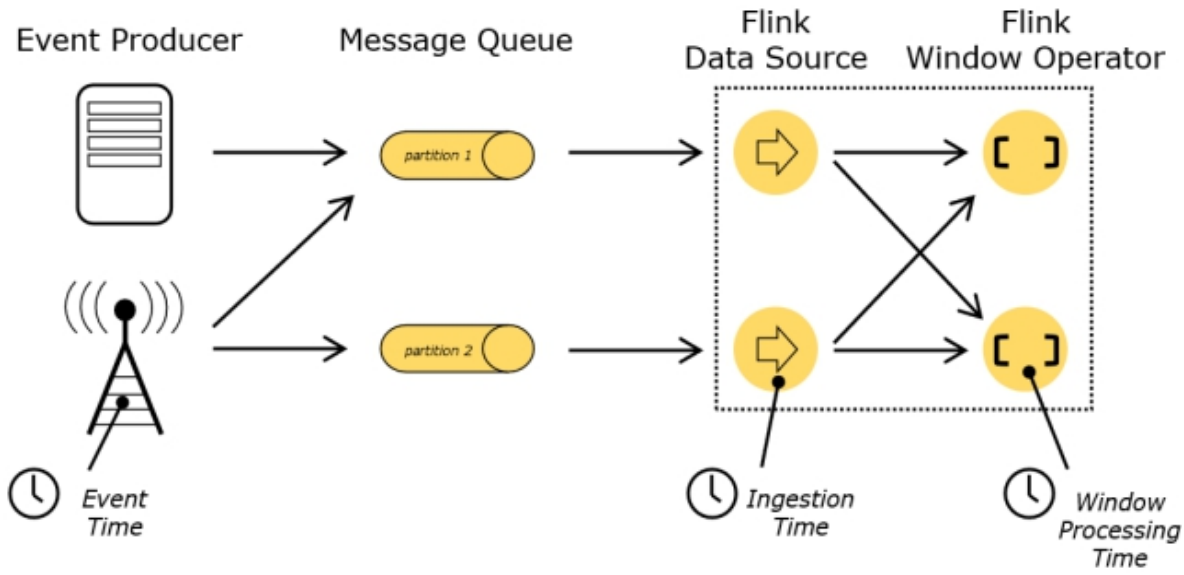
> 适度运动，保持充沛的精力

Flink之时间语义和Watermark

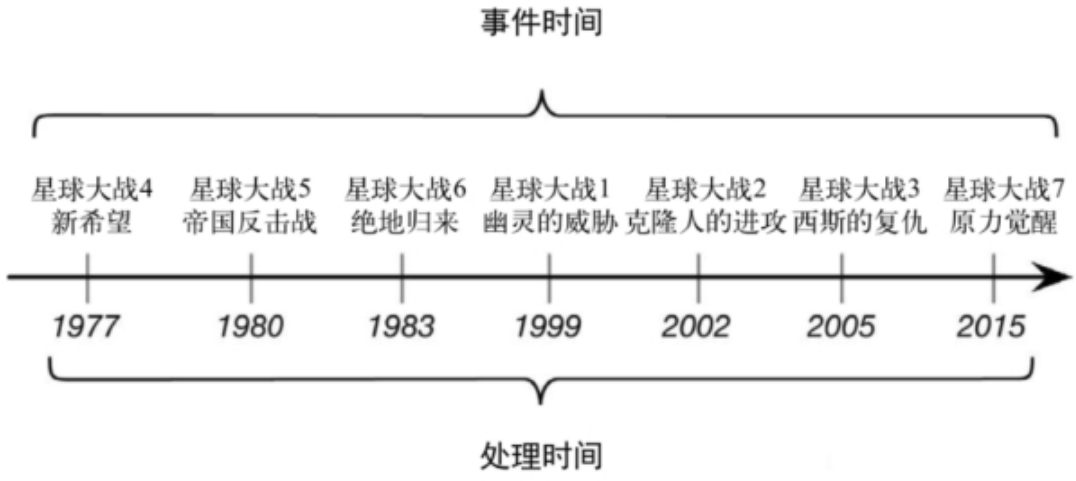
Flink中的时间语义

在Flink的流式处理中，会涉及到时间的不同概念，如下图所示：



- **Event Time**：是事件创建的时间。它通常由事件中的时间戳描述，例如采集的日志数据中，每一条日志都会记录自己的生成时间，Flink通过时间戳分配器访问事件时间戳。
- **Ingestion Time**：是数据进入Flink的时间。
- **Processing Time**：是每一个执行基于时间操作的算子的本地系统时间，与机器相关，默认的时间属性就是Processing Time。

一个例子——电影《星球大战》：




```
例如，一条日志进入Flink的时间为2017-11-12 10:00:00.123，
到达Window的系统时间为2017-11-12 10:00:01.234，日志的内容如下：
2017-11-02 18:37:15.624 INFO Fail over to rm2
```

对于业务来说，要统计1min内的故障日志个数，哪个时间是最有意义的？—— **eventTime**，因为我们要根据日志的生成时间进行统计。

EventTime的引入

> **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。

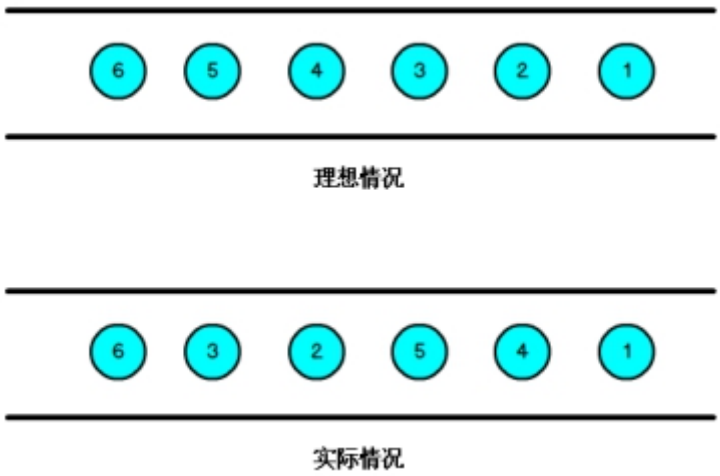
如果要使用EventTime，那么需要引入EventTime的时间属性，引入方式如下所示：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
// 从调用时刻开始给env创建的每一个stream追加时间特征
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
```

Watermark

> 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、分布式等原因，导致乱序的产生，所谓乱序**，就是指Flink接收到的事件的先后顺序不是严格按照事件的Event Time顺序排列的。【有问题都可以私聊我WX：focusbigdata，或者关注我的公众号：FocusBigData，注意大小写】

【Watermark就是处理乱序数据的机制！！】

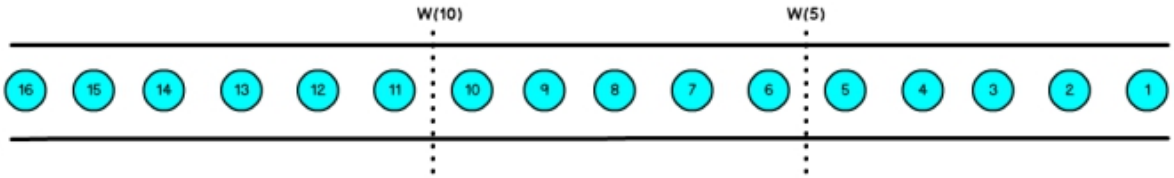


那么此时出现一个问题，一旦出现乱序，如果只根据eventTime决定window的运行，我们**不能明确数据是否全部到位，但又不能无限期的等下去**，此时必须要有个机制来**保证一个特定的时间后，必须触发window去进行计算了**，这个特别的机制，就是Watermark。

- Watermark是一种衡量Event Time进展的机制。
- 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。
- 数据流中的Watermark用于表示timestamp小于Watermark的数据，都已经到达了，因此，window的执行也是由Watermark触发的。
- Watermark可以理解成一个**延迟触发机制**，我们可以设置Watermark的延时时长t，每次系统会校验已经到达的数据中最大的maxEventTime，**然后认定eventTime小于maxEventTime - t的所有数据都已经到达**，**如果有窗口的停止时间等于maxEventTime – t**，那么这个窗口被触发执行。

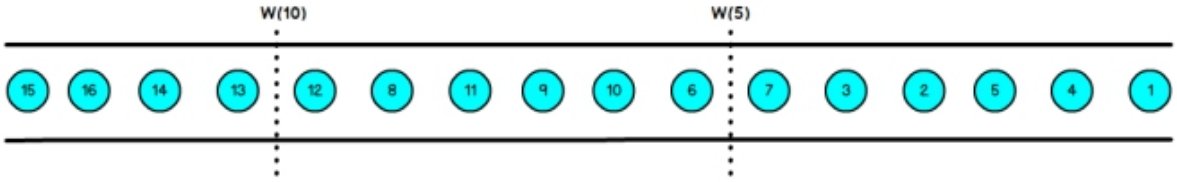
【认定Watermark之前的数据都是到达的！！！保证顺序！！！】

有序流的Watermarker如下图所示：（Watermark设置为0）



有序数据的Watermark

乱序流的Watermarker如下图所示：（Watermark设置为2）



当Flink接收到数据时，会按照一定的规则去生成Watermark，这条**Watermark就等于当前所有到达数据中的maxEventTime - 延迟时长**，也就是说，**Watermark是基于数据携带的时间戳生成的**，一旦Watermark比当前未触发的窗口的停止时间要晚，那么就会触发相应窗口的执行。由于event time是由数据携带的，因此，**如果运行过程中无法获取新的数据，那么没有被触发的窗口将永远都不被触发。**

上图中，我们设置的允许最大延迟到达时间为2s，所以时间戳为7s的事件对应的Watermark是5s，时间戳为12s的事件的Watermark是10s，如果我们的窗口1是1s~5s，窗口2是6s~10s，那么时间戳为7s的事件到达时的Watermarker恰好触发窗口1，时间戳为12s的事件到达时的Watermark恰好触发窗口2。

Watermark 就是触发前一窗口的“关窗时间”，一旦触发关门那么以当前时刻为准在窗口范围内的所有所有数据都会收入窗中。**只要没有达到水位那么不管现实中的时间推进了多久都不会触发关窗。**

WaterMark引入

```
// 其实就是定义时间戳的产生方式【这里就是取事件时间】
// 还有延迟的时间就行了
dataStream.assignTimestampsAndWatermarks( new
BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.milliseconds(1000))
{
  override def extractTimestamp(element: SensorReading): Long = {
    element.timestamp * 1000
  }
} )
```

Event Time的使用一定要**指定数据源中的时间戳。**否则程序无法知道事件的事件时间是什么(数据源里的数据没有时间戳的话，就只能使用Processing Time了)。

我们看到上面的例子中创建了一个看起来有点复杂的类，这个类实现的其实就是分配时间戳的接口。Flink暴露了TimestampAssigner接口供我们实现，使我们可以自定义如何从事件数据中抽取时间戳。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 从调用时刻开始给env创建的每一个stream追加时间特性
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

val readings: DataStream[SensorReading] = env
.addSource(new SensorSource)
.assignTimestampsAndWatermarks(new MyAssigner())

// MyAssigner就是你自己定义产生Watermark的方式
//  一共有两种方式，下面开始介绍
```

Assigner with periodic watermarks

> 周期性的生成watermark：系统会周期性的将watermark插入到流中(水位线也是一种特殊的事件!)。默认周期是200毫秒。可以使用ExecutionConfig.setAutoWatermarkInterval()方法进行设置。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

// 每隔5秒产生一个watermark
env.getConfig.setAutoWatermarkInterval(5000)
```

产生watermark的逻辑：**每隔5秒钟，Flink会调用AssignerWithPeriodicWatermarks的getCurrentWatermark()方法。**如果方法返回一个时间戳大于之前水位的时间戳，新的watermark会被插入到流中。这个检查保证了水位线是**单调递增**的。如果方法返回的时间戳小于等于之前水位的时间戳，则不会产生新的watermark。【保证单调递增！】

例子，自定义一个周期性的时间戳抽取：

```
class PeriodicAssigner extends AssignerWithPeriodicWatermarks[SensorReading] {
val bound: Long = 60 * 1000 // 延时为1分钟
var maxTs: Long = Long.MinValue // 观察到的最大时间戳

override def getCurrentWatermark: Watermark = {
    new Watermark(maxTs - bound)
}

override def extractTimestamp(r: SensorReading, previousTS: Long) = {
    maxTs = maxTs.max(r.timestamp)
        zr.timestamp
}
}
```

一种简单的特殊情况是，如果我们事先得知数据流的时间戳是单调递增的，也就是说没有乱序，那我们可以使用**assignAscendingTimestamps**，这个方法会直接使用数据的时间戳生成watermark。【一般生产环境的数据都是乱序的，所以这个用的少】

```
val stream: DataStream[SensorReading] = ...
val withTimestampsAndWatermarks = stream
.assignAscendingTimestamps(e => e.timestamp)

>> result:  E(1), W(1), E(2), W(2), ...
```

而对于乱序数据流，如果我们能大致估算出数据流中的事件的最大延迟时间，就可以使用如下代码：

```
val stream: DataStream[SensorReading] = ...
val withTimestampsAndWatermarks = stream.assignTimestampsAndWatermarks(
new SensorTimeAssigner
)

class SensorTimeAssigner extends
BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.seconds(5)) {
// 抽取时间戳
override def extractTimestamp(r: SensorReading): Long = r.timestamp
}
```

Assigner with punctuated watermarks

> 间断式地生成watermark。和周期性生成的方式不同，这种方式不是固定时间的，而是可以根据需要对每条数据进行筛选和处理。直接上代码来举个例子，我们只给sensor_1的传感器的数据流插入watermark：

```
class PunctuatedAssigner extends AssignerWithPunctuatedWatermarks[SensorReading]
{
val bound: Long = 60 * 1000

override def checkAndGetNextWatermark(r: SensorReading, extractedTS: Long):
Watermark = {
if (r.id == "sensor_1") {
    new Watermark(extractedTS - bound)
} else {
    null
}
}
override def extractTimestamp(r: SensorReading, previousTS: Long): Long = {
    r.timestamp
}
}
```

EvnetTime在window中的使用

滚动窗口（TumblingEventTimeWindows）

```
def main(args: Array[String]): Unit = {
    //  环境
    val env: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.setParallelism(1)

    val dstream: DataStream[String] = env.socketTextStream("localhost",7777)

    val textWithTsDstream: DataStream[(String, Long, Int)] = dstream.map { text
=>
      val arr: Array[String] = text.split(" ")
      (arr(0), arr(1).toLong, 1)
    }
```

```
    val textWithEventTimeDstream: DataStream[(String, Long, Int)] =
textWithTsDstream.assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor[(String, Long, Int)]
(Time.milliseconds(1000)) {
      override def extractTimestamp(element: (String, Long, Int)): Long = {

       return  element._2
      }
    })

    val textKeyStream: KeyedStream[(String, Long, Int), Tuple] =
textWithEventTimeDstream.keyBy(0)
    textKeyStream.print("textkey:")

    val windowStream: WindowedStream[(String, Long, Int), Tuple, TimeWindow] =
textKeyStream.window(TumblingEventTimeWindows.of(Time.seconds(2)))

    val groupDstream: DataStream[mutable.HashSet[Long]] = windowStream.fold(new
mutable.HashSet[Long]()) { case (set, (key, ts, count)) =>
      set += ts
    }

    groupDstream.print("window::::").setParallelism(1)

    env.execute()
  }
}
```

> 结果是按照Event Time的时间窗口计算得出的，而无关系统的时间（包括输入的快慢）。!!!

滑动窗口（SlidingEventTimeWindows）

```
def main(args: Array[String]): Unit = {
  //  环境
  val env: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment
  env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
  env.setParallelism(1)

  val dstream: DataStream[String] = env.socketTextStream("localhost",7777)

  val textWithTsDstream: DataStream[(String, Long, Int)] = dstream.map { text =>
    val arr: Array[String] = text.split(" ")
    (arr(0), arr(1).toLong, 1)
  }
  val textWithEventTimeDstream: DataStream[(String, Long, Int)] =
textWithTsDstream.assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor[(String, Long, Int)]
(Time.milliseconds(1000)) {
    override def extractTimestamp(element: (String, Long, Int)): Long = {

     return  element._2
    }
  })

  val textKeyStream: KeyedStream[(String, Long, Int), Tuple] =
textWithEventTimeDstream.keyBy(0)
```

```
  textKeyStream.print("textkey:")

  val windowStream: WindowedStream[(String, Long, Int), Tuple, TimeWindow] =
textKeyStream.window(SlidingEventTimeWindows.of(Time.seconds(2),Time.millisecond
s(500)))

  val groupDstream: DataStream[mutable.HashSet[Long]] = windowStream.fold(new
mutable.HashSet[Long]()) { case (set, (key, ts, count)) =>
    set += ts
  }

  groupDstream.print("window::::").setParallelism(1)

  env.execute()
}
```

会话窗口（EventTimeSessionWindows）

> 相邻两次数据的EventTime的时间差超过指定的时间间隔就会触发执行。如果加入Watermark， 会在符合窗口触发的情况下进行延迟。到达延迟水位再进行窗口触发。

```
def main(args: Array[String]): Unit = {
    //  环境
    val env: StreamExecutionEnvironment =
StreamExecutionEnvironment.getExecutionEnvironment

    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
    env.setParallelism(1)

    val dstream: DataStream[String] = env.socketTextStream("localhost",7777)

    val textWithTsDstream: DataStream[(String, Long, Int)] = dstream.map { text
=>
      val arr: Array[String] = text.split(" ")
      (arr(0), arr(1).toLong, 1)
    }
    val textWithEventTimeDstream: DataStream[(String, Long, Int)] =
textWithTsDstream.assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor[(String, Long, Int)]
(Time.milliseconds(1000)) {
      override def extractTimestamp(element: (String, Long, Int)): Long = {

       return  element._2
      }
    })

    val textKeyStream: KeyedStream[(String, Long, Int), Tuple] =
textWithEventTimeDstream.keyBy(0)
    textKeyStream.print("textkey:")

    val windowStream: WindowedStream[(String, Long, Int), Tuple, TimeWindow] =
textKeyStream.window(EventTimeSessionWindows.withGap(Time.milliseconds(500)) )

    windowStream.reduce((text1,text2)=>
      (  text1._1,0L,text1._3+text2._3)
    )  .map(_._3).print("windows:::").setParallelism(1)
```

```
    env.execute()

  }
```