> 容易走的路，一般都是拥挤的

FlinkSql之窗口

> 时间语义，要配合窗口操作才能发挥作用。最主要的用途，当然就是开窗口、根据时间段做计算了。下面我们就来看看Table API和SQL中，怎么利用时间字段做窗口操作。

在Table API和SQL中，主要有两种窗口：**Group Windows和Over Windows**

分组窗口 Group Windows

> 分组窗口（Group Windows）**会根据时间或行计数间隔**，将行聚合到有限的组（Group）中，并对每个组的数据执行一次聚合函数。

Table API中的Group Windows都是使用.window（w:GroupWindow）子句定义的，并且必须由as子句指定一个别名。为了按窗口对表进行分组，窗口的别名必须在group by子句中，像常规的分组字段一样引用。

```
val table = input
  .window([w: GroupWindow] as 'w) // 定义窗口，别名 w
  .groupBy('w, 'a)  // 以属性a和窗口w作为分组的key
  .select('a, 'b.sum)  // 聚合字段b的值，求和
```

或者，还可以把窗口的相关信息，作为字段添加到结果表中：

```
val table = input
  .window([w: GroupWindow] as 'w)
  .groupBy('w, 'a)
  .select('a, 'w.start, 'w.end, 'w.rowtime, 'b.count)
```

> Table API提供了一组具有特定语义的预定义Window类，这些类会被转换为底层DataStream或DataSet的窗口操作。Table API支持的窗口定义，和我们熟悉的一样，主要也是三种：**滚动（Tumbling）、滑动（Sliding）和会话（Session）。**

滚动窗口

滚动窗口（Tumbling windows）要用Tumble类来定义，另外还有三个方法：

- over：定义窗口长度
- on：用来分组（按时间间隔）或者排序（按行数）的时间字段
- as：别名，必须出现在后面的groupBy中

代码如下

```
// Tumbling Event-time Window（事件时间字段rowtime）
.window(Tumble over 10.minutes on 'rowtime as 'w)

// Tumbling Processing-time Window（处理时间字段proctime）
.window(Tumble over 10.minutes on 'proctime as 'w)

// Tumbling Row-count Window (类似于计数窗口，按处理时间排序，10行一组)
.window(Tumble over 10.rows on 'proctime as 'w)
```

滑动窗口

滑动窗口（Sliding windows）要用Slide类来定义，另外还有四个方法：

- over：定义窗口长度
- every：定义滑动步长
- on：用来分组（按时间间隔）或者排序（按行数）的时间字段
- as：别名，必须出现在后面的groupBy中

代码如下

```
// Sliding Event-time Window
.window(Slide over 10.minutes every 5.minutes on 'rowtime as 'w)

// Sliding Processing-time window
.window(Slide over 10.minutes every 5.minutes on 'proctime as 'w)

// Sliding Row-count window
.window(Slide over 10.rows every 5.rows on 'proctime as 'w)
```

会话窗口

会话窗口（Session windows）要用Session类来定义，另外还有三个方法：

- withGap：会话时间间隔
- on：用来分组（按时间间隔）或者排序（按行数）的时间字段
- as：别名，必须出现在后面的groupBy中

代码如下：

```
// Session Event-time Window
.window(Session withGap 10.minutes on 'rowtime as 'w)

// Session Processing-time Window
.window(Session withGap 10.minutes on 'proctime as 'w)
```

相邻窗口 Over Windows

> Over window聚合是标准SQL中已有的（Over子句），可以在查询的SELECT子句中定义。Over window 聚合，会针对每个输入行，计算相邻行范围内的聚合。【注意这里是相邻行范围！！！】

使用.window（w:overwindows*）子句定义，并在select（）方法中通过别名来引用。

```
val table = input
  .window([w: OverWindow] as 'w)
  .select('a, 'b.sum over 'w, 'c.min over 'w)
```

Table API提供了Over类，来配置Over窗口的属性。可以在事件时间或处理时间，以及指定为时间间隔、或行计数的范围内，定义Over windows。

无界的over window是使用常量指定的。也就是说，时间间隔要指定UNBOUNDED_RANGE，或者行计数间隔要指定UNBOUNDED_ROW。而有界的over window是用间隔的大小指定的。

实际代码应用如下：

（1）无界的 over window

```
// 无界的事件时间over window （时间字段 "rowtime"）
.window(Over partitionBy 'a orderBy 'rowtime preceding UNBOUNDED_RANGE as 'w)

//无界的处理时间over window （时间字段"proctime"）
.window(Over partitionBy 'a orderBy 'proctime preceding UNBOUNDED_RANGE as 'w)

// 无界的事件时间Row-count over window （时间字段 "rowtime"）
.window(Over partitionBy 'a orderBy 'rowtime preceding UNBOUNDED_ROW as 'w)

//无界的处理时间Row-count over window （时间字段 "rowtime"）
.window(Over partitionBy 'a orderBy 'proctime preceding UNBOUNDED_ROW as 'w)
```

（2）有界的over window

```
// 有界的事件时间over window （时间字段 "rowtime"，之前1分钟）
.window(Over partitionBy 'a orderBy 'rowtime preceding 1.minutes as 'w)

// 有界的处理时间over window （时间字段 "rowtime"，之前1分钟）
.window(Over partitionBy 'a orderBy 'proctime preceding 1.minutes as 'w)

// 有界的事件时间Row-count over window （时间字段 "rowtime"，之前10行）
.window(Over partitionBy 'a orderBy 'rowtime preceding 10.rows as 'w)

// 有界的处理时间Row-count over window （时间字段 "rowtime"，之前10行）
.window(Over partitionBy 'a orderBy 'proctime preceding 10.rows as 'w)
```

SQL中窗口的定义

> 我们已经了解了在Table API里window的调用方式，同样，我们也可以在SQL中直接加入窗口的定义和使用。

Group Windows

> Group Windows在SQL查询的Group BY子句中定义。与使用常规GROUP BY子句的查询一样，使用GROUP BY子句的查询会计算每个组的单个结果行。

SQL支持以下Group窗口函数:

- TUMBLE(time_attr, interval)

> 定义一个滚动窗口，第一个参数是时间字段，第二个参数是窗口长度。

- HOP(time_attr, interval, interval)

> 定义一个滑动窗口，第一个参数是时间字段，第二个参数是窗口滑动步长，第三个是窗口长度。

- SESSION(time_attr, interval)

> 定义一个会话窗口，第一个参数是时间字段，第二个参数是窗口间隔（Gap）。

另外还有一些辅助函数，可以用来选择Group Window的开始和结束时间戳，以及时间属性。

这里只写TUMBLE*，*滑动和会话窗口是类似的（HOP，SESSION_）。

- TUMBLE_START(time_attr, interval)
- TUMBLE_END(time_attr, interval)
- TUMBLE_ROWTIME(time_attr, interval)
- TUMBLE_PROCTIME(time_attr, interval)

Over Windows

> 由于Over本来就是SQL内置支持的语法，**所以这在SQL中属于基本的聚合操作。所有聚合必须在同一窗口上定义**，也就是说，必须是相同的分区、排序和范围。目前仅支持在当前行范围之前的窗口（无边界和有边界）。

注意，ORDER BY必须在单一的时间属性上指定。

代码如下

```
SELECT COUNT(amount) OVER (
  PARTITION BY user
  ORDER BY proctime
  ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
FROM Orders

// 也可以做多个聚合
SELECT COUNT(amount) OVER w, SUM(amount) OVER w
FROM Orders
WINDOW w AS (
  PARTITION BY user
  ORDER BY proctime
  ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
```

案例

需求

> 统计10秒内出现的每个sensor的个数【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】

代码实现

```
def main(args: Array[String]): Unit = {
val env = StreamExecutionEnvironment.getExecutionEnvironment
```

```
env.setParallelism(1)
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

val streamFromFile: DataStream[String] = env.readTextFile("sensor.txt")
val dataStream: DataStream[SensorReading] = streamFromFile
.map( data => {
val dataArray = data.split(",")
SensorReading(dataArray(0).trim, dataArray(1).trim.toLong,
dataArray(2).trim.toDouble)
} )
.assignTimestampsAndWatermarks( new
BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.seconds(1)) {
override def extractTimestamp(element: SensorReading): Long = element.timestamp
* 1000L
} )

val settings: EnvironmentSettings = EnvironmentSettings
.newInstance()
.useOldPlanner()
.inStreamingMode()
.build()
val tableEnv: StreamTableEnvironment =
StreamTableEnvironment.create(env, settings)

val dataTable: Table = tableEnv
.fromDataStream(dataStream, 'id, 'temperature, 'timestamp.rowtime)

val resultTable: Table = dataTable
.window(Tumble over 10.seconds on 'timestamp as 'tw)
.groupBy('id, 'tw)
.select('id, 'id.count)

val sqlDataTable: Table = dataTable
.select('id, 'temperature, 'timestamp as 'ts)
val resultSqlTable: Table = tableEnv
.sqlQuery("select id, count(id) from "
+ sqlDataTable
+ " group by id,tumble(ts,interval '10' second)")

// 把 Table转化成数据流
val resultDstream: DataStream[(Boolean, (String, Long))] = resultSqlTable
.toRetractStream[(String, Long)]

resultDstream.filter(_._1).print()
env.execute()
}
```