|  |
| --- |
| 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

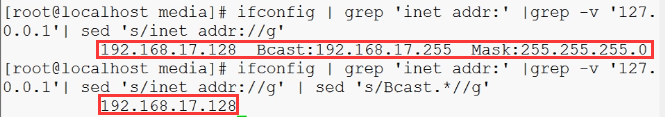
cat /etc/passwd | sed '匹配条件/执行的动作'

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

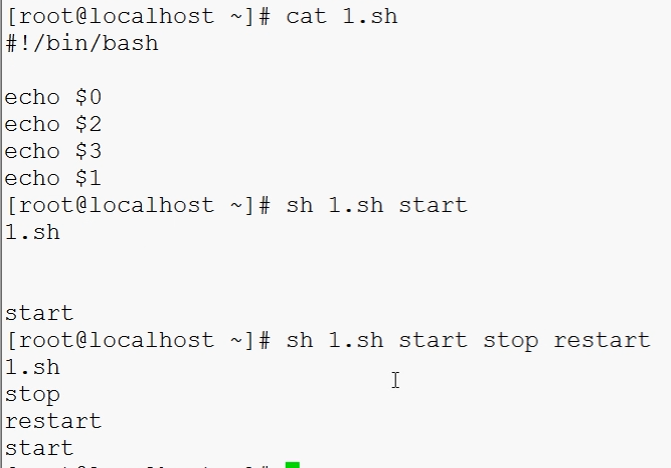
#### 位置变量

--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

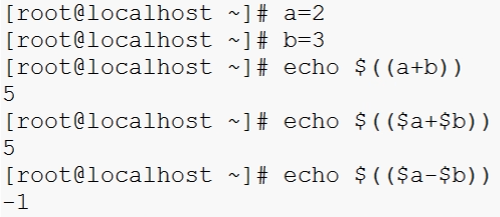
-- + - \* / %

--$((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

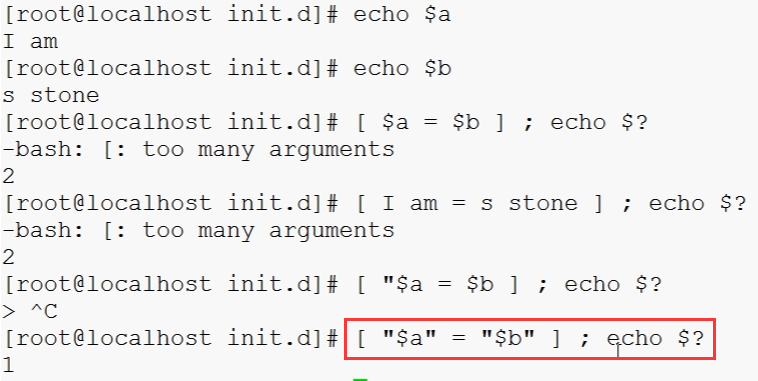
#### 字符串测试

= 等于则为真

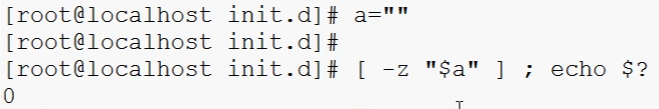
!= 不相等则为真

-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：

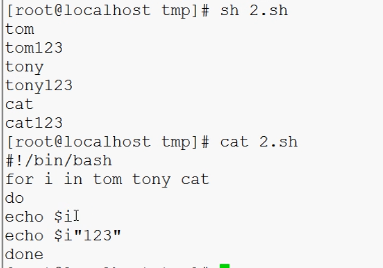
for 变量 in 值1 值2 … 值N

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

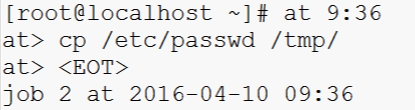
用法：at [时间]

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

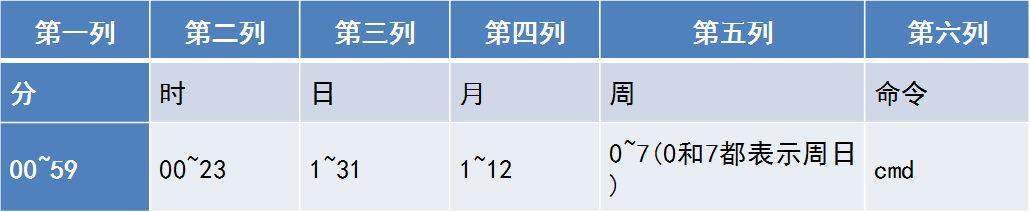
-e：制定计划任务

-l：列出计划任务

①确保crontab启动

service crond status #查看crontab状态

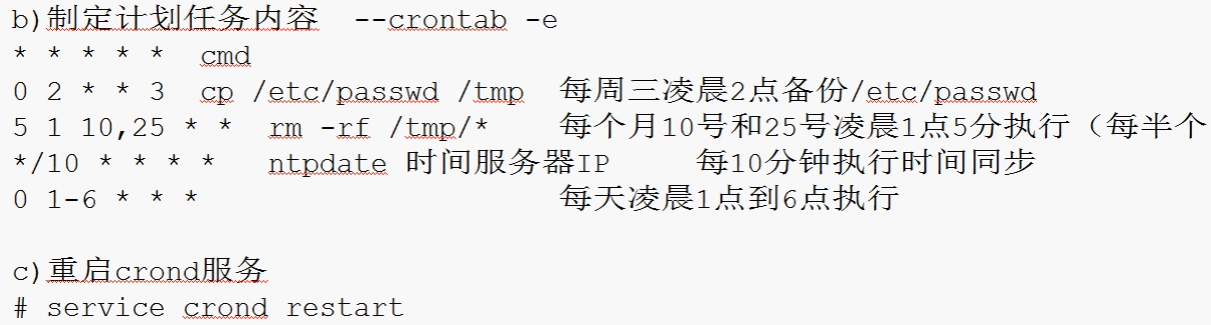
②任务格式



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)

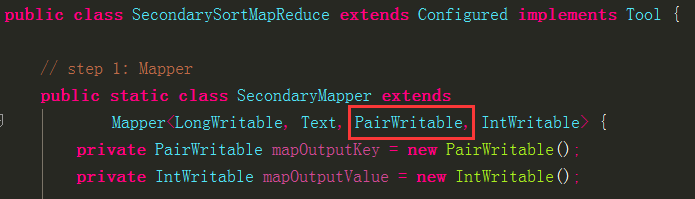
分区时不同的key可能分到不同的分区，分区后进行排序再分组，而分组时相同的key一定会分到相同的组，并且会形成一个keyvalues对，values为该key的所有value的集合。

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

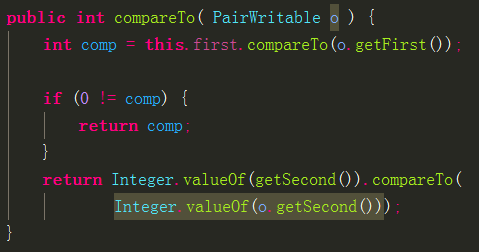
### 组合key

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

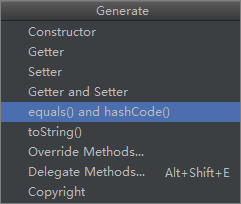


[自定义数据类型生成步骤：](WorkSpace/bigdata-hdfs/src/main/java/com/ibeifeng/mapreduce/PairWritable.java)

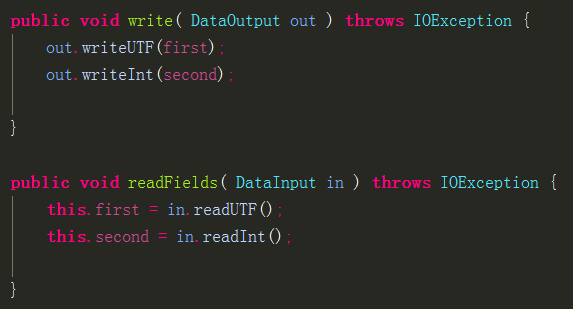
1. 实现WritableComparable接口，添加未实现方法；
2. 添加类成员变量并生成get和set方法；
3. 生成无参和有参构造方法；
4. 实现compareTo方法，用来定义排序规则；



1. 生成hashCode和equals方法；



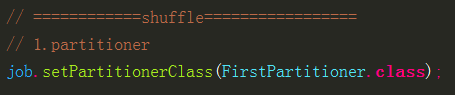
1. 生成tostring方法；
2. 实现写和度方法（两个方法参数顺序要一致）



### 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)

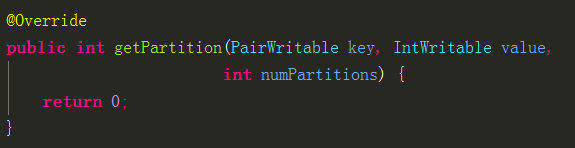
1. Job设置分区类



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

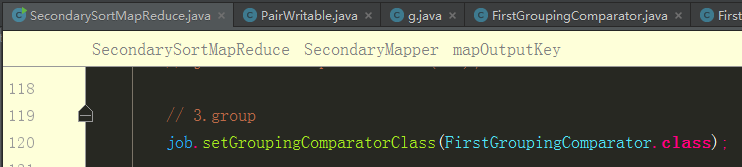
hashPartitioner分区是MR框架默认的分区计算方法



### 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)

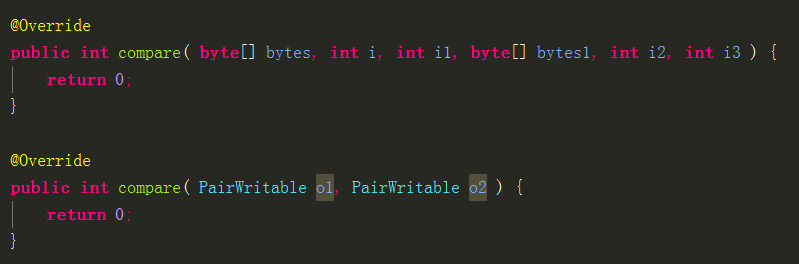
1. Job设置分组类



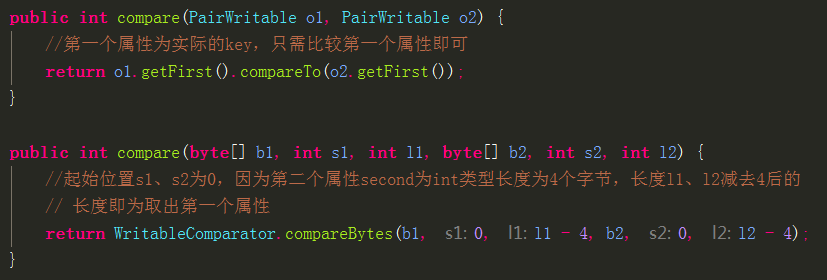
1. 新建分组类，实现RawComparator接口，泛型中的类为要分组排序的数据类



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

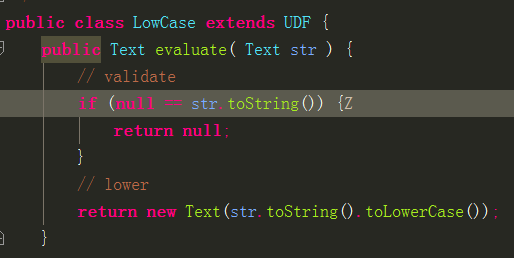
 注意被cluster by指定的列只能是降序，不能指定asc和desc。

### 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

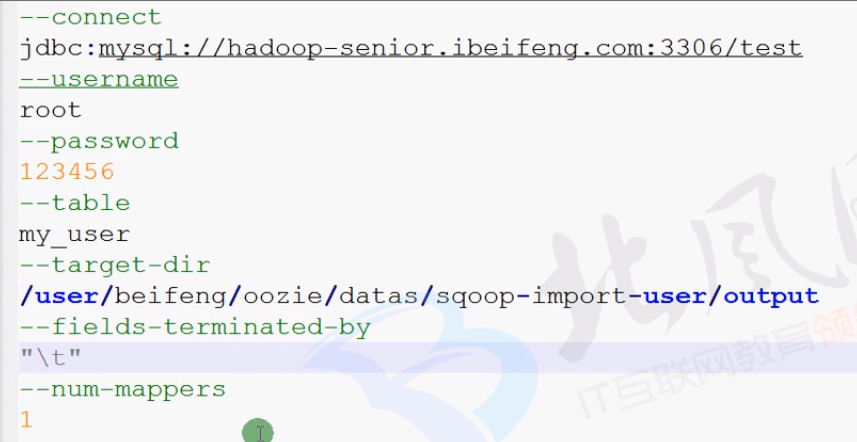
load data ...

## 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：

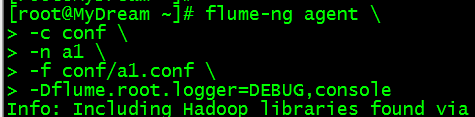
flume-ng agent \

-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

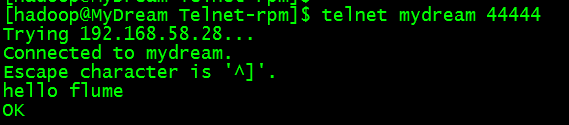
-Dflume.root.logger=DEBUG,console 显示信息到控制台



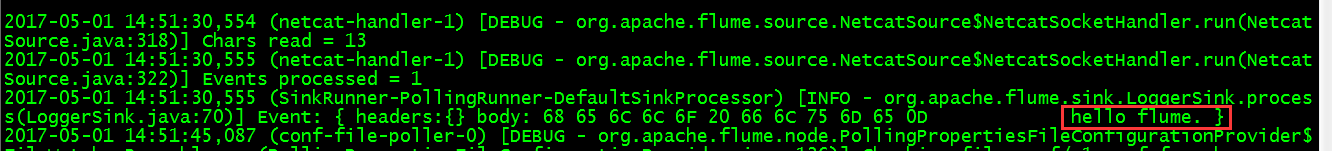
### 连接a1.conf配置文件指定的端口

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

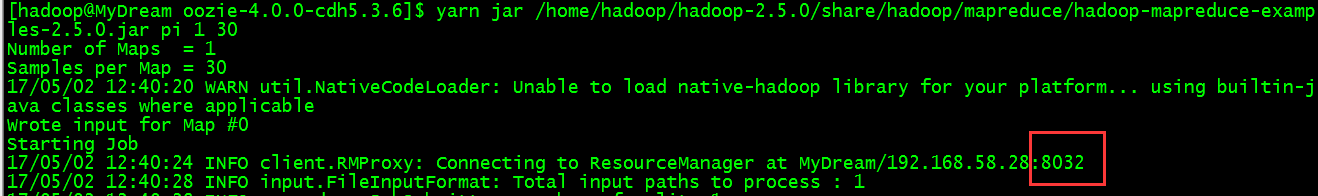
## Oozie安装

[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

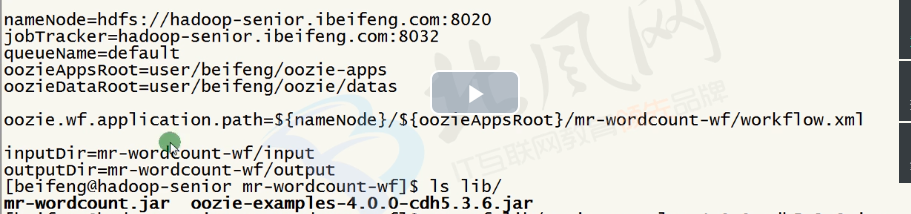
OozieDatapath= user/hadoop/wordcount/data 定义变量

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

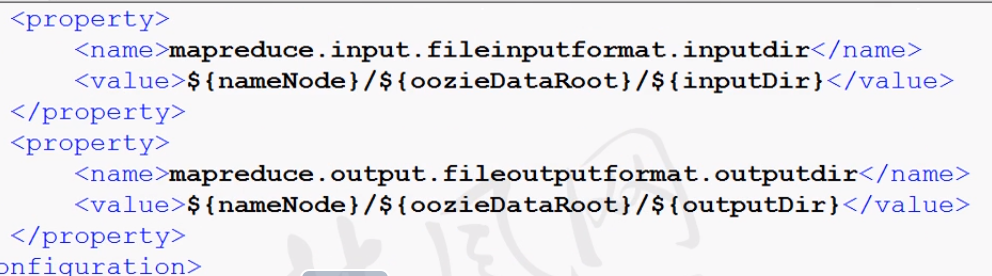


将自定义的MapReduce输出结果的目录配置进

<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

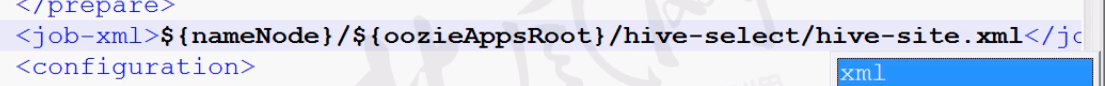
将hive的hive-site.xml文件放到复制/home/hadoop/oozie-4.0.0-cdh5.3.6/examples/apps/hive出来后的目录下，和xml properties 脚本放在同一个文件夹，并在该文件夹新建lib目录，将MySQL jdbc jar包放到该目录。

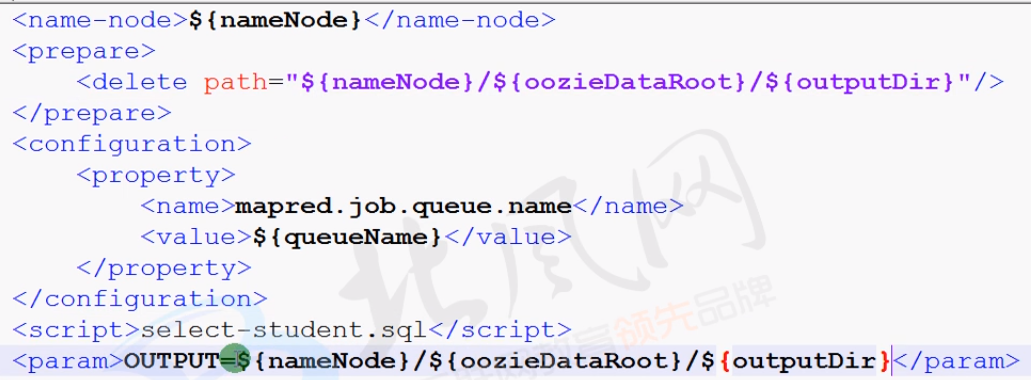
编辑job.properties



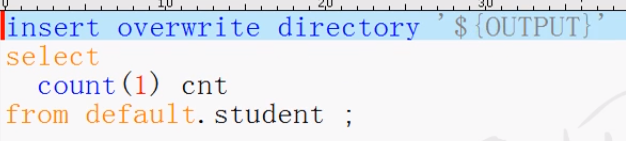
编辑workflow.xml

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

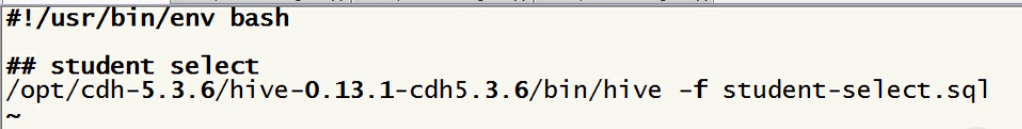
新建lib目录，复制MySQL jdbc jar包进去



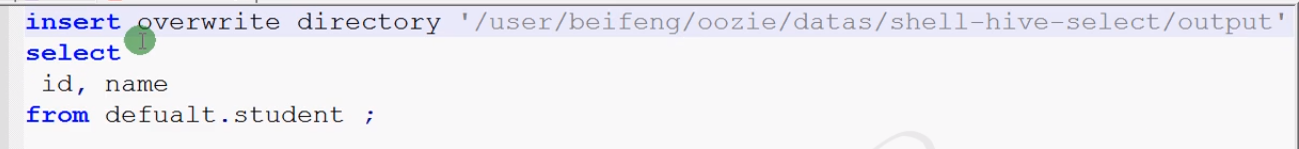
## 编写shell action

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());

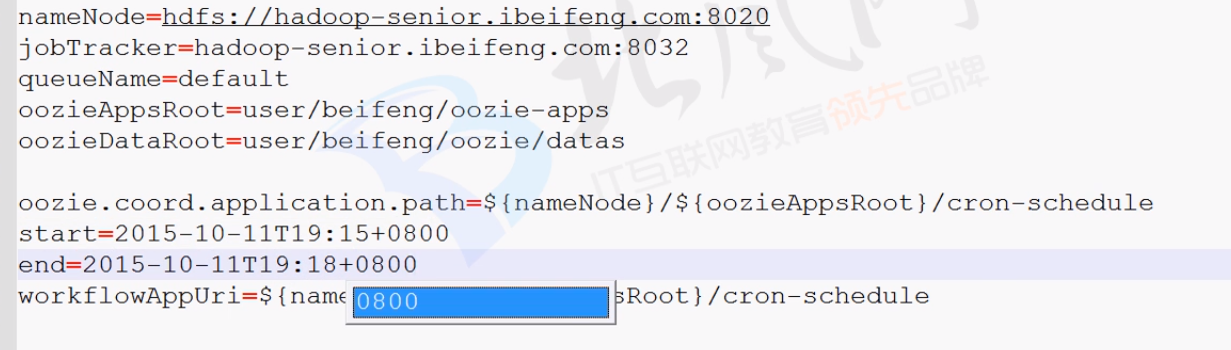
return Ext.state.Manager.get("TimezoneId","GMT+0800");

}

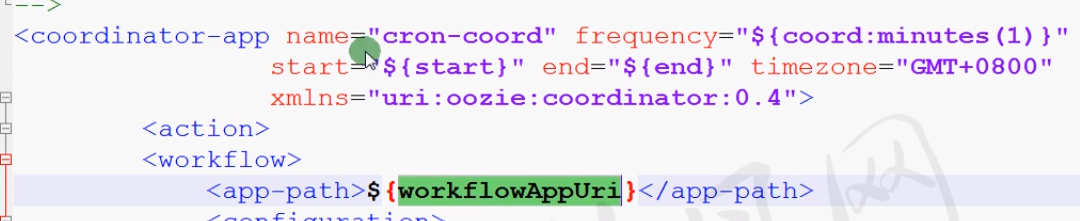
## 编写定时任务

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

编辑job.properties



编辑coordinator.xml

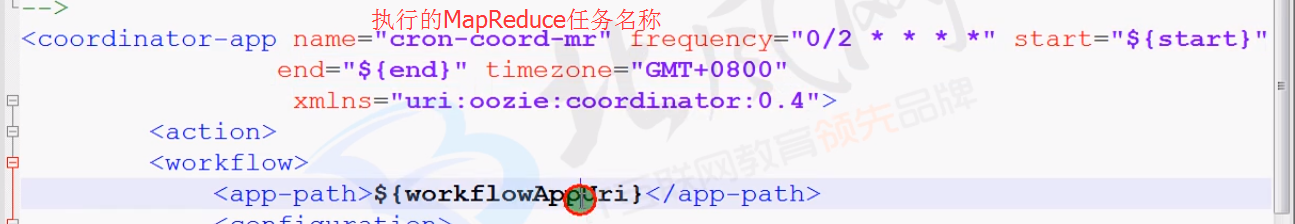


## 测试cron任务

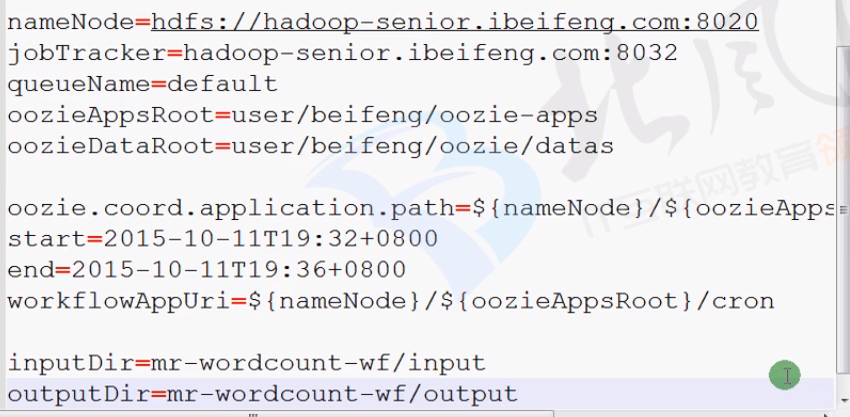
参考/home/hadoop/oozie-4.0.0-cdh5.3.6/examples/apps/cron

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

编辑coordinator.xml



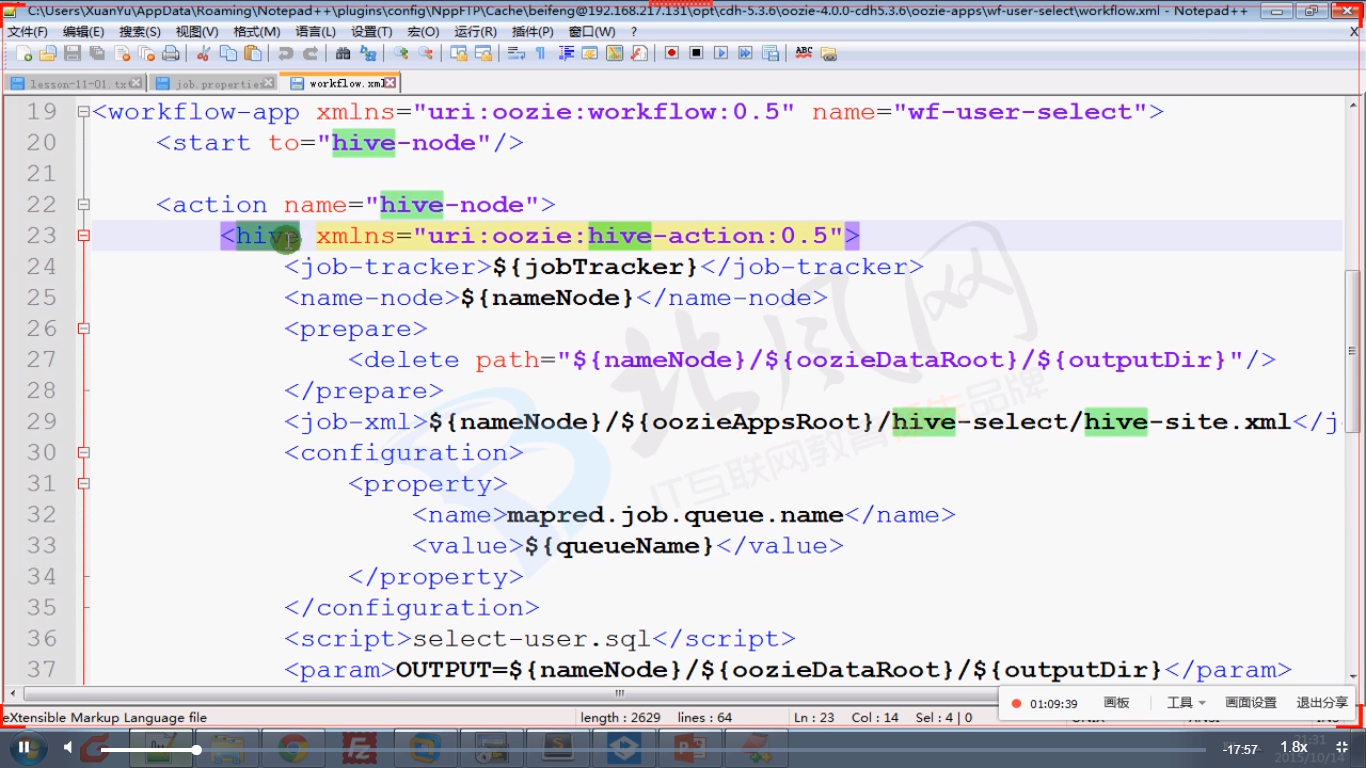
编辑job.properties

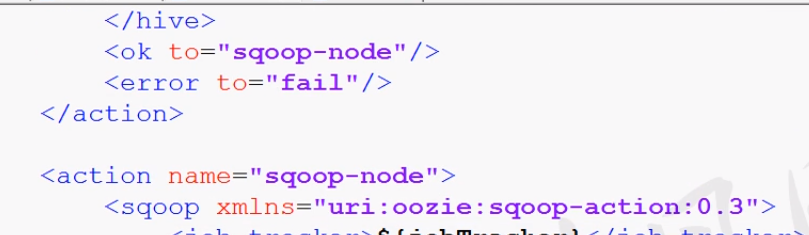


## 多个action

将两个或多个

配置workflow.xml action跳转

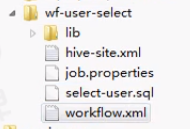




Job.properties



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)

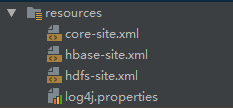
## Hbase使用

### 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

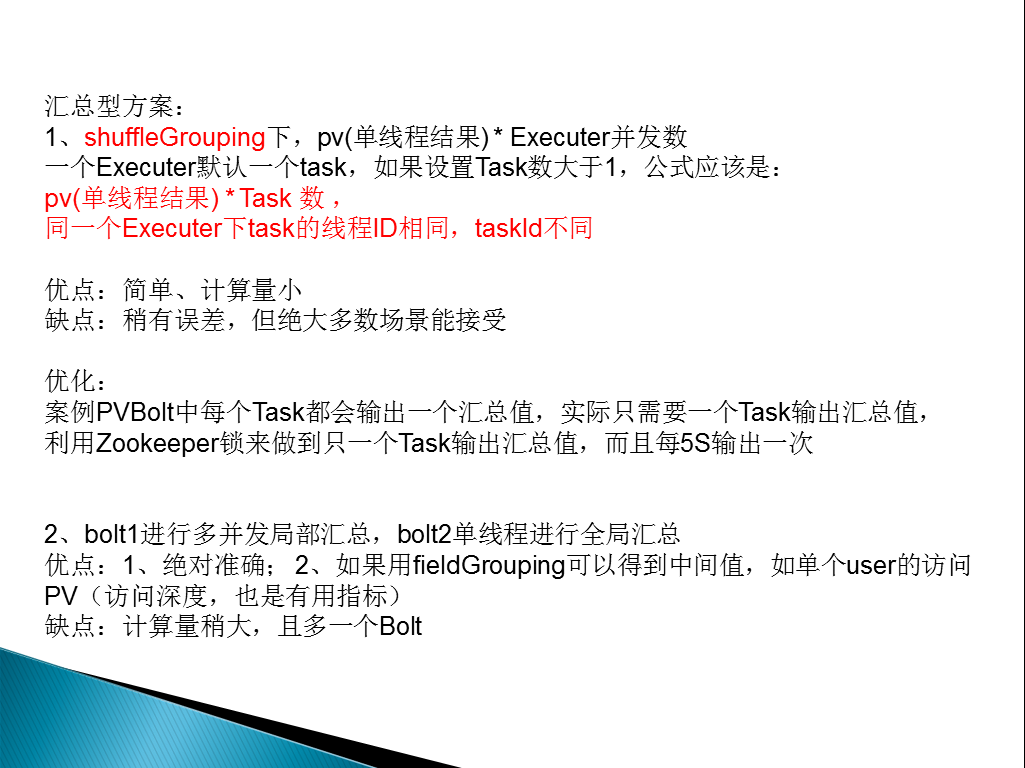
/storms Topology的基本配置信息

/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

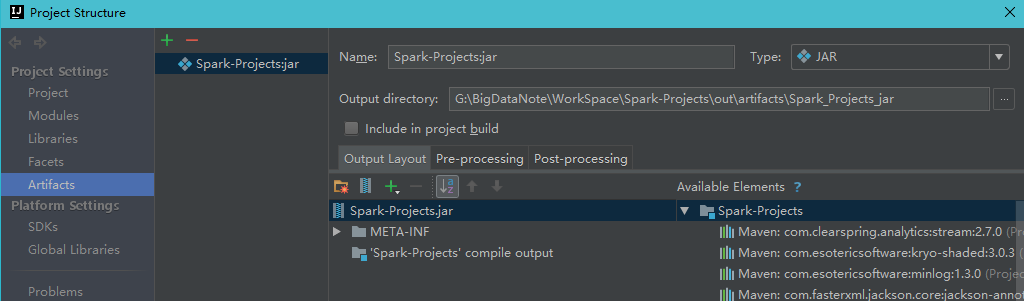
[示例代码](WorkSpace/Spark-Projects/src/main/scala/spark/app)

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

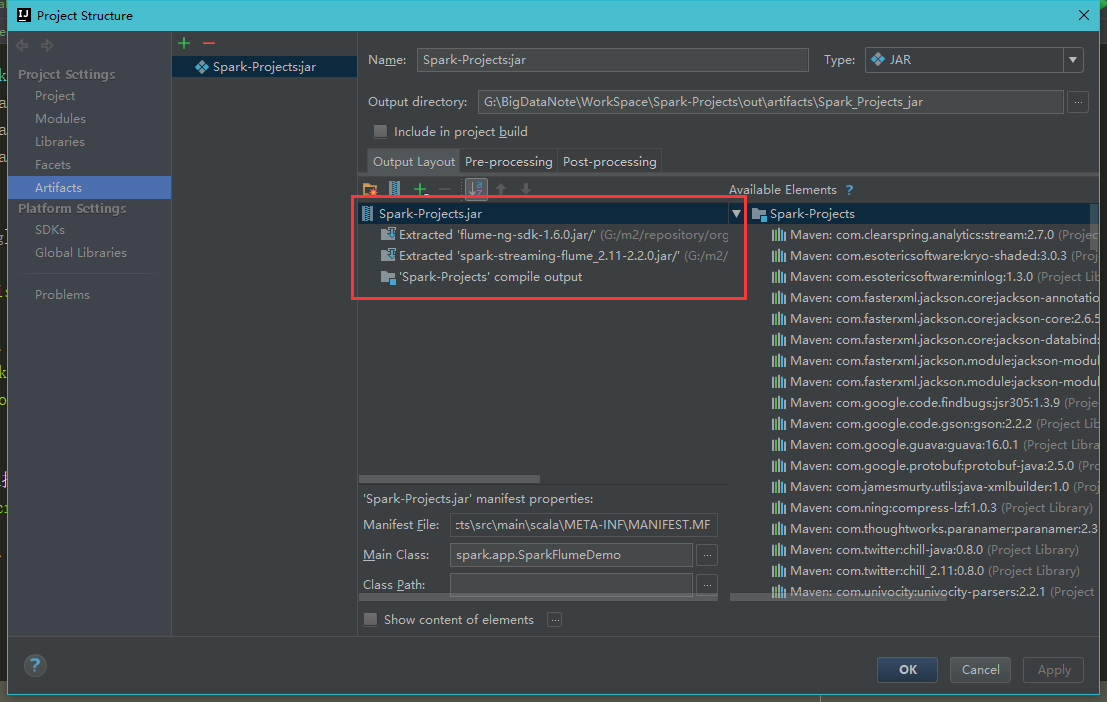
编写flume配置文件监控某文件

[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\_DISK\_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

## DataFrame的使用

Spark SQL支持用spark.sql.dialect参数设置SQL的方言。使用SQLContext的setConf()即可进行设置。对于SQLContext，它只支持“sql”一种方言。对于HiveContext，它默认的方言是“hiveql”。

DataFrame操作

[java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/sql/DataFrameCreateAndOperation.java)

[scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/sql/DataFrameCreateAndOperation.scala)

打包代码并上传，提交任务（需启动HDFS，spark，mysql）

spark-submit \

--class spark.sql.DataFrameCreateAndOperation \

--num-executors 3 \

--driver-memory 100m \

--executor-memory 100m \

--executor-cores 3 \

--files /home/zkj/apache-hive-2.3.0-bin/conf/hive-site.xml \

--driver-class-path /home/zkj/apache-hive-2.3.0-bin/lib/mysql-connector-java-5.1.17.jar \

/home/zkj/Spark-Projects-0.0.1-SNAPSHOT-jar-with-dependencies.jar

### RDD转换为DataFrame

Spark SQL支持两种方式来将RDD转换为DataFrame。

第一种方式，是使用反射来推断包含了特定数据类型的RDD的元数据。这种基于反射的方式，代码比较简洁，当你已经知道你的RDD的元数据时，是一种非常不错的方式。

第二种方式，是通过编程接口来创建DataFrame，你可以在程序运行时动态构建一份元数据，然后将其应用到已经存在的RDD上。这种方式的代码比较冗长，但是如果在编写程序时，还不知道RDD的元数据，只有在程序运行时，才能动态得知其元数据，那么只能通过这种动态构建元数据的方式。

#### 使用反射方式推断元数据

Java版本：Spark SQL是支持将包含了JavaBean的RDD转换为DataFrame的。JavaBean的信息，就定义了元数据。Spark SQL现在是不支持将包含了嵌套JavaBean或者List等复杂数据的JavaBean，作为元数据的。只支持一个包含简单数据类型的field的JavaBean。

Scala版本：而Scala由于其具有隐式转换的特性，所以Spark SQL的Scala接口，是支持自动将包含了case class的RDD转换为DataFrame的。case class就定义了元数据。Spark SQL会通过反射读取传递给case class的参数的名称，然后将其作为列名。与Java不同的是，Spark SQL是支持将包含了嵌套数据结构的case class作为元数据的，比如包含了Array等。

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/sql/RDD2DataFrameReflection.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/sql/RDD2DataFrameReflection.scala)

#### 使用编程方式指定元数据

Java版本：当JavaBean无法预先定义和知道的时候，比如要动态从一个文件中读取数据结构，那么就只能用编程方式动态指定元数据了。首先要从原始RDD创建一个元素为Row的RDD；其次要创建一个StructType，来代表Row；最后将动态定义的元数据应用到RDD<Row>上。

Scala版本：Scala的实现方式，与Java是基本一样的。

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/sql/RDD2DataFrameProgrammatically.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/sql/RDD2DataFrameProgrammatically.scala)

## 通用的load和save操作

对于Spark SQL的DataFrame来说，无论是从什么数据源创建出来的DataFrame，都有一些共同的load和save操作。load操作主要用于加载数据，创建出DataFrame；save操作，主要用于将DataFrame中的数据保存到文件中。

Java版本

DataFrame df = sqlContext.read().load("users.parquet");

df.select("name", "favorite\_color").write().save("namesAndFavColors.parquet");

Scala版本

val df = sqlContext.read.load("users.parquet")

df.select("name", "favorite\_color").write.save("namesAndFavColors.parquet")

也可以手动指定用来操作的数据源类型。数据源通常需要使用其全限定名来指定，比如parquet是org.apache.spark.sql.parquet。但是Spark SQL内置了一些数据源类型，比如json，parquet，jdbc等等。实际上，通过这个功能，就可以在不同类型的数据源之间进行转换了 比如将json文件中的数据保存到parquet文件中。默认情况下，如果不指定数据源类型，那么就是parquet。

Java版本

DataFrame df = sqlContext.read().format("json").load("people.json");

df.select("name", "age").write().format("parquet").save("namesAndAges.parquet");

Scala版本

val df = sqlContext.read.format("json").load("people.json")

df.select("name", "age").write.format("parquet").save("namesAndAges.parquet")

Spark SQL对于save操作，提供了不同的save mode。主要用来处理，当目标位置，已经有数据时，应该如何处理。而且save操作并不会执行锁操作，并且不是原子的，因此是有一定风险出现脏数据的。

|  |  |
| --- | --- |
| **Save Mode** | **意义** |
| SaveMode.ErrorIfExists (默认) | 如果目标位置已经存在数据，那么抛出一个异常 |
| SaveMode.Append | 如果目标位置已经存在数据，那么将数据追加进去 |
| SaveMode.Overwrite | 如果目标位置已经存在数据，那么就将已经存在的数据删除，用新数据进行覆盖 |
| SaveMode.Ignore | 如果目标位置已经存在数据，那么就忽略，不做任何操作。 |

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/sql/loadSave)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/sql/loadSave)

## 数据源Parquet之使用编程方式加载数据

Parquet是面向分析型业务的列式存储格式。

列式存储和行式存储相比有哪些优势呢？

1、可以跳过不符合条件的数据，只读取需要的数据，降低IO数据量。

2、压缩编码可以降低磁盘存储空间。由于同一列的数据类型是一样的，可以使用更高效的压缩编码（例如Run Length Encoding和Delta Encoding）进一步节约存储空间。

3、只读取需要的列，支持向量运算，能够获取更好的扫描性能。

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/sql/loadSave/ParquetLoadData.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/sql/loadSave/ParquetLoadData.scala)

## Parquet数据源之自动分区推断

表分区是一种常见的优化方式，比如Hive中就提供了表分区的特性。在一个分区表中，不同分区的数据通常存储在不同的目录中，分区列的值通常就包含在了分区目录的目录名中。Spark SQL中的Parquet数据源，支持自动根据目录名推断出分区信息。例如，如果将人口数据存储在分区表中，并且使用性别和国家作为分区列。那么目录结构可能如下所示：

tableName

|- gender=male

|- country=US

...

...

...

|- country=CN

...

|- gender=female

|- country=US

...

|- country=CH

如果将/tableName传入SQLContext.read.parquet()或者SQLContext.read.load()方法，那么Spark SQL就会自动根据目录结构，推断出分区信息，是gender和country。即使数据文件中只包含了两列值，name和age，但是Spark SQL返回的DataFrame，调用printSchema()方法时，会打印出四个列的值：name，age，country，gender。这就是自动分区推断的功能。

此外，分区列的数据类型，也是自动被推断出来的。目前，Spark SQL仅支持自动推断出数字类型和字符串类型。有时，用户也许不希望Spark SQL自动推断分区列的数据类型。此时只要设置一个配置即可， spark.sql.sources.partitionColumnTypeInference.enabled，默认为true，即自动推断分区列的类型，设置为false，即不会自动推断类型。禁止自动推断分区列的类型时，所有分区列的类型，就统一默认都是String。

案例：自动推断用户数据的性别和国家

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/sql/ParquetPartitionDiscovery.java)

## Parquet数据源之合并元数据

如同ProtocolBuffer，Avro，Thrift一样，Parquet也是支持元数据合并的。用户可以在一开始就定义一个简单的元数据，然后随着业务需要，逐渐往元数据中添加更多的列。在这种情况下，用户可能会创建多个Parquet文件，有着多个不同的但是却互相兼容的元数据。Parquet数据源支持自动推断出这种情况，并且进行多个Parquet文件的元数据的合并。

因为元数据合并是一种相对耗时的操作，而且在大多数情况下不是一种必要的特性，从Spark 1.5.0版本开始，默认是关闭Parquet文件的自动合并元数据的特性的。可以通过以下两种方式开启Parquet数据源的自动合并元数据的特性：

1、读取Parquet文件时，将数据源的选项，mergeSchema，设置为true

2、使用SQLContext.setConf()方法，将spark.sql.parquet.mergeSchema参数设置为true

案例：合并学生的基本信息，和成绩信息的元数据

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/sql/ParquetMergeSchema.scala)

## JSON数据源

Spark SQL可以自动推断JSON文件的元数据，并且加载其数据，创建一个DataFrame。可以使用SQLContext.read.json()方法，针对一个元素类型为String的RDD，或者是一个JSON文件。

但是要注意的是，这里使用的JSON文件与传统意义上的JSON文件是不一样的。每行都必须，也只能包含一个，单独的，自包含的，有效的JSON对象。不能让一个JSON对象分散在多行。否则会报错。

综合性复杂案例：查询成绩为80分以上的学生的基本信息与成绩信息

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/sql/JSONDataSource.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/sql/JSONDataSource.scala)

## Hive数据源

Spark SQL支持对Hive中存储的数据进行读写。操作Hive中的数据时，必须创建HiveContext，而不是SQLContext。HiveContext继承自SQLContext，但是增加了在Hive元数据库中查找表，以及用HiveQL语法编写SQL的功能。除了sql()方法，HiveContext还提供了hql()方法，从而用Hive语法来编译sql。

使用HiveContext，可以执行Hive的大部分功能，包括创建表、往表里导入数据以及用SQL语句查询表中的数据。查询出来的数据是一个Row数组。

将hive-site.xml拷贝到spark/conf目录下，将mysql connector拷贝到spark/lib目录下

HiveContext sqlContext = new HiveContext(sc);

sqlContext.sql("CREATE TABLE IF NOT EXISTS students (name STRING, age INT)");

sqlContext.sql("LOAD DATA LOCAL INPATH '/usr/local/spark-study/resources/students.txt' INTO TABLE students");

Row[] teenagers = sqlContext.sql("SELECT name, age FROM students WHERE age<=18").collect();

Spark SQL还允许将数据保存到Hive表中。调用DataFrame的saveAsTable命令，即可将DataFrame中的数据保存到Hive表中。与registerTempTable不同，saveAsTable是会将DataFrame中的数据物化到Hive表中的，而且还会在Hive元数据库中创建表的元数据。

默认情况下，saveAsTable会创建一张Hive Managed Table，也就是说，数据的位置都是元数据库中的信息控制的。当Managed Table被删除时，表中的数据也会一并被物理删除。

registerTempTable只是注册一个临时的表，只要Spark Application重启或者停止了，那么表就没了。而saveAsTable创建的是物化的表，无论Spark Application重启或者停止，表都会一直存在。

调用HiveContext.table()方法，还可以直接针对Hive中的表，创建一个DataFrame。

案例：查询分数大于80分的学生的完整信息

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/sql/HiveDataSource.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/sql/HiveDataSource.scala)

## JDBC数据源实战

Java版本

Map<String, String> options = new HashMap<String, String>();

options.put("url", "jdbc:mysql://spark1:3306/testdb");

options.put("dbtable", "students");

DataFrame jdbcDF = sqlContext.read().format("jdbc"). options(options).load();

Scala版本

val jdbcDF = sqlContext.read.format("jdbc").options(

Map("url" -> "jdbc:mysql://spark1:3306/testdb",

"dbtable" -> "students")).load()

案例：查询分数大于80分的学生信息

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/sql/JDBCDataSource.java)

## 案例实战

### 内置函数以及每日uv、销售额统计案例实战

在Spark 1.5.x版本，增加了一系列内置函数到DataFrame API中，并且实现了code-generation的优化。与普通的函数不同，DataFrame的函数并不会执行后立即返回一个结果值，而是返回一个Column对象，用于在并行作业中进行求值。Column可以用在DataFrame的操作之中，比如select，filter，groupBy等。函数的输入值，也可以是Column。

|  |  |
| --- | --- |
| **种类** | **函数** |
| 聚合函数 | approxCountDistinct, avg, count, countDistinct, first, last, max, mean, min, sum, sumDistinct |
| 集合函数 | array\_contains, explode, size, sort\_array |

|  |  |
| --- | --- |
| **种类** | **函数** |
| 数学函数 | abs, acros, asin, atan, atan2, bin, cbrt, ceil, conv, cos, sosh, exp, expm1, factorial, floor, hex, hypot, log, log10, log1p, log2, pmod, pow, rint, round, shiftLeft, shiftRight, shiftRightUnsigned, signum, sin, sinh, sqrt, tan, tanh, toDegrees, toRadians, unhex |

|  |  |
| --- | --- |
| **种类** | **函数** |
| 混合函数 | array, bitwiseNOT, callUDF, coalesce, crc32, greatest, if, inputFileName, isNaN, isnotnull, isnull, least, lit, md5, monotonicallyIncreasingId, nanvl, negate, not, rand, randn, sha, sha1, sparkPartitionId, struct, when |

|  |  |
| --- | --- |
| 字符串函数 | ascii, base64, concat, concat\_ws, decode, encode, format\_number, format\_string, get\_json\_object, initcap, instr, length, levenshtein, locate, lower, lpad, ltrim, printf, regexp\_extract, regexp\_replace, repeat, reverse, rpad, rtrim, soundex, space, split, substring, substring\_index, translate, trim, unbase64, upper |
| 窗口函数 | cumeDist, denseRank, lag, lead, ntile, percentRank, rank, rowNumber |

案例实战：根据每天的用户访问日志和用户购买日志，统计每日的uv和销售额

[Scala代码示例](WorkSpace/Spark-Projects/src/main/scala/spark1/sql/DailySale)

### 开窗函数以及top3销售额统计案例实战

案例：统计每个种类的销售额排名前3的产品

[Java代码示例](WorkSpace/Spark-Projects/src/main/java/spark/sql/RowNumberWindowFunction.java)

### UDF自定义函数实战

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/sql/UDF.scala)

### UDAF自定义函数实战

UDF，其实更多的是针对单行输入，返回一个输出

这里的UDAF，则可以针对多行输入，进行聚合计算，返回一个输出，功能更加强大

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/sql/UDAF)

## Spark SQL：工作原理剖析以及性能优化

**工作原理剖析**

1、SqlParse

2、Analyser

3、Optimizer

4、SparkPlan

[原理图](参考文件/原理图/SparkSQL工作原理剖析.png)

**性能优化**

1、设置Shuffle过程中的并行度：spark.sql.shuffle.partitions（SQLContext.setConf()）

2、在Hive数据仓库建设过程中，合理设置数据类型，比如能设置为INT的，就不要设置为BIGINT。减少数据类型导致的不必要的内存开销。

3、编写SQL时，尽量给出明确的列名，比如select name from students。不要写select \*的方式。

4、并行处理查询结果：对于Spark SQL查询的结果，如果数据量比较大，比如超过1000条，那么就不要一次性collect()到Driver再处理。使用foreach()算子，并行处理查询结果。

5、缓存表：对于一条SQL语句中可能多次使用到的表，可以对其进行缓存，使用SQLContext.cacheTable(tableName)，或者DataFrame.cache()即可。Spark SQL会用内存列存储的格式进行表的缓存。然后Spark SQL就可以仅仅扫描需要使用的列，并且自动优化压缩，来最小化内存使用和GC开销。SQLContext.uncacheTable(tableName)可以将表从缓存中移除。用SQLContext.setConf()，设置spark.sql.inMemoryColumnarStorage.batchSize参数（默认10000），可以配置列存储的单位。

6、广播join表：spark.sql.autoBroadcastJoinThreshold，默认10485760 (10 MB)。在内存够用的情况下，可以增加其大小，概参数设置了一个表在join的时候，最大在多大以内，可以被广播出去优化性能。

7、钨丝计划：spark.sql.tungsten.enabled，默认是true，自动管理内存。

最有效的，其实就是第四点、缓存表和广播join表，也是非常不错的！

## Hive On Spark

4、SparkContext生命周期

Hive On Spark会为每个用户的会话，比如执行一次SQL语句，创建一个SparkContext。但是Spark不允许在一个JVM内创建多个SparkContext。因此，需要在单独的JVM中启动每个会话的SparkContext，然后通过RPC与远程JVM中的SparkContext进行通信。

5、本地和远程运行模式

Hive On Spark提供两种运行模式，本地和远程。如果将Spark Master设置为local，比如set spark.master=local，那么就是本地模式，SparkContext与客户端运行在一个JVM中。否则，如果将Spark Master设置为Master的地址，那么就是远程模式，SparkContext会在远程的JVM中启动。

远程模式下，每个用户Session都会创建一个SparkClient，SparkClient启动RemoteDriver，RemoteDriver负责创建SparkContext。

Hive On Spark做了一些优化：

1、Map Join

Spark SQL默认对join是支持使用broadcast机制将小表广播到各个节点上，以进行join的。但是问题是，这会给Driver和Worker带来很大的内存开销。因为广播的数据要一直保留在Driver内存中。所以目前采取的是，类似乎MapReduce的Distributed Cache机制，即提高HDFS replica factor的复制因子，以让数据在每个计算节点上都有一个备份，从而可以在本地进行数据读取。

2、Cache Table

对于某些需要对一张表执行多次操作的场景，Hive On Spark内部做了优化，即将要多次操作的表cache到内存中，以便于提升性能。但是这里要注意，并不是对所有的情况都会自动进行cache。所以说，Hive On Spark还有很多不完善的地方。

[Hive On Spark环境搭建](参考文件/Hive%20On%20Spark环境搭建.docx)

## 每日top3热点搜索词统计案例实战

**数据格式：**

日期 用户 搜索词 城市 平台 版本

**需求：**

1、筛选出符合查询条件（城市、平台、版本）的数据

2、统计出每天搜索uv排名前3的搜索词

3、按照每天的top3搜索词的uv搜索总次数，倒序排序

4、将数据保存到hive表中

**实现思路分析**

1、针对原始数据（HDFS文件），获取输入的RDD

2、使用filter算子，去针对输入RDD中的数据，进行数据过滤，过滤出符合查询条件的数据。

2.1 普通的做法：直接在fitler算子函数中，使用外部的查询条件（Map），但是，这样做的话，是不是查询条件Map，会发送到每一个task上一份副本。（性能并不好）

2.2 优化后的做法：将查询条件，封装为Broadcast广播变量，在filter算子中使用Broadcast广播变量进行数据筛选。

3、将数据转换为“(日期\_搜索词, 用户)”格式，然后呢，对它进行分组，然后再次进行映射，对每天每个搜索词的搜索用户进行去重操作，并统计去重后的数量，即为每天每个搜索词的uv。最后，获得“(日期\_搜索词, uv)”

4、将得到的每天每个搜索词的uv，RDD，映射为元素类型为Row的RDD，将该RDD转换为DataFrame

5、将DataFrame注册为临时表，使用Spark SQL的开窗函数，来统计每天的uv数量排名前3的搜索词，以及它的搜索uv，最后获取，是一个DataFrame

6、将DataFrame转换为RDD，继续操作，按照每天日期来进行分组，并进行映射，计算出每天的top3搜索词的搜索uv的总数，然后将uv总数作为key，将每天的top3搜索词以及搜索次数，拼接为一个字符串

7、按照每天的top3搜索总uv，进行排序，倒序排序

8、将排好序的数据，再次映射回来，变成“日期\_搜索词\_uv”的格式

9、再次映射为DataFrame，并将数据保存到Hive中即可

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/sql/DailyTop3Keyword.java)

## 实时wordcount程序开发

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/stream/WordCount.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/stream/WordCount.scala)

提交应用：

spark-submit \

--class cn.spark.study.streaming.WordCount \

--num-executors 3 \

--driver-memory 100m \

--executor-memory 100m \

--executor-cores 3 \

/usr/local/spark-study/scala/streaming/spark-study-scala.jar \

## StreamingContext详解

有两种创建StreamingContext的方式：

val conf = new SparkConf().setAppName(appName).setMaster(master);

val ssc = new StreamingContext(conf, Seconds(1));

StreamingContext，还可以使用已有的SparkContext来创建

val sc = new SparkContext(conf)

val ssc = new StreamingContext(sc, Seconds(1));

appName，是用来在Spark UI上显示的应用名称。master，是一个Spark、Mesos或者Yarn集群的URL，或者是local[\*]。

batch interval可以根据你的应用程序的延迟要求以及可用的集群资源情况来设置。

一个StreamingContext定义之后，必须做以下几件事情：

1、通过创建输入DStream来创建输入数据源。

2、通过对DStream定义transformation和output算子操作，来定义实时计算逻辑。

3、调用StreamingContext的start()方法，来开始实时处理数据。

4、调用StreamingContext的awaitTermination()方法，来等待应用程序的终止。可以使用CTRL+C手动停止，或者就是让它持续不断的运行进行计算。

5、也可以通过调用StreamingContext的stop()方法，来停止应用程序。

需要注意的要点：

1、只要一个StreamingContext启动之后，就不能再往其中添加任何计算逻辑了。比如执行start()方法之后，还给某个DStream执行一个算子。

2、一个StreamingContext停止之后，是肯定不能够重启的。调用stop()之后，不能再调用start()

3、一个JVM同时只能有一个StreamingContext启动。在你的应用程序中，不能创建两个StreamingContext。

4、调用stop()方法时，会同时停止内部的SparkContext，如果不希望如此，还希望后面继续使用SparkContext创建其他类型的Context，比如SQLContext，那么就用stop(false)。

5、一个SparkContext可以创建多个StreamingContext，只要上一个先用stop(false)停止，再创建下一个即可。

## 输入DStream和Receiver详解

输入DStream代表了来自数据源的输入数据流。在之前的wordcount例子中，lines就是一个输入DStream（JavaReceiverInputDStream），代表了从netcat（nc）服务接收到的数据流。除了文件数据流之外，所有的输入DStream都会绑定一个Receiver对象，该对象是一个关键的组件，用来从数据源接收数据，并将其存储在Spark的内存中，以供后续处理。

Spark Streaming提供了两种内置的数据源支持；  
1、基础数据源：StreamingContext API中直接提供了对这些数据源的支持，比如文件、socket、Akka Actor等。

2、高级数据源：诸如Kafka、Flume、Kinesis、Twitter等数据源，通过第三方工具类提供支持。这些数据源的使用，需要引用其依赖。

3、自定义数据源：我们可以自己定义数据源，来决定如何接受和存储数据。

要注意的是，如果你想要在实时计算应用中并行接收多条数据流，可以创建多个输入DStream。这样就会创建多个Receiver，从而并行地接收多个数据流。但是要注意的是，一个Spark Streaming Application的Executor，是一个长时间运行的任务，因此，它会独占分配给Spark Streaming Application的cpu core。从而只要Spark Streaming运行起来以后，这个节点上的cpu core，就没法给其他应用使用了。

使用本地模式，运行程序时，绝对不能用local或者local[1]，因为那样的话，只会给执行输入DStream的executor分配一个线程。而Spark Streaming底层的原理是，至少要有两条线程，一条线程用来分配给Receiver接收数据，一条线程用来处理接收到的数据。因此必须使用local[n]，n>=2的模式。

如果不设置Master，也就是直接将Spark Streaming应用提交到集群上运行，那么首先，必须要求集群节点上，有>1个cpu core，其次，给Spark Streaming的每个executor分配的core，必须>1，这样，才能保证分配到executor上运行的输入DStream，两条线程并行，一条运行Receiver，接收数据；一条处理数据。否则的话，只会接收数据，不会处理数据。

[Receiver和cpu core分配说明原理图](参考文件/原理图/Receiver和cpu%20core分配说明.png)

## 输入DStream之基础数据源以及基于HDFS的实时wordcount程序

1、Socket：之前的wordcount例子，已经演示过了，StreamingContext.socketTextStream()

2、HDFS文件

基于HDFS文件的实时计算，其实就是，监控一个HDFS目录，只要其中有新文件出现，就实时处理。相当于处理实时的文件流。

streamingContext.fileStream<KeyClass, ValueClass, InputFormatClass>(dataDirectory)

streamingContext.fileStream[KeyClass, ValueClass, InputFormatClass](dataDirectory)

Spark Streaming会监视指定的HDFS目录，并且处理出现在目录中的文件。要注意的是，所有放入HDFS目录中的文件，都必须有相同的格式；必须使用移动或者重命名的方式，将文件移入目录；一旦处理之后，文件的内容即使改变，也不会再处理了；基于HDFS文件的数据源是没有Receiver的，因此不会占用一个cpu core。

基于HDFS的实时wordcount程序：

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/stream/HDFSWordCount.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/stream/HDFSWordCount.scala)

## 输入DStream之Kafka数据源实战

### 基于Receiver的方式

这种方式使用Receiver来获取数据。Receiver是使用Kafka的高层次Consumer API来实现的。receiver从Kafka中获取的数据都是存储在Spark Executor的内存中的，然后Spark Streaming启动的job会去处理那些数据。

然而，在默认的配置下，这种方式可能会因为底层的失败而丢失数据。如果要启用高可靠机制，让数据零丢失，就必须启用Spark Streaming的预写日志机制（Write Ahead Log，WAL）。该机制会同步地将接收到的Kafka数据写入分布式文件系统（比如HDFS）上的预写日志中。所以，即使底层节点出现了失败，也可以使用预写日志中的数据进行恢复。

1、在maven添加依赖

groupId = org.apache.spark

artifactId = spark-streaming-kafka\_2.10

version = 1.5.1

2、使用第三方工具类创建输入DStream

JavaPairReceiverInputDStream<String, String> kafkaStream =

KafkaUtils.createStream(streamingContext,

[ZK quorum], [consumer group id], [per-topic number of Kafka partitions to consume]);

1、Kafka中的topic的partition，与Spark中的RDD的partition是没有关系的。所以，在KafkaUtils.createStream()中，提高partition的数量，只会增加一个Receiver中，读取partition的线程的数量。不会增加Spark处理数据的并行度。

2、可以创建多个Kafka输入DStream，使用不同的consumer group和topic，来通过多个receiver并行接收数据。

3、如果基于容错的文件系统，比如HDFS，启用了预写日志机制，接收到的数据都会被复制一份到预写日志中。因此，在KafkaUtils.createStream()中，设置的持久化级别是StorageLevel.MEMORY\_AND\_DISK\_SER。

bin/kafka-topics.sh --zookeeper 192.168.1.107:2181,192.168.1.108:2181,192.168.1.109:2181 --topic TestTopic --replication-factor 1 --partitions 1 --create

bin/kafka-console-producer.sh --broker-list 192.168.1.107:9092,192.168.1.108:9092,192.168.1.109:9092 --topic TestTopic

[java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/stream/KafkaReceiverWordCount.java)

### 基于Direct的方式

这种新的不基于Receiver的直接方式，是在Spark 1.3中引入的，从而能够确保更加健壮的机制。替代掉使用Receiver来接收数据后，这种方式会周期性地查询Kafka，来获得每个topic+partition的最新的offset，从而定义每个batch的offset的范围。当处理数据的job启动时，就会使用Kafka的简单consumer api来获取Kafka指定offset范围的数据。

这种方式有如下优点：

1、简化并行读取：如果要读取多个partition，不需要创建多个输入DStream然后对它们进行union操作。Spark会创建跟Kafka partition一样多的RDD partition，并且会并行从Kafka中读取数据。所以在Kafka partition和RDD partition之间，有一个一对一的映射关系。

2、高性能：如果要保证零数据丢失，在基于receiver的方式中，需要开启WAL机制。这种方式其实效率低下，因为数据实际上被复制了两份，Kafka自己本身就有高可靠的机制，会对数据复制一份，而这里又会复制一份到WAL中。而基于direct的方式，不依赖Receiver，不需要开启WAL机制，只要Kafka中作了数据的复制，那么就可以通过Kafka的副本进行恢复。

3、一次且仅一次的事务机制：

基于receiver的方式，是使用Kafka的高阶API来在ZooKeeper中保存消费过的offset的。这是消费Kafka数据的传统方式。这种方式配合着WAL机制可以保证数据零丢失的高可靠性，但是却无法保证数据被处理一次且仅一次，可能会处理两次。因为Spark和ZooKeeper之间可能是不同步的。

基于direct的方式，使用kafka的简单api，Spark Streaming自己就负责追踪消费的offset，并保存在checkpoint中。Spark自己一定是同步的，因此可以保证数据是消费一次且仅消费一次。

JavaPairReceiverInputDStream<String, String> directKafkaStream =

KafkaUtils.createDirectStream(streamingContext,

[key class], [value class], [key decoder class], [value decoder class],

[map of Kafka parameters], [set of topics to consume]);

[java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/stream/KafkaDirectWordCount.java)

## DStream的transformation操作概览

|  |  |
| --- | --- |
| **Transformation** | **Meaning** |
| map | 对传入的每个元素，返回一个新的元素 |
| flatMap | 对传入的每个元素，返回一个或多个元素 |
| filter | 对传入的元素返回true或false，返回的false的元素被过滤掉 |
| union | 将两个DStream进行合并 |
| count | 返回元素的个数 |
| reduce | 对所有values进行聚合 |
| countByValue | 对元素按照值进行分组，对每个组进行计数，最后返回<K, V>的格式 |
| reduceByKey | 对key对应的values进行聚合 |
| cogroup | 对两个DStream进行连接操作，一个key连接起来的两个RDD的数据，都会以Iterable<V>的形式，出现在一个Tuple中。 |
| join | 对两个DStream进行join操作，每个连接起来的pair，作为新DStream的RDD的一个元素 |
| transform | 对数据进行转换操作 |
| updateStateByKey | 为每个key维护一份state，并进行更新（这个，我认为，是在普通的实时计算中，最有用的一种操作） |
| window | 对滑动窗口数据执行操作（实时计算中最有特色的一种操作） |

## updateStateByKey以及基于缓存的实时wordcount程序

updateStateByKey操作，可以让我们为每个key维护一份state，并持续不断的更新该state。1、首先，要定义一个state，可以是任意的数据类型；

2、其次，要定义state更新函数——指定一个函数如何使用之前的state和新值来更新state。

对于每个batch，Spark都会为每个之前已经存在的key去应用一次state更新函数，无论这个key在batch中是否有新的数据。如果state更新函数返回none，那么key对应的state就会被删除。

当然，对于每个新出现的key，也会执行state更新函数。

注意：updateStateByKey操作，要求必须开启Checkpoint机制。

案例：基于缓存的实时wordcount程序

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/stream/UpdateStateByKeyWordCount.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/stream/UpdateStateByKeyWordCount.scala)

## transform以及实时黑名单过滤案例实战

transform操作，应用在DStream上时，可以用于执行任意的RDD到RDD的转换操作。它可以用于实现，DStream API中所没有提供的操作。比如说，DStream API中，并没有提供将一个DStream中的每个batch，与一个特定的RDD进行join的操作。但是我们自己就可以使用transform操作来实现该功能。

DStream.join()，只能join其他DStream。在DStream每个batch的RDD计算出来之后，会去跟其他DStream的RDD进行join。

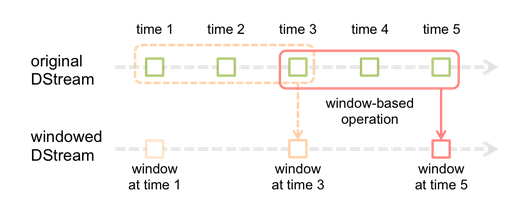
案例：广告计费日志实时黑名单过滤

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/stream/TransformBlacklist.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/stream/TransformBlacklist.scala)

## window滑动窗口以及热点搜索词滑动统计案例实战

Spark Streaming提供了滑动窗口操作的支持，从而让我们可以对一个滑动窗口内的数据执行计算操作。每次掉落在窗口内的RDD的数据，会被聚合起来执行计算操作，然后生成的RDD，会作为window DStream的一个RDD。比如下图中，就是对每三秒钟的数据执行一次滑动窗口计算，这3秒内的3个RDD会被聚合起来进行处理，然后过了两秒钟，又会对最近三秒内的数据执行滑动窗口计算。所以每个滑动窗口操作，都必须指定两个参数，窗口长度以及滑动间隔，而且这两个参数值都必须是batch间隔的整数倍。（Spark Streaming对滑动窗口的支持，是比Storm更加完善和强大的）



|  |  |
| --- | --- |
| **Transform** | **意义** |
| window | 对每个滑动窗口的数据执行自定义的计算 |
| countByWindow | 对每个滑动窗口的数据执行count操作 |
| reduceByWindow | 对每个滑动窗口的数据执行reduce操作 |
| reduceByKeyAndWindow | 对每个滑动窗口的数据执行reduceByKey操作 |
| countByValueAndWindow | 对每个滑动窗口的数据执行countByValue操作 |

案例：热点搜索词滑动统计，每隔10秒钟，统计最近60秒钟的搜索词的搜索频次，并打印出排名最靠前的3个搜索词以及出现次数

## DStream的output操作以及foreachRDD详解

**output操作概览**

|  |  |
| --- | --- |
| **Output** | **Meaning** |
| print | 打印每个batch中的前10个元素，主要用于测试，或者是不需要执行什么output操作时，用于简单触发一下job。 |
| saveAsTextFile(prefix, [suffix]) | 将每个batch的数据保存到文件中。每个batch的文件的命名格式为：prefix-TIME\_IN\_MS[.suffix] |
| saveAsObjectFile | 同上，但是将每个batch的数据以序列化对象的方式，保存到SequenceFile中。 |
| saveAsHadoopFile | 同上，将数据保存到Hadoop文件中 |
| foreachRDD | 最常用的output操作，遍历DStream中的每个产生的RDD，进行处理。可以将每个RDD中的数据写入外部存储，比如文件、数据库、缓存等。通常在其中，是针对RDD执行action操作的，比如foreach。 |

DStream中的所有计算，都是由output操作触发的，比如print()。如果没有任何output操作，那么，压根儿就不会执行定义的计算逻辑。

此外，即使你使用了foreachRDD output操作，也必须在里面对RDD执行action操作，才能触发对每一个batch的计算逻辑。否则，光有foreachRDD output操作，在里面没有对RDD执行action操作，也不会触发任何逻辑。

**foreachRDD详解**

通常在foreachRDD中，都会创建一个Connection，比如JDBC Connection，然后通过Connection将数据写入外部存储。

误区一：在RDD的foreach操作外部，创建Connection

这种方式是错误的，因为它会导致Connection对象被序列化后传输到每个Task中。而这种Connection对象，实际上一般是不支持序列化的，也就无法被传输。

dstream.foreachRDD { rdd =>

val connection = createNewConnection()

rdd.foreach { record => connection.send(record)

}

}

误区二：在RDD的foreach操作内部，创建Connection

这种方式是可以的，但是效率低下。因为它会导致对于RDD中的每一条数据，都创建一个Connection对象。而通常来说，Connection的创建，是很消耗性能的。

dstream.foreachRDD { rdd =>

rdd.foreach { record =>

val connection = createNewConnection()

connection.send(record)

connection.close()

}

}

合理方式一：使用RDD的foreachPartition操作，并且在该操作内部，创建Connection对象，这样就相当于是，为RDD的每个partition创建一个Connection对象，节省资源的多了。

dstream.foreachRDD { rdd =>

rdd.foreachPartition { partitionOfRecords =>

val connection = createNewConnection()

partitionOfRecords.foreach(record => connection.send(record))

connection.close()

}

}

合理方式二：自己手动封装一个静态连接池，使用RDD的foreachPartition操作，并且在该操作内部，从静态连接池中，通过静态方法，获取到一个连接，使用之后再还回去。这样的话，甚至在多个RDD的partition之间，也可以复用连接了。而且可以让连接池采取懒创建的策略，并且空闲一段时间后，将其释放掉。

dstream.foreachRDD { rdd =>

rdd.foreachPartition { partitionOfRecords =>

val connection = ConnectionPool.getConnection()

partitionOfRecords.foreach(record => connection.send(record))

ConnectionPool.returnConnection(connection)

}

}

案例：改写UpdateStateByKeyWordCount，将每次统计出来的全局的单词计数，写入一份，到MySQL数据库中。

建表语句：

create table wordcount (

id integer auto\_increment primary key,

updated\_time timestamp NOT NULL default CURRENT\_TIMESTAMP on update CURRENT\_TIMESTAMP,

word varchar(255),

count integer

);

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/stream/PersistWordCount.java)

## 与Spark SQL结合使用之top3热门商品实时统计案例实战

Spark Streaming最强大的地方在于，可以与Spark Core、Spark SQL整合使用，之前已经通过transform、foreachRDD等算子看到，如何将DStream中的RDD使用Spark Core执行批处理操作。现在就来看看，如何将DStream中的RDD与Spark SQL结合起来使用。

案例：每隔10秒，统计最近60秒的，每个种类的每个商品的点击次数，然后统计出每个种类top3热门的商品。

[Java示例代码](WorkSpace/Spark-Projects/src/main/java/spark/stream/Top3HotProduct.java)

[Scala示例代码](WorkSpace/Spark-Projects/src/main/scala/spark1/stream/Top3HotProduct.scala)

## 缓存与持久化机制

与RDD类似，Spark Streaming也可以让开发人员手动控制，将数据流中的数据持久化到内存中。对DStream调用persist()方法，就可以让Spark Streaming自动将该数据流中的所有产生的RDD，都持久化到内存中。如果要对一个DStream多次执行操作，那么，对DStream持久化是非常有用的。因为多次操作，可以共享使用内存中的一份缓存数据。

对于基于窗口的操作，比如reduceByWindow、reduceByKeyAndWindow，以及基于状态的操作，比如updateStateByKey，默认就隐式开启了持久化机制。即Spark Streaming默认就会将上述操作产生的Dstream中的数据，缓存到内存中，不需要开发人员手动调用persist()方法。

对于通过网络接收数据的输入流，比如socket、Kafka、Flume等，默认的持久化级别，是将数据复制一份，以便于容错。相当于是，用的是类似MEMORY\_ONLY\_SER\_2。

与RDD不同的是，默认的持久化级别，统一都是要序列化的。

## Checkpoint机制

**Checkpoint机制概述**

每一个Spark Streaming应用，正常来说，都是要7 \* 24小时运转的，这就是实时计算程序的特点。因为要持续不断的对数据进行计算。因此，对实时计算应用的要求，应该是必须要能够对与应用程序逻辑无关的失败，进行容错。

如果要实现这个目标，Spark Streaming程序就必须将足够的信息checkpoint到容错的存储系统上，从而让它能够从失败中进行恢复。有两种数据需要被进行checkpoint：

1、元数据checkpoint——将定义了流式计算逻辑的信息，保存到容错的存储系统上，比如HDFS。当运行Spark Streaming应用程序的Driver进程所在节点失败时，该信息可以用于进行恢复。元数据信息包括了：

1.1 配置信息——创建Spark Streaming应用程序的配置信息，比如SparkConf中的信息。

1.2 DStream的操作信息——定义了Spark Stream应用程序的计算逻辑的DStream操作信息。

1.3 未处理的batch信息——那些job正在排队，还没处理的batch信息。

2、数据checkpoint——将实时计算过程中产生的RDD的数据保存到可靠的存储系统中。

对于一些将多个batch的数据进行聚合的，有状态的transformation操作，这是非常有用的。在这种transformation操作中，生成的RDD是依赖于之前的batch的RDD的，这会导致随着时间的推移，RDD的依赖链条变得越来越长。

要避免由于依赖链条越来越长，导致的一起变得越来越长的失败恢复时间，有状态的transformation操作执行过程中间产生的RDD，会定期地被checkpoint到可靠的存储系统上，比如HDFS。从而削减RDD的依赖链条，进而缩短失败恢复时，RDD的恢复时间。

一句话概括，元数据checkpoint主要是为了从driver失败中进行恢复；而RDD checkpoint主要是为了，使用到有状态的transformation操作时，能够在其生产出的数据丢失时，进行快速的失败恢复。

**何时启用Checkpoint机制**

1、使用了有状态的transformation操作——比如updateStateByKey，或者reduceByKeyAndWindow操作，被使用了，那么checkpoint目录要求是必须提供的，也就是必须开启checkpoint机制，从而进行周期性的RDD checkpoint。

2、要保证可以从Driver失败中进行恢复——元数据checkpoint需要启用，来进行这种情况的恢复。

要注意的是，并不是说，所有的Spark Streaming应用程序，都要启用checkpoint机制，如果即不强制要求从Driver失败中自动进行恢复，又没使用有状态的transformation操作，那么就不需要启用checkpoint。事实上，这么做反而是有助于提升性能的。

**如何启用Checkpoint机制**

1、对于有状态的transformation操作，启用checkpoint机制，定期将其生产的RDD数据checkpoint，是比较简单的。

可以通过配置一个容错的、可靠的文件系统（比如HDFS）的目录，来启用checkpoint机制，checkpoint数据就会写入该目录。使用StreamingContext的checkpoint()方法即可。然后，你就可以放心使用有状态的transformation操作了。

2、如果为了要从Driver失败中进行恢复，那么启用checkpoint机制，是比较复杂的。需要改写Spark Streaming应用程序。

当应用程序第一次启动的时候，需要创建一个新的StreamingContext，并且调用其start()方法，进行启动。当Driver从失败中恢复过来时，需要从checkpoint目录中记录的元数据中，恢复出来一个StreamingContext。

**为Driver失败的恢复机制重写程序（Java）**

JavaStreamingContextFactory contextFactory = new JavaStreamingContextFactory() {

@Override

public JavaStreamingContext create() {

JavaStreamingContext jssc = new JavaStreamingContext(...);

JavaDStream<String> lines = jssc.socketTextStream(...);

jssc.checkpoint(checkpointDirectory);

return jssc;

}

};

JavaStreamingContext context = JavaStreamingContext.getOrCreate(checkpointDirectory, contextFactory);

context.start();

context.awaitTermination();

**为Driver失败的恢复机制重写程序（Scala）**

def functionToCreateContext(): StreamingContext = {

val ssc = new StreamingContext(...)

val lines = ssc.socketTextStream(...)

ssc.checkpoint(checkpointDirectory)

ssc

}

val context = StreamingContext.getOrCreate(checkpointDirectory, functionToCreateContext \_)

context.start()

context.awaitTermination()

**配置spark-submit提交参数**

按照上述方法，进行Spark Streaming应用程序的重写后，当第一次运行程序时，如果发现checkpoint目录不存在，那么就使用定义的函数来第一次创建一个StreamingContext，并将其元数据写入checkpoint目录；当从Driver失败中恢复过来时，发现checkpoint目录已经存在了，那么会使用该目录中的元数据创建一个StreamingContext。

但是上面的重写应用程序的过程，只是实现Driver失败自动恢复的第一步。第二步是，必须确保Driver可以在失败时，自动被重启。

要能够自动从Driver失败中恢复过来，运行Spark Streaming应用程序的集群，就必须监Driver运行的过程，并且在它失败时将它重启。对于Spark自身的standalone模式，需要进行一些配置去supervise driver，在它失败时将其重启。

首先，要在spark-submit中，添加--deploy-mode参数，默认其值为client，即在提交应用的机器上启动Driver；但是，要能够自动重启Driver，就必须将其值设置为cluster；此外，需要添加--supervise参数。

使用上述第二步骤提交应用之后，就可以让driver在失败时自动被重启，并且通过checkpoint目录的元数据恢复StreamingContext。

**Checkpoint的说明**

将RDD checkpoint到可靠的存储系统上，会耗费很多性能。当RDD被checkpoint时，会导致这些batch的处理时间增加。因此，checkpoint的间隔，需要谨慎的设置。对于那些间隔很多的batch，比如1秒，如果还要执行checkpoint操作，则会大幅度削减吞吐量。而另外一方面，如果checkpoint操作执行的太不频繁，那就会导致RDD的lineage变长，又会有失败恢复时间过长的风险。

对于那些要求checkpoint的有状态的transformation操作，默认的checkpoint间隔通常是batch间隔的数倍，至少是10秒。使用DStream的checkpoint()方法，可以设置这个DStream的checkpoint的间隔时长。通常来说，将checkpoint间隔设置为窗口操作的滑动间隔的5~10倍，是个不错的选择。

## 部署、升级和监控应用程序

**部署应用程序**

1、有一个集群资源管理器，比如standalone模式下的Spark集群，Yarn模式下的Yarn集群等。

2、打包应用程序为一个jar包，课程中一直都有演示。

3、为executor配置充足的内存，因为Receiver接受到的数据，是要存储在Executor的内存中的，所以Executor必须配置足够的内存来保存接受到的数据。要注意的是，如果你要执行窗口长度为10分钟的窗口操作，那么Executor的内存资源就必须足够保存10分钟内的数据，因此内存的资源要求是取决于你执行的操作的。

4、配置checkpoint，如果你的应用程序要求checkpoint操作，那么就必须配置一个Hadoop兼容的文件系统（比如HDFS）的目录作为checkpoint目录.

5、配置driver的自动恢复，如果要让driver能够在失败时自动恢复，之前已经讲过，一方面，要重写driver程序，一方面，要在spark-submit中添加参数。

**部署应用程序：启用预写日志机制**

预写日志机制，简写为WAL，全称为Write Ahead Log。从Spark 1.2版本开始，就引入了基于容错的文件系统的WAL机制。如果启用该机制，Receiver接收到的所有数据都会被写入配置的checkpoint目录中的预写日志。这种机制可以让driver在恢复的时候，避免数据丢失，并且可以确保整个实时计算过程中，零数据丢失。

要配置该机制，首先要调用StreamingContext的checkpoint()方法设置一个checkpoint目录。然后需要将spark.streaming.receiver.writeAheadLog.enable参数设置为true。

然而，这种极强的可靠性机制，会导致Receiver的吞吐量大幅度下降，因为单位时间内，有相当一部分时间需要将数据写入预写日志。如果又希望开启预写日志机制，确保数据零损失，又不希望影响系统的吞吐量，那么可以创建多个输入DStream，启动多个Rceiver。

此外，在启用了预写日志机制之后，推荐将复制持久化机制禁用掉，因为所有数据已经保存在容错的文件系统中了，不需要在用复制机制进行持久化，保存一份副本了。只要将输入DStream的持久化机制设置一下即可，persist(StorageLevel.MEMORY\_AND\_DISK\_SER)。（之前讲过，默认是基于复制的持久化策略，\_2后缀）

**部署应用程序：设置Receiver接收速度**

如果集群资源有限，并没有大到，足以让应用程序一接收到数据就立即处理它，Receiver可以被设置一个最大接收限速，以每秒接收多少条单位来限速。

spark.streaming.receiver.maxRate和spark.streaming.kafka.maxRatePerPartition参数可以用来设置，前者设置普通Receiver，后者是Kafka Direct方式。

Spark 1.5中，对于Kafka Direct方式，引入了backpressure机制，从而不需要设置Receiver的限速，Spark可以自动估计Receiver最合理的接收速度，并根据情况动态调整。只要将spark.streaming.backpressure.enabled设置为true即可。

在企业实际应用场景中，通常是推荐用Kafka Direct方式的，特别是现在随着Spark版本的提升，越来越完善这个Kafka Direct机制。优点：1、不用receiver，不会独占集群的一个cpu core；2、有backpressure自动调节接收速率的机制.

**升级应用程序**

由于Spark Streaming应用程序都是7 \* 24小时运行的。因此如果需要对正在运行的应用程序，进行代码的升级，那么有两种方式可以实现：

1、升级后的Spark应用程序直接启动，先与旧的Spark应用程序并行执行。当确保新的应用程序启动没问题之后，就可以将旧的应用程序给停掉。但是要注意的是，这种方式只适用于，能够允许多个客户端读取各自独立的数据，也就是读取相同的数据。

2、小心地关闭已经在运行的应用程序，使用StreamingContext的stop()方法，可以确保接收到的数据都处理完之后，才停止。然后将升级后的程序部署上去，启动。这样，就可以确保中间没有数据丢失和未处理。因为新的应用程序会从老的应用程序未消费到的地方，继续消费。但是注意，这种方式必须是支持数据缓存的数据源才可以，比如Kafka、Flume等。如果数据源不支持数据缓存，那么会导致数据丢失。

注意：配置了driver自动恢复机制时，如果想要根据旧的应用程序的checkpoint信息，启动新的应用程序，是不可行的。需要让新的应用程序针对新的checkpoint目录启动，或者删除之前的checkpoint目录。

**监控应用程序**

当Spark Streaming应用启动时，Spark Web UI会显示一个独立的streaming tab，会显示Receiver的信息，比如是否活跃，接收到了多少数据，是否有异常等；还会显示完成的batch的信息，batch的处理时间、队列延迟等。这些信息可以用于监控spark streaming应用的进度。

Spark UI中，以下两个统计指标格外重要：

1、处理时间——每个batch的数据的处理耗时

2、调度延迟——一个batch在队列中阻塞住，等待上一个batch完成处理的时间

如果batch的处理时间，比batch的间隔要长的话，而且调度延迟时间持续增长，应用程序不足以使用当前设定的速率来处理接收到的数据，此时，可以考虑增加batch的间隔时间。

## 容错机制以及事务语义详解

**容错机制的背景**

要理解Spark Streaming提供的容错机制，先回忆一下Spark RDD的基础容错语义：

1、RDD，Ressilient Distributed Dataset，是不可变的、确定的、可重新计算的、分布式的数据集。每个RDD都会记住确定好的计算操作的血缘关系，（val lines = sc.textFile(hdfs file); val words = lines.flatMap(); val pairs = words.map(); val wordCounts = pairs.reduceByKey()）这些操作应用在一个容错的数据集上来创建RDD。

2、如果因为某个Worker节点的失败（挂掉、进程终止、进程内部报错），导致RDD的某个partition数据丢失了，那么那个partition可以通过对原始的容错数据集应用操作血缘，来重新计算出来。

3、所有的RDD transformation操作都是确定的，最后一个被转换出来的RDD的数据，一定是不会因为Spark集群的失败而丢失的。

Spark操作的通常是容错文件系统中的数据，比如HDFS。因此，所有通过容错数据生成的RDD也是容错的。然而，对于Spark Streaming来说，这却行不通，因为在大多数情况下，数据都是通过网络接收的（除了使用fileStream数据源）。要让Spark Streaming程序中，所有生成的RDD，都达到与普通Spark程序的RDD，相同的容错性，接收到的数据必须被复制到多个Worker节点上的Executor内存中，默认的复制因子是2。

基于上述理论，在出现失败的事件时，有两种数据需要被恢复：

1、数据接收到了，并且已经复制过——这种数据在一个Worker节点挂掉时，是可以继续存活的，因为在其他Worker节点上，还有它的一份副本。

2、数据接收到了，但是正在缓存中，等待复制的——因为还没有复制该数据，因此恢复它的唯一办法就是重新从数据源获取一份。

此外，还有两种失败是我们需要考虑的：

1、Worker节点的失败——任何一个运行了Executor的Worker节点的挂掉，都会导致该节点上所有在内存中的数据都丢失。如果有Receiver运行在该Worker节点上的Executor中，那么缓存的，待复制的数据，都会丢失。

2、Driver节点的失败——如果运行Spark Streaming应用程序的Driver节点失败了，那么显然SparkContext会丢失，那么该Application的所有Executor的数据都会丢失。

[RDD的基本容错原理](参考文件/原理图/RDD的基本容错原理.png)

**Spark Streaming容错语义的定义**

流式计算系统的容错语义，通常是以一条记录能够被处理多少次来衡量的。有三种类型的语义可以提供：

1、最多一次：每条记录可能会被处理一次，或者根本就不会被处理。可能有数据丢失。

2、至少一次：每条记录会被处理一次或多次，这种语义比最多一次要更强，因为它确保零数据丢失。但是可能会导致记录被重复处理几次。

3、一次且仅一次：每条记录只会被处理一次——没有数据会丢失，并且没有数据会处理多次。这是最强的一种容错语义。

**Spark Streaming的基础容错语义**

在Spark Streaming中，处理数据都有三个步骤：

1、接收数据：使用Receiver或其他方式接收数据。

2、计算数据：使用DStream的transformation操作对数据进行计算和处理。

3、推送数据：最后计算出来的数据会被推送到外部系统，比如文件系统、数据库等。

如果应用程序要求必须有一次且仅一次的语义，那么上述三个步骤都必须提供一次且仅一次的语义。每条数据都得保证，只能接收一次、只能计算一次、只能推送一次。Spark Streaming中实现这些语义的步骤如下：

1、接收数据：不同的数据源提供不同的语义保障。

2、计算数据：所有接收到的数据一定只会被计算一次，这是基于RDD的基础语义所保障的。即使有失败，只要接收到的数据还是可访问的，最后一个计算出来的数据一定是相同的。

3、推送数据：output操作默认能确保至少一次的语义，因为它依赖于output操作的类型，以及底层系统的语义支持（比如是否有事务支持等），但是用户可以实现它们自己的事务机制来确保一次且仅一次的语义。

**接收数据的容错语义**

1、基于文件的数据源

如果所有的输入数据都在一个容错的文件系统中，比如HDFS，Spark Streaming一定可以从失败进行恢复，并且处理所有数据。这就提供了一次且仅一次的语义，意味着所有的数据只会处理一次。

2、基于Receiver的数据源

对于基于Receiver的数据源，容错语义依赖于失败的场景和Receiver类型。

可靠的Receiver：这种Receiver会在接收到了数据，并且将数据复制之后，对数据源执行确认操作。如果Receiver在数据接收和复制完成之前，就失败了，那么数据源对于缓存的数据会接收不到确认，此时，当Receiver重启之后，数据源会重新发送数据，没有数据会丢失。

不可靠的Receiver：这种Receiver不会发送确认操作，因此当Worker或者Driver节点失败的时候，可能会导致数据丢失。

不同的Receiver，提供了不同的语义。如果Worker节点失败了，那么使用的是可靠的Receiver的话，没有数据会丢失。使用的是不可靠的Receiver的话，接收到，但是还没复制的数据，可能会丢失。如果Driver节点失败的话，所有过去接收到的，和复制过缓存在内存中的数据，全部会丢失。

要避免这种过去接收的所有数据都丢失的问题，Spark从1.2版本开始，引入了预写日志机制，可以将Receiver接收到的数据保存到容错存储中。如果使用可靠的Receiver，并且还开启了预写日志机制，那么可以保证数据零丢失。这种情况下，会提供至少一次的保障。（Kafka是可以实现可靠Receiver的）

从Spark 1.3版本开始，引入了新的Kafka Direct API，可以保证，所有从Kafka接收到的数据，都是一次且仅一次。基于该语义保障，如果自己再实现一次且仅一次语义的output操作，那么就可以获得整个Spark Streaming应用程序的一次且仅一次的语义。

[可靠Receiver的原理](参考文件/原理图/可靠Receiver的原理.png)

**输出数据的容错语义**

output操作，比如foreachRDD，可以提供至少一次的语义。那意味着，当Worker节点失败时，转换后的数据可能会被写入外部系统一次或多次。对于写入文件系统来说，这还是可以接受的，因为会覆盖数据。但是要真正获得一次且仅一次的语义，有两个方法：

1、幂等更新：多次写操作，都是写相同的数据，例如saveAs系列方法，总是写入相同的数据。

2、事务更新：所有的操作都应该做成事务的，从而让写入操作执行一次且仅一次。给每个batch的数据都赋予一个唯一的标识，然后更新的时候判定，如果数据库中还没有该唯一标识，那么就更新，如果有唯一标识，那么就不更新。

dstream.foreachRDD { (rdd, time) =>

rdd.foreachPartition { partitionIterator =>

val partitionId = TaskContext.get.partitionId()

val uniqueId = generateUniqueId(time.milliseconds, partitionId)

// partitionId和foreachRDD传入的时间，可以构成一个唯一的标识

}

}

**Storm的容错语义**

Storm首先，它可以实现消息的高可靠性，就是说，它有一个机制，叫做Acker机制，可以保证，如果消息处理失败，那么就重新发送。保证了，至少一次的容错语义。但是光靠这个，还是不行，数据可能会重复。

Storm提供了非常非常完善的事务机制，可以实现一次且仅一次的事务机制。事务Topology、透明的事务Topology、非透明的事务Topology，可以应用各种各样的情况。对实现一次且仅一次的这种语义的支持。用事务机制，可以获得它内部提供的一个唯一的id，然后基于这个id，就可以实现，output操作，输出，推送数据的时候，先判断，该数据是否更新过，如果没有的话，就更新；如果更新过，就不要重复更新了。

## 架构原理深度剖析

[Spark Streaming架构原理深度剖析](参考文件/原理图/Spark%20Streaming架构原理深度剖析.png)

## 数据接收原理剖析与源码分析

[数据接收原理剖析](参考文件/原理图/数据接收原理剖析.png)

## 数据处理原理剖析与源码分析（block与batch关系透彻解析）

[数据处理原理剖析](参考文件/原理图/数据处理原理剖析.png)

## 性能调优

**数据接收并行度调优**

通过网络接收数据时（比如Kafka、Flume），会将数据反序列化，并存储在Spark的内存中。如果数据接收称为系统的瓶颈，那么可以考虑并行化数据接收。每一个输入DStream都会在某个Worker的Executor上启动一个Receiver，该Receiver接收一个数据流。因此可以通过创建多个输入DStream，并且配置它们接收数据源不同的分区数据，达到接收多个数据流的效果。比如说，一个接收两个Kafka Topic的输入DStream，可以被拆分为两个输入DStream，每个分别接收一个topic的数据。这样就会创建两个Receiver，从而并行地接收数据，进而提升吞吐量。多个DStream可以使用union算子进行聚合，从而形成一个DStream。然后后续的transformation算子操作都针对该一个聚合后的DStream即可。

int numStreams = 5;

List<JavaPairDStream<String, String>> kafkaStreams = new ArrayList<JavaPairDStream<String, String>>(numStreams);

for (int i = 0; i < numStreams; i++) {

kafkaStreams.add(KafkaUtils.createStream(...));

}

JavaPairDStream<String, String> unifiedStream = streamingContext.union(kafkaStreams.get(0), kafkaStreams.subList(1, kafkaStreams.size()));

unifiedStream.print();

数据接收并行度调优，除了创建更多输入DStream和Receiver以外，还可以考虑调节block interval。通过参数，spark.streaming.blockInterval，可以设置block interval，默认是200ms。对于大多数Receiver来说，在将接收到的数据保存到Spark的BlockManager之前，都会将数据切分为一个一个的block。而每个batch中的block数量，则决定了该batch对应的RDD的partition的数量，以及针对该RDD执行transformation操作时，创建的task的数量。每个batch对应的task数量是大约估计的，即batch interval / block interval。

例如说，batch interval为2s，block interval为200ms，会创建10个task。如果你认为每个batch的task数量太少，即低于每台机器的cpu core数量，那么就说明batch的task数量是不够的，因为所有的cpu资源无法完全被利用起来。要为batch增加block的数量，那么就减小block interval。然而，推荐的block interval最小值是50ms，如果低于这个数值，那么大量task的启动时间，可能会变成一个性能开销点。

除了上述说的两个提升数据接收并行度的方式，还有一种方法，就是显式地对输入数据流进行重分区。使用inputStream.repartition(<number of partitions>)即可。这样就可以将接收到的batch，分布到指定数量的机器上，然后再进行进一步的操作。

**任务启动调优**

每秒钟启动的task过于多，比如每秒钟启动50个，那么发送这些task去Worker节点上的Executor的性能开销，会比较大，而且此时基本就很难达到毫秒级的延迟了。使用下述操作可以减少这方面的性能开销：

1、Task序列化：使用Kryo序列化机制来序列化task，可以减小task的大小，从而减少发送这些task到各个Worker节点上的Executor的时间。

2、执行模式：在Standalone模式下运行Spark，可以达到更少的task启动时间。

上述方式，也许可以将每个batch的处理时间减少100毫秒。从而从秒级降到毫秒级。

**数据处理并行度调优**

如果在计算的任何stage中使用的并行task的数量没有足够多，那么集群资源是无法被充分利用的。举例来说，对于分布式的reduce操作，比如reduceByKey和reduceByKeyAndWindow，默认的并行task的数量是由spark.default.parallelism参数决定的。你可以在reduceByKey等操作中，传入第二个参数，手动指定该操作的并行度，也可以调节全局的spark.default.parallelism参数。

**数据序列化调优**

数据序列化造成的系统开销可以由序列化格式的优化来减小。在流式计算的场景下，有两种类型的数据需要序列化。

1、输入数据：默认情况下，接收到的输入数据，是存储在Executor的内存中的，使用的持久化级别是StorageLevel.MEMORY\_AND\_DISK\_SER\_2。这意味着，数据被序列化为字节从而减小GC开销，并且会复制以进行executor失败的容错。因此，数据首先会存储在内存中，然后在内存不足时会溢写到磁盘上，从而为流式计算来保存所有需要的数据。这里的序列化有明显的性能开销——Receiver必须反序列化从网络接收到的数据，然后再使用Spark的序列化格式序列化数据。

2、流式计算操作生成的持久化RDD：流式计算操作生成的持久化RDD，可能会持久化到内存中。例如，窗口操作默认就会将数据持久化在内存中，因为这些数据后面可能会在多个窗口中被使用，并被处理多次。然而，不像Spark Core的默认持久化级别，StorageLevel.MEMORY\_ONLY，流式计算操作生成的RDD的默认持久化级别是StorageLevel.MEMORY\_ONLY\_SER ，默认就会减小GC开销。

在上述的场景中，使用Kryo序列化类库可以减小CPU和内存的性能开销。使用Kryo时，一定要考虑注册自定义的类，并且禁用对应引用的tracking（spark.kryo.referenceTracking）。

在一些特殊的场景中，比如需要为流式应用保持的数据总量并不是很多，也许可以将数据以非序列化的方式进行持久化，从而减少序列化和反序列化的CPU开销，而且又不会有太昂贵的GC开销。举例来说，如果你数秒的batch interval，并且没有使用window操作，那么你可以考虑通过显式地设置持久化级别，来禁止持久化时对数据进行序列化。这样就可以减少用于序列化和反序列化的CPU性能开销，并且不用承担太多的GC开销。

**batch interval调优（最重要）**

如果想让一个运行在集群上的Spark Streaming应用程序可以稳定，它就必须尽可能快地处理接收到的数据。换句话说，batch应该在生成之后，就尽可能快地处理掉。对于一个应用来说，这个是不是一个问题，可以通过观察Spark UI上的batch处理时间来定。batch处理时间必须小于batch interval时间。

基于流式计算的本质，batch interval对于，在固定集群资源条件下，应用能保持的数据接收速率，会有巨大的影响。例如，在WordCount例子中，对于一个特定的数据接收速率，应用业务可以保证每2秒打印一次单词计数，而不是每500ms。因此batch interval需要被设置得，让预期的数据接收速率可以在生产环境中保持住。

为你的应用计算正确的batch大小的比较好的方法，是在一个很保守的batch interval，比如5~10s，以很慢的数据接收速率进行测试。要检查应用是否跟得上这个数据速率，可以检查每个batch的处理时间的延迟，如果处理时间与batch interval基本吻合，那么应用就是稳定的。否则，如果batch调度的延迟持续增长，那么就意味应用无法跟得上这个速率，也就是不稳定的。因此你要想有一个稳定的配置，可以尝试提升数据处理的速度，或者增加batch interval。记住，由于临时性的数据增长导致的暂时的延迟增长，可以合理的，只要延迟情况可以在短时间内恢复即可。

**内存调优**

优化Spark应用的内存使用和GC行为，在Spark Core的调优中，已经讲过了。这里讲一下与Spark Streaming应用相关的调优参数。

Spark Streaming应用需要的集群内存资源，是由使用的transformation操作类型决定的。举例来说，如果想要使用一个窗口长度为10分钟的window操作，那么集群就必须有足够的内存来保存10分钟内的数据。如果想要使用updateStateByKey来维护许多key的state，那么你的内存资源就必须足够大。反过来说，如果想要做一个简单的map-filter-store操作，那么需要使用的内存就很少。

通常来说，通过Receiver接收到的数据，会使用StorageLevel.MEMORY\_AND\_DISK\_SER\_2持久化级别来进行存储，因此无法保存在内存中的数据会溢写到磁盘上。而溢写到磁盘上，是会降低应用的性能的。因此，通常是建议为应用提供它需要的足够的内存资源。建议在一个小规模的场景下测试内存的使用量，并进行评估。

内存调优的另外一个方面是垃圾回收。对于流式应用来说，如果要获得低延迟，肯定不想要有因为JVM垃圾回收导致的长时间延迟。有很多参数可以帮助降低内存使用和GC开销：

**1、DStream的持久化**：正如在“数据序列化调优”一节中提到的，输入数据和某些操作生产的中间RDD，默认持久化时都会序列化为字节。与非序列化的方式相比，这会降低内存和GC开销。使用Kryo序列化机制可以进一步减少内存使用和GC开销。进一步降低内存使用率，可以对数据进行压缩，由spark.rdd.compress参数控制（默认false）。

**2、清理旧数据**：默认情况下，所有输入数据和通过DStream transformation操作生成的持久化RDD，会自动被清理。Spark Streaming会决定何时清理这些数据，取决于transformation操作类型。例如，你在使用窗口长度为10分钟内的window操作，Spark会保持10分钟以内的数据，时间过了以后就会清理旧数据。但是在某些特殊场景下，比如Spark SQL和Spark Streaming整合使用时，在异步开启的线程中，使用Spark SQL针对batch RDD进行执行查询。那么就需要让Spark保存更长时间的数据，直到Spark SQL查询结束。可以使用streamingContext.remember()方法来实现。

**3、CMS垃圾回收器**：使用并行的mark-sweep垃圾回收机制，被推荐使用，用来保持GC低开销。虽然并行的GC会降低吞吐量，但是还是建议使用它，来减少batch的处理时间（降低处理过程中的gc开销）。如果要使用，那么要在driver端和executor端都开启。在spark-submit中使用--driver-java-options设置；使用spark.executor.extraJavaOptions参数设置。-XX:+UseConcMarkSweepGC。