> 纵有疾风起，人生不言弃

Flink之Window概述

> streaming流式计算是一种被设计用于处理无限数据集的数据处理引擎，而无限数据集是指一种不断增长的本质上无限的数据集，而window是一种**切割无限数据为有限块进行处理**的手段。
>
> Window是无限数据流处理的核心，Window将一个无限的stream拆分成有限大小的"buckets"桶，我们可以在这些桶上做计算操作。

【在Spark中那里讲了Window概念了，它是处理无限流的一种手段而已，可以理解为根据什么样的规则来聚合数据】

Window类型

Window可以分成两类：

- CountWindow：按照指定的数据条数生成一个Window，与时间无关。
- TimeWindow：按照时间生成Window。

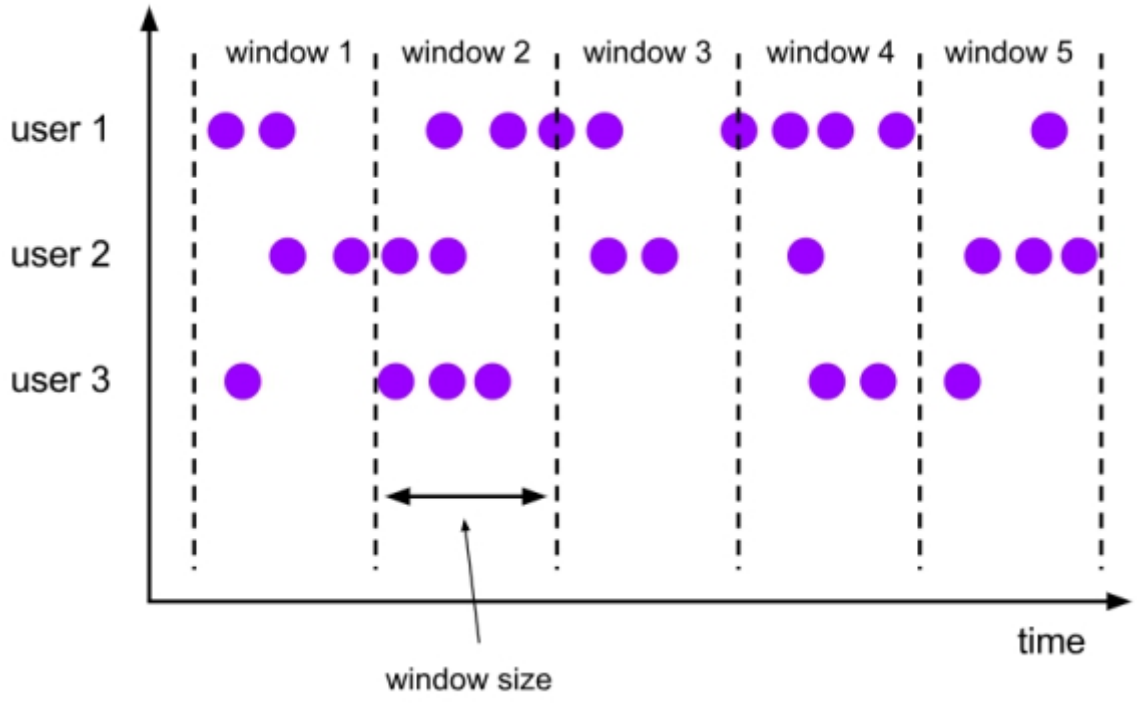
> 对于TimeWindow，可以根据窗口实现原理的不同分成三类：滚动窗口（Tumbling Window）、滑动窗口（Sliding Window）和会话窗口（Session Window）。【有问题都可以私聊我WX：focusbigdata，或者关注我的公众号：FocusBigData，注意大小写】

滚动窗口（Tumbling Windows）

> 将数据依据固定的窗口长度对数据进行切片。

特点：时间对齐，窗口长度固定，没有重叠

滚动窗口分配器将每个元素分配到一个指定窗口大小的窗口中，滚动窗口有一个固定的大小，并且不会出现重叠。例如：如果你指定了一个5分钟大小的滚动窗口，窗口的创建如下图所示：



适用场景：适合做BI统计等（做每个时间段的聚合计算）

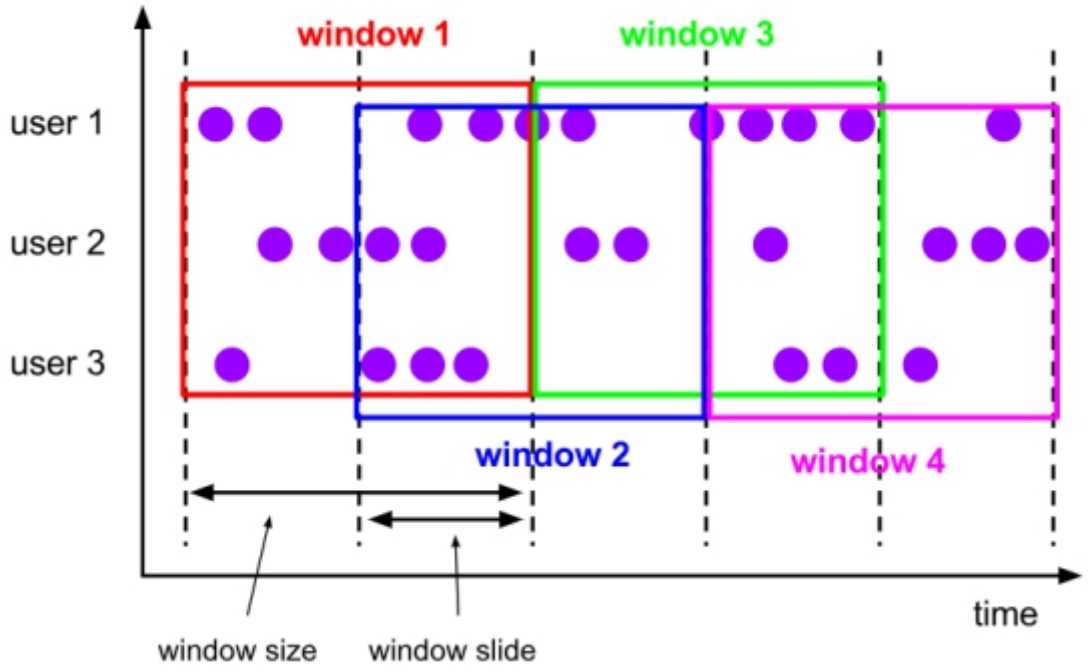
滑动窗口（Sliding Windows）

> 滑动窗口是固定窗口的更广义的一种形式，滑动窗口由固定的窗口长度和滑动间隔组成。

特点：时间对齐，窗口长度固定，可以有重叠

滑动窗口分配器将元素分配到固定长度的窗口中，与滚动窗口类似，窗口的大小由窗口大小参数来配置，另一个窗口滑动参数控制滑动窗口开始的频率。因此，滑动窗口如果滑动参数小于窗口大小的话，**窗口是可以重叠的**，在这种情况下元素会被分配到多个窗口中。

例如，你有10分钟的窗口和5分钟的滑动，那么每个窗口中5分钟的窗口里包含着上个10分钟产生的数据，如下图所示：



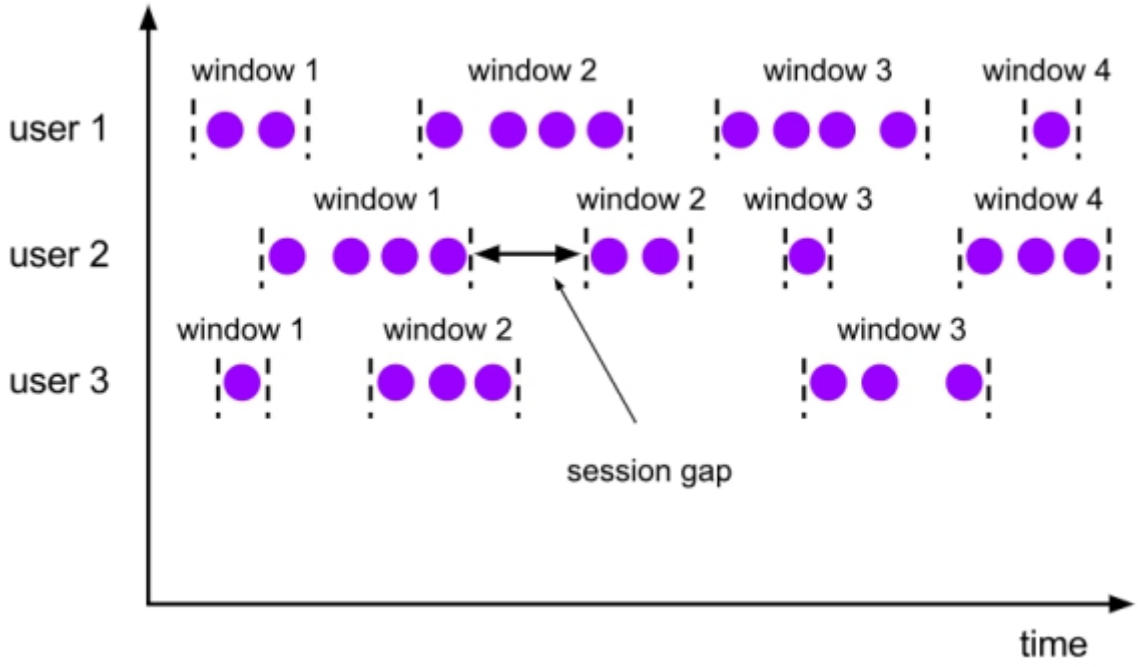
适用场景：对最近一个时间段内的统计（求某接口最近5min的失败率来决定是否要报警）

会话窗口（Session Windows）

> 由一系列事件组合一个指定时间长度的timeout间隙组成，类似于web应用的session，也就是一段时间没有接收到新数据就会生成新的窗口。

特点：时间无对齐

session窗口分配器通过session活动来**对元素进行分组**，session窗口跟滚动窗口和滑动窗口相比，**不会有重叠和固定的开始时间和结束时间的情况**，相反，当它在一个固定的时间周期内不再收到元素，即非活动间隔产生，那个这个窗口就会关闭。**一个session窗口通过一个session间隔来配置**，这个session间隔定义了非活跃周期的长度，当这个非活跃周期产生，那么当前的session将关闭并且后续的元素将被分配到新的session窗口中去。



Window API

TimeWindow时间窗口

> TimeWindow是将指定时间范围内的所有数据组成一个window，一次对一个window里面的所有数据进行计算。

滚动时间窗口API

> Flink默认的时间窗口根据Processing Time 进行窗口的划分，将Flink获取到的数据根据进入Flink的时间划分到不同的窗口中。

```
val minTempPerWindow = dataStream
  .map(r => (r.id, r.temperature))
  .keyBy(_._1)
  .timeWindow(Time.seconds(15))
  .reduce((r1, r2) => (r1._1, r1._2.min(r2._2)))
```

> 时间间隔可以通过Time.milliseconds(x)，Time.seconds(x)，Time.minutes(x)等其中的一个来指定。

滑动时间窗口API

> 滑动窗口和滚动窗口的函数名是完全一致的，只是在传参数时需要传入两个参数，**一个是window_size，一个是sliding_size。**【有问题都可以私聊我WX：focusbigdata，或者关注我的公众号：FocusBigData，注意大小写】

下面代码中的sliding_size设置为了5s，也就是说，每5s就计算输出结果一次，每一次计算的window范围是15s内的所有元素。

```
val minTempPerWindow: DataStream[(String, Double)] = dataStream
  .map(r => (r.id, r.temperature))
  .keyBy(_._1)
  .timeWindow(Time.seconds(15), Time.seconds(5))
  .reduce((r1, r2) => (r1._1, r1._2.min(r2._2)))

// .window(SlidingEventTimeWindows.of(Time.seconds(15),Time.seconds(5))
```

时间间隔可以通过Time.milliseconds(x)，Time.seconds(x)，Time.minutes(x)等其中的一个来指定。

CountWindow计数窗口

> CountWindow根据窗口中相同key元素的数量来触发执行，执行时只计算元素数量达到窗口大小的key对应的结果。

注意：CountWindow的window_size指的是相同Key的元素的个数，不是输入的所有元素的总数。

滚动计数窗口API

> 默认的CountWindow是一个滚动窗口，只需要指定窗口大小即可，当元素数量达到窗口大小时，就会触发窗口的执行。

```
val minTempPerWindow: DataStream[(String, Double)] = dataStream
  .map(r => (r.id, r.temperature))
  .keyBy(_._1)
  .countWindow(5)
  .reduce((r1, r2) => (r1._1, r1._2.max(r2._2)))
```

滑动计数窗口API

> 滑动窗口和滚动窗口的函数名是完全一致的，只是在传参数时需要传入两个参数，一个是window_size，一个是sliding_size。

下面代码中的sliding_size设置为了2，也就是说，每收到两个相同key的数据就计算一次，每一次计算的window范围是10个元素。

```
val keyedStream: KeyedStream[(String, Int), Tuple] = dataStream.map(r => (r.id,
r.temperature)).keyBy(0)
//每当某一个key的个数达到2的时候,触发计算，计算最近该key最近10个元素的内容
val windowedStream: WindowedStream[(String, Int), Tuple, GlobalWindow] =
keyedStream.countWindow(10,2)
val sumDstream: DataStream[(String, Int)] = windowedStream.sum(1)
```

WindowFunction【重点掌握】

window function 定义了要对窗口中收集的数据做的计算操作，主要可以分为两类：

- 增量聚合函数（incremental aggregation functions）

  > 每条数据到来就进行计算，保持一个简单的状态。典型的增量聚合函数有ReduceFunction, AggregateFunction。全窗口函数（full window functions）

- 先把窗口所有数据收集起来，等到计算的时候会遍历所有数据。

> ProcessWindowFunction就是一个全窗口函数。可以获得窗口的时间戳哦

其它可选API

- trigger() —— 触发器

> 定义 window 什么时候关闭，触发计算并输出结果

- evitor() —— 移除器

定义移除某些数据的逻辑

- allowedLateness() —— 允许处理迟到的数据
- sideOutputLateData() —— 将迟到的数据放入侧输出流
- getSideOutput() —— 获取侧输出流

Keyed Windows

```
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"
```

Non-Keyed Windows

```
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"
```