Flink的Watermark机制（基于Flink 1.11.0实现）

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，用于衡量Event Time进展的机制。watermark可以翻译为水位线。

Watermark的核心本质可以理解成一个延迟触发机制。
在 Flink 的窗口处理过程中，如果确定全部数据到达，就可以对 Window 的所有数据做 窗口计算操作（如汇总、分组等），如果数据没有全部到达，则继续等待该窗口中的数据全 部到达才开始处理。这种情况下就需要用到水位线（WaterMarks）机制，它能够衡量数据处 理进度（表达数据到达的完整性），保证事件数据（全部）到达 Flink 系统，或者在乱序及 延迟到达时，也能够像预期一样计算出正确并且连续的结果。当任何 Event 进入到 Flink 系统时，会根据当前最大事件时间产生 Watermarks 时间戳。

那么 Flink 是怎么计算 Watermak 的值呢?

Watermark =进入Flink 的最大的事件时间(mxtEventTime)-指定的延迟时间(t)

那么有 Watermark 的 Window 是怎么触发窗口函数的呢?
如果有窗口的停止时间等于或者小于 maxEventTime - t(当时的warkmark)，那么这个窗口被触发执行。

其核心处理流程如下图所示。

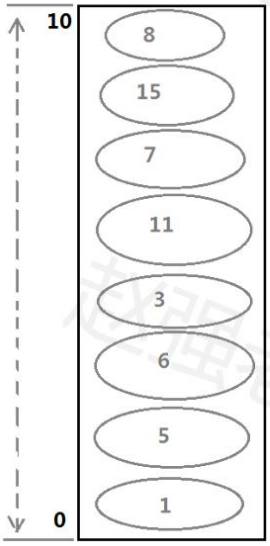
Flink Watermark的执行原理（本质就是延时计算）

（*）水位线的计算公式：Watermark =进入Flink 的最大的事件时间(mxtEventTime)-指定的延迟时间(t)

（*）触发窗口计算的时机：窗口的停止时间 <= 当前的Watermark

时间窗口：0~10

如左图所示：时间窗口：0~10（EventTime）。假设指定的延迟时间是3秒



到达的事件时间	Flink的最大事件时间	当前的Watermark	是否大于等于窗口的停止时间	是否触发计算	计算的事件
1	1	-2	否	否	
5	5	2	否	否	
6	6	3	否	否	
3	6	3	否	否	
11（不属于该窗口）	11	8	否	否	
7	11	8	否	否	
15（不属于该窗口）	15	12	是	是	1,5,6,3,7
8	15	12	是	是	1,5,6,3,7,8

如果数据元素的事件时间是有序的，Watermark 时间戳会随着数据元素的事件时间按顺 序生成，此时水位线的变化和事件时间保持一直（因为既然是有序的时间，就不需要设置延迟了，那么t就是 0。所以 watermark=maxtime-0 = maxtime），也就是理想状态下的水位 线。当 Watermark 时间大于 Windows 结束时间就会触发对 Windows 的数据计算，以此类推， 下一个 Window 也是一样。**这种情况其实是乱序数据的一种特殊情况。**

现实情况下数据元素往往并不是按照其产生顺序接入到 Flink 系统中进行处理，而频繁 出现乱序或迟到的情况，这种情况就需要使用 Watermarks 来应对。比如下图，设置延迟时间t为2。

在多并行度的情况下，Watermark 会有一个对齐机制，这个对齐机制会取所有 Channel 中最小的 Watermark。

```
sEnv.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

```
.assignTimestampsAndWatermarks(WatermarkStrategy.<StationLog>forBoundedOutOfOrderness(Duration.ofSeconds(3))
        .withTimestampAssigner(new SerializableTimestampAssigner<StationLog>() {

            @Override
            public long extractTimestamp(StationLog element, long recordTimestamp) {
                return element.getCallTime(); //指定EventTime对应的字段
            }
        })
)
```

指定延时处理的时间

指定EventTime（事件时间）对应的字段

注意：不管是数据是否有序，都可以使用上面的代码。有序的数据只是无序数据的一种特殊情况。

测试数据：基站的手机通话数据，如下：

```
基站ID  主叫方          被叫方          时长    呼叫发起时间
station1,18688822219,18684812319,10,1595158485855
station5,13488822219,13488822219,50,1595158490856
station5,13488822219,13488822219,50,1595158495856
station5,13488822219,13488822219,50,1595158500856
```

需求：按基站，每5秒统计通话时间最长的记录。

- StationLog用于封装基站数据

```
package watermark;

//station1,18688822219,18684812319,10,1595158485855
public class StationLog {
    private String stationID;    //基站ID
    private String from;         //呼叫放
    private String to;           //被叫方
    private long duration;       //通话的持续时间
    private long callTime;       //通话的呼叫时间
    public StationLog(String stationID, String from,
            String to, long duration,
            long callTime) {
        this.stationID = stationID;
        this.from = from;
        this.to = to;
        this.duration = duration;
        this.callTime = callTime;
    }
    public String getStationID() {
        return stationID;
    }
```

```
    public void setStationID(String stationID) {
        this.stationID = stationID;
    }
    public long getCallTime() {
        return callTime;
    }
    public void setCallTime(long callTime) {
        this.callTime = callTime;
    }
    public String getFrom() {
        return from;
    }
    public void setFrom(String from) {
        this.from = from;
    }

    public String getTo() {
        return to;
    }
    public void setTo(String to) {
        this.to = to;
    }
    public long getDuration() {
        return duration;
    }
    public void setDuration(long duration) {
        this.duration = duration;
    }
}
```

- 代码实现：WaterMarkDemo用于完成计算（注意：为了方便咱们测试设置任务的并行度为1）

```
package watermark;

import java.time.Duration;
import org.apache.flink.api.common.eventtime.SerializableTimestampAssigner;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.FilterFunction;
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.ReduceFunction;
import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.streaming.api.TimeCharacteristic;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.windowing.ProcessWindowFunction;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.api.windowing.windows.TimeWindow;
import org.apache.flink.util.Collector;

//每隔五秒，将过去是10秒内，通话时间最长的通话日志输出。
public  class WaterMarkDemo {
    public static void main(String[] args) throws Exception {
```

```
        //得到Flink流式处理的运行环境
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionE
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
        env.setParallelism(1);
        //设置周期性的产生水位线的时间间隔。当数据流很大的时候，如果每个事件都产生水位线，会
        env.getConfig().setAutoWatermarkInterval(100);//默认100毫秒

        //得到输入流
        DataStreamSource<String> stream = env.socketTextStream("bigdata111", 1234);
        stream.flatMap(new FlatMapFunction<String, StationLog>() {

            public void flatMap(String data, Collector<StationLog> output) throws Exception {
                String[] words = data.split(",");
                //              基站ID       from   to     通话时长
                output.collect(new StationLog(words[0], words[1],words[2], Long.
            }
        }).filter(new FilterFunction<StationLog>() {

            @Override
            public boolean filter(StationLog value) throws Exception {
                return value.getDuration() > 0?true:false;
            }
        }).assignTimestampsAndWatermarks(WatermarkStrategy.<StationLog>forBounde
                .withTimestampAssigner(new SerializableTimestampAssigner<Station
                    @Override
                    public long extractTimestamp(StationLog element, long recordTimestamp)
                        return element.getCallTime(); //指定EventTime对应的字段
                    }
                })
        ).keyBy(new KeySelector<StationLog, String>(){
            @Override
            public String getKey(StationLog value) throws Exception {
                return value.getStationID();  //按照基站分组
            }}
        ).timeWindow(Time.seconds(5)) //设置时间窗口
        .reduce(new MyReduceFunction(),new MyProcessWindows()).print();

        env.execute();
    }
}
//用于如何处理窗口中的数据，即：找到窗口内通话时间最长的记录。
class MyReduceFunction implements ReduceFunction<StationLog> {
    @Override
    public StationLog reduce(StationLog value1, StationLog value2) throws Exception {
        // 找到通话时间最长的通话记录
        return value1.getDuration() >= value2.getDuration() ? value1 : value2;
    }
}
//窗口处理完成后，输出的结果是什么
class MyProcessWindows extends ProcessWindowFunction<StationLog, String, String, TimeWir
    @Override
    public void process(String key, ProcessWindowFunction<StationLog, String, String, TimeWindow
      Iterable<StationLog> elements, Collector<String> out) throws Exception {
        StationLog maxLog = elements.iterator().next();
```

```
        StringBuffer sb = new StringBuffer();
        sb.append("窗口范围是:").append(context.window().getStart()).append("----").
        sb.append("基站ID：").append(maxLog.getStationID()).append("\t")
          .append("呼叫时间：").append(maxLog.getCallTime()).append("\t")
          .append("主叫号码：").append(maxLog.getFrom()).append("\t")
          .append("被叫号码：")  .append(maxLog.getTo()).append("\t")
          .append("通话时长：").append(maxLog.getDuration()).append("\n");
        out.collect(sb.toString());
    }
}
```