# 第三章 Apache Flink架构

在第二章，我们已经讨论了比如并行度、时间和状态等分布式流处理中的重要概念；对于这些方面，Flink是怎样处理的呢？本章将会就此做一个Flink架构的高层级介绍。我们要特别介绍一下Flink的分布式架构，看看它是如何在流处理应用程序中处理时间和状态的，另外还要讨论一下它的容错（fault-tolerance）机制。本章中也会给出一些用Flink实现和操作流处理程序的一些背景知识，这可以帮助我们更好地理解Flink的内部结构以及流处理程序的性能和行为。

## 3.1 系统架构

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

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

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

### Flink运行时组件

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

* 作业管理器（JobManager）是控制一个应用程序执行的主进程，也就是说，每个应用程序都会被一个不同的JobManager所控制执行。JobManager会先接收到要执行的应用程序。这个应用程序会包括：作业图（JobGraph）、逻辑数据流图（logical dataflow graph）和打包了所有的类、库和其它资源的JAR包。JobManager会把JobGraph转换成一个物理层面的数据流图，这个图被叫做“执行图”（ExecutionGraph），包含了所有可以并发执行的任务。JobManager会向资源管理器（ResourceManager）请求执行任务必要的资源，也就是任务管理器（TaskManager）上的插槽（slot）。一旦它获取到了足够的资源，就会将执行图分发到真正运行它们的TaskManager上。而在运行过程中，JobManager会负责所有需要中央协调的操作，比如说检查点（checkpoints）的协调。
* ResourceManager主要负责管理任务管理器（TaskManager）的插槽（slot），TaskManger插槽是Flink中定义的处理资源单元。Flink为不同的环境和资源管理工具提供了不同资源管理器（ResourceManager），比如YARN、Mesos、K8s，以及standalone部署。当JobManager申请插槽资源时，ResourceManager会将有空闲插槽的TaskManager分配给JobManager。如果ResourceManager没有足够的插槽来满足JobManager的请求，它还可以向资源提供平台发起会话，以提供启动TaskManager进程的容器。另外，ResourceManager还负责终止空闲的TaskManager，释放计算资源。
* 任务管理器（TaskManager）是Flink中的工作进程。通常在Flink中会有多个TaskManager运行，每一个TaskManager都包含了一定数量的插槽（slots）。插槽的数量限制了TaskManager能够执行的任务数量。启动之后，TaskManager会向资源管理器注册它的插槽；收到资源管理器的指令后，TaskManager就会将一个或者多个插槽提供给JobManager调用。JobManager就可以向插槽分配任务（tasks）来执行了。在执行过程中，一个TaskManager可以跟其它运行同一应用程序的TaskManager交换数据。任务的执行和插槽的概念会在“任务执行”一节做具体讨论。
* 分发器（Dispatcher）可以跨作业运行，它为应用提交提供了REST接口。当一个应用被提交执行时，分发器就会启动并将应用移交给一个JobManager。由于是REST接口，所以Dispatcher可以作为集群的一个HTTP接入点，这样就能够不受防火墙阻挡。Dispatcher也会启动一个Web UI，用来方便地展示和监控作业执行的信息。Dispatcher在架构中可能并不是必需的，这取决于应用提交运行的方式。

图3-1展现了当一个应用提交执行时，Flink的各个组件是如何交互协作的：

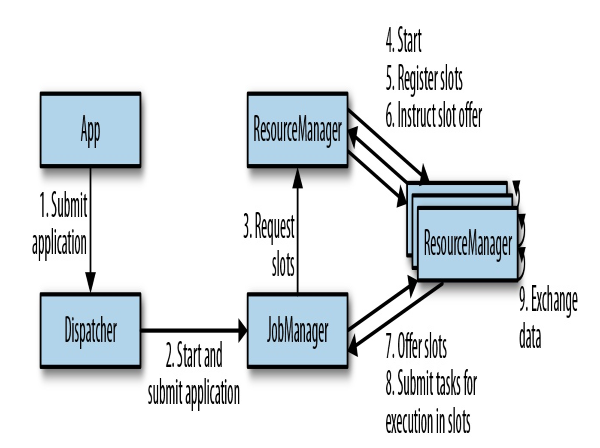


图3-1 应用提交和组件交互

注：

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

### 应用部署

Flink应用程序可以用以下两种不同的方式部署：

* 框架（Framework）方式

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

* 库（Library）方式

在这个模式下，Flink Application 会被打包在一个容器（container） 镜像里，例如一个Docker 镜像。此镜像包含了运行JobManager和ResourceManager的代码。当一个容器从镜像启动后，它会自动启动ResourceManager和JobManager，并提交打包好的应用。另一种方法是：将应用打包到镜像后，只用于部署TaskManager容器。从镜像启动的容器会自动启动一个TaskManager，然后连接ResourceManager并注册它的slots。这些镜像的启动以及失败重启，通常都会由一个外部的资源管理器管理（比如Kubernetes）。

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

### 任务执行

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

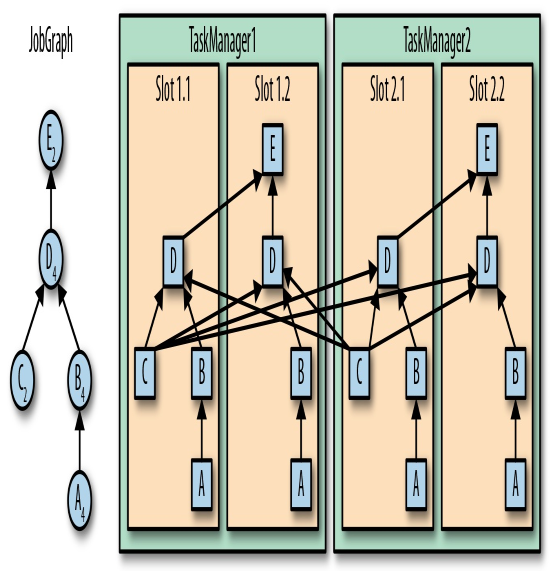


图3-2 算子（operators），任务（tasks）和处理插槽（slots）

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

TaskManager在同一个JVM中以多线程的方式执行任务。线程较进程会更轻量级，但是线程之间并没有对任务进行严格隔离。所以，单个任务的异常行为有可能会导致整个TaskManager进程挂掉，当然也同时包括运行在此进程上的所有任务。通过为每个TaskManager配置单独的slot，就可以将应用在TaskManager上相互隔离开来。TaskManager内部有多线程并行的机制，而且在一台主机上可以部署多个TaskManager，所以Flink在资源配置上非常灵活，在部署应用时可以充分权衡性能和资源的隔离。我们将会在第九章对Flink集群的配置和搭建继续做详细讨论。

### 高可用配置

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

* TaskManager 故障

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

* JobManager故障

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

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

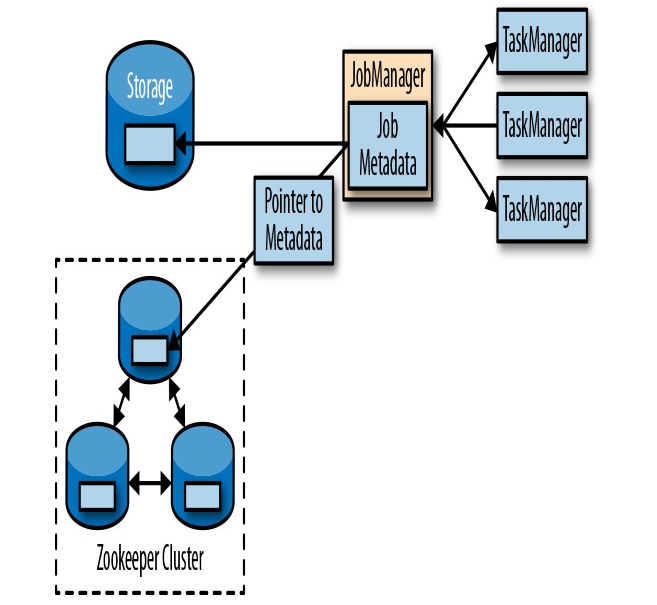


图3-3 Flink高可用架构

当一个JobManager失败，所有属于这个应用的任务都会自动取消。一个新的JobManager接管工作，会执行以下操作：

1. 从ZooKeeper请求存储位置（storage location），从远端存储获取JobGraph，Jar文件，以及应用最近一次检查点（checkpoint）的状态句柄（state handles）

2. 从ResourceManager请求slots，用来继续运行应用

3. 重启应用，并将所有任务的状态，重设为最近一次已完成的检查点

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

## 3.2 Flink中的数据传输

运行中的应用任务，会持续不断地交换数据。TaskManager负责将数据从“发送任务”（sending tasks）传递到“接收任务”（receiving tasks）。TaskManager的网络组件会在缓冲区中收集数据，然后再将其发送，也就是说，数据不是逐条发送的，而是在缓冲区中“攒”成了一批。这种技术是有效利用网络资源和实现高吞吐量的基础，机制类似于网络或磁盘I/O协议中使用的缓冲技术。

注：

通过缓冲区来传递数据，意味着Flink的处理模型是基于微批的。

每个TaskManager都有一个网络缓冲池（默认大小为32KB），用于发送和接收数据。如果发送任务和接收任务运行在不同的TaskManager进程中，那么它们会通过操作系统的网络栈来进行通信。流应用程序需要以管道方式传递数据，所以每对TaskManager之间都需要维护一个永久TCP连接，用来交换数据。在无序连接模式下，每个发送任务都需要能向任何接收任务传递数据。所以我们发现，TaskManager需要为每一个接收任务设置一个专用的网络缓冲区，因为其中的每一个任务都需要接收数据。图3-4展示了这种架构。

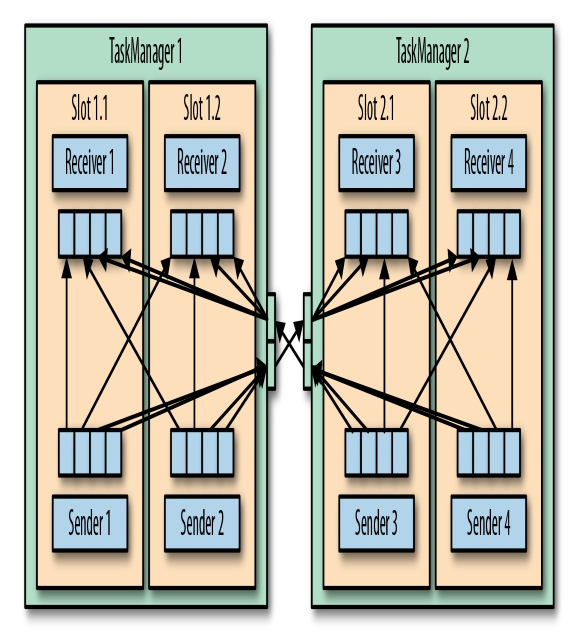


图3-4 TaskManager之间的数据传输

如图3-4所示，四个发送任务中的每一个都需要至少四个网络缓冲区，用来向每个接收任务发送数据，而每个接收任务也需要至少四个缓冲区来接收数据。需要发送到另一个TaskManager的缓冲数据，会复用同一网络连接。为了实现平滑的管道数据传输，TaskManager必须能够提供足够的缓冲，来同时为所有传出和传入连接提供服务。对于无序或广播连接，每个发送任务都需要为每个接收任务提供一个缓冲；所以，所需缓冲区的数量是相关算子任务数量的平方。Flink网络缓冲区的默认配置足以满足中小型应用；对于更大的应用场景，就需要按照“主内存和网络缓冲区”一节中的叙述调整配置了。

当发送任务和接收任务在同一个TaskManager进程中运行时，发送任务会将传出的数据序列化，放入字节缓冲区，并在缓冲区填满后将其放入队列。接收任务从队列中提取缓冲数据并对其进行反序列化。因此，在同一个TaskManager上运行的任务，它们之间的数据传输不会导致网络通信。

Flink采用不同的技术来降低任务之间的通信成本。在下面的部分中，我们会简要讨论基于信任度（Credit）的流控制和任务链。

### 基于信任度（credit）的流控制

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

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

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

### 任务链（Task Chaining）

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

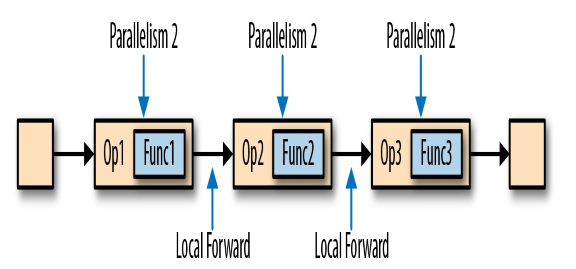


图3-5 符合任务链要求的算子管道

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

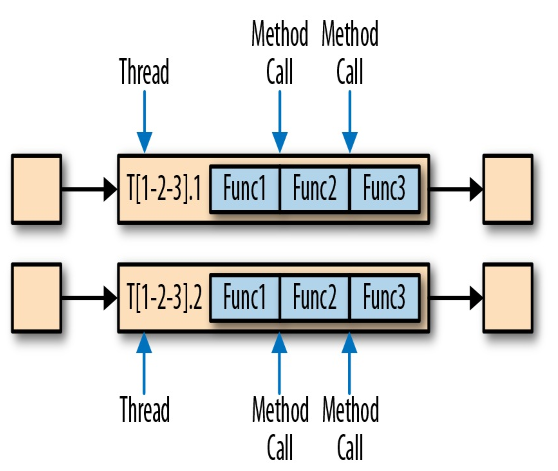


图3-6 任务链的执行过程

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

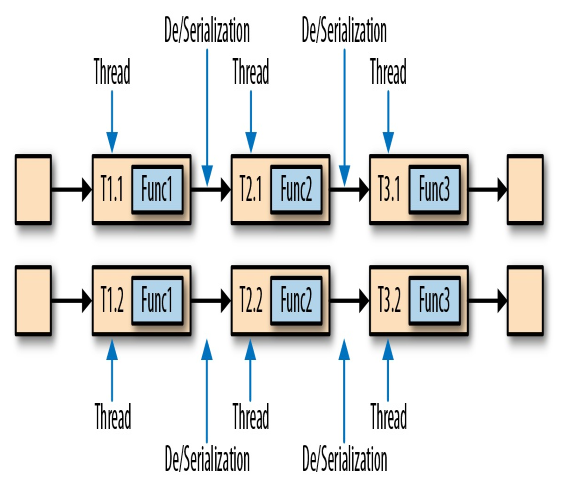


图3-7 未链接的任务执行过程

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

## 3.3 事件时间（Event-Time）处理

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

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

### 时间戳（Timestamps）

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

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

### 水位线（Watermarks）

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

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

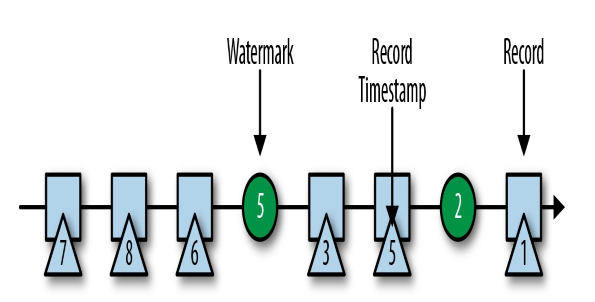


图3-8 具有时间戳记录和水位线的数据流

水位线有两个基本属性：

1. 必须单调递增，以确保任务的事件时间时钟在向前推进，而不是在后退。

2. 它们与数据的时间戳相关。带有时间戳T的水位线表示，所有后续数据的时间戳都应该大于T。

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

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

### watermark的传递和事件时间

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

当任务收到watermark时，将执行以下操作：

1. 任务根据watermark的时间戳更新其内部事件时钟。

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

3. 任务会发出一个带有更新后的事件时间的watermark。

注：

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

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

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

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

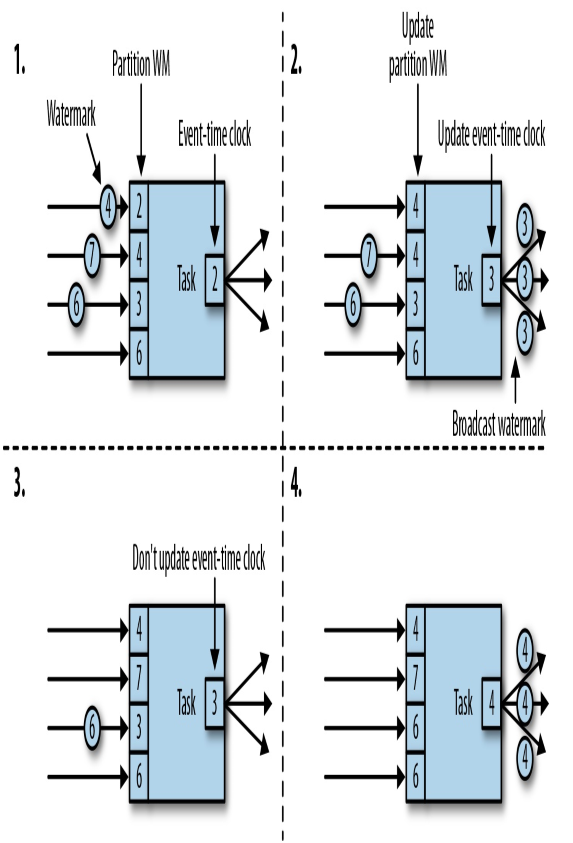


图3-9 通过水位线（watermark）来更新任务的事件时间

具有两个或多个输入流（如Union或CoFlatMap）的算子任务（参见“多流转换”一节）也会以所有分区watermark的最小值作为事件时间时钟。它们并不区分不同输入流的分区watermark，所以两个输入流的数据都是基于相同的事件时间时钟进行处理的。当然我们可以想到，如果应用程序的各个输入流的事件时间不一致，那么这种处理方式可能会导致问题。

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

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

### 时间戳的分配和水位线的产生

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

1. 在数据源（source）处分配：当数据流被摄入到应用程序中时，可以由“源函数”SourceFunction分配和生成时间戳和watermark。SourceFunction可以产生并发送一个数据流；数据会与相关的时间戳一起发送出去，而watermark可以作为一条特殊数据在任何时间点发出。如果SourceFunction（暂时）不再发出watermark，它可以声明自己处于“空闲”（idle）状态。Flink会在后续算子的水位计算中，把空闲的SourceFunction产生的流分区排除掉。source的这一空闲机制，可以用来解决前面提到的水位不再上升的问题。源函数（Source Function）在“实现自定义源函数”一节中进行了更详细的讨论。

2. 定期分配：在Flink中，DataStream API提供一个名为AssignerWithPeriodicWatermarks的用户定义函数，它可以从每个数据中提取时间戳，并被定期调用以生成当前watermark。提取出的时间戳被分配给相应的数据，而生成的watermark也会添加到流中。这个函数将在“分配时间戳和生成水位线”一节中讨论。

3. 间断分配：AssignerWithPunctuatedWatermarks是另一个用户定义的函数，它同样会从每个数据中提取一个时间戳。它可以用于生成特殊输入数据中的watermark。与AssignerWithPeriodicWatermarks相比，此函数可以（但不是必须）从每个记录中提取watermark。我们在“分配时间戳和生成水位线”一节中同样讨论了该函数。

用户定义的时间戳分配函数并没有严格的限制，通常会放在尽可能靠近source算子的位置，因为当经过一些算子处理后，数据及其时间戳的顺序就更加难以解释了。所以尽管我们可以在流应用程序的中段覆盖已有的时间戳和watermark——Flink通过用户定义的函数提供了这种灵活性，但这显然并不是推荐的做法。

## 3.4 状态管理

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

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

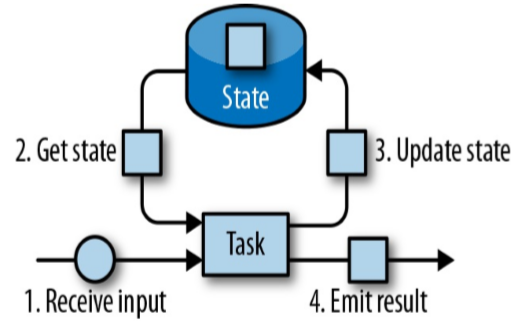


图3-10 有状态的流处理任务

任务会接收一些输入数据。在处理数据时，任务可以读取和更新状态，并根据输入数据和状态计算结果。最简单的例子，就是统计接收到多少条数据的任务。当任务收到新数据时，它会访问状态以获取当前的计数，然后让计数递增，更新状态并发送出新的计数。

应用程序里，读取和写入状态的逻辑一般都很简单直接，而有效可靠的状态管理会复杂一些。这包括如何处理很大的状态——可能会超过内存，并且保证在发生故障时不会丢失任何状态。幸运的是，Flink会帮我们处理这相关的所有问题，包括状态一致性、故障处理以及高效存储和访问，以便开发人员可以专注于应用程序的逻辑。

在Flink中，状态始终与特定算子相关联。为了使运行时的Flink了解算子的状态，算子需要预先注册其状态。总的说来，有两种类型的状态：算子状态（operator state）和键控状态（keyed state），它们有着不同的范围访问，我们将在下面展开讨论。

### 算子状态

算子状态的作用范围限定为算子任务。这意味着由同一并行任务所处理的所有数据都可以访问到相同的状态，状态对于同一任务而言是共享的。算子状态不能由相同或不同算子的另一个任务访问。图3-11显示了任务如何访问算子状态。

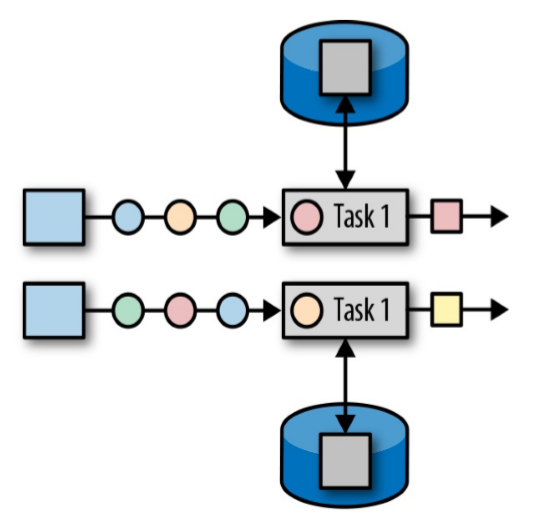


图3-11 具有算子状态的任务

Flink为算子状态提供三种基本数据结构：

* 列表状态（List state）

将状态表示为一组数据的列表。

* 联合列表状态（Union list state）

也将状态表示为数据的列表。它与常规列表状态的区别在于，在发生故障时，或者从保存点（savepoint）启动应用程序时如何恢复。我们将在后面继续讨论。

* 广播状态（Broadcast state）

如果一个算子有多项任务，而它的每项任务状态又都相同，那么这种特殊情况最适合应用广播状态。在保存检查点和重新调整算子并行度时，会用到这个特性。这两部分内容将在本章后面讨论。

### 键控状态（Keyed State）

顾名思义，键控状态是根据输入数据流中定义的键（key）来维护和访问的。Flink为每个键值维护一个状态实例，并将具有相同键的所有数据，都分区到同一个算子任务中，这个任务会维护和处理这个key对应的状态。当任务处理一条数据时，它会自动将状态的访问范围限定为当前数据的key。因此，具有相同key的所有数据都会访问相同的状态。图3-12显示了任务如何与键控状态进行交互。

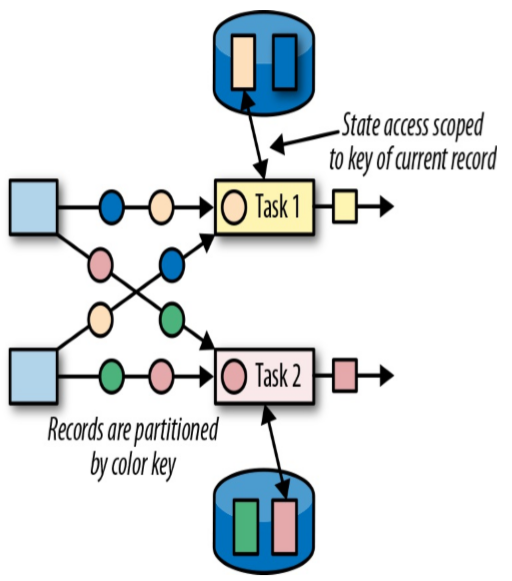


图3-12 具有键控状态的任务

我们可以将键控状态看成是在算子所有并行任务上，对键进行分区（或分片）之后的一个键值映射（key-value map）。 Flink为键控状态提供不同的数据结构，用于确定map中每个key存储的值的类型。我们简单了解一下最常见的键控状态。

* 值状态（Value state）

为每个键存储一个任意类型的单个值。复杂数据结构也可以存储为值状态。

* 列表状态（List state）

为每个键存储一个值的列表。列表里的每个数据可以是任意类型。

* 映射状态（Map state）

为每个键存储一个键值映射（map）。map的key和value可以是任意类型。

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

### 状态后端（State Backends）

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

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

状态检查点的写入也非常重要，这是因为Flink是一个分布式系统，而状态只能在本地维护。 TaskManager进程（所有任务在其上运行）可能在任何时间点挂掉。因此，它的本地存储只能被认为是不稳定的。状态后端负责将任务的状态检查点写入远程的持久存储。写入检查点的远程存储可以是分布式文件系统，也可以是数据库。不同的状态后端在状态检查点的写入机制方面有所不同。例如，RocksDB状态后端支持增量的检查点，这对于非常大的状态来说，可以显著减少状态检查点写入的开销。

我们将在“选择状态后端”一节中更详细地讨论不同的状态后端及其优缺点。

### 调整有状态算子的并行度

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

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

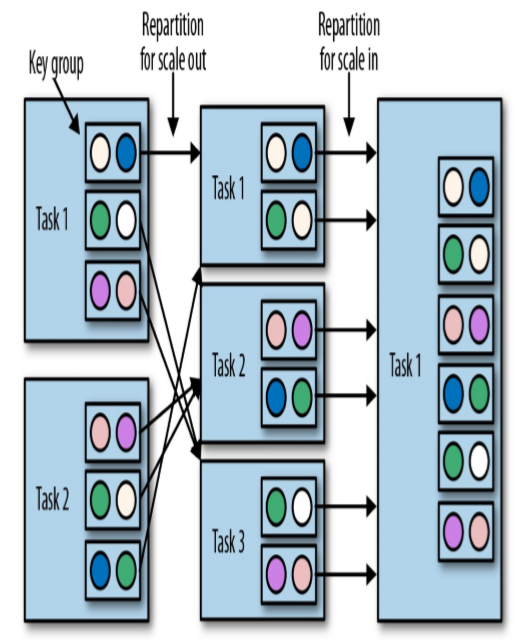


图3-13 具有键控状态的算子并行度的缩放

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

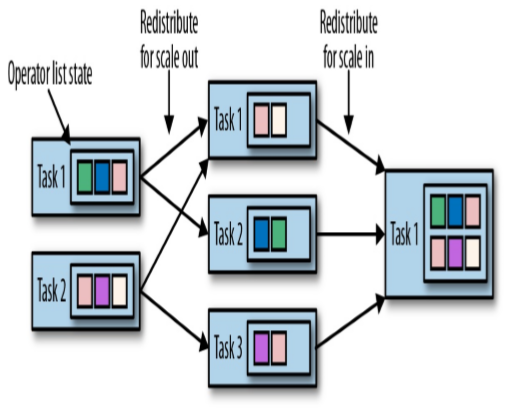


图3-14 具有算子列表状态的算子并行度的缩放

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

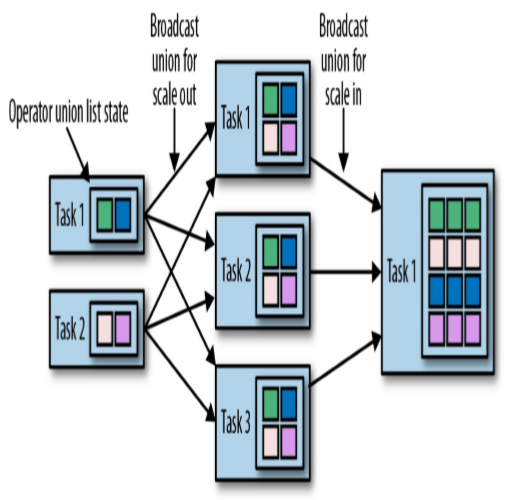


图3-15 具有算子联合列表状态的算子并行度的缩放

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

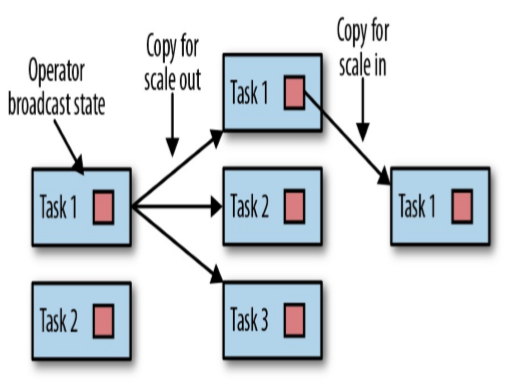


图3-16 具有广播状态的算子并行度的缩放

## 3.5 检查点，保存点和状态恢复

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

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

### 一致的检查点（Checkpoints）

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

1. 暂停所有输入流的摄取，也就是不再接收新数据的输入。

2. 等待所有正在处理的数据计算完毕，这意味着结束时，所有任务都已经处理了所有输入数据。

3. 通过将每个任务的状态复制到远程持久存储，来得到一个检查点。所有任务完成拷贝操作后，检查点就完成了。

4. 恢复所有输入流的摄取。

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

图3-17显示了一个简单应用中的一致检查点。

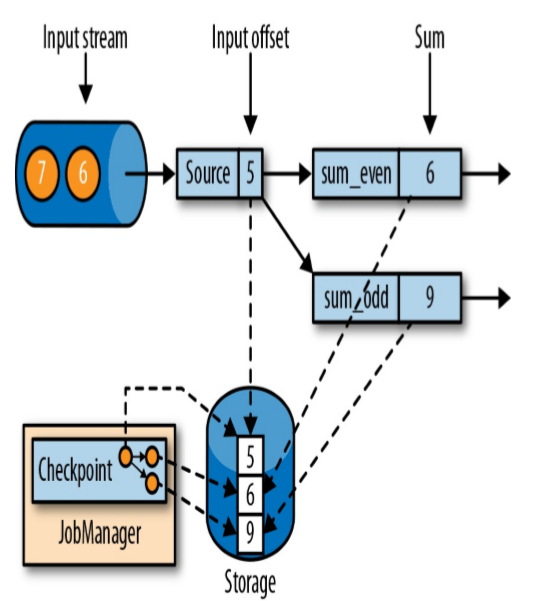


图3-17 流应用程序的一致检查点

上面的应用程序中具有单一的输入源（source）任务，输入数据就是一组不断增长的数字的流——1,2,3等。数字流被划分为偶数流和奇数流。求和算子（sum）的两个任务会分别实时计算当前所有偶数和奇数的总和。源任务会将其输入流的当前偏移量存储为状态，而求和任务则将当前的总和值存储为状态。在图3-17中，Flink在输入偏移量为5时，将检查点写入了远程存储，当前的总和为6和9。

### 从一致检查点中恢复状态

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

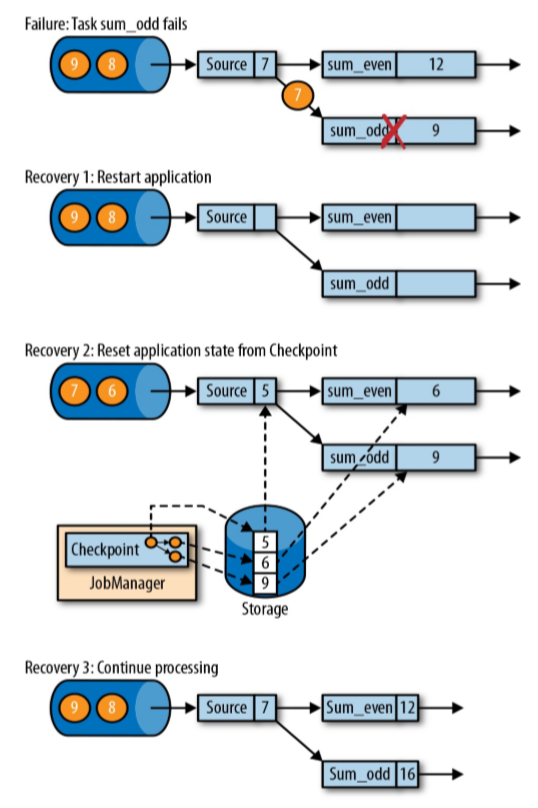


图3-18 从检查点恢复应用程序

应用程序从检查点的恢复分为三步：

1. 重新启动整个应用程序。

2. 将所有的有状态任务的状态重置为最近一次的检查点。

3. 恢复所有任务的处理。

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

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

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

### Flink的检查点算法

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

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

我们用一个简单的流应用程序作为示例，来一步一步解释这个算法。该应用程序有两个源（source）任务，每个任务都消费一个增长的数字流。源任务的输出被划分为两部分：偶数和奇数的流。每个分区由一个任务处理，该任务计算所有收到的数字的总和，并将更新的总和转发给输出（sink）任务。这个应用程序的结构如图3-19所示。

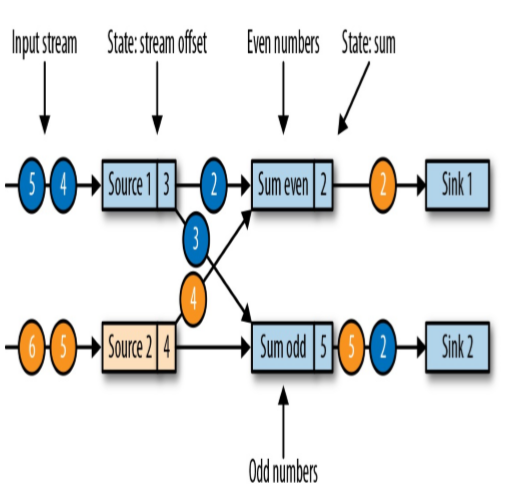


图3-19 具有两个有状态的源、有状态的算子任务和无状态输出的流应用程序

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

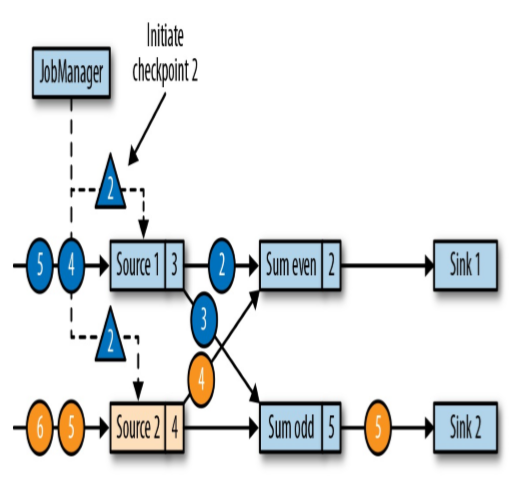


图3-20 JobManager通过向所有source发送信息，初始化一个检查点

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

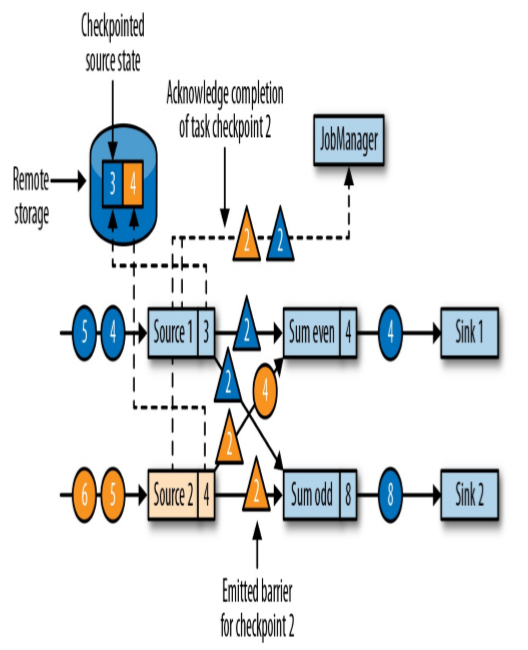


图3-21 数据源将它们的状态写入检查点，并发出一个检查点分界线

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

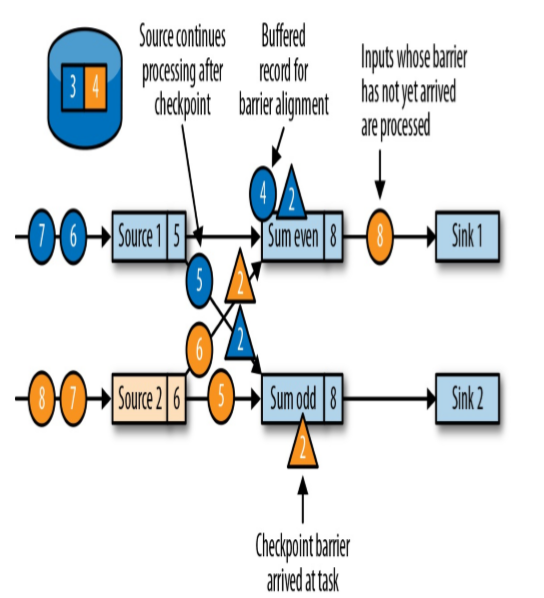


图3-22 任务会等待所有输入分区的barrier到达；

对于barrier已经到达的分区，继续到达的数据会被缓存；

而barrier尚未到达的分区，数据会被正常处理

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

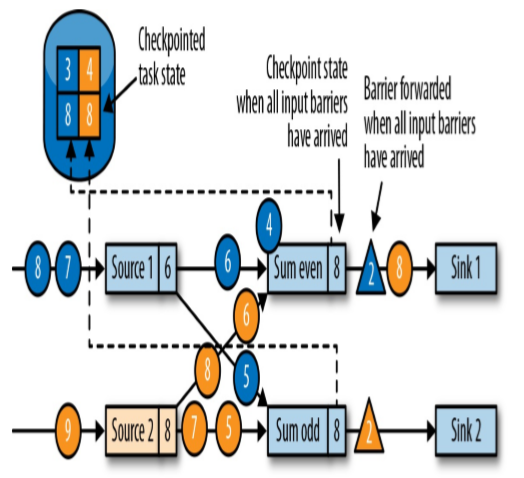


图3-23 当收到所有输入分区的barrier时，任务将其状态保存到检查点中，

然后将barrier继续向下游转发

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

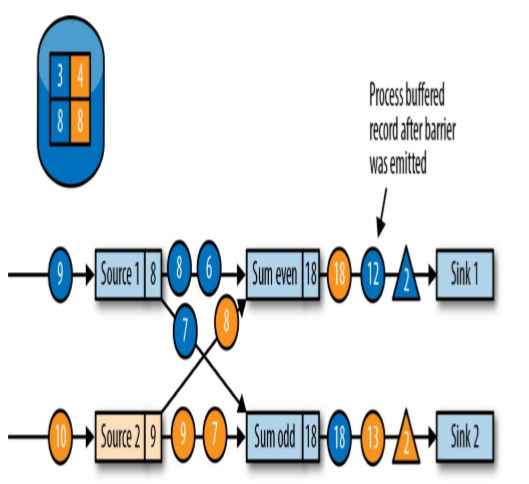


图3-24 向下游转发检查点分界线后，任务继续正常的流处理

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

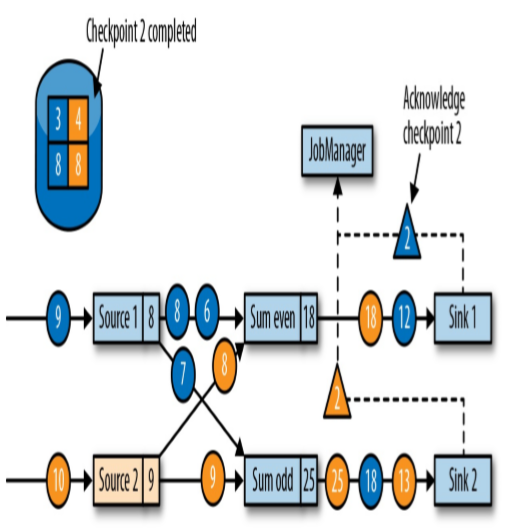


图3-25 Sink任务向JobManager确认收到检查点的分界线，

当所有任务都确认已成功将状态保存到检查点时，检查点就真正完成了

### 检查点的性能影响

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

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

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

### 保存点（Savepoints）

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

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

#### 使用保存点

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

* 可以从保存点启动不同但兼容的应用程序。这样一来，我们就可以及时修复应用程序中的逻辑bug，并让流式应用的源尽可能多地提供之前发生的事件，然后重新处理，以便修复之前的计算结果。修改后的应用程序还可用于运行A / B测试，或者具有不同业务逻辑的假设场景。这里要注意，应用程序和保存点必须兼容才可以这么做——也就是说，应用程序必须能够加载保存点的状态。
* 可以使用不同的并行度来启动相同的应用程序，可以将应用程序的并行度增大或减小。
* 可以在不同的集群上启动同样的应用程序。这非常有意义，意味着我们可以将应用程序迁移到较新的Flink版本或不同的集群上去。
* 可以使用保存点暂停应用程序，稍后再恢复。这样做的意义在于，可以为更高优先级的应用程序释放集群资源，或者在输入数据不连续生成时释放集群资源。
* 还可以将保存点设置为某一版本，并归档（archive）存储应用程序的状态。

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

#### 从保存点启动应用程序

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

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

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

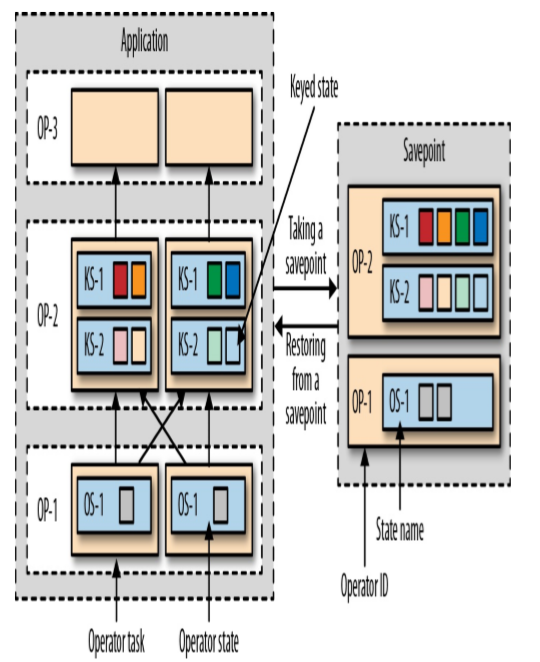


图3-26 从应用程序创建保存点，以及从保存点恢复应用程序

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

注：

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

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

## 3.6 总结

在本章中，我们讨论了Flink的高级架构，事件时间处理模式，状态管理和故障恢复机制。在以后我们设计高级流应用程序，配置集群，运行流应用程序以及分析器性能时，这些知识都会派上用场。