三、Watermark机制

3.1 什么是Watermark

在处理无限数据流的情况下，还有一大难题便是**“乱序”**。事件的到达时间并不能按照事件的发生时间的 相同顺序的得到，就如同现实生活中的网络延时的不同。加入我们简单的用来一个数据处理一个数据的 方式，很有可能由于没有按照正确的发生顺序执行导致了错误的结果，这样的后果是不堪设想的。举一 个简单的例子，如果我们在线上抢答，如果是按照接受信号的顺序处理，那么抢答的就成了完全变成比 拼网速波动的结果，这是没有意义的。当然我们可以自然地想到一种解决方案，就是给事件标记时间 戳，我们处理完**1**，就等**2**，等到了在处理，再等**3...**，但是这样处理的问题在于，我万一**2**数据丢失了， 那我就一直等**2**，那不就卡死了，所以面对这种情况我们引入了水位线的概念。

> **Watermarks are the progress indicators in the data streams. A watermark signifies that no events with a timestamp smaller or equal to the watermark's time will occur after the water. A watermark with timestamp `T` indicates that the stream's event time has progressed to time `T`.**
> **Watermarks are created at the sources and propagate through the streams and operators. In some cases a watermark is only a heuristic, meaning some events with a lower timestamp may still follow. In that case, it is up to the logic of the operators to decide what to do with the "late events". Operators can for example ignore these late events, route them to a different stream, or send update to their previously emitted results.**
> **When a source reaches the end of the input, it emits a final watermark with timestamp {@code Long.MAX_VALUE}, indicating the "end of time".**
> **Note: A stream's time starts with a watermark of {@code Long.MIN_VALUE}. That means that all records in the stream with a timestamp of {@code Long.MIN_VALUE} are immediately late.**

上文源自源码中对于**watermark**的阐述，简单来说，就是具有时间戳 `t`的 **watermark** 可以被理解为断言了所有时间戳**小于或等于** `t`的事件都（在某种合理的概率上）已经到达了。这正是 ***watermark*** 的作用，他们定义了何时不再等待更早的数据。（用水位线来命名也合乎情理，水位以下事件全部到达，可以开始处理）

3.2 Watermark的作用

watermark是用于处理乱序事件的，而正确的处理乱序事件，通常用**watermark**机制结合**window**来实 现。

我们知道，流处理从事件产生，到流经**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、背压等原因，导致乱序的产生（**out-of-order**或者说**late element**）。

但是对于**late element**，我们又不能无限期的等下去，必须要有个机制来保证一个特定的时间后，必须触发**window**去进行计算了。这个特别的机制，就是**watermark**。

3.3 Watermark的设定

知道了**watermark**的作用，我们需不需要每个数据之后都设一个**watermark**呢？这是一个相当消耗资源的事情，所以我们理所应当的***设想不同的策略来生成 watermark。***

Watermark的设定主要分为两种：

第一种是**Punctuated Watermark**（间断的水位线）。如果数据是间断性的，那么可以使用这个作为产生**watermark**的方式。数据流中每一个递增的**EventTime**都会产生一个**Watermark**。

第二种是**Periodic Watermark** （周期性水位线）。周期性的获取**timestamp**和生成**watermark**。可以依赖流元素的时间，比如**EventTime**或者**ProcessingTime**。

上述两种会在下文中的源码分析中进行更详尽的阐述。

3.4 Time中概念简述

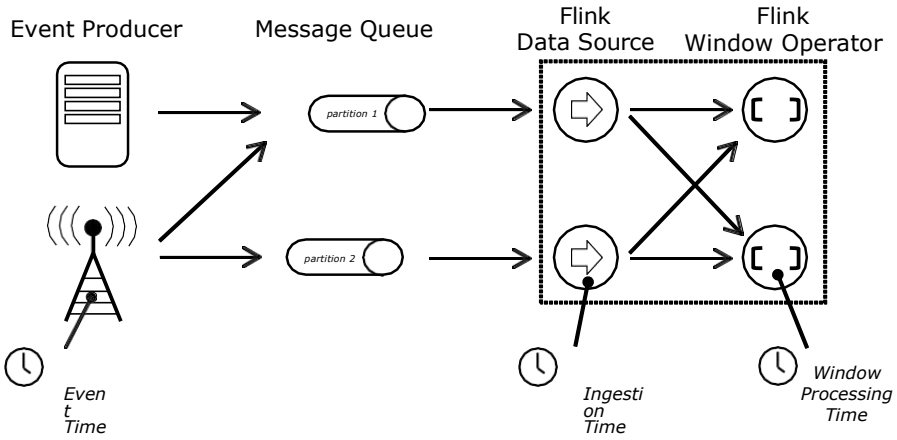
为了便于之后的理解，在这里我们简单讲述一下**apache flink**的中有关时间的定义。

那么**Flink**的**Time**分为三种：

ProcessingTime ： 处理时间，即算子处理数据的机器产生的时间，数据的流程一般是`Source` **->** `Transform` **(Operator**，即算子） **->** `Sink`（保存数据）。 `ProcessingTime`出现在`Transform`，算子计算的点。这个比较好生成，数据一旦到了算子就会生成。如果在分布式系统中或异步系统中，对于一 个消息系统，有的机器处理的块，有的机器消费的慢，算子和算子之间的处理速度，还有可能有些机器 出故障了， `ProcessingTime` 将不能根据时间产生决定数据的结果，因为什么时候到了算子才会生成这个时间。

EventTime : 事件时间，此事件一般是产生数据的源头生成的。带着`event time`的事件进入**flink** 的`source` 之后就可以把事件事件进行提取，提取出来之后可以根据这个时间处理需要一致性和决定性的结果。比如，计算一个小时或者五分钟内的数据访问量。当然数据的`EventTime`可以是有序的，也可以是无序的。有序的数据大家比较好理解，比如，第一秒到第一条，第二秒到第二条数据。无序的数据，举个例子要计算五秒的数据，假如现在为**10:00:00,** 那么数据`EventTime`在**[10:00:00 10:00:05), [10:00:05 10:00:10)**，加入一条数据是**04**秒产生的，那么由于机器处理的慢，该数据在**08**秒的时候到了，这个时候我们理解该数据就是无序的。

IngestionTime ： 摄入时间，即数据进入**Flink**的 `Source`的时候计入的时间。相对于以上两个时间， `IngestionTime` 介于 `ProcessingTime` 和 `EventTime`之间，相比于 `ProcessingTime`，生成的更加方便快捷， `ProcessingTime`每次进入一个`Operator` **(**算子，即**map**、**flatMap**、**reduce**等）都会产生一个时间，而`IngestionTime` 在进入**Flink**的时候就产生了 `timestamp` 。相比于`eventTime` ，它不能处理无序的事件，因为每次进入`source`产生的时间都是有序的， `IngestionTime`也无须产生`WaterMark`，因为会自动生成。



3.5 Watermark在Eventtime中的源码分析

源码分析不会对每个文件都进行细致的分析，只会对具有代表性，体现**flink**设计的独特性的代码部分进行分析和解读。在同类文件中只会对一种进行细致的解读，同时也会提及其他同类文件类似之处，以此 避免源码分析的冗余和枯燥。

3.5.1 水位线

flink-core\src\main\java\org\apache\flink\api\common\eventtime\Watermark.java

```
public final class Watermark implements Serializable {

    private static final long serialVersionUID = 1L;
```

```
    /** Thread local formatter for stringifying the timestamps. */
    private static final ThreadLocal<SimpleDateFormat> TS_FORMATTER =
ThreadLocal.withInitial(
        () -> new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS"));

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

    /** 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;
    }

    /**
     * Returns the timestamp associated with this Watermark.
     */
    public long getTimestamp()
        { return timestamp;
    }

    /**
     * Formats the timestamp of this watermark, assuming it is a millisecond
timestamp.
     * The returned format is "yyyy-MM-dd HH:mm:ss.SSS".
     */
    public String getFormattedTimestamp() {
        return TS_FORMATTER.get().format(new Date(timestamp));
    }

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

    @Override
    public boolean equals(Object o)
        { return this == o ||
                o != null &&
                o.getClass() == Watermark.class &&
                ((Watermark) o).timestamp ==this.timestamp;
    }

    @Override
    public int hashCode() {
        return Long.hashCode(timestamp);
    }

    @Override
    public String toString() {
        return "Watermark @ " + timestamp + " (" + getFormattedTimestamp() +
')';
    }
```

```
}
```

这里就给出了我们探讨已久的水位线的定义。可以到这个被封装好的水位线内部的变量。MAX_WATERMARK标记了事件结束的水位线（包含事件结束的时间戳），timestamp则是水位线内部的时间戳（水位线的本质就是时间戳，只不过它有特殊的含义，标记了一个流时间的最低点）。我们再看 封装好的函数： Watermark 是初始化函数，用来新建一个水位线，getTimestamp 用来获取内部的时间戳的值， getFormattedTimestamp 则是带格式的输出时间戳（便于调试和阅读），equals 给出了水位线时间比较是否相等的方式， hashCode则是以哈希值输出时间戳，最后toString是输出一端有意义有格式便于阅读的字符串。

我们在这一提一句WatermarkGenerator这一对象，这一对象包含了 AssignerWithPeriodicWatermarks 和 AssignerWithPunctuatedWatermarks 两个接口的优势。也就是说他既可以生成**Punctuated**的水位线，也可以生成的**Periodic**水位线，看接口如何定义。

3.5.2 有界乱序水位线

flink-core\src\main\java\org\apache\flink\api\common\eventtime\BoundedOutOfOrdernessWatermar ks.java

```
public class BoundedOutOfOrdernessWatermarks<T> implements WatermarkGenerator<T>
{

    /** The maximum timestamp encountered so far. */
    private long maxTimestamp;

    /** The maximum out-of-orderness that this watermark generator assumes. */
    private final long outOfOrdernessMillis;

    /**
     * Creates a new watermark generator with the given out-of-orderness bound.
     *
     * @param maxOutOfOrderness The bound for the out-of-orderness of the event
timestamps.
     */
    public BoundedOutOfOrdernessWatermarks(Duration maxOutOfOrderness)
        { checkNotNull(maxOutOfOrderness, "maxOutOfOrderness");
        checkArgument(!maxOutOfOrderness.isNegative(),"maxOutOfOrderness cannot
be negative");

        this.outOfOrdernessMillis = maxOutOfOrderness.toMillis();

        // start so that our lowest watermark would be Long.MIN_VALUE.
        this.maxTimestamp = Long.MIN_VALUE + outOfOrdernessMillis + 1;
    }

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

    @Override
    public void onEvent(T event, long eventTimestamp, WatermarkOutput output)
        { maxTimestamp = Math.max(maxTimestamp, eventTimestamp);
    }

    @Override
    public void onPeriodicEmit(WatermarkOutput output) {
```

```
        output.emitWatermark(new Watermark(maxTimestamp - outOfOrdernessMillis -
1));
    }
}
```

首先从上述源码中可以看出，`BoundedOutOfOrdernessWatermarks`这一函数继承了`WatermarkGenerator`的两个接口（由于只是继承了接口，所以`WatermarkGenerator`不在赘述），`onEvent`和 `onPeriodicEmit`。我们先从该对象的内部变量开始分析。其中`outOfOrdernessMillis`这一变量体现这个对象的主要特点：能够容纳一定的乱序性。 `outOfOrdernessMillis` 指该水位线能容纳的最大乱序性，假设目前该水位线在T处，并不是说比T时间早的流不会再之后（未来）再到达，而是 比 T - B时间早的流不会再之后（未来）再到达，其中B就是该对象的容错，这一容错就体现在`outOfOrdernessMillis`这一变量中。而这一值通过传入的`duration`类型的参数得到。

其 中 `maxTimestamp`以 及 `emitWatermark`产 生 的 新`Watermark`都是根据这一容错值`outOfOrdernessMillis`进行了修正。其中`onEvent`函数会在每次事件都被调用，检测该事件的时间戳，并根据函数本身会制定规则根据事件本身性质决定是否发射新的水位线。可以看出该对象并没有发 射流水线，而只是单纯更新目前已到达事件的最大时间戳。而`onPeriodicEmit`则是会间断性的被调用，之后按照函数内容发射新的水位线。该对象则是根据这一容错值`outOfOrdernessMillis`进行了修正发射水位线。这便是上述**Periodic Watermark**的一种。

3.5.3 水位线输出的多路选择器

flink-core\src\main\java\org\apache\flink\api\common\eventtime\WatermarkOutputMultiplexer.java

```
public class WatermarkOutputMultiplexer {

    /**
     * The {@link WatermarkOutput} that we use to emit our multiplexed watermark
updates. We assume
     * that outside code holds a coordinating lock so we don't lock in this class
when accessing
     * this {@link WatermarkOutput}.
     */
    private final WatermarkOutput underlyingOutput;

    /** The combined watermark over the per-output watermarks. */
    private long combinedWatermark = Long.MIN_VALUE;

    /**
     * Map view, to allow finding them when requesting the {@link
WatermarkOutput} for a given id.
     */
    private final Map<String, OutputState> watermarkPerOutputId;

    /**
     * List of all watermark outputs, for efficient access.
     */
    private final List<OutputState> watermarkOutputs;

    /**
     * Creates a new {@link WatermarkOutputMultiplexer} that emits combined
updates to the given
     * {@link WatermarkOutput}.
     */
    public WatermarkOutputMultiplexer(WatermarkOutput underlyingOutput) {
```

```
        this.underlyingOutput = underlyingOutput;
        this.watermarkPerOutputId = new HashMap<>();
        this.watermarkOutputs = new ArrayList<>();
    }

    /**
     * Registers a new multiplexed output, which creates internal states for that
output and returns
     * an output ID that can be used to get a deferred or immediate {@link
WatermarkOutput} for that
     * output.
     */
    public void registerNewOutput(String id) {
        final OutputState outputState = new OutputState();

        final OutputState previouslyRegistered = watermarkPerOutputId.putIfAbsent(id,
outputState);
        checkState(previouslyRegistered == null, "Already contains an output for ID
%s", id);

        watermarkOutputs.add(outputState);
    }

    public boolean unregisterOutput(String id) {
        final OutputState output = watermarkPerOutputId.remove(id);
        if (output != null) {
            watermarkOutputs.remove(output);
            return true;
        } else {
            return false;
        }
    }

    /**
     * Returns an immediate {@link WatermarkOutput} for the given output ID.
     *
     * <p>>See {@link WatermarkOutputMultiplexer} for a description of immediate and
deferred
     * outputs.
     */
    public WatermarkOutput getImmediateOutput(String outputId) {
        final OutputState outputState = watermarkPerOutputId.get(outputId);
        Preconditions.checkArgument(outputState != null, "no output registered
under id %s", outputId);
        return new ImmediateOutput(outputState);
    }

    /**
     * Returns a deferred {@link WatermarkOutput} for the given output ID.
     *
     * <p>>See {@link WatermarkOutputMultiplexer} for a description of immediate and
deferred
     * outputs.
     */
    public WatermarkOutput getDeferredOutput(String outputId) {
        final OutputState outputState = watermarkPerOutputId.get(outputId);
        Preconditions.checkArgument(outputState != null, "no output registered
under id %s", outputId);
```

```
        return new DeferredOutput(outputState);
    }

    /**
     * Tells the {@link WatermarkOutputMultiplexer} to combine all outstanding
deferred watermark
     * updates and possibly emit a new update to the underlying {@link
WatermarkOutput}.
     */
    public void onPeriodicEmit()
        { updateCombinedWatermark()
        ;
    }

    /**
     * Checks whether we need to update the combined watermark. Should be called when
a newly
     * emitted per-output watermark is higher than the max so far or if we need to
combined the
     * deferred per-output updates.
     */
    private void updateCombinedWatermark() {
        long minimumOverAllOutputs = Long.MAX_VALUE;

        boolean hasOutputs = false;
        boolean allIdle = true;
        for (OutputState outputState : watermarkOutputs) { if
            (!outputState.isIdle()) {
                minimumOverAllOutputs = Math.min(minimumOverAllOutputs,
outputState.getWatermark());
                allIdle = false;
            }
            hasOutputs = true;
        }

        // if we don't have any outputs minimumOverAllOutputs is not valid, it's
still
        // at its initial Long.MAX_VALUE state and we must not emit that if
        (!hasOutputs) {
            return;
        }

        if (allIdle)
            { underlyingOutput.markIdle();
        } else if (minimumOverAllOutputs > combinedWatermark)
            { combinedWatermark = minimumOverAllOutputs;
            underlyingOutput.emitWatermark(new
Watermark(minimumOverAllOutputs));
        }
    }

    /**
     * Per-output watermark state.
     */
    private static class OutputState {
        private long watermark = Long.MIN_VALUE;
        private boolean idle = false;

        /**
```

```
         * Returns the current watermark timestamp. This will throw {@link
IllegalStateException} if
         * the output is currently idle.
         */
        public long getWatermark()
            { checkState(!idle, "Output is idle.");
            return watermark;
        }

        /**
         * Returns true if the watermark was advanced, that is if the new
watermark is larger than
         * the previous one.
         *
         * <p>Setting a watermark will clear the idleness flag.
         */
        public boolean setWatermark(long watermark)
            { this.idle = false;
            final boolean updated = watermark > this.watermark;
            this.watermark = Math.max(watermark, this.watermark);
            return updated;
        }

        public boolean isIdle()
            { return idle;
        }

        public void setIdle(boolean idle)
            { this.idle = idle;
        }
    }

    /**
     * Updating the state of an immediate output can possible lead to a combined
watermark update to
     * the underlying {@link WatermarkOutput}.
     */
    private class ImmediateOutput implements WatermarkOutput {

        private final OutputState state;

        public ImmediateOutput(OutputState state)
            { this.state = state;
        }

        @Override
        public void emitWatermark(Watermark watermark)
            { long timestamp = watermark.getTimestamp();
            boolean wasUpdated = state.setWatermark(timestamp);

            // if it's higher than the max watermark so far we might have to
update the
            // combined watermark
            if (wasUpdated && timestamp > combinedWatermark)
                { updateCombinedWatermark();
            }
        }
```

```
        @Override
        public void markIdle()
            { state.setIdle(true);

            // this can always lead to an advancing watermark. We don't know if
this output
            // was holding back the watermark or not.
            updateCombinedWatermark();
        }
    }

    /**
     * Updating the state of a deferred output will never lead to a combined
watermark update. Only
     * when {@link WatermarkOutputMultiplexer#onPeriodicEmit()} is called will
the deferred updates
     * be combined into a potential combined update of the underlying {@link
WatermarkOutput}.
     */
    private static class DeferredOutput implements WatermarkOutput {

        private final OutputState state;

        public DeferredOutput(OutputState state)
            { this.state = state;
        }

        @Override
        public void emitWatermark(Watermark watermark)
            { state.setWatermark(watermark.getTimestamp());
        }

        @Override
        public void markIdle()
            { state.setIdle(true);
        }
    }
}
```

这是本次我的源码分析中代码最长，结构算是最复杂的文件，然而只要我们利用面向对象的思想来解读，实际上还是能很快掌握此对象的功能与结构。首先`WatermarkOutputMultiplexer`是协助`WatermarkOutput`完成功能的对象。`WatermarkOutput`是基础层。`WatermarkOutputMultiplexer` 通过将多个分区的水位线更新结合成新的水位线，并输入基础层`WatermarkOutput` 。他有两种输出方式，一种立即输出，一种延时输出，立即输出会在更新内部状态后立即输出给基础层，而延时输出只有在调 用`onPeriodicEmit`时才会输出到基础层。

同 时`WatermarkOutputMultiplexer` 还需要通过注册获得**id**的方式才能调用`getImmediateOutput` 和`getDeferredOutput` 这两个函数来获得对应的立即输出和延迟输出的值。我们不禁会问，这里的注册有什么意义？实际上这正体现了面向对象的思想，封装。实际上该对象的目的是处理多水位线并输出的 基础层，这一段输出是封装好的，对外界是完全不可见的，但是如果你想知道中间的过程，你必须通过 方法来注册，此时它才会新建一个`OutputState`来给你输出你想看到的状态值。

3.5.4 时间戳赋值器

flink-core\src\main\java\org\apache\flink\api\common\eventtime\TimestampAssigner.java

```
public interface TimestampAssigner<T> {

    /**
     * The value that is passed to {@link #extractTimestamp} when there is no
previous timestamp
     * attached to the record.
     */
    long NO_TIMESTAMP = Long.MIN_VALUE;

    /**
     * Assigns a timestamp to an element, in milliseconds since the Epoch. This is
independent of
     * any particular time zone or calendar.
     *
     * <p>The method is passed the previously assigned timestamp of the element.
     * That previous timestamp may have been assigned from a previous assigner.
If the element did
     * not carry a timestamp before, this value is {@link #NO_TIMESTAMP} (=
{@code Long.MIN_VALUE}:
     * {@value Long#MIN_VALUE}).
     *
     * @param element The element that the timestamp will be assigned to.
     * @param recordTimestamp The current internal timestamp of the element,
     *                        or a negative value, if no timestamp has been
assigned yet.
     * @return The new timestamp.
     */
    long extractTimestamp(T element, long recordTimestamp);
}
```

`TimestampAssigner` 这一对象的代码实际上非常简单，可以看到没有什么实质内容，但是笔者想通过这个通用的框架来把不同类别时间戳的赋值器对象一同总结。可以看到该对象的变量就只有一个时间戳，一毫秒为单位。其他类别的`TimestampAssigner` 如 `TimestampAssigner` ，`IngestionTimeAssigner` 只是根据定义，将不同的定义时间戳赋值进变量，并且按照定义赋值接口`extractTimestamp`完成获取时间戳的功能。

3.6 设计模式浅析

实际上本人阅读的这一部分代码并没有很明确的设计模式的体现，只是对于类（接口）单纯的继承，很多人喜欢把自己不知道的设计模式都归类为单例，就是只是单纯有一个继承的的关系。但是这并不对，因为单例要保证的是一个对象只有一个实例，这个其实很多代码设计中是完全没有这个**“保证”**的体现的。这里我更愿意把**eventtime**中，尤其是水位线**watermark**的类的设计模式归类到工厂模式，也就是**java**最常用的设计模式。

工厂模式（**Factory Pattern**）是**Java**中最常用的设计模式之一。这种类型的设计模式属于创建型模式， 它提供了一种创建对象的最佳方式。在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过使用一个共同的接口来指向新创建的对象。在本代码中就是，首先对于水位线**watermark**接口进行定义（参见**3.5.1)**，之后在对于特定的水位线完善接口函数（参见**3.5.2).**

四、感想与总结

实际上**apache flink eventtime**的源码部分实质代码非常少，但实际工作量绝不在少数。**apache flink**的代码是大道至简，虽然只有区区几行，对象里也仅有几个方法和特性，不过想要理解代码的含义，门槛非常之高，需要极度充足的预备知识，甚至要把**apache flink**很多其他部分的代码也理解透彻，而对**flink**的运行原理和机制更要非常熟悉，不然就面对短短几行代码，就犹如面对天书。这次源码分析让我对大数据的流处理有了相当深刻的理解（相比之前的我），收获颇丰，当然实际上我的源码阅读报告还挖了很多坑等待填满，也有很多内容有待学习，很多方面有待更详细的叙述，希望在之后的版本中更新的更好。

Reference

1、*Apache Flink*介绍 左边的天堂 **https://blog.csdn.net/tiantang_1986/article/details/81204597**

2、*Apache Flink*最详细的概述假的鱼 **https://blog.csdn.net/m0_37803704/article/details/86563457**

3、*Apache Flink* 官方文档 **https://flink.apache.org/flink-architecture.html**

4、*Apache Flink* 维基百科 **https://en.wikipedia.org/wiki/Apache_Flink**

5、*Flink* 原理与实现：*Window* 机制 伍翀 **http://wuchong.me/blog/2016/05/25/flink-internals-window-mechanism/**

6、*Apache Flink*：流处理中*Window*的概念 **lmalds**李麦迪 **https://blog.csdn.net/lmalds/article/details/52704170**

7、*Apache Flink Watermark* 金竹 **https://developer.aliyun.com/article/666056**

8、*Flink*系列之*Time*和*WaterMark* 黄青石 **https://www.cnblogs.com/huangqingshi/p/12178460.html**

9、*Flink*的*Window*源码剖析 **WenWu_Both** **https://blog.csdn.net/wenwu_both/article/details/100879556**