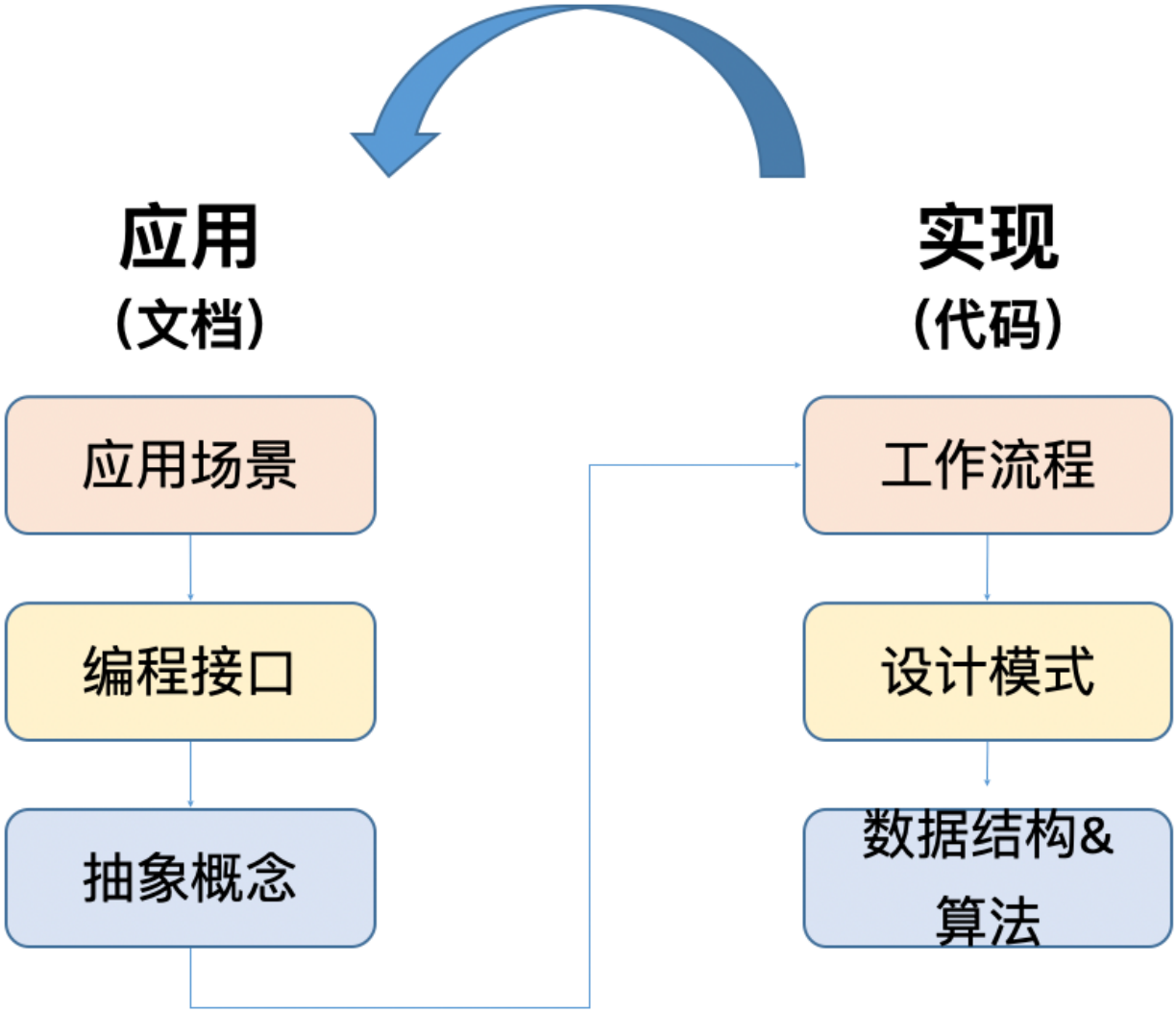
Flink 窗口的应用与实现

作者 | 张俊（OPPO大数据平台研发负责人）
整理 | 祝尚（Flink 社区志愿者）
校对 | 邹志业（Flink 社区志愿者）

> 摘要：本文根据 Apache Flink 系列直播整理而成，由 Apache Flink Contributor、OPPO 大数据平台研发负责人张俊老师分享。主要内容如下：
>
> 1. 整体思路与学习路径
> 2. 应用场景与编程模型
> 3. 工作流程与实现机制
>
> Tips：点击「下方链接」可查看更多数仓系列直播视频～
>
> 数仓系列直播：
> https://ververica.cn/developers/flink-training-course-data-warehouse/

整体思路与学习路径



当我们碰到一项新的技术时，我们应该怎样去学习并应用它呢？在我个人看来，有这样一个学习的路径，应该把它拆成应用和实现两块。首先应该从它的应用入手，然后再深入它的实现。

应用主要分为三个部分，首先应该了解它的应用场景，比如窗口的一些使用场景。然后，进一步地我们去了解它的编程接口，最后再深入了解它的一些抽象概念。因为一个框架或一项技术，肯定有它的编程接口和抽象概念来组成它的编程模型。我们可以通过查看文档的方式来熟悉它的应用。在对应用这三个部分有了初步的了解后，我们就可以通过阅读代码的方式去了解它的一些实现了。

实现部分也分三个阶段，首先从工作流程开始，可以通过 API 层面不断的下钻来了解它的工作流程。接下来是它整体的设计模式，通常对一些框架来说，如果能构建一个比较成熟的生态，一定是在设计模式上有一些独特的地方，使其有一个比较好的扩展性。最后是它的数据结构和算法，因为为了能够处理海量数据并达到高性能，它的数据结构和算法一定有独到之处。我们可以做些深入了解。

以上大概是我们学习的一个路径。从实现的角度可以反哺到应用上来，通常在应用当中，刚接触某个概念的时候会有一些疑惑。当我们对实现有一些了解之后，应用中的这些疑惑就会迎刃而解。

为什么要关心实现

举个例子：

```
DataStream<Tuple2<String, Long>> input = ...;

input
    .keyBy(<key selector>)
    .window(<window assigner>)
    .reduce(new ReduceFunction<Tuple2<String, Long>> {
      public Tuple2<String, Long> reduce(Tuple2<String, Long> v1, Tuple2<String, Long> v2) {
        return new Tuple2<>(v1.f0, v1.f1 + v2.f1);
      }
    });
```

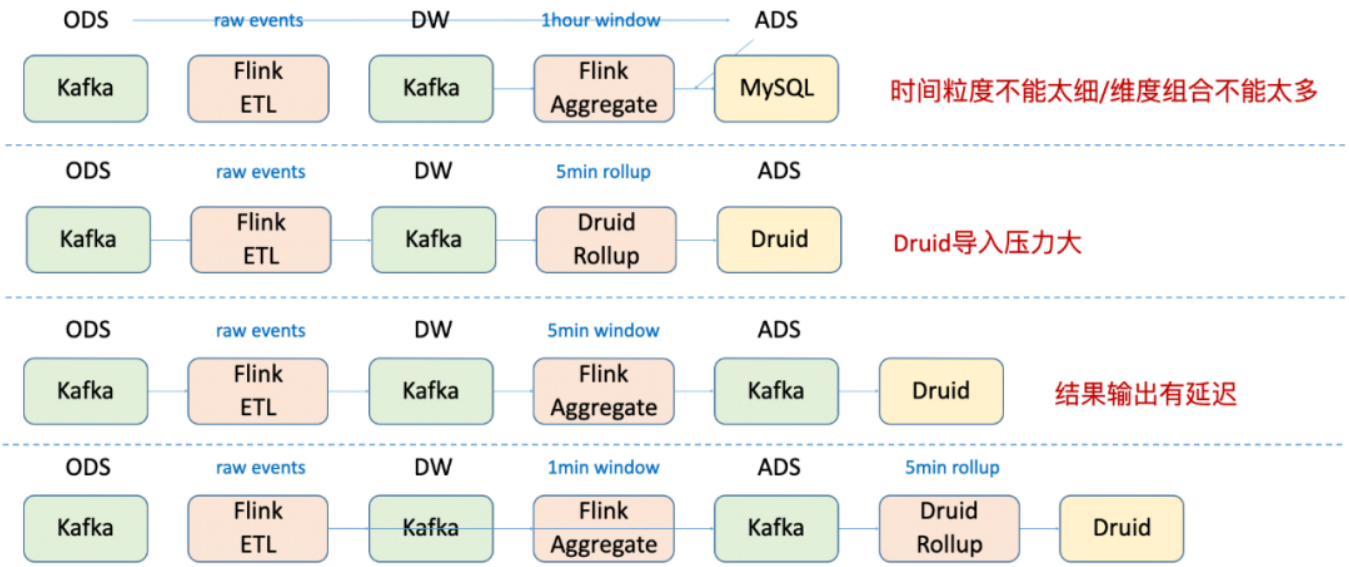
看了这个例子我们可能会有些疑惑：

- ReduceFunction 为什么不用计算每个 key 的聚合值?
- 当 key 基数很大时，如何有效地触发每个 key 窗口计算?
- 窗口计算的中间结果如何存储，何时被清理?
- 窗口计算如何容忍 late data ?

当你了解了实现部分再回来看应用这部分，可能就有种醍醐灌顶的感觉。

应用场景与编程模型

实时数仓的典型架构



■ **第一种最简单架构**，ODS 层的 Kafka 数据经过 Flink 的 ETL 处理后写入 DW 层的 Kafka，再通过 Flink 聚合写入 ADS 层的 MySQL 中，做这样一个实时报表展现。

缺点：由于 MySQL 存储数据有限，所以聚合的时间粒度不能太细，维度组合不能太多。

■ **第二种架构**相对于第一种引入了 OLAP 引擎，同时也不用 Flink 来做聚合，通过 Druid 的 Rollup 来做聚合。

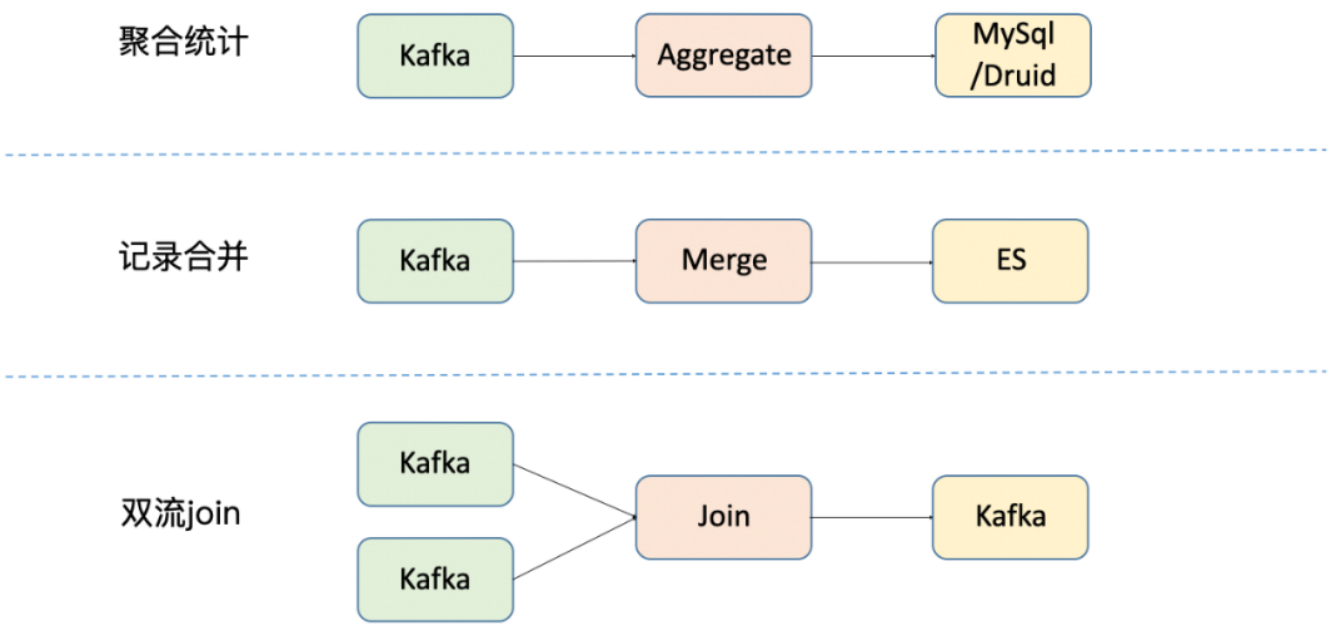
缺点：因为 Druid 是一个存储和查询引擎，不是计算引擎。当数据量巨大时，比如每天上百亿、千亿的数据量，会加剧 Druid 的导入压力。

■ **第三种架构**在第二种基础上，采用 Flink 来做聚合计算写入 Kafka，最终写入 Druid。

缺点：当窗口粒度比较长时，结果输出会有延迟。

■ 第四种架构在第三种基础上，结合了 Flink 聚合和 Druid Rollup。Flink 可以做轻度的聚合，Druid 做 Rollup 的汇总。好处是 Druid 可以实时看到 Flink 的聚合结果。

Window 应用场景

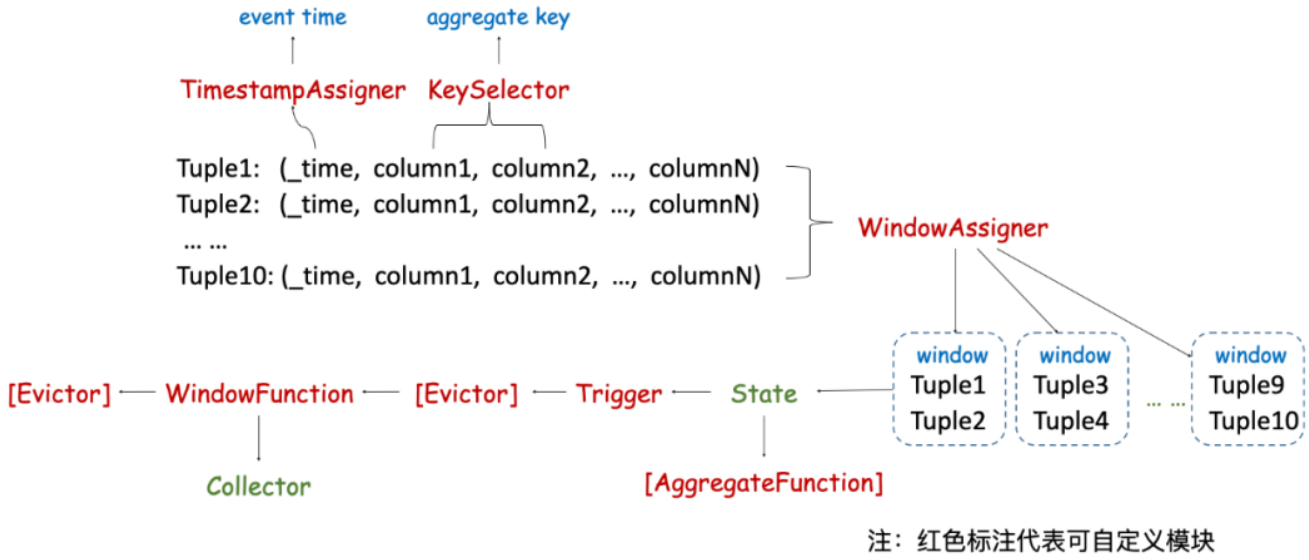


■ **聚合统计**：从 Kafka 读取数据，根据不同的维度做1分钟或5分钟的聚合计算，然后结果写入 MySQL 或 Druid 中。

■ **记录合并**：对多个 Kafka 数据源在一定的窗口范围内做合并，结果写入 ES。例如：用户的一些行为数据，针对每个用户，可以对其行为做一定的合并，减少写入下游的数据量，降低 ES 的写入压力。

■ **双流 join**：针对双流 join 的场景，如果全量 join 的话，成本开销会非常大。所以就要考虑基于窗口来做 join。

Window 抽象概念



■ **TimestampAssigner:** 时间戳分配器，假如我们使用的是 EventTime 时间语义，就需要通过 TimestampAssigner 来告诉Flink 框架，元素的哪个字段是事件时间，用于后面的窗口计算。

■ **KeySelector**：Key 选择器，用来告诉 Flink 框架做聚合的维度有哪些。

■ **WindowAssigner**：窗口分配器，用来确定哪些数据被分配到哪些窗口。

■ **State**：状态，用来存储窗口内的元素，如果有 AggregateFunction，则存储的是增量聚合的中间结果。

■ **AggregateFunction（可选）**：增量聚合函数，主要用来做窗口的增量计算，减轻窗口内 State 的存储压力。

■ **Trigger**：触发器，用来确定何时触发窗口的计算。

■ **Evictor（可选）**：驱逐器，用于在窗口函数计算之前（后）对满足驱逐条件的数据做过滤。

■ **WindowFunction**：窗口函数，用来对窗口内的数据做计算。

■ **Collector**：收集器，用来将窗口的计算结果发送到下游。

上图中红色部分都是可以自定义的模块，通过自定义这些模块的组合，我们可以实现高级的窗口应用。同时 Flink 也提供了一些内置的实现，可以用来做一些简单应用。

Window 编程接口

```
stream
  .assignTimestampsAndWatermarks(…)     <-    TimestampAssigner
  .keyBy(...)                           <-    KeySelector
  .window(...)                          <-    WindowAssigner
  [.trigger(...)]                       <-    Trigger
  [.evictor(...)]                       <-    Evictor
  .reduce/aggregate/process()           <-    Aggregate/Window function
```

首先我们先指定时间戳和 Watermark 如何生成；然后选择需要聚合的维度的 Key；再选择一个窗口和选择用什

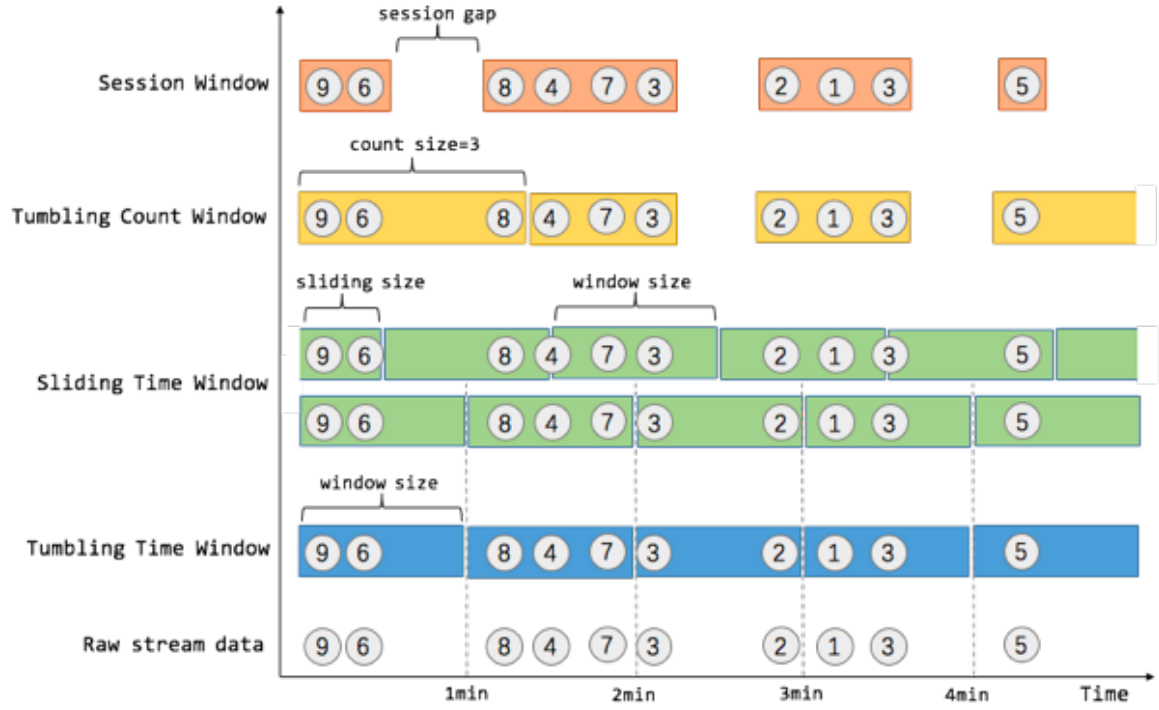
么样的触发器来触发窗口计算，以及选择驱逐器做什么样的过滤；最后确定窗口应该做什么样计算。

下面是一个示例：

```
DataStream<Tuple3<Long, TimeWindow, Integer>> result = orders
        .assignTimestampsAndWatermarks(new BoundedOutOfOrdernessTimestampExtractor<Order>(Time.seconds(5)) {
            public long extractTimestamp(Order element) { return element.ts.getTime(); }})
        .keyBy(new KeySelector<Order, Long>(){
            public Long getKey(Order value) { return value.user; }})
        .window(TumblingEventTimeWindows.of(Time.minutes(5)))
        .trigger(CountTrigger.of(1))
        .aggregate(new MyAggregateFunction(), new MyProcessWindowFunction());
```

接下来我们详细看下每个模块。

■ Window Assigner



总结一下主要有3类窗口：

- Time Window
- Count Window
- Custom Window

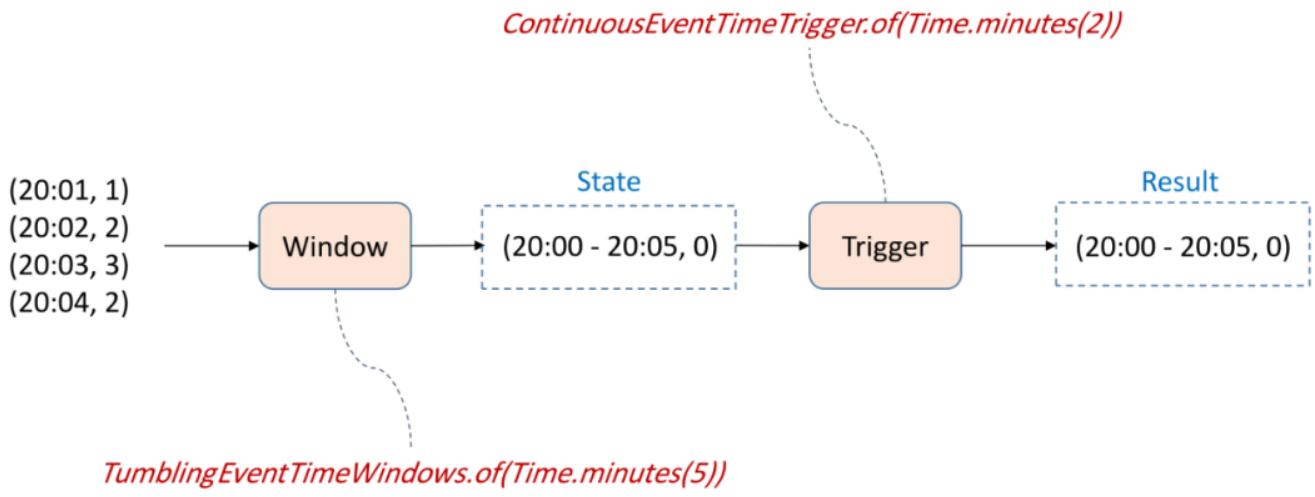
■ Window Trigger

Trigger 是一个比较重要的概念，用来确定窗口什么时候触发计算。

Flink 内置了一些 Trigger 如下图：

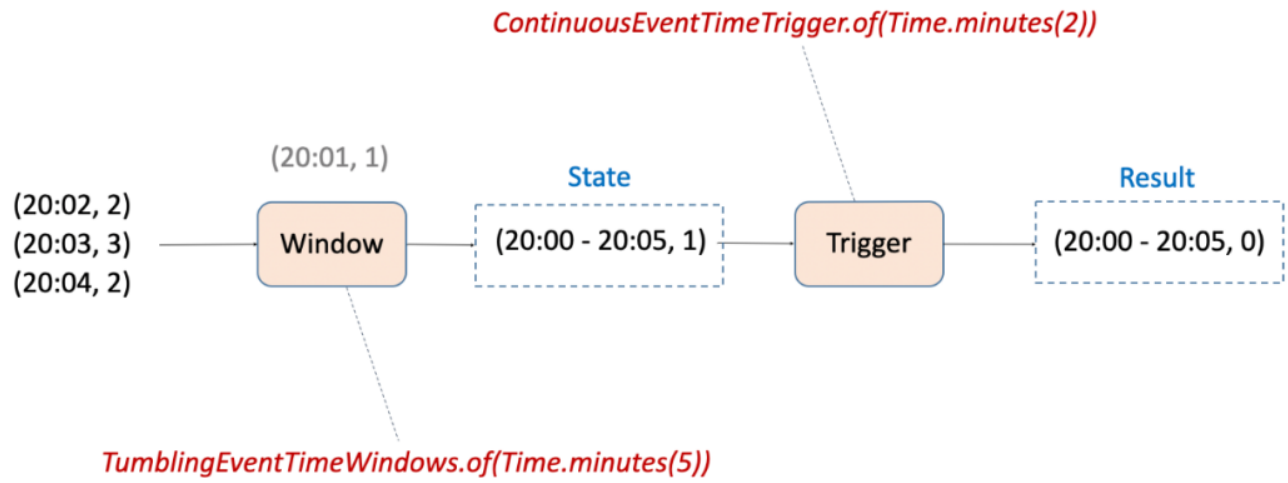
内置Trigger	说明
ProcessingTimeTrigger	一次触发，machine time大于窗口结束时间时触发
EventTimeTrigger	一次触发，watermark大于窗口结束时间时触发
ContinuousProcessingTimeTrigger	多次触发，基于processing time的固定时间间隔
ContinuousEventTimeTrigger	多次触发，基于event time的固定时间间隔
CountTrigger	多次触发，基于element的固定条数
DeltaTrigger	多次触发，当前element与上次触发trigger的element做delta计算，超过threshold时触发
PurgingTrigger	trigger wrapper，当nested trigger触发时，额外会清理窗口当前的中间状态

■ Trigger 示例

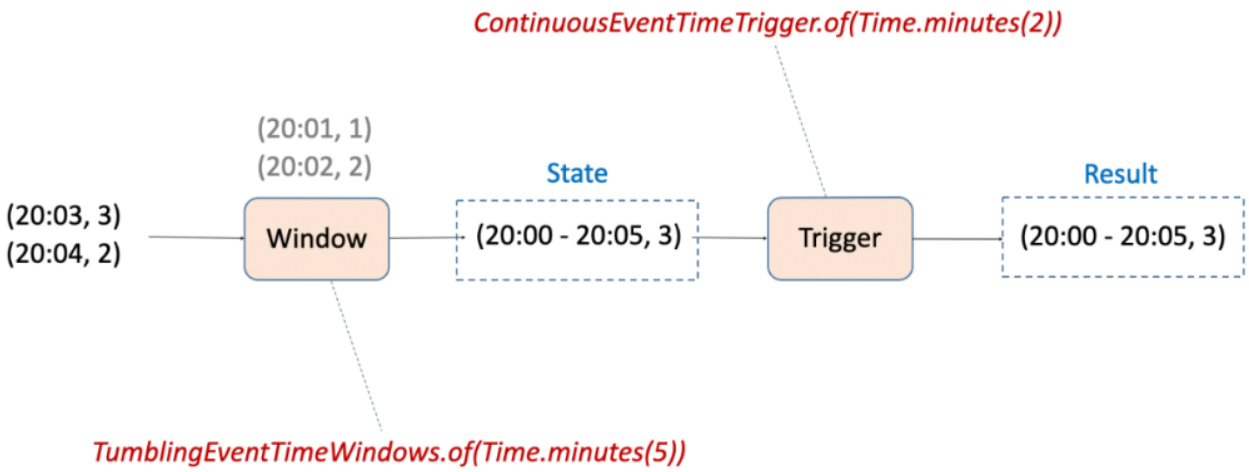


假如我们定义一个5分钟的基于 EventTime 的滚动窗口，定义一个每2分触发计算的 Trigger，有4条数据事件时间分别是20:01、20:02、20:03、20:04，对应的值分别是1、2、3、2，我们要对值做 Sum 操作。

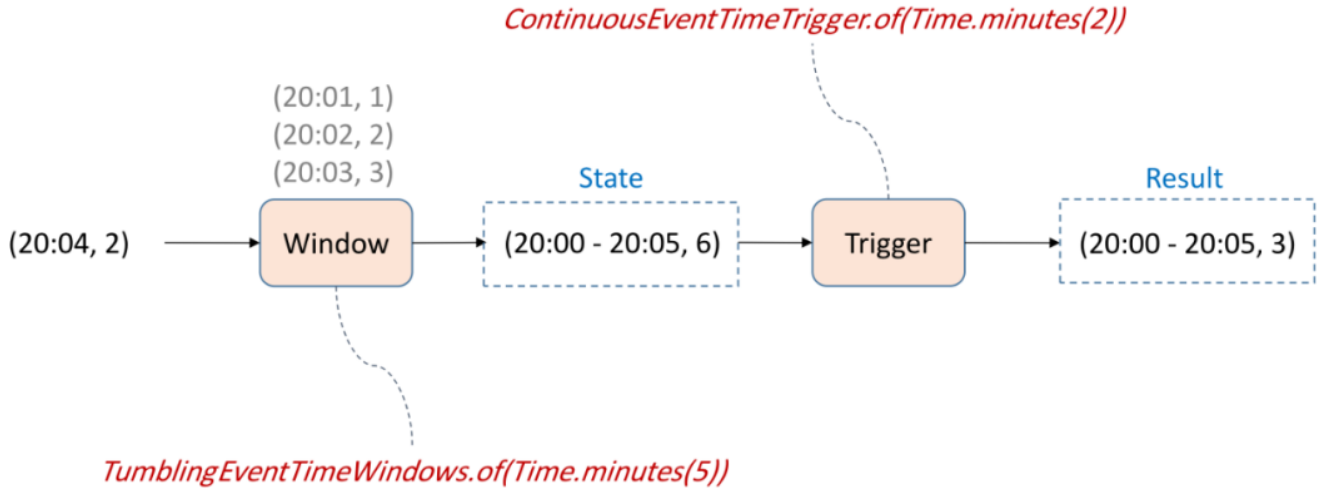
初始时，State 和 Result 中的值都为0。



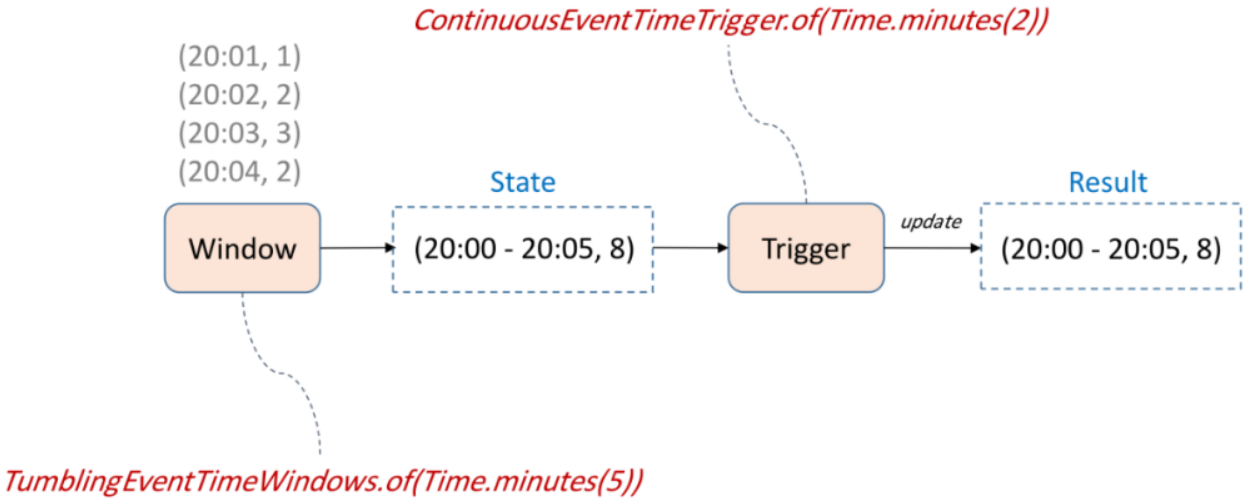
当第一条数据在20:01进入窗口时，State 的值为1，此时还没有到达 Trigger 的触发时间。



第二条数据在20:02进入窗口，State 中的值为1+2=3，此时达到2分钟满足 Trigger 的触发条件，所以 Result 输出结果为3。

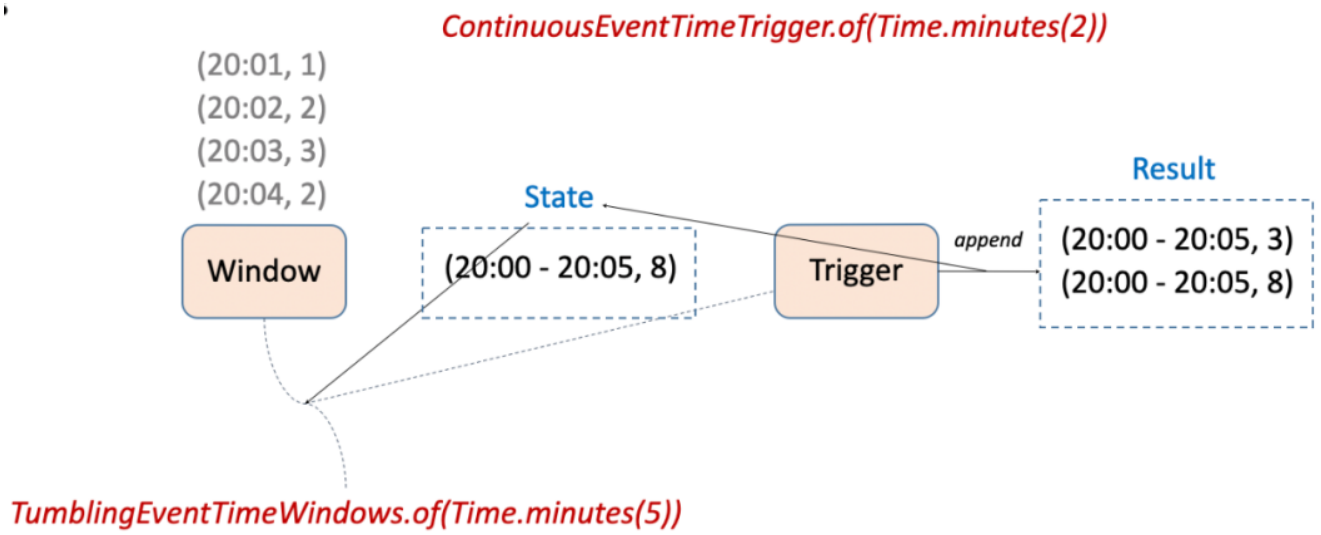


第三条数据在20:03进入窗口，State 中的值为3+3 = 6，此时未达到 Trigger 触发条件，没有结果输出。



第四条数据在20:04进入窗口，State中的值更新为6+2=8，此时又到了2分钟达到了 Trigger 触发时间，所以输出结果为8。如果我们把结果输出到支持 update 的存储，比如 MySQL，那么结果值就由之前的3更新成了8。

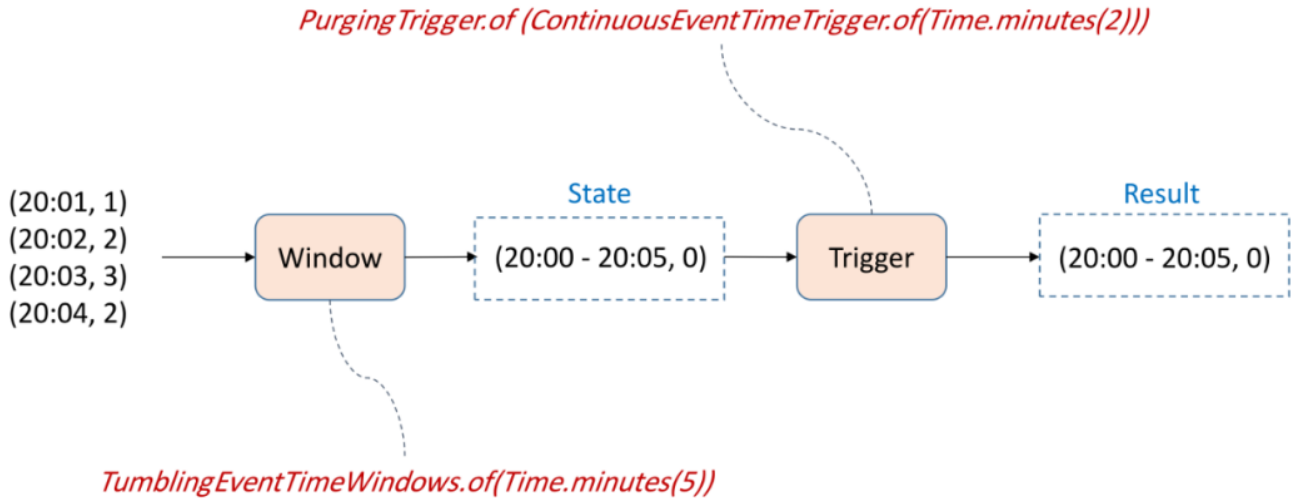
■ 问题：如果 Result 只能 append?



如果 Result 不支持 update 操作，只能 append 的话，则会输出2条记录，在此基础上再做计算处理就会引起错误。

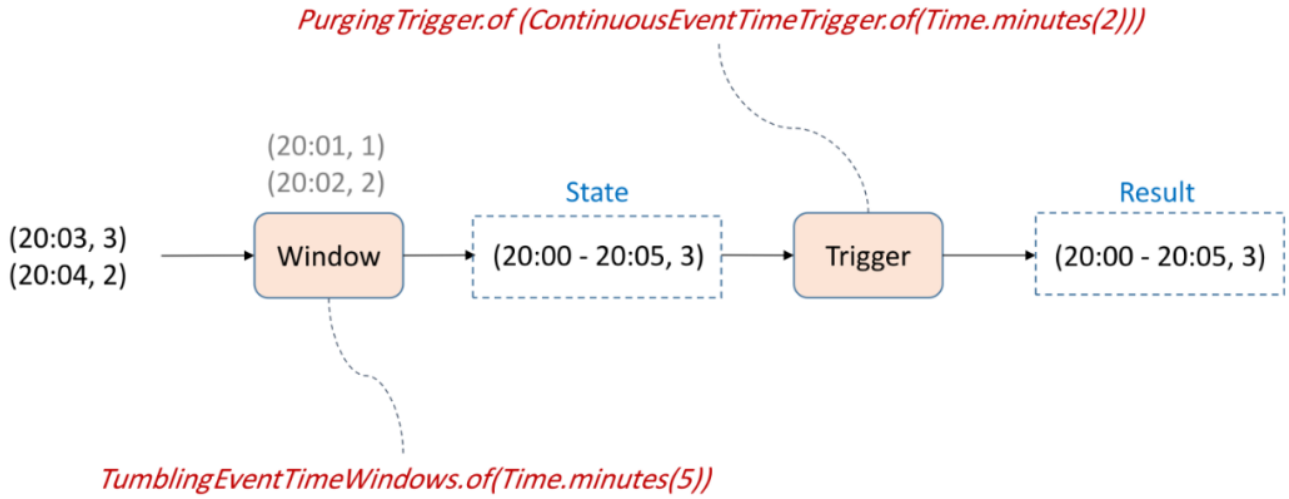
这样就需要 PurgingTrigger 来处理上面的问题。

■ **PurgingTrigger 的应用**

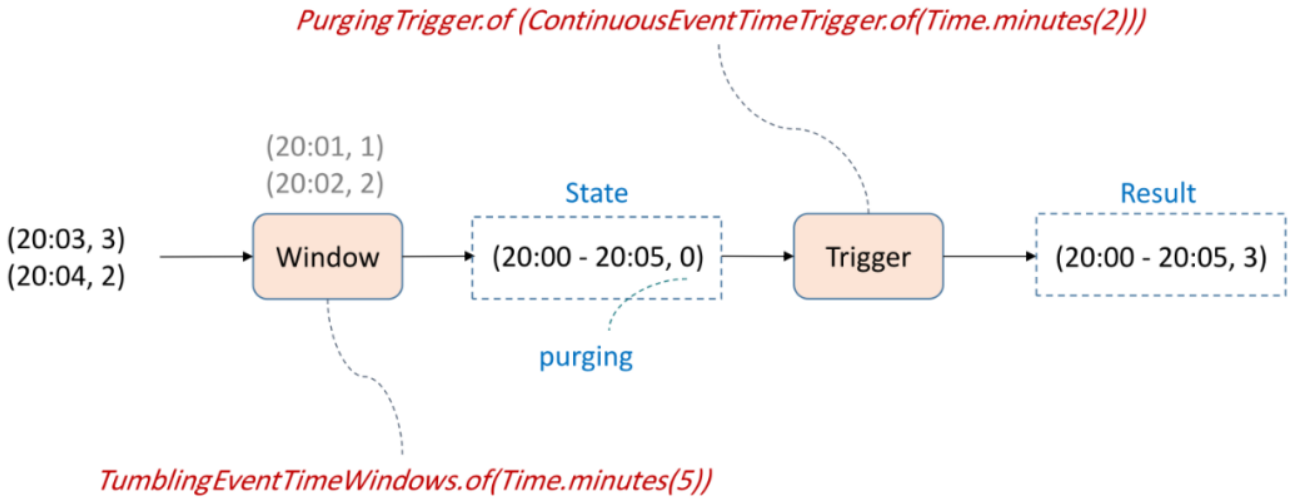


和上面的示例一样，唯一的不同是在 ContinuousEventTimeTrigger 外面包装了一个 PurgingTrigger，其作用是在 ContinuousEventTimeTrigger 触发窗口计算之后将窗口的 State 中的数据清除。

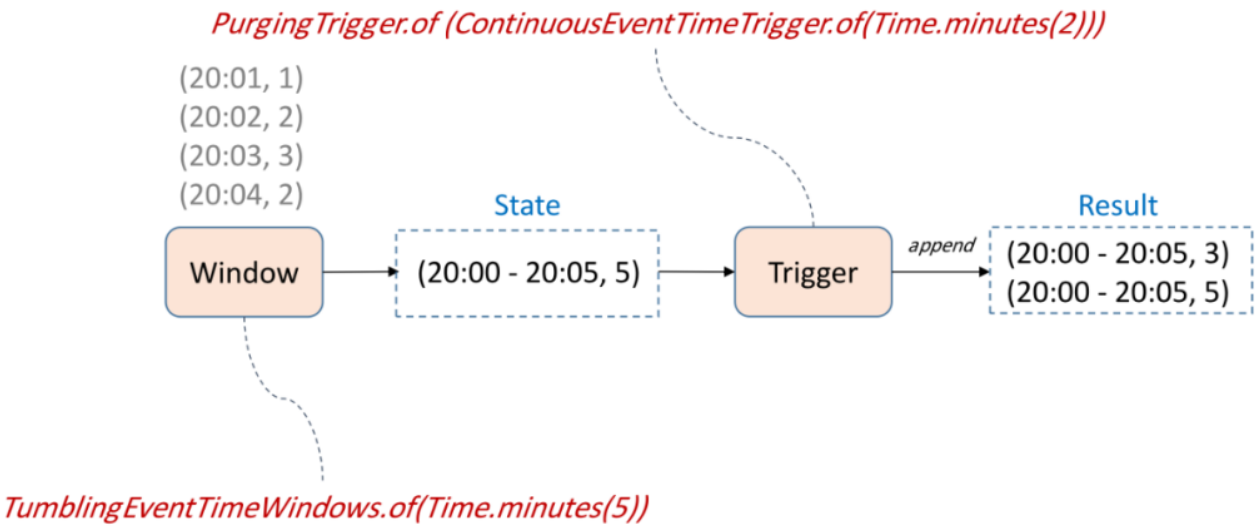
再看下流程：



前两条数据先后于20:01和20:02进入窗口，此时 State 中的值更新为3，同时到了Trigger的触发时间，输出结果为3。



由于 PurgingTrigger 的作用，State 中的数据会被清除。



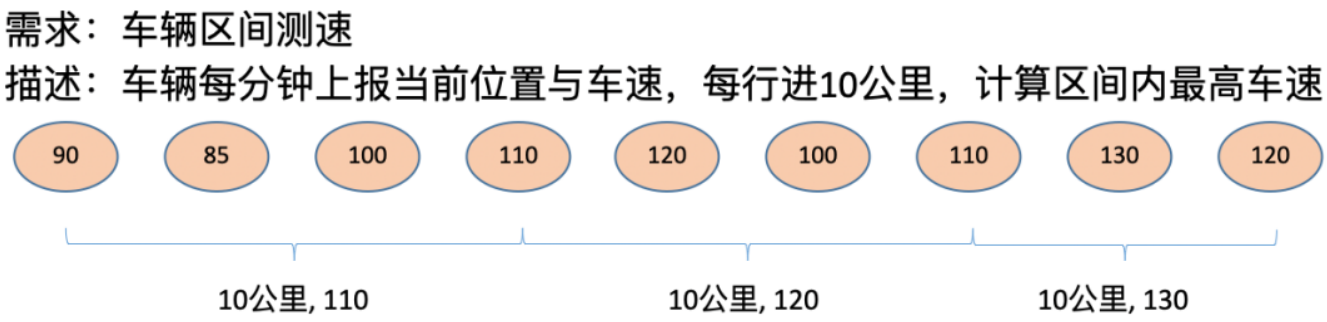
当后两条数据进入窗口之后，State 重新从0开始累计并更新为5，输出结果为5。

由于结果输出是 append 模式，会输出3和5两条数据，然后再做 Sum 也能得到正确的结果。

上面就是 PurgingTrigger 的一个简单的示例，它还支持很多有趣的玩法。

■ DeltaTrigger 的应用

有这样一个车辆区间测试的需求，车辆每分钟上报当前位置与车速，每行进10公里，计算区间内最高车速。



首先需要考虑的是如何来划分窗口，它不是一个时间的窗口，也不是一个基于数量的窗口。用传统的窗口实现比较困难，这种情况下我们考虑使用 DeltaTrigger 来实现。

下面是简单的代码实现：

```
DataStream<Tuple4<Integer, Integer, Double, Long>> topSpeeds = carData
        .assignTimestampsAndWatermarks(new CarTimestamp())
        .keyBy( ...fields: 0)
        .window(GlobalWindows.create())
        .trigger(DeltaTrigger.of( threshold: 10000,
                new DeltaFunction<Tuple4<Integer, Integer, Double, Long>>() {
                    private static final long serialVersionUID = 1L;

                    @Override
                    public double getDelta(
                            Tuple4<Integer, Integer, Double, Long> oldDataPoint,
                            Tuple4<Integer, Integer, Double, Long> newDataPoint) {
                        return newDataPoint.f2 - oldDataPoint.f2;
                    }
                }, carData.getType().createSerializer(env.getConfig())))
        .max( positionToMax: 1);
```

如何提取时间戳和生成水印，以及选择聚合维度就不赘述了。这个场景不是传统意义上的时间窗口或数量窗口，可以创建一个 GlobalWindow，所有数据都在一个窗口中，我们通过定义一个 DeltaTrigger，并设定一个阈值，这里是10000（米）。每个元素和上次触发计算的元素比较是否达到设定的阈值，这里比较的是每个元素上报的位置，如果达到了10000（米），那么当前元素和上一个触发计算的元素之间的所有元素落在同一个窗口里计算，然后可以通过 Max 聚合计算出最大的车速。

■ 思考点

上面这个例子中我们通过 GlobalWindow 和 DeltaTrigger 来实现了自定义的 Window Assigner 的功能。对于一些复杂的窗口，我们还可以自定义 WindowAssigner，但实现起来不一定简单，倒不如利用 GlobalWindow 和自定义 Trigger 来达到同样的效果。

下面这个是 Flink 内置的 CountWindow 的实现，也是基于 GlobalWindow 和 Trigger 来实现的。

GlobalWindows + Trigger = Custom Window Assigner

```java
/**
 * Windows this {@code KeyedStream} into tumbling count windows.
 *
 * @param size The size of the windows in number of elements.
 */
public WindowedStream<T, KEY, GlobalWindow> countWindow(long size) {
    return window(GlobalWindows.create()).trigger(PurgingTrigger.of(CountTrigger.of(size)));
}

/**
 * Windows this {@code KeyedStream} into sliding count windows.
 *
 * @param size The size of the windows in number of elements.
 * @param slide The slide interval in number of elements.
 */
public WindowedStream<T, KEY, GlobalWindow> countWindow(long size, long slide) {
    return window(GlobalWindows.create())
            .evictor(CountEvictor.of(size))
            .trigger(CountTrigger.of(slide));
}
```

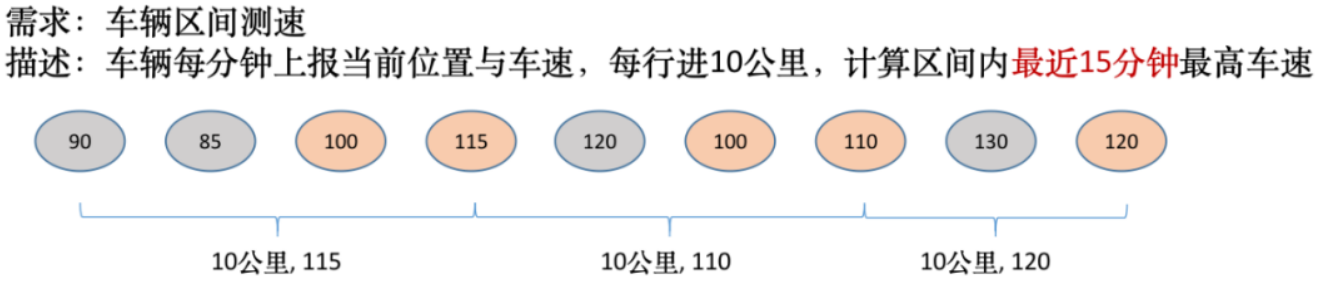
■ Window Evictor

Flink 内置了一些 Evictor 的实现。

内置Evictor	说明
CountEvictor	窗口计算时，只保留最近N条element
TimeEvictor	窗口计算时，只保留最近N段时间范围的element
DeltaEvictor	窗口计算时，最新的一条element与其他element做delta计算，只保留delta在threshold内的element

■ TimeEvictor 的应用

基于上面的区间测速的场景，每行进10公里，计算区间内最近15分钟最高车速。



实现上只是在前面基础上增加了 Evictor 的使用，过滤掉窗口最后15分钟之前的数据。

```
DataStream<Tuple4<Integer, Integer, Double, Long>> topSpeeds = carData
        .assignTimestampsAndWatermarks(new CarTimestamp())
        .keyBy( ...fields: 0)
        .window(GlobalWindows.create())
        .evictor(TimeEvictor.of(Time.minutes(15)))
        .trigger(DeltaTrigger.of( threshold: 10000,
                new DeltaFunction<Tuple4<Integer, Integer, Double, Long>>() {
                    private static final long serialVersionUID = 1L;

                    @Override
                    public double getDelta(
                            Tuple4<Integer, Integer, Double, Long> oldDataPoint,
                            Tuple4<Integer, Integer, Double, Long> newDataPoint) {
                        return newDataPoint.f2 - oldDataPoint.f2;
                    }
                }, carData.getType().createSerializer(env.getConfig())))
        .max( positionToMax: 1);
```

■ Window Function

Flink 内置的 WindowFunction 有两种类型，第一种是 AggregateFunction，它是高级别的抽象，主要用来做增量聚合，每来一条元素都做一次聚合，这样状态里只需要存最新的聚合值。

- 优点：增量聚合，实现简单。
- 缺点：输出只有一个聚合值，使用场景比较局限。

```
interface AggregateFunction<IN, ACC, OUT>  extends Function {
    ACC  createAccumulator();
    ACC  add(IN value, ACC accumulator);
    OUT  getResult(ACC accumulator);
    ACC  merge(ACC a, ACC b);
}
```

incremental aggregation
high-level abstraction

第二种是 ProcessWindowFunction，它是低级别的抽象用来做全量聚合，每来一条元素都存在状态里面，只有当窗口触发计算时才会调用这个函数。

```
abstract class ProcessWindowFunction<IN, OUT, KEY, W extends Window>
extends AbstractRichFunction {
    public abstract void process(KEY key, Context context, Iterable<IN>
elements, Collector<OUT> out)
}
```

full aggregation
low-level abstraction

- 优点：可以获取到窗口内所有数据的迭代器，实现起来比较灵活；可以获取到聚合的 Key 以及可以从上下文 Context 中获取窗口的相关信息。
- 缺点：需要存储窗口内的全量数据，State 的压力较大。

同时我们可以把这两种方式结合起来使用，通过 AggregateFunction 做增量聚合，减少中间状态的压力。通过 ProcessWindowFunction 来输出我们想要的信息，比如聚合的 Key 以及窗口的信息。

工作流程和实现机制

上一节我们介绍了窗口的一些抽象的概念，包括它的编程接口，通过一些简单的示例介绍了每个抽象概念的的用

法。

这一节我们深入的研究以下窗口底层是怎么实现的。

WindowOperator 工作流程

首先看下 WindowOperator 的工作流程，代码做了一些简化，只保留了核心步骤。

```
public void processElement(StreamRecord<IN> element) throws Exception {
    final Collection<W> elementWindows = windowAssigner.assignWindows(
        element.getValue(), element.getTimestamp(), windowAssignerContext);

    final K key = this.<K>getKeyedStateBackend().getCurrentKey();

    for (W window: elementWindows) {
        // drop if the window is already late
        if (isWindowLate(window)) {
            continue;
        }

        windowState.setCurrentNamespace(window);
        windowState.add(element.getValue());

        triggerContext.key = key;
        triggerContext.window = window;

        TriggerResult triggerResult = triggerContext.onElement(element);

        if (triggerResult.isFire()) {
            ACC contents = windowState.get();
            if (contents == null) {
                continue;
            }
            emitWindowContents(window, contents);
        }

        if (triggerResult.isPurge()) {
            windowState.clear();
        }
        registerCleanupTimer(window);
    }
```

主要包括以下8个步骤：

1. 获取 element 归属的 windows
2. 获取 element 对应的 key
3. 如果 late data，跳过
4. 将 element 存入 window state
5. 判断 element 是否触发 trigger
6. 获取 window state，注入 window function
7. 清除 window state
8. 注册 timer，到窗口结束时间清理 window

Window State

前面提到的增量聚合计算和全量聚合计算，这两个场景所应用的 State 是不一样的。

如果是全量聚合，元素会添加到 ListState 当中，当触发窗口计算时，再把 ListState 中所有元素传递给窗口函数。

```java
public void add(V value) {
    Preconditions.checkNotNull(value, errorMessage: "You ca

    final N namespace = currentNamespace;

    final StateTable<K, N, List<V>> map = stateTable;
    List<V> list = map.get(namespace);

    if (list == null) {
        list = new ArrayList<>();
        map.put(namespace, list);
    }
    list.add(value);
}
```

ListState
process(...)/evitor(...)

如果是增量计算，使用的是 AggregatingState，每条元素进来会触发 AggregateTransformation 的计算。

```java
public void add(IN value) throws IOException {
    final N namespace = currentNamespace;

    if (value == null) {
        clear();
        return;
    }

    try {
        stateTable.transform(namespace, value, aggregateTransformation
    } catch (Exception e) {
        throw new IOException("Exception while applying AggregateFunct
    }
}
```

AggregatingState
reduce(...)/aggregate(...)

看下 AggregateTransformation 的实现，它会调用我们定义的 AgregateFunction 中的 createAccumulator 方法和 add 方法并将 add 的结果返回，所以 State 中存储的就是 accumulator 的值，所以比较轻量级。

```java
static final class AggregateTransformation<IN, ACC, OUT> implements StateTransformationFunction<ACC, IN> {

    private final AggregateFunction<IN, ACC, OUT> aggFunction;

    AggregateTransformation(AggregateFunction<IN, ACC, OUT> aggFunction) {
        this.aggFunction = Preconditions.checkNotNull(aggFunction);
    }

    @Override
    public ACC apply(ACC accumulator, IN value) {
        if (accumulator == null) {
            accumulator = aggFunction.createAccumulator();
        }
        return aggFunction.add(value, accumulator);
    }
}
```

Window Function

在触发窗口计算时会将窗口中的状态传递给 emitWindowContents 方法。这里会调用我们定义的窗口函数中的 process 方法，将当前的 Key、Window、上下文 Context、窗口的内容作为参数传给它。在此之前和之后会分别调用 evictBefore 和evictAfter 方法把一些元素过滤掉。最终会调用 windowState 的 clear 方法，再把过滤之后的记录存到 windowState 中去。从而达到 evictor 过滤元素的效果。

```
vate void emitWindowContents(W window, Iterable<StreamRecord<IN>> contents,
                             ListState<StreamRecord<IN>> windowState) throws Exception
 // Work around type system restrictions...
 FluentIterable<TimestampedValue<IN>> recordsWithTimestamp = FluentIterable
     .from(contents)
     .transform(new Function<StreamRecord<IN>, TimestampedValue<IN>>() {...});

 evictorContext.evictBefore(recordsWithTimestamp, Iterables.size(recordsWithTimestamp)

 FluentIterable<IN> projectedContents = recordsWithTimestamp
     .transform(new Function<TimestampedValue<IN>, IN>() {...});

 processContext.window = triggerContext.window;
 userFunction.process(triggerContext.key, triggerContext.window, processContext,
     projectedContents, timestampedCollector);

 evictorContext.evictAfter(recordsWithTimestamp, Iterables.size(recordsWithTimestamp))

 //work around to fix FLINK-4369, remove the evicted elements from the windowState.
 //this is inefficient, but there is no other way to remove elements from ListState, w
 windowState.clear();
 for (TimestampedValue<IN> record : recordsWithTimestamp) {
     windowState.add(record.getStreamRecord());
 }
```

Window Trigger

最后看下 Trigger 的实现原理。当我们有大量的 Key，同时每个 Key 又属于多个窗口时，我们如何有效的触发窗口的计算呢?

Flink 利用定时器来保证窗口的触发，通过优先级队列来存储定时器。队列头的定时器表示离当前时间最近的一个，如果当前定时器比队列头的定时器时间还要早，则取消掉队列头的定时器，把当前的时间注册进去。

```
public class ProcessingTimeTrigger extends Trigger<Object, TimeWindow> {
    private static final long serialVersionUID = 1L;

    private ProcessingTimeTrigger() {}

    @Override
    public TriggerResult onElement(Object element, long timestamp, TimeWindow window, TriggerContext ctx) {
        ctx.registerProcessingTimeTimer(window.maxTimestamp());
        return TriggerResult.CONTINUE;
    }
```

```
public void registerProcessingTimeTimer(N namespace, long time) {
    InternalTimer<K, N> oldHead = processingTimeTimersQueue.peek();
    if (processingTimeTimersQueue.add(new TimerHeapInternalTimer<>(time, (K) keyContext.getCurrentKey(), namespace))) {
        long nextTriggerTime = oldHead != null ? oldHead.getTimestamp() : Long.MAX_VALUE;
        // check if we need to re-schedule our timer to earlier
        if (time < nextTriggerTime) {
            if (nextTimer != null) {
                nextTimer.cancel( mayInterruptIfRunning: false);
            }
            nextTimer = processingTimeService.registerTimer(time, this::onProcessingTime);
        }
    }
}
```

当这次定时器触发之后，再从优先级队列中取下一个 Timer，去调用 trigger 处理的函数，再把下一个 Timer 的时间注册为定时器。这样就可以循环迭代下去。

```
private void onProcessingTime(long time) throws Exception {
    // null out the timer in case the Triggerable calls registerProcessingTimeTimer()
    // inside the callback.
    nextTimer = null;

    InternalTimer<K, N> timer;

    while ((timer = processingTimeTimersQueue.peek()) != null && timer.getTimestamp() <= time) {
        processingTimeTimersQueue.poll();
        keyContext.setCurrentKey(timer.getKey());
        triggerTarget.onProcessingTime(timer);
    }

    if (timer != null && nextTimer == null) {
        nextTimer = processingTimeService.registerTimer(timer.getTimestamp(), this::onProcessingTime);
    }
}
```

总结

本文主要分享了 Flink 窗口的应用与实现。首先介绍了学习一项新技术的整体思路与学习路径，从应用入手慢慢深入它的实现。然后介绍了实时数仓的典型架构发展历程，之后从窗口的应用场景、抽象概念、编程结构详细说明了窗口的各个组成部分。并通过一些示例详细展示了各个概念之间配合使用可以满足什么样的使用场景。最后深入窗口的实现，从源码层面说明了窗口各模块的工作流程。