

# Flink Time和Watermark的理解

[jianshu.com](http://jianshu.com) 已更新2020年1月1日

## Flink Time和Watermark的理解

### 1. Time

#### 背景

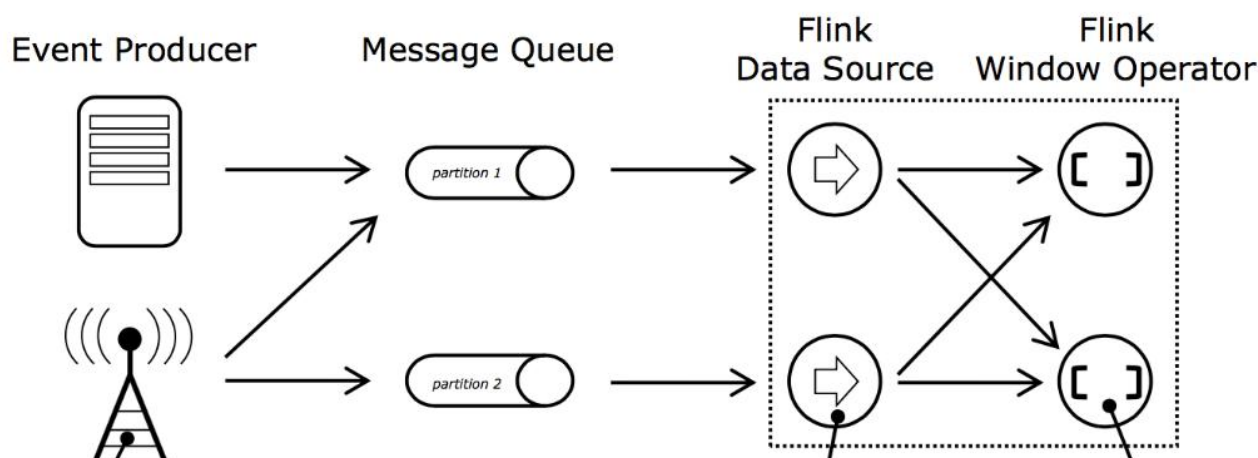
在实际开发过程中，我们可能需要接入各种流数据源，比如在线业务用户点击流数据、监控系统实时收集到的事件流数据、从传感器采集到的实时数据，等等，为了处理方便他们可能会写入Kafka消息中间件集群中某个/某些topic中，或者选择其它的缓冲/存储系统。这些数据源中数据元素具有固定的时间属性，是在流数据处理系统之外的其它系统生成的。比如，上亿用户通过手机终端操作触发生成的事件数据，都具有对应的事件时间；再特殊一点，可能我们希望回放（Replay）上一年手机终端用户的历史行为数据，与当前某个流数据集交叉分析才能够得到支持某类业务的特定结果，这种情况下，基于数据所具有的事件时间进行处理，就具有很重要的意义了。

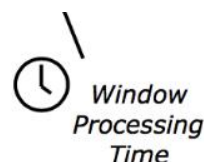
下面，我们先从Flink支持的3个与流数据处理相关的时间概念（Time Notion）：

ProcessTime、EventTime、IngestionTime。有些系统对时间概念的抽象有其它叫法，比如，Google Cloud Dataflow中称为时间域（Time Domain）。在Flink中，基于不同的Time Notion来处理流数据，具有不同的意义和结果，所以了解这3个Time Notion非常关键。

#### Time Notion

我们先看下，Apache Flink官网文档给出的一张概念图，非常形象地展示了Process Time、Event Time、Ingestion Time这三个时间分别所处的位置，如下图所示：





Time

下面，分别对这3个Time Notion进行说明如下：

## ProcessTime--事件被处理时当前系统的时间

Flink中有对数据处理的操作进行抽象，称为Transformation Operator，而对于整个Dataflow的开始和结束分别对应着Source Operator和Sink Operator，这些Operator都是在Flink集群系统所在的主机节点上，所以在基于ProcessTime的Notion进行与时间相关的数据处理时，数据处理依赖于Flink程序运行所在的主机节点系统时钟（System Clock）。

因为我们关心的是数据处理时间（Process Time），比如进行Time Window操作，对Window的指派就是基于当前Operator所在主机节点的系统时钟。也就是说，每次创建一个Window，计算Window对应的起始时间和结束时间都使用Process Time，它与外部进入的数据元素的事件时间无关。那么，后续作用于Window的操作（Function）都是基于具有Process Time特性的Window进行的。

使用ProcessTime的场景，比如，我们需要对某个App应用的用户行为进行实时统计分析与监控，由于用户可能使用不同的终端设备，这样可能会造成数据并非是实时的（如用户手机没电，导致2小时以后才会将操作行为记录批量上传上来）。而此时，如果我们按照每分钟的时间粒度做实时统计监控，那么这些数据记录延迟的太严重，如果为了等到这些记录上传上来（无法预测，具体什么时间能获取到这些数据）再做统计分析，对每分钟之内的数据进行统计分析的结果恐怕要到几个小时甚至几天后才能计算并输出结果，这不是我们所希望的。而且，数据处理系统可能也没有这么大的容量来处理海量数据的情况。结合业务需求，其实我们只需要每分钟时间内进入的数据记录，依赖当前数据处理系统的处理时间（Process Time）生成每分钟的Window，指派数据记录到指定Window并计算结果，这样就不用考虑数据元素本身自带的事件时间了。

## EventTime--事件产生的时间，它通常由事件中的时间戳描述

流数据中的数据元素可能会具有不变的事件时间（Event Time）属性，该事件时间是数据元素所代表的行为发生时就不会改变。最简单的情况下，这也最容易理解：所有进入到Flink处理系统的流数据，都是在外部的其它系统中产生的，它们产生后具有了事件时间，经过传输后，进入到Flink处理系统，理论上（如果所有系统都具有相同系统时钟）该事件时间对应的时间戳要早于进入到Flink处理系统中进行处理的时间戳，但实际应用中会出现数据记录乱序、延迟到达等问题，这也是非常普遍的。

基于EventTime的Notion，处理数据的进度（Progress）依赖于数据本身，而不是当前Flink处理系统中Operator所在主机节点的系统时钟。所以，需要有一种机制能够控制数据处理的进度，比如一个基于事件时间的Time Window创建后，具体怎么确定属于该Window的数据元素都已经到达？如果确定都到达了，然后就可以对属于这个Window的所有数据元素做满足需要的处理（如汇总、分组等）。这就要用到WaterMark机制，它能够衡量数据处理进度（表达数据到达的完整性）。

WaterMark带有一个时间戳，假设为X，进入到数据处理系统中的数据元素具有事件时间，记为Y，如果 $Y < X$ ，则所有的数据元素均已到达，可以计算并输出结果。反过来说，可能更容易理解一些：要想触发对当前Window中的数据元素进行计算，必须保证对所有进入到系统的数据元素，其事件时间 $Y \geq X$ 。如果数据元素的事件时间是有序的，那么当出现一个数据元素的事件时间 $Y < X$ ，则触发对当前Window计算，并创建另一个新的Window来指派事件时间 $Y < X$ 的数据元素到该新的Window中。

可以看到，有了WaterMark机制，对基于事件时间的流数据处理会变得特别灵活，可以根据实际业务需要选择各种组件和处理策略。比如，上面我们说到，当 $Y < X$ 则触发当前Window计算，记为t1时刻，如果流数据元素是乱序的，经过一段时间，假设t2时刻有一个数据元素的事件时间 $Y \geq X$ ，这时该怎么办呢？如果t1时刻的Window已经不存在了，但我们还是希望新出现的乱序数据元素加入到t1时刻Window的计算中，这时可以实现自定义的Trigger来满足各种业务场景的需要。

## IngestionTime--事件进入Flink的时间

IngestionTime是数据进入到Flink流数据处理系统的时间，该时间依赖于Source Operator所在主机节点的系统时钟，会为到达的数据记录指派Ingestion Time。基于IngestionTime的Notion，存在多个Source Operator的情况下，每个Source Operator会使用自己本地系统时钟指派Ingestion Time。后续基于时间相关的各种操作，都会使用数据记录中的Ingestion Time。

与EventTime相比，IngestionTime不能处理乱序、延迟到达事件的应用场景，它也就不需要指定如何生成WaterMark。

## 设定时间特性

Flink DataStream 程序的第一部分通常是设置基本时间特性。该设置定义了数据流源的行为方式（例如：它们是否将分配时间戳），以及像

**`**KeyedStream.timeWindow(Time.seconds(30))`** 这样的窗口操作应该使用上面哪种时间概念。

以下示例显示了一个 Flink 程序，该程序在每小时时间窗口中聚合事件。

```
1 final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
2
3 env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);
4
5 // 其他
6 // env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime);
7 // env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
8
9 DataStream<MyEvent> stream = env.addSource(new FlinkKafkaConsumer09<MyEvent>(topic, sc
10
11 stream
12     .keyBy( (event) -> event.getUser() )
13     .timeWindow(Time.hours(1))
14     .reduce( (a, b) -> a.add(b) )
15     .addSink(...);
```

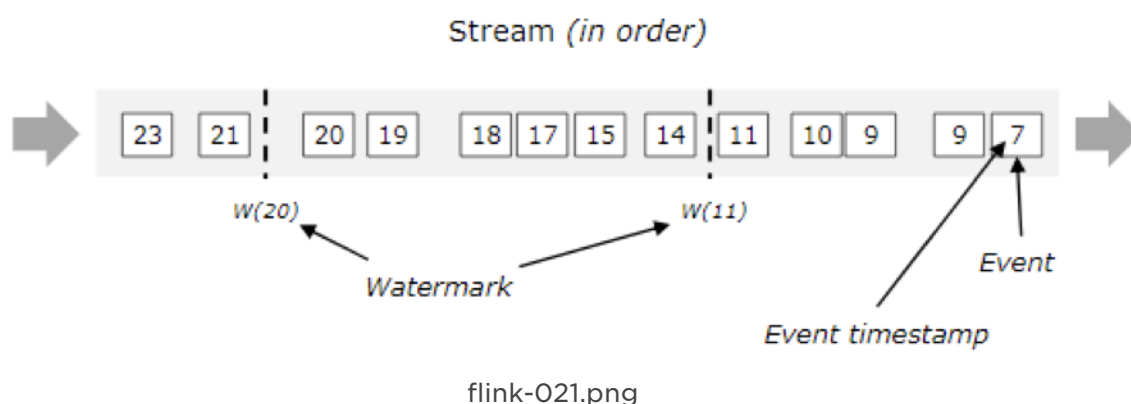
## 2. Watermark

## Watermark的类型

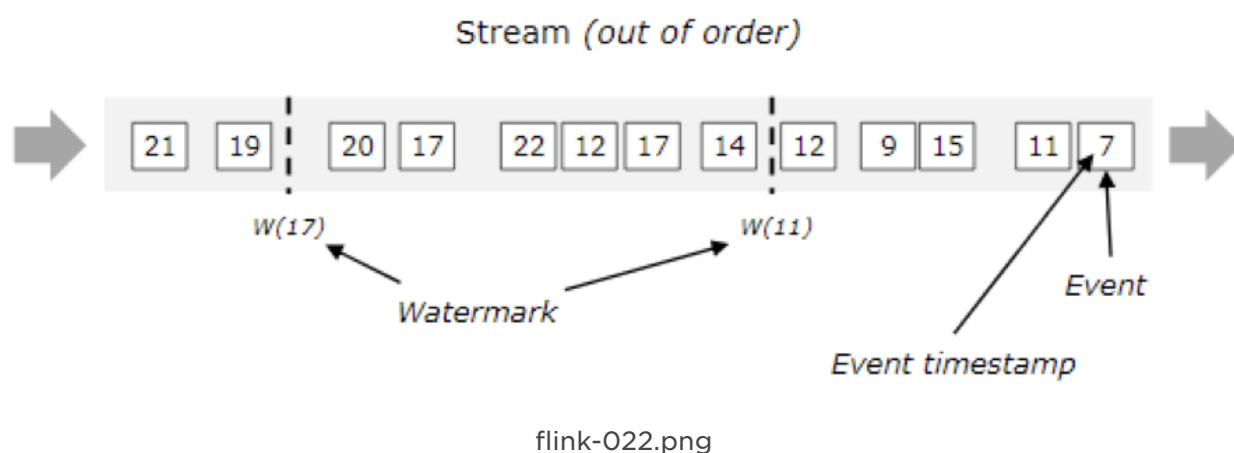
### EventTime和Watermarks

- 在使用eventTime的时候如何处理乱序数据？
- 我们知道，流处理从事件产生，到流经source，再到operator，中间是有一个过程和时间。虽然大部分情况下，流到operator的数据都是按照事件产生的时间顺序来的，但是也不排除由于网络延迟等原因，导致乱序的产生，特别是使用kafka的话，多个分区的数据无法保证有序。所以在进行window计算的时候，我们又不能无限期的等下去，必须要有个机制来保证一个特定的时间后，必须触发window去进行计算了。这个特别的机制，就是watermark，watermark是用于处理乱序事件的。
- watermark可以翻译为水位线

### 有序的流的watermarks

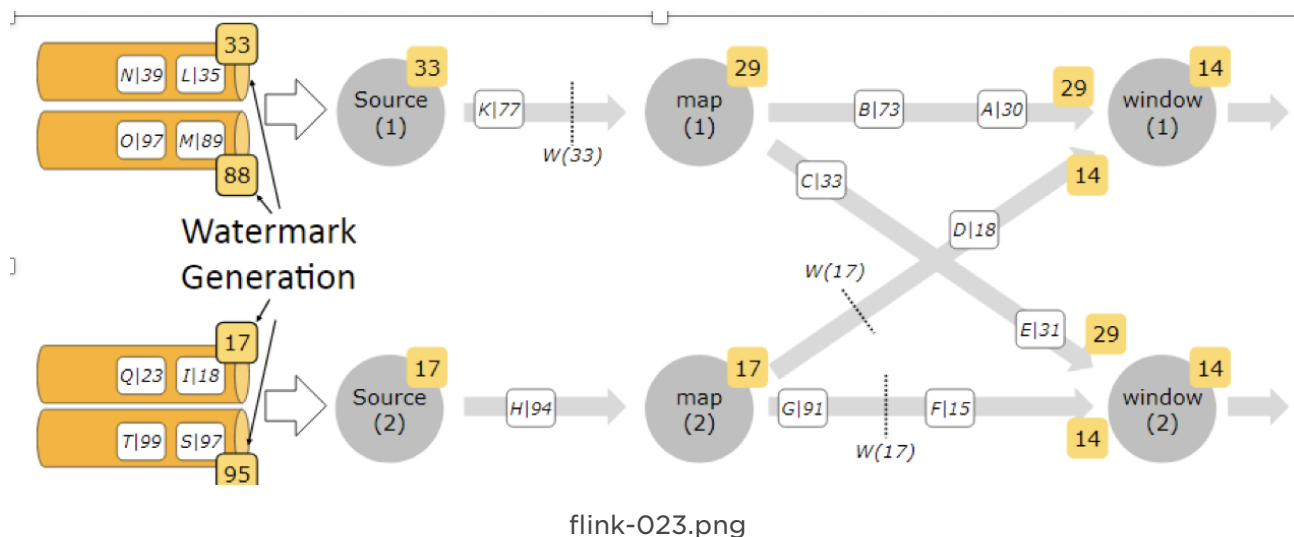


### 无序的流的watermarks



### 多并行度流的watermarks

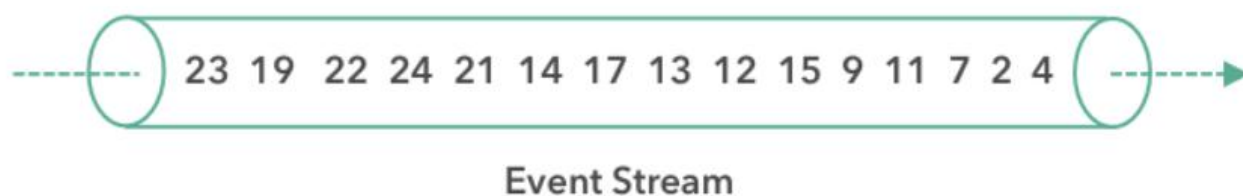
注意：多并行度的情况下，watermark对齐会取所有channel最小的watermark



## 在Apache Flink中使用watermark的4个理解

当人们第一次使用Flink时，经常会对watermark感到困惑。但其实watermark并不复杂。让我们通过一个简单的例子来说明为什么我们需要watermark，以及它的工作机制是什么样的。

在下文中的例子中，我们有一个带有时间戳的事件流，但是由于某种原因它们并不是按顺序到达的。图中的数字代表事件发生的时间戳。第一个到达的事件发生在时间4，然后它后面跟着的是发生在更早时间（时间2）的事件，以此类推：



flink-020.png

注意这是一个按照事件时间处理的例子，这意味着时间戳反映的是事件发生的时间，而不是处理事件的时间。事件时间（Event-Time）处理的强大之处在于，无论是在处理实时的数据还是重新处理历史的数据，基于事件时间创建的流计算应用都能保证结果是一样的。

现在假设我们正在尝试创建一个流计算排序算子。也就是处理一个乱序到达的事件流，并按照事件时间的顺序输出事件。

### 理解1

数据流中的第一个元素的时间是4，但是我们不能直接将它作为排序后数据流的第一个元素并输出它。因为数据是乱序到达的，也许有一个更早发生的数据还没有到达。事实上，我们能预见一些这个流的未来，也就是我们的排序算子至少要等到2这条数据的到达再输出结果。

**有缓存，就必然有延迟。**

### 理解2

如果我们做错了，我们可能会永远等待下去。首先，我们的应用程序从看到时间4的数据，然后看到时间2的数据。是否会有一个比时间2更早的数据到达呢？也许会，也许不会。我们可以一直等下去，但可能永远看不到1。

*最终，我们必须勇敢地输出 2 作为排序流的第一个结果*

### 理解3

我们需要的是某种策略，它定义了对于任何带时间戳的事件流，何时停止等待更早数据的到来。

*这正是 watermark 的作用，他们定义了何时不再等待更早的数据。*

Flink中的事件时间处理依赖于一种特殊的带时间戳的元素，成为watermark，它们会由数据源或是watermark生成器插入数据流中。具有时间戳 $t$ 的watermark可以被理解为断言了所有时间戳小于或等于 $t$ 的事件都（在某种合理的概率上）已经到达了。

1 | 注：此处原文是“小于”，译者认为应该是“小于或等于”，因为 Flink 源码中采用的是“小于或等于”的机制。

何时我们的排序算子应该停止等待，然后将事件2作为首个元素输出？答案是当收到时间戳为2（或更大）的watermark时。

### 理解4

*我们可以设想不同的策略来生成watermark。*

我们知道每个事件都会延迟一段时间才到达，而这些延迟差异会比较大，所以有些事件会比其他事件延迟更多。一种简单的方法是假设这些延迟不会超过某个最大值。Flink 把这种策略称作“有界无序生成策略”（bounded-out-of-orderness）。当然也有很多更复杂的方式去生成watermark，但是对于大多数应用来说，固定延迟的方式已经足够了。

如果想要构建一个类似排序的流应用，可以使用Flink的ProcessFunction。它提供了对事件时间计时器（基于watermark触发回调）的访问，还提供了可以用来缓存数据的托管状态接口。

## Watermark案例

### 1.watermarks的生成方式

- 通常，在接收到source的数据后，应该立刻生成watermark；但是，也可以在source后，应用简单的map或者filter操作后，再生成watermark。
- 注意：如果指定多次watermark，后面指定的会覆盖前面的值。
- 生成方式

- **With Periodic Watermarks**

- 周期性的触发watermark的生成和发送，默认是100ms
- 每隔N秒自动向流里注入一个WATERMARK
- 时间间隔由ExecutionConfig.setAutoWatermarkInterval 决定.
- 每次调用getCurrentWatermark 方法, 如果得到的WATERMARK
- 不为空并且比之前的大就注入流中
- 可以定义一个最大允许乱序的时间，这种比较常用
- 实现AssignerWithPeriodicWatermarks接口

- **With Punctuated Watermarks**

- 基于某些事件触发watermark的生成和发送
- 基于事件向流里注入一个WATERMARK，每一个元素都有机会判断是否生成一个WATERMARK.
- 如果得到的WATERMARK 不为空并且比之前的大就注入流中
- 实现AssignerWithPunctuatedWatermarks接口

## 2.watermark和window案例

这里写了一个watermark&window的flink程序，从socket读取数据  
代码：

```
1 public class StreamingWindowWatermark {
2
3     private static final Logger log = LoggerFactory.getLogger(StreamingWindowWatermark
4
5     public static void main(String[] args) throws Exception {
6         //定义socket的端口号
7         int port = 9000;
8         //获取运行环境
9         StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvirc
10
11         //设置使用eventtime，默认是使用processtime
12         env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
13
14
15
16         //设置并行度为1,默认并行度是当前机器的cpu数量
17         env.setParallelism(1);
```



```

18 //连接socket获取输入的数据
19 DataStream<String> text = env.socketTextStream("zzy", port, "\n");
20
21
22 //解析输入的数据,每行数据按逗号分隔
23 DataStream<Tuple2<String, Long>> inputMap = text.map(new MapFunction<String, T
24     @Override
25     public Tuple2<String, Long> map(String value) throws Exception {
26         String[] arr = value.split(",");
27         return new Tuple2<>(arr[0], Long.parseLong(arr[1]));
28     }
29 });
30
31 //抽取timestamp和生成watermark
32 DataStream<Tuple2<String, Long>> waterMarkStream = inputMap.assignTimestampsAr
33
34
35     Long currentMaxTimestamp = 0L;
36     final Long maxOutOfOrderness = 10000L;// 最大允许的乱序时间是10s
37
38     SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
39
40     /**
41      * 定义生成watermark的逻辑, 比当前最大时间戳晚10s
42      * 默认100ms被调用一次
43      */
44     @Nullable
45     @Override
46     public Watermark getCurrentWatermark() {
47         return new Watermark(currentMaxTimestamp - maxOutOfOrderness);
48     }
49
50 //定义如何提取timestamp
51 @Override
52 public long extractTimestamp(Tuple2<String, Long> element, long previousEl
53     long timestamp = element.f1;
54     currentMaxTimestamp = Math.max(timestamp, currentMaxTimestamp);
55     //设置多并行度时获取线程id
56     long id = Thread.currentThread().getId();
57     log.info("extractTimestamp=====>" + ",currentThreadId:" + id + ",key
58         "currentMaxTimestamp:[" + currentMaxTimestamp + "]" +
59         sdf.format(currentMaxTimestamp) + "],watermark:[" + getCurrent
60 //         System.out.println("currentThreadId:" + id + ",key:" + element.f0 +
61 //         sdf.format(currentMaxTimestamp) + "],watermark:[" + getCurre
62 //         return timestamp;
63     }
64 }
65 });
66
67 DataStream<String> window = waterMarkStream.keyBy(0)//分组
68     .window(TumblingEventTimeWindows.of(Time.seconds(3)))//按照消息的EventT
69     .apply(new WindowFunction<Tuple2<String, Long>, String, Tuple, TimeWir
70     /**
71      * 对window内的数据进行排序, 保证数据的顺序
72      * @param tuple
73      * @param window
74      * @param input
75      * @param out
76      * @throws Exception
77      */
78     @Override
79     public void apply(Tuple tuple, TimeWindow window, Iterable<Tuple2<

```



```

81 |         String key = tuple.toString();
82 |         List<Long> arrarList = new ArrayList<Long>();
83 |         Iterator<Tuple2<String, Long>> it = input.iterator();
84 |         while (it.hasNext()) {
85 |             Tuple2<String, Long> next = it.next();
86 |             //时间戳放到了arrarList里
87 |             arrarList.add(next.f1);
88 |         }
89 |         Collections.sort(arrarList);
90 |         SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
91 |         String result = key + "," + arrarList.size() + "," + sdf.format(window.getStart()) + "," + sdf.format(window.getEnd());
92 |         out.collect(result);
93 |     }
94 | }
95 | }
96 | });
97 | //测试-把结果打印到控制台即可
98 | window.print();
99 |
//注意：因为flink是懒加载的，所以必须调用execute方法，上面的代码才会执行
env.execute("eventtime-watermark");
}
}

```

## 启动程序StreamingWindowWatermark

打印日志：

```

2019-02-14 11:57:36,674 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [org.apache.flink.streaming.
backend has been configured, using default (Memory / JobManager) MemoryStateBackend (data in heap memory / check
savepoints: 'null', asynchronous: TRUE, maxStateSize: 5242880)
2019-02-14 11:57:36,715 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [org.apache.flink.streaming.
[INFO] - Connecting to server socket zzy:9000
2019-02-14 11:57:36,741 [Window(TumblingEventTimeWindows(3000), EventTimeTrigger, WindowFunction$3) -> Sink: Print
.heap.HeapKevdStateBackend] [INFO] - Initializing heap keyed state backend with stream factory.

```

flink-O24.png

```

1 | 2019-02-14 11:57:36,715 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
2 | 2019-02-14 11:57:36,741 [Window(TumblingEventTimeWindows(3000), EventTimeTrigger, Winc

```

首先，我们开启socket，输入第一条数据，数据格式是(id,时间戳)：

```

1 | → /data nc -l 9000
2 | 0001,1550116440000

```

输出如下：

```

savepoints: 'null', asynchronous: TRUE, maxStateSize: 5242880)
2019-02-14 11:57:36,715 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [org.apache.flink.streaming.api.functions.source.SocketTextStreamFunction]
[INFO] - Connecting to server socket zzy:9000
2019-02-14 11:57:36,741 [Window(TumblingEventTimeWindows(3000), EventTimeTrigger, WindowFunction$3) -> Sink: Print to Std. Out (1/1)] [org.apache.flink.runtime]

```

```
.heap.HeapKeyedStateBackend] [INFO] - Initializing heap keyed state backend with stream factory.  
2019-02-14 11:58:48,690 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [xuwei.tech.streaming.watermark.StreamingWindowWatermark] [INFO] -  
extractTimestamp=====>,currentThreadId:37,key:0001,eventtime:[1550116440000|2019-02-14 11:54:00.000],currentMaxTimestamp:[1550116440000|2019-02-14 11:54:00  
watermark:[1550116430000|2019-02-14 11:53:50.000]
```

flink-O25.png

1 | 019-02-14 11:58:48,690 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [

汇总下表：

key	EventTime	currentMaxTimestamp	watermark
000001	1550116440000.00	1550116440000.00	1550116430000.00
	2019/2/14 11:54:00	2019/2/14 11:54:00	2019/2/14 11:53:50

flink-O26.png

此时，wartermark的时间按照逻辑，已经落后于currentMaxTimestamp10秒了。

我们继续输入：

0001,1550116444000

输出内容如下：

1 | 2019-02-14 12:08:25,474 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]

再次汇总表：

key	EventTime	currentMaxTimestamp	watermark
000001	1550116440000.00	1550116440000.00	1550116430000.00
	2019/2/14 11:54:00	2019/2/14 11:54:00	2019/2/14 11:53:50
000001	1550116444000.00	1550116444000.00	1550116434000.00
	2019/2/14 11:54:04	2019/2/14 11:54:04	2019/2/14 11:53:54

flink-O27.png

继续输入：

0001,1550116450000

输出内容如下：

1 | 2019-02-14 14:30:27,480 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]

汇总下表：

key	EventTime	currentMaxTimestamp	watermark
000001	1550116440000.00	1550116440000.00	1550116430000.00
	2019/2/14 11:54:00	2019/2/14 11:54:00	2019/2/14 11:53:50
000001	1550116444000.00	1550116444000.00	1550116434000.00
	2019/2/14 11:54:04	2019/2/14 11:54:04	2019/2/14 11:53:54
000001	1550116450000.00	1550116450000.00	1550116440000.00
	2019/2/14 11:54:10	2019/2/14 11:54:10	2019/2/14 11:54:00

flink-O28.png

到这里，window仍然没有被触发，此时watermark的时间已经等于了第一条数据的Event Time了。那么window到底什么时候被触发呢？我们再次输入：  
0001,1550116451000  
输出内容如下：

1 | 2019-02-14 14:36:01,479 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]

汇总如下：

key	EventTime	currentMaxTimestamp	watermark
000001	1550116440000.00	1550116440000.00	1550116430000.00
	2019/2/14 11:54:00	2019/2/14 11:54:00	2019/2/14 11:53:50
000001	1550116444000.00	1550116444000.00	1550116434000.00
	2019/2/14 11:54:04	2019/2/14 11:54:04	2019/2/14 11:53:54
000001	1550116450000.00	1550116450000.00	1550116440000.00
	2019/2/14 11:54:10	2019/2/14 11:54:10	2019/2/14 11:54:00
000001	1550116451000.00	1550116451000.00	1550116441000.00
	2019/2/14 11:54:11	2019/2/14 11:54:11	2019/2/14 11:54:01

flink-O29.png

可以看到window仍然没有触发，此时，我们的数据已经发到2019-02-14 11:54:11.000了，最早的数据已经过去了11秒了，还没有开始计算。那是不是要等到13（10+3）秒过去了，才开始触发window呢？答案是否定的。

我们再次增加1秒，输入：

0001,1550116452000

输出内容如下：

```
1 | 2019-02-14 14:40:50,332 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
```

汇总如下：

key	EventTime	currentMaxTimestamp	watermark	window_start_time	window_end_time
000001	1550116440000.00	1550116440000.00	1550116430000.00		
	2019/2/14 11:54:00	2019/2/14 11:54:00	2019/2/14 11:53:50		
000001	1550116444000.00	1550116444000.00	1550116434000.00		
	2019/2/14 11:54:04	2019/2/14 11:54:04	2019/2/14 11:53:54		
000001	1550116450000.00	1550116450000.00	1550116440000.00		
	2019/2/14 11:54:10	2019/2/14 11:54:10	2019/2/14 11:54:00		
000001	1550116451000.00	1550116451000.00	1550116441000.00		
	2019/2/14 11:54:11	2019/2/14 11:54:11	2019/2/14 11:54:01		
	1550116452000.00	1550116452000.00	1550116442000.00		
	2019/2/14 11:54:12	2019/2/14 11:54:12	2019/2/14 11:54:02 [2019/2/14 11:54:00	2019/2/14 11:54:03]	

flink-030.png

Window依旧没有触发

我们再次增加1s，输入：

0001,1550116453000

输出内容如下：

```
1 | 2019-02-14 14:51:10,020 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
2 | (0001),1,2019-02-14 11:54:00.000,2019-02-14 11:54:00.000,2019-02-14 11:54:00.000,2019-
```

可以看到触发了window操作，打印数据到控制台了

```
watermark:[1550116442000|2019-02-14 11:54:02.000]
2019-02-14 14:51:10,020 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [xuwei.tech.streaming
extractTimestamp=====>,currentThreadId:37,key:0001,eventtime:[1550116453000|2019-02-14 11:54:13.000],cur
watermark:[1550116443000|2019-02-14 11:54:03.000]
(0001),1,2019-02-14 11:54:00.000 2019-02-14 11:54:00.000,2019-02-14 11:54:00.000,2019-02-14 11:54:03.000
```

flink-031.png

```

1 | String result = key + "," + arrarList.size() + "," + sdf.format(arrarList.get(0)) + ",
2 |           + "," + sdf.format(window.getStart()) + "," + sdf.format(window.getEnd());
3 | out.collect(result);

```

汇总如下：

key	EventTime	currentMaxTimestamp	watermark	window_start_time	window_end_time
000001	1550116440000.00	1550116440000.00	1550116430000.00		
	2019/2/14 11:54:00	2019/2/14 11:54:00	2019/2/14 11:53:50		
000001	1550116444000.00	1550116444000.00	1550116434000.00		
	2019/2/14 11:54:04	2019/2/14 11:54:04	2019/2/14 11:53:54		
000001	1550116450000.00	1550116450000.00	1550116440000.00		
	2019/2/14 11:54:10	2019/2/14 11:54:10	2019/2/14 11:54:00		
000001	1550116451000.00	1550116451000.00	1550116441000.00		
	2019/2/14 11:54:11	2019/2/14 11:54:11	2019/2/14 11:54:01		
000001	1550116452000.00	1550116452000.00	1550116442000.00		
	2019/2/14 11:54:12	2019/2/14 11:54:12	2019/2/14 11:54:02 [2019/2/14 11:54:00	2019/2/14 11:54:03)	
000001	1550116453000.00	1550116453000.00	1550116443000.00		
	2019/2/14 11:54:13	2019/2/14 11:54:13	2019/2/14 11:54:03 [2019/2/14 11:54:00	2019/2/14 11:54:03)	

flink-O32.png

到这里，我们做一个说明：

window的触发机制，是先按照自然时间将window划分，如果window大小是3秒，那么1分钟内会把window划分为如下的形式（注意window是左闭右开的）：

```

1 | [00:00:00,00:00:03)
2 | [00:00:03,00:00:06)
3 | ...
4 | [00:00:57,00:01:00)

```

如果window大小是10秒，则window会被分为如下的形式：

```

1 | [00:00:00,00:00:10)
2 | [00:00:10,00:00:20)
3 | ...
4 | [00:00:50,00:01:00)

```

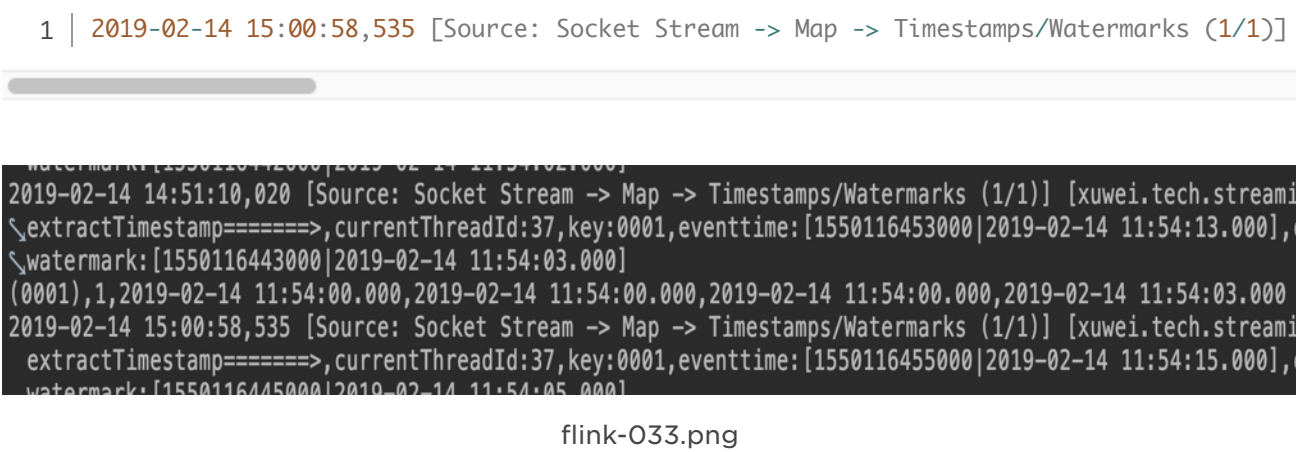
**window**的设定无关数据本身，而是系统定义好了的。



输入的数据中，根据自身的Event Time，将数据划分到不同的window中，如果window中有数据，则当watermark时间>=Event Time时，就符合了window触发的条件了，最终决定window触发，还是由数据本身的Event Time所属的window中的window\_end\_time决定。

上面的测试中，最后一条数据到达后，其水位线已经升至19:34:24秒，正好是最早的一条记录所在window的window\_end\_time，所以window就被触发了。

为了验证window的触发机制，我们继续输入数据：  
0001,1550116455000  
输出内容如下：



汇总表：

key	EventTime	currentMaxTimestamp	watermark	window_start_time	window_end_time
000001	1550116440000.00	1550116440000.00	1550116430000.00		
	2019/2/14 11:54:00	2019/2/14 11:54:00	2019/2/14 11:53:50		
000001	1550116444000.00	1550116444000.00	1550116434000.00		
	2019/2/14 11:54:04	2019/2/14 11:54:04	2019/2/14 11:53:54		
000001	1550116450000.00	1550116450000.00	1550116440000.00		
	2019/2/14 11:54:10	2019/2/14 11:54:10	2019/2/14 11:54:00		
000001	1550116451000.00	1550116451000.00	1550116441000.00		
	2019/2/14 11:54:11	2019/2/14 11:54:11	2019/2/14 11:54:01		
000001	1550116452000.00	1550116452000.00	1550116442000.00		
	2019/2/14 11:54:12	2019/2/14 11:54:12	2019/2/14 11:54:02 [2019/2/14 11:54:00	2019/2/14 11:54:03)	
000001	1550116453000.00	1550116453000.00	1550116443000.00		
	2019/2/14 11:54:13	2019/2/14 11:54:13	2019/2/14 11:54:03 [2019/2/14 11:54:00	2019/2/14 11:54:03)	
	1550116455000.00	1550116455000.00	1550116445000.00		
	2019/2/14 11:54:15	2019/2/14 11:54:15	2019/2/14 11:54:05		

flink-O34.png

此时，watermark时间虽然已经达到了第二条数据的时间，但是由于其没有达到第二条数据

所在window的结束时间，所以window并没有被触发。那么，第二条数据所在的window时间是：

[2019/2/14 11:54:03, 2019/2/14 11:54:06)

也就是说，我们必须输入一个11: 54: 06秒的数据，第二条数据所在的window才会被触发。

我们继续输入：

0001,1550116456000

输出如下：

```
1 | 2019-02-14 15:07:48,879 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
2 | (0001),1,2019-02-14 11:54:04.000,2019-02-14 11:54:04.000,2019-02-14 11:54:03.000,2019-
```

```
2019-02-14 15:00:58,535 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [xuwei.tech.streaming
extractTimestamp=====>,currentThreadId:37,key:0001,eventtime:[1550116455000|2019-02-14 11:54:15.000],cu
watermark:[1550116445000|2019-02-14 11:54:05.000]
2019-02-14 15:07:48,879 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [xuwei.tech.streaming
extractTimestamp=====>,currentThreadId:37,key:0001,eventtime:[1550116456000|2019-02-14 11:54:16.000],cu
watermark:[1550116446000|2019-02-14 11:54:06.000]
(0001),1,2019-02-14 11:54:04.000,2019-02-14 11:54:04.000,2019-02-14 11:54:03.000,2019-02-14 11:54:06.000
```

flink-O35.png

可以看到是有触发windows操作的

汇总：

key	EventTime	currentMaxTimestamp	watermark	window_start_time	window_end_time
000001	1550116440000.00	1550116440000.00	1550116430000.00		
	2019/2/14 11:54:00	2019/2/14 11:54:00	2019/2/14 11:53:50	2019/2/14 11:54:00	2019/2/14 11:54:03)
000001	1550116444000.00	1550116444000.00	1550116434000.00		
	2019/2/14 11:54:04	2019/2/14 11:54:04	2019/2/14 11:53:54	2019/2/14 11:54:03	2019/2/14 11:54:06)
000001	1550116450000.00	1550116450000.00	1550116440000.00		
	2019/2/14 11:54:10	2019/2/14 11:54:10	2019/2/14 11:54:00		
000001	1550116451000.00	1550116451000.00	1550116441000.00		
	2019/2/14 11:54:11	2019/2/14 11:54:11	2019/2/14 11:54:01		
000001	1550116452000.00	1550116452000.00	1550116442000.00		
	2019/2/14 11:54:12	2019/2/14 11:54:12	2019/2/14 11:54:02	2019/2/14 11:54:00	2019/2/14 11:54:03)
000001	1550116453000.00	1550116453000.00	1550116443000.00		
	2019/2/14 11:54:13	2019/2/14 11:54:13	2019/2/14 11:54:03	2019/2/14 11:54:00	2019/2/14 11:54:03)
000001	1550116455000.00	1550116455000.00	1550116445000.00		
	2019/2/14 11:54:15	2019/2/14 11:54:15	2019/2/14 11:54:05		
	1550116456000.00	1550116456000.00	1550116446000.00		
	2019/2/14 11:54:16	2019/2/14 11:54:16	2019/2/14 11:54:06		



下面划重点了

### watermark触发条件

此时，我们已经看到，window的触发要符合以下几个条件：

- 1、**watermark时间 >= window\_end\_time**
- 2、在**[window\_start\_time,window\_end\_time)**中有数据存在

同时满足了以上2个条件，window才会触发。

而且，这里要强调一点，watermark是一个全局的值，不是某一个key下的值，所以即使不是同一个key的数据，其watermark也会增加，例如：

0002, 1550116458000

输出如下：

```
1 | 2019-02-14 15:22:04,219 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
```

我们看到，currentMaxTimestamp也增加到2019-02-14 11:54:08.000了。

### watermark+window处理乱序

我们上面的测试，数据都是按照时间顺序递增的，现在，我们输入一些乱序的（late）数据，看看watermark结合window机制，是如何处理乱序的。

输入：

```
1 | 0001,1550116440000
2 | 0001,1550116441000
3 | 0001,1550116442000
4 | 0001,1550116443000
5 | 0001,1550116444000
6 | 0001,1550116445000
7 | 0001,1550116446000
8 | 0001,1550116450000
9 | 0001,1550116451000
10 | 0001,1550116452000
11 | 0001,1550116453000
12 | 0001,1550116456000
13 | 0001,1550116460000
14 | 0001,1550116461000
15 | 0001,1550116462000
16 | 0001,1550116464000
```

输出如下：

```

1 | 2019-02-14 15:34:49,469 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
2 | 2019-02-14 15:34:50,276 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
3 | (0001),3,2019-02-14 11:54:00.000,2019-02-14 11:54:02.000,2019-02-14 11:54:00.000,2019-
4 | 2019-02-14 15:35:05,916 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
5 | (0001),3,2019-02-14 11:54:03.000,2019-02-14 11:54:05.000,2019-02-14 11:54:03.000,2019-
6 | 2019-02-14 15:35:17,804 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
7 | 2019-02-14 15:35:17,804 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
8 | 2019-02-14 15:35:17,804 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
9 | (0001),1,2019-02-14 11:54:06.000,2019-02-14 11:54:06.000,2019-02-14 11:54:06.000,2019-
10 | (0001),2,2019-02-14 11:54:10.000,2019-02-14 11:54:11.000,2019-02-14 11:54:09.000,2019-
11 | 2019-02-14 15:35:48,356 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]

```

再输入：

0001,1550116454000

输出如下：

```

1 | 2019-02-14 15:40:41,051 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]

```

汇总：

key	EventTime	currentMaxTimestamp	watermark	window_start_time	window_end_time
000001	1550116451000.00	1550116451000.00	1550116441000.00		
	2019/2/14 11:54:11	2019/2/14 11:54:11	2019/2/14 11:54:01		
000001	1550116452000.00	1550116452000.00	1550116442000.00		
	2019/2/14 11:54:12	2019/2/14 11:54:12	2019/2/14 11:54:02 [2019/2/14 11:54:00	2019/2/14 11:54:03)	
000001	1550116453000.00	1550116453000.00	1550116443000.00		
	2019/2/14 11:54:13	2019/2/14 11:54:13	2019/2/14 11:54:03 [2019/2/14 11:54:00	2019/2/14 11:54:03)	
000001	1550116455000.00	1550116455000.00	1550116445000.00		
	2019/2/14 11:54:15	2019/2/14 11:54:15	2019/2/14 11:54:05		
000001	1550116456000.00	1550116456000.00	1550116446000.00		
	2019/2/14 11:54:16	2019/2/14 11:54:16	2019/2/14 11:54:06		
000002	1550116458000.00	1550116458000.00	1550116448000.00		
	2019/2/14 11:54:18	2019/2/14 11:54:18	2019/2/14 11:54:08		
000001	1550116464000.00	1550116464000.00	1550116454000.00		
	2019/2/14 11:54:24	2019/2/14 11:54:24	2019/2/14 11:54:14		
000001	1550116454000.00	1550116464000.00	1550116454000.00		
	2019/2/14 11:54:14	2019/2/14 11:54:24	2019/2/14 11:54:14 [2019/2/14 11:54:12	2019/2/14 11:54:15)	

flink-037.png

可以看到，虽然我们输入了一个2019/2/14 11:54:14的数据，但是currentMaxTimestamp和watermark都没变。

此时，按照我们上面提到的公式：

- 1、watermark时间 >= window\_end\_time

- 2、在[window\_start\_time,window\_end\_time)中有数据存在

那如果我们再次输入一条2019/2/14 11:54:25的数据，此时watermark时间会升高到19:34:33，这时的window一定就会触发了，我们试一试：

输入：

0001,1550116465000

输出如下：

```
1 | 2019-02-14 15:48:07,322 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
2 | (0001),3,2019-02-14 11:54:12.000,2019-02-14 11:54:14.000,2019-02-14 11:54:12.000,2019-
```

可以看到触发了window操作，打印了2019/2/14 11:54:14这条数据

汇总：

key	EventTime	currentMaxTimestamp	watermark	window_start_time	window_end_time
000001	1550116455000.00	1550116455000.00	1550116445000.00		
	2019/2/14 11:54:15	2019/2/14 11:54:15	2019/2/14 11:54:05		
000001	1550116456000.00	1550116456000.00	1550116446000.00		
	2019/2/14 11:54:16	2019/2/14 11:54:16	2019/2/14 11:54:06		
000002	1550116458000.00	1550116458000.00	1550116448000.00		
	2019/2/14 11:54:18	2019/2/14 11:54:18	2019/2/14 11:54:08		
000001	1550116464000.00	1550116464000.00	1550116454000.00		
	2019/2/14 11:54:24	2019/2/14 11:54:24	2019/2/14 11:54:14		
000001	1550116454000.00	1550116464000.00	1550116454000.00		
	2019/2/14 11:54:14	2019/2/14 11:54:24	2019/2/14 11:54:14	2019/2/14 11:54:12	2019/2/14 11:54:15
000001	1550116465000.00	1550116465000.00	1550116455000.00		
	2019/2/14 11:54:25	2019/2/14 11:54:25	2019/2/14 11:54:15		

flink-038.png

上边的结果，已经表明，对于out-of-order的数据，Flink可以通过watermark机制结合window的操作，来处理一定范围内的乱序数据。那么对于“迟到”太多的数据，Flink是怎么处理的呢？

## late element的处理

运行代码：StreamingWindowWatermark2

```

1 public class StreamingWindowWatermark2 {
2
3     private static final Logger log = LoggerFactory.getLogger(StreamingWindowWatermark
4
5
6     public static void main(String[] args) throws Exception {
7         //定义socket的端口号
8         int port = 9000;
9         //获取运行环境
10        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvirc
11
12
13        //设置使用eventtime, 默认是使用processtime
14        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
15
16        //设置并行度为1, 默认并行度是当前机器的cpu数量
17        env.setParallelism(1);
18
19        //连接socket获取输入的数据
20        DataStream<String> text = env.socketTextStream("zzy", port, "\n");
21
22        //解析输入的数据
23        DataStream<Tuple2<String, Long>> inputMap = text.map(new MapFunction<String, T
24            @Override
25            public Tuple2<String, Long> map(String value) throws Exception {
26                String[] arr = value.split(",");
27                return new Tuple2<>(arr[0], Long.parseLong(arr[1]));
28            }
29        });
30
31        //抽取timestamp和生成watermark
32        DataStream<Tuple2<String, Long>> waterMarkStream = inputMap.assignTimestampsAr
33
34
35        Long currentMaxTimestamp = 0L;
36        final Long maxOutOfOrderness = 10000L; // 最大允许的乱序时间是10s--乱序时间
37
38        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");
39        /**
40         * 定义生成watermark的逻辑
41         * 默认100ms被调用一次
42         */
43        @Nullable
44        @Override
45        public Watermark getCurrentWatermark() {
46            return new Watermark(currentMaxTimestamp - maxOutOfOrderness);
47        }
48
49        //定义如何提取timestamp
50        @Override
51        public long extractTimestamp(Tuple2<String, Long> element, long previousEl
52            long timestamp = element.f1;
53            currentMaxTimestamp = Math.max(timestamp, currentMaxTimestamp);
54            log.info("key:"+element.f0+",eventtime:["+element.f1+"|"+sdf.format(el
55                sdf.format(currentMaxTimestamp)+"],watermark:["+getCurrentWate
56
57
58        //            System.out.println("key:"+element.f0+",eventtime:["+element.f1+"|"+s
59        //            sdf.format(currentMaxTimestamp)+"],watermark:["+getCurrentWc
60            return timestamp;
61        }
62    }

```

```

63     });
64
65     //保存被丢弃的数据
66     OutputTag

```

我们输入一个乱序很多的数据来测试下：  
输入：

```
1 | → /data nc -l 9000
2 | 0001,1550116440000
3 | 0001,1550116443000
4 | 0001,1550116444000
5 | 0001,1550116445000
6 | 0001,1550116446000
7 | 0001,1550116450000
8 | 0001,1550116451000
9 | 0001,1550116452000
10 | 0001,1550116453000
11 | 0001,1550116441000
12 | 0001,1550116454000
13 | 0001,1550116455000
14 | 0001,1550116455000
15 | 0001,1550116457000
16 | 0001,1550116458000
```

输出如下:

```
1 | 2019-02-14 16:34:27,881 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
2 | 2019-02-14 16:34:27,881 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
3 | 2019-02-14 16:34:27,882 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
4 | key:(0001),size:3,2019-02-14 11:54:03.000,2019-02-14 11:54:05.000,2019-02-14 11:54:03.
5 | 2019-02-14 16:34:28,420 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
```

输入数据:

```
0001,1550116447000
0001,1550116446000
```

输出如下:

```
1 | 2019-02-14 16:35:25,902 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
2 | 2019-02-14 16:39:11,450 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
```

```
2019-02-14 16:34:27,882 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [xuwei.tec
key:0001,eventtime:[1550116457000|2019-02-14 11:54:17.000],currentMaxTimestamp:[1550116457000|
11:54:07.000]
key:(0001),size:3,2019-02-14 11:54:03.000,2019-02-14 11:54:05.000,2019-02-14 11:54:03.000,2019-0
2019-02-14 16:34:28,420 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [xuwei.tec
key:0001,eventtime:[1550116458000|2019-02-14 11:54:18.000],currentMaxTimestamp:[1550116458000|
11:54:08.000]
2019-02-14 16:35:25,902 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [xuwei.tec
key:0001,eventtime:[1550116447000|2019-02-14 11:54:07.000],currentMaxTimestamp:[1550116458000|
11:54:08.000]
2019-02-14 16:39:11,450 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [xuwei.tec
key:0001,eventtime:[1550116446000|2019-02-14 11:54:06.000],currentMaxTimestamp:[1550116458000|
```

flink-039.png

没有触发window

550116446000|2019-02-14 11:54:06.000 对应的window是  
[2019-02-14 11:54:06.000, 2019-02-14 11:54:09.000)

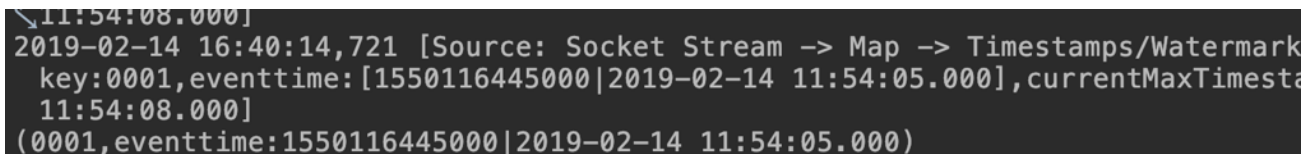
而现在的watermark是2019-02-14 11:54:08.000 比2019-02-14 11:54:09.000小，输入eventtime是1550116445000|2019-02-14 11:54:05.000的事件

输入：

0001,1550116445000

输出：

```
1 | 2019-02-14 16:40:14,721 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
2 | (0001,eventtime:1550116445000|2019-02-14 11:54:05.000)
```



```
2019-02-14 16:40:14,721 [Source: Socket Stream -> Map -> Timestamps/Watermark
key:0001,eventtime:[1550116445000|2019-02-14 11:54:05.000],currentMaxTimesta
11:54:08.000]
(0001,eventtime:1550116445000|2019-02-14 11:54:05.000)
```

flink-040.png

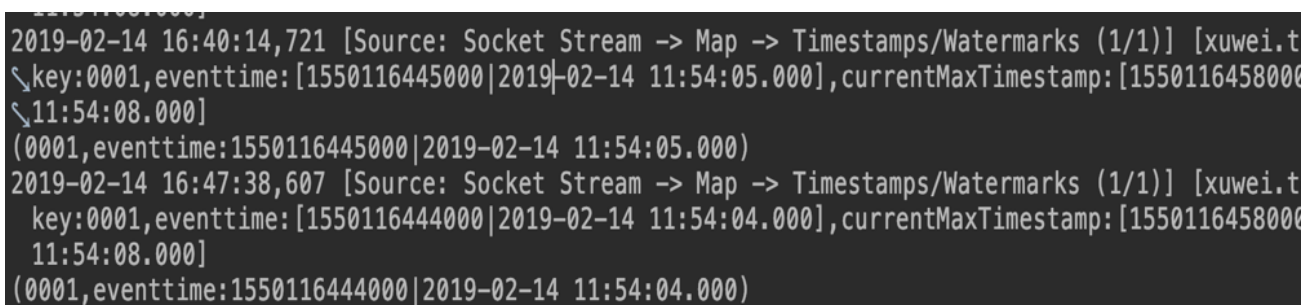
我们输入数据：

0001,1550116444000

输出：

```
1 | 2019-02-14 16:47:38,607 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)]
2 | (0001,eventtime:1550116444000|2019-02-14 11:54:04.000)
```

可以看出来是有触发window的



```
2019-02-14 16:40:14,721 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [xuwei.t
key:0001,eventtime:[1550116445000|2019-02-14 11:54:05.000],currentMaxTimestamp:[1550116458000
11:54:08.000]
(0001,eventtime:1550116445000|2019-02-14 11:54:05.000)
2019-02-14 16:47:38,607 [Source: Socket Stream -> Map -> Timestamps/Watermarks (1/1)] [xuwei.t
key:0001,eventtime:[1550116444000|2019-02-14 11:54:04.000],currentMaxTimestamp:[1550116458000
11:54:08.000]
(0001,eventtime:1550116444000|2019-02-14 11:54:04.000)
```

flink-041.png

## 总结

- 1.Flink如何处理乱序？

watermark+window机制，window中可以对input进行按照Event Time排序，使得完全按照Event Time发生的顺序去处理数据，以达到处理乱序数据的目的。

- 2. Flink何时触发window？

- 1、 watermark时间  $\geq$  window\_end\_time（对于out-of-order以及正常的数据而



- 2、在[window\_start\_time>window\_end\_time)中有数据存在
- 3.Flink应该如何设置最大乱序时间？  
这个要结合自己的业务以及数据情况去设置。如果maxOutOfOrderness设置的太小，而自身数据发送时由于网络等原因导致乱序或者late太多，那么最终的结果就是会有很多单条的数据在window中被触发，数据的正确性影响太大。

参考：