尚硅谷大数据技术之Flink电商实时数仓

DWD层数据准备

(作者：尚硅谷大数据研发部)

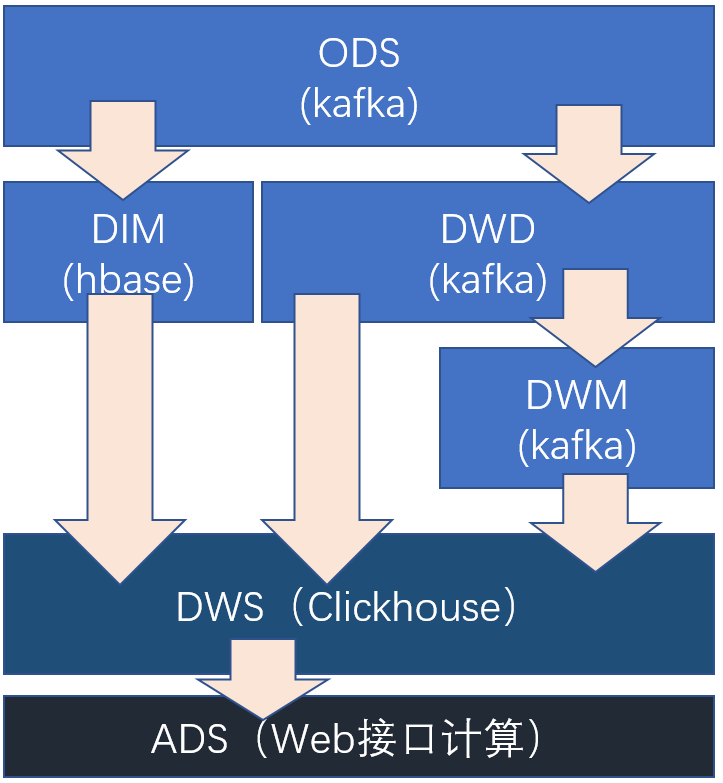
版本：V1.1

# 需求分析及实现思路

## 分层需求分析

在之前介绍实时数仓概念时讨论过，建设实时数仓的目的，主要是增加数据计算的复用性。每次新增加统计需求时，不至于从原始数据进行计算，而是从半成品继续加工而成。

我们这里从kafka的ods层读取用户行为日志以及业务数据，并进行简单处理，写回到kafka作为dwd层。



## 每层的职能

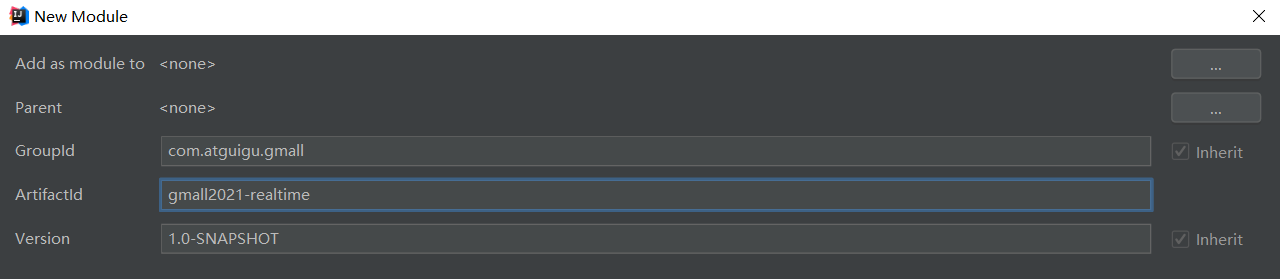
|  |  |  |  |
| --- | --- | --- | --- |
| 分层 | 数据描述 | 生成计算工具 | 存储媒介 |
| ODS | 原始数据，日志和业务数据 | 日志服务器，maxwell | kafka |
| DWD | 根据数据对象为单位进行分流，比如订单、页面访问等等。 | FLINK | kafka |
| DWM | 对于部分数据对象进行进一步加工，比如独立访问、跳出行为。依旧是明细数据。 | FLINK | kafka |
| DIM | 维度数据 | FLINK | HBase |
| DWS | 根据某个维度主题将多个事实数据轻度聚合，形成主题宽表。 | FLINK | Clickhouse |
| ADS | 把Clickhouse中的数据根据可视化需要进行筛选聚合。 | Clickhouse SQL | 可视化展示 |

## DWD层数据准备实现思路

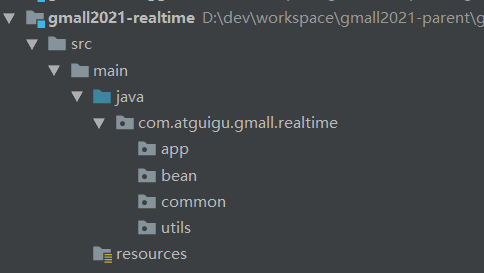
* 功能1：环境搭建
* 功能2：计算用户行为日志DWD层
* 功能3：计算业务数据DWD层

# 功能1：环境搭建

## 在工程中新建模块gmall2021-realtime



## 创建如下包结构



|  |  |
| --- | --- |
| 目录 | 作用 |
| app | 产生各层数据的flink任务 |
| bean | 数据对象 |
| common | 公共常量 |
| utils | 工具类 |

## 修改配置文件

### 在pom.xml添加如下配置

**<properties>**

**<java.version>1.8</java.version>  
 <maven.compiler.source>${java.version}</maven.compiler.source>  
 <maven.compiler.target>${java.version}</maven.compiler.target>  
 <flink.version>1.12.0</flink.version>  
 <scala.version>2.12</scala.version>**

**<hadoop.version>3.1.3</hadoop.version>  
</properties>  
  
<dependencies>  
 <dependency>  
 <groupId>org.apache.flink</groupId>  
 <artifactId>flink-java</artifactId>  
 <version>${flink.version}</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.apache.flink</groupId>  
 <artifactId>flink-streaming-java\_${scala.version}</artifactId>  
 <version>${flink.version}</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.apache.flink</groupId>  
 <artifactId>flink-connector-kafka\_${scala.version}</artifactId>  
 <version>${flink.version}</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.apache.flink</groupId>  
 <artifactId>flink-clients\_${scala.version}</artifactId>  
 <version>${flink.version}</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.apache.flink</groupId>  
 <artifactId>flink-cep\_${scala.version}</artifactId>  
 <version>${flink.version}</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.apache.flink</groupId>  
 <artifactId>flink-json</artifactId>  
 <version>${flink.version}</version>  
 </dependency>  
  
 <dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>fastjson</artifactId>  
 <version>1.2.68</version>  
 </dependency>**

**<!--如果保存检查点到hdfs上，需要引入此依赖-->**

**<dependency>  
 <groupId>org.apache.hadoop</groupId>  
 <artifactId>hadoop-client</artifactId>  
 <version>${hadoop.version}</version>  
 </dependency>**

**<!--Flink默认使用的是slf4j记录日志，相当于一个日志的接口,我们这里使用log4j作为具体的日志实现-->  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-api</artifactId>  
 <version>1.7.25</version>  
 </dependency>  
  
 <dependency>  
 <groupId>org.slf4j</groupId>  
 <artifactId>slf4j-log4j12</artifactId>  
 <version>1.7.25</version>  
 </dependency>**

**<dependency>  
 <groupId>org.apache.logging.log4j</groupId>  
 <artifactId>log4j-to-slf4j</artifactId>  
 <version>2.14.0</version>  
 </dependency>**

**</dependencies>  
  
<build>  
 <plugins>  
 <plugin>  
 <groupId>org.apache.maven.plugins</groupId>  
 <artifactId>maven-assembly-plugin</artifactId>  
 <version>3.0.0</version>  
 <configuration>  
 <descriptorRefs>  
 <descriptorRef>jar-with-dependencies</descriptorRef>  
 </descriptorRefs>  
 </configuration>  
 <executions>  
 <execution>  
 <id>make-assembly</id>  
 <phase>package</phase>  
 <goals>  
 <goal>single</goal>  
 </goals>  
 </execution>  
 </executions>  
 </plugin>  
 </plugins>  
</build>**

### 在resources目录下创建log4j.properties配置文件

**log4j.rootLogger=warn,stdout**

**log4j.appender.stdout=org.apache.log4j.ConsoleAppender  
log4j.appender.stdout.target=System.out  
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n**

# 功能2：准备用户行为日志DWD层

我们前面采集的日志数据已经保存到Kafka中，作为日志数据的ODS层，从kafka的ODS层读取的日志数据分为3类, 页面日志、启动日志和曝光日志。这三类数据虽然都是用户行为数据，但是有着完全不一样的数据结构，所以要拆分处理。将拆分后的不同的日志写回Kafka不同主题中，作为日志DWD层。

页面日志输出到主流,启动日志输出到启动侧输出流,曝光日志输出到曝光侧输出流

## 主要任务

### 识别新老用户

本身客户端业务有新老用户的标识，但是不够准确，需要用实时计算再次**确认(不涉及业务操作，只是单纯的做个状态确认)**。

### 利用侧输出流实现数据拆分

根据日志数据内容,将日志数据分为3类, 页面日志、启动日志和曝光日志。页面日志输出到主流,启动日志输出到启动侧输出流,曝光日志输出到曝光日志侧输出流

### 将不同流的数据推送下游的kafka的不同Topic中

## 代码实现

### 接收Kafka数据，并进行转换

#### 封装操作Kafka的工具类，并提供获取kafka消费者的方法（读）

**package com.atguigu.gmall.realtime.utils;**

**import org.apache.flink.api.common.serialization.SimpleStringSchema;  
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;  
import org.apache.kafka.clients.consumer.ConsumerConfig;  
import java.util.Properties;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 操作Kafka的工具类  
 \*/*public class MyKafkaUtil {  
 private static String *kafkaServer* = "hadoop202:9092,hadoop203:9092,hadoop204:9092";  
  
 //封装Kafka消费者  
 public static FlinkKafkaConsumer<String> getKafkaSource(String topic,String groupId){  
 Properties prop = new Properties();  
 prop.setProperty(ConsumerConfig.*GROUP\_ID\_CONFIG*,groupId);  
 prop.setProperty(ConsumerConfig.*BOOTSTRAP\_SERVERS\_CONFIG*,*kafkaServer*);  
 return new FlinkKafkaConsumer<String>(topic,new SimpleStringSchema(),prop);  
 }  
}**

#### Flink调用工具类读取数据的主程序

**package com.atguigu.gmall.realtime.app.dwd;**

**import com.alibaba.fastjson.JSON;  
import com.alibaba.fastjson.JSONObject;  
import com.atguigu.gmall.realtime.utils.MyKafkaUtil;  
import org.apache.flink.api.common.functions.MapFunction;  
import org.apache.flink.runtime.state.filesystem.FsStateBackend;  
import org.apache.flink.streaming.api.CheckpointingMode;  
import org.apache.flink.streaming.api.datastream.DataStreamSource;  
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;  
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 从Kafka中读取ods层用户行为日志数据  
 \*/*public class BaseLogApp {  
 //定义用户行为主题信息  
 private static final String *TOPIC\_START* ="dwd\_start\_log";  
 private static final String *TOPIC\_PAGE* ="dwd\_page\_log";  
 private static final String *TOPIC\_DISPLAY* ="dwd\_display\_log";  
  
 public static void main(String[] args) throws Exception {**

**//*TODO 0.基本环境准备***

**//创建Flink流处理执行环境**

**StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();**

**//设置并行度 这里和kafka分区数保持一致**

**env.setParallelism(4);**

**//设置CK相关的参数**

**//设置精准一次性保证（默认） 每5000ms开始一次checkpoint**

**env.enableCheckpointing(5000, CheckpointingMode.*EXACTLY\_ONCE*);**

**//Checkpoint必须在一分钟内完成，否则就会被抛弃**

**env.getCheckpointConfig().setCheckpointTimeout(60000);**

**env.setStateBackend(new FsStateBackend("hdfs://hadoop202:8020/gmall/flink/checkpoint"));**

**System.*setProperty*("HADOOP\_USER\_NAME","atguigu");**

**//指定消费者配置信息**

**String groupId = "ods\_dwd\_base\_log\_app";**

**String topic = "ods\_base\_log";**

**//*TODO 1.从kafka中读取数据***

**//调用Kafka工具类，从指定Kafka主题读取数据**

**FlinkKafkaConsumer<String> kafkaSource = MyKafkaUtil.*getKafkaSource*(topic, groupId);**

**DataStreamSource<String> kafkaDS = env.addSource(kafkaSource);**

**//转换为json对象**

**SingleOutputStreamOperator<JSONObject> jsonObjectDS = kafkaDS.map(**

**new MapFunction<String, JSONObject>() {**

**public JSONObject map(String value) throws Exception {**

**JSONObject jsonObject = JSON.*parseObject*(value);**

**return jsonObject;**

**}**

**}**

**);**

**//打印测试**

**jsonObjectDS.print();**

**//执行**

**env.execute("dwd\_base\_log Job");**

**}**

**}**

### 识别新老访客

保存每个mid的首次访问日期，每条进入该算子的访问记录，都会把mid对应的首次访问时间读取出来，跟当前日期进行比较，只有首次访问时间不为空，且首次访问时间早于当日的，则认为该访客是老访客，否则是新访客。

同时如果是新访客且没有访问记录的话，会写入首次访问时间。

**//*TODO 2.识别新老访客*//按照mid进行分组  
KeyedStream<JSONObject, String> midKeyedDS = jsonObjectDS.keyBy(  
 data -> data.getJSONObject("common").getString("mid"));  
  
//校验采集到的数据是新老访客  
SingleOutputStreamOperator<JSONObject> midWithNewFlagDS = midKeyedDS.map(  
 new RichMapFunction<JSONObject, JSONObject>() {  
 //声明第一次访问日期的状态  
 private ValueState<String> firstVisitDataState;  
 //声明日期数据格式化对象  
 private SimpleDateFormat simpleDateFormat;  
  
 @Override  
 public void open(Configuration parameters) throws Exception {  
 //初始化数据  
 firstVisitDataState = getRuntimeContext().getState(  
 new ValueStateDescriptor<String>("newMidDateState", String.class)  
 );  
 simpleDateFormat = new SimpleDateFormat("yyyyMMdd");  
 }  
  
 @Override  
 public JSONObject map(JSONObject jsonObj) throws Exception {  
 //打印数据  
 System.*out*.println(jsonObj);  
 //获取访问标记 0表示老访客 1表示新访客  
 String isNew = jsonObj.getJSONObject("common").getString("is\_new");  
 //获取数据中的时间戳  
 Long ts = jsonObj.getLong("ts");  
  
 //判断标记如果为"1",则继续校验数据  
 if ("1".equals(isNew)) {  
 //获取新访客状态  
 String newMidDate = firstVisitDataState.value();  
 //获取当前数据访问日期  
 String tsDate = simpleDateFormat.format(new Date(ts));  
  
 //如果新访客状态不为空,说明该设备已访问过 则将访问标记置为"0"  
 if (newMidDate != null && newMidDate.length()!=0) {**

**if(!newMidDate.equals(tsDate)){**

**isNew = "0";  
 jsonObj.getJSONObject("common").put("is\_new", isNew);**

**}  
 }else{  
 //如果复检后，该设备的确没有访问过，那么更新状态为当前日期  
 firstVisitDataState.update(tsDate);  
 }  
 }  
 //返回确认过新老访客的json数据  
 return jsonObj;  
 }  
 }  
);**

**//打印测试  
midWithNewFlagDS.print();**

### 利用侧输出流实现数据拆分

根据日志数据内容,将日志数据分为3类, 页面日志、启动日志和曝光日志。页面日志输出到主流,启动日志输出到启动侧输出流,曝光日志输出到曝光日志侧输出流

**//*TODO 3.利用侧输出流实现数据拆分***

**//定义启动和曝光数据的侧输出流标签  
OutputTag<String> startTag = new OutputTag<String>("start"){};  
OutputTag<String> displayTag = new OutputTag<String>("display"){};  
  
//日志页面日志、启动日志、曝光日志  
//将不同的日志输出到不同的流中 页面日志输出到主流,启动日志输出到启动侧输出流,曝光日志输出到曝光日志侧输出流  
SingleOutputStreamOperator<String> pageDStream = midWithNewFlagDS.process(  
 new ProcessFunction<JSONObject, String>() {  
 @Override  
 public void processElement(JSONObject jsonObj, Context ctx, Collector<String> out) throws Exception {  
 //获取数据中的启动相关字段  
 JSONObject startJsonObj = jsonObj.getJSONObject("start");  
 //将数据转换为字符串，准备向流中输出  
 String dataStr = jsonObj.toString();  
 //如果是启动日志，输出到启动侧输出流  
 if (startJsonObj != null && startJsonObj.size() > 0) {  
 ctx.output(startTag, dataStr);  
 } else {  
 //非启动日志,则为页面日志或者曝光日志(携带页面信息)  
 System.*out*.println("PageString:" + dataStr);**

**//将页面数据输出到主流  
 out.collect(dataStr);   
 //获取数据中的曝光数据,如果不为空,则将每条曝光数据取出输出到曝光日志侧输出流  
 JSONArray displays = jsonObj.getJSONArray("displays");  
 if (displays != null && displays.size() > 0) {  
 for (int i = 0; i < displays.size(); i++) {  
 JSONObject displayJsonObj = displays.getJSONObject(i);  
 //获取页面id  
 String pageId = jsonObj.getJSONObject("page").getString("page\_id");  
 //给每条曝光信息添加上pageId  
 displayJsonObj.put("page\_id", pageId);  
 //将曝光数据输出到测输出流  
 ctx.output(displayTag, displayJsonObj.toString());  
 }  
 }**

**}  
 }  
 }  
);  
  
//获取侧输出流  
DataStream<String> startDStream = pageDStream.getSideOutput(startTag);  
DataStream<String> displayDStream = pageDStream.getSideOutput(displayTag);  
  
//打印测试  
pageDStream.print("page");  
startDStream.print("start");  
displayDStream.print("display");**

### 将不同流的数据推送到下游kafka的不同Topic（分流）

#### 在MyKafkaUtil工具类中封装获取生产者的方法（写）

**//封装Kafka生产者**

**public static FlinkKafkaProducer<String> getKafkaSink(String topic) {  
 return new FlinkKafkaProducer<>(*kafkaServer*,topic,new SimpleStringSchema());  
}**

#### 程序中调用kafka工具类获取sink

**//打印测试**

**//pageDStream.print("page");  
//startDStream.print("start");  
//displayDStream.print("display");  
  
//*TODO 4.将数据输出到kafka不同的主题中*FlinkKafkaProducer<String> startSink = MyKafkaUtil.*getKafkaSink*(*TOPIC\_START*);  
FlinkKafkaProducer<String> pageSink = MyKafkaUtil.*getKafkaSink*(*TOPIC\_PAGE*);  
FlinkKafkaProducer<String> displaySink = MyKafkaUtil.*getKafkaSink*(*TOPIC\_DISPLAY*);  
  
startDStream.addSink(startSink);  
pageDStream.addSink(pageSink);  
displayDStream.addSink(displaySink);**

#### 测试

* Idea中运行DwdBaseLog类
* 运行logger.sh,启动Nginx以及日志处理服务
* 运行rt\_applog下模拟生成数据的jar包
* 到kafka不同的主题下查看输出效果

# 功能3：准备业务数据DWD层

业务数据的变化，我们可以通过Maxwell采集到，但是MaxWell是把全部数据统一写入一个Topic中, 这些数据包括业务数据，也包含维度数据，这样显然不利于日后的数据处理，所以这个功能是从Kafka的业务数据ODS层读取数据，经过处理后，将维度数据保存到Hbase，将事实数据写回Kafka作为业务数据的DWD层。

## 主要任务

### 接收Kafka数据，过滤空值数据

对Maxwell抓取数据进行ETL，有用的部分保留，没用的过滤掉

### 实现动态分流功能

由于MaxWell是把全部数据统一写入一个Topic中, 这样显然不利于日后的数据处理。所以需要把各个表拆开处理。但是由于每个表有不同的特点，有些表是维度表，有些表是事实表，有的表既是事实表在某种情况下也是维度表。

在实时计算中一般把维度数据写入存储容器，一般是方便通过主键查询的数据库比如HBase,Redis,MySQL等。一般把事实数据写入流中，进行进一步处理，最终形成宽表。但是作为Flink实时计算任务，如何得知哪些表是维度表，哪些是事实表呢？而这些表又应该采集哪些字段呢？

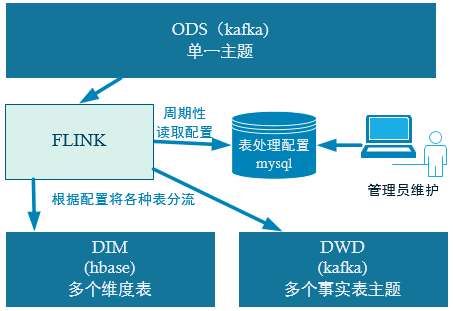
我们可以将上面的内容放到某一个地方，集中配置。这样的配置不适合写在配置文件中，因为业务端随着需求变化每增加一张表，就要修改配置重启计算程序。所以这里需要一种动态配置方案，把这种配置长期保存起来，一旦配置有变化，实时计算可以自动感知。

**这种可以有两个方案实现**

* 一种是用Zookeeper存储，通过Watch感知数据变化。
* 另一种是用mysql数据库存储，周期性的同步。

这里选择第二种方案，主要是mysql对于配置数据初始化和维护管理，用sql都比较方便，虽然周期性操作时效性差一点，但是配置变化并不频繁。

所以就有了如下图：



### 把分好的流保存到对应表、主题中

业务数据保存到Kafka的主题中

维度数据保存到Hbase的表中

## 代码实现

### 接收Kafka数据，过滤空值数据

**package com.atguigu.gmall.realtime.app.dwd;**

**import com.alibaba.fastjson.JSON;  
import com.alibaba.fastjson.JSONObject;  
import com.atguigu.gmall.realtime.utils.MyKafkaUtil;  
import org.apache.flink.runtime.state.filesystem.FsStateBackend;  
import org.apache.flink.streaming.api.CheckpointingMode;  
import org.apache.flink.streaming.api.datastream.DataStream;  
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;  
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 从Kafka中读取ods层业务数据 并进行处理 发送到DWD层  
 \*/*public class BaseDBApp {  
 public static void main(String[] args) throws Exception {  
 //*TODO 0.基本环境准备* //Flink流式处理环境  
 StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 env.setParallelism(4);  
  
 //设置CK相关参数  
 env.enableCheckpointing(5000, CheckpointingMode.*EXACTLY\_ONCE*);  
 env.getCheckpointConfig().setCheckpointTimeout(60000);  
 env.setStateBackend(new FsStateBackend("hdfs://hadoop202:8020/gmall/flink/checkpoint"));  
 System.*setProperty*("HADOOP\_USER\_NAME", "atguigu");  
  
 //*TODO 1.接收Kafka数据，过滤空值数据* //定义消费者组以及指定消费主题  
 String topic = "ods\_base\_db\_m";  
 String groupId = "ods\_base\_group";  
  
 //从Kafka主题中读取数据  
 FlinkKafkaConsumer<String> kafkaSource = MyKafkaUtil.*getKafkaSource*(topic,groupId);  
 DataStream<String> jsonDstream = env.addSource(kafkaSource);  
 //jsonDstream.print("data json:::::::");  
  
 //对数据进行结构的转换 String->JSONObject  
 DataStream<JSONObject> jsonStream = jsonDstream.map(jsonStr -> JSON.*parseObject*(jsonStr));  
 //DataStream<JSONObject> jsonStream = jsonDstream.map(JSON::parseObject);  
  
 //过滤为空或者 长度不足的数据  
 SingleOutputStreamOperator<JSONObject> filteredDstream = jsonStream.filter(  
 jsonObject -> {  
 boolean flag = jsonObject.getString("table") != null  
 && jsonObject.getJSONObject("data") != null  
 && jsonObject.getString("data").length() > 3;  
 return flag;  
 }) ;  
 filteredDstream.print("json::::::::");  
  
 env.execute();  
 }  
}**

### 根据MySQL的配置表，动态进行分流

#### 准备工作

导入依赖，创建表、JavaBean以及工具类

1. 引入pom.xml 依赖

**<!--lomback插件依赖-->**

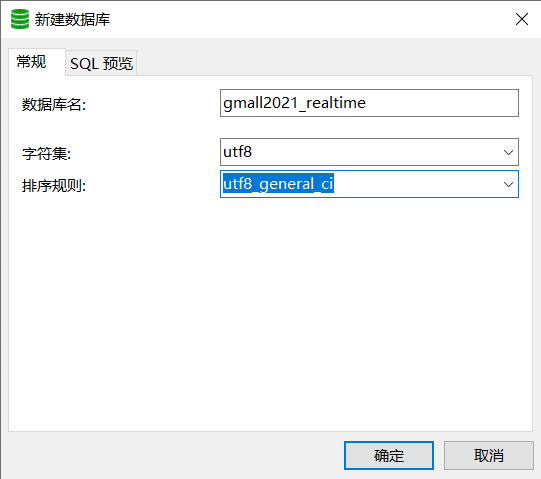
**<dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>1.18.12</version>  
 <scope>provided</scope>  
</dependency>**

**<!--commons-beanutils是Apache开源组织提供的用于操作JAVA BEAN的工具包。**

**使用commons-beanutils，我们可以很方便的对bean对象的属性进行操作-->  
<dependency>  
 <groupId>commons-beanutils</groupId>  
 <artifactId>commons-beanutils</artifactId>  
 <version>1.9.3</version>  
</dependency>  
<!--Guava工程包含了若干被Google的Java项目广泛依赖的核心库,方便开发-->  
<dependency>  
 <groupId>com.google.guava</groupId>  
 <artifactId>guava</artifactId>  
 <version>29.0-jre</version>  
</dependency>  
  
<dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>5.1.47</version>  
</dependency>**

1. 在Mysql中创建数据库

注意:和gmall2021业务库区分开



1. 在gmall2021\_realtime库中创建配置表table\_process

CREATE TABLE `table\_process` (  
  `source\_table` varchar(200) NOT NULL COMMENT '来源表',  
  `operate\_type` varchar(200) NOT NULL COMMENT '操作类型 insert,update,delete',  
   `sink\_type` varchar(200) DEFAULT NULL COMMENT '输出类型 hbase kafka',  
  `sink\_table` varchar(200) DEFAULT NULL COMMENT '输出表(主题)',  
  `sink\_columns` varchar(2000) DEFAULT NULL COMMENT '输出字段',  
  `sink\_pk` varchar(200) DEFAULT NULL COMMENT '主键字段',  
  `sink\_extend` varchar(200) DEFAULT NULL COMMENT '建表扩展',  
  PRIMARY KEY (`source\_table`,`operate\_type`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8

1. 创建配置表实体类

**package com.atguigu.gmall.realtime.bean;**

**import lombok.Data;  
  
@Data  
public class TableProcess {**

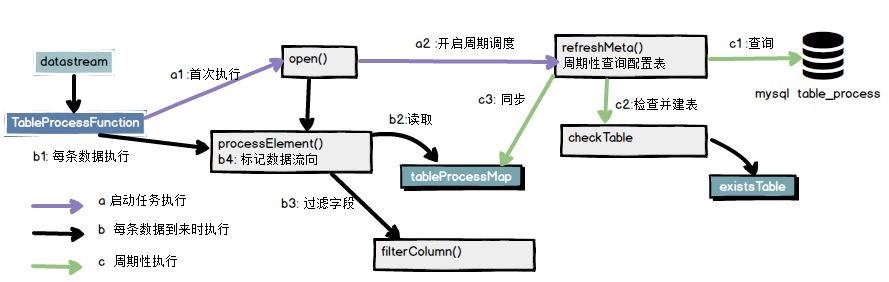
**//动态分流Sink常量 改为小写和脚本一致  
 public static final String *SINK\_TYPE\_HBASE* = "hbase";  
 public static final String *SINK\_TYPE\_KAFKA* = "kafka";  
 public static final String *SINK\_TYPE\_CK* = "clickhouse";  
 //来源表  
 String sourceTable;  
 //操作类型 insert,update,delete  
 String operateType;  
 //输出类型 hbase kafka  
 String sinkType;  
 //输出表(主题)  
 String sinkTable;  
 //输出字段  
 String sinkColumns;  
 //主键字段  
 String sinkPk;  
 //建表扩展  
 String sinkExtend;  
}**

1. 编写操作MySQL工具类

**package com.atguigu.gmall.realtime.utils;**

**import com.atguigu.gmall.realtime.bean.TableProcess;  
import com.google.common.base.CaseFormat;  
import org.apache.commons.beanutils.BeanUtils;  
  
import java.sql.\*;  
import java.util.ArrayList;  
import java.util.List;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 从MySQL读取数据的工具类  
 \*/*public class MySQLUtil {  
 */\*\*  
 \* mysql查询方法，根据给定的class类型 返回对应类型的元素列表  
 \*  
 \* @param sql  
 \* @param clazz  
 \* @param underScoreToCamel 是否把对应字段的下划线名转为驼峰名  
 \* @param* <*T*>  
 *\* @return  
 \*/* public static <T> List<T> queryList(String sql, Class<T> clazz, Boolean underScoreToCamel) {  
 Connection conn = null;  
 PreparedStatement ps = null;  
 ResultSet rs = null;  
 try {  
 //注册驱动  
 Class.*forName*("com.mysql.jdbc.Driver");  
 //建立连接  
 conn = DriverManager.*getConnection*(  
 "jdbc:mysql://hadoop202:3306/gmall2021\_realtime?characterEncoding=utf-8&useSSL=false",  
 "root",  
 "123456");  
 //创建数据库操作对象  
 ps = conn.prepareStatement(sql);  
 //执行SQL语句  
 rs = ps.executeQuery();  
 //处理结果集  
 ResultSetMetaData md = rs.getMetaData();  
 //声明集合对象，用于封装返回结果  
 List<T> resultList = new ArrayList<T>();  
  
 //每循环一次，获取一条查询结果  
 while (rs.next()) {  
 //通过反射创建要将查询结果转换为目标类型的对象  
 T obj = clazz.newInstance();  
 //对查询出的列进行遍历，每遍历一次得到一个列名  
 for (int i = 1; i <= md.getColumnCount(); i++) {  
 String propertyName = md.getColumnName(i);  
 //如果开启了下划线转驼峰的映射，那么将列名里的下划线转换为属性的打  
 if (underScoreToCamel) {  
 //直接调用Google的guava的CaseFormat LOWER\_UNDERSCORE小写开头+下划线->LOWER\_CAMEL小写开头+驼峰  
 propertyName = CaseFormat.*LOWER\_UNDERSCORE*.to(CaseFormat.*LOWER\_CAMEL*, propertyName);  
 }  
 //调用apache的commons-bean中的工具类，给Bean的属性赋值  
 BeanUtils.*setProperty*(obj, propertyName, rs.getObject(i));  
 }  
 resultList.add(obj);  
 }  
 return resultList;  
 } catch (Exception e) {  
 e.printStackTrace();  
 throw new RuntimeException("查询mysql失败！");  
 }finally {  
 //释放资源  
 if(rs!=null){  
 try {  
 rs.close();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
 if(ps!=null){  
 try {  
 ps.close();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
 if(conn!=null){  
 try {  
 conn.close();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
  
 */\*\*  
 \* 测试验证  
 \* @param args  
 \*/* public static void main(String[] args) {  
 List<TableProcess> tableProcesses = *queryList*("select \* from table\_process", TableProcess.class, true);  
 for (TableProcess tableProcess : tableProcesses) {  
 System.*out*.println(tableProcess);  
 }  
 }  
}**

#### 程序流程分析



**TableProcessFunction是一个自定义算子,主要包括三条时间线任务**

* 图中紫线，这个时间线与数据流入无关，只要任务启动就会执行。主要的任务方法是open()这个方法在任务启动时就会执行。他的主要工作就是初始化一些连接，开启周期调度。
* 图中绿线，这个时间线也与数据流入无关，只要周期调度启动，会自动周期性执行。主要的任务是同步配置表（tableProcessMap）。通过在open()方法中加入timer实现。同时还有个附带任务就是如果发现不存在数据表，要根据配置自动创建数据库表。
* 图中黑线，这个时间线就是随着数据的流入持续发生，这部分的任务就是根据同步到内存的tableProcessMap，来为流入的数据进行标识，同时清理掉没用的字段。

#### 自定义函数TableProcessFunction-基本信息定义

**package com.atguigu.gmall.realtime.app.func;**

**import com.alibaba.fastjson.JSONObject;  
import com.atguigu.gmall.realtime.bean.TableProcess;  
import org.apache.flink.streaming.api.functions.ProcessFunction;  
import org.apache.flink.util.OutputTag;  
import java.sql.Connection;  
import java.util.\*;  
*/\*\*  
 \* Author: Felix  
 \* Desc: 用于对业务数据进行分流处理的自定义处理函数  
 \*/*public class TableProcessFunction extends ProcessFunction<JSONObject, JSONObject> {  
  
 //因为要将维度数据写到侧输出流，所以定义一个侧输出流标签  
 private OutputTag<JSONObject> outputTag;  
  
 public TableProcessFunction(OutputTag<JSONObject> outputTag) {  
 this.outputTag = outputTag;  
 }  
  
 //用于在内存中存储表配置对象 [表名,表配置信息]  
 private Map<String, TableProcess> tableProcessMap = new HashMap<>();  
  
 //表示目前内存中已经存在的HBase表  
 private Set<String> existsTables = new HashSet<>();  
  
 //声明Phoenix连接  
 private Connection connection = null;**

**}**

#### 自定义函数TableProcessFunction-open

生命周期方法,初始化连接,初始化配置表信息并开启定时任务,用于不断读取配置表信息

**@Override**

**public void open(Configuration parameters) throws Exception {  
 //初始化Phoenix连接  
 Class.*forName*("org.apache.phoenix.jdbc.PhoenixDriver");  
 connection = DriverManager.*getConnection*(GmallConfig.*PHOENIX\_SERVER*);  
 //初始化配置表信息  
 initTableProcessMap();  
 //开启定时任务,用于不断读取配置表信息 从现在起过delay毫秒以后，每隔period更新一次  
 Timer timer = new Timer();  
 timer.schedule(new TimerTask() {  
 @Override  
 public void run() {  
 initTableProcessMap();  
 }  
 }, 5000, 5000);  
}**

#### 自定义函数TableProcessFunction-initTableProcessMap

读取MySQL中配置表信息，存入到内存Map中

**//读取MySQL中配置表信息，存入到内存Map中**

**private void initTableProcessMap() {  
 System.*out*.println("更新配置的处理信息");  
 //查询MySQL中的配置表数据  
 List<TableProcess> tableProcessList = MySQLUtil.*queryList*("select \* from table\_process", TableProcess.class, true);  
 //遍历查询结果,将数据存入结果集合  
 for (TableProcess tableProcess : tableProcessList) {  
 //获取源表表名  
 String sourceTable = tableProcess.getSourceTable();  
 //获取数据操作类型  
 String operateType = tableProcess.getOperateType();  
 //获取结果表表名  
 String sinkTable = tableProcess.getSinkTable();  
 //获取sink类型  
 String sinkType = tableProcess.getSinkType();  
 //拼接字段创建主键  
 String key = sourceTable + ":" + operateType;  
 //将数据存入结果集合  
 tableProcessMap.put(key, tableProcess);  
 //如果是向Hbase中保存的表，那么判断在内存中维护的Set集合中是否存在  
 if ("insert".equals(operateType) && "hbase".equals(sinkType)) {  
 boolean notExist = existsTables.add(sourceTable);  
 //如果表信息数据不存在内存,则在Phoenix中创建新的表  
 if (notExist) {  
 checkTable(sinkTable, tableProcess.getSinkColumns(), tableProcess.getSinkPk(), tableProcess.getSinkExtend());  
 }  
 }  
 }  
 if (tableProcessMap==null || tableProcessMap.size() == 0) {  
 throw new RuntimeException("缺少处理信息");  
 }  
}**

#### 自定义函数TableProcessFunction-checkTable

如果MySQL的配置表中添加了数据，该方法用于检查Hbase中是否创建过表，如果没有则通过Phoenix中创建新增的表

**private void checkTable(String tableName, String fields, String pk, String ext) {**

**//主键不存在,则给定默认值  
 if (pk == null) {  
 pk = "id";  
 }  
 //扩展字段不存在,则给定默认值  
 if (ext == null) {  
 ext = "";  
 }  
 //创建字符串拼接对象,用于拼接建表语句SQL  
 StringBuilder createSql = new StringBuilder("create table if not exists " + GmallConfig.*HBASE\_SCHEMA* + "." + tableName + "(");  
 //将列做切分,并拼接至建表语句SQL中  
 String[] fieldsArr = fields.split(",");  
 for (int i = 0; i < fieldsArr.length; i++) {  
 String field = fieldsArr[i];  
 if (pk.equals(field)) {  
 createSql.append(field).append(" varchar primary key ");  
 } else {  
 createSql.append("info.").append(field).append(" varchar");  
 }  
 if (i < fieldsArr.length - 1) {  
 createSql.append(",");  
 }  
 }  
 createSql.append(")");  
 createSql.append(ext);  
  
 try {  
 //执行建表语句在Phoenix中创建表  
 System.*out*.println(createSql);  
 PreparedStatement ps = connection.prepareStatement(createSql.toString());  
 ps.execute();  
 ps.close();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 throw new RuntimeException("建表失败！！！");  
 }  
}**

#### 定义一个项目中常用的配置常量类GmallConfig

**package com.atguigu.gmall.realtime.common;**

***/\*\*  
 \* Author: Felix  
 \* Desc: 项目配置常量类  
 \*/*public class GmallConfig {  
 public static final String *HBASE\_SCHEMA*="GMALL2021\_REALTIME";  
 public static final String *PHOENIX\_SERVER*="jdbc:phoenix:hadoop202,hadoop203,hadoop204:2181";  
}**

#### 自定义函数TableProcessFunction-filterColumn()

校验字段，过滤掉多余的字段

**private void filterColumn(JSONObject data, String sinkColumns) {**

**String[] cols = StringUtils.*split*(sinkColumns, ",");  
 Set<Map.Entry<String, Object>> entries = data.entrySet();  
 List<String> columnList = Arrays.*asList*(cols);  
 for (Iterator<Map.Entry<String, Object>> iterator = entries.iterator(); iterator.hasNext(); ) {  
 Map.Entry<String, Object> entry = iterator.next();  
 if (!columnList.contains(entry.getKey())) {  
 iterator.remove();  
 }  
 }  
}**

#### 自定义函数TableProcessFunction-processElement()

核心处理方法，根据MySQL配置表的信息为每条数据打标签，走kafka还是hbase

**@Override**

**public void processElement(JSONObject jsonObj, Context ctx, Collector<JSONObject> out) throws Exception {  
 String table = jsonObj.getString("table");  
 String type = jsonObj.getString("type");  
 JSONObject dataJsonObj = jsonObj.getJSONObject("data");  
  
 //如果是使用Maxwell的初始化功能，那么type类型为bootstrap-insert,我们这里也标记为insert，方便后续处理  
 if (type.equals("bootstrap-insert")) {**

**type = "insert";**

**jsonObj.put("type", type);**

**}  
  
 //获取配置表的信息  
 if (tableProcessMap != null && tableProcessMap.size() > 0) {  
 String key = table + ":" + type;  
 TableProcess tableProcess = tableProcessMap.get(key);  
 if (tableProcess != null) {  
 jsonObj.put("sink\_table", tableProcess.getSinkTable());  
 if (tableProcess.getSinkColumns() != null && tableProcess.getSinkColumns().length() > 0) {  
 filterColumn(jsonObj.getJSONObject("data"), tableProcess.getSinkColumns());  
 }  
 } else {  
 System.*out*.println("No This Key:" + key);  
 }  
 if (tableProcess != null && TableProcess.*SINK\_TYPE\_HBASE*.equalsIgnoreCase(tableProcess.getSinkType())) {  
 ctx.output(outputTag, jsonObj);  
 } else if (tableProcess != null && TableProcess.*SINK\_TYPE\_KAFKA*.equalsIgnoreCase(tableProcess.getSinkType())) {  
 out.collect(jsonObj);  
 }  
 }  
}**

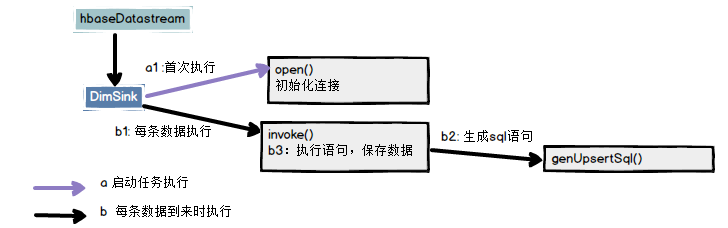
#### 主程序BaseDBApp中调用TableProcessFunction进行分流

**//*TODO 2.动态分流 事实表放入主流，作为DWD层；维度表放入侧输出流***

**//定义输出到Hbase的侧输出流标签  
OutputTag<JSONObject> hbaseTag = new OutputTag<JSONObject>(TableProcess.*SINK\_TYPE\_HBASE*) {};  
  
//使用自定义ProcessFunction进行分流处理  
SingleOutputStreamOperator<JSONObject> kafkaDStream = filteredDstream.process(new TableProcessFunction(hbaseTag));  
  
//获取侧输出流，即将通过Phoenix写到Hbase的数据  
DataStream<JSONObject> hbaseDStream = kafkaDStream.getSideOutput(hbaseTag);**

### 分流Sink之保存维度到HBase(Phoenix)

#### 程序流程分析



* DimSink 继承了RickSinkFunction，这个function得分两条时间线。
* 一条是任务启动时执行open操作（图中紫线），我们可以把连接的初始化工作放在此处一次性执行。
* 另一条是随着每条数据的到达反复执行invoke()（图中黑线）,在这里面我们要实现数据的保存，主要策略就是根据数据组合成sql提交给hbase。

#### 引入phoenix依赖

**<dependency>**

**<groupId>org.apache.phoenix</groupId>  
 <artifactId>phoenix-spark</artifactId>  
 <version>5.0.0-HBase-2.0</version>  
 <exclusions>  
 <exclusion>  
 <groupId>org.glassfish</groupId>  
 <artifactId>javax.el</artifactId>  
 </exclusion>  
 </exclusions>  
</dependency>**

#### 因为要用单独的schema,所以在Idea程序中加入hbase-site.xml

**<?xml version="1.0"?>**

**<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>  
  
<configuration>  
 <property>  
 <name>hbase.rootdir</name>  
 <value>hdfs://hadoop202:8020/hbase</value>  
 </property>  
  
 <property>  
 <name>hbase.cluster.distributed</name>  
 <value>true</value>  
 </property>  
  
 <property>  
 <name>hbase.zookeeper.quorum</name>  
 <value>hadoop202,hadoop203,hadoop204</value>  
 </property>  
  
 <property>  
 <name>hbase.unsafe.stream.capability.enforce</name>  
 <value>false</value>  
 </property>  
  
 <property>  
 <name>hbase.wal.provider</name>  
 <value>filesystem</value>  
 </property>  
  
 <property>  
 <name>phoenix.schema.isNamespaceMappingEnabled</name>  
 <value>true</value>  
 </property>  
  
 <property>  
 <name>phoenix.schema.mapSystemTablesToNamespace</name>  
 <value>true</value>  
 </property>  
  
</configuration>**

注意：为了开启hbase的namespace和phoenix的schema的映射，在程序中需要加这个配置文件，另外在linux服务上，也需要在hbase以及phoenix的hbase-site.xml配置文件中，加上以上两个配置，并使用xsync进行同步。

#### 在phoenix中执行

create schema GMALL2021\_REALTIME;

#### DimSink

**package com.atguigu.gmall.realtime.app.func;**

**import com.alibaba.fastjson.JSONObject;  
import com.atguigu.gmall.realtime.common.GmallConfig;  
import org.apache.commons.lang3.StringUtils;  
import org.apache.flink.configuration.Configuration;  
import org.apache.flink.streaming.api.functions.sink.RichSinkFunction;  
  
import java.sql.\*;  
import java.util.Set;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 通过Phoenix向Hbase表中写数据  
 \*/*public class DimSink extends RichSinkFunction<JSONObject> {  
  
 Connection connection = null;  
  
 @Override  
 public void open(Configuration parameters) throws Exception {  
 Class.*forName*("org.apache.phoenix.jdbc.PhoenixDriver");  
 connection = DriverManager.*getConnection*(GmallConfig.*PHOENIX\_SERVER*);  
 }  
  
 */\*\*  
 \* 生成语句提交hbase  
 \* @param jsonObject  
 \* @param context  
 \* @throws Exception  
 \*/* @Override  
 public void invoke(JSONObject jsonObject, Context context) throws Exception {  
  
 String tableName = jsonObject.getString("sink\_table");  
 JSONObject dataJsonObj = jsonObject.getJSONObject("data");  
 if (dataJsonObj != null && dataJsonObj.size() > 0) {  
 String upsertSql = genUpsertSql(tableName.toUpperCase(), jsonObject.getJSONObject("data"));  
 try {  
 System.*out*.println(upsertSql);  
 PreparedStatement ps = connection.prepareStatement(upsertSql);  
 ps.executeUpdate();  
 connection.commit();  
 ps.close();  
 } catch (Exception e) {  
 e.printStackTrace();  
 throw new RuntimeException("执行sql失败！");  
 }  
 }  
 }  
  
 public String genUpsertSql(String tableName, JSONObject jsonObject) {  
 Set<String> fields = jsonObject.keySet();  
 String upsertSql = "upsert into " + GmallConfig.*HBASE\_SCHEMA* + "." + tableName + "(" + StringUtils.*join*(fields, ",") + ")";  
 String valuesSql = " values ('" + StringUtils.*join*(jsonObject.values(), "','") + "')";  
 return upsertSql + valuesSql;  
 }  
}**

#### 主程序BaseDBApp中调用DimSink

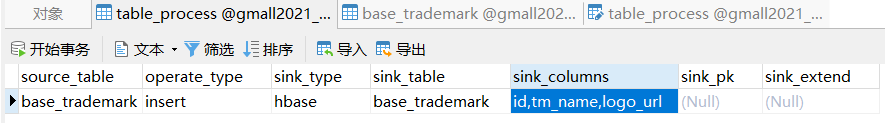
**//*TODO 3.将侧输出流数据写入HBase(Phoenix)***

**hbaseDStream.print("hbase::::");**

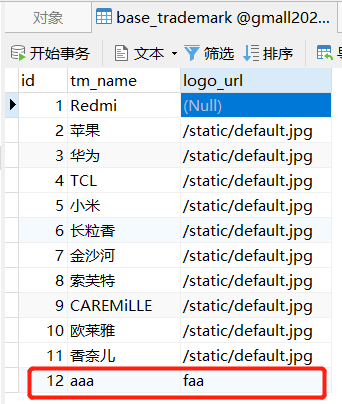
**hbaseDStream.addSink(new DimSink());**

#### 测试

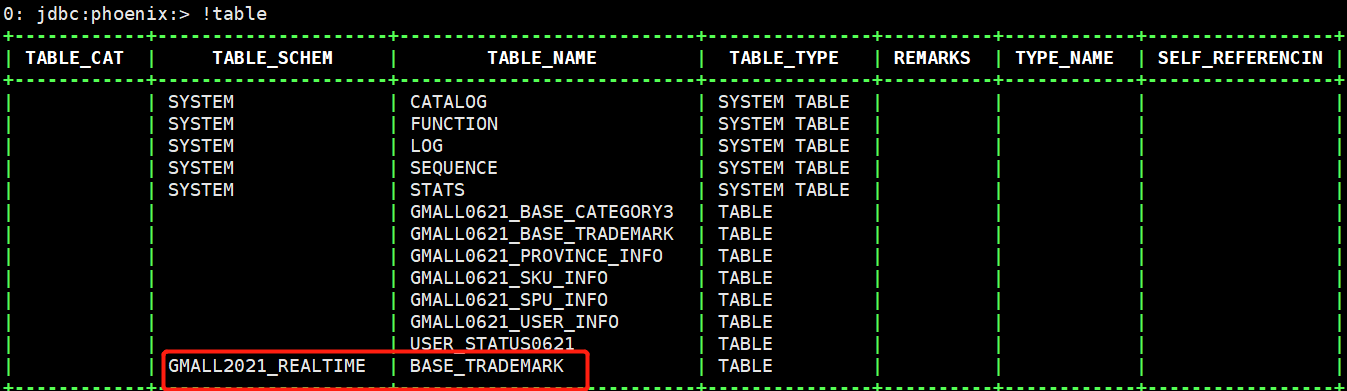
* 启动hdfs、zk、kafka、Maxwell、hbase
* 向gmall2021\_realtime数据库的table\_process表中插入测试数据

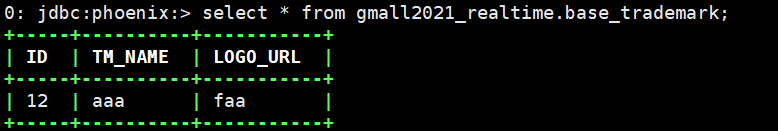


* 运行idea中的BaseDBApp
* 向gmall2021数据库的base\_trademark表中插入一条数据



* 通过phoenix查看hbase的schema以及表情况





### 分流Sink之保存业务数据到Kafka主题

#### 在MyKafkaUtil中添加如下方法

**//封装Kafka生产者 动态指定多个不同主题**

**public static <T> FlinkKafkaProducer<T> getKafkaSinkBySchema(KafkaSerializationSchema<T> serializationSchema) {  
 Properties prop =new Properties();  
 prop.setProperty(ProducerConfig.*BOOTSTRAP\_SERVERS\_CONFIG*,*kafkaServer*);  
 //如果15分钟没有更新状态，则超时 默认1分钟  
 prop.setProperty(ProducerConfig.*TRANSACTION\_TIMEOUT\_CONFIG*,1000\*60\*15+"");  
 return new FlinkKafkaProducer<>(*DEFAULT\_TOPIC*, serializationSchema, prop, FlinkKafkaProducer.Semantic.*EXACTLY\_ONCE*);  
}**

#### 在MyKafkaUtil中添加属性定义

**private static final String *DEFAULT\_TOPIC*="DEFAULT\_DATA";**

#### 两个创建FlinkKafkaProducer方法对比

* 前者给定确定的Topic
* 而后者除了缺省情况下会采用DEFAULT\_TOPIC，一般情况下可以根据不同的业务数据在KafkaSerializationSchema中通过方法实现。
* 可以查看一下FlinkKafkaProducer中的invoke方法源码

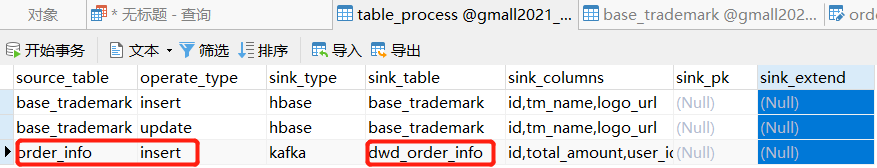
#### 在主程序BaseDBApp中加入新KafkaSink

**//*TODO 4将主流数据写入Kafka***

**FlinkKafkaProducer<JSONObject> kafkaSink = MyKafkaUtil.*getKafkaSinkBySchema*(new KafkaSerializationSchema<JSONObject>() {  
 @Override  
 public void open(SerializationSchema.InitializationContext context) throws Exception {  
 System.*out*.println("启动Kafka Sink");  
 }  
  
 //从每条数据得到该条数据应送往的主题名  
 @Override  
 public ProducerRecord<byte[], byte[]> serialize(JSONObject jsonObject, @Nullable Long aLong) {  
 String topic = jsonObject.getString("sink\_table");  
 JSONObject dataJsonObj = jsonObject.getJSONObject("data");  
 return new ProducerRecord(topic, dataJsonObj.toJSONString().getBytes());  
 }  
});  
kafkaDStream.print("kafka ::::");  
kafkaDStream.addSink(kafkaSink);**

#### 测试

* 启动hdfs、zk、kafka、Maxwell、hbase
* 向gmall2021\_realtime数据库的table\_process表中插入测试数据



* 运行idea中的BaseDBApp
* 运行rt\_dblog下的jar包，模拟生成数据
* 查看控制台输出以及在配置表中配置的kafka主题名消费情况

# 总结

DWD的实时计算核心就是数据分流，其次是状态识别。在开发过程中我们实践了几个灵活度较强算子，比如RichMapFunction, ProcessFunction, RichSinkFunction。 那这几个我们什么时候会用到呢？如何选择？

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Function | 可转换结构 | 可过滤数据 | 侧输出 | open方法 | 可以使用状态 | 输出至 |
| MapFunction | Yes | No | No | No | No | 下游算子 |
| FilterFunction | No | Yes | No | No | No | 下游算子 |
| RichMapFunction | Yes | No | No | Yes | Yes | 下游算子 |
| RichFilterFunction | No | Yes | No | Yes | Yes | 下游算子 |
| ProcessFunction | Yes | Yes | Yes | Yes | Yes | 下游算子 |
| SinkFunction | Yes | Yes | No | No | No | 外部 |
| RichSinkFunction | Yes | Yes | No | Yes | Yes | 外部 |

从对比表中能明显看出，Rich系列能功能强大，ProcessFunction功能更强大，但是相对的越全面的算子使用起来也更加繁琐。