|  |
| --- |
| 葵花宝典 |
|  |
| **编者：郑可佳**  **日期：2017/4/17** |
|  |



目 录

[葵花宝典 0](#_Toc491018879)

[1 Linux基础 5](#_Toc491018880)

[1.1 Linux分区操作 5](#_Toc491018881)

[1.2 Linux命令 5](#_Toc491018882)

[1.2.1 grep命令 5](#_Toc491018883)

[1.2.2 Sed命令 5](#_Toc491018884)

[1.2.3 awk命令 6](#_Toc491018885)

[1.2.4 uniq命令 6](#_Toc491018886)

[1.2.5 查看某端口是否被占用 6](#_Toc491018887)

[1.2.6 移动文件 6](#_Toc491018888)

[1.2.7 命令行修改替换文件 6](#_Toc491018889)

[1.3 Shell编程 7](#_Toc491018890)

[1.3.1 变量 7](#_Toc491018891)

[1.3.2 Shell算数运算 8](#_Toc491018892)

[1.3.3 Shell内置测试判断 8](#_Toc491018893)

[1.3.4 控制语句 11](#_Toc491018894)

[1.3.5 函数使用 14](#_Toc491018895)

[1.3.6 Crontab计划任务 15](#_Toc491018896)

[2 Windows系统执行hadoop程序 16](#_Toc491018897)

[3 Maven基础 16](#_Toc491018898)

[3.1 环境搭建 16](#_Toc491018899)

[3.2 pom文件 16](#_Toc491018900)

[3.2.1 配置 16](#_Toc491018901)

[3.3 maven 常用命令 17](#_Toc491018902)

[3.4 Maven配置镜像仓库 17](#_Toc491018903)

[3.5 dos命令生成Maven骨架 17](#_Toc491018904)

[3.6 maven 依赖传递 18](#_Toc491018905)

[3.7 maven生命周期 18](#_Toc491018906)

[3.8 maven 常见错误 18](#_Toc491018907)

[4 Hadoop基础 18](#_Toc491018908)

[4.1 Hadoop安装 18](#_Toc491018909)

[4.2 MapReduce深入理解 18](#_Toc491018910)

[4.3 MapReduce二次排序 19](#_Toc491018911)

[4.3.1 组合key 19](#_Toc491018912)

[4.3.2 分区规则 20](#_Toc491018913)

[4.3.3 分组规则 21](#_Toc491018914)

[4.4 MapReduce Join 22](#_Toc491018915)

[4.5 HA机制 23](#_Toc491018916)

[4.5.1 需求背景 23](#_Toc491018917)

[4.5.2 HA配置 23](#_Toc491018918)

[4.6 Resourcemanager HA 23](#_Toc491018919)

[4.7 HDFS Federation 23](#_Toc491018920)

[4.8 分布式拷贝工具DistCP 23](#_Toc491018921)

[5 Zookeeper基础 24](#_Toc491018922)

[5.1 Zookeeper安装 24](#_Toc491018923)

[6 Hive基础 24](#_Toc491018924)

[6.1 MySQL安装 24](#_Toc491018925)

[6.2 Hive安装 24](#_Toc491018926)

[6.3 Hive使用 24](#_Toc491018927)

[6.3.1 分区表 24](#_Toc491018928)

[6.3.2 加载数据 25](#_Toc491018929)

[6.3.3 导出数据 25](#_Toc491018930)

[6.3.4 having关键词 25](#_Toc491018931)

[6.3.5 数据导入导出 26](#_Toc491018932)

[6.3.6 四个by关键词 26](#_Toc491018933)

[6.3.7 UDF函数 27](#_Toc491018934)

[6.3.8 Hive的三种启动方式 27](#_Toc491018935)

[6.3.9 数据存储 27](#_Toc491018936)

[6.3.10 SQL执行Python脚本 30](#_Toc491018937)

[7 sqoop基础 31](#_Toc491018938)

[7.1 sqoop安装 31](#_Toc491018939)

[7.2 sqoop使用 31](#_Toc491018940)

[7.2.1 查看数据库 31](#_Toc491018941)

[7.2.2 导入数据 31](#_Toc491018942)

[7.2.3 导出数据 35](#_Toc491018943)

[7.2.4 Shell导入导出数据 35](#_Toc491018944)

[8 flume基础 36](#_Toc491018945)

[8.1 flume安装 36](#_Toc491018946)

[8.2 flume使用 36](#_Toc491018947)

[8.2.1 安装rpm 36](#_Toc491018948)

[8.2.2 启动xinetd服务 36](#_Toc491018949)

[8.2.3 启动flume服务 36](#_Toc491018950)

[8.2.4 连接a1.conf配置文件指定的端口 37](#_Toc491018951)

[8.2.5 收集hive日志 37](#_Toc491018952)

[8.2.6 动态创建hdfs文件夹 38](#_Toc491018953)

[8.2.7 创建多个agent 38](#_Toc491018954)

[9 Oozie基础 39](#_Toc491018955)

[9.1 Oozie安装 39](#_Toc491018956)

[9.2 运行job 39](#_Toc491018957)

[9.3 编写MapReduce action 39](#_Toc491018958)

[9.4 编写hive action 41](#_Toc491018959)

[9.5 编写sqoop action 42](#_Toc491018960)

[9.6 编写shell action 43](#_Toc491018961)

[9.7 Oozie修改时区 44](#_Toc491018962)

[9.7.1 修改oozie-site.xml文件 44](#_Toc491018963)

[9.7.2 修改web页面显示时区 44](#_Toc491018964)

[9.8 编写定时任务 44](#_Toc491018965)

[9.9 测试cron任务 45](#_Toc491018966)

[9.10 多个action 45](#_Toc491018967)

[10 Hue基础 47](#_Toc491018968)

[10.1 Hue安装 47](#_Toc491018969)

[11 Hbase基础 47](#_Toc491018970)

[11.1 Hbase安装 47](#_Toc491018971)

[11.2 Hbase使用 47](#_Toc491018972)

[11.2.1 Java api操作hbase表 47](#_Toc491018973)

[11.2.2 运行hbase自带MapReduce程序 48](#_Toc491018974)

[11.2.3 编写hbase的MapReduce 48](#_Toc491018975)

[11.2.4 Importtsv导入hdfs文件到hbase表 49](#_Toc491018976)

[11.2.5 bulk批量导入数据到hbase表 49](#_Toc491018977)

[11.2.6 表预分区 50](#_Toc491018978)

[11.2.7 Hbase与hive集成 50](#_Toc491018979)

[11.2.8 Hbase与hue集成 51](#_Toc491018980)

[12 CDH安装 51](#_Toc491018981)

[12.1 CM及CDH5.7安装文档 51](#_Toc491018982)

[13 Kafka基础 51](#_Toc491018983)

[13.1 Kafka安装 51](#_Toc491018984)

[14 Storm基础 51](#_Toc491018985)

[14.1 Storm安装 51](#_Toc491018986)

[14.2 Storm的安装部署架构 52](#_Toc491018987)

[14.3 Storm的任务 52](#_Toc491018988)

[14.4 zookeeper在Storm中的作用 53](#_Toc491018989)

[14.5 实现Topology 53](#_Toc491018990)

[14.5.1 Spout+bolt模式实现 53](#_Toc491018991)

[14.5.2 批处理底层代码实现 58](#_Toc491018992)

[14.5.3 Trident模式实现 58](#_Toc491018993)

[14.5.4 DRPC 62](#_Toc491018994)

[14.5.5 利用Zookeeper锁实现指定时间访问结果 62](#_Toc491018995)

[14.6 Storm批启动脚本编写 62](#_Toc491018996)

[14.6.1 zookeeper批启动脚本 62](#_Toc491018997)

[14.6.2 Storm停止脚本 63](#_Toc491018998)

[14.6.3 批启动脚本 64](#_Toc491018999)

[14.7 storm项目 64](#_Toc491019000)

[14.7.1 需求背景 65](#_Toc491019001)

[14.7.2 测试数据准备 65](#_Toc491019002)

[14.7.3 订单处理 65](#_Toc491019003)

[15 Scala基础 67](#_Toc491019004)

[15.1 Scala基础语法 67](#_Toc491019005)

[15.1.1 for循环 69](#_Toc491019006)

[15.1.2 数组 70](#_Toc491019007)

[15.1.3 集合Map和元组tuple 71](#_Toc491019008)

[15.1.4 Scala集合list和set 72](#_Toc491019009)

[15.1.5 Scala数组的转换操作 73](#_Toc491019010)

[15.1.6 Scala面向对象编程 73](#_Toc491019011)

[15.1.7 高阶函数 73](#_Toc491019012)

[15.1.8 隐式转换函数 74](#_Toc491019013)

[15.1.9 Scala中的模式匹配 75](#_Toc491019014)

[15.1.10 Scala中的trait 76](#_Toc491019015)

[15.1.11 Actor多线程编程 77](#_Toc491019016)

[16 Spark基础 79](#_Toc491019017)

[16.1 Spark安装 79](#_Toc491019018)

[16.2 Spark使用 79](#_Toc491019019)

[16.2.1 Spark实现wordcount 79](#_Toc491019020)

[16.2.2 Spark案例 79](#_Toc491019021)

[16.2.3 Spark Application运行的两种方式 84](#_Toc491019022)

[16.2.4 运行在yarn上 84](#_Toc491019023)

# Linux基础

## Linux分区操作

（1）磁盘分区

主分区 + 扩展分区(逻辑分区) <= 4 3+1 2+1 1+1

Linux系统默认所有设备文件都在/dev下面

/dev/sda --硬盘1 sda1第一个分区 sda2第二个分区 sda3第三个分区

/dev/sdb --硬盘2 sdb1第一个分区 sdb2第二个分区 sdb3第三个分区

/dev/sdc --硬盘3

# fdisk -l 查看系统所有硬盘的分区情况

（1）系统一共有几块硬盘，每个硬盘的容量大小

（2）每个硬盘的分区情况（硬盘空间是否还有剩余）

分区步骤：

(1)fdisk 设备名称（/dev/sdb）

(2)partx -a /dev/sdb

(3)格式化 # mkfs.ext4 /dev/sdb6

(4)挂载 # mount /dev/sdb6 /mnt （临时生效）

修改/etc/fstab （永久生效）

/dev/sdb6 /wwwroot ext4 defaults 0 0 （/wwwroot挂载点）

## Linux命令

### grep命令

案例：取出符合某条件并且不符合某条件的字符串

ifconfig | grep 'inet addr:' |grep -v '127.0.0.1'

（包含inet addr 并且不包含127.0.0.1 -v表示反选）

### Sed命令

用法：sed '匹配条件/执行的动作' /etc/passwd

或者

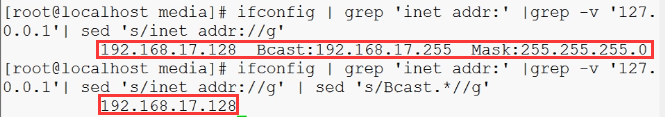
cat /etc/passwd | sed '匹配条件/执行的动作'

替换： sed 's/old/new/g'

s：替换 old：要替换的字符串 new：替换后的字符串 g:整行替换所有匹配的字符串

案例：取出符合某条件并且不符合某条件的字符串并进行替换

ifconfig | grep 'inet addr:' |grep -v '127.0.0.1'| sed 's/inet addr://g' | sed 's/Bcast.\*//g'



### awk命令

用法：awk -F: '{print $1}' /etc/passwd

-F指定分割符（这里为:） $0代表整行 $1代表第一列 $2第二列……. /etc/passwd操作的文件

用途：取出目标文件的低级列字符串

### uniq命令

用法：uniq –c

用途：计算单词出现次数（仅对相邻相同单词计算次数，可以先通过sort排序再进行计数）



### 查看某端口是否被占用

lsof -i:3306

或

netstat -anp|grep 80

### 移动文件

find /log/localhost\*.txt -size +0c -exec mv {} /home \;

解释：将/log目录下前缀为localhost并且大小大于0字节的txt文件移动到/home目录下

### 命令行修改替换文件

vi命令进入文件

按出 :

设置显示出行号： set nu

:1,3s/f/1/g（将第一到第三行的f替换为1）

:3,$s/f/1/g（将第三行到最后一行的f替换为1）

## Shell编程

### 变量

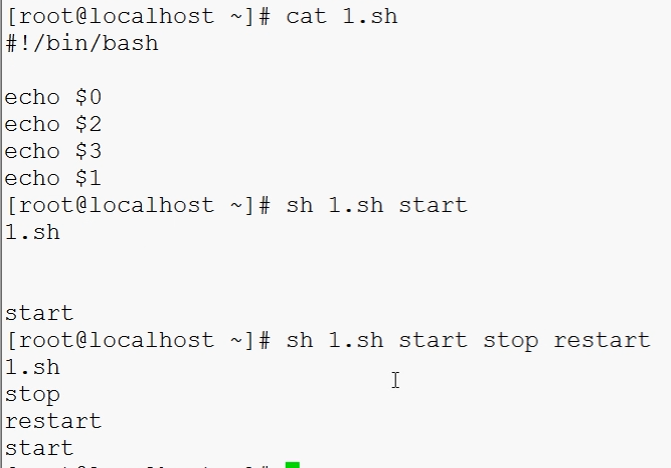
#### 位置变量

--通常和脚本联合使用

--$0 脚本名称

--$1、$2、$3、$4…$9 位置参数

案例：新建1.sh脚本，运行该脚本，并传入参数，$0对应脚本名称，$1 $2 ……分别顺序对应脚本传入的参数



#### 预定义变量

--$? 表示程序退出的代表(一般0代表执行成功，非0表示执行失败)

--$# 代表当前shell的参数个数

--$\* 代表所有参数（整体读取）

--$@ 代表所有参数（逐个读取）

--$$ 当前进程的PID

--$! 后台运行的最后一个进程的PID号

#### 自定义变量

--语法格式为：name=[value]

--注意：

变量等号两边不能有空格

变量对大小写敏感

--定义好后使用（$变量名）来调用变量的值

### Shell算数运算

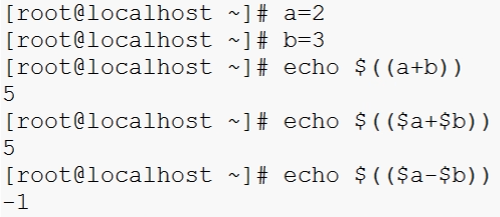
-- + - \* / %

--$((expression))

--$[expression]

--expr expression

（$((expression))与$[expression]效果一样）



（在表达式内变量前的$可加可不加）

a=2

b=3

# echo $(($a+$b))

# echo $[$a+$b]

# expr 5 + 2

# expr $a + $b

（expr运算符两边一定要有空格，并且引用变量是一定要有$符号）

### Shell内置测试判断

两种方式（效果一样）：

--test 测试表达式

--[ 测试表达式 ] 最常见的

--注意： 测试表达式与中括号直接一定要有空格

#### 数值比较

-eq 等于则为真

-ne 不等于则为真

-gt 大于则为真

-ge 大于等于则为真

-lt 小于则为真

-le 小于等于则为真

&& 逻辑与

-- cmd1 && cmd2 cmd1成功了才会执行cmd2

|| 逻辑或

-- cmd1 || cmd2 cmd1失败了才会执行cmd2

; 无逻辑关系

-- cmd1 ; cmd2 cmd1执行完后，执行cmd2

案例1：

[root@localhost ~]# echo $a $b

2 3

[root@localhost ~]# test $a -lt $b

[root@localhost ~]# echo $?

0

[root@localhost ~]# test $a -gt $b

[root@localhost ~]# echo $?

1:

案例2：

[root@localhost ~]# echo $a $b

2 3

[root@localhost ~]# [ $a -lt $b ]

[root@localhost ~]# echo $?

0

[root@localhost ~]# [ $a -gt $b ]

[root@localhost ~]# echo $?

1

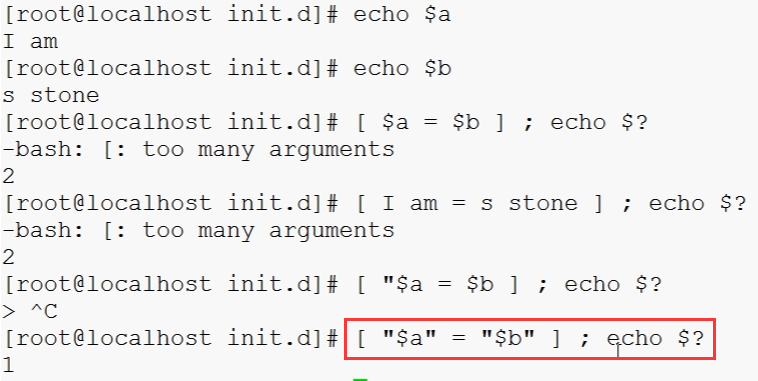
#### 字符串测试

= 等于则为真

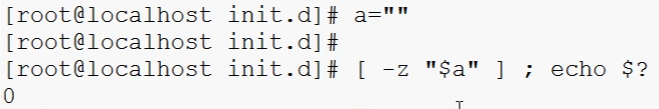
!= 不相等则为真

-z 字串 字串长度0则为真

-n 字串 字串长度不0则为真



（字符串比较时最好给变量加双引号）



#### 文件测试

-e 文件名 如果文件存在则为真

-d 文件名 如果文件存在且为目录则为真

-f 文件名 如果文件存在且为普通文件则为真

-r 文件名 如果文件存在且可读则为真

-w 文件名 如果文件存在且可写则为真

-x 文件名 如果文件存在且可执行则为真

-s 文件名 如果文件存在且至少有一个字符则为真

-b 文件名 如果文件存在且为块特殊文件则为真

-c 文件名 如果文件存在且为字符型特殊文件则为真

（文件名也可以为目录）

#### 其它

①Linux还提供了非（！）、或（-o）、与（-a）三个逻辑操作符，用于将测试条件连接起来，其优先顺序为：！最高，-a次之，-o最低。

②取字符串操作

${变量名:offset:length}

案例1：

[root@localhost ~]# a="201604091527"

[root@localhost ~]# echo $a

201604091527

[root@localhost ~]# echo ${a:2:3}

160

[root@localhost ~]# echo ${a:4:2}

04

案例2：

取字符串 echo ${变量名%.\*}

[root@localhost ~]# a="123.log"

[root@localhost ~]# echo ${a%.\*}

123

### 控制语句

#### for语句

语法1：

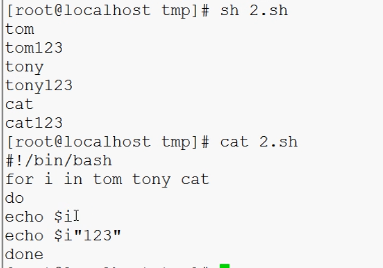
for 变量 in 值1 值2 … 值N

do

命令序列

Done

案例1：



案例2：在某文件夹下创建50个名字为dir1 dir2 ……..的文件

#!/bin/bash

for i in {1..50}

do

mkdir /usr/local/src/dir$i

done

（有同样效果：

#!/bin/bash

for i in `seq 50` #反引号代表命令预先执行

do

mkdir /usr/local/src/dir$i

done

）

语法2：

for ((初始化变量；结束循环条件；运算))

do

命令序列

Done

案例1：计算1+2+…+50的值

#!/bin/bash

for((i=1;i<=10;i++));do

SUM=$((SUM+i))

done

echo $SUM

#### while语句

语法1：

while [ 条件 ]

do

命令序列

Done

案例1：

#!/bin/bash

i=1

while [ $i -le 10 ]

do

SUM=$((SUM+i))

i=$[i+1]

done

echo $SUM

语法2：

while read -r line

do

命令序列

Done

案例1：读取并输出文件内容

#!/bin/bash

while read -r line

do

echo $line:HELLO

done < /etc/passwd

#< /etc/passwd为将该文件内容输出给while语句，只能结合read使用

案例2：读取并输出文件每行以：分隔的第一列的内容并在每行内容后面添加:HELLO

#!/bin/bash

while read -r line

do

echo `echo $line | awk -F: '{print $1}'`:HELLO

done < /etc/passwd

#### if与case语句

##### if语句

语法1：

if 条件

then

命令序列

fi

语法2：

if 条件

then

命令序列

else

命令序列

Fi

语法3：

if 条件

then

命令序列

elif 条件

then

命令序列

elif 条件

then

命令序列

else

命令序列

Fi

案例1：判断是否存在/tmp/123目录，存在则列出该目录的文件否则新建该目录

#!/bin/bash

if [ -d /tmp/123 ];then

ls /tmp/123

else

mkdir /tmp/123

fi

##### case语句

语法1：

case $变量名称 in

条件1）

命令序列

；；

条件2）

命令序列

；；

条件3）

命令序列

；；

\*）

Esac

语法2：

case $变量名称 in

条件1|条件2）

命令序列

；；

条件3|条件4）

命令序列

；；

条件5|条件6）

命令序列

；；

\*）

esac

案例1：

#!/bin/bash

case $1 in

top)

top # top命令用来显示执行中的程序进程

;;

free)

free # free命令用来显示内存的使用情况

;;

df)

df #查看磁盘盘剩余空间

;;

\*)

echo "usage:$0{top|free|df}" #前面都不匹配输出

esac

### 函数使用

语法1：

name() {

命令序列

}

语法2：

function name {

命令序列

}

案例1：声明sum加法函数并调用

#!/bin/bash

sum(){

echo $(($1+$2))

}

sum 5 6

### Crontab计划任务

#### 一次性计划任务

at --指定时间执行特定命令

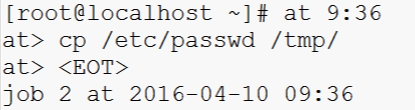
用法：at [时间]

案例1：

at 4:17 #指定在当天凌晨4点17分执行计划任务

at> cp /etc/passwd /tmp #计划任务内容

at> <EOT> #输入完成后，按ctrl+d结束



#### 周期性计划任务

crontab --周期性执行计划任务

用法：crontab [-u 用户] [-l|-r|-e]

参数：

-u：指定某个用户，不加-u选项则为当前用户

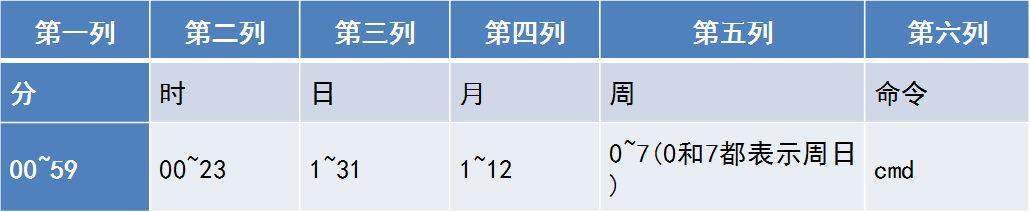
-e：制定计划任务

-l：列出计划任务

①确保crontab启动

service crond status #查看crontab状态

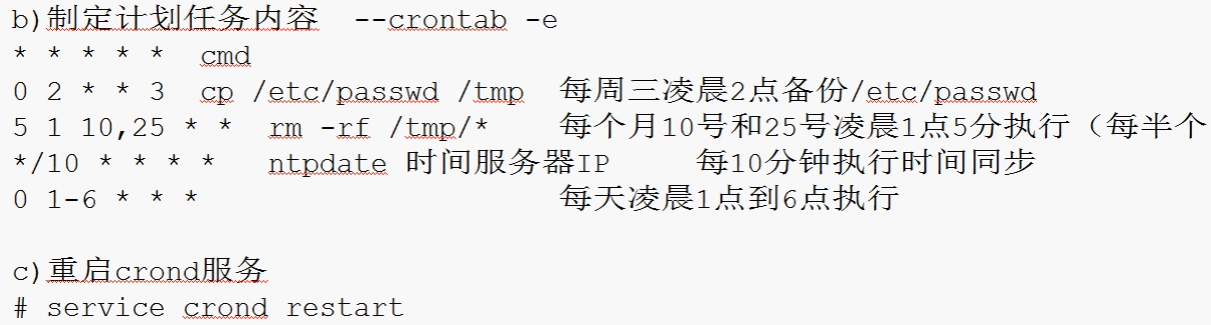
②任务格式



如上图，如果时间是时间段，可以使用横杠（-）来表示一段连续的时间，使用（，）表示若干不连续的时间，使用星号（\*）表示所有的时间，使用除号（/）表示间隔时间。

案例1：每十分钟执行一次时间同步

\*/10 \* \* \* \* ntpdate 时间服务器IP



添加新计划任务后要重启crontab服务。

# Windows系统执行hadoop程序

1. 解压hadoop压缩包
2. 下载winutils.exe将其放到bin目录
3. 配置环境变量

HADOOP\_HOME

D:\Hadoop\hadoop-2.5.0

Path

%HADOOP\_HOME%\bin

1. 重启系统

# Maven基础

## 环境搭建

1. 添加系统变量

我的电脑（计算机）----右击--->属性--->高级系统设置--->环境变量---->系统变量下新建的M2\_HOME=E:\softwares\apache-maven-3.3.9

---->添加M2\_HOME变量到path 路径中

1. 验证

验证环境变量：mvn -v 或者 mvn –version

## pom文件

### 配置

<!-- 指定当前pom 的版本-->

<modelVersion>4.0.0</modelVersion>

<!-- 项目包名: 公司地址名称反写+项目名称-->

<groupId>com.ibeifeng.maven</groupId>

<!--项目模块名称：一般为 项目名-模块名 -->

<artifactId>maven-demo1</artifactId>

<!--

标识当前项目版本号

第一个.号之前的数字标识 表示大版本号

第二个.号之前的数字标识 表示分支版本号

第三个.号之前的数字标识 表示小版本号

SNAPSHOT：快照版本

Release 发布版本

Alpha :内部测试版本

GA：正式发布的版本

-->

<version>1.0.0SNAPSHOT</version>

## maven 常用命令

1. mvn -v 或者 mvn -version：

验证环境变量

2. mvn help:system ：

打印出所有的系统属性和环境变量

3. mvn compile：

编译项目源代码（不会编译test 目录的元代）（会产生target 文件）

4. mvn test: 运行应用程序中的单元测试

5. mvn test-compile 编译测试代码，compile 之后生成的targer 文件夹 主程序编译在classes 文件夹下面，测试程序代码放在test-classes 文件夹下

6. mvn clean 删除target 文件夹

## Maven配置镜像仓库

镜像仓库：为国内的服务器，下载速度更快

conf/settings.xml

--mirrors

--mirror:

<mirror>

<id>repo2</id>

<mirrorOf>central</mirrorOf>

<name>Human Readable Name for this Mirror.</name>

<url>http://repo2.maven.org/maven2/</url>

</mirror>

NOTE:

maven setttings.xml 不要去修改maven 安装目录下的conf/settings.xml （全局），推荐大家 拷贝settings.xml 到你对应的本地仓库目录下面(C:\Users\daibin\.m2/settings.xml)

## dos命令生成Maven骨架

简单使用：

mvn archetype:generate

817 回车 提供的骨架 maven-archetype-quickstart

一步到位：

mvn archetype:generate -DgroupId=com.ibeifeng.maven04 -DartifactId=maven-demo04 -Dversion=1.0-SNAPSHOT -Dpackage=com.ibeifeng.maven04

-DgroupId 公司名称反写+项目名称

-DartifactId 项目名称-模块名称

-Dversion 项目版本号

-Dpackage 源码包目录

NOTE:DgroupId 和 Dpackage 是相同的，官方推荐的，看起来更加清晰

maven的Responsitory 里面支持的archetype 大概有1600+，

其实常用的archetypee 就那么几个

1.quick start

2.webapp

3.simple

很自然的就会去想到 能不能用什么简便的方式只从上面这个list 里选择就可以

怎么去实现？

解决思路：

1. mvn archetype:crawl

:会在本地仓库目录(C:\Users\daibin\.m2\repository)下生成archetype-catalog.xml

2. 将生成archetype-catalog.xml 上移到C:\Users\daibin\.m2 目录下面

3.mvn archetype:generate -DarchetypeCatalog=local

## maven 依赖传递

[maven 依赖传递](参考文件/maven%20依赖传递.docx)

## maven生命周期

[maven生命周期](参考文件/maven生命周期.docx)

## maven 常见错误

[maven 常见错误](参考文件/maven%20常见错误.docx)

# Hadoop基础

## Hadoop安装

[Hadoop伪分布式安装教程](参考文件/Hadoop伪分布式安装教程.docx)

[Hadoop分布式安装教程](参考文件/Hadoop分布式安装教程.docx)

## MapReduce深入理解

[MapReduce过程图解](参考文件/MapReduce过程.png)

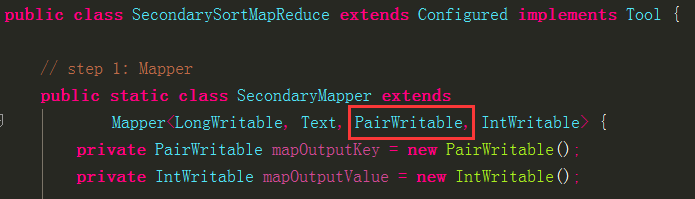
分区时不同的key可能分到不同的分区，分区后进行排序再分组，而分组时相同的key一定会分到相同的组，并且会形成一个keyvalues对，values为该key的所有value的集合。

进入reduce阶段时，若ReduceTasks个数设置为一个，即setup函数和cleanup函数只会被调用一次，调用reduce函数时传入的数据为同一个分组的数据，有多少个分组reduce函数就会被调用多少次。若ReduceTasks个数设置为多个，即各个分组会被分到不同的ReduceTasks，每一个ReduceTasks启动时都会调用一次setup函数，结束时调用一次cleanup函数，若分到该ReduceTask的分组为n个即reduce函数会被调用n次。

## MapReduce二次排序

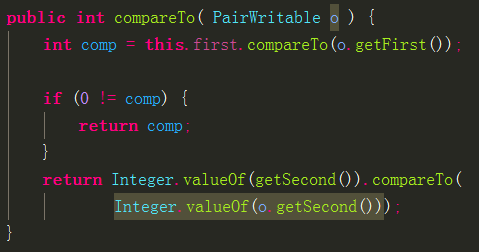
### 组合key

[自定义数据类型作为组合key](WorkSpace/bigdata-hdfs/src/main/java/com/ibeifeng/mapreduce/SecondarySortMapReduce.java)

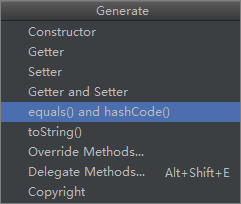


[自定义数据类型生成步骤：](WorkSpace/bigdata-hdfs/src/main/java/com/ibeifeng/mapreduce/PairWritable.java)

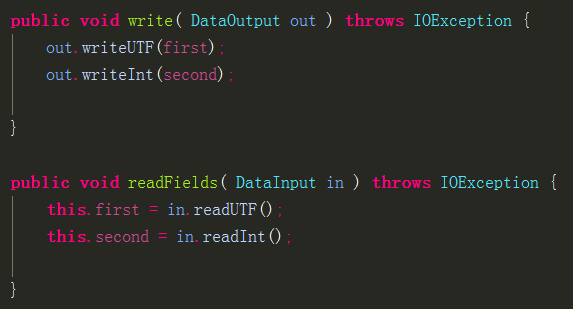
1. 实现WritableComparable接口，添加未实现方法；
2. 添加类成员变量并生成get和set方法；
3. 生成无参和有参构造方法；
4. 实现compareTo方法，用来定义排序规则；



1. 生成hashCode和equals方法；



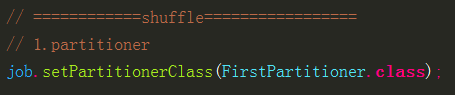
1. 生成tostring方法；
2. 实现写和度方法（两个方法参数顺序要一致）



### 分区规则

[保证原来的分区，需要自定义分区规则](WorkSpace/bigdata-hdfs/src/main/java/com/ibeifeng/mapreduce/FirstPartitioner.java)

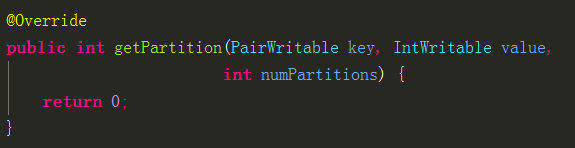
1. Job设置分区类



1. 新建分区类，继承Partitioner类，泛型中的类为Map输出的类型



1. 生成getPartition方法



1. 实现getPartition方法

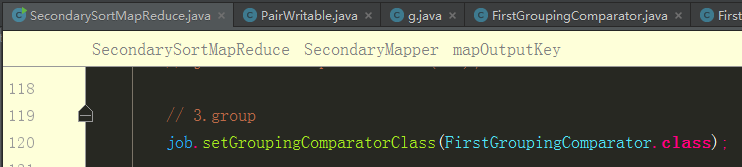
hashPartitioner分区是MR框架默认的分区计算方法



### 分组规则

[保证原来的分组，需要自定义分组规则。](WorkSpace/bigdata-hdfs/src/main/java/com/ibeifeng/mapreduce/FirstGroupingComparator.java)

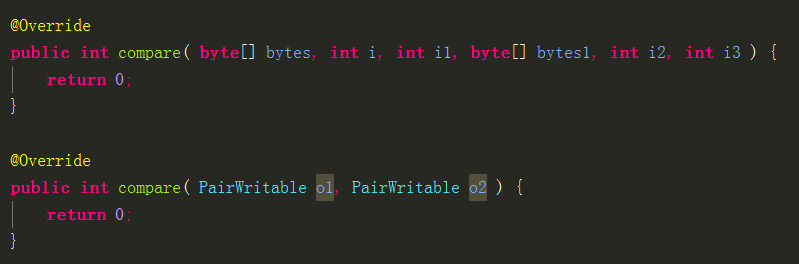
1. Job设置分组类



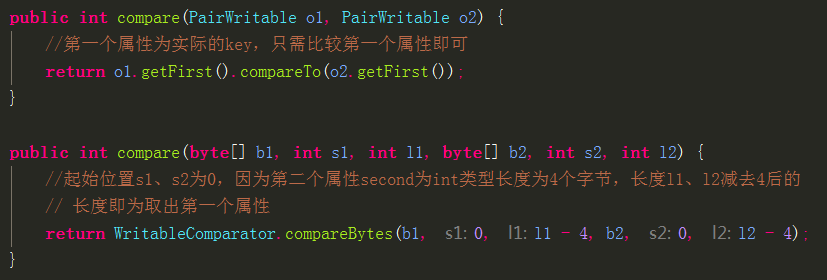
1. 新建分组类，实现RawComparator接口，泛型中的类为要分组排序的数据类



1. 生成compare方法



1. 实现compare方法



## MapReduce Join

[示例代码](WorkSpace/bigdata-hdfs/src/main/java/com/ibeifeng/mapreduce/join)

1、举例，网上购物（账户，订单）

账户信息（customer）----》小表

用户ID、名称、收货地址、电话联系方式

cid cname address phone

订单信息（order）-----》大表

订单ID、用户ID、价格、名称

oid cid price pname

2、组合，长信息

cid cname address phone oid cid price pname

3、设计<key value>，两张表中的cid作为一个公共的连接点，map输出的key

stepup()准备工作--》在map没有处理之前，先将小表customer的数据读到，存储，加到内存中

map()-----》读取大表order

cleanup()清理工作

将cid作为一个key,<cid,customerinfo>

<cid,orderinfo>

4、map join适合数据量较小的

5、reduce join

Reduce input key value

<cid,list(cInfo,orderInfo,orderInfo,orderInfo,orderInfo....)>

map

customer

<cid,cInfo>

cInfo

tag：customer , order

data: cInfo， ordrInfo

order

<cid,orderInfo>

设置一个标识：判断是用户还是订单

reduce join 可以叫 shuffle join

NOTE：最终reduce函数接受到的数据都是经过shuffle阶段后把相同key放在一起的数据，两表之间的join正是利用了shuffle阶段将相同key发送给同一个reduce接收的特点成功实现join。

## HA机制

### 需求背景

1、元数据同步

-》保证两个namenode内存中存储的文件系统元数据是一致的

2、思路，namenode启动过程

-》读取fsimage和edits文件-》读取后会生成新的fsimage和edits文件

-》另一个namenode同样需要去读取这两个文件，变化后的edits日志文件，同样需要读取

-》注册心跳、块的报告，需要向另一个namenode实时的汇报

3、日志文件安全性

-》cloudera公司提出：分布式日志存储方案

-》找到一个datanode节点目录，zookeeper有一个2n+1概念 n+1

-》写多份再读取，节点数目必须是奇数

-》还可以存储在zookeeper

4、通过代理来让客户端判断现在对外提供服务的是哪台namenode

5、两个namenode，但是必须在任何的情况下，只能有一个namenode对外提供服务

-》隔离

6、JournalNode日志节点

7、secondaryNameNode在HA架构下就不需要了

### HA配置

[HA机制配置](参考文件/HA机制配置.docx)

## Resourcemanager HA

[Resourcemanager HA配置](参考文件/Resourcemanager%20HA配置.docx)

## HDFS Federation

[HDFS Federation配置](参考文件/HDFS%20Federation配置.docx)

## 分布式拷贝工具DistCP

分布式拷贝

作用：用于大的内部或者外部的集群之间的数据拷贝

底层是：MapReduce，但是只有map没有reduce

用法：

distcp <srcurl>源端 <desturl>目标端

案例：

将第二台机器的文件拷贝到第一台的HDFS目录中去

$ hadoop distcp hdfs://MyDream2:8020/ wc.input hdfs://MyDream1:8020/ conf

# Zookeeper基础

## Zookeeper安装

[Zookeeper单机安装](参考文件/Zookeeper单机安装.docx)

[Zookeeper分布式安装](参考文件/Zookeeper分布式安装.docx)

# Hive基础

## MySQL安装

[MySQL安装](参考文件/MySQL安装.docx)

## Hive安装

[Hive安装](参考文件/Hive安装.docx)

## Hive使用

建表

create table student(id int, name string) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

显示表信息

desc extended student ;

desc formatted student ;

加载数据

load data local inpath '/opt/datas/student.txt'into table db\_hive.student ;

显示函数

show functions ;

查看函数用法/查看函数用法并举例

desc function upper ;

desc function extended upper ;

查看变量

set

### 分区表

建表

create EXTERNAL table IF NOT EXISTS default.emp\_partition(

empno int,

ename string,

job string,

mgr int,

hiredate string,

sal double,

comm double,

deptno int

)

partitioned by (month string,day string)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' ;

加载数据

load data local inpath '/opt/datas/emp.txt' into table default.emp\_partition partition (month='201509',day='13') ;

查询

select \* from emp\_partition where month = '201509' and day = '13' ;

修复分区

msck repair table tablename;

### 加载数据

load data [local] inpath 'filepath' [overwrite] into table tablename [partition (partcol1=val1,...)];

\* 原始文件存储的位置

\* 本地 local

\* hdfs

\* 对表的数据是否覆盖

\* 覆盖 overwrite

\* 追加

\* 分区表加载，特殊性

partition (partcol1=val1,...) （以及分区，二级分区，三级分区………）

1）加载本地文件到hive表

load data local inpath '/opt/datas/emp.txt' into table default.emp ;

2）加载hdfs文件到hive中

load data inpath '/user/beifeng/hive/datas/emp.txt' overwrite into table default.emp ;

3）加载数据覆盖表中已有的数据

load data inpath '/user/beifeng/hive/datas/emp.txt' into table default.emp ;

4）创建表是通过insert加载

create table default.emp\_ci like emp ;

insert into table default.emp\_ci select \* from default.emp ;

5）创建表的时候通过location指定加载

### 导出数据

1. 导出到本地

insert overwrite local directory '/opt/datas/hive\_exp\_emp2'

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' COLLECTION ITEMS TERMINATED BY '\n'

select \* from default.emp ;

hive -e "select \* from default.emp ;" > /opt/datas/exp\_res.txt

1. 导出到hdfs

insert overwrite directory '/user/beifeng/hive/hive\_exp\_emp'

select \* from default.emp ;

### having关键词

\* where 是针对单条记录进行筛选

\* having 是针对分组结果进行筛选

求每个部门的平均薪水大于2000的部门

select deptno, avg(sal) avg\_sal from emp group by deptno having avg\_sal > 2000;

### 数据导入导出

#### Export

导出，将Hive表中的数据，导出到外部

#### Import

导入，将外部数据导入Hive表中

EXPORT TABLE default.emp TO '/user/beifeng/hive/export/emp\_exp' ;

export\_target\_path：

指的是HDFS上路径

import table db\_hive.emp from '/user/beifeng/hive/export/emp\_exp';

### 四个by关键词

#### order by

对全局数据的一个排序，仅仅只有一个reduce

select \* from emp order by empno desc ;

#### sort by

对每一个reduce内部数据进行排序的，全局结果集来说不是排序

set mapreduce.job.reduces= 3;

select \* from emp sort by empno asc ;

insert overwrite local directory '/opt/datas/sortby-res' select \* from emp sort by empno asc ;

#### distribute by

分区partition

类似于MapReduce中分区partition,对数据进行分区，结合sort by进行使用

insert overwrite local directory '/opt/datas/distby-res' select \* from emp distribute by deptno sort by empno asc ;

注意事项：

distribute by 必须要在sort by 前面。

#### cluster by

当distribute by和sort by 字段相同时，可以使用cluster by ;

insert overwrite local directory '/opt/datas/cluster-res' select \* from emp cluster by empno ;

不同时可

select mid, money, name from store cluster by mid sort by money

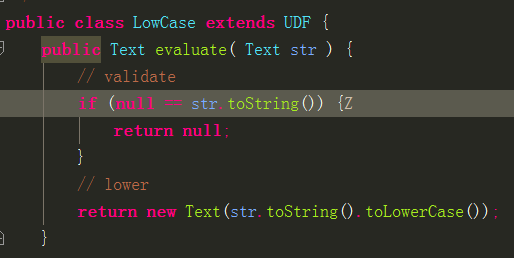
 注意被cluster by指定的列只能是降序，不能指定asc和desc。

### UDF函数

#### Creating Custom UDFs

[示例代码](WorkSpace/Hive-Projects/src/main/java/LowCase.java)

1)First, you need to create a new class that extends UDF, with one or more methods named evaluate.



2)Usage

Jar包在本地

add jar /opt/datas/hiveudf.jar ;

create temporary function my\_lower as "com.beifeng.senior.hive.udf.LowerUDF" ;

select ename, my\_lower(ename) lowername from emp limit 5 ;

jar包需上传

CREATE FUNCTION self\_lower AS 'com.beifeng.senior.hive.udf.LowerUDF' USING JAR 'hdfs://MyDream:8020/user/beifeng/hive/jars/hiveudf.jar';

select ename, self\_lower(ename) lowername from emp limit 5 ;

### Hive的三种启动方式

1， hive 命令行模式

进入hive安装目录，输入bin/hive的执行程序，或者输入 hive –service cli

用于Linux平台命令行查询，查询语句基本跟MySQL查询语句类似

2， hive web界面的启动方式

bin/hive –service hwi （& 表示后台运行）

用于通过浏览器来访问hive，浏览器访问地址是：127.0.0.1:9999/hwi

3， hive 远程服务 (端口号10000) 启动方式

bin/hive –service hiveserver2 &（&表示后台运行）

用java，Python等程序实现通过jdbc等驱动的访问hive就用这种启动方式了。

### 数据存储

\* 按行存储数据

\* 按列存储数据

create table page\_views(

track\_time string,

url string,

session\_id string,

referer string,

ip string,

end\_user\_id string,

city\_id string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'

STORED AS TEXTFILE ;

load data local inpath '/opt/datas/page\_views.data' into table page\_views ;

dfs -du -h /user/hive/warehouse/page\_views/ ;

18.1 M /user/hive/warehouse/page\_views/page\_views.data

>>>>>>>orc

create table page\_views\_orc(

track\_time string,

url string,

session\_id string,

referer string,

ip string,

end\_user\_id string,

city\_id string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'

STORED AS orc ;

insert into table page\_views\_orc select \* from page\_views ;

dfs -du -h /user/hive/warehouse/page\_views\_orc/ ;

2.6 M /user/hive/warehouse/page\_views\_orc/000000\_0

>>>>>>>> parquet

create table page\_views\_parquet(

track\_time string,

url string,

session\_id string,

referer string,

ip string,

end\_user\_id string,

city\_id string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'

STORED AS PARQUET ;

insert into table page\_views\_parquet select \* from page\_views ;

dfs -du -h /user/hive/warehouse/page\_views\_parquet/ ;

13.1 M /user/hive/warehouse/page\_views\_parquet/000000\_0

select session\_id,count(\*) cnt from page\_views group by session\_id order by cnt desc limit 30 ;

select session\_id,count(\*) cnt from page\_views\_orc group by session\_id order by cnt desc limit 30 ;

-------

select session\_id from page\_views limit 30 ;

select session\_id from page\_views\_orc limit 30 ;

select session\_id from page\_views\_parquet limit 30 ;

========================================================

create table page\_views\_orc\_snappy(

track\_time string,

url string,

session\_id string,

referer string,

ip string,

end\_user\_id string,

city\_id string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'

STORED AS orc tblproperties ("orc.compress"="SNAPPY");

insert into table page\_views\_orc\_snappy select \* from page\_views ;

dfs -du -h /user/hive/warehouse/page\_views\_orc\_snappy/ ;

--------------

create table page\_views\_orc\_none(

track\_time string,

url string,

session\_id string,

referer string,

ip string,

end\_user\_id string,

city\_id string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'

STORED AS orc tblproperties ("orc.compress"="NONE");

insert into table page\_views\_orc\_none select \* from page\_views ;

dfs -du -h /user/hive/warehouse/page\_views\_orc\_none/ ;

--------------------

set parquet.compression=SNAPPY ;

create table page\_views\_parquet\_snappy(

track\_time string,

url string,

session\_id string,

referer string,

ip string,

end\_user\_id string,

city\_id string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'

STORED AS parquet;

insert into table page\_views\_parquet\_snappy select \* from page\_views ;

dfs -du -h /user/hive/warehouse/page\_views\_parquet\_snappy/ ;

总结：

在实际的项目开发当中，hive表的数据

\* 存储格式

orcfile / qarquet

\* 数据压缩

Snappy

### SQL执行Python脚本

建表

CREATE TABLE u\_data\_new (

userid INT,

movieid INT,

rating INT,

weekday INT)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

添加脚本进入到hive

add FILE /opt/datas/ml-100k/weekday\_mapper.py;

执行脚本

INSERT OVERWRITE TABLE u\_data\_new

SELECT

TRANSFORM (userid, movieid, rating, unixtime) -- input from source table

USING 'python weekday\_mapper.py' -- script

AS (userid, movieid, rating, weekday) --output from python

FROM u\_data;

INSERT OVERWRITE TABLE u\_data\_new

SELECT

TRANSFORM (userid, movieid, rating, unixtime)

USING 'python weekday\_mapper.py'

AS (userid, movieid, rating, weekday)

FROM u\_data;

SELECT weekday, COUNT(\*) FROM u\_data\_new GROUP BY weekday;

SELECT weekday, COUNT(1) cnt FROM u\_data\_new GROUP BY weekday order by cnt desc;

脚本内容

import sys

import datetime

for line in sys.stdin:

line = line.strip()

userid, movieid, rating, unixtime = line.split('\t')

weekday = datetime.datetime.fromtimestamp(float(unixtime)).isoweekday()

print '\t'.join([userid, movieid, rating, str(weekday)])

# sqoop基础

## sqoop安装

[sqoop安装](参考文件/sqoop安装.docx)

## sqoop使用

### 查看数据库

sqoop list-databases \

--connect jdbc:mysql://MyDream:3306 \

--username root \

--password 123456

### 导入数据

--添加测试mysql表

CREATE TABLE `my\_user` (

`id` tinyint(4) NOT NULL AUTO\_INCREMENT,

`account` varchar(255) DEFAULT NULL,

`passwd` varchar(255) DEFAULT NULL,

PRIMARY KEY (`id`)

);

INSERT INTO `my\_user` VALUES ('1', 'admin', 'admin');

INSERT INTO `my\_user` VALUES ('2', 'pu', '12345');

INSERT INTO `my\_user` VALUES ('3', 'system', 'system');

INSERT INTO `my\_user` VALUES ('4', 'zxh', 'zxh');

INSERT INTO `my\_user` VALUES ('5', 'test', 'test');

INSERT INTO `my\_user` VALUES ('6', 'pudong', 'pudong');

INSERT INTO `my\_user` VALUES ('7', 'qiqi', 'qiqi');

#### 导入数据

sqoop import \

--connect jdbc:mysql://MyDream:3306/test \

--username root \

--password 123456 \

--table my\_user

#### 导入数据并设置导入目录和map个数

sqoop import \

--connect jdbc:mysql://MyDream:3306/test \

--username root \

--password 123456 \

--table my\_user \

--target-dir /user/beifeng/sqoop/imp\_my\_user \

--num-mappers 1

#### 导入数据并制定存储格式

sqoop import \

--connect jdbc:mysql://MyDream:3306/test \

--username root \

--password 123456 \

--table my\_user \

--target-dir /user/beifeng/sqoop/imp\_my\_user\_parquet \

--fields-terminated-by ',' \

--num-mappers 1 \

--as-parquetfile

#### 导入指定列的数据

sqoop import \

--connect jdbc:mysql://MyDream:3306/test \

--username root \

--password 123456 \

--table my\_user \

--target-dir /user/beifeng/sqoop/imp\_my\_user\_column \

--num-mappers 1 \

--columns id,account

#### 导入sql查询后的数据

\* 在实际的项目中，要处理的数据，需要进行初步清洗和过滤

\* 某些字段过滤

\* 条件

\* join

sqoop import \

--connect jdbc:mysql://MyDream:3306/test \

--username root \

--password 123456 \

--query 'select id, account from my\_user' \

--target-dir /user/beifeng/sqoop/imp\_my\_user\_query \

--num-mappers 1

>>>>>>>>>>>>>>出错了？？？？

15/09/01 07:10:39 ERROR tool.ImportTool: Encountered IOException running import job: java.io.IOException: Query [select id, account from my\_user] must contain '$CONDITIONS' in WHERE clause.

at org.apache.sqoop.manager.ConnManager.getColumnTypes(ConnManager.java:300)

at org.apache.sqoop.orm.ClassWriter.getColumnTypes(ClassWriter.java:1833)

at org.apache.sqoop.orm.ClassWriter.generate(ClassWriter.java:1645)

at org.apache.sqoop.tool.CodeGenTool.generateORM(CodeGenTool.java:96)

必须加上where $CONDITIONS

sqoop import \

--connect jdbc:mysql://MyDream:3306/test \

--username root \

--password 123456 \

--query 'select id, account from my\_user where $CONDITIONS' \

--target-dir /user/beifeng/sqoop/imp\_my\_user\_query \

--num-mappers 1

#### 导入数据并压缩和删除已存在目录和指定字段分隔符

sqoop import \

--connect jdbc:mysql://MyDream:3306/test \

--username root \

--password 123456 \

--table my\_user \

--target-dir /user/beifeng/sqoop/imp\_my\_sannpy \

--delete-target-dir \

--num-mappers 1 \

--compress \

--compression-codec org.apache.hadoop.io.compress.SnappyCodec \

--fields-terminated-by '\t'

#### 增量数据的导入

有一个唯一标识符，通常这个表都有一个字段，类似于插入时间createtime

\* query //通过sql查询导入

where createtime => 20150924000000000 and createtime < 20150925000000000

\* sqoop //通过sqoop指定字段并指定该字段上次导入的最后一个记录

Incremental import arguments:

--check-column <column> Source column to check for incremental

change

--incremental <import-type> Define an incremental import of type

'append' or 'lastmodified'

--last-value <value> Last imported value in the incremental

check column

导入id=4以后的记录

sqoop import \

--connect jdbc:mysql://MyDream:3306/test \

--username root \

--password 123456 \

--table my\_user \

--target-dir /user/beifeng/sqoop/imp\_my\_incr \

--num-mappers 1 \

--incremental append \

--check-column id \

--last-value 4

#### 直接导入，不走MapReduce

sqoop import \

--connect jdbc:mysql://MyDream:3306/test \

--username root \

--password 123456 \

--table my\_user \

--target-dir /user/beifeng/sqoop/imp\_my\_incr \

--num-mappers 1 \

--delete-target-dir \

--direct

#### 导入数据到hive表

Hive数据存储在hdfs上

use default ;

drop table if exists user\_hive ;

create table user\_hive(

id int,

account string,

password string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' ;

sqoop import \

--connect jdbc:mysql://MyDream:3306/test \

--username root \

--password 123456 \

--table my\_user \

--fields-terminated-by '\t' \

--delete-target-dir \

--num-mappers 1 \

--hive-import \

--hive-database default \

--hive-table user\_hive

### 导出数据

#### 导出数据到MySQL表

touch /opt/datas/user.txt

vi /opt/datas/user.txt

12,beifeng,beifeng

13xuanyun,xuanyu

bin/hdfs dfs -mkdir -p /user/beifeng/sqoop/exp/user/

bin/hdfs dfs -put /opt/datas/user.txt /user/beifeng/sqoop/exp/user/

sqoop export \

--connect jdbc:mysql://MyDream:3306/test \

--username root \

--password 123456 \

--table my\_user \

--export-dir /user/beifeng/sqoop/exp/user/ \

--num-mappers 1

--input-fields-terminated-by '\t'

--input-fields-terminated-by '\t'

### Shell导入导出数据

shell scripts

## step 1

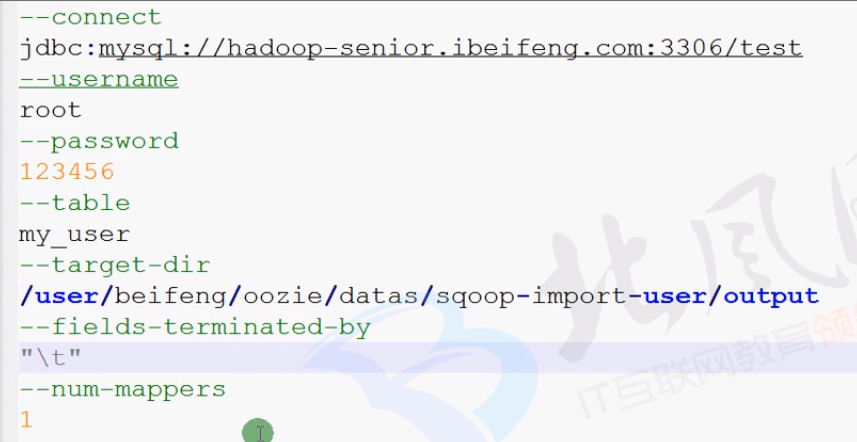
load data ...

## step 2

bin/hive -f xxxx

## step 3

编写脚本，将之前的命令写进脚本



可以不写某些参数，执行脚本的时候再追加：

sqoop --options-file /opt/datas/sqoop-import-hdfs.txt --num-mappers 1

# flume基础

## flume安装

[flume安装](参考文件/flume安装.docx)

## flume使用

上传[Telnet工具](参考文件/telnet-rpms)到虚拟机

### 安装rpm

mv netcat-1.10-891.2.x86\_64.rpm ../

rpm -ivh ./\*.rpm

### 启动xinetd服务

/etc/rc.d/init.d/xinetd restart

### 启动flume服务

Flume命令

[root@MyDream ~]# flume-ng

Usage: /home/hadoop/apache-flume-1.5.0-cdh5.3.6-bin/bin/flume-ng <command> [options]...

commands:

agent run a Flume agent

global options:

--conf,-c <conf> use configs in <conf> directory

-Dproperty=value sets a Java system property value

agent options:

--name,-n <name> the name of this agent (required)

--conf-file,-f <file> specify a config file (required if -z missing)

上传[配置文件](参考文件/flume脚本/a1.conf)到flume安装目录下的conf目录

启动服务：

flume-ng agent --conf conf --name agent-test --conf-file test.conf

或者

flume-ng agent -c conf -n agent-test -f test.conf

解释：

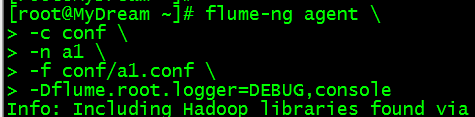
flume-ng agent \

-c conf \ 启用flume安装目录下的conf文件夹

-n a1 \ agent的名字为a1

-f conf/a1.conf \ 指定配置文件为conf目录下的a1.conf

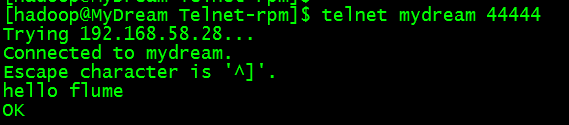
-Dflume.root.logger=DEBUG,console 显示信息到控制台



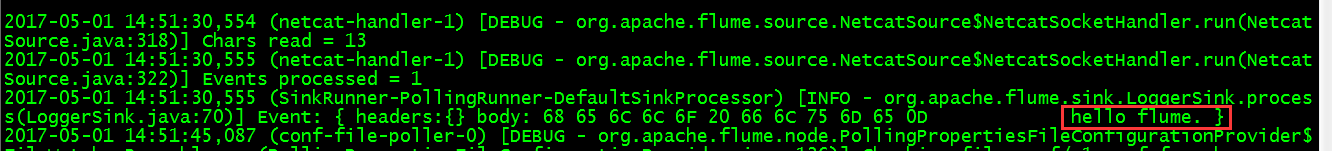
### 连接a1.conf配置文件指定的端口

telnet mydream 44444

发送信息



Flume收到信息



### 收集hive日志

编辑[配置文件](参考文件/flume脚本/flume-tail.conf)

新建上传日志的hdfs目录

hdfs dfs -mkdir -p /user/hadoop/flume/hive-log

启动脚本：

flume-ng agent \

-c conf \

-n a2 \

-f conf/flume-tail.conf \

-Dflume.root.logger=DEBUG,console

同步日志到hdfs成功



NOTE：若配置文件里面配置的是HA访问hdfs，需将hadoop的core-site.xml和hdfs-site.xml两个配置文件放到flume的conf文件下下面去

### 动态创建hdfs文件夹

按日期创建文件夹

hdfs://hadoop-senior.ibeifeng.com:8020/user/beifeng/flume/applogs/%Y%m%d

需设置使用服务器时间戳

hdfs.useLocalTimeStamp = true

编辑[配置文件](参考文件/flume脚本/flume-app.conf)

新建上传日志的hdfs目录

hdfs dfs -mkdir -p /user/hadoop/flume/sp-log

启动脚本：

flume-ng agent \

-c conf \

-n a2 \

-f conf/flume-app.conf \

-Dflume.root.logger=DEBUG,console

同步日志到hdfs成功

### 创建多个agent

[示例代码](WorkSpace/Hadoop-Projects/other/flume/app1/app1.cf)

agent1为创建监听端口的agent

测试方法：telnet 0 6666

agent2为创建监听系统日志agent

测试方法：logger “内容”

agent3为创建agent接受agent1和agent2输出并将两个结果输出到hdfs（输出hdfs文件名按自定义agent命名）

# Oozie基础

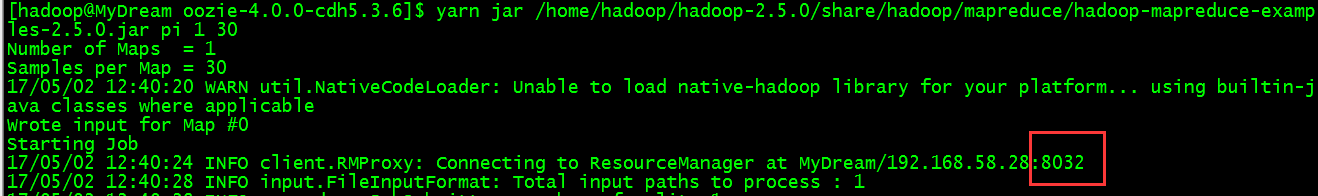
## Oozie安装

[Oozie安装](参考文件/Oozie安装.docx)

## 运行job

测试jobtrack的端口

yarn jar /home/hadoop/hadoop-2.5.0/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.5.0.jar pi 1 30



修改properties文件

修改/home/hadoop/oozie-4.0.0-cdh5.3.6/examples/apps/map-reduce/job.properties

nameNode=hdfs://MyDream:8020

jobTracker=MyDream:8032

运行job

oozie job -oozie http://localhost:11000/oozie -config examples/apps/map-reduce/job.properties –run

## 编写MapReduce action

新建文件夹

mkdir myMapReduce

复制example到文件夹下

cp -r /home/hadoop/oozie-4.0.0-cdh5.3.6/examples/apps/map-reduce/\* ./

编辑workflow.xml

（本配置过程相当于在编写MapReduce程序时的job设置）

替换为新api的参数

mapred.job.queue.name替换为mapreduce.job.queuename

mapred.mapper.class替换为mapreduce.job.map.class

mapred.reducer.class替换为mapreduce.job.reduce.class

mapred.input.dir替换为mapreduce.input.fileinputformat.inputdir

mapred.output.dir替换为mapreduce.output.fileoutputformat.outputdir

删除

<property>

<name>mapred.map.tasks</name>

<value>1</value>

</property>

添加

<property>

<name>mapreduce.map.output.key.class</name>

<value>org.apache.hadoop.io.Text</value>

</property>

<property>

<name>mapreduce.map.output.value.class</name>

<value>org.apache.hadoop.io.IntWritable</value>

</property>

<property>

<name>mapreduce.job.output.key.class</name>

<value>org.apache.hadoop.io.Text</value>

</property>

<property>

<name>mapreduce.job.output.value.class</name>

<value>org.apache.hadoop.io.IntWritable</value>

</property>

开头添加，开启新api

<property>

<name>mapred.maper.new-api</name>

<value>true</value>

</property>

<property>

<name>mapred.reducer.new-api</name>

<value>true</value>

</property>

自定义名字（第2 3个名字不能超过20个字符）

<workflow-app xmlns="uri:oozie:workflow:0.2" name="myWordcount">

<start to="wc-node"/>

<action name="wc-node">

Job.properties自定义应用路径

examplesRoot=examples

oozie.wf.application.path=${nameNode}/user/${user.name}/${examplesRoot}/apps/map-reduce/workflow.xml

可改为：

OozieAppPath=user/hadoop/wordcount/app 定义变量

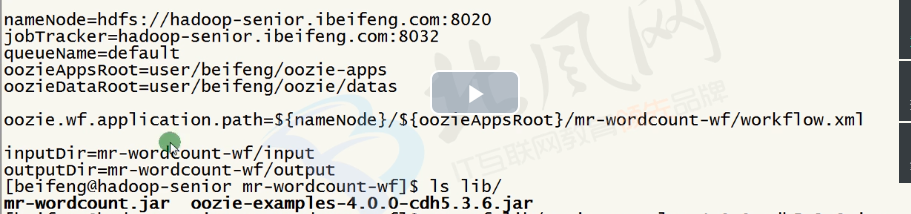
OozieDatapath= user/hadoop/wordcount/data 定义变量

oozie.wf.application.path=${nameNode}/${OozieAppPath}/myWordcount/workflow.xml

指定配置文件路径，需上传myWordcount目录下的文件到hdfs的该路径下

拷贝依赖的jar包（要运行的编写的MapReducejar包）到myMapReduce的lib目录下

更改workflow.xml的maper类和reducer类型自己编写的maper类 reducer类

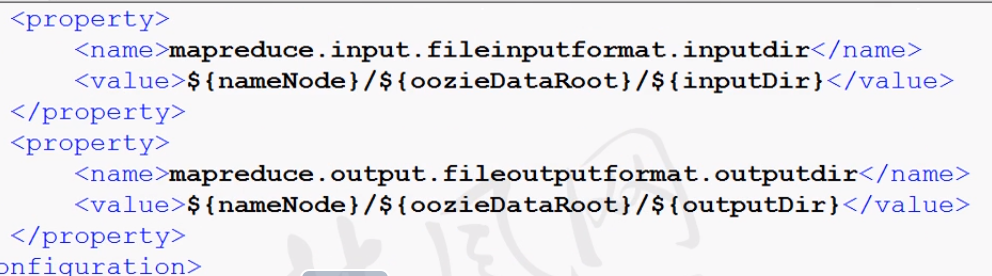


将自定义的MapReduce输出结果的目录配置进

<prepare>

<delete path="${nameNode}/user/${wf:user()}/${examplesRoot}/output-data/${outputDir}"/>

</prepare>



运行

oozie job -oozie http://localhost:11000/oozie -config wordcount/app /myWordcount /job.properties –run



## 编写hive action

参考/home/hadoop/oozie-4.0.0-cdh5.3.6/examples/apps/hive

其中hive新旧api都接受，所以不用更改api参数

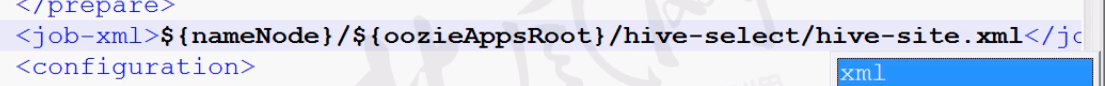
将hive的hive-site.xml文件放到复制/home/hadoop/oozie-4.0.0-cdh5.3.6/examples/apps/hive出来后的目录下，和xml properties 脚本放在同一个文件夹，并在该文件夹新建lib目录，将MySQL jdbc jar包放到该目录。

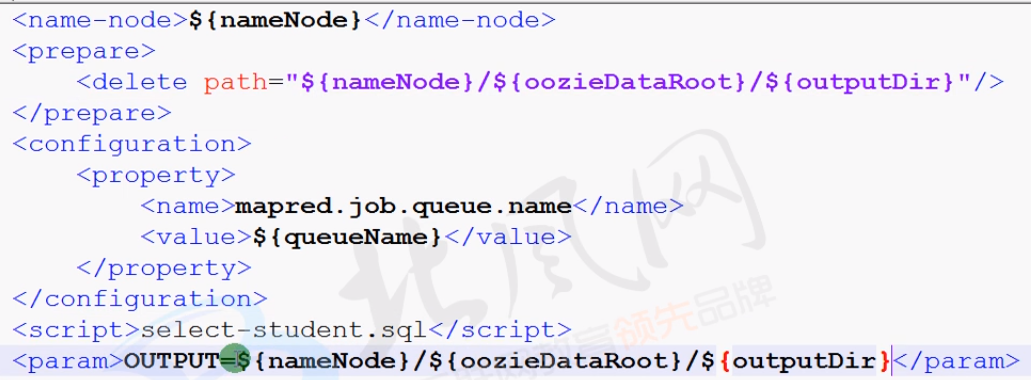
编辑job.properties



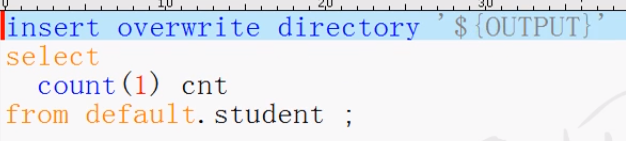
编辑workflow.xml

配置hive-site.xml文件路径





编辑脚本



其他同MapReduce action，参考[9.3](#_编写MapReduce__action)

## 编写sqoop action

Sqoop使用的是新的api，需要向[9.3](#_编写MapReduce__action)一样替换为新的api参数。

Command为sqoop导入导出数据命令，其中字段分隔符要用双引号引起来

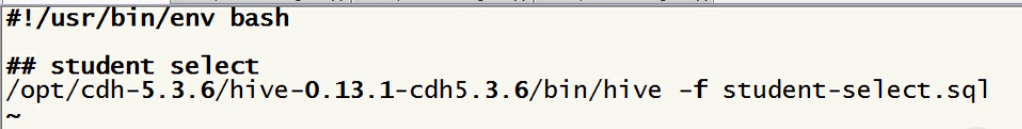
新建lib目录，复制MySQL jdbc jar包进去



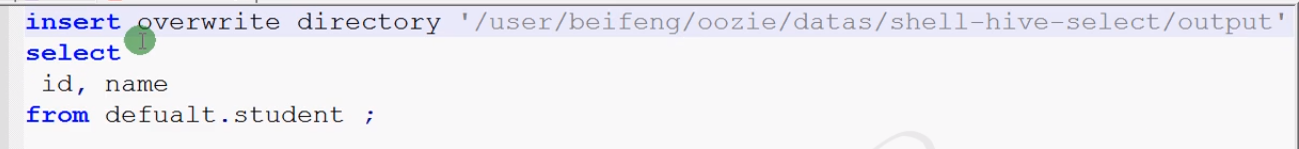
## 编写shell action

参考/home/hadoop/oozie-4.0.0-cdh5.3.6/examples/apps/shell

编辑student-select.sh



编辑student-select.sql



编辑job.properties



编辑workflow.xml



## Oozie修改时区

### 修改oozie-site.xml文件

添加

<property>

<name>oozie.processing.timezone</name>

<value>GMT+0800</value>

</property>

禁用最小时间频率

<property>

<name>oozie.service.coord.check.maximum.frequency</name>

<value>false</value>

</property>

### 修改web页面显示时区

修改/home/hadoop/oozie-4.0.0-cdh5.3.6/oozie-server/webapps/oozie/oozie-console.js

179行

function getTimeZone() {

Ext.state.Manager.setProvider(new Ext.state.CookieProvider());

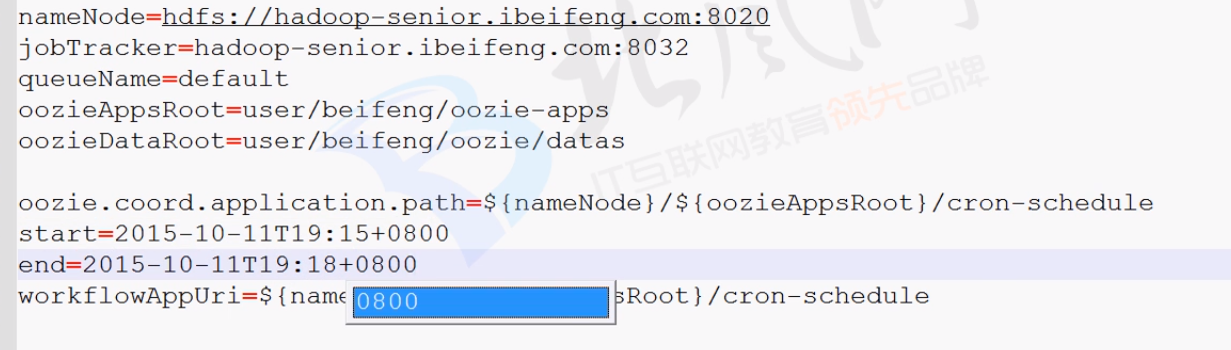
return Ext.state.Manager.get("TimezoneId","GMT+0800");

}

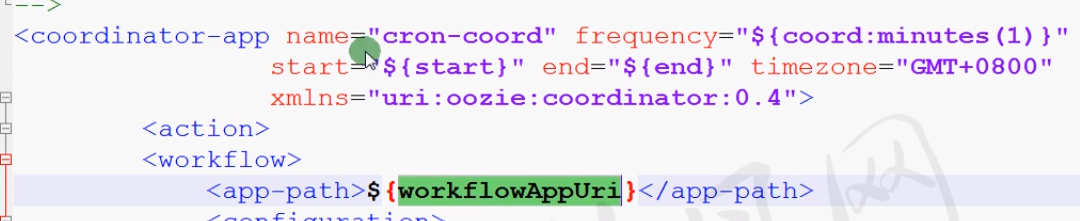
## 编写定时任务

参考/home/hadoop/oozie-4.0.0-cdh5.3.6/examples/apps/cron-schedule

编辑job.properties



编辑coordinator.xml

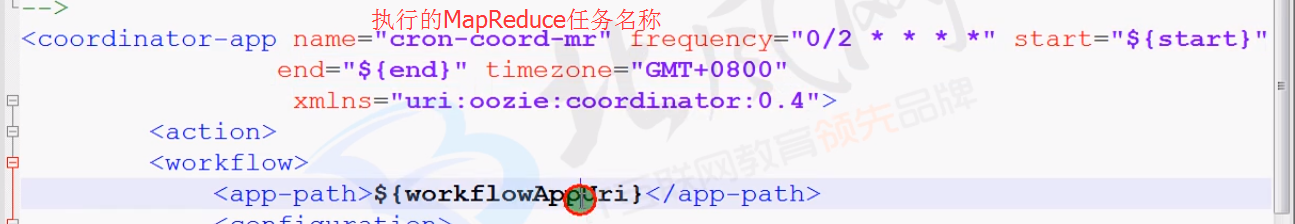


## 测试cron任务

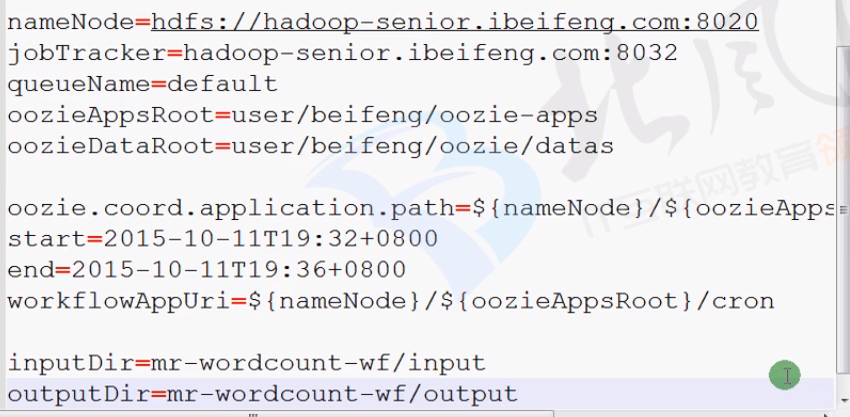
参考/home/hadoop/oozie-4.0.0-cdh5.3.6/examples/apps/cron

将之前的任意MapReduce action的workflow.xml和lib复制过来

编辑coordinator.xml



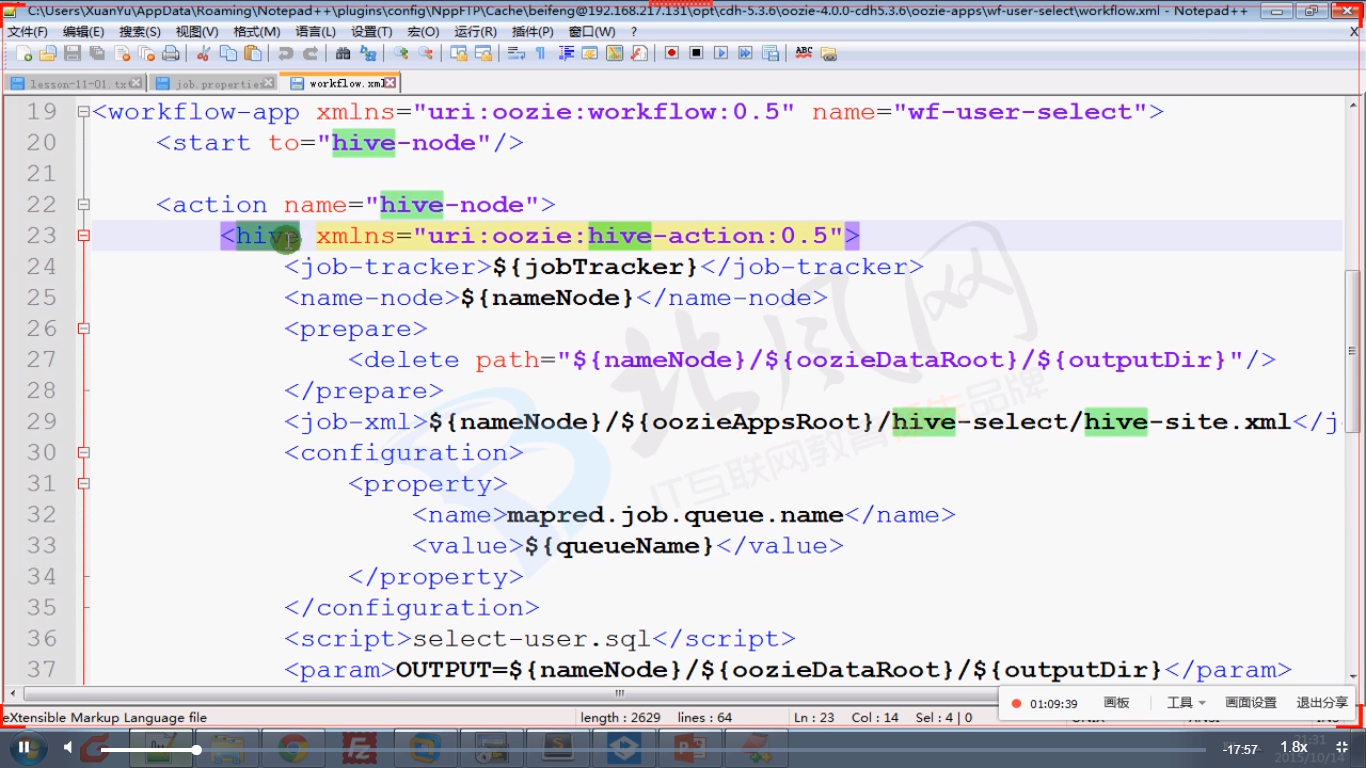
编辑job.properties

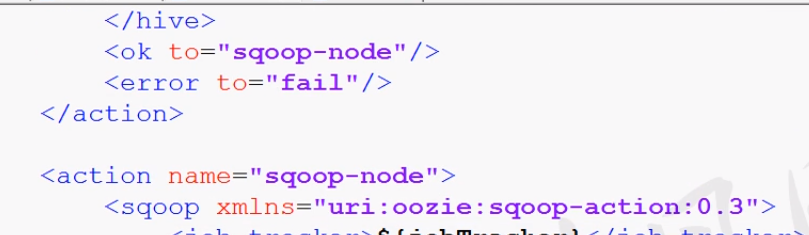


## 多个action

将两个或多个

配置workflow.xml action跳转

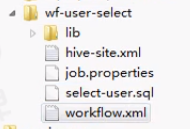




Job.properties



目录结构



任务的调度执行周期在coordinator.xml配置。

# Hue基础

## Hue安装

[Hue安装](参考文件/Hue安装.docx)

# Hbase基础

HBase新版本中，有了类似于RDBMS中DataBase的概念

命令空间

用户自定义的表，默认情况下命名空间

default

系统自带的元数据表的命名空间

hbase

## Hbase安装

[Hbase安装](参考文件/Hbase安装.docx)

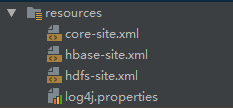
## Hbase使用

### Java api操作hbase表

1. 导入jar包

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
 <hadoop.version>2.5.0</hadoop.version>  
 <hive.version>0.13.1</hive.version>  
 <hbase.version>0.98.6-hadoop2</hbase.version>  
</properties>  
<repositories>  
 <repository>  
 <id>nexus-aliyun</id>  
 <name>Nexus aliyun</name>  
 <url>http://maven.aliyun.com/nexus/content/groups/public</url>  
 </repository>  
</repositories>  
<dependencies>  
 <dependency>  
 <groupId>org.apache.hbase</groupId>  
 <artifactId>hbase-server</artifactId>  
 <version>${hbase.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.apache.hbase</groupId>  
 <artifactId>hbase-client</artifactId>  
 <version>${hbase.version}</version>  
 </dependency>  
</dependencies>

1. 导入hadoop和hbase配置文件



1. 编写[Java类](WorkSpace/Hbase-Projects/src/main/hbaseOperations/HBaseOperation.java)

### 运行hbase自带MapReduce程序

1. 设置hadoop类路径

export HADOOP\_CLASSPATH=`${HBASE\_HOME}/bin/hbase mapredcp`

1. 命令提示

$HADOOP\_HOME/bin/yarn jar $HBASE\_HOME/lib/hbase-server-0.98.6-hadoop2.jar

CellCounter: Count cells in HBase table

completebulkload: Complete a bulk data load.

copytable: Export a table from local cluster to peer cluster

export: Write table data to HDFS.

import: Import data written by Export.

importtsv: Import data in TSV format.

rowcounter: Count rows in HBase table

verifyrep: Compare the data from tables in two different clusters. WARNING: It doesn't work for incrementColumnValues'd cells since the timestamp is changed after being appended to the log.

1. 运行一个命令

$HADOOP\_HOME/bin/yarn jar $HBASE\_HOME/lib/hbase-server-0.98.6-hadoop2.jar rowcounter 'user'

#### TSV和CSV区别

TSV

Tab分割

>> student.tsv

1001 zhangsan 26 shanghai

CSV

逗号分割

>> student.csv

1001,zhangsan,26,shanghai

### 编写hbase的MapReduce

从一张hbase表导出数据到另一张hbase表

[User2BasicMapReduce.java](WorkSpace/HbaseProjects/src/main/hbaseOperations/User2BasicMapReduce.java)

1. Mapper类继承TableMapper类
2. 将rowkey对应的value放进一个put里面
3. Reducer类继承TableReducer类
4. 写进context
5. 打包运行jar

export HADOOP\_CLASSPATH=`${HBASE\_HOME}/bin/hbase mapredcp`

$HADOOP\_HOME/bin/yarn jar $HADOOP\_HOME/jars/hbase-mr-user2basic.jar

### Importtsv导入hdfs文件到hbase表

1. 准备数据

student.tsv

10001 zhangsan 35 male beijing 0109876543

10002 lisi 32 male shanghia 0109876563

10003 zhaoliu 35 female hangzhou 01098346543

10004 qianqi 35 male shenzhen 01098732543

1. 上传数据

hdfs dfs –put student.tsv /user/hadoop/hbase/importtsv

1. 创建hbase表

create ‘student’ , ‘info’

1. 设置hadoop类路径

export HADOOP\_CLASSPATH=`${HBASE\_HOME}/bin/hbase mapredcp`:${HBASE\_HOME}/conf

1. 导入数据

${HADOOP\_HOME}/bin/yarn jar \

${HBASE\_HOME}/lib/hbase-server-0.98.6-hadoop2.jar importtsv \

-Dimporttsv.columns=HBASE\_ROW\_KEY,\

info:name,info:age,info:sex,info:address,info:phone \

student \

hdfs://MyDream:8020/user/hadoop/hbase/importtsv

### bulk批量导入数据到hbase表

不占内存

1. 设置hadoop类路径

export HADOOP\_CLASSPATH=`${HBASE\_HOME}/bin/hbase mapredcp`:${HBASE\_HOME}/conf

1. 将要导入的文件转换成Hfile格式

${HADOOP\_HOME}/bin/yarn jar \

${HBASE\_HOME}/lib/hbase-server-0.98.6-hadoop2.jar importtsv \

-Dimporttsv.columns=HBASE\_ROW\_KEY,\

info:name,info:age,info:sex,info:address,info:phone \

-Dimporttsv.bulk.output=hdfs://hadoop-senior.ibeifeng.com:8020/user/hadoop/hbase/hfileoutput \

student2 \

hdfs://MyDream:8020/user/hadoop/hbase/importtsv

1. 导入Hfile格式数据到hbase表

${HADOOP\_HOME}/bin/yarn jar \

${HBASE\_HOME}/lib/hbase-server-0.98.6-hadoop2.jar \

completebulkload \

hdfs://MyDream:8020/user/hadoop/hbase/hfileoutput \

student

### 表预分区

创建表后一般导入数据，当导入数据过多时region分割成两个region时负担由一个regionserver承担，容易产生问题，预分区将建立多个region由多个regionserver管理数据，负载平衡。

按时间创建分区

1. 命令行指定分区

create 'bflogs', 'info', SPLITS => ['20151001000000000', '20151011000000000', '20151021000000000']

插入数据时按rowkey插入对应region。

1. 文本文件指定分区

create 'bflogs2', 'info', SPLITS\_FILE => '/opt/datas/bflogs-split.txt'

bflogs-split.txt

20151001000000000

20151011000000000

20151021000000000

### Hbase与hive集成

1. 将hbase jar包导入hive

ln -s $HBASE\_HOME/lib/hbase-server-0.98.6-hadoop2.jar $HIVE\_HOME/hbase-server-0.98.6-hadoop2.jar

ln -s $HBASE\_HOME/lib/hbase-client-0.98.6-hadoop2.jar $HIVE\_HOME/hbase-client-0.98.6-hadoop2.jar

ln -s $HBASE\_HOME/lib/hbase-protocol-0.98.6-hadoop2.jar $HIVE\_HOME/hbase-protocol-0.98.6-hadoop2.jar

ln -s $HBASE\_HOME/lib/hbase-it-0.98.6-hadoop2.jar $HIVE\_HOME/hbase-it-0.98.6-hadoop2.jar

ln -s $HBASE\_HOME/lib/htrace-core-2.04.jar $HIVE\_HOME/htrace-core-2.04.jar

ln -s $HBASE\_HOME/lib/hbase-hadoop2-compat-0.98.6-hadoop2.jar $HIVE\_HOME/lib/hbase-hadoop2-compat-0.98.6-hadoop2.jar

ln -s $HBASE\_HOME/lib/hbase-hadoop-compat-0.98.6-hadoop2.jar $HIVE\_HOME/lib/hbase-hadoop-compat-0.98.6-hadoop2.jar

ln -s /opt/modules/hbase-0.98.6-hadoop2/lib/high-scale-lib-1.1.1.jar /opt/modules/hive-0.13.1/lib/high-scale-lib-1.1.1.jar

1. 配置hive的hive-site.xml文件

<property>

<name>hbase.zookeeper.quorum</name>

<value>MyDream</value>

</property>

两种方式：

管理表

创建hive表的时候，指定数据存储在hbase表中。

CREATE TABLE hbase\_table\_1(key int, value string)

STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'

WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,cf1:val")

TBLPROPERTIES ("hbase.table.name" = "xyz");

>> 外部表

现在已经存在一个HBase表，需要对表中数据进行分析。

CREATE EXTERNAL TABLE hbase\_user(id int, name string,age int)

STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'

WITH SERDEPROPERTIES ("hbase.columns.mapping" = ":key,info:name,info:age")

TBLPROPERTIES ("hbase.table.name" = "user");

### Hbase与hue集成

1. 配置hue的hue.ini文件

hbase\_clusters=(Cluster|MyDream:9090)

hbase\_conf\_dir=/home/hadoop/hbase-0.98.6-hadoop2/conf

1. 启动thrift服务

hbase-daemon.sh start thrift

1. 启动hadoop和hbase master

start-all.sh

start-hbase.sh

hbase-daemon.sh start master

1. 启动hue

Supervisor

# CDH安装

## CM及CDH5.7安装文档

[CM及CDH5.7安装文档](参考文件/CM及CDH5.7安装文档.docx)

[参考文件](测试数据/课程十一、企业大数据平台高级应用)

# Kafka基础

## Kafka安装

[Kafka安装](参考文件/kafka安装.docx)

## Kafka使用

[示例代码](WorkSpace/Kafka-Projects)

# Storm基础

## Storm安装

[Storm伪分布式安装](参考文件/Storm伪分布式安装.docx)

## Storm的安装部署架构

nimbus主节点：

1）接收客户端提交的任务请求，任务由Nimbus进行分配，将分配信息提交到

Zookeeper集群（在Zookeeper相应的znode节点上写入任务分配信息，由supervisor查看这些znode上的任务分配信息，获取分配到的任务）

2）监控整个集群的状态（从Zookeeper集群中相应znode上读取supervisor、worker进程的状态信息数据）

3）容错：当任务在某些Supervisor节点上运行的时候由于Supervisor进程失效，重新将这些任务分配给其他supervisor运行

Supervisor：

1）需要定时将自己的运行状态信息（心跳信息）汇报给zookeeper（在Zookeeper相应的znode节点上写入心跳信息）

2）接收Nimbus分配给它的任务，负责启动、停止工作进程worker，其本身并不是执行任务的工作进程，worker的容错由Supervisor进程负责

worker进程：

并不是常驻进程，不能通过手动启动

真正执行任务的进程

worker进程启动后，也会定时将状态信息汇报给zookeeper

executor 线程：

worker进程来启动，executor负责执行客户提交到Storm集群上任务中Task(spout/bolt)

## Storm的任务

1、杀掉任务

$ bin/storm kill wordcount

2、Topology 提交到Storm集群上的任务

拓扑图 有向无环图 DAG

比较Storm与MapReduce：

1）Topology 提交到Storm集群上运行，除非手动执行kill命令，否则将一直永远运行下

MapReduce任务把数据处理完就终止

2）实时处理 批处理

地铁站：自动扶梯

普通电梯

Topology：

spout 数据采集器 ，由它负责从数据源上获取数据，转发给后面的bolt进行处理

bolt：数据处理器，在bolt里面实现数据的处理逻辑

stream：Tuple格式 keyvalue对的集合

{"name":"zhangsan","sex","M","age":26}

## zookeeper在Storm中的作用

znode

/storm

/workerbeats worker工作进程的状态信息

/errors topology在运行过程中出现异常的task信息，方便Nimbus将运行出错的任务进行重新分配

/supervisors Supervisor节点的状态信息

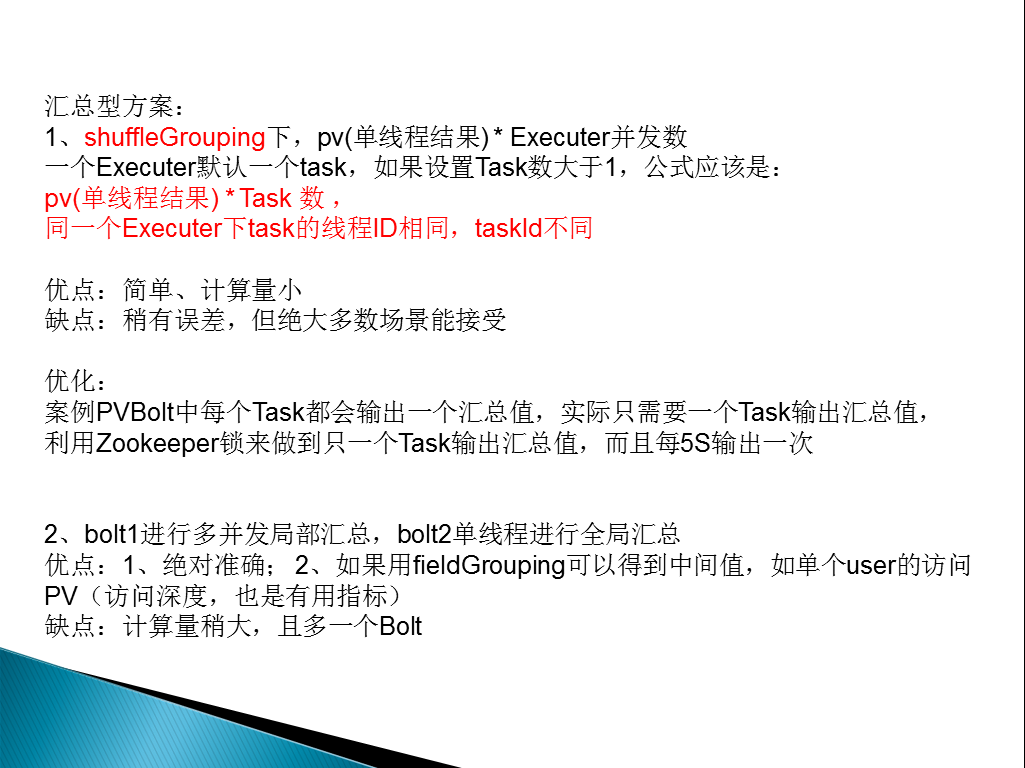
/storms Topology的基本配置信息

/assignments Topology任务的分配信息

/storm/workerbeats/wordcount-3-1471753943/各个worker对应的znode

## 实现Topology

[Storm编程](WorkSpace/Storm-Projects)模型



### Spout+bolt模式实现

#### Topology的构造

[backtype.storm.topology.TopologyBuilder](WorkSpace/Storm-Projects/src/main/java/storm/topology/WordCountTopology.java)



#### Spout组件的编写

实现接口 backtype.storm.topology.IRichSpout;

或者继承backtype.storm.topology.base.BaseRichSpout;

[示例代码](WorkSpace/Storm-Projects/src/main/java/storm/topology/SentenceSpout.java)

##### 方法介绍

@Override

public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {

// TODO Auto-generated method stub

}

open 方法，是spout的组件初始化方法，而且Spout实例创建后首先被调用，只调用一次

@Override

public void close() {

// 对于资源的释放关闭，可以在该方法中实现

}

@Override

public void nextTuple() {

// 实现如何从数据源上获取数据的逻辑

// 以及向后面的组件bolt发射数据

}

nextTuple 循环调用

@Override

public void ack(Object msgId) {

// TODO Auto-generated method stub

}

Topology启用了消息可靠性保障机制，当某个Tuple在Topology上处理成功后，调用ack方法执行一些消息处理成功后该干的事情

@Override

public void fail(Object msgId) {

// Topology启用了消息可靠性保障机制，某个Tuple在后面处理失败，该干什么

// 比如重试，重试达到最大可重试就丢弃

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

// 声明向后面组件发射的Tuple keys依次是什么

}

@Override

public Map<String, Object> getComponentConfiguration() {

// 设置该组件Spout一些专用的参数

return null;

}

kafkaSpout 向后发射的Tuple {"str":"msg"}

注意点：

Topology中使用的一些类，最好都要实现序列化接口 java.io.Serializable

#### Bolt组件的编写

实现backtype.storm.topology.IRichBolt

或者继承backtype.storm.topology.base.BaseRichBolt

[示例代码](WorkSpace/Storm-Projects/src/main/java/storm/topology/SplitBolt.java)

@Override

public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {

//类似于spout中open方法

}

SpoutOutputCollector spout组件中tuple的发射器

OutputCollector bolt组件中tuple发射器

@Override

public void execute(Tuple input) {

// TODO Auto-generated method stub

}

execute 类似于Spout的nextTuple方法

@Override

public void cleanup() {

// TODO Auto-generated method stub

}

类似于spout中close方法

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

// 声明向后面组件发射的Tuple keys依次是什么

}

@Override

public Map<String, Object> getComponentConfiguration() {

// 设置该组件Spout一些专用的参数

return null;

}

#### 数据流分组

方式

shuffleGrouping 随机分配

fieldsGrouping 根据key分组进行分配

globalGrouping 全局分组 只会将tuple往后面组件中固定一个上发送

#### 消息可靠性保障机制

启用消息可靠性保障机制：

ack

fail

方法

Spout端：

1）发射器发射tuple时，需要指定一个msgID

collector.emit(new Values(sentence),mssageId );

2）使用缓存所发射的tuple，Map key=msgID,value = Values

private Map<Object,Values> tuples;

3）ack方法

// 确认发射成功，将tuple从缓存中移除

tuples.remove(msgId);

4）fail方法

重试

// 重试

Values values = tuples.get(msgId);

// 重新发射

collector.emit(values,msgId );

Bolt端：

1）如果bolt端继续往后面组件发射，需要锚定前面的tuple

// 启用消息可靠性保障机制，需要锚定接收到tuple

collector.emit(input,new Values(word));

2）处理完tuple后

// 确认处理结束

collector.ack(input);

try{

}catch{

// 处理失败

collector.fail(input);

}

#### Storm的并发

worker工作进程的并发度：

Topology由多少个worker来执行

Config conf = new Config();

conf.setNumWorkers(2);

executor线程的并发度：

指Spout、bolt由多少个线程来执行

builder.setSpout(SPOUT\_ID, new SentenceSpout(),2);

builder.setBolt(SPLIT\_BOLT, new SplitBolt(),3).shuffleGrouping(SPOUT\_ID);

直接指定task并发度：

builder.setBolt(SPLIT\_BOLT, new SplitBolt(),3).setNumTasks(12).shuffleGrouping(SPOUT\_ID);

bolt有12个Task来运行，12task由3个executor执行

每个executor线程要运行4个task，4个Task只能在executor线程上轮流执行

acker组件：

执行消息可靠性保障机制的任务

如果启用了消息可靠性保障机制，对Topology运行性能一定会造成下降影响，如果下降程度超过可接受范围，解决办法：增加acker组件的并发度（增加执行acker任务的executor线程个数）

conf.setNumAckers(4);

### 批处理底层代码实现

[示例代码](大数据视频笔记/课程七、大数据核心开发技术%20-%20Storm实时数据处理/代码/15.Storm事务案例实战之%20ITransactionalSpout/code)

1. 新建批处理spout类

public class MyTxSpout implements ITransactionalSpout<MyMata>

1. 实现源数据类MyMeta

定义开始位置和批处理tuple个数两个变量，并生成get和set和toString方法。

1. MyTxSpout添加序列化字段serialVersionUID并实现为实现方法
2. 新建MyCoordinator类

public class MyCoordinator implements ITransactionalSpout.Coordinator<MyMata>

实现initializeTransaction和isReady方法

1. MyTxSpout 类的getCoordinator方法返回MyCoordinator
2. 新建MyEmitter类（发射数据用）

public class MyEmitter implements ITransactionalSpout.Emitter<MyMata>

1. Fds

### Trident模式实现

#### 概念

1）Storm高层次的抽象

2）在Trident中保留了Spout，但是不再有Bolt组件，将之前Storm在Bolt组件中所实现的数据处理逻辑抽象成一系列的Operation，比如过滤、函数、分组统计等等。

3）Trident封装好了消息可靠性保障机制

4）Trident 批次概念

a.将固定条数的Tuple划分为一个批次

b.给每个批次一个编号

c.更新统计结果状态，要严格按照批次顺序进行更新

5）事务控制

3个层次：

NON-Transactional：非事务控制

允许同一个批次内的

部分处理成功

失败的Tuple，可以在其他批次内进行重试，有可能导致被成功处理多次（在后面的多个批次内被重试）

也有可能不进行重试，被成功处理零次

Transactional：严格的事务控制

要求批次内处理失败的Tuple只能在本批次内进行重试（以相同的批次号进行重试）

如果tuple一直重试不成功，就会将整个任务程序挂起，不会进行下个批次的处理

没有容错

Opaque-Transactional：透明事务控制

批次内的tuple处理完后，先把成功的更新掉

失败的tuple允许在其他批次内进行重试，只会有一次成功处理

有容错

#### Trident的编码开发

##### 构造topology

storm.trident.TridentTopology

storm.trident.Stream

trident中的Spout：从数据源上获取数据

将获取到的数据封装到一个个批次内，而且给每个批次指定一个批次号

each方法：链式调用，前面的each方法调用过滤器后过滤的结果对后面的each有作用。

>>>>>>>> 过滤操作Filter： 对满足条件的tuple进行保留，不满足的丢弃

isKeep方法：实现tuple是否继续保留在Stream的逻辑，返回TRUE则保留，FALSE则丢弃。

在Trident编码中，要注意的：

Stream流在经过各种操作后 Tuple的演变

Tuple中keyvalue对的演变（keyvalue保留、丢弃、追加、替换）

比如：Filter 仅仅只是在Stream保留或者丢弃tuple，而不会对Tuple中的keyvalue对进行更改

>>>>>> Functions函数操作

实现storm.trident.operation.Function

execute方法

Stream经过函数操作后，将新产生的keyvalue对追加到原来的Tuple中，注意的是，如果没有新产生keyvalue对，那么相应的tuple将会被丢弃掉

>>>>>>>> Stream.project方法，

指定保留Tuple中的哪些keyvalue、丢弃哪些keyvalue

NOTE：Stream.project方法与过滤器区别，过滤器不会将原来的keyvalues对丢弃，如原来key有str，word，经过过滤器后只会将这两个key中不符合要求的keyvalues过滤掉，而project函数将未指定的key全部扔掉。

>>>>> topology提交storm集群测试

// 当args没有值，运行过程中没有指定参数

if(args == null || args.length <= 0){

// 本地测试

LocalCluster localCluter = new LocalCluster();

localCluter.submitTopology("wordcountTrident", config, topology.build());

}else{

// 提交集群运行

try {

StormSubmitter.submitTopology(args[0], config, topology.build());

} catch (AlreadyAliveException e) {

e.printStackTrace();

} catch (InvalidTopologyException e) {

e.printStackTrace();

}

}

maven打包

maven-assembly-plugin 插件

$ bin/storm jar /home/ibeifeng/storm-test-0.0.1-SNAPSHOT-jar-with-dependencies.jar com.ibeifeng.storm.trident.WordCountTrident wordcountTrident

##### Trident的并发度

.each(new Fields("str"), new SplitFunction(),new Fields("word"))

//设置2个executor来执行splitfunction操作

.parallelismHint(2)

在某个操作调用后，stream调用 parallelismHint，设置前面这个操作的并发度

##### Operation的特性

是否同一分区内、是否需要跨网络

分区：指的就是一个在excutor线程中运行的task

Filter Function project 分区内的操作，只是对本分区内的Tuple进行操作

##### 重分区操作

和数据流分组一样

partitionBy 类似于 mapreduce分区

例如：.partitionBy(new Fields("word"))

取Tuple，key名称为word的keyvalue的value值 求hashcode ，然后根据哈希值 % 分区数进行取模，分区数由下一个.parallelismHint(3)指定。分区号可在下一个function的prepare函数中查看：

partitionIndex = context.getPartitionIndex();

System.err.println("countFunction的分区编号：" + partitionIndex);

特点：

相同的keyvalue对 进入同一个分区

同一个分区内的keyvalue对是否都相同？不一定相同

groupBy：// MapReduce 分区 + 分组

例如：.groupBy(new Fields("word"))

根据指定的keyvalue对进行分组

实质上进行了两步操作：

1）partitionBy

2）再将同一个分区内的相同的keyvalue对分为一组

Groupby后需转换为stream，

groupBy(**new** Fields("word")).toStream()

groupBy：

1）同一批次内各个分区的分组统计（局部，不需要跨网络传输）

##### partitionAggregate

.chainedAgg()

.chainEnd()

构造聚合链

只对同一个批次内进行统计

CountAggregator实现：

1. 定义内部类
2. Aggregate方法实现累加
3. Complete方法提交tuple

.chainedAgg()  
// 同一批次分区内分组统计  
.partitionAggregate(**new** Fields("word"), **new** CountAggregator(), **new** Fields("count"))  
.chainEnd()

2）全局统计 ： 需要跨网络传输

persistentAggregate

先局部再全局，可用于避免数据倾斜

.chainedAgg()  
// 同一批次分区内分组统计  
.partitionAggregate(**new** Fields("word"), **new** CountAggregator(), **new** Fields("count"))  
.chainEnd()  
.parallelismHint(3)  
//.each(new Fields("word","count"),new PrintTestFilter())  
.groupBy(**new** Fields("word"))  
//对count进行sum操作，new MemoryMapState.Factory()指定存储在内存中  
.persistentAggregate(**new** MemoryMapState.Factory(), **new** Fields("count"), **new** Sum(),**new** Fields("globalCount"))  
.newValuesStream()

直接全局统计

.persistentAggregate(**new** MemoryMapState.Factory(), **new** Count(),  
**new** Fields("globalCount"))  
.newValuesStream()

shuffle 随机分配 前面组件产生的Tuple随机分给后面组件的各个分区

类似于shuffleGrouping

global 类似于gloablGrouping 所有的Tuple只会进入到后面一个分区内 （慎重使用）

batchGlobal：同一批次内的tuple只会进入到后面一个分区内，不同批次内的Tuple可能进入不同的分区内

broadcast 广播 将tuple复制给后面的所有分区，一般结合drpc使用

### DRPC

分布式远程过程调用

请求-响应

需求：查询当前统计结果

\*统计结果数据存入到HBase、Redis

TridentState 统计结果状态

TridentTopology.newDRPCStream

[本地DRPC](WorkSpace/Storm-Projects/src/main/java/com/ibeifeng/storm/trident/WordCountTridentWithLocalDRPC.java)：

LocalDRPC localDRPC = **new** LocalDRPC();  
// 指定drpc查询服务名称为drpcService  
topology.newDRPCStream("drpcService", localDRPC)  
//对localDRPC.execute方法传过来的参数进行解析  
//　 key 名称一定是args  
.each(**new** Fields("args"), **new** SplitFunction1(),**new** Fields("word"))

//从状态state中取出要查询的字段  
.stateQuery(state, **new** Fields("word"),**new** MapGet(), **new** Fields("count1"))  
.each(**new** Fields("word","count1"), **new** PrintTestFilter());

String jsonResult = localDRPC.execute("drpcService", "hadoop mapreduce storm");  
System.out.println(jsonResult);

[集群上使用drpc](WorkSpace/Storm-Projects/src/main/java/storm/trident1/WordCountDrpcClient.java)

：要启用drpc server进程 需要先修改storm.yaml配置文件添加

drpc.servers

drpc.port

重启storm集群

添加启动 drpc服务进程

$ nohup bin/storm drpc > /dev/null 2>&1 &

### 利用Zookeeper锁实现指定时间访问结果

[示例代码](大数据视频笔记/课程七、大数据核心开发技术%20-%20Storm实时数据处理/代码/10.案例优化引入Zookeeper锁控制线程操作/code/lesson/visits/PVBolt.java)

## Storm批启动脚本编写

脚本反引号``为先执行``里面的命令

统一写在节点nimbus服务器

### zookeeper批启动脚本

zoo.cfg

server.1=hostname:2888:388

zookeeper.sh 内容：

#!/bin/bash

if [ $# -ne 1 ];then

echo "Usage: bin/zkServer.sh {start|start-foreground|stop|restart|status|upgrade|print-cmd}"

exit 4

fi

source /etc/profile

ZK\_HOME=/opt/modules/zookeeper-3.4.5-cdh5.3.6

for node in hive-stu.ibeifeng.com

do

echo "$1 zookeeper in $node"

ssh $node "source /etc/profile && $ZK\_HOME/bin/zkServer.sh $1"

done

### Storm停止脚本

主节点

nimbus

ui

从节点

logviewer

supervisor

在storm的主节点上添加一个指定supervisor节点的文件

supervisors

/opt/modules/apache-storm-0.9.6/conf/supervisors

内容 一行一个Supervisor节点服务器主机名

$ kill -9 `ps -ef | grep daemon.nimbus | awk '{print $2}' | head -n 1`

$ kill -9 `ps -ef | grep daemon.supervisor | awk '{print $2}' | head -n 1`

$ kill -9 `ps -ef | grep ui.core | awk '{print $2}' | head -n 1`

$ kill -9 `ps -ef | grep daemon.logviewer | awk '{print $2}' | head -n 1`

stop-storm.sh

内容：

#!/bin/bash

source /etc/profile

STORM\_HOME=/opt/modules/apache-storm-0.9.6

#先在主节点上停止nimbus和ui进程

kill -9 `ps -ef | grep daemon.nimbus | awk '{print $2}' | head -n 1`

kill -9 `ps -ef | grep ui.core | awk '{print $2}' | head -n 1`

#在从节点上停止logviewer和supervisor

SUPERVISORS=$(cat $STORM\_HOME/conf/supervisors)

for supervisor in $SUPERVISORS

do

echo "stop supervisor and logviewer in $supervisor"

ssh $supervisor kill -9 `ssh $supervisor ps -ef | grep daemon.supervisor | awk '{print $2}'|head -n 1`

ssh $supervisor kill -9 `ssh $supervisor ps -ef | grep daemon.logviewer | awk '{print $2}'|head -n 1`

done

### 批启动脚本

start-storm.sh

内容：

#!/bin/bash

source /etc/profile

STORM\_HOME=/opt/modules/apache-storm-0.9.6

#先在主节点上启动nimbus和ui进程

$STORM\_HOME/bin/storm nimbus >/dev/null 2>&1 &

$STORM\_HOME/bin/storm ui >/dev/null 2>&1 &

#在从节点上启动logviewer和supervisor

SUPERVISORS=$(cat $STORM\_HOME/conf/supervisors)

for supervisor in $SUPERVISORS

do

echo "start supervisor and logviewer in $supervisor"

ssh $supervisor "source /etc/profile && nohup $STORM\_HOME/bin/storm supervisor >/dev/null 2>&1" >/dev/null 2>&1 &

ssh $supervisor "source /etc/profile && nohup $STORM\_HOME/bin/storm logviewer >/dev/null 2>&1" >/dev/null 2>&1 &

done

## storm项目

### 需求背景

订单分析

项目需求：

电商

用户提交订单并且成功支付

收集订单详情

分析：

1、每天（每小时、每分钟）电商网站总销售额、订单笔数

2、基于地域（国家、省份、城市）、时段

3、从客户角度，统计客户消费情况

客户在电商网站消费行为

客户流失趋势

推荐系统

技术框架 ： Storm的Trident

准确度

批次

### 测试数据准备

[生成测试数据](WorkSpace/Storm-Projects/src/main/java/storm/trident2/test/OrderDataGenerator.java)

订单记录格式

"timestamp" "consumer" "productName" "price" "country" "province" "city"

### 订单处理

[订单处理Topology](WorkSpace/Storm-Projects/src/main/java/storm/trident2/orderprocess/OrderProcessingTrident.java)

实现：

1）从kafka上读取数据

kafkaspout

Trident kafkaspout

TransactionalTridentKafkaSpout

消息处理失败重试：只能在之前的批次内进行重试，而且如果一直不成功，将会挂起Storm任务应用程序

OpaqueTridentKafkaSpout 透明

提供容错机制

处理失败的消息可以在其他批次内进行重试，但是只会被成功处理一次

exactly-once 有且只有一次成功

2）解析订单记录

Function函数操作

[订单解析function](WorkSpace/Storm-Projects/src/main/java/storm/trident2/orderprocess/OrderParseFunction.java)

3）进入分析

完成“每天（每小时、每分钟）电商网站总销售额、订单笔数”

select sum(price),count(1),date from 订单记录 group by date

分组统计：

两种方法：

1）直接进行全局统计（跨网络数据传输）（数据量不大情况下）

2）先在同一批次各分区内进行局部统计（不需要进行跨网络传输），然后对局部统计好的结果进行全局统计

group by

partitionAggregate 局部统计

persistentAggregate 全局统计

聚合操作链

作用：一次groupby要对多个集合操作起作用，下面的聚合操作都是对前面的groopby结果集进行操作

.chainedAgg()

// 统计同一批次内各分区中订单金额总和

.partitionAggregate(new Fields("price"), new SaleSum(),

new Fields("saleTotalAmtOfPartDay"))

// 统计同一个批次内各分区中的订单笔数之和

.partitionAggregate(new Count(), new Fields("numOrderOfPartDay"))

.chainEnd()

project 将某些不用的keyvalue对从Tuple中移除掉

将某些tuple从Stream中移除掉 过滤器

4）使用HBase数据库进行存储统计结果状态

Trident State：

rowkey value:

NON-TRANSACTIONAL 统计值

TANSACTIONAL ： BATCH\_ID 统计值

OPAQUE TRANSACTIONAL ： BATCH\_ID 统计值 上个批次的统计值

在maven项目中创建src/main/resources的 source foulder，将hbase-site.xml放进去

状态信息

zookeeper

$ bin/storm jar /home/ibeifeng/storm-test-0.0.1-SNAPSHOT-jar-with-dependencies.jar com.ibeifeng.storm.orderprocess.OrderProcessingTrident orderProcessTrident

# Scala基础

## Scala基础语法

1、第一种定义变量方式

val xx = 123

-》值不可改变（常量）

2、第二种定义变量方式

var age: Int = 20

var name = "zhangsan"

-》值可变

3、注意：变量名后面需要跟上冒号，类型首字母必须是大写

4、var sex: String = \_

下划线表示占位符，注意这种写法需要指定一个类型

Unit相当于是void

Null空值或者空引用

Nothing表示没有值

Any表示是所有类型的超类，所有类型都继承于它

5、懒执行

关键词：lazy

lazy val price = 66.58

在一开始定义的时候不会初始化值

而是在第一次调用的时候才去初始化

使用的场景比如：读取一些大文本文件的时候，可以定义lazy

6、Scala函数的定义

-》Scala中不需要return返回，最后一个函数就是返回值

-》Scala中也不需要分号结尾

无参无返回值定义函数：

def add()={

println("say hello...")

}

由于没有参数那么括号可以不加

add()或add

嵌套函数（内部函数）

def fun1(a: Int){

def fun2(b: Int){

println("fun1...fun2..." + (a + b))

}

fun2(100)

}

匿名函数：没有函数名称

标志 =>

(x: Int) => x + 1

将函数作为参数进行传递，这种定义的方式我们叫做：高阶函数

def add = (x: Int,y: Int) => x + y

def min = (x: Int , y: Int) => {

if(x < y)

x

else

y

}

val xx = (x: Int) => x + 1

小结：匿名函数不但可以赋值给函数，也可以赋值给变量

下划线前面必须空格，这是语法格式

将函数赋值给变量

val add1 = add \_

7、Scala默认值（缺省值）

def sayName(name: String = "zhangsan"){

println("Hello " + name + "!")

}

def sayHello(name: String, age: Int = 20){

print("Hello ," + name + " , your age is " + age)

}

8、Scala可变参数

def printCourses(course : String\*)={

course.foreach(x => println(x))

}

匿名函数：

(x => println(x))

等同于这样的写法：

(x: String) => println(x)

-》\*代表的是参数可重复

scala> val arr = Array("aa","bb","cc","dd")

arr: Array[String] = Array(aa, bb, cc, dd)

scala> printCourses(arr: \_\*)

aa

bb

cc

dd

\_\*代表数组中元素一个个进行传递，而不是整个数组传递，传递整个数组是不认可的。

9、Scala控制语句

-》if(age > 18) 1 else 0

-》if表达式本身是有值的

-》以这个例子为列，整个表达式的值就是1

-》最后一个语句就是表达式的返回值

这条语句没有返回值，因为isadult = 1仅仅只是赋值

if(age > 18) isadult = 1 else isadult = 0

if语句的类型推断，如果两个类型不同Scala编译器就会去找两个类型的公共父类

if(age > 18) "adult" else 0

if( age < 12 ) "child"等于if( age < 12 ) "child" else ()

10、Scala语句终结符和块表达式

1、输出，格式化输出

printf("Hello , my name is %s , I'm %d years old ", "jack",20)

2、输入readLine()，类似Java中的System.in或者scanner

### for循环

1、1 to 10 一种表现的形式，可以理解为是一个范围

2、for(i <- 1 to 10) print( i + " ")

-》返回Range可以理解为是一个序列、集合或者是一个范围

for(i <- 1 until n) print( i + " ")

until和to相比，它是不包含上限区间或者上边界

for(c <- "Hello World") print(c + " ")

类似Java中的（String arg:args）

可以指定步长，Range和until一样有一个左闭右开的规律

Range(1,10,2)

直接在for循环中加入if语句这样的写法，称作：if守卫

for( x <- 1 to 20 if x % 2 == 0 ) println(x)

for推导式：

for( i <- 1 to 10) yield i

for循环中的yield会把当前的元素记录下来，保存在集合中，当循环结束

就会返回该集合。

for(i <- 1 to 9; j <- 1 to 9){

if( j == 9 ){

println(i \* j)

}else{

print(i \* j + "\t")

}

}

相当于Java中的for嵌套循环

scala中包的符号使用： 下划线相当于Java中的\*

import scala.util.control.Breaks.\_

### 数组

1、定义数组

val arr = Array("aa","bb","cc")

2、访问数组

arr(0)，下标从0开始，没有方括号的概念，不能使用

3、修改元素

arr(1) = "Hello"

4、可以是不同类型的元素，会取公共父类型

val b = Array("jack",20)

5、val aa = new Array[String](3)另一种创建的方式

-》需要指定类型和长度

6、aa.length访问数组长度

7、改变数组长度可以使用ArrayBuffer

-》使用之前需要导包：import scala.collection.mutable.ArrayBuffer

-》val c = ArrayBuffer[Int]()创建空的ArrayBuffer

-》添加元素使用+=符号c += 1

-》添加一个集合使用++=符号c ++= Array(4,5,6,7,8,9)

-》一次删除多个元素，c.trimEnd(6)，注意是从尾端开始删除，指定元素个数

-》c.insert(0,9)，在任意位置插入指定的元素

-》c.remove(0,3)指定删除元素

8、Array和ArrayBuffer可以进行互相的转换

9、遍历

for(i <- 0 until c.length) println(c(i))

0 until c.length相当于是0 until c.length -1

for(i <- 0 until (c.length,2)) println(c(i))

for(e <- c) println(e)

for(i <- (0 until c.length).reverse) println(c(i))

scala.collection.mutable -> 可变的

scala.collection.immutable -> 不可变的

### 集合Map和元组tuple

1、map代表的是<key,value>的映射

2、创建不可变的map，默认情况下都是不可变的，一般不常用

val stu = Map("jack" -> 20 , "tom" -> 21, "peter" -> 22)

3、创建可变的map

val stu = scala.collection.mutable.Map("jack" -> 20 , "tom" -> 21, "peter" -> 22)

4、修改元素的值

stu("jack") = 30

5、一个key对应一个value，由多个<key,value>组成的集合叫做map

6、val stu = Map(("jack",30),("tom",20),("peter",21))

-》另一种创建的方式

7、创建一个空的map

val stu = new scala.collection.mutable.HashMap[String,Int]

8、判断key是否存在

val stus = if(stu.contains("jack")) stu("jack") else 0

val stus = stu.getOrElse("jack",0)

修改值：

stu("jack") = 40

添加：默认情况下map的key排列是无序的

stu +=("mike" -> 35)

mutable可以使用 +=、++=、-=、--=

immutable可以使用+、-、++、--

(key,value)就称作：tuple元组队

遍历的第一种方式：

for((key,value) <- stu) println(key + ": " + value)

遍历的第二种方式：

for(key <- stu.keySet) println(key)

for(key <- stu.values) println(key)

安照key的字母进行排序

val stu = scala.collection.SortedMap("dd" -> 25,"cc" -> 30,"aa" -> 25,"bb" -> 29)

按照插入元素的顺序进行排列

val stu = new scala.collection.mutable.LinkedHashMap[String,Int]

zip拉链操作

scala> val name = Array("jack","tom","leo")

scala> val age = Array(20,30,40)

scala> name

scala> age

scala> val nameAge = name.zip(age)

nameAge: Array[(String, Int)] = Array((jack,20), (tom,30), (leo,40))

遍历数组：

for((name,age) <- nameAge) println(name + ": " + age)

### Scala集合list和set

1、seq中包含了：Range和ArrayBuffer以及List

2、集合也是系统的区分了可变和不可变

3、创建list集合

val list = List(1,2,3,4,5)

4、Scala中的list包含两个部分，head+tail或head+Nil

list.head 代表返回第一个元素

list.tail 代表返回一个不包含第一个元素的集合

Nil代表空的list集合

val list2 = 1::Nil

-》注意集合的顺序，前面是元素后面是集合

val list3 = 2::list2

-》前面的元素就是新的list的head，后面的就是新的list的tail

5、创建一个可变的list集合

val listBuffer = scala.collection.mutable.ListBuffer[Int]()

添加元素：

listBuffer += 2

listBuffer +=(3,4,5)

listBuffer ++= List(6,7,8)

listBuffer -= (4,7)

6、list练习【指定前缀】

需求：使用递归函数给list中的每个元素都加上指定的前缀，并且打印出加上前缀的元素。

注意：最后一个元素的时候它的tail就是Nil

def dtor(list: List[Int], pfix: String){

if(list != Nil){

println(pfix + list.head)

dtor(list.tail,pfix)

}

}

需求：去掉数组

7、set集合创建

不可变：val set = Set(1,2,3,4)

可变：val s = scala.collection.mutable.Set(1,2)

添加元素：-》不可变的：+ ++ - --

-》可变的：+= ++= -= --=

-》不可变的：+ ++ - --

### Scala数组的转换操作

1、对数组中的元素进行通用的转换，转换成一个新的数组

2、第一种方式：yield转换

val a2 = for(ele <- a ) yield ele \* ele

val b2 = for(ele <- b) yield ele \* ele

val a3 = for(ele <- a if ele % 2 == 0 ) yield ele \* ele

3、第二种方式：函数式编程转换

a.filter(\_%2 == 0).map(2 \* \_)

-》下划线代表占位符

-》filter首先过滤出偶数

-》接着map对其进行映射，把过滤出的元素转换成它对的2倍

4、算法练习：移除第一个负数后的所有负数

（1,2,3,4,5，-1，-3，-5，-7）

var foundFirstNegative = false

val keepindexs = for(i <- 0 until a.length if !foundFirstNegative || a(i)>=0)yield{

if(a(i)<0)foundFirstNegative = true

i

}

for(i <- 0 until keepindexs.length){a(i) = a(keepindexs(i))}

a.trimEnd(a.length - keepindexs.length)

### Scala面向对象编程

1、半生类和半生对象

2、Scala中构造方法

-》主构造方法

-》附属构造方法

3、调用对象的时候会调用主构造方法

4、附属构造方法必须去调用主构造方法

5、小结：

1、主构造方法直接跟在类名的后面，主构造方法中的参数最后会被编译成字段或属性

2、主构造方法执行的时候，会执行类中的所有语句

3、如果参数声明的时候不带val或者是var这类修饰符的话，那么相当于是private[this]，只能在内部进行使用

### 高阶函数

函数作为参数传递

val sayHelloFunc = (name: String) => println("Hello , " + name)

def greeting(func:(String) => Unit,name: String) {func(name)}

常用高阶函数方法

Array(1,2,3,4,5).map((num: Int) => num \* num)

函数作为返回值，返回类型可以是函数的函数

def greetingFunc(msg: String) = (name: String) => println(msg + "," + name)

val greetingFuncHello = greetingFunc("Hello")

自动推断类型

def greeting(func:(String) => Unit, name: String) {func(name)}

greeting(name => println("Hello , " + name),"leo")

使用return返回值

def greeting(name: String) = {

def sayHello(name: String): String ={

return "Hello ," + name

}

sayHello(name)

}

class SpecialPerson(val name: String)

class Student(val name : String)

class Older(val name : String)

### 隐式转换函数

implicit def object2Specialerson(obj: Object):SpecialPerson ={

if(obj.getClass == classOf[Older]){

val older = obj.asInstanceOf[Older]

val name = older.name

new SpecialPerson(name)

}else{

new SpecialPerson("None")

}

}

var ticketNumber = 0

def buySpecialTicket(p: SpecialPerson) = {

ticketNumber += 1

"T-" + ticketNumber

}

buySpecialTicket(new Older("older"))

class SignPen {

def write(content: String) = println(content)

}

implicit val signPen = new SignPen

def signForExam(name: String) (implicit signPen: SignPen) {

signPen.write(name + " come to exam in time.")

}

scala> signForExam("leo")

leo come to exam in time.

### Scala中的模式匹配

1、语法：

变量 match{

case 值1 =》 代码1

case 值2 =》 代码2

...

case \_ =》 代码N

}

2、对于值的匹配

def judgeGrade(grade: String): Unit ={

grade match {

case "A" => println("Excellent")

case "B" => println("Good")

case "C" => println("just so so")

case \_ => println("You need word harder")

}

}

judgeGrade("A")

3、对Array的不同情况进行匹配

def greeting(arr:Array[String]): Unit ={

arr match {

case Array("x1") => println("Hello x1 !!!")

case Array(a1,a2,a3) => println(a1 + "," + a2 + "," + a3)

case Array("beifeng",\_\*) => println("Hello beifeng.....")

case \_ => println("===========================")

}

}

greeting(Array("beifeng","x1","x2","x3"))

4、list模式匹配语法：

def greeting2(list: List[String]) {

list match {

case "Leo" :: Nil => println("Hi, Leo!")

case girl1 :: girl2 :: girl3 :: Nil => println( girl1 + " and " + girl2 + " and " + girl3)

case "Leo" :: tail => println("Hi, Leo.")

case \_ => println("hey, who are you?")

}

}

greeting2(List("jack"))

"Hello Spark" foreach(c => println(c match {

case ' ' => "space"

case ch => "Char: " + ch

}))

### Scala中的trait

【一】定义接口

trait SayHello{

def sayHello(name: String)

}

trait MakeFriends{

def makeFriends(p: Person)

}

class Person(val name : String ) extends SayHello with MakeFriends{

def sayHello(otherName: String) = println("Hello, " + otherName + ", I m " + name)

def makeFriends(p: Person) = println("Hello " + p.name + " , I m " + name + ", I want to make friends with you.")

}

【二】定义方法

trait Logger{

def log(msg: String) = println("log : " + msg)

}

class Person(val name : String) extends Logger {

def sayHello{println ("Hello, I' m " + name);log("sayHello is invoked")}

}

【三】定义字段

trait Person{

val eyeNum : Int = 2

}

class Student(val name :String) extends Person{

def sayHello = println("Hi I'm" + name + " , I have " + eyeNum + "eyes.")

}

【四】定义抽象字段

trait SayHello{

val msg : String

def sayHello(name : String) = println(msg + "," + name)

}

class Person(val name : String) extends SayHello{

val msg : String = "hello"

def makeFriends(p: Person){

sayHello(p.name)

println("I'm " + name + " , I want to make friends with you.")

}

}

【五】混入trait

trait Logged {

def log(msg: String){}

}

trait MyLogger extends Logged{

override def log(msg: String){println("log: " + msg)}

}

class Person(val name : String) extends Logged{

def sayHello{println("Hi , I' m " + name);log("sayHello is invoked")}

}

val p3 = new Person("jack") with MyLogger

### Actor多线程编程

1、创建、启动和收发消息

2、收发样例类的消息（case class）

3、创建：

-》导包import scala.actors.Actor

创建：

class HelloActor extends Actor{

def act(){

while (true){

receive{

case name : String => println("Hello , " + name)

}

}

}

}

act等于是一个线程体

while无限循环接收消息

receive关键词接收消息

启动：

创建实例

val helloActor = new HelloActor

启动线程

helloActor.start()

发送消息，使用感叹号

helloActor ! "leo"

收发样例类消息

练习【用户注册登录】

case class Lg(username: String, password: String)

case class Rs(username: String, password: String)

class UserActor extends Actor{

def act(){

while (true){

receive{

case Lg(username,password) => println("login: " + username + "," + password)

case Rs(username,password) => println("register : " + username + "," + password)

}

}

}

}

多个Actor之间通信

练习【两个人打电话】

case class Message(content: String, sender: Actor)

class LeoActor extends Actor{

def act(){

while (true){

receive{

case Message(content,sender)=>{

println("leo: " + content)

sender ! "please call me after 10 minutes."

}

}

}

}

}

class JackActor(val LeoActor: Actor) extends Actor{

def act(){

LeoActor ! Message("Hello , leo, I'm Jack. Are you free Now?",this)

while (true){

receive{

case resphone:String => println("Jack: " + resphone)

}

}

}

}

# Spark基础

## Spark安装

[Spark安装](参考文件/Spark安装.docx)

## Spark使用

Spark Github地址

<https://github.com/apache/spark>

### 解决shell无法加载外部jar包

新建一个目录存放外部jar包

启动shell命令

spark-shell –jars /path/jar1,/path/jar2

(多个jar包用逗号分开)

### Spark实现wordcount

val linesRdd = sc.textFile("hdfs://mydream:9000/user/zkj/input/test.txt")

val wordsRdd = linesRdd.flatMap(line => line.split(" "))

val keyvalRdds = wordsRdd.map(word => (word,1))

val countRdd = keyvalRdds.reduceByKey((a,b) => (a + b))

countRdd.collect

OR

sc.textFile("hdfs://mydream:9000/user/zkj/input/test.txt")

.flatMap(\_.split(" ")).map((\_,1)).reduceByKey(\_ + \_).collect

### Spark案例

案例1：WordCount对结果排序

val rdd = sc.textFile("hdfs://mydream:9000/user/zkj/input/test.txt")

val wordcount = rdd.flatMap(\_.split(" ")).map((\_,1)).reduceByKey(\_ + \_)

wordcount.collect

wordcount.saveAsTextFile("hdfs://mydream:9000/user/zkj/output/outputtest.txt")

======result

(hive,2)

(mapreduce,1)

(mapreduce2,1)

(hadoop,5)

(hdfs,1)

发现：

Spark 运行WordCoun程序，并没有像MapReduce程序那样，对Key进行排序。

## Key Sort

wordcount.sortByKey().collect ## 默认情况是 升序

wordcount.sortByKey(true).collect ##与默认情况相同

wordcount.sortByKey(false).collect

========================== result==================================

(hive,2)

(mapreduce,1)

(mapreduce2,1)

(hadoop,5)

(hdfs,1)

需求：

按照value值进行降序

## Value Sort

##将key和value倒过来排序再倒回去

wordcount.map(x => (x.\_2,x.\_1)).sortByKey(false).collect

wordcount.map(x => (x.\_2,x.\_1)).sortByKey(false).map(x => (x.\_2,x.\_1)).collect

##取出Top N

wordcount.map(x => (x.\_2,x.\_1)).sortByKey(false).map(x => (x.\_2,x.\_1)).take(3)

[java示例代码](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/SortWordCount.java)

[scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/wordcount/core/SortWordCount.scala)

+++++++++++++++++++++++++++++++++++++++++++

Group Top Key

WordCount 程序，前KEY值

需求：

类似MapReduce中的二次排序

1) 按照第一个字段进行分组

2) 对分组中的第二字段进行排序(降序)

3) 获取每个分组Top N，比如获取前三个值

输入文件内容：

aa 78

bb 98

aa 80

cc 98

aa 69

cc 87

bb 97

cc 86

aa 97

bb 78

bb 34

cc 85

bb 92

cc 72

bb 32

bb 23

功能分析：

(aa,list(78,80,69,97)) -> (aa,list(69,78,80,97)) -> (aa,list(69,78,80))

val rdd = sc.textFile("hdfs://hadoop-senior.ibeifeng.com:8020/user/beifeng/spark/grouptop/input/score.input")

rdd.map(\_.split(" ")).collect

结果：Array(aa, 78)

rdd.map(\_.split(" ")).map(x => (x(0),x(1))).collect

结果：(aa,78)

rdd.map(\_.split(" ")).map(x => (x(0),x(1))).groupByKey.collect

结果：(aa,CompactBuffer(78, 80, 69, 97))

rdd.map(\_.split(" ")).map(x => (x(0),x(1))).groupByKey.map(

x => {

val xx = x.\_1

val yy = x.\_2

yy ##返回值，获取value值，即CompactBuffer(78, 80, 69, 97)

}

).collect

Iterable[String]

Iterable 方法：

def toList: List[A]

返回包含此遍历的迭代器的所有元素的列表

rdd.map(\_.split(" ")).map(x => (x(0),x(1))).groupByKey.map(

x => {

val xx = x.\_1

val yy = x.\_2

yy.toList ##转化为list

}

).collect

结果：List[String]

List(78, 80, 69, 97)

List 方法：

def sorted[B >: A]: List[A]

根据排序对列表进行排序

rdd.map(\_.split(" ")).map(x => (x(0),x(1))).groupByKey.map(

x => {

val xx = x.\_1

val yy = x.\_2

yy.toList.sorted #排序，默认升序

}

).collect

List[String]

List(69, 78, 80, 97)

List 方法：

def reverse: List[A]

返回新列表，在相反的顺序元素

rdd.map(\_.split(" ")).map(x => (x(0),x(1))).groupByKey.map(

x => {

val xx = x.\_1

val yy = x.\_2

yy.toList.sorted.reverse

}

).collect

结果：List[String]

List(97, 80, 78, 69)

List 方法：

def take(n: Int): List[A]

返回前n个元素

def takeRight(n: Int): List[A]

返回最后n个元素

rdd.map(\_.split(" ")).map(x => (x(0),x(1))).groupByKey.map(

x => {

val xx = x.\_1

val yy = x.\_2

yy.toList.sorted.reverse.take(3)

}

).collect

要求返回的是一个元组对

rdd.map(\_.split(" ")).map(x => (x(0),x(1))).groupByKey.map(

x => {

val xx = x.\_1

val yy = x.\_2

(xx,yy.toList.sorted.reverse.take(3)) ##返回元组对

}

).collect

最终：

val groupTopKeyRdd = rdd.map(\_.split(" ")).map(x => (x(0),x(1))).groupByKey.map(

x => {

val xx = x.\_1

val yy = x.\_2

(xx,yy.toList.sorted.reverse.take(3))

}

)

存储结果：

groupTopKeyRdd.saveAsTextFile("hdfs://hadoop-senior.ibeifeng.com:8020/user/beifeng/spark/grouptop/output")

[java示例代码](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/GroupTop3.java)

案例2：开发一个scala程序

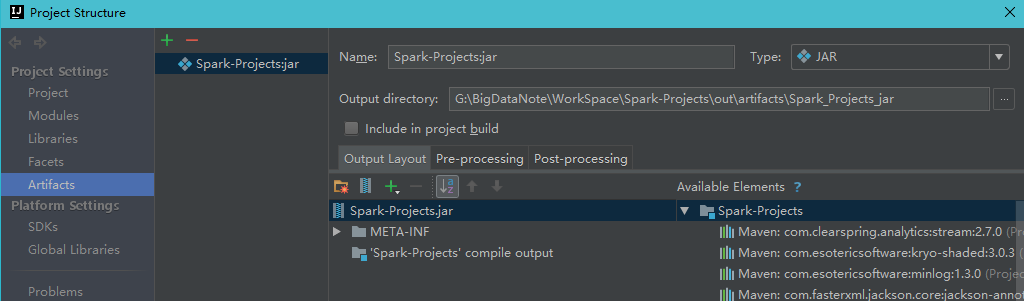
[示例代码](WorkSpace/Spark-Projects/src/main/scala/spark/app)

打包上传：

新建Artifact



去掉依赖jar包：



打包

Build—>Build Artifact

上传jar包并运行

spark-submit --master spark://mydream:7077 Spark-Projects.jar

### Spark Application运行的两种方式

客户端模式

spark-submit \

--master spark://mydream:7077 \

--deploy-mode client \

Spark-Projects.jar

Cluster模式

spark-submit \

--master spark://mydream:7077 \

--deploy-mode cluster \

Spark-Projects.jar

### 运行在yarn上

spark-shell --master yarn-client

## Spark Stream

### Spark Stream Demo

1. 检查是否安装nc

rpm -qa|grep nc

1. [没有需安装](参考文件/nc-rpm)

rpm -ivh nc-1.84-24.el6.x86\_64.rpm

1. 启动端口

nc -lk 9999

1. 开启hadoop&spark

start-dfs.sh start-master.sh

1. 运行spark命令

run-example streaming.NetworkWordCount mydream 9999

1. nc客户端输入数据

[自己编写代码](WorkSpace/Spark-Projects/src/main/scala/spark/app/StreamingWordCountDemo.scala)

### Spark Stream与Flume集成

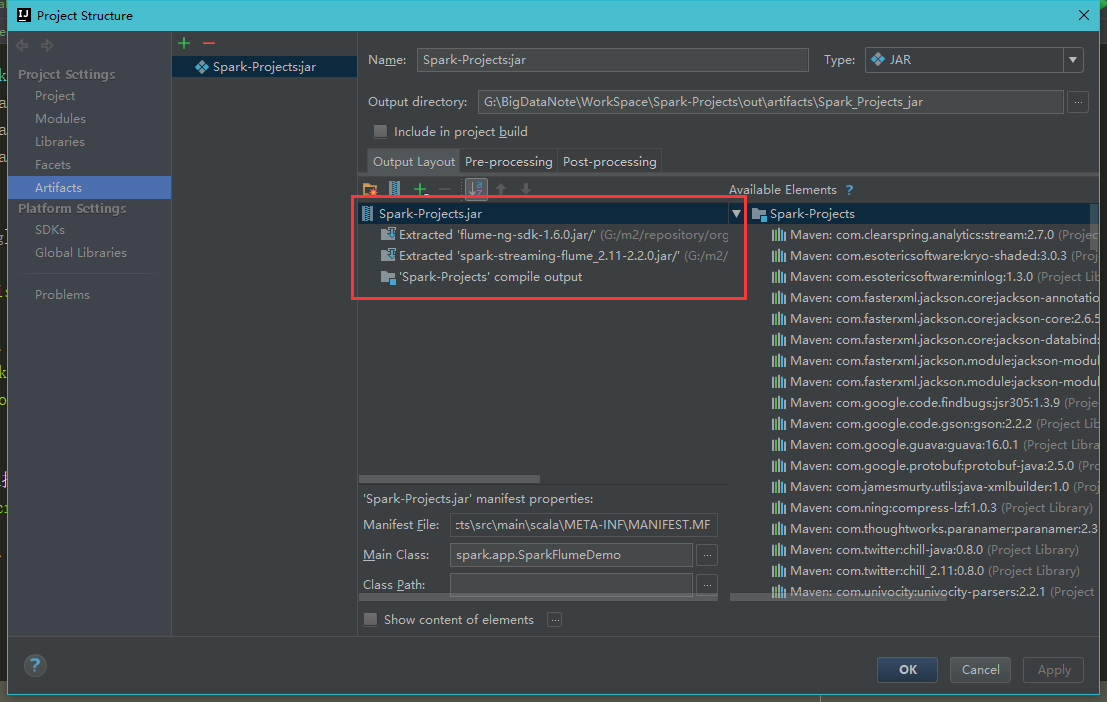
编写flume配置文件监控某文件

[配置文件](参考文件/flume脚本/spark-flume.conf)

编写scala源码监控flume发送的端口

[源码](WorkSpace/Spark-Projects/src/main/scala/spark/app/SparkFlumeDemo.scala)

打包jar包并上传



启动spark程序

spark-submit --master spark://mydream:7077 --deploy-mode client Spark-Projects.jar

启动flume

flume-ng agent -c conf -n a1 -f ./spark-flume.conf -Dflume.root.logger=DEBUG,console

### Spark与Kafka集成

启动kafka

./bin/kafka-server-start.sh config/server.properties

创建主题

kafka-topics.sh --create --zookeeper MyDream:2181 --replication-factor 1 --partitions 1 --topic zkjTopic

启动生产者

kafka-console-producer.sh --broker-list MyDream:9092 --topic zkjTopic

启动消费者

打包[jar包](WorkSpace/Spark-Projects/src/main/scala/spark/app/SparkKafkaDemo.scala)

上传启动服务

spark-submit --master spark://mydream:7077 --deploy-mode client Spark-Projects.jar

### java开发Wordcount

本地运行

[示例代码](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/WordCountLocal.java)

提交集群运行

[示例代码](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/WordCountCluster.java)

提交脚本：

/usr/local/spark/bin/spark-submit \

--class cn.spark.sparktest.core.WordCountCluster \

--num-executors 3 \

--driver-memory 100m \

--executor-memory 100m \

--executor-cores 3 \

/usr/local/SparkTest-0.0.1-SNAPSHOT-jar-with-dependencies.jar \

### wordcount程序原理深度剖析

[原理图](参考文件/原理图/wordcount程序原理深度剖析.png)

### Spark架构原理

[原理图](参考文件/原理图/Spark架构原理.png)

### 创建RDD

#### 并行化集合创建RDD

[java创建](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/rdd/ParallelizeCollection.java)

[scala创建](WorkSpace/Spark-Projects/src/main/scala/spark1/wordcount/core/rdd/ParallelizeCollection.scala)

#### 使用本地文件和HDFS创建RDD

NOTE：

1、如果是针对本地文件的话，如果是在windows上本地测试，windows上有一份文件即可；如果是在spark集群上针对linux本地文件，那么需要将文件拷贝到所有worker节点上。

2、Spark的textFile()方法支持针对目录、压缩文件以及通配符进行RDD创建。

3、Spark默认会为hdfs文件的每一个block创建一个partition，但是也可以通过textFile()的第二个参数手动设置分区数量，只能比block数量多，不能比block数量少。

[本地文件java创建](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/rdd/HDFSFile.java)

[本地文件scala创建](WorkSpace/Spark-Projects/src/main/scala/spark1/wordcount/core/rdd/LocalFile.scala)

[HDFS文件java创建](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/rdd/HDFSFile.java)

[HDFS文件scala创建](WorkSpace/Spark-Projects/src/main/scala/spark1/wordcount/core/rdd/HDFSFile.scala)

Spark的textFile()除了可以针对上述几种普通的文件创建RDD之外，还有一些特列的方法来创建RDD：

1、SparkContext.wholeTextFiles()方法，可以针对一个目录中的大量小文件，返回<filename, fileContent>组成的pair，作为一个PairRDD，而不是普通的RDD。普通的textFile()返回的RDD中，每个元素就是文件中的一行文本。

2、SparkContext.sequenceFile[K, V]()方法，可以针对SequenceFile创建RDD，K和V泛型类型就是SequenceFile的key和value的类型。K和V要求必须是Hadoop的序列化类型，比如IntWritable、Text等。

3、SparkContext.hadoopRDD()方法，对于Hadoop的自定义输入类型，可以创建RDD。该方法接收JobConf、InputFormatClass、Key和Value的Class。

4、SparkContext.objectFile()方法，可以针对之前调用RDD.saveAsObjectFile()创建的对象序列化的文件，反序列化文件中的数据，并创建一个RDD。

### transformation操作开发实战

**常用transformation介绍**

|  |  |
| --- | --- |
| **操作** | **介绍** |
| map | 将RDD中的每个元素传入自定义函数，获取一个新的元素，然后用新的元素组成新的RDD |
| filter | 对RDD中每个元素进行判断，如果返回true则保留，返回false则剔除。 |
| flatMap | 与map类似，但是对每个元素都可以返回一个或多个新元素。 |
| gropuByKey | 根据key进行分组，每个key对应一个Iterable<value> |
| reduceByKey | 对每个key对应的value进行reduce操作。 |
| sortByKey | 对每个key对应的value进行排序操作。 |
| join | 对两个包含<key,value>对的RDD进行join操作，每个key join上的pair，都会传入自定义函数进行处理。 |
| cogroup | 同join，但是是每个key对应的Iterable<value>都会传入自定义函数进行处理。 |

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/TransformationOperation.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/wordcount/core/TransformationOperation.scala)

### action操作开发实战

**常用action介绍**

|  |  |
| --- | --- |
| **操作** | **介绍** |
| reduce | 将RDD中的所有元素进行聚合操作。第一个和第二个元素聚合，值与第三个元素聚合，值与第四个元素聚合，以此类推。 |
| collect | 将RDD中所有元素获取到本地客户端。 |
| count | 获取RDD元素总数。 |
| take(n) | 获取RDD中前n个元素。 |
| saveAsTextFile | 将RDD元素保存到文件中，对每个元素调用toString方法 |
| countByKey | 对每个key对应的值进行count计数。 |
| foreach | 遍历RDD中的每个元素。 |

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/ActionOperation.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/wordcount/core/ActionOperation.scala)

### RDD持久化详解

**RDD持久化策略**

|  |  |
| --- | --- |
| **持久化级别** | **含义** |
| MEMORY\_ONLY | 以非序列化的Java对象的方式持久化在JVM内存中。如果内存无法完全存储RDD所有的partition，那么那些没有持久化的partition就会在下一次需要使用它的时候，重新被计算。 |
| MEMORY\_AND\_DISK | 同上，但是当某些partition无法存储在内存中时，会持久化到磁盘中。下次需要使用这些partition时，需要从磁盘上读取。 |
| MEMORY\_ONLY\_SER | 同MEMORY\_ONLY，但是会使用Java序列化方式，将Java对象序列化后进行持久化。可以减少内存开销，但是需要进行反序列化，因此会加大CPU开销。 |
| MEMORY\_AND\_DSK\_SER | 同MEMORY\_AND\_DSK。但是使用序列化方式持久化Java对象。 |
| DISK\_ONLY | 使用非序列化Java对象的方式持久化，完全存储到磁盘上。 |
| MEMORY\_ONLY\_2  MEMORY\_AND\_DISK\_2  等等 | 如果是尾部加了2的持久化级别，表示会将持久化数据复用一份，保存到其他节点，从而在数据丢失时，不需要再次计算，只需要使用备份数据即可。 |

Spark提供的多种持久化级别，主要是为了在CPU和内存消耗之间进行取舍。下面是一些通用的持久化级别的选择建议：

1、优先使用MEMORY\_ONLY，如果可以缓存所有数据的话，那么就使用这种策略。因为纯内存速度最快，而且没有序列化，不需要消耗CPU进行反序列化操作。

2、如果MEMORY\_ONLY策略，无法存储的下所有数据的话，那么使用MEMORY\_ONLY\_SER，将数据进行序列化进行存储，纯内存操作还是非常快，只是要消耗CPU进行反序列化。

3、如果需要进行快速的失败恢复，那么就选择带后缀为\_2的策略，进行数据的备份，这样在失败时，就不需要重新计算了。

4、能不使用DISK相关的策略，就不用使用，有的时候，从磁盘读取数据，还不如重新计算一次。

[RDD持久化java示例代码](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/rdd/Persist.java)

### 共享变量（Broadcast Variable和Accumulator）

[共享变量的工作原理](参考文件/原理图/共享变量的工作原理.png)

默认情况下，如果在一个算子的函数中使用到了某个外部的变量，那么这个变量的值会被拷贝到每个task中。此时每个task只能操作自己的那份变量副本。如果多个task想要共享某个变量，那么这种方式是做不到的。

Spark为此提供了两种共享变量，一种是Broadcast Variable（广播变量），另一种是Accumulator（累加变量）。Broadcast Variable会将使用到的变量，仅仅为每个节点拷贝一份，更大的用处是优化性能，减少网络传输以及内存消耗。Accumulator则可以让多个task共同操作一份变量，主要可以进行累加操作。

#### Broadcast Variable

Spark提供的Broadcast Variable，是只读的。并且在每个节点上只会有一份副本，而不会为每个task都拷贝一份副本。因此其最大作用，就是减少变量到各个节点的网络传输消耗，以及在各个节点上的内存消耗。此外，spark自己内部也使用了高效的广播算法来减少网络消耗。

可以通过调用SparkContext的broadcast()方法，来针对某个变量创建广播变量。然后在算子的函数内，使用到广播变量时，每个节点只会拷贝一份副本了。每个节点可以使用广播变量的value()方法获取值。记住，广播变量，是只读的。

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/BroadcastVariable.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/wordcount/core/BroadcastVariable.scala)

#### Accumulator

Spark提供的Accumulator，主要用于多个节点对一个变量进行共享性的操作。Accumulator只提供了累加的功能。但是确给我们提供了多个task对一个变量并行操作的功能。但是task只能对Accumulator进行累加操作，不能读取它的值。只有Driver程序可以读取Accumulator的值。

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/AccumulatorVariable.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/wordcount/core/AccumulatorVariable.scala)

### 二次排序

\* 1、实现自定义的key，要实现Ordered接口和Serializable接口，在key中实现自己对多个列的排序算法

首先在自定义key里面，定义需要进行排序的列

为要进行排序的多个列，提供getter和setter方法，以及hashcode和equals方法，compare和compareTo方法

以及四个比较方法

\* 2、将包含文本的RDD，映射成key为自定义key，value为文本的JavaPairRDD

\* 3、使用sortByKey算子按照自定义的key进行排序

\* 4、再次映射，剔除自定义的key，只保留文本行

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark1/wordcount/core/SecondarySort.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/wordcount/core/SecondSort.scala)

## Spark内核架构

1、Application

2、spark-submit

3、Driver

4、SparkContext

5、Master

6、Worker

7、Executor

8、Job

9、DAGScheduler

10、TaskScheduler

11、ShuffleMapTask and ResultTask

### Spark的三种提交模式

1、Spark内核架构，其实就是第一种模式，standalone模式，基于Spark自己的Master-Worker集群。

2、第二种，是基于YARN的yarn-cluster模式。

3、第三种，是基于YARN的yarn-client模式。

4、如果，你要切换到第二种和第三种模式，很简单，将我们之前用于提交spark应用程序的spark-submit脚本，加上--master参数，设置为yarn-cluster，或yarn-client，即可。如果你没设置，那么，就是standalone模式。

### job触发流程原理

val lines = sc.textFile()

val words = lines.flatMap(line => line.split(" "))

val pairs = words.map(word => (word, 1))

// 其实RDD里是没有reduceByKey的，因此对RDD调用reduceByKey()方法的时候，会触发scala的隐式转换；此时就会在作用域内，寻找隐式转换，会在RDD中找到rddToPairRDDFunctions()隐式转换，然后将RDD转换为PairRDDFunctions。

// 接着会调用PairRDDFunctions中的reduceByKey()方法

val counts = pairs.reduceByKey(\_ + \_)

counts.foreach(count => println(count.\_1 + ": " + count.\_2))

### Checkpoint原理

checkpoint功能是什么意思？checkpoint就是说，对于一个复杂的RDD chain，我们如果担心中间某些关键的，在后面会反复几次使用的RDD，可能会因为节点的故障，导致持久化数据的丢失，那么就可以针对该RDD格外启动checkpoint机制，实现容错和高可用。

首先呢，要调用SparkContext的setCheckpointDir()方法，设置一个容错的文件系统的目录，比如说HDFS；然后，对RDD调用调用checkpoint()方法。之后，在RDD所处的job运行结束之后，会启动一个单独的job，来将checkpoint过的RDD的数据写入之前设置的文件系统，进行高可用、容错的类持久化操作。

那么此时，即使在后面使用RDD时，它的持久化的数据，不小心丢失了，但是还是可以从它的checkpoint文件中直接读取其数据，而不需要重新计算。（CacheManager）

## Spark源码剖析

[Spark原理图](参考文件/原理图/Spark源码剖析原理图)

## Spark性能优化技术

[性能优化原理图](参考文件/原理图/Spark性能优化)

Spark的性能优化，主要手段包括：

1、使用高性能序列化类库

2、优化数据结构

3、对多次使用的RDD进行持久化 / Checkpoint

4、使用序列化的持久化级别

5、Java虚拟机垃圾回收调优

6、提高并行度

7、广播共享数据

8、数据本地化

9、reduceByKey和groupByKey的合理使用

10、Shuffle调优（核心中的核心，重中之重）

### 内存都花费在哪里了？

1、每个Java对象，都有一个对象头，会占用16个字节，主要是包括了一些对象的元信息，比如指向它的类的指针。如果一个对象本身很小，比如就包括了一个int类型的field，那么它的对象头实际上比对象自己还要大。

2、Java的String对象，会比它内部的原始数据，要多出40个字节。因为它内部使用char数组来保存内部的字符序列的，并且还得保存诸如数组长度之类的信息。而且因为String使用的是UTF-16编码，所以每个字符会占用2个字节。比如，包含10个字符的String，会占用60个字节。

3、Java中的集合类型，比如HashMap和LinkedList，内部使用的是链表数据结构，所以对链表中的每一个数据，都使用了Entry对象来包装。Entry对象不光有对象头，还有指向下一个Entry的指针，通常占用8个字节。

4、元素类型为原始数据类型（比如int）的集合，内部通常会使用原始数据类型的包装类型，比如Integer，来存储元素。

### 如何判断你的程序消耗了多少内存？

这里有一个非常简单的办法来判断，你的spark程序消耗了多少内存。

1、首先，自己设置RDD的并行度，有两种方式：要不然，在parallelize()、textFile()等方法中，传入第二个参数，设置RDD的task / partition的数量；要不然，用SparkConf.set()方法，设置一个参数，spark.default.parallelism，可以统一设置这个application所有RDD的partition数量。

2、其次，在程序中将RDD cache到内存中，调用RDD.cache()方法即可。

3、最后，观察Driver的log，你会发现类似于：“INFO BlockManagerMasterActor: Added rdd\_0\_1 in memory on mbk.local:50311 (size: 717.5 KB, free: 332.3 MB)”的日志信息。这就显示了每个partition占用了多少内存。

4、将这个内存信息乘以partition数量，即可得出RDD的内存占用量。

### 高性能序列化类库

Spark自身默认就会在一些地方对数据进行序列化，比如Shuffle。还有就是，如果我们的算子函数使用到了外部的数据（比如Java内置类型，或者自定义类型），那么也需要让其可序列化。

而Spark自身对于序列化的便捷性和性能进行了一个取舍和权衡。默认，Spark倾向于序列化的便捷性，使用了Java自身提供的序列化机制——基于ObjectInputStream和ObjectOutputStream的序列化机制。因为这种方式是Java原生提供的，很方便使用

Spark实际上提供了两种序列化机制，它只是默认使用了第一种：

1、Java序列化机制：默认情况下，Spark使用Java自身的ObjectInputStream和ObjectOutputStream机制进行对象的序列化。只要你的类实现了Serializable接口，那么都是可以序列化的。而且Java序列化机制是提供了自定义序列化支持的，只要你实现Externalizable接口即可实现自己的更高性能的序列化算法。Java序列化机制的速度比较慢，而且序列化后的数据占用的内存空间比较大。

2、Kryo序列化机制：Spark也支持使用Kryo类库来进行序列化。Kryo序列化机制比Java序列化机制更快，而且序列化后的数据占用的空间更小，通常比Java序列化的数据占用的空间要小10倍。Kryo序列化机制之所以不是默认序列化机制的原因是，有些类型虽然实现了Seriralizable接口，但是它也不一定能够进行序列化；此外，如果你要得到最佳的性能，Kryo还要求你在Spark应用程序中，对所有你需要序列化的类型都进行注册。

如果要使用Kryo序列化机制，首先要用SparkConf设置一个参数，使用new SparkConf().set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")即可，即将Spark的序列化器设置为KryoSerializer。这样，Spark在内部的一些操作，比如Shuffle，进行序列化时，就会使用Kryo类库进行高性能、快速、更低内存占用量的序列化了。

使用Kryo时，它要求是需要序列化的类，是要预先进行注册的，以获得最佳性能——如果不注册的话，那么Kryo必须时刻保存类型的全限定名，反而占用不少内存。Spark默认是对Scala中常用的类型自动注册了Kryo的，都在AllScalaRegistry类中。

但是，比如自己的算子中，使用了外部的自定义类型的对象，那么还是需要将其进行注册。

（实际上，下面的写法是错误的，因为counter不是共享的，所以累加的功能是无法实现的）

val counter = new Counter();

val numbers = sc.parallelize(Array(1, 2, 3, 4, 5))

numbers.foreach(num => counter.add(num));

如果要注册自定义的类型，那么就使用如下的代码，即可：

Scala版本：

val conf = new SparkConf().setMaster(...).setAppName(...)

conf.registerKryoClasses(Array(classOf[Counter] ))

val sc = new SparkContext(conf)

Java版本：

SparkConf conf = new SparkConf().setMaster(...).setAppName(...)

conf.registerKryoClasses(Counter.class)

JavaSparkContext sc = new JavaSparkContext(conf)

#### 优化Kryo类库的使用

1、优化缓存大小

如果注册的要序列化的自定义的类型，本身特别大，比如包含了超过100个field。那么就会导致要序列化的对象过大。此时就需要对Kryo本身进行优化。因为Kryo内部的缓存可能不够存放那么大的class对象。此时就需要调用SparkConf.set()方法，设置spark.kryoserializer.buffer.mb参数的值，将其调大。

默认情况下它的值是2，就是说最大能缓存2M的对象，然后进行序列化。可以在必要时将其调大。比如设置为10。

2、预先注册自定义类型

虽然不注册自定义类型，Kryo类库也能正常工作，但是那样的话，对于它要序列化的每个对象，都会保存一份它的全限定类名。此时反而会耗费大量内存。因此通常都建议预先注册号要序列化的自定义的类。

#### 在什么场景下使用Kryo序列化类库？

那么，这里针对的Kryo序列化类库的使用场景，就是算子函数使用到了外部的大数据的情况。比如说吧，我们在外部定义了一个封装了应用所有配置的对象，比如自定义了一个MyConfiguration对象，里面包含了100m的数据。然后，在算子函数里面，使用到了这个外部的大对象。

此时呢，如果默认情况下，让Spark用java序列化机制来序列化这种外部的大对象，那么就会导致，序列化速度缓慢，并且序列化以后的数据还是比较大，比较占用内存空间。

因此，在这种情况下，比较适合，切换到Kryo序列化类库，来对外部的大对象进行序列化操作。一是，序列化速度会变快；二是，会减少序列化后的数据占用的内存空间。

### 优化数据结构

要减少内存的消耗，除了使用高效的序列化类库以外，还有一个很重要的事情，就是优化数据结构。从而避免Java语法特性中所导致的额外内存的开销，比如基于指针的Java数据结构，以及包装类型。

有一个关键的问题，就是优化什么数据结构？其实主要就是优化你的算子函数，内部使用到的局部数据，或者是算子函数外部的数据。都可以进行数据结构的优化。优化之后，都会减少其对内存的消耗和占用。

#### 如何优化数据结构（一）

1、优先使用数组以及字符串，而不是集合类。也就是说，优先用array，而不是ArrayList、LinkedList、HashMap等集合。

比如，有个List<Integer> list = new ArrayList<Integer>()，将其替换为int[] arr = new int[]。这样的话，array既比List少了额外信息的存储开销，还能使用原始数据类型（int）来存储数据，比List中用Integer这种包装类型存储数据，要节省内存的多。

还比如，通常企业级应用中的做法是，对于HashMap、List这种数据，统一用String拼接成特殊格式的字符串，比如Map<Integer, Person> persons = new HashMap<Integer, Person>()。可以优化为，特殊的字符串格式：id:name,address|id:name,address...。

#### 如何优化数据结构（二）

2、避免使用多层嵌套的对象结构。比如说，public class Teacher { private List<Student> students = new ArrayList<Student>() }。就是非常不好的例子。因为Teacher类的内部又嵌套了大量的小Student对象。

比如说，对于上述例子，也完全可以使用特殊的字符串来进行数据的存储。比如，用json字符串来存储数据，就是一个很好的选择。

{"teacherId": 1, "teacherName": "leo", students:[{"studentId": 1, "studentName": "tom"},{"studentId":2, "studentName":"marry"}]}

3、对于有些能够避免的场景，尽量使用int替代String。因为String虽然比ArrayList、HashMap等数据结构高效多了，占用内存量少多了，但是之前分析过，还是有额外信息的消耗。比如之前用String表示id，那么现在完全可以用数字类型的int，来进行替代。

这里提醒，在spark应用中，id就不要用常用的uuid了，因为无法转成int，就用自增的int类型的id即可。（sdfsdfdf-234242342-sdfsfsfdfd）

### 使用序列化的持久化级别

除了对多次使用的RDD进行持久化操作之外，还可以进一步优化其性能。因为很有可能，RDD的数据是持久化到内存，或者磁盘中的。那么，此时，如果内存大小不是特别充足，完全可以使用序列化的持久化级别，比如MEMORY\_ONLY\_SER、MEMORY\_AND\_DISK\_SER等。使用RDD.persist(StorageLevel.MEMORY\_ONLY\_SER)这样的语法即可。

这样的话，将数据序列化之后，再持久化，可以大大减小对内存的消耗。此外，数据量小了之后，如果要写入磁盘，那么磁盘io性能消耗也比较小。

对RDD持久化序列化后，RDD的每个partition的数据，都是序列化为一个巨大的字节数组。这样，对于内存的消耗就小的多了。但是唯一的缺点就是，获取RDD数据时，需要对其进行反序列化，会增大其性能开销。

因此，对于序列化的持久化级别，还可以进一步优化，也就是说，使用Kryo序列化类库，这样，可以获得更快的序列化速度，并且占用更小的内存空间。但是要记住，如果RDD的元素（RDD<T>的泛型类型），是自定义类型的话，在Kryo中提前注册自定义类型。

### Java虚拟机垃圾回收调优

#### 监测垃圾回收

我们可以对垃圾回收进行监测，包括多久进行一次垃圾回收，以及每次垃圾回收耗费的时间。只要在spark-submit脚本中，增加一个配置即可，--conf "spark.executor.extraJavaOptions=-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps"。

但是要记住，这里虽然会打印出Java虚拟机的垃圾回收的相关信息，但是是输出到了worker上的日志中，而不是driver的日志中。

但是这种方式也只是一种，其实也完全可以通过SparkUI（4040端口）来观察每个stage的垃圾回收的情况。

#### 优化executor内存比例

对于垃圾回收来说，最重要的就是调节RDD缓存占用的内存空间，与算子执行时创建的对象占用的内存空间的比例。默认情况下，Spark使用每个executor 60%的内存空间来缓存RDD，那么在task执行期间创建的对象，只有40%的内存空间来存放。

在这种情况下，很有可能因为你的内存空间的不足，task创建的对象过大，那么一旦发现40%的内存空间不够用了，就会触发Java虚拟机的垃圾回收操作。因此在极端情况下，垃圾回收操作可能会被频繁触发。

在上述情况下，如果发现垃圾回收频繁发生。那么就需要对那个比例进行调优，使用new SparkConf().set("spark.storage.memoryFraction", "0.5")即可，可以将RDD缓存占用空间的比例降低，从而给更多的空间让task创建的对象进行使用。

因此，对于RDD持久化，完全可以使用Kryo序列化，加上降低其executor内存占比的方式，来减少其内存消耗。给task提供更多的内存，从而避免task的执行频繁触发垃圾回收。

#### 高级垃圾回收调优（一）

Java堆空间被划分成了两块空间，一个是年轻代，一个是老年代。年轻代放的是短时间存活的对象，老年代放的是长时间存活的对象。年轻代又被划分了三块空间，Eden、Survivor1、Survivor2。

首先，Eden区域和Survivor1区域用于存放对象，Survivor2区域备用。创建的对象，首先放入Eden区域和Survivor1区域，如果Eden区域满了，那么就会触发一次Minor GC，进行年轻代的垃圾回收。Eden和Survivor1区域中存活的对象，会被移动到Survivor2区域中。然后Survivor1和Survivor2的角色调换，Survivor1变成了备用。

如果一个对象，在年轻代中，撑过了多次垃圾回收，都没有被回收掉，那么会被认为是长时间存活的，此时就会被移入老年代。此外，如果在将Eden和Survivor1中的存活对象，尝试放入Survivor2中时，发现Survivor2放满了，那么会直接放入老年代。此时就出现了，短时间存活的对象，进入老年代的问题。

如果老年代的空间满了，那么就会触发Full GC，进行老年代的垃圾回收操作。

#### 高级垃圾回收调优（二）

Spark中，垃圾回收调优的目标就是，只有真正长时间存活的对象，才能进入老年代，短时间存活的对象，只能呆在年轻代。不能因为某个Survivor区域空间不够，在Minor GC时，就进入了老年代。从而造成短时间存活的对象，长期呆在老年代中占据了空间，而且Full GC时要回收大量的短时间存活的对象，导致Full GC速度缓慢。

如果发现，在task执行期间，大量full gc发生了，那么说明，年轻代的Eden区域，给的空间不够大。此时可以执行一些操作来优化垃圾回收行为：

1、包括降低spark.storage.memoryFraction的比例，给年轻代更多的空间，来存放短时间存活的对象；

2、给Eden区域分配更大的空间，使用-Xmn即可，通常建议给Eden区域，预计大小的4/3；

3、如果使用的是HDFS文件，那么很好估计Eden区域大小，如果每个executor有4个task，然后每个hdfs压缩块解压缩后大小是3倍，此外每个hdfs块的大小是64M，那么Eden区域的预计大小就是：4 \* 3 \* 64MB，然后呢，再通过-Xmn参数，将Eden区域大小设置为4 \* 3 \* 64 \* 4/3。

#### 一些高级的参数

-XX:SurvivorRatio=4：如果值为4，那么就是两个Survivor跟Eden的比例是2:4，也就是说每个Survivor占据的年轻代的比例是1/6，所以，你其实也可以尝试调大Survivor区域的大小。

-XX:NewRatio=4：调节新生代和老年代的比例

### 提高并行度

实际上Spark集群的资源并不一定会被充分利用到，所以要尽量设置合理的并行度，来充分地利用集群的资源。才能充分提高Spark应用程序的性能。

Spark会自动设置以文件作为输入源的RDD的并行度，依据其大小，比如HDFS，就会给每一个block创建一个partition，也依据这个设置并行度。对于reduceByKey等会发生shuffle的操作，就使用并行度最大的父RDD的并行度即可。

可以手动使用textFile()、parallelize()等方法的第二个参数来设置并行度；也可以使用spark.default.parallelism参数，来设置统一的并行度。Spark官方的推荐是，给集群中的每个cpu core设置2~3个task。

比如说，spark-submit设置了executor数量是10个，每个executor要求分配2个core，那么application总共会有20个core。此时可以设置new SparkConf().set("spark.default.parallelism", "60")来设置合理的并行度，从而充分利用资源。

### 广播共享数据

如果你的算子函数中，使用到了特别大的数据，那么，这个时候，推荐将该数据进行广播。这样的话，就不至于将一个大数据拷贝到每一个task上去。而是给每个节点拷贝一份，然后节点上的task共享该数据。

这样的话，就可以减少大数据在节点上的内存消耗。并且可以减少数据到节点的网络传输消耗。

### 数据本地化

#### 数据本地化背景

数据本地化对于Spark Job性能有着巨大的影响。如果数据以及要计算它的代码是在一起的，那么性能当然会非常高。但是，如果数据和计算它的代码是分开的，那么其中之一必须到另外一方的机器上。通常来说，移动代码到其他节点，会比移动数据到代码所在的节点上去，速度要快得多，因为代码比较小。Spark也正是基于这个数据本地化的原则来构建task调度算法的。

数据本地化，指的是，数据离计算它的代码有多近。基于数据距离代码的距离，有几种数据本地化级别：

1、PROCESS\_LOCAL：数据和计算它的代码在同一个JVM进程中。

2、NODE\_LOCAL：数据和计算它的代码在一个节点上，但是不在一个进程中，比如在不同的executor进程中，或者是数据在HDFS文件的block中。

3、NO\_PREF：数据从哪里过来，性能都是一样的。

4、RACK\_LOCAL：数据和计算它的代码在一个机架上。

5、ANY：数据可能在任意地方，比如其他网络环境内，或者其他机架上。

#### 数据本地化优化

Spark倾向于使用最好的本地化级别来调度task，但是这是不可能的。如果没有任何未处理的数据在空闲的executor上，那么Spark就会放低本地化级别。这时有两个选择：第一，等待，直到executor上的cpu释放出来，那么就分配task过去；第二，立即在任意一个executor上启动一个task。

Spark默认会等待一会儿，来期望task要处理的数据所在的节点上的executor空闲出一个cpu，从而将task分配过去。只要超过了时间，那么Spark就会将task分配到其他任意一个空闲的executor上。

可以设置参数，spark.locality系列参数，来调节Spark等待task可以进行数据本地化的时间。spark.locality.wait（3000毫秒）、spark.locality.wait.node、spark.locality.wait.process、spark.locality.wait.rack。

### reduceByKey和groupByKey

val counts = pairs.reduceByKey(\_ + \_)

val counts = pairs.groupByKey().map(wordCounts => (wordCounts.\_1, wordCounts.\_2.sum))

如果能用reduceByKey，那就用reduceByKey，因为它会在map端，先进行本地combine，可以大大减少要传输到reduce端的数据量，减小网络传输的开销。

只有在reduceByKey处理不了时，才用groupByKey().map()来替代。

### shuffle性能优化

new SparkConf().set(“spark.shuffle.consolidateFiles”,”true”)

spark.shuffle.consolidateFiles : 是否开启shuffle block file的合并，默认为false

spark.reducer.maxSizeInFlight : reduce task的拉取缓存，默认为48M

spark.shuffle.file.buffer : map task的写磁盘缓存，默认32k

spark.shuffle.io.maxRetries : 拉取失败的最大重试次数，默认3次

spark.shuffle.io.retryWait : 拉取失败的重试间隔，默认5s

spark.shuffle.memoryFraction : 用于reduce端聚合的内存比例，默认0.2，超过比例就会溢出到磁盘上

详细情况参考原理图