DataStream API

# Flink程序运行

|  |  |
| --- | --- |
| 阶段 | 实现 |
| 1.执行环境设定 | StreamExecutionEnvironment env **=** StreamExecutionEnvironment**.**getExecutionEnvironment**();** |
| 2.加载数据源（Source） | DataStream<Tuple2<String, Integer>> dataStream = env  .socketTextStream("localhost", 9999); |
| 3.指定数据集的转换 | dataStream**.**flatMap**(new** **Splitter()).**keyBy**(**0**)**                 **.**timeWindow**(**Time**.**seconds**(**5**)).**sum**(**1**);** |
| 4.指定结果存放的位置（sink） | dataStream**.**print**(); // data sink** |
| 5.触发执行 | env.execute(); |

* execution environment：有三种用法，getExecutionEnvironment（）、createLocalEnvironment()、createRemoteEnvironment（host，port，jarFiles）。其中getExecutionEnvironment（）可以自动判断使用哪种方式创建上下文（如果是IDE中运行的，会自动创建为本地环境；如果是打成jar包，会自动创建为cluster模式）
* 获取初始数据集可以从CSV文件中读取，也可以使用完全自定义的数据输入格式。例如 DataStream<String> input = env.readTextFile("file:///path/to/file");
* 指定需要进行的转换：input.map(new MapFunction<String,Integer>(){});
* 指定存放结果的位置：writeAsText(String path)或print()
* 触发执行：execute();该方法返回一个JobExecutionResult，包含了执行时间以及其他累加器的结果。

# Transformation

Transformation将一个或多个DataStream转换成一个DataStream。

|  |  |
| --- | --- |
| Transformation | Description |
| Map（DataStream-》DataStream） | 取出一个元素，通过计算产生一个新的元素。  DataStream**<**Integer**>** dataStream **=** *//...*  dataStream**.**map**(new** MapFunction**<**Integer**,** Integer**>()** **{**      @Override     **public** Integer **map(**Integer value**)** **throws** Exception **{**         **return** 2 **\*** value**;**     **}});** |
| FlatMap（DataStream-》DataStream ） | 取出一个元素，产生零个、一个或多个元素。（如将句子分割成单词）  dataStream**.**flatMap**(new** FlatMapFunction**<**String**,** String**>()** **{**     @Override     **public** **void** **flatMap(**String value**,** Collector**<**String**>** out**)**         **throws** Exception **{**         **for(**String word: value**.**split**(**" "**)){**             out**.**collect**(**word**);**         **}**     **}});** |
| Filter（DataStream-》DataStream ） | 对数据流进行过滤，满足条件的返回true，不满足的返回false。  dataStream**.**filter**(new** FilterFunction**<**Integer**>()** **{**     @Override     **public** **boolean** **filter(**Integer value**)** **throws** Exception **{**         **return** value **!=** 0**;**     **}});** |
| KeyBy（DataStream-》KeyedStream） | 逻辑上将流分为不相交的分区，每个分区包含的元素的Key相同。内部是通过哈希分区实现的。  dataStream**.**keyBy**(**"someKey"**)** *// Key by field "someKey"*  dataStream**.**keyBy**(**0**)** *// Key by the first element of a Tuple* |
| Reduce（KeyedStream-》DataStream） | 将当前元素和上次reduce的值合并，并发送产生的新的value。  keyedStream**.**reduce**(new** ReduceFunction**<**Integer**>()** **{**     @Override     **public** Integer **reduce(**Integer value1**,** Integer value2**)**     **throws** Exception **{**         **return** value1 **+** value2**;**     **}});** |
| Aggregations（KeyedStream-》DataStream） | min返回最小值，minBy返回具有最小值的元素。max和maxBy与此类似。  keyedStream**.**sum**(**0**);**  keyedStream**.**sum**(**"key"**);**  keyedStream**.**min**(**0**);**  keyedStream**.**min**(**"key"**);**  keyedStream**.**max**(**0**);**  keyedStream**.**max**(**"key"**);**  keyedStream**.**minBy**(**0**);**  keyedStream**.**minBy**(**"key"**);**  keyedStream**.**maxBy**(**0**);**  keyedStream**.**maxBy**(**"key"**);** |
| Window（KeyedStream-》WindowedStream） | 可以在已经分区的KeyedStream上定义window。它可以根据某些特性对每个key中的数据进行分组。  dataStream**.**keyBy**(**0**).**window**(**TumblingEventTimeWindows**.**of**(**Time**.**seconds**(**5**)));** *// Last 5 seconds of data* |

# 3. Window[1]

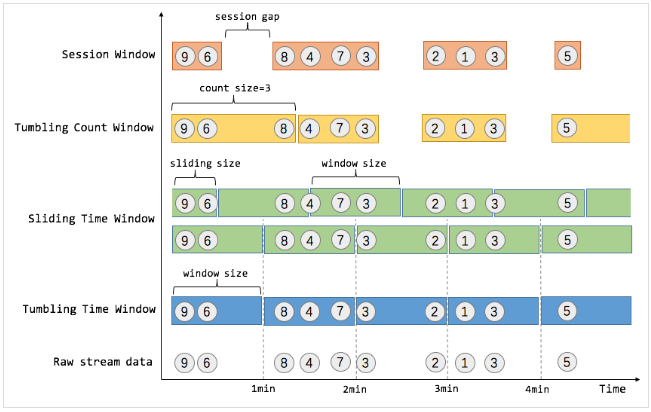
Flink 认为 Batch 是 Streaming 的一个特例，所以 Flink 底层引擎是一个流式引擎，在上面实现了流处理和批处理。而窗口（window）就是从 Streaming 到 Batch 的一个桥梁。

## 3.1 什么是Window？

在流处理应用中，数据是连续不断的，因此我们不可能等到所有数据都到了才开始处理。当然可以每来一个消息就处理一次，但是有时需要做一些聚合类的处理，例如：在过去的1分钟内有多少用户点击了我们的网页。在这种情况下，就必须定义一个窗口，用来收集最近一分钟内的数据，并对这个窗口内的数据进行计算。

窗口可以是时间驱动的（Time Window，例如：每30秒钟），也可以是数据驱动的（Count Window，例如：每一百个元素）。一种经典的窗口分类可以分成：翻滚窗口（Tumbling Window，无重叠），滚动窗口（Sliding Window，有重叠），和会话窗口（Session Window，活动间隙）。虽然说是有时间驱动和数据驱动的，但是在最新的源码中并没有看到数据驱动的Count Window。

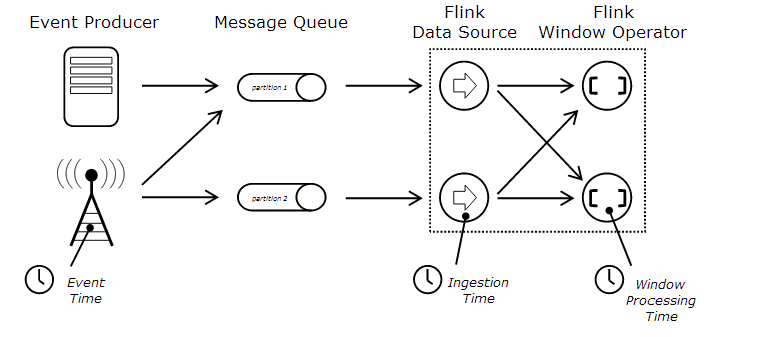
场景：淘宝网会记录每个用户每次购买的商品个数，我们要做的是统计不同窗口中用户购买商品的总数。下图给出了几种经典的窗口切分概述图：



图表 1 窗口切分

图1中，raw data stream 代表用户的购买行为流，圈中的数字代表该用户本次购买的商品个数，事件是按时间分布的，所以可以看出事件之间是有time gap的。Flink 提供了上图中所有的窗口类型，下面我们会逐一进行介绍。

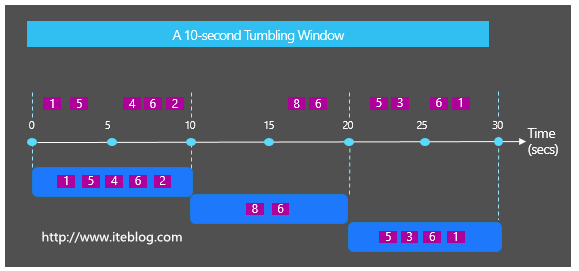
Flink 提出了三种时间的概念，分别是event time（事件时间：事件发生时的时间），ingestion time（摄取时间：事件进入流处理系统的时间），processing time（处理时间：消息被计算处理的时间）。底层实现上分为2种：Processing Time与Event Time，而Ingestion Time本质上也是一种Event Time。Flink中 Time Characteristic的默认使用的就是Processing Time。



图表 2 时间机制

Flink预先提供的窗口机制包括tumbling Windows【滚动窗口 】，sliding Windows 【滑动窗口 】，session Windows 【会话窗口 】和global Windows 【全局窗口 】。 Flink中窗口机制和时间类型是完全解耦的，也就是说当需要改变时间类型时不需要更改窗口逻辑相关的代码。

1. Tumbling Time Window：例如我们需要统计每一分钟中用户购买的商品的总数，需要将用户的行为事件按每一分钟进行切分，这种切分被成为翻滚时间窗口（Tumbling Time Window）。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。

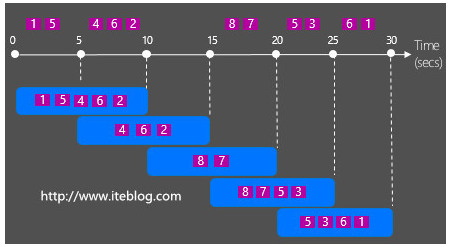


图表 3 Tumbling time window

代码如下：

|  |
| --- |
| DataStream**<**T**>** input **=** **...;**  *// tumbling event-time windows*  input**.**keyBy**(<**key selector**>)**     **.**window**(**TumblingEventTimeWindows**.**of**(**Time**.**seconds**(**5**)))**     **.<**windowed transformation**>(<**window function**>);** |

1. Sliding Time Window：但是对于某些应用，它们需要的窗口是不间断的，需要平滑地进行窗口聚合。比如，我们可以每30秒计算一次最近一分钟用户购买的商品总数。这种窗口我们称为滑动时间窗口（Sliding Time Window）。在滑窗中，一个元素可以对应多个窗口。sliding window是在数据流中的滑动，窗口之间可以有重叠，可以在传入的数据流中进行平滑的聚合。

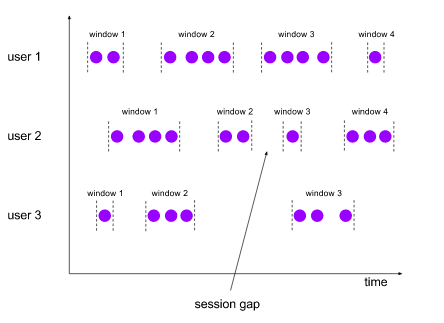


图表 4 Sliding Time window

代码如下：

|  |
| --- |
| *// sliding event-time windows*  input**.**keyBy**(<**key selector**>)**  **.**window**(**SlidingEventTimeWindows**.**of**(**Time**.**seconds**(**10**),** Time**.**seconds**(**5**)))** **//Time.seconds(10)指定窗口时间；Time.seconds(5)指定滑动时间；**  **.<**windowed transformation**>(<**window function**>);** |

1. Session Window：当我们需要分析用户的一段交互的行为事件时，通常的想法是将用户的事件流按“session”来分组。session 是指一段持续活跃的期间，由活跃间隙分隔开。通俗一点说，消息之间的间隔小于超时阈值（sessionGap）的，则被分配到同一个窗口，间隔大于阈值的，则被分配到不同的窗口。 Flink 对 session window 的支持主要借鉴自 Google 的 DataFlow 。 Flink 会基于元素的时间戳，自动地将元素放到不同的session window中。如果两个元素的时间戳间隔小于 session gap，则会在同一个session中。如果两个元素之间的间隔大于session gap，且没有元素能够填补上这个gap，那么它们会被放到不同的session中。如图1所示，就是需要计算每个用户在活跃期间总共购买的商品数量，如果用户30秒没有活动则视为会话断开（假设raw data stream是单个用户的购买行为流）。



图表 5 Session Window

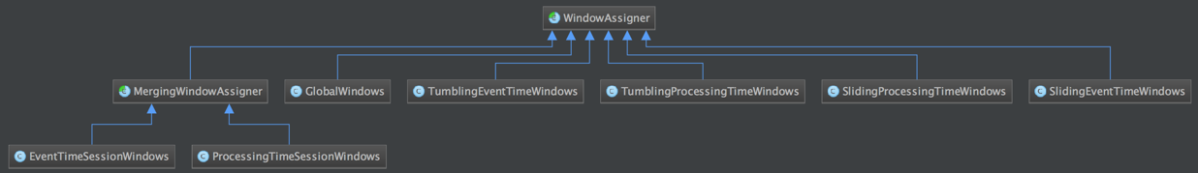
代码如下：

|  |
| --- |
| *// event-time session windows*  input**.**keyBy**(<**key selector**>)**     **.**window**(**EventTimeSessionWindows**.**withGap**(**Time**.**minutes**(**10**)))**     **.<**windowed transformation**>(<**window function**>);** |

## 3.2 Window API

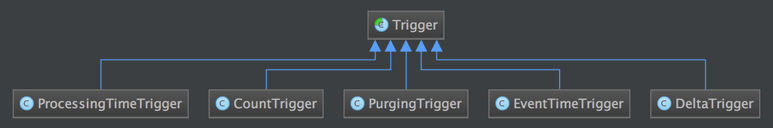
Flink 中定义一个窗口主要需要以下三个组件：Window assigner、Trigger、Evictor。

* Window Assigner：用来决定某个元素被分配到哪个/哪些窗口中去。



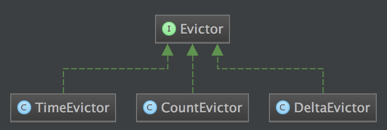
图表 6

* Trigger：触发器。决定了一个窗口何时能够被计算或清除，每个窗口都会拥有一个自己的Trigger。



图表 7

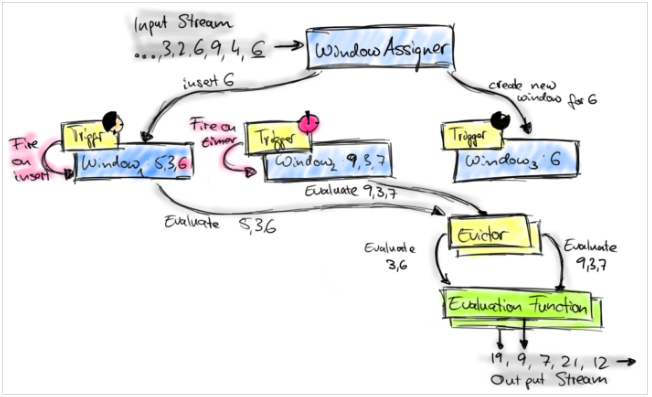
* Evictor：可以译为“驱逐者”。在Trigger触发之后，在窗口被处理之前，Evictor（如果有Evictor的话）会用来剔除窗口中不需要的元素，相当于一个filter。



图表 8

## 3.3 Window的实现

下图描述了 Flink 的窗口机制以及各组件之间是如何相互工作的。



图表 9 Window的实现原理

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），可能会创建新窗口。因为一个元素可以被放入多个窗口中，所以同时存在多个窗口是可能的。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）。为了保证窗口的容错性，该实现依赖了 Flink 的 State 机制（参见 [state 文档](https://ci.apache.org/projects/flink/flink-docs-master/apis/streaming/state.html)）。

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。

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。

## 3.4 源码分析

首先在KeyedStream中调用window方法，同时确定使用哪个Window assigner来创建window。

public <W extends Window> WindowedStream<T, KEY, W> window(WindowAssigner<? super T, W> assigner) {  
 return new WindowedStream<>(this, assigner);  
}

下面是window assigner中的SlidingEventTimeWindows中分配窗口的代码。

public Collection<TimeWindow> assignWindows(Object element, long timestamp, WindowAssignerContext context) {  
 if (timestamp > Long.*MIN\_VALUE*) {  
 List<TimeWindow> windows = new ArrayList<>((int) (size / slide));

// 对齐时间戳  
 long lastStart = timestamp - timestamp % slide;  
 for (long start = lastStart;  
 start > timestamp - size;  
 start -= slide) {

// 当前时间戳对应多个window  
 windows.add(new TimeWindow(start, start + size));  
 }  
 return windows;  
 } else {  
 throw new RuntimeException("Record has Long.MIN\_VALUE timestamp (= no timestamp marker). " +  
 "Is the time characteristic set to 'ProcessingTime', or did you forget to call " +  
 "'DataStream.assignTimestampsAndWatermarks(...)'?");  
 }  
}

# Trigger

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。

## 4.1 什么是Trigger？

Trigger是用来决定一个窗口合适被计算或清除的。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的返回结果如下：

*/\*\*  
 \* No action is taken on the window.  
 \*/  
CONTINUE*(false, false),  
  
*/\*\*  
 \* {****@code*** *FIRE\_AND\_PURGE} evaluates the window function and emits the window  
 \* result.   
 \*/  
FIRE\_AND\_PURGE*(true, true),  
  
*/\*\*  
 \* On {****@code*** *FIRE}, the window is evaluated and results are emitted.  
 \* The window is not purged, though, all elements are retained.  
 \*/  
FIRE*(true, false),  
  
*/\*\*  
 \* All elements in the window are cleared and the window is discarded,  
 \* without evaluating the window function or emitting any elements.  
 \*/  
PURGE*(false, true);

Trigger的返回结果可以是 continue（不做任何操作），fire（处理窗口数据，且窗口中的所有元素都被保留），purge（不处理window中的数据，直接移除窗口和窗口中的数据），或者 fire + purge（处理完窗口中的数据后清除窗口）。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知道它被 purge 了。在purge之前，窗口会一直占用着内存。

## 4.2 源码分析

使用到的EventTimeTrigger，每个元素进入窗口都会调用onElement方法。

public TriggerResult onElement(Object element, long timestamp, TimeWindow window, TriggerContext ctx) throws Exception {  
 if (window.maxTimestamp() <= ctx.getCurrentWatermark()) {  
 // if the watermark is already past the window fire immediately  
 return TriggerResult.*FIRE*;  
 } else {

//注册定时器，当系统时间到达window end timestamp时，会调用EventTimeTrigger中的onEventTIme方法。  
 ctx.registerEventTimeTimer(window.maxTimestamp());  
 return TriggerResult.*CONTINUE*;  
 }  
}

onEventTime即如果当前的时间达到了最大时间戳的话，就出发计算操作，否则继续等待。

public TriggerResult onEventTime(long time, TimeWindow window, TriggerContext ctx) {  
 return time == window.maxTimestamp() ?  
 TriggerResult.*FIRE* :  
 TriggerResult.*CONTINUE*;  
}

# 参考资料

[1] Flink 原理与实现：Window 机制. [推酷]. 2016.05.25. <http://wuchong.me/blog/2016/05/25/flink-internals-window-mechanism/>.

[2] http://blog.csdn.net/lmalds/article/details/51699037