### 1.trigger

窗口的计算触发依赖于窗口触发器，每种类型的窗口都有对应的窗口触发机制，都有一个默认的窗口触发器，触发器的作用就是去控制什么时候来触发我们的聚合方法。Flink内部定义实现了如EventTimeTrigger、ProcessTimeTrigger、CountTrigger以及DeltaTrigger等等。一般地，每种触发器对应于不同的Window Assigner，例如EventTime类型的Windows对应的触发器是EventTimeTrigger，工作原理是判断当前的Watermark是否超过了窗口的EndTime，如果超过则触发对窗口内数据的计算，否则不触发计算。

(1)EventTimeTrigger：通过对比Watermark和窗口的Endtime确定是否触发窗口计算，如果Watermark大于Window EndTime则触发，否则不触发，窗口将继续等待。

(2)ProcessTimeTrigger：通过对比ProcessTime和窗口EndTme确定是否触发窗口，如果ProcessTime大于EndTime则触发计算，否则窗口继续等待。

(3)ContinuousEventTimeTrigger：根据间隔时间周期性触发窗口或者Window的结束时间小于当前EndTime触发窗口计算。

(4)ContinuousProcessingTimeTrigger：根据间隔时间周期性触发窗口或者Window的结束时间小于当前ProcessTime触发窗口计算。

(5)CountTrigger：根据接入数据量是否超过设定的阙值判断是否触发窗口计算。

(6)DeltaTrigger：根据接入数据计算出来的Delta指标是否超过指定的Threshold去判断是否触发窗口计算。

(7)PurgingTrigger：可以将任意触发器作为参数转换为Purge类型的触发器，计算完成后数据将被清理。

### 自定义触发器

OnElement ：每一个数据进入窗口都会触发。

OnEventTime ：根据接入窗口的EventTime进行触发操作

OnProcessTime ： 根据接入窗口的ProcessTime进行触发操作

Clear ： 执行窗口及状态数据的清除方法。

窗口触发方法返回结果的类型：

CONTINUE ： 不进行操作，等待。

FIRE ： 触发计算且数据保留。

PRUGE ： 窗口内部数据清除且不触发计算。

FIRE\_AND\_PURGE : 触发计算并清除对应的数据。

开发：

**package** com.sqq2;  
**import** org.apache.flink.api.common.functions.FlatMapFunction;  
**import** org.apache.flink.api.java.tuple.Tuple2;  
**import** org.apache.flink.streaming.api.datastream.DataStream;  
**import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
**import** org.apache.flink.streaming.api.windowing.time.Time;  
**import** org.apache.flink.streaming.api.windowing.triggers.Trigger;  
**import** org.apache.flink.streaming.api.windowing.triggers.TriggerResult;  
**import** org.apache.flink.streaming.api.windowing.windows.TimeWindow;  
**import** org.apache.flink.util.Collector;  
  
**public class** wr7 {  
  
 **public static void** main(String[] args) **throws** Exception {  
 StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 DataStream<String> text = env.socketTextStream(**"192.168.8.201"**, 9999);  
  
 DataStream<Tuple2<String, Integer>> windowCounts = text  
 .flatMap(**new** FlatMapFunction<String, Tuple2<String, Integer>>() {  
 @Override  
 **public void** flatMap(String value, Collector<Tuple2<String, Integer>> out) {  
 **for** (String word : value.split(**" "**)) {  
 out.collect(Tuple2.*of*(word, 1));  
 }  
 }  
 })  
 .keyBy(0)  
 .timeWindow(Time.*seconds*(100))  
 .trigger(CustomTrigger1.*create*())  
 .sum(1);  
  
 windowCounts.print().setParallelism(1);  
  
 env.execute(**"wr6"**);  
 }  
}  
  
**class** CustomTrigger1 **extends** Trigger<Object, TimeWindow> {  
 **private static final long *serialVersionUID*** = 1L;  
  
 **private static int** *flag* = 0;  
  
 @Override  
 **public** TriggerResult onElement(Object element, **long** timestamp, TimeWindow window, TriggerContext ctx) **throws** Exception {  
 ctx.registerEventTimeTimer(window.maxTimestamp());  
 **if** (*flag* > 4) {  
 *flag* = 0;  
 **return** TriggerResult.***FIRE***;  
 }**else**{  
 *flag* ++;  
 }  
 System.***out***.println(**"onElement: "** + element);  
 **return** TriggerResult.***CONTINUE***;  
 }  
  
 @Override  
 **public** TriggerResult onEventTime(**long** time, TimeWindow window, TriggerContext ctx) **throws** Exception{  
 **return** TriggerResult.***CONTINUE***;  
 }  
  
 @Override  
 **public** TriggerResult onProcessingTime(**long** time, TimeWindow window, TriggerContext ctx) **throws** Exception {  
 **return** TriggerResult.***FIRE***;  
 }  
  
 @Override  
 **public void** clear(TimeWindow window, TriggerContext ctx) **throws** Exception{  
 ctx.deleteProcessingTimeTimer(window.maxTimestamp());  
 }  
  
 @Override  
 **public** String toString(){  
 **return "CustomTrigger"**;  
 }  
  
 **public static** CustomTrigger create(){  
 **return new** CustomTrigger();  
 }  
}