Flink 四大基石之时间和水位线原理介绍！

本文作者：在 IT 中穿梭旅行

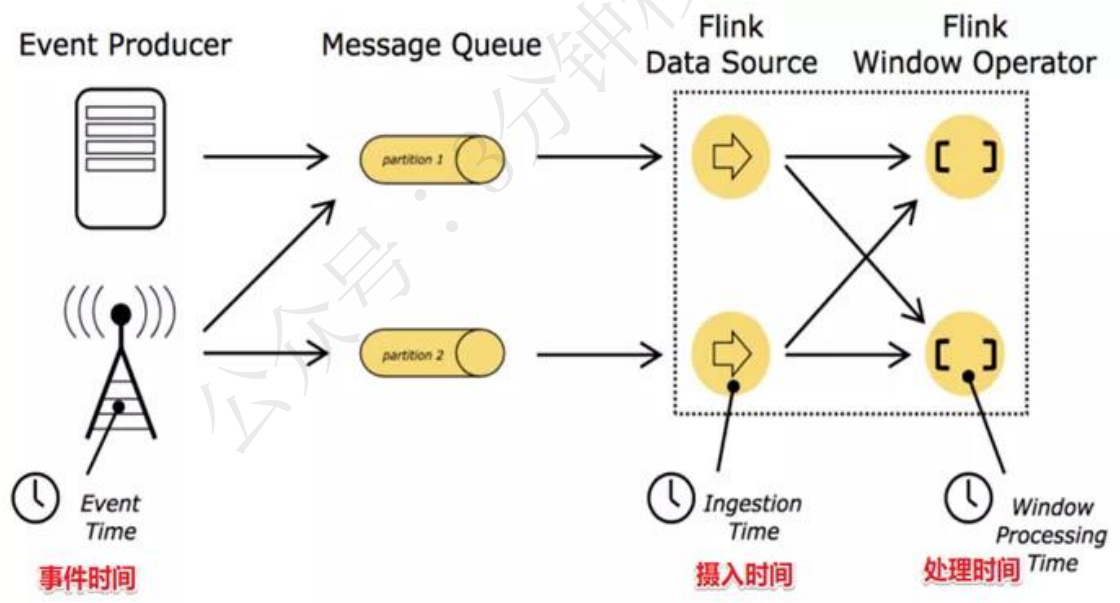
大家伙，我是土哥。

在 Flink 中，涉及到时间和水位线这一概念，时间是 Flink 中的四大基石（Checkpoint、State、Time、Window）之一，是实现流批统一的一个重要特性。本文讲解内容包含以下六部分：

1. Flink 时间分类
2. WaterMark 引入
3. 水印运行原理
4. 水印 API 调用
5. 侧道输出保障超过 WaterMark 数据不丢失
6. WaterMark+EventTimeWindow+Allowed Lateness 案例

1.Flink 时间分类

在 Flink 的流式处理中，会涉及到时间的不同概念，如下图所示：



- EventTime[事件时间]

事件发生的时间，例如：点击网站上的某个链接的时间，每一条日志都会记录自己的生成时间

如果以 EventTime 为基准来定义时间窗口那将形成 EventTimeWindow,要求消息本身就应该携带 EventTime

- IngestionTime[摄入时间]

数据进入 Flink 的时间，如某个 Flink 节点的 sourceoperator 接收到数据的时间，例如：某个 source 消费到 kafka 中的数据

如果以 IngesingtTime 为基准来定义时间窗口那将形成 IngestingTimeWindow,以 source 的 systemTime 为准

- ProcessingTime[处理时间]

某个 Flink 节点执行某个 operation 的时间，例如：timeWindow 处理数据时的系统时间，默认的时间属性就是 Processing Time

如果以 ProcessingTime 基准来定义时间窗口那将形成 ProcessingTimeWindow，以 operator 的 systemTime 为准

在 Flink 的流式处理中，**绝大部分的业务都会使用 EventTime**，一般只在 EventTime 无法使用时，才会被迫使用 ProcessingTime 或者 IngestionTime。

如果要使用 EventTime，那么需要引入 EventTime 的时间属性，引入方式如下所示：

三种时间设置代码如下：

```
final StreamExecutionEnvironment env
    = StreamExecutionEnvironment.getExecutionEnvironrnent();
// 使用处理时间
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime) ;
// 使用摄入时间

env.setStrearnTimeCharacteristic(TimeCharacteristic.IngestionTime);

// 使用事件时间
env.setStrearnTimeCharacteristic(TimeCharacteri stic Eve~tTime);
```

2 WaterMark(水印)

通过两个案例来了解什么是水印，主要解决什么问题？

案例 1： 假你正在去往地下停车场的路上，并且打算用手机点一份外卖。

选好了外卖后，你就用在线支付功能付款了，这个时候是 11 点 50 分。恰好这时，你走进了地下停车库，而这里并没有手机信号。因此外卖的在线支付并没有立刻成功，而支付系统一直在 Retry 重试"支付"这个操作。

当你找到自己的车并且开出地下停车场的时候，已经是 12 点 05 分了。这个时候手机重新有了信号，手机上的支付数据成功发到了外卖在线支付系统，支付完成。

在上面这个场景中你可以看到，支付数据的事件时间是 11 点 50 分，而支付数据的处理时间是 12 点 05 分

案例 2： 如上图所示，某 App 会记录用户的所有点击行为，并回传日志（在网络不好的情况下，先保存在本地，延后回传）。 A 用户在 11:02 对 App 进行操作，B 用户在 11:03 操作了 App，

但是 A 用户的网络不太稳定，回传日志延迟了，导致我们在服务端先接受到 B 用户 11:03 的消息，然后再接受到 A 用户 11:02 的消息，消息乱序了。

通过上面的例子,我们知道,在进行数据处理的时候应该按照事件时间进行处理,也就是窗口应该要考虑到事件时间，但是窗口不能无限的一直等到延迟数据的到来,需要有一个触发窗口计算的机制，就是 watermaker 水位线/水印机制。 所以：水印是用来解决数据延迟、数据乱序等问题，总结如下图所示：

图片

水印就是一个时间戳（timestamp），Flink 可以给数据流添加水印

- 水印并不会影响原有 Eventtime 事件时间
- 当数据流添加水印后，会按照水印时间来触发窗口计算,也就是说 watermark 水印是用来触发窗口计算的
- 设置水印时间，会比事件时间小几秒钟,表示最大允许数据延迟达到多久
- 水印时间 = 事件时间 - 允许延迟时间 (例如：10:09:57 = 10:10:00 - 3s)

3 水印运行原理

如下图所示： 图片

窗口是 10 分钟触发一次，现在在 12:00 - 12:10 有一个窗口，本来有一条数据是在 12:00 - 12:10 这个窗口被计算，但因为延迟，12:12 到达，这时 12:00 - 12:10 这个窗口就会被关闭，只能将数据下发到下一个窗口进行计算，这样就产生了数据延迟，造成计算不准确。

现在添加一个水位线：数据时间戳为 2 分钟。这时用数据产生的事件时间 12:12 - 允许延迟的水印 2 分钟 = 12:10 >= 窗口结束时间 。窗口触发计算，该数据就会被计算到这个窗口里

4 水印 API 调用

在 DataStream API 中使用 TimestampAssigner 接口定义时间戳的提取行为，包含两个子接口 AssignerWithPeriodicWatermarks 接口和 AssignerWithPunctuatedWaterMarks 接口

- 定义 抽取时间戳，以及生成 watermark 的方法，有两种类型
  - AssignerWithPeriodicWatermarks
    - 周期性的生成 watermark：系统会周期性的将 watermark 插入到流中
    - 默认周期是200毫秒，可以使用 ExecutionConfig.setAutoWatermarkInterval() 设置
    - BoundedOutOfOrderness 是基于周期性 watermark 的。
  - AssignerWithPunctuatedWatermarks
    - 没有时间周期规律，可打断的生成 watermark

定期生成	根据特殊记录生成
现实时间驱动	数据驱动
每隔一段时间调用生成方法	每一次分配Timestamp都会调用生成方法
实现AssignerWithPeriodicWatermarks	实现AssignerWithPunctuatedWatermarks

5 侧道输出保证超过 WaterMark 数据不丢失

使用 WaterMark+ EventTimeWindow 机制可以在一定程度上解决数据乱序的问题，但是，WaterMark 水位线也不是万能的，在某些情况下，数据延迟会非常严重，即使通过 Watermark + EventTimeWindow 也无法等到数据全部进入窗口再进行处

理，因为窗口触发计算后，对于延迟到达的本属于该窗口的数据，Flink 默认会将这些延迟严重的数据进行丢弃

那么如果想要让一定时间范围的延迟数据不会被丢弃，可以使用 Allowed Lateness(允许迟到机制/侧道输出机制)设定一个允许延迟的时间和侧道输出对象来解决

即使用 WaterMark + EventTimeWindow + Allowed Lateness 方案（包含侧道输出），可以做到数据不丢失。

API 调用

1. allowedLateness(lateness:Time)---设置允许延迟的时间

该方法传入一个 Time 值，设置允许数据迟到的时间，这个时间和 watermark 中的时间概念不同。再来回顾一下，

watermark=数据的事件时间-允许乱序时间值

随着新数据的到来，watermark 的值会更新为最新数据事件时间-允许乱序时间值，但是如果这时候来了一条历史数据，watermark 值则不会更新。

总的来说，watermark 永远不会倒退它是为了能接收到尽可能多的乱序数据。

那这里的 Time 值呢？主要是为了等待迟到的数据，如果属于该窗口的数据到来，仍会进行计算，后面会对计算方式仔细说明

注意：该方法只针对于基于 event-time 的窗口

1. sideOutputLateData(outputTag:OutputTag[T])--保存延迟数据 该方法是将迟来的数据保存至给定的 outputTag 参数，而 OutputTag 则是用来标记延迟数据的一个对象。

2. DataStream.getSideOutput(tag:OutputTag[X])--获取延迟数据 通过 window 等操作返回的 DataStream 调用该方法，传入标记延迟数据的对象来获取延迟的数据

6 WaterMark+EventTimeWindow+Allowed Lateness 案例

```
package cn.itcast.watermark

import org.apache.flink.api.scala._
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.functions.AssignerWithPeriodicWat
ermarks
import org.apache.flink.streaming.api.scala.function.WindowFunction
import org.apache.flink.streaming.api.scala.{DataStream, OutputTag, Str
eamExecutionEnvironment, WindowedStream}
```

```
import org.apache.flink.streaming.api.watermark.Watermark
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.util.Collector

/*
演示水印如何解决数据乱序问题
 */
object SlideOutputDemo {
  //样例类 CarWc(信号等id,数量)
  case class CarWc(id: String, num: Int, ts: Long)
  def main(args: Array[String]): Unit = {
    // 1 获取运行环境
    val env: StreamExecutionEnvironment = StreamExecutionEnvironment.ge
tExecutionEnvironment
    //方便观察数据设置并行度为1
    env.setParallelism(1)
    //设置基于事件时间进行计算
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    //2 接收数据
    val socketDs = env.socketTextStream("hlink161", 9999)
    // 3 transformation 接收到数据之后按照逗号切分，拿到红绿灯id,数量（通过
汽车数量）
    val carWcDs: DataStream[CarWc] = socketDs.map(
      line => {
        val arr: Array[String] = line.split(",")
        // println("====="+line)
        //3.1 封装样例类
        CarWc(arr(0), arr(1).toInt, arr(2).toLong)
      }
    )
    // 3.1 设置水印机制,我们自己动手实现
    val waterMarkDs: DataStream[CarWc] = carWcDs.assignTimestampsAndWat
ermarks(new AssignerWithPeriodicWatermarks[CarWc]{
      /** The current maximum timestamp seen so far. */
      private var currentMaxTimestamp = 0L
      //定义最大允许的延迟时间
      private var maxOutOfOrderness=0

      /** The timestamp of the last emitted watermark. */ //巨大的负数
      private var lastEmittedWatermark:Long= Long.MinValue
      //3.1.1 计算并生成水印
      override def getCurrentWatermark: Watermark = {
        // this guarantees that the watermark never goes backwards.
        //计算出的watermark 时间
        val potentialWM = currentMaxTimestamp - maxOutOfOrderness
        //保证水印时间不会回退！！
        if (potentialWM >= lastEmittedWatermark)
```

```
          lastEmittedWatermark = potentialWM
        return new Watermark(lastEmittedWatermark)
      }
      //3.1.2 获取到eventtime 字段
      override def extractTimestamp(element: CarWc, previousElementTime
stamp: Long): Long = {
        //获取日志的eventtime
        val timestamp = element.ts
        //判断比较 eventtime 是不是大于最大的eventtime 时间
        if (timestamp > currentMaxTimestamp)
          currentMaxTimestamp = timestamp
        //返回日志的eventtime 时间
        timestamp
      }
    } )

    // pre-4 定义一个侧输出流
    val outputTag: OutputTag[CarWc] = new OutputTag[CarWc]("delayCarWc")
    // 4 设置窗口

    val windowStream: WindowedStream[CarWc, String, TimeWindow] = water
MarkDs.keyBy(_.id).timeWindow(Time.seconds(5))
      .allowedLateness(Time.seconds(5))
      .sideOutputLateData(outputTag)


    //5 使用apply 方法执行聚合计算
    val windowRes: DataStream[CarWc] = windowStream.apply(new WindowFun
ction[CarWc, CarWc, String, TimeWindow] {
      override def apply(key: String, window: TimeWindow, input: Iterab
le[CarWc], out: Collector[CarWc]): Unit = {
        val iter: Iterator[CarWc] = input.iterator
        println("窗口开始时间 》》"+window.getStart+"== 窗口结束时间》》"+
window.getEnd+"数据 【"+input.iterator.mkString(";")+"】")
        val wc: CarWc = iter.reduce((t1, t2) =>
          CarWc(t1.id, t1.num + t2.num, t1.ts)
        )
        out.collect(wc)
      }
    })
    windowRes.print("窗口计算结果>>")
    //获取到侧输出流结果
    val outputDs: DataStream[CarWc] = windowRes.getSideOutput(outputTag)
    outputDs.print("侧输出流数据》》")
    env.execute()
  }
}
```

以上就是 Flink Time 的讲解内容！觉得好的，点赞，在看，分享三连击，谢谢！！！

最近整理了一份计算机类的书籍，包含 python、java、大数据、人工智能、算法等，种类特别齐全。

获取方式：关注公众号：**3 分钟秒懂大数据**，回复：**福利**，就可以获得这份超级大礼！