# 第七章 EventTime与Window

## 7.1 EventTime的引入

**在Flink的流式处理中，绝大部分的业务都会使用eventTime，一般只在eventTime无法使用时，才会被迫使用ProcessingTime或者IngestionTime**。

如果要使用EventTime，那么需要引入EventTime的时间属性，引入方式如下所示：

**val env = StreamExecutionEnvironment.getExecutionEnvironment**

**// 从调用时刻开始给env创建的每一个stream追加时间特征**

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

## 7.2 Watermark

### 7.2.1 基本概念

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的，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、分布式等原因，导致乱序的产生，所谓乱序，就是指Flink接收到的事件的先后顺序不是严格按照事件的Event Time顺序排列的。

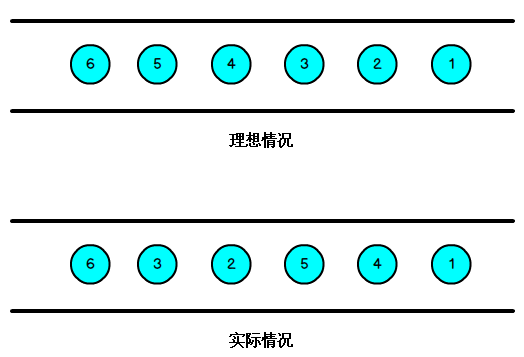


图 数据的乱序

那么此时出现一个问题，一旦出现乱序，如果只根据eventTime决定window的运行，我们不能明确数据是否全部到位，但又不能无限期的等下去，此时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是一种衡量Event Time进展的机制，它是数据本身的一个隐藏属性，数据本身携带着对应的Watermark。

**Watermark是用于处理乱序事件的，而正确的处理乱序事件，通常用Watermark机制结合window来实现**。

**数据流中的Watermark用于表示timestamp小于Watermark的数据，都已经到达了，因此，window的执行也是由Watermark触发的**。

**Watermark可以理解成一个延迟触发机制，我们可以设置Watermark的延时时长t，每次系统会校验已经到达的数据中最大的maxEventTime，然后认定eventTime小于maxEventTime - t的所有数据都已经到达，如果有窗口的停止时间等于maxEventTime – t，那么这个窗口被触发执行**。

有序流的Watermarker如下图所示：（Watermark设置为0）

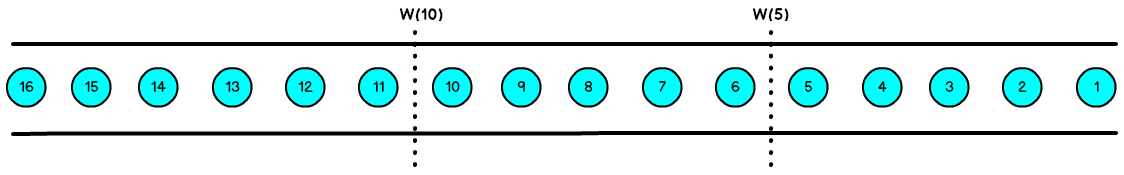


图 有序数据的Watermark

乱序流的Watermarker如下图所示：（Watermark设置为2）

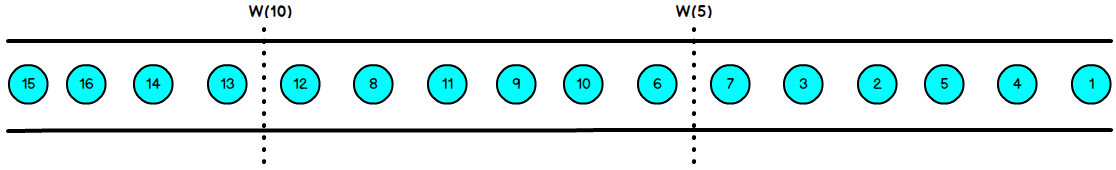


图 无序数据的Watermark

**当Flink接收到每一条数据时，都会产生一条Watermark，这条Watermark就等于当前所有到达数据中的maxEventTime - 延迟时长，也就是说，Watermark是由数据携带的，一旦数据携带的Watermark比当前未触发的窗口的停止时间要晚，那么就会触发相应窗口的执行。由于Watermark是由数据携带的，因此，如果运行过程中无法获取新的数据，那么没有被触发的窗口将永远都不被触发**。

上图中，我们设置的允许最大延迟到达时间为2s，所以时间戳为7s的事件对应的Watermark是5s，时间戳为12s的事件的Watermark是10s，如果我们的窗口1是1s~5s，窗口2是6s~10s，那么时间戳为7s的事件到达时的Watermarker恰好触发窗口1，时间戳为12s的事件到达时的Watermark恰好触发窗口2。

Watermark 就是触发前一窗口的“关窗时间”，一旦触发关门那么以当前时刻为准在窗口范围内的所有所有数据都会收入窗中。

只要没有达到水位那么不管现实中的时间推进了多久都不会触发关窗。

### 7.2.2 Watermark的引入

|  |
| --- |
| **val** textWithEventTimeDstream: DataStream[(String, Long, Int)] = textWithTsDstream.assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[(String, Long, Int)](Time.*milliseconds*(1000)) {  **override def** extractTimestamp(element: (String, Long, Int)): Long = {   **return** element.\_2  } }) |

## 7.3 EvnetTimeWindow API

### 7.3.1 滚动窗口（TumblingEventTimeWindows）

|  |
| --- |
| **def** main(args: Array[String]): Unit = {  *// 环境* **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  env.setParallelism(1)   **val** dstream: DataStream[String] = env.socketTextStream(**"hadoop1"**,7777)     **val** textWithTsDstream: DataStream[(String, Long, Int)] = dstream.map { text =>  **val** arr: Array[String] = text.split(**" "**)  (arr(0), arr(1).toLong, 1)  }  **val** textWithEventTimeDstream: DataStream[(String, Long, Int)] = textWithTsDstream.assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[(String, Long, Int)](Time.*milliseconds*(1000)) {  **override def** extractTimestamp(element: (String, Long, Int)): Long = {   **return** element.\_2  }  })   **val** textKeyStream: KeyedStream[(String, Long, Int), Tuple] = textWithEventTimeDstream.keyBy(0)  textKeyStream.print(**"textkey:"**)   **val** windowStream: WindowedStream[(String, Long, Int), Tuple, TimeWindow] = textKeyStream.window(TumblingEventTimeWindows.*of*(Time.*seconds*(2)))   **val** groupDstream: DataStream[mutable.HashSet[Long]] = windowStream.fold(**new** mutable.HashSet[Long]()) { **case** (set, (key, ts, count)) =>  set += ts  }   groupDstream.print(**"window::::"**).setParallelism(1)   env.execute()   }  } |

**结果是按照Event Time的时间窗口计算得出的，而无关系统的时间（包括输入的快慢）**。

### 7.3.2 滑动窗口（SlidingEventTimeWindows）

|  |
| --- |
| **def** main(args: Array[String]): Unit = {  *// 环境* **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  env.setParallelism(1)   **val** dstream: DataStream[String] = env.socketTextStream(**"hadoop1"**,7777)     **val** textWithTsDstream: DataStream[(String, Long, Int)] = dstream.map { text =>  **val** arr: Array[String] = text.split(**" "**)  (arr(0), arr(1).toLong, 1)  }  **val** textWithEventTimeDstream: DataStream[(String, Long, Int)] = textWithTsDstream.assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[(String, Long, Int)](Time.*milliseconds*(1000)) {  **override def** extractTimestamp(element: (String, Long, Int)): Long = {   **return** element.\_2  }  })   **val** textKeyStream: KeyedStream[(String, Long, Int), Tuple] = textWithEventTimeDstream.keyBy(0)  textKeyStream.print(**"textkey:"**)   **val** windowStream: WindowedStream[(String, Long, Int), Tuple, TimeWindow] = textKeyStream.window(SlidingEventTimeWindows.*of*(Time.*seconds*(2),Time.*milliseconds*(500)))   **val** groupDstream: DataStream[mutable.HashSet[Long]] = windowStream.fold(**new** mutable.HashSet[Long]()) { **case** (set, (key, ts, count)) =>  set += ts  }   groupDstream.print(**"window::::"**).setParallelism(1)   env.execute()  } |

### 7.3.3 会话窗口（EventTimeSessionWindows）

**相邻两次数据的EventTime的时间差超过指定的时间间隔就会触发执行**。如果加入Watermark， 会在符合窗口触发的情况下进行延迟。到达延迟水位再进行窗口触发。

|  |
| --- |
| **def** main(args: Array[String]): Unit = {  *// 环境* **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  env.setParallelism(1)   **val** dstream: DataStream[String] = env.socketTextStream(**"hadoop1"**,7777)     **val** textWithTsDstream: DataStream[(String, Long, Int)] = dstream.map { text =>  **val** arr: Array[String] = text.split(**" "**)  (arr(0), arr(1).toLong, 1)  }  **val** textWithEventTimeDstream: DataStream[(String, Long, Int)] = textWithTsDstream.assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[(String, Long, Int)](Time.*milliseconds*(1000)) {  **override def** extractTimestamp(element: (String, Long, Int)): Long = {   **return** element.\_2  }  })   **val** textKeyStream: KeyedStream[(String, Long, Int), Tuple] = textWithEventTimeDstream.keyBy(0)  textKeyStream.print(**"textkey:"**)   **val** windowStream: WindowedStream[(String, Long, Int), Tuple, TimeWindow] = textKeyStream.window(EventTimeSessionWindows.*withGap*(Time.*milliseconds*(500)) )  windowStream.reduce((text1,text2)=>  ( text1.\_1,0L,text1.\_3+text2.\_3)  ) .map(\_.\_3).print(**"windows:::"**).setParallelism(1)   env.execute()   } |

# 第八章 Table API 与SQL

Table API是流处理和批处理通用的关系型API，Table API可以基于流输入或者批输入来运行而不需要进行任何修改。Table API是SQL语言的超集并专门为Apache Flink设计的，Table API是Scala 和Java语言集成式的API。与常规SQL语言中将查询指定为字符串不同，Table API查询是以Java或Scala中的语言嵌入样式来定义的，具有IDE支持如:自动完成和语法检测。

## 1 需要引入的pom依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table\_2.11</**artifactId**>  <**version**>1.7.0</**version**> </**dependency**> |

## 2 构造表环境

|  |
| --- |
| **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* **val** myKafkaConsumer: FlinkKafkaConsumer011[String] = MyKafkaUtil.*getConsumer*(**"GMALL\_STARTUP"**)  **val** dstream: DataStream[String] = env.addSource(myKafkaConsumer)   **val** tableEnv: StreamTableEnvironment = TableEnvironment.*getTableEnvironment*(env)   **val** startupLogDstream: DataStream[StartupLog] = dstream.map{ jsonString =>JSON.*parseObject*(jsonString,*classOf*[StartupLog]) }   **val** startupLogTable: Table = tableEnv.fromDataStream(startupLogDstream)   **val** table: Table = startupLogTable.select(**"mid,ch"**).filter(**"ch ='appstore'"**)   **val** midchDataStream: DataStream[(String, String)] = table.toAppendStream[(String,String)]   midchDataStream.print()  env.execute() } |

### 动态表

如果流中的数据类型是case class可以直接根据case class的结构生成table

|  |
| --- |
| tableEnv.fromDataStream(startupLogDstream) |

或者根据字段顺序单独命名

|  |
| --- |
| tableEnv.fromDataStream(startupLogDstream,’mid,’uid .......) |

最后的动态表可以转换为流进行输出

|  |
| --- |
| table.toAppendStream[(String,String)] |

### 字段

用一个单引放到字段前面 来标识字段名, 如 ‘name , ‘mid ,’amount 等

## 3 通过一个例子 了解TableAPI

|  |
| --- |
| *//每10秒中渠道为appstore的个数* **def** main(args: Array[String]): Unit = {  *//sparkcontext* **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment   //时间特性改为eventTime* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   **val** myKafkaConsumer: FlinkKafkaConsumer011[String] = MyKafkaUtil.*getConsumer*(**"GMALL\_STARTUP"**)  **val** dstream: DataStream[String] = env.addSource(myKafkaConsumer)   **val** startupLogDstream: DataStream[StartupLog] = dstream.map{ jsonString =>JSON.*parseObject*(jsonString,*classOf*[StartupLog]) }  *//告知watermark 和 eventTime如何提取* **val** startupLogWithEventTimeDStream: DataStream[StartupLog] = startupLogDstream.assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[StartupLog](Time.*seconds*(0L)) {  **override def** extractTimestamp(element: StartupLog): Long = {  element.ts  }  }).setParallelism(1)   *//SparkSession* **val** tableEnv: StreamTableEnvironment = TableEnvironment.*getTableEnvironment*(env)   *//把数据流转化成Table* **val** startupTable: Table = tableEnv.fromDataStream(startupLogWithEventTimeDStream , **'mid**,**'uid**,**'appid**,**'area**,**'os**,**'ch**,**'logType**,**'vs**,**'logDate**,**'logHour**,**'logHourMinute**,**'ts**.rowtime)   *//通过table api 进行操作  // 每10秒 统计一次各个渠道的个数 table api 解决  //1 groupby 2 要用 window 3 用eventtime来确定开窗时间* **val** resultTable: Table = startupTable.window(Tumble *over* 10000.millis on **'ts** as **'tt**).groupBy(**'ch**,**'tt** ).select( **'ch**, **'ch**.count)    *//把Table转化成数据流  //val appstoreDStream: DataStream[(String, String, Long)] = appstoreTable.toAppendStream[(String,String,Long)]* **val** resultDstream: DataStream[(Boolean, (String, Long))] = resultSQLTable.toRetractStream[(String,Long)]   resultDstream.filter(\_.\_1).print()   env.execute()  } |

### 关于group by

* 1. 如果使用 groupby table转换为流的时候只能用toRetractDstream

|  |
| --- |
| **val** rDstream: DataStream[(Boolean, (String, Long))] = table.toRetractStream[(String,Long)] |

* 1. toRetractDstream 得到的第一个boolean型字段标识 true就是最新的数据，false表示过期老数据

|  |
| --- |
| **val** rDstream: DataStream[(Boolean, (String, Long))] = table.toRetractStream[(String,Long)]  rDstream.filter(\_.\_1).print() |

* 1. 如果使用的api包括时间窗口，那么时间的字段必须，包含在group by中。

|  |
| --- |
| **val** table: Table = startupLogTable.filter(**"ch ='appstore'"**).window(Tumble *over* 10000.millis on **'ts** as **'tt**).groupBy(**'ch** ,**'tt**).select(**"ch,ch.count "**) |

### 关于时间窗口

1. 用到时间窗口，必须提前声明时间字段，如果是processTime直接在创建动态表时进行追加就可以

|  |
| --- |
| **val** startupLogTable: Table = tableEnv.fromDataStream(startupLogWithEtDstream,**'mid**,**'uid**,**'appid**,**'area**,**'os**,**'ch**,**'logType**,**'vs**,**'logDate**,**'logHour**,**'logHourMinute**,**'ts**.rowtime) |

1. 如果是EventTime要在创建动态表时声明

|  |
| --- |
| **val** startupLogTable: Table = tableEnv.fromDataStream(startupLogWithEtDstream,**'mid**,**'uid**,**'appid**,**'area**,**'os**,**'ch**,**'logType**,**'vs**,**'logDate**,**'logHour**,**'logHourMinute**,**'ps**.processtime) |

1. 滚动窗口可以使用Tumble *over* 10000.millis on

|  |
| --- |
| **val** table: Table = startupLogTable.filter(**"ch ='appstore'"**).window(Tumble *over* 10000.millis on **'ts** as **'tt**).groupBy(**'ch** ,**'tt**).select(**"ch,ch.count "**) |

## 4 SQL如何编写

|  |
| --- |
| **def** main(args: Array[String]): Unit = {  *//sparkcontext* **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment   //时间特性改为eventTime* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   **val** myKafkaConsumer: FlinkKafkaConsumer011[String] = MyKafkaUtil.*getConsumer*(**"GMALL\_STARTUP"**)  **val** dstream: DataStream[String] = env.addSource(myKafkaConsumer)   **val** startupLogDstream: DataStream[StartupLog] = dstream.map{ jsonString =>JSON.*parseObject*(jsonString,*classOf*[StartupLog]) }  *//告知watermark 和 eventTime如何提取* **val** startupLogWithEventTimeDStream: DataStream[StartupLog] = startupLogDstream.assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[StartupLog](Time.*seconds*(0L)) {  **override def** extractTimestamp(element: StartupLog): Long = {  element.ts  }  }).setParallelism(1)   *//SparkSession* **val** tableEnv: StreamTableEnvironment = TableEnvironment.*getTableEnvironment*(env)   *//把数据流转化成Table* **val** startupTable: Table = tableEnv.fromDataStream(startupLogWithEventTimeDStream , **'mid**,**'uid**,**'appid**,**'area**,**'os**,**'ch**,**'logType**,**'vs**,**'logDate**,**'logHour**,**'logHourMinute**,**'ts**.rowtime)   *//通过table api 进行操作  // 每10秒 统计一次各个渠道的个数 table api 解决  //1 groupby 2 要用 window 3 用eventtime来确定开窗时间* **val** resultTable: Table = startupTable.window(Tumble *over* 10000.millis on **'ts** as **'tt**).groupBy(**'ch**,**'tt** ).select( **'ch**, **'ch**.count)  *// 通过sql 进行操作* **val** resultSQLTable : Table = tableEnv.sqlQuery( **"select ch ,count(ch) from "**+startupTable+**" group by ch ,Tumble(ts,interval '10' SECOND )"**)   *//把Table转化成数据流  //val appstoreDStream: DataStream[(String, String, Long)] = appstoreTable.toAppendStream[(String,String,Long)]* **val** resultDstream: DataStream[(Boolean, (String, Long))] = resultSQLTable.toRetractStream[(String,Long)]   resultDstream.filter(\_.\_1).print()   env.execute()  } |