案例一

**1、实验概要**

在前面学习中使用Scrapy添加Cookie抓取数据，接下来通过爬虫招聘网站作为项目实验，来深入了解下Scrapy。招聘网站的数据需要获取到相应的cookie才能加载数据，而且每次登陆之后cookie会改变，我们需要找到并手动添加到程序中。

将在windows和linux环境下实现spark对招聘数据的清洗，案例涉及数据的读取、清洗、过滤、数据预处理、数据分析以及代码编写等。本案例适合高校大数据教学，可以作为学习大数据课程后的综合实践案例。通过本案例，将有助于读者综合运用大数据课程知识以及各种工具软件，实现数据全流程操作。

编写一个Flask可视化程序，读取已经清洗好的数并存储到mysql的数据，编写Flask需要的使用python安装flask框架，它是python的web框架，此外还要下载ehcarts等js包，其主要的作用调用可视化模块显示结果。

**2、实验内容**

* 登录cookie爬虫
* 网页post表单加载数据
* 安装Flask框架及其依赖和Echarts
* 获取MYSQL数据库数据

**3、实验目标**

* 了解登录表单与登陆cookie爬虫
* 掌握含有登录cookie网络爬虫代码
* 熟悉spark数据清洗以及分析的一般流程
* 学会使用scala语言编写spark程序进行数据的清洗、过滤、预处理以及数据分析
* 掌握spark的一些原理以及机制
* 学会使用sqoop进行hive到mysql的数据传输
* 熟悉hive的基本语法

**4、环境要求**

* 实验主机：
  + Windows10主机一台
* 所需软件：
  + 正常连接互联网并获取免费开源软件
* 实验是否需要联网：是

本案例可以在单机上完成，也可以在集群环境下完成。单机上完成本案例实验时，建议计算机硬件配置为：500GB以上硬盘，8GB以上内存。

为了顺利完成本案例各项实验，需要完成以下系统和软件的安装，来满足实验能顺利运行所需的环境：

* Windows安装IDEA、JDK1.8.0以及scala2.11.8
* Linux安装spark2.0.0、hadoop2.6.0、JDK1.8.0、scala2.11.8、hive1.1.0以及mysql5.17

**6实验二 数据清洗与数据分析**

## 6.1实验数据

将实验数据上传至HDFS的/recruitdata/目录下

|  |
| --- |
| [hadoop@master ~]$ hdfs dfs -mkdir /recruitdata  [hadoop@master ~]$ hdfs dfs -put recruit.json /recruitdata |



## 6.2实验步骤

**6.1.1实验任务一：将字段中存在空值的数据以及“薪资”字段不合规的数据剔除出原始数据集(RecruitTask1.scala)**

**1 知识要点**

（1）创建sparkcontext

|  |
| --- |
| //设置程序名为RddSparkApp，并设置Master为spark://master:7077  val conf = new SparkConf().setAppName("RddSparkApp").setMaster("spark://master:7077")  val sc = new SparkContext(conf) |

1. 创建累加器，用来保存剔除的记录条数

|  |
| --- |
| //创建累加器  val accum = sc.accumulator(0, "Example Accumulator") |

1. 读取指定路径下的文件

|  |
| --- |
| val rdd1 = sc.textFile("hdfs://master:9000/recruitdata") |

1. 由于字段中存在\r和\n这些特殊符号，需要删除这些特殊符号，否则会影响后续的清洗操作

map(x => {方法体})中的x表示的是上一操作的返回值，这里表示的是读取的每一行数据，方法体中可以编写各种对x的处理，并返回最终结果，当方法体内的代码只有一行时，可以省略{}，scala默认会将最后一行代码作为返回值，不需要return关键字，该返回值会作为下一个操作的输入

|  |
| --- |
| //消除字段中的回车符和换行符  .map(x => x.replaceAll("\\\\r|\\\\n", "")) |

1. 查看原始数据集，可以看到第一行和最后一行的数据是无意义的，故剔除第一行和最后一行数据

filter(x => {方法体})，x为上一操作的返回值，这里为上一步剔除\r和\n 后的数据，方法体需要返回布尔类型，可以使用x进行条件判断，如果返回true，则保留这一行数据，如返回false，则删除这一行数据。

|  |
| --- |
| //过滤掉第一行跟最后一行  .filter(x => !(x.contains("[")||x.contains("]")||x.equals("")||x==null)) |

1. 查看原始数据集，可以看到每行json数据末尾都有一个逗号，为了使每行数据都符合json格式，需要剔除每行数据末尾的逗号

|  |
| --- |
| .map(x => {  //删除读取每行最后的逗号  var rawJson = x.substring(0, x.length()-1)  //由于上一条语句可能会造成读取的行数据末尾缺少一个},所以需确认读取的行结尾是否}}结尾，即是否符合一条json数据格式，如不符合则添加多一个}。  if(!rawJson.substring(rawJson.length()-2, rawJson.length()).equals("}}")) {  rawJson = rawJson+"}"  }  rawJson  }) |

1. 使用fastjson对每行数据进行json解析，并使用数组存放每个字段的数据，返回该数组

.getOrDefault(x,y)方法中需要传入两个参数，x表示提取json数据中哪个key的数据，y表示默认值，如果找不到x设置的key，则这个方法返回默认值，否则返回key对应的value值

|  |
| --- |
| .map(x => {  //json解析，并赋值  val jsonObject = JSON.parseObject(x)  val name = jsonObject.getOrDefault("name","").toString  val values = jsonObject.getOrDefault("detail","").toString  val jsonObject2 = JSON.parseObject(values)  val company\_name = jsonObject2.getOrDefault("公司名称","").toString  val city = jsonObject2.getOrDefault("工作城市","").toString  val require = jsonObject2.getOrDefault("工作要求","").toString  val man = jsonObject2.getOrDefault("招聘人数","").toString  val cost\_max = jsonObject2.getOrDefault("薪资上限","").toString  val cost\_min = jsonObject2.getOrDefault("薪资下限","").toString  val skill = jsonObject2.getOrDefault("工作技术","").toString  val time = jsonObject2.getOrDefault("发布时间","").toString  val sex = jsonObject2.getOrDefault("招聘性别","").toString  val introduce = jsonObject2.getOrDefault("公司描述","").toString  val need = jsonObject2.getOrDefault("学历要求","").toString  Array(name,company\_name,city,require,man,cost\_max,cost\_min,skill,time,sex,introduce,need)  }) |

1. 遍历上一步返回的数组，剔除字段为空的数据，即如果数组中含有空的值，则剔除这一行

|  |
| --- |
| //剔除字段为空的数据  .filter(x => {  //count用来存放数组中有多少空字段  var count = 0  for (y <- x){  if(y.equals("")||y==null){  count += 1  }  }  if(count==0){  true  }else{  false  }  }) |

1. 剔除不合规的薪资数据，并使用上面定义的累加器存放删除的记录条数

|  |
| --- |
| //薪资字段如果出现-，则表示薪资存在负值，需要剔除,并使计数器+1  .filter(x => {  //x(5)和x(6)分别为薪资的上限和下限  if(x(5).contains("-")||x(6).contains("-")){  accum += 1  false  }else{  true  }  }) |

1. 剔除工作技术字段中的【】

|  |
| --- |
| .map(x => {  //x(7)为工作技术字段  x(7) = x(7).replaceAll("【|】", "")  x  }) |

1. 遍历上一步返回的结果，将结果数组中的数据用|拼凑成字符串

|  |
| --- |
| .map(x => {  var result = ""  for( i <- x) {  if(i.equals(x(x.size-1))) {  result = result+i.trim()  }else {  result = result+i.trim()+"|"  }  }  result  }) |

1. 输出最终结果到指定目录

|  |
| --- |
| rdd1.saveAsTextFile("hdfs://master:9000/recruitspark1") |

1. 打印累加器的值，即在控制台打印删除的记录数

|  |
| --- |
| println("-------------删除的记录数为"+accum.value+"-------------") |

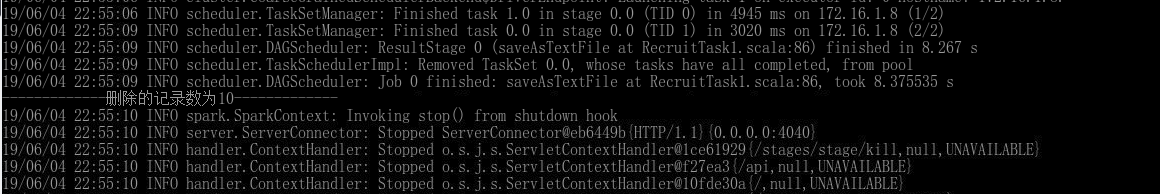
1. spark执行程序命令

|  |
| --- |
| #首先进入spark的bin目录  [hadoop@master ~]$ cd /usr/local/src/spark/bin/  #spark-submit --class 主程序的类名 jar包路径  [hadoop@master bin]$ ./spark-submit --class RecruitTask1 /home/hadoop/RecruitTask1.jar |

**2 完整代码**

|  |
| --- |
| import org.apache.spark.{SparkConf, SparkContext}  import com.alibaba.fastjson.JSON  object RecruitTask1 {  def main(args: Array[String]): Unit = {  val conf = new SparkConf().setAppName("RddSparkApp").setMaster("spark://master:7077")  val sc = new SparkContext(conf)  val accum = sc.accumulator(0, "Example Accumulator")  //读取指定路径的数据  val rdd1 = sc.textFile("hdfs://master:9000/recruitdata")  //消除字段中的回车符和换行符  .map(x => x.replaceAll("\\\\r|\\\\n", ""))  //过滤掉第一行跟最后一行  .filter(x => !(x.contains("[")||x.contains("]")||x.equals("")||x==null))  .map(x => {  //删除读取每行最后的逗号  var rawJson = x.substring(0, x.length()-1)  //由于上一条语句可能会造成读取的行数据末尾缺少一个},所以需确认读取的行结尾是否}}结尾，即是否符合一条json数据格式，如不符合则添加多一个}。  if(!rawJson.substring(rawJson.length()-2, rawJson.length()).equals("}}")) {  rawJson = rawJson+"}"  }  rawJson  })  .map(x => {  //json解析，并赋值  val jsonObject = JSON.parseObject(x)  val name = jsonObject.getOrDefault("name","").toString  val values = jsonObject.getOrDefault("detail","").toString  val jsonObject2 = JSON.parseObject(values)  val company\_name = jsonObject2.getOrDefault("公司名称","").toString  val city = jsonObject2.getOrDefault("工作城市","").toString  val require = jsonObject2.getOrDefault("工作要求","").toString  val man = jsonObject2.getOrDefault("招聘人数","").toString  val cost\_max = jsonObject2.getOrDefault("薪资上限","").toString  val cost\_min = jsonObject2.getOrDefault("薪资下限","").toString  val skill = jsonObject2.getOrDefault("工作技术","").toString  val time = jsonObject2.getOrDefault("发布时间","").toString  val sex = jsonObject2.getOrDefault("招聘性别","").toString  val introduce = jsonObject2.getOrDefault("公司描述","").toString  val need = jsonObject2.getOrDefault("学历要求","").toString  Array(name,company\_name,city,require,man,cost\_max,cost\_min,skill,time,sex,introduce,need)  })  //剔除字段为空的数据  .filter(x => {  var count = 0  for (y <- x){  if(y.equals("")||y==null){  count += 1  }  }  if(count==0){  true  }else{  false  }  })  //以下为剔除不合规的工资数据  //薪资字段如果出现-，则表示薪资存在负值，需要剔除,并使计数器+1  .filter(x => {  if(x(5).contains("-")||x(6).contains("-")){  accum += 1  false  }else{  true  }  })  //剔除工作技术的【】  .map(x => {  x(7) = x(7).replaceAll("【|】", "")  x  })  //遍历添加分隔符  .map(x => {  var result = ""  for( i <- x) {  if(i.equals(x(x.size-1))) {  result = result+i.trim()  }else {  result = result+i.trim()+"|"  }  }  result  })  //输出文件  rdd1.saveAsTextFile("hdfs://master:9000/recruitspark1")  println("-------------删除的记录数为"+accum.value+"-------------")  }  } |

**3 样例输出**



**6.1.2 实验任务二：将薪资格式统一为“下限-上限”，对个别下限大于上限的数据进行次序调整(RecruitTask2.scala)**

**1 知识要点**

1. 对输入的每一行数据按|进行切割

|  |
| --- |
| .map(x => x.split("\\|")) |

1. 对薪资的两个字段进行大小比较，使用累加器记录个别下限大于上限的数据条数，将两个薪资字段合并为下限-上限的格式

|  |
| --- |
| .map(x => {  var cost = ""  if(x(5).toInt>=x(6).toInt){  cost = x(6)+"-"+x(5)  }else{  accum += 1  cost = x(5)+"-"+x(6)  }  Array(x(0),x(1),x(2),x(3),x(4),cost,x(7),x(8),x(9),x(10),x(11))  }) |

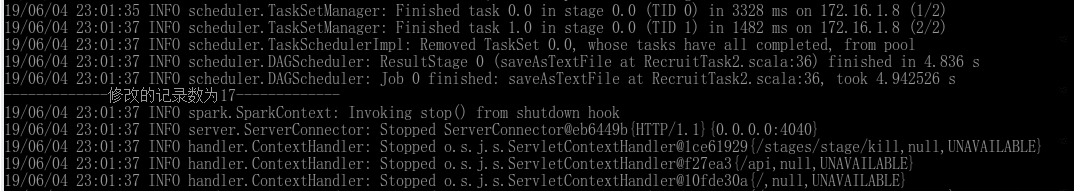
1. 遍历上一步返回的结果，将结果数组中的数据用|拼凑成字符串

|  |
| --- |
| //遍历添加分隔符|  .map(x => {  var result = ""  for( i <- x) {  if(i.equals(x(x.size-1))) {  result = result+i.trim()  }else {  result = result+i.trim()+"|"  }  }  result  }) |

**2 完整代码**

|  |
| --- |
| import org.apache.spark.{SparkConf, SparkContext}  object RecruitTask2 {  def main(args: Array[String]): Unit = {  val conf = new SparkConf().setAppName("RddSparkApp").setMaster("spark://master:7077")  val sc = new SparkContext(conf)  //创建累加器  val accum = sc.accumulator(0, "Example Accumulator")  //读取指定路径的数据  val rdd1 = sc.textFile("hdfs://master:9000/recruitspark1")  .map(x => x.split("\\|"))  //对薪资的两个字段进行判断，并修改为下限-上限的格式  .map(x => {  var cost = ""  if(x(5).toInt>=x(6).toInt){  cost = x(6)+"-"+x(5)  }else{  accum += 1  cost = x(5)+"-"+x(6)  }  Array(x(0),x(1),x(2),x(3),x(4),cost,x(7),x(8),x(9),x(10),x(11))  })  //遍历添加分隔符|  .map(x => {  var result = ""  for( i <- x) {  if(i.equals(x(x.size-1))) {  result = result+i.trim()  }else {  result = result+i.trim()+"|"  }  }  result  })  //输出文件  rdd1.saveAsTextFile("hdfs://master:9000/recruitspark2")  println("-------------修改的记录数为"+accum.value+"-------------")  }  } |

**3 样例输出**



**6.1.3 实验任务三：将日期字段中的日期格式统一为yyyy-mm-dd(RecruitTask3.scala)**

**1 知识要点**

1. 查看原始数据集，建立时间的数据映射表

|  |
| --- |
| val timemap = new util.HashMap[String,String]()  timemap.put("Jan", "01")  timemap.put("Feb", "02")  timemap.put("Mar", "03")  timemap.put("Apr", "04")  timemap.put("May", "05")  timemap.put("Jun", "06")  timemap.put("Jul", "07")  timemap.put("Aug", "08")  timemap.put("Sept", "09")  timemap.put("Oct", "10")  timemap.put("Nov", "11")  timemap.put("Dec", "12") |

1. 查看原始数据集，发现有六种不合规的数据格式，编译正则表达式，提取里面的年月日

|  |
| --- |
| //Dec 29 2008 11:45 PM  val partition1 = "([a-zA-Z]{3,4}) ([0-9]{2}) ([0-9]{4}) [0-9]{1,2}:[0-9]{1,2} [a-zA-Z]{2}"  //12-29-2008  val partition2 = "([0-9]{2})-([0-9]{2})-([0-9]{4})"  //29 Dec 2008 16:25:46  val partition3 = "([0-9]{2}) ([a-zA-Z]{3,4}) ([0-9]{4}) [0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}"  //18th,Apr,2019,10:52:53:000000  val partition4 = "([0-9]{2})[a-z]{2},([a-zA-Z]{3,4}),([0-9]{4}),[0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}:[0-9]{6}"  //Apr,18th,2019,10:46 AM  val partition5 = "([a-zA-Z]{3,4}),([0-9]{2})[a-z]{2},([0-9]{4}),[0-9]{1,2}:[0-9]{1,2} [a-zA-Z]{2}"  //18th,Apr,2019 10:52:53  val partition6 = "([0-9]{2})[a-z]{2},([a-zA-Z]{3,4}),([0-9]{4}) [0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}"  //编译正则表达式  val r1 = Pattern.compile(partition1)  val r2 = Pattern.compile(partition2)  val r3 = Pattern.compile(partition3)  val r4 = Pattern.compile(partition4)  val r5 = Pattern.compile(partition5)  val r6 = Pattern.compile(partition6) |

1. 对输入的每一行数据按|进行切割

|  |
| --- |
| .map(x => x.split("\\|")) |

1. 使用正则表达式对日期字段进行匹配，提取年月日，并修改为yyyy-mm-dd的格式

|  |
| --- |
| .map(x => {  var m = r1.matcher(x(7))  if(m.find()) {  //如果匹配到则说明格式需要修改，累加器+1，提取年月日并修改为yyyy-mm-dd  accum += 1  x(7) = m.group(3) + "-" + timemap.get(m.group(1)) + "-" + m.group(2)  }else{  m = r2.matcher(x(7))  if (m.find) {  accum += 1  x(7) = m.group(3) + "-" + m.group(2) + "-" + m.group(1)  }  else {  m = r3.matcher(x(7))  if (m.find) {  accum += 1  x(7) = m.group(3) + "-" + timemap.get(m.group(2)) + "-" + m.group(1)  }else{  m = r4.matcher(x(7))  if(m.find()){  accum += 1  x(7) = m.group(3) + "-" + timemap.get(m.group(2)) + "-" + m.group(1)  }else{  m = r5.matcher(x(7))  if(m.find()){  accum += 1  x(7) = m.group(3) + "-" + timemap.get(m.group(1)) + "-" + m.group(2)  }else{  m = r6.matcher(x(7))  if(m.find()){  accum += 1  x(7) = m.group(3) + "-" + timemap.get(m.group(2)) + "-" + m.group(1)  }  }  }  }  }  }  x  }) |

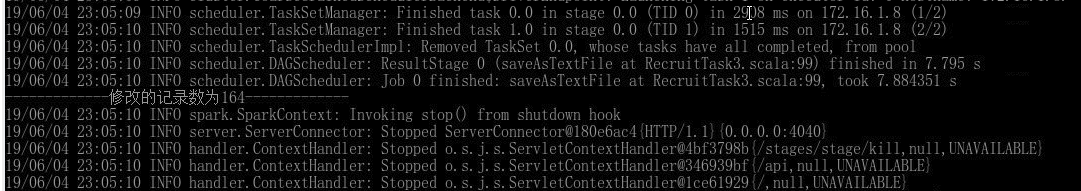
1. 遍历上一步返回的结果，将结果数组中的数据用|拼凑成字符串

|  |
| --- |
| .map(x => {  var result = ""  for( i <- x) {  if(i.equals(x(x.size-1))) {  result = result+i.trim()  }else {  result = result+i.trim()+"|"  }  }  result  }) |

**2 完整代码**

|  |
| --- |
| import java.util  import java.util.regex.Pattern  import org.apache.spark.{SparkConf, SparkContext}  object RecruitTask3 {  def main(args: Array[String]): Unit = {  val conf = new SparkConf().setAppName("RddSparkApp").setMaster("spark://master:7077")  val sc = new SparkContext(conf)  //建立时间的数据映射  val timemap = new util.HashMap[String,String]()  timemap.put("Jan", "01")  timemap.put("Feb", "02")  timemap.put("Mar", "03")  timemap.put("Apr", "04")  timemap.put("May", "05")  timemap.put("Jun", "06")  timemap.put("Jul", "07")  timemap.put("Aug", "08")  timemap.put("Sept", "09")  timemap.put("Oct", "10")  timemap.put("Nov", "11")  timemap.put("Dec", "12")  //根据三种格式，创建正则表达式  //Dec 29 2008 11:45 PM  val partition1 = "([a-zA-Z]{3,4}) ([0-9]{2}) ([0-9]{4}) [0-9]{1,2}:[0-9]{1,2} [a-zA-Z]{2}"  //12-29-2008  val partition2 = "([0-9]{2})-([0-9]{2})-([0-9]{4})"  //29 Dec 2008 16:25:46  val partition3 = "([0-9]{2}) ([a-zA-Z]{3,4}) ([0-9]{4}) [0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}"  //18th,Apr,2019,10:52:53:000000  val partition4 = "([0-9]{2})[a-z]{2},([a-zA-Z]{3,4}),([0-9]{4}),[0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}:[0-9]{6}"  //Apr,18th,2019,10:46 AM  val partition5 = "([a-zA-Z]{3,4}),([0-9]{2})[a-z]{2},([0-9]{4}),[0-9]{1,2}:[0-9]{1,2} [a-zA-Z]{2}"  //18th,Apr,2019 10:52:53  val partition6 = "([0-9]{2})[a-z]{2},([a-zA-Z]{3,4}),([0-9]{4}) [0-9]{1,2}:[0-9]{1,2}:[0-9]{1,2}"  //编译正则表达式  val r1 = Pattern.compile(partition1)  val r2 = Pattern.compile(partition2)  val r3 = Pattern.compile(partition3)  val r4 = Pattern.compile(partition4)  val r5 = Pattern.compile(partition5)  val r6 = Pattern.compile(partition6)  //创建累加器  val accum = sc.accumulator(0, "Example Accumulator")  //读取指定路径的数据  val rdd1 = sc.textFile("hdfs://master:9000/recruitspark2")  .map(x => x.split("\\|"))  //对x(7)字段进行匹配，并修改成yyyy-mm-dd  .map(x => {  var m = r1.matcher(x(7))  if(m.find()) {  accum += 1  x(7) = m.group(3) + "-" + timemap.get(m.group(1)) + "-" + m.group(2)  }else{  m = r2.matcher(x(7))  if (m.find) {  accum += 1  x(7) = m.group(3) + "-" + m.group(2) + "-" + m.group(1)  }  else {  m = r3.matcher(x(7))  if (m.find) {  accum += 1  x(7) = m.group(3) + "-" + timemap.get(m.group(2)) + "-" + m.group(1)  }else{  m = r4.matcher(x(7))  if(m.find()){  x(7) = m.group(3) + "-" + timemap.get(m.group(2)) + "-" + m.group(1)  }else{  m = r5.matcher(x(7))  if(m.find()){  x(7) = m.group(3) + "-" + timemap.get(m.group(1)) + "-" + m.group(2)  }else{  m = r6.matcher(x(7))  if(m.find()){  x(7) = m.group(3) + "-" + timemap.get(m.group(2)) + "-" + m.group(1)  }  }  }  }  }  }  x  })  //遍历添加分隔符  .map(x => {  var result = ""  for( i <- x) {  if(i.equals(x(x.size-1))) {  result = result+i.trim()  }else {  result = result+i.trim()+"|"  }  }  result  })  //输出路径  rdd1.saveAsTextFile("hdfs://master:9000/recruitspark3")  println("-------------修改的记录数为"+accum.value+"-------------")  }  } |

4.3.3 样例输出



**6.1.4 实验任务四：将清洗后的招聘数据导入到hive中**

1. 在hive中创建recruitdata库

hive> create database recruitdata;

1. 在recruitdata库中创建rawdata表

hive> use recruitdata;

hive> create table rawdata (name String, com\_name String, city String, requirement String, number String, salary String, skill String, time String, gender String, intro String, edu String) row format delimited fields terminated by "|" stored as textfile;

row format delimited fields terminated by "|" 的意义为指定“|”为字段间的分隔符

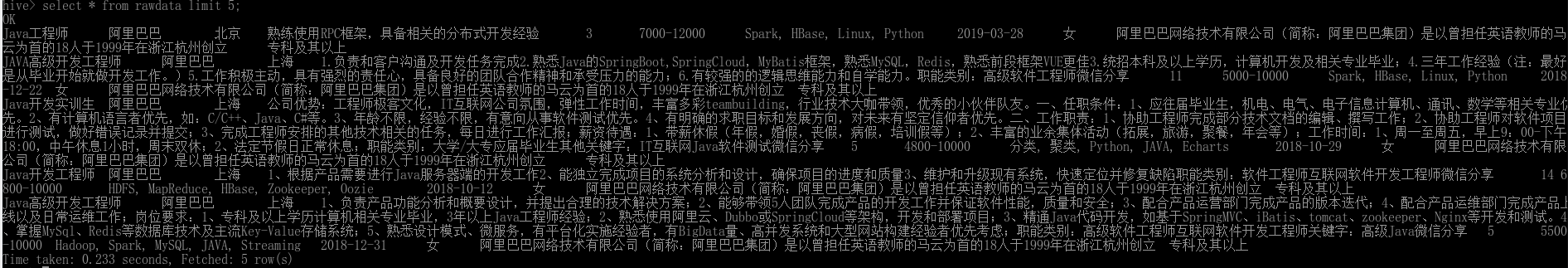
stored as textfile 的意义为保存为纯文本文件，如果数据需要压缩，可以使用STORED AS SEQUENCEFILE

1. 将清洗后的招聘数据导入rawdata表中

hive> load data inpath "/recruitspark3/\*" into table rawdata;

1. 查询rawdata表前5行数据

hive> select \* from rawdata limit 5;



1. 匹配职位名称中包含“大数据”的数据，将这些数据的工作技术字段抽取出来保存至hdfs指定目录中

hive> insert overwrite directory '/wordcount' select skill from rawdata where name like '%大数据%';

**6.1.5 实验任务五：统计职位名称中包含“大数据”的所有工作技术标签出现的频次(recruit:com.org.task4)**

**1 知识要点**

**CleanMapper.java**

1. 创建一个CleanMapper类，继承了org.apache.hadoop.mapreduce.Mapper，该类提供了用户自定义map操作的方法，Mapper后面的<>里的参数意义为<输入的key类型,输入的value类型，输出的key类型，输出的value类型>，以上的类型均为hadoop的io类型，常见的hadoop io类型为Text、IntWritable、LongWritable、FloatWritable等

|  |
| --- |
| public class CleanMapper extends Mapper<LongWritable,Text,Text,LongWritable>{} |

1. map方法体内就可以写自己map阶段的操作逻辑，key为输入数据的key，此处为读取文件指针的偏移量，value为输入数据的值，此处为一行数据

|  |
| --- |
| public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {} |

1. 将输入的每一行数据以逗号分隔，并遍历分隔后的数据，将所有的数据以(key,value)的形式输出，这里的key为单个工作技术标签，value为1，即出现频次为一次，此处输出的数据类型要与类创建时<>里输出的数据类型一致

|  |
| --- |
| String str = value.toString();  String[] datas = str.split(", ");  for(String x:datas) {  context.write(new Text(x.toLowerCase()), new LongWritable(1));  } |

**CleanReducer.java**

1. 创建一个CleanReducer类，继承了hadoop包中的Reducer类，该类提供了用户自定义reduce操作的方法，Reducer后面的<>里的参数意义为<输入的key类型(要与前面mapper输出的key类型一致),输入的value类型(要与前面mapper输出的value类型一致)，输出的key类型，输出的value类型>

|  |
| --- |
| public class CleanReducer extends Reducer<Text,LongWritable,Text,NullWritable>{} |

1. reduce方法体内就可以写自己reduce阶段的操作逻辑，key是前面map的输出key，values为相同key的value的集合

|  |
| --- |
| public void reduce(Text key, Iterable<LongWritable> values, Context context) throws IOException, InterruptedException {} |

1. 定义一个长整型，用来存放单个工作技术标签出现的频次，遍历values，用长整型来累加每个value，最终获得一个工作技术标签出现的总频次，并输出最终结果(工作标签|频次,null)

|  |
| --- |
| long i = 0;  for(LongWritable x:values) {  i = i + x.get();  }  context.write(new Text(key+"|"+String.valueOf(i)), NullWritable.get()); |

**CleanJob.java**

1. CleanJob类为程序的主入口
2. 首先需要通过hadoop的configuration类创建一个Job

|  |
| --- |
| Configuration conf = new Configuration();  Job job = Job.getInstance(conf); |

1. 设置job的参数

|  |
| --- |
| //设置job类  job.setJarByClass(CleanJob.class);  //设置map和reduce的类  job.setMapperClass(CleanMapper.class);  job.setReducerClass(CleanReducer.class);  //设置map输出key，value类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(LongWritable.class);  //设置reduce输出的key，value类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(NullWritable.class); |

1. 设置输入的hdfs路径以及输出的hdfs路径

|  |
| --- |
| //设置输入的hdfs路径  FileInputFormat.addInputPath(job, new Path("/wordcount"));  //设置输出的hdfs路径，注意：hdfs不能存在输出的文件夹，mapreduce会自动创建  FileOutputFormat.setOutputPath(job, new Path("/file3\_5")); |

1. 等待job执行完后，结束整个程序

|  |
| --- |
| System.exit(job.waitForCompletion(true)?0:1); |

**2 完整代码**

**CleanMapper.java**

|  |
| --- |
| package com.org.task4;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Mapper;  public class CleanMapper extends Mapper<LongWritable,Text,Text,LongWritable>{  public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  String str = value.toString();  String[] datas = str.split(", ");  for(String x:datas) {  context.write(new Text(x.toLowerCase()), new LongWritable(1));  }  }  } |

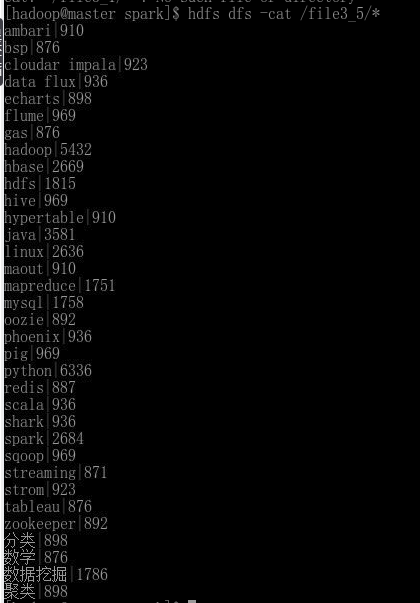
**CleanReducer.java**

|  |
| --- |
| package com.org.task4;  import java.io.IOException;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Reducer;  public class CleanReducer extends Reducer<Text,LongWritable,Text,NullWritable>{  public void reduce(Text key, Iterable<LongWritable> values, Context context) throws IOException, InterruptedException {  long i = 0;  for(LongWritable x:values) {  i = i + x.get();  }  context.write(new Text(key+"|"+String.valueOf(i)), NullWritable.get());  }  } |

**CleanJob.java**

|  |
| --- |
| package com.org.task4;  import java.io.IOException;  import org.apache.hadoop.conf.Configuration;  import org.apache.hadoop.fs.Path;  import org.apache.hadoop.io.LongWritable;  import org.apache.hadoop.io.NullWritable;  import org.apache.hadoop.io.Text;  import org.apache.hadoop.mapreduce.Job;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  public class CleanJob {  public static void main(String[] args) throws IOException, ClassNotFoundException, InterruptedException {  // TODO Auto-generated method stub  Configuration conf = new Configuration();  Job job = Job.getInstance(conf);  //设置job类  job.setJarByClass(CleanJob.class);  //设置map和reduce的类  job.setMapperClass(CleanMapper.class);  job.setReducerClass(CleanReducer.class);  //设置map输出key，value类型  job.setMapOutputKeyClass(Text.class);  job.setMapOutputValueClass(LongWritable.class);  //设置reduce输出的key，value类型  job.setOutputKeyClass(Text.class);  job.setOutputValueClass(NullWritable.class);  //设置输入的hdfs路径  FileInputFormat.addInputPath(job, new Path("/wordcount"));  //设置输出的hdfs路径，注意：hdfs不能存在输出的文件夹，mapreduce会自动创建  FileOutputFormat.setOutputPath(job, new Path("/file3\_5"));  System.exit(job.waitForCompletion(true)?0:1);  }  } |

**3 样例输出**



**6.16 实验任务六：编写hive自定义函数，实现薪资均值的计算(下限和上限的均值)，并使用hive引入该自定义函数，统计各城市平均薪资(recruit:com.org.udf)**

**1 知识要点**

**AvgCost.java**

1. 该类继承了hive的UDF类，UDF类提供了一个方法evaluate()，用户可以通过重写这个方法来实现函数的自定义，其中传入的参数类型和返回的数据类型可以根据自己处理的逻辑来定，在本次任务中，我们需要对薪资字段进行均值计算，所以这里传入的参数为一个String类型即可，返回值也设置为String类型

|  |
| --- |
| public class AvgCost extends UDF{  public String evaluate(String raw){  }  } |

1. 将输入的数据按“-”切割，并提取出最大值和最小值，最后返回平均值

|  |
| --- |
| String[] data = raw.split("-");  int min = Integer.parseInt(data[0]);  int max = Integer.parseInt(data[1]);  return String.valueOf((min+max)/2); |

1. 编写好自定义函数以后，我们将这个java文件打成jar包，然后在hive中引入该函数

hive> use recruitdata;

hive> add jar /home/hadoop/Recruit.jar;

hive> create temporary function avgcost as 'com.org.udf.AvgCost';

1. 创建好表table3\_6，用来存放查询结果

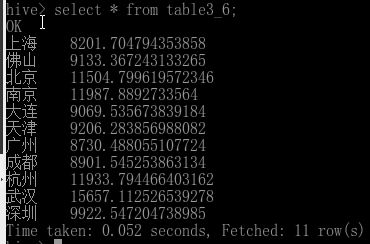
hive> create table table3\_6 (city String, avg\_cost String) row format delimited fields terminated by "|" stored as textfile;

1. 使用hql语句统计各城市的平均薪资，并写入到table3\_6中

hive> insert overwrite table table3\_6 select city,avg(avgcost(salary)) from rawdata group by city;

1. 查看表中的内容

hive> select \* from table3\_6;



**2 完整代码**

|  |
| --- |
| package com.org.udf;  import org.apache.hadoop.hive.ql.exec.UDF;  public class AvgCost extends UDF{  public String evaluate(String raw){  String[] data = raw.split("-");  int min = Integer.parseInt(data[0]);  int max = Integer.parseInt(data[1]);  return String.valueOf((min+max)/2);  }  } |

**6.1.7实验任务七：使用hive统计各岗位招聘数量，并将结果通过sqoop导入到mysql中**

1. 在hive的recruitdata库中创建表table3\_7\_1，用来存放查询得到的结果

hive> create table table3\_7\_1 (name String, number String) row format delimited fields terminated by "|" stored as textfile;

1. 使用hql语句统计各岗位的招聘数量，并将结果保存到table3\_7\_1中

hive> insert overwrite table table3\_7\_1 select name,count(\*) from rawdata group by name;

1. 在mysql中创建recruitdata数据库以及table3\_7\_2表

mysql> create database recruitdata default character set utf8 collate utf8\_general\_ci;

mysql> use recruitdata;

mysql> create table table3\_7\_2(city varchar(255) not null, number varchar(255) not null) default charset=utf8;

1. 使用sqoop将hive的table3\_7\_1表中的数据导入到mysql的table3\_7\_2中

[hadoop@master spark]$ sqoop export --connect "jdbc:mysql://localhost:3306/recruitdata?useUnicode=true&characterEncoding=utf8" --username root --password Password123$ --table table3\_7\_2 -input-fields-terminated-by "|" --export-dir /user/hive/warehouse/recruitdata.db/table3\_7\_1

1. 导入完后在mysql中查询岗位数量前5的数据

mysql> select \* from table3\_7\_2 order by cast(number as signed) desc limit 5;

