> 风平浪静，那是因为有人替你负重前行

Flink之状态编程

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。

有状态的算子

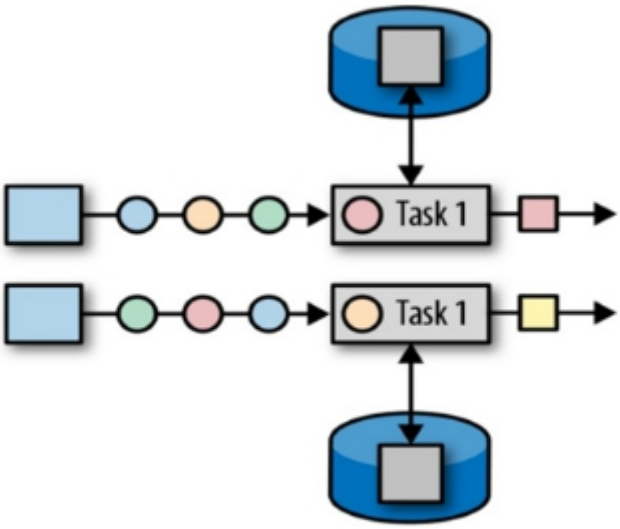
> Flink内置的很多算子，数据源surce，数据存储sink都是有状态的，**流中的数据都是buffer recrds**，会保存一定的元素或者元数据。例如: PrcessWindwFunctin会缓存输入流的数据，PrcessFunctin会保存设置的定时器信息等等。

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（peratr state）
- 键控状态（keyed state）

算子状态

> 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（Unin list state）

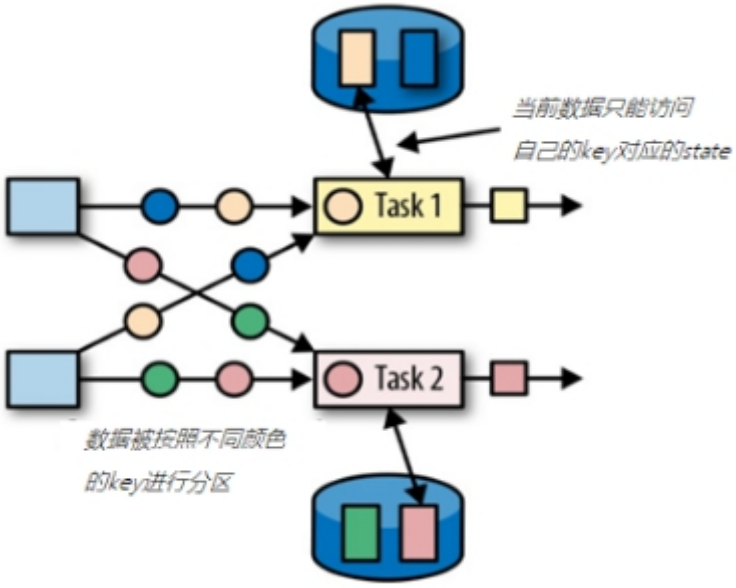
> 也将状态表示为数据的列表。它与常规列表状态的区别在于，在发生故障时，或者从保存点（savepint）启动应用程序时如何恢复。

- 广播状态（Bradcast 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。

键控状态【重点掌握】

> 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.cntains(key: K)
MapState.remve(key: K)
```

- ReducingState[T]
- AggregatingState[I,]

代码实例【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】

```
val sensorData: DataStream[SensorReading] = ...
val keyedData: KeyedStream[SensorReading, String] = sensorData.keyBy(_.id)

val alerts: DataStream[(String, Double, Double)] = keyedData
.flatMap(new TemperatureAlertFunction(1.7))

class TemperatureAlertFunction(val threshold: Double) extends
RichFlatMapFunction[SensorReading, (String, Double, Double)] {
private var lastTempState: ValueState[Double] = _

override def open(parameters: Configuration): Unit = {
val lastTempDescriptor = new ValueStateDescriptor[Double]("lastTemp",
classOf[Double])
    lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)
}

override def flatMap(reading: SensorReading,
out: Collector[(String, Double, Double)]): Unit = {
    val lastTemp = lastTempState.value()
    val tempDiff = (reading.temperature - lastTemp).abs
    if (tempDiff > threshold) {
    out.collect((reading.id, reading.temperature, tempDiff))
}
this.lastTempState.update(reading.temperature)
}
}
```

> 通过RuntimeContext注册StateDescriptor。StateDescriptor以状态state的名字和存储的数据类型为参数。在open()方法中创建state变量。注意复习之前的RichFunction相关知识。接下来我们使用了FlatMap with keyed ValueState的快捷方式flatMapWithState实现以上需求。

```
val alerts: DataStream[(String, Double, Double)] = keyedSensorData
  .flatMapWithState[(String, Double, Double), Double] {
    case (in: SensorReading, None) =>
      (List.empty, Some(in.temperature))
    case (r: SensorReading, lastTemp: Some[Double]) =>
      val tempDiff = (r.temperature - lastTemp.get).abs
      if (tempDiff > 1.7) {
        (List((r.id, r.temperature, tempDiff)), Some(r.temperature))
      } else {
        (List.empty, Some(r.temperature))
      }
  }
```

状态一致性【重点掌握】

> 当在分布式系统中引入状态时，自然也引入了一致性问题。一致性实际上是"正确性级别"的另一种说法，**也就是说在成功处理故障并恢复之后得到的结果，与没有发生任何故障时得到的结果相比，前者到底有多正确?** 举例来说，假设要对最近一小时登录的用户计数。在系统经历故障之后，计数结果是多少？如果有偏差，是有漏掉的计数还是重复计数?

一致性级别

在流处理中，一致性可以分为3个级别：

- at-most-once:

  > 这其实是没有正确性保障的委婉说法——故障发生之后，计数结果可能丢失。同样的还有udp。

- at-least-once

  > 这表示计数结果可能大于正确值，但绝不会小于正确值。也就是说，计数程序在发生故障后可能多算，但是绝不会少算。

- exactly-once

  > 这指的是系统保证在发生故障后得到的计数结果与正确值一致。

曾经，at-least-once非常流行。第一代流处理器(如Storm和Samza)刚问世时只保证at-least-once，原因有二。

（1）**保证exactly-once的系统实现起来更复杂。**这在基础架构层(决定什么代表正确，以及exactly-once的范围是什么)和实现层都很有挑战性。

（2）**流处理系统的早期用户愿意接受框架的局限性**，并在应用层想办法弥补(例如使应用程序具有幂等性，或者用批量计算层再做一遍计算)。

> 最先保证exactly-once的系统(Storm Trident和Spark Streaming)在性能和表现力这**两个方面付出了很大的代价。**为了保证exactly-once，这些系统无法单独地对每条记录运用应用逻辑，而是同时处理多条(一批)记录，**保证对每一批的处理要么全部成功，要么全部失败。**这就导致在得到结果前，必须等待一批记录处理结束。因此，用户经常不得不使用两个流处理框架(一个用来保证exactly-once，另一个用来对每个元素做低延迟处理)，**结果使基础设施更加复杂。**曾经，用户不得不在保证exactly-once与获得低延迟和效率之间权衡利弊。**Flink避免了这种权衡。**

Flink的一个重大价值在于，它既保证了exactly-once，也具有低延迟和高吞吐的处理能力。

从根本上说，Flink通过使自身满足所有需求来避免权衡，它是业界的一次意义重大的技术飞跃。尽管这在外行看来很神奇，但是一旦了解，就会恍然大悟。

端到端状态一致性

目前我们看到的一致性保证都是由流处理器实现的，也就是说都是在 Flink 流处理器内部保证的；而在真实应用中，**流处理应用除了流处理器以外还包含了数据源（例如 Kafka）和输出到持久化系统。**

> 端到端的一致性保证，**意味着结果的正确性贯穿了整个流处理应用的始终**；每一个组件都保证了它自己的一致性，整个端到端的一致性级别取决于所有组件中一致性最弱的组件。具体可以划分如下：

- 内部保证 —— 依赖checkpoint
- source 端 —— 需要外部源可重设数据的读取位置
- sink 端 —— 需要保证从故障恢复时，数据不会重复写入外部系统

而对于**sink端**，又有两种具体的实现方式：幂等（Idempotent）写入和事务性（Transactional）写入。

幂等写入

> 所谓幂等操作，是说一个操作，可以重复执行很多次，但只导致一次结果更改，也就是说，后面再重复执行就不起作用了。

事务写入

> 需要构建事务来写入外部系统，构建的事务对应着 checkpoint，等到 checkpoint 真正完成的时候，才把所有对应的结果写入 sink 系统中。

对于事务性写入，具体又有两种实现方式：**预写日志（WAL）和两阶段提交（2PC）**。DataStream API 提供了GenericWriteAheadSink模板类和TwoPhaseCommitSinkFunction 接口，可以方便地实现这两种方式的事务性写入。

不同 Source 和 Sink 的一致性保证可以用下表说明：

sink \ source	不可重置	可重置
任意（Any）	At-most-once	At-least-once
幂等	At-most-once	Exactly-once (故障恢复时会出现暂时不一致)
预写日志（WAL）	At-most-once	At-least-once
两阶段提交（2PC）	At-most-once	Exactly-once