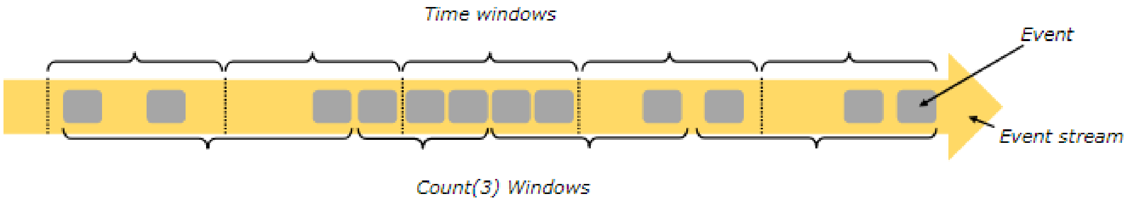
Apache Flink高阶-window开发

Objective(本课目标)

- ☑ 掌握window的类型
- ☑ 掌握window的常用方法

1. window概述

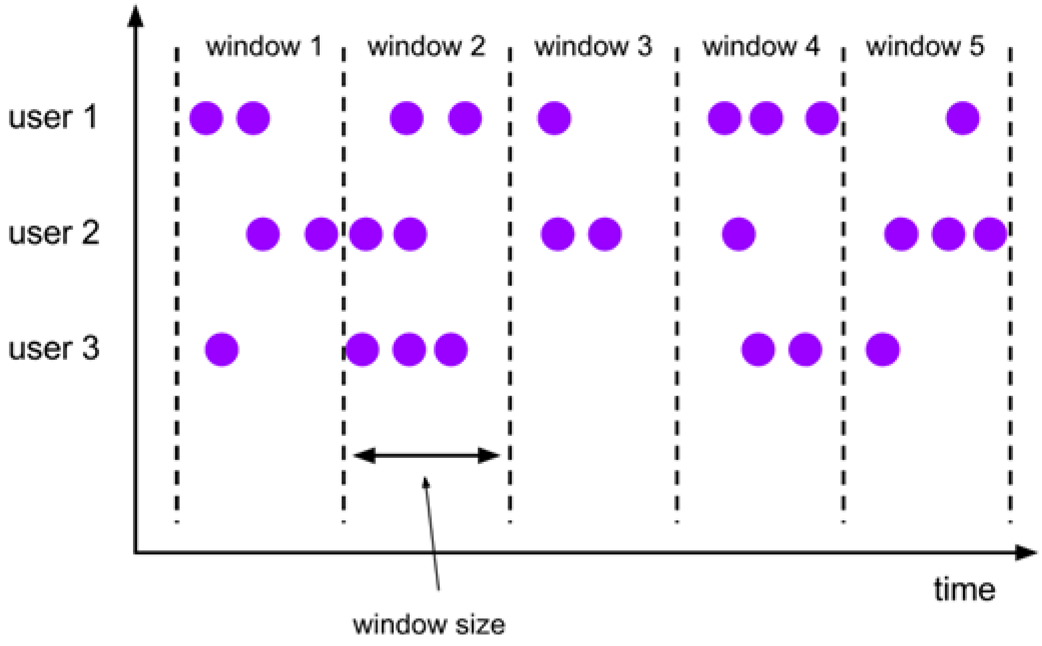
> 聚合事件（比如计数、求和）在流上的工作方式与批处理不同。比如，对流中的所有元素进行计数是不可能的，因为通常流是无限的（无界的）。所以，流上的聚合需要由 window 来划定范围，比如 “计算过去的5分钟” ，或者 “最后100个元素的和” 。window是一种可以把无限数据切割为有限数据块的手段。
>
> 窗口可以是时间驱动的 【Time Window】 （比如：每30秒）或者 数据驱动的【Count Window】 （比如：每100个元素）



2. Window类型

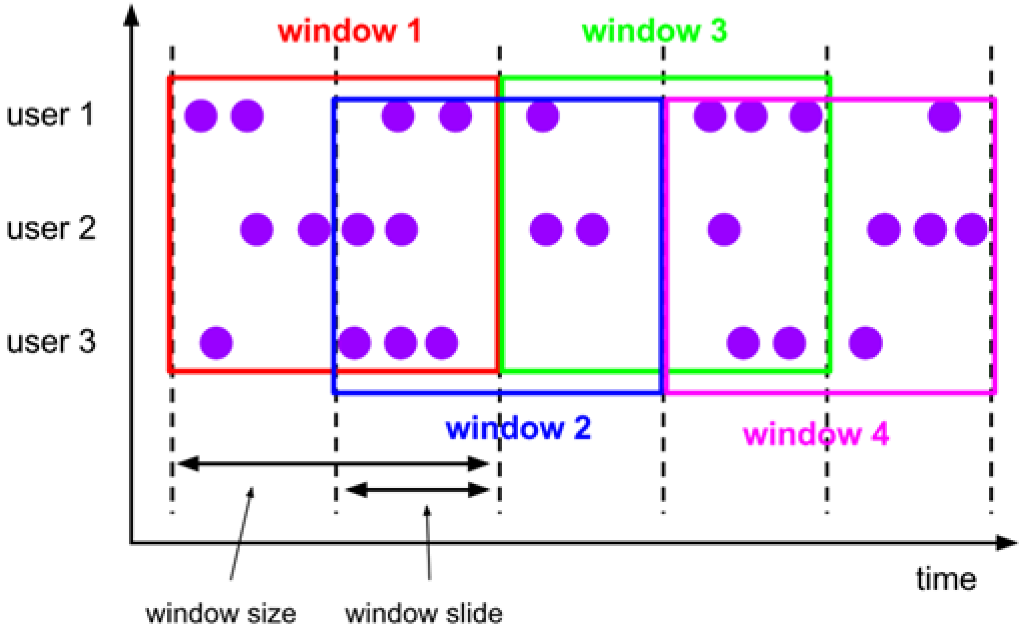
- 窗口通常被区分为不同的类型:
  - tumbling windows：滚动窗口 【没有重叠】
  - sliding windows：滑动窗口 【有重叠】
  - session windows：会话窗口
  - global windows: 没有窗口

2.1 tumblingwindows：滚动窗口【没有重叠】



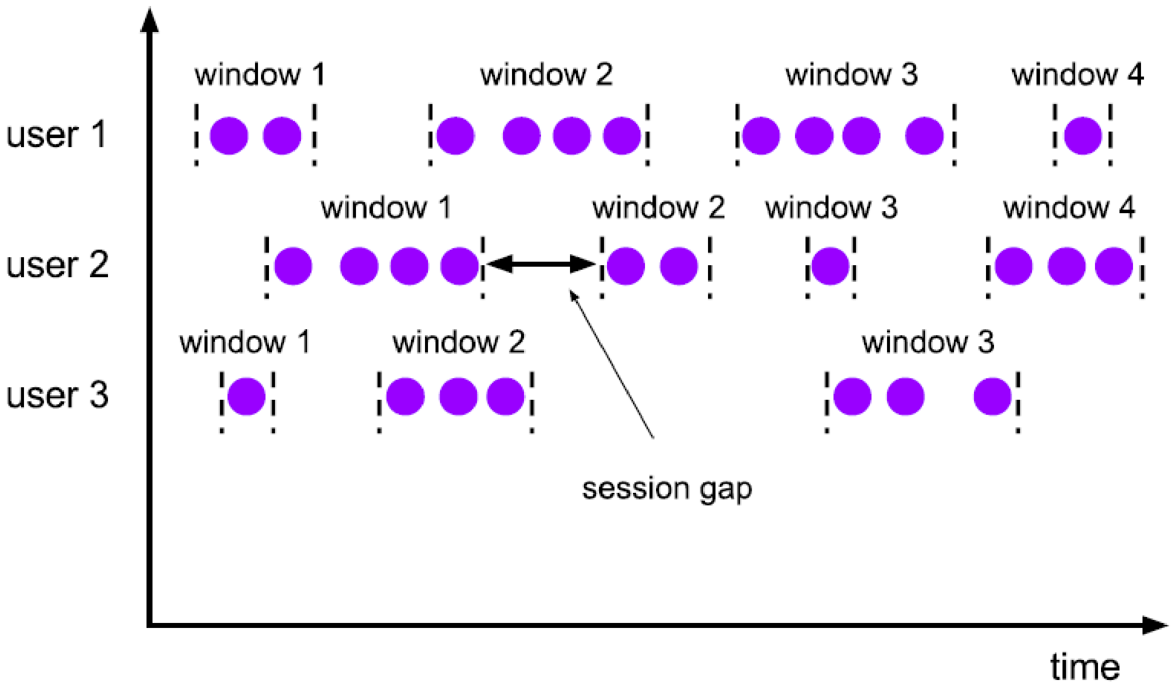
2.2 slidingwindows：滑动窗口 【有重叠】

- SparkStreaming就是滑动窗口



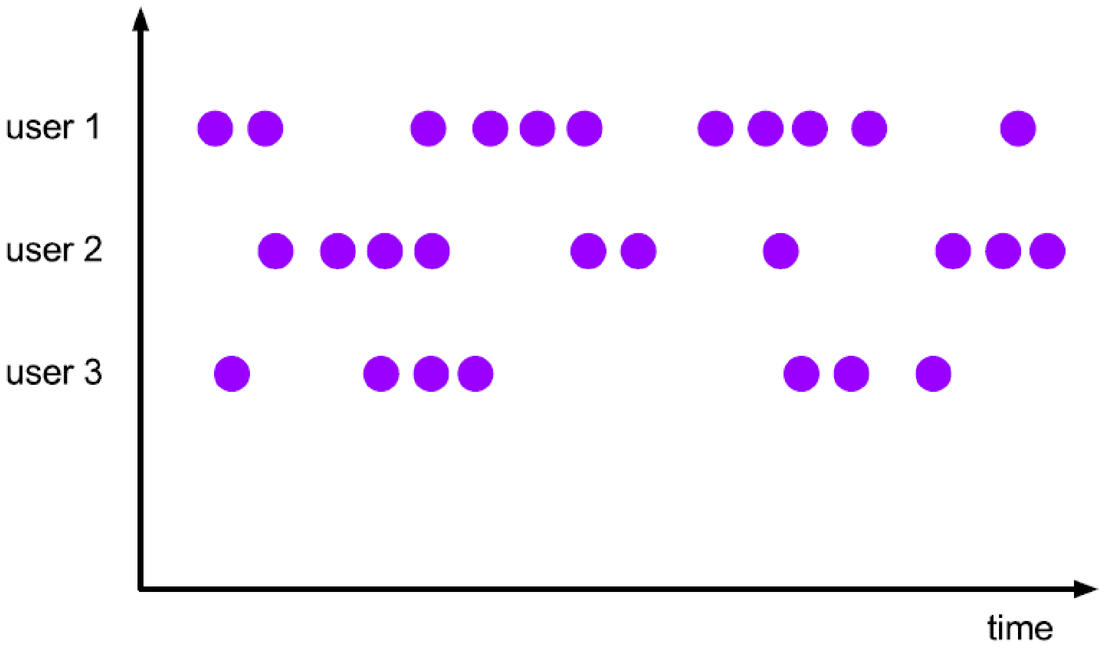
2.3 session windows

- 需求：实时计算每个单词出现的次数，如果一个单词过了5秒就没出现过了，那么就输出这个单词。
- 使用方式：只能基于时间触发，.window(ProcessingTimeSessionWindows.withGap(Time.seconds(5)))



2.4 global windows

- 无界窗口，也就是没有，这种情况就可以自定义窗口



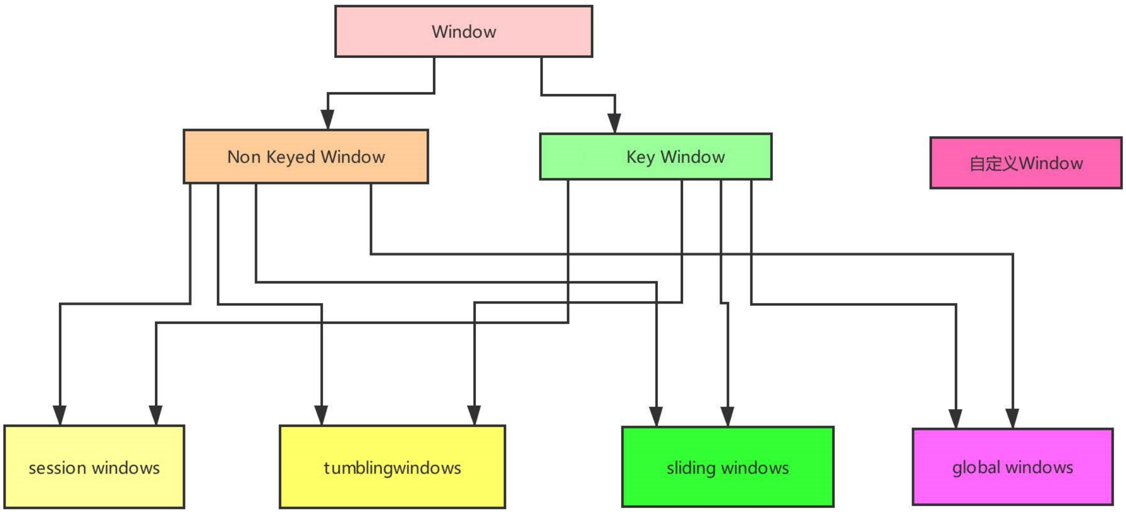
3. Window类型总结

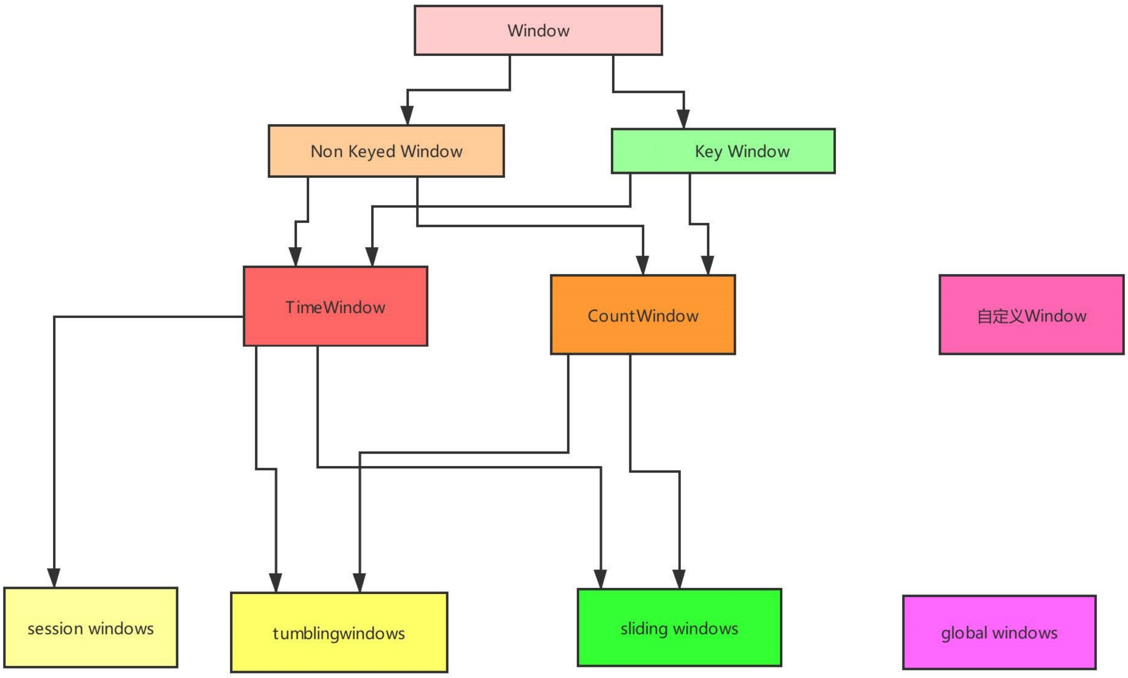
3.1 Keyed Window 和 Non Keyed Window

- 前面是keyBy操作后面执行window操作就是Keyed Window
- 前面没有keyBy操作后面执行window操作就是 Non Keyed Window
- 经过keyBy调用的是timeWindow，没有经过keyBy调用的是timeWindowAll，本质上是一样的。

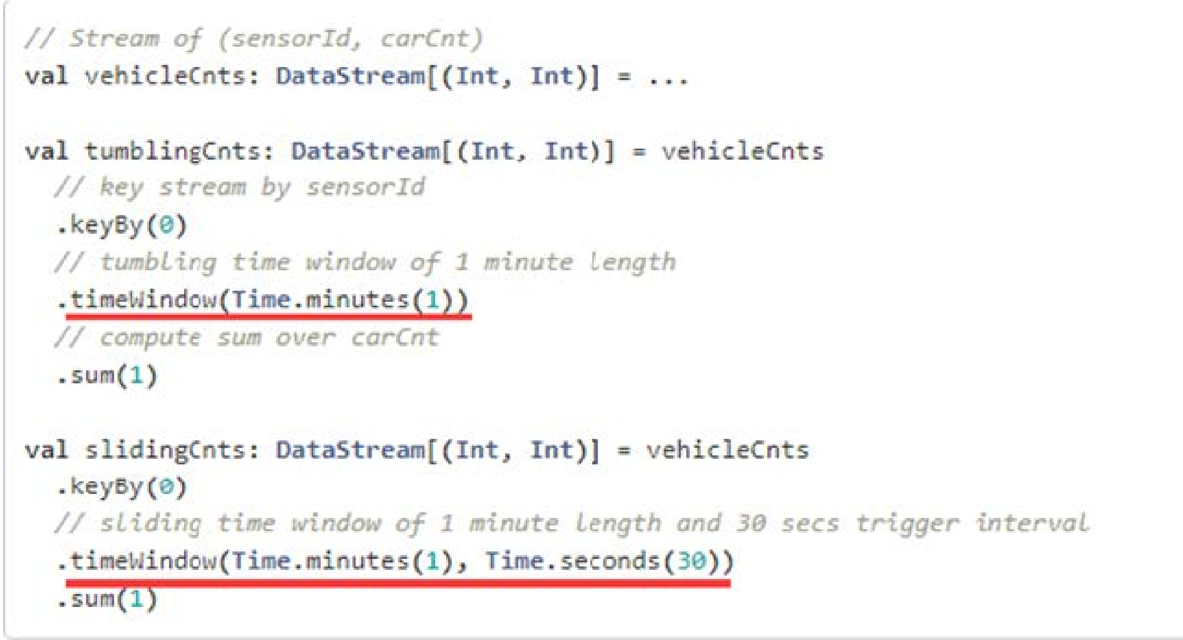
```
案例1 -> job1
/**
* Non keyed Stream
```

```
*/
public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> dataStream =
env.socketTextStream("192.168.134.130", 9999);
        SingleOutputStreamOperator<Tuple2<String, Integer>> streamResult =
dataStream.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public void flatMap(String line, Collector<Tuple2<String, Integer>>
collector) throws Exception {
                String[] fields = line.split(",");
                for (String word : fields) {
                    collector.collect(Tuple2.of(word, 1));
                }
            }
        });
        //Non keyed Stream
        AllWindowedStream<Tuple2<String, Integer>, TimeWindow> nonkeyedStream =
streamResult.timeWindowAll(Time.seconds(3));
        nonkeyedStream.sum(1).print();
        env.execute("WindowType");
    }



案例2 -> job2
// keyed Stream
public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> dataStream =
env.socketTextStream("192.168.134.130", 9999);

        SingleOutputStreamOperator<Tuple2<String, Integer>> stream =
dataStream.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public void flatMap(String line, Collector<Tuple2<String, Integer>>
collector) throws Exception {
                String[] fields = line.split(",");
                for (String word : fields) {
                    System.out.println(word);
                    collector.collect(Tuple2.of(word, 1));
                }
            }
        });
        //Keyed Stream
        stream.keyBy(0)
                .timeWindow(Time.seconds(3))
                .sum(1)
                .print();
        env.execute("WindowType");
    }
```



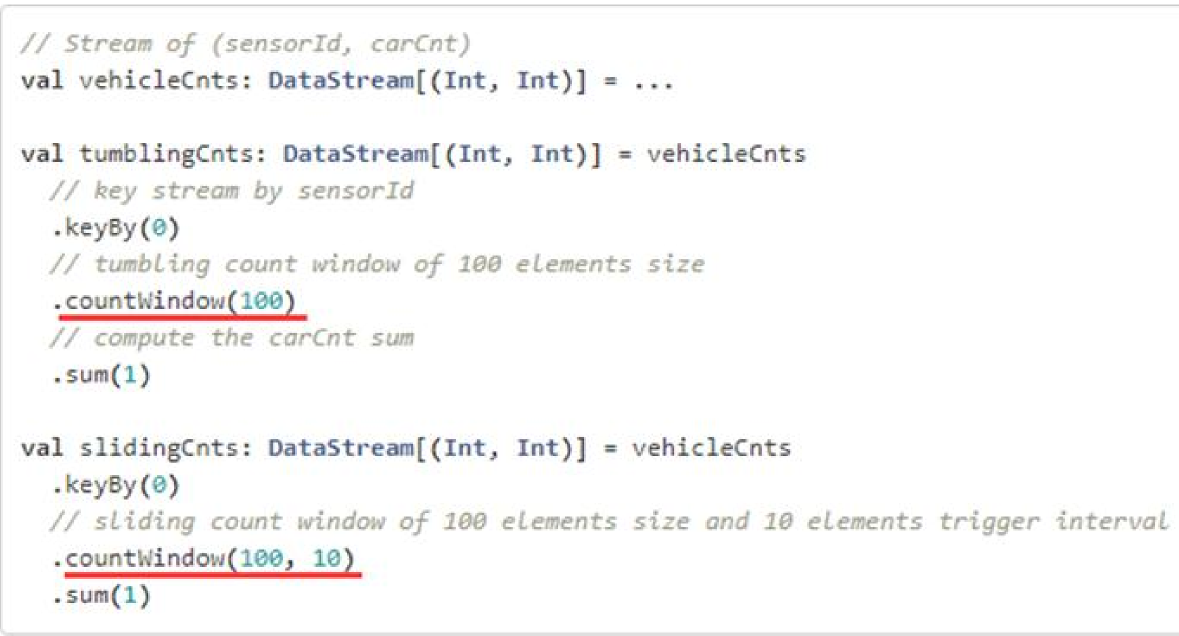


3.2 TimeWindow



```
// Stream of (sensorId, carCnt)
val vehicleCnts: DataStream[(Int, Int)] = ...

val tumblingCnts: DataStream[(Int, Int)] = vehicleCnts
  // key stream by sensorId
  .keyBy(0)
  // tumbling time window of 1 minute length
  .timeWindow(Time.minutes(1))
  // compute sum over carCnt
  .sum(1)

val slidingCnts: DataStream[(Int, Int)] = vehicleCnts
  .keyBy(0)
  // sliding time window of 1 minute length and 30 secs trigger interval
  .timeWindow(Time.minutes(1), Time.seconds(30))
  .sum(1)
```

3.3 CountWindow



```
// Stream of (sensorId, carCnt)
val vehicleCnts: DataStream[(Int, Int)] = ...

val tumblingCnts: DataStream[(Int, Int)] = vehicleCnts
  // key stream by sensorId
  .keyBy(0)
  // tumbling count window of 100 elements size
  .countWindow(100)
  // compute the carCnt sum
  .sum(1)

val slidingCnts: DataStream[(Int, Int)] = vehicleCnts
  .keyBy(0)
  // sliding count window of 100 elements size and 10 elements trigger interval
  .countWindow(100, 10)
  .sum(1)
```

```
案例3 -> job3 (TimeWindow和CountWindow的区别)
public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> dataStream =
env.socketTextStream("192.168.134.130", 8888);

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

        /**
         *
         *  滚动窗口 和 滑动窗口的区别: 1个参数的是滚动,2个参数的是滑动
         *  .timeWindow(Time.seconds(2))  ==
window(TumblingProcessingTimeWindows.of(Time.seconds(2)))
         *  .timeWindow(Time.seconds(6),Time.seconds(4))
         */
        stream.keyBy("0")
                //每隔10个元素,统计最近100个元素的情况
//                  .countWindow(100,10)
                // 每100个元素统计一次
//                   .countWindow(100)
//                  .timeWindow(Time.seconds(5))
                //.timeWindow(Time.seconds(3),Time.seconds(5))
                .sum(1)
                .print();
        //滚动窗口
        stream.keyBy(0)
               // .timeWindow(Time.seconds(2))
               .window(TumblingProcessingTimeWindows.of(Time.seconds(2)))
                .sum(1)
                .print();

        //滑动窗口
        stream.keyBy(0)
//
.window(SlidingProcessingTimeWindows.of(Time.seconds(6),Time.seconds(4)))
                .timeWindow(Time.seconds(10),Time.seconds(5))
                .sum(1)
                .print();
        env.execute("word count");
    }
```

3.4 自定义Window

- 一般前面两种window就能解决我们所遇到的业务场景了，自定义window作为了解

4.window操作

4.1 Keyed Windows方法

```
stream
    .keyBy(...) <- keyed versus non-keyed windows
    .window(...) <- required: "assigner"
    [.trigger(...)] <- optional: "trigger" (else default trigger)
    [.evictor(...)] <- optional: "evictor" (else no evictor)
    [.allowedLateness(...)] <- optional: "lateness" (else zero)
    [.sideOutputLateData(...)] <- optional: "output tag" (else no side output for
late data)
    .reduce/aggregate/fold/apply() <- required: "function"
    [.getSideOutput(...)] <- optional: "output tag"
```

4.2 Non-Keyed Windows方法

```
stream
    .windowAll(...) <- required: "assigner"
    [.trigger(...)] <- optional: "trigger" (else default trigger)
    [.evictor(...)] <- optional: "evictor" (else no evictor)
    [.allowedLateness(...)] <- optional: "lateness" (else zero)
    [.sideOutputLateData(...)] <- optional: "output tag" (else no side output for
late data)
     .reduce/aggregate/fold/apply() <- required: "function"
    [.getSideOutput(...)] <- optional: "output tag"
```

4.3 window function

4.1.1 Tumbling window和slide window

```
//滚动窗口
stream.keyBy(0)
.window(TumblingEventTimeWindows.of(Time.seconds(2)))
.sum(1)
.print();

//滑动窗口
stream.keyBy(0)
.window(SlidingProcessingTimeWindows.of(Time.seconds(6),Time.seconds(4)))
.sum(1)
.print();
```

4.1.2 session window 案例

```
/**
* 5秒过去以后，该单词不出现就打印出来该单词
*/
案例 -> job4
public static void main(String[] args) throws  Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> dataStream =
env.socketTextStream("192.168.134.130", 9999);

        SingleOutputStreamOperator<Tuple2<String, Integer>> stream =
dataStream.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
            @Override
            public void flatMap(String line, Collector<Tuple2<String, Integer>>
collector) throws Exception {
                String[] fields = line.split(",");
                for (String word : fields) {
                    collector.collect(Tuple2.of(word, 1));
                }
            }
        });
        stream.keyBy(0)
                .window(ProcessingTimeSessionWindows.withGap(Time.seconds(5)))
                .sum(1)
                .print();
        env.execute("SessionWindowTest");
    }
```

4.1.3 global window 案例

- global window + trigger 一起配合才能使用
- 需求：单词每出现三次统计一次

```
/**
* 单词每出现三次统计一次
*/
public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> dataStream =
env.socketTextStream("192.168.134.130", 9999);

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

        stream.keyBy(0)
```

```
                //.countWindow(3)
                //如果不配合 trigger 不会运行该窗口
                .window(GlobalWindows.create())
                .trigger(new CountTrigger(3))
                .sum(1)
                .print();

        env.execute("SessionWindowTest");
    }

    /**
     *自定义一个触发策略：数据的个数达到某个值的时候，触发计算
     * state：用来统计数据的个数
     */
    private static class CountTrigger extends Trigger<Tuple2<String,Integer>,
GlobalWindow>{

        private long maxCount;

        // 注册状态
        ReducingStateDescriptor<Long> descriptor =
                new ReducingStateDescriptor<Long>(
                        "count",  // 状态的名字
                        new ReduceFunction<Long>() { // 聚合函数
                            @Override
                            public Long reduce(Long value1, Long value2) throws
Exception {
                                return value1 + value2;
                            }
                        }, Long.class); // 状态存储的数据类型


        public CountTrigger(long maxCount){
            this.maxCount = maxCount;
        }

        /**
         *
         * 每个元素来了以后都会调用这个方法。
         * @param element
         * @param timestamp
         * @param window
         * @param ctx
         * @return
         * @throws Exception
         *
         *
         *
        CONTINUE 表示对窗口不进行任何处理
        FIRE_AND_PURGE ： 触发window的计算，并且清除当前window的数据
        FIRE 触发window的计算
        PURGE 清除window中所有的数据
```

```
         *
         */
        @Override
        public TriggerResult onElement(Tuple2<String, Integer> element,
                                       long timestamp, GlobalWindow window,
                                       TriggerContext ctx) throws Exception {

            //获取当前key对应的count
            ReducingState<Long> count = ctx.getPartitionedState(descriptor);
            count.add(1L);
            if(count.get() == maxCount){
                count.clear();
                return TriggerResult.FIRE_AND_PURGE;
            }

            return TriggerResult.CONTINUE;
        }

        @Override
        public TriggerResult onProcessingTime(long time, GlobalWindow window,
TriggerContext ctx) throws Exception {
            return TriggerResult.CONTINUE;
        }

        @Override
        public TriggerResult onEventTime(long time, GlobalWindow window,
TriggerContext ctx) throws Exception {
            return TriggerResult.CONTINUE;
        }

        @Override
        public void clear(GlobalWindow window, TriggerContext ctx) throws
Exception {

            ctx.getPartitionedState(descriptor).clear();
        }
    }

执行结果:
flink,3
flink,6
flink,9
总结：效果跟CountWindow(3) 很像，但又有点不像，因为如果是CountWindow(3)，单词每次出现的
都是3次，不会包含之前的次数，而我们刚刚的这个每次都包含了之前的次数
```

4.3 Trigger

- 需求：自定义一个CountWindow
- 注：效果跟CountWindow一模一样

4.4 Evictor

- 需求：实现每隔2个单词，计算最近3个单词

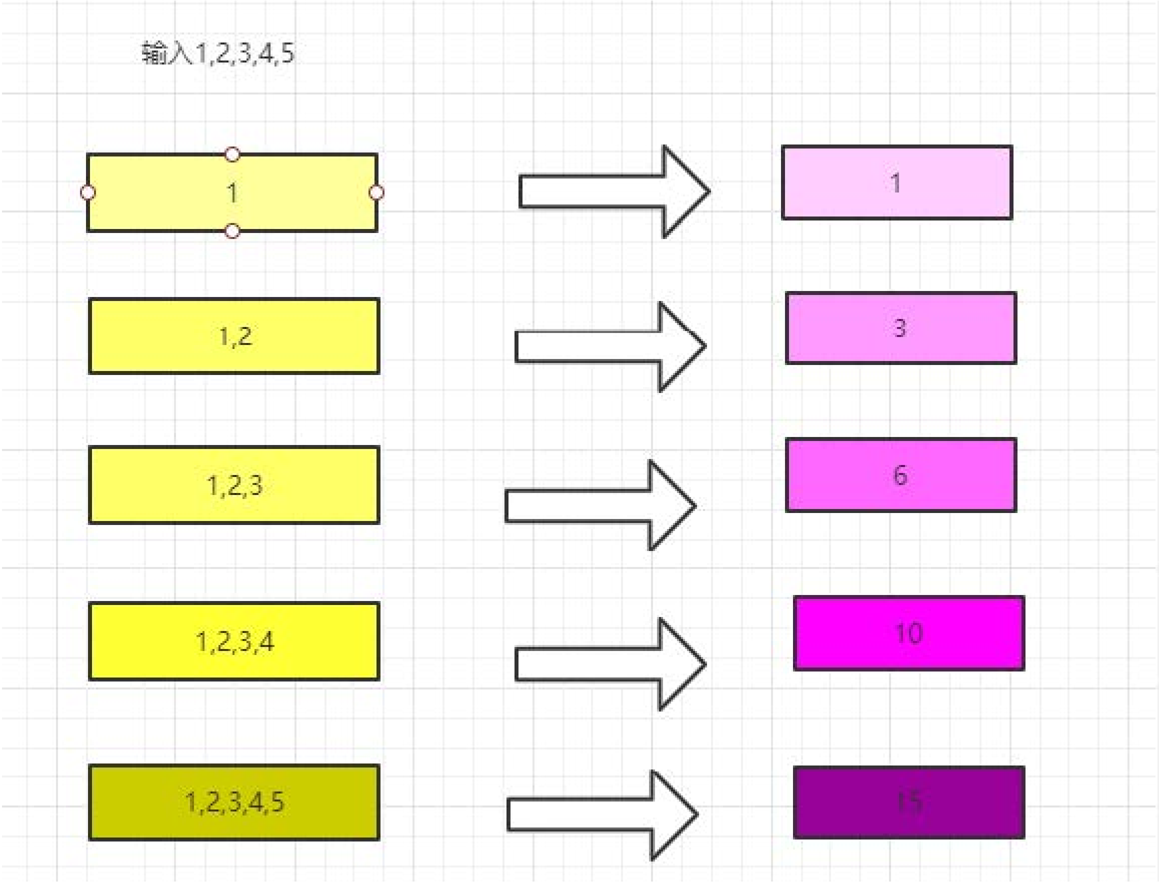
```
案例6 -> job6
public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStreamSource<String> dataStream =
env.socketTextStream("192.168.134.130", 9999);
        env.setParallelism(1);

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





        WindowedStream<Tuple2<String, Integer>, Tuple, GlobalWindow> keyedWindow
= stream.keyBy(0)
              .window(GlobalWindows.create())
                .trigger(new MyCountTrigger(3))
                .evictor(new MyCountEvictor(3));




        DataStream<Tuple2<String, Integer>> wordCounts = keyedWindow.sum(1);

        wordCounts.print().setParallelism(1);

        env.execute("Streaming WordCount");
    }



    private static class MyCountTrigger
            extends Trigger<Tuple2<String, Integer>, GlobalWindow> {
        // 表示指定的元素的最大的数量
        private long maxCount;

        // 用于存储每个 key 对应的 count 值
```

```java
        private ReducingStateDescriptor<Long> stateDescriptor
                = new ReducingStateDescriptor<Long>("count", new
ReduceFunction<Long>() {
            @Override
            public Long reduce(Long aLong, Long t1) throws Exception {
                return aLong + t1;
            }
        }, Long.class);

        public MyCountTrigger(long maxCount) {
            this.maxCount = maxCount;
        }

        /**
         *  当一个元素进入到一个 window 中的时候就会调用这个方法
         * @param element   元素
         * @param timestamp 进来的时间
         * @param window    元素所属的窗口
         * @param ctx 上下文
         * @return TriggerResult
         *      1. TriggerResult.CONTINUE ：表示对 window 不做任何处理
         *      2. TriggerResult.FIRE ：表示触发 window 的计算
         *      3. TriggerResult.PURGE ：表示清除 window 中的所有数据
         *      4. TriggerResult.FIRE_AND_PURGE ：表示先触发 window 计算，然后删除
window 中的数据
         * @throws Exception
         */
        @Override
        public TriggerResult onElement(Tuple2<String, Integer> element,
                                       long timestamp,
                                       GlobalWindow window,
                                       TriggerContext ctx) throws Exception {
            // 拿到当前 key 对应的 count 状态值
            ReducingState<Long> count = ctx.getPartitionedState(stateDescriptor);
            // count 累加 1
            count.add(1L);
            // 如果当前 key 的 count 值等于 maxCount
            if (count.get() == maxCount) {
                count.clear();
                // 触发 window 计算
                return TriggerResult.FIRE;
            }
            // 否则，对 window 不做任何的处理
            return TriggerResult.CONTINUE;
        }

        @Override
        public TriggerResult onProcessingTime(long time,
                                              GlobalWindow window,
                                              TriggerContext ctx) throws
Exception {
            // 写基于 Processing Time 的定时器任务逻辑
            return TriggerResult.CONTINUE;
        }

        @Override
        public TriggerResult onEventTime(long time,
                                         GlobalWindow window,
                                         TriggerContext ctx) throws Exception {
            // 写基于 Event Time 的定时器任务逻辑
            return TriggerResult.CONTINUE;
        }

        @Override
        public void clear(GlobalWindow window, TriggerContext ctx) throws
Exception {
            // 清除状态值
            ctx.getPartitionedState(stateDescriptor).clear();
        }
    }



    private static class MyCountEvictor
            implements Evictor<Tuple2<String, Integer>, GlobalWindow> {
        // window 的大小
        private long windowCount;

        public MyCountEvictor(long windowCount) {
            this.windowCount = windowCount;
        }

        /**
         *  在 window 计算之前删除特定的数据
         * @param elements  window 中所有的元素
         * @param size  window 中所有元素的大小
         * @param window    window
         * @param evictorContext    上下文
         */
        @Override
        public void evictBefore(Iterable<TimestampedValue<Tuple2<String,
Integer>>> elements,
                                int size,
                                GlobalWindow window,
                                EvictorContext evictorContext) {
            if (size <= windowCount) {
                return;
            } else {
                int evictorCount = 0;
                Iterator<TimestampedValue<Tuple2<String, Integer>>> iterator =
elements.iterator();
                while (iterator.hasNext()) {
                    iterator.next();
                    evictorCount++;
                    // 如果删除的数量小于当前的 window 大小减去规定的 window 的大小，就需要删除当前的元素
                    if (evictorCount > size - windowCount) {
                        break;
                    } else {
                        iterator.remove();
                    }
                }
            }
        }

        /**
         *  在 window 计算之后删除特定的数据
         * @param elements  window 中所有的元素
         * @param size  window 中所有元素的大小
         * @param window    window
         * @param evictorContext    上下文
         */
        @Override
        public void evictAfter(Iterable<TimestampedValue<Tuple2<String, Integer>>> elements,
                               int size, GlobalWindow window, EvictorContext evictorContext) {

        }
    }
```

4.5 window增量聚合

- 窗口中每进入一条数据，就进行一次计算，等时间到了展示最后的结果
- 常用的聚合算子

```
reduce(reduceFunction)
aggregate(aggregateFunction)
sum(),min(),max()
```



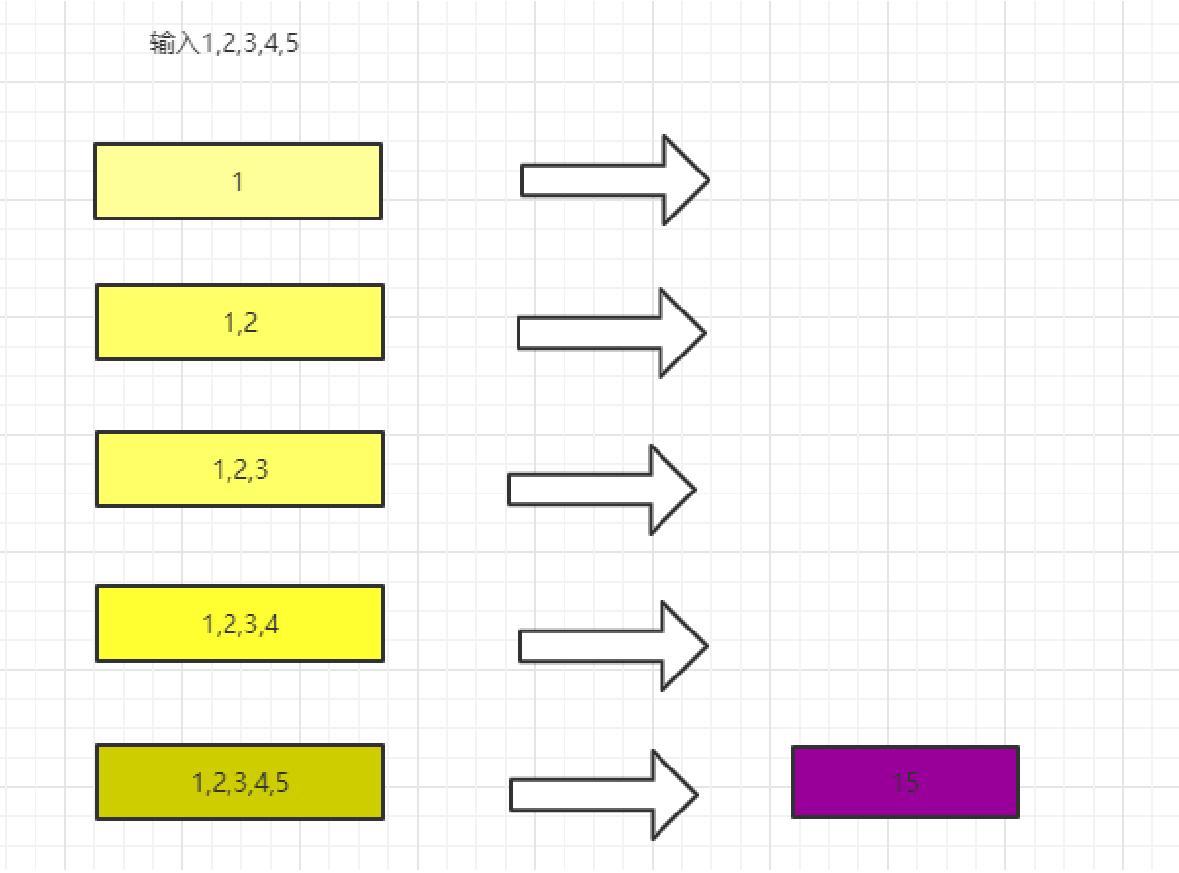
```
/**
* 演示增量聚合
*/
public class SocketDemoIncrAgg {
public static void main(String[] args) throws Exception{
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
DataStreamSource<String> dataStream = env.socketTextStream("localhost",
8888);
SingleOutputStreamOperator<Integer> intDStream = dataStream.map(number -
> Integer.valueOf(number));
AllWindowedStream<Integer, TimeWindow> windowResult =
intDStream.timeWindowAll(Time.seconds(10));
windowResult.reduce(new ReduceFunction<Integer>() {
@Override
public Integer reduce(Integer last, Integer current) throws Exception
{
System.out.println("执行逻辑"+last + " "+current);
return last+current;
}
}).print();
env.execute(SocketDemoIncrAgg.class.getSimpleName());
}
}

aggregate算子
需求：求每隔窗口里面的数据的平均值
/**
```

```
* 求每隔窗口中的数据的平均值
*/
public class aggregateWindowTest {
public static void main(String[] args) throws Exception{
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
DataStreamSource<String> dataStream =
env.socketTextStream("10.148.15.10", 8888);
SingleOutputStreamOperator<Integer> numberStream = dataStream.map(line -
> Integer.valueOf(line));
AllWindowedStream<Integer, TimeWindow> windowStream =
numberStream.timeWindowAll(Time.seconds(5));
windowStream.aggregate(new MyAggregate())
.print();
env.execute("aggregateWindowTest");
}
/**
* IN, 输入的数据类型
* ACC,自定义的中间状态
* Tuple2<Integer,Integer>:
* key: 计算数据的个数
* value:计算总值
* OUT, 输出的数据类型
*/
private static class MyAggregate
implements AggregateFunction<Integer,Tuple2<Integer,Integer>,Double>
{
/**
* 初始化 累加器
* @return
*/
@Override
public Tuple2<Integer, Integer> createAccumulator() {
return new Tuple2<>(0,0);
}
/**
* 针对每个数据的操作
* @return
*/
@Override
public Tuple2<Integer, Integer> add(Integer element,
Tuple2<Integer, Integer>
accumulator) {
//个数+1
//总的值累计
return new Tuple2<>(accumulator.f0+1,accumulator.f1+element);
}
@Override
public Double getResult(Tuple2<Integer, Integer> accumulator) {
return (double)accumulator.f1/accumulator.f0;
}
@Override
public Tuple2<Integer, Integer> merge(Tuple2<Integer, Integer> a1,
Tuple2<Integer, Integer> b1) {
return Tuple2.of(a1.f0+b1.f0,a1.f1+b1.f1);
}
}
}
```

4.6 window全量聚合

- 等属于窗口的数据到齐，才开始进行聚合计算【可以实现对窗口内的数据进行排序等需求】

```
apply(windowFunction)
process(processWindowFunction)
processWindowFunction比windowFunction提供了更多的上下文信息。类似于map和RichMap的关系
```



```
/**
* 全量计算
*/
public class SocketDemoFullAgg {
public static void main(String[] args) throws Exception {
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
DataStreamSource<String> dataStream = env.socketTextStream("localhost",
8888);
SingleOutputStreamOperator<Integer> intDStream = dataStream.map(number -
> Integer.valueOf(number));
AllWindowedStream<Integer, TimeWindow> windowResult =
intDStream.timeWindowAll(Time.seconds(10));
windowResult.process(new ProcessAllWindowFunction<Integer, Integer,
TimeWindow>() {
@Override
public void process(Context context, Iterable<Integer> iterable,
Collector<Integer> collector) throws Exception {
System.out.println("执行计算逻辑");
int count=0;
Iterator<Integer> numberiterator = iterable.iterator();
while (numberiterator.hasNext()){
Integer number = numberiterator.next();
count+=number;
}
collector.collect(count);
}
}).print();
env.execute("socketDemoFullAgg");
}
}
```

- ## 5. window join

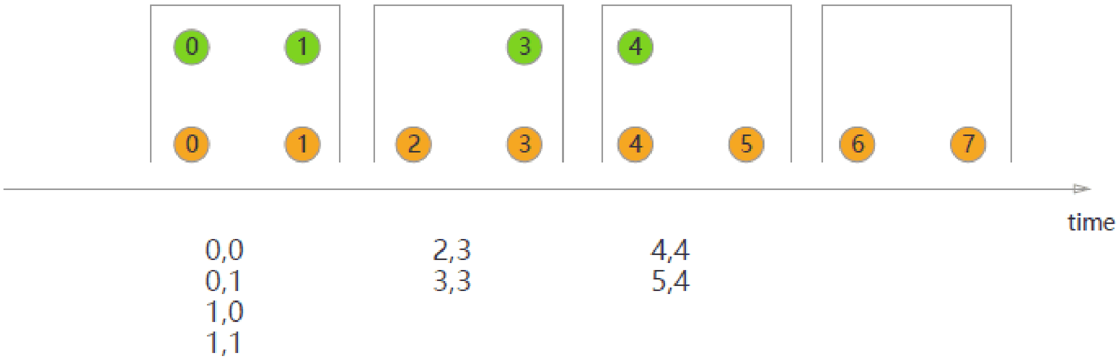
  - 两个window之间可以进行join，join操作只支持三种类型的window：滚动窗口，滑动窗口，会话窗口

```
使用方式：
stream.join(otherStream) //两个流进行关联
    .where(<KeySelector>) //选择第一个流的key作为关联字段
    .equalTo(<KeySelector>)//选择第二个流的key作为关联字段
    .window(<WindowAssigner>)//设置窗口的类型
    .apply(<JoinFunction>) //对结果做操作
```

5.1 Tumbling Window Join

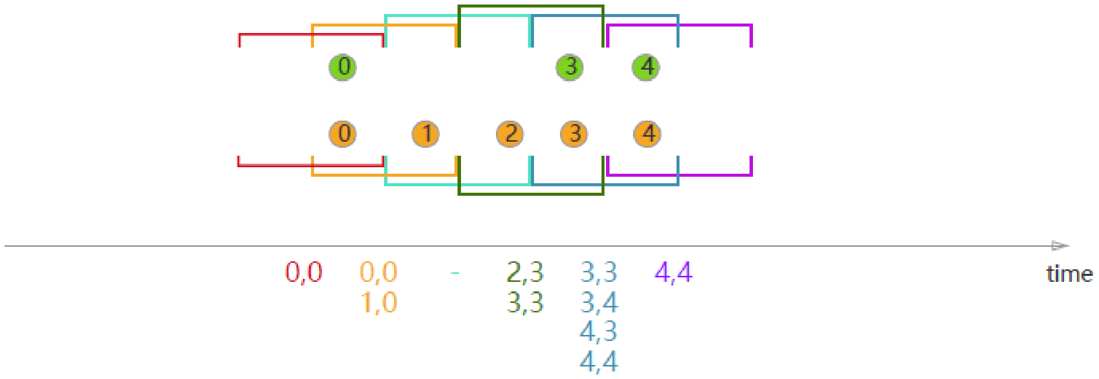
```
import org.apache.flink.api.java.functions.KeySelector;
import
org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
...
DataStream<Integer> orangeStream = ...
DataStream<Integer> greenStream = ...
orangeStream.join(greenStream)
.where(<KeySelector>)
.equalTo(<KeySelector>)
.window(TumblingEventTimeWindows.of(Time.milliseconds(2)))
.apply (new JoinFunction<Integer, Integer, String> (){
```

```
@Override
public String join(Integer first, Integer second) {
return first + "," + second;
}
});
```



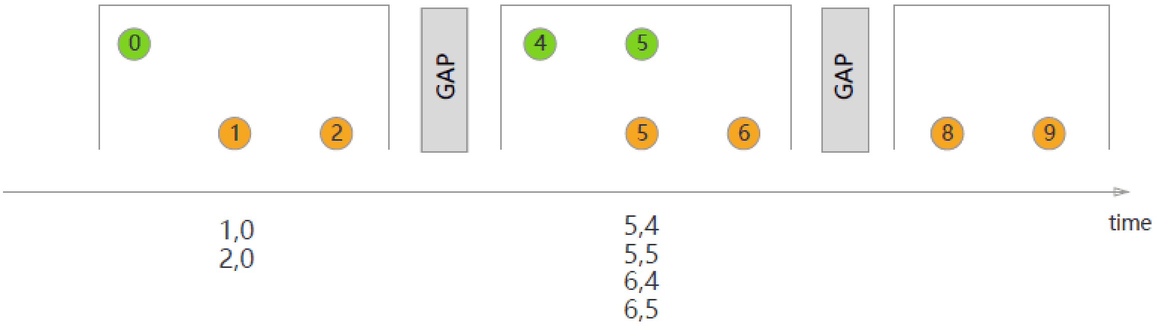
5.2 Sliding Window Join

```
import org.apache.flink.api.java.functions.KeySelector;
import
org.apache.flink.streaming.api.windowing.assigners.SlidingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
...
DataStream<Integer> orangeStream = ...
DataStream<Integer> greenStream = ...
orangeStream.join(greenStream)
.where(<KeySelector>)
.equalTo(<KeySelector>)
.window(SlidingEventTimeWindows.of(Time.milliseconds(2) /* size */,
Time.milliseconds(1) /* slide */))
.apply (new JoinFunction<Integer, Integer, String> (){
@Override
public String join(Integer first, Integer second) {
return first + "," + second;
}
});
```



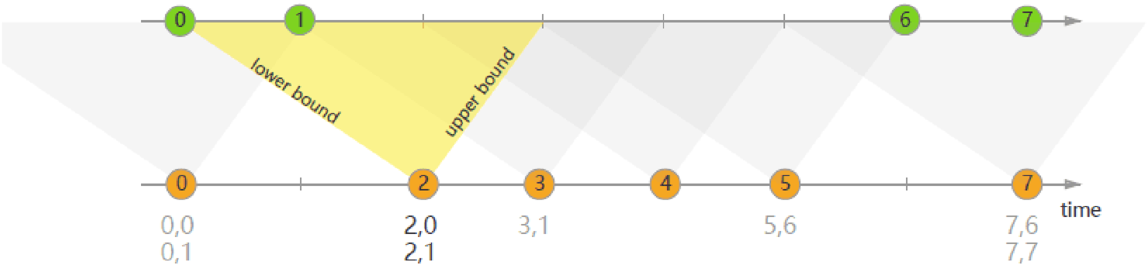
5.3 Session Window Join

```
import org.apache.flink.api.java.functions.KeySelector;
import
org.apache.flink.streaming.api.windowing.assigners.EventTimeSessionWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
...
DataStream<Integer> orangeStream = ...
DataStream<Integer> greenStream = ...
orangeStream.join(greenStream)
.where(<KeySelector>)
.equalTo(<KeySelector>)
    //1秒钟没有出现过
.window(EventTimeSessionWindows.withGap(Time.milliseconds(1)))
.apply (new JoinFunction<Integer, Integer, String> (){
@Override
public String join(Integer first, Integer second) {
return first + "," + second;
}
});
```



5.4 Interval Join

```
import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.streaming.api.functions.co.ProcessJoinFunction;
import org.apache.flink.streaming.api.windowing.time.Time;
...
DataStream<Integer> orangeStream = ...
DataStream<Integer> greenStream = ...
orangeStream
.keyBy(<KeySelector>)
.intervalJoin(greenStream.keyBy(<KeySelector>))
    //往前推2s,往后推1s
.between(Time.milliseconds(-2), Time.milliseconds(1))
.process (new ProcessJoinFunction<Integer, Integer, String(){
@Override
public void processElement(Integer left, Integer right, Context ctx,
Collector<String> out) {
out.collect(first + "," + second);
}
});
```



0
1
6
7
lower bound
upper bound
0
2
3
4
5
7
time
0,0
0,1
2,0
2,1
3,1
5,6
7,6
7,7