flink窗口、时间和水印

主要介绍 Flink 中的时间和水印。
我们在之前的课时中反复提到过窗口和时间的概念，Flink 框架中支持事件时间、摄入时间和处理时间三种。而当我们在流式计算环境中数据从 Source 产生，再到转换和输出，这个过程由于网络和反压的原因会导致消息乱序。因此，需要有一个机制来解决这个问题，这个特别的机制就是“水印”。
Flink 的窗口和时间
我们在第 05 课时中讲解过 Flink 窗口的实现，根据窗口数据划分的不同，目前 Flink 支持如下 3 种：
滚动窗口，窗口数据有固定的大小，窗口中的数据不会叠加；
滑动窗口，窗口数据有固定的大小，并且有生成间隔；
会话窗口，窗口数据没有固定的大小，根据用户传入的参数进行划分，窗口数据无叠加。
Flink 中的时间分为三种：
事件时间（Event Time），即事件实际发生的时间；
摄入时间（Ingestion Time），事件进入流处理框架的时间；
处理时间（Processing Time），事件被处理的时间。
下面的图详细说明了这三种时间的区别和联系：

事件时间（Event Time）
事件时间（Event Time）指的是数据产生的时间，这个时间一般由数据生产方自身携带，比如 Kafka 消息，每个生成的消息中自带一个时间戳代表每条数据的产生时间。Event Time 从消息的产生就诞生了，不会改变，也是我们使用最频繁的时间。
利用 Event Time 需要指定如何生成事件时间的“水印”，并且一般和窗口配合使用，具体会在下面的“水印”内容中详细讲解。
我们可以在代码中指定 Flink 系统使用的时间类型为 EventTime：
复制final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
//设置时间属性为 EventTime
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
DataStream<MyEvent> stream = env.addSource(new FlinkKafkaConsumer09<MyEvent>(topic, schema, props));
stream
    .keyBy((event) -> event.getUser())
    .timeWindow(Time.hours(1))
    .reduce((a, b) -> a.add(b))
    .addSink(...);
Flink 注册 EventTime 是通过 InternalTimerServiceImpl.registerEventTimeTimer 来实现的：

可以看到，该方法有两个入参：namespace 和 time，其中 time 是触发定时器的时间，namespace 则被构造成为一个 TimerHeapInternalTimer 对象，然后将其放入 KeyGroupedInternalPriorityQueue 队列中。
那么 Flink 什么时候会使用这些 timer 触发计算呢？答案在这个方法里：
复制InternalTimeServiceImpl.advanceWatermark。
public void advanceWatermark(long time) throws Exception {
  currentWatermark = time;
  InternalTimer<K, N> timer;
  while ((timer = eventTimeTimersQueue.peek()) != null && timer.getTimestamp() <= time) {
    eventTimeTimersQueue.poll();

```
    keyContext.setCurrentKey(timer.getKey());
    triggerTarget.onEventTime(timer);
  }
}
```

这个方法中的 while 循环部分会从 eventTimeTimersQueue 中依次取出触发时间小于参数 time 的所有定时器，调用 triggerTarget.onEventTime() 方法进行触发。

这就是 EventTime 从注册到触发的流程。

处理时间（Processing Time）

处理时间（Processing Time）指的是数据被 Flink 框架处理时机器的系统时间，Processing Time 是 Flink 的时间系统中最简单的概念，但是这个时间存在一定的不确定性，比如消息到达处理节点延迟等影响。

我们同样可以在代码中指定 Flink 系统使用的时间为 Processing Time：

复制final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);

同样，也可以在源码中找到 Flink 是如何注册和使用 Processing Time 的。

registerProcessingTimeTimer() 方法为我们展示了如何注册一个 ProcessingTime 定时器：

每当一个新的定时器被加入到 processingTimeTimersQueue 这个优先级队列中时，如果新来的 Timer 时间戳更小，那么更小的这个 Timer 会被重新注册 ScheduledThreadPoolExecutor 定时执行器上。

Processing Time 被触发是在 InternalTimeServiceImpl 的 onProcessingTime() 方法中：

一直循环获取时间小于入参 time 的所有定时器，并运行 triggerTarget 的 onProcessingTime() 方法。

摄入时间（Ingestion Time）

摄入时间（Ingestion Time）是事件进入 Flink 系统的时间，在 Flink 的 Source 中，每个事件会把当前时间作为时间戳，后续做窗口处理都会基于这个时间。理论上 Ingestion Time 处于 Event Time 和 Processing Time之间。

与事件时间相比，摄入时间无法处理延时和无序的情况，但是不需要明确执行如何生成 watermark。在系统内部，摄入时间采用更类似于事件时间的处理方式进行处理，但是有自动生成的时间戳和自动的 watermark。

可以防止 Flink 内部处理数据是发生乱序的情况，但无法解决数据到达 Flink 之前发生的乱序问题。如果需要处理此类问题，建议使用 EventTime。

Ingestion Time 的时间类型生成相关的代码在 AutomaticWatermarkContext 中：

我们可以看出，这里会设置一个 watermark 发送定时器，在 watermarkInterval 时间之后触发。

处理数据的代码在 processAndCollect() 方法中：

水印（WaterMark）

水印（WaterMark）是 Flink 框架中最晦涩难懂的概念之一，有很大一部分原因是因为翻译的原因。

WaterMark 在正常的英文翻译中是水位，但是在 Flink 框架中，翻译为“水位线”更为合理，它在本质上是一个时间戳。

在上面的时间类型中我们知道，Flink 中的时间：

EventTime 每条数据都携带时间戳；

ProcessingTime 数据不携带任何时间戳的信息；

IngestionTime 和 EventTime 类似，不同的是 Flink 会使用系统时间作为时间戳绑定到每条数据，可以防止 Flink 内部处理数据是发生乱序的情况，但无法解决数据到达 Flink 之前发生的乱序问题。

所以，我们在处理消息乱序的情况时，会用 EventTime 和 WaterMark 进行配合使用。

首先我们要明确几个基本问题。

水印的本质是什么

水印的出现是为了解决实时计算中的数据乱序问题，它的本质是 DataStream 中一个带有时间戳的元素。如果 Flink 系统中出现了一个 WaterMark T，那么就意味着 EventTime < T 的数据都已经到达，窗口的结束时间和 T 相同的那个窗口被触发进行计算了。

也就是说：水印是 Flink 判断迟到数据的标准，同时也是窗口触发的标记。
在程序并行度大于 1 的情况下，会有多个流产生水印和窗口，这时候 Flink 会选取时间戳最小的水印。
水印是如何生成的
Flink 提供了 assignTimestampsAndWatermarks() 方法来实现水印的提取和指定，该方法接受的入参有 AssignerWithPeriodicWatermarks 和 AssignerWithPunctuatedWatermarks 两种。
整体的类图如下：

水印种类
周期性水印
我们在使用 AssignerWithPeriodicWatermarks 周期生成水印时，周期默认的时间是 200ms，这个时间的指定位置为：
复制@PublicEvolving
public void setStreamTimeCharacteristic(TimeCharacteristic characteristic) {
    this.timeCharacteristic = Preconditions.checkNotNull(characteristic);
    if (characteristic == TimeCharacteristic.ProcessingTime) {
        getConfig().setAutoWatermarkInterval(0);
    } else {
        getConfig().setAutoWatermarkInterval(200);
    }
}
是否还记得上面我们在讲时间类型时会通过 env.setStreamTimeCharacteristic() 方法指定 Flink 系统的时间类型，这个 setStreamTimeCharacteristic() 方法中会做判断，如果用户传入的是 TimeCharacteristic.eventTime 类型，那么 AutoWatermarkInterval 的值则为 200ms ，如上述代码所示。当前我们也可以使用 ExecutionConfig.setAutoWatermarkInterval() 方法来指定自动生成的时间间隔。
在上述的类图中可以看出，我们需要通过 TimestampAssigner 的 extractTimestamp() 方法来提取 EventTime。
Flink 在这里提供了 3 种提取 EventTime() 的方法，分别是：
AscendingTimestampExtractor
BoundedOutOfOrdernessTimestampExtractor
IngestionTimeExtractor
这三种方法中 BoundedOutOfOrdernessTimestampExtractor() 用的最多，需特别注意，在这个方法中的 maxOutOfOrderness 参数，该参数指的是允许数据乱序的时间范围。简单说，这种方式允许数据迟到 maxOutOfOrderness 这么长的时间。
复制   public BoundedOutOfOrdernessTimestampExtractor(Time maxOutOfOrderness) {
        if (maxOutOfOrderness.toMilliseconds() < 0) {
            throw new RuntimeException("Tried to set the maximum allowed " +
                "lateness to " + maxOutOfOrderness + ". This parameter cannot be negative.");
        }
        this.maxOutOfOrderness = maxOutOfOrderness.toMilliseconds();
        this.currentMaxTimestamp = Long.MIN_VALUE + this.maxOutOfOrderness;
    }
    public abstract long extractTimestamp(T element);
    @Override
    public final Watermark getCurrentWatermark() {
        long potentialWM = currentMaxTimestamp - maxOutOfOrderness;
        if (potentialWM >= lastEmittedWatermark) {
            lastEmittedWatermark = potentialWM;
        }
        return new Watermark(lastEmittedWatermark);
    }

```
    @Override
    public final long extractTimestamp(T element, long previousElementTimestamp) {
        long timestamp = extractTimestamp(element);
        if (timestamp > currentMaxTimestamp) {
            currentMaxTimestamp = timestamp;
        }
        return timestamp;
    }
```

PunctuatedWatermark 水印

这种水印的生成方式 Flink 没有提供内置实现，它适用于根据接收到的消息判断是否需要产生水印的情况，用这种水印生成的方式并不多见。

举个简单的例子，假如我们发现接收到的数据 MyData 中以字符串 watermark 开头则产生一个水印：

复制

```
data.assignTimestampsAndWatermarks(new AssignerWithPunctuatedWatermarks<UserActionRecord>() {
      @Override
      public Watermark checkAndGetNextWatermark(MyData data, long l) {
        return data.getRecord.startsWith("watermark") ? new Watermark(l) : null;
      }
      @Override
      public long extractTimestamp(MyData data, long l) {
        return data.getTimestamp();
      }
    });

class MyData{
    private String record;
    private Long timestamp;
    public String getRecord() {
        return record;
    }
    public void setRecord(String record) {
        this.record = record;
    }
    public Timestamp getTimestamp() {
        return timestamp;
    }
    public void setTimestamp(Timestamp timestamp) {
        this.timestamp = timestamp;
    }
}
```

案例

我们上面讲解了 Flink 关于水印和时间的生成，以及使用，下面举一个例子来讲解。

模拟一个实时接收 Socket 的 DataStream 程序，代码中使用 AssignerWithPeriodicWatermarks 来设置水印，将接收到的数据进行转换，分组并且在一个 5

秒的窗口内获取该窗口中第二个元素最小的那条数据。

复制

```
public static void main(String[] args) throws Exception {
    StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironment();
    //设置为eventtime事件类型
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

```
    //设置水印生成时间间隔100ms
    env.getConfig().setAutoWatermarkInterval(100);
    DataStream<String> dataStream = env
            .socketTextStream("127.0.0.1", 9000)
            .assignTimestampsAndWatermarks(new AssignerWithPeriodicWatermarks<String>() {
                private Long currentTimeStamp = 0L;
                //设置允许乱序时间
                private Long maxOutOfOrderness = 5000L;
                @Override
                public Watermark getCurrentWatermark() {
                    return new Watermark(currentTimeStamp - maxOutOfOrderness);
                }
                @Override
                public long extractTimestamp(String s, long l) {
                    String[] arr = s.split(",");
                    long timeStamp = Long.parseLong(arr[1]);
                    currentTimeStamp = Math.max(timeStamp, currentTimeStamp);
                    System.err.println(s + ",EventTime:" + timeStamp + ",watermark:" + (currentTimeStamp -
maxOutOfOrderness));
                    return timeStamp;
                }
            });
    dataStream.map(new MapFunction<String, Tuple2<String, Long>>() {
        @Override
        public Tuple2<String, Long> map(String s) throws Exception {
            String[] split = s.split(",");
            return new Tuple2<String, Long>(split[0], Long.parseLong(split[1]));
        }
    })
            .keyBy(0)
            .window(TumblingEventTimeWindows.of(Time.seconds(5)))
            .minBy(1)
            .print();
    env.execute("WaterMark Test Demo");
}
```

我们第一次试验的数据如下：

```
复制flink,1588659181000
flink,1588659182000
flink,1588659183000
flink,1588659184000
flink,1588659185000
```

可以做一个简单的判断，第一条数据的时间戳为 1588659181000，窗口的大小为 5 秒，那么应该会在 flink,1588659185000 这条数据出现时触发窗口的计算。

我们用 nc -lk 9000 命令启动端口，然后输出上述试验数据，看到控制台的输出：

很明显，可以看到当第五条数据出现后，窗口触发了计算。

下面再模拟一下数据乱序的情况，假设我们的数据来源如下：

```
复制flink,1588659181000
flink,1588659182000
```

flink,1588659183000
flink,1588659184000
flink,1588659185000
flink,1588659180000
flink,1588659186000
flink,1588659187000
flink,1588659188000
flink,1588659189000
flink,1588659190000
其中的 flink,1588659180000 为乱序消息，来看看会发生什么?

可以看到，时间戳为 1588659180000 的这条消息并没有被处理，因为此时代码中的允许乱序时间 private Long maxOutOfOrderness = 0L 即不处理乱序消息。
下面修改 private Long maxOutOfOrderness = 5000L，即代表允许消息的乱序时间为 5 秒，然后把同样的数据发往 socket 端口。
可以看到，我们把所有数据发送出去仅触发了一次窗口计算，并且输出的结果中 watermark 的时间往后顺延了 5 秒钟。所以，maxOutOfOrderness 的设置会影响窗口的计算时间和水印的时间，如下图所示：

假如我们继续向 socket 中发送数据：
复制flink,1588659191000
flink,1588659192000
flink,1588659193000
flink,1588659194000
flink,1588659195000
可以看到下一次窗口的触发时间：

在这里要特别说明，Flink 在用时间 + 窗口 + 水印来解决实际生产中的数据乱序问题，有如下的触发条件：
watermark 时间 >= window_end_time；
在 [window_start_time,window_end_time) 中有数据存在，这个窗口是左闭右开的。
此外，因为 WaterMark 的生成是以对象的形式发送到下游，同样会消耗内存，因此水印的生成时间和频率都要进行严格控制，否则会影响我们的正常作业。
总结
这一课时我们学习了 Flink 的时间类型和水印生成，内容偏多并且水印部分理解起来需要时间，建议你结合源码再进一步学习。