3. >>> pairRDD.sortByKey().foreach(print)
4. (Hadoop,1)
5. (Hive,1)
6. (Spark,1)
7. (Spark,1)
```

mapValues(func)

我们经常会遇到一种情形，我们只想对键值对 RDD 的 value 部分进行处理，而不是同时对 key 和 value 进行处理。对于这种情形，Spark 提供了 mapValues(func)，它的功能是，对键值对 RDD 中的每个 value 都应用一个函数，但是，key 不会发生变化。比如，对四个键值对("spark",1)、("spark",2)、("hadoop",3)和 ("hadoop",5)构成的 pairRDD，如果执行 pairRDD.mapValues (lambda x : x+1)，就会得到一个新的键值对 RDD，它包含下面四个键值对("spark",2)、("spark",3)、("hadoop",4)和("hadoop",6)。

我们对上面第二种方式创建得到的 pairRDD 进行 keys 操作，代码如下：

```
1. >>> pairRDD.mapValues(lambda x : x+1)
2. PythonRDD[38] at RDD at PythonRDD.scala:48
3. >>> pairRDD.mapValues( lambda x : x+1).foreach(pr
int)
```

4. (Hadoop,2)

5. (Spark,2)

6. (Hive,2)

7. (Spark,2)

join

join(连接)操作是键值对常用的操作。“连接”(join)这个概念来自于关系数据库领域，因此，join 的类型也和关系数据库中的 join 一样，包括内连接、左外连接、右外连接等。最常用的情形是内连接，所以，join 就表示内连接。

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

比如，pairRDD1 是一个键值对集合{("spark",1)、("spark",2)、("hadoop",3)和("hadoop",5)}，pairRDD2 是一个键值对集合{("spark","fast")}，那么，pairRDD1.join(pairRDD2)的结果就是一个新的 RDD，这个新的 RDD 是键值对集合{("spark",1,"fast"), ("spark",2,"fast")}。对于这个实例，我们下面在 pyspark 中运行一下：

```
1. >>> pairRDD1 = sc.parallelize([('spark',1),('spark',2),('hadoop',3),('hadoop',5)])
```

```
2. >>> pairRDD2 = sc.parallelize([('spark','fast')])  
3. >>> pairRDD1.join(pairRDD2)  
4. PythonRDD[49] at RDD at PythonRDD.scala:48  
5. >>> pairRDD1.join(pairRDD2).foreach(println)
```

5、一个综合实例：

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

很显然，对于上面的题目，结果是很显然的， ("spark",4), ("hadoop",5)。

下面，我们在 **pyspark** 中演示代码执行过程：

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

要注意，上面语句中，`mapValues(lambda x : (x,1))`中出现了变量 `x`，`reduceByKey(lambda x,y : (x[0]+y[0],x[1]+ y[1]))` 中也出现了变量 `x`，`mapValues(lambda x : (x[0] / x[1]))` 也出现了变量 `x`。

但是，必须要清楚，这三个地方出现的 `x`，虽然都具有相同的变量名称 `x`，但是，彼此之间没有任何关系，它们都处在不同的变量作用域内。如果你觉得这样会误导自己，造成理解上的掌握，实际上，你可以把三个出现 `x` 的地方分别替换成 `x1`、`x2`、`x3` 也是可以的，但是，很显然没有必要这么做。

上面是完整的语句和执行过程，可能不太好理解，下面我们进行逐条语句分解给大家介绍。每条语句执行后返回的屏幕信息，可以帮助大家更好理解语句的执行效果，比如生成了什么类型的 RDD。

(1) 首先构建一个数组，数组里面包含了四个键值对，然后，调用 `parallelize()` 方法生成 RDD，从执行结果反馈信息，可以看出，`rdd` 类型是 `RDD[(String, Int)]`。

```
1. >>> rdd = sc.parallelize([('spark',2),("hadoop",6),("hadoop",4),("spark",6)])
```

(2) 针对构建得到的 `rdd`，我们调用 `mapValues()` 函数，把 `rdd` 中的每个每个键值对(`key,value`)的 `value` 部分进行修改，把 `value` 转换为键值对(`value,1`)，其中，数值 1 表示这个 `key` 在 `rdd` 中出现了 1 次，为什么要记录出现次数呢？

因为，我们最终要计算每个 `key` 对应的平均值，所以，必须记住这个 `key` 出现了几次，最后用 `value` 的总和除以 `key` 的出现次数，就是这个 `key` 对应的平均值。

(3) 然后，再对上一步得到的 `RDD` 调用 `reduceByKey()` 函数，在 `spark-shell` 中演示如下：

```
1. >>> rdd.mapValues(lambda x : (x,1)).reduceByKey(lambda x,y : (x[0]+y[0],x[1] + y[1])).collect()
```

这里，必须要十分准确地理解 `reduceByKey()` 函数的功能。可以参考上面我们对该函数的介绍，`reduceByKey(func)` 的功能是使用 `func` 函数合并具有相同键的值。

这里的 `func` 函数就是 `Lambda` 表达式 `x,y : (x[0]+y[0],x[1] + y[1])`，这个表达式中，`x` 和 `y` 都是 `value`，而且是具有相同 `key` 的两个键值对所对应的 `value`，比如，在这个例子中，`("hadoop", (6,1))` 和 `("hadoop", (4,1))` 这两个键值对具有相同的 `key`，所以，对于函数中的输入参数 `(x,y)` 而言，`x` 就是 `(6,1)`，序列从 0 开始计算，`x[0]` 表示这个键值对中的第 1 个元素 6，`x[1]` 表示这个键值对中的第二个元素 1，`y` 就是 `(4,1)`，`y[0]` 表示这个键值对中的第 1 个元素 4，`y[1]` 表示这个键值对中的第二个元素 1，所以，函数体 `(x[0]+y[0],x[1] + y[1])`，相当于生成一个新的键值对 `(key,value)`，其中，`key` 是 `x[0]+y[0]`，也就是 `6+4=10`，`value` 是 `x[1] + y[1]`，也就是 `1+1=2`，因此，函数

体($x[0]+y[0]$, $x[1] + y[1]$)执行后得到的 value 是(10,2)，但是，要注意，这个(10,2)是 reduceByKey()函数执行后，“hadoop”这个 key 对应的 value，也就是，实际上 reduceByKey()函数执行后，会生成一个键值对("hadoop", (10,2))，其中，10 表示 hadoop 书籍的总销量，2 表示两天。同理，reduceByKey()函数执行后会生成另外一个键值对("spark", (8,2))。

最后，就可以求出最终结果。我们可以对上面得到的两个键值对 ("hadoop", (10,2)) 和 ("spark", (8,2)) 所构成的 RDD 执行 mapValues() 操作，得到每种书的每天平均销量。当第一个键值对("hadoop", (10,2)) 输入给 mapValues($x \Rightarrow (x[0] / x[1])$) 操作时，key 是 "hadoop"，保持不变，value 是(10,2)，会被赋值给 Lamda 表达式 $x \Rightarrow (x[0] / x[1])$ 中的 x，因此，x 的值就是(10,2)， $x[0]$ 就是 10，表示 hadoop 书总销量是 10， $x[1]$ 就是 2，表示 2 天，因此，hadoop 书籍的每天平均销量就是 $x[0] / x[1]$ ，也就是 5。mapValues() 输出的一个键值对就是("hadoop", 5)。同理，当把("spark", (8,2)) 输入给 mapValues() 时，会计算得到另外一个键值对("spark", 4)。

在 pyspark 中演示如下：

```
1. >>> rdd.mapValues(lambda x: (x[0],x[1])).reduceByKey(lambda x,y: (x[0]+y[0],x[1]+y[1])).mapValues(lambda x: (x[0]/x[1])).collect()
```

2. [('hadoop', 5.0), ('spark', 4.0)]

第七次上机实验内容

传统的机器学习算法，由于技术和单机存储的限制，只能在少量数据上使用。即以前的统计/机器学习依赖于数据抽样。但实际过程中样本往往很难做好随机，导致学习的模型不是很准确，在测试数据上的效果也可能不太好。随着 **HDFS(Hadoop Distributed File System)** 等分布式文件系统出现，存储海量数据已经成为可能。在全量数据上进行机器学习也成为了可能，这顺便也解决了统计随机性的问题。然而，由于 **MapReduce** 自身的限制，使得使用 **MapReduce** 来实现分布式机器学习算法非常耗时和消耗磁盘 **IO**。因为通常情况下机器学习算法参数学习的过程都是迭代计算的，即本次计算的结果要作为下一次迭代的输入，这个过程中，如果使用 **MapReduce**，我们只能把中间结果存储磁盘，然后在下一次计算的时候从新读取，这对于迭代 频发的算法显然是致命的性能瓶颈。

在大数据上进行机器学习，需要处理全量数据并进行大量的迭代计算，这要求机器学习平台具备强大的处理能力。**Spark** 立足于内存计算，天然的适应于迭代式计算。即便如此，对于普通开发者来说，实现一个分布式机器学习算法仍然是一件极具挑战的事情。幸运的是，**Spark** 提供了一个基于海量数据的机器学习库，它提供了常用机器学习算法的分布式实现，开发者只需要有 **Spark** 基础并且了解机器学习算法的原理，以及方法相关参数的含义，就可以轻松的通过调用相应的 **API** 来实现基于海量数据的机器学习过程。其

次，**Spark-Shell** 的即席查询也是一个关键。算法工程师可以边写代码边运行，边看结果。**spark** 提供的各种高效的工具正使得机器学习过程更加直观便捷。比如通过 **sample** 函数，可以非常方便的进行抽样。当然，**Spark** 发展到后面，拥有了实时批计算，批处理，算法库，**SQL**、流计算等模块等，基本可以看做是全平台的系统。把机器学习作为一个模块加入到 **Spark** 中，也是大势所趋。

1、下面将练习如何构建一个机器学习工作流：

本节以逻辑斯蒂回归为例，构建一个典型的机器学习过程，来具体介绍一下工作流是如何应用的。我们的目的是查找出所有包含 "spark" 的句子，即将包含"spark"的句子的标签设为 1，没有"spark"的句子的标签设为 0。

Spark2.0 以上版本的 **pyspark** 创建一个名为 **spark** 的 **SparkSession** 对象，当需要手工创建时，**SparkSession** 可以由其伴生对象的 **builder()**方法创建出来，如下代码段所示：

```
1. spark = SparkSession.builder.master("local").appName("Word Count").getOrCreate()
```

下文中，我们默认名为 **spark** 的 **SparkSession** 已经创建。

pyspark.ml 依赖 **numpy** 包，Ubuntu 自带 **python3** 是没有 **numpy** 的，执行如下命令安装：

```
1. sudo pip3 install numpy
```

然后，我们引入要包含的包并构建训练数据集。

```
1. from pyspark.ml import Pipeline  
2. from pyspark.ml.classification import LogisticRe  
gression  
3. from pyspark.ml.feature import HashingTF, Tokeniz  
er  
4.  
5. # Prepare training documents from a list of (id, t  
ext, label) tuples.  
6. training = spark.createDataFrame([  
7.     (0, "a b c d e spark", 1.0),  
8.     (1, "b d", 0.0),  
9.     (2, "spark f g h", 1.0),  
10.    (3, "hadoop mapreduce", 0.0)  
11. ], ["id", "text", "label"])
```

在这一步中我们要定义 Pipeline 中的各个工作流阶段 Pipeline Stage，包括转换器和评估器，具体的，包含 tokenizer, hashingTF 和 lr 三个步骤。

```
1. tokenizer = Tokenizer(inputCol="text", outputCol="words")  
2. hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")  
3. lr = LogisticRegression(maxIter=10, regParam=0.001)
```

有了这些处理特定问题的转换器和评估器，接下来就可以按照具体的处理逻辑有序的组织 `PipelineStages` 并创建一个 `Pipeline`。

```
1. pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
```

现在构建的 `Pipeline` 本质上是一个 `Estimator`，在它的 `fit` 方法运行之后，它将产生一个 `PipelineModel`，它是一个 `Transformer`。

```
1. model = pipeline.fit(training)
```

可以看到，`model` 的类型是一个 `PipelineModel`，这个管道模型将在测试数据的时候使用。所以接下来，我们先构建测试数据。

```
1. test = spark.createDataFrame([(4, "spark i j k"), (5, "l m n"),
```

```
4.      (6, "spark hadoop spark"),
5.      (7, "apache hadoop")
6. ], ["id", "text"])
```

然后，我们调用我们训练好的 PipelineModel 的 transform () 方法，让测试数据按顺序通过拟合的工作流，生成我们所需要的预测结果。

```
1. prediction = model.transform(test)
2. selected = prediction.select("id", "text", "probability", "prediction")
3. for row in selected.collect():
4.     rid, text, prob, prediction = row
5.     print("%d, %s) --> prob=%s, prediction=%f" %
       (rid, text, str(prob), prediction))
6.
7. (4, spark i j k) --> prob=[0.155543713844,0.84445
   6286156], prediction=1.000000
8. (5, l m n) --> prob=[0.830707735211,0.16929226478
   9], prediction=0.000000
9. (6, spark hadoop spark) --> prob=[0.069621840619
   5,0.93037815938], prediction=1.000000
```

```
10. (7, apache hadoop) --> prob=[0.981518350351,0.01  
8481649649], prediction=0.000000
```

通过上述结果，我们可以看到，第 4 句第 6 句中都包含 spark，其中第六句的预测是 1，与我们希望的相符；而第 4 句虽然预测的依然是 0，但是通过概率我们可以看到，第 4 句有 46% 的概率预测是 1，而第 5 句、第 7 句分别只有 7% 和 2% 的概率预测为 1，这是由于训练数据集较少，如果有更多的测试数据进行学习，预测的准确率将会有显著提升。

2、特征提取

(1) “词频—逆向文件频率” (TF-IDF)

首先，导入 TFIDF 所需要的包：

```
1. from pyspark.ml.feature import HashingTF, IDF, Tok  
enizer
```

准备工作完成后，我们创建一个简单的 DataFrame，每一个句子代表一个文档。

```
1. sentenceData = spark.createDataFrame([(0, "I hear  
d about Spark and I love Spark"),(0, "I wish Java  
could use case classes"),(1, "Logistic regression  
models are neat")]).toDF("label", "sentence")
```

在得到文档集合后，即可用 `tokenizer` 对句子进行分词

```
1. tokenizer = Tokenizer(inputCol="sentence", outputCol="words")  
2. wordsData = tokenizer.transform(sentenceData)
```

得到分词后的文档序列后，即可使用 `HashingTF` 的 `transform()` 方法把句子哈希成特征向量，这里设置哈希表的桶数为 2000。

```
1. hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=20)  
2. featurizedData = hashingTF.transform(wordsData)
```

可以看到，分词序列被转换成一个稀疏特征向量，其中每个单词都被散列成了一个不同的索引值，特征向量在某一维度上的值即该词汇在文档中出现的次数。

最后，使用 `IDF` 来对单纯的词频特征向量进行修正，使其更能体现不同词汇对文本的区别能力，`IDF` 是一个 `Estimator`，调用 `fit()` 方法并将词频向量传入，即产生一个 `IDFModel`。

```
1. idf = IDF(inputCol="rawFeatures", outputCol="features")  
2. idfModel = idf.fit(featurizedData)
```

很显然，`IDFModel` 是一个 `Transformer`，调用它的 `transform()` 方法，即可得到每一个单词对应的 **TF-IDF** 度量值。

```
1. rescaledData = idfModel.transform(featurizedData)  
2. rescaledData.select("label", "features").show()
```

可以看到，特征向量已经被其在语料库中出现的总次数进行了修正，通过 **TF-IDF** 得到的特征向量，在接下来可以被应用到相关的机器学习方法中。

(2) Word2Vec

首先，导入 `Word2Vec` 所需要的包，并创建三个词语序列，每个代表一个文档：

```
1. from pyspark.ml.feature import Word2Vec  
2. documentDF = spark.createDataFrame([  
3.     ("Hi I heard about Spark".split(" "), ),  
4.     ("I wish Java could use case classes".split(" "  
    ), ),  
5.     ("Logistic regression models are neat".split(" "  
    ), )  
6. ], [ "text" ])
```

新建一个 Word2Vec，显然，它是一个 Estimator，设置相应的超参数，这里设置特征向量的维度为 3，Word2Vec 模型还有其他可设置的超参数，具体的超参数描述可以参见[这里](#)。

```
1. word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="text", outputCol="result")
```

读入训练数据，用 fit()方法生成一个 Word2VecModel。

```
1. model = word2Vec.fit(documentDF)
```

利用 Word2VecModel 把文档转变成特征向量。

```
1. result = model.transform(documentDF)
2. for row in result.collect():
3.     text, vector = row
4.     print("Text: [%s] => \nVector: %s\n" % (", ".join(text), str(vector)))
5.
6.
7. Text: [Hi, I, heard, about, Spark] =>
8. Vector: [0.0127797678113,-0.0934097565711,-0.108308439702]
9.
```

```
10. Text: [I, wish, Java, could, use, case, classes]
=>

11. Vector: [0.0761276933564, 0.0345174372196, -0.0429
060061329]

12.

13. Text: [Logistic, regression, models, are, neat]
=>

14. Vector: [-0.0675941422582, 0.0452983468771, 0.0530
217912048]

15.
```

可以看到，文档被转变为了一个 3 维的特征向量，这些特征向量就可以被应用到相关的机器学习方法中。

(3) CountVectorizer

我们接下来通过一个例子来进行介绍。首先，导入 CountVectorizer 所需要的包：

```
1. from pyspark.ml.feature import CountVectorizer
```

假设我们有如下的 DataFrame，其包含 id 和 words 两列，可以看成是一个包含两个文档的迷你语料库。

```
1. df = spark.createDataFrame([
```

```
2.    (0, "a b c".split(" ")),  
3.    (1, "a b b c a".split(" "))  
4. ], ["id", "words"])
```

随后，通过 `CountVectorizer` 设定超参数，训练一个 `CountVectorizer`，这里设定词汇表的最大量为 3，设定词汇表中的词至少要在 2 个文档中出现过，以过滤那些偶然出现的词汇。

```
1. # fit a CountVectorizerModel from the corpus.  
2. cv = CountVectorizer(inputCol="words", outputCol=  
    "features", vocabSize=3, minDF=2.0)
```

在训练结束后，可以通过 `cv` 对 `DataFrame` 进行 `fit`, 获得到模型的词汇表：

```
1. model = cv.fit(df)
```

使用这一模型对 `DataFrame` 进行变换，可以得到文档的向量化表示

```
result = model.transform(df)  
result.show(truncate=False)  
  
+---+-----+-----+  
| id |words      |features          |  
+---+-----+-----+
```

+-----+	+-----+
0 [a, b, c]	(3,[0,1,2],[1.0,1.0,1.0])
1 [a, b, b, c, a]	(3,[0,1,2],[2.0,2.0,1.0])
+-----+	+-----+

第八次上机实验内容

1、标签和索引的转化

在机器学习处理过程中，为了方便相关算法的实现，经常需要把标签数据（一般是字符串）转化成整数索引，或是在计算结束后将整数索引还原为相应的标签，这需要通过转换器完成。

Spark ML 包中提供了几个相关转换器，例如：`StringIndexer`、`IndexToString`、`OneHotEncoder`、`VectorIndexer`，它们提供了十分方便的特征转换功能，这些转换器类都位于 `org.apache.spark.ml.feature` 包下。

值得注意的是，用于特征转换的转换器和其他的机器学习算法一样，也属于 ML Pipeline 模型的一部分，可以用来构成机器学习流水线，以 `StringIndexer` 为例，其存储着进行标签数值化过程的相关超参数，是一个 `Estimator`，对其调用 `fit(..)` 方法即可生成相应的模型 `StringIndexerModel` 类，很显然，它存储了用于 `DataFrame` 进行相关处理的参数，是一个 `Transformer`。

下面对几个常用的转换器依次进行介绍。

(1) `StringIndexer`:

`StringIndexer` 转换器可以把一列类别型的特征（或标签）进行编码，使其数值化，索引的范围从 0 开始，该过程可以使得相应的

特征索引化，使得某些无法接受类别型特征的算法可以使用，并提高诸如决策树等机器学习算法的效率。

索引构建的顺序为标签的频率，优先编码频率较大的标签，所以出现频率最高的标签为 0 号。

如果输入的是数值型的，我们会把它转化成字符型，然后再对其进行编码。

首先，引入必要的包，并创建一个简单的 DataFrame，它只包含一个 id 列和一个标签列 category：

```
1. from pyspark.ml.feature import StringIndexer  
2.  
3. df = spark.createDataFrame(  
4.     [(0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "  
a"), (5, "c")],  
5.     ["id", "category"])
```

随后，我们创建一个 StringIndexer 对象，设定输入输出列名，其余参数采用默认值，并对这个 DataFrame 进行训练，产生 StringIndexerModel 对象：

```
1. indexer = StringIndexer(inputCol="category", outp  
utCol="categoryIndex")
```

```
2. model = indexer.fit(df)
```

随后即可利用该对象对 DataFrame 进行转换操作，可以看到，
StringIndexerModel 依次按照出现频率的高低，把字符标签进行了
排序，即出现最多的“a”被编号成 0，“c”为 1，出现最少的“b”为 0。

```
1. indexed = model.transform(df)
```

```
2. indexed.show()
```

```
3.
```

```
4. +---+-----+-----+
```

```
5. | id|category|categoryIndex|
```

```
6. +---+-----+-----+
```

```
7. | 0|      a|      0.0|
```

```
8. | 1|      b|      2.0|
```

```
9. | 2|      c|      1.0|
```

```
10. | 3|      a|      0.0|
```

```
11. | 4|      a|      0.0|
```

```
12. | 5|      c|      1.0|
```

```
13. +---+-----+-----+
```

```
14.
```

(2) IndexToString

与 `StringIndexer` 相对应，`IndexToString` 的作用是把标签索引的一列重新映射回原有的字符型标签。

其主要使用场景一般都是和 `StringIndexer` 配合，先用 `StringIndexer` 将标签转化成标签索引，进行模型训练，然后在预测标签的时候再把标签索引转化成原有的字符标签。当然，你也可以另外定义其他的标签。

首先，和 `StringIndexer` 的实验相同，我们用 `StringIndexer` 读取数据集中的“category”列，把字符型标签转化成标签索引，然后输出到“categoryIndex”列上，构建出新的 `DataFrame`。

```
1. from pyspark.ml.feature import IndexToString, StringIndexer  
2.  
3. df = spark.createDataFrame(  
4.     [(0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c")],  
5.     ["id", "category"])  
6.  
7. indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")  
8. model = indexer.fit(df)
```

```
9. indexed = model.transform(df)
```

随后，创建 `IndexToString` 对象，读取“`categoryIndex`”上的标签索引，获得原有数据集的字符型标签，然后再输出到“`originalCategory`”列上。最后，通过输出“`originalCategory`”列，可以看到数据集中原有的字符标签。

```
1. converter = IndexToString(inputCol="categoryIndex", outputCol="originalCategory")  
2. converted = converter.transform(indexed)  
3. converted.select("id", "categoryIndex", "originalCategory").show()  
4.  
5. +---+-----+-----+  
6. | id|categoryIndex|originalCategory|  
7. +---+-----+-----+  
8. | 0|      0.0|          a|  
9. | 1|      2.0|          b|  
10. | 2|      1.0|          c|  
11. | 3|      0.0|          a|  
12. | 4|      0.0|          a|  
13. | 5|      1.0|          c|  
14. +---+-----+-----+
```

15.

(3) OneHotEncoder

独热编码（One-Hot Encoding）是指把一列类别性特征映射成一系列的二元连续特征的过程，原有的类别性特征有几种可能取值，这一特征就会被映射成几个二元连续特征，每一个特征代表一种取值，若该样本表现出该特征，则取 1，否则取 0。

One-Hot 编码适合一些期望类别特征为连续特征的算法，比如说逻辑斯蒂回归等。

首先创建一个 DataFrame，其包含一列类别性特征，需要注意的是，在使用 OneHotEncoder 进行转换前，DataFrame 需要先使用 StringIndexer 将原始标签数值化：

```
1. from pyspark.ml.feature import OneHotEncoder, StringIndexer  
2.  
3. df = spark.createDataFrame([  
4.     (0, "a"),  
5.     (1, "b"),  
6.     (2, "c"),  
7.     (3, "a"),
```

```
8.      (4, "a"),
9.      (5, "c")
10. ], ["id", "category"])
11.
12. stringIndexer = StringIndexer(inputCol="category",
13.                                outputCol="categoryIndex")
14. model = stringIndexer.fit(df)
15. indexed = model.transform(df)
```

随后，我们创建 `OneHotEncoder` 对象对处理后的 `DataFrame` 进行编码，可以看见，编码后的二进制特征呈稀疏向量形式，与 `StringIndexer` 编码的顺序相同，需注意的是最后一个 Category ("b") 被编码为全 0 向量，若希望 "b" 也占有一个二进制特征，则可在创建 `OneHotEncoder` 时指定 `setDropLast(false)`。

```
1. encoder = OneHotEncoder(inputCol="categoryIndex",
                           outputCol="categoryVec")
2. encoded = encoder.transform(indexed)
3. encoded.show()
4.
5. +---+-----+-----+-----+
6. | id|category|categoryIndex|  categoryVec|
7. +---+-----+-----+-----+
```

```
8. |  0|      a|      0.0|(2,[0],[1.0])|
9. |  1|      b|      2.0|(2,[],[])
10. |  2|      c|      1.0|(2,[1],[1.0])|
11. |  3|      a|      0.0|(2,[0],[1.0])|
12. |  4|      a|      0.0|(2,[0],[1.0])|
13. |  5|      c|      1.0|(2,[1],[1.0])|
14. +---+-----+-----+-----+
```

(4) VectorIndexer

之前介绍的 `StringIndexer` 是针对单个类别型特征进行转换，倘若所有特征都已经被组织在一个向量中，又想对其中某些单个分量进行处理时，`Spark ML` 提供了 `VectorIndexer` 类来解决向量数据集中的类别性特征转换。

通过为其提供 `maxCategories` 超参数，它可以自动识别哪些特征是类别型的，并且将原始值转换为类别索引。它基于不同特征值的数量来识别哪些特征需要被类别化，那些取值可能性最多不超过 `maxCategories` 的特征需要会被认为是类别型的。

在下面的例子中，我们读入一个数据集，然后使用 `VectorIndexer` 训练出模型，来决定哪些特征需要被作为类别特征，将类别特征转换为索引，这里设置 `maxCategories` 为 10，即只有种类少于 10 的特征才被认为是类别型特征，否则被认为是连续型特征：

```
from pyspark.ml.feature import VectorIndexer

data = spark.read.format('libsvm').load('file:///usr/local/spark/data/mllib/sample_libsvm_data.txt')

indexer = VectorIndexer(inputCol="features", outputCol
="indexed", maxCategories=10)

indexerModel = indexer.fit(data)

categoricalFeatures = indexerModel.categoryMaps

indexedData = indexerModel.transform(data)

indexedData.show()
```

```
+-----+-----+-----+
|label|      features|      indexed|
+-----+-----+-----+
| 0.0|(692,[127,128,129...|(692,[127,128,129...
| 0.0|(692,[152,153,154...|(692,[152,153,154...
```

```
| 1.0|(692,[97,98,99,12...|(692,[97,98,99,12...
```

```
| 1.0|(692,[124,125,126...|(692,[124,125,126...
```

```
+-----+-----+-----+-----+
```

2、特征选择

特征选择指的是在特征向量中选择出那些“优秀”的特征，组成新的、更“精简”的特征向量的过程。它在高维数据分析中十分常用，可以剔除掉“冗余”和“无关”的特征，提升学习器的性能。

特征选择方法和分类方法一样，也主要分为有监督和无监督两种，卡方选择则是统计学上常用的一种有监督特征选择方法，它通过对特征和真实标签之间进行卡方检验，来判断该特征和真实标签的关联程度，进而确定是否对其进行选择。

和 ML 库中的大多数学习方法一样，ML 中的卡方选择也是以 estimator+transformer 的形式出现的，其主要由 ChiSqSelector 和 ChiSqSelectorModel 两个类来实现。

在进行实验前，首先进行环境的设置。引入卡方选择器所需要使用的类：

```
1. from pyspark.ml.feature import ChiSqSelector  
2. from pyspark.ml.linalg import Vectors
```

默认名为 spark 的 SparkSession 已经创建。

随后，创造实验数据，这是一个具有三个样本，四个特征维度的数据集，标签有 1, 0 两种，我们将在此数据集上进行卡方选择：

```
1. df = spark.createDataFrame([  
2.     (7, Vectors.dense([0.0, 0.0, 18.0, 1.0]), 1.  
0,),  
3.     (8, Vectors.dense([0.0, 1.0, 12.0, 0.0]), 0.  
0,),  
4.     (9, Vectors.dense([1.0, 0.0, 15.0, 0.1]), 0.  
0,)], ["id", "features", "clicked"])
```

现在，用卡方选择进行特征选择器的训练，为了观察地更明显，我们设置只选择和标签关联性最强的一个特征可以通过 numTopFeatures 参数方法进行设置：

```
1. selector = ChiSqSelector(numTopFeatures=1, featur  
esCol="features",  
2.                             outputCol="selectedFeature  
s", labelCol="clicked")
```

3.

用训练出的模型对原数据集进行处理，可以看见，第三列特征被选出作为最有用的特征列：

```
1. result = selector.fit(df).transform(df)  
2. result.show()
```

第九次上机实验内容

分类是一种重要的机器学习和数据挖掘技术。分类的目的是根据数据集的特点构造一个分类函数或分类模型，该模型能把未知类别的样本映射到给定类别中的一种技术。

构造分类模型的过程一般分为训练和测试两个阶段。在构造模型之前，将数据集随机地分为训练数据集和测试数据集。先使用训练数据集来构造分类模型，然后使用测试数据集来评估模型的分类准确率。如果认为模型的准确率可以接受，就可以用该模型对其它数据元组进行分类。一般来说，测试阶段的代价远低于训练阶段。

分类算法基于不同的思想，算法也不尽相同，例如支持向量机 SVM、决策树算法、贝叶斯算法、KNN 算法等。`spark.mllib` 包支持各种分类方法，主要包含二分类，多分类和回归分析。下表列出了每种类型的问题支持的算法。

问题类型	支持的方法
二分类	线性支持向量机，Logistic 回归，决策树，随机森林，梯度上升树，朴素贝叶斯
多类分类	Logistic 回归，决策树，随机森林，朴素贝叶斯
回归	线性最小二乘法，Lasso，岭回归，决策树，随机森林，梯度上升树，isotonic regression

其中 `spark.mllib` 包支持的算法较为完善，也正逐步迁移到 `spark.ml` 包中。本节将介绍 `spark.ml` 包中一些典型的分类算法。

1、构建逻辑斯蒂回归分类器

逻辑斯蒂回归是统计学习中的经典分类方法，属于对数线性模型。

logistic 回归的因变量可以是二分类的，也可以是多分类的。

我们以 `iris` 数据集为例进行分析。`iris` 以鸢尾花的特征作为数据来源，数据集包含 150 个数据集，分为 3 类，每类 50 个数据，每个数据包含 4 个属性，是在数据挖掘、数据分类中非常常用的测试集、训练集。为了便于理解，我们这里主要用后两个属性（花瓣的长度和宽度）来进行分类。目前 `spark.ml` 中支持二分类和多分类，我们将分别从“用二项逻辑斯蒂回归来解决二分类问题”、“用多项逻辑斯蒂回归来解决二分类问题”、“用多项逻辑斯蒂回归来解决多分类问题”三个方面进行分析。

首先我们先取其中的后两类数据，用二项逻辑斯蒂回归进行二分类分析。

(1) 导入需要的包：

```
1. from pyspark.sql import Row,functions  
2. from pyspark.ml.linalg import Vector,Vectors  
3. from pyspark.ml.evaluation import MulticlassClassificationEvaluator  
4. from pyspark.ml import Pipeline
```

```
5. from pyspark.ml.feature import IndexToString, StringIndexer, VectorIndexer, HashingTF, Tokenizer  
6. from pyspark.ml.classification import LogisticRegression, LogisticRegressionModel, BinaryLogisticRegressionSummary, LogisticRegression
```

(2) 读取数据，简要分析：

我们定制一个函数，来返回一个指定的数据，然后读取文本文件，第一个 map 把每行的数据用“,”隔开，比如在我们的数据集中，每行被分成了 5 部分，前 4 部分是鸢尾花的 4 个特征，最后一部分是鸢尾花的分类；我们这里把特征存储在 Vector 中，创建一个 Iris 模式的 RDD，然后转化成 dataframe；最后调用 show()方法来查看一下部分数据。

```
1. def f(x):  
2.     rel = {}  
3.     rel['features'] = Vectors.dense(float(x[0]), float(x[1]), float(x[2]), float(x[3]))  
4.     rel['label'] = str(x[4])  
5.     return rel  
6.
```

```
7. data = spark.sparkContext.textFile("file:///usr/local/spark/iris.txt").map(lambda line: line.split(',')) .map(lambda p: Row(*f(p))).toDF()

8.

9. +-----+-----+
10. |      features|      label|
11. +-----+-----+
12. | [5.1,3.5,1.4,0.2]| Iris-setosa|
13. | [4.9,3.0,1.4,0.2]| Iris-setosa|
14. | [4.7,3.2,1.3,0.2]| Iris-setosa|
15. . . .
16. +-----+-----+
17. only showing top 20 rows
```

因为我们现在处理的是 2 分类问题，所以我们不需要全部的 3 类数据，我们要从中选出两类的数据。这里首先把刚刚得到的数据注册成一个表 `iris`，注册成这个表之后，我们就可以通过 `sql` 语句进行数据查询，比如我们这里选出了所有不属于“`Iris-setosa`”类别的数据；选出我们需要的数据后，我们可以把结果打印出来看一下，这时就已经没有“`Iris-setosa`”类别的数据。

```
1. data.createOrReplaceTempView("iris")

2. df = spark.sql("select * from iris where label != 'Iris-setosa'")
```

```
3. rel = df.rdd.map(lambda t : str(t[1])+":" +str(t[0])).collect()

4. for item in rel:

5.     print(item)

6.

7. Iris-versicolor:[7.0,3.2,4.7,1.4]

8. Iris-versicolor:[6.4,3.2,4.5,1.5]

9. ...
```

(3) 构建 ML 的 pipeline

分别获取标签列和特征列，进行索引，并进行了重命名。

```
1. labelIndexer = StringIndexer().setInputCol("label"
    ).setOutputCol("indexedLabel").fit(df)

2. featureIndexer = VectorIndexer().setInputCol("features")
    .setOutputCol("indexedFeatures").fit(df)

3. featureIndexer: org.apache.spark.ml.feature.VectorIndexerModel = vecIdx_53b988077b38
```

接下来，我们把数据集随机分成训练集和测试集，其中训练集占 70%。

```
1. trainingData, testData = df.randomSplit([0.7,0.
    3])
```

然后，我们设置 logistic 的参数，这里我们统一用 setter 的方法来设置，也可以用 ParamMap 来设置（具体的可以查看 spark mllib 的官网）。这里我们设置了循环次数为 10 次，正则化项为 0.3 等，具体的可以设置的参数可以通过 explainParams() 来获取，还能看到我们已经设置的参数的结果。

```
1. lr = LogisticRegression().setLabelCol("indexedLabel")
   .setFeaturesCol("indexedFeatures").setMaxIter(10)
   .setRegParam(0.3).setElasticNetParam(0.8)

2. print("LogisticRegression parameters:\n" + lr.explainParams())

3.

4. LogisticRegression parameters:

5. aggregationDepth: suggested depth for treeAggregation (>= 2). (default: 2)

6. elasticNetParam: the ElasticNet mixing parameter,
   in range [0, 1]. For alpha = 0, the penalty is an L2 penalty. For alpha = 1, it is an L1 penalty.
   (default: 0.0, current: 0.8)

7. family: The name of family which is a description of the label distribution to be used in the model.
   Supported options: auto, binomial, multinomial
   (default: auto)
```

```
8. featuresCol: features column name. (default: features, current: indexedFeatures)

9. fitIntercept: whether to fit an intercept term. (default: True)

10. labelCol: label column name. (default: label, current: indexedLabel)

11. maxIter: max number of iterations (>= 0). (default: 100, current: 10)

12. predictionCol: prediction column name. (default: prediction)

13. probabilityCol: Column name for predicted class conditional probabilities. Note: Not all models output well-calibrated probability estimates! These probabilities should be treated as confidences, not precise probabilities. (default: probability)

14. rawPredictionCol: raw prediction (a.k.a. confidence) column name. (default: rawPrediction)

15. regParam: regularization parameter (>= 0). (default: 0.0, current: 0.3)
```

16. standardization: whether to standardize the training features before fitting the model. (default: `True`)
17. threshold: `Threshold` in binary classification prediction, `in` range `[0, 1]`. `If` `threshold` and `thresholds` are both set, they must match.e.g. `if threshold is p, then thresholds must be equal to [1-p, p]`. (default: `0.5`)
18. thresholds: `Thresholds` in multi-class classification to adjust the probability of predicting each class. `Array` must have length equal to the number of classes, `with` values `> 0`, excepting that at most one value may be `0`. The class with largest value p/t `is` predicted, where p `is` the original probability of that class and t `is` the class's threshold. (undefined)
19. tol: the convergence tolerance for iterative algorithms (`>= 0`). (default: `1e-06`)
20. weightCol: weight column name. If this is not set or empty, we treat all instance weights as 1.0. (undefined)
- 21.

这里我们设置一个 `labelConverter`, 目的是把预测的类别重新转化成字符型的。

```
1. labelConverter = IndexToString().setInputCol("prediction").setOutputCol("predictedLabel").setLabelS(labelIndexer.labels)
```

构建 `pipeline`, 设置 `stage`, 然后调用 `fit()` 来训练模型。

```
1. lrPipeline = Pipeline().setStages([labelIndexer, featureIndexer, lr, labelConverter])  
2. lrPipelineModel = lrPipeline.fit(trainingData)
```

`pipeline` 本质上是一个 `Estimator`, 当 `pipeline` 调用 `fit()` 的时候就产生了一个 `PipelineModel`, 本质上是一个 `Transformer`。然后这个 `PipelineModel` 就可以调用 `transform()` 来进行预测, 生成一个新的 `DataFrame`, 即利用训练得到的模型对测试集进行验证。

```
1. lrPredictions = lrPipelineModel.transform(testData)
```

最后我们可以输出预测的结果, 其中 `select` 选择要输出的列, `collect` 获取所有行的数据, 用 `foreach` 把每行打印出来。其中打印出来的值依次分别代表该行数据的真实分类和特征值、预测属于不同分类的概率、预测的分类。

```
1. preRel = lrPredictions.select("predictedLabel", "label", "features", "probability").collect()
```

```
2. for item in preRel:  
3.     print(str(item['label'])+', '+str(item['features'])+'-->prob=' +str(item['probability'])+',predictedLabel'+str(item['predictedLabel']))  
4.  
5. Iris-versicolor,[5.2,2.7,3.9,1.4]-->prob=[0.47412  
5433289,0.525874566711],predictedLabelIris-virgin  
ica  
6. Iris-versicolor,[5.5,2.3,4.0,1.3]-->prob=[0.49872  
4224708,0.501275775292],predictedLabelIris-virgin  
ica  
7. Iris-versicolor,[5.6,3.0,4.5,1.5]-->prob=[0.45665  
9495584,0.543340504416],predictedLabelIris-virgin  
ica  
8....
```

(4) 模型评估

创建一个 MulticlassClassificationEvaluator 实例，用 `set` 方法把预测分类的列名和真实分类的列名进行设置；然后计算预测准确率和错误率。

```
1. evaluator = MulticlassClassificationEvaluator().s  
etLabelCol("indexedLabel").setPredictionCol("pred  
iction")
```

```
2. lrAccuracy = evaluator.evaluate(lrPredictions)  
3. print("Test Error = " + str(1.0 - lrAccuracy))  
4. Test Error = 0.3511111111111115
```

从上面可以看到预测的准确性达到 65%，接下来我们可以通过 `model` 来获取我们训练得到的逻辑斯蒂模型。前面已经说过 `model` 是一个 `PipelineModel`，因此我们可以通过调用它的 `stages` 来获取模型，具体如下：

```
1. lrModel = lrPipelineModel.stages[2]  
2. print("Coefficients: " + str(lrModel.coefficient  
s)+"Intercept: "+str(lrModel.intercept)+"numClass  
es: "+str(lrModel.numClasses)+"numFeatures: "+str  
(lrModel.numFeatures))  
3. Coefficients: [-0.0396171957643483,0.0,0.0,0.0724  
0315639651046]Intercept: -0.23127346342015379numC  
lasses: 2numFeatures: 4
```

(5) 模型评估

spark 的 `ml` 库还提供了一个对模型的摘要总结（`summary`），不过目前只支持二项逻辑斯蒂回归，而且要显示转化成 `Binary LogisticRegressionSummary`。在下面的代码中，首先获得二项逻辑斯模型的摘要；然后获得 10 次循环中损失函数的变化，并将结果打印出来，可以看到损失函数随着循环是逐渐变小的，损失函数越小，模型就越好；接下来，我们把摘要强制转化为 `Binary`

`LogisticRegressionSummary`，来获取用来评估模型性能的矩阵；通过获取 ROC，我们可以判断模型的好坏，`areaUnderROC` 达到了 0.969551282051282，说明我们的分类器还是不错的；最后，我们通过最大化 `fMeasure` 来选取最合适的阈值，其中 `fMeasure` 是一个综合了召回率和准确率的指标，通过最大化 `fMeasure`，我们可以选取到用来分类的最合适的阈值。

```
1. trainingSummary = lrModel.summary  
2. objectiveHistory = trainingSummary.objectiveHistory  
3. for item in objectiveHistory:  
4. ...     print(item)  
5. ....  
6. 0.6930582890371242  
7. 0.6899151958544979.  
8. ....  
9. print(trainingSummary.areaUnderROC)  
10. 0.9889758179231863  
11.  
12. fMeasure = trainingSummary.fMeasureByThreshold  
13.  
14. maxFMeasure = fMeasure.select(functions.max("F-M  
easure")).head()[0]
```

```
15. 0.9599999999999999
16.
17. bestThreshold = fMeasure.where(fMeasure["F-Measure"] == maxFMeasure).select("threshold").head()[0]
18. 0.5487261156903904
19.
20. lr.setThreshold(bestThreshold)
```

2、构建决策树分类器

决策树是一种基本的分类与回归方法，这里主要介绍用于分类的决策树。决策树模式呈树形结构，其中每个内部节点表示一个属性上的测试，每个分支代表一个测试输出，每个叶节点代表一种类别。学习时利用训练数据，根据损失函数最小化的原则建立决策树模型；预测时，对新的数据，利用决策树模型进行分类。

我们以 `iris` 数据集为例进行分析。`iris` 以鸢尾花的特征作为数据来源，数据集包含 150 个数据集，分为 3 类，每类 50 个数据，每个数据包含 4 个属性，是在数据挖掘、数据分类中非常常用的测试集、训练集。决策树可以用于分类和回归，接下来我们将在代码中分别进行介绍。

(1) 导入需要的包

```
1. from pyspark.ml.linalg import Vector, Vectors
```

```
2. from pyspark.sql import Row  
3. from pyspark.ml import Pipeline  
4. from pyspark.ml.feature import IndexToString,Stri  
ngIndexer,VectorIndexer
```

(2)读取数据，简要分析：

读取文本文件，第一个 `map` 把每行的数据用“,”隔开，比如在我们的数据集中，每行被分成了 5 部分，前 4 部分是鸢尾花的 4 个特征，最后一部分是鸢尾花的分类；我们这里把特征存储在 `Vector` 中，创建一个 `Iris` 模式的 `RDD`，然后转化成 `dataframe`；然后把刚刚得到的数据注册成一个表 `iris`，注册成这个表之后，我们就可以通过 `sql` 语句进行数据查询；选出我们需要的数据后，我们可以把结果打印出来查看一下数据。

```
1. def f(x):  
2.     rel = {}  
3.     rel['features'] = Vectors.dense(float(x[0]),f  
loat(x[1]),float(x[2]),float(x[3]))  
4.     rel['label'] = str(x[4])  
5.     return rel
```

```
6. data = spark.sparkContext.textFile("file:///usr/local/spark/iris.txt").map(lambda line: line.split( ',' )).map(lambda p: Row(**f(p))).toDF()

7. data.createOrReplaceTempView("iris")

8. df = spark.sql("select * from iris")

9. rel = df.rdd.map(lambda t : str(t[1])+"："+str(t[0])).collect()

10. for item in rel:

11.     print(item)

12. Iris-setosa:[5.1,3.5,1.4,0.2]

13. Iris-setosa:[4.9,3.0,1.4,0.2]

14. .....
```

(3)进一步处理特征和标签，以及数据分组：

```
1. //分别获取标签列和特征列，进行索引，并进行了重命名。

2. labelIndexer = StringIndexer().setInputCol("label")
   .setOutputCol("indexedLabel").fit(df)

3. featureIndexer = VectorIndexer().setInputCol("features")
   .setOutputCol("indexedFeatures").setMaxCategories(4).fit(df)

4. //这里我们设置一个 labelConverter，目的是把预测的类别
   重新转化成字符型的。
```

```
5. labelConverter = IndexToString().setInputCol("prediction").setOutputCol("predictedLabel").setLabels(labelIndexer.labels)

6. //接下来，我们把数据集随机分成训练集和测试集，其中训练集占 70%。

7. trainingData, testData = data.randomSplit([0.7, 0.3])
```

(4) 构建决策树分类模型

```
1. //导入所需要的包

2. from pyspark.ml.classification import DecisionTreeClassificationModel,DecisionTreeClassifier

3. from pyspark.ml.evaluation import MulticlassClassificationEvaluator

4. //训练决策树模型，这里我们可以通过 setter 的方法来设置决策树的参数，也可以用 ParamMap 来设置（具体的可以查看 spark mllib 的官网）。具体的可以设置的参数可以通过 explainParams() 来获取。

5. dtClassifier = DecisionTreeClassifier().setLabelCol("indexedLabel").setFeaturesCol("indexedFeatures")

6. //在 pipeline 中进行设置
```

```
7. pipelinedClassifier = Pipeline().setStages([label
    Indexer, featureIndexer, dtClassifier, labelConve
    rter])

8. //训练决策树模型

9. modelClassifier = pipelinedClassifier.fit(trainin
    gData)

10. //进行预测

11. predictionsClassifier = modelClassifier.transfor
    m(testData)

12. //查看部分预测的结果

13. predictionsClassifier.select("predictedLabel", "
    label", "features").show(20)

14. +-----+-----+-----+
   |-----+-----+-----+
   | predictedLabel |      label |      feature
   |-----+-----+-----+
   | s |
```

predictedLabel	label	feature
s		

```
15. +-----+-----+-----+
   |-----+-----+-----+
   | Iris-setosa | Iris-setosa | [4.3,3.0,1.1,0.
   |-----+-----+-----+
   | 1] |
```

Iris-setosa	Iris-setosa	[4.3,3.0,1.1,0.1]
Iris-setosa	Iris-setosa	[4.3,3.0,1.1,0.1]

```
16. +-----+-----+-----+
   |-----+-----+-----+
   | Iris-versicolor | Iris-versicolor | [6.1,2.9,4.7,1.
   |-----+-----+-----+
   | 4] |
```

Iris-versicolor	Iris-versicolor	[6.1,2.9,4.7,1.4]
Iris-versicolor	Iris-versicolor	[6.1,2.9,4.7,1.4]

```
19. . . . .
20. +-----+-----+
21. only showing top 20 rows
```

(5) 评估决策树分类模型

```
1. evaluatorClassifier = MulticlassClassificationEvaluator()
   .setLabelCol("indexedLabel").setPredictionCol("prediction")
   .setMetricName("accuracy")

2. accuracy = evaluatorClassifier.evaluate(predictionClassifier)

3. print("Test Error = " + str(1.0 - accuracy))

4. Test Error = 0.05882352941176472

5. treeModelClassifier = modelClassifier.stages[2]

6. print("Learned classification tree model:\n" + str(treeModelClassifier.toDebugString()))

7. Learned classification tree model:

8. DecisionTreeClassificationModel (uid=DecisionTree
   Classifier_4e57b26beacfd363271a) of depth 3 with
   7 nodes

9. If (feature 2 <= 1.9)
10.   Predict: 2.0
11. Else (feature 2 > 1.9)
12.   If (feature 3 <= 1.6)
```

```
13.      If (feature 2 <= 4.9)
14.          Predict: 0.0
15.      Else (feature 2 > 4.9)
16.          Predict: 1.0
17.      Else (feature 3 > 1.6)
18.          Predict: 1.0
```

从上述结果可以看到模型的预测准确率为 0.94 以及训练的决策树模型结构。

第十次上机实验内容

1、KMeans 聚类算法

KMeans 是一个迭代求解的聚类算法，其属于划分型的聚类方法，即首先创建 K 个划分，然后迭代地将样本从一个划分转移到另一个划分来改善最终聚类的质量。

ML 包下的 KMeans 方法位于 org.apache.spark.ml.clustering 包下，其过程大致如下：

1. 根据给定的 k 值，选取 k 个样本点作为初始划分中心；
2. 计算所有样本点到每一个划分中心的距离，并将所有样本点划分到距离最近的划分中心；
3. 计算每个划分中样本点的平均值，将其作为新的中心；

循环进行 2~3 步直至达到最大迭代次数，或划分中心的变化小于某一预定义阈值。

在使用前，引入需要的包：

```
1. from pyspark.sql import Row  
2. from pyspark.ml.clustering import KMeans, KMeansModel  
3. from pyspark.ml.linalg import Vectors
```

下文中，我们默认名为 spark 的 SparkSession 已经创建。

```
1. rawData = sc.textFile("file:///usr/local/spark/iris.txt")  
2. def f(x):  
3.     rel = {}  
4.     rel['features'] = Vectors.dense(float(x[0]), float(x[1]), float(x[2]), float(x[3]))  
5.     return rel  
6.  
7. df = sc.textFile("file:///usr/local/spark/iris.txt").map(lambda line: line.split(',')).map(lambda p: Row(**f(p))).toDF()
```

在得到数据后，我们即可通过 ML 包的固有流程：创建 Estimator 并调用其 fit() 方法来生成相应的 Transformer 对象，很显然，在这里 KMeans 类是 Estimator，而用于保存训练后模型的 Kmeans Model 类则属于 Transformer：

```
1. kmeansmodel = KMeans().setK(3).setFeaturesCol('features').setPredictionCol('prediction').fit(df)
```

与 MLlib 版本类似，ML 包下的 KMeans 方法也有 Seed（随机数种子）、Tol（收敛阈值）、K（簇个数）、MaxIter（最大迭代次数）、initMode（初始化方式）、initStep（KMeans 方法的步数）

等参数可供设置，和其他的 ML 框架算法一样，用户可以通过相应的 setXXX() 方法来进行设置，或以 ParamMap 的形式传入参数，这里为了简介期间，使用 setXXX() 方法设置了参数 K，其余参数均采用默认值。

与 MLlib 中的实现不同，KMeansModel 作为一个 Transformer，不再提供 predict() 样式的方法，而是提供了一致性的 transform() 方法，用于将存储在 DataFrame 中的给定数据集进行整体处理，生成带有预测簇标签的数据集：

```
1. results = kmeansmodel.transform(df).collect()  
2. for item in results:  
3. ...     print(str(item[0])+' is predicted as cluster'+  
        str(item[1]))  
4. ...  
5. [5.1, 3.5, 1.4, 0.2] is predicted as cluster1  
6. [4.9, 3.0, 1.4, 0.2] is predicted as cluster1  
7. [4.7, 3.2, 1.3, 0.2] is predicted as cluster1  
8. .....  
9. [5.8, 4.0, 1.2, 0.2] is predicted as cluster1
```

也可以通过 KMeansModel 类自带的 clusterCenters 属性获取到模型的所有聚类中心情况：

```
1. results2 = kmeansmodel.clusterCenters()  
2. for item in results2:  
3. ...     print(item)  
4. ...  
5. [ 5.9016129   2.7483871   4.39354839   1.43387097]  
6. [ 5.006   3.418   1.464   0.244]  
7. [ 6.85       3.07368421   5.74210526   2.07105263]  
8.
```

与 MLlib 下的实现相同， KMeansModel 类也提供了计算集合内误差平方和的方法来度量聚类的有效性，在真实 K 值未知的情况下，该值的变化可以作为选取合适 K 值的一个重要参考：

```
1. kmeansmodel.computeCost(data)  
2. 78.94084142614622
```

2、高斯混合模型(GMM)聚类算法

高斯混合模型是一种概率式的聚类方法，属于生成式模型，它假设所有的数据样本都是由某一个给定参数的多元高斯分布所生成的，其过程大致如下：

1. 根据给定的 K 值，初始化 K 个多元高斯分布以及其权重；

2. 根据贝叶斯定理，估计每个样本由每个成分生成的后验概率；
3. 根据均值，协方差的定义以及 2 步求出的后验概率，更新均值向量、协方差矩阵和权重；
重复 2~3 步，直到似然函数增加值已小于收敛阈值，或达到最大迭代次数。

当参数估计过程完成后，对于每一个样本点，根据贝叶斯定理计算出其属于每一个簇的后验概率，并将样本划分到后验概率最大的簇上去。相对于 KMeans 等直接给出样本点的簇划分的聚类方法，GMM 这种给出样本点属于每个簇的概率的聚类方法，被称为软聚类。

在使用前，引入需要的包：

```
1. from pyspark.sql import Row  
2. from pyspark.ml.clustering import GaussianMixture, GaussianMixtureModel  
3. from pyspark.ml.linalg import Vectors
```

为了便于生成相应的 DataFrame，我们定义一个函数，来生成我们想要的数据。

```
1. def f(x):  
2.     rel = {}
```

```
3.     rel['features'] = Vectors.dense(float(x[0]), float(x[1]), float(x[2]), float(x[3]))  
4.     return rel
```

在定义数据类型完成后，即可将数据读入 RDD[model_instance] 的结构中，并通过 RDD 的隐式转换 toDF() 方法完成 RDD 到 DataFrame 的转换：

```
1. df = sc.textFile("file:///usr/local/spark/iris.txt").map(lambda line: line.split(',')).map(lambda p: Row(*p)).toDF()
```

可以通过创建一个 GaussianMixture 类，设置相应的超参数，并调用 fit(..) 方法来训练一个 GMM 模型 GaussianMixtureModel

这里，我们建立一个简单的 GaussianMixture 对象，设定其聚类数目为 3，其他参数取默认值。

```
1. gm = GaussianMixture().setK(3).setPredictionCol("Prediction").setProbabilityCol("Probability")  
2. gmm = gm.fit(df)
```

和 KMeans 等硬聚类方法不同的是，除了可以得到对样本的聚簇归属预测外，还可以得到样本属于各个聚簇的概率（这里我们存在 "Probability" 列中）。

调用 `transform()`方法处理数据集之后，打印数据集，可以看到每一个样本的预测簇以及其概率分布向量：

```
1. result = gmm.transform(df)
2. result.show(150, False)

3.

4. +-----+-----+
   |       |
   +-----+-----+
5. |features      |Prediction|Probability
   |             |          |
   +-----+-----+
6. +-----+-----+
   |       |
   +-----+-----+
7. | [5.1, 3.5, 1.4, 0.2] | 2 | [7.6734208374482E-12, 3.8
   |                         | 34611512428377E-17, 0.9999999999923265] |
   |                         |
8. ..... |

9. | [5.0, 3.4, 1.5, 0.2] | 2 | [1.5177480903927398E-10,
   |                         | 3.3953493957830917E-17, 0.9999999998482252] |
   |                         |
10. | [4.4, 2.9, 1.4, 0.2] | 2 | [8.259607931046402E-7,
    |                         | 1.383366216513306E-16, 0.9999991740392067] |
```

得到模型后，即可查看模型的相关参数，与 KMeans 方法不同，GMM 不直接给出聚类中心，而是给出各个混合成分（多元高斯分

布) 的参数。在 ML 的实现中, GMM 的每一个混合成分都使用一个 MultivariateGaussian 类 (位于 org.apache.spark.ml.stat.distribution 包) 来存储, 我们可以使用 GaussianMixtureModel 类的 weights 成员获取到各个混合成分的权重, 使用 gaussians 成员来获取到各个混合成分的参数 (均值向量和协方差矩阵) :

```
1. for i in range(3):  
2.     print("Component "+str(i)+" : weight is "+str(gmm.  
weights[i])+"\n mu vector is "+str( gmm.gaussiansDF.se  
lect(' mean').head())+"\n sigma matrix is "+ str(gmm.g  
aussiansDF.select(' cov').head()))  
3.  
4. ...  
5. Component 0 : weight is 0.6537361109963744  
6. mu vector is Row(mean=DenseVector([6.2591, 2.8764, 4.  
9057, 1.6784]))  
7. sigma matrix is Row(cov=DenseMatrix(4, 4, [0.4414, 0.  
1249, 0.4524, 0.1676, 0.1249, 0.1103, 0.1469, 0.081,  
0.4524, 0.1469, 0.6722, 0.2871, 0.1676, 0.081, 0.2871,  
0.1806], False))  
8. Component 1 : weight is 0.03291269344724756
```

```
9. mu vector is Row(mean=DenseVector([6.2591, 2.8764, 4.  
9057, 1.6784]))  
  
10. sigma matrix is Row(cov=DenseMatrix(4, 4, [0.4414,  
0.1249, 0.4524, 0.1676, 0.1249, 0.1103, 0.1469, 0.081,  
0.4524, 0.1469, 0.6722, 0.2871, 0.1676, 0.081, 0.287  
1, 0.1806], False))  
  
11. Component 2 : weight is 0.31335119555637797  
  
12. mu vector is Row(mean=DenseVector([6.2591, 2.8764,  
4.9057, 1.6784]))  
  
13. sigma matrix is Row(cov=DenseMatrix(4, 4, [0.4414,  
0.1249, 0.4524, 0.1676, 0.1249, 0.1103, 0.1469, 0.081,  
0.4524, 0.1469, 0.6722, 0.2871, 0.1676, 0.081, 0.287  
1, 0.1806], False))
```

3、协同过滤算法

协同过滤是一种基于一组兴趣相同的用户或项目进行的推荐，它根据邻居用户（与目标用户兴趣相似的用户）的偏好信息产生对目标用户的推荐列表。协同过滤算法主要分为基于用户的协同过滤算法和基于项目的协同过滤算法。MLlib 当前支持基于模型的协同过滤，其中用户和商品通过一小组隐语义因子进行表达，并且这些因子也用于预测缺失的元素。Spark MLlib 实现了交替最小二乘法来学习这些隐

性语义因子。

下面的例子中，我们将获取 MovieLens 数据集，其中每行包含一个用户、一个电影、一个该用户对该电影的评分以及时间戳。我们使用默认的 ALS.train() 方法，即显性反馈（默认 implicitPrefs 为 false）来构建推荐模型并根据模型对评分预测的均方根误差来对模型进行评估。

导入需要的包

```
1. from pyspark.ml.evaluation import RegressionEvaluator  
2. from pyspark.ml.recommendation import ALS  
3. from pyspark.sql import Row
```

根据数据结构创建读取规范

创建一个函数，返回即 [Int, Int, Float, Long] 的对象

```
1. def f(x):  
2.     rel = {}  
3.     rel['userId'] = int(x[0])  
4.     rel['movieId'] = int(x[1])  
5.     rel['rating'] = float(x[2])  
6.     rel['timestamp'] = float(x[3])  
7.     return rel
```

读取数据

```
1. ratings = sc.textFile("file:///usr/local/spark/data/ml  
lib/als/sample_movielens_ratings.txt").map(lambda lin  
e: line.split('::')).map(lambda p: Row(**f(p))).toDF()
```

然后把数据打印出来：

```
1. ratings.show()  
2. +-----+-----+-----+-----+  
3. |movieId|rating|    timestamp|userId|  
4. +-----+-----+-----+-----+  
5. |      2|    3.0|1.424380312E9|      0|  
6. |      3|    1.0|1.424380312E9|      0|  
7. .....  
8. |      37|    1.0|1.424380312E9|      0|  
9. |      41|    2.0|1.424380312E9|      0|  
10. +-----+-----+-----+-----+  
11. only showing top 20 rows  
12.
```

构建模型

把 MovieLens 数据集划分训练集和测试集

```
1. training, test = ratings.randomSplit([0.8, 0.2])
```

使用 ALS 来建立推荐模型，这里我们构建了两个模型，一个是显性反馈，一个是隐性反馈

```
1. alsExplicit = ALS(maxIter=5, regParam=0.01, userCol="userId", itemCol="movieId", ratingCol="rating")  
2. alsImplicit = ALS(maxIter=5, regParam=0.01, implicitPrefs=True, userCol="userId", itemCol="movieId", ratingCol="rating")
```

接下来，把推荐模型放在训练数据上训练：

```
1. modelExplicit = alsExplicit.fit(training)  
2. modelImplicit = alsImplicit.fit(training)
```

模型预测

使用训练好的推荐模型对测试集中的用户商品进行预测评分，得到预测评分的数据集

```
1. predictionsExplicit = modelExplicit.transform(test)  
2. predictionsImplicit = modelImplicit.transform(test)
```

我们把结果输出，对比一下真实结果与预测结果：

```
1. predictionsExplicit.show()

2.

3. +-----+-----+-----+-----+
   | movieId|rating| timestamp|userId| prediction|
   +-----+-----+-----+-----+
6. |      31|    4.0|1.424380312E9|     12|  1.6642745|
7. .....
8. |      34|    1.0|1.424380312E9|     14| -0.2330051|
9. |      81|    3.0|1.424380312E9|     26|  5.684144|
10. +-----+-----+-----+-----+
11. only showing top 20 rows
12. predictionsImplicit.show()

13.

14. +-----+-----+-----+-----+
   | movieId|rating| timestamp|userId| prediction|
   +-----+-----+-----+-----+
17. |      31|    4.0|1.424380312E9|     12| 0.012495369|
18. |      .....
19. |      34|    1.0|1.424380312E9|     14| 0.116244696|
20. |      81|    3.0|1.424380312E9|     26|  0.3299607|
```

```
21. +-----+-----+-----+-----+-----+
```

```
22. only showing top 20 rows
```

模型评估

通过计算模型的均方根误差来对模型进行评估，均方根误差越小，模型越准确：

```
1. evaluator = RegressionEvaluator().setMetricName("rmse")
   .setLabelCol("rating").setPredictionCol("prediction")
```

打印出两个模型的均方根误差：

```
1. print("Explicit:Root-mean-square error = "+str(rmseExplicit))
2. Explicit:Root-mean-square error = 1.9093067340213505
3. print("Implicit:Root-mean-square error = "+str(rmseImplicit))
4. Implicit:Root-mean-square error = 1.9965886603519194
```

第十一、十二次上机实验内容

这两次上机课，需要各位综合上述所学完成一个实际案例——淘宝双十一数据分析与预测。

1、准备实验环境

(1) MySQL 安装:

使用以下命令即可进行 mysql 安装，注意安装前先更新一下软件源以获得最新版本：

```
1. sudo apt-get update #更新软件源  
2. sudo apt-get install mysql-server #安装mysql
```

启动和关闭 mysql 服务器：

```
1. service mysql start  
2. service mysql stop
```

确认是否启动成功，mysql 节点处于 LISTEN 状态表示启动成功：

```
3. sudo netstat -tap | grep mysql
```

```
dblab@ubuntu:~$ sudo netstat -tap | grep mysql
tcp        0      0 localhost:mysql          *:*                  LISTEN
6350/mysqld
```

进入 mysql shell 界面:

```
1. mysql -u root -p
```

导致导入时中文乱码的原因是 `character_set_server` 默认设置是 `latin1`, 如下图:

```
mysql> show variables like "char%";  
+-----+-----+  
| Variable_name      | Value |  
+-----+-----+  
| character_set_client | utf8  |  
| character_set_connection | utf8  |  
| character_set_database | utf8  |  
| character_set_filesystem | binary |  
| character_set_results | utf8  |  
| character_set_server | latin1 |←  
| character_set_system | utf8  |  
| character_sets_dir   | /usr/share/mysql/charsets/ |  
+-----+-----+  
8 rows in set (0.00 sec)
```

可以单个设置修改编码方式 `set character_set_server=utf8;`
但是重启会失效

(2) HIVE 安装:

下载 hive 安装包

```
1. sudo tar -zxvf ./apache-hive-1.2.1-bin.tar.gz -C  
/usr/local    # 解压到 /usr/local 中  
  
2. cd /usr/local/  
  
3. sudo mv apache-hive-1.2.1-bin hive          # 将文件  
夹名改为 hive  
  
4. sudo chown -R hadoop:hadoop hive           # 修改  
文件权限
```

为了方便使用，我们把 `hive` 命令加入到环境变量中去，请使用 `vim` 编辑器打开 `.bashrc` 文件，命令如下：

```
1. vim ~/.bashrc
```

在该文件最前面一行添加如下内容：

```
export HIVE_HOME=/usr/local/hive  
export PATH=$PATH:$HIVE_HOME/bin  
export HADOOP_HOME=/usr/local/hadoop
```

`HADOOP_HOME` 需要被配置成你机器上 Hadoop 的安装路径，比如这里是安装在 `/usr/local./hadoop` 目录。

保存退出后，运行如下命令使配置立即生效：

```
1. source ~/.bashrc
```

修改 `/usr/local/hive/conf` 下的 `hive-site.xml`

执行如下命令：

```
1. cd /usr/local/hive/conf  
2. mv hive-default.xml.template hive-default.xml
```

上面命令是将 `hive-default.xml.template` 重命名为 `hive-default.xml`；

然后，使用 `vim` 编辑器新建一个配置文件 `hive-site.xml`，命令如下：

```
1. cd /usr/local/hive/conf  
2. vim hive-site.xml
```

在 `hive-site.xml` 中添加如下配置信息：

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>  
  
<?xml-stylesheet type="text/xsl" href="configuration.x  
sl"?>  
  
<configuration>  
  
<property>  
  
    <name>javax.jdo.option.ConnectionURL</name>  
  
    <value>jdbc:mysql://localhost:3306/hive?createData  
baseIfNotExist=true</value>  
  
    <description>JDBC connect string for a JDBC metast  
ore</description>
```

```
</property>

<property>

    <name>javax.jdo.option.ConnectionDriverName</name>

    <value>com.mysql.jdbc.Driver</value>

    <description>Driver class name for a JDBC metastor
e</description>

</property>

<property>

    <name>javax.jdo.option.ConnectionUserName</name>

    <value>hive</value>

    <description>username to use against metastore dat
abase</description>

</property>

<property>

    <name>javax.jdo.option.ConnectionPassword</name>

    <value>hive</value>
```

```
<description>password to use against metastore dat  
abase</description>  
  
</property>  
  
</configuration>
```

然后，按键盘上的“**ESC**”键退出 **vim** 编辑状态，再输入:**wq**，保存并退出 **vim** 编辑器。

(3) 安装并配置 mysql:

下载 mysql jdbc 包

```
1. tar -zxvf mysql-connector-java-5.1.40.tar.gz #  
解压  
2. cp mysql-connector-java-5.1.40/mysql-connector-ja  
va-5.1.40-bin.jar /usr/local/hive/lib #将mysql-c  
onnector-java-5.1.40-bin.jar 拷贝到/usr/local/hive  
/lib 目录下
```

启动并登陆 mysql shell

```
1. service mysql start #启动mysql 服务  
2. mysql -u root -p #登陆shell 界面
```

新建 hive 数据库

```
1. mysql> create database hive;      #这个hive 数据库与  
          hive-site.xml 中localhost:3306/hive 的hive 对应, 用  
          来保存hive 元数据
```

配置 mysql 允许 hive 接入:

```
1. mysql> grant all on *.* to hive@localhost identif  
          ied by 'hive';    #将所有数据库的所有表的所有权限赋给  
          hive 用户, 后面的hive 是配置hive-site.xml 中配置的连  
          接密码  
2. mysql> flush privileges; #刷新mysql 系统权限关系表
```

启动 hive

启动 hive 之前, 请先启动 hadoop 集群。

```
1. start-all.sh #启动hadoop  
2. hive #启动hive
```

注意, 我们这里已经配置了 PATH, 所以, 不要把 start-all.sh 和
hive 命令的路径加上。如果没有配置 PATH, 请加上路径才能运行
命令, 比如, 本教程 Hadoop 安装目录是“/usr/local/hadoop”, Hive

的安装目录是“/usr/local/hive”，因此，启动 hadoop 和 hive，也可以使用下面带路径的方式：

```
1. cd /usr/local/hadoop  
2. ./sbin/start-all.sh  
3. cd /usr/local/hive  
4. ./bin/hive
```

```
dblab@ubuntu:/usr/local/hive$ hive
```

```
Logging initialized using configuration in jar:file:/usr/local/hive/lib/hive  
Thu Oct 13 20:01:26 CST 2016 WARN: Establishing SSL connection without serv  
.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established  
ng applications not using SSL the verifyServerCertificate property is set to  
SSL=false, or set useSSL=true and provide truststore for server certificate  
Thu Oct 13 20:01:26 CST 2016 WARN: Establishing SSL connection without serv  
.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established  
ng applications not using SSL the verifyServerCertificate property is set to  
SSL=false, or set useSSL=true and provide truststore for server certificate  
Thu Oct 13 20:01:26 CST 2016 WARN: Establishing SSL connection without serv  
.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established  
ng applications not using SSL the verifyServerCertificate property is set to  
SSL=false, or set useSSL=true and provide truststore for server certificate  
Thu Oct 13 20:01:26 CST 2016 WARN: Establishing SSL connection without serv  
.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established  
ng applications not using SSL the verifyServerCertificate property is set to  
SSL=false, or set useSSL=true and provide truststore for server certificate  
Thu Oct 13 20:01:26 CST 2016 WARN: Establishing SSL connection without serv  
.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established  
ng applications not using SSL the verifyServerCertificate property is set to  
SSL=false, or set useSSL=true and provide truststore for server certificate  
Thu Oct 13 20:01:27 CST 2016 WARN: Establishing SSL connection without serv  
.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established  
ng applications not using SSL the verifyServerCertificate property is set to  
SSL=false, or set useSSL=true and provide truststore for server certificate  
Thu Oct 13 20:01:27 CST 2016 WARN: Establishing SSL connection without serv  
.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established  
ng applications not using SSL the verifyServerCertificate property is set to  
SSL=false, or set useSSL=true and provide truststore for server certificate  
Thu Oct 13 20:01:27 CST 2016 WARN: Establishing SSL connection without serv  
.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established  
ng applications not using SSL the verifyServerCertificate property is set to  
SSL=false, or set useSSL=true and provide truststore for server certificate  
Thu Oct 13 20:01:27 CST 2016 WARN: Establishing SSL connection without serv  
.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established  
ng applications not using SSL the verifyServerCertificate property is set to  
SSL=false, or set useSSL=true and provide truststore for server certificate  
hive> show databases;
```

解决 Hive 启动，**Hive metastore database is not initialized** 的错误。出错原因：以前曾经安装了 Hive 或 MySQL，重新安装 Hive 和 MySQL 以后，导致版本、配置不一致。解决方法是，使用 schematool 工具。Hive 现在包含一个用于 Hive Metastore 架构操控的脱机工具，名为 `schematool`。此工具可用于初始化当前 Hive 版本的 Metastore 架构。此外，其还可处理从较旧版本到新版本的架构升级。所以，解决上述错误，你可以在终端执行如下命令：

```
1. schematool -dbType mysql -initSchema
```

执行后，再启动 Hive，应该就正常了。

启动进入 Hive 的交互式执行环境以后，会出现如下命令提示符：

```
1. hive>
```

可以在里面输入 SQL 语句，如果要退出 Hive 交互式执行环境，可以输入如下命令：

```
1. hive>exit;
```

(4) 安装 Sqoop

1. 下载并解压 sqoop1.4.6

下面执行以下命令

```
1. cd ~ #进入当前用户的用户目录  
2. cd 下载 #sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz  
      文件下载后就被保存在该目录下面  
3. sudo tar -zxvf sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz -C /usr/local #解压安装文件  
4. cd /usr/local  
5. sudo mv sqoop-1.4.6.bin__hadoop-2.0.4-alpha sqoop  
      #修改文件名  
6. sudo chown -R hadoop:hadoop sqoop #修改文件夹属主,  
      如果你当前登录用户名不是 hadoop, 请修改成你自己的用户名
```

2. 修改配置文件 sqoop-env.sh

```
1. cd sqoop/conf/  
2. cat sqoop-env-template.sh >> sqoop-env.sh #将sqoop-env-template.sh 复制一份并命名为sqoop-env.sh  
3. vim sqoop-env.sh #编辑sqoop-env.sh
```

修改 sqoop-env.sh 的如下信息

```
1. export HADOOP_COMMON_HOME=/usr/local/hadoop
```

```
2. export HADOOP_MAPRED_HOME=/usr/local/hadoop  
3. export HBASE_HOME=/usr/local/hbase  
4. export HIVE_HOME=/usr/local/hive  
5. #export ZOOCFGDIR= #如果读者配置了ZooKeeper, 也需要  
在此配置ZooKeeper 的路径
```

3. 配置环境变量

打开当前用户的环境变量配置文件:

```
1. vim ~/.bashrc
```

在配置文件第一行键入如下信息:

```
export SQOOP_HOME=/usr/local/sqoop  
export PATH=$PATH:$SBT_HOME/bin:$SQOOP_HOME/bin  
export CLASSPATH=$CLASSPATH:$SQOOP_HOME/lib
```

保存该文件，退出 vim 编辑器。

然后，执行下面命令让配置文件立即生效:

```
1. source ~/.bashrc
```

4. 将 mysql 驱动包拷贝到\$SQOOP_HOME/lib

下面要把 MySQL 驱动程序拷贝到\$SQOOP_HOME/lib 目录下，一般文件会被浏览器默认放置在当前用户的下载目录下，本教程采用 hadoop 用户登录 Linux 系统，因此，下载文件被默认放置在“/home/hadoop/下载”目录下面。

下面执行命令拷贝文件：

```
1. cd ~/下载 #切换到下载路径, 如果你下载的文件不在这个目录下, 请切换到下载文件所保存的目录  
2. sudo tar -zxvf mysql-connector-java-5.1.40.tar.gz #解压mysql 驱动包  
3. ls #这时就可以看到解压缩后得到的目录mysql-connector  
-java-5.1.40  
4. cp ./mysql-connector-java-5.1.40/mysql-connector-  
java-5.1.40-bin.jar /usr/local/sqoop/lib
```

5. 测试与 MySQL 的连接

首先请确保 mysql 服务已经启动了，如果没有启动，请执行下面命令启动：

```
1. service mysql start
```

然后就可以测试 sqoop 与 MySQL 之间的连接是否成功：

```
1. sqoop list-databases --connect jdbc:mysql://127.  
0.0.1:3306/ --username root -P
```

mysql 的数据库列表显示在屏幕上表示连接成功，如下图：

```
dblab@ubuntu:~/Downloads$ sqoop list-databases --connect jdbc:mysql://127.0.  
0.1:3306/ --username hive -P  
16/11/12 22:01:42 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6  
Enter password:  
16/11/12 22:01:45 INFO manager.MySQLManager: Preparing to use a MySQL streaming resultset.  
Sat Nov 12 22:01:46 CST 2016 WARN: Establishing SSL connection without server's identity verification is not recommended. According to MySQL 5.5.45+, 5.6.26+ and 5.7.6+ requirements SSL connection must be established by default if explicit option isn't set. For compliance with existing applications not using SSL the verifyServerCertificate property is set to 'false'. You need either to explicitly disable SSL by setting useSSL=false, or set useSSL=true and provide truststore for server certificate verification.  
information_schema  
dblab  
hive  
mysql  
performance_schema  
sys  
test  
dblab@ubuntu:~/Downloads$
```

(5) 安装 Eclipse

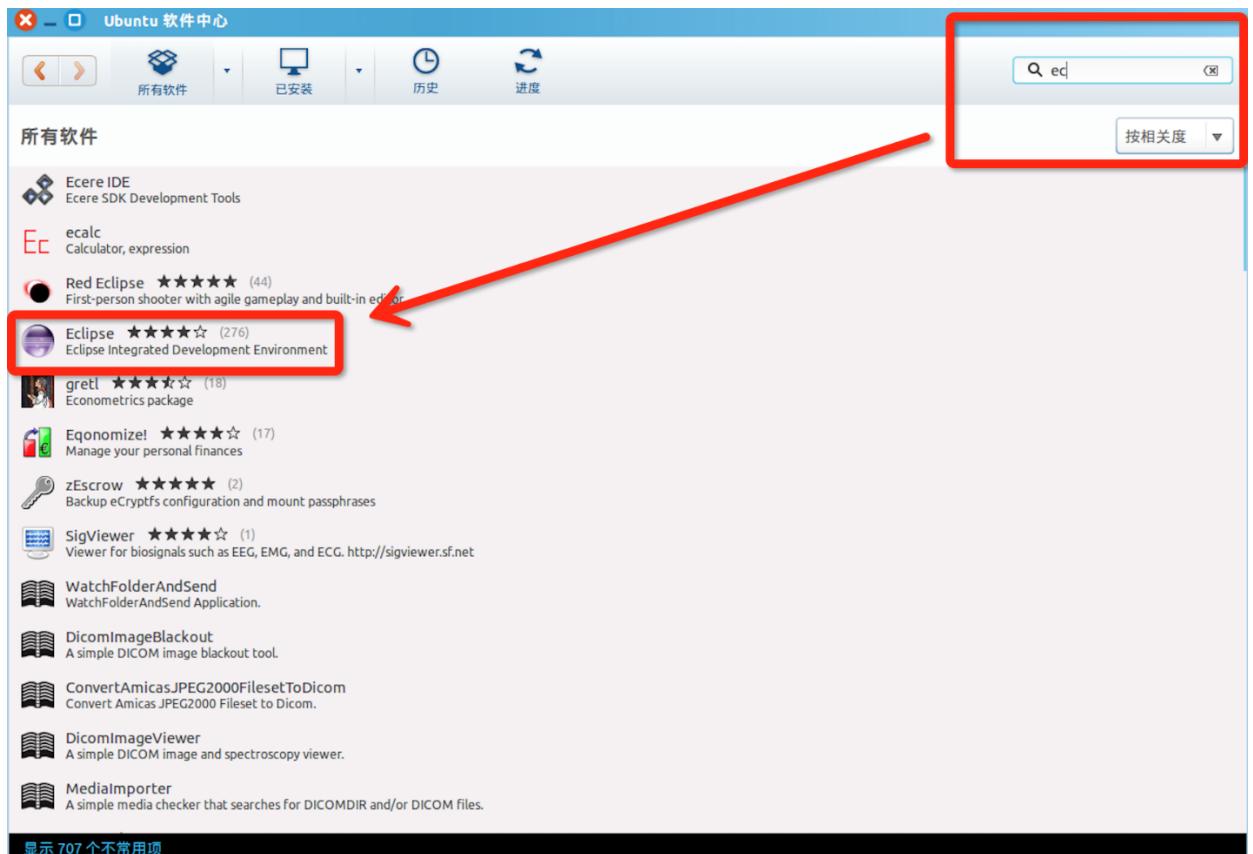
利用 Ubuntu 左侧边栏自带的软件中心安装软件，在 Ubuntu 左侧边栏打开软件中心



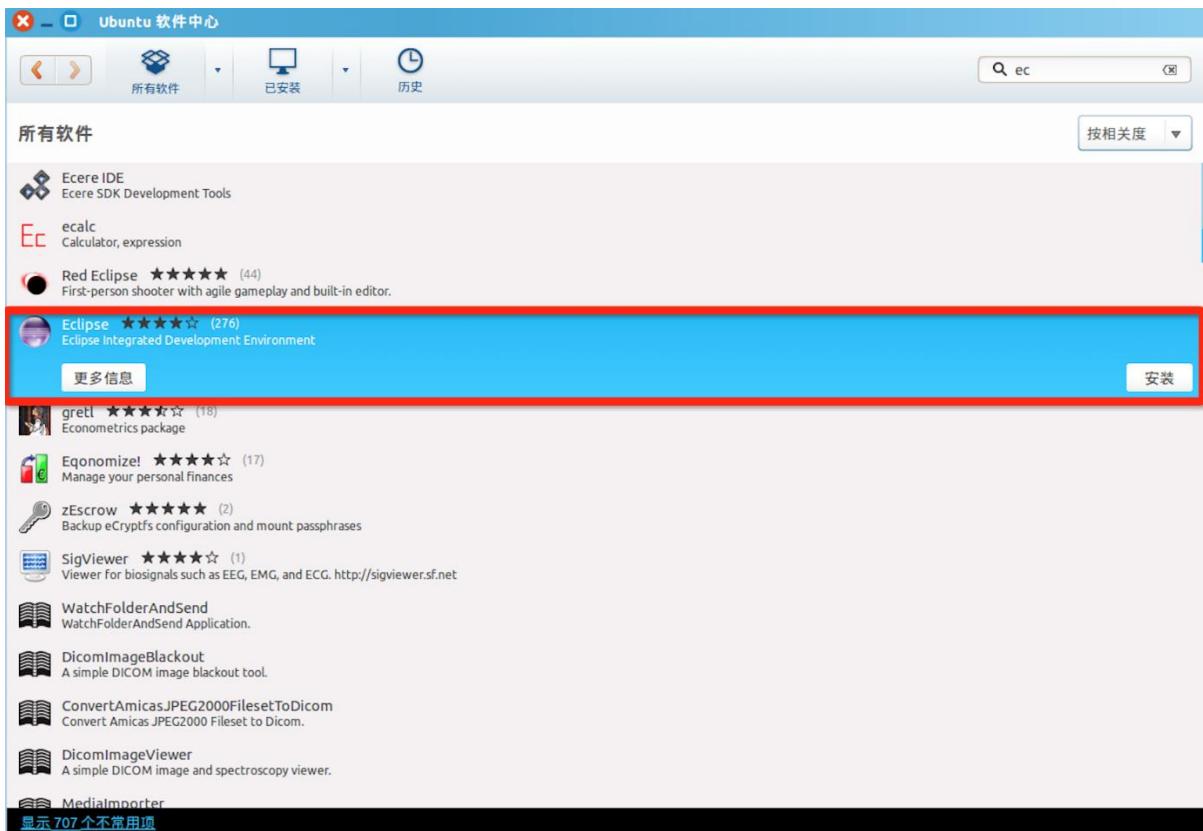
打开软件中心后，呈现如下界面



在软件中心搜索栏输入“ec”，软件中心会自动搜索相关的软件



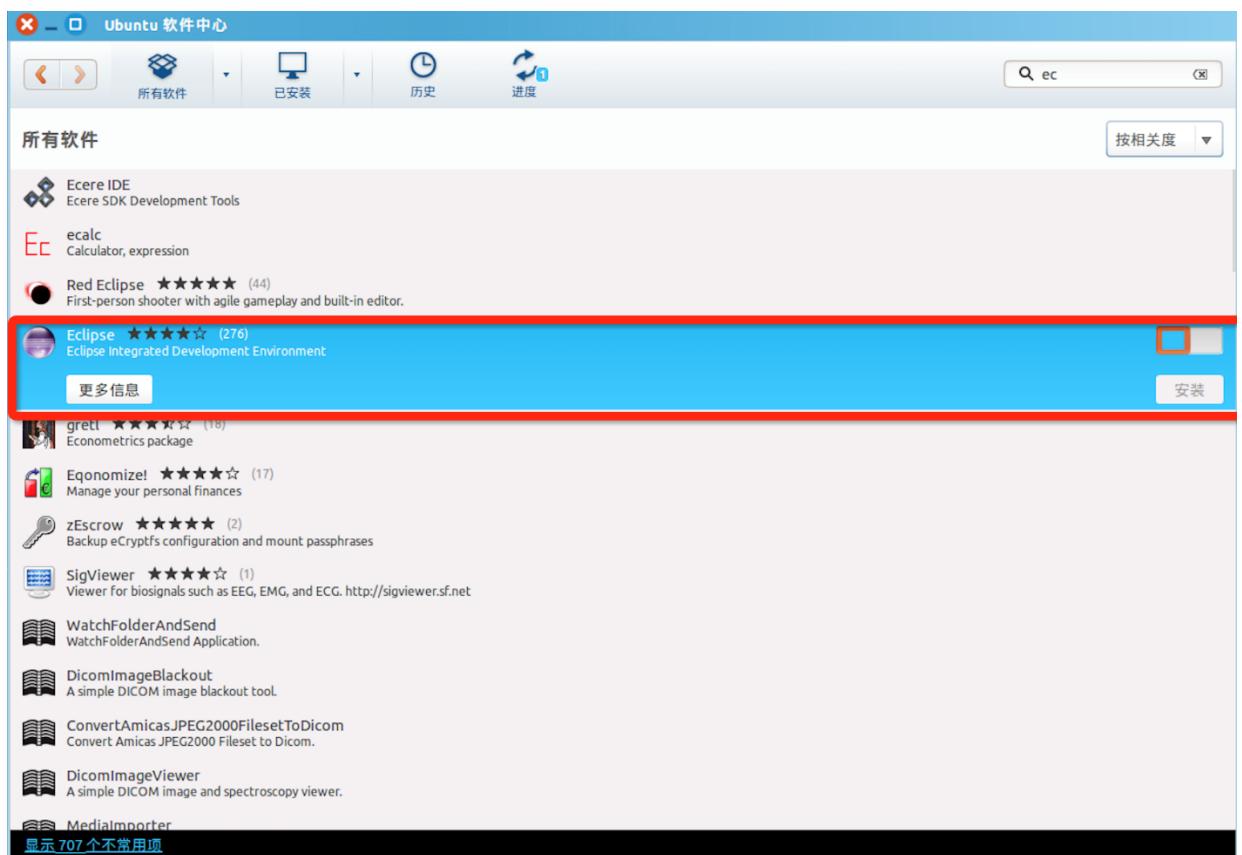
点击如下图中 Eclipse，进行安装



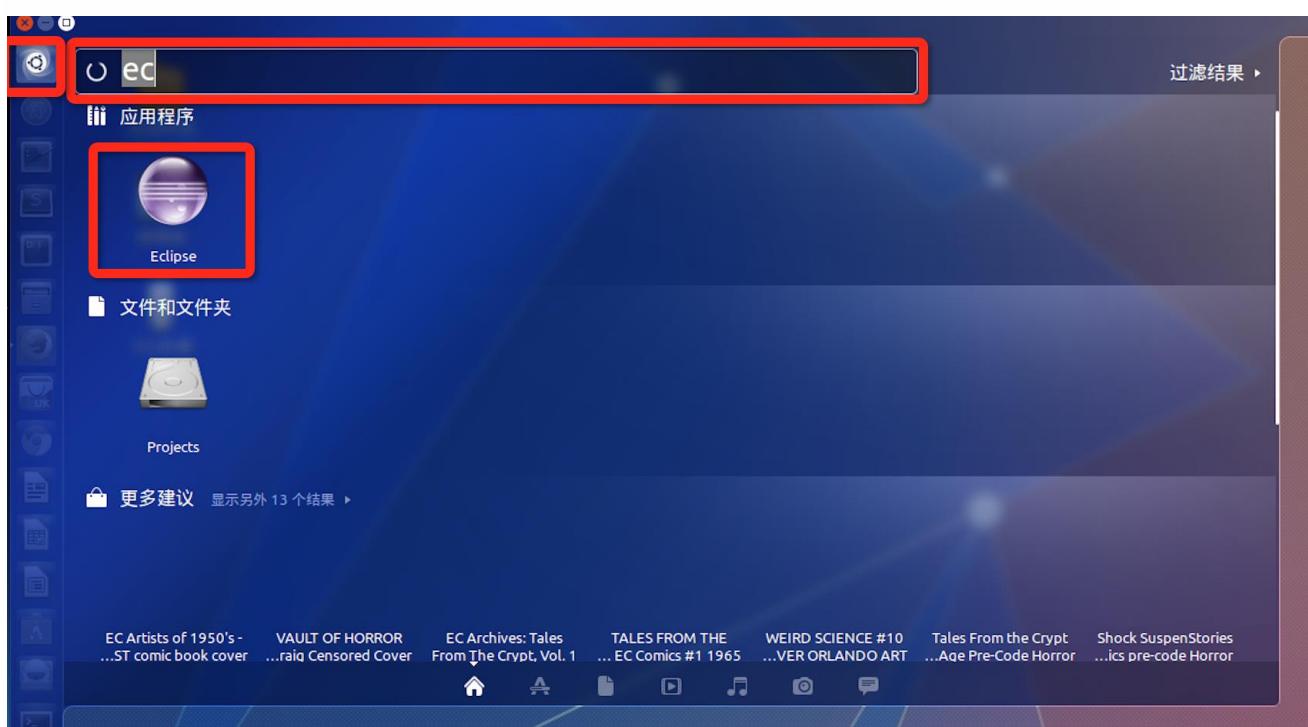
安装需要管理员权限，Ubuntu 系统需要用户认证，弹出“认证”窗口，请输入当前用户的登录密码



ubuntu 便会进入如下图的安装过程中,安装结束后安装进度条便会消失。



点击 Ubuntu 左侧边栏的搜索工具，输入“ec”，自动搜索已经安装好的相关软件



2、本地数据集上传到数据仓库 Hive

本案例采用的数据集压缩包为 `data_format.zip`, 该数据集压缩包是淘宝 2015 年双 11 前 6 个月(包含双十一)的交易数据, 里面包含 3 个文件, 分别是用户行为日志文件 `user_log.csv` 、回头客训练集 `train.csv` 、回头客测试集 `test.csv`. 下面列出这 3 个文件的数据格式定义:

用户行为日志 `user_log.csv`, 日志中的字段定义如下:

1. `user_id` | 买家 id
2. `item_id` | 商品 id
3. `cat_id` | 商品类别 id
4. `merchant_id` | 卖家 id
5. `brand_id` | 品牌 id
6. `month` | 交易时间:月
7. `day` | 交易事件:日
8. `action` | 行为,取值范围{0,1,2,3},0 表示点击, 1 表示加入购物车, 2 表示购买, 3 表示关注商品
9. `age_range` | 买家年龄分段: 1 表示年龄<18,2 表示年龄在 [18,24], 3 表示年龄在[25,29], 4 表示年龄在[30,34], 5 表示年龄在 [35,39], 6 表示年龄在[40,49], 7 和 8 表示年龄>=50,0 和 NULL 则表示未知

10. gender | 性别:0 表示女性， 1 表示男性， 2 和 NULL 表示未知

11. province| 收获地址省份

回头客训练集 `train.csv` 和回头客测试集 `test.csv`，训练集和测试集拥有相同的字段，字段定义如下：

1. `user_id` | 买家 id

2. `age_range` | 买家年龄分段：1 表示年龄<18,2 表示年龄在 [18,24]，3 表示年龄在[25,29]，4 表示年龄在[30,34]，5 表示年龄在[35,39]，6 表示年龄在[40,49]，7 和 8 表示年龄>=50,0 和 NULL 则表示未知

3. `gender` | 性别:0 表示女性， 1 表示男性， 2 和 NULL 表示未知

4. `merchant_id` | 商家 id

5. `label` | 是否是回头客，0 值表示不是回头客，1 值表示回头客， -1 值表示该用户已经超出我们所需要考虑的预测范围。

NULL 值只存在测试集，在测试集中表示需要预测的值。

下面，请登录 Linux 系统（本教程统一采用 `hadoop` 用户登录）。数据文件保存到`/home/hadoop/下载/`这目录下面。在 Linux 系统中打开一个终端，执行下面命令：

```
1. cd /home/hadoop/下载
```

```
2. ls
```

通过上面命令，就进入到了 `data_format.zip` 文件所在的目录，并且可以看到有个 `data_format.zip` 文件。注意，如果你把 `data_format.zip` 下载到了其他目录，这里请进入到你自己的存放 `data_format.zip` 的目录。

下面需要把 `data_format.zip` 进行解压缩，我们需要首先建立一个用于运行本案例的目录 `dbtaobao`，请执行以下命令：

```
1. cd /usr/local  
2. ls  
3. sudo mkdir dbtaobao  
4. //这里会提示你输入当前用户（本教程是 hadoop 用户名）的  
   密码  
5. //下面给 hadoop 用户赋予针对 dbtaobao 目录的各种操作权  
   限  
6. sudo chown -R hadoop:hadoop ./dbtaobao  
7. cd dbtaobao  
8. //下面创建一个 dataset 目录，用于保存数据集  
9. mkdir dataset  
10. //下面就可以解压缩 data_format.zip 文件  
11. cd ~ //表示进入 hadoop 用户的目录  
12. cd 下载  
13. ls
```

```
14. unzip data_format.zip -d /usr/local/dbtaobao/dataset  
15. cd /usr/local/dbtaobao/dataset  
16. ls
```

现在你就可以看到在 `dataset` 目录下有三个文件：`test.csv`、`train.csv`、`user_log.csv`

我们执行下面命令取出 `user_log.csv` 前面 5 条记录看一下
执行如下命令：

```
1. head -5 user_log.csv
```

可以看到，前 5 行记录如下：

```
user_id,item_id,cat_id,merchant_id,brand_id,month,day,action,age_range,gender,province  
328862,323294,833,2882,2661,08,29,0,0,1,内蒙古  
328862,844400,1271,2882,2661,08,29,0,1,1,山西  
328862,575153,1271,2882,2661,08,29,0,2,1,山西  
328862,996875,1271,2882,2661,08,29,0,1,1,内蒙古
```

数据集的预处理

1.删除文件第一行记录，即字段名称

`user_log.csv` 的第一行都是字段名称，我们在文件中的数据导入到数据仓库 `Hive` 中时，不需要第一行字段名称，因此，这里在做数据预处理时，删除第一行

```
1. cd /usr/local/dbtaobao/dataset  
2. //下面删除 user_log.csv 中的第 1 行  
3. sed -i '1d' user_log.csv //1d 表示删除第 1 行，同理，  
3d 表示删除第 3 行，nd 表示删除第 n 行  
4. //下面再用 head 命令去查看文件的前 5 行记录，就看不到字  
段名称这一行了  
5. head -5 user_log.csv
```

2.获取数据集中双 11 的前 100000 条数据

由于数据集中交易数据太大，这里只截取数据集中在双 11 的前 10000 条交易数据作为小数据集 `small_user_log.csv`

下面我们建立一个脚本文件完成上面截取任务，请把这个脚本文件放在 `dataset` 目录下和数据集 `user_log.csv`:

```
1. cd /usr/local/dbtaobao/dataset
```

2. vim predeal.sh

上面使用 vim 编辑器新建了一个 predeal.sh 脚本文件，请在这个脚本文件中加入下面代码：

```
#!/bin/bash

#下面设置输入文件，把用户执行 predeal.sh 命令时提供的第一个参数作为输入文件名称

infile=$1

#下面设置输出文件，把用户执行 predeal.sh 命令时提供的第二个参数作为输出文件名称

outfile=$2

#注意！！ 最后的$infile > $outfile 必须跟在}'这两个字符的后面

awk -F "," 'BEGIN{

    id=0;

}

{

    if($6==11 && $7==11){

        id=id+1;

        print $1,"$2","$3","$4","$5","$6","$7","$8","$9","$10","$11

    if(id==10000){

        exit

    }

}
```

```
}' $infile > $outfile
```

下面就可以执行 `predeal.sh` 脚本文件，截取数据集中在双 11 的前 10000 条数据作为小数据集 `small_user_log.csv`，命令如下：

```
1. chmod +x ./predeal.sh  
2. ./predeal.sh ./user_log.csv ./small_user_log.csv
```

3. 导入数据库

下面要把 `small_user_log.csv` 中的数据最终导入到数据仓库 `Hive` 中。为了完成这个操作，我们会首先把这个文件上传到分布式文件系统 `HDFS` 中，然后，在 `Hive` 中创建两个外部表，完成数据的导入。

启动 `HDFS`

```
1. cd /usr/local/hadoop  
2. ./sbin/start-dfs.sh
```

然后，执行 `jps` 命令看一下当前运行的进程：

```
1. jps
```

如果出现下面这些进程，说明 `Hadoop` 启动成功了。

3765 NodeManager

```
3639 ResourceManager
```

```
3800 Jps
```

```
3261 DataNode
```

```
3134 NameNode
```

```
3471 SecondaryNameNode
```

把 user_log.csv 上传到 HDFS 中：

现在，我们要把 Linux 本地文件系统中的 user_log.csv 上传到分布式文件系统 HDFS 中，存放在 HDFS 中的“/dbtaobao/dataset”目录下。

首先，请执行下面命令，在 HDFS 的根目录下面创建一个新的目录 dbtaobao，并在这个目录下创建一个子目录 dataset，如下：

```
1. cd /usr/local/hadoop  
2. ./bin/hdfs dfs -mkdir -p /dbtaobao/dataset/user_log
```

然后，把 Linux 本地文件系统中的 small_user_log.csv 上传到分布式文件系统 HDFS 的“/dbtaobao/dataset”目录下，命令如下：

```
1. cd /usr/local/hadoop  
2. ./bin/hdfs dfs -put /usr/local/dbtaobao/dataset/small_user_log.csv /dbtaobao/dataset/user_log
```

下面可以查看一下 HDFS 中的 small_user_log.csv 的前 10 条记录，命令如下：

```
1. cd /usr/local/hadoop  
2. ./bin/hdfs dfs -cat /dbtaobao/dataset/user_log/small_user_log.csv | head -10
```

在 Hive 上创建数据库

新建一个终端，因为需要借助于 MySQL 保存 Hive 的元数据，所以，请首先启动 MySQL 数据库：

```
1. service mysql start #可以在 Linux 的任何目录下执行  
该命令
```

由于 Hive 是基于 Hadoop 的数据仓库，使用 HiveQL 语言撰写的查询语句，最终都会被 Hive 自动解析成 MapReduce 任务由 Hadoop 去具体执行，因此，需要启动 Hadoop，然后再启动 Hive。由于前面我们已经启动了 Hadoop，所以，这里不需要再次启动 Hadoop。

下面，在这个新的终端中执行下面命令进入 Hive：

```
1. cd /usr/local/hive  
2. ./bin/hive # 启动Hive
```

启动成功以后，就进入了“hive>”命令提示符状态，可以输入类似 SQL 语句的 HiveQL 语句。

下面，我们要在 Hive 中创建一个数据库 dbtaobao，命令如下：

```
1. hive> create database dbtaobao;  
2. hive> use dbtaobao;
```

创建外部表：

请在 `hive` 命令提示符下输入如下命令：

```
1. hive> CREATE EXTERNAL TABLE dbtaobao.user_log(us  
er_id INT,item_id INT,cat_id INT,merchant_id INT,  
brand_id INT,month STRING,day STRING,action INT,a  
ge_range INT,gender INT,province STRING) COMMENT  
'Welcome to xmj dblab,Now create dbtaobao.user_lo  
g!' ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
STORED AS TEXTFILE LOCATION '/dbtaobao/dataset/u  
ser_log';
```

查询数据：

上面已经成功把 HDFS 中的“/dbtaobao/dataset/user_log”目录下的 `small_user_log.csv` 数据加载到了数据仓库 Hive 中，我们现在可以使用下面命令查询一下：

```
1. hive> select * from user_log limit 10;
```

本部分内容结束。

3、Hive 数据分析

一、操作 Hive

请登录 Linux 系统启动 MySQL 数据库

```
1. service mysql start # 可以在Linux 的任何目录下执行  
该命令
```

由于 Hive 是基于 Hadoop 的数据仓库，使用 HiveQL 语言撰写的查询语句，最终都会被 Hive 自动解析成 MapReduce 任务由 Hadoop 去具体执行，因此，需要启动 Hadoop，然后再启动 Hive。

请执行下面命令启动 Hadoop（如果你已经启动了 Hadoop 就不用再次启动了）：

```
1. cd /usr/local/hadoop  
2. ./sbin/start-dfs.sh
```

然后，执行 jps 命令看一下当前运行的进程：

1. jps

Shell 命令

如果出现下面这些进程，说明 Hadoop 启动成功了。

```
3765 NodeManager  
3639 ResourceManager  
3800 Jps  
3261 DataNode  
3134 NameNode  
3471 SecondaryNameNode
```

下面，继续执行下面命令启动进入 Hive：

```
1. cd /usr/local/hive  
2. ./bin/hive //启动 Hive
```

通过上述过程，我们就完成了 MySQL、Hadoop 和 Hive 三者的启动。

启动成功以后，就进入了“hive>”命令提示符状态，可以输入类似 SQL 语句的 HiveQL 语句。

然后，在“hive>”命令提示符状态下执行下面命令：

```
1. hive> use dbtaobao; -- 使用 dbtaobao 数据库
```

```
2. hive> show tables; -- 显示数据库中所有表。  
3. hive> show create table user_log; -- 查看 user_log  
表的各种属性;
```

执行结果如下：

```
OK  
  
CREATE EXTERNAL TABLE `user_log` (  
    `user_id` int,  
    `item_id` int,  
    `cat_id` int,  
    `merchant_id` int,  
    `brand_id` int,  
    `month` string,  
    `day` string,  
    `action` int,  
    `age_range` int,  
    `gender` int,  
    `province` string)  
  
COMMENT 'Welcome to xmj dblab,Now create dbtaobao.user_log!'  
  
ROW FORMAT SERDE  
    'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'  
  
WITH SERDEPROPERTIES (  
    'field.delim'=',',
```

```
'serialization.format'='')

STORED AS INPUTFORMAT

'org.apache.hadoop.mapred.TextInputFormat'

OUTPUTFORMAT

'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'

LOCATION

'hdfs://localhost:9000/dbtaobao/dataset/user_log'

TBLPROPERTIES (

'numFiles'='1',

'totalSize'='4729522',

'transient_lastDdlTime'='1487902650')

Time taken: 0.084 seconds, Fetched: 28 row(s)
```

可以执行下面命令查看表的简单结构：

```
1. hive> desc user_log;
```

执行结果如下：

```
OK

user_id          int
item_id          int
cat_id           int
merchant_id      int
brand_id         int
```

```
month          string  
day           string  
action         int  
age_range     int  
gender         int  
province      string  
  
Time taken: 0.029 seconds, Fetched: 11 row(s)
```

二、简单查询分析

先测试一下简单的指令：

```
1. hive> select brand_id from user_log limit 10; --  
查看日志前 10 个交易日志的商品品牌
```

执行结果如下：

```
OK
```

```
5476
```

```
5476
```

```
6109
```

```
5476
```

```
5476
```

```
5476
```

```
5476
```

```
5476
```

```
5476
```

```
6300
```

如果要查出每位用户购买商品时的多种信息，输出语句格式为

```
select 列 1, 列 2, ...., 列 n from 表名;
```

比如我们现在查询前 20 个交易日志中购买商品时的时间和商品的种类：

```
1. hive> select month,day,cat_id from user_log limit  
20;
```

执行结果如下：

```
OK
```

```
11 11 1280
```

```
11 11 1280
```

```
11 11 1181
```

```
11 11 1280
```

```
11 11 1280
```

```
11 11 1280
```

```
11 11 1280
```

```
11 11 1280
```

```
11 11 1280
```

```
11 11 962  
11 11 81  
11 11 1432  
11 11 389  
11 11 1208  
11 11 1611  
11 11 420  
11 11 1611  
11 11 1432  
11 11 389  
11 11 1432
```

有时我们在表中查询可以利用嵌套语句，如果列名太复杂可以设置该列的别名，以简化我们操作的难度，以下我们可以举个例子：

```
1. hive> select ul.at, ul.ci  from (select action as  
at, cat_id as ci from user_log) as ul limit 20;
```

执行结果如下：

```
OK  
0 1280  
0 1280  
0 1181  
2 1280
```

```
0 1280
0 1280
0 1280
0 1280
0 1280
0 962
2 81
2 1432
0 389
2 1208
0 1611
0 420
0 1611
0 1432
0 389
0 1432
```

这里简单的做个讲解，`action as at ,cat_id as ci` 就是把 `action` 设置别名 `at` ,`cat_id` 设置别名 `ci`, `FROM` 的括号里的内容我们也设置了别名 `ul`, 这样调用时用 `ul.at,ul.ci`, 可以简化代码。

三、查询条数统计分析

经过简单的查询后我们同样也可以在 `select` 后加入更多的条件对表进行查询,下面可以用函数来查找我们想要的内容。

(1)用聚合函数 count()计算出表内有多少条行数据

```
1. hive> select count(*) from user_log; -- 用聚合函数  
count()计算出表内有多少条行数据
```

执行结果如下：

```
1.  WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a  
2.  Query ID = hadoop_20170224103108_d6361e99-e76a-43e6-94b5-3fb0397e3ca6  
3.  Total jobs = 1  
4.  Launching Job 1 out of 1  
5.  Number of reduce tasks determined at compile time: 1  
6.  In order to change the average load for a reducer (in bytes):  
7.    set hive.exec.reducers.bytes.per.reducer=<number>  
8.  In order to limit the maximum number of reducers:  
9.    set hive.exec.reducers.max=<number>  
10. In order to set a constant number of reducers:  
11.   set mapreduce.job.reduces=<number>  
12. Job running in-process (local Hadoop)  
13. 2017-02-24 10:31:10,085 Stage-1 map = 100%,  reduce = 100%  
14. Ended Job = job_local1792612260_0001  
15. MapReduce Jobs Launched:  
16. Stage-Stage-1:  HDFS Read: 954982 HDFS Write: 0 SUCCESS  
17. Total MapReduce CPU Time Spent: 0 msec  
18. OK  
19. 10000  
20. Time taken: 1.585 seconds, Fetched: 1 row(s)
```

(2)在函数内部加上 distinct, 查出 uid 不重复的数据有多少条

下面继续执行操作：

```
1. hive> select count(distinct user_id) from user_lo  
g; -- 在函数内部加上 distinct, 查出 user_id 不重复的  
数据有多少条
```

执行结果如下：

```

WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions.
Query ID = hadoop_20170224103141_47682fd4-132b-4401-813a-0ed88f0fb01f
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Job running in-process (local Hadoop)
2017-02-24 10:31:42,501 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_local1198900757_0002
MapReduce Jobs Launched:
Stage-Stage-1: HDFS Read: 1901772 HDFS Write: 0 SUCCESS
Total MapReduce CPU Time Spent: 0 msec
OK
358
Time taken: 1.283 seconds, Fetched: 1 row(s)

```

(3)查询不重复的数据有多少条(为了排除客户刷单情况)

```

1. hive> select count(*) from (select user_id,item_i
d,cat_id,merchant_id,brand_id,month,day,action fr
om user_log group by user_id,item_id,cat_id,merch
ant_id,brand_id,month,day,action having count(*)=
1)a;

```

执行结果如下：

```

WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions. Consider using a
Query ID = hadoop_20170224103334_3391e361-c710-4162-b022-2658f41fc228
Total jobs = 2
Launching Job 1 out of 2
Number of reduce tasks not specified. Estimated from input data size: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Job running in-process (local Hadoop)
2017-02-24 10:33:35,890 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_local1670662918_0003
Launching Job 2 out of 2
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Job running in-process (local Hadoop)
2017-02-24 10:33:37,026 Stage-2 map = 100%,  reduce = 100%
Ended Job = job_local2041177199_0004
MapReduce Jobs Launched:
Stage-Stage-1: HDFS Read: 2848562 HDFS Write: 0 SUCCESS
Stage-Stage-2: HDFS Read: 2848562 HDFS Write: 0 SUCCESS
Total MapReduce CPU Time Spent: 0 msec
OK
4754
Time taken: 2.478 seconds, Fetched: 1 row(s)

```

可以看出，排除掉重复信息以后，只有 4754 条记录。

注意：嵌套语句最好取别名，就是上面的 a，否则很容易出现如下错误。

```
FAILED: ParseException line 1:131 cannot recognize input near '<EOF>' '<EOF>' '<EOF>' in subquery source
```

四. 关键字条件查询分析

1.以关键字的存在区间为条件的查询

使用 **where** 可以缩小查询分析的范围和精确度，下面用实例来测试一下。

(1) 查询双 11 那天有多少人购买了商品

```
1. hive> select count(distinct user_id) from user_lo  
g where action='2';
```

执行结果如下：

```
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions.  
Query ID = hadoop_20170224103500_44e669ed-af51-4856-8963-002d85112f32  
Total jobs = 1  
Launching Job 1 out of 1  
Number of reduce tasks determined at compile time: 1  
In order to change the average load for a reducer (in bytes):  
  set hive.exec.reducers.bytes.per.reducer=<number>  
In order to limit the maximum number of reducers:  
  set hive.exec.reducers.max=<number>  
In order to set a constant number of reducers:  
  set mapreduce.job.reduces=<number>  
Job running in-process (local Hadoop)  
2017-02-24 10:35:01,940 Stage-1 map = 100%,  reduce = 100%  
Ended Job = job_local1951453719_0005  
MapReduce Jobs Launched:  
Stage-Stage-1: HDFS Read: 3795352 HDFS Write: 0 SUCCESS  
Total MapReduce CPU Time Spent: 0 msec  
OK  
358  
Time taken: 1.231 seconds, Fetched: 1 row(s)
```

2.关键字赋予给定值为条件，对其他数据进行分析

取给定时间和给定品牌，求当天购买的此品牌商品的数量

```
1. hive> select count(*) from user_log where action='2' and brand_id=2661;
```

执行结果如下：

```
WARNING: Hive-on-MR is deprecated in Hive 2 and may not be available in the future versions.
Query ID = hadoop_20170224103541_4640ca81-1d25-48f8-8d9d-6027f2befdb9
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Job running in-process (local Hadoop)
2017-02-24 10:35:42,457 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_local1749705881_0006
MapReduce Jobs Launched:
Stage-Stage-1: HDFS Read: 4742142 HDFS Write: 0 SUCCESS
Total MapReduce CPU Time Spent: 0 msec
OK
3
Time taken: 1.258 seconds, Fetched: 1 row(s)
```

五. 根据用户行为分析

从现在开始，我们只给出查询语句，将不再给出执行结果。

1. 查询一件商品在某天的购买比例或浏览比例

```
1. hive> select count(distinct user_id) from user_log where action='2'; -- 查询有多少用户在双11购买了商品
1. hive> select count(distinct user_id) from user_log; -- 查询有多少用户在双11点击了该店
```

根据上面语句得到购买数量和点击数量，两个数相除即可得出当天该商品的购买率。

2.查询双 11 那天，男女买家购买商品的比例

```
1. hive> select count(*) from user_log where gender=0; --查询双 11 那天女性购买商品的数量  
2. hive> select count(*) from user_log where gender=1; --查询双 11 那天男性购买商品的数量
```

上面两条语句的结果相除，就得到了要要求的比例。

3.给定购买商品的数量范围，查询某一天在该网站的购买该数量

商品的用户 id

```
1. hive> select user_id from user_log where action='2' group by user_id having count(action='2')>5; -  
- 查询某一天在该网站购买商品超过 5 次的用户 id
```

六.用户实时查询分析

不同的品牌的浏览次数

```
1. hive> create table scan(brand_id INT,scan INT) COMMENT 'This is the search of bigdatataobao' ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE; -- 创建新的数据表进行存储
```

```
2. hive> insert overwrite table scan select brand_id, count(action) from user_log where action='2' group by brand_id; --导入数据  
3. hive> select * from scan; -- 显示结果
```

到这里，Hive 数据分析实验顺利结束。

4、将数据从 Hive 导入到 MySQL

Hive 预操作：

如果你还没有启动 Hive，请首先启动 Hive。登录 Linux 系统启动 MySQL 数据库：

```
1. service mysql start # 可以在Linux 的任何目录下执行该命令
```

启动 Hadoop，然后再启动 Hive：

请执行下面命令启动 Hadoop（如果你已经启动了 Hadoop 就不用再次启动了）：

```
1. cd /usr/local/hadoop  
2. ./sbin/start-all.sh
```

然后，执行 `jps` 命令看一下当前运行的进程：

```
1. jps
```

如果出现下面这些进程，说明 Hadoop 启动成功了。

```
3765 NodeManager
```

```
3639 ResourceManager
```

```
3800 Jps
```

```
3261 DataNode
```

```
3134 NameNode
```

```
3471 SecondaryNameNode
```

下面，继续执行下面命令启动进入 Hive：

```
1. cd /usr/local/hive  
2. ./bin/hive #启动Hive
```

通过上述过程，我们就完成了 MySQL、Hadoop 和 Hive 三者的启动。

启动成功以后，就进入了“`hive>`”命令提示符状态，可以输入类似 SQL 语句的 HiveQL 语句。

然后，在“`hive>`”命令提示符状态下执行下面命令：

1、创建临时表 `inner_user_log` 和 `inner_user_info`

```
1. hive> create table dbtaobao.inner_user_log(user_id INT,item_id INT,cat_id INT,merchant_id INT,brand_id INT,month STRING,day STRING,action INT,age_range INT,gender INT,province STRING) COMMENT 'Welcome to XMU dblab! Now create inner table inner_user_log ' ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;
```

这个命令执行完以后，Hive 会自动在 HDFS 文件系统中创建对应的数据文件“/user/hive/warehouse/dbtaobao.db/inner_user_log”。

2、将 user_log 表中的数据插入到 inner_user_log,

在步骤:本地数据集上传到数据仓库 Hive 中，我们已经在 Hive 中的 dbtaobao 数据库中创建了一个外部表 user_log。下面把 dbtaobao.user_log 数据插入到 dbtaobao.inner_user_log 表中，命令如下：

```
1. hive> INSERT OVERWRITE TABLE dbtaobao.inner_user_log select * from dbtaobao.user_log;
```

请执行下面命令查询上面的插入命令是否成功执行：

```
1. hive> select * from inner_user_log limit 10;
```

三、使用 Sqoop 将数据从 Hive 导入 MySQL

1、启动 Hadoop 集群、MySQL 服务。前面我们已经启动了 Hadoop 集群和 MySQL 服务。请确认已经按照前面操作启动成功。

2、将前面生成的临时表数据从 Hive 导入到 MySQL 中，包含如下四个步骤。

(1) 登录 MySQL

请在 Linux 系统中新建一个终端，执行下面命令：

```
1. mysql -u root -p
```

为了简化操作，本教程直接使用 root 用户登录 MySQL 数据库，但是，在实际应用中，建议在 MySQL 中再另外创建一个用户。执行上面命令以后，就进入了“mysql>”命令提示符状态。

(2) 创建数据库

```
1. mysql> show databases; # 显示所有数据库  
2. mysql> create database dbtaobao; # 创建 dbtaobao 数  
据库  
3. mysql> use dbtaobao; # 使用数据库
```

注意：请使用下面命令查看数据库的编码：

```
1. mysql> show variables like "char%";
```

会显示类似下面的结果：

```
+-----+-----+
| Variable_name          | Value      |
+-----+-----+
| character_set_client   | utf8       |
| character_set_connection | utf8       |
| character_set_database  | latin1     |
| character_set_filesystem | binary     |
| character_set_results   | utf8       |
| character_set_server    | latin1     |
| character_set_system    | utf8       |
| character_sets_dir      | /usr/share/mysql/charsets/ |
+-----+-----+
8 rows in set (0.00 sec)
```

请确认当前编码为 `utf8`，否则无法导入中文。下面是笔者电脑上修改了编码格式后的结果：

```
+-----+-----+
| Variable_name          | Value      |
+-----+-----+
| character_set_client   | utf8       |
+-----+-----+
```

```
| character_set_connection | utf8          |
| character_set_database   | utf8          |
| character_set_filesystem | binary        |
| character_set_results   | utf8          |
| character_set_server    | utf8          |
| character_set_system    | utf8          |
| character_sets_dir      | /usr/share/mysql/charsets/ |
+-----+-----+
8 rows in set (0.00 sec)
```

(3) 创建表

下面在 MySQL 的数据库 dbtaobao 中创建一个新表 user_log，并设置其编码为 utf-8:

```
1. mysql> CREATE TABLE `dbtaobao`.`user_log`(`user_id` varchar(20),`item_id` varchar(20),`cat_id` varchar(20),`merchant_id` varchar(20),`brand_id` varchar(20),`month` varchar(6),`day` varchar(6),`action` varchar(6),`age_range` varchar(6),`gender` varchar(6),`province` varchar(10)) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

提示：语句中的引号是反引号`，不是单引号'。需要注意的是，sqoop 抓数据的时候会把类型转为 string 类型，所以 mysql 设计字段的时候，设置为 varchar。

创建成功后，输入下面命令退出 MySQL：

```
1. mysql> exit;
```

(4) 导入数据(执行时间：20 秒左右)

注意，刚才已经退出 MySQL，回到了 Shell 命令提示符状态。下面就可以执行数据导入操作，

```
1. cd /usr/local/sqoop  
2. bin/sqoop export --connect jdbc:mysql://localhost:  
    3306/dbtaobao --username root --password root -  
    -table user_log --export-dir '/user/hive/warehouse/  
    dbtaobao.db/inner_user_log' --fields-terminated  
    -by ',';
```

查看 MySQL 中 user_log 或 user_info 表中的数据：

下面需要再次启动 MySQL，进入“mysql>”命令提示符状态：

```
1. mysql -u root -p
```

会提示你输入 MySQL 的 root 用户的密码，本教程中安装的 MySQL 数据库的 root 用户的密码是 hadoop。

然后执行下面命令查询 user_action 表中的数据：

```
1. mysql> use dbtaobao;  
2. mysql> select * from user_log limit 10;
```

会得到类似下面的查询结果：

	user_id	item_id	cat_id	merchant_id	brand_id	month	day	action	age_range	gender	province	
1	414196	1109106	1188	0	3518	4805	11	1	0	4	宁夏	
1	414196	380046	4	2	231	6065	11	11	0	5	陕西	
1	414196	1109106	1188	0	3518	4805	11	1	0	7	山西	
1	414196	1109106	1188	0	3518	4805	11	1	0	6	河南	
1	414196	1109106	1188	0	3518	763	11	1	2	2	四川	

```

| 414196 | 944554 | 1432 | 323 | 320 | 11 | 1
1 | 2 | 7 | 2 | 青海 | | |
| 414196 | 1110009 | 1188 | 298 | 7907 | 11 | 1
1 | 2 | 3 | 1 | 澳门 | | |
| 414196 | 146482 | 513 | 2157 | 6539 | 11 | 1
1 | 0 | 1 | 0 | 上海市 | | |
| 414196 | 944554 | 1432 | 323 | 320 | 11 | 1
1 | 0 | 2 | 1 | 宁夏 | | |
| 414196 | 1109106 | 1188 | 3518 | 4805 | 11 | 1
1 | 0 | 7 | 0 | 新疆 | | |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+
10 rows in set (0.03 sec)

```

本部分内容完成

5、利用 Spark 预测回头客

预先处理 test.csv 数据集，把这 test.csv 数据集里 label 字段表示-1 值剔除掉，保留需要预测的数据。并假设需要预测的数据中 label 字段均为 1：

```

1. cd /usr/local/dbtaobao/dataset
2. vim predeal_test.sh

```

上面使用 vim 编辑器新建了一个 predeal_test.sh 脚本文件，请在这个脚本文件中加入下面代码：

```
#!/bin/bash

#下面设置输入文件，把用户执行 predeal_test.sh 命令时提供的第一个参数作为输入文件名称

infile=$1

#下面设置输出文件，把用户执行 predeal_test.sh 命令时提供的第二个参数作为输出文件名称

outfile=$2

#注意！！ 最后的$infile > $outfile 必须跟在}'这两个字符的后面

awk -F "," 'BEGIN{

id=0;

}

{

if($1 && $2 && $3 && $4 && !$5){

id=id+1;

print $1","$2","$3","$4","1

if(id==10000){

exit

}

}

}' $infile > $outfile
```

下面就可以执行 predeal_test.sh 脚本文件，截取测试数据集需要预测的数据到 test_after.csv，命令如下：

```
1. chmod +x ./predeal_test.sh  
2. ./predeal_test.sh ./test.csv ./test_after.csv
```

train.csv 的第一行都是字段名称，不需要第一行字段名称,这里在对 train.csv 做数据预处理时，删除第一行

```
1. sed -i '1d' train.csv
```

然后剔除掉 train.csv 中字段值部分字段值为空的数据。

```
1. cd /usr/local/dbtaobao/dataset  
2. vim predeal_train.sh
```

上面使用 vim 编辑器#新建了一个 predeal_train.sh 脚本文件，请在这个脚本文件中加入下面代码：

```
#!/bin/bash  
  
#下面设置输入文件，把用户执行 predeal_train.sh 命令时提供的第一个参数作为输入文件名称  
  
infile=$1  
  
#下面设置输出文件，把用户执行 predeal_train.sh 命令时提供的第二个参数作为输出文件名称
```

```
outfile=$2

#注意！！ 最后的$infile > $outfile 必须跟在}'这两个字符的后面

awk -F "," 'BEGIN{

id=0;

}

{

if($1 && $2 && $3 && $4 && ($5!=-1)) {

id=id+1;

print $1","$2","$3","$4","$5

if(id==10000){

exit

}

}

}' $infile > $outfile
```

下面就可以执行 predeal_train.sh 脚本文件，截取测试数据集需要预测的数据到 train_after.csv，命令如下：

```
1. chmod +x ./predeal_train.sh

2. ./predeal_train.sh ./train.csv ./train_after.csv
```

预测回头客：

启动 hadoop：

请先确定 **Spark** 的运行方式，如果 **Spark** 是基于 **Hadoop** 伪分布式运行，那么请先运行 **Hadoop**。

如果 **Hadoop** 没有运行，请执行如下命令：

```
1. cd /usr/local/hadoop/  
2. sbin/start-dfs.sh
```

将两个数据集分别存取到 **HDFS** 中

```
1. bin/hadoop fs -mkdir -p /dbtaobao/dataset  
2. bin/hadoop fs -put /usr/local/dbtaobao/dataset/tr  
ain_after.csv /dbtaobao/dataset  
3. bin/hadoop fs -put /usr/local/dbtaobao/dataset/te  
st_after.csv /dbtaobao/dataset
```

启动 **MySQL** 服务

```
1. service mysql start  
2. mysql -uroot -p #会提示让你输入数据库密码
```

输入密码后，你就可以进入“**mysql>**”命令提示符状态，然后就可以输入下面的 **SQL** 语句完成表的创建：

```
1. use dbtaobao;
```

```
2. create table rebuy (score varchar(40),label varchar(40));
```

启动 pyspark

Spark 支持通过 JDBC 方式连接到其他数据库获取数据生成 DataFrame。

执行如下命令：（之前执行过就不用在此执行了）

```
1. cd ~/下载/  
2. unzip mysql-connector-java-5.1.40.zip -d /usr/local/spark/jars
```

接下来正式启动 pyspark

```
1. cd /usr/local/spark  
2. ./bin/pyspark --jars /usr/local/spark/jars/mysql-  
    connector-java-5.1.40/mysql-connector-java-5.1.40  
    -bin.jar --driver-class-path /usr/local/spark/jar  
    s/mysql-connector-java-5.1.40/mysql-connector-jav  
    a-5.1.40-bin.jar
```

支持向量机 SVM 分类器预测回头客

在 pyspark 中执行如下操作：

1. 导入需要的包：首先，我们导入需要的包：

```
1. from pyspark import SparkContext  
2. from pyspark.mllib.regression import LabeledPoint  
3. from pyspark.mllib.linalg import Vectors,Vector  
4. from pyspark.mllib.classification import SVMModel  
    , SVMWithSGD  
5. from pyspark.python.pyspark.shell import spark  
6. from pyspark.sql.types import Row  
7. from pyspark.sql.types import *
```

2. 读取训练数据

首先，读取训练文本文件；然后，通过 map 将每行的数据用“,”隔开，在数据集中，每行被分成了 5 部分，前 4 部分是用户交易的 3 个特征，最后一部分是用户交易的分类。把这里我们用 LabeledPoint 来存储标签列和特征列。LabeledPoint 在监督学习中常用来存储标签和特征，其中要求标签的类型是 double，特征的类型是 Vector。

```
1. sc = SparkContext.getOrCreate()  
2. train_data = sc.textFile("/dbtaobao/dataset/train  
    _after.csv")
```

```
3. test_data = sc.textFile("/dbtaobao/dataset/test_a  
fter.csv")
```

3.构建模型

```
1. def GetParts(line):  
2.     parts = line.split(',')  
3.     return LabeledPoint(float(parts[4]), Vectors.dense(float(parts[1]), float(parts[2]), float(parts[3])))  
4. train = train_data.map(lambda line: GetParts(line))  
5. test = test_data.map(lambda line: GetParts(line))
```

接下来，通过训练集构建模型 SVMWithSGD。这里的 SGD 即著名的随机梯度下降算法（Stochastic Gradient Descent）。设置迭代次数为 1000，除此之外还有 stepSize（迭代步伐大小）， regParam（regularization 正则化控制参数）， miniBatchFraction（每次迭代参与计算的样本比例）， initialWeights（weight 向量初始值）等参数可以进行设置。

```
1. numIterations = 1000  
2. model = SVMWithSGD.train(train, numIterations)
```

4.评估模型

接下来，我们清除默认阈值，这样会输出原始的预测评分，即带有确信度的结果。

```
1. def Getpoint(point):  
2.     score = model.predict(point.features)  
3.     return str(score) + " " + str(point.label)  
4. model.clearThreshold()  
5. scoreAndLabels = test.map(lambda point: Getpoint  
(point))  
6. scoreAndLabels.foreach(lambda x : print(x))
```

spark-shell 会打印出如下结果

-21754.021986991942 1.0

-50378.971923247285 1.0

-11646.722569368836 1.0

.....

如果我们设定了阈值，则会把大于阈值的结果当成正预测，小于阈值的结果当成负预测。

```
1. model.setThreshold(0.0)  
2. scoreAndLabels.foreach(lambda x : print(x))
```

5. 把结果添加到 mysql 数据库中

现在我们将上面没有设定阀值的测试集结果存入到 MySQL 数据中。

```
1. def Getpoint(point):
2.     score = model.predict(point.features)
3.     return str(score) + " " + str(point.label)
4. model.clearThreshold()
5. scoreAndLabels = test.map(lambda point: Getpoint
    (point))
6. //设置回头客数据
7. rebuyRDD = scoreAndLabels.map(lambda x: x.split(
    ""))
8. //下面要设置模式信息
9. schema = StructType([StructField("score", StringType(),
    True),StructField("label", StringType(), True)])
10. //下面创建 Row 对象，每个 Row 对象都是 rowRDD 中的一行
11. rowRDD = rebuyRDD.map(lambda p : Row(p[0].strip(),
    (), p[1].strip()))
12. //建立起 Row 对象和模式之间的对应关系，也就是把数据和
    模式对应起来
```

```
13. rebuyDF = spark.createDataFrame(rowRDD, schema,  
True)  
  
14. //下面创建一个 prop 变量用来保存 JDBC 连接参数  
  
15. prop = {}  
  
16. prop['user'] = 'root' //表示用户名是 root  
  
17. prop['password'] = '123' //表示密码是 123  
  
18. prop['driver'] = "com.mysql.jdbc.Driver" 、、 //  
表示驱动程序是 com.mysql.jdbc.Driver  
  
19. //下面就可以连接数据库，采用 append 模式，表示追加记  
录到数据库 dbtaobao 的 rebuy 表中  
  
20. rebuyDF.write.jdbc("jdbc:mysql://localhost:3306/  
dbtaobao", 'dbtaobao.rebuy', 'append', prop)
```

到这里，第四个步骤的实验内容顺利结束。

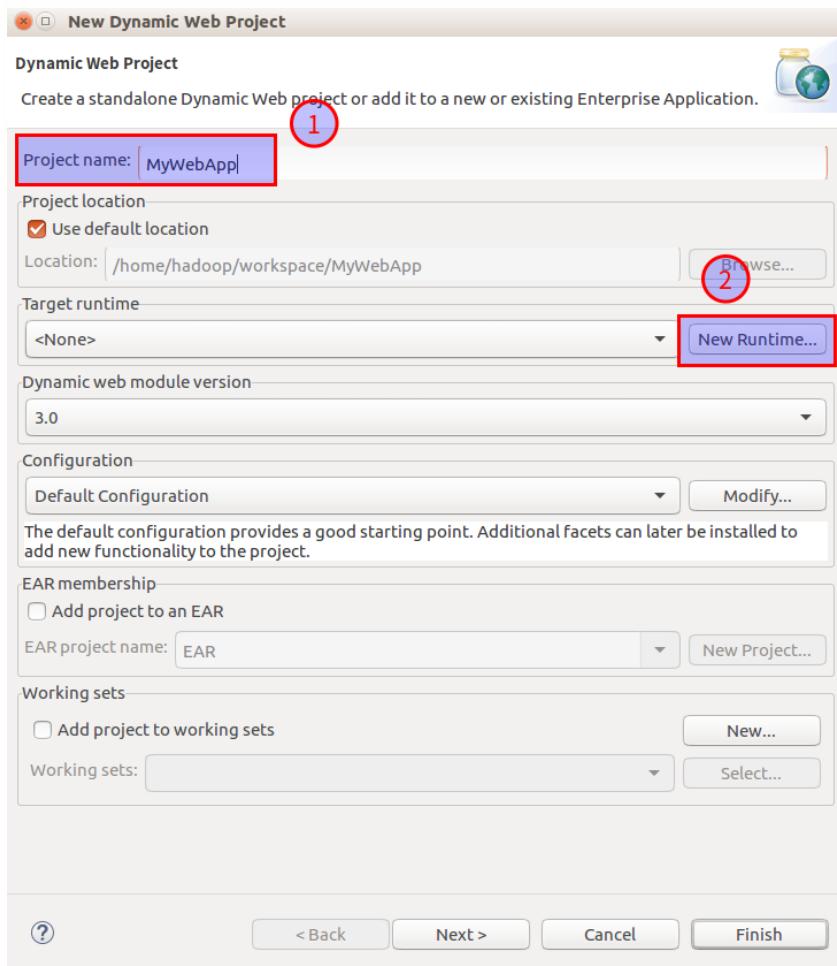
6、利用 ECharts 进行数据可视化分析

启动 mysql：执行如下命令，启动 mysql

```
1. service mysql start
```

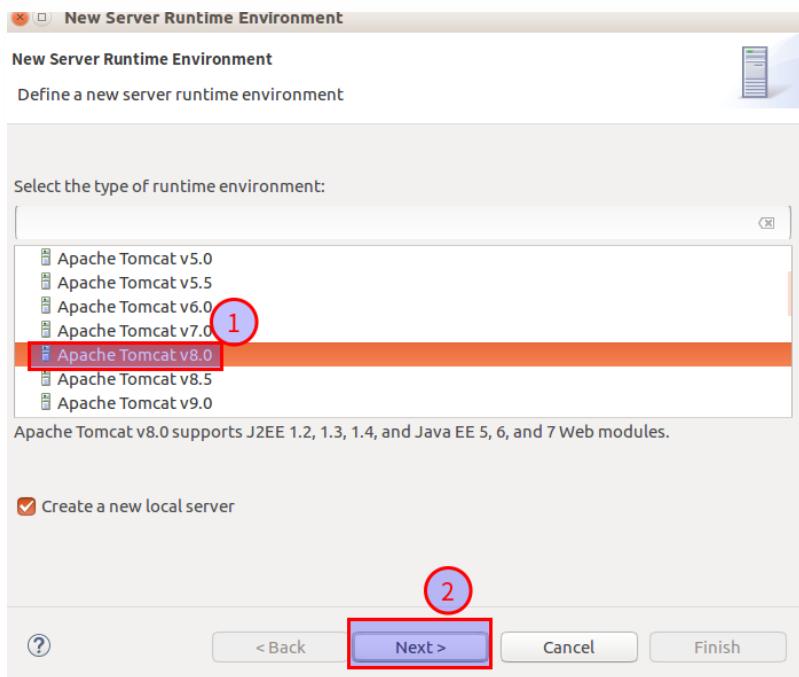
利用 Eclipse 新建可视化 Web 应用

1. 打开 Eclipse，点击“File”菜单，或者通过工具栏的“New”创建 Dynamic Web Project，弹出向导对话框，填入 Project name 后，并点击“New Runtime”，如下图所示：

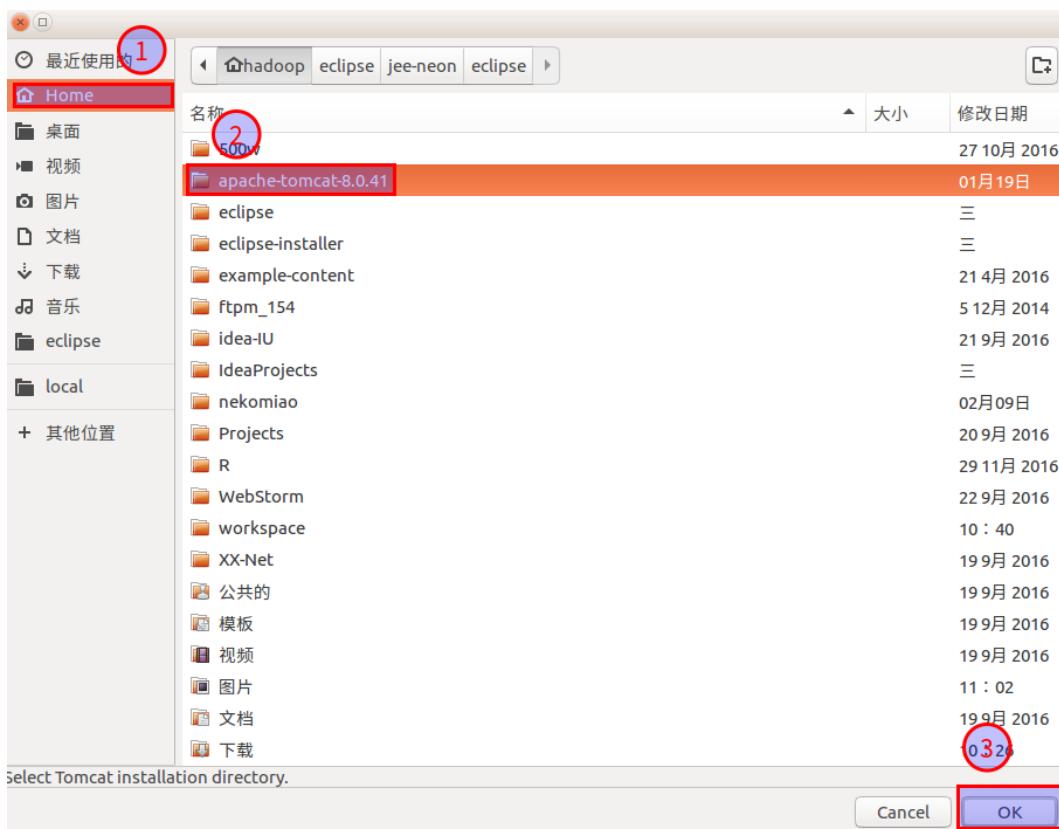
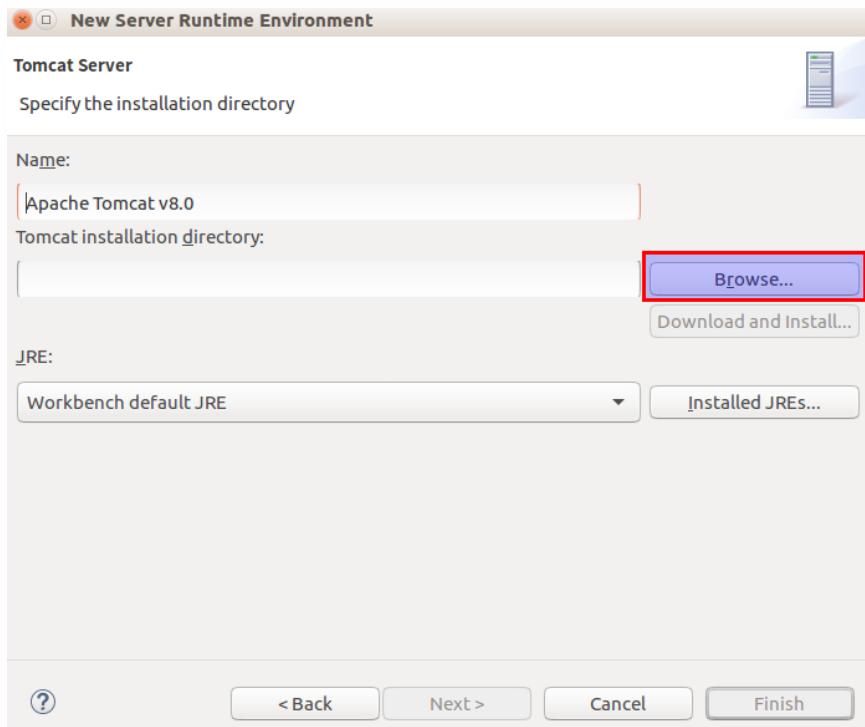


2. 出现 New Server Runtime Environment 向导对话框,选择“Apache

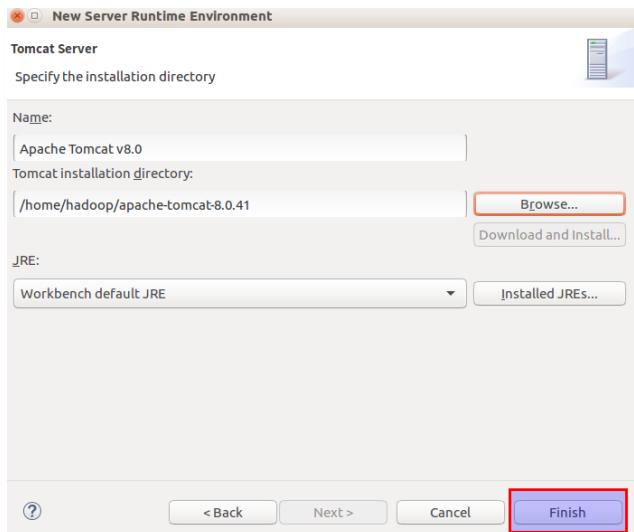
Tomcat v8.0”,点击 next 按钮,如下图:



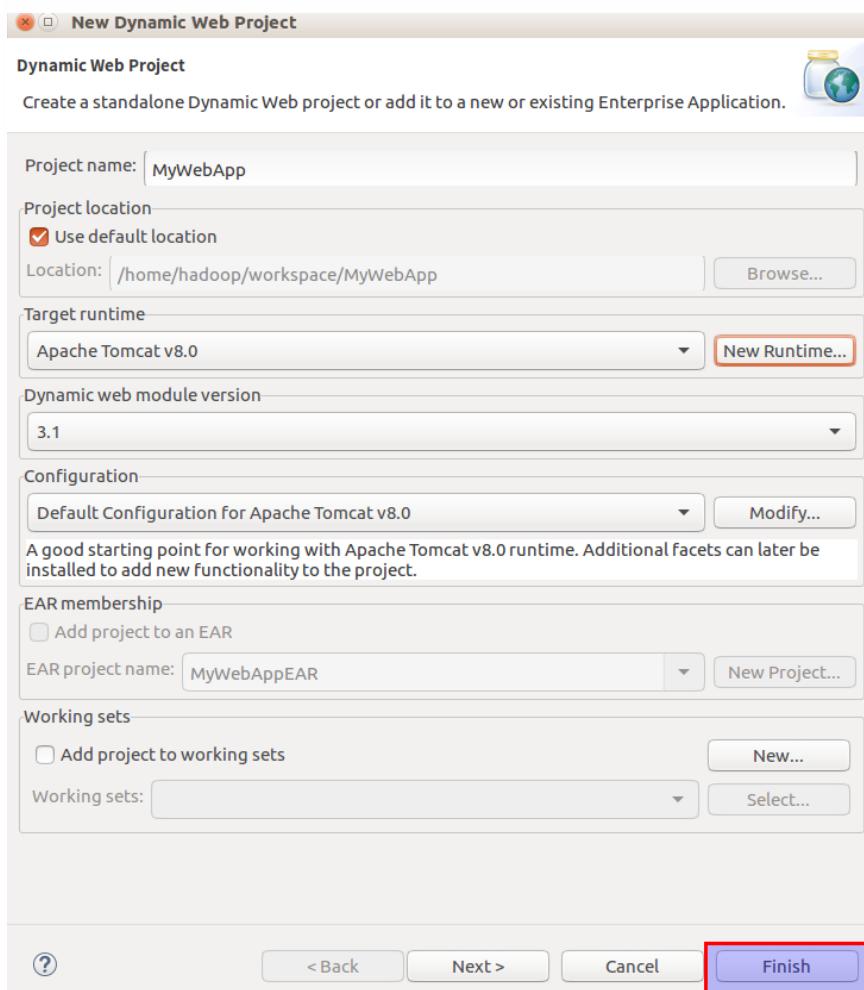
3 选择 Tomcat 安装文件夹, 如下图:



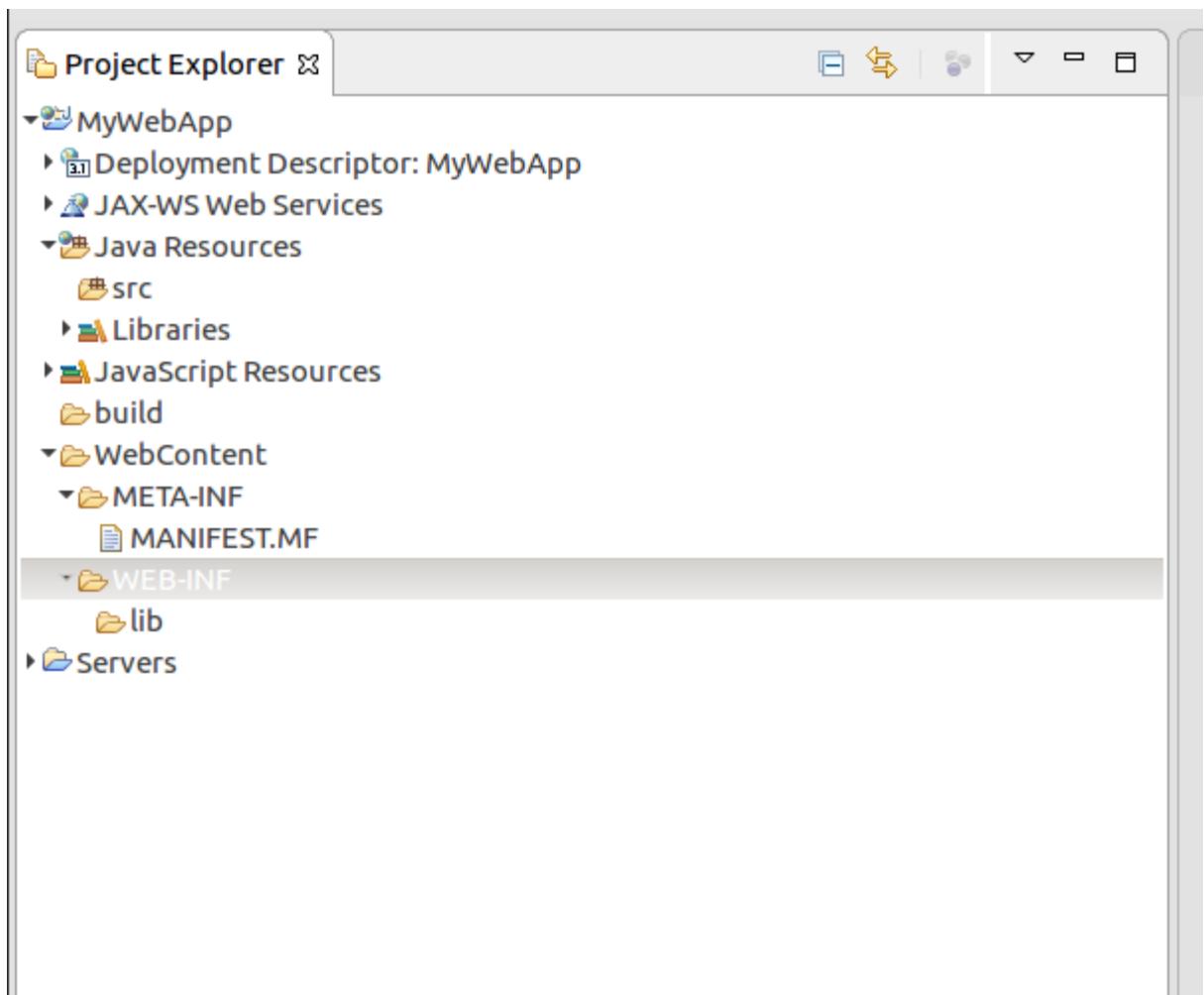
4. 返回 New Server Runtime Environment 向导对话框，点击 finish 即可。如下图：



5. 返回 Dynamic Web Project 向导对话框，点击 finish 即可。如图



6. 这样新建一个 Dynamic Web Project 就完成了。在 Eclipse 中展开新建的 MyWebApp 项目，初始整个项目框架如下：



src 文件夹用来存放 Java 服务端的代码，例如:读取数据库 MySQL 中的数据

WebContent 文件夹则用来存放前端页面文件，例如：前端页面资源 css、img、js，前端 JSP 页面

7. 安装 mysql-connector-java

mysql-connector-java-* .zip 是 Java 连接 MySQL 的驱动包,默认会下载到"~/下载/"目录

执行如下命令：

```

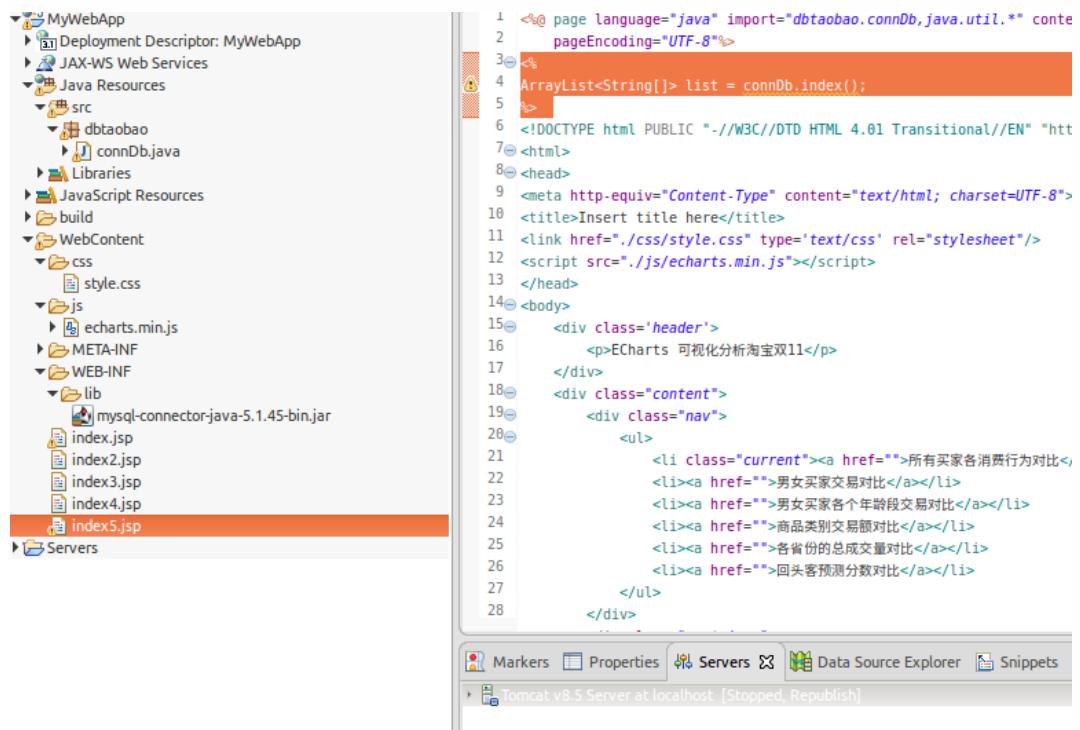
1. cd ~/下载/
2. unzip mysql-connector-java-5.1.40.zip -d ~
3. cd ~/mysql-connector-java-5.1.40/
4. mv ./mysql-connector-java-5.1.40-bin.jar ~/workspace/MyWebApp/WebContent/WEB-INF/lib/mysql-connector-java-5.1.40-bin.jar

```

上述操作完成后，即可开发可视化应用了。

利用 Eclipse 开发 Dynamic Web Project 应用

整个项目开发完毕的项目结构，如下：



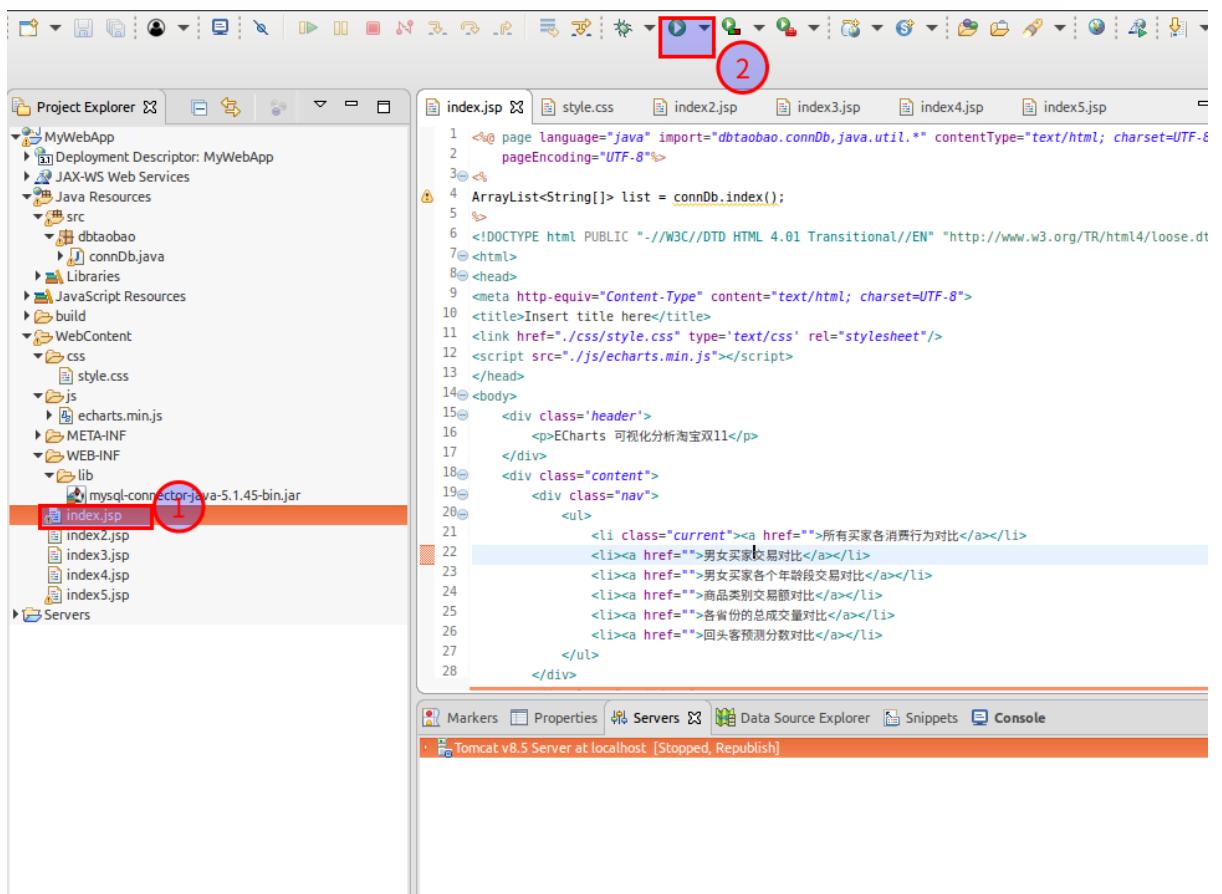
src 目录用来存放服务端 Java 代码，WebContent 用来存放前端页面的文件资源与代码。其中 css 目录用来存放外部样式表文件、

font 目录用来存放字体文件、**img** 目录存放图片资源文件、**js** 目录存放 JavaScript 文件，**lib** 目录存放 Java 与 mysql 的连接库。

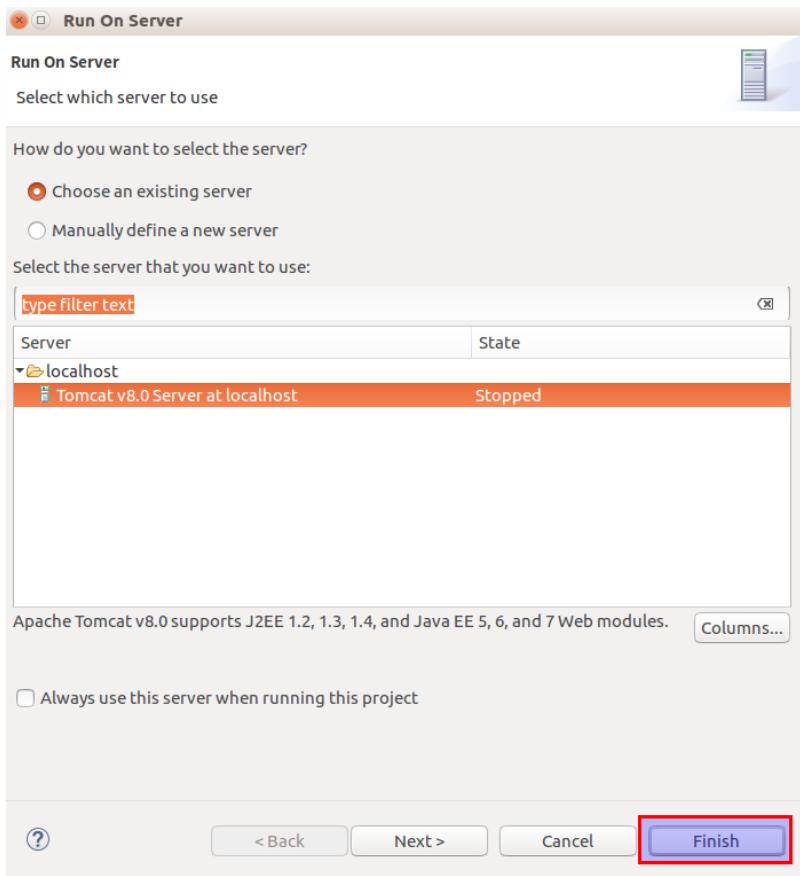
这里没有列出如何创建文件的图例，请读者自己去创建。创建完所有的文件后，运行 MyWebApp，查看我的应用。首次运行 MyWebApp，请按照如下操作，才能启动项目：

双击打开 **index.jsp** 文件，然后顶部 Run 菜单选择：Run As-

->Run on Server

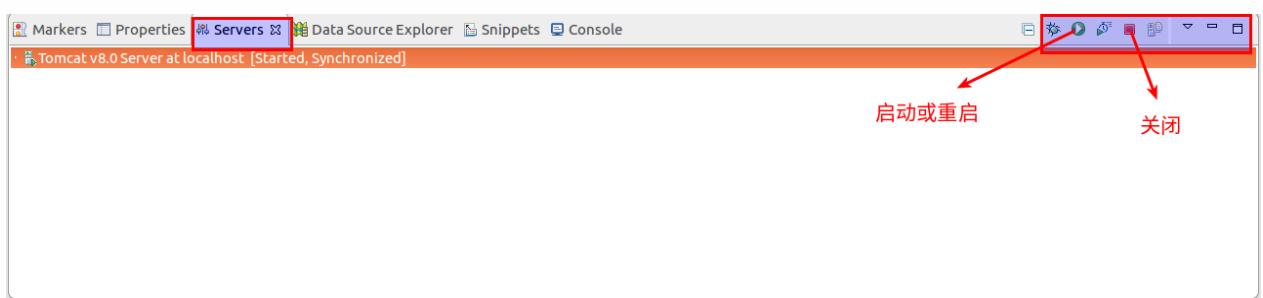


出现如下对话框，直接点击 **finish** 即可。



此时通过外部浏览器，例如火狐浏览器，打开 MyWebApp 地址，也能查看到该项目应用。

以后如果要再次运行 MyWebApp，只需要直接启动 Tomcat 服务器即可，关闭服务器也可以通过如下图关闭。



重要代码解析：

服务端代码解析：

整个项目，Java 后端从数据库中查询的代码都集中在项目文件夹下/Java Resources/src/dbtaobao/connDb.java

代码如下：

```
1. package dbtaobao;
2. import java.sql.*;
3. import java.util.ArrayList;
4.
5. public class connDb {
6.     private static Connection con = null;
7.     private static Statement stmt = null;
8.     private static ResultSet rs = null;
9.
10.    //连接数据库方法
11.    public static void startConn(){
12.        try{
13.            Class.forName("com.mysql.jdbc.Driver");
14.            //连接数据库中间件
15.            try{
16.                con = DriverManager.getConnection("jdbc:MySQL://localhost:3306/dbtaobao","root","root");
17.            }catch(SQLException e){
18.                e.printStackTrace();
19.            }
20.            }catch(ClassNotFoundException e){
21.                e.printStackTrace();
22.            }
23.        }
24.
25.    //关闭连接数据库方法
26.    public static void endConn() throws SQLException{
27.        if(con != null){
28.            con.close();
29.            con = null;
30.        }
31.        if(rs != null){
32.            rs.close();
```

```
33.         rs = null;
34.     }
35.     if(stmt != null){
36.         stmt.close();
37.         stmt = null;
38.     }
39. }
40. //数据库双11 所有买家消费行为比例
41. public static ArrayList index() throws SQLException{
42.     ArrayList<String[]> list = new ArrayList();
43.     startConn();
44.     stmt = con.createStatement();
45.     rs = stmt.executeQuery("select action,count(*) num from user_log group by action desc");
46.     while(rs.next()){
47.         String[] temp={rs.getString("action"),rs.getString("num")};
48.         list.add(temp);
49.     }
50.     endConn();
51.     return list;
52. }
53. //男女买家交易对比
54. public static ArrayList index_1() throws SQLException{
55.     ArrayList<String[]> list = new ArrayList();
56.     startConn();
57.     stmt = con.createStatement();
58.     rs = stmt.executeQuery("select gender,count(*) num from user_log group by gender desc");
59.     while(rs.next()){
60.         String[] temp={rs.getString("gender"),rs.getString("num")};
61.         list.add(temp);
62.     }
63.     endConn();
64.     return list;
65. }
66. //男女买家各个年龄段交易对比
67. public static ArrayList index_2() throws SQLException{
68.     ArrayList<String[]> list = new ArrayList();
69.     startConn();
70.     stmt = con.createStatement();
71.     rs = stmt.executeQuery("select gender,age_range,count(*) num from user_log group by gender,age_range desc");
```

```
72.         while(rs.next()){
73.             String[] temp={rs.getString("gender"),rs.getString("ag
e_range"),rs.getString("num")};
74.             list.add(temp);
75.         }
76.         endConn();
77.         return list;
78.     }
79.     //获取销量前五的商品类别
80.     public static ArrayList index_3() throws SQLException{
81.         ArrayList<String[]> list = new ArrayList();
82.         startConn();
83.         stmt = con.createStatement();
84.         rs = stmt.executeQuery("select cat_id,count(*) num from us
er_log group by cat_id order by count(*) desc limit 5");
85.         while(rs.next()){
86.             String[] temp={rs.getString("cat_id"),rs.getString("nu
m")};
87.             list.add(temp);
88.         }
89.         endConn();
90.         return list;
91.     }
92.     //各个省份的总成交量对比
93.     public static ArrayList index_4() throws SQLException{
94.         ArrayList<String[]> list = new ArrayList();
95.         startConn();
96.         stmt = con.createStatement();
97.         rs = stmt.executeQuery("select province,count(*) num from user
_log group by province order by count(*) desc");
98.         while(rs.next()){
99.             String[] temp={rs.getString("province"),rs.getString("num
")};
100.             list.add(temp);
101.         }
102.         endConn();
103.         return list;
104.     }
105. }
```

前端代码解析：

前端页面想要获取服务端的数据，还需要导入相关的包，例如：

/WebContent/index.jsp 部分（jsp）代码如下：

```
1. <%@ page language="java" import="dbtaobao.connDb,  
   java.util.*" contentType="text/html; charset=UTF-  
   8"  
  
2.      pageEncoding="UTF-8"%>  
  
3. <%  
  
4. ArrayList<String[]> list = connDb.index();  
  
5. %>
```

前端 JSP 页面使用 ECharts 来展现可视化。每个 JSP 页面都需要导入相关 ECharts.js 文件，如需要中国地图的可视化，还需要另外导入 china.js 文件。

那么如何使用 ECharts 的可视化逻辑代码，我们在每个 jsp 的底部编写可视化逻辑代码。这里展示 index.jsp 中可视化逻辑代码：

```
1. <script>  
2. // 基于准备好的dom，初始化echarts实例  
3. var myChart = echarts.init(document.getElementById('main'));  
4. // 指定图表的配置项和数据  
5. option = {  
6.     backgroundColor: '#2c343c',  
7.     title: {  
8.         text: '所有买家消费行为比例图',  
9.         left: 'center',  
10.        top: 20,  
11.        textStyle: {
```

```
13.             color: '#ccc'
14.         }
15.     },
16.
17.     tooltip : {
18.         trigger: 'item',
19.         formatter: "{a} <br/>{b} : {c} ({d}%)"
20.     },
21.
22.     visualMap: {
23.         show: false,
24.         min: 80,
25.         max: 600,
26.         inRange: {
27.             colorLightness: [0, 1]
28.         }
29.     },
30.     series : [
31.         {
32.             name:'消费行为',
33.             type:'pie',
34.             radius : '55%',
35.             center: ['50%', '50%'],
36.             data:[
37.                 {value:<%=list.get(0)[1]%, name:'特别关注'},
38.                 {value:<%=list.get(1)[1]%, name:'购买'},
39.                 {value:<%=list.get(2)[1]%, name:'添加购物车'},
40.                 {value:<%=list.get(3)[1]%, name:'点击'}
41.             ].sort(function (a, b) { return a.value - b.value
e}),
42.             roseType: 'angle',
43.             label: {
44.                 normal: {
45.                     textStyle: {
46.                         color: 'rgba(255, 255, 255, 0.3)'
47.                     }
48.                 }
49.             },
50.             labelLine: {
51.                 normal: {
52.                     lineStyle: {
53.                         color: 'rgba(255, 255, 255, 0.3)'
54.                     },
55.                     smooth: 0.2,
```

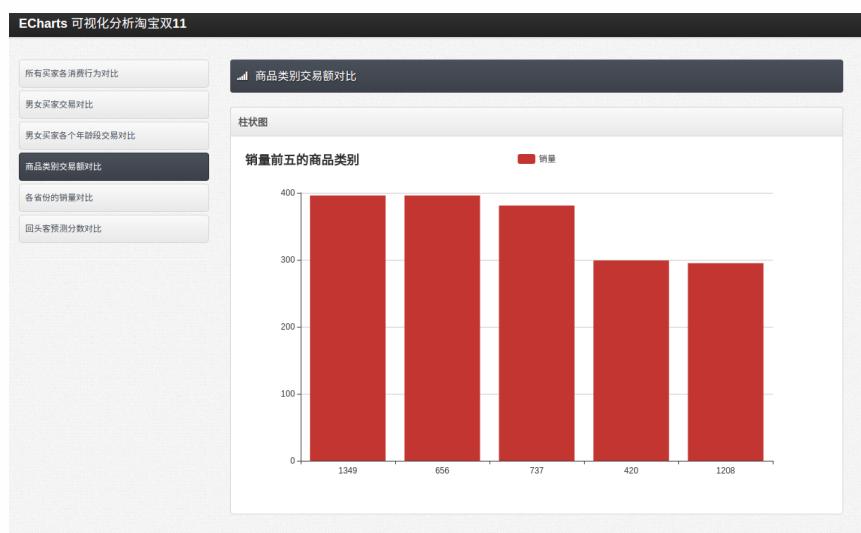
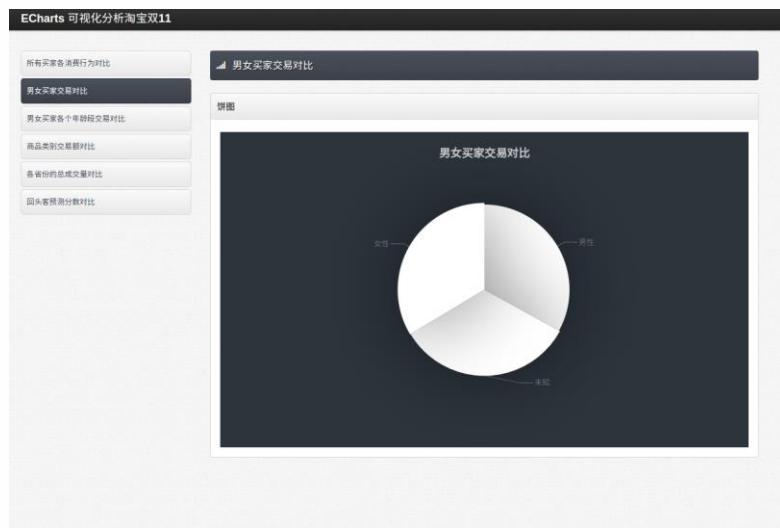
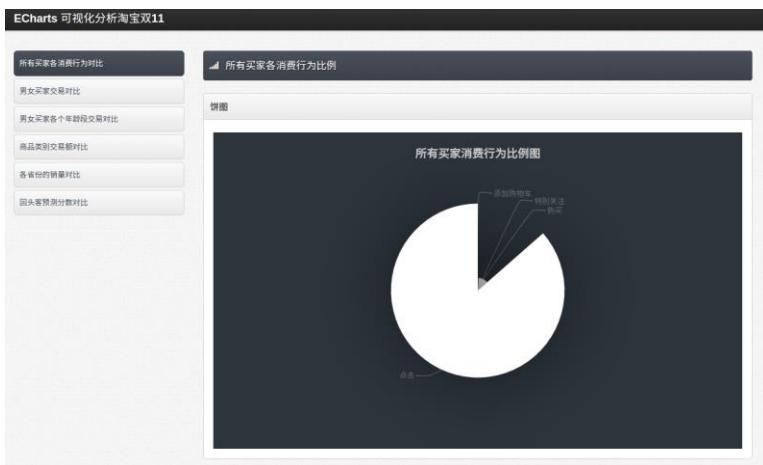
```
56.           length: 10,
57.           length2: 20
58.       },
59.   },
60.   itemStyle: {
61.     normal: {
62.       color: '#c23531',
63.       shadowBlur: 200,
64.       shadowColor: 'rgba(0, 0, 0, 0.5)'
65.     }
66.   },
67.
68.   animationType: 'scale',
69.   animationEasing: 'elasticOut',
70.   animationDelay: function (idx) {
71.     return Math.random() * 200;
72.   }
73. }
74. ]
75. );
76.
77. // 使用刚指定的配置项和数据显示图表。
78. myChart.setOption(option);
79.</script>
```

ECharts 包含各种各样的可视化图形，每种图形的逻辑代码，请参考 ECharts 官方示例代码,请读者自己参考 index.jsp 中的代码,再根据 ECharts 官方示例代码,自行完成其他可视化比较。

页面效果:

最终,我自己使用饼图,散点图,柱状图,地图等完成了如下效果,读者如果觉得有更适合的可视化图形,也可以自己另行修改。

最后展示所有页面的效果图：



拓展案例及往届优秀作业

拓展案例

1、拓展—海难幸存者预测分析

本次实验需要各位结合以往所学以及大数据课程的知识，使用 kaggle 上的著名数据集 Titanic 进行海难幸存者预测分析：根据训练集中的乘客数据和存活情况进行建模，进而使用分类模型预测测试集中的乘客是否会存活。

2、数据集简介

数据分为两组：训练集（train.csv），测试集（test.csv）

训练集用于构建机器学习模型。训练集提供了每位乘客的结果。模型将基于乘客的性别和年龄等特征，还可以使用特征工程来创建新特征。

测试集用于查看模型在看不见的数据上的表现如何。测试集不提供每位乘客的结果，对于测试集中的每位乘客，使用训练的模型来预测他们是否在泰坦尼克号沉没中幸存下来。

部分训练数据与测试数据：

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
1	0	3	Braund, Mr. Owen Gridley	male	22	1	0	A/5 21171	7.25	null	S
2	1	1	Cumings, Mrs. John Bra...	female	38	1	0	PC 17599	71.2833	C85	C
3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2. 3101282	7.925	null	S
4	1	1	Futrelle, Mrs. Jaqueline	female	35	1	0	113803	53.1	C123	S
5	0	3	Allen, Mr. William Henry	male	35	0	0	373450	8.05	null	S

only showing top 5 rows

```

+-----+-----+-----+-----+-----+-----+-----+-----+
| PassengerId | Pclass |          Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
+-----+-----+-----+-----+-----+-----+-----+-----+
|     892 |     3 | Kelly, Mr. James | male | 34.5 |    0 |    0 | 330911 | 7.8292 | null |      Q |
|     893 |     3 | Wilkes, Mrs. Jame... | female | 47 |    1 |    0 | 363272 |      7 | null |      S |
|     894 |     2 | Myles, Mr. Thomas... | male | 62 |    0 |    0 | 240276 | 9.6875 | null |      Q |
|     895 |     3 | Wirz, Mr. Albert | male | 27 |    0 |    0 | 315154 | 8.6625 | null |      S |
|     896 |     3 | Hirvonen, Mrs. Al... | female | 22 |    1 |    1 | 3101298 | 12.2875 | null |      S |
+-----+-----+-----+-----+-----+-----+-----+-----+
only showing top 5 rows

root
├— PassengerId: integer (nullable = true)
├— Pclass: string (nullable = true)
├— Name: string (nullable = true)
├— Sex: string (nullable = true)
├— SibSp: integer (nullable = true)
├— Parch: integer (nullable = true)
├— Fare: double (nullable = true)
├— Embarked: string (nullable = true)
└— AgeC: double (nullable = true)

```

3、实验要求：

本次实验的核心内容是使用 spark 相关操作对原始数据进行处理以提升模型准确性，要求各位通过适当的特征选取、缺失值处理、数据归一化等方法处理数据，并通过 spark Mllib 库中自带的逻辑回归、决策树等方法训练模型得到尽可能高的准确率。

实验允许各位使用擅长的编程环境和编程工具，只要使用 spark 的方法进行处理即可。

相关文件在大数据部分安装包/拓展案例中，代码海难幸存者预测.ipynb仅供参考。

往届优秀项目

中国 2021 年气象数据分析

一、前言

本案例利用 Spark 提供的 Java API，完成了对中国 2021 年气象

数据的简单分析，可供大数据学习者参考。此文涵盖了从环境配置到数据分析的所有内容，力争做到详略得当。读者需具有 Java 基础，熟悉 Linux 的基本操作，同时了解 Spark 的相关概念。

说明：

宿主机：安装虚拟机软件（如 VMWare Workstation）的主机，搭载 Windows 系统。

虚拟机：虚拟机软件，安装 Ubuntu 系统。

教程环境：

Ubuntu 20.04

JDK 8（虚拟机）、JDK11（宿主机）

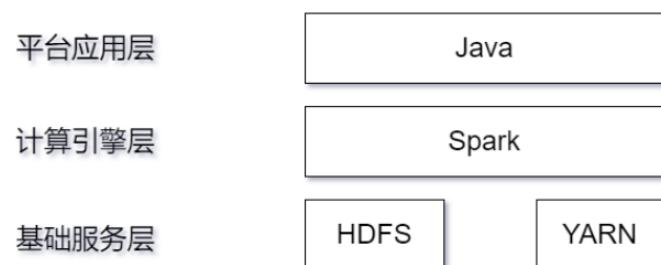
Hadoop 2.7.7Spark 3.1.2

Scala 2.12

Maven 3.8.1

MySQL 8.0.24

项目架构图：



二、数据获取

数据集为中国 2021 年的气象数据，来自 NCDC（美国国家气候数据中心）。数据集包括中国 369 个监测站点全年的气象数据，数据属性包括温度、风速、降雨量等，如下图所示。

STATION	NAME	LATITUDE	LONGITUDE	ELEVATION	DATE	MAX	MIN	PRCP	TEMP	WDSP	
0	56586099999	ZHAOTONG, CH	27.333333	103.75	1950.0	2021-01-01	41.9	25.2	0.0	32.8	1.9
1	56586099999	ZHAOTONG, CH	27.333333	103.75	1950.0	2021-01-02	53.4	25.5	0.0	34.9	3.9
2	56586099999	ZHAOTONG, CH	27.333333	103.75	1950.0	2021-01-03	53.4	27.1	0.0	30.0	2.8
3	56586099999	ZHAOTONG, CH	27.333333	103.75	1950.0	2021-01-04	43.3	27.5	0.0	33.0	3.9
4	56586099999	ZHAOTONG, CH	27.333333	103.75	1950.0	2021-01-05	43.3	28.6	0.0	30.7	3.2

数据网站：<https://www.ncei.noaa.gov/maps/daily/>

数据下载参考：<https://blog.csdn.net/lianqian66/article/details/121210437>

数据集属性信息：

Station：监测点编号

Name；监测点名称

Latitude：监测点纬度

Longitude：监测点经度

Elevation：监测点海拔

Date：监测日期

Max：最高温度

Min：最低温度

Prcp：降水量

Temp：平均温度

Wdsp：风向风速

安装虚拟机、hadoop、JDK，hadoop 使用配置及用户登录等步骤

参照上方教程即可。

其他安装配置：安装和配置 Maven：从官网下载 Maven 并完成安装，教程中使用的是 Maven 3.8.1；安装和配置 MySQL：从官网下载 MySQL 并完成安装，教程中使用的是 MySQL 8.0.24 版本，官网：

<https://downloads.mysql.com/archives/community/>。

安装完成后，启动 MySQL 服务，使用 root 用户登录，新建一个名为 spark 的数据库，字符集采用 utf8。在 spark 数据库下新建三张表，temperature，precipitation，windspeed，表字段及相应的数据

结构参照下图：



部分字段的含义：

温度表：Max（当日最高温），Min（当日最低温），Avg（当日平均温度）。

降雨量表：Sid（监测站编号）、Precipitation（监测点年降雨量）。

4、新建 Maven 项目

注：教程中使用的 Idea 版本为 2021.2.1，高版本的 Maven 和低版本的 Idea 可能会存在兼容性问题。在 Idea 中新建 maven 项目 Spark，修改 pom.xml 文件，声明项目所需的依赖

```

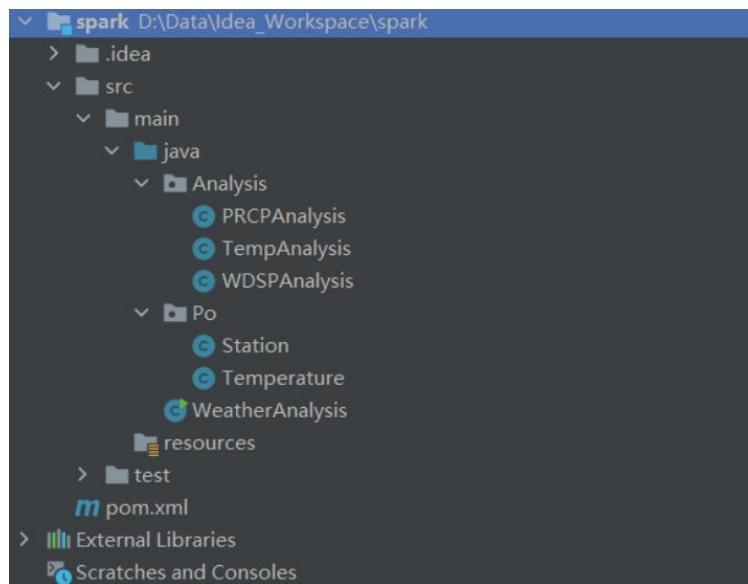
1 <!---pom.xml---->
2 <?xml version="1.0" encoding="UTF-8"?>
3 <project xmlns="http://maven.apache.org/POM/4.0.0"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
6   http://maven.apache.org/xsd/maven-4.0.0.xsd">
7
8   <modelVersion>4.0.0</modelVersion>
9   <groupId>com.example</groupId>
10  <artifactId>Spark</artifactId>
11  <version>1.0-SNAPSHOT</version>
12
13  <properties>
14    <maven.compiler.source>11</maven.compiler.source>
15    <maven.compiler.target>11</maven.compiler.target>
16    <!-----声明spark和scala的版本----->
17    <spark.version>3.1.2</spark.version>
18    <scala.version>2.12</scala.version>
19  </properties>
20
21  <dependencies>
22
23    <!-----添加不含hadoop hdfs客户端的spark核心依赖----->
24    <dependency>
25      <groupId>org.apache.spark</groupId>
26      <artifactId>spark-core_${scala.version}</artifactId>
27      <version>${spark.version}</version>
28      <exclusions>
29        <exclusion>
30          <groupId>org.apache.hadoop</groupId>
31          <artifactId>hadoop-hdfs-client</artifactId>
32        </exclusion>
33      </exclusions>
34    </dependency>
35    <!-----添加hadoop通用依赖----->
36    <dependency>
37      <groupId>org.apache.hadoop</groupId>
38      <artifactId>hadoop-common</artifactId>
39      <version>2.7.7</version>
40    </dependency>
41    <!-----添加hadoop hdfs依赖----->
42    <dependency>
43      <groupId>org.apache.hadoop</groupId>
44      <artifactId>hadoop-hdfs</artifactId>
45      <version>2.7.7</version>
46    </dependency>
47    <!-----添加mysql驱动依赖----->
48    <dependency>
49      <groupId>mysql</groupId>
50      <artifactId>mysql-connector-java</artifactId>
51      <version>8.0.29</version>
52    </dependency>

```

5、编写业务代码

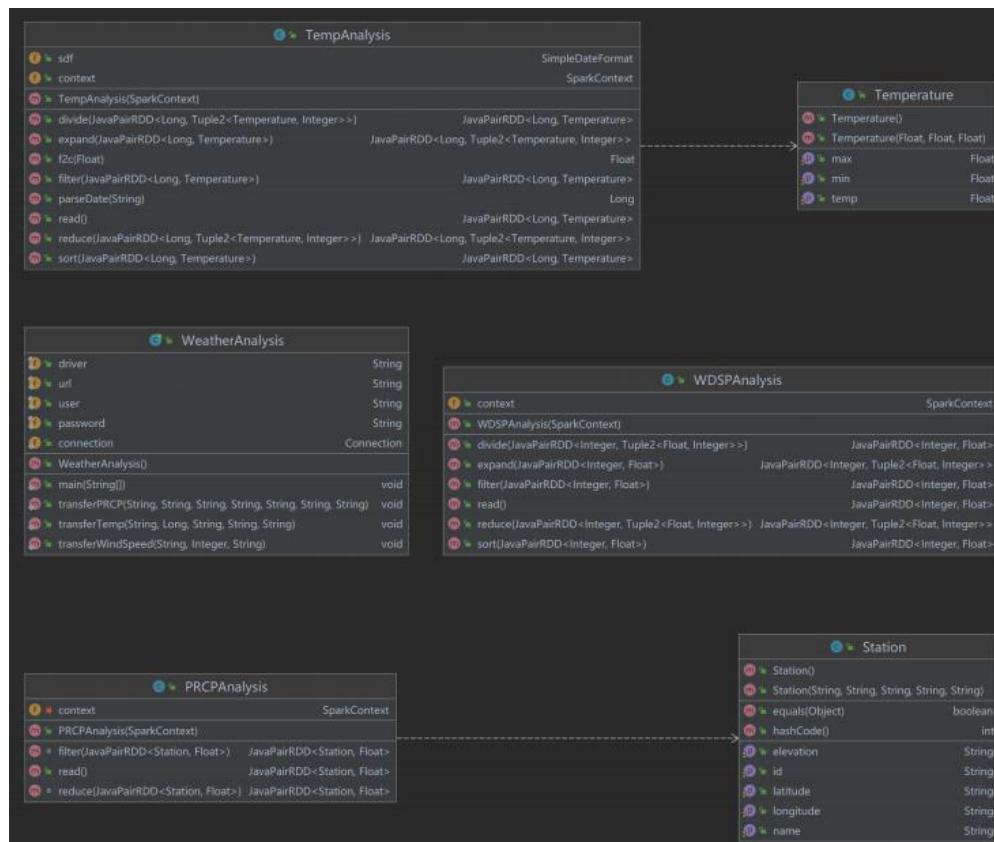
待 maven 将所有依赖下载完成后，进入编码环节。本项目中一共涉及到六个类，两个实体类 Station 和 Temperature，三个功能类

PRCPAnalysis、TempAnalysis 与 WDSPAnalysis，以及一个主类 WeatherAnalysis。为了便于管理，建立了两个包 Analysis 和 Po 来分别放置，项目目录结构如下图所示。



具体代码见 PDF 文件

最终项目类结构如下图所示：



6、运行项目

代码编写完毕，确认虚拟机 Ubuntu 系统的 HDFS 服务和宿主机系统的 MySQL 服务均已启动，运行项目，观察控制台输出和数据库变化。

控制台输出中可能会给出下图中的 Failed to locate the winutils binary in the hadoop binary path 错误提示，一般不影响使用。

```
22/08/29 09:50:51 ERROR Shell: Failed to locate the winutils binary in the hadoop binary path
java.io.IOException Create breakpoint : Could not locate executable null\bin\winutils.exe in the Hadoop binaries.
    at org.apache.hadoop.util.Shell.getQualifiedBinPath(Shell.java:382)
    at org.apache.hadoop.util.Shell.getWinUtilsPath(Shell.java:397)
    at org.apache.hadoop.util.Shell.<clinit>(Shell.java:390)
    at org.apache.hadoop.util.StringUtils.<clinit>(StringUtils.java:80)
    at org.apache.hadoop.security.SecurityUtil.getAuthenticationMethod(SecurityUtil.java:610)
    at org.apache.hadoop.security.UserGroupInformation.initialize(UserGroupInformation.java:277)
    at org.apache.hadoop.security.UserGroupInformation.ensureInitialized(UserGroupInformation.java:265)
    at org.apache.hadoop.security.UserGroupInformation.loginUserFromSubject(UserGroupInformation.java:810)
    at org.apache.hadoop.security.UserGroupInformation.getLoginUser(UserGroupInformation.java:788)
    at org.apache.hadoop.security.UserGroupInformation.getCurrentUser(UserGroupInformation.java:653)
    at org.apache.spark.util.Utils$.anonfun$getCurrentUserName$1(Utils.scala:2487)
    at scala.Option.getOrElse(Option.scala:189)
    at org.apache.spark.util.Utils$.getCurrentUserName(Utils.scala:2487)
    at org.apache.spark.SparkContext.<init>(SparkContext.scala:314)
    at WeatherAnalysis.main(WeatherAnalysis.java:23)
```

第一条输出：

```
22/08/29 09:51:04 INFO DAGScheduler: Job 4 finished: first at WeatherAnalysis.java:41, took 0.072323 s
22/08/29 09:51:04 INFO MemoryStore: Block broadcast_8 stored as values in memory (estimated size 211.1 KiB, free
1031.5 MiB)
1609430400000 -4.6044745 2.4100308 -11.655251
22/08/29 09:51:04 INFO MemoryStore: Block broadcast_8_piece0 stored as bytes in memory (estimated size 30.8 KiB,
free 1031.5 MiB)
22/08/29 09:51:04 WARN ProcfsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of
ProcessTree metrics is stopped
22/08/29 09:51:04 INFO BlockManagerInfo: Added broadcast_8_piece0 in memory on DESKTOP-DUSEVP7:58463 (size: 30.8
KiB, free: 1031.9 MiB)
```

第二条输出：

```
22/08/29 09:51:06 INFO DAGScheduler: Job 11 is finished. Cancelling potential speculative or zombie tasks for this
job
22/08/29 09:51:06 INFO TaskSchedulerImpl: Killing all running tasks in stage 28: Stage finished
22/08/29 09:51:06 INFO DAGScheduler: Job 11 finished: first at WeatherAnalysis.java:57, took 0.038277 s
54049099999 CHANGLING 44.25 123.9666666 190 630.936
22/08/29 09:51:06 INFO MemoryStore: Block broadcast_17 stored as values in memory (estimated size 211.1 KiB, free
1031.3 MiB)
22/08/29 09:51:06 INFO MemoryStore: Block broadcast_17_piece0 stored as bytes in memory (estimated size 30.8 KiB,
free 1031.3 MiB)
22/08/29 09:51:06 INFO BlockManagerInfo: Added broadcast_17_piece0 in memory on DESKTOP-DUSEVP7:58463 (size: 30.8
KiB, free: 1031.9 MiB)
```

第三条输出：

```
22/08/29 09:51:07 INFO DAGScheduler: Job 13 finished: collect at WeatherAnalysis.java:74, took 0.024691 s
22/08/29 09:51:07 INFO SparkContext: Invoking stop() from shutdown hook
[(1,4.771494), (2,4.9828167), (3,5.2540975), (4,5.582341), (5,5.650656), (6,4.919014), (7,4.7592273), (8,4.4316473),
 , (9,4.2401843), (10,4.5061283), (11,4.5359645), (12,4.2395554)]
22/08/29 09:51:07 INFO SparkUI: Stopped Spark web UI at http://DESKTOP-DUSEVP7:4040
22/08/29 09:51:07 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
22/08/29 09:51:07 INFO MemoryStore: MemoryStore cleared
```

最后程序以 0 代码结束，说明程序运行成功。（Pr 第二条输出：

ocess finished with exit code 0) 数据库表：

Temperature:

T WHERE		E ORDER BY			
	id	Date	Max	Min	Avg
1	1816	1609430400000	2.4100308	-11.655251	-4.6044745
2	1817	1609516800000	3.5604925	-10.637193	-3.446913
3	1818	1609603200000	3.6158948	-9.6642	-3.215896
4	1819	1609689600000	3.8043947	-9.209531	-3.0467355
5	1820	1609776000000	2.8935184	-9.3586445	-3.972995
6	1821	1609862400000	1.3348765	-11.295525	-6.191978
7	1822	1609948800000	-1.0472223	-13.216974	-8.223305
8	1823	1610035200000	-1.296145	-13.355318	-7.0566087
9	1824	1610121600000	0.5287036	-12.243363	-5.1623464
10	1825	1610208000000	1.2760804	-10.9404335	-4.4141984
11	1826	1610294400000	1.8604938	-10.721143	-4.4481463
12	1827	1610380800000	5.2601833	-10.22932	-1.8628091
13	1828	1610467200000	7.5467596	-9.302777	-0.80925936
14	1829	1610553600000	8.600616	-8.48318	-0.1560186
15	1830	1610640000000	7.6991634	-9.028938	-1.4331478
16	1831	1610726400000	5.62469	-9.649229	-3.0625002

Precipitation:

Screenshot of a database interface showing the 'precipitation' table. The table has 369 rows and contains columns: id, Sid, Name, Latitude, Longitude, Elevation, and Precipitation.

	id	Sid	Name	Latitude	Longitude	Elevation	Precipitation
1	1963	54049099999	CHANGLING	44.25	123.9666666	190	630.936
2	1964	52533199999	ZHONGCHUAN	36.515231	103.620078	1947.06	8.0
3	1965	52836099999	DOULAN	36.3	98.1	3190	329.94592
4	1966	53564099999	HEQU	39.3666666	111.2166666	1037	308.60986
5	1967	55664099999	TINGRI	28.6333333	87.0833333	4300	308.35596
6	1968	54602099999	BAODING	38.7333333	115.4833333	17	959.61206
7	1969	54374099999	LINJIANG	41.8	126.9166666	381	807.21246
8	1970	59855099999	QIONGHAII	19.2333333	110.4666666	25	1976.1201
9	1971	53480099999	JINING	41.0333333	113.0666666	1416	334.7719
10	1972	52866099999	XINING	36.6166667	101.7666667	2262	477.51996
11	1973	58238099999	LUKOU	31.742042	118.862025	14.93	1265.4282
12	1974	56786099999	ZHANYI	25.5833333	103.8333333	1900	657.8604
13	1975	50745099999	SANJIAZI	47.239628	123.918131	145.38	677.4184
14	1976	53772099999	WUSU	37.746897	112.628428	784.86	605.0281
15	1977	57067099999	LUSHI	34.0833333	111.0666666	659.5	846.0742
16	1978	56096099999	WUDU	33.4	104.9166666	1079	624.3321

WindSpeed:

Screenshot of a database interface showing the 'windspeed' table. The table has 12 rows and contains columns: id, Month, and WindSpeed.

	id	Month	WindSpeed
1	49	1	4.771494
2	50	2	4.9828167
3	51	3	5.2540975
4	52	4	5.582341
5	53	5	5.650656
6	54	6	4.919014
7	55	7	4.7592273
8	56	8	4.4316473
9	57	9	4.2401843
10	58	10	4.5061283
11	59	11	4.5359645
12	60	12	4.2395554

五、后记

本教程涉及的大数据处理内容较为简单，读者可更换数据集进行更多的复杂处理。受自身水平限制，文中难免出现部分疏漏，还望读者批评指正。

教程中用到的数据集和软件已随教程给出，读者也可以自己从官网下载。

作者：胡声洋 邮箱：rsune@qq.com

常见问题

一、spark 独立编程报错：

```
>>> from pyspark import SparkContext
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/spark/python/pyspark/__init__.py", line 53, in <module>
    from pyspark.rdd import RDD, RDDBarrier
  File "/usr/local/spark/python/pyspark/rdd.py", line 48, in <module>
    from pyspark.traceback_utils import SCCallSiteSync
  File "/usr/local/spark/python/pyspark/traceback_utils.py", line 23, in <module>
    CallSite = namedtuple("CallSite", "function file linenum")
  File "/usr/local/spark/python/pyspark/serializers.py", line 390, in namedtuple
    for k, v in _old_ntuple_kwdefaults.items():
AttributeError: 'NoneType' object has no attribute 'items'
```

原因：pyspark 版本过高，与 ubuntu 内置 python 版本不兼容

解决方案：卸载 pyspark，重新安装较低版本（如 2.4.7）

二、“py4j.protocol.Py4JError: An error occurred while calling None”

原因：环境变量中的 py4j 版本与实际版本不符

解决方案：查询本机的 py4j 版本，修改环境变量(vim ~/.bashrc ;
source ~/bashrc)

三、

```
>>> val textFile = sc.textFile("hdfs://localhost:9000/user/liyu/word.txt")
  File "<stdin>", line 1
    val textFile = sc.textFile("hdfs://localhost:9000/user/liyu/word.txt")
          ^
SyntaxError: invalid syntax
```

解决：把 ‘val’ 去掉

四、zookeeper 启动失败

如果你的压缩包是 3.5 以上的版本，随着版本的更新，3.5 版本以后的压缩包分成了两种我们需要使用文件名带有 bin 的那个压缩包，例如：apache-zookeeper-3.5.9-bin.tar.gz 这样解压后才会有 lib 目录下的那些 jar 包

五、spark SQL 插入中文时报错

ERROR 1366 (HY000)错误类型

在插入中文时，报错显示这种错误，是因为编码的问题，应该选择 utf8 类型编码。用以下编码就能解决：

```
1. alter table table_name(表名) convert to character set utf8 ;
```