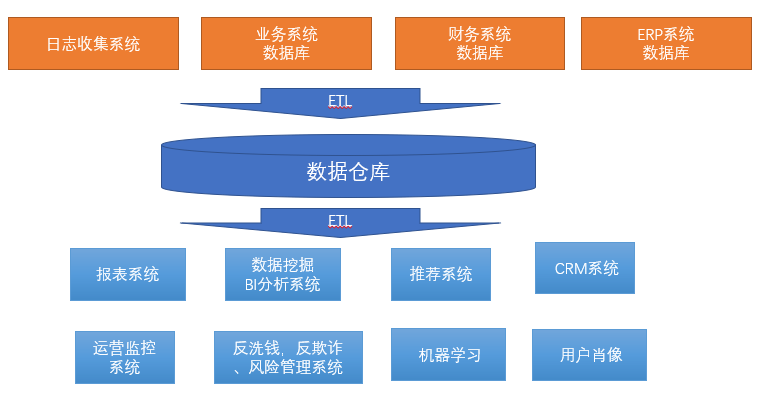
# 数仓离线

学习本项目的目的：为了熟悉公司的大数据业务处理场景，对于一个电商公司来说，其部门的核心可能只是专注于业务的，而其大数据业务只是交给了云厂商，设计该数仓项目为了将日志性为进行采集，然后结合公司的业务表进行相关的进行离线分析，比如分析分析，活跃用户的topN，分析用户的留存率；分析转化率；分析品牌的复购率等。

## 数仓的架构



数仓不是数据最终的目的，而是为了数据的最终目的做好准备。这些准备包扩了对数据的清洗，转义，分类，重组，合并，拆分，统计等。

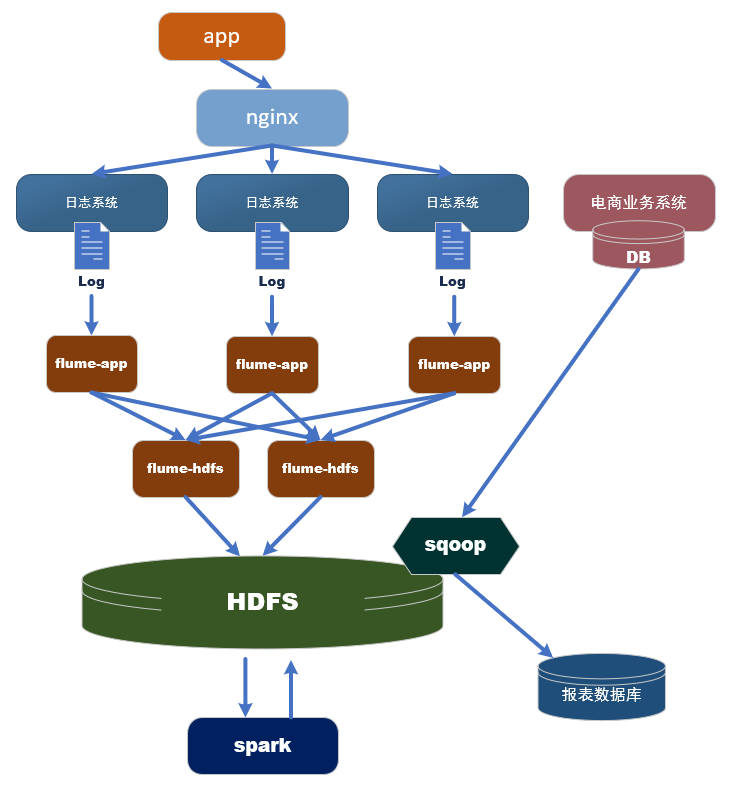
ETL：Extract Transform Load 的缩写，用来描述将数据从的源端，抽取，交互转化，加载到目的端的过程，使用到的工具有kafka，Flume，Sqoop等。

BI分析系统：Business Intelligence 商业智能系统。

CRM系统：Customer Relation Management 用户关系管理系统

可以总结为：数仓的核心是用来管理，我们的数据。

## 离线项目架构



这里设计双层Flume的原因：削峰填谷，用来缓冲；归集，便于管理。

这里的日志采集到HDFS的过程，一般就是叫做ETL；

这里的设计结构是电商业务的服务系统和大数据的数据采集是分开的，因为某些公司可能没有能力搭建大数据框架，只是购买了云厂商的大数据服务，因此两者是分开的。云厂商托管了你司产生的数据，并提供接口，你可以查看的到。

关于日志埋点的两种策略：

1. 周期性发送：把一段时间的日志积攒起来，周期性发送，非实时，对日志系统的压力小
2. 直接发送，时效性好，后台收集的压力大

Springboot：

## 本项目的搭建：

本项目，由于没有实际的生产环境作为依托，所以需要搭建生产数据的模块，比如日志信息的生成，比如每当用户产生一次登录，或者是产生了一次点击行为，都会产生一条日志信息，所以需要搭建**模拟日志的生成模块**，除此之外，模拟日志的生成是在客户端生成的（也就是模拟用户在APP上产生的行为）当用户产生了行为日志之后，发送到日志服务器，由日志服务器进行日志的采集工作，所以需要搭建**日志服务器模块**。

### 日志生成模块imall-mocker（也即生成模拟日志）

日志生成的代码见IDEA：

|  |  |
| --- | --- |
| 用户使用的APP | imall |
| <http://toLinuxLogServer/isea_log> | 日志服务器地址 |
| toLinuxLogServer | Hadoop101 -> 192.168.1.101 |
| 模拟日志的默认发送的端口 | 80端口 |
|  |  |

模拟日志生成的样例：

|  |
| --- |
| *\**  *{"area":"shan1xi","uid":"309","itemid":22,"npgid":44,"evid":"addFavor","os":"ios","pgid":3,"appid":"imall","mid":"mid\_213","type":"event","ts":1550823573057} \* {"area":"tianjin","uid":"485","os":"andriod","ch":"website","appid":"imall","mid":"mid\_116","type":"startup","vs":"1.1.3","ts":1550823573059} \*/* |

### Nginx负载均衡+反向代理模块

|  |
| --- |
| upstream toLinuxLogServer{  server hadoop101:8080 weight=1;  server hadoop102:8080 weight=1;  server hadoop103:8080 weight=1;  }  server {  listen 80;  server\_name toLinuxLogServer;  #对应着用户APP产生的日志的请求地址  #charset koi8-r;  #access\_log logs/host.access.log main;  # 对所有的做反向代理，和负载均衡  location / {  root html;  index index.html index.htm;  # proxy\_pass指令用来指定请求转向代理的后端服务器定义的服务器列表  proxy\_pass http://toLinuxLogServer;  proxy\_connect\_timeout 10;  } |

简单来说，就是Nginx做为负载均衡，将用户App产生的用户日志发送给了Nginx，然后Nginx转发给日志服务器：

### 日志采集模块imall-logger（也即日志服务器）

使用**springboot**（约定大于配置，不需要写那么多的配置文件，而且内嵌了tomcat）来接收模拟生成的日志。在日志服务器和用户APP之间会有一层Nginx服务器，用来做负载均衡，负载均衡监听80 端口，来接收用户APP发送来的日志。然后设置Nginx服务器的上行服务器为toLinuxLogServer,。具体的Nginx服务器的配置信息上面已经写了：

|  |
| --- |
| package com.isea.imall.dw.imall.dw.logger.controller;  import com.alibaba.fastjson.JSON; import com.alibaba.fastjson.JSONObject; import org.slf4j.LoggerFactory; import org.springframework.web.bind.annotation.PostMapping; import org.springframework.web.bind.annotation.RequestParam; import org.springframework.web.bind.annotation.RestController;  @RestController // = Controller+ResponseBody public class LoggerController {  private static final org.slf4j.Logger *logger* = LoggerFactory.*getLogger*(LoggerController.class);   @PostMapping("isea\_log") // 映射请求的URL  // 会认为success 不是一个页面，而仅仅是个结果，就是一个字符串  public String log(@RequestParam("log") String logJson) { // 请求的参数,注入到一个变量中，log必须是请求的时候，=左边的字符串  // 这里是处理具体的业务->使用logger4j来专门的管理日志，将日志输出到目的地。   System.*out*.println(logJson); // 在后端打印出请求传入的内容   JSONObject jsonObject = JSON.*parseObject*(logJson);  jsonObject.put("ts", System.*currentTimeMillis*());   *logger*.info(jsonObject.toJSONString());   return "success..."; // 返回前端，或者是请求者的信息。多数时候这里是返回一个页面给前端。  } } |

日志服务器将用户日志落盘，写成了log日志文件，等待Flume的采集。

服务器返回400 ，表示请求无效。

Nginx：是一个高性能的HTTP服务器和反向代理服务器， 其有三大功能：

1. 反向代理
2. 负载均衡
3. 动静分离

正向代理服务器：使用ssr来访问Google；面向用户的代理。

反向代理，用户不关心哪台服务器提供的服务，是面向服务器的代理。

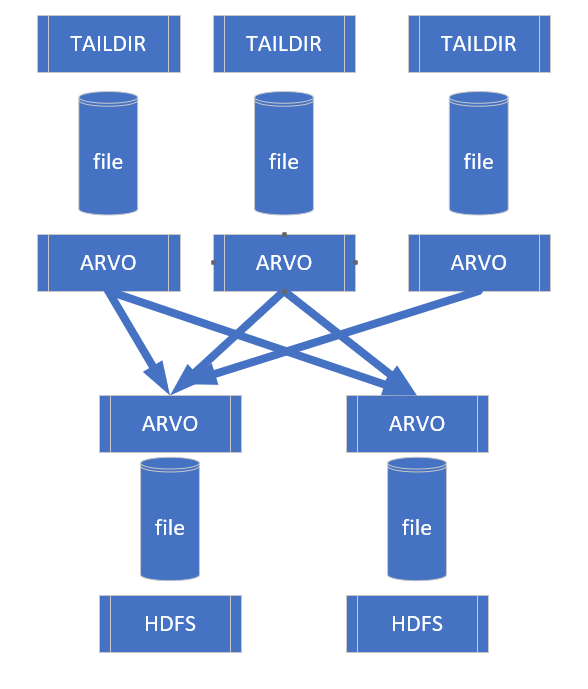
先拦截，在选择。**Flume的数据可能会重复，但是不会丢失，但是kafka可能会丢失数据。**

tairdir 可以记录偏移量：存储形式为json文件，在flume宕机了之后，在flume重启之后，可以接着上次度的文件继续读，而exec会重复读，造成数据重复。

群起日志服务器的脚本：

|  |
| --- |
| #!/bin/bash  app\_name=imall\_one  logger\_name=imall-logger-0.0.1-SNAPSHOT  case $1 in  "start"){  for i in hadoop101 hadoop102 hadoop103  do  echo ----------$i----------  ssh isea@$i "source /etc/profile ; java -jar /applog/imall\_one/imall-logger-0.0.1-SNAPSHOT.jar --server.port=8080 >/dev/null 2>&1 & "  done  /usr/local/nginx/sbin/nginx  };;  "stop"){  /usr/local/nginx/sbin/nginx -s stop  for i in hadoop101 hadoop102 hadoop103  do  echo ----------$i----------  ssh isea@$i "source /etc/profile ; jps -l |grep $logger\_name |awk '{print \$1}'|xargs kill" >/dev/null 2>&1 &  done  };;  esac |

### 双层Flume模块



一共需要启动五个Flume服务，包括三个Flume-app，和两个Flume-HDFS，在Flume-HDFS需要使用拦截器和选择器，**将source中的event拦截，然后按照type字段的startup和event事件，进行选择，发完不同的channel**，**拦截器是自定义的，选择器使用默认的**。最后写到不同的HDFS目录上。

#### Flume-app

|  |
| --- |
| #agent-app  agent-app.sources = sc1  agent-app.channels = ch1  agent-app.sinks = sk1 sk2  #source1  agent-app.sources.sc1.type = TAILDIR  agent-app.sources.sc1.positionFile =/applog/imall\_one/flume\_data/taildir\_position1.json  agent-app.sources.sc1.channels = ch1  agent-app.sources.sc1.filegroups=f1  #source的采集文件的地址  agent-app.sources.sc1.filegroups.f1=/applog/imall\_one/log/app.log  agent-app.sources.sc1.fileHeader = true  #channel1  agent-app.channels.ch1.type = file  #存储一些状态，上下文的信息  agent-app.channels.ch1.checkpointDir =/applog/imall\_one/flume\_data/app\_ch1\_tmp1  #存储真正的数据  agent-app.channels.ch1.dataDirs =/applog/imall\_one/flume\_data/app\_ch1\_tmpdata  agent-app.channels.ch1.maxFileSize = 104857600  #存储10000个event的大小  agent-app.channels.ch1.capacity= 10000  #sink1  agent-app.sinks.sk1.type = avro  agent-app.sinks.sk1.channel = ch1  agent-app.sinks.sk1.batchSize = 10  agent-app.sinks.sk1.hostname = hadoop101  agent-app.sinks.sk1.port = 4444  #sink2  agent-app.sinks.sk2.type = avro  agent-app.sinks.sk2.channel = ch1  agent-app.sinks.sk2.batchSize = 10  agent-app.sinks.sk2.hostname = hadoop102  agent-app.sinks.sk2.port = 4444  #sinkgroups  agent-app.sinkgroups=sg1  agent-app.sinkgroups.sg1.sinks=sk1 sk2  agent-app.sinkgroups.sg1.processor.type=load\_balance  agent-app.sinkgroups.sg1.processor.selector=round\_robin  agent-app.sinkgroups.sg1.processor.backoff=true  agent-app.sinkgroups.sg1.processor.selector.maxTimeOut=10000 |

#### 自定义拦截器：

|  |
| --- |
| package com.isea.imall.dw.fi;  import com.google.gson.Gson; import org.apache.flume.Context; import org.apache.flume.Event; import org.apache.flume.interceptor.Interceptor;  import java.util.HashMap; import java.util.List; import java.util.Map;  public class MyInterceptor implements Interceptor {  public static final String *FI\_HEADER\_TYPE* = "logType";  public static final String *FI\_HEADER\_TYPE\_STARTUP* = "startup";  public static final String *FI\_HEADER\_TYPE\_EVENT* = "event";  Gson gson = null;   @Override  public void initialize() {  gson = new Gson();  }   @Override  public Event intercept(Event event) {  String logString = new String(event.getBody());  HashMap logMap = gson.fromJson(logString, HashMap.class);  String type = (String) logMap.get("type");  Map<String, String> headers = event.getHeaders();  if (*FI\_HEADER\_TYPE\_STARTUP*.equals(type)) {  headers.put(*FI\_HEADER\_TYPE*, *FI\_HEADER\_TYPE\_STARTUP*);  } else {  headers.put(*FI\_HEADER\_TYPE*, *FI\_HEADER\_TYPE\_EVENT*);  }  return event;  }   @Override  public List<Event> intercept(List<Event> list) {  for (Event event : list) {  intercept(event);  }  return list;  }   @Override  public void close() {   }   */\*\*  \* 该静态内部类的作用就是用来创建自定义对象，提供给flume使用，  \*/* public static class Builder implements Interceptor.Builder {   */\*\*  \** ***@return*** *返回创建的自定义的拦截器对象  \*/* @Override  public Interceptor build() {  return new MyInterceptor();  }   @Override  public void configure(Context context) {  // 可以通过context得到flume.conf 中设置的参数，传递给Interceptor  }  } } |

然后将以上的代码打包，然后放置到flume的lib包下。

#### Flume-hdfs

|  |
| --- |
| #agent-hdfs  agent-hdfs.sources = sc1  agent-hdfs.channels = ch1 ch2  agent-hdfs.sinks = sk1 sk2  #source1  agent-hdfs.sources.sc1.type = avro  agent-hdfs.sources.sc1.channels = ch1 ch2  agent-hdfs.sources.sc1.bind = 0.0.0.0  agent-hdfs.sources.sc1.port = 4444  #interceptor  agent-hdfs.sources.sc1.interceptors=i1  agent-hdfs.sources.sc1.interceptors.i1.type=com.isea.imall.dw.fi.MyInterceptor$Builder  #selector  agent-hdfs.sources.sc1.selector.type = multiplexing  agent-hdfs.sources.sc1.selector.header= logType  agent-hdfs.sources.sc1.selector.mapping.startup = ch1  agent-hdfs.sources.sc1.selector.mapping.event = ch2  agent-hdfs.sources.sc1.selector.default = ch2  #channel1  agent-hdfs.channels.ch1.type = file  agent-hdfs.channels.ch1.checkpointDir =/applog/imall\_one/flume\_data/hdfs\_ch1\_tmp  agent-hdfs.channels.ch1.dataDirs =/applog/imall\_one/flume\_data/hdfs\_ch1\_tmpdata  agent-hdfs.channels.ch1.maxFileSize = 104857600  agent-hdfs.channels.ch1.capacity= 1000000  #channel2  agent-hdfs.channels.ch2.type = file  agent-hdfs.channels.ch2.checkpointDir =/applog/imall\_one/flume\_data/hdfs\_ch2\_tmp  agent-hdfs.channels.ch2.dataDirs =/applog/imall\_one/flume\_data/hdfs\_ch2\_tmpdata  agent-hdfs.channels.ch2.maxFileSize = 104857600  agent-hdfs.channels.ch2.capacity= 1000000  #sink1  agent-hdfs.sinks.sk1.channel = ch1  #agent-hdfs.sinks.sk1.type =logger  agent-hdfs.sinks.sk1.type = hdfs  agent-hdfs.sinks.sk1.hdfs.path = hdfs://hadoop101:9000/origin\_data/imall\_one/log/startup\_log/%Y-%m-%d  #agent-hdfs.sinks.sk1.hdfs.fileType = DataStream  agent-hdfs.sinks.sk1.hdfs.fileType = CompressedStream  agent-hdfs.sinks.sk1.hdfs.fileSuffix=.gz  agent-hdfs.sinks.sk1.hdfs.codeC=gzip  agent-hdfs.sinks.sk1.hdfs.writeFormat = TEXT  agent-hdfs.sinks.sk1.hdfs.rollInteval = 30  agent-hdfs.sinks.sk1.hdfs.useLocalTimeStamp = true  agent-hdfs.sinks.sk1.hdfs.rollCount=0  agent-hdfs.sinks.sk1.hdfs.rollSize=0  #sink2  agent-hdfs.sinks.sk2.channel = ch2  agent-hdfs.sinks.sk2.type = hdfs  agent-hdfs.sinks.sk2.hdfs.path = hdfs://hadoop101:9000/origin\_data/imall\_one/log/event\_log/%Y-%m-%d  #agent-hdfs.sinks.sk2.hdfs.fileType = DataStream  agent-hdfs.sinks.sk2.hdfs.fileType = CompressedStream  agent-hdfs.sinks.sk2.hdfs.fileSuffix=.gz  agent-hdfs.sinks.sk2.hdfs.codeC=gzip  agent-hdfs.sinks.sk2.hdfs.writeFormat = TEXT  agent-hdfs.sinks.sk2.hdfs.rollInteval = 30  agent-hdfs.sinks.sk2.hdfs.roundUnit=second  agent-hdfs.sinks.sk2.hdfs.useLocalTimeStamp = true  agent-hdfs.sinks.sk2.hdfs.rollCount=0  agent-hdfs.sinks.sk2.hdfs.rollSize=0 |

#### 群起Flume脚本：

|  |
| --- |
| #!/bin/bash  FLUME\_HOME=/opt/module/flume  PROJECT=imall\_one  JOB\_HOME=~/$PROJECT/flume-job/  case $1 in  "start") {  for i in hadoop101 hadoop102  do  echo "=====启动 $i agent-hdfs flume====="  ssh isea@$i "source /etc/profile; $FLUME\_HOME/bin/flume-ng agent -c $FLUME\_HOME/conf -f $JOB\_HOME/flume-hdfs.conf -n agent-hdfs -Dflume.root.logger=info,console >$FLUME\_HOME/logs/flume-hdfs.log 2>&1 &"  done  for i in hadoop101 hadoop102 hadoop103  do  echo "=====启动 $i agent-app flume====="  ssh isea@$i " source /etc/profile; $FLUME\_HOME/bin/flume-ng agent -c $FLUME\_HOME/conf -f $JOB\_HOME/flume-app.conf -n agent-app -Dflume.root.logger=info >/dev/null 2>&1 &"  done  };;  "stop") {  for i in hadoop101 hadoop102 hadoop103  do  echo "====关闭 $i agent-app flume ====="  ssh isea@$i "ps -ef|grep agent-app |grep -v grep|awk '{print \$2}'|xargs kill" >/dev/null 2>&1  done  for i in hadoop101 hadoop102  do  echo "====关闭 $i agent-hdfs flume ====="  ssh isea@$i "ps -ef|grep agent-hdfs |grep -v grep|awk '{print \$2}'|xargs kill" >/dev/null 2>&1  done  };;  esac |

## 数仓分层概念：

当数据经过了ETL的Flume，Sqoop等框架之后，落盘到HDFS文件系统上，需要进一步将数据导入到Hive表中，在导入到Hive表的过程中，我们将数据分为四层，来方便业务逻辑的处理。

#### ODS

（Operational Data Store）原始数据层，一般使用压缩率较大的压缩方式。Gzip。

#### DWD

Data Warehouse Detail明细数据层，基本上和原始数据没有什么变化，区别是这一层要清除脏数据，空值，超过极限范围的数据，还有一个就是降维，变为星型表。或者叫DWI。

在这一层，行式存储变为列式存储，压缩的算法改变为较快速度的压缩算法。压缩：gzip->lzo

Lzo->snappy, 高压缩比->第压缩比

#### DWS

Data Warehouse Service服务数据层，该层是承上启下的作用，可认为是数据的半成品。（这一层看设计）这一层会根据业务，进行一定程度的聚合，用户，商品，商家，形成一个宽表。

#### ADS

Application Data Store数据应用层，可以认为是报表的了，本层看SQL的能力，本层还可以叫DM层（数据集市，广义的数据集市是只要加工了，都是数据集市）APP层，DAL层（data access layer）

分层的好处是：清晰便于管理。

在数仓分层的过程中，我们需要做的事情就是将HDFS上的数据导入到Hive的表中，这里一般企业使用的都是外部表，来location到数据在HDFS的位置。

如果是比较冷的数据，使用高压缩比的压缩算法的压缩方式，比如Gzip；如果是需要经常运算的热数据，建议使用处理速度比较快的snappy。

Ods可以使用Gzip压缩；dwd，dws层可以使用snappy。

行式存储可以选择三种方式：rcfile，orc，parquet，如果是hive引擎，orc是最佳的选择；如果是spark选择parquet是最佳选择。

使用列式存储的好处是，过滤查询速度比行式存储速度快；配合压缩算法压缩比会更高。省空间和时间。

## 需求1：活跃用户

1. 每日活跃用户数
2. 每周活跃用户数
3. 每月活跃用户数

以上以Mid为准。也即存在着多个注册用户使用同一台机器来登录和操作的情况，此时，将机器作为用户的活跃数量。

实际工作的过程中都是建立外部表，但是这里为了试验的方便，我们这里建立内部表，便于删除和调试。建表的时候，使用partition by的好处是在做数据查询的时候，只是查询一个文件，提高查询的效率。

Hive中的库名：imall\_one，接下来的所有的表都将放置在该数据库里面：

### ODS层

该层几乎就是原始数据，储存产生的日志的进入日志服务器时候的数据信息。

由于HDFS中的数据是Json格式的，所以建表的时候，需要指定支持的JSON格式，在Hive的lib目录或者是spark的jars目录下引入json-serde-1.3.8-jar-with-dependencies.jar包。

#### 创建ods\_startup\_log表和ods\_enent\_log表

|  |
| --- |
| drop table if exists ods\_startup\_log;  create table **ods\_startup\_log** (  `mid` string COMMENT '设备唯一标识',  `uid` string COMMENT '用户标识',  `os` string COMMENT '操作系统',  `ch` string COMMENT '渠道',  `appid` string COMMENT '应用id',  `vs` string COMMENT '版本号',  `ts` bigint COMMENT '启动时间戳',  `area` string COMMENT '地区'  ) COMMENT '启动日志'  PARTITIONED BY ( `**dt**` string)  row format serde 'org.openx.data.jsonserde.JsonSerDe'  location '/warehouse/imall\_one/ods/ods\_startup\_log/';  ods\_startup\_log九个字段，其中有一个伪列；该表存放的位置在location位置  drop table if exists ods\_event\_log ;  CREATE TABLE **ods\_event\_log** (  `mid` string COMMENT '设备唯一 表示',  `uid` string COMMENT '用户标识',  `os` string COMMENT '操作系统',  `appid` string COMMENT '应用id',  `area` string COMMENT '地区' ,  `evid` string COMMENT '事件id',  `pgid` string COMMENT '当前页',  `npgid` string COMMENT '跳转页',  `itemid` string COMMENT '商品id',  `ts` bigint COMMENT '时间戳'  )COMMENT '事件日志'  PARTITIONED BY ( `**dt**` string)  row format serde 'org.openx.data.jsonserde.JsonSerDe'  location '/warehouse/imall\_one/ods/ods\_event\_log/';  ods\_event\_log该表有11个字段。 |

#### 将当日的原始数据导入到ods层

每日的数据会在第二天的凌晨1点执行导入。凌晨1点导入的目的是确保当日的数据全部都被flume采集到了HDFS中，**今天的数据就是等到今天过完才导入**。

ods\_load\_log.sh

|  |
| --- |
| #!/bin/bash  #将存储在HDFS文件系统上的日志信息导入到ODS层的Hive表中  APP=imall\_one  if [ -n $1 ] ;then  log\_date=$1  else  log\_date=`date -d "-1 day" +%F`  fi  echo "===日志日期为 $log\_date==="  hive=/opt/module/hive/bin/hive  $hive -e "load data inpath '/origin\_data/$APP/log/startup\_log/$log\_date' into table "$APP".ods\_startup\_log partition(dt='$log\_date')"  $hive -e "load data inpath '/origin\_data/$APP/log/event\_log/$log\_date' into table "$APP".ods\_event\_log partition(dt='$log\_date')" |

### DWD层

该层的作用主要是对ods的数据进行一些清洗，比如去null等或者是对表的降维。

#### 建立dwd\_startup\_log，dwd\_event\_log表

|  |
| --- |
| drop table if exists dwd\_startup\_log;  create table **dwd\_startup\_log**(  `mid` string COMMENT '设备唯一标识',  `uid` string COMMENT '用户标识',  `os` string COMMENT '操作系统',  `ch` string COMMENT '渠道',  `appid` string COMMENT '应用id',  `vs` string COMMENT '版本号',  `ts` bigint COMMENT '启动时间戳',  `area` string COMMENT '地区'  ) COMMENT '启动明细表'  PARTITIONED BY ( `**dt**` string)  stored as **parquet**  location '/warehouse/imall\_one/dwd/dwd\_startup\_log/'  tblproperties ("parquet.compression"="snappy");    该表有九个字段，其中一个伪字段。此时有行存储变为列存储，压缩方式变为snappy。  drop table if exists dwd\_event\_log;  CREATE TABLE **dwd\_event\_log** (  `mid` string COMMENT '设备唯一 表示',  `uid` string COMMENT '用户标识',  `os` string COMMENT '操作系统',  `appId` string COMMENT '应用id',  `area` string COMMENT '地区' ,  `evid` string COMMENT '事件id',  `pgid` string COMMENT '当前页',  `npgid` string COMMENT '跳转页',  `itemid` string COMMENT '商品id',  `ts` bigint COMMENT '时间戳'  )COMMENT '事件明细表'  PARTITIONED BY ( `**dt**` string)  stored as parquet  location '/warehouse/imall\_one/dwd/dwd\_event\_log/'  tblproperties ("parquet.compression"="snappy"); |

#### 将ods层的数据清洗之后导入到dwd层

dwd\_log.sh

|  |  |
| --- | --- |
| #!/bin/bash  #将ods层的数据，进行一定程度的清洗，然后存储到dwd层  APP=imall\_one  do\_date=$1  if [ ! -n "$do\_date" ] ;then  do\_date=`date -d "-1 day" +%F`  fi  sql=" use $APP;  **set hive.exec.dynamic.partition.mode=nonstrict;**  insert overwrite table dwd\_startup\_log partition(dt)  select  \*  from ods\_startup\_log  where dt='$do\_date' and mid is not null;  insert overwrite table dwd\_event\_log partition(dt)  select  \*  from ods\_event\_log  where dt='$do\_date' and mid is not null;"  /opt/module/spark/bin/spark-sql -e "$sql"   |  | | --- | | 案例：  set hive.exec.dynamic.partition.mode=nonstrict;    insert overwrite table dwd\_startup\_log partition(dt)  select  \*  from ods\_startup\_log  where dt='2019-02-25' and mid is not null;      insert overwrite table dwd\_event\_log partition(dt)  select  \*  from ods\_event\_log  where dt='2019-02-25' and mid is not null; | |

关于**非严格模式**：在关系型数据库中，对分区表insert数据的时候，数据库会根据分区字段的值，将数据插入到对应的分区中，Hive中也提供了相似的机制，即动态分区。但是需要设置：

set hive.exec.dynamic.partition.mode=true（默认为true，开启）非严格模式表示允许所有分区字段都可以使用动态分区。默认是严格模式，strict，表示指定至少一个分区为静态分区。

### DWS层

该层是最重要的层，是明细层，对于本项目来说，用户的每一个登录行为或者点击行为都在这里可以找到对应的明细信息。

不同的用户，在不同的时间，在不同的地点，执行了不同的点击操作，都会被记录下来。但是我们这里统计的时候：我们**统计当日，当周，当月活动的每个设备的明细**。

#### 每日活跃明细，dws\_uv\_detail\_day表

既然是活跃的用户明细，因此只要是登录了之后的，就可以算作是活跃的用户，所以统计的是startup\_log日志中的数据

##### 建表：

一个APP，每天登录的用户都有很多，所以很这里按日期进行建立分区，每天登录的用户都会作为HDFS文件系统上的独立的文件存在。

|  |
| --- |
| drop table if exists dws\_uv\_detail\_day;  create table **dws\_uv\_detail\_day**(  `mid` string COMMENT '设备唯一标识',  `uid` string COMMENT '用户标识',  `vs` string COMMENT '版本号',  `os` string COMMENT '操作系统',  `ch` string COMMENT '渠道',  `appid` string COMMENT '应用id',  `area` string COMMENT '地区'  ) COMMENT '活跃用户按天明细'  PARTITIONED BY ( `**dt**` string)  stored as parquet  location '/warehouse/imall\_one/dws/dws\_uv\_detail\_day/'  tblproperties ("parquet.compression"="snappy"); |

##### 导入数据

每日的凌晨一点执行数据导入到操作，当 group by 按照mid,dt,appid 为key来进行聚合的时候，多个人用同一个机器，同一个日期，登录了不同的appid，都会聚合到同一个组来，然后使用collect\_set函数，只取得该组中的一个。

另外一种理解：

一台机器的某个APP在一天里只有两种状态，活跃或者是不活跃。不活跃，不会出现在日志里，在活跃的情况下，由于可能存在着多个用户在同一天用用一个机器登录同一个APP的情况，也就会出现多条数据，这个时候我们只是取一条。

|  |
| --- |
| set hive.exec.dynamic.partition.mode=nonstrict;  insert overwrite table dws\_uv\_detail\_day partition(**dt='2019-02-25')**  select  mid,  **collect**\_set(uid)[0] uid,  collect\_set(vs)[0] vs,  collect\_set(os)[0] os,  collect\_set(ch)[0] ch,  appid ,  collect\_set(area)[0] area  from dwd\_startup\_log  where dt='2019-02-25'  **group by mid,dt,appid;** |

#### 每周活跃明细dws\_uv\_detail\_wk表

在一个周里，一台机器只有两种状态，活跃或者不活跃，不活跃的不会产生log日志，如果活跃了可能会在这个周每天都活跃，在上面dws\_uv\_detail\_day的基础上，每天只有一条数据，那么每天都活跃就会有多条数据，而我们只是需要一条就OK了。

##### 建表

|  |
| --- |
| drop table if exists dws\_uv\_detail\_wk;  create table **dws\_uv\_detail\_wk**(  `mid` string COMMENT '设备唯一标识',  `uid` string COMMENT '用户标识',  `vs` string COMMENT '版本号',  `os` string COMMENT '操作系统',  `ch` string COMMENT '渠道',  `appid` string COMMENT '应用id',  `area` string COMMENT '地区' ,  `**monday\_date**` string COMMENT '周一日期',  `**sunday\_date**` string COMMENT '周日日期'  ) COMMENT '活跃用户按周明细'  PARTITIONED BY ( `**wk\_dt**` string)  stored as parquet  location '/warehouse/imall\_one/dws/dws\_uv\_detail\_wk/'  tblproperties ("parquet.compression"="snappy"); |

##### 导入数据：

在一个周中的数据，我们只需要采集一条就可以了，所以先**用where将一个周的数据过滤出来，然后在获取一条**。

|  |
| --- |
| set hive.exec.dynamic.partition.mode=nonstrict;  insert overwrite table dws\_uv\_detail\_wk partition( wk\_dt)  select  mid,  collect\_set(uid)[0] uid,  collect\_set(vs)[0] vs,  collect\_set(os)[0] os,  collect\_set(ch)[0] ch,  appid ,  collect\_set(area)[0] area,  date\_add(next\_day('2019-02-25','MO'),-7) monday\_date,  date\_add(next\_day('2019-02-25','MO'),-1) sunday\_date,  concat(date\_add(next\_day('2019-02-25','MO'),-7),'\_' ,date\_add(next\_day('2019-02-25','MO'),-1)) **wk\_dt**  from dws\_uv\_detail\_day  **where dt>=date\_add(next\_day('2019-02-25','MO'),-7) and dt<=date\_add(next\_day('2019-02-25','MO'),-1)**  group by **mid,appid**; |

#### 每月活跃明细dws\_uv\_detail\_mn表

##### 建表

|  |
| --- |
| drop table if exists dws\_uv\_detail\_mn;  create external table dws\_uv\_detail\_mn(  `mid` string COMMENT '设备唯一标识',  `uid` string COMMENT '用户标识',  `vs` string COMMENT '版本号',  `os` string COMMENT '操作系统',  `ch` string COMMENT '渠道',  `appid` string COMMENT '应用id',  `area` string COMMENT '地区'  ) COMMENT '活跃用户按月明细'  PARTITIONED BY ( `mn` string)  stored as parquet  location '/warehouse/imall\_one/dws/dws\_uv\_detail\_mn/'  tblproperties ("parquet.compression"="snappy"); |

##### 导入数据

一个用户在一个月有两种状态，活跃或者是不活跃，活跃的话，可能每天都活跃，如此一来就会有多条数据，但是我们只是需要一条，所以使用where过滤，然后只取一条。

|  |
| --- |
| set hive.exec.dynamic.partition.mode=nonstrict;  insert overwrite table dws\_uv\_detail\_mn partition( mn)  select  mid,  collect\_set(uid)[0] uid,  collect\_set(vs)[0] vs,  collect\_set(os)[0] os,  collect\_set(ch)[0] ch,  appid ,  collect\_set(area)[0] area,  date\_format('2019-02-25','yyyy-MM')  from dws\_uv\_detail\_day  where date\_format(dt,'yyyy-MM') = date\_format('2019-02-25','yyyy-MM')  group by mid,appid; |

#### dws\_uv.sh脚本：

|  |
| --- |
| #!/bin/bash  #将DWD层Hive表中的数据做进一步的筛选，在将数据存储到DWS层的Hive表中  APP=imall\_one  do\_date=$1  if [ ! -n "$do\_date" ] ;then  do\_date=`date -d "-1 day" +%F`  fi  sql=" use $APP;  set hive.exec.dynamic.partition.mode=nonstrict;  insert overwrite table **dws\_uv\_detail\_day** partition(dt='$do\_date')  select  mid,  collect\_set(uid)[0] uid,  collect\_set(vs)[0] vs,  collect\_set(os)[0] os,  collect\_set(ch)[0] ch,  appid ,  collect\_set(area)[0] area  from dwd\_startup\_log  where dt='$do\_date'  group by **mid,dt,appid**;    insert overwrite table **dws\_uv\_detail\_wk** partition( wk\_dt)  select  mid,  collect\_set(uid)[0] uid,  collect\_set(vs)[0] vs,  collect\_set(os)[0] os,  collect\_set(ch)[0] ch,  appid ,  collect\_set(area)[0] area,  date\_add(next\_day('$do\_date','MO'),-7) monday\_date,  date\_add(next\_day('$do\_date','MO'),-1) sunday\_date,  concat(date\_add(next\_day('$do\_date','MO'),-7),'\_' ,date\_add(next\_day('$do\_date','MO'),-1)) wk\_dt  from **dws\_uv\_detail\_day**  w**here dt>=date\_add(next\_day('$do\_date','MO'),-7) and dt<=date\_add(next\_day('$do\_date','MO'),-1)**  group by **mid,appid**;    insert overwrite table **dws\_uv\_detail\_mn** partition( mn)  select  mid,  collect\_set(uid)[0] uid,  collect\_set(vs)[0] vs,  collect\_set(os)[0] os,  collect\_set(ch)[0] ch,  appid ,  collect\_set(area)[0] area,  date\_format('$do\_date','yyyy-MM')  from dws\_uv\_detail\_day  where date\_format(dt,'yyyy-MM') =date\_format('$do\_date','yyyy-MM')  group by **mid,appid;**  "  /opt/module/spark/bin/spark-sql -e "$sql" |

### AWS层

这一层统计当日，当周，当月的活跃设备数：此时的数据不需要再压缩，所以建表的时候不用在压缩，但是我们还是压缩了。

#### 建立ads\_uv\_count表

|  |
| --- |
| drop table if exists ads\_uv\_count;  create table **ads\_uv\_count**(  `dt` string COMMENT '统计日期',  `day\_count` long COMMENT '当日用户数量',  `wk\_count` long COMMENT '当周用户数量',  `mn\_count` long COMMENT '当月用户数量',  `is\_weekend` string COMMENT 'Y,N是否是周末,用于得到本周最终结果',  `is\_monthend` string COMMENT 'Y,N是否是月末,用于得到本月最终结果'  ) COMMENT '每日活跃用户数量'  stored as **parquet**  location '/warehouse/imall\_one/ads/ads\_uv\_count\_day/'  tblproperties ("parquet.compression"="snappy"); |

#### 导入数据

从DWS表中获取数据。

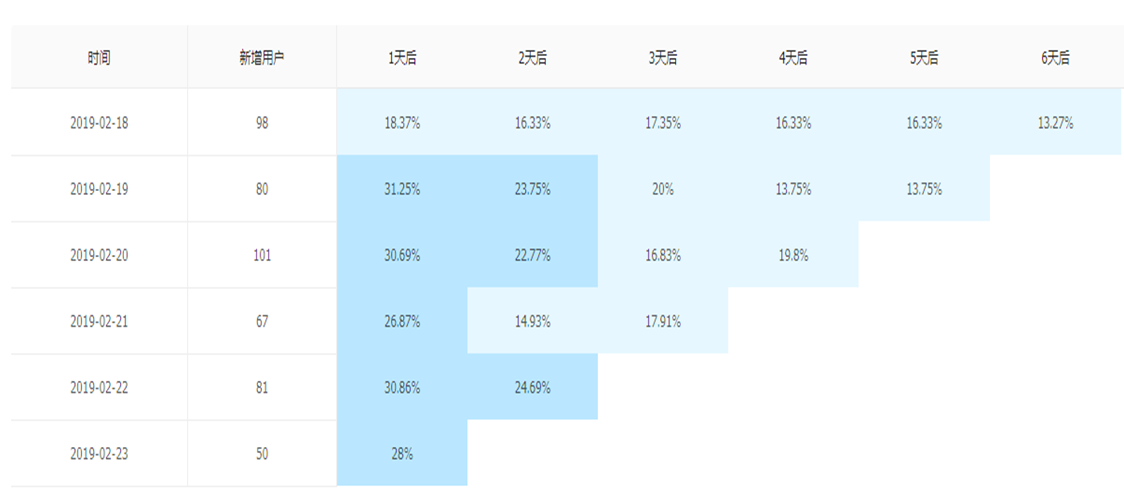
|  |
| --- |
| insert into ads\_uv\_count  select dayct.dt,dayct.ct,wkct.ct,mnct.ct,  if(date\_add(next\_day('2019-02-25','MO'),-1)=='2019-02-25','Y','N') is\_monthend,  if(last\_day('2019-02-25')=='2019-02-25','Y','N') is\_monthend  from  (  select '2019-02-25' dt,count(\*) ct  from dws\_uv\_detail\_day  where dt='2019-02-25'  )dayct  join  (  select '2019-02-25' dt, count(\*) ct  from dws\_uv\_detail\_wk  where wk\_dt=concat(date\_add(next\_day('2019-02-25','MO'),-7), '\_',date\_add(next\_day('2019-02-25','MO'),-1))  )wkct on wkct.dt=dayct.dt  join  (  select '2019-02-25' dt,count(\*) ct  from dws\_uv\_detail\_mn  where mn = date\_format('2019-02-25','yyyy-MM')  )mnct on mnct.dt=dayct.dt; (  select '2019-02-25' dt, count(\*) ct  from dws\_uv\_detail\_wk  where wk\_dt=concat(date\_add(next\_day('2019-02-25','MO'),-7), '\_',date\_add(next\_day('2019-02-25','MO'),-1))  )wkct on wkct.dt=dayct.dt  join  (  select '2019-02-25' dt,count(\*) ct  from dws\_uv\_detail\_mn  where mn = date\_format('2019-02-25','yyyy-MM')  )mnct on mnct.dt=dayct.dt; |

#### ads\_uv.sh脚本

|  |
| --- |
| #!/bin/bash  #将DWS层中Hive的数据最最后一步筛选，存储到ADS层的Hive表中。  APP=imall\_one  do\_date=$1  if [ ! -n "$do\_date" ] ;then  do\_date=`date -d "-1 day" +%F`  fi  sql=" use $APP;  insert into ads\_uv\_count  select dayct.dt,dayct.ct,wkct.ct,mnct.ct,  if(date\_add(next\_day('${do\_date}','MO'),-1)=='${do\_date}','Y','N') is\_monthend,  if(last\_day('${do\_date}')=='${do\_date}','Y','N') is\_monthend  from  (  select '${do\_date}' dt,count(\*) ct  from dws\_uv\_detail\_day  where dt='${do\_date}'  )dayct  join  (  select '${do\_date}' dt, count(\*) ct  from dws\_uv\_detail\_wk  where wk\_dt=concat(date\_add(next\_day('${do\_date}','MO'),-7), '\_',date\_add(next\_day('${do\_date}','MO'),-1))  )wkct on wkct.dt=dayct.dt  join  (  select '${do\_date}' dt,count(\*) ct  from dws\_uv\_detail\_mn  where mn = date\_format('${do\_date}','yyyy-MM')  )mnct on mnct.dt=dayct.dt;  "  /opt/module/spark/bin/spark-sql -e "$sql" |

## 需求2：新增用户留存率

新增用户留存率。



### 思路：

1. 记录当日的新增机器，（必须是明细）
2. 得到当日的登陆过机器（也即日活用户），要明细\*（在日活跃用户中已经得到）
3. 当日活跃用户与某日（前1,2,3，，，）的新增用户的**交集**，存储为明细=>留存用户
4. 留存用户和 某日（前1,2,3，，）的新增用户的比值，就是留存率。

这里涉及到的都是新增的问题，只是涉及到了用户的操作行为，所以还是针对startup\_log日志中数据。

Hive表分区与否，要看数据量（如果数据量非常的大，就需要分区），还要看查询的范围（经常性的要框定分区的范围需要分区）。

### DWS层

#### 新增设备明细表

什么是新增用户，就是该用户第一次登陆，我们使用一张表来存储新增用户，以后每天的新增用户都插入到这张表中。

##### 建表

**创建时间**的字段，表示这是什么时间新增的用户。

|  |
| --- |
| drop table if exists `dws\_new\_mid\_day`;  create table `**dws\_new\_mid\_day**`  (  `**mid**` string comment '设备id',  appid string comment 'appid',  uid string comment '用户id',  vs string comment '版本',  os string comment '系统',  ch string comment '渠道',  area string comment '地区',  `create\_date` string comment '创建时间'  ) COMMENT '每日新增设备信息'  stored as parquet  location '/warehouse/imall\_one/dws/dws\_new\_mid\_day/'  tblproperties ("parquet.compression"="snappy"); |

##### 每日新增用户怎么求？

使用每日新增设备表和日活表取差集（出现在日活明细表中，但是没有出现在每日新增设备表中的机器）这里可以使用日活表left join新增设备表使用mid关联，将为每日新增的mid 为null的选出来。这里利用了left join的特点。

##### 插入数据

|  |
| --- |
| insert into table dws\_new\_mid\_day  select  uv.mid,  uv.appid,  uv.uid,  uv.vs,  uv.os,  uv.ch,  uv.area,  '2019-02-26'  from dws\_uv\_detail\_day uv  left join dws\_new\_mid\_day nm  on uv.mid=nm.mid  where **nm.mid is null;** |

#### 用户留存表

每一条数据记录create\_date天新增的机器，在dt天还在活跃的机器。

##### 建表

|  |
| --- |
| drop table if exists `dws\_user\_retention\_day`;  create table `**dws\_user\_retention\_day**`  (  `mid` string comment '设备id',  uid string comment '用户id',  vs string comment '版本',  os string comment '系统',  ch string comment '渠道',  appid string comment 'appid',  area string comment '地区',  `**create\_date**` string comment '设备新增时间',  `**retention\_day**` int comment '截止当前日期留存天数'  ) COMMENT '每日用户留存情况'  PARTITIONED BY ( `**dt**` string)  stored as parquet  location '/warehouse/imall\_one/dws/dws\_user\_retention\_day/'  tblproperties ("parquet.compression"="snappy"); |

这里的设备新增时间+截止当前日期留存的时间=dt。

##### 留存用户怎么求？

当日活跃用户与某日（前1,2,3，，，）的新增用户的**交集**就是用户留存表。翻译成人话就是，每日活跃表（dws\_uv\_detail\_day）中的dt字段 -x =每日新增设备表（dws\_new\_mid\_day）中create\_date 字段，即可得到x天的留存明细。

##### 插入数据：

|  |
| --- |
| insert overwrite table dws\_user\_retention\_day partition(dt='2019-02-27')  select  uv.mid,  uv.uid,  uv.vs,  uv.os,  uv.ch,  uv.appid,  uv.area,  date\_add('2019-02-27',-1) new\_create\_date,  1 retention\_days  from dws\_new\_mid\_day nm join dws\_uv\_detail\_day uv on nm.mid=uv.mid  where nm.create\_date=date\_add('2019-02-27',-1) and uv.dt='2019-02-27'  union all  select  uv.mid,  uv.uid,  uv.vs,  uv.os,  uv.ch,  uv.appid,  uv.area,  date\_add('2019-02-27',-1) new\_create\_date,  2 retention\_days  from dws\_new\_mid\_day nm join dws\_uv\_detail\_day uv on nm.mid=uv.mid  where nm.create\_date=date\_add('2019-02-27',-2) and uv.dt='2019-02-27'  union all  select  uv.mid,  uv.uid,  uv.vs,  uv.os,  uv.ch,  uv.appid,  uv.area,  date\_add('2019-02-27',-1) new\_create\_date,  3 retention\_days  from dws\_new\_mid\_day nm join dws\_uv\_detail\_day uv on nm.mid=uv.mid  where nm.create\_date=date\_add('2019-02-27',-3) and uv.dt='2019-02-27'  union all  select  uv.mid,  uv.uid,  uv.vs,  uv.os,  uv.ch,  uv.appid,  uv.area,  date\_add('2019-02-27',-1) new\_create\_date,  4 retention\_days  from dws\_new\_mid\_day nm join dws\_uv\_detail\_day uv on nm.mid=uv.mid  where nm.create\_date=date\_add('2019-02-27',-4) and uv.dt='2019-02-27'  union all  select  uv.mid,  uv.uid,  uv.vs,  uv.os,  uv.ch,  uv.appid,  uv.area,  date\_add('2019-02-27',-1) new\_create\_date,  5 retention\_days  from dws\_new\_mid\_day nm join dws\_uv\_detail\_day uv on nm.mid=uv.mid  where nm.create\_date=date\_add('2019-02-27',-5) and uv.dt='2019-02-27'  union all  select  uv.mid,  uv.uid,  uv.vs,  uv.os,  uv.ch,  uv.appid,  uv.area,  date\_add('2019-02-27',-1) new\_create\_date,  6 retention\_days  from dws\_new\_mid\_day nm join dws\_uv\_detail\_day uv on nm.mid=uv.mid  where nm.create\_date=date\_add('2019-02-27',-6) and uv.dt='2019-02-27'; |

#### DWS层脚本

|  |
| --- |
| #!/bin/bash  # 向新增用户表和用户留存表中导入数据  APP=imall\_one  do\_date=$1  if [ ! -n "$do\_date" ] ;then  do\_date=`date -d "-1 day" +%F`  fi  sql="  use $APP;  insert into table dws\_new\_mid\_day  select  uv.mid ,  uv.appid,  uv.uid,  uv.vs,  uv.os ,  uv.ch ,  uv.area,  '$do\_date'  from dws\_uv\_detail\_day uv  left join dws\_new\_mid\_day nm  on uv.mid =nm.mid  where nm.mid is null;    insert overwrite table dws\_user\_retention\_day partition(dt='$do\_date')  select  uv.mid ,  uv.uid, uv.vs, uv.os , uv.ch ,  uv.appid,  uv.area,  date\_add('$do\_date',-1) new\_create\_date,  1 retention\_days  from dws\_new\_mid\_day nm join dws\_uv\_detail\_day uv on nm.mid=uv.mid  where nm.create\_date= date\_add('$do\_date',-1) and uv.dt='$do\_date'  union all  select  uv.mid ,  uv.uid, uv.vs, uv.os , uv.ch ,  uv.appid,  uv.area,  date\_add('$do\_date',-2) new\_create\_date,  2 retention\_days  from dws\_new\_mid\_day nm join dws\_uv\_detail\_day uv on nm.mid=uv.mid  where nm.create\_date= date\_add('$do\_date',-2) and uv.dt='$do\_date'  union all  select  uv.mid ,  uv.uid, uv.vs, uv.os , uv.ch ,  uv.appid,  uv.area,  date\_add('$do\_date',-3) new\_create\_date,  3 retention\_days  from dws\_new\_mid\_day nm join dws\_uv\_detail\_day uv on nm.mid=uv.mid  where nm.create\_date= date\_add('$do\_date',-3) and uv.dt='$do\_date'  union all  select  uv.mid ,  uv.uid, uv.vs, uv.os , uv.ch ,  uv.appid,  uv.area,  date\_add('$do\_date',-4) new\_create\_date,  4 retention\_days  from dws\_new\_mid\_day nm join dws\_uv\_detail\_day uv on nm.mid=uv.mid  where nm.create\_date= date\_add('$do\_date',-4) and uv.dt='$do\_date'  union all  select  uv.mid ,  uv.uid, uv.vs, uv.os , uv.ch ,  uv.appid,  uv.area,  date\_add('$do\_date',-5) new\_create\_date,  5 retention\_days  from dws\_new\_mid\_day nm join dws\_uv\_detail\_day uv on nm.mid=uv.mid  where nm.create\_date= date\_add('$do\_date',-5) and uv.dt='$do\_date'  union all  select  uv.mid ,  uv.uid, uv.vs, uv.os , uv.ch ,  uv.appid,  uv.area,  date\_add('$do\_date',-6) new\_create\_date,  6 retention\_days  from dws\_new\_mid\_day nm join dws\_uv\_detail\_day uv on nm.mid=uv.mid  where nm.create\_date= date\_add('$do\_date',-6) and uv.dt='$do\_date'  ;  "  /opt/module/spark/bin/spark-sql -e "$sql" |

**问题：还有一个SQL，更改的问题**

### ADS层

#### 每日新增设备数量表

##### 建表：

统计出每日新增的设备的数量表：

|  |
| --- |
| drop table if exists `dws\_new\_mid\_count`;  create table `**dws\_new\_mid\_count**`  (  `create\_date` string comment '创建时间' ,  `new\_mid\_count` BIGINT comment '新增设备数量'  ) COMMENT '每日新增设备信息数量'  stored as parquet  location '/warehouse/imall\_one/dws/dws\_new\_mid\_count/'  tblproperties ("parquet.compression"="snappy"); |

##### 插入数据：

|  |
| --- |
| insert into table ads\_new\_mid\_count  select '2019-02-26',count(\*) from dws\_new\_mid\_day where create\_date='2019-02-26'; |

#### 留存设备数量表

##### 建表

当日（这个当日就是新增的那天，对应着设备新增日期）新增设备数就是分母

留存数量是分子，统计当日的留存数量；

新增设备日期就是分母对应的日期。

新增设备日期 + 截止到当前留存的天数=当前日期

|  |
| --- |
| drop table if exists `ads\_user\_retention\_day\_count`;  create table `**ads\_user\_retention\_day\_count**`  (  stat\_date string comment '统计日期',  retention\_count bigint comment '留存数量',  `create\_date` string comment '设备新增日期',  `new\_mid\_count` string comment '当日设备新增数量',  `retention\_day` int comment '截止当前日期留存天数',  `retention\_ratio` decimal(10,2) comment '留存率'  ) COMMENT '每日用户留存情况'  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ads/ads\_user\_retention\_day\_count/'; |

##### 导入数据：

|  |
| --- |
| insert into table ads\_user\_retention\_day\_count  select  '2019-02-27' stat\_dt, --统计日期  ur.retention\_count, --留存数量  ur.create\_date, --新增用户的日期  nm.new\_mid\_count, --当日新增设备数  ur.retention\_day, --留存天数  cast(retention\_count / nm.new\_mid\_count as decimal(10,2)) retention\_ratio  from  (select retention\_day,create\_date,count(\*) retention\_count from dws\_user\_retention\_day  where dt='2019-02-27'  group by retention\_day,create\_date  )  ur join ads\_new\_mid\_count nm on ur.create\_date=nm.create\_date; |

### ads\_retention.sh脚本

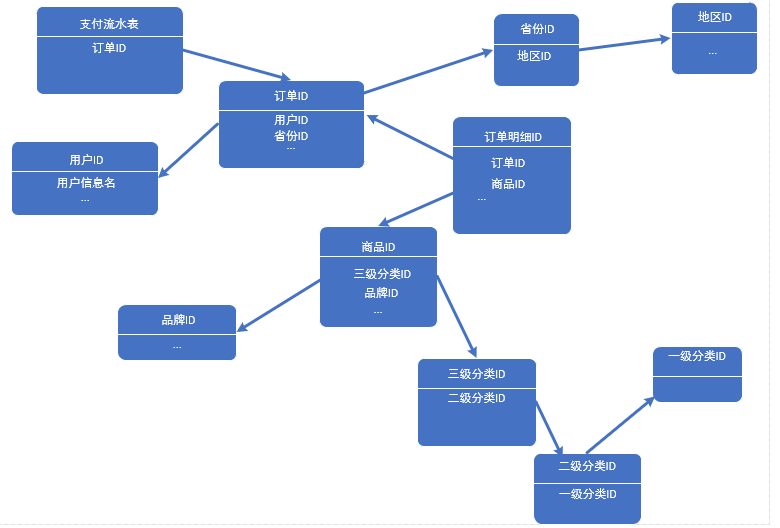
该脚本完成最终的报表数据

|  |
| --- |
| #!/bin/bash  APP=imall\_one  do\_date=$1  if [ ! -n "$do\_date" ] ;then  do\_date=`date -d "-1 day" +%F`  fi  sql="  use $APP;  insert into table ads\_user\_retention\_day\_count  select  '$do\_date' stat\_dt, --统计日期  if(ur.retention\_count is null,0,ur.retention\_count) retention\_count, --留存数量  nm.create\_date, --新增用户的日期  nm.new\_mid\_count, --当日新增设备数  datediff('$do\_date',nm.create\_date) retention\_day, --留存天数  cast(retention\_count / nm.new\_mid\_count as decimal(10,2)) retention\_ratio  from  (select create\_date,new\_mid\_count from ads\_new\_mid\_count where  create\_date < '$do\_date'  and create\_date >= date\_add('$do\_date',-6)  ) nm  left join  (select retention\_day,create\_date,count(\*) retention\_count from dws\_user\_retention\_day  where dt='$do\_date'  group by retention\_day,create\_date  )  ur on ur.create\_date=nm.create\_date;"  /opt/module/spark/bin/spark-sql -e "$sql" |

## 业务系统中结构化数据的采集

### 表结构

电商的业务中会存在很多的表，如下：

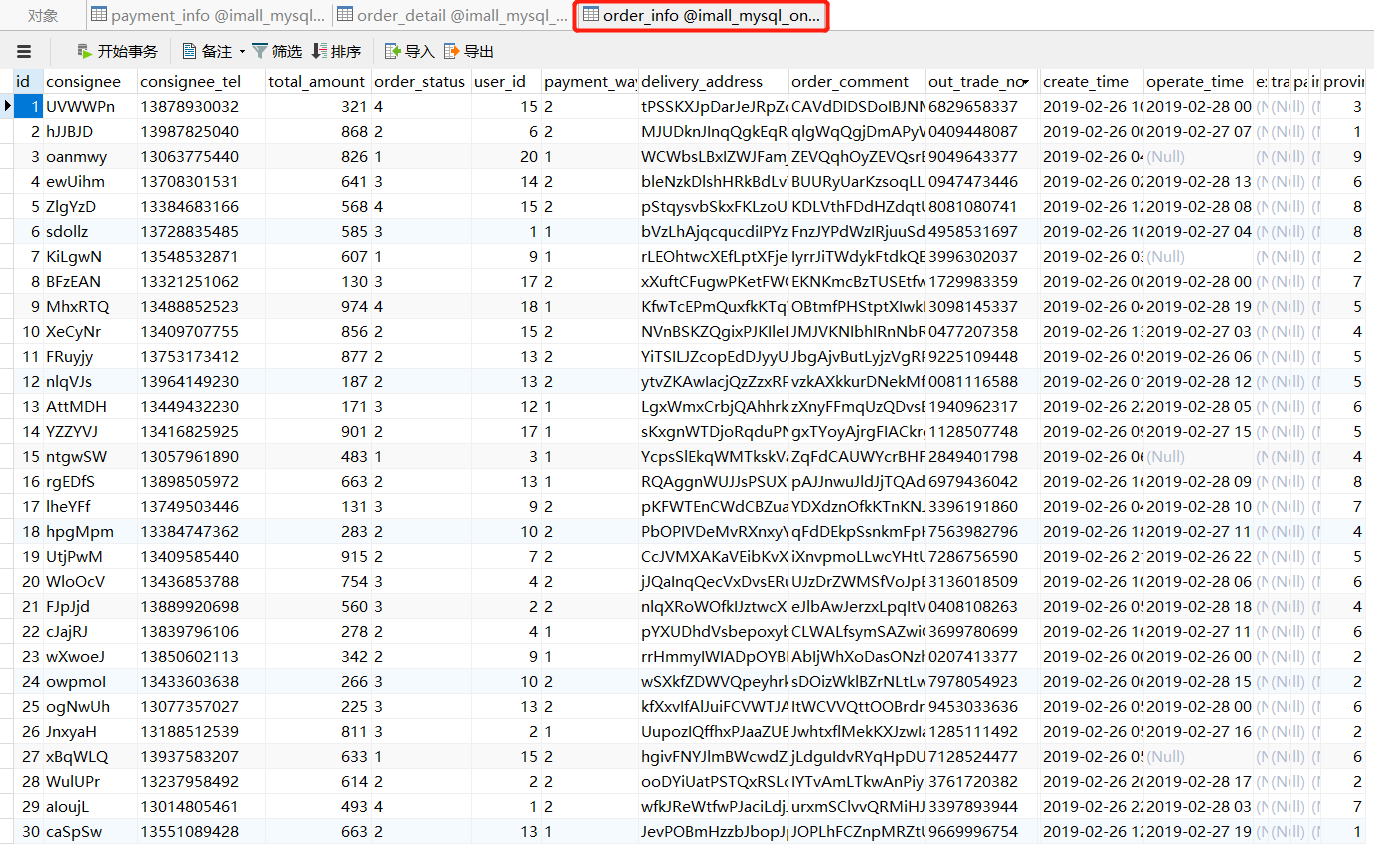


### 重要的表的样例信息

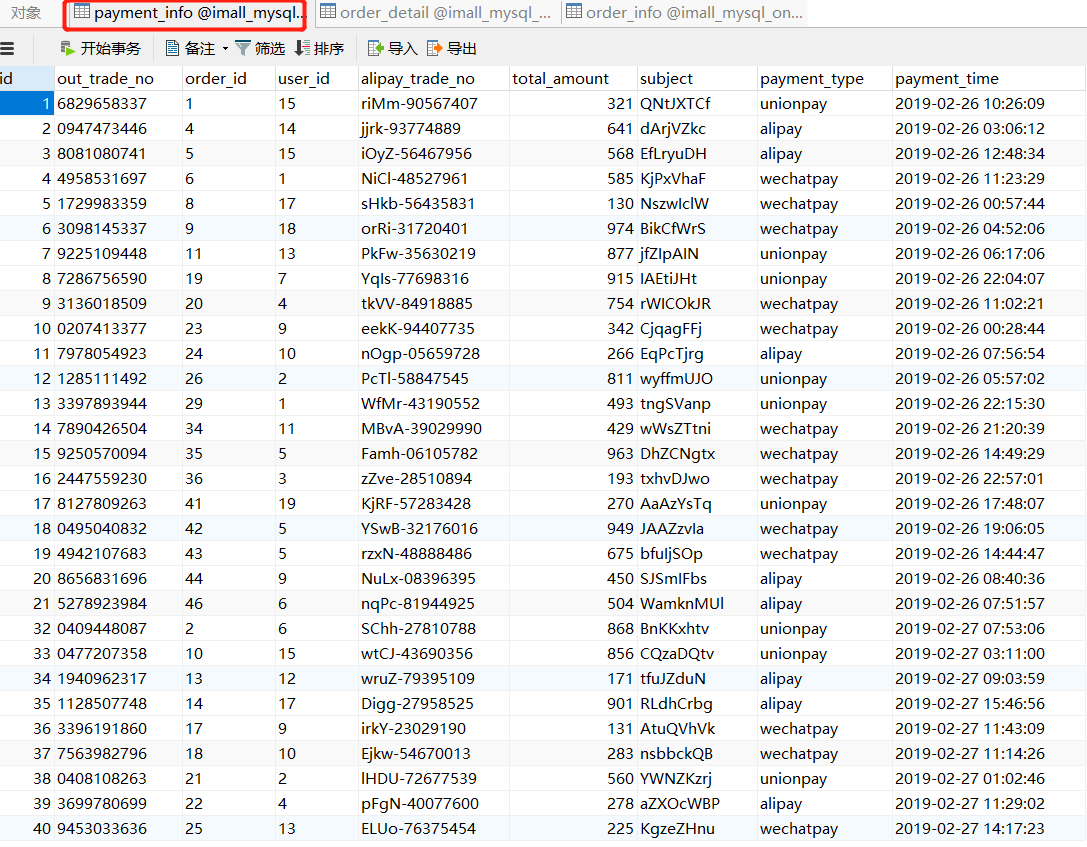
#### 订单详情表：



#### 订单表：



#### 账单流水表：



### 模拟数据

SQL在MySQL中建表，作为模拟数据。

1. 创建了数据库imall\_mysql\_one，
2. 建立base\_category1，一级分类表（id1，name），并添加了数据18条 。每日全量
3. 建立base\_category2，二级分类表（id2，name，id1），并添加了数据125条。每日全量
4. 建立base\_category3，三级分类表，（id3，name，id2），并添加数据1149条。
5. 建立base\_province，省份表（id，name，region\_id）并添加了9条数据。固定值
6. 建立base\_region，地区表（id，name）添加8条数据。固定值。
7. 建立base\_trademark，品牌表（id，name）添加了5条数据。固定值。
8. 建立order\_detail，订单详情表（id，order\_id，sku\_id，sku\_name，img\_url，order\_price，sku\_num购买的个数）添加了953条数据。但是多个id可属于同一个订单编号。共计350个订单。增量
9. 建立order\_info，订单表，订单表，包含收货人，收货人电话，订单状态，付款方式，创建时间，失效时间等字段信息，添加了350条数据。增量。
10. 建立payment\_info，支付流水表，包含id，订单编号，支付金额，支付方式，支付时间等字段。增量。
11. 建立sku\_info 表，有库存id，商品id，价格，sku名称，商品规格的描述，品牌，三级分类的信息，添加了10条数据。全量。
12. 建立user\_info 用户信息表，有编号，用户名称，昵称，性别，手机号，邮箱，创建时间等信息，添加了60条数据。每日全量。
13. 建立和多个函数和存储过程。

### 数据同步策略

#### 实体表

现实存在的一个实体，比如用户，商品等；这类数据的同步策略可以是每日全量；但是如果数据量比较大的话，可以将（半年以前的数据）做成拉链表，更久远的数据直接落盘。

这类表的特点是有新增，同时之前表中的数据还有可能被改变。

#### 维度表

维度表，一般对应着一些业务的状态，比如地区表，商品分类表，其同步策略是每日全量。

#### 固定维度表

不会变化的维度，比如性别，地区。此时做成固定值。

#### 事务型事实表

一旦发生，就不会在改变。同步策略是每日增量。

#### 周期型事实表

指的是随着业务的变化，不断产生变化的数据，比如订单的状态，有未支付到已支付。此时需要做成拉链表，只获取每日新增和变化的数据。

### Sqoop导入数据

我们要将存储在关系型数据库，MySQL中的数据通过Sqoop导入到HDFS文件系统中。这里的脚本每天都要被执行，先从数据库中查询，然后将查询的结果通过sqoop导入到HDFS文件系统中，这里查询的数据有的是每日全量，有的是每日增量。

#### 表分析



#### 导入脚本

**import\_db.sh**

|  |
| --- |
| #!/bin/bash  #将mysql中的数据，通过sqoop导入到HDFS中。  db\_date=$2  echo $db\_date  db\_name=imall\_mysql\_one  import\_data() {  /opt/module/sqoop/bin/sqoop import \  --connect jdbc:mysql://hadoop101:3306/$db\_name \  --username root \  --password isea \  --target-dir /origin\_data/imall\_one/db/$1/$db\_date \  --delete-target-dir \  --num-mappers 1 \  --fields-terminated-by "\t" \  --query "$2"' and $CONDITIONS;'  }  #每日全量  import\_base\_category1(){  import\_data "base\_category1" "select id,name from base\_category1 where 1=1"  }  #每日全量  import\_base\_category2(){  import\_data "base\_category2" "select id,name,category1\_id from base\_category2 where 1=1"  }  #每日全量  import\_base\_category3(){  import\_data "base\_category3" "select id,name,category2\_id from base\_category3 where 1=1"  }  #每日全量和变化量  import\_order\_info(){  }  #订单详情表，每日新增，如何知道是新增的呢？和订单表关联，是今天创建的就是新增的  import\_order\_detail(){  }  #每日全量  import\_sku\_info(){  import\_data "sku\_info" "select id,spu\_id,price,sku\_name,sku\_desc,weight,tm\_id,category3\_id,create\_time from sku\_info where 1=1"  }  #每日全量  import\_user\_info(){  import\_data "user\_info" "select id,name,birthday,gender,email,user\_level,create\_time from user\_info where 1=1"  }  #品牌表，每日全量  import\_base\_trademark(){  import\_data "base\_trademark" "select tm\_id,tm\_name from base\_trademark where 1=1"  }  #每日增量，付款的日期就是新增量  import\_payment\_info(){  import\_data "payment\_info" "select id, out\_trade\_no,order\_id,user\_id,alipay\_trade\_no,total\_amount, subject ,payment\_type,payment\_time from payment\_info where DATE\_FORMAT(payment\_time,'%Y-%m-%d')='$db\_date'"  }  #函数的调用  case $1 in  "base\_category1")  import\_base\_category1  ;;  "base\_category2")  import\_base\_category2  ;;  "base\_category3")  import\_base\_category3  ;;  "order\_info")  import\_order\_info  ;;  "order\_detail")  import\_order\_detail  ;;  "sku\_info")  import\_sku\_info  ;;  "base\_trademark")  import\_base\_trademark  ;;  "user\_info")  import\_user\_info  ;;  "payment\_info")  import\_payment\_info  ;;  "all")  import\_base\_category1  import\_base\_category2  import\_base\_category3  import\_order\_info  import\_order\_detail  import\_base\_trademark  import\_sku\_info  import\_user\_info  import\_payment\_info  ;;  esac |

## 数据建模理论：

### 函数依赖

#### 完全函数依赖：

只能用A来唯一确定B

#### 部分函数依赖：

通过AB能够唯一确定C，但是单独的A或者是单独的B也能确定C，那么C部分依赖于AB

##### 传递函数依赖：

A能确定B，B可以确定C，C不能确定A，那么C传递依赖于A。

### 范式

#### 第一范式

不能存在表中表

#### 第二范式

不能存在部分函数依赖

OLAP 和 OLTP是什么东西?

#### 第三范式

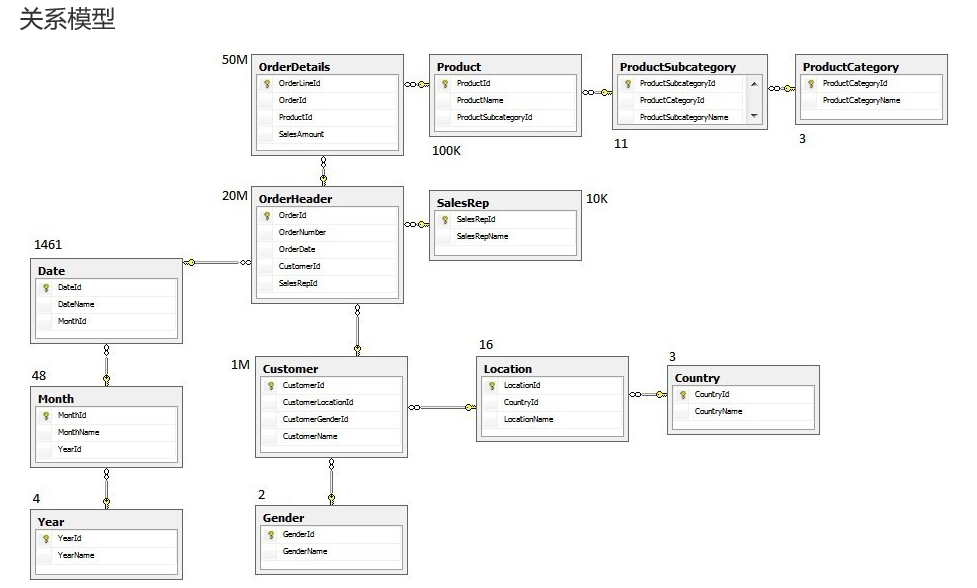
不存在传递函数依赖。

如果只有一个主键，那么一定满足第二范式；如果有两个属作为主键，不一定满足二范式。

### 建模

#### 关系建模

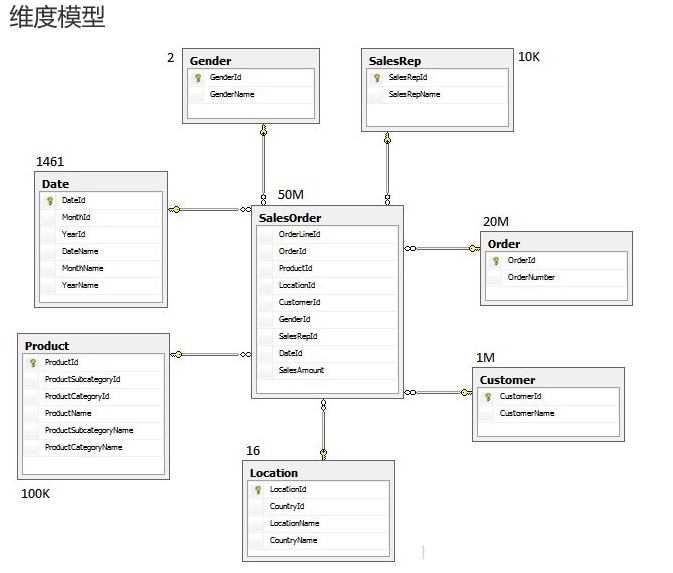
关系建模建立在范式的基础之上，表和表之间会建立比较紧密的联系。



关系模型只要应用于OLTP（T表示事务）系统中，为了保证数据的一致性和避免冗余，所以大部分业务系统中的表都是遵循第三范式的。

#### 维度建模

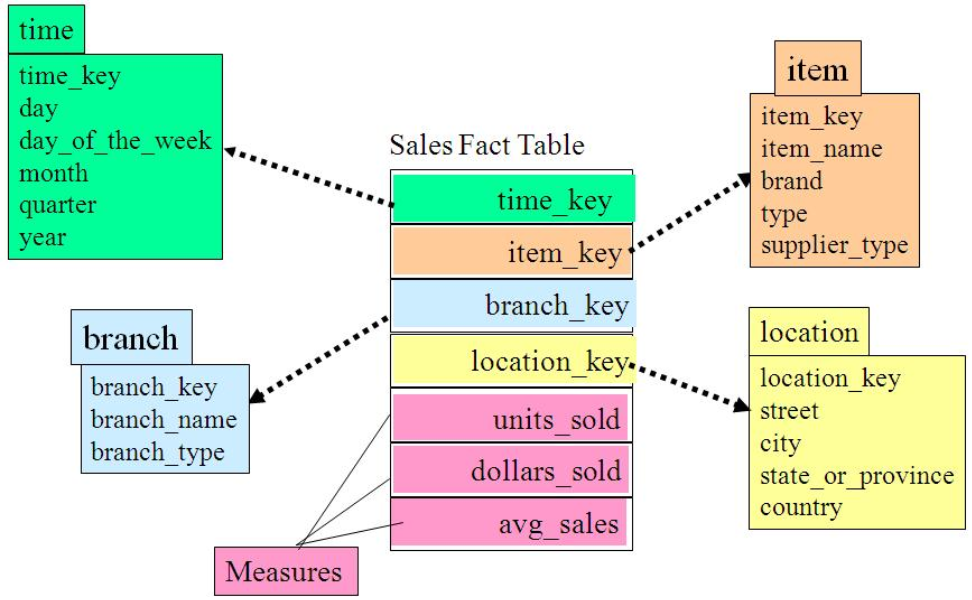
在维度建模的时候，将实体表和维度表都作为事实表；将事实表作为事实表。



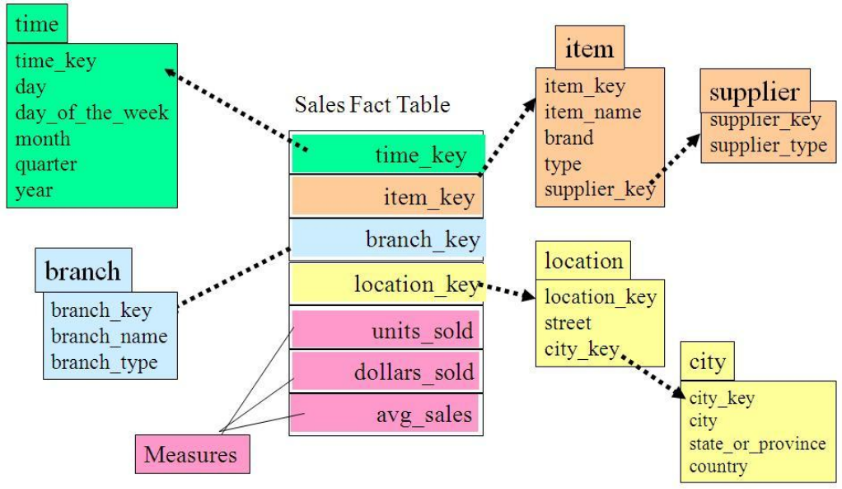
以一个事实表为中心，其他的维度表作为环绕。

#### 维度建模的三种模型：

##### 雪花模型

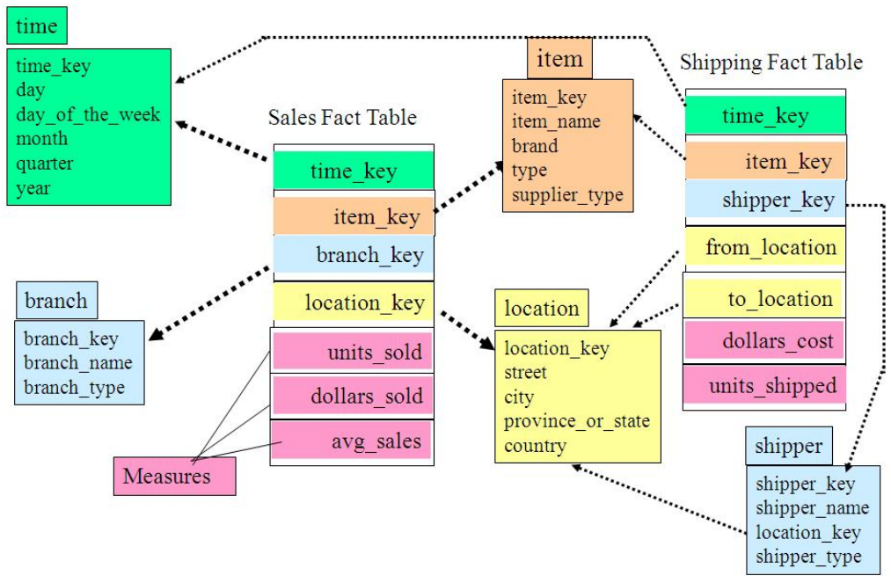
雪花模型与星型模型的区别主要在于维度的层级，标准的星型模型维度只有一层，而雪花模型可能会涉及多级

##### 星型模型



雪花模型介于关系模型和星座模型之间，比较靠近3NF,但是无法完全遵守，因为遵循3NF的性能成本太高**。**

##### 星座模型



星座模型与前两种情况的区别是事实表的数量，星座模型是基于多个事实表。

## 数据建模

上面一章节是数据建模理论，这一章节，我们将具体应用上一章的数据建模理论，对业务系统中的表进行建模。

### ODS

#### 建表

|  |
| --- |
| drop table if exists ods\_order\_info;  create table **ods\_order\_info** (  `id` string COMMENT '订单编号',  `total\_amount` decimal(10,2) COMMENT '订单金额',  `order\_status` string COMMENT '订单状态',  `user\_id` string COMMENT '用户id' ,  `payment\_way` string COMMENT '支付方式',  `out\_trade\_no` string COMMENT '支付流水号',  `create\_time` string COMMENT '创建时间',  `operate\_time` string COMMENT '操作时间'  ) COMMENT '订单表'  PARTITIONED BY ( `dt` string)  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ods/ods\_order\_info/';        drop table if exists ods\_order\_detail;  create table **ods\_order\_detai**l(  `id` string COMMENT '订单编号',  `order\_id` string COMMENT '订单号',  `user\_id` string COMMENT '用户id' ,  `sku\_id` string COMMENT '商品id',  `sku\_name` string COMMENT '商品名称',  `order\_price` string COMMENT '下单价格',  `sku\_num` string COMMENT '商品数量',  `create\_time` string COMMENT '创建时间'  ) COMMENT '订单明细表'  PARTITIONED BY ( `dt` string)  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ods/ods\_order\_detail/';      drop table if exists ods\_sku\_info;  create table **ods\_sku\_info**(  `id` string COMMENT 'skuId',  `spu\_id` string COMMENT 'spuid',  `price` decimal(10,2) COMMENT '价格' ,  `sku\_name` string COMMENT '商品名称',  `sku\_desc` string COMMENT '商品描述',  `weight` string COMMENT '重量',  `tm\_id` string COMMENT '品牌id',  `category3\_id` string COMMENT '品类id',  `create\_time` string COMMENT '创建时间'  ) COMMENT '商品信息'  PARTITIONED BY ( `dt` string)  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ods/ods\_sku\_info/';      drop table if exists ods\_user\_info;  create table **ods\_user\_info**(  `id` string COMMENT '用户id',  `name` string COMMENT '姓名',  `birthday` string COMMENT '生日' ,  `gender` string COMMENT '性别',  `email` string COMMENT '邮箱',  `user\_level` string COMMENT '用户等级',  `create\_time` string COMMENT '创建时间'  ) COMMENT '用户信息'  PARTITIONED BY ( `dt` string)  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ods/ods\_user\_info/';        drop table if exists ods\_base\_category1;  create table **ods\_base\_category1**(  `id` string COMMENT '用户id',  `name` string COMMENT '名称'  ) COMMENT '用户信息'  PARTITIONED BY ( `dt` string)  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ods/ods\_base\_category1/';    drop table if exists ods\_base\_category2;  create external **table ods\_base\_category2**(  `id` string COMMENT '用户id',  `name` string COMMENT '名称',  category1\_id string COMMENT '一级品类id'  ) COMMENT '用户信息'  PARTITIONED BY ( `dt` string)  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ods/ods\_base\_category2/';    drop table if exists ods\_base\_category3;  create table **ods\_base\_category3**(  `id` string COMMENT '用户id',  `name` string COMMENT '名称',  category2\_id string COMMENT '二级品类id'  ) COMMENT '用户信息'  PARTITIONED BY ( `dt` string)  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ods/ods\_base\_category3/';    drop table if exists `ods\_payment\_info`;  create table `**ods\_payment\_info**`(  `id` bigint COMMENT '编号',  `out\_trade\_no` string COMMENT '对外业务编号',  `order\_id` string COMMENT '订单编号',  `user\_id` string COMMENT '用户编号',  `alipay\_trade\_no` string COMMENT '支付宝交易流水编号',  `total\_amount` decimal(16,2) COMMENT '支付金额',  `subject` string COMMENT '交易内容',  `payment\_type` string COMMENT '支付类型',  `payment\_time` string COMMENT '支付时间'  ) COMMENT '支付流水表'  PARTITIONED BY ( `dt` string)  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ods/ods\_payment\_info/';      drop table if exists `ods\_base\_province`;  create table `**ods\_base\_province**`(`id` int COMMENT '省份id',`name` string COMMENT '名称',`region\_id` int COMMENT '地区id')  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ods/ods\_base\_province';  drop table if exists `**ods\_base\_trademark**`;  create table `ods\_base\_trademark`(`tm\_id` int COMMENT '品牌id',`tm\_name` string COMMENT '品牌名称')  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ods/ods\_base\_trademark';  drop table if exists `ods\_base\_region`;  create table `**ods\_base\_region**`(`id` int COMMENT '地区id',`region\_name` string COMMENT '地区名称')  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ods/ods\_base\_region'; |

这里一共建立了11张表

#### 导入数据

##### 将固定值表数据的导入

|  |
| --- |
| insert into table `ods\_base\_province`(`id`,`name`,`region\_id`) values  (1,'北京','1'),  (2,'上海','2'),  (3,'广东','5'),  (4,'黑龙江','3'),  (5,'陕西','7'),  (6,'四川','8'),  (7,'河北','1'),  (8,'浙江','6'),  (9,'安徽','6');      insert into table `ods\_base\_region`(`id`,`region\_name`) values  ('1','华北'),  ('2','华东'),  ('3','东北'),  ('4','华中'),  ('5','华南'),  ('6','东南'),  ('7','西北'),  ('8','西南'); |

##### ODS层导入脚本

把MySQL中的数据导入到HDFS文件系统中

将同步策略为每日全量或者是每日增量的数据导入

|  |
| --- |
| #!/bin/bash  #将从MySQL导入到HDFS中的数据，加载到hive的表中  do\_date=$1  project=imall\_one  hive=/opt/module/hive/bin/hive  #override表示对之前的数据进行覆盖  $hive -e " load data inpath '/origin\_data/$project/db/order\_info/$do\_date' OVERWRITE into table $project"".ods\_order\_info partition(dt='$do\_date')";  $hive -e " load data inpath '/origin\_data/$project/db/base\_trademark/$do\_date' OVERWRITE into table $project"".ods\_base\_trademark partition(dt='$do\_date')";  $hive -e " load data inpath '/origin\_data/$project/db/order\_detail/$do\_date' OVERWRITE into table $project"".ods\_order\_detail partition(dt='$do\_date')";  $hive -e "load data inpath '/origin\_data/$project/db/sku\_info/$do\_date' OVERWRITE into table $project"".ods\_sku\_info partition(dt='$do\_date')";  $hive -e "load data inpath '/origin\_data/$project/db/user\_info/$do\_date' OVERWRITE into table $project"".ods\_user\_info partition(dt='$do\_date')";  $hive -e "load data inpath '/origin\_data/$project/db/payment\_info/$do\_date' OVERWRITE into table $project"".ods\_payment\_info partition(dt='$do\_date')";  $hive -e "load data inpath '/origin\_data/$project/db/base\_category1/$do\_date' OVERWRITE into table $project"".ods\_base\_category1 partition(dt='$do\_date')";  $hive -e "load data inpath '/origin\_data/$project/db/base\_category2/$do\_date' OVERWRITE into table $project"".ods\_base\_category2 partition(dt='$do\_date')";  $hive -e "load data inpath '/origin\_data/$project/db/base\_category3/$do\_date' OVERWRITE into table $project"".ods\_base\_category3 partition(dt='$do\_date')"; |

### DWD

对表数据降维

#### 建表

|  |
| --- |
| drop table if exists dwd\_order\_info;  create external table **dwd\_order\_info** (  `id` string COMMENT '',  `total\_amount` decimal(10,2) COMMENT '',  `order\_status` string COMMENT ' 1 未支付 2 已支付 3 待发货 4 已发货 5 已签收',  `user\_id` string COMMENT 'id' ,  `payment\_way` string COMMENT '',  `out\_trade\_no` string COMMENT '',  `create\_time` string COMMENT '',  `operate\_time` string COMMENT '' ,  `province\_id` string COMMENT ''  ) COMMENT ''  PARTITIONED BY ( `dt` string)  stored as parquet  location '/warehouse/imall\_one/dwd/dwd\_order\_info/'  tblproperties ("parquet.compression"="snappy");    drop table if exists dwd\_order\_detail;  create external table **dwd\_order\_detail**(  `id` string COMMENT '',  `order\_id` decimal(10,2) COMMENT '',  `user\_id` string COMMENT 'id' ,  `sku\_id` string COMMENT 'id',  `sku\_name` string COMMENT '',  `order\_price` string COMMENT '',  `sku\_num` string COMMENT '',  `create\_time` string COMMENT ''  ) COMMENT ''  PARTITIONED BY ( `dt` string)  stored as parquet  location '/warehouse/imall\_one/dwd/dwd\_order\_detail/'  tblproperties ("parquet.compression"="snappy")  ;    drop table if exists dwd\_sku\_info;  create external table **dwd\_sku\_info**(  `id` string COMMENT 'skuId',  `spu\_id` string COMMENT 'spuid',  `price` decimal(10,2) COMMENT '' ,  `sku\_name` string COMMENT '',  `sku\_desc` string COMMENT '',  `weight` string COMMENT '',  `tm\_id` string COMMENT 'id',  `tm\_name` string COMMENT '品牌名称',  `category3\_id` string COMMENT '1id',  `category2\_id` string COMMENT '2id',  `category1\_id` string COMMENT '3id',  `category3\_name` string COMMENT '3',  `category2\_name` string COMMENT '2',  `category1\_name` string COMMENT '1',  `create\_time` string COMMENT ''  ) COMMENT ''  PARTITIONED BY ( `dt` string)  stored as parquet  location '/warehouse/imall\_one/dwd/dwd\_sku\_info/'  tblproperties ("parquet.compression"="snappy")  ;      drop table if exists dwd\_user\_info;  create external table **dwd\_user\_info**(  `id` string COMMENT 'id',  `name` string COMMENT '',  `birthday` string COMMENT '' ,  `gender` string COMMENT '',  `email` string COMMENT '',  `user\_level` string COMMENT '',  `create\_time` string COMMENT ''  ) COMMENT ''  PARTITIONED BY ( `dt` string)  stored as parquet  location '/warehouse/imall\_one/dwd/dwd\_user\_info/'  tblproperties ("parquet.compression"="snappy");      drop table if exists `dwd\_payment\_info`;  create external table `**dwd\_payment\_info**`(  `id` bigint COMMENT '',  `out\_trade\_no` string COMMENT '',  `order\_id` string COMMENT '',  `user\_id` string COMMENT '',  `alipay\_trade\_no` string COMMENT '',  `total\_amount` decimal(16,2) COMMENT '',  `subject` string COMMENT '',  `payment\_type` string COMMENT '',  `payment\_time` string COMMENT ''  ) COMMENT ''  PARTITIONED BY ( `dt` string)  stored as parquet  location '/warehouse/imall\_one/dwd/dwd\_payment\_info/'  tblproperties ("parquet.compression"="snappy"); |

对于其中一个固定表，插入固定的数据，只存一份：

|  |
| --- |
| drop table if exists dim\_base\_province;  create external table **dim\_base\_province**(  `id` string COMMENT 'id',  `province\_name` string COMMENT '',  `region\_id` string COMMENT '',  `region\_name` string COMMENT ''  ) COMMENT ''  stored as parquet  location '/warehouse/imall\_one/dim/dim\_base\_province/'  tblproperties ("parquet.compression"="snappy")  ;    insert overwrite table dim\_base\_province  select  p.id,  p.name,  p.region\_id,  r.region\_name  from ods\_base\_province p,ods\_base\_region r  where p.region\_id=r.id; |

#### 导入数据

|  |
| --- |
| set hive.exec.dynamic.partition.mode=nonstrict;  insert overwrite table dwd\_order\_info partition(dt)  select  \*  from ods\_order\_info  where dt='2019-02-28' and id is not null;      insert overwrite table dwd\_order\_detail partition(dt)  select  \*  from ods\_order\_detail  where dt='2019-02-28' and id is not null;    insert overwrite table dwd\_sku\_info partition(dt)  select  sku.`id` ,  `spu\_id` ,  `price` ,  `sku\_name` ,  `sku\_desc` ,  `weight` ,  sku.tm\_id ,  t.tm\_name ,  `category3\_id` ,  c2.id category2\_id ,  c1.id category1\_id,  c3.name category3\_name,  c2.name category2\_name,  c1.name category1\_name,  `create\_time` ,  sku.dt  from  ods\_sku\_info sku  left join ods\_base\_category3 c3 on sku.category3\_id=c3.id  left join ods\_base\_category2 c2 on c3.category2\_id=c2.id  left join ods\_base\_category1 c1 on c2.category1\_id=c1.id  left join ods\_base\_trademark t on t.tm\_id=sku.tm\_id  where sku.dt='2019-02-28' and c2.dt='2019-02-28'  and c3.dt='2019-02-28' and c1.dt='2019-02-28'  and t.dt='2019-02-28'  and sku.id is not null;    insert overwrite table dwd\_user\_info partition(dt)  select  \*  from ods\_user\_info  where dt='2019-02-28' and id is not null;      insert overwrite table dwd\_payment\_info partition(dt)  select  \*  from ods\_payment\_info  where dt='2019-02-28' and id is not null; |

#### ODS层导入脚本

|  |
| --- |
| #!/bin/bash  # 将ODS层Hive表中的数据导入到DWD层的Hive表中  APP=imall\_one  do\_date=$1  if [ ! -n "$do\_date" ] ;then  do\_date=`date -d "-1 day" +%F`  fi  sql=" use $APP;    set hive.exec.dynamic.partition.mode=nonstrict;    insert overwrite table dwd\_order\_info partition(dt)  select  \*  from ods\_order\_info  where dt='$do\_date' and id is not null;        insert overwrite table dwd\_order\_detail partition(dt)  select  \*  from ods\_order\_detail  where dt='$do\_date' and id is not null;    insert overwrite table dwd\_sku\_info partition(dt)  select  sku.id ,  spu\_id,  price,  sku\_name,  sku\_desc,  weight,  sku.tm\_id,  t.tm\_name,  category3\_id,  c2.id category2\_id ,  c1.id category1\_id,  c3.name category3\_name,  c2.name category2\_name,  c1.name category1\_name,  create\_time,  sku.dt  from  ods\_sku\_info sku  left join ods\_base\_category3 c3 on sku.category3\_id=c3.id  left join ods\_base\_category2 c2 on c3.category2\_id=c2.id  left join ods\_base\_category1 c1 on c2.category1\_id=c1.id  left join ods\_base\_trademark t on t.tm\_id=sku.tm\_id  where sku.dt='$do\_date' and c2.dt='$do\_date'  and c3.dt='$do\_date' and c1.dt='$do\_date'  and t.dt='$do\_date'  and sku.id is not null;    insert overwrite table dwd\_user\_info partition(dt)  select  \*  from ods\_user\_info  where dt='$do\_date' and id is not null;    insert overwrite table dwd\_payment\_info partition(dt)  select  \*  from ods\_payment\_info  where dt='$do\_date' and id is not null;  "    /opt/module/spark/bin/spark-sql -e "$sql"; |

### DWS

#### 设计用户行为宽表

上面我们在DWD层，将多个维度表进行了聚合，接下来我们需要将多个事实表进行聚合

需求目标：把每个用户单日的操作聚合起来组成一张多列宽表，以便以后关联用户维度信息，进行不同角度的分析。

每个用户都有自己信息和各站操作（比如下单，添加到购物车，付款等）

这张表的特点是，可以以用户为单位，进行各种各样的聚合，是我们达成目的的一个中间表

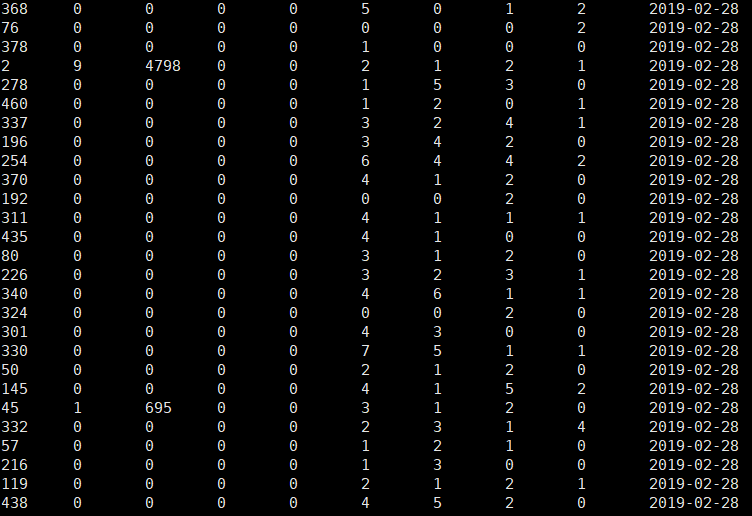
一条记录标明的是某一个用户在某一天，干了什么事情 => （用户+日期）

#### 建表

|  |
| --- |
| drop table if exists dws\_user\_action;  create table dws\_user\_action  ( user\_id string comment '用户 id',  order\_count bigint comment '下单次数 ',  order\_amount decimal(16,2) comment '下单金额 ',  payment\_count bigint comment '支付次数',  payment\_amount decimal(16,2) comment '支付金额 ',  clickItem\_ct bigint comment '点击数',  addcart\_ct bigint comment '购物车添加次数 ',  favorite\_ct bigint comment '收藏次数',  comment\_ct bigint comment '评论次数'  ) COMMENT '每日用户行为宽表'  PARTITIONED BY ( `dt` string)  stored as parquet  location '/warehouse/imall\_one/dws/dws\_user\_action/'  tblproperties ("parquet.compression"="snappy"); |

#### 导入数据

|  |
| --- |
| with  tmp\_order as(  select user\_id,count(\*) order\_count,sum(total\_amount) order\_amount  from  dwd\_order\_info where dt='2019-02-28'  group by user\_id  ),  tmp\_payment as(  select user\_id,count(\*) payment\_count,sum(total\_amount) payment\_amount  from  dwd\_payment\_info where dt='2019-02-29'  group by user\_id  ),  tmp\_event as(  select uid,  sum(if(evid='clickItem',1,0)) click\_item\_ct,  sum(if(evid='addComment',1,0)) click\_comment\_ct,  sum(if(evid='addCart',1,0)) click\_cart\_ct,  sum(if(evid='addFavor',1,0)) click\_favor\_ct  from  dwd\_event\_log  where dt='2019-02-28'  group by uid  )  insert overwrite table dws\_user\_action partition(dt='2019-02-28')  select user\_id,sum(order\_count),sum(order\_amount),sum(payment\_count),sum(payment\_amount),sum(click\_item\_ct),sum(click\_comment\_ct),sum(click\_cart\_ct),sum(click\_favor\_ct)  from  (  select user\_id,order\_count,order\_amount,0 payment\_count,0 payment\_amount,0 click\_item\_ct,0 click\_comment\_ct,0 click\_cart\_ct,0 click\_favor\_ct from tmp\_order  union all  select user\_id,0 order\_count,0 order\_amount, payment\_count,payment\_amount,0 click\_item\_ct,0 click\_comment\_ct,0 click\_cart\_ct,0 click\_favor\_ct from tmp\_payment  union all  select uid user\_id,0 order\_count,0 order\_amount,0 payment\_count,0 payment\_amount,click\_item\_ct,click\_comment\_ct,click\_cart\_ct,click\_favor\_ct from tmp\_event  ) total  group by user\_id; |



## 需求3：GMV

Gross Merchandise Volume 在一定时间内的总成交额

## 需求4：转化率

第五天最后一节的视屏就是讲这些内容的。

## 拉链表

拉链表，记录每条信息的生命周期，一旦记录的生命周期结束，就重新生成新的记录，并把当前日期放入生效开始日期，如果当前日期至今有效，在生效结束日期中填入一个极大的值，比如9999等。



### 为什么要做拉链表？

需要查看某些业务信息的某一个时间点当日的信息；

数据会发生变化，但是大部分时间是不变的（无法做每日增量）

数据量有一定的规模，无法按照每日全量的方式保存（没法做每日全量）

### 拉链表制作过程：

准备一份，完整的数据，开始的时候做一次，以后则不用了

把所有的数据，都加上生效开始日期，和有效的结束日期。

MySQL中的表和Hive的ODS层的表。



Hive中的表：



1月2号，MySQL中的订单全量表：



针对MySQL中的数据，通过判断是否是新增和变化的，新增的时间和修改时间得到订单变化表（新增+修改）然后在导入到Hive中。



然后将刚得到的这张表和初始化的拉链表合并，



产生一个新的拉链表：



##### 小总结：

1. 初始化，准备一份完整的数据，加上生效开始日期，和生效结束日期，做成拉链表
2. 每天：
   1. 从关系型数据库中导入，每日新增及变化数据
   2. 用新增及变化数据与之前的拉链表进行合并，写到临时拉链表
   3. 用临时拉链表覆盖掉之前的拉链表

### 如何得到新的拉链表？

我们上诉的两张表都在Hive中，而Hive中的数据是不能被修改的，那么如何做到将数据合并得到新的拉链表呢？

这里我们使用一个结构一致的临时拉链表，将老的拉链表和刚得到的表（增量和变化量）放置在临时拉链表中，然后用临时拉链表对原来的拉链表进行覆盖。

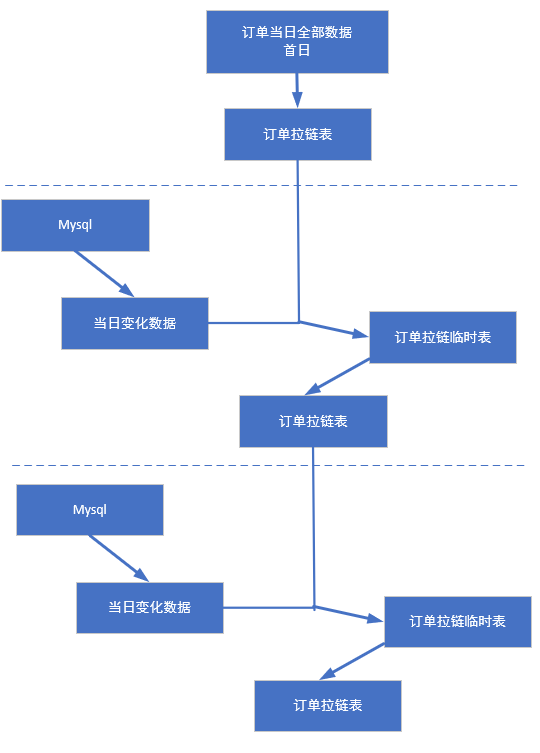
如何使用拉链表？

通过某个日期>=生效开始日期 且 某个日期<= 生效结束日期

##### 如何获取变化数据？

1. 如果在表中有相应的字段可以获知（比如在订单表中维护了一个修改时间的字段，可以获取变化的时间）如果在表中没有维护这样的字段。
2. 可以利用第三方的监控工具，比如canal，监控MySQL的实时变化，并做记录
3. 逐行比较前后两天的数据，（将可能变化的字段，联合起来，作MD5，看值是否相同）

##### 图解过程：



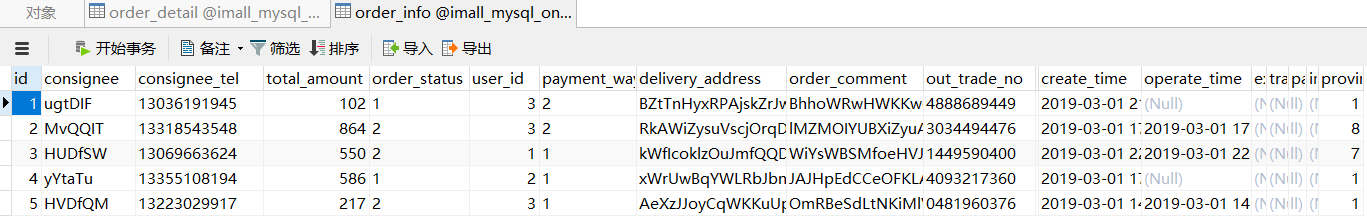
### 用户订单拉链表

#### 拉链表初始化

##### MySQL中全量表建表

需要MySQL中先生成新的数据：

在2019-03-01生成5条订单表，3个用户，覆盖之前的数据



##### Sqoop导入到HDFS中

|  |
| --- |
| /opt/module/sqoop/bin/sqoop import \  --connect jdbc:mysql://hadoop101:3306/imall\_mysql\_one \  --username root \  --password isea \  --target-dir /origin\_data/imall\_one/db/order\_info\_all/2019-03-01 \  --delete-target-dir \  --num-mappers 1 \  --fields-terminated-by "\t" \  --query 'select id,total\_amount,order\_status,user\_id,payment\_way,out\_trade\_no,create\_time,operate\_time,province\_id from order\_info where 1=1 and $CONDITIONS;' |

##### Hive中建全量表

|  |
| --- |
| drop table if exists ods\_order\_info\_all;  create table ods\_order\_info\_all(  `id` string COMMENT '订单编号',  `total\_amount` decimal(10,2) COMMENT '订单金额',  `order\_status` string COMMENT '订单状态',  `user\_id` string COMMENT '用户id' ,  `payment\_way` string COMMENT '支付方式',  `out\_trade\_no` string COMMENT '支付流水号',  `create\_time` string COMMENT '创建时间',  `operate\_time` string COMMENT '操作时间' ,  `province\_id` string COMMENT '省份'  ) COMMENT '订单表'  PARTITIONED BY ( `dt` string)  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ods/ods\_order\_info\_all/'; |

##### 加载数据到Hive表中

|  |
| --- |
| load data inpath '/origin\_data/imall\_one/db/order\_info\_all/2019-03-01' OVERWRITE into table imall\_one.ods\_order\_info\_all partition(dt='2019-03-01'); |

###### 拉链表是否分区？

是否分区，主要看数据的规模，一般而言，拉链表的数据规模远不及全量表；所以不必分区；是否分区还要看查询的时候是够框定条件，而拉链表很有可能造成跨分区搜索；

所以拉链表一般不分区，但是如果数据量非常的大的话，可以按照当前有效数据和历史数据区分开，增加标志is\_active把所有的结束日期为9999的放置在一个分区中。

##### 创建拉链表：

|  |
| --- |
| drop table if exists dwd\_order\_info\_his;  create table dwd\_order\_info\_his(  `id` string COMMENT '订单编号',  `total\_amount` decimal(10,2) COMMENT '订单金额',  `order\_status` string COMMENT '订单状态',  `user\_id` string COMMENT '用户id' ,  `payment\_way` string COMMENT '支付方式',  `out\_trade\_no` string COMMENT '支付流水号',  `create\_time` string COMMENT '创建时间',  `operate\_time` string COMMENT '操作时间' ,  `province\_id` string COMMENT '省份id' ,  **`start\_date` string COMMENT '有效开始日期',**  **`end\_date` string COMMENT '有效结束日期'**  ) COMMENT '订单拉链表'  stored as parquet  location '/warehouse/imall\_one/dwd/dwd\_order\_info\_his/'  tblproperties ("parquet.compression"="snappy"); |

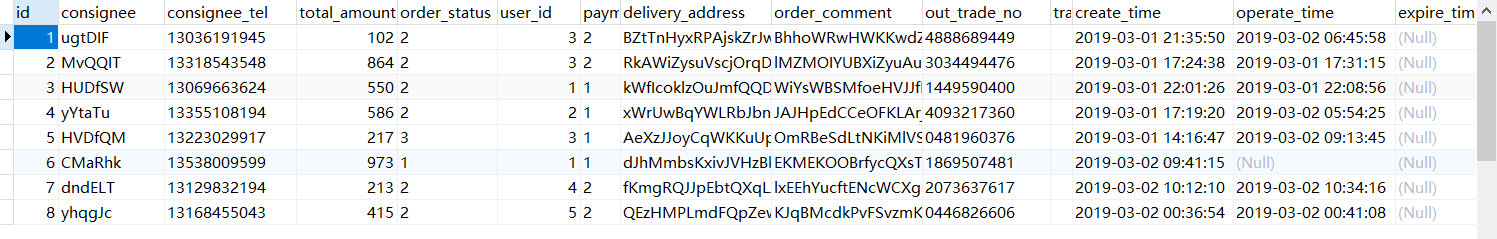
##### 向拉链表中导入数据

|  |
| --- |
| insert overwrite table dwd\_order\_info\_his  select  id ,  total\_amount ,  order\_status ,  user\_id ,  payment\_way ,  out\_trade\_no,  create\_time ,  operate\_time ,  province\_id ,  '2019-03-01' ,  '9999-99-99'  from ods\_order\_info\_all o where o.dt='2019-03-01'; |

#### 每天拉链表更新

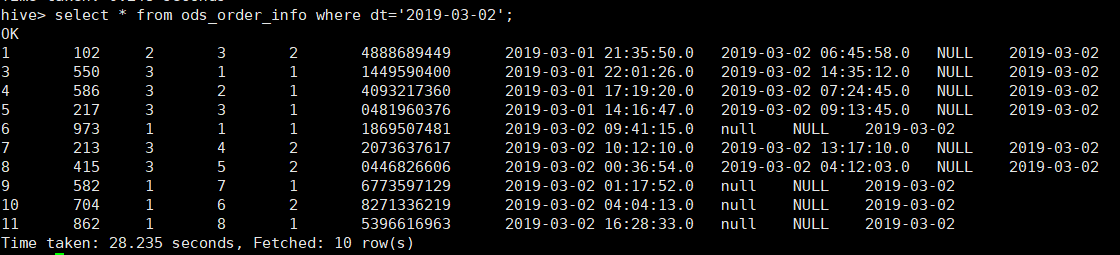
##### 新数据产生并->HDFS->Hive中

生成新一天的数据，3月2日，生成3条订单，3个用户，不覆盖，（原来的订单状态随机发生变化）

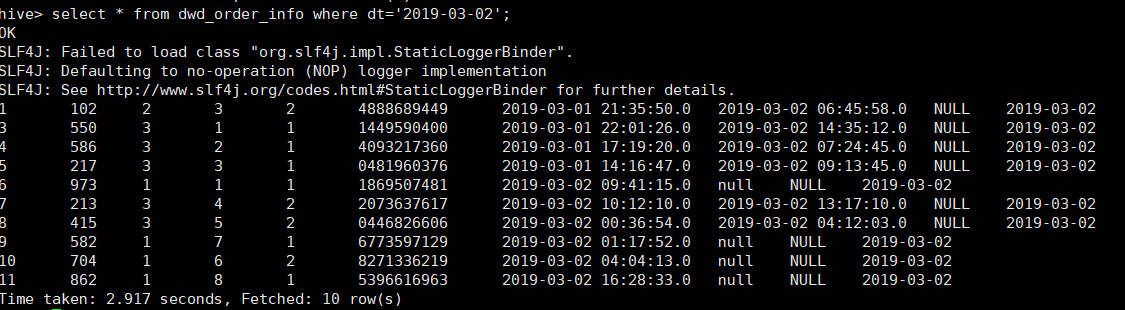


然后使用shell脚本将MySQL中的数据导入到HDFS文件系统中，**import\_db.sh**。

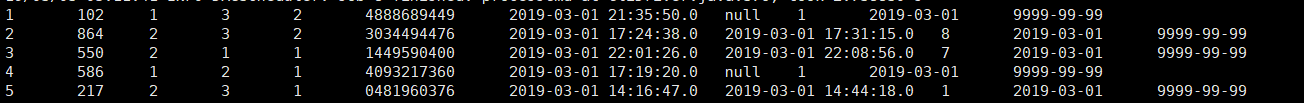
执行**ods\_load\_db.sh 脚本**



执行**dwd\_db.sh脚本**



上面的表将要和下面的拉链表，构成新的拉链表

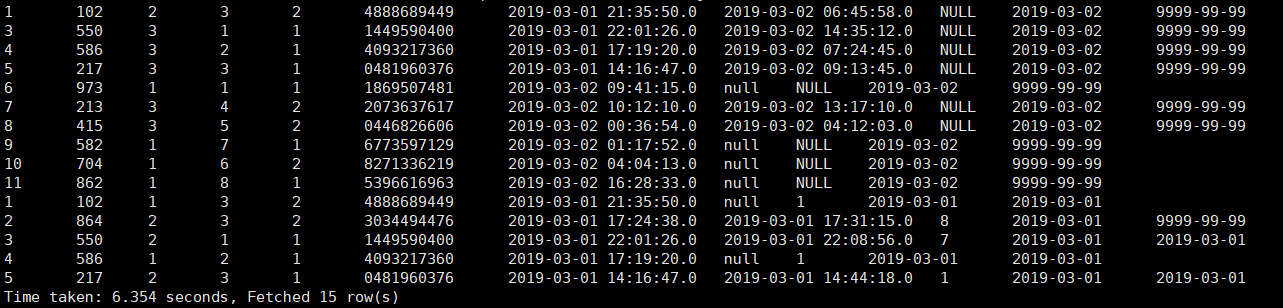


##### SQL语句：

要完成两者的合并，需要两步

1. 开链，将新增的数据查询出来，并且添加end\_state的值为9999-99-99
2. 关链，将更新的数据查询出来，并且修改将end\_state的时间9999-99-99替换掉

|  |
| --- |
| select  id,  total\_amount,  order\_status,  user\_id,  payment\_way,  out\_trade\_no,  create\_time,  operate\_time,  province\_id,  '2019-03-02' start\_date,  '9999-99-99' end\_date  from dwd\_order\_info o  where dt='2019-03-02' --开链  union all  select  his.id,  his.total\_amount,  his.order\_status,  his.user\_id,  his.payment\_way,  his.out\_trade\_no,  his.create\_time,  his.operate\_time,  his.province\_id,  his.start\_date,  if(upt.id is null,his.end\_date, date\_add('2019-03-02',-1)) end\_date  from  dwd\_order\_info\_his his left join  (select \* from dwd\_order\_info where dt='2019-03-02'  and date\_format(operate\_time,'yyyy-MM-dd')='2019-03-02') upt  on his.id=upt.id and his.end\_date='9999-99-99'; |



##### 临时拉链表覆盖原拉链表

1. 新建临时拉链表
2. 然后将查询结果导入到临时拉链表
3. 使用临时拉链表替换老的拉链表

新建临时拉链表

|  |
| --- |
| drop table if exists dwd\_order\_info\_his\_tmp;  create table dwd\_order\_info\_his\_tmp(  `id` string COMMENT '订单编号',  `total\_amount` decimal(10,2) COMMENT '订单金额',  `order\_status` string COMMENT '订单状态',  `user\_id` string COMMENT '用户id' ,  `payment\_way` string COMMENT '支付方式',  `out\_trade\_no` string COMMENT '支付流水号',  `create\_time` string COMMENT '创建时间',  `operate\_time` string COMMENT '操作时间',  `province\_id` string COMMENT '省份id',  `start\_date` string COMMENT '有效开始日期',  `end\_date` string COMMENT '有效结束日期'  ) COMMENT '订单拉链临时表'  stored as parquet  location '/warehouse/imall\_one/dwd/dwd\_order\_info\_his\_tmp/'  tblproperties ("parquet.compression"="snappy"); |

往临时拉链表中导入数据

|  |
| --- |
| insert into table dwd\_order\_info\_his\_tmp  select  id,  total\_amount,  order\_status,  user\_id,  payment\_way,  out\_trade\_no,  create\_time,  operate\_time,  province\_id,  '2019-03-02' start\_date,  '9999-99-99' end\_date  from dwd\_order\_info o  where dt='2019-03-02' --开链  union all  select  his.id,  his.total\_amount,  his.order\_status,  his.user\_id,  his.payment\_way,  his.out\_trade\_no,  his.create\_time,  his.operate\_time,  his.province\_id,  his.start\_date,  if(upt.id is null,his.end\_date, date\_add('2019-03-02',-1)) end\_date  from  dwd\_order\_info\_his his left join  (select \* from dwd\_order\_info where dt='2019-03-02'  and date\_format(operate\_time,'yyyy-MM-dd')='2019-03-02') upt  on his.id=upt.id and his.end\_date='9999-99-99' --关链  ; |

用临时拉链表替换老的拉链表：

|  |
| --- |
| insert overwrite table dwd\_order\_info\_his  select \* from --这里要写全  dwd\_order\_info\_his\_tmp ; |

##### 形成shell脚本：

|  |
| --- |
| #!/bin/bash  APP=imall\_one  do\_date=$1  if [ ! -n "$do\_date" ] ;then  do\_date=`date -d "-1 day" +%F`  fi  sql="  use $APP;  insert overwrite table dwd\_order\_info\_his\_tmp  select  id,  total\_amount ,  order\_status ,  user\_id ,  payment\_way ,  out\_trade\_no,  create\_time ,  operate\_time ,  province\_id,  '$do\_date' start\_date,  '9999-99-99' end\_date  from dwd\_order\_info o  where dt='$do\_date' ---开链  union all  select  his.id,  his.total\_amount ,  his.order\_status ,  his.user\_id ,  his.payment\_way ,  his.out\_trade\_no,  his.create\_time ,  his.operate\_time ,  his.province\_id,  his.start\_date,  if( upt.id is null ,his.end\_date, date\_add('$do\_date' ,-1)) end\_date  from dwd\_order\_info\_his his left join  (select \* from dwd\_order\_info where dt='$do\_date'  and date\_format(operate\_time,'yyyy-MM-dd')='$do\_date') upt  on his.id =upt.id and his.end\_date='9999-99-99';    insert overwrite table dwd\_order\_info\_his  select  \*  from dwd\_order\_info\_his\_tmp;"  /opt/module/spark/bin/spark-sql -e "$sql" |

测试：

1. 先在MySQL中产生新的订单，CALL init\_data('2019-03-03',2,3,FALSE)
2. ./import\_db.sh all 2019-03-03
3. ./ods\_load\_db.sh 2019-03-03
4. ./dwd\_db.sh 2019-03-03
5. ./his.sh 2019-03-03
6. 检查拉链表中的数据是否是正确的

### 用户拉链表

#### 用户的场景：

由于用户量的增加，（到百万千万级，）每日全量的存储方式会造成较大的冗余，而用户实际变化却很小，最近三个月之内的数据，用全量表统计，并且每个月做一次拉链表，对于三个月之前的数据弃用，三个月之前的全量数据可以弃用放弃归档，三个月之前的历史查询以拉链表为准。

#### 思路：

1. 初始化拉链表：利用全量表，直接从每日全量中生成
2. 之后每天：从关系型数据库中获取每日的新增和变化，
3. 也就是说拿第二天（昨日）的全量和第一天的全量做对比，然后得出一个差异，就是变化的差异（这里使用MD5）在组合create\_time得到当日新增。
4. 把每日新增和变化明细和 拉链表进行合并，存放到临时拉链表中
5. 把临时拉链表覆盖原拉链表

#### 形成过程；

##### 创建拉链表并初始化

|  |
| --- |
| drop table if exists dwd\_user\_info\_his;  create table dwd\_user\_info\_his(  `id` string COMMENT '用户id',  `name` string COMMENT '姓名',  `birthday` string COMMENT '生日' ,  `gender` string COMMENT '性别',  `email` string COMMENT '邮箱',  `user\_level` string COMMENT '用户等级',  `create\_time` string COMMENT '创建时间',  start\_date string COMMENT '有效开始日期',  end\_date string COMMENT '有效截止日期'  ) COMMENT '用户信息拉链表'  stored as parquet  location '/warehouse/imall\_one/dwd/dwd\_user\_info\_his/'  tblproperties ("parquet.compression"="snappy"); |

|  |
| --- |
| insert overwrite table dwd\_user\_info\_his  select \*,  '9999-99-99' end\_date  from dwd\_user\_info where dt='2019-03-03'; |

##### 保存新增及变化的表

|  |
| --- |
| drop table if exists dwd\_user\_info\_change;  create table dwd\_user\_info\_change(  `id` string COMMENT '用户id',  `name` string COMMENT '姓名',  `birthday` string COMMENT '生日' ,  `gender` string COMMENT '性别',  `email` string COMMENT '邮箱',  `user\_level` string COMMENT '用户等级',  `create\_time` string COMMENT '创建时间'  ) COMMENT '用户信息新增及变化表'  stored as parquet  location '/warehouse/imall\_one/dwd/dwd\_user\_info\_change/'  tblproperties ("parquet.compression"="snappy"); |

##### 查询新增和变化插入到变化表中

|  |
| --- |
| insert overwrite table dwd\_user\_info\_change  select  ui.id ,  ui.name,  ui.birthday ,  ui.gender ,  ui.email ,  ui.user\_level ,  ui.create\_time  from dwd\_user\_info ui  where dt='2019-03-05' and id not in  (select id from dwd\_user\_info\_his where end\_date ='9999-99-99')  union ALL  select  tday.id ,  tday.name,  tday.birthday ,  tday.gender ,  tday.email ,  tday.user\_level ,  tday.create\_time  from  (select \* ,md5(concat(name,gender)) md5\_ from dwd\_user\_info where where dt='2019-03-05') tday  join  (select \*,md5(concat(name,gender)) md5\_ from dwd\_user\_info\_his where end\_date='9999-99-99' ) yday  on tday.id=yday.id and tday.md5\_ <> yday.md5\_; |

##### 创建临时拉链表：

|  |
| --- |
| drop table if exists dwd\_user\_info\_his\_tmp;  create table dwd\_user\_info\_his\_tmp(  `id` string COMMENT '用户id',  `name` string COMMENT '姓名',  `birthday` string COMMENT '生日' ,  `gender` string COMMENT '性别',  `email` string COMMENT '邮箱',  `user\_level` string COMMENT '用户等级',  `create\_time` string COMMENT '创建时间',  start\_date string COMMENT '有效开始日期',  end\_date string COMMENT '失效日期'  ) COMMENT '用户信息拉链表'  stored as parquet  location '/warehouse/imall\_one/dwd/dwd\_user\_info\_his\_tmp/'  tblproperties ("parquet.compression"="snappy"); |

##### 将每日新增和变化明细和拉链表合并；放入到临时拉链表

|  |
| --- |
| insert overwrite table dwd\_user\_info\_his\_tmp  select --开链  ui.id ,  ui.name,  ui.birthday ,  ui.gender ,  ui.email ,  ui.user\_level ,  ui.create\_time ,  '2019-03-05' start\_date,  '9999-99-99' end\_date  from dwd\_user\_info\_change ui  union ALL  select --关链，关闭匹配上，且正在生效中的数据  his.id ,  his.name,  his.birthday ,  his.gender ,  his.email ,  his.user\_level ,  his.create\_time ,  his.start\_date,  if(uc.id is null,his.end\_date,date\_add('2019-03-05',-1)) end\_date --有效截止日期应该是当天的前一天，没有匹配则保持原来的值  from dwd\_user\_info\_his his left join  dwd\_user\_info\_change uc  on his.id = uc.id and his.end\_date = '9999-99-99'; |

##### 临时拉链表覆盖老的拉链表

|  |
| --- |
| insert overwrite table dwd\_user\_info\_his  select \* from dwd\_user\_info\_his\_tmp; |

##### 脚本dwd\_uv\_his.sh

|  |
| --- |
| #!/bin/bash  # 制作拉链表  APP=imall\_one  do\_date=$1  if [ ! -n "$do\_date" ] ;then  do\_date=`date -d "-1 day" +%F`  fi  sql=" use $APP;  ---制作新增及变化明细  insert overwrite table dwd\_user\_info\_change  select ui.id ,  ui.name,  ui.birthday ,  ui.gender ,  ui.email ,  ui.user\_level ,  ui.create\_time  from dwd\_user\_info ui -- 新增 比较原拉链表是否存在  where dt='$do\_date' and id not in  ( select id from dwd\_user\_info\_his where end\_date='9999-99-99' )  union all  select  tday.id ,  tday.name,  tday.birthday ,  tday.gender ,  tday.email ,  tday.user\_level ,  tday.create\_time from -- 变化数据 比较原拉链表是否一致  ( select \*,md5(concat(name,gender)) md5\_ from dwd\_user\_info where dt='$do\_date') tday  join  ( select \*,md5(concat(name,gender)) md5\_ from dwd\_user\_info\_his where end\_date='9999-99-99' ) yday  on tday.id =yday.id and tday.md5\_ <>yday.md5\_ ;  --- 把每日新增及变化明细 和 拉链表 进行合并 存放到 临时拉链表  insert overwrite table dwd\_user\_info\_his\_tmp  select \* from  (  select ---开链  ui.id ,  ui.name,  ui.birthday ,  ui.gender ,  ui.email ,  ui.user\_level ,  ui.create\_time,  '$do\_date' start\_date,  '9999-99-99' end\_date  from dwd\_user\_info\_change ui  union all  select his.id,  his.name,  his.birthday ,  his.gender ,  his.email ,  his.user\_level ,  his.create\_time,  his.start\_date,  if( uc.id is null ,his.end\_date,date\_add('$do\_date',-1)) end\_date --所有匹配上的进行关链（取前一天） ，没匹配保持原日期  from dwd\_user\_info\_his his left join -- 关链 关闭所有匹配上的 且正在生效中的数据  dwd\_user\_info\_change uc on his.id=uc.id and his.end\_date='9999-99-99'  ) order by id ,start\_date;    --- 临时拉链表 覆盖原拉链表  insert overwrite table dwd\_user\_info\_his  select \* from dwd\_user\_info\_his\_tmp ;  "  /opt/module/spark/bin/spark-sql -e "$sql"; |

insert overwrite是删除原有数据然后在新增数据，如果有分区那么只会删除指定分区数据，其他分区数据不受影响。

## 需求5：复购率

留存G复活

复购率：又买的人数 / 买过的人数

各个一级品类下，品牌月单次复购率，和多次复购率。

单次复购率：买过两次以上的人 / 所有买过的人

多次复购率：买过三次以上的人 / 所有买过的人

以品牌为单位，因为一个品牌可能有多个品类，这里统计的是品牌下的品类：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 品牌 | 一级品类 | 品类名称 | 购买过该品牌的人数 | 购买过2次及2次以上的人数 | 单次复购率 | 购买过3次及3次以上的人数 | 多次复购率 |
| 品牌 A | 1 | 品类 A | 100 | 80 | 80% | 50 | 50% |
| 品牌 B | 20 | 品类 B | 200 | 120 | 60% | 40 | 20% |

### 思路：

考虑到以后的业务的扩展，我们应该在DWS层形成一张**宽表**：

什么人，在什么日子，买了什么东西，几次。

Dwd\_user\_info， dwd\_order\_detail，dwd\_sku\_info；

### 实现

#### 建表

|  |
| --- |
| drop table if exists dws\_sale\_detail\_daycount;  create table dws\_sale\_detail\_daycount  ( user\_id string comment '用户 id',  sku\_id string comment '商品 Id',  user\_gender string comment '用户性别',  user\_age string comment '用户年龄',  user\_level string comment '用户等级',  sku\_price decimal(10,2) comment '订单价格',  sku\_name string comment '商品名称',  sku\_tm\_id string comment '品牌id',  sku\_tm\_name string comment '品牌名称',  sku\_category3\_id string comment '商品三级品类id',  sku\_category2\_id string comment '商品二级品类id',  sku\_category1\_id string comment '商品一级品类id',  sku\_category3\_name string comment '商品三级品类名称',  sku\_category2\_name string comment '商品二级品类名称',  sku\_category1\_name string comment '商品一级品类名称',  spu\_id string comment '商品 spu',  sku\_num int comment '购买个数',  order\_count string comment '当日下单单数',  order\_amount string comment '当日下单金额'  ) COMMENT '用户购买商品明细表'  PARTITIONED BY ( `dt` string)  stored as parquet  location '/warehouse/imall\_one/dws/dws\_user\_sale\_detail\_daycount/'  tblproperties ("parquet.compression"="snappy"); |

向表中插入数据

|  |
| --- |
| with  tmp\_detail as  (  select  user\_id,  sku\_id,  sum(sku\_num) sku\_num,  count(\*) order\_count,  sum(order\_price \* sku\_name) order\_amount  from dwd\_order\_detail  where dt='2019-03-02'  group by user\_id,sku\_id  )  insert overwrite table dws\_sale\_detail\_daycount partition(dt='2019-03-02')  select  tmp\_detail.user\_id ,  tmp\_detail.sku\_id,  ui.gender,  round(months\_between('2019-03-02', ui.birthday)/12) age,  ui.user\_level,  price,  sku\_name,  tm\_id,  tm\_name,  category3\_id ,  category2\_id ,  category1\_id ,  category3\_name ,  category2\_name ,  category1\_name ,  spu\_id,  tmp\_detail.sku\_num,  tmp\_detail.order\_count,  tmp\_detail.order\_amount  from tmp\_detail  left join dwd\_user\_info ui  on tmp\_detail.user\_id = ui.id  left join dwd\_sku\_info si  on tmp\_detail.sku\_id = si.id  where  ui.dt='2019-03-02'  and  si.dt='2019-03-02'; |

转成报表

建表：

|  |
| --- |
| drop table ads\_sale\_tm\_category1\_stat\_mn;  create table ads\_sale\_tm\_category1\_stat\_mn  ( tm\_id string comment '品牌id ' ,  tm\_name string comment '品牌名称',  category1\_id string comment '1级品类id ',  category1\_name string comment '1级品类名称 ',  buycount bigint comment '购买人数',  buy\_twice\_last bigint comment '两次以上购买人数',  buy\_twice\_last\_ratio decimal(10,2) comment '单次复购率',  buy\_3times\_last bigint comment '三次以上购买率',  buy\_3times\_last\_ratio decimal(10,2) comment '多次复购率' ,  stat\_mn string comment '统计月份',  stat\_date string comment '统计日期'  ) COMMENT '复购率统计'  row format delimited fields terminated by '\t'  location '/warehouse/imall\_one/ads/ads\_sale\_tm\_category1\_stat\_mn/'; |

向表中插入数据：

|  |
| --- |
| with  tmp\_detail\_mn as  (  select  user\_id,  sku\_tm\_id,  sku\_tm\_name,  sku\_category1\_id,  sku\_category1\_name,  sum(sku\_num) sku\_num  from dws\_sale\_detail\_daycount  where  date\_format(dt,'yyyy-MM') = date\_format('2019-03-02','yyyy-MM')  group by user\_id, sku\_tm\_id,sku\_tm\_name,sku\_category1\_id,sku\_category1\_name  )  insert into table ads\_sale\_tm\_category1\_stat\_mn  select  sku\_tm\_id,sku\_tm\_name,sku\_category1\_id,sku\_category1\_name,  sum(if(sku\_num>0,1,0)), --买过的人  sum(if(sku\_num>=2,1,0)), --至少2次买过的人  sum(if(sku\_num>=2,1,0)) / sum(if(sku\_num>0,1,0)),  sum(if(sku\_num>=3,1,0)), --至少3次买过的人  sum(if(sku\_num>=3,1,0)) / sum(if(sku\_num>0,1,0)),  date\_format('2019-03-02','yyyy-MM') mn,  '2019-03-02' dt  from tmp\_detail\_mn  group by sku\_tm\_id,sku\_tm\_name,sku\_category1\_id,sku\_category1\_name; |

### 脚本

##### 脚本dws\_sale.sh

|  |
| --- |
| #!/bin/bash  APP=imall\_one  do\_date=$1  if [ ! -n "$do\_date" ] ;then  do\_date=`date -d "-1 day" +%F`  fi  sql="  use $APP;  with  tmp\_detail as  (  select  user\_id,  sku\_id,  sum(sku\_num) sku\_num,  count(\*) order\_count,  sum(order\_price \* sku\_name) order\_amount  from dwd\_order\_detail  where dt='${do\_date}'  group by user\_id,sku\_id  )  insert overwrite table dws\_sale\_detail\_daycount partition(dt='${do\_date}')  select  tmp\_detail.user\_id ,  tmp\_detail.sku\_id,  ui.gender,  round(months\_between('${do\_date}', ui.birthday)/12) age,  ui.user\_level,  price,  sku\_name,  tm\_id,  tm\_name,  category3\_id ,  category2\_id ,  category1\_id ,  category3\_name ,  category2\_name ,  spu\_id,  tmp\_detail.sku\_num,  tmp\_detail.order\_count,  tmp\_detail.order\_amount  from tmp\_detail  left join dwd\_user\_info ui  on tmp\_detail.user\_id = ui.id  left join dwd\_sku\_info si  on tmp\_detail.sku\_id = si.id  where  ui.dt='${do\_date}'  and  si.dt='${do\_date}';"  /opt/module/spark/bin/spark-sql -e "$sql" |

##### 脚本ads\_sale.sh

|  |
| --- |
| #!/bin/bash  APP=imall\_one  do\_date=$1  if [ ! -n "$do\_date" ] ;then  do\_date=`date -d "-1 day" +%F`  fi  sql="  use $APP;  with  tmp\_detail\_mn as  (  select  user\_id,  sku\_tm\_id,  sku\_tm\_name,  sku\_category1\_id,  sku\_category1\_name,  sum(sku\_num) sku\_num  from dws\_sale\_detail\_daycount  where  date\_format(dt,'yyyy-MM') = date\_format('${do\_date}','yyyy-MM')  group by user\_id, sku\_tm\_id,sku\_tm\_name,sku\_category1\_id,sku\_category1\_name  )  insert into table ads\_sale\_tm\_category1\_stat\_mn  select  sku\_tm\_id,sku\_tm\_name,sku\_category1\_id,sku\_category1\_name,  sum(if(sku\_num>0,1,0)), --买过的人  sum(if(sku\_num>=2,1,0)), --至少2次买过的人  sum(if(sku\_num>=2,1,0)) / sum(if(sku\_num>0,1,0)),  sum(if(sku\_num>=3,1,0)), --至少3次买过的人  sum(if(sku\_num>=3,1,0)) / sum(if(sku\_num>0,1,0)),  date\_format('${do\_date}','yyyy-MM') mn,  '${do\_date}' dt  from tmp\_detail\_mn  group by sku\_tm\_id,sku\_tm\_name,sku\_category1\_id,sku\_category1\_name;"  /opt/module/spark/bin/spark-sql -e "$sql" |

关于项目的一些技术的问题：

1. Nginx的负载均衡的策略：轮询，权重
2. taildir

记录偏移量，可以做到断点续传（也就是当flume宕机的时候，依然可以从上一次监控的文件开始监控文件）exec的升级版，

1. flume在项目中如何实现分流？

在拦截器中 event的header加入标记，在选择器中，根据标记分到不同的channel中 这个过程在source中完成。拦截器自定义，分流器可以使用默认的

4）为什么使用双层Flume？

缓冲的作用，从日志文件到Flume，这等于是有一个通道，有一定缓冲的作用。聚合

聚合的作用，日志服务器会比较多，那么就需要较多的flume-app，使用第二层flume-hdfs来聚合flume-app的数据，更加方面管理

1. 数仓是如何分层的，各层都做了哪些工作？

Ods：保存一份原始的数据，使用压缩率最大的压缩方式

Dwd：数据过滤，数据脱敏，数据降维，（数据降维的原则是：围绕事实表它的维度表只有一层，一层以上的都变为一层，把一个雪花模型改为星型模型 ）；还有就是行式存储变为列式存储（text->parquet）；压缩算法也改变了（gzip->snappy）如此提升压缩算法，提升压缩速度。

Dws：根据业务对数据进行轻度聚合，以用户，商品等制作为一个宽表

Ods：面向报表的应用，基于报表需求

1. Parquet存储，有什么好处？

过滤数据比较快，列式存储，与parquet+sanppy spark配合使用，压缩比更高。

1. 说一下你们如何划分表的同步策略

事实表： 事实表有周期性事实表，增加和变化，策略时候增加和变化量，可以直接做拉链表，

；事务性事实表，只会增加，所以直接做每日增量即可

维度表 ：一般维度表，数据量小，做每日全量

固定维度表：固定存一份。

实体表：会增加和变化，数据量中等 ，近期做每日全量，如果数据多了，太久的历史数据直接存档，最近的数据可以做每日全量，隔一段时间可以做拉链表

1. 说一下你们统计的留存率的计算过程：

8）1NF、2NF，3NF

1NF：不能有表中表；2NF：不能存在部分函数依赖；3NF：不能有传递函数依赖

9）雪花模型，星型模型

星型模型，一个事实表只能围绕一层维度表（多层维度压成单层，维度会有冗余）；

雪花模型，一个事实表可能有多层的维度表（查询的时候，需要进行join，会影响性能）

该项目使用星型模型

10）拉链表的生成过程：

获取到全量表，导入到拉链表，即为初始化拉链表；然后查出新增和变化；将新增和变化合并到拉链表放入到临时拉链表；使用临时拉链表替换老的拉链表

11）说说你知道的日期函数

date\_format date\_add next\_day（获取下一个星期几）datediff（日期差）month\_between（月差）last\_day（月末日期）current\_day（当前日期），from\_unix（date->ts）

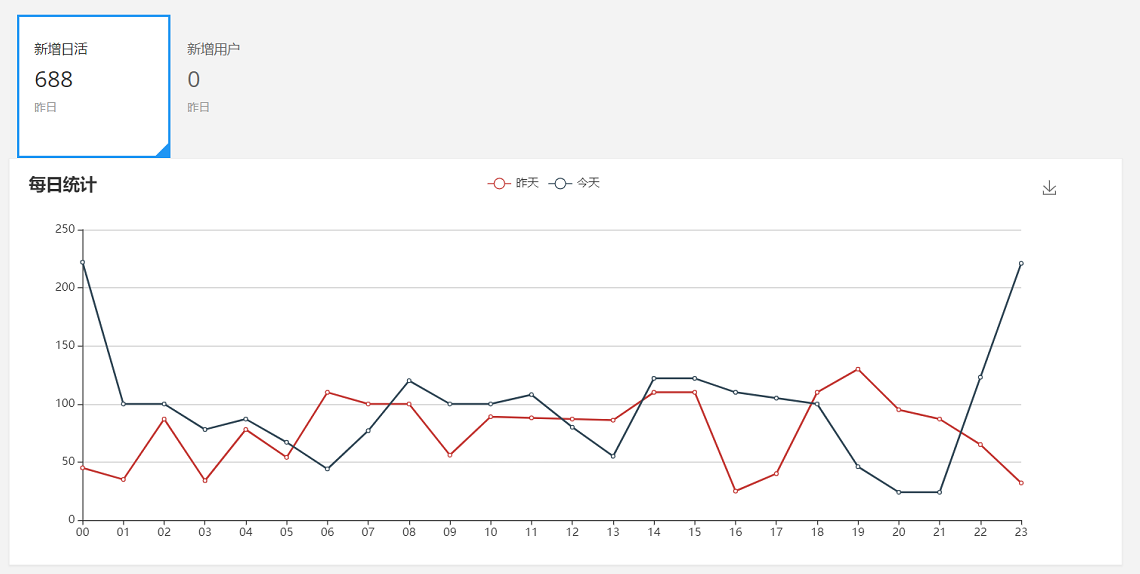
# 数据实时

## 项目需求

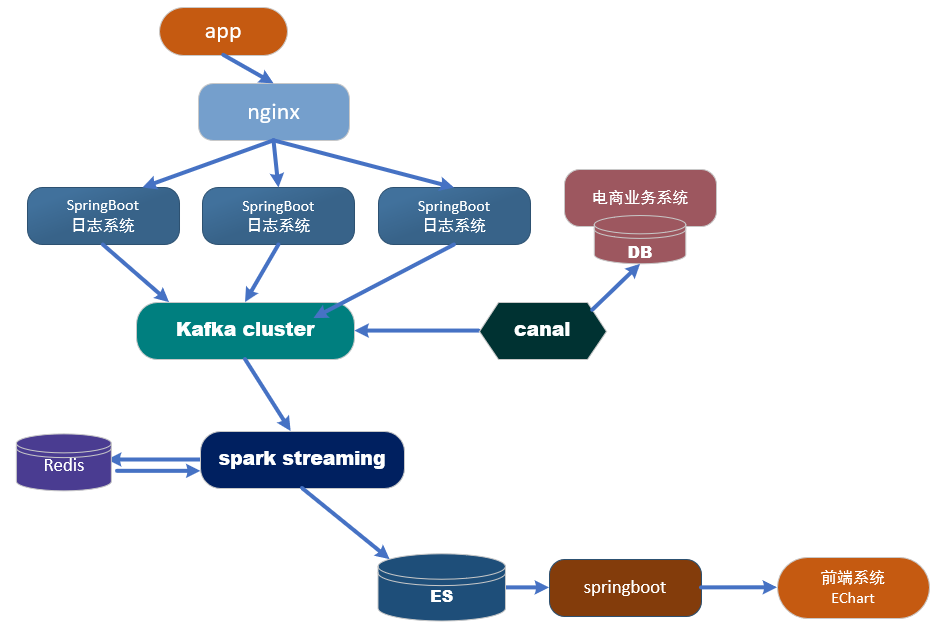
相对于离线的项目，实时的项目主要侧重于对当日数据的监控，获取的数据也要求有一定的时效性。主要有下面四个需求：

1. 当日活跃用户和分时趋势图，昨日对比图
2. 当日新增用户和分时趋势图，昨日对比图
3. 当日交易额和分时趋势图，昨日对比图
4. 当日订单数和分时趋势图，昨日对比图

实例如下：



## 项目架构



### 和日志相关的实时分析

用户从APP客户端产生log日志，通过Nginx到达日志服务器，然后由日志服务器发送到Kafka中进行临时存储，然后spark streaming作为kafka的消费者来消费kafka中的数据，并且封装为DStream。

当数据进入了spark streaming之后，要先过滤（依据redis中的数据集），然后再将数据存储到Redis中，最后将数据存储到ES中，由于ES支持一定的查询，所以我们的策略是将需求的一些明细存储到ES中，接下来的springboot的作用是用来响应用户的查询请求，最后返回到前端。

### 和业务相关的实时分析

对于电商系统来说，用户的购买，下单，付款行为会被落盘关系型数据库，比如MySQL。这里需要监控MySQL中对应的表的数据的变化，然后将这些变化了的字段通过canal捕获，然后将这些数据发送到Kafka中，接下来几乎和日志相关的实时分析是一致的。

## 日志采集模块：

在离线项目的基础上，增加将数据发往Kafka的代码。并按照日志的类型，event和startup来给将数据发送到不同topic中：

### 与Kafka建立连接

Logger日志服务器模块，如果想要将数据发送到Kafka，就必须要和Kafka建立连接，这一点，**SpringBoot已经做好了和Kafka的整合**，只需要引入

|  |  |
| --- | --- |
| <dependency>  <groupId>org.springframework.kafka</groupId>  <artifactId>spring-kafka</artifactId>  <version>1.3.2.RELEASE</version> </dependency>  然后在代码中，直接使用注入的方式，获取一个**KafkaTemplate**。使用该类的对象，可以直接往Kafka中发送数据了。  同时在application.properties的配置文件中，添加当前的程序要连接的Kafka地址及Kafka消息的编解码方式：  #============== kafka ===================  # 指定kafka 代理地址，可以多个 spring.kafka.bootstrap-servers=hadoop101:9092   # 指定消息key和消息体的编解码方式 spring.kafka.producer.key-serializer=org.apache.kafka.common.serialization.StringSerializer spring.kafka.producer.value-serializer=org.apache.kafka.common.serialization.StringSerializer  可见，这里的Logger就相当于是Kafka的生产者，或者是连接Kafka的客户端。最后，   1. 开启一个Kafka的消费者 2. 开启Logger采集的服务器 3. Mocker模拟日志的生成模块开启   在Shell中开启一个Kafka消费者的命令：   |  | | --- | | /opt/module/kafka/bin/kafka-console-consumer.sh --bootstrap-server hadoop101:9092,hadoop102:9092,hadoop103:9092 --topic IMALL\_STARTUP --from-beginning | |

|  |  |  |
| --- | --- | --- |
| 类型 | 常量名 | Kafka中存储的主题名 |
| Startup | KAFKA\_TOPIC\_STARTUP | IMALL\_STARTUP |
| Event | KAFKA\_TOPIC\_EVENT | IMALL\_EVENT |

### 代码

Logger日志收集的模块，添加发送日志到kafka的代码：

|  |
| --- |
| package com.isea.imall.dw.imall.dw.logger.controller;  import com.alibaba.fastjson.JSON; import com.alibaba.fastjson.JSONObject; import com.isea.imall.dw.common.constant.ImallConstant; import org.slf4j.LoggerFactory; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.kafka.core.KafkaTemplate; import org.springframework.web.bind.annotation.PostMapping; import org.springframework.web.bind.annotation.RequestParam; import org.springframework.web.bind.annotation.RestController;  @RestController // = Controller+ResponseBody public class LoggerController {   @Autowired  KafkaTemplate kafkaTemplate;   private static final org.slf4j.Logger *logger* = LoggerFactory.*getLogger*(LoggerController.class);   @PostMapping("isea\_log") // 映射请求的URL  // 会认为success 不是一个页面，而仅仅是个结果，就是一个字符串  public String log(@RequestParam("log") String logJson) { // 请求的参数,注入到一个变量中，log必须是请求的时候，=左边的字符串  // 这里是处理具体的业务->使用logger4j来专门的管理日志，将日志输出到目的地。   System.*out*.println(logJson); // 在后端打印出请求传入的内容   JSONObject jsonObject = JSON.*parseObject*(logJson);  jsonObject.put("ts", System.*currentTimeMillis*());   //离线数据发送到Kafka中  sendToKafka(jsonObject);   // 实时的数据发送到Logger日志服务器中  *logger*.info(jsonObject.toJSONString());   return "success..."; // 返回前端，或者是请求者的信息。多数时候这里是返回一个页面给前端。  }   private void sendToKafka(JSONObject jsonObject){  // 根据logJSON中type的类型，来分到不同的topic中  if (jsonObject.getString("type").equals("startup")){  kafkaTemplate.send(ImallConstant.*KAFKA\_TOPIC\_STARTUP*,jsonObject.toJSONString());  }else {  kafkaTemplate.send(ImallConstant.*KAFKA\_TOPIC\_EVENT*,jsonObject.toJSONString());  }  } } |

## 数据库数据采集模块

思考一个问题：

MySQL数据库中表的数据如果变化了，而在表中又没有维护相应的字段来表示修改操作，我们也就无法获知该字段是否被修改过，我们的需求3和需求4中要求找到当日的交易额，交易额依赖于订单中的是否支付字段，如果从未支付到已支付，MySQL需要一个update操作，我们如何能获知MySQL执行了这个操作呢？在产生新的订单的时候，我们是如何获知insert的呢？

这里需要使用一个新的技术：**canal**，（运河的意思）

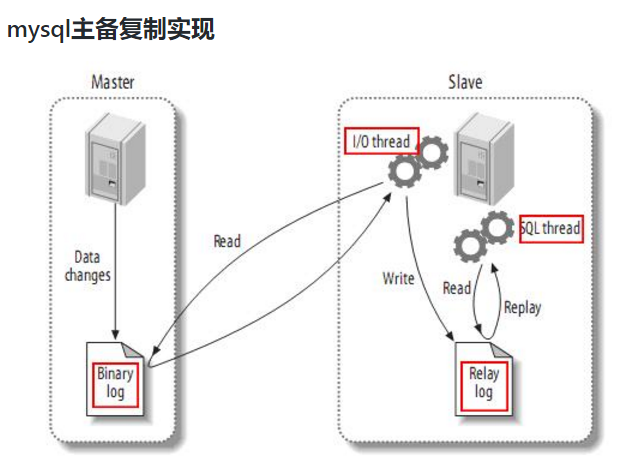
Canal的作用就是同步MySQL，具体的使用场景是做拉链表的时候，获取变化量，另外就是当使用Redis做前置层的时候，通过canal来更新Redis中的数据

### Canal的原理

Canal的工作依赖于MySQL的主从复制：

#### MySQL的主从复制

单台的MySQL无法实现高可用和安全性，也不能实现高并发，所以需要MySQL的集群，一般来说，MySQL是通过主从复制的方式来同步数据，在通过读写分离来提升并发访问的能力。



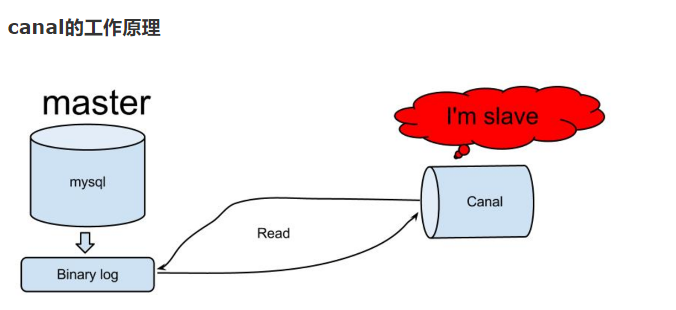
想要实现MySQL的主从复制，需要在MySQL的配置文件中，设置MySQL的binlog开启，MySQL数据库中的数据每一次被更改之后的信息放置在binarylog中，slave会读去master的这个文件，然后写到自己的relay log文件中，然后在按照**relay**（中继的意思） log文件来实现数据的同步。

#### Binlog的模式：

1. STATEMENT ，binlog会记录每一写操作的语句，优点是比较节省空间，缺点是可能会造成数据不一致，比如出现uuid ，update \*\* date = now
2. ROW，binlog会记录下每一次操作之后，每行记录的变化。缺点是比较占空间
3. MIXED，STATEMENT的升级，当出现了UDF，或UUID等的时候，会变成row。

#### Canal的模式：

Canal的工作原理把自己伪装成slave，去读取MySQL主机的binlog。此时的canal是作为一个服务器，服务器中保存着MySQL的数据，需要实现一个客户端来获取canal中的数据。



### Canal代码模块

本模块的主要作用就是获取canal服务器中的数据，然后发送到Kafka中，

|  |
| --- |
| package com.isea.imall.dw.canal.client;  import com.alibaba.fastjson.JSONObject; import com.alibaba.otter.canal.client.CanalConnector; import com.alibaba.otter.canal.client.CanalConnectors; import com.alibaba.otter.canal.protocol.CanalEntry; import com.alibaba.otter.canal.protocol.Message; import com.google.common.base.CaseFormat; import com.google.protobuf.InvalidProtocolBufferException; import com.isea.imall.dw.common.constant.ImallConstant; import com.isea.imall.dw.common.util.MyKafkaSender;  import java.net.InetSocketAddress; import java.util.List;  public class CanalClient {   // 该主函数开启了之后，能够一直监听canal服务器监听MySQL中数据库的变化，这是canal的客户端  public static void main(String[] args) {  // canal客户端连接cannal服务器。  CanalConnector canalConnector = CanalConnectors.*newSingleConnector*(new InetSocketAddress("hadoop101", 11111), "example", "", "");    while (true) {  // 建立canal服务器  canalConnector.connect();   // 订阅数据库和表  canalConnector.subscribe("imall\_mysql\_one.order\_info");   // 获取发生的sql，得到message,我们这里一个message里面抓取100条SQL（但是不一定能够抓取到100条）  Message message = canalConnector.get(100);  System.*out*.println("获取到" + message.getEntries().size() + "个SQL");   if (message.getEntries().size() == 0) {  System.*out*.println("imall\_mysql\_one中的order\_info表没有写操作");  try {  Thread.*sleep*(5000);  } catch (InterruptedException e) {  e.printStackTrace();  }  } else {  // 展开所有的SQL  for (CanalEntry.Entry entry : message.getEntries()) {  // 过滤掉对数据不造成影响的SQL，是rowdata类型的才可以通过  if (entry.getEntryType() == CanalEntry.EntryType.*ROWDATA*) {  try {  // 对entry反序列化，得到rowchange  CanalEntry.RowChange rowChange = CanalEntry.RowChange.*parseFrom*(entry.getStoreValue());  // 从rowchange里面得到变化的行的集合  List<CanalEntry.RowData> rowDatasList = rowChange.getRowDatasList();   // 得到操作的类型：  CanalEntry.EventType eventType = rowChange.getEventType();  // 得到表名  String tableName = entry.getHeader().getTableName();   // 将对某个表做个某个操作，受影响的行，做处理（最终发送到kafka中）  *handle*(tableName, eventType, rowDatasList);   } catch (InvalidProtocolBufferException e) {  e.printStackTrace();  }  }  }  }  }  }   private static void handle(String tableName, CanalEntry.EventType eventType, List<CanalEntry.RowData> rowDatasList) {  if ("order\_info".equals(tableName) && eventType == CanalEntry.EventType.*INSERT* && rowDatasList.size() > 0) {  for (CanalEntry.RowData rowData : rowDatasList) { // 展开行集  List<CanalEntry.Column> afterColumnsList = rowData.getAfterColumnsList(); // 得到变化后列的集合  JSONObject jsonObject = new JSONObject();  // 展开列集  for (CanalEntry.Column column : afterColumnsList) {  System.*out*.println(column.getName() + ";" + column.getValue());   // 做成json的格式  String colNameCamel = CaseFormat.*LOWER\_UNDERSCORE*.to(CaseFormat.*LOWER\_CAMEL*, column.getName());  jsonObject.put(colNameCamel, column.getValue());  }  // 将封装好的json串发送到Kafka中，发过去的是一行数据，  MyKafkaSender.*send*(ImallConstant.*KAFKA\_TOPIC\_ORDER*, jsonObject.toJSONString());  }  }  } }  **MyKafkaSender代码**  package com.isea.imall.dw.common.util;  import org.apache.kafka.clients.producer.KafkaProducer; import org.apache.kafka.clients.producer.ProducerRecord;  import java.util.Properties;  public class MyKafkaSender {  private static KafkaProducer<String, String> *kafkaProducer* = null;   private static KafkaProducer<String, String> createKafkaProducer() {  Properties properties = new Properties();  properties.put("bootstrap.servers", "hadoop101:9092,hadoop102:9092,hadoop103:9092");  properties.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");  properties.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");   KafkaProducer<String, String> producer = null;   try {  producer = new KafkaProducer<String, String>(properties);  } catch (Exception e) {  e.printStackTrace();  }  return producer;  }   public static void send(String topic, String msg) {  if (*kafkaProducer* == null) {  *kafkaProducer* = *createKafkaProducer*();  }  *kafkaProducer*.send(new ProducerRecord<String, String>(topic, msg));  } } |

## 需求1：DAU：

当日活跃用户分时趋势图，昨日对比图；这里解决问题的答题的逻辑是，记录下每个机器当日第一次登录系统的明细，具体的统计工作，交给到ES去做。分时对比图只需要按照小时分组即可。

### 思路

1. 消费Kafka中的数据
2. 过滤掉当日已经计算过的日活设备
3. 将每批次新增的当日日活明细保存到ES中
4. 从ES中查出数据，发布成数据接口

### 为什么是ES？

这里的为什么不将数据在spark做统计，然后存储到MySQL中，从存储的**速度和存储**的容量来解释。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | redis | mysql | elasticsearch | hbase | hadoop/hive |
| 容量/容量扩展 | 低 | 中 | 较大 | 海量 | 海量 |
| 查询时效性 | 极高 | 中等 | 较高 | 中等 | 低 |
| 查询灵活性 | 较差 k-v模式 | 非常好，支持sql | 较好，关联查询较弱，但是可以全文检索，DSL语言可以处理过滤、匹配、排序、聚合等各种操作 | 较差，主要靠rowkey,  scan的话性能不行，或者建立二级索引 | 非常好，支持sql |
| 写入速度 | 极快 | 中等 | 较快 | 较快 | 慢 |
| 一致性、事务 | 弱 | **强** | 弱 | 弱 | 弱 |

mysql虽然交互性好，但是容量扩展性有限。hbase虽然能够支持海量数据，但是查询的灵活度不足。所以ES在容量及交互性上达到一个非常不错的平衡，**而且还能支持全文检索**。

### 保存到ES

在这里，我们将首次登陆得用户保存到ES中，在MySQL中保存数据的时候，需要提前建好库，表，约束等信息，ES在导入数据之前，也需要定义好字段。同时还需要指定**是否需要索引或者是否支持分词**等。

#### 设计ES索引结构

发送到ES中的数据格式是一个个对象：

|  |
| --- |
| **case class** StartupLog(  mid: String,  appid: String,  area: String,  os: String,  ch: String,  logType: String,  vs: String,  **var** logDate: String,  **var** logHour: String,  **var** logHourMinute: String,  **var** ts: Long  ) { } |

我们将所有的字段，分成三类：

1. 需要索引，也需要分词：**type：“text”**
2. 需要索引，但是不需要分词 ：以上的字段都属于这类 **type：“keyword”**
3. 既不需要索引，也不需要分词：**index：false**

|  |
| --- |
| PUT imall\_one\_dau  {  "mappings": {  "\_doc":{  "properties":{  **"mid"**:{  "type":"keyword"  },  "uid":{  "type":"keyword"  },  "area":{  "type":"keyword"  },  "os":{  "type":"keyword"  },  "ch":{  "type":"keyword"  },  "vs":{  "type":"keyword"  },  "logDate":{  "type":"keyword"  },  "logHour":{  "type":"keyword"  },  **"logHourMinute"**:{  "type":"keyword"  },  "ts":{  "type":"long"  }  }  }  }  } |

#### 保存到ES

##### ES客户端

Jest，这种客户端可以直接使用**DSL**语句拼成字符串，直接传给服务端，然后返回Json字符串。并且**和springboot进行了集成**。

### 代码imall-realtime模块：

#### PropertiesUtil代码：

主要用来加载配置文件：

|  |
| --- |
| **package** com.isea.imall.dw.realtime.util  **import** java.io.InputStreamReader **import** java.util.Properties **object** PropertiesUtil {  **def** load(propertiesName : String): Properties = {  **val** properties = **new** Properties()  properties.load(**new** InputStreamReader(Thread.*currentThread*().getContextClassLoader  .getResourceAsStream(propertiesName),"UTF-8"))  properties  } } |

#### MyKafkaUtil

和Kafka建立连接，并消费Kafka中的数据，封装成DStream

|  |
| --- |
| **package** com.isea.imall.dw.realtime.util  **import** org.apache.kafka.clients.consumer.ConsumerRecord **import** org.apache.kafka.common.serialization.StringDeserializer **import** org.apache.spark.streaming.StreamingContext **import** org.apache.spark.streaming.dstream.InputDStream **import** org.apache.spark.streaming.kafka010.{ConsumerStrategies, KafkaUtils, LocationStrategies}  **object** MyKafkaUtil {   **val** *prop* = PropertiesUtil.*load*("config.properties")  **val** *broker\_list* = *prop*.getProperty("kafka.broker.list")   // kafka消费者配置  **val** *kafkaParam* = *Map*(  "bootstrap.servers" -> *broker\_list*,//用于初始化链接到集群的地址  "key.deserializer" -> *classOf*[StringDeserializer],  "value.deserializer" -> *classOf*[StringDeserializer],  //用于标识这个消费者属于哪个消费团体  "group.id" -> "commerce-consumer-group",  //如果没有初始化偏移量或者当前的偏移量不存在任何服务器上，可以使用这个配置属性  //可以使用这个配置，latest自动重置偏移量为最新的偏移量  "auto.offset.reset" -> "latest",  //如果是true，则这个消费者的偏移量会在后台自动提交,但是kafka宕机容易丢失数据  //如果是false，会需要手动维护kafka偏移量  "enable.auto.commit" -> (**true**: java.lang.Boolean)  )   // 创建DStream，返回接收到的输入数据  // LocationStrategies：根据给定的主题和集群地址创建consumer  // LocationStrategies.PreferConsistent：持续的在所有Executor之间分配分区  // ConsumerStrategies：选择如何在Driver和Executor上创建和配置Kafka Consumer  // ConsumerStrategies.Subscribe：订阅一系列主题   // 使用sparkStreaming来消费Kafka中的数据，组装成DStream，也即SparkStreaming获取Kafka中的数据  **def** getKafkaStream(topic: String,ssc:StreamingContext): InputDStream[ConsumerRecord[String,String]]={  **val** dStream = KafkaUtils.*createDirectStream*[String,String](ssc, LocationStrategies.*PreferConsistent*,ConsumerStrategies.*Subscribe*[String,String](*Array*(topic),*kafkaParam*))  dStream  } } |

#### Config.properties

|  |
| --- |
| # Kafka配置  kafka.broker.list=hadoop101:9092,hadoop102:9092,hadoop103:9092  # Redis配置 redis.host=hadoop103 redis.port=6379 |

#### RedisUtil

|  |
| --- |
| **package** com.isea.imall.dw.realtime.util  **import** redis.clients.jedis.{Jedis, JedisPool, JedisPoolConfig}  **object** RedisUtil {   **var** *jedisPool*:JedisPool=**null   def** getJedisClient: Jedis = {  **if**(*jedisPool*==**null**){  *println*("开辟一个连接池")  **val** prop = PropertiesUtil.*load*("config.properties")  **val** host = prop.getProperty("redis.host")  **val** port = prop.getProperty("redis.port").toInt   **val** jedisPoolConfig = **new** JedisPoolConfig()  jedisPoolConfig.setMaxTotal(100) //最大连接数  jedisPoolConfig.setMaxIdle(20) //最大空闲  jedisPoolConfig.setMinIdle(20) //最小空闲  jedisPoolConfig.setBlockWhenExhausted(**true**) //忙碌时是否等待  jedisPoolConfig.setMaxWaitMillis(500)//忙碌时等待时长 毫秒  jedisPoolConfig.setTestOnBorrow(**true**) //每次获得连接的进行测试   *jedisPool* = **new** JedisPool(jedisPoolConfig,host,port)  }  *println*(s"jedisPool.getNumActive = **$**{*jedisPool*.getNumActive}")  *println*("获得一个连接")  *jedisPool*.getResource  } } |

#### StartupLog

Case class：的主要作用是将，APP🡪Logger日志服务器🡪Kafka🡪SparkStreaming的数据结构化为JavaBean

|  |
| --- |
| **package** com.isea.imall.dw.realtime.bean  // 写在构造函数中的参数，如果没有加任何的东西，就认为是val的，仍是私有的，但是可以读， // 如果被声明成了var，那么还是私有的属性，但是可读可写 **case class** StartupLog(  mid: String,  appid: String,  area: String,  os: String,  ch: String,  logType: String,  vs: String,  **var** logDate: String,  **var** logHour: String,  **var** logHourMinute: String,  **var** ts: Long  ) { } |

#### MyESUtil

|  |
| --- |
| **package** com.isea.imall.dw.common.util  **import** java.util.Objects  **import** io.searchbox.client.config.HttpClientConfig **import** io.searchbox.client.{JestClient, JestClientFactory} **import** io.searchbox.core.{Bulk, Index}  **object** MyEsUtil {  **private val** *ES\_HOST* = "http://hadoop101"  **private val** *ES\_HTTP\_PORT* = 9200  **private var** *factory* : JestClientFactory = **null   def** getClient:JestClient = {  **if** (*factory* == **null**) *build*()  *factory*.getObject  }   */\*\*  \* 关闭客户端  \** ***@param client*** *\** ***@return*** *\*/* **def** close(client: JestClient) ={  **if** (!Objects.*isNull*(client)) **try** client.hashCode()  **catch**{  **case** e :Exception =>  e.printStackTrace()  }  }   */\*\*  \* 建立连接  \*/* **private def** build() ={  *factory* = **new** JestClientFactory  *factory*.setHttpClientConfig(**new** HttpClientConfig.Builder(*ES\_HOST* + ":" + // 和哪台机器的哪个端口建立连接  *ES\_HTTP\_PORT*).multiThreaded(**true**).maxTotalConnection(20) // 连接总数  .connTimeout(10000).readTimeout(10000).build()) // 连接建立的延迟，和读取数据的延迟  }   **def** insertBuilk(indexName : String,docLit: List[Any]) ={  **val** jest: JestClient = *getClient* **val** bulkBuilder = **new** Bulk.Builder   bulkBuilder.defaultIndex(indexName).defaultType("\_doc")   *println*(docLit.mkString("\n"))   **for** (doc <- docLit) {  **val** index: Index = **new** Index.Builder(doc).build()  bulkBuilder.addAction(index)  }   jest.execute(bulkBuilder.build())  *close*(jest)  }    **def** main(args: Array[String]): Unit = {  **val** jest: JestClient = *getClient* **val** doc : String = "{\n \"name\":\"isea\",\n \"age\":22\n}"  **val** index: Index = **new** Index.Builder(doc).index("test").`type`("\_doc").build()  jest.execute(index)  } } |

#### StartupApp

在该部分的代码中，完成主要的功能模块：

1. 消费Kafka中的数据，以DStream的形式
2. 将log日志的信息，转为JavaBean的格式
3. 利用广播变量，**广播Redis中的数据**，过滤掉已经在Redis存在的数据（也就是不是今天第一次登陆的用户）使用广播变量的时候使用**transform**算子。
4. 将今天首次登陆的用户详情保存到Redis中。
5. 将今天首次访问的用户详情保存到ES中。

|  |
| --- |
| **package** com.isea.imall.dw.realtime.app  **import** java.text.SimpleDateFormat **import** java.util **import** java.util.Date  **import** com.alibaba.fastjson.JSON **import** com.isea.imall.dw.common.constant.ImallConstant **import** com.isea.imall.dw.common.util.MyEsUtil **import** com.isea.imall.dw.realtime.bean.StartupLog **import** com.isea.imall.dw.realtime.util.{MyKafkaUtil, RedisUtil} **import** org.apache.kafka.clients.consumer.ConsumerRecord **import** org.apache.spark.broadcast.Broadcast **import** org.apache.spark.rdd.RDD **import** org.apache.spark.streaming.dstream.{DStream, InputDStream} **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.spark.{SparkConf, SparkContext} **import** redis.clients.jedis.Jedis  **import** scala.collection.mutable.ListBuffer  **object** StartUpApp {  **def** main(args: Array[String]): Unit = {  **val** sparkConf: SparkConf = **new** SparkConf().setMaster("local[\*]").setAppName("imall-realtime-startup")  **val** sc = **new** SparkContext(sparkConf)   **val** ssc = **new** StreamingContext(sc, *Seconds*(5))  // 这里是key是kafka消息的消息的key，value是kafka的message的值  **val** recordDStream: InputDStream[ConsumerRecord[String, String]] = MyKafkaUtil.*getKafkaStream*(ImallConstant.*KAFKA\_TOPIC\_STARTUP*, ssc)   /\* recordDStream.map(\_.value()).foreachRDD( rdd =>   println(rdd.collect().mkString("\n"))  )\*/   // 转成Javabean 结构化数据  **val** startupLogDStream: DStream[StartupLog] = recordDStream.map(\_.value()).map { jsonString =>  **val** startupLog: StartupLog = JSON.*parseObject*(jsonString, *classOf*[StartupLog])   **val** dateFormat = **new** SimpleDateFormat("yyyy-MM-dd HH:mm")  **val** logDateTime: String = dateFormat.format(**new** Date(startupLog.ts))  startupLog.logDate = logDateTime.split(" ")(0)  startupLog.logHourMinute = logDateTime.split(" ")(1)  startupLog.logHour = logDateTime.split(" ")(1).split(":")(0)  startupLog  }   // 过滤今天登陆过的用户，利用广播变量，将今天redis中的登录用户广播到Executor中  **val** filterStarupLogDStream: DStream[StartupLog] = startupLogDStream.**transform** { rdd =>   *println*("过滤前：" + rdd.count())   // Driver端  **val** dateFormat: SimpleDateFormat = **new** SimpleDateFormat("yyyy-MM-dd")  **val** today: String = dateFormat.format(**new** Date())   **val** jedisClient: Jedis = RedisUtil.*getJedisClient*  *// 获取当天日期下redis中所有的活跃用户*  **val** dauSet: util.Set[String] = jedisClient.**smembers**("dau:" + today)  **val** dauSetBC: Broadcast[util.Set[String]] = sc.broadcast(dauSet)   // Executor端  **val** filteredRDD: RDD[StartupLog] = rdd.filter { startupLog =>  **if** (dauSetBC.value != **null** && dauSetBC.value.contains(startupLog.mid)){  **false** } **else**{  **true** }  }  *println*("过滤后：" + filteredRDD.count())  filteredRDD  }   /\*  使用的数据类型：每天都会有一个key，但是对应着多个value  使用set，key："dau"+date daily active user value:mid  \*/  // 把今日访问的用户保存到redis中   filterStarupLogDStream.foreachRDD(rdd =>  //driver  rdd.foreachPartition{startupLogItr =>  //executor  **val** jedisClient: Jedis = RedisUtil.*getJedisClient* **val** listBuffer = **new** ListBuffer[Any]  **for** (startupLog <- startupLogItr) {  **val** dauKey: String = "dau:" + startupLog.logDate  jedisClient.sadd(dauKey,startupLog.mid)  listBuffer += startupLog  }  jedisClient.close()   // 将数据存储到ES中  MyEsUtil.*insertBuilk*(ImallConstant.*ES\_INDEX\_DAU*,listBuffer.toList)  }  )   // 目标：保存到redis中，  ssc.start()  ssc.awaitTermination()  } } |

### 数据发布接口imall-publisher：

**该模块为spring模块**，

#### 访问路径与返回的数据格式

请求：http://publisher:8070/realtime-total?**date**=**2019-03-05**

返回：

[{"id":"dau","name":"新增日活","value":1200},{"id":"new\_mid","name":"新增用户","value":233}]

分时统计：

http://publisher:8070/realtime-hour?i**d=dau&&date=2019-03-05**

返回

{"yesterday":{"11":383,"12":123,"17":88,"19":200 },"today":{"12":38,"13":1233,"17":123,"19":688 }}

第一个是条件查询，只要date = 2019-03-05 这天的；

第二个需要按照小时进行聚合，如此一来就能够得到每个小时，活跃人数。

#### DSL查询语句

|  |
| --- |
| GET imall\_one\_dau/**\_search**  {  "query":{  "bool":{  "filter":{  "match":{  "logDate":"**2019-03-05" // logDate是要过滤的字段**  }  }  }  }    , "aggs": { // 使用聚合  "groupBy\_logHour": { // 本次聚合叫什么名字，自定义  "terms": { // terms相当于 mysql中的count(\*)  "field": "**logHour.keyword**",// 此处不进行分词 ,要进行聚合的字段，常用的还是avg，sum等  "size": 24 // 表示你要分成多少组，默认是10组。  }  }  }  } |

match\_al这是匹配，匹配的意思是查询一些涉及到分词的，如果查询小米手机，只要键入“小米”，或者是“手机”，只要是匹配上一个词，就能查询出来。

Bool -> filter 过滤是指：使用数字，或者是id，必须要求是一模一样。

Bool是混合查询，query是查询，混合查询的意思是，当组合里面有过滤和匹配的时候，可以都放置在一个bool下面

匹配写must

ES中只是支持。一次查询，只是支持一次分组。

#### 代码

##### Service层

|  |
| --- |
| package com.isea.dw.imall.publisher.service.impl;  import com.isea.dw.imall.publisher.service.PublishService; import com.isea.imall.dw.common.constant.ImallConstant; import io.searchbox.client.JestClient; import io.searchbox.core.Search; import io.searchbox.core.SearchResult; import org.elasticsearch.index.query.BoolQueryBuilder; import org.elasticsearch.index.query.MatchQueryBuilder; import org.elasticsearch.search.builder.SearchSourceBuilder; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.stereotype.Service;  import java.io.IOException;  @Service public class PublishServiceImpl implements PublishService {    @Autowired  JestClient jestClient; // 访问的是在application.properties配置文件中的es服务器   @Override  public int getDauTotalByTime(String date) {   int total = 0;   // 封装DSL  SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();  BoolQueryBuilder boolQueryBuilder = new BoolQueryBuilder();  MatchQueryBuilder matchQueryBuilder = new MatchQueryBuilder("logDate", date);   boolQueryBuilder.filter(matchQueryBuilder);  searchSourceBuilder.query(boolQueryBuilder);   Search search = new Search.Builder(searchSourceBuilder.toString()).addIndex(ImallConstant.*ES\_INDEX\_DAU*).addType(ImallConstant.*ES\_DEFAULT\_TYPE*).build();   try {  SearchResult searchResult = jestClient.execute(search);  total = searchResult.getTotal();  } catch (IOException e) {  e.printStackTrace();  }   System.*out*.println("total:" + total);  return total;  } } |

##### Controller层

|  |
| --- |
| package com.isea.dw.imall.publisher.controller;  import com.alibaba.fastjson.JSON; import com.isea.dw.imall.publisher.service.PublishService; import org.springframework.beans.factory.annotation.Autowired; import org.springframework.web.bind.annotation.GetMapping; import org.springframework.web.bind.annotation.RequestParam; import org.springframework.web.bind.annotation.RestController;  import java.util.ArrayList; import java.util.HashMap; import java.util.List;  @RestController public class PublishController {   @Autowired  PublishService publishService;   @GetMapping("realtime-total")  public String getRealTimeTotal(@RequestParam("date") String date){   int dauTotalByTime = publishService.getDauTotalByTime(date);   List<Object> list = new ArrayList<>();   HashMap<Object, Object> dauMap = new HashMap<>();  HashMap<Object, Object> newMidMap = new HashMap<>();   dauMap.put("id","dau");  dauMap.put("name","新增日活"); // 这里其实是截止到当时的日活用户  dauMap.put("value",dauTotalByTime);   newMidMap.put("id","new\_mid");  newMidMap.put("name","新增用户");  newMidMap.put("value",1200);   list.add(dauMap);  list.add(newMidMap);  return JSON.*toJSONString*(list);  } } |

##### Application.properties

需要和ES建立连接

|  |
| --- |
| server.port=8070 spring.elasticsearch.jest.uris=http://hadoop101:9200,http://hadoop102:9200 |

如此一来，我们就做好了数据接口，用户就可以直接访问这些接口，完成数据的获取了。

## 需求2：

**当日交易额，和分时趋势图，昨日对比图**

这里我们认为，只要用户下单了，即计算到交易额中。

因为这里有分时的统计，所以我们需要将小时拆解出来。ES在存储的时候，一定要预计一下在查询的时候，**打算使用什么分组，打算使用什么聚合**。

这里是按照小时来分组的。此外一些敏感信息，还要脱敏（比如电话去掉中间的几个数字）

在将数据插入到ES之前，要建立好ES的索引，要确定每个字段是否需要建立索引，是否需要分词，这里有一个原则就是所有脱敏的信息都不需要建索引或者分词了，数字，ID，等不需要分词，只要用户有可能做查询，那么就需要做索引。

这里需要启动的有：

Canal服务器启动起来，canal的客户端启动起来

MySQL新增订单

Kafka采集数据

NewOrderApp启动处理Kafka数据

ES启动起来收集SparkStreaming的数据

在ES里面创建索引的语句：

|  |
| --- |
| PUT imall\_one\_new\_order  {  "mappings" : {  "\_doc" : {  "properties" : {  "provinceId" : {  "type" : "keyword"  },  "consignee" : {  "type" : "keyword",  "index":false  },  "consigneeTel" : {  "type" : "keyword",  "index":false  },  "createDate" : {  "type" : "keyword"  },  "createHour" : {  "type" : "keyword"  },  "createHourMinute" : {  "type" : "keyword"  },  "createTime" : {  "type" : "keyword"  },  "deliveryAddress" : {  "type" : "keyword"  },  "expireTime" : {  "type" : "keyword"  },  "id" : {  "type" : "keyword"  },  "imgUrl" : {  "type" : "keyword",  "index":false  },  "operateTime" : {  "type" : "keyword"  },  "orderComment" : {  "type" : "keyword",  "index":false  },  "orderStatus" : {  "type" : "keyword"  },  "outTradeNo" : {  "type" : "keyword",  "index":false  },  "parentOrderId" : {  "type" : "keyword"  },  "paymentWay" : {  "type" : "keyword"  },  "totalAmount" : {  "type" : "double"  },  "trackingNo" : {  "type" : "keyword"  },  "tradeBody" : {  "type" : "keyword",  "index":false  },  "userId" : {  "type" : "keyword"  }  }  }  }  } |

### 代码

### 最后一个需求（可以做全文的匹配搜索）

根据条件分析用的购买行为：在数据仓库中存储着大量的明细数据信息，但是hadoop存储的数据仓库计算必须要经过MR，，及时性非常的差，为了方便吉时查询，希望在客户端完成按照关键词来查询这些信息。

在我们的数据仓库中，是存储的有宽表的，我们需要将每天新增购买信息的数据导入到ES中。这里主要针对订单表，而且还要支持关键词搜索。比如商品字段，使用一个词和商品名称做全文匹配。所以数仓中的宽表，除了要计算复购率之外，在计算之前还要发往ES中一份。然后ES提供一个数据查询的接口，用户可以用来搜索关键词匹配的订单信息。

1. 数仓准备宽表
2. 使用spark写程序，做一个jar放置到服务器数仓中，将宽表导入到ES中
3. 提供一个ES查询接口