Flink-day5笔记

复习

- 状态：flink是一个基于状态的流计算引擎
    1. flink自身通过状态完成流计算处理
    2. flink支持自定义function

       可以通过使用状态对象完成对应的业务处理

       MapFunction/ProcessFunction

       open方法：创建状态对象

       map方法/process方法：通过状态完成业务功能的实现
    3. 状态开发/通过状态存储数据

       根据数据流对状态进行了划分：keyed state+operator state(non-keyed state)

       根据flink是否可管理：managed state+raw state

       managed keyed state

状态类	描述	方法
ValueState	存储单一值	update/value
ListState	存储多个值	add/update/get
MapState	存储的是多个key-value结构的数据	contains/get/put/keys/values/entries
ReducingState	存储单一值;可以自动运算	add/get
AggregatingState	存储单一值;可以自动运算;输入和输出类型可以不一致（支持中间类型）	add/get

       所有的状态都可以使用clear方法，对状态中存储的数据进行清楚
    4. 状态进阶理解
        1. TTL(Time To Live)-剩余存活时间
            - 设置时间
            - 设置时间的更新机制
            - 设置过期数据的清理机制==》GC
        2. Checkpoint-->状态中的数据进行持久化

           由jobmanager发起，把barrier添加到数据流中，所有的算子在接收到barrier的时候都应该做预提交处理。所有的算子都做了对应的应当，这一次checkpoint才算完成

           新的checkpoint生成，就会把老的checkpoint丢弃掉

3. state backend：决定数据持久化到哪里

   jobmanager:持久化到jobmanager的内存中

   filesystem：hdfs（flink和hadoop集成起来才可以正常使用）

   rocksdb：flink内置的一个基于内存和磁盘的数据库，以key-value结构的方式存储数据的数据库

4. 广播状态

   能够在一个数据流中使用到另外一个数据流中的内容

   把低吞吐量流转换成广播流，把数据放入到状态中

   在高吞吐量流里面读取到状态中的数据

   根据高吞吐量流是dataStream还是keyedStream完成对应的自定义processFunction

   dataStream:BroadcastProcessFunction

   keyedStream:KeyedBroadcastProcessFunction

   processFunction里面提供的由俩个方法完成数据的处理

5. 状态可查询

   状态可查询只支持keyed state

   flink提供了一个功能，允许第三方程序到flink服务器中获取到状态的数据

   - flink服务器需要激活状态可查询
   - 在代码中设置，状态是可查询的：descriptor.setQueryable
   - 把代码部署到服务器

   状态可查询的架构

   - queryableStateClient:客户端--》第三方程序
   - queryableStateClientProxy:客户端代理，他是taskmanager里面的一个角色，用来接收client
   - queryableStateServer:存储状态

   状态可查询的整体执行流程

   1. 客户端发送请求到taskmanager，taskmanager里面的querableStateClientProxy负责处理请求。
   2. taskmanager需要向jobmanager发送请求确认要查找key对应的状态数据在哪里存储（哪个taskmanager）
   3. jobmanager会响应一个taskmanager
   4. 到对应的taskmanager的queryableStateServer里面读取到这个可以对应的状态，响应给客户端

Windows(流计算核心)

参考https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/stream/operators/windows.html

```
Windows are at the heart of processing infinite streams. Windows split the stream into “buckets”
of finite size, over which we can apply computations.
```

窗口是处理无限流的核心。Windows将流分成有限大小的“bucket”，我们可以在上面进行计算。

在Flink中，将窗口划分成了两大类：keyed窗口和non-keyed窗口。代码结构如下：

Keyed windows

```
stream
       .keyBy(...)               <-  keyed versus non-keyed windows
       .window(...)              <-  required: "assigner"//分配器，用来指定如何生成窗口
      [.trigger(...)]            <-  optional: "trigger" (else default trigger)//触发器，何时开始
计算
      [.evictor(...)]            <-  optional: "evictor" (else no evictor)//剔除器，数据是否需要剔
除
      [.allowedLateness(...)]    <-  optional: "lateness" (else zero)
      [.sideOutputLateData(...)] <-  optional: "output tag" (else no side output for late data)
       .reduce/aggregate/fold/apply()      <-  required: "function"//window function，计算
      [.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"
```

从上面的代码结构看，keyed windows与non-keyed windows的唯一区别就是，keyedStream上通过keyBy().window()完成而non-keyedStream通过windowAll()

Window Lifecycle

当应该属于某一个窗口的第一个元素到达时，该窗口被创建。当时间超过了窗口的结束时间加上允许的延迟时间是，窗口被彻底删除。Flink只删除基于时间的窗口，而不删除像全局窗口这样的其他类型的窗口。

每个window都会绑定一个function和一个trigger,function的作用是对窗口中的内容进行计算，trigger的作用是决定什么时候开始应用function进行计算---》触发器决定了窗口什么时候处于就绪状态

除了function和trigger之外，还可以指定evictor。evictor的作用是在trigger触发之后到function运行之前和/或者运行之后这段时间间，把window中的元素删除

Keyed vs Non-Keyed Windows

Keyed Windows：

keyed Stream会根据key讲原始流切分成多个逻辑keyed Stream，每个逻辑keyed Stream都可以独立于其他任务进行处理，所以Keyed Stream允许多个并行任务执行窗体计算。同一个key的所有元素被分发到同一个并行任务中。简而言之，在某一个时刻，会触发多个window任务，取决于Key的种类。

Non-Keyed Windows：没有key的概念，所以不会将原始流拆分成多个逻辑流，所有窗口逻辑将有单个任务执行，也就是说并行度是1。简而言之，任意时刻只有一个window任务执行

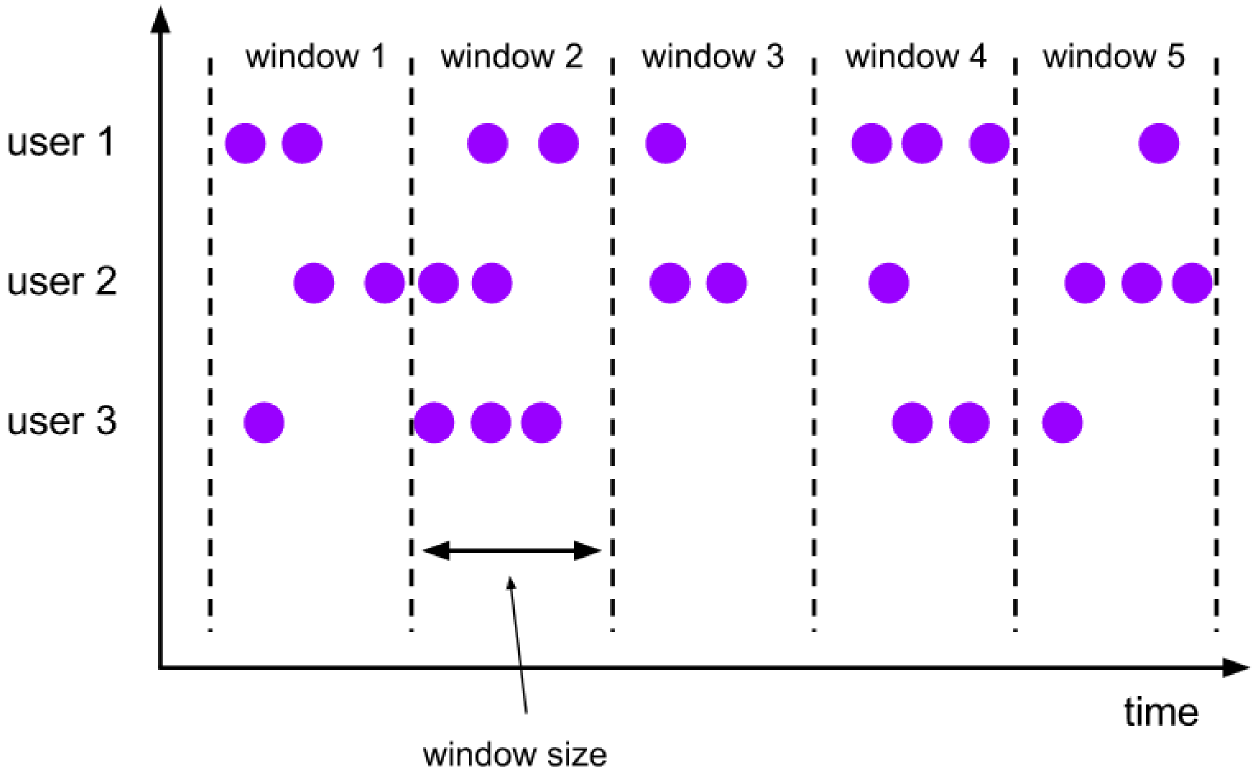
Window Assigners:指派器/分配器；怎么切分无限数据流

Window Assigner定义了如何将元素划分给窗口。通过调用window()方法或者windowAll()方法传递一个WindowAssigner对象参数完成。

WindoAssigner负责将传递过来的元素分配给1个或多个窗口。Flink中提供了一些WindowAssigner： *tumbling windows*, *sliding windows*, *session windows* and *global windows*。也可以自定义WindowAssigner，通过继承WindowAssigner类完成自定义。除了global windows之外的其他窗口，都是基于时间的窗口，这类窗口会有一个开始时间戳（包含）和一个结束时间戳（不包含）用来描述窗口的大小

Tumbling Windows（滚动窗口）

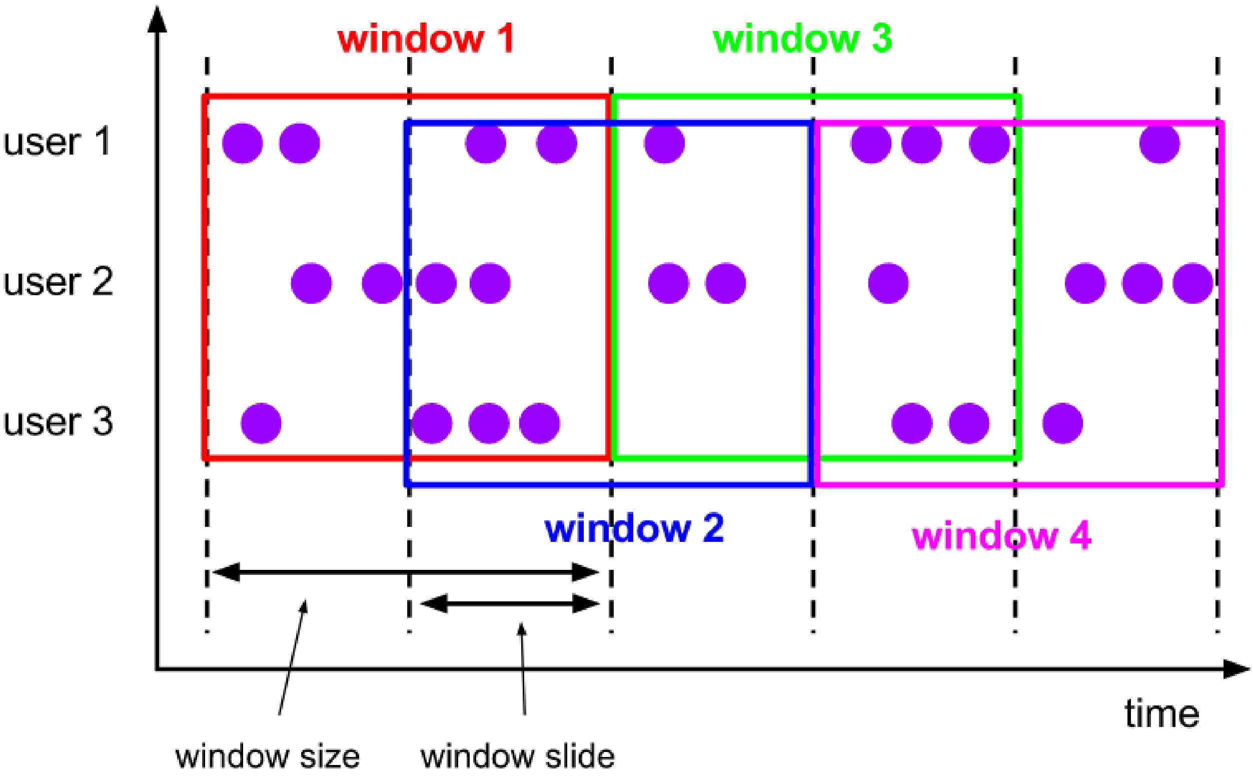
滚动窗口分配器将每个元素分配给指定大小的窗口。滚动窗口具有固定的大小，并且不重叠。例如，如果指定一个大小为5分钟的滚动窗口，则将计算当前窗口，并每隔5分钟启动一个新窗口，如下图所示。



```
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.streaming.api.scala._

object TumblingWindows {
  //Tumbling Windows
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      .keyBy(0)

      .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
      .reduce((v1, v2) => (v1._1, v1._2 + v2._2))

    result.print()

    environment.execute("Tumbling Windows")
  }

}
```

Sliding Windows（滑动窗口）

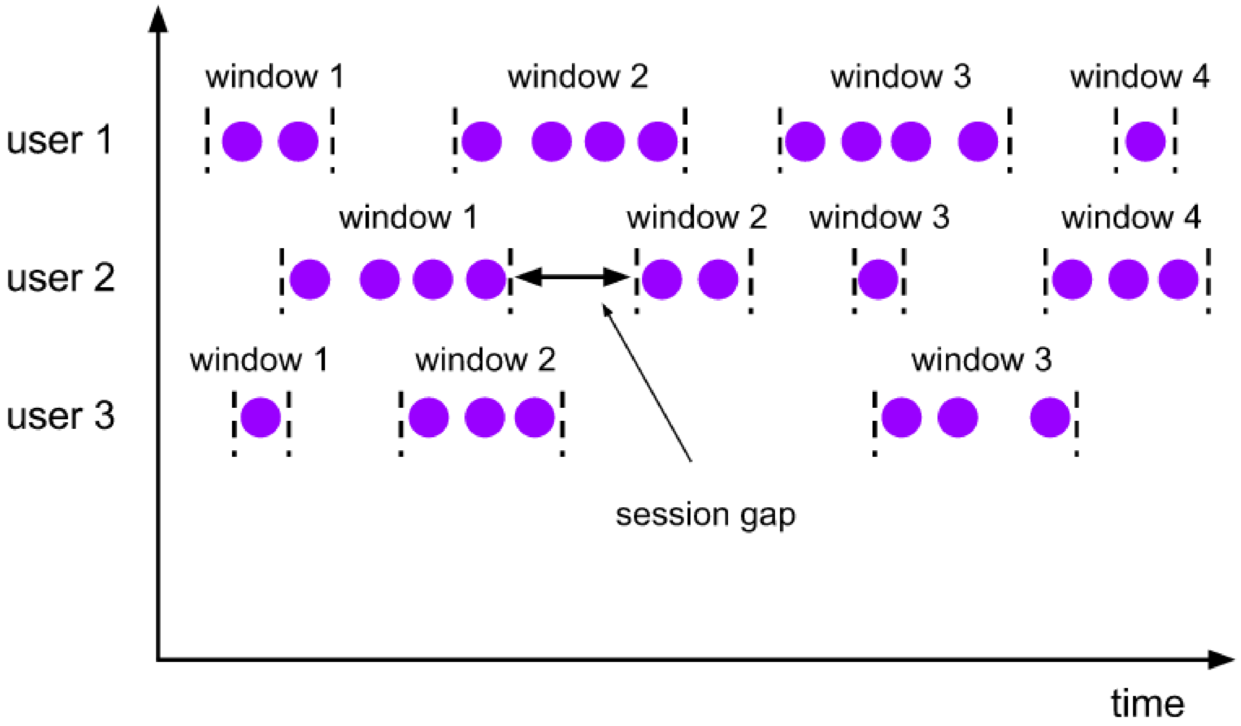
滑动窗口分配器将元素分配给固定长度的窗口。与滚动窗口分配程序类似，窗口大小由窗口大小参数配置。另一个，窗口滑动参数控制滑动窗口的创建频率。因此，如果滑动参数小于窗口大小，则滑动窗口可能重叠。在这种情况下，元素被分配给多个窗口。



```
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.{SlidingProcessingTimeWindows,
TumblingProcessingTimeWindows}
import org.apache.flink.streaming.api.windowing.time.Time

object SlidingWindows {
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      .keyBy(0)
      .window(SlidingProcessingTimeWindows.of(Time.seconds(10),Time.seconds(5)))
      .reduce((v1, v2) => (v1._1, v1._2 + v2._2))
    //也可以使用aggregate，结合自定义aggregateFunction完成求和统计

    result.print()

    environment.execute("Sliding Windows")
  }

}
```

Session Windows

会话窗口分配器按活动的会话对元素进行分组。与滚动窗口和滑动窗口相比，会话窗口不重叠，也没有固定的开始和结束时间。相反，当会话窗口在一段时间内没有接收到元素（也就是说当出现不活动的间隙）时，会话窗口将关闭。会话窗口分配器可以配置为静态会话间隙，也可以自定义会话间隙。当此时间段过期时，当前会话将关闭，后续元素将分配给新会话窗口。

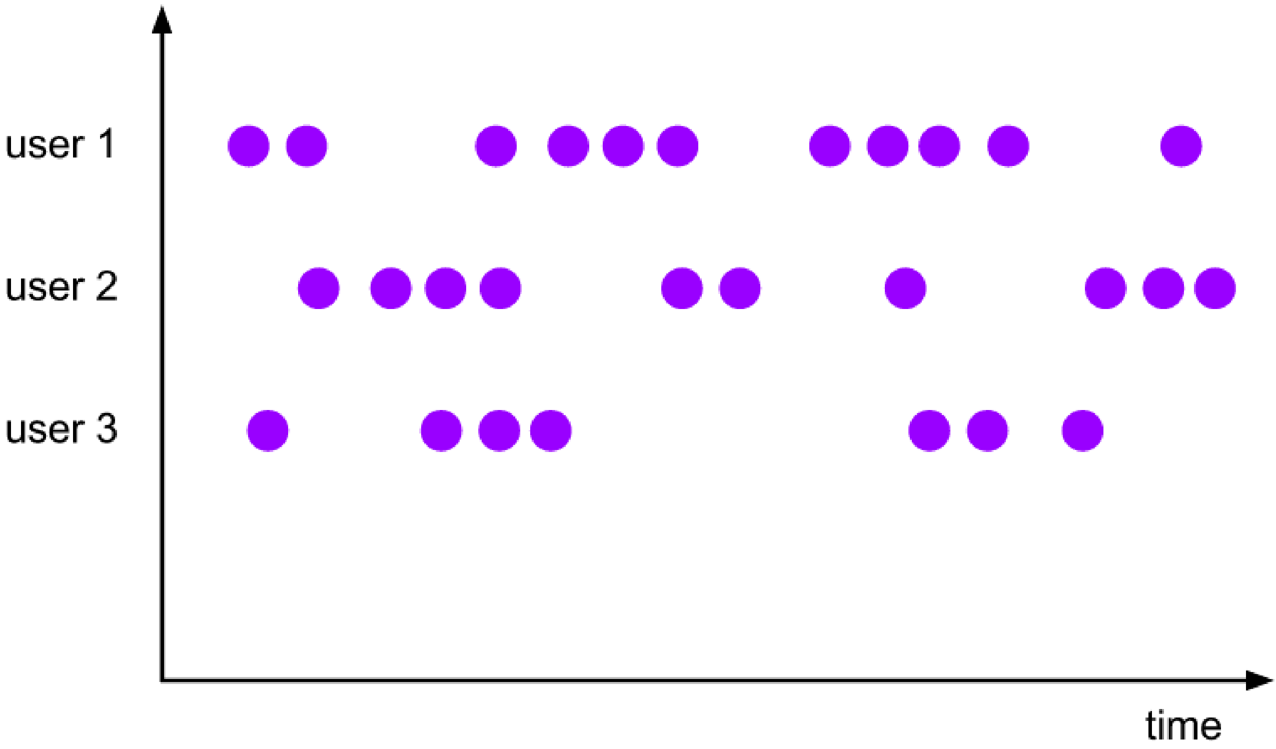


```
import org.apache.flink.streaming.api.scala.function.WindowFunction
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.util.Collector

class MyWindowFunctionForSessionWindows extends  WindowFunction[(String,Int),(String,Int),String,TimeWindow]{
  override def apply(key: String, window: TimeWindow, input: Iterable[(String, Int)], out: Collector[(String, Int)]): Unit = {

    //统计
    val sum = input.map(_._2).sum

    var start = window.getStart
    var end = window.getEnd
    println(start+"~"+end+"==>total milleseconds:"+(end-start)+"。==》 "+key+":"+sum)
    }
  }
```

```
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.{ProcessingTimeSessionWindows,
SlidingProcessingTimeWindows}
import org.apache.flink.streaming.api.windowing.time.Time

object SessionWindows {
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      //特别注意这里，不能keyBy(0)
      .keyBy(t=>t._1)
      .window(ProcessingTimeSessionWindows.withGap(Time.seconds(5)))
      .apply(new MyWindowFunctionForSessionWindows)

    result.print()

    environment.execute("Session Windows")
  }
}
```

Global Windows

全局窗口分配器将具有相同键的所有元素分配给同一个全局窗口。只有指定了触发器时，此窗口才可用。否则，将不会执行任何计算，因为全局窗口没有可以处理聚合元素的自然结束。



```
import org.apache.flink.streaming.api.scala.function.WindowFunction
import org.apache.flink.streaming.api.windowing.windows.GlobalWindow
import org.apache.flink.util.Collector

class MyWindowFunctionForGlobalWindows extends WindowFunction[(String,Int),
(String,Int),String,GlobalWindow]{
  override def apply(key: String, window: GlobalWindow, input: Iterable[(String, Int)], out:
Collector[(String, Int)]): Unit = {
    val sum = input.map(_._2).sum

    out.collect((key,sum))
  }
}
```

```
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.GlobalWindows
import org.apache.flink.streaming.api.windowing.triggers.CountTrigger

object MyGlobalWindows {
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      //特别注意这里，不能keyBy(0)
      .keyBy(t=>t._1)
      .window(GlobalWindows.create())
      .trigger(CountTrigger.of(4))
      .apply(new MyWindowFunctionForGlobalWindows)

    result.print()

    environment.execute("Global Windows")
  }
}
```

Window Functions

定义窗口分配器之后，我们需要指定要在每个窗口上执行的计算，这是Window Function的职责。一旦系统确定窗口已准备好处理时，就可以处理每个窗口的元素。窗口函数可以是ReduceFunction， AggregateFunction，FoldFunction、ProcessWindowFunction、WindowFunction（古董）之一。其中 ReduceFunction和AggregateFunction在运行效率上比ProcessWindowFunction高，因为前俩个执行的是增量计算，只要有数据抵达窗口，系统就会ReduceFunction，AggregateFunction实现增量计算；ProcessWindowFunction在窗口触发之前会一直缓存接收数据，只有当窗口就绪的时候才会对窗口中 的元素做批量计算，但是该方法可以获取窗口的元数据信息。因此可以通过将ProcessWindowFunction与 ReduceFunction，AggregateFunction结合使用，即可以获得窗口元素的增量聚合，又可以接收到窗口元数据。

ReduceFunction

```scala
import org.apache.flink.api.common.functions.ReduceFunction

class MyReduceFunction extends ReduceFunction[(String,Int)]{
  /**
    *
    * @param value1 累计值
    * @param value2 新值
    * @return 聚合之后的结果
    */
  override def reduce(value1: (String, Int), value2: (String, Int)): (String, Int) = {
    //验证一下增量的概念：每进来一个元素都会执行一次打印语句
    println("value1:"+value1+",value2:"+value2)

    (value1._1,value1._2+value2._2)
  }
}
```

```scala
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.streaming.api.scala._

object TumblingWindowsUsingReduceFunction {
  //Tumbling Windows
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      .keyBy(0)
      .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
      .reduce(new MyReduceFunction)

    result.print()

    environment.execute("Tumbling Windows")
  }

}
```

AggregateFunction

```scala
import org.apache.flink.api.common.functions.AggregateFunction

class MyAggregateFunction extends AggregateFunction[(String,Int),(String,Int),(String,Int)]{
  override def createAccumulator(): (String, Int) = {
    ("",0)
  }

  override def add(value: (String, Int), accumulator: (String, Int)): (String, Int) = {
    (value._1,value._2+accumulator._2)
  }

  override def getResult(accumulator: (String, Int)): (String, Int) = {
    accumulator
  }

  override def merge(a: (String, Int), b: (String, Int)): (String, Int) = {
    (a._1,a._2+b._2)
  }
}
```

```
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.{SlidingEventTimeWindows,
SlidingProcessingTimeWindows, TumblingProcessingTimeWindows}
import org.apache.flink.streaming.api.windowing.time.Time

object TumblingWindowsUsingAggregateFunction {
  //Tumbling Windows
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      .keyBy(0)
      .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
      .aggregate(new MyAggregateFunction)

    result.print()

    environment.execute("Tumbling Windows")
  }

}
```

ProcessWindowFunction

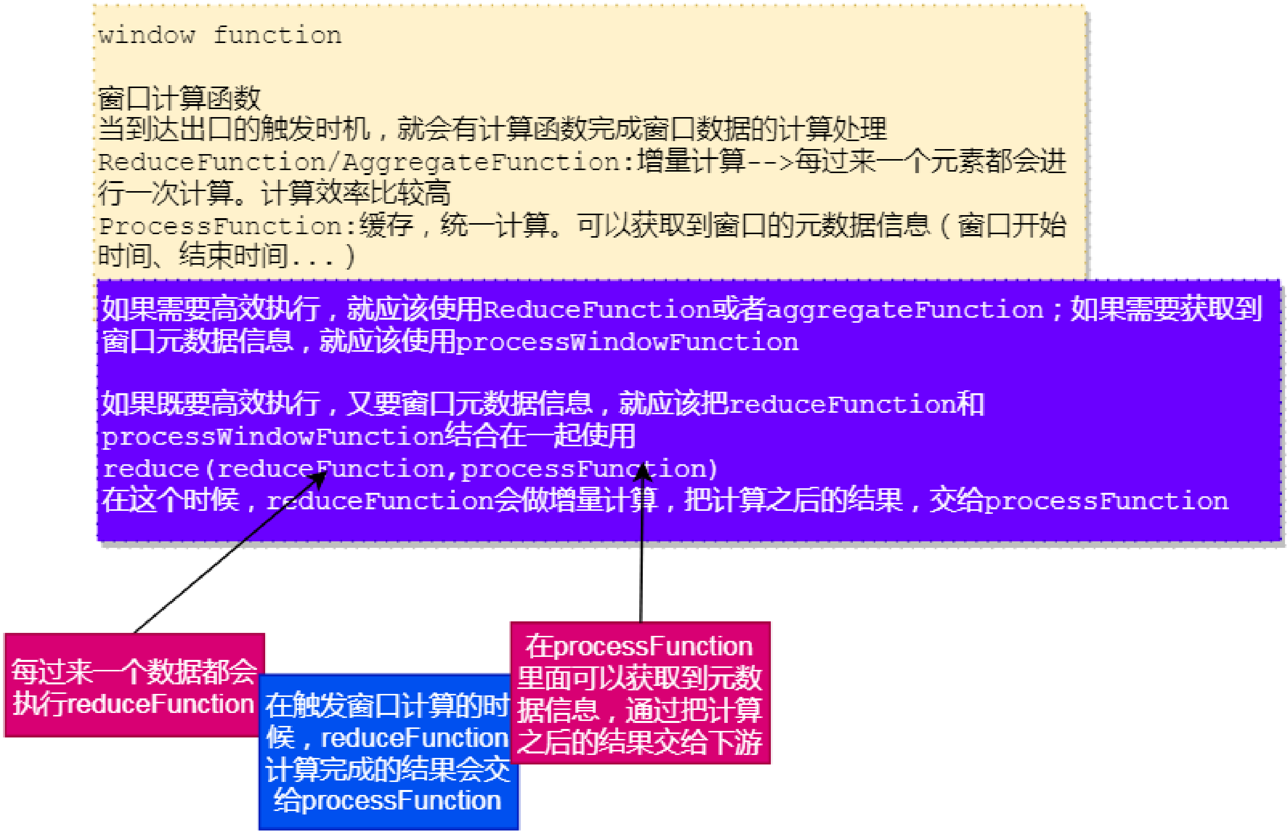
```
import java.text.SimpleDateFormat

import org.apache.flink.streaming.api.scala.function.ProcessWindowFunction
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.util.Collector


/**
```

```
  * 注意：import org.apache.flink.streaming.api.scala.function.ProcessWindowFunction
  */
class MyProcessWindowFunction extends ProcessWindowFunction[(String,Int),
(String,Int),String,TimeWindow]{
  private val format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")

  override def process(key: String, context: Context, elements: Iterable[(String, Int)], out:
Collector[(String, Int)]): Unit = {
    //获取到window对象
    val window: TimeWindow = context.window

    val start: Long = window.getStart
    val end: Long = window.getEnd

    val startStr: String = format.format(start)
    val endStr: String = format.format(end)

    val total: Int = elements.map(_._2).sum
    out.collect(startStr+"~"+endStr+"==>key:"+key,total)

  }
}
```

```
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time

object TumblingWindowsUsingProcessWindowFunction {
  //Tumbling Windows
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      .keyBy(_._1)
      .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
      .process(new MyProcessWindowFunction)
    result.print()

    environment.execute("Tumbling Windows")
  }

}
```

ProcessWindowFunction with Incremental Aggregation（重点）--processWindowFunction实现增量计算

通过ProcessWindowFunction实现既获取窗口元数据，又要做增量计算



window function

窗口计算函数
当到达出口的触发时机，就会有计算函数完成窗口数据的计算处理
ReduceFunction/AggregateFunction:增量计算-->每过来一个元素都会进行一次计算。计算效率比较高
ProcessFunction:缓存，统一计算。可以获取到窗口的元数据信息（窗口开始时间、结束时间...）

如果需要高效执行，就应该使用ReduceFunction或者aggregateFunction；如果需要获取到窗口元数据信息，就应该使用processWindowFunction

如果既要高效执行，又要窗口元数据信息，就应该把reduceFunction和processWindowFunction结合在一起使用
reduce(reduceFunction,processFunction)
在这个时候，reduceFunction会做增量计算，把计算之后的结果，交给processFunction

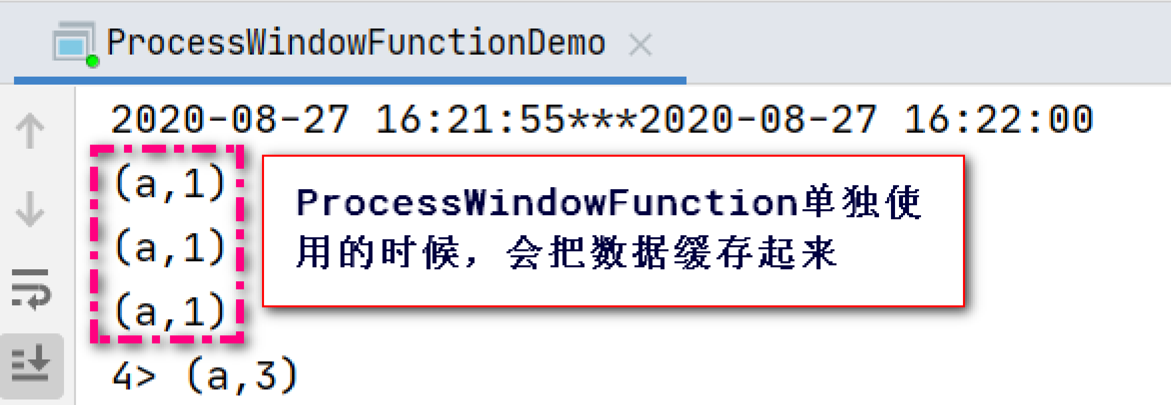
每过来一个数据都会执行reduceFunction

在触发窗口计算的时候，reduceFunction计算完成的结果会交给processFunction

在processFunction里面可以获取到元数据信息，通过把计算之后的结果交给下游

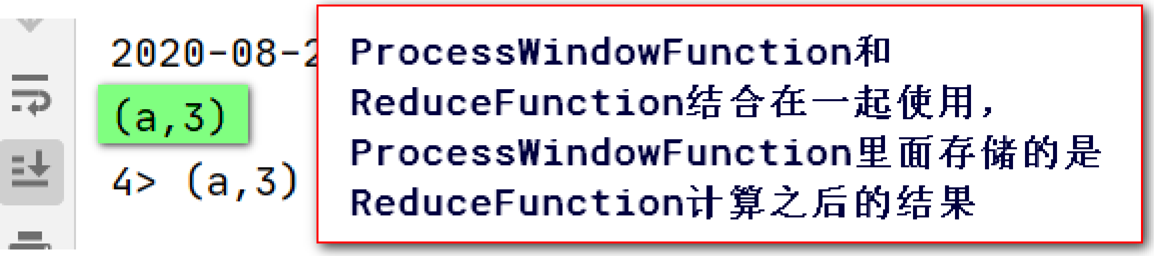
```scala
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time

object TumblingWindowsUsingProcessWindowFunctionAndReduceFunction {
  //Tumbling Windows
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      .keyBy(_._1)
      .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
      .reduce(new MyReduceFunction,new MyProcessWindowFunction)
    result.print()

    environment.execute("Tumbling Windows")
  }

}
```

> note:
>
> 注意观察MyProcessWindowFunction中process方法里的elements



```
2020-08-27 16:21:55***2020-08-27 16:22:00
(a,1)
(a,1)
(a,1)
4> (a,3)
```

ProcessWindowFunction单独使用的时候，会把数据缓存起来



```
2020-08-2
(a,3)
4> (a,3)
```

ProcessWindowFunction和ReduceFunction结合在一起使用，ProcessWindowFunction里面存储的是ReduceFunction计算之后的结果

Using per-window state in ProcessWindowFunction（重点）-在processWindowFunction里面使用状态

通过ProcessWindowFunction可以获取到每一个窗口的状态数据 windowState，也可以获取到所有窗口汇总数据globalState

There are two methods on the `Context` object that a `process()` invocation receives that allow access to the two types of state:

- `globalState()`, which allows access to keyed state that is not scoped to a window
- `windowState()`, which allows access to keyed state that is also scoped to the window

```
import java.text.SimpleDateFormat

import org.apache.flink.api.common.state.{ValueState, ValueStateDescriptor}
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.scala.function.ProcessWindowFunction
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.util.Collector
import org.apache.flink.streaming.api.scala._

class MyProcessWindowFunctionWithState extends ProcessWindowFunction[(String,Int),
(String,Int),String,TimeWindow]{
  private val format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")

  var windowStateDescriptor:ValueStateDescriptor[Int]=_
  var globalStateDescriptor:ValueStateDescriptor[Int]=_


  override def open(parameters: Configuration): Unit = {
    windowStateDescriptor = new ValueStateDescriptor[Int]
("windowStateDescriptor",createTypeInformation[Int])

    globalStateDescriptor = new ValueStateDescriptor[Int]
("globalStateDescriptor",createTypeInformation[Int])

  }

  override def process(key: String, context: Context, elements: Iterable[(String, Int)], out:
Collector[(String, Int)]): Unit = {
    //获取到window对象
    val window: TimeWindow = context.window

    val start: Long = window.getStart
    val end: Long = window.getEnd

    val startStr: String = format.format(start)
    val endStr: String = format.format(end)


    val total: Int = elements.map(_._2).sum

    val windowState: ValueState[Int] = context.windowState.getState(windowStateDescriptor)
    val globalState: ValueState[Int] = context.globalState.getState(globalStateDescriptor)

    windowState.update(windowState.value()+total)
    globalState.update(globalState.value()+total)

    println(windowState.value()+"***"+globalState.value())

    out.collect(startStr+"~"+endStr+"==>key:"+key,total)
  }
}
```

```
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time

object TumblingWindowsUsingProcessWindowFunctionState {
  //Tumbling Windows
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      .keyBy(_._1)
      .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
      .process(new MyProcessWindowFunctionWithState)
    result.print()

    environment.execute("Tumbling Windows")
  }

}
```

WindowFunction (Legacy)-不说

在某些可以使用ProcessWindowFunction的地方，也可以使用WindowFunction。这是较旧版本的ProcessWindowFunction。WindowFunction提供的不是上下文对象而是window对象，所以在使用ProcessWindowFunction中通过上下文对象获取的信息，在使用WindowFunction时就没有办法继续使用。比如说globalState

```
import java.text.SimpleDateFormat

import org.apache.flink.streaming.api.scala.function.WindowFunction
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.util.Collector

class MyWindowFunction extends WindowFunction[(String,Int),(String,Int),String,TimeWindow]{

  private val format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
  override def apply(key: String, window: TimeWindow, input: Iterable[(String, Int)], out:
Collector[(String, Int)]): Unit = {

    val start: Long = window.getStart
    val end: Long = window.getEnd

    val startStr: String = format.format(start)
    val endStr: String = format.format(end)


    val total: Int = input.map(_._2).sum
    out.collect(startStr+"~"+endStr+"==>key:"+key,total)
  }
}
```

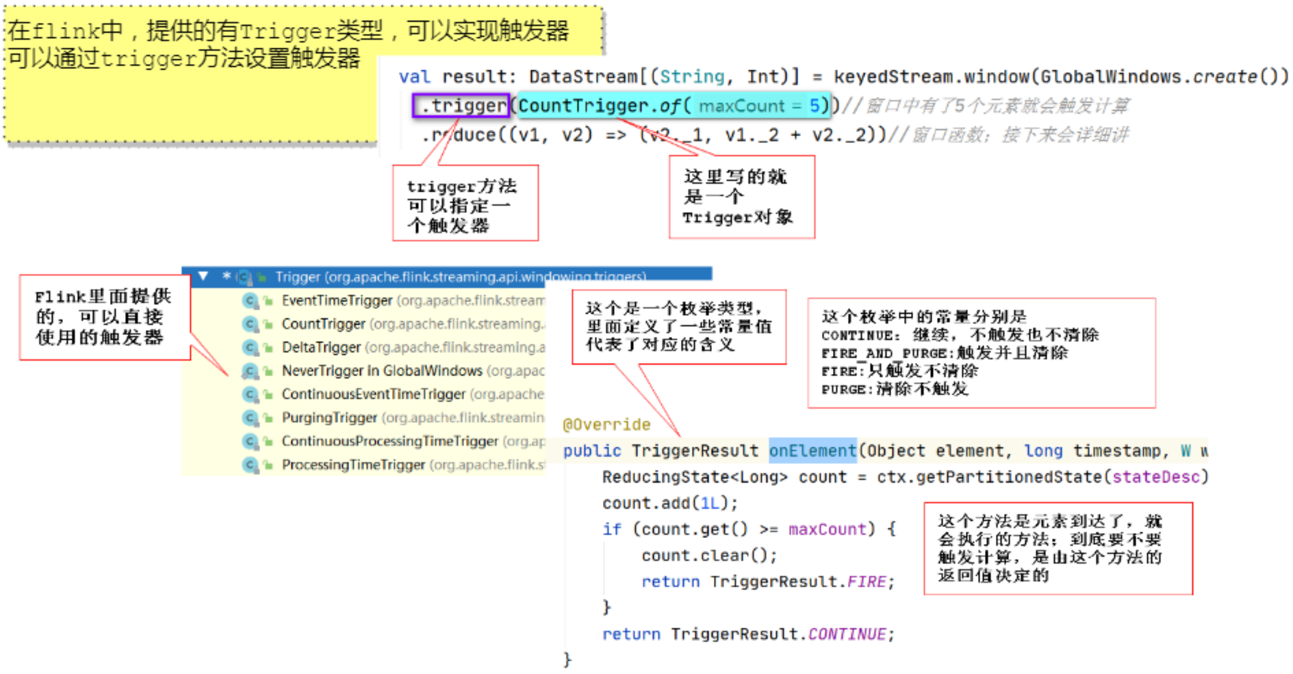
```
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time

object TumblingWindowsUsingWindowFunction {
  //Tumbling Windows
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      .keyBy(t=>t._1)
      .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
      .apply(new MyWindowFunction)

    result.print()

    environment.execute("Tumbling Windows")
    }

  }
```

Triggers（高级特性-触发器）

Trigger决定了窗口何时可以通过windowFunction进行计算。也就是说Trigger确定了窗口何时就绪。每一个windowAssinger都有一个默认的trigger，当默认的trigger不能满足需要时，可以自定义trigger

默认触发器

窗口类型	触发器	触发时机
ProcessingTimeWindows (TumblingProcessingTimeWindows SlidingProcessingTimeWindows ProcessingTimeSessionWindows)	ProcessingTimeTrigger	A Trigger that fires once the current system time passes the end of the window 系统时间超过了窗口的最后时间就会触发
EventTimeWindows (TumblingEventTimeWindows SlidingEventTimeWindows EventTimeSessionWindows)	EventTimeTrigger	A Trigger that fires once the watermark passes the end of the window
GlobalWindows	NeverTrigger	A trigger that never fires 永不触发



在flink中，提供的有Trigger类型，可以实现触发器
可以通过trigger方法设置触发器

```
val result: DataStream[(String, Int)] = keyedStream.window(GlobalWindows.create())
  .trigger(CountTrigger.of( maxCount = 5))//窗口中有了5个元素就会触发计算
  .reduce((v1, v2) => (v2._1, v1._2 + v2._2))//窗口函数：接下来会详细讲
```

trigger方法可以指定一个触发器

这里写的就是一个Trigger对象

Flink里面提供的，可以直接使用的触发器

Trigger (org.apache.flink.streaming.api.windowing.triggers)
EventTimeTrigger (org.apache.flink.stream
CountTrigger (org.apache.flink.streaming.
DeltaTrigger (org.apache.flink.streaming.a
NeverTrigger in GlobalWindows (org.apac
ContinuousEventTimeTrigger (org.apache
PurgingTrigger (org.apache.flink.streamin
ContinuousProcessingTimeTrigger (org.ap
ProcessingTimeTrigger (org.apache.flink.s

这个是一个枚举类型，里面定义了一些常量值代表了对应的含义

这个枚举中的常量分别是
CONTINUE：继续，不触发也不清除
FIRE_AND_PURGE：触发并且清除
FIRE：只触发不清除
PURGE：清除不触发

```
@Override
public TriggerResult onElement(Object element, long timestamp, W w
    ReducingState<Long> count = ctx.getPartitionedState(stateDesc)
    count.add(1L);
    if (count.get() >= maxCount) {
        count.clear();
        return TriggerResult.FIRE;
    }
    return TriggerResult.CONTINUE;
}
```

这个方法是元素到达了，就会执行的方法；到底要不要触发计算，是由这个方法的返回值决定的

自定义触发器（理解触发器中的定义机制）

对比现有的触发器实现自己定义的触发器

触发器接口中，需要重点关注五个方法。这些方法运行触发器对不同事件作出反应：

- onElement() -每添加一个元素到指定窗口，就会调用一次这个方法
- onEventTime()-当注册的event-time计时器触发时，会调用该方法
- onProcessingTime()-当注册的processing-time计时器触发时，会调用该方法
- onMerge()-当多个窗口合并到一个窗口时触发，例如session window
- clear() -在删除相应窗口时执行所需的任何操作，比如清除定时器、删除存储的状态等

以上五个方法，需要注意两件事情

- 前三个方法通过返回TriggerResult来决定，如何处理它们的调用事件
  - `CONTINUE`: 继续，当前窗口不做任何处理，
  - `FIRE`: 触发计算，
  - `PURGE`: 清理窗口中的元素
  - `FIRE_AND_PURGE`: 触发计算，然后清除窗口中的元素
- 这些方法中的任何一个都可以用于process-time和event-time计时器以完成后续的操作

show time

```scala
import org.apache.flink.streaming.api.scala.function.AllWindowFunction
import org.apache.flink.streaming.api.windowing.windows.GlobalWindow
import org.apache.flink.util.Collector

class MyAllWindowFunctionForGlobalWindows extends AllWindowFunction[String,String,GlobalWindow]{
  override def apply(window: GlobalWindow, input: Iterable[String], out: Collector[String]):
Unit = {

    val list: List[String] = input.toList
    println(list.mkString(" | "))
  }
}
```

```scala
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.GlobalWindows
import org.apache.flink.streaming.api.windowing.triggers.CountTrigger

object MyGlobalWindowsUsingTrigger {
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .windowAll(GlobalWindows.create())
      .trigger(CountTrigger.of(4))
      .apply(new MyAllWindowFunctionForGlobalWindows)

    environment.execute("Global Windows")
  }

}
```

结合CountTrigger源代码理解上述代码

高阶应用--自定义触发器（很少使用）

```
import org.apache.flink.api.common.functions.ReduceFunction
import org.apache.flink.api.common.state.{ReducingState, ReducingStateDescriptor}
import org.apache.flink.streaming.api.windowing.triggers.{Trigger, TriggerResult}
import org.apache.flink.streaming.api.windowing.windows.Window
import org.apache.flink.streaming.api.scala._

class MyCountTrigger(maxCount:Long) extends Trigger[String,Window]{

  var reduceFunction:ReduceFunction[Long]=new ReduceFunction[Long] {
    override def reduce(value1: Long, value2: Long): Long = {
      value1+value2
    }
  }
  var reducingStateDescriptor:ReducingStateDescriptor[Long]=new ReducingStateDescriptor[Long]("reducingStateDescriptor",reduceFunction,createTypeInformation[Long])

  override def onElement(element: String, timestamp: Long, window: Window, ctx: Trigger.TriggerContext): TriggerResult = {

    val reducingState: ReducingState[Long] = ctx.getPartitionedState(reducingStateDescriptor)

    reducingState.add(1L)
    if(reducingState.get()>=maxCount){
      reducingState.clear()
      return TriggerResult.FIRE_AND_PURGE
    }

    TriggerResult.CONTINUE
  }

  override def onProcessingTime(time: Long, window: Window, ctx: Trigger.TriggerContext): TriggerResult = {
    TriggerResult.CONTINUE
  }

  override def onEventTime(time: Long, window: Window, ctx: Trigger.TriggerContext): TriggerResult = {
    TriggerResult.CONTINUE
  }

  override def clear(window: Window, ctx: Trigger.TriggerContext): Unit = {
    ctx.getPartitionedState(reducingStateDescriptor).clear()
  }

}
```

```
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.GlobalWindows

object MyGlobalWindowsUsingTrigger {
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .windowAll(GlobalWindows.create())
//      .trigger(CountTrigger.of(4))
      .trigger(new MyCountTrigger(4))
      .apply(new MyAllWindowFunctionForGlobalWindows)

    environment.execute("Global Windows")
  }
}
```

对于上述代码，也可以切换成TumblingProecessingTimeWindow

> 扩展DeltaTrigger的使用（扩展）
>
> ```
> import org.apache.flink.streaming.api.functions.windowing.delta.DeltaFunction
> import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
> import org.apache.flink.streaming.api.windowing.assigners.GlobalWindows
> import org.apache.flink.streaming.api.windowing.triggers.DeltaTrigger
> import org.apache.flink.streaming.api.windowing.windows.GlobalWindow
>
> object MyGlobalWindowsUsingTrigger {
> def main(args: Array[String]): Unit = {
> val environment = StreamExecutionEnvironment.getExecutionEnvironment
> val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)
>
> val result: DataStream[Double] = dataStream.flatMap((line => line.split("\\s+")))
> .windowAll[GlobalWindow](GlobalWindows.create())
> .trigger(DeltaTrigger.of[String, GlobalWindow](2, new DeltaFunction[String] {
> override def getDelta(oldDataPoint: String, newDataPoint: String): Double = {
>  println(newDataPoint+"***"+oldDataPoint)
>  newDataPoint.toDouble - oldDataPoint.toDouble
> }
> }, createTypeInformation[String].createSerializer(environment.getConfig)))
> .apply(new MyAllWindowFunctionForGlobalWindowsDoubleType)
>
> environment.execute("Global Windows")
> }
> }
> ```

```
import org.apache.flink.streaming.api.scala.function.AllWindowFunction
import org.apache.flink.streaming.api.windowing.windows.GlobalWindow
import org.apache.flink.util.Collector

class MyAllWindowFunctionForGlobalWindowsDoubleType extends
AllWindowFunction[String,Double,GlobalWindow]{
override def apply(window: GlobalWindow, input: Iterable[String], out: Collector[Double]):
Unit = {

val list: List[String] = input.toList
println(list.mkString(" | "))
}
}
```

Evictors（高级特性-剔除器）

Flink的窗口，除了允许指定WindowAssigner以及Trigger，还可以指定Evictor，可以使用evictor(...)方法完成。evictor可以从窗口中移除元素。evictor会在trigger触发之后，window function执行之前或之后执行。

Evictor接口提供了两个方法

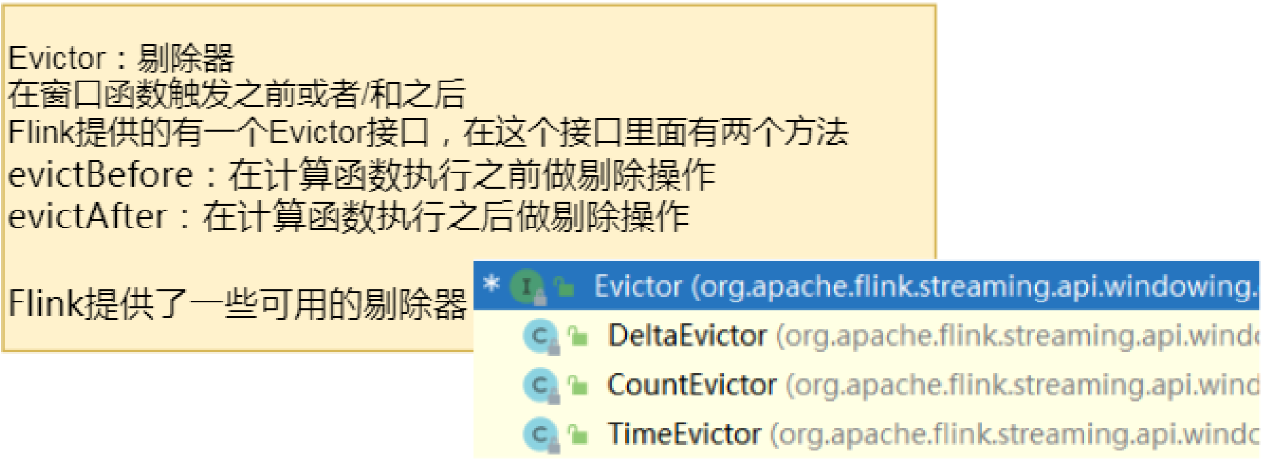
```
public interface Evictor<T, W extends Window> extends Serializable {

    /**
     * 在window function执行之前做剔除会调用这个方法
     * Optionally evicts elements. Called before windowing function.
     *
     * @param elements The elements currently in the pane.当前窗口中的所有元素
     * @param size The current number of elements in the pane.当前窗口中元素的个数
     * @param window The {@link Window} 当前窗口对象
     * @param evictorContext The context for the Evictor 上下午
     */
    void evictBefore(Iterable<TimestampedValue<T>> elements, int size, W window, EvictorContext
evictorContext);

    /**
     * 在window function执行之后调用这个方法
     * Optionally evicts elements. Called after windowing function.
     *
     * @param elements The elements currently in the pane.
     * @param size The current number of elements in the pane.
     * @param window The {@link Window}
     * @param evictorContext The context for the Evictor
     */
    void evictAfter(Iterable<TimestampedValue<T>> elements, int size, W window, EvictorContext
evictorContext);

}
```

Flink提供的Evictor

Flink提供了三个Evictor实现类

- CountEvictor-在窗口中保留用户指定数量的元素，并从窗口缓冲区的开头丢弃其余的元素
- DeltaEvictor-需要一个DeltaFunction和阈值，计算窗口缓冲区中最后一个元素与其余每个元素之间的增量，并删除增量大于或等于阈值的元素
- TimeEvictor-需要一个毫秒为单位的参数interval，对于给定的窗口，在其元素中查找最大时间戳max_ts，并删除时间戳小于max_ts-interval的所有元素

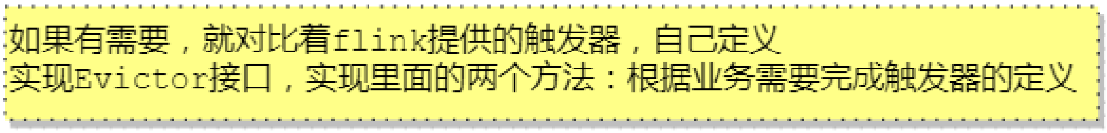
```
import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.GlobalWindows
import org.apache.flink.streaming.api.windowing.evictors.CountEvictor
import window.trigger.{MyAllWindowFunctionForGlobalWindows, MyCountTrigger}

object MyGlobalWindowsUsingCountEvictor {
  def main(args: Array[String]): Unit = {
    val environment = StreamExecutionEnvironment.getExecutionEnvironment
    val dataStream = environment.socketTextStream("flink.baizhiedu.com",9999)

    val result = dataStream.flatMap(line => line.split("\\s+"))
      .windowAll(GlobalWindows.create())
      .trigger(new MyCountTrigger(4))
      .evictor(CountEvictor.of(3))
      .apply(new MyAllWindowFunctionForGlobalWindows)

    environment.execute("Global Windows")
  }
}
```



Evictor：剔除器
在窗口函数触发之前或者/和之后
Flink提供的有一个Evictor接口，在这个接口里面有两个方法
evictBefore：在计算函数执行之前做剔除操作
evictAfter：在计算函数执行之后做剔除操作

Flink提供了一些可用的剔除器

Evictor (org.apache.flink.streaming.api.windowing.
DeltaEvictor (org.apache.flink.streaming.api.winde
CountEvictor (org.apache.flink.streaming.api.wind
TimeEvictor (org.apache.flink.streaming.api.windc



如果有需要，就对比着flink提供的触发器，自己定义
实现Evictor接口，实现里面的两个方法：根据业务需要完成触发器的定义

自定义Evictor（不常用）

参考CountEvictor完成自定义Evictor

```scala
import org.apache.flink.streaming.api.scala.function.ProcessAllWindowFunction
import org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.util.Collector

object MyTumblingWindowsUsingMyEvictor {
  def main(args: Array[String]): Unit = {
    val environment: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment

    val dataStream: DataStream[String] =
environment.socketTextStream("flink.baizhiedu.com",9999)

    dataStream.windowAll(TumblingProcessingTimeWindows.of(Time.seconds(5)))
      .evictor(new MyEvictor(false,"error"))
        .process(new MyProcessAllWindowFunction).print()

    environment.execute("MyTumblingWindowsUsingMyEvictorJob")
  }

}

class MyProcessAllWindowFunction extends ProcessAllWindowFunction[String,String,TimeWindow]{
  override def process(context: Context, elements: Iterable[String], out: Collector[String]):
Unit = {
    out.collect(elements.mkString(" | "))
  }
}
```

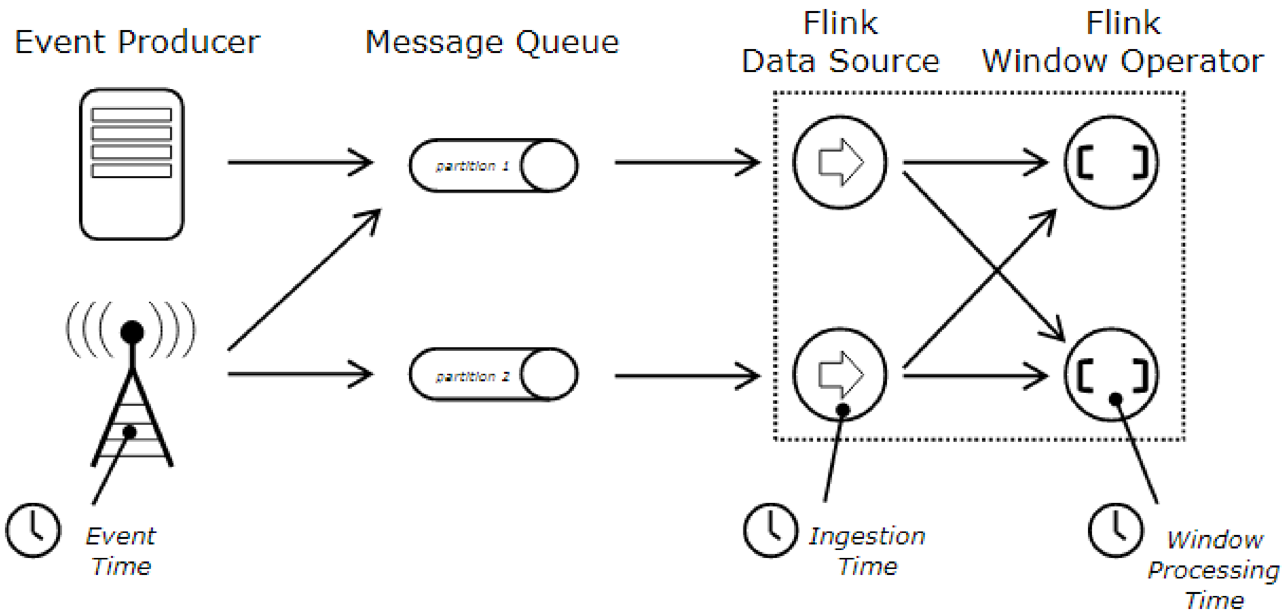
```scala
/**
  * 把一句话中包含指定单词的内容剔除掉
  */
class MyEvictor(doEvictAfter:Boolean,execludWords:String) extends Evictor[String,TimeWindow]{


  override def evictBefore(elements: lang.Iterable[TimestampedValue[String]], size: Int, window:
TimeWindow, evictorContext: Evictor.EvictorContext): Unit = {

    if(!doEvictAfter){
      evict(elements,size,evictorContext)
    }
  }

  override def evictAfter(elements: lang.Iterable[TimestampedValue[String]], size: Int, window:
TimeWindow, evictorContext: Evictor.EvictorContext): Unit = {

    if(doEvictAfter){
      evict(elements,size,evictorContext)
    }
  }
  private def evict(elements: lang.Iterable[TimestampedValue[String]], size: Int, context:
Evictor.EvictorContext):Unit={

    val iterator: util.Iterator[TimestampedValue[String]] = elements.iterator()
    while(iterator.hasNext){
      val element: TimestampedValue[String] = iterator.next()
      if(element.getValue.contains(execludWords)){
        iterator.remove()
      }
    }
  }
}
```

Event-time Window

Event Time

Flink在流计算的过程中，支持多种时间概念。Event Time / Processing Time / Ingestion Time

- Processing Time：处理时间是指执行相应操作的机器的系统时间。
- Event Time：事件时间是每个事件在其生产设备上发生的时间。处理乱序数据（数据的处理和数据的生成顺序乱啦）
- Ingestion：摄取时间是事件进入Flink的时间



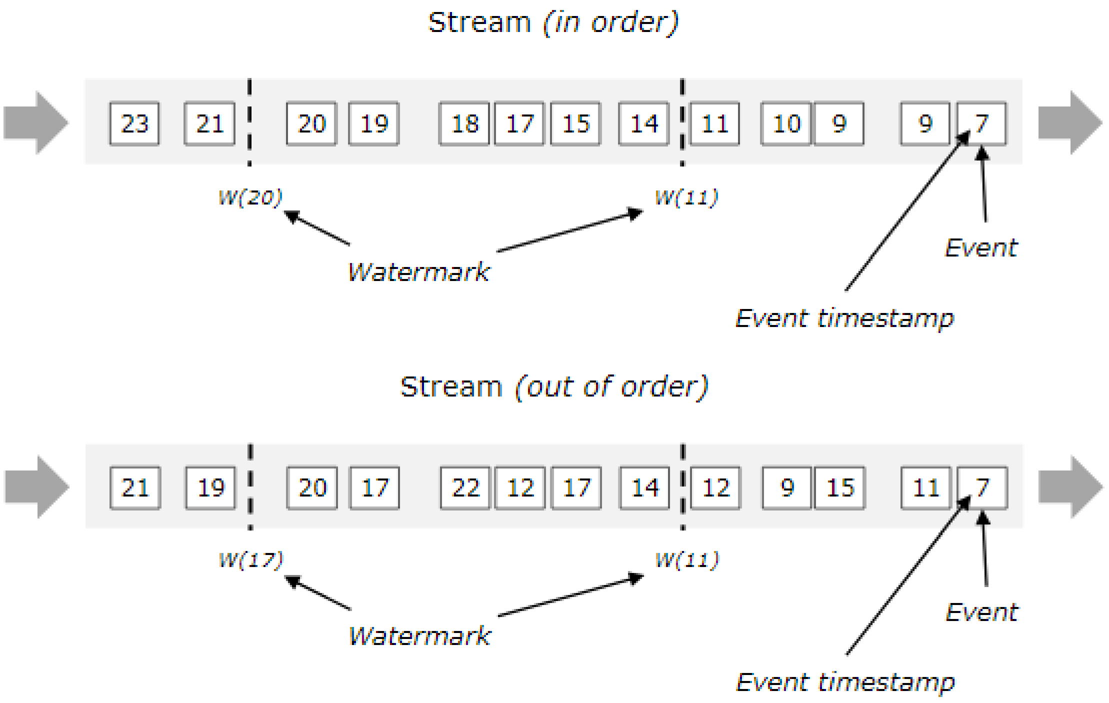
在Flink的窗口计算中，如果Flink在使用的时候不做显示声明，默认使用的是ProcessingTime。IngestionTime和ProcessingTime类似都是由系统自动产生，不同的是IngestionTime是由DataSource产生，ProcessingTime由算子产生。因此以上两种时间策略都不能很好的表达在流计算中事件产生时间（因为存在网络延时迟）。

Flink支持事件时间策略。事件时间是每个事件在其生产设备上发生的时间。此时间通常在记录进入Flink之前嵌入到记录中，并且可以从每个记录中提取事件时间戳。在事件时间中，时间的进度取决于数据，而不是任何时钟。事件时间程序必须指定如何生成事件时间*Watermarks*，这是事件时间进程的信号机制

Watermark-水位线

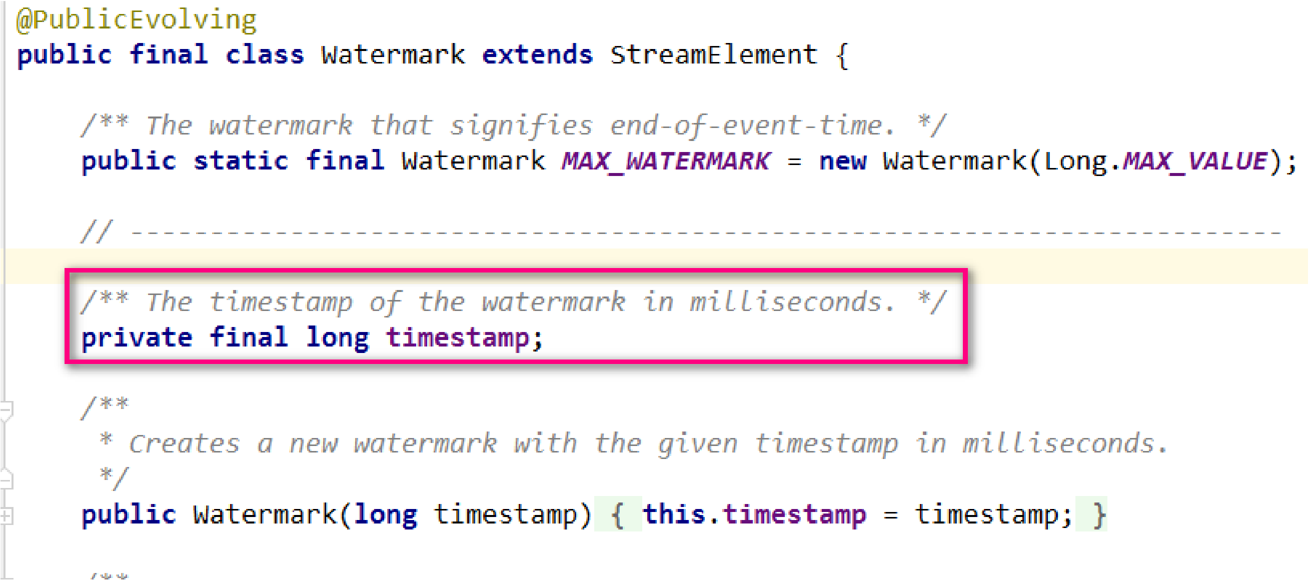
概念

流处理从事件产生，到source，再到operator，中间是有一个过程和时间的，虽然大部分情况下，流到operator的数据都是按照事件产生的时间顺序来的，但是也不排除由于网络等原因，导致乱序的产生，所谓**乱序**，就是指Flink接收到的事件的先后顺序不是严格按照事件的Event Time顺序排列的。



因为有可能乱序，如果只根据eventTime决定窗口的运行，就不能明确数据是否全部到位，但又不能无限期的等待，此时必须要有一种机制来保证一个特定的时间后，必须触发window function进行计算，这个机制就是Watermark。

Watermark是Flink中测量事件时间进度的机制。Watermark作为数据流的一部分流动，并带有时间戳t。数据流中的Watermark用于表示时间戳小于Watermark的数据，都已经到达了。因此流中不应该再有时间戳t'<=watermark的元素。因此只有水位线越过对应窗口的结束时间，窗口才会关闭和进行计算



```
@PublicEvolving
public final class Watermark extends StreamElement {

    /** The watermark that signifies end-of-event-time. */
    public static final Watermark MAX_WATERMARK = new Watermark(Long.MAX_VALUE);

    // ------------------------------------------------------------------------

    /** The timestamp of the watermark in milliseconds. */
    private final long timestamp;

    /**
     * Creates a new watermark with the given timestamp in milliseconds.
     */
    public Watermark(long timestamp) { this.timestamp = timestamp; }

    /**
```

Flink接收到每一条数据时，都会产生一条Watermark，这条Watermark就等于当前所有到达数据中的maxEventTime - 允许延迟的时间

```
watermark=maxEventTime-maxAllowedLateness
```

总结watermark

- watermark是一个衡量事件时间进度的机制:水位线就是时间戳
- watermark是解决乱序问题的重要依据
- 在窗口计算中，watermark没过了窗口的结束时间就触发窗口计算（属于该窗口的元素都已经到达）；被认为早于这个时间的数据都已经到达
- watermark的计算应该是：最大事件时间-最大允许迟到时间

watermark计算

Flink中常用watermark计算方式有两种

- **With Periodic Watermarks**--定期提取水位线：比如每间隔1秒计算出来一个水位线
- **With Punctuated Watermarks**--每一个event计算一个水位线（不常用）

1. With Periodic Watermarks

```
import java.text.SimpleDateFormat

import org.apache.flink.streaming.api.functions.AssignerWithPeriodicWatermarks
import org.apache.flink.streaming.api.watermark.Watermark

/***
  * 这种方式计算水位线是常用的方式
  */
class MyAssignerWithPeriodicWatermarks extends
AssignerWithPeriodicWatermarks[(String,Long)]{

  //最大事件时间
  var maxEventTime:Long=_

  //最大允许迟到时间
  var maxAllowedLateness:Long=2000

  //用来计算水位线；由系统自动调用（每间隔一定时间调用一次）--以固定的频率调用
  override def getCurrentWatermark: Watermark = {
    new Watermark(maxEventTime-maxAllowedLateness)
  }

  /**
    * 有一个元素就会执行一次
    * @param element 流过来的元素
    * @param previousElementTimestamp
    * @return
    */
  override def extractTimestamp(element: (String, Long), previousElementTimestamp: Long):
Long = {
    //时间格式化器

    val format = new SimpleDateFormat("HH:mm:ss")
    //计算最大的事件时间
    maxEventTime=Math.max(maxEventTime,element._2)

    println("当前元素："+(element._1,format.format(element._2))+",水位
线："+format.format(maxEventTime-maxAllowedLateness))

    //方法返回值是当前元素的时间戳
    element._2
  }
}
```

```
import java.text.SimpleDateFormat

import org.apache.flink.streaming.api.scala.function.AllWindowFunction
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.util.Collector

class MyAllWindowFunctionForEventTime extends
AllWindowFunction[(String,Long),String,TimeWindow]{
  override def apply(window: TimeWindow, input: Iterable[(String, Long)], out:
Collector[String]): Unit = {
    val start: Long = window.getStart
    val end: Long = window.getEnd

    //把窗口的开始时间和结束时间打印一下
    //note:窗口的边界规则：前开后闭（前包含后不包含）
    val format = new SimpleDateFormat("HH:mm:ss")
    println("窗口时间范围：["+format.format(start)+","+format.format(end)+")")

    //把处理之后的元素发送到下游
    val elements: String = input.map(word=>word._1+":"+format.format(word._2)).mkString(" |
")
    out.collect(elements)

  }
}
```

```
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.{TumblingEventTimeWindows,
TumblingProcessingTimeWindows}
import org.apache.flink.streaming.api.windowing.time.Time

object AssignerWithPeriodicWatermarksDemo {
  def main(args: Array[String]): Unit = {
    val environment: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment

    //能够保证测试有效果，就应该把并行度设置为1

    environment.setParallelism(1)
    //因为flink在做流计算时默认使用的是processingTime。如果要使用eventTime，就需要显示声明
    environment.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    //每1秒钟调用一个水位线生成函数
    environment.getConfig.setAutoWatermarkInterval(1000)

    //要求输入的数据是   数据 时间戳，比如： a 1590647654000
    val dataStream: DataStream[String] =
environment.socketTextStream("flink.baizhiedu.com",9999)

    val result: DataStream[String] = dataStream.map(line => line.split("\\s+"))
      .map(word => (word(0), word(1).toLong))
      //指定水位线生成策略---》另外的一句话讲，以何种方式生成水位线
      .assignTimestampsAndWatermarks(new MyAssignerWithPeriodicWatermarks)
      .windowAll(TumblingEventTimeWindows.of(Time.seconds(2)))
      .apply(new MyAllWindowFunctionForEventTime)


    result.print()

    environment.execute("AssignerWithPeriodicWatermarksDemoJob")

  }

}
```

2. With Punctuated Watermarks(不推荐使用)--不说

  每一个事件都会生成一个水位线。在生产环境中，过多的生成水位线会影响程序的性能

```
import java.text.SimpleDateFormat

import org.apache.flink.streaming.api.functions.AssignerWithPunctuatedWatermarks
import org.apache.flink.streaming.api.watermark.Watermark

class MyAssignerWithPunctuatedWatermarks extends
AssignerWithPunctuatedWatermarks[(String,Long)]{

  //最大事件时间
  var maxEventTime:Long=Long.MinValue

  //最大允许迟到时间
  var maxAllowedLateness:Long=2000
  /**
    * 每过来一个元素。执行完extractTimestamp方法再执行这个方法
    * 计算水位线的方法
    * @param lastElement 当前元素
    * @param extractedTimestamp
 * @return

    */
  override def checkAndGetNextWatermark(lastElement: (String, Long), extractedTimestamp:
Long): Watermark = {

    maxEventTime=Math.max(maxEventTime,lastElement._2)

    //时间格式化器
    val format = new SimpleDateFormat("HH:mm:ss")
    println("当前元素："+(lastElement._1,format.format(lastElement._2))+",水位
线："+format.format(maxEventTime-maxAllowedLateness))

    new Watermark(maxEventTime-maxAllowedLateness)
  }

  /**
    *  每过来一个元素。都会先执行这个方法
    * 返回当前元素的时间戳
    * @param element 当前元素
    * @param previousElementTimestamp
    * @return
    */
  override def extractTimestamp(element: (String, Long), previousElementTimestamp: Long):
Long = {
    element._2
  }
}
```

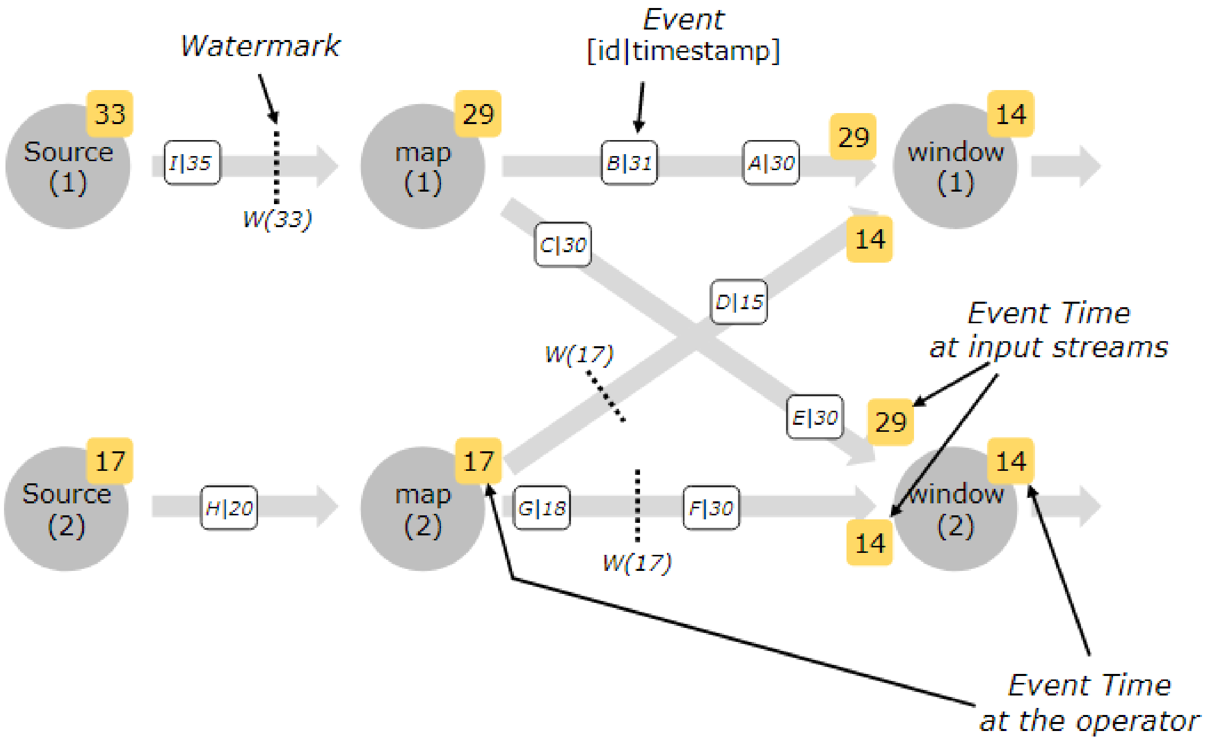
```
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time

object AssignerWithPunctuatedWatermarksDemo {
  def main(args: Array[String]): Unit = {
    val environment: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment

    //能够保证测试有效果，就应该把并行度设置为1
    environment.setParallelism(1)

    //因为flink在做流计算时默认使用的是processingTime。如果要使用eventTime，就需要显示声明
    environment.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    //要求输入的数据是   数据 时间戳，比如： a 1590647654000
    val dataStream: DataStream[String] =
environment.socketTextStream("flink.baizhiedu.com",9999)

    val result: DataStream[String] = dataStream.map(line => line.split("\\s+"))
      .map(word => (word(0), word(1).toLong))
      //指定水位线生成策略---》另外的一句话讲，以何种方式生成水位线
      .assignTimestampsAndWatermarks(new MyAssignerWithPunctuatedWatermarks)
      .windowAll(TumblingEventTimeWindows.of(Time.seconds(2)))

      .apply(new MyAllWindowFunctionForEventTime)
    result.print()

    environment.execute("AssignerWithPeriodicWatermarksDemoJob")

  }

}
```

Note

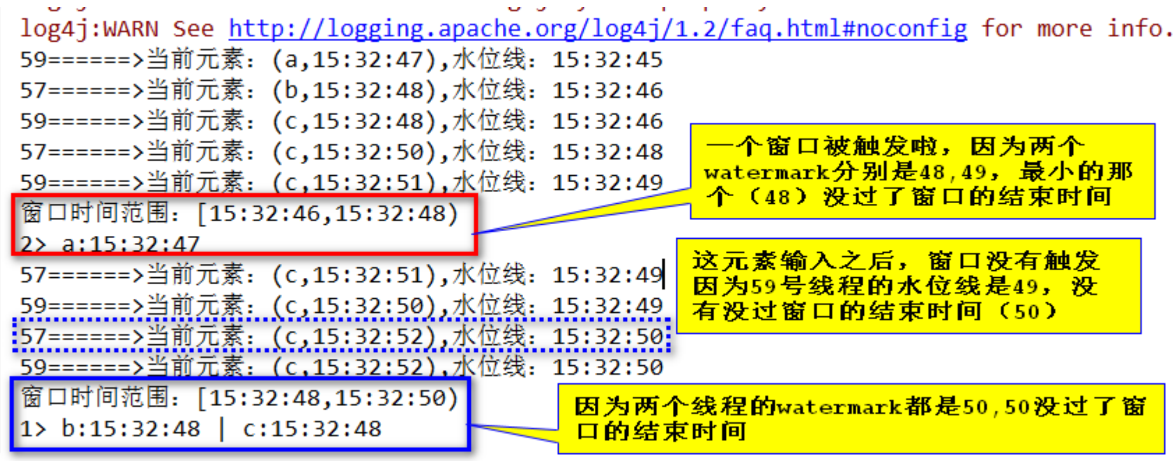
在并行情况下

- 算子的eventTime以最小的输入流eventTime为准

  比如：window[2]算子在继续往下游走的时候，使用的eventTime是14而不是29



- 当流中存在多个watermark时，以最小值为watermark



```
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
59======>当前元素：(a,15:32:47),水位线：15:32:45
57======>当前元素：(b,15:32:48),水位线：15:32:46
59======>当前元素：(c,15:32:48),水位线：15:32:46
57======>当前元素：(c,15:32:50),水位线：15:32:48
59======>当前元素：(c,15:32:51),水位线：15:32:49
窗口时间范围：[15:32:46,15:32:48)
2> a:15:32:47
57======>当前元素：(c,15:32:51),水位线：15:32:49
59======>当前元素：(c,15:32:50),水位线：15:32:49
57======>当前元素：(c,15:32:52),水位线：15:32:50
59======>当前元素：(c,15:32:52),水位线：15:32:50
窗口时间范围：[15:32:48,15:32:50)
1> b:15:32:48 | c:15:32:48
```

一个窗口被触发啦，因为两个watermark分别是48,49，最小的那个（48）没过了窗口的结束时间

这元素输入之后，窗口没有触发因为59号线程的水位线是49，没有没过窗口的结束时间（50）

因为两个线程的watermark都是50,50没过了窗口的结束时间

迟到数据

在flink中对于迟到数据进行了三种处理

- 默认处理方式：直接丢弃（spark就是采用这个方式）
- 在允许迟到的范围内，会重新开启窗口进行重新计算
- 超出了范围的数据tooLate，可以采用边输出的方式呈现处理方便后续的处理

在Flink中，水位线一旦没过窗口的EndTime，如果还有数据落入到此窗口，这些数据被定义为迟到数据。默认情况下，迟到数据将被删除。但是，Flink允许为窗口操作符指定允许的最大延迟，在允许的延迟范围内到达的元素仍然会添加到窗口中。根据使用的触发器，延迟但未丢弃的元素可能会导致窗口再次触发。

- 如果Watermarker时间t < 窗口EndTime t'' +允许迟到时间 t' ， 则该数据还可以参与窗口计算。

```
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time

object AssignerWithPunctuatedWatermarksForLatenessDemo {
  def main(args: Array[String]): Unit = {
    val environment: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment

    //能够保证测试有效果，就应该把并行度设置为1
    environment.setParallelism(1)

    //因为flink在做流计算时默认使用的是processingTime。如果要使用eventTime，就需要显示声明
    environment.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    //要求输入的数据是   数据 时间戳，比如：  a 1590647654000
    val dataStream: DataStream[String] =
environment.socketTextStream("flink.baizhiedu.com",9999)

    val result: DataStream[String] = dataStream.map(line => line.split("\\s+"))
      .map(word => (word(0), word(1).toLong))
      //指定水位线生成策略---》另外的一句话讲，以何种方式生成水位线
      .assignTimestampsAndWatermarks(new MyAssignerWithPunctuatedWatermarks)
      .windowAll(TumblingEventTimeWindows.of(Time.seconds(2)))
      ///==============================新添加的内容在这里===========================
      .allowedLateness(Time.seconds(2))
      .apply(new MyAllWindowFunctionForEventTime)


    result.print()

    environment.execute("AssignerWithPeriodicWatermarksDemoJob")

  }

}
```

- 如果Watermarker时间t >= 窗口EndTime t'' + 允许迟到时间t' 则该数据会被丢弃。为了能够更直观的呈现这些too late数据，可以通过side out输出

```
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time

object AssignerWithPunctuatedWatermarksForTooLateDemo {
  def main(args: Array[String]): Unit = {
    val environment: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment

    //能够保证测试有效果，就应该把并行度设置为1
    environment.setParallelism(1)

    //因为flink在做流计算时默认使用的是processingTime。如果要使用eventTime，就需要显示声明
    environment.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    //要求输入的数据是    数据 时间戳，比如：  a 1590647654000
    val dataStream: DataStream[String] =
environment.socketTextStream("flink.baizhiedu.com",9999)

    //要把too late数据通过边输出显示，就需要outputTag
    var outputTag:OutputTag[(String,Long)]=new OutputTag[(String,Long)]("too late")

    val result: DataStream[String] = dataStream.map(line => line.split("\\s+"))
      .map(word => (word(0), word(1).toLong))
      //指定水位线生成策略---》另外的一句话讲，以何种方式生成水位线
      .assignTimestampsAndWatermarks(new MyAssignerWithPunctuatedWatermarks)
      .windowAll(TumblingEventTimeWindows.of(Time.seconds(2)))
      .allowedLateness(Time.seconds(2))
      ///===============================新添加的内容在这里===========================
      //迟到时间>=允许允许迟到时间的数据，通过边输出的方式
      .sideOutputLateData(outputTag)
      .apply(new MyAllWindowFunctionForEventTime)


    result.print("正常")
    result.getSideOutput(outputTag).printToErr("太迟的数据")

    environment.execute("AssignerWithPeriodicWatermarksDemoJob")

  }

}
```