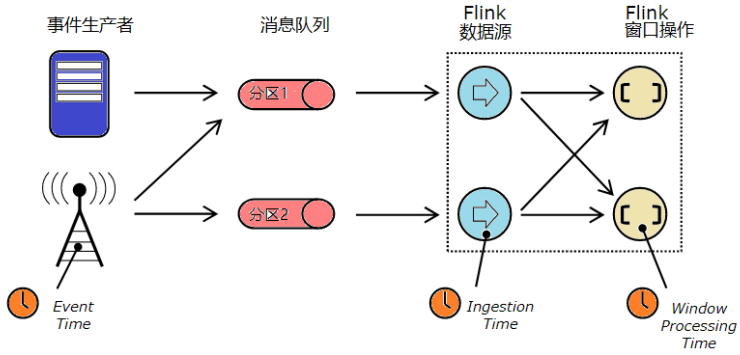
Flink的时间与watermarks详解

我们在使用Flink的时候，避免不了要和时间(time)、水位线(watermarks)打交道，理解这些概念是开发分布式流处理应用的基础。那么Flink支持哪些时间语义？Flink是如何处理乱序事件的？什么是水位线？水位线是如何生成的？水位线的传播方式是什么？让我们带着这些问题来开始本文的内容。

时间语义

基本概念

时间是Flink等流处理中最重要的概念之一，在 Flink 中 Time 可以分为三种：Event-Time，Processing-Time 以及 Ingestion-Time，如下图所示：



- Event Time

事件时间，事件(Event)本身的时间，即数据流中事件实际发生的时间，通常使用事件发生时的时间戳来描述，这些事件的时间戳通常在进入流处理应用之前就已经存在了，事件时间反映了事件真实的发生时间。所以，基于事件时间的计算操作，其结果是具有确定性的，无论数据流的处理速度如何、事件到达算子的顺序是否会乱，最终生成的结果都是一样的。

- Ingestion Time

摄入时间，事件进入Flink的时间，即将每一个事件在数据源算子的处理时间作为事件时间的时间戳，并自动生成水位线(watermarks,关于watermarks下文会详细分析)。

Ingestion Time从概念上讲介于Event Time和Processing Time之间。与Processing Time相比，它的性能消耗更多一些，但结果却更可预测。由于 Ingestion Time使用稳定的时

间戳（在数据源处分配了一次），因此对记录的不同窗口操作将引用相同的时间戳，而在Processing Time中每个窗口算子都可以将记录分配给不同的窗口。

与Event Time相比，Ingestion Time无法处理任何乱序事件或迟到的数据，即无法提供确定的结果，但是程序不必指定如何生成水位线。在内部，Ingestion Time与Event Time非常相似，但是可以实现自动分配时间戳和自动生成水位线的功能。

- Processing Time

处理时间，根据处理机器的系统时钟决定数据流当前的时间，即事件被处理时当前系统的时间。还以窗口算子为例(关于window，下文会详细分析)，基于处理时间的窗口操作是以机器时间来进行触发的，由于数据到达窗口的速率不同，所以窗口算子中使用处理时间会导致不确定的结果。在使用处理时间时，无需等待水位线的到来后进行触发窗口，所以可以提供较低的延迟。

对比

经过上面的分析，应该对Flink的时间语义有了大致的了解。不知道你会不会有这样一个疑问：既然事件时间已经能够解决所有的问题了，那为何还要用处理时间呢？其实处理时间有其特定的使用场景，处理时间由于不用考虑事件的延迟与乱序，所以其处理数据的延迟较低。因此如果一些应用比较重视处理速度而非准确性，那么就可以使用处理时间，比如要实时监控仪表盘。总之，虽然处理时间的延迟较低，但是其结果具有不确定性，事件时间虽然有延迟，但是能够保证处理的结果具有准确性，并且可以处理延迟甚至无序的数据。

使用

上一小结讲述了三种时间语义的基本概念，接下来将从代码层面讲解在程序中该如何配置这三种时间语义。首先来看一段代码：

```
/** The time characteristic that is used if none other is set. */
        private static final TimeCharacteristic DEFAULT_TIME_CHARACTERIST
IC = TimeCharacteristic.ProcessingTime;
//省略的代码
/** The time characteristic used by the data streams. */
        private TimeCharacteristic timeCharacteristic = DEFAULT_TIME_CHA
RACTERISTIC;
```

上述两行代码摘自StreamExecutionEnvironment类，可以看出，Flink在流处理程序中默认的时间语义是Processing Time，那么该如何修改默认的时间语义呢？很简单，再来看一段代码，下面的代码片段同样来自于StreamExecutionEnvironment类：

```
    /**
         * 如果使用Processing Time或者Event Time，默认的水位线间隔时间是200毫秒
```

```
         * 可以通过ExecutionConfig#setAutoWatermarkInterval(long)设置
         * @param characteristic The time characteristic.
         */
        @PublicEvolving
        public void setStreamTimeCharacteristic(TimeCharacteristic chara
cteristic) {
                this.timeCharacteristic = Preconditions.checkNotNull(cha
racteristic);
                if (characteristic == TimeCharacteristic.ProcessingTime)
 {
                        getConfig().setAutoWatermarkInterval(0);
                } else {
                        getConfig().setAutoWatermarkInterval(200);
                }
        }
```

上述的方法可以配置不同的时间语义，参数TimeCharacteristic是一个枚举类，包括ProcessingTime，IngestionTime，EventTime三个元素。具体使用方式如下：

```
//env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime);
//env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

watermarks

在解释watermarks(水位线)之前，先看一个我们身边发生的真实案例。高考，是大家非常熟悉的场景。如果把高考的考试安排简单地看作是一个流处理应用，那么，每一个考试科目的开始时间到结束时间就是一个窗口，每个考生可以理解成一条记录，考生到达考场的时间可以理解成记录的时间戳，而考试可以理解成某种算子操作。大家都知道，高考考试在开考后15分钟是不允许进场的，这个规定可以理解成一个水位线，比如，上午第一场语文考试，开考时间是9:30，允许在9:45之前进入考场，那么9:45这个时间可以理解成一个水位线。在开考之前，有的同学喜欢提前到考场，有的同学喜欢卡点到考场。假设有个同学叫**考必胜**,ta是卡着时间点到的考场，但是早上由于吃了不干净的东西，突然感觉肚子不适，无奈之下在厕所里耽误了16分钟，那么按照规定，此时考必胜是不能够进入考场的，因为此时已经默认所有考生都已经在考场了，此时考试也已经触发，那么卡必胜就可以理解为迟到的事件。以上就是对窗口、事件时间以及水位线的简单理解，下面开始详细解释什么水位线。

基本概念

在上一节中，详细讲解了Flink提供的三种时间语义，在讲解这三种时间语义的时候，提到了一个名词—水位线，那么究竟什么是水位线呢？先来看一个例子，假如要每5分钟统计一次过去1个小时内的热门商品的topN，这是一个典型的滑动窗口操作，那么基于事件时间的窗口该在什么时候出发计算呢？换句话说，我们要等多久才能够确定已经接收到

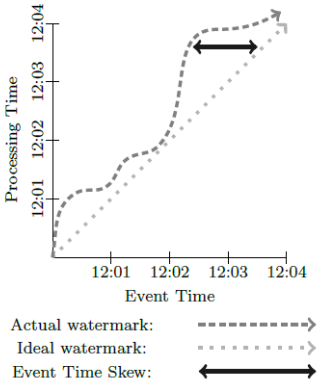
了特定时间点之前的所有事件，另一方面，由于网络延迟等原因，会产生乱序的数据，在进行窗口操作时，不能够无限期的等待下去，需要一个机制来告诉窗口在某个特定时间来触发window计算，即认为小于等于该时间点的数据都已经到来了。这个机制就是watermark(水位线)，可以用来处理乱序事件。

水位线是一个全局的进度指标，表示可以确定不会再有延迟的事件到来的某个时间点。从本质上讲，水位线提供了一个逻辑时钟，用来通知系统当前的事件时间。比如，当一个算子接收到了W(T)时刻的水位线，就可以大胆的认为不会再接收到任何时间戳小于或等于W(T)的事件了。水位线对于基于事件时间的窗口和处理乱序数据是非常关键的，算子一旦接收到了某个水位线，就相当于接到一支穿云箭的信号：所有特定时间区间的数据都已集结完毕，可以进行窗口触发计算。

既然已经说了，事件是会存在乱序的，那这个乱序的程度究竟有多大呢，这个就不太好确定了，总之总会有些迟到的事件慢慢悠悠的到来。所以，水位线其实是一种在**准确性**与**延迟**之间的权衡，如果水位线设置的非常苛刻，即不允许有掉队的数据出现，虽然准确性提高了，但这在无形之中增加了数据处理的延迟。反之，如果水位线设置的非常激进，即允许有迟到的数据发生，那么虽然降低了数据处理的延迟，但数据的准确性会较低。

所以，水位线是中庸之道，过犹不及。在很多现实应用中，系统无法获取足够多的信息来确定完美的水位线，那么该怎么办呢？Flink提供了某些机制来处理那些可能晚于水位线的迟到时间，用户可以根据应用的需求不同，可以将这些漏网之鱼(迟到的数据)舍弃掉，或者写入日志，或者利用他们修正之前的结果。

上面说到没有完美的水位线，可能还是很抽象。接下来，我们再看一幅图，从图中可以很直观地观察真实的水位线与理想中的完美水位线之间的关系，如下图：

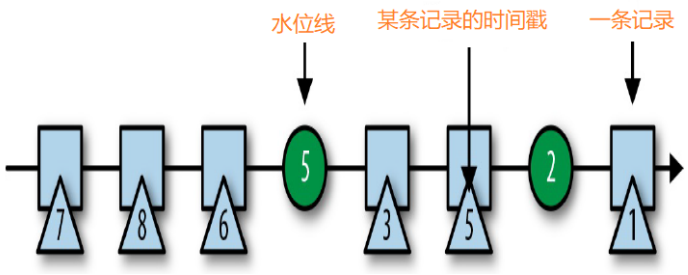


上图的浅灰色直虚线表示理想的水位线，深灰色的弯曲虚线表示现实中的水位线，黑色直线表示两者之间的偏差。在理想状态下，这种偏差为0，因为总是在时间发生时就会立即处理，即事件的真实时间与处理事件的时间是一致的。比如，12:01产生的事件刚好在12:01时被处理，12:02产生的事件刚好在12:02时被处理。但是现实总会有迟到的数据产生，比如网络延迟的原因，所以真实的情况会像深灰色的弯曲虚线表示的那样，即12:01

产生的数据可能会在12:01之后被处理，12:02产生的数据在12:02时被处理，12:03时产生的数据会被在12:03之后处理。这种动态的偏差在分布式处理系统中是非常常见的。

水位线图解

在上一小节，通过语言描述对水位线的概念进行了详细解读，在本小节会通过图解的方式解析水位线的含义，这样更能加深对水位线的理解。如下图所示：



如上图，矩形表示一条记录，三角表示该条记录的时间戳(真实发生时间)，圆圈表示水位线。可以看到上面的数据是乱序的，比如当算子接收到为2的水位线时，就可以认为时间戳小于等于2的数据都已经到来了，此时可以触发计算。同理，接收到为5的水位线时，就可以认为时间戳小于或等于5的数据都已经到来了，此时可以触发计算。

可以看出水位线是单调递增的，并且和记录的时间戳存在联系，一个时间戳为T的水位线表示接下来所有记录的时间戳一定都会大于T。

水位线的传播

现在，或许你已经对水位线是什么有了一个初步的认识，接下来将会介绍水位线是怎么在Flink内部传播的。关于水位线的传播策略可以归纳为3点：

- 首先，水位线是以广播的形式在算子之间进行传播
- Long.MAX_VALUE表示事件时间的结束，即未来不会有数据到来了
- 单个分区的输入取最大值，多个分区的输入取最小值

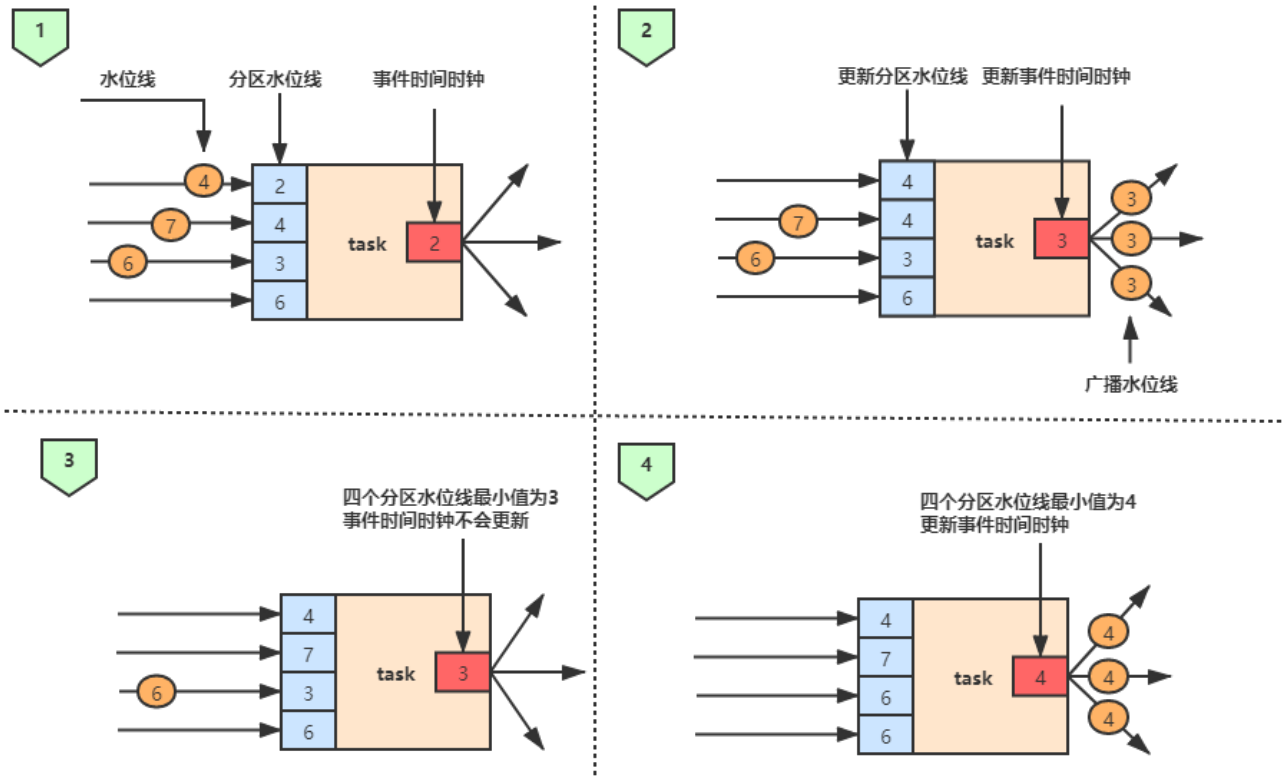
关于Long.MAX_VALUE的解释，先看一段代码，如下：

```
 /**
  * 当一个source关闭时，会输出一个Long.MAX_VALUE的水位线，当一个算子接收到该水位线时
,
  * 相当于接收到一个信号：未来不会再有数据输入了
  */
@PublicEvolving
public final class Watermark extends StreamElement {

```

```
        //表示事件时间的结束
        public static final Watermark MAX_WATERMARK = new Watermark(Long.
MAX_VALUE);
    //省略的代码
}
```

关于另外两条策略的解释，可以从下图中得到：



如上图，一个任务会为它的每个分区都维护一个分区水位线(partition watermark)，当收到每个分区传来的水位线时，任务首先会让当前分区水位线的值与接收的水位线值相比较，如果新接收的水位线值大于当前分区水位线值，则会将对应的分区水位线值更新为较大的水位线值(如上图中的2步骤)，接着，任务会把事件时钟调整为当前分区水位线值的最小值，如上图步骤2 ，由于当前分区水位线的最小值为3，所以将事件时间时钟更新为3，然后将值为3的水位线广播到下游任务。步骤3与步骤4的处理逻辑同上。

同时我们可以注意到这种设计其实有一个局限，具体体现在没有对分区(partition)是否来自于不同的流进行区分，比如对于两条流或多条流的Union或Connect操作，同样是按照全部分区水位线中最小值来更新事件时间时钟，这就导致所有的输入记录都会按照基于同一个事件时间时钟来处理，这种一刀切的做法对于同一个流的不同分区而言是无可厚非的，但是对于多条流而言，强制使用一个时钟进行同步会对整个集群带来较大的性能开销，比如当两个流的水位线相差很大是，其中的一个流要等待最慢的那条流，而较快的流的记录会在状态中缓存，直到事件时间时钟到达允许处理它们的那个时间点。

水位线的生成方式

通常情况下，在接收到数据源之后应该马上为其生成水位线，即越靠近数据源越好。Flink提供两种方式生成水位线，其中一种方式为在数据源完成的，即利用SourceFunction在应用读入数据流的时候分配时间戳与水位线。另一种方式是通过实现接口的自定义函数，该方式又包括两种实现方式：一种为周期性生成水位线，即实现AssignerWithPeriodicWatermarks接口，另一种为定点生成水位线，即实AssignerWithPunctuatedWatermarks接口。具体如下图所示：

数据源方式

该方式主要是实现自定义数据源，数据源分配时间戳和水位线主要是通过内部的SourceContext对象实现的，先看一下SourceFunction的源码，如下：

```
public interface SourceFunction<T> extends Function, Serializable {

        void cancel();

        interface SourceContext<T> {

                void collect(T element);
        /**
                * 用于输出记录并附属一个与之关联的时间戳
                */
                @PublicEvolving
                void collectWithTimestamp(T element, long timestamp);
        /**
                * 用于输出传入的水位线
                */
                @PublicEvolving
                void emitWatermark(Watermark mark);
        /**
                * 将自身标记为空闲状态
                * 某个某个分区不在产生数据，会阻碍全局水位线前进，
                * 因为收不到新的记录，意味着不会发出新的水位线，
                * 根据水位线的传播策略，会导致整个应用都停止工作
                * Flink提供一种机制，将数据源函数暂时标记为空闲，
                * 在空闲状态下，Flink水位线的传播机制会忽略掉空闲的数据流分区
                */
```

```
                @PublicEvolving
                void markAsTemporarilyIdle();

                Object getCheckpointLock();

                void close();
        }
}
```

从上面对的代码可以看出，通过SourceContext对象的方法可以实现时间戳与水位线的分配。

自定义函数的方式

使用自定义函数的方式分配时间戳，只需要调用assignTimestampsAndWatermarks()方法，传入一个实现AssignerWithPeriodicWatermarks或者AssignerWithPunctuatedWatermarks接口的分配器即可，如下代码所示：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecution
Environment()
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
        SingleOutputStreamOperator<UserBehavior> userBehavior = env
                .addSource(new MysqlSource())
                .assignTimestampsAndWatermarks(new MyTimestampsAndWaterm
arks());
```

- **周期分配器(AssignerWithPeriodicWatermarks)**

该分配器是实现了一个AssignerWithPeriodicWatermarks的用户自定义函数，通过重写extractTimestamp()方法来提取时间戳，提取出来的时间戳会附加在各自的记录上，查询得到的水位线会注入到数据流中。

周期性的生成水位线是指以固定的时间间隔来发出水位线并推进事件时间的前进，关于默认的时间间隔在上文中也有提到，根据选择的时间语义确定默认的时间间隔，如果使用Processing Time或者Event Time，默认的水位线间隔时间是200毫秒，当然用户也可以自己设定时间间隔，关于如何设定，先看一段代码，代码来自于ExecutionConfig类：

```
    /**
    * 设置生成水位线的时间间隔
    * 注：自动生成watermarks的时间间隔不能是负数
         */
        @PublicEvolving
        public ExecutionConfig setAutoWatermarkInterval(long interval) {
                Preconditions.checkArgument(interval >= 0, "Auto waterma
```

```
rk interval must not be negative.");
                this.autoWatermarkInterval = interval;
                return this;
            }
```

所以，如果要调整默认的200毫秒的间隔，可以调用setAutoWatermarkInterval()方法，具体使用如下：

```
  //每3秒生成一次水位线
env.getConfig().setAutoWatermarkInterval(3000);
```

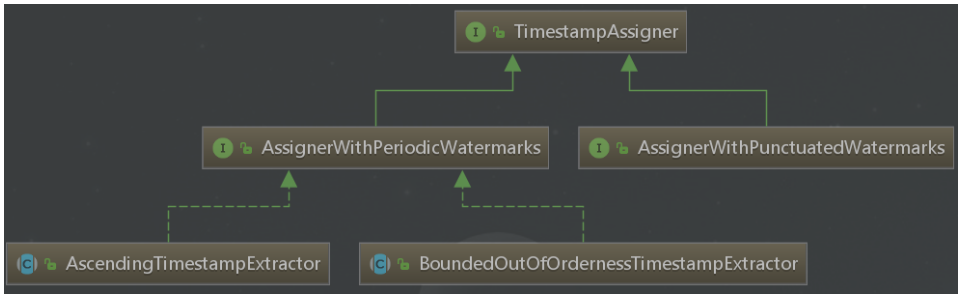
上面指定了每隔3秒生成一次水位线，即每隔3秒会自动向流里注入一个水位线，在代码层面，Flink会每隔3秒钟调用一次AssignerWithPeriodicWatermarks的getCurrentWatermark()方法，每次调用该方法时，如果得到的值不为空并且大于上一个水位线的时间戳，那么就会向流中注入一个新的水位线。这项检查可以有效地保证了事件时间的递增的特性，一旦检查失败也就不会生成水位线。下面给出一个实现周期分配水位线的例子：

```
public class MyTimestampsAndWatermarks implements AssignerWithPeriodicWa
termarks<UserBehavior> {
    // 定义1分钟的容忍间隔时间，即允许数据的最大乱序时间
    private long maxOutofOrderness = 60 * 1000;
    // 观察到的最大时间戳
    private long currentMaxTs = Long.MIN_VALUE;

    @Nullable
    @Override
    public Watermark getCurrentWatermark() {
        // 生成具有1分钟容忍度的水位线
        return new Watermark(currentMaxTs - maxOutofOrderness);
    }

    @Override
    public long extractTimestamp(UserBehavior element, long previousElem
entTimestamp) {
        //获取当前记录的时间戳
        long currentTs = element.timestamp;
        // 更新最大的时间戳
        currentMaxTs = Math.max(currentMaxTs, currentTs);
        // 返回记录的时间戳
        return currentTs;
    }
}
```

通过查看TimestampAssignerd 继承关系可以发现(继承关系如下图)，除此之外，Flink还提供了两种内置的水位线分配器，分别为：AscendingTimestampExtractor和BoundedOutOfOrdernessTimestampExtractor两个抽象类。



关于**AscendingTimestampExtractor**，一般是在数据集的时间戳是单调递增的且没有乱序时使用，该方法使用当前的时间戳生成水位线，使用方式如下：

```
SingleOutputStreamOperator<UserBehavior> userBehavior = env
                .addSource(new MysqlSource())
                .assignTimestampsAndWatermarks(new AscendingTimestampExt
ractor<UserBehavior>() {
                    @Override
                    public long extractAscendingTimestamp(UserBehavior e
lement) {
                        return element.timestamp*1000;
                    }
                });
```

关于**BoundedOutOfOrdernessTimestampExtractor**，是在数据集中存在乱序数据的情况下使用，即数据有延迟(任意新到来的元素与已经到来的时间戳最大的元素之间的时间差)，这种方式可以接收一个表示最大预期延迟参数，具体如下：

```
SingleOutputStreamOperator<UserBehavior> userBehavior = env
                .addSource(new MysqlSource())
                .assignTimestampsAndWatermarks(new BoundedOutOfOrderness
TimestampExtractor<UserBehavior>(Time.seconds(10)) {
                    @Override
                    public long extractTimestamp(UserBehavior element) {
                        return element.timestamp*1000;
                    }
                } );
```

上述的代码接收了一个10秒钟延迟的参数，这10秒钟意味着如果当前元素的事件时间与到达的元素的最大时间戳的差值在10秒之内，那么该元素会被处理，如果差值超过10

秒，表示其本应该参与的计算，已经完成了，Flink称之为迟到的数据，Flink提供了不同的策略来处理这些迟到的数据。

- **定点水位线分配器(AssignerWithPunctuatedWatermarks)**

该方式是基于某些事件(指示系统进度的特殊元祖或标记)触发水位线的生成与发送，基于特定的事件向流中注入一个水位线，流中的每一个元素都有机会判断是否生成一个水位线，如果得到的水位线不为空并且大于之前的水位线，就生成水位线并注入流中。

实现AssignerWithPunctuatedWatermarks接口，重写checkAndGetNextWatermark()方法，该方法会在针对每个事件的extractTimestamp()方法后立即调用，以此来决定是否生成一个新的水位线，如果该方法返回一个非空并且大于之前值的水位线，就会将这个新的水位线发出。

下面将会实现一个简单的定点水位线分配器

```
public class MyPunctuatedAssigner implements AssignerWithPunctuatedWater
marks<UserBehavior> {
    // 定义1分钟的容忍间隔时间，即允许数据的最大乱序时间
    private long maxOutofOrderness = 60 * 1000;
    @Nullable
    @Override
    public Watermark checkAndGetNextWatermark(UserBehavior element, long
 extractedTimestamp) {
        // 如果读取数据的用户行为是购买，就生成水位线
        if(element.action.equals("buy")){
           return new Watermark(extractedTimestamp - maxOutofOrderness);
        }else{
            // 不发出水位线
            return null;
        }
    }
    @Override
    public long extractTimestamp(UserBehavior element, long previousElem
entTimestamp) {
        return element.timestamp;
    }
}
```

迟到的数据

上文已经说过，现实中很难生成一个完美的水位线，水位线就是在延迟与准确性之前做的一种权衡。那么，如果生成的水位线过于紧迫，即水位线可能会大于后来数据的时间戳，这就意味着数据有延迟，关于延迟数据的处理，Flink提供了一些机制，具体如下：

- 直接将迟到的数据丢弃
- 将迟到的数据输出到单独的数据流中，即使用sideOutputLateData(new OutputTag<>()）实现侧输出
- 根据迟到的事件更新并发出结果

由于篇幅限制，关于迟到数据的具体处理在本文先不做太多的讨论，在后续的文章中会对其详细进行说明。

总结

本文从Flink的时间语义开始说起，详细介绍了三种时间语义的概念、特点及使用方式，接着对Flink处理乱序数据的一种机制—水位线进行详细说明，主要描述了水位线的基本概念，传播方式、生成方式，并对其中的细节部分进行了图解，可以加深对水位线的理解。最后，简单说明了一下Flink对于迟到数据的处理方式。