新闻大数据平台及数据仓库项目

(作者：罗圣杰)

# 第1章 项目业务及技术框架

## 1.1 概述

随着人们对手机依赖程度的增加，手机已经成为了绝大多数人社交、购物、休闲娱乐、学习、发表见解、获取时事新闻等需求的主要实现渠道。移动端已经是绝大数用户流量的入口

* 当一款APP上线后，为了更好地改进这款APP，公司需要了解到这款APP的详细使用情况，例如新增用户数、活跃用户数等；为了了解用户对于这款APP不同版本的接受程度，需要了解这款APP在用户群体中的版本分布情况。
* 目前移动端作为手机的巨大入口，一些工具类的app用户停留时长不高。为了延长用户在手机上的停留时长，方法之一是在app中嵌入内容。信息流包括音频，视频，文字，图片等。丰富的内容，能极大限度的留住用户。这需要对用户进行个性化分析，所以信息流推荐系统就起到非常大的作用。

以上这些需求的实现，都要建立在一个稳定的分布式日志采集和统计分析系统之上。

在大数据技术趋于成熟的今天，大数据技术的发展使采集海量用户信息并分析用户行为进而有目的的改进APP这一需求的实现成为可能，本项目就致力于打造一套成熟的日志数据统计分析系统，通过离线数据分析系统和实时数据分析系统两个模块的协同作用，完成APP各项指标的分析，同时为信息流提供相关的数据支持与实现。

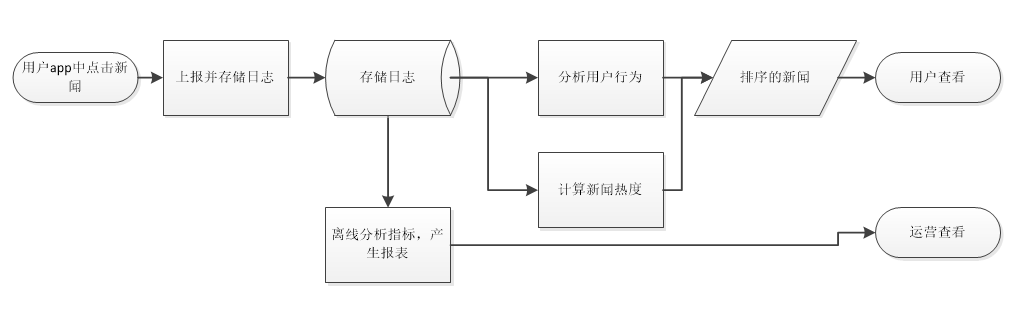
## 1.2 项目需求

### 1.2.1 日志业务流程

现在我们需要这样一个系统，来承载信息流，实现下面的功能：

分析用户行为功能：

1. 收集用户观看新闻的日志
2. 每日分析用户的行为
3. 实时对用户的行为作出反馈和推荐



图为业务流程

**对象说明：**

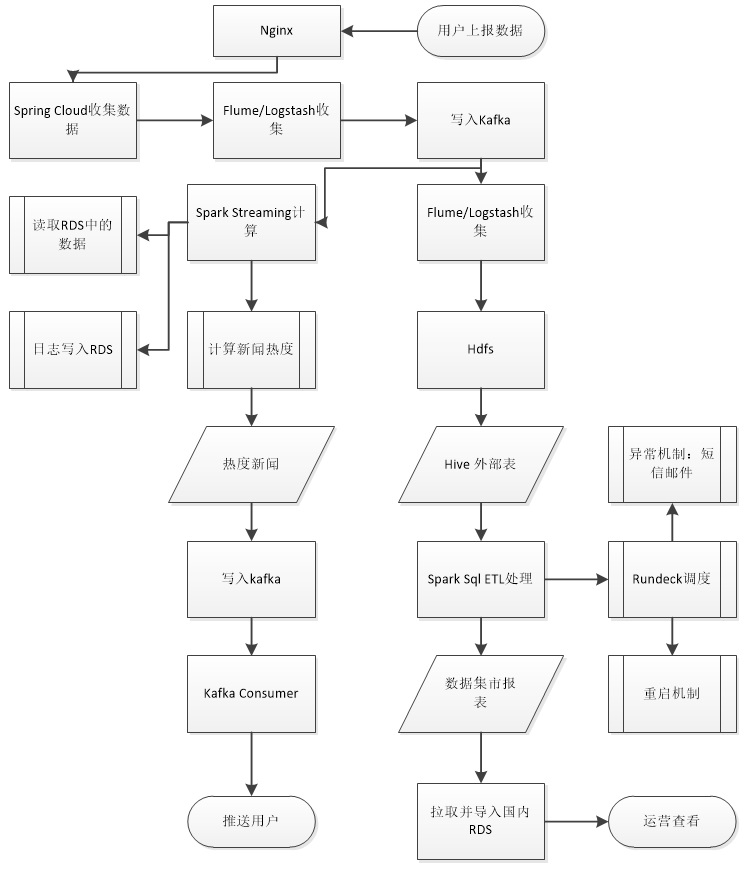
* 用户:用户由设备上传的唯一ID辨别。Id由前端产生，用来标注每一个用户。
* 新闻:包括图片，文字组合成的新闻，带有唯一id，计算过程以id处理
* 新闻质量:与点击率相关联,第三方爬取计算
* 用户中心：数据仓库中维护所有点击过新闻的用户。
* 新闻中心：数据仓库中维护所有新闻的集合



图为数据流程

## 1.3 项目框架

### 1.3.1 数据仓库流程



图：整个数据仓库的数据流程

**注意：**

1. **其中RDS可以使用Elasticsearch**
2. **Logstash与Flume可以互换**
3. **Spring Boot是Tomcat的替代方案，也可以用tomcat+Javaweb**
4. **Flume，kafka，spark可以使用老版本**

**热度新闻：新用户**

**个性化推荐：产生过点击的用户**

根据图中的项目框架可知，本项目由离线数据处理系统和实时数据处理系统两个部分组成。

离线数据处理系统与实时数据处理系统共用一套日志采集系统，日志采集系统采用了双层Flume拓扑结构，第一层实现数据的采集，第二层实现数据的集中聚合处理；手机APP客户端的日志数据被日志采集系统采集完成后，被分别输送到离线和实时数据处理系统中进行处理。

首先，APP客户端的日志数据定时（如0.5小时/次）服务器发送日志，服务器端通过Nginx实现负载均衡，Nginx将日志数据负载均衡到多给web应用上，然后通过日志写在磁盘。

随后，双层Flume架构中的第一层数据采集Flume将对应的Tomcat生成的日志文件采集到其拓扑结构中，随后多个第一层数据采集Flume的数据汇总到第二层的数据聚合Flume上，完成数据的聚合和集中处理。

然后，第二层的数据聚合Flume根据日志数据的类型，将日志数据发送到不同的Kafka主题中，在Kafka中完成数据的分布式存储。

最后，离线数据处理系统和实时数据处理系统分别从Kafka中消费消息，各自完成数据的离线分析处理和实时分析处理。

### 1.3.2 技术组件



数据收集层组件

用户在使用APP的过程中，会不断产生日志数据。当用户后台运行的APP的时候，会产生后台日志。打开APP的时候，会产生前台APP启动日志，启动日志中记录了APP启动时间，运行时长等信息；

APP中运行的服务会定时将手机系统中产生的日志提取出来，然后经过一定的处理，例如合并等，目的是减少对服务端的压力，数据合并完成后，**在APP中有一个数据库，合并的数据会放入这一数据库中**，因为用户有可能断网，导致数据不能及时的发送到服务器，因此先存储在APP的本地数据库中，等到用户联网的时候再一并发送出来。

当手机客户端需要发送数据时，会对数据进行对称加密和压缩，压缩可以减少服务器的带宽，然后调用服务器的接口上传数据，服务端对数据进行解压和解密，然后用SLF4J写入本地磁盘中。（**写入本地磁盘的目的是让业务系统与采集系统完全解耦，同时缓冲峰值数据**）

**所以有几类产品：**



* 数据集成:Flume，DataX，Kafka
* 数据计算:Spark，Hive
* 数据存储:Hive，S3，Mysql,HDFS
* 数据管理:Rundeck，元数据管理
* 数据查询:Presto,Impala

### 1.5.4 产品选型列表

|  |  |
| --- | --- |
| **产品** | **版本** |
| **Hadoop** | 2.7.7 |
| **Spring Cloud** | Finchley.SR1 |
| **Spark** | 2.3.1 |
| **Flume** | 1.8 |
| **Kafka** | 1.1.1 |
| **Hive** | 2.3.3 |
| **Presto** | presto-server-0.211, |
| **Mysql** | Community Server 8.0.13 |
| **Java** | 1.8 |

### 1.5.4 选型注意事项

* Hadoop选择稳定的版本,不能选最新的beta版本。
* Hive 3点几的版本只支持 hadoop 3点几，所以选2点几的版本
* Java 1.8的版本相对稳定，选1.8
* Spark的版本需要和hadoop兼容
* Kafka需要支持direct的模式

# 第2章 数据平台框架搭建

## 3.1 Hadoop

Hadoop 先将三台机器配置互相免密登陆。

安装教程

<https://www.jianshu.com/p/b33568c4f84c>

Hadoop 不能直接用root用户来运行，需要新建一个用户。项目中新建了一个hdfs用户，用来运行所有的框架。一个worker用户，用来跑所有的脚本。

并且我们要将 hdfs 用户添加到 sudo中

sudo visudo

在里面修改成下面：

## Same thing without a password

# %wheel ALL=(ALL) NOPASSWD: ALL

hdfs ALL=(ALL) NOPASSWD:ALL

work ALL=(ALL) NOPASSWD:ALL

后面我们先用hdfs 来进行所组件的安装

设置环境变量

### 启动

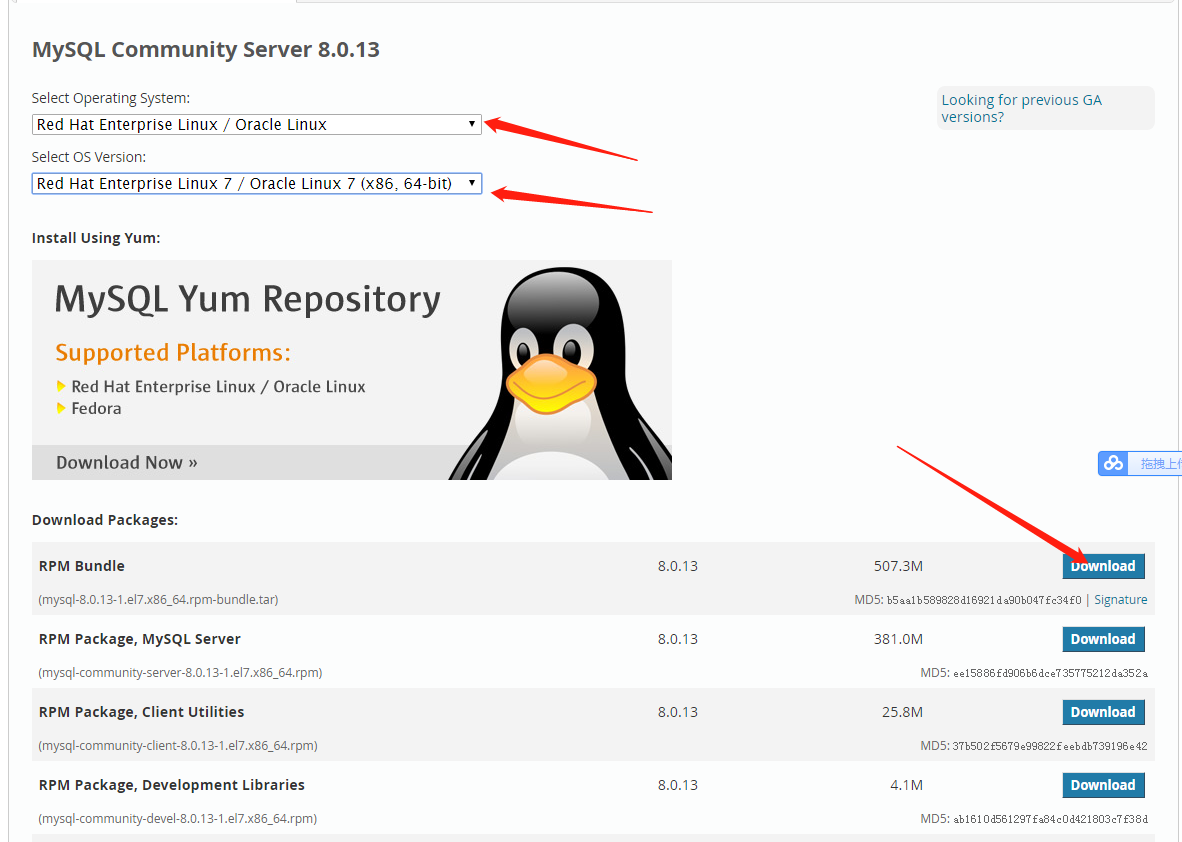
sbin/start-dfs.sh

sbin/start-yarn.sh

## Mysql

### 3.2.1 Mysql安装

直接从<https://dev.mysql.com/downloads/mysql/> 下载



Centos 是红帽的内核，所以选择redhat，然后选择RPM bundle 进行下载。使用命令rpm –ivh xxxx-community-server.rpm.中间提示的依赖包，直接安装相关的依赖包

如果出现

Removing: 1:mariadb-server-5.5.35-3.el7.x86\_64 (@anaconda)

mariadb-server = 1:5.5.35-3.el7

Obsoleted By: mysql-community-server-5.6.25-2.el7.x86\_64 (mysql56-community)

需要执行命令

# yum -y remove mariadb-libs

### 3.2.2 启动

sudo service mysqld start

### 3.2.3 初始密码

#grep 'temporary password' /var/log/mysqld.log | awk '{print $NF}'

通过这个命令查看

完成之后 mysql –uroot –p 登陆。

设置密码复杂度为0，就是最低，设置密码长度为你想要的长度（因为mysql比较新的版本都有密码强壮度检验插件）：

set global validate\_password\_policy=0;

set global validate\_password\_length=6;

修改密码成想要的密码：

SET PASSWORD FOR 'root'@'localhost' =PASSWORD('root');

或者：

ALTER USER 'root'@'localhost' IDENTIFIED BY 'root';

## Hive

访问网站 <https://hive.apache.org/downloads.html>

下载hive。

先登陆mysql，创建对应的用户

CREATE USER 'hive'@'%' IDENTIFIED BY '123456';

drop database if exists hive;

create database hive DEFAULT CHARSET utf8 COLLATE utf8\_general\_ci;

GRANT ALL ON hive.\* TO 'hive'@'%';

flush privileges;

### 修改 hive-site.xml

cp hive-default.xml.template hive-site.xml

将文件中的所有 ${system:java.io.tmpdir}替换成  /tmp/hive

将文件中所有的 ${system:user.name} 替换成hdfs

<property>

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

<value>jdbc:mysql://dn130:3306/hive?createDatabaseIfNotExist=true</value>

<description>

JDBC connect string for a JDBC metastore.

To use SSL to encrypt/authenticate the connection, provide database-specific SSL flag in the connection URL.

For example, jdbc:postgresql://myhost/db?ssl=true for postgres database.

</description>

</property>

搜索javax.jdo.option.ConnectionDriverName，将该name对应的value修改为MySQL驱动类路径:

<property>

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

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

<description>Driver class name for a JDBC metastore</description>

</property>

<property>

搜索javax.jdo.option.ConnectionUserName，将对应的value修改为MySQL数据库登录名:

<property>

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

<value>hive</value>

<description>Username to use against metastore database</description>

</property>

搜索javax.jdo.option.ConnectionPassword，将对应的value修改为MySQL数据库的登录密码:

<property>

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

<value>hive</value>

<description>password to use against metastore database</description>

</property>

搜索hive.metastore.schema.verification，将对应的value修改为false：

<property>

<name>hive.metastore.schema.verification</name>

<value>false</value>

<description>

Enforce metastore schema version consistency.

True: Verify that version information stored in is compatible with one from Hive jars. Also disable automatic

schema migration attempt. Users are required to manually migrate schema after Hive upgrade which ensures

proper metastore schema migration. (Default)

False: Warn if the version information stored in metastore doesn't match with one from in Hive jars.

</description>

</property>

自动建表设置成true

<property>

<name>datanucleus.schema.autoCreateAll</name>

<value>true</value>

<description>Auto creates necessary schema on a startup if one doesn't exist. Set this to false, after creating it once.To enable auto create also set hive.metastore.schema.verification=false. Auto creation is not recommended for production use cases, run schematool command instead.</description>

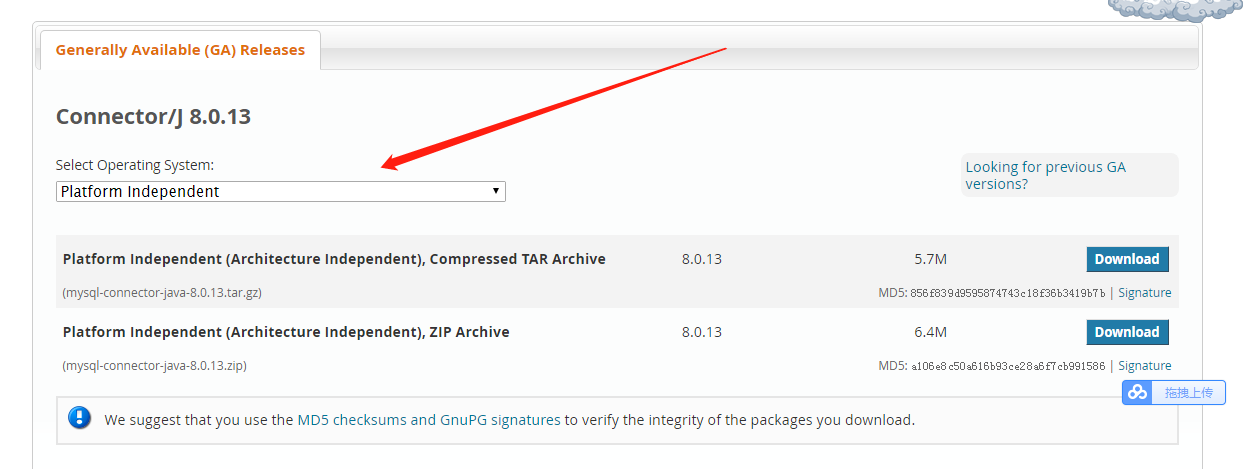
</property>

### 依赖包

另外需要下载mysql 的jar包

<https://dev.mysql.com/downloads/connector/j/>

选



然后放入某个路径下面。/mnt/deps/libs/mysql-connector-java-8.0.13.jar

### 设置环境变量

cp hive-env.sh.template hive-env.sh 修改hive-env.sh

# Set HADOOP\_HOME to point to a specific hadoop install directory

export HADOOP\_HOME=/mnt/apps/hadoop-2.7.7

# Hive Configuration Directory can be controlled by:

export HIVE\_CONF\_DIR=/mnt/apps/apache-hive-2.3.3-bin/conf

# Folder containing extra libraries required for hive compilation/execution can be controlled by:

export HIVE\_AUX\_JARS\_PATH=/mnt/deps/libs/mysql-connector-java-8.0.13.jar

### 设置Tez

修改 hive-site.xml 为

<property>

<name>hive.execution.engine</name>

<value>tez</value>

</property>

启动的时候出现这个错误

Caused by: com.mysql.cj.exceptions.InvalidConnectionAttributeException: The server time zone value 'EDT' is unrecognized or represents more than one time zone. You must configure either the server or JDBC driver (via the serverTimezone configuration property) to use a more specifc time zone value if you want to utilize time zone support.

at sun.reflect.GeneratedConstructorAccessor43.newInstance(Unknown Source)

at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)

at java.lang.reflect.Constructor.newInstance(Constructor.java:423)

at com.mysql.cj.exceptions.ExceptionFactory.createException(ExceptionFactory.java:61)

at com.mysql.cj.exceptions.ExceptionFactory.createException(ExceptionFactory.java:85)

at com.mysql.cj.util.TimeUtil.getCanonicalTimezone(TimeUtil.java:132)

at com.mysql.cj.protocol.a.NativeProtocol.configureTimezone(NativeProtocol.java:2234)

at com.mysql.cj.protocol.a.NativeProtocol.initServerSession(NativeProtocol.java:2258)

at com.mysql.cj.jdbc.ConnectionImpl.initializePropsFromServer(ConnectionImpl.java:1319)

at com.mysql.cj.jdbc.ConnectionImpl.connectOneTryOnly(ConnectionImpl.java:966)

at com.mysql.cj.jdbc.ConnectionImpl.createNewIO(ConnectionImpl.java:825)

在mysql 里面设置

SET @@session.time\_zone = '+00:00';

SET @@global.time\_zone = '+00:00';

这后面是时区，也可以设置成其他的时区

### 小结

1. Tez为什么优于MR？



传统的MR（包括Hive，Pig和直接编写MR程序）。假设有四个有依赖关系的MR作业（1个较为复杂的Hive SQL语句或者Pig脚本可能被翻译成4个有依赖关系的MR作业）或者用Oozie描述的4个有依赖关系的作业，运行过程如下（其中，绿色是Reduce Task，需要写HDFS）：

 云状表示写屏蔽（write barrier，一种内核机制，持久写）

Tez可以将多个有依赖的作业转换为一个作业（这样只需写一次HDFS，且中间节点较少），从而大大提升DAG作业的性能

1. MetaStore的作用

metadata即元数据。元数据包含用Hive创建的database、tabel等的元信息。  
元数据存储在关系型数据库中。如Derby、MySQL等。

Metastore的作用是：客户端连接metastore服务，metastore再去连接MySQL数据库来存取元数据。有了metastore服务，就可以有多个客户端同时连接，而且这些客户端不需要知道MySQL数据库的用户名和密码，只需要连接metastore 服务即可。

## Zookeeper

### 集群启动

ZkServer.sh start

## Kafka

### 安装

配置文件

############################# Server Basics #############################

# The id of the broker. This must be set to a unique integer for each broker.

broker.id=0

############################# Socket Server Settings #############################

# The address the socket server listens on. It will get the value returned from

# java.net.InetAddress.getCanonicalHostName() if not configured.

# FORMAT:

# listeners = listener\_name://host\_name:port

# EXAMPLE:

# listeners = PLAINTEXT://your.host.name:9092

#listeners=PLAINTEXT://:9092

# Hostname and port the broker will advertise to producers and consumers. If not set,

# it uses the value for "listeners" if configured. Otherwise, it will use the value

# returned from java.net.InetAddress.getCanonicalHostName().

#advertised.listeners=PLAINTEXT://your.host.name:9092

# Maps listener names to security protocols, the default is for them to be the same. See the config documentation for more details

#listener.security.protocol.map=PLAINTEXT:PLAINTEXT,SSL:SSL,SASL\_PLAINTEXT:SASL\_PLAINTEXT,SASL\_SSL:SASL\_SSL

# The number of threads that the server uses for receiving requests from the network and sending responses to the network

num.network.threads=3

# The number of threads that the server uses for processing requests, which may include disk I/O

num.io.threads=8

# The send buffer (SO\_SNDBUF) used by the socket server

socket.send.buffer.bytes=102400

# The receive buffer (SO\_RCVBUF) used by the socket server

socket.receive.buffer.bytes=102400

# The maximum size of a request that the socket server will accept (protection against OOM)

socket.request.max.bytes=104857600

############################# Log Basics #############################

# A comma separated list of directories under which to store log files

log.dirs=/tmp/kafka-logs

# The default number of log partitions per topic. More partitions allow greater

# parallelism for consumption, but this will also result in more files across

# the brokers.

num.partitions=1

# The number of threads per data directory to be used for log recovery at startup and flushing at shutdown.

# This value is recommended to be increased for installations with data dirs located in RAID array.

num.recovery.threads.per.data.dir=1

############################# Internal Topic Settings #############################

# The replication factor for the group metadata internal topics "\_\_consumer\_offsets" and "\_\_transaction\_state"

# For anything other than development testing, a value greater than 1 is recommended for to ensure availability such as 3.

offsets.topic.replication.factor=1

transaction.state.log.replication.factor=1

transaction.state.log.min.isr=1

############################# Log Flush Policy #############################

# Messages are immediately written to the filesystem but by default we only fsync() to sync

# the OS cache lazily. The following configurations control the flush of data to disk.

# There are a few important trade-offs here:

# 1. Durability: Unflushed data may be lost if you are not using replication.

# 2. Latency: Very large flush intervals may lead to latency spikes when the flush does occur as there will be a lot of data to flush.

# 3. Throughput: The flush is generally the most expensive operation, and a small flush interval may lead to excessive seeks.

# The settings below allow one to configure the flush policy to flush data after a period of time or

# every N messages (or both). This can be done globally and overridden on a per-topic basis.

# The number of messages to accept before forcing a flush of data to disk

#log.flush.interval.messages=10000

# The maximum amount of time a message can sit in a log before we force a flush

#log.flush.interval.ms=1000

############################# Log Retention Policy #############################

# The following configurations control the disposal of log segments. The policy can

# be set to delete segments after a period of time, or after a given size has accumulated.

# A segment will be deleted whenever \*either\* of these criteria are met. Deletion always happens

# from the end of the log.

# The minimum age of a log file to be eligible for deletion due to age

log.retention.hours=168

# A size-based retention policy for logs. Segments are pruned from the log unless the remaining

# segments drop below log.retention.bytes. Functions independently of log.retention.hours.

#log.retention.bytes=1073741824

# The maximum size of a log segment file. When this size is reached a new log segment will be created.

log.segment.bytes=1073741824

# The interval at which log segments are checked to see if they can be deleted according

# to the retention policies

log.retention.check.interval.ms=300000

############################# Zookeeper #############################

# Zookeeper connection string (see zookeeper docs for details).

# This is a comma separated host:port pairs, each corresponding to a zk

# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".

# You can also append an optional chroot string to the urls to specify the

# root directory for all kafka znodes.

zookeeper.connect=localhost:2181

# Timeout in ms for connecting to zookeeper

zookeeper.connection.timeout.ms=6000

############################# Group Coordinator Settings #############################

# The following configuration specifies the time, in milliseconds, that the GroupCoordinator will delay the initial consumer rebalance.

# The rebalance will be further delayed by the value of group.initial.rebalance.delay.ms as new members join the group, up to a maximum of max.poll.interval.ms.

# The default value for this is 3 seconds.

# We override this to 0 here as it makes for a better out-of-the-box experience for development and testing.

# However, in production environments the default value of 3 seconds is more suitable as this will help to avoid unnecessary, and potentially expensive, rebalances during application startup.

group.initial.rebalance.delay.ms=0

#此处为了测试，将此开关打开

delete.topic.enable=true

### 启动

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

### 日志路径

看一下Kafka的log4j的文件

log4j.rootLogger=INFO, stdout, kafkaAppender

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=[%d] %p %m (%c)%n

log4j.appender.kafkaAppender=org.apache.log4j.DailyRollingFileAppender

log4j.appender.kafkaAppender.DatePattern='.'yyyy-MM-dd-HH

log4j.appender.kafkaAppender.File=${kafka.logs.dir}/server.log

log4j.appender.kafkaAppender.layout=org.apache.log4j.PatternLayout

log4j.appender.kafkaAppender.layout.ConversionPattern=[%d] %p %m (%c)%n

log4j.appender.stateChangeAppender=org.apache.log4j.DailyRollingFileAppender

log4j.appender.stateChangeAppender.DatePattern='.'yyyy-MM-dd-HH

log4j.appender.stateChangeAppender.File=${kafka.logs.dir}/state-change.log

log4j.appender.stateChangeAppender.layout=org.apache.log4j.PatternLayout

log4j.appender.stateChangeAppender.layout.ConversionPattern=[%d] %p %m (%c)%n

log4j.appender.requestAppender=org.apache.log4j.DailyRollingFileAppender

log4j.appender.requestAppender.DatePattern='.'yyyy-MM-dd-HH

log4j.appender.requestAppender.File=${kafka.logs.dir}/kafka-request.log

log4j.appender.requestAppender.layout=org.apache.log4j.PatternLayout

log4j.appender.requestAppender.layout.ConversionPattern=[%d] %p %m (%c)%n

log4j.appender.cleanerAppender=org.apache.log4j.DailyRollingFileAppender

log4j.appender.cleanerAppender.DatePattern='.'yyyy-MM-dd-HH

log4j.appender.cleanerAppender.File=${kafka.logs.dir}/log-cleaner.log

log4j.appender.cleanerAppender.layout=org.apache.log4j.PatternLayout

log4j.appender.cleanerAppender.layout.ConversionPattern=[%d] %p %m (%c)%n

log4j.appender.controllerAppender=org.apache.log4j.DailyRollingFileAppender

log4j.appender.controllerAppender.DatePattern='.'yyyy-MM-dd-HH

log4j.appender.controllerAppender.File=${kafka.logs.dir}/controller.log

log4j.appender.controllerAppender.layout=org.apache.log4j.PatternLayout

log4j.appender.controllerAppender.layout.ConversionPattern=[%d] %p %m (%c)%n

log4j.appender.authorizerAppender=org.apache.log4j.DailyRollingFileAppender

log4j.appender.authorizerAppender.DatePattern='.'yyyy-MM-dd-HH

log4j.appender.authorizerAppender.File=${kafka.logs.dir}/kafka-authorizer.log

log4j.appender.authorizerAppender.layout=org.apache.log4j.PatternLayout

log4j.appender.authorizerAppender.layout.ConversionPattern=[%d] %p %m (%c)%n

# Change the two lines below to adjust ZK client logging

log4j.logger.org.I0Itec.zkclient.ZkClient=INFO

log4j.logger.org.apache.zookeeper=INFO

# Change the two lines below to adjust the general broker logging level (output to server.log and stdout)

log4j.logger.kafka=INFO

log4j.logger.org.apache.kafka=INFO

# Change to DEBUG or TRACE to enable request logging

log4j.logger.kafka.request.logger=WARN, requestAppender

log4j.additivity.kafka.request.logger=false

# Uncomment the lines below and change log4j.logger.kafka.network.RequestChannel$ to TRACE for additional output

# related to the handling of requests

#log4j.logger.kafka.network.Processor=TRACE, requestAppender

#log4j.logger.kafka.server.KafkaApis=TRACE, requestAppender

#log4j.additivity.kafka.server.KafkaApis=false

log4j.logger.kafka.network.RequestChannel$=WARN, requestAppender

log4j.additivity.kafka.network.RequestChannel$=false

log4j.logger.kafka.controller=TRACE, controllerAppender

log4j.additivity.kafka.controller=false

log4j.logger.kafka.log.LogCleaner=INFO, cleanerAppender

log4j.additivity.kafka.log.LogCleaner=false

log4j.logger.state.change.logger=TRACE, stateChangeAppender

log4j.additivity.state.change.logger=false

# Access denials are logged at INFO level, change to DEBUG to also log allowed accesses

log4j.logger.kafka.authorizer.logger=INFO, authorizerAppender

log4j.additivity.kafka.authorizer.logger=false

### 小结

1. Kafka中的数据如何查看。

直接找到相关的文件，然后vim直接查看。

1. 如何压测

我们可以用kafka 下面的脚本，对kafka进行压测。然后在压测的时候，看到哪个地方出现了瓶颈。（io,cpu,内存,网络）一般都是磁盘io达到瓶颈.结果一般都是M/s

1. 如何设计kafka集群的大小

先要预估一天大概有多少数据量，然后再去用压测的标准计算一天能承受多少的负载。然后在上面乘以一个3倍，考虑读和写的情况。

1. Kafka的日志留存设置。

Kafka的日志留存一般有两种，一种是时间，一种是大小。我们设置至少保留3天的数据量，取中间的最大值。大小和性能并无太大关系，都是O(1) ，所以越大越好。

1. Kafka监控

公司自带的agent部署在每台服务器上面，出现程序奔溃则会报警。

负载过高也会报警（磁盘，cpu，内存等）

## Spark

# 第3章 源数据

## 手机上报日志

### 基本格式

* 公共字段:基本所有安卓手机都包含的字段
* 业务字段:埋点上报的字段，有具体的业务类型

下面就是一个示例，表示业务字段的上传。

{

"ap":"xxxxx",//产品字段 app key

"cm": { //公共字段

"uid": "", // (String) 客户端唯一标识 uuid

"vc": "1", // (String) versionCode，程序版本号

"vn": "1.0", // (String) versionName，程序版本名

"l": "zh", // (String) 系统语言

"sr": "", // (String) 渠道号，应用从哪个渠道来的。

"os": "7.1.1", // (String) Android系统版本

"ar": "CN", // (String) 区域

"md": "BBB100-1", // (String) 手机型号

"ba": "blackberry", // (String) 手机品牌

"sv": "V2.2.1", // (String) sdkVersion

"g": "", // (String) gmail

"hw": "1620x1080", // (String) heightXwidth，屏幕宽高

"t": "1506047606608", // (String) 客户端日志产生时的时间

"nw": "WIFI", // (String) 网络模式

"ln": 0, // (double) lng经度

"la": 0 // (double) lat 纬度

},

"et": [ //事件

{

"ett": "1506047605364", //客户端事件产生时间

"en": "request", //事件名称

"kv": { //事件结果，以key-value形式自行定义

"your key1": "your value1",

"your key2": "your value2",

"your key n": "your value n"

}

}

]

}

示例日志（加上ip和时间戳）：

**192.168.1.103，1540934156385|{**http://tool.oschina.net/tools/json_format/Expanded.gif

**"ap"**: "browser"**,**

**"cm"**: **{**http://tool.oschina.net/tools/json_format/Expanded.gif

**"uid"**: "1234"**,**

**"vc"**: "2"**,**

**"vn"**: "1.0"**,**

**"la"**: "EN"**,**

**"sr"**: ""**,**

**"os"**: "7.1.1"**,**

**"ar"**: "CN"**,**

**"md"**: "BBB100-1"**,**

**"ba"**: "blackberry"**,**

**"sv"**: "V2.2.1"**,**

**"g"**: "abc@gmail.com"**,**

**"hw"**: "1620x1080"**,**

**"t"**: "1506047606608"**,**

**"nw"**: "WIFI"**,**

**"ln"**: 0

**},**

**"et"**: **[**http://tool.oschina.net/tools/json_format/Expanded.gif

**{**http://tool.oschina.net/tools/json_format/Expanded.gif

**"ett"**: "1506047605364"**,**

**"en"**: "request"**,**

**"kv"**: **{**http://tool.oschina.net/tools/json_format/Expanded.gif

**"url"**: "www.baidu.com"**,**

**"click"**: "1"

**}**

**}**

**]**

**}**

**}**

下面是各个埋点日志格式。其中新闻点击属于信息流的范畴

### 新闻点击/展示（display）

事件标签：display

|  |  |  |
| --- | --- | --- |
| **标签** | | **含义** |
| **action** | 动作：曝光新闻=1，点击新闻=2， | |
| **newsid** | 新闻ID（服务端下发的ID） | |
| **place** | 顺序（第几条新闻，第一条为0，第二条为1，如此类推） | |
| **showtype** | 新闻类型：1、正版内容，2、爬虫内容  ，3、其它内容 | |
| **copyright** | 版权方（内容源头，如NewYorkTimes接入的来源） | |
| **content\_provider** | 合作方（合作方名称，如Reuters） | |
| **newstype** | Type: 1- 图文 2-图集 3-段子 4-GIF 5-视频 6-调查 7-纯文  8-视频+图文  9-GIF+图文  0-其他 | |
| **extend1** | 曝光类型：1 - 首次曝光 2-重复曝光（没有使用） | |
| **extend2** | 扩展字段 Extend2：新闻样式： | |
|  | 0、无图 | |
|  | 1、一张大图 | |
|  | 2、两张图 | |
|  | 3、三张小图 | |
|  | 4、一张小图 | |
|  | 5、一张大图两张小图 | |
| **category** | 分类ID（服务端定义的分类ID） | |

### 新闻详情页(newsdetailpro)

事件标签：newsdetailpro

|  |  |  |
| --- | --- | --- |
| **标签** | | **含义** |
| **entry** | 页面入口来源：应用首页=1、push=2、详情页相关推荐=3 | |
| **action** | 动作：开始加载=1，加载成功=2（pv），加载失败=3, 退出页面=4 | |
| **newsid** | 新闻ID（服务端下发的ID） | |
| **showtype** | 新闻类型：1、正版内容 ，2、爬虫内容   （无论开始加载/加载成功/加载失败/退出页面 都会上报1或2）， 3、其它内容文章加载失败/页面加载失败无法返回类型） | |
| **copyright** | 版权方（内容源头，如NewYorkTimes接入的来源） | |
| **content\_provider** | 合作方（合作方名称，如Reuters） | |
| **newstype** | Type: 1- 图文 2-图集 3-段子 4-GIF 5-视频 6-调查 7-纯文 8-视频+图文  9-GIF+图文  0-其他 | |
| **show\_style** | 新闻样式：0、无图 | |
|  | 1、一张大图 | |
|  | 2、两张图 | |
|  | 3、三张小图 | |
|  | 4、一张小图 | |
|  | 5、一张大图两张小图 | |
|  | 来源于详情页相关推荐的新闻，上报样式都为0（因为都是左文右图） | |
| **news\_staytime** | 页面停留时长：从新闻开始加载时开始计算，到用户关闭页面所用的时间。若中途用跳转到其它页面了，则暂停计时，待回到详情页时恢复计时。或中途划出的时间超过10分钟，则本次计时作废，不上报本次数据。如未加载成功退出，则报空。 | |
| **loading\_time** | 加载时长：计算页面开始加载到接口返回数据的时间 （开始加载报0，加载成功或加载失败才上报时间） | |
| **type1** | 加载失败码：把加载失败状态码报回来（报空为加载成功，没有失败） | |
| **category** | 分类ID（服务端定义的分类ID） | |
| **content** | 动作对象：1、返回（包括点击按钮返回与系统返回按钮） ，2、分享，3、更多按钮 ，4、点击图片（把缩略图放大/加载gif）（3034版本没有实现该功能） ，5、点击图片（进入看图模式）6、点击相关推荐文章 ,7、折叠按钮（视频详情页，上下折叠），8、查看源网址（点击更多后-点击源网址） | |

### 新闻列表页(loading)

事件名称：loading

|  |  |
| --- | --- |
| **标签** | **含义** |
| **action** | 动作：开始加载=1，加载成功=2，加载失败=3 |
| **loading\_time** | **加载时长：计算下拉开始到接口返回数据的时间，（开始加载报0，加载成功或加载失败才上报时间）** |
| **loading\_way** | 加载类型：1-读取缓存，2-从接口拉新数据 （加载成功才上报加载类型） |
| **extend1** | **扩展字段 Extend1** |
| **extend2** | 扩展字段 Extend2 |
| **type** | **加载类型：自动加载=1，用户下拽加载=2，底部加载=3（底部条触发点击底部提示条/点击返回顶部加载）** |
| **type1** | 加载失败码：把加载失败状态码报回来（报空为加载成功，没有失败） |

### 广告(ad)

事件名称：ad

|  |  |
| --- | --- |
| **标签** | **含义** |
| **entry** | 入口：新闻列表页=1  应用首页=2 新闻详情页=3 |
| **action** | 动作：请求广告=1 取缓存广告=2  广告位展示=3 广告展示=4 广告点击=5 |
| **content** | 状态：成功=1  失败=2 |
| **detail** | 失败码（没有则上报空） |
| **source** | 广告来源:admob=1 facebook=2  ADX（百度）=3 VK（俄罗斯）=4 |
| **behavior** | 用户行为： 主动获取广告=1   被动获取广告=2 |
| **newstype** | Type: 1- 图文 2-图集 3-段子 4-GIF 5-视频 6-调查 7-纯文 8-视频+图文  9-GIF+图文  0-其他 |
| **show\_style** | 内容样式：无图(纯文字)=6 一张大图=1  三站小图+文=4 一张小图=2 一张大图两张小图+文=3 图集+文 = 5  一张大图+文=11   GIF大图+文=12  视频(大图)+文 = 13 来源于详情页相关推荐的新闻，上报样式都为0（因为都是左文右图） |

### 应用启动(start)

事件标签: start action=1可以算成前台活跃

|  |  |
| --- | --- |
| **标签** | **含义** |
| **entry** | 入口： push=1，widget=2，icon=3，notification=4, lockscreen\_widget =5 |
| **open\_ad\_type** | 开屏广告类型: 开屏原生广告=1, 开屏插屏广告=2 |
| **action** | 状态：成功=1  失败=2 |
| **loading\_time** | 加载时长：计算下拉开始到接口返回数据的时间，（开始加载报0，加载成功或加载失败才上报时间） |
| **detail** | 失败码（没有则上报空） |
| **extend1** | 失败的message（没有则上报空） |

### 消息通知(notification)

事件标签：notification

|  |  |
| --- | --- |
| **标签** | **含义** |
| **action** | 动作：通知产生=1，通知弹出=2，通知点击=3，常驻通知展示（不重复上报，一天之内只报一次）=4 |
| **type** | 通知id：预警通知=1，天气预报（早=2，晚=3），常驻=4 |
| **ap\_time** | 客户端弹出时间 |
| **content** | 备用字段 |

### ~~用户前台活跃~~

事件标签: active\_foreground

|  |  |
| --- | --- |
| **标签** | **含义** |
| **push\_id** | 推送的消息的id，如果不是从推送消息打开，传空 |
| **access** | 1.push 2.icon 3.其他 |

### 用户后台活跃(active\_background)

事件标签: active\_background

|  |  |
| --- | --- |
| **标签** | **含义** |
| **active\_source** | 1=upgrade,2=download(下载),3=plugin\_upgrade |

## Mysql 数据

Mysql 则为对应的一些数据表

# 第3章 数据通道启动

## SpringBoot

Spring boot 这一层已经做了相关的数据过滤和验证，丢弃了没有用的数据。但是本身自己还会打印一些日志。

Springboot一般都是后台去配，我们是从日志里面拿数据。

## SL4J日志配置

Spring boot x

集成logback与SL4J一起使用，在磁盘打印日志，然后收集。

其中日志分为了3种，

* 控制台
* 错误日志输出（后台需要查看）
* 标准日志输出

<?xml version="1.0" encoding="UTF-8"?>  
<configuration debug="false">  
 <!--定义日志文件的存储地址 勿在 LogBack 的配置中使用相对路径 -->  
 <property name="LOG\_HOME" value="/tmp/logs/" />  
 <!--配置规则类的位置-->  
 <conversionRule conversionWord="ip" converterClass="com.tom.business.IPLogConfig" />  
 <!-- 控制台输出 -->  
 <appender name="STDOUT"  
 class="ch.qos.logback.core.ConsoleAppender">  
 <encoder  
 class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">  
 <!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符 -->  
 <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n</pattern>  
 </encoder>  
 </appender>  
   
 <!-- 按照每天生成日志文件。存储事件日志 -->  
 <appender name="FILE"  
 class="ch.qos.logback.core.rolling.RollingFileAppender">  
 <!-- <File>${LOG\_HOME}/app.log</File>设置日志不超过${log.max.size}时的保存路径，注意如果   
 是web项目会保存到Tomcat的bin目录 下 -->   
 <rollingPolicy  
 class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
 <!--日志文件输出的文件名 -->  
 <FileNamePattern>${LOG\_HOME}/app-%d{yyyy-MM-dd}.log</FileNamePattern>  
 <!--日志文件保留天数 -->  
 <MaxHistory>30</MaxHistory>  
 </rollingPolicy>  
 <encoder  
 class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">  
 <pattern>%ip,%msg%n</pattern>  
 </encoder>  
 <!--日志文件最大的大小 -->  
 <triggeringPolicy  
 class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">  
 <MaxFileSize>10MB</MaxFileSize>  
 </triggeringPolicy>  
 </appender>  
   
 <!--错误日志统一输出到这里-->  
 <appender name="error" class="ch.qos.logback.core.rolling.RollingFileAppender">  
 <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  
 <!--日志文件输出的文件名-->  
 <FileNamePattern>${LOG\_HOME}/error-%d{yyyy-MM-dd}.log</FileNamePattern>  
 <!--日志文件保留天数-->  
 <MaxHistory>30</MaxHistory>  
 </rollingPolicy>  
 <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">  
 <!--格式化输出：%d表示日期，%thread表示线程名，%-5level：级别从左显示5个字符宽度%msg：日志消息，%n是换行符-->  
 <pattern> %d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n</pattern>  
 </encoder>  
 <!--日志文件最大的大小-->  
 <triggeringPolicy class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">  
 <MaxFileSize>10MB</MaxFileSize>  
 </triggeringPolicy>  
 <!-- 所有error日志都在这里-->  
 <filter class="ch.qos.logback.classic.filter.LevelFilter">  
 <level>ERROR</level>  
 <onMatch>ACCEPT</onMatch>  
 <onMismatch>DENY</onMismatch>  
 </filter>  
 </appender>  
  
 <!-- 日志输出级别 -->  
 <root level="INFO">  
 <appender-ref ref="STDOUT" />  
 <appender-ref ref="FILE" />  
 <appender-ref ref="error" />  
 </root>  
</configuration>

## Kafka

### 创建 Kafka topic

Kafka的主题 \_和. 在处理的时候是一样的，所以用横线 –

所有日志的主题

bin/kafka-topics.sh --zookeeper localhost:2181 --create --replication-factor 2 --partitions 2 --topic log-test

实时信息的主题

bin/kafka-topics.sh --zookeeper localhost:2181 --create --replication-factor 2 --partitions 2 --topic display-test

### 删除 kafka topic

bin/kafka-topics.sh --delete --zookeeper localhost:2181 --topic log\_test

小结

分区数和冗余数的设定？

冗余数一般我们设置成2个。根据ISR，

分区数的话一般是3个。

## 收集日志的Flume

### 多层 Flume 部署

数据采集层 Flume 配置

代码清单 3-7 数据采集层Flume配置

a1.sources = r1

a1.channels = c1

a1.sinkgroups = g1

a1.sinks = k1 k2

a1.sources.r1.type = TAILDIR

a1.sources.r1.channels = c1

a1.sources.r1.positionFile=/mnt/flume/taildir\_position.json

a1.sources.r1.filegroups = f1

a1.sources.r1.filegroups.f1=/tmp/logs/app.+

a1.sources.r1.fileHeader = true

a1.channels.c1.type = file

a1.channels.c1.checkpointDir = /opt/modules/flume/checkpoint/behavior

a1.channels.c1.dataDirs = /opt/modules/flume/data/behavior/

a1.channels.c1.maxFileSize = 104857600

a1.channels.c1.capacity = 90000000

a1.channels.c1.keep-alive = 60

a1.sinkgroups.g1.sinks = k1 k2

a1.sinkgroups.g1.processor.type = load\_balance

a1.sinkgroups.g1.processor.backoff = true

a1.sinkgroups.g1.processor.selector = round\_robin

a1.sinkgroups.g1.processor.selector.maxTimeOut=10000

a1.sinks.k1.type = avro

a1.sinks.k1.channel = c1

a1.sinks.k1.batchSize = 1

a1.sinks.k1.hostname = dn130

a1.sinks.k1.port = 1234

a1.sinks.k2.type = avro

a1.sinks.k2.channel = c1

a1.sinks.k2.batchSize = 1

a1.sinks.k2.hostname = hadoop-senior03.itguigu.com

a1.sinks.k2.port = 1234

聚合层 Flume 配置

代码清单 3-8 聚合层Flume配置

a1.sources = r1

a1.channels = c1

a1.sinks = k1

a1.sources.r1.type = avro

a1.sources.r1.channels = c1

a1.sources.r1.bind = 0.0.0.0

a1.sources.r1.port = 1234

a1.channels.c1.type = file

a1.channels.c1.checkpointDir = /opt/modules/flume/checkpoint/behavior\_collect

a1.channels.c1.dataDirs = /opt/modules/flume/data/behavior\_collect

a1.channels.c1.maxFileSize = 104857600

a1.channels.c1.capacity = 90000000

a1.channels.c1.keep-alive = 60

a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink

a1.sinks.k1.topic = analysis-test

a1.sinks.k1.brokerList= localhost:9092

a1.sinks.k1.requiredAcks = 1

a1.sinks.k1.kafka.producer.type = sync

a1.sinks.k1.batchSize = 1

a1.sinks.k1.channel = c1

### 单层 Flume 部署

因为测试环境，所以采用单层flume部署

代码清单 3-9 单层Flume配置

a1.sources = r1 r2

a1.channels = c1 c2

a1.sinks = k1 k2

#读取全量日志的

a1.sources.r1.type = TAILDIR

a1.sources.r1.channels = c1

a1.sources.r1.positionFile=/mnt/flume/log\_position.json

a1.sources.r1.filegroups = f1

a1.sources.r1.filegroups.f1=/tmp/logs/app.+

a1.sources.r1.fileHeader = true

a1.channels.c1.type = file

a1.channels.c1.checkpointDir = /mnt/flume/checkpoint/behavior\_collect

a1.channels.c1.dataDirs = /mnt/flume/data/behavior\_collect

a1.channels.c1.maxFileSize = 104857600

a1.channels.c1.capacity = 90000000

a1.channels.c1.keep-alive = 60

a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink

a1.sinks.k1.kafka.topic = log-test

a1.sinks.k1.brokerList= localhost:9092

a1.sinks.k1.requiredAcks = 1

a1.sinks.k1.batchSize = 1

a1.sinks.k1.channel = c1

# 读取实时信息流的

a1.sources.r2.type = TAILDIR

a1.sources.r2.channels = c2

a1.sources.r2.positionFile=/mnt/flume/display\_position.json

a1.sources.r2.filegroups = f2

a1.sources.r2.filegroups.f2=/tmp/logs/app.+

a1.sources.r2.fileHeader = true

a1.sources.r2.interceptors = i1

a1.sources.r2.interceptors.i1.type = com.tom.flume.interceptor.DisplayInterceptor$Builder

a1.channels.c2.type = file

a1.channels.c2.checkpointDir = /mnt/flume/checkpoint/display\_collect

a1.channels.c2.dataDirs = /mnt/flume/data/display\_collect

a1.channels.c2.maxFileSize = 104857600

a1.channels.c2.capacity = 90000000

a1.channels.c2.keep-alive = 60

a1.sinks.k2.type = org.apache.flume.sink.kafka.KafkaSink

a1.sinks.k2.kafka.topic = display-test

a1.sinks.k2.brokerList= localhost:9092

a1.sinks.k2.requiredAcks = 1

a1.sinks.k2.kafka.producer.type = sync

a1.sinks.k2.batchSize = 1

a1.sinks.k2.channel = c2

Flume 一般都是部署在服务器上，由运维统一配置部署。此处直接读log日志的数据，log日志的格式是app-yyyy-mm-dd.log 不会发生日志名字的改变，所以可以直接读取，不需要再修改flume的taildir的innode的代码

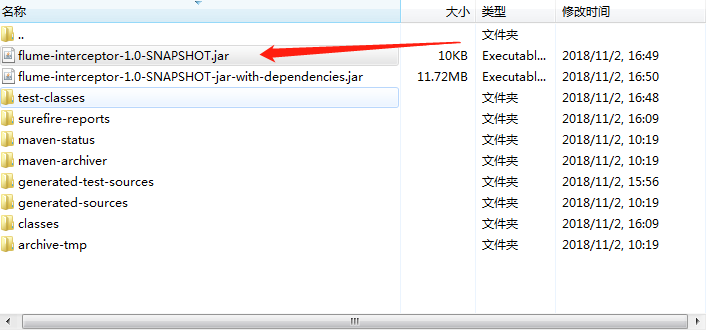
### 拦截器代码

拦截器用来过滤日志，把实时部分的日志抽离出来，然后单独写入一个topic里面。

拦截器的项目用maven引入对应flume的依赖，只需要core就可以了

package com.tom.flume.interceptor;  
  
import org.apache.flume.Context;  
import org.apache.flume.Event;  
import org.apache.flume.interceptor.Interceptor;  
  
import java.nio.charset.Charset;  
import java.util.ArrayList;  
import java.util.List;  
  
public class DisplayInterceptor implements Interceptor {  
 public void initialize() {  
 }  
  
  
 public Event intercept(Event event) {  
 String body = new String(event.getBody(), Charset.*forName*("UTF-8"));  
 // 判断是否为display的数据类型  
 if (body.contains("\"display\"")) {  
 return event;  
 }  
 return null;  
 }  
  
 public List<Event> intercept(List<Event> events) {  
 List<Event> intercepted = new ArrayList<>(events.size());  
 for (Event event : events) {  
 Event interceptedEvent = intercept(event);  
 if (interceptedEvent != null) {  
 intercepted.add(interceptedEvent);  
 }  
 }  
 return intercepted;  
 }  
  
  
 public void close() {  
 }  
  
 public static class Builder implements Interceptor.Builder {  
 public Interceptor build() {  
 return new DisplayInterceptor();  
 }  
  
 public void configure(Context context) {  
  
 }  
 }  
  
}

拦截器打包之后，只需要单独包，不需要将依赖的包上传



### Flume启动

生产启动

bin/flume-ng agent --conf ./conf -f ./conf/log-kafka.properties --name a1 -Dflume.root.logger=INFO,LOGFILE >/dev/null 2>&1 &

测试脚本

bin/flume-ng agent --conf ./conf -f ./conf/log-kafka.properties --name a1 -Dflume.root.logger=INFO,console

### 小结

1. Flume是如何实现事务的
2. Flume 中是否会出现数据丢失的情况。

一般不会，flume的source，channel，sink都是有事务的性质存在。Source 失败则重复读取。Channel和sink都是成功再删除。

但在升级的时候，或是在source源不是taildir的时候，

1. 两层架构和单层架构的优缺点是什么。

两层架构，方便kafka升级，减少kafka层的连接数目。增加批次写入的能力

多个flume的收集端会集中在flume的两台机器上面，同时增加flume流通的流程。增加时间。

单层flume的话时间较快。

连接集中在kafka层，在服务器很多的时候kafka层连接会非常多。

1. Flume如何升级

Flume 升级的话，分两种flume：

一种是写入kafka的flume。

1. 如果升级的同是我们需要先停掉flume，并且将channel的路径指向同一个，共用同一个taildir的位置json。这种情况会造成日志的损失，因为无法知道已经写到哪条日志了。同时要考虑到兼容的问题
2. 一种方法是启动一个新的flume，同时读取同一个日志文件。并且也写入同一个topic，这个时候启动新的flume，跑一会儿可以的话，再停掉老的flume。这样会造成数据的冗余。生产这么做，冗余是可以接受的。
3. 还有一种方法，我们在日志上加上服务器的ip。这样的话，我们先停掉flume，可以在kafka端消费查看，最后这个服务器是到哪一行日志了。然后我们修改taildir\_position.json 的位置。重新起一个flume，从这个位置开始读取。这种过程比较复杂
4. Flume如何监控

我们通过自己开发的系统agent来监控进程。进程出现异常，则会短信报错。

负载过高也会报警（磁盘，cpu，内存等）

1. Flume更改配置之后不需要重新启动，它会自己启动。
2. 日志的验证越早做，越好，就在最开始的时候做。能减少数据量，提高性能。
3. 压缩率。目前采用lzop的压缩方式，压缩率大概在3-5倍，170万的数据大概1个G

# 第4章 数据仓库

## Flume读取Kafka数据写hdfs

### 配置

项目中此处采用单层flum,但多个flume,同时往数据里面写

agent.sources= r1

agent.sinks= k1

agent.channels= c1

agent.sources.r1.type= org.apache.flume.source.kafka.KafkaSource

agent.sources.r1.kafka.bootstrap.servers=localhost:9092

agent.sources.r1.kafka.topics=log-test

agent.sources.r1.kafka.consumer.group.id = flume-test

agent.sources.r1.flumeBatchSize=1000

agent.sources.r1.useFlumeEventFormat=false

agent.sources.r1.restart=true

agent.sources.r1.batchSize=1000

agent.sources.r1.batchTimeout=3000

agent.sources.r1.channels=c1

agent.channels.c1.type = file

agent.channels.c1.checkpointDir = /mnt/flume/checkpoint/kafka\_collect1

agent.channels.c1.dataDirs = /mnt/flume/data/kafka\_collect1

agent.channels.c1.maxFileSize = 104857600

agent.channels.c1.capacity = 90000000

agent.channels.c1.keep-alive = 60

agent.sinks.k1.channel=c1

agent.sinks.k1.type=hdfs

agent.sinks.k1.hdfs.path=hdfs:///tmp/flume/%Y%m%d

agent.sinks.k1.hdfs.writeFormat=Text

agent.sinks.k1.hdfs.rollSize=0

agent.sinks.k1.hdfs.rollCount=0

#这个单位是秒，时间过少会造成文件数量非常多

agent.sinks.k1.hdfs.rollInterval=300

agent.sinks.k1.hdfs.threadsPoolSize=30

agent.sinks.k1.hdfs.fileType = CompressedStream

agent.sinks.k1.hdfs.fileSuffix=.lzo

# 这个前缀必须要加，因为有多个flume会写入文件

agent.sinks.k1.hdfs.filePrefix=log1.

agent.sinks.k1.hdfs.codeC = lzop

### 启动脚本

bin/flume-ng agent --conf ./conf -f ./conf/kafka-hdfs.properties --name agent -Dflume.root.logger=INFO,LOGFILE >/dev/null 2>&1 &

测试脚本

bin/flume-ng agent --conf ./conf -f ./conf/kafka-hdfs.properties --name agent -Dflume.root.logger=INFO,console

### 小结

Kafka的topic \_和 . 存在冲突，所以我们用 – 符号来区别含义

离线数据处理系统中的 Kafka 高级消费者程序将消息从 Kafka 集群中消费出来，然后写入指定的 HDFS 文件中。采用lzo压缩文件。此时为5分钟压缩一次文件

代码清单 4-1 Flume 读取kafka的数据

agent.sources=r1

agent.sinks=k1

agent.channels=c1

agent.sources.r1.type= org.apache.flume.source.kafka.KafkaSource

agent.sources.r1.kafka.bootstrap.servers=localhost:9092

agent.sources.r1.kafka.topics=test

#agent.sources.r1.serializer.class=kafka.serializer.StringEncoder

agent.sources.r1.kafka.consumer.group.id = flumetest

agent.sources.r1.flumeBatchSize=1000

agent.sources.r1.useFlumeEventFormat=false

agent.sources.r1.restart=true

agent.sources.r1.batchSize=1000

agent.sources.r1.batchTimeout=3000

agent.sources.r1.channels=c1

agent.channels.c1.type=memory

agent.channels.c1.capacity=102400

agent.channels.c1.transactionCapacity=1000

agent.channels.c1.byteCapacity=134217728

agent.channels.c1.byteCapacityBufferPercentage=80

agent.sinks.k1.channel=c1

agent.sinks.k1.type=hdfs

agent.sinks.k1.hdfs.path=hdfs:///tmp/flume/%Y%m%d

agent.sinks.k1.hdfs.writeFormat=Text

agent.sinks.k1.hdfs.rollSize=0

agent.sinks.k1.hdfs.rollCount=0

agent.sinks.k1.hdfs.rollInterval=120

agent.sinks.k1.hdfs.threadsPoolSize=30

agent.sinks.k1.hdfs.fileType = CompressedStream

agent.sinks.k1.hdfs.fileSuffix=.lzo

agent.sinks.k1.hdfs.codeC = lzop

## 测试脚本启动

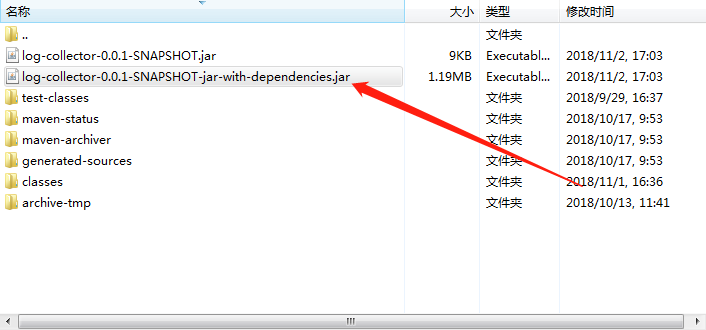
### 测试脚本

package com.tom.business;  
  
import java.util.Random;  
  
import com.alibaba.fastjson.JSONArray;  
import com.alibaba.fastjson.JSONObject;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
*/\*\*  
 \* Hello world!  
 \*/*public class App {  
  
 private final static Logger *logger* = LoggerFactory.*getLogger*(App.class);  
 static Random *rand* = new Random();  
  
 public static void main(String[] args) {  
// 循环遍历次数  
 int loop\_len = args.length > 1 ? Integer.*parseInt*(args[1]) : 10 \* 10000;  
 long millis = System.*currentTimeMillis*();  
// uid的长度  
 int uid\_length = args.length > 2 ? Integer.*parseInt*(args[2]) : 4;  
// 新闻id的长度  
 int newsid\_length = args.length > 3 ? Integer.*parseInt*(args[3]) : 4;  
 for(int i=0;i<loop\_len;i++) {  
 JSONObject json = new JSONObject();  
 json.put("ap", "weather");  
 json.put("cm", *generateComFields*(uid\_length));  
 JSONArray eventsArray = new JSONArray();  
// 新闻点击，展示  
 if (*rand*.nextBoolean()) {  
 eventsArray.add(*generateDisplay*(newsid\_length));  
 }  
// 新闻详情页  
 if (*rand*.nextBoolean()) {  
 eventsArray.add(*generateNewsDetail*(newsid\_length));  
 }  
// 新闻列表页  
 if (*rand*.nextBoolean()) {  
 eventsArray.add(*generateNewList*());  
 }  
// 广告  
 if (*rand*.nextBoolean()) {  
 eventsArray.add(*generateAd*());  
 }  
// 应用启动  
 if (*rand*.nextBoolean()) {  
 eventsArray.add(*generateStart*());  
 }  
// 消息通知  
 if (*rand*.nextBoolean()) {  
 eventsArray.add(*generateNotification*());  
 }  
// 用户后台活跃  
 if (*rand*.nextBoolean()) {  
 eventsArray.add(*generateBackground*());  
 }  
 json.put("et", eventsArray);  
// 时间  
 long log\_time = millis - *rand*.nextInt(1000 \* 60 \* 60);  
 *logger*.info(log\_time + "|" + json.toJSONString());  
 }  
 }  
  
 */\*\*  
 \* 公共字段设置  
 \*  
 \** ***@param*** *uid\_length  
 \** ***@return*** *\*/* static JSONObject generateComFields(int uid\_length) {  
// 公共字段  
 JSONObject common = new JSONObject();  
// 用户id  
 common.put("uid", 'u' + *getRandomDigits*(uid\_length));  
// 程序版本号 5,6等  
 common.put("vc", "" + (*rand*.nextInt(20)));  
// 程序版本名 v1.1.1  
 common.put("vn", "1." + *rand*.nextInt(4) + "." + *rand*.nextInt(10));  
// 安卓系统版本  
 common.put("os","8."+*rand*.nextInt(3)+"."+*rand*.nextInt(10));  
// 语言 es,en,pt  
 int flag = *rand*.nextInt(3);  
 switch (flag) {  
 case (0):  
 common.put("l", "es");  
 break;  
 case (1):  
 common.put("l", "en");  
 break;  
 case (2):  
 common.put("l", "pt");  
 break;  
 }  
// 渠道号 从哪个渠道来的  
 common.put("sr", *getRandomChar*(1));  
// 区域  
 flag = *rand*.nextInt(2);  
 switch (flag) {  
 case 0:  
 common.put("ar", "BR");  
 case 1:  
 common.put("ar", "MX");  
 }  
// 手机品牌 ba ,手机型号 md，就取2位数字了  
 flag = *rand*.nextInt(3);  
 switch (flag) {  
 case 0:  
 common.put("ba", "Sumsung");  
 common.put("md", "sumsung-" + *rand*.nextInt(20));  
 break;  
 case 1:  
 common.put("ba", "Huawei");  
 common.put("md", "huawei-" + *rand*.nextInt(20));  
 break;  
 case 2:  
 common.put("ba", "HTC");  
 common.put("md", "htc-" + *rand*.nextInt(20));  
 break;  
 }  
// 嵌入sdk 的版本  
 common.put("sv", "V2." + *rand*.nextInt(10) + "." + *rand*.nextInt(10));  
// gmail  
 common.put("g", *getRandomCharAndNumr*(8) + "@gmail.com");  
// 屏幕宽高 hw  
 flag = *rand*.nextInt(4);  
 switch (flag) {  
 case 0:  
 common.put("hw", "640\*960");  
 break;  
 case 1:  
 common.put("hw", "640\*1136");  
 break;  
 case 2:  
 common.put("hw", "750\*1134");  
 break;  
 case 3:  
 common.put("hw", "1080\*1920");  
 break;  
 }  
// 客户端产生日志时间  
 long millis = System.*currentTimeMillis*();  
 common.put("t", "" + (millis - *rand*.nextInt(99999999)));  
// 手机网络模式 3G,4G,WIFI  
 flag = *rand*.nextInt(3);  
 switch (flag) {  
 case 0:  
 common.put("nw", "3G");  
 break;  
 case 1:  
 common.put("nw", "4G");  
 break;  
 case 2:  
 common.put("nw", "WIFI");  
 break;  
 }  
// 拉丁美洲 西经34°46′至西经117°09；北纬32°42′至南纬53°54′  
// 经度  
 common.put("ln", (-34 - *rand*.nextInt(83)-*rand*.nextInt(60)/10.0) + "");  
// 纬度  
 common.put("la", (32 - *rand*.nextInt(85)-*rand*.nextInt(60)/10.0) + "");  
 return common;  
 }  
  
 */\*\*  
 \* 新闻展示事件  
 \*  
 \** ***@return*** *\*/* static JSONObject generateDisplay(int newsid\_length) {  
 JSONObject jsonObject = new JSONObject();  
 boolean boolFlag = *rand*.nextInt(10) < 7 ? true : false;  
// 动作：曝光新闻=1，点击新闻=2，  
 if (boolFlag) {  
 jsonObject.put("action", "1");  
 } else {  
 jsonObject.put("action", "2");  
 }  
// 新闻id  
 String newsId = 'n' + *getRandomDigits*(newsid\_length);  
 jsonObject.put("newsid", newsId);  
// 顺序 设置成6条吧  
 int flag = *rand*.nextInt(6);  
 jsonObject.put("place", "" + flag);  
// 新闻来源类型  
 flag = 1 + *rand*.nextInt(3);  
 jsonObject.put("showtype", flag + "");  
// 版权方 NewYorkTimes,AOL,Yahoo,CNN,ESPN,Reuters  
 flag = *rand*.nextInt(6);  
 switch (flag) {  
 case 0:  
 jsonObject.put("copyright", "NewYorkTimes");  
 break;  
 case 1:  
 jsonObject.put("copyright", "AOL");  
 break;  
 case 2:  
 jsonObject.put("copyright", "Yahoo");  
 break;  
 case 3:  
 jsonObject.put("copyright", "CNN");  
 break;  
 case 4:  
 jsonObject.put("copyright", "ESPN");  
 break;  
 case 5:  
 jsonObject.put("copyright", "Reuters");  
 break;  
 }  
// 合作方  
 flag = *rand*.nextInt(6);  
 switch (flag) {  
 case 0:  
 jsonObject.put("content\_provider", "NewYorkTimes");  
 break;  
 case 1:  
 jsonObject.put("content\_provider", "AOL");  
 break;  
 case 2:  
 jsonObject.put("content\_provider", "Yahoo");  
 break;  
 case 3:  
 jsonObject.put("content\_provider", "CNN");  
 break;  
 case 4:  
 jsonObject.put("content\_provider", "ESPN");  
 break;  
 case 5:  
 jsonObject.put("content\_provider", "Reuters");  
 break;  
 }  
// 新闻内容类型  
 flag = *rand*.nextInt(10);  
 jsonObject.put("newstype", "" + flag);  
// 曝光类型  
 flag = 1 + *rand*.nextInt(2);  
 jsonObject.put("extend1", "" + flag);  
// 新闻样式  
 flag = *rand*.nextInt(6);  
 jsonObject.put("extend2", "" + flag);  
// 分类  
 flag = 1 + *rand*.nextInt(100);  
 jsonObject.put("category", "" + flag);  
 return *packEventJson*("display",jsonObject);  
 }  
  
 */\*\*  
 \* 新闻详情页  
 \*  
 \** ***@param*** *newsid\_length  
 \** ***@return*** *\*/* static JSONObject generateNewsDetail(int newsid\_length) {  
 JSONObject eventJson = new JSONObject();  
 int flag = 1 + *rand*.nextInt(3);  
// 页面入口来源  
 eventJson.put("entry", flag + "");  
// 动作  
 eventJson.put("action", "" + (*rand*.nextInt(4) + 1));  
// 新闻id  
 eventJson.put("newsid", 'n' + *getRandomDigits*(newsid\_length));  
 // 新闻来源类型  
 flag = 1 + *rand*.nextInt(3);  
 eventJson.put("showtype", flag + "");  
 // 版权方 NewYorkTimes,AOL,Yahoo,CNN,ESPN,Reuters  
 flag = *rand*.nextInt(6);  
 switch (flag) {  
 case 0:  
 eventJson.put("copyright", "NewYorkTimes");  
 break;  
 case 1:  
 eventJson.put("copyright", "AOL");  
 break;  
 case 2:  
 eventJson.put("copyright", "Yahoo");  
 break;  
 case 3:  
 eventJson.put("copyright", "CNN");  
 break;  
 case 4:  
 eventJson.put("copyright", "ESPN");  
 break;  
 case 5:  
 eventJson.put("copyright", "Reuters");  
 break;  
 }  
// 合作方  
 flag = *rand*.nextInt(6);  
 switch (flag) {  
 case 0:  
 eventJson.put("content\_provider", "NewYorkTimes");  
 break;  
 case 1:  
 eventJson.put("content\_provider", "AOL");  
 break;  
 case 2:  
 eventJson.put("content\_provider", "Yahoo");  
 break;  
 case 3:  
 eventJson.put("content\_provider", "CNN");  
 break;  
 case 4:  
 eventJson.put("content\_provider", "ESPN");  
 break;  
 case 5:  
 eventJson.put("content\_provider", "Reuters");  
 break;  
 }  
// 新闻内容类型  
 flag = *rand*.nextInt(10);  
 eventJson.put("newstype", "" + flag);  
 // 新闻样式  
 flag = *rand*.nextInt(6);  
 eventJson.put("show\_style", "" + flag);  
// 页面停留时长  
 flag = *rand*.nextInt(10) \* *rand*.nextInt(7);  
 eventJson.put("news\_staytime", flag + "");  
// 加载时长  
 flag = *rand*.nextInt(10) \* *rand*.nextInt(7);  
 eventJson.put("loading\_time", flag + "");  
// 加载失败码  
 flag = *rand*.nextInt(10);  
 switch (flag) {  
 case 1:  
 eventJson.put("type1", "102");  
 break;  
 case 2:  
 eventJson.put("type1", "201");  
 break;  
 case 3:  
 eventJson.put("type1", "325");  
 break;  
 case 4:  
 eventJson.put("type1", "433");  
 break;  
 case 5:  
 eventJson.put("type1", "542");  
 break;  
 default:  
 eventJson.put("type1", "");  
 break;  
 }  
 // 分类  
 flag = 1 + *rand*.nextInt(100);  
 eventJson.put("category", "" + flag);  
// 动作对象  
 flag = 1 + *rand*.nextInt(8);  
 eventJson.put("content", "" + flag);  
 return *packEventJson*("newsdetailpro", eventJson);  
 }  
  
 */\*\*  
 \* 新闻列表  
 \** ***@return*** *\*/* static JSONObject generateNewList(){  
 JSONObject jsonObject=new JSONObject();  
// 动作  
 int flag=*rand*.nextInt(3)+1;  
 jsonObject.put("action",flag+"");  
// 加载时长  
 flag = *rand*.nextInt(10) \* *rand*.nextInt(7);  
 jsonObject.put("loading\_time", flag + "");  
 // 失败码  
 flag = *rand*.nextInt(10);  
 switch (flag) {  
 case 1:  
 jsonObject.put("type1", "102");  
 break;  
 case 2:  
 jsonObject.put("type1", "201");  
 break;  
 case 3:  
 jsonObject.put("type1", "325");  
 break;  
 case 4:  
 jsonObject.put("type1", "433");  
 break;  
 case 5:  
 jsonObject.put("type1", "542");  
 break;  
 default:  
 jsonObject.put("type1", "");  
 break;  
 }  
 // 页面 加载类型  
 flag = 1 + *rand*.nextInt(2);  
 jsonObject.put("loading\_way", "" + flag);  
// 扩展字段1  
 jsonObject.put("extend1", "");  
// 扩展字段2  
 jsonObject.put("extend2", "");  
// 用户加载类型  
 flag = 1 + *rand*.nextInt(3);  
 jsonObject.put("type", "" + flag);  
 return *packEventJson*("loading",jsonObject);  
 }  
  
 */\*\*  
 \* 广告相关字段  
 \** ***@return*** *\*/* static JSONObject generateAd(){  
 JSONObject jsonObject=new JSONObject();  
// 入口  
 int flag=*rand*.nextInt(3)+1;  
 jsonObject.put("entry",flag+"");  
// 动作  
 flag=*rand*.nextInt(5)+1;  
 jsonObject.put("action",flag+"");  
// 状态  
 flag=*rand*.nextInt(10)>6?2:1;  
 jsonObject.put("content",flag+"");  
// 失败码  
 flag = *rand*.nextInt(10);  
 switch (flag) {  
 case 1:  
 jsonObject.put("detail", "102");  
 break;  
 case 2:  
 jsonObject.put("detail", "201");  
 break;  
 case 3:  
 jsonObject.put("detail", "325");  
 break;  
 case 4:  
 jsonObject.put("detail", "433");  
 break;  
 case 5:  
 jsonObject.put("detail", "542");  
 break;  
 default:  
 jsonObject.put("detail", "");  
 break;  
 }  
// 广告来源  
 flag=*rand*.nextInt(4)+1;  
 jsonObject.put("source",flag+"");  
// 用户行为  
 flag=*rand*.nextInt(2)+1;  
 jsonObject.put("behavior",flag+"");  
// 新闻类型  
 flag = *rand*.nextInt(10);  
 jsonObject.put("newstype", "" + flag);  
// 展示样式  
 flag = *rand*.nextInt(6);  
 jsonObject.put("show\_style", "" + flag);  
 return *packEventJson*("ad",jsonObject);  
 }  
  
 */\*\*  
 \* 启动日志  
 \** ***@return*** *\*/* static JSONObject generateStart(){  
 JSONObject jsonObject=new JSONObject();  
// 入口  
 int flag=*rand*.nextInt(5)+1;  
 jsonObject.put("entry",flag+"");  
// 开屏广告类型  
 flag=*rand*.nextInt(2)+1;  
 jsonObject.put("open\_ad\_type",flag+"");  
// 状态  
 flag=*rand*.nextInt(10)>8?2:1;  
 jsonObject.put("action",flag+"");  
// 加载时长  
 jsonObject.put("loading\_time",*rand*.nextInt(20)+"");  
// 失败码  
 flag = *rand*.nextInt(10);  
 switch (flag) {  
 case 1:  
 jsonObject.put("detail", "102");  
 break;  
 case 2:  
 jsonObject.put("detail", "201");  
 break;  
 case 3:  
 jsonObject.put("detail", "325");  
 break;  
 case 4:  
 jsonObject.put("detail", "433");  
 break;  
 case 5:  
 jsonObject.put("detail", "542");  
 break;  
 default:  
 jsonObject.put("detail", "");  
 break;  
 }  
 return *packEventJson*("start",jsonObject);  
 }  
  
 */\*\*  
 \* 消息通知  
 \** ***@return*** *\*/* static JSONObject generateNotification(){  
 JSONObject jsonObject=new JSONObject();  
 int flag=*rand*.nextInt(4)+1;  
// 动作  
 jsonObject.put("action",flag+"");  
// 通知id  
 flag=*rand*.nextInt(4)+1;  
 jsonObject.put("type",flag+"");  
// 客户端弹时间  
 jsonObject.put("ap\_time",(System.*currentTimeMillis*()-*rand*.nextInt(99999999))+"");  
// 备用字段  
 jsonObject.put("content","");  
 return *packEventJson*("notification",jsonObject);  
 }  
  
 */\*\*  
 \* 后台活跃  
 \** ***@return*** *\*/* static JSONObject generateBackground(){  
 JSONObject jsonObject=new JSONObject();  
// 启动源  
 int flag=*rand*.nextInt(3)+1;  
 jsonObject.put("active\_source",flag+"");  
 return *packEventJson*("active\_background",jsonObject);  
 }  
  
  
  
 static JSONObject packEventJson(String eventName, JSONObject jsonObject) {  
 JSONObject eventJson = new JSONObject();  
 eventJson.put("ett", (System.*currentTimeMillis*() - *rand*.nextInt(99999999)) + "");  
 eventJson.put("en", eventName);  
 eventJson.put("kv", jsonObject);  
 return eventJson;  
 }  
  
 */\*\*  
 \* 获取定长度的数字  
 \*  
 \** ***@param*** *leng  
 \** ***@return*** *\*/* static String getRandomDigits(int leng) {  
 String result = "";  
 for (int i = 0; i < leng; i++) {  
 result += *rand*.nextInt(10);  
 }  
 return result;  
 }  
  
 */\*\*  
 \* 获取随机字母组合  
 \*  
 \** ***@param*** *length 字符串长度  
 \** ***@return*** *\*/* public static String getRandomChar(Integer length) {  
 String str = "";  
 Random random = new Random();  
 for (int i = 0; i < length; i++) {  
 // 字符串  
 str += (char) (65 + random.nextInt(26));// 取得大写字母  
 }  
 return str;  
 }  
  
 */\*\*  
 \* 获取随机字母数字组合  
 \*  
 \** ***@param*** *length 字符串长度  
 \** ***@return*** *\*/* public static String getRandomCharAndNumr(Integer length) {  
 String str = "";  
 Random random = new Random();  
 for (int i = 0; i < length; i++) {  
 boolean b = random.nextBoolean();  
 if (b) { // 字符串  
 // int choice = random.nextBoolean() ? 65 : 97; 取得65大写字母还是97小写字母  
 str += (char) (65 + random.nextInt(26));// 取得大写字母  
 } else { // 数字  
 str += String.*valueOf*(random.nextInt(10));  
 }  
 }  
 return str;  
 }  
  
  
}

Maven的配置中有插件，将依赖包全部打成一个包

<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-assembly-plugin</artifactId>  
 <version>2.5.5</version>  
 <configuration>  
 <descriptorRefs>  
 <descriptorRef>jar-with-dependencies</descriptorRef>  
 </descriptorRefs>  
 </configuration>  
 <executions>  
 <execution>  
 <id>make-assembly</id>  
 <phase>package</phase>  
 <goals>  
 <goal>assembly</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 <!--jdk配置 -->  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-compiler-plugin</artifactId>  
 <version>3.6.1</version>  
 <configuration>  
 <source>1.8</source>  
 <target>1.8</target>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

测试脚本需要依赖多个包，所以把依赖包打包成一个包，然后放进来



### 启动

java -classpath log-collector-0.0.1-SNAPSHOT-jar-with-dependencies.jar com.tom.business.App >test.log

测试整个流程。

1. 先看kafka是否收到数据
2. 再看 hdfs端是否有存在数据

Kafka端测试脚本

bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic log-test

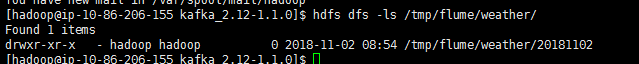
bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic display-test

成功出现数据，则接受成功

再看hdfs 端

Hdfs dfs –ls /tmp/flume/

此时出现了日志目录

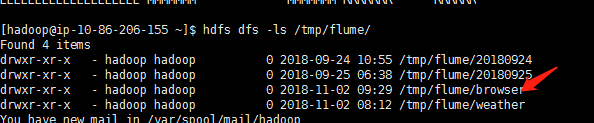


zk 看各个消费者，broker

### 测试多个应用

java -classpath log-collector-0.0.1-SNAPSHOT-jar-with-dependencies.jar com.tom.business.App browser >test.log

此时出现了browser 这个应用的日志目录



这样每个平台就出现了多个应用

## Spark SQL

Spark SQL 的命令基本同hive，spark执行脚本的命令

spark-sql -e "$sql" --driver-java-options "-Dlog4j.configuration=file:///mnt/log4j.properties"

## 事实表

### 创建schema

drop schema if exists dw\_weather;

create schema dw\_weather;

use dw\_weather;

### 原始日志表

drop table if exists stage\_originlog\_lzo\_dt;

CREATE EXTERNAL TABLE `stage\_originlog\_lzo\_dt`(`line` string)

PARTITIONED BY (`dt` string)

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'

WITH SERDEPROPERTIES (

'serialization.format' = '1'

)

STORED AS

INPUTFORMAT 'com.hadoop.mapred.DeprecatedLzoTextInputFormat'

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

LOCATION '/tmp/flume/weather';

添加分区

alter table stage\_originlog\_lzo\_dt add partition(dt='2018-11-03') location '/tmp/flume/weather/2018-11-03';

### 小结

这里面就是将数据写入进hdfs，通过hive将原始数据关联起来了。后面将根据原始日志数据来进行ETL,获取数据。

## 细节数据表

明细表用于存储事实表转换过来的明细数据。

drop table if exists ods\_basedata\_dt;

CREATE EXTERNAL TABLE `ods\_basedata\_dt`(`user\_id` string, `version\_code` string, `version\_name` string, `lang` string, `source` string, `os` string, `area` string, `model` string,`brand` string, `sdk\_version` string, `gmail` string, `height\_width` string, `app\_time` string, `network` string, `lng` string, `lat` string, `server\_time` string, `ip` string, `event\_name` string, `event\_json` string)

PARTITIONED BY (`dt` string)

location '/tmp/teach/ods\_basedata\_dt/';

其中event\_name 和 event\_json 用来对应事件名和业务字段

### Hive函数

UDF 用于解析公共字段

package com.tom.udf;  
  
import org.apache.commons.lang3.StringUtils;  
import org.apache.hadoop.hive.ql.exec.UDF;  
import org.json.JSONException;  
import org.json.JSONObject;  
  
public class BaseFieldUDF extends UDF {  
 */\*\*  
 \* 现在只支持字符串格式的内容  
 \*  
 \** ***@param*** *line  
 \** ***@param*** *key  
 \** ***@param*** *defaultVal  
 \** ***@return*** *\*/  
 /\*\*  
 \* jsonKeys是对应json里面的key，这个是要区分大小写的  
 \*  
 \** ***@param*** *line  
 \** ***@param*** *jsonKeysString  
 \** ***@return*** *\*/* public String evaluate(String line, String jsonKeysString) {  
 String[] jsonKeys = jsonKeysString.split(",");  
 String splitSign = "\t";  
 StringBuilder sb = new StringBuilder();  
 JSONObject json = null;  
 if (StringUtils.*isBlank*(line)) {  
 for (int i = 0; i < jsonKeys.length; i++) {  
 sb.append("" + splitSign);  
 }  
 } else {  
 try {  
// 提取公共字段  
 String[] logContents = line.split("\\|");  
 String serverTime = "";  
 String ip = "";  
 String jsonline = "";  
 // 判斷是否出现非法日志，一般都是3个  
 if (logContents.length == 3) {  
 serverTime = logContents[1];  
 ip = logContents[0];  
 jsonline = logContents[2];  
 } else {  
 return "";  
 }  
 if (StringUtils.*isNotBlank*(jsonline)) {  
 json = new JSONObject(jsonline);  
 }  
 if (json != null) {  
 JSONObject basep = json.getJSONObject("cm");  
 for (int i = 0; i < jsonKeys.length; i++) {  
 String fieldName = jsonKeys[i].trim();  
 sb.append(*getValue*(basep,fieldName,"")).append(splitSign);  
 }  
 }else {  
 sb.append(splitSign);  
 }  
// 添加业务字段  
 sb.append(json.getString("et")).append(splitSign);  
 sb.append(serverTime).append(splitSign)  
 .append(ip);  
 } catch (Exception e) {  
 System.*err*.println(line);  
 e.printStackTrace();  
 }  
 }  
 return sb.toString();  
 }  
  
 */\*\*  
 \* 获取json里面的值,自定义默认值，不存在这个key不会报错  
 \** ***@param*** *json  
 \** ***@param*** *key  
 \** ***@param*** *defValue  
 \** ***@return*** *\*/* public static String getValue(JSONObject json, String key, String defValue) {  
 if (json == null)  
 return defValue;  
 try {  
 if (json.has(key)) {  
 return json.getString(key);  
 }else {  
 return defValue;  
 }  
 } catch (JSONException e) {  
 }  
 return defValue;  
 }  
  
 public static void main(String[] args) {  
 String line = "10.86.206.155|1541217850324|{\"cm\":{\"ln\":\"-74.8\",\"sv\":\"V2.2.2\",\"os\":\"8.1.3\",\"g\":\"P7XC9126@gmail.com\",\"nw\":\"3G\",\"l\":\"es\",\"vc\":\"6\",\"hw\":\"640\*960\",\"uid\":\"u8739\",\"ar\":\"MX\",\"t\":\"1541204134250\",\"la\":\"-31.7\",\"md\":\"huawei-17\",\"vn\":\"1.1.2\",\"sr\":\"O\",\"ba\":\"Huawei\"},\"ap\":\"weather\",\"et\":[{\"ett\":\"1541146624055\",\"en\":\"display\",\"kv\":{\"newsid\":\"n4195\",\"copyright\":\"ESPN\",\"content\_provider\":\"CNN\",\"extend2\":\"5\",\"action\":\"2\",\"extend1\":\"2\",\"place\":\"3\",\"showtype\":\"2\",\"category\":\"72\",\"newstype\":\"5\"}},{\"ett\":\"1541213331817\",\"en\":\"loading\",\"kv\":{\"extend2\":\"\",\"loading\_time\":\"15\",\"action\":\"3\",\"extend1\":\"\",\"type1\":\"\",\"type\":\"3\",\"loading\_way\":\"1\"}},{\"ett\":\"1541126195645\",\"en\":\"ad\",\"kv\":{\"entry\":\"3\",\"show\_style\":\"0\",\"action\":\"2\",\"detail\":\"325\",\"source\":\"4\",\"behavior\":\"2\",\"content\":\"1\",\"newstype\":\"5\"}},{\"ett\":\"1541202678812\",\"en\":\"notification\",\"kv\":{\"ap\_time\":\"1541184614380\",\"action\":\"3\",\"type\":\"4\",\"content\":\"\"}},{\"ett\":\"1541194686688\",\"en\":\"active\_background\",\"kv\":{\"active\_source\":\"3\"}}]}";  
 String x = new BaseFieldUDF().evaluate(line, "uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t");  
 System.*out*.println(x);  
 }  
  
}

UDTF 用于展开业务字段

package com.tom.udtf;  
import org.apache.commons.lang3.StringUtils;  
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;  
import org.apache.hadoop.hive.ql.metadata.HiveException;  
import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;  
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;  
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;  
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;  
import org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFactory;  
import org.json.JSONArray;  
import org.json.JSONException;  
  
import java.util.ArrayList;  
  
*/\*\*  
 \* 用于将json串中的json数组展平，将eventname字段提取出来，并且保存成event\_name 和event\_json  
 \*  
 \** ***@author*** *tomluo  
 \*  
 \*/*public class EventJsonUDTF extends GenericUDTF {  
  
 @Override  
 public StructObjectInspector initialize(StructObjectInspector argOIs) throws UDFArgumentException {  
 ArrayList<String> fieldNames = new ArrayList<>();  
 ArrayList<ObjectInspector> fieldOIs = new ArrayList<>();  
 fieldNames.add("event\_name");  
 fieldOIs.add(PrimitiveObjectInspectorFactory.*javaStringObjectInspector*);  
 fieldNames.add("event\_json");  
 fieldOIs.add(PrimitiveObjectInspectorFactory.*javaStringObjectInspector*);  
 return ObjectInspectorFactory.*getStandardStructObjectInspector*(fieldNames, fieldOIs);  
 }  
  
 @Override  
 public void process(Object[] objects) throws HiveException {  
 String input = objects[0].toString();  
 if ("".equals(input) || input == null) {  
 return;  
 }  
 JSONArray ja = null;  
 if (StringUtils.*isNotBlank*(input)) {  
 try {  
 ja = new JSONArray(input);  
 } catch (JSONException e) {  
 e.printStackTrace();  
 }  
 if (ja == null)  
 return;  
 for (int i = 0; i < ja.length(); i++) {  
 String[] result = new String[2];  
 try {  
 result[0] = ja.getJSONObject(i).getString("en");  
 result[1] = ja.getString(i);  
 } catch (JSONException e) {  
 continue;  
 }  
 forward(result);  
 }  
 }  
 }  
  
 @Override  
 public void close() throws HiveException {  
  
 }  
  
 }

### ETL脚本

此处可以存为shell脚本执行。其中event\_name 和event\_json用于处理数据各个埋点事件

use dw\_weather;

alter table stage\_originlog\_lzo\_dt add partition(dt='2018-11-03') location '/tmp/flume/logs/weather/2018-11-03';

add jar /home/hadoop/hive-function-1.0-SNAPSHOT.jar;

create temporary function base\_analizer as 'com.tom.udf.BaseFieldUDF';

create temporary function flat\_analizer as 'com.tom.udtf.EventJsonUDTF';

set hive.exec.dynamic.partition.mode=nonstrict;

insert overwrite table ods\_basedata\_dt

PARTITION (dt)

select

user\_id,

version\_code,

version\_name,

lang,

source ,

os ,

area ,

model ,

brand ,

sdk\_version ,

gmail ,

height\_width ,

network ,

lng ,

lat ,

app\_time ,

event\_name ,

event\_json ,

server\_time ,

ip ,

dt

from

(

select

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[0] as user\_id,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[1] as version\_code,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[2] as version\_name,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[3] as lang,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[4] as source,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[5] as os,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[6] as area,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[7] as model,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[8] as brand,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[9] as sdk\_version,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[10] as gmail,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[11] as height\_width,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[12] as network,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[13] as lng,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[14] as lat,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[15] as app\_time,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[16] as ops,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[17] as server\_time,

split(base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t'),'\t')[18] as ip,

dt

from stage\_originlog\_lzo\_dt where dt='2018-11-03' and base\_analizer(line,'uid,vc,vn,l,sr,os,ar,md,ba,sv,g,hw,nw,ln,la,t')<>''

) sdk\_log lateral view flat\_analizer(ops) tmp\_k as event\_name, event\_json;

### 小结

基础明细表存储基本是完整的数据

## 新闻点击展示表

## 详情页表

### 建表语句

drop table if exists ods\_newsdetailpro\_dt;

CREATE EXTERNAL TABLE `ods\_newsdetailpro\_dt`(`user\_id` string, `version\_code` string, `version\_name` string, `lang` string, `source` string, `os` string, `area` string, `model` string,`brand` string, `sdk\_version` string, `gmail` string, `height\_width` string, `network` string, `lng` string, `lat` string, `app\_time` string,entry string,action string,newsid string,showtype string,copyright string,content\_provider string,newstype string,show\_style string,news\_staytime string,loading\_time string,type1 string,category string,content string,`server\_time` string, `ip` string)

PARTITIONED BY (dt string)

location '/tmp/teach/ods\_newsdetailpro\_dt/';

### ETL脚本

insert overwrite table ods\_newsdetailpro\_dt

PARTITION (dt)

select

user\_id,

version\_code,

version\_name,

lang,

source,

os,

area,

model,

brand,

sdk\_version,

gmail,

height\_width,

network,

lng,

lat,

app\_time,

get\_json\_object('event\_json','$.kv.entry') entry,

get\_json\_object('event\_json','$.kv.action') action,

get\_json\_object('event\_json','$.kv.newsid') newsid,

get\_json\_object('event\_json','$.kv.showtype') showtype,

get\_json\_object('event\_json','$.kv.copyright') copyright,

get\_json\_object('event\_json','$.kv.content\_provider') content\_provider,

get\_json\_object('event\_json','$.kv.newstype') newstype,

get\_json\_object('event\_json','$.kv.show\_style') show\_style,

get\_json\_object('event\_json','$.kv.news\_staytime') news\_staytime,

get\_json\_object('event\_json','$.kv.loading\_time') loading\_time,

get\_json\_object('event\_json','$.kv.type1') type1,

get\_json\_object('event\_json','$.kv.category') category,

get\_json\_object('event\_json','$.kv.content') content,

server\_time,

ip,

dt

from ods\_basedata\_dt where dt='2018-11-03' and event\_name='newsdetailpro';

## 列表页

### 建表语句

drop table if exists ods\_loading\_dt;

CREATE EXTERNAL TABLE `ods\_loading\_dt`(`user\_id` string, `version\_code` string, `version\_name` string, `lang` string, `source` string, `os` string, `area` string, `model` string,`brand` string, `sdk\_version` string, `gmail` string, `height\_width` string, `network` string, `lng` string, `lat` string, `app\_time` string,action string,loading\_time string,loading\_way string,extend1 string,extend2 string,load\_type string,type1 string,`server\_time` string,`ip` string)

PARTITIONED BY (dt string)

location '/tmp/teach/ods\_loading\_dt/';

### ETL脚本

insert overwrite table ods\_loading\_dt

PARTITION (dt)

select

user\_id,

version\_code,

version\_name,

lang,

source,

os,

area,

model,

brand,

sdk\_version,

gmail,

height\_width,

network,

lng,

lat,

app\_time,

get\_json\_object('event\_json','$.kv.action') action,

get\_json\_object('event\_json','$.kv.loading\_time') loading\_time,

get\_json\_object('event\_json','$.kv.loading\_way') loading\_way,

get\_json\_object('event\_json','$.kv.extend1') extend1,

get\_json\_object('event\_json','$.kv.extend2') extend2,

get\_json\_object('event\_json','$.kv.type') load\_type,

get\_json\_object('event\_json','$.kv.type1') type1,

server\_time,

ip,

dt

from ods\_basedata\_dt where dt='2018-11-03' and event\_name='loading';

## 广告

### 建表语句

drop table if exists ods\_ad\_dt;

CREATE EXTERNAL TABLE `ods\_ad\_dt`(`user\_id` string, `version\_code` string, `version\_name` string, `lang` string, `source` string, `os` string, `area` string, `model` string,`brand` string, `sdk\_version` string, `gmail` string, `height\_width` string, `network` string, `lng` string, `lat` string, `app\_time` string,entry string,action string,content string,detail string,ad\_source string,behavior string,newstype string,show\_style string,`server\_time` string,`ip` string)

PARTITIONED BY (dt string)

location '/tmp/teach/ods\_ad\_dt/';

### ETL脚本

insert overwrite table ods\_ad\_dt

PARTITION (dt)

select

user\_id,

version\_code,

version\_name,

lang,

source,

os,

area,

model,

brand,

sdk\_version,

gmail,

height\_width,

network,

lng,

lat,

app\_time,

get\_json\_object('event\_json','$.kv.entry') entry,

get\_json\_object('event\_json','$.kv.action') action,

get\_json\_object('event\_json','$.kv.content') content,

get\_json\_object('event\_json','$.kv.detail') detail,

get\_json\_object('event\_json','$.kv.source') ad\_source,

get\_json\_object('event\_json','$.kv.behavior') behavior,

get\_json\_object('event\_json','$.kv.newstype') newstype,

get\_json\_object('event\_json','$.kv.show\_style') show\_style,

server\_time,

ip,

dt

from ods\_basedata\_dt where dt='2018-11-03' and event\_name='ad';

## 应用启动

### 建表语句

drop table if exists ods\_start\_dt;

CREATE EXTERNAL TABLE `ods\_start\_dt`(`user\_id` string, `version\_code` string, `version\_name` string, `lang` string, `source` string, `os` string, `area` string, `model` string,`brand` string, `sdk\_version` string, `gmail` string, `height\_width` string, `network` string, `lng` string, `lat` string, `app\_time` string,entry string,open\_ad\_type string,action string,loading\_time string,detail string,extend1 string,`server\_time` string,`ip` string)

PARTITIONED BY (dt string)

location '/tmp/teach/ods\_start\_dt/';

### ETL脚本

insert overwrite table ods\_start\_dt

PARTITION (dt)

select

user\_id,

version\_code,

version\_name,

lang,

source,

os,

area,

model,

brand,

sdk\_version,

gmail,

height\_width,

network,

lng,

lat,

app\_time,

get\_json\_object('event\_json','$.kv.entry') entry,

get\_json\_object('event\_json','$.kv.open\_ad\_type') open\_ad\_type,

get\_json\_object('event\_json','$.kv.action') action,

get\_json\_object('event\_json','$.kv.loading\_time') loading\_time,

get\_json\_object('event\_json','$.kv.detail') detail,

get\_json\_object('event\_json','$.kv.extend1') extend1,

server\_time,

ip,

dt

from ods\_basedata\_dt where dt='2018-11-03' and event\_name='start';

## 消息通知

### 建表语句

drop table if exists ods\_notification\_dt;

CREATE EXTERNAL TABLE `ods\_notification\_dt`(`user\_id` string, `version\_code` string, `version\_name` string, `lang` string, `source` string, `os` string, `area` string, `model` string,`brand` string, `sdk\_version` string, `gmail` string, `height\_width` string, `network` string, `lng` string, `lat` string, `app\_time` string,action string,noti\_type string,ap\_time string,content string,`server\_time` string,`ip` string)

PARTITIONED BY (dt string)

location '/tmp/teach/ods\_notification\_dt/';

### ETL脚本

insert overwrite table ods\_notification\_dt

PARTITION (dt)

select

user\_id,

version\_code,

version\_name,

lang,

source,

os,

area,

model,

brand,

sdk\_version,

gmail,

height\_width,

network,

lng,

lat,

app\_time,

get\_json\_object('event\_json','$.kv.action') action,

get\_json\_object('event\_json','$.kv.noti\_type') noti\_type,

get\_json\_object('event\_json','$.kv.ap\_time') ap\_time,

get\_json\_object('event\_json','$.kv.content') content,

server\_time,

ip,

dt

from ods\_basedata\_dt where dt='2018-11-03' and event\_name='notification';

## 后台活跃

### 建表语句

drop table if exists ods\_background\_dt;

CREATE EXTERNAL TABLE `ods\_background\_dt`(`user\_id` string, `version\_code` string, `version\_name` string, `lang` string, `source` string, `os` string, `area` string, `model` string,`brand` string, `sdk\_version` string, `gmail` string, `height\_width` string, `network` string, `lng` string, `lat` string, `app\_time` string,active\_source string,`server\_time` string,`ip` string)

PARTITIONED BY (dt string)

location '/tmp/teach/ods\_background\_dt/';

### ETL脚本

insert overwrite table ods\_background\_dt

PARTITION (dt)

select

user\_id,

version\_code,

version\_name,

lang,

source,

os,

area,

model,

brand,

sdk\_version,

gmail,

height\_width,

network,

lng,

lat,

app\_time,

get\_json\_object('event\_json','$.kv.active\_source') active\_source,

server\_time,

ip,

dt

from ods\_basedata\_dt where dt='2018-11-03' and event\_name='active\_background';

# 第5章 分析业务指标

## 历史用户表

### 说明

历史用户表用来存储所有用过我们系统的用户。这个表可以用来分析许多的指标。

目前如果采用fulljoin的方式来进行前后用户的关联，数据倾斜较大，所以我们用了union和udaf函数来做这个清洗。

### 建表语句

drop table if exists mid\_user\_history\_dt;

CREATE EXTERNAL TABLE `mid\_user\_history\_dt`(`user\_id` string, `area` string,version\_name string,lang string,source string,first\_version string,`first\_dat` string, first\_source string,`last\_dat` string)

PARTITIONED BY (`dt` string)

location '/tmp/teach/mid\_user\_history\_dt/';

### UDAF函数

package com.tom.udaf;  
  
import org.apache.hadoop.hive.ql.exec.UDFArgumentTypeException;  
import org.apache.hadoop.hive.ql.metadata.HiveException;  
import org.apache.hadoop.hive.ql.parse.SemanticException;  
import org.apache.hadoop.hive.ql.udf.generic.AbstractGenericUDAFResolver;  
import org.apache.hadoop.hive.ql.udf.generic.GenericUDAFEvaluator;  
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;  
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;  
import org.apache.hadoop.hive.serde2.objectinspector.PrimitiveObjectInspector;  
import org.apache.hadoop.hive.serde2.typeinfo.TypeInfo;  
import org.apache.hadoop.hive.serde2.typeinfo.TypeInfoUtils;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
  
  
public class FirstValueDateUDAF extends AbstractGenericUDAFResolver {  
  
 static final Logger *logger* = LoggerFactory.*getLogger*(FirstValueDateUDAF.class);  
  
 */\*\*  
 \* 做类型和参数长度验证  
 \*  
 \** ***@param*** *parameters  
 \** ***@return*** *\** ***@throws*** *SemanticException  
 \*/* public GenericUDAFEvaluator getEvaluator(TypeInfo[] parameters) throws SemanticException {  
 if (parameters == null) {  
 new FirstValueByDateEvaluator();  
 }  
  
 ObjectInspector oi = TypeInfoUtils.*getStandardJavaObjectInspectorFromTypeInfo*(parameters[0]);  
  
 if (oi.getCategory() != ObjectInspector.Category.*PRIMITIVE*) {  
 throw new UDFArgumentTypeException(0,  
 "Argument must be PRIMITIVE, but "  
 + oi.getCategory().name()  
 + " was passed.");  
 }  
  
 PrimitiveObjectInspector inputOI = (PrimitiveObjectInspector) oi;  
  
 if (inputOI.getPrimitiveCategory() != PrimitiveObjectInspector.PrimitiveCategory.*STRING*) {  
 throw new UDFArgumentTypeException(0,  
 "Argument must be String, but "  
 + inputOI.getPrimitiveCategory().name()  
 + " was passed.");  
 }  
 *logger*.info("create udaf");  
 return new FirstValueByDateEvaluator();  
 }  
  
 public static class FirstValueByDateEvaluator extends GenericUDAFEvaluator {  
  
 @Override  
 public ObjectInspector init(Mode m, ObjectInspector[] parameters)  
 throws HiveException {  
 super.init(m, parameters);  
 ObjectInspector outputOI;  
 // 指定各个阶段输出数据格式都为String类型  
 outputOI = ObjectInspectorFactory.*getReflectionObjectInspector*(String.class,  
 ObjectInspectorFactory.ObjectInspectorOptions.*JAVA*);  
 return outputOI;  
 }  
  
 */\*\*  
 \* 存储根据时间得到的类  
 \*/* static class InfoAgg implements AggregationBuffer {  
 String info;  
 boolean ascFlag = true;  
 public void reset() {  
 *logger*.info("wow invoke the reset");  
 ascFlag = true;  
 info = "";  
 }  
 }  
  
 @Override  
 public AggregationBuffer getNewAggregationBuffer() {  
 return new InfoAgg();  
 }  
  
 @Override  
 public void reset(AggregationBuffer agg) {  
 InfoAgg infoAgg = (InfoAgg) agg;  
 infoAgg.reset();  
 }  
  
 */\*\*  
 \* 遍历参数  
 \*  
 \** ***@param*** *agg  
 \** ***@param*** *parameters  
 \** ***@throws*** *HiveException  
 \*/* @Override  
 public void iterate(AggregationBuffer agg, Object[] parameters) {  
 if (parameters != null) {  
 InfoAgg infoAgg = (InfoAgg) agg;  
 if (parameters.length == 2) {  
 infoAgg.ascFlag = "asc".equalsIgnoreCase(parameters[1].toString().trim());  
 }  
 if (parameters[0] != null) {  
 if (infoAgg.info != null && !infoAgg.info.isEmpty()) {  
 String dataInfo = parameters[0].toString();  
 rankVal(infoAgg, dataInfo);  
 } else {  
 infoAgg.info = parameters[0].toString();  
 }  
 }  
 }  
 }  
  
 private void rankVal(InfoAgg infoAgg, String dataInfo) {  
 // 拿出 yyyy-MM-dd 这种类型的日期  
 int dat = Integer.*parseInt*(dataInfo.split("\\,")[0].replace("-", ""));  
 if (infoAgg.info == null || infoAgg.info.isEmpty()) {  
 infoAgg.info = dataInfo;  
 } else {  
 int aggDat = Integer.*parseInt*(infoAgg.info.split("\\,")[0].replace("-", ""));  
 if ((aggDat > dat && infoAgg.ascFlag) || (aggDat < dat && !infoAgg.ascFlag)) {  
 infoAgg.info = dataInfo;  
 }  
 }  
 }  
  
 */\*\*  
 \* 部分map的返回结果  
 \*  
 \** ***@param*** *agg  
 \** ***@return*** *\** ***@throws*** *HiveException  
 \*/* @Override  
 public Object terminatePartial(AggregationBuffer agg) {  
 InfoAgg infoAgg = (InfoAgg) agg;  
 return infoAgg.info;  
 }  
  
 */\*\*  
 \* 合并  
 \*  
 \** ***@param*** *agg  
 \** ***@param*** *partial  
 \** ***@throws*** *HiveException  
 \*/* @Override  
 public void merge(AggregationBuffer agg, Object partial) {  
 InfoAgg infoAgg = (InfoAgg) agg;  
 if (partial != null) {  
 String dataInfo = partial.toString();  
// 拿出 yyyy-MM-dd 这种类型的日期  
 rankVal(infoAgg, dataInfo);  
 }  
 }  
  
 */\*\*  
 \* 最终汇总  
 \*  
 \** ***@param*** *agg  
 \** ***@return*** *\** ***@throws*** *HiveException  
 \*/* @Override  
 public Object terminate(AggregationBuffer agg) {  
 InfoAgg infoAgg = (InfoAgg) agg;  
 return terminatePartial(agg);  
 }  
 // UDAF logic goes here!  
 }  
}

出现异常：

java.lang.NullPointerException

at java.util.concurrent.ConcurrentHashMap.get(ConcurrentHashMap.java:936)

at org.apache.hadoop.hive.serde2.typeinfo.TypeInfoUtils.getStandardWritableObjectInspectorFromTypeInfo(TypeInfoUtils.java:569)

at org.apache.hadoop.hive.ql.exec.ColumnInfo.<init>(ColumnInfo.java:81)

at org.apache.hadoop.hive.ql.exec.ColumnInfo.<init>(ColumnInfo.java:68)

at org.apache.hadoop.hive.ql.parse.SemanticAnalyzer.genGroupByPlanMapGroupByOperator(SemanticAnalyzer.java:5085)

at org.apache.hadoop.hive.ql.parse.SemanticAnalyzer.genGroupByPlanMapAggrNoSkew(SemanticAnalyzer.java:6097)

at org.apache.hadoop.hive.ql.parse.SemanticAnalyzer.genBodyPlan(SemanticAnalyzer.java:9370)

at org.apache.hadoop.hive.ql.parse.SemanticAnalyzer.genPlan(SemanticAnalyzer.java:10269)

at org.apache.hadoop.hive.ql.parse.SemanticAnalyzer.genPlan(SemanticAnalyzer.java:10147)

at org.apache.hadoop.hive.ql.parse.SemanticAnalyzer.genOPTree(SemanticAnalyzer.java:10820)

at org.apache.hadoop.hive.ql.parse.CalcitePlanner.genOPTree(CalcitePlanner.java:371)

at org.apache.hadoop.hive.ql.parse.SemanticAnalyzer.analyzeInternal(SemanticAnalyzer.java:10831)

at org.apache.hadoop.hive.ql.parse.CalcitePlanner.analyzeInternal(CalcitePlanner.java:246)

at org.apache.hadoop.hive.ql.parse.BaseSemanticAnalyzer.analyze(BaseSemanticAnalyzer.java:250)

at org.apache.hadoop.hive.ql.Driver.compile(Driver.java:477)

at org.apache.hadoop.hive.ql.Driver.compileInternal(Driver.java:1242)

at org.apache.hadoop.hive.ql.Driver.runInternal(Driver.java:1384)

at org.apache.hadoop.hive.ql.Driver.run(Driver.java:1171)

at org.apache.hadoop.hive.ql.Driver.run(Driver.java:1161)

at org.apache.hadoop.hive.cli.CliDriver.processLocalCmd(CliDriver.java:232)

at org.apache.hadoop.hive.cli.CliDriver.processCmd(CliDriver.java:183)

at org.apache.hadoop.hive.cli.CliDriver.processLine(CliDriver.java:399)

at org.apache.hadoop.hive.cli.CliDriver.executeDriver(CliDriver.java:776)

at org.apache.hadoop.hive.cli.CliDriver.run(CliDriver.java:714)

at org.apache.hadoop.hive.cli.CliDriver.main(CliDriver.java:641)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:498)

at org.apache.hadoop.util.RunJar.run(RunJar.java:221)

at org.apache.hadoop.util.RunJar.main(RunJar.java:136)

的原因，是因为init这个方法没有实现

### 脚本

初始化数据

insert overwrite table mid\_user\_history\_dt

partition (dt)

select user\_id,

version\_code,

version\_name,

lang,

source,

first\_version,

first\_dat,

first\_source,

last\_dt,dt from (select

row\_number() over (partition by user\_id order by server\_time) rn,

user\_id,

version\_code,

version\_name,

lang,source,

version\_name first\_version,

dt first\_dat,

source first\_source,

dt last\_dt,

dt

from ods\_basedata\_dt where dt='2018-11-03') t where rn=1;

-- 先创建一个临时表，去掉重复数据，通过group by来清理掉所有的数据

drop table if exists tmp\_user\_today;

create table tmp\_user\_today as

select user\_id,

area,

version\_name,

lang,

source,

first\_version,

first\_dat,

first\_source,

last\_dat,dt from (select

row\_number() over (partition by user\_id order by server\_time) rn,

user\_id,

area,

version\_name,

lang,source,

version\_name first\_version,

dt first\_dat,

source first\_source,

dt last\_dat,

dt

from ods\_basedata\_dt where dt='2018-11-08') t where rn=1;

再清洗

add jar /home/hadoop/hive-function-1.0-SNAPSHOT.jar;

create temporary function first\_val as 'com.tom.udaf.FirstValueDateUDAF';

set hive.exec.dynamic.partition.mode=nonstrict;

use dw\_weather;

insert overwrite table mid\_user\_history\_dt partition (dt)

select user\_id,

split(first\_val(current\_info,"desc"),",")[0] last\_dat,

split(first\_val(current\_info,"desc"),",")[1] area,

split(first\_val(current\_info,"desc"),",")[2] version\_name,

split(first\_val(current\_info,"desc"),",")[3] lang,

split(first\_val(current\_info,"desc"),",")[4] source,

split(first\_val(old\_info),",")[0] first\_dat,

split(first\_val(old\_info),",")[1] first\_version,

split(first\_val(old\_info),",")[2] first\_source,

'2018-11-08' dt

from (

select

user\_id,

concat\_ws(',',last\_dat,area,version\_name,lang,source) current\_info,

concat\_ws(',',first\_dat,first\_version,first\_source) old\_info

from mid\_user\_history\_dt where dt='2018-11-03'

union

select

user\_id,

concat\_ws(',',last\_dat,area,version\_name,lang,source) current\_info,

concat\_ws(',',first\_dat,first\_version,first\_source) old\_info

from tmp\_user\_today ) t group by user\_id;

### 小结

* 历史用户表是非常重要的一个表。我们的历史用户表有5千多万的用户出现过。
* UDAF的调试，可以使用异常抛出，来进行调试，但相对比较麻烦。

## 新增用户统计

### 需求分析

此处需要先计算出历史用户表，然后再根据用户**第一次出现的时间**为当天时间，算出为新增的用户。

### 脚本

每日新增：

每日新增用户则是计算用户的出现时间和当前时间是一样的，这种用户算作新增用户。所有新增用户都通过前端生成的唯一id来进行标识。

select count(user\_id) from mid\_user\_history\_dt where dt='2018-09-26' and current\_dat=dt

每周新增：

出现时间和最后一次出现时间是在7天之内。

select count(user\_id) from mid\_user\_history\_dt where dt='2018-09-26' and first\_dat<=dt  
and first\_dat> date\_add(dt, -7)

每月新增：

每月新增是用户的第一次出现和最后一次出现是在本月之内。直接通过时间进行比较就行。

select count(user\_id) from mid\_user\_history\_dt where dt='2018-09-26' and first\_dat<=dt  
and first\_dat> date\_add(dt, -30)

## 前台，后台活跃用户数统计（日，月）

创建一个表

统计每个日志的**唯一用户数.**

select t1.day\_foreground,t3.month\_foreground,t2.day\_background,t4.month\_background,t1.dt from (

select count(distinct user\_id) day\_foreground,dt from ods\_start\_dt where dt='2018-11-08' and action='1' group by dt

) t1 join

(select count(distinct user\_id) month\_foreground from ods\_start\_dt where dt>='2018-11-01' and dt<='2018-11-08' and action='1') t3 on 1=1

join

(select count(distinct user\_id) day\_background,dt from ods\_background\_dt where dt='2018-11-08' group by dt) t2 on t1.dt=t2.dt

join

(select count(distinct user\_id) month\_background from ods\_background\_dt where dt>='2018-11-01' and dt<='2018-11-08' ) t4 on 1=1;

## 用户版本分布统计

用户版本分析包括前台和后台活跃用户所在的版本分布，也是通过用户id的唯一来标识的。

### 建表语句

drop table if exists dm\_version\_allocation;

create table dm\_version\_allocation (

version\_name string,

user\_num\_fore int,

user\_num\_back int

)PARTITIONED BY (`dt` string)

location '/tmp/teach/dm\_version\_allocation/';

### 脚本

insert overwrite table dm\_version\_allocation

partition(dt='2018-11-08')

select t1.\*,t2.user\_num\_back from (

select version\_name,count(distinct user\_id) user\_num\_fore from ods\_start\_dt where dt='2018-11-08' and action='1' group by version\_name

) t1 join

(select version\_name,count(distinct user\_id) user\_num\_back from ods\_background\_dt where dt='2018-11-08' group by version\_name) t2 on t1.version\_name=t2.version\_name order by t1.version\_name;

## 沉默用户统计

沉默用户是这样。可以分析**第一次出现和最后一次出现**的日期相等，并且是2天前的用户。此报表依然依赖于每日历史用户分析表。

* 从历史用户表中找出前天出现，并且最后一天也是前天的用户
* 计算出用户数

select count(distinct user\_id) num from mid\_user\_history\_dt  
 where dt='2018-09-30' and first\_dat=current\_dat and first\_dat=date\_add(dt,-2);

## 新鲜度分析

用户新鲜度 = 某段时间的新增用户数/某段时间的活跃的用户数

今天新增用户（为 n）

select count(user\_id) from dw\_history\_users where dt='2018-09-26' and current\_dat=dt

今天活跃用户（m）

select count(distinct uid) forground\_total,dt from dw\_foreground where dt='2018-09-25' group by select count(distinct uid) forground\_total,dt from dw\_foreground where dt='2018-09-25' group by dt;

3. 新鲜度 = n / m

注意判断 m 等于 0 的情况

## 每日历史新闻分析

所有新闻的历史信息，并且保存。每天进行更新。用来分析新闻

* 跟昨天的新闻进行比较，更新新闻的出现时间
* 需要统计新闻的总共点击和展示

create temp view tmp\_today\_news as  
 select t1.\*,t2.total\_click,t2.total\_display from  
 (  
select news\_id,area,min(first\_display\_time) first\_display\_time, max(last\_display\_time) last\_display\_time, min(first\_click\_time) first\_click\_time, max(last\_click\_time) last\_click\_time from (  
select news\_id,area,min(server\_time) first\_display\_time,max(server\_time) last\_display\_time,null first\_click\_time,null last\_click\_time from ods\_display\_dt where dt='2018-09-30' and action='1' group by news\_id,area  
union  
select news\_id,area,null first\_display\_time,null last\_display\_time,min(server\_time) first\_click\_time,max(server\_time) last\_click\_time from ods\_display\_dt where dt='2018-09-30' and action='2' group by news\_id,area  
)  
group by news\_id,area  
) t1 join mid\_daily\_news\_dt t2  
on t1.news\_id =t2.news\_id;  
  
insert overwrite table mid\_news\_history\_dt  
partition (dt='2018-09-30')  
select  
news\_id,area,  
min(first\_display\_time) first\_display\_time,  
max(last\_display\_time) last\_display\_time,  
min(first\_click\_time) first\_click\_time,  
max(last\_click\_time) last\_click\_time,  
sum(total\_display) total\_display,  
sum(total\_click) total\_click  
from  
(  
select \* from tmp\_today\_news  
union all  
select news\_id,area,first\_display\_time,last\_display\_time,first\_click\_time,last\_click\_time,total\_display,total\_click from mid\_news\_history\_dt where dt='2018-09-29'  
) group by news\_id,area;

## 每日新闻展示点击详情

通过聚合函数来计算每日新闻的总量，新闻展示

创建点击和展示的详细临时表

drop table if exists tmp\_news\_display;

create table tmp\_news\_display as

select newsid,lang,version\_name,area,user\_id from ods\_display\_dt

where dt='2018-11-08' and action='1';

drop table if exists tmp\_news\_click;

create table tmp\_news\_click as

select newsid,lang,version\_name,area,user\_id from ods\_display\_dt

where dt='2018-11-08' and action='2';

将数据插入

insert overwrite table mid\_daily\_news\_dt

partition(dt='2018-11-08')

select t1.\*,t2.total\_click from (

select newsid news\_id,

lang,version\_name,area,count(distinct user\_id) total\_display from tmp\_news\_display

group by newsid,lang,version\_name,area

) t1

join

(select newsid news\_id,

lang,version\_name,area,count(distinct user\_id) total\_click from tmp\_news\_click

group by newsid,lang,version\_name,area

) t2

on t1.news\_id=t2.news\_id and t1.lang=t2.lang and t1.version\_name=t2.version\_name;

这样得到的数据是经过去重了的数据。这个表里面有好几个维度

## 新闻的总点击总展示（1天，3天，7天）

这个时候需要用到right join，因为7天的肯定大于3天和1天的。其中用到了date\_add函数，他是天生支持yyyy-MM-dd这种格式的数据

select t3.\*,t2.total\_display3,t2.total\_click3,t1.total\_display,t1.total\_click from (

select news\_id,sum(total\_display) total\_display,sum(total\_click) total\_click

from mid\_daily\_news\_dt where dt='2018-11-08' group by news\_id

) t1 right join

(select news\_id,sum(total\_display) total\_display3,sum(total\_click) total\_click3

from mid\_daily\_news\_dt where dt>date\_add('2018-11-08',-3) group by news\_id) t2

on t1.news\_id=t2.news\_id

right join

(select news\_id,sum(total\_display) total\_display7,sum(total\_click) total\_click7

from mid\_daily\_news\_dt where dt>date\_add('2018-11-08',-7) group by news\_id) t3

on t1.news\_id=t3.news\_id;

## 新闻不同地区的总点击总展示（1天，3天，7天）

这个时候需要加一个维度，就是地区的维度，然后在join的时候，需要将新闻id和地区两个进行join。

select t3.\*,t2.total\_display3,t2.total\_click3,t1.total\_display,t1.total\_click from (

select news\_id,area,sum(total\_display) total\_display,sum(total\_click) total\_click

from mid\_daily\_news\_dt where dt='2018-11-08' group by news\_id,area

) t1 right join

(select news\_id,area,sum(total\_display) total\_display3,sum(total\_click) total\_click3

from mid\_daily\_news\_dt where dt>date\_add('2018-11-08',-3) group by news\_id,area) t2

on t1.news\_id=t2.news\_id and t1.area=t2.area

right join

(select news\_id,area,sum(total\_display) total\_display7,sum(total\_click) total\_click7

from mid\_daily\_news\_dt where dt>date\_add('2018-11-08',-7) group by news\_id,area) t3

on t1.news\_id=t3.news\_id and t1.area=t2.area;

## 曝光新闻条数

曝光新闻条数就是展示给用户看到的新闻条数，在埋点中我们定义了action=1的这种情况

select count(distinct news\_id) news\_num from ods\_display\_dt where dt='2018-11-08' and action='1'

## 曝光用户数

select count(distinct user\_id) user\_num from ods\_display\_dt where dt='2018-11-03' and action='1'

## 新闻平均曝光次数

3个报表合在一起。一天有多少新闻曝光，多少个用户被曝光了新闻，一个用户被曝光了多少次

select news\_num,user\_num,news\_num/user\_num avg\_display\_news from (

select count(distinct news\_id) news\_num from ods\_display\_dt where dt='2018-11-08' and action='1') t1 cross join

(select count(distinct user\_id) user\_num from ods\_display\_dt where dt='2018-11-03' and action='1') t2

## 曝光新闻次数

一天曝光了多少次新闻。这个是用日志的条数来决定

select count(0) news\_num from ods\_display\_dt where dt='2018-11-08' and action='1';

## 人均新闻曝光次数

每个人被曝光了多少次新闻

select news\_display\_num,user\_num,news\_display\_num/user\_num avg\_display\_news from (

select count(0) news\_display\_num from ods\_display\_dt where dt='2018-11-08' and action='1') t1 cross join

(select count(distinct user\_id) user\_num from ods\_display\_dt where dt='2018-11-03' and action='1') t2

## 点击新闻用户量

多少用户点击了新闻

select count(distinct user\_id) click\_user\_num from ods\_display\_dt where dt='2018-11-03' and action='2';

## 点击新闻条数

-- 点击新闻条数

select count(distinct news\_id) click\_news\_num from ods\_display\_dt where dt='2018-11-03' and action='2';

## 人均新闻点击量

每个人点击了多少新闻

select click\_user\_num,click\_news\_num,click\_news\_num/(click\_user\_num\*1.0) avg\_click\_amount from

(select count(distinct user\_id) click\_user\_num from ods\_display\_dt where dt='2018-11-03' and action='2') t1 cross join

(select count(distinct news\_id) click\_news\_num from ods\_display\_dt where dt='2018-11-03' and action='2') t2

## 点击新闻次数

select count(0) click\_news\_num from ods\_display\_dt where dt='2018-11-03' and action='2';

## 人均新闻点击次数

select click\_user\_num,click\_news\_times,(click\_news\_times\*1.0)/click\_user\_num avg\_click\_times from

(select count(distinct user\_id) click\_user\_num from ods\_display\_dt where dt='2018-11-03' and action='2') t1 cross join

(select count(0) click\_news\_times from ods\_display\_dt where dt='2018-11-03' and action='2') t2

## 新闻展示用户数

展示这个概念需要将曝光和点击合并到一起

select count(distinct user\_id) user\_num from ods\_display\_dt where dt='2018-11-03'

## 新闻展示次数

次数用日志的数量来决定，

select count(0) display\_num from ods\_display\_dt where dt='2018-11-03'

## 新闻人均展示数

将三个报表合在一起，然后新闻展示次数/人就是人均展示次数

select user\_num,display\_num,display\_num/(user\_num\*1.0) rate from (

select count(distinct user\_id) user\_num from ods\_display\_dt where dt='2018-11-03' ) t1 cross join

(select count(0) display\_num from ods\_display\_dt where dt='2018-11-03' ) t2;

## 当天所有新闻总点击率

select display\_num,click\_num,display\_num/click\_num from

(select count(distinct user\_id) click\_num from ods\_display\_dt where dt='2018-11-08' and action='2') t1 cross join

(select count(distinct user\_id) display\_num from ods\_display\_dt where dt='2018-11-08' and action='1') t2;

# 第6章 实时数据处理系统

## 6.1 实时热度

### 6.1.1 需求

1. 需求：根据函数实时算出新闻热度，
2. 并且对新闻进行排序

上游的数据是用户上报的日志，下游的处理系统是后台，拉到我们的数据的时候，直接给用户拉取。

### 6.1.2 Mysql脚本

用rds做临时存储24小时数据， 并且计算热度

-- 用户24小时点击记录

drop table if exists news\_display;

create table news\_display(

area varchar(20),

news\_id varchar(20),

user\_id varchar(20),

time\_stam long

);

alter table news\_display add unique(area,news\_id,user\_id);

--用户24小时展示记录

drop table if exists news\_click;

create table news\_click(

area varchar(20),

news\_id varchar(20),

user\_id varchar(20),

time\_stam long

);

alter table news\_click add unique(area,news\_id,user\_id);

-- 存临时插入的数据

drop table if exists tmp\_display\_click;

create table tmp\_display\_click(

action int(2),

area varchar(20),

news\_id varchar(20),

user\_id varchar(20),

time\_stam long

);

alter table tmp\_display\_click add index(action,area,news\_id,user\_id);

### 6.1.3 MysqlService 脚本

MysqlService

package com.teach.business;  
  
import java.sql.\*;  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.List;  
import java.util.Map;  
  
public class MysqlService {  
  
  
 // 三剑客  
 static Connection *con* = null;// 连接对象  
 static PreparedStatement *pstmt* = null;// 语句对象  
 static ResultSet *rs* = null;// 结果集对象  
  
 */\*\*  
 \* 获得连接对象  
 \*  
 \** ***@return*** *连接对象  
 \** ***@throws*** *ClassNotFoundException  
 \** ***@throws*** *SQLException  
 \*/* public static Connection getConnection() throws ClassNotFoundException,  
 SQLException {  
 Class.*forName*("com.mysql.jdbc.Driver");  
 *con* = DriverManager.*getConnection*(ReadNewsStream.*serverProps*.getProperty("mysql.url"),  
 ReadNewsStream.*serverProps*.getProperty("mysql.user"),  
 ReadNewsStream.*serverProps*.getProperty("mysql.password"));  
 return *con*;  
 }  
  
 */\*\*  
 \* 关闭三剑客  
 \*  
 \** ***@throws*** *SQLException  
 \*/* public static void close(ResultSet rs, PreparedStatement pstmt, Connection con) {  
  
 try {  
 if (rs != null)  
 rs.close();  
 if (pstmt != null)  
 pstmt.close();  
 if (con != null)  
 con.close();  
 } catch (SQLException e) {  
 // *TODO: handle exception* e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* 执行更新  
 \*  
 \** ***@param*** *sql 传入的预设的 sql语句  
 \** ***@param*** *params 问号参数列表  
 \** ***@return*** *影响行数  
 \*/* public static int execUpdate(String sql, Object[] params) {  
 try {  
 *getConnection*();// 获得连接对象  
 *pstmt* = *con*.prepareStatement(sql);// 获得预设语句对象  
  
 if (params != null) {  
 // 设置参数列表  
 for (int i = 0; i < params.length; i++) {  
 // 因为问号参数的索引是从1开始，所以是i+1，将所有值都转为字符串形式，好让setObject成功运行  
 *pstmt*.setObject(i + 1, params[i] + "");  
 }  
 }  
  
 return *pstmt*.executeUpdate(); // 执行更新，并返回影响行数  
  
 } catch (ClassNotFoundException | SQLException e) {  
 // *TODO Auto-generated catch block* e.printStackTrace();  
 } finally {  
 *close*(*rs*, *pstmt*, *con*);  
 }  
 return 0;  
 }  
  
 */\*\*  
 \* 执行查询  
 \*  
 \** ***@param*** *sql 传入的预设的 sql语句  
 \** ***@param*** *params 问号参数列表  
 \** ***@return*** *查询后的结果  
 \*/* public static List<Map<String, Object>> execQuery(String sql, Object[] params) {  
  
 try {  
 *getConnection*();// 获得连接对象  
 *pstmt* = *con*.prepareStatement(sql);// 获得预设语句对象  
  
 if (params != null) {  
 // 设置参数列表  
 for (int i = 0; i < params.length; i++) {  
 // 因为问号参数的索引是从1开始，所以是i+1，将所有值都转为字符串形式，好让setObject成功运行  
 *pstmt*.setObject(i + 1, params[i] + "");  
 }  
 }  
  
 // 执行查询  
 ResultSet rs = *pstmt*.executeQuery();  
  
 List<Map<String, Object>> al = new ArrayList<>();  
  
 // 获得结果集元数据（元数据就是描述数据的数据，比如把表的列类型列名等作为数据）  
 ResultSetMetaData rsmd = rs.getMetaData();  
  
 // 获得列的总数  
 int columnCount = rsmd.getColumnCount();  
  
 // 遍历结果集  
 while (rs.next()) {  
 Map<String, Object> hm = new HashMap<String, Object>();  
 for (int i = 0; i < columnCount; i++) {  
 // 根据列索引取得每一列的列名,索引从1开始  
 String columnName = rsmd.getColumnName(i + 1);  
 // 根据列名获得列值  
 Object columnValue = rs.getObject(columnName);  
 // 将列名作为key，列值作为值，放入 hm中，每个 hm相当于一条记录  
 hm.put(columnName, columnValue);  
 }  
 // 将每个 hm添加到al中, al相当于是整个表，每个 hm是里面的一条记录  
 al.add(hm);  
 }  
  
 return al;  
  
 } catch (ClassNotFoundException | SQLException e) {  
 e.printStackTrace();  
 } finally {  
 *close*(*rs*, *pstmt*, *con*);  
 }  
 return null;  
 }  
}

### 6.1.4 ReadNewsStream 脚本

package com.teach.business;  
  
import com.fasterxml.jackson.databind.JsonNode;  
import com.fasterxml.jackson.databind.ObjectMapper;  
import org.apache.hadoop.conf.Configuration;  
import org.apache.hadoop.fs.FileSystem;  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.hdfs.DistributedFileSystem;  
import org.apache.kafka.clients.consumer.ConsumerRecord;  
import org.apache.kafka.clients.producer.KafkaProducer;  
import org.apache.kafka.clients.producer.ProducerRecord;  
import org.apache.spark.SparkConf;  
import org.apache.spark.api.java.JavaRDD;  
import org.apache.spark.api.java.function.ForeachPartitionFunction;  
import org.apache.spark.api.java.function.Function0;  
import org.apache.spark.broadcast.Broadcast;  
import org.apache.spark.sql.\*;  
import org.apache.spark.sql.api.java.UDF3;  
import org.apache.spark.sql.types.DataTypes;  
import org.apache.spark.sql.types.StructField;  
import org.apache.spark.sql.types.StructType;  
import org.apache.spark.streaming.Durations;  
import org.apache.spark.streaming.api.java.JavaInputDStream;  
import org.apache.spark.streaming.api.java.JavaStreamingContext;  
import org.apache.spark.streaming.kafka010.\*;  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import static org.apache.spark.sql.functions.col;  
  
import java.util.\*;  
  
  
public class ReadNewsStream {  
 private static final Logger logger = LoggerFactory.getLogger(ReadNewsStream.class);  
 public static final Properties serverProps = PropertiesUtils.getProperties("config.properties");  
 static Broadcast<Properties> serverPropsBroadcast=null;  
  
  
 public static void main(String[] args) throws Exception {  
 /\* 获取 checkpoint 的 hdfs 路径 \*/  
 String checkpointPath = serverProps.getProperty("streaming.checkpoint.path");  
 /\* 如果 checkpointPath hdfs 目录下的有文件，则反序列化文件生产 context, 否则使用函数 createContext 返回的 context 对象 \*/  
 JavaStreamingContext javaStreamingContext = JavaStreamingContext.getOrCreate(checkpointPath, createContext(serverProps));  
 javaStreamingContext.start();  
 javaStreamingContext.awaitTermination();  
 }  
  
 */\*\*  
 \* 根据配置文件以及业务逻辑创建 JavaStreamingContext  
 \*  
 \** ***@param*** *serverProps  
 \** ***@return*** *\*/* public static Function0<JavaStreamingContext> createContext(final Properties serverProps) {  
 Function0<JavaStreamingContext> createContextFunc = new Function0<JavaStreamingContext>() {  
 public JavaStreamingContext call() throws Exception {  
 //获取配置中的 topic  
 logger.info("配置是" + serverProps.toString());  
 String topicStr = serverProps.getProperty("kafka.topic");  
 Collection<String> topics = Arrays.asList(topicStr.split(","));  
 //获取配置中的 groupId  
 final String groupId = serverProps.getProperty("kafka.groupId");  
 //获取批次的时间间隔，比如 5s  
 final Long streamingInterval = Long.parseLong(serverProps.getProperty("streaming.interval"));  
 //获取 checkpoint 的 hdfs 路径  
 final String checkpointPath = serverProps.getProperty("streaming.checkpoint.path");  
 //获取 kafka broker 列表  
 final String kafkaBrokerList = serverProps.getProperty("kafka.broker.list");  
 //组合 kafka 参数  
 final Map<String, Object> kafkaParams = new HashMap();  
 kafkaParams.put("bootstrap.servers", kafkaBrokerList);  
 kafkaParams.put("group.id", groupId);  
 kafkaParams.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  
 kafkaParams.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  
  
 // 创建 SparkConf 对象  
 SparkConf sparkConf = new SparkConf().setMaster("local[\*]").setAppName("spark-news");  
  
 */\*\*  
 优雅停止 Spark. 暴力停掉 sparkstreaming 是有可能出现问题的，比如你的数据源是 kafka，  
 已经加载了一批数据到 sparkstreaming 中正在处理，如果中途停掉，  
 这个批次的数据很有可能没有处理完，就被强制 stop 了，  
 下次启动时候会重复消费或者部分数据丢失。  
 \*/* sparkConf.set("spark.streaming.stopGracefullyOnShutdown", "true");  
  
 /\*在 Spark 的架构中，在网络中传递的或者缓存在内存、硬盘中的对象需要进行序列化操作，序列化的作用主要是利用时间换空间\*/  
 sparkConf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");  
  
 /\*增加 MyRegistrator 类，注册需要用 Kryo 序列化的类,Kryo 的序列化会比java的要更省空间\*/  
 sparkConf.set("spark.kryo.registrator", "com.teach.business.MyKryoRegistor");  
  
 /\* 每秒钟对于每个 partition 读取spark.streaming.backpressure.enabled多少条数据。如果不进行设置，Spark Streaming 会一开始就读取 partition 中的所有数据到内存，给内存造成巨大压力  
 设置此参数后可以很好地控制 Spark Streaming 读取的数据量，也可以说控制了读取的进度 \*/  
 sparkConf.set("spark.streaming.kafka.maxRatePerPartition", serverProps.getProperty("streaming.kafka.maxRatePerPartition"));  
  
 /\* 创建 javaStreamingContext，设置 每隔5s 执行一次\*/  
 JavaStreamingContext javaStreamingContext = new JavaStreamingContext(sparkConf, Durations.seconds(streamingInterval));  
  
 serverPropsBroadcast=javaStreamingContext.sparkContext().broadcast(serverProps);  
  
 /\* check point 的存在，每次都会从中读取元数据和RDD，则每次的代码修改是无效。因为kafka的offset存在，所以checkPoint可以不用。仅做参考 \*/  
// javaStreamingContext.checkpoint(checkpointPath);  
  
 //创建 kafka DStream  
 final JavaInputDStream<ConsumerRecord<String, String>> kafkaMessage = KafkaUtils.createDirectStream(javaStreamingContext, LocationStrategies.PreferConsistent(),  
 ConsumerStrategies.<String, String>Subscribe(topics, kafkaParams));  
  
  
 /\* spark sql 会话 \*/  
 logger.info("spark conf 是{}",javaStreamingContext.sparkContext().sc().conf());  
 SparkSession sparkSession = SparkSession.builder().enableHiveSupport().sparkContext(javaStreamingContext.sparkContext().sc()).config(sparkConf).getOrCreate();  
  
 /\* 启动的时候 先删除 停止路径 \*/  
 Configuration conf = javaStreamingContext.sparkContext().sc().hadoopConfiguration();  
 FileSystem fs = DistributedFileSystem.get(conf);  
 Path stopPath=new org.apache.hadoop.fs.Path(serverProps.getProperty("streaming.stop.path"));  
 if(fs.exists(stopPath)){  
 logger.info("删除历史中的stop path");  
 fs.delete(stopPath,true);  
 }  
  
 // 需要把每个批次的 offset 保存  
 kafkaMessage.foreachRDD(rdd -> {  
 logger.info("开始迭代");  
  
 /\* 如果stop path出现,就停止系统 \*/  
 if(fs.exists(stopPath)){  
 logger.info("开始停止spark程序");  
 javaStreamingContext.stop(true);  
 }  
  
 /\* 表示具有[[OffsetRange]]集合的任何对象，这可以用来访问由直 Direct Kafka DStream 生成的 RDD 中的偏移量范围\*/  
 OffsetRange[] offsetRanges = ((HasOffsetRanges) rdd.rdd()).offsetRanges();  
 /\* 逻辑处理在这里\*/  
 executeData(rdd, sparkSession, serverProps);  
 /\* kafka offset 写入 zk\*/  
 ((CanCommitOffsets) kafkaMessage.inputDStream()).commitAsync(offsetRanges);  
 });  
 // 将 kafka 中的消息转换成对象并过滤不合法的消息  
 return javaStreamingContext;  
 }  
 };  
 return createContextFunc;  
 }  
  
 */\*\*  
 \* 处理业务逻辑  
 \** ***@param*** *rdd  
 \** ***@param*** *sparkSession  
 \** ***@param*** *serverProps  
 \*/* static private void executeData(JavaRDD<ConsumerRecord<String, String>> rdd, SparkSession sparkSession, Properties serverProps) {  
 //1.计算威尔逊热度  
 JavaRDD<Row> rowRDD = rdd.filter(stringStringConsumerRecord -> {  
 /\*过滤出来只有点击的日志\*/  
 String content = stringStringConsumerRecord.value();  
 return content.contains("display") && content.contains("|");  
 }).flatMap(stringStringConsumerRecord -> {  
 /\*组装成新闻的点击格式\*/  
 String val = stringStringConsumerRecord.value();  
 ObjectMapper objectMapper = new ObjectMapper();  
 Long timestam = Long.parseLong(val.split("\\|")[0]);  
 JsonNode jsonNode = objectMapper.readTree(val.split("\\|")[1]);  
 // 用户id，news id，area，时间  
 String userid = jsonNode.get("he").get("userId").asText();  
 String area = jsonNode.get("he").get("area").asText();  
 Iterator<JsonNode> iterater = jsonNode.get("et").iterator();  
 List<Row> results = new ArrayList<>();  
 while (iterater.hasNext()) {  
 JsonNode event = iterater.next();  
 if ("display".equals(event.get("eventName").asText())) {  
 int action = event.get("kv").get("action").asInt();  
 String newsId = event.get("kv").get("newsId").asText();  
 Row row = RowFactory.create(action, area, newsId, userid, timestam);  
 results.add(row);  
 }  
 }  
 return results.iterator();  
 });  
 ArrayList<StructField> fields = new ArrayList<>();  
 StructField field = null;  
 field = DataTypes.createStructField("action", DataTypes.IntegerType, true);  
 fields.add(field);  
 field = DataTypes.createStructField("area", DataTypes.StringType, true);  
 fields.add(field);  
 field = DataTypes.createStructField("news\_id", DataTypes.StringType, true);  
 fields.add(field);  
 field = DataTypes.createStructField("user\_id", DataTypes.StringType, true);  
 fields.add(field);  
 field = DataTypes.createStructField("time\_stam", DataTypes.LongType, true);  
 fields.add(field);  
 StructType schema = DataTypes.createStructType(fields);  
  
 if (!rowRDD.isEmpty()) {  
 Dataset<Row> df = sparkSession.createDataFrame(rowRDD, schema);  
 /\* 先对写入的日志去重，减少写入RDS的数据 \*/  
 df.registerTempTable("tmp\_input\_display");  
 df=sparkSession.sql("select action,area,news\_id,user\_id,time\_stam from tmp\_input\_display group by action,area,news\_id,user\_id,time\_stam");  
 /\* 写入rds,根据24小时的数据去重 \*/  
 Properties prop = new java.util.Properties();  
 prop.setProperty("driver", "com.mysql.jdbc.Driver");  
 prop.setProperty("user", serverProps.getProperty("mysql.user"));  
 prop.setProperty("password", serverProps.getProperty("mysql.password"));  
 df.write().mode(SaveMode.Overwrite).jdbc(serverProps.getProperty("mysql.url"), "tmp\_display\_click", prop);  
 /\* 去重点击和展示两个表 \*/  
 MysqlService.execUpdate("delete t1 from news\_display t1 join (select area,news\_id,user\_id,time\_stam from tmp\_display\_click where action=1) t2 " +  
 "on t1.area=t2.area and t1.news\_id=t2.news\_id and t1.user\_id=t2.user\_id", null);  
 MysqlService.execUpdate("insert ignore into news\_display select area,news\_id,user\_id,time\_stam from tmp\_display\_click where action=1", null);  
 MysqlService.execUpdate("delete t1 from news\_display t1 join (select area,news\_id,user\_id,time\_stam from tmp\_display\_click where action=2) t2 \n" +  
 "on t1.area=t2.area and t1.news\_id=t2.news\_id and t1.user\_id=t2.user\_id", null);  
 MysqlService.execUpdate("insert ignore into news\_click select area,news\_id,user\_id,time\_stam from tmp\_display\_click where action=2", null);  
 /\* 获取新闻的点击率 24小时内的数据比较少，可以直接在mysql里面join了。\*/  
 df=sparkSession.read().jdbc(serverProps.getProperty("mysql.url"), "(select t1.\*," +  
 "case when t2.click is not null then t2.click else 0 end click from \n" +  
 "(select count(distinct user\_id) display,news\_id,area from news\_display group by area,news\_id) t1 " +  
 "left join " +  
 "(select count(distinct user\_id) click ,news\_id,area from news\_click group by area,news\_id) t2 " +  
 "on t1.news\_id = t2.news\_id) click\_info", prop);  
// df.show();  
 df.registerTempTable("click\_info");  
 /\* 删除超过24小时的数据，此处因为测试，所以没写了。\*/  
 /\* 读取24小时内的新闻质量 \*/  
 long afterTime=System.currentTimeMillis()/1000-36\*24\*60\*60;  
 Dataset<Row> qualityDf=sparkSession.read().jdbc(serverProps.getProperty("mysql.url"), "(select news\_id,quality from news\_quality where time\_stam>"+afterTime+") tmp\_news\_quality", prop);  
 qualityDf.registerTempTable("tmp\_news\_quality");  
 sparkSession.sqlContext().udf().register("cal\_score",(UDF3<Long,Long,Double,Double>)( click, display, quality)-> {  
 /\* 根据点击率，质量，计算新闻的分数,此处是调用算法的函数，所以写了一个随机数 \*/  
 return new Random().nextDouble();  
 },DataTypes.DoubleType);  
  
 /\* 点击和质量数据进行join，然后算出打分，最后排序 \*/  
 df=sparkSession.sql("select t1.news\_id,t1.area,t1.display,t1.click,t2.quality,cal\_score(t1.click,t1.display,t2.quality) score from click\_info t1 join tmp\_news\_quality t2 " +  
 "on t1.news\_id=t2.news\_id");  
 df.show();  
 df.registerTempTable("tmp\_news\_score");  
  
 //生产者发送消息  
 df.foreachPartition((ForeachPartitionFunction<Row>) (iterator)->{  
 Properties props = new Properties();  
 props.put("bootstrap.servers", serverPropsBroadcast.getValue().getProperty("kafka.broker.list"));  
 props.put("acks", "all");  
 props.put("retries", 0);  
 props.put("batch.size", 16384);  
 props.put("linger.ms", 1);  
 props.put("buffer.memory", 33554432);  
 props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");  
 props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");  
 KafkaProducer<String, String> scoreProducer = new KafkaProducer<>(props);  
  
 while (iterator.hasNext()){  
 Row row=iterator.next();  
 String json=String.format("{\"news\_id\":\"%s\",\"score\":%s}",row.getString(0),row.getDouble(5));  
 ProducerRecord<String, String> msg = new ProducerRecord<>("news\_score\_test",json);  
 scoreProducer.send(msg);  
 }  
 scoreProducer.flush();  
 });  
// 最后将新闻排序,取前1000条，写入kafka，传给后台  
 df=sparkSession.sql("select news\_id from tmp\_news\_score order by score limit 1000");Properties props = new Properties();  
 props.put("bootstrap.servers", serverPropsBroadcast.getValue().getProperty("kafka.broker.list"));  
 props.put("acks", "all");  
 props.put("retries", 0);  
 props.put("batch.size", 16384);  
 props.put("linger.ms", 1);  
 props.put("buffer.memory", 33554432);  
 props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");  
 props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");  
 KafkaProducer<String, String> scoreProducer = new KafkaProducer<>(props);  
 scoreProducer.send(StringUtils.join(df.collect(),","));  
 }  
 }  
  
  
  
}

### 6.1.6 启动

其中 driver-memory, executor 数，可以用来调节spark程序

spark-submit \

--class com.tom.spark.SparkNewsCaculator \

--master yarn \

--deploy-mode cluster \

--queue migsz \

--driver-memory 5G \

--executor-memory 10G \

--num-executors 10 \

--executor-cores 4 \

--conf spark.shuffle.io.maxRetries=6

Spark-news.jar

### 6.1.5 小结

* 系统必须要有监控的脚本，当streaming的时间超时的时候，要杀掉进程，重新启动。简单的shell脚本就好
* 当上游数据过多的时候，系统会越来越慢，时间越来越长。我们是可以允许丢数据的这种情况。重启应用之后，更改consumer group的id，防止滞后这种情况，同时考虑优化系统，提高处理能力。

# 第7章 系统指标

* 大数据实时处理部分控制在5分钟之内。
* 所有离线数据报表控制在8小时之内
* 集群负载一般只要不宕机就没有问题，然后负载过高需要报警。一天平均的负载要在70%以下。
* HDFS和硬盘空闲控制在70%以下

# 第8章 项目总结

## 6.1 面试题

### 6.1.2 项目相关

整个数据流程中，如何保证数据一致性？

从日志到kafka过程，和从kafka到hdfs的过程中，分别用到了kafka的ack事务性和flume的事务性。而从数据事实表到数据产品，中间通过脚本的依赖，来保证数据的一致。

这个信息流是实时性的么，瓶颈在哪里？

准实时。整个时间是从api收集日志到计算完成新闻排序，中间经过了springboot，kafka，然后数据需要写一次RDS，读一次RDS，读一次HDFS最后得到相关的结果。

瓶颈在spark与rds，hdfs的多次交互，读取kafka全量数据。

为什么不用hbase而用RDS做热度统计？

Hbase：吞吐量高，可以直接去重，但聚合函数复杂,所以聚合函数是瓶颈

RDS：吞吐量略低，插入时候要创建索引。但在统计和计算点击率join的时候可以走索引，同时清楚历史数据也可以走索引，速度非常快。

项目配置：

机器，数据量，延时，日活

主要面java基础比较多。



这个系统有什么可以优化的地方？

* RDS可以用elasticsearch代替
* 可以在flume层对日志进行分离。将实时数据和离线数据分开，这样能减少spark读取量
* Flume可以考虑用其他的组件
* 数据仓库流程可视化，引入kettle

在这个系统中遇到的最大的问题是什么？

数据仓库：整个清洗流程是8小时左右。几亿条的数据。然后

* 多个报表的并行，yarn队列的划分能大大提高速度
* Sparksql替代tez能提高效率

你们系统是怎么容灾的呢？

* 我们将原始日志gzip压缩，然后传送到我们自己托管的服务器。这样哪怕亚马逊出现问题，也没有关系。
* 用亚马逊的s3多区域存储。将数据同步存在两个区域

用到了哪些工具。

Rundeck,datax, yanagishima（presto的可视化界面）

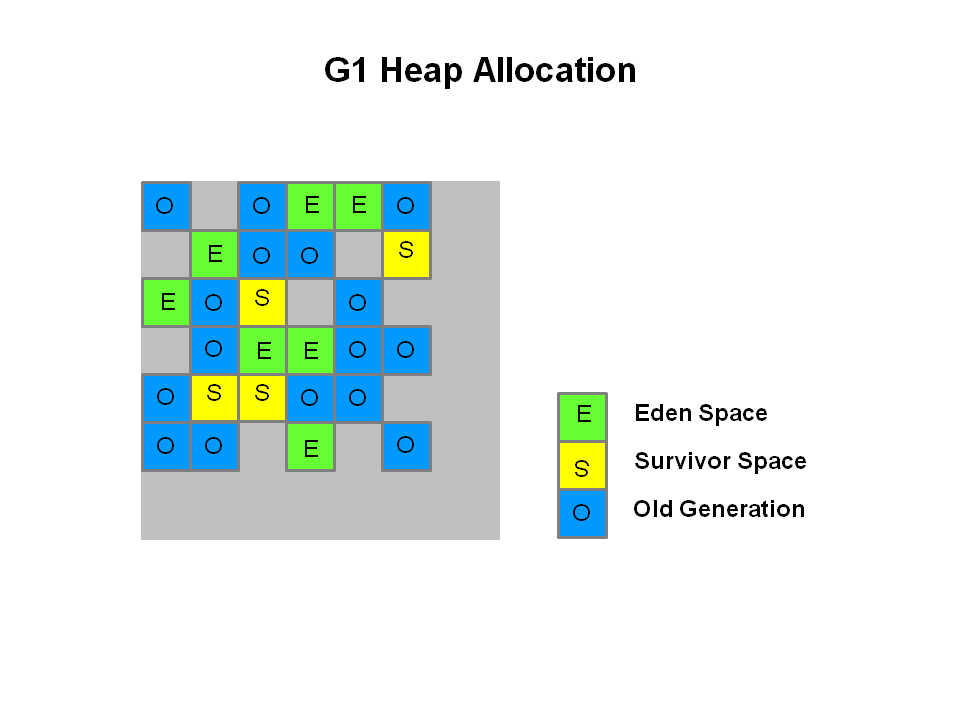
跨国之间的数据通过什么来传输？

S3可以跨国传输数据。不需要走公网。

### 6.1.3 技术相关

Java

* Java1.8的垃圾回收机制



1.8是将eden区，存活区，年老代，分成了很多小块，这样对象就可以分散到各个块中。不会像1.7那样产生很多碎空间。也是各个区满了之后，会将对象迁移到下一个区。

顺序：

一个对象被创建后，就会出现在Eden区，每过一段时间GC就会过来进行回收，当对象没有引会被回收，而还在被引用的就会移到下一个区。

Eden区-》存活区-》年老代

* Compares ,equals和 =的区别

 == : 如果比较的是基本数据类型，那么比较的是变量的值

== ：如果比较的是引用数据类型，那么比较的是地址值（两个对象是否指向同一块内存）

 equals:如果没重写equals方法比较的是两个对象的地址值。

 如果重写了equals方法后我们往往比较的是对象中的属性的内容.

Compares 多用来做排序，一般是类继承compares接口，然后在List等类sort的时候，调用compares的方法。

* HashMap查询的时间复杂度

最好的情况O（1），最差的情况O（n），这种情况是发生hash碰撞的时候，就要来遍历了。

* Lock 和synchronized 的区别

Lock相当于重入锁，线程可以通过判断锁的状态不等待。

用到 tryLock() 和 unlock（） 两个方法比较多。前者如果判断锁住，是会立刻返回结果的。后者在异常的时候手动释放锁

Synchronized 相当于一直占有锁，其他线程需要等待该线程运行完再执行。加在静态方法上的就是类锁，加在对象上的就是对象锁

* Thread 和 runnable的区别

Thread是类，只可以单继承，重写run的方法就可以了。

Runnable 是接口，可以多实现。两者都是调用start方法来运行run的内容

* Sleep和wait的区别

对于sleep()方法，我们首先要知道该方法是属于Thread类中的。而wait()方法，则是属于Object类中的。

sleep()方法导致了程序暂停执行指定的时间，让出cpu该其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态。

在调用sleep()方法的过程中，线程不会释放对象锁。

而当调用wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备

* Viotile关键字

当我们使用volatile关键字去修饰变量的时候，所以线程都会直接读取该变量并且不缓存它。这就确保了线程读取到的变量是同内存中是一致的。不会从cpu缓存中读取该变量

* CopyOnWriteArrayList和CopyOnWriteArraySet

CopyOnWrite容器即写时复制的容器。通俗的理解是当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对CopyOnWrite容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以CopyOnWrite容器也是一种读写分离的思想，读和写不同的容器。

所以支持并发读和并发写。

Spark：（问的最多）

* Shuffle的过程，为什么要shuffle

因为要宽依赖，一个分区的数据要被多个子分区使用。

过程：

1. 写：先写入缓冲区-》本地文件（默认比例80%）。
2. 数据分区的规则：是通过分区器hash来实现的。默认是取分区的字段hash值，然后取余
3. 拉取：reducer拉取之后在缓存里面合并

* Shuffle ,Join的 实现方式(HashJoin ,Sorted-merge,broadcast join):

Sort merge join：

1. shuffle read 阶段：将两张大表根据join key进行重新分区，两张表数据会分布到整个集群，以便分布式并行处理
2. sort阶段：对单个分区节点的streamIter和buildIter，分别进行排序
3. merge阶段：对排好序的两张分区表数据执行join操作。join操作很简单，分别遍历两个有序序列，碰到相同join key就merge输出，否则取更小一边.
4. 在遍历streamIter时，对于每条记录，都采用顺序查找的方式从buildIter查找对应的记录，由于两个表都是排序的，每次处理完streamIter的一条记录后，对于streamIter的下一条记录，只需从buildIter中上一次查找结束的位置开始查找，所以说每次在buildIter中查找不必重头开始，查找性能还是较优的。

HashJoin 就是对两个分区表不排序。遍历streamIter 表join

Broadcast 就是将小表分发到map上，然后在各个map中进行join了。

* Spark stage和job的划分

Action操作划分job。

宽依赖划分stage

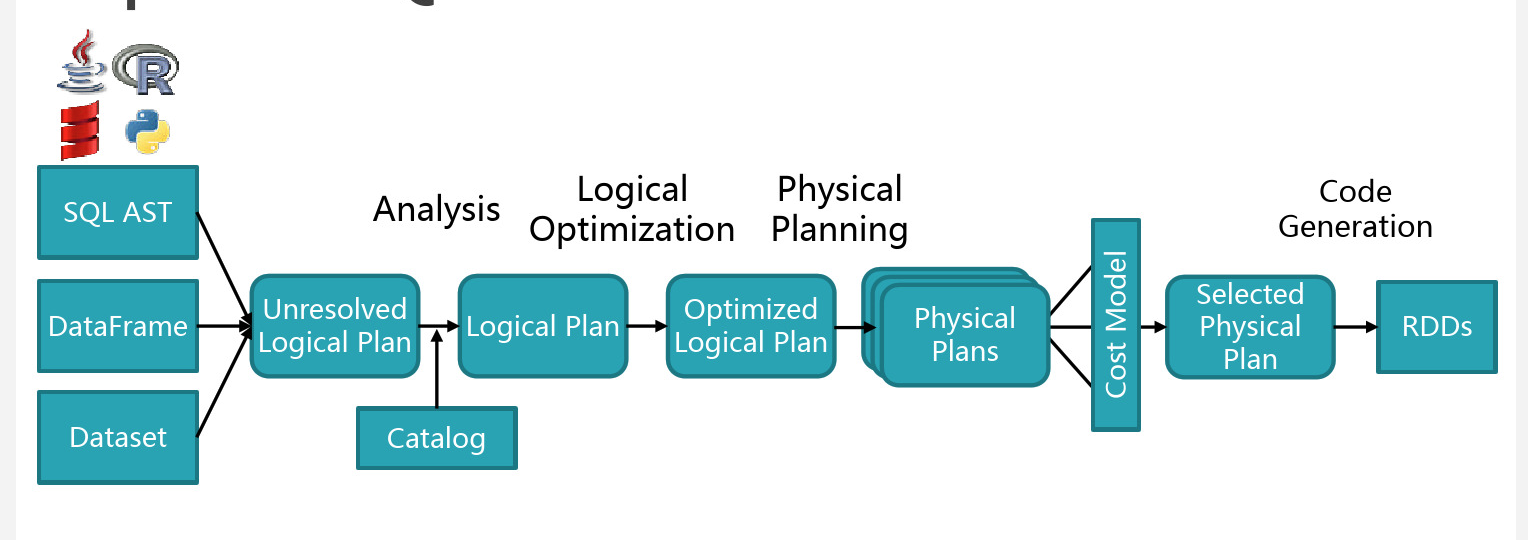
* Kryo序列化

kryo是个高效的java序列化/反序列化库,远远强于java自带的serialize序列化。使用的时候需要对相关的类进行注册

* Spark sql执行的流程

sql语句被解析成抽象语法树（AST），或者我们通过Dataset/DataFrame提供的APIs组织成查询计划，查询计划分为两大类：逻辑计划和物理计划。

逻辑计划-》经过语法分析(Analyzer)、一系列查询优化(Optimizer)-》优化后的逻辑计划-》映射成物理计划-》转换成RDD



* Coalesce 和 repartition 的区别

repartition一定会触发shuffle，而coalesce默认是不触发shuffle的。repartition其实就是调用了coalesce的触发shuffle的

* Spark碰到数据倾斜的办法

数据倾斜解决方案，第一个方案和第二个方案，一起来讲。这两个方案是最直接、最有效、最简单的解决数据倾斜问题的方案。

第一个方案：聚合源数据。

第二个方案：过滤导致倾斜的key。

* MemoryOverHead对应什么

一个Executor对应一个JVM进程。

从Spark的角度看，Executor占用的内存分为两部分：ExecutorMemory和MemoryOverhead

MemoryOverhead是JVM进程中除Java堆以外占用的空间大小，包括方法区（永久代）、Java虚拟机栈、本地方法栈、JVM进程本身所用的内存、直接内存（Direct Memory）等。通过spark.yarn.executor.memoryOverhead设置，单位MB。

ExecutorMemory为JVM进程的Java堆区域。

Hive:

* Clustered by ,sort by ,distribute by 的区别

distribute by 是根据某个id，均匀的分布到reduce任务中。sort by是在reduce出来数据进行排序。cluster by 是distribute by 和 sort by的合并。order by是在一个reduce job中。

* 如何分组取前几条（partition by）

select \* from (select newsid,system\_setting\_area,row\_number() over(partition by system\_setting\_area order by score desc) rn from tmp\_scored\_news) where rn<=100

* Hive 如何自定义map数？（设置读取数据大小）

//设置map和reduce数量

SET mapred.max.split.size = 256000000;

SET mapred.min.split.size = 64000000;

SET mapred.min.split.size.per.node = 64000000;

SET mapred.min.split.size.per.rack = 64000000;

通过这种设置大小的方式来调优

Kafka

* 解释一下ISR

ISR（In-Sync Replicas），副本同步队列。一般由leader维护，是同步的副本数量+1，大大提高可用性。

一个是时间的维度，一个是消息数量的维度来控制同步的情况

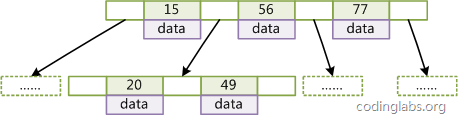
* 解释一下LEO和Highwater Mark

1. HW:消费端能看到的broker上消息的位置，客户端只能消费到HW位置。
2. LEO：消息在log文件中最新的位置。

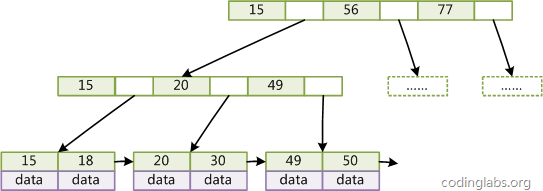
数据库：

* 解释一下B+ tree和B-tree，

B-Tree，节点上会存储数据，在查询单个数据的时候，会比较快。但插入删除新的数据记录会破坏B-Tree的性质，因此在插入删除时，需要对树进行一个分裂、合并、转移等操作以保持B-Tree性质。



B+tree，内节点不存储data，只存储key；叶子节点不存储指针。这也是Mysql通用的方式。



一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。

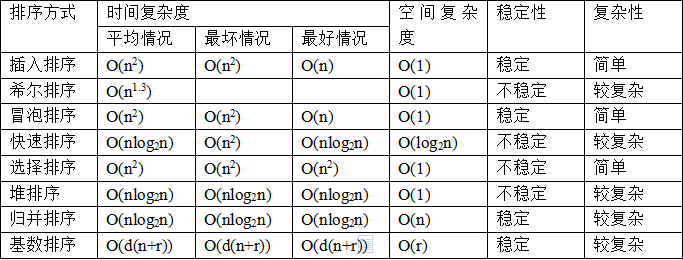
* 什么是数据库的三范式

1. 第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值，就说明该数据库表满足了第一范式。
2. 第二范式在第一范式的基础之上更进一层。第二范式需要确保数据库表中的每一列都和主键相关，而不能跟主键没有关系。也就是说在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。
3. 第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

算法：

* 说一下快排的时间复杂度
* 堆排的时间复杂度

（需要证明）



数据仓库：

* 说一下拉链表

拉链表存在于数据同步的时候，只需要从业务数据库同步更新的数据过来，然后更新数据仓库中的数据，就可以实现两表的同步，而不需要将整表同步过来。

* ER建模和Kimball 建模

ER建模是满足数据三范式建模。

Kimball建模就是维度建模，允许冗余。将数据转化成事实表和维度表。

* 星型模型和雪花模型

星型架构是一种非正规化的结构，多维数据集的每一个维度都直接与事实表相连接，不存在渐变维度，所以数据有一定的冗余，

雪花模型是对星型模型的扩展。它对星型模型的维表进一步层次化，原有的各维表可能被扩展为小的事实表，形成一些局部的 " 层次 " 区域，这些被分解的表都连接到主维度表而不是事实表。

## 6.2 总结

1. 用户行为数据通过http接口上报到web服务器，web服务型将上报的用户行为数据写入本地磁盘，由此实现了业务与数据处理的解耦，之后再由Flume实时监控文件并进行采集。
2. Flume数据采集构架采用双层Flume拓扑架构，第一层实现数据的采集，第二层实现数据的聚合，并通过Flume拓扑结构中的Sinkgroup实现了Load Balance，充分利用了资源。
3. 第一层Flumeagent在每台web服务器上部署一个，当挂掉重启后不能丢失数据，所以我们用了TaildirSource数据源，此数据源会记录每个文件采集到的位置，重启后会从记录的位置采集，但是此数据源有缺陷(当文件重命名后会重复采集)，故我们针对此缺陷进行了源码修改。
4. 第二层聚合Flumeagent需要部署2个及以上，我们的实战项目中有两个聚合agent,其中一个挂掉后，系统可以继续运行，不丢数据，不影响结果。
5. 第二层聚合Flumeagent将数据写入kafka, 当由于某种原因比如网络问题或者kafka停服不能写入时，数据会在第二层agent中的channel中累积，不影响第一层agent的采集，当kafka能够写入时，kafkasink会继续消费第二层angent的channel中的数据。
6. 当第二层聚合Flumeagent全部挂掉时，数据会在第一层的采集agent中的channel中累积，不影响数据采集。
7. Spark Streaming需要7x24小时运行，一旦Driver挂掉后能够自动重启,我们让其在YARN中以Cluster模式运行，此时Driver运行在ApplicationMaster中，当ApplicationMaster挂掉后会重新启动。
8. 为了Driver重启后需要能恢复到之前的状态，所以我们要设置Checkpoint目录，以持久化Kafka offset、未完成的job等到HDFS。
9. 当Spark Streaming的代码修改后启动时，反序列化Checkpoint目录中的数据失败，所以Kafkaoffset会丢失，此时不知道从哪里消费Kafka的数据，所以我们要将Kafka offset保存到ZooKeeper中一份，当Spark Streaming优雅停止后，删除Checkpoint目录然后从ZooKeeper中读取Kafka offset再启动SparkSteaming。