

尚硅谷大数据之电商实时数仓 之 DWD 层实现

(作者：尚硅谷研究院)

版本：V3.0

DWD 层设计要点：

- DWD 层的设计依据是维度建模理论，该层存储维度模型的事实表
- DWD 层表名的命名规范为 `dwd_数据域_表名`

第 1 章 流量域未经加工的事务事实表

1.1 主要任务

1.1.1 数据清洗（ETL）

数据传输过程中可能会出现部分数据丢失的情况，导致 JSON 数据结构不再完整，因此需要对脏数据进行过滤。

1.1.2 新老访客状态标记修复

日志数据 `common` 字段下的 `is_new` 字段是用来标记新老访客状态的，1 表示新访客，0 表示老访客。前端埋点采集到的数据可靠性无法保证，可能会出现老访客被标记为新访客的问题，因此需要对该标记进行修复。

1.1.3 分流

本节将通过分流对日志数据进行拆分，生成五张事务事实表写入 Kafka

- 流量域页面浏览事务事实表
- 流量域启动事务事实表
- 流量域动作事务事实表
- 流量域曝光事务事实表

- 流量域错误事务事实表

1.2 思路

1.2.1 数据清洗 (ETL)

对流中数据进行解析，将字符串转换为 JSONObject，如果解析报错则必然为脏数据。

定义侧输出流，将脏数据发送到侧输出流，写入 Kafka 脏数据主题

1.2.2 新老访客状态标记修复

(1) 前端埋点新老访客状态标记设置规则

以神策提供的第三方埋点服务中新老访客状态标记设置规则为例

- Web 端：用户第一次访问埋入神策 SDK 页面的当天（即第一天），JS SDK 会在网页的 cookie 中设置一个首日访问的标记，并设置第一天 24 点之前，该标记为 true，即第一天触发的网页端所有事件中，is_new = 1。第一天之后，该标记则为 false，即第一天之后触发的网页端所有事件中，is_new = 0；
- 小程序端：用户第一天访问埋入神策 SDK 的页面时，小程序 SDK 会在 storage 缓存中创建一个首日为 true 的标记，并且设置第一天 24 点之前，该标记均为 true。即第一天触发的小程序端所有事件中，is_new = 1。第一天之后，该标记则为 false，即第一天之后触发的小程序端所有事件中，is_new = 0；
- APP 端：用户安装 App 后，第一次打开埋入神策 SDK 的 App 的当天，Android/iOS SDK 会在手机本地缓存内，创建一个首日为 true 的标记，并且设置第一天 24 点之前，该标记均为 true。即第一天触发的 APP 端所有事件中，is_new = 1。第一天之后，该标记则为 false，即第一天之后触发的 APP 端所有事件中，is_new = 0。

本项目模拟生成的是 APP 端日志数据。对于此类日志, 如果首日之后用户~~清除了手机本地缓存中的标记~~, 再次启动 APP 会重新设置一个首日为 true 的标记, 导致本应为 0 的 is_new 字段被置为 1, 可能会给相关指标带来误差。因此, 有必要对新老访客状态标记进行修复。

(2) 新老访客状态标记修复思路

运用 Flink 状态编程, 为每个 mid 维护一个~~键控状态~~, 记录~~首次访问日期~~。

- 如果 is_new 的值为 1
 - 如果键控状态为 null, 认为本次是该访客首次访问 APP, 将日志中 ts 对应的日期更新到状态中, 不对 is_new 字段做修改;
 - 如果键控状态不为 null, 且首次访问日期不是当日, 说明访问的是老访客, 将 is_new 字段置为 0;
 - 如果键控状态不为 null, 且首次访问日期是当日, 说明访问的是新访客, 不做操作;
- 如果 is_new 的值为 0
 - 如果键控状态为 null, 说明访问 APP 的是老访客但本次是该访客的页面日志首次进入程序。当前端新老访客状态标记丢失时, 日志进入程序被判定为新访客, Flink 程序就可以纠正被误判的访客状态标记, 只要将状态中的日期设置为今天之前即可。本程序选择将状态更新为昨日;
 - 如果键控状态不为 null, 说明程序已经维护了首次访问日期, 不做操作。

1.2.3 分流：利用侧输出流实现数据拆分

(1) 埋点日志结构分析

前端埋点获取的 JSON 字符串（日志）可能存在 common、start、page、displays、actions、err、ts 七种字段。其中

- common 对应的是公共信息，是所有日志都有的字段
- err 对应的是错误信息，所有日志都可能有的字段
- start 对应的是启动信息，启动日志才有的字段
- page 对应的是页面信息，页面日志才有的字段
- displays 对应的是曝光信息，曝光日志才有的字段，曝光日志可以归为页面日志，因此必然有 page 字段
- actions 对应的是动作信息，动作日志才有的字段，同样属于页面日志，必然有 page 字段。动作信息和曝光信息可以同时存在。
- ts 对应的是时间戳，单位：毫秒，所有日志都有的字段

综上，我们可以将前端埋点获取的日志分为两大类：启动日志和页面日志。二者都有 common 字段和 ts 字段，都可能有 err 字段。页面日志一定有 page 字段，一定没有 start 字段，可能有 displays 和 actions 字段；启动日志一定有 start 字段，一定没有 page、displays 和 actions 字段。

(2) 分流日志分类

本节将按照内容，将日志分为以下五类

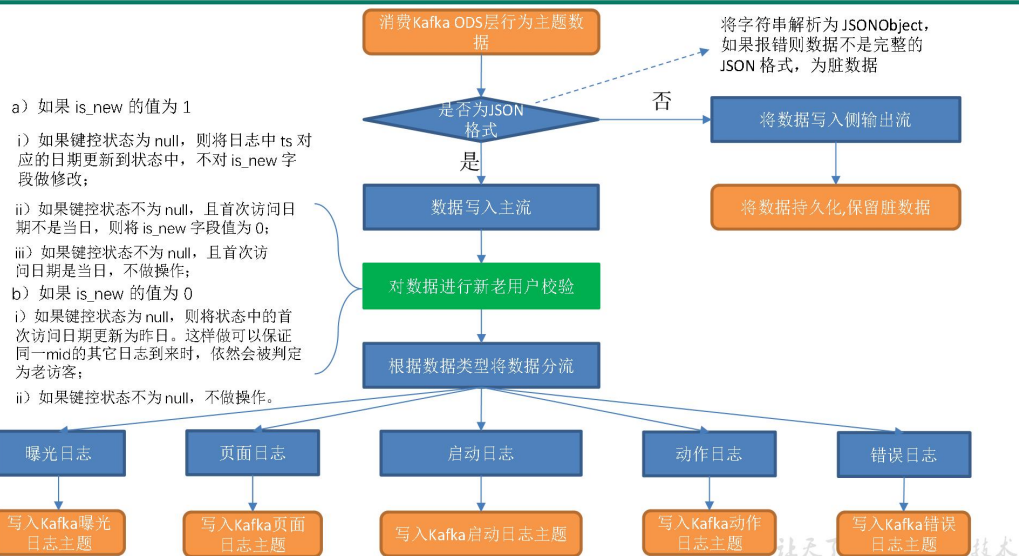
- 启动日志
- 页面日志
- 曝光日志
- 动作日志
- 错误日志

(3) 分流思路

- 所有日志数据都可能拥有 err 字段,所有首先获取 err 字段,如果返回值不为 null 则将整条日志数据发送到错误侧输出流。然后删掉 JSONObject 中的 err 字段及对应值;
- 判断是否有 start 字段,如果有则说明数据为启动日志,将其发送到启动侧输出流;如果没有则说明为页面日志,进行下一步;
- 页面日志必然有 page 字段、common 字段和 ts 字段,获取它们的值,ts 封装为包装类 Long,其余两个字段的值封装为 JSONObject;
- 判断是否有 displays 字段,如果有,将其值封装为 JSONArray,遍历该数组,依次获取每个元素(记为 display),封装为 JSONObject。创建一个空的 JSONObject,将 display、common、page 和 ts 添加到该对象中,获得处理好的曝光数据,发送到曝光侧输出流。动作日志的处理与曝光日志相同(注意:一条页面日志可能既有曝光数据又有动作数据,二者没有任何关系,因此曝光数据不为 null 时仍要对动作数据进行处理);
- 动作日志和曝光日志处理结束后删除 displays 和 actions 字段,此时主流的 JSONObject 中只有 common 字段、page 字段和 ts 字段,即为最终的页面日志。处理结束后,页面日志数据位于主流,其余四种日志分别位于对应的侧输出流,将五条流的数据写入 Kafka 对应主题即可。

1.3 图解

行为数据DWD层代码流程图



1.4 代码

1.4.1 主程序

```
package com.atguigu.gmall.realtime.app.dwd.log;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONArray;
import com.alibaba.fastjson.JSONObject;
import com.atguigu.gmall.realtime.util.DateFormatUtil;
import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import org.apache.flink.api.common.restartstrategy.RestartStrategies;
import org.apache.flink.api.common.state.ValueState;
import org.apache.flink.api.common.state.ValueStateDescriptor;
import org.apache.flink.api.common.time.Time;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.runtime.state.hashmap.HashMapStateBackend;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.datastream.KeyedStream;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.KeyedProcessFunction;
import org.apache.flink.streaming.api.functions.ProcessFunction;
import org.apache.flink.util.Collector;
import org.apache.flink.util.OutputTag;

import java.util.concurrent.TimeUnit;
```

/**

```

* Author: Felix
* Date: 2022/5/10
* Desc: 日志数据分流 -- 获得流量域相关事实表
*/
public class DwdTrafficBaseLogSplit {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);

        // TODO 2. 启用状态后端
        env.enableCheckpointing(3000L, CheckpointingMode.EXACTLY_ONCE);
        env.getCheckpointConfig().setCheckpointTimeout(60 * 1000L);
        env.getCheckpointConfig().setMinPauseBetweenCheckpoints(3000L);
        env.getCheckpointConfig().enableExternalizedCheckpoints(
        CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION
        );
        env.setRestartStrategy(RestartStrategies
        .failureRateRestart(3,
        Time.of(3L, TimeUnit.DAYS),
        Time.of(1L, TimeUnit.MINUTES)));
        env.setStateBackend(new HashMapStateBackend());

        env.getCheckpointConfig().setCheckpointStorage("hdfs://hadoop102:8020/gmall/
        ck");
        System.setProperty("HADOOP_USER_NAME", "atguigu");

        // TODO 3. 从 Kafka 读取主流数据
        String topic = "topic_log";
        String groupId = "base_log_group";
        DataStreamSource<String> kafkaStrDS =
        env.addSource(MyKafkaUtil.getKafkaConsumer(topic, groupId));

        // TODO 4. 数据清洗, 转换结构
        // 4.1 定义错误侧输出流
        OutputTag<String> dirtyStreamTag = new OutputTag<String>("dirtyStream")
        {};

        // 4.2 转换 清洗
        SingleOutputStreamOperator<JSONObject> cleanedStream =
        kafkaStrDS.process(
        new ProcessFunction<String, JSONObject>() {
            @Override
            public void processElement(String jsonStr, Context ctx, Collector<
            JSONObject> out) throws Exception {
                try {
                    JSONObject jsonObj = JSON.parseObject(jsonStr);
                    out.collect(jsonObj);
                } catch (Exception e) {
                    ctx.output(dirtyStreamTag, jsonStr);
                }
            }
        }
        );

        // 4.3 将脏数据写出到 Kafka 指定主题
        DataStream<String> dirtyStream =
        cleanedStream.getSideOutput(dirtyStreamTag);
    }
}

```



```

String dirtyTopic = "dirty_data";
dirtyStream.addSink(MyKafkaUtil.getKafkaProducer(dirtyTopic));

// TODO 5. 新老访客状态标记修复
// 5.1 按照 mid 对数据进行分组
KeyedStream<JSONObject, String> keyedStream = mappedStream.keyBy(r ->
r.getJSONObject("common").getString("mid"));

// 5.2 新老访客状态标记修复
SingleOutputStreamOperator<JSONObject> fixedStream =
keyedStream.process(
    new KeyedProcessFunction<String, JSONObject, JSONObject>() {

        ValueState<String> lastVisitDateState;

        @Override
        public void open(Configuration param) throws Exception {
            lastVisitDateState = getRuntimeContext().getState(new
ValueStateDescriptor<String>(
                "lastVisitDateState", String.class
            ));
        }

        @Override
        public void processElement(JSONObject jsonObj, Context ctx,
Collector<JSONObject> out) throws Exception {
            String isNew = jsonObj.getJSONObject("common").getString("is_new");
            String lastVisitDate = lastVisitDateState.value();
            Long ts = jsonObj.getLong("ts");
            String curVisitDate = DateFormatUtil.toDate(ts);

            if ("1".equals(isNew)) {
                if (lastVisitDate == null) {
                    lastVisitDateState.update(curVisitDate);
                } else {
                    if (!lastVisitDate.equals(curVisitDate)) {
                        isNew = "0";
                        jsonObj.getJSONObject("common").put("is_new",
isNew);
                    }
                }
            } else {
                if (lastVisitDate == null) {
                    // 将首次访问日期置为昨日
                    String yesterday = DateFormatUtil.toDate(ts - 1000 * 60
* 60 * 24);
                    lastVisitDateState.update(yesterday);
                }
            }

            out.collect(jsonObj);
        }
    }
);

// TODO 6. 分流
// 6.1 定义启动、曝光、动作、错误侧输出流
OutputTag<String> startTag = new OutputTag<String>("startTag") {};
OutputTag<String> displayTag = new OutputTag<String>("displayTag") {};
OutputTag<String> actionTag = new OutputTag<String>("actionTag") {};

```

```

OutputTag<String> errorTag = new OutputTag<String>("errorTag") {};

// 6.2 分流
SingleOutputStreamOperator<String> separatedStream =
fixedStream.process(
    new ProcessFunction<JSONObject, String>() {
        @Override
        public void processElement(JSONObject jsonObj, Context context,
Collector<String> out) throws Exception {

            // 6.2.1 收集错误数据
            JSONObject error = jsonObj.getJSONObject("err");
            if (error != null) {
                context.output(errorTag, jsonObj.toJSONString());
                // 剔除 "err" 字段
                jsonObj.remove("err");
            }

            // 6.2.2 收集启动数据
            JSONObject start = jsonObj.getJSONObject("start");
            if (start != null) {
                context.output(startTag, jsonObj.toJSONString());
            } else {
                // 获取 "page" 字段
                JSONObject page = jsonObj.getJSONObject("page");
                // 获取 "common" 字段
                JSONObject common = jsonObj.getJSONObject("common");
                // 获取 "ts"
                Long ts = jsonObj.getLong("ts");

                // 6.2.3 收集曝光数据
                JSONArray displays = jsonObj.getJSONArray("displays");
                if (displays != null) {
                    for (int i = 0; i < displays.size(); i++) {
                        JSONObject display = displays.getJSONObject(i);
                        JSONObject displayObj = new JSONObject();
                        displayObj.put("display", display);
                        displayObj.put("common", common);
                        displayObj.put("page", page);
                        displayObj.put("ts", ts);
                        context.output(displayTag,
displayObj.toJSONString());
                    }
                }

                // 6.2.4 收集动作数据
                JSONArray actions = jsonObj.getJSONArray("actions");
                if (actions != null) {
                    for (int i = 0; i < actions.size(); i++) {
                        JSONObject action = actions.getJSONObject(i);
                        JSONObject actionObj = new JSONObject();
                        actionObj.put("action", action);
                        actionObj.put("common", common);
                        actionObj.put("page", page);
                        context.output(actionTag,
actionObj.toJSONString());
                    }
                }
            }
        }
    }
);

```

```

        // 6.2.5 收集页面数据
        jsonObj.remove("displays");
        jsonObj.remove("actions");
        out.collect(jsonObj.toJSONString());
    }

    }

);

// 打印主流和各侧输出流查看分流效果
separatedStream.print("page>>>");
separatedStream.getSideOutput(startTag).print("start!!!");
separatedStream.getSideOutput(displayTag).print("display@@@");
separatedStream.getSideOutput(actionTag).print("action###");
separatedStream.getSideOutput(errorTag).print("error$$$");

// TODO 7. 将数据输出到 Kafka 的不同主题
// 7.1 提取各侧输出流
DataStream<String> startDS = separatedStream.getSideOutput(startTag);
DataStream<String> displayDS = separatedStream.getSideOutput(displayTag);
DataStream<String> actionDS = separatedStream.getSideOutput(actionTag);
DataStream<String> errorDS = separatedStream.getSideOutput(errorTag);

// 7.2 定义不同日志输出到 Kafka 的主题名称
String page_topic = "dwd_traffic_page_log";
String start_topic = "dwd_traffic_start_log";
String display_topic = "dwd_traffic_display_log";
String action_topic = "dwd_traffic_action_log";
String error_topic = "dwd_traffic_error_log";

separatedStream.addSink(MyKafkaUtil.getKafkaProducer(page_topic));
startDS.addSink(MyKafkaUtil.getKafkaProducer(start_topic));
displayDS.addSink(MyKafkaUtil.getKafkaProducer(display_topic));
actionDS.addSink(MyKafkaUtil.getKafkaProducer(action_topic));
errorDS.addSink(MyKafkaUtil.getKafkaProducer(error_topic));

env.execute();
}
}

```

1.4.2 在 MyKafkaUtil 工具类中补充 getKafkaProducer() 方法

```

//获取 kafka 生产者对象方法
public static FlinkKafkaProducer<String> getKafkaProducer(String topic) {
    Properties prop = new Properties();
    prop.setProperty("bootstrap.servers", BOOTSTRAP_SERVERS);
    prop.setProperty(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 60 * 15 * 1000
+ "");
    FlinkKafkaProducer<String> producer = new FlinkKafkaProducer<String>(
        DEFAULT_TOPIC,
        new KafkaSerializationSchema<String>() {

```

```

        @Override
        public ProducerRecord<byte[], byte[]> serialize(String jsonStr,
@Nullable Long timestamp) {
            return new ProducerRecord<byte[], byte[]>(topic,
jsonStr.getBytes());
        }
    },
    prop,
    FlinkKafkaProducer.Semantic.EXACTLY_ONCE);
    return producer;
}

```

1.4.3 创建 DateFormatUtil 工具类用于日期格式化

```

package com.atguigu.gmall.realtime.util;

import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.util.Date;

/**
 * Author: Felix
 * Date: 2022/5/10
 * Desc: 日期转换工具类
 */
public class DateFormatUtil {
    private static final DateTimeFormatter dtf =
DateTimeFormatter.ofPattern("yyyy-MM-dd");
    private static final DateTimeFormatter dtfFull =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

    public static Long toTs(String dtStr, boolean isFull) {

        LocalDateTime localDateTime = null;
        if (!isFull) {
            dtStr = dtStr + " 00:00:00";
        }
        localDateTime = LocalDateTime.parse(dtStr, dtfFull);

        return localDateTime.toInstant(ZoneOffset.of("+8")).toEpochMilli();
    }

    public static Long toTs(String dtStr) {
        return toTs(dtStr, false);
    }

    public static String toDate(Long ts) {
        Date dt = new Date(ts);
        LocalDateTime localDateTime = LocalDateTime.ofInstant(dt.toInstant(),
ZoneId.systemDefault());
        return dtf.format(localDateTime);
    }

    public static String toYmdHms(Long ts) {
        Date dt = new Date(ts);
        LocalDateTime localDateTime = LocalDateTime.ofInstant(dt.toInstant(),
ZoneId.systemDefault());
        return dtfFull.format(localDateTime);
    }
}

```

```
}

public static void main(String[] args) {
    System.out.println(toYmdHms(System.currentTimeMillis()));
}
}
```

1.5 测试

- 启动 zk、kafka、flume(f1.sh start)
- 运行 DwdTrafficBaseLogSplit
- 运行生成日志数据的 jar(lg.sh)
- 创建消费者对象查看 kafka 各个 dwd 日志主题的输出

第 2 章 流量域独立访客事务事实表

2.1 主要任务

过滤出页面数据中的独立访客访问记录。

2.2 思路分析

2.2.1 过滤 last_page_id 不为 null 的数据

独立访客数据对应的页面必然是会话起始页面，last_page_id 必为 null。过滤 last_page_id != null 的数据，减小数据量，提升计算效率。

2.2.2 筛选独立访客记录

运用 Flink 状态编程，为每个 mid 维护一个键控状态，记录末次登录日期。

如果末次登录日期为 null 或者不是今日，则本次访问是该 mid 当日首次访问，保留

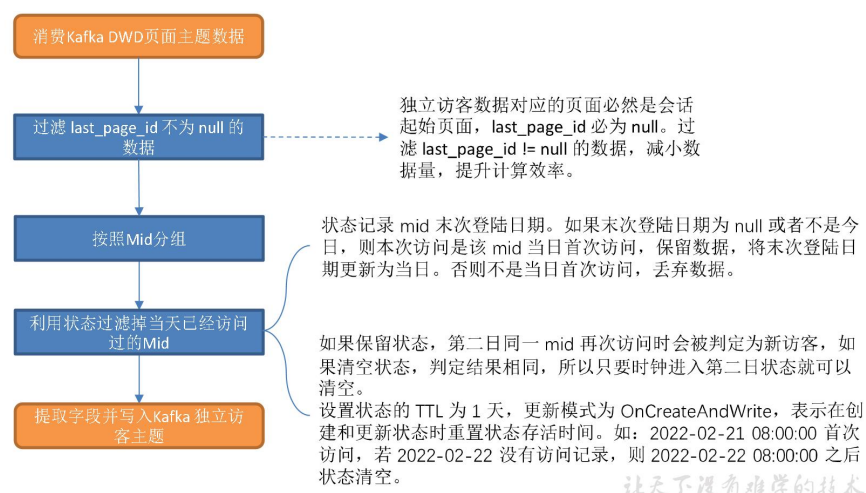
数据，将末次登录日期更新为当日。否则不是当日首次访问，丢弃数据。

2.2.3 状态存活时间设置

如果保留状态，第二日同一 mid 再次访问时会被判定为新访客，如果清空状态，判定结果相同，所以只要时钟进入第二日状态就可以清空。

设置状态的 TTL 为 1 天，更新模式为 OnCreateAndWrite，表示在创建和更新状态时重置状态存活时间。如：2022-02-21 08:00:00 首次访问，若 2022-02-22 没有访问记录，则 2022-02-22 08:00:00 之后状态清空。

2.3 图解



2.4 代码

```
package com.atguigu.gmall.realtime.app.dwd.log;
import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONAware;
import com.alibaba.fastjson.JSONObject;
import com.atguigu.gmall.realtime.util.DateFormatUtil;
import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import org.apache.flink.api.common.functions.RichFilterFunction;
import org.apache.flink.api.common.restartstrategy.RestartStrategies;
import org.apache.flink.api.common.state.StateTtlConfig;
```

```

import org.apache.flink.api.common.state.ValueState;
import org.apache.flink.api.common.state.ValueStateDescriptor;
import org.apache.flink.api.common.time.Time;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.runtime.state.hashmap.HashMapStateBackend;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.datastream.KeyedStream;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.CheckpointConfig;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer;

/**
 * Author: Felix
 * Date: 2022/5/10
 * Desc: 流量域独立访客事务事实表
 */
public class DwdTrafficUniqueVisitorDetail {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);

        // TODO 2. 检查点设置
        env.enableCheckpointing(3000L, CheckpointingMode.EXACTLY_ONCE);
        env.getCheckpointConfig().setCheckpointTimeout(30 * 1000L);
        env.getCheckpointConfig().setMinPauseBetweenCheckpoints(3000L);
        env.getCheckpointConfig().enableExternalizedCheckpoints(
            CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION
        );
        env.setRestartStrategy(RestartStrategies.failureRateRestart(
            3, Time.days(1), Time.minutes(1)
        ));
        env.setStateBackend(new HashMapStateBackend());
        env.getCheckpointConfig().setCheckpointStorage(
            "hdfs://hadoop102:8020/ck"
        );
        System.setProperty("HADOOP_USER_NAME", "atguigu");

        // TODO 3. 从 kafka dwd_traffic_page_log 主题读取日志数据, 封装为流
        String topic = "dwd_traffic_page_log";
        String groupId = "dwd_traffic_user_jump_detail";
        FlinkKafkaConsumer<String> kafkaConsumer =
            MyKafkaUtil.getKafkaConsumer(topic, groupId);
        DataStreamSource<String> pageStrDS = env.addSource(kafkaConsumer);

        // TODO 4. 转换结构
        SingleOutputStreamOperator<JSONObject> jsonObjDS =
            pageStrDS.map(JSON::parseObject);

        // TODO 5. 按照 mid 分组
        KeyedStream<JSONObject, String> keyedStream = jsonObjDS
            .keyBy(jsonObj -> jsonObj.getJSONObject("common").getString("mid"));

        // TODO 7. 通过 Flink 状态编程过滤独立访客记录
        SingleOutputStreamOperator<JSONObject> filteredStream =

```

```

keyedStream.filter(
    new RichFilterFunction<JSONObject>() {

        private ValueState<String> lastVisitDateState;

        @Override
        public void open(Configuration parameters) throws Exception {
            ValueStateDescriptor<String> valueStateDescriptor =
                new ValueStateDescriptor<>("lastVisitDateState",
String.class);
            valueStateDescriptor.enableTimeToLive(
                StateTtlConfig
                    .newBuilder(Time.days(1L))
                    // 设置在创建和更新状态时更新存活时间
                    .setUpdateType(StateTtlConfig.UpdateType.OnCreateAndW
rite)
                    .build()
            );
            lastVisitDateState =
getRuntimeContext().getState(valueStateDescriptor);
        }

        @Override
        public boolean filter(JSONObject jsonObj) throws Exception {
            String lastPageId =
jsonObj.getJSONObject("page").getString("last_page_id");
            //过滤 last_page_id 不为 null 的数据
            if (lastPageId != null) {
                return false;
            }
            String curVisitDate =
DateFormatUtil.toDate(jsonObj.getLong("ts"));
            String lastVisitDate = lastVisitDateState.value();
            if (lastVisitDate == null
|| !lastVisitDate.equals(curVisitDate)) {
                lastVisitDateState.update(curVisitDate);
                return true;
            }
            return false;
        }
    }
);

filteredStream.print(">>>>");
// TODO 8. 将独立访客数据写入 Kafka dwd_traffic_unique_visitor_detail 主题
String targetTopic = "dwd_traffic_unique_visitor_detail";
filteredStream
    .map(JSONAware::toJSONString)
    .addSink(MyKafkaUtil.getKafkaProducer(targetTopic));

// TODO 9. 启动任务
env.execute();
}
}

```


2.5 测试

- 启动 zk、kafka、flume(f1.sh start)
- 运行 DwdTrafficBaseLogSplit、DwdTrafficUniqueVisitorDetail
- 运行生成日志数据的 jar(lg.sh)
- 创建消费者对象查看 kafka 主题 `dwd_traffic_unique_visitor_detail` 的输出

第 3 章 流量域用户跳出事务事实表

3.1 主要任务

过滤用户跳出明细数据。

3.2 思路分析

3.2.1 筛选策略

跳出是指会话中只有一个页面的访问行为，如果能获取会话的所有页面，只要筛选页面数为 1 的会话即可获取跳出明细数据。

- (1) 离线数仓中我们可以获取一整天的数据，结合访问时间、`page_id` 和 `last_page_id` 字段对整体数据集做处理可以按照会话对页面日志进行划分，从而获得每个会话的页面数，只要筛选页面数为 1 的会话即可提取跳出明细数据；
- (2) 实时计算中无法考虑整体数据集，很难按照会话对页面访问记录进行划分。而本项目模拟生成的日志数据中没有 `session_id` (会话 id) 字段，也无法通过按照 `session_id` 分组的方式计算每个会话的页面数。

(3) 因此，我们需要换一种解决思路。如果能判定首页日志之后没有同一会话的页面访问记录同样可以筛选跳出数据。如果日志数据完全有序，会话页面不存在交叉情况，则跳出页面的判定可以分为三种情况

- ① 两条紧邻的首页日志进入算子，可以判定第一条首页日志所属会话为跳出会话；
- ② 第一条首页日志进入算子后，接收到的第二条日志为非首页日志，则第一条日志所属会话不是跳出会话；
- ③ 第一条首页日志进入算子后，没有收到第二条日志，此时无法得出结论，必须继续等待。但是无休止地等待显然是不现实的。因此，人为设定超时时间，超时时间内没有第二条数据就判定为跳出行为，这是一种近似处理，存在误差，但若能结合业务场景设置合理的超时时间，误差是可以接受的。本程序为了便于测试，设置超时时间为 10s，为了更快看到效果可以设置更小的超时时间，生产环境的设置结合业务需求确定。

由上述分析可知，情况 ① 的首页数据和情况 ③ 中的超时数据为跳出明细数据。

3.2.2 知识储备

(1) Flink CEP

跳出行为需要考虑会话中的两条页面日志数据(第一条为首页日志且超时时间内没有接收到第二条，或两条紧邻的首页日志到来可以判定第一条为跳出数据)，要筛选的是组合事件，用 filter 无法实现这样的功能，由此引出 Flink CEP。

Flink CEP (Complex Event Processing 复杂事件处理) 是在 Flink 上层实现的复杂事件处理库，可以在无界流中检测出特定的事件模型。用户定义复杂规则 (Pattern)，将其应用到流上，即可从流中提取满足 Pattern 的一个或多个简单事件构成的复杂事件。

(2) Flink CEP 定义的规则之间的连续策略

- 严格连续: 期望所有匹配的事件严格的一个接一个出现, 中间没有任何不匹配的事件。

对应方法为 `next()`;

- 松散连续: 忽略匹配的事件之间的不匹配的事件。对应方法为 `followedBy()`;
- 不确定的松散连续: 更进一步的松散连续, 允许忽略掉一些匹配事件的附加匹配。对应方法为 `followedByAny()`。

3.2.3 实现步骤

(1) 按照 mid 分组

不同访客的浏览记录互不干涉, 跳出行为的分析应在相同 mid 下进行, 首先按照 mid 分组。

(2) 定义 CEP 匹配规则

- 规则一

跳出行为对应的页面日志必然为某一会话的首页, 因此第一个规则判定 `last_page_id` 是否为 `null`, 是则返回 `true`, 否则返回 `false`;

- 规则二

规则二和规则一之间的策略采用严格连续, 要求二者之间不能有其它事件。判断 `last_page_id` 是否为 `null`, 在数据完整有序的前提下, 如果不是 `null` 说明本条日志的页面不是首页, 可以断定它与规则一匹配到的事件同属于一个会话, 返回 `false`; 如果是 `null` 则开启了一个新的会话, 此时可以判定上一条页面日志所属会话为跳出会话, 是我们需要的数据, 返回 `true`;

- 超时时间

超时时间内规则一被满足，未等到第二条数据则会被判定为超时数据。

(3) 把匹配规则 (Pattern) 应用到流上

根据 Pattern 定义的规则对流中数据进行筛选。

(4) 提取超时流

提取超时流，超时流中满足规则一的数据即为跳出明细数据，取出。

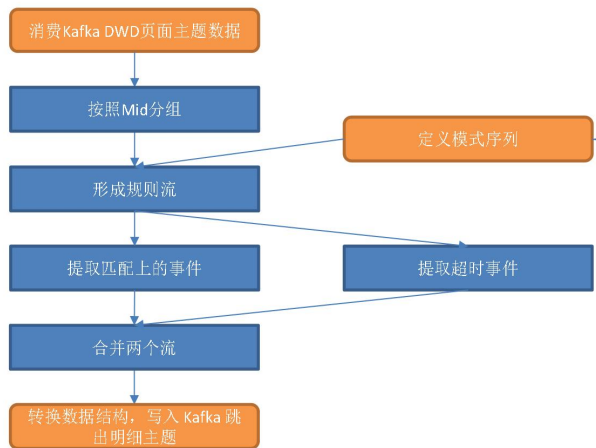
(5) 合并主流和超时流，写入 Kafka 调出明细主题

(6) 结果分析

理论上 Flink 可以通过设置水位线保证数据严格有序 (超时时间足够大)，在此前提下，同一 mid 的会话之间不会出现交叉。若假设日志数据没有丢失，按照上述匹配规则，我们可以获得两类明细数据

- 两个规则都被满足，满足规则一的数据为跳出明细数据。在会话之间不会交叉且日志数据没有丢失的前提下，此时获取的跳出明细数据没有误差；
- 第一条数据满足规则一，超时时间内没有接收到第二条数据，水位线达到超时时间，第一条数据被发送到超时侧输出流。即便在会话之间不交叉且日志数据不丢失的前提下，此时获取的跳出明细数据仍有误差，因为超时时间之后会话可能并未结束，如果此时访客在同一会话内跳转到了其它页面，就会导致会话页面数大于 1 的访问被判定为跳出行为，下游计算的跳出率偏大。误差大小和设置的超时时间呈负相关关系，超时时间越大，理论上误差越小。

3.3 图解



前提：理论上 Flink 通过设置水位线可以保证数据严格有序（水位线的乱序程度足够大）从而确保会话之间不会交叉，且数据不丢失

1、规则一：last_page_id == null 则返回 true 发生跳出行为的必然是会话起始页面，上页 id 必然为 null，舍弃了非会话起始页的日志数据。

2、规则二：last_page_id == null 则返回 true 规则二和规则一之间的连续策略为严格连续。如果两条数据匹配了规则一和规则二，则满足规则一的数据为跳出明细数据。

3、指定超时时间
超时时间内第二条数据没有到来时，第一条数据是目标数据，被判定为跳出明细数据。

让天下没有难学的技术

3.4 代码

3.4.1 添加 CEP 相关依赖

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-cep_${scala.version}</artifactId>
  <version>${flink.version}</version>
</dependency>
```

3.4.2 主程序

```
package com.atguigu.gmall.realtime.app.dwd.log;

import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.JSONAware;
import com.alibaba.fastjson.JSONObject;
import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import org.apache.flink.api.common.eventtime.SerializableTimestampAssigner;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.cep.CEP;
import org.apache.flink.cep.PatternFlatSelectFunction;
import org.apache.flink.cep.PatternFlatTimeoutFunction;
import org.apache.flink.cep.PatternStream;
import org.apache.flink.cep.pattern.Pattern;
import org.apache.flink.cep.pattern.conditions.SimpleCondition;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
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.connectors.kafka.FlinkKafkaConsumer;
import org.apache.flink.util.Collector;
import org.apache.flink.util.OutputTag;

import java.util.List;
import java.util.Map;

/**
 * Author: Felix
 * Date: 2022/5/10
 * Desc: 流量域用户跳出明细事实表
 */
public class DwdTrafficUserJumpDetail {

    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 从 kafka dwd_traffic_page_log 主题读取日志数据, 封装为流
        String topic = "dwd_traffic_page_log";
        String groupId = "dwd_traffic_user_jump_detail";
        FlinkKafkaConsumer<String> kafkaConsumer =
            MyKafkaUtil.getKafkaConsumer(topic, groupId);
        DataStreamSource<String> kafkaStrDS = env.addSource(kafkaConsumer);

        /** 测试数据
        DataStream<String> kafkaStrDS = 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} "
            );*/

        // TODO 4. 转换结构
        SingleOutputStreamOperator<JSONObject> mappedStream =
            kafkaStrDS.map(JSON::parseObject);

        // TODO 5. 设置水位线, 用于用户跳出统计
        SingleOutputStreamOperator<JSONObject> withWatermarkStream =
            mappedStream.assignTimestampsAndWatermarks(
                WatermarkStrategy
                    .<JSONObject>forMonotonousTimestamps()
                    .withTimestampAssigner(

```

```

        new SerializableTimestampAssigner<JSONObject>() {
            @Override
            public long extractTimestamp(JSONObject jsonObj, long
recordTimestamp) {
                return jsonObj.getLong("ts");
            }
        }
    );

    // TODO 6. 按照 mid 分组
    KeyedStream<JSONObject, String> keyedStream =
withWatermarkStream.keyBy(jsonObjb ->
jsonObjb.getJSONObject("common").getString("mid"));

    // TODO 7. 定义 CEP 匹配规则
    Pattern<JSONObject, JSONObject> pattern =
Pattern.<JSONObject>begin("first").where(
        new SimpleCondition<JSONObject>() {
            @Override
            public boolean filter(JSONObject jsonObj) throws Exception {
                String lastPageId =
jsonObj.getJSONObject("page").getString("last_page_id");
                return lastPageId == null;
            }
        }
    ).next("second").where(
        new SimpleCondition<JSONObject>() {
            @Override
            public boolean filter(JSONObject jsonObj) throws Exception {
                String lastPageId =
jsonObj.getJSONObject("page").getString("last_page_id");
                return lastPageId == null;
            }
        }
    );
    // 上文调用了同名 Time 类, 此处需要使用全类名
    ).within(org.apache.flink.streaming.api.windowing.time.Time.seconds(10
L));

    // TODO 8. 把 Pattern 应用到流上
    PatternStream<JSONObject> patternStream = CEP.pattern(keyedStream,
pattern);

    // TODO 9. 提取匹配上的事件以及超时事件
    OutputTag<JSONObject> timeoutTag = new
OutputTag<JSONObject>("timeoutTag") {
    };
    SingleOutputStreamOperator<JSONObject> flatSelectStream =
patternStream.flatSelect(
        timeoutTag,
        new PatternFlatTimeoutFunction<JSONObject, JSONObject>() {
            @Override
            public void timeout(Map<String, List<JSONObject>> pattern, long
timeoutTimestamp, Collector<JSONObject> out) throws Exception {
                JSONObject element = pattern.get("first").get(0);
                out.collect(element);
            }
        },
        new PatternFlatSelectFunction<JSONObject, JSONObject>() {

```

```

        @Override
        public void flatSelect(Map<String, List<JSONObject>> pattern,
Collector<JSONObject> out) throws Exception {
            JSONObject element = pattern.get("first").get(0);
            out.collect(element);
        }
    }
);

DataStream<JSONObject>                                timeOutDStream =
flatSelectStream.getSideOutput(timeoutTag);

// TODO 11. 合并两个流并将数据写出到 Kafka
DataStream<JSONObject>                                unionDStream =
flatSelectStream.union(timeOutDStream);

unionDStream.print(">>>>");
String targetTopic = "dwd_traffic_user_jump_detail";
unionDStream .map(JSONAware::toJSONString)
    .addSink(MyKafkaUtil.getKafkaProducer(targetTopic));

env.execute();
}
}

```

3.5 测试

- 启动 zk、kafka、flume(f1.sh start)
- 运行 DwdTrafficBaseLogSplit、DwdTrafficUserJumpDetail
- 运行生成日志数据的 jar(lg.sh)
- 创建消费者对象查看 kafka 主题 dwd_traffic_user_jump_detail 的输出

第 4 章 交易域加购事务事实表

4.1 主要任务

提取加购操作生成加购表，并将字典表中的相关维度退化到加购表中，写出到 Kafka 对应主题。

4.2 思路分析

4.2.1 维度关联（维度退化）实现策略分析

本章业务事实表的构建全部使用 FlinkSQL 实现，字典表数据存储在 MySQL 的业务数据库中，要做维度退化，就要将这些数据从 MySQL 中提取出来封装成 FlinkSQL 表，Flink 的 JDBC SQL Connector 可以实现我们的需求。

4.2.2 知识储备

(1) 回顾 DataStreamAPI 的 IntervalJoin

➤ 测试实体类

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Emp {
    private int empno;
    private String ename;
    private int deptno;
    private Long ts;
}

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Dept {
    private int deptno;
    private String dname;
    private Long ts;
}
```

➤ 测试程序

```
package com.atguigu.gmall;

import org.apache.flink.api.common.eventtime.SerializableTimestampAssigner;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.java.tuple.Tuple2;
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.util.Collector;

/**
 * Author: Felix
 * Date: 2022/5/22
 * Desc: 该案例演示了 FlinkDataStream 中的 IntervalJoin
 */
```

```

*/
public class Test01_IntervalJoin {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        SingleOutputStreamOperator<Emp> empDS =
        env.socketTextStream("hadoop102", 8888).map(
            lineStr -> {
                String[] fieldArr = lineStr.split(",");
                return new Emp(Integer.parseInt(fieldArr[0]), fieldArr[1],
                Integer.parseInt(fieldArr[2]), Long.parseLong(fieldArr[3]));
            }
        ).assignTimestampsAndWatermarks(
            WatermarkStrategy
            .<Emp>forMonotonousTimestamps()
            .withTimestampAssigner(
                new SerializableTimestampAssigner<Emp>() {
                    @Override
                    public long extractTimestamp(Emp emp, long recordTimestamp)
                {
                    return emp.getTs();
                }
            }
        );

        SingleOutputStreamOperator<Dept> deptDS =
        env.socketTextStream("hadoop102", 8889).map(
            lineStr -> {
                String[] fieldArr = lineStr.split(",");
                return new Dept(Integer.parseInt(fieldArr[0]), fieldArr[1],
                Long.parseLong(fieldArr[2]));
            }
        ).assignTimestampsAndWatermarks(
            WatermarkStrategy
            .<Dept>forMonotonousTimestamps()
            .withTimestampAssigner(
                new SerializableTimestampAssigner<Dept>() {
                    @Override
                    public long extractTimestamp(Dept dept, long
                recordTimestamp) {
                    return dept.getTs();
                }
            }
        );

        SingleOutputStreamOperator<Tuple2<Emp, Dept>> joinedDS =
        empDS.keyBy(Emp::getDeptno)
            .intervalJoin(deptDS.keyBy(Dept::getDeptno))
            .between(Time.milliseconds(-5), Time.milliseconds(5))
            .process(
                new ProcessJoinFunction<Emp, Dept, Tuple2<Emp, Dept>>() {
                    @Override
                    public void processElement(Emp emp, Dept dept, Context ctx,
                    Collector<Tuple2<Emp, Dept>> out) throws Exception {
                        out.collect(Tuple2.of(emp, dept));
                    }
                }
            );
    }
}

```

```

        joinedDS.print(">>>>");

        env.execute();
    }
}

```

(2) FlinkSQLJoin 以及 LeftJoin 实现原理

```

package com.atguigu.gmall;

import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

import java.time.Duration;

/**
 * Author: Felix
 * Date: 2022/5/22
 * Desc: 该案例演示了 FlinkSQL 中的 join
 * FlinkSQL 各连接方式状态保存时间不一样
 *
 *          左          右
 * 内连接      createAndWrite  createAndWrite
 * 左外连接    readAndWrite   createAndWrite
 * 右外连接    createAndWrite  readAndWrite
 * 全连接      readAndWrite   readAndWrite
 */
public class Test02_FlinkSQLJoin {
    public static void main(String[] args) {
        //TODO 1. 基本环境准备
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        tableEnv.getConfig().setIdleStateRetention(Duration.ofSeconds(20));

        //TODO 2. 创建 socket 流读取指定端口数据 并进行类型转换
        SingleOutputStreamOperator<Emp> empDS =
env.socketTextStream("hadoop102", 8888).map(
    lineStr -> {
        String[] fieldArr = lineStr.split(",");
        return new Emp(Integer.parseInt(fieldArr[0]), fieldArr[1],
Integer.parseInt(fieldArr[2]), Long.parseLong(fieldArr[3]));
    }
);

        SingleOutputStreamOperator<Dept> deptDS =
env.socketTextStream("hadoop102", 8889).map(
    lineStr -> {
        String[] fieldArr = lineStr.split(",");
        return new Dept(Integer.parseInt(fieldArr[0]), fieldArr[1],
Long.parseLong(fieldArr[2]));
    }
);

        //TODO 3. 将流转换为动态表
        tableEnv.createTemporaryView("emp", empDS);
        tableEnv.createTemporaryView("dept", deptDS);
    }
}

```

```

//TODO 4.内连接测试 -- 底层维护两个状态 默认永不过期
//tableEnv.executeSql("select e.empno,e.ename,d.deptno,d.dname from emp
e join dept d on e.deptno = d.deptno").print();

//TODO 5.左外连接测试 左: readAndWrite 右: createAndWrite
//tableEnv.executeSql("select e.empno,e.ename,d.deptno,d.dname from emp
e left join dept d on e.deptno = d.deptno").print();

//TODO 6.右外连接测试
//tableEnv.executeSql("select e.empno,e.ename,d.deptno,d.dname from emp
e right join dept d on e.deptno = d.deptno").print();

//TODO 7.全外连接测试
//tableEnv.executeSql("select e.empno,e.ename,d.deptno,d.dname from emp
e full join dept d on e.deptno = d.deptno").print();

//TODO 8.左连接数据写到 kafka
/*left join 实现过程
假设 A 表作为主表与 B 表做等值左外联。
当 A 表数据进入算子, 而 B 表数据未至时会先生成一条 B 表字段均为 null 的关联数据 ab1, 其
标记为 +I。
其后, B 表数据到来, 会先将之前的数据撤回, 即生成一条与 ab1 内容相同, 但标记为-D 的数据,
再生成一条关联后的数据, 标记为 +I。
这样生成的动态表对应的流称之为回撤流。
这里如果将 left join 之后的数据写到 kafka 主题的话, 会有 null 值,我们不能使用 kafka
connector, 需要使用 kafka upsert connector;
如 果 使 用 new FlinkKafkaConsumer<String>("topic",new
SimpleStringSchema(),prop);消费数据, SimpleStringSchema 进行
序列化的时候, 要求 value 不能为空, 会报错; 所以我们自己重写反序列化方法。*/
tableEnv.executeSql("CREATE TABLE kafka_emp (\n" +
    " `empno` BIGINT,\n" +
    " `ename` String,\n" +
    " `deptno` bigint,\n" +
    " `dname` STRING,\n" +
    " PRIMARY KEY (empno) NOT ENFORCED" +
    ") WITH (\n" +
    " 'connector' = 'upsert-kafka',\n" +
    " 'topic' = 'first',\n" +
    " 'properties.bootstrap.servers' = 'hadoop102:9092',\n" +
    " 'key.format' = 'json',\n" +
    " 'value.format' = 'json'\n" +
    ")");

tableEnv.executeSql("insert into kafka_emp select
e.empno,e.ename,d.deptno,d.dname from emp e left join dept d on e.deptno =
d.deptno");
}
}

```

(3) JDBC SQL Connector

JDBC 连接器可以让 Flink 程序从拥有 JDBC 驱动的任何关系型数据库中读取数据

或将数据写入数据库。

如果在 Flink SQL 表的 DDL 语句中定义了主键，则会以 upsert 模式将流中数据写入数据库，此时流中可以存在 UPDATE/DELETE（更新/删除）类型的数据。否则，会以 append 模式将数据写出到数据库，此时流中只能有 INSERT（插入）类型的数据。

DDL 用法实例如下

```
CREATE TABLE MyUserTable (  
  id BIGINT,  
  name STRING,  
  age INT,  
  status BOOLEAN,  
  PRIMARY KEY (id) NOT ENFORCED  
) WITH (  
  'connector' = 'jdbc',  
  'url' = 'jdbc:mysql://localhost:3306/mydatabase',  
  'table-name' = 'users'  
);
```

(4) Lookup Cache

JDBC 连接器可以作为时态表关联中的查询数据源（又称维表）。目前，仅支持同步查询模式。

默认情况下，查询缓存(Lookup Cache)未被启用，需要设置 lookup.cache.max-rows 和 lookup.cache.ttl 参数来启用此功能。

Lookup 缓存是用来提升有 JDBC 连接器参与的时态关联性能的。默认情况下，缓存未启用，所有的请求会被发送到外部数据库。当缓存启用时，每个进程（即 TaskManager）维护一份缓存。收到请求时，Flink 会先查询缓存，如果缓存未命中才会向外部数据库发送请求，并用查询结果更新缓存。如果缓存中的记录条数达到了 lookup.cache.max-rows 规定的最大行数时将清除存活时间最久的记录。如果缓存中的记录存活时间超过了 lookup.cache.ttl 规定的最大存活时间，同样会被清除。

缓存中的记录未必是最新的，可以将 lookup.cache.ttl 设置为一个更小的值来获得时效性更好的数据，但这样做会增加发送到数据库的请求数量。所以需要在吞吐量和正确性之

间寻求平衡。

(5) Lookup Join

Lookup Join 通常在 Flink SQL 表和外部系统查询结果关联时使用。这种关联要求一张表（主表）有处理时间字段，而另一张表（维表）由 Lookup 连接器生成。

Lookup Join 做的是维度关联，而维度数据是有时效性的，那么我们就需要一个时间字段来对数据的版本进行标识。因此，Flink 要求我们提供处理时间用作版本字段。

此处选择调用 PROCTIME() 函数获取系统时间，将其作为处理时间字段。

示例如下

```
package com.atguigu.gmall;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/**
 * Author: Felix
 * Date: 2022/5/22
 * Desc: 该案例演示了 Flink Jdbc Connector 以及 LookupJoin
 */
public class Test03_JdbcConnector {
    public static void main(String[] args) {
        //TODO 1. 基本环境准备
        StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        //TODO 2. 从 kafka 读取数据并创建表
        tableEnv.executeSql("CREATE TABLE emp (\n" +
            "    empno BIGINT, \n" +
            "    ename STRING,\n" +
            "    deptno bigINT,\n" +
            "    proc_time as proctime()\n" +
            ") WITH (\n" +
            "    'connector' = 'kafka',\n" +
            "    'topic' = 'first',\n" +
            "    'properties.bootstrap.servers' = 'hadoop102:9092',\n" +
            "    'properties.group.id' = 'testGroup',\n" +
            "    'scan.startup.mode' = 'latest-offset',\n" +
            "    'value.format' = 'json'\n" +
            ")");

        //TODO 5. 使用 jdbc 连接器创建动态表映射 MySQL 数据库中的表---参考官网文档
        tableEnv.executeSql("CREATE TABLE dept (" +
            "    deptno bigINT, " +
            "    dname STRING, " +
            "    ts bigINT " +
```

```

        ") WITH (" +
        " 'connector' = 'jdbc'," +
        " 'url' = 'jdbc:mysql://hadoop102:3306/gmall2022_config'," +
        " 'username' = 'root'," +
        " 'password' = '123456'," +
        " 'table-name' = 't_dept'" +
        ")");

//TODO 6.lookup join 测试
tableEnv.executeSql("select e.empno,e.ename,d.deptno,d.dname from emp e "
+
        "left join dept " +
        "FOR SYSTEM_TIME AS OF proc_time AS d " +
        "on e.deptno = d.deptno").print();

```

(6) JDBC SQL Connector 参数

- connector: 连接器类型, 此处为 jdbc
- url: 数据库 url
- table-name: 数据库中表名
- lookup.cache.max-rows: lookup 缓存中的最大记录条数
- lookup.cache.ttl: lookup 缓存中每条记录的最大存活时间
- username: 访问数据库的用户名
- password: 访问数据库的密码
- driver: 数据库驱动, 注意: 通常注册驱动可以省略, 但是自动获取的驱动是 `com.mysql.jdbc.Driver`, Flink CDC 2.1.0 要求 mysql 驱动版本必须为 8 及以上, 在 `mysql-connector-8.x` 中该驱动已过时, 新的驱动为 `com.mysql.cj.jdbc.Driver`。

省略该参数控制台打印的警告如下

```

Loading class `com.mysql.jdbc.Driver`. This is deprecated. The new
driver class is `com.mysql.cj.jdbc.Driver`. The driver is
automatically registered via the SPI and manual loading of the driver
class is generally unnecessary.

```

4.2.3 执行步骤

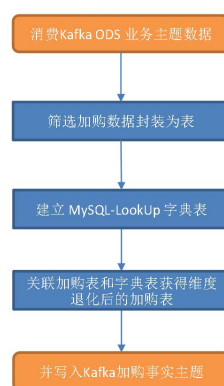
- 设置表状态的 ttl

look up 表的 cache 是由建表时指定的相关参数决定的，与普通连接的 ttl 无关。

- 读取购物车表数据。
- 建立 Mysql-LookUp 字典表。
- 关联购物车表和字典表，维度退化。

4.3 图解

业务数据DWD层加购业务代码流程图



让天下没有难学的技术

4.4 代码

(1) 补充 Flink SQL 相关依赖

要执行 Flink SQL 程序，补充相关依赖。JDBC SQL Connector 需要的依赖包含在

Flink CDC 需要的依赖中，不可重复引入。

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner-blink_${scala.version}</artifactId>
  <version>${flink.version}</version>
</dependency>
```

(2) 主程序

```
package com.atguigu.gmall.realtime.app.dwd.db;

import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import com.atguigu.gmall.realtime.util.MySqlUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
```



```

import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

import java.time.ZoneId;

/**
 * Author: Felix
 * Date: 2022/5/10
 * Desc: 交易域加购事务事实表
 */
public class DwdTradeCartAdd {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // 设定 Table 中的时区为本地时区
        tableEnv.getConfig().setLocalTimezone(ZoneId.of("GMT+8"));

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 从 Kafka 读取业务数据, 封装为 Flink SQL 表
        tableEnv.executeSql(" " +
            "create table topic_db(" +
            "`database` string," +
            "`table` string," +
            "`type` string," +
            "`data` map<string, string>," +
            "`old` map<string, string>," +
            "`ts` string," +
            "`proc_time` as PROCTIME()" +
            ")" + MyKafkaUtil.getKafkaDDL("topic_db", "dwd_trade_cart_add"));

        // TODO 4. 从业务表中过滤出购物车表数据
        Table cartAdd = tableEnv.sqlQuery(
            "select " +
            "data['id'] id," +
            "data['user_id'] user_id," +
            "data['sku_id'] sku_id," +
            "data['source_id'] source_id," +
            "data['source_type'] source_type," +
            "if(`type` = 'insert'," +
            "data['sku_num'], cast((cast(data['sku_num'] as int) -
cast(`old`['sku_num'] as int)) as string)) sku_num," +
            "ts," +
            "proc_time" +
            " from `topic_db` " +
            " where `table` = 'cart_info' " +
            " and (`type` = 'insert' " +
            " or (`type` = 'update' " +
            " and `old`['sku_num'] is not null " +
            " and cast(data['sku_num'] as int) > cast(`old`['sku_num'] as int)))");
        tableEnv.createTemporaryView("cart_add", cartAdd);

        // TODO 5. 建立 MySQL-LookUp 字典表
        tableEnv.executeSql(MySqlUtil.getBaseDicLookUpDDL());
    }
}

```

```

// TODO 6. 关联两张表获得加购明细表
Table resultTable = tableEnv.sqlQuery("select " +
    "cadd.id," +
    "user_id," +
    "sku_id," +
    "source_id," +
    "source_type," +
    "dic_name source_type_name," +
    "sku_num," +
    "ts" +
    " from cart_add cadd " +
    " join base_dic for system_time as of cadd.proc_time as dic " +
    " on cadd.source_type=dic.dic_code");
tableEnv.createTemporaryView("result_table", resultTable);

//tableEnv.executeSql("select * from result_table").print();

// TODO 7. 建立 Upsert-Kafka dwd_trade_cart_add 表
tableEnv.executeSql("" +
    "create table dwd_trade_cart_add(" +
    "id string," +
    "user_id string," +
    "sku_id string," +
    "source_id string," +
    "source_type_code string," +
    "source_type_name string," +
    "sku_num string," +
    "ts string," +
    "primary key(id) not enforced" +
    ")" + MyKafkaUtil.getUpsertKafkaDDL("dwd_trade_cart_add"));

// TODO 8. 将关联结果写入 Upsert-Kafka 表
tableEnv.executeSql("" +
    "insert into dwd_trade_cart_add select * from result_table");
}
}

```

(3) 在 KafkaUtil 中补充 getKafkaDDL 方法和 getUpsertKafkaDDL 方法

```

/**
 * Kafka-Source DDL 语句
 *
 * @param topic 数据源主题
 * @param groupId 消费者组
 * @return 拼接好的 Kafka 数据源 DDL 语句
 */
public static String getKafkaDDL(String topic, String groupId) {

    return " with ('connector' = 'kafka', " +
        "'topic' = '" + topic + "', " +
        "'properties.bootstrap.servers' = '" + BOOTSTRAP_SERVERS + "', " +
        "'properties.group.id' = '" + groupId + "', " +
        "'format' = 'json', " +
        "'scan.startup.mode' = 'group-offsets')";
}

/**

```

```

* FlinkSQL Kafka-Sink DDL 语句
* @param topic 输出到 Kafka 的目标主题
* @return 拼接好的 Kafka-Sink DDL 语句
*/
public static String getUpsertKafkaDDL(String topic) {

    return "WITH ( " +
        " 'connector' = 'upsert-kafka', " +
        " 'topic' = '" + topic + "', " +
        " 'properties.bootstrap.servers' = '" + BOOTSTRAP_SERVERS + "', " +
        " 'key.format' = 'json', " +
        " 'value.format' = 'json' " +
        " );";
}

```

(4) 创建 MysqlUtil 工具类

封装 mysqlLookupTableDDL() 方法和 getBaseDicLookupDDL() 方法，用于将 MySQL 数据库中的字典表读取为 Flink LookUp 表，以便维度退化。

```

package com.atguigu.gmall.realtime.util;

/**
 * Author: Felix
 * Date: 2022/5/11
 * Desc: 操作MySQL 工具类
 */
public class MysqlUtil {
    public static String getBaseDicLookupDDL() {

        return "create table `base_dic` (" +
            "`dic_code` string," +
            "`dic_name` string," +
            "`parent_code` string," +
            "`create_time` timestamp," +
            "`operate_time` timestamp," +
            "primary key(`dic_code`) not enforced" +
            ")" + mysqlLookupTableDDL("base_dic");
    }

    public static String mysqlLookupTableDDL(String tableName) {

        String ddl = "WITH ( " +
            "'connector' = 'jdbc'," +
            "'url' = 'jdbc:mysql://hadoop202:3306/gmall2022'," +
            "'table-name' = '" + tableName + "'," +
            "'lookup.cache.max-rows' = '500'," +
            "'lookup.cache.ttl' = '1 hour'," +
            "'username' = 'root'," +
            "'password' = '123456'," +
            "'driver' = 'com.mysql.cj.jdbc.Driver'" +
            " );";
        return ddl;
    }
}

```

4.5 测试

- 启动 zk、kafka、maxwell
- 运行 DwdTradeCartAdd
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_trade_cart_add 的输出

第 5 章 交易域订单预处理表

5.1 主要任务

经过分析，订单明细表和取消订单明细表的数据来源、表结构都相同，差别只在业务过程和过滤条件，为了减少重复计算，将两张表公共的关联过程提取出来，形成订单预处理表。

关联订单明细表、订单表、订单明细活动关联表、订单明细优惠券关联表四张事实业务表和字典表（维度业务表）形成订单预处理表，写入 Kafka 对应主题。

本节形成的预处理表中要保留 ods 层过来的订单表 type 和 old 字段，用于过滤订单明细数据和取消订单明细数据。

5.2 思路分析

5.2.1 知识储备

(1) left join 实现过程

假设 A 表作为主表与 B 表做等值左外联。当 A 表数据进入算子，而 B 表数据未至时会先生成一条 B 表字段均为 null 的关联数据 ab1，其标记为 +l。其后，B 表数据到

来，会先将之前的数据撤回，即生成一条与 ab1 内容相同，但标记为 -D 的数据，再生成一条关联后的数据，标记为 +I。这样生成的动态表对应的流称之为回撤流。

(2) Kafka SQL Connector

KafkaSQLConnector 分为 KafkaSQLConnector 和 UpsertKafkaSQLConnector

① 功能

- Upsert Kafka Connector 支持以 upsert 方式从 Kafka topic 中读写数据
- Kafka Connector 支持从 Kafka topic 中读写数据

② 区别

➤ 建表语句的主键

- Kafka Connector 要求表不能有主键，如果设置了主键，报错信息如下

```
Caused by: org.apache.flink.table.api.ValidationException: The Kafka table 'default_catalog.default_database.normal_sink_topic' with 'json' format doesn't support defining PRIMARY KEY constraint on the table, because it can't guarantee the semantic of primary key.
```

- 而 UpsertKafkaConnector 要求表必须有主键，如果没有设置主键，报错信息如下

```
Caused by: org.apache.flink.table.api.ValidationException: 'upsert-kafka' tables require to define a PRIMARY KEY constraint. The PRIMARY KEY specifies which columns should be read from or write to the Kafka message key. The PRIMARY KEY also defines records in the 'upsert-kafka' table should update or delete on which keys.
```

- 语法：primary key(id) not enforced

注意：not enforced 表示不对来往数据做约束校验，Flink 并不是数据的主人，因此只支持 not enforced 模式

如果没有 not enforced，报错信息如下

```
Exception in thread "main"
org.apache.flink.table.api.ValidationException: Flink doesn't support ENFORCED mode for PRIMARY KEY constaint. ENFORCED/NOT ENFORCED controls if the constraint checks are performed on the incoming/outgoing data. Flink does not own the data therefore the only supported mode is the NOT ENFORCED mode
```

- 对表中数据操作类型的要求

- Kafka Connector 不能消费带有 Upsert/Delete 操作类型数据的表，如 left join 生成的动态表。如果对这类表进行消费，报错信息如下

```
Exception in thread "main" org.apache.flink.table.api.TableException:
Table sink 'default_catalog.default_database.normal_sink_topic'
doesn't support consuming update and delete changes which is produced
by node TableSourceScan(table=[[default_catalog, default_database,
Unregistered_DataStream_Source_9]], fields=[l_id, tag_left,
tag_right])
```

- Upsert Kafka Connector 将 INSERT/UPDATE_AFTER 数据作为正常的 Kafka 消息写入，并将 DELETE 数据以 value 为空的 Kafka 消息写入（表示对应 key 的消息被删除）。Flink 将根据主键列的值对数据进行分区，因此同一主键的更新/删除消息将落在同一分区，从而保证同一主键的消息有序。

③ 参数解读

本节需要用到 Kafka 连接器的明细表数据来源于 topic_db 主题，于 Kafka 而言，该主题的数据的操作类型均为 INSERT，所以读取数据使用 Kafka Connector 即可。而由于 left join 的存在，流中存在修改数据，所以写出数据使用 Upsert Kafka Connector。

➤ Kafka Connector 参数

- connector: 指定使用的连接器，对于 Kafka，只用 'kafka'
- topic: 主题
- properties.bootstrap.servers: 以逗号分隔的 Kafka broker 列表。注意：可以通过 properties.* 的方式指定配置项，*的位置用 Kafka 官方规定的配置项的 key 替代。并不是所有的配置都可以通过这种方式配置，因为 Flink 可能会将它们覆盖，如：
'key.deserializer' 和 'value.deserializer'
- properties.group.id: 消费者组 ID
- format: 指定 Kafka 消息中 value 部分的序列化的反序列化方式，'format' 和 'value.format' 二者必有其一

- `scan.startup.mode`: Kafka 消费者启动模式, 取值方式如下
 - ✓ `'earliest-offset'`: 从偏移量最早的位置开始读取数据
 - ✓ `'latest-offset'`: 从偏移量最新的位置开始读取数据
 - ✓ `'group-offsets'`: 从 Zookeeper/Kafka broker 中维护的消费者组偏移量开始读取数据
 - ✓ `'timestamp'`: 从用户为每个分区提供的时间戳开始读取数据
 - ✓ `'specific-offsets'`: 从用户为每个分区提供的偏移量开始读取数据

默认值为 `group-offsets`。要注意: `latest-offset` 与 Kafka 官方提供的配置项 `latest` 不同, Flink 会将偏移量置为最新位置, 覆盖掉 Zookeeper 或 Kafka 中维护的偏移量。与官方提供的 `latest` 相对应的是此处的 `group-offsets`。

➤ Upsert Kafka Connector 参数

- `connector`: 指定使用的连接器, 对于 Upsert Kafka, 使用 `'upsert-kafka'`
- `topic`: 主题
- `properties.bootstrap.servers`: 以逗号分隔的 Kafka broker 列表
- `key.format`: key 的序列化和反序列化格式
- `value.format`: value 的序列化和反序列化格式

(3) FlinkSQL 提供了几个可以获取当前时间戳的函数

- `localtimestamp`: 返回本地时区的当前时间戳, 返回类型为 `TIMESTAMP(3)`。在流处理模式下会对每条记录计算一次时间。而在批处理模式下, 仅在查询开始时计算一次时间, 所有数据使用相同的时间。
- `current_timestamp`: 返回本地时区的当前时间戳, 返回类型为 `TIMESTAMP_LTZ(3)`。在流处理模式下会对每条记录计算一次时间。而在批处理模式下, 仅在查询开始时计算

一次时间，所有数据使用相同的时间。

- `now()`: 与 `current_timestamp` 相同。
- `current_row_timestamp()`: 返回本地时区的当前时间戳，返回类型为 `TIMESTAMP_LTZ(3)`。无论在流处理模式还是批处理模式下，都会对每行数据计算一次时间。

函数测试。查询语句如下。

```
tableEnv.sqlQuery("select localtimeStamp, " +  
    "current_timestamp, " +  
    "now(), " +  
    "current_row_timestamp()")  
    .execute()  
    .print();
```

查询结果如下。

```
+-----+-----+-----+-----+  
|op|      localtimeStamp |      current_timestamp |      EXPR$2 |      EXPR$3 |  
+-----+-----+-----+-----+  
|+I|2022-04-13 20:42:28.529|2022-04-13 20:42:28.529|2022-04-13 20:42:28.529|2022-04-13 20:42:28.529|  
+-----+-----+-----+-----+  
1 row in set
```

动态表属于流处理模式，所以四种函数任选其一即可。此处选择 `current_row_timestamp()`。

5.2.2 执行步骤

预处理表与订单明细事务事实表的区别只在于前者不会按照订单数据的 `type` 和 `old` 字段做过滤，且在表中增加了这两个字段。二者的粒度、聚合逻辑都相同，因此按照订单明细表的思路对预处理表进行分析即可。

(1) 设置 `ttl`;

`ttl` (time-to-live) 即存活时间。表之间做普通关联时，底层会将两张表的数据维护到状态中，默认情况下状态永远不会清空，这样会对内存造成极大的压力。表状态的 `ttl` 是 `Idle` (空闲，即状态未被更新) 状态被保留的最短时间，假设 `ttl` 为 10s，若状态中的数据

在 10s 内未被更新，则未来的某个时间会被清除（故而 ttl 是最短存活时间）。ttl 默认值为 0，表示永远不会清空状态。

下单操作发生时，订单明细表、订单表、订单明细优惠券关联表和订单明细活动关联表的数据操作类型均为 insert，不存在业务上的滞后问题，只考虑可能的数据乱序即可；而取消订单时只有订单表的状态发生变化，要和其它三张表关联，就需要考虑取消订单的延迟，通常在支付前均可取消订单，因此将 ttl 设置为 15min + 5s。

要注意：前文提到，本项目保证了同一分区、同一并行度的数据有序。此处的乱序与之并不冲突，以下单业务过程为例，用户完成下单操作时，订单表中会插入一条数据，订单明细表中会插入与之对应的多条数据，本项目业务数据是按照主键分区进入 Kafka 的，虽然同分区数据有序，但是同一张业务表的数据可能进入多个分区，会乱序。这样一来，订单表数据与对应的订单明细数据可能被属于其它订单的数据“插队”，因而导致主表或从表数据迟到，可能 join 不上，为了应对这种情况，设置乱序程度，让状态中的数据等待一段时间。

(2) 从 Kafka topic_db 主题读取业务数据

这一步要调用 PROCTIME() 函数获取系统时间作为与字典表做 Lookup Join 的处理时间字段

(3) 筛选订单明细表数据

应尽可能保证事实表的粒度为最细粒度，在下单业务过程中，最细粒度的事件为一个订单的一个 SKU 的下单操作，订单明细表的粒度与最细粒度相同，将其作为**主表**

(4) 筛选订单表数据

通过该表获取 user_id 和 province_id。保留 type 字段和 old 字段用于过滤订单明细数据和取消订单明细数据

(5) 筛选订单明细活动关联表数据

通过该表获取活动 id 和活动规则 id

(6) 筛选订单明细优惠券关联表数据;

通过该表获取优惠券 id。

(7) 建立 MySQL-Lookup 字典表

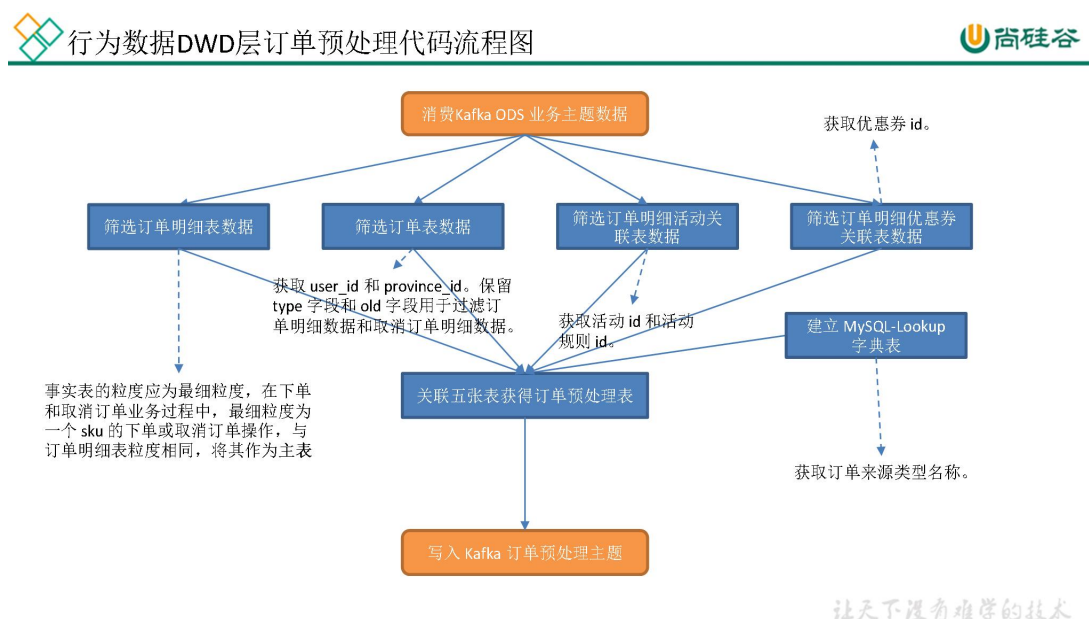
通过字典表获取订单来源类型名称

(8) 关联上述五张表获得订单宽表, 写入 Kafka 主题

事实表的粒度应为最细粒度, 在下单和取消订单业务过程中, 最细粒度为一个 sku 的下单或取消订单操作, 与订单明细表粒度相同, 将其作为主表。

- 订单明细表和订单表的所有记录在另一张表中都有对应数据, 内连接即可。
- 订单明细数据未必参加了活动也未必使用了优惠券, 因此要保留订单明细独有数据, 所以与订单明细活动关联表和订单明细优惠券关联表的关联使用 left join。
- 与字典表的关联是为了获取 source_type 对应的 source_type_name, 订单明细数据在字典表中一定有对应, 内连接即可。

5.3 图解



5.4 代码

```
package com.atguigu.gmall.realtime.app.dwd.db;

import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import com.atguigu.gmall.realtime.util.MySqlUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/**
 * Author: Felix
 * Date: 2022/5/11
 * Desc: 交易域订单预处理表
 */
public class DwdTradeOrderPreProcess {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 从 Kafka 读取业务数据, 封装为 Flink SQL 表
        tableEnv.executeSql("create table topic_db(" +
            "`database` String,\n" +
            "`table` String,\n" +
            "`type` String,\n" +
            "`data` map<String, String>,\n" +
            "`old` map<String, String>,\n" +
            "`proc_time` as PROCTIME(),\n" +
            "`ts` string\n" +
            ") " +
            MyKafkaUtil.getKafkaDDL("topic_db",
            "dwd_trade_order_pre_process"));

        // TODO 4. 读取订单明细表数据
        Table orderDetail = tableEnv.sqlQuery("select \n" +
            "data['id'] id,\n" +
            "data['order_id'] order_id,\n" +
            "data['sku_id'] sku_id,\n" +
            "data['sku_name'] sku_name,\n" +
            "data['create_time'] create_time,\n" +
            "data['source_id'] source_id,\n" +
            "data['source_type'] source_type,\n" +
            "data['sku_num'] sku_num,\n" +
            "cast(cast(data['sku_num'] as decimal(16,2)) * " +
            "cast(data['order_price'] as decimal(16,2)) as String)
            split_original_amount,\n" +
            "data['split_total_amount'] split_total_amount,\n" +
            "data['split_activity_amount'] split_activity_amount,\n" +
            "data['split_coupon_amount'] split_coupon_amount,\n" +
            "ts od_ts,\n" +
            "proc_time\n" +
            "from `topic_db` where `table` = 'order_detail' " +
            "and `type` = 'insert'\n");
        tableEnv.createTemporaryView("order_detail", orderDetail);
```

```

// TODO 5. 读取订单表数据
Table orderInfo = tableEnv.sqlQuery("select \n" +
    "data['id'] id,\n" +
    "data['user_id'] user_id,\n" +
    "data['province_id'] province_id,\n" +
    "data['operate_time'] operate_time,\n" +
    "data['order_status'] order_status,\n" +
    "`type`,\n" +
    "`old`,\n" +
    "ts oi_ts\n" +
    "from `topic_db`\n" +
    "where `table` = 'order_info'\n" +
    "and (`type` = 'insert' or `type` = 'update')");
tableEnv.createTemporaryView("order_info", orderInfo);

// TODO 6. 读取订单明细活动关联表数据
Table orderDetailActivity = tableEnv.sqlQuery("select \n" +
    "data['order_detail_id'] order_detail_id,\n" +
    "data['activity_id'] activity_id,\n" +
    "data['activity_rule_id'] activity_rule_id\n" +
    "from `topic_db`\n" +
    "where `table` = 'order_detail_activity'\n" +
    "and `type` = 'insert'\n");
tableEnv.createTemporaryView("order_detail_activity",
orderDetailActivity);

// TODO 7. 读取订单明细优惠券关联表数据
Table orderDetailCoupon = tableEnv.sqlQuery("select\n" +
    "data['order_detail_id'] order_detail_id,\n" +
    "data['coupon_id'] coupon_id\n" +
    "from `topic_db`\n" +
    "where `table` = 'order_detail_coupon'\n" +
    "and `type` = 'insert'\n");
tableEnv.createTemporaryView("order_detail_coupon", orderDetailCoupon);

// TODO 8. 建立 MySQL-LookUp 字典表
tableEnv.executeSql(MySqlUtil.getBaseDicLookUpDDL());

// TODO 9. 关联五张表获得订单明细表
Table resultTable = tableEnv.sqlQuery("select \n" +
    "od.id,\n" +
    "od.order_id,\n" +
    "od.sku_id,\n" +
    "od.sku_name,\n" +
    "date_format(od.create_time, 'yyyy-MM-dd') date_id,\n" +
    "od.create_time,\n" +
    "od.source_id,\n" +
    "od.source_type,\n" +
    "od.sku_num,\n" +
    "od.split_original_amount,\n" +
    "od.split_activity_amount,\n" +
    "od.split_coupon_amount,\n" +
    "od.split_total_amount,\n" +
    "od.od_ts,\n" +
    "oi.user_id,\n" +
    "oi.province_id,\n" +
    "date_format(oi.operate_time, 'yyyy-MM-dd') operate_date_id,\n" +
    "oi.operate_time,\n" +
    "oi.order_status,\n");

```

```

        "oi.`type`,\n" +
        "oi.`old`,\n" +
        "oi.oi_ts,\n" +

        "act.activity_id,\n" +
        "act.activity_rule_id,\n" +

        "cou.coupon_id,\n" +

        "dic.dic_name source_type_name,\n" +

        "current_row_timestamp() row_op_ts\n" +
        "from order_detail od \n" +
        "join order_info oi\n" +
        "on od.order_id = oi.id\n" +
        "left join order_detail_activity act\n" +
        "on od.id = act.order_detail_id\n" +
        "left join order_detail_coupon cou\n" +
        "on od.id = cou.order_detail_id\n" +
        "join `base_dic` for system_time as of od.proc_time as dic\n" +
        "on od.source_type = dic.dic_code");
tableEnv.createTemporaryView("result_table", resultTable);

// TODO 10. 建立 Upsert-Kafka dwd_trade_order_pre_process 表
tableEnv.executeSql("" +
    "create table dwd_trade_order_pre_process(\n" +
    "id string,\n" +
    "order_id string,\n" +
    "sku_id string,\n" +
    "sku_name string,\n" +
    "date_id string,\n" +
    "create_time string,\n" +
    "source_id string,\n" +
    "source_type string,\n" +
    "sku_num string,\n" +
    "split_original_amount string,\n" +
    "split_activity_amount string,\n" +
    "split_coupon_amount string,\n" +
    "split_total_amount string,\n" +
    "od_ts string,\n" +

    "user_id string,\n" +
    "province_id string,\n" +
    "operate_date_id string,\n" +
    "operate_time string,\n" +
    "order_status string,\n" +
    "`type` string,\n" +
    "`old` map<string,string>,\n" +
    "oi_ts string,\n" +

    "activity_id string,\n" +
    "activity_rule_id string,\n" +

    "coupon_id string,\n" +

    "source_type_name string,\n" +

    "row_op_ts timestamp_ltz(3),\n" +
    "primary key(id) not enforced\n" +
    ") " + MyKafkaUtil.getUpsertKafkaDDL("dwd_trade_order_pre_process"));

```

```

// TODO 11. 将关联结果写入 Upsert-Kafka 表

```

```
tableEnv.executeSql("'" +
    "insert into dwd_trade_order_pre_process \n" +
    "select * from result_table");
}
```

5.5 测试

- 启动 zk、kafka、maxwell
- 运行 DwdTradeOrderPreProcess
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_trade_order_pre_process 的输出

第 6 章 交易域下单事务事实表

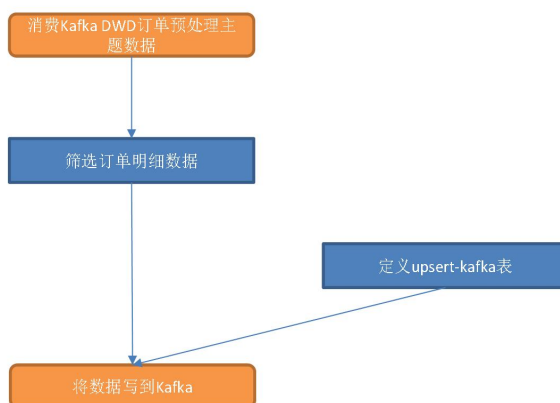
6.1 主要任务

从 Kafka 读取订单预处理表数据，筛选订单明细数据，写入 Kafka 对应主题。

6.2 思路分析

- (1) 从 Kafka dwd_trade_order_pre_process 主题读取订单预处理数据；
- (2) 筛选订单明细数据：新增数据，即订单表操作类型为 insert 的数据即为订单明细数据；
- (3) 写入 Kafka 订单明细主题。

6.3 图解



让天下没有难学的技术

6.4 代码

```
package com.atguigu.gmall.realtime.app.dwd.db;

import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

import java.time.ZoneId;

/**
 * Author: Felix
 * Date: 2022/5/11
 * Desc: 交易域下单事务事实表
 */
public class DwdTradeOrderDetail {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // TODO 2. 启用状态后端(略)

        // TODO 3. 读取 Kafka dwd_trade_order_pre_process 主题数据
        tableEnv.executeSql(" " +
            "create table dwd_trade_order_pre_process(\n" +
            "id string,\n" +
            "order_id string,\n" +
            "user_id string,\n" +
```

```

"order_status string,\n" +
"sku_id string,\n" +
"sku_name string,\n" +
"province_id string,\n" +
"activity_id string,\n" +
"activity_rule_id string,\n" +
"coupon_id string,\n" +
"date_id string,\n" +
"create_time string,\n" +
"operate_date_id string,\n" +
"operate_time string,\n" +
"source_id string,\n" +
"source_type string,\n" +
"source_type_name string,\n" +
"sku_num string,\n" +
"split_original_amount string,\n" +
"split_activity_amount string,\n" +
"split_coupon_amount string,\n" +
"split_total_amount string,\n" +
"`type` string,\n" +
"`old` map<string,string>,\n" +
"od_ts string,\n" +
"oi_ts string,\n" +
"row_op_ts timestamp_ltz(3)\n" +
")" + MyKafkaUtil.getKafkaDDL(
"dwd_trade_order_pre_process", "dwd_trade_order_detail"));

// TODO 4. 过滤下单数据
Table filteredTable = tableEnv.sqlQuery("'" +
"select " +
"id,\n" +
"order_id,\n" +
"user_id,\n" +
"sku_id,\n" +
"sku_name,\n" +
"province_id,\n" +
"activity_id,\n" +
"activity_rule_id,\n" +
"coupon_id,\n" +
"date_id,\n" +
"create_time,\n" +
"source_id,\n" +
"source_type source_type_code,\n" +
"source_type_name,\n" +
"sku_num,\n" +
"split_original_amount,\n" +
"split_activity_amount,\n" +
"split_coupon_amount,\n" +
"split_total_amount,\n" +
"od_ts ts,\n" +
"row_op_ts\n" +
"from dwd_trade_order_pre_process " +
"where `type`='insert'");
tableEnv.createTemporaryView("filtered_table", filteredTable);

// TODO 5. 创建 Kafka 下单明细表
tableEnv.executeSql("'" +
"create table dwd_trade_order_detail(\n" +
"id string,\n" +
"order_id string,\n" +
"user_id string,\n" +
"sku_id string,\n" +

```



```

        "sku_name string,\n" +
        "province_id string,\n" +
        "activity_id string,\n" +
        "activity_rule_id string,\n" +
        "coupon_id string,\n" +
        "date_id string,\n" +
        "create_time string,\n" +
        "source_id string,\n" +
        "source_type_code string,\n" +
        "source_type_name string,\n" +
        "sku_num string,\n" +
        "split_original_amount string,\n" +
        "split_activity_amount string,\n" +
        "split_coupon_amount string,\n" +
        "split_total_amount string,\n" +
        "ts string,\n" +
        "row_op_ts timestamp_ltz(3),\n" +
        "primary key(id) not enforced\n" +
        ")") + MyKafkaUtil.getUpsertKafkaDDL("dwd_trade_order_detail"));

// TODO 6. 将数据写出到 Kafka
tableEnv.executeSql("insert into dwd_trade_order_detail select * from
filtered_table");
    }
}

```

6.5 测试

- 启动 zk、kafka、maxwell
- 运行 DwdTradeOrderPreProcess、DwdTradeOrderDetail
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_trade_order_detail 的输出

第 7 章 交易域取消订单事务事实表

7.1 主要任务

从 Kafka 读取订单预处理表数据，筛选取消订单明细数据，写入 Kafka 对应主题。

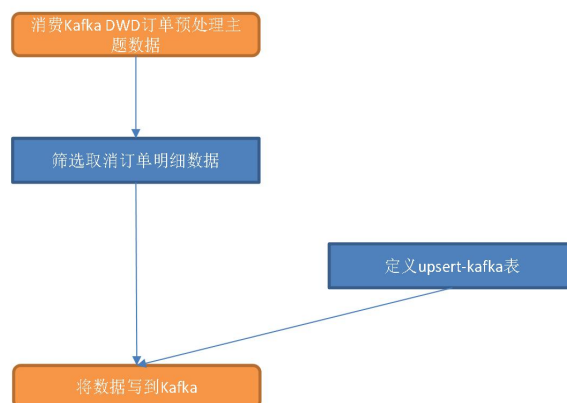
7.2 思路分析

实现步骤

- (1) 从 Kafka dwd_trade_order_pre_process 主题读取订单预处理数据;
- (2) 筛选取消订单明细数据: 保留修改了 order_status 字段且修改后该字段值为 "1003" 的数据;
- (3) 写入 Kafka 取消订单主题。

7.3 图解

行为数据DWD层取消订单明细代码流程图



让天下没有难学的技术

7.4 代码

```
package com.atguigu.gmall.realtime.app.dwd.db;

import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

import java.time.ZoneId;

/**
 * Author: Felix
 * Date: 2022/5/11
 */
```

```

* Desc: 交易域取消订单事务事实表
*/
public class DwdTradeOrderCancelDetail {
    public static void main(String[] args) throws Exception {
        // TODO 1. 环境准备
        StreamExecutionEnvironment env
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 读取 Kafka dwd_trade_order_pre_process 主题数据
        tableEnv.executeSql("" +
            "create table dwd_trade_order_pre_process(\n" +
            "id string,\n" +
            "order_id string,\n" +
            "user_id string,\n" +
            "order_status string,\n" +
            "sku_id string,\n" +
            "sku_name string,\n" +
            "province_id string,\n" +
            "activity_id string,\n" +
            "activity_rule_id string,\n" +
            "coupon_id string,\n" +
            "date_id string,\n" +
            "create_time string,\n" +
            "operate_date_id string,\n" +
            "operate_time string,\n" +
            "source_id string,\n" +
            "source_type string,\n" +
            "source_type_name string,\n" +
            "sku_num string,\n" +
            "split_original_amount string,\n" +
            "split_activity_amount string,\n" +
            "split_coupon_amount string,\n" +
            "split_total_amount string,\n" +
            "`type` string,\n" +
            "`old` map<string,string>,\n" +
            "od_ts string,\n" +
            "oi_ts string,\n" +
            "row_op_ts timestamp_ltz(3)\n" +
            ")") + MyKafkaUtil.getKafkaDDL(
            "dwd_trade_order_pre_process", "dwd_trade_cancel_detail"));

        // TODO 4. 筛选取消订单明细数据
        Table filteredTable = tableEnv.sqlQuery("" +
            "select\n" +
            "id,\n" +
            "order_id,\n" +
            "user_id,\n" +
            "sku_id,\n" +
            "sku_name,\n" +
            "province_id,\n" +
            "activity_id,\n" +
            "activity_rule_id,\n" +
            "coupon_id,\n" +
            "operate_date_id date_id,\n" +
            "operate_time cancel_time,\n" +
            "source_id,\n" +
            "source_type source_type_code,\n" +

```

```

        "source_type_name,\n" +
        "sku_num,\n" +
        "split_original_amount,\n" +
        "split_activity_amount,\n" +
        "split_coupon_amount,\n" +
        "split_total_amount,\n" +
        "oi_ts ts,\n" +
        "row_op_ts\n" +
        "from dwd_trade_order_pre_process\n" +
        "where `type` = 'update'\n" +
        "and `old`['order_status'] is not null\n" +
        "and order_status = '1003'");
tableEnv.createTemporaryView("filtered_table", filteredTable);

// TODO 5. 建立 Upsert-Kafka dwd_trade_cancel_detail 表
tableEnv.executeSql("create table dwd_trade_cancel_detail(\n" +
    "id string,\n" +
    "order_id string,\n" +
    "user_id string,\n" +
    "sku_id string,\n" +
    "sku_name string,\n" +
    "province_id string,\n" +
    "activity_id string,\n" +
    "activity_rule_id string,\n" +
    "coupon_id string,\n" +
    "date_id string,\n" +
    "cancel_time string,\n" +
    "source_id string,\n" +
    "source_type_code string,\n" +
    "source_type_name string,\n" +
    "sku_num string,\n" +
    "split_original_amount string,\n" +
    "split_activity_amount string,\n" +
    "split_coupon_amount string,\n" +
    "split_total_amount string,\n" +
    "ts string,\n" +
    "row_op_ts timestamp_ltz(3),\n" +
    "primary key(id) not enforced\n" +
    ") " + MyKafkaUtil.getUpsertKafkaDDL("dwd_trade_cancel_detail"));

// TODO 6. 将数据写出到 Kafka
tableEnv.executeSql(
    "insert into dwd_trade_cancel_detail select * from filtered_table");
}
}

```

7.5 测试

- 启动 zk、kafka、maxwell
- 运行 DwdTradeOrderPreProcess、DwdTradeOrderCancelDetail
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_trade_cancel_detail 的输出

第 8 章 交易域支付成功事务事实表

8.1 主要任务

从 Kafka topic_db 主题筛选支付成功数据、从 dwd_trade_order_detail 主题中读取订单事实数据、MySQL-LookUp 字典表，关联三张表形成支付成功宽表，写入 Kafka 支付成功主题。

8.2 思路分析

(1) 设置 ttl

支付成功事务事实表需要将业务数据库中的支付信息表 payment_info 数据与订单明细表关联。订单明细数据是在下单时生成的，经过一系列的处理进入订单明细主题，通常支付操作在下单后 15min 内完成即可，因此，支付明细数据可能比订单明细数据滞后 15min。考虑到可能的乱序问题，ttl 设置为 15min + 5s。

(2) 获取订单明细数据

用户必然要先下单才有可能支付成功，因此支付成功明细数据集必然是订单明细数据集的子集。

(3) 筛选支付表数据

获取支付类型、回调时间（支付成功时间）、支付成功时间戳。

生产环境下，用户支付后，业务数据库的支付表会插入一条数据，此时的回调时间和回调内容为空。通常底层会调用第三方支付接口，接口会返回回调信息，如果支付成功则回调信息不为空，此时会更新支付表，补全回调时间和回调内容字段的值，并将 payment_status 字段的值修改为支付成功对应的状态码（本项目为 1602）。支付成功之后，支付表数据不

会发生变化。因此，只要操作类型为 update 且状态码为 1602 即为支付成功数据。

由上述分析可知，支付成功对应的业务数据库变化日志应满足两个条件：

- payment_status 字段的值为 1602；
- 操作类型为 update。

注：1602 在字典表中并没有对应记录，且 payment_info 中 payment_status 字段的值均为 null，这是模拟数据的问题，并不影响业务逻辑，无须深究。为了看到效果，注释对条件（1）的判断。此外，模拟的 payment_info 表数据没有上述提到的变化，只在支付成功时插入一条数据，此时的 callback_time 字段值已经不为 null，即该表中的所有数据均为支付成功数据。为了看到效果，注释对条件（2）的判断。

本程序为了去除重复数据，在关联后的宽表中补充了处理时间字段，DWS 层将进行详细介绍。支付成功表是由支付成功数据与订单明细做内连接，而后与字典表做 LookUp Join 得来。这个过程中不会出现回撤数据，关联后表的重复数据来源于订单明细表，所以应按照订单明细表的处理时间字段去重，故支付成功明细表的 row_op_ts 取自订单明细表。

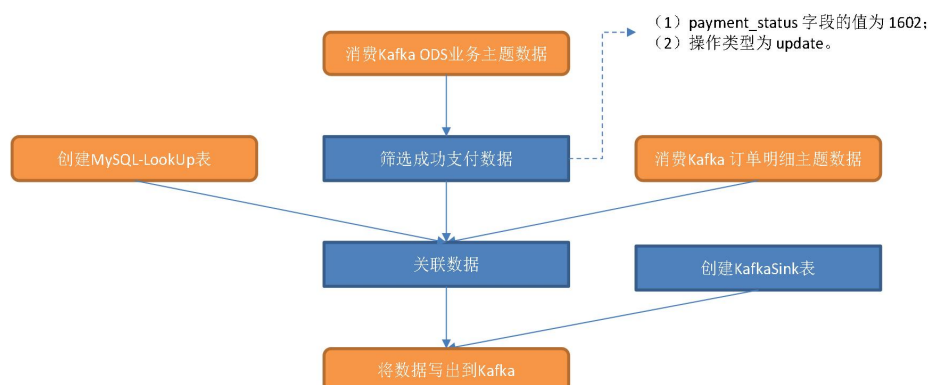
(4) 构建 MySQL-LookUp 字典表

(5) 关联上述三张表形成支付成功宽表，写入 Kafka 支付成功主题

支付成功业务过程的最细粒度为一个 sku 的支付成功记录，order_detail 表的粒度与最细粒度相同，将其作为主表。

- order_detail 表在 payment_info 表中必然存在对应数据，主表不存在独有数据，因此通过内连接与订单明细表关联；
- 与字典表的关联是为了获取 payment_type 对应的支付类型名称，主表不存在独有数据，通过内连接与字典表关联。下文与字典表的关联同理，不再赘述。

8.3 图解



让天下没有难学的技术

8.4 代码

```
package com.atguigu.gmall.realtime.app.dwd.db;

import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import com.atguigu.gmall.realtime.util.MySqlUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

import java.time.ZoneId;

/**
 * Author: Felix
 * Date: 2022/5/11
 * Desc: 交易域支付成功事务事实表
 */
public class DwdTradePayDetailSuc {
    public static void main(String[] args) throws Exception {

        // TODO 1. 基本环境准备
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);
        tableEnv.getConfig().setIdleStateRetention(Duration.ofSeconds(60*15));

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 读取 Kafka dwd_trade_order_detail 主题数据, 封装为 Flink SQL 表
        tableEnv.executeSql(" +
```

```

        "create table dwd_trade_order_detail(\n" +
        "id string,\n" +
        "order_id string,\n" +
        "user_id string,\n" +
        "sku_id string,\n" +
        "sku_name string,\n" +
        "province_id string,\n" +
        "activity_id string,\n" +
        "activity_rule_id string,\n" +
        "coupon_id string,\n" +
        "date_id string,\n" +
        "create_time string,\n" +
        "source_id string,\n" +
        "source_type_code string,\n" +
        "source_type_name string,\n" +
        "sku_num string,\n" +
        "split_original_amount string,\n" +
        "split_activity_amount string,\n" +
        "split_coupon_amount string,\n" +
        "split_total_amount string,\n" +
        "ts string,\n" +
        "row_op_ts timestamp_ltz(3)\n" +
        ") " + MyKafkaUtil.getKafkaDDL("dwd_trade_order_detail",
"dwd_trade_pay_detail_suc"));

// TODO 4. 从 Kafka 读取业务数据, 封装为 Flink SQL 表
tableEnv.executeSql("create table topic_db(" +
    "`database` String,\n" +
    "`table` String,\n" +
    "`type` String,\n" +
    "`data` map<String, String>,\n" +
    "`old` map<String, String>,\n" +
    "`proc_time` as PROCTIME(),\n" +
    "`ts` string\n" +
    ") " + MyKafkaUtil.getKafkaDDL("topic_db",
"dwd_trade_pay_detail_suc"));

// TODO 5. 筛选支付成功数据
Table paymentInfo = tableEnv.sqlQuery("select\n" +
    "data['user_id'] user_id,\n" +
    "data['order_id'] order_id,\n" +
    "data['payment_type'] payment_type,\n" +
    "data['callback_time'] callback_time,\n" +
    "`proc_time`,\n" +
    "ts\n" +
    "from topic_db\n" +
    "where `table` = 'payment_info'\n" +
    +
    "and `type` = 'update'\n" +
    "and data['payment_status']='1602'"
);
tableEnv.createTemporaryView("payment_info", paymentInfo);

// TODO 6. 建立 MySQL-LookUp 字典表
tableEnv.executeSql(MySqlUtil.getBaseDicLookupDDL());

// TODO 7. 关联 3 张表获得支付成功宽表
Table resultTable = tableEnv.sqlQuery("" +
    "select\n" +
    "od.id order_detail_id,\n" +

```



```

        "od.order_id,\n" +
        "od.user_id,\n" +
        "od.sku_id,\n" +
        "od.sku_name,\n" +
        "od.province_id,\n" +
        "od.activity_id,\n" +
        "od.activity_rule_id,\n" +
        "od.coupon_id,\n" +
        "pi.payment_type payment_type_code,\n" +
        "dic.dic_name payment_type_name,\n" +
        "pi.callback_time,\n" +
        "od.source_id,\n" +
        "od.source_type_code,\n" +
        "od.source_type_name,\n" +
        "od.sku_num,\n" +
        "od.split_original_amount,\n" +
        "od.split_activity_amount,\n" +
        "od.split_coupon_amount,\n" +
        "od.split_total_amount split_payment_amount,\n" +
        "pi.ts,\n" +
        "od.row_op_ts row_op_ts\n" +
        "from dwd_trade_order_detail od\n" +
        "join payment_info pi\n" +
        "on pi.order_id = od.order_id\n" +
        " join `base_dic` for system_time as of pi.proc_time as dic\n" +
        "on pi.payment_type = dic.dic_code");
tableEnv.createTemporaryView("result_table", resultTable);

```

// TODO 8. 创建 Kafka dwd_trade_pay_detail 表

```

tableEnv.executeSql("create table dwd_trade_pay_detail_suc(\n" +
    "order_detail_id string,\n" +
    "order_id string,\n" +
    "user_id string,\n" +
    "sku_id string,\n" +
    "sku_name string,\n" +
    "province_id string,\n" +
    "activity_id string,\n" +
    "activity_rule_id string,\n" +
    "coupon_id string,\n" +
    "payment_type_code string,\n" +
    "payment_type_name string,\n" +
    "callback_time string,\n" +
    "source_id string,\n" +
    "source_type_code string,\n" +
    "source_type_name string,\n" +
    "sku_num string,\n" +
    "split_original_amount string,\n" +
    "split_activity_amount string,\n" +
    "split_coupon_amount string,\n" +
    "split_payment_amount string,\n" +
    "ts string,\n" +
    "row_op_ts timestamp_ltz(3),\n" +
    "primary key(order_detail_id) not enforced\n" +
    ")") + MyKafkaUtil.getUpsertKafkaDDL("dwd_trade_pay_detail_suc"));

```

// TODO 9. 将关联结果写入 Upsert-Kafka 表

```

tableEnv.executeSql("" +
    "insert into dwd_trade_pay_detail_suc select * from result_table");
}

```

```

}

```

8.5 测试

- 启动 zk、kafka、maxwell
- 运行 DwdTradeOrderPreProcess、DwdTradeOrderDetail、DwdTradePayDetailSuc
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_trade_pay_detail_suc 的输出

第 9 章 交易域退单事务事实表

9.1 主要任务

从 Kafka 读取业务数据，筛选退单表数据，筛选满足条件的订单表数据，建立 MySQL-Lookup 字典表，关联三张表获得退单明细宽表。

9.2 思路分析

(1) 设置 ttl

用户执行一次退单操作时，order_refund_info 会插入多条数据，同时 order_info 表的一条对应数据会发生修改，所以两张表不存在业务上的时间滞后问题，因此仅考虑可能的乱序即可，ttl 设置为 5s。

(2) 筛选退单表数据

退单业务过程最细粒度的操作为一个订单中一个 SKU 的退单操作，退单表粒度与最细粒度相同，将其作为**主表**。

(3) 筛选订单表数据

获取 province_id 维度。退单操作发生时，订单表的 order_status 字段值会由 1002

(已支付)更新为 1005 (退款中)。订单表中的数据要满足三个条件:

- order_status 为 1005 (退款中);
- 操作类型为 update;
- 更新的字段为 order_status。

该字段发生变化时,变更数据中 old 字段下 order_status 的值不为 null(为 1002)。

(4) 建立 MySQL-Lookup 字典表

获取退款类型名称和退款原因类型名称。

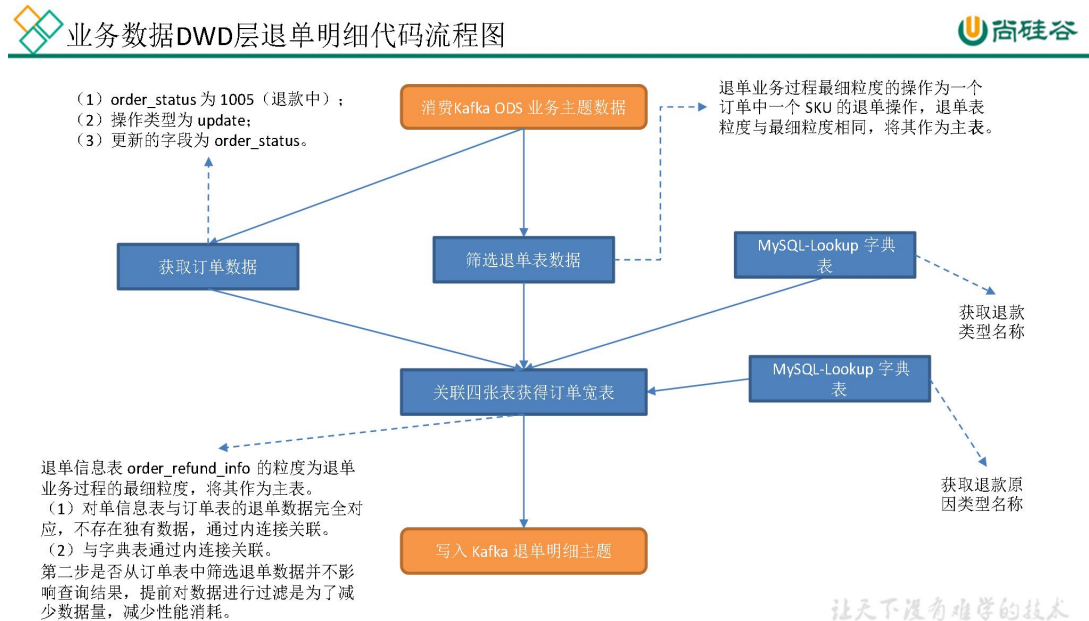
(5) 关联这几张表获得退单明细宽表, 写入 Kafka 退单明细主题

退单信息表 order_refund_info 的粒度为退单业务过程的最细粒度, 将其作为主表。

- 退单信息表与订单表的退单数据完全对应, 不存在独有数据, 通过内连接关联
- 与字典表通过内连接关联

第三步是否对订单表数据筛选并不影响查询结果, 提前对数据进行过滤是为了减少数据量, 减少性能消耗。下文同理, 不再赘述。

9.3 图解



9.4 代码

主程序

```
package com.atguigu.gmall.realtime.app.dwd.db;
import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import com.atguigu.gmall.realtime.util.MySqlUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/**
 * Author: Felix
 * Date: 2022/5/12
 * Desc: 交易域退单事务事实表
 */
public class DwdTradeOrderRefund {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 从 Kafka 读取 topic_db 数据, 封装为 Flink SQL 表
        tableEnv.executeSql(MyKafkaUtil.getTopicDbDDL("dwd_trade_order_refund"));
```

// TODO 4. 读取退单表数据

```
Table orderRefundInfo = tableEnv.sqlQuery("select\n" +  
    "data['id'] id,\n" +  
    "data['user_id'] user_id,\n" +  
    "data['order_id'] order_id,\n" +  
    "data['sku_id'] sku_id,\n" +  
    "data['refund_type'] refund_type,\n" +  
    "data['refund_num'] refund_num,\n" +  
    "data['refund_amount'] refund_amount,\n" +  
    "data['refund_reason_type'] refund_reason_type,\n" +  
    "data['refund_reason_txt'] refund_reason_txt,\n" +  
    "data['create_time'] create_time,\n" +  
    "proc_time,\n" +  
    "ts\n" +  
    "from topic_db\n" +  
    "where `table` = 'order_refund_info'\n" +  
    "and `type` = 'insert'");  
tableEnv.createTemporaryView("order_refund_info", orderRefundInfo);
```

// TODO 5. 读取订单表数据, 筛选退单数据

```
Table orderInfo = tableEnv.sqlQuery("select\n" +  
    "data['id'] id,\n" +  
    "data['province_id'] province_id,\n" +  
    "`old`\n" +  
    "from topic_db\n" +  
    "where `table` = 'order_info'\n" +  
    "and `type` = 'update'\n" +  
    "and data['order_status']='1005'\n" +  
    "and `old`['order_status'] is not null");  
  
tableEnv.createTemporaryView("order_info", orderInfo);
```

// TODO 6. 建立 MySQL-LookUp 字典表

```
tableEnv.executeSql(MySqlUtil.getBaseDicLookUpDDL());
```

// TODO 7. 关联三张表获得退单宽表

```
Table resultTable = tableEnv.sqlQuery("select \n" +  
    "ri.id,\n" +  
    "ri.user_id,\n" +  
    "ri.order_id,\n" +  
    "ri.sku_id,\n" +  
    "oi.province_id,\n" +  
    "date_format(ri.create_time,'yyyy-MM-dd') date_id,\n" +  
    "ri.create_time,\n" +  
    "ri.refund_type,\n" +  
    "type_dic.dic_name,\n" +  
    "ri.refund_reason_type,\n" +  
    "reason_dic.dic_name,\n" +  
    "ri.refund_reason_txt,\n" +  
    "ri.refund_num,\n" +  
    "ri.refund_amount,\n" +  
    "ri.ts,\n" +  
    "current_row_timestamp() row_op_ts\n" +  
    "from order_refund_info ri\n" +  
    "join \n" +  
    "order_info oi\n" +  
    "on ri.order_id = oi.id\n" +  
    "join \n" +  
    "base_dic for system_time as of ri.proc_time as type_dic\n" +  
    "on ri.refund_type = type_dic.dic_code\n" +  
    "join\n" +
```

```

        "base_dic for system_time as of ri.proc_time as reason_dic\n" +
        "on ri.refund_reason_type=reason_dic.dic_code");
tableEnv.createTemporaryView("result_table", resultTable);

// TODO 8. 建立 Upsert-Kafka dwd_trade_order_refund 表
tableEnv.executeSql("create table dwd_trade_order_refund(\n" +
    "id string,\n" +
    "user_id string,\n" +
    "order_id string,\n" +
    "sku_id string,\n" +
    "province_id string,\n" +
    "date_id string,\n" +
    "create_time string,\n" +
    "refund_type_code string,\n" +
    "refund_type_name string,\n" +
    "refund_reason_type_code string,\n" +
    "refund_reason_type_name string,\n" +
    "refund_reason_txt string,\n" +
    "refund_num string,\n" +
    "refund_amount string,\n" +
    "ts string,\n" +
    "row_op_ts timestamp_ltz(3),\n" +
    "primary key(id) not enforced\n" +
    ") " + MyKafkaUtil.getUpsertKafkaDDL("dwd_trade_order_refund"));

// TODO 9. 将关联结果写入 Upsert-Kafka 表
tableEnv.executeSql("" +
    "insert into dwd_trade_order_refund select * from result_table");
}
}

```

➤ 在 MyKafkaUtil 中添加获取创建 topic_db 业务表 SQL 的方法

```

/**
 * Kafka-Source 获取 TopicDb 的 SQL 语句
 * @param groupId 消费者组
 * @return 创建 topic_db 表的建表语句
 */
public static String getTopicDbDDL(String groupId) {

    return "create table topic_db(" +
        "`database` String,\n" +
        "`table` String,\n" +
        "`type` String,\n" +
        "`data` map<String, String>,\n" +
        "`old` map<String, String>,\n" +
        "`proc_time` as PROCTIME(),\n" +
        "`ts` string\n" +
        ") with ('connector' = 'kafka', " +
        " 'topic' = 'topic_db'," +
        " 'properties.bootstrap.servers' = '" + BOOTSTRAP_SERVERS + "', " +
        " 'properties.group.id' = '" + groupId + "', " +
        " 'format' = 'json'," +
        " 'scan.startup.mode' = 'group-offsets')";
}

```

9.5 测试

- 启动 zk、kafka、maxwell
- 运行 DwdTradeOrderRefund
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_trade_order_refund 的输出

第 10 章 交易域退款成功事务事实表

10.1 主要任务

- (1) 从退款表中提取退款成功数据，并将字典表的 dic_name 维度退化到表中
- (2) 从订单表中提取退款成功订单数据
- (3) 从退单表中提取退款成功的明细数据

10.2 思路分析

(1) 设置 ttl

一次退款支付操作成功时，refund_payment 表会新增记录，订单表 order_info 和退单表 order_refund_info 的对应数据会发生修改，几张表之间不存在业务上的时间滞后。与字典表的关联分析同上，不再赘述。因而，仅考虑可能的数据乱序即可。将 ttl 设置为 5s。

(2) 建立 MySQL-Lookup 字典表

获取支付类型名称。

(3) 读取退款表数据，筛选退款成功数据

退款表 refund_payment 的粒度为一个订单中一个 SKU 的退款记录，与退款业务过

程的最细粒度相同，将其作为**主表**。

退款操作发生时，业务数据库的退款表会先插入一条数据，此时 `refund_status` 状态码应为 0701（商家审核中），`callback_time` 为 null，而后经历一系列业务过程：商家审核、买家发货、退单完成。退单完成时会将状态码由 0701 更新为 0705（退单完成），同时将 `callback_time` 更新为退款支付成功的回调时间。

由上述分析可知，退款成功记录应满足三个条件：

- 数据操作类型为 `update`；
- `refund_status` 为 0705；
- 修改的字段包含 `refund_status`。

注：本项目生成模拟数据的退款表中并没有上述分析提到的完整业务流程中多业务过程的变化，仅在退款操作完成时插入一条数据，此时的 `refund_status` 为 0701，`callback_time` 不为 null，因此要想观测到查询效果，**查询条件应做修改**：

- 注释对操作类型的判断；
- `where` 子句中 `refund_status` 的值应为 0701；
- 该条件的校验是通过判断 `old['refund_status']` 的值是否为 null 来实现的，模拟数据没有修改，`old['refund_status']` 必为 null，需要注释该条件。

模拟数据生成的问题并不影响我们对业务逻辑的分析，无须在意。

(4) 读取订单表数据，过滤退款成功订单数据

用于获取 `user_id` 和 `province_id`。退款操作完后时，订单表的 `order_status` 字段会更新为 1006（退款完成），因此退单成功对应的订单数据应满足三个条件：

- 操作类型为 `update`；
- `order_status` 为 1006；

- 修改了 order_status 字段。

order_status 值更改为 1006 之后对应的订单表数据就不会再发生变化，所以只要满足前两个条件，第三个条件必定满足。

(5) 筛选退款成功的退单明细数据

用于获取退单件数 refund_num。退单成功时 order_refund_info 表中的 refund_status 字段会修改为 0705（退款成功状态码）。因此筛选条件有三：

- 操作类型为 update；
- refund_status 为 0705；
- 修改了 refund_status 字段。

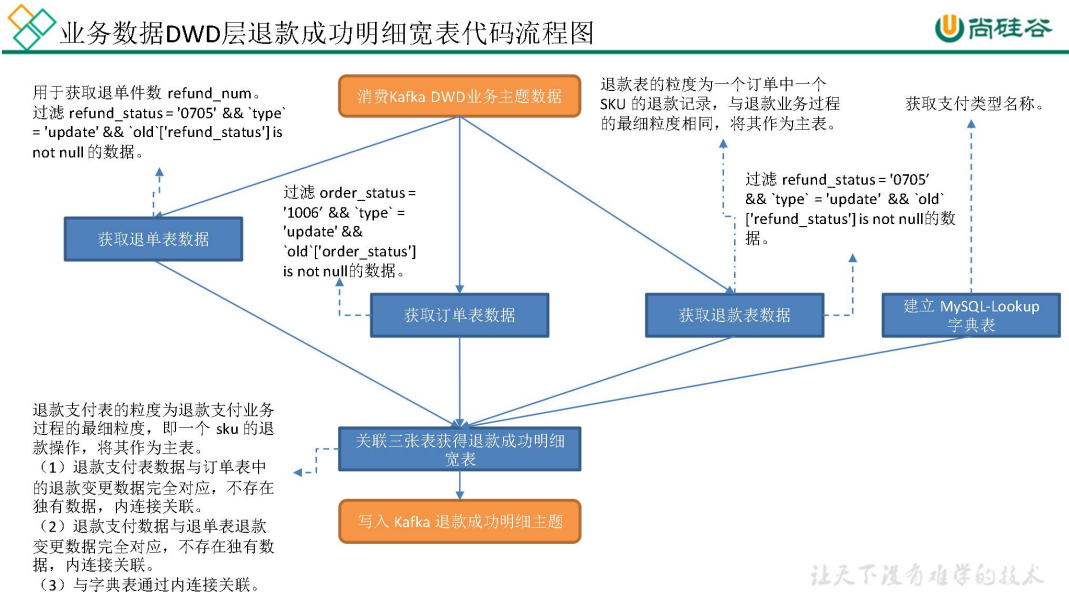
筛选方式同上。

(6) 关联四张表并写出到 Kafka 退款成功主题

退款支付表的粒度为退款支付业务过程的最细粒度，即一个 sku 的退款操作，将其作为主表。

- 退款支付表数据与订单表中的退款变更数据完全对应，不存在独有数据，内连接关联。
- 退款支付数据与退单表退款变更数据完全对应，不存在独有数据，内连接关联。
- 与字典表通过内连接关联。

10.3 图解



10.4 代码

```
package com.atguigu.gmall.realtime.app.dwd.db;

import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import com.atguigu.gmall.realtime.util.MySqlUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/**
 * Author: Felix
 * Date: 2022/5/12
 * Desc: 交易域退款成功事务事实表
 */
public class DwdTradeRefundPaySuc {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 从 Kafka 读取 topic_db 数据, 封装为 Flink SQL 表
        tableEnv.executeSql(MyKafkaUtil.getTopicDbDDL("dwd_trade_order_refund"));

        // TODO 4. 建立 MySQL-Lookup 字典表
        tableEnv.executeSql(MySqlUtil.getBaseDicLookupDDL());
    }
}
```

// TODO 5. 读取退款表数据, 并筛选退款成功数据

```
Table refundPayment = tableEnv.sqlQuery("select\n" +  
    "data['id'] id,\n" +  
    "data['order_id'] order_id,\n" +  
    "data['sku_id'] sku_id,\n" +  
    "data['payment_type'] payment_type,\n" +  
    "data['callback_time'] callback_time,\n" +  
    "data['total_amount'] total_amount,\n" +  
    "proc_time,\n" +  
    "ts\n" +  
    "from topic_db\n" +  
    "where `table` = 'refund_payment'\n" +  
    //"and `type` = 'update'\n" +  
    "and data['refund_status'] = '0701'\n" +  
    //+  
    //"and `old`['refund_status'] is not null"\n  
);  
  
tableEnv.createTemporaryView("refund_payment", refundPayment);
```

// TODO 6. 读取订单表数据并过滤退款成功订单数据

```
Table orderInfo = tableEnv.sqlQuery("select\n" +  
    "data['id'] id,\n" +  
    "data['user_id'] user_id,\n" +  
    "data['province_id'] province_id,\n" +  
    "`old`\n" +  
    "from topic_db\n" +  
    "where `table` = 'order_info'\n" +  
    "and `type` = 'update'\n" +  
    +  
    "and data['order_status']='1006'\n" +  
    "and `old`['order_status'] is not null"\n  
);  
  
tableEnv.createTemporaryView("order_info", orderInfo);
```

// TODO 7. 读取退单表数据并过滤退款成功数据

```
Table orderRefundInfo = tableEnv.sqlQuery("select\n" +  
    "data['order_id'] order_id,\n" +  
    "data['sku_id'] sku_id,\n" +  
    "data['refund_num'] refund_num,\n" +  
    "`old`\n" +  
    "from topic_db\n" +  
    "where `table` = 'order_refund_info'\n" +  
    +  
    "and `type` = 'update'\n" +  
    "and data['refund_status']='0705'\n" +  
    "and `old`['refund_status'] is not null"\n  
);  
  
tableEnv.createTemporaryView("order_refund_info", orderRefundInfo);
```

// TODO 8. 关联四张表获得退款成功表

```
Table resultTable = tableEnv.sqlQuery("select\n" +  
    "rp.id,\n" +  
    "oi.user_id,\n" +  
    "rp.order_id,\n" +  
    "rp.sku_id,\n" +  
    "oi.province_id,\n" +
```

```

        "rp.payment_type,\n" +
        "dic.dic_name payment_type_name,\n" +
        "date_format(rp.callback_time,'yyyy-MM-dd') date_id,\n" +
        "rp.callback_time,\n" +
        "ri.refund_num,\n" +
        "rp.total_amount,\n" +
        "rp.ts,\n" +
        "current_row_timestamp() row_op_ts\n" +
        "from refund_payment rp\n" +
        "join\n" +
        "order_info oi\n" +
        "on rp.order_id = oi.id\n" +
        "join\n" +
        "order_refund_info ri\n" +
        "on rp.order_id = ri.order_id\n" +
        "and rp.sku_id = ri.sku_id\n" +
        "join\n" +
        "base_dic for system_time as of rp.proc_time as dic\n" +
        "on rp.payment_type = dic.dic_code\n");
tableEnv.createTemporaryView("result_table", resultTable);

// TODO 9. 创建 Upsert-Kafka dwd_trade_refund_pay_suc 表
tableEnv.executeSql("create table dwd_trade_refund_pay_suc(\n" +
    "id string,\n" +
    "user_id string,\n" +
    "order_id string,\n" +
    "sku_id string,\n" +
    "province_id string,\n" +
    "payment_type_code string,\n" +
    "payment_type_name string,\n" +
    "date_id string,\n" +
    "callback_time string,\n" +
    "refund_num string,\n" +
    "refund_amount string,\n" +
    "ts string,\n" +
    "row_op_ts timestamp_ltz(3),\n" +
    "primary key(id) not enforced\n" +
    ")") + MyKafkaUtil.getUpsertKafkaDDL("dwd_trade_refund_pay_suc"));

// TODO 10. 将关联结果写入 Upsert-Kafka 表
tableEnv.executeSql("" +
    "insert into dwd_trade_refund_pay_suc select * from result_table");
}
}

```

10.5 测试

- 启动 zk、kafka、maxwell
- 运行 DwdTradeRefundPaySuc
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_trade_refund_pay_suc 的输出

第 11 章工具域优惠券领取事务事实表(练习)


11.1 主要任务

读取优惠券领用数据，写入 Kafka 优惠券领用主题

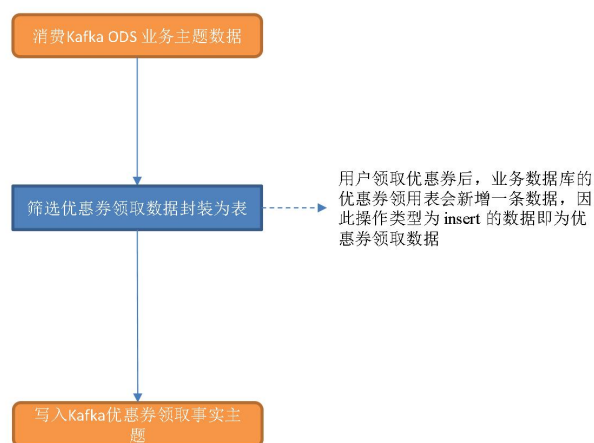
11.2 思路分析

用户领取优惠券后，业务数据库的优惠券领用表会新增一条数据，因此操作类型为 insert 的数据即为优惠券领取数据。

11.3 图解

 业务数据DWD层优惠券领取代码流程图

 尚硅谷



让天下没有难学的技术

11.4 代码

```
package com.atguigu.gmall.realtime.app.dwd.db;

import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
```

```

import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/**
 * Author: Felix
 * Date: 2022/5/12
 * Desc:工具域优惠券领取事务事实表
 */
public class DwdToolCouponGet {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 从 Kafka 读取 topic_db 数据, 封装为 Flink SQL 表
        tableEnv.executeSql(MyKafkaUtil.getTopicDbDDL("dwd_tool_coupon_get"));

        // TODO 4. 读取优惠券领用数据, 封装为表
        Table resultTable = tableEnv.sqlQuery("select\n" +
            "data['id'],\n" +
            "data['coupon_id'],\n" +
            "data['user_id'],\n" +
            "date_format(data['get_time'],'yyyy-MM-dd') date_id,\n" +
            "data['get_time'],\n" +
            "ts\n" +
            "from topic_db\n" +
            "where `table` = 'coupon_use'\n" +
            "and `type` = 'insert'\n");
        tableEnv.createTemporaryView("result_table", resultTable);

        // TODO 5. 建立 Upsert-Kafka dwd_tool_coupon_get 表
        tableEnv.executeSql("create table dwd_tool_coupon_get (\n" +
            "id string,\n" +
            "coupon_id string,\n" +
            "user_id string,\n" +
            "date_id string,\n" +
            "get_time string,\n" +
            "ts string,\n" +
            "primary key(id) not enforced\n" +
            ") " + MyKafkaUtil.getUpsertKafkaDDL("dwd_tool_coupon_get"));

        // TODO 6. 将数据写入 Upsert-Kafka 表
        tableEnv.executeSql("" +
            "insert into dwd_tool_coupon_get select * from result_table");

    }
}

```

11.5 测试

- 启动 zk、kafka、maxwell

- 运行 DwdToolCouponGet
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_tool_coupon_get 的输出

第 12 章工具域优惠券使用（下单）事务事实表(练习)

12.1 主要任务

读取优惠券领用表数据，筛选优惠券下单数据，写入 Kafka 优惠券下单主题。

12.2 思路分析

用户使用优惠券下单时，优惠券领用表的 using_time 字段会更新为下单时间，同时 coupon_status 字段会由 1401 更改为 1402，因此优惠券下单数据应满足三个条件：

- 操作类型为 update；
- 当前 coupon_status 字段的值为 1402；
- 修改了 coupon_status 字段。

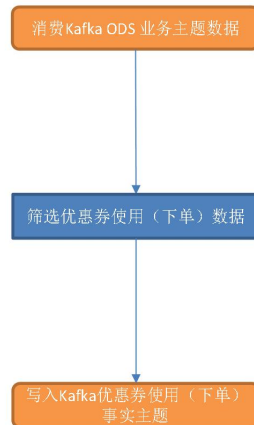
执行步骤

- 通过三个条件筛选优惠券使用（下单）数据。
- 建立 Upsert-Kafka 表，将优惠券使用（下单）数据写入 Kafka 优惠券使用（下单）事实主题。

12.3 图解



业务数据DWD层优惠券使用（下单）代码流程图



用户使用优惠券下单时，优惠券领用表的 `using_time` 字段会更新为下单时间，同时 `coupon_status` 字段会由 1401 更改为 1402，因此优惠券下单数据应满足三个条件：① 操作类型为 `update`；② 当前 `coupon_status` 字段的值为 1402；③ 修改了 `coupon_status` 字段。

让天下没有难学的技术

12.4 代码

```
package com.atguigu.gmall.realtime.app.dwd.db;

import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/**
 * Author: Felix
 * Date: 2022/5/12
 * Desc: 工具域优惠券使用（下单）事实事实表
 */
public class DwdToolCouponOrder {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 从 Kafka 读取 topic_db 数据，封装为 Flink SQL 表
        tableEnv.executeSql(MyKafkaUtil.getTopicDbDDL("dwd_tool_coupon_order"));

        // TODO 4. 读取优惠券领用表数据，筛选满足条件的优惠券下单数据
        Table couponUseOrder = tableEnv.sqlQuery("select\n" +
```



```

        "data['id'] id,\n" +
        "data['coupon_id'] coupon_id,\n" +
        "data['user_id'] user_id,\n" +
        "data['order_id'] order_id,\n" +
        "date_format(data['using_time'],'yyyy-MM-dd') date_id,\n" +
        "data['using_time'] using_time,\n" +
        "ts\n" +
        "from topic_db\n" +
        "where `table` = 'coupon_use'\n" +
        "and `type` = 'update'\n" +
        "and data['coupon_status'] = '1402'\n" +
        "and `old`['coupon_status'] = '1401'");

tableEnv.createTemporaryView("result_table", couponUseOrder);

// TODO 5. 建立 Upsert-Kafka dwd_tool_coupon_order 表
tableEnv.executeSql("create table dwd_tool_coupon_order(\n" +
    "id string,\n" +
    "coupon_id string,\n" +
    "user_id string,\n" +
    "order_id string,\n" +
    "date_id string,\n" +
    "order_time string,\n" +
    "ts string,\n" +
    "primary key(id) not enforced\n" +
    ")") + MyKafkaUtil.getUpsertKafkaDDL("dwd_tool_coupon_order"));

// TODO 6. 将数据写入 Upsert-Kafka 表
tableEnv.executeSql(" " +
    "insert into dwd_tool_coupon_order select " +
    "id,\n" +
    "coupon_id,\n" +
    "user_id,\n" +
    "order_id,\n" +
    "date_id,\n" +
    "using_time order_time,\n" +
    "ts from result_table");

    }
}

```

12.5 测试

- 启动 zk、kafka、maxwell
- 运行 DwdToolCouponOrder
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_tool_coupon_order 的输出

第 13 章工具域优惠券使用(支付)事务事实表(练习)

13.1 主要任务

读取优惠券领用表数据，筛选优惠券支付数据，写入 Kafka 优惠券支付主题。

13.2 思路分析

用户使用优惠券支付时，优惠券领用表的 `used_time` 字段会更新为支付时间，因此优惠券支付数据应满足两个条件：

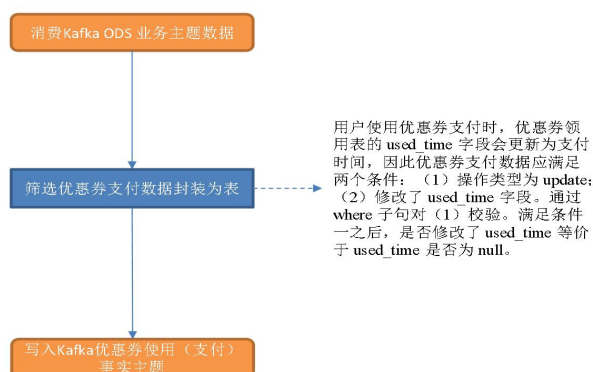
- 操作类型为 `update`；
- 修改了 `used_time` 字段。

使用优惠券支付后，优惠券领用表数据就不会再发生变化，所以在操作类型为 `update` 的前提下，只要 `used_time` 不为 `null`，就可以断定本次操作修改的是 `used_time` 字段。

13.3 图解

 业务数据DWD层优惠券使用（支付）代码流程图

 尚硅谷



让天下没有难学的技术

13.4 代码

```
package com.atguigu.gmall.realtime.app.dwd.db;

import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/**
 * Author: Felix
 * Date: 2022/5/12
 * Desc:工具域优惠券使用(支付)事务事实表
 */
public class DwdToolCouponPay {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 从 Kafka 读取 topic_db 数据, 封装为 Flink SQL 表
        tableEnv.executeSql(MyKafkaUtil.getTopicDbDDL("dwd_tool_coupon_pay"));

        // TODO 4. 读取优惠券领用表数据, 筛选优惠券使用(支付)数据
        Table couponUsePay = tableEnv.sqlQuery("select\n" +
            "data['id'] id,\n" +
            "data['coupon_id'] coupon_id,\n" +
            "data['user_id'] user_id,\n" +
            "data['order_id'] order_id,\n" +
            "date_format(data['used_time'],'yyyy-MM-dd') date_id,\n" +
            "data['used_time'] used_time,\n" +
            "`old`,\n" +
            "ts\n" +
            "from topic_db\n" +
            "where `table` = 'coupon_use'\n" +
            "and `type` = 'update'\n" +
            "and data['used_time'] is not null");

        tableEnv.createTemporaryView("coupon_use_pay", couponUsePay);

        // TODO 5. 建立 Upsert-Kafka dwd_tool_coupon_order 表
        tableEnv.executeSql("create table dwd_tool_coupon_pay(\n" +
            "id string,\n" +
            "coupon_id string,\n" +
            "user_id string,\n" +
            "order_id string,\n" +
            "date_id string,\n" +
            "payment_time string,\n" +
            "ts string,\n" +
            "primary key(id) not enforced\n" +
            ") " + MyKafkaUtil.getUpsertKafkaDDL("dwd_tool_coupon_pay"));

        // TODO 6. 将数据写入 Upsert-Kafka 表
```

```
tableEnv.executeSql("'" +
    "insert into dwd_tool_coupon_pay select " +
    "id,\n" +
    "coupon_id,\n" +
    "user_id,\n" +
    "order_id,\n" +
    "date_id,\n" +
    "used_time payment_time,\n" +
    "ts from coupon_use_pay");
}
```

13.5 测试

- 启动 zk、kafka、maxwell
- 运行 DwdToolCouponPay
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_tool_coupon_pay 的输出

第 14 章 互动域收藏商品事务事实表(练习)

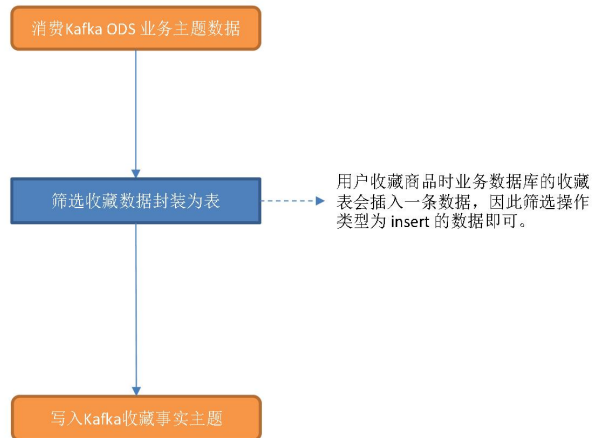
14.1 主要任务

读取收藏数据，写入 Kafka 收藏主题

14.2 思路分析

用户收藏商品时业务数据库的收藏表会插入一条数据，因此筛选操作类型为 insert 的数据即可。

14.3 图解



让天下没有难学的技术

14.4 代码

```
package com.atguigu.gmall.realtime.app.dwd.db;

import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/**
 * Author: Felix
 * Date: 2022/5/12
 * Desc: 互动域收藏商品事务事实表
 */
public class DwdInteractionFavorAdd {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 从 Kafka 读取 topic_db 数据, 封装为 Flink SQL 表
        tableEnv.executeSql(MyKafkaUtil.getTopicDbDDL("dwd_interaction_favor_add"));

        // TODO 4. 读取收藏表数据
        Table favorInfo = tableEnv.sqlQuery("select\n" +
            "data['id'] id,\n" +
```

```

        "data['user_id'] user_id,\n" +
        "data['sku_id'] sku_id,\n" +
        "date_format(data['create_time'],'yyyy-MM-dd') date_id,\n" +
        "data['create_time'] create_time,\n" +
        "ts\n" +
        "from topic_db\n" +
        "where `table` = 'favor_info'\n" +
        "and `type` = 'insert'\n");
tableEnv.createTemporaryView("favor_info", favorInfo);

// TODO 5. 创建 Upsert-Kafka dwd_interaction_favor_add 表
tableEnv.executeSql("create table dwd_interaction_favor_add (\n" +
    "id string,\n" +
    "user_id string,\n" +
    "sku_id string,\n" +
    "date_id string,\n" +
    "create_time string,\n" +
    "ts string,\n" +
    "primary key(id) not enforced\n" +
    ") " + MyKafkaUtil.getUpsertKafkaDDL("dwd_interaction_favor_add"));

// TODO 6. 将数据写入 Upsert-Kafka 表
tableEnv.executeSql("" +
    "insert into dwd_interaction_favor_add select * from favor_info");
    }
}

```

14.5 测试

- 启动 zk、kafka、maxwell
- 运行 DwdInteractionFavorAdd
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_interaction_favor_add 的输出

第 15 章 互动域评价事务事实表(练习)

15.1 主要任务

建立 MySQL-Lookup 字典表，读取评论表数据，关联字典表以获取评价（好评、中评、差评、自动），将结果写入 Kafka 评价主题。

15.2 思路分析

(1) 设置 ttl

前文提到，与字典表关联时 ttl 的设置主要是考虑到从外部介质查询维度数据的时间，此处设置为 5s。

(2) 筛选评论数据

用户提交评论时评价表会插入一条数据，筛选操作类型为 insert 的数据即可。

(3) 建立 Mysql-Lookup 字典表

(4) 关联两张表

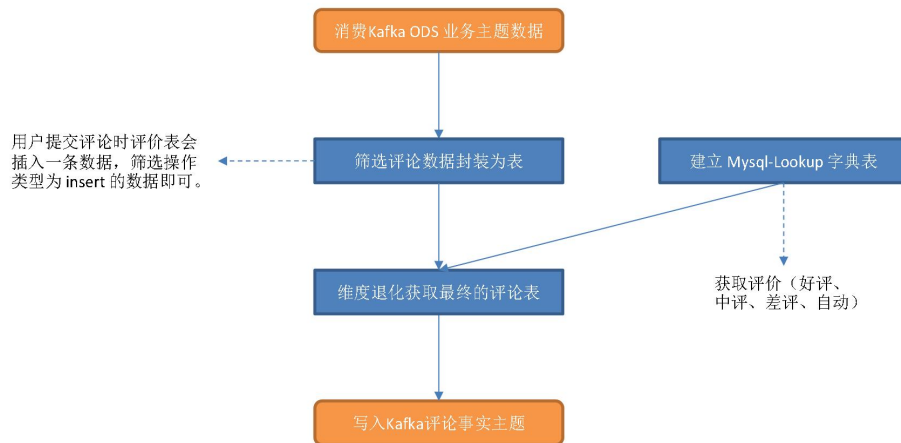
(5) 写入 Kafka 互动域评论事实主题

15.3 图解



业务数据DWD层评论代码流程图

尚硅谷



让天下没有难学的技术

15.4 代码

```
package com.atguigu.gmall.realtime.app.dwd.db;

import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import com.atguigu.gmall.realtime.util.MySqlUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
```

```

import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/**
 * Author: Felix
 * Date: 2022/5/12
 * Desc: 互动域评价事务事实表
 */
public class DwdInteractionComment {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 从 Kafka 读取 topic_db 数据, 封装为 Flink SQL 表
        tableEnv.executeSql(MyKafkaUtil.getTopicDbDDL("dwd_interaction_comment"));

        // TODO 4. 读取评论表数据
        Table commentInfo = tableEnv.sqlQuery("select\n" +
            "data['id'] id,\n" +
            "data['user_id'] user_id,\n" +
            "data['sku_id'] sku_id,\n" +
            "data['order_id'] order_id,\n" +
            "data['create_time'] create_time,\n" +
            "data['appraise'] appraise,\n" +
            "proc_time,\n" +
            "ts\n" +
            "from topic_db\n" +
            "where `table` = 'comment_info'\n" +
            "and `type` = 'insert'\n");
        tableEnv.createTemporaryView("comment_info", commentInfo);

        // TODO 5. 建立 MySQL-LookUp 字典表
        tableEnv.executeSql(MySqlUtil.getBaseDicLookUpDDL());

        // TODO 6. 关联两张表
        Table resultTable = tableEnv.sqlQuery("select\n" +
            "ci.id,\n" +
            "ci.user_id,\n" +
            "ci.sku_id,\n" +
            "ci.order_id,\n" +
            "date_format(ci.create_time,'yyyy-MM-dd') date_id,\n" +
            "ci.create_time,\n" +
            "ci.appraise,\n" +
            "dic.dic_name,\n" +
            "ts\n" +
            "from comment_info ci\n" +
            "join\n" +
            "base_dic for system_time as of ci.proc_time as dic\n" +
            "on ci.appraise = dic.dic_code");
        tableEnv.createTemporaryView("result_table", resultTable);

        // TODO 7. 建立 Upsert-Kafka dwd_interaction_comment 表
        tableEnv.executeSql("create table dwd_interaction_comment(\n" +

```



```

        "id string,\n" +
        "user_id string,\n" +
        "sku_id string,\n" +
        "order_id string,\n" +
        "date_id string,\n" +
        "create_time string,\n" +
        "appraise_code string,\n" +
        "appraise_name string,\n" +
        "ts string,\n" +
        "primary key(id) not enforced\n" +
        ") " + MyKafkaUtil.getUpsertKafkaDDL("dwd_interaction_comment"));

    // TODO 8. 将关联结果写入 Upsert-Kafka 表
    tableEnv.executeSql("" +
        "insert into dwd_interaction_comment select * from result_table");
}
}

```

15.5 测试

- 启动 zk、kafka、maxwell
- 运行 DwdInteractionComment
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_interaction_comment 的输出

第 16 章 用户域用户注册事务事实表(练习)

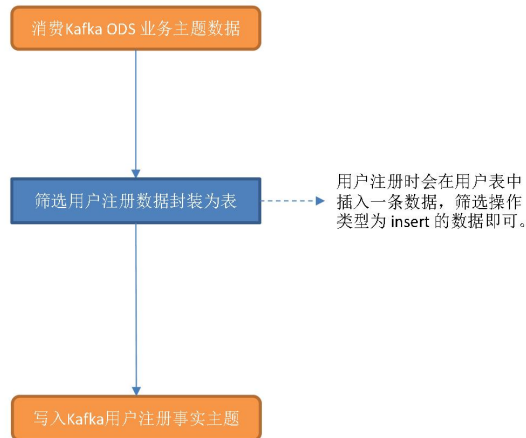
16.1 主要任务

读取用户表数据，获取注册时间，将用户注册信息写入 Kafka 用户注册主题。

16.2 思路分析

用户注册时会在用户表中插入一条数据，筛选操作类型为 insert 的数据即可。

16.3 图解



让天下没有难学的技术

16.4 代码

```
package com.atguigu.gmall.realtime.app.dwd.db;

import com.atguigu.gmall.realtime.util.MyKafkaUtil;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

import java.time.ZoneId;

/**
 * Author: Felix
 * Date: 2022/5/12
 * Desc: 用户域用户注册事务事实表
 */
public class DwdUserRegister {
    public static void main(String[] args) throws Exception {

        // TODO 1. 环境准备
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(4);
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);
        tableEnv.getConfig().setLocalTimeZone(ZoneId.of("GMT+8"));

        // TODO 2. 检查点相关设置(略)

        // TODO 3. 从 Kafka 读取 topic_db 数据, 封装为 Flink SQL 表
        tableEnv.executeSql(MyKafkaUtil.getTopicDbDDL("dwd_user_register_group"));
```

```

// TODO 4. 读取用户表数据
Table userInfo = tableEnv.sqlQuery("select\n" +
    "data['id'] user_id,\n" +
    "data['create_time'] create_time,\n" +
    "ts\n" +
    "from topic_db\n" +
    "where `table` = 'user_info'\n" +
    "and `type` = 'insert'\n");
tableEnv.createTemporaryView("user_info", userInfo);

// TODO 5. 创建 Upsert-Kafka dwd_user_register 表
tableEnv.executeSql("create table `dwd_user_register`(\n" +
    "`user_id` string,\n" +
    "`date_id` string,\n" +
    "`create_time` string,\n" +
    "`ts` string,\n" +
    "primary key(`user_id`) not enforced\n" +
    ") " + MyKafkaUtil.getUpsertKafkaDDL("dwd_user_register"));

// TODO 6. 将输入写入 Upsert-Kafka 表
tableEnv.executeSql("insert into dwd_user_register\n" +
    "select \n" +
    "user_id,\n" +
    "date_format(create_time, 'yyyy-MM-dd') date_id,\n" +
    "create_time,\n" +
    "ts\n" +
    "from user_info");
}
}

```

16.5 测试

- 启动 zk、kafka、maxwell
- 运行 DwdUserRegister
- 修改生成业务数据的配置文件，确保每次运行的时候有新的用户生成
- 运行生成业务数据的 jar
- 创建消费者对象查看 kafka 主题 dwd_user_register 的输出