14 | 状态管理：为什么说流计算是有“状态”的?

如果你曾经访问过 Flink 官网的话，你会看到 Flink 是这么描述它自己的：

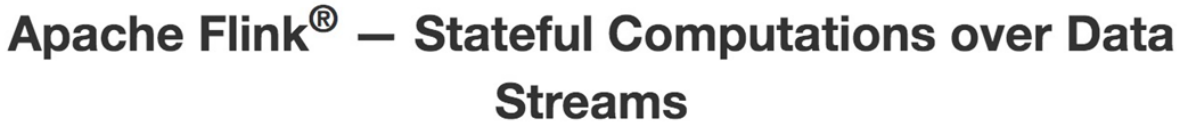

图1 Flink是基于流数据的有状态计算

看到没，第一个词就是**Stateful**（状态）！而且，正是因为这个 stateful，Flink 才会从众多开源流计算框架中脱颖而出，一下子就成为那个最靓的仔，还引得其他流计算框架开始纷纷效仿。

那为什么“状态”对于流计算是如此重要呢？其实关于这个问题，大家也是在不断的实践中，才逐渐弄明白。可以这样说，只有理解了“状态”，才能够真正理解“流计算”。

所以今天，我们就来详细讨论下实时流计算中有关于“状态”的问题吧！

流的状态区分

说到流计算的“状态”，我们在前面讲解流计算的五类算法时，其实就已经接触过了。比如，关联操作中临时保存的窗口数据、时间维度聚合值计算时使用的寄存器、CEP 中的有限状态机、统计或机器学习模型的参数等，这些都是“状态”。

但是，上面这些“状态”是有区别的！具体来说，这些“状态”可以分为两类，一类是**流数据状态，另一类是流信息状态**。下面我针对这两种状态分别解释下。

- 首先是**流数据状态**。在流计算过程中，我们需要处理事件窗口、时间乱序、多流关联等问题。解决这些问题，通常需要对部分流数据进行临时缓存，并在计算完成时再将这些临时缓存清理掉。因此，我们将这些**临时保存的部分流数据称为“流数据状态”**。
- 然后是**流信息状态**。在流计算过程中，我们会得到一些有用的业务信息，比如时间维度的聚合值、关联图谱的一度关联节点数、CEP 的有限状态机等，这些信息会在后续被继续使用，从而需要将它们保存下来。同时在之后的流计算过程中，这些信息还会被不断地查询和更新。因此，我们**将这些分析所得并保存下来的业务信息称为“流信息状态”**。

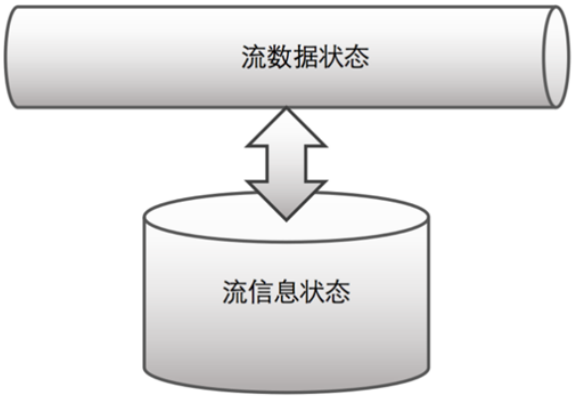

图2 流数据状态和流信息状态

为什么区分“流数据状态”和“流信息状态”非常重要呢？ 思考这么一个问题，如果要计算“用户过去 7 天交易总金额”，你准备怎么做？

一种显而易见的方法，是直接使用各种流计算框架提供的窗口函数来实现。比如在 Flink 中，就是如下代码：

```
userTransactions
.keyBy(0)
// 滑动窗口，每1秒钟计算一次7天窗口内的交易金额
.timeWindow(Time.days(7), Time.seconds(1))
.sum(1);
```

上面的 Flink 代码使用了 timeWindow 窗口，每 1 秒钟计算一次 7 天窗口内的总交易金额。其他流计算平台，像 Spark Streaming、Storm 等，也有类似的方法。
但你一定会发现，上面的实现似乎有些奇怪。到底哪里奇怪呢？我认为至少有以下几点非常不妥。

- 一是，这个计算是每 1 秒钟才能输出结果。如果我们的业务需求是，**每来一个事件**就对该事件所代表的用户，计算他在“过去 7 天交易的总金额”，那这种按照 1 秒钟做滑动窗口进行处理的方式，就与业务需求不相符了。
- 二是，窗口为 7 天，滑动步长为 1 秒，这两个时间相差的数量级过于巨大！这意味着对于同一份数据，在窗口滑动过程中，会被反复计算“7(天) ÷ 1(秒) ≈ 60万”次！当然，这里设置 1 秒是因为我们想尽可能地“实时”。如果觉得 1 秒太“过分”，你也可以设置滑动步长为 30 秒、60 秒等，但这并不能改变重复计算的本质，并且滑动步长越长，离“实时计算”越远。
- 三是，窗口为 7 天，就意味着在流计算系统中需要保存 7 天的数据。而我们想要计算的其实只是一个 sum 聚合值，保存 7 天的数据不仅会占用大量的内存和磁盘，还会显著降低处理速度。所以针对聚合值计算，还需要保存窗口内的全部数据，这显然是一个严重的问题。对于这个问题，一些流计算框架已经做了优化。比如 Flink 在计算 sum 一类的窗口聚合值时，默认是不用保存窗口内数据的，但是一旦用户需要做些定制化改动，则又退化到要保存窗口内全部数据的情况了。
- 四是，如果我们要在一个事件上，计算**几十个**类似于“用户过去 7 天交易总金额”这样的指标，按照 timeWindow 的实现方法，每个指标可能会有**不同的时间窗口**和**滑动步长**。如此一来，同步这几十个指标的计算过程并汇总它们的计算结果，也成了一件不容易的事情。

那么，是什么原因造成了以上这些不妥之处呢？这是因为，我们混淆了“数据窗口”和“业务窗口”。下面详细说明。

在大多数开源流计算框架中，它们定义的“窗口”函数，其实是针对“流数据”的“分块”管理。我们用这些“窗口”函数，将原本“无穷无尽”的流数据，分割成一个个的“数据块”，然后在“数据块”上做各种计算。这种做法，本质上是对流数据的“分而治之”管理，所以我将这种窗口，称之为“数据窗口”。

而在“用户过去 7 天交易总金额”的计算中，这里的“7 天”是属于业务意义上的时间窗口，所以我将这种窗口，称之为“业务窗口”。

澄清了这两种不同的窗口后，我们再来分析下前面的 Flink 代码有什么问题。

在前面的 Flink 代码中，我们直接使用 timeWindow(Time.days(7)，Time.seconds(1)) 来实现 “7 天”的时间窗口。这里的 Time.days(7) 属于 Flink 对“流数据”的“分块”管理，所以是“数据窗口”！我们将它设置为与“业务窗口”一致，其实是强行将“数据窗口”和“业务窗口”耦合起来。毕竟，我要实现“7 天”的“业务窗口”，为啥一定要用“7 天”的“数据窗口”来对流数据进行分块呢？

比如，我先将交易数据保存到数据库中，然后将“数据窗口”设置为 1 小时按批处理。或者，我甚至可以根本就不使用“数据窗口”，而是逐个事件处理。这两种方法，它们的“数据窗口”都不是“7 天”，但它们都可以算出用户在“7 天”的“业务窗口”内交易的总金额！

所以说，正是由于混淆了“数据窗口”与“业务窗口”，将它们强行耦和起来，才造成前面所说的各种不妥！

接下来，我们来做更进一步的分析。也就是，如果从“状态”的角度看，“数据窗口”和“业务窗口”又有什么区别？

在实现“数据窗口”时，我们工作的核心是**将部分流数据缓存起来**，并对缓存的数据按“窗口”划分，最后再将窗口内的数据按批次处理。所以，**实现“数据窗口”所要保存的“状态”是流数据本身**，这种“状态”就是“**流数据状态**”。

而在实现“业务窗口”时，我们工作的核心是**保存业务信息**，比如**每天的交易总金额**。这样当我们要计算 7 天的总交易金额时，只需要将这 7 天中**每天的交易总金额**读取出来汇总即可。所以，**实现“业务窗口”所要保存的“状态”是流数据所含的业务信息**，这种“状态”就是“**流信息状态**”。

经过上面的分析，你现在应该理解为啥要区分“流数据状态”和“流信息状态”了吧？**“流数据状态”和“流信息状态”根本就是两种不同的状态，“流数据状态”是在管理流数据过程中产生的状态，而“流信息状态”则是在分析流数据业务信息中产生的状态。**

所以接下来，我就详细地讲解下这两种状态。

流数据状态

我们先来看**流数据状态**。说到流数据状态，其最重要的用途是实现“事件窗口”“时间乱序处理”和“流的关联”。下面我就这三种用途逐一讲解下。

事件窗口

首先是“事件窗口”。

在模块 2 时，我们实现过一个流计算框架。在这个框架中，事件处理的方式是来一个就处理一个，并没有“窗口”的概念。但在实际很多场景中，我们并不需要每来一个事件就处理一个事件，而是按照一定的间隔和窗口来处理事件。比如“每 30 秒钟计算一次过去五分钟交易总额”“每满 100 个事件计算平均交易金额”“统计用户在一次活跃期间点击过的商品数量”等。

对于这些以“窗口”为单元来处理事件的方式，我们需要用一个缓冲区（buffer）临时存储过去一段时间接收到的事件。只有等到触发窗口计算的条件满足时，才开始处理窗口内的事件。最后当窗口里的数据被处理完时，还需要将以后无须再使用的数据清理掉。

可以看出，以上“事件窗口”功能得以实现，是非常依赖于在缓冲区中保存部分流数据的。这种保存在缓冲区中的部分流数据，就是一种“流数据状态”。

时间乱序

接下来是“时间乱序”。

由于网络传输和并发处理的原因，在流计算系统接收到事件时，非常有可能事件已经在时间上乱序了。比如时间戳为 1532329665005 的事件，比时间戳为 1532329665001 的事件先到达流计算系统。怎样处理这种事件在时间上乱序的问题呢？通常的做法就是将收到的事件先保存起来，**等过一段时间**后乱序的事件到达了，再将保存的事件按时间排序，这样就恢复了事件的时间顺序。

当然，上面的过程存在一个问题，就是“**等过一段时间**”到底是怎样等以及等多久？针对这个问题有一个非常优秀的解决方案，就是水印（watermark）。

使用“水印”解决时间乱序的原理是这样的。在流计算数据中，按照一定的规律（比如以特定周期）插入“水印”，水印是一个时间戳。当处理单元接收到水印时，表示应该**处理所有时间戳比水印小的事件**。我们通常将水印设置为“事件的时间戳**减去一段时间**”所得的值，这样就给“先到的时间戳较大的事件”一个等待“晚到的时间戳较小的事件”的机会，而且确保了不会没完没了地等待下去。在这个过程中，等待时间的大小就是那个**减去的时间段**了。

不过，“水印”这种方案也不是百分百地解决了乱序问题，那些实在太晚到达的事件，就只能是过期不候了。另外，由于解决“时间乱序”的问题需要等待晚到的事件，所以不可避免地会对当前事件的处理带来一定延迟。

总的来说，在使用“水印”解决“时间乱序”问题时，总是会将时间戳大于水印的数据先缓存起来。这些缓存的部分流数据，就是“流数据状态”。

流的关联

最后是“流的关联”。

在关系型数据库中，关联操作是一种非常普遍的行为，现在这种操作也被引进到流计算中来。比如 join、union 等，都是流计算中常用的关联操作。

虽然“流数据”和“块数据”在数据形式和处理方式上都十分不同，但就“关联”操作而言，它们的工作原理是非常相似的。

比如，流计算系统在实现 join 操作时，需要先将参与 join 的各个流在一段时间窗口内的全部数据都缓存起来。然后，以这些窗口内的数据为基础，做类似于关系型数据库中表与表之间的 join 计算。最后计算完成时，再将结果以流的方式输出。

很显然，上面实现流数据 join 操作时，也是需要临时保存部分流数据的，因此“流的关联”操作也用到了“流数据状态”。

除了以上三种主要用途外，流计算系统在实现其他一些功能时，也需要缓存部分的流数据，比如排序（sorting）、分组（group by）等。因此，这些功能也都使用到了“流数据状态”。

接下来，我们再看下有关“流数据状态”的存储问题。

流数据状态的存储

“流数据状态”最理想的情况是全部保存在内存中，只有在做持久化（checkpoint）时，才写入磁盘。这样做的原因在于，流数据从接收、处理到删除的过程，具有实时、快速和临时的特点，如果每次接收到一个新事件，都要将其写入磁盘，势必会引起性能的急剧下降。

但是，将所有数据全都放在内存，终究还是太过理想。因为大多数场景下，“数据窗口”内的数据量都超过了内存容量。所以，此时也可以将“流数据状态”存放在文件或其他外部存储系统中。这样会对性能造成一定的影响，但避免了内存对数据量的限制。

至此，我们就讨论完所有“流数据状态”相关的内容了。总的来说，“流数据状态”最重要的功能是实现窗口、关联和乱序处理。在后续模块四的课时中，我还会针对各种开源流计算框架，讲解它们对“流数据状态”的支持情况，这里就不再继续展开了。

流信息状态

接下来，我们再来看**流信息状态**。“流信息状态”最主要的功能，是记录在流计算过程中分析出的业务信息。

还是拿前面“用户过去 7 天交易总金额”的例子来说。在实时风控系统中，通常会要求针对每一个到来的交易事件，计算该交易事件的用户在过去 7 天交易的总金额。这种情况下，我们可以将每天的交易金额记录为一条“状态”。当一个交易事件到来时，只需要将过去 7 天每天的交易金额“状态”读取出来，然后汇总求和，就得到了 7 天的总交易金额。

在上面这个例子中，**将每天的交易金额记录为一条“状态”，就是我们说的“流信息状态”**。

流信息状态的存储，通常是依赖于数据库完成的。这有三方面的原因。

- 一是，“流信息状态”通常需要保存较长时间，数据量也不小，还需要频繁查询和更新，将它存放在数据库中，能方便地长期保存和增删查改。
- 二是，“流信息状态”存在“数据变冷”和“过期淘汰”的问题，使用数据库的“热数据缓存”和“TTL 机制”，能方便有效地解决这两个问题。
- 三是，“流信息状态”通常数据量会很大，单个存储节点往往是不够用的，选择合适的数据库能够方便地扩展为集群。

以上就是我们选择用数据库来存储“流信息状态”的原因了。

接下来我们就来看看，具体如何用数据库存储“流信息状态”吧！

使用 Apache Ignite 存储流信息状态

实际上，我们在 10 课时中，已经使用 Redis 数据库来存储“流信息状态”了。当时，我们为了计算“过去一周内在相同设备上交易的次数”，使用 Redis 的 INCR 指令来实现了计数功能，将记录交易次数信息的寄存器，存放在 Redis 中。这个过程其实就是在使用 Redis 存储“流信息状态”。

不过现在，我们使用另外一种不同的方案来存储“流信息状态”，这就是 Apache Ignite。这里之所以要“多此一举”地用两种不同的方案来实现相同的功能，也是有重要原因的。具体为什么，你在下一个课时就会知道了。

这里我重点讲解如何用 Apache Ignite 存储“流信息状态”。

Apache Ignite 是一种**基于内存的数据网格**方案，也可以用作分布式内存数据库。它提供了符合 JCache 标准的数据访问接口，支持丰富的数据结构，并且支持 SQL 查询功能。

为了更好地对比，我这里还是计算“过去一周在相同设备上交易次数”。下面就是具体的实现过程。

首先，我们需要定义一个用于存储“流信息状态”的“表”，这个“表”是数据存储的格式。具体定义如下：

```
class CountTable implements Serializable {
    @QuerySqlField(index = true)
    private String name;
    @QuerySqlField(index = true)
    private long timestamp;
    @QuerySqlField
    private double amount;
    public CountTable(String name, long timestamp, double amount) {
        this.name = name;
        this.timestamp = timestamp;
        this.amount = amount;
    }
    // 必需重写equals方法，否则在经过序列化和反序列化后，Ignite会视为不同记录，实际上它们是同一条记录
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        CountTable that = (CountTable) o;
        if (timestamp != that.timestamp) return false;
        if (Double.compare(that.amount, amount) != 0) return false;
        return name != null ? name.equals(that.name) : that.name == null;
    }
    // 因为重写了equals方法，所以hashCode()方法也跟着一起重写
    @Override
    public int hashCode() {
        int result;
        long temp;
        result = name != null ? name.hashCode() : 0;
        result = 31 * result + (int) (timestamp ^ (timestamp >>> 32));
        temp = Double.doubleToLongBits(amount);
        result = 31 * result + (int) (temp ^ (temp >>> 32));
        return result;
    }
}
```

在上面的表定义中：

- name 用于记录状态的关键字；
- timestamp 用于记录事件处理时的时间戳；
- amount 用于记录状态发生的次数。

另外，我重写了表 CountTable 类的 equals 方法和 hashCode 方法，这是因为 Apache Ignite 在执行 replace 这类方法时，会进行对象的比较。而由于 Apache Ignite 本身是一个分布式系统，在查询过程中会涉及对象序列化和反序列化的过程。这个时候如果不重写 equals 方法，会导致原本字段完全一样的记录会被视为不同记录，使得程序运行错误。

与用 Redis 时的实现思路完全一致，我也将 7 天的时间窗口划分为 7 个小窗口，每个小窗口代表 1 天。在每个小窗口内，分配一个用来记录这个窗口事件数的关键字，也就是 CountTable 表定义中的 name 字段，name 取值的格式如下：

```
$event_type.$device_id.$window_unit.$window_index
```

其中，“$event_type”表示事件类型，“$device_id”表示设备id，“$window_unit”表示时间窗口单元，“$window_index”表示时间窗口索引。

比如，对于“device_id”为“d000001”的设备，如果在时间戳为“1532496076032”的时刻更新窗口，则计算的伪代码如下：

```
$event_type = "transaction"
$device_id = "d000001"
$window_unit = 86400000  # 时间窗口单元为1天，即86400000毫秒
$window_index = 1532496076032 / $window_unit = 17737    # 用时间戳除以时间窗口单元，得到时间窗口索引
$atTime = ($window_index + 1) * $window_unit
$name = "$event_type.$device_id.$window_unit.$window_index"
$cache = ignite.getOrCreateCache()
$id = md5($name);
$newRecord = new CountTable($name, $atTime, 1);
do {
    $oldRecord = $cache.get($id);
    if ($oldRecord != null) {
        $newRecord.amount = $oldRecord.amount + 1;
    } else {
        $oldRecord = $newRecord;
        $cache.putIfAbsent($id, oldRecord);
    }
    $succeed = $cache.replace($id, $oldRecord, $newRecord);
} while (!$succeed);
```

上面的伪代码描述了使用 Apache Ignite 的 JCache 接口更新某个窗口计数的方法，它实现的功能与之前用 Redis 时并无二致。

但是，由于 Apache Ignite 没有提供类似于 Redis 中 INCR 指令那样的原子加操作，所以需要自行实现并发安全的累加操作。这里我没有采用“锁”的方案，而是使用了 CAS（Compare And Swap）。CAS 是一种无锁机制，在高并发场景下会比传统的锁具备更好的性能表现。在上面代码的“do while”循环部分，就是 CAS 的实现，其中“$cache.replace”是一个原子操作，保证了 CAS 的并发安全。

在更新完子窗口的计数后，就是查询完整窗口的总计数了。我们只需要对子时间窗口内的计数做查询并汇总即可。具体实现如下：

```
$event_type = "transaction"
$device_id = "d000001"
$window_unit = 86400000  # 时间窗口单元为1天，即86400000毫秒
$window_index = 1532496076032 / $window_unit = 17737    # 用时间戳除以时间窗口单元，得到时间窗口索引
$atTime = ($window_index + 1) * $window_unit
$startTime = $atTime - $window_unit * 7;    # 窗口为7天
$name = "$event_type.$device_id.$window_unit.$window_index"
$cache = ignite.getOrCreateCache()
$sumQuery = "SELECT sum(amount) FROM CountTable " +
                "WHERE name = $name and timestamp > $startTime and timestamp <= $atTime";
sum = $cache.query($sumQuery)
return sum
```

上面的伪代码充分利用了 Apache Ignite 支持 SQL 查询的便利性，很容易地计算出过去 7 天交易的总次数。

至此，我们就使用 Apache Ignite 实现了“过去一周在相同设备上交易次数”的计算。

总的来说，对于“流信息状态”，我们通常是使用数据库来进行管理，比如 Redis、Apache Ignite、RocksDB 等。我们一般还会要求这些数据库是高性能的，并且可以扩展为集群。对于扩展为集群这点，我会在下一个课时重点讨论。

小结

今天，我们讨论了实时流计算应用中状态管理的问题，并将实时流计算应用中的状态分为了“流数据状态”和“流信息状态”。

可以说，“流数据状态”和“流信息状态”分别是从两个不同的维度对“流”进行的管理。**前者“流数据状态”是从“时间”角度对流进行管理，而后者“流信息状态”则是从“空间”角度对流进行管理。**“流信息状态”弥补了“流数据状态”只是对事件在时间序列上做管理的不足，将流的状态扩展到了任意的空间。

将“流数据状态”和“流信息状态”这两个概念区分开，会指引我们将“流计算应用本身的执行过程”和“流数据的信息管理机制”解耦，这使得实时流计算系统的整体结构更加清晰。

今天的课程内容稍微偏长了，这是因为“状态”问题对于流计算系统来说，实在是太过重要的内容。它是你以后能否灵活运用各种流计算框架解决业务问题的关键所在，所以请你务必掌握好今天的内容。

最后，我们留一个小作业。对于“过去 1 小时成交总金额”和“过去 3 个月成交总金额”这两个计算任务，用 Flink 的话你会分别怎样实现呢？可以将你的想法或问题写在留言区。

下面是本课时的知识脑图，以便于你理解。