> 人生没有彩排，每天都是现场直播

Flink之ProcessFunctionAPI

> 之前学习的**转换算子**是无法访问**事件的时间戳信息和水位线信息的**。而这在一些应用场景下，极为重要。例如MapFunction这样的map转换算子就无法访问时间戳或者当前事件的事件时间。

基于此，DataStream API提供了一系列的Low-Level转换算子。可以**访问时间戳**、watermark**以及注册定时事件**。还可以输出**特定的一些事件**，例如超时事件等。Process Function用来构建事件驱动的应用以及实现自定义的业务逻辑(使用之前的window函数和转换算子无法实现)。例如，Flink SQL就是使用Process Function实现的。

Flink提供了8个Process Function：

- ProcessFunction
- KeyedProcessFunction
- CoProcessFunction
- ProcessJoinFunction
- BroadcastProcessFunction
- KeyedBroadcastProcessFunction
- ProcessWindowFunction
- ProcessAllWindowFunction

KeyedProcessFunction【重点掌握】

> 这里我们重点介绍KeyedProcessFunction。KeyedProcessFunction用来操作KeyedStream。KeyedProcessFunction会**处理流的每一个元素**，输出为0个、1个或者多个元素。所有的Process Function都继承自RichFunction接口，所以都有open()、close()和getRuntimeContext()等方法。

而KeyedProcessFunction[KEY, IN, OUT]还额外提供了两个方法:

- processElement(v: IN, ctx: Context, out: Collector[OUT])

  > 流中的每一个元素都会调用这个方法，调用结果将会放在Collector数据类型中输出。**Context**可以访问元素的时间戳，元素的key，以及**TimerService**时间服务。**Context**还可以将结果输出到别的流(side outputs)。

- onTimer(timestamp: Long, ctx: OnTimerContext, out: Collector[OUT])

  > 它是一个回调函数。当之前注册的定时器触发时调用。参数timestamp为定时器所设定的触发的时间戳。Collector为输出结果的集合。OnTimerContext和processElement的Context参数一样，提供了上下文的一些信息，例如**定时器触发的时间信息**(事件时间或者处理时间)。

TimerService 和 定时器（Timers）

Context和OnTimerContext所持有的TimerService对象拥有以下方法:

- currentProcessingTime(): Long 返回当前处理时间

- currentWatermark(): Long 返回当前watermark的时间戳
- registerProcessingTimeTimer(timestamp: Long): Unit 会注册当前key的processing time的定时器。当processing time到达定时时间时，触发timer。
- registerEventTimeTimer(timestamp: Long): Unit 会注册当前key的event time 定时器。当水位线大于等于定时器注册的时间时，触发定时器执行回调函数。
- deleteProcessingTimeTimer(timestamp: Long): Unit 删除之前注册处理时间定时器。如果没有这个时间戳的定时器，则不执行。
- deleteEventTimeTimer(timestamp: Long): Unit 删除之前注册的事件时间定时器，如果没有此时间戳的定时器，则不执行。

当定时器timer触发时，会执行回调函数onTimer()。注意定时器timer只能在keyed streams上面使用。

下面举个例子说明KeyedProcessFunction如何操作KeyedStream。

需求：监控温度传感器的温度值，如果温度值在一秒钟之内(processing time)连续上升，则报警。

```
val warnings = readings
.keyBy(_.id)
.process(new TempIncreaseAlertFunction)

// 开始实现TempIncreaseAlertFunction
class TempIncreaseAlertFunction extends KeyedProcessFunction[String,
SensorReading, String] {
  // 保存上一个传感器温度值
  lazy val lastTemp: ValueState[Double] = getRuntimeContext.getState(
    new ValueStateDescriptor[Double]("lastTemp", Types.of[Double])
  )

  // 保存注册的定时器的时间戳
  lazy val currentTimer: ValueState[Long] = getRuntimeContext.getState(
    new ValueStateDescriptor[Long]("timer", Types.of[Long])
  )

  override def processElement(r: SensorReading,
                              ctx: KeyedProcessFunction[String, SensorReading,
String]#Context,
                              out: Collector[String]): Unit = {
    // 取出上一次的温度
    val prevTemp = lastTemp.value()
    // 将当前温度更新到上一次的温度这个变量中
    lastTemp.update(r.temperature)

    val curTimerTimestamp = currentTimer.value()
    if (prevTemp == 0.0 || r.temperature < prevTemp) {
      // 温度下降或者是第一个温度值，删除定时器
      ctx.timerService().deleteProcessingTimeTimer(curTimerTimestamp)
      // 清空状态变量
      currentTimer.clear()
    } else if (r.temperature > prevTemp && curTimerTimestamp == 0) {
      // 温度上升且我们并没有设置定时器
      val timerTs = ctx.timerService().currentProcessingTime() + 1000
      ctx.timerService().registerProcessingTimeTimer(timerTs)

      currentTimer.update(timerTs)
```

```
    }
  }

  override def onTimer(ts: Long,
                       ctx: KeyedProcessFunction[String, SensorReading,
String]#OnTimerContext,
                       out: Collector[String]): Unit = {
    out.collect("传感器id为: " + ctx.getCurrentKey + "的传感器温度值已经连续1s上升
了。")
    currentTimer.clear()
  }
}
```

> 现在有没有找到状态编程的感觉了

侧输出流（SideOutput）

> 大部分的DataStream API的算子的输出是单一输出，也就是某种数据类型的流。除了split算子，可以将一条流分成多条流，这些流的数据类型也都相同。process function的side outputs功能可以产生多条流，并且这些流的数据类型可以不一样。**一个side output可以定义为OutputTag[X]对象，X是输出流的数据类型。**process function可以通过Context对象发射一个事件到一个或者多个side outputs。

下面是一个示例程序:

```
val monitoredReadings: DataStream[SensorReading] = readings
  .process(new FreezingMonitor)

monitoredReadings
  .getSideOutput(new OutputTag[String]("freezing-alarms"))
  .print()

readings.print()
```

接下来我们实现FreezingMonitor函数，用来监控传感器温度值，将温度值低于32F的温度输出到side output

```
class FreezingMonitor extends ProcessFunction[SensorReading, SensorReading] {
  // 定义一个侧输出标签
  lazy val freezingAlarmOutput: OutputTag[String] =
    new OutputTag[String]("freezing-alarms")

  override def processElement(r: SensorReading,
                              ctx: ProcessFunction[SensorReading,
SensorReading]#Context,
                              out: Collector[SensorReading]): Unit = {
    // 温度在32F以下时，输出警告信息
    if (r.temperature < 32.0) {
      ctx.output(freezingAlarmOutput, s"Freezing Alarm for ${r.id}")
    }
    // 所有数据直接常规输出到主流
    out.collect(r)
  }
}
```

CoProcessFunction

对于两条输入流，DataStream API提供了CoProcessFunction这样的low-level操作。CoProcessFunction提供了操作每一个输入流的方法: processElement1()和processElement2()。

类似于ProcessFunction，这两种方法都通过Context对象来调用。这个Context对象可以访问事件数据，定时器时间戳，TimerService，以及side outputs。CoProcessFunction也提供了onTimer()回调函数。