FlinkNote

> 参考书籍
>
> - Stream Processing with Apache Flinkhttps://www.oreilly.com/library/view/stream-processing-with/9781491974285/
> - 《基于Apache Flink的流处理》https://book.douban.com/subject/34912177/

> 注：本文主要是针对《基于Apache Flink的流处理》1-8章的笔记

第1章 状态化流处理概述

Apache Flink 是一个**分布式流处理引擎**，具有直观而富于表现力的API来实现有状态流处理应用程序。

1.1 传统数据处理架构

几十年来，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，很多公司都设计和构建了**管理**这些**数据**的**基础架构**。

大多数企业实现的**传统架构 区分**了**两种类型**的**数据处理**

- 事务型处理
- 分析型处理

1.1.1 事务型处理

公司在日常业务活动中使用各种应用程序，如企业资源规划(ERP)系统、客户关系管理(CRM)软件和基于网络的应用程序。这些系统通常设计有独立的**数据处理层(应用程序本身)**和**数据存储层(事务数据库系统)**，如图1-1所示。

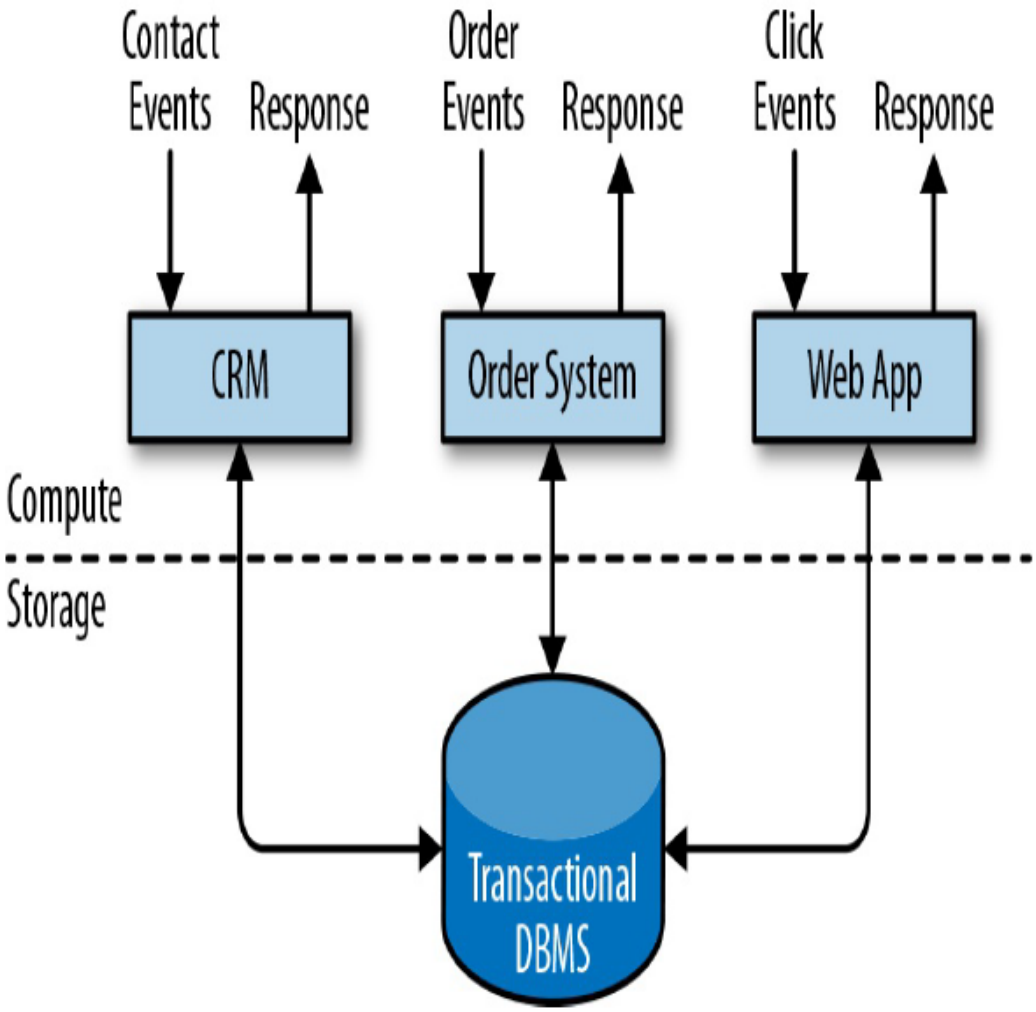


Figure 1-1. Traditional design of transactional applications that store data in a remote database system

当应用程序需要更新或扩展时，这种应用程序设计可能会导致问题。

克服应用程序之间紧耦合的最新方法是**微服务**设计。微服务被设计成**小型**、**独立**的应用程序。他们遵循**UNIX哲学**：**做一件事并把它做好**。更复杂的应用程序是通过将几个微服务相互连接而构建的，这些微服务只**通过标准接口进行通信**，如RESTful HTTP连接。因为微服务彼此严格分离，并且只通过定义明确的接口进行通信，所以每个微服务都可以用不同的技术堆栈来实现，包括编程语言、库和数据存储。微服务和所有必需的软件和服务通常被打包并部署在独立的容器中。图1-2描述了一个微服务架构。

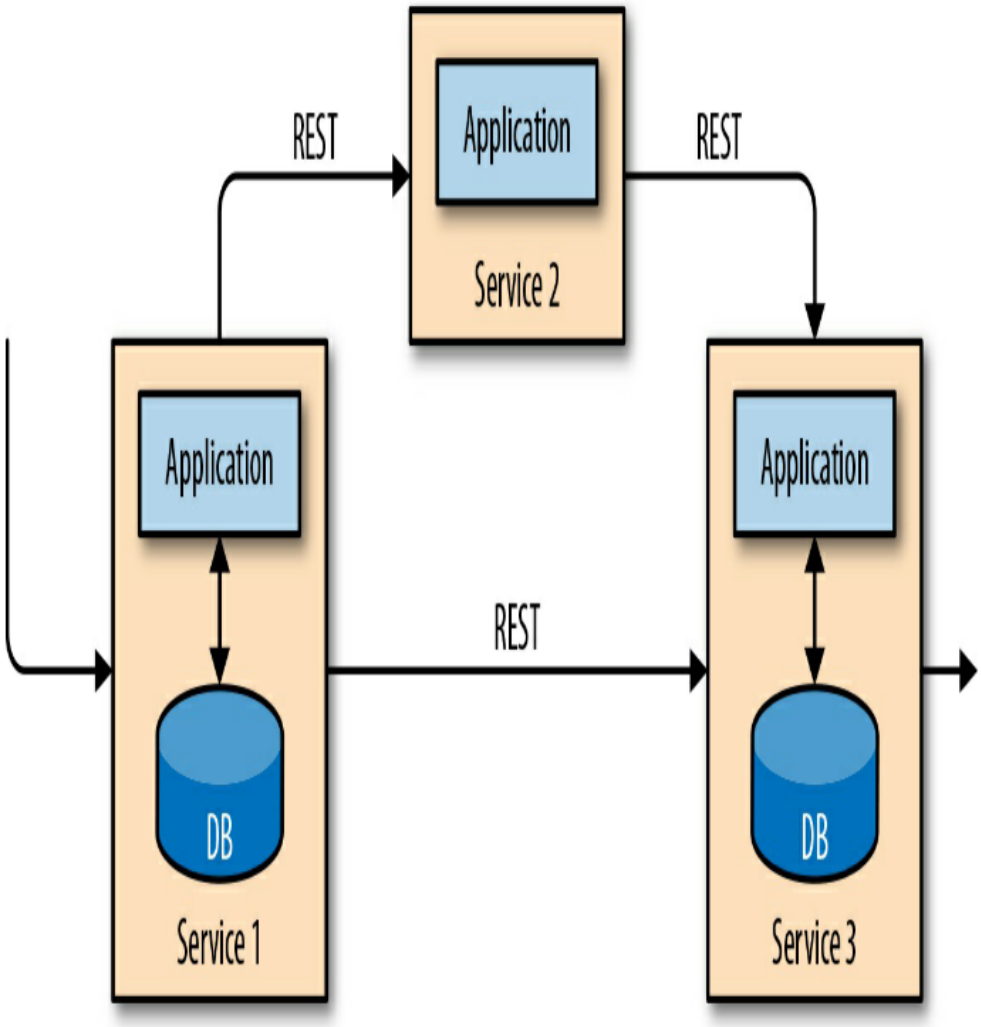


Figure 1-2. A microservices architecture

1.1.2 分析型处理

存储在公司各种**交易数据库**系统中的**数据**可以**提供**关于公司业务运营的有**价值**的见解。然而，事务性数据通常分布在几个不相连的数据库系统中，如果可以**联合分析**，事务性数据更有价值。

对于**分析类查询**，数据通常**复制到数据仓库**中，而不是直接在事务数据库上运行分析查询，数据仓库是用于分析查询工作的专用数据存储。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到数据仓库中。将数据复制到数据仓库的过程称为提取-转换-加载(extract–transform–load,ETL)。

ETL过程

1. 从事务性数据库中**提取**数据，
2. 将其**转换**为通用表示形式，可能包括数据验证、数据规范化、编码、去重和表模式转换，
3. 最后将其加载到分析性数据库中。

ETL过程可能非常**复杂**，通常需要技术上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**。

一旦数据被导入数据仓库，就可以对其进行查询和分析。通常，数据仓库上查询分为两类

1. 第一类是定期报告查询
2. 第二种类型是即席查询（ad-hoc query）

数据仓库以**批处理**方式执行这**两种查询**，如下图1-3所示

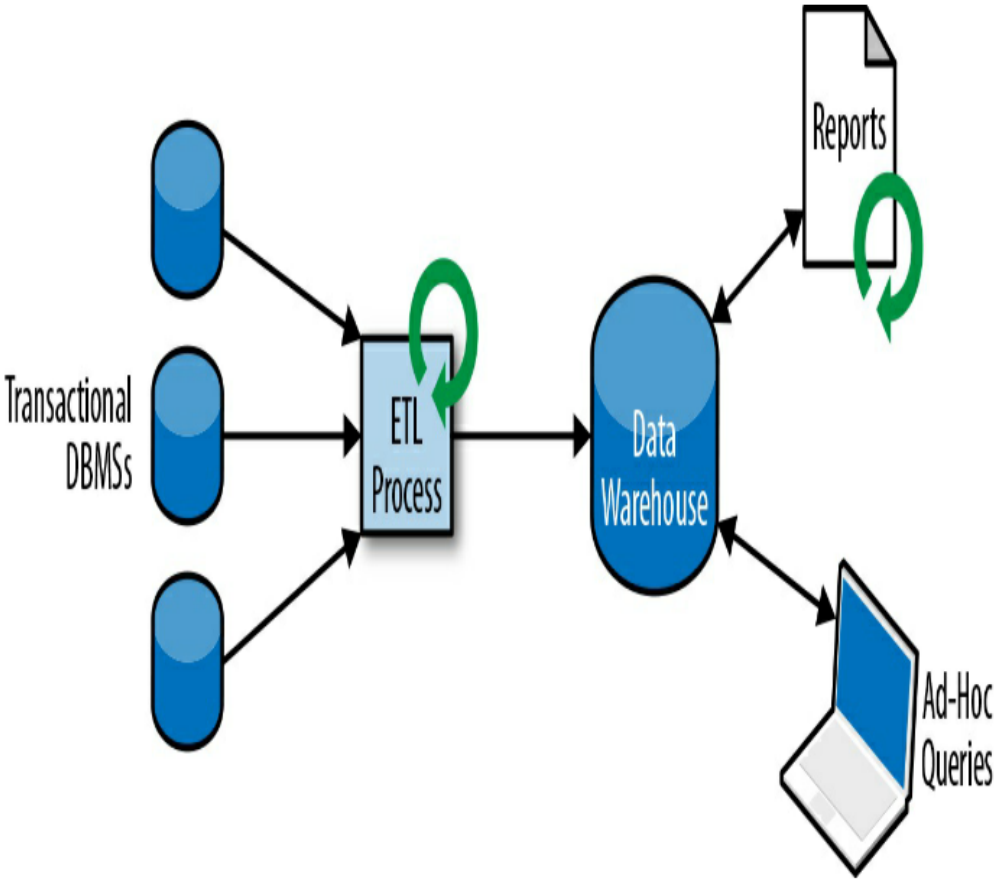


Figure 1-3. A traditional data warehouse architecture for data analytics

如今，Apache Hadoop生态系统是许多企业的信息技术基础架构中不可或缺的一部分。

- 大量**数据**(如日志文件、社交媒体或网络点击日志)被**存储在**Hadoop的分布式文件系统(**HDFS**)、**S3**或其他大容量数据存储区(如Apache **HBase**)，而不是将所有数据存储在关系型数据库系统，Apache Hbase以较小的成本提供了巨大的存储容量。
- 驻留在这种存储系统中的数据可以**通过Hadoop**上的**SQL引擎**进行**查询和处理**，例如**Apache Hive**、**Apache Drill**或**Apache Impala**。
- 这就是前文描述的**传统架构**的一种实践

1.2 状态化流处理

任何**处理事件流**并且不只是简单的只依赖一条事件的**应用程序**都需要是**有状态**的，具有**存储**和**访问**中间数据的能力。

- 当**应用程序接收到事件**时，它可以**依赖状态**来执行任意**计算**，包括从状态中读取数据或向状态中写入数据。
- 原则上，状态可以在**许多不同的存储方案**，包括程序变量、本地文件、嵌入式或外部数据库。

Apache Flink将应用程序**状态**存储在**本地内存**或**本地嵌入式数据库**中并且会**定期**向**远程持久化存储**来**同步**

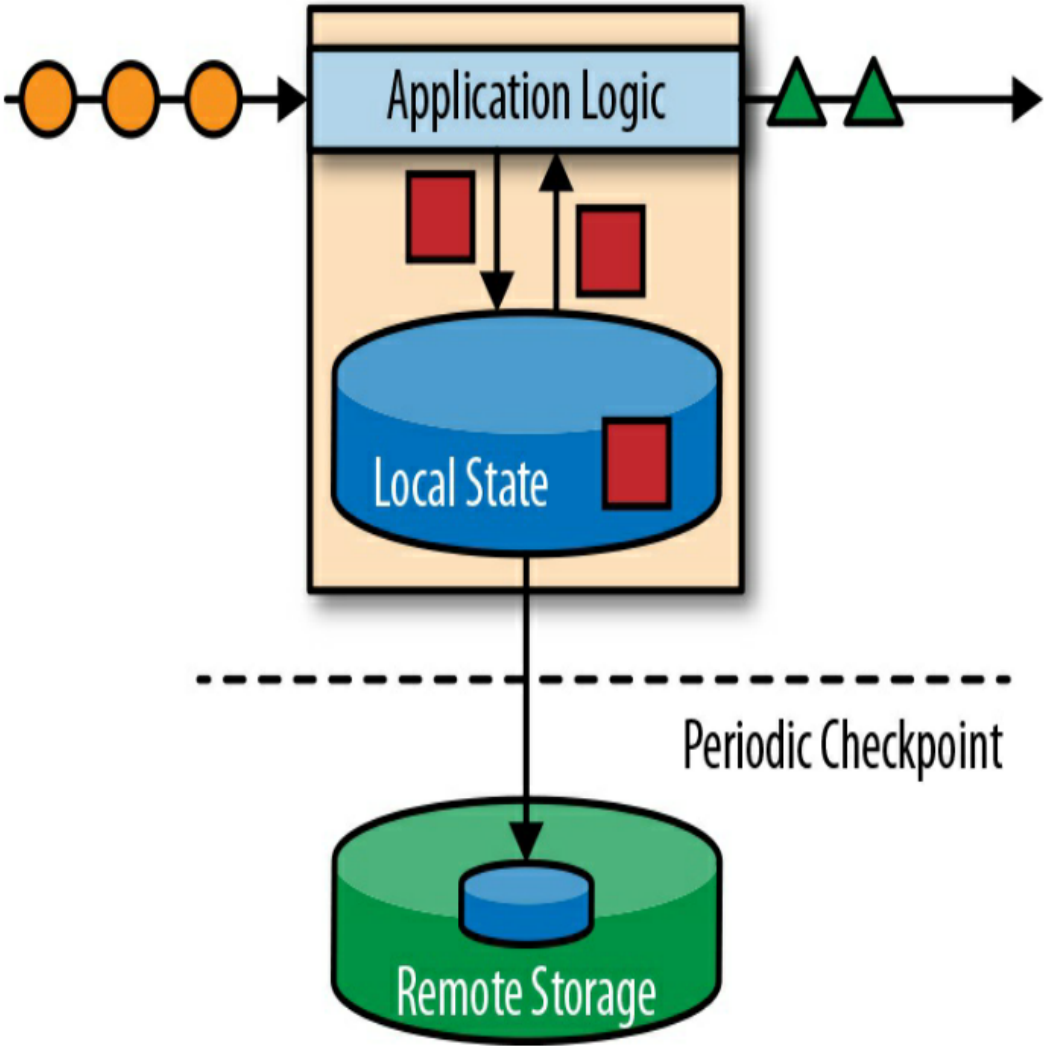


Figure 1-4. A stateful streaming application

有状态流处理应用程序通常从**事件日志**中**获取**它们的输入事件。**事件日志**负责**存储**和**分发**事件流。事件被写入持久的、只能追加的日志，这意味着不能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**使用者 多次读取。**由于**日志**的**仅支持追加**，事件总是**以完全相同的顺序**发布给所有使用者。有几个很好的**开源事件日志系统**，**Apache Kafka**是最受欢迎的。

出于多种原因，将Flink上的**有状态流应用程序**和**事件日志系统**来**配合**使用效果很棒。

- 在这种体系结构中，**事件日志**用来**持久化 输入**事件，并可以按确定的顺序**重放**它们。
- 在**出现故障**的情况下，Flink通过**从以前的检查点恢复状态**并**重置事件日志上的读取位置**来恢复有状态的流应用程序。

如前所述，有状态流处理是一种通用和灵活的设计架构，可以用于许多不同的用例。

在下文中，我们介绍了通常使用状态流处理实现的三类应用程序:

1. 事件驱动型应用程序
2. 数据管道型应用程序
3. 数据分析型应用程序

1.2.1 事件驱动型应用

事件驱动型应用程序是**有状态的流应用程序**，它们**接收事件流**并**使用**特定于应用程序的**业务逻辑**来**处理**事件。

事件驱动型应用程序是**微服务**的演变和**发展。**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。**图1-5显示了由事件驱动的流应用程序组成的服务架构。

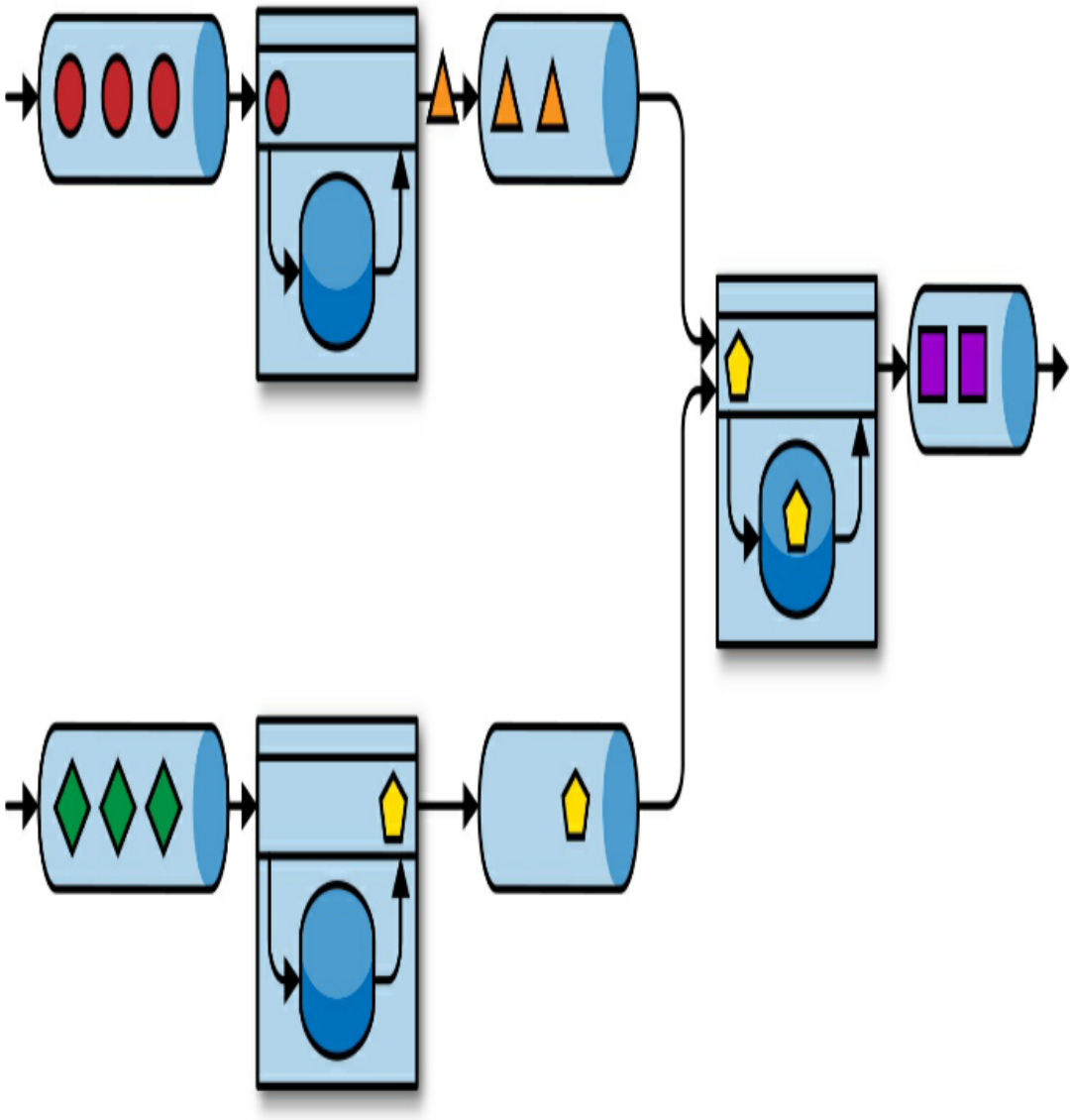


Figure 1-5. An event-driven application architecture

图1-5中的应用程序由**事件日志**相互**连接**。

- 一个**应用程序**将其**输出**发送到**事件日志**，另一个**应用程序**使用另一个应用程序发送的事件（从**事件日志**上**读取**）。
- 事件日志分离发送方和接收方，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，并且可以在**本地**管理自己的状态。
- 应用程序也可以**独立**部署和扩容。

事件驱动的应用程序对运行它们的**流处理引擎**有**很高要求**。并非所有的流处理器都同样适合运行事件驱动的应用程序。应用编程接口的可表达性以及状态处理和事件时间支持的质量决定了可以实现和执行的业务逻辑。Apache Flink是运行这类应用程序的非常好的选择。

1.2.2 数据管道

当今的大公司IT架构包括许多**不同种类的数据存储**，如关系型数据库系统、事件日志系统、分布式文件系统、内存缓存和搜索索引。为了在各自的应用场景下提供最佳性能表现，所有这些系统都以**不同的格式**和数据结构**存储数据**。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，网上商城提供的商品信息可以存储在事务数据库、网络缓存和搜索索引中。由于这种**数据复制**，**数据存储必须保持同步**。

同步不同存储系统中数据的传统方法是定期ETL作业，但是延时性太高了。另一种方法是使用**事件日志**来**分发更新**。更新由事件日志写入和分发。**日志的使用者**（也就是基于**Flink**的某个应用）将**更新同步**到需要同步的数据存储中，我们把这类应用称为**数据管道**。**用Flink来实现一个数据管道应用是很方便的**，它可以支持对不同种类数据存储的读写，而且可以在短时间处理大量数据。

1.2.3 流式分析

流分析应用程序**持续接收事件流**，并通过**低延迟**地合并最新事件来**更新结果**。通常，流式应用程序将结果存储在支持高效更新的外部数据存储中，如数据库或键值存储。流式分析应用程序的实时更新结果可以在仪表板应用程序中实时地看到，如图1-6所示。

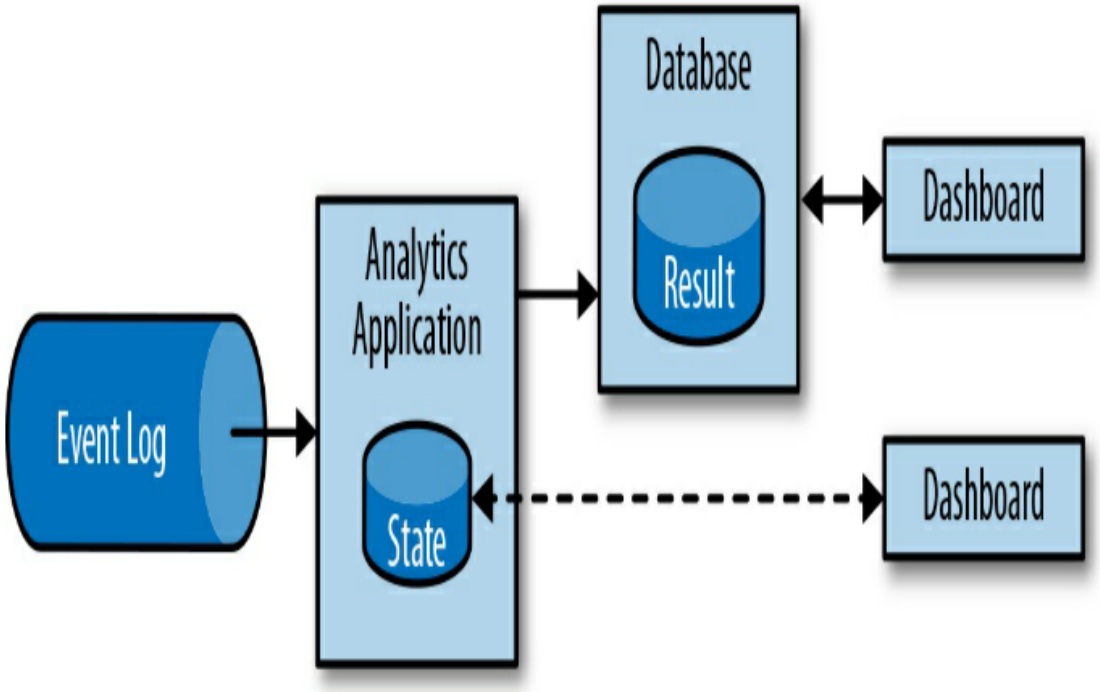


Figure 1-6. A streaming analytics application

可以运行有状态流应用程序的**流处理引擎 负责所有处理步骤**，包括事件摄取、包括状态维护在内的连续计算以及更新结果。

值得一提的是，Flink还支持对于数据流的分析型SQL查询

1.3 开源流处理的演变

略

1.4 Flink 快览

####

Apache Flink是很棒的第三代分布式流处理器。它以高吞吐量和低延迟大规模提供精确的流处理。

以下**特性**让Flink脱颖而出:

- 同时支持**事件时间**和**处理时间**语义。
- 提供**精确一次**(exactly-once)的状态一致性保证。
- 每秒处理数百万个事件时的**毫秒级延迟**。Flink应用程序可以扩展到在数千个内核上运行。
- 分层设计的API
- 可以轻松**连接**到最常用的**存储系统**，如Kafka、Cassandra、Elasticsearch、JDBC、HDFS、S3等。
- 能够全天运行流式应用程序，**宕机时间**非常**少**
- 能够在**不会丢失应用程序的状态**的**前提**下，**更新**作业的应用程序代码或者将作业迁移到不同的Flink集群，。
- 提供详细的**指标**
- 支持**批处理**
- 对开发者友好。API极为易用。并且**嵌入式执行模式**可以将应用连同整个Flink都嵌入到单个JVM进程中，**方便在IDE里运行和调试**基于Flink的应用

第2章 流处理基础

本章的目标是介绍流处理的基本概念以及对其处理框架的要求。

2.1 Dataflow编程概述

2.1.1 Dataflow图

Dataflow程序通常表示为**有向图**，

- 其中**节点**称为**算子**，代表计算，**边**代表**数据依赖**。
- **算子**是**数据流应用程序**的**基本功能单元**。它们从输入中获取数据，对数据进行计算，然后将数据输出到输出端进行进一步处理。

- **没有输入端**的算子称为**数据源**，**没有输出端**的算子称为**数据汇。**
- **数据流图**必须**至少**有**一个数据源**和**一个数据汇。**

图2-1显示了一个数据流程序，它从文章的输入流中提取并计数一些标签。

image-20201029121349342

像图2-1中的数据流图被称为逻辑图，因为它们传达了计算逻辑的高级视图。为了执行数据流程序，它的逻辑图被转换成物理Dataflow图，该图详细说明了程序是如何执行的。例如，如果我们使用分布式处理引擎，**每个 算子**可能有**几个**并行**任务**在不同的物理机器上运行。

图2-2显示了图2-1的逻辑图的物理数据流图。在**逻辑Dataflow图**中，节点代表**算子**，而在**物理Dataflow图**中，节点代表**任务。**每个任务负责计算一部分的输入数据。

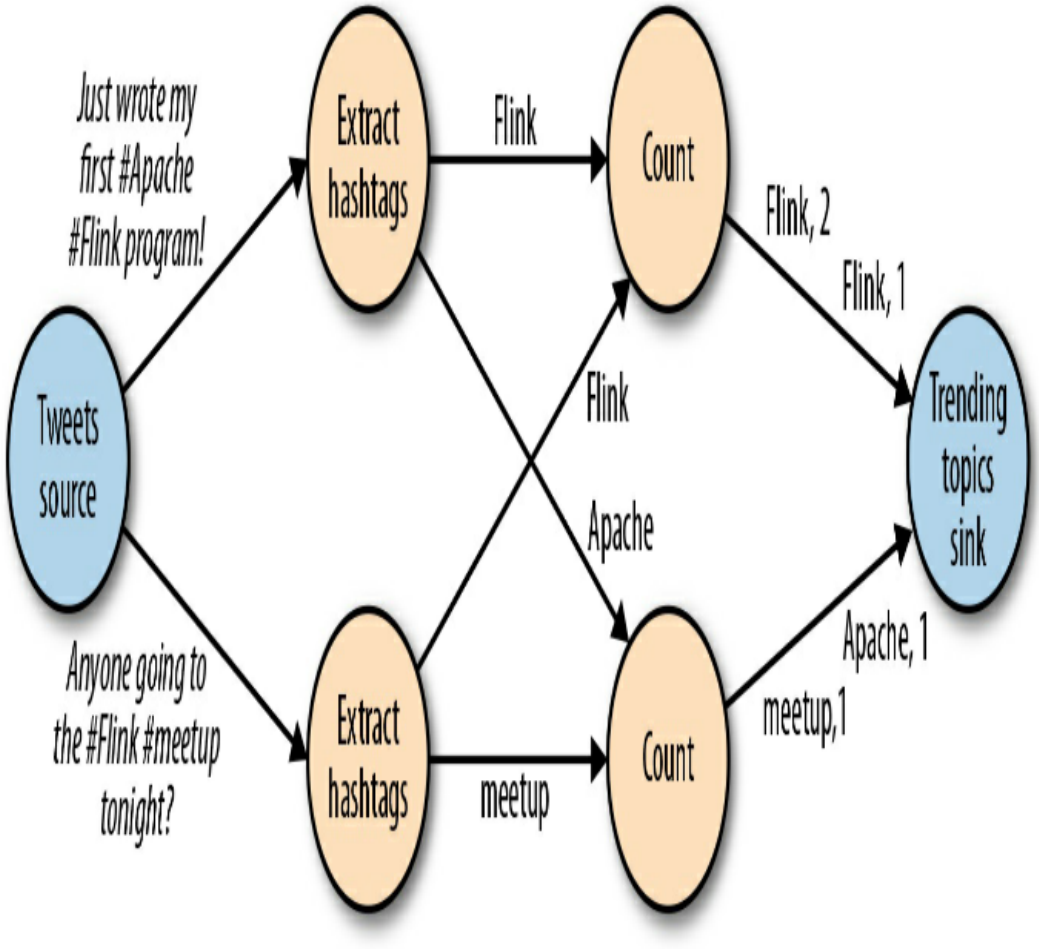


Figure 2-2. A physical dataflow plan for counting hashtags (nodes represent tasks)

2.1.2 数据并行和任务并行

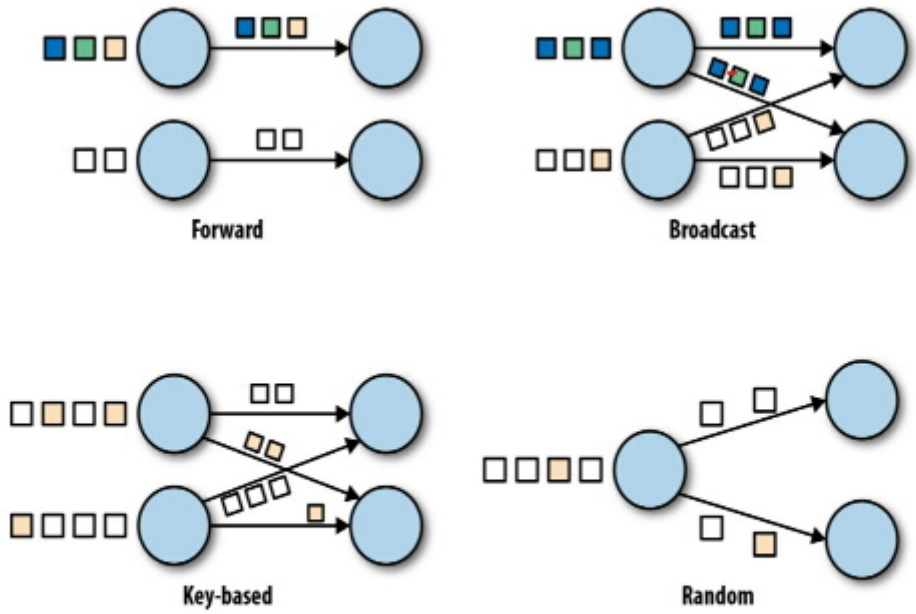
可以以不同的方式利用数据流图中的并行性。

首先，可以对**某个算子**的**输入数据**进行**分区**，并**在数据子集**上**并行执行相同操作**的任务。这种类型的并行称为**数据并行。**数据并行非常有用，因为它允许将大量计算数据分布到多个不同的物理节点上并行执行。

其次，可以让**不同算子**的**任务 并行执行**相同或不同数据的计算。这种类型的并行称为**任务并行。**使用任务并行，可以更好地利用集群的计算资源。

2.1.3 数据交换策略

数据交换策略定义了**数据项如何被分配给物理Dataflow图中的不同任务。**在这里，我们简要介绍一些常见的数据交换策略，如图2-3所示。



- **转发策略**在**发送端任务**和**接收端任务**之间**一对一**地进行**数据传输。**如果两个任务位于同一个物理机器上(这通常由任务调度器来保证)，这种交换策略避免了网络通信。
- **广播策略**将**每个数据项**发送给算子的**所有并行任务。**因为这种策略复制数据并涉及网络通信，所以成本相当高。
- **基于键值的策略**通过键属性**划分数据**，并保证具有**相同键的数据项**将**由相同的任务处理。**
- **随机策略**将数据项**均匀**地**随机**分配给**任务**，以便**负载均衡**

2.2 并行流处理

下面看看如何将Dataflow的概念运用到**并行数据流处理**中。我们先给出数据流的定义：**数据流是一个长度可能无限长的事件序列**

数据流的例子如下：监控器产生的监控数据、传感器产生的测量数据、信用卡交易数据、气象站观测数据、搜索引擎搜索记录等

2.2.1 延迟和吞吐

对于批处理应用程序，我们通常关心作业的总执行时间，或者我们的处理引擎读取输入、执行计算和写回结果需要多长时间。由于流应用程序连续运行，并且输入可能是无限的，因此在**流处理**中**没有总执行时间的概念。**取而代之的是，**流处理**必须尽可能**快**地为传入数据**提供结果**（延迟），同时还要应对**很高的事件输入速率**（吞吐）。我们用延迟和吞吐来表示这两方面的性能需求。

2.2.1.1 延迟

延迟表示**处理一个事件所需的时间。**本质上，它是接收事件到在输出中看到事件处理效果的时间间隔。

在数据流中，延迟以时间为单位进行衡量，例如**毫秒。**根据应用程序的不同，可能会关心**平均延迟**、**最大延迟**或**百分比延迟。**例如，10ms的平均延迟意味着平均在10ms内处理事件。或者，10毫秒的95%延迟值意味着95%的事件在10毫秒内得到处理。

像Apache Flink这样的现代流处理引擎可以提供低至几毫秒的延迟。

2.2.1.2 吞吐

吞吐量是对**系统处理能力**的一种**度量**——它的**处理速率。**也就是说，**吞吐量**告诉我们系统**每单位时间可以处理多少个事件。**

需要注意的是，处理的速率取决于事件到达速率；低吞吐量不一定表示性能差。在流式系统中，通常希望确保系统能够处理**最大的预期事件到达速率。**也就是说，主要关心的是**确定峰值吞吐量**，即系统处于最大负载时的性能限制。

一旦事件到达速率超过了预期的最大值，我们就不得不开始缓冲事件。如果系统继续以**超过其处理能力**的接收速率**接收数据**，缓冲区可能会变得不可用，数据可能会丢失。这种情况通常被称为**背压。**

2.2.1.3 延迟与吞吐

此时，应该清楚的是，**延迟**和**吞吐 不是独立的指标。**

- 如果事件需要很长时间才能在数据处理管道中传输，我们就无法轻松确保高吞吐量（延迟影响了吞吐）。
- 同样，如果系统的处理能力过低，事件将被缓冲，必须等待才能得到处理（吞吐影响了延迟）。

降低延迟可提高吞吐量。如果一个系统可以更快地执行操作，它可以在相同的时间内执行更多的操作。而一个很好的方式就是**并行处理**

2.2.2 数据流上的操作

流处理引擎通常提供一组**内置操作**来接收、转换和输出数据流。这些**操作**可以用来**构成Dataflow图**来**代表流式应用的逻辑。**在本节中，我们将介绍最**常见的流式操作。**

操作可以是无状态的，也可以是有状态的。

- **无状态**操作**不维护任何内部状态。**也就是说，**一个事件的处理 不依赖于 任何历史事件**，也不保留历史数据。无状态操作很**容易并行化。**
- **有状态**操作会**维护他们以前接收到的事件的信息。状态**会**通过传入的事件**来**更新**，并且**在未来事件**的**处理逻辑中使用。**有状态流处理应用程序在**并行化**和**容错**方面**更具挑战性**

2.2.2.1 数据接入和数据输出

数据接入和数据输出操作允许流处理器与外部系统通信。

数据接入是从**外部系统 获取**原始数据并将其**转换**为适合处理格式的操作。实现**数据接入逻辑**的**算子**称为**数据源。**

数据输出是以适合**外部系统**使用的形式产生**输出**的操作。实现**数据输出**的**算子**称为**数据汇**，

2.2.2.2 转换操作

转换操作是**单程操作**（single-pass），**每个事件**都**独立处理。**操作**一个接一个**地**处理事件**，并对事件数据进行一些转换，产生一个新的输出流。一般来说，转换操作比较简单，**不用维护内部状态**

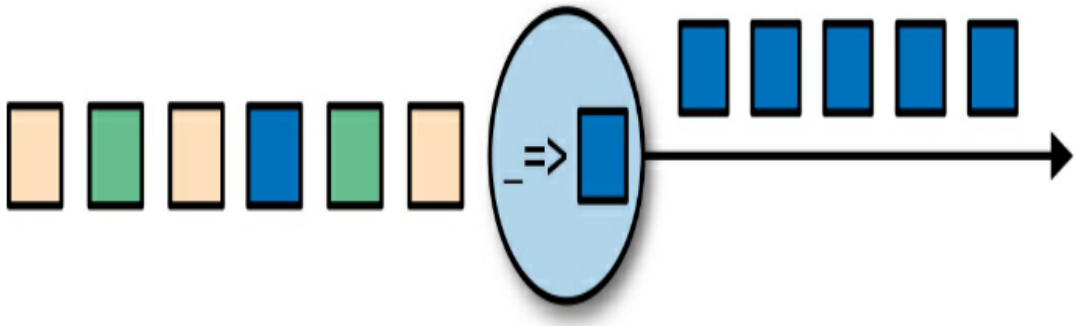


Figure 2-4. A streaming operator with a function that turns each incoming event into a darker event

转换操作的算子可以接受多个输入并产生多个输出流。他们还可以通过将一个流分成多个流或将多个流合并成一个流来修改数据流图的结构。

2.2.2.3 滚动聚合

滚动聚合是针对**每个输入事件**不断**更新**的**聚合操作**，比如**总和**、**最小值**和**最大值**。**聚合操作**是**有状态**的，并将**当前状态**与**传入事件相结合**以**生成新的聚合值**。图2-5显示了一个滚动最小聚合。操作符保持当前的最小值，并针对每个传入事件相应地更新它。

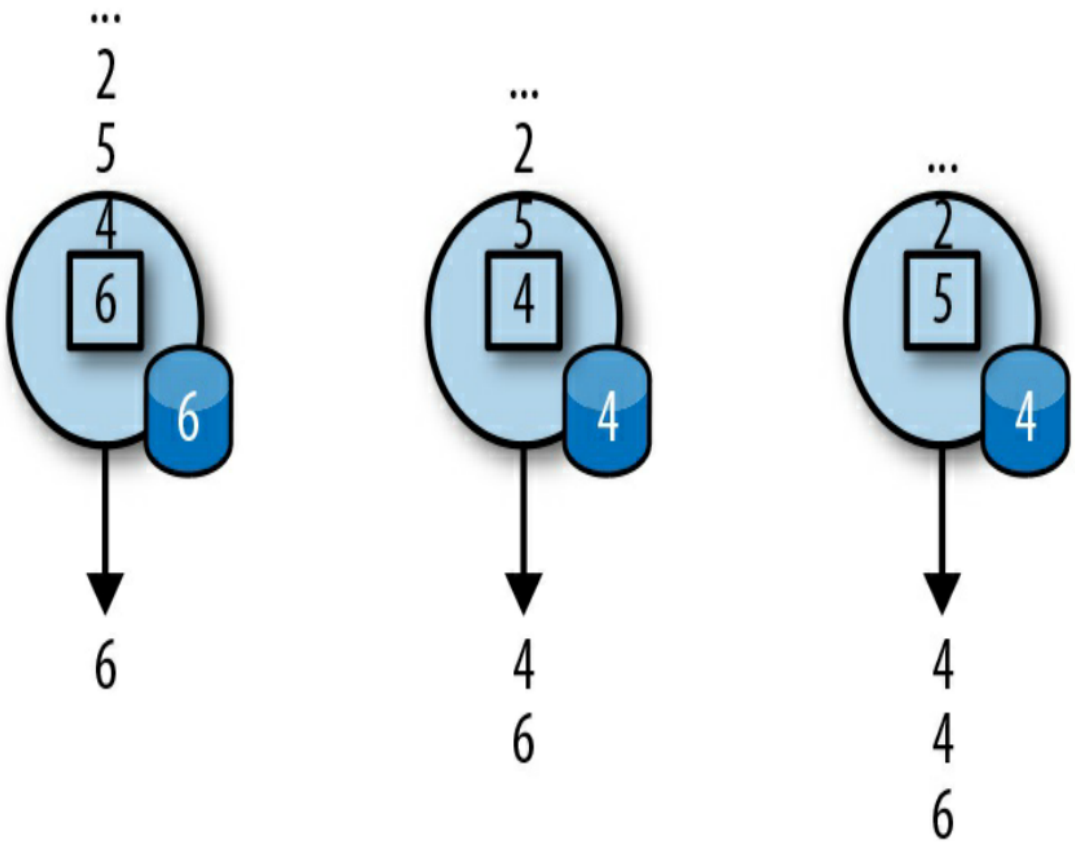


Figure 2-5. A rolling minimum aggregation operation

2.2.2.4 窗口操作

转换和**滚动聚合 每次处理一个事件**，以生成输出事件并更新状态。但是，**有些操作**必须**收集**和**缓存**事件。例如**求中位数的函数。**为了在**无限流**上**高效地计算这些操作**，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。**

窗口还支持在**数据流**上**进行一些有趣的查询。**例如：如果有一个为司机提供实时交通信息的应用程序。在这个场景中，您想知道在过去几分钟内某个位置是否发生了拥堵。这时候我们只关注过去几分钟这个窗口的数据。

窗口操作不断地从一个**无界事件流**中**创建 长度有限的事件集**(称为**桶**)，并让我们对这些**桶 执行计算。**事件通常根据数据属性或时间分配到桶中。**窗口的行为**由**一组策略定义。窗口策略**决定**何时创建**新的存储桶，将哪些**事件分配**给哪些存储桶，以及**何时计算**桶中的数据。窗口**策略**的**指定**可以基于**时间**、**数量**或**其他数据属性**

下面介绍常见的窗口类型的语义

2.2.2.4.1 滚动窗口

滚动窗口将**事件分配**到**长度固定**的**不重叠**的桶中。当**窗口边界通过**时，所有**事件**都被**发送到**一个**计算函数**进行处理。基于计数的滚动窗口定义了在触发评估之前收集了多少事件。图2-6显示了一个基于计数的滚动窗口，它将输入流分到**四个元素**组成的桶。基于时间的滚动窗口定义了桶中事件的时间间隔。图2-7显示了一个基于时间的滚动窗口，它将事件收集到桶中，并**每10分钟**触发一次计算。

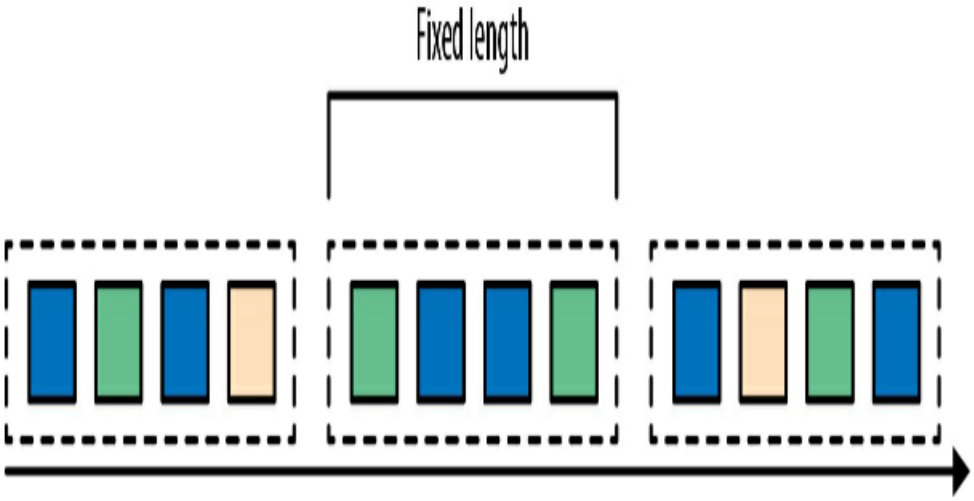


Figure 2-6. Count-based tumbling window

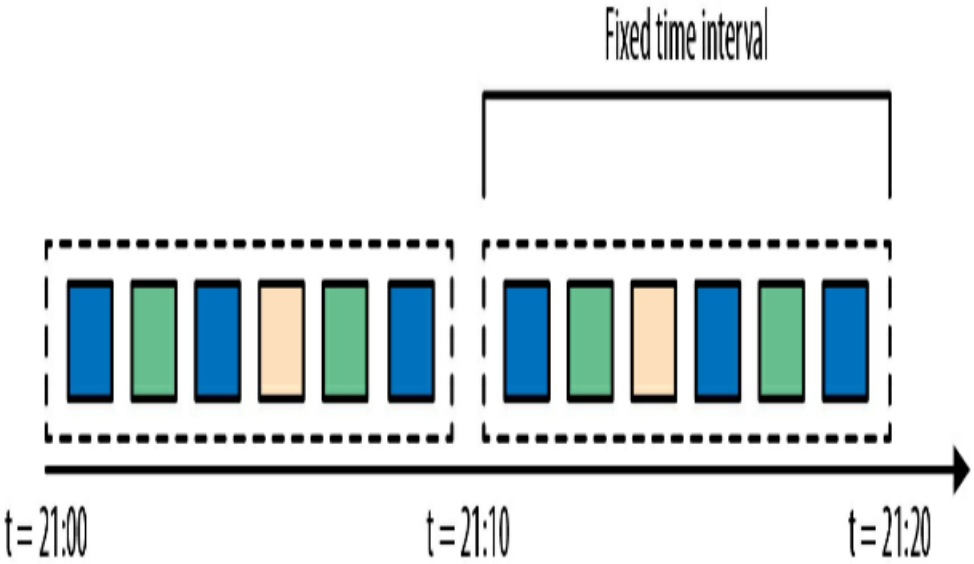
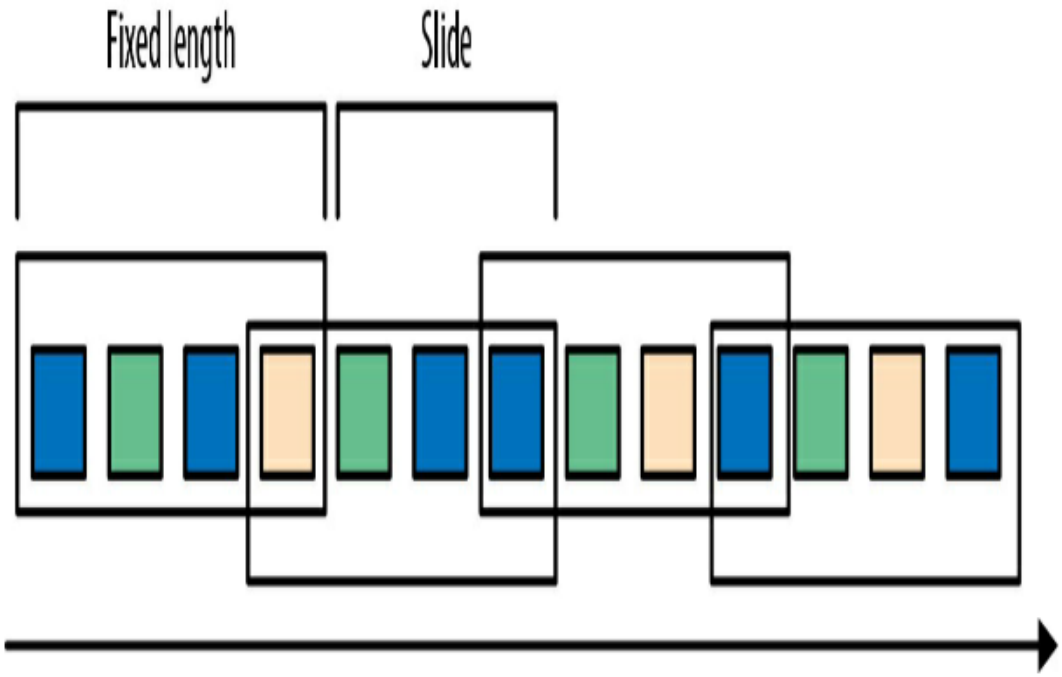


Figure 2-7. Time-based tumbling window

2.2.2.4.2 滑动窗口

滑动窗口将事件分配到**固定大小**的**允许互相重叠**的桶中。因此，**一个事件 可能**属于**多个桶**。我们通过指定**桶的长度**和**滑动间隔**来定义滑动窗口。图2-8中的窗口长度为4，滑动间隔为3。



2.2.2.4.3 会话窗口

会话窗口在常见的现实场景中非常有用，在这些场景中，滚动窗口和滑动窗口都不能应用。考虑一个分析在线用户行为的应用程序。在这样的应用程序中，我们希望将来自**同一会话**的**事件**分到**一组。**

会话窗口根据**会话间隔**（session gap）对**事件**进行**分组**，会话间隔定义了认为会话已关闭的非活动时间。（也就是如果用户在很长的一段时间内没有与服务器通信就认为他的会话已经关闭了）

窗口操作与流处理中的两个主要概念密切相关:**时间语义**和**状态管理。**

- 流数据通常会有延迟或者乱序到达，这时如何保证窗口正确划分就很重要
- 此外，为了避免故障，需要在生成结果之前将窗口中的数据都采取安全措施保护起来

2.3 时间语义

2.3.2 处理时间

处理时间是**机器**上**本地时钟**的时间。处理时间窗口**包括在**一段时间内** **碰巧到达窗口**的**所有事件**，由机器的本地时钟测量。如图2-12所示

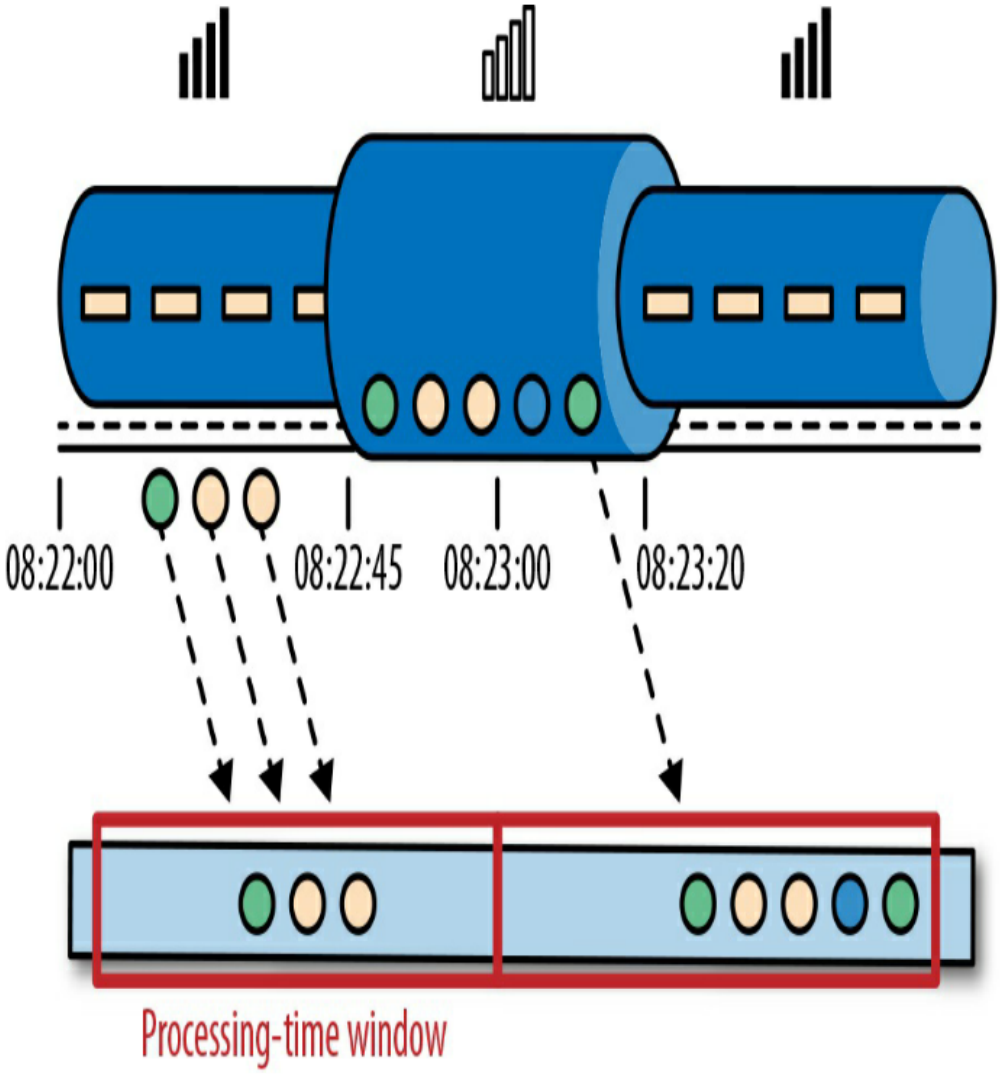


Figure 2-12. A processing-time window continues counting time even after Alice's phone gets disconnected

2.3.3 事件时间

事件时间是流中的**事件实际发生**的**时间**。事件时间通过**附加到流事件**的**时间戳**来判断。

图2-13显示：**即使**事件有**延迟**，事件时间窗口也能**准确**地把事件**分配**到正确的窗口中，从而**反映事情发生的真实情况。**

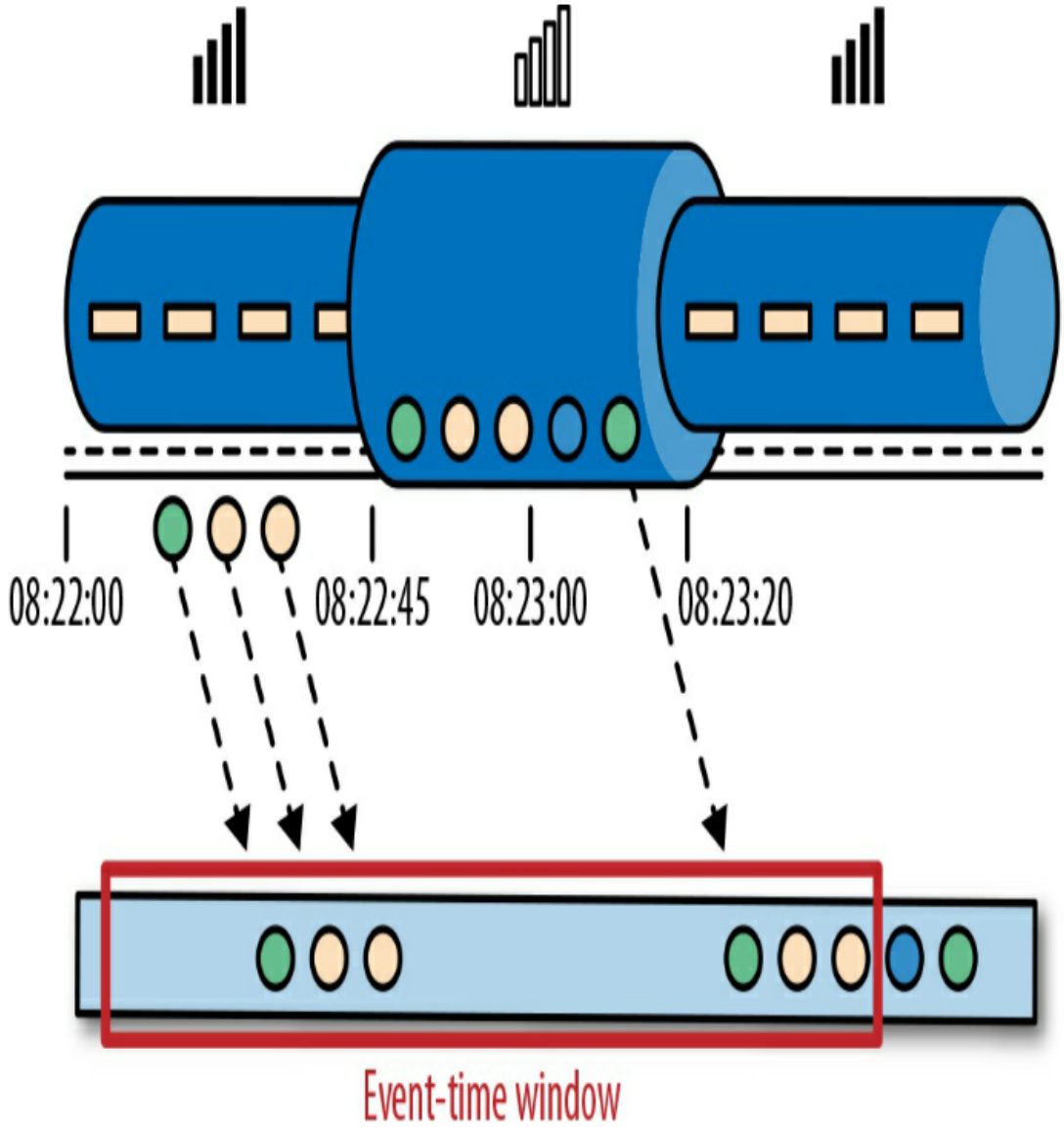


Figure 2-13. Event time correctly places events in a window, reflecting the reality of how things happened

无论数据流的处理速度有多快，事件到达算子的顺序是怎样的，事件时间窗口的计算将产生相同的结果。

通过依赖事件时间，即使是在**无序数据**的情况下，我们**也可以保证结果的正确性。**此外，当与可重放的流结合时，时间戳的确定性使你能够回到过去。也就是说，你可以重放一个流并分析历史数据，就像事件是实时发生的一样。

2.3.4 水位线

到目前为止，在我们关于事件时间窗口的讨论中，我们忽略了一个非常重要的方面:我们如何决定**事件时间窗口**的**触发时机**（什么时候停止收集并开始计算）？也就是说，我们要等多久才能确定我们已经收到了某个时间点之前发生的所有事件？考虑到分布式系统的不可预测性和由外部带来的各种延迟，**这些问题没有绝对正确的答案。**

水位线(watermark)是一种**全局进度度量**，它是一个**时间点**。它表明我们确信**这个时间点之前的事件全部到达了**。本质上，水位线提供了一个**逻辑时钟**，通知系统**当前的事件时间**。当操作员收到时间为T的水位线时，可以假设不会再收到时间戳小于T的事件。水位线对于事件时间窗口和处理无序事件的算子都是必不可少的。

水位线提供了结果可信度和延迟之间**trade-off**。

- **激进**的水位线确保**低延迟**，但提供**较低**的**可信度**。
- **保守**的水位线带来**高延迟**，但同时带来**较高**的**可信度**。

流处理系统会**提供某种机制**来**处理**在**水位线之后到达**的事件。

2.3.5 处理时间与事件时间

此刻你可能会想，既然事件时间解决了我们所有的问题，为什么我们还要去关心处理时间?

事实是，在某些情况下，处理时间确实很有用。

- 处理时间窗口**引入了尽可能低的延迟**。
- 当你需要定期**实时报告**结果时，但是**不太关注**结果的**精度**时，处理时间是更合适的。
- 最后，处理时间窗口提供了**流本身的真实情况**，这对于一些用例来说可能是一个理想的属性。

2.4 状态和一致性模型

状态在数据处理中无处不在。任何复杂一点的计算都需要它。为了产生结果，函数在一段时间或多个事件上**累积状态**(例如，计算聚集或检测模式)。**有状态算子**使用**传入事件**和**内部状态**来**计算它们的输出**并**更新状态**。

在连续运行的流作业中，状态在事件之间是持久的，我们可以在编程模型中将其作为一级公民公开。而在之前的批处理中，后一个批次的数据是看不到前一个批次的数据的。

由于流操作引擎有可能处理的是**无限流**，因此应小心**不要让内部状态无限增长**。为了限制状态的大小，**算子**通常会对到目前为止看到的事件进行某种**总结**或**概要**。这样的摘要可以是**计数**、**总和**、迄今为止所看到的事件的**抽样**、**窗口**缓冲区。

支持有状态算子会带来很多实现上的挑战:

1. **状态管理**：系统需要有效地**管理状态**，并**确保它不受并发更新的影响**。
2. **状态划分**：并行化变得复杂，因为结果取决于状态和传入的事件。幸运的是，在许多情况下，您可以通过一个键来划分状态，并独立管理每个分区的状态。例如，比如正在处理来自一组传感器的测量流，可以用不同的分区来处理不同的传感器。

3. **状态恢复**：有状态操作符带来的第三个也是最大的挑战是**确保状态可以恢复**，并且即使在出现故障的情况下结果也是正确的。

2.4.1 任务故障

流式作业中的**算子状态**非常**重要**，应防止出现故障。如果**状态在故障期间丢失**，恢复后的结果将是不正确的。流处理引擎不仅需要保证在出现任务故障时可以正常运行，还需要保证结果和算子状态的正确性。

对于输入流中的每个事件，任务执行以下步骤:

1. **接收**事件，将其**存储在本地缓冲区**中；
2. **更新内部状态**；
3. 产生**输出**记录。

在这些步骤中的任何一个都可能发生故障，系统必须清楚地定义其在每种**故障场景**中的**如何处理。**例如，一个定义完整的流式处理系统需要明确以下问题：如果任务在第一步失败，事件会丢失吗？如果在更新了内部状态后失败了，恢复后还会再更新吗？而在上面这些情况下，输出还是正确的吗？

2.4.2 结果保障

在批处理场景中，所有这些问题都得到了回答，因为批处理作业可以简单地从头开始重新启动。因此，没有事件丢失，状态完全是从零开始建立的。然而，在流处理中，这些问题很棘手。流式系统通过提供**结果保障**(result guarantee)来**定义它们在出现故障时的行为。**接下来，我们回顾了现代流处理引擎提供的几种不同级别的结果保障。

2.4.2.1 至多一次(AT-MOST-ONCE)

当任务失败时，最简单的方法就是**不做任何事情**来恢复丢失的状态和重放丢失的事件。**至多一次只保证每个事件至多处理一次。**换句话说，系统可以简单地丢弃事件，不做任何事情来确保结果的正确性。这种类型的保障也被称为"**无保障**"，因为即使是系统丢弃所有事件也可以提供这种保证。

2.4.2.2 至少一次(AT-LEAST-ONCE)

在大多数现实世界的应用程序中，人们期望事件不会丢失。这种类型的保证被称为**至少一次**，**这意味着所有事件都将被处理，并且其中一些事件有可能被处理多次。**如果应用程序的正确性仅取决于信息的完整性，重复处理可能是可以接受的。

为了确保**至少一次**这种结果保障，需要有一种方法来**重放(replay)事件**——要么从源(source)，要么从某个缓冲区(buffer)。

下面介绍两种保证至少一次的方式

1. **持久事件日志**将所有事件写入持久存储，以便在任务失败时可以重放(replay)。
2. 另一种方法是**使用记录确认**。此方法将每个事件存储在缓冲区中，直到管道中的所有任务都确认这个事件已经处理过了，此时可以丢弃该事件。

2.4.2.3 精确一次(EXACTLY-ONCE)

精确一次是最严格的保证，也很难实现。它意味着不仅**不会有事件丢失**，而且**每个事件只允许处理一次**。从本质上来说，精确一次**意味着我们的应用程序将提供完全正确的结果，就好像从未发生过失败一样**。

精确一次是**以至少一次为前提**的，因此**数据重放机制**必不可少。

而且在故障恢复之后，处理引擎应该知道一个事件的更新是否已经反映在状态上。有两种实现方式：

- **事务性更新**是实现这一结果的一种方式，但是它们会导致大量的性能开销。
- 相反，Flink使用轻量级**快照机制**来实现一次结果保证

2.4.2.4 端到端精确一次(END-TO-END EXACTLY-ONCE)

端到端保证指的是整个数据处理流水线上的结果正确性。流水线上的每个组件都提供自己的保证，完整管道的**端到端保证**将由所有组件中**最弱的**那个组件来决定。有时候弱的保障可能会表现出强的语义，比如，你使用至少一次来求最大值或者最小值，管道的其他组件都使用精确一次，那么这个管道也是端到端精确一次的。

第3章 Apache Flink架构

在这一章中，我们对**Flink的架构**进行了一个**高层次的介绍**，并描述了Flink如何解决我们之前讨论过的流处理相关问题。特别地，我们重点解释Flink的**分布式架构**，展示它在**流处理应用**中是如何处理**时间**和**状态**的，并讨论了它的**容错机制**。

3.1 系统架构

Flink是一个用于**状态化并行数据流处理**的**分布式系统**。Flink设置由多个进程组成，这些进程通常分布在多台机器上运行。

分布式系统需要解决的常见挑战是

1. 集群中**计算资源**的**分配和管理**

2. **进程协调**
3. 持久和高可用性**数据存储**
4. **故障恢复**

Flink本身并没有实现所有这些功能。**它只关注于其核心功能——分布式数据流处理**，但是利用了很多现有的开源中间件和框架来实现其他非核心部分。

- Flink与**集群资源管理器**(如Apache Mesos、YARN和Kubernetes)集成得很好，但也可以配置为作为独立集群运行。
- Flink不提供持久的**分布式存储**。相反，它利用了像HDFS这样的分布式文件系统或S3这样的对象存储。
- 对于高可用设置中的**领导选举**，Flink依赖于Apache ZooKeeper。

3.1.1 搭建Flink所需的组件

Flink的搭建由四个不同的组件组成，它们一起工作来执行流应用程序。这些组件是**JobManager**、**ResourceManager**、**TaskManager**和**Dispatcher**。由于Flink是用Java和Scala实现的，所以**所有组件都运行在Java虚拟机(jvm)上**。各组成部分的职责将在下面四个子小节分别介绍。

3.1.1.1 JobManager

> 应用管理

JobManager是**控制 单个应用程序执行**的**主进程**，每个应用程序由一个的JobManager控制。（**一对一**关系）

- JobManager负责**接收要执行的应用程序**。该应用程序由一个所谓的JobGraph（一个逻辑数据流图）和一个JAR文件组成（JAR文件捆绑了该应用程序所有必需的类、库和其他资源）。
- JobManager将JobGraph**转换**为名为ExecutionGraph的**物理数据流图**，该数据流图**由可并行执行的任务组成**。
- JobManager从ResourceManager**请求必要的资源**(TaskManager槽)来执行任务。一旦它**接收到足够数量的TaskManager槽**，它就会**将**ExecutionGraph的**任务分配给**执行它们的**TaskManager**。
- 在**执行期间**，JobManager**负责所有需要集中协调的操作**，如**检查点**的协调。

3.1.1.2 RecourceManager

> 资源管理

Flink**为 不同**的环境和**资源提供者**(如YARN、Mesos、Kubernetes和独立部署)**提供了多个资源管理器**。

- ResourceManager负责管理Flink的**处理资源单元**---**TaskManager槽**。
- 当JobManager请求TaskManager槽时，ResourceManager会**命令某个带有空闲槽的TaskManager将它的空闲槽提供给JobManager**。
- 如果ResourceManager**没有足够的槽**来满足JobManager的请求，则ResourceManager可以**与资源提供者对话**，让资源提供者尝试启动更多的TaskManager。
- ResourceManager还负责**终止空闲的TaskManager**以释放计算资源。

3.1.1.3 TaskManager

> 工作进程，执行任务的

TaskManager是Flink的**工作进程**(worker process，工人)

- 通常，在一个Flink集群中有多个TaskManager在运行。
- 每个TaskManager提供**一定数量的槽。槽的数量**限制了TaskManager**可以执行的任务数量。**
  - 在TaskManager**启动之后**，TaskManager**将它的槽注册到ResourceManager。**
  - 当JobManager请求槽的时候，根据ResourceManager的指示，TaskManager**向JobManager提供一个或多个槽。**
  - 然后JobManager可以**将任务分配到槽中**，让TaskManager执行这些任务。
  - 在执行期间，TaskManager与运行**相同应用**但是**不同任务**的其他TaskManager**交换数据。**

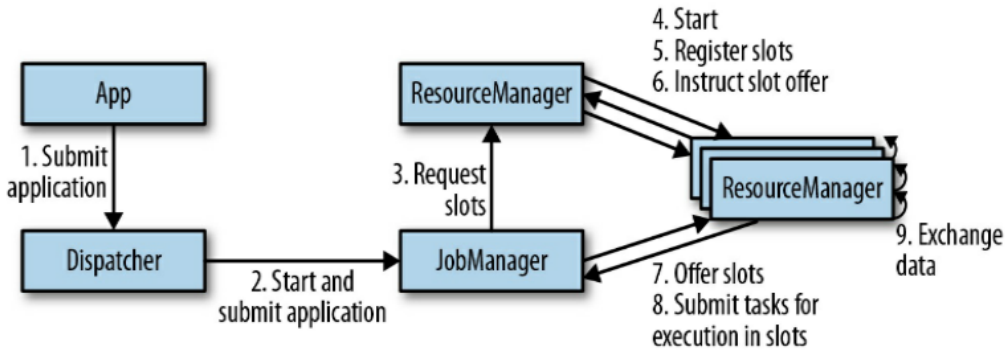
3.1.1.4 Dispatcher

> 与用户直接对话

Dispatcher提供一个REST接口让用户提交要执行的应用。

- 应用提交执行后，它将启动JobManager，并将应用交给它来执行。
- Dispatcher还运行一个web仪表盘来提供关于作业执行的信息。

3.1.1.5 整体架构图



3.1.2 应用部署

Flink应用程序可以以两种不同的模式来部署。

3.1.2.1 框架模式

在这种模式下，**Flink应用程序**被**打包到一个JAR文件中**，并由客户端提交给一个正在运行的服务。该服务可以是Flink Dispatcher、Flink JobManager或YARN的ResourceManager。

- 如果应用程序被**提交到JobManager**，它将**立即开始执行应用程序。**
- 如果应用程序被**提交给Dispatcher或YARN ResourceManager**，它将**启动JobManager**并**移交应用程序**，然后JobManager将开始执行应用程序。

3.1.2.2 库模式

在这种模式下，**Flink应用程序**被**绑定在**一个应用程序特定的容器镜像中，比如**Docker镜像。**

- 该镜像还包括运行JobManager和ResourceManager的代码。
- 当容器从镜像启动时，它会自动启动ResourceManager和JobManager，并执行绑定的应用程序。
- **第二个**独立于应用程序的**镜像**用于**部署TaskManager容器。**
    - 从这个镜像启动的容器会自动启动TaskManager，它连接到ResourceManager并注册它的槽。
    - 通常，外部资源管理器(如**Kubernetes**)负责**启动镜像**，并负责在**发生故障时重新启动**容器。

第一种模式比较传统，第二种模式常用于微服务中。

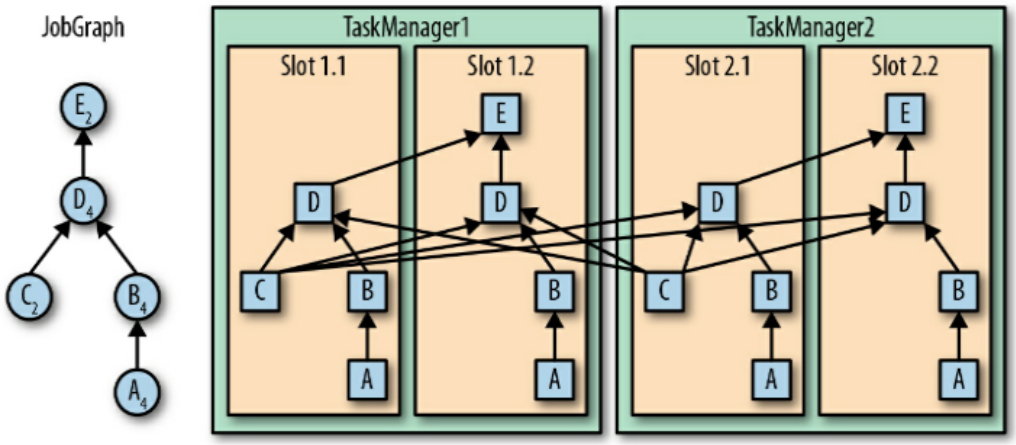
3.1.3 任务执行

TaskManager可以**同时执行多个任务。**

这些任务可以

- 属于同一算子(**数据并行**)、
- 不同算子(**任务并行**)的子任务
- 甚至是来自**不同应用程序**的子任务(**应用并行**)。

TaskManager提供**固定数量的处理槽**来**控制**它能够**并发**执行的任务的**数量。一个处理槽**可以执行**应用程序**的**某个算子**的**一个并行任务。**下图是一个TaskManager、处理槽、任务以及算子关系的例子。



左侧是一个**JobGraph**（应用程序的非并行表示，**逻辑图**）。

- 它由5个算子组成。
- 算子A和C是数据源，算子E是数据汇。

右侧是一个**ExecutionGraph（物理图）**

- 算子C和E的**并行度**为2。其他算子的并行度为4。
- 由于**最大算子并行度**是4个，应用程序至少需要**4个可用的处理槽**来执行。
- 给定两个各有两个处理槽的Taskmanager，就满足了这个需求。
- JobManager将JobGraph扩展为一个ExecutionGraph，并将任务分配给四个可用插槽。
- 并行度为4的算子各自有4个并行任务，这些任务 被分配给每个槽。
- 运算符C和E的各自有两个并行任务，分别被分配到槽1.1和2.1以及槽1.2和2.2。
- 将**多个不同算子**的**任务 分配到同一个插槽**的优点是**这些任务可以在同一个进程中高效地交换数据**，而不需要访问网络。

每个TaskManager是一个JVM，而每个Slot是JVM中的一个线程。TaskManager在同一个JVM进程中以多线程方式执行它的任务。**线程**比单独的进程**更轻量**，通信成本更低，但**不会严格地将任务彼此隔离。**因此，一个行为不正常的任务可以杀死整个TaskManager进程和运行在它上面的所有任务。

3.1.4 高可用性设置

流式应用程序通常设计为24x7运行。因此，即使内部进程失败，也不能停止运行。

而要想从失败中恢复

1. 系统首先需要重新启动失败的进程
2. 其次，重新启动应用程序并恢复其状态。

本小节主要学习如何重新启动失败的进程。

3.1.4.1 TaskManager故障

下面举例说明TaskManager故障应该如何处理

- 假设我们的应用程序要以最大并行度为8来执行，那么四个TaskManager(每个TaskManager提供两个插槽)可以满足我们对并行度的需求。
- 如果其中一个TaskManager发生故障，可用插槽的数量将减少到6个。
- 在这种情况下，JobManager将**请求ResourceManager提供更多处理槽。**
- 如果**请求失败**，JobManager会**按照一定的时间间隔连续地重启应用。**直到重启成功（有足够多的空闲插槽就能重启成功）。

3.1.4.2 JobManager故障

比TaskManager失败更具挑战性的问题是JobManager失败。

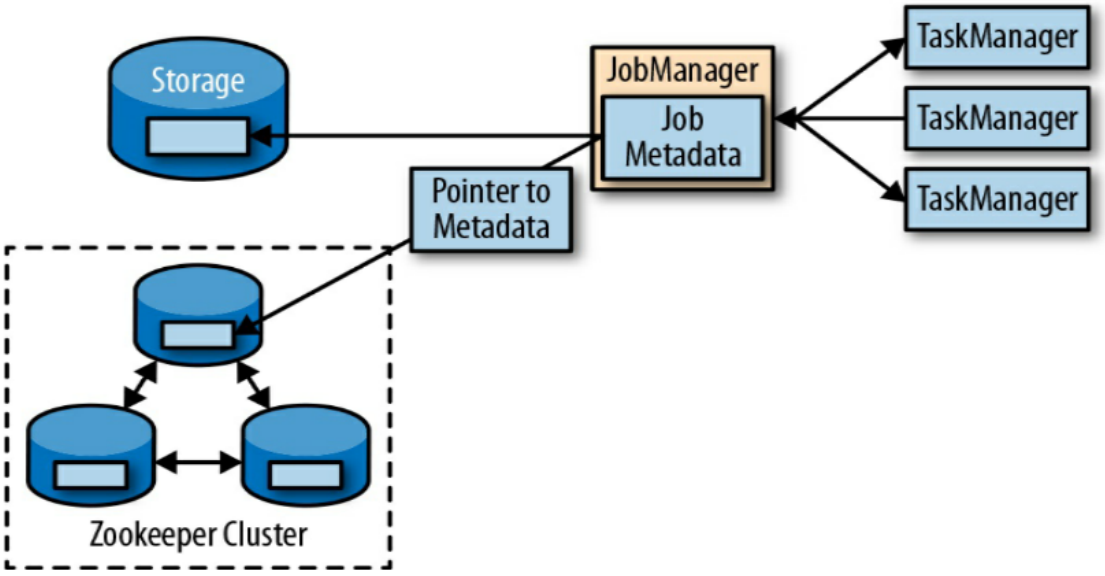
- JobManager**控制流应用程序的执行**，并保存有关其执行的元数据，例如指向已完成检查点的指针。

- 如果负责的JobManager进程失败，流应用程序将无法继续处理。
- 这使得JobManager成为Flink中的应用程序的一个**单点失效组件**（也就是如果这个组件失效，那么整个系统失效）。

为了克服这个问题，Flink支持一种**高可用模式**，该模式可以在原始JobManager失效时将**应用的管理权和应用的元数据 迁移**到另一个JobManager。

Flink的**高可用模式 基于 ZooKeeper**

- 它是一个分布式系统，来**提供分布式协调**和**共识服务**。
- Flink使用ZooKeeper进行**领袖选举**，并将其作为一个高可用性和持久的**数据存储**。
- 在**高可用性模式下**操作时，**JobManager**将**JobGraph**和所有必需的**元数据**(如应用程序的JAR文件)**写入远程持久存储系统**。
- 此外，JobManager将一个**指向存储位置的指针 写入ZooKeeper**的数据存储中。
- 在应用程序执行期间，JobManager接收各个任务检查点的状态句柄(存储位置)。当**检查点完成后，JobManager将状态写入远程存储，并将指向此远程存储位置的指针写入ZooKeeper**。
- 因此，**从JobManager故障中恢复所需的所有数据都存储在远程存储中，而ZooKeeper持有指向存储位置的指针**。
- 图3-3说明了这种设计。



当JobManager失败时，接管它工作的新JobManager执行以下步骤:

1. 它**从ZooKeeper请求存储位置，然后从远程存储中获取**JobGraph、JAR文件和应用程序最后一个检查点的存储位置。
2. 它向ResourceManager**请求处理槽**以**继续执行应用程序。**
3. 它将**重新启动应用程序**，并将其所有任务的**状态重置**为**检查点中的状态值。**

最后还有一个问题，当TaskManager或者JobManager失效时，谁会**触发它们的重启**?

- 在容器环境(如Kubernetes)中作为库部署运行应用程序时，失败的JobManager或TaskManager容器通常由容器编排服务自动重新启动。

- 在YARN或Mesos上运行时，Flink的其余进程将触发JobManager或TaskManager进程的重新启动。

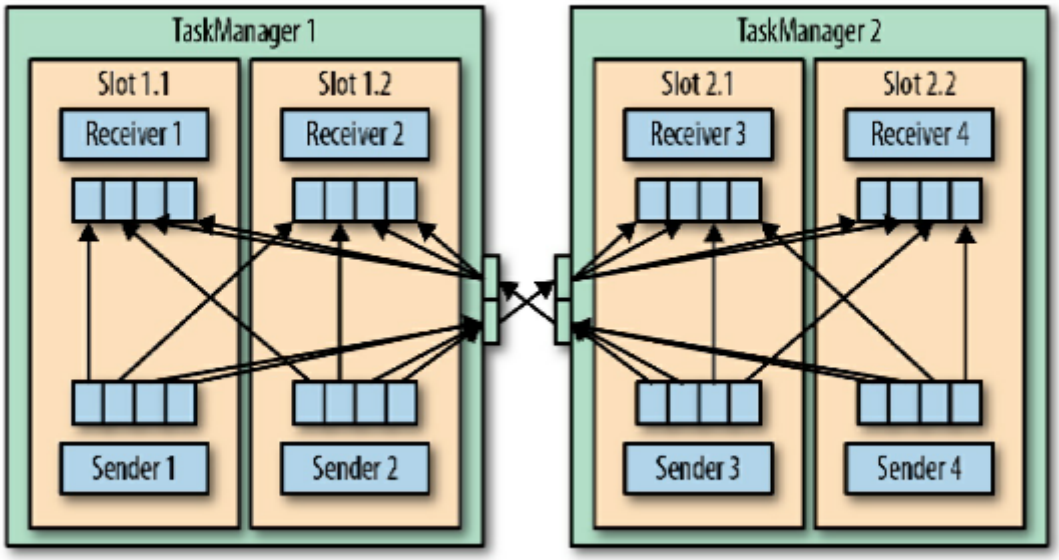
3.2 Flink中的数据传输

在运行过程中，**应用的任务**不断地**交换数据。TaskManager 负责**将数据从发送任务发送到接收任务。TaskManager的**网络组件**在发送记录之前在**缓冲区中收集记录**，就是说，记录不是一个一个发送的，而是先缓存到缓冲区中然后一批一批发送。这种技术是有效使用网络资源和实现高吞吐量的基础。

每个**TaskManager**都有**一个 网络缓冲池**(默认大小为32 KB)用于发送和接收数据。

- 如果**发送方任务**和**接收方任务**在**不同的TaskManager进程**中运行，则它们通过**网络通信。**
- **每对TaskManager**维护一个**永久的TCP连接**来交换数据。
- 使用shuffle连接模式时，每个发送方任务都需要能够向每个接收方任务发送数据。TaskManager需要为**每个接收任务**提供**一个专用的网络缓冲区**，此任务对应的发送方会向该缓冲区发送数据。

图3-4显示了这个架构。



- 在shuffle连接模式下，由于**接收端的并行度为4**，所以**每个发送端**都需要**4个网络缓冲区**来向接收端任务发送数据
- 由于**发送端的并行度也是4**，所以**每个接收端**也都需要**4个网络缓冲区**来接受发送端发送的数据
- **同一个TaskManager中的缓存区**会**共用同一条网络连接**
- 在shuffle模式或者broadcast模式下，需要的**缓冲区的大小**将是**并行度的平方级**
- Flink的网络缓冲区的默认配置对于中小型的设置是足够的。

当**发送方任务**和**接收方任务**在**同一个TaskManager进程**中运行时

1. 发送方任务将传出的记录序列化到缓冲区中，并在缓冲区**填满**后将其**放入队列**中。
2. 接收任务从队列中获取缓冲区，并对传入的记录进行反序列化。

3. 因此，在**同一TaskManager上运行的任务之间的数据传输不会导致网络通信。**

3.2.1 基于信用值的流量控制

通过网络连接发送单条记录很低效，并且造成很大的开销。**缓冲**是**充分利用网络连接的带宽**的关键。在流处理上下文中，**缓冲**的一个**缺点**是**增加了延迟**，因为**记录**是在缓冲区中收集的，而**不是立即发送的。**

Flink实现了一个基于信用值的流控制机制，其工作原理如下。

1. **接收任务**向**发送任务 授予一定的信用值**，也就是告诉发送端为了接收其数据，我为你保留的缓冲区的大小
2. 一旦发送方收到信用值通知，就会**在信用值允许范围内尽可能多的传输缓冲数据**，并会**附带**上**积压量大小**（已经填满准备传输的网络缓冲数目）
3. 接收方使用预留的缓冲来处理发送的数据，同时**依据**各发送端的**积压量信息**来**计算**所有发送方在**下一轮的信用值**分别是多少。

基于信用值的好处

- 基于信用的流控制**减少了延迟**，因为一旦接收方有足够的资源接受数据，发送方就可以发送数据。
- 此外，在数据分布不均的情况下，它是一种**有效的分配网络资源**的机制，因为信用是根据发送方的积压的大小授予的。
- 因此，基于信用的流控制是Flink实现**高吞吐**和**低延迟**的**重要一环。**

3.2.2 任务链接

Flink提供了一种被称为**任务链接**的优化技术，它可以**减少特定条件下本地通信的开销。**

- 为了满足任务链接的要求，**被链接的所有算子**必须配置**相同的并行性**，并通过**本地转发通道**进行**连接。**
- 图3-5所示的操作管道满足这些要求。它由**三个算子组成**，它们都**被配置为任务并行度为2**，并与本地转发连接连接。

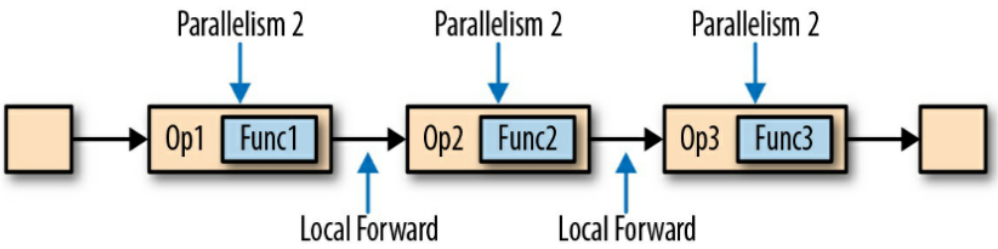
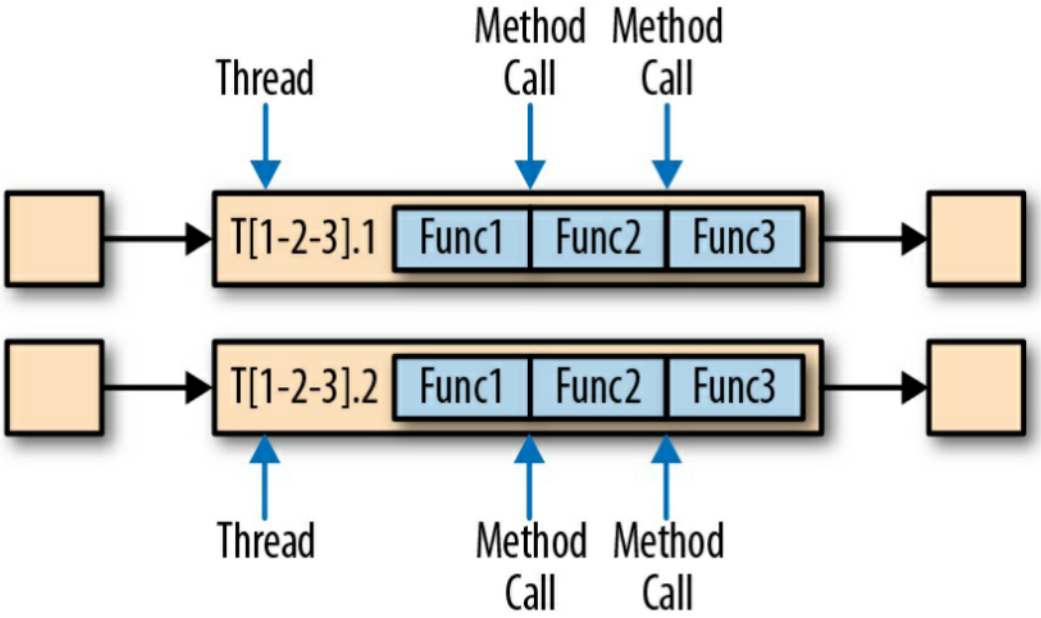


图3-6描述了如何在任务链接模式下执行管道。

- 多个算子函数被融合到单个任务中，由单个线程执行。
- 通过一个简单的方法调用，一个函数产生的记录被单独地移交给下一个函数。
- 因此，在**函数之间传递记录**基本上**没有序列化开销**和**没有通信开销。**



Flink在默认情况下会开启任务链接，但是也可以通过配置关闭这个功能

3.3 事件时间处理

正如上一节所述，**事件时间语义会生成可重复且一致性的结果**，这是许多流应用的刚性需求。下面，我们将描述**Flink如何在内部实现和处理事件时间戳和水位线**，以支持具有事件时间语义的流应用。

3.3.1 时间戳

Flink事件时间流应用处理的**所有记录都必须带时间戳。**时间戳将记录与特定的时间点关联起来，**通常是记录所表示的事件发生的时间点。**此外，在现实环境中，**时间戳乱序**几乎不可避免。

当Flink以事件时间模式处理数据流时，它会根据记录的事件时间戳来触发基于时间的算子操作。

- 例如，**时间窗口操作符**根据相关的时间戳将记录分配给窗口。
- Flink将**时间戳**编码为8字节长的**Long值**，并将它们**作为元数据附加到记录中。**
- 然后内置算子或者用户自定义的算子解析这个Long值就可以获得事件时间。

3.3.2 水位线

水位线用于标注事件时间应用程序中每个任务当前的事件时间。

- 基于时间的操作符使用这段时间来触发相关的计算计算并推动这个流进行。
- 例如，基于时间窗口的任务会在水位线超过窗口边界的时候触发计算并且发出结果

在Flink中，**水位线**被实现为一种带时间戳的**特殊记录。**如图3-8所示，水位线**像常规记录一样**在数据流中移动。

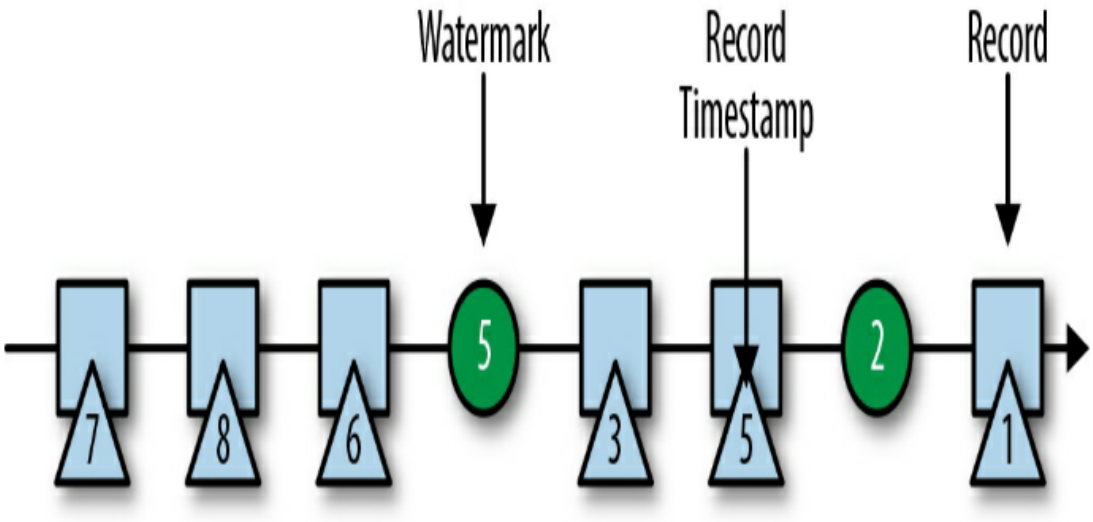


Figure 3-8. A stream with timestamped records and watermarks

水位线有两个**基本特性**:

1. 水位线必须是**单调递增**的，以确保任务的**事件时间时钟**是**前进**的，而不是向后的。
2. 水位线与记录的时间戳存在关系。一个时间戳为T的水位线表示：所有后续记录的时间戳都应该大于T。

第二个属性用于处理数据流中**时间戳乱序的记录**，例如图3-8中具有时间戳2和5的记录。

- 基于时间的算子任务可能会处理带有无序时间戳的记录，每个任务都会维护一个自己的事件时钟，并通过时间戳来更新这个时钟。
- 任务有可能接收到违反水位线属性且**时间戳 小于先前接收的水位线**的**记录**，该记录所属的计算可能已经完成。这样的记录称为**迟到记录。**

水位线的一个意义是，它们**允许应用控制结果**的**完整性**和**延迟。**

3.3.3 水位线传播和事件时间

在本节中，我们将讨论算子如何处理水位线。

- Flink将水位线实现为算子任务 **接收**和**发出**的**特殊记录。**
- 任务内部的时间服务会维护一些**计时器**(Timer)，任务可以在计时器服务上**注册计时器，以便将来在特定的时间点执行计算**，这些计时器依靠收到的水位线来激活。
- 例如，窗口操作符为每个活动窗口注册一个计时器，当事件时间超过窗口的结束时间时，计时器将清除窗口的状态。

当一个**任务收到水位线**时，会发生以下操作:

1. 任务根据**水位线的时间戳 更新**其**内部事件时间时钟**。
2. 任务的时间服务**根据更新后的时钟来执行那些超时计时器的回调**。对于每个过期的计时器，任务将调用一个回调函数，该函数可以执行计算并发出记录。
3. 任务根据更新后的时钟向下游任务发送水位线。

考虑到任务并行，我们将详细介绍一个任务如何将水位线发送到多个下游任务，以及它从多个上游任务获取水位线之后如何推动事件时间时钟前进。具体的方式如下

1. **任务**为**每个输入分区 维护 分区水位线**。
2. 当它**从一个分区接收到水位线**时，它**将**相应的**分区水位线 更新**为接收值和当前值的最大值。
3. **随后**，任务将其**内部事件时间时钟 更新**为所有分区水印的最小值。
4. 如果事件时间时钟前进，任务**处理所有触发的计时器**，最后通过**向**所有连接的**输出分区 发出更新后的水位线**，向所有下游任务**广播**它的新事件时间。

下图举了一个有4个输入分区和3个输出分区的任务在接受到水位线之后是如何更新它的分区水位线和事件时间时钟的。

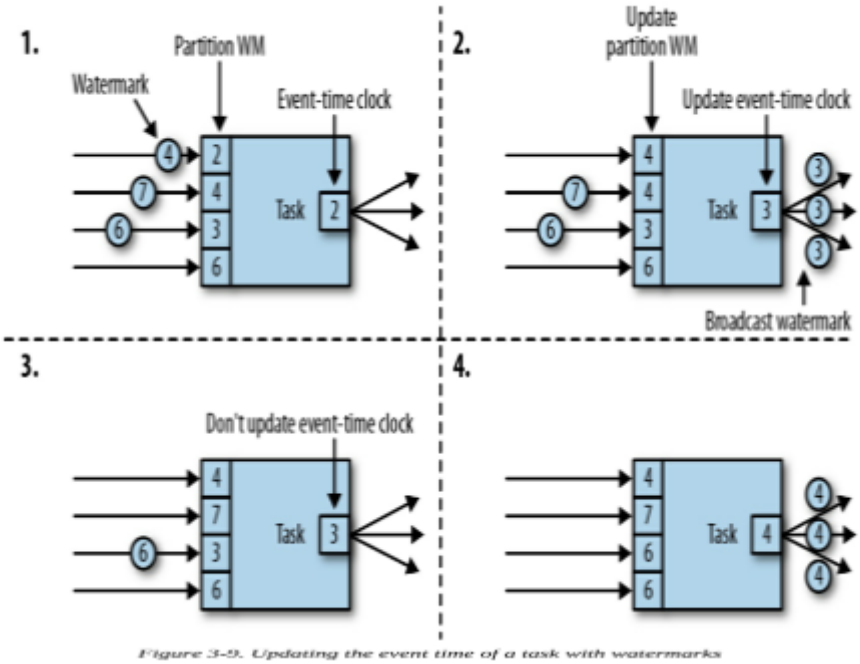


Figure 3-9. Updating the event time of a task with watermarks

Flink的水位线传播算法确保算子任务所发出**带时间戳的记录**和**水位线**一定会**对齐**。

- 然而，它依赖于这样一个事实，即所有的分区都不断地提供自增的水位线。
- 一旦一个分区不推进它的水位线，或者变成完全空闲而不再发送任何记录和水位线，任务的事件时间时钟将不会推进，进而导致计时器不会触发。
- 因此，**如果一个任务没有定期从所有输入任务接收到新的水位线**，那么**任务**的**处理延迟**和**状态大小**会**显著增加**。

对于具有两个输入流且**水位线差距很大**的算子，也会出现类似的效果。具有两个输入流的任务的事件时间时钟将**受制于较慢的流**，通常较快的流的记录或中间结果将处于缓冲状态，直到事件时间时钟允许处理它们。

3.3.4 时间戳分配和水位线生成

下面介绍时间戳和水位线是如何产生的。

时间戳和水位线通常是在流应用**接收数据流时 分配和生成**的。Flink DataStream应用可以通过三种方式完成该工作

1. **在数据源完成**：当一个流被读入到一个应用中时。数据源算子将产生带有时间戳的记录流。水位线可以作为特殊记录在任何时间点发出。如果数据源暂时不再发出水位线了，可以将自己声明为空闲，Flink会在后续算子计算水位线时将那些来自空闲数据源的流分区排除在外。
2. **周期性分配器**(Periodic Assigner)：这个Assigner可以从每个记录中提取一个时间戳，并定期查询当前的水位线。提取到的时间戳被分配给相应的记录，所查询的水印被加入到流中。
3. **定点分配器**(Punctuated Assigner)：它可以用于**根据特殊输入记录来生成水位线**

3.4 状态管理

大多数**流应用**是**有状态**的。许多算子不断读取和更新某种状态。不管是内置状态还是用户自定义状态，Flink的处理方式都是一样的。

在本节中，我们将讨论

1. Flink支持的**不同类型**的状态。
2. **状态后端**如何存储和维护状态
3. 有状态应用程序如何通过进行**状态再分配**来实现**扩缩容**。

通常，**需要任务去维护并用于计算结果的数据都属于任务的状态。**图3-10显示了任务与其状态之间的典型交互。

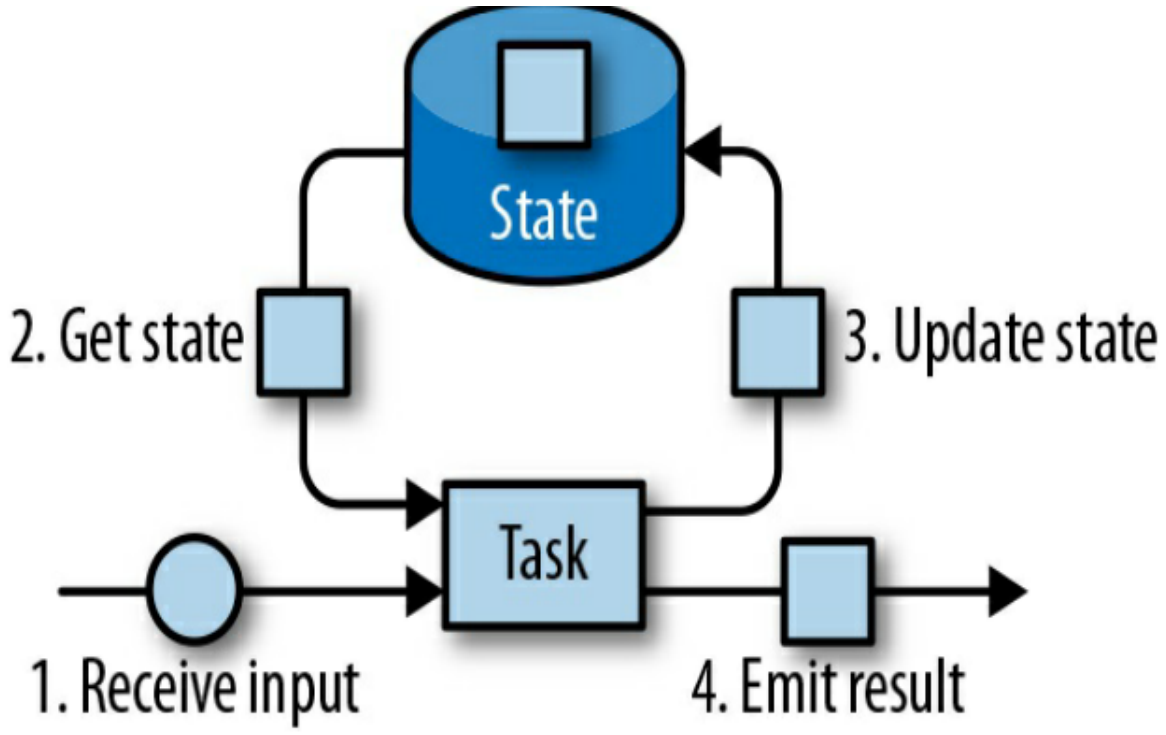


Figure 3-10. A stateful stream processing task

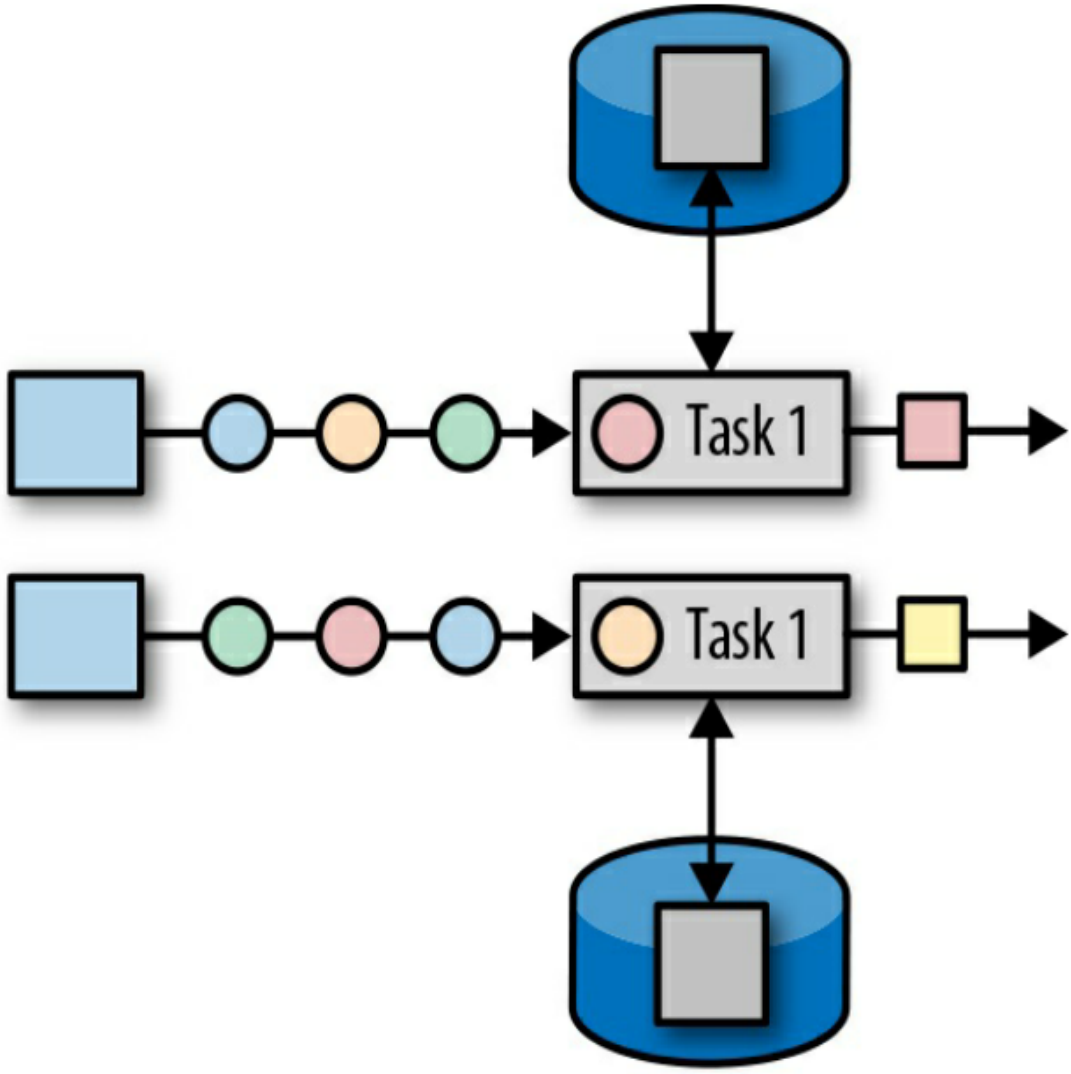
- 任务接收一些输入数据。
- 在处理数据时，任务可以读取和更新其状态，
- 并根据其输入数据和状态计算其结果。

然而，高效可靠的状态管理更具挑战性。这包括处理非常大的状态(可能超过内存)，并确保在发生故障时不会丢失任何状态。所有与状态一致性、故障处理、高效存储和访问相关的问题都由Flink处理，以便开发人员能够将重点放在应用程序的逻辑上。

在Flink中，状态总是与一个特定的算子相关联。为了让Flink的运行时知道算子有哪些状态，算子需要对其状态进行注册。根据**作用域的不同**，有**两种类型的状态**：**算子状态**和**键值分区状态**

3.4.1 算子状态

算子状态的**作用域**为**算子的单个任务**。这意味着由同一并行任务之内的记录都可以访问同一状态。算子状态不能被其他任务访问。如下图

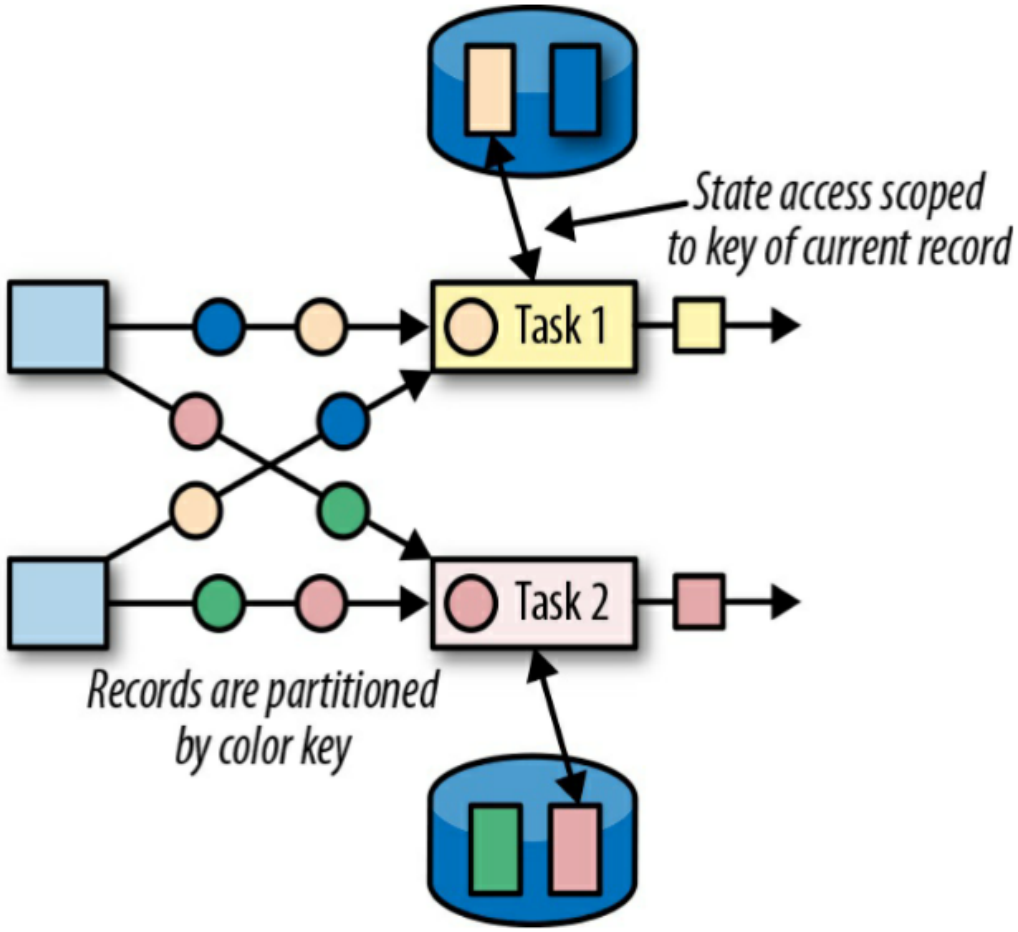


Flink为算子状态提供了三类原语

- 列表状态：将**状态**表示为**一个条目列表**
- 联合列表状态：同样将状态表示为一个条目列表。但是，在出现故障或从保存点启动应用程序时，它的恢复方式与常规列表状态不同。
- 广播状态：专门**为哪些需要保证算子的每个任务状态都相同的场景**而设计

3.4.2 键值分区状态

键值分区状态是根据算子**输入记录中定义的键**来维护和访问的。Flink**为每个键维护一个状态实例，该状态实例总是位于那个处理对应键值记录的任务上。**当任务处理一个记录时，它自动将状态访问范围限制到当前记录的键。因此，具有相同键值分区的所有记录都访问相同的状态。图3-12显示了任务如何与键值分区状态交互。



键值分区状态是一个在算子的所有并行任务上进行分区的分布式键值映射。键值分区状态原语如下

- **单值状态**：为每个键存储一个任意类型的值。该值可以是一个任意复杂的数据结构。
- **列表状态**：为每个键储存一个列表。列表条目可以是任意类型。
- **映射状态**：为每个键存储键值映射。映射的键和值可以是任意类型。

3.4.3 状态后端

为了确保快速的状态访问，每个并行任务都在本地维护其状态。至于**状态的具体**存储、访问和维护，则**由**一个称为**状态后端**的可拔插组件来**完成**。

状态后端负责两件事:

1. **本地状态管理**
2. 将状态以检查点的形式**写入远程存储**

对于本地状态管理，Flink提供两种实现

- 第一种状态后端，将**状态**作为存储在**JVM**堆内存数据结构中的对象进行管理。
- 第二种状态后端，**序列化**状态对象并将它们**放入RocksDB**中，这种方式是**基于硬盘**的。
- 虽然第一种实现提供非常快的访问速度，但它受到内存空间大小的限制。访问RocksDB会比较慢，但是空间大。

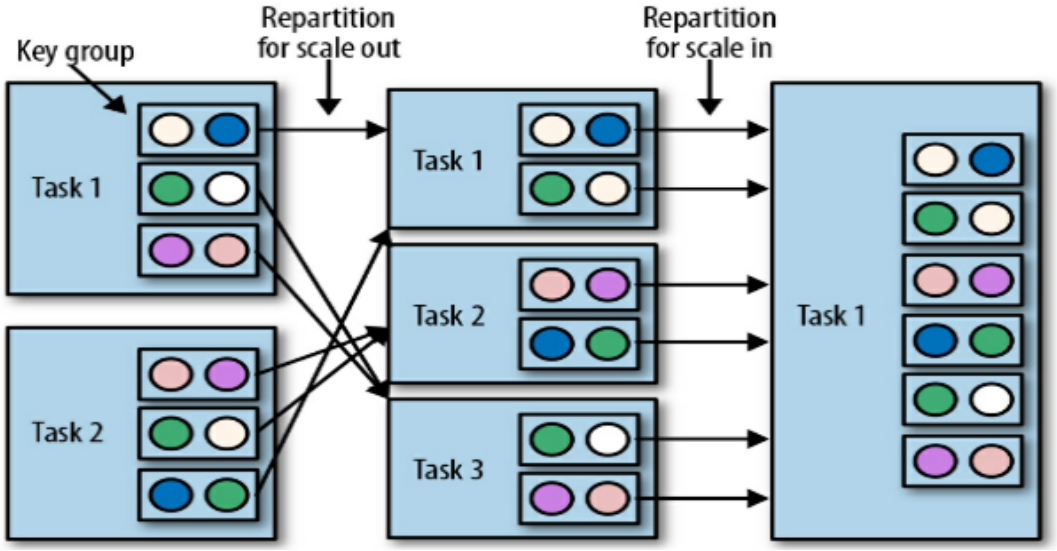
状态检查点很重要，因为Flink是一个分布式系统，状态只能在本地维护。TaskManager进程可能在任何时间点失败。因此，它的存储必须被认为是易失的。**状态后端负责将任务的状态检查点指向远程和持久存储**。用于检查点的远程存储可以是分布式文件系统或数据库系统。状态后端在状态检查点的方式上有所不同。例如，RocksDB状态后端支持增量检查点，这可以显著减少非常大的状态的检查点开销。

3.4.4 有状态的算子的扩缩容

流应用的一个基本需求是**根据输入速率**的增加或减少而**调整算子的并行性**。有状态算子，调整并行度比较难。因为我们需要把状态重新分组，分配到与之前数量不等的并行任务上。

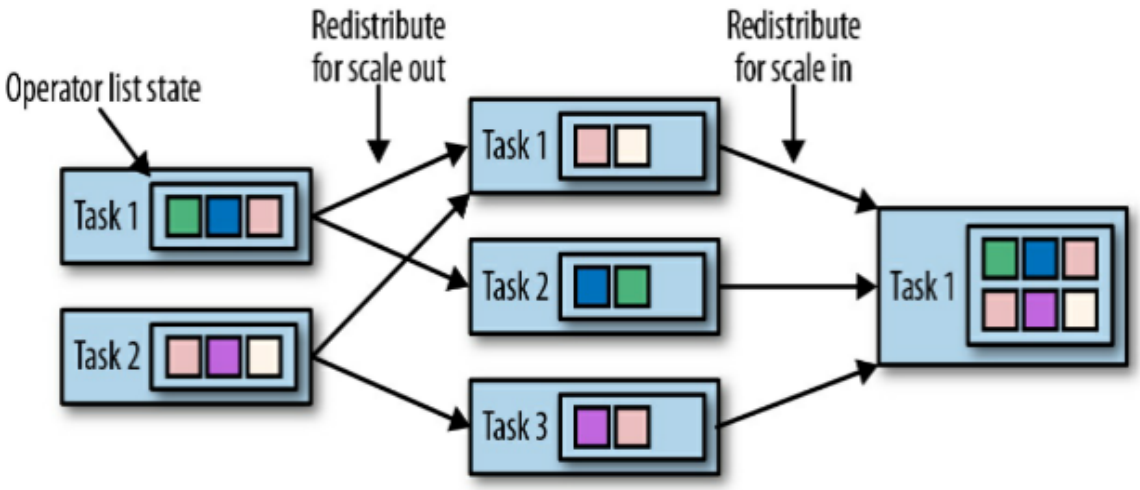
3.4.4.1 带有键值分区状态的算子扩缩容

带有键值分区状态的算子可以通过将键重新划分来进行任务的扩缩容。但是，为了提高效率，Flink不会以键为单位来进行划分。相反，Flink以**键组**作为单位**来重新分配**，每个**键组**里面包含了多个键。



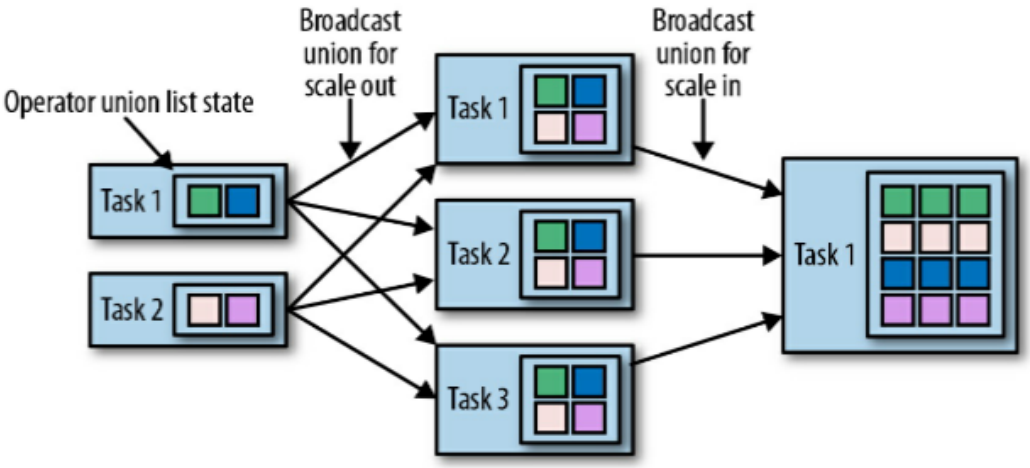
3.4.4.2 带有算子列表状态的算子扩缩容

带有算子列表状态的算子在扩缩容时会**对列表中的条目进行重新分配。**理论上来说，所有并行任务的列表项会被**统一收集**起来，并**再均匀重新分配。**如果列表项的数量少于算子的新并行度，一些任务将以空状态开始。图3-14显示了操作符列表状态的重新分配。



3.4.4.3 带有算子联合状态的算子扩缩容

带有算子联合状态的算子会在**扩缩容时**把**状态列表中的全部条目 广播到全部任务中。**然后，任务自己来选择使用哪些项和丢弃哪些项。如图3-15显示。



3.4.4.4 带有算子广播状态的算子扩缩容

带有算子广播状态的算子在扩缩容时会把状态拷贝到全部新任务上。这样做是因为广播状态要确保所有任务具有相同的状态。在缩容的情况下，直接简单地停掉多余的任务即可。如图3-16显示。

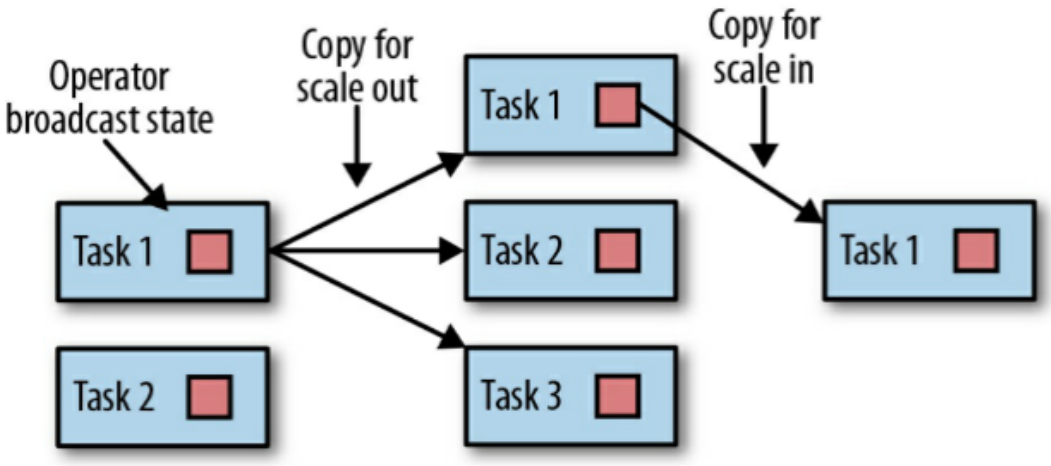


Figure 3-16. Scaling an operator with operator broadcast state out and in

3.5 检查点、保存点、状态恢复

Flink是一个分布式的数据处理系统，且任务在本地维护它们的状态，Flink必须确保这种状态不会丢失，并且在发生故障时保持一致。

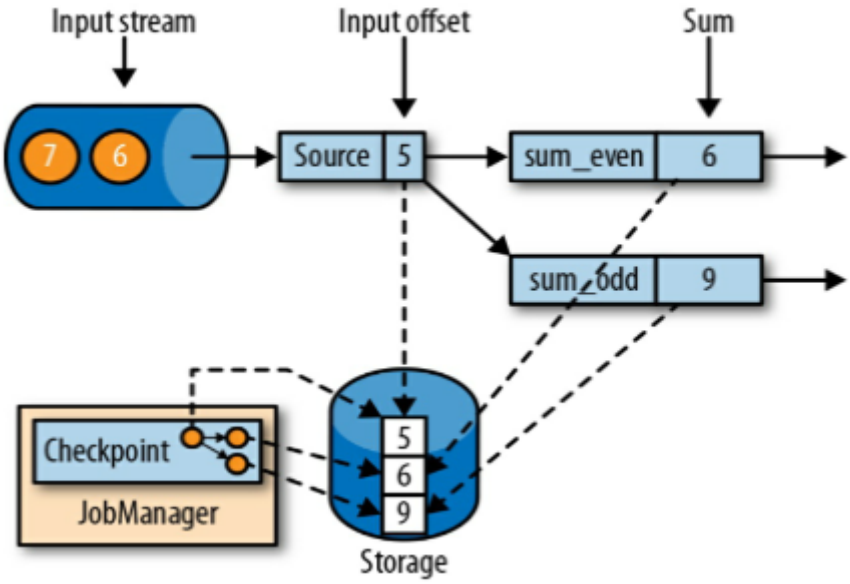
在本节中，我们将介绍Flink的**检查点**和**故障恢复机制**，看一下它们是如何提供精确一次的状态一致性保障。此外，我们还讨论了Flink独特的保存点(savepoint)功能，它就像一把瑞士军刀，解决了运行流式应用过程中的诸多难题。

3.5.1 一致性检查点

有状态流应用程序的一致检查点是在所有任务都处理完等量的原始输出后对全部任务状态进行的一个拷贝。我们可以通过一个朴素算法来对应用建立一致性检查点的过程进行解释。朴素算法的步骤为:

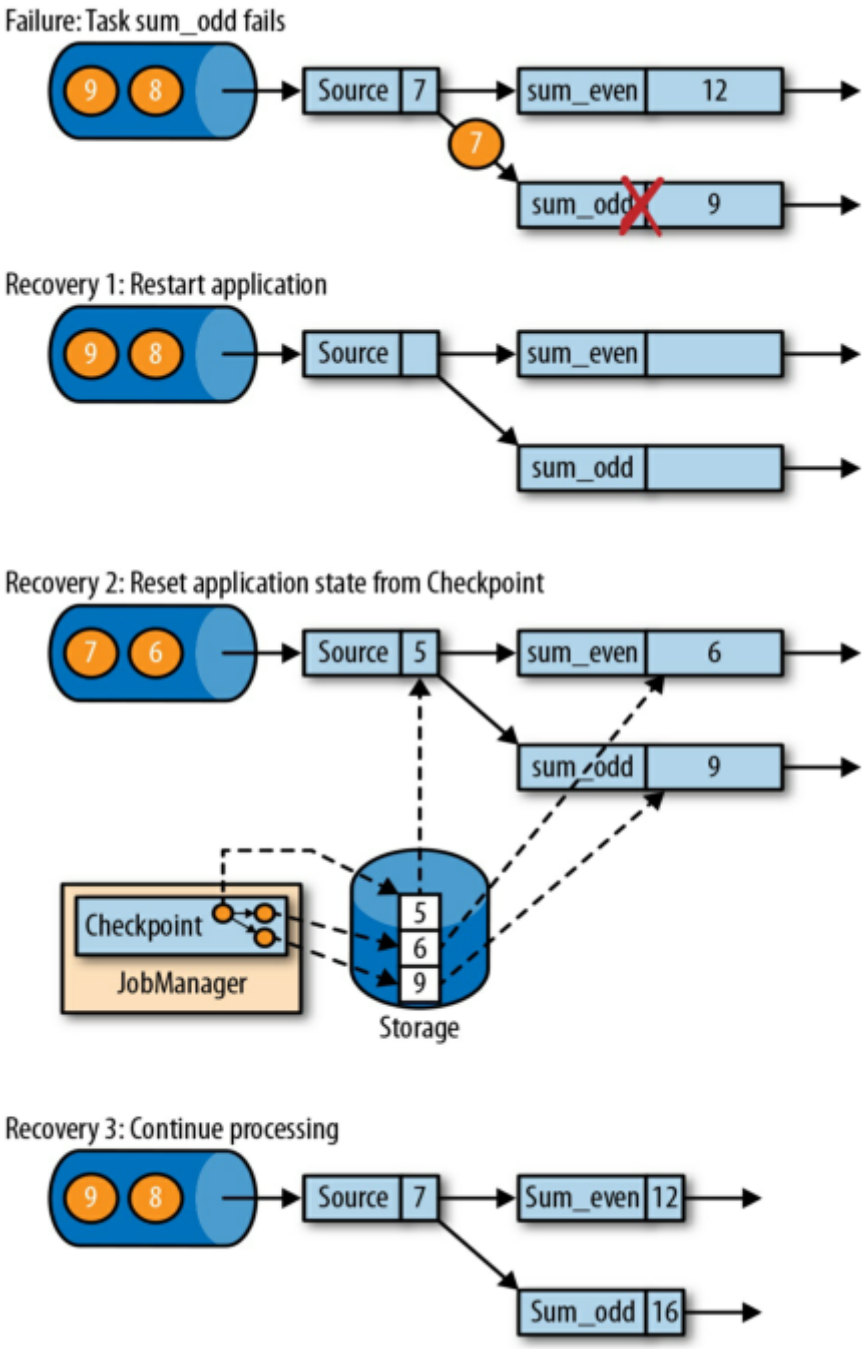
1. **暂停接收**所有输入流。
2. **等待**所有**流入系统的数据**被**完全处理**，即所有任务已经处理完所有的输入数据。
3. 将所有任务的**状态复制**到远程持久存储，**生成检查点。**当所有任务拷贝完成后，检查点就完成了
4. **恢复接收所有输入流。**

下图展示了一个一致性检查点的例子，这个算法读取数据，然后对奇数和偶数分别求和



3.5.2 从一致性检查点中恢复

在流应用执行期间，Flink周期性为应用程序生成检查点。一旦发生故障，**Flink会使用最新的检查点将应用状态恢复到某个一致性的点并重启应用。**图3-18显示了恢复过程。



应用程序恢复分为三个步骤:

1. **重启**整个应用程序。
2. 将所有**状态重置**为最新的检查点。
3. **恢复所有任务的处理。**

假设所有算子都将它们的状态写入检查点并从中恢复，并且所有输入流的消费位置都能重置到检查点生成那一刻，那么这种**检查点和恢复机制**可以为整个应用提供**精确一次**的**一致性保障。**输入流是否可以重置，取决于它的具体实现以及所消费外部系统是否提供相关接口。例如，像Apache Kafka这样的事件日志可以从之前的某个偏移读取记录。相反，如果是从socket消费而来则无法重置，因为socket一旦消耗了数据就会丢弃数据。

我们必须指出，Flink的检查点和恢复机制只能重置流应用内部的状态。根据应用所采用的数据汇算子，在**恢复期间**，**某些结果记录可能被多次发送到下游系统**，例如事件日志、文件系统或数据库。对于某些存储系统，Flink提供的数据汇可以保证了精确一次输出。

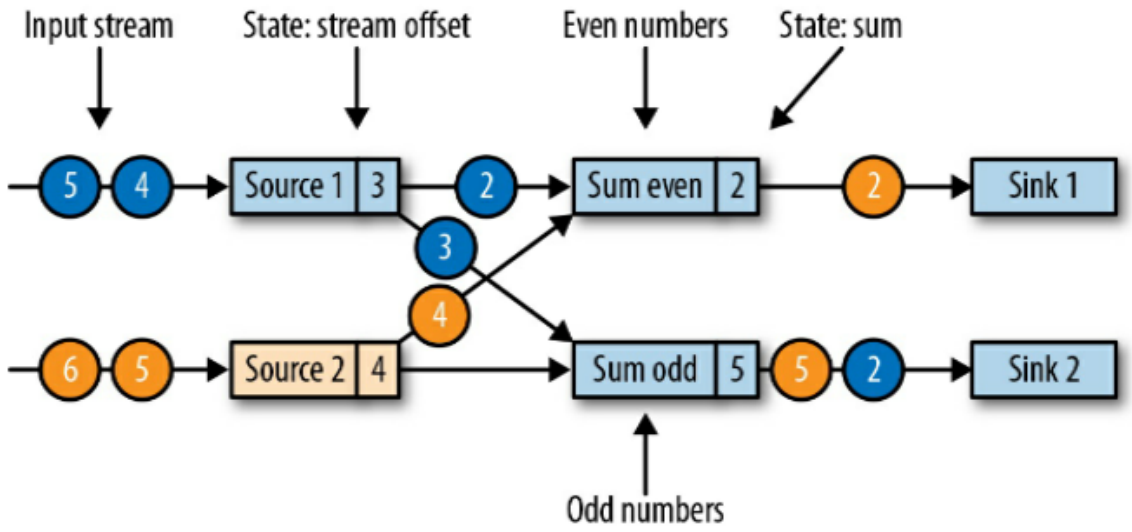
3.5.3 Flink检查点算法

Flink基于Chandy-Lamport的分布式快照算法来实现检查点。该算法并不会暂停整个应用程序，在部分任务持久化状态的过程中，其他任务可以继续执行。

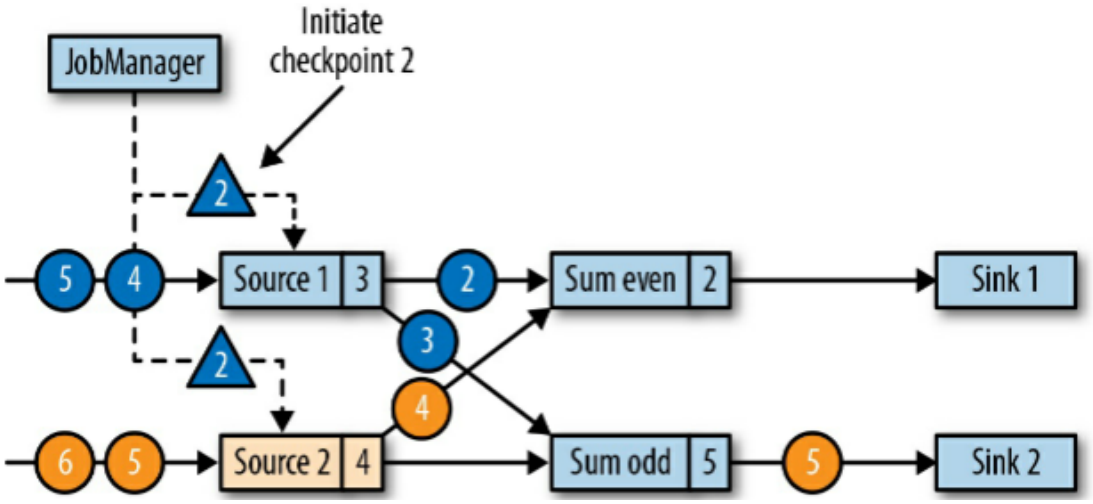
Flink的检查点算法使用一种称为检查点分隔符的特殊类型的记录，它与水位线类似。检查点分隔符携带一个检查点ID来标识它所属的检查点，分隔符从逻辑上将流分割为两个部分。由**检查点之前的记录 引起的所有状态修改**都包含在分隔符对应的检查点中，而由**屏障之后的记录引起的所有修改**都**不包含**在分隔符对应的检查点中。

下面我们通过一个简单的例子来解释这个算法

我们使用一个简单的流应用程序示例逐步解释该算法。应用程序由两个数据源任务组成，每个数据源任务消耗一个不断增长的数字流。数据源任务的分别输出奇数分区和偶数分区。每个分区都由一个任务处理，该任务计算所有接收到的数字的总和，并将更新后的总和发送给下游数据汇。该应用程序如图3-19所示。

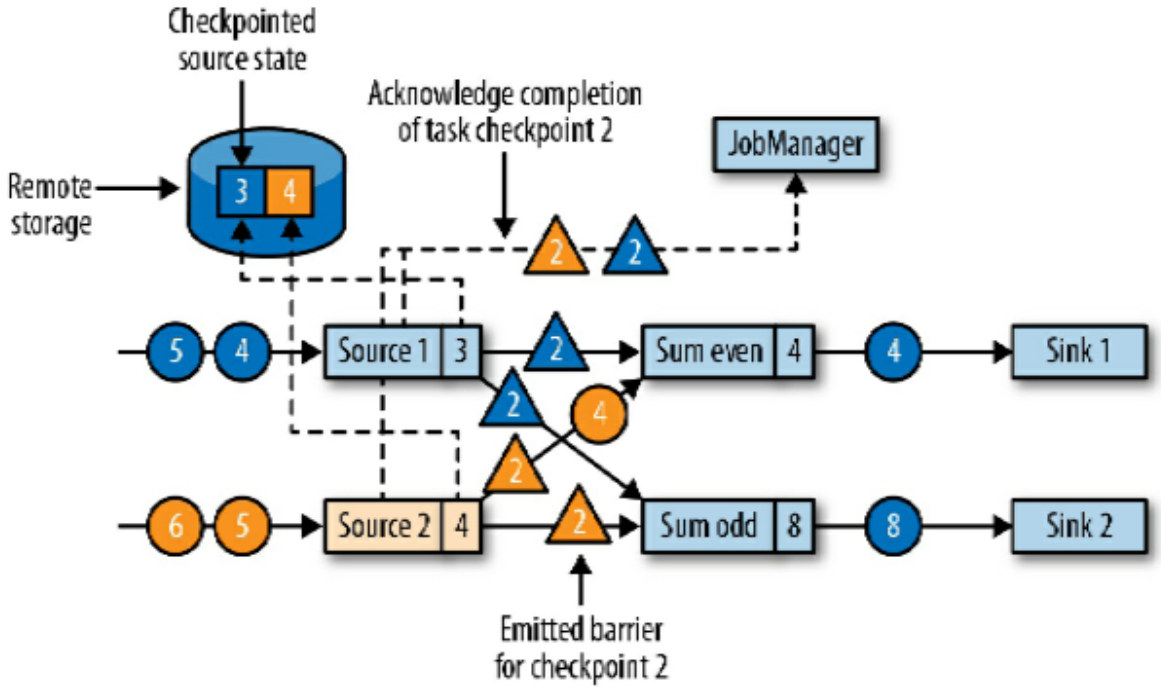


JobManager通过**向每个数据源任务 发送**一个新的**带有检查点编号的消息**来**启动检查点生成流程**，如图3-20所示。

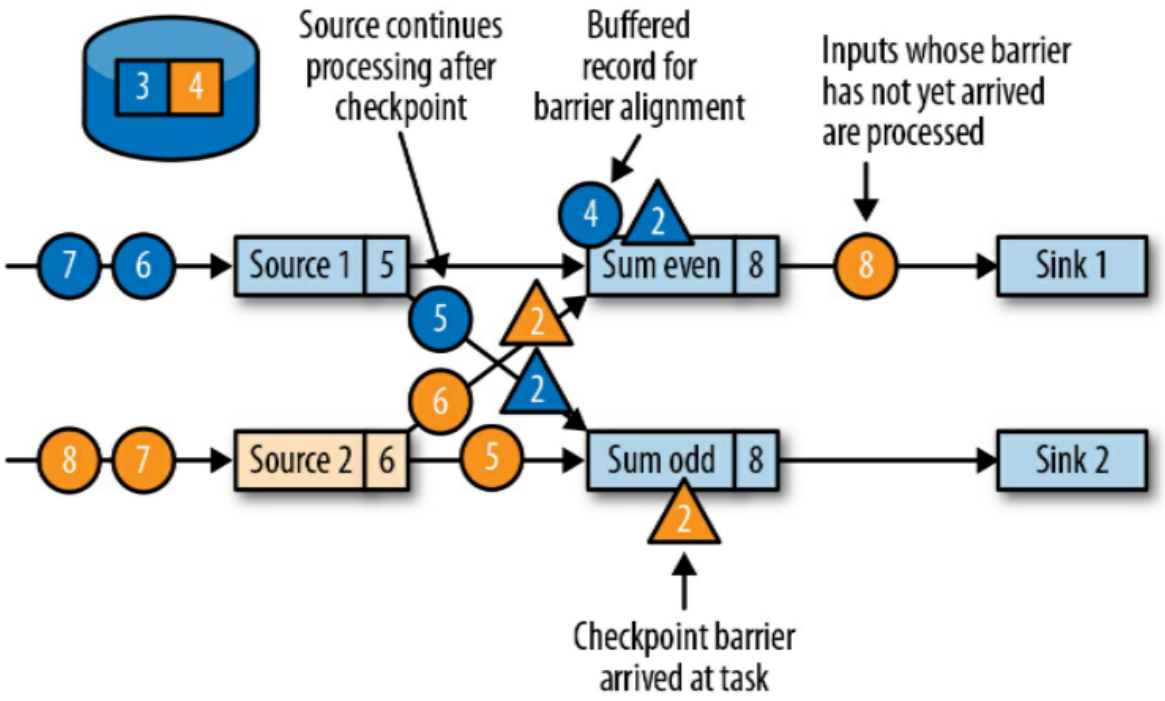


当数据源任务**接收到检查点消息**时，

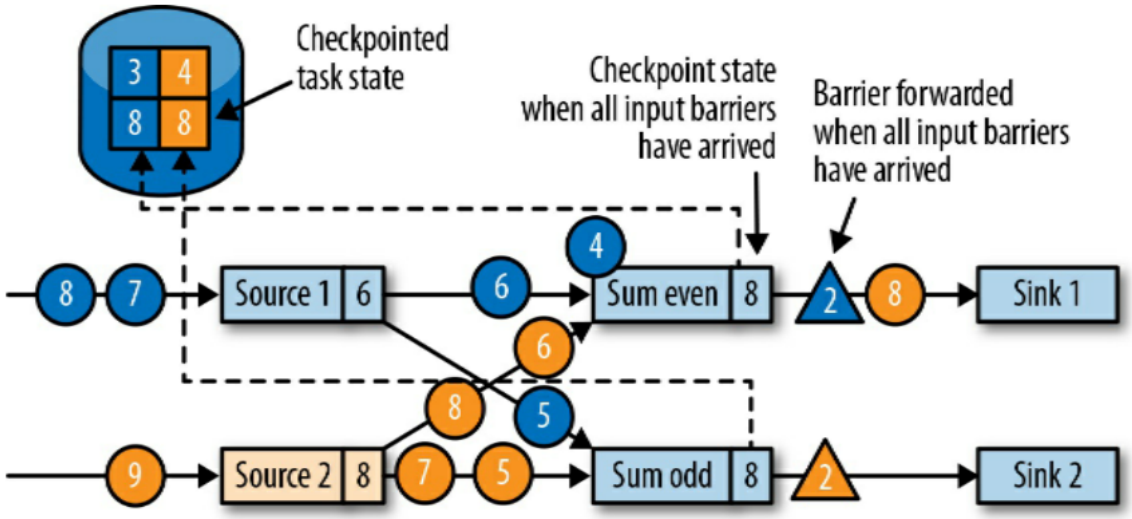
1. 它**暂停**处理数据流，并**利用状态后端 生成本地状态的检查点，并发送到远程存储**
2. 把该**检查点分隔符广播至**所有**下游任务。**
3. 状态后端会在检查点保存好之后通知TaskManager，**TaskManager会给JobManager发送确认消息。**
4. 在发出了分隔符之后，数据源将**恢复**正常的工作状态。
5. 如下图所示



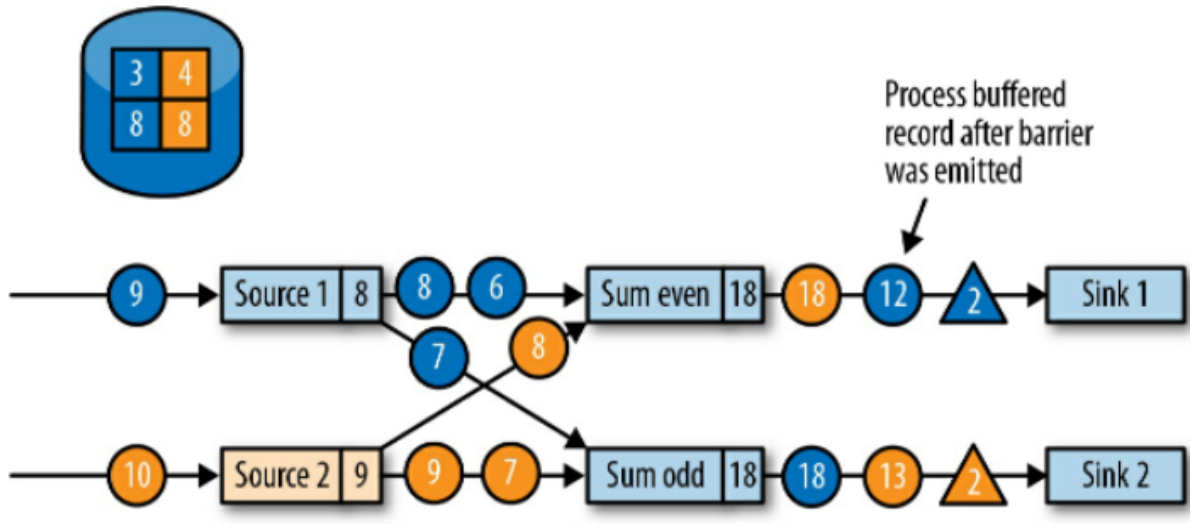
数据源发出的检查点分隔符被广播给下游任务。当**下游任务接收到新的检查点分隔符**时，将继续**等待来自所有其他上游任务**的分隔符**到达**检查点。在等待期间，它继续**处理**那些**尚未提供分隔符**的上游任务的记录，而那些**提供了分隔符**的上游任务的记录会被**缓存**，等待稍后处理。等待所有检查点到达的过程称为**检查点对齐**，如图3-22所示。



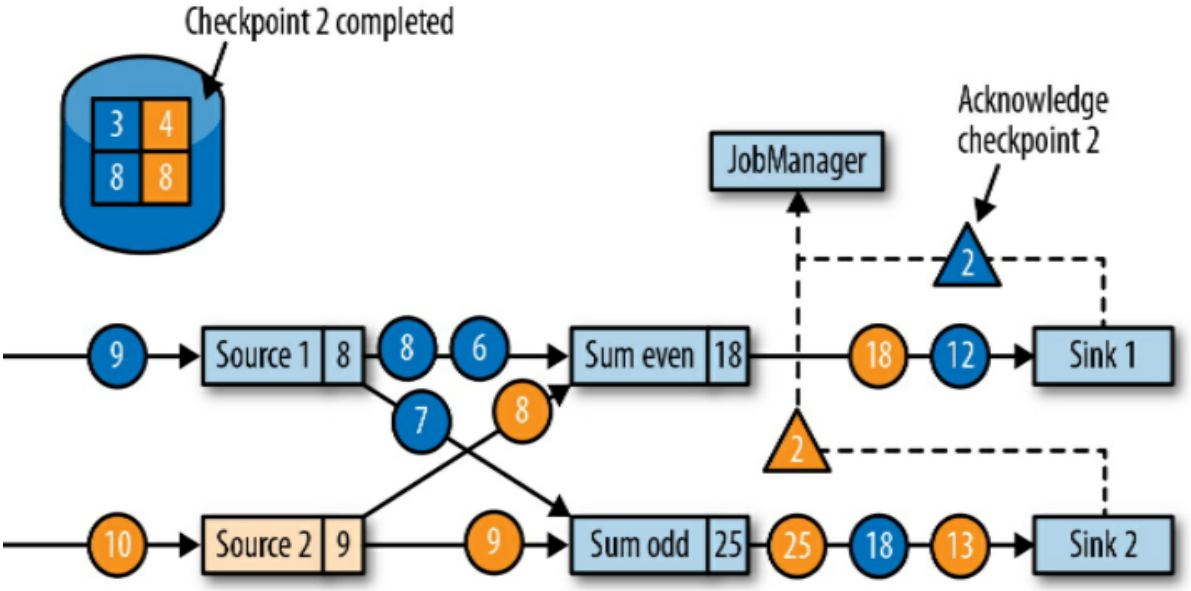
一旦一个任务从它的**所有上游任务**收到**分隔符**，它就会让状态后端**生成一个检查点**，并将检查点分隔符**广播**给它的所有下游任务，如图3-23所示。



在**发出检查点分隔符后**，任务就开始**处理缓冲的记录**。在处理完所有缓冲记录之后，任务会继续处理其输入流。图3-24显示了此时的应用程序。



最后，检查点分隔符到达数据汇。当数据汇接收到分割符时，会先进行对齐操作，然后将自身状态写入检查点，并向JobManager确认接收到该分隔符。**一旦应用的所有任务都发送了检查点确认，JobManager就会将应用程序的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。如前所述，已完成的检查点可用于从故障中恢复应用。



3.5.4 检查点对性能的影响

Flink的**检查点算法**从**流应用**中**产生**一致的分布式**检查点**，而**不会停止整个应用**。但是，它会增加应用的处理延迟。Flink实现了一些调整，可以在某些条件下减轻性能影响。

任务在将其状态写入检查点的过程中，将被阻塞。一种好的方法是先将检查点写入本地，然后任务继续执行它的常规处理，另一个进程负责将检查点传到远端存储。

此外，还可以在分隔符对齐的过程中不缓存那些已经收到分隔符所对应分区的记录，而是直接处理。但这会让一致性保证从精确一次降低到至少一次

3.5.5 保存点

Flink最有价值和最独特的功能之一是保存点。原则上，保存点的生成算法与检查点生成算法一样，因此可以把保存点看作是带有一些额外元数据的检查点。Flink不会自动生成保存点，而是需要用户显式的调用来生成保存点。

3.5.5.1 保存点的使用

给定一个**应用**和一个**兼容的保存点**，我们可以**从该保存点启动应用**。这将把应用的状态初始化为保存点的状态，并从获取保存点的位置运行应用。

保存点可以用在很多情况

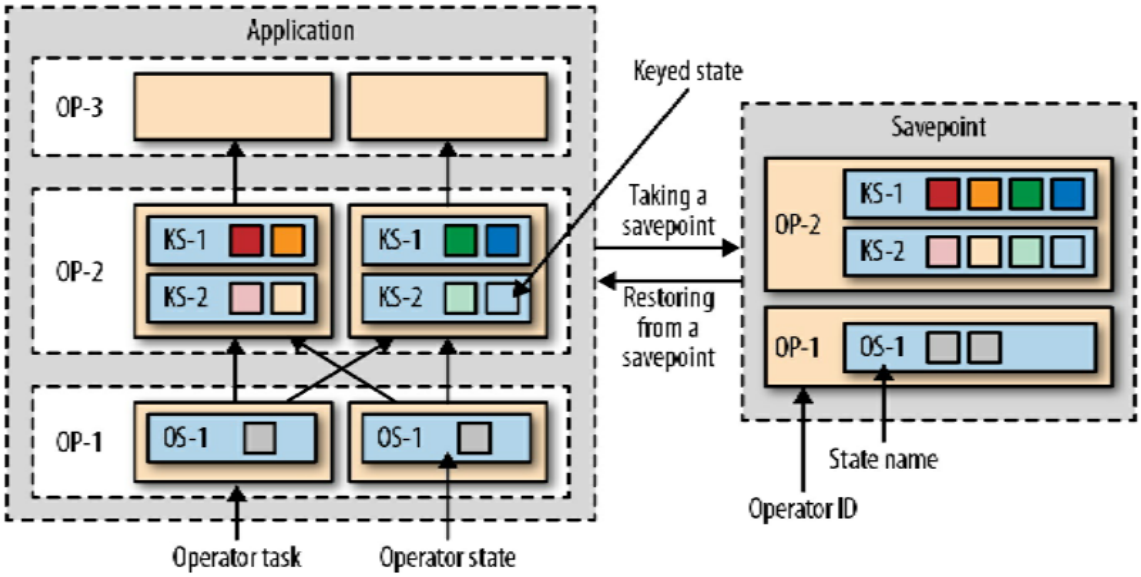
- 可以从保存点启动一个不同但兼容的应用程序。这意味着可以**修复一些小bug**之后从保存点重启
- 可以使用**不同的并行度**启动原应用
- 可以在**不同的集群**上启动原应用
- 可以使用保存点**暂停应用程序**并在**稍后恢复**它。这样就可以为其他高优先级的应用腾出集群资源
- 可以用保存点来完成**归档操作**

3.5.5.2 从保存点启动应用

在本节中，我们将描述Flink在从保存点启动时如何去初始化应用状态。

一个典型的应用程序包含多个状态，它们分布在不同算子的不同任务上。

下图显示了一个具有**三个算子的应用程序**，每个算子各运行两个任务。其中一个算子(OP-1)有一个算子状态(OS-1)，另一个算子(OP-2)有两个键值分区状态(KS-1和KS-2)。当生成保存点时，所有任务的状态都会被复制到一个持久化存储位置上。



保存点中的**状态**副本会**按照算子标识符和状态名称进行组织。**该算子标识符和状态名需要能够将保存点的状态数据映射到应用启动后的状态上。当从保存点启动应用程序时，Flink将保存点数据重新分发给相应算子的任务。

如果应用发生了修改，**只有那些算子标识符和状态名称没变的状态副本才能被成功还原。**默认情况下，Flink会分配唯一的算子标识符。但是，算子的标识符是基于其前面算子的标识符生成的。这样，假如上游的算子标识符发生了变化，那么下游的算子也会变化。因此，我们**强烈建议为操作符手动分配唯一标识符，而不依赖于Flink的默认赋值。**

第4章 设置Flink开发环境

Flink有一种适合开发时使用的执行模式：当程序的`execute()`方法被调用时，会在同一个JVM中以独立线程的方式启动一个JobManager线程和一个TaskManager线程。这样，整个Flink应用会以多线程的方式在同一个JVM进程中执行。该模式可用于**在IDE中执行Flink应用。**

由于单JVM执行模式的存在，你可以像调试其他程序一样在IDE中调试Flink应用。

第5章 DataStreamAPI

本章介绍了Flink的DataStream API的**基础知识。**我们将展示**常用的Flink流应用程序**的结构和组件，讨论Flink的**类型系统**和支持的数据类型，并介绍**数据转换**(data transformation)和**分区转换**(partitioning transformation)。读完这一章，你将知道**如何实现一个具有基本功能的流处理应用程序。**

5.1 Hello,Flink!

首先举一个简单的例子作为开始

```
/** 定义一个case class作为传感器读取数据的数据类型*/
/** Case class to hold the SensorReading data. */
case class SensorReading(id: String, timestamp: Long, temperature: Double)

/** 放主函数的object*/
/** Object that defines the DataStream program in the main() method */
object AverageSensorReadings {

  /** main() defines and executes the DataStream program */
  def main(args: Array[String]) {

    // 设置流式执行环境
    // set up the streaming execution environment
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 在应用中使用事件时间
    // use event time for the application
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    // 配置水位线
    // configure watermark interval
    env.getConfig.setAutoWatermarkInterval(1000L)

    // 从流式数据源中创建DataStream[SensorReading]对象
    // ingest sensor stream
    val sensorData: DataStream[SensorReading] = env
      // 添加传感器Source
      // SensorSource generates random temperature readings
      .addSource(new SensorSource)
      // 设置时间戳和水位线
      // assign timestamps and watermarks which are required for event time
      .assignTimestampsAndWatermarks(new SensorTimeAssigner)

    val avgTemp: DataStream[SensorReading] = sensorData
      // 将温度从华氏温度转换为摄氏温度
      // convert Fahrenheit to Celsius using an inlined map function
      .map( r =>
      SensorReading(r.id, r.timestamp, (r.temperature - 32) * (5.0 / 9.0)) )
      // 根据传感器id来分组数据
      // organize stream by sensorId
      .keyBy(_.id)
      // 按照1秒的滚动窗口分组
      // group readings in 1 second windows
      .timeWindow(Time.seconds(1))
      // 使用用户自定义函数来计算平均温度
      // compute average temperature using a user-defined function
      .apply(new TemperatureAverager)

    // 打印到控制台
    // print result stream to standard out
```

```
    avgTemp.print()

    // 开始执行应用
    // execute application
    env.execute("Compute average sensor temperature")
  }
}
```

构建一个典型的Flink流式程序需要以下几步

1. 设置执行**环境**
2. 从**数据源**中读取一条或多条流
3. 通过一系列**流式转换**来实现**应用逻辑**
4. 选择性地将结果输出到一个或多个**数据汇**中
5. **执行**程序

5.1.1 设置执行环境

Flink应用程序需要做的第一件事是**设置它的执行环境。**执行环境确定程序是**在本地机器上运行**还是**在集群上运行。**在DataStream API中，应用程序的执行环境由`StreamExecutionEnvironment`表示。

有两种设置执行环境的方式

1. 调用静态`getExecutionEnvironment()`方法来**检索执行环境。**此方法返回本地或远程环境，具体**取决于**调用该方法的**上下文。**如果通过连接到远程集群从提交客户端调用该方法，则返回远程执行环境。否则，它返回一个本地环境。
2. 也可以通过`createxxx`方法来显式设置执行环境，具体代码如下

```
// 创建一个本地的流式执行环境
val localEnv = StreamExecutionEnvironment.createLocalEnvironment()
// 创建一个远程的流式执行环境
val remoteEnv = StreamExecutionEnvironment.createRemoteEnvironment(
    "host",                   // JobManager的主机名
    1234,                     // JobManager的端口号
    "path/to/jarFile.jar"     // 需要传输到JobManager的JAR包
)
```

执行环境还提供了很多**配置选项**，比如

1. 通过`env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)`指定当前应用使用**事件时间**语义
2. 设置**并行度**
3. 启动**容错**等

5.1.2 读取输入流

`StreamExecutionEnvironment`提供了一系列**创建流式数据源**的**方法**，用来将数据流读取到应用中。这些数据流的**来源**可以是**消息队列**或者**文件**，也可以**动态生成。**

在实例中，读取代码如下

```
// 从流式数据源中创建DataStream[SensorReading]对象
// ingest sensor stream
val sensorData: DataStream[SensorReading] = env
    // 添加传感器Source
    // SensorSource generates random temperature readings
    .addSource(new SensorSource)
    // 设置时间戳和水位线
    // assign timestamps and watermarks which are required for event time
    .assignTimestampsAndWatermarks(new SensorTimeAssigner)
```

5.1.3 应用转换(Apply Transformation)

当获取到了`DataStream`，就可以对其应用转换(we can **apply a transformation** on it)。转换的类型有很多：

1. 有些转换可以生成新的`DataStream`，并且可能是不同类型的(eg. `DataStream[Int] => DataStream[String]`)
2. 有些转换不修改`DataStream`中的条目，而是通过**分区**或**分组**对其进行**重新组织。**
3. **应用程序的逻辑**是通过一系列**转换**定义的。

在实例中，转换代码如下

```
val avgTemp: DataStream[SensorReading] = sensorData
    // 将温度从华氏温度转换为摄氏温度
    // convert Fahrenheit to Celsius using an inlined map function
    .map( r =>
    SensorReading(r.id, r.timestamp, (r.temperature - 32) * (5.0 / 9.0)) )
    // 根据传感器id来分组数据
    // organize stream by sensorId
    .keyBy(_.id)
    // 按照1秒的滚动窗口分组
    // group readings in 1 second windows
    .timeWindow(Time.seconds(1))
    // 使用用户自定义函数来计算平均温度
    // compute average temperature using a user-defined function
    .apply(new TemperatureAverager)
```

5.1.4 输出结果

流应用程序通常将其结果发送到一些**外部系统**(external system)，如Apache Kafka、文件系统或数据库。Flink提供了一组**流式数据汇**，可用于**将数据写入不同的系统**。也可以实现自己的流式数据汇。还有**一些**应用程序**不发出结果**，而是通过Flink的可查询状态(queryable state)功能在**内部保存**结果。

在我们的示例中，会将`DataStream[SensorReading]`中的记录作为结果输出。每个记录包含传感器在5秒内的平均温度。通过调用print()将结果流写入标准输出:

```
avgTemp.print()
```

5.1.5 执行

当应用定义完成后，可以通过调用`StreamExecutionEnvironment.execute()`来执行它:

```
env.execute("Compute average sensor temperature")
```

Flink程序都是通过**延迟计算**(lazily execute)的方式执行。

- 也就是说，那些创建数据源和转换操作的API调用不会立即触发任何实际的数据处理。
- 相反，这些API调用只是**在执行环境**中创建一个**执行计划**。该**计划包括**从环境创建的**流式数据源**以及应用于这些数据源之上的**一系列转换**。
- 只有在调用`execute()`时，系统**才会触发程序的执行**。

构建完成的计划会被转换为**JobGraph**并**提交给JobManager**执行。

- 根据**执行环境的类型**，系统可能需要将JobGraph发送到作为**本地**线程启动的JobManager，或将JobGraph发送到**远程**JobManager。
- 如果JobManager**远程**运行，除了JobGraph之外，我们**还需要**提供一个包含应用程序的所有类和所需依赖项的**JAR文件**。

5.2 转换操作

在本节中，我们将概述DataStream API中的**基本转换**。

- 流式转换以一个或多个数据流作为输入，并将它们转换为一个或多个输出流。
- 编写一个DataStream API程序**本质上**可以归结为：通过**组合**不同的**转换**来**创建**一个**满足应用逻辑**的**Dataflow图**。

大多数流式转换都基于用户自定义的函数来完成。**这些函数封装了用户的逻辑**，指定了如何将输入流的元素转换为输出流的元素。**函数**可以**通过实现**某个特定转换的**接口类**来定义，例如下面的`MapFunction`

```
class MyMapFunction extends MapFunction[Int, Int] {
    override def map(value: Int): Int = value + 1
}
```

DataStream API为那些最常见的数据转换操作都提供了对应的转换抽象，我们将DataStreamAPI的**转换分为四类**

1. 作用于**单个事件**的**基本转换**
2. 针对**相同键值**事件的**KeyedStream转换**
3. 将多条数据流**合并**为一条或将一条数据流**拆分**成多条流的转换
4. 对流中的事件进行**重新组织**的**分发转换**

5.2.1 基本转换

基本转换**单独处理**每个事件，这意味着**每个输出记录**都是**由单个输入记录生成**的。常见的基本转换函数有：简单的值转换、记录拆分或过滤等。

5.2.1.1 Map

通过调用`DataStream.map()`方法可以指定**map转换**来**产生一个新的**`DataStream`。它将每个**输入事件**传递给用户自定义的映射器(user-defined mapper)，**映射器返回一个输出事件**，这个输出事件可能是不同类型的(eg, DataStream[Int] => DataStream[String])。图5-1显示了将每个正方形转换为圆形的map转换。

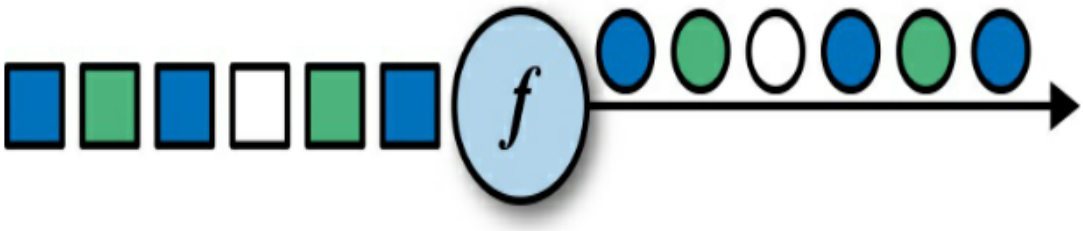


Figure 5-1. A map operation that transforms every square into a circle of the same color

MapFunction的两个类型参数分别是是输入事件的类型和输出事件的类型，`MapFunction`的`map()`方法将每个输入事件准确地转换为一个输出事件:

```
// T: 输入元素的类型
// O: 输出元素的类型
MapFunction[T,O]
    > map(T): O
```

下面举一个简单的例子

```
val sensorIds: DataStream[String] = reading.map(new MyMapFunction)

class MyMapFunction extends MapFunction[SensorReading, String] {
    override def map(r: SensorReading): String = r.id
}
```

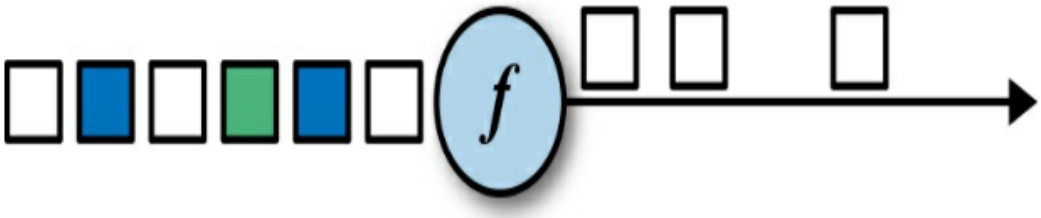
也可以用Lambda表达式进一步简化

```
val sensorIds: DataStream[String] = reading.map(r => r.id)
```

5.2.1.2 Filter

fliter转换通过一个**返回值为** `Boolean` **类型的函数**来**决定事件的去留**:

- 如果返回值为true，那么它会保留输入事件并且将其转发到输出，
- 否则它会把事件丢弃。
- 通过调用**DataStream.filter()**方法可以指定过滤器转换，并生成与输入DataStream**相同类型**的输出DataStream。
- 图5-2显示了一个只保留白色方块的过滤操作。



`FilterFunction` **类型参数**是**输入流的类型**，它的 `filter()` 方法**接收一个输入事件**并**返回一个布尔值**:

```
FilterFunction[T]
    > filter(T): Boolean
```

下面举个简单的例子

```
var filteredSensors = readings.filter(r => r.temperature >= 25)
```

5.2.1.3 FlatMap

flatMap转换与map类似，但是它可以为每个输入事件生成**零个**、**一个**或**多个 输出事件。**

图5-3显示了一个基于传入事件的颜色区分其输出的flatMap操作。

- 如果输入是白色方块，则不加改动直接输出。
- 将黑色方块复制，
- 将灰色方块丢弃掉。

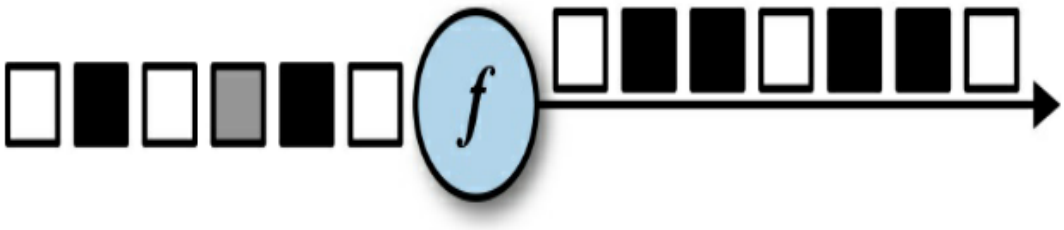


Figure 5-3. A flatMap operation that outputs white squares, duplicates black squares, and drops gray squares

flatMap函数定义如下，可以通过向`Collector`对象传递数据的方式来返回零个、一个或多个事件作为结果

```
// T: 输入元素的类型
// O: 输出元素的类型
FlatMapFunction[T, O]
    // 返回值为Unit，也就是不返回
    // Collector[O]作为输出参数
    > flatMap(T, Collector[O]): Unit
```

flatMap函数还可以如下定义

```
FlatMapFunction[T, O]
    > flatMap(T):  TraversableOnce[O]
```

下面举一个简单的例子

```
val words = sensorData.flatMap(r => r.id.split(" "))
```

5.2.2 基于KeyedStream的转换

KeyedStream抽象可以从逻辑上**将事件按照键值**分配到**多个独立的事件子流**中。

KeyedStream可以根据键来维护内部状态，所有具有相同键的事件可以访问相同的状态。

接下来先介绍**keyBy转换**，它可以将要一个`DataStream`转换为一个`KeyedStream`。然后介绍**滚动聚合**和**Reduce**，它们可以作用在`KeyedStream`上

5.2.2.1 keyBy

keyBy转换通过**键**来**将DataStream转换为KeyedStream。**数据流中的事件会根据不同的键被分配到不同的分区(**partition**)，具有**相同键的所有事件**都由**下游算子**的**同一个任务**处理。

我们假设以输入事件的颜色作为键，图5-4将黑色事件分配给一个分区，将所有其他事件分配给另一个分区。

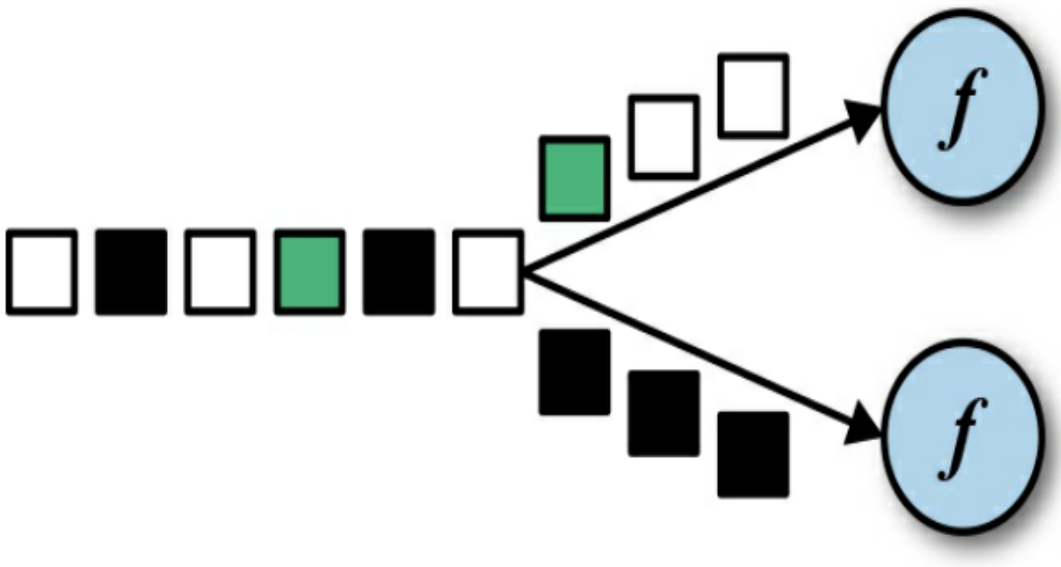
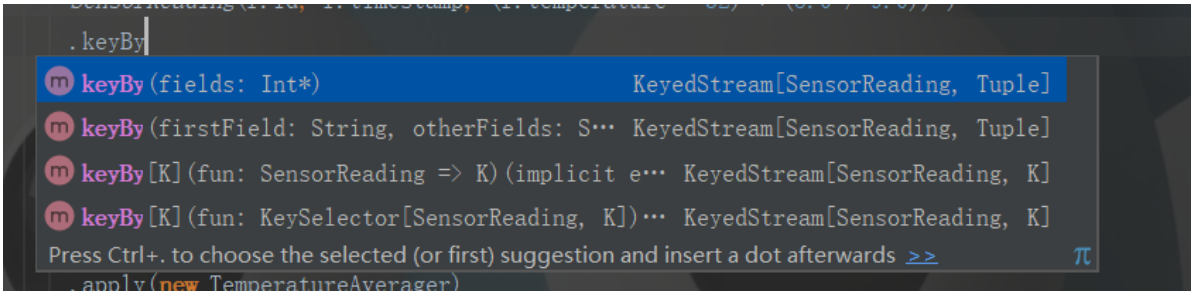


Figure 5-4. A keyBy operation that partitions events based on color

keyBy可以用多种方式来设置如何分类，如下所示



下面举一个keyBy的例子

```
val readings: DataStream[SensorReading] = ...
val keyed: KeyedStream[SensorReading, String] = readings.keyBy(r => r.id)
```

5.2.2.2 滚动聚合

滚动聚合 应用于KeyedStream上，它生成一个包含**聚合结果**（如求和、最小值和最大值）的DataStream。

- 滚动聚合操作符为**每个键**保存**一个聚合值**。
- 对于每个**输入事件**，算子**更新**相应的**聚合值**，并将**更新后的值**作为输出事件**发送给下游**。
- 滚动聚合操作需要接收一个用于指定**聚合目标字段**的参数，该参数指定在哪个字段上计算聚合。

DataStream API提供了以下滚动聚合方法:

名称	描述
`sum()`	滚动计算输入流在指定字段上的**和**
`max()`	滚动计算输入流在指定字段上的**最大值**
`min()`	滚动计算输入流在指定字段上的**最小值**
`minBy()`	滚动计算输入流中迄今为止最小值，返回该值所在事件
`maxBy()`	滚动计算输入流中起劲为止最大值，返回该值所在事件

注意：**不能**将多个**滚动聚合**方法**组合**使用，**每次只能计算一个。**

例子：对一个`Tuple3[Int, Int, Int]`类型的数据在第一个字段上按照键值分组，然后滚动计算第二个字段的和

```
val inputStream: DataStream[(Int, Int, Int)] = env.fromElements(
    (1, 2, 2),
    (2, 3, 1),
    (2, 2, 4),
    (1, 5, 3))

val resultStream: DataStream[(Int, Int, Int)] = inputStream
    .keyBy(0)
    .sum(1)

"""output
(1, 2, 2)
(2, 3, 1)
(2, 5, 1)
(1, 7, 2)
第一个字段是分组，第二个字段是计算之后的和，第三个字段没有意义
"""
```

5.2.2.3 Reduce

reduce转换是**滚动聚合**的**一般化**(generalization)。

- 它在KeyedStream上应用了一个ReduceFunction，该函数将每个**输入事件**与**当前的reduce结果**进行一次**组合**，并输出一个DataStream。
- reduce**不会改变DataStream的类型**，输出流的类型与输入流的类型相同。

ReduceFunction接口定义如下

```
// T: 元素类型
ReduceFunction[T]
    > reduce(T, T): T
```

下面举一个reduce转换的例子。在下面的例子中，数据流是会以语言类型作为键来进行分区，最终结果是针对每个语言产生一个不断更新的单词列表:

```
val inputStream: DataStream[(String, List[String])] = env.fromElements(
    ("en", List("tea")),
    ("fr", List("vin")),
    ("en", List("cake")))

val resultStream: DataStream[(String, List[String])] = inputstream
    .keyBy(0)
    .reduce((x, y) => (x._1, x._2 ::: y._2))

"""output
("en", List("tea"))
("fr", List("vin"))
("en", List("tea", "cake"))
"""
```

5.2.3 多流转换

许多应用需要将多个输入流联合起来处理，还有一些应用需要将一条流分割成多条子流以应用不同的逻辑。下面，我们将讨论那些同时处理多个输入流或产生多个输出流的DataStream API转换。

5.2.3.1 Union

`DataStream.union()`方法可以合并两个或多个**相同类型**的`DataStream`，并生成一个新的类型相同的`DataStream`。

图5-5显示了一个union操作，它将黑色和灰色事件合并到单个输出流中。

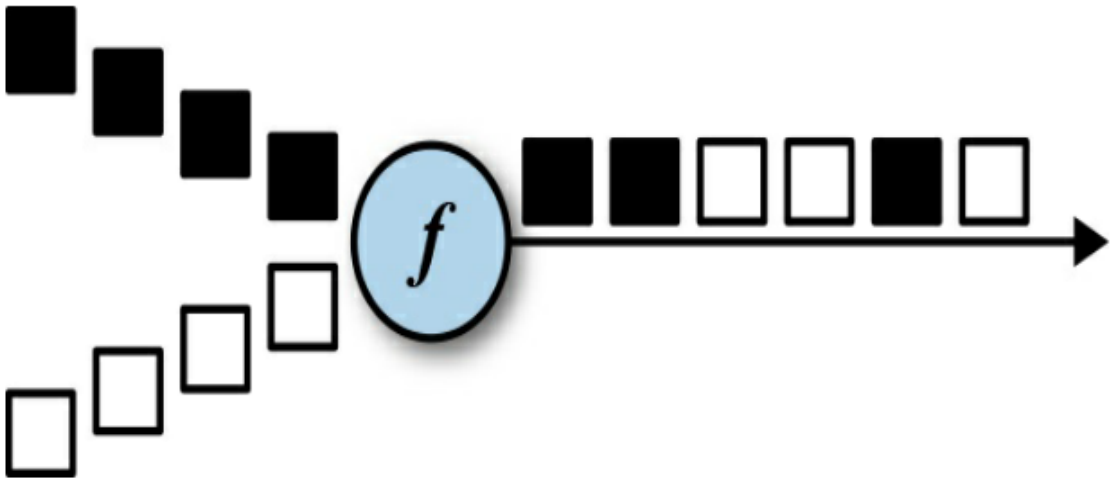


Figure 5-5. A union operation that merges two input streams into one

union执行过程中，来自两条流的事件会以**FIFO的方式合并**，其**顺序无法保证**(**The operator does not produce a specific order of events**.)。此外，union操作符**不会**对数据进行**去重**。每个输入事件都被发送到下游。

下面举个把三条数据流合并为一条的例子

```
val parisStream: DataStream[SensorReading] = ...
val tokyoStream: DataStream[SensorReading] = ...
val rioStream: DataStream[SensorReading] = ...
val allCities: DataStream[SensorReading] = parisStream.union(tokyoStream,
rioStream)
```

5.2.3.2 Connect，coMap，coFlatMap

考虑这样一个应用，它监视森林区域，并在发生火灾的风险很高时发出警报。应用从温度传感器和烟感传感器上接收数据。当**温度超过给定的阈值 并且 烟雾水平很高**时，应用程序会发出火灾警报。这时，为了判断两者是否同时成立，我们需要**合并两条流**来根据两条流的信息来**综合判断**

由此可见，合并两个流的事件是流处理中非常常见的需求。下面来看看相关的API

`DataStream.connect()`方法接收一个DataStream并返回一个ConnectedStreams对象，该对象表示两个联结在一起的流:

```
val first: DataStream[Int] = ...
val second: DataStream[String] = ...

val connected: ConnectedStreams[Int, String] = first.connect(second)
```

ConnectedStreams对象提供了map()和flatMap()，具体用法略

默认情况下，connect()不会在两个流的**事件之间 建立关系**，因此两个流的事件被**随机分配**给算子任务。这种行为会产生**不确定的结果**，通常是不希望看到的。**为了在ConnectedStreams上产生确定性结果，可以将connect()与keyBy()或broadcast()结合使用。**

- 当使用`keyBy()`时，`connect()`转换会将两条数据流中具有**相同键的事件**发送到**同一个算子任务**上
- 而当使用`broadcast()`时，两条流中有一条被广播，它的事件被分发给下游算子的所有任务上。这样可以保证联合处理这两个输入流的元素。

5.2.3.1 Split和Select

split是union的逆操作。它将**输入流 分割**为与输入流相同类型的两个或**多个输出流。每个输入事件**可以被发送给**零个**、**一个**或**多个 输出流。**因此，split操作还可以用于**过滤**或**复制**事件。

图5-6显示了一个split算子，它将所有白色事件与其他事件分开，发往不同的数据流。

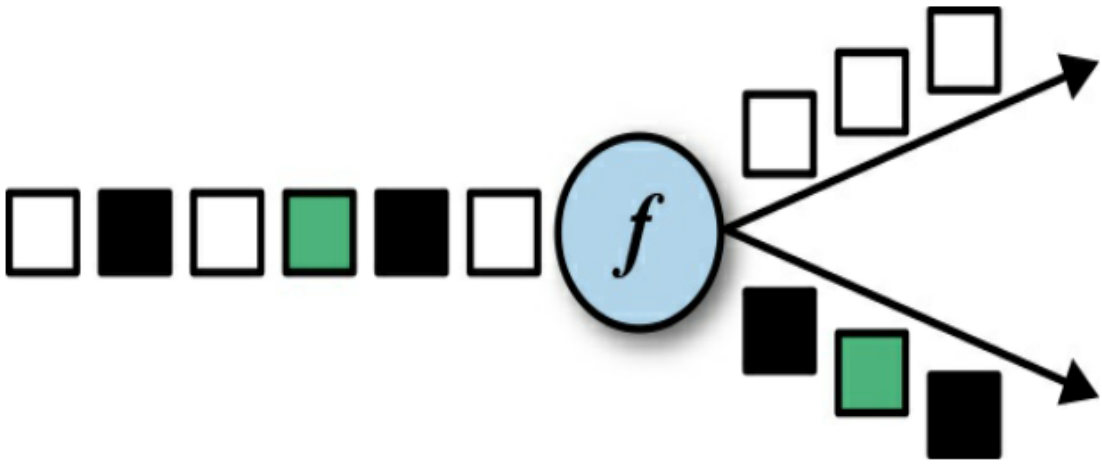


Figure 5-6. A split operation that splits the input stream into a stream of white events and a stream of others

split()方法以一个OutputSelector函数接口作为参数。

```
// IN: 同DataStream的元素类型
OutputSelector[IN]
    > select(IN): Iterable[String]
```

每个输入事件到来时都会调用`OutputSelector.select()`方法，并随即返回一个`java.lang.Iterable[String]`。返回的这个`String`列表中的每个`String`是这个事件所属的输出流的名称。

`split()`方法返回一个`SplitStream`对象，这个对象提供一个`select()`方法，通过指定输出名称从SplitStream中选择一条或多条流。

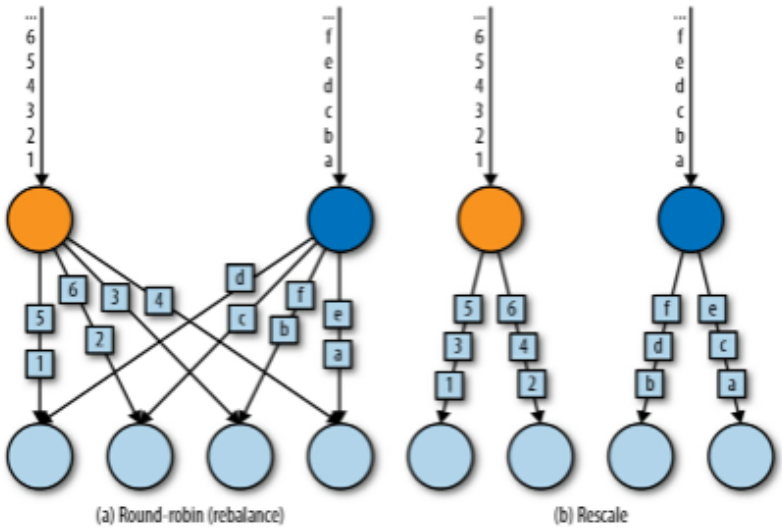
实例5-2：将一个数字流分成一个大数字流和一个小数字流

```
val inputStream: DataStream[(Int, String)] = ...
val splitted: SplitStream[(Int, String)] = inputStream
    .split(t => if (t._1 > 1000) Seq("large") else Seq("small"))

val large: DataStream[(Int, String)] = splitted.select("large")
val small: DataStream[(Int, String)] = splitted.select("small")
val all: DataStream[(Int, String)] = splitted.select("large", "small")
```

5.2.4 分发转换

当使用DataStream API构建应用程序时，系统会根据**操作语义**和配置的**并行度**来**自动选择数据分区策略**并将数据转发到正确的目标。有时我们可能希望能够**手动选择分区策略**。在本节中，我们将介绍DataStream中用于控制分区策略或自定义分区策略的方法。

下面是常见的分区策略

名称	描述
随机	随机数据交换策略由DataStream.shuffle()方法实现。该方法将事件随机分配到下游算子的并行任务中。
轮流(Round-Robin)	轮流方法将输入流的事件以轮流方式均匀分配给后继任务。
重调(Rescale)	rescale()也是以轮流的方式对事件进行分发，但是每个上游任务只与一部分下游任务建立发送通道。当上游任务数远小于下游任务数时，这种方法比较好用，下图展示了轮流和重调的区别 
广播(Broadcast)	broadcast()将输入流中的事件复制，并发送给发送给下游算子的所有并行任务。
全局(global)	global()方法将**输入数据流的所有事件**发送给下游操作符的**第一个并行任务**。必须小心使用这种分区策略，因为将所有事件路由到同一任务可能会**影响应用程序性能**
自定义	如果所有预定义的分区策略都不合适，你可以利用partitionCustom()方法来自定义分区策略

自定义分区例子如下

```
val numbers: DataStream[(Int)] = ...

numbers.partitionCustom(myPartitioner, 0)

object myPartitioner extends Partitioner[Int]{
    val r = scala.util.Random

    override def partition(key: Int, numPartitions: Int): Int = {
        if (key < 0) 0 else r.nextInt(numPartitions)
    }
}
```

5.3 设置并行度

算子的并行度可以在**执行环境级别**或**算子级别**进行**控制。默认情况**下，应用的所有**算子的并行度**被设置为应用**执行环境的并行度。**而**执行环境的并行度**将**根据**应用**启动时所处**的上下文**自动初始化。**

- 如果应用程序在**本地执行环境**中运行，则将并行度设置为**与CPU内核数量相等。**
- 在向运行的**Flink集群**提交应用程序时，除非用户显示指定，否则环境并行度将设置为**集群的默认并行度**

最好将**算子并行度**设置为**随环境并行度变化的值**而不要设置为定值，例如：假设环境并行度为`x`，可以设置算子并行度为`y = x/2`。当在本机运行时`x=8, y=4`。而在集群运行时`x=32,y=16`。这样当运行环境变化时，算子并行度也可以随之变化。

下面的例子演示了如何获取环境并行度以及如何设置环境并行度

```
// 获取环境并行度
val env = StreamExecutionEnvironment.getExecutionEnvironment
val defaultParallelism = env.getParallelism

// 设置环境并行度
env.setParallelism(32)
```

下面的例子演示了如何设置算子的并行度

```
// 获取环境并行度
val env = StreamExecutionEnvironment.getExecutionEnvironment
val defaultParallelism = env.getParallelism

val result = env.addSource(new CustomSource)
    // 设置map的并行度为默认并行度的两倍
    .map(new MyMapper).setParallelism(defalutParallelism * 2)
    // print数据汇的并行度固定为2
    .print().setParallelism(2)
```

5.4 类型

Flink DataStream应用所处理的**事件**会**以数据对象的形式存在。**这些**数据对象**需要能够被**序列化**和**反序列化**，以通过网络发送它们，或将它们写入状态后端、检查点和保存点，或从状态后端读取。Flink使用**类型信息(**type information)的概念来表示数据类型，并为每种数据类型**自动生成**特定的**序列化器、反序列化器和比较器。**

一般情况下，Flink都可以**自动提取**数据对象的**类型信息**，但当自动提取器**失效**时，我们也需要**手动**指定类型信息。

本节我们会讨论Flink支持的类型，如何为数据类型创建类型信息，以及当Flink无法自动推断函数的返回类型的时如何以提示的方式帮助类型系统。

5.4.1 支持的数据类型

Flink支持Java和Scala中可用的所有常见数据类型，可以分为以下类别

- **原始类型**
- Java和Scala**元组**
- Scala**样例类**(case class)
- **POJO**
- 一些特殊类型：数组、列表、映射、枚举等

对于**POJO**的解释：如果一个类满足如下条件，它会被Flink看作POJO

- 是一个公有类
- 有一个公有的无参默认构造函数
- 所有字段都是公有的或者提供了相应的`getter`以及`setter`方法
- 所有字段类型都必须是Flink支持的

对**特殊类型**的解释：Flink支持多种特殊类型，比如

- 原始或者对象类型的数组；
- Java的ArrayList、HashMap和Enum类型
- Hadoop的Writable类型。
- Scala的Either、Option和Try类型以及Flink内部实现的Java版本的Either类型

5.4.2 为数据类型创建类型信息

在Flink的类型系统中，**核心类是**`TypeInformation`。它**为系统**生成**序列化器**和**比较器**提供了**必要的信息**。当应用提交执行时，Flink的类型系统尝试为框架处理的每个数据类型**自动推断**`TypeInformation`。因此，**大多数情况下**，我们都**没有必要手动**指定类型信息，但当自动推断**失灵**时，就需要我们为特定数据类型**手动**生成TypeInformation了。

下面举几个生成TypeInformation的例子

```
// 原始类型的TypeInformation
val stringType: TypeInformation[String] = Types.STRING
// Scala元祖的TypeInformation
val tupleType: TypeInformation[(Int, Long)] = Types.TUPLE[(Int, Long)]
// case class的TypeInformation
val caseClassType: TypeInformation[Person] = Types.CASE_CLASS[Person]
```

5.4.3 显式提供类型信息

显示提供TypeInformation的方式有**两种。第一种**是通过**实现**`ResultTypeQueryable`**接口**来扩展函数。如下面例子所示

```
class Tuple2ToPersonMapper extends MapFunction[(String, Int), Person] with
ResultTypeQueryable[Person] {
    override def map(v: (String, Int)): Person = Person(v._1, v._2)

    //实现ResultTypeQueryable
    override def getProducedType: TypeInformation[Person] =
Types.CASE_CLASS[Person]
}
```

第二种，在定义Dataflow时使用Java DataStream API中的`returns()`**方法**来**显式指定**某算子的返回类型

```
persons = inputStream
    .map(t => new Person(t._1, t._2))
    .returns(Types.CASE_CLASS[Person])
```

5.5 定义键和引用字段

在Flink中有很多需要使用**键索引**(key specification)和**字段引用(**field reference)的地方。**Flink采用各种各样的灵活方式来定义键**：通过元素的字段位置来定义、通过基于字符串的字段表达式来定义、通过KeySelector函数来定义

5.5.1 字段位置

如果数据类型是**元组**，则只需使用对应元组元素的**字段位置**就可以定义键。

例如下面这个例子使用**元组的第二个字段**作为**输入流的键值**

```
val input: DataStream[(Int, String, Long)] = ...
val keyed = input.keyBy(1)
```

此外，还可以使用多个元组字段来定义复合键

```
val keyed2 = input.keyBy(1, 2)
```

5.5.2 字段表达式

另一种定义键和选择字段的方法是使用**基于字符串的字段表达式**。字段表达式适用于元组、pojo和case类。

```
// 最简单的字段表达式
val keyedSensors = sensorStream.keyBy("id")

// 在元组类型上使用字段表达式
val keyed = inputStream.keyBy("_1")

// 使用点运算符来嵌套POJO字段为键值
val persons = inputStream.keyBy("address.zip")

// 使用通配符 `_` 来选择元组中的全部字段作为键
val keyed = inputStream.keyBy("birthday._")
```

5.5.3 KeySelector函数

第三种指定键的方式是使用**KeySelector函数**。它可以从输入事件中提取键

```
// T: 输入元素的类型
// KEY: 键值类型
KeySelector[IN, KEY]
    > getKey(IN): KEY
```

下面的例子会返回元组中的最大字段来作为键值

```
val input = DataStream[(Int, Int)] = ...
val keyedStream = input.keyBy(value => math.max(value._1, value._2))
```

5.6 实现函数

在DataStream API中有很多地方需要使用自定义函数。本节将介绍Flink中定义函数的几种方式

5.6.1 函数类

Flink中所有**用户自定义函数**都是以**接口**或者**抽象类**的形式对外暴露的，如MapFunction、FilterFunction和ProcessFunction等

我们可以通过实现接口或者继承抽象类的方式来定义函数，例如下面的例子

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

val filted = sentences.filter(new MyFilter())
```

> **函数必须是可序列化的**
>
> Flink使用Java序列化来序列化所有函数对象，以便将它们发送给对应的算子任务。用户函数中包含的所有内容都必须是可序列化的。如果您的函数需要一个非序列化的对象实例，您可以将其实现为一个富函数，并在open()方法中初始化非序列化字段，或者重写Java序列化和反序列化方法。

5.6.2 Lambda函数

也可以通过Lambda表达式的方式来定义函数

```
val filted = sentences.filter(_.contains("flink"))
```

5.6.3 富函数

有时，我们需要在**函数 处理第一个记录之前**进行一些**初始化工作**或者取得**函数执行**相关的**上下文**信息。DataStream API提供了丰富的函数，它和我们之前见到的普通函数相比可以对外提供更多功能。

DataStream API中的所有转换函数都有对应的富函数，富函数的使用位置和普通函数以及Lambda函数相同。富函数的名称以Rich开头，例如RichMapFunction、RichFlatMapFunction等。

当使用富函数时，你可以对应函数的生命周期实现两个额外的方法：

- `open()`方法是富函数的**初始化方法**。它在每个任务首次调用转换方法之前调用一次
- `close()`方法是富函数的**终止方法**，会在每个任务**最后一次调用转换方法**后**调用一次**。因此，它通常用于**清理**和**释放**资源。
- 另外，还可以使用富函数自带的`getRuntimeContext()`方法来从函数的Runtime中获取一些信息

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

    override def open(config: Configuration): Unit = {
```

```
        subTaskIndex = getRuntimeContext.getIndexOfThisSubtask
        //进行一些初始化工作
    }

    override def flatMap(in: Int, out: Collector[(Int, Int)]): Unit = {
        //子任务的编号从0开始
        if(in % 2 == subTaskIndex){
            out.collect((subTaskIndex, in))
        }
        //做一些额外处理工作
    }

    override def close(): Unit = {
        //做一些清理工作
    }
}
```

5.7 导入外部和Flink依赖

在实现Flink应用时经常需要添加一些外部依赖。应用在执行时，必须能够访问到所有依赖。**默认情况**下，Flink集群**只加载核心API依赖**(DataStream和DataSet API)，对于应用的**其他依赖**则必须**显式提供。**

有**两种方法**来确保所在执行应用时可以访问到所有依赖:

1. 将所有依赖打进应用的Jar包中，生成一个“胖Jar”
2. 将依赖放到Flink的`./lib`目录下，这样在Flink进程启动时就会将依赖加载到Classpath中

推荐使用第一种方式。

第6章 基于时间和窗口的算子

在本章:

1. 首先，我们将学习如何配置**时间特性**、**时间戳**和**水位线。**
2. 然后，我们将介绍**处理函数(process functions)**，它提供了对时间戳和水位线的访问并可以注册计时器，属于比较底层的API。
3. 接下来，我们将使用Flink的**窗口API**，它针对几个最常见的窗口类型都提供了内置实现。
4. 你还将了解如何**自定义窗口算子。**
5. 最后，我们将讨论数据流的**JOIN函数**以及**处理延迟事件**的策略。

6.1 配置时间特性

在DataStream API中，您可以使用**时间特性(the time characteristic)**告诉Flink在创建窗口时如何定义时间。时间特性是`StreamExecutionEnvironment`的一个属性，它接受以下值:

值	解释
ProcessingTime(处理时间)	指定算子根据本地的**机器时钟**来**确定数据流当前的时间**。好处是延迟比较低，坏处是精度很差
EventTime(事件时间)	指定算子使用来自**数据本身的信息**来确定当前时间。**每个事件**都带有一个**时间戳**，**系统的逻辑时间**由**水位线**定义。
IngestionTime(摄入时间)	指定每个接收的记录都把**数据源算子的处理时间**作为**事件时间的时间戳**，并**自动生成水位线**。

下面举一个设置时间特性的例子

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
// 在应用中使用事件时间
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
```

6.1.1 分配时间戳和生成水位线

为了能正常在**事件时间**的时间特性**下**工作，**应用程序**需要向Flink提供两个重要的信息

1. **每个事件**都必须与**时间戳**关联，时间戳通常表示事件实际发生的时间。
2. **事件时间数据流**还需要携带**水位线**，算子从中推断系统的当前事件时间。

时间戳和**水位线**都是通过从1970-01-01 00:00:00以来的**毫秒数**指定。**水位线**告诉**算子**，**不必再等**那些**时间戳小于或等于水位线**的事件。

DataStream API中提供了`TimestampAssigner`接口（**时间戳分配器**），用于在**事件**被**输入到流应用后**从事件中**提取时间戳**。通常，时间戳分配器会在数据源生成之后立即调用。此外，为了确保依赖事件时间的算子能正常工作，必须在**任何依赖事件时间**的算子计算**之前**调用**时间戳分配器**

时间戳分配器的**工作原理**和其他**转换算子 类似**。它们会**作用在事件流**上，并**生成**一个**带有时间戳和水位线**的**新数据流**。时间戳分配器**不会更改**DataStream的**数据类型**。

下面展示自定义时间戳分配器的使用方法，主要利用`assignTimestampsAndWatermarks()`方法

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

val readings: DataStream[SensorReading] = env
    .addSource(new SensorSource)
    // 使用自定义的时间戳分配器来分配时间戳并生成水位线
    .assignTimestampsAndWatermarks(new MyAssigner)
```

自定义时间戳分配器主要分为两种

1. **周期性水位线分配器**：周期性地发出水位线
2. **定点水位线分配器**：根据输入事件中的某个属性或者标记来生成水位线

6.1.1.1 周期性水位线分配器

周期性分配水位线的含义是**系统**以**固定的机器时间间隔**来**发出水位线**并推动事件时钟前进。**默认**的间隔时间设置为**200毫秒。**可以使用 `ExecutionConfig.setAutoWatermarkInterval()` 方法对**间隔时间**进行**配置**:

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

//设置水位线间隔为每隔5秒一次
env.getConfig.setAutoWatermarkInternal(5000)
```

示例6-3展示了一个**周期性水位线分配器**，它通过跟踪到目前为止遇到的事件的最大时间戳来生成水位线。当需要生成新水印时，分配器返回一个时间戳等于**最大时间戳减去1分钟容忍间隔**的**水位线。**

```
/**定义一个周期性水位线时间戳分配器*/
class PeriodicAssigner extends AssignerWithPeriodicWatermarks[SensorReading] {
    val bound: Long = 60 * 1000 // 1分钟的毫秒数，容忍间隔
    val maxTs: Long = Long.MinValue // 观察到的最大时间戳

    /**用来生成水位线的方法*/
    override def getCurrentWatermark: Watermark = {
        new Watermark(maxTs - bound)
    }

    /**用来生成时间戳的方法*/
    override def extractTimestamp(
        r: SensorReading
        previousTS: Long): Long = {
        // 更新最大时间戳
        maxTs = maxTs.max(r.timestamp)
        // 返回记录的时间戳
        r.timestamp
    }
}
```

```
/**调用这个分配器*/
val readings: DataStream[SensorReading] = env
    .addSource(new SensorSource)
    .assignTimestampAndWatermarks(new PeriodicAssigner())
```

DataStream API内置了**两个**针对**常见情况**的周期性水位线时间戳**分配器。**

1 assignAscendingTimestamps

如果您的输入元素的时间戳是单调递增的，那么您可以使用方法assignascendingtimestamp。此方法使用**当前时间戳**来**生成水位线。**但是这种方式没有考虑延迟情况，因此可能比较**激进**

```
val stream: DataStream[SensorReading] = ...
val r = stream.assignAscendingTimestamps(e => e.timestamp)
```

2 BoundedOutOfOrdernessTimestampExtractor

周期性水位线生成的另一种常见情况是，你可以**预测到**在输入流中会遇到的**最大延迟**（任何新到元素的时间戳与所有先前到达的元素的时间戳最大值之间的差异）。对于这种情况，Flink提供了`BoundedOutOfOrdernessTimestampExtractor`，它将**最大预期延迟**作为一个**参数**:

```
val stream: DataStream[SensorReading] = ...
val r = stream.assignTimestampAndWatermarks(
    new BoundedOutOfOrdernessTimestampExtractor[SensorReading]
(Times.seconds(10)){
        override def extractTimestamp(e: SensorReading): Long = e.timestamp
    }
)

"""
我们假设延迟为10秒
并且实现抽象方法extractTimestamp
"""
```

6.1.1.2 定点水位线分配器

有时，输入流包含一些**指示系统进度**的**特殊元组或标记。**Flink为这种情况提供了`assignerwithPunctuatedWatermarks`接口。该接口定义了`checkAndGetNextWatermark()`**方法**，该方法将在**每个事件的**`extractTimestamp()`**之后**被**调用。**该方法可以**决定是否生成新的水位线。**

下面举个例子

```
/**
  * Assigns timestamps to records and emits a watermark for each reading with
sensorId == "sensor_1".
  */
class PunctuatedAssigner extends AssignerWithPunctuatedWatermarks[SensorReading]
{

  // 1 min in ms
  val bound: Long = 60 * 1000

  // 如果该方法返回一个非空、且大于之前值的水位线，算子会将这个新水位线发出
  override def checkAndGetNextWatermark(r: SensorReading, extractedTS: Long):
Watermark = {
    if (r.id == "sensor_1") {
      // emit watermark if reading is from sensor_1
      new Watermark(extractedTS - bound)
    } else {
      // do not emit a watermark
      null
    }
  }
  // 生成时间戳
  override def extractTimestamp(r: SensorReading, previousTS: Long): Long = {
    // assign record timestamp
    r.timestamp
  }
}
```

6.1.2 水位线、延迟及完整性问题

水位线主要用于平衡**延迟**和**结果的完整性**。

- 如果水位线设置的过于**宽松**，会导致**较大的延迟**并且需要**更多的存储空间**来缓存数据；但是得到的**结果相对准确**
- 如果水位线设置的过于**紧迫**，则**相反**
- 但是无论水位线设置得多宽松，**总会出现迟到数据**
- 对于流处理应用，就是要在延迟和完整性之间做一个**取舍**

6.2 处理函数(Process Function)

下面看看如何在**转换算子**中**访问**到**时间戳和水位线**信息。

DataStream API提供一系列相对底层的转换操作——**处理函数**，这些转换的语义很强大，

- 可以**访问**事件的**时间戳**和**水位线**，
- 还可以**注册**在特定时间触发的**计时器**，
- 还可以通过**副输出(side output)**功能，发出记录到多个输出流。

目前，Flink提供了**8种**不同的**处理函数**:

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

下面以KeyedProcessFunction为例

- KeyedProcessFunction是一个**非常灵活**的函数，**作用于KeyedStream**上。
- 对流的**每条记录**调用该函数，会返回零条、一条或多条记录。
- 并且它**实现了RichFunction接口**，因此提供了open()、close()和getRuntimeContext()方法。
- 另外，KeyedProcessFunction[KEY，IN, OUT]还额外提供了以下两种抽象方法:
    1. processElement(v: IN, ctx: Context, out: Collector[out])：**Context**是进程函数的**灵活之处**。它提供对**当前事件**的**时间戳**、**键**、**TimerService**等的**访问**。此外，Context可以将记录发送到副输出。
    2. onTimer(timestamp:Long, ctx: OnTimerContext, out: Collector[out])：它是一个回调函数，当之前注册的**计时器触发时**，它会被调用，来**执行**计时器绑定的**逻辑操作**。

KeyedProcessFunction接口的源代码如下

```
/**
 * A keyed function that processes elements of a stream.
 *
 * @param <K> Type of the key.
 * @param <I> Type of the input elements.
 * @param <O> Type of the output elements.
 */
@PublicEvolving
public abstract class KeyedProcessFunction<K, I, O> extends AbstractRichFunction
{

    private static final long serialVersionUID = 1L;

    public abstract void processElement(
        I value, Context ctx, Collector<O> out) throws Exception;

    public void onTimer(
        long timestamp, OnTimerContext ctx, Collector<O> out) throws Exception
{}

    /**
     * Information available in an invocation of {@link #processElement(Object,
Context, Collector)}
     * or {@link #onTimer(long, OnTimerContext, Collector)}.
     */
    public abstract class Context {
```

```
        public abstract Long timestamp();

        public abstract TimerService timerService();

        public abstract <X> void output(OutputTag<X> outputTag, X value);

        public abstract K getCurrentKey();
    }

    /**
     * Information available in an invocation of {@link #onTimer(long,
OnTimerContext, Collector)}.
     */
    public abstract class OnTimerContext extends Context {

        public abstract TimeDomain timeDomain();

        @Override
        public abstract K getCurrentKey();
    }

}
```

6.2.1 TimerService和Timer

观察上面的源码我们不难发现，`Context`提供一个`timerSercvice()`方法，它会返回一个`TimerService`。这个接口提供了一系列时间相关的操作。具体如下源码

```
public interface TimerService {

    /**返回当前的处理时间*/
    long currentProcessingTime();

    /**返回当前的水位线*/
    long currentWatermark();

    /**注册处理时间计时器*/
    void registerProcessingTimeTimer(long time);

    /**注册事件时间计时器*/
    void registerEventTimeTimer(long time);

    /**移除处理时间计时器*/
    void deleteProcessingTimeTimer(long time);

    /**移除事件时间计时器*/
    void deleteEventTimeTimer(long time);
}
```

在KeyedProcessFunction中，**每个键**的**每个时间戳**只能注册**一个计时器**，这些计时器会按照时间戳顺序放到一个**优先队列**中。

当需要生成检查点时，**计时器也会被写入检查点。**如果应用程序需要**从故障中恢复**，那么在应用程序重新启动时**过期的所有处理时间计时器**将在应用程序恢复时**立即触发。**

下面我们实现一个KeyedProcessFunction。它监测传感器的温度，如果传感器的温度在处理时间语义中单调增加1秒，就会发出警告:

```
object ProcessFunctionTimers {

  def main(args: Array[String]) {

    // set up the streaming execution environment
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // use event time for the application
    env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)

    // ingest sensor stream
    val readings: DataStream[SensorReading] = env
      // SensorSource generates random temperature readings
      .addSource(new SensorSource)

    val warnings = readings
      // key by sensor id
      .keyBy(_.id)
      // 调用KeyedProcessFunction的具体实现TempIncreaseAlertFunction
      .process(new TempIncreaseAlertFunction)

    warnings.print()

    env.execute("Monitor sensor temperatures.")
  }
}

/** Emits a warning if the temperature of a sensor
  * monotonically increases for 1 second (in processing time).
  */
class TempIncreaseAlertFunction
  extends KeyedProcessFunction[String, SensorReading, String] {

  // 存储上一个到来的传感器温度
  lazy val lastTemp: ValueState[Double] =
    getRuntimeContext.getState(
      new ValueStateDescriptor[Double]("lastTemp", Types.of[Double])
    )

  // 存储当前活跃的一个计时器
  lazy val currentTimer: ValueState[Long] =
    getRuntimeContext.getState(
      new ValueStateDescriptor[Long]("timer", Types.of[Long])
    )
  /**处理每个元素*/
```

```
  override def processElement(
      r: SensorReading,
      ctx: KeyedProcessFunction[String, SensorReading, String]#Context,
      out: Collector[String]): Unit = {

    // get previous temperature
    val prevTemp = lastTemp.value()
    // update last temperature
    lastTemp.update(r.temperature)

    val curTimerTimestamp = currentTimer.value()
    // 这个计时器的第一个事件到来了，不做任何处理
    if (prevTemp == 0.0) {
      // first sensor reading for this key.
      // we cannot compare it with a previous value.
    }
    // 如果新的温度小于老的温度，在上下文中注销计时器并且清空保存的计时器状态
    else if (r.temperature < prevTemp) {
      // temperature decreased. Delete current timer.
      ctx.timerService().deleteProcessingTimeTimer(curTimerTimestamp)
      currentTimer.clear()
    }
    // 如果新的温度大于老的温度并且当前没有计时器，就创建一个1s后触发的计时器并且保存到内部状态
里面
    else if (r.temperature > prevTemp && curTimerTimestamp == 0) {
      // temperature increased and we have not set a timer yet.
      // set timer for now + 1 second
      val timerTs = ctx.timerService().currentProcessingTime() + 1000
      ctx.timerService().registerProcessingTimeTimer(timerTs)
      // remember current timer
      currentTimer.update(timerTs)
    }
  }

  /**当计时器触发时会调用这个函数*/
  override def onTimer(
      ts: Long,
      ctx: KeyedProcessFunction[String, SensorReading, String]#OnTimerContext,
      out: Collector[String]): Unit = {
    // 把这个事件的String放到输出中，相当于发出了一条警报
    out.collect("Temperature of sensor '" + ctx.getCurrentKey +
      "' monotonically increased for 1 second.")
    // 清空计时器状态
    currentTimer.clear()
  }
}
```

6.2.2 向副输出发送数据（Emitting to Side Outputs）

副输出(side outputs)是处理函数的一个特性，它可以从同一个函数**发出多条数据流**，且**副输出**的元素**类型**可以与输入**不同。**

下面直接举个例子

```
/**
 * 对于温度低于32F的读数，会向副输出发出警报
 */
object SideOutputs {

  def main(args: Array[String]): Unit = {

    // ingest sensor stream
    val readings: DataStream[SensorReading] = ...

    val monitoredReadings: DataStream[SensorReading] = readings
      // monitor stream for readings with freezing temperatures
      // 调用处理函数
      .process(new FreezingMonitor)

    // retrieve and print the freezing alarms
    monitoredReadings
      .getSideOutput(new OutputTag[String]("freezing-alarms"))
      .print()

    // print the main output
    readings.print()

    env.execute()
  }
}

/** Emits freezing alarms to a side output for readings with a temperature below
32F. */
class FreezingMonitor extends ProcessFunction[SensorReading, SensorReading] {

  // define a side output tag
  // 定义一个副输出标签
  lazy val freezingAlarmOutput: OutputTag[String] =
    new OutputTag[String]("freezing-alarms")

  // 处理每个元素的方法
  override def processElement(
      r: SensorReading,
      ctx: ProcessFunction[SensorReading, SensorReading]#Context,
      out: Collector[SensorReading]): Unit = {
    // 如果SensorReading的温度小于32F则放入Freezing Alarm副输出
    // emit freezing alarm if temperature is below 32F.
    if (r.temperature < 32.0) {
      ctx.output(freezingAlarmOutput, s"Freezing Alarm for ${r.id}")
    }
    // 所有的SensorReading都output到常规输出
    // forward all readings to the regular output
    out.collect(r)
  }
}
```

6.2.3 CoProcessFunction

对于有两个输入的底层操作，DataStream API还提供了CoProcessFunction。与CoFlatMapFunction类似，CoProcessFunction也提供了一对作用在每个输入上的转换方法`processElement1()`和`processElement2()`。

下面直接举个例子

```
object CoProcessFunctionTimers {

  def main(args: Array[String]) {

    // set up the streaming execution environment
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // use event time for the application
    env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)

    // switch messages disable filtering of sensor readings for a specific amount
of time
    val filterSwitches: DataStream[(String, Long)] = env
      .fromCollection(Seq(
        ("sensor_2", 10 * 1000L), // sensor_2 前进2秒
        ("sensor_7", 60 * 1000L)) // sensor_7 前进7秒
      )

    // ingest sensor stream
    val readings: DataStream[SensorReading] = env
      // SensorSource generates random temperature readings
      .addSource(new SensorSource)

    val forwardedReadings = readings
      // 联结读数和开关 connect readings and switches
      .connect(filterSwitches)
      // 设定这两个流分别根据什么来分区 key by sensor ids
      .keyBy(_.id, _._1)
      // 调用下面代码实现的一个CoProcessFunction的接口 apply filtering
CoProcessFunction
      .process(new ReadingFilter)

    forwardedReadings
      .print()

    env.execute("Monitor sensor temperatures.")
  }
}

class ReadingFilter
  extends CoProcessFunction[SensorReading, (String, Long), SensorReading] {

  // 一个Boolean类型的状态 允许转发的开关 switch to enable forwarding
  lazy val forwardingEnabled: ValueState[Boolean] =
    getRuntimeContext.getState(
      new ValueStateDescriptor[Boolean]("filterSwitch", Types.of[Boolean])
```

```
    )

  // 保存一个时钟 hold timestamp of currently active disable timer
  lazy val disableTimer: ValueState[Long] =
    getRuntimeContext.getState(
      new ValueStateDescriptor[Long]("timer", Types.of[Long])
    )

  // 处理第一条流的数据
  override def processElement1(
      reading: SensorReading,
      ctx: CoProcessFunction[SensorReading, (String, Long),
SensorReading]#Context,
      out: Collector[SensorReading]): Unit = {
    // 检查开关是否为true，true的时候才转发，false时直接丢弃应该是
    // check if we may forward the reading
    if (forwardingEnabled.value()) {
      out.collect(reading)
    }
  }

  // 处理第二条流的数据
  override def processElement2(
      switch: (String, Long),
      ctx: CoProcessFunction[SensorReading, (String, Long),
SensorReading]#Context,
      out: Collector[SensorReading]): Unit = {

    // 打开转发开关
    // enable reading forwarding
    forwardingEnabled.update(true)
    // 设置停止转发的计时器
    // set disable forward timer
    val timerTimestamp = ctx.timerService().currentProcessingTime() + switch._2
    val curTimerTimestamp = disableTimer.value()
    // 比较和当前的计时器相比哪个比较大，如果新计时器大就移除老计时器，重新设置为新计时器
    if (timerTimestamp > curTimerTimestamp) {
      // remove current timer and register new timer
      ctx.timerService().deleteProcessingTimeTimer(curTimerTimestamp)
      ctx.timerService().registerProcessingTimeTimer(timerTimestamp)
      disableTimer.update(timerTimestamp)
    }
  }

  // 计时器被触发时，这个方法会被调用
  override def onTimer(
      ts: Long,
      ctx: CoProcessFunction[SensorReading, (String, Long),
SensorReading]#OnTimerContext,
      out: Collector[SensorReading]): Unit = {

    // remove all state. Forward switch will be false by default.
    // 开关会被设置为false，也就是停止转发
    forwardingEnabled.clear()
    disableTimer.clear()
  }
}
```

6.3 窗口算子

窗口是流式应用中的常见操作。它们可以在**无限数据流**的**有界区间**上实现**聚合**等操作。通常，这些间隔是使用基于时间的逻辑定义的。窗口算子提供了一种基于有限大小的桶对事件进行分组的方法，并对这些桶中的有限内容进行计算。

6.3.1 定义窗口算子

窗口算子可以应用于键值分区(keyed)或非键值分区(nonkeyed)的数据流上。**键值分区**上的窗口算子**并行计算**，而**非键值分区**的窗口算子在**单个线程**中处理。

在流上应用窗口算子需要两步：

1. 第一步是调用 `keyBy()` 指定一个**窗口分配器**，它会决定对输入流中的元素如何划分到各个桶中
2. 第二步是调用某一种**窗口函数**来**处理**分配到窗口中的元素

下面的例子展示了分区和不分区的**窗口算子定义方式**

```
stream
    .keyBy(...)      // 分区
    .window(...)     // 指定窗口分配器
    .reduce/aggregate/process(...)  // 指定窗口函数
```

```
stream
    .windowAll(...) //指定窗口分配器，不分区(window-all，全量窗口)
    .reduce/aggregate/process(...)  // 指定窗口函数
```

6.3.2 内置窗口分配器(Built-in Window Assigners)

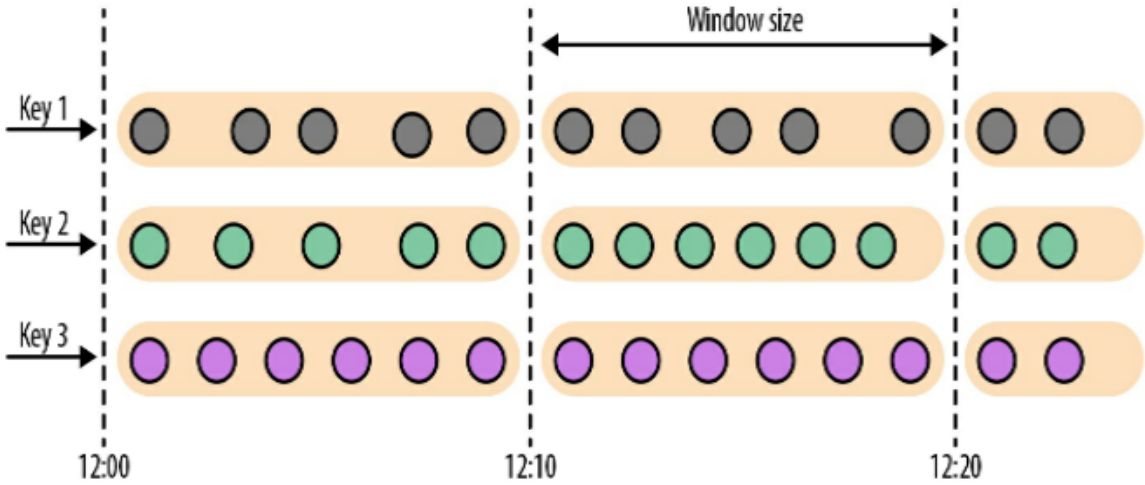
Flink为最常见的窗口使用场景提供了内置的窗口分配器。本节我们**只讨论基于时间**的窗口分配器。基于时间的窗口分配器**根据**元素**事件时间**的时间戳或当前**处理时间**将元素**分配**给窗口。**每个时间窗口**有一个**开始时间戳**和一个**结束时间戳。**

所有**内置的窗口分配器**都提供了一个**默认触发器**，当(处理或事件)时间经过**窗口末尾**时，该触发器将触发对窗口的计算，然后指定的**窗口函数**就会**被调用。**

Flink的内置窗口分配器**所创建的窗口**的**类型**为`TimeWindow`。此**窗口类型**实际上表示两个时间戳之间的**时间区间**（左闭右开）。

6.3.2.1 滚动窗口(Tumbling windows)

滚动窗口分配器将元素放入**不重叠**的**固定大小**的窗口中，如图6-1所示。



Datastream API提供了两个分配器：

- TumblingEventTimeWindows：用于**事件时间**
- TumblingProcessingTimeWindow：用于**处理时间**
- 滚动窗口分配器只接收一个参数：**窗口大小**

例子如下

```
val sensorData: DataStream[SensorReading] = ...

// 按照事件时间来分配窗口
val avgTemp = sensorData
  .keyBy(_.id)
   // 按照事件时间，大小为1s，来划分窗口
  .window(TumblingEventTimeWindows.of(Time.seconds(1)))
  .process(new TemperatureAverager)

// 按照处理时间来分配窗口
val avgTemp = sensorData
  .keyBy(_.id)
   // 按照处理时间，大小为1s，来划分窗口
  .window(TumblingProcessingTimeWindows.of(Time.seconds(1)))
  .process(new TemperatureAverager)

// 利用一个更快捷的方法来分配窗口
val avgTemp = sensorData
  .keyBy(_.id)
   // 大小为1s，来划分窗口,具体按照处理时间还是事件时间要结合当前环境来进行判断
```

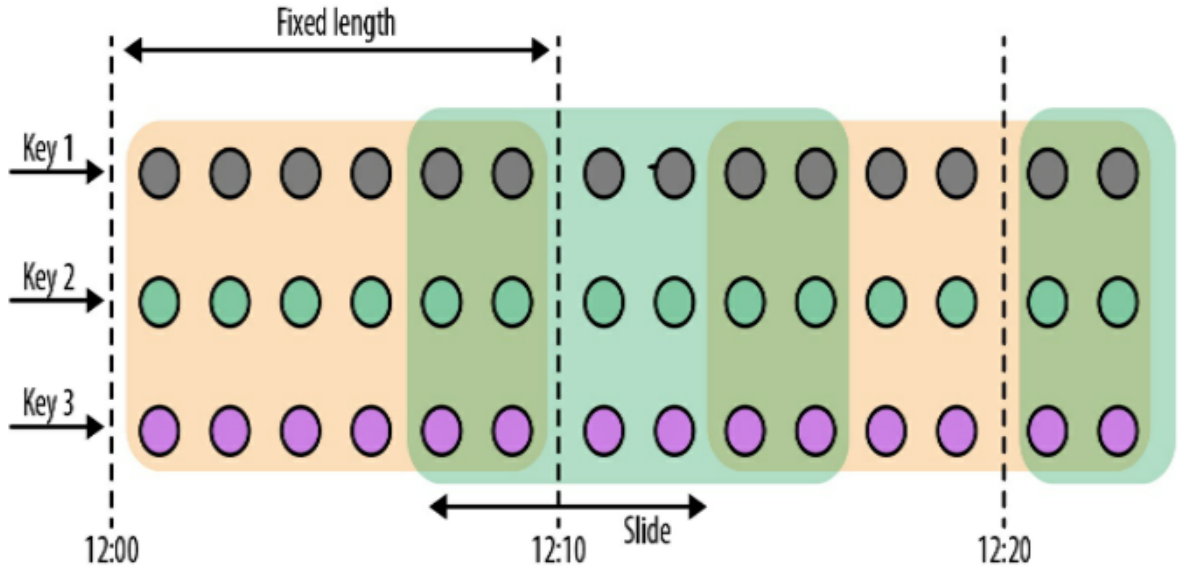
```
  .timeWindow(Time.second(1))
  .process(new TemperatureAverager)
```

默认情况下，滚动窗口与纪元时间1970-01-01-00:00:00.000对齐。例如，大小为1小时的分配器将在00:00:00、01:00:00、02:00:00等位置定义窗口。或者，你可以在分配器中通过第二个参数指定一个**偏移量。**下面的代码显示了偏移值为15分钟的窗口，它从00:15:00、01:15:00、02:15:00等位置定义窗口：

```
val avgTemp = sensorData
  .keyBy(_.id)
   // group readings in 1 hour windows with 15 min offset
  .window(TumblingEventTimeWindows.of(Time.hours(1),
Time.minutes(15)))
  .process(new TemperatureAverager)
```

6.3.2.2 滑动窗口(Sliding windows)

滑动窗口分配器将元素分配给**大小固定**且**按指定滑动间隔移动**的窗口，如图6-2所示



对于滑动窗口，必须指定**窗口大小**和**滑动间隔**，以定义新窗口启动的频率。

- 当**滑动间隔小于窗口大小**时，**窗口会重叠**，元素可以分配给多个窗口。
- 当**滑动间隔大于窗口大小**时，有些元素不会被分配给任何窗口，会被**丢弃。**

下面举个例子

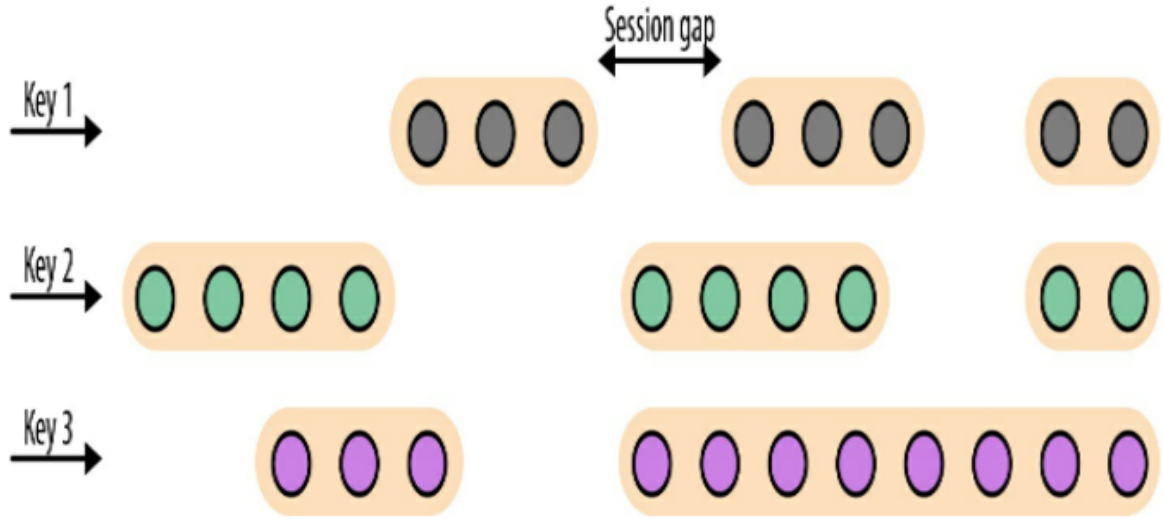
```
// 事件时间
// event-time sliding windows assigner
val slidingAvgTemp = sensorData
```

```
  .keyBy(_.id)
   // create 1h event-time windows every 15 minutes
  .window(SlidingEventTimeWindows.of(Time.hours(1), Time.minutes(15)))
  .process(new TemperatureAverager)

// 处理时间
// processing-time sliding windows assigner
val slidingAvgTemp = sensorData
  .keyBy(_.id)
   // create 1h processing-time windows every 15 minutes
  .window(SlidingProcessingTimeWindows.of(Time.hours(1), Time.minutes(15)))
  .process(new TemperatureAverager)

// 简写
// sliding windows assigner using a shortcut method
val slidingAvgTemp = sensorData
  .keyBy(_.id)
   // shortcut for window.(SlidingEventTimeWindow.of(size, slide))
  .timeWindow(Time.hours(1), Time(minutes(15)))
  .process(new TemperatureAverager)
```

6.3.2.3 会话窗口(Session windows)

会话窗口分配器将元素放入**长度可变**但是**不重叠**的窗口中。**会话窗口的边界**由**不活动时间间隔**(session gap)（没有接收到记录的时间间隔）定义。

图6-3说明了如何将元素分配给会话窗口。



下面举一个例子

```
/** session gap 设置为15分钟*/
// event-time session windows assigner
val sessionWindows = sensorData
  .keyBy(_.id)
```

```
    // create event-time session windows with a 15 min gap
  .window(EventTimeSessionWindows.withGap(Time.minutes(15)))
  .process(...)

// processing-time session windows assigner
val sessionWindows = sensorData
  .keyBy(_.id)
    // create processing-time session windows with a 15 min gap
  .window(ProcessingTimeSessionWindows.withGap(Time.minutes(15)))
  .process(...)
```

由于会话窗口的**开始时间**和**结束时间**都**依赖于输入元素**，所以窗口分配器**不能立即**将所有元素**分配**给**正确**的窗口。

- 因此，会话窗口分配器**最初**将**每个输入元素**映射到自己的**单独窗口**中，**开始时间**为元素的**时间戳**，**窗口大小**为**会话间隔。**
- 然后，**分配器**会将**具有重叠范围**的所有窗口**合并。**

6.3.3 在窗口上应用函数

如6.3.1小节所示，**窗口的计算逻辑**由**窗口函数**负责定义。

可用于窗口的函数类型有两种

1. **增量聚合函数**（Incremental aggregation functions）：
    - 它的应用场景是**窗口**内**以状态形式 存储某个值**并且需要**根据每个加入窗口的元素**对该值进行**更新**
    - 此类函数通常非常**节省空间**且最终会将**聚合值**作为单个结果发出
    - 下文介绍的**ReduceFunction**和**AggregateFunction**都是增量聚合函数
2. **全量窗口函数**（Full window functions）
    - 全量窗口函数**收集**一个**窗口的所有元素**，并在**计算**时**遍历所有元素**来获取计算结果。
    - 全窗口函数通常**需要更多空间**，但比增量聚合函数**支持更复杂的逻辑。**
    - 下文介绍的**ProcessWindowFunction**就是一个全量窗口函数。

6.3.3.1 ReduceFunction

ReduceFunction接受两个相同类型的值，并将它们组合成一个类型相同的值。当在一个Windowed Stream上应用ReduceFunction语义时，ReduceFunction增量地聚合窗口中的元素。窗口只存储聚合的**当前结果**，它是一个**和输入输出类型相同的值。**当接收到新元素时，算子会从窗口读取当前状态并调用ReduceFunction结合新元素来更新状态。

下面举个例子

```
val minTempPerWindow: DataStream[(String, Double)] = sensorData
  .map(r => (r.id, r.temperature))
  .keyBy(_._1)
  .timeWindow(Time.seconds(15))
  // 计算并输出15s窗口中的最小值
  .reduce((r1, r2) => (r1._1, r1._2.min(r2._2)))
```

6.3.3.2 AggregateFunction

AggregateFunction是一个比ReduceFunction**更灵活**的窗口函数，其接口定义如下

```
/**
* IN 输入类型
* ACC 累加器类型(内部状态)
* OUT 输出类型
*/
public interface AggregateFunction<IN, ACC, OUT> extends Function, Serializable
{
  // 创建一个累加器来启动聚合
  ACC createAccumulator();
  // 向累加器中添加一个输入元素并返回累加器
  ACC add(IN value, ACC accumulator);
  // 根据累加器来返回结果
  OUT getResult(ACC accumulator);
  // 合并两个累加器
  ACC merge(ACC a, ACC b);
}
```

与ReduceFunction不同的是，AggregateFunction的**中间数据类型**和**输出类型 不依赖**于**输入类型。**

下面举个例子

```
// 计算每个窗口的传感器读数的平均温度。
val avgTempPerWindow: DataStream[(String, Double)] = sensorData
  .map(r => (r.id, r.temperature))
  // 根据id来分key
  .keyBy(_._1)
  // 15秒的窗口
  .timeWindow(Time.seconds(15))
  .aggregate(new AvgTempFunction)

// An AggregateFunction to compute the average tempeature per sensor.
// The accumulator holds the sum of temperatures and an event count.
class AvgTempFunction extends AggregateFunction
[(String, Double), (String, Double, Int), (String, Double)] {

  // 创建累加器，累加器的
  override def createAccumulator() = {("", 0.0, 0) // (ID, 累加器, 计数器}

  // 加和
  override def add(in: (String, Double), acc: (String, Double, Int)) = {
```

```
    (in._1, in._2 + acc._2, 1 + acc._3)
  }

  // 求平均作为结果返回
  override def getResult(acc: (String, Double, Int)) = {
    (acc._1, acc._2 / acc._3)
  }

  // 合并累加器的方法
  override def merge(acc1: (String, Double, Int), acc2: (String, Double, Int)) =
{
    (acc1._1, acc1._2 + acc2._2, acc1._3 + acc2._3)
  }
}
```

6.3.3.3 ProcessWindowFunction

ProcessWindowFunction是一个Full Window Function，它会将窗口的所有元素**收集**起来**先不做处理**，等**完全收集好**之后，**再处理。**它比增量聚合应用更广，比如计算窗口内数据的中值或者出现频率最高的值等。

```
/**
* IN: 输入类型
* OUT: 输出类型
* KEY: 键的类型
* W: 窗口元数据的类型
*/
public abstract class ProcessWindowFunction<IN, OUT, KEY, W extends Window>
    extends AbstractRichFunction {

  // 对窗口执行计算
  void process(KEY key, Context ctx, Iterable<IN> vals,
               Collector<OUT> out) throws Exception;

  // 当窗口要被删除时，清理一些自定义的状态
  public void clear(Context ctx) throws Exception {}

  // Context窗口的上下文
  public abstract class Context implements Serializable {

    // 返回窗口元数据
    public abstract W window();

    // 返回当前处理时间
    public abstract long currentProcessingTime();

    // 返回当前事件时间戳
    public abstract long currentWatermark();

    // State accessor for per-window state 每个窗口的状态
    public abstract KeyedStateStore windowState();
```

```
    // State accessor for per-key global state 每个键的全局状态
    public abstract KeyedStateStore globalState();

    // Emits a record to the side output identified by the OutputTag.
    // 向OutputTag标识的副输出发送记录
    public abstract <X> void output(OutputTag<X> outputTag, X value);
  }
}
```

`process()`和`clear()`都有一个Context对象作为参数，这个参数功能很强大

- 访问窗口元数据(Window类型)
- 访问当前处理时间和水位线
- 管理每个窗口的状态和每个键的全局状态
  - **窗口状态**只有**当前窗口**才能访问到
  - **全局状态**可用于在**同一键上的多个窗口**之间共享信息。
- 副输出

注意：**使用了窗口状态**这一功能的ProcessWindowFunction**需要实现clear()**方法，来**在窗口被删除之前清理自定义的窗口状态。**

下面举个例子

```
// output the lowest and highest temperature reading every 5 seconds
val minMaxTempPerWindow: DataStream[MinMaxTemp] = sensorData
  // 这里keyBy中key的类型需要与ProcessWindowFunction的KEY类型参数一致
  .keyBy(_.id)
  .timeWindow(Time.seconds(5))
  .process(new HighAndLowTempProcessFunction)

case class MinMaxTemp(id: String, min: Double, max:Double, endTs: Long)

/**
 * A ProcessWindowFunction that computes the lowest and highest
temperature
 * reading per window and emits them together with the
 * end timestamp of the window.
 * [IN, OUT, KEY, W]
 */
class HighAndLowTempProcessFunction extends ProcessWindowFunction
[SensorReading, MinMaxTemp, String, TimeWindow] {

    override def process(key: String, // 键：这里是传感器的ID
                         ctx: Context,// 上下文
                         vals: Iterable[SensorReading],// 桶中的全部元素
                         out: Collector[MinMaxTemp]): Unit = { // 输出

      val temps = vals.map(_.temperature)
      val windowEnd = ctx.window.getEnd
      out.collect(MinMaxTemp(key, temps.min, temps.max, windowEnd))
```

```
  }
}
```

6.3.3.4 增量聚合与ProcessWindowFunction结合使用

很多情况下用于窗口上的**逻辑**都可以**表示为增量聚合**，只不过它还**需要访问窗口的元数据或状态。**可以将增量聚合函数与ProcessWindowFunction结合使用。

- **分配给窗口的元素**将**立即聚合**，
- 当窗口的触发器触发时，**聚合的结果**将被**传递给ProcessWindowFunction**
- process()方法的`Iterable`参数将**只提供单个值**，即增量聚合的结果。

在DataStream API中，这实现上述过程的途径是**将ProcessWindowFunction作为reduce()或aggregate()方法的第二个参数**，如下面的代码所示:

```
input
  .keyBy(...)
  .timeWindow(...)
  .reduce(
    incrAggregator: ReduceFunction[IN],
    function: ProcessWindowFunction[IN, OUT, K, W])
input
  .keyBy(...)
  .timeWindow(...)
  .aggregate(
    incrAggregator: AggregateFunction[IN, ACC, V],
    windowFunction: ProcessWindowFunction[V, OUT, K, W])
```

下面举一个例子

```
case class MinMaxTemp(id: String, min: Double, max:Double, endTs: Long)

val minMaxTempPerWindow2: DataStream[MinMaxTemp] = sensorData
  .map(r => (r.id, r.temperature, r.temperature))
  .keyBy(_._1)
  .timeWindow(Time.seconds(5))
  .reduce(

    // 增量计算最低和最高温度 [IN, ACC, V]
    (r1: (String, Double, Double), r2: (String, Double, Double)) => {
      (r1._1, r1._2.min(r2._2), r1._3.max(r2._3))
    },

    // 在ProcessWindowFunction中计算最终结果
    new AssignWindowEndProcessFunction()
  )

// [V, OUT, K, W]
class AssignWindowEndProcessFunction extends ProcessWindowFunction
[(String, Double, Double), MinMaxTemp, String, TimeWindow] {
```

```
  override def process(
      key: String,
      ctx: Context,
      minMaxIt: Iterable[(String, Double, Double)],
      out: Collector[MinMaxTemp]): Unit = {

    // 取下序列中的唯一一个元素
    val minMax = minMaxIt.head
    // 从上下文中获取窗口的结束时间
    val windowEnd = ctx.window.getEnd
    // 输出
    out.collect(MinMaxTemp(key, minMax._2, minMax._3, windowEnd))

  }
}
```

6.3.4 自定义窗口算子

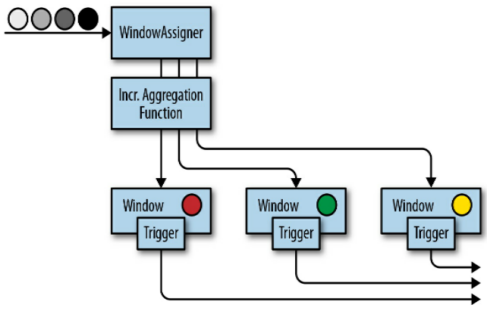
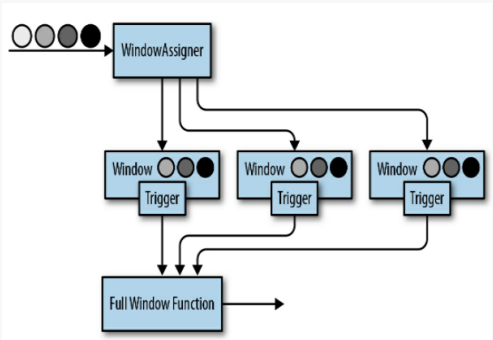
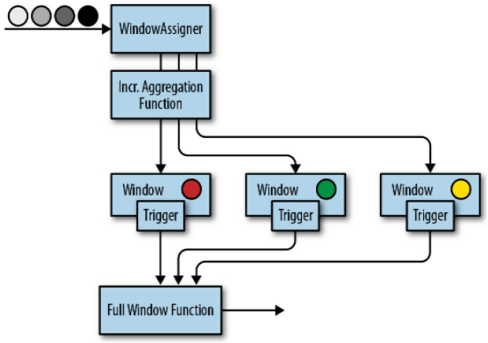
基于Flink的**内置窗口分配器 定义的窗口算子**可以**应对许多常见情况。**但是如果你需要更复杂的逻辑，也可以自定义窗口算子。DataStream API**对外暴露了自定义窗口算子**的接口和方法。你可以实现自己的**分配器(assigner)**、**触发器(trigger)**和**移除器(evictor)**，再加上前面一节提到的**窗口函数**，就可以**组合出一个自定义窗口算子。**

当一个**元素到达**一个窗口算子时，它将**被传递给窗口分配器。**该**分配器**会**决定 元素需要被放置在哪个窗口。**如果这个窗口还不存在，就会直接创建。

如果窗口算子**配置了增量聚合函数**，则会**立即聚合**新添加的元素，并**将结果存储为窗口的状态。**如果窗口算子**没有配置增量聚合函数**，则将新元素**追加到**一个用来存储所有窗口分配元素的**ListState**上。

每当一个元素被添加到一个窗口时，它也被传递到该窗口的**触发器。**触发器定义**何时执行窗口计算**、**何时清除窗口及其状态。**

触发器成功触发后会调用窗口函数，根据窗口函数的不同，触发器的行为具体来说分以下三种

说明	图例
如果算子**只配置了增量聚合函数**，则调用`getResult()`发出当前聚合结果。	
如果算子只配置了`ProcessWindowFunction`（**全量窗口函数**），那么该函数的`process()`将被调用并发出结果	
如果算子**既配置了增量聚合函数，又配置全量窗口函数**，则对聚合函数的**聚合值 应用全量窗口函数**并发出结果。（具体见6.3.3.4小节）	

移除器是一个**可选组件**，可以在调用ProcessWindowFunction**之前**或**之后**注入。移除器可以**从窗口中删除**所收集的**某些元素**。因为它必须遍历所有元素，所以只能在没有指定增量聚合函数的情况下使用它（指定了增量聚合函数的话，此时已经增量聚合结束了，再删除元素会导致计算结果不准确）。

下面的代码展示了如何**组合**前文介绍的各种组件，来**生成自定义算子**

```
stream
  .keyBy(...)
  .window(...)                    // specify the window assigner 指定分配器
 [.trigger(...)]                  // optional: specify the trigger 指定触发器
 [.evictor(...)]                  // optional: specify the evictor 指定移除器
  .reduce/aggregate/process(...) // specify the window function 指定窗口函数
```

此外，当没有显式指定`trigger`时，Flink会提供一个默认的`trigger`，它会在时间戳移动到**窗口右边界**时**触发**

6.3.4.1 窗口的生命周期

在本节中，我们将讨论窗口的生命周期——何时创建，由哪些信息组成，何时删除。

6.3.4.1.1 何时创建

当**窗口分配器**需要**向窗口分配第一个元素**时，就会**创建**一个窗口。因此，一个窗口至少包含一个元素。

6.3.4.1.2 由哪些信息组成

一个窗口由以下不同的状态组成:

状态	描述
窗口内容	如果窗口算子配置了ReduceFunction或AggregateFunction，则窗口内容包含**增量聚合的结果。**如果窗口算子配置了ProcessFunction，则窗口内容包含**分配给窗口的元素**
窗口对象	窗口分配器返回零个、一个或多个窗口对象。窗口算子根据返回的对象对元素进行分组。因此**窗口对象中保存**用于**区分窗口的信息。**每个窗口对象都有一个**结束时间戳**，它定义了可以删除窗口及其状态的时间点。
触发器定时器	可以在触发器中注册计时器，以便在特定的时间点回调
触发器中的自定义状态	触发器可以定义和使用针对每个窗口、每个键的**自定义状态。**这种状态完全**由触发器控制**，而不是由窗口算子维护。

6.3.4.1.3 何时删除

窗口算子会在**窗口结束时间**(由窗口对象的结束时间戳定义)**删除窗口。**

删除一个窗口时，**窗口算子**会**自动清除 窗口内容**并丢弃**窗口对象**，但**不会清除 自定义的触发器状态**和**触发器计时器。**因此，**触发器**必须**实现** `trigger.clear()` **方法**来**清理**这些，以**防止状态泄漏。**

6.3.4.2 窗口分配器

WindowAssigner用于决定将到达的元素分配给哪些窗口。

下面首先看看`WindowAssigner`接口的源码

```
// T: 流中的元素类型
// W: 窗口元数据类型
public abstract class WindowAssigner<T, W extends Window> implements
Serializable {

  // Returns a collection of windows to which the element is assigned
  // 返回元素分配的目标窗口集合
  public abstract Collection<W> assignWindows(T element,
                                              long timestamp,
                                              WindowAssignerContext context);

  // 返回窗口分配器的默认触发器(用于算子没有显式指定触发器的情况)
  public abstract Trigger<T, W> getDefaultTrigger(StreamExecutionEnvironment
env);

  // Returns the TypeSerializer for the windows of this WindowAssigner
  public abstract TypeSerializer<W> getWindowSerializer(ExecutionConfig
executionConfig);

  // 判断这个窗口分配器使用的是不是事件时间
  public abstract boolean isEventTime();

  // 窗口分配器的上下文
  public abstract static class WindowAssignerContext {

    // 返回当前处理时间
    public abstract long getCurrentProcessingTime();
  }
}
```

下面展示如何**自定义一个窗口分配器**

```
/** A custom window that groups events into 30-second tumbling windows. */
class ThirtySecondsWindows extends WindowAssigner[Object, TimeWindow]
  // 窗口尺寸
  val windowSize: Long = 30 * 1000L

  // 分配窗口
  override def assignWindows(o: Object,ts: Long,
    ctx: WindowAssigner.WindowAssignerContext):java.util.List[TimeWindow] = {

    // 计算所属窗口的开始时间和结束时间
    // rounding down by 30 seconds
    val startTime = ts - (ts % windowSize)
    val endTime = startTime + windowSize
    // 返回一个列表，列表中的每个元素都是当前事件所属的窗口
    // emitting the corresponding time window
    Collections.singletonList(new TimeWindow(startTime, endTime))
  }

  // 获取默认触发器
  override def getDefaultTrigger(env: environment.StreamExecutionEnvironment)
    : Trigger[Object, TimeWindow] = {
```

```
      // 直接返回一个事件时间触发器
      EventTimeTrigger.create()
  }

  // 窗口序列化器
  override def getWindowSerializer(executionConfig: ExecutionConfig)
    :TypeSerializer[TimeWindow] = {
    new TimeWindow.Serializer
  }
  // 使用的是处理时间
  override def isEventTime = true
}
```

6.3.4.3 触发器

触发器定义了**何时**进行**窗口计算**并**发出结果。**触发器可以根据**时间**或**特定的数据条件**触发。例如对于基于时间窗口的默认触发器来说，当处理时间或水位线超过窗口结束边界的时间戳时，默认触发器将触发。

触发器的功能很强大，它可以访问时间属性和计时器，并且可以使用状态。

每次**调用触发器**时，它都会**生成一个TriggerResult**来决定窗口应该发生什么。TriggerResult可以取以下值之一:

TriggerResult	描述
CONTINUE	什么都不做
FIRE	如果窗口算子配置了**ProcessWindowFunction**，则调用该函数**计算并发出**结果。如果窗口只配置了**增量聚合函数**，则会**发出**当前聚合结果。
PURGE	**窗口内容**将被完全**丢弃**，**窗口**将被**删除。**此外，**ProcessWindowFunction.clear()**方法被调用以清除所有自定义的每个窗口状态。
FIRE_AND_PURG	首先**计算**窗口(触发)，然后**删除**所有状态和元数据(清除)。

下面展示一下触发器的接口源码

```
public abstract class Trigger<T, W extends Window> implements Serializable {

  // 每当有元素被添加到窗口时，这个方法都会被调用
  // Called for every element that gets added to a window
  TriggerResult onElement(T element, long timestamp, W window, TriggerContext
ctx);

  // 当一个处理时间计时器触发时调用
  // Called when a processing-time timer fires
  public abstract TriggerResult onProcessingTime(long timestamp,
```

```
                                                W window, TriggerContext ctx);
  // 当一个事件时间计时器触发时调用
  // Called when an event-time timer fires
  public abstract TriggerResult onEventTime(long timestamp,
                                            W window, TriggerContext ctx);

  // 返回触发器是否支持状态的合并
  // Returns true if this trigger supports merging of trigger state
  public boolean canMerge();

  // 当多个窗口需要合并时调用，多个窗口中触发器的状态也要被合并
  // Called when several windows have been merged into one window
  // and the state of the triggers needs to be merged
  public void onMerge(W window, OnMergeContext ctx);

  // 这个方法会在一个窗口被清除时调用，它应该清除触发器自定义的各种
  // Clears any state that the trigger might hold for the given window
  // This method is called when a window is purged
  public abstract void clear(W window, TriggerContext ctx);

  // 用于触发器中方法的上下文对象
  // A context object that is given to Trigger methods to allow them
  // to register timer callbacks and deal with state
  public interface TriggerContext {
      // 获取当前处理时间
      // Returns the current processing time
      long getCurrentProcessingTime();
      // 获取当前水位线
      // Returns the current watermark time
      long getCurrentWatermark();
      // 注册处理时间计时器
      // Registers a processing-time timer
      void registerProcessingTimeTimer(long time);
      // 注册事件时间计时器
      // Registers an event-time timer
      void registerEventTimeTimer(long time);
      // 删除处理时间计时器
      // Deletes a processing-time timer
      void deleteProcessingTimeTimer(long time);
      // 删除事件时间计时器
      // Deletes an event-time timer
      void deleteEventTimeTimer(long time);
      // 获取一个作用域为触发器键值和当前窗口的状态对象
      // Retrieves a state object that is scoped to the window and the key of
the trigger
      <S extends State> S getPartitionedState(StateDescriptor<S, ?>
stateDescriptor);
  }

  // 用于onMerge方法的特殊上下文
  // Extension of TriggerContext that is given to the Trigger.onMerge() method
  public interface OnMergeContext extends TriggerContext {
      // Merges per-window state of the trigger
      // The state to be merged must support merging
      void mergePartitionedState(StateDescriptor<S, ?> stateDescriptor);
  }

}
```

但是有两点要特别注意：**状态清理**和**合并触发器**

1. 状态清理：
    - 在**触发器中**使用**单窗口状态**时，需要确保在删除窗口时正确删除该状态。否则，算子将**随着时间积累越来越多的状态。**
    - 为了在删除窗口时清除所有状态，**触发器的clear()方法**需要**删除所有自定义的单窗口状态**，并使用TriggerContext对象**删除所有计时器。**
2. 合并触发器
    - 在处理合并的时候，一定要注意**合并 触发器的自定义状态**(onMerge())

下面我们举一个自定义触发器的例子

```
/** 一个会每隔1s提前触发一次计算的触发器 */
class OneSecondIntervalTrigger extends Trigger[SensorReading, TimeWindow] {

  // 处理每个事件
  override def onElement(r: SensorReading, timestamp: Long,
      window: TimeWindow, ctx: Trigger.TriggerContext): TriggerResult = {

    // firstSeen是一个Boolean类型的状态，初始值为false
    // firstSeen will be false if not set yet
    val firstSeen: ValueState[Boolean] = ctx.getPartitionedState(
      new ValueStateDescriptor[Boolean]("firstSeen", classOf[Boolean]))

    // 当第一个事件到达时，注册两个计时器
    // register initial timer only for first element
    if (!firstSeen.value()) {
      // compute time for next early firing by rounding watermark to second
      val t = ctx.getCurrentWatermark + (1000 - (ctx.getCurrentWatermark %
1000))
      // 注册第一个计时器，当前水位线+1s
      ctx.registerEventTimeTimer(t)
      // 注册第二个计时器，当前窗口的结束时间
      // register timer for the window end
      ctx.registerEventTimeTimer(window.getEnd)
      // 更新firstSeen状态为true
      firstSeen.update(true)
    }
    // 返回Continue，意思是什么都不做
    // Continue. Do not evaluate per element
    TriggerResult.CONTINUE
  }

  // 当事件时间计时器触发时，这个方法被调用
  override def onEventTime(
      timestamp: Long,
      window: TimeWindow,
      ctx: Trigger.TriggerContext): TriggerResult = {
    // 如果是窗口的结束时间的那个计时器
    if (timestamp == window.getEnd) {
      // 执行计算并且清除窗口
      // final evaluation and purge window state
```

```
        TriggerResult.FIRE_AND_PURGE
    // 如果+1s的计时器
    } else {
        // register next early firing timer
        // 先注册下一个计时器，还是+1s
        val t = ctx.getCurrentWatermark + (1000 - (ctx.getCurrentWatermark %
1000))
        if (t < window.getEnd) {
          ctx.registerEventTimeTimer(t)
        }
        // 执行计算
        // fire trigger to evaluate window
        TriggerResult.FIRE
      }
    }

    // 当处理时间计时器触发时，这个方法被调用，没啥用
    override def onProcessingTime(
        timestamp: Long,
        window: TimeWindow,
        ctx: Trigger.TriggerContext): TriggerResult = {
      // Continue. We don't use processing time timers
      TriggerResult.CONTINUE
    }

    // 当窗口要被删除时，这个方法被调用，我们需要手动清理firstSeen状态
    override def clear(window: TimeWindow, ctx: Trigger.TriggerContext): Unit = {

      // clear trigger state
      val firstSeen: ValueState[Boolean] = ctx.getPartitionedState(
        new ValueStateDescriptor[Boolean]("firstSeen", classOf[Boolean]))

      firstSeen.clear()
    }
}
```

6.3.4.4 移除器

在Flink的窗口机制中，**移除器**是一个**可选组件**。它可以在**窗口函数**计算**之前**或**之后** **删除窗口中的元素。**

下面示例展示了`Evictor`接口的源码

```
public interface Evictor<T, W extends Window> extends Serializable {

  // Optionally evicts elements. Called before windowing function.
  void evictBefore(Iterable<TimestampedValue<T>> elements, int size,
                   W window, EvictorContext evictorContext);

  // Optionally evicts elements. Called after windowing function.
  void evictAfter(Iterable<TimestampedValue<T>> elements, int size,
    W window, EvictorContext evictorContext);
```

```
  // A context object that is given to Evictor methods.
  interface EvictorContext {
    // Returns the current processing time.
    long getCurrentProcessingTime();
    // Returns the current event time watermark.
    long getCurrentWatermark();
  }
}
```

6.4 Joining Streams on Time

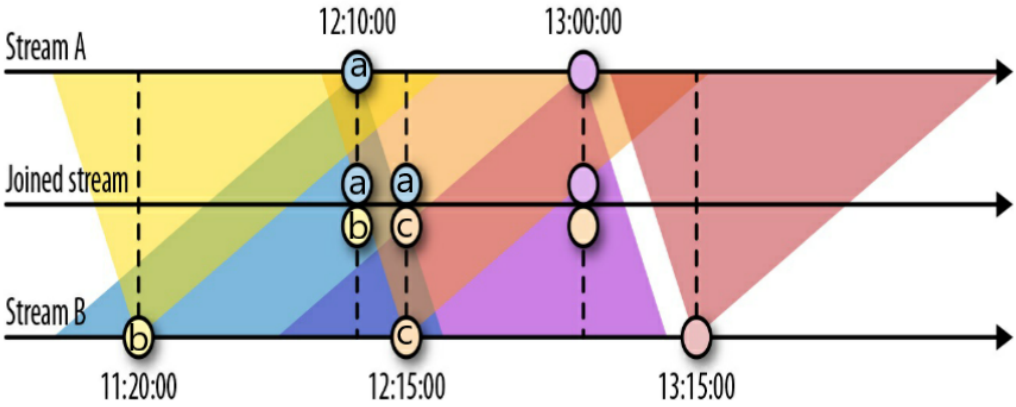
在处理流时，一个常见的需求是**connect** or **join** the events of two streams。Flink的DataStream API提供了两个内置的算子：**Interval join** 和 **Window join**。在本节中，我们将描述这两个算子。

6.4.1 Interval Join

Interval Join对于两个流中拥有**相同的键**，并且彼此之间的**时间戳间隔 不超过指定的间隔**的事件进行join操作。

下图显示的两条流A和B。B中某个事件会与A中的一些事件join成对，**具体步骤**如下

- 以B中某个事件为**基事件**
- **从A中选择**那些相较于基事件，**时间戳间隔**在$[-1h, +15min]$范围内的事件
- 将这些事件**join成事件对**，来**一起处理**。如下图的**a事件**和**b事件**就会组成**事件对**，**a事件**和**c事件**也会组成**事件对**



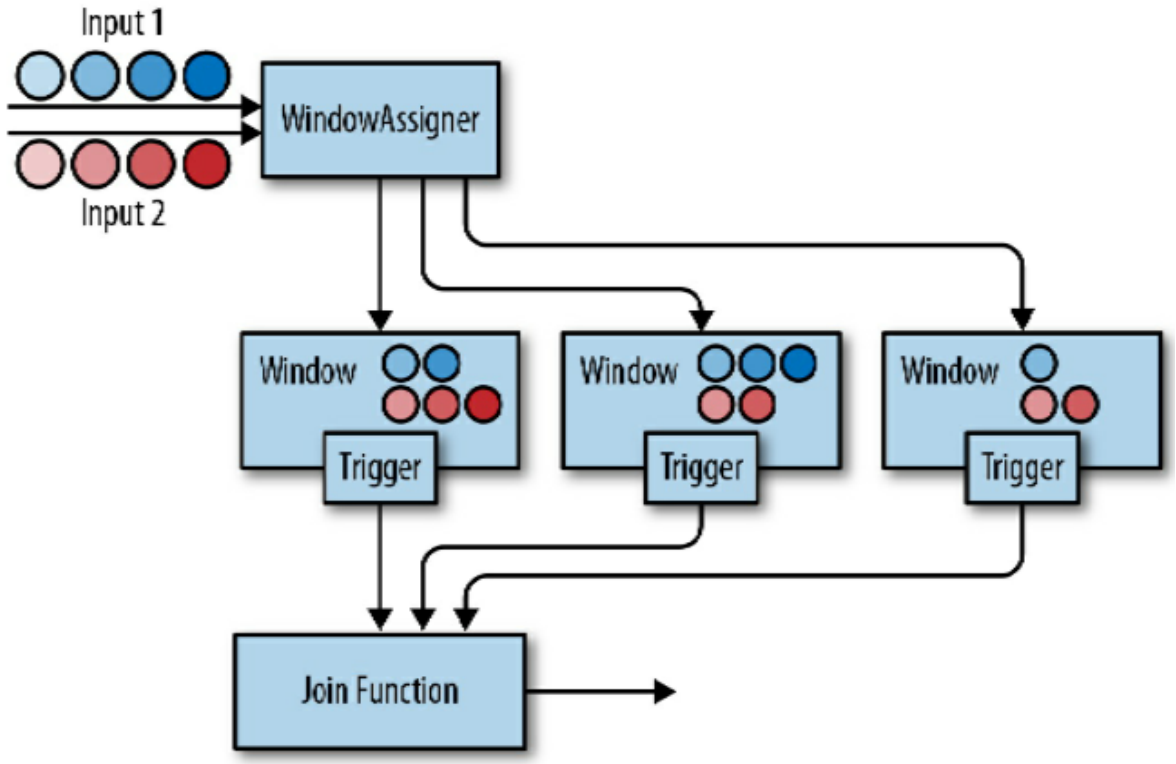
Interval Join的API使用方法如下

```
input1
  .keyBy(…) // 按键分区
  .between(Time.hour(-1), Time.minute(15)) // 指定事件的上下界
  .process(ProcessJoinFunction) // JOIN成功的事件对会被发送给这个函数，由它来处理
```

6.4.2 Window Join

顾名思义，Window Join是**基于**Flink的**窗口机制**的。两个输入流的元素都被分配到**同一个公共窗口**，并在这个窗口收集完成时进行叉乘积联接，然后交给ProcessJoinFunction计算。

工作流程如下图所示



上图的解释如下

- 两个输入流都**根据**它们各自的**键**属性进行**分区**，
- **公共窗口分配器**将两个流的事件映射到**公共窗口**，这意味着**公共窗口同时存储着来自两条输入流的事件。**
- 当**窗口的触发器触发**时，将对两个输入流中的每个元素**组合**（叉乘积）**调用JoinFunction。**
- 此外还可以**自定义 触发器**和**移除器。**由于这两个流的事件被映射到相同的窗口中，因此触发器和移除器的行为与常规窗口算子中的触发器和移除器行为完全相同。

下面来看看我们怎么使用这个API

```
input1.join(input2)
  .where(...)       // specify key attributes for input1 指定第一条流的key状态
  .equalTo(...)     // specify key attributes for input2 指定第二条流的key状态
  .window(...)      // specify the WindowAssigner 指定窗口分配器
 [.trigger(...)]    // optional: specify a Trigger 指定触发器(可以不指定)
 [.evictor(...)]    // optional: specify an Evictor 指定移除器(可以不指定)
  .apply(...)       // specify the JoinFunction 指定处理函数
```

除了Join之外，还可以使用cogroup()。Join和**CoGroup**的总体逻辑是相似的，Join对来自两个输入的每一对事件调用JoinFunction，而CoGroup对窗口中两条输入序列的遍历器调用GroupFunction。（也就是说参数不同）

6.5 处理迟到数据

迟到事件是在**算子需要执行的计算已经完成时 到达算子**的事件。在**事件时间窗口算子**这种情况下，如果事件到达算子，但是窗口分配器将其分配到了一个**已经完成了计算的窗口**（也就是算子的水位线超过了窗口的结束时间的窗口），则该事件就是迟到的。

DataStream API提供了三种处理迟到事件的方案:

- Dropping: 简单地**丢弃**迟到事件。
- Redirect: 将迟到事件**重定向到单独的流**中。
- Update: 根据迟到事件**更新计算结果**，并且发出结果。

下面三小节分别介绍这三种情况

6.5.1 丢弃迟到事件(Dropping)

处理迟到事件最简单的方法就是丢弃。这也是事件时间窗口的**默认行为**。因此，迟到的元素将不会创建新窗口。

6.5.2 重定向迟到事件(Redirect)

迟到事件还可以使用副输出特性重定向到另一个DataStream，这样就可以根据业务需求来进行各种不同种类的后期处理

下面举个例子，说明如何**将迟到事件重定向到副输出**

```
// 处理正常流
val filteredReadings: DataStream[SensorReading] = readings
    .process(new LateReadingsFilter)

// 取出副输出
val lateReadings: DataStream[SensorReading] = filteredReadings
    .getSideOutput(lateReadingsOutput)

// 对正常流进行后续处理 print the filtered stream
filteredReadings.print()
```

```
// 对副输出进行后续处理 print messages for late readings
lateReadings
    .map(r => "*** late reading *** " + r.id)
    .print()

/** A ProcessFunction that filters out late sensor readings and re-directs them
to a side output */
class LateReadingsFilter extends ProcessFunction[SensorReading, SensorReading] {

  override def processElement(
      r: SensorReading,
      ctx: ProcessFunction[SensorReading, SensorReading]#Context,
      out: Collector[SensorReading]): Unit = {

    // compare record timestamp with current watermark
    if (r.timestamp < ctx.timerService().currentWatermark()) {
      // this is a late reading => redirect it to the side output
      // 迟到的事件重定向到副输出中
      ctx.output(LateDataHandling.lateReadingsOutput, r)
    } else {
      // 正常的事件直接输出
      out.collect(r)
    }
  }
}
```

6.5.3 基于迟到事件更新结果(Update)

另一种策略是重新计算结果并发出更新。但是，为了能够重新计算和更新结果，需要考虑一些问题。

- 支持Update策略的算子需要在第一次**结果发出后 保存**计算所需的所有**状态。**
- **下游算子**或**外部系统 能够处理**得了这些更新。

窗口算子API提供了一个方法来显式声明你希望能够处理迟到事件。在使用事件时间窗口时，可以指定一个额外的时间段，称为**延迟容忍度**(allowed lateness)。配置了该属性的窗口不会被立刻删除，而是会被保存到延迟容忍度再删除。

下面举个例子，来看看**延迟容忍度**怎么使用

```
val readings: DataStream[SensorReading] = ???
val countPer10Secs: DataStream[(String, Long, Int, String)] = readings
    .keyBy(_.id)
    .timeWindow(Time.seconds(10))
    // process late readings for 5 additional seconds 设置延迟容忍度为5s
    .allowedLateness(Time.seconds(5))
    // count readings and update results if late readings arrive
    .process(new UpdatingWindowCountFunction)

/**这个处理函数会采用Update策略处理迟到事件*/
class UpdatingWindowCountFunction extends ProcessWindowFunction[SensorReading,
    (String, Long, Int, String), String, TimeWindow] {
```

```
  override def process(
      id: String,
      ctx: Context,
      elements: Iterable[SensorReading],
      out: Collector[(String, Long, Int, String)]): Unit = {
    // count the number of readings
    val cnt = elements.count(_ => true)
    // 这个状态用来标记是否是首次计算
    val isUpdate = ctx.windowState.getState(
      new ValueStateDescriptor[Boolean]("isUpdate", Types.of[Boolean]))
    if (!isUpdate.value()) {
      // 首次计算并发出结果
      out.collect((id, ctx.window.getEnd, cnt, "first"))
      isUpdate.update(true)
    } else {
      // 不是首次计算，发出更新
      out.collect((id, ctx.window.getEnd, cnt, "update"))
    }
  }
}
```

第7章 有状态算子和应用

大多数复杂一点的算子都需要存储部分数据。Flink的许多**内置**的数据流**算子**、**数据源**和**数据汇**都是**有状态的**。例如：

- 窗口算子为ProcessWindowFunction**收集输入事件**或为ReduceFunction**保存聚合结果**
- **ProcessFunction**需要**保存计时器**
- 一些**数据汇函数**需要**维护事务**的状态

除了内置的算子、数据源、数据汇之外，Flink的DataStream API还提供了一些接口，用于在**用户自定义函数(user-defined function)**中**注册**、**维护**和**访问 状态**。

本章重点介绍

1. **如何**在用户定义的函数中**定义不同类型的状态**并与之交互。
2. 我们还将讨论**性能**方面以及**如何控制函数状态的大小**。
3. 最后，我们将展示如何将键状态配置为**可查询状态**，以及如何从外部应用程序访问它。

7.1 实现有状态函数

函数有两种类型的状态：**键状态**(keyed state)和**算子状态**(operator state)。在本节中，我们将一一介绍如何实现具有键状态的函数和具有算子状态的函数。

7.1.1 键状态

用户自定义函数可以在键属性的上下文中**存储**和**访问 对应的键状态**。Flink会为**键域中**的**每个键**都维护**一个状态实例**，这些实例会分布在算子的那些并行任务上。也就是说函数所在算子的**每个并行任务**都负责**键域的一个子域**，并维护**子域上每个键 对应**的那个**状态实例**。因此，键状态非常类似于**分布式键值映射**(distributed key-value map)。

键状态只能**应用在KeyedStream上**。

Flink为**键状态**提供了多个**状态原语**。状态原语定义了在**单个键上状态实例**的**存储结构**。Flink支持以下状态原语:

状态原语	描述
ValueState[T]	**保存**类型为T的**单个值**。可以通过`ValueState.value()`来**获取**，通过`ValueState.update(value: T)`来**更新**
ListState[T]	保存类型为T的元素**列表**。支持add、addAll、get等操作。但是**不支持对单个元素的删除**，但可以通过`update`方法来用新列表替换原来的列表
MapState[K, V]	保存一个键和值的**映射**。该原语提供了许多Java Map接口的常规方法
ReducingState[T]	**用于聚合操作**，与ListState[T]的API大致相似。但调用它的add()会立即使用ReduceFunction聚合值。并且它的get()方法只会返回一个**单值列表**：这个**值是聚合结果**
AggregatingState[I, O]	**用于聚合操作**，比ReducingState更通用一点。调用它的add()会立即使用AggregateFunction聚合值

下面举个例子。如果传感器测量的温度自上次测量以来发生了超过阈值的变化，示例应用程序将发出警报事件。

```
object KeyedStateFunction {

  /** main() defines and executes the DataStream program */
  def main(args: Array[String]) {

    // 省略不重要代码
    val env = ...

    // 省略不重要代码
    val sensorData: DataStream[SensorReading] = ...

    // 先分key
    val keyedSensorData: KeyedStream[SensorReading, String] =
sensorData.keyBy(_.id)

    // 对keyedStream调用flatMap
```

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

    // print result stream to standard out
    alerts.print()

    // execute application
    env.execute("Generate Temperature Alerts")
  }
}

/**
  * The function emits an alert if the temperature measurement of a sensor
changed by more than
  * a configured threshold compared to the last reading.
  *
  * @param threshold The threshold to raise an alert.
  */
class TemperatureAlertFunction(val threshold: Double)
    extends RichFlatMapFunction[SensorReading, (String, Double, Double)] {

  // 用来存储 上一次温度 的状态
  // the state handle object
  private var lastTempState: ValueState[Double] = _

  // 初始化工作
  override def open(parameters: Configuration): Unit = {

    // create state descriptor 创建一个状态描述符
    val lastTempDescriptor = new ValueStateDescriptor[Double](
        "lastTemp", classOf[Double])

    // obtain the state handle 初始化并获取 上一次温度 状态的引用
    lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)
  }

  // 真正的处理函数
  override def flatMap(reading: SensorReading,
    out: Collector[(String, Double, Double)]): Unit = {

    // fetch the last temperature from state 从状态中拿到上一次的温度
    val lastTemp = lastTempState.value()
    // check if we need to emit an alert 计算差值
    val tempDiff = (reading.temperature - lastTemp).abs
    // 如果差值超过阈值，就会被输出
    if (tempDiff > threshold) {
      // temperature changed by more than the threshold
      out.collect((reading.id, reading.temperature, tempDiff))
    }

    // update lastTemp state 更新状态
    this.lastTempState.update(reading.temperature)
  }
}
```

下面对例子中的几个重点进行解释

- 要**创建**一个**状态对象**，我们**必须**通过RichFunction的RuntimeContext向Flink**注册**一个**状态描述符。**
  - **每种状态原语**都有自己**特定的状态描述符**，描述符中包括**状态的名称**和**状态的数据类型。**例如，`val lastTempDescriptor = new ValueStateDescriptor[Double]("lastTemp", classOf[Double])`中，状态原语ValueState有自己的状态描述符类`ValueStateDescriptor`，我们通过提供状态的名称`lastTemp`和状态的类型`Double`来实例化状态描述符类。
  - **ReducingState**和**AggregatingState**的**描述符**在实例化时还需要**额外提供ReduceFunction或AggregateFunction**来对加入该状态的值进行聚合。
- 通过注册多个状态描述符，让处理函数拥有多个状态对象
- 因为Flink需要创建合适的**序列化器**，所以描述符中**必须指定数据类型。**（如前面类型指定的`classOf[Double]`）
- 一般把**状态引用**声明为类中的**成员变量。**然后，**状态引用**会在**open()方法**中被**初始化。**例如，前面例子的`lastTempState`这个状态被声明为了类的成员变量，并且在`open()`方法中`lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)`，它被初始化了。

当一个函数注册一个状态描述符时，Flink会**检查状态后端**中**是否已经**有该状态描述符对应的状态。在**从故障恢复算子**或者**从保存点启动应用**时，可能会**发生这种检查。**在这两种情况下，Flink都将**新注册**的状态引用**链接到**状态后端中**描述符相同**的**现存状态。**如果**找不到**，就将状态初始化为**空。**

此外，FlinkAPI还提供了一种更简洁的写法，我们用这种简便写法实现与上例逻辑相同的功能。

```
val alerts: DataStream[(String, Double, Double)] = keyedData
  .flatMapWithState[(String, Double, Double), Double] {
    // 处理第一个事件到达的情况（这时没法对比阈值差距）
    // 第一个参数是当前事件
    // 第二个参数是当前状态(Flink会从后端中取出状态并填充到这里)
    // 第一个返回值是flatMap的结果列表
    // 第二个返回值是处理后的状态(Flink会用这个值来对后端中的状态进行更新)
    case (in: SensorReading, None) =>{
      // no previous temperature defined; just update the last temperature
      (List.empty, Some(in.temperature))
    }

    // 处理非第一个事件到达的情况（正常的处理，这里才能比较阈值）
    case (r: SensorReading, lastTemp: Some[Double]) => {
      // compare temperature difference with threshold
      val tempDiff = (r.temperature - lastTemp.get).abs
      if (tempDiff > 1.7) {
        // threshold exceeded; emit an alert and update the last temperature
        (List((r.id, r.temperature, tempDiff)), Some(r.temperature))
      } else {
        // threshold not exceeded; just update the last temperature
        (List.empty, Some(r.temperature))
      }
    }
}
```

7.1.2 算子状态

每个算子的并行任务管理**算子状态**。在算子的同一并行任务中处理的所有事件都可以访问相同的状态。Flink支持三种算子状态：列表状态、联合列表状态以及广播状态。

处理函数可以通过**实现** `ListCheckpointed` **接口**来**处理算子列表状态**。处理函数会将**算子状态**当作**常规的成员变量**来实现，然后**通过**ListCheckpoint接口中的两个**回调函数**与**状态后端 交互**:

下面来看看ListCheckpointed接口的源码

```
public interface ListCheckpointed<T extends Serializable> {

    // 以列表的形式返回一个函数状态的快照
    List<T> snapshotState(long checkpointId, long timestamp) throws Exception;

    // 根绝提供的列表来恢复函数状态
    void restoreState(List<T> state) throws Exception;
}
```

下面我们举个例子，它显示了如何为一个函数实现ListCheckpoint接口，该函数为算子的每个并行任务统计温度超过阈值的数目

```
class HighTempCounter(val threshold: Double)
    extends RichFlatMapFunction[SensorReading, (Int, Long)]
    with ListCheckpointed[java.lang.Long] {
  // 获取当前任务的索引标志
  private lazy val subtaskIdx = getRuntimeContext.getIndexOfThisSubtask
  // 本地计数器（当前任务的本地状态）
  private var highTempCnt = 0L
  // 处理函数，每当温度超过阈值就对本地计数器+1
  override def flatMap(
      in: SensorReading,
      out: Collector[(Int, Long)]): Unit = {
    if (in.temperature > threshold) {
      // increment counter if threshold is exceeded
      highTempCnt += 1
      // emit update with subtask index and counter
      out.collect((subtaskIdx, highTempCnt))
    }
  }
 /**
 * 用来返回快照，这个函数会在生成检查点时被调用
 * @Param chkpntId: 检查点编号
 * @Param ts: JobManager创建检测点时的时间戳
 * @Return 将本地状态转换为一个列表来返回
 */
  override def snapshotState(
      chkpntId: Long,
      ts: Long): java.util.List[java.lang.Long] = {
    // snapshot state as list with a single count
```

```
    java.util.Collections.singletonList(highTempCnt)
  }
 /**
 * 用于恢复本地状态，这个函数在初始化函数状态时被调用
 * @Param state: 将这个列表转换为本地状态
 */
  override def restoreState(
      state: util.List[java.lang.Long]): Unit = {
    highTempCnt = 0
    // restore state by adding all longs of the list
    for (cnt <- state.asScala) {
      highTempCnt += cnt
    }
  }
}
```

7.1.2.1 为什么要把算子状态当作列表来处理呢?

看完上个示例，您可能想知道为什么算子状态作为**状态对象列表(a list of state objects)**处理。这是因为**List结构 支持改变**带算子状态的函数的**并行性。**为了增加或减少具有算子状态的函数的并行性，需要将算子状态重新分配给更多或更少的任务实例。这需要**分割**或**合并 状态**对象，而一般来说，**列表**要比单个值**更加适合 分割**和**合并**

通过提供状态对象列表，具有算子状态的函数可以使用`snapshotState()`和`restoreState()`方法实现此逻辑。

- `snapshotState()`方法将本地算子状态**分解**为多个部分
- `restoreState()`方法将本地算子状态从多个部分**组合**起来
- 当一个算子状态被恢复时，该状态的各个部分被分配到算子的所有**并行任务**中，并调用restoreState()方法。
- 如果**并行任务**比**状态对象 多**，那么**有些任务启动时分配不到状态**，那么传入restoreState()方法时，**入参**就是个**空列表。**

那么，我们前面那个例子只是`java.util.Collections.singletonList(highTempCnt)`，这可能在增加并行度时导致有的任务获取不到状态，所以我们改造一下这个方法，如下代码

```
// 分割操作符列表状态，以便在重新缩放期间更好地分布
override def snapshotState(
    chkpntId: Long,
    ts: Long): java.util.List[java.lang.Long] = {

  // split count into ten partial counts
  val div = highTempCnt / 10
  val mod = (highTempCnt % 10).toInt
  // 把这个本地计数器拆分成10份，返回一个长度为10的列表
  // return count as ten parts
  (List.fill(mod)(new java.lang.Long(div + 1))
    ++ List.fill(10 - mod)(new java.lang.Long(div))).asJava
}
```

7.1.3 使用连接广播状态(Using Connected Broadcast State)

流应用中的一个常见需求是将**相同的信息 分发**到一个算子的**所有并行任务**中，并将其作为可恢复状态进行维护。

> 例如，一个规则流和一个依赖于这个规则的事件流。函数 connect 这两个输入流。它需要使用规则来处理所有的输入事件。因此我们需要对规则流进行广播，来让所有处理事件流的任务都接收到规则，都用这个规则来处理输入事件。

下面还是直接看个例子，看看如何实现用**广播流**来**动态配置阈值**的温度报警系统

```
object BroadcastStateFunction {

  /** main() defines and executes the DataStream program */
  def main(args: Array[String]) {

    // set up the streaming execution environment
    val env = ...

    // ingest sensor stream
    val sensorData: DataStream[SensorReading] =

    // define a stream of thresholds
    // 定义一个阈值的流（规则流，需要广播，这样可以动态配置规则）
    val thresholds: DataStream[ThresholdUpdate] = env.fromElements(
      ThresholdUpdate("sensor_1", 5.0d),
      ThresholdUpdate("sensor_2", 0.9d),
      ThresholdUpdate("sensor_3", 0.5d),
      ThresholdUpdate("sensor_1", 1.2d),  // update threshold for sensor_1
      ThresholdUpdate("sensor_3", 0.0d))  // disable threshold for sensor_3

    // 对事件流进行分key
    val keyedSensorData: KeyedStream[SensorReading, String] =
sensorData.keyBy(_.id)

    // 创建一个广播状态描述符（一个键状态描述符）
    // 名字是"thresholds"
    // 键的类型是String
    // 值的类型是Double
    val broadcastStateDescriptor = new MapStateDescriptor[String, Double](
        "thresholds",
        classOf[String],
        classOf[Double]
    )

    // 使用一个或多个广播状态描述符来创建一个广播流
    val broadcastThresholds: BroadcastStream[ThresholdUpdate] = thresholds
      .broadcast(broadcastStateDescriptor)

    // 将正常流与广播流联结，并且使用处理函数处理
    val alerts: DataStream[(String, Double, Double)] = keyedSensorData
```

```
      .connect(broadcastThresholds)
      .process(new UpdatableTemperatureAlertFunction())

    // print result stream to standard out
    alerts.print()

    // execute application
    env.execute("Generate Temperature Alerts")
    }
}

case class ThresholdUpdate(id: String, threshold: Double)

/**
  * 处理函数需要实现KeyedBroadcastProcessFunction接口，它有四个类型参数
  * The key type of the input keyed stream.
  * 事件流中元素的类型(The input type of the keyed (non-broadcast) side.)
  * 广播流中元素的类型(The input type of the broadcast side.)
  * 输出的类型
  */
class UpdatableTemperatureAlertFunction()
    extends KeyedBroadcastProcessFunction[String,
                                          SensorReading,
                                          ThresholdUpdate,
                                          (String, Double, Double)] {

  // 阈值状态描述符
  private lazy val thresholdStateDescriptor
        = new MapStateDescriptor[String, Double]
        ("thresholds", classOf[String], classOf[Double])

  // ValueState[Double]状态 用来储存上一个温度的
  private var lastTempState: ValueState[Double] = _

  // 初始化函数
  override def open(parameters: Configuration): Unit = {
    // 创建上一个温度状态的描述符
    val lastTempDescriptor
      = new ValueStateDescriptor[Double]("lastTemp", classOf[Double])
    // 根据描述符来初始化上一个温度状态
    lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)
  }

  // 处理广播流的事件
  override def processBroadcastElement(
      update: ThresholdUpdate,
      ctx: KeyedBroadcastProcessFunction[
          String,
          SensorReading,
          ThresholdUpdate,
          (String, Double, Double)]#Context,
      out: Collector[(String, Double, Double)]): Unit = {

    // 从上下文获取广播状态
    val thresholds = ctx.getBroadcastState(thresholdStateDescriptor)

    // 如果传入状态的阈值不等于0.0d，更新广播状态
    if (update.threshold != 0.0d) {
```

```
        thresholds.put(update.id, update.threshold)
      } else {
        // 如果传入的阈值为0.0d，就说明用户不想设置阈值了，就直接删掉广播状态中对应键
        thresholds.remove(update.id)
      }
    }

    // 处理正常流的事件
    override def processElement(
        reading: SensorReading,
        readOnlyCtx: KeyedBroadcastProcessFunction[
            String,
            SensorReading,
            ThresholdUpdate,
            (String, Double, Double)]#ReadOnlyContext,
        out: Collector[(String, Double, Double)]): Unit = {

      // 获得只读的广播状态
      val thresholds = readOnlyCtx.getBroadcastState(thresholdStateDescriptor)
      // 获取key对应的阈值，然后比较是否超过阈值，超过就报警
      if (thresholds.contains(reading.id)) {
        // get threshold for sensor
        val sensorThreshold: Double = thresholds.get(reading.id)

        // fetch the last temperature from state
        val lastTemp = lastTempState.value()
        // check if we need to emit an alert
        val tempDiff = (reading.temperature - lastTemp).abs
        if (tempDiff > sensorThreshold) {
          // temperature increased by more than the threshold
          out.collect((reading.id, reading.temperature, tempDiff))
        }
      }

      // 更新上一个温度
      this.lastTempState.update(reading.temperature)
    }
}
```

对于上面的示例，有几点要说明一下

- KeyedBroadcastProcessFunction的元素处理方法是**不对称**的。方法processElement()和processBroadcastElement()参数中的上下文**一个支持读写**，**一个只可读。**

7.1.4 使用CheckpointedFunction接口

CheckpointedFunction接口是**指定有状态函数**的**最底层接口。**它提供了**钩子方法**来**注册**和**维护 键状态**和**算子状态**，并且是唯一允许访问算子列表联合状态(在恢复或保存点重新启动时完全复制的操作符状态)的接口。

CheckpointedFunction接口定义了两个方法，initializeState()和snapshotState()。

- **initializeState()**方法在**创建**CheckpointedFunction的**并行任务时**被调用。当启动应用程序或由于故障而重新启动任务时，就会创建并行任务从而调用这个方法。该方法用于**初始化**状态或者**恢复**状态
- 在**生成检查点之前**，**snapshotState()**方法被调用，snapshotState()方法的目的是确保**在检查点生成完成之前 更新**所有**状态对象**（这样检查点拿到的本地任务状态就是新的）。

下面还是举个例子，这个例子**同时具有键状态和算子状态**，它统计每个键和每个算子任务有多少传感器读数超过了指定的阈值。

```
class HighTempCounter(val threshold: Double)
  extends FlatMapFunction[SensorReading, (String, Long, Long)]
    // 需要实现CheckpointedFunction接口
    with CheckpointedFunction {

  // 本地变量，用来统计当前任务超过阈值的温度个数
  var opHighTempCnt: Long = 0

  // 键状态，用来储存当前key超过阈值的温度个数
  var keyedCntState: ValueState[Long] = _

  // 算子状态，用来储存当前任务超过阈值的温度个数
  var opCntState: ListState[Long] = _

  // 处理函数
  override def flatMap(
      v: SensorReading,
      out: Collector[(String, Long, Long)]): Unit = {
    if (v.temperature > threshold) {
      // 本地变量++
      opHighTempCnt += 1
      // 更新键状态
      val keyHighTempCnt = keyedCntState.value() + 1
      keyedCntState.update(keyHighTempCnt)

      // emit new counters 输出
      out.collect((v.id, keyHighTempCnt, opHighTempCnt))
    }
  }

  // 初始化
  override def initializeState(initContext: FunctionInitializationContext): Unit
= {
    // initialize keyed state 初始化键状态
    val keyCntDescriptor = new ValueStateDescriptor[Long]("keyedCnt",
classOf[Long])
    keyedCntState = initContext.getKeyedStateStore.getState(keyCntDescriptor)

    // initialize operator state 初始化算子状态
    val opCntDescriptor = new ListStateDescriptor[Long]("opCnt", classOf[Long])
    opCntState = initContext.getOperatorStateStore.getListState(opCntDescriptor)
    // initialize local variable with state 用算子状态来设置本地变量的值
    opHighTempCnt = opCntState.get().asScala.sum
  }

  // 快照（用于检查点前的更新）
```

```
  override def snapshotState(snapshotContext: FunctionSnapshotContext): Unit = {
    // update operator state with local state 把本地变量储存的值更新到算子状态
    opCntState.clear()
    opCntState.add(opHighTempCnt)
  }
}
```

7.1.5 接收检查点完成的通知

由于检查点机制，Flink可以实现非常好的性能。然而，另一个含义是，**除了生成检查点时的几个逻辑时间点，应用永远不会处于一致的状态。对于一些算子**来说，**知道检查点是否完成**是很**重要**的。（例如，旨在将数据写入具有严格一致性要求的外部系统的数据汇必须只发出在检查点完成之前接收到的记录，以确保在出现故障的情况下不会重新发送数据。）

只有所有算子任务的状态都成功地写入检查点存储时，检查点**才算成功。**因此，**只有JobManager**才**能确定**检查点是否成功。

需要接收检查点完成的通知的算子可以**实现CheckpointListener接口。**这个接口提供了notifyCheckpointComplete(long chkpntId)方法，当JobManager确定一个检查点成功完成后，该方法会被调用。

7.2 为有状态的应用开启故障恢复

Flink通过**检查点机制**来启动故障恢复，我们**需要显式地启动**检查点机制，如下所示

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

// set checkpointing interval to 10 seconds (10000 milliseconds)
env.enableCheckpointing(10000L)
```

检查点间隔(如前文的10s)会**影响**检查点机制在**常规处理期间的开销**以及**从故障中恢复所需的时间。**较短的检查点间隔在常规处理期间会导致较高的开销，但可以实现更快的恢复，因为需要重新处理的数据更少。

Flink提供了其他一些可供调节的配置选项，比如

- **一致性保障**(精确一次或至少一次)的选择
- **并发检查点的数量**
- 用来取消长时间运行检查点的**超时时间**
- 和**状态后端相关**的选项。

7.3 确保有状态应用的可维护性

已经运行了几周的应用的**状态**可能**代价很高**，甚至无法重新计算。同时，长时间运行的应用程序也需要一些维护，比如修复bug、添加、调整或删除功能，或者调整算子的并行性等。因此，能够将状态**迁移到新版本**应用或**重分配**到更多或更少的算子任务是很重要的。

Flink提供**保存点**来**维护**应用**状态**。但是，它要求**初始版本的应用**的全部有状态**算子**都**指定两个参数**，以确保将来可以正确地维护应用程序。

1. 算子的**唯一标识符**
2. **最大并行度**(只针对基于键的算子)。

算子的**唯一标识符**和**最大并行度**被**固化**到保存点中，不能更改。如果标识符或最大并行度被更改，则**不能**从以前的保存点**重启**应用。

7.3.1 指定算子唯一标识

应该为应用的每个算子指定唯一标识符。这个标识符是保存点中的元数据。当从保存点启动应用程序时，标识符用于将保存点中的状态映射到已启动应用的相应算子。只有当已启动应用程序的算子的标识符相同时，才能将保存点状态恢复到它们。

设置方式如下，强烈建议手动设置

```
val alerts: DataStream[(String, Double, Double)] = keyedSensorData
  .flatMap(new TemperatureAlertFunction(1.1))
  // uid方法，用来设置并行度
  .uid("TempAlert")
```

7.3.2 为使用键状态的算子定义最大并行度

算子的**最大并行度**参数定义了算子在对键状态进行分割时，所用到的**键值组数目**。该数量限制了键状态的**算子**可以被扩展到的**最大并行任务数**。（因为**一个并行任务 至少**要有**一个键值组**）

下面展示如何设置最大并行度

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 为应用设置最大并行度
env.setMaxParallelism(512)

// 为算子设置最大并行度
val alerts: DataStream[(String, Double, Double)] = keyedSensorData
  .flatMap(new TemperatureAlertFunction(1.1))
  // set the maximum parallelism for this operator and
  // override the application-wide value
  .setMaxParallelism(1024)
```

如果**算子**在首个版本**没有设置最大并行度**，则会根据首个版本的并行度来**推断**它

- 如果并行度小于等于128，则最大并行度为128。
- 如果该操作符的并行度大于128，则最大并行度计算为`Min(nextPowerOfTwo(parallelism + (parallelism / 2)),2^15)`。

7.4 有状态应用的性能及鲁棒性

算子与状态的交互会影响应用程序的**健壮性**(robustness)和**性能**(perfermance)。比如选择状态后端、检查点算法的配置、状态的大小都会影响健壮性和性能。

7.4.1 选择状态后端

状态后端负责**存储 每个任务**的**本地状态**，并在**执行检查点时**将其持久化到**远程存储**。由于本地状态可以以不同的方式进行维护和检查点，状态后端是**可插拔的(pluggable)。不同的应用**可以**选择 不同的状态后端**实现来维护它们的状态。状态后端的选择对有状态应用程序的健壮性和性能有影响。**每种状态后端**都**提供了状态原语的实现**，比如ValueState、ListState和MapState。

Flink提供了三种状态后端

- MemoryStateBackend
- FsStateBackend
- RocksDBStateBackend

7.4.1.1 MemoryStateBackend(内存式的，易丢但快)

MemoryStateBackend将状态作为**常规对象**存储在TaskManager **JVM进程的堆**上。

- 例如，MapState由Java HashMap对象支持。
- 虽然这种方法提供了非常低的读写延迟，但它对应用程序的健壮性有影响。

    - 如果任务实例的状态变得太大，**JVM**和在其上运行的所有任务实例可能会由于**OutOfMemoryError**而**被杀死。**
    - 此外，这种方法可能会出现**垃圾回收暂停**(garbage collection pause)问题，因为它将许多长期存在的对象放在堆内存上。
- 当**生成检查点**时，MemoryStateBackend将状态发送给JobManager，**JobManager**将其**存储在堆内存中。**因此，应用程序的总状态必须符合JobManager的内存。因为它的内存是易失性的，所以在JobManager失败的情况下会丢失状态。
- 由于这些限制，MemoryStateBackend仅推荐用于**开发**和**调试**目的。

7.4.1.2 FsStateBackend(本地状态在内存，检查点会持久化)

- FsStateBackend在TaskManager的**JVM**堆上**存储本地状态**，就像MemoryStateBackend一样。
- 然而，FsStateBackend方式下，**检查点**会被写入**远程持久文件系统。**
- FsStateBackend提供了**内存读写速度级别**的**本地访问**和**持久化级别**的**故障容错。**
- 但是，它受到TaskManager内存大小的限制，可能会出现垃圾收集暂停。

7.4.1.3 RocksDBStateBackend（本地持久化，检查点也持久化）

- RocksDBStateBackend将所有**本地状态**存储到**本地RocksDB实例中。**
- RocksDB是一个嵌入式键值存储，它**将数据持久化到本地磁盘。**为了从RocksDB读写数据，需要进行序列化和反序列化。在**生成检查点**时，RocksDBStateBackend还将状态发送到**远程持久文件系统。**
- 所以对于具有非常大状态的应用程序，RocksDBStateBackend是一个不错的选择。

下面展示如何为应用配置状态后端

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val checkpointPath: String = ???
// new 一个RocksDB状态后端实例
val backend = new RocksDBStateBackend(checkpointPath)
// configure the state backend 配置它
env.setStateBackend(backend)
```

7.4.2 选择状态原语

因为RocksDB等后端的状态**读写**涉及**序列化**和**反序列化**，所以**状态原语的选择**很**影响算子性能。**例如：

- ValueState在被访问时是完全反序列化，在被更新时会完全序列化。
- RocksDBStateBackend的ListState在**读取值**之前对**所有列表条目进行反序列化。**但是，向ListState**添加单个值**是一种**廉价的操作**，因为只有追加的值需要序列化，而整个列表不需要完全序列化。
- MapState的RocksDBStateBackend允许读写在键的级别上提供序列化控制。（可以只序列化或者反序列化某个键及其对应的值）。

举例子，

- 针对RocksDBStateBackend来说，使用`MapState[X,Y]`要比`ValueState[HashMap[X,Y]]`更高效。
- 如果经常在列表尾部添加元素，但很少访问列表，那么`ListState[X]`要比`ValueState[List[X]]`更高效。

此外，建议对于同一个状态来说，每次函数调用只更新一次状态。

7.4.3 防止状态泄露

流应用程序通常设计为连续运行数月或数年。如果**应用状态 不断增加**，在**某个时刻**它会变得太大并**杀死应用程序**，除非采取措施为应用提供更多的资源。为了**防止应用的资源消耗随着时间而增加**，**控制算子状态的大小**非常重要。由于状态的处理直接影响算子语义，所以**Flink不能自动清除状态**并释放存储空间。相反，**所有有状态算子都必须承担控制其状态的大小的责任，确保它不会无限增长。**

状态无限增长的一个常见**原因**是**键域无限增长。**

> 例如，在点击事件流，点击事件有一个session_id属性，该属性在一段时间后失效。在这种情况下，具有键状态的函数将积累越来越多键的状态。**随着键域的增大，状态不断增大，但是过期键的状态是陈旧而无用**。这个问题的解决方案是**删除过期键。**
>
> 并且这种情况也会发生在部分DataStreamAPI的**内置算子**上，比如：针对KeyedStream的那些内置聚合函数，`min`、`max`、`sum`等等。所以，在使用这些内置算子时，一定要**保证键域不是无限增加**的

我们可以通过注册计时器，并声明回调函数的方式来清理过期的键

下面举个例子。它会清除在一小时内没有提供任何新的温度测量值的键。

```
object StatefulProcessFunction {

  /** main() defines and executes the DataStream program */
  def main(args: Array[String]) {

    // set up the streaming execution environment
    val env = ...

    // ingest sensor stream
    val sensorData: DataStream[SensorReading] = ...

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

    val alerts: DataStream[(String, Double, Double)] = keyedSensorData
      .process(new SelfCleaningTemperatureAlertFunction(1.5))
```

```
    // print result stream to standard out
    alerts.print()

    // execute application
    env.execute("Generate Temperature Alerts")
  }
}


class SelfCleaningTemperatureAlertFunction(val threshold: Double)
    extends KeyedProcessFunction[String, SensorReading, (String, Double,
Double)] {

  // 状态，用来保存上一个温度
  private var lastTempState: ValueState[Double] = _

  // 状态，用来保存上一个计时器的时间点
  private var lastTimerState: ValueState[Long] = _

  override def open(parameters: Configuration): Unit = {
    // 注册并初始化上一个温度状态
    val lastTempDescriptor = new ValueStateDescriptor[Double]("lastTemp",
        classOf[Double])

    lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)

    // 注册并初始化上一个计时器状态
    val timestampDescriptor: ValueStateDescriptor[Long]
      = new ValueStateDescriptor[Long](
          "timestampState", classOf[Long])

    lastTimerState = getRuntimeContext.getState(timestampDescriptor)
  }

  // 处理函数
  override def processElement(
      reading: SensorReading,
      ctx: KeyedProcessFunction[String, SensorReading, (String, Double,
Double)]#Context,
      out: Collector[(String, Double, Double)]): Unit = {

    // 计算新计时器的触发时间
    val newTimer = ctx.timestamp() + (3600 * 1000)
    // 获取当前计时器
    val curTimer = lastTimerState.value()
    // 删除当前计时器然后注册新计时器
    ctx.timerService().deleteEventTimeTimer(curTimer)
    ctx.timerService().registerEventTimeTimer(newTimer)
    // 更新计时器触发时间到lastTimerState状态上
    lastTimerState.update(newTimer)

    // 获取上一个温度，比较然后决定是否发出警报
    val lastTemp = lastTempState.value()
    val tempDiff = (reading.temperature - lastTemp).abs
    if (tempDiff > threshold) {
      out.collect((reading.id, reading.temperature, tempDiff))
    }
```

```
    // 更新上一个温度
    this.lastTempState.update(reading.temperature)
  }

  // 计时器到时间时，这个函数被触发
  override def onTimer(
      timestamp: Long,
      ctx: KeyedProcessFunction[String, SensorReading, (String, Double,
Double)]#OnTimerContext,
      out: Collector[(String, Double, Double)]): Unit = {

    // 清除一个小时都没有收到事件的键对应的状态
    lastTempState.clear()
    lastTimerState.clear()
  }
}
```

7.5 更新有状态应用

对长时间运行的**有状态流应用**进行**错误修复**或**业务逻辑更改**常常发生。因此，我们要保证在流应用发生**版本更新**时，**不丢失应用状态**

Flink通过三个步骤来实现版本更新

1. 为正在运行的应用**生成保存点**
2. **停止老版本**应用
3. **从保存点启动新版本**的应用。

但是只有新老版本的**保存点兼容**时，才能完成更新。也就是说，Flink只支持下面三种更新（只有这三种更新的保存点是兼容的）

- 在**不更改**或删除现有**状态**的情况下**更改**应用的**逻辑。**这包括向应用中添加算子。
- 从应用中**删除某个状态。**
- 通过**更改状态原语类型或状态的数据类型**来修改现有算子的状态（只有部分情况可以兼容）

7.5.1 保持现有状态更新应用

如果应用**在没有删除或更改现有状态的情况下**进行了更新，那么它始终是与保存点**兼容**的，并且可以从老版本的保存点启动。

如果向应用**添加**新的有状态**算子**，或向现有**算子**添加新的**状态**，则在从保存点启动应用时，**该状态将初始化为空。**（新添加的算子或状态，启动时初始化为空）

7.5.2 从应用中删除状态

还可以通过**删除状态**来调整应用程序。可以**删除完整的算子**或仅从算子中**删除一个状态**。这种情况下，保存点的部分状态将无法映射到新版本应用。

默认情况下，Flink将**不会启动 没有恢复**保存点中包含的**所有状态**的应用程序，以避免丢失保存点中的状态。但是，可以禁用此安全检查

7.5.3 修改算子的状态

增删状态比较容易兼容，但是修改状态很难做到兼容。

一般来说，会发生两种情况的修改

- **更改状态的数据类型**，例如将ValueState[Int]更改为ValueState[Double]，
- **更改状态原语的类型**，例如，将ValueState[List[String]]更改为ListState[String]

对于这两种情况，Flink会进行如下处理

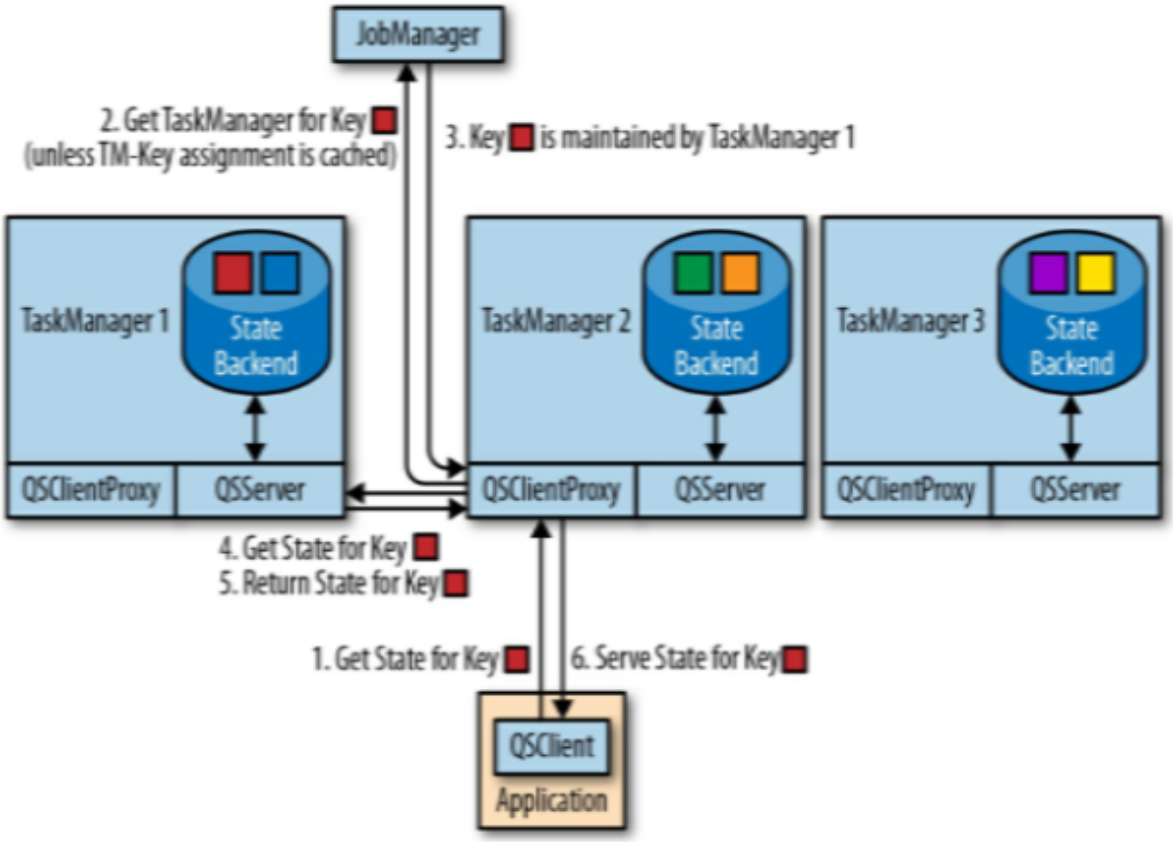
- Flink目前还**不支持更改状态原语的类型**
- 由于涉及到序列化和反序列化机制。对于更改状态的数据类型：在Flink 1.7中，如果数据类型被定义为Apache Avro类型，并且新数据类型也是根据Avro的模式演化规则从原始类型演变而来的Avro类型，那么支持更改状态的数据类型。

7.6 可查询式状态(queryable state)

许多流处理应用需要与其他应用共享它的结果。Apache Flink提供了**可查询状态**的特性来**与其他应用共享结果**。在Flink中，任何键状态都可以作为可查询状态以只读的形式暴露给外部应用程序。

7.6.1 可查询式状态服务的架构及启动方式

Flink的可查询状态服务由三个组件组成:



- **QueryableStateClient**：**外部应用 使用**QueryableStateClient来提交查询和获取结果。
- **QueryableStateClientProxy**：QueryableStateClientProxy**接受并响应QSClient的请求。**每个TaskManager上都运行一个客户端代理。因为状态分布在各个TaskManager，因此ClientProxy先询问JobManager来得知需要查询的状态在哪个TaskManager上，然后向这个TaskManager的QSServer发送请求。
- **QueryableStateServer**：QueryableStateServer响应客户端代理的请求。每个TaskManager都运行一个StateServer，该Server**从本地状态后端获取键状态**，并将其返回给QSClientProxy。

7.6.2 对外暴露可查询式状态

实现一个**带有可查询式状态**的**流应用**很容易。你要做的就是定义一个带有键状态的函数，并在获得状态引用之前，调用setQueryable(String)方法使状态变成可查询的。如下例所示

```
override def open(parameters: Configuration): Unit = {

   // 创建状态描述符
   val lastTempDescriptor = new ValueStateDescriptor[Double]("lastTemp",
classOf[Double])

   // 启动可查询状态，并设置查询标识符
   lastTempDescriptor.setQueryable("lastTemperature")

   // 注册并初始化状态
   lastTempState = getRuntimeContext.getState[Double](lastTempDescriptor)
}
```

除此之外，Flink还**支持利用数据汇**将流中的**所有事件**都**存到可查询状态**中

```
val tenSecsMaxTemps: DataStream[(String, Double)] = sensorData
  // project to sensor id and temperature
  .map(r => (r.id, r.temperature))
  // compute every 10 seconds the max temperature per sensor
  .keyBy(_._1)
  .timeWindow(Time.seconds(10))
  .max(1)

// store max temperature of the last 10 secs for each sensor
// in a queryable state
tenSecsMaxTemps
  // key by sensor id
  .keyBy(_._1)
  .asQueryableState("maxTemperature")
```

7.6.3 从外部系统查询状态

任何基于jvm的应用都可以**使用QueryableStateClient 查询正在运行的Flink应用**的**可查询状态。**

下面举个例子

```
object TemperatureDashboard {
  // assume local setup and TM runs on same machine as client
  val proxyHost = "127.0.0.1"
  val proxyPort = 9069
  // jobId of running QueryableStateJob
  // can be looked up in logs of running job or the web UI
  val jobId = "d2447b1a5e0d952c372064c886d2220a"
  // how many sensors to query
  val numSensors = 5
  // how often to query the state
  val refreshInterval = 10000
  def main(args: Array[String]): Unit = {
    // configure client with host and port of queryable state proxy
    val client = new QueryableStateClient(proxyHost, proxyPort)
    val futures = new Array[
      CompletableFuture[ValueState[(String, Double)]]](numSensors)

    val results = new Array[Double](numSensors)
    // print header line of dashboard table

    val header = (for (i <- 0 until numSensors) yield "sensor_" + (i + 1)
        .mkString("\t| ")

    println(header)
    // loop forever
    while (true) {
      // send out async queries
```

```
      for (i <- 0 until numSensors) {
        futures(i) = queryState("sensor_" + (i + 1), client)
      }
      // wait for results
      for (i <- 0 until numSensors) {
        results(i) = futures(i).get().value()._2
      }
      // print result
      val line = results.map(t => f"$t%1.3f").mkString("\t| ")
      println(line)
      // wait to send out next queries
      Thread.sleep(refreshInterval)
    }
    client.shutdownAndWait()
  }
  def queryState(
      key: String,
      client: QueryableStateClient)
    : CompletableFuture[ValueState[(String, Double)]] = {

    client
      .getKvState[String, ValueState[(String, Double)], (String, Double)](
        JobID.fromHexString(jobId), // jobId
        "maxTemperature", //状态标志符
        key, // 键
        Types.STRING, // 键的类型
        new ValueStateDescriptor[(String, Double)]( //状态描述符
          "", // state name not relevant here 此处与状态名称无关，随便取名
          Types.TUPLE[(String, Double)]))
  }
}
```

第8章 读写外部系统

数据可以**存储**在许多**不同的系统**中，比如**文件系统**、**对象存储**、**关系数据库系统**、**键值存储**、**搜索索引(search indexes)**、**事件日志**、**消息队列**等等。每一类系统都是为特定的访问模式而设计的，有各自擅长的领域。因此，当今的**数据基础设施**常常**由许多不同种类的存储系统组成。**

数据处理系统(如Apache Flink)**通常不包含自己的存储层**，而是**依赖于外部存储系统**来摄取和持久存储数据。因此，对于像Flink这样的数据处理系统来说，**提供**一套齐全的从外部系统读取数据和向外部系统写入数据的**连接器库**以及**提供一套能够自定义连接器的API**是很重要的。

8.1 应用的一致性保障

要**保障应用的一致性**除了依赖于**Flink的检查点机制**之外，还需要**数据源和数据汇连接器**提供一些其他的特性与检查点机制配合。换句话说，应用的一致性保障依赖于三方面

- **检查点机制**

- 数据源连接器提供的**读取位置重置**
- 数据汇连接器提供的**幂等性写**或者**事务性写**

8.1.1 数据源连接器提供的一致性保障

为了为应用程序提供**精确一次**的状态一致性保障，应用的每个**数据源连接器**都需要能够将其**读取位置重置。**

- 当**生成检查点**时，**数据源算子**将**存储其当前读取位置标识符**并在**需要恢复时利用远程存储来恢复这些位置标识符。**
- 支持**读取位置重置**的例子有:
    - **基于文件的数据源**（会存储文件字节流的读取偏置）
    - **Kafka数据源**（会存储消费主题分区的读取偏置）
- 如果**数据源连接器 无法存储和重置**读取位置，那么此应用只能提供**最多一次保证。**

读取位置重置再加上**检查点机制**，就可以提供**至少一次的一致性保障**，如果还需要精确一次的一致性保障，那我们就需要数据汇连接器再提供一些其他的特性。

8.1.2 数据汇连接器提供的一致性保障

8.1.2.1 幂等性写

幂等性是指：对于**同一个系统**，在**同样条件**下，**一次请求**和**重复多次请求 对资源的影响**是**一致**的，就称该操作为幂等的。

> 生活中的幂等操作
>
> 1. 博客系统同一个用户对同一个文章点赞，即使这人单身30年手速疯狂按点赞，那么实际上也只能给这个文章 +1 赞
> 2. 在微信支付的时候，一笔订单应当只能扣一次钱，那么无论是网络问题或者bug等而重新付款，都只应该扣一次钱

在Flink中我们主要在**数据汇连接器**上应用**幂等性的概念**来**提供端到端精确一次的一致性保障。**

> 举个简单的例子，假设我们要统计多个传感器的最高温度并且使用一个键值存储作为数据汇写入的外部系统。那么我们只要在数据汇算子中对**键值存储系统**进行**upsert操作**(看看key是否存在，存在就update，不存在就insert)就可以保证**幂等性。**因为对于一个传感器来说，多次插入数据，最终都只会产生一个k-v对。
>
> 再具体点：

> 1. 经过计算，数据汇发出一条`("sensor_1", 100.0)`，然后这条记录被写入了键值存储系统。
> 2. 但是写入之后，发生了故障，系统恢复故障并重置到上一个检查点，经过重放，数据汇又发出了一条`("sensor_1", 100.0)`，然后这条记录也被写入了键值存储系统。
> 3. 但是幸运的是，因为**upsert**操作是幂等的，插入两条`("sensor_1", 100.0)`与插入一条`("sensor_1", 100.0)`对键值存储系统来说，影响是一样的。
> 4. 借此，我们成功保证了精确一次的一致性

8.1.2.2 事务性写

第二种完全实现**端到端精确一次**的一致性方法是**事务性写**。这里的基本思想是只将那些位于上一个成功检查点之前的事件发送给外部系统。

相比于幂等性写，事务性写**延迟更高**，但是**不会重复发送事件**给外部系统，因此不会出现恢复过程中的不一致。

Flink中有两种事务性数据汇连接器

- write-ahead-log (WAL)数据汇
- 两阶段提交(2PC)数据汇

8.1.2.2.1 WAL

- WAL数据汇将所有**事件 先写入到算子状态**，并在**接收到检查点完成的通知**后再将它们**发送**给外部系统。
- WAL**通用性**很好，可以应对任何类型的外部系统（因为状态是缓存在算子状态中的）
- 但是，它不能提供100%的一致性保证，并且增加了应用的状态大小。

8.1.2.2.2 2PC

- 2PC数据汇**需要外部系统提供对事务的支持。**
- 对于**每个检查点**，2PC数据汇**启动外部系统的一个事务**，并将所有接收到的记录写入到该事务。
- 当它**收到检查点完成的通知**时，它**提交事务。**
- 2PC协议依赖于Flink现有的检查点机制。
    - **检查点分隔符**可以作为**启动新事务**的指令
    - 所有**算子任务**关于其**单个检查点成功**向JobManager**发通知**可以看作是它们的**提交投票**
    - 而来自**JobManager**的**检查点成功**的消息是**提交事务**的指令。

下表展示了不同类型的数据源和数据汇连接器的搭配，能提供的一致性保障

\	不可重置数据源	可重置数据源
任意数据汇	至多一次	至少一次
幂等性数据汇	至多一次	精确一次(故障恢复过程中，会出现临时的不一致，故障恢复后，就恢复精确一次)
WAL数据汇	至多一次	至少一次(无法100%提供精确一次保障)
2PC数据汇	至多一次	精确一次

8.2 内置连接器

Apache Flink提供了各种内置连接器来从各种外部系统读取数据和向各种外部系统写入数据。

- 像Kafka这样的消息队列和事件日志是常见的**外部源**
- 而消息队列、文件系统、键值系统、数据库系统是常见的**外部汇**

为了在应用中使用这些连接器，你需要将相应的依赖项添加到项目的构建文件中。

8.2.1 Apache Kafka 数据源连接器

首先介绍**Kafka**

- Apache Kafka是一个分布式流处理平台。
- 它的核心是一个**分布式的发布-订阅消息系统**，该系统被广泛用于摄入(ingest)和分发(distribute)事件流。
- Kafka将**事件流**组织为所谓的**主题**。
    - 主题是一种事件日志(event log)，它保证事件顺序。
    - 我们可以将**主题划分为多个分区**，这些分区分布在一个集群中。
    - **有序保证仅限于单个分区**—kafka在从不同分区读取时不提供有序保证。Kafka在**分区中读取位置**称为**偏移量**(offset)。

Flink Kafka连接器可以**并行读取**事件流。

- **每个**数据源**算子任务**可以**从一个或多个分区读取**。
- **任务 跟踪**每个分区的当前读取**偏移**，并**将偏置记录到其检查点数据**中。
- 从故障中恢复时，偏移量被取出，任务继续从检查点偏移量读取事件流。

图8-1显示了为数据源算子任务分配分区的情况

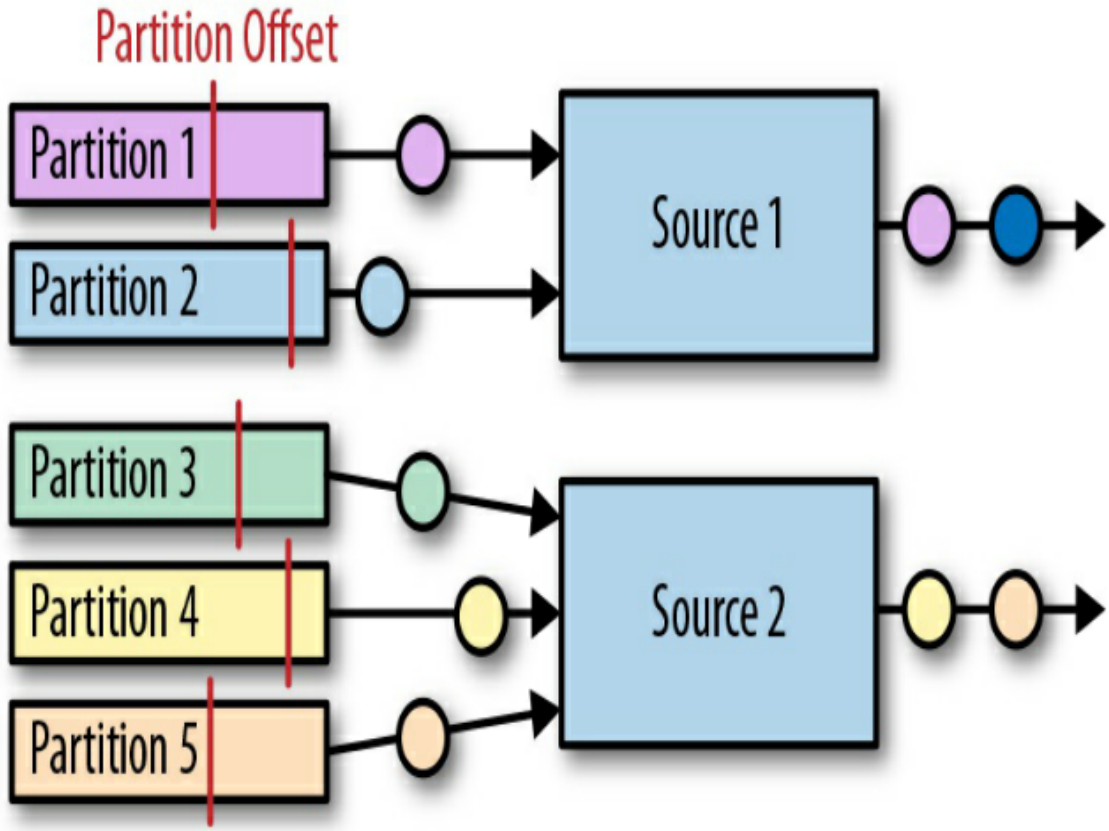


Figure 8-1. Read offsets of Kafka topic partitions

下面看看如何创建一个Kafka数据源连接器

```
val properties = new Properties()
properties.setProperty("bootstrap.servers", "localhost:9092")
properties.setProperty("group.id", "test")
val stream: DataStream[String] = env.addSource(
  new FlinkKafkaConsumer[String](
    "topic", // 主题
    new SimpleStringSchema(), // 反序列化
    properties)) // 配置参数
```

8.2.2 Apache Kafka 数据汇连接器

下面看看如何创建一个Kafka数据汇连接器

```
val stream: DataStream[String] = ...
val myProducer = new FlinkKafkaProducer[String](
  "localhost:9092",         // broker服务器列表
  "topic",                  // 当前主题
  new SimpleStringSchema)   // 序列化
stream.addSink(myProducer) // 设置为数据汇
```

8.2.2.1 Kafka数据汇的至少一次保障

Kafka数据汇在**以下条件都满足**的情况下提供**至少一次的保证**:

- Flink的**检查点机制**是**开启**的
- 所有**数据源**都是**可重置**的
- 如果**写入未成功**，数据汇连接器要**抛出异常**（这样可以让应用失败然后恢复）
- **在提交检查点之前**，**数据汇连接器**要**等待 Kafka 确认**传输中的记录全部写入完毕。

8.2.2.2 Kafka数据汇的精确一次保障

Kafka支持事务性写，因此Flink的Kafka数据汇也能够提供精确一次的一致性保障。但精确一次保障同样需要应用满足8.2.2.1中的几个条件。

FlinkKafkaProducer提供了一个带有Semantic参数的构造函数，该构造函数可以控制数据汇提供的一致性保证级别，该配置的选项如下:

- Semantic.NONE：**不提供**任何一致性保证——记录可能会丢失或多次写入
- Semantic.AT_LEAST_ONCE：**至少一次保证**，记录不会丢失，但可能会重复。这是默认设置。
- Semantic.EXACTLY_ONCE：**精确一次保证**

8.2.2.3 CUSTOM PARTITIONING AND WRITING MESSAGE TIMESTAMPS

当向Kafka主题写入消息时，Flink Kafka**数据源算子任务**可以**选择**要写入主题的哪个**分区**。

- 你可以通过提供一个自定义的FlinkKafkaPartitioner来控制数据到主题分区的路由方式。
- 默认情况下，Flink将每个任务映射到一个Kafka分区。也就是说，由同一个任务发出的所有记录都写入到同一个分区。

通过调用数据汇上的`setWriteTimestampToKafka(true)`，可以**将记录的事件时间戳写入Kafka**。

8.2.3 文件系统数据源连接器

文件系统通常用于以**高性价比**的方式**存储大量数据**。在大数据架构中，它们经常作为批处理应用程序的数据源和数据接收器。通过与高级文件格式(如Apache Parquet或Apache ORC)相结合，**文件系统**可以有效地**为分析查询引擎**(如Apache Hive、Apache Impala或Presto)**服务**。因此，文件系统通常用于**"连接" 流处理应用**和**批处理应用**。

Apache Flink内置了一个文件数据源连接器，它**支持重置**，并可以将文件中的数据**作为数据流**输入到应用中。此外，它还支持多种类型的文件系统，比如：本地文件系统、HDFS、S3等等。

文件系统数据源的创建方法如下

```
val lineReader = new TextInputFormat(null)
val lineStream: DataStream[String] = env.readFile[String](
  lineReader,                 // The FileInputFormat 文件输入格式，FileInputFormat
的子类
  "hdfs:///path/to/my/data",  // The path to read 路径
  FileProcessingMode.PROCESS_CONTINUOUSLY,    // The processing mode 处理模式
  30000L) // The monitoring interval in ms 扫描的间隔时间
```

FileProcessingMode是目标路径的读取模式，有如下两种选择

- 在**PROCESS_ONCE**模式下，当**作业启动**并读取所有匹配的文件时，读取路径只被**扫描一次。**
- 在**PROCESS_CONTINUOUSLY**中，会**定期扫描路径**(在初始扫描之后)，并持续读取新的和修改的文件。

8.2.4 文件系统数据汇连接器

将数据流写入文件是一种常见的需求。由于大多数应用只能在文件最终完成后读取文件，并且流应用运行时间很长，因此数据汇连接器通常会**将输出分块存储**到多个文件中。

当满足如下条件时，**文件系统数据汇连接器**可以为应用提供端到端的**精确一次保障**：

1. 应用开启**检查点机制**
2. 所有**数据源**都支持**重置**

下面看看如何创建一个文件系统数据汇连接器

```
val input: DataStream[String] = …
val sink: StreamingFileSink[String] = StreamingFileSink
  .forRowFormat(
    new Path("/base/path"),  //基础路径
    new SimpleStringEncoder[String]("UTF-8")) //编码器
  .build()
input.addSink(sink)
```

数据汇的文件分块机制分为三级：

- **数据流被划分为多个桶**
  - 每条记录都会被分配到一个桶中
  - 每个**桶**都**对应**基础路径下的一个**子路径**
  - 如果没有显式指定，Flink**根据事件的处理时间**将记录分配给**每小时一个**的桶中。
- **每个桶中包含多个part文件**
  - 每个bucket路径下包含多个part文件
  - 这些part文件由数据汇算子的**多个任务并发地写入。**
  - 此外，每个并行任务也将其输出分割成多个part文件。

    - 这些分块文件路径：`[base-path]/[bucket-path]/part-[task-idx]-[id]`
    - 例如，基础路径为`/johndoe/demo/`，分块文件前缀为`part`，则路径`/johndoe/demo/2018-07-22-17/part-4-8`指向2018年7月22日下午5点这个桶内(`/2018-07-22-17/`)，由第4个接收任务的写入的第8个文件(`/part-4-8`)。

StreamingFileSink提供了两种文件编码模式：**行编码**和**批量编码**。

- 在行编码模式中，每个记录都被单独编码然后写入到一个part文件中。我们调用`StreamingFileSink.forRowFormat()`方法就是使用行编码模式
- 在批量编码模式中，多条记录被一起编码然后写入到一个part文件中。我们调用`StreamingFileSink.forBulkFormat()`方法就是使用批量编码模式

8.2.5 Apache Cassandra 数据汇连接器

Apache Cassandra是一种**可伸缩**的、**高可用**的**列式存储**数据库系统。

- Cassandra将数据集建模为由多个类型的列所组成的行表(Cassandra models datasets as tables of rows that consist of multiple typed columns.)。
- 可以将一个或多个列定义为(复合)主键。每一行都可以通过其主键唯一地标识。
- Cassandra提供了Cassandra查询语言(CQL)，这是一种类似sql的语言，用于读写记录以及创建、修改和删除数据库对象。

Flink的Cassandra连接器是可以**提供精确一次保障**的

- Cassandra的数据模型**基于主键**(K-V式的)，所有写到Cassandra的操作都使用**upsert语义**(有就update，没有就insert)。因此，基于Cassandra的数据汇写入操作是**幂等**的
- 与此同时，为了避免故障恢复期间的短暂不一致，Cassandra连接器可以通过配置**开启WAL机制**。

下面是例子

首先定义一个Cassandra表结构

```
// 创建键空间
CREATE KEYSPACE IF NOT EXISTS example
  WITH replication = {'class': 'SimpleStrategy', 'replication_factor': '1'};
// 创建表
CREATE TABLE IF NOT EXISTS example.sensors (
  sensorId VARCHAR,
  temperature FLOAT,
  PRIMARY KEY(sensorId)
);
```

下面先展示如何将tuple、case class等类型写入Cassandra

```
val readings: DataStream[(String, Float)] = ???
// 新建数据汇构造器
val sinkBuilder: CassandraSinkBuilder[(String, Float)] =
CassandraSink.addSink(readings)
// 构造
sinkBuilder
  .setHost("localhost")
  //CQL insert语句
  .setQuery("INSERT INTO example.sensors(sensorId, temperature) VALUES (?, ?);")
  .build()
```

- 写入tuple、case class时需要指定CQL INSERT查询。
- 数据汇将查询注册为PreparedStatement，并将tuple或case class的字段转换为PreparedStatement中的参数。
- **字段 根据其位置**映射到参数；第一个值被转换为第一个参数。

下面再展示如何将POJO写入Cassandra

```
val readings: DataStream[SensorReading] = ???
CassandraSink.addSink(readings)
  .setHost("localhost")
  .build()
```

```
@Table(keyspace = "example", name = "sensors")
class SensorReadings(
  @Column(name = "sensorId") var id: String,
  @Column(name = "temperature") var temp: Float) {
  def this() = {
      this("", 0.0)
 }
  def setId(id: String): Unit = this.id = id
  def getId: String = id
  def setTemp(temp: Float): Unit = this.temp = temp
  def getTemp: Float = temp
}
```

- 需要通过**注解**来**映射**POJO字段到Cassandra列

8.3 实现自定义数据源函数

DataStream API提供了**两个接口**来**自定义数据源连接器**，这两个接口都有相应的RichFunction抽象类:

- SourceFunction用于**非并行**的数据源连接器
- ParallelSourceFunction用于支持**同时运行多个任务**的数据源连接器

这两个接口中的方法一样，如下所示

签名	描述
`void run(SourceContext<T> ctx)`	run()方法负责执行实际的事件读入工作。Flink会**开一个专门的线程**，然后在这个线程中调用run()方法一次。
`void cancel()`	负责**终止事件读入**

下面举一个简单的例子，它创建一个简单的从0数到Long.maxValue的自定义数据源函数

```
class CountSource extends SourceFunction[Long] {

  var isRunning: Boolean = true

  // 实际的数据读入工作
  override def run(ctx: SourceFunction.SourceContext[Long]): Unit = {

    var cnt: Long = -1
    while (isRunning && cnt < Long.MaxValue) {
      // 不断的++
      cnt += 1
      ctx.collect(cnt)
    }

  }

  // 终止数据读入
  override def cancel(): Unit = isRunning = false
}
```

8.3.1 可重置的数据源函数

Flink只能在数据源连接器支持重放输入数据时才能提供各种一致性保证。

- 如果**外部源系统**提供了**读取偏移量**和**重置偏移量**的API，则**数据源算子可以重放输入数据。**
- 比如：**文件系统。**它提供文件流偏移量，它还提供将文件流移动到特定位置的seek方法，
- 再比如：**Apache Kafka**，它为主题的每个分区提供偏移量，并可以设置分区的读取位置。
- 一个**反例是web socket**，它从网络套接字读取数据，并且立即丢弃交付的数据，因此它不支持重放

支持重放的数据源算子要与**检查点机制 配合**

- 在**生成检查点**时，数据源算子要**持久化**所有当前**读取偏移量。**
- **故障恢复**时，数据源算子要从最新的检查点**恢复偏移量**到本地。

如果数据源算子要**支持重放**，则需要实现CheckpointedFunction接口，具体例子如下

```
// 可重置的SourceFunction
```

```
class ReplayableCountSource extends SourceFunction[Long] with
CheckpointedFunction {

  var isRunning: Boolean = true
  var cnt: Long = _
  // 用来保存offset的算子列表状态
  var offsetState: ListState[Long] = _
  // 实际的数据读入工作
  override def run(ctx: SourceFunction.SourceContext[Long]): Unit = {

    while (isRunning && cnt < Long.MaxValue) {
      //加锁，但是讲道理我感觉只在这里加锁没什么用，除非框架进行了特殊的处理
      ctx.getCheckpointLock.synchronized {
        // increment cnt
        cnt += 1
        ctx.collect(cnt)
      }
    }
  }

  override def cancel(): Unit = isRunning = false

  // 生成检查点时，这个方法被调用来同步状态
  override def snapshotState(snapshotCtx: FunctionSnapshotContext): Unit = {
    // remove previous cnt
    offsetState.clear()
    // add current cnt
    offsetState.add(cnt)
  }

  // 初始化和故障恢复时会调用这个方法
  override def initializeState(initCtx: FunctionInitializationContext): Unit = {
    // obtain operator list state to store the current cnt
    val desc = new ListStateDescriptor[Long]("offset", classOf[Long])
    offsetState = initCtx.getOperatorStateStore.getListState(desc)
    // 从检查点中恢复当前的进度
    // initialize cnt variable from the checkpoint
    val it = offsetState.get()
    cnt = if (null == it || !it.iterator().hasNext) {
      -1L
    } else {
      it.iterator().next()
    }
  }
}
```

8.3.2 数据源函数、时间戳及水位线

DataStream API提供了**两种方式**来**分配时间戳**和**生成水位线**。

- 时间戳和水位线可以由专用的`TimestampAssigner`分配和生成
- 时间戳和水位线也可以由`SourceFunction`分配和生成。

SourceFunction的`SourceContext`对象提供了分配水位线和时间戳的方法

```
@PublicEvolving
void collectWithTimestamp(T element, long timestamp);

@PublicEvolving
void emitWatermark(Watermark mark);
```

注意：当**单个数据源算子任务 并行处理 多个partition**时，使用`SourceFunction`来生成水位线比使用单独的`TimestampAssigner`更好，具体原因略、

8.4 实现自定义数据汇函数

DataStream API提供了SinkFunction接口来自定义数据汇函数。SinkFunction接口中只有一个方法:

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

下面示例显示了一个简单的SinkFunction，它将传感器读数写入套接字。

```
// 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)

//......

/**
  * Writes a stream of [[SensorReading]] to a socket.
  */
class SimpleSocketSink(val host: String, val port: Int)
    extends RichSinkFunction[SensorReading] {

  var socket: Socket = _
  var writer: PrintStream = _

  //设置socket连接和writer
  override def open(config: Configuration): Unit = {
    // open socket and writer
    socket = new Socket(InetAddress.getByName(host), port)
    writer = new PrintStream(socket.getOutputStream)
  }

  //用writer写入到来的事件
  override def invoke(
      value: SensorReading,
      ctx: SinkFunction.Context[_]): Unit = {
    // write sensor reading to socket
    writer.println(value.toString)
    writer.flush()
```

```
  }

  //关闭
  override def close(): Unit = {
    // close writer and socket
    writer.close()
    socket.close()
  }
}
```

为了实现**端到端精确一次保障**，数据汇连接器需要**幂等**或支持**事务**，下面我们看看这两种

8.4.1 幂等性数据汇连接器

如果外部汇系统满足如下两个条件，SinkFunction接口就足以实现幂等性数据汇连接器

- 结果数据具有**固定的键**，可以**在该键上执行幂等更新。**
  - 对于计算每个传感器和每分钟的平均温度的应用程序，固定的键可以是传感器的ID和每分钟的时间戳。
- **外部系统支持按键更新**，例如关系数据库系统可以按照主键更新(update … where sensor_id = xxx)或键值存储可以按照key更新

下面的例子演示了如何实现和使用一个幂等的SinkFunction，该函数将事件写入JDBC数据库。

```
class DerbyUpsertSink extends RichSinkFunction[SensorReading] {

  var conn: Connection = _
  var insertStmt: PreparedStatement = _
  var updateStmt: PreparedStatement = _

  // 初始化：准备好insertStmt和updateStmt
  override def open(parameters: Configuration): Unit = {
    // connect to embedded in-memory Derby
    val props = new Properties()
    conn = DriverManager.getConnection("jdbc:derby:memory:flinkExample", props)
    // prepare insert and update statements
    insertStmt = conn.prepareStatement(
      "INSERT INTO Temperatures (sensor, temp) VALUES (?, ?)")
    updateStmt = conn.prepareStatement(
      "UPDATE Temperatures SET temp = ? WHERE sensor = ?")
  }

  // 实际的处理函数：先尝试更新失败就插入
  override def invoke(r: SensorReading, context: Context[_]): Unit = {
    // set parameters for update statement and execute it
    updateStmt.setDouble(1, r.temperature)
    updateStmt.setString(2, r.id)
    updateStmt.execute()
    // execute insert statement if update statement did not update any row
```

```
    if (updateStmt.getUpdateCount == 0) {
      // set parameters for insert statement
      insertStmt.setString(1, r.id)
      insertStmt.setDouble(2, r.temperature)
      // execute insert statement
      insertStmt.execute()
    }
  }

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

8.4.2 事务性数据汇连接器

为了简化事务性数据汇的实现，Flink的DataStream API提供了两个模板，可以通过继承这些模板来实现自定义数据汇算子。两个模板都实现了CheckpointListener接口来接收JobManager发出的检查点已完成的通知

- **GenericWriteAheadSink**模板暂存每个检查点周期内所有需要发出到外部系统的记录，并将它们存储在数据汇算子任务的算子状态中。当任务接收到检查点完成通知时，它将已完成的那个检查点周期内的记录写入外部系统。
- **TwoPhaseCommitSinkFunction**模板**利用了外部汇系统的事务特性。**对于每个检查点，它启动一个新事务，并将这个检查点周期内的事件写入到这个事务中。

8.4.2.1 GenericWriteAheadSink(WAL)

GenericWriteAheadSink的工作方式如下：

- 它将所有接收到的记录添加(append)到使用检查点来分段(segmented)的**write ahead log**(WAL)中。
- 每次数据汇算子接收到**检查点分隔符**时，它**将开启一个新的section**，并将以下所有记录追加到新section。
- **WAL作为算子状态**被存储，当生成检查点时，WAL被发送给远程存储。
- 由于**WAL可以在出现故障时恢复**，因此不会丢失任何记录。

当GenericWriteAheadSink收到关于已完成检查点的通知时，它会将WAL中对应这个检查点的section中所有记录发送给外部系统。

当所有记录都被成功发出后，GenericWriteAheadSink将在**内部提交**相应的**检查点。**检查点需要通过两个步骤提交。

- 首先，数据汇**永久存储**"这个检查点已提交了"这条**提交信息。**
- 其次，它**从WAL中删除**对应的section中全部**记录。**
- 注意：**提交信息不存储算子状态中。**相反，GenericWriteAheadSink依赖于一个称为**CheckpointCommitter的可插入组件**来**存储和查找提交信息。**

下面看看如何利用GenericWriteAheadSink来实现一个事务性数据汇连接器

```
avgTemp.transform(
        "WriteAheadSink",
        new StdOutWriteAheadSink)
      // enforce sequential writing
      .setParallelism(1)
/**
  * Write-ahead sink that prints to standard out and commits checkpoints to the
local file system.
  */
// GenericWriteAheadSink抽象类需要传入三个参数
// 1 用于存储提交信息的CheckpointCommitter
// 2 TypeSerializer 用来序列化
// 3 任务id
class StdOutWriteAheadSink extends GenericWriteAheadSink[(String, Double)](
    // CheckpointCommitter that commits checkpoints to the local file system
    new FileCheckpointCommitter(System.getProperty("java.io.tmpdir")),
    // Serializer for records
    createTypeInformation[(String, Double)].createSerializer(new
ExecutionConfig),
    // Random JobID used by the CheckpointCommitter
    UUID.randomUUID.toString) {

  // 并且我们需要自己实现sendValues方法
  override def sendValues(
      readings: Iterable[(String, Double)],
      checkpointId: Long,
      timestamp: Long): Boolean = {

    for (r <- readings.asScala) {
      // write record to standard out 将输入写入到标准输出中(就举个简单例子)
      // 运行时可以看到stdout中没有重复记录也不少记录
      println(r)
    }
    true
  }
}
```

如前所述，GenericWriteAheadSink不能提供100%的精确一次一致性保证。有两种失败情况会导致记录被多次发出:

- 程序在任务**运行sendValues()方法时发生故障。**
  - 这时，单个section中有的记录被写入了外部系统但是有的没有写入。
  - 然后由于检查点尚未提交，当恢复时，数据汇将再次写入这个section中的所有记录。
  - 从而导致记录多次发送。

- 所有记录都被正确写入，sendValues()方法返回true；但是，**程序在调用CheckpointerCommitter之前失败，或者CheckpointerCommitter未能提交检查点**。这样，在恢复期间，系统会误认为这个checkpoint未内部提交，并再次写入这个section中的所有记录。

8.4.2.1 TwoPhaseCommitSinkFunction(2PC)

TwoPhaseCommitSinkFunction是这样实现2PC协议的。

- 在数据汇向外部汇系统发出它的第一个记录之前，它在外部汇系统上启动一个事务。
- 这之后收到的所有记录都写入到这个事务中。
- 当JobManager**启动一个检查点并在数据源中放入检查点分隔符时**，2PC协议的**投票阶段开始**。
- 当**一般算子**接收到检查点分隔符时，它向**远程存储**同步自己的状态，并在完成之后**向JobManager发送确认消息**。
- 当**数据汇算子任务**接收到检查点分隔符时，它向**远程存储**同步自己的状态，准备好要提交事务，并在完成之后**向JobManager发送确认消息**。
- JobManager收到的确认消息类似于2PC协议的提交投票。**数据汇任务此时还不能提交事务**，因为不能保证所有其他数据汇任务都到达了检查点。此时**数据汇任务会再开启一个新的事务**，将到来的记录写入到这个新事务中。
- 当JobManager**从所有任务上接收到确认消息**时，它将**检查点完成通知 发送**给所有订阅通知的任务。此通知对应于2PC协议的commit命令。
- 当**数据汇任务收到通知**时，它**提交检查点对应的事务**。
- **当所有的数据汇任务都提交了它们的事务**时，2PC协议的迭代就**成功**了。

要实现2PC协议，还需要外部汇系统满足一些要求

- **外部汇系统**必须**提供事务支持**。
- 在**检查点间隔**期间，**事务**必须**打开**并接受**写**操作。
- **事务必须直到收到检查点完成通知才能提交**。在恢复周期的情况下，这可能需要一些时间。如果接收系统关闭事务(例如因超时关闭了事务)，未提交的数据将丢失。
- 外部汇系统必须能够在进程**失败**后**恢复**事务。
- **提交事务必须是幂等操作**——外部汇系统应该能够注意到事务已经提交，或者重复提交必须没有效果。

下面实例将2PC作用在一个文件系统上

```
class TransactionalFileSink(val targetPath: String, val tempPath: String)
    // IN => (String, Double) 输入记录的类型
    // TXT => String 事务标识符的类型
    // CONTEXT => Void 上下文类型，void代表不需要上下文
    extends TwoPhaseCommitSinkFunction[(String, Double), String, Void](
      // 序列化器
      createTypeInformation[String].createSerializer(new ExecutionConfig),
      // 序列化器
      createTypeInformation[Void].createSerializer(new ExecutionConfig)) {

  var transactionWriter: BufferedWriter = _

  /**
    * 当需要开启新事务的时候就新建一个临时文件
    */
```

```
  override def beginTransaction(): String = {

    // path of transaction file is constructed from current time and task index
    val timeNow = LocalDateTime.now(ZoneId.of("UTC"))
      .format(DateTimeFormatter.ISO_LOCAL_DATE_TIME)
    val taskIdx = this.getRuntimeContext.getIndexOfThisSubtask
    // 文件名 = 时间 + 任务id
    val transactionFile = s"$timeNow-$taskIdx"

    // 创建文件和writer
    val tFilePath = Paths.get(s"$tempPath/$transactionFile")
    Files.createFile(tFilePath)
    this.transactionWriter = Files.newBufferedWriter(tFilePath)
    println(s"Creating Transaction File: $tFilePath")

    // 文件名被返回，作为事务的标识符
    transactionFile
  }

  /** Write record into the current transaction file. */
  /** 将记录写入到当前临时文件. */
  override def invoke(transaction: String, value: (String, Double), context:
Context[_]): Unit = {
    transactionWriter.write(value.toString)
    transactionWriter.write('\n')
  }

  /** Flush and close the current transaction file. */
  override def preCommit(transaction: String): Unit = {
    transactionWriter.flush()
    transactionWriter.close()
  }

  /** Commit a transaction by moving the pre-committed transaction file
    * to the target directory.
    */
  /** 通过将临时文件移动到目标路径来提交事务    */
  override def commit(transaction: String): Unit = {
    val tFilePath = Paths.get(s"$tempPath/$transaction")
    // check if the file exists to ensure that the commit is idempotent.
    if (Files.exists(tFilePath)) {
      val cFilePath = Paths.get(s"$targetPath/$transaction")
      Files.move(tFilePath, cFilePath)
    }
  }

  /** Aborts a transaction by deleting the transaction file. */
  /** 通过将临时文件删除来关闭事务    */
  override def abort(transaction: String): Unit = {
    val tFilePath = Paths.get(s"$tempPath/$transaction")
    if (Files.exists(tFilePath)) {
      Files.delete(tFilePath)
    }
  }
}
```

8.5 异步访问外部系统

除了接收(ingest)或发出(emit)数据流之外，**通过在远程数据库中查找信息来丰富数据流**是另一个需要与外部存储系统交互的常见场景。比如：雅虎的广告分析系统，它需要使用存储在键值存储中的对应广告的详细信息来丰富广告点击流。

对于这类场景，**最直接的方法是实现一个MapFunction**，它为每个记录查询外部系统，等待查询返回结果，丰富记录，并发出结果。虽然这种方法很容易实现，但存在一个主要问题:**对外部系统的每个请求都会增加显著的延迟**(一个请求/响应涉及两个网络消息)，而MapFunction将大部分时间用于等待查询结果。

Flink提供了**AsyncFunction**来**减轻远程I/O调用的延迟**。AsyncFunction**并发地发送**多个**查询**并**异步地处理**它们的**结果**。

下面来看看`AsyncFunction`的源代码

```
trait AsyncFunction[IN, OUT] extends Function {
  // input: 输入记录
  // ResultFuture[OUT]: 用于返回函数结果的异步Future对象(也有可能返回一个异常)
  def asyncInvoke(input: IN, resultFuture: ResultFuture[OUT]): Unit
}
```

下面展示如何使用`AsyncFunction`来并发查询关系型数据库(这里使用Derby数据库)

```
val readings: DataStream[SensorReading] = ???
val sensorLocations: DataStream[(String, String)] =
AsyncDataStream
  .orderedWait(
    readings,
    new DerbyAsyncFunction,
    5, TimeUnit.SECONDS,     // timeout requests after 5 seconds
    100)                     // at most 100 concurrent requests
// .....
class DerbyAsyncFunction extends AsyncFunction[SensorReading, (String, String)]
{

  // caching execution context used to handle the query threads
  private lazy val cachingPoolExecCtx =
    ExecutionContext.fromExecutor(Executors.newCachedThreadPool())
  // 用于将结果Future转发给回调
  private lazy val directExecCtx =
    ExecutionContext.fromExecutor(
      org.apache.flink.runtime.concurrent.Executors.directExecutor())

  /** Executes JDBC query in a thread and handles the resulting Future
    * with an asynchronous callback. */
  override def asyncInvoke(
      reading: SensorReading,
```

```
      resultFuture: ResultFuture[(String, String)]): Unit = {

    val sensor = reading.id

    // 创建一个Future，从数据库中获取room信息
    val room: Future[String] = Future {
      // Creating a new connection and statement for each record.
      // Note: This is NOT best practice!
      // Connections and prepared statements should be cached.
      val conn = DriverManager
        .getConnection("jdbc:derby:memory:flinkExample", new Properties())
      val query = conn.createStatement()

      // submit query and wait for result. this is a synchronous call.
      val result = query.executeQuery(
        s"SELECT room FROM SensorLocations WHERE sensor = '$sensor'")

      // get room if there is one
      val room = if (result.next()) {
        result.getString(1)
      } else {
        "UNKNOWN ROOM"
      }

      // close resultset, statement, and connection
      result.close()
      query.close()
      conn.close()

      // sleep to simulate (very) slow requests
      Thread.sleep(2000L)

      // return room
      room
    }(cachingPoolExecCtx) // 第二个参数是一个线程池

    // 给room这个Future注册一个回调
    room.onComplete {
      case Success(r) => resultFuture.complete(Seq((sensor, r)))
      case Failure(e) => resultFuture.completeExceptionally(e)
    }(directExecCtx)
  }
}
```

- asyncInvoke()方法包装了阻塞式的JDBC查询
- 这些JDBC查询通过CachedThreadPool执行。
- Future[String]返回JDBC查询的结果。
- 最后，我们调用Future对象(也就是上面代码的room常量)onComplete()回调，并将查询结果传递给ResultFuture handler。
- 将查询结果传递给ResultFuture handler是由DirectExecutor处理的。

值得一提的是，`asyncInvoke`**方法本身**与其他算子函数没有区别，都是**串行调用**的。只不过它在内部使用了Future对象来发出并回收异步请求。