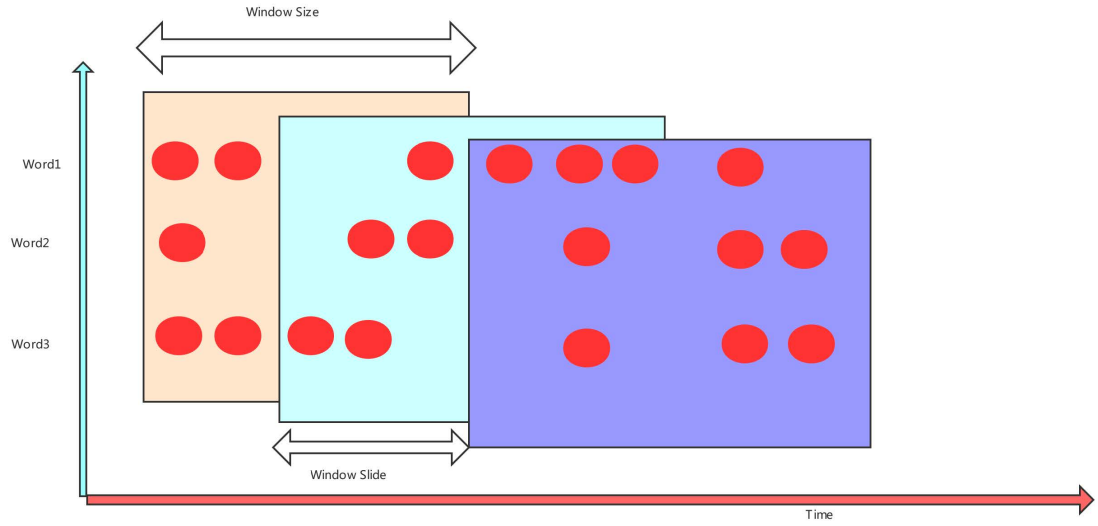
Apache Flink waterMark的机制

本课目标

- [x] 掌握WaterMark的原理
- [x] 掌握WaterMark的运用

1. 案例需求

- 需求描述：每隔5秒，计算最近10秒单词出现的次数



1.1 TimeWindow实现

```
//案例1 (job1)  -> timeWindow

public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<String> dataStream =
env.socketTextStream("192.168.134.130", 8888);
        dataStream.flatMap(new FlatMapFunction<String, Tuple2<String,Integer>>()
{
            @Override
            public void flatMap(String line, Collector<Tuple2<String, Integer>>
out) throws Exception {
                String[] fields = line.split(",");
                for(String word:fields){
                    out.collect(Tuple2.of(word,1));
                }
```

```
            }
        }).keyBy(0).timeWindow(Time.seconds(10),Time.seconds(5))
                .sum(1)
                .print().setParallelism(1);

        env.execute("TimeWindowWordCount");
    }
```

1.2 TimeWindowProcessTest

```
//案例2（job2） -> TimeWindowProcessTest
public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<String> dataStream =
env.socketTextStream("192.168.134.130", 9999);
        dataStream.flatMap(new FlatMapFunction<String, Tuple2<String,Integer>>()
{
            @Override
            public void flatMap(String line,
                                Collector<Tuple2<String, Integer>> out) throws
Exception {
                String[] fields = line.split(",");
                for(String word:fields){
                    out.collect(Tuple2.of(word,1));
                }

            }
        }).keyBy(0).timeWindow(Time.seconds(10),Time.seconds(5))
                .process(new MySumProcessWindowFunction()) // 相当于spark里面的
foreach
                .print().setParallelism(1);

        env.execute("TimeWindowWordCount");


    }

    /**
     * <IN, OUT, KEY, W extends Window>
     *     IN: 输入的数据类型
     *     OUT: 输出的数据类型
     *     KEY: key的数据类型
     *     W: 窗口的数据类型
     *
     */
    public  static class MySumProcessWindowFunction extends
ProcessWindowFunction<Tuple2<String,Integer>,Tuple2<String,Integer>,Tuple,TimeWin
dow>{
```

```
        FastDateFormat dataformat = FastDateFormat.getInstance("HH:mm:ss");

        //elements: 当前窗口里面的数据
        @Override
        public void process(Tuple key, Context context,
                            Iterable<Tuple2<String, Integer>> elements,
                            Collector<Tuple2<String, Integer>> out) throws
Exception {

            System.out.println("当前系统时
间: "+dataformat.format(System.currentTimeMillis()));
            System.out.println("窗口处理时
间: "+dataformat.format(context.currentProcessingTime()));
            System.out.println("窗口开始时
间: "+dataformat.format(context.window().getStart()));


            int sum=0;
            for (Tuple2<String,Integer> ele:elements){
                sum += 1;
            }

            out.collect(Tuple2.of(key.getField(0),sum));
            System.out.println("窗口结束时
间: "+dataformat.format(context.window().getEnd()));

 System.out.println("====================================================");
        }
    }
```

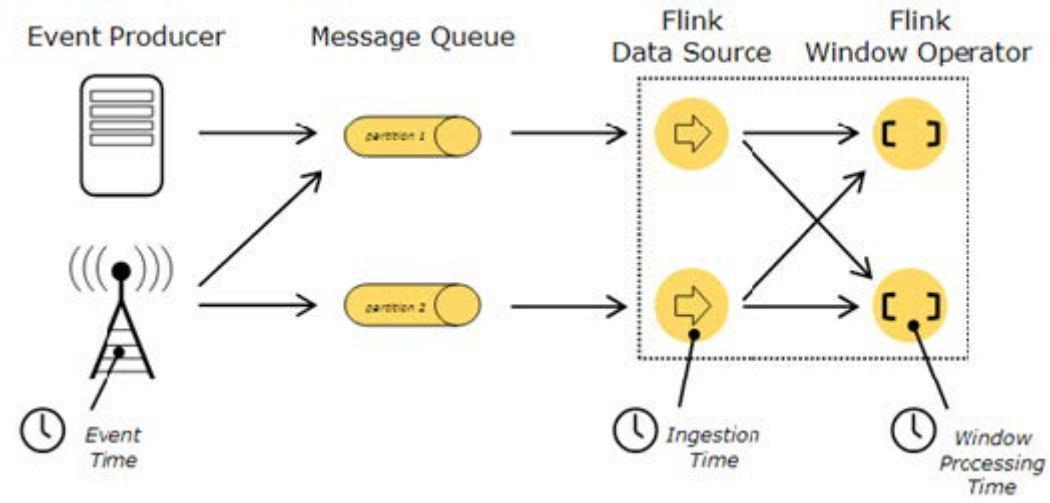
运行结果:

```
当前系统时间: 09:57:35
窗口处理时间: 09:57:35
窗口开始时间: 09:57:25
窗口结束时间: 09:57:35
(hive,1)

当前系统时间: 09:57:40
窗口处理时间: 09:57:40
窗口开始时间: 09:57:30
窗口结束时间: 09:57:40
(hive,2)

当前系统时间: 09:57:45
窗口处理时间: 09:57:45
窗口开始时间: 09:57:35
窗口结束时间: 09:57:45
(hive,1)

当前系统时间: 09:57:50
窗口处理时间: 09:57:50
```

```
窗口开始时间：09:57:40
窗口结束时间：09:57:50

根据每隔5秒执行最近10秒的数据，Flink划分的窗口
[00:00:00, 00:00:05) [00:00:05, 00:00:10)
[00:00:10, 00:00:15) [00:00:15, 00:00:20)
[00:00:20, 00:00:25) [00:00:25, 00:00:30)
[00:00:30, 00:00:35) [00:00:35, 00:00:40)
[00:00:40, 00:00:45) [00:00:45, 00:00:50)
[00:00:50, 00:00:55) [00:00:55, 00:01:00)
[00:01:00, 00:01:05) ...
```

2. Time的种类

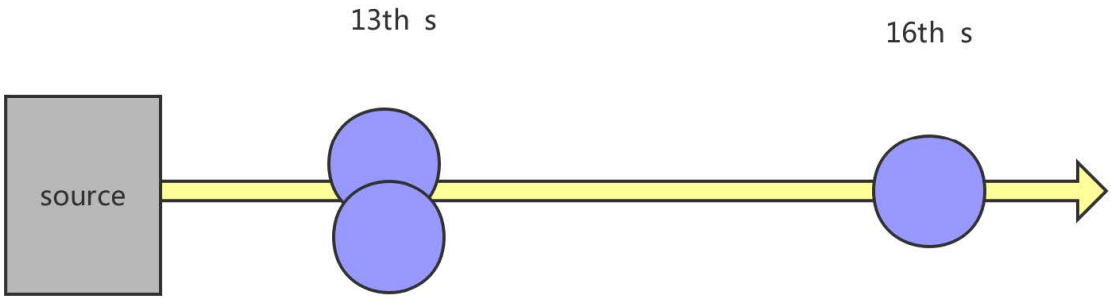
- 针对stream数据中的时间，可以分为以下三种：
  - Event Time：事件产生的时间，它通常由事件中的时间戳描述
  - Ingestion time：事件进入Flink的时间
  - Processing Time：事件被处理时当前系统的时间（默认）




```
案例演示：原始日志如下
2021-10-10 13:00:01,134 INFO executor.Executor: Finished task in state 0.0

2021-10-10 13:00:01,134 是Event time
2021-10-10 20:00:03,102 是Ingestion time
2021-10-10 20:00:05,100 是Processing time

思考：
如果我们想要统计每分钟内接口调用失败的错误日志个数，使用哪个时间才有意义？
```

3.Process Time Window（有序）

- 需求：每隔5秒计算最近10秒的单词出现的次数
- 自定义source，模拟：第 13 秒的时候连续发送 2 个事件，第 16 秒的时候再发送 1 个事件

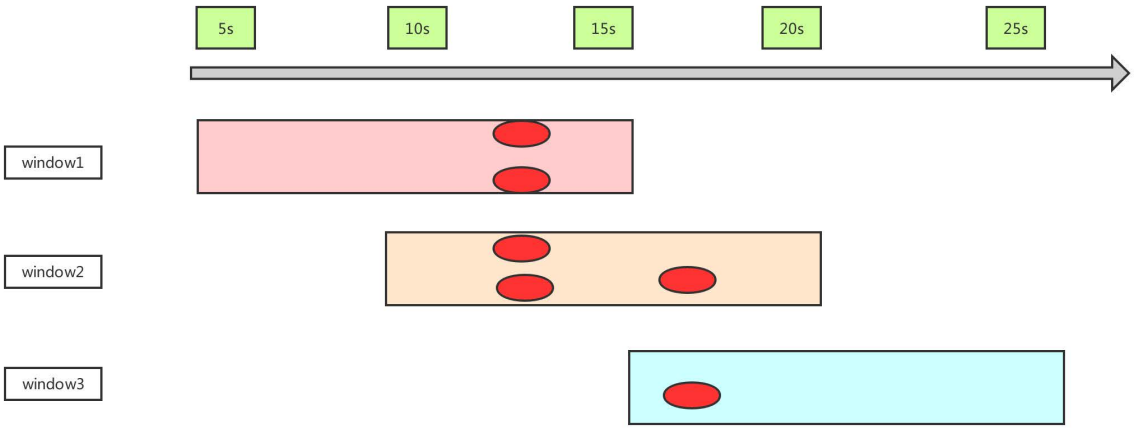



```
//案例3 -> job3
public static void main(String[] args) throws  Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<String> dataStream = env.addSource(new TestSource());
        dataStream.flatMap(new FlatMapFunction<String, Tuple2<String,Integer>>()
{
            @Override
            public void flatMap(String line, Collector<Tuple2<String, Integer>>
out) throws Exception {
                String[] fields = line.split(",");
                for (String word:fields){
                    out.collect(Tuple2.of(word,1));
                }
            }
        })
                .keyBy(0)
                .timeWindow(Time.seconds(10),Time.seconds(5))
                .process(new SumProcessFunction()).print().setParallelism(1);
        env.execute("WindowWordCountSortTest");
    }

    public static class TestSource implements SourceFunction<String> {
        FastDateFormat dateformat =  FastDateFormat.getInstance("HH:mm:ss");

        @Override
        public void run(SourceContext<String> cxt) throws Exception {

            String currTime = String.valueOf(System.currentTimeMillis());

            System.out.println(currTime);

            //这个操作是我为了保证是10secondes的倍数。
            while(Integer.valueOf(currTime.substring(currTime.length() - 4)) >
10){
                currTime=String.valueOf(System.currentTimeMillis());
                continue;
            }
```

```
            System.out.println("开始发送事件的时
间: "+dateformat.format(System.currentTimeMillis()));
            TimeUnit.SECONDS.sleep(3);
            cxt.collect("flink");
            cxt.collect("flink");

            TimeUnit.SECONDS.sleep(3);
            cxt.collect("flink");

            //程序等待
            TimeUnit.SECONDS.sleep(500000);
        }

        @Override
        public void cancel() {

        }
    }

    /**
     * IN
     * OUT
     * KEY -> is Tuple
     * W extends Window
     *
     */
    public static class  SumProcessFunction extends
ProcessWindowFunction<Tuple2<String,Integer>,Tuple2<String,Integer>,Tuple,TimeWin
dow>{

        FastDateFormat dataformat=FastDateFormat.getInstance("HH:mm:ss");

        @Override
        public void process(Tuple tuple, Context context,
                            Iterable<Tuple2<String, Integer>> allElements,
                            Collector<Tuple2<String, Integer>> out) {
           int count = 0;
           for (Tuple2<String,Integer> e:allElements){
               count++;
           }
            out.collect(Tuple2.of(tuple.getField(0),count));
        }
    }
```

输出结果:

```
(flink 2)
(flink 3)
(flink 1)
```



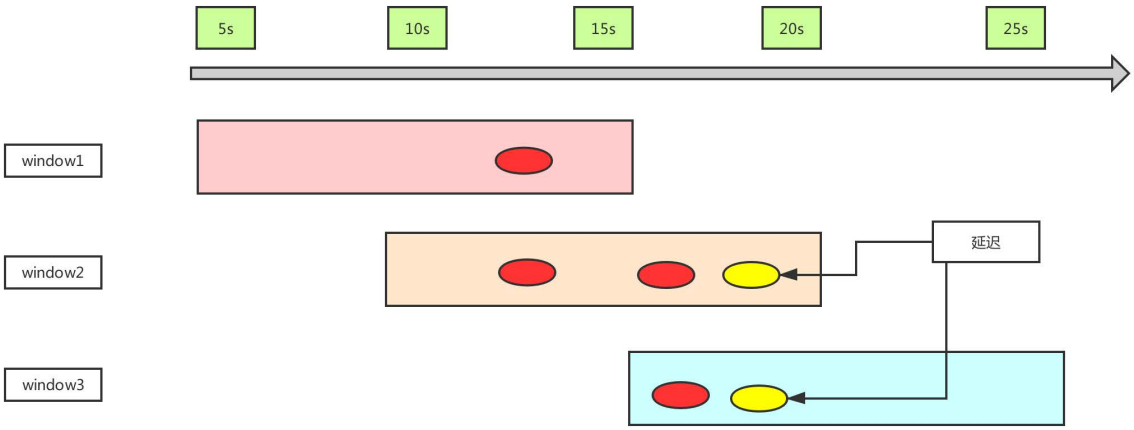
4. Process Time Window（无序）

- 自定义source模拟：第 13 秒的时候连续发送2个事件，第一个事件在第13秒的时候发送出去了，第二个事件在13的时候产生因为网络延迟等原因，在19秒的时候才发送出去，第三个事件16 秒的时候发送了出去。

```
//案例4 -> job4
public static void main(String[] args) throws  Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        DataStreamSource<String> dataStream = env.addSource(new TestSource());
        dataStream.flatMap(new FlatMapFunction<String, Tuple2<String,Integer>>()
{
            @Override
            public void flatMap(String line,
                                Collector<Tuple2<String, Integer>> out) throws
Exception {
                String[] fields = line.split(",");
                for (String word:fields){
                    out.collect(Tuple2.of(word,1));
                }
            }
        })
                .keyBy(0)
                .timeWindow(Time.seconds(10),Time.seconds(5))
                .process(new SumProcessFunction()).print().setParallelism(1);


        env.execute("WindowBySort2");

    }

    public static class TestSource implements
            SourceFunction<String>{
        FastDateFormat dateformat =  FastDateFormat.getInstance("HH:mm:ss");
        @Override
```

```
        public void run(SourceContext<String> cxt) throws Exception {
            String currTime = String.valueOf(System.currentTimeMillis());
            while(Integer.valueOf(currTime.substring(currTime.length() - 4)) >
100){
                currTime=String.valueOf(System.currentTimeMillis());
                continue;
            }
            System.out.println("开始发送事件的时
间: "+dateformat.format(System.currentTimeMillis()));

            TimeUnit.SECONDS.sleep(3);
            //实际上我们的数据是在13秒的时候生成的，只是19的时候被发送出去。
            String event = "flink";
            //13s 第一个事件发送
            cxt.collect(event);

            //16s 第三个事件发送
            TimeUnit.SECONDS.sleep(3);
            cxt.collect("flink");

            //19s 第三个事件发送
            TimeUnit.SECONDS.sleep(3);
            cxt.collect(event);
            TimeUnit.SECONDS.sleep(500000);
        }

        @Override
        public void cancel() {

        }
    }

    /**
     * IN
     * OUT
     * KEY
     * W extends Window
     *
     */
    public static class  SumProcessFunction
            extends
ProcessWindowFunction<Tuple2<String,Integer>,Tuple2<String,Integer>,Tuple,TimeWin
dow>{

        FastDateFormat dataformat=FastDateFormat.getInstance("HH:mm:ss");
        @Override
        public void process(Tuple tuple, Context context,
                            Iterable<Tuple2<String, Integer>> allElements,
                            Collector<Tuple2<String, Integer>> out) {
           int count=0;
           for (Tuple2<String,Integer> e:allElements){
               count++;
```

```
            }
            out.collect(Tuple2.of(tuple.getField(0),count));
        }
    }

结果:
(flink,1)
(flink,3)
(flink,2)
```



- **5. 使用Event Time处理无序**

```
//案例5 -> job5

public static void main(String[] args) throws  Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        //步骤一：设置时间类型，默认的是Processtime
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

        DataStreamSource<String> dataStream = env.addSource(new TestSource());
        dataStream.map(new MapFunction<String, Tuple2<String,Long>>() {
            @Override
            public Tuple2<String, Long> map(String line) throws Exception {
                String[] fields = line.split(",");
                    //key/value
                return new Tuple2<>(fields[0],Long.valueOf(fields[1]));
            }
            //作用：指定时间字段
        }).assignTimestampsAndWatermarks(new EventTimeExtractor())
                .keyBy(0)
                .timeWindow(Time.seconds(10),Time.seconds(5))
                .process(new SumProcessFunction()).print().setParallelism(1);


        env.execute("WindowWordCountAndTime");
```

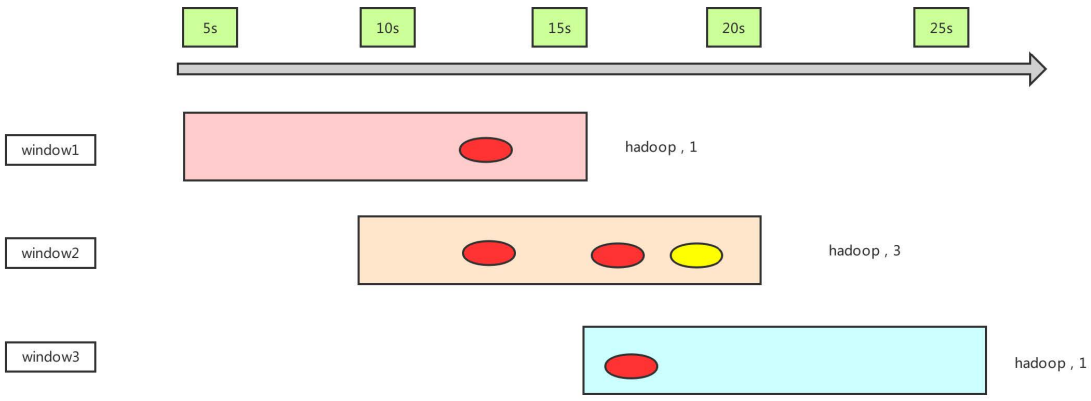
```
    }

    public static class TestSource implements SourceFunction<String>{
        FastDateFormat dateformat =  FastDateFormat.getInstance("HH:mm:ss");
        @Override
        public void run(SourceContext<String> cxt) throws Exception {
            String currTime = String.valueOf(System.currentTimeMillis());
            while(Integer.valueOf(currTime.substring(currTime.length() - 4)) >
100){
                currTime=String.valueOf(System.currentTimeMillis());
                continue;
            }
            System.out.println("开始发送事件的时
间: "+dateformat.format(System.currentTimeMillis()));
            TimeUnit.SECONDS.sleep(3);
            //13s 第1个事件
            String event="flink,"+System.currentTimeMillis();//时间
            cxt.collect(event);

            TimeUnit.SECONDS.sleep(3);//16s 第2个事件
            cxt.collect("flink,"+System.currentTimeMillis());

            TimeUnit.SECONDS.sleep(3);
            //19s 第3个事件
            cxt.collect(event);

            TimeUnit.SECONDS.sleep(3000);




        }

        @Override
        public void cancel() {

        }
    }


    private static class EventTimeExtractor
            implements AssignerWithPeriodicWatermarks<Tuple2<String,Long>>{
        @Nullable
        @Override
        public Watermark getCurrentWatermark() {
            return new Watermark(System.currentTimeMillis());
        }

        //指定时间
        @Override
        public long extractTimestamp(Tuple2<String, Long> element, long l) {
```

```
            return element.f1;
        }
    }

    /**
     * IN
     * OUT
     * KEY
     * W extends Window
     */
    public static class  SumProcessFunction
            extends
ProcessWindowFunction<Tuple2<String,Long>,Tuple2<String,Integer>,Tuple,TimeWindow
>{

        FastDateFormat dataformat=FastDateFormat.getInstance("HH:mm:ss");
        @Override
        public void process(Tuple tuple, Context context,
                            Iterable<Tuple2<String, Long>> allElements,
                            Collector<Tuple2<String, Integer>> out) {
           int count=0;
           for (Tuple2<String,Long> e:allElements){
               count++;
           }
            out.collect(Tuple2.of(tuple.getField(0),count));

        }
    }
```

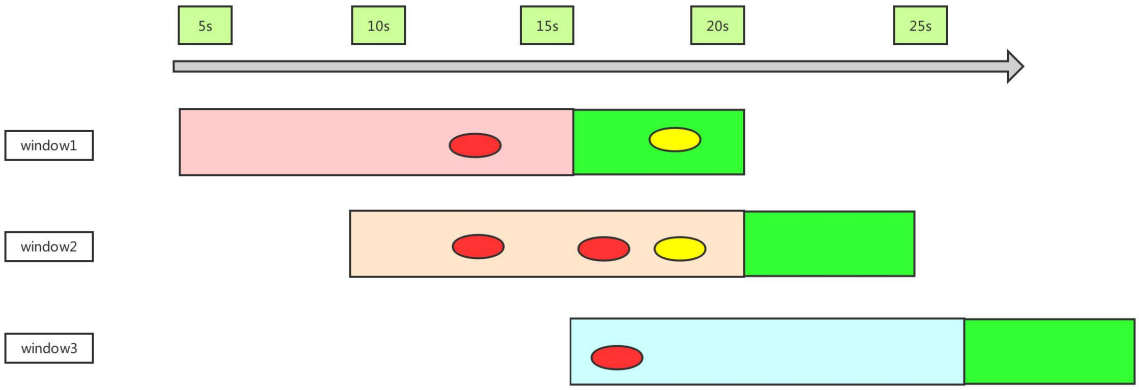
运行结果：但是存在数据丢失的问题

```
(flink,1)
(flink,3)
(flink,1)
```

PS：现在我们第三个window的结果已经计算准确了，但是我们还是没有彻底的解决问题。接下来就需要我们使用WaterMark机制来解决了。



- **6. 使用WaterMark机制解决无序**




```
//案例6 -> job6

public static void main(String[] args) throws  Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        //步骤一：设置时间类型，默认的是Processtime
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

        DataStreamSource<String> dataStream = env.addSource(new TestSource());
        dataStream.map(new MapFunction<String, Tuple2<String,Long>>() {
            @Override
            public Tuple2<String, Long> map(String line) throws Exception {
                String[] fields = line.split(",");
                return new Tuple2<>(fields[0],Long.valueOf(fields[1]));
            }
        }).assignTimestampsAndWatermarks(new EventTimeExtractor())
                .keyBy(0)
                .timeWindow(Time.seconds(10),Time.seconds(5))
                .process(new SumProcessFunction()).print().setParallelism(1);


        env.execute("WindowWordCountAndTime");

    }

    public static class TestSource implements SourceFunction<String>{
        FastDateFormat dateformat =  FastDateFormat.getInstance("HH:mm:ss");
        @Override
        public void run(SourceContext<String> cxt) throws Exception {
            String currTime = String.valueOf(System.currentTimeMillis());
            while(Integer.valueOf(currTime.substring(currTime.length() - 4)) >
100){
                currTime=String.valueOf(System.currentTimeMillis());
                continue;
            }
            System.out.println("开始发送事件的时
间: "+dateformat.format(System.currentTimeMillis()));
            TimeUnit.SECONDS.sleep(3);
```

```
            String event="flink,"+System.currentTimeMillis();
            cxt.collect(event);

            TimeUnit.SECONDS.sleep(3);
            cxt.collect("flink,"+System.currentTimeMillis());

            TimeUnit.SECONDS.sleep(3);
            cxt.collect(event);
            TimeUnit.SECONDS.sleep(500000);
        }

        @Override
        public void cancel() {

        }
    }


    private static class EventTimeExtractor implements
AssignerWithPeriodicWatermarks<Tuple2<String,Long>>{
        //设置 5s的延迟（乱序）
        @Nullable
        @Override
        public Watermark getCurrentWatermark() {
//            System.out.println("water
maker:....."+System.currentTimeMillis());
            return new Watermark(System.currentTimeMillis() - 5000);
        }

        @Override
        public long extractTimestamp(Tuple2<String, Long> element, long l) {
            return element.f1;
        }
    }

    /**
     * IN
     * OUT
     * KEY
     * W extends Window
     *
     */
    public static class  SumProcessFunction
            extends
ProcessWindowFunction<Tuple2<String,Long>,Tuple2<String,Integer>,Tuple,TimeWindow
>{

        FastDateFormat dataformat=FastDateFormat.getInstance("HH:mm:ss");
        @Override
        public void process(Tuple tuple, Context context,
                            Iterable<Tuple2<String, Long>> allElements,
                            Collector<Tuple2<String, Integer>> out) {
```

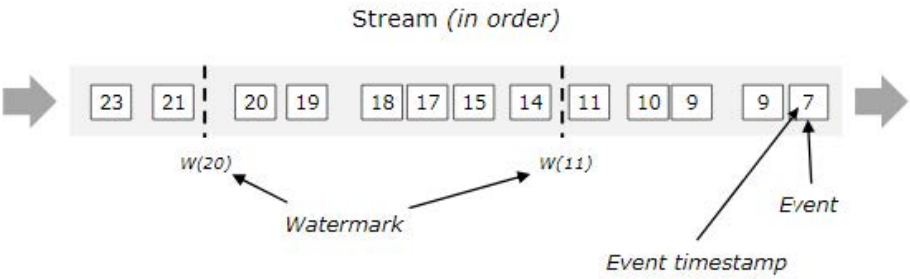
```
            int count=0;
            for (Tuple2<String,Long> e:allElements){
                count++;
            }
             out.collect(Tuple2.of(tuple.getField(0),count));

        }
    }
```

7. WaterMark机制

> 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watermark，watermark是用于处理乱序事件的。watermark可以翻译为水位线。
>
> WaterMark：是周期性的运行的。在一个窗口周期当中

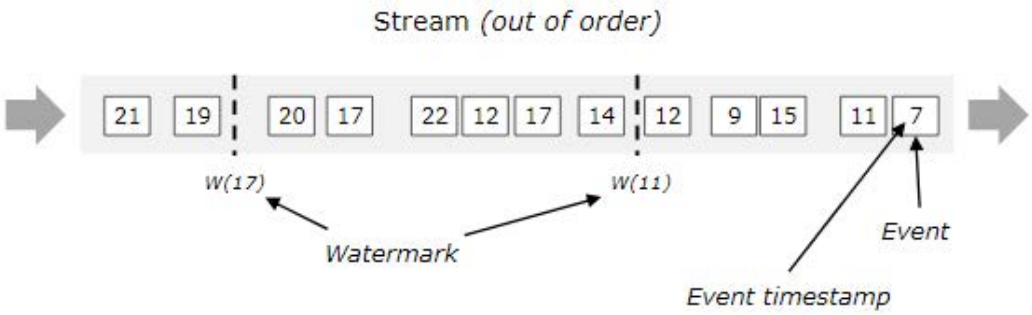
7.1 有序的流的watermarks

- 到达w(11)，一定会触发右边的window执行。
- 到达w(20)，一定会触发右边的window执行。
- 如果数据是有序的话，watermark意义不大



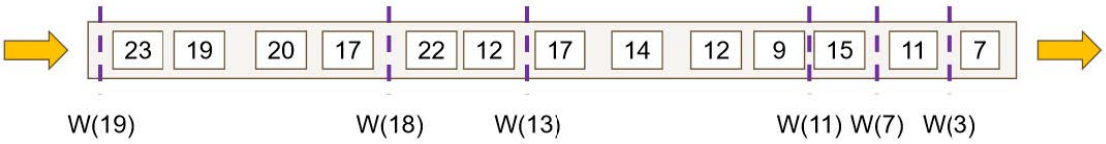
7.2 无序的流的watermarks

- w(11)：小于11的数据开始进行window计算，12，15不参与计算。
- w(17)：小于17的数据开始进行window计算



7.3 案例需求

- 得到并打印每隔 3 秒钟统计前 3 秒内的相同的 key 的所有的事件
- 9，12，14，都比15小，所以他们的watermark都是11，因为当前事件的eventTime取的是区间的最大值.




```
案例7 -> job7
基于waterMark演示有序的数据计算
```

7.4 计算window的触发时间

- window触发的时间
  - watermark 时间 >= window_end_time
  - 在 [window_start_time, window_end_time) 区间中有数据存在，注意是左闭右开的区间，而且是以 event time 来计算的
  - waterMark出发时间 19:34:24，这个时候刚好运算[19:34:21-19:34:24)产生的数据，也就是运行10秒前产生的数据，窗口为3秒。也就是根据当前的时间运行过去10秒之前的watermark。

Key	EventTime	CurrentMaxTimestamp	CurrentWaterMark	window_start_time	window_end_time
hadoop	19:34:22	19:34:22	19:34:12		
hadoop	19:34:26	19:34:26	19:34:16		
hadoop	19:34:32	19:34:32	19:34:22		
hadoop	19:34:33	19:34:33	19:34:23		
hadoop	19:34:34	19:34:34	19:34:24	[19:34:21	19:34:24)
hadoop	19:34:36	19:34:36	19:34:26		
hadoop	19:34:37	19:34:37	19:34:27	[19:34:24	19:34:27)

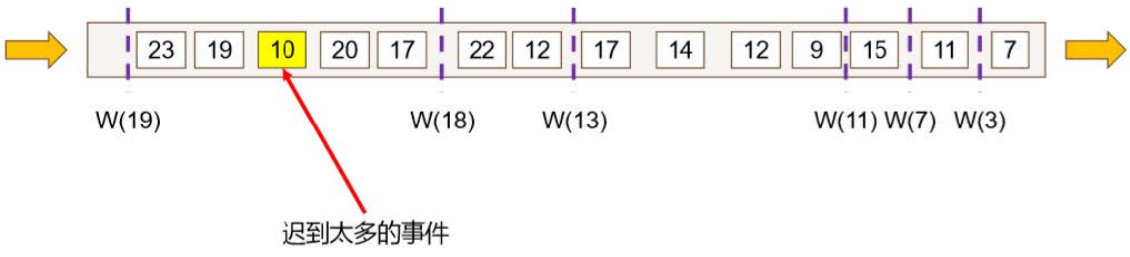
```
[00:00:00,00:00:03)
[00:00:03,00:00:06)
[00:00:06,00:00:09)
[00:00:09,00:00:12)
[00:00:12,00:00:15)
[00:00:15,00:00:18)
[00:00:18,00:00:21)
[00:00:21,00:00:24)
[00:00:24,00:00:27)
[00:00:27,00:00:30)
[00:00:30,00:00:33)
[00:00:33,00:00:36)
[00:00:36,00:00:39)
[00:00:39,00:00:42)
[00:00:42,00:00:45)
[00:00:45,00:00:48)
[00:00:48,00:00:51)
[00:00:51,00:00:54)
[00:00:54,00:00:57)
[00:00:57,00:01:00)
...
```

7.5 WaterMark+Window 处理乱序时间

```
案例7 -> job7
基于waterMark实现无序的数据计算
000001,1461756879000
000001,1461756871000
000001,1461756883000
```

7.6 事件迟到

- 丢弃，这个是默认的处理方式
- allowedLateness 指定允许数据延迟的时间
- sideOutputLateData 收集迟到的数据



7.6.1 丢弃

- 不推荐使用该方式

```
案例8 -> job8
重启程序，做测试。
输入数据:
000001,1461756870000
000001,1461756883000

000001,1461756870000
000001,1461756871000
000001,1461756872000

发现迟到太多数据就会被丢弃
```

7.6.2 再次推迟迟到的时间（了解）

- 当我们设置允许迟到 2 秒的事件，第一次 window 触发的条件是 watermark >=window_end_time
- 第二次(或者多次)触发的条件是 watermark < window_end_time + allowedLateness

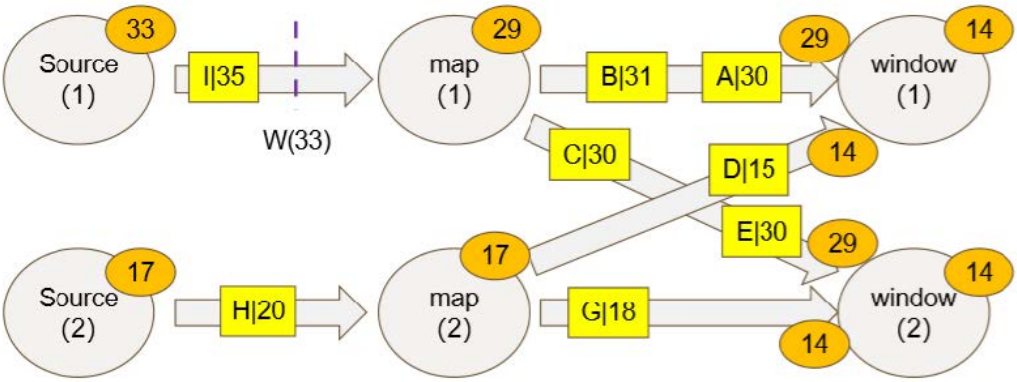
```
案例9 -> job9
assignTimestampsAndWatermarks(new EventTimeExtractor())
                .keyBy(0)
                .timeWindow(Time.seconds(3))
                .allowedLateness(Time.seconds(2)) // 允许事件迟到 2 秒
```

7.6.3 收集迟到的数据（推荐）

```
案例9 -> job9
OutputTag<Tuple2<String, Long>> outputTag = new OutputTag<Tuple2<String, Long>>
("late-date"){};
.sideOutputLateData(outputTag) //保留迟到太多的数据
result.getSideOutput(outputTag).map(new MapFunction<Tuple2<String,Long>, String>
() {
            @Override
            public String map(Tuple2<String, Long> stringLongTuple2) throws
Exception {
                return "迟到数据: "+stringLongTuple2.toString();
            }
        }).print();
```

8. 多并行度下的WaterMark

- 一个window可能会接受到多个waterMark，我们以最小的为准。




```
案例演示：job10

000001,1461756870000
000001,1461756883000
000001,1461756888000

当前线程ID：55event = (000001,1461756883000)|19:34:43|19:34:43|19:34:33
当前线程ID：56event = (000001,1461756870000)|19:34:30|19:34:30|19:34:20
当前线程ID：56event = (000001,1461756888000)|19:34:48|19:34:48|19:34:38
处理时间：19:31:25
window start time : 19:34:30
2> [(000001,1461756870000)|19:34:30]
window end time : 19:34:33

ID为56的线程有两个WaterMark：20,38
那么38这个会替代20，所以ID为56的线程的WaterMark是38
然后ID为55的线程的WaterMark是33，而ID为56是WaterMark是38，会在里面求一个小的值作为
waterMark，就是33，这个时候会触发Window为30-33的窗口，那这个窗口里面就有
(000001,1461756870000)这条数据。
```