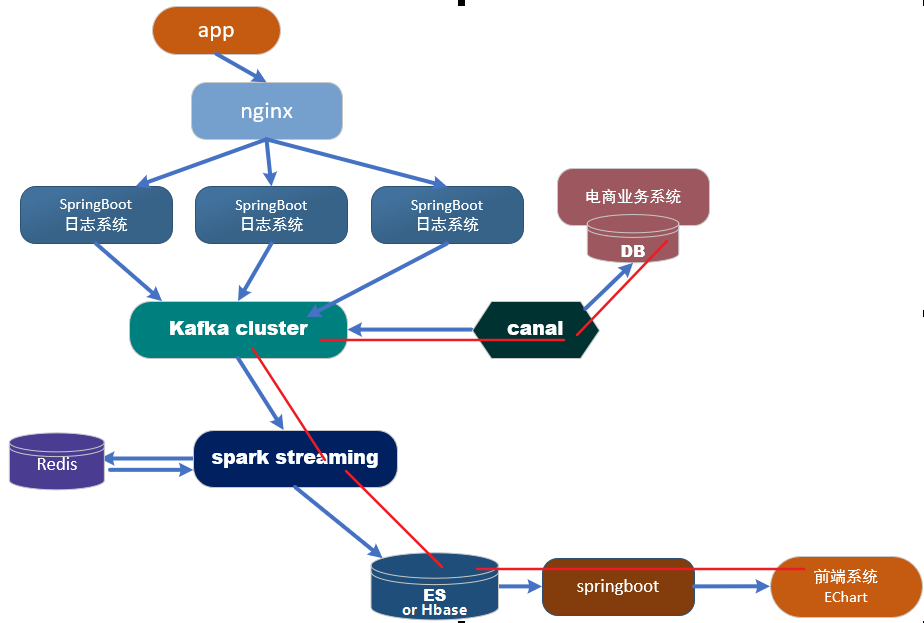
尚硅谷电商数仓项目--实时计算（交易额）

版本：V 2.0

张晨

# ·采集数据

## 1 框架流程



## 2 canal 入门

### 2.1 什么是 canal

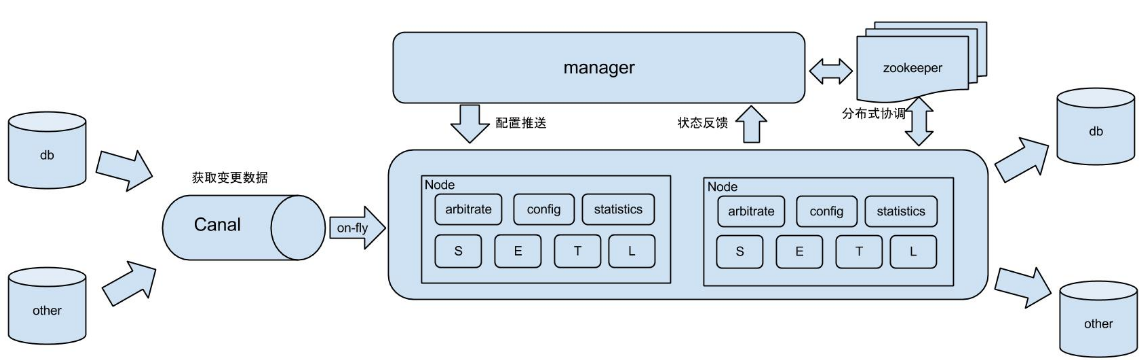
阿里巴巴B2B公司，因为业务的特性，卖家主要集中在国内，买家主要集中在国外，所以衍生出了杭州和美国异地机房的需求，从2010年开始，阿里系公司开始逐步的尝试基于数据库的日志解析，获取增量变更进行同步，由此衍生出了增量订阅&消费的业务。

canal是用java开发的基于数据库增量日志解析，提供增量数据订阅&消费的中间件。目前，canal主要支持了MySQL的binlog解析，解析完成后才利用canal client 用来处理获得的相关数据。（数据库同步需要阿里的otter中间件，基于canal）。

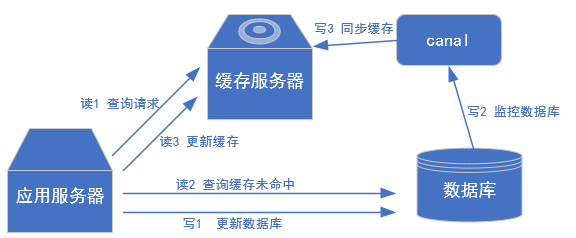
### 2.2 使用场景

1）原始场景： 阿里otter中间件的一部分

otter是阿里用于进行异地数据库之间的同步框架，canal是其中一部分。



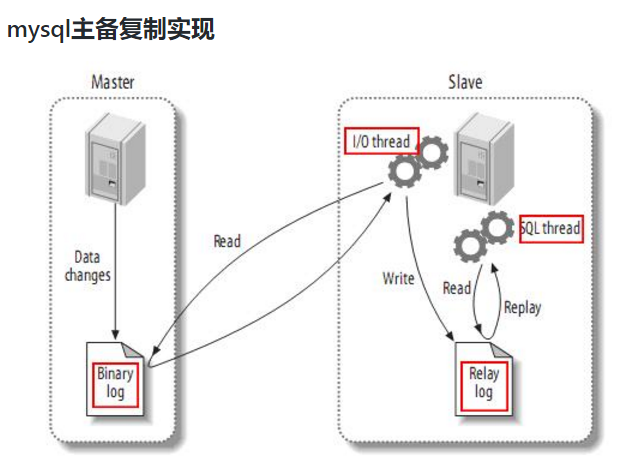
2） 常见场景1：更新缓存



3） 场景2：抓取业务数据新增变化表，用于制作拉链表。

4） \*\*场景3：抓取业务表的新增变化数据，用于制作实时统计。

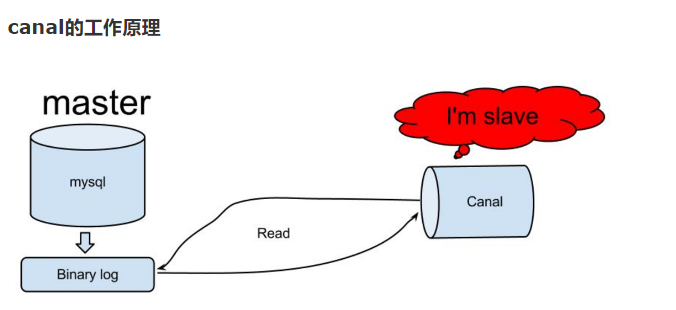
### 2.3 canal的工作原理



复制过程分成三步：

1. Master主库将改变记录，写到二进制日志(binary log)中
2. Slave从库向mysql master发送dump协议，将master主库的binary log events拷贝到它的中继日志(relay log)；
3. Slave从库读取并重做中继日志中的事件，将改变的数据同步到自己的数据库。

canal的工作原理很简单，就是**把自己伪装成slave，假装从master复制数据**。



### 2.4 mysql的binlog

#### 2.4.1 什么是binlog

MySQL的二进制日志可以说是MySQL最重要的日志了，它记录了所有的DDL和DML(除了数据查询语句)语句，以事件形式记录，还包含语句所执行的消耗的时间，MySQL的二进制日志是事务安全型的。

一般来说开启二进制日志大概会有1%的性能损耗。二进制有两个最重要的使用场景:

其一：MySQL Replication在Master端开启binlog，Mster把它的二进制日志传递给slaves来达到master-slave数据一致的目的。

其二：自然就是数据恢复了，通过使用mysqlbinlog工具来使恢复数据。

二进制日志包括两类文件：二进制日志索引文件（文件名后缀为.index）用于记录所有的二进制文件，二进制日志文件（文件名后缀为.00000\*）记录数据库所有的DDL和DML(除了数据查询语句)语句事件。

#### 2.4.2 binlog的开启

在mysql的配置文件(Linux: /etc/my.cnf , Windows: \my.ini)下,修改配置

在[mysqld] 区块

设置/添加

|  |
| --- |
| log-bin=mysql-bin |

这个表示binlog日志的前缀是mysql-bin ，以后生成的日志文件就是 mysql-bin.123456 的文件后面的数字按顺序生成。 每次mysql重启或者到达单个文件大小的阈值时，新生一个文件，按顺序编号。

### 2.4.3 binlog的分类设置

mysql binlog的格式，那就是有三种，分别是STATEMENT,MIXED,ROW。

在配置文件中可以选择配置

|  |
| --- |
| binlog\_format=row |

区别：

**1） statement**

语句级，binlog会记录每次一执行写操作的语句。

相对row模式节省空间，但是可能产生不一致性，比如

update tt set create\_date=now()

如果用binlog日志进行恢复，由于执行时间不同可能产生的数据就不同。

优点： 节省空间

缺点： 有可能造成数据不一致。

**2） row**

行级， binlog会记录每次操作后每行记录的变化。

优点：保持数据的绝对一致性。因为不管sql是什么，引用了什么函数，他只记录执行后的效果。

缺点：占用较大空间。

**3） mixed**

statement的升级版，一定程度上解决了，因为一些情况而造成的statement模式不一致问题

在某些情况下譬如：

当函数中包含 UUID() 时；

包含 AUTO\_INCREMENT 字段的表被更新时；

执行 INSERT DELAYED 语句时；

用 UDF 时；

会按照 ROW的方式进行处理

优点：节省空间，同时兼顾了一定的一致性。

缺点：还有些极个别情况依旧会造成不一致，另外statement和mixed对于需要对binlog的监控的情况都不方便。

## 3 mysql的准备

### 3.1 导入模拟业务数据库

### 3.2 赋权限

在mysql中执行

|  |
| --- |
| GRANT SELECT, REPLICATION SLAVE, REPLICATION CLIENT ON \*.\* TO 'canal'@'%' IDENTIFIED BY 'canal' ; |

### 3.3 修改/etc/my.cnf文件

|  |
| --- |
| server-id= 1  log-bin=mysql-bin  binlog\_format=row  binlog-do-db=gmallXXXXX |

### 3.4 重启Mysql

## 4 canal 安装

### 4.1 canal的下载

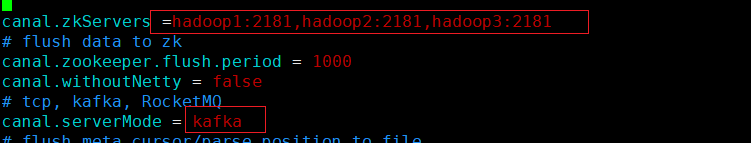
https://github.com/alibaba/canal/releases

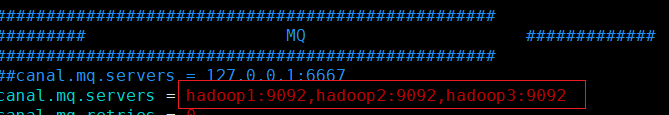
把canal.deployer-1.1.4.tar.gz拷贝到linux,解压缩

### 4.2 修改canal的配置

vim conf/canal.properties

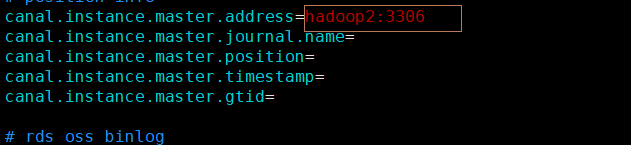
这个文件是canal的基本通用配置，主要关心一下端口号，不改的话默认就是11111

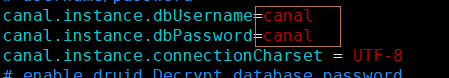
：

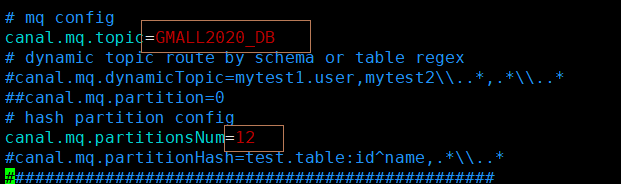


vim conf/example/instance.properties

instance.properties是针对要追踪的mysql的实例配置





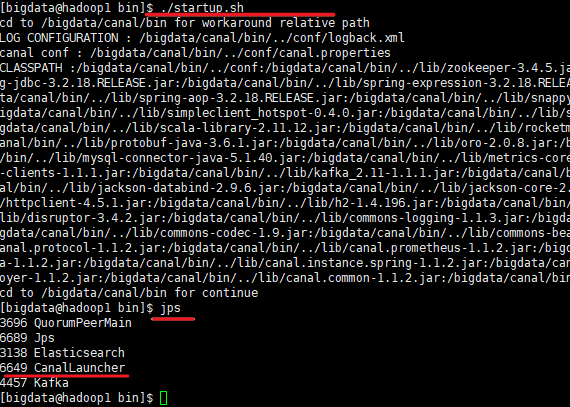


### 4.3 把canal目录分发给其他虚拟机

### 4.4 在2-3台节点中启动canal

启动canal

|  |
| --- |
| ./bin/startup.sh |



### 4.5 只是高可用，不是高负载

这种zookeeper为观察者监控的模式，只能实现高可用，而不是负载均衡，即同一时点只有一个canal-server节点能够监控某个数据源，只要这个节点能够正常工作，那么其他监控这个数据源的canal-server只能做stand-by，直到工作节点停掉，其他canal-server节点才能抢占。

## 5 kafka客户端测试

|  |
| --- |
| /bigdata/kafka\_2.11-0.11.0.2/bin/kafka-console-consumer.sh --bootstrap-server hadoop1:9092,hadoop2:9092,hadoop3:9092 --topic **GMALL2020\_DB** |

# ODS层处理

## 1 数据格式

### 样例sql

|  |
| --- |
| INSERT INTO z\_user\_info VALUES(16,'zhang3','13810001010'),(17,'zhang3','13810001010'); |

### kafka中收到canal的消息

|  |
| --- |
| {**"data"**:[{"id":"16","user\_name":"zhang3","tel":"13810001010"},{"id":"17","user\_name":"zhang3","tel":"13810001010"}],"database":"gmall-2020-04","es":1589196502000,"id":4,"isDdl":false,"mysqlType":{"id":"bigint(20)","user\_name":"varchar(20)","tel":"varchar(20)"},"old":null,"pkNames":["id"],"sql":"","sqlType":{"id":-5,"user\_name":12,"tel":12},"table":"z\_user\_info","ts":1589196502433,"type":"INSERT"} |

## 2 TOPIC分流

canal会追踪整个数据库的变更，把所有的数据变化都发到一个topic中了，但是为了后续处理方便，应该把这些数据根据不同的表，分流到不同的主题中去。

## 3 sparkstreaming分流业务代码

|  |
| --- |
| **object** BaseDBCanalApp {   **def** main(args: Array[String]): Unit = {  **val** sparkConf: SparkConf = **new** SparkConf().setAppName(**"canal\_app"**).setMaster(**"local[\*]"**)  **val** ssc = **new** StreamingContext(sparkConf,*Seconds*(5))  **val** topic=**"ODS\_BASE\_DB"  val** groupId=**"gmall\_base\_db\_canal\_group"** *//读取redis中的偏移量* **val** offsets: Map[TopicPartition, Long] = OffsetManager.*getOffset*(groupId,topic)   *//加载数据流* **var** dbInputDstream: InputDStream[ConsumerRecord[String, String]]=**null  if**(offsets==**null**){  *println*(**"no offsets"**)  dbInputDstream = MyKafkaUtil.*getKafkaStream*(topic,ssc,groupId )  }**else**{  *println*(**"offset:"**+offsets.mkString(**","**))  dbInputDstream = MyKafkaUtil.*getKafkaStream*(topic,ssc,offsets,groupId)  }  *//读取偏移量移动位置* **var** offsetRanges: Array[OffsetRange] = Array.*empty*[OffsetRange]  **val** recordDstream: DStream[ConsumerRecord[String, String]] = dbInputDstream.transform { rdd =>  offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges  rdd  }   **val** jsonDstream: DStream[JSONObject] = recordDstream.map { record =>  **val** jsonString: String = record.value()  **val** jSONObject: JSONObject = JSON.*parseObject*(jsonString)  jSONObject  }   *//写入数据* jsonDstream.foreachRDD { rdd =>  rdd.foreachPartition { jsonItr =>  **if**(offsetRanges!=**null**&&offsetRanges.size>0){  **val** offsetRange: OffsetRange = offsetRanges(TaskContext.*get*().partitionId())  *println*(**"from:"**+offsetRange.fromOffset +**" --- to:"**+offsetRange.untilOffset)  }  **for** (json <- jsonItr) {  *//* **val** tableName: String = json.getString(**"table"**)  **val** jSONArray: JSONArray = json.getJSONArray(**"data"**)  **for**(i <- 0 to jSONArray.size()-1 ){  **val** jsonObj: JSONObject = jSONArray.getJSONObject(i)  **val** topic: String = **"DW\_"**+tableName.toUpperCase  **val** key:String =tableName+**"\_"**+jsonObj.getString(**"id"**)  MyKafkaSink.*send*(topic,key, jsonObj.toJSONString)  }    }  }  *// 偏移量移动位置写入redis* OffsetManager.*saveOffset*(groupId, topic, offsetRanges)   }    ssc.start()  ssc.awaitTermination()   }  } |

Kafka发送数据工具类

|  |
| --- |
| **import** java.util.Properties  **import** com.atguigu.gmall2020.realtime.util.MyKafkaUtil.*properties* **import** org.apache.kafka.clients.producer.{KafkaProducer, ProducerRecord}  **object** MyKafkaSink {  **private val** *properties*: Properties = PropertiesUtil.*load*(**"config.properties"**)  **val** *broker\_list* = *properties*.getProperty(**"kafka.broker.list"**)  **var** *kafkaProducer*: KafkaProducer[String, String] = **null   def** createKafkaProducer: KafkaProducer[String, String] = {  **val** properties = **new** Properties  properties.put(**"bootstrap.servers"**, *broker\_list*)  properties.put(**"key.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**)  properties.put(**"value.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**)  properties.put(**"enable.idompotence"**,(**true**: java.lang.Boolean))  **var** producer: KafkaProducer[String, String] = **null  try** producer = **new** KafkaProducer[String, String](properties)  **catch** {  **case** e: Exception =>  e.printStackTrace()  }  producer  }   **def** send(topic: String, msg: String): Unit = {  **if** (*kafkaProducer* == **null**) *kafkaProducer* = *createKafkaProducer  kafkaProducer*.send(**new** ProducerRecord[String, String](topic, msg))   }   **def** send(topic: String,key:String, msg: String): Unit = {  **if** (*kafkaProducer* == **null**) *kafkaProducer* = *createKafkaProducer  kafkaProducer*.send(**new** ProducerRecord[String, String](topic,key, msg))   } } |

# Maxwell版本的ODS层

## 1 Maxwell

maxwell 是由美国zendesk开源，用java编写的Mysql实时抓取软件。 其抓取的原理也是基于binlog。

### 1.1 工具对比

1. Maxwell 没有 Canal那种server+client模式，只有一个server把数据发送到消息队列或redis。
2. Maxwell 有一个亮点功能，就是Canal只能抓取最新数据，对已存在的历史数据没有办法处理。而Maxwell有一个bootstrap功能，可以直接引导出完整的历史数据用于初始化，非常好用。
3. Maxwell不能直接支持HA，但是它支持断点还原，即错误解决后重启继续上次点儿读取数据。
4. Maxwell只支持json格式，而Canal如果用Server+client模式的话，可以自定义格式。
5. Maxwell比Canal更加轻量级。

### 1.2 安装Maxwell

解压缩maxwell-1.25.0.tar.gz 到某个目录下。

### 1.3 使用前准备工作

在数据库中建立一个maxwell库用于存储Maxwell的元数据。

|  |
| --- |
| CREATE DATABASE maxwell ; |

并且分配一个账号可以操作该数据库

|  |
| --- |
| GRANT ALL ON maxwell.\* TO 'maxwell'@'%' IDENTIFIED BY '123123'; |

分配这个账号可以监控其他数据库的权限

|  |
| --- |
| GRANT SELECT ,REPLICATION SLAVE , REPLICATION CLIENT ON \*.\* TO maxwell@'%' |

### 1.4 使用Maxwell监控抓取MySql数据

在任意位置建立maxwell.properties 文件

|  |
| --- |
| producer=kafka  kafka.bootstrap.servers=hadoop1:9092,hadoop2:9092,hadoop3:9092  kafka\_topic=ODS\_DB\_GMALL2020\_M  host=hadoop2  user=maxwell  password=123123  client\_id=maxwell\_1 |

启动程序

|  |
| --- |
| /ext/maxwell-1.25.0/bin/maxwell --config /xxx/xxxx/maxwell.properties >/dev/null 2>&1 & |

### 1.5 修改或插入mysql数据，并消费kafka进行观察

|  |
| --- |
| /ext/kafka\_2.11-1.0.0/bin/kafka-topics.sh --create --topic ODS\_DB\_GMALL2020\_M --zookeeper hadoop1:2181,hadoop2:2181,hadoop3:2181 --partitions 12 --replication-factor 1 |

执行测试语句

|  |
| --- |
| INSERT INTO z\_user\_info VALUES(30,'zhang3','13810001010'),(31,'li4','1389999999'); |

对比

|  |  |
| --- | --- |
| canal | maxwell |
| {"data":[{"id":"30","user\_name":"zhang3","tel":"13810001010"},{"id":"31","user\_name":"li4","tel":"1389999999"}],"database":"gmall-2020-04","es":1589385314000,"id":2,"isDdl":false,"mysqlType":{"id":"bigint(20)","user\_name":"varchar(20)","tel":"varchar(20)"},"old":null,"pkNames":["id"],"sql":"","sqlType":{"id":-5,"user\_name":12,"tel":12},"table":"z\_user\_info","ts":1589385314116,"type":"INSERT"} | {"database":"gmall-2020-04","table":"z\_user\_info","type":"insert","ts":1589385314,"xid":82982,"xoffset":0,"data":{"id":**30**,"user\_name":"zhang3","tel":"13810001010"}}  {"database":"gmall-2020-04","table":"z\_user\_info","type":"insert","ts":1589385314,"xid":82982,"commit":true,"data":{"id":31,"user\_name":"li4","tel":"1389999999"}} |

**执行update操作**

|  |
| --- |
| UPDATE z\_user\_info SET user\_name='wang55' WHERE id IN(30,31) |

|  |  |
| --- | --- |
| canal | maxwell |
| {"data":[{"id":"30","user\_name":"wang55","tel":"13810001010"},{"id":"31","user\_name":"wang55","tel":"1389999999"}],"database":"gmall-2020-04","es":1589385508000,"id":3,"isDdl":false,"mysqlType":{"id":"bigint(20)","user\_name":"varchar(20)","tel":"varchar(20)"},"old":[{"user\_name":"zhang3"},{"user\_name":"li4"}],"pkNames":["id"],"sql":"","sqlType":{"id":-5,"user\_name":12,"tel":12},"table":"z\_user\_info","ts":1589385508676,"type":"UPDATE"} | {"database":"gmall-2020-04","table":"z\_user\_info","type":"update","ts":1589385508,"xid":83206,"xoffset":0,"data":{"id":30,"user\_name":"wang55","tel":"13810001010"},"old":{"user\_name":"zhang3"}}  {"database":"gmall-2020-04","table":"z\_user\_info","type":"update","ts":1589385508,"xid":83206,"commit":true,"data":{"id":31,"user\_name":"wang55","tel":"1389999999"},"old":{"user\_name":"li4"}} |

**delete操作**

|  |
| --- |
| DELETE FROM z\_user\_info WHERE id IN(30,31) |

|  |  |
| --- | --- |
| canal | maxwell |
| {"data":[{"id":"30","user\_name":"wang55","tel":"13810001010"},{"id":"31","user\_name":"wang55","tel":"1389999999"}],"database":"gmall-2020-04","es":1589385644000,"id":4,"isDdl":false,"mysqlType":{"id":"bigint(20)","user\_name":"varchar(20)","tel":"varchar(20)"},"old":null,"pkNames":["id"],"sql":"","sqlType":{"id":-5,"user\_name":12,"tel":12},"table":"z\_user\_info","ts":1589385644829,"type":"DELETE"} | {"database":"gmall-2020-04","table":"z\_user\_info","type":"delete","ts":1589385644,"xid":83367,"xoffset":0,"data":{"id":30,"user\_name":"wang55","tel":"13810001010"}}  {"database":"gmall-2020-04","table":"z\_user\_info","type":"delete","ts":1589385644,"xid":83367,"commit":true,"data":{"id":31,"user\_name":"wang55","tel":"1389999999"}} |

**总结数据特点：**

**一 日志结构**

canal 每一条SQL会产生一条日志，如果该条Sql影响了多行数据，则已经会通过集合的方式归集在这条日志中。（即使是一条数据也会是数组结构）

maxwell 以影响的数据为单位产生日志，即每影响一条数据就会产生一条日志。如果想知道这些日志是否是通过某一条sql产生的可以通过xid进行判断，相同的xid的日志来自同一sql。

**二 数字类型**

当原始数据是数字类型时,maxwell会尊重原始数据的类型不增加双引，变为字符串。

canal一律转换为字符串。

**三 带原始数据字段定义**

canal数据中会带入表结构。maxwell更简洁。

### 1.6 完成Maxwell的ODS层分流

|  |
| --- |
| **object** DbGmallMaxwellApp {   **def** main(args: Array[String]): Unit = {  **val** sparkConf: SparkConf = **new** SparkConf().setAppName(**"canal\_app"**).setMaster(**"local[\*]"**)  **val** ssc = **new** StreamingContext(sparkConf,*Seconds*(5))  **val** topic=**"ODS\_DB\_GMALL2020\_M"  val** groupId=**"gmall\_base\_db\_maxwell\_group"** *//读取redis中的偏移量* **val** offsets: Map[TopicPartition, Long] = OffsetManager.*getOffset*(groupId,topic)   *//加载数据流* **var** dbInputDstream: InputDStream[ConsumerRecord[String, String]]=**null  if**(offsets==**null**){  *println*(**"no offsets"**)  dbInputDstream = MyKafkaUtil.*getKafkaStream*(topic,ssc,groupId )  }**else**{  *println*(**"offset:"**+offsets.mkString(**","**))  dbInputDstream = MyKafkaUtil.*getKafkaStream*(topic,ssc,offsets,groupId)  }  *//读取偏移量移动位置* **var** offsetRanges: Array[OffsetRange] = Array.*empty*[OffsetRange]  **val** recordDstream: DStream[ConsumerRecord[String, String]] = dbInputDstream.transform { rdd =>  offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges  rdd  }   **val** jsonDstream: DStream[JSONObject] = recordDstream.map { record =>  **val** jsonString: String = record.value()  **val** jSONObject: JSONObject = JSON.*parseObject*(jsonString)  jSONObject  }   *//写入数据* jsonDstream.foreachRDD { rdd =>  rdd.foreachPartition { jsonItr =>  **if**(offsetRanges!=**null**&&offsetRanges.size>0){  **val** offsetRange: OffsetRange = offsetRanges(TaskContext.*get*().partitionId())  *println*(**"from:"**+offsetRange.fromOffset +**" --- to:"**+offsetRange.untilOffset)  } **if**(!json.getString(**"type"**).equals(**"bootstrap-start"**)&& !json.getString(**"type"**).equals(**"bootstrap-complete"**)){  **val** tableName: String = json.getString(**"table"**)  **val** jsonObj: JSONObject = json.getJSONObject(**"data"**)  **val** topic: String = **"ODS\_T\_"**+tableName.toUpperCase  **val** key:String =tableName+**"\_"**+jsonObj.getString(**"id"**)  MyKafkaSink.*send*(topic,key, jsonObj.toJSONString)  }  }  *// 偏移量移动位置写入redis* OffsetManager.*saveOffset*(groupId, topic, offsetRanges)   }   ssc.start()  ssc.awaitTermination()   }  } |
|  |

只有红字部分有别于Canal的消费端

# 当日新增付费用户首单分析

## 1、需求分析：按地区（用户性别、用户年龄段）统计当日新增付费用户首单平均消费

**难点：**1 每笔订单都要判断是否是该用户的首单。

2 无论是地区名称、用户性别、用户年龄，订单表中都没有这些字段。但是可以通过外键的形式关联其他表得到这些值。

## 2、 处理判断首单业务的策略

判断是否首单的要点，在于该用户之前是否参与过消费（下单）。那么如何知道用户之前是否参与过消费，如果临时从所有消费记录中查询，是非常不现实的。那么只有将“用户是否消费过”这个状态进行保存并长期维护起来。在有需要的时候通过用户id进行关联查询。

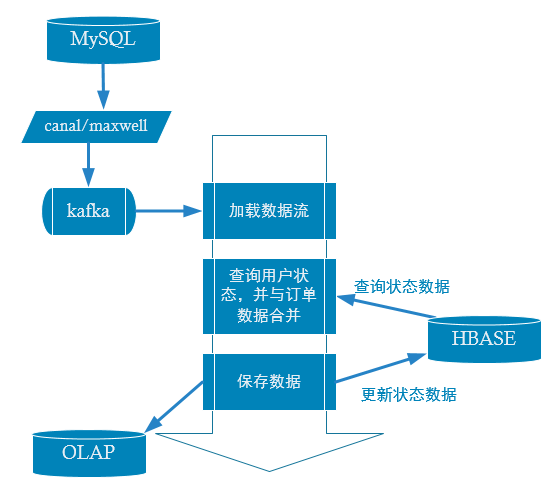
在实际生产中，这种用户状态是非常常见的比如“用户是否退过单”、“用户是否投过诉”、“用户是否是高净值用户”等等。

那么现在问题就变为，如何保存并长期维护这种状态。考虑到

1. **这是一个保存周期较长的数据。**
2. **必须可修改状态值。**
3. **查询模式基本上是k-v模式的查询。**

所以综上这三点比较适合保存在**Hbase**中。

## 3 、数据流程图



## 4 hbase （with phoenix）建表

|  |
| --- |
| create table user\_state2020( user\_id varchar primary key ,state.if\_consumed varchar ) SALT\_BUCKETS = 3 |

## 5、代码实现

### pom.xml

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.phoenix</**groupId**>  <**artifactId**>phoenix-spark</**artifactId**>  <**version**>4.14.2-HBase-1.3</**version**> </**dependency**>  <**dependency**>  <**groupId**>org.apache.spark</**groupId**>  <**artifactId**>spark-sql\_2.11</**artifactId**>  <**version**>${spark.version}</**version**> </**dependency**> |

### case class

|  |
| --- |
| **case class** OrderInfo(  id: Long,  province\_id: Long,  order\_status: String,  user\_id: Long,  final\_total\_amount: Double,  benefit\_reduce\_amount: Double,  original\_total\_amount: Double,  feight\_fee: Double,  expire\_time: String,  create\_time: String,  operate\_time: String,  **var** create\_date: String,  **var** create\_hour: String,  **var** if\_first\_order:String,   **var** province\_name:String,  **var** province\_area\_code:String,   **var** user\_age\_group:String,  **var** user\_gender:String    ) |

### phoenix查询工具类

|  |
| --- |
| **import** java.sql.{Connection, DriverManager, ResultSet, ResultSetMetaData, Statement} **import** com.alibaba.fastjson.JSONObject **import** scala.collection.mutable.ListBuffer  **object** PhoenixUtil {   **def** main(args: Array[String]): Unit = {  **val** list: List[ JSONObject] = *queryList*(**"select \* from CUSTOMER0919"**)  *println*(list)  }    **def** queryList(sql:String):List[JSONObject]={  Class.*forName*(**"org.apache.phoenix.jdbc.PhoenixDriver"**)  **val** resultList: ListBuffer[JSONObject] = **new** ListBuffer[ JSONObject]()  **val** conn: Connection = DriverManager.*getConnection*(**"jdbc:phoenix:hadoop1,hadoop2,hadoop3:2181"**)  **val** stat: Statement = conn.createStatement  *println*(sql)  **val** rs: ResultSet = stat.executeQuery(sql )  **val** md: ResultSetMetaData = rs.getMetaData  **while** ( rs.next ) {  **val** rowData = **new** JSONObject();  **for** (i <-1 to md.getColumnCount ) {  rowData.put(md.getColumnName(i), rs.getObject(i))  }  resultList+=rowData  }   stat.close()  conn.close()  resultList.toList  }   } |

### 实时计算代码

|  |
| --- |
| **import** com.alibaba.fastjson.{JSON, JSONObject} **import** com.atguigu.gmall2020.realtime.bean.dim.{ProvinceInfo, UserState} **import** com.atguigu.gmall2020.realtime.bean.dw.OrderInfo **import** com.atguigu.gmall2020.realtime.util.{MyKafkaUtil, OffsetManager, PhoenixUtil} **import** org.apache.hadoop.conf.Configuration **import** org.apache.kafka.clients.consumer.ConsumerRecord **import** org.apache.kafka.common.TopicPartition **import** org.apache.spark.SparkConf **import** org.apache.spark.streaming.dstream.{DStream, InputDStream} **import** org.apache.spark.streaming.kafka010.{HasOffsetRanges, OffsetRange} **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.phoenix.spark.\_ **import** org.apache.spark.broadcast.Broadcast **import** org.apache.spark.rdd.RDD  **object** OrderInfoApp {    **def** main(args: Array[String]): Unit = {  **val** sparkConf: SparkConf = **new** SparkConf().setMaster(**"local[\*]"**).setAppName(**"dau\_app"**)  **val** ssc = **new** StreamingContext(sparkConf, *Seconds*(5))  **val** groupId = **"GMALL\_ORDER\_INFO\_CONSUMER"  val** topic = **"ODS\_T\_ORDER\_INFO"** *//从redis读取偏移量* **val** orderOffsets: Map[TopicPartition, Long] = OffsetManager.*getOffset*(groupId, topic)   *//根据偏移起始点获得数据  //判断如果之前没有在redis保存，则从kafka最新加载数据* **var** orderInfoInputDstream: InputDStream[ConsumerRecord[String, String]] = **null  if** (orderOffsets != **null** && orderOffsets.size > 0) {  orderInfoInputDstream = MyKafkaUtil.*getKafkaStream*(topic, ssc, orderOffsets, groupId)  } **else** {  orderInfoInputDstream = MyKafkaUtil.*getKafkaStream*(topic, ssc, groupId)  }   *//获得偏移结束点* **var** orderInfoOffsetRanges: Array[OffsetRange] = Array.*empty*[OffsetRange]  **val** orderInfoInputGetOffsetDstream: DStream[ConsumerRecord[String, String]] = orderInfoInputDstream.transform { rdd =>  orderInfoOffsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges  rdd  }   **val** orderInfoDstream: DStream[OrderInfo] = orderInfoInputGetOffsetDstream.map { record =>  **val** jsonString: String = record.value()  **val** orderInfo: OrderInfo = JSON.*parseObject*(jsonString, *classOf*[OrderInfo])   **val** createTimeArr: Array[String] = orderInfo.create\_time.split(**" "**)  orderInfo.create\_date = createTimeArr(0)  orderInfo.create\_hour = createTimeArr(1).split(**":"**)(0)   orderInfo  }    **val** orderWithIfFirstDstream: DStream[OrderInfo] = orderInfoDstream.mapPartitions { orderInfoItr =>  **val** orderInfoList: List[OrderInfo] = orderInfoItr.toList  **if** (orderInfoList.size > 0) {  *//针对分区中的订单中的所有客户 进行批量查询* **val** userIds: String = orderInfoList.map(**"'"** + \_.user\_id + **"'"**).mkString(**","**)   **val** userStateList: List[JSONObject] = PhoenixUtil.*queryList*(**"select user\_id,if\_consumed from GMALL0919\_USER\_STATE where user\_id in ("** + userIds + **")"**)  *// [{USERID:123, IF\_ORDERED:1 },{USERID:2334, IF\_ORDERED:1 },{USERID:4355, IF\_ORDERED:1 }]  // 进行转换 把List[Map] 变成Map* **val** userIfOrderedMap: Map[Long, String] = userStateList.map(userStateJsonObj => (userStateJsonObj.getLong(**"USER\_ID"**).toLong, userStateJsonObj.getString(**"IF\_ORDERED"**))).toMap  *//{123:1,2334:1,4355:1}  //进行判断 ，打首单表情* **for** (orderInfo <- orderInfoList) {  **val** ifOrderedUser: String = userIfOrderedMap.getOrElse(orderInfo.user\_id, **"0"**) *//  //是下单用户不是首单 否->首单* **if** (ifOrderedUser == **"1"**) {  orderInfo.if\_first\_order = **"0"** } **else** {  orderInfo.if\_first\_order = **"1"** }  }  orderInfoList.toIterator  } **else** {  orderInfoItr  }   }    *//在一个批次内 第一笔如果是首单 那么本批次的该用户其他单据改为非首单  // 以userId 进行分组* **val** groupByUserDstream: DStream[(Long, Iterable[OrderInfo])] = orderWithIfFirstDstream.map(orderInfo => (orderInfo.user\_id, orderInfo)).groupByKey()   **val** orderInfoFinalDstream: DStream[OrderInfo] = groupByUserDstream.flatMap { **case** (userId, orderInfoItr) =>  **val** orderList: List[OrderInfo] = orderInfoItr.toList  *//* **if** (orderList.size > 1) { *// 如果在这个批次中这个用户有多笔订单* **val** sortedOrderList: List[OrderInfo] = orderList.sortWith((orderInfo1, orderInfo2) => orderInfo1.create\_time < orderInfo2.create\_time)  **if** (sortedOrderList(0).if\_first\_order == **"1"**) { *//排序后，如果第一笔订单是首单，那么其他的订单都取消首单标志* **for** (i <- 1 to sortedOrderList.size - 1) {  sortedOrderList(i).if\_first\_order = **"0"** }  }  sortedOrderList  } **else** {  orderList  }  }   orderInfoFinalDstream.cache()   *//同步到userState表中 只有标记了 是首单的用户 才需要同步到用户状态表中* **val** userStateDstream: DStream[UserState] = orderInfoFinalDstream.filter(\_.if\_first\_order == **"1"**).map(orderInfo => *UserState*(orderInfo.id, orderInfo.if\_first\_order))  userStateDstream.foreachRDD { rdd =>  rdd.saveToPhoenix(**"GMALL1122\_USER\_STATE"**, *Seq*(**"USER\_ID"**, **"IF\_CONSUMED"**), **new** Configuration(), Some(**"hadoop1,hadoop2,hadoop3:2181"**))  OffsetManager.*saveOffset*(groupId ,topic, orderInfoOffsetRanges)  }   orderInfoFinalDstream.print(1000)  ssc.start()  ssc.awaitTermination()   }   } |

# 第五章 维度数据业务流程

## 1 处理维度数据合并的策略

维度数据和状态数据非常像，但也有不同之处：

**共同点：**

* 长期保存维护
* 可修改
* 使用k-v方式查询

**不同点：**

* 数据变更的时机不同

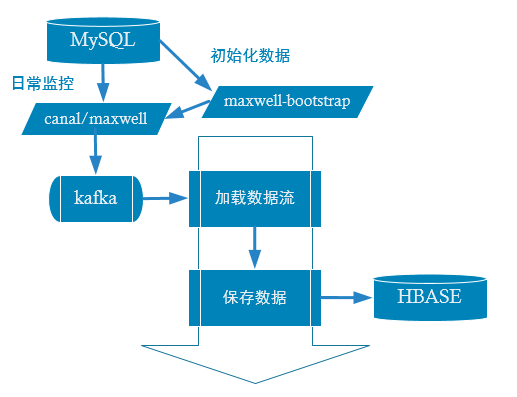
状态数据往往因为事实数据的新增变化而变更

维度数据只会受到业务数据库中的变化而变更

**综上 ：**

根据共同点，维度数据也是非常适合使用hbase存储的，稍有不同的是维度数据必须启动单独的实时计算来监控维度表变化来更新实时数据。

## 2 实时处理流程



## 3 在hbase中建表

|  |
| --- |
| create table gmall2020\_province\_info ( id varchar primary key , info.name varchar , info.region\_id varchar , info.area\_code varchar )SALT\_BUCKETS = 3 |

## 4 实时计算代码

case class

|  |
| --- |
| **case class** ProvinceInfo(id:String,  name:String,  region\_id:String,  area\_code:String) {  } |

provinceInfoApp

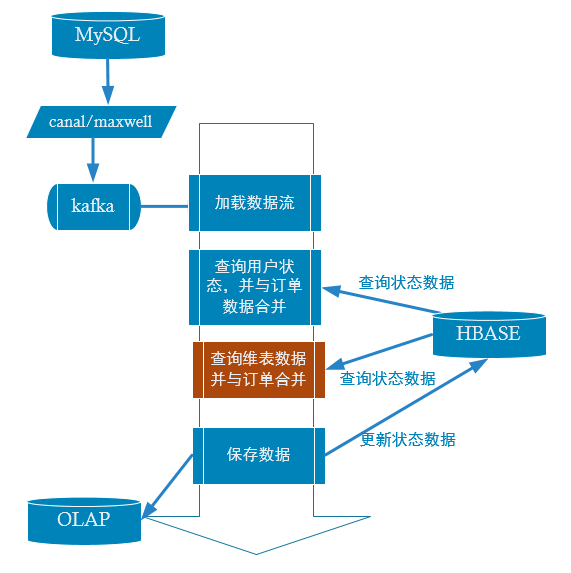
|  |
| --- |
| **import** com.alibaba.fastjson.JSON **import** com.atguigu.gmall2020.realtime.bean.dim.ProvinceInfo **import** com.atguigu.gmall2020.realtime.util.{MyKafkaUtil, OffsetManager} **import** org.apache.hadoop.conf.Configuration **import** org.apache.kafka.clients.consumer.ConsumerRecord **import** org.apache.kafka.common.TopicPartition **import** org.apache.spark.SparkConf **import** org.apache.spark.streaming.dstream.{DStream, InputDStream} **import** org.apache.spark.streaming.kafka010.{HasOffsetRanges, OffsetRange} **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.phoenix.spark.\_  **object** ProvinceInfoApp {  **def** main(args: Array[String]): Unit = {    **val** sparkConf: SparkConf = **new** SparkConf().setAppName(**"province\_info\_app"**).setMaster(**"local[\*]"**)  **val** ssc = **new** StreamingContext(sparkConf, *Seconds*(5))   **val** groupId = **"gmall\_province\_group"  val** topic = **"ODS\_T\_BASE\_PROVINCE"  val** offsets: Map[TopicPartition, Long] = OffsetManager.*getOffset*(groupId, topic)   **var** inputDstream: InputDStream[ConsumerRecord[String, String]] = **null  if** (offsets != **null** && offsets.size > 0) {  inputDstream = MyKafkaUtil.*getKafkaStream*(topic, ssc, offsets, groupId)  } **else** {  inputDstream = MyKafkaUtil.*getKafkaStream*(topic, ssc, groupId)  }   *//获得偏移结束点* **var** offsetRanges: Array[OffsetRange] = Array.*empty*[OffsetRange]  **val** inputGetOffsetDstream: DStream[ConsumerRecord[String, String]] = inputDstream.transform { rdd =>  offsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges  rdd  }   **val** provinceInfoDstream: DStream[ProvinceInfo] = inputDstream.map { record =>  **val** jsonString: String = record.value()  **val** provinceInfo: ProvinceInfo = JSON.*parseObject*(jsonString, *classOf*[ProvinceInfo])  provinceInfo  }   provinceInfoDstream.cache()   provinceInfoDstream.print(1000)   provinceInfoDstream.foreachRDD { rdd =>  rdd.saveToPhoenix(**"gmall2020\_province\_info"**, *Seq*(**"ID"**, **"NAME"**, **"REGION\_ID"**, **"AREA\_CODE"**), **new** Configuration, Some(**"hadoop1,hadoop2,hadoop3:2181"**))   OffsetManager.*saveOffset*(groupId, topic, offsetRanges)  }  ssc.start()  ssc.awaitTermination()   } |

## 5 利用maxwell-bootstrap 初始化数据

|  |
| --- |
| bin/maxwell-bootstrap --user maxwell --password 123123 --host hadoop2 --database gmall-2020-04 --table base\_province --client\_id maxwell\_1 |

其中client\_id 是指另一个已启动的maxwell监控进程的client\_id

## 6 实时业务中加入合并维表



4 完整代码

|  |
| --- |
| **package** com.atguigu.gmall2020.realtime.app.o2d   **import** java.text.SimpleDateFormat **import** java.util.Date  **import** com.alibaba.fastjson.{JSON, JSONObject} **import** com.atguigu.gmall2020.realtime.bean.dim.{ProvinceInfo, UserState} **import** com.atguigu.gmall2020.realtime.bean.dw.OrderInfo **import** com.atguigu.gmall2020.realtime.util.{MyEsUtil, MyKafkaUtil, OffsetManager, PhoenixUtil} **import** org.apache.hadoop.conf.Configuration **import** org.apache.kafka.clients.consumer.ConsumerRecord **import** org.apache.kafka.common.TopicPartition **import** org.apache.spark.SparkConf **import** org.apache.spark.streaming.dstream.{DStream, InputDStream} **import** org.apache.spark.streaming.kafka010.{HasOffsetRanges, OffsetRange} **import** org.apache.spark.streaming.{Seconds, StreamingContext} **import** org.apache.phoenix.spark.\_ **import** org.apache.spark.broadcast.Broadcast **import** org.apache.spark.rdd.RDD  **object** OrderInfoApp {    **def** main(args: Array[String]): Unit = {  **val** sparkConf: SparkConf = **new** SparkConf().setMaster(**"local[\*]"**).setAppName(**"order\_info\_app"**)  **val** ssc = **new** StreamingContext(sparkConf, *Seconds*(5))  **val** groupId = **"GMALL\_ORDER\_INFO\_CONSUMER"  val** topic = **"ODS\_T\_ORDER\_INFO"** *//从redis读取偏移量* **val** orderOffsets: Map[TopicPartition, Long] = OffsetManager.*getOffset*(groupId, topic)   *//根据偏移起始点获得数据  //判断如果之前没有在redis保存，则从kafka最新加载数据* **var** orderInfoInputDstream: InputDStream[ConsumerRecord[String, String]] = **null  if** (orderOffsets != **null** && orderOffsets.size > 0) {  orderInfoInputDstream = MyKafkaUtil.*getKafkaStream*(topic, ssc, orderOffsets, groupId)  } **else** {  orderInfoInputDstream = MyKafkaUtil.*getKafkaStream*(topic, ssc, groupId)  }   *//获得偏移结束点* **var** orderInfoOffsetRanges: Array[OffsetRange] = Array.*empty*[OffsetRange]  **val** orderInfoInputGetOffsetDstream: DStream[ConsumerRecord[String, String]] = orderInfoInputDstream.transform { rdd =>  orderInfoOffsetRanges = rdd.asInstanceOf[HasOffsetRanges].offsetRanges  rdd  }   **val** orderInfoDstream: DStream[OrderInfo] = orderInfoInputGetOffsetDstream.map { record =>  **val** jsonString: String = record.value()  **val** orderInfo: OrderInfo = JSON.*parseObject*(jsonString, *classOf*[OrderInfo])   **val** createTimeArr: Array[String] = orderInfo.create\_time.split(**" "**)  orderInfo.create\_date = createTimeArr(0)  orderInfo.create\_hour = createTimeArr(1).split(**":"**)(0)   orderInfo  }    **val** orderWithIfFirstDstream: DStream[OrderInfo] = orderInfoDstream.mapPartitions { orderInfoItr =>  **val** orderInfoList: List[OrderInfo] = orderInfoItr.toList  **if** (orderInfoList.size > 0) {  *//针对分区中的订单中的所有客户 进行批量查询* **val** userIds: String = orderInfoList.map(**"'"** + \_.user\_id + **"'"**).mkString(**","**)   **val** userStateList: List[JSONObject] = PhoenixUtil.*queryList*(**"select user\_id,if\_consumed from GMALL0919\_USER\_STATE where user\_id in ("** + userIds + **")"**)  *// [{USERID:123, IF\_ORDERED:1 },{USERID:2334, IF\_ORDERED:1 },{USERID:4355, IF\_ORDERED:1 }]  // 进行转换 把List[Map] 变成Map* **val** userIfOrderedMap: Map[Long, String] = userStateList.map(userStateJsonObj => (userStateJsonObj.getLong(**"USER\_ID"**).toLong, userStateJsonObj.getString(**"IF\_ORDERED"**))).toMap  *//{123:1,2334:1,4355:1}  //进行判断 ，打首单表情* **for** (orderInfo <- orderInfoList) {  **val** ifOrderedUser: String = userIfOrderedMap.getOrElse(orderInfo.user\_id, **"0"**) *//  //是下单用户不是首单 否->首单* **if** (ifOrderedUser == **"1"**) {  orderInfo.if\_first\_order = **"0"** } **else** {  orderInfo.if\_first\_order = **"1"** }  }  orderInfoList.toIterator  } **else** {  orderInfoItr  }   }    *//在一个批次内 第一笔如果是首单 那么本批次的该用户其他单据改为非首单  // 以userId 进行分组* **val** groupByUserDstream: DStream[(Long, Iterable[OrderInfo])] = orderWithIfFirstDstream.map(orderInfo => (orderInfo.user\_id, orderInfo)).groupByKey()   **val** orderInfoFinalDstream: DStream[OrderInfo] = groupByUserDstream.flatMap { **case** (userId, orderInfoItr) =>  **val** orderList: List[OrderInfo] = orderInfoItr.toList  *//* **if** (orderList.size > 1) { *// 如果在这个批次中这个用户有多笔订单* **val** sortedOrderList: List[OrderInfo] = orderList.sortWith((orderInfo1, orderInfo2) => orderInfo1.create\_time < orderInfo2.create\_time)  **if** (sortedOrderList(0).if\_first\_order == **"1"**) { *//排序后，如果第一笔订单是首单，那么其他的订单都取消首单标志* **for** (i <- 1 to sortedOrderList.size - 1) {  sortedOrderList(i).if\_first\_order = **"0"** }  }  sortedOrderList  } **else** {  orderList  }  }   orderInfoFinalDstream.cache()   *//同步到userState表中 只有标记了 是首单的用户 才需要同步到用户状态表中* **val** userStateDstream: DStream[UserState] = orderInfoFinalDstream.filter(\_.if\_first\_order == **"1"**).map(orderInfo => *UserState*(orderInfo.id, orderInfo.if\_first\_order))  userStateDstream.foreachRDD { rdd =>  rdd.saveToPhoenix(**"GMALL2020\_USER\_STATE"**, *Seq*(**"USER\_ID"**, **"IF\_CONSUMED"**), **new** Configuration(), Some(**"hadoop1,hadoop2,hadoop3:2181"**))  }   orderInfoFinalDstream.print(1000) ////////////////////////////////////////////////////////////  //////////////////////合并维表代码//////////////////////  ////////////////////////////////////////////////////////////  **val** orderInfoFinalWithProvinceDstream: DStream[OrderInfo] = orderInfoFinalDstream.transform { rdd =>  **val** provinceJsonObjList: List[JSONObject] = PhoenixUtil.*queryList*(**"select id,name,region\_id ,area\_code from GMALL2020\_PROVINCE\_INFO "**)   **val** provinceMap: Map[Long, ProvinceInfo] = provinceJsonObjList.map { jsonObj => (jsonObj.getLong(**"id"**).toLong, jsonObj.toJavaObject(*classOf*[ProvinceInfo])) }.toMap  **val** provinceMapBC: Broadcast[Map[Long, ProvinceInfo]] = ssc.sparkContext.broadcast(provinceMap)   **val** orderInfoWithProvinceRDD: RDD[OrderInfo] = rdd.map { orderInfo =>  **val** provinceMap: Map[Long, ProvinceInfo] = provinceMapBC.value  **val** provinceInfo: ProvinceInfo = provinceMap.getOrElse(orderInfo.province\_id, **null**)  **if** (provinceInfo != **null**) {  orderInfo.province\_name = provinceInfo.name  orderInfo.province\_area\_code = provinceInfo.area\_code  }  orderInfo  }  orderInfoWithProvinceRDD  }  ////////////////////////////////////////////////////////////  //////////////////////最终存储代码//////////////////////  ////////////////////////////////////////////////////////////  orderInfoFinalWithProvinceDstream.foreachRDD{rdd=>  rdd.cache()  *//存储用户状态* **val** userStateRdd: RDD[UserState] = rdd.filter(\_.if\_first\_order == **"1"**).map(orderInfo => *UserState*(orderInfo.id, orderInfo.if\_first\_order))  userStateRdd.saveToPhoenix(**"GMALL2020\_USER\_STATE"**, *Seq*(**"USER\_ID"**, **"IF\_CONSUMED"**), **new** Configuration(), Some(**"hadoop1,hadoop2,hadoop3:2181"**))   rdd.foreachPartition { orderInfoItr =>   **val** orderInfoList: List[(String, OrderInfo)] = orderInfoItr.toList.map(orderInfo => (orderInfo.id.toString, orderInfo))  **val** dateStr: String = **new** SimpleDateFormat(**"yyyyMMdd"**).format(**new** Date())  MyEsUtil.*bulkInsert*(orderInfoList, **"gmall2020\_order\_info\_"** + dateStr)  **for** ((id,orderInfo) <- orderInfoList ) {  MyKafkaSink.*send*(**"DW\_ORDER\_INFO"**,id,JSON.*toJSONString*(orderInfo,**new** SerializeConfig(**true**))) }  }  OffsetManager.*saveOffset*(groupId, topic, orderInfoOffsetRanges)    }    ssc.start()  ssc.awaitTermination()   }   } |

## 4 elasticsearch建立索引

|  |
| --- |
| PUT \_template/gmall1122\_order\_info\_template  {  "index\_patterns": ["gmall2020\_order\_info\*"],  "settings": {  "number\_of\_shards": 3  },  "aliases" : {  "{index}-query": {},  "gmall2020\_order\_info-query":{}  },  "mappings": {  "\_doc":{  "properties":{  "id":{  "type":"long"  },  "province\_id":{  "type":"long"  },  "order\_status":{  "type":"keyword"  },  "user\_id":{  "type":"long"  },  "final\_total\_amount":{  "type":"double"  },  "benefit\_reduce\_amount":{  "type":"double"  },  "original\_total\_amount":{  "type":"double"  },  "feight\_fee":{  "type":"double"  },  "expire\_time":{  "type":"keyword"  },  "create\_time":{  "type":"keyword"  },  "create\_date":{  "type":"date"  },  "create\_hour":{  "type":"keyword"  },  "if\_first\_order":{  "type":"keyword"  },  "province\_name":{  "type":"keyword"  },  "province\_area\_code":{  "type":"keyword"  },  "user\_age\_group":{  "type":"keyword"  },  "user\_gender":{  "type":"keyword"  }  }  }  }  } |