



尚硅谷 Flink 教程

作者：左元

组织：尚硅谷大数据组

时间：April 24, 2021

版本：3.1

Flink 版本：1.12



笔 记

[1] ETL：将数据复制到数据仓库的过程 (extract-transform-load)

目录

1 有状态的流式处理	1
1.1 传统数据处理架构	1
1.1.1 事务处理	1
1.1.2 分析处理	2
1.2 有状态的流式处理	3
1.2.1 事件驱动应用程序	4
1.2.2 数据管道	5
1.2.3 流分析	6
1.3 开源流处理的演进	7
1.4 Flink 简介	8
1.5 编写第一个 Flink 程序	9
1.5.1 在 IDEA 中编写 Flink 程序	9
1.5.2 下载 Flink 运行时环境, 提交 Jar 包的运行方式	10
2 流处理基础	11
2.1 数据流编程	11
2.1.1 数据流图	11
2.1.2 数据并行和任务并行	11
2.1.3 数据交换策略	12
2.2 并行处理流数据	12
2.2.1 延迟和吞吐量	13
2.2.2 延迟	13
2.2.3 吞吐量	13
2.2.4 延迟与吞吐量的对比	14
2.3 数据流上的操作	14
2.3.1 数据摄入和数据吞吐量	15
2.3.2 转换算子	15
2.3.3 滚动聚合	15
2.3.4 窗口操作符	15
2.4 时间语义	18
2.4.1 在流处理中一分钟代表什么?	18
2.4.2 处理时间	19
2.4.3 事件时间	20
2.4.4 水位线	20
2.4.5 处理时间和事件时间	21

2.5 状态和持久化模型	21
2.5.1 任务失败	22
3 Flink 运行架构	25
3.1 系统架构	25
3.1.1 Flink 运行时组件	25
3.1.2 应用部署	26
3.1.3 任务执行	27
3.1.4 高可用配置	28
3.2 Flink 中的数据传输	29
3.2.1 基于信任度的流控制	29
3.2.2 任务链	30
3.3 事件时间处理	31
3.3.1 时间戳	31
3.3.2 水位线	31
3.3.3 watermark 的传递和事件时间	32
3.3.4 时间戳的分配和水位线的产生	34
3.4 状态管理	35
3.4.1 算子状态	36
3.4.2 键控状态	36
3.4.3 状态后端	37
3.4.4 调整有状态算子的并行度	38
3.5 检查点, 保存点和状态恢复	40
3.5.1 一致的检查点	40
3.5.2 从一致检查点中恢复状态	41
3.5.3 Flink 的检查点算法	42
3.5.4 检查点的性能影响	45
3.5.5 保存点	46
4 Flink DataStream API	48
4.1 读取数据源	48
4.1.1 定义 POJO 类	48
4.1.2 从集合中读取	48
4.1.3 从文件中读取	49
4.1.4 从 socket 读取	49
4.1.5 从 Kafka 读取	49
4.1.6 自定义数据源读取	50
4.2 基本转换算子的使用	51
4.3 键控流转换算子	52

4.4 分布式转换算子	54
4.5 设置并行度	56
4.6 Flink 的类型系统	57
4.6.1 支持的数据类型	57
4.6.2 为数据类型创建类型信息	59
4.6.3 Java Lambda 表达式	59
4.7 富函数	62
4.8 写入下游设备	63
4.8.1 写入文件	63
4.8.2 写入 Kafka	63
4.8.3 写入 Redis	64
4.8.4 写入 ElasticSearch	65
4.8.5 写入 MySQL	66
4.8.6 自定义 SINK	67
5 基于时间和窗口的操作符	69
5.1 水位线的设定	69
5.1.1 水位线的一般设置方案	69
5.1.2 自定义水位线的产生逻辑	70
5.1.3 水位线总结	72
5.2 设置时间属性	73
5.2.1 处理时间设定	74
5.2.2 事件时间设定和水位线的产生	74
5.2.3 窗口聚合函数的使用	75
6 状态变量的初步使用	82
7 处理函数	83
7.1 KeyedProcessFunction 的使用	83
7.2 CoProcessFunction 的使用	86
7.3 基于时间的双流 Join	88
7.3.1 基于间隔的 Join	88
7.3.2 基于窗口的 Join	90
7.4 处理迟到的元素	93
7.4.1 抛弃迟到元素	93
7.4.2 重定向迟到元素	93
7.4.3 使用迟到元素更新窗口计算结果	96

8 有状态算子和应用	99
8.1 实现有状态的用户自定义函数	99
8.2 状态后端	101
8.2.1 选择一个状态后端	101
8.2.2 防止状态泄露	101
8.2.3 配置检查点	102
9 读写外部系统	103
9.1 应用的一致性保证	103
9.1.1 幂等性写入	104
9.1.2 事务性写入	104
9.1.3 Flink 提供的连接器	105
9.1.4 Apache Kafka Source 连接器	105
9.1.5 Apache Kafka Sink 连接器	107
9.1.6 Kafka Sink 的 at-least-once 保证	107
9.1.7 Kafka Sink 的恰好处理一次语义保证	107
9.1.8 Redis Sink 连接器	108
9.1.9 ElasticSearch Sink 连接器	109
9.2 实现自定义源函数	110
9.2.1 可重置的源函数	111
9.3 实现自定义 sink 函数	113
9.3.1 幂等 sink 连接器	114
9.3.2 事务性 sink 连接器	116
10 搭建 Flink 运行流式应用	121
10.1 独立集群部署	121
10.2 YARN 部署	122
10.3 高可用部署	124
10.4 保存点操作	124
10.5 取消一个应用	124
10.6 扩容, 改变并行度操作	125
11 Flink CEP 简介	126
12 Flink Table API & SQL	133
12.1 什么是 Table API 和 Flink SQL	133
12.2 需要引入的依赖	133
12.3 概念与通用 API	134
12.3.1 Table API 和 SQL 程序的结构	134
12.3.2 创建 TableEnvironment	134

12.3.3 在 Catalog 中创建表	136
12.3.4 查询表	138
12.3.5 输出表	139
12.3.6 与 DataStream 和 DataSet API 结合	140
12.4 流式概念	146
12.4.1 动态表 (Dynamic Table)	146
12.4.2 时间属性	150
12.5 例子	153
12.5.1 分流求平均值	153
12.5.2 求窗口温度平均值	154
12.5.3 追加流	155
12.5.4 事件时间例子	156
13 案例集	158
13.1 数据倾斜如何解决	158
13.2 独立访客计算	160
13.2.1 原始版	160
13.2.2 改进版	162
13.2.3 布隆过滤器理论	164
13.2.4 布隆过滤器版	167
13.3 订单超时检测	169
13.4 实时对帐	171
13.5 实时热门商品	173
13.5.1 实现思路	173
13.5.2 底层 API 实现方式	175
13.5.3 Flink SQL 实现方式	180
A 逻辑时钟 - 如何刻画分布式中的事件顺序	183
A.1 如何确定分布式系统中事件的发生顺序?	183
A.2 全局物理时钟	186
A.3 全序和偏序	186
A.4 事件先后的相对性	187
A.5 逻辑时钟 (Lamport's Timestamp)	190
A.6 向量时钟 (Vector Clocks)	192
A.7 向量时钟看前面的问题	194
A.8 向量时钟的不足	195
A.9 总结与感想	195

第一章 有状态的流式处理

Apache Flink 是一个分布式流处理器，具有直观和富有表现力的 API，可实现有状态的流处理应用程序。它以容错的方式有效地大规模运行这些应用程序。Flink 于 2014 年 4 月加入 Apache 软件基金会作为孵化项目，并于 2015 年 1 月成为顶级项目。从一开始，Flink 就拥有一个非常活跃且不断增长的用户和贡献者社区。到目前为止，已有超过五百人为 Flink 做出贡献，并且它已经发展成为最复杂的开源流处理引擎之一，并得到了广泛采用的证明。Flink 为不同行业和全球的许多公司和企业提供大规模的商业关键应用。

流处理技术在大大小小的公司中越来越受欢迎，因为它为许多已建立的用例（如数据分析，ETL 和事务应用程序）提供了卓越的解决方案，同时也促进了新颖的应用程序，软件架构和商机。接下来我们将讨论，为什么有状态流处理变得如此受欢迎并评估其潜力。我们首先回顾传统的数据应用程序架构并指出它们的局限性。接下来，我们介绍基于状态流处理的应用程序设计与传统方法相比，它具有许多有趣的特征最后，我们简要讨论开源流处理器的发展，并在本地 Flink 实例上运行流应用程序。

1.1 传统数据处理架构

数十年来，数据和数据处理在企业中无处不在。多年来，数据的收集和使用一直在增长，公司已经设计并构建了基础架构来管理数据。大多数企业实施的传统架构区分了两种类型的数据处理：事务处理（OLTP）和分析处理（OLAP）。

1.1.1 事务处理

公司将各种应用程序用于日常业务活动，例如企业资源规划（ERP）系统，客户关系管理（CRM）软件和基于 Web 的应用程序。这些系统通常设计有单独的层，用于数据处理（应用程序本身）和数据存储（事务数据库系统），如图所示。

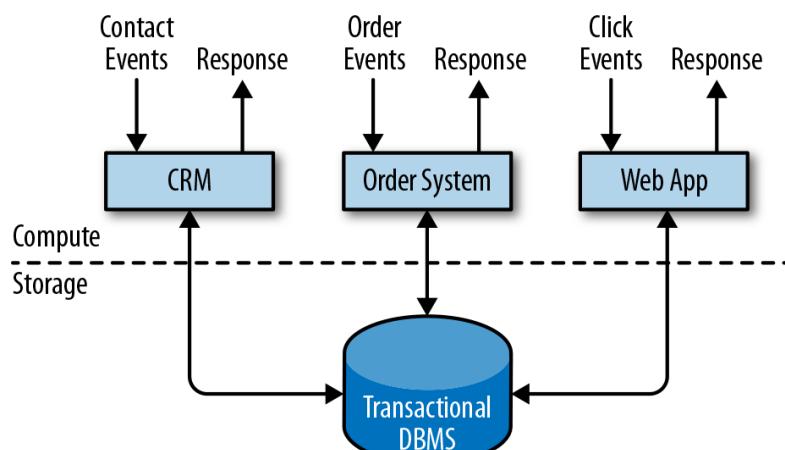


图 1.1: 事务处理

应用程序通常连接到外部服务或直接面向用户，并持续处理传入的事件，如网站上

的订单，电子邮件或点击。处理事件时，应用程序将会读取远程数据库的状态，或者通过运行事务来更新它。通常，一个数据库系统可以服务于多个应用程序，它们有时会访问相同的数据库或表。

当应用程序需要扩展时，这样的设计可能会导致问题。由于多个应用程序可能会同时用到相同的数据表示，或者共享相同的基础设施，因此想要更改表的结构或扩展数据库，就需要仔细的规划和大量的工作。克服紧耦合应用程序的最新方法是微服务设计模式。微服务被设计为小型、完备且独立的应用程序。他们遵循 UNIX 的理念，即“只做一件事并且把它做好”。通过将几个微服务相互连接来构建更复杂的应用程序，这些微服务仅通过标准化接口（例如 RESTful HTTP 连接）进行通信。由于微服务严格地彼此分离并且仅通过明确定义的接口进行通信，因此每个微服务都可以用不同技术栈来实现，包括编程语言、类库和数据存储。微服务和所有必需的软件和服务通常捆绑在一起并部署在独立的容器中。下图描绘了一种微服务架构。

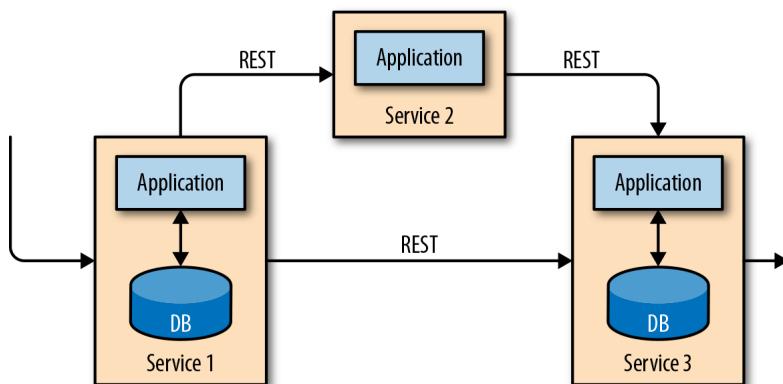


图 1.2: 事务处理

1.1.2 分析处理

大量数据存储在公司的各种事务数据库系统中，它们可以为公司业务运营提供宝贵的参考意见。例如，分析订单处理系统的数据，可以获得销量随时间的增长曲线；可以识别延迟发货的原因；还可以预测未来的销量以便提前调整库存。但是，事务数据通常分布在多个数据库中，它们往往汇总起来联合分析时更有价值。而且，数据通常需要转换为通用格式。

所以我们一般不会直接在事务数据库上运行分析查询，而是复制数据到数据仓库。数据仓库是对工作负载进行分析和查询的专用数据存储。为了填充数据仓库，需要将事务数据库系统管理的数据复制过来。将数据复制到数据仓库的过程称为 extract-transform-load (ETL)。ETL 过程从事务数据库中提取数据，将其转换为某种通用的结构表示，可能包括验证，值的规范化，编码，重复数据删除（去重）和模式转换，最后将其加载到分析数据库中。ETL 过程可能非常复杂，并且通常需要技术复杂的解决方案来满足性能要求。ETL 过程需要定期运行以保持数据仓库中的数据同步。

将数据导入数据仓库后，可以查询和分析数据。通常，在数据仓库上执行两类查询。第一种类型是定期报告查询，用于计算与业务相关的统计信息，比如收入、用户增长或

者输出的产量。这些指标汇总到报告中，帮助管理层评估业务的整体健康状况。第二种类型是即席查询，旨在提供特定问题的答案并支持关键业务决策，例如收集统计在投放商业广告上的花费，和获取的相应收入，以评估营销活动的有效性。两种查询由批处理方式由数据仓库执行，如图所示。

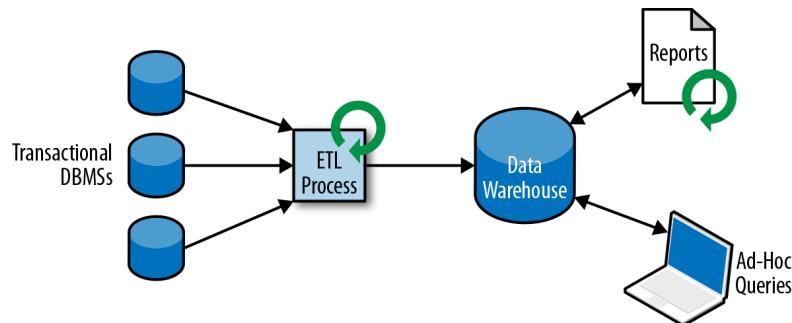


图 1.3: 事务处理

如今，Apache Hadoop 生态系统的组件，已经是许多企业 IT 基础架构中不可或缺的组成部分。现在的做法不是直接将所有数据都插入关系数据库系统，而是将大量数据（如日志文件，社交媒体或 Web 点击日志）写入 Hadoop 的分布式文件系统（HDFS）、S3 或其他批量数据存储库，如 Apache HBase，以较低的成本提供大容量存储容量。驻留在此类存储系统中的数据可以通过 SQL-on-Hadoop 引擎查询和处理，例如 Apache Hive，Apache Drill 或 Apache Impala。但是，基础结构与传统数据仓库架构基本相同。

1.2 有状态的流式处理

日常生活中，所有数据都是作为连续的事件流创建的。比如网站或者移动应用中的用户交互动作，订单的提交，服务器日志或传感器测量数据：所有这些都是事件流。实际上，很少有应用场景，能一次性地生成所需要的完整（有限）数据集。实际应用中更多的是无限事件流。有状态的流处理就是用于处理这种无限事件流的应用程序设计模式，在公司的IT基础设施中有广泛的应用场景。在我们讨论其用例之前，我们将简要介绍有状态流处理的工作原理。

如果我们想要无限处理事件流，并且不愿意繁琐地每收到一个事件就记录一次，那这样的应用程序就需要是有状态的，也就是说能够存储和访问中间数据。当应用程序收到一个新事件时，它可以从状态中读取数据，或者向该状态写入数据，总之可以执行任何计算。原则上讲，我们可以在各种不同的地方存储和访问状态，包括程序变量（内存）、本地文件，还有嵌入式或外部数据库。

Apache Flink 将应用程序状态，存储在内存或者嵌入式数据库中。由于 Flink 是一个分布式系统，因此需要保护本地状态以防止在应用程序或计算机故障时数据丢失。Flink 通过定期将应用程序状态的一致性检查点（check point）写入远程且持久的存储，来保证这一点。状态、状态一致性和 Flink 的检查点将在后面的章节中更详细地讨论，但是，现在，下图显示了有状态的流式 Flink 应用程序。

有状态的流处理应用程序，通常从事件日志中提取输入事件。事件日志就用来存储

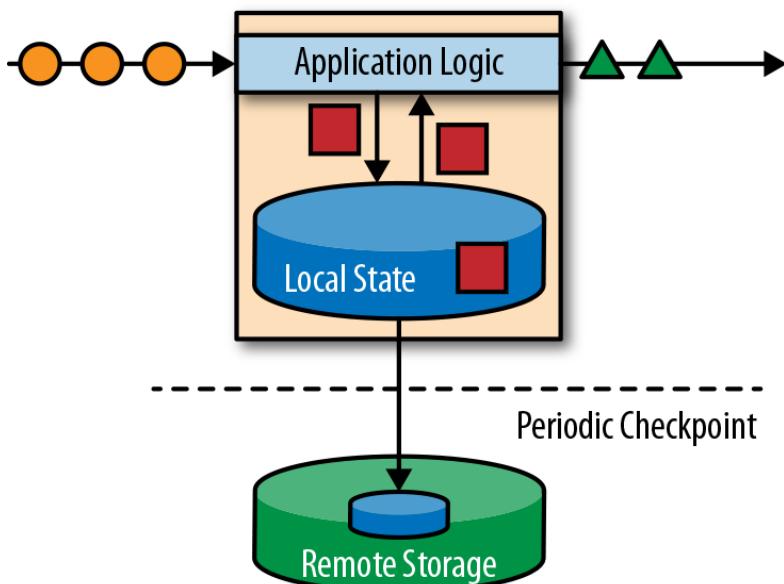


图 1.4: 有状态的流式处理

和分发事件流。事件被写入持久的仅添加（append-only）日志，这意味着无法更改写入事件的顺序。写入事件日志的流，可以被相同或不同的消费者多次读取。由于日志的仅附加（append-only）属性，事件始终以完全相同的顺序发布给所有消费者。现在已有几种事件日志系统，其中 Apache Kafka 是最受欢迎的，可以作为开源软件使用，或者是云计算提供商提供的集成服务。

在 Flink 上运行的有状态的流处理应用程序，是很有意思的一件事。在这个架构中，事件日志会按顺序保留输入事件，并且可以按确定的顺序重播它们。如果发生故障，Flink 将从先前的检查点（check point）恢复其状态，并重置事件日志上的读取位置，这样就可以恢复整个应用。应用程序将重放（并快进）事件日志中的输入事件，直到它到达流的尾部。此技术一般用于从故障中恢复，但也可用于更新应用程序、修复 bug 或者修复以前发出的结果，另外还可以用于将应用程序迁移到其他群集，或使用不同的应用程序版本执行 A / B 测试。

如前所述，有状态的流处理是一种通用且灵活的设计架构，可用于许多不同的场景。在下文中，我们提出了三类通常使用有状态流处理实现的应用程序：(1) 事件驱动应用程序，(2) 数据管道应用程序，以及 (3) 数据分析应用程序。

我们将应用程序分类描述，是为了强调有状态流处理适用于多种业务场景；而实际的应用中，往往具有以上多种情况的特征。

1.2.1 事件驱动应用程序

事件驱动的应用程序是有状态的流应用程序，它们使用特定的业务逻辑来提取事件流并处理事件。根据业务逻辑，事件驱动的应用程序可以触发诸如发送警报、或电子邮件之类的操作，或者将事件写入向外发送的事件流以供另一个应用程序使用。

事件驱动应用程序的典型场景包括：

- 实时推荐（例如，在客户浏览零售商网站时推荐产品）

- 行为模式检测或复杂事件处理（例如，用于信用卡交易中的欺诈检测）
- 异常检测（例如，检测侵入计算机网络的尝试）

事件驱动应用程序是微服务的演变。它们通过事件日志而不是 REST 调用进行通信，并将应用程序数据保存为本地状态，而不是将其写入外部数据存储区（例如关系数据库或键值数据库）。下图显示了由事件驱动的流应用程序组成的服务架构。

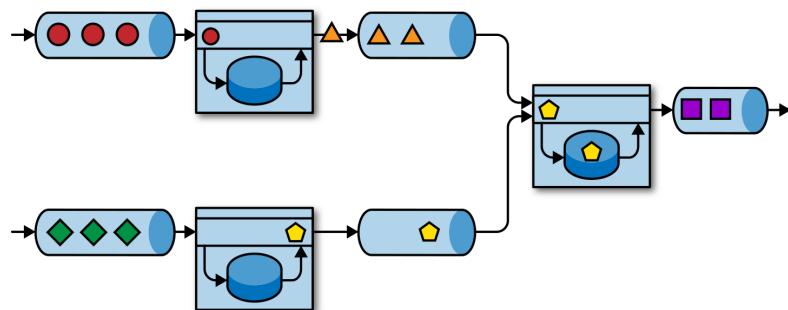


图 1.5: 事件驱动

上图中的应用程序通过事件日志连接。一个应用程序将其输出发送到事件日志通道（kafka），另一个应用程序使用其他应用程序发出的事件。事件日志通道将发送者和接收者分离，并提供异步、非阻塞的事件传输。每个应用程序都可以是有状态的，并且可以本地管理自己的状态而无需访问外部数据存储。应用程序也可以单独处理和扩展。

与事务性应用程序或微服务相比，事件驱动的应用程序具有多种优势。与读写远程数据库相比，本地状态访问提供了非常好的性能。扩展性和容错性都由流处理器来保证，并且以事件日志作为输入源，应用程序的整个输入数据可以可靠地存储，并且可以确定性地重放。此外，Flink 可以将应用程序的状态重置为先前的保存点（save point），从而可以在不丢失状态的情况下更新或重新扩展应用程序。

事件驱动的应用程序对运行它们的流处理器有很高的要求，并不是所有流处理器都适合运行事件驱动的应用程序。API 的表现力，以及对状态处理和事件时间支持的程度，决定了可以实现和执行的业务逻辑。这方面取决于流处理器的 API，主要看它能提供什么样的状态类型，以及它对事件时间处理的支持程度。此外，精确一次（exactly-once）的状态一致性和扩展应用程序的能力是事件驱动应用程序的基本要求。Apache Flink 符合所有的这些要求，是运行此类应用程序的一个非常好的选择。

1.2.2 数据管道

当今的 IT 架构包括许多不同的数据存储，例如关系型数据库和专用数据库系统、事件日志、分布式文件系统，内存中的缓存和搜索索引。所有这些系统都以不同的格式和数据结构存储数据，为其特定的访问模式提供最佳性能。公司通常将相同的数据存储在多个不同的系统中，以提高数据访问的性能。例如，网上商店中提供的产品信息，可以存储在交易数据库中，同时也存储在缓存（如 redis）和搜索索引（如 ES）中。由于数据的这种复制，数据存储必须保持同步。

在不同存储系统中同步数据的传统方法是定期 ETL 作业。但是，它们不能满足当今

许多场景的延迟要求。另一种方法是使用事件日志 (event log) 来发布更新。更新将写入事件日志并由事件日志分发。日志的消费者获取到更新之后，将更新合并到受影响的数据存储中。根据使用情况，传输的数据可能需要标准化、使用外部数据进行扩展，或者在目标数据存储提取之前进行聚合。

以较低的延迟，来提取、转换和插入数据是有状态流处理应用程序的另一个常见应用场景。这种类型的应用程序称为数据管道 (data pipeline)。数据管道必须能够在短时间内处理大量数据。操作数据管道的流处理器还应具有许多源 (source) 和接收器 (sink) 的连接器，以便从各种存储系统读取数据并将数据写入各种存储系统。当然，同样地，Flink 完成了所有这些功能。

1.2.3 流分析

ETL 作业定期将数据导入数据存储区，数据的处理是由即席查询（用户自定义查询）或设定好的通常查询来做的。无论架构是基于数据仓库还是基于 Hadoop 生态系统的组件，这都是批处理。多年来最好的处理方式就是，定期将数据加载到数据分析系统中，但它给分析管道带了的延迟相当大，而且无法避免。

根据设定好的时间间隔，可能需要数小时或数天才能将数据点包含在报告中。我们前面已经提到，数据管道可以实现低延迟的 ETL，所以在某种程度上，可以通过使用数据管道将数据导入存储区来减少延迟。但是，即使持续不停地进行 ETL 操作，在用查询来处理事件之前总会有延迟。虽然这种延迟在过去可能是可以接受的，但是今天的应用程序，往往要求必须能够实时收集数据，并立即对其进行操作（例如，在手机游戏中去适应不断变化的条件，或者在电商网站中提供个性化的用户体验）。

流式分析应用程序不是等待定期触发，而是连续地提取事件流，并且通过纳入最新事件来更新其计算结果，这个过程是低延迟的。这有些类似于数据库中用于更新视图 (views) 的技术。通常，流应用程序将其结果存储在支持更新的外部数据存储中，例如数据库或键值 (key-value) 存储。流分析应用程序的实时更新结果可用于驱动监控仪表板 (dashboard) 应用程序，如下图所示。

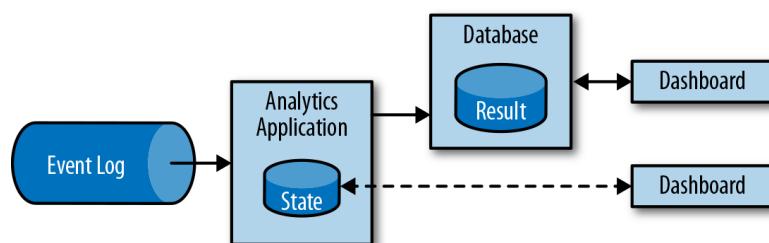


图 1.6：流分析

流分析应用程序最大的优势就是，将每个事件纳入到分析结果所需的时间短得多。除此之外，流分析应用程序还有另一个不太明显的优势。传统的分析管道由几个独立的组件组成，例如 ETL 过程、存储系统、对于基于 Hadoop 的环境，还包括用于触发任务 (jobs) 的数据处理和调度程序。相比之下，如果我们运行一个有状态流应用程序，那么流处理器就会负责所有这些处理步骤，包括事件提取、带有状态维护的连续计算以及更

新结果。此外，流处理器可以从故障中恢复，并且具有精确一次（exactly-once）的状态一致性保证，还可以调整应用程序的计算资源。像 Flink 这样的流处理器还支持事件时间（event-time）处理，这可以保证产生正确和确定的结果，并且能够在很短的时间内处理大量数据。

流分析应用程序通常用于：

- 监控手机网络的质量分析
- 移动应用中的用户行为
- 实时数据的即席分析

虽然我们不在此处介绍，但 Flink 还提供对流上的分析 SQL 查询的支持。

1.3 开源流处理的演进

数据流处理并不是一项新技术。一些最初的研究原型和商业产品可以追溯到 20 世纪 90 年代（1990s）。然而，在很大程度上，过去采用的流处理技术是由成熟的开源流处理器驱动的。如今，分布式开源流处理器在不同行业的许多企业中，处理着核心业务应用，比如电商、社交媒体、电信、游戏和银行等。开源软件是这一趋势的主要驱动力，主要原因有两个：

- 开源流处理软件是大家每一个人都可以评估和使用的产品。
- 由于许多开源社区的努力，可扩展流处理技术正在迅速成熟和发展。

仅仅一个 Apache 软件基金会就支持了十几个与流处理相关的项目。新的分布式流处理项目不断进入开源阶段，并不断增加新的特性和功能。开源社区不断改进其项目的功能，并正在推动流处理的技术边界。我们将简要介绍一下过去，看看开源流处理的起源和今天的状态。

第一代分布式开源流处理器（2011）专注于具有毫秒延迟的事件处理，并提供了在发生故障时防止事件丢失的保证。这些系统具有相当低级的 API，并且对于流应用程序的准确性和结果的一致性，不提供内置支持，因为结果会取决于到达事件的时间和顺序。另外，即使事件没有丢失，也可能不止一次地处理它们。与批处理器相比，第一代开源流处理器牺牲了结果准确性，用来获得更低的延迟。为了让当时的数据处理系统，可以同时提供快速和准确的结果，人们设计了所谓的 lambda 架构，如图所示。

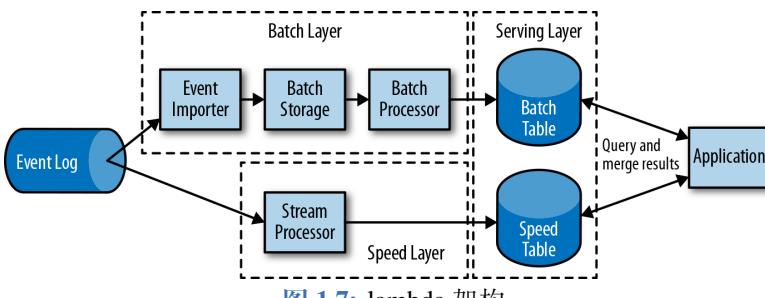


图 1.7：lambda 架构

lambda 架构增强了传统的批处理架构，其“快速层”（speed layer）由低延迟的流处理器来支持。数据到达之后由流处理器提取出来，并写入批处理存储。流处理器近乎实

时地计算近似结果并将它们写入“快速表”(speed table)。批处理器定期处理批量存储中的数据，将准确的结果写入批处理表，并从速度表中删除相应的不准确结果。应用程序会合并快速表中的近似结果和批处理表中的准确结果，然后消费最终的结果。

lambda 架构现在已经不再是最先进的，但仍在许多地方使用。该体系结构的最初目标是改善原始批处理分析体系结构的高延迟。但是，它有一些明显的缺点。首先，它需要对一个应用程序，做出两个语义上等效的逻辑实现，用于两个独立的、具有不同 API 的处理系统。其次，流处理器计算的结果只是近似的。第三，lambda 架构很难建立和维护。

通过在第一代基础上进行改进，下一代分布式开源流处理器（2013）提供了更好的故障保证，并确保在发生故障时，每个输入记录仅对结果产生一次影响（exactly-once）。此外，编程 API 从相当低级的操作符接口演变为高级 API。但是，一些改进（例如更高的吞吐量和更好的故障保证）是以将处理延迟从毫秒增加到几秒为代价的。此外，结果仍然取决于到达事件的时间和顺序。

第三代分布式开源流处理器（2015）解决了结果对到达事件的时间和顺序的依赖性。结合精确一次（exactly-once）的故障语义，这一代系统是第一个具有计算一致性和准确结果的开源流处理器。通过基于实际数据来计算结果（“重演”数据），这些系统还能够以与“实时”数据相同的方式处理历史数据。另一个改进是解决了延迟/吞吐量无法同时保证的问题。先前的流处理器仅能提供高吞吐量或者低延迟（其中之一），而第三代系统能够同时提供这两个特性。这一代的流处理器使得 lambda 架构过时了。当然，这一代流处理以 flink 为代表。

除了目前讨论的特性，例如容错、性能和结果准确性之外，流处理器还不断添加新的操作功能，例如高可用性设置，与资源管理器（如 YARN 或 Kubernetes）的紧密集成，以及能够动态扩展流应用程序。其他功能包括：支持升级应用程序代码，或将作业迁移到其他群集或新版本的流处理器，而不会丢失当前状态。

1.4 Flink 简介

Apache Flink 是第三代分布式流处理器，它拥有极富竞争力的功能。它提供准确的大规模流处理，具有高吞吐量和低延迟。特别的是，以下功能使 Flink 脱颖而出：

- 事件时间(event-time)和处理时间(processing-time)语义。即使对于无序事件流，事件时间(event-time)语义仍然能提供一致且准确的结果。而处理时间(processing-time)语义可用于具有极低延迟要求的应用程序。
- 精确一次(exactly-once)的状态一致性保证。
- 每秒处理数百万个事件，毫秒级延迟。Flink 应用程序可以扩展为在数千个核(cores)上运行。
- 分层 API，具有不同的权衡表现力和易用性。本书介绍了 DataStream API 和过程函数(process function)，为常见的流处理操作提供原语，如窗口和异步操作，以及精确控制状态和时间的接口。本书不讨论 Flink 的关系 API，SQL 和 LINQ 风格的 Table API。

- 连接到最常用的存储系统，如 Apache Kafka, Apache Cassandra, Elasticsearch, JDBC, Kinesis 和（分布式）文件系统，如 HDFS 和 S3。
- 由于其高可用的设置（无单点故障），以及与 Kubernetes, YARN 和 Apache Mesos 的紧密集成，再加上从故障中快速恢复和动态扩展任务的能力，Flink 能够以极少的停机时间 7*24 全天候运行流应用程序。
- 能够更新应用程序代码并将作业（jobs）迁移到不同的 Flink 集群，而不会丢失应用程序的状态。
- 详细且可自定义的系统和应用程序指标集合，以提前识别问题并对其做出反应。
- 最后但同样重要的是，Flink 也是一个成熟的批处理器。

除了这些功能之外，Flink 还是一个非常易于开发的框架，因为它易于使用的 API。嵌入式执行模式，可以在单个 JVM 进程中启动应用程序和整个 Flink 系统，这种模式一般用于在 IDE 中运行和调试 Flink 作业。在开发和测试 Flink 应用程序时，此功能非常有用。

1.5 编写第一个 Flink 程序

1.5.1 在 IDEA 中编写 Flink 程序

1. 使用 IntelliJ IDEA 创建一个 Maven 新项目
2. 勾选 Create from archetype，然后点击 Add Archetype 按钮
3. GroupId 中输入 org.apache.flink，ArtifactId 中输入 flink-quickstart-java，Version 中输入 Flink 的版本号 1.12.0，然后点击 OK
4. 点击向右箭头，出现下拉列表，选中 flink-quickstart-java: 版本号，点击 Next
5. Name 中输入 FlinkTutorial，GroupId 中输入 com.atguigu，ArtifactId 中输入 FlinkTutorial，点击 Next
6. 最好使用 IDEA 默认的 Maven 工具：Bundled（Maven 3），点击 Finish，等待一会儿，项目就创建好了

编写 WordCount.java 程序

```

1 public class WordCount {
2     public static void main(String[] args) throws Exception {
3         final StreamExecutionEnvironment env = StreamExecutionEnvironment.
4             getExecutionEnvironment();
5             env.setParallelism(1);
6
7             DataStream<String> stream = env.fromElements("Hello World", "Hello World");
8
9             stream
10            .flatMap(new Tokenizer())
11            .keyBy(r -> r.f0)
12            .sum(1)
13            .print();

```

```

13     env.execute("单词计数");
14 }
15
16
17 public static class Tokenizer implements FlatMapFunction<String, Tuple2<String,
18 Integer>> {
19     @Override
20     public void flatMap(String value, Collector<Tuple2<String, Integer>> out) throws
21     Exception {
22         String[] arr = value.split("\\s");
23         for (String s : arr) {
24             out.collect(Tuple2.of(s, 1));
25         }
26     }
27 }
```

1.5.2 下载 Flink 运行时环境，提交 Jar 包的运行方式

先下载压缩包，然后进行解压。然后进入文件夹。

```
$ ./bin/start-cluster.sh
```

可以打开 Flink WebUI 查看集群状态: <http://localhost:8081>

在 IDEA 中使用 maven package 打包。

提交打包好的 JAR 包:

```
$ ./bin/flink run \colorbox{yellow}{打包好的 JAR 包的绝对路径}
```

停止 Flink 集群

```
$ ./bin/stop-cluster.sh
```

查看标准输出日志的位置，在 log 文件夹中，后缀是.out。

```
$ cd log/
```

第二章 流处理基础

2.1 数据流编程

在我们深入研究流处理的基础知识之前，让我们来看看在数据流程编程的背景和使用的术语。

2.1.1 数据流图

顾名思义，数据流程序描述了数据如何在算子之间流动。数据流程序通常表示为有向图，其中节点称为算子，用来表示计算，边表示数据之间的依赖性。算子是数据流程序的基本功能单元。他们从输入消耗数据，对它们执行计算，并生成数据输出用于进一步处理。一个数据流图必须至少有一个数据源和一个数据接收器。

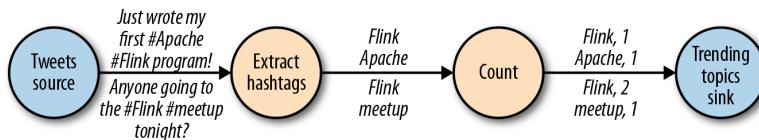


图 2.1: 逻辑流图

像上图中的数据流图被称为逻辑流图，因为它们表示了计算逻辑的高级视图。为了执行一个数据流程序，Flink 会将逻辑流图转换为物理数据流图，详细说明程序的执行方式。例如，如果我们使用分布式处理引擎，每个算子在不同的物理机器可能有几个并行的任务运行。图 2-2 显示了图 2-1 逻辑图的物理数据流图。而在逻辑数据流图中节点表示算子，在物理数据流图中，节点是任务。“Extract hashtags” 和 “Count” 算子有两个并行算子任务，每个算子任务对输入数据的子集执行计算。

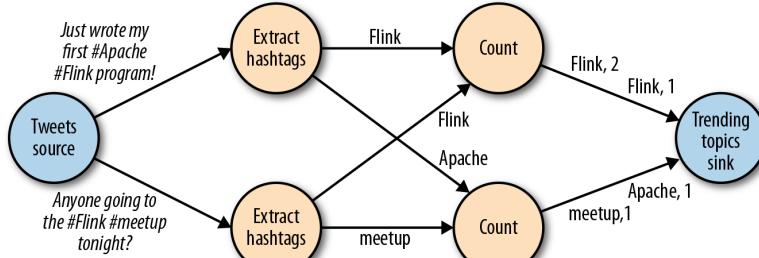


图 2.2: 物理数据流图

2.1.2 数据并行和任务并行

我们可以以不同方式利用数据流图中的并行性。第一，我们可以对输入数据进行分区，并在数据的子集上并行执行具有相同算子的任务并行。这种类型的并行性被称为数据并行性。数据并行是有用的，因为它允许处理大量数据，并将计算分散到不同的计算

节点上。第二，我们可以将不同的算子在相同或不同的数据上并行执行。这种并行性称为任务并行性。使用任务并行性，我们可以更好地利用计算资源。

2.1.3 数据交换策略

数据交换策略定义了在物理执行流图中如何将数据分配给任务。数据交换策略可以由执行引擎自动选择，具体取决于算子的语义或我们明确指定的语义。在这里，我们简要回顾一些常见的数据交换策略，如图 2-3 所示。

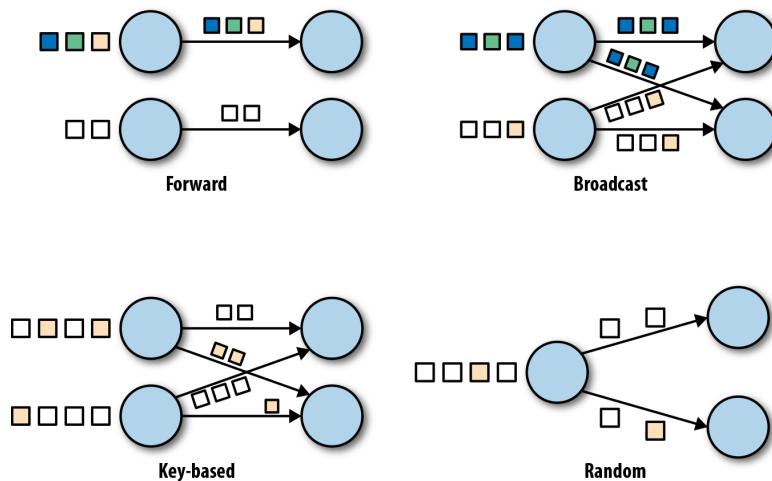


图 2.3: 数据交换策略

- 前向策略将数据从一个任务发送到接收任务。如果两个任务都位于同一台物理计算机上（这通常由任务调度器确保），这种交换策略可以避免网络通信。
- 广播策略将所有数据发送到算子的所有并行任务上面去。因为这种策略会复制数据和涉及网络通信，所以代价相当昂贵。
- 基于键控的策略通过 Key 值（键）对数据进行分区保证具有相同 Key 的数据将由同一任务处理。在图 2-2 中，输出“Extract hashtags”算子使用键来分区（hashtag），以便 count 算子的任务可以正确计算每个 # 标签的出现次数。
- 随机策略统一将数据分配到算子的任务中去，以便均匀地将负载分配到不同的计算任务。

2.2 并行处理流数据

既然我们熟悉了数据流编程的基础知识，现在是时候看看这些概念如何应用于并行的处理数据流了。但首先，让我们定义术语数据流：数据流是一个可能无限的事件序列。

数据流中的事件可以表示监控数据，传感器测量数据，信用卡交易数据，气象站观测数据，在线用户交互数据，网络搜索数据等。在本节中，我们将学习如何并行处理无限流，使用数据流编程范式。

2.2.1 延迟和吞吐量

流处理程序不同与批处理程序。在评估性能时，要求也有所不同。对于批处理程序，我们通常关心一个作业的总的执行时间，或我们的处理引擎读取输入所需的时间，执行计算，并回写结果。由于流处理程序是连续运行的，输入可能是无界的，所以数据流处理中没有总执行时间的概念。相反，流处理程序必须尽可能快的提供输入数据的计算结果。我们使用延迟和吞吐量来表征流处理的性能要求。

2.2.2 延迟

延迟表示处理事件所需的时间。它是接收事件和看到在输出中处理此事件的效果之间的时间间隔。要直观的理解延迟，考虑去咖啡店买咖啡。当你进入咖啡店时，可能还有其他顾客在里面。因此，你排队等候直到轮到你下订单。收银员收到你的付款并通知准备饮料的咖啡师。一旦你的咖啡准备好了，咖啡师会叫你的名字，你可以到柜台拿你的咖啡。服务延迟是从你进入咖啡店的那一刻起，直到你喝上第一口咖啡之间的时间间隔。

在数据流中，延迟是以时间为单位测量的，例如毫秒。根据应用程序，我们可能会关心平均延迟，最大延迟或百分位延迟。例如，平均延迟值为 10ms 意味着处理事件的平均时间在 10 毫秒内。或者，延迟值为 95%，10ms 表示 95% 的事件在 10ms 内处理完毕。平均值隐藏了处理延迟的真实分布，可能会让人难以发现问题。如果咖啡师在准备卡布奇诺之前用完了牛奶，你必须等到他们从供应室带来一些。虽然你可能会因为这么长时间的延迟而生气，但大多数其他客户仍然会感到高兴。

确保低延迟对于许多流应用程序来说至关重要，例如欺诈检测，系统警报，网络监控和提供具有严格服务水平协议的服务。低延迟是流处理的关键特性，它实现了我们所谓的实时应用程序。像 Apache Flink 这样的现代流处理器可以提供低至几毫秒的延迟。相比之下，传统批处理程序延迟通常从几分钟到几个小时不等。在批处理中，首先需要收集事件批次，然后才能处理它们。因此，延迟是受每个批次中最后一个事件的到达时间的限制。所以自然而然取决于批的大小。真正的流处理不会引入这样的人为延迟，因此可以实现真正的低延迟。真的流模型，事件一进入系统就可以得到处理。延迟更密切地反映了在每个事件上必须进行的实际工作。

2.2.3 吞吐量

吞吐量是衡量系统处理能力的指标，也就是处理速率。也就是说，吞吐量告诉我们每个时间单位系统可以处理多少事件。重温咖啡店的例子，如果商店营业时间为早上 7 点至晚上 7 点。当天为 600 个客户提供了服务，它的平均吞吐量将是每小时 50 个客户。虽然我们希望延迟尽可能低，但我们通常也需要吞吐量尽可能高。

吞吐量以每个时间单位系统所能处理的事件数量或操作数量来衡量。值得注意的是，事件处理速率取决于事件到达的速率，低吞吐量并不一定表示性能不佳。在流式系统中，我们通常希望确保我们的系统可以处理最大的预期事件到达的速率。也就是说，我们主

要的关注点在于确定的峰值吞吐量是多少，当系统处于最大负载时性能怎么样。为了更好地理解峰值吞吐量的概念，让我们考虑一个流处理程序没有收到任何输入的数据，因此没有消耗任何系统资源。当第一个事件进来时，它会尽可能以最小延迟立即处理。例如，如果你是第一个出现在咖啡店的顾客，在早上开门后，你将立即获得服务。理想情况下，您希望此延迟保持不变，并且独立于传入事件的速率。但是，一旦我们达到使系统资源被完全使用的事件传入速率，我们将不得不开始缓冲事件。在咖啡店里，午餐后会看到这种情况发生。许多人出现在同一时间，必须排队等候。在此刻，咖啡店系统已达到其峰值吞吐量，进一步增加事件传入的速率只会导致更糟糕的延迟。如果系统继续以可以处理的速率接收数据，缓冲区可能变为不可用，数据可能会丢失。这种情况是众所周知的作为背压，有不同的策略来处理它。

2.2.4 延迟与吞吐量的对比

此时，应该清楚延迟和吞吐量不是独立指标。如果事件需要在处理流水线中待上很长时间，我们不能轻易确保高吞吐量。同样，如果系统容量很小，事件将被缓冲，而且必须等待才能得到处理。

让我们重温一下咖啡店的例子来阐明一下延迟和吞吐量如何相互影响。首先，应该清楚存在没有负载时的最佳延迟。也就是说，如果你是咖啡店的唯一客户，会很快得到咖啡。然而，在繁忙时期，客户将不得不排队等待，并且会有延迟增加。另一个影响延迟和吞吐量的因素是处理事件所花费的时间或为每个客户提供服务所花费的时间。想象一下，期间圣诞节假期，咖啡师不得不为每杯咖啡画圣诞老人。这意味着准备一杯咖啡需要的时间会增加，导致每个人花费更多的时间在等待咖啡师画圣诞老人，从而降低整体吞吐量。

那么，你可以同时获得低延迟和高吞吐量吗？或者这是一个无望的努力？我们可以降低得到咖啡的延迟，方法是：聘请一位更熟练的咖啡师来准备咖啡。在高负载时，这种变化也会增加吞吐量，因为会在相同的时间内为更多的客户提供服务。实现相同结果的另一种方法是雇用第二个咖啡师来利用并行性。这里的主要想法是降低延迟来增加吞吐量。当然，如果系统可以更快的执行操作，它可以在相同的时间内执行更多操作。事实上，在流中利用并行性时也会发生这种情况。通过并行处理多个流，在同时处理更多事件的同时降低延迟。

2.3 数据流上的操作

流处理引擎通常提供一组内置操作：摄取(ingest)，转换(transform)和输出流(output)。这些操作可以结合到数据流图中来实现逻辑流处理程序。在本节中，我们描述最常见的流处理操作。

操作可以是无状态的或有状态的。无状态操作不保持任何内部状态。也就是说，事件的处理不依赖于过去看到的任何事件，也没有保留历史。无状态操作很容易并行化，因为事件可以彼此独立地处理，也独立于事件到达的顺序(和事件到达顺序没有关系)。而

且，在失败的情况下，无状态操作可以是简单的重新启动并从中断处继续处理。相反，有状态操作可能会维护之前收到的事件的信息。此状态可以通过传入事件更新，也可以用于未来事件的处理逻辑。有状态的流处理应用程序更难以并行化和以容错的方式来运行，因为状态需要有效的进行分区和在发生故障的情况下可靠地恢复。

2.3.1 数据摄入和数据吞吐量

数据摄取和数据出口操作允许流处理程序与外部系统通信。数据摄取是操作从外部源获取原始数据并将其转换为其他格式 (ETL)。实现数据提取逻辑的运算符被称为数据源。数据源可以从 TCP Socket，文件，Kafka Topic 或传感器数据接口中提取数据。数据出口是以适合消费的形式产出到外部系统。执行数据出口的运算符称为数据接收器，包括文件，数据库，消息队列和监控接口。

2.3.2 转换算子

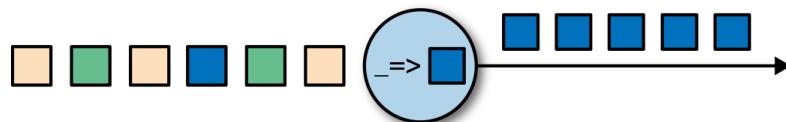


图 2.4: 转换算子

转换算子是单遍处理算子，碰到一个事件处理一个事件。这些操作在使用后会消费一个事件，然后对事件数据做一些转换，产生一个新的输出流。转换逻辑可以集成在操作符中或由 UDF 函数提供，如图所示图 2-4。程序员编写实现自定义计算逻辑。

操作符可以接受多个输入流并产生多个输出流。他们还可以通过修改数据流图的结构要么将流分成多个流，要么将流合并为一条流。

2.3.3 滚动聚合

滚动聚合是一种聚合，例如 sum，minimum 和 maximum，为每个输入事件不断更新。聚合操作是有状态的，并将当前状态与传入事件一起计算以产生更新的聚合值。请注意能够有效地将当前状态与事件相结合产生单个值，聚合函数必须是关联的和可交换的。否则，操作符必须存储完整的流数据历史。图 2-5 显示了最小滚动聚合。操作符保持当前的最小值和相应地为每个传入的事件来更新最小值。

2.3.4 窗口操作符

转换和滚动聚合一次处理一个事件产生输出事件并可能更新状态。但是，有些操作必须收集并缓冲数据以计算其结果。例如，考虑不同流之间的连接或整体聚合这样的操作，例如中值函数。为了在无界流上高效运行这些操作符，我们需要限制这些操作维护的数据量。在本节中，我们将讨论窗口操作，提供此服务。

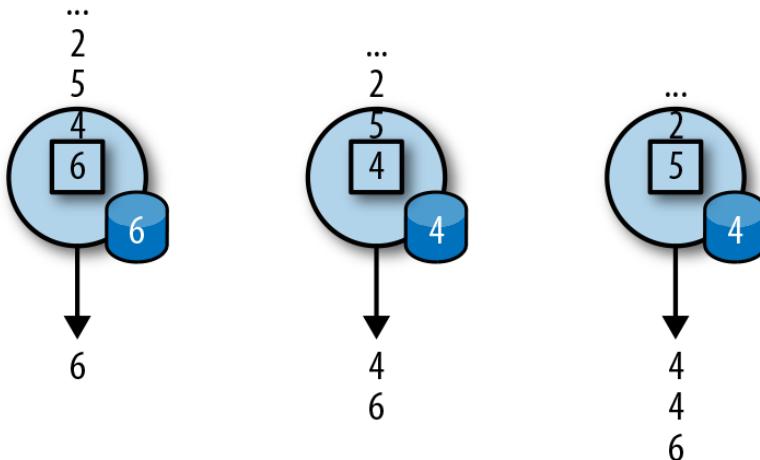


图 2.5: 滚动聚合

窗口还可以在语义上实现关于流的比较复杂的查询。我们已经看到了滚动聚合的方式，以聚合值编码整个流的历史数据来为每个事件提供低延迟的结果。但如果我们只对最近的数据感兴趣的话会怎样？考虑给司机提供实时交通信息的应用程序。这个程序可以使他们避免拥挤的路线。在这种场景下，你想知道某个位置在最近几分钟内是否有事故发生。另一方面，了解所有发生过的事故在这个应用场景下并没有什么卵用。更重要的是，通过将流历史缩减为单一聚合值，我们将丢失这段时间内数据的变化。例如，我们可能想知道每 5 分钟有多少车辆穿过某个路口。

窗口操作不断从无限事件流中创建有限的事件集，好让我们执行有限集的计算。通常会基于数据属性或基于时间的窗口来分配事件。要正确定义窗口运算符语义，我们需要确定如何给窗口分配事件以及对窗口中的元素进行求值的频率是什么样的。窗口的行为由一组策略定义。窗口策略决定何时创建新的窗口以及要分配的事件属于哪个窗口，以及何时对窗口中的元素进行求值。而窗口的求值基于触发条件。一旦触发条件得到满足，窗口的内容将会被发送到求值函数，求值函数会将计算逻辑应用于窗口中的元素。求值函数可以是 sum 或 minimal 或自定义的聚合函数。求值策略可以根据时间或者数据属性计算（例如，在过去五秒内收到的事件或者最近的一百个事件等等）。接下来，我们描述常见窗口类型的语义。

- 滚动窗口是将事件分配到固定大小的不重叠的窗口中。当通过窗口的结尾时，全部事件被发送到求值函数进行处理。基于计数的滚动窗口定义了在触发求值之前需要收集多少事件。图 2-6 显示了一个基于计数的翻滚窗口，每四个元素一个窗口。基于时间的滚动窗口定义一个时间间隔，包含在此时间间隔内的事件。图 2-7 显示了基于时间的滚动窗口，将事件收集到窗口中每 10 分钟触发一次计算。

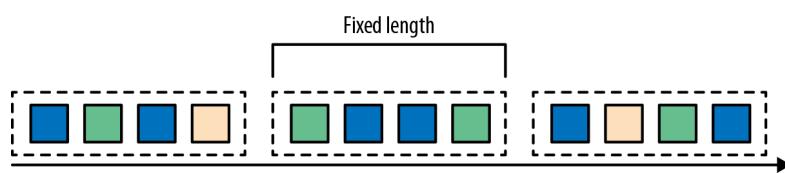


图 2.6: 计数滚动窗口

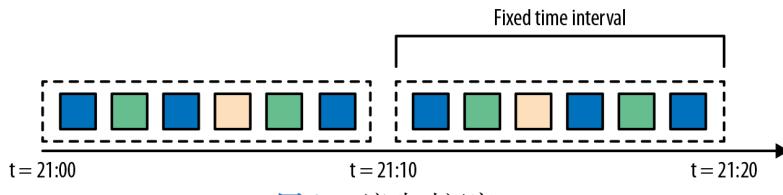


图 2.7: 滚动时间窗口

- 滑动窗口将事件分配到固定大小的重叠的窗口中去。因此，事件可能属于多个桶。我们通过提供窗口的长度和滑动距离来定义滑动窗口。滑动距离定义了创建新窗口的间隔。基于滑动计数的窗口，图 2-8 的长度为四个事件，三个为滑动距离。

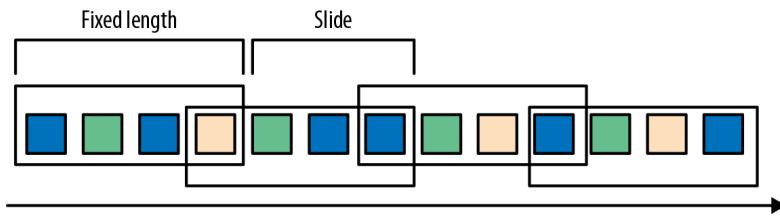


图 2.8: 滑动窗口

- 会话窗口在常见的真实场景中很有用，一些场景既不能使用滚动窗口也不能使用滑动窗口。考虑一个分析在线用户行为的应用程序。在应用程序里，我们想把源自同一时期的用户活动或会话事件分组在一起。会话由一系列相邻时间发生的事件组成，接下来有一段时间没有活动。例如，用户在 App 上浏览一系列的新闻，然后关掉 App，那么浏览新闻这段时间的浏览事件就是一个会话。会话窗口事先没有定义窗口的长度，而是取决于数据的实际情况，滚动窗口和滑动窗口无法应用于这个场景。相反，我们需要将同一会话中的事件分配到同一个窗口中去，而不同的会话可能窗口长度不一样。会话窗口会定义一个间隙值来区分不同的会话。间隙值的意思是：用户一段时间内不活动，就认为用户的会话结束了。图 2-9 显示了一个会话窗口。

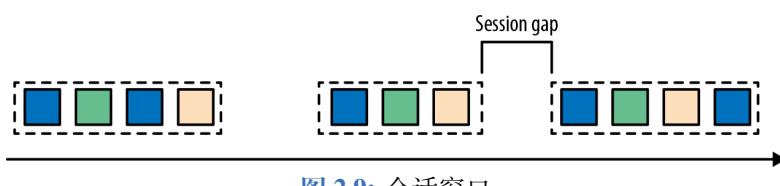


图 2.9: 会话窗口

到目前为止，所有窗口类型都是在整条流上去做窗口操作。但实际上你可能想要将一条流分流成多个逻辑流并定义并行窗口。例如，如果我们正在接收来自不同传感器的测量结果，那么可能想要在做窗口计算之前按传感器 ID 对流进行分流操作。在并行窗口中，每条流都独立于其他流，然后应用了窗口逻辑。图 2-10 显示了一个基于计数的长度为 2 的并行滚动窗口，根据事件颜色分流。

在流处理中，窗口操作与两个主要概念密切相关：时间语义和状态管理。时间也许是流处理最重要的方面。即使低延迟是流处理的一个有吸引力的特性，它的真正价值不仅仅是快速分析。
真实世界的系统，网络和通信渠道远非完美，流数据经常被推迟或无

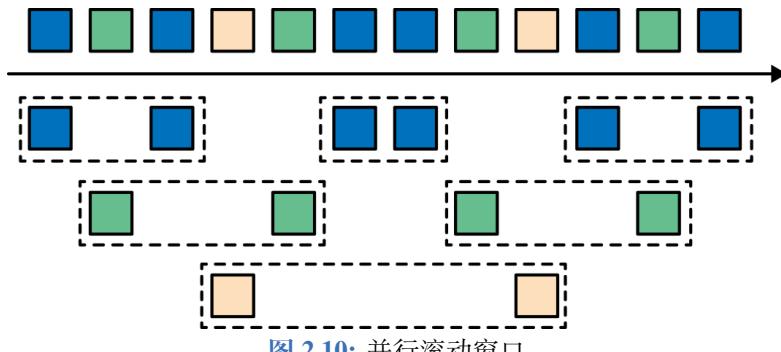


图 2.10: 并行滚动窗口

序(乱序)到达。理解如何在这种条件下提供准确和确定的结果是至关重要的。更重要的是，流处理程序可以按原样处理事件制作的也应该能够处理相同的历史事件方式，从而实现离线分析甚至时间旅行分析。当然，前提是我们的系统可以保存状态，因为可能有故障发生。到目前为止，我们看到的所有窗口类型在产生结果前都需要保存之前的数据。实际上，如果我们想计算任何指标，即使是简单的计数，我们也需要保存状态。考虑到流处理程序可能会运行几天，几个月甚至几年，我们需要确保状态可以在发生故障的情况下可靠地恢复。并且即使程序崩溃，我们的系统也能保证计算出准确的结果。本章，我们将深入探讨时间和状态的概念。

2.4 时间语义

在本节中，我们将介绍时间语义，并描述流中不同的时间概念。我们将讨论流处理器在乱序事件流的情况下如何提供准确的计算结果，以及我们如何处理历史事件流，如何在流中进行时间旅行。

2.4.1 在流处理中一分钟代表什么？

在处理可能是无限的事件流(包含了连续到达的事件)，时间成为流处理程序的核心方面。假设我们想要连续的计算结果，可能每分钟就要计算一次。在我们的流处理程序上下文中，一分钟的意思是什么？

考虑一个程序需要分析一款移动端的在线游戏的用户所产生的事件流。游戏中的用户分了组，而应用程序将收集每个小组的活动数据，基于小组中的成员多快达到了游戏设定的目标，然后在游戏中提供奖励。例如额外的生命和用户升级。例如，如果一个小组中的所有用户在一分钟之内都弹出了500个泡泡，他们将升一级。Alice是一个勤奋的玩家，她在每天早晨的通勤时间玩游戏。问题在于Alice住在柏林，并且乘地铁去上班。而柏林的地铁手机信号很差。我们设想一个这样的场景，Alice当她的手机连上网时，开始弹泡泡，然后游戏会将数据发送到我们编写的应用程序中，这时地铁突然进入了隧道，她的手机也断网了。Alice还在玩这个游戏，而产生的事件将会缓存在手机中。当地铁离开隧道，Alice的手机又在线了，而手机中缓存的游戏事件将发送到应用程序。我们的应用程序应该如何处理这些数据？在这个场景中一分钟的意思是什么？这个一分钟应该包含Alice离线的那段时间吗？下图展示了这个问题。

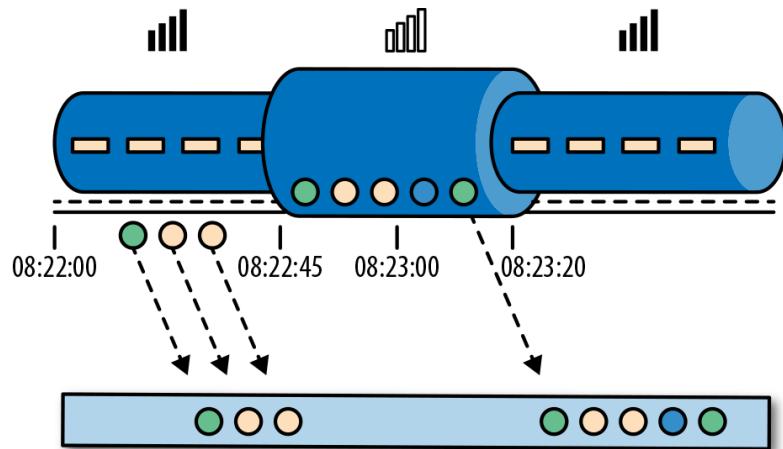


图 2.11: lambda 架构

在线手游是一个简单的场景，展示了应用程序的运算应该取决于事件实际发生的时间，而不是应用程序收到事件的时间。如果我们按照应用程序收到事件的时间来进行处理的话，最糟糕的后果就是，Alice 和她的朋友们再也不玩这个游戏了。但是还有很多时间语义非常关键的应用程序，我们需要保证时间语义的正确性。如果我们只考虑我们在一分钟之内收到了多少数据，我们的结果会变化，因为结果取决于网络连接的速度或处理的速度。相反，定义一分钟之内的事件数量，这个一分钟应该是数据本身的时间。

在 Alice 的这个例子中，流处理程序可能会碰到两个不同的时间概念：处理时间和事件时间。我们将在接下来的部分，讨论这两个概念。

2.4.2 处理时间

处理时间是处理流的应用程序的机器的本地时钟的时间（墙上时钟）。处理时间的窗口包含了一个时间段内来到机器的所有事件。这个时间段指的是机器的墙上时钟。如下图所示，在 Alice 的这个例子中，处理时间窗口在 Alice 的手机离线的情况下，时间将会继续行走。但这个处理时间窗口将不会收集 Alice 的手机离线时产生的事件。

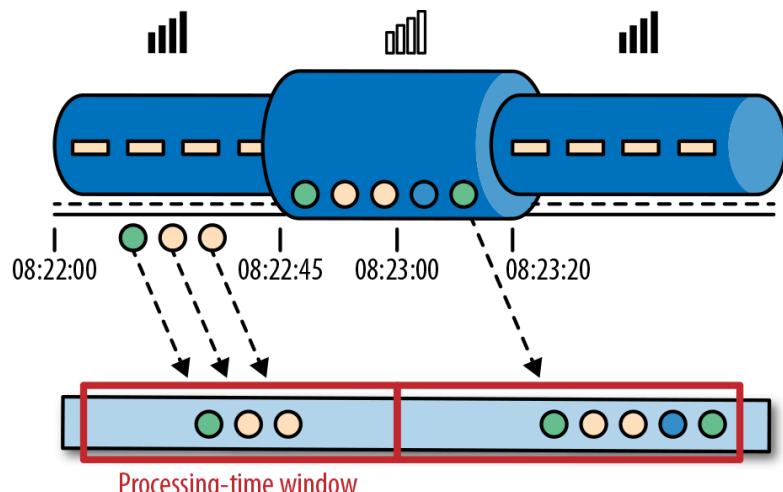


图 2.12: 处理时间窗口

2.4.3 事件时间

事件时间是流中的事件实际发生的时间。事件时间基于流中的事件所包含的时间戳。通常情况下，在事件进入流处理程序前，事件数据就已经包含了时间戳。下图展示了事件时间窗口将会正确的将事件分发到窗口中去。可以如实反应事情是怎么发生的。即使事件可能存在延迟。

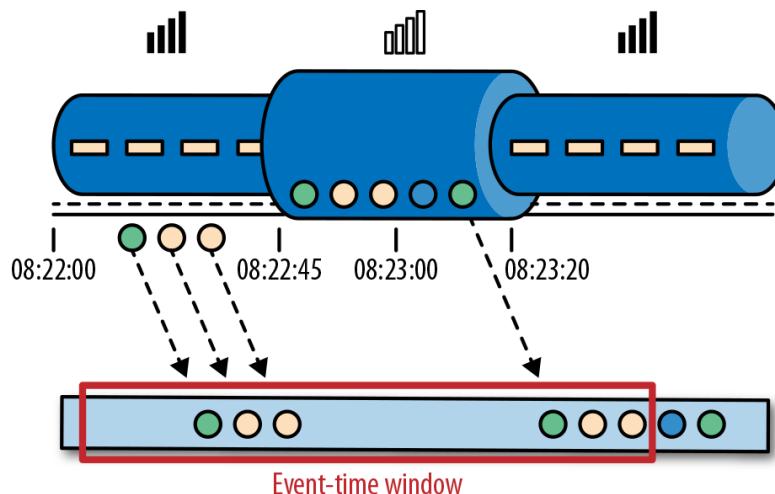


图 2.13: 事件时间窗口

事件时间使得计算结果的过程不需要依赖处理数据的速度。基于事件时间的操作是可以预测的，而计算结果也是确定的。无论流处理程序处理流数据的速度快或是慢，无论事件到达流处理程序的速度快或是慢，事件时间窗口的计算结果都是一样的。

可以处理迟到的事件只是我们使用事件时间所克服的一个挑战而已。普遍存在的事件乱序问题可以使用事件时间得到解决。考虑和 Alice 玩同样游戏的 Bob，他恰好和 Alice 在同一趟地铁上。Alice 和 Bob 虽然玩的游戏一样，但他们的手机信号是不同的运营商提供的。当 Alice 的手机没信号时，Bob 的手机依然有信号，游戏数据可以正常发送出去。

如果使用事件时间，即使碰到了事件乱序到达的情况，我们也可以保证结果的正确性。还有，当我们在处理可以重播的流数据时，由于时间戳的确定性，我们可以快进过去。也就是说，我们可以重播一条流，然后分析历史数据，就好像流中的事件是实时发生一样。另外，我们可以快进历史数据来使我们的应用程序追上现在的事件，然后应用程序仍然是一个实时处理程序，而且业务逻辑不需要改变。

2.4.4 水位线

在我们对事件时间窗口的讨论中，我们忽略了一个很重要的方面：我们应该怎样去决定何时触发事件时间窗口的计算？也就是说，在我们可以确定一个时间点之前的所有事件都已经到达之前，我们需要等待多久？我们如何知道事件是迟到的？在分布式系统无法准确预测行为的现实条件下，以及外部组件所引发的事件的延迟，以上问题并没有准确的答案。在本小节中，我们将会看到如何使用水位线来设置事件时间窗口的行为。

水位线是全局进度的度量标准。系统可以确信在一个时间点之后，不会有早于这个

时间点发生的事件到来了。本质上，水位线提供了一个逻辑时钟，这个逻辑时钟告诉系统当前的事件时间。当一个运算符接收到含有时间 T 的水位线时，这个运算符会认为早于时间 T 的发生的事件已经全部都到达了。**对于事件时间窗口和乱序事件的处理，水位线非常重要。运算符一旦接收到水位线，运算符会认为一段时间内发生的所有事件都已经观察到，可以触发针对这段时间内所有事件的计算了。**

水位线提供了一种结果可信度和延时之间的妥协。激进的水位线设置可以保证低延迟，但结果的准确性不够。在这种情况下，迟到的事件有可能晚于水位线到达，我们需要编写一些代码来处理迟到事件。另一方面，如果水位线设置的过于宽松，计算的结果准确性会很高，但可能会增加流处理程序不必要的延时。

在很多真实世界的场景里面，系统无法获得足够的知识来完美的确定水位线。在手游这个场景中，我们无法得知一个用户离线时间会有多长，他们可能正在穿越一条隧道，可能正在乘飞机，可能永远不会再玩儿了。水位线无论是用户自定义的或者是自动生成的，在一个分布式系统中追踪全局的时间进度都不是很容易。所以仅仅依靠水位线可能并不是一个很好的主意。流处理系统还需要提供一些机制来处理迟到的元素（在水位线之后到达的事件）。根据应用场景，我们可能需要把迟到事件丢弃掉，或者写到日志里，或者使用迟到事件来更新之前已经计算好的结果。

2.4.5 处理时间和事件时间

大家可能会有疑问，既然事件时间已经可以解决我们的所有问题，为什么我们还要对比这两个时间概念？真相是，处理时间在很多情况下依然很有用。处理时间窗口将会带来理论上最低的延迟。因为我们不需要考虑迟到事件以及乱序事件，所以一个窗口只需要简单的缓存窗口内的数据即可，一旦机器时间超过指定的处理时间窗口的结束时间，就会触发窗口的计算。所以对于一些处理速度比结果准确性更重要的流处理程序，**处理时间就派上用场了**。另一个应用场景是，当我们需要在真实的时间场景下，周期性的报告结果时，同时不考虑结果的准确性。一个例子就是一个实时监控的仪表盘，负责显示当事件到达时立即聚合的结果。最后，处理时间窗口可以提供流本身数据的忠实表达，对于一些案例可能是很必要的特性。例如我们可能对观察流和对每分钟事件的计数（检测可能存在的停电状况）很感兴趣。简单的说，**处理时间提供了低延迟，同时结果也取决于处理速度，并且也不能保证确定性**。另一方面，事件时间保证了结果的确定性，同时还可以使我们能够处理迟到的或者乱序的事件流。

2.5 状态和持久化模型

我们现在转向另一个对于流处理程序非常重要的话题：状态。在数据处理中，状态是普遍存在的。任何稍微复杂一点的计算，都涉及到状态。为了产生计算结果，一个函数在一段时间内的一定数量的事件上来累加状态（例如，聚合计数或者模式匹配）。有状态的运算符使用输入的事件以及内部保存的状态来计算得到输出。例如，一个滚动聚合运算符需要输出这个运算符所观察到的所有事件的累加和。这个运算符将会在内部保存当

前观察到的所有事件的累加和，同时每输入一个事件就更新一次累加和的计算结果。相似的，当一个运算符检测到一个“高温”事件紧接着十分钟以内检测到一个“烟雾”事件时，将会报警。直到运算符观察到一个“烟雾”事件或者十分钟的时间段已经过去，这个运算符需要在内部状态中一直保存着“高温”事件。

当我们考虑一下使用批处理系统来分析一个无界数据集时，会发现状态的重要性显而易见。在现代流处理器兴起之前，处理无界数据集的一个通常做法是将输入的事件攒成微批，然后交由批处理器来处理。当一个任务结束时，计算结果将被持久化，而所有的运算符状态就丢失了。一旦一个任务在计算下一个微批次的数据时，这个任务是无法访问上一个任务的状态的（都丢掉了）。这个问题通常使用将状态代理到外部系统（例如数据库）的方法来解决。相反，在一个连续不间断运行的流处理任务中，事件的状态是一直存在的，我们可以将状态暴露出来作为编程模型中的一等公民。当然，我们的确可以使用外部系统来管理流的状态，即使这个解决方案会带来额外的延迟。

由于流处理运算符默认处理的是无界数据流。所以我们必须要注意不要让内部状态无限的增长。为了限制状态的大小，运算符通常情况下会保存一些之前所观察到的事件流的总结或者概要。这个总结可能是一个计数值，一个累加和，或者事件流的采样，窗口的缓存操作，或者是一个自定义的数据结构，这个数据结构用来保存数据流中感兴趣的一些特性。

我们可以想象的到，支持有状态的运算符可能会碰到一些实现上的挑战：

- 状态管理：系统需要高效的管理状态，并保证针对状态的并发更新，不会产生竞争条件（race condition）。
- 状态分区：并行会带来复杂性。因为计算结果同时取决于已经保存的状态和输入的事件流。幸运的是，大多数情况下，我们可以使用 Key 来对状态进行分区，然后独立的管理每一个分区。例如，当我们处理一组传感器的测量事件流时，我们可以使用分区的运算符状态来针对不同的传感器独立的保存状态。
- 状态恢复：第三个挑战是有状态的运算符如何保证状态可以恢复，即使出现任务失败的情况，计算也是正确的。

下一节，我们将讨论任务失败和计算结果的保证。

2.5.1 任务失败

流任务中的运算符状态是很宝贵的，也需要抵御任务失败带来的问题。如果在任务失败的情况下，状态丢失的话，在任务恢复以后计算的结果将是不正确的。流任务会连续不断的运行很长时间，而状态可能已经收集了几天甚至几个月。在失败的情况下，重新处理所有的输入并重新生成一个丢失的状态，将会很浪费时间，开销也很大。

在本章开始时，我们看到如何将流的编程建模成数据流模型。在执行之前，流程序将会被翻译成物理层数据流图，物理层数据流图由连接的并行任务组成，而一个并行任务运行一些运算符逻辑，消费输入流数据，并为其他任务产生输出流数据。真实场景下，可能有数百个这样的任务并行运行在很多的物理机器上。在长时间的运行中，流任务中的任意一个任务在任意时间点都有可能失败。我们如何保证任务的失败能被正确的处理，

以使任务能继续的运行下去呢？事实上，我们可能希望我们的流处理器不仅能在任务失败的情况下继续处理数据，还能保证计算结果的正确性以及运算符状态的安全。我们在本小节来讨论这些问题。

什么是任务失败？

对于流中的每一个事件，一个处理任务分为以下步骤：(1) 接收事件，并将事件存储在本地的缓存中；(2) 可能会更新内部状态；(3) 产生输出记录。这些步骤都能失败，而系统必须对于在失败的场景下如何处理有清晰的定义。如果任务在第一步就失败了，事件会丢失吗？如果当更新内部状态的时候任务失败，那么内部状态会在任务恢复以后更新吗？在以上这些场景中，输出是确定性的吗？

在批处理场景下，所有的问题都不是问题。因为我们可以很方便的重新计算。所以不会有事件丢失，状态也可以得到完全恢复。在流的世界里，处理失败不是一个小问题。流系统在失败的情况下需要保证结果的准确性。接下来，我们需要看一下现代流处理系统所提供的一些保障，以及实现这些保障的机制。

结果的保证

当我们讨论保证计算的结果时，我们的意思是流处理器的内部状态需要保证一致性。也就是说我们关心的是应用程序的代码在故障恢复以后看到的状态值是什么。要注意保证应用程序状态的一致性并不是保证应用程序的输出结果的一致性。一旦输出结果被持久化，结果的准确性就很难保证了。除非持久化系统支持事务。

AT-MOST-ONCE

当任务故障时，最简单的做法是什么都不干，既不恢复丢失的状态，也不重播丢失的事件。At-most-once 语义的含义是最多处理一次事件。换句话说，事件可以被丢弃掉，也没有任何操作来保证结果的准确性。这种类型的保证也叫“没有保证”，因为一个丢弃掉所有事件的系统其实也提供了这样的保障。没有保障听起来是一个糟糕的主意，但如果我们将接受近似的结果，并且希望尽可能低的延迟，那么这样也挺好。

AT-LEAST-ONCE

在大多数的真实应用场景，我们希望不丢失事件。这种类型的保障成为 at-least-once，意思是所有的事件都得到了处理，而且一些事件还可能被处理多次。如果结果的正确性仅仅依赖于数据的完整性，那么重复处理是可以接受的。例如，判断一个事件是否在流中出现过，at-least-once 这样的保证完全可以正确的实现。在最坏的情况下，我们多次遇到了这个事件。而如果我们要对一个特定的事件进行计数，计算结果就可能是错误的了。

为了保证在 at-least-once 语义的保证下，计算结果也能正确。我们还需要另一套系统来从数据源或者缓存中重新播放数据。持久化的事件日志系统将会把所有的事件写入到持久化存储中。所以如果任务发生故障，这些数据可以重新播放。还有一种方法可以获得同等的效果，就是使用结果承认机制。这种方法将会把每一条数据都保存在缓存中，直到数据的处理等到所有的任务的承认。一旦得到所有任务的承认，数据将被丢弃。

EXACTLY-ONCE

恰好处理一次是最严格的保证，也是最难实现的。恰好处理一次语义不仅仅意味着没有事件丢失，还意味着针对每一个数据，内部状态仅仅更新一次。本质上，恰好处理

一次语义意味着我们的应用程序可以提供准确的结果，就好像从未发生过故障。

提供恰好处理一次语义的保证必须有至少处理一次语义的保证才行，同时还需要数据重放机制。另外，流处理器还需要保证内部状态的一致性。也就是说，在故障恢复以后，流处理器应该知道一个事件有没有在状态中更新。事务更新是达到这个目标的一种方法，但可能引入很大的性能问题。`Flink` 使用了一种轻量级快照机制来保证恰好处理一次语义。

端到端恰好处理一次

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。而在真实世界中，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。而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。要注意的是，我们可以通过弱一致性来实现更强的一致性语义。例如，当任务的操作具有幂等性时，比如流的最大值或者最小值的计算。在这种场景下，我们可以通过最少处理一次这样的一致性来实现恰好处理一次这样的最高级别的一致性。

第三章 Flink 运行架构

3.1 系统架构

Flink 是一个用于有状态的并行数据流处理的分布式系统。它由多个进程构成，这些进程一般会分布运行在不同的机器上。对于分布式系统来说，面对的常见问题有：集群中资源的分配和管理、进程协调调度、持久化和高可用的数据存储，以及故障恢复。

对于这些分布式系统的经典问题，业内已有比较成熟的解决方案和服务。所以 Flink 并不会自己去处理所有的问题，而是利用了现有的集群架构和服务，这样它就可以把精力集中在核心工作——分布式数据流处理上了。Flink 与一些集群资源管理工具有很好的集成，比如 Apache Mesos、YARN 和 Kubernetes；同时，也可以配置为独立（stand-alone）集群运行。Flink 自己并不提供持久化的分布式存储，而是直接利用了已有的分布式文件系统（比如 HDFS）或者对象存储（比如 S3）。对于高可用的配置，Flink 需要依靠 Apache ZooKeeper 来完成。

在本节中，我们将介绍 Flink 的不同组件，以及在运行程序时它们如何相互作用。我们会讨论部署 Flink 应用程序的两种模式，并且了解每种模式下分发和执行任务的方式。最后，我们还会解释一下 Flink 的高可用性模式是如何工作的。

3.1.1 Flink 运行时组件

Flink 运行时架构主要包括四个不同的组件，它们会在运行流处理应用程序时协同工作：作业管理器（JobManager）、资源管理器（ResourceManager）、任务管理器（TaskManager），以及分发器（Dispatcher）。因为 Flink 是用 Java 和 Scala 实现的，所以所有组件都会运行在 Java 虚拟机（JVMs）上。每个组件的职责如下：

- 作业管理器（JobManager）是控制一个应用程序执行的主进程，也就是说，每个应用程序都会被一个不同的作业管理器所控制执行。作业管理器会先接收到要执行的应用程序。这个应用程序会包括：作业图（JobGraph）、逻辑数据流图（logical dataflow graph）和打包了所有的类、库和其它资源的 JAR 包。作业管理器会把 JobGraph 转换成一个物理层面的数据流图，这个图被叫做“执行图”（ExecutionGraph），包含了所有可以并发执行的任务。作业管理器会向资源管理器（ResourceManager）请求执行任务必要的资源，也就是任务管理器（TaskManager）上的插槽（slot）。一旦它获取到了足够的资源，就会将执行图分发到真正运行它们的 TaskManager 上。而在运行过程中，作业管理器会负责所有需要中央协调的操作，比如说检查点（checkpoints）的协调。
- ResourceManager 主要负责管理任务管理器（TaskManager）的插槽（slot），TaskManager 插槽是 Flink 中定义的处理资源单元。Flink 为不同的环境和资源管理工具提供了不同资源管理器（ResourceManager），比如 YARN、Mesos、K8s，以及 standalone 部署。当作业管理器申请插槽资源时，ResourceManager 会将有空闲插槽的 TaskManager

分配给作业管理器。如果 ResourceManager 没有足够的插槽来满足作业管理器的请求，它还可以向资源提供平台发起会话，以提供启动 TaskManager 进程的容器。另外，ResourceManager 还负责终止空闲的 TaskManager，释放计算资源。

- 任务管理器(TaskManager)是 Flink 中的工作进程。通常在 Flink 中会有多个 TaskManager 运行，每一个 TaskManager 都包含了一定数量的插槽 (slots)。插槽的数量限制了 TaskManager 能够执行的任务数量。启动之后，TaskManager 会向资源管理器注册它的插槽；收到资源管理器的指令后，TaskManager 就会将一个或者多个插槽提供给作业管理器调用。作业管理器就可以向插槽分配任务 (tasks) 来执行了。在执行过程中，一个 TaskManager 可以跟其它运行同一应用程序的 TaskManager 交换数据。任务的执行和插槽的概念会在“任务执行”一节做具体讨论。
- 分发器 (Dispatcher) 可以跨作业运行，它为应用提交提供了 REST 接口。当一个应用被提交执行时，分发器就会启动并将应用移交给一个作业管理器。由于是 REST 接口，所以 Dispatcher 可以作为集群的一个 HTTP 接入点，这样就能够不受防火墙阻挡。Dispatcher 也会启动一个 Web UI，用来方便地展示和监控作业执行的信息。Dispatcher 在架构中可能并不是必需的，这取决于应用提交运行的方式。

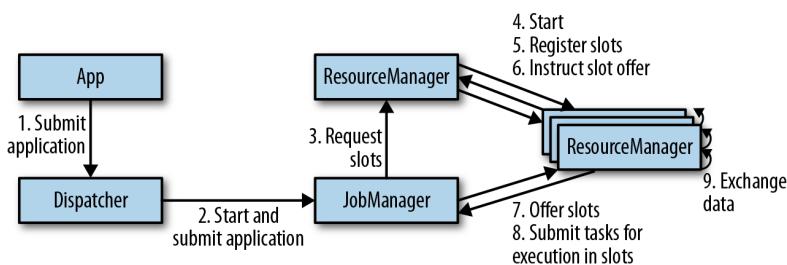


图 3.1: 作业提交流程



笔记 上图是从一个较为高级别的视角，来看应用中各组件的交互协作。如果部署的集群环境不同（例如 YARN，Mesos，Kubernetes，standalone 等），其中一些步骤可以被省略，或是有些组件会运行在同一个 JVM 进程中。

3.1.2 应用部署

Flink 应用程序可以用以下两种不同的方式部署：

框架 (Framework) 方式

在这个模式下，Flink 应用被打包成一个 Jar 文件，并由客户端提交给一个运行服务 (running service)。这个服务可以是一个 Flink 的 Dispatcher，也可以是一个 Flink 的作业管理器，或是 Yarn 的 ResourceManager。如果 application 被提交给一个作业管理器，则它会立即开始执行这个 application。如果 application 被提交给了一个 Dispatcher，或是 Yarn ResourceManager，则它会启动一个作业管理器，然后将 application 交给它，再由作业管理器开始执行此应用。

库 (Library) 方式

在这个模式下，Flink Application 会被打包在一个容器 (container) 镜像里，例如一个 Docker 镜像。此镜像包含了运行作业管理器和 ResourceManager 的代码。当一个容器

从镜像启动后，它会自动启动 ResourceManager 和作业管理器，并提交打包好的应用。另一种方法是：将应用打包到镜像后，只用于部署 TaskManager 容器。从镜像启动的容器会自动启动一个 TaskManager，然后连接 ResourceManager 并注册它的 slots。这些镜像的启动以及失败重启，通常都会由一个外部的资源管理器管理（比如 Kubernetes）。

框架模式遵循了传统的任务提交方式，从客户端提交到 Flink 运行服务。而在库模式下，没有运行的 Flink 服务。它是将 Flink 作为一个库，与应用程序一同打包到了一个容器镜像。这种部署方式在微服务架构中较为常见。我们会在“运行管理流式应用程序”一节对这个话题做详细讨论。

3.1.3 任务执行

一个 TaskManager 可以同时执行多个任务 (tasks)。这些任务可以是同一个算子 (operator) 的子任务 (数据并行)，也可以是来自不同算子的 (任务并行)，甚至可以是另一个不同应用程序的 (作业并行)。TaskManager 提供了一定数量的处理插槽 (processing slots)，用于控制可以并行执行的任务数。一个 slot 可以执行应用的一个分片，也就是应用中每一个算子的一个并行任务。图 3-2 展示了 TaskManagers, slots, tasks 以及 operators 之间的关系：

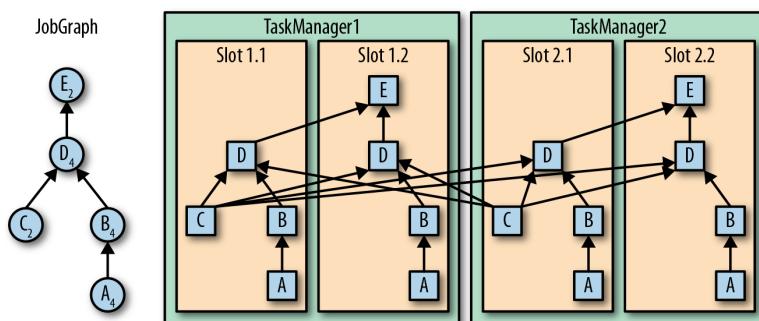


图 3.2: 并行执行任务

最左边是一个“作业图” (JobGraph)，包含了 5 个算子——它是应用程序的非并行表示。其中算子 A 和 C 是数据源 (source)，E 是输出端 (sink)。C 和 E 并行度为 2，而其他的算子并行度为 4。因为最高的并行度是 4，所以应用需要至少四个 slot 来执行任务。现在有两个 TaskManager，每个又各有两个 slot，所以我们的需求是满足的。作业管理器将 JobGraph 转化为“执行图” (ExecutionGraph)，并将任务分配到四个可用的 slot 上。对于有 4 个并行任务的算子，它的 task 会分配到每个 slot 上。而对于并行度为 2 的 operator C 和 E，它们的任务被分配到 slot 1.1、2.1 以及 slot 1.2、2.2。将 tasks 调度到 slots 上，可以让多个 tasks 跑在同一个 TaskManager 内，也就可以是的 tasks 之间的数据交换更高效。然而将太多任务调度到同一个 TaskManager 上会导致 TaskManager 过载，继而影响效率。之后我们会在“控制任务调度”一节继续讨论如何控制任务的调度。

TaskManager 在同一个 JVM 中以多线程的方式执行任务。线程较进程会更轻量级，但是线程之间并没有对任务进行严格隔离。所以，单个任务的异常行为有可能会导致整个 TaskManager 进程挂掉，当然也同时包括运行在此进程上的所有任务。通过为每个

TaskManager 配置单独的 slot，就可以将应用在 TaskManager 上相互隔离开来。TaskManager 内部有多线程并行的机制，而且在一台主机上可以部署多个 TaskManager，所以 Flink 在资源配置上非常灵活，在部署应用时可以充分权衡性能和资源的隔离。我们将会在第九章对 Flink 集群的配置和搭建继续做详细讨论。

3.1.4 高可用配置

流式应用程序一般被设计为 7×24 小时运行。所以很重要的一点是：即使出现了进程挂掉的情况，应用仍需要继续保持运行。为了从故障恢复，系统首先需要重启进程、然后重启应用并恢复它的状态。接下来，我们就来了解 Flink 如何重启失败的进程。

TaskManager 故障

如前所述，Flink 需要足够数目的 slot，来执行一个应用的所有任务。假设一个 Flink 环境有 4 个 TaskManager，每个提供 2 个插槽，那么流应用程序执行的最高并行度为 8。如果其中一个 TaskManager 挂掉了，那么可用的 slots 会降到 6。在这种情况下，作业管理器会请求 ResourceManager 提供更多的 slots。如果此请求无法满足——例如应用跑在一个独立集群——那么作业管理器在有足够的 slots 之前，无法重启应用。应用的重启策略决定了作业管理器的重启频率，以及两次重启尝试之间的时间间隔。

作业管理器故障

比 TaskManager 故障更严重的问题是作业管理器故障。作业管理器控制整个流应用程序的执行，并维护执行中的元数据——例如指向已完成检查点的指针。若是对应的作业管理器挂掉，则流程序无法继续运行。所以这就导致在 Flink 应用中，作业管理器是单点故障。为了解决这个问题，Flink 提供了高可用模式。在原先的作业管理器挂掉后，可以将一个作业的状态和元数据迁移到另一个作业管理器，并继续执行。

Flink 的高可用模式基于 Apache ZooKeeper，我们知道，ZooKeeper 是用来管理需要协调和共识的分布式服务的系统。Flink 主要利用 ZooKeeper 来进行领导者 (leader) 的选举，并把它作为一个高可用和持久化的数据存储。当在高可用模式下运行时，作业管理器会将 JobGraph 以及所有需要的元数据（例如应用程序的 jar 文件），写入到一个远程的持久化存储系统中。而且，作业管理器会将指向存储位置的指针，写入到 ZooKeeper 的数据存储中。在执行一个应用的过程中，作业管理器会接收每个独立任务检查点的状态句柄（也就是存储位置）。当一个检查点完成时（所有任务已经成功地将它们的状态写入到远程存储），作业管理器把状态句柄写入远程存储，并将指向这个远程存储的指针写入 ZooKeeper。这样，一个作业管理器挂掉之后再恢复，所需要的所有数据信息已经都保存在了远程存储，而 ZooKeeper 里存有指向此存储位置的指针。图 3-3 描述了这个设计：

当一个作业管理器失败，所有属于这个应用的任务都会自动取消。一个新的作业管理器接管工作，会执行以下操作：

- 从 ZooKeeper 请求存储位置 (storage location)，从远端存储获取 JobGraph，Jar 文件，以及应用最近一次检查点 (checkpoint) 的状态句柄 (state handles)
- 从 ResourceManager 请求 slots，用来继续运行应用
- 重启应用，并将所有任务的状态，重设为最近一次已完成的检查点

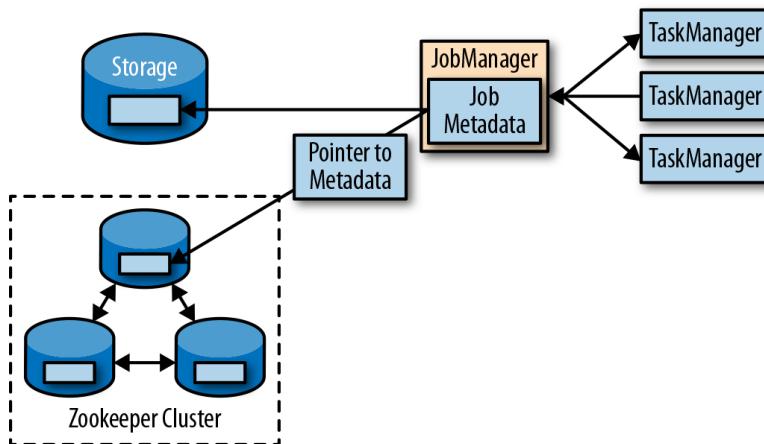


图 3.3: 高可用架构

如果我们是在容器环境里运行应用（如 Kubernetes），故障的作业管理器或 TaskManager 容器通常会由容器服务自动重启。当运行在 YARN 或 Mesos 之上时，作业管理器或 TaskManager 进程会由 Flink 的保留进程自动触发重启。而在 standalone 模式下，Flink 并未提供重启故障进程的工具。所以，此模式下我们可以增加备用（standby）的作业管理器和 TaskManager，用于接管故障的进程。我们将会在“高可用配置”一节中做进一步讨论。

3.2 Flink 中的数据传输

3.2.1 基于信任度的流控制

通过网络连接来发送每条数据的效率很低，会导致很大的开销。为了充分利用网络连接的带宽，就需要进行缓冲了。在流处理的上下文中，缓冲的一个缺点是会增加延迟，因为数据需要在缓冲区中进行收集，而不是立即发送。

Flink 实现了一个基于信任度的流量控制机制，其工作原理如下。接收任务授予发送任务一些“信任度”（credit），也就是为了接收其数据而保留的网络缓冲区数。当发送者收到一个信任度通知，它就会按照被授予的信任度，发送尽可能多的缓冲数据，并且同时发送目前积压数据的大小——也就是已填满并准备发送的网络缓冲的数量。接收者用保留的缓冲区处理发来的数据，并对发送者传来的积压量进行综合考量，为其所有连接的发送者确定下一个信用度授权的优先级。

基于信用度的流控制可以减少延迟，因为发送者可以在接收者有足够的资源接受数据时立即发送数据。此外，在数据倾斜的情况下，这样分配网络资源是一种很有效的机制，因为信用度是根据发送者积压数据量的规模授予的。因此，基于信用的流量控制是 Flink 实现高吞吐量和低延迟的重要组成部分。

3.2.2 任务链

Flink 采用了一种称为任务链的优化技术，可以在特定条件下减少本地通信的开销。为了满足任务链的要求，必须将两个或多个算子设为相同的并行度，并通过本地转发（local forward）的方式进行连接。图 3-5 所示的算子管道满足这些要求。它由三个算子组成，这些算子的任务并行度都被设为 2，并且通过本地转发方式相连接。

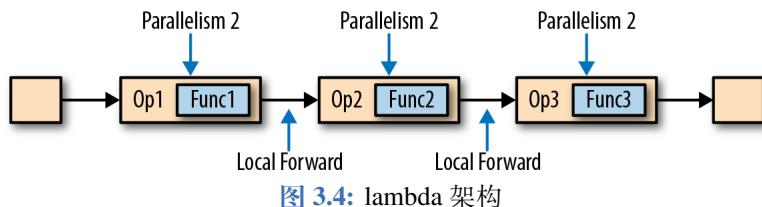


图 3.4: lambda 架构

图 3-6 展示了管道以任务链方式运行的过程。算子的函数被融合成了一个单一的任务，由一个线程执行。由函数生成的数据通过一个简单的方法调用移交给下一个函数；这样在函数之间直接传递数据，基本上没有序列化和通信成本。

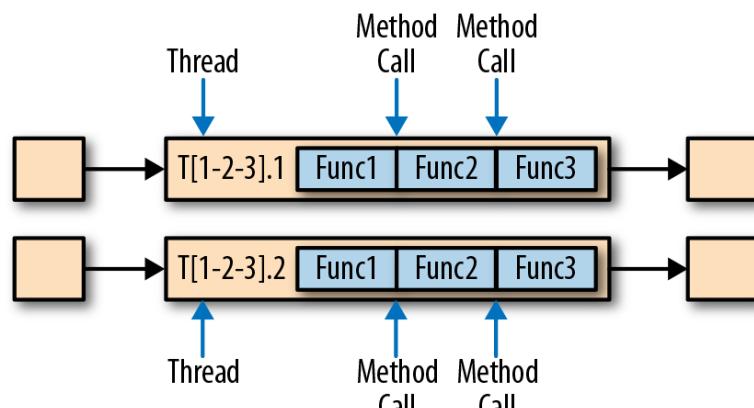


图 3.5: lambda 架构

任务链可以显著降低本地任务之间的通信成本，但也有一些场景，在没有链接的情况下运行管道操作是有意义的。例如，如果任务链中某个函数执行的开销巨大，那就将一条长的任务链管道断开，或者将一条链断开为两个任务，从而可以将这个开销大的函数调度到不同的槽（slots）中。图 3-7 显示了在没有任务链的情况下相同管道操作的执行情况。所有函数都由独立的单个任务来评估，每个任务都在专有的线程中运行。

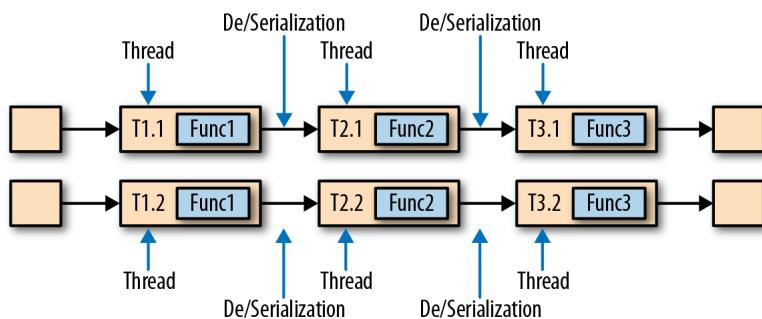


图 3.6: 任务链

任务链在 Flink 中默认会启用。在“控制任务链”一节中，我们展示了如何禁用应用程序的任务链，以及如何控制各个算子的链接行为。

3.3 事件时间处理

在“时间语义”一节，我们重点强调了时间语义在流处理应用中的重要性，并且解释了处理时间（processing time）和事件时间（event time）的不同。处理时间比较好理解，因为它是基于处理器本地时间的；但同时，它会带来比较混乱、不一致、并且不可重现的结果。相比之下，事件时间语义能够产生可重现且一致的结果，这也是许多流处理场景希望解决的一大难题。但是，与处理时间应用程序相比，事件时间应用程序会更复杂，需要额外的配置。另外，支持事件时间的流处理器，也比纯粹在处理时间中运行的系统内部更为复杂。

Flink 为常见的事件时间处理操作提供了直观且易于使用的原语，同时暴露了表达性很强的 API，用户可以使用自定义算子实现更高级的事件时间应用程序。很好地理解 Flink 的内部时间处理，对于实现这样的高级应用程序会有很大帮助，有时也是必需的。上一章介绍了 Flink 利用两个概念来支持事件时间语义：记录时间戳（timestamps）和水位线（watermarks）。接下来，我们将描述 Flink 如何在内部实现并处理时间戳和水位线，进而支持具有事件时间语义的流式应用程序。

3.3.1 时间戳

由 Flink 事件时间流应用程序处理的所有记录都必须伴有时间戳。时间戳将数据与特定时间点相关联，通常就是数据所表示的事件发生的时间点。而只要时间戳大致跟数据流保持一致，基本上随着数据流的前进而增大，应用程序就可以自由选择时间戳的含义。不过正如“时间语义”一节中所讨论的，在现实场景中，时间戳基本上都是乱序的，所以采用“事件时间”而非“处理事件”往往会显得更为重要。

当 Flink 以事件时间模式处理数据流时，它会根据数据记录的时间戳来处理基于时间的算子。例如，时间窗口算子根据相关时间戳将数据分配给不同的时间窗口。Flink 将时间戳编码为 16 字节的长整型值，并将其作为元数据附加到数据记录中。它的内置运算符会将这个长整型值解释为一个具有毫秒精度的 Unix 时间戳，也就是 1970-01-01-00:00:00.000 以来的毫秒数。当然，如果用户进行了自定义，那么运算符可以有自己的解释，例如，可以将精度调整到微秒。

3.3.2 水位线

除了时间戳，基于事件时间的 Flink 应用程序还必须支持水位线（watermark）。在基于事件时间的应用中，水位线用于生成每个任务的当前事件时间。基于时间的算子使用这个“当前事件时间”来触发计算和处理操作。例如，一个时间窗口任务（time-window task）会在任务的事件时间超出窗口的关闭边界时，完成窗口计算，并输出计算结果。

在 Flink 中，水位线被实现为一条特殊的数据记录，它里面以长整型值保存了一个时间戳。水位线在带有时间戳的数据流中，跟随着其它数据一起流动，如图 3-8 所示。

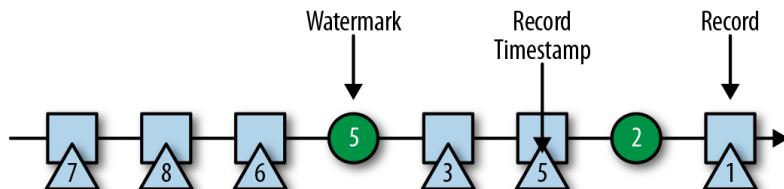


图 3.7: 水位线

水位线有两个基本属性：

- 必须单调递增，以确保任务的事件时间时钟在向前推进，而不是在后退。
- 它们与数据的时间戳相关。带有时间戳 T 的水位线表示，所有后续数据的时间戳都应该大于 T 。

上面的第二个属性用于处理带有乱序时间戳的数据流，比如图 3-8 中时间戳 3 和 5 的数据。基于时间的算子任务会收集和处理数据（这些数据可能具有乱序的时间戳），并在事件时间时钟到达某个时刻时完成计算。这个时刻就表示数据收集的截止，具有之前时间戳的数据应该都已经到达、不再需要了；而其中的事件时间时钟，正是由当前接收到的水位线来指示的。如果任务再接收到的数据违反了 watermark 的这一属性，也就是时间戳小于以前接收到的水位线时，它所属的那部分计算可能已经完成了。这种数据被称为延迟数据（late records）。Flink 提供了处理延迟数据的不同方式，我们会在“处理延迟数据”一节中讨论。

水位线还有一个很有趣的特性，它允许应用程序自己来平衡结果的完整性和延迟。如果水位线与数据的时间戳非常接近，那么我们可以得到较低的处理延迟，因为任务在完成计算之前只会短暂地等待更多数据到达。而同时，结果的完整性可能会受到影响，因为相关数据可能因为迟到而被视为“延迟数据”，这样就不会包含在结果中。相反，非常保守的水位线提供了足够的时间去等待所有数据到达，这样会增加处理延迟，但提高了结果的完整性。

3.3.3 watermark 的传递和事件时间

在本节中，我们将讨论算子如何处理水位线。Flink 把 watermark 作为一条特殊的数据来实现，它也会由算子任务接收和发送。任务会有一个内部的时间服务，它会维护定时器，并在收到 watermark 时触发。任务可以在计时器服务中注册定时器，以便在将来特定的时间点执行计算。例如，窗口算子为每个活动窗口注册一个定时器，当事件时间超过窗口的结束时间时，该计时器将清除窗口的状态。

当任务收到 watermark 时，将执行以下操作：

- 任务根据 watermark 的时间戳更新其内部事件时钟。
- 任务的时间服务会将所有过期的计时器标识出来，它们的时间小于当前的事件时间。对于每个过期的计时器，任务调用一个回调函数，该函数可以执行计算并发送结果。

- 任务会发出一个带有更新后的事件时间的 watermark。



笔记 Flink 限制通过 DataStream API 访问时间戳和 watermark。函数不能读取或修改数据的时间戳和 watermark，但底层的“处理函数”（process functions）除外，它们可以读取当前处理数据的时间戳、请求算子的当前事件时间，还可以注册定时器。通常的函数都不会暴露这些可以设置时间戳、操作任务事件时间时钟、或者发出水位线的 API。而基于时间的数据流算子任务则会配置发送出的数据的时间戳，以确保它们能够与已到达的水位线平齐。例如，窗口计算完成后，时间窗口的算子任务会将窗口的结束时间作为时间戳附加到将要发送出的结果数据上，然后再使用触发窗口计算的时间戳发出 watermark。

现在，让我们更详细地解释一下任务在接收到新的 watermark 时，如何继续发送 watermark 并更新其事件时钟。正如我们在“数据并发和任务并发”中所了解的，Flink 将数据流拆分为多个分区，并通过单独的算子任务并行地处理每个分区。每个分区都是一个流，里面包含了带着时间戳的数据和 watermark。一个算子与它前置或后续算子的连接方式有多种情况，所以它对应的任务可以从一个或多个“输入分区”接收数据和 watermark，同时也可以将数据和 watermark 发送到一个或多个“输出分区”。接下来，我们将详细描述一个任务如何向多个输出任务发送 watermark，以及如何通过接收到的 watermark 来驱动事件时间时钟前进。

任务为每个输入分区维护一个分区水位线（watermark）。当从一个分区接收到 watermark 时，它会比较新接收到的值和当前水位值，然后将相应的分区 watermark 更新为两者最大值。然后，任务会比较所有分区 watermark 的大小，将其事件时钟更新为所有分区 watermark 的最小值。如果事件时间时钟前进了，任务就将处理所有被触发的定时器操作，并向所有连接的输出分区发送出相应的 watermark，最终将新的事件时间广播给所有下游任务。

图 3-9 显示了具有四个输入分区和三个输出分区的任务如何接收 watermark、更新分区 watermark 和事件时间时钟，以及向下游发出 watermark。

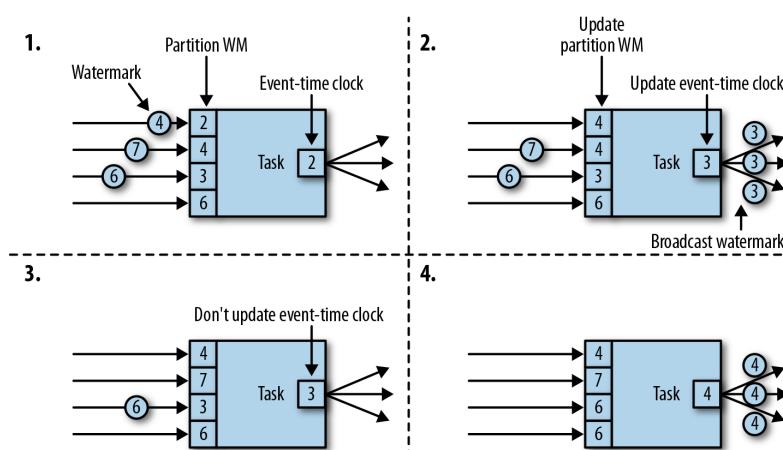


图 3.8: 水位线传播

具有两个或多个输入流（如 Union 或 CoFlatMap）的算子任务（参见“多流转换”一节）也会以所有分区 watermark 的最小值作为事件时间时钟。它们并不区分不同输入流的分区 watermark，所以两个输入流的数据都是基于相同的事件时间时钟进行处理的。当

然我们可以想到，如果应用程序的各个输入流的事件时间不一致，那么这种处理方式可能会导致问题。

Flink 的水位处理和传递算法，确保了算子任务发出的时间戳和 watermark 是“对齐”的。不过它依赖一个条件，那就是所有分区都会提供不断增长的 watermark。一旦一个分区不再推进水位线的上升，或者完全处于空闲状态、不再发送任何数据和 watermark，任务的事件时间时钟就将停滞不前，任务的定时器也就无法触发了。对于基于时间的算子来说，它们需要依赖时钟的推进来执行计算和清除状态，这种情况显然就会有问题。如果任务没有定期从所有输入任务接收到新的 watermark，那么基于时间的算子的处理延迟和状态空间的大小都会显著增加。

对于具有两个输入流而且 watermark 明显不同的算子，也会出现类似的情况。具有两个输入流的任务的事件时间时钟，将同较慢的那条流的 watermark 保持一致，而通常较快流的数据或者中间结果会在 state 中缓冲，直到事件时间时钟达到这条流的 watermark，才会允许处理它们。

3.3.4 时间戳的分配和水位线的产生

我们已经解释了什么是时间戳和水位线，以及它们是如何由 Flink 内部处理的；然而我们还没有讨论它们的产生。流应用程序接收到数据流时，通常就会先分配时间戳并生成水位线（watermark）。因为时间戳的选择是由不同的应用程序决定的，而且 watermark 取决于时间戳和流的特性，所以应用程序必须首先显式地分配时间戳并生成 watermark。Flink 流应用程序可以通过三种方式分配时间戳和生成 watermark：

- 在数据源（source）处分配：当数据流被摄入到应用程序中时，可以由“源函数”`SourceFunction` 分配和生成时间戳和 watermark。`SourceFunction` 可以产生并发送一个数据流；数据会与相关的时间戳一起发送出去，而 watermark 可以作为一条特殊数据在任何时间点发出。如果 `SourceFunction`（暂时）不再发出 watermark，它可以声明自己处于“空闲”（idle）状态。Flink 会在后续算子的水位计算中，把空闲的 `SourceFunction` 产生的流分区排除掉。`source` 的这一空闲机制，可以用来解决前面提到的水位不再上升的问题。源函数（`Source Function`）在“实现自定义源函数”一节中进行了更详细的讨论。
- 定期分配：在 Flink 中，`DataStream API` 提供一个名为 `AssignerWithPeriodicWatermarks` 的用户定义函数，它可以从每个数据中提取时间戳，并被定期调用以生成当前 watermark。提取出的时间戳被分配给相应的数据，而生成的 watermark 也会添加到流中。这个函数将在“分配时间戳和生成水位线”一节中讨论。
- 间断分配：`AssignerWithPunctuatedWatermarks` 是另一个用户定义的函数，它同样会从每个数据中提取一个时间戳。它可用于生成特殊输入数据中的 watermark。与 `AssignerWithPeriodicWatermarks` 相比，此函数可以（但不是必须）从每个记录中提取 watermark。我们在“分配时间戳和生成水位线”一节中同样讨论了该函数。

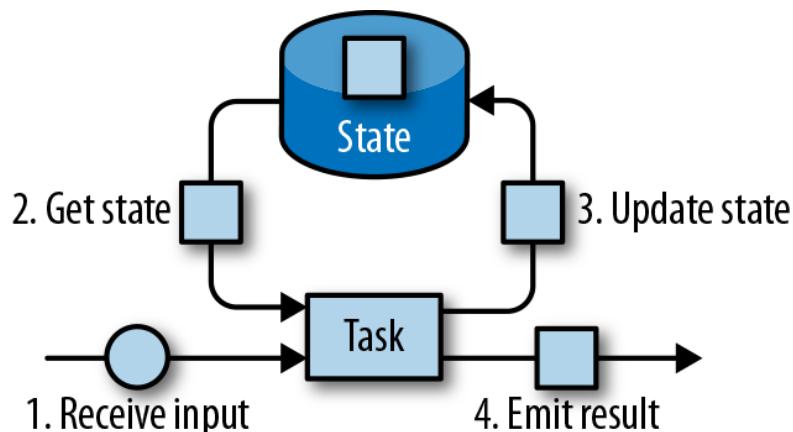
用户定义的时间戳分配函数并没有严格的限制，通常会放在尽可能靠近 source 算子的位置，因为当经过一些算子处理后，数据及其时间戳的顺序就更加难以解释了。所以

尽管我们可以在流应用程序的中段覆盖已有的时间戳和 watermark——Flink 通过用户定义的函数提供了这种灵活性，但这显然并不是推荐的做法。

3.4 状态管理

在第 2 章中，我们已经知道大多数流应用程序都是有状态的。许多算子会不断地读取和更新状态，例如在窗口中收集的数据、读取输入源的位置，或者像机器学习模型那样的用户定制化的算子状态。Flink 用同样的方式处理所有的状态，无论是内置的还是用户自定义的算子。本节我们将会讨论 Flink 支持的不同类型的状态，并解释“状态后端”是如何存储和维护状态的。

一般来说，由一个任务维护，并且用来计算某个结果的所有数据，都属于这个任务的状态。你可以认为状态就是一个本地变量，可以被任务的业务逻辑访问。图 3-10 显示了任务与其状态之间的交互。



3.4.1 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。图 3-11 显示了任务如何访问算子状态。

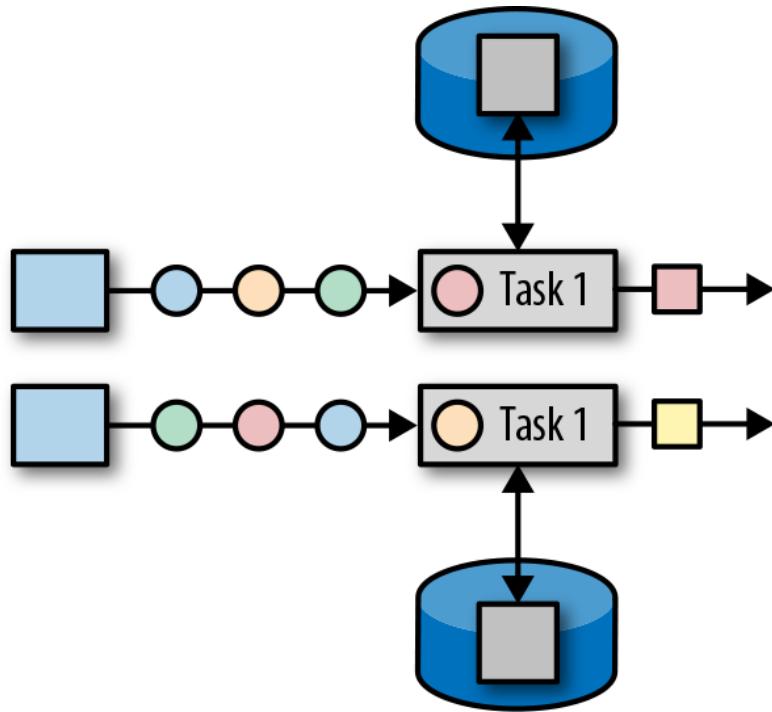


图 3.10: 算子状态

Flink 为算子状态提供三种基本数据结构：

- 列表状态：将状态表示为一组数据的列表。
- 联合列表状态：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。我们将在后面继续讨论。
- 广播状态：如果一个算子有多个任务，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。在保存检查点和重新调整算子并行度时，会用到这个特性。这两部分内容将在本章后面讨论。

3.4.2 键控状态

顾名思义，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。图 3-12 显示了任务如何与键控状态进行交互。

我们可以将键控状态看成是在算子所有并行任务上，对键进行分区（或分片）之后的一个键值映射（key-value map）。Flink 为键控状态提供不同的数据结构，用于确定 map

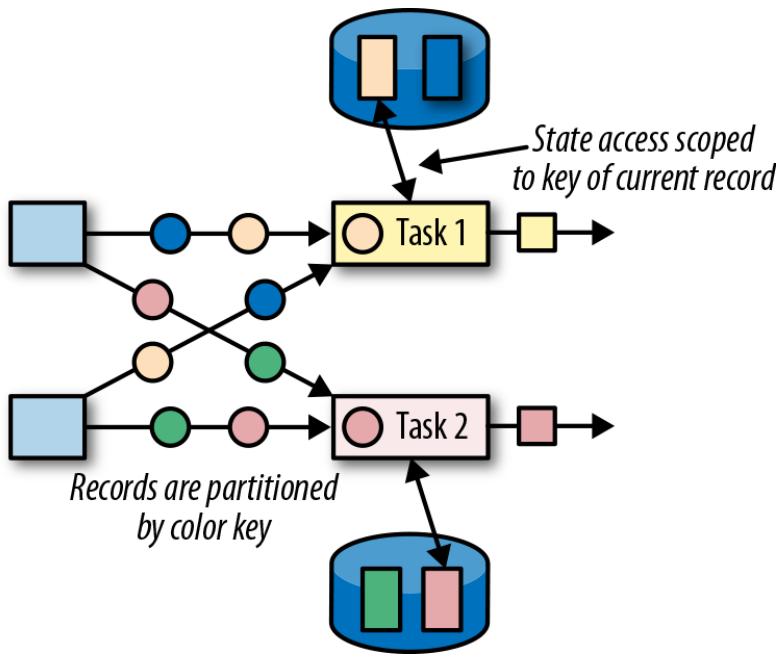


图 3.11: 键控状态

中每个 key 存储的值的类型。我们简单了解一下最常见的键控状态。

- 值状态：为每个键存储一个任意类型的单个值。复杂数据结构也可以存储为值状态。
- 列表状态：为每个键存储一个值的列表。列表里的每个数据可以是任意类型。
- 映射状态：为每个键存储一个键值映射（map）。map 的 key 和 value 可以是任意类型。

状态的数据结构可以让 Flink 实现更有效的状态访问。我们将在“在运行时上下文（RuntimeContext）中声明键控状态”中做进一步讨论。

3.4.3 状态后端

每传入一条数据，有状态的算子任务都会读取和更新状态。由于有效的状态访问对于处理数据的低延迟至关重要，因此每个并行任务都会在本地维护其状态，以确保快速的状态访问。状态到底是如何被存储、访问以及维护的？这件事由一个可插入的组件决定，这个组件就叫做状态后端（state backend）。状态后端主要负责两件事：本地的状态管理，以及将检查点（checkpoint）状态写入远程存储。

对于本地状态管理，状态后端会存储所有键控状态，并确保所有的访问都被正确地限定在当前键范围。Flink 提供了默认的状态后端，会将键控状态作为内存中的对象进行管理，将它们存储在 JVM 堆上。另一种状态后端则会把状态对象进行序列化，并将它们放入 RocksDB 中，然后写入本地硬盘。第一种方式可以提供非常快速的状态访问，但它受内存大小的限制；而访问 RocksDB 状态后端存储的状态速度会较慢，但其状态可以增长到非常大。

状态检查点的写入也非常重要，这是因为 Flink 是一个分布式系统，而状态只能在本地维护。TaskManager 进程（所有任务在其上运行）可能在任何时间点挂掉。因此，它的本地存储只能被认为是不稳定的。状态后端负责将任务的状态检查点写入远程的持久

存储。写入检查点的远程存储可以是分布式文件系统，也可以是数据库。不同的状态后端在状态检查点的写入机制方面有所不同。例如，RocksDB 状态后端支持增量的检查点，这对于非常大的状态来说，可以显著减少状态检查点写入的开销。

我们将在“选择状态后端”一节中更详细地讨论不同的状态后端及其优缺点。

3.4.4 调整有状态算子的并行度

流应用程序的一个常见要求是，为了增大或较小输入数据的速率，需要灵活地调整算子的并行度。对于无状态算子而言，并行度的调整没有任何问题，但更改有状态算子的并行度显然就没那么简单了，因为它们的状态需要重新分区并分配给更多或更少的并行任务。Flink 支持四种模式来调整不同类型的状态。

具有键控状态的算子通过将键重新分区为更少或更多任务来缩放并行度。不过，并行度调整时任务之间会有一些必要的状态转移。为了提高效率，Flink 并不会对单独的 key 做重新分配，而是用所谓的“键组”（key group）把键管理起来。键组是 key 的分区形式，同时也是 Flink 为任务分配 key 的方式。图 3-13 显示了如何在键组中重新分配键控状态。

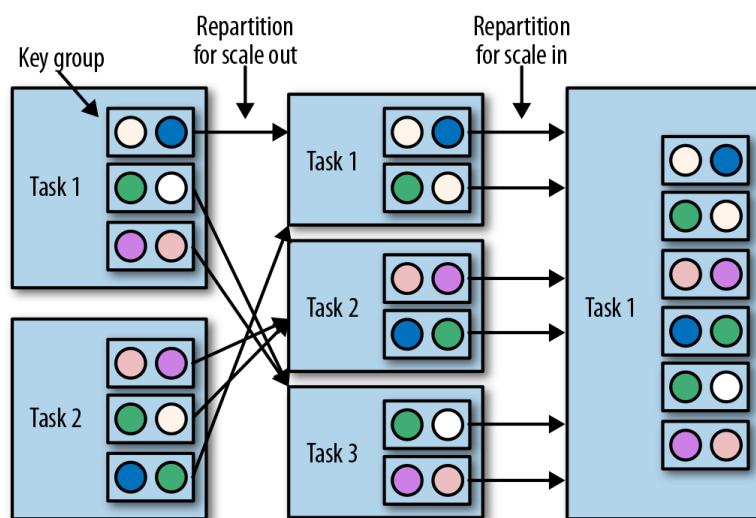


图 3.12: lambda 架构

具有算子列表状态的算子，会通过重新分配列表中的数据项目来进行并行度缩放。从概念上讲，所有并行算子任务的列表项目会被收集起来，并将其均匀地重新分配给更少或更多的任务。如果列表条目少于算子的新并行度，则某些任务将以空状态开始。图 3-14 显示了算子列表状态的重新分配。

具有算子联合列表状态的算子，会通过向每个任务广播状态的完整列表，来进行并行度的缩放。然后，任务可以选择要使用的状态项和要丢弃的状态项。图 3-15 显示了如何重新分配算子联合列表状态。

具有算子广播状态的算子，通过将状态复制到新任务，来增大任务的并行度。这是没问题的，因为广播状态保证了所有任务都具有相同的状态。而对于缩小并行度的情况，我们可以直接取消剩余任务，因为状态是相同的，已经被复制并且不会丢失。图 3-16 显示了算子广播状态的重新分配。

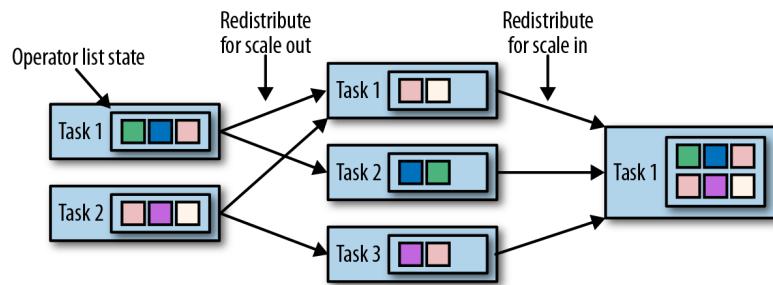


图 3.13: lambda 架构

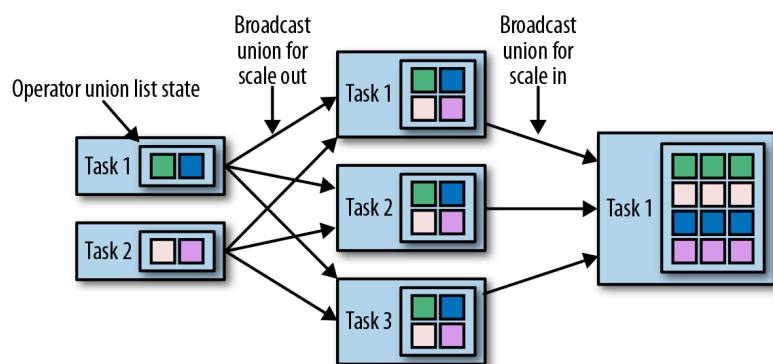


图 3.14: lambda 架构

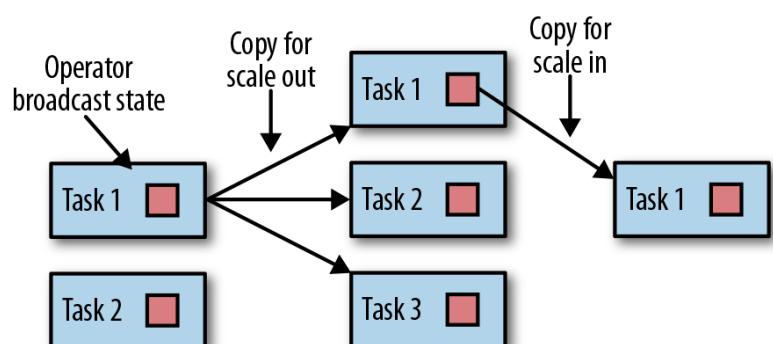


图 3.15: lambda 架构

3.5 检查点，保存点和状态恢复

Flink 是一个分布式数据处理系统，因此必须有一套机制处理各种故障，比如被杀掉的进程，故障的机器和中断的网络连接。任务都是在本地维护状态的，所以 Flink 必须确保状态不会丢失，并且在发生故障时能够保持一致。

在本节中，我们将介绍 Flink 的检查点（checkpoint）和恢复机制，这保证了“精确一次”（exactly-once）的状态一致性。我们还会讨论 Flink 独特的保存点（savepoint）功能，这是一个“瑞士军刀”式的工具，可以解决许多操作数据流时面对的问题。

3.5.1 一致的检查点

Flink 的恢复机制的核心，就是应用状态的一致检查点。有状态流应用的一致检查点，其实就是所有任务状态在某个时间点的一份拷贝，而这个时间点应该是所有任务都恰好处理完一个相同的输入数据的时候。这个过程可以通过一致检查点的一个简单算法步骤来解释。这个算法的步骤是：

- 暂停所有输入流的摄取，也就是不再接收新数据的输入。
- 等待所有正在处理的数据计算完毕，这意味着结束时，所有任务都已经处理了所有输入数据。
- 通过将每个任务的状态复制到远程持久存储，来得到一个检查点。所有任务完成拷贝操作后，检查点就完成了。
- 恢复所有输入流的摄取。

需要注意，Flink 实现的并不是这种简单的机制。我们将在本节后面介绍 Flink 更精妙的检查点算法。

图 3-17 显示了一个简单应用中的一致检查点。

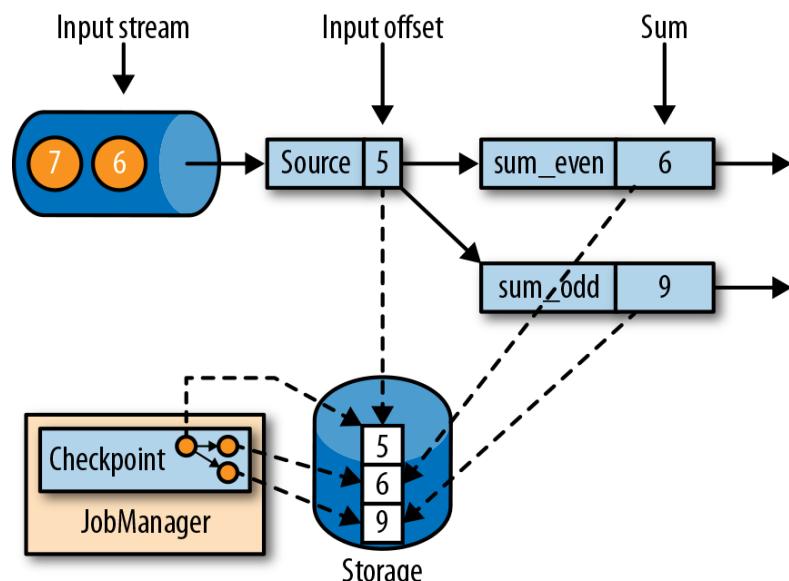


图 3.16: lambda 架构

上面的应用程序中具有单一的输入源（source）任务，输入数据就是一组不断增长的

数字的流——1,2,3 等。数字流被划分为偶数流和奇数流。求和算子（sum）的两个任务会分别实时计算当前所有偶数和奇数的总和。源任务会将其输入流的当前偏移量存储为状态，而求和任务则将当前的总和值存储为状态。在图 3-17 中，Flink 在输入偏移量为 5 时，将检查点写入了远程存储，当前的总和为 6 和 9。

3.5.2 从一致检查点中恢复状态

在执行流应用程序期间，Flink 会定期检查状态的一致检查点。如果发生故障，Flink 将会使用最近的检查点来一致恢复应用程序的状态，并重新启动处理流程。图 3-18 显示了恢复过程。

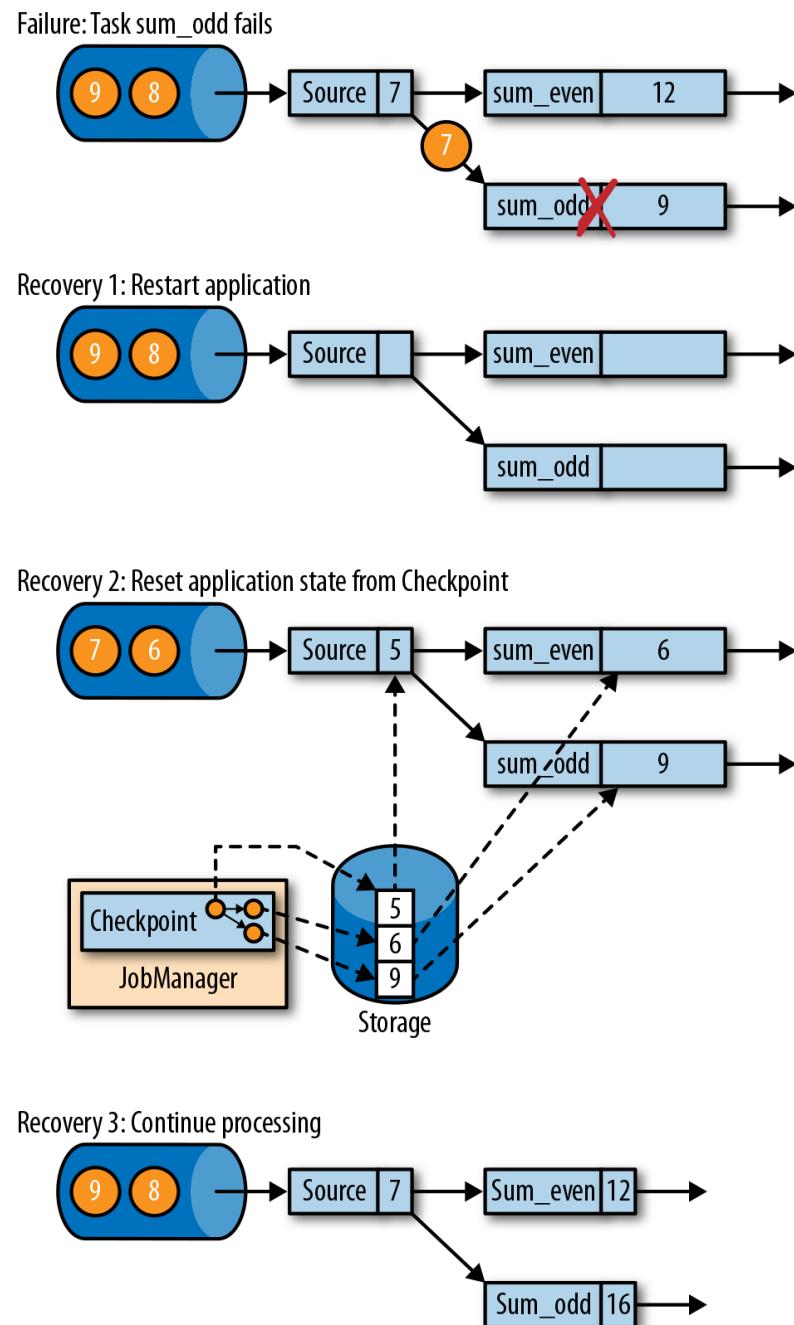


图 3.17: lambda 架构

应用程序从检查点的恢复分为三步：

- 重新启动整个应用程序。
- 将所有的有状态任务的状态重置为最近一次的检查点。
- 恢复所有任务的处理。

这种检查点的保存和恢复机制可以为应用程序状态提供“精确一次”（exactly-once）的一致性，因为所有算子都会保存检查点并恢复其所有状态，这样一来所有的输入流就都会被重置到检查点完成时的位置。至于数据源是否可以重置它的输入流，这取决于其实现方式和消费流数据的外部接口。例如，像 Apache Kafka 这样的事件日志系统可以提供流上之前偏移位置的数据，所以我们可以将源重置到之前的偏移量，重新消费数据。而从套接字（socket）消费数据的流就不能被重置了，因为套接字的数据一旦被消费就会丢弃掉。因此，对于应用程序而言，只有当所有的输入流消费的都是可重置的数据源时，才能确保在“精确一次”的状态一致性下运行。

从检查点重新启动应用程序后，其内部状态与检查点完成时的状态完全相同。然后它就会开始消费并处理检查点和发生故障之间的所有数据。尽管这意味着 Flink 会对一些数据处理两次（在故障之前和之后），我们仍然可以说这个机制实现了精确一次的一致性语义，因为所有算子的状态都已被重置，而重置后的状态下还不曾看到这些数据。

我们必须指出，Flink 的检查点保存和恢复机制仅仅可以重置流应用程序的内部状态。对于应用中的一些的输出（sink）算子，在恢复期间，某些结果数据可能会多次发送到下游系统，比如事件日志、文件系统或数据库。对于某些存储系统，Flink 提供了具有精确一次输出功能的 sink 函数，比如，可以在检查点完成时提交发出的记录。另一种适用于许多存储系统的方法是幂等更新。在“应用程序一致性保证”一节中，我们还会详细讨论如何解决应用程序端到端的精确一次一致性问题。

3.5.3 Flink 的检查点算法

Flink 的恢复机制，基于它的一致性检查点。前面我们已经了解了从流应用中创建检查点的简单方法——先暂停应用，保存检查点，然后再恢复应用程序，这种方法很好理解，但它的理念是“停止一切”，这对于即使是中等延迟要求的应用程序而言也是不实用的。所以 Flink 没有这么简单粗暴，而是基于 Chandy-Lamport 算法实现了分布式快照的检查点保存。该算法并不会暂停整个应用程序，而是将检查点的保存与数据处理分离，这样就可以实现在其它任务做检查点状态保存状态时，让某些任务继续进行而不受影响。接下来我们将解释此算法的工作原理。

Flink 的检查点算法用到了一种称为“检查点分界线”（checkpoint barrier）的特殊数据形式。与水位线（watermark）类似，检查点分界线由 source 算子注入到常规的数据流中，它的位置是限定好的，不能超过其他数据，也不能被后面的数据超过。检查点分界线带有检查点 ID，用来标识它所属的检查点；这样，这个分界线就将一条流逻辑上分成了两部分。分界线之前到来的数据导致的状态更改，都会被包含在当前分界线所属的检查点中；而基于分界线之后的数据导致的所有更改，就会被包含在之后的检查点中。

我们用一个简单的流应用程序作为示例，来一步一步解释这个算法。该应用程序有

两个源（source）任务，每个任务都消费一个增长的数字流。源任务的输出被划分为两部分：偶数和奇数的流。每个分区由一个任务处理，该任务计算所有收到的数字的总和，并将更新的总和转发给输出（sink）任务。这个应用程序的结构如图 3-19 所示。

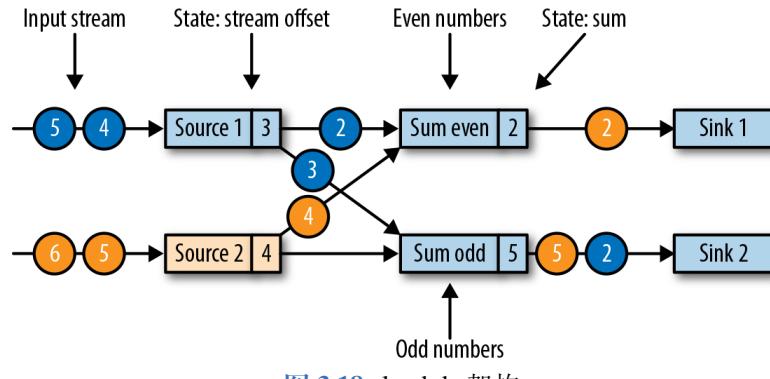


图 3.18: lambda 架构

作业管理器会向每个数据源（source）任务发送一条带有新检查点 ID 的消息，通过这种方式来启动检查点，如图 3-20 所示。

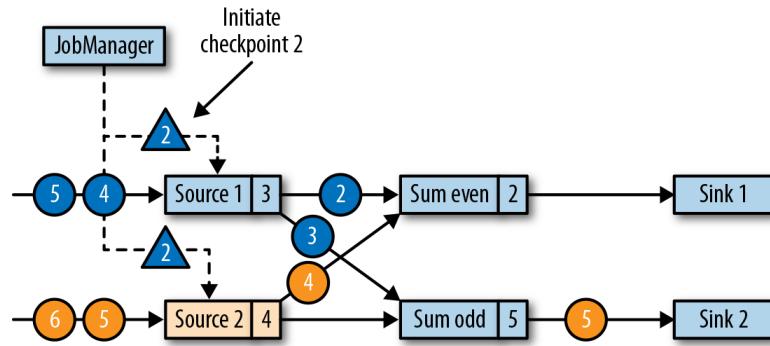


图 3.19: lambda 架构

当 source 任务收到消息时，它会暂停发出新的数据，在状态后端触发本地状态的检查点保存，并向所有传出的流分区广播带着检查点 ID 的分界线（barriers）。状态后端在状态检查点完成后会通知任务，而任务会向作业管理器确认检查点完成。在发出所有分界线后，source 任务就可以继续常规操作，发出新的数据了。通过将分界线注入到输出流中，源函数（source function）定义了检查点在流中所处的位置。图 3-21 显示了两个源任务将本地状态保存到检查点，并发出检查点分界线之后的流应用程序。

源任务发出的检查点分界线（barrier），将被传递给所连接的任务。与水位线（watermark）类似，barrier 会被广播到所有连接的并行任务，以确保每个任务从它的每个输入流中都能接收到。当任务收到一个新检查点的 barrier 时，它会等待这个检查点的所有输入分区的 barrier 到达。在等待的过程中，任务并不会闲着，而是会继续处理尚未提供 barrier 的流分区中的数据。对于那些 barrier 已经到达的分区，如果继续有新的数据到达，它们就不会被立即处理，而是先缓存起来。这个等待所有分界线到达的过程，称为“分界线对齐”（barrier alignment），如图 3-22 所示。

当任务从所有输入分区都收到 barrier 时，它就会在状态后端启动一个检查点的保存，并继续向所有下游连接的任务广播检查点分界线，如图 3-23 所示。

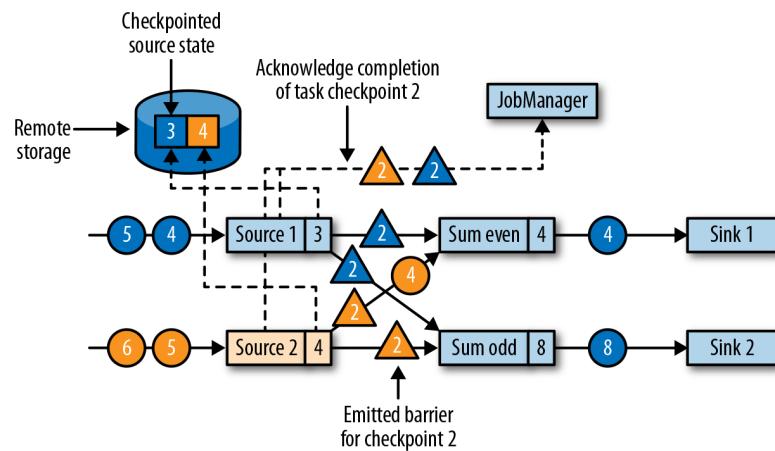


图 3.20: lambda 架构

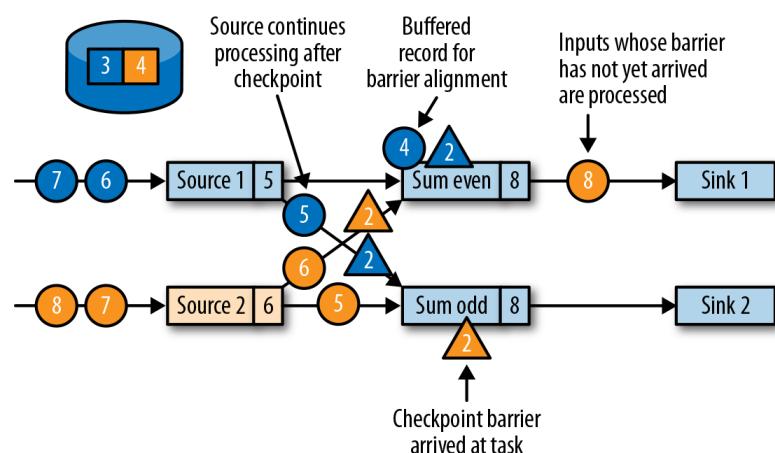


图 3.21: lambda 架构

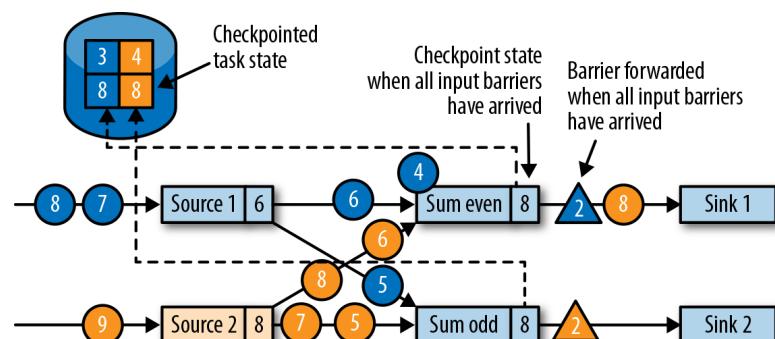


图 3.22: lambda 架构

所有的检查点 barrier 都发出后，任务就开始处理之前缓冲的数据。在处理并发出所有缓冲数据之后，任务就可以继续正常处理输入流了。图 3-24 显示了此时的应用程序。

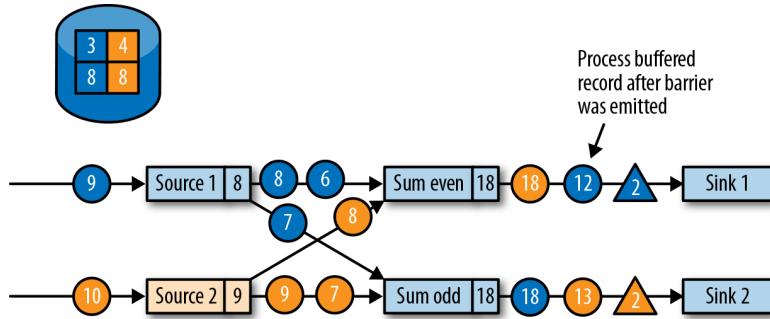


图 3.23: lambda 架构

最终，检查点分界线会到达输出（sink）任务。当 sink 任务接收到 barrier 时，它也会先执行“分界线对齐”，然后将自己的状态保存到检查点，并向作业管理器确认已接收到 barrier。一旦从应用程序的所有任务收到一个检查点的确认信息，作业管理器就会将这个检查点记录为已完成。图 3-25 显示了检查点算法的最后一步。这样，当发生故障时，我们就可以用已完成的检查点恢复应用程序了。

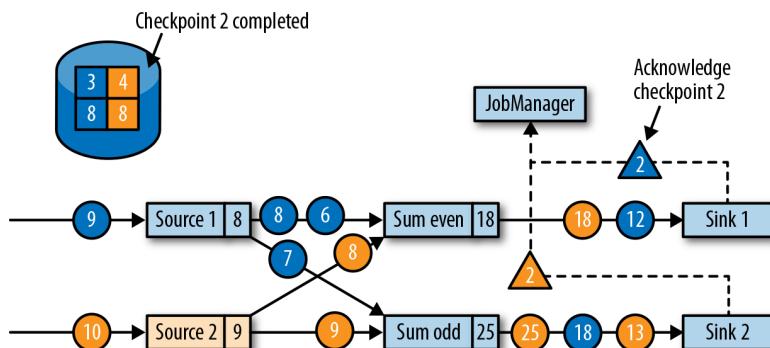


图 3.24: lambda 架构

3.5.4 检查点的性能影响

Flink 的检查点算法可以在不停止整个应用程序的情况下，生成一致的分布式检查点。但是，它可能会增加应用程序的处理延迟。Flink 对此有一些调整措施，可以在某些场景下显得对性能的影响没那么大。

当任务将其状态保存到检查点时，它其实处于一个阻塞状态，而此时新的输入会被缓存起来。由于状态可能变得非常大，而且检查点需要通过网络将数据写入远程存储系统，检查点的写入很容易就会花费几秒到几分钟的时间——这对于要求低延迟的应用程序而言，显然是不可接受的。在 Flink 的设计中，真正负责执行检查点写入的，其实是状态后端。具体怎样复制任务的状态，取决于状态后端的实现方式。例如，文件系统（FileSystem）状态后端和 RocksDB 状态后端都支持了异步（asynchronous）检查点。触发检查点操作时，状态后端会先创建状态的本地副本。本地拷贝完成后，任务就将继续常规的数据处理，这往往并不会花费太多时间。一个后台线程会将本地快照异步复制到远程存储，并

在完成检查点后再回来通知任务。异步检查点的机制，显著减少了任务继续处理数据之前的等待时间。此外，RocksDB 状态后端还实现了增量的检查点，这样可以大大减少要传输的数据量。

为了减少检查点算法对处理延迟的影响，另一种技术是调整分界线对齐的步骤。对于需要非常低的延迟、并且可以容忍“至少一次”(at-least-once)状态保证的应用程序，Flink 可以将检查点算法配置为，在等待 barrier 对齐期间处理所有到达的数据，而不是把 barrier 已经到达的那些分区的数据缓存起来。当检查点的所有 barrier 到达，算子任务就会将状态写入检查点——当然，现在的状态中，就可能包括了一些“提前”的更改，这些更改由本该属于下一个检查点的数据到来时触发。如果发生故障，从检查点恢复时，就将再次处理这些数据：这意味着检查点现在提供的是“至少一次”(at-least-once)而不是“精确一次”(exactly-once)的一致性保证。

3.5.5 保存点

Flink 的恢复算法是基于状态检查点的。Flink 根据可配置的策略，定期保存并自动丢弃检查点。检查点的目的是确保在发生故障时可以重新启动应用程序，所以当应用程序被显式地撤销(cancel)时，检查点会被删除掉。除此之外，应用程序状态的一致性快照还可用于除故障恢复之外的更多功能。

Flink 中一个最有价值，也是最独特的功能是保存点(savepoints)。原则上，创建保存点使用的算法与检查点完全相同，因此保存点可以认为就是具有一些额外元数据的检查点。Flink 不会自动创建保存点，因此用户（或者外部调度程序）必须明确地触发创建操作。同样，Flink 也不会自动清理保存点。第 10 章将会具体介绍如何触发和处理保存点。

使用保存点

有了应用程序和与之兼容的保存点，我们就可以从保存点启动应用程序了。这会将应用程序的状态初始化为保存点的状态，并从保存点创建时的状态开始运行应用程序。虽然看起来这种行为似乎与用检查点从故障中恢复应用程序完全相同，但实际上故障恢复只是一种特殊情况，它只是在相同的集群上以相同的配置启动相同的应用程序。而从保存点启动应用程序会更加灵活，这就可以让我们做更多事情了。

- 可以从保存点启动不同但兼容的应用程序。这样一来，我们就可以及时修复应用程序中的逻辑 bug，并让流式应用的源尽可能多地提供之前发生的事件，然后重新处理，以便修复之前的计算结果。修改后的应用程序还可用于运行 A/B 测试，或者具有不同业务逻辑的假设场景。这里要注意，应用程序和保存点必须兼容才可以这么做——也就是说，应用程序必须能够加载保存点的状态。
- 可以使用不同的并行度来启动相同的应用程序，可以将应用程序的并行度增大或减小。
- 可以在不同的集群上启动同样的应用程序。这非常有意义，意味着我们可以将应用程序迁移到较新的 Flink 版本或不同的集群上去。
- 可以使用保存点暂停应用程序，稍后再恢复。这样做的意义在于，可以为更高优先

级的应用程序释放集群资源，或者在输入数据不连续生成时释放集群资源。

- 还可以将保存点设置为某一版本，并归档（archive）存储应用程序的状态。

保存点是非常强大的功能，所以许多用户会定期创建保存点以便能够及时退回之前的状态。我们见到的各种场景中，保存点一个最有趣的应用是不断将流应用程序迁移到更便宜的数据中心上去。

从保存点启动应用程序

前面提到的保存点的所有用例，都遵循相同的模式。那就是首先创建正在运行的应用程序的保存点，然后在一个新启动的应用程序中用它来恢复状态。之前我们已经知道，保存点的创建和检查点非常相似，而接下来我们就将介绍对于一个从保存点启动的应用程序，Flink 如何初始化其状态。

应用程序由多个算子组成。每个算子可以定义一个或多个键控状态和算子状态。算子由一个或多个算子任务并行执行。因此，一个典型的应用程序会包含多个状态，这些状态分布在多个算子任务中，这些任务可以运行在不同的 TaskManager 进程上。

图 3-26 显示了一个具有三个算子的应用程序，每个算子执行两个算子任务。一个算子 (OP-1) 具有单一的算子状态 (OS-1)，而另一个算子 (OP-2) 具有两个键控状态 (KS-1 和 KS-2)。当保存点创建时，会将所有任务的状态复制到持久化的存储位置。

保存点中的状态拷贝会以算子标识符 (operator ID) 和状态名称 (state name) 组织起来。算子 ID 和状态名称必须能够将保存点的状态数据，映射到一个正在启动的应用程序的算子状态。从保存点启动应用程序时，Flink 会将保存点的数据重新分配给相应的算子任务。



笔记 请注意，保存点不包含有关算子任务的信息。这是因为当应用程序以不同的并行度启动时，任务数量可能会更改。

如果我们要从保存点启动一个修改过的应用程序，那么保存点中的状态只能映射到符合标准的应用程序——它里面的算子必须具有相应的 ID 和状态名称。默认情况下，Flink 会自动分配唯一的算子 ID。然而，一个算子的 ID，是基于它之前算子的 ID 确定性地生成的。因此，算子的 ID 会在其前序算子改变时改变，比如，当我们添加了新的或移除掉一个算子时，前序算子 ID 改变，当前算子 ID 就会变化。所以对于具有默认算子 ID 的应用程序而言，如果想在不丢失状态的前提下升级，就会受到极大的限制。因此，我们强烈建议在程序中为算子手动分配唯一 ID，而不是依靠 Flink 的默认分配。我们将在“指定唯一的算子标识符”一节中详细说明如何分配算子标识符。

第四章 Flink DataStream API

4.1 读取数据源

4.1.1 定义 POJO 类

```
public static class UserBehavior {
    public String userId;
    public String itemId;
    public String categoryId;
    public String behaviorType;
    public Long timestamp;

    public UserBehavior() {
    }

    public UserBehavior(String userId, String itemId, String categoryId, String
        behaviorType, Long timestamp) {
        this.userId = userId;
        this.itemId = itemId;
        this.categoryId = categoryId;
        this.behaviorType = behaviorType;
        this.timestamp = timestamp;
    }

    @Override
    public String toString() {
        return "UserBehavior{" +
            "userId='" + userId + '\'' +
            ", itemId='" + itemId + '\'' +
            ", categoryId='" + categoryId + '\'' +
            ", behaviorType='" + behaviorType + '\'' +
            ", timestamp=" + new Timestamp(timestamp) +
            '}';
    }
}
```

4.1.2 从集合中读取

```
public static void main(String[] args) throws Exception {
    StreamExecutionEnvironment env =
        StreamExecutionEnvironment.getExecutionEnvironment();
```

```

    env.setParallelism(1);

    env.fromElements(
        new UserBehavior("543462", "1715", "1464116", "pv", 1511658000 * 1000L),
        new UserBehavior("662867", "2244074", "1575622", "pv", 1511658000 * 1000L)
    );

    ArrayList<UserBehavior> userBehaviors = new ArrayList<>();
    userBehaviors.add(new UserBehavior("543462", "1715", "1464116", "pv", 1511658000 *
        ↳ 1000L));
    userBehaviors.add(new UserBehavior("662867", "2244074", "1575622", "pv", 1511658000 *
        ↳ 1000L));

    env.fromCollection(userBehaviors);

    env.execute();
}

```

4.1.3 从文件中读取

```
env.readTextFile("UserBehavior.csv")
```

4.1.4 从 socket 读取

```
env.socketTextStream(host, port);
```

4.1.5 从 Kafka 读取

首先添加驱动

```

<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
</dependency>

```

代码如下：

```

Properties properties = new Properties();
properties.setProperty("bootstrap.servers", "localhost:9092");
properties.setProperty("group.id", "consumer-group");

```

```

properties.setProperty("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
properties.setProperty("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
properties.setProperty("auto.offset.reset", "latest");

env
    .addSource(new FlinkKafkaConsumer<String>(
        "userbehavior",
        new SimpleStringSchema(),
        properties
    ));

```

4.1.6 自定义数据源读取

数据源的定义

```

public class UserBehaviorSource extends RichParallelSourceFunction<UserBehavior> {
    private Boolean running = true;

    @Override
    public void run(SourceContext<UserBehavior> sourceContext) throws Exception {
        String strCurrentLine;
        BufferedReader reader = new BufferedReader(new FileReader("UserBehavior.csv"));
        while (running && (strCurrentLine = reader.readLine()) != null) {
            String[] arr = strCurrentLine.split(",");
            sourceContext.collect(new UserBehavior(arr[0], arr[1], arr[2], arr[3],
                Long.parseLong(arr[4]) * 1000L));
            Thread.sleep(100L);
        }
    }

    @Override
    public void cancel() {
        running = false;
    }
}

```

数据源的使用方法

```

env.addSource(new UserBehaviorSource()).print();

```

你可能已经注意到 Flink 程序的定义和提交执行使用的就是正常的 Java 的方法。大多数情况下，这些代码都写在一个静态 main 方法中。

Flink 流处理程序的结构如下：

1. 创建 Flink 程序执行环境。

2. 从数据源读取一条或者多条流数据
3. 使用流转换算子实现业务逻辑
4. 将计算结果输出到一个或者多个外部设备（可选）
5. 执行程序

4.2 基本转换算子的使用

基本转换算子的定义：作用在数据流中的每一条单独的数据上的算子。

基本转换算子会针对流中的每一个单独的事件做处理，也就是说每一个输入数据会产生一个输出数据。单值转换，数据的分割，数据的过滤，都是基本转换操作的典型例子。我们将解释这些算子的语义并提供示例代码。

MAP

下面的代码实现了将 UserBehavior 中的 itemId 字段抽取出来的功能。

```
readings.map(r -> r.itemId);

readings.map(new MapFunction<UserBehavior, String>() {
    @Override
    public String map(UserBehavior r) throws Exception {
        return r.itemId;
    }
});

readings.map(new IdExtractor());

public static class IdExtractor implements MapFunction<UserBehavior, String> {
    @Override
    public String map(UserBehavior r) throws Exception {
        return r.itemId;
    }
}
```

FILTER

```
readings.filter(r -> r.behaviorType.equals("pv"));

readings.filter(new FilterFunction<UserBehavior>() {
    @Override
    public Boolean filter(UserBehavior r) throws Exception {
        return r.behaviorType.equals("pv");
    }
});

readings.filter(new PvExtractor());
```

```

public static class IdExtractor implements FilterFunction<UserBehavior> {
    @Override
    public Boolean filter(UserBehavior r) throws Exception {
        return r.behaviorType.equals("pv");
    }
}

```

FLATMAP

```

DataStreamSource<String> stream = env.fromElements("white", "black", "gray");

stream
    .flatMap(
        (FlatMapFunction<String, String>)(s, out) -> {
            if (s.equals("white")) {
                out.collect(s);
            } else if (s.equals("black")) {
                out.collect(s);
                out.collect(s);
            }
        })
    .returns(Types.STRING)
    .print();

stream.flatMap(new MyFlatMap()).print();

public static class MyFlatMap implements FlatMapFunction<String, String> {
    @Override
    public void flatMap(String value, Collector<String> out) throws Exception {
        if (value.equals("white")) {
            out.collect(value);
        } else if (value.equals("black")) {
            out.collect(value);
            out.collect(value);
        }
    }
}

```

4.3 键控流转换算子

很多流处理程序的一个基本要求就是要能对数据进行分组，分组后的数据共享某一个相同的属性。DataStream API 提供了一个叫做 KeyedStream 的抽象，此抽象会从逻辑上对 DataStream 进行分区，分区后的数据拥有同样的 Key 值，分区后的流互不相关。

针对 KeyedStream 的状态转换操作可以读取数据或者写入数据到当前事件 Key 所对应的状态中。这表明拥有同样 Key 的所有事件都可以访问同样的状态，也就是说所以这些事件可以一起处理。

KeyedStream 可以使用 map, flatMap 和 filter 算子来处理。接下来我们会使用 keyBy 算子来将 DataStream 转换成 KeyedStream，并讲解基于 key 的转换操作：滚动聚合和 reduce 算子。

KEYBY

keyBy 通过指定 key 来将 DataStream 转换成 KeyedStream。基于不同的 key，流中的事件将被分配到不同的分区中去。所有具有相同 key 的事件将会在接下来的操作符的同一个子任务槽中进行处理。拥有不同 key 的事件可以在同一个任务中处理。但是算子只能访问当前事件的 key 所对应的状态。

keyBy() 方法接收一个参数，这个参数指定了 key 或者 keys，有很多不同的方法来指定 key。我们将在后面讲解。下面的代码声明了 itemId 这个字段为 UserBehavior 流的 key。

```
KeyedStream<UserBehavior, String> keyed = stream.keyBy(r -> r.itemId);

stream
    .keyBy(new KeySelector<UserBehavior, String>() {
        @Override
        public String getKey(UserBehavior value) throws Exception {
            return value.itemId;
        }
    });
}
```

滚动聚合

滚动聚合算子由 KeyedStream 调用，并生成一个聚合以后的 DataStream，例如：sum, minimum, maximum。一个滚动聚合算子会为每一个观察到的 key 保存一个聚合的值。针对每一个输入事件，算子将会更新保存的聚合结果，并发送一个带有更新后的值的事件到下游算子。滚动聚合不需要用户自定义函数，但需要接受一个参数，这个参数指定了在哪一个字段上面做聚合操作。DataStream API 提供了以下滚动聚合方法。

- sum(): 在输入流上对指定的字段做滚动相加操作。
- min(): 在输入流上对指定的字段求最小值。
- max(): 在输入流上对指定的字段求最大值。
- minBy(): 在输入流上针对指定字段求最小值，并返回包含当前观察到的最小值的事件。
- maxBy(): 在输入流上针对指定字段求最大值，并返回包含当前观察到的最大值的事件。

滚动聚合算子无法组合起来使用，每次计算只能使用一个单独的滚动聚合算子。

下面的例子根据第一个字段来对类型为 Tuple3<Int, Int, Int> 的流做分流操作，然后针对第二个字段做滚动求和操作。

```

DataStreamSource<Tuple3<Integer, Integer, Integer>> inputStream = env
    .fromElements(
        Tuple3.of(1, 2, 2),
        Tuple3.of(2, 3, 1),
        Tuple3.of(2, 2, 4),
        Tuple3.of(1, 5, 3)
    );

DataStream<Tuple3<Integer, Integer, Integer>> resultStream = inputStream
    .keyBy(0) // key on first field of the tuple
    .sum(1); // sum the second field of the tuple in place

```

在这个例子里面，输入流根据第一个字段来分流，然后在第二个字段上做计算。对于 key 1，输出结果是 (1,2,2),(1,7,2)。对于 key 2，输出结果是 (2,3,1),(2,5,1)。第一个字段是 key，第二个字段是求和的数值，第三个字段未定义。

REDUCE

reduce 算子是滚动聚合的泛化实现。它将一个 ReduceFunction 应用到了一个 KeyedStream 上面去。reduce 算子将会把每一个输入事件和当前已经 reduce 出来的值做聚合并计算。reduce 操作不会改变流的事件类型。输出流数据类型和输入流数据类型是一样的。

reduce 函数可以通过实现接口 ReduceFunction 来创建一个类。ReduceFunction 接口定义了 reduce() 方法，此方法接收两个输入事件，输出一个相同类型的事件。

上面的例子中的数据流求解第二个字段的最大值的元组

```

inputStream
    .keyBy(r -> r.f0)
    .reduce(new ReduceFunction<Tuple3<Integer, Integer, Integer>>() {
        @Override
        public Tuple3<Integer, Integer, Integer> reduce(Tuple3<Integer, Integer,
                Integer> value1, Tuple3<Integer, Integer, Integer> value2) throws Exception {
            if (value1.f1 > value2.f1) {
                return value1;
            } else {
                return value2;
            }
        }
    })
    .print();

```

4.4 分布式转换算子

分区操作对应于我们之前讲过的“数据交换策略”这一节。这些操作定义了事件如何分配到不同的任务中去。当我们使用 DataStream API 来编写程序时，系统将自动的选择

数据分区策略，然后根据操作符的语义和设置的并行度将数据路由到正确的地方去。有些时候，我们需要在应用程序的层面控制分区策略，或者自定义分区策略。例如，如果我们知道会发生数据倾斜，那么我们想要针对数据流做负载均衡，将数据流平均发送到接下来的操作符中去。又或者，应用程序的业务逻辑可能需要一个算子所有的并行任务都需要接收同样的数据。再或者，我们需要自定义分区策略的时候。在这一小节，我们将展示 DataStream 的一些方法，可以使我们来控制或者自定义数据分区策略。



笔记 `keyBy()` 方法不同于分布式转换算子。所有的分布式转换算子将产生 DataStream 数据类型。而 `keyBy()` 产生的类型是 KeyedStream，它拥有自己的 keyed state。

Random

随机数据交换由 `DataStream.shuffle()` 方法实现。`shuffle` 方法将数据随机的分配到下游算子的并行任务中去。

Round-Robin

`rebalance()` 方法使用 Round-Robin 负载均衡算法将输入流平均分配到随后的并行运行的任务中去。图 5-7 为 round-robin 分布式转换算子的示意图。

Rescale

`rescale()` 方法使用的也是 round-robin 算法，但只会将数据发送到接下来的并行运行的任务中的一部分任务中。本质上，当发送者任务数量和接收者任务数量不一样时，`rescale` 分区策略提供了一种轻量级的负载均衡策略。如果接收者任务的数量是发送者任务的数量的倍数时，`rescale` 操作将会效率更高。

`rebalance()` 和 `rescale()` 的根本区别在于任务之间连接的机制不同。`rebalance()` 将会针对所有发送者任务和所有接收者任务之间建立通信通道，而 `rescale()` 仅仅针对每一个任务和下游算子的一部分子并行任务之间建立通信通道。`rescale` 的示意图为图 5-7。

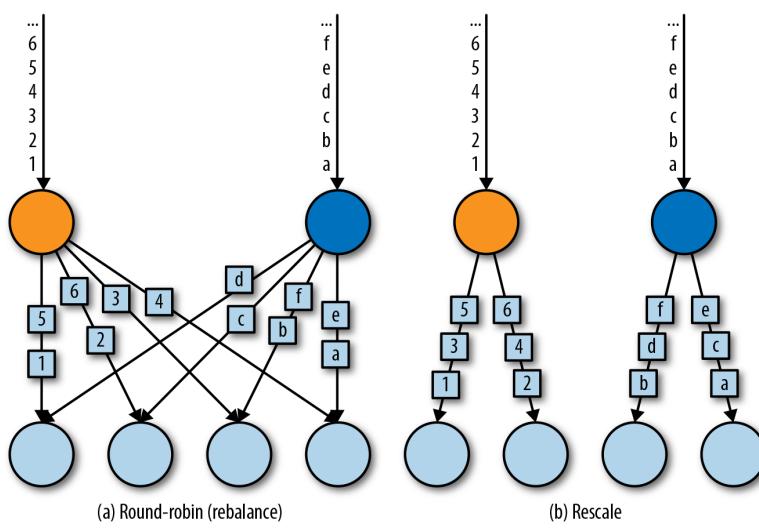


图 4.1: rebalance 和 rescale

Broadcast

`broadcast()` 方法将输入流的所有数据复制并发送到下游算子的所有并行任务中去。

Global

`global()` 方法将所有的输入流数据都发送到下游算子的第一个并行任务中去。这个操作需要很谨慎，因为将所有数据发送到同一个 task，将会对应用程序造成很大的压力。

Custom

当 Flink 提供的分区策略都不适用时，我们可以使用 `partitionCustom()` 方法来自定义分区策略。这个方法接收一个 `Partitioner` 对象，这个对象需要实现分区逻辑以及定义针对流的哪一个字段或者 key 来进行分区。

4.5 设置并行度

Flink 应用程序在一个像集群这样的分布式环境中并行执行。当一个数据流程序提交到作业管理器执行时，系统将会创建一个数据流图，然后准备执行需要的操作符。每一个操作符将会并行化到一个或者多个任务中去。每个算子的并行任务都会处理这个算子的输入流中的一份子集。一个算子并行任务的个数叫做算子的并行度。它决定了算子执行的并行化程度，以及这个算子能处理多少数据量。

算子的并行度可以在执行环境这个层级来控制，也可以针对每个不同的算子设置不同的并行度。默认情况下，应用程序中所有算子的并行度都将设置为执行环境的并行度。执行环境的并行度（也就是所有算子的默认并行度）将在程序开始运行时自动初始化。如果应用程序在本地执行环境中运行，并行度将被设置为 CPU 的核数。当我们把应用程序提交到一个处于运行中的 Flink 集群时，执行环境的并行度将被设置为集群默认的并行度，除非我们在客户端提交应用程序时显式的设置好并行度。

通常情况下，将算子的并行度定义为和执行环境并行度相关的数值会是个好主意。这允许我们通过在客户端调整应用程序的并行度就可以将程序水平扩展了。我们可以使用以下代码来访问执行环境的默认并行度。

我们还可以重写执行环境的默认并行度，但这样的话我们将再也不能通过客户端来控制应用程序的并行度了。

算子默认的并行度也可以通过重写来明确指定。在下面的例子里面，数据源的操作符将会按照环境默认的并行度来并行执行，`map` 操作符的并行度将会是默认并行度的 2 倍，`sink` 操作符的并行度为 2。

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
int defaultP = env.getParallelism();
env
    .addSource(new CustomSource())
    .map(new MyMapper())
    .setParallelism(defaultP * 2)
    .print()
    .setParallelism(2);
```

当我们通过客户端将应用程序的并行度设置为 16 并提交执行时，`source` 操作符的并行度为 16，`mapper` 并行度为 32，`sink` 并行度为 2。如果我们在本地环境运行应用程序的

话，例如在 IDE 中运行，机器是 8 核，那么 source 任务将会并行执行在 8 个任务上面，mapper 运行在 16 个任务上面，sink 运行在 2 个任务上面。

 **笔记** 并行度是动态概念，任务槽数量是静态概念。并行度 \leq 任务槽数量。一个任务槽最多运行一个并行度。

4.6 Flink 的类型系统

Flink 程序所处理的流中的事件一般是对象类型。操作符接收对象输出对象。所以 Flink 的内部机制需要能够处理事件的类型。在网络中传输数据，或者将数据写入到状态后端、检查点和保存点中，都需要我们对数据进行序列化和反序列化。为了高效的进行此类操作，Flink 需要流中事件类型的详细信息。Flink 使用了 Type Information 的概念来表达数据类型，这样就能针对不同的数据类型产生特定的序列化器，反序列化器和比较操作符。

Flink 也能够通过分析输入数据和输出数据来自动获取数据的类型信息以及序列化器和反序列化器。尽管如此，在一些特定的情况下，例如匿名函数或者使用泛型的情况下，我们需要明确的提供数据的类型信息，来提高我们程序的性能。

在这一节中，我们将讨论 Flink 支持的类型，以及如何为数据类型创建相应的类型信息，还有就是在 Flink 无法推断函数返回类型的情况下，如何帮助 Flink 的类型系统去做类型推断。

4.6.1 支持的数据类型

Flink 支持 Java 提供的所有普通数据类型。最常用的数据类型可以做以下分类：

- 原始数据类型
- Flink 专门为 Java 实现的元组
- POJO 类型

接下来让我们一探究竟。

原始数据类型

Java 提供的所有原始数据类型都支持，例如 Integer, String, Double 等等。下面举一个例子：

```
DataStream<Long> numbers = env.fromElements(1L, 2L, 3L, 4L);
numbers.map(n -> n + 1);
```

元组

元组是一种组合数据类型，由固定数量的元素组成。

Flink 为 Java 的 Tuple 提供了高效的实现。Flink 实现的 Java Tuple 最多可以有 25 个元素，根据元素数量的不同，Tuple 都被实现成了不同的类：Tuple1, Tuple2, 一直到 Tuple25。Tuple 类是强类型。

```
DataStream<Tuple2<String, Integer>> persons = env
    .fromElements(
        Tuple2.of("Adam", 17),
        Tuple2.of("Sarah", 23)
    );

persons.filter(p -> p.f1 > 18);
```

Tuple 的元素可以通过它们的 public 属性访问——f0, f1, f2 等等。或者使用 getField(int pos) 方法来访问，元素下标从 0 开始：

```
import org.apache.flink.api.java.tuple.Tuple2;

Tuple2<String, Integer> personTuple = Tuple2.of("Alex", 42);
Integer age = personTuple.getField(1); // age = 42
```

Flink 为 Java 实现的 Tuple 是可变数据结构，所以 Tuple 中的元素可以重新进行赋值。重复利用 Java 的 Tuple 可以减轻垃圾收集的压力。举个例子：

```
personTuple.f1 = 42; // set the 2nd field to 42
personTuple.setField(43, 1); // set the 2nd field to 43
```

POJO

POJO 类的定义：

- 公有类
- 无参数的公有构造器
- 所有的字段都是公有的，可以通过 getters 和 setters 访问。
- 所有字段的数据类型都必须是 Flink 支持的数据类型。

举个例子：

```
public class Person {
    public String name;
    public int age;

    public Person() {}

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

DataStream<Person> persons = env.fromElements(
    new Person("Alex", 42),
```

```

    new Person("Wendy", 23)
);

```

4.6.2 为数据类型创建类型信息

Flink 类型系统的核心类是 TypeInformation。它为系统在产生序列化器和比较操作符时，提供了必要的类型信息。例如，如果我们想使用某个 key 来做联结查询或者分组操作，TypeInformation 可以让 Flink 做更严格的类型检查。

Flink 针对 Java 提供了类来产生类型信息。在 Java 中，类是

```
org.apache.flink.api.common.typeinfo.Types
```

举个例子：

```

TypeInformation<Integer> intType = Types.INT;

TypeInformation<Tuple2<Long, String>> tupleType = Types.TUPLE(Types.LONG,
    Types.STRING);

TypeInformation<Person> personType = Types.POJO(Person.class);

```

4.6.3 Java Lambda 表达式

Java 8 引入了几种新的语言特性，旨在实现更快、更清晰的编码。作为最重要的特性，即所谓的“Lambda 表达式”，它开启了函数式编程的大门。Lambda 表达式允许以简捷的方式实现和传递函数，而无需声明额外的（匿名）类。



笔记 Flink 支持对 Java API 的所有算子使用 Lambda 表达式，但是，当 Lambda 表达式使用 Java 泛型时，你需要显式声明类型信息。

下例演示了如何实现一个简单的行内 map() 函数，它使用 Lambda 表达式计算输入的平方。不需要声明 map() 函数的输入 i 和输出参数的数据类型，因为 Java 编译器会对它们做出推断。

```

env.fromElements(1, 2, 3)
// 返回 i 的平方
.map(i -> i*i)
.print();

```

由于 OUT 是 Integer 而不是泛型，Flink 可以由方法签名 OUT map(IN value) 的实现中自动提取出结果的类型信息。

不幸的是，flatMap() 这样的函数，它的签名 void flatMap(IN value, Collector<OUT> out) 被 Java 编译器编译为 void flatMap(IN value, Collector out)。这样 Flink 就无法自动推断输出的类型信息了。

Flink 很可能抛出类似如下的异常：

```
org.apache.flink.api.common.functions.InvalidTypesException: The generic type
↳ parameters of 'Collector' are missing.
In many cases lambda methods don't provide enough information for automatic type
↳ extraction when Java generics are involved.
An easy workaround is to use an (anonymous) class instead that implements the
↳ 'org.apache.flink.api.common.functions.FlatMapFunction' interface.
Otherwise the type has to be specified explicitly using type information.
```

在这种情况下，需要显式指定类型信息，否则输出将被视为 Object 类型，这会导致低效的序列化。

```
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.util.Collector;

DataSet<Integer> input = env.fromElements(1, 2, 3);

// 必须声明 collector 类型
input.flatMap((Integer number, Collector<String> out) -> {
    StringBuilder builder = new StringBuilder();
    for(int i = 0; i < number; i++) {
        builder.append("a");
        out.collect(builder.toString());
    }
})
// 显式提供类型信息
.returns(Types.STRING)
// 打印 "a", "a", "aa", "a", "aa", "aaa"
.print();
```

当使用 map() 函数返回泛型类型的时候也会发生类似的问题。下例中的方法签名 Tuple2<Integer, Integer> map(Integer value) 被擦除为 Tuple2 map(Integer value)。

```
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.tuple.Tuple2;

env.fromElements(1, 2, 3)
.map(i -> Tuple2.of(i, i))    // 没有关于 Tuple2 字段的信息
.print();
```

一般来说，这些问题可以通过多种方式解决：

```

import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.tuple.Tuple2;

// 使用显式的 ".returns(...)"
env.fromElements(1, 2, 3)
.map(i -> Tuple2.of(i, i))
.returns(Types.TUPLE(Types.INT, Types.INT))
.print();

// 使用类来替代
env.fromElements(1, 2, 3)
.map(new MyTuple2Mapper())
.print();

public static class MyTuple2Mapper extends MapFunction<Integer, Tuple2<Integer,
    <@Integer>> {
    @Override
    public Tuple2<Integer, Integer> map(Integer i) {
        return Tuple2.of(i, i);
    }
}

// 使用匿名类来替代
env.fromElements(1, 2, 3)
.map(new MapFunction<Integer, Tuple2<Integer, Integer>> {
    @Override
    public Tuple2<Integer, Integer> map(Integer i) {
        return Tuple2.of(i, i);
    }
})
.print();

// 或者在这个例子中用 Tuple 的子类来替代
env.fromElements(1, 2, 3)
.map(i -> new DoubleTuple(i, i))
.print();

public static class DoubleTuple extends Tuple2<Integer, Integer> {
    public DoubleTuple(int f0, int f1) {
        this.f0 = f0;
        this.f1 = f1;
    }
}

```

4.7 富函数

我们经常会有这样的需求：在函数处理数据之前，需要做一些初始化的工作；或者需要在处理数据时可以获得函数执行上下文的一些信息；以及在处理完数据时做一些清理工作。而 DataStream API 就提供了这样的机制。

DataStream API 提供的所有转换操作函数，都拥有它们的富版本，并且我们在使用常规函数或者匿名函数的地方来使用富函数。例如下面就是富函数的一些例子，可以看出，只需要在常规函数的前面加上 Rich 前缀就是富函数了。

- RichMapFunction
- RichFlatMapFunction
- RichFilterFunction
- ...

当我们使用富函数时，我们可以实现两个额外的方法：

- 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。open() 函数通常用来做一些只需要做一次即可的初始化工作。
- close() 方法是生命周期中的最后一个调用的方法，通常用来做一些清理工作。

另外，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，当前子任务的索引，当前子任务的名字。同时还它还包含了访问分区状态的方法。下面看一个例子：

```
public static class MyFlatMap extends RichFlatMapFunction<Integer, Tuple2<Integer,
    → Integer>> {
    private int subTaskIndex = 0;

    @Override
    public void open(Configuration configuration) {
        int subTaskIndex = getRuntimeContext().getIndexOfThisSubtask();
        // 做一些初始化工作
        // 例如建立一个和 HDFS 的连接
        System.out.println("enter life cycle");
    }

    @Override
    public void flatMap(Integer in, Collector<Tuple2<Integer, Integer>> out) {
        if (in % 2 == subTaskIndex) {
            out.collect((subTaskIndex, in));
        }
    }

    @Override
    public void close() {
        // 清理工作，断开和 HDFS 的连接。
        System.out.println("exit life cycle");
    }
}
```

```

    }
}
```

4.8 写入下游设备

4.8.1 写入文件

```

import org.apache.flink.api.common.serialization.SimpleStringEncoder;
import org.apache.flink.core.fs.Path;
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink;
import org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.DefaultRo

DataStream<String> input = ...;

final StreamingFileSink<String> sink = StreamingFileSink
    .forRowFormat(new Path(outputPath), new SimpleStringEncoder<String>("UTF-8"))
    .withRollingPolicy(
        DefaultRollingPolicy.builder()
            .withRolloverInterval(TimeUnit.MINUTES.toMillis(15))
            .withInactivityInterval(TimeUnit.MINUTES.toMillis(5))
            .withMaxPartSize(1024 * 1024 * 1024)
        .build())
    .build();

input.addSink(sink);
```

这个例子创建了一个简单的 Sink，将记录分配给默认的一小时时间桶。它还指定了一个滚动策略，该策略在以下三种情况下滚动处于 In-progress 状态的部分文件 (part file):

- 它至少包含 15 分钟的数据
- 最近 5 分钟没有收到新的记录
- 文件大小达到 1GB (写入最后一条记录后)

4.8.2 写入 Kafka

依赖如同读取 Kafka 一节中引入的一样

```

Properties properties = new Properties();
properties.put("bootstrap.servers", "localhost:9092");
```

```
DataStreamSource<String> stream = env.addSource(new UserBehaviorSource());

stream
    .addSink(new FlinkKafkaProducer<String>(
        "userbehavior",
        new SimpleStringSchema(),
        properties
    ));

```

4.8.3 写入 Redis

导入依赖

```
<dependency>
    <groupId>org.apache.bahir</groupId>
    <artifactId>flink-connector-redis_2.11</artifactId>
    <version>1.0</version>
</dependency>

FlinkJedisPoolConfig conf = new FlinkJedisPoolConfig.Builder().setHost("localhost").build();

env
    .addSource(new UserBehaviorSource())
    .addSink(new RedisSink<UserBehavior>(conf, new MyRedisSink()));

public static class MyRedisSink implements RedisMapper<UserBehavior> {
    @Override
    public String getKeyFromData(UserBehavior r) {
        return r.itemId;
    }

    @Override
    public String getValueFromData(UserBehavior r) {
        return r.categoryId;
    }

    @Override
    public RedisCommandDescription getCommandDescription() {
        return new RedisCommandDescription(RedisCommand.HSET, "UserBehavior");
    }
}
```

```

    }
}

```

4.8.4 写入 ElasticSearch

导入依赖，这里写入的 ES 版本是 ES 7

```

<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-elasticsearch7_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
</dependency>

```

代码如下

```

DataStreamSource<UserBehavior> stream = env.addSource(new UserBehaviorSource());

ArrayList<HttpHost> httpHosts = new ArrayList<>();
httpHosts.add(new HttpHost("127.0.0.1", 9200, "http"));

ElasticsearchSink.Builder<UserBehavior> esBuilder = new ElasticsearchSink.Builder<>(
    httpHosts,
    new ElasticsearchSinkFunction<UserBehavior>() {
        @Override
        public void process(UserBehavior r, RuntimeContext runtimeContext, RequestIndexer
            IndexRequest indexRequest = Requests
                .indexRequest()
                .index("UserBehavior")
                .source(data);

            requestIndexer.add(indexRequest);
        }
    }
);

esBuilder.setBulkFlushMaxActions(1);

stream.addSink(esBuilder.build());

```

4.8.5 写入 MySQL

这里 MySQL 版本是 MySQL 8

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.21</version>
</dependency>
```

代码如下，这里是幂等性写入的

```
env
    .addSource(new UserBehaviorSource())
    .addSink(new MyJDBC());

public static class MyJDBC extends RichSinkFunction<UserBehavior> {
    private Connection conn;
    private PreparedStatement insertStmt;
    private PreparedStatement updateStmt;
    @Override
    public void open(Configuration parameters) throws Exception {
        super.open(parameters);
        conn = DriverManager.getConnection(
            "jdbc:mysql://localhost:3306/userbehavior",
            "zuoyuan",
            "zuoyuan"
        );
        insertStmt = conn.prepareStatement("INSERT INTO userbehavior (itemId, categoryId");
        updateStmt = conn.prepareStatement("UPDATE userbehavior SET categoryId = ? WHERE
    }

    @Override
    public void invoke(UserBehavior r, Context context) throws Exception {
        updateStmt.setString(1, r.categoryId);
        updateStmt.setString(2, r.itemId);
        updateStmt.execute();

        if (updateStmt.getUpdateCount() == 0) {
            insertStmt.setString(1, r.itemId);
```

```

        insertStmt.setString(2, r.categoryId);
        insertStmt.execute();
    }

}

@Override
public void close() throws Exception {
    super.close();
    insertStmt.close();
    updateStmt.close();
    conn.close();
}
}

```

4.8.6 自定义 SINK

写入 socket

```

DataStreamSource<UserBehavior> stream = env.addSource(new UserBehaviorSource());

stream
    .addSink(new SimpleSocketSink("localhost", 9999))
    .setParallelism(1);

public static class SimpleSocketSink extends RichSinkFunction<UserBehavior> {
    private Socket socket;
    private PrintStream writer;
    private String host;
    private Integer port;

    public SimpleSocketSink(String host, Integer port) {
        this.host = host;
        this.port = port;
    }

    @Override
    public void open(Configuration parameters) throws Exception {
        super.open(parameters);
        socket = new Socket(InetAddress.getByName(host), port);
        writer = new PrintStream(socket.getOutputStream());
    }
}

```

```
}

@Override
public void invoke(UserBehavior value, Context context) throws Exception {
    writer.println(value);
    writer.flush();
}

@Override
public void close() throws Exception {
    super.close();
    writer.close();
    socket.close();
}

}
```

第五章 基于时间和窗口的操作符

在本章，我们将要学习 DataStream API 中处理时间和基于时间的操作符，例如窗口操作符。

首先，我们会学习如何定义时间属性，时间戳和水位线。然后我们将会学习底层操作 process function，它可以让访问时间戳和水位线，以及注册定时器事件。接下来，我们将会使用 Flink 的 window API，它提供了通常使用的各种窗口类型的内置实现。我们将会学到如何进行用户自定义窗口操作符，以及窗口的核心功能：assigners（分配器）、triggers（触发器）和 evictors（清理器）。最后，我们将讨论如何基于时间来做流的联结查询，以及处理迟到事件的策略。

5.1 水位线的设定

5.1.1 水位线的一般设置方案

如果使用事件时间，那么流中的事件必须包含这个事件真正发生的时间。使用了事件时间的流必须携带水位线。

时间戳和水位线的单位是毫秒，记时从 1970-01-01T00:00:00Z 开始。到达某个操作符的水位线就会告知这个操作符：小于等于水位线中携带的时间戳的事件都已经到达这个操作符了。时间戳和水位线可以由 SourceFunction 产生，或者由用户自定义的时间戳分配器和水位线产生器来生成。

Flink 暴露了 TimestampAssigner 接口供我们实现，使我们可以自定义如何从事件数据中抽取时间戳。一般来说，时间戳分配器需要在 source 操作符后马上进行调用。

 **笔记** 因为时间戳分配器看到的元素的顺序应该和 source 操作符产生数据的顺序是一样的，否则就乱了。这就是为什么我们经常将 source 操作符的并行度设置为 1 的原因。

也就是说，任何分区操作都会将元素的顺序打乱，例如：并行度改变，keyBy() 操作等等。

所以最佳实践是：在尽量接近数据源 source 操作符的地方分配时间戳和产生水位线，甚至最好在 SourceFunction 中分配时间戳和产生水位线。当然在分配时间戳和产生水位线之前可以对流进行 map 和 filter 操作是没问题的，也就是说必须是窄依赖。

以下这种写法是可以的。

```
DataStream<T> stream = env
    .addSource(...)
    .map(...)
    .filter(...)
    .assignTimestampsAndWatermarks(...)
```

我们来看一个例子，这个例子的最大延迟时间是 5 秒钟。

```

env
    .addSource(new UserBehaviorSource())
    .assignTimestampsAndWatermarks(
        WatermarkStrategy
            .

```

如果我们已经知道到来的事件的时间戳是升序的，也就是最大延迟时间为 0，那么我们可以使用如下写法：

```

env
    .addSource(new UserBehaviorSource())
    .assignTimestampsAndWatermarks(
        WatermarkStrategy
            .<UserBehavior>forMonotonousTimestamps()
            .withTimestampAssigner(new SerializableTimestampAssigner<UserBehavior>() {
                @Override
                public long extractTimestamp(UserBehavior element, long
                    recordTimestamp) {
                    return element.timestamp;
                }
            })
        )
    .keyBy(r -> r.itemId)
    .window(TumblingEventTimeWindows.of(Time.seconds(5)))
    .aggregate(...)

```

5.1.2 自定义水位线的产生逻辑

产生水位线的接口

```

@Public
public interface WatermarkGenerator<T> {
    /**

```

```

    * 每来一个事件都会调用，允许水位线产生器记忆和检查事件的时间戳。
    * 允许水位线产生器基于事件本身发射水位线。
    */
    void onEvent(T event, long eventTimestamp, WatermarkOutput output);

    /**
     * 周期性的调用（默认 200ms 调用一次），可能会产生新的水位线，也可能不会。
     *
     * 调用周期通过 ExecutionConfig#getAutoWatermarkInterval() 方法来配置。
     */
    void onPeriodicEmit(WatermarkOutput output);
}

```

周期性产生水位线

```

public static class CustomWatermarkStrategy implements
    WatermarkStrategy<UserBehavior>() {
    @Override
    public TimestampAssigner<UserBehavior>
        createTimestampAssigner(TimestampAssignerSupplier.Context context) {
        return new SerializableTimestampAssigner<UserBehavior>() {
            @Override
            public long extractTimestamp(UserBehavior element, long recordTimestamp) {
                return element.timestamp;
            }
        };
    }

    @Override
    public WatermarkGenerator<UserBehavior>
        createWatermarkGenerator(WatermarkGeneratorSupplier.Context context) {
        return new BoundedOutOfOrdernessGenerator();
    }
}

public static class BoundedOutOfOrdernessGenerator implements
    WatermarkGenerator<UserBehavior> {

    private final long maxOutOfOrderness = 3500; // 最大延迟时间是 3.5s

    private long currentMaxTimestamp;

    @Override
    public void onEvent(UserBehavior r, long eventTimestamp, WatermarkOutput output) {
        currentMaxTimestamp = Math.max(currentMaxTimestamp, r.timestamp);
    }
}

```

```

@Override
public void onPeriodicEmit(WatermarkOutput output) {
    // 产生水位线的公式：观察到的最大时间戳 - 最大延迟时间 - 1ms
    output.emitWatermark(new Watermark(currentMaxTimestamp - maxOutOfOrderliness -
        ↪ 1));
}

env
.addSource(new UserBehaviorSource())
.assignTimestampsAndWatermarks(
    new CustomWatermarkStrategy());
}

```

不规则水位线的产生

```

public class PunctuatedAssigner implements WatermarkGenerator<UserBehavior> {

    @Override
    public void onEvent(UserBehavior r, long eventTimestamp, WatermarkOutput output) {
        if (r.itemId.equals("1")) {
            output.emitWatermark(new Watermark(r.timestamp - 1));
        }
    }

    @Override
    public void onPeriodicEmit(WatermarkOutput output) {
        // 不需要做任何事情，因为我们在 onEvent 方法中发射了水位线
    }
}

```

5.1.3 水位线总结

定义 5.1. 水位线定义

水位线是一种逻辑时钟，Flink 认为时间戳小于等于水位线的事件都到了。



定义 5.2. 计算公式

水位线 = 观察到的事件所包含的最大时间戳 - 最大延迟时间 - 1 毫秒 (5.1)



笔记 在事件时间的世界里，时间就是水位线。

现在我们要讨论一下水位线会对我们的程序产生什么样的影响。

水位线用来平衡延迟和计算结果的正确性。水位线告诉我们，在触发计算（例如关闭窗口并触发窗口计算）之前，我们需要等待事件多长时间。基于事件时间的操作符根据水位线来衡量系统的逻辑时间的进度。

完美的水位线永远不会错：时间戳小于水位线的事件不会再出现。在特殊情况下（例如非乱序事件流），最近一次事件的时间戳就可能是完美的水位线。启发式水位线则相反，它只估计时间，因此有可能出错，即迟到的事件（其时间戳小于水位线标记时间）晚于水位线出现。针对启发式水位线，Flink 提供了处理迟到元素的机制。

设定水位线通常需要用到领域知识。举例来说，如果知道事件的迟到时间不会超过 5 秒，就可以将水位线标记时间设为收到的最大时间戳减去 5 秒。另一种做法是，采用一个 Flink 作业监控事件流，学习事件的迟到规律，并以此构建水位线生成模型。

如果最大延迟时间设置的很大，计算出的结果会更精确，但收到计算结果的速度会很慢，同时系统会缓存大量的数据，并对系统造成比较大的压力。如果最大延迟时间设置的很小，那么收到计算结果的速度会很快，但可能收到错误的计算结果。不过 Flink 处理迟到数据的机制可以解决这个问题。上述问题看起来很复杂，但是恰恰符合现实世界的规律：大部分真实的事件流都是乱序的，并且通常无法了解它们的乱序程度（因为理论上不能预见未来）。水位线是唯一让我们直面乱序事件流并保证正确性的机制；否则只能选择忽视事实，假装错误的结果是正确的。

5.2 设置时间属性

如果我们想要在分布式流处理应用程序中定义有关时间的操作，彻底理解时间的语言是非常重要的。当我们指定了一个窗口去收集某 1 分钟内的数据时，这个长度为 1 分钟的桶中，到底应该包含哪些数据？

ProcessingTime

 **笔记** 机器时间在分布式系统中又叫做“墙上时钟”。

当操作符执行时，此操作符看到的时间是操作符所在机器的机器时间。Processing-time window 的触发取决于机器时间，窗口包含的元素也是那个机器时间段内到达的元素。通常情况下，窗口操作符使用 processing time 会导致不确定的结果，因为基于机器时间的窗口中收集的元素取决于元素到达的速度快慢。使用 processing time 会为程序提供极低的延迟，因为无需等待水位线的到达。

 **笔记** 如果要追求极限的低延迟，请使用 ProcessingTime。

EventTime

当操作符执行时，操作符看的当前时间是由流中元素所携带的信息决定的。流中的每一个元素都必须包含时间戳信息。而系统的逻辑时钟由水位线（Watermark）定义。我们之前学习过，时间戳要么在事件进入流处理程序之前已经存在，要么就需要在程序的数据源（source）处进行分配。当水位线宣布特定时间段的数据都已经到达，事件时间窗口将会被触发计算。即使数据到达的顺序是乱序的，事件时间窗口的计算结果也将是确定性的。窗口的计算结果并不取决于元素到达的快与慢。

当水位线超过事件时间窗口的结束时间时，窗口将会闭合，不再接收数据，并触发计算。

IngestionTime

当事件进入 source 操作符时，source 操作符所在机器的机器时间，就是此事件的“摄入时间”（IngestionTime），并同时产生水位线。IngestionTime 相当于 EventTime 和 ProcessingTime 的混合体。一个事件的 IngestionTime 其实就是它进入流处理器中的时间。

 **笔记** IngestionTime 没什么价值，既有 EventTime 的执行效率（比较低），又没有 EventTime 计算结果的准确性。

Flink 1.12 通过指定不同类型的窗口来使用不同的时间属性。

5.2.1 处理时间设定

我们写 Flink 程序一般遵循以下步骤：

1. 读取数据源
2. 分流
3. 开窗
4. 聚合
5. 输出结果

滚动窗口

```
.keyBy(...)
.window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
.aggregate(...)
```

滑动窗口

```
.keyBy(...)
.window(SlidingProcessingTimeWindows.of(Time.seconds(10), Time.seconds(5)))
.aggregate(...)
```

会话窗口

```
.keyBy(...)
.window(ProcessingTimeSessionWindows.withGap(Time.seconds(10)))
.aggregate(...)
```

5.2.2 事件时间设定和水位线的产生

滚动窗口

```
.keyBy(...)
.window(TumblingEventTimeWindows.of(Time.seconds(5)))
.aggregate(...)
```

滑动窗口

```
.keyBy(...)
.window(SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(5)))
.aggregate(...)
```

会话窗口

```
.keyBy(...)
.window(EventTimeSessionWindows.withGap(Time.seconds(10)))
.aggregate(...)
```

5.2.3 窗口聚合函数的使用

窗口聚合函数定义了窗口中数据的计算逻辑。有两种计算逻辑：

- 增量聚合函数：当一个事件被添加到窗口时，触发函数计算，并且更新 window 的状态（单个值）。最终聚合的结果将作为输出。ReduceFunction 和 AggregateFunction 是增量聚合函数。
- 全窗口函数：这个函数将会收集窗口中所有的元素，可以做一些复杂计算。ProcessWindowFunction 是全窗口聚合函数。

增量聚合函数

- 每条数据到来就进行计算，只保存一个简单状态（累加器）
- ReduceFunction, AggregateFunction
- 当窗口闭合的时候，增量聚合完成
- 处理时间：当机器时间超过窗口结束时间的时候，窗口闭合
- 事件时间：当水位线超过窗口结束时间的时候，窗口闭合
- 来一条数据计算一次

REDUCE

计算每个用户的 PV 数据

```
stream
    .map(new MapFunction<UserBehavior, Tuple2<String, Long>>() {
        @Override
        public Tuple2<String, Long> map(UserBehavior value) throws Exception {
            return Tuple2.of(value.userId, 1L);
        }
    })
```

```

.keyBy(r -> r.f0)
.window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
.reduce(new ReduceFunction<Tuple2<String, Long>>() {
    @Override
    public Tuple2<String, Long> reduce(Tuple2<String, Long> value1, Tuple2<String,
        Long> value2) throws Exception {
        return Tuple2.of(value1.f0, value1.f1 + value2.f1);
    }
});

```

AggregateFunction

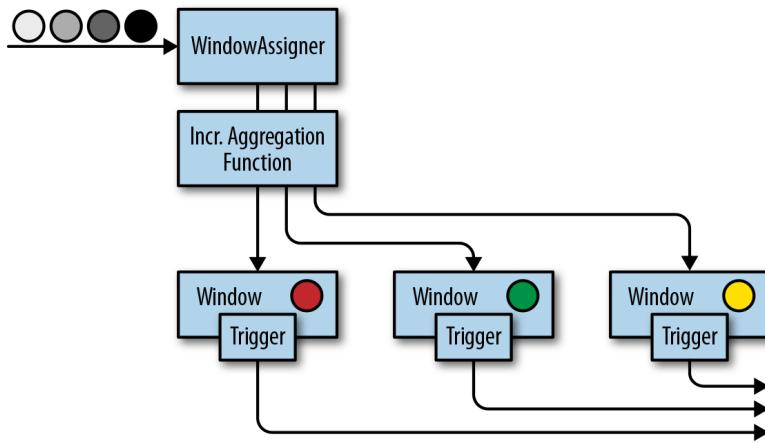


图 5.1: 增量聚合

先来看接口定义

```

public interface AggregateFunction<IN, ACC, OUT> extends Function, Serializable {

    // 创建新累加器
    ACC createAccumulator();

    // 累加操作的逻辑: 将每一条输入元素累加到累加器上, 并返回累加器
    ACC add(IN value, ACC accumulator);

    // 返回累加结果
    OUT getResult(ACC accumulator);

    // 将两个累加器聚合, 只在事件时间会话窗口才会用到
    ACC merge(ACC a, ACC b);
}

```

IN 是输入元素的类型，ACC 是累加器的类型，OUT 是输出元素的类型。

例子：计算每个用户的平均访问次数

```

stream
    .keyBy(
        new KeySelector<UserBehavior, String>() {
            @Override
            public String getKey(UserBehavior value) throws Exception {
                return "key";
            }
        }
    )
    .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
    .aggregate(new AvgPv())
    .print();

public static class AvgPv implements AggregateFunction<UserBehavior,
    Tuple2<HashSet<String>, Long>, Double> {
    @Override
    public Tuple2<HashSet<String>, Long> createAccumulator() {
        return Tuple3.of(new HashSet<String>(), 0L);
    }

    @Override
    public Tuple2<HashSet<String>, Long> add(UserBehavior value,
        Tuple2<HashSet<String>, Long> accumulator) {
        return Tuple2.of(accumulator.f0.add(value.userId), accumulator.f1 + 1L);
    }

    @Override
    public Double getResult(Tuple2<HashSet<String>, Long> accumulator) {
        return accumulator.f0.size() / accumulator.f1;
    }

    @Override
    public Tuple2<HashSet<String>, Long> merge(Tuple2<HashSet<String>, Long> a,
        Tuple2<HashSet<String>, Long> b) {
        return null;
    }
}

```

全窗口聚合函数

一些业务场景，我们需要收集窗口内所有的数据进行计算，例如计算窗口数据的中位数，或者计算窗口数据中出现频率最高的值。这样的需求，使用 ReduceFunction 和 AggregateFunction 就无法实现了。这个时候就需要全窗口聚合函数了。

先来看接口定义

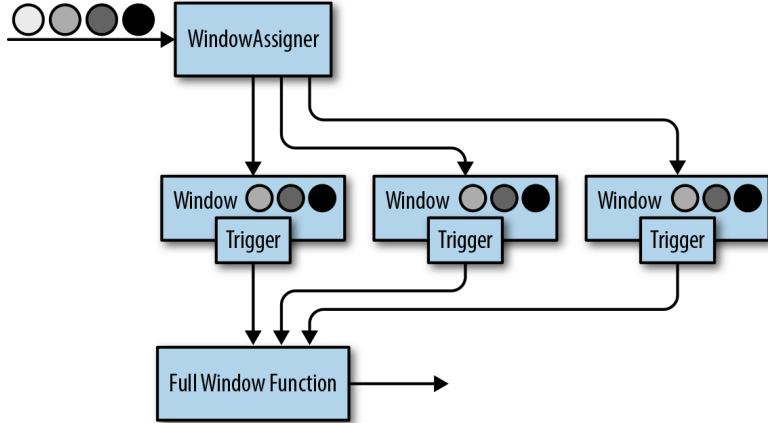


图 5.2: 全窗口聚合

```

public interface WindowFunction<IN, OUT, KEY, W extends Window> extends Function,
→ Serializable {
    void apply(KEY key, W window, Iterable<IN> input, Collector<OUT> out) throws
    → Exception;
}
  
```

apply() 方法接受的参数为: window 的 key, input 迭代器包含窗口的所有元素, out 用于输出结果流, window 包含了窗口的信息, 比如窗口的开始时间和结束时间。

例子: 计算每个窗口的 UV 数据

```

env
    .addSource(new UserBehaviorSource())
    .assignTimestampsAndWatermarks(
        WatermarkStrategy.<Example1.UserBehavior>forMonotonousTimestamps()
            .withTimestampAssignor(new
                → SerializableTimestampAssignor<Example1.UserBehavior>() {
                    @Override
                    public long extractTimestamp(Example1.UserBehavior element, long
                        → recordTimestamp) {
                        return element.timestamp;
                    }
                })
            )
        .keyBy(new KeySelector<Example1.UserBehavior, String>() {
            @Override
            public String getKey(Example1.UserBehavior value) throws Exception {
                return "key";
            }
        })
        .window(TumblingEventTimeWindows.of(Time.hours(1)))
        .apply(new WindowFunction<Example1.UserBehavior, String, String, TimeWindow>() {
            @Override
  
```

```

public void apply(String s, TimeWindow timeWindow,
    Iterable<Example1.UserBehavior> iterable, Collector<String> collector)
    throws Exception {
    HashSet<String> set = new HashSet<>();
    for (Example1.UserBehavior e : iterable) {
        set.add(e.userId);
    }
    collector.collect(set.size() + "");
}
.print();

```

增量聚合和全窗口聚合结合使用

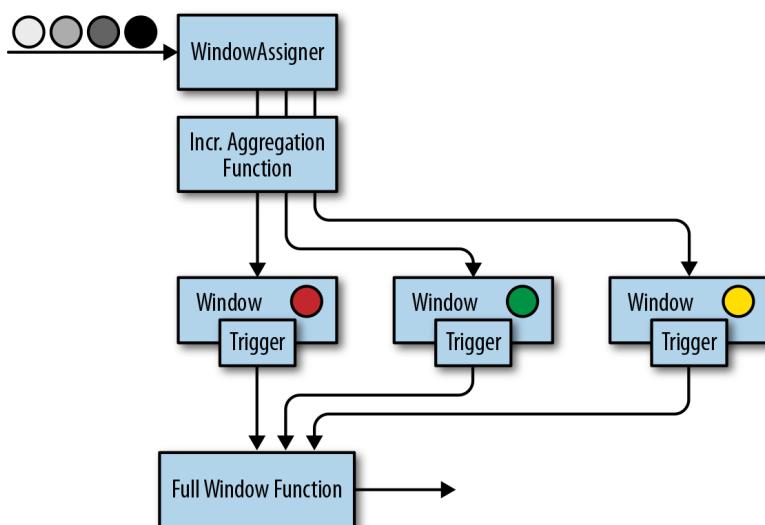


图 5.3: 增量聚合和全窗口聚合结合使用

我们还可以将 `ReduceFunction/AggregateFunction` 和 `WindowFunction` 结合起来使用。 `ReduceFunction/AggregateFunction` 做增量聚合， `WindowFunction` 提供更多的对数据流的访问权限。如果只使用 `WindowFunction`(底层的实现为将事件都保存在 `ListState` 中)，将会非常占用空间。分配到某个窗口的元素将被提前聚合，而当窗口的 trigger 触发时，也就是窗口收集完数据关闭时，将会把聚合结果发送到 `WindowFunction` 中，这时 `Iterable` 参数将会只有一个值，就是前面聚合的值。

我们实现一个需求：求每个商品在每个窗口的 pv 数据

```

SingleOutputStreamOperator<ItemViewCount> ivcStream = pvStream
    .keyBy(r -> r.itemId)
    .window(SlidingEventTimeWindows.of(Time.hours(1), Time.minutes(5)))
    .aggregate(new CountAgg(), new WindowResult());

public static class CountAgg implements AggregateFunction<UserBehavior, Long, Long> {
    @Override

```

```

public Long createAccumulator() {
    return 0L;
}

@Override
public Long add(UserBehavior value, Long accumulator) {
    return accumulator + 1L;
}

@Override
public Long getResult(Long accumulator) {
    return accumulator;
}

@Override
public Long merge(Long a, Long b) {
    return null;
}

public static class WindowResult implements WindowFunction<Long, ItemViewCount, String,
    TimeWindow> {

    @Override
    public void apply(String s, TimeWindow timeWindow, Iterable<Long> iterable,
        Collector<ItemViewCount> collector) throws Exception {
        collector.collect(new ItemViewCount(s, iterable.iterator().next(),
            timeWindow.getStart(), timeWindow.getEnd()));
    }
}

public static class ItemViewCount {

    public String itemId;
    public Long count;
    public Long windowStart;
    public Long windowEnd;

    public ItemViewCount() {
    }

    public ItemViewCount(String itemId, Long count, Long windowStart, Long windowEnd) {
        this.itemId = itemId;
        this.count = count;
        this.windowStart = windowStart;
        this.windowEnd = windowEnd;
    }
}

```

```
@Override
public String toString() {
    return "ItemViewCount{" +
        "itemId='" + itemId + '\'' +
        ", count=" + count +
        ", windowStart=" + new Timestamp(windowStart) +
        ", windowEnd=" + new Timestamp(windowEnd) +
        '}';
}
```

第六章 状态变量的初步使用

第七章 处理函数

我们之前学习的转换算子是无法访问事件的时间戳信息和水位线信息的。而这在一些应用场景下，极为重要。例如 MapFunction 这样的 map 转换算子就无法访问时间戳或者当前事件的事件时间。

基于此，DataStream API 提供了一系列的 Low-Level 转换算子。可以访问时间戳、水位线以及注册定时事件。还可以输出特定的一些事件，例如超时事件等。Process Function 用来构建事件驱动的应用以及实现自定义的业务逻辑(使用之前的 window 函数和转换算子无法实现)。例如，Flink-SQL 就是使用 Process Function 实现的。

Flink 提供了 8 个 Process Function：

- ProcessFunction
- KeyedProcessFunction
- CoProcessFunction
- ProcessJoinFunction
- BroadcastProcessFunction
- KeyedBroadcastProcessFunction
- ProcessWindowFunction
- ProcessAllWindowFunction

7.1 KeyedProcessFunction 的使用

我们这里详细介绍一下 KeyedProcessFunction。

KeyedProcessFunction 用来操作 KeyedStream。KeyedProcessFunction 会处理流的每一个元素，输出为 0 个、1 个或者多个元素。所有的 Process Function 都继承自 RichFunction 接口，所以都有 open()、close() 和 getRuntimeContext() 等方法。而 KeyedProcessFunction<KEY, IN, OUT> 还额外提供了两个方法：

- processElement(I value, Context ctx, Collector<O> out): 流中的每一个元素都会调用这个方法，调用结果将会放在 Collector 数据类型中输出。Context 可以访问元素的时间戳，元素的 key，以及 TimerService 时间服务。Context 还可以将结果输出到别的流(side outputs)。
- onTimer(long timestamp, OnTimerContext ctx, Collector<O> out): 本方法是一个回调函数。当之前注册的定时器触发时调用。参数 timestamp 为定时器所设定的触发的时间戳。Collector 为输出结果的集合。OnTimerContext 和 processElement 的 Context 参数一样，提供了上下文的一些信息，例如 firing trigger 的时间信息(事件时间或者处理时间)。

时间服务和定时器

Context 和 OnTimerContext 所持有的 TimerService 对象拥有以下方法：

- currentProcessingTime() 返回当前处理时间
- currentWatermark() 返回当前水位线的时间戳
- registerProcessingTimeTimer(long time) 会注册当前 key 的 processing time 的 timer。当 processing time 到达定时时间时，触发 timer。
- registerEventTimeTimer(long time) 会注册当前 key 的 event time timer。当水位线大于等于定时器注册的时间时，触发定时器执行回调函数。
- deleteProcessingTimeTimer(long time) 删除之前注册处理时间定时器。如果没有这个时间戳的定时器，则不执行。
- deleteEventTimeTimer(long time) 删除之前注册的事件时间定时器，如果没有此时间戳的定时器，则不执行。

当定时器 timer 触发时，执行回调函数 onTimer()。processElement() 方法和 onTimer() 方法是同步（不是异步）方法，这样可以避免并发访问和操作状态。

针对每一个 key 和 timestamp，只能注册一个定期器。也就是说，每一个 key 可以注册多个定时器，但在每一个时间戳只能注册一个定时器。KeyedProcessFunction 默认将所有定时器的时间戳放在一个优先队列中。在 Flink 做检查点操作时，定时器也会被保存到状态后端中。

举个例子说明 KeyedProcessFunction 如何操作 KeyedStream。

下面的程序展示了如何监控温度传感器的温度值，如果温度值在一秒钟之内 (processing time) 连续上升，报警。

主程序

```
DataStream<String> warings = readings
    .keyBy(r -> r.id)
    .process(new TempIncreaseAlertFunction());
```

处理函数的实现如下

```
public static class TempIncreaseAlertFunction extends KeyedProcessFunction<String,
    SensorReading, String> {

    private ValueState<Double> lastTemp;
    private ValueState<Long> currentTimer;

    @Override
    public void open(Configuration parameters) throws Exception {
        super.open(parameters);
        lastTemp = getRuntimeContext().getState(
            new ValueStateDescriptor<>("last-temp", Types.DOUBLE)
        );
        currentTimer = getRuntimeContext().getState(
            new ValueStateDescriptor<>("current-timer", Types.LONG)
        );
    }
}
```

```

}

@Override
public void processElement(SensorReading r, Context ctx, Collector<String> out)
    throws Exception {
    // 取出上一次的温度
    Double prevTemp = 0.0;
    if (lastTemp.value() != null) {
        prevTemp = lastTemp.value();
    }
    // 将当前温度更新到上一次的温度这个变量中
    lastTemp.update(r.temperature);

    Long curTimerTimestamp = 0L;
    if (currentTimer.value() != null) {
        curTimerTimestamp = currentTimer.value();
    }
    if (prevTemp == 0.0 || r.temperature < prevTemp) {
        // 温度下降或者是第一个温度值，删除定时器
        ctx.timerService().deleteProcessingTimeTimer(curTimerTimestamp);
        // 清空状态变量
        currentTimer.clear();
    } else if (r.temperature > prevTemp && curTimerTimestamp == 0) {
        // 温度上升且我们并没有设置定时器
        long timerTs = ctx.timerService().currentProcessingTime() + 1000L;
        ctx.timerService().registerProcessingTimeTimer(timerTs);
        // 保存定时器时间戳
        currentTimer.update(timerTs);
    }
}

@Override
public void onTimer(long timestamp, OnTimerContext ctx, Collector<String> out)
    throws Exception {
    super.onTimer(timestamp, ctx, out);
    out.collect(" 传感器 id 为: "
        + ctx.getCurrentKey()
        + " 的传感器温度值已经连续 1s 上升了。");
    currentTimer.clear();
}
}

```

将事件发送到旁路输出

大部分的 DataStream API 的算子的输出是单一输出，也就是某种数据类型的流。除了 split 算子，可以将一条流分成多条流，这些流的数据类型也都相同。process function 的 side outputs 功能可以产生多条流，并且这些流的数据类型可以不一样。一个 side output 可

以定义为 OutputTag<X> 对象，X 是输出流的数据类型。process function 可以通过 Context 对象发射一个事件到一个或者多个 side outputs。

```
public class SideOutputExample {

    private static OutputTag<String> output = new OutputTag<String>("side-output") {};

    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        DataStream<SensorReading> stream = env.addSource(new SensorSource());

        SingleOutputStreamOperator<SensorReading> warnings = stream
            .process(new ProcessFunction<SensorReading, SensorReading>() {
                @Override
                public void processElement(SensorReading value, Context ctx,
                    Collector<SensorReading> out) throws Exception {
                    if (value.temperature < 32) {
                        ctx.output(output, " 温度小于 32 度! ");
                    }
                    out.collect(value);
                }
            });
        warnings.print();
        warnings.getSideOutput(output).print();

        env.execute();
    }
}
```

7.2 CoProcessFunction 的使用

对于两条输入流，DataStream API 提供了 CoProcessFunction 这样的 low-level 操作。CoProcessFunction 提供了操作每一个输入流的方法：processElement1() 和 processElement2()。类似于 ProcessFunction，这两种方法都通过 Context 对象来调用。这个 Context 对象可以访问事件数据，定时器时间戳，TimerService，以及 side outputs。CoProcessFunction 也提供了 onTimer() 回调函数。下面的例子展示了如何使用 CoProcessFunction 来合并两条流。

```
public class SensorSwitch {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
```

```

env.setParallelism(1);

KeyedStream<SensorReading, String> stream = env
    .addSource(new SensorSource())
    .keyBy(r -> r.id);

KeyedStream<Tuple2<String, Long>, String> switches = env
    .fromElements(Tuple2.of("sensor_2", 10 * 1000L))
    .keyBy(r -> r.f0);

stream
    .connect(switches)
    .process(new SwitchProcess())
    .print();

env.execute();
}

public static class SwitchProcess extends CoProcessFunction<SensorReading,
    Tuple2<String, Long>, SensorReading> {

    private ValueState<Boolean> forwardingEnabled;

    @Override
    public void open(Configuration parameters) throws Exception {
        super.open(parameters);
        forwardingEnabled = getRuntimeContext().getState(
            new ValueStateDescriptor<>("filterSwitch", Types.BOOLEAN)
        );
    }

    @Override
    public void processElement1(SensorReading value, Context ctx,
        Collector<SensorReading> out) throws Exception {
        if (forwardingEnabled.value() != null && forwardingEnabled.value()) {
            out.collect(value);
        }
    }

    @Override
    public void processElement2(Tuple2<String, Long> value, Context ctx,
        Collector<SensorReading> out) throws Exception {
        forwardingEnabled.update(true);

        ctx.timerService().registerProcessingTimeTimer(ctx.timerService().currentProcessingTime()
            + value.f1);
    }
}

```

```

    }

    @Override
    public void onTimer(long timestamp, OnTimerContext ctx,
        Collector<SensorReading> out) throws Exception {
        super.onTimer(timestamp, ctx, out);
        forwardingEnabled.clear();
    }
}
}

```

7.3 基于时间的双流 Join

数据流操作的另一个常见需求是对两条数据流中的事件进行联结（connect）或 Join。Flink DataStream API 中内置有两个可以根据时间条件对数据流进行 Join 的算子：基于间隔的 Join 和基于窗口的 Join。本节我们会对它们进行介绍。

如果 Flink 内置的 Join 算子无法表达所需的 Join 语义，那么你可以通过 CoProcessFunction、BroadcastProcessFunction 或 KeyedBroadcastProcessFunction 实现自定义的 Join 逻辑。

注意，你要设计的 Join 算子需要具备高效的状态访问模式及有效的状态清理策略。

7.3.1 基于间隔的 Join

基于间隔的 Join 会对两条流中拥有相同键值以及彼此之间时间戳不超过某一指定间隔的事件进行 Join。

下图展示了两条流（A 和 B）上基于间隔的 Join，如果 B 中事件的时间戳相较于 A 中事件的时间戳不早于 1 小时且不晚于 15 分钟，则会将两个事件 Join 起来。Join 间隔具有对称性，因此上面的条件也可以表示为 A 中事件的时间戳相较 B 中事件的时间戳不早于 15 分钟且不晚于 1 小时。

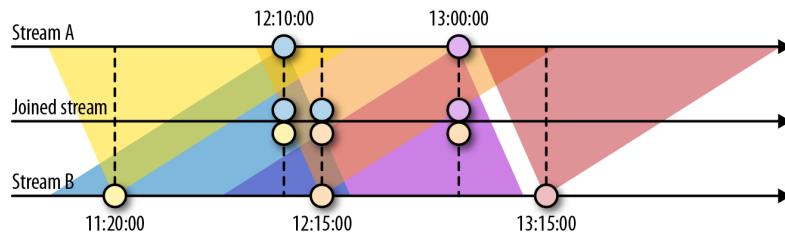


图 7.1: 基于间隔的 Join

基于间隔的 Join 目前只支持事件时间以及 INNER JOIN 语义（无法发出未匹配成功的事件）。下面的例子定义了一个基于间隔的 Join。

```

input1
    .intervalJoin(input2)
    .between(<lower-bound>, <upper-bound>) // 相对于 input1 的上下界
    .process(ProcessJoinFunction) // 处理匹配的事件对

```

Join 成功的事件对会发送给 ProcessJoinFunction。下界和上界分别由负时间间隔和正时间间隔来定义，例如 between(Time.hour(-1), Time.minute(15))。在满足下界值小于上界值的前提下，你可以任意对它们赋值。例如，允许出现 B 中事件的时间戳相较 A 中事件的时间戳早 1~2 小时这样的条件。

基于间隔的 Join 需要同时对双流的记录进行缓冲。对第一个输入而言，所有时间戳大于当前水位线减去间隔上界的数据都会被缓冲起来；对第二个输入而言，所有时间戳大于当前水位线加上间隔下界的数据都会被缓冲起来。注意，两侧边界值都有可能为负。上图中的 Join 需要存储数据流 A 中所有时间戳大于当前水位线减去 15 分钟的记录，以及数据流 B 中所有时间戳大于当前水位线减去 1 小时的记录。不难想象，如果两条流的事件时间不同步，那么 Join 所需的存储就会显著增加，因为水位线总是由“较慢”的那条流来决定。

例子：每个用户的点击 Join 这个用户最近 10 分钟内的浏览

```

public class IntervalJoinExample {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        KeyedStream<Tuple3<String, Long, String>, String> stream1 = env
            .fromElements(
                Tuple3.of("user_1", 10 * 60 * 1000L, "click")
            )
            .assignTimestampsAndWatermarks(
                WatermarkStrategy
                    .<Tuple3<String, Long, String>>forMonotonousTimestamps()
                    .withTimestampAssigner(new
                        SerializableTimestampAssigner<Tuple3<String, Long, String>>() {
                            @Override
                            public long extractTimestamp(Tuple3<String, Long, String>
                                stringLongStringTuple3, long l) {
                                return stringLongStringTuple3.f1;
                            }
                        })
            )
            .keyBy(r -> r.f0);

        KeyedStream<Tuple3<String, Long, String>, String> stream2 = env
            .fromElements(

```

```

        Tuple3.of("user_1", 5 * 60 * 1000L, "browse"),
        Tuple3.of("user_1", 6 * 60 * 1000L, "browse")
    )
    .assignTimestampsAndWatermarks(
        WatermarkStrategy
            .Tuple3<String, Long, String>forMonotonousTimestamps()
            .withTimestampAssigner(new
                SerializableTimestampAssigner<Tuple3<String, Long, String>>() {
                    @Override
                    public long extractTimestamp(Tuple3<String, Long, String>
                        stringLongStringTuple3, long l) {
                        return stringLongStringTuple3.f1;
                    }
                })
)
.keyBy(r -> r.f0);

stream1
    .intervalJoin(stream2)
    .between(Time.minutes(-10), Time.minutes(0))
    .process(new ProcessJoinFunction<Tuple3<String, Long, String>,
        Tuple3<String, Long, String>, String>() {
            @Override
            public void processElement(Tuple3<String, Long, String>
                stringLongStringTuple3, Tuple3<String, Long, String>
                stringLongStringTuple32, Context context, Collector<String>
                collector) throws Exception {
                collector.collect(stringLongStringTuple3 + " => " +
                    stringLongStringTuple32);
            }
        })
    .print();

env.execute();
}

}

```

7.3.2 基于窗口的 Join

顾名思义，基于窗口的 Join 需要用到 Flink 中的窗口机制。其原理是将两条输入流中的元素分配到公共窗口中并在窗口完成时进行 Join（或 Cogroup）。

下面的例子展示了如何定义基于窗口的 Join。

```

input1.join(input2)
    .where(...)      // 为 input1 指定键值属性
    .equalTo(...)     // 为 input2 指定键值属性
    .window(...)     // 指定 WindowAssigner
    [.trigger(...)]  // 选择性的指定 Trigger
    [.evictor(...)]  // 选择性的指定 Evictor
    .apply(...)       // 指定 JoinFunction

```

下图展示了 DataStream API 中基于窗口的 Join 是如何工作的。

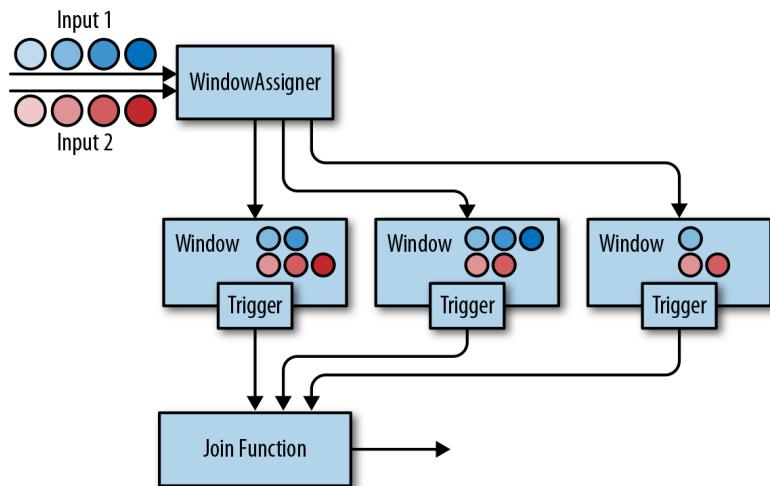


图 7.2: 基于间隔的 Join

两条输入流都会根据各自的键值属性进行分区，公共窗口分配器会将二者的事件映射到公共窗口内（其中同时存储了两条流中的数据）。当窗口的计时器触发时，算子会遍历两个输入中元素的每个组合（叉乘积）去调用 JoinFunction。同时你也可以自定义触发器或移除器。由于两条流中的事件会被映射到同一个窗口中，因此该过程中的触发器和移除器与常规窗口算子中的完全相同。

除了对窗口中的两条流进行 Join，你还可以对它们进行 Cogroup，只需将算子定义开始位置的 join 改为 coGroup() 即可。Join 和 Cogroup 的总体逻辑相同，二者的唯一区别是：Join 会为两侧输入中的每个事件对调用 JoinFunction；而 Cogroup 中用到的 CoGroupFunction 会以两个输入的元素遍历器为参数，只在每个窗口中被调用一次。

 **笔记** 注意，对划分窗口后的数据流进行 Join 可能会产生意想不到的语义。例如，假设你为执行 Join 操作的算子配置了 1 小时的滚动窗口，那么一旦来自两个输入的元素没有被划分到同一窗口，它们就无法 Join 在一起，即使二者彼此仅相差 1 秒钟。

```

public class TwoWindowJoinExample {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
    }
}

```

```

DataStream<Tuple2<String, Long>> stream1 = env
    .fromElements(
        Tuple2.of("a", 1000L),
        Tuple2.of("b", 1000L),
        Tuple2.of("a", 2000L),
        Tuple2.of("b", 2000L)
    )
    .assignTimestampsAndWatermarks(
        WatermarkStrategy
            .<Tuple2<String, Long>>forMonotonousTimestamps()
            .withTimestampAssigner(
                new SerializableTimestampAssigner<Tuple2<String, Long>>() {
                    @Override
                    public long extractTimestamp(Tuple2<String, Long>
                        → stringLongTuple2, long l) {
                        return stringLongTuple2.f1;
                    }
                }
            )
    );
}

DataStream<Tuple2<String, Long>> stream2 = env
    .fromElements(
        Tuple2.of("a", 3000L),
        Tuple2.of("b", 3000L),
        Tuple2.of("a", 4000L),
        Tuple2.of("b", 4000L)
    )
    .assignTimestampsAndWatermarks(
        WatermarkStrategy
            .<Tuple2<String, Long>>forMonotonousTimestamps()
            .withTimestampAssigner(
                new SerializableTimestampAssigner<Tuple2<String, Long>>() {
                    @Override
                    public long extractTimestamp(Tuple2<String, Long>
                        → stringLongTuple2, long l) {
                        return stringLongTuple2.f1;
                    }
                }
            )
    );
}

stream1
    .join(stream2)
    .where(r -> r.f0)
    .equalTo(r -> r.f0)

```

```

>window(TumblingEventTimeWindows.of(Time.seconds(5)))
.apply(new JoinFunction<Tuple2<String, Long>, Tuple2<String, Long>,
      String>() {
    @Override
    public String join(Tuple2<String, Long> stringLongTuple2,
                      Tuple2<String, Long> stringLongTuple22) throws Exception {
        return stringLongTuple2 + " => " + stringLongTuple22;
    }
})
.print();

env.execute();
}
}

```

7.4 处理迟到的元素

水位线可以用来平衡计算的完整性和延迟两方面。除非我们选择一种非常保守的水位线策略 (最大延时设置的非常大, 以至于包含了所有的元素, 但结果是非常大的延迟), 否则我们总需要处理迟到的元素。

迟到的元素是指当这个元素来到时, 这个元素所对应的窗口已经计算完毕了 (也就是说水位线已经没过窗口结束时间了)。这说明迟到这个特性只针对事件时间。

DataStream API 提供了三种策略来处理迟到元素

- 直接抛弃迟到的元素
- 将迟到的元素发送到另一条流中去
- 可以更新窗口已经计算完的结果, 并发出计算结果。

7.4.1 抛弃迟到元素

抛弃迟到的元素是事件时间窗口操作符的默认行为。也就是说一个迟到的元素不会创建一个新的窗口。

process function 可以通过比较迟到元素的时间戳和当前水位线的大小来很轻易的过滤掉迟到元素。

7.4.2 重定向迟到元素

迟到的元素也可以使用旁路输出 (side output) 特性被重定向到另外的一条流中去。迟到元素所组成的旁路输出流可以继续处理或者 sink 到持久化设施中去。

例子

```

public class RedirectLateEvent {

    private static OutputTag<Tuple2<String, Long>> output = new
    → OutputTag<Tuple2<String, Long>>("late-readings"){};

    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
        → StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        DataStream<Tuple2<String, Long>> stream = env
            .socketTextStream("localhost", 9999)
            .map(new MapFunction<String, Tuple2<String, Long>>() {
                @Override
                public Tuple2<String, Long> map(String s) throws Exception {
                    String[] arr = s.split(" ");
                    return Tuple2.of(arr[0], Long.parseLong(arr[1]) * 1000L);
                }
            })
            .assignTimestampsAndWatermarks(
                WatermarkStrategy.
                    // like scala: assignAscendingTimestamps(_._2)
                    <Tuple2<String, Long>>forMonotonousTimestamps()
                    .withTimestampAssigner(new
                        → SerializableTimestampAssigner<Tuple2<String, Long>>() {
                            @Override
                            public long extractTimestamp(Tuple2<String, Long> value, long
                                → l) {
                                return value.f1;
                            }
                        })
            );
    }

    SingleOutputStreamOperator<String> lateReadings = stream
        .keyBy(r -> r.f0)
        .window(TumblingEventTimeWindows.of(Time.seconds(5)))
        .sideOutputLateData(output) // use after keyBy and timeWindow
        .process(new ProcessWindowFunction<Tuple2<String, Long>, String, String,
        → TimeWindow>() {
            @Override
            public void process(String s, Context context, Iterable<Tuple2<String,
            → Long>> iterable, Collector<String> collector) throws Exception {
                long exactSizeIfKnown =
                → iterablespliterator().getExactSizeIfKnown();
                collector.collect(exactSizeIfKnown + " of elements");
            }
        });
}

```

```

    });

    lateReadings.print();
    lateReadings.getSideOutput(output).print();

    env.execute();
}
}

```

下面这个例子展示了 ProcessFunction 如何过滤掉迟到的元素然后将迟到的元素发送到旁路输出流中去。

```

public class RedirectLateEvent {

    private static OutputTag<String> output = new OutputTag<String>("late-readings"){};

    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        SingleOutputStreamOperator<Tuple2<String, Long>> stream = env
            .socketTextStream("localhost", 9999)
            .map(new MapFunction<String, Tuple2<String, Long>>() {
                @Override
                public Tuple2<String, Long> map(String s) throws Exception {
                    String[] arr = s.split(" ");
                    return Tuple2.of(arr[0], Long.parseLong(arr[1]) * 1000L);
                }
            })
            .assignTimestampsAndWatermarks(
                WatermarkStrategy.
                    <Tuple2<String, Long>>forMonotonousTimestamps()
                    .withTimestampAssigner(new
                        SerializableTimestampAssigner<Tuple2<String, Long>>() {
                            @Override
                            public long extractTimestamp(Tuple2<String, Long> value, long
                                → 1) {
                                return value.f1;
                            }
                        })
            )
            .process(new ProcessFunction<Tuple2<String, Long>, Tuple2<String, Long>>()
            {
                @Override
                public void processElement(Tuple2<String, Long> stringLongTuple2,
                    Context context, Collector<Tuple2<String, Long>> collector) throws
                    Exception {

```

```

        if (stringLongTuple2.f1 <
            ↪ context.timerService().currentWatermark()) {
            context.output(output, "late event is comming!");
        } else {
            collector.collect(stringLongTuple2);
        }

    }
});

stream.print();
stream.getSideOutput(output).print();

env.execute();
}
}

```

7.4.3 使用迟到元素更新窗口计算结果

由于存在迟到的元素，所以已经计算出的窗口结果是不准确和不完全的。我们可以使用迟到元素更新已经计算完的窗口结果。

如果我们要求一个 operator 支持重新计算和更新已经发出的结果，就需要在第一次发出结果以后也要保存之前所有的状态。但显然我们不能一直保存所有的状态，肯定会在某一个时间点将状态清空，而一旦状态被清空，结果就再也不能重新计算或者更新了。而迟到的元素只能被抛弃或者发送到旁路输出流。

window operator API 提供了方法来明确声明我们要等待迟到元素。当使用 event-time window，我们可以指定一个时间段叫做 allowed lateness。window operator 如果设置了 allowed lateness，这个 window operator 在水位线没过窗口结束时间时也将不会删除窗口和窗口中的状态。窗口会在一段时间内 (allowed lateness 设置的) 保留所有的元素。

当迟到元素在 allowed lateness 时间内到达时，这个迟到元素会被实时处理并发送到触发器 (trigger)。当水位线没过了窗口结束时间 +allowed lateness 时间时，窗口会被删除，并且所有后来的迟到的元素都会被丢弃。

Allowed lateness 可以使用 allowedLateness() 方法来指定，如下所示：

```

public class UpdateWindowResultWithLateEvent {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            ↪ StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        DataStreamSource<String> stream = env.socketTextStream("localhost", 9999);
    }
}

```

```

    stream
        .map(new MapFunction<String, Tuple2<String, Long>>() {
            @Override
            public Tuple2<String, Long> map(String s) throws Exception {
                String[] arr = s.split(" ");
                return Tuple2.of(arr[0], Long.parseLong(arr[1]) * 1000L);
            }
        })
        .assignTimestampsAndWatermarks(
            WatermarkStrategy.<Tuple2<String,
            → Long>>forBoundedOutOfOrderness(Duration.ofSeconds(5))
            .withTimestampAssigner(new SerializableTimestampAssigner<Tuple2<String,
            → Long>>() {
                @Override
                public long extractTimestamp(Tuple2<String, Long> stringLongTuple2,
                → long l) {
                    return stringLongTuple2.f1;
                }
            })
        )
        .keyBy(r -> r.f0)
        .window(TumblingEventTimeWindows.of(Time.seconds(5)))
        .allowedLateness(Time.seconds(5))
        .process(new UpdateWindowResult())
        .print();

    env.execute();
}

public static class UpdateWindowResult extends ProcessWindowFunction<Tuple2<String,
→ Long>, String, String, TimeWindow> {
    @Override
    public void process(String s, Context context, Iterable<Tuple2<String, Long>>
    → iterable, Collector<String> collector) throws Exception {
        long count = 0L;
        for (Tuple2<String, Long> i : iterable) {
            count += 1;
        }

        // 可见范围比 getRuntimeContext.getState 更小，只对当前 key、当前 window 可见
        // 基于窗口的状态变量，只能当前 key 和当前窗口访问
        ValueState<Boolean> isUpdate = context.windowState().getState(
            new ValueStateDescriptor<Boolean>("isUpdate", Types.BOOLEAN)
        );

        // 当水位线超过窗口结束时间时，触发窗口的第一次计算！
    }
}

```

```
if (isUpdate.value() == null) {
    collector.collect(" 窗口第一次触发计算！一共有 " + count + " 条数据！ ");
    isUpdate.update(true);
} else {
    collector.collect(" 窗口更新了！一共有 " + count + " 条数据！ ");
}
}
```

第八章 有状态算子和应用

状态操作符和用户自定义函数都是我们在写流处理程序时，常用的工具。事实上，大部分稍微复杂一点的逻辑都需要保存数据或者保存计算结果。很多 Flink 内置的操作符例如：source 操作符，sink 操作符等等都是有状态的，也就是说会缓存流数据或者计算结果。例如，窗口操作符将会为 ProcessWindowFunction 收集输入的数据，或者收集 ReduceFunction 计算的结果。而 ProcessFunction 也会保存定时器事件，一些 sink 方法为了做到 exactly-once，会将事务保存下来。除了内置的操作符以及提供的 source 和 sink 操作符，Flink 的 DataStream API 还在 UDF 函数中暴露了可以注册、保存和访问状态的接口。

本章重点讨论有状态的用户自定义函数的实现，以及讨论有状态应用的性能和健壮性。特别的，我们将解释在用户自定义函数中，如何定义不同类型的状态，以及如何与状态进行交互。我们还讨论了性能方面的问题以及如何控制状态大小的问题。

8.1 实现有状态的用户自定义函数

我们知道函数有两种状态，键控状态 (keyed state) 和操作符状态 (operator state)。我们重点来看一下键控状态。我们来看一下如何在 RuntimeContext 中定义键控状态。

用户自定义函数可以使用 keyed state 来存储和访问 key 对应的状态。对于每一个 key，Flink 将会维护一个状态实例。一个操作符的状态实例将会被分发到操作符的所有并行任务中去。这表明函数的每一个并行任务只为所有 key 的某一部分 key 保存 key 对应的状态实例。所以 keyed state 和分布式 key-value map 数据结构非常类似。

keyed state 仅可用于 KeyedStream。Flink 支持以下数据类型的状态变量：

- ValueState<T> 保存单个的值，值的类型为 T。
 - get 操作: ValueState.value()
 - set 操作: ValueState.update(T value)
- 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(T value)
 - ListState.addAll(List<T> values)
 - ListState.get() 返回 Iterable<T>
 - ListState.update(List<T> values)
- MapState<K, V> 保存 Key-Value 对。
 - MapState.get(K key)
 - MapState.put(K key, V value)
 - MapState.contains(K key)
 - MapState.remove(K key)
- ReducingState<T>

- AggregatingState<I, O>

State.clear() 是清空操作。

```
public class ListStateExample {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        env
            .addSource(new SensorSource())
            .filter(r -> r.id.equals("sensor_1"))
            .keyBy(r -> r.id)
            .process(new KeyedProcessFunction<String, SensorReading, String>() {

                private ListState<SensorReading> readings;
                private ValueState<Long> timerTs;

                @Override
                public void open(Configuration parameters) throws Exception {
                    super.open(parameters);
                    readings = getRuntimeContext().getListState(
                        new ListStateDescriptor<SensorReading>("readings",
                            SensorReading.class));
                }

                timerTs = getRuntimeContext().getState(
                    new ValueStateDescriptor<Long>("ts", Types.LONG));
                }

                @Override
                public void processElement(SensorReading sensorReading, Context
                    context, Collector<String> collector) throws Exception {
                    readings.add(sensorReading);
                    if (timerTs.value() == null) {

                        context.timerService().registerProcessingTimeTimer(context.timerService().currentProcessingTime() +
                            + 10 * 1000L);
                        timerTs.update(context.timerService().currentProcessingTime() +
                            10 * 1000L);
                    }
                }

                @Override
                public void onTimer(long timestamp, OnTimerContext ctx,
                    Collector<String> out) throws Exception {

```

```

        super.onTimer(timestamp, ctx, out);
        long count = 0L;
        for(SensorReading r : readings.get()) {
            count++;
        }
        out.collect("there are " + count + " readings");
        timerTs.clear();
    }
}

env.execute();
}
}

```

上面例子中的 KeyedProcessFunction 只能访问当前处理的元素所包含的 key 所对应的状态变量。



笔记 不同 key 对应的 keyed state 是相互隔离的。

- 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。数据类型必须指定，因为 Flink 需要选择合适的序列化器。
- 在 open() 方法中创建 state 变量。注意复习之前的 RichFunction 相关知识。

当一个函数注册了 StateDescriptor 描述符，Flink 会检查状态后端是否已经存在这个状态。这种情况通常出现在应用挂掉要从检查点或者保存点恢复的时候。在这两种情况下，Flink 会将注册的状态连接到已经存在的状态。如果不存在状态，则初始化一个空的状态。

8.2 状态后端

8.2.1 选择一个状态后端

- MemoryStateBackend 将状态当作 Java 的对象(没有序列化操作)存储在 TaskManager JVM 进程的堆上。
- FsStateBackend 将状态存储在本地的文件系统或者远程的文件系统如 HDFS。
- RocksDBStateBackend 将状态存储在 RocksDB 中。

```

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setStateBackend(new FsStateBackend("file:///tmp/checkpoints", false))

```

8.2.2 防止状态泄露

流应用通常需要运行几个月或者几年。如果 state 数据不断增长的话，会爆炸。所以控制 state 数据的大小十分重要。而 Flink 并不会清理 state 和 gc。所以所有的 stateful

operator 都需要控制他们各自的状态数据大小，保证不爆炸。

例如我们之前讲过增量聚合函数 ReduceFunction/AggregateFunction，就可以提前聚合而不给 state 太多压力。

我们来看一个例子，我们实现了一个 KeyedProcessFunction，用来计算连续两次的温度的差值，如果差值超过阈值，报警。

我们之前实现过这个需求，但没有清理掉状态数据。比如一小时内不再产生温度数据的传感器对应的状态数据就可以清理掉了。

8.2.3 配置检查点

10 秒钟保存一次检查点。

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.enableCheckpointing(10000L);
```

第九章 读写外部系统

数据可以存储在不同的系统中，例如：文件系统，对象存储系统（OSS），关系型数据库，Key-Value 存储，搜索引擎索引，日志系统，消息队列，等等。每一种系统都是给特定的应用场景设计的，在某一个特定的目标上超越了其他系统。今天的数据架构，往往包含着很多不同的存储系统。在将一个组件加入到我们的系统中时，我们需要问一个问题：“这个组件和架构中的其他组件能多好的一起工作？”

添加一个像 Flink 这样的数据处理系统，需要仔细的考虑。因为 Flink 没有自己的存储层，而是读取数据和持久化数据都需要依赖外部存储。所以，对于 Flink，针对外部系统提供良好的读取和写入的连接器就很重要了。尽管如此，仅仅能够读写外部系统对于 Flink 这样想要提供任务故障情况下一致性保证的流处理器来讲，是不够的。

在本章中，我们将会讨论 source 和 sink 的连接器。这些连接器影响了 Flink 的一致性保证，也提供了对于最流行的一些外部系统的读写的连接器。我们还将学习如何实现自定义 source 和 sink 连接器，以及如何实现可以向外部系统发送异步读写请求的函数。

9.1 应用的一致性保证

Flink 的检查点和恢复机制定期的会保存应用程序状态的一致性检查点。在故障的情况下，应用程序的状态将会从最近一次完成的检查点恢复，并继续处理。尽管如此，可以使用检查点来重置应用程序的状态无法完全达到令人满意的一致性保证。相反，source 和 sink 的连接器需要和 Flink 的检查点和恢复机制进行集成才能提供有意义的一致性保证。

为了给应用程序提供恰好处理一次语义的状态一致性保证，应用程序的 source 连接器需要能够将 source 的读位置重置到之前保存的检查点位置。当处理一次检查点时，source 操作符将会把 source 的读位置持久化，并在恢复的时候从这些读位置开始重新读取。支持读位置的检查点的 source 连接器一般来说是基于文件的存储系统，如：文件流或者 Kafka source（检查点会持久化某个正在消费的 topic 的读偏移量）。如果一个应用程序从一个无法存储和重置读位置的 source 连接器摄入数据，那么当任务出现故障的时候，数据就会丢失。也就是说我们只能提供 at-most-once）的一致性保证。

Flink 的检查点和恢复机制和可以重置读位置的 source 连接器结合使用，可以保证应用程序不会丢失任何数据。尽管如此，应用程序可能会发出两次计算结果，因为从上一次检查点恢复的应用程序所计算的结果将会被重新发送一次（一些结果已经发送出去了，这时任务故障，然后从上一次检查点恢复，这些结果将被重新计算一次然后发送出去）。所以，可重置读位置的 source 和 Flink 的恢复机制不足以提供端到端的恰好处理一次语义，即使应用程序的状态是恰好处理一次一致性级别。

一个志在提供端到端恰好处理一次语义一致性的应用程序需要特殊的 sink 连接器。sink 连接器可以在不同的情况下使用两种技术来达到恰好处理一次一致性语义：幂等性

写入和事务性写入。

9.1.1 幂等性写入

一个幂等操作无论执行多少次都会返回同样的结果。例如，重复的向 hashmap 中插入同样的 key-value 对就是幂等操作，因为头一次插入操作之后所有的插入操作都不会改变这个 hashmap，因为 hashmap 已经包含这个 key-value 对了。另一方面，append 操作就不是幂等操作了，因为多次 append 同一个元素将会导致列表每次都会添加一个元素。在流处理程序中，幂等写入操作是很有意思的，因为幂等写入操作可以执行多次但不改变结果。所以它们可以在某种程度上缓和 Flink 检查点机制带来的重播计算结果的效应。

需要注意的是，依赖于幂等性 sink 来达到 exactly-once 语义的应用程序，必须保证在从检查点恢复以后，它将会覆盖之前已经写入的结果。例如，一个包含有 sink 操作的应用在 sink 到一个 key-value 存储时必须保证它能够确定的计算出将要更新的 key 值。同时，从 Flink 程序 sink 到的 key-value 存储中读取数据的应用，在 Flink 从检查点恢复的过程中，可能会看到不想看到的结果。当重播开始时，之前已经发出的计算结果可能会被更早的结果所覆盖（因为在恢复过程中）。所以，一个消费 Flink 程序输出数据的应用，可能会观察到时间回退，例如读到了比之前小的计数。也就是说，当流处理程序处于恢复过程中时，流处理程序的结果将处于不稳定的状态，因为一些结果被覆盖掉，而另一些结果还没有被覆盖。一旦重播完成，也就是说应用程序已经通过了之前出故障的点，结果将会继续保持一致性。

9.1.2 事务性写入

第二种实现端到端的恰好处理一次一致性语义的方法基于事务性写入。其思想是只将最近一次成功保存的检查点之前的计算结果写入到外部系统中去。这样就保证了在任务故障的情况下，端到端恰好处理一次语义。应用将被重置到最近一次的检查点，而在该检查点之后并没有向外部系统发出任何计算结果。通过只有当检查点保存完成以后再写入数据这种方法，事务性的方法将不会遭受幂等性写入所遭受的重播不一致的问题。尽管如此，事务性写入却带来了延迟，因为只有在检查点完成以后，我们才能看到计算结果。

Flink 提供了两种构建模块来实现事务性 sink 连接器：write-ahead-log (WAL, 预写式日志) sink 和两阶段提交 sink。WAL 式 sink 将会把所有计算结果写入到应用程序的状态中，等到检查点完成的通知，才会将计算结果发送到 sink 系统。因为 sink 操作会把数据都缓存在状态后段，所以 WAL 可以使用在任何外部 sink 系统上。尽管如此，WAL 还是无法提供刀枪不入的恰好处理一次语义的保证，再加上由于要缓存数据带来的状态后段的状态大小的问题，WAL 模型并不十分完美。

与之形成对比的，2PC sink 需要 sink 系统提供事务的支持或者可以模拟出事务特性的模块。对于每一个检查点，sink 开始一个事务，然后将所有的接收到的数据都添加到事务中，并将这些数据写入到 sink 系统，但并没有提交 (commit) 它们。当事务接收到检查点完成的通知时，事务将被 commit，数据将被真正的写入 sink 系统。这项机制主要

依赖于一次 sink 可以在检查点完成之前开始事务，并在应用程序从一次故障中恢复以后再 commit 的能力。

2PC 协议依赖于 Flink 的检查点机制。检查点屏障是开始一个新的事务的通知，所有操作符自己的检查点成功的通知是它们可以 commit 的投票，而作业管理器通知一个检查点成功的消息是 commit 事务的指令。于 WAL sink 形成对比的是，2PC sinks 依赖于 sink 系统和 sink 本身的实现可以实现恰好处理一次语义。更多的，2PC sink 不断的将数据写入到 sink 系统中，而 WAL 写模型就会有之前所述的问题。

表 9.1: 总结

	不可重置的源	可重置的源
Any Sink	at-most-once	at-least-once
幂等性 sink	at-most-once	exactly-once
预写式日志 sink	at-most-once	at-least-once
2PC sink	at-most-once	exactly-once

9.1.3 Flink 提供的连接器

Flink 提供了读写很多存储系统的连接器。消息队列，日志系统，例如 Apache Kafka, Kinesis, RabbitMQ 等等这些是常用的数据源。在批处理环境中，数据流很可能是监听一个文件系统，而当新的数据落盘的时候，读取这些新数据。

在 sink 一端，数据流经常写入到消息队列中，以供接下来的流处理程序消费。数据流也可能写入到文件系统中做持久化，或者交给批处理程序来进行分析。数据流还可能被写入到 key-value 存储或者关系型数据库中，例如 Cassandra, ElasticSearch 或者 MySQL 中，这样数据可供查询，还可以在仪表盘中显示出来。

不幸的是，对于大多数存储系统并没有标准接口，除了针对 DBMS 的 JDBC。相反，每一个存储系统都需要有自己的特定的连接器。所以，Flink 需要维护针对不同存储系统（消息队列，日志系统，文件系统，k-v 数据库，关系型数据库等等）的连接器实现。

Flink 提供了针对 Apache Kafka, Kinesis, RabbitMQ, Apache Nifi, 各种文件系统，Cassandra, Elasticsearch, 还有 JDBC 的连接器。除此之外，Apache Bahir 项目还提供了额外的针对例如 ActiveMQ, Akka, Flume, Netty, 和 Redis 等的连接器。

9.1.4 Apache Kafka Source 连接器

Apache Kafka 是一个分布式流式平台。它的核心是一个分布式的发布订阅消息系统。

Kafka 将事件流组织为所谓的 topics。一个主题就是一个事件日志系统，Kafka 可以保证主题中的数据在被读取时和这些数据在被写入时相同的顺序。为了扩大读写的规模，主题可以分裂为多个分区，这些分区分布在一个集群上面。这时，读写顺序的保证就限制到了分区这个粒度，Kafka 并没有提供从不同分区读取数据时的顺序保证。Kafka 分区的读位置称为偏移量 (offset)。

Kafka 的依赖引入如下：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
</dependency>
```

Flink Kafka 连接器并行的摄入事件流。每一个并行 source 任务可以从一个或者多个分区中读取数据。任务将会跟踪每一个分区当前的读偏移量，然后将读偏移量写入到检查点数据中。当从任务故障恢复时，读偏移量将被恢复，而 source 任务将从检查点保存的读偏移量开始重新读取数据。Flink Kafka 连接器并不依赖 Kafka 自己的 offset-tracking 机制（基于消费者组实现）。下图展示了分区如何分配给 source 实例。

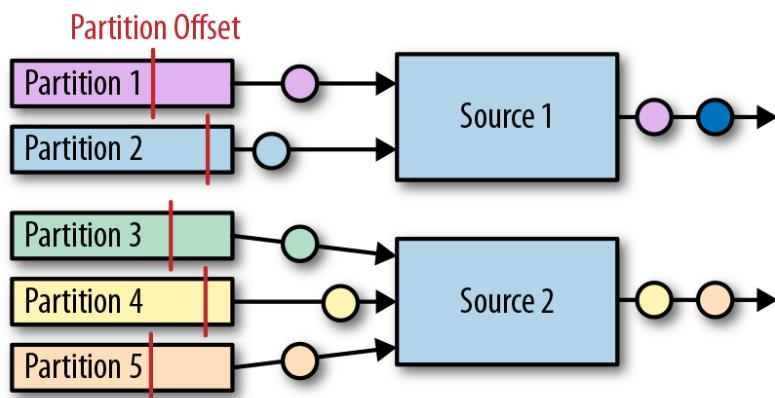


图 9.1: 消费 Kafka

Kafka source 连接器使用如下代码创建

```
Properties properties = new Properties();
properties.setProperty("bootstrap.servers", "localhost:9092");
properties.setProperty("group.id", "consumer-group");
properties.setProperty("key.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
properties.setProperty("value.deserializer",
    "org.apache.kafka.common.serialization.StringDeserializer");
properties.setProperty("auto.offset.reset", "latest");

env
    .addSource(new FlinkKafkaConsumer<String>(
        "atguigu",
        new SimpleStringSchema(),
        properties
    ))
    .print();
```

构造器接受三个参数。第一个参数定义了从哪些 topic 中读取数据，可以是一个 topic，也可以是 topic 列表，还可以是匹配所有想要读取的 topic 的正则表达式。当从多个 topic

中读取数据时，Kafka 连接器将会处理所有 topic 的分区，将这些分区的数据放到一条流中去。

第二个参数是一个 DeserializationSchema 或者 KeyedDeserializationSchema。Kafka 消息被存储为原始的字节数据，所以需要反序列化成 Java 对象。上例中使用的 SimpleStringSchema，是一个内置的 DeserializationSchema，它仅仅是简单的将字节数组反序列化成字符串。DeserializationSchema 和 KeyedDeserializationSchema 是公共的接口，所以我们自定义反序列化逻辑。

第三个参数是一个 Properties 对象，设置了用来读写的 Kafka 客户端的一些属性。

9.1.5 Apache Kafka Sink 连接器

下面的例子展示了如何创建一个 Kafka sink

```
Properties properties = new Properties();
properties.put("bootstrap.servers", "localhost:9092");
stream
    .addSink(
        new FlinkKafkaProducer<String>(
            "test",
            new SimpleStringSchema(),
            properties
        ));

```

9.1.6 Kafka Sink 的 at-least-once 保证

Flink 的 Kafka sink 提供了基于配置的一致性保证。Kafka sink 使用下面的条件提供了至少处理一次保证：

- Flink 检查点机制开启，所有的数据源都是可重置的。
- 当写入失败时，sink 连接器将会抛出异常，使得应用程序挂掉然后重启。这是默认行为。应用程序内部的 Kafka 客户端还可以配置为重试写入，只要提前声明当写入失败时，重试几次这样的属性（retries property）。
- sink 连接器在完成它的检查点之前会等待 Kafka 发送已经将数据写入的通知。

9.1.7 Kafka Sink 的恰好处理一次语义保证

Kafka 0.11 版本引入了事务写特性。由于这个新特性，Flink Kafka sink 可以为输出结果提供恰好处理一次语义的一致性保证，只要经过合适的配置就行。Flink 程序必须开启检查点机制，并从可重置的数据源进行消费。FlinkKafkaProducer 还提供了包含 Semantic 参数的构造器来控制 sink 提供的一致性保证。可能的取值如下：

- Semantic.NONE，不提供任何一致性保证。数据可能丢失或者被重写多次。
- Semantic.AT_LEAST_ONCE，保证无数据丢失，但可能被处理多次。这个是默认设置。

- Semantic.EXACTLY_ONCE，基于 Kafka 的事务性写入特性实现，保证每条数据恰好处理一次。

例子

```
DataStreamSource<String> text = env.socketTextStream("localhost", 9999, "\n");

String brokerList = "localhost:9092";
String topic = "topic";

Properties prop = new Properties();
prop.setProperty("bootstrap.servers", brokerList);

//第一种解决方案，设置 FlinkKafkaProducer 里面的事务超时时间
//设置事务超时时间
//prop.setProperty("transaction.timeout.ms", "60000*15"");

//第二种解决方案，设置 kafka 的最大事务超时时间

//FlinkKafkaProducer<String> myProducer = new FlinkKafkaProducer<>(brokerList, topic,
//→ new SimpleStringSchema());

//使用仅一次语义的 kafkaProducer
FlinkKafkaProducer<String> myProducer = new FlinkKafkaProducer<>(
    topic,
    new KeyedSerializationSchemaWrapper<String>(new SimpleStringSchema()),
    prop,
    FlinkKafkaProducer.Semantic.EXACTLY_ONCE
);
text.addSink(myProducer);
```

9.1.8 Redis Sink 连接器

导入依赖

```
<dependency>
    <groupId>org.apache.bahir</groupId>
    <artifactId>flink-connector-redis_${scala.binary.version}</artifactId>
    <version>1.0</version>
</dependency>
```

例子

```
public class WriteToRedisExample {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
```

```

env.setParallelism(1);

DataStream<SensorReading> stream = env.addSource(new SensorSource());

FlinkJedisPoolConfig conf = new
    FlinkJedisPoolConfig.Builder().setHost("localhost").build();

stream.addSink(new RedisSink<Event>(conf, new MyRedisSink()));

env.execute();
}

public static class MyRedisSink implements RedisMapper<SensorReading> {
    @Override
    public String getKeyFromData(SensorReading r) {
        return r.id;
    }

    @Override
    public String getValueFromData(SensorReading r) {
        return r.temperature + "";
    }

    @Override
    public RedisCommandDescription getCommandDescription() {
        return new RedisCommandDescription(RedisCommand.HSET, "sensor");
    }
}
}

```

9.1.9 ElasticSearch Sink 连接器

导入依赖

```

<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-elasticsearch7_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
</dependency>

```

例子

```

public class SinkToES {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
    }
}

```

```

env.setParallelism(1);

ArrayList<HttpHost> httpHosts = new ArrayList<>();
httpHosts.add(new HttpHost("127.0.0.1", 9200, "http"));

ElasticsearchSink.Builder<SensorReading> esBuilder = new
    ↪ ElasticsearchSink.Builder<>(
        httpHosts,
        new ElasticsearchSinkFunction<SensorReading>() {
            @Override
            public void process(SensorReading r, RuntimeContext runtimeContext,
                ↪ RequestIndexer requestIndexer) {
                HashMap<String, String> data = new HashMap<>();
                data.put(r.id, r.temperature + "");
            }

            IndexRequest indexRequest = Requests
                .indexRequest()
                .index("sensor-reading")
                .source(data);

            requestIndexer.add(indexRequest);
        }
    );
);

esBuilder.setBulkFlushMaxActions(1);

DataStream<Event> stream = env.addSource(new SensorSource());

stream.addSink(esBuilder.build());

env.execute();
}
}

```

9.2 实现自定义源函数

DataStream API 提供了两个接口来实现 source 连接器：

- SourceFunction 和 RichSourceFunction 可以用来定义非并行的 source 连接器，source 跑在单任务上。
- ParallelSourceFunction 和 RichParallelSourceFunction 可以用来定义跑在并行实例上的 source 连接器。

除了并行和非并行的区别，这两种接口完全一样。就像 process function 的 rich 版本一样，RichSourceFunction 和 RichParallelSourceFunction 的子类可以 override open() 和

`close()` 方法，也可以访问 `RuntimeContext`，`RuntimeContext` 提供了并行任务实例的数量，当前任务实例的索引，以及一些其他信息。

`SourceFunction` 和 `ParallelSourceFunction` 定义了两种方法：

- `void run(SourceContext ctx)`
- `cancel()`

`run()` 方法用来读取或者接收数据然后将数据摄入到 Flink 应用中。根据接收数据的系统，数据可能是推送的也可能是拉取的。Flink 仅仅在特定的线程调用 `run()` 方法一次，通常情况下会是一个无限循环来读取或者接收数据并发送数据。任务可以在某个时间点被显式的取消，或者由于流是有限流，当数据被消费完毕时，任务也会停止。

当应用被取消或者关闭时，`cancel()` 方法会被 Flink 调用。为了优雅的关闭 Flink 应用，`run()` 方法需要在 `cancel()` 被调用以后，立即终止执行。下面的例子显示了一个简单的源函数的例子：从 0 数到 `Long.MAX_VALUE`。

```
public static class CountSource implements SourceFunction<Long> {
    private Boolean isRunning = true;

    @Override
    public void run(SourceContext<Long> ctx) throws Exception {
        long cnt = -1;
        while (isRunning && cnt < Long.MAX_VALUE) {
            cnt += 1;
            ctx.collect(cnt);
        }
    }

    @Override
    public void cancel() {
        isRunning = false;
    }
}
```

9.2.1 可重置的源函数

之前我们讲过，应用程序只有使用可以重播输出数据的数据源时，才能提供令人满意的一致性保证。如果外部系统暴露了获取和重置读偏移量的 API，那么 `source` 函数就可以重播源数据。这样的例子包括一些能够提供文件流的偏移量的文件系统，或者提供 `seek` 方法用来移动到文件的特定位置的文件系统。或者 Apache Kafka 这种可以为每一个主题的分区提供偏移量并且可以设置分区的读位置的系统。一个反例就是 `source` 连接器连接的是 `socket`，`socket` 将会立即丢弃已经发送过的数据。

支持重播输出的源函数需要和 Flink 的检查点机制集成起来，还需要在检查点被处理时，持久化当前所有的读取位置。当应用从一个保存点（`savepoint`）恢复或者从故障恢复时，Flink 会从最近一次的检查点或者保存点中获取读偏移量。如果程序开始时并不存

在状态，那么读偏移量将会被设置到一个默认值。一个可重置的源函数需要实现 CheckpointedFunction 接口，还需要能够存储读偏移量和相关的元数据，例如文件的路径，分区的 ID。这些数据将被保存在 list state 或者 union list state 中。

下面的例子将 CountSource 重写为可重置的数据源。

```
public static class ResettableCountSource
    implements SourceFunction<Long>, CheckpointedFunction {

    private Boolean isRunning = true;
    private Long cnt;
    private ListState<Long> offsetState;

    @Override
    public void run(SourceContext<Long> ctx) throws Exception {
        while (isRunning && cnt < Long.MAX_VALUE) {
            // synchronize data emission and checkpoints
            synchronized (ctx.getCheckpointLock()) {
                cnt += 1;
                ctx.collect(cnt);
            }
        }
    }

    @Override
    public void cancel() {
        isRunning = false;
    }

    @Override
    public void snapshotState(FunctionSnapshotContext context) throws Exception {
        // remove previous cnt
        offsetState.clear();
        // add current cnt
        offsetState.add(cnt);
    }

    @Override
    public void initializeState(FunctionInitializationContext context) throws Exception
    {
        ListStateDescriptor<Long> desc = new ListStateDescriptor<>(
            "offset", Types.LONG);
        offsetState = context
            .getOperatorStateStore()
            .getListState(desc);
        // initialize cnt variable
        Iterable<Long> it = offsetState.get();
    }
}
```

```

        if (null == it || !it.iterator().hasNext()) {
            cnt = -1L;
        } else {
            cnt = it.iterator().next();
        }
    }
}

```

9.3 实现自定义 sink 函数

DataStream API 中，任何运算符或者函数都可以向外部系统发送数据。DataStream 不需要最终流向 sink 运算符。例如，我们可能实现了一个 FlatMapFunction，这个函数将每一个接收到的数据通过 HTTP POST 请求发送出去，而不使用 Collector 发送到下一个运算符。DataStream API 也提供了 SinkFunction 接口以及对应的 rich 版本 RichSinkFunction 抽象类。SinkFunction 接口提供了一个方法：

```
void invoke(IN value, Context ctx)
```

SinkFunction 的 Context 可以访问当前处理时间，当前水位线，以及数据的时间戳。

下面的例子展示了一个简单的 SinkFunction，可以将传感器读数写入到 socket 中去。需要注意的是，我们需要在启动 Flink 程序前启动一个监听相关端口的进程。否则将会抛出 ConnectException 异常。可以运行 nc -l localhost 9191 命令。

```

DataStream<SensorReading> readings = env.addSource(new SensorSource());

// write the sensor readings to a socket
readings
    .addSink(new SimpleSocketSink("localhost", 9191))
    // set parallelism to 1 because only one thread can write to a socket
    .setParallelism(1);

public static class SimpleSocketSink extends RichSinkFunction<SensorReading> {
    private String host;
    private Integer port;

    private Socket socket;
    private PrintStream writer;

    public SimpleSocketSink(String host, Integer port) {
        this.host = host;
        this.port = port;
    }
}

```

```

@Override
public void open(Configuration parameters) throws Exception {
    super.open(parameters);
    socket = new Socket(InetAddress.getByName(host), port);
    writer = new PrintStream(socket.getOutputStream());
}

@Override
public void invoke(SensorReading value, Context context) throws Exception {
    writer.println(value.toString());
    writer.flush();
}

@Override
public void close() throws Exception {
    super.close();
    writer.close();
    socket.close();
}
}

```

之前我们讨论过，端到端的一致性保证建立在 sink 连接器的属性上面。为了达到端到端的恰好处理一次语义的目的，应用程序需要幂等性的 sink 连接器或者事务性的 sink 连接器。上面例子中的 SinkFunction 既不是幂等写入也不是事务性的写入。由于 socket 具有只能添加（append-only）这样的属性，所以不可能实现幂等性的写入。又因为 socket 不具备内置的事务支持，所以事务性写入就只能使用 Flink 的 WAL sink 特性来实现了。接下来我们将学习如何实现幂等 sink 连接器和事务 sink 连接器。

9.3.1 幂等 sink 连接器

对于大多数应用，SinkFunction 接口足以实现一个幂等性写入的 sink 连接器了。需要以下两个条件：

- 结果数据必须具有确定性的 key，在这个 key 上面幂等性更新才能实现。例如一个计算每分钟每个传感器的平均温度值的程序，确定性的 key 值可以是传感器的 ID 和每分钟的时间戳。确定性的 key 值，对于在故障恢复的场景下，能够正确的覆盖结果非常的重要。
- 外部系统支持针对每个 key 的更新，例如关系型数据库或者 key-value 存储。

下面的例子展示了如何实现一个针对 JDBC 数据库的幂等写入 sink 连接器，这里使用的是 MySQL 数据库。

建表语句

```
create database sensor;
create table temps(id varchar(20), temp float);
```

导入依赖

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.21</version>
</dependency>
```

代码

```
public class WriteToMySQLExample {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        DataStream<SensorReading> stream = env.addSource(new SensorSource());

        stream.addSink(new MyJDBCSink());

        env.execute();
    }

    public static class MyJDBCSink extends RichSinkFunction<SensorReading> {
        private Connection conn;
        private PreparedStatement insertStmt;
        private PreparedStatement updateStmt;

        @Override
        public void open(Configuration parameters) throws Exception {
            super.open(parameters);
            conn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/sensor",
                "zuoyuan",
                "zuoyuan"
            );
            insertStmt = conn.prepareStatement("INSERT INTO temps (id, temp) VALUES (?, ?");
            updateStmt = conn.prepareStatement("UPDATE temps SET temp = ? WHERE id = ?");
        }

        @Override
```

```
public void invoke(SensorReading value, Context context) throws Exception {
    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();
    }
}

@Override
public void close() throws Exception {
    super.close();
    insertStmt.close();
    updateStmt.close();
    conn.close();
}
```

9.3.2 事务性 sink 连接器

事务写入 sink 连接器需要和 Flink 的检查点机制集成，因为只有在检查点成功完成以后，事务写入 sink 连接器才会向外部系统 commit 数据。

为了简化事务性 sink 的实现, Flink 提供了两个模版用来实现自定义 sink 运算符。这两个模版都实现了 CheckpointListener 接口。CheckpointListener 接口将会从作业管理器接收到检查点完成的通知。

- `GenericWriteAheadSink` 模版会收集检查点之前的所有数据，并将数据存储到 sink 任务的运算符状态中。状态保存到了检查点中，并在任务故障的情况下恢复。当任务接收到检查点完成的通知时，任务会将所有的数据写入到外部系统中。
 - `TwoPhaseCommitSinkFunction` 模版利用了外部系统的事务特性。对于每一个检查点，任务首先开始一个新的事务，并将接下来所有的数据都写到外部系统的当前事务上下文中去。当任务接收到检查点完成的通知时，sink 连接器将会 commit 这个事务。

GENERICWRITEAHEADSINK

GenericWriteAheadSink 使得 sink 运算符可以很方便的实现。这个运算符和 Flink 的检查点机制集成使用，目标是将每一条数据恰好一次写入到外部系统中去。需要注意的是，在发生故障的情况下，write-ahead log sink 可能会不止一次的发送相同的数据。所以 GenericWriteAheadSink 无法提供完美无缺的恰好处理一次语义的一致性保证，而是仅能提供 at-least-once 这样的保证。我们接下来详细的讨论这些场景。

GenericWriteAheadSink 的原理是将接收到的所有数据都追加到有检查点分割好的预写式日志中去。每当 sink 运算符碰到检查点屏障，运算符将会开辟一个新的 section，并将接下来的所有数据都追加到新的 section 中去。WAL（预写式日志）将会保存到运算符状态中。由于 log 能被恢复，所有不会有数据丢失。

当 GenericWriteAheadSink 接收到检查点完成的通知时，将会发送对应检查点的 WAL 中存储的所有数据。当所有数据发送成功，对应的检查点必须在内部提交。

检查点的提交分两步。第一步，sink 持久化检查点被提交的信息。第二步，删除 WAL 中所有的数据。我们不能将 commit 信息保存在 Flink 应用程序状态中，因为状态不是持久化的，会在故障恢复时重置状态。相反，GenericWriteAheadSink 依赖于可插拔的组件在一个外部持久化存储中存储和查找提交信息。这个组件就是 CheckpointCommitter。

继承 GenericWriteAheadSink 的运算符需要提供三个构造器函数。

- CheckpointCommitter
- TypeSerializer，用来序列化输入数据。
- 一个 job ID，传给 CheckpointCommitter，当应用重启时可以识别 commit 信息。

还有，write-ahead 运算符需要实现一个单独的方法：

```
boolean sendValues(Iterable<IN> values, long chkpntId, long timestamp)
```

当检查点完成时，GenericWriteAheadSink 调用 sendValues() 方法来将数据写入到外部存储系统中。这个方法接收一个检查点对应的所有数据的迭代器，检查点的 ID，检查点被处理时的时间戳。当数据写入成功时，方法必须返回 true，写入失败返回 false。

之前我们讲过，GenericWriteAheadSink 无法提供完美的 exactly-once 保证。有两个故障状况会导致数据可能被发送不止一次。

- 当任务执行 sendValues() 方法时，程序挂掉了。如果外部系统无法原子性的写入所有数据（要么都写入要么都不写），一些数据可能会写入，而另一些数据并没有被写入。由于 checkpoint 还没有 commit，所以在任务恢复的过程中一些数据可能会被再次写入。
- 所有数据都写入成功了，sendValues() 方法也返回 true 了；但在 CheckpointCommitter 方法被调用之前程序挂了，或者 CheckpointCommitter 在 commit 检查点时失败了。那么在恢复的过程中，所有未被提交的检查点将会被重新写入。

TWOPHASECOMMITTSINKFUNCTION

Flink 提供了 TwoPhaseCommitSinkFunction 接口来简化 sink 函数的实现。这个接口保证了端到端的 exactly-once 语义。2PC sink 函数是否提供这样的一致性保证取决于我们的实现细节。我们需要讨论一个问题：“2PC 协议是否开销太大？”

通常来讲，为了保证分布式系统的一致性，2PC 是一个非常昂贵的方法。尽管如此，在 Flink 的语境下，2PC 协议针对每一个检查点只运行一次。TwoPhaseCommitSinkFunction 和 WAL sink 很相似，不同点在于前者不会将数据收集到 state 中，而是会写入到外部系统事务的上下文中。

TwoPhaseCommitSinkFunction 实现了以下协议。在 sink 任务发送出第一条数据之前，

任务将在外部系统中开始一个事务，所有接下来的数据将被写入这个事务的上下文中。当作业管理器初始化检查点并将检查点屏障插入到流中的时候，2PC 协议的投票阶段开始。当运算符接收到检查点屏障，运算符将保存它的状态，当保存完成时，运算符将发送一个 acknowledgement 信息给作业管理器。当 sink 任务接收到检查点屏障时，运算符将会持久化它的状态，并准备提交当前的事务，以及 acknowledge JobManager 中的检查点。发送给作业管理器的 acknowledgement 信息类似于 2PC 协议中的 commit 投票。sink 任务还不能提交事务，因为它还没有保证所有的任务都已经完成了它们的检查点操作。sink 任务也会为下一个检查点屏障之前的所有数据开始一个新的事务。

当作业管理器成功接收到所有任务实例发出的检查点操作成功的通知时，作业管理器将会把检查点完成的通知发送给所有感兴趣的事务。这里的通知对应于 2PC 协议的提交命令。当 sink 任务接收到通知时，它将 commit 所有处于开启状态的事务。一旦 sink 任务 acknowledge 了检查点操作，它必须能够 commit 对应的事务，即使任务发生故障。如果 commit 失败，数据将会丢失。

让我们总结一下外部系统需要满足什么样的要求：

- 外部系统必须提供事务支持，或者 sink 的实现能在外部系统上模拟事务功能。
- 在检查点操作期间，事务必须处于 open 状态，并接收这段时间数据的持续写入。
- 事务必须等到检查点操作完成的通知到来才可以提交。在恢复周期中，可能需要一段时间等待。如果 sink 系统关闭了事务（例如超时了），那么未被 commit 的数据将会丢失。
- sink 必须在进程挂掉后能够恢复事务。一些 sink 系统会提供事务 ID，用来 commit 或者 abort 一个开始的事务。
- commit 一个事务必须是一个幂等性操作。sink 系统或者外部系统能够观察到事务已经被提交，或者重复提交并没有副作用。

下面的例子可能会让上面的一些概念好理解一些。

```
public static class TransactionalFileSink extends TwoPhaseCommitSinkFunction<Long,
    String, Void> {

    private BufferedWriter transactionWriter;

    public TransactionalFileSink() {
        super(StringSerializer.INSTANCE, VoidSerializer.INSTANCE);
    }

    @Override
    protected String beginTransaction() throws Exception {
        long timeNow = System.currentTimeMillis();
        int taskIdx = this.getRuntimeContext().getIndexOfThisSubtask();
        String transactionFile = timeNow + "-" + taskIdx;
        Path tFilePath = Paths.get("/home/zuoyuan/filetemp/" + transactionFile);
        Files.createFile(tFilePath);
```

```

        this.transactionWriter = Files.newBufferedWriter(tFilePath);
        System.out.println("create tx");
        return transactionFile;
    }

    @Override
    protected void invoke(String transaction, Long value, Context context) throws
    → Exception {
        transactionWriter.write(value.toString());
        transactionWriter.write('\n');
    }

    @Override
    protected void preCommit(String transaction) throws Exception {
        transactionWriter.flush();
        transactionWriter.close();
    }

    @Override
    protected void commit(String transaction) {
        Path tFilePath = Paths.get("/home/zuoyuan/filetemp/" + transaction);
        if (Files.exists(tFilePath)) {
            try {
                Path cFilePath = Paths.get("/home/zuoyuan/filetarget/" + transaction);
                Files.move(tFilePath, cFilePath);
                System.out.println("commit complete");
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    @Override
    protected void abort(String transaction) {
        Path tFilePath = Paths.get("/home/zuoyuan/filetemp/" + transaction);
        if (Files.exists(tFilePath)) {
            try {
                Files.delete(tFilePath);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

TwoPhaseCommitSinkFunction<IN, TXN, CONTEXT> 包含如下三个范型参数：

- IN 表示输入数据的类型。
- TXN 定义了一个事务的标识符，可以用来识别和恢复事务。
- CONTEXT 定义了自定义的上下文。

TwoPhaseCommitSinkFunction 的构造器需要两个 TypeSerializer。一个是 TXN 的类型，另一个是 CONTEXT 的类型。

最后，TwoPhaseCommitSinkFunction 定义了五个需要实现的方法：

- beginTransaction() 开始一个事务，并返回事务的标识符。
- invoke(TXN txn, IN value, Context context) 将值写入到当前事务中。
- preCommit(TXN txn) 预提交一个事务。一个预提交的事务不会接收新的写入。
- commit(TXN txn) 提交一个事务。这个操作必须是幂等的。
- abort(TXN txn) 终止一个事务。

第十章 搭建 Flink 运行流式应用

10.1 独立集群部署

独立集群包含至少一个 master 进程，以及至少一个 TaskManager 进程，TaskManager 进程运行在一台或者多台机器上。所有的进程都是 JVM 进程。下图展示了独立集群的部署。

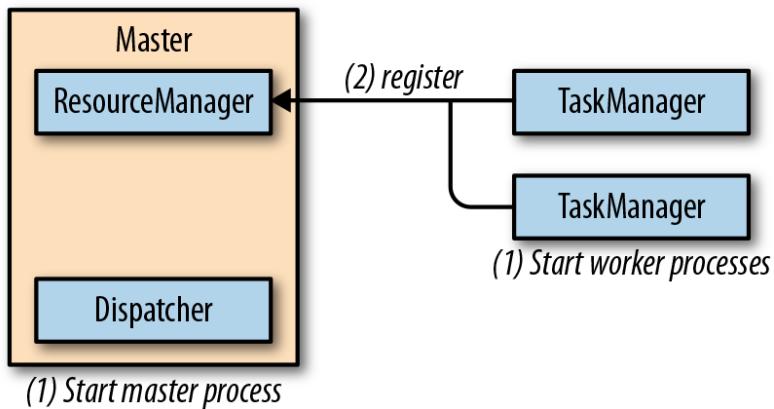


图 10.1: 独立集群部署

master 进程在不同的线程中运行了一个 Dispatcher 和一个 ResourceManager。一旦它们开始运行，所有 TaskManager 都将在 ResourceManager 中进行注册。下图展示了一个任务如何提交到一个独立集群中去。

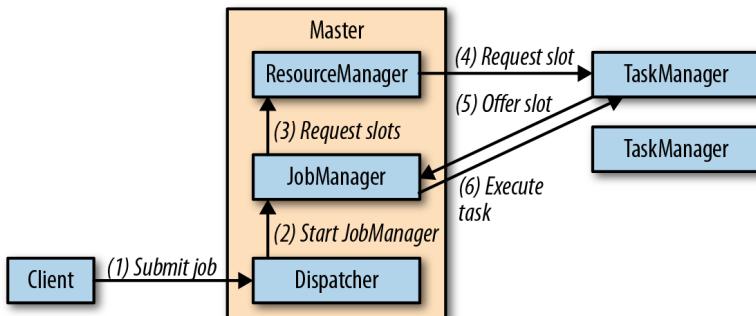


图 10.2: 提交流程

客户端向 Dispatcher 提交了一个任务，Dispatcher 将会启动一个作业管理器线程，并提供执行所需的 JobGraph。作业管理器向 ResourceManager 请求必要的 task slots。一旦请求的 slots 分配好，作业管理器就会部署 job。

在 standalone 这种部署方式中，master 和 worker 进程在失败以后，并不会自动重启。如果有足够的 slots 可供使用，job 是可以从一次 worker 失败中恢复的。只要我们运行多个 worker 就好了。但如果 job 想从 master 失败中恢复的话，则需要进行高可用 (HA) 的配置了。

10.2 YARN 部署

YARN 是 Apache Hadoop 的资源管理组件。用来计算集群环境所需要的 CPU 和内存资源，然后提供给应用程序请求的资源。

Flink 在 YARN 上运行，有两种模式：job 模式和 session 模式。在 job 模式中，Flink 集群用来运行一个单独的 job。一旦 job 结束，Flink 集群停止，并释放所有资源。下图展示了 Flink 的 job 如何提交到 YARN 集群。

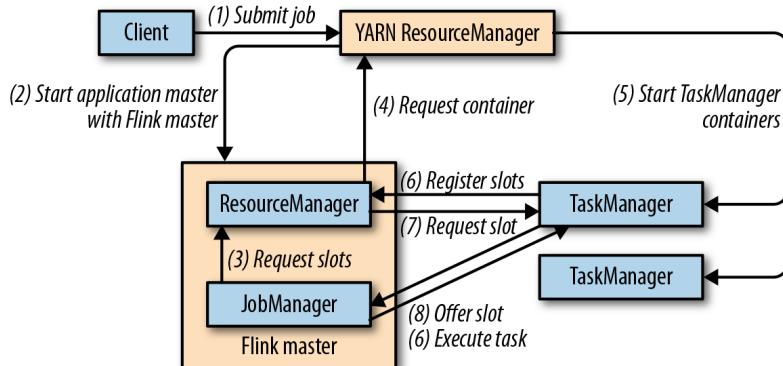


图 10.3: YARN 部署

当客户端提交任务时，客户端将建立和 YARN ResourceManager 的连接，然后启动一个新的 YARN 应用的 master 进程，进程中包含一个作业管理器线程和一个 ResourceManager。作业管理器向 ResourceManager 请求所需要的 slots，用来运行 Flink 的 job。接下来，Flink 的 ResourceManager 将向 Yarn 的 ResourceManager 请求容器，然后启动 TaskManager 进程。一旦启动，TaskManager 会将 slots 注册在 Flink 的 ResourceManager 中，Flink 的 ResourceManager 将把 slots 提供给作业管理器。最终，作业管理器把 job 的任务提交给 TaskManager 执行。

sesison 模式将启动一个长期运行的 Flink 集群，这个集群可以运行多个 job，需要手动停止集群。如果以 session 模式启动，Flink 将会连接到 YARN 的 ResourceManager，然后启动一个 master 进程，包括一个 Dispatcher 线程和一个 Flink 的 ResourceManager 的线程。下图展示了一个 Flink YARN session 的启动。

当一个作业被提交运行，分发器将启动一个作业管理器线程，这个线程将向 Flink 的 ResourceManager 请求所需要的 slots。如果没有足够的 slots，Flink 的 ResourceManager 将向 YARN 的 ResourceManager 请求额外的容器，来启动 TaskManager 进程，并在 Flink 的 ResourceManager 中注册。一旦所需 slots 可用，Flink 的 ResourceManager 将把 slots 分配给作业管理器，然后开始执行 job。下图展示了 job 如何在 session 模式下执行。

无论是作业模式还是会话模式，Flink 的 ResourceManager 都会自动对故障的 TaskManager 进行重启。你可以通过 `./conf/flink-conf.yaml` 配置文件来控制 Flink 在 YARN 上的故障恢复行为。例如，可以配置有多少容器发生故障后终止应用。

无论使用 job 模式还是 sesison 模式，都需要能够访问 Hadoop。

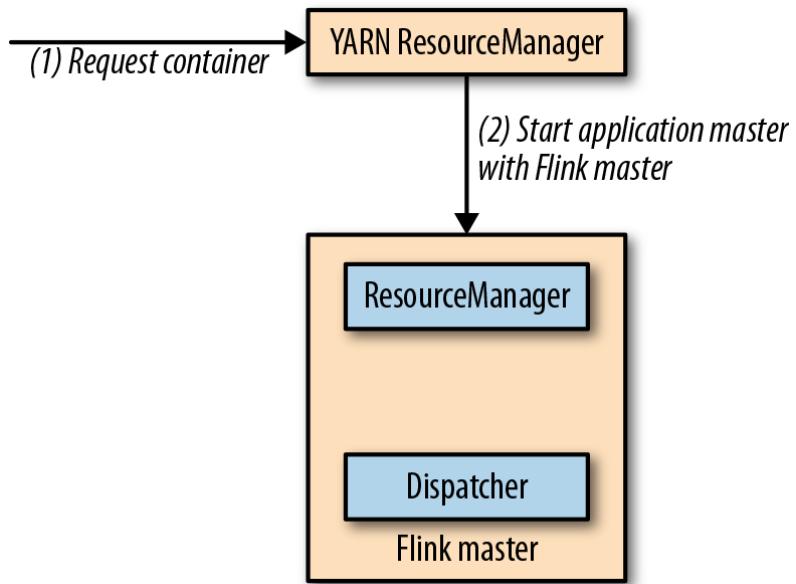


图 10.4: YARN 部署

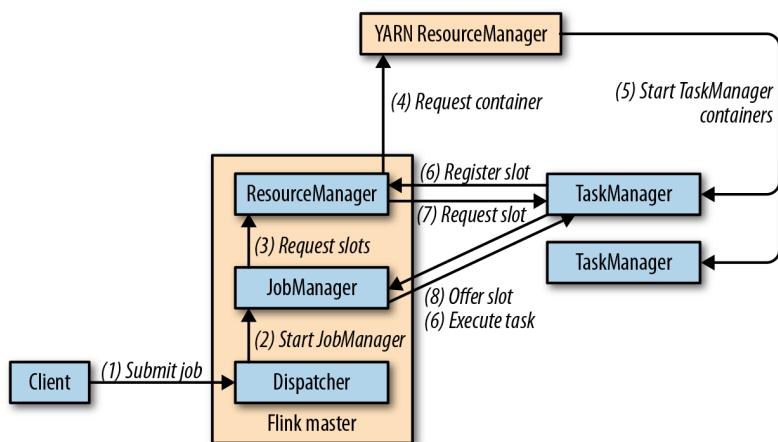


图 10.5: YARN 部署

10.3 高可用部署

Flink 的高可用配置需要 Apache ZooKeeper 组件，以及一个分布式文件系统，例如 HDFS 等等。作业管理器将会把相关信息都存储在文件系统中，并将指向文件系统中相关信息的指针保存在 ZooKeeper 中。一旦失败，一个新的作业管理器将从 ZooKeeper 中指向相关信息的指针所指向的文件系统中读取元数据，并恢复运行。

10.4 保存点操作

```
$ ./bin/flink savepoint <jobId> [<savepointPath>]
```

例如

```
$ ./bin/flink savepoint bc0b2ad61ecd4a615d92ce25390f61ad \
hdfs://xxx:50070/savepoints
Triggering savepoint for job bc0b2ad61ecd4a615d92ce25390f61ad.
Waiting for response...
Savepoint completed.
Path: hdfs://xxx:50070/savepoints/savepoint-bc0b2a-63cf5d5cccef8
You can resume your program from this savepoint with the run command.
```

删除保存点文件

```
$ ./bin/flink savepoint -d <savepointPath>
```

例子

```
$ ./bin/flink savepoint -d \
hdfs://xxx:50070/savepoints/savepoint-bc0b2a-63cf5d5cccef8
Disposing savepoint 'hdfs://xxx:50070/savepoints/savepoint-bc0b2a-63cf5d5cccef8'.
Waiting for response...
Savepoint 'hdfs://xxx:50070/savepoints/savepoint-bc0b2a-63cf5d5cccef8' disposed.
```

从保存点启动应用程序

```
$ ./bin/flink run -s <savepointPath> [<options>] <jobJar> [<arguments>]
```

10.5 取消一个应用

```
$ ./bin/flink cancel <jobId>
```

取消的同时做保存点操作

```
$ ./bin/flink cancel -s [savepointPath] <jobId>
```

例如

```
$ ./bin/flink cancel -s \
hdfs://xxx:50070/savepoints d5fdaff43022954f5f02fc8f25ef855
 Cancelling job bc0b2ad61ecd4a615d92ce25390f61ad
 with savepoint to hdfs://xxx:50070/savepoints.
 Cancelled job bc0b2ad61ecd4a615d92ce25390f61ad.
 Savepoint stored in hdfs://xxx:50070/savepoints/savepoint-bc0b2a-d08de07fbb10.
```

10.6 扩容，改变并行度操作

```
$ ./bin/flink modify <jobId> -p <newParallelism>
```

例子

```
$ ./bin/flink modify bc0b2ad61ecd4a615d92ce25390f61ad -p 16
 Modify job bc0b2ad61ecd4a615d92ce25390f61ad.
 Rescaled job bc0b2ad61ecd4a615d92ce25390f61ad. Its new parallelism is 16.
```

第十一章 Flink CEP 简介

什么是复杂事件 CEP ?

一个或多个由简单事件构成的事件流通过一定的规则匹配，然后输出用户想得到的数据，满足规则的复杂事件。

特征

- 目标：从有序的简单事件流中发现一些高阶特征
- 输入：一个或多个由简单事件构成的事件流
- 处理：识别简单事件之间的内在联系，多个符合一定规则的简单事件构成复杂事件
- 输出：满足规则的复杂事件

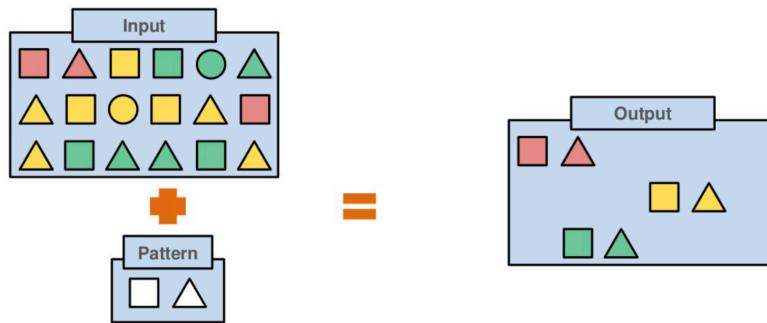


图 11.1: CEP

CEP 用于分析低延迟、频繁产生的不同来源的事件流。CEP 可以帮助在复杂的、不相关的事件流中找出有意义的模式和复杂的关系，以接近实时或准实时的获得通知并阻止一些行为。

CEP 支持在流上进行模式匹配，根据模式的条件不同，分为连续的条件或不连续的条件；模式的条件允许有时间的限制，当在条件范围内没有达到满足的条件时，会导致模式匹配超时。

看起来很简单，但是它有很多不同的功能：

- 输入的流数据，尽快产生结果
- 在 2 个 event 流上，基于时间进行聚合类的计算
- 提供实时/准实时的警告和通知
- 在多样的数据源中产生关联并分析模式
- 高吞吐、低延迟的处理

市场上有多种 CEP 的解决方案，例如 Spark、Samza、Beam 等，但他们都没有提供专门的 library 支持。但是 Flink 提供了专门的 CEP library。

Flink 为 CEP 提供了专门的 Flink CEP library，它包含如下组件：

- Event Stream
- pattern 定义
- pattern 检测

- 生成 Alert

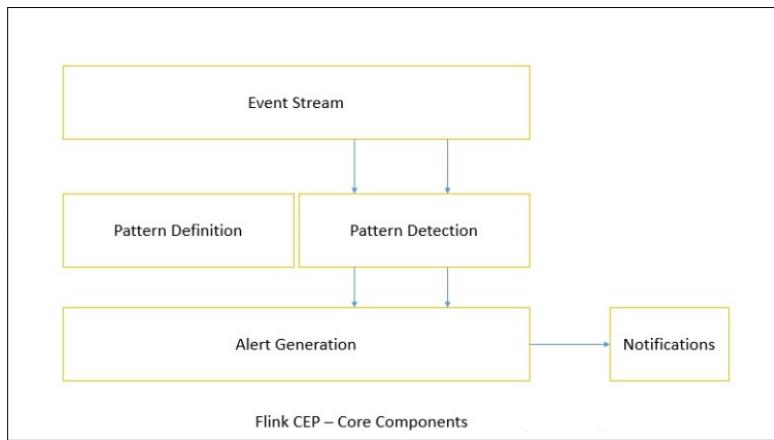


图 11.2: CEP

首先，开发人员要在 DataStream 流上定义出模式条件，之后 Flink CEP 引擎进行模式检测，必要时生成告警。

为了使用 Flink CEP，我们需要导入依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-cep_${scala.binary.version}</artifactId>
  <version>${flink.version}</version>
</dependency>
```

需求：5s 之内连续三次登录失败，报警

登录事件 POJO 类实现：

```
public class LoginEvent {
    public String userId;
    public String ipAddress;
    public String eventType;
    public Long eventTime;

    public LoginEvent(String userId, String ipAddress, String eventType, Long
        eventTime) {
        this.userId = userId;
        this.ipAddress = ipAddress;
        this.eventType = eventType;
        this.eventTime = eventTime;
    }

    public LoginEvent() {}

    @Override
    public String toString() {
```

```

        return "LoginEvent{" +
            "userId='" + userId + '\'' +
            ", ipAddress='" + ipAddress + '\'' +
            ", eventType='" + eventType + '\'' +
            ", eventTime=" + eventTime +
            '}';
    }
}

```

业务逻辑编写

```

public class LoginFailDetect {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        // 登录事件流
        KeyedStream<LoginEvent, String> stream = env
            .fromElements(
                new LoginEvent("user_1", "0.0.0.0", "fail", 2000L),
                new LoginEvent("user_1", "0.0.0.1", "fail", 3000L),
                new LoginEvent("user_1", "0.0.0.2", "fail", 4000L)
            )
            .assignTimestampsAndWatermarks(
                WatermarkStrategy.<LoginEvent>forMonotonousTimestamps()
                    .withTimestampAssignner(
                        new SerializableTimestampAssigner<LoginEvent>() {
                            @Override
                            public long extractTimestamp(LoginEvent loginEvent, long l) {
                                return loginEvent.eventTime;
                            }
                        }
                    )
            )
            .keyBy(r -> r.userId);

        // 模板定义
        Pattern<LoginEvent, LoginEvent> pattern = Pattern
            .<LoginEvent>begin("first")
            .where(new SimpleCondition<LoginEvent>() {
                @Override
                public boolean filter(LoginEvent loginEvent) throws Exception {
                    return loginEvent.eventType.equals("fail");
                }
            })
    }
}

```

```

        .next("second")
        .where(new SimpleCondition<LoginEvent>() {
            @Override
            public boolean filter(LoginEvent loginEvent) throws Exception {
                return loginEvent.eventType.equals("fail");
            }
        })
        .next("third")
        .where(new SimpleCondition<LoginEvent>() {
            @Override
            public boolean filter(LoginEvent loginEvent) throws Exception {
                return loginEvent.eventType.equals("fail");
            }
        })
        .within(Time.seconds(5));

    // 在流上使用模板来匹配
    PatternStream<LoginEvent> patternedStream = CEP.pattern(stream, pattern);

    // 将匹配到的事件选择出来，然后输出
    patternedStream
        .select(new PatternSelectFunction<LoginEvent, Tuple4<String, String,
        → String, String>>() {
            @Override
            public Tuple4<String, String, String, String> select(Map<String,
            → List<LoginEvent>> map) throws Exception {
                LoginEvent first = map.get("first").get(0);
                LoginEvent second = map.get("second").get(0);
                LoginEvent third = map.get("third").get(0);
                return Tuple4.of(first.userId, first.ipAddress, second.ipAddress,
                → third.ipAddress);
            }
        })
        .print();

    env.execute();
}
}

```

案例：订单超时检测

在电商平台中，最终创造收入和利润的是用户下单购买的环节；更具体一点，是用户真正完成支付动作的时候。用户下单的行为可以表明用户对商品的需求，但在现实中，并不是每次下单都会被用户立刻支付。当拖延一段时间后，用户支付的意愿会降低。所以为了让用户更有紧迫感从而提高支付转化率，同时也为了防范订单支付环节的安全风险，电商网站往往会对订单状态进行监控，设置一个失效时间（比如 15 分钟），如果下

单后一段时间仍未支付，订单就会被取消。

我们将会利用 CEP 库来实现这个功能。我们先将事件流按照订单号 orderId 分流，然后定义这样的一个事件模式：在 15 分钟内，事件“create”与“pay”严格紧邻：

```
Pattern<OrderEvent, OrderEvent> pattern = Pattern
    .<OrderEvent>begin("create")
    .where(new SimpleCondition<OrderEvent>() {
        @Override
        public boolean filter(OrderEvent value) throws Exception {
            return value.eventType.equals("create");
        }
    })
    .next("pay")
    .where(new SimpleCondition<OrderEvent>() {
        @Override
        public boolean filter(OrderEvent value) throws Exception {
            return value.eventType.equals("pay");
        }
    })
    .within(Time.seconds(5));
```

这样调用.select 方法时，就可以同时获取到匹配出的事件和超时未匹配的事件了。在 src/main-scala 下继续创建 OrderTimeout.scala 文件，新建一个单例对象。定义样例类 OrderEvent，这是输入的订单事件流；另外还有 OrderResult，这是输出显示的订单状态结果。由于没有现成的数据，我们还是用几条自定义的示例数据来做演示。

完整代码如下：

POJO 类实现

```
public class OrderEvent {
    public String orderId;
    public String eventType;
    public Long eventTime;

    public OrderEvent() {
    }

    public OrderEvent(String orderId, String eventType, Long eventTime) {
        this.orderId = orderId;
        this.eventType = eventType;
        this.eventTime = eventTime;
    }

    @Override
    public String toString() {
        return "OrderEvent{" +
    }}
```

```

        "orderId'" + orderId + '\'' + 
        ", eventType'" + eventType + '\'\' + 
        ", eventTime'" + eventTime + 
        '\'';
    }
}

```

检测程序如下：

```

public class OrderTimeoutDetect {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
        env.setParallelism(1);

        SingleOutputStreamOperator<OrderEvent> stream = env
            .fromElements(
                new OrderEvent("order_1", "create", 1000L),
                new OrderEvent("order_2", "create", 2000L),
                new OrderEvent("order_1", "pay", 3000L)
            )
            .assignTimestampsAndWatermarks(
                WatermarkStrategy.<OrderEvent>forMonotonousTimestamps()
                    .withTimestampAssigner(new
                        SerializableTimestampAssigner<OrderEvent>() {
                            @Override
                            public long extractTimestamp(OrderEvent element, long
                                recordTimestamp) {
                                return element.eventTime;
                            }
                        })
            );
    }

    Pattern<OrderEvent, OrderEvent> pattern = Pattern
        .<OrderEvent>begin("create")
        .where(new SimpleCondition<OrderEvent>() {
            @Override
            public boolean filter(OrderEvent value) throws Exception {
                return value.eventType.equals("create");
            }
        })
        .next("pay")
        .where(new SimpleCondition<OrderEvent>() {
            @Override
            public boolean filter(OrderEvent value) throws Exception {

```

```

        return value.eventType.equals("pay");
    }
}

.within(Time.seconds(5));

PatternStream<OrderEvent> patternStream = CEP.pattern(stream.keyBy(r ->
    r.orderId), pattern);

SingleOutputStreamOperator<String> result = patternStream
    .select(
        new OutputTag<String>("order-timeout") {
            },
        new PatternTimeoutFunction<OrderEvent, String>() {
            @Override
            public String timeout(Map<String, List<OrderEvent>> map, long l)
                throws Exception {
                return " 订单 ID 为 " + map.get("create").get(0).orderId + " 没
                    有支付! ";
            }
        },
        new PatternSelectFunction<OrderEvent, String>() {
            @Override
            public String select(Map<String, List<OrderEvent>> map) throws
                Exception {
                return " 订单 ID 为 " + map.get("pay").get(0).orderId + " 已经
                    支付! ";
            }
        }
    );
}

result.print();

result.getSideOutput(new OutputTag<String>("order-timeout") {}).print();

env.execute();
}
}

```

第十二章 Flink Table API & SQL

12.1 什么是 Table API 和 Flink SQL

Flink 本身是批流统一的处理框架，所以 Table API 和 SQL，就是批流统一的上层处理 API。目前功能尚未完善，处于活跃的开发阶段。

Table API 是一套内嵌在 Java 和 Scala 语言中的查询 API，它允许我们以非常直观的方式，组合来自一些关系运算符的查询（比如 select、filter 和 join）。而对于 Flink SQL，就是直接可以在代码中写 SQL，来实现一些查询（Query）操作。Flink 的 SQL 支持，基于实现了 SQL 标准的 Apache Calcite（Apache 开源 SQL 解析工具）。

无论输入是批输入还是流式输入，在这两套 API 中，指定的查询都具有相同的语义，得到相同的结果。

12.2 需要引入的依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table-api-java-bridge_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
</dependency>

<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table-planner-blink_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
</dependency>
```

内部实现上，部分 table 相关的代码是用 Scala 实现的。所以，下面的依赖也需要添加到你的程序里，不管是批式还是流式的程序：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-scala_${scala.binary.version}</artifactId>
    <version>${flink.version}</version>
</dependency>
```

扩展依赖

如果你想实现自定义格式来解析 Kafka 数据，或者自定义函数，下面的依赖就足够了，编译出来的 jar 文件可以直接给 SQL Client 使用：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table-common</artifactId>
    <version>${flink.version}</version>
</dependency>
```

当前，本模块包含以下可以扩展的接口：

- `SerializationSchemaFactory`
- `DeserializationSchemaFactory`
- `ScalarFunction`
- `TableFunction`
- `AggregateFunction`

12.3 概念与通用 API

12.3.1 Table API 和 SQL 程序的结构

所有用于批处理和流处理的 Table API 和 SQL 程序都遵循相同的模式。下面的代码示例展示了 Table API 和 SQL 程序的通用结构。

```
// create a TableEnvironment for specific planner batch or streaming
TableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

// create an input Table
tableEnv.executeSql("CREATE TEMPORARY TABLE table1 ... WITH ( 'connector' = ... )");
// register an output Table
tableEnv.executeSql("CREATE TEMPORARY TABLE outputTable ... WITH ( 'connector' = ...
→  )");

// create a Table object from a Table API query
Table table2 = tableEnv.from("table1").select(...);
// create a Table object from a SQL query
Table table3 = tableEnv.sqlQuery("SELECT ... FROM table1 ... ");

// emit a Table API result Table to a TableSink, same for SQL result
TableResult tableResult = table2.executeInsert("outputTable");
tableResult...
```

12.3.2 创建 TableEnvironment

`TableEnvironment` 是 Table API 和 SQL 的核心概念。它负责：

- 在内部的 catalog 中注册 Table
- 注册外部的 catalog

- 加载可插拔模块
- 执行 SQL 查询
- 注册自定义函数（scalar、table 或 aggregation）
- 将 DataStream 或 DataSet 转换成 Table
- 持有对 ExecutionEnvironment 或 StreamExecutionEnvironment 的引用

Table 总是与特定的 TableEnvironment 绑定。不能在同一条查询中使用不同 TableEnvironment 中的表，例如，对它们进行 join 或 union 操作。

TableEnvironment 可以通过静态方法 BatchTableEnvironment.create() 或者 StreamTableEnvironment.create() 在 StreamExecutionEnvironment 或者 ExecutionEnvironment 中创建，TableConfig 是可选项。TableConfig 可用于配置 TableEnvironment 或定制的查询优化和转换过程。

```
// ****
// FLINK STREAMING QUERY
// ****
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

EnvironmentSettings fsSettings =
    EnvironmentSettings.newInstance().useOldPlanner().inStreamingMode().build();
StreamExecutionEnvironment fsEnv =
    StreamExecutionEnvironment.getExecutionEnvironment();
StreamTableEnvironment fsTableEnv = StreamTableEnvironment.create(fsEnv, fsSettings);
// or TableEnvironment fsTableEnv = TableEnvironment.create(fsSettings);

// ****
// FLINK BATCH QUERY
// ****
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.table.api.bridge.java.BatchTableEnvironment;

ExecutionEnvironment fbEnv = ExecutionEnvironment.getExecutionEnvironment();
BatchTableEnvironment fbTableEnv = BatchTableEnvironment.create(fbEnv);

// ****
// BLINK STREAMING QUERY
// ****
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

StreamExecutionEnvironment bsEnv =
    StreamExecutionEnvironment.getExecutionEnvironment();
```

```

EnvironmentSettings bsSettings =
    EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();
StreamTableEnvironment bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings);
// or TableEnvironment bsTableEnv = TableEnvironment.create(bsSettings);

// *****
// BLINK BATCH QUERY
// *****

import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.TableEnvironment;

EnvironmentSettings bbSettings =
    EnvironmentSettings.newInstance().useBlinkPlanner().inBatchMode().build();
TableEnvironment bbTableEnv = TableEnvironment.create(bbSettings);

```

12.3.3 在 Catalog 中创建表

TableEnvironment 维护着一个由标识符 (identifier) 创建的表 catalog 的映射。标识符由三个部分组成：catalog 名称、数据库名称以及对象名称。如果 catalog 或者数据库没有指明，就会使用当前默认值。

Table 可以是虚拟的（视图 VIEWS）也可以是常规的（表 TABLES）。视图 VIEWS 可以从已经存在的 Table 中创建，一般是 Table API 或者 SQL 的查询结果。表 TABLES 描述的是外部数据，例如文件、数据库表或者消息队列。

临时表（Temporary Table）和永久表（Permanent Table）

表可以是临时的，并与单个 Flink 会话（session）的生命周期相关，也可以是永久的，并且在多个 Flink 会话和群集（cluster）中可见。

永久表需要 catalog（例如 Hive Metastore）以维护表的元数据。一旦永久表被创建，它将对任何连接到 catalog 的 Flink 会话可见且持续存在，直至被明确删除。

另一方面，临时表通常保存于内存中并且仅在创建它们的 Flink 会话持续期间存在。这些表对于其它会话是不可见的。它们不与任何 catalog 或者数据库绑定但可以在一个命名空间（namespace）中创建。即使它们对应的数据库被删除，临时表也不会被删除。

创建表

虚拟表

在 SQL 的术语中，Table API 的对象对应于视图（虚拟表）。它封装了一个逻辑查询计划。它可以通过以下方法在 catalog 中创建：

```

// get a TableEnvironment
TableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

// table is the result of a simple projection query
Table projTable = tableEnv.from("X").select(...);

```

```
// register the Table projTable as table "projectedTable"
tableEnv.createTemporaryView("projectedTable", projTable);
```

Connector Tables

另外一个方式去创建 TABLE 是通过 connector 声明。Connector 描述了存储表数据的外部系统。存储系统例如 Apache Kafka 或者常规的文件系统都可以通过这种方式来声明。

```
tableEnvironment
    .connect(...)
    .withFormat(...)
    .withSchema(...)
    .inAppendMode()
    .createTemporaryTable("MyTable")
```

扩展表标识符

表总是通过三元标识符注册，包括 catalog 名、数据库名和表名。

用户可以指定一个 catalog 和数据库作为“当前 catalog”和“当前数据库”。有了这些，那么刚刚提到的三元标识符的前两个部分就可以被省略了。如果前两部分的标识符没有指定，那么会使用当前的 catalog 和当前数据库。用户也可以通过 Table API 或 SQL 切换当前的 catalog 和当前的数据库。

标识符遵循 SQL 标准，因此使用时需要用反引号 (`) 进行转义。

```
TableEnvironment tEnv = ...;
tEnv.useCatalog("custom_catalog");
tEnv.useDatabase("custom_database");

Table table = ...;

// register the view named 'exampleView' in the catalog named 'custom_catalog'
// in the database named 'custom_database'
tableEnv.createTemporaryView("exampleView", table);

// register the view named 'exampleView' in the catalog named 'custom_catalog'
// in the database named 'other_database'
tableEnv.createTemporaryView("other_database.exampleView", table);

// register the view named 'example.View' in the catalog named 'custom_catalog'
// in the database named 'custom_database'
tableEnv.createTemporaryView("`example.View`", table);

// register the view named 'exampleView' in the catalog named 'other_catalog'
```

```
// in the database named 'other_database'
tableEnv.createTemporaryView("other_catalog.other_database.exampleView", table);
```

12.3.4 查询表

Table API

Table API 是关于 Scala 和 Java 的集成语言式查询 API。与 SQL 相反，Table API 的查询不是由字符串指定，而是在宿主语言中逐步构建。

Table API 是基于 Table 类的，该类表示一个表（流或批处理），并提供使用关系操作的方法。这些方法返回一个新的 Table 对象，该对象表示对输入 Table 进行关系操作的结果。一些关系操作由多个方法调用组成，例如 table.groupBy(...).select()，其中 groupBy(...) 指定 table 的分组，而 select(...) 在 table 分组上的投影。

文档 Table API 说明了所有流处理和批处理表支持的 Table API 算子。

以下示例展示了一个简单的 Table API 聚合查询：

```
// get a TableEnvironment
TableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

// register Orders table

// scan registered Orders table
Table orders = tableEnv.from("Orders");
// compute revenue for all customers from France
Table revenue = orders
    .filter($"cCountry".isEqual("FRANCE"))
    .groupBy($"cID", $"cName")
    .select($"cID", $"cName", $"revenue".sum().as("revSum"));

// emit or convert Table
// execute query
```

SQL

Flink SQL 是基于实现了 SQL 标准的 Apache Calcite 的。SQL 查询由常规字符串指定。

文档 SQL 描述了 Flink 对流处理和批处理表的 SQL 支持。

下面的示例演示了如何指定查询并将结果作为 Table 对象返回。

```
// get a TableEnvironment
TableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

// register Orders table
```

```
// compute revenue for all customers from France
Table revenue = tableEnv.sqlQuery(
    "SELECT cID, cName, SUM(revenue) AS revSum " +
    "FROM Orders " +
    "WHERE cCountry = 'FRANCE' " +
    "GROUP BY cID, cName"
);

// emit or convert Table
// execute query
```

如下的示例展示了如何指定一个更新查询，将查询的结果插入到已注册的表中。

```
// get a TableEnvironment
TableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

// register "Orders" table
// register "RevenueFrance" output table

// compute revenue for all customers from France and emit to "RevenueFrance"
tableEnv.executeSql(
    "INSERT INTO RevenueFrance " +
    "SELECT cID, cName, SUM(revenue) AS revSum " +
    "FROM Orders " +
    "WHERE cCountry = 'FRANCE' " +
    "GROUP BY cID, cName"
);
```

混用 Table API 和 SQL

Table API 和 SQL 查询的混用非常简单因为它们都返回 Table 对象：

- 可以在 SQL 查询返回的 Table 对象上定义 Table API 查询。
- 在 TableEnvironment 中注册的结果表可以在 SQL 查询的 FROM 子句中引用，通过这种方法就可以在 Table API 查询的结果上定义 SQL 查询。

12.3.5 输出表

Table 通过写入 TableSink 输出。TableSink 是一个通用接口，用于支持多种文件格式（如 CSV、Apache Parquet、Apache Avro）、存储系统（如 JDBC、Apache HBase、Apache Cassandra、Elasticsearch）或消息队列系统（如 Apache Kafka、RabbitMQ）。

批处理 Table 只能写入 BatchTableSink，而流处理 Table 需要指定写入 AppendStreamTableSink，RetractStreamTableSink 或者 UpsertStreamTableSink。

方法 Table.executeInsert(String tableName) 将 Table 发送至已注册的 TableSink。该方法通过名称在 catalog 中查找 TableSink 并确认 Table schema 和 TableSink schema 一致。

下面的示例演示如何输出 Table：

```
// get a TableEnvironment
TableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

// create an output Table
final Schema schema = new Schema()
    .field("a", DataTypes.INT())
    .field("b", DataTypes.STRING())
    .field("c", DataTypes.BIGINT());

tableEnv.connect(new FileSystem().path("/path/to/file"))
    .withFormat(new Csv().fieldDelimiter('|').deriveSchema())
    .withSchema(schema)
    .createTemporaryTable("CsvSinkTable");

// compute a result Table using Table API operators and/or SQL queries
Table result = ...
// emit the result Table to the registered TableSink
result.executeInsert("CsvSinkTable");
```

12.3.6 与 DataSet 和 DataStream API 结合

通过 DataSet 或 DataStream 创建视图

在 TableEnvironment 中可以将 DataStream 或 DataSet 注册成视图。结果视图的 schema 取决于注册的 DataStream 或 DataSet 的数据类型。



笔记 通过 DataStream 或 DataSet 创建的视图只能注册成临时视图。

```
// get StreamTableEnvironment
// registration of a DataSet in a BatchTableEnvironment is equivalent
StreamTableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

DataStream<Tuple2<Long, String>> stream = ...

// register the DataStream as View "myTable" with fields "f0", "f1"
tableEnv.createTemporaryView("myTable", stream);

// register the DataStream as View "myTable2" with fields "myLong", "myString"
tableEnv.createTemporaryView("myTable2", stream, $("myLong"), $("myString"));
```

将 DataStream 或 DataSet 转换成表

与在 TableEnvironment 注册 DataStream 或 DataSet 不同，DataStream 和 DataSet 还可以转换成 Table。如果你想在 Table API 的查询中使用表，这将非常便捷。

```
// get StreamTableEnvironment
// registration of a DataSet in a BatchTableEnvironment is equivalent
StreamTableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

DataStream<Tuple2<Long, String>> stream = ...

// Convert the DataStream into a Table with default fields "f0", "f1"
Table table1 = tableEnv.fromDataStream(stream);

// Convert the DataStream into a Table with fields "myLong", "myString"
Table table2 = tableEnv.fromDataStream(stream, $("myLong"), $("myString"));
```

将表转换成 DataStream 或 DataSet

Table 可以被转换成 DataStream 或 DataSet。通过这种方式，定制的 DataSet 或 DataStream 程序就可以在 Table API 或者 SQL 的查询结果上运行了。

将 Table 转换为 DataStream 或者 DataSet 时，你需要指定生成的 DataStream 或者 DataSet 的数据类型，即，Table 的每行数据要转换成的数据类型。通常最方便的选择是转换成 Row。以下列表概述了不同选项的功能：

- Row: 字段按位置映射，字段数量任意，支持 null 值，无类型安全（type-safe）检查。
- POJO: 字段按名称映射（POJO 必须按 Table 中字段名称命名），字段数量任意，支持 null 值，无类型安全检查。
- Case Class: 字段按位置映射，不支持 null 值，有类型安全检查。
- Tuple: 字段按位置映射，字段数量少于 22（Scala）或者 25（Java），不支持 null 值，无类型安全检查。
- Atomic Type: Table 必须有一个字段，不支持 null 值，有类型安全检查。

将表转换成 DataStream

流式查询（streaming query）的结果表会动态更新，即，当新纪录到达查询的输入流时，查询结果会改变。因此，像这样将动态查询结果转换成 DataStream 需要对表的更新方式进行编码。

将 Table 转换为 DataStream 有两种模式：

- Append Mode: 仅当动态 Table 仅通过 INSERT 更改进行修改时，才可以使用此模式，即，它仅是追加操作，并且之前输出的结果永远不会更新。
- Retract Mode: 任何情形都可以使用此模式。它使用 boolean 值对 INSERT 和 DELETE 操作的数据进行标记。

```
// get StreamTableEnvironment.
StreamTableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

// Table with two fields (String name, Integer age)
Table table = ...
```

```
// convert the Table into an append DataStream of Row by specifying the class
DataStream<Row> dsRow = tableEnv.toAppendStream(table, Row.class);

// convert the Table into an append DataStream of Tuple2<String, Integer>
// via a TypeInformation
TupleTypeInfo<Tuple2<String, Integer>> tupleType = new TupleTypeInfo<>(
    Types.STRING(),
    Types.INT());
DataStream<Tuple2<String, Integer>> dsTuple =
    tableEnv.toAppendStream(table, tupleType);

// convert the Table into a retract DataStream of Row.
// A retract stream of type X is a DataStream<Tuple2<Boolean, X>>.
// The boolean field indicates the type of the change.
// True is INSERT, false is DELETE.
DataStream<Tuple2<Boolean, Row>> retractStream =
    tableEnv.toRetractStream(table, Row.class);
```

将表转换成 *DataSet*

将 Table 转换成 DataSet 的过程如下：

```
// get BatchTableEnvironment
BatchTableEnvironment tableEnv = BatchTableEnvironment.create(env);

// Table with two fields (String name, Integer age)
Table table = ...

// convert the Table into a DataSet of Row by specifying a class
DataSet<Row> dsRow = tableEnv.toDataSet(table, Row.class);

// convert the Table into a DataSet of Tuple2<String, Integer> via a TypeInformation
TupleTypeInfo<Tuple2<String, Integer>> tupleType = new TupleTypeInfo<>(
    Types.STRING(),
    Types.INT());
DataSet<Tuple2<String, Integer>> dsTuple =
    tableEnv.toDataSet(table, tupleType);
```

数据类型到 Table Schema 的映射

Flink 的 DataStream 和 DataSet APIs 支持多样的数据类型。例如 Tuple (Scala 内置以及 Flink Java tuple)、POJO 类型、Scala case class 类型以及 Flink 的 Row 类型等允许嵌套且有多个可在表的表达式中访问的字段的复合数据类型。其他类型被视为原子类型。下面，我们讨论 Table API 如何将这些数据类型类型转换为内部 row 表示形式，并提供将 DataStream 转换成 Table 的样例。

数据类型到 table schema 的映射有两种方式：基于字段位置或基于字段名称。

基于位置映射

基于位置的映射可在保持字段顺序的同时为字段提供更有意义的名称。这种映射方式可用于具有特定的字段顺序的复合数据类型以及原子类型。如 tuple、row 以及 case class 这些复合数据类型都有这样的字段顺序。然而，POJO 类型的字段则必须通过名称映射。可以将字段投影出来，但不能使用 as 重命名。

定义基于位置的映射时，输入数据类型中一定不能存在指定的名称，否则 API 会假定应该基于字段名称进行映射。如果未指定任何字段名称，则使用默认的字段名称和复合数据类型的字段顺序，或者使用 f0 表示原子类型。

```
// get a StreamTableEnvironment, works for BatchTableEnvironment equivalently
StreamTableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section;

DataStream<Tuple2<Long, Integer>> stream = ...

// convert DataStream into Table with default field names "f0" and "f1"
Table table = tableEnv.fromDataStream(stream);

// convert DataStream into Table with field "myLong" only
Table table = tableEnv.fromDataStream(stream, $("myLong"));

// convert DataStream into Table with field names "myLong" and "myInt"
Table table = tableEnv.fromDataStream(stream, $("myLong"), $("myInt"));
```

基于名称的映射

基于名称的映射适用于任何数据类型包括 POJO 类型。这是定义 table schema 映射最灵活的方式。映射中的所有字段均按名称引用，并且可以通过 as 重命名。字段可以被重新排序和映射。

若果没有指定任何字段名称，则使用默认的字段名称和复合数据类型的字段顺序，或者使用 f0 表示原子类型。

```
// get a StreamTableEnvironment, works for BatchTableEnvironment equivalently
StreamTableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

DataStream<Tuple2<Long, Integer>> stream = ...

// convert DataStream into Table with default field names "f0" and "f1"
Table table = tableEnv.fromDataStream(stream);

// convert DataStream into Table with field "f1" only
Table table = tableEnv.fromDataStream(stream, $("f1"));

// convert DataStream into Table with swapped fields
Table table = tableEnv.fromDataStream(stream, $("f1"), $("f0"));

// convert DataStream into Table with swapped fields and field names "myInt" and
↪ "myLong"
```

```
Table table = tableEnv.fromDataStream(stream, $("f1").as("myInt"),
    $("f0").as("myLong"));
```

原子类型

Flink 将基础数据类型（Integer、Double、String）或者通用数据类型（不可再拆分的数据类型）视为原子类型。原子类型的 DataStream 或者 DataSet 会被转换成只有一条属性的 Table。属性的数据类型可以由原子类型推断出，还可以重新命名属性。

```
// get a StreamTableEnvironment, works for BatchTableEnvironment equivalently
StreamTableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

DataStream<Long> stream = ...

// convert DataStream into Table with default field name "f0"
Table table = tableEnv.fromDataStream(stream);

// convert DataStream into Table with field name "myLong"
Table table = tableEnv.fromDataStream(stream, $("myLong"));
```

Tuple 类型

Flink 支持 Scala 的内置 tuple 类型并给 Java 提供自己的 tuple 类型。两种 tuple 的 DataStream 和 DataSet 都能被转换成表。可以通过提供所有字段名称来重命名字段（基于位置映射）。如果没有指明任何字段名称，则会使用默认的字段名称。如果引用了原始字段名称（对于 Flink tuple 为 f0、f1 ……），则 API 会假定映射是基于名称的而不是基于位置的。基于名称的映射可以通过 as 对字段和投影进行重新排序。

```
// get a StreamTableEnvironment, works for BatchTableEnvironment equivalently
StreamTableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

DataStream<Tuple2<Long, String>> stream = ...

// convert DataStream into Table with default field names "f0", "f1"
Table table = tableEnv.fromDataStream(stream);

// convert DataStream into Table with renamed field names "myLong", "myString"
// (position-based)
Table table = tableEnv.fromDataStream(stream, $("myLong"), $("myString"));

// convert DataStream into Table with reordered fields "f1", "f0" (name-based)
Table table = tableEnv.fromDataStream(stream, $("f1"), $("f0"));

// convert DataStream into Table with projected field "f1" (name-based)
Table table = tableEnv.fromDataStream(stream, $("f1"));
```

```
// convert DataStream into Table with reordered and aliased fields "myString", "myLong"
↪ (name-based)

Table table = tableEnv.fromDataStream(stream, $("f1").as("myString"),
↪ $("f0").as("myLong"));
```

POJO 类型

Flink 支持 POJO 类型作为复合类型。确定 POJO 类型的规则记录在这里。

在不指定字段名称的情况下将 POJO 类型的 DataStream 或 DataSet 转换成 Table 时，将使用原始 POJO 类型字段的名称。名称映射需要原始名称，并且不能按位置进行。字段可以使用别名（带有 as 关键字）来重命名，重新排序和投影。

```
// get a StreamTableEnvironment, works for BatchTableEnvironment equivalently
StreamTableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

// Person is a POJO with fields "name" and "age"
DataStream<Person> stream = ...

// convert DataStream into Table with default field names "age", "name" (fields are
↪ ordered by name!)
Table table = tableEnv.fromDataStream(stream);

// convert DataStream into Table with renamed fields "myAge", "myName" (name-based)
Table table = tableEnv.fromDataStream(stream, $("age").as("myAge"),
↪ $("name").as("myName"));

// convert DataStream into Table with projected field "name" (name-based)
Table table = tableEnv.fromDataStream(stream, $("name"));

// convert DataStream into Table with projected and renamed field "myName"
↪ (name-based)
Table table = tableEnv.fromDataStream(stream, $("name").as("myName"));
```

Row 类型

Row 类型支持任意数量的字段以及具有 null 值的字段。字段名称可以通过 RowTypeInfo 指定，也可以在将 Row 的 DataStream 或 DataSet 转换为 Table 时指定。Row 类型的字段映射支持基于名称和基于位置两种方式。字段可以通过提供所有字段的名称的方式重命名（基于位置映射）或者分别选择进行投影/排序/重命名（基于名称映射）。

```
// get a StreamTableEnvironment, works for BatchTableEnvironment equivalently
StreamTableEnvironment tableEnv = ...; // see "Create a TableEnvironment" section

// DataStream of Row with two fields "name" and "age" specified in `RowTypeInfo`
DataStream<Row> stream = ...
```

```
// convert DataStream into Table with default field names "name", "age"
Table table = tableEnv.fromDataStream(stream);

// convert DataStream into Table with renamed field names "myName", "myAge"
// (position-based)
Table table = tableEnv.fromDataStream(stream, $("myName"), $("myAge"));

// convert DataStream into Table with renamed fields "myName", "myAge" (name-based)
Table table = tableEnv.fromDataStream(stream, $("name").as("myName"),
    $("age").as("myAge"));

// convert DataStream into Table with projected field "name" (name-based)
Table table = tableEnv.fromDataStream(stream, $("name"));

// convert DataStream into Table with projected and renamed field "myName"
// (name-based)
Table table = tableEnv.fromDataStream(stream, $("name").as("myName"));
```

12.4 流式概念

Flink 的 Table API 和 SQL 是流批统一的 API。这意味着 Table API & SQL 在无论有限的批式输入还是无限的流式输入下，都具有相同的语义。因为传统的关系代数以及 SQL 最开始都是为了批式处理而设计的，关系型查询在流式场景下不如在批式场景下容易懂。

12.4.1 动态表 (Dynamic Table)

SQL 和关系代数在设计时并未考虑流数据。因此，在关系代数 (和 SQL) 之间几乎没有概念上的差异。

本文会讨论这种差异，并介绍 Flink 如何在无界数据集上实现与数据库引擎在有界数据上的处理具有相同的语义。

DataStream 上的关系查询

下表比较了传统的关系代数和流处理与输入数据、执行和输出结果的关系。

表 12.1: 比较

关系代数 / SQL	流处理
关系 (或表) 是有界 (多) 元组集合	流是一个无限元组序列
对批数据 (例如关系数据库中的表) 执行的查询可以访问完整的输入数据	流式查询在启动时不能访问所有数据，必须“等待”数据流入
批处理查询在产生固定大小的结果后终止	流查询不断地根据接收到的记录更新其结果，并且始终不会结束

尽管存在这些差异，但是使用关系查询和 SQL 处理流并不是不可能的。高级关系数据库系统提供了一个称为物化视图 (Materialized Views) 的特性。物化视图被定义为一条

SQL 查询，就像常规的虚拟视图一样。与虚拟视图相反，物化视图缓存查询的结果，因此在访问视图时不需要对查询进行计算。缓存的一个常见难题是防止缓存为过期的结果提供服务。当其定义查询的基表被修改时，物化视图将过期。即时视图维护 (Eager View Maintenance) 是一种一旦更新了物化视图的基表就立即更新视图的技术。

如果我们考虑以下问题，那么即时视图维护和流上的 SQL 查询之间的联系就会变得显而易见：

- 数据库表是 INSERT、UPDATE 和 DELETE DML 语句的 stream 的结果，通常称为 changelog stream。
- 物化视图被定义为一条 SQL 查询。为了更新视图，查询不断地处理视图的基本关系的 changelog 流。
- 物化视图是流式 SQL 查询的结果。

了解了这些要点之后，我们将在下一节中介绍动态表 (Dynamic tables) 的概念。

动态表 & 连续查询 (Continuous Query)

动态表是 Flink 的支持流数据的 Table API 和 SQL 的核心概念。与表示批处理数据的静态表不同，动态表是随时间变化的。可以像查询静态批处理表一样查询它们。查询动态表将生成一个连续查询。一个连续查询永远不会终止，结果会生成一个动态表。查询不断更新其 (动态) 结果表，以反映其 (动态) 输入表上的更改。本质上，动态表上的连续查询非常类似于定义物化视图的查询。

需要注意的是，连续查询的结果在语义上总是等价于以批处理模式在输入表快照上执行的相同查询的结果。

下图显示了流、动态表和连续查询之间的关系：



图 12.1: 动态表

1. 将流转换为动态表。
2. 在动态表上计算一个连续查询，生成一个新的动态表。
3. 生成的动态表被转换回流。



笔记 动态表首先是一个逻辑概念。在查询执行期间不一定 (完全) 物化动态表。

在下面，我们将解释动态表和连续查询的概念，并使用具有以下模式的单击事件流：

```
[
  user: VARCHAR,      // 用户名
  cTime: TIMESTAMP,   // 访问 URL 的时间
  url:  VARCHAR       // 用户访问的 URL
]
```

在流上定义表

为了使用关系查询处理流，必须将其转换成 Table。从概念上讲，流的每条记录都被

解释为对结果表的 INSERT 操作。本质上我们正在从一个 INSERT-only 的 changelog 流构建表。

下图显示了单击事件流(左侧)如何转换为表(右侧)。当插入更多的单击流记录时,结果表将不断增长。

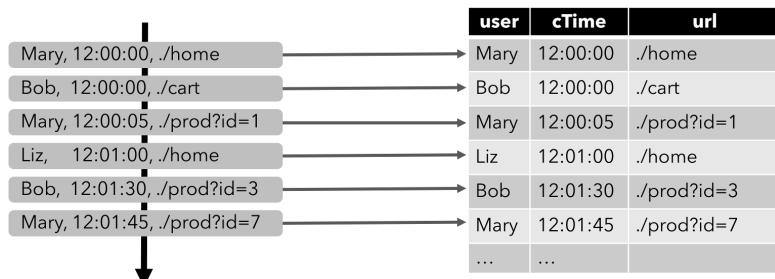


图 12.2: 流转换成表

笔记 在流上定义的表在内部没有物化。

连续查询

在动态表上计算一个连续查询，并生成一个新的动态表。与批处理查询不同，连续查询从不终止，并根据其输入表上的更新更新其结果表。在任何时候，连续查询的结果在语义上与以批处理模式在输入表快照上执行的相同查询的结果相同。

在接下来的代码中，我们将展示 clicks 表上的两个示例查询，这个表是在点击事件流上定义的。

第一个查询是一个简单的 GROUP-BY COUNT 聚合查询。它基于 user 字段对 clicks 表进行分组，并统计访问的 URL 的数量。下面的图显示了当 clicks 表被附加的行更新时，查询是如何被评估的。

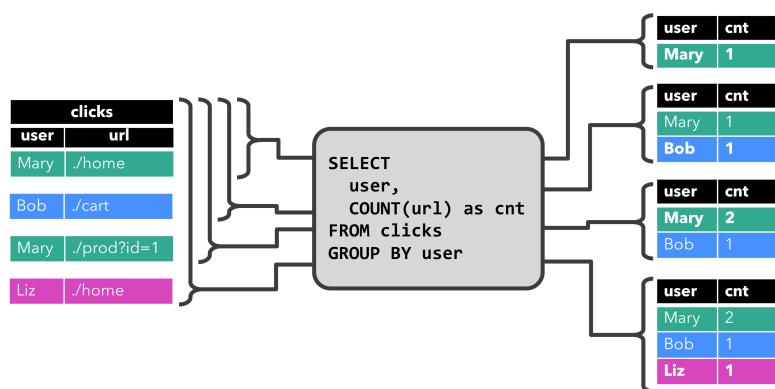


图 12.3: 查询

当查询开始，clicks 表(左侧)是空的。当第一行数据被插入到 clicks 表时，查询开始计算结果表。第一行数据 [Mary,./home] 插入后，结果表(右侧，上部)由一行 [Mary, 1] 组成。当第二行 [Bob, ./cart] 插入到 clicks 表时，查询会更新结果表并插入了一行新数据 [Bob, 1]。第三行 [Mary, ./prod?id=1] 将产生已计算的结果行的更新，[Mary, 1] 更新成 [Mary, 2]。最后，当第四行数据加入 clicks 表时，查询将第三行 [Liz, 1] 插入到结果表中。

第二条查询与第一条类似，但是除了用户属性之外，还将 clicks 分组至每小时滚动

窗口中，然后计算 url 数量（基于时间的计算，例如基于特定时间属性的窗口，后面会讨论）。同样，该图显示了不同时间点的输入和输出，以可视化动态表的变化特性。

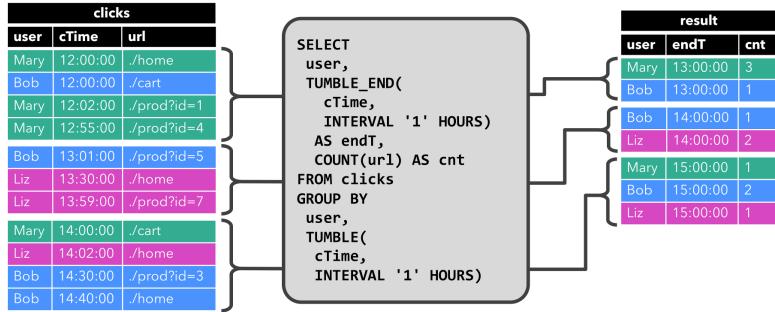


图 12.4: 查询

与前面一样，左边显示了输入表 `clicks`。查询每小时持续计算结果并更新结果表。`clicks` 表包含四行带有时间戳 (cTime) 的数据，时间戳在 12:00:00 和 12:59:59 之间。查询从这个输入计算出两个结果行 (每个 user 一个)，并将它们附加到结果表中。对于 13:00:00 和 13:59:59 之间的下一个窗口，`clicks` 表包含三行，这将导致另外两行被追加到结果表。随着时间的推移，更多的行被添加到 `click` 中，结果表将被更新。

更新和追加查询

虽然这两个示例查询看起来非常相似 (都计算分组计数聚合)，但它们在一个重要方面不同：

- 第一个查询更新先前输出的结果，即定义结果表的 changelog 流包含 `INSERT` 和 `UPDATE` 操作。
- 第二个查询只附加到结果表，即结果表的 changelog 流只包含 `INSERT` 操作。

一个查询是产生一个只追加的表还是一个更新的表有一些含义：

- 产生更新更改的查询通常必须维护更多的状态 (请参阅以下部分)。
- 将 `append-only` 的表转换为流与将已更新的表转换为流是不同的。

表到流的转换

动态表可以像普通数据库表一样通过 `INSERT`、`UPDATE` 和 `DELETE` 来不断修改。它可能是一个只有一行、不断更新的表，也可能是一个 `insert-only` 的表，没有 `UPDATE` 和 `DELETE` 修改，或者介于两者之间的其他表。

在将动态表转换为流或将其写入外部系统时，需要对这些更改进行编码。Flink 的 Table API 和 SQL 支持三种方式来编码一个动态表的变化：

- Append-only 流：仅通过 `INSERT` 操作修改的动态表可以通过输出插入的行转换为流。
- Retract 流：retract 流包含两种类型的 message：`add messages` 和 `retract messages`。通过将 `INSERT` 操作编码为 `add message`、将 `DELETE` 操作编码为 `retract message`、将 `UPDATE` 操作编码为更新 (先前) 行的 `retract message` 和更新 (新) 行的 `add message`，将动态表转换为 retract 流。下图显示了将动态表转换为 retract 流的过程。
- Upsert 流：upsert 流包含两种类型的 message：`upsert messages` 和 `delete messages`。转换为 upsert 流的动态表需要 (可能是组合的) 唯一键。通过将 `INSERT` 和 `UPDATE` 操

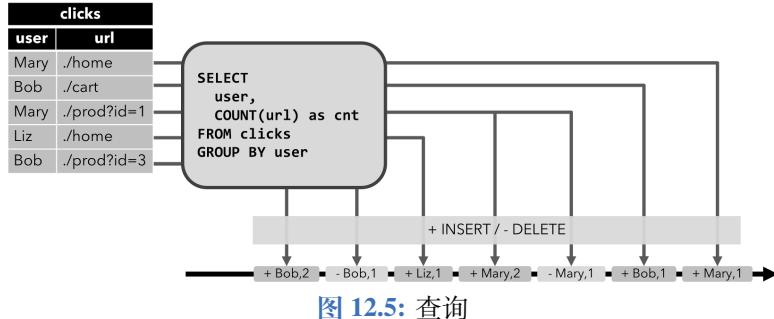


图 12.5: 查询

将编码为 upsert message，将 DELETE 操作编码为 delete message，将具有唯一键的动态表转换为流。消费流的算子需要知道唯一键的属性，以便正确地应用 message。与 retract 流的主要区别在于 UPDATE 操作是用单个 message 编码的，因此效率更高。下图显示了将动态表转换为 upsert 流的过程。

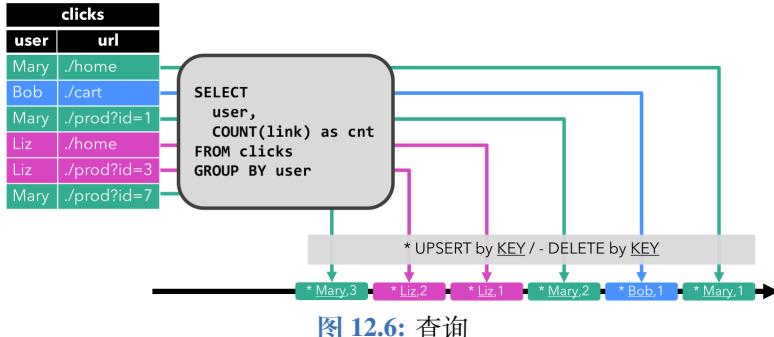


图 12.6: 查询



笔记 在将动态表转换为 DataStream 时，只支持 append 流和 retract 流。

12.4.2 时间属性

Flink 可以基于几种不同的时间概念来处理数据。

- 处理时间指的是执行具体操作时的机器时间（也称作“挂钟时间”）
- 事件时间指的是数据本身携带的时间。这个时间是在事件产生时的时间。
- 摄入时间指的是数据进入 Flink 的时间；在系统内部，会把它当做事件时间来处理。

时间属性介绍

像窗口（在 Table API 和 SQL）这种基于时间的操作，需要有时间信息。因此，Table API 中的表就需要提供逻辑时间属性来表示时间，以及支持时间相关的操作。

每种类型的表都可以有时间属性，可以在用 CREATE TABLE DDL 创建表的时候指定、也可以在 DataStream 中指定、也可以在定义 TableSource 时指定。一旦时间属性定义好，它就可以像普通列一样使用，也可以在时间相关的操作中使用。

只要时间属性没有被修改，而是简单地从一个表传递到另一个表，它就仍然是一个有效的时间属性。时间属性可以像普通的时间戳的列一样被使用和计算。一旦时间属性被用在了计算中，它就会被物化，进而变成一个普通的时间戳。普通的时间戳是无法跟 Flink 的时间以及 watermark 等一起使用的，所以普通的时间戳就无法用在时间相关的操作中。

处理时间

处理时间是基于机器的本地时间来处理数据，它是最简单的一种时间概念，但是它不能提供确定性。它既不需要从数据里获取时间，也不需要生成 watermark。

共有三种方法可以定义处理时间。

在创建表的 *DDL* 中定义

处理时间属性可以在创建表的 DDL 中用计算列的方式定义，用 PROCTIME() 就可以定义处理时间。

```
CREATE TABLE user_actions (
    user_name STRING,
    data STRING,
    user_action_time AS PROCTIME() -- 声明一个额外的列作为处理时间属性
) WITH (
    ...
);

SELECT TUMBLE_START(user_action_time, INTERVAL '10' MINUTE), COUNT(DISTINCT user_name)
FROM user_actions
GROUP BY TUMBLE(user_action_time, INTERVAL '10' MINUTE);
```

在 *DataStream* 到 *Table* 转换时定义

处理时间属性可以在 schema 定义的时候用.proctime 后缀来定义。时间属性一定不能定义在一个已有字段上，所以它只能定义在 schema 定义的最后。

```
DataStream<Tuple2<String, String>> stream = ...;

// 声明一个额外的字段作为时间属性字段
Table table = tEnv.fromDataStream(stream, $("user_name"), $("data"),
    $("user_action_time").proctime());

WindowedTable windowedTable = table.window(
    Tumble.over(lit(10).minutes())
        .on($("user_action_time"))
        .as("userActionWindow"));
```

事件时间

事件时间允许程序按照数据中包含的时间来处理，这样可以在有乱序或者晚到的数据的情况下产生一致的处理结果。它可以保证从外部存储读取数据后产生可以复现（replayable）的结果。

除此之外，事件时间可以让程序在流式和批式作业中使用同样的语法。在流式程序中的事件时间属性，在批式程序中就是一个正常的时间字段。

为了能够处理乱序的事件，并且区分正常到达和晚到的事件，Flink 需要从事件中获取事件时间并且产生 watermark (watermarks)。

事件时间属性也有类似于处理时间的三种定义方式：在 DDL 中定义、在 DataStream 到 Table 转换时定义、用 TableSource 定义。

在 DDL 中定义

事件时间属性可以用 WATERMARK 语句在 CREATE TABLE DDL 中进行定义。WATERMARK 语句在一个已有字段上定义一个 watermark 生成表达式，同时标记这个已有字段为时间属性字段。

```
CREATE TABLE user_actions (
    user_name STRING,
    data STRING,
    user_action_time TIMESTAMP(3),
    -- 声明 user_action_time 是事件时间属性，并且用 延迟 5 秒的策略来生成 watermark
    WATERMARK FOR user_action_time AS user_action_time - INTERVAL '5' SECOND
) WITH (
    ...
);

SELECT TUMBLE_START(user_action_time, INTERVAL '10' MINUTE), COUNT(DISTINCT user_name)
FROM user_actions
GROUP BY TUMBLE(user_action_time, INTERVAL '10' MINUTE);
```

在 DataStream 到 Table 转换时定义

事件时间属性可以用.rowtime 后缀在定义 DataStream schema 的时候来定义。时间戳和 watermark 在这之前一定是在 DataStream 上已经定义好了。

在从 DataStream 到 Table 转换时定义事件时间属性有两种方式。取决于用.rowtime 后缀修饰的字段名字是否是已有字段，事件时间字段可以是：

- 在 schema 的结尾追加一个新的字段
- 替换一个已经存在的字段。

不管在哪种情况下，事件时间字段都表示 DataStream 中定义的事件的时间戳。

```
// Option 1:

// 基于 stream 中的事件产生时间戳和 watermark
DataStream<Tuple2<String, String>> stream =
    → inputStream.assignTimestampsAndWatermarks(...);

// 声明一个额外的逻辑字段作为事件时间属性
Table table = tEnv.fromDataStream(stream, $("user_name"), $("data"),
    → $("user_action_time").rowtime());

// Option 2:

// 从第一个字段获取事件时间，并且产生 watermark
```

```

DataStream<Tuple3<Long, String, String>> stream =
    → inputStream.assignTimestampsAndWatermarks(...);

// 第一个字段已经用作事件时间抽取了，不用再用一个新字段来表示事件时间了
Table table = tEnv.fromDataStream(stream, $("user_action_time").rowtime(),
    → $("user_name"), $("data"));

// Usage:

WindowedTable windowedTable = table.window(Tumble
    .over(lit(10).minutes())
    .on($("user_action_time"))
    .as("userActionWindow"));

```

12.5 例子

12.5.1 分流求平均值

```

public class AvgTemp {
    public static void main(String[] args) throws Exception {
        // 获取流执行环境
        StreamExecutionEnvironment env =
            → StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        // 表环境配置
        EnvironmentSettings settings = EnvironmentSettings
            .newInstance()
            .inStreamingMode()
            .build();

        // 获取表环境
        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);

        SingleOutputStreamOperator<SensorReading> stream = env.addSource(new
            → SensorSource()).filter(r -> r.id.equals("sensor_1"));

        // Table API
        // 将 DataStream 转换成 Table
        Table table = tEnv.fromDataStream(stream, $("id"), $("temperature").as("temp"),
            → $("timestamp").as("ts"));

        // 查询
        Table tableResult = table
    }
}

```

```

    .groupBy($"id")
    .select($"id", $"temp").avg());

    // 将 Table 转换成 DataStream[Row]
    // 只要查询中有聚合操作，必须使用 toRetractStream
    tEnv.toRetractStream(tableResult, Row.class).print();

    // SQL

    // 将 DataStream 转换成临时视图
    tEnv.createTemporaryView("sensor", stream, $"id",
        $"temperature".as("temp"), $"timestamp".as("ts"));

    // 查询
    Table sqlResult = tEnv
        .sqlQuery("SELECT id, AVG(temp) FROM sensor GROUP BY id");

    // 转换成 DataStream
    tEnv.toRetractStream(sqlResult, Row.class).print();

    // 别忘了执行
    env.execute();
}

}

```

12.5.2 求窗口温度平均值

```

// 处理时间，10s 滚动窗口的温度平均值
public class AvgTempPerWindow {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        EnvironmentSettings settings =
            EnvironmentSettings.newInstance().inStreamingMode().build();

        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);

        SingleOutputStreamOperator<SensorReading> stream = env.addSource(new
            SensorSource()).filter(r -> r.id.equals("sensor_1"));

        // 添加处理时间字段
        Table table = tEnv.fromDataStream(stream, $"id", $"temperature".as("temp"),
            $"timestamp".as("ts"), $"ptime".proctime());
    }
}

```

```

Table tableResult = table
    .window(Tumble.over(lit(10).seconds()).on($"ptime").as("w"))
    .groupBy($"id"), $("w"))
    .select($"id"), $("temp").avg(), $("w").end());
tEnv.toRetractStream(tableResult, Row.class).print();

// SQL

tEnv.createTemporaryView("sensor", stream, $"id",
    $("temperature").as("temp"), $("timestamp").as("ts"),
    $("ptime").proctime());
Table sqlResult = tEnv
    .sqlQuery("SELECT id, AVG(temp), TUMBLE_START(ptime, INTERVAL '10'
    → SECOND), TUMBLE_END(ptime, INTERVAL '10' SECOND) FROM sensor GROUP
    → BY id, TUMBLE(ptime, INTERVAL '10' SECOND)");
tEnv.toRetractStream(sqlResult, Row.class).print();

env.execute();
}
}

```

12.5.3 追加流

```

public class AppendExample {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        EnvironmentSettings settings =
            EnvironmentSettings.newInstance().inStreamingMode().build();

        StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);

        SingleOutputStreamOperator<SensorReading> stream = env.addSource(new
            SensorSource()).filter(r -> r.id.equals("sensor_1"));

        Table table = tEnv.fromDataStream(stream);

        Table result = table
            .select($"id"), $("temperature"));

        // 查询中没有聚合操作的情况下，使用 toAppendStream
    }
}

```

```

    tEnv.toAppendStream(result, Row.class).print();

    env.execute();

}
}

```

12.5.4 事件时间例子

```

public class EventTimeExample {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        SingleOutputStreamOperator<Tuple2<String, Long>> stream = env
            .socketTextStream("localhost", 9999)
            .map(new MapFunction<String, Tuple2<String, Long>>() {
                @Override
                public Tuple2<String, Long> map(String value) throws Exception {
                    String[] arr = value.split(" ");
                    return Tuple2.of(arr[0], Long.parseLong(arr[1]) * 1000L);
                }
            })
            .assignTimestampsAndWatermarks(WatermarkStrategy.<Tuple2<String,
                Long>>forMonotonousTimestamps()
                    .withTimestampAssigner(new
                        SerializableTimestampAssigner<Tuple2<String, Long>>() {
                            @Override
                            public long extractTimestamp(Tuple2<String, Long> element,
                                long recordTimestamp) {
                                return element.f1;
                            }
                        }));
    }

    EnvironmentSettings settings =
        EnvironmentSettings.newInstance().inStreamingMode().build();

    StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);

    // .rowtime 将字段指定为事件时间
    Table table = tEnv.fromDataStream(stream, $("key"), $("ts").rowtime());
    Table tableResult = table
        .window(Tumble.over(lit(5).seconds()).on($("ts")).as("w"))
        .groupBy($("key"), $("w"))

```

```
.select($"key"), $("key").count()));

tEnv.toRetractStream(tableResult, Row.class).print();

// SQL
tEnv.createTemporaryView("t", stream, $("key"), $("ts").rowtime());
Table sqlResult = tEnv.sqlQuery("SELECT key, COUNT(key) FROM t GROUP BY key,
→ TUMBLE(ts, INTERVAL '5' SECOND)");
tEnv.toRetractStream(sqlResult, Row.class).print();

env.execute();
}

}
```

第十三章 案例集

13.1 数据倾斜如何解决

```
public class DataSkew {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        env
            .fromElements(
                Tuple3.of("a", 1L, 1000L),
                Tuple3.of("a", 1L, 2000L),
                Tuple3.of("a", 1L, 3000L),
                Tuple3.of("a", 1L, 4000L),
                Tuple3.of("a", 1L, 5000L),
                Tuple3.of("a", 1L, 6000L),
                Tuple3.of("a", 1L, 7000L),
                Tuple3.of("a", 1L, 8000L),
                Tuple3.of("a", 1L, 9000L),
                Tuple3.of("a", 1L, 10000L),
                Tuple3.of("b", 1L, 11000L)
            )
            .map(new MapFunction<Tuple3<String, Long, Long>, Tuple3<String, Long,
            Long>>() {
                @Override
                public Tuple3<String, Long, Long> map(Tuple3<String, Long, Long> value)
                    throws Exception {
                    Random rand = new Random();
                    return Tuple3.of(value.f0 + "-" + rand.nextInt(4), value.f1,
                        value.f2);
                }
            })
            .assignTimestampsAndWatermarks(WatermarkStrategy.<Tuple3<String, Long,
            Long>>forMonotonousTimestamps())
            .withTimestampAssigner(new SerializableTimestampAssigner<Tuple3<String,
            Long, Long>>() {
                @Override
                public long extractTimestamp(Tuple3<String, Long, Long> element, long
                    recordTimestamp) {
                    return element.f2;
                }
            })
        );
    }
}
```

```

.keyBy(r -> r.f0)
.process(new KeyedProcessFunction<String, Tuple3<String, Long, Long>,
    Tuple2<String, Long>>() {
    private ValueState<Tuple2<String, Long>> sum;
    private ValueState<Long> timerTs;
    @Override
    public void open(Configuration parameters) throws Exception {
        super.open(parameters);
        sum = getRuntimeContext().getState(new
            ValueStateDescriptor<Tuple2<String, Long>>("sum",
            Types.TUPLE(Types.STRING, Types.LONG)));
        timerTs = getRuntimeContext().getState(new
            ValueStateDescriptor<Long>("timer", Types.LONG));
    }

    @Override
    public void processElement(Tuple3<String, Long, Long> value, Context
        ctx, Collector<Tuple2<String, Long>> out) throws Exception {
        if (sum.value() == null) {
            sum.update(Tuple2.of(value.f0, value.f1));
            ctx.timerService().registerEventTimeTimer(value.f2 + 10 *
                1000L);
            timerTs.update(value.f2 + 10 * 1000L);
        } else {
            Long cnt = sum.value().f1;
            sum.update(Tuple2.of(value.f0, cnt + value.f1));
            if (timerTs.value() == null) {
                ctx.timerService().registerEventTimeTimer(value.f2 + 10 *
                    1000L);
                timerTs.update(value.f2 + 10 * 1000L);
            }
        }
    }

    @Override
    public void onTimer(long timestamp, OnTimerContext ctx,
        Collector<Tuple2<String, Long>> out) throws Exception {
        super.onTimer(timestamp, ctx, out);
        out.collect(Tuple2.of(ctx.getCurrentKey(), sum.value().f1));
        timerTs.clear();
    }
})
.map(new MapFunction<Tuple2<String, Long>, Tuple3<String, Integer, Long>>()
    {
        @Override
        public Tuple3<String, Integer, Long> map(Tuple2<String, Long> value)
            throws Exception {

```

```

        return Tuple3.of(value.f0.split("-")[0],
                           Integer.parseInt(value.f0.split("-")[1]), value.f1);
    }
}

.keyBy(r -> r.f0)
.process(new KeyedProcessFunction<String, Tuple3<String, Integer, Long>,
         Tuple2<String, Long>>() {
    private MapState<Long, Long> mapState;

    @Override
    public void open(Configuration parameters) throws Exception {
        super.open(parameters);
        mapState = getRuntimeContext().getMapState(
            new MapStateDescriptor<Long, Long>("map", Types.LONG,
                                              Types.LONG)
        );
    }

    @Override
    public void processElement(Tuple3<String, Integer, Long> value, Context
                               ctx, Collector<Tuple2<String, Long>> out) throws Exception {
        mapState.put((long)value.f1, value.f2);
        long sum = 0L;
        for (Long v : mapState.values()) {
            sum += v;
        }
        out.collect(Tuple2.of(value.f0, sum));
    }
}
.print();

env.execute();
}
}

```

13.2 独立访客计算

13.2.1 原始版

```

public class UV {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);
    }
}

```

```

env
    .readTextFile("UserBehavior.csv")
    .map(new MapFunction<String, UserBehavior>() {
        @Override
        public UserBehavior map(String value) throws Exception {
            String[] arr = value.split(",");
            return new UserBehavior(
                arr[0], arr[1], arr[2], arr[3],
                Long.parseLong(arr[4]) * 1000L
            );
        }
    })
    .filter(r -> r.behaviorType.equals("pv"))

    → .assignTimestampsAndWatermarks(WatermarkStrategy.<UserBehavior>forMonotonousTimestamps()
    .withTimestampAssigner(new SerializableTimestampAssigner<UserBehavior>() {
        @Override
        public long extractTimestamp(UserBehavior element, long
        → recordTimestamp) {
            return element.timestamp;
        }
    }))
    .map(new MapFunction<UserBehavior, Tuple2<String, String>>() {
        @Override
        public Tuple2<String, String> map(UserBehavior value) throws Exception
        → {
            return Tuple2.of("key", value.userId);
        }
    })
    .keyBy(r -> r.f0)
    .window(TumblingEventTimeWindows.of(Time.hours(1)))
    .process(new ProcessWindowFunction<Tuple2<String, String>, String, String,
    → TimeWindow>() {
        @Override
        public void process(String s, Context context, Iterable<Tuple2<String,
        → String>> elements, Collector<String> out) throws Exception {
            HashSet<String> set = new HashSet<>();
            for (Tuple2<String, String> e : elements) {
                set.add(e.f1);
            }
            String start = new
            → Timestamp(context.window().getStart()).toString();
            String end = new Timestamp(context.window().getEnd()).toString();

            out.collect("窗口 " + start + " ~~~ " + end + " 的 UV 数据是: " +
            → set.size());
        }
    })
}

```

```

        }
    })
    .print();

    env.execute();
}
}

```

13.2.2 改进版

```

public class UVImproved {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        env
            .readTextFile("UserBehavior.csv")
            .map(new MapFunction<String, UserBehavior>() {
                @Override
                public UserBehavior map(String value) throws Exception {
                    String[] arr = value.split(",");
                    return new UserBehavior(
                        arr[0], arr[1], arr[2], arr[3],
                        Long.parseLong(arr[4]) * 1000L
                    );
                }
            })
            .filter(r -> r.behaviorType.equals("pv"))

            .assignTimestampsAndWatermarks(WatermarkStrategy.<UserBehavior>forMonotonousTimestamps()
                .withTimestampAssigner(new
                    SerializableTimestampAssigner<UserBehavior>() {
                        @Override
                        public long extractTimestamp(UserBehavior element, long
                            recordTimestamp) {
                            return element.timestamp;
                        }
                    }));
            .map(new MapFunction<UserBehavior, Tuple2<String, String>>() {
                @Override
                public Tuple2<String, String> map(UserBehavior value) throws Exception
                {
                    return Tuple2.of("key", value.userId);
                }
            });
    }
}

```

```

    })
    .keyBy(r -> r.f0)
    .window(TumblingEventTimeWindows.of(Time.hours(1)))
    .aggregate(new Agg(), new WindowResult())
    .print();

env.execute();
}

public static class Agg implements AggregateFunction<Tuple2<String, String>,
    <-- HashSet<String>, Long> {
    @Override
    public HashSet<String> createAccumulator() {
        return new HashSet<>();
    }

    @Override
    public HashSet<String> add(Tuple2<String, String> value, HashSet<String>
        <-- accumulator) {
        accumulator.add(value.f1);
        return accumulator;
    }

    @Override
    public Long getResult(HashSet<String> accumulator) {
        return (long) accumulator.size();
    }

    @Override
    public HashSet<String> merge(HashSet<String> a, HashSet<String> b) {
        return null;
    }
}

public static class WindowResult extends ProcessWindowFunction<Long, String,
    <-- String, TimeWindow> {
    @Override
    public void process(String s, Context context, Iterable<Long> elements,
        <-- Collector<String> out) throws Exception {
        String start = new Timestamp(context.window().getStart()).toString();
        String end = new Timestamp(context.window().getEnd()).toString();

        out.collect("窗口 " + start + " ~~~ " + end + " 的 UV 数据是: " +
            <-- elements.iterator().next());
    }
}

```

}

13.2.3 布隆过滤器理论

什么是布隆过滤器

本质上布隆过滤器是一种数据结构，比较巧妙的概率型数据结构（probabilistic data structure），特点是高效地插入和查询，可以用来告诉你“某样东西一定不存在或者可能存在”。

相比于传统的 List、Set、Map 等数据结构，它更高效、占用空间更少，但是缺点是其返回的结果是概率性的，而不是确切的。

实现原理

HashMap 的问题

讲述布隆过滤器的原理之前，我们先思考一下，通常你判断某个元素是否存在用的是什么？应该蛮多人回答 *HashMap* 吧，确实可以将值映射到 *HashMap* 的 Key，然后可以在 O(1) 的时间复杂度内返回结果，效率奇高。但是 *HashMap* 的实现也有缺点，例如存储容量占比高，考虑到负载因子的存在，通常空间是不能被用满的，而一旦你的值很多例如上亿的时候，那 *HashMap* 占据的内存大小就变得很可观了。

还比如说你的数据集存储在远程服务器上，本地服务接受输入，而数据集非常大不可能一次性读进内存构建 *HashMap* 的时候，也会存在问题。

布隆过滤器数据结构

布隆过滤器是一个 bit 向量或者说 bit 数组，长这样：

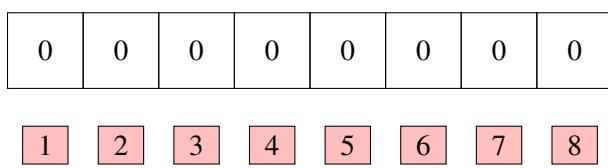


图 13.1：数组和索引

如果我们要映射一个值到布隆过滤器中，我们需要使用多个不同的哈希函数生成多个哈希值，并对每个生成的哈希值指向的 bit 位置 1，例如针对值“baidu”和三个不同的哈希函数分别生成了哈希值 1、4、7，则上图转变为：

Ok，我们现在再存一个值“tencent”，如果哈希函数返回 3、4、8 的话，图继续变为：

值得注意的是，4 这个 bit 位由于两个值的哈希函数都返回了这个 bit 位，因此它被覆盖了。现在我们如果想查询“dianping”这个值是否存在，哈希函数返回了 1、5、8 三个值，结果我们发现 5 这个 bit 位上的值为 0，说明没有任何一个值映射到这个 bit 位上，因此我们可以很确定地说“dianping”这个值不存在。而当我们需要查询“baidu”这个值是否存在的话，那么哈希函数必然会返回 1、4、7，然后我们检查发现这三个 bit 位上的值均为 1，那么我们可以说“baidu”存在了么？答案是不可以，只能是“baidu”这个值可能存在。

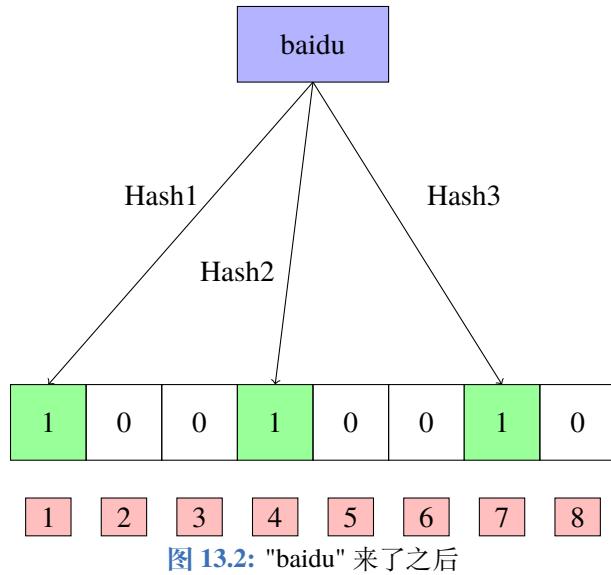


图 13.2: "baidu" 来了之后

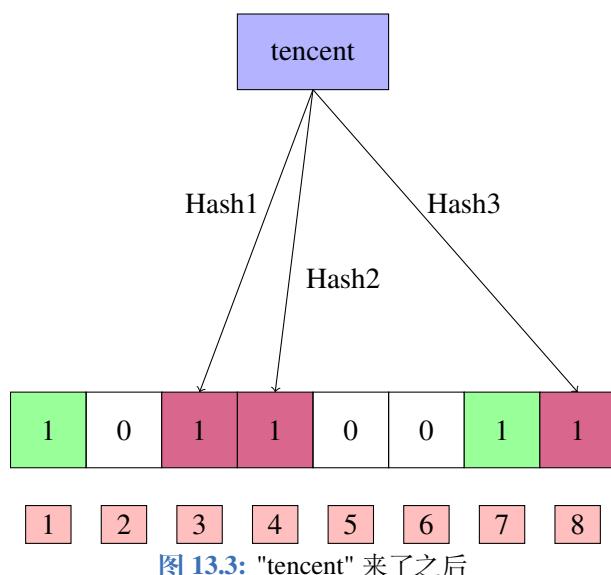


图 13.3: "tencent" 来了之后

这是为什么呢？答案很简单，因为随着增加的值越来越多，被置为 1 的 bit 位也会越来越多，这样某个值“taobao”即使没有被存储过，但是万一哈希函数返回的三个 bit 位都被其他值置位了 1，那么程序还是会判断“taobao”这个值存在。

如何选择哈希函数个数和布隆过滤器长度

很显然，过小的布隆过滤器很快所有的 bit 位均为 1，那么查询任何值都会返回“可能存在”，起不到过滤的目的了。布隆过滤器的长度会直接影响误报率，布隆过滤器越长其误报率越小。

另外，哈希函数的个数也需要权衡，个数越多则布隆过滤器 bit 位置位 1 的速度越快，且布隆过滤器的效率越低；但是如果太少的话，那我们的误报率会变高。

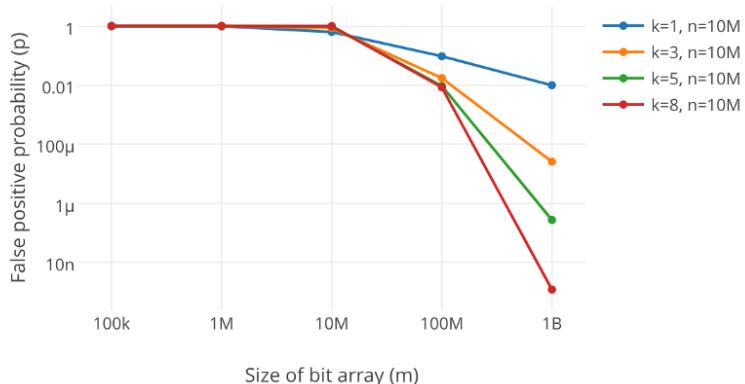


图 13.4: 参数的影响

如何选择适合业务的 k 和 m 值呢，这里直接贴一个公式：

$$m = -\frac{n \ln p}{(\ln 2)^2} \quad (13.1)$$

$$k = \frac{m}{n} \ln 2 \quad (13.2)$$

如何推导这个公式这里只是提一句，因为对于使用来说并没有太大的意义，你让一个高中生来推会推得很快。k 次哈希函数某一 bit 位未被置为 1 的概率为：

$$(1 - \frac{1}{m})^k \quad (13.3)$$

插入 n 个元素后依旧为 0 的概率和为 1 的概率分别是：

$$(1 - \frac{1}{m})^{nk} \text{ 和 } 1 - (1 - \frac{1}{m})^{nk}.$$

标明某个元素是否在集合中所需的 k 个位置都按照如上的方法设置为 1，但是该方法可能会使算法错误的认为某一原本不在集合中的元素却被检测为在该集合中（False Positives），该概率由以下公式确定

$$[1 - (1 - \frac{1}{m})^{nk}]^k \approx (1 - e^{-\frac{kn}{m}})^k \quad (13.4)$$

最佳实践

常见的使用场景有，利用布隆过滤器减少磁盘 IO 或者网络请求，因为一旦一个值必定不存在的话，我们可以不用进行后续昂贵的查询请求。

另外，既然你使用布隆过滤器来加速查找和判断是否存在，那么性能很低的哈希函数不是个好选择，推荐 MurmurHash、Fnv 这些。

大 Value 拆分

Redis 因其支持 setbit 和 getbit 操作，且纯内存性能高等特点，因此天然就可以作为布隆过滤器来使用。但是布隆过滤器的不当使用极易产生大 Value，增加 Redis 阻塞风险，因此生成环境中建议对体积庞大的布隆过滤器进行拆分。

拆分的形式方法多种多样，但是本质是不要将 Hash(Key) 之后的请求分散在多个节点的多个小 bitmap 上，而是应该拆分成多个小 bitmap 之后，对一个 Key 的所有哈希函数都落在这一个小 bitmap 上。

13.2.4 布隆过滤器版

```
public class UVBloomFilter {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        env
            .readTextFile("UserBehavior.csv")
            .map(new MapFunction<String, UserBehavior>() {
                @Override
                public UserBehavior map(String value) throws Exception {
                    String[] arr = value.split(",");
                    return new UserBehavior(
                        arr[0], arr[1], arr[2], arr[3],
                        Long.parseLong(arr[4]) * 1000L
                    );
                }
            })
            .filter(r -> r.behaviorType.equals("pv"))

        .assignTimestampsAndWatermarks(WatermarkStrategy.<UserBehavior>forMonotonousTimestamps()
            .withTimestampAssigner(new
                SerializableTimestampAssigner<UserBehavior>() {
                    @Override
                    public long extractTimestamp(UserBehavior element, long
                        recordTimestamp) {
                        return element.timestamp;
                    }
                })
            .map(new MapFunction<UserBehavior, Tuple2<String, String>>() {

```

```

    @Override
    public Tuple2<String, String> map(UserBehavior value) throws Exception
    {
        return Tuple2.of("key", value.userId);
    }
}

.keyBy(r -> r.f0)
.window(TumblingEventTimeWindows.of(Time.hours(1)))
.aggregate(new Agg(), new WindowResult())
.print();

env.execute();
}

public static class Agg implements AggregateFunction<Tuple2<String, String>,
    Tuple2<BloomFilter<Long>, Long>, Long> {
    @Override
    public Tuple2<BloomFilter<Long>, Long> createAccumulator() {
        // 假设独立用户数量是一百万，误判率是 0.01
        return Tuple2.of(BloomFilter.create(Funnels.longFunnel(), 1000000, 0.01),
            0L);
    }

    @Override
    public Tuple2<BloomFilter<Long>, Long> add(Tuple2<String, String> value,
        Tuple2<BloomFilter<Long>, Long> accumulator) {
        if (!accumulator.f0.mightContain(Long.parseLong(value.f1))) {
            // 如果 userID 没来过，那么执行 put 操作
            accumulator.f0.put(Long.parseLong(value.f1));
            accumulator.f1 += 1L; // UV 数量加一
        }
        return accumulator;
    }

    @Override
    public Long getResult(Tuple2<BloomFilter<Long>, Long> accumulator) {
        return accumulator.f1;
    }

    @Override
    public Tuple2<BloomFilter<Long>, Long> merge(Tuple2<BloomFilter<Long>, Long> a,
        Tuple2<BloomFilter<Long>, Long> b) {
        return null;
    }
}

```

```

public static class WindowResult extends ProcessWindowFunction<Long, String,
→ String, TimeWindow> {
    @Override
    public void process(String s, Context context, Iterable<Long> elements,
→ Collector<String> out) throws Exception {
        String start = new Timestamp(context.window().getStart()).toString();
        String end = new Timestamp(context.window().getEnd()).toString();

        out.collect(" 窗口 " + start + " ~~~ " + end + " 的 UV 数据是: " +
→ elements.iterator().next());
    }
}

```

13.3 订单超时检测

```

public class OrderTimeoutDetectWithoutCEP {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
→ StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        SingleOutputStreamOperator<Tuple3<String, String, Long>> orderStream = env
            .fromElements(
                Tuple3.of("order-1", "create", 1000L),
                Tuple3.of("order-2", "create", 2000L),
                Tuple3.of("order-1", "pay", 3000L)
            )
            .assignTimestampsAndWatermarks(WatermarkStrategy.<Tuple3<String, String,
→ Long>>forMonotonousTimestamps()
            .withTimestampAssigner(new SerializableTimestampAssigner<Tuple3<String,
→ String, Long>>() {
                @Override
                public long extractTimestamp(Tuple3<String, String, Long> element,
→ long recordTimestamp) {
                    return element.f2;
                }
            }));
    }

    orderStream
        .keyBy(r -> r.f0)
        .process(new KeyedProcessFunction<String, Tuple3<String, String, Long>,
→ String>() {

```

```

private ValueState<Tuple3<String, String, Long>> orderState;
@Override
public void open(Configuration parameters) throws Exception {
    super.open(parameters);
    orderState = getRuntimeContext().getState(
        new ValueStateDescriptor<Tuple3<String, String,
        Long>>("order-state", Types.TUPLE(Types.STRING,
        Types.STRING, Types.LONG))
    );
}

@Override
public void processElement(Tuple3<String, String, Long> value, Context
    ctx, Collector<String> out) throws Exception {
    if (value.f1.equals("pay")) {
        out.collect(" 订单 ID: " + value.f0 + " 已经支付");
        orderState.update(value);
    } else if (value.f1.equals("create")) {
        if (orderState.value() == null) {
            // 说明 pay 事件没有在 create 事件之前到达
            ctx.timerService().registerEventTimeTimer(value.f2 + 5 *
                1000L);
            orderState.update(value);
        }
    }
}

@Override
public void onTimer(long timestamp, OnTimerContext ctx,
    Collector<String> out) throws Exception {
    super.onTimer(timestamp, ctx, out);
    if (orderState.value() != null &&
        orderState.value().f1.equals("create")) {
        out.collect(" 订单 ID: " + ctx.getCurrentKey() + " 支付超时");
        orderState.clear();
    }
}
.print();

env.execute();
}
}

```

13.4 实时对帐

```

public class TwoStreamJoin {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        SingleOutputStreamOperator<Tuple3<String, String, Long>> orderStream = env
            .fromElements(
                Tuple3.of("order-1", "pay", 1000L),
                Tuple3.of("order-2", "pay", 2000L)
            )
            .assignTimestampsAndWatermarks(WatermarkStrategy.<Tuple3<String,
                String, Long>>forMonotonousTimestamps()
                    .withTimestampAssigner(new
                        SerializableTimestampAssigner<Tuple3<String, String,
                        Long>>() {
                            @Override
                            public long extractTimestamp(Tuple3<String, String, Long>
                                element, long recordTimestamp) {
                                return element.f2;
                            }
                        }));
    }

    SingleOutputStreamOperator<Tuple3<String, String, Long>> payStream = env
        .fromElements(
            Tuple3.of("order-1", "weixin", 3000L),
            Tuple3.of("order-3", "weixin", 4000L)
        )
        .assignTimestampsAndWatermarks(WatermarkStrategy.<Tuple3<String,
            String, Long>>forMonotonousTimestamps()
                .withTimestampAssigner(new
                    SerializableTimestampAssigner<Tuple3<String, String,
                    Long>>() {
                        @Override
                        public long extractTimestamp(Tuple3<String, String, Long>
                            element, long recordTimestamp) {
                            return element.f2;
                        }
                    }));
    }

    SingleOutputStreamOperator<String> result = orderStream
        .keyBy(r -> r.f0)
        .connect(payStream.keyBy(r -> r.f0))
        .process(new MatchFunction());
}

```

```

        result.print();

        env.execute();
    }

    public static class MatchFunction extends CoProcessFunction<Tuple3<String, String,
    → Long>, Tuple3<String, String, Long>, String> {
        private ValueState<Tuple3<String, String, Long>> orderState;
        private ValueState<Tuple3<String, String, Long>> payState;
        @Override
        public void open(Configuration parameters) throws Exception {
            super.open(parameters);
            orderState = getRuntimeContext().getState(new
                → ValueStateDescriptor<Tuple3<String, String, Long>>("order",
                → Types.TUPLE(Types.STRING, Types.STRING, Types.LONG)));
            payState = getRuntimeContext().getState(new
                → ValueStateDescriptor<Tuple3<String, String, Long>>("pay",
                → Types.TUPLE(Types.STRING, Types.STRING, Types.LONG)));
        }

        @Override
        public void processElement1(Tuple3<String, String, Long> value, Context ctx,
        → Collector<String> out) throws Exception {
            if (payState.value() != null) {
                out.collect(" 订单" + value.f0 + " 对账成功");
                payState.clear();
            } else {
                orderState.update(value); // 说明 app 端支付事件先到了
                ctx.timerService().registerEventTimeTimer(value.f2 + 5000L); // order
                → 事件等待 pay 事件 5 秒钟
            }
        }

        @Override
        public void processElement2(Tuple3<String, String, Long> value, Context ctx,
        → Collector<String> out) throws Exception {
            if (orderState.value() != null) {
                out.collect(" 订单" + value.f0 + " 对账成功");
                orderState.clear();
            } else {
                payState.update(value);
                ctx.timerService().registerEventTimeTimer(value.f2 + 5000L); // pay 事
                → 件等待 order 事件 5 秒钟
            }
        }
    }
}

```

```
    @Override
    public void onTimer(long timestamp, OnTimerContext ctx, Collector<String> out)
        throws Exception {
        super.onTimer(timestamp, ctx, out);
        if (orderState.value() != null) {
            out.collect(" 订单" + orderState.value().f0 + " 对账失败");
            orderState.clear();
        }
        if (payState.value() != null) {
            out.collect(" 订单" + payState.value().f0 + " 对账失败");
            payState.clear();
        }
    }
}
```

13.5 实时热门商品

13.5.1 实现思路

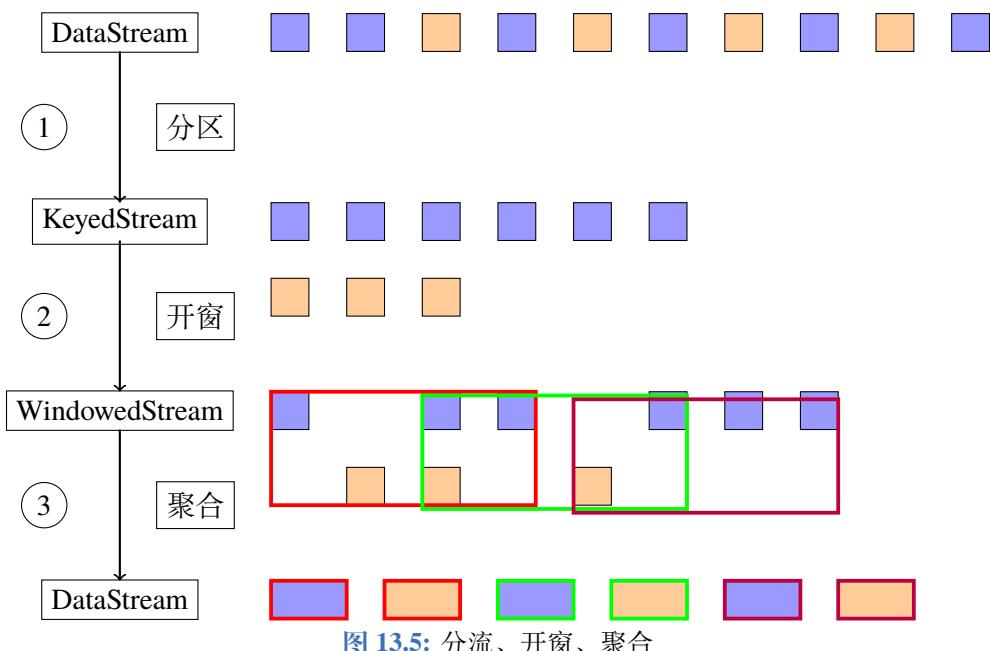


图 13.5: 分流、开窗、聚合

先来做分流

```
.keyBy(r -> r.itemId)
```

开窗

聚合

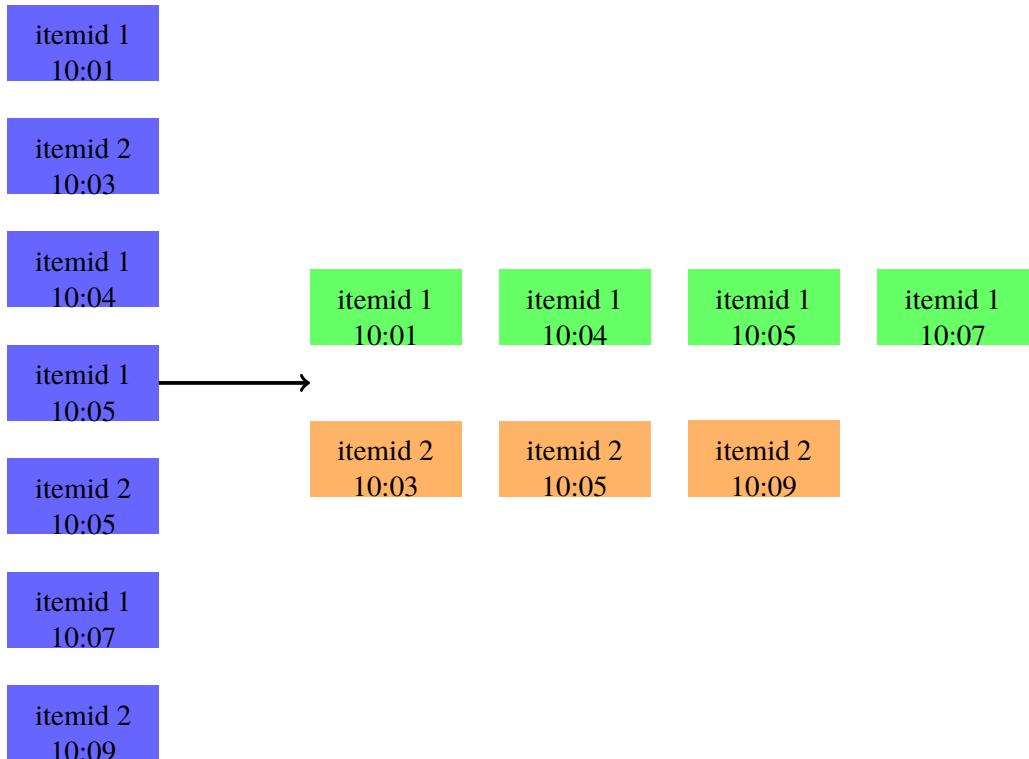


图 13.6: 分流

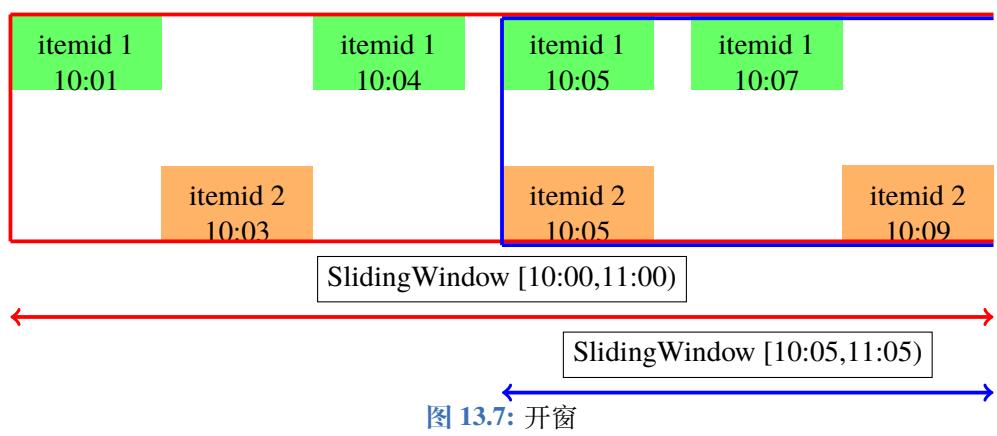


图 13.7: 开窗

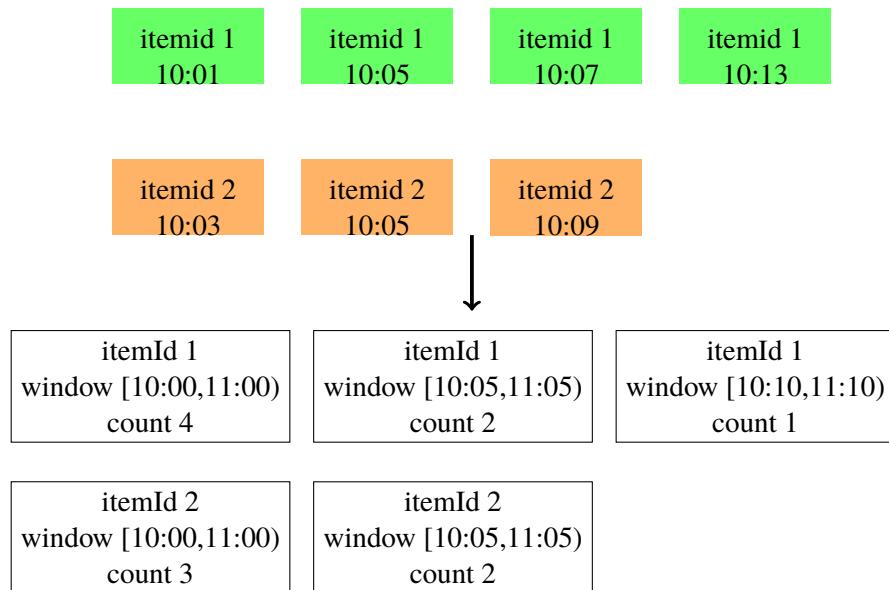


图 13.8: 聚合

使用窗口结束时间分流

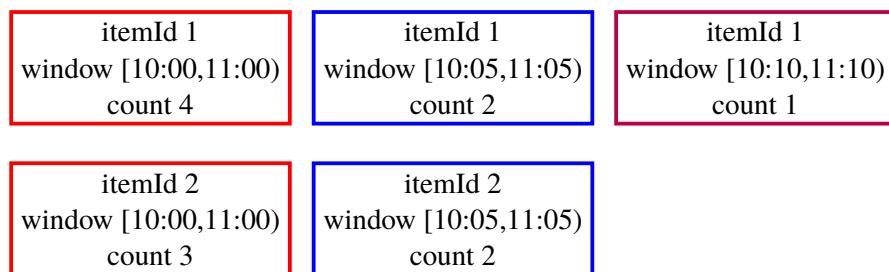


图 13.9: 继续分流

13.5.2 底层 API 实现方式

UserBehavior POJO 类

```
// 用户行为的每一条数据就是一个 POJO 类
public class UserBehavior {
    public String userId;
    public String itemId;
    public String categoryId;
    public String behaviorType;
    public Long timestamp;

    public UserBehavior() {
    }

    public UserBehavior(String userId, String itemId, String categoryId, String
    behaviorType, Long timestamp) {
        this.userId = userId;
    }
}
```

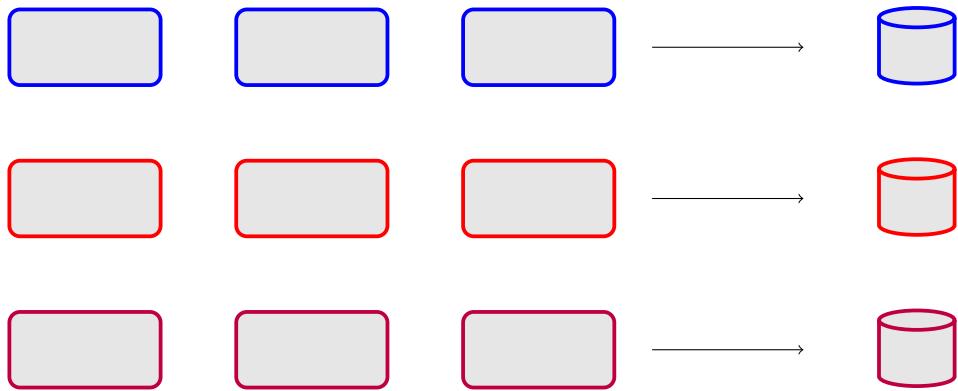


图 13.10: 使用窗口结束时间分流

```

    this.itemId = itemId;
    this.categoryId = categoryId;
    this.behaviorType = behaviorType;
    this.timestamp = timestamp;
}

@Override
public String toString() {
    return "UserBehavior{" +
        "userId=" + userId + '\'' +
        ", itemId=" + itemId + '\'' +
        ", categoryId=" + categoryId + '\'' +
        ", behaviorType=" + behaviorType + '\'' +
        ", timestamp=" + new Timestamp(timestamp) +
        '}';
}
}

```

ItemCount POJO 类

```

// 每个窗口里面，商品 ID 的 PV 数据
public class ItemViewCount {
    public String itemId;
    public Long count; // pv 数据
    public Long windowStart;
    public Long windowEnd;

    public ItemViewCount() {
    }

    public ItemViewCount(String itemId, Long count, Long windowStart, Long windowEnd) {
        this.itemId = itemId;
        this.count = count;
        this.windowStart = windowStart;
    }
}

```

```

    this.windowEnd = windowEnd;
}

@Override
public String toString() {
    return "ItemViewCount{" +
        "itemId='" + itemId + '\'' +
        ", count=" + count +
        ", windowStart=" + new Timestamp(windowStart) +
        ", windowEnd=" + new Timestamp(windowEnd) +
        '}';
}
}

```

主体程序

```

public class TopNHotItems {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env =
            StreamExecutionEnvironment.getExecutionEnvironment();
        env.setParallelism(1);

        SingleOutputStreamOperator<UserBehavior> userBehaviorStream = env
            .readTextFile("/Users/yuanzuo/Desktop/Flink1125SH/src/main/resources/UserBehavior.csv")
            .map(new MapFunction<String, UserBehavior>() {
                @Override
                public UserBehavior map(String value) throws Exception {
                    String[] arr = value.split(",");
                    return new UserBehavior(
                        arr[0], arr[1], arr[2], arr[3],
                        Long.parseLong(arr[4]) * 1000L
                    );
                }
            })
            .filter(r -> r.behaviorType.equals("pv"))
            // 对于离线数据集来讲, flink 只会插入连个水位线, 开始的负无穷大, 末尾的正
            // 无穷大
            .assignTimestampsAndWatermarks(WatermarkStrategy.<UserBehavior>forMonotonousTimestamps()
                .withTimestampAssigner(new
                    SerializableTimestampAssigner<UserBehavior>() {
                        @Override
                        public long extractTimestamp(UserBehavior element, long
                            recordTimestamp) {
                            return element.timestamp;
                        }
                    })
            );
    }
}

```

```

        }

    })));
}

SingleOutputStreamOperator<ItemViewCount> itemViewCountStream =
    userBehaviorStream
        .keyBy(r -> r.itemId)
        // 进到了窗口的数据有什么特点？
        // 相同的 itemId
        .window(SlidingEventTimeWindows.of(Time.hours(1), Time.minutes(5)))
        .aggregate(new CountAgg(), new WindowResult());

SingleOutputStreamOperator<String> result = itemViewCountStream
    // 每一条支流，属于同一个窗口的不同 itemId 的 pv 数量
    .keyBy(r -> r.windowEnd)
    .process(new TopN(3L));

result.print();

env.execute();
}

public static class CountAgg implements AggregateFunction<UserBehavior, Long, Long>
{
    @Override
    public Long createAccumulator() {
        return 0L;
    }

    @Override
    public Long add(UserBehavior value, Long accumulator) {
        return accumulator + 1L;
    }

    @Override
    public Long getResult(Long accumulator) {
        return accumulator;
    }

    @Override
    public Long merge(Long a, Long b) {
        return null;
    }
}

public static class WindowResult extends ProcessWindowFunction<Long, ItemViewCount,
    String, TimeWindow> {
}

```

```

@Override
public void process(String itemId, Context ctx, Iterable<Long> elements,
    Collector<ItemViewCount> out) throws Exception {
    out.collect(new ItemViewCount(itemId, elements.iterator().next(),
        ctx.window().getStart(), ctx.window().getEnd()));
}

public static class TopN extends KeyedProcessFunction<Long, ItemViewCount, String>
{
    public Long threshold;

    public ListState<ItemViewCount> itemViewCountListState;

    public TopN(Long threshold) {
        this.threshold = threshold;
    }

    @Override
    public void open(Configuration parameters) throws Exception {
        super.open(parameters);
        itemViewCountListState = getRuntimeContext().getListState(
            new ListStateDescriptor<ItemViewCount>("list",
                Types.POJO(ItemViewCount.class))
        );
    }

    @Override
    public void processElement(ItemViewCount value, Context ctx, Collector<String>
        out) throws Exception {
        itemViewCountListState.add(value);
        // 某一个时间戳只能注册一个定时器
        // 所以这里只会在第一条数据到来的时候注册一个定时器
        ctx.timerService().registerEventTimeTimer(value.windowEnd + 1L);
    }

    @Override
    public void onTimer(long timestamp, OnTimerContext ctx, Collector<String> out)
        throws Exception {
        super.onTimer(timestamp, ctx, out);
        // 定时器用来排序
        ArrayList<ItemViewCount> itemViewCountArrayList = new ArrayList<>();
        for (ItemViewCount ivc : itemViewCountListState.get()) {
            itemViewCountArrayList.add(ivc);
        }
        // 清空列表状态变量
    }
}

```

```
itemViewCountListState.clear();

// 排序
itemViewCountArrayList.sort(new Comparator<ItemViewCount>() {
    @Override
    public int compare(ItemViewCount o1, ItemViewCount o2) {
        return o2.count.intValue() - o1.count.intValue();
    }
});

StringBuilder result = new StringBuilder();

result
    .append("=====\n")
    .append(" 窗口结束时间: ")
    .append(new Timestamp(timestamp - 1L)) // 恢复窗口结束时间
    .append("\n");

for (int i = 0; i < this.threshold; i++) {
    ItemViewCount itemViewCount = itemViewCountArrayList.get(i);
    result
        .append(" 第" + (i + 1) + " 名商品的 ID 是: " +
               itemViewCount.itemId)
        .append("\n")
        .append("PV 数量是: " + itemViewCount.count)
        .append("\n");
}

result
    .append("=====\n");
out.collect(result.toString());
}
}
```

13.5.3 Flink SQL 实现方式

```
public class TopNSQL {  
    public static void main(String[] args) throws Exception {  
        StreamExecutionEnvironment env =  
            StreamExecutionEnvironment.getExecutionEnvironment();  
        env.setParallelism(1);  
  
        SingleOutputStreamOperator<UserBehavior> stream = env  
            .readTextFile("UserBehavior.csv")
```

```

.map(new MapFunction<String, UserBehavior>() {
    @Override
    public UserBehavior map(String value) throws Exception {
        String[] arr = value.split(",");
        return new UserBehavior(arr[0], arr[1], arr[2], arr[3],
            Long.parseLong(arr[4]) * 1000L);
    }
})
.filter(r -> r.behaviorType.equals("pv"))

→ .assignTimestampsAndWatermarks(WatermarkStrategy.<UserBehavior>forMonotonousTimestamps(
    .withTimestampAssigner(new
        → SerializableTimestampAssigner<UserBehavior>() {
            @Override
            public long extractTimestamp(UserBehavior element, long
                → recordTimestamp) {
                return element.timestamp;
            }
        }));
    }));
```

// SQL

```

EnvironmentSettings settings =
    → EnvironmentSettings.newInstance().inStreamingMode().build();

StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);

tEnv.createTemporaryView("t", stream, ${"itemId"},
    → ${"timestamp"} .rowtime().as("ts"));

// 按照 itemid 分流开窗聚合，结果是每一个 itemid 在每一个窗口的浏览量
String innerSQL = "SELECT itemId, COUNT(itemId) as itemCount, HOP_END(ts,
    → INTERVAL '5' MINUTE, INTERVAL '1' HOUR) as windowEnd" +
    " FROM t GROUP BY HOP(ts, INTERVAL '5' MINUTE, INTERVAL '1' HOUR),
    → itemId";
```

// 按照 windowEnd 分区，然后按照浏览量降序排列

```

String midSQL = "SELECT *, ROW_NUMBER() OVER (PARTITION BY windowEnd ORDER BY
    → itemCount DESC) as row_num" +
    " FROM (" + innerSQL + ")";
```

// 取出前三名

```

String outerSQL = "SELECT * FROM (" + midSQL + ") WHERE row_num <= 3";

Table result = tEnv.sqlQuery(outerSQL);

tEnv.toRetractStream(result, Row.class).print();
```

```
    env.execute();  
}  
}
```

附录 A 逻辑时钟 - 如何刻画分布式中的事件顺序

逻辑时钟是描述分布式系统中时序和因果关系的一种机制。由于网络延迟、时钟漂移等现实问题，我们无法建立一个全局物理时钟来描述时序，因此区别于物理时钟的「逻辑时钟」机制应运而生。第一个逻辑时钟算法是由分布式领域的「大神」Lamport 在 1978 年提出的 Lamport 时钟算法。



图 A.1: 时钟

本文将从一个基础问题的讨论切入，逐步介绍：

- 为什么需要逻辑时钟—物理时钟是否可行？
 - 相对论带来了什么启示—事件的相对性
 - 最早期的逻辑时钟算法有什么特点—Lamport 逻辑时钟
 - 后续改进的算法有哪些—「向量时钟」和「版本向量」等等
- 逻辑时钟的思想有趣而深刻，值得探究。

A.1 如何确定分布式系统中事件的发生顺序？

首先，我们来观察一个「如何确定分布式系统中事件的发生顺序」的问题：

在这个分布式系统中，有三个独立的进程 A、B、C：

1. 首先，进程 A 中发生了一个事件 a，并把这个事件消息同步给另外两个进程 B 和 C。
2. B 收到消息后，发生了一个事件 b，并把这个事件消息同步给进程 C。
3. 但是由于无法确定的网络延迟原因，导致进程 A 发出的消息到达 C 晚于进程 B 发出的消息到达 C，这样进程 C 的视角上，最终看到的事件顺序是 b,a，但是这与事实是相悖的。

可以看到，在分布式通信中，由于网络延迟的不确定性，**仅仅以接收顺序作为整个分布式系统中事件的发生顺序是不可取的**。

下面我们再观察一个稍微现实点的例子。

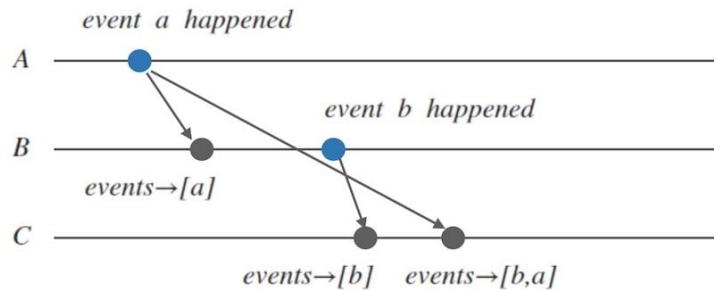


图 A.2: 由于网络延迟, 进程 B 发出的消息晚于进程 A 发出的消息到达进程 C

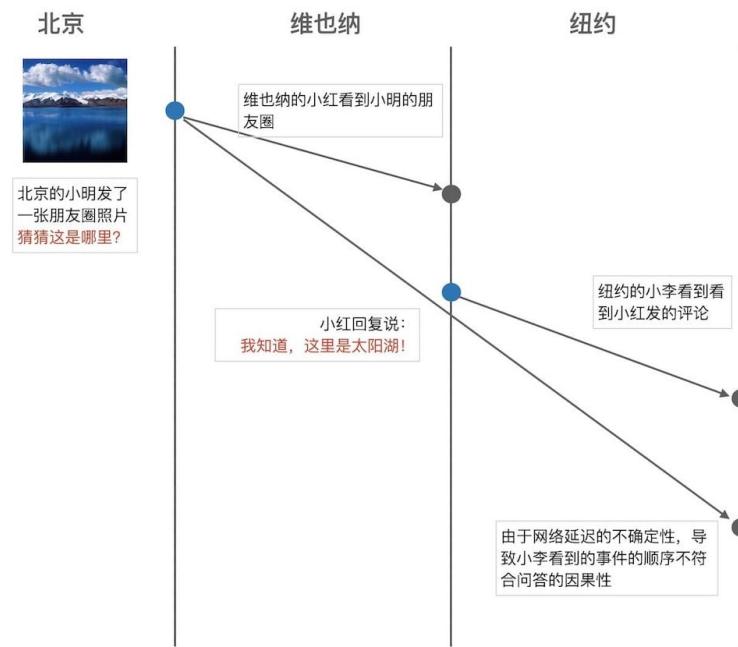


图 A.3: 由于网络延迟, 小李看到的事件顺序并不符合问答的因果关系

1. 假设朋友圈有三个数据中心，分别在北京、维也纳和纽约。
2. 北京小明在朋友圈中发了一张风景图，并问到「猜猜这里是哪里？」，这条消息被扩散到其他数据中心。
3. 维也纳的小红看到这个消息后，回复说她知道。这条回复也被扩散到其他数据中心。
4. 但是由于无法确定的网络延迟的原因，导致纽约的数据中心先收到小红的回复，而后收到了原始的提问消息。这样，导致最终小李看到的问答顺序是不符合问答的因果一致性的。

以上的例子，是腾讯微信朋友圈真实碰到并解决的问题。

下面是微信朋友圈架构设计的两个关于因果一致性算法设计的 PPT 截图：

在腾讯的分享 PPT 中，提到了「向量时钟」，不过我们稍后再揭开它的神秘面纱。

而这两个例子中，我们面临的要解决的问题，其实是分布式系统中的因果一致性，换句话说，就是如何准确刻画分布式中的事件顺序，**显然仅仅简单地依靠进程接收到的(看到的)事件发生顺序是不准确的。**

因果一致性

- 有因才有果
- 因发生在果之前
- 先看见因，才看见果
 - 先看见评论（因），才看见对此评论的回复（果）

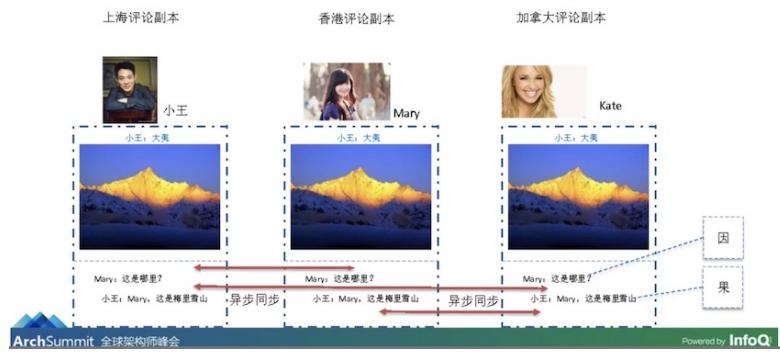


图 A.4: 《微信朋友圈技术之道》中分享的关于因果性的 PPT

因果一致性算法

- 每条评论都有一个唯一的且递增的数字 ID，确保排重
- 每条评论的 ID 都必须比本地已经见过的全局最大的 ID 大，确保因果关系
- 广播本地看到的所有评论和新评论到其它 IDC；相同 ID 的评论合并排重

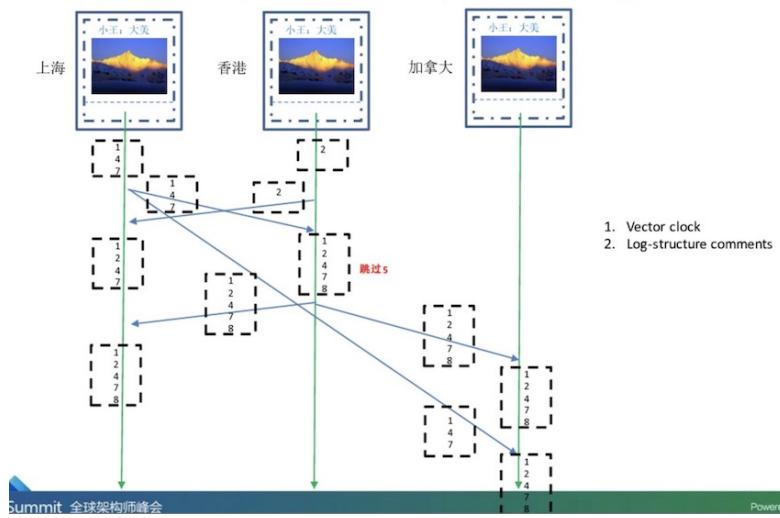


图 A.5: 《微信朋友圈技术之道》中分享的关于向量时钟的 PPT

A.2 全局物理时钟

对于上面提到的两个问题，我们试图解决的一个天然想法是，记录每个分布式进程中发生事件的原始时间戳，并把它连同事件本身扩散到其他节点，这样其他节点的视角上就可以观察到完整的因果顺序了？

If you have one clock, you know what the time is. If you have two, you are not sure.

大家都清楚的一点是，不同节点的物理时钟其实是不一致的，而且无法做到精确一致。

其原因：

1. 仍然是由于网络延迟的不确定，我们无法通过网络同步时间来获取一个全局一致的物理时钟。
2. 现实中的多个时钟，即使时间已经调成一致，但是由于日积月累的计时速率的差异，会导致时钟漂移而显示不同的时间。

如此看来，寄希望于一个全局的时钟来对事件顺序做全局标定也是不现实的。

A.3 全序和偏序

在数学上，「顺序」是如何描述的？

我们看下序理论中的两种序关系：偏序 (partial ordering) 和全序 (total ordering)。

偏序：假设 \leq 是集合 S 上的一个二元关系，如果 \leq 满足：

1. 自反性：对 S 中任意的元素 a ，都有 $a \leq a$
2. 反对称性：如果对于 S 中的两个元素 a 和 b ， $a \leq b$ 且 $b \leq a$ ，那么 $a = b$
3. 传递性：如果对于 S 中的三个元素，有 $a \leq b$ 且 $b \leq c$ ，那么 $a \leq c$

以上的数学内容其实不那么重要 关键理解：偏序关系是一种序关系，但只是部分元素有序，并不是全部元素都可以比较。

全序则比偏序的要求更为严格一些，在偏序的基础上，多了一个完全性的条件：

1. 完全性：对于 S 中的任意 a 和 b 元素，必然有 $a \leq b$ 或 $b \leq a$

可以看出，实际上，全序就是在偏序的基础上，要求全部元素都必须可以比较。

总结来看，简短说：偏序是部分可比较的序关系，全序是全部可比较的序关系。

举例来看：

1. 自然数集合中的比较大小的关系，就是一种全序关系。
2. 集合之间的包含关系，则是一种偏序关系（集合之间可以有包含关系，也可以没有包含关系）。
3. 复数之间的大小关系，是一种偏序关系，而复数的模的大小关系，则是一种全序关系。

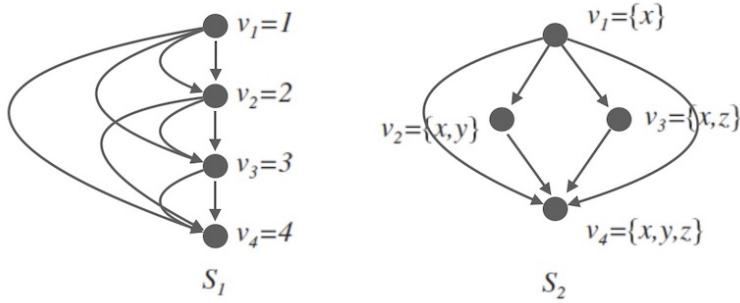


图 A.6: \$S_1\$ 描述了全序关系; \$S_2\$ 描述了偏序关系

下面用简单的有向图(哈斯图)来描述下全序和偏序的区别:

下图中 \$S_1\$ 上的图示关系, 描述的是整数之间的大小顺序, 是一种全序关系, 可以看到任意两个元素之间都可以比较顺序。

而 \$S_2\$ 上的关系, 描述的是集合之间的包含关系, 是一种偏序关系, 其中 \$v_2\$ 和 \$v_3\$ 是不可比较的。

现在我们回头看, 寻求全局时钟为分布式节点中的时间做顺序标定的方式, 其实是在寻求一种全序关系来描述分布式中的事件顺序, 而且是严格对齐真实的物理时钟的全序关系。

A.4 事件先后的相对性

逻辑时钟的概念是由著名的分布式系统科学家 Leslie Lamport (2013 年图灵奖得主) 提出的, 在他的那篇著名的论文「Time, Clocks and the Ordering of Events in a Distributed System」的介绍上, Lamport 提到了著名的狭义相对论:

Special relativity teaches us that there is no invariant total ordering of events in space-time; different observers can disagree about which of two events happened first. There is only a partial order in which an event e1 precedes an event e2 iff e1 can causally affect e2.

— Leslie Lamport 《Time, Clocks and the Ordering of Events in a Distributed System》

爱因斯坦的狭义相对论告诉我们, **时空中不存在绝对的全序事件顺序, 不同的观察者可能对哪个事件是先发生的无法达成一致。但是有偏序关系存在, 当事件 \$e_2\$ 是由事件 \$e_1\$ 引起的时候, \$e_1\$ 和 \$e_2\$ 之间才有先后关系。**

对于「不同的观察者可能对哪个事件是先发生的无法达成一致」这个说法, 我们从同时的相对性开始说起:

根据狭义相对论, 发生在空间中不同位置的两个事件, 它们的同时性并不具有绝对的意义, 我们没办法肯定地说它们是否为同时发生。若在某一参考系中此两事件是同时的, 则在另一相对于原参考系等速运动的新参考系中, 此两事件将不再同时。

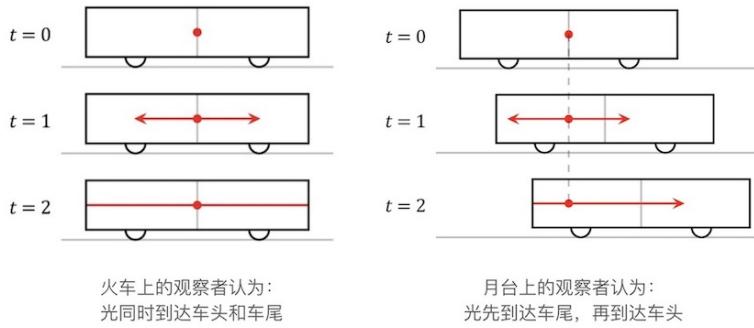


图 A.7: 狹義相對論中著名的火車思想實驗

—维基百科「同时的相对性」

因为狭义相对论最基本的假设，光速不变原理：无论在何种惯性参考系下，光在真空中的传播速度相对观察者都是一个常数。所以「同时」这个概念也是相对的。

关于同时的相对性，有一个著名的火车思想实验：

有一个观察者 A 在移动的火车中间，有另一位观察者 B 在地面上的月台上，当两个观察者相遇时，一道闪光从火车的中央发出。

对于火车上的观察者 A 而言，由于火车头和火车尾距离光源的距离是相同的，因此观察到了光同时到达了车头和车尾。

但是对于月台的观察者 B 而言，火车的尾部会迎向光移动，而车头会远离光移动，而且光速是有限的，且相对于两个观察者都是相同的常数，所以 B 认为光会先到达车尾，后到达车头。

这样，对于不同参照系的观察者而言，事件的顺序并没有一个一致性的结论。之所以得出这样神奇的结论，仍然是因为关键的「光速不变原理」。但是，这并不意味着发生了因果上的逻辑矛盾，我们在这种情况下，只是无法在不同的参照系下观察到一致性的事件顺序。

我们经验上所说的「同时发生」，是因为光速太大，或者我们生活的尺度太小，所以同时是一种近乎同时。

相对论告诉我们，光速是物质移动的最大速度，信息传播的速度不可能超过这个速度。假如太阳消失了，地球上的我们也要在 8 分钟之后感知到太阳消失了。

也就是说，一个事件 A 发生后，载有这个事件信息的光（引力、无论什么，快不过光速）到达观察者之前，观察者是无法没有任何感知的，这时我们就无法定义事件的顺序。而当信息最终传播到达观察者时，这个事件也就对观察者发生了影响，造成了一个新的事件 B ，叫做「观察到了事件 A 的事件 B 」。这时我们才有 A happened before B 的因果关系。

可以看到，时空对于描述事件顺序的「happened before」同样是偏序关系。

因为事件 A 的发生，造成了事件 B 的发生（包括 B 这种观察 A 而发生的事件），那么 A 和 B 就存在因果关系。

Lamport 受相对论中事件顺序的相对性的启发，创造了 Lamport Logical Clocks。

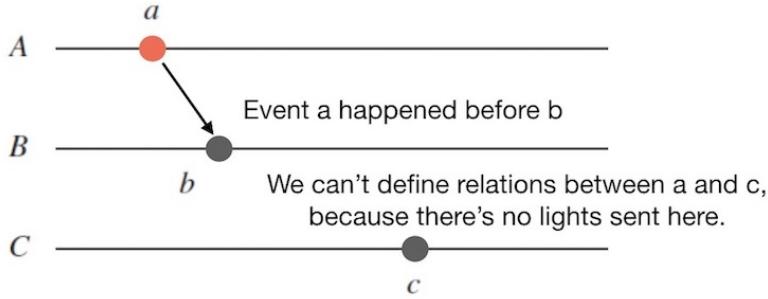
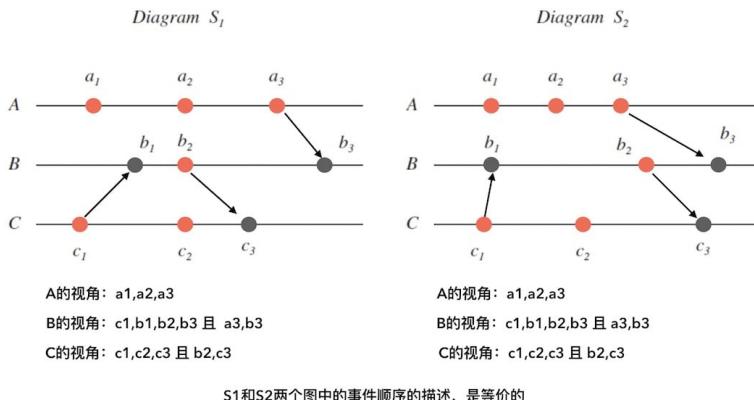


图 A.8: 物理时空中因果关系建立了事件顺序

图 A.9: S_1 和 S_2 虽然在物理时序上不一样, 但是在各个进程的视角上推导出来却是一样的

我们的分布式系统, 和相对论有很多相似之处:

- 在物理时空中, 信息是通过光速传播的, 而在分布式系统中, 信息是通过网络传播的。
- 在物理时空中, 不同参照系下的观察者可能对于事件顺序无法达成一致, 而在分布式系统中, 由于全局物理时钟无法实现, 不存在进程拥有全局视角。而如果进程间的事件没有因果关系, 那么就无法达成顺序上的一致性。
- 在物理时空中, 由于光速限制, 观察者在观察到事件 A 的时候, 才确定了事件 B 和 A 的因果关系。那么在分布式系统中, 我们同样可以通过消息传递来创造因果关系。

总体来说, 逻辑时钟尝试用「通过进程间创造通信以添加因果关系」的方式来对分布式中的事件顺序做描述。

下面, 我们来看下, 「通过通信创造因果关系」这个设计对于刻画分布式系统中的事件顺序有多么重要。

观察下图中的两个图:

- 红色的点表示自发性事件。黑色的表示『观察到其他进程事件』而发生的事件。
- 横向黑色实线代表物理时钟。
- 带箭头的线表示进程中一个事件发生时, 向另外一个进程传播这个事件。

我们试着从每个进程的视角, 依次对图 S_1 和 S_2 进行推导一下, 会发现, 其实两个图所描述的事件顺序, 在进程的相对视角中, 是一样的。

我们的逻辑时序应该越接近物理时序越好, 然而两个图对时序的刻画, 出现了歧义

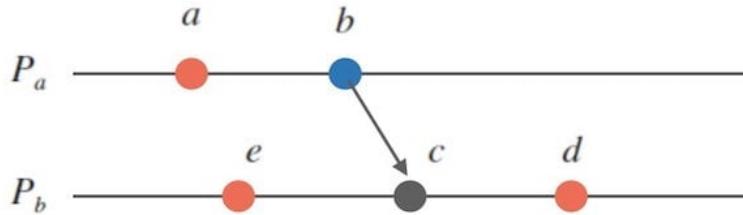


图 A.10: 分布式中事件的分类

(比如无法确定 a_3 和 b_2 的顺序)。根本上是没有做充分的消息传递来添加因果关系。

A.5 逻辑时钟 (Lamport's Timestamp)

现在，我们开始讨论 Lamport 的逻辑时钟算法。

首先我们需要明确一点：逻辑时钟并不度量时间本身，仅区分事件发生的前后顺序。

那么，「事件」是如何分类的：

1. 进程内自发的事件（如下图中的红色标记的事件）。
2. 发送一个消息，是一个事件（如下图中的蓝色标记的事件）。
3. 接收一个消息，是一个事件（如下图中的黑灰色标记的事件）。

我们上面有提到「happened before」的关系，我们知道，因果关系是推导「happened before」关系的重要一环：

if e_i causes e_j , then e_i must happen before e_j –Logical Time

道理是显然的：如果事件 e_i 导致了事件 e_j ，那么一定 e_i 发生在 e_j 之前。

我们现在给「happened before」这个关系一个记号： \rightarrow ，事件 a 在事件 b 之前发生则表示为 $a \rightarrow b$ ，那么我们有：

1. 如果 a 和 b 是同一个进程内的事件，并且 a 在 b 之前发生，则 $a \rightarrow b$ 。
2. 如果事件 a 是「发送了一个消息」，而事件 b 是接收了这个消息，则 $a \rightarrow b$ 。
3. 如果 $a \rightarrow b$ 并且 $b \rightarrow c$ ，那么 $a \rightarrow c$ （即传递性）。

那么，是否存在两个事件并无顺序关系吗？经过前面的讨论，答案当然是肯定的。

如果两个事件无法推导出顺序关系的话，我们称两个事件是并发的，记作 $a \parallel b$ 。

这样，我们可以这样描述上面的图中存在的事件顺序：

- $a \rightarrow b \rightarrow c \rightarrow d$
- $e \rightarrow c \rightarrow d$
- $a \parallel e$
- $b \parallel e$

可以看出，「happened before」 \rightarrow 是一个偏序关系。

Lamport 的时钟算法：

1. 每个进程 P_i 内维护本地一个计数器 C_i ，初始为 0.

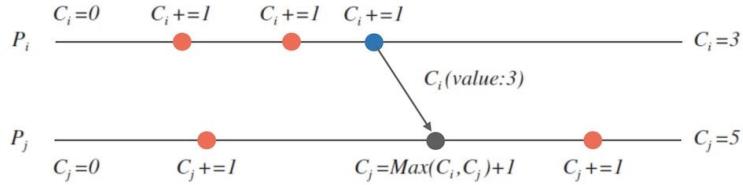
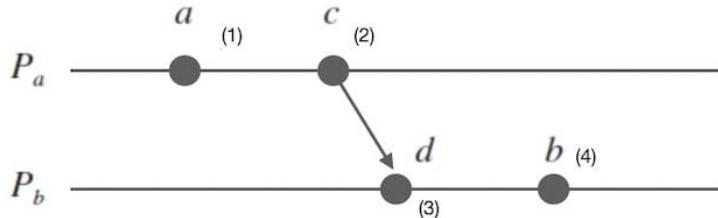


图 A.11: Lamport 时钟算法

图 A.12: 事件 a 和 b 在不同进程的情况下，中间一定有消息传递，否则两个事件并发

2. 每次执行一个事件，计数器 C_i 自增（假设自增量为 1）。
3. 进程 P_i 发消息给进程 P_j 时，需要在消息上附带自己的计数器 C_i 。
4. 当进程 P_j 接收到消息时，更新自己的计数器 $C_j = \max(C_i, C_j) + 1$ 。

下面的图是这个算法的示意图，可以推算最后的时钟计数器的值： $C_i = 3, C_j = 5$

下面，将证明：如果 $a \rightarrow b$ ，那么一定有 $C_a < C_b$ 。

1. 假设 a 和 b 发生在同一个进程中，显然 $C_a < C_b$ 。
2. 假设 a 和 b 分别处在不同的进程中，如 P_a 和 P_b ，根据事件先后的定义，必然存在一个不早于 a 且不晚于 b 的由 P_a 到 P_b 的通信（否则 $a \parallel b$, 矛盾）。那么假设两个进程在 a 和 b 之间最近一次通信是由 P_a 向 P_b 发送了消息 $a \rightarrow b$ ：易得 $a \rightarrow c \rightarrow d \rightarrow b$ （其中可能 $a = c$ 或者 $d = b$ ）。根据算法定义，得：
 - (a). $C_a \leq C_c$ （进程中计数器自增）。
 - (b). $C_d \leq C_b$ （进程中计数器自增）。
 - (c). $C_c < C_d$ （进程间通信，观察者事件已经严格大于发生者事件的计数器）。

那么，最终推导出 $C_a < C_b$ （严格小于）。

以上，得证 $a \rightarrow b \Rightarrow C_a < C_b$ 。

但是，如果我们已知 $C_a < C_b$ 的话，是否可以推导出 $a \rightarrow b$ 呢？

悲哀的是，不能。下面的图是个反例：

这样，我们反证了 $C_a < C_b \not\Rightarrow a \rightarrow b$ 。

我们无法推导出 $C_a < C_b \Rightarrow a \rightarrow b$ 的原因，在于 a 可能和 b 并发。

但是，如果 $C_a < C_b$ ，一定不会有 $b \rightarrow a$ 的关系存在。

Lamport 的逻辑时钟算法构建了一个全序 (total ordering) 时钟来描述事件顺序，但是我们知道「happened before」是一个偏序关系，用全序关系的一维计数器来描述「happened before」的话，就会导致无法等价化描述的结果，Lamport 时钟的缺陷在于：如果两个事件并不相关，那么这个时钟给出的大小关系则没有意义，这个缺陷其实恰好就是全序和偏序的不同点而已。

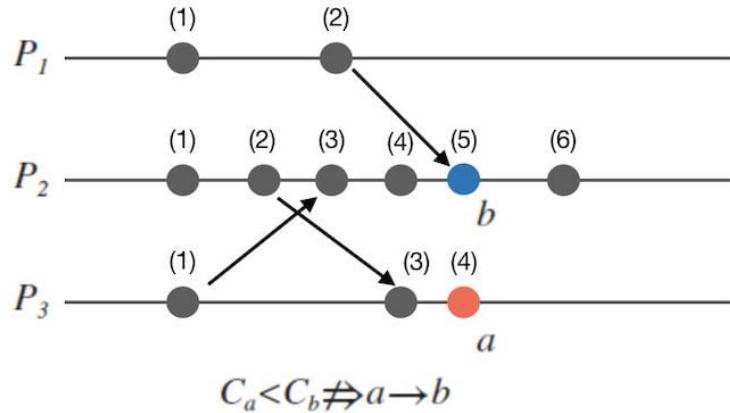


图 A.13: Lamport 时钟无法反向推导事件顺序的反例: 红色 a 和蓝色 b 是并发的

所以, 要准确描述事件顺序, 我们终究要寻求偏序方法。

于是, 我们继续探讨向量时钟。

A.6 向量时钟 (Vector Clocks)

向量时钟, 其实是对 Lamport 的时钟的一个延伸思考, 算法结构一致, 只是多传了一部分信息。

对每个进程, 定义一个向量 VC , 向量的长度是 n , n 是进程数目。

1. 初始化各个进程 P_i 的向量, 全部抹零: $VC_i = [0, \dots, 0]$ 。
2. 进程 P_i 每发生一个事件时, 其向量的第 i 个元素自增: $VC_i[i] += 1$ 。
3. 当进程 P_i 发消息给进程 P_j 时, 需要在消息上附带自己的向量 VC_i 。
4. 当进程 P_j 接收到消息时, 对齐对方的时钟, 并在自己的时钟上自增: 对 $[0, n)$ 上的任意一个整数 k 执行 $VC_j[k] = \max(VC_j[k], VC_i[k])$, 接着, 对应第 2 点: $VC_j[j] += 1$ 。

下面图是一个向量时钟算法的示意图:

1. 和 Lamport 时钟算法示意图一样: (红色点、蓝色点、黑灰色点...)
2. 图的右方部分, 总结了这个算法对不同事件的操作:
 - (a). 对于红色和蓝色, 也就是进程内自发性事件和发送消息的事件, 向量内相应的计数器自增。
 - (b). 发送消息的时候, 需要传播出去自己的整个向量 (也就是广播自己对整个系统的视角)。
 - (c). 接收到消息的时候, 也就是蓝色事件, 需要对齐对方的向量, 并应用第一条规则, 即自己的向量内相应计数器自增。

可以发现, 向量时钟和 Lamport 的时钟算法结构一样, 不同的点在于: Lamport 时钟只是在对齐时钟的计数器, 而向量时钟是在对齐各自对整个系统的视角。

我们可以推导出来关于向量时钟比较大小的几个性质:

1. **向量的各维相等, 则向量相等。** 这个是显而易见的。
2. **向量时钟是有序的 (充要), 即: VC_i 的各维上的值不大于 VC_j 对应维上的值, 则**

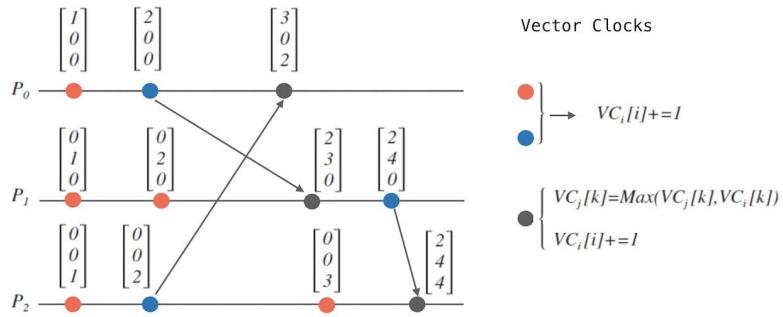
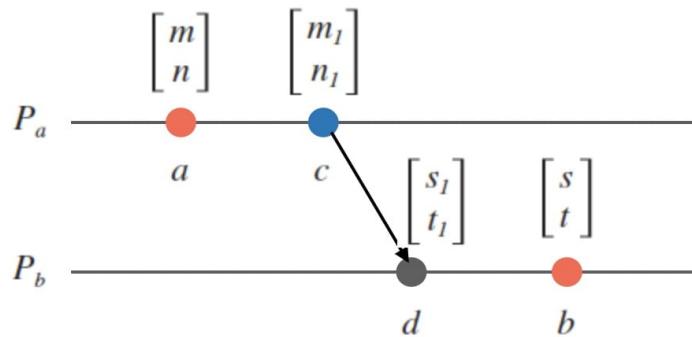


图 A.14: 向量时钟示意图

图 A.15: 中间必然存在 P_a 向 P_b 发送了消息

认为 VC_i 不大于 VC_j 。

3. 向量时钟有序性质的进一步细化, 定义了严格小于: 如果 VC_i 不大于 VC_j , 并且至少存在一个维, 在这个维上 VC_i 的值严格小于 VC_j 在这个维上的值, 则认为 VC_i 小于 VC_j 。
4. 如果两个向量不存在大小关系, 则认为两个向量平行, 记作 $VC_i \parallel VC_j$.

这几个性质看起来复杂, 其实都是在定义向量时钟之间的大小关系。我们可以得出一个结论:

向量时钟之间的大小关系是一种偏序关系。

和 Lamport 时钟一样, 我们可以利用类似的推导方式, 证明 $a \rightarrow b \Rightarrow VC_a < VC_b$ 。这里不再描述证明思路。

我们接下来要花一定篇幅论证下 $VC_a < VC_b \Rightarrow a \rightarrow b$ 。

1. 如果 a 和 b 两个事件处在同一个进程中, 显而易见 $a \rightarrow b$ 。
2. 假设 a 和 b 分别处在不同的进程中, 如 P_a 和 P_b , 设 $VC_a = [m, n], VC_b = [s, t]$ 。因为 $VC_a < VC_b$, 所以 ms , 所以必然在不早于 a 之前和不晚于 b 之后的时间内, P_a 向 P_b 发送了消息 (否则 P_b 对 P_a 的计数器得不到及时刷新, s 就不会不小于 m)。实际上, 可以分为以下几种情况:

1. 当 $a = c$ 且 $d = b$, 易得 $a \rightarrow b$.
2. 当 $a = c$ 且 $d \rightarrow b$, 由传递性, 得 $a \rightarrow b$.
3. 同样对于 $d = b$ 且 $a \rightarrow c$ 的情况.
4. 当 $a \rightarrow c$ 且 $d \rightarrow b$, 根据进程内的算法逻辑和传递性, 也很容易得出结论。

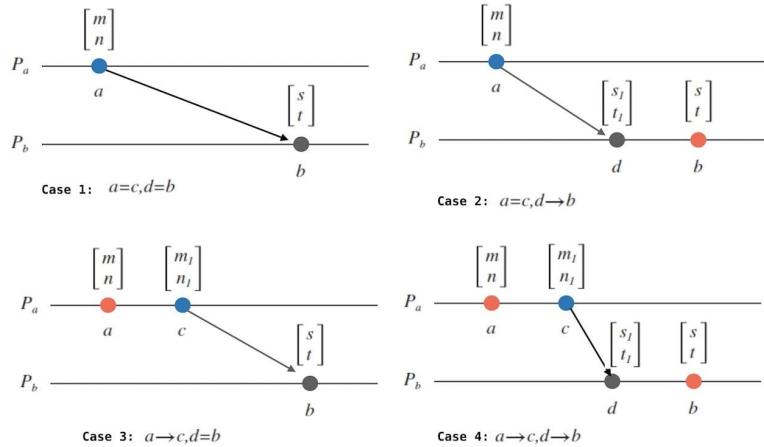


图 A.16: 可能出现的 4 种情况

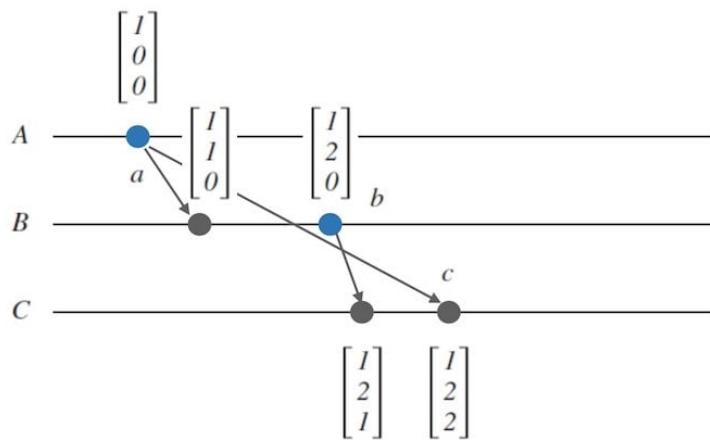


图 A.17: 用向量时钟的方法解答文章开始提出的问题

综上: $VC_a < VC_b \Rightarrow a \rightarrow b$ 得证。

进一步的, 我们可以得出这样的结论:

1. 向量有序, 则事件有序 (充要): $VC_a < VC_b \leftrightarrow a \rightarrow b$
2. 向量平行, 则事件并发 (充要): $VC_a \parallel VC_b \leftrightarrow a \parallel b$

是的, 向量时钟可以准确刻画事件顺序。

其本质在于将 Lamport 时钟的全序计数器的方式改造成向量时钟的偏序大小关系。

A.7 向量时钟看前面的问题

现在我们回到文中一开始提到的问题。

由于网络延迟的不确定性, 我们在文章开始提出了一个 A, B, C 三个进程中如何确定事件顺序的问题。

我们可以从下面的图中看出, $VC_a < VC_b < VC_c$, 那么可以确定 $a \rightarrow b \rightarrow c$ 。也就是我们找到了一种方法来描述这种情况下的事件顺序。进程 C 的视角下观察 a 和 b 的顺序的问题也有了明确的答案。

最后, 我们回到朋友圈的例子。下面图中可以看出, $VC_a < VC_b < VC_c < VC_d <$

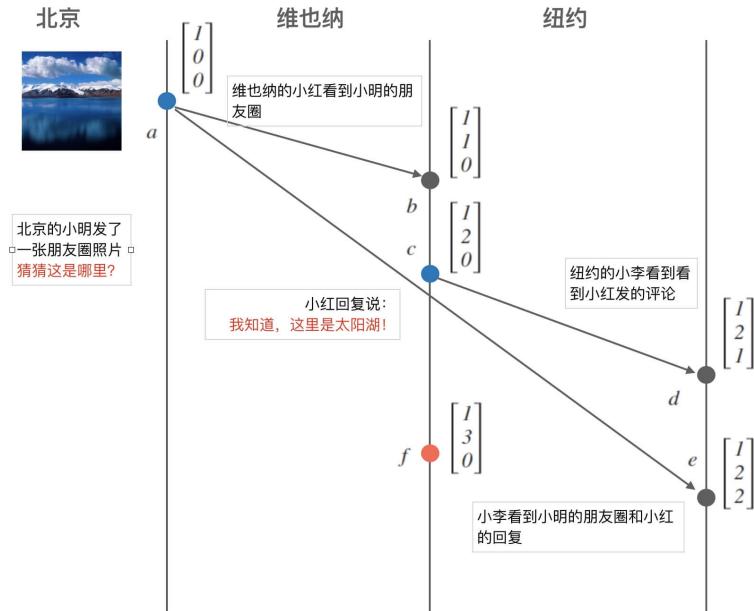


图 A.18: 用向量时钟的方法看朋友圈问题

VC_e , 显然可以确定 $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$ 。

在小李看到小明的朋友圈和小红的评论的时候, 也收到了这两条数据的向量, 我们可以根据向量时钟来确定事件的先后关系, 从而不会显示出因果矛盾。

上图中还有一个事件 f , 它的发生可能是小红又发了一条评论。我们可以看到 $VC_f \parallel VC_e$ 。这时候, 无法确定事件 f 和 e 的先后关系, 也就是说 $f \parallel e$ 。

但是, 这时候小李还没有看见这个事情。所谓「因果」: 有因才有果。看见也是一个事件的果。这样说, f 和 e 没有因果关系, 因为小李还没有看见这个事件。所以讨论也就没有意义。

A.8 向量时钟的不足

- 只考虑了固定数量的节点, 没有考虑节点的动态添加和销毁。
- 假设节点数量是 N , 那么每个节点需要维护的空间复杂度是 $O(N)$ 。通信的信息量的复杂度也是 $O(N)$ 。

A.9 总结与感想

- 现实中, 无法构建精确的全局时钟来描述事件顺序。
 - 受狭义相对论的启发, 我们用因果关系来描述事件顺序。
 - 因果关系是一种偏序关系。
 - Lamport 时钟构造的计数器之间的大小关系是一种全序关系, 无法准确刻画事件顺序的偏序关系。
 - 向量时钟是一种对 Lamport 时钟的延伸, 以偏序关系准确刻画了事件的因果顺序。
- 此外, 向量时钟给我一种感想, 对每个分布式节点来说:

- 我把我的视角分享给其他节点。
- 我对齐我看到的其他节点的视角。

本质上，是在做**视角对齐**。

这自然地，让我想起了 Gossip 谣言传播算法。Gossip 算法如其名，每个分布式参与者都在散播自己视角的信息，以达到谣言扩散的效果。同样是在做视角对齐。

对于分布式系统中的事件顺序的刻画，就讨论到这里。