Flink

第一章 Flink简介

认识Flink

Flink起源于Stratosphere项目，Stratosphere是在2010~2014年由3所地处柏林的大学和欧洲的一些其他的大学共同进行的研究项目，2014年4月Stratosphere的代码被复制并捐赠给了Apache软件基金会，参加这个孵化项目的初始成员是Stratosphere系统的核心开发人员，2014年12月，Flink一跃成为Apache软件基金会的顶级项目。

在德语中，Flink一词表示快速和灵巧，项目采用一只松鼠的彩色图案作为logo，这不仅是因为松鼠具有快速和灵巧的特点，还因为柏林的松鼠有一种迷人的红棕色，而Flink的松鼠logo拥有可爱的尾巴，尾巴的颜色与Apache软件基金会的logo颜色相呼应，也就是说，这是一只Apache风格的松鼠。

Flink项目的理念是：“Apache Flink是`为分布式、高性能、随时可用以及准确的流处理应用程序打造的开源流处理框架`”。

Apache Flink是一个框架和分布式处理引擎，用于对无界和有界数据流进行有状态计算。Flink被设计在所有常见的集群环境中运行，以内存执行速度和任意规模来执行计算。

Flink的重要特点

事件驱动型(Event-driven)

事件驱动型应用是一类具有状态的应用，它从一个或多个事件流提取数据，并根据到来的事件触发计算、状态更新或其他外部动作。比较典型的就是以kafka为代表的消息队列几乎都是事件驱动型应用。

流与批的世界观

批处理的特点是`有界、持久、大量`，非常适合需要`访问全套记录`才能完成的计算工作，一般用于离线统计。

流处理的特点是`无界、实时`,无需针对整个数据集执行操作，而是对通过系统传输的每个数据项执行操作，一般`用于实时统计`。

在spark的世界观中，一切都是由批次组成的，离线数据是一个大批次，而实时数据是由一个一个无限的小批次组成的。

而在flink的世界观中，一切都是由流组成的，离线数据是有界限的流，实时数据是一个没有界限的流，这就是所谓的有界流和无界流。

无界数据流：无界数据流有一个开始但是没有结束，它们不会在生成时终止并提供数据，必须连续处理无界流，也就是说必须在获取后立即处理event。对于无界数据流我们无法等待所有数据都到达，因为输入是无界的，并且在任何时间点都不会完成。处理无界数据通常要求以特定顺序（例如事件发生的顺序）获取event，以便能够推断结果完整性。

有界数据流：有界数据流有明确定义的开始和结束，可以在执行任何计算之前通过获取所有数据来处理有界流，处理有界流不需要有序获取，因为可以始终对有界数据集进行排序，有界流的处理也称为批处理。

这种以流为世界观的架构，获得的最大好处就是具有极低的延迟。

分层api

最底层级的抽象仅仅提供了有状态流，它将通过过程函数（Process Function）被嵌入到 `DataStream API` 中。底层过程函数（Process Function） 与 DataStream API 相集成，使其可以对某些特定的操作进行底层的抽象，它允许用户可以自由地处理来自一个或多个数据流的事件，并使用一致的容错的状态。除此之外，用户可以注册事件时间并处理时间回调，从而使程序可以处理复杂的计算。

实际上，大多数应用并不需要上述的底层抽象，而是针对核心API（Core APIs） 进行编程，比如DataStream API（有界或无界流数据）以及DataSet API（有界数据集）。这些API为数据处理提供了通用的构建模块，比如由用户定义的多种形式的转换（transformations），连接（joins），聚合（aggregations），窗口操作（windows）等等。DataSet API 为有界数据集提供了额外的支持，例如循环与迭代。这些API处理的数据类型以类（classes）的形式由各自的编程语言所表示。

Table API 是 `以表为中心的声明式编程`，其中表可能会动态变化（在表达流数据时）。Table API遵循（扩展的）关系模型：表有二维数据结构（schema）（类似于关系数据库中的表），同时API提供可比较的操作，例如select、project、join、group-by、aggregate等。Table API程序声明式地定义了什么逻辑操作应该执行，而不是准确地确定这些操作代码的看上去如何。

尽管Table API可以通过多种类型的用户自定义函数（UDF）进行扩展，其仍不如核心API更具表达能力，但是使用起来却更加简洁（代码量更少）。除此之外，Table API程序在执行之前会经过内置优化器进行优化。

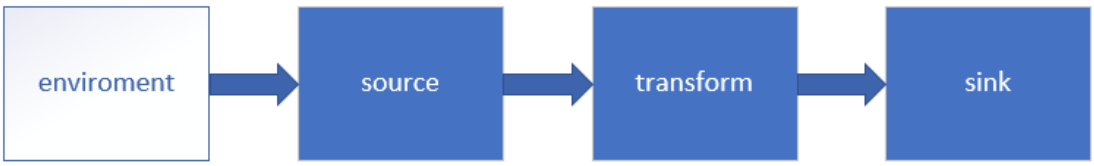
你可以在表与 DataStream/DataSet 之间无缝切换，以允许程序将 Table API 与 DataStream 以及 DataSet 混合使用。

`Flink提供的最高层级的抽象是 SQL` 。这一层抽象在语法与表达能力上与 Table API 类似，但是是以SQL查询表达式的形式表现程序。 `SQL抽象与Table API交互密切，同时SQL查询可以直接在Table API定义的表上执行` 。

目前Flink作为批处理还不是主流，不如Spark成熟，所以DataSet使用的并不是很多。Flink Table API和Flink SQL也并不完善，大多都由各大厂商自己定制。所以我们主要学习DataStream API的使用。实际上Flink作为最接近Google DataFlow模型的实现，是流批统一的观点，所以基本上使用DataStream就可以了。

第二章 DataStreamAPI

Flink流处理过程如下：



Environment

getExecutionEnvironment

创建一个执行环境，表示当前执行程序的上下文。如果程序是独立调用的，则此方法返回本地执行环境；如果从命令行客户端调用程序以提交到集群，则此方法返回此集群的执行环境，也就是说，getExecutionEnvironment会根 `据查询运行的方式决定返回什么样的运行环境` ，是 `最常用` 的一种创建执行环境的方式。

```
val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
env.setParallelism(1)
```

如果没有设置并行度，会以flink-conf.yaml中的配置为准，默认是1。

createLocalEnvironment

返回本地执行环境，需要在调用时指定默认的并行度。

```
val env = StreamExecutionEnvironment.createLocalEnvironment(1)
```

createRemoteEnvironment

返回集群执行环境，将Jar提交到远程服务器。需要在调用时指定JobManager的IP和端口号，并指定要在集群中运行的Jar包。

```
val env = ExecutionEnvironment.createRemoteEnvironment("jobmanage-hostname",
6123,"YOURPATH//wordcount.jar")
```

Source

从集合读取数据

```
case class SensorReading(id: String, timestamp: Long, temperature: Double)

object Source {
   def main(args: Array[String]): Unit = {
      val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment

      env.setParallelism(1)
      // 从集合中读取数据
      val stream1: DataStream[SensorReading] = env.fromCollection(List(
         SensorReading("sensor_1", 1547718199, 35.8),
         SensorReading("sensor_6", 1547718201, 15.4),
         SensorReading("sensor_7", 1547718202, 6.7),
         SensorReading("sensor_10", 1547718205, 38.1)
      ))
      stream1.print("stream1:")

      env.execute()
   }
}
```

从文件读取数据

```
val stream2 = env.readTextFile("YOUR_FILE_PATH")
```

以Kafka消息队列的数据作为来源

```
object Source {
   def main(args: Array[String]): Unit = {
      val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
      env.setParallelism(1)

      // 集成Kafka（从kafka中读取数据）
```

```
        val properties: Properties = new Properties()
        properties.put("bootstrap.servers", "master:9092")

        val kafkaDS: DataStream[String] = env.addSource(new FlinkKafkaConsumer[String](
           "lwPigKing",
           new SimpleStringSchema(),
           properties
        ))

        env.execute()
    }
}
```

自定义Source

继承SourceFunction

```
package Tutorial.DataStreamAPI

import org.apache.flink.streaming.api.functions.source.SourceFunction

import java.util.Calendar
import scala.util.Random

/**
 * Copyright (c) 2020-2030 尚硅谷 All Rights Reserved
 *
 * Project:  FlinkTutorial
 *
 * Created by  wushengran
 */
case class Event(user: String, url: String, timestamp: Long)

class ClickSource extends SourceFunction[Event]{
   // 标志位
   var running = true

   override def run(ctx: SourceFunction.SourceContext[Event]): Unit = {
      // 随机数生成器
      val random = new Random()
      // 定义数据随机选择的范围
      val users = Array("Mary", "Alice", "Bob", "Cary")
      val urls = Array("./home", "./cart", "./fav", "./prod?id=1", "./prod?id=2", "./prod?id=3")

      // 用标志位作为循环判断条件，不停地发出数据
      while (running){
         val event = Event(users(random.nextInt(users.length)), urls(random.nextInt(urls.length)),
Calendar.getInstance.getTimeInMillis)
//        // 为要发送的数据分配时间戳
//        ctx.collectWithTimestamp(event, event.timestamp)
//
//        // 向下游直接发送水位线
//        ctx.emitWatermark(new Watermark(event.timestamp - 1L))

         // 调用ctx的方法向下游发送数据
         ctx.collect(event)

         // 每隔1s发送一条数据
```

```
            Thread.sleep(1000)
        }
    }

    override def cancel(): Unit = running = false
}

```

```
package Tutorial.DataStreamAPI

import org.apache.flink.streaming.api.functions.source.SourceFunction

import scala.collection.immutable
import scala.util.Random
import scala.util.Random.nextGaussian

/**
 * Project:  BigDataCode
 * Create date:  2023/6/24
 * Created by fujiahao
 */

/**
 * 自定义Source
 * 需要继承SourceFunction
 */

class MySensorSource extends SourceFunction[SensorReading] {

    // flag:表示数据源是否还在正常运行
    var running: Boolean = true

    override def run(sourceContext: SourceFunction.SourceContext[SensorReading]): Unit = {

        // 初始化一个随机数发生器
        val random: Random = new Random()

        var curTemp: immutable.IndexedSeq[(String, Double)] = 1.to(10).map(
            i => ("sensor_" + i, 65 + random.nextGaussian() * 20)
        )

        while (running) {
            // 更新温度值
            curTemp = curTemp.map(
                t => (t._1, t._2 + random.nextGaussian() )
            )

            // 获取时间戳
            val curTime: Long = System.currentTimeMillis()

            curTemp.foreach(
                t => sourceContext.collect(SensorReading(t._1, curTime, t._2))
            )

            Thread.sleep(100)
        }
    }
```

```
50
51      override def cancel(): Unit = {
52        running = false
53      }
54    }
```

Transform

map

```
1   val stream: DataStream[Int] = env.fromElements(1, 2, 3, 4)
2   val streamMap: DataStream[Int] = stream.map(_ * 2)
3   streamMap.print()
```

flatMap

```
1   val stream: DataStream[String] = env.fromCollection(
2         List("Hello Hadoop", "Hello Flink", "Hello Spark")
3   )
4   val streamFlatMap: DataStream[String] = stream.flatMap(_.split(" "))
5   streamFlatMap.print()
```

filter

```
1   val stream: DataStream[Int] = env.fromElements(1, 2, 3, 4)
2   val streamFilter: DataStream[Int] = stream.filter(_ == 1)
3   streamFilter.print()
```

keyBy

DataStream -> KeyedStream(逻辑地讲一个流拆分成不相交的分区，每个分区`包含具有相同key元素`)

```
1   val stream: DataStream[String] = env.fromElements(
2         "Hello Spark",
3         "Hello Hadoop",
4         "Hello Flink"
5   )
6   val streamFlatMap: DataStream[String] = stream.flatMap(_.split(" "))
7   val streamMap: DataStream[(String, Int)] = streamFlatMap.map(word => (word, 1))
8   val value: KeyedStream[(String, Int), String] = streamMap.keyBy(_._1)
9   value.print()
```

Rolling Aggregation

滚动聚合算子，这些算子可以`针对KeyedStream的每一个支流做聚合`

sum()、min()、max()、minBy()、maxBy()

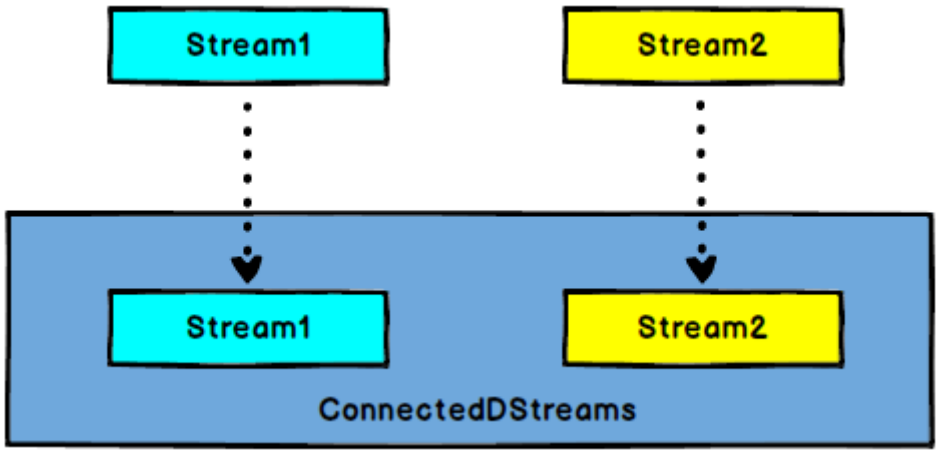
```
val stream: DataStream[String] = env.fromElements(
      "Hello Spark",
      "Hello Hadoop",
      "Hello Flink"
)
val streamFlatMap: DataStream[String] = stream.flatMap(_.split(" "))
val streamMap: DataStream[(String, Int)] = streamFlatMap.map(word => (word, 1))
val value: DataStream[(String, Int)] = streamMap.keyBy(_._1).sum(1)
value.print()
```

Reduce

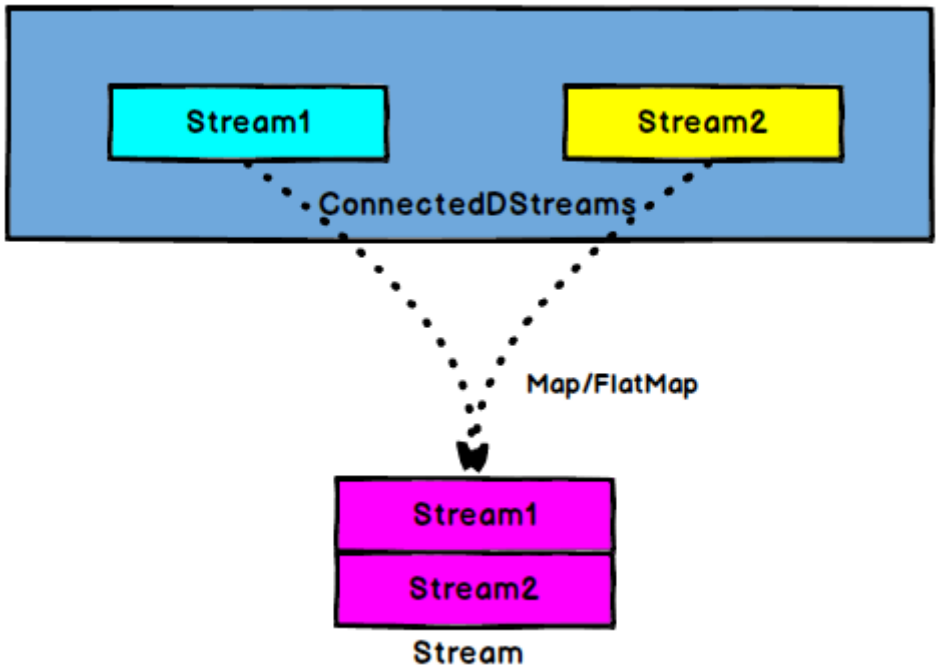
`KeyedStream → DataStream`：一个分组数据流的聚合操作，合并当前的元素和上次聚合的结果，产生一个新的值，返回的流中包含每一次聚合的结果，而不是只返回最后一次聚合的最终结果。

```
val stream2 = env.readTextFile("YOUR_PATH\\sensor.txt")
  .map( data => {
      val dataArray = data.split(",")
      SensorReading(dataArray(0).trim, dataArray(1).trim.toLong, dataArray(2).trim.toDouble)
  })
  .keyBy("id")
  .reduce( (x, y) => SensorReading(x.id, x.timestamp + 1, y.temperature) )
```

Connect和CoMap



`Connect`： `DataStream,DataStream → ConnectedStreams`：连接两个保持他们类型的数据流，两个数据流被Connect之后，只是被放在了一个同一个流中，内部依然保持各自的数据和形式不发生任何变化，两个流相互独立。

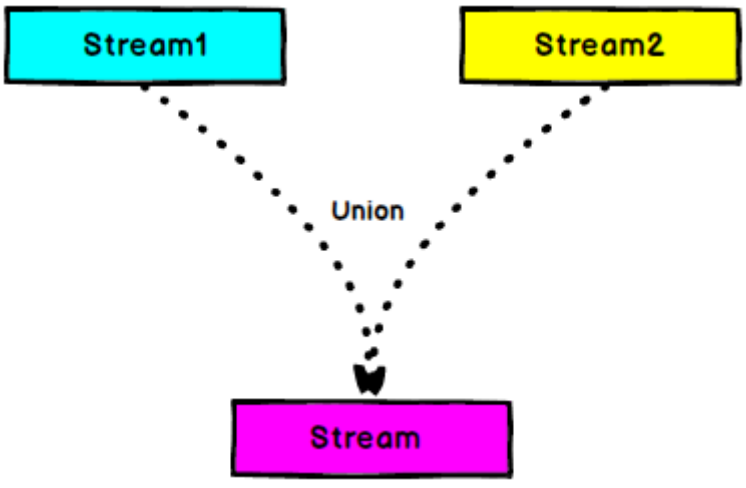


`CoMap`：`ConnectedStreams → DataStream`：作用于ConnectedStreams上，功能与map和flatMap一样，对ConnectedStreams中的每一个Stream分别进行map和flatMap处理。

`Connect如果和CoMap一起使用，得继承CoMapFunction。`

```
val intStream: DataStream[Int] = env.fromElements(1, 2, 3, 4, 5)
val stringStream: DataStream[String] = env.fromElements("A", "B", "C", "D", "E")
val connectedStream: ConnectedStreams[Int, String] = intStream.connect(stringStream)
val result: DataStream[String] = connectedStream.map(new CoMapFunction[Int, String, String] {
      override def map1(in1: Int): String = "Number: " + in1.toString
    override def map2(in2: String): String = "String: " + in2.toLowerCase
})
result.print()
```

Union



合流操作DataStream -> DataStream：对两个或者两个以上的DataStream进行Union操作
产生一个包含所有DataStream元素的新DataStream。

```
val intStream: DataStream[Int] = env.fromElements(1, 2, 3, 4, 5)
val intStream2: DataStream[Int] = env.fromElements(6, 7, 8, 9, 10)
val value: DataStream[Int] = intStream.union(intStream2)
value.print()
```

Connect和Union的区别

`Union之前两个流的类型必须一样`；`Connect可以不一样，在之后的CoMap中再去调整`
`Connect只能操作两个流；Union可以操作多个`

支持的数据类型

Flink流应用程序处理的是以数据对象表示的事件流。所以在Flink内部，我们需要能够处理这些对象。它们需要被序列化和反序列化，以便通过网络传送它们；或者从状态后端、检查点和保存点读取它们。为了有效地做到这一点，Flink需要明确知道应用程序所处理的数据类型。Flink使用类型信息的概念来表示数据类型，并为每个数据类型生成特定的序列化器、反序列化器和比较器。

Flink还具有一个类型提取系统，该系统分析函数的输入和返回类型，以自动获取类型信息，从而获得序列化器和反序列化器。但是，在某些情况下，例如lambda函数或泛型类型，需要显式地提供类型信息，才能使应用程序正常工作或提高其性能。

Flink支持Java和Scala中所有常见数据类型。使用最广泛的类型有以下几种。

基础数据类型

Flink支持所有的Java和Scala基础数据类型，Int, Double, Long, String..

```
val numbers: DataStream[Long] = env.fromElements(1L, 2L, 3L, 4L)
val value: DataStream[Long] = numbers.map(_ + 1)
value.print()
```

Java和Scalau元组

```
val persons: DataStream[(String, Int)] = env.fromElements(
     ("Adam", 17),
     ("Sarah", 23)
)
val value: DataStream[(String, Int)] = persons.filter(_._2 > 18)
value.print()
```

Scala样例类

```
case class Person(name: String, age: Int)
val persons: DataStream[Person] = env.fromElements(
      Person("Adam", 17),
      Person("Sarah", 23)
)
val value: DataStream[Person] = persons.filter(_.age > 18)
value.print()
```

Java简单对象

```java
public class Person {
      public String name;
      public int age;
      public Person() {}
      public Person(String name, int age) {
            this.name = name;
            this.age = age;
      }
}
```

```scala
val person: DataStream[Person] = env.fromElements(
      new Person("Alex", 42),
      new Person("Wendy", 23)
)
person.print()
```

其他

Flink对Java和Scala中的一些特殊目的的类型也都是支持的，比如Java的ArrayList，HashMap，Enum等等。

实现UDF函数

函数类

Flink暴露了所有udf函数的接口(实现方式为接口或者抽象类)。例如MapFunction, FilterFunction, ProcessFunction等等。

```scala
  val word: DataStream[String] = env.fromElements(
        "hello spark",
        "hello hadoop",
        "hello flink"
  )
  val wordFilter: DataStream[String] = word.filter(new myFilter)
  wordFilter.print()


  class myFilter extends FilterFunction[String] {
        override def filter(value: String): Boolean = value.contains("flink")
  }
```

还可以通过实现匿名类的方法

```scala
val word: DataStream[String] = env.fromElements("hello spark", "hello hadoop", "hello flink")
val value1: DataStream[String] = word.filter(new FilterFunction[String] {
      override def filter(value: String): Boolean = value.contains("hadoop")
})
value1.print()
```

还可以直接当做参数传入

```
val word: DataStream[String] = env.fromElements(
      "hello spark",
      "hello hadoop",
      "hello flink"
)
val value: DataStream[String] = word.filter(new keywordFilter("flink"))
value.print()


class keywordFilter(keyword: String) extends FilterFunction[String] {
      override def filter(value: String): Boolean = value.contains(keyword)
}
```

匿名函数

```
val word: DataStream[String] = env.fromElements(
      "hello spark",
      "hello hadoop",
      "hello flink"
)
word.filter(_.contains("flink")).print()
```

富函数

富函数是DataStream API提供的一个函数类的接口，所有Flink函数类都有其Rich版本。它与常规函数的不同在于，可以获取运行环境的上下文，并拥有一些生命周期方法，所以可以实现更复杂的功能。

Rich Function有一个生命周期的概念。典型的生命周期方法有：

open()方法是rich function的初始化方法，当一个算子例如map或者filter被调用之前open()会被调用。

close()方法是生命周期中的最后一个调用的方法，做一些清理工作。

getRuntimeContext()方法提供了函数的RuntimeContext的一些信息，例如函数执行的并行度，任务的名字，以及state状态。

```
class MyFlatMap extends RichFlatMapFunction[Int, (Int, Int)] {
      var subTaskIndex = 0

      override def open(parameters: Configuration): Unit = {
            subTaskIndex = getRuntimeContext.getIndexOfThisSubtask
      }

      override def flatMap(value: Int, out: Collector[(Int, Int)]): Unit = {
         if (value % 2 == subTaskIndex) {
            out.collect((subTaskIndex, value))
         }
      }

      override def close(): Unit = {

      }

}
```

Sink

Flink没有类似于spark中foreach方法，让用户进行迭代的操作。虽有对外的输出操作都要利用Sink完成。最后通过类似如下方式完成整个任务最终输出操作。

```
stream.addSink(new MySink(xxxx))
```

KafkaSink

```
val stream: DataStream[Int] = env.fromElements(1, 2, 3, 4)
val Res: DataStream[String] = stream.map(_.toString)
Res.addSink(new FlinkKafkaProducer[String](
      "master:9092",
      "lwPigKing",
      new SimpleStringSchema()
))
```

RedisSink

```
 val SensorDS: DataStream[SensorReading] = env.fromElements(
       SensorReading("1001", 32321423, 56.0),
       SensorReading("1002", 32321423, 56.0),
       SensorReading("1003", 32321423, 56.0)
 )

 val conf: FlinkJedisPoolConfig = new
 FlinkJedisPoolConfig.Builder().setHost("master").setPort(6379).build()

 SensorDS.addSink(new RedisSink[SensorReading](
       conf,
       new MyRedisMapper
 ))


 class MyRedisMapper extends RedisMapper[SensorReading] {
       override def getCommandDescription: RedisCommandDescription = {
             new RedisCommandDescription(RedisCommand.SET)
       }

       override def getKeyFromData(t: SensorReading): String = {
          t.temperature.toString
       }

       override def getValueFromData(t: SensorReading): String = {
          t.id
       }
 }
```

MySQLSink

```
 SensorDS.addSink(new MyJdbcSink)

 class MyJdbcSink extends RichSinkFunction[SensorReading]{
    var conn: Connection = _
    var insertStmt: PreparedStatement = _
    var updateStmt: PreparedStatement = _

```

```
    // open主要是创建连接
    override def open(parameters: Configuration): Unit = {
      conn = DriverManager.getConnection("jdbc:mysql://master:3306/lwPigKing", "root", "123456")
    }

    // 调用连接，执行sql
    override def invoke(value: SensorReading, context: SinkFunction.Context): Unit = {
      insertStmt = conn.prepareStatement("INSERT INTO temperatures (sensor, temp) VALUES (?, ?)")
      updateStmt = conn.prepareStatement("UPDATE temperatures SET temp = ? WHERE sensor = ?")
      updateStmt.setDouble(1, value.temperature)
      updateStmt.setString(2, value.id)
      updateStmt.execute()

      if (updateStmt.getUpdateCount == 0){
        insertStmt.setString(1, value.id)
        insertStmt.setDouble(2, value.temperature)
        insertStmt.execute()
      }
    }

    // 关闭连接
    override def close(): Unit = {
      insertStmt.close()
      updateStmt.close()
      conn.close()
    }
  }
```

ClickHouseSink

```
class ClickHouseSink(url: String, username: String, password: String) extends
RichSinkFunction[(String, Int)]{
  private var connection: ClickHouseConnection = _

  private val properties: Properties = new Properties()
  properties.put("username", username)
  properties.put("password", password)

  override def open(parameters: Configuration): Unit = {
    val dataSource: ClickHouseDataSource = new ClickHouseDataSource(url, properties)
    connection = dataSource.getConnection
  }

  override def invoke(value: (String, Int), context: SinkFunction.Context): Unit = {

    val statement: PreparedStatement = connection.prepareStatement(
      "insert into station_flow (station, flow) values (?, ?)"
    )

    statement.setString(1, value._1)
    statement.setInt(2, value._2)
    statement.executeUpdate()
    statement.close()
  }

  override def close(): Unit = {
    if (connection != null) {
```

```
            connection.close()
        }
    }

}
```

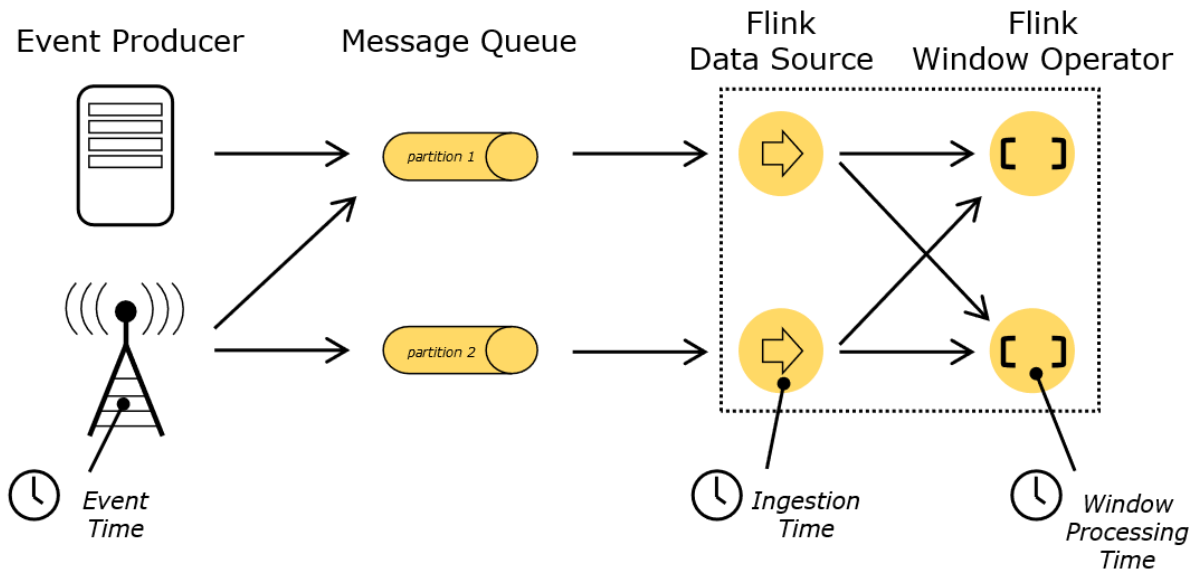
HBaseSink

```
class StationHBaseSink extends RichSinkFunction[(String, String, Long)]{
    var conn: Connection = null

    override def open(parameters: Configuration): Unit = {
        val config: conf.Configuration = HBaseConfiguration.create
        config.set(HConstants.ZOOKEEPER_QUORUM,"master,slave1,slave2")
        config.set(HConstants.ZOOKEEPER_CLIENT_PORT,"2181")
        conn = ConnectionFactory.createConnection(config)
    }

    override def invoke(value: (String, String, Long), context: SinkFunction.Context): Unit = {
        val tableName: String = "station_flow"
        val rowKey: String = value._1
        val station: String = value._2
        val flow: Long = value._3
        val table: Table = conn.getTable(TableName.valueOf(tableName))
        val put: Put = new Put(Bytes.toBytes(rowKey))
        put.addColumn(Bytes.toBytes("cf"), Bytes.toBytes("station"), Bytes.toBytes(station))
        put.addColumn(Bytes.toBytes("cf"), Bytes.toBytes("flow"), Bytes.toBytes(flow.toString))
        table.put(put)
    }

    override def close(): Unit = {
        if (conn != null) {
            conn.close()
        }
    }
}
```

第三章 TimeAndWindow

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。

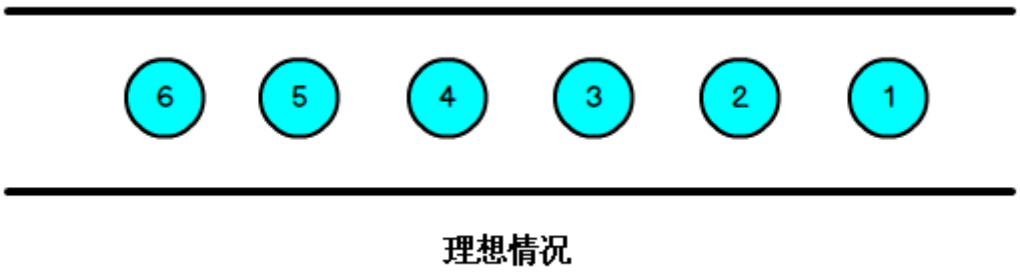
Ingestion Time：是数据进入Flink的时间。

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。

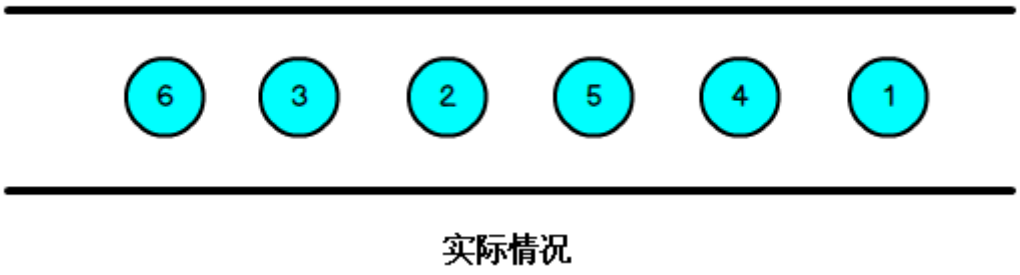
水位线Watermark

基本概念

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。



理想情况



实际情况

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。

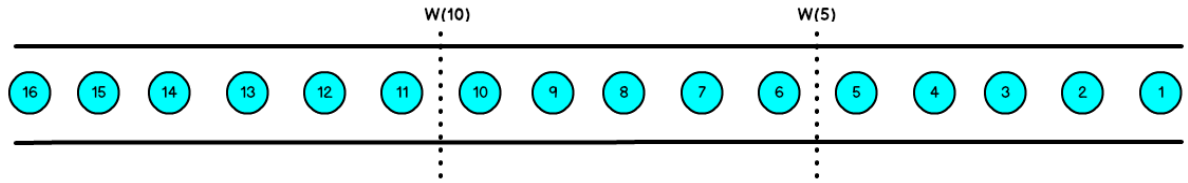
Watermark是一种衡量Event Time进展的机制。

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。

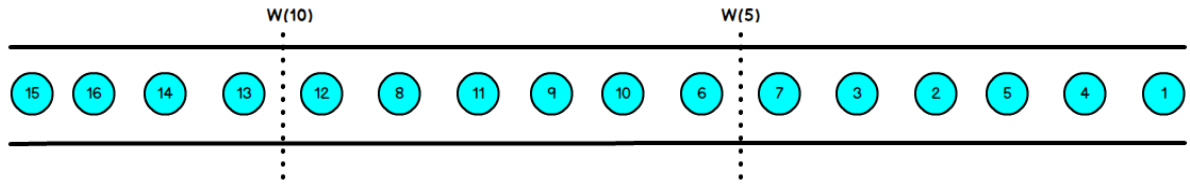
数据流中的Watermark用于表示timestamp小于Watermark的数据，都已经到达了，因此，window的执行也是由Watermark触发的。

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。

有序流的Watermarker如下图所示：（Watermark设置为0）



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。

水位线生成策略

水位线生成策略（Watermark Strategies）
在Flink的DataStreamAPI中，有一个单独用于生成水位线的方法：
assignTimestampsAndWatermarks()，它主要用来为流中的数据分配时间戳，并且生成水位线来指示事件时间。具体使用时，直接用DataStream调用该方法即可。

assignTimestampsAndWatermarks()方法需要传入一个水位线生成策略（WatermarkStrategy）作为参数。WatermarkStrategy中包含了一个时间戳分配器和一个水位线生成器。Flink提供了内置的水位线生成器（WatermarkTGenerator），不仅开箱即用简化了编程，而且也为我们自定义水位线策略提供了模板。这两个生成器可以通过调用WatermarkStrategy的静态辅助方法来创建。它们都是周期性生成水位线，分别对应着处理有序流和乱序流的场景。

有序流

对于有序流，主要特点就是时间戳单调增长，所以永远不会出现迟到数据的问题。这是周期性生成水位线的最简单的场景，直接调用WatermarkStrategy.forMonotonousTimestamps()方法即可。简单来说，就是直接拿当前最大的时间戳作为水位线就可以了。

```
val stream: DataStreamSource[Event] = env.addSource(new ClickSource)
stream.assignTimestampsAndWatermarks(
      WatermarkStrategy.forMonotonousTimestamps[Event]()
     .withTimestampAssigner(new SerializableTimestampAssigner[Event] {
           override def extractTimestamp(element: Event, recordTimestamp: Long): Long =
element.timestamp
     })
)
```

无序流

乱序流由于乱序流中需要等待迟到数据到齐，所以必须设置一个固定量的延迟时间（Fixed Amount of Late）。这时生成水位线的时间戳，就是当前数据流中最大的时间戳减去延迟的结果，相当于把表调慢，当前时钟会滞后于数据的最大时间戳。调用WatermarkStrategy.forBoundedOutOfOrderness()方法就可以实现。这个方法需要传入一个maxOutOfOrderness参数，表示最大乱序程度，它表示数据流中乱序数据时间戳的最大差值；如果能确定乱序程度，那么设置对应时间长度的延迟，就可以等到所有的乱序数据了。

```
stream.assignTimestampsAndWatermarks(
      WatermarkStrategy.forBoundedOutOfOrderness[Event](Duration.ofSeconds(5))
     .withTimestampAssigner(new SerializableTimestampAssigner[Event] {
           override def extractTimestamp(element: Event, recordTimestamp: Long): Long =
element.timestamp
     })
)
.print()
```

窗口Window API

Window概述

streaming流式计算是一种被设计用于处理无限数据集的数据处理引擎，而无限数据集是指一种不断增长的本质上无限的数据集，而window是一种切割无限数据为有限块进行处理的手段。

Window是无限数据流处理的核心，Window将一个无限的stream拆分成有限大小的”buckets”桶，我们可以在这些桶上做计算操作。

Window类型

Window可以分成两类：

CountWindow：按照指定的数据条数生成一个Window，与时间无关。

TimeWindow：按照时间生成Window。

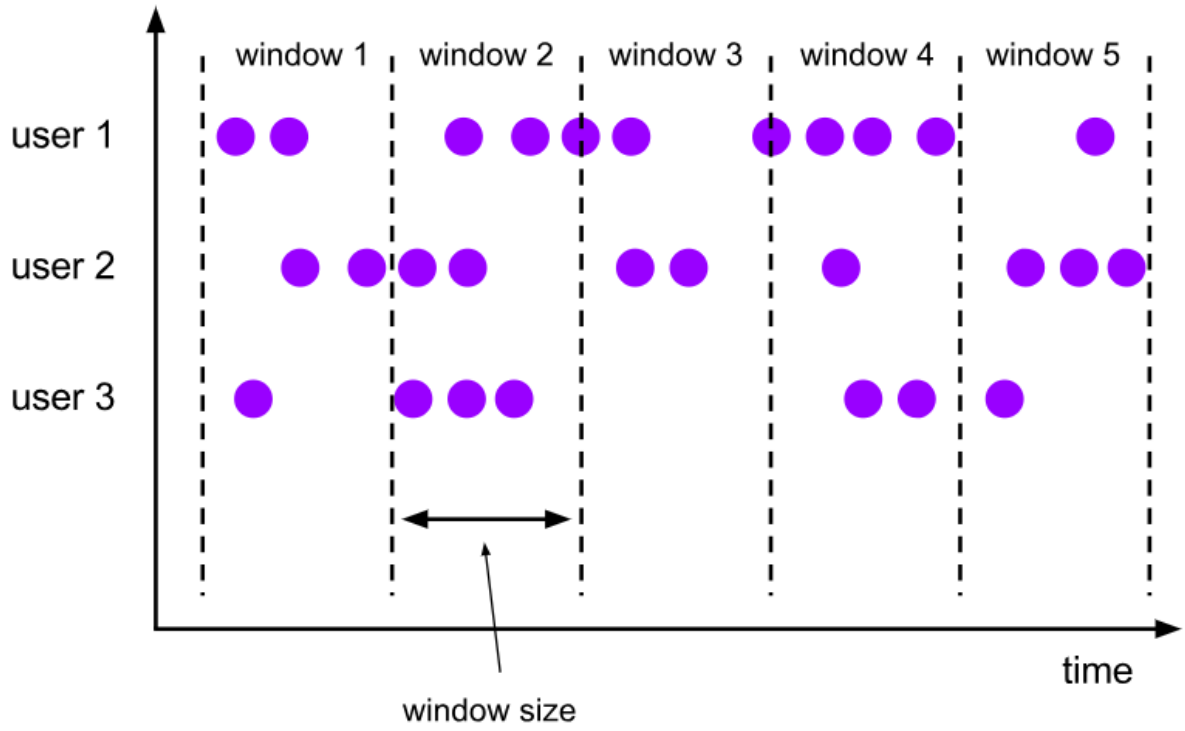
对于TimeWindow，可以根据窗口实现原理的不同分成三类：滚动窗口（Tumbling Window）、滑动窗口（Sliding Window）和会话窗口（Session Window）。

滚动窗口（Tumbling Windows）将数据依据固定的窗口长度对数据进行切片。

特点：时间对齐，窗口长度固定，没有重叠。

适用场景：适合做BI统计等（做每个时间段的聚合计算）。

滚动窗口分配器将每个元素分配到一个指定窗口大小的窗口中，滚动窗口有一个固定的大小，并且不会出现重叠。例如：如果你指定了一个5分钟大小的滚动窗口，窗口的创建如下图所示：



滑动窗口（Sliding Windows）

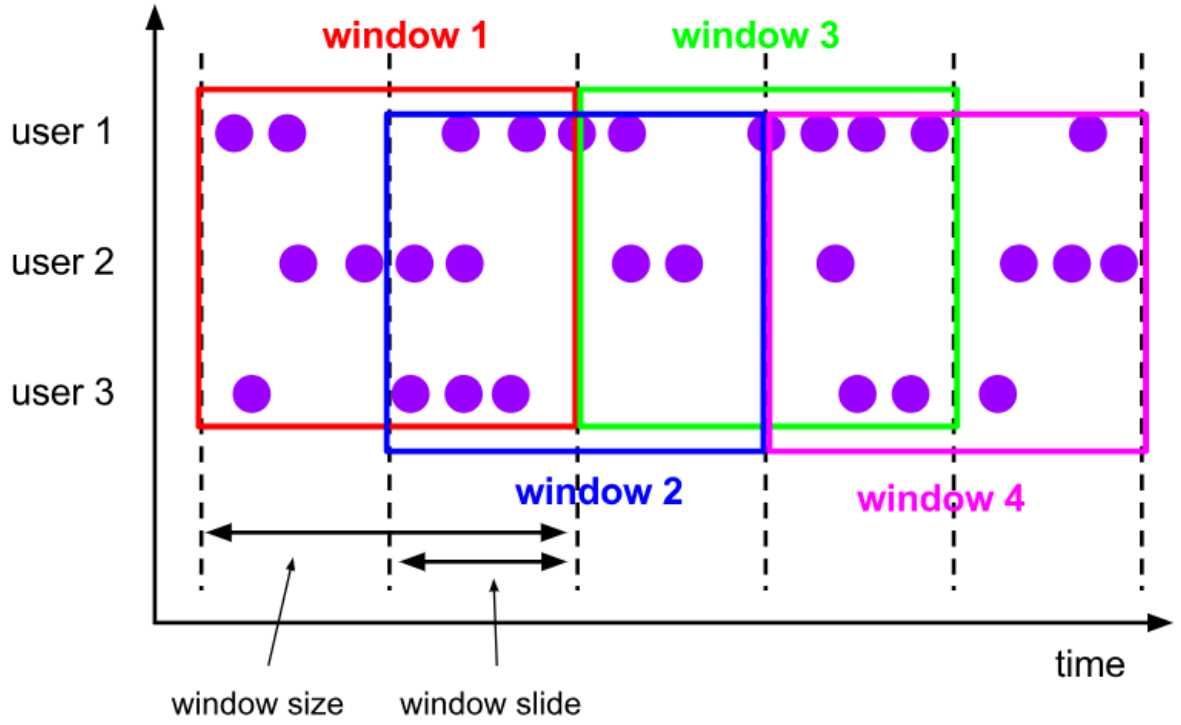
滑动窗口是固定窗口的更广义的一种形式，滑动窗口由固定的窗口长度和滑动间隔组成。

特点：时间对齐，窗口长度固定，可以有重叠。

适用场景：对最近一个时间段内的统计（求某接口最近5min的失败率来决定是否要报警）。

滑动窗口分配器将元素分配到固定长度的窗口中，与滚动窗口类似，窗口的大小由窗口大小参数来配置，另一个窗口滑动参数控制滑动窗口开始的频率。因此，滑动窗口如果滑动参数小于窗口大小的话，窗口是可以重叠的，在这种情况下元素会被分配到多个窗口中。

例如，你有10分钟的窗口和5分钟的滑动，那么每个窗口中5分钟的窗口里包含着上个10分钟产生的数据，如下图所示：

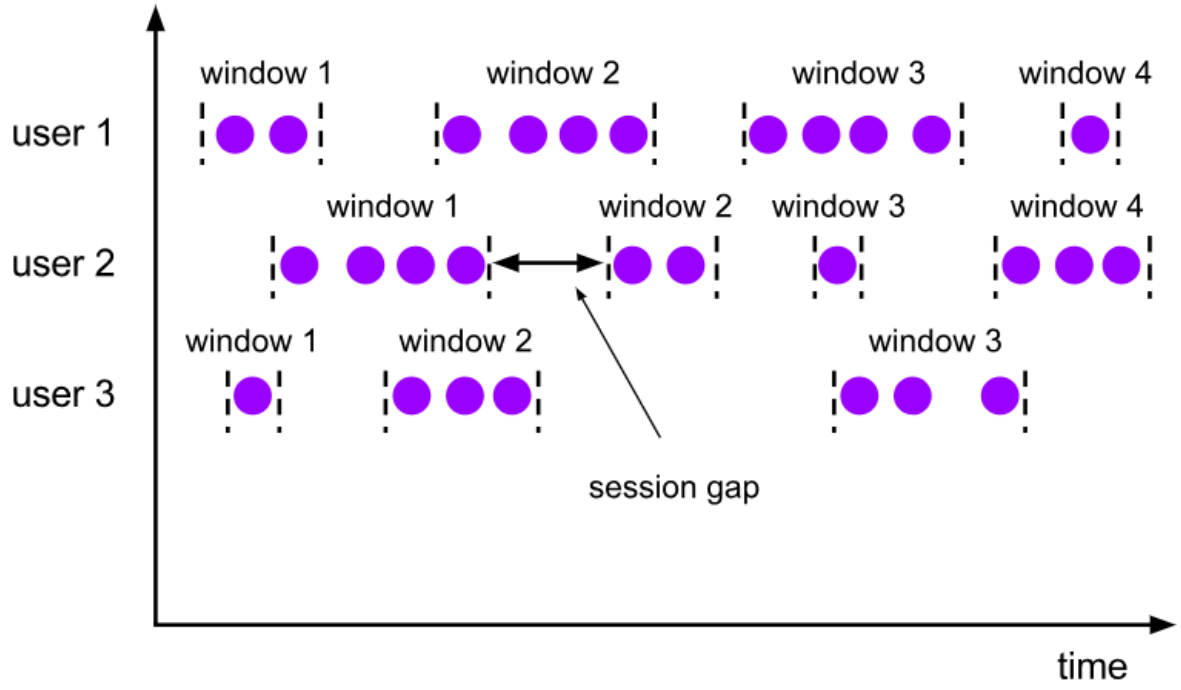


会话窗口（Session Windows）

由一系列事件组合一个指定时间长度的timeout间隙组成，类似于web应用的session，也就是一段时间没有接收到新数据就会生成新的窗口。

特点：时间无对齐。

session窗口分配器通过session动来对元素进行分组，session窗口跟滚动窗口和滑动窗口相比，不会有重叠和固定的开始时间和结束时间的情况，相反，当它在一个固定的时间周期内不再收到元素，即非活动间隔产生，那个这个窗口就会关闭。一个session窗口通过一个session间隔来配置，这个session间隔定义了非活跃周期的长度，当这个非活跃周期产生，那么当前的session将关闭并且后续的元素将被分配到新的session窗口中去。



窗口分配器

TimeWindow

TimeWindow是将指定时间范围内的所有数据组成一个window，一次对一个window里面的所有数据进行计算。

滚动窗口

```
dataStream.map(r => (r.id, r.temperature))
    .keyBy(_._1)
     .window(TumblingProcessingTimeWindows.of(Time.seconds(1)))
     .reduce((r1, r2) => (r1._1, r1._2.min(r2._2)))
```

滑动窗口

```
dataStream.map(r => (r.id, r.temperature))
    .keyBy(_._1)
    .window(SlidingProcessingTimeWindows.of(Time.seconds(15), Time.seconds(5)))
```

会话窗口

```
dataStream.map(r => (r.id, r.temperature))
    .keyBy(_._1)
    .window(ProcessingTimeSessionWindows.withGap(Time.seconds(10)))
```

CountWindow

CountWindow根据窗口中相同key元素的数量来触发执行，执行时只计算元素数量达到窗口大小的key对应的结果。
注意：CountWindow的window_size指的是相同key的元素的个数，不是输入的所有元素的总数。

滚动计数窗口

```
// 滚动计数窗口：只需传入参数size（表示窗口大小）
// 定义一个长度为5的滚动计数窗口，当窗口中元素数量达到5时，就会触发计算执行并关闭窗口
dataStream.map(r => (r.id, r.temperature))
    .keyBy(_._1)
     .countWindow(5)
     .reduce((r1, r2) => (r1._1, r1._2.max(r2._2)))
```

滑动计数窗口

```
// 滑动计数窗口：需传入参数size和slide
// 前者表示窗口大小，后者表示滑动步长
// 定义一个长度为10、滑动步长为3的滑动计数窗口。
// 每个窗口统计10个数据，每隔3个数据就统计输出一次结果。
dataStream.map(r => (r.id, r.temperature))
    .keyBy(_._1)
     .countWindow(10, 3)
```

窗口函数

定义了窗口分配器，我们只是知道了数据属于哪个窗口，可以将数据收集起来了；至于收集起来到底要做什么，其实还完全没有头绪。所以在窗口分配器之后，必须再接上一个定义窗口如何进行计算的操作，这就是所谓的“窗口函数”（window functions）。

经窗口分配器处理之后，数据可以分配到对应的窗口中，而数据流经过转换得到的数据类型是 WindowedStream。这个类型并不是 DataStream，所以并不能直接进行其他转换，而必须进一步调用窗口函数，对收集到的数据进行处理计算之后，才能最终再次得到 DataStream。

窗口函数定义了要对窗口中收集的数据做的计算操作，根据处理的方式可以分为两类：增量聚合函数和全窗口函数。

增量聚合函数

为了提高实时性，我们可以像 DataStream 的简单聚合一样，每来一条数据就立即进行计算，中间只要保持一个简单的聚合状态就可以了；区别只是在于不立即输出结果，而是要等到窗口结束时间。等到窗口到了结束时间需要输出计算结果的时候，我们只需要拿出之前聚合的状态直接输出，这无疑就大大提高了程序运行的效率和实时性。

典型的增量聚合函数有两个：ReduceFunction 和 AggregateFunction。

ReduceFunction

只要基于 WindowedStream 调用.reduce()方法，然后传入 ReduceFunction 作为参数，就可以指定以归约两个元素的方式去对窗口中数据进行聚合了。

```
env.addSource(new ClickSource)
    .assignAscendingTimestamps(_.timestamp)
    .map(r => (r.user, 1L))
    // 使用用户名对数据流进行分组
    .keyBy(_._1)
    // 设置5秒钟的滚动事件窗口
    .window(TumblingEventTimeWindows.of(Time.seconds(5)))
    // 保留第一个字段，针对第二个字段进行聚合
    .reduce((r1, r2) => (r1._1, r1._2 + r2._2))
    .print()
```

AggregateFunction

ReduceFunction 可以解决大多数归约聚合的问题，但是这个接口有一个限制，就是聚合状态的类型、输出结果的类型都必须和输入数据类型一样。为了更加灵活地处理窗口计算，Flink的Window API提供了更加一般化的aggregate()方法。直接基于 WindowedStream 调用 aggregate()方法，就可以定义更加灵活的窗口聚合操作。这个方法需要传入一个 AggregateFunction 的实现类作为参数。

AggregateFunction 可以看作是 ReduceFunction 的通用版本，这里有三种类型：输入类型（IN）、累加器类型（ACC）和输出类型（OUT）。输入类型 IN 就是输入流中元素的数据类型；累加器类型 ACC 则是我们进行聚合的中间状态类型；而输出类型当然就是最终计算结果的类型了。

AggregateFunction 接口中有四个方法：

createAccumulator()：创建一个累加器，这就是为聚合创建了一个初始状态，每个聚合任务只会调用一次。

add()：将输入的元素添加到累加器中。这就是基于聚合状态，对新来的数据进行进一步聚合的过程。方法传入两个参数：当前新到的数据 value，和当前的累加器 accumulator；返回一个新的累加器值，也就是对聚合状态进行更新。每条数据到来之后都会调用这个方法。

getResult()：从累加器中提取聚合的输出结果。也就是说，我们可以定义多个状态，然后再基于这些聚合的状态计算出一个结果进行输出。比如之前我们提到的计算平均值，就可以把 sum 和 count 作为状态放入累加器，而在调用这个方法时相除得到最终结果。这个方法只在窗口要输出结果时调用。

merge()：合并两个累加器，并将合并后的状态作为一个累加器返回。这个方法只在需要合并窗口的场景下才会被调用；最常见的合并窗口（Merging Window）的场景就是会话窗口（Session Windows）。

```
env.addSource(new ClickSource)
    .assignAscendingTimestamps(_.timestamp)
    // 通过为每条数据分配同样的key，来将数据发送到同一个分区
    .keyBy(_ => "key")
    .window(SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(2)))
    .aggregate(new AvgPv)
    .print()


class AvgPv extends AggregateFunction[Event, (Set[String], Double), Double] {

      // 创建空累加器，类型是元组，元组的第一个元素类型为Set（用来对用户名进行去重）
      // 第二个元素用来累加PV操作（每来一条数据就加一）
      override def createAccumulator(): (Set[String], Double) = (Set[String](), 0L)

      // 累加规则
      override def add(value: Event, accumulator: (Set[String], Double)): (Set[String], Double) =
{
        (accumulator._1 + value.user, accumulator._2 + 1L)
      }

      // 获取窗口关闭时向下游发送的结果
      // PV / UV
      override def getResult(accumulator: (Set[String], Double)): Double = {
         accumulator._2 / accumulator._1.size
      }

      // merge方法只有在事件时间的会话窗口时，才需要实现，这里无需实现
      override def merge(a: (Set[String], Double), b: (Set[String], Double)): (Set[String],
Double) = ???
}
```

全窗口函数

窗口操作中的另一大类就是全窗口函数。与增量聚合函数不同，全窗口函数需要先收集窗口中的数据，并在内部缓存起来，等到窗口要输出结果的时候再取出数据进行计算。在 Flink 中，全窗口函数也有两种：WindowFunction 和 ProcessWindowFunction。

WindowFunction

WindowFunction 字面上就是“窗口函数”，它其实是老版本的通用窗口函数接口。我们可以基于 WindowedStream 调用.apply()方法，传入一个 WindowFunction 的实现类。（基本不用）

ProcessWindowFunction

ProcessWindowFunction 是 Window API 中最底层的通用窗口函数接口。之所以说它“最底层”，是因为除了可以拿到窗口中的所有数据之外，ProcessWindowFunction 还可以获取到一个“上下文对象”（Context）。这个上下文对象非常强大，不仅能够获取窗口信息，还可以访问当前的时间和状态信息。这里的时间就包括了处理时间（processing time）和事件时间水位线（event time watermark）。这就使得 ProcessWindowFunction 更加灵活、功能更加丰富，可以认为是一个增强版的 WindowFunction。 具体使用跟 WindowFunction 非常类似，我们可以基于 WindowedStream 调用 process()方法，传入一个 ProcessWindowFunction 的实现类。

```
env.addSource(new ClickSource)
    .assignAscendingTimestamps(_.timestamp)
    .keyBy(_ => "key")
    .window(TumblingEventTimeWindows.of(Time.seconds(10)))
    .process(new UvCountByWindow)
    .print()

class UvCountByWindow extends ProcessWindowFunction[Event, String, String, TimeWindow] {
      override def process(key: String, context: Context, elements: Iterable[Event], out: Collector[String]): Unit = {
          // 初始化一个Set，用来对用户名进行去重
          var userSet: Set[String] = Set[String]()
          // 将所用用户名进行去重
          elements.foreach(userSet += _.user)
          // 结合窗口信息，包装输出内容
          val windowStart: Long = context.window.getStart
          val windowEnd: Long = context.window.getEnd
          out.collect("窗口：" + windowStart + "~~~~~~~~" + windowEnd +"的PV值是" + userSet.size)
      }
}
```

增量聚合和全窗口结合使用

增量聚合函数处理计算会更高效。增量聚合相当于把计算量“均摊”到了窗口收集数据的过程中，自然就会比全窗口聚合更加高效、输出更加实时。 而全窗口函数的优势在于提供了更多的信息，可以认为是更加“通用”的窗口操作，窗口计算更加灵活，功能更加强大。 所以在实际应用中，我们往往希望兼具这两者的优点，把它们结合在一起使用。Flink 的Window API 就给我们实现了这样的用法。

我们之前在调用 WindowedStream 的 reduce()和 aggregate()方法时，只是简单地直接传入了一个 ReduceFunction 或 AggregateFunction 进行增量聚合。除此之外，其实还可以传入第二个参数：一个全窗口函数，可以是 WindowFunction 或者 ProcessWindowFunction。

```
env.addSource(new ClickSource)
    .assignAscendingTimestamps(_.timestamp)
    // 使用url作为key对数据进行分区
    .keyBy(_.url)
    .window(SlidingEventTimeWindows.of(Time.minutes(1), Time.seconds(20)))
    // 第一个参数增量聚合函数：聚合操作
    // 第二个参数全窗口函数：输出结果
```

```
        // 全窗口函数的IN类型就是增量聚合函数的OUT类型
        .aggregate(new UrlViewCountAgg, new UrlViewCountResult)
        .print()


class UrlViewCountAgg extends AggregateFunction[Event, Long, Long] {
      override def createAccumulator(): Long = 0L

      override def add(value: Event, accumulator: Long): Long = accumulator + 1L

      override def getResult(accumulator: Long): Long = accumulator

      override def merge(a: Long, b: Long): Long = ???
}

class UrlViewCountResult extends ProcessWindowFunction[Long, UrlViewCount, String, TimeWindow] {
      override def process(key: String, context: Context, elements: Iterable[Long], out:
Collector[UrlViewCount]): Unit = {
          val start: Long = context.window.getStart
          val end: Long = context.window.getEnd
          val date: Date = new Date()
          out.collect(UrlViewCount(key, elements.iterator.next(), date))
      }
}

case class UrlViewCount(url: String, count: Long, period: Date)
```

其他API

对于一个窗口算子而言，窗口分配器和窗口函数是必不可少的。除此之外，Flink 还提供了其他一些可选的 API，让我们可以更加灵活地控制窗口行为。

触发器

触发器主要是用来控制窗口什么时候触发计算。所谓的“触发计算”，本质上就是执行窗口函数，所以可以认为是计算得到结果并输出的过程。基于 WindowedStream 调用 trigger()方法，就可以传入一个自定义的窗口触发器（Trigger）。

```
stream.keyBy(_.user)
      .window(TumblingEventTimeWindows.of(Time.minutes(1)))
      .trigger(new MyTrigger())

class MyTrigger extends Trigger[Any, Window] {
      // 窗口中每到来一个元素，都会调用这个方法
      override def onElement(t: Any, l: Long, w: Window, triggerContext: Trigger.TriggerContext):
TriggerResult = {

      }

      // 当注册的处理时间定时器触发时，将调用这个方法
      override def onProcessingTime(l: Long, w: Window, triggerContext: Trigger.TriggerContext):
TriggerResult = ???

      // 当注册的事件时间定时器触发时，将调用这个方法
```

```
15          override def onEventTime(l: Long, w: Window, triggerContext: Trigger.TriggerContext):
    TriggerResult = ???
16        // 当窗口关闭销毁时，调用这个方法，一般用来清楚自定义的状态
17          override def clear(w: Window, triggerContext: Trigger.TriggerContext): Unit = ???
18      }
```

移除器

移除器主要用来定义移除某些数据的逻辑。基于 WindowedStream 调用.evictor()方法，就可以传入一个自定义的移除器（Evictor）。Evictor 是一个接口，不同的窗口类型都有各自预实现的移除器。

```
1   stream.keyBy(_.user)
2           .window(TumblingEventTimeWindows.of(Time.minutes(1)))
3           .evictor(new Evictor[Event, Window] {
4               // 定义执行窗口函数之前的移除数据操作
5               override def evictBefore(iterable: lang.Iterable[TimestampedValue[Event]], i: Int, w:
    Window, evictorContext: Evictor.EvictorContext): Unit = ???
6               // 定义执行窗口函数之后的移除数据操作
7               override def evictAfter(iterable: lang.Iterable[TimestampedValue[Event]], i: Int, w:
    Window, evictorContext: Evictor.EvictorContext): Unit = ???
8   })
```

允许延迟

在事件时间语义下，窗口中可能会出现数据迟到的情况。迟到数据默认会被直接丢弃，不会进入窗口进行统计计算。这样可能会导致统计结果不准确。 为了解决迟到数据的问题，Flink 提供了一个特殊的接口，可以为窗口算子设置一个“允许的最大延迟”（Allowed Lateness）。也就是说，我们可以设定允许延迟一段时间，在这段时间内，窗口不会销毁，继续到来的数据依然可以进入窗口中并触发计算。直到水位线推进到了窗口结束时间 + 延迟时间，才真正将窗口的内容清空，正式关闭窗口。 基于 WindowedStream 调用 allowedLateness()方法，传入一个 Time 类型的延迟时间，就可以表示允许这段时间内的延迟数据。

```
1   stream.keyBy(_.user)
2           .window(TumblingEventTimeWindows.of(Time.seconds(1)))
3           .allowedLateness(Time.minutes(1))
```

测输出流

Flink 还提供了另外一种方式处理迟到数据。我们可以将未收入窗口的迟到数据，放入“侧输出流”（side output）进行另外的处理。所谓的侧输出流，相当于是数据流的一个“分支”，这个流中单独放置那些本该被丢弃的数据。 基于 WindowedStream 调用 sideOutputLateData() 方法，就可以实现这个功能。方法需要传入一个“输出标签”（OutputTag），用来标记分支的迟到数据流。因为保存的就是流中的原始数据，所以 OutputTag 的类型与流中数据类型相同。

```
1   val OutputTag: OutputTag[Event] = new OutputTag[Event]("late")
2   stream.keyBy(_.user)
3           .window(TumblingEventTimeWindows.of(Time.seconds(1)))
4           .sideOutputLateData(OutputTag)
```

迟到数据处理

所谓的“迟到数据”（late data），是指某个水位线之后到来的数据，它的时间戳其实是在水位线之前的。所以只有在事件时间语义下，讨论迟到数据的处理才是有意义的。

设置水位线延迟时间

水位线是事件时间的进展，它是我们整个应用的全局逻辑时钟。水位线生成之后，会随着数据在任务间流动，从而给每个任务指明当前的事件时间。所以从这个意义上讲，水位线是一个覆盖万物的存在，它并不只针对事件时间窗口有效。

水位线其实是所有事件时间定时器触发的判断标准。那么水位线的延迟，当然也就是全局时钟的滞后。

既然水位线这么重要，那一般情况就不应该把它的延迟设置得太大，否则流处理的实时性就会大大降低。因为水位线的延迟主要是用来对付分布式网络传输导致的数据乱序，而网络传输的乱序程度一般并不会很大，大多集中在几毫秒至几百毫秒。所以实际应用中，我们往往会给水位线设置一个“能够处理大多数乱序数据的小延迟”，视需求一般设在毫秒~秒级。

当我们设置了水位线延迟时间后，所有定时器就都会按照延迟后的水位线来触发。如果一个数据所包含的时间戳，小于当前的水位线，那么它就是所谓的“迟到数据”。

允许窗口处理迟到数据

除设置水位线延迟外，Flink 的窗口也是可以设置延迟时间，允许继续处理迟到数据的。

这种情况下，由于大部分乱序数据已经被水位线的延迟等到了，所以往往迟到的数据不会太多。这样，我们会在水位线到达窗口结束时间时，先快速地输出一个近似正确的计算结果；然后保持窗口继续等到延迟数据，每来一条数据，窗口就会再次计算，并将更新后的结果输出。

这样就可以逐步修正计算结果，最终得到准确的统计值了。

这其实就是著名的 Lambda 架构。原先需要两套独立的系统来同时保证实时性和结果的最终正确性，如今 Flink 一套系统就全部搞定了。

将迟到数据放入窗口侧输出流

只能将之前的窗口计算结果保存下来，然后获取侧输出流中的迟到数据，判断数据所属的窗口，手动对结果进行合并更新。尽管有些烦琐，实时性也不够强，但能够保证最终结果一定是正确的。

总结

Flink 处理迟到数据，对于结果的正确性有三重保障：水位线的延迟，窗口允许迟到数据，以及将迟到数据放入窗口侧输出流。

```
1   package Tutorial.WindowAPI
2
3   import Tutorial.DataStreamAPI.ClickSource
4   import org.apache.flink.api.common.eventtime.{SerializableTimestampAssigner, WatermarkStrategy}
5   import org.apache.flink.api.common.functions.AggregateFunction
6   import org.apache.flink.streaming.api.scala._
7   import org.apache.flink.streaming.api.scala.function.ProcessWindowFunction
8   import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows
9   import org.apache.flink.streaming.api.windowing.time.Time
10  import org.apache.flink.streaming.api.windowing.windows.TimeWindow
11  import org.apache.flink.util.Collector
12
13  import java.time.Duration
14  import java.util.Date
15
16  /**
```

```
 * Project:  BigDataCode
 * Create date:  2023/6/29
 * Created by lwPigKing
 */
object AllowLateness {
  def main(args: Array[String]): Unit = {

    val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
    env.setParallelism(1)

    val stream: DataStream[Event] = env.socketTextStream("localhost", 7777)
      .map(data => {
        val fields: Array[String] = data.split(",")
        Event(fields(0), fields(1), fields(2).toLong)
      })
      // 方法一：设置watermark延迟时间2秒钟
      .assignTimestampsAndWatermarks(
        WatermarkStrategy.forBoundedOutOfOrderness[Event](Duration.ofSeconds(2))
          .withTimestampAssigner(new SerializableTimestampAssigner[Event] {
            override def extractTimestamp(element: Event, recordTimestamp: Long): Long = element.timestamp
          })
      )

    val outputTag: OutputTag[Event] = OutputTag[Event]("late")
    val result: DataStream[UrlViewCount] = stream.keyBy(_.url)
      .window(TumblingEventTimeWindows.of(Time.seconds(10)))
      // 方式二：允许窗口处理迟到数据，设置1分钟的等待时间
      .allowedLateness(Time.minutes(1))
      // 方式三：将最后的迟到数据输出到侧输出流
      .sideOutputLateData(outputTag)
      .aggregate(new UrlViewCountAgg, new UrlViewCountResult)

    result.print("result")
    result.getSideOutput(outputTag).print("late")
    stream.print("input")


    env.execute()
  }
  case class Event(user: String, url: String, timestamp: Long)
  case class UrlViewCount(url: String, count: Long, windowStart: Long, windowEnd: Long)

  class UrlViewCountAgg extends AggregateFunction[Event, Long, Long] {
    override def createAccumulator(): Long = 0L

    override def add(value: Event, accumulator: Long): Long = accumulator + 1L

    override def getResult(accumulator: Long): Long = accumulator

    override def merge(a: Long, b: Long): Long = ???
  }

  class UrlViewCountResult extends ProcessWindowFunction[Long, UrlViewCount, String, TimeWindow] {
    override def process(key: String, context: Context, elements: Iterable[Long], out: Collector[UrlViewCount]): Unit = {
```

```
            out.collect(UrlViewCount(key, elements.iterator.next(), context.window.getStart,
    context.window.getEnd))
        }
    }
}
```

第四章 ProcessFunctionAPI

之前所介绍的流处理 API，无论是基本的转换、聚合，还是更为复杂的窗口操作，其实都是基于 DataStream 进行转换的；所以可以统称为 DataStream API，这也是 Flink 编程的核心。而我们知道，为了让代码有更强大的表现力和易用性，Flink 本身提供了多层 API，DataStream API 只是中间的一环，在更底层，我们可以不定义任何具体的算子（比如 map()，filter()，或者 window()），而只是提炼出一个统一的“处理”（process）操作——它是所有转换算子的一个概括性的表达，可以自定义处理逻辑，所以这一层接口就被叫作“处理函数”（process function）。

ProcessFunction

处理函数主要是定义数据流的转换操作，所以也可以把它归到转换算子中。我们知道在 Flink 中几乎所有转换算子都提供了对应的函数类接口，处理函数也不例外；它所对应的函数类，就叫作 ProcessFunction。

我们之前学习的转换算子，一般只是针对某种具体操作来定义的，能够拿到的信息比较有限。比如 map()算子，我们实现的 MapFunction 中，只能获取到当前的数据，定义它转换之后的形式；而像窗口聚合这样的复杂操作，AggregateFunction 中除数据外，还可以获取到当前的状态（以累加器 Accumulator 形式出现）。另外我们还介绍过富函数类，比如 RichMapFunction，它提供了获取运行时上下文的方法 getRuntimeContext()，可以拿到状态，还有并行度、任务名称之类的运行时信息。

但是无论哪种算子，如果我们想要访问事件的时间戳，或者当前的水位线信息，都是完全做不到的。这时就需要使用处理函数（ProcessFunction）。

处理函数提供了一个“定时服务”（TimerService），我们可以通过它访问流中的事件（event）、时间戳（timestamp）、水位线（watermark），甚至可以注册“定时事件”。而且处理函数继承了 AbstractRichFunction 抽象类，所以拥有富函数类的所有特性，同样可以访问状态（state）和其他运行时信息。此外，处理函数还可以直接将数据输出到侧输出流（side output）中。所以，处理函数是最为灵活的处理方法，可以实现各种自定义的业务逻辑；同时也是整个DataStream API 的底层基础。 处理函数的使用与基本的转换操作类似，只需要直接基于 DataStream 调用 process()方法就可以了。方法需要传入一个 ProcessFunction 作为参数，用来定义处理逻辑。

```
val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
env.setParallelism(1)

env.addSource(new ClickSource)
    .assignAscendingTimestamps(_.timestamp)
    .process(new ProcessFunction[Event, String] {
          override def processElement(i: Event, context: ProcessFunction[Event, String]#Context,
collector: Collector[String]): Unit = {
                if (i.user.equals("Mary")) {
                      collector.collect(i.user)
                } else if (i.user.equals("Bob")) {
                      collector.collect(i.user)
                }
                println(context.timerService().currentWatermark())
```

```
14              }
15          })
16          .print()
```

Flink 中的处理函数其实是一个大家族，ProcessFunction 只是其中一员。

Flink 提供了 8 个不同的处理函数：

（1）ProcessFunction

最基本的处理函数，基于 DataStream 直接调用 process()时作为参数传入。

（2）KeyedProcessFunction

对流按键分区后的处理函数，基于 KeyedStream 调用 process()时作为参数传入。要想使用

定时器，必须基于 KeyedStream。

（3）ProcessWindowFunction

开窗之后的处理函数，也是全窗口函数的代表。基于 WindowedStream 调用 process()时作

为参数传入。

（4）ProcessAllWindowFunction

同样是开窗之后的处理函数，基于 AllWindowedStream 调用 process()时作为参数传入。

（5）CoProcessFunction

合并（connect）两条流之后的处理函数，基于 ConnectedStreams 调用 process()时作为参

数传入。

（6）ProcessJoinFunction

间隔连接（interval join）两条流之后的处理函数，基于 IntervalJoined 调用 process()时作为参数传入。

（7）BroadcastProcessFunction

广播连接流处理函数，基于 BroadcastConnectedStream 调用 process()时作为参数传入。这里的“广播连接流”BroadcastConnectedStream，是一个未 keyBy 的普通 DataStream 与一个广播流（BroadcastStream）做连接（conncet）之后的产物。

（8）KeyedBroadcastProcessFunction

按键分区的广播连接流处理函数，同样是基于 BroadcastConnectedStream 调用 process()时作为参数传入。与 BroadcastProcessFunction 不同的是，这时的广播连接流，是一个 KeyedStream 与广播流（BroadcastStream）做连接之后的产物。

KeyedProcessFunction

在 Flink 程序中，为了实现数据的聚合统计，或者开窗计算之类的功能，我们一般都要先用 keyBy()算子对数据流进行“按键分区”，得到一个 KeyedStream。而只有在 KeyedStream 中，才支持使用 TimerService 设置定时器的操作。所以一般情况下，我们都是先做了 keyBy()分区之后，再去定义处理操作；代码中更加常见的处理函数是 KeyedProcessFunction。

KeyedProcessFunction 的一个特色，就是可以灵活地使用定时器。

定时器（timers）是处理函数中进行时间相关操作的主要机制。在 onTimer()方法中可以实现定时处理的逻辑，而它能触发的前提，就是之前曾经注册过定时器、并且现在已经到了触发时间。注册定时器的功能，是通过上下文中提供的“定时服务”（TimerService）来实现的。ProcessFunction的上下文（Context）中提供了timerService()方法，可以直接返回一个TimerService对象。

代码中使用KeyedProcessFunction，只要基于keyBy()之后的KeyedStream，直接调用process()方法，这时需要传入的参数就是KeyedProcessFunction的实现类。

```
val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
env.addSource(new ClickSource)
    .keyBy( r => true)
     .process(new KeyedProcessFunction[Boolean, Event, String] {
            override def processElement(i: Event, context: KeyedProcessFunction[Boolean, Event, String]#Context, collector: Collector[String]): Unit = {
                  val currTs: Long = context.timerService().currentProcessingTime()
                  collector.collect("数据到达，到达时间：" + new Timestamp(currTs))
                  // 注册10秒钟之后的处理时间定时器
                  context.timerService().registerProcessingTimeTimer(currTs + 10 * 1000L)
            }

             override def onTimer(timestamp: Long, ctx: KeyedProcessFunction[Boolean, Event, String]#OnTimerContext, out: Collector[String]): Unit = {
                  out.collect("定时器触发，触发时间：" + new Timestamp(timestamp))
            }
      })
    .print()
env.execute()
```

第五章 状态编程

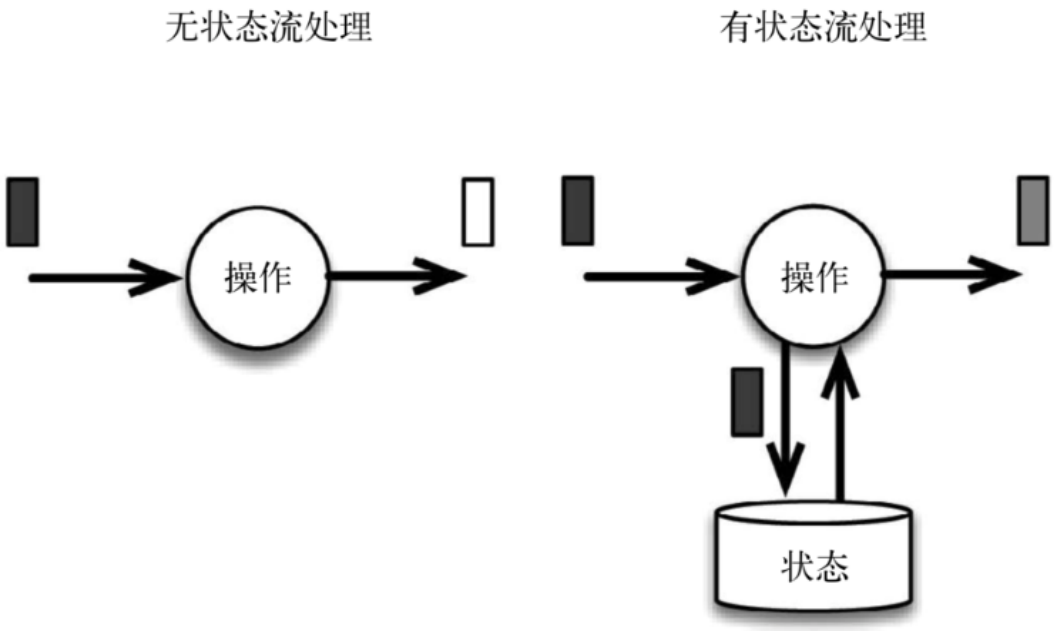
流式计算分为无状态和有状态两种情况。无状态的计算观察每个独立事件，并根据最后一个事件输出结果。例如，流处理应用程序从传感器接收温度读数，并在温度超过90度时发出警告。有状态的计算则会基于多个事件输出结果。以下是一些例子。

所有类型的窗口。例如，计算过去一小时的平均温度，就是有状态的计算。

所有用于复杂事件处理的状态机。例如，若在一分钟内收到两个相差20度以上的温度读数，则发出警告，这是有状态的计算。

流与流之间的所有关联操作，以及流与静态表或动态表之间的关联操作，都是有状态的计算。

下图展示了无状态流处理和有状态流处理的主要区别。无状态流处理分别接收每条数据记录(图中的黑条)，然后根据最新输入的数据生成输出数据(白条)。有状态流处理会维护状态(根据每条输入记录进行更新)，并基于最新输入的记录和当前的状态值生成输出记录(灰条)。



上图中输入数据由黑条表示。无状态流处理每次只转换一条输入记录，并且仅根据最新的输入记录输出结果(白条)。有状态 流处理维护所有已处理记录的状态值，并根据每条新输入的记录更新状态，因此输出记录(灰条)反映的是综合考虑多个事件之后的结果。

尽管无状态的计算很重要，但是流处理对有状态的计算更感兴趣。事实上，正确地实现有状态的计算比实现无状态的计算难得多。旧的流处理系统并不支持有状态的计算，而新一代的流处理系统则将状态及其正确性视为重中之重。

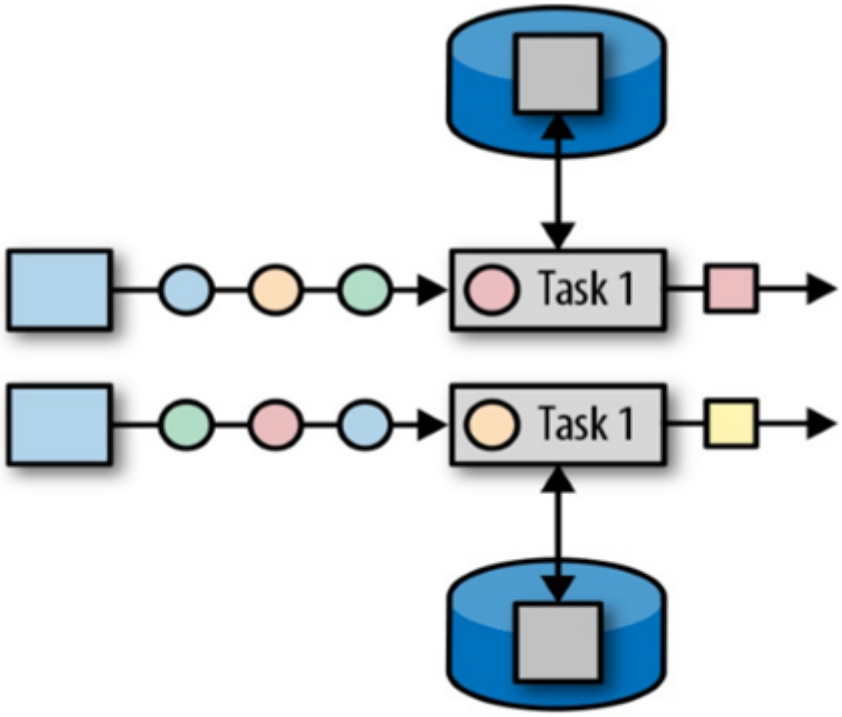
Flink内置的很多算子，数据源source，数据存储sink都是有状态的，流中的数据都是buffer records，会保存一定的元素或者元数据。例如: ProcessWindowFunction会缓存输入流的数据，ProcessFunction会保存设置的定时器信息等等。

在Flink中，状态始终与特定算子相关联。总的来说，有两种类型的状态：

算子状态（operator state）、键控状态（keyed state）

operator state

算子状态的作用范围限定为算子任务。这意味着由同一并行任务所处理的所有数据都可以访问到相同的状态，状态对于同一任务而言是共享的。算子状态不能由相同或不同算子的另一个任务访问。



Flink为算子状态提供三种基本数据结构：

列表状态（List state）：将状态表示为一组数据的列表。

联合列表状态（Union list state）：也将状态表示为数据的列表。它与常规列表状态的区别在于，在发生故障时，或者从保存点（savepoint）启动应用程序时如何恢复。

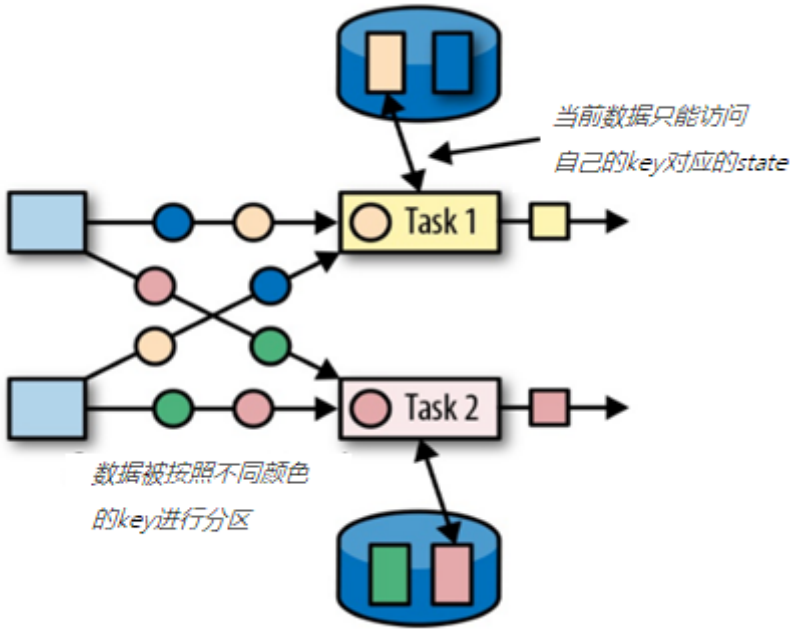
广播状态（Broadcast state）：如果一个算子有多项任务，而它的每项任务状态又都相同，那么这种特殊情况最适合应用广播状态。

```
package Tutorial.StateProgamming

import Tutorial.DataStreamAPI.{ClickSource, Event}
import org.apache.flink.api.common.state.{ListState, ListStateDescriptor}
import org.apache.flink.runtime.state.{FunctionInitializationContext, FunctionSnapshotContext}
import org.apache.flink.streaming.api.checkpoint.CheckpointedFunction
import org.apache.flink.streaming.api.functions.sink.SinkFunction
import org.apache.flink.streaming.api.scala._

import scala.collection.mutable.ListBuffer

/**
 * Project:  BigDataCode
 * Create date:  2023/6/30
 * Created by lwPigKing
 */
object BufferingSinkExample {
   def main(args: Array[String]): Unit = {

      /**
       * 接下来我们举一个算子状态的应用案例。在下面的例子中，自定义的 SinkFunction 会在 CheckpointedFunction 中进行数据缓存，然后统一发送到下游。
       * 这个例子演示了列表状态的平均分割重组（event-split redistribution）。
       */

      val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
```

```
        env.setParallelism(1)

        env.addSource(new ClickSource)
          .assignAscendingTimestamps(_.timestamp)
          .addSink(new BufferingSink(10))

        env.execute()

    }

    // 实现自定义的SinkFunction
    class BufferingSink(threshold: Int) extends SinkFunction[Event] with CheckpointedFunction {

        // 定义列表状态，保存要缓冲的数据
        var bufferedState: ListState[Event] = _

        // 定义一个本地变量列表
        val bufferedList: ListBuffer[Event] = ListBuffer[Event]()

        override def invoke(value: Event, context: SinkFunction.Context): Unit = {
            // 缓冲数据
            bufferedList += value

            // 判断是否达到了阈值
            if (bufferedList.size == threshold) {
                // 输出到外部系统，用打印到控制台模拟
                bufferedList.foreach(data => println(data))
                println("==========输出完毕==========")

                // 清空缓冲
                bufferedList.clear()
            }
        }

        override def snapshotState(functionSnapshotContext: FunctionSnapshotContext): Unit = {
            // 清空状态
            bufferedState.clear()
            for (data <- bufferedList) {
                bufferedState.add(data)
            }
        }

        override def initializeState(context: FunctionInitializationContext): Unit = {
            bufferedState = context.getOperatorStateStore.getListState(new ListStateDescriptor[Event]
("buffered-list", classOf[Event]))
            // 判断如果是从故障中恢复，那么就将状态中的数据添加到局部变量中
            if (context.isRestored) {
                import scala.collection.convert.ImplicitConversions._
                for (data <- bufferedState.get()) {
                    bufferedList += data
                }
            }
        }
    }

}
```

keyed state

键控状态是根据输入数据流中定义的键（key）来维护和访问的。Flink为每个键值维护一个状态实例，并将具有相同键的所有数据，都分区到同一个算子任务中，这个任务会维护和处理这个key对应的状态。当任务处理一条数据时，它会自动将状态的访问范围限定为当前数据的key。因此，具有相同key的所有数据都会访问相同的状态。Keyed State很类似于一个分布式的key-value map数据结构，只能用于KeyedStream（keyBy算子处理之后）。



Flink的Keyed State支持以下数据类型：

ValueState[T]保存单个的值，值的类型为T。

- get操作: ValueState.value()
- set操作: ValueState.update(value: T)

ListState[T]保存一个列表，列表里的元素的数据类型为T。基本操作如下：

- ListState.add(value: T)
- ListState.addAll(values: java.util.List[T])
- ListState.get()返回Iterable[T]
- ListState.update(values: java.util.List[T])

MapState[K, V]保存Key-Value对。

- MapState.get(key: K)
- MapState.put(key: K, value: V)
- MapState.contains(key: K)
- MapState.remove(key: K)

ReducingState[T]

AggregatingState[I, O]

State.clear()是清空操作。

ValueState

```scala
package Tutorial.StateProgamming

import Tutorial.DataStreamAPI.{ClickSource, Event}
import org.apache.flink.api.common.state.{ValueState, ValueStateDescriptor}
import org.apache.flink.streaming.api.functions.KeyedProcessFunction
import org.apache.flink.streaming.api.scala._
import org.apache.flink.util.Collector



/**
 * Project:  BigDataCode
 * Create date:  2023/6/30
 * Created by lwPigKing
 */
object PeriodicPVExample {
   def main(args: Array[String]): Unit = {
      /**
       * 1.值状态（ValueState）
       * 我们这里会使用用户id来进行分流，然后分别统计每个用户的pv数据，由于我们并不想每次pv 加一，就将统计结果发送到下游去，所以这里我们注册了一个定时器，
       * 用来隔一段时间发送pv的统计结果，这样对下游算子的压力不至于太大。具体实现方式是定义一个用来保存定时器时间戳的值状态变量。
       * 当定时器触发并向下游发送数据以后，便清空储存定时器时间戳的状态变量，这样当新的数据到来时，发现并没有定时器存在，就可以注册新的定时器了，
       * 注册完定时器之后将定时器的时间戳继续保存在状态变量中。
       */

      val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
      env.setParallelism(1)

      env.addSource(new ClickSource)
        .assignAscendingTimestamps(_.timestamp)
        .keyBy(_.user)
        .process(new PeriodicPvResult)
        .print()

      env.execute()

  }

   class PeriodicPvResult extends KeyedProcessFunction[String, Event, String] {

      // 懒加载值状态变量，用来存储当前pv数据
      lazy val countState: ValueState[Long] = getRuntimeContext.getState(new ValueStateDescriptor[Long]("count", classOf[Long]))

      // 懒加载状态变量，用来存储发送pv数据的定时器的时间戳
      lazy val timerTsState: ValueState[Long] = getRuntimeContext.getState(new ValueStateDescriptor[Long]("timer-ts", classOf[Long]))

```

```
47          override def processElement(i: Event, context: KeyedProcessFunction[String, Event,
    String]#Context, collector: Collector[String]): Unit = {
48            // 更新count值
49            val count: Long = countState.value()
50            countState.update(count + 1)
51
52            // 如果保存发送pv数据的定时器的时间戳的状态变量为0L，则注册一个10秒后定时器
53            if (timerTsState.value() == 0L) {
54              // 注册定时器
55              context.timerService().registerEventTimeTimer(i.timestamp + 10 * 1000L)
56              // 将定时器对的时间戳保存在状态变量中
57              timerTsState.update(i.timestamp + 10 * 1000L)
58            }
59        }
60
61        override def onTimer(timestamp: Long, ctx: KeyedProcessFunction[String, Event,
    String]#OnTimerContext, out: Collector[String]): Unit = {
62            out.collect("用户: " + ctx.getCurrentKey + "的PV是: " + countState.value())
63            // 清空保存定时器时间戳的状态变量，这样新数据到来时又可以注册定时器了
64            timerTsState.clear()
65        }
66
67
68      }
69
70  }
71
```

ListState

```
1   package Tutorial.StateProgamming
2
3   import org.apache.flink.api.common.state.{ListState, ListStateDescriptor}
4   import org.apache.flink.streaming.api.functions.co.CoProcessFunction
5   import org.apache.flink.streaming.api.scala._
6   import org.apache.flink.util.Collector
7
8   /**
9    * Project:  BigDataCode
10   * Create date:  2023/6/30
11   * Created by lwPigKing
12   */
13  object TwoStreamJoinExample {
14    def main(args: Array[String]): Unit = {
15      val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
16      env.setParallelism(1)
17
18      /**
19       * 2.列表状态（ListState）
20       * 在Flink SQL中，支持两条流的全量join，语法如下：
21       * SELECT * FROM A INNER JOIN B WHERE A.id = B.id;
22       * 这样一条 SQL语句要慎用，因为 Flink会将A流和B流的所有数据都保存下来，然后进行 join。
23       * 不过在这里可以用列表状态变量来实现一下这个 SQL 语句的功能。
24       */
25
26      val stream1: DataStream[(String, String, Long)] = env.fromElements(
```

```
27          ("a", "stream-1", 1000L),
28          ("b", "stream-1", 2000L)
29        )
30          .assignAscendingTimestamps(_._3)
31
32        val stream2: DataStream[(String, String, Long)] = env.fromElements(
33          ("a", "stream-2", 3000L),
34          ("b", "stream-2", 4000L)
35        ).assignAscendingTimestamps(_._3)
36
37        // 连接两条流
38        stream1.keyBy(_._1)
39          .connect(stream2.keyBy(_._1))
40          .process(new CoProcessFunction[(String, String, Long), (String, String, Long), String] {
41
42              // 保存来自第一条流的事件的列表状态变量
43              lazy val stream1ListState: ListState[(String, String, Long)] =
    getRuntimeContext.getListState(new ListStateDescriptor[(String, String, Long)]("stream1-list",
    classOf[(String, String, Long)]))
44
45              // 保存来自第二条流的事件的列表状态变量
46              lazy val stream2ListState: ListState[(String, String, Long)] =
    getRuntimeContext.getListState(new ListStateDescriptor[(String, String, Long)]("stream2-list",
    classOf[(String, String, Long)]))
47
48              override def processElement1(in1: (String, String, Long), context:
    CoProcessFunction[(String, String, Long), (String, String, Long), String]#Context, collector:
    Collector[String]): Unit = {
49                  // 将事件添加到列表状态变量
50                  stream1ListState.add(in1)
51
52                  import scala.collection.convert.ImplicitConversions._
53                  for (in2 <- stream2ListState.get) {
54                      collector.collect(in1 + "=>" + in2)
55                  }
56              }
57
58              override def processElement2(in2: (String, String, Long), context:
    CoProcessFunction[(String, String, Long), (String, String, Long), String]#Context, collector:
    Collector[String]): Unit = {
59                  stream2ListState.add(in2)
60                  import scala.collection.convert.ImplicitConversions._
61                  for (in1 <- stream1ListState.get) {
62                      collector.collect(in1 + "=>" + in2)
63                  }
64              }
65          })
66          .print()
67
68        env.execute()
69
70    }
71  }
72
```

MapState

```scala
package Tutorial.StateProgamming

import Tutorial.DataStreamAPI.{ClickSource, Event}
import org.apache.flink.api.common.state.{MapState, MapStateDescriptor}
import org.apache.flink.streaming.api.functions.KeyedProcessFunction
import org.apache.flink.streaming.api.scala._
import org.apache.flink.util.Collector
import org.apache.spark.sql.execution.streaming.FileStreamSource.Timestamp

/**
 * Project:  BigDataCode
 * Create date:  2023/6/30
 * Created by lwPigKing
 */
object FakeWindowExample {
   def main(args: Array[String]): Unit = {
      val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
      env.setParallelism(1)

      /**
       * 3.映射状态（MapState）
       * 映射状态的用法和Java中的HashMap很相似。
       * 在这里可以通过 MapState的使用来探索一下窗口的底层实现，也就是要用映射状态来完整模拟窗口的功能。
       * 这里模拟一个滚动窗口。要计算的是每一个url在每一个窗口中的pv数据。
       * 之前使用增量聚合和全窗口聚合结合的方式实现过这个需求。这里用MapState再来实现一下。
       */

      // 统计每10s滚动窗口内，每个url的pv
      env.addSource(new ClickSource)
        .assignAscendingTimestamps(_.timestamp)
        .keyBy(_.url)
        .process(new FakeWindowResult(10000L))
        .print()

      env.execute()
  }

   class FakeWindowResult(windowSize: Long) extends KeyedProcessFunction[String, Event, String] {

      // 初始化一个MapState状态变量，key为窗口的开窗时间m，value为窗口对应的pv数据
      lazy val windowPvMapState: MapState[Long, Long] = getRuntimeContext.getMapState(new MapStateDescriptor[Long, Long]("window-pv", classOf[Long], classOf[Long]))

      override def processElement(i: Event, context: KeyedProcessFunction[String, Event, String]#Context, collector: Collector[String]): Unit = {

         // 根据事件的时间戳，计算当前事件所属的窗口开始和结束时间
         val windowStart: Long = i.timestamp / windowSize * windowSize
         val windowEnd: Long = windowStart + windowSize

         // 注册一个windowEnd-1ms的定时器，用来触发窗口计算
         context.timerService().registerEventTimeTimer(windowEnd-1)

         // 更新状态中的pv值
```

```
            if (windowPvMapState.contains(windowStart)) {
                val pv: Long = windowPvMapState.get(windowStart)
                windowPvMapState.put(windowStart, pv + 1L)
            } else {
                // 如果key不存在，说明当前窗口的第一一个事件到达
                windowPvMapState.put(windowStart, 1L)
            }
        }

        override def onTimer(timestamp: Long, ctx: KeyedProcessFunction[String, Event,
String]#OnTimerContext, out: Collector[String]): Unit = {
            // 计算窗口的结束时间和开始时间
            val windowEnd: Long = timestamp + 1L
            val windowStart: Long = windowEnd - windowSize

            // 发送窗口计算的结果
            out.collect("url: " + ctx.getCurrentKey + "访问量: " + windowPvMapState.get(windowStart)
+ "窗口: " + windowStart + "~~~~~" + windowEnd)
            // 模拟窗口的销毁，清除map中的key
            windowPvMapState.remove(windowStart)
        }

    }

}
```

AggregatingState

```
package Tutorial.StateProgamming

import Tutorial.DataStreamAPI.{ClickSource, Event}
import org.apache.flink.api.common.functions.{AggregateFunction, RichFlatMapFunction}
import org.apache.flink.api.common.state.{AggregatingState, AggregatingStateDescriptor,
ValueState, ValueStateDescriptor}
import org.apache.flink.streaming.api.scala._
import org.apache.flink.util.Collector

/**
 * Project:  BigDataCode
 * Create date:  2023/6/30
 * Created by lwPigKing
 */
object AverageTimestampExample {
   def main(args: Array[String]): Unit = {

      /**
       * 4.聚合状态（AggregatingState）
       * 举一个简单的例子，对用户点击事件流每5个数据统计一次平均时间戳。
       * 这是一个类似计数窗口（CountWindow）求平均值的计算，这里可以使用一个有聚合状态的
RichFlatMapFunction来实现。
       */

      val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
      env.setParallelism(1)

```

```
        env.addSource(new ClickSource)
          .assignAscendingTimestamps(_.timestamp)
          .keyBy(_.user)
          .flatMap(new AvgTsResult)
          .print()

        env.execute()
    }

    class AvgTsResult extends RichFlatMapFunction[Event, String] {

        // 定义聚合状态
         lazy val avgTsAggState: AggregatingState[Event, Long] =
getRuntimeContext.getAggregatingState(new AggregatingStateDescriptor[Event, (Long, Long), Long](
          "avg-ts",
          new AggregateFunction[Event, (Long, Long), Long] {
             override def createAccumulator(): (Long, Long) = (0L, 0L)

             override def add(value: Event, accumulator: (Long, Long)): (Long, Long) = {
               (accumulator._1 + value.timestamp, accumulator._2 + 1)
             }

             override def getResult(accumulator: (Long, Long)): Long = accumulator._1 /
accumulator._2

             override def merge(a: (Long, Long), b: (Long, Long)): (Long, Long) = ???
          },
          classOf[(Long, Long)]
        ))

        // 定义一个值状态，保存当前已经到达的数据个数
        lazy val countState: ValueState[Long] = getRuntimeContext.getState(new
ValueStateDescriptor[Long]("count", classOf[Long]))

        override def flatMap(value: Event, out: Collector[String]): Unit = {
          avgTsAggState.add(value)
          // 更新count值
          val count: Long = countState.value()
          countState.update(count + 1)

          // 判断是否达到了技术窗口的长度，输出结果
          if (countState.value() == 5) {
             out.collect(s"${value.user}的平均时间戳为${avgTsAggState.get()}")
             // 窗口销毁
             countState.clear()
          }
        }

    }

}
```

状态持久化和状态后端

在 Flink 的状态管理机制中，很重要的一个功能就是对状态进行持久化（persistence）保存，这样就可以在发生故障后进行重启恢复。Flink 对状态进行持久化的方式，就是将当前所有分布式状态进行“快照”保存，写入一个“检查点”（checkpoint）或者保存点（savepoint）保存到外部存储系统中。具体的存储介质，一般是分布式文件系统（distributed file system）。

checkpoint

有状态流应用中的检查点（checkpoint），其实就是所有任务的状态在某个时间点的一个快照（一份拷贝）。简单来讲，就是一次“存盘”，让我们之前处理数据的进度不要丢掉。在一个流应用程序运行时，Flink 会定期保存检查点，在检查点中会记录每个算子的 id 和状态；如果发生故障，Flink 就会用最近一次成功保存的检查点来恢复应用的状态，重新启动处理流程，就如同“读档”一样。

如果保存检查点之后又处理了一些数据，然后发生了故障，那么重启恢复状态之后这些数据带来的状态改变会丢失。为了让最终处理结果正确，我们还需要让源（Source）算子重新读取这些数据，再次处理一遍。这就需要流的数据源具有“数据重放”的能力，一个典型的例子就是 Kafka，我们可以通过保存消费数据的偏移量、故障重启后重新提交来实现数据的重放。这是对“至少一次”（at least once）状态一致性的保证，如果希望实现“精确一次”（exactly once）的一致性，还需要数据写入外部系统时的相关保证。默认情况下，检查点是被禁用的，需要在代码中手动开启。直接调用执行环境的.enableCheckpointing()方法就可以开启检查点。

除了检查点之外，Flink 还提供了“保存点”（savepoint）的功能。保存点在原理和形式上跟检查点完全一样，也是状态持久化保存的一个快照；区别在于，保存点是自定义的镜像保存，所以不会由 Flink 自动创建，而需要用户手动触发。这在有计划地停止、重启应用时非常有用。

State Backends

检查点的保存离不开 JobManager 和 TaskManager，以及外部存储系统的协调。在应用进行检查点保存时，首先会由 JobManager 向所有 TaskManager 发出触发检查点的命令； TaskManger 收到之后，将当前任务的所有状态进行快照保存，持久化到远程的存储介质中；完成之后向 JobManager 返回确认信息。这个过程是分布式的，当 JobManger 收到所有 TaskManager 的返回信息后，就会确认当前检查点成功保存，而这一切工作的协调，就需要一个“专职人员”来完成。

在 Flink 中，状态的存储、访问以及维护，都是由一个可插拔的组件决定的，这个组件就叫作状态后端（state backend）。状态后端主要负责两件事：一是本地的状态管理，二是将检查点（checkpoint）写入远程的持久化存储。

第六章 FlinkSQL&TableAPI

这部分相对于SparkSQL理解起来可能会比较难（主要是FlinkSQL的时间语义并不好写），并且如果要构成一个完整的FlinkSQL和TableAPI的知识体系，难度还是比较大的。只有多练习了。

获取表环境

```
val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
env.setParallelism(1)
// 获取表环境
val tableEnv: StreamTableEnvironment = StreamTableEnvironment.create(env)
```

连接外部系统

```
// 创建输入表，连接外部系统读取数据
tableEnv.executeSql("create temporary table inputTable ... with ('connector' = ...)")

// 注册一个表，连接到外部系统，用于输出
tableEnv.executeSql("create temporary table outputTable ... with ('connect' = ...)")
```

表的查询

```
// 用SQL的方式提取数据
val visitTable: Table = tableEnv.sqlQuery(s"select url, user from ${eventTable}")

// 用Table API方式提取数据
val visitTable: Table = eventTable.select($("url"), $("user"))

// 执行SQL对表进行查询转换，得到一个新的表
val table1: Table = tableEnv.sqlQuery("select ... from inputTable...")

// 使用TableAPI对表进行查询转换，得到一个新的表
val table2: Table = tableEnv.from("inputTable").select("...")

// 将得到的结果写入输出表
val tableResult: TableResult = table1.executeInsert("outputTable")

// 虚拟表(临时视图)
// 得到的newTable是一个中间转换结果，如果之后又希望直接使用这个表执行SQL，就可以创建虚拟表
val newTable: Table = tableEnv.sqlQuery("select ... from MyTable")
// 第一个参数是注册的表名，第二个参数是Table对象
tableEnv.createTemporaryView("newTable", newTable)

```

表和流相互转换

```
// 表 -> 流：写完SQL后转换成流然后print测试
// toDataStream并不适合聚合操作，是因为无法进行更新
tableEnv.toDataStream(aliceVisitTable).print()
// toChangelogStream更新日志
val urlCountTable2: Table = tableEnv.sqlQuery("select user, count(url) from EventTable group by user")
tableEnv.toChangelogStream(urlCountTable2).print()

// 流 -> 表
val eventTable3: Table = tableEnv.fromDataStream(eventStream)
val eventTable4: Table = tableEnv.fromDataStream(eventStream, $("timestamp").as("ts"), $("url"))
tableEnv.createTemporaryView("EventTable", eventStream, $("timestamp").as("ts"), $("url"))
```

TableAPI过程

```scala
package Tutorial.TableAPIAndSQL

import Tutorial.DataStreamAPI.SensorReading
import org.apache.flink.streaming.api.functions.timestamps.BoundedOutOfOrdernessTimestampExtractor
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.table.api.Expressions.$
import org.apache.flink.table.api.{AnyWithOperations, FieldExpression, LiteralIntExpression, Table, Tumble, UnresolvedFieldExpression}
import org.apache.flink.table.api.bridge.scala.{StreamTableEnvironment, dataStreamConversions, tableConversions}

/**
 * Project:  BigDataCode
 * Create date:  2023/7/7
 * Created by lwPigKing
 */
object TableAPI {
   def main(args: Array[String]): Unit = {
      /**
       * 简单了解一下TableAPI的过程
       */
      val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
      env.setParallelism(1)

      val inputStream: DataStream[String] = env.readTextFile("D:\\BigDataCode\\Code\\Flink\\Tutorial\\TableAPIAndSQL\\sensor.txt")
      val dataStream: DataStream[SensorReading] = inputStream.map(data => {
         val dataArray: Array[String] = data.split(",")
         SensorReading(dataArray(0).trim, dataArray(1).trim.toLong, dataArray(2).trim.toDouble)
      })

      val tableEnvironment: StreamTableEnvironment = StreamTableEnvironment.create(env)
      val dataTable: Table = tableEnvironment.fromDataStream(dataStream)

      val selectedTable: Table = dataTable.select($("id"), $("temperature")).filter("id = 'sensor_1'")

      tableEnvironment.toDataStream(selectedTable).print()

      env.execute()


      /**
       * TableAPI的窗口聚合操作
       */
      // 统计每10秒中每个传感器温度值的个数
      val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
      env.setParallelism(1)

      val inputStream: DataStream[String] = env.readTextFile("D:\\BigDataCode\\Code\\Flink\\Tutorial\\TableAPIAndSQL\\sensor.txt")
      val dataStream: DataStream[SensorReading] = inputStream.map(data => {
         val dataArray: Array[String] = data.split(",")
         SensorReading(dataArray(0).trim, dataArray(1).trim.toLong, dataArray(2).toDouble)
```

```
51          }).assignTimestampsAndWatermarks(new BoundedOutOfOrdernessTimestampExtractor[SensorReading]
    (Time.seconds(1)) {
52              override def extractTimestamp(element: SensorReading): Long = element.timestamp * 1000L
53          })
54
55          val tableEnvironment: StreamTableEnvironment = StreamTableEnvironment.create(env)
56          // 处理时间：.proctime  事件时间：rowtime
57          val dataTable: Table = tableEnvironment.fromDataStream(dataStream, $("id"),
    $("temperature"), $("timestamp").rowtime())
58
59          // 按照时间开窗聚合统计
60          val resultTable: Table = dataTable
61            .window(Tumble over 10.seconds on $"timestamp" as "tw")
62            .groupBy($"id", $"tw")
63            .select($"id", $"id".count)
64          // 如果使用了groupBy，table转换成流的时候只能用toRetractStream
65          // Boolean:true表示最新的数据（Insert），false表示过期老数据（Delete）
66          val selectedStream: DataStream[(Boolean, (String, Long))] =
    resultTable.toRetractStream[(String, Long)]
67          selectedStream.print()
68
69          env.execute()
70
71
72      }
73  }
74
```

SQL过程

```
1   package Tutorial.TableAPIAndSQL
2
3   import Tutorial.DataStreamAPI.SensorReading
4   import org.apache.flink.streaming.api.functions.timestamps.BoundedOutOfOrdernessTimestampExtractor
5   import org.apache.flink.streaming.api.scala._
6   import org.apache.flink.streaming.api.windowing.time.Time
7   import org.apache.flink.table.api.Expressions.$
8   import org.apache.flink.table.api.Table
9   import org.apache.flink.table.api.bridge.scala.{StreamTableEnvironment, tableConversions}
10
11  /**
12   * Project:  BigDataCode
13   * Create date:  2023/7/7
14   * Created by lwPigKing
15   */
16  object SQLExample {
17      def main(args: Array[String]): Unit = {
18          val env: StreamExecutionEnvironment = StreamExecutionEnvironment.getExecutionEnvironment
19          env.setParallelism(1)
20
21          val inputStream: DataStream[String] =
    env.readTextFile("D:\\BigDataCode\\Code\\Flink\\Tutorial\\TableAPIAndSQL\\sensor.txt")
22          val dataStream: DataStream[SensorReading] = inputStream.map(data => {
23              val dataArray: Array[String] = data.split(",")
24              SensorReading(dataArray(0).trim, dataArray(1).trim.toLong, dataArray(2).toDouble)
```

```
25          }).assignTimestampsAndWatermarks(new BoundedOutOfOrdernessTimestampExtractor[SensorReading]
    (Time.seconds(1)) {
26              override def extractTimestamp(element: SensorReading): Long = element.timestamp * 1000L
27          })
28
29          val tableEnvironment: StreamTableEnvironment = StreamTableEnvironment.create(env)
30          // 处理时间：.proctime  事件时间：rowtime
31          val dataTable: Table = tableEnvironment.fromDataStream(dataStream, $("id"),
    $("temperature"), $("timestamp").rowtime().as("ts"))
32          tableEnvironment.createTemporaryView("dataTable", dataTable)
33
34          val resultSQLTable: Table = tableEnvironment.sqlQuery("select id, count(id) from
    table(tumble (table dataTable, descriptor(ts), interval '15' second)) group by id, window_start,
    window_end")
35
36          tableEnvironment.toDataStream(resultSQLTable).print()
37
38          env.execute()
39
40      }
41  }
```