透过窗口看无限数据流——Flink的Window全面解析

窗口是流式计算中非常常用的算子之一，通过窗口可以将无限流切分成有限流，然后在每个窗口之上使用计算函数，可以实现非常灵活的操作。Flink提供了丰富的窗口操作，除此之外，用户还可以根据自己的处理场景自定义窗口。通过本文，你可以了解到：

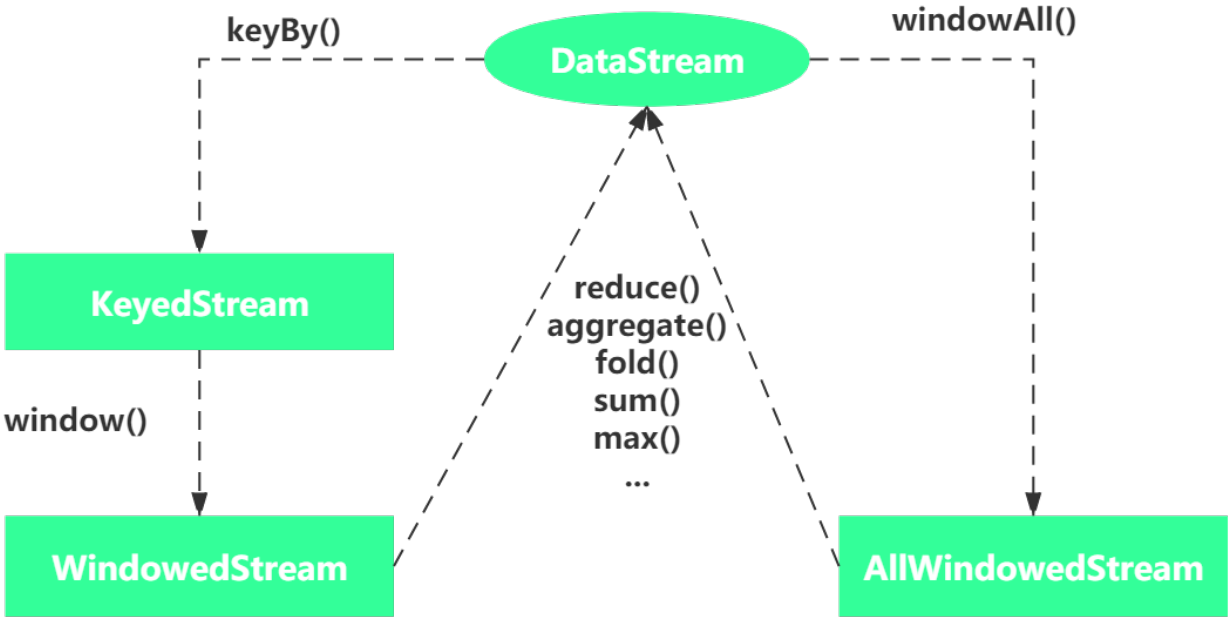
- 窗口的基本概念和简单使用
- 内置Window Assigners的分类、源码及使用
- Window Function的分类及使用
- 窗口的组成部分及生命周期源码解读
- 完整的窗口使用Demo案例

Quick Start

是什么

Window(窗口)是处理无界流的核心算子，Window可以将数据流分为固定大小的"桶(buckets)"(即通过按照固定时间或长度将数据流切分成不同的窗口)，在每一个窗口上，用户可以使用一些计算函数对窗口内的数据进行处理，从而得到一定时间范围内的统计结果。比如统计每隔5分钟输出最近一小时内点击量最多的前 N 个商品，这样就可以使用一个小时的时间窗口将数据限定在固定时间范围内，然后可以对该范围内的有界数据执行聚合处理。

根据作用的数据流(DataStream、KeyedStream)，Window可以分为两种：**Keyed Windows**与**Non-Keyed Windows**。其中Keyed Windows是在KeyedStream上使用window(…)操作，产生一个WindowedStream。Non-Keyed Windows是在DataStream上使用windowAll(…)操作，产生一个AllWindowedStream。具体的转换关系如下图所示。注意：一般不推荐使用`AllWindowedStream`，因为在普通流上进行窗口操作，会将所有分区的流都汇集到单个的Task中，即并行度为1，从而会影响性能。



如何用

上面我们介绍了什么是窗口，那么该如何使用窗口呢?具体如下面的代码片段:

Keyed Windows

```
stream
       .keyBy(...)               // keyedStream上使用window
       .window(...)              // 必选: 指定窗口分配器( window assigner)
      [.trigger(...)]            // 可选: 指定触发器(trigger),如果不指定，则
使用默认值
      [.evictor(...)]            // 可选: 指定清除器(evictor),如果不指定，则
没有
      [.allowedLateness(...)]    // 可选: 指定是否延迟处理数据，如果不指定，默
认使用0
      [.sideOutputLateData(...)] // 可选: 配置side output，如果不指定，则没有
       .reduce/aggregate/fold/apply() // 必选: 指定窗口计算函数
      [.getSideOutput(...)]      // 可选: 从side output中获取数据
```

Non-Keyed Windows

```
stream
       .windowAll(...)           // 必选: 指定窗口分配器( window assigner)
      [.trigger(...)]            // 可选: 指定触发器(trigger),如果不指定，则
使用默认值
      [.evictor(...)]            // 可选: 指定清除器(evictor),如果不指定，则
没有
      [.allowedLateness(...)]    // 可选: 指定是否延迟处理数据，如果不指定，默
```

```
    认使用0
          [.sideOutputLateData(...)] // 可选: 配置side output, 如果不指定, 则没有
           .reduce/aggregate/fold/apply() // 必选: 指定窗口计算函数
          [.getSideOutput(...)]      // 可选: 从side output中获取数据
```

简写window操作

上面的代码片段中，要在keyedStream上使用window(…)或者在DataStream上使用windowAll(…)，需要传入一个window assigner的参数，关于window assigner下文会进行详细解释。如下面代码片段：

```
// ------------------------------------------
//  Keyed Windows
// ------------------------------------------
stream
       .keyBy(id)
       .window(TumblingEventTimeWindows.of(Time.seconds(5))) // 5S的滚动窗口
       .reduce(MyReduceFunction)
// ------------------------------------------
//  Non-Keyed Windows
// ------------------------------------------
stream
       .windowAll(TumblingEventTimeWindows.of(Time.seconds(5))) // 5S的滚动窗口
       .reduce(MyReduceFunction)
```

上面的代码可以简写为：

```
// ------------------------------------------
//  Keyed Windows
// ------------------------------------------
stream
       .keyBy(id)
       .timeWindow(Time.seconds(5)) // 5S的滚动窗口
       .reduce(MyReduceFunction)
// ------------------------------------------
//  Non-Keyed Windows
// ------------------------------------------
stream
       .timeWindowAll(Time.seconds(5)) // 5S的滚动窗口
       .reduce(MyReduceFunction)
```

关于上面的简写，以KeyedStream为例，对于看一下具体的KeyedStream源码片段，可以看出底层调用的还是非简写时的代码。关于timeWindowAll()的代码也是一样的，可以参考DataStream源码，这里不再赘述。

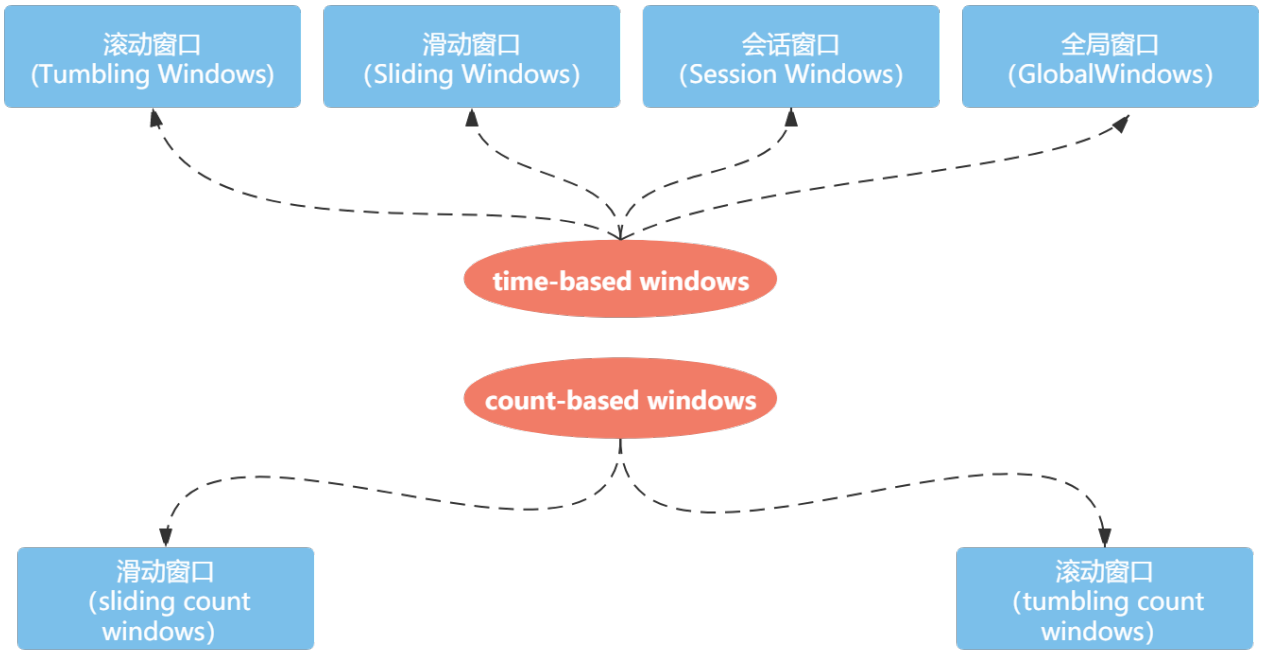
```
// 会根据用户的使用的时间类型，调用不同的内置window Assigner
public WindowedStream<T, KEY, TimeWindow> timeWindow(Time size) {
                if (environment.getStreamTimeCharacteristic() == TimeCha
racteristic.ProcessingTime) {
                        return window(TumblingProcessingTimeWindows.of(s
ize));
                } else {
                        return window(TumblingEventTimeWindows.of(size))
;
                }
        }
```

Window Assigners

分类

WindowAssigner负责将输入的数据分配到一个或多个窗口，Flink内置了许多WindowAssigner，这些WindowAssigner可以满足大部分的使用场景。比如**tumbling windows**, **sliding windows**, **session windows** , **global windows**。如果这些内置的WindowAssigner不能满足你的需求，可以通过继承WindowAssigner类实现自定义的WindowAssigner。

上面的WindowAssigner是基于时间的(time-based windows)，除此之外，Flink还提供了基于数量的窗口(count-based windows),即根据窗口的元素数量定义窗口大小，这种情况下，如果数据存在乱序，将导致窗口计算结果不确定。本文重点介绍基于时间的窗口使用，由于篇幅有限，关于基于数量的窗口将不做讨论。



使用介绍

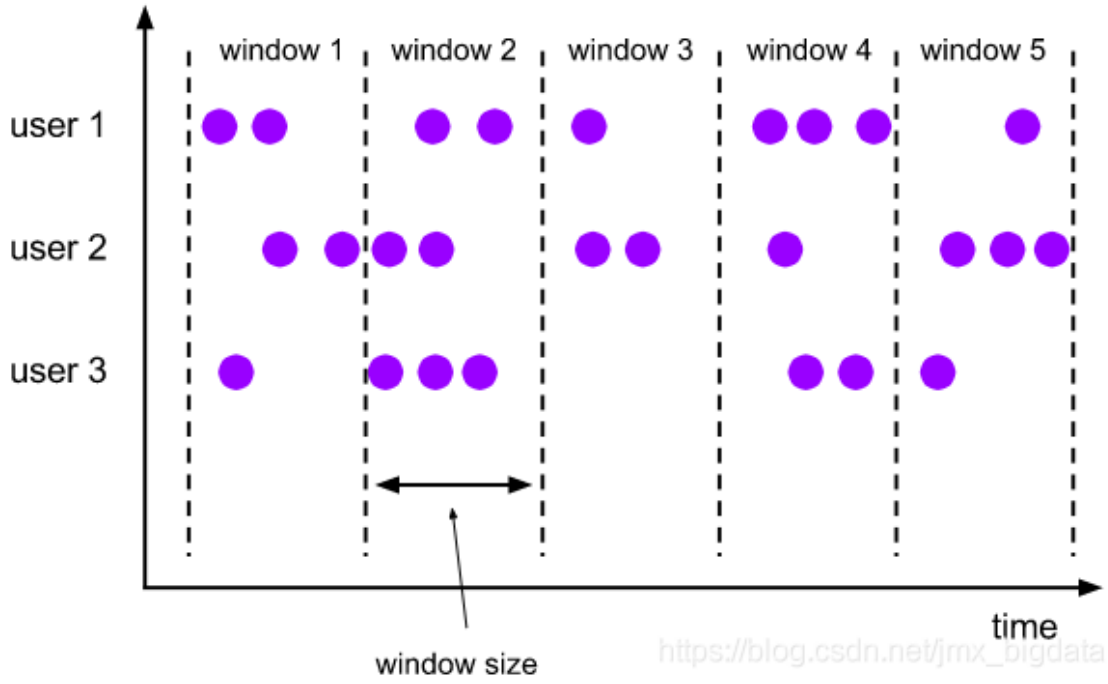
下面将会对Flink内置的四种基于时间的windowassigner，进行一一分析。

Tumbling Windows

- 图解

Tumbling Windows(滚动窗口)是将数据分配到确定的窗口中，根据固定时间或大小进行切分，每个窗口有固定的大小且窗口之间不存在重叠(如下图所示)。这种比较简单，适用于按照周期统计某一指标的场景。

关于时间的选择，可以使用Event Time或者Processing Time，分别对应的window assigner为：TumblingEventTimeWindows、TumblingProcessingTimeWindows。用户可以使用window assigner的of(size)方法指定时间间隔，其中时间单位可以是Time.milliseconds(x)、Time.seconds(x)或Time.minutes(x)等。



- 使用

```
// 使用EventTime
datastream
            .keyBy(id)
            .window(TumblingEventTimeWindows.of(Time.seconds(10)))
            .process(new MyProcessFunction())
// 使用processing-time
datastream
            .keyBy(id)
            .window(TumblingProcessingTimeWindows.of(Time.seconds(10)))
            .process(new MyProcessFunction())
```

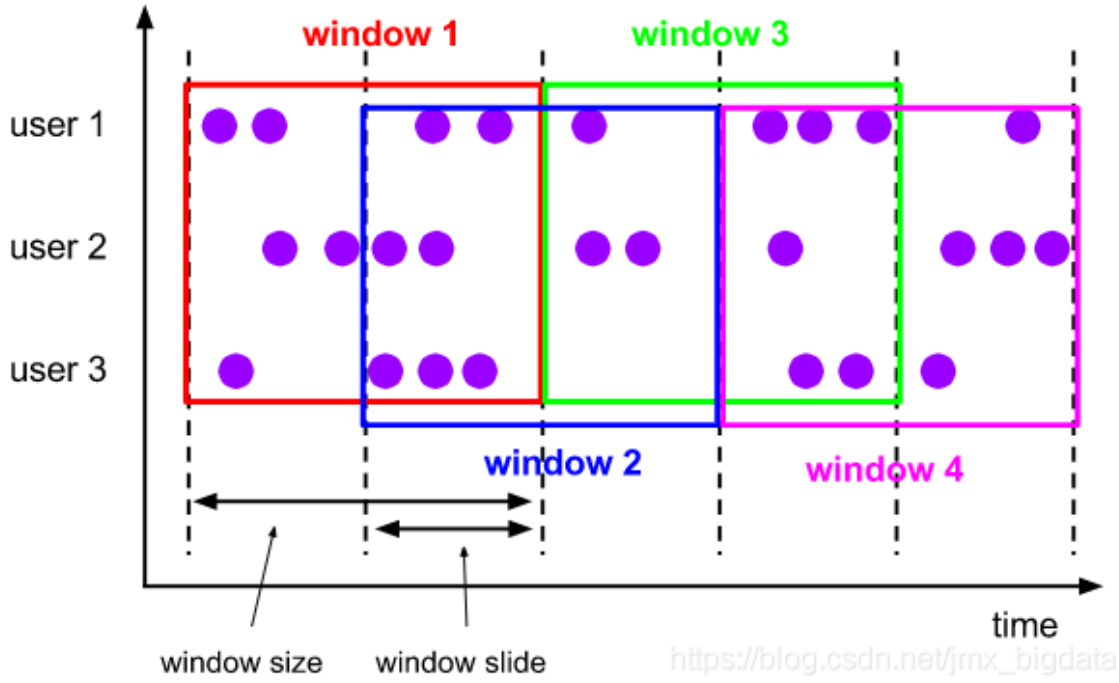
Sliding Windows

- 图解

Sliding Windows(滑动窗口)在滚动窗口之上加了一个滑动窗口的时间，这种类型的窗口是会存在窗口重叠的(如下图所示)。滚动窗口是按照窗口固定的时间大小向前滚动，而滑动窗口是根据设定的滑动时间向前滑动。窗口之间的重叠部分的大小取决于窗口大小与滑动的时间大小，当滑动时间小于窗口时间大小时便会出现重叠。当滑动时间大于窗口时间大小时，会出现窗口不连续的情况，导致数据可能不属于任何一个窗口。当两者相等时，其功能就和滚动窗口相同了。滑动窗口的使用场景是：用户根据设定的统计周期来计算指定窗口时间大小的指标，比如每隔5分钟输出最近一小时内点击量最多的前 N 个商品。

关于时间的选择，可以使用Event Time或者Processing Time，分别对应的window assigner为：SlidingEventTimeWindows、SlidingProcessingTimeWindows。用户可以使

用window assigner的of(size)方法指定时间间隔，其中时间单位可以是Time.milliseconds(x)、Time.seconds(x)或Time.minutes(x)等。



- 使用

```
// 使用EventTime
datastream
            .keyBy(id)
            .window(SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(5)))
            .process(new MyProcessFunction())
// 使用processing-time
datastream
            .keyBy(id)
            .window(SlidingProcessingTimeWindows.of(Time.seconds(10), Time.seconds(5)))
            .process(new MyProcessFunction())
```

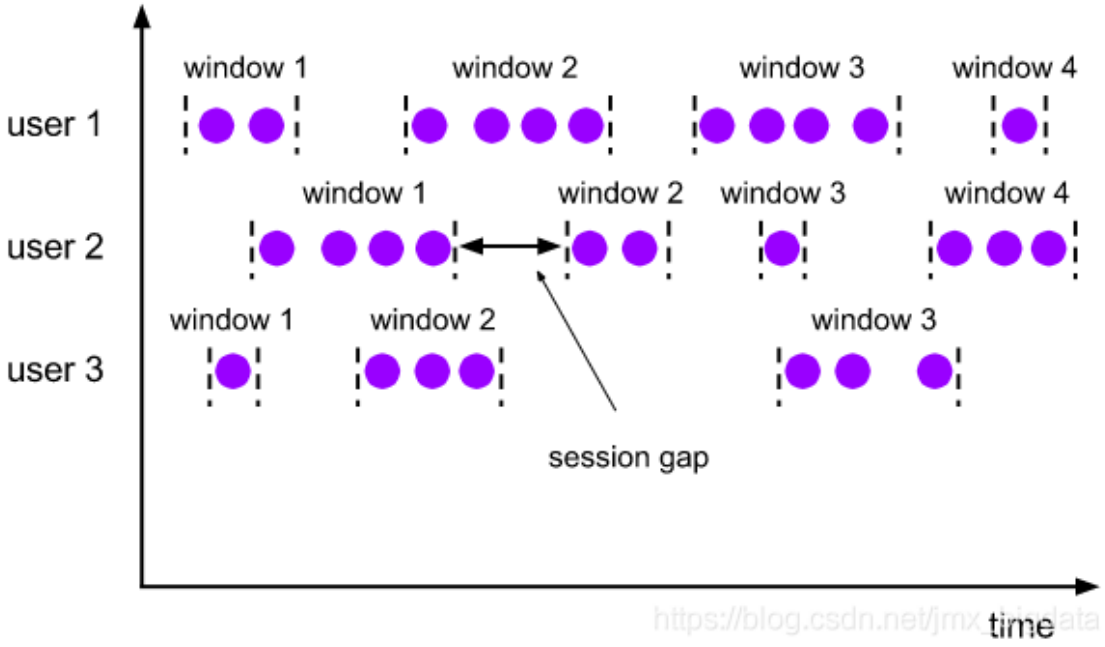
Session Windows

- 图解

Session Windows(会话窗口)主要是将某段时间内活跃度较高的数据聚合成一个窗口进行计算，窗口的触发的条件是Session Gap，是指在规定的时间内如果没有数据活跃接入，则认为窗口结束，然后触发窗口计算结果。需要注意的是如果数据一直不间断地进入窗口，也会导致窗口始终不触发的情况。与滑动窗口、滚动窗口不同的是，Session Windows不需要有固定窗口大小(window size)和滑动时间(slide time)，只需要定义session gap，来规定不活跃数据的时间上限即可。如下图所示。Session Windows窗口

类型比较适合非连续型数据处理或周期性产生数据的场景，根据用户在线上某段时间内的活跃度对用户行为数据进行统计。

关于时间的选择，可以使用Event Time或者Processing Time，分别对应的window assigner为：EventTimeSessionWindows和ProcessTimeSessionWindows。用户可以使用window assigner的withGap()方法指定时间间隔，其中时间单位可以是Time.milliseconds(x)、Time.seconds(x)或Time.minutes(x)等。



- 使用

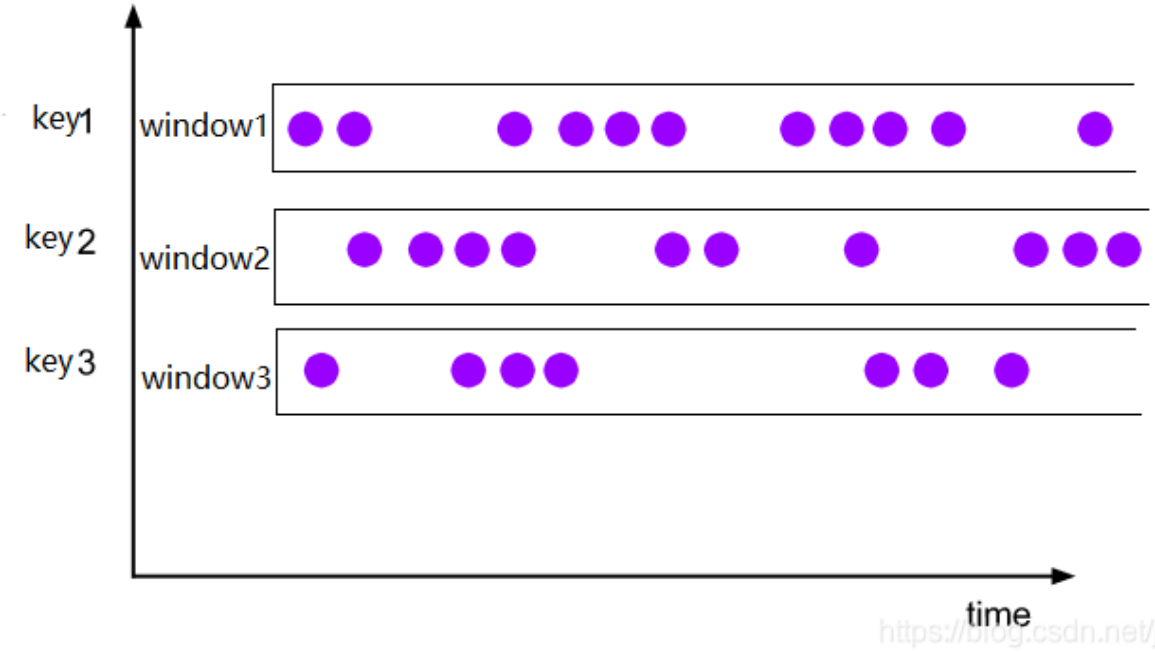
```
// 使用EventTime
datastream
            .keyBy(id)
            .window((EventTimeSessionWindows.withGap(Time.minutes(15)))
            .process(new MyProcessFunction())
// 使用processing-time
datastream
            .keyBy(id)
            .window(ProcessingTimeSessionWindows.withGap(Time.minutes(15)
))
            .process(new MyProcessFunction())
```

注意：由于session window的开始时间与结束时间取决于接收的数据。windowassigner不会立即分配所有的元素到正确的窗口，SessionWindow会为每个接收的元素初始化一个以该元素的时间戳为开始时间的窗口，使用session gap作为窗口大小，然后再合并重叠部分的窗口。所以， session window 操作需要指定用于合并的 Trigger 和 Window Function，比如 `ReduceFunction`，`AggregateFunction`，or `ProcessWindowFunction`。

Global Windows

- 图解

Global Windows(全局窗口)将所有相同的key的数据分配到单个窗口中计算结果，窗口没有起始和结束时间，窗口需要借助于Triger来触发计算，如果不对Global Windows指定Triger，窗口是不会触发计算的。因此，使用Global Windows需要非常慎重，用户需要非常明确自己在整个窗口中统计出的结果是什么，并指定对应的触发器，同时还需要有指定相应的数据清理机制，否则数据将一直留在内存中。



- 使用

```
datastream
    .keyBy(id)
    .window(GlobalWindows.create())
    .process(new MyProcessFunction())

```

Window Functions

分类

Flink提供了两大类窗口函数，分别为增量聚合函数和全量窗口函数。其中增量聚合函数的性能要比全量窗口函数高，因为增量聚合窗口是基于中间结果状态计算最终结果的，即窗口中只维护一个中间结果状态，不要缓存所有的窗口数据。相反，对于全量窗口函数而言，需要对所以进入该窗口的数据进行缓存，等到窗口触发时才会遍历窗口内所有数据，进行结果计算。如果窗口数据量比较大或者窗口时间较长，就会耗费很多的资源缓存数据，从而导致性能下降。

- 增量聚合函数

包括：ReduceFunction、AggregateFunction和FoldFunction

- 全量窗口函数

  包括：ProcessWindowFunction

使用介绍

ReduceFunction

输入两个相同类型的数据元素按照指定的计算方法进行聚合，然后输出类型相同的一个结果元素。要求输入元素的数据类型与输出元素的数据类型必须一致。实现的效果是使用上一次的结果值与当前值进行聚合。具体使用案例如下：

```
public class ReduceFunctionExample {
    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getE
xecutionEnvironment().setParallelism(1);
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

        // 模拟数据源
        SingleOutputStreamOperator<Tuple3<Long, Integer, Long>> input =
env.fromElements(
                Tuple3.of(1L, 10, 1588491228L),
                Tuple3.of(1L, 15, 1588491229L),
                Tuple3.of(1L, 20, 1588491238L),
                Tuple3.of(1L, 25, 1588491248L),
                Tuple3.of(2L, 10, 1588491258L),
                Tuple3.of(2L, 30, 1588491268L),
                Tuple3.of(2L, 20, 1588491278L)).assignTimestampsAndWater
marks(new AscendingTimestampExtractor<Tuple3<Long, Integer, Long>>() {
            @Override
            public long extractAscendingTimestamp(Tuple3<Long, Integer,
Long> element) {
                return element.f2 * 1000;
            }
        });

        input
                .map(new MapFunction<Tuple3<Long, Integer, Long>, Tuple2
<Long, Integer>>() {
                    @Override
                    public Tuple2<Long, Integer> map(Tuple3<Long, Intege
r, Long> value) {
                        return Tuple2.of(value.f0, value.f1);
                    }

```

```
                })
                .keyBy(0)
                .window(TumblingEventTimeWindows.of(Time.seconds(10)))
                .reduce(new ReduceFunction<Tuple2<Long, Integer>>() {
                    @Override
                    public Tuple2<Long, Integer> reduce(Tuple2<Long, Int
eger> value1, Tuple2<Long, Integer> value2) throws Exception {
                        // 根据第一个元素分组，求第二个元素的累计和
                        return Tuple2.of(value1.f0, value1.f1 + value2.f
1);
                    }
                }).print();

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

AggregateFunction

与ReduceFunction相似，AggregateFunction也是基于中间状态计算结果的增量计算函数，相比ReduceFunction，AggregateFunction在窗口计算上更加灵活，但是实现稍微复杂，需要实现AggregateFunction接口，重写四个方法。其最大的优势就是中间结果的数据类型和最终的结果类型不依赖于输入的数据类型。关于AggregateFunction的源码，如下所示：

```
/**
*  @param <IN>  输入元素的数据类型
 * @param <ACC> 中间聚合结果的数据类型
 * @param <OUT> 最终聚合结果的数据类型
 */
@PublicEvolving
public interface AggregateFunction<IN, ACC, OUT> extends Function, Seria
lizable {

        /**
         * 创建一个新的累加器
         */
        ACC createAccumulator();

        /**
         * 将新的数据与累加器进行聚合，返回一个新的累加器
         */
        ACC add(IN value, ACC accumulator);

        /**
         从累加器中计算最终结果并返回
```

```
         */
        OUT getResult(ACC accumulator);

        /**
         * 合并两个累加器并返回结果
         */
        ACC merge(ACC a, ACC b);
}
```

具体使用代码案例如下：

```
public class AggregateFunctionExample {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getE
xecutionEnvironment().setParallelism(1);
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

        // 模拟数据源
        SingleOutputStreamOperator<Tuple3<Long, Integer, Long>> input =
env.fromElements(
                Tuple3.of(1L, 10, 1588491228L),
                Tuple3.of(1L, 15, 1588491229L),
                Tuple3.of(1L, 20, 1588491238L),
                Tuple3.of(1L, 25, 1588491248L),
                Tuple3.of(2L, 10, 1588491258L),
                Tuple3.of(2L, 30, 1588491268L),
                Tuple3.of(2L, 20, 1588491278L)).assignTimestampsAndWater
marks(new AscendingTimestampExtractor<Tuple3<Long, Integer, Long>>() {
            @Override
            public long extractAscendingTimestamp(Tuple3<Long, Integer,
Long> element) {
                return element.f2 * 1000;
            }
        });

        input.keyBy(0)
             .window(TumblingEventTimeWindows.of(Time.seconds(10)))
             .aggregate(new MyAggregateFunction()).print();
        env.execute("AggregateFunctionExample");

    }

    private static class MyAggregateFunction implements AggregateFunctio
n<Tuple3<Long, Integer, Long>,Tuple2<Long,Integer>,Tuple2<Long,Integer>>
 {
        /**
```

```
         * 创建一个累加器,初始化值
         * @return
         */
        @Override
        public Tuple2<Long, Integer> createAccumulator() {
            return Tuple2.of(0L,0);
        }

        /**
         *
         * @param value 输入的元素值
         * @param accumulator 中间结果值
         * @return
         */
        @Override
        public Tuple2<Long, Integer> add(Tuple3<Long, Integer, Long> val
ue, Tuple2<Long, Integer> accumulator) {
            return Tuple2.of(value.f0,value.f1 + accumulator.f1);
        }

        /**
         * 获取计算结果值
         * @param accumulator
         * @return
         */
        @Override
        public Tuple2<Long, Integer> getResult(Tuple2<Long, Integer> acc
umulator) {
            return Tuple2.of(accumulator.f0,accumulator.f1);
        }

        /**
         * 合并中间结果值
         * @param a 中间结果值a
         * @param b 中间结果值b
         * @return
         */
        @Override
        public Tuple2<Long, Integer> merge(Tuple2<Long, Integer> a, Tupl
e2<Long, Integer> b) {
            return Tuple2.of(a.f0,a.f1 + b.f1);
        }
    }
}
```

FoldFunction

FoldFunction定义了如何将窗口中的输入元素与外部的元素合并的逻辑,该接口已标记过时，建议用户使用AggregateFunction来替换使用FoldFunction。

```
public class FoldFunctionExample {

    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getE
xecutionEnvironment().setParallelism(1);
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

        // 模拟数据源
        SingleOutputStreamOperator<Tuple3<Long, Integer, Long>> input =
env.fromElements(
                Tuple3.of(1L, 10, 1588491228L),
                Tuple3.of(1L, 15, 1588491229L),
                Tuple3.of(1L, 20, 1588491238L),
                Tuple3.of(1L, 25, 1588491248L),
                Tuple3.of(2L, 10, 1588491258L),
                Tuple3.of(2L, 30, 1588491268L),
                Tuple3.of(2L, 20, 1588491278L)).assignTimestampsAndWater
marks(new AscendingTimestampExtractor<Tuple3<Long, Integer, Long>>() {
            @Override
            public long extractAscendingTimestamp(Tuple3<Long, Integer,
Long> element) {
                return element.f2 * 1000;
            }
        });

        input.keyBy(0)
             .window(TumblingEventTimeWindows.of(Time.seconds(10)))
             .fold("用户",new FoldFunction<Tuple3<Long, Integer, Long>,St
ring>() {
                 @Override
                 public String fold(String accumulator, Tuple3<Long, Int
eger, Long> value) throws Exception {
                     // 为第一个元素的值拼接一个"用户"字符串,进行输出
                      return accumulator + value.f0 ;
                 }
             }).print();

        env.execute("FoldFunctionExample");

    }
}
```

ProcessWindowFunction

前面提到的ReduceFunction和AggregateFunction都是基于中间状态实现增量计算的窗口函数。有些时候需要使用整个窗口的所有数据进行计算，比如求中位数和众数。另外，ProcessWindowFunction的Context对象可以访问窗口的一些元数据信息，比如窗口结束时间、水位线等。ProcessWindowsFunction能够更加灵活地支持基于窗口全部数据元素的结果计算。

在系统内部，由ProcessWindowFunction处理的窗口会将所有已分配的数据存储到ListState中，通过将数据收集起来且提供对于窗口的元数据及其他一些特性的访问和使用，应用场景比ReduceFunction和AggregateFunction更加广泛。关于ProcessWindowFunction抽象类的源码，如下所示：

```
/**
 * @param <IN> 输入的数据类型
 * @param <OUT> 输出的数据类型
 * @param <KEY> key的数据类型
 * @param <W> window的类型
 */
@PublicEvolving
public abstract class ProcessWindowFunction<IN, OUT, KEY, W extends Wind
ow> extends AbstractRichFunction {
        private static final long serialVersionUID = 1L;
        /**
         * 计算窗口数据，输出0个或多个元素
         * @param key 窗口的key
         * @param context 窗口的上下文
         * @param elements 窗口内的所有元素
         * @param out 输出元素的collector对象
         * @throws Exception
         */
        public abstract void process(KEY key, Context context, Iterable<
IN> elements, Collector<OUT> out) throws Exception;
        /**
         * 当窗口被销毁时，删除状态
         * @param context
         * @throws Exception
         */
        public void clear(Context context) throws Exception {}
        //context可以访问窗口的元数据信息.
        public abstract class Context implements java.io.Serializable {
        //返回当前被计算的窗口
                public abstract W window();
    // 返回当前processing time.
                public abstract long currentProcessingTime();
        // 返回当前event-time 水位线.
                public abstract long currentWatermark();
    //每个key和每个window的状态访问器
                public abstract KeyedStateStore windowState();
```

```
        // 每个key的global state的状态访问器.
                public abstract KeyedStateStore globalState();
                /**
                 * 向side output输出数据
                 * @param outputTag the {@code OutputTag}  side output 输出的标识.
                 * @param value 输出的数据.
                 */
                public abstract <X> void output(OutputTag<X> outputTag, X value);
        }
}
```

具体的使用案例如下：

```
public class ProcessWindowFunctionExample {

    public static void main(String[] args) throws Exception {

        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment().setParallelism(1);
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

        // 模拟数据源
        SingleOutputStreamOperator<Tuple3<Long, Integer, Long>> input = env.fromElements(
                Tuple3.of(1L, 10, 1588491228L),
                Tuple3.of(1L, 15, 1588491229L),
                Tuple3.of(1L, 20, 1588491238L),
                Tuple3.of(1L, 25, 1588491248L),
                Tuple3.of(2L, 10, 1588491258L),
                Tuple3.of(2L, 30, 1588491268L),
                Tuple3.of(2L, 20, 1588491278L)).assignTimestampsAndWatermarks(new AscendingTimestampExtractor<Tuple3<Long, Integer, Long>>() {
            @Override
            public long extractAscendingTimestamp(Tuple3<Long, Integer, Long> element) {
                return element.f2 * 1000;
            }
        });

        input.keyBy(t -> t.f0)
             .window(TumblingEventTimeWindows.of(Time.seconds(10)))
             .process(new MyProcessWindowFunction())
             .print();
    }

```

```
33
34      private static class MyProcessWindowFunction extends ProcessWindowFu
35  nction<Tuple3<Long, Integer, Long>,Tuple3<Long,String,Integer>,Long,Time
36  Window> {
37          @Override
38          public void process(
39                  Long aLong,
40                  Context context,
41                  Iterable<Tuple3<Long, Integer, Long>> elements,
42                  Collector<Tuple3<Long, String, Integer>> out) throws Exc
43  eption {
44              int count = 0;
45              for (Tuple3<Long, Integer, Long> in: elements) {
                    count++;
                }
                // 统计每个窗口数据个数，加上窗口输出
                out.collect(Tuple3.of(aLong,"" + context.window(),count));
            }
        }
    }
```

增量聚合函数和ProcessWindowFunction整合

ProcessWindowFunction提供了很强大的功能，但是唯一的缺点就是需要更大的状态存储数据。在很多时候，增量聚合的使用是非常频繁的，那么如何实现既支持增量聚合又支持访问窗口元数据的操作呢？可以将ReduceFunction和AggregateFunction与ProcessWindowFunction整合在一起使用。通过这种组合方式，分配给窗口的元素会立即被执行计算，当窗口触发时，会把聚合的结果传给ProcessWindowFunction，这样ProcessWindowFunction的process方法的Iterable参数被就只有一个值，即增量聚合的结果。

- ReduceFunction与ProcessWindowFunction组合

```
1   public class ReduceProcessWindowFunction {
2       public static void main(String[] args) throws Exception {
3
4           StreamExecutionEnvironment env = StreamExecutionEnvironment.getE
5   xecutionEnvironment().setParallelism(1);
6           env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
7
8           // 模拟数据源
9           SingleOutputStreamOperator<Tuple3<Long, Integer, Long>> input =
10  env.fromElements(
11                  Tuple3.of(1L, 10, 1588491228L),
12                  Tuple3.of(1L, 15, 1588491229L),
```

```
                Tuple3.of(1L, 20, 1588491238L),
                Tuple3.of(1L, 25, 1588491248L),
                Tuple3.of(2L, 10, 1588491258L),
                Tuple3.of(2L, 30, 1588491268L),
                Tuple3.of(2L, 20, 1588491278L)).assignTimestampsAndWaterm
arks(new AscendingTimestampExtractor<Tuple3<Long, Integer, Long>>() {
            @Override
            public long extractAscendingTimestamp(Tuple3<Long, Integer,
Long> element) {
                return element.f2 * 1000;
            }
        });

        input.map(new MapFunction<Tuple3<Long, Integer, Long>, Tuple2<Lo
ng, Integer>>() {
            @Override
            public Tuple2<Long, Integer> map(Tuple3<Long, Integer, Long>
 value) {
                return Tuple2.of(value.f0, value.f1);
            }
        })
             .keyBy(t -> t.f0)
             .window(TumblingEventTimeWindows.of(Time.seconds(10)))
             .reduce(new MyReduceFunction(),new MyProcessWindowFunction(
))
             .print();

        env.execute("ProcessWindowFunctionExample");
    }

    private static class MyReduceFunction implements ReduceFunction<Tupl
e2<Long, Integer>> {
        @Override
        public Tuple2<Long, Integer> reduce(Tuple2<Long, Integer> value1
, Tuple2<Long, Integer> value2) throws Exception {
            //增量求和
            return Tuple2.of(value1.f0,value1.f1 + value2.f1);
        }
    }

    private static class MyProcessWindowFunction extends ProcessWindowFu
nction<Tuple2<Long,Integer>,Tuple3<Long,Integer,String>,Long,TimeWindow>
 {
        @Override
        public void process(Long aLong, Context ctx, Iterable<Tuple2<Lon
g, Integer>> elements, Collector<Tuple3<Long, Integer, String>> out) thr
ows Exception {
            // 将求和之后的结果附带窗口结束时间一起输出
            out.collect(Tuple3.of(aLong,elements.iterator().next().f1,"w
```

```
indow_end" + ctx.window().getEnd()));
        }
    }
}
```

- AggregateFunction与ProcessWindowFunction组合

```
public class AggregateProcessWindowFunction {

    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getE
xecutionEnvironment().setParallelism(1);
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

        // 模拟数据源
        SingleOutputStreamOperator<Tuple3<Long, Integer, Long>> input =
env.fromElements(
                Tuple3.of(1L, 10, 1588491228L),
                Tuple3.of(1L, 15, 1588491229L),
                Tuple3.of(1L, 20, 1588491238L),
                Tuple3.of(1L, 25, 1588491248L),
                Tuple3.of(2L, 10, 1588491258L),
                Tuple3.of(2L, 30, 1588491268L),
                Tuple3.of(2L, 20, 1588491278L))
                .assignTimestampsAndWatermarks(new AscendingTimestampExt
ractor<Tuple3<Long, Integer, Long>>() {
                    @Override
                    public long extractAscendingTimestamp(Tuple3<Long, I
nteger, Long> element) {
                        return element.f2 * 1000;
                    }
                });

        input.keyBy(t -> t.f0)
                .window(TumblingEventTimeWindows.of(Time.seconds(10)))
                .aggregate(new MyAggregateFunction(),new MyProcessWindow
Function())
                .print();

        env.execute("AggregateFunctionExample");

    }

    private static class MyAggregateFunction implements AggregateFunctio
n<Tuple3<Long, Integer, Long>, Tuple2<Long, Integer>, Tuple2<Long, Integ
er>> {
```

```
        /**
         * 创建一个累加器,初始化值
         *
         * @return
         */
        @Override
        public Tuple2<Long, Integer> createAccumulator() {
            return Tuple2.of(0L, 0);
        }

        /**
         * @param value       输入的元素值
         * @param accumulator 中间结果值
         * @return
         */
        @Override
        public Tuple2<Long, Integer> add(Tuple3<Long, Integer, Long> val
ue, Tuple2<Long, Integer> accumulator) {
            return Tuple2.of(value.f0, value.f1 + accumulator.f1);
        }

        /**
         * 获取计算结果值
         *
         * @param accumulator
         * @return
         */
        @Override
        public Tuple2<Long, Integer> getResult(Tuple2<Long, Integer> acc
umulator) {
            return Tuple2.of(accumulator.f0, accumulator.f1);
        }

        /**
         * 合并中间结果值
         *
         * @param a 中间结果值a
         * @param b 中间结果值b
         * @return
         */
        @Override
        public Tuple2<Long, Integer> merge(Tuple2<Long, Integer> a, Tupl
e2<Long, Integer> b) {
            return Tuple2.of(a.f0, a.f1 + b.f1);
        }
    }

    private static class MyProcessWindowFunction extends ProcessWindowFu
nction<Tuple2<Long,Integer>,Tuple3<Long,Integer,String>,Long,TimeWindow>
```

```
        {
            @Override
            public void process(Long aLong, Context ctx, Iterable<Tuple2<Lon
        g, Integer>> elements, Collector<Tuple3<Long, Integer, String>> out) thr
        ows Exception {
                // 将求和之后的结果附带窗口结束时间一起输出
                out.collect(Tuple3.of(aLong,elements.iterator().next().f1,"w
        indow_end" + ctx.window().getEnd()));
            }
        }
    }
```

window 生命周期解读

生命周期图解

窗口从创建到执行窗口计算再到被清除，需要经过一系列的过程，这个过程就是窗口的生命周期。

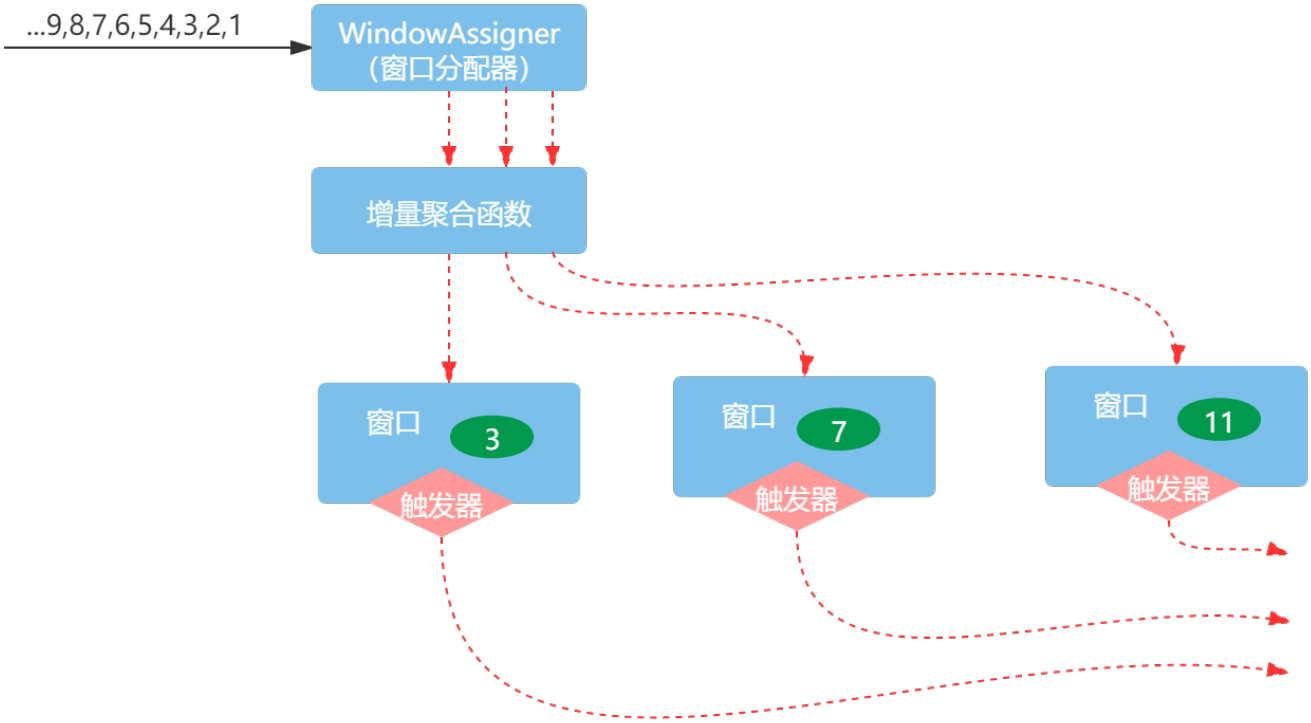
首先，当一个元素进入窗口算子之前，会由WindowAssigner分配该元素进入哪个或哪几个窗口，如果窗口不存在，则创建窗口。

其次，数据进入了窗口，这时要看有没有使用增量聚合函数，如果使用了增量聚合函数ReduceFunction或AggregateFunction，新加入窗口的元素会立即触发增量计算，计算的结果作为窗口的内容。如果没有使用增量聚合函数，则会将进入窗口的数据存储到ListState状态中，进一步等待窗口触发时，遍历窗口元素进行聚合计算。

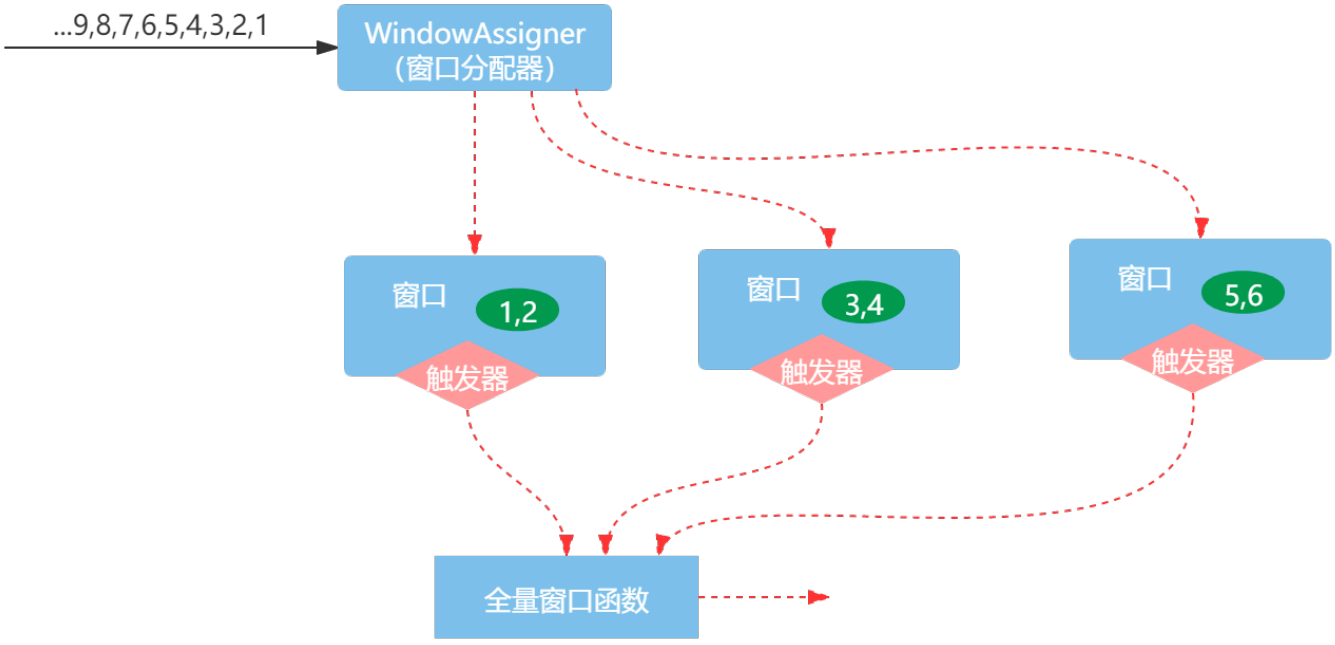
然后，每个元素在进入窗口之后会传递至该窗口的触发器，触发器决定了窗口何时被执行计算及何时需要清除自身和保存的内容。触发器可以根据已分配的元素或注册的计时器来决定某些特定时刻执行窗口计算或清除窗口内容。

最后，触发器成功触发之后的操作取决于使用的窗口函数，如果使用的是增量聚合函数，如ReduceFunction或AggregateFunction，则会直接输出聚合的结果。如果只包含一个全量窗口函数，如ProcessWindowFunction，则会作用窗口的所有元素，执行计算，输出结果。如果组合使用了ReduceFunction和ProcessWindowFunction，即组合使用了增量聚合窗口函数和全量窗口函数，全量窗口函数会作用于增量聚合函数的聚合值，然后再输出最终的结果。

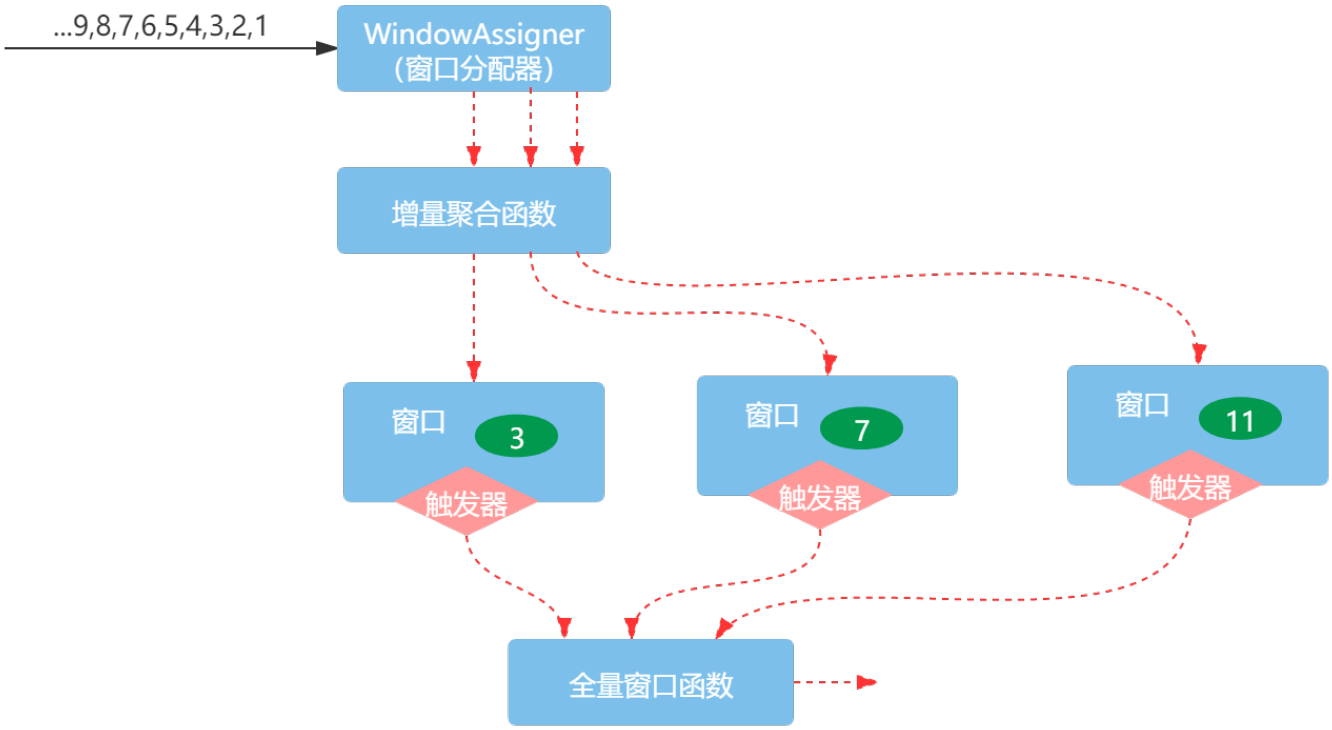
- 情况1：仅使用增量聚合窗口函数



- 情况2：仅使用全量窗口函数

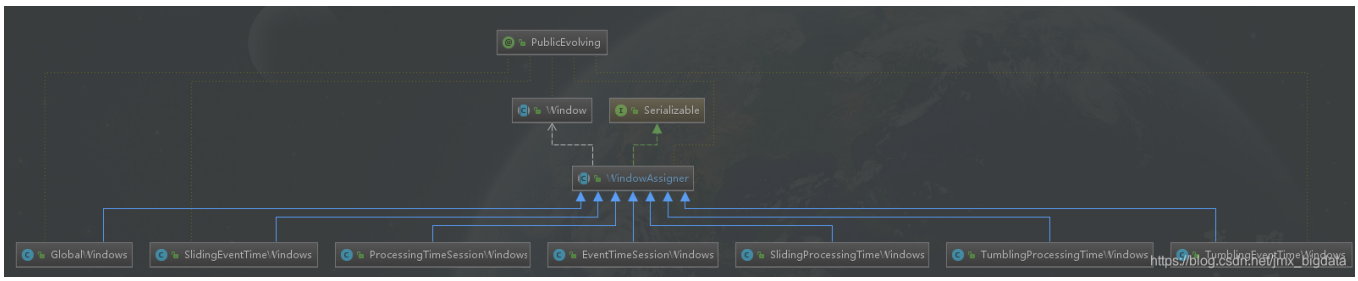


- 情况3：组合使用增量聚合窗口函数与全量窗口函数



分配器(Window Assigners)

WindowAssigner的作用是将输入的元素分配到一个或多个窗口，当WindowAssigner将第一个元素分配到窗口时，就会创建该窗口，所以一个窗口一旦被创建，窗口中必然至少有一个元素。Flink内置了很多WindowAssigners,本文主要讨论基于时间的WindowAssigners，这些分配器都继承了WindowAssigner抽象类。关于常用的分配器，上文已经做了详细解释。下面先来看一下继承关系图：



接下来，将会对WindowAssigner抽象类的源码进行分析，具体代码如下：

```
/**
 * WindowAssigner分配一个元素到0个或多个窗口
 * 在一个窗口算子内部，元素是按照key进行分组的(使用KeyedStream)，
 * 相同key和window的元素集合称之为一个pane(格子)
 * @param <T> 要分配元素的数据类型
 * @param <W> window的类型:TimeWindow、GlobalWindow
 */
@PublicEvolving
public abstract class WindowAssigner<T, W extends Window> implements Ser
ializable {
        private static final long serialVersionUID = 1L;
```

```
        /**
         * 返回一个向其分配元素的窗口集合
         * @param element 待分配的元素
         * @param timestamp 元素的时间戳
         * @param context WindowAssignerContext对象
         * @return
         */
        public abstract Collection<W> assignWindows(T element, long time
stamp, WindowAssignerContext context);
        /**
         * 返回一个与该WindowAssigner相关的默认trigger(触发器)
         * @param env 执行环境
         * @return
         */
        public abstract Trigger<T, W> getDefaultTrigger(StreamExecutionE
nvironment env);

        /**
         * 返回一个窗口序列化器
         * @param executionConfig
         * @return
         */
        public abstract TypeSerializer<W> getWindowSerializer(ExecutionC
onfig executionConfig);
        /**
         * 如果元素是基于event time分配到窗口的，则返回true
         * @return
         */
        public abstract boolean isEventTime();
        /**
         * 该Context允许访问当前的处理时间processing time
         */
        public abstract static class WindowAssignerContext {

                /**
                 * 返回当前的处理时间
                 */
                public abstract long getCurrentProcessingTime();
        }
}
```

触发器(Triggers)

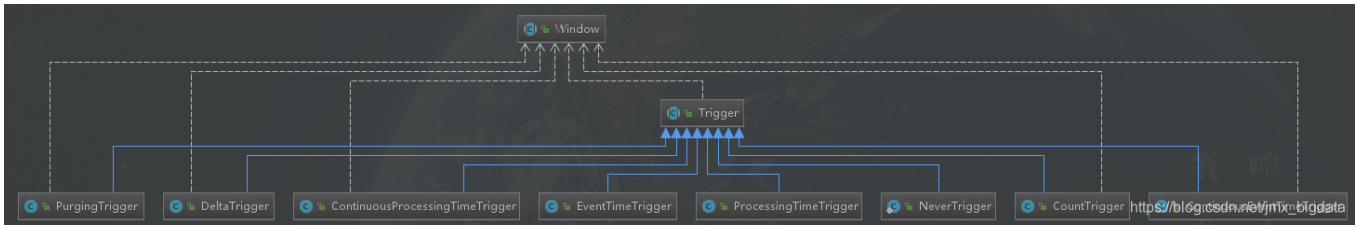
数据接入窗口后，窗口是否触发WindowFunciton计算，取决于窗口是否满足触发条件。Triggers就是决定窗口何时触发计算并输出结果的条件，Triggers可以根据时间或者具体的数据条件进行触发，比如进入窗口元素的个数或者进入窗口的某些特定的元素值等。

前面讨论的内置WindowAssigner都有各自默认的触发器，当使用的是Processing Time时，则当处理时间超过窗口结束时间时会被触发。当使用Event Time时，当水位线超过窗口结束时间时会被触发。

Flink在内部提供很多内置的触发器，常用的主要有EventTimeTrigger、ProcessTimeTrigger以及CountTrigger等。每种每种触发器都对应于不同的Window Assigner，例如Event Time类型的Windows对应的触发器是EventTimeTrigger，其基本原理是判断当前的Watermark是否超过窗口的EndTime，如果超过则触发对窗口内数据的计算，反之不触发计算。关于上面分析的内置WindowAssigner的默认trigger，可以从各自的源码中看到，具体罗列如下：

分配器	对应的源码
TumblingEventTimeWindows	public Trigger<Object, TimeWindow> getDefaultTrigger(StreamExecutionEnvironment env) { retu EventTimeTrigger.create(); }
TumblingProcessingTimeWindows	public Trigger<Object, TimeWindow> getDefaultTrigger(StreamExecutionEnvironment env) { retu ProcessingTimeTrigger.create(); }
SlidingEventTimeWindows	public Trigger<Object, TimeWindow> getDefaultTrigger(StreamExecutionEnvironment env) { retu EventTimeTrigger.create(); }
SlidingProcessingTimeWindows	public Trigger<Object, TimeWindow> getDefaultTrigger(StreamExecutionEnvironment env) { retu ProcessingTimeTrigger.create(); }
EventTimeSessionWindows	public Trigger<Object, TimeWindow> getDefaultTrigger(StreamExecutionEnvironment env) { retu EventTimeTrigger.create(); }
ProcessingTimeSessionWindows	public Trigger<Object, TimeWindow> getDefaultTrigger(StreamExecutionEnvironment env) { retu ProcessingTimeTrigger.create(); }
GlobalWindows	public Trigger<Object, GlobalWindow> getDefaultTrigger(StreamExecutionEnvironment env) { return new Nev

这些Trigger都继承了Trigger抽象类，具体的继承关系，如下图：



关于这些内置的Trigger的具体解释如下：

Trigger	解释
EventTimeTrigger	当前的Watermark是否超过窗口的EndTime，如果超过则触发对窗口内
ProcessTimeTrigger	当前的Processing Time是否超过窗口的EndTime，如果超过则触发对窗
ContinuousEventTimeTrigger	根据间隔时间周期性触发窗口或者Window的结束时间小于当前EventTir
ContinuousProcessingTimeTrigger	根据间隔时间周期性触发窗口或者Window的结束时间小于当前Process
CountTrigger	根据窗口的数据条数是否超过设定的阈值确定是否触发窗口计算；
DeltaTrigger	根据窗口的数据计算出来的Delta指标是否超过指定的阈值，判断是否触
PurgingTrigger	可以将任意触发器作为参数转换为Purge类型触发器，计算完成后数据将

关于抽象类Trigger的源码解释如下：

```
/**
 * @param <T> 元素的数据类型
 * @param <W> Window的类型
 */
@PublicEvolving
public abstract class Trigger<T, W extends Window> implements Serializab
le {

        private static final long serialVersionUID = -4104633972991191369
L;
        /**
         * 每个元素被分配到窗口时都会调用该方法，返回一个TriggerResult枚举
         * 该枚举包含很多触发的类型：CONTINUE、FIRE_AND_PURGE、FIRE、PURGE
         *
         * @param element   进入窗口的元素
         * @param timestamp 进入窗口元素的时间戳
         * @param window    窗口
         * @param ctx       上下文对象，可以注册计时器(timer)回调函数
         * @return
         * @throws Exception
         */
        public abstract TriggerResult onElement(T element, long timestam
p, W window, TriggerContext ctx) throws Exception;
        /**
         * 当使用TriggerContext注册的processing-time计时器被触发时,会调用该方法
         *
         * @param time   触发计时器的时间戳
         * @param window 计时器触发的window
         * @param ctx    上下文对象，可以注册计时器(timer)回调函数
```

```
         * @return
         * @throws Exception
         */
        public abstract TriggerResult onProcessingTime(long time, W wind
ow, TriggerContext ctx) throws Exception;
        /**
         * 当使用TriggerContext注册的event-time计时器被触发时,会调用该方法
         *
         * @param time   触发计时器的时间戳
         * @param window 计时器触发的window
         * @param ctx    上下文对象，可以注册计时器(timer)回调函数
         * @return
         * @throws Exception
         */
        public abstract TriggerResult onEventTime(long time, W window, T
riggerContext ctx) throws Exception;
        /**
         * 如果触发器支持合并触发器状态，将返回true
         *
         * @return
         */
        public boolean canMerge() {
                return false;
        }

        /**
         * 当多个窗口被合并成一个窗口时，会调用该方法
         *
         * @param window 合并之后的window
         * @param ctx    上下文对象，可以注册计时器回调函数，也可以访问状态
         * @throws Exception
         */
        public void onMerge(W window, OnMergeContext ctx) throws Excepti
on {
                throw new UnsupportedOperationException("This trigger do
es not support merging.");
        }
        /**
         * 清除所有Trigger持有的窗口状态
         * 当窗口被销毁时，调用该方法
         *
         * @param window
         * @param ctx
         * @throws Exception
         */
        public abstract void clear(W window, TriggerContext ctx) throws
Exception;
        /**
         * Context对象，传给Trigger的方法参数中，用于注册计时器回调函数和处理状态
```

```
         */
        public interface TriggerContext {
                // 返回当前处理时间
                long getCurrentProcessingTime();
                MetricGroup getMetricGroup();
                // 返回当前水位线时间戳
                long getCurrentWatermark();
                // 注册一个processing-time的计时器
                void registerProcessingTimeTimer(long time);
                // 注册一个EventTime计时器
                void registerEventTimeTimer(long time);
                //  删除一个processing-time的计时器
                void deleteProcessingTimeTimer(long time);
                // 删除一个EventTime计时器
                void deleteEventTimeTimer(long time);
                /**
                 * 提取状态当前Trigger的窗口和Key的状态
                 */
                <S extends State> S getPartitionedState(StateDescriptor<
S, ?> stateDescriptor);

                // 与getPartitionedState功能相同，该方法已被标记过时
                @Deprecated
                <S extends Serializable> ValueState<S> getKeyValueState(
String name, Class<S> stateType, S defaultState);
                // 同getPartitionedState功能，该方法已被标记过时
                @Deprecated
                <S extends Serializable> ValueState<S> getKeyValueState(
String name, TypeInformation<S> stateType, S defaultState);
        }
        // TriggerContext的扩展
        public interface OnMergeContext extends TriggerContext {
                // 合并每个window的状态，状态必须支持合并
                <S extends MergingState<?, ?>> void mergePartitionedStat
e(StateDescriptor<S, ?> stateDescriptor);
        }
}
```

上面的源码可以看出,每当触发器调用时，会产生一个TriggerResult对象，该对象是一个枚举类，其包括的属性决定了作用在窗口上的操作是什么。总共有四种行为：CONTINUE、FIRE_AND_PURGE、FIRE、PURGE，关于每种类型的具体含义，我们先看一下TriggerResult源码：

```
/**
 * 触发器方法的结果类型，决定在窗口上执行什么操作，比如是否调用window function
 * 或者是否需要销毁窗口
```

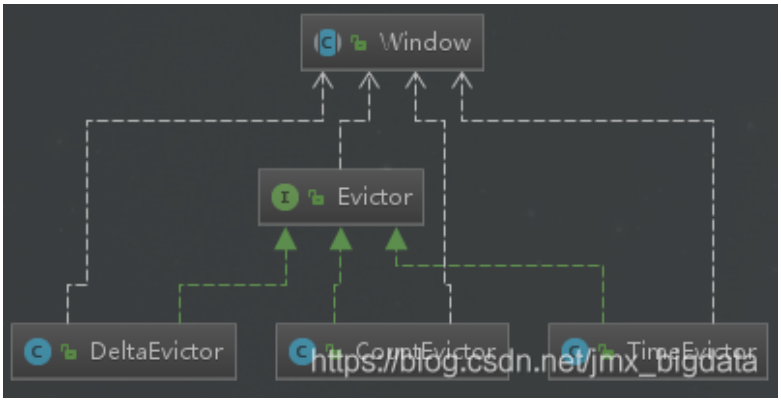
```
 * 注意：如果一个Trigger返回的是FIRE或者FIRE_AND_PURGE，但是窗口中没有任何元素，则窗口函数不会被调用
 */
public enum TriggerResult {

        // 什么都不做，当前不触发计算，继续等待
        CONTINUE(false, false),

        // 执行 window function，输出结果，之后清除所有状态
        FIRE_AND_PURGE(true, true),

        // 执行 window function，输出结果，窗口不会被清除，数据继续保留
        FIRE(true, false),

        // 清除窗口内部数据，但不触发计算
        PURGE(false, true);

}
```

清除器(Evictors)

Evictors是一个可选的组件，其主要作用是对进入WindowFuction前后的数据进行清除处理。Flink内置了三种Evictors：分别为CountEvictor、DeltaEvictor、TimeEvitor。如果用户不指定Evictors，也不会有默认值。

- **CountEvictor**：保持在窗口中具有固定数量的元素，将超过指定窗口元素数量的数据在窗口计算前剔除；
- **DeltaEvictor**：通过定义DeltaFunction和指定threshold，并计算Windows中的元素与最新元素之间的Delta大小，如果超过threshold则将当前数据元素剔除；
- **TimeEvictor**：通过指定时间间隔，将当前窗口中最新元素的时间减去Interval，然后将小于该结果的数据全部剔除，其本质是将具有最新时间的数据选择出来，删除过时的数据。

Evictors继承关系图如下：



关于Evictors接口的源码，如下：

```
/**
 * 在WindowFunction计算之前或者之后进行清除窗口元素
 * @param <T> 元素的数据类型
 * @param <W> 窗口类型
 */
@PublicEvolving
public interface Evictor<T, W extends Window> extends Serializable {
        /**
         * 选择性剔除元素，在windowing function之前调用
         * @param elements 窗口中的元素
         * @param size  窗口中元素个数
         * @param window 窗口
         * @param evictorContext
         */
        void evictBefore(Iterable<TimestampedValue<T>> elements, int siz
e, W window, EvictorContext evictorContext);
        /**
         * 选择性剔除元素，在windowing function之后调用
         * @param elements 窗口中的元素.
         * @param size 窗口中元素个数.
         * @param window 窗口
         * @param evictorContext
         */
        void evictAfter(Iterable<TimestampedValue<T>> elements, int size
, W window, EvictorContext evictorContext);
        // 传递给Evictor方法参数的值
        interface EvictorContext {
                // 返回当前processing time
                long getCurrentProcessingTime();
                MetricGroup getMetricGroup();
                // 返回当前的水位线时间戳
                long getCurrentWatermark();
        }
}
```

小结

本文首先给出了窗口使用的快速入门，介绍了窗口的基本概念、分类及简单使用。然后对Flink内置的Window Assigner进行了一一解读，并给出了图解与使用的代码片段。接着对Flink的Window Function进行介绍，包括窗口函数的分类及详细使用案例。最后分析了Window生命周期所涉及的组件，并对每个组件的源码进行分析。