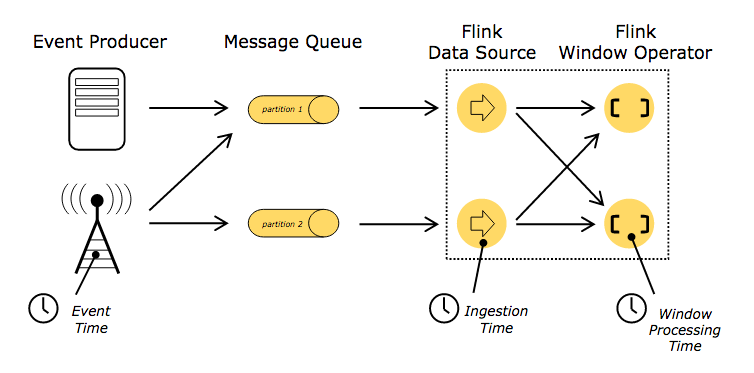
### flink时间概念

Flink的三种时间概念：

event time（事件时间：事件发生时的时间，可能是由生产事件的传感器或生产服务来附加），

ingestion time（摄取时间：事件进入流处理系统的时间），

processing time（处理时间：事件被计算处理的时间）



### window（窗口）

Flink是基于流处理的，而在流处理中，数据是连续不断的，于是对于一些聚合等操作，我们不能等到所有的数据结束了才进行处理（因为没有结束），但是我们可以定义这样一种操作，例如：在过去的1分钟内有多少用户点击了我们的网页，每100个用户计算平均停留时间…在这种情况下，我们必须定义一个范围，无论是基于时间还是基于事件个数，并对这个范围内的数据进行计算，这个范围就叫做窗口。

通常来说，窗口就是用来对一个无限的流设置一个有限的集合，在有界数据上进行操作的一种机制。

### 窗口分类

Flink定义的窗口机制，根据驱动的方式，我们可以将窗口分为三类：

时间驱动（Time Window）

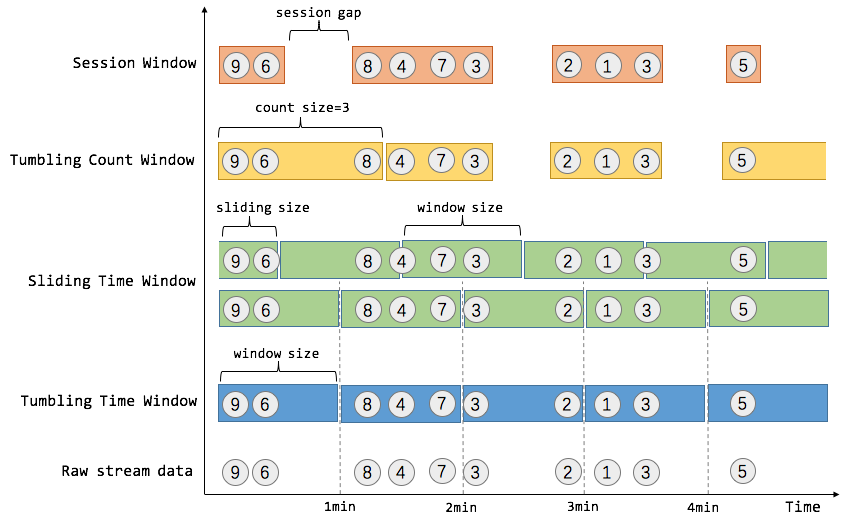
数据驱动（Count Window）。

会话驱动（Session Window）。

再根据窗口的特性，在数据驱动和时间驱动中，我们还可以分成两类：

翻滚窗口（Tumbling Window），

滚动窗口（Sliding Window）



1. Tumbling Time Window（翻滚时间窗口）

按照相同的时间段长度将数据流进行切分，这种切分被成为翻滚时间窗口（Tumbling Time Window），如上图统计每分钟中事件发生的总数，需要将行为事件按每一分钟进行切分。翻滚窗口能将数据流切分成不重叠的窗口，每一个事件只能属于一个窗口。

参数为1个，即窗口大小size

1. Sliding Time Window（滑动时间窗口）

指定一个时间段长度作为窗口大小，并以一定的步长在数据流中进行滑动，这种窗口我们称为滑动时间窗口（Sliding Time Window），这种窗口的特点是允许重叠的窗口存在。比如，我们可以每30秒计算一次最近一分钟事件发生总数。在滑窗中，一个元素可以对应多个窗口。

参数为2个，窗口大小size和滑动值slide

1. Tumbling Count Window（翻滚计数窗口）

类似于翻滚时间窗口，只是窗口大小从时间变成了事件个数，当窗口大小达到指定事件个数时，触发窗口计算，这种窗口我们称之为翻滚计数窗口（Tumbling Count Window），上图所示窗口大小为3个。

参数为1个，窗口大小size

1. Sliding Count Window

类似于滑动时间窗口，事件个数作为窗口大小，并以一定的步长在数据流中进行滑动和窗口触发条件，这种窗口我们称之为滑动计数窗口（Tumbling Count Window）。我们可以每10个元素计算一次最近100个元素的总和。

参数为2个，窗口大小size和滑动值slide

1. Session window

这种窗口经常用于用户交互事件流中，将一段用户持续活动周期的一些列事件聚集到一个窗口中，由非活动的间隙分隔开，这样的窗口称为会话窗口。

参数为一个，窗口的非活动间隙gap

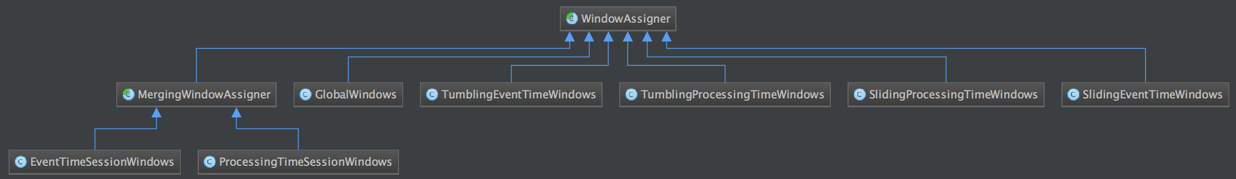
1. Advanced window (自定义窗口)

除此之外，flink还支持自定义窗口创建，自定义窗口需要自己指定3个function，分别是window assigner,Trigger,Evictor

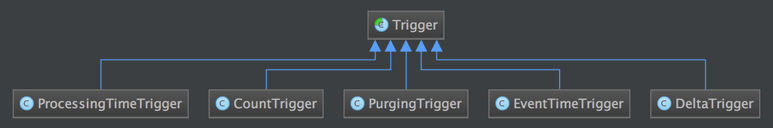
可能出现问题：

### Window工作流程

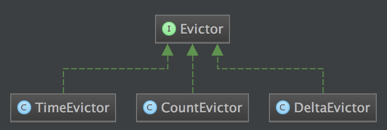
Window assigner：决定某个元素被分配到哪个/哪些窗口中去



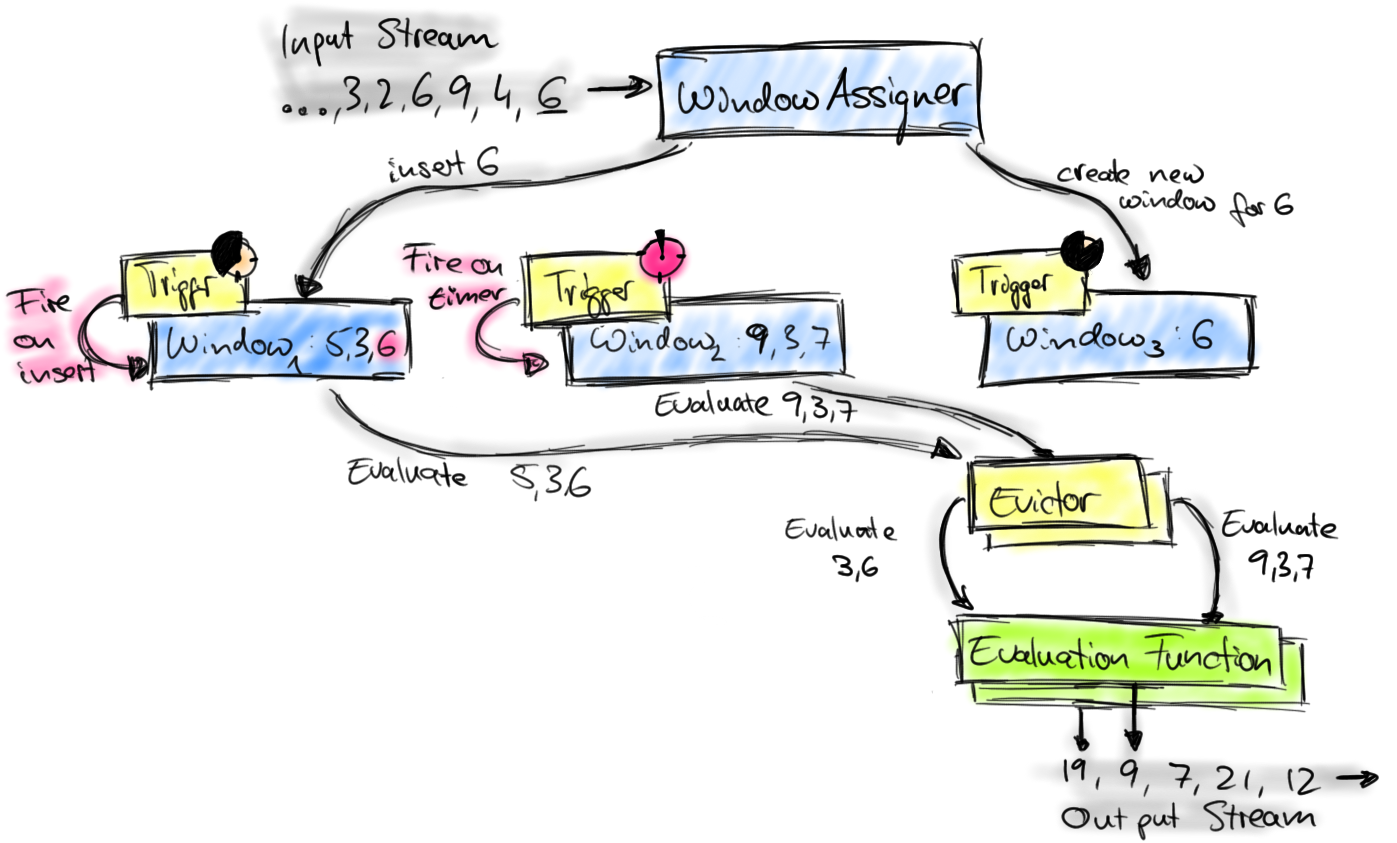
Trigger：触发器，决定一个窗口何时被计算或清除，每个窗口都有一个自己的Trigger，返回值有几种，continue（不做任何操作），fire（处理窗口数据），purge（清除窗口和窗口数据），或者fire+purge()



Evictor：驱逐器，在Trigger触发之后，在窗口被处理之前，Evictor（如果有Evictor的话）会用来剔除窗口中不需要的元素。



各组件之间相互工作图



Window assigner -> trigger -> evictor -> function

步骤如下：

首先上图中的组件都位于一个算子（window operator）中，数据流源源不断地进入算子，每一个到达的元素都会被交给 WindowAssigner。WindowAssigner 会决定元素被放到哪个或哪些窗口（window，如果是滑动窗口的话就是多个窗口，其他的就是一个窗口），也可能会创建新窗口。注意，Window本身只是一个ID标识符，其内部可能存储了一些元数据，如TimeWindow中有开始和结束时间，但是并不会存储窗口中的元素。窗口中的元素实际存储在 Key/Value State 中，key为Window，value为元素集合（或聚合值）。

每一个窗口都拥有一个属于自己的 Trigger，Trigger上会有定时器，用来决定一个窗口何时能够被计算或清除。每当有元素加入到该窗口，或者之前注册的定时器超时了，那么Trigger都会被调用。Trigger的返回结果可以是 continue（不做任何操作），fire（处理窗口数据），purge（移除窗口和窗口中的数据），或者 fire + purge。一个Trigger的调用结果只是fire的话，那么会计算窗口并保留窗口原样，也就是说窗口中的数据仍然保留不变，等待下次Trigger fire的时候再次执行计算。一个窗口可以被重复计算多次直到它被 purge 了。在purge之前，窗口会一直占用着内存。

当Trigger fire了，窗口中的元素集合就会交给Evictor（如果指定了的话）。Evictor 主要用来遍历窗口中的元素列表，并决定最先进入窗口的多少个元素需要被移除。剩余的元素会交给用户指定的函数进行窗口的计算。如果没有 Evictor 的话，窗口中的所有元素会一起交给函数进行计算。

计算函数收到了窗口的元素（可能经过了 Evictor 的过滤），并计算出窗口的结果值，并发送给下游。窗口的结果值可以是一个也可以是多个。DataStream API 上可以接收不同类型的计算函数，包括预定义的sum(),min(),max()，还有 ReduceFunction，FoldFunction，还有WindowFunction。WindowFunction 是最通用的计算函数，其他的预定义的函数基本都是基于该函数实现的。

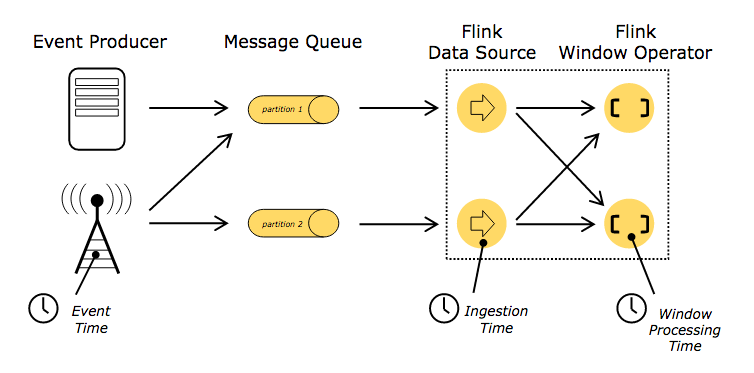
Flink 对于一些聚合类的窗口计算（如sum,min）做了优化，因为聚合类的计算不需要将窗口中的所有数据都保存下来，只需要保存一个result值就可以了。每个进入窗口的元素都会执行一次聚合函数并修改result值。这样可以大大降低内存的消耗并提升性能。但是如果用户定义了 Evictor，则不会启用对聚合窗口的优化，因为 Evictor 需要遍历窗口中的所有元素，必须要将窗口中所有元素都存下来。

### 消息乱序与watermark

Flink将时间分为3种，event time, processing time和ingestion time，其中ingestion time的处理类似于event time。

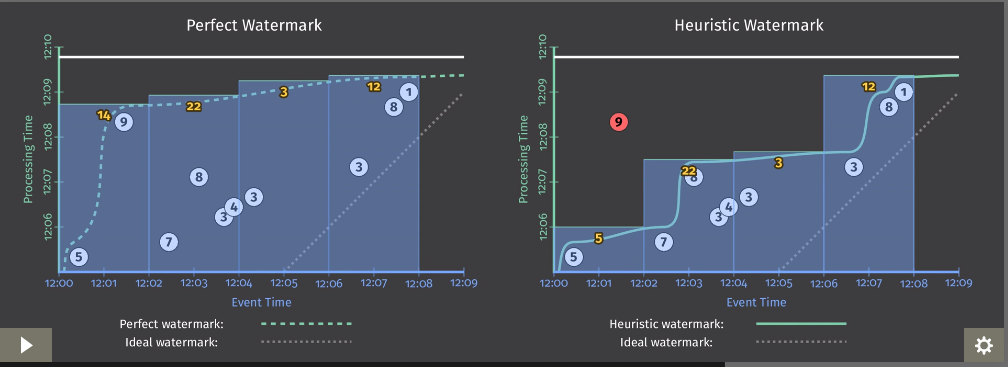
当我们只关心事件的processing time，而processing Time是在事件被算子处理时打上的时间戳，不会产生乱序问题。

但是很多情况下，我们真正关心的是事件实际发生的时间，即event time，若事件自带时间戳，根据消息的时间戳进行处理时，要确保时间戳在同一个时间窗口的所有消息一定会被正确处理。网络延迟等因素，消息可能乱序流入task，所以task需要缓存当前时间窗口消息处理的状态，直到确认属于该时间窗口的所有消息都被处理，才可以释放，如果乱序的消息延迟很高会影响分布式系统的吞吐量和延迟。消息乱序就变成了一个不可避免的问题。



## Watermark

Flink借鉴了Google的MillWheel项目，通过WaterMark来支持基于Event Time的时间窗口。



WaterMark包含一个时间戳，Flink使用WaterMark标记所有小于该时间戳的消息都已流入，Flink的数据源在确认所有小于某个时间戳的消息都已输出到Flink流处理系统后，会生成一个包含该时间戳的WaterMark，插入到消息流中输出到Flink流处理系统中，Flink操作符按照时间窗口缓存所有流入的消息，当操作符处理到WaterMark时，它对所有小于该WaterMark时间戳的时间窗口数据进行处理并发送到下一个操作符节点，然后也将WaterMark发送到下一个操作符节点。

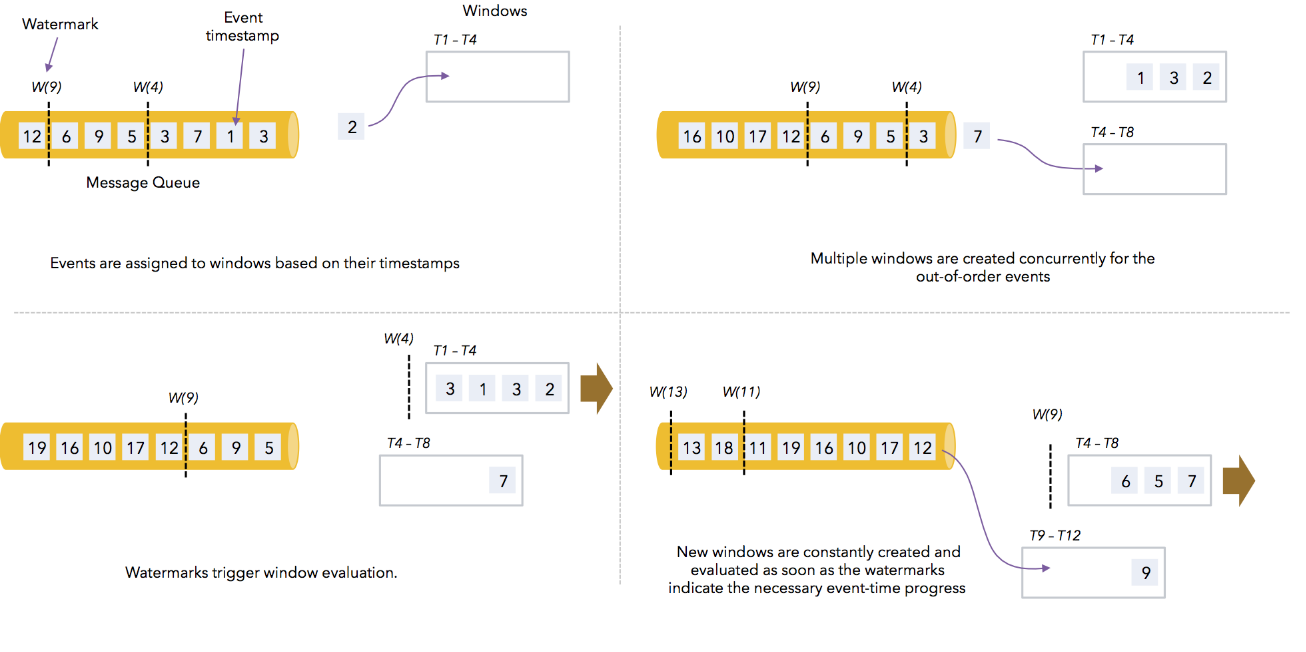
Flink生成watermark的方式主要有两种：

1. 周期性的watermark，周期性的产生watermark，如果有效就发送这个watermark到下游
2. 间断性的watermark，每来一个元素，进行判断是否要发送watermark

多数情况下，我们使用周期性的watermark来产生watermark，它可以设置一个最大乱序时间来处理乱序问题。

当我们将watermark运用到window机制中时，由于有watermark的存在，window的触发条件为：

1. 在[window\_start\_time,window\_end\_time)中有数据存在
2. 当前最大的watermark时间戳 >= window\_end\_time



**Late element**

当然，即使我们采用了watermark的方式，还是可能有一些在watermark(t)之后来的数据，即它的时间戳小于watermark(t)(因为我们不能一直推迟watermark的发送，这样会推迟窗口的计算)，我们将这些数据称为late element。默认情况下，这些late element数据会被删除。

但是flink也提供了allowedLateness方法来处理。简单来讲，对于watermark超过end-of-window之后，还允许有一段时间（也是以event time来衡量）来等待之前的数据到达，以便再次处理这些数据。

|  |
| --- |
| val input: DataStream[T] = …  input.keyby(<key selector>)  .window(<window assigner>)  .allowedLateness(<time>)  .<window function> |

默认情况下，如果不指定allowedLateness，其值是0。即对于watermark超过end-of-window之后，还有此window的数据到达时，这些数据被删除掉了。allowedLateness会再次触发窗口的计算，而之前触发的数据，会缓存起来，直到watermark超过end-of-window + allowedLateness的时间，窗口的数据及元数据信息才会被删除。同时，对于session window的情况，当late element在allowedLateness范围之内到达时，可能会引起窗口的merge。