## Flink-Action综合练习

## 课程目标

* 掌握使用Flink实现模拟双十一实时大屏统计
* 掌握使用Flink实现订单自动好评

## Flink模拟双十一实时大屏统计

### 需求



在大数据的实时处理中，实时的大屏展示已经成了一个很重要的展示项，比如最有名的双十一大屏实时销售总价展示。除了这个，还有一些其他场景的应用，比如我们在我们的后台系统实时的展示我们网站当前的pv、uv等等，其实做法都是类似的。

今天我们就做一个最简单的模拟电商统计大屏的小例子，

需求如下：

1.实时计算出当天零点截止到当前时间的销售总额

2.计算出各个分类的销售top3

3.每秒钟更新一次统计结果

### 数据

首先我们通过自定义source 模拟订单的生成，生成了一个Tuple2,第一个元素是分类，第二个元素表示这个分类下产生的订单金额，金额我们通过随机生成.

|  |
| --- |
| */\*\*  \* 自定义数据源实时产生订单数据Tuple2<分类, 金额>  \*/* **public static class** MySource **implements** SourceFunction<Tuple2<String, Double>>{  **private boolean flag** = **true**;  **private** String[] **categorys** = {**"女装"**, **"男装"**,**"图书"**, **"家电"**,**"洗护"**, **"美妆"**,**"运动"**, **"游戏"**,**"户外"**, **"家具"**,**"乐器"**, **"办公"**};  **private** Random **random** = **new** Random();   @Override  **public void** run(SourceContext<Tuple2<String, Double>> ctx) **throws** Exception {  **while** (**flag**){  *//随机生成分类和金额* **int** index = **random**.nextInt(**categorys**.**length**);*//[0~length) ==> [0~length-1]* String category = **categorys**[index];*//获取的随机分类* **double** price = **random**.nextDouble() \* 100;*//注意nextDouble生成的是[0~1)之间的随机数,\*100之后表示[0~100)* ctx.collect(Tuple2.*of*(category,price));  Thread.*sleep*(20);  }  }   @Override  **public void** cancel() {  **flag** = **false**;  } } |

### 编码步骤:

1. env
2. source
3. transformation

3.1定义大小为一天的窗口,第二个参数表示中国使用的UTC+08:00时区比UTC时间早

.keyBy(0)

window(TumblingProcessingTimeWindows.of(Time.days(1), Time.hours(-8))

3.2定义一个1s的触发器

.trigger(ContinuousProcessingTimeTrigger.of(Time.seconds(1)))

3.3聚合结果.aggregate(new PriceAggregate(), new WindowResult());

3.4看一下聚合的结果

CategoryPojo(category=男装, totalPrice=17225.26, dateTime=2020-10-20 08:04:12)

1. 使用上面聚合的结果,实现业务需求:

result.keyBy("dateTime")

//每秒钟更新一次统计结果

.window(TumblingProcessingTimeWindows.of(Time.seconds(1)))

//在ProcessWindowFunction中实现该复杂业务逻辑

.process(new WindowResultProcess());

* 1. 实时计算出当天零点截止到当前时间的销售总额

4.2.计算出各个分类的销售top3

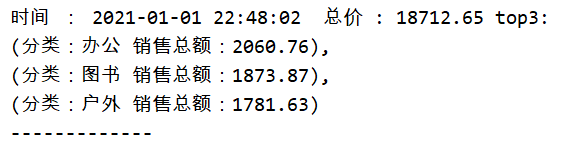
4.3.每秒钟更新一次统计结果

1. execute

### 代码实现

|  |
| --- |
| **package** com.nhbd.flink.ch4.action;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.commons.lang3.StringUtils; **import** org.apache.flink.api.common.functions.AggregateFunction; **import** org.apache.flink.api.java.tuple.Tuple; **import** org.apache.flink.api.java.tuple.Tuple1; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.source.SourceFunction; **import** org.apache.flink.streaming.api.functions.windowing.ProcessWindowFunction; **import** org.apache.flink.streaming.api.functions.windowing.WindowFunction; **import** org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows; **import** org.apache.flink.streaming.api.windowing.time.Time; **import** org.apache.flink.streaming.api.windowing.triggers.ContinuousProcessingTimeTrigger; **import** org.apache.flink.streaming.api.windowing.windows.TimeWindow; **import** org.apache.flink.util.Collector;  **import** java.math.BigDecimal; **import** java.text.SimpleDateFormat; **import** java.util.\*; **import** java.util.stream.Collectors;  */\*\*  \*   \* Desc  \* 模拟双11商品实时交易大屏统计分析  \*/* **public class** DoubleElevenBigScreem {  **public static void** main(String[] args) **throws** Exception{  *//编码步骤:  //1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);*//学习测试方便观察   //2.source  //模拟实时订单信息* DataStreamSource<Tuple2<String, Double>> sourceDS = env.addSource(**new** MySource());   */\*  注意:需求如下：  -1.实时计算出11月11日00:00:00零点开始截止到当前时间的销售总额  -2.计算出各个分类的销售额top3  -3.每1秒钟更新一次统计结果  如果使用之前学习的简单的timeWindow(Time size窗口大小, Time slide滑动间隔)来处理,  如xxx.timeWindow(24小时,1s),计算的是需求中的吗?  不是!如果使用之前的做法那么是完成不了需求的,因为:  如11月11日00:00:01计算的是11月10号[00:00:00~23:59:59s]的数据  而我们应该要计算的是:11月11日00:00:00~11月11日00:00:01  所以不能使用之前的简单做法!\*/   //3.transformation  //.keyBy(0)* SingleOutputStreamOperator<CategoryPojo> tempAggResult = sourceDS.keyBy(0)  *//3.1定义大小为一天的窗口,第二个参数表示中国使用的UTC+08:00时区比UTC时间早  /\*  of(Time 窗口大小, Time 带时间校准的从哪开始)源码中有解释:  如果您居住在不使用UTC±00：00时间的地方，例如使用UTC + 08：00的中国，并且您需要一个大小为一天的时间窗口，  并且窗口从当地时间的每00:00:00开始，您可以使用of(Time.days(1)，Time.hours(-8))  注意:该代码如果在11月11日运行就会从11月11日00:00:00开始记录直到11月11日23:59:59的1天的数据  注意:我们这里简化了没有把之前的Watermaker那些代码拿过来,所以直接ProcessingTime  \*/* .window(TumblingProcessingTimeWindows.*of*(Time.*days*(1), Time.*hours*(-8)))*//仅仅只定义了一个窗口大小  //3.2定义一个1s的触发器* .trigger(ContinuousProcessingTimeTrigger.*of*(Time.*seconds*(1)))  *//上面的3.1和3.2相当于自定义窗口的长度和触发时机  //3.3聚合结果.aggregate(new PriceAggregate(), new WindowResult());  //.sum(1)//以前的写法用的默认的聚合和收集  //现在可以自定义如何对price进行聚合,并自定义聚合结果用怎样的格式进行收集* .aggregate(**new** PriceAggregate(), **new** WindowResult());  *//3.4看一下初步聚合的结果* tempAggResult.print(**"初步聚合结果"**);  *//CategoryPojo(category=运动, totalPrice=118.69, dateTime=2020-10-20 08:04:12)  //上面的结果表示:当前各个分类的销售总额   /\*  注意:需求如下：  -1.实时计算出11月11日00:00:00零点开始截止到当前时间的销售总额  -2.计算出各个分类的销售额top3  -3.每1秒钟更新一次统计结果  \*/  //4.使用上面初步聚合的结果,实现业务需求,并sink* tempAggResult.keyBy(**"dateTime"**)*//按照时间分组是因为需要每1s更新截至到当前时间的销售总额  //每秒钟更新一次统计结果  //Time size 为1s,表示计算最近1s的数据* .window(TumblingProcessingTimeWindows.*of*(Time.*seconds*(1)))  *//在ProcessWindowFunction中实现该复杂业务逻辑,一次性将需求1和2搞定  //-1.实时计算出11月11日00:00:00零点开始截止到当前时间的销售总额  //-2.计算出各个分类的销售额top3  //-3.每1秒钟更新一次统计结果* .process(**new** WindowResultProcess());*//window后的process方法可以处理复杂逻辑    //5.execute* env.execute();  }   */\*\*  \* 自定义数据源实时产生订单数据Tuple2<分类, 金额>  \*/* **public static class** MySource **implements** SourceFunction<Tuple2<String, Double>>{  **private boolean flag** = **true**;  **private** String[] **categorys** = {**"女装"**, **"男装"**,**"图书"**, **"家电"**,**"洗护"**, **"美妆"**,**"运动"**, **"游戏"**,**"户外"**, **"家具"**,**"乐器"**, **"办公"**};  **private** Random **random** = **new** Random();   @Override  **public void** run(SourceContext<Tuple2<String, Double>> ctx) **throws** Exception {  **while** (**flag**){  *//随机生成分类和金额* **int** index = **random**.nextInt(**categorys**.**length**);*//[0~length) ==> [0~length-1]* String category = **categorys**[index];*//获取的随机分类* **double** price = **random**.nextDouble() \* 100;*//注意nextDouble生成的是[0~1)之间的随机数,\*100之后表示[0~100)* ctx.collect(Tuple2.*of*(category,price));  Thread.*sleep*(20);  }  }   @Override  **public void** cancel() {  **flag** = **false**;  }  }   */\*\*  \* 自定义价格聚合函数,其实就是对price的简单sum操作  \* AggregateFunction<IN, ACC, OUT>  \* AggregateFunction<Tuple2<String, Double>, Double, Double>  \*  \*/* **private static class** PriceAggregate **implements** AggregateFunction<Tuple2<String, Double>, Double, Double> {  *//初始化累加器为0* @Override  **public** Double createAccumulator() {  **return** 0D; *//D表示Double,L表示long* }   *//把price往累加器上累加* @Override  **public** Double add(Tuple2<String, Double> value, Double accumulator) {  **return** value.**f1** + accumulator;  }   *//获取累加结果* @Override  **public** Double getResult(Double accumulator) {  **return** accumulator;  }   *//各个subTask的结果合并* @Override  **public** Double merge(Double a, Double b) {  **return** a + b;  }  }   */\*\*  \* 用于存储聚合的结果  \*/* @Data  @AllArgsConstructor  @NoArgsConstructor  **public static class** CategoryPojo {  **private** String **category**;*//分类名称* **private double totalPrice**;*//该分类总销售额* **private** String **dateTime**;*// 截止到当前时间的时间,本来应该是EventTime,但是我们这里简化了直接用当前系统时间即可* }   */\*\*  \* 自定义WindowFunction,实现如何收集窗口结果数据  \* interface WindowFunction<IN, OUT, KEY, W extends Window>  \* interface WindowFunction<Double, CategoryPojo, Tuple的真实类型就是String就是分类, W extends Window>  \*/* **private static class** WindowResult **implements** WindowFunction<Double, CategoryPojo, Tuple, TimeWindow> {  *//定义一个时间格式化工具用来将当前时间(双十一那天订单的时间)转为String格式* SimpleDateFormat **df** = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**);  @Override  **public void** apply(Tuple tuple, TimeWindow window, Iterable<Double> input, Collector<CategoryPojo> out) **throws** Exception {  String category = ((Tuple1<String>) tuple).**f0**;   Double price = input.iterator().next();  *//为了后面项目铺垫,使用一下用Bigdecimal来表示精确的小数* BigDecimal bigDecimal = **new** BigDecimal(price);  *//setScale设置精度保留2位小数,* **double** roundPrice = bigDecimal.setScale(2, BigDecimal.***ROUND\_HALF\_UP***).doubleValue();*//四舍五入* **long** currentTimeMillis = System.*currentTimeMillis*();  String dateTime = **df**.format(currentTimeMillis);   CategoryPojo categoryPojo = **new** CategoryPojo(category, roundPrice, dateTime);  out.collect(categoryPojo);  }  }   */\*\*  \* 实现ProcessWindowFunction  \* abstract class ProcessWindowFunction<IN, OUT, KEY, W extends Window>  \* abstract class ProcessWindowFunction<CategoryPojo, Object, Tuple就是String类型的dateTime, TimeWindow extends Window>  \*  \* 把各个分类的总价加起来，就是全站的总销量金额，  \* 然后我们同时使用优先级队列计算出分类销售的Top3，  \* 最后打印出结果，在实际中我们可以把这个结果数据存储到hbase或者redis中，以供前端的实时页面展示。  \*/* **private static class** WindowResultProcess **extends** ProcessWindowFunction<CategoryPojo, Object, Tuple, TimeWindow> {  @Override  **public void** process(Tuple tuple, Context context, Iterable<CategoryPojo> elements, Collector<Object> out) **throws** Exception {  String dateTime = ((Tuple1<String>)tuple).**f0**;  *//Java中的大小顶堆可以使用优先级队列来实现  //https://blog.csdn.net/hefenglian/article/details/81807527  //注意:  // 小顶堆用来计算:最大的topN  // 大顶堆用来计算:最小的topN* Queue<CategoryPojo> queue = **new** PriorityQueue<>(3,*//初始容量  //正常的排序,就是小的在前,大的在后,也就是c1>c2的时候返回1,也就是小顶堆* (c1, c2) -> c1.getTotalPrice() >= c2.getTotalPrice() ? 1 : -1);   *//在这里我们要完成需求:  // \* -1.实时计算出11月11日00:00:00零点开始截止到当前时间的销售总额,其实就是把之前的初步聚合的price再累加!* **double** totalPrice = 0D;  **double** roundPrice = 0D;  Iterator<CategoryPojo> iterator = elements.iterator();  **for** (CategoryPojo element : elements) {  **double** price = element.**totalPrice**;*//某个分类的总销售额* totalPrice += price;  BigDecimal bigDecimal = **new** BigDecimal(totalPrice);  roundPrice = bigDecimal.setScale(2, BigDecimal.***ROUND\_HALF\_UP***).doubleValue();*//四舍五入  // \* -2.计算出各个分类的销售额top3,其实就是对各个分类的price进行排序取前3  //注意:我们只需要top3,也就是只关注最大的前3个的顺序,剩下不管!所以不要使用全局排序,只需要做最大的前3的局部排序即可  //那么可以使用小顶堆,把小的放顶上  // c:80  // b:90  // a:100  //那么来了一个数,和最顶上的比,如d,  //if(d>顶上),把顶上的去掉,把d放上去,再和b,a比较并排序,保证顶上是最小的  //if(d<=顶上),不用变* **if** (queue.size() < 3) {*//小顶堆size<3,说明数不够,直接放入* queue.add(element);  }**else**{*//小顶堆size=3,说明,小顶堆满了,进来一个需要比较  //"取出"顶上的(不是移除)* CategoryPojo top = queue.peek();  **if**(element.**totalPrice** > top.**totalPrice**){  *//queue.remove(top);//移除指定的元素* queue.poll();*//移除顶上的元素* queue.add(element);  }  }  }  *// \* -3.每1秒钟更新一次统计结果,可以直接打印/sink,也可以收集完结果返回后再打印,  // 但是我们这里一次性处理了需求1和2的两种结果,不好返回,所以直接输出!  //对queue中的数据逆序  //各个分类的销售额top3* List<String> top3Result = queue.stream()  .sorted((c1, c2) -> c1.getTotalPrice() > c2.getTotalPrice() ? -1 : 1)*//逆序* .map(c -> **"(分类："** + c.getCategory() + **" 销售总额："** + c.getTotalPrice() + **")"**)  .collect(Collectors.*toList*());  System.***out***.println(**"时间 ： "** + dateTime + **" 总价 : "** + roundPrice + **" top3:\n"** + StringUtils.*join*(top3Result, **",\n"**));  System.***out***.println(**"-------------"**);   }  } } |

### 效果



## Flink实现订单自动好评

### 需求





在电商领域会有这么一个场景，如果用户买了商品，在订单完成之后，一定时间之内没有做出评价，系统自动给与五星好评，我们今天主要使用Flink的定时器来简单实现这一功能。

### 数据

自定义source模拟生成一些订单数据.  
在这里，我们生了一个最简单的二元组Tuple3,包含用户id,订单id和订单完成时间三个字段.

|  |
| --- |
| */\*\*  \* 自定义source实时产生订单数据Tuple3<用户id,订单id, 订单生成时间>  \*/* **public static class** MySource **implements** SourceFunction<Tuple3<String, String, Long>> {  **private boolean flag** = **true**;  @Override  **public void** run(SourceContext<Tuple3<String, String, Long>> ctx) **throws** Exception {  Random random = **new** Random();  **while** (**flag**) {  String userId = random.nextInt(5) + **""**;  String orderId = UUID.*randomUUID*().toString();  **long** currentTimeMillis = System.*currentTimeMillis*();  ctx.collect(Tuple3.*of*(userId, orderId, currentTimeMillis));  Thread.*sleep*(500);  }  }   @Override  **public void** cancel() {  **flag** = **false**;  } } |

### 编码步骤

1. env
2. source
3. transformation

设置经过interval毫秒用户未对订单做出评价，自动给与好评.为了演示方便，设置5s的时间

long interval = 5000L;

分组后使用自定义KeyedProcessFunction完成定时判断超时订单并自动好评

dataStream.keyBy(0).process(new TimerProcessFuntion(interval));

3.1定义MapState类型的状态，key是订单号，value是订单完成时间

3.2创建MapState

MapStateDescriptor<String, Long> mapStateDesc =

new MapStateDescriptor<>("mapStateDesc", String.class, Long.class);

mapState = getRuntimeContext().getMapState(mapStateDesc);

3.3注册定时器

mapState.put(value.f0, value.f1);

ctx.timerService().registerProcessingTimeTimer(value.f1 + interval);

3.4定时器被触发时执行并输出结果

1. sink
2. execute

### 代码实现

|  |
| --- |
| **package** com.nhbd.flink.ch4.action;  **import** org.apache.flink.api.common.state.MapState; **import** org.apache.flink.api.common.state.MapStateDescriptor; **import** org.apache.flink.api.java.tuple.Tuple; **import** org.apache.flink.api.java.tuple.Tuple3; **import** org.apache.flink.configuration.Configuration; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.KeyedProcessFunction; **import** org.apache.flink.streaming.api.functions.source.SourceFunction; **import** org.apache.flink.util.Collector;  **import** java.util.Iterator; **import** java.util.Map; **import** java.util.Random; **import** java.util.UUID;  */\*\*  \*   \* Desc  \* 在电商领域会有这么一个场景，如果用户买了商品，在订单完成之后，一定时间之内没有做出评价，系统自动给与五星好评，  \* 我们今天主要使用Flink的定时器来简单实现这一功能。  \*/* **public class** OrderAutomaticFavorableComments {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  *//2.source* DataStreamSource<Tuple3<String, String, Long>> sourceDS = env.addSource(**new** MySource());  *//这里可以使用订单生成时间作为事件时间,代码和之前的一样  //这里不作为重点,所以简化处理!   //3.transformation  //设置经过interval用户未对订单做出评价，自动给与好评.为了演示方便，设置5000ms的时间* **long** interval = 5000L;  *//分组后使用自定义KeyedProcessFunction完成定时判断超时订单并自动好评* sourceDS.keyBy(0) *//实际中可以对用户id进行分组  //KeyedProcessFunction:进到窗口的数据是分好组的  //ProcessFunction:进到窗口的数据是不区分分组的* .process(**new** TimerProcessFuntion(interval));  *//4.execute* env.execute();  }   */\*\*  \* 自定义source实时产生订单数据Tuple2<订单id, 订单生成时间>  \*/* **public static class** MySource **implements** SourceFunction<Tuple3<String, String, Long>> {  **private boolean flag** = **true**;  @Override  **public void** run(SourceContext<Tuple3<String, String, Long>> ctx) **throws** Exception {  Random random = **new** Random();  **while** (**flag**) {  String userId = random.nextInt(5) + **""**;  String orderId = UUID.*randomUUID*().toString();  **long** currentTimeMillis = System.*currentTimeMillis*();  ctx.collect(Tuple3.*of*(userId, orderId, currentTimeMillis));  Thread.*sleep*(500);  }  }   @Override  **public void** cancel() {  **flag** = **false**;  }  }   */\*\*  \* 自定义处理函数用来给超时订单做自动好评!  \* 如一个订单进来:<订单id, 2020-10-10 12:00:00>  \* 那么该订单应该在12:00:00 + 5s 的时候超时!  \* 所以我们可以在订单进来的时候设置一个定时器,在订单时间 + interval的时候触发!  \* KeyedProcessFunction<K, I, O>  \* KeyedProcessFunction<Tuple就是String, Tuple3<用户id, 订单id, 订单生成时间>, Object>  \*/* **public static class** TimerProcessFuntion **extends** KeyedProcessFunction<Tuple, Tuple3<String, String, Long>, Object> {  **private long interval**;   **public** TimerProcessFuntion(**long** interval) {  **this**.**interval** = interval;*//传过来的是5000ms/5s* }   *//3.1定义MapState类型的状态，key是订单号，value是订单完成时间  //定义一个状态用来记录订单信息  //MapState<订单id, 订单完成时间>* **private** MapState<String, Long> **mapState**;   *//3.2初始化MapState* @Override  **public void** open(Configuration parameters) **throws** Exception {  *//创建状态描述器* MapStateDescriptor<String, Long> mapStateDesc = **new** MapStateDescriptor<>(**"mapState"**, String.**class**, Long.**class**);  *//根据状态描述器初始化状态* **mapState** = getRuntimeContext().getMapState(mapStateDesc);  }    *//3.3注册定时器  //处理每一个订单并设置定时器* @Override  **public void** processElement(Tuple3<String, String, Long> value, Context ctx, Collector<Object> out) **throws** Exception {  **mapState**.put(value.**f1**, value.**f2**);  *//如一个订单进来:<订单id, 2020-10-10 12:00:00>  //那么该订单应该在12:00:00 + 5s 的时候超时!  //在订单进来的时候设置一个定时器,在订单时间 + interval的时候触发!!!* ctx.timerService().registerProcessingTimeTimer(value.**f2** + **interval**);  }   *//3.4定时器被触发时执行并输出结果并sink* @Override  **public void** onTimer(**long** timestamp, OnTimerContext ctx, Collector<Object> out) **throws** Exception {  *//能够执行到这里说明订单超时了!超时了得去看看订单是否评价了(实际中应该要调用外部接口/方法查订单系统!,我们这里没有,所以模拟一下)  //没有评价才给默认好评!并直接输出提示!  //已经评价了,直接输出提示!* Iterator<Map.Entry<String, Long>> iterator = **mapState**.iterator();  **while** (iterator.hasNext()) {  Map.Entry<String, Long> entry = iterator.next();  String orderId = entry.getKey();  *//调用订单系统查询是否已经评价* **boolean** result = isEvaluation(orderId);  **if** (result) {*//已评价* System.***out***.println(**"订单(orderid: "** + orderId + **")在"** + **interval** + **"毫秒时间内已经评价，不做处理"**);  } **else** {*//未评价* System.***out***.println(**"订单(orderid: "** + orderId + **")在"** + **interval** + **"毫秒时间内未评价，系统自动给了默认好评!"**);  *//实际中还需要调用订单系统将该订单orderId设置为5星好评!* }  *//从状态中移除已经处理过的订单,避免重复处理* iterator.remove();  }  }   *//在生产环境下，可以去查询相关的订单系统.* **private boolean** isEvaluation(String key) {  **return** key.hashCode() % 2 == 0;*//随机返回订单是否已评价* }  } } |

### 效果

