31 | Dataflow（三）：一个统一的编程模型

2021-12-22 徐文浩

《大数据经典论文解读》 课程介绍 >

讲述：徐文浩
时长 19:02 大小 17.45M

你好，我是徐文浩。

在过去的几讲里，我们看到了大数据的流式处理系统是如何一步一步进化的。从最早出现的 S4，到能够做到“至少一次”处理的 Storm，最后是能够做到“正好一次”数据处理的 MillWheel。你应该能发现，这些流式处理框架，每一个都很相似，它们都采用了有向无环图一样的设计。但是在实现和具体接口上又很不一样，每一个框架都定义了一个属于自己的逻辑。

S4 是无中心的架构，一切都是 PE；Storm 是中心化的架构，定义了发送数据的 Spout，处理数据的 Bolt；而 MillWheel 则更加复杂，不仅有 Computation、Stream、Key 这些有向无环图里的逻辑概念，还引入了 Timer、State 这些为了持久化状态和处理时钟差异的概念。

和我们在大数据的批处理看到的不同，S4、Storm 以及 MillWheel 其实是某一个数据处理系统，而不是 MapReduce 这样高度抽象的编程模型。每一个流式数据处理系统各自有各自对于问题的抽象和理解，**很多概念不是从模型角度的“该怎么样”抽象出来，而是从实际框架里具体实现的“是怎么样”的角度，抽象出来的。**

不过，我们也看到了这些系统有很多相似之处，它们都采用了有向无环图模型，也都把同一个 Key 的数据在逻辑上作为一个单元进行抽象。随着工业界对于流式数据处理系统的不断研发和运用，到了 2015 年，仍然是 Google，发表了今天我们要解读的这一篇《The Dataflow Model》的论文。

那么，在学完这一讲之后，我希望你能够对过去几讲的论文进一步融会贯通，能够做到：

从抽象模型的角度来理解流式数据处理，而不仅仅是从系统框架如何实现的角度，思考流式数据处理。

掌握 Dataflow 里 ParDo、GroupByKey，以及窗口、触发器和增量数据处理这些概念。

能够把这些概念和之前学习过的 S4、Storm，以及 MillWheel 系统联系起来，思考如何在这些系统上扩展，以实现 Dataflow 的编程模型。

Dataflow 的基础模型

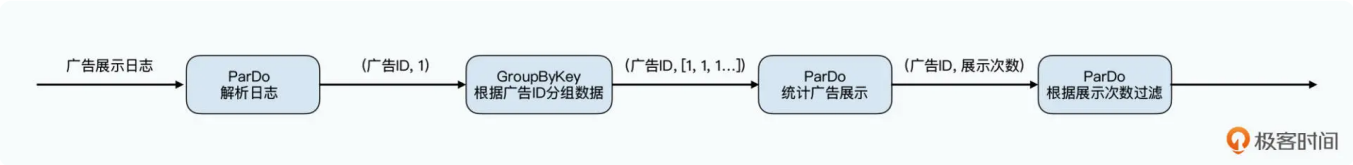
Dataflow 的核心计算模型非常简单，它只有两个概念，一个叫做 ParDo，顾名思义，也就是并行处理的意思。另一个叫做 GroupByKey，也就是按照 Key 进行分组数据处理的问题。

ParDo，地位相当于 MapReduce 里的 Map 阶段。所有的输入数据，都会被一个 DoFn，也就是处理函数处理。但是这些数据，不是在一台服务器上处理的，而是和 MapReduce 一样，会在很多台机器上被并行处理。只不过 MapReduce 里的数据处理，只有一个 Map 阶段和一个 Reduce 阶段。而在 Dataflow 里，Pardo 会和下面的 GroupByKey 组合起来，可以有很多层，就好像是很多个 MapReduce 串在一起一样。

而 GroupByKey，地位则是 MapReduce 里的 Shuffle 操作。在 Dataflow 里，所有的数据都被抽象成了 key-value 对。前面的 ParDo 的输入和 Map 函数一样，是一个 key-

value 对，输出也是一系列的 key-value 对。而 GroupByKey，则是把相同的 Key 汇总到一起，然后再通过一个 ParDo 下的 DoFn 进行处理。

比如，我们有一个不断输入的日志流，想要统计所有广告展示次数超过 100 万次的广告。那么，我们可以先通过一个 Pardo 解析日志，然后输出（广告 ID，1）这样的 key-value 对，通过 GroupByKey，把相同的广告 ID 的数据分组到一起。然后再通过一个 ParDo，并行统计每一个广告 ID 下的展示次数。最后再通过一个 ParDo，过滤掉所有展示次数少于 100 万次的广告就好了。



Dataflow的编程模型，就是一系列ParDo和GroupByKey串接在一起

理解流批一体

那么这样看起来，Dataflow 不就是个 MapReduce 吗？它无非是可以把多个 MapReduce 的过程串接在一起就是了。当然，答案并没有那么简单，因为在 Dataflow 里，我们还有一个很重要的维度没有加入进来，这个维度就是**时间**。

Dataflow 里的 GroupByKey，会把相同 Key 的数据 Shuffle 到一起供后续处理，但是它并没有定义在什么时间，这些数据会被 Shuffle 到一起。

在 MapReduce 的计算模型下，会有哪些输入数据，是在 MapReduce 的任务开始之前就确定的。这意味着数据从 Map 端被 Shuffle 到 Reduce 端，只依赖于我们的 CPU、网络这些硬件处理能力。而在 Dataflow 里，输入的数据集是无边界的，随着时间的推移，不断会有新的输入数据加入进来。

如果从这个角度来思考，那么我们之前把大数据处理分成批处理和流式处理，其实并没有找到两种数据处理的核心差异。因为，**对于一份预先确定、边界明确的数据，我们一样可以使用流式处理。**比如，我们可以把一份固定大小日志，放到 Kakfa 里，重放一遍给一个 Storm 的 Topology 来处理，那也是流式处理，但这是处理的有边界的数据。

而对于不断增长的实时数据，我们一样可以不断定时执行 MapReduce 这样的批处理任务，或者通过 Spark Streaming 这样看起来是流式处理，其实是**微批**（Mini-Batch）的处

理方式。

事实上，即使是所谓的“流式”数据处理系统，往往也会为了性能考虑，通过微批的方式来提升性能。一个典型的例子，就是上一讲我们看过的 MillWheel 里的 Checkpoint，就会在等待多条记录处理完之后批量进行。

一旦从这个视角来观察，那么批和流本身是一回事儿。当我们把“批（Batch）”的记录数限制到了每批一条，那么它就是所谓的流了。进一步地，MapReduce 的“有边界（Bounded）”的数据集，也只是 Dataflow 的“无边界（Unbounded）”的数据集的一种特殊情况。所以，Jay Kreps 才会在 2014 年提出流批一体的 Kappa 架构，而到了 2015 年的 Dataflow，我们就看到了批处理本来就是流处理的一种特殊情况。

时间窗口的分配与合并

在 MillWheel 的论文里，我们已经看到了一个非常完善的流式数据处理系统了。不过，在这个流式处理系统里，对于“时间”的处理还非常粗糙。MillWheel 的确已经开始区分事件的处理时间（Processing Time）和事件的发生时间（Event Time）了，也引入了时间窗口的概念。但是，对于计算结果何时输出，它仍然采用的是一个简单的定时器（Timer）的方案。而到了 Dataflow 论文里，对这些概念的梳理和抽象就变成了重中之重。

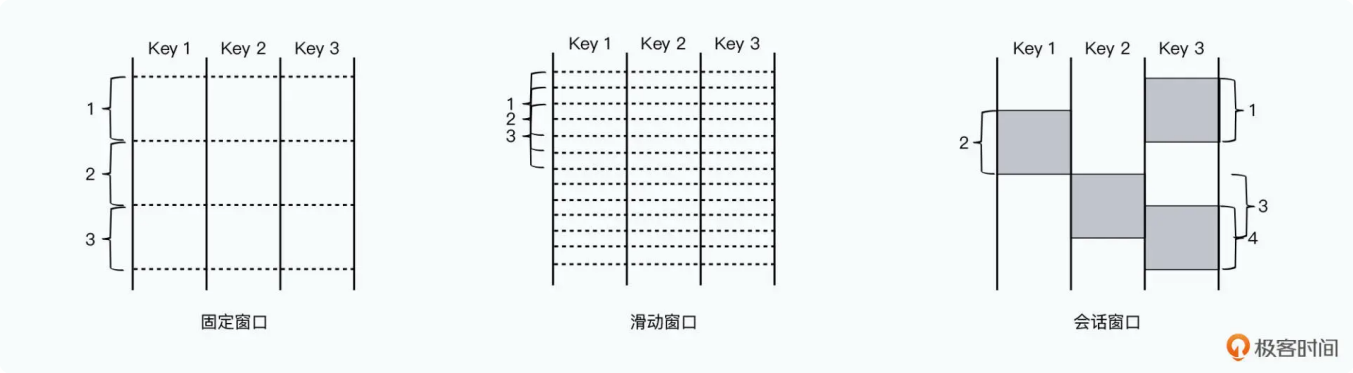
我们先来看一看时间窗口的概念，在流式数据处理里，我们需要的往往不是“统计所有的广告展示数量”，而往往是“每 5 分钟统计一次广告展示数量”，或者“统计过去 5 分钟的广告展示数量”。我们常用的时间窗口，也会分成好几种：

首先是**固定窗口**（Fixed Window）。比如，我们统计“每小时的广告展示数量”，那么我们的数据，就会被划分成 0 点到 1 点、1 点到 2 点，这样一个个固定区间的窗口。

然后是**滑动窗口**（Sliding Window），也就是窗口随着时间的变动在“滑动”。比如，我们要统计“过去 2 分钟的广告展示”，那么我们的窗口并不是划分成 12:00~12:02，12:02~12:04 这样一段段。而是 12:00~12:02，然后一分钟之后变成 12:01~12:03，在这个例子里，2 分钟被称之为**窗口大小**，而窗口每 1 分钟“滑动”一次，这个 1 分钟被称之为**滑动周期**。

最后是**会话窗口**（Session Window）。这个常常用在统计用户的会话上，对于会话的划分，往往是通过我们设置的两次事件之间的一个“超时时间”来定义的。比如，我们有一个客服聊天系统，如果用户和客服之间超过 30 分钟没有互动，我们就认为上一次

会话结束了。在这之后无论是用户主动发言，还是客服主动回复，我们都会认为是进入了一个新的会话。



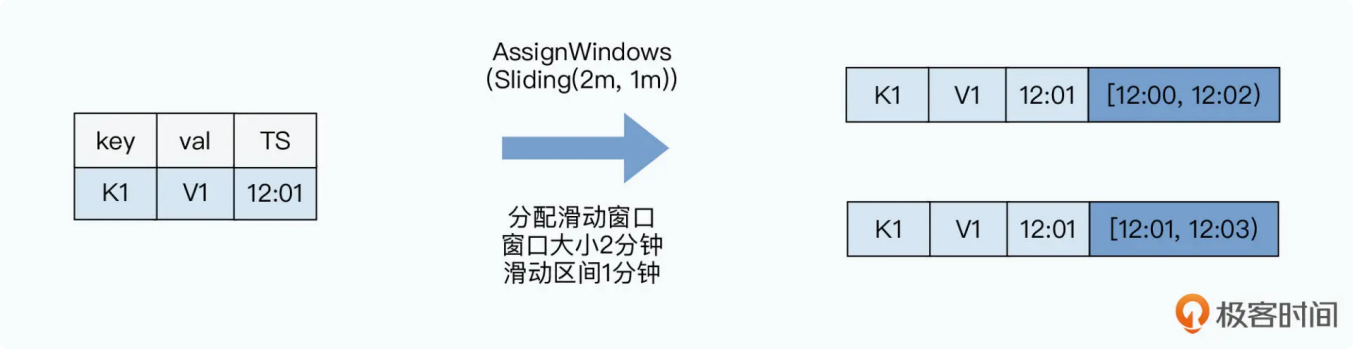
论文中的图1，不同的窗口类型

既然引入了时间窗口这个概念，相信你很容易理解，我们在 Dataflow 模型里，需要的不只是 GroupByKey，实际在统计数据的时候，往往需要的是 **GroupByKeyAndWindow**。统计一个不考虑任何时间窗口的数据，往往是没有意义的，1 分钟内广告展示了 100 万次，和 1 个月内展示了 100 万次代表着完全不同的广告投放力度。**我们需要根据特定的时间窗口，来进行数据统计。**

而在实际的逻辑实现层面，Dataflow 最重要的两个函数，也就是 **AssignWindows 函数和 MergeWindows 函数**。每一个原始的事件，在我们的业务处理函数之前，其实都是（key, value, event_time）这样一个三元组。而 AssignWindows 要做的，就是把这个三元组，根据我们的处理逻辑，变成（key, value, event_time, window）这样的四元组。

需要注意，一个事件不只可以分配给一个时间窗口，而是可以分配给多个时间窗口。比如，我们有一个广告在 12:01 展示给了用户，但是我们统计的是“过去 2 分钟的广告展示”，那么这个事件，就会被分配给[12:00, 12:02) 和[12:01, 12:03) 两个时间窗口，我们原先一条的事件就可以变成多条记录。

而在有了 Window 的信息之后，如果我们想要按照固定窗口或者滑动窗口统计数据，我们可以很容易地根据 Key+Window 进行聚合，完成相应的计算。



分配窗口的时候，可能会生成多条记录，你也可以去查看一下论文里的图3

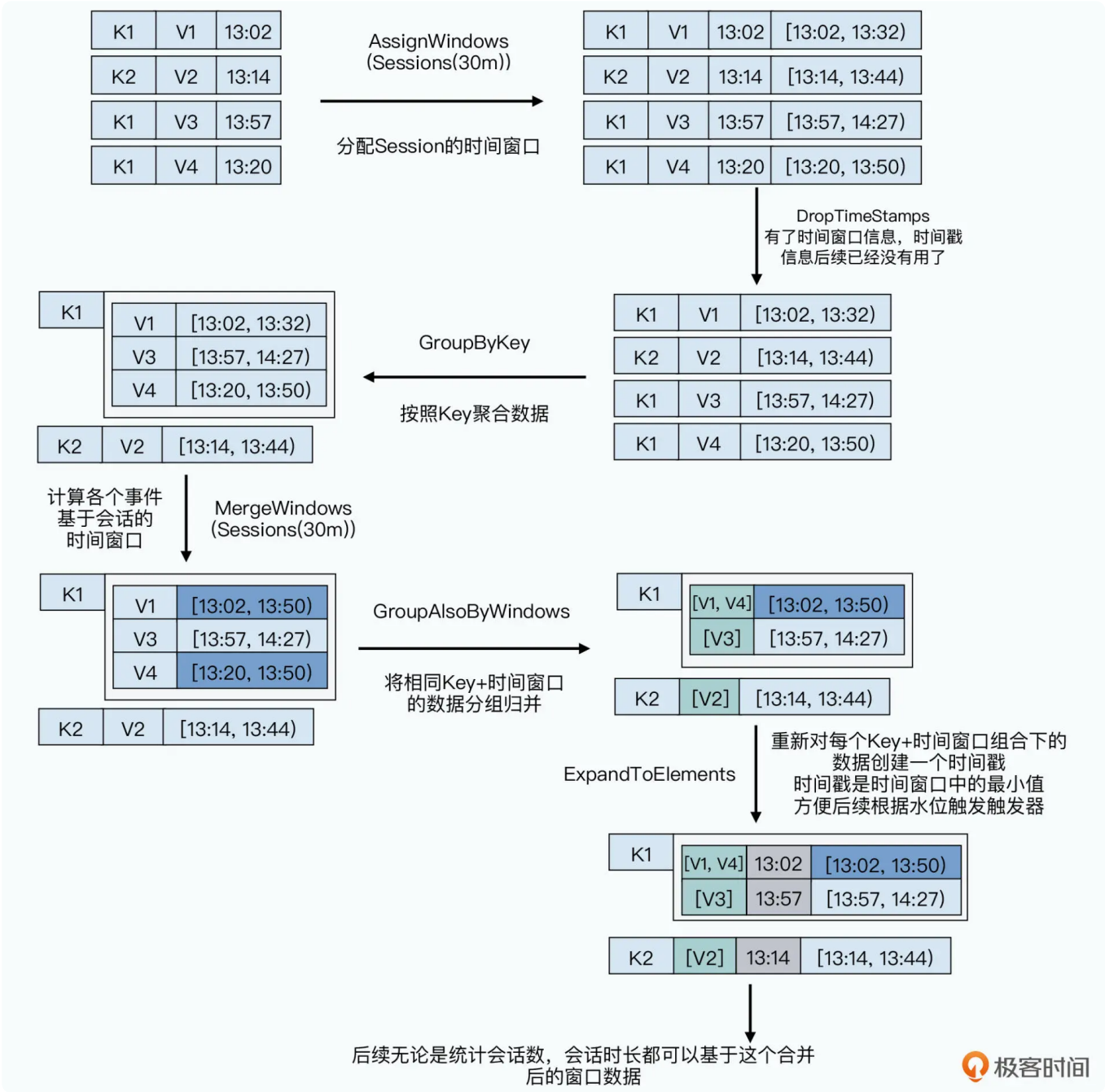
但是，有些窗口函数的计算并不容易，比如我们前面讲过的第三种会话窗口，每个事件的发生时间都是不一样的。那么这个时间窗口就很难定义。

而 Dataflow 里的做法，是通过 **AssignWindows+MergeWindows** 的组合，来进行相应的数据统计。我们还是以前面说的，客服 30 分钟没有互动就算作超时的例子来看看。

> 因为我们要根据同一个用户的行为进行分析，所以 Key 自然是用户 ID 了。那么对应的 Value 里，我们可以记录是用户发送的消息还是客服回复的消息，以及对应的消息内容。而 event_time，则是实际消息发送的时间。
>
> 对于每一个事件，我们进行 AssignWindows 的时候，都是把对应的时间窗口，设置成 $[event_time, event_time + 30)$。也就是事件发生之后的 30 分钟超时时间之内，都是这个事件对应会话的时间窗口。
>
> 而在同一个 Key 的多个事件，我们可以把这些窗口合并。对于会话窗口，如果两个事件的窗口之间有重合部分，我们就可以把它们合并成一个更大的时间窗口。而如果不同事件之间的窗口没有重合，那么这两个事件就还是两个各自独立的时间窗口。在所有的事件合并完成之后，我们只需要去数有几个时间窗口，就能知道有几个会话了。

比如同一个用户下，有三个事件，发生的时间分别是 13:02、13:14、13:57。那么分配窗口的时候，三个窗口会是 $[13:02, 13:32)$，$[13:14, 13:44)$ 以及 $[13:57, 14:27)$。□前两个时间窗口是有重叠部分的，但是第三个时间窗口并没有重叠，对应的窗口会合并成 $[13:02, 13:44)$ 以及 $[13:57, 14:27)$ 这样两个时间窗口。

窗口的分配和合并功能，就使得 Dataflow 可以处理乱序数据。相同的数据以不同的顺序到达我们的计算节点，计算的结果仍然是相同的。并且在这个过程里，我们可以把上一次计算完的结果作为状态持久化下来，然后每一个新进入的事件，都按照 AssignWindows 和 MergeWindows 的方式不断对数据进行化简。



论文中的图5，如何通过AssignWindows和MergeWindows来进行数据计算，数据乱序也不影响计算结果

你可以来看下论文里的图 5，这个图有助于你去理解 Dataflow 是如何通过它的一些基础操作，来完成对应的数据化简和统计的。

触发器和增量数据处理

这样一来，有了对应的窗口函数逻辑，如果我们的输入数据是确定的，能够一次性都给出来，我们就很容易统计会话数这样的数据了，即使数据是乱序的也没有关系。但是，在实际情况里，我们的输入数据是以流的形式传输到每个计算节点的。并且，我们会遇到延时、容错等情况，所以我们还需要有一个机制告诉我们，在什么时候数据都已经到了，我们可以把计算结果向下游输出了。

在 MillWheel 的论文里，我们是通过计算一个低水位（Low Watermark）来解决这个问题的。我们会根据获取到的低水位信息，判断是否该处理的事件都已经处理完了，可以把计算结果向下游发送。

但是，这个基于水位的方法在实践中，必然会遇到这样两个问题：

第一个，**在实际的水位标记之后，仍然有新的日志到达**。比如，水位信息告诉我们最早的还没有处理的日志是 12:01 的，那么我们自然可以把 12:00 的统计数据发出去。但是，很有可能一分钟后，我们收到了一条 12:00 的日志数据，这是因为之前某一个节点挂掉了，恢复传输花了一些时间。那么在这种情况下，我们已经往下游发送的数据就是不准确的。而这种情况，对于数据准确性要求高的需求来说，比如广告计费，就让人难以接受。

第二个，我们的水位标记，因为需要考虑所有节点。**只要有一条日志来晚了，我们的水位就会特别“低”，导致我们迟迟无法输出计算结果**。比如，虽然已经到了 12:00 了，但是我们就是偶尔会出现一条 11:05 的日志，那么我们的水位一直会卡在 11:05，计算结果就会迟迟不能向下游发送。

那么，Dataflow 里，是怎么解决这个问题的呢？答案是 Lamdba 架构。

这里的 Lambda 架构，并不是需要去搭建一个数据的批处理层，而是利用 Nathan Marz 的 Lambda 架构的核心思想，就是我们可以尽快给出一个计算结果，但是在后续根据获得的新的数据，不断去修正这个计算结果。而这个思路，在 Dataflow 里，就体现为**触发器**（Trigger）机制。

在 MillWheel 里，我们向下游输出数据，只能通过定时器（Timer）来触发，本质上也就是通过“时间”这一个维度而已。这个定时器，在 Millwheel 里其实就被改造成了完成度触发器，我们可以根据当前的水位和时间，来判断日志处理的进度进而决定是否触发向下游输出的动作。而在 Dataflow 里，除了内置的基于水位信息的完成度触发器，它还能够支持基于处理时间、记录数等多个参数组合触发。而且用户可以实现自定义触发器，完全根据自己的需要来实现触发器逻辑。

复制代码

```
PCollection<String> pc = ...;
pc.apply(Window.<String>into(FixedWindows.of(1, TimeUnit.MINUTES))
   .triggering(AfterProcessingTime.pastFirstElementInPane()
```

```
4    .plusDelayOf(Duration.standardMinutes(1)))
5    .discardingFiredPanes());
```

> 来自 Apache Beam 的文档教程。

我们可以看一下 Apache Beam 项目里的一段示例代码。可以看到，在这段代码里，先是设立了一个 1 分钟的固定窗口。然后在触发器层面，则是设置了在对应的窗口的第一条数据被处理之后，延迟一分钟触发。在 Apache Beam 的文档里，你还能看到更多不同的触发器策略，你也可以根据自己的需要，来撰写专属于你自己的触发器代码。

而除了确定对应的数据计算什么时候触发，你还可以定义触发之后的输出策略是什么样的。

首先是**抛弃（Discarding）策略**，也就是触发之后，对应窗口内的数据就被抛弃掉了。这意味着后续如果有窗口内的数据到达，也没法和上一次触发时候的结果进行合并计算。但这样做的好处是，每个计算节点的存储空间占用不会太大。一旦触发向下游输出计算结果了，现有的数据我们也就不需要了。比如，一个监控系统，根据本地时间去统计错误日志的数量并告警，使用这种策略就会比较合适。

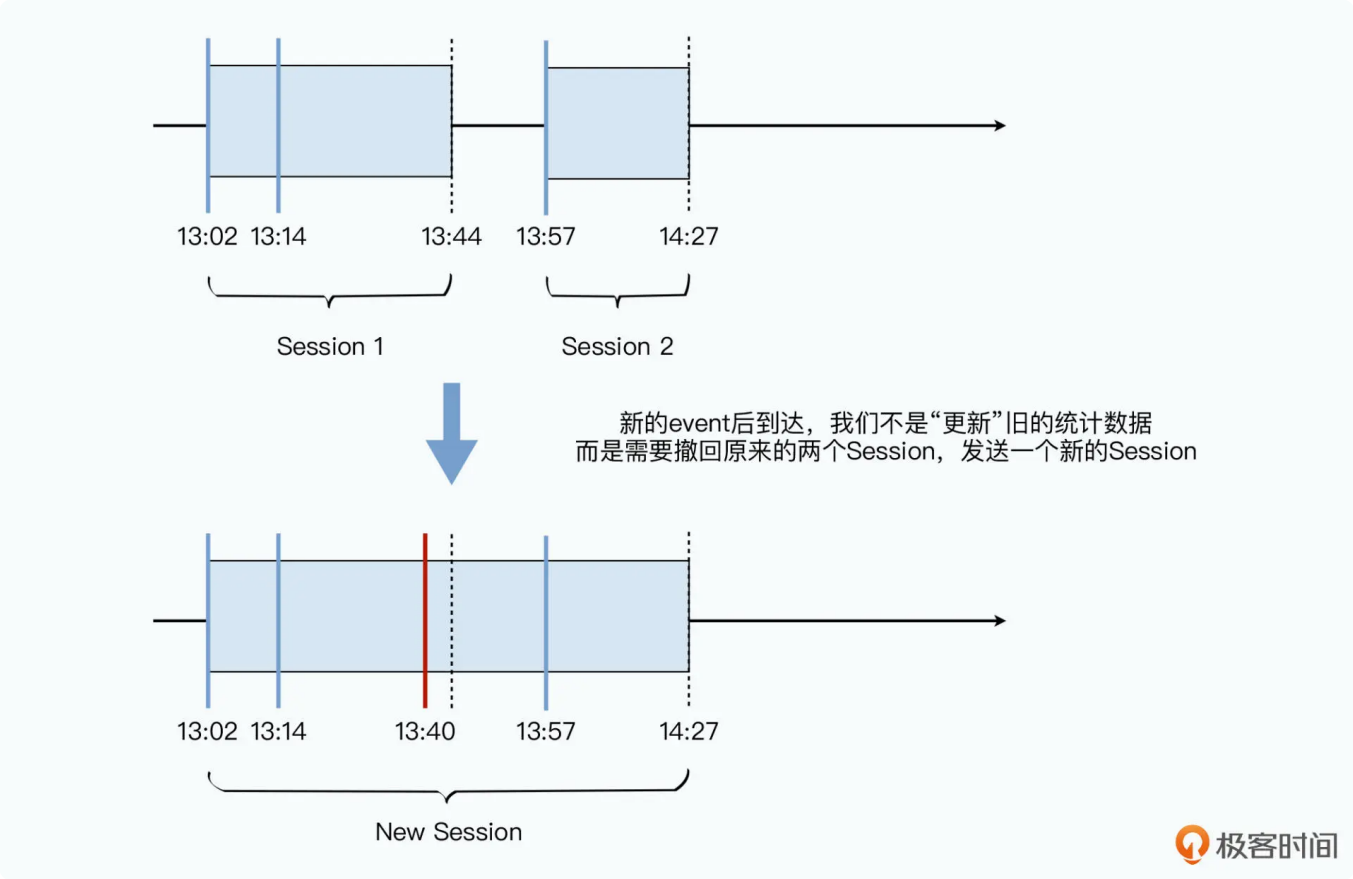
然后是**累积（Accumulating）策略**，也就是触发之后，对应窗口内的数据，仍然会持久化作为状态保存下来。当有新的日志过来，我们仍然会计算新的计算结果，并且我们可以再次触发，向下游发送新的计算结果，而下游也会用新的计算结果来覆盖掉老的计算结果。

这个是一个典型的 Lambda 架构的思路。我们一般的统计数据，都可以采用这个策略。一方面，我们会尽快根据水位信息，把计算结果发送给下游，使得计算结果的延时尽可能得小。另一方面，在有新的数据过来的时候，我们也会重新修正计算结果。

最后是**累积并撤回（Accumulating & Retracting）策略**，也就是我们除了“修正”计算结果之外，可能还要“撤回”计算结果。还是以前面的客服会话为例：

> 原本我们先收到了三个事件，13:02、13:14、13:57，根据 30 分钟的会话窗口，我们的逻辑计算完，窗口就变成了 $[13:02, 13:44)$ 以及 $[13:57, 14:27)$ 这样两个时间窗口。并且，这两个会话分别作为两条记录，向下游的不同计算节点下发了。

这个时候，我们又接收到了一条姗姗来迟的新日志，日志的时间是 13:40。那么，根据我们的业务逻辑，这个用户其实只有一个会话 $[13:02, 14:27)$。所以，我们不仅要向下游发送一个新会话出去，还需要能够“撤回”之前已经发送的两个错误的会话。



当然，这只是我们最理想的状况，抛弃和累积这两种策略并不难实现，但是累积并撤回并不容易实现，即使在 2021 年的今天，Apache Beam 也还没有支持撤回（Retraction）功能。不过，即使你从没有使用过对应的功能，你也需要理解为什么我们需要这样的功能。因为没有这个功能的话，我们的计算结果的正确性，在有些情况下是保障不了的。从这个角度来看，Lambda 架构仍未彻底过时。

小结

好了，到这里，我们也算是为整个课程里，大数据的流式处理画上一个句号了。随着时代洪流滚滚向前，Google 也针对自己发表的 Dataflow 这个编程模型，孵化出了 Apache Beam 这个项目。而在这个时间节点之后，像 Apache Flink 这样的开源流式处理项目，也都向 Dataflow 的编程模型靠拢，并实现了 Apache Beam 的接口。

在 Dataflow 的论文里，Google 把整个大数据的流式处理，抽象成了三个概念。第一个，是对于乱序数据，能够按照事件发生时间计算时间窗口的模型。第二个，是根据数据处理

的多维度特征，来决定计算结果什么时候输出的触发器模型。第三个，则是能够把数据的更新和撤回，与前面的窗口模型和触发器模型集成的增量处理策略。

Dataflow 不是一篇介绍具体系统实现的论文，而是一篇更加高屋建瓴，**从模型角度思考无边界的大数据处理应该如何抽象**的论文。

像 MapReduce 一样，Dataflow 是一个抽象的计算模型而不是一个具体的系统实现。用 MapReduce 的时候，你并不需要 Google 的 C++ 的原版实现，而完全可以用 Java 写的 Hadoop。而在 Dataflow 这里，Google 更进一步，不仅给出了整个的计算模型，后续还推动了 Apache Beam 这个项目，希望能让流式数据处理的接口统一。无论你的底层实现是什么，只要能够按照 Dataflow 里的语义实现对应的接口，那么就算是别人来撰写代码，也都可以实现相同的计算结果。

推荐阅读

想要对大数据的流式处理有深入的了解，我们必须要读的一本书就是《Streaming Systems》。这本书的作者泰勒·阿克道（Tyler Akidau）也是这篇 Dataflow 论文的作者。我在之前的几讲里，已经推荐过这本书了，目前国内也已经出版了影印本。如果你想深入大数据领域的研发，特别是流式数据处理这个领域，这本书你一定要买回来好好研读一下。

思考题

我们说，Dataflow 已经是一个流批一体的计算模型了，有边界的数据也只是无边界的流式数据的一种特殊情况。对于有边界的固定数据，我们当然可以通过重放日志把数据给到 Dataflow 系统。那么在窗口和触发器层面，我们应该用什么窗口和触发器，来得到我们想要的计算结果呢？

欢迎在留言区分享你的答案和思考，也欢迎你把今天的内容分享给更多的朋友。

赞 1 提建议

上一篇 30 | Dataflow（二）：MillWheel，一个早期实现

下一篇 复习课（一）| The Google File System

精选留言 (2)

写留言

那时刻

2021-12-22

对于有边界的固定数据，我们当然可以通过重放日志把数据给到 Dataflow 系统， 我之前采用的是global window以及default trigger来处理的

共 1 条评论 > 1

在路上

2021-12-24

徐老师好，DataFlow论文第3.2节Design Principles，提到Support robust analysis of data in the context in which they occurred，数据的健壮分析是指什么？