# **mapGroupsWithState（需要先分组groupby）**

每次trigger后，将给定的function应用于有数据的每个分组，同时维护每组的状态

* func transformed to return a single-element Iterator

// S: 状态类型 U: 返回类型

// func: 应用于每组上的函数。K: 当前分组的Key Iterator[V]:当前批次下当前分组内的数据(V: 组内每条数据的类型) GroupState[S]: 当前这个分组的状态(状态里保存了老的聚合结果以及当前的状态是否超时等等)

// timeoutConf: 超时配置。有三种: GroupStateTimeout.ProcessingTimeTimeout()、GroupStateTimeout.EventTimeTimeout()、GroupStateTimeout.NoTimeout() 。

// 不超时

def mapGroupsWithState[S: Encoder, U: Encoder](func: (K, Iterator[V], GroupState[S]) => U): Dataset[U]

// 柯里化函数，可自定义超时模式, 一般用这个

def mapGroupsWithState[S: Encoder, U: Encoder](timeoutConf: GroupStateTimeout)(func: (K, Iterator[V], GroupState[S]) => U): Dataset[U]

// 不超时

def mapGroupsWithState[S, U](func: MapGroupsWithStateFunction[K, V, S, U],stateEncoder: Encoder[S],outputEncoder: Encoder[U]): Dataset[U]

// 可自定义超时模式

def mapGroupsWithState[S, U](func: MapGroupsWithStateFunction[K, V, S, U],stateEncoder: Encoder[S],outputEncoder: Encoder[U],timeoutConf: GroupStateTimeout): Dataset[U]

---

mapGroupsWithState支持update mode

超时配置（3种）

notimeout

eventtimeout，必须设置watermark

processingtimeout；无须；所有数据参与状态计算，分组的状态要手动清理，设置GroupState.setTimeoutDuration，超时后调用置GroupState.remove清除

flatMapGroupsWithState，支持append和update模式

---

不管是基于watermark的窗口计算还是自维护的状态流，它们都是有状态的，watermark只是规定了数据进入“状态”（有资格参与状态计算）的条件，并没有（也不适合）声明状态的“退出”机制。对于watermark的窗口计算来说，它们的“退出”机制是：如果最近某个还处于active状态的窗口它的EndTime比当前批次中最新的一个事件时间减去watermark规定的阈值还要“早”，说明这个窗口所有的数据都就绪了，不会再被更新了，就可以把正式“decommission”了。由于这个逻辑对于watermark的窗口计算来说是通行的， 所以被Spark封装在窗口计算中，对开发人员是透明的，但是对于

自维护的状态来说“退出”机制是要根据实际情况进行处理的，因此必须要由开发员人员通过编码来实现，这其中除了业务逻辑上决定的“主动”退出（例如接收到了某类session关闭的消息）之外，还需要有一种“保底”的推出机制：状态超时，对于某些有状态的流，可能并没有对应的关闭消息，可以约定在多长时间内没有收到消息就认定状态终结了，那这时就是基于时间阈值的判断，那就又会涉及到是基于事件时间还是处理时间，显然，Spark是同时支持两种模式的，只是有一点会让新人疑惑的是：在基于事件时间的有状态的流计算上，在API层面“似乎”没有给开发人员一个声明“哪个字段是事件时间”的地方，转而是这样约定的：**如果开发人员需要开发基于事件时间的有状态的流计算，则必须使用watermark机制，对应到代码层面就是，当你使用mapGroupsWithState(GroupStateTimeout.EventTimeTimeout())(yourGroupStateUpdateFunc)时，前面一定要先声明withWatermark("yourEnenTimeColumnName", yourWatermarkDuration)**

**从“状态”是由Spark维护还是自维护的角度总结是：**

1. **如果“状态”是由Spark维护的基于窗口的流，如果使用withWatermark(…)，则数据进入状态与状态退出都是基于eventTime的。如果没有使用withWatermark(…)，不存在基于哪种时间决定数据进入状态与状态退出，而是无条件进入与永不退出！**
2. **如果是“状态”自维护的计算，如果使用withWatermark(…)，则数据是否进入状态是基于eventTime判断的，状态退出可以基于eventTime或processingTime判断，取决于设置的是GroupStateTimeout.EventTimeTimeout还是GroupStateTimeout.ProcessTimeTimeout**

**从是否使用withWatermark的角度总结是：**

1. **一旦使用withWatermark(…), 就意味着：在决定数据是否能进入状态时，完全基于withWatermark指定的eventTime去计算，而在决定状态是否退出时，对于基于窗口的由Spark来维护的状态计算，会基于withWatermark指定的阈值基于eventTime进行判断，对于自维护的状态，可以基于eventTime，也可以基于processingTime。**
2. **但是如果没有使用withWatermark(…)，就意味着：在决定数据是否能进入状态时，是无条件的，所有数据都会进入状态，而在决定状态是否退出时，对于基于窗口的由Spark维护的状态计算，状态永不退出，对于自维护的状态，可以永不退出，如果基于时间进行退出，只能基于“processingTime”，因为没有使用withWatermark(…)就意味着没有开启时间时间框架，数据进入状态与状态退出都不能依赖eventTime。**