尚硅谷大数据技术之Flink电商实时数仓

DWM层业务实现

(作者：尚硅谷大数据研发部)

版本：V 1.1

# DWS层与DWM层的设计

## 设计思路

我们在之前通过分流等手段，把数据分拆成了独立的kafka topic。那么接下来如何处理数据，就要思考一下我们到底要通过实时计算出哪些指标项。

因为实时计算与离线不同，实时计算的开发和运维成本都是非常高的，要结合实际情况考虑是否有必要象离线数仓一样，建一个大而全的中间层。

如果没有必要大而全，这时候就需要大体规划一下要实时计算出的指标需求了。把这些指标以主题宽表的形式输出就是我们的DWS层。

## 1.2 需求梳理

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 统计主题 | 需求指标 | 输出方式 | 计算来源 | 来源层级 |
| 访客 | pv | 可视化大屏 | page\_log直接可求 | dwd |
| uv | 可视化大屏 | 需要用page\_log过滤去重 | dwm |
| 跳出明细 | 可视化大屏 | 需要通过page\_log行为判断 | dwm |
| 进入页面数 | 可视化大屏 | 需要识别开始访问标识 | dwd |
| 连续访问时长 | 可视化大屏 | page\_log直接可求 | dwd |
| 商品 | 点击 | 多维分析 | page\_log直接可求 | dwd |
| 收藏 | 多维分析 | 收藏表 | dwd |
| 加入购物车 | 多维分析 | 购物车表 | dwd |
| 下单 | 可视化大屏 | 订单宽表 | dwm |
| 支付 | 多维分析 | 支付宽表 | dwm |
| 退款 | 多维分析 | 退款表 | dwd |
| 评论 | 多维分析 | 评论表 | dwd |
| 地区 | pv | 多维分析 | page\_log直接可求 | dwd |
| uv | 多维分析 | 需要用page\_log过滤去重 | dwm |
| 下单 | 可视化大屏 | 订单宽表 | dwm |
| 关键词 | 搜索关键词 | 可视化大屏 | 页面访问日志 直接可求 | dwd |
| 点击商品关键词 | 可视化大屏 | 商品主题下单再次聚合 | dws |
| 下单商品关键词 | 可视化大屏 | 商品主题下单再次聚合 | dws |

当然实际需求还会有更多，这里主要以为可视化大屏为目的进行实时计算的处理。

DWM层的定位是什么，DWM层主要服务DWS，因为部分需求直接从DWD层到DWS层中间会有一定的计算量，而且这部分计算的结果很有可能被多个DWS层主题复用，所以部分DWD层会形成一层DWM，我们这里主要涉及业务

* 访问UV计算
* 跳出明细计算
* 订单宽表
* 支付宽表

# DWM层-访客UV计算

## 需求分析与思路

UV，全称是Unique Visitor，即独立访客，对于实时计算中，也可以称为DAU(Daily Active User)，即每日活跃用户，因为实时计算中的uv通常是指当日的访客数。

那么如何从用户行为日志中识别出当日的访客，那么有两点：

* 其一，是识别出该访客打开的第一个页面，表示这个访客开始进入我们的应用
* 其二，由于访客可以在一天中多次进入应用，所以我们要在一天的范围内进行去重

## 代码实现

### 从kafka的dwd\_page\_log主题接收数据

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

**import com.alibaba.fastjson.JSON;  
import com.alibaba.fastjson.JSONObject;  
import com.atguigu.gmall.realtime.utils.MyKafkaUtil;  
import org.apache.flink.streaming.api.datastream.DataStream;  
import org.apache.flink.streaming.api.datastream.DataStreamSource;  
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;  
*/\*\*  
 \* Author: Felix  
 \* Desc: 访客UV的明细准备  
 \*/*public class UniqueVisitApp {  
 public static void main(String[] args) throws Exception{**

**//*TODO 0.基本环境准备*  
 StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 env.setParallelism(4);  
 //env.enableCheckpointing(5000, CheckpointingMode.EXACTLY\_ONCE);  
 //env.getCheckpointConfig().setCheckpointTimeout(60000);  
 //StateBackend fsStateBackend = new FsStateBackend("hdfs://hadoop202:8020/gmall/flink/checkpoint/UniqueVisitApp");  
 //env.setStateBackend(fsStateBackend);  
 //System.setProperty("HADOOP\_USER\_NAME", "atguigu");  
 //*TODO 1.从Kafka中读取数据*  
 String groupId = "unique\_visit\_app";  
 String sourceTopic = "dwd\_page\_log";  
 String sinkTopic = "dwm\_unique\_visit";  
  
 //读取kafka数据  
 FlinkKafkaConsumer<String> source = MyKafkaUtil.*getKafkaSource*(sourceTopic, groupId);  
 DataStreamSource<String> jsonStream = env.addSource(source);  
  
 //对读取的数据进行结构的转换  
 DataStream<JSONObject> jsonObjStream = jsonStream.map(jsonString -> JSON.*parseObject*(jsonString));  
  
 jsonObjStream.print("uv:");  
  
 env.execute();  
 }  
}**

**测试**

* 启动logger.sh、zk、kafka
* 运行Idea中的BaseLogApp
* 运行Idea中的UniqueVisitApp
* 查看控制台输出
* 执行流程

模拟生成数据->日志处理服务器->写到kafka的ODS层（ods\_base\_log）->BaseLogApp分流->dwd\_page\_log->UniqueVisitApp读取输出

### 核心的过滤代码

* 首先用keyby按照mid进行分组，每组表示当前设备的访问情况
* 分组后使用keystate状态，记录用户进入时间，实现RichFilterFunction完成过滤
* 重写open 方法用来初始化状态
* 重写filter方法进行过滤
* 可以直接筛掉last\_page\_id不为空的字段，因为只要有上一页，说明这条不是这个用户进入的首个页面。
* 状态用来记录用户的进入时间，只要这个lastVisitDate是今天，就说明用户今天已经访问过了所以筛除掉。如果为空或者不是今天，说明今天还没访问过，则保留。
* 因为状态值主要用于筛选是否今天来过，所以这个记录过了今天基本上没有用了，这里enableTimeToLive 设定了1天的过期时间，避免状态过大。

**//jsonObjStream.print("uv:");**

**//*TODO 2.核心的过滤代码*//按照设备id进行分组  
KeyedStream<JSONObject, String> keyByWithMidDstream =  
 jsonObjStream.keyBy(jsonObj -> jsonObj.getJSONObject("common").getString("mid"));  
  
SingleOutputStreamOperator<JSONObject> filteredJsonObjDstream =  
 keyByWithMidDstream.filter(new RichFilterFunction<JSONObject>() {  
 //定义状态用于存放最后访问的日期  
 ValueState<String> lastVisitDateState = null;  
 //日期格式  
 SimpleDateFormat simpleDateFormat = null;  
  
 //初始化状态 以及时间格式器  
 @Override  
 public void open(Configuration parameters) throws Exception {  
 simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd");  
 if (lastVisitDateState == null) {  
 ValueStateDescriptor<String> lastViewDateStateDescriptor =  
 new ValueStateDescriptor<>("lastViewDateState", String.class);  
 //因为统计的是当日UV，也就是日活，所有为状态设置失效时间  
 StateTtlConfig stateTtlConfig=StateTtlConfig.*newBuilder*(Time.*days*(1)).build();  
 //默认值 表明当状态创建或每次写入时都会更新时间戳  
 //.setUpdateType(StateTtlConfig.UpdateType.OnCreateAndWrite)  
 //默认值 一旦这个状态过期了，那么永远不会被返回给调用方，只会返回空状态  
 //.setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired).build();  
 lastViewDateStateDescriptor.enableTimeToLive(stateTtlConfig );  
 lastVisitDateState = getRuntimeContext().getState(lastViewDateStateDescriptor);  
 }  
 }  
  
 //首先检查当前页面是否有上页标识，如果有说明该次访问一定不是当日首次  
 @Override  
 public boolean filter(JSONObject jsonObject) throws Exception {  
 String lastPageId = jsonObject.getJSONObject("page").getString("last\_page\_id");  
 if(lastPageId!=null&&lastPageId.length()>0){  
 return false;  
 }   
 Long ts = jsonObject.getLong("ts");  
 String logDate = simpleDateFormat.format(ts);  
 String lastViewDate = lastVisitDateState.value();  
   
 if (lastViewDate!=null&&lastViewDate.length()>0&&logDate.equals(lastViewDate)){  
 System.*out*.println("已访问：lastVisit:"+lastViewDate+"|| logDate："+ logDate);  
 return false;  
 } else {  
 System.*out*.println("未访问：lastVisit:"+lastViewDate+"|| logDate："+logDate);  
 lastVisitDateState.update(logDate);  
 return true;  
 }  
 }  
 }).uid("uvFilter");  
  
SingleOutputStreamOperator<String> dataJsonStringDstream =  
 filteredJsonObjDstream.map(jsonObj -> jsonObj.toJSONString());  
dataJsonStringDstream.print("uv");**

### 将过滤处理后的UV写入到Kafka的dwm\_unique\_visit

**//*TODO 3.将过滤处理后的UV写入到Kafka的dwm主题***

**dataJsonStringDstream.addSink(MyKafkaUtil.*getKafkaSink*(sinkTopic));**

### 测试

* 启动logger.sh、zk、kafka
* 运行Idea中的BaseLogApp
* 运行Idea中的UniqueVisitApp
* 查看控制台输出以及kafka的dwm\_unique\_visit主题
* 执行流程

模拟生成数据->日志处理服务器->写到kafka的ODS层（ods\_base\_log）->BaseLogApp分流->dwd\_page\_log->UniqueVisitApp读取并处理->写回到kafka的dwm层

# DWM层-跳出明细计算

## 需求分析与思路

### 什么是跳出

跳出就是用户成功访问了网站的一个页面后就退出，不在继续访问网站的其它页面。而跳出率就是用跳出次数除以访问次数。

关注跳出率，可以看出引流过来的访客是否能很快的被吸引，渠道引流过来的用户之间的质量对比，对于应用优化前后跳出率的对比也能看出优化改进的成果。

### 计算跳出行为的思路

首先要识别哪些是跳出行为，要把这些跳出的访客最后一个访问的页面识别出来。那么要抓住几个特征：

* 该页面是用户近期访问的第一个页面

这个可以通过该页面是否有上一个页面（last\_page\_id）来判断，如果这个表示为空，就说明这是这个访客这次访问的第一个页面。

* 首次访问之后很长一段时间（自己设定），用户没继续再有其他页面的访问。

这第一个特征的识别很简单，保留last\_page\_id为空的就可以了。但是第二个访问的判断，其实有点麻烦，首先这不是用一条数据就能得出结论的，需要组合判断，要用一条存在的数据和不存在的数据进行组合判断。而且要通过一个不存在的数据求得一条存在的数据。更麻烦的他并不是永远不存在，而是在一定时间范围内不存在。那么如何识别有一定失效的组合行为呢？

最简单的办法就是Flink自带的CEP技术。这个CEP非常适合通过多条数据组合来识别某个事件。

用户跳出事件，本质上就是一个条件事件加一个超时事件的组合。

## 代码实现

### 从kafka的dwd\_page\_log主题中读取页面日志

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

**import com.alibaba.fastjson.JSON;  
import com.alibaba.fastjson.JSONObject;  
import com.atguigu.gmall.realtime.utils.MyKafkaUtil;  
import org.apache.flink.streaming.api.datastream.DataStream;  
import org.apache.flink.streaming.api.datastream.DataStreamSource;  
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 访客跳出情况判断  
 \*/*public class UserJumpDetailApp {  
 public static void main(String[] args) throws Exception {  
 //*TODO 0.基本环境准备* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 env.setParallelism(4);  
  
 //*TODO 1.从kafka的dwd\_page\_log主题中读取页面日志* String sourceTopic = "dwd\_page\_log";  
 String groupId = "userJumpDetailApp";  
 String sinkTopic = "dwm\_user\_jump\_detail";  
  
 //从kafka中读取数据  
 DataStreamSource<String> dataStream = env.addSource(MyKafkaUtil.*getKafkaSource*(sourceTopic, groupId));  
  
 //对数据进行结构的转换  
 DataStream<JSONObject> jsonObjStream = dataStream.map(jsonString -> JSON.*parseObject*(jsonString));  
   
 jsonObjStream.print("json:");  
   
 env.execute();  
 }  
}**

### 通过Flink的CEP完成跳出判断

#### 确认添加了CEP的依赖包

#### 设定时间语义为事件时间并指定数据中的ts字段为事件时间

由于这里涉及到时间的判断，所以必须设定数据流的EventTime和水位线。这里没有设置延迟时间，实际生产情况可以视乱序情况增加一些延迟。

增加延迟把forMonotonousTimestamps换为forBoundedOutOfOrderness即可。

注意：flink1.12默认的时间语义就是事件时间，所以不需要执行

env.setStreamTimeCharacteristic(TimeCharacteristic.***EventTime***);

**//*TODO 2.指定事件时间字段***

**SingleOutputStreamOperator<JSONObject> jsonObjWithEtDstream = jsonObjStream.assignTimestampsAndWatermarks(WatermarkStrategy.<JSONObject>*forMonotonousTimestamps*().withTimestampAssigner(new SerializableTimestampAssigner<JSONObject>() {  
 @Override  
 public long extractTimestamp(JSONObject jsonObject, long recordTimestamp) {  
 return jsonObject.getLong("ts");  
 }  
}));**

#### 根据日志数据的mid进行分组

因为用户的行为都是要基于相同的Mid的行为进行判断，所以要根据Mid进行分组。

**//*TODO 3.根据日志数据的mid进行分组***

**KeyedStream<JSONObject, String> jsonObjectStringKeyedStream = jsonObjWithEtDstream.keyBy(  
 jsonObj -> jsonObj.getJSONObject("common").getString("mid")  
);**

#### 配置CEP表达式

**//*TODO 4.配置CEP表达式***

**Pattern<JSONObject, JSONObject> pattern = Pattern.<JSONObject>*begin*("GoIn").where(  
 new SimpleCondition<JSONObject>() {  
 @Override // 条件1 ：进入的第一个页面  
 public boolean filter(JSONObject jsonObj) throws Exception {  
 String lastPageId = jsonObj.getJSONObject("page").getString("last\_page\_id");  
 System.*out*.println("first in :" + lastPageId);  
 if (lastPageId == null || lastPageId.length() == 0) {  
 return true;  
 }  
 return false;  
 }  
 }  
).next("next").where(  
 new SimpleCondition<JSONObject>() {  
 @Override //条件2： 在10秒时间范围内必须有第二个页面  
 public boolean filter(JSONObject jsonObj) throws Exception {  
 String pageId = jsonObj.getJSONObject("page").getString("page\_id");  
 System.*out*.println("next:" + pageId);  
 if (pageId != null && pageId.length() > 0) {  
 return true;  
 }  
 return false;  
 }  
 }  
).within(Time.*milliseconds*(10000));**

#### 根据表达式筛选流

**//*TODO 5.根据表达式筛选流***

**PatternStream<JSONObject> patternedStream = CEP.*pattern*(jsonObjectStringKeyedStream, pattern);**

#### 提取命中的数据

* 设定超时时间标识 timeoutTag
* flatSelect方法中，实现PatternFlatTimeoutFunction中的timeout方法。
* 所有out.collect的数据都被打上了超时标记
* 本身的flatSelect方法因为不需要未超时的数据所以不接受数据。
* 通过SideOutput侧输出流输出超时数据

**//*TODO 6.提取命中的数据***

**final OutputTag<String> timeoutTag = new OutputTag<String>("timeout") {};  
SingleOutputStreamOperator<String> filteredStream = patternedStream.flatSelect(  
 timeoutTag,  
 new PatternFlatTimeoutFunction<JSONObject, String>() {  
 @Override  
 public void timeout(Map<String, List<JSONObject>> pattern, long timeoutTimestamp, Collector<String> out) throws Exception {  
 List<JSONObject> objectList = pattern.get("GoIn");  
 //这里进入out的数据都被timeoutTag标记  
 for (JSONObject jsonObject : objectList) {  
 out.collect(jsonObject.toJSONString());  
 }  
 }  
 },  
 new PatternFlatSelectFunction<JSONObject, String>() {  
 @Override  
 public void flatSelect(Map<String, List<JSONObject>> pattern, Collector<String> out) throws Exception {  
 //因为不超时的事件不提取，所以这里不写代码  
 }  
 });  
//通过SideOutput侧输出流输出超时数据  
DataStream<String> jumpDstream = filteredStream.getSideOutput(timeoutTag);  
jumpDstream.print("jump::");**

### 将跳出数据写回到kafka的DWM层

**//*TODO 7.将跳出数据写回到kafka的DWM层***

**jumpDstream.addSink(MyKafkaUtil.*getKafkaSink*(sinkTopic));**

### 测试

#### 利用测试数据验证

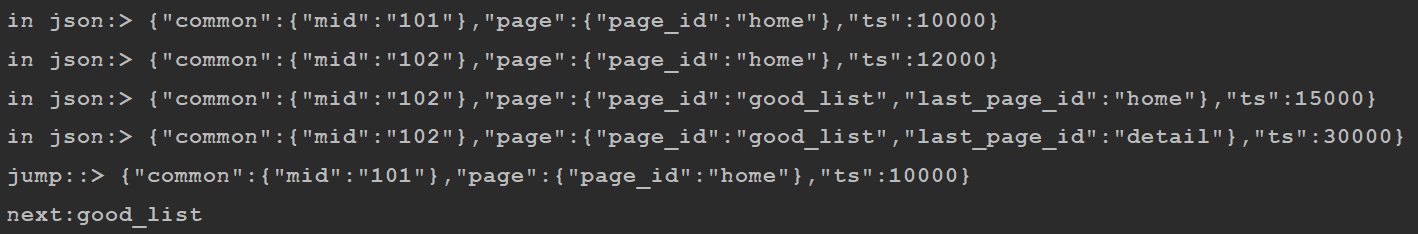
**DataStream<String> dataStream = env**

**.fromElements(  
 "{\"common\":{\"mid\":\"101\"},\"page\":{\"page\_id\":\"home\"},\"ts\":10000} ",  
 "{\"common\":{\"mid\":\"102\"},\"page\":{\"page\_id\":\"home\"},\"ts\":12000}",  
 "{\"common\":{\"mid\":\"102\"},\"page\":{\"page\_id\":\"good\_list\",\"last\_page\_id\":" +  
 "\"home\"},\"ts\":15000} ",  
 "{\"common\":{\"mid\":\"102\"},\"page\":{\"page\_id\":\"good\_list\",\"last\_page\_id\":" +  
 "\"detail\"},\"ts\":30000} "  
 );**

**dataStream.print("in json:");**

#### 查看控制台以及dwm\_user\_jump\_detail输出效果

注意：为了看效果，设置并行度为1



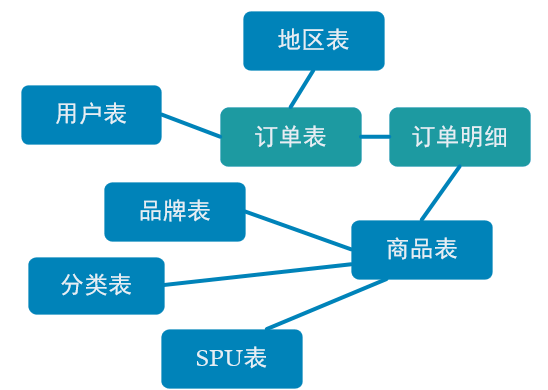
# DWM层-订单宽表

## 需求分析与思路

订单是统计分析的重要的对象，围绕订单有很多的维度统计需求，比如用户、地区、商品、品类、品牌等等。

为了之后统计计算更加方便，减少大表之间的关联，所以在实时计算过程中将围绕订单的相关数据整合成为一张订单的宽表。

那究竟哪些数据需要和订单整合在一起？



如上图，由于在之前的操作我们已经把数据分拆成了事实数据和维度数据，事实数据（绿色）进入kafka数据流（DWD层）中，维度数据（蓝色）进入hbase中长期保存。那么我们在DWM层中要把实时和维度数据进行整合关联在一起，形成宽表。那么这里就要处理有两种关联，事实数据和事实数据关联、事实数据和维度数据关联。

* 事实数据和事实数据关联，其实就是流与流之间的关联。
* 事实数据与维度数据关联，其实就是流计算中查询外部数据源。

## 订单和订单明细关联代码实现

### 从Kafka的dwd层接收订单和订单明细数据

#### 创建订单实体类

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

**import lombok.Data;  
import java.math.BigDecimal;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 订单实体类  
 \*/*@Data  
public class OrderInfo {  
  
 Long id;  
 Long province\_id;  
 String order\_status;  
 Long user\_id;  
 BigDecimal total\_amount;  
 BigDecimal activity\_reduce\_amount;  
 BigDecimal coupon\_reduce\_amount;  
 BigDecimal original\_total\_amount;  
 BigDecimal feight\_fee;  
 String expire\_time;  
 String create\_time;  
 String operate\_time;  
 String create\_date; // 把其他字段处理得到  
 String create\_hour;  
 Long create\_ts;  
}**

#### 创建订单明细实体类

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

**import lombok.Data;  
import java.math.BigDecimal;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc:订单明细实体类  
 \*/*@Data  
public class OrderDetail {  
 Long id;  
 Long order\_id ;  
 Long sku\_id;  
 BigDecimal order\_price ;  
 Long sku\_num ;  
 String sku\_name;  
 String create\_time;  
 BigDecimal split\_total\_amount;  
 BigDecimal split\_activity\_amount;  
 BigDecimal split\_coupon\_amount;  
 Long create\_ts;  
}**

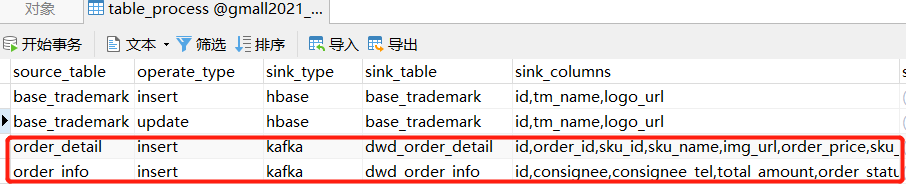
#### 在dwm包下创建OrderWideApp读取订单和订单明细数据

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

**import com.alibaba.fastjson.JSON;  
import com.atguigu.gmall.realtime.bean.OrderDetail;  
import com.atguigu.gmall.realtime.bean.OrderInfo;  
import com.atguigu.gmall.realtime.utils.MyKafkaUtil;  
import org.apache.flink.api.common.functions.RichMapFunction;  
import org.apache.flink.configuration.Configuration;  
import org.apache.flink.runtime.state.StateBackend;  
import org.apache.flink.runtime.state.filesystem.FsStateBackend;  
import org.apache.flink.streaming.api.CheckpointingMode;  
import org.apache.flink.streaming.api.TimeCharacteristic;  
import org.apache.flink.streaming.api.datastream.DataStream;  
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;  
  
import java.text.SimpleDateFormat;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 处理订单和订单明细数据形成订单宽表  
 \*/*public class OrderWideApp {  
 public static void main(String[] args) throws Exception {  
 //*TODO 0.基本环境准备* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 //设置并行度读取kafka分区数据  
 env.setParallelism(4);  
 /\*  
 //设置CK相关配置  
 env.enableCheckpointing(5000, CheckpointingMode.EXACTLY\_ONCE);  
 env.getCheckpointConfig().setCheckpointTimeout(60000);  
 StateBackend fsStateBackend = new FsStateBackend("hdfs://hadoop:8020/gmall/flink/checkpoint/OrderWideApp");  
 env.setStateBackend(fsStateBackend);  
 System.setProperty("HADOOP\_USER\_NAME", "atguigu");  
 \*/  
 //*TODO 1.从Kafka的dwd层接收订单和订单明细数据* String orderInfoSourceTopic = "dwd\_order\_info";  
 String orderDetailSourceTopic = "dwd\_order\_detail";  
 String orderWideSinkTopic = "dwm\_order\_wide";  
 String groupId = "order\_wide\_group";  
  
 //从Kafka中读取数据  
 FlinkKafkaConsumer<String> sourceOrderInfo = MyKafkaUtil.*getKafkaSource*(orderInfoSourceTopic,groupId);  
 FlinkKafkaConsumer<String> sourceOrderDetail = MyKafkaUtil.*getKafkaSource*(orderDetailSourceTopic,groupId);  
 DataStream<String> orderInfojsonDStream = env.addSource(sourceOrderInfo);  
 DataStream<String> orderDetailJsonDStream = env.addSource(sourceOrderDetail);  
  
  
 //对读取的数据进行结构的转换  
 DataStream<OrderInfo> orderInfoDStream = orderInfojsonDStream.map(  
 new RichMapFunction<String, OrderInfo>() {  
 SimpleDateFormat simpleDateFormat=null;  
 @Override  
 public void open(Configuration parameters) throws Exception {  
 super.open(parameters);  
 simpleDateFormat=new SimpleDateFormat("yyyy-MM-dd");  
 }  
 @Override  
 public OrderInfo map(String jsonString) throws Exception {  
 OrderInfo orderInfo = JSON.*parseObject*(jsonString, OrderInfo.class);  
 orderInfo.setCreate\_ts(simpleDateFormat.parse(orderInfo.getCreate\_time()).getTime());  
 return orderInfo;  
 }  
 }  
 );  
 DataStream<OrderDetail> orderDetailDStream = orderDetailJsonDStream.map(new RichMapFunction<String, OrderDetail>() {  
 SimpleDateFormat simpleDateFormat=null;  
 @Override  
 public void open(Configuration parameters) throws Exception {  
 super.open(parameters);  
 simpleDateFormat=new SimpleDateFormat("yyyy-MM-dd");  
 }  
 @Override  
 public OrderDetail map(String jsonString) throws Exception {  
 OrderDetail orderDetail = JSON.*parseObject*(jsonString, OrderDetail.class);  
 orderDetail.setCreate\_ts (simpleDateFormat.parse(orderDetail.getCreate\_time()).getTime());  
 return orderDetail;  
 }  
 });  
  
 orderInfoDStream.print("orderInfo::::");  
 orderDetailDStream.print("orderDetail::::");  
  
 env.execute();  
 }  
}**

#### 测试

* 启动Maxwell、zk、kafka、hdfs、hbase
* 运行Idea中的BaseDBApp
* 运行Idea中的OrderWideApp
* 在数据库gmall2021\_realtime的配置表中配置订单和订单明细



注意：会根据配置分流dwd层，dwd层还是保留的原始数据，所有我们这里sink\_columns的内容和数据库表中的字段保持一致，可以使用文本编辑工具处理。

* 执行rt\_dblog下的jar，生成模拟数据
* 查看控制台输出
* 执行流程

业务数据生成**->**Maxwell同步**->**Kafka的ods\_base\_db\_m主题**->**BaseDBApp分流写回kafka**->**dwd\_order\_info和dwd\_order\_detail**->**OrderWideApp从kafka的dwd层读数据，打印输出

### 订单和订单明细关联(双流join)

在flink中的流join大体分为两种，一种是基于时间窗口的join（Time Windowed Join），比如join、coGroup等。另一种是基于状态缓存的join（Temporal Table Join），比如intervalJoin。

这里选用intervalJoin，因为相比较窗口join，intervalJoin使用更简单，而且避免了应匹配的数据处于不同窗口的问题。intervalJoin目前只有一个问题，就是还不支持left join。

但是我们这里是订单主表与订单从表之间的关联不需要left join，所以intervalJoin是较好的选择。

#### 设定事件时间水位线

**//*TODO 2.设定事件时间水位***

**SingleOutputStreamOperator<OrderInfo> orderInfoWithEventTimeDstream = orderInfoDStream.assignTimestampsAndWatermarks(  
 WatermarkStrategy.<OrderInfo>*forMonotonousTimestamps*().withTimestampAssigner(new SerializableTimestampAssigner<OrderInfo>() {  
 @Override  
 public long extractTimestamp(OrderInfo orderInfo, long recordTimestamp) {  
 return orderInfo.getCreate\_ts();  
 }  
 })  
);  
  
SingleOutputStreamOperator<OrderDetail> orderDetailWithEventTimeDstream = orderDetailDStream.assignTimestampsAndWatermarks(  
 WatermarkStrategy.<OrderDetail>*forMonotonousTimestamps*().withTimestampAssigner(new SerializableTimestampAssigner<OrderDetail>() {  
  
 @Override  
 public long extractTimestamp(OrderDetail orderDetail, long recordTimestamp) {  
 return orderDetail.getCreate\_ts();  
 }  
 }));**

#### 创建合并后的宽表实体类

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

**import lombok.AllArgsConstructor;  
import lombok.Data;  
import org.apache.commons.lang3.ObjectUtils;  
  
import java.math.BigDecimal;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 订单和订单明细关联宽表对应实体类  
 \*/*@Data  
@AllArgsConstructor  
public class OrderWide {  
 Long detail\_id;  
 Long order\_id ;  
 Long sku\_id;  
 BigDecimal order\_price ;  
 Long sku\_num ;  
 String sku\_name;  
 Long province\_id;  
 String order\_status;  
 Long user\_id;  
  
 BigDecimal total\_amount;  
 BigDecimal activity\_reduce\_amount;  
 BigDecimal coupon\_reduce\_amount;  
 BigDecimal original\_total\_amount;  
 BigDecimal feight\_fee;  
 BigDecimal split\_feight\_fee;  
 BigDecimal split\_activity\_amount;  
 BigDecimal split\_coupon\_amount;  
 BigDecimal split\_total\_amount;  
  
 String expire\_time;  
 String create\_time;  
 String operate\_time;  
 String create\_date; // 把其他字段处理得到  
 String create\_hour;  
  
 String province\_name;//查询维表得到  
 String province\_area\_code;  
 String province\_iso\_code;  
 String province\_3166\_2\_code;  
  
 Integer user\_age ;  
 String user\_gender;  
  
 Long spu\_id; //作为维度数据 要关联进来  
 Long tm\_id;  
 Long category3\_id;  
 String spu\_name;  
 String tm\_name;  
 String category3\_name;  
  
 public OrderWide(OrderInfo orderInfo, OrderDetail orderDetail){  
 mergeOrderInfo(orderInfo);  
 mergeOrderDetail(orderDetail);  
  
 }  
  
 public void mergeOrderInfo(OrderInfo orderInfo ) {  
 if (orderInfo != null) {  
 this.order\_id = orderInfo.id;  
 this.order\_status = orderInfo.order\_status;  
 this.create\_time = orderInfo.create\_time;  
 this.create\_date = orderInfo.create\_date;  
 this.activity\_reduce\_amount = orderInfo.activity\_reduce\_amount;  
 this.coupon\_reduce\_amount = orderInfo.coupon\_reduce\_amount;  
 this.original\_total\_amount = orderInfo.original\_total\_amount;  
 this.feight\_fee = orderInfo.feight\_fee;  
 this.total\_amount = orderInfo.total\_amount;  
 this.province\_id = orderInfo.province\_id;  
 this.user\_id = orderInfo.user\_id;  
 }  
 }  
  
 public void mergeOrderDetail(OrderDetail orderDetail ) {  
 if (orderDetail != null) {  
 this.detail\_id = orderDetail.id;  
 this.sku\_id = orderDetail.sku\_id;  
 this.sku\_name = orderDetail.sku\_name;  
 this.order\_price = orderDetail.order\_price;  
 this.sku\_num = orderDetail.sku\_num;  
 this.split\_activity\_amount=orderDetail.split\_activity\_amount;  
 this.split\_coupon\_amount=orderDetail.split\_coupon\_amount;  
 this.split\_total\_amount=orderDetail.split\_total\_amount;  
 }  
 }  
  
 public void mergeOtherOrderWide(OrderWide otherOrderWide){  
 this.order\_status = ObjectUtils.*firstNonNull*( this.order\_status ,otherOrderWide.order\_status);  
 this.create\_time = ObjectUtils.*firstNonNull*(this.create\_time,otherOrderWide.create\_time);  
 this.create\_date = ObjectUtils.*firstNonNull*(this.create\_date,otherOrderWide.create\_date);  
 this.coupon\_reduce\_amount = ObjectUtils.*firstNonNull*(this.coupon\_reduce\_amount,otherOrderWide.coupon\_reduce\_amount);  
 this.activity\_reduce\_amount = ObjectUtils.*firstNonNull*(this.activity\_reduce\_amount,otherOrderWide.activity\_reduce\_amount);  
 this.original\_total\_amount = ObjectUtils.*firstNonNull*(this.original\_total\_amount,otherOrderWide.original\_total\_amount);  
 this.feight\_fee = ObjectUtils.*firstNonNull*( this.feight\_fee,otherOrderWide.feight\_fee);  
 this.total\_amount = ObjectUtils.*firstNonNull*( this.total\_amount,otherOrderWide.total\_amount);  
 this.user\_id = ObjectUtils.<Long>*firstNonNull*(this.user\_id,otherOrderWide.user\_id);  
 this.sku\_id = ObjectUtils.*firstNonNull*( this.sku\_id,otherOrderWide.sku\_id);  
 this.sku\_name = ObjectUtils.*firstNonNull*(this.sku\_name,otherOrderWide.sku\_name);  
 this.order\_price = ObjectUtils.*firstNonNull*(this.order\_price,otherOrderWide.order\_price);  
 this.sku\_num = ObjectUtils.*firstNonNull*( this.sku\_num,otherOrderWide.sku\_num);  
 this.split\_activity\_amount=ObjectUtils.*firstNonNull*(this.split\_activity\_amount);  
 this.split\_coupon\_amount=ObjectUtils.*firstNonNull*(this.split\_coupon\_amount);  
 this.split\_total\_amount=ObjectUtils.*firstNonNull*(this.split\_total\_amount);  
 }  
}**

#### 设定关联的key

**//*TODO 3.设定关联的key***

**KeyedStream<OrderInfo, Long> orderInfoKeyedDstream = orderInfoWithEventTimeDstream.keyBy(orderInfo -> orderInfo.getId());  
KeyedStream<OrderDetail, Long> orderDetailKeyedStream = orderDetailWithEventTimeDstream.keyBy(orderDetail -> orderDetail.getOrder\_id());**

#### 订单和订单明细关联 intervalJoin

这里设置了正负5秒，以防止在业务系统中主表与从表保存的时间差

**//*TODO 4.订单和订单明细关联 intervalJoin***

**SingleOutputStreamOperator<OrderWide> orderWideDstream = orderInfoKeyedDstream.intervalJoin(orderDetailKeyedStream)  
 .between(Time.*seconds*(-5), Time.*seconds*(5))  
 .process(new ProcessJoinFunction<OrderInfo, OrderDetail, OrderWide>() {  
 @Override  
 public void processElement(OrderInfo orderInfo, OrderDetail orderDetail, Context ctx, Collector<OrderWide> out) throws Exception {  
 out.collect(new OrderWide(orderInfo, orderDetail));  
 }  
 });  
  
orderWideDstream.print("joined ::");**

#### 测试

测试过程和上面测试读取数据过程一样

## 维表关联代码实现

维度关联实际上就是在流中查询存储在hbase中的数据表。但是即使通过主键的方式查询，hbase速度的查询也是不及流之间的join。外部数据源的查询常常是流式计算的性能瓶颈，所以咱们再这个基础上还有进行一定的优化。

### 先实现基本的维度查询功能

#### 封装Phoenix查询的工具类PhoenixUtil

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

**import com.alibaba.fastjson.JSONObject;  
import com.atguigu.gmall.realtime.common.GmallConfig;  
import org.apache.commons.beanutils.BeanUtils;  
import java.sql.\*;  
import java.util.ArrayList;  
import java.util.List;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 查询Phoenix的工具类  
 \*/*public class PhoenixUtil {  
 public static Connection *conn* = null;  
 public static void main(String[] args) {  
 List<JSONObject> objectList = *queryList*("select \* from base\_trademark", JSONObject.class);  
 System.*out*.println(objectList);  
 }  
  
  
 public static void queryInit() {  
 try {  
 Class.*forName*("org.apache.phoenix.jdbc.PhoenixDriver");  
 *conn* = DriverManager.*getConnection*(GmallConfig.*PHOENIX\_SERVER*);  
 *conn*.setSchema(GmallConfig.*HBASE\_SCHEMA*);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 }  
  
 public static <T> List<T> queryList(String sql, Class<T> clazz) {  
 if (*conn* == null) {  
 *queryInit*();  
 }  
 List<T> resultList = new ArrayList();  
 PreparedStatement ps = null;  
 try {  
 ps = *conn*.prepareStatement(sql);  
 ResultSet rs = ps.executeQuery();  
 ResultSetMetaData md = rs.getMetaData();  
 while (rs.next()) {  
 T rowData = clazz.newInstance();  
 for (int i = 1; i <= md.getColumnCount(); i++) {  
 BeanUtils.*setProperty*(rowData, md.getColumnName(i), rs.getObject(i));  
 }  
 resultList.add(rowData);  
 }  
 ps.close();  
 rs.close();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 return resultList;  
 }  
}**

#### 封装查询维度的工具类DimUtil（直接查询Phoenix）

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

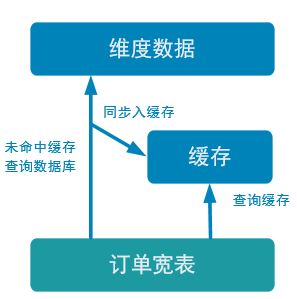
**import com.alibaba.fastjson.JSONObject;  
import org.apache.flink.api.java.tuple.Tuple2;  
import java.util.List;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 查询维度的工具类  
 \*/*public class DimUtil {  
 //直接从Phoenix查询，没有缓存  
 public static JSONObject getDimInfoNoCache(String tableName, Tuple2<String, String>... colNameAndValue) {  
 //组合查询条件  
 String wheresql = new String(" where ");  
 for (int i = 0; i < colNameAndValue.length; i++) {  
 //获取查询列名以及对应的值  
 Tuple2<String, String> nameValueTuple = colNameAndValue[i];  
 String fieldName = nameValueTuple.f0;  
 String fieldValue = nameValueTuple.f1;  
 if (i > 0) {  
 wheresql += " and ";  
 }  
 wheresql += fieldName + "='" + fieldValue + "'";  
 }  
 //组合查询SQL  
 String sql = "select \* from " + tableName + wheresql;  
 System.*out*.println("查询维度SQL:" + sql);  
 JSONObject dimInfoJsonObj = null;  
 List<JSONObject> dimList = PhoenixUtil.*queryList*(sql, JSONObject.class);  
 if (dimList != null && dimList.size() > 0) {  
 //因为关联维度，肯定都是根据key关联得到一条记录  
 dimInfoJsonObj = dimList.get(0);  
 }else{  
 System.*out*.println("维度数据未找到:" + sql);  
 }  
 return dimInfoJsonObj;  
 }  
  
 public static void main(String[] args) {  
 JSONObject dimInfooNoCache = DimUtil.*getDimInfooNoCache*("base\_trademark", Tuple2.*of*("id", "13"));  
 System.*out*.println(dimInfooNoCache);  
 }  
}**

#### 运行main方法测试

### 优化1：加入旁路缓存模式 （cache-aside-pattern）

我们在上面实现的功能中，直接查询的Hbase。外部数据源的查询常常是流式计算的性能瓶颈，所以我们需要在上面实现的基础上进行一定的优化。我们这里使用旁路缓存。

旁路缓存模式是一种非常常见的按需分配缓存的模式。如下图，任何请求优先访问缓存，缓存命中，直接获得数据返回请求。如果未命中则，查询数据库，同时把结果写入缓存以备后续请求使用。



#### 这种缓存策略有几个注意点

缓存要设过期时间，不然冷数据会常驻缓存浪费资源。

要考虑维度数据是否会发生变化，如果发生变化要主动清除缓存。

#### 缓存的选型

一般两种：堆缓存或者独立缓存服务(redis，memcache)，

堆缓存，从性能角度看更好，毕竟访问数据路径更短，减少过程消耗。但是管理性差，其他进程无法维护缓存中的数据。

独立缓存服务（redis,memcache）本事性能也不错，不过会有创建连接、网络IO等消耗。但是考虑到数据如果会发生变化，那还是独立缓存服务管理性更强，而且如果数据量特别大，独立缓存更容易扩展。

因为咱们的维度数据都是可变数据，所以这里还是采用Redis管理缓存。

#### 代码实现

##### 在pom.xml文件中添加Redis的依赖包

**<dependency>**

**<groupId>redis.clients</groupId>  
 <artifactId>jedis</artifactId>  
 <version>3.3.0</version>  
</dependency>**

##### 封装RedisUtil，通过连接池获得Jedis

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

**import redis.clients.jedis.Jedis;  
import redis.clients.jedis.JedisPool;  
import redis.clients.jedis.JedisPoolConfig;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 通过连接池获取Jedis的工具类  
 \*/*public class RedisUtil {  
   
 public static JedisPool *jedisPool*=null;  
 public static Jedis getJedis(){  
  
 if(*jedisPool*==null){  
 JedisPoolConfig jedisPoolConfig =new JedisPoolConfig();  
 jedisPoolConfig.setMaxTotal(100); //最大可用连接数  
 jedisPoolConfig.setBlockWhenExhausted(true); //连接耗尽是否等待  
 jedisPoolConfig.setMaxWaitMillis(2000); //等待时间  
  
 jedisPoolConfig.setMaxIdle(5); //最大闲置连接数  
 jedisPoolConfig.setMinIdle(5); //最小闲置连接数  
  
 jedisPoolConfig.setTestOnBorrow(true); //取连接的时候进行一下测试 ping pong  
  
 *jedisPool*=new JedisPool( jedisPoolConfig, "hadoop202",6379 ,1000);  
 System.*out*.println("开辟连接池");  
 return *jedisPool*.getResource();  
  
 }else{  
 System.*out*.println(" 连接池:"+*jedisPool*.getNumActive());  
 return *jedisPool*.getResource();  
 }  
 }  
}**

##### 在DimUtil中加入缓存，如果缓存没有再从的Phoenix查询

**//先从Redis中查，如果缓存中没有再通过Phoenix查询 固定id进行关联**

**public static JSONObject getDimInfo(String tableName, String id) {  
 Tuple2<String, String> kv = Tuple2.*of*("id", id);  
 return *getDimInfo*(tableName, kv);  
}  
  
//先从Redis中查，如果缓存中没有再通过Phoenix查询 可以使用其它字段灵活关联  
public static JSONObject getDimInfo(String tableName, Tuple2<String, String>... colNameAndValue) {  
 //组合查询条件  
 String wheresql = " where ";  
 String redisKey = "";  
 for (int i = 0; i < colNameAndValue.length; i++) {  
 Tuple2<String, String> nameValueTuple = colNameAndValue[i];  
 String fieldName = nameValueTuple.f0;  
 String fieldValue = nameValueTuple.f1;  
 if (i > 0) {  
 wheresql += " and ";  
 // 根据查询条件组合redis key ，  
 redisKey += "\_";  
 }  
 wheresql += fieldName + "='" + fieldValue + "'";  
 redisKey += fieldValue;  
 }  
   
 Jedis jedis = null;  
 String dimJson = null;  
 JSONObject dimInfo = null;  
 String key = "dim:" + tableName.toLowerCase() + ":" + redisKey;  
 try {  
 // 从连接池获得连接  
 jedis = RedisUtil.*getJedis*();  
 // 通过key查询缓存  
 dimJson = jedis.get(key);  
 } catch (Exception e) {  
 System.*out*.println("缓存异常！");  
 e.printStackTrace();  
 }  
   
 if (dimJson != null) {  
 dimInfo = JSON.*parseObject*(dimJson);  
 } else {  
 String sql = "select \* from " + tableName + wheresql;  
 System.*out*.println("查询维度sql:" + sql);  
 List<JSONObject> dimList = PhoenixUtil.*queryList*(sql, JSONObject.class);  
 if (dimList.size() > 0) {  
 dimInfo = dimList.get(0);  
 if (jedis != null) {  
 //把从数据库中查询的数据同步到缓存  
 jedis.setex(key, 3600 \* 24, dimInfo.toJSONString());  
 }  
 } else {  
 System.*out*.println("维度数据未找到：" + sql);  
 }  
 }  
 if (jedis != null) {  
 jedis.close();  
 System.*out*.println("关闭缓存连接 ");  
 }  
 return dimInfo;  
}**

##### 运行main方法测试和前面直接查询对比

使用缓存后，查询时间明显小于没有使用缓存之前

##### 在DimUtil中增加失效缓存的方法

维表数据变化时要失效缓存

**//根据key让Redis中的缓存失效**

**public static void deleteCached( String tableName, String id){  
 String key = "dim:" + tableName.toLowerCase() + ":" + id;  
 try {  
 Jedis jedis = RedisUtil.*getJedis*();  
 // 通过key清除缓存  
 jedis.del(key);  
 jedis.close();  
 } catch (Exception e) {  
 System.*out*.println("缓存异常！");  
 e.printStackTrace();  
 }  
}**

##### 修改DimSink的invoke方法

如果维度数据发生了变化，同时失效该数据对应的Redis中的缓存

**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！");  
 }  
 }  
 //如果维度数据发生变化，那么清空当前数据在Redis中的缓存  
 if(jsonObject.getString("type").equals("update")  
 ||jsonObject.getString("type").equals("delete")){  
 DimUtil.*deleteCached*(tableName,dataJsonObj.getString("id"));  
 }  
}**

##### 思考：应该先失效缓存还是先写入数据库，为什么？

先写入数据库，再失效缓存

##### 测试

* 启动Maxwell、ZK、Kafka、HDFS、Hbase、Redis
* 确定在Redis中存在某一个维度数据的缓存，如果没有运行DimUtil的main方法生成
* 运行Idea中的BaseDBApp
* 修改数据库gmall2021中的维度表和Redis缓存对应的数据，该数据会通过Maxwell同步到Kafka，然后BaseDBApp同步到Hbase的维度表中
* 查看Redis中的缓存是否被删除了

### 优化2：异步查询

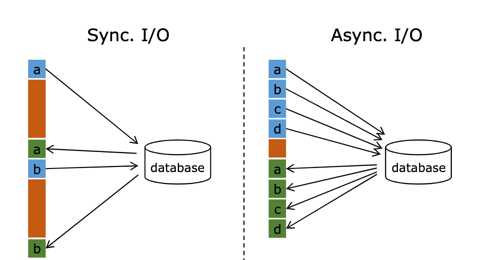
在Flink 流处理过程中，经常需要和外部系统进行交互，用维度表补全事实表中的字段。

例如：在电商场景中，需要一个商品的skuid去关联商品的一些属性，例如商品所属行业、商品的生产厂家、生产厂家的一些情况；在物流场景中，知道包裹id，需要去关联包裹的行业属性、发货信息、收货信息等等。

默认情况下，在Flink的MapFunction中，单个并行只能用同步方式去交互: 将请求发送到外部存储，IO阻塞，等待请求返回，然后继续发送下一个请求。这种同步交互的方式往往在网络等待上就耗费了大量时间。为了提高处理效率，可以增加MapFunction的并行度，但增加并行度就意味着更多的资源，并不是一种非常好的解决方式。

Flink 在1.2中引入了Async I/O，在异步模式下，将IO操作异步化，单个并行可以连续发送多个请求，哪个请求先返回就先处理，从而在连续的请求间不需要阻塞式等待，大大提高了流处理效率。

Async I/O 是阿里巴巴贡献给社区的一个呼声非常高的特性，解决与外部系统交互时网络延迟成为了系统瓶颈的问题。



异步查询实际上是把维表的查询操作托管给单独的线程池完成，这样不会因为某一个查询造成阻塞，单个并行可以连续发送多个请求，提高并发效率。

这种方式特别针对涉及网络IO的操作，减少因为请求等待带来的消耗。

#### 封装线程池工具类

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

**import java.util.concurrent.LinkedBlockingDeque;  
import java.util.concurrent.ThreadPoolExecutor;  
import java.util.concurrent.TimeUnit;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 线程池工具类  
 \*/*public class ThreadPoolUtil {  
 public static ThreadPoolExecutor *pool*;  
  
 /\*** **获取单例的线程池对象  
 corePoolSize:指定了线程池中的线程数量，它的数量决定了添加的任务是开辟新的线程去执行，还是放到workQueue任务队列中去；  
 maximumPoolSize:指定了线程池中的最大线程数量，这个参数会根据你使用的workQueue任务队列的类型，决定线程池会开辟的最大线程数量；  
 keepAliveTime:当线程池中空闲线程数量超过corePoolSize时，多余的线程会在多长时间内被销毁；  
 unit:keepAliveTime的单位  
 workQueue:任务队列，被添加到线程池中，但尚未被执行的任务  
 \*/  
 public static ThreadPoolExecutor getInstance() {  
 if (*pool* == null) {  
 synchronized (ThreadPoolUtil.class) {  
 if (*pool* == null) {  
 System.*out*.println ("开辟程池！！！！！");  
 *pool*=new ThreadPoolExecutor(4, 20, 300, TimeUnit.*SECONDS*,  
 new** **LinkedBlockingDeque<Runnable>(Integer.*MAX\_VALUE*));  
 }  
 }  
 }  
 return *pool*;  
 }  
}**

#### 自定义维度查询接口

这个异步维表查询的方法适用于各种维表的查询，用什么条件查，查出来的结果如何合并到数据流对象中，需要使用者自己定义。

这就是自己定义了一个接口DimJoinFunction<T>包括两个方法。

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

**import com.alibaba.fastjson.JSONObject;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 维度关联查询的接口  
 \*/*public interface DimJoinFunction<T> {  
 */\*\*  
 \* 需要实现如何把结果装配给数据流对象  
 \* @param t 数据流对象  
 \* @param jsonObject 异步查询结果  
 \* @throws Exception  
 \*/* public void join(T t , JSONObject jsonObject) throws Exception;  
  
 */\*\*  
 \* 需要实现如何从流中对象获取主键  
 \* @param t 数据流对象  
 \*/* public String getKey(T t);  
}**

#### 封装维度异步查询的函数类DimAsyncFunction

该类继承异步方法类RichAsyncFunction，实现自定义维度查询接口

其中RichAsyncFunction<IN,OUT>是Flink提供的异步方法类，此处因为是查询操作输入类和返回类一致，所以是<T,T>。

RichAsyncFunction这个类要实现两个方法:

open用于初始化异步连接池。

asyncInvoke方法是核心方法，里面的操作必须是异步的，如果你查询的数据库有异步api也可以用线程的异步方法，如果没有异步方法，就要自己利用线程池等方式实现异步查询。

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

**import com.alibaba.fastjson.JSONObject;  
import org.apache.flink.configuration.Configuration;  
import org.apache.flink.streaming.api.functions.async.ResultFuture;  
import org.apache.flink.streaming.api.functions.async.RichAsyncFunction;  
import java.util.Arrays;  
import java.util.concurrent.ExecutorService;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc: 自定义维度查询异步执行函数  
 \* RichAsyncFunction： 里面的方法负责异步查询  
 \* DimJoinFunction： 里面的方法负责将为表和主流进行关联  
 \*/*public abstract class DimAsyncFunction<T> extends RichAsyncFunction<T,T> implements DimJoinFunction<T>{  
  
 ExecutorService executorService = null;  
  
 public String tableName=null;  
  
  
 public DimAsyncFunction(String tableName){  
 this.tableName=tableName;  
 }  
  
 public void open(Configuration parameters ) {  
 System.*out*.println ("获得线程池！ ");  
 executorService = ThreadPoolUtil.*getInstance*() ;  
 }  
  
 @Override  
 public void asyncInvoke(T obj,** **ResultFuture<T> resultFuture) throws Exception {  
 executorService.submit(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 long start = System.*currentTimeMillis*();  
 //从流对象中获取主键  
 String key = getKey(obj);  
 //根据主键获取维度对象数据  
 JSONObject dimJsonObject = DimUtil.*getDimInfo*(tableName,key);  
 System.*out*.println("dimJsonObject:"+dimJsonObject);  
 if(dimJsonObject!=null){  
 //维度数据和流数据关联  
 join(obj,dimJsonObject) ;  
 }  
 System.*out*.println("obj:"+obj);  
 long end = System.*currentTimeMillis*();  
 System.*out*.println("异步耗时："+(end-start)+"毫秒");  
 resultFuture.complete(Arrays.*asList*(obj));  
 } catch ( Exception e) {  
 System.*out*.println(String.*format*(tableName+"异步查询异常. %s", e));  
 e.printStackTrace();  
 }  
 }  
 });  
 }  
}**

#### 如何使用这个DimAsyncFunction

核心的类是AsyncDataStream，这个类有两个方法一个是有序等待（orderedWait），一个是无序等待（unorderedWait）。

* **无序等待（unorderedWait）**

后来的数据，如果异步查询速度快可以超过先来的数据，这样性能会更好一些，但是会有乱序出现。

* **有序等待**（orderedWait）

严格保留先来后到的顺序，所以后来的数据即使先完成也要等前面的数据。所以性能会差一些。

* **注意**
  + 这里实现了用户维表的查询，那么必须重写装配结果join方法和获取查询rowkey的getKey方法。
  + 方法的最后两个参数10, TimeUnit.***SECONDS*** ，标识次异步查询最多执行10秒，否则会报超时异常。

##### 关联用户维度（在OrderWideApp中）

**//*TODO 5.关联用户维度***

**SingleOutputStreamOperator<OrderWide> orderWideWithUserDstream = AsyncDataStream.*unorderedWait*(  
 orderWideDstream, new DimAsyncFunction<OrderWide>("DIM\_USER\_INFO") {  
 @Override  
 public void join(OrderWide orderWide, JSONObject jsonObject) throws Exception {  
 SimpleDateFormat formattor = new SimpleDateFormat("yyyy-MM-dd");  
 String birthday = jsonObject.getString("BIRTHDAY");  
 Date date = formattor.parse(birthday);  
  
 Long curTs = System.*currentTimeMillis*();  
 Long betweenMs = curTs - date.getTime();  
 Long ageLong = betweenMs / 1000L / 60L / 60L / 24L / 365L;  
 Integer age = ageLong.intValue();  
 orderWide.setUser\_age(age);  
 orderWide.setUser\_gender(jsonObject.getString("GENDER"));  
 }  
  
 @Override  
 public String getKey(OrderWide orderWide) {  
 return String.*valueOf*(orderWide.getUser\_id());  
 }  
 }, 60, TimeUnit.*SECONDS*);  
  
 orderWideWithUserDstream.print("dim join user:");**

* 测试用户维度关联
* 将table\_process表中的数据删除掉，执行2.资料的table\_process初始配置.sql
* 启动Maxwell、ZK、Kafka、HDFS、Hbase、Redis
* 运行运行Idea中的BaseDBApp
* 初始化用户维度数据到Hbase（通过Maxwell的Bootstrap）

bin/maxwell-bootstrap --user maxwell --password 123456 --host hadoop202 --database gmall2021 --table user\_info --client\_id maxwell\_1

* 运行Idea中的OrderWideApp
* 执行模拟生成业务数据的jar包
* 查看控制台输出可以看到用户的年龄以及性别

##### 关联省市维度

**//*TODO 6.关联省市维度***

**SingleOutputStreamOperator<OrderWide> orderWideWithProvinceDstream = AsyncDataStream.*unorderedWait*(  
 orderWideWithUserDstream, new DimAsyncFunction<OrderWide>("DIM\_BASE\_PROVINCE") {  
 @Override  
 public void join(OrderWide orderWide, JSONObject jsonObject) throws Exception {  
 orderWide.setProvince\_name(jsonObject.getString("NAME"));  
 orderWide.setProvince\_3166\_2\_code(jsonObject.getString("ISO\_3166\_2"));  
 orderWide.setProvince\_iso\_code(jsonObject.getString("ISO\_CODE"));  
 orderWide.setProvince\_area\_code(jsonObject.getString("AREA\_CODE"));  
 }  
  
 @Override  
 public String getKey(OrderWide orderWide) {  
 return String.*valueOf*(orderWide.getProvince\_id());  
 }  
 }, 60, TimeUnit.*SECONDS*);**

* 初始化省市维度数据到Hbase（通过Maxwell的Bootstrap）

bin/maxwell-bootstrap --user maxwell --password 123456 --host hadoop202 --database gmall2021 --table base\_province --client\_id maxwell\_1

* 测试省市维度关联

##### 关联SKU维度

**//*TODO 7.关联SKU维度***

**SingleOutputStreamOperator<OrderWide> orderWideWithSkuDstream = AsyncDataStream.*unorderedWait*(  
 orderWideWithProvinceDstream, new DimAsyncFunction<OrderWide>("DIM\_SKU\_INFO") {  
 @Override  
 public void join(OrderWide orderWide, JSONObject jsonObject) throws Exception {  
 orderWide.setSku\_name(jsonObject.getString("SKU\_NAME"));  
 orderWide.setCategory3\_id(jsonObject.getLong("CATEGORY3\_ID"));  
 orderWide.setSpu\_id(jsonObject.getLong("SPU\_ID"));  
 orderWide.setTm\_id(jsonObject.getLong("TM\_ID"));  
 }  
  
 @Override  
 public String getKey(OrderWide orderWide) {  
 return String.*valueOf*(orderWide.getSku\_id());  
 }  
 }, 60, TimeUnit.*SECONDS*);**

* 初始化SKU维度数据到Hbase（通过Maxwell的Bootstrap）

bin/maxwell-bootstrap --user maxwell --password 123456 --host hadoop202 --database gmall2021 --table sku\_info --client\_id maxwell\_1

* 测试SKU维度关联

##### 关联SPU维度

**//*TODO 8.关联SPU商品维度***

**SingleOutputStreamOperator<OrderWide> orderWideWithSpuDstream = AsyncDataStream.*unorderedWait*(  
 orderWideWithSkuDstream, new DimAsyncFunction<OrderWide>("DIM\_SPU\_INFO") {  
 @Override  
 public void join(OrderWide orderWide, JSONObject jsonObject) throws Exception {  
 orderWide.setSpu\_name(jsonObject.getString("SPU\_NAME"));  
 }  
  
 @Override  
 public String getKey(OrderWide orderWide) {  
 return String.*valueOf*(orderWide.getSpu\_id());  
 }  
 }, 60, TimeUnit.*SECONDS*);**

* 初始化SPU维度数据到Hbase（通过Maxwell的Bootstrap）

bin/maxwell-bootstrap --user maxwell --password 123456 --host hadoop202 --database gmall2021 --table spu\_info --client\_id maxwell\_1

* 测试SPU维度关联

##### 关联品类维度

**//*TODO 9.关联品类维度***

**SingleOutputStreamOperator<OrderWide> orderWideWithCategory3Dstream = AsyncDataStream.*unorderedWait*(  
 orderWideWithSpuDstream, new DimAsyncFunction<OrderWide>("DIM\_BASE\_CATEGORY3") {  
 @Override  
 public void join(OrderWide orderWide, JSONObject jsonObject) throws Exception {  
 orderWide.setCategory3\_name(jsonObject.getString("NAME"));  
 }  
  
 @Override  
 public String getKey(OrderWide orderWide) {  
 return String.*valueOf*(orderWide.getCategory3\_id());  
 }  
 }, 60, TimeUnit.*SECONDS*);**

* 初始化品类维度数据到Hbase（通过Maxwell的Bootstrap）

bin/maxwell-bootstrap --user maxwell --password 123456 --host hadoop202 --database gmall2021 --table base\_category3 --client\_id maxwell\_1

* 测试品类维度关联

##### 关联品牌维度

//*TODO 10.关联品牌维度*

SingleOutputStreamOperator<OrderWide> orderWideWithTmDstream = AsyncDataStream.*unorderedWait*(  
 orderWideWithCategory3Dstream, new DimAsyncFunction<OrderWide>("DIM\_BASE\_TRADEMARK") {  
 @Override  
 public void join(OrderWide orderWide, JSONObject jsonObject) throws Exception {  
 orderWide.setTm\_name(jsonObject.getString("TM\_NAME"));  
 }  
  
 @Override  
 public String getKey(OrderWide orderWide) {  
 return String.*valueOf*(orderWide.getTm\_id());  
 }  
 }, 60, TimeUnit.*SECONDS*);

* 初始化品牌维度数据到Hbase（通过Maxwell的Bootstrap）

bin/maxwell-bootstrap --user maxwell --password 123456 --host hadoop202 --database gmall2021 --table base\_trademark --client\_id maxwell\_1

* 测试整体维度关联

### 结果写入kafka sink

**//*TODO 11.将订单和订单明细Join之后以及维度关联的宽表写到Kafka的dwm层***

**orderWideWithTmDstream.map(orderWide -> JSON.*toJSONString*(orderWide))  
 .addSink(MyKafkaUtil.*getKafkaSink*(orderWideSinkTopic));**

# DWM层-支付宽表（练习）

## 需求分析与思路

支付宽表的目的，最主要的原因是支付表没有到订单明细，支付金额没有细分到商品上，没有办法统计商品级的支付状况。

所以本次宽表的核心就是要把支付表的信息与订单明细关联上。

**解决方案有两个**

* 一个是把订单明细表（或者宽表）输出到Hbase上，在支付宽表计算时查询hbase，这相当于把订单明细作为一种维度进行管理。
* 一个是用流的方式接收订单明细，然后用双流join方式进行合并。因为订单与支付产生有一定的时差。所以必须用intervalJoin来管理流的状态时间，保证当支付到达时订单明细还保存在状态中。

## 功能实现参考

### 创建支付实体类PaymentInfo

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

**import lombok.Data;  
import java.math.BigDecimal;  
*/\*\*  
 \* Author: Felix  
 \* Desc: 支付信息实体类  
 \*/*@Data  
public class PaymentInfo {  
 Long id;  
 Long order\_id;  
 Long user\_id;  
 BigDecimal total\_amount;  
 String subject;  
 String payment\_type;  
 String create\_time;  
 String callback\_time;  
}**

### 创建支付宽表实体类PaymentWide

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

**import lombok.AllArgsConstructor;  
import lombok.Data;  
import lombok.NoArgsConstructor;  
import org.apache.commons.beanutils.BeanUtils;  
import java.lang.reflect.InvocationTargetException;  
import java.math.BigDecimal;  
*/\*\*  
 \* Author: Felix  
 \* Desc: 支付宽表实体类  
 \*/*@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class PaymentWide {  
  
 Long payment\_id;  
 String subject;  
 String payment\_type;  
 String payment\_create\_time;  
 String callback\_time;  
 Long detail\_id;  
 Long order\_id ;  
 Long sku\_id;  
 BigDecimal order\_price ;  
 Long sku\_num ;  
 String sku\_name;  
 Long province\_id;  
 String order\_status;  
 Long user\_id;  
 BigDecimal total\_amount;  
 BigDecimal activity\_reduce\_amount;  
 BigDecimal coupon\_reduce\_amount;  
 BigDecimal original\_total\_amount;  
 BigDecimal feight\_fee;  
 BigDecimal split\_feight\_fee;  
 BigDecimal split\_activity\_amount;  
 BigDecimal split\_coupon\_amount;  
 BigDecimal split\_total\_amount;  
 String order\_create\_time;  
  
 String province\_name;//查询维表得到  
 String province\_area\_code;  
 String province\_iso\_code;  
 String province\_3166\_2\_code;  
 Integer user\_age ;  
 String user\_gender;  
  
 Long spu\_id; //作为维度数据 要关联进来  
 Long tm\_id;  
 Long category3\_id;  
 String spu\_name;  
 String tm\_name;  
 String category3\_name;  
  
 public PaymentWide(PaymentInfo paymentInfo, OrderWide orderWide){  
 mergeOrderWide(orderWide);  
 mergePaymentInfo(paymentInfo);  
  
  
 }  
  
 public void mergePaymentInfo(PaymentInfo paymentInfo ) {  
 if (paymentInfo != null) {  
 try {  
 BeanUtils.*copyProperties*(this,paymentInfo);  
 payment\_create\_time=paymentInfo.create\_time;  
 payment\_id = paymentInfo.id;  
 } catch (IllegalAccessException e) {  
 e.printStackTrace();  
 } catch (InvocationTargetException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 public void mergeOrderWide(OrderWide orderWide ) {  
 if (orderWide != null) {  
 try {  
 BeanUtils.*copyProperties*(this,orderWide);  
 order\_create\_time=orderWide.create\_time;  
 } catch (IllegalAccessException e) {  
 e.printStackTrace();  
 } catch (InvocationTargetException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}**

### 支付宽表处理主程序

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

**import com.alibaba.fastjson.JSON;  
import com.atguigu.gmall.realtime.bean.OrderWide;  
import com.atguigu.gmall.realtime.bean.PaymentInfo;  
import com.atguigu.gmall.realtime.bean.PaymentWide;  
import com.atguigu.gmall.realtime.utils.DateTimeUtil;  
import com.atguigu.gmall.realtime.utils.MyKafkaUtil;  
import org.apache.flink.api.common.eventtime.WatermarkStrategy;  
import org.apache.flink.streaming.api.datastream.DataStream;  
import org.apache.flink.streaming.api.datastream.KeyedStream;  
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;  
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
import org.apache.flink.streaming.api.functions.co.ProcessJoinFunction;  
import org.apache.flink.streaming.api.windowing.time.Time;  
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;  
import org.apache.flink.util.Collector;  
  
import java.time.Duration;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc:支付宽表处理主程序  
 \*/*public class PaymentWideApp {  
 public static void main(String[] args) throws Exception {  
 //*TODO 0.基本环境准备* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 env.setParallelism(4);  
 /\*  
 //设置CK相关配置  
 env.enableCheckpointing(5000, CheckpointingMode.EXACTLY\_ONCE);  
 env.getCheckpointConfig().setCheckpointTimeout(60000);  
 StateBackend fsStateBackend = new FsStateBackend("hdfs://hadoop:8020/gmall/flink/checkpoint/OrderWideApp");  
 env.setStateBackend(fsStateBackend);  
 System.setProperty("HADOOP\_USER\_NAME", "atguigu");  
 \*/  
  
 //*TODO 1.接收数据流* String groupId = "payment\_wide\_group";  
 String paymentInfoSourceTopic = "dwd\_payment\_info";  
 String orderWideSourceTopic = "dwm\_order\_wide";  
 String paymentWideSinkTopic = "dwm\_payment\_wide";  
  
 //封装Kafka消费者 读取支付流数据  
 FlinkKafkaConsumer<String> paymentInfoSource = MyKafkaUtil.*getKafkaSource*(paymentInfoSourceTopic, groupId);  
 DataStream<String> paymentInfojsonDstream = env.addSource(paymentInfoSource);  
 //对读取的支付数据进行转换  
 DataStream<PaymentInfo> paymentInfoDStream =  
 paymentInfojsonDstream.map(jsonString -> JSON.*parseObject*(jsonString, PaymentInfo.class));  
  
 //封装Kafka消费者 读取订单宽表流数据  
 FlinkKafkaConsumer<String> orderWideSource = MyKafkaUtil.*getKafkaSource*(orderWideSourceTopic, groupId);  
 DataStream<String> orderWidejsonDstream = env.addSource(orderWideSource);  
 //对读取的订单宽表数据进行转换  
 DataStream<OrderWide> orderWideDstream =  
 orderWidejsonDstream.map(jsonString -> JSON.*parseObject*(jsonString, OrderWide.class));  
  
 //设置水位线  
 SingleOutputStreamOperator<PaymentInfo> paymentInfoEventTimeDstream =  
 paymentInfoDStream.assignTimestampsAndWatermarks(  
 WatermarkStrategy.<PaymentInfo>*forBoundedOutOfOrderness*(Duration.*ofSeconds*(3))  
 .withTimestampAssigner(  
 (paymentInfo, ts) -> DateTimeUtil.*toTs*(paymentInfo.getCallback\_time())  
 ));  
  
 SingleOutputStreamOperator<OrderWide> orderInfoWithEventTimeDstream =  
 orderWideDstream.assignTimestampsAndWatermarks(WatermarkStrategy.  
 <OrderWide>*forBoundedOutOfOrderness*(Duration.*ofSeconds*(3))  
 .withTimestampAssigner(  
 (orderWide, ts) -> DateTimeUtil.*toTs*(orderWide.getCreate\_time())  
 )  
 );  
  
 //设置分区键  
 KeyedStream<PaymentInfo, Long> paymentInfoKeyedStream =  
 paymentInfoEventTimeDstream.keyBy(PaymentInfo::getOrder\_id);  
 KeyedStream<OrderWide, Long> orderWideKeyedStream =  
 orderInfoWithEventTimeDstream.keyBy(OrderWide::getOrder\_id);  
  
 //关联数据  
 SingleOutputStreamOperator<PaymentWide> paymentWideDstream =  
 paymentInfoKeyedStream.intervalJoin(orderWideKeyedStream).  
 between(Time.*seconds*(-1800), Time.*seconds*(0)).  
 process(new ProcessJoinFunction<PaymentInfo, OrderWide, PaymentWide>() {  
 @Override  
 public void processElement(PaymentInfo paymentInfo,  
 OrderWide orderWide,  
 Context ctx, Collector<PaymentWide> out) throws Exception {  
 out.collect(new PaymentWide(paymentInfo, orderWide));  
 }  
 }).uid("payment\_wide\_join");  
  
 SingleOutputStreamOperator<String> paymentWideStringDstream = paymentWideDstream.map(paymentWide -> JSON.*toJSONString*(paymentWide));  
 paymentWideStringDstream.print("pay:");  
 paymentWideStringDstream.addSink(  
 MyKafkaUtil.*getKafkaSink*(paymentWideSinkTopic));  
  
 env.execute();  
  
 }  
}**

### 封装日期转换工具类

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

**import java.time.LocalDateTime;  
import java.time.ZoneId;  
import java.time.ZoneOffset;  
import java.time.format.DateTimeFormatter;  
import java.util.Date;  
  
*/\*\*  
 \* Author: Felix  
 \* Desc:日期转换工具类***

***JDK8的DateTimeFormatter替换SimpleDateFormat，因为SimpleDateFormat存在线程安全问题  
 \*/*public class DateTimeUtil {  
 public final static** **DateTimeFormatter *formator* = DateTimeFormatter.*ofPattern*("yyyy-MM-dd HH:mm:ss");  
  
 public static String toYMDhms(Date date) {  
 LocalDateTime localDateTime = LocalDateTime.*ofInstant*(date.toInstant(), ZoneId.*systemDefault*());  
 return *formator*.format(localDateTime);  
 }  
  
 public static Long toTs(String YmDHms) {  
 // System.out.println ("YmDHms:"+YmDHms);  
 LocalDateTime localDateTime = LocalDateTime.*parse*(YmDHms, *formator*);  
 long ts = localDateTime.toInstant(ZoneOffset.*of*("+8")).toEpochMilli();  
 return ts;  
 }  
}**

# 总结

DWM层部分的代码主要的责任，是通过计算把一种明细转变为另一种明细以应对后续的统计。学完本阶段内容要求掌握

* 学会利用状态（state）进行去重操作。（需求：UV计算）
* 学会利用CEP可以针对一组数据进行筛选判断。需求：跳出行为计算
* 学会使用intervalJoin处理流join
* 学会处理维度关联，并通过缓存和异步查询对其进行性能优化。