第 1 章

CHAPTER 1

Flink 架构与集群安装

现实世界中，许多系统数据是作为连续的事件流进行的，例如汽车 GPS 定位信号、金融交易记录、手机信号塔与智能手机用户之间的信号交换、网络流量、服务器日志、工业传感器和可穿戴设备的测量等。如果用户能够及时有效地大规模分析这些流数据，就能够更好地理解这些系统，并及时地进行分析。

实时数据分析一直是个热门话题，需要实时数据分析的场景也越来越多，如金融支付中的风控、基础运维中的监控告警、实时大盘等，此外，AI 模型也需要依据更为实时的聚合结果来达到很好的预测效果。

Apache Flink 是下一代开源大数据处理引擎。它是一个分布式大数据处理引擎，可对有限数据流和无限数据流进行有状态计算；可部署在各种集群环境，对各种大小的数据规模进行快速计算，如图 1-1 所示。

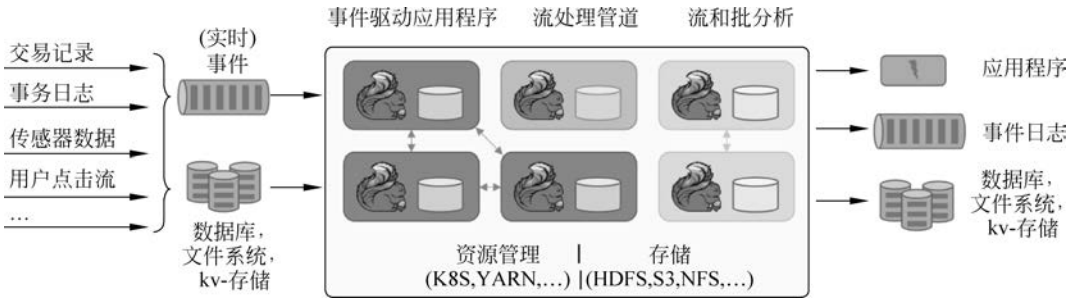

图 1-1　Apache Flink 大数据处理引擎

1.1　Flink 简介

29min

作为下一代开源大数据处理引擎，首先要了解它的发展历程的特性。

1.1.1 Flink 发展历程

Flink 起源于 Stratosphere 项目，这是 2010—2014 年由三所柏林大学和其他欧洲大学共同开展的一项研究项目。2014 年 4 月，Stratosphere 代码的一个分支被捐赠给了 Apache 软件基金会作为一个孵化项目，其初始提交者由系统的核心开发人员组成。此后不久，许多创始人离开大学，创办了一家名叫 Data Artisans 的公司，用于将 Flink 商业化。在孵化期间，为了防止与其他不相关的项目混淆，对项目名称进行了更改，选择 Flink 作为该项目的新名称。

注意：Data Artisans 公司于 2019 年 1 月被阿里巴巴以 9000 万欧元收购。

在德语中，Flink 一词的意思是快速或敏捷，它代表该项目所具有的流和批处理程序的风格。因为松鼠速度快、敏捷，所以 Flink 选择柏林郊外的一种红棕色松鼠作为 Logo。在图 1-2 中，左图为柏林郊外的红棕色松鼠，右图为 Flink 的 Logo。

图 1-2　Apache Flink 名称的由来和 Logo

项目快速完成孵化，2014 年 12 月，Flink 成为 Apache 软件基金会的顶级项目。

Flink 是 Apache 软件基金会最大的 5 个大数据项目之一，在全球拥有超过 200 名开发人员的社区。作为公认的新一代大数据计算引擎，Flink 已成为阿里巴巴、腾讯、滴滴、美团、字节跳动、Netflix、Lyft 等国内外知名公司建设流计算平台的首选。部分使用 Apache Flink 的企业如图 1-3 所示。

图 1-3　部分使用 Apache Flink 的成功企业

1.1.2 Flink 特性

Flink 支持流和批处理、复杂的状态管理、事件时间处理语义，以及对状态的一次一致

性保证。此外，Flink 可以部署在各种资源提供者（如 YARN、Apache Mesos 和 Kubernetes）上，也可以作为独立集群部署在裸机硬件上。可以将 Flink 集群配置为高可用的以避免单点故障。

Flink 设计用于在任何规模上运行有状态流应用程序。应用程序可能被并行化为数千个任务，这些任务分布在集群中并且并行执行，因此，一个应用程序可以利用几乎无限数量的 CPU、主内存、磁盘和网络 IO。此外，Flink 很容易维护非常大的应用程序状态。它的异步和增量检查点算法在保证精确一次性的状态一致性的同时，确保对处理时延的影响最小。

Apache Flink 为用户提供了更强大的计算能力和更易用的编程接口：

（1）批流统一。Flink 在 Runtime 和 SQL 层批流统一，提供高吞吐低延时计算能力和更强大的 SQL 支持。

（2）生态兼容。Flink 能与 Hadoop YARN/Apache Mesos/Kubernetes 集成，并且支持单机模式运行。

（3）性能卓越。Flink 提供了性能卓越的批处理与流处理支持。

（4）规模计算。Flink 的作业可被分解成上千个任务，分布在集群中并行执行。

Flink 已经被证明可以扩展到数千个内核和 TB 级的应用程序状态，提供高吞吐量和低时延，并支持世界上一些要求最高的流处理应用程序。例如，Apache Flink 在 2019 年阿里巴巴“双 11”场景中突破实时计算消息处理峰值达到 25 亿条/秒，2020 年“双 11”当时的实时计算峰值达到了破纪录的每秒 40 亿条记录，数据量也达到了惊人的每秒 7TB，相当于一秒需要读完 500 万本《新华字典》！随着 2020 年“双 11”阿里巴巴基于 Flink 实时计算场景的成功，毋庸置疑，Flink 将会加速成为大厂主流的数据处理框架，最终化身为下一代大数据处理标准。


2min

1.2 Flink 应用场景

Apache Flink 是一款非常适合做流批处理的计算框架，适用于以下场景：

（1）事件驱动类型，例如信用卡交易、刷单、监控等。

（2）数据分析类型，例如库存分析、“双 11”数据分析等。

（3）数据管道类型，也就是 ETL 场景，例如一些日志的解析等。

1.2.1 事件驱动应用程序

任何类型的数据都是作为事件流产生的。信用卡交易、传感器测量、机器日志、网站或移动应用程序上的用户交互，所有这些数据都以流的形式生成。

事件驱动的应用程序是一个有状态的应用程序，它从一个或多个事件流中摄取事件，并通过触发计算、状态更新或外部操作来响应传入的事件。

事件驱动的应用程序是传统应用程序（具有独立的计算层和数据存储层）设计的演化。

在这种传统体系结构中，应用程序从远程事务数据库读取数据并将数据持久存储。

而事件驱动的应用程序则基于有状态流来处理应用程序。在这种设计中，数据和计算是共存的，从而产生本地（内存或磁盘）数据访问。通过定期将检查点写入远程持久存储，可以实现容错。传统事务型应用程序和事件驱动应用程序体系结构之间的区别如图 1-4 所示。

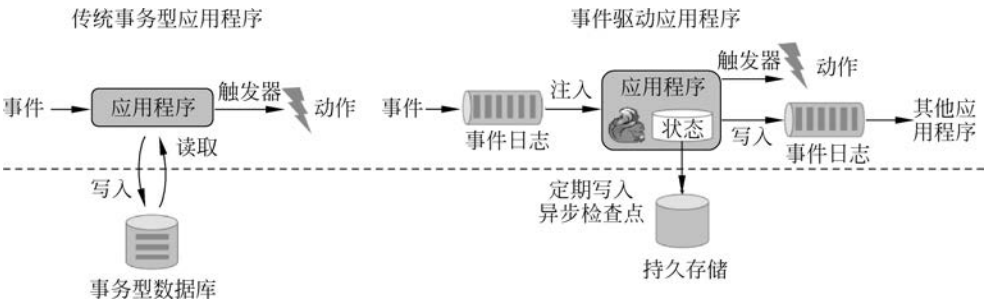

图 1-4　传统事务型应用程序和事件驱动应用程序体系结构之间的区别

事件驱动的应用程序访问本地数据，而不是查询远程数据库，从而在吞吐量和时延方面获得更好的性能。远程持久存储的定期检查点可以异步和增量地完成，因此，检查点对常规事件处理的影响非常小。事件驱动的应用程序设计提供的好处不仅是本地数据访问。在分层体系结构中，多个应用程序共享同一个数据库是很常见的，因此，需要协调数据库的任何更改，例如由于应用程序更新或扩展服务而更改数据布局。由于每个事件驱动的应用程序都负责自己的数据，因此对数据表示的更改或应用程序的扩展需要较少的协调。

事件驱动应用程序的限制由流处理器处理时间和状态的能力来定义，Flink 的许多突出特性都围绕这些概念。Flink 提供了一组丰富的状态原语，这些原语可以管理非常大的数据量（最多可达几 TB），并且具有严格的精确一次性的一致性保证。此外，Flink 支持事件时间、高度可定制的窗口逻辑及 ProcessFunction 提供的对时间的细粒度控制，从而支持高级业务逻辑的实现。此外，Flink 还提供了一个用于复杂事件处理（Complex Event Processing，CEP）的库，用于检测数据流中的模式。

对于事件驱动的应用程序，Flink 的突出特性是保存点（savepoint）。保存点是一个一致的状态镜像，可以用作兼容应用程序的起点。给定一个保存点，应用程序可以更新或调整其规模，或者可以启动应用程序的多个版本进行 A/B 测试。

典型的事件驱动程序包括：

（1）欺诈检测。

（2）异常检测。

（3）基于规则的提醒。

（4）业务流程监控。

（5）Web 应用程序（例如，社交网络）。

1.2.2 数据分析应用程序

分析工作从原始数据中提取信息。传统上，分析是作为对记录事件的有界数据集的批处理查询或应用程序执行的。为了将最新的数据合并到分析结果中，必须将其添加到待分析的数据集中，然后重新运行查询或应用程序。结果被写入存储系统或作为报告发出。

使用复杂的流处理引擎，还可以实时执行分析。流查询或应用程序不是读取有限的数据集，而是摄入实时事件流，并随着事件被消费而不断生成和更新结果。结果要么写入外部数据库，要么作为内部状态维护。例如，Dashboard 应用程序可以从外部数据库读取最新结果，也可以直接查询应用程序的内部状态。

Apache Flink 支持流及批处理分析应用程序，如图 1-5 所示。

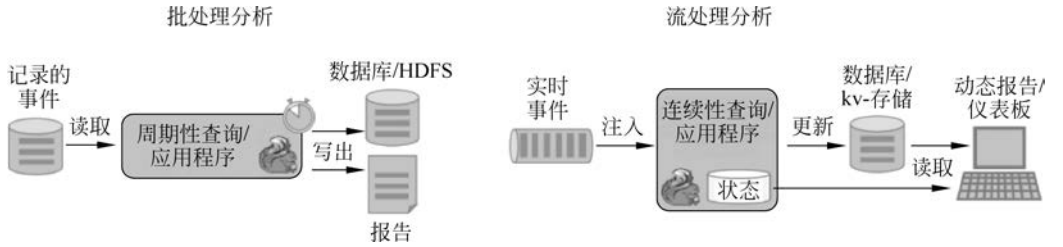

图 1-5 Apache Flink 支持流及批处理分析应用程序

与批处理分析相比，连续流分析的优势并不仅在于低时延（由于消除了周期性导入和查询执行，从事件到洞察的时延要低得多），另一个方面是更简单的应用程序架构。批处理分析管道由几个独立组件组成，用于定期调度数据摄入和查询执行。可靠地运行这样的管道并不简单，因为一个组件的故障会影响管道的后续步骤。相比之下，流分析应用程序运行在复杂的流处理器（如 Flink）上，它包含从数据摄取到连续结果计算的所有步骤，因此，它可以依靠引擎的故障恢复机制。

Flink 为连续流和批处理分析提供了非常好的支持。具体来讲，它具有一个符合 ANSI 的 SQL 接口，具有用于批处理和流查询的统一语义。无论 SQL 查询是在记录事件的静态数据集上运行，还是在实时事件流上运行，它们都会计算出相同的结果。对用户定义函数的丰富支持确保可以在 SQL 查询中执行定制代码。如果需要更多的定制逻辑，Flink 的 DataStream API 或 DataSet API 提供了更多的底层控制。此外，Flink 的 Gelly 库为批量数据集的大规模和高性能图分析提供了算法和构建块。

典型的数据分析应用程序包括：

（1）电信网络质量监控。

（2）移动应用程序中的产品更新及用户体验分析。

（3）消费者技术中实时数据的即时（Ad Hoc）分析。

（4）大规模图分析。

1.2.3 数据管道应用程序

提取-转换-加载（Extract-Transform-Load，ETL）是在存储系统之间转换和移动数据的常用方法。ETL 作业通常定期触发，以便将数据从事务型数据库系统复制到分析数据库或数据仓库。

数据管道的作用类似于 ETL 作业。它们转换和丰富数据，并能将数据从一个存储系统移动到另一个存储系统，然而，它们以连续流模式运行，而不是周期性地触发，因此，它们能够从不断产生数据的源读取记录，并以较低的时延将其移动到目标。例如，数据管道可以监视文件系统目录中的新文件，并将其数据写入事件日志。另一个应用程序可能将事件流物化到数据库，或者增量地构建和细化搜索索引。

Apache Flink 周期性 ETL 作业和连续数据管道之间的区别如图 1-6 所示。

图 1-6 Apache Flink 周期性 ETL 作业和连续数据管道之间的区别

与定期 ETL 作业相比，连续数据管道的明显优势是减少了将数据移动到其目的地的时延。此外，数据管道更加通用，可以用于更多的用例，因为它们能够持续地消费和发出数据。

Flink 的 SQL 接口（或 Table API）及其对用户定义函数（UDF）的支持可以解决许多常见的数据转换问题。使用更加通用的 DataStream API 可以实现具有更高要求的数据管道。Flink 为各种存储系统（如 Kafka、Kinesis、Elasticsearch 和 JDBC 数据库系统）提供了一组丰富的连接器。它还为文件系统提供了连续源，用于监视以时间间隔方式写文件的目录和接收器。

典型的数据管道应用程序包括：

（1）电子商务中的实时搜索索引构建。

（2）电子商务中的持续 ETL。

13min

1.3 Flink 体系架构

在大数据领域，有许多流计算框架，但是通常很难兼顾时延性和吞吐量。Apache Storm 支持低时延，但目前不支持高吞吐量，也不支持在发生故障时正确处理状态。Apache Spark Streaming 的微批处理方法实现了高吞吐量的容错性，但是难以实现真正的低延时和实时处理，并且表达能力方面也不是特别丰富；而 Apache Flink 兼顾了低时延和高吞吐量，是企业部署流计算时的首选。对 Storm、Spark Streaming 和 Flink 这 3 种流计算

框架的比较，见表 1-1。

表 1-1 3 种流计算框架比较

流处理框架	高吞吐量	低时延	易于使用和表达	正确的时间/窗口语义	压力下保持正确性
Storm	×	√	×	×	×
Spark Streaming	√	×	×	×	√
Flink	√	√	√	√	√

1.3.1 Flink 系统架构

Flink 可以运行在多种不同的环境中，例如，它可以通过单进程多线程的方式直接运行，从而提供调试的能力。它也可以运行在 YARN 或者 K8S 这种资源管理系统上，也可以在各种云环境中执行。

Flink 的整体系统架构如图 1-7 所示。

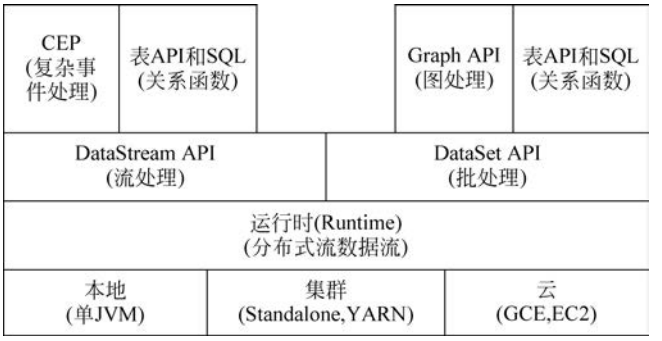

图 1-7 Apache Flink 的系统架构

针对不同的执行环境，Flink 提供了一套统一的分布式作业执行引擎，也就是 Flink 运行时层。Flink 在运行时层之上提供了 DataStream 和 DataSet 两套 API，分别用来编写流作业与批作业，以及一组更高级的 API 库来简化特定作业的编写（例如用于复杂事件处理的 CEP 库）。

1.3.2 Flink 运行时架构

Flink 运行时是 Flink 的核心计算结构，这是一个分布式系统，它接受流数据处理程序，并在一台或多台机器上以容错的方式执行这些数据流程序。这个运行时可以作为 YARN 的应用程序在集群中运行，也可以在 Mesos 集群中运行，或者在一台机器中运行（通常用于调试 Flink 应用程序）。

Flink 运行时层的整个架构采用了标准 Master-Slave 的结构，即总是由一个 Flink Master（JobManager）和一个或多个 Flink Slave（TaskManager）组成。Flink 运行时层的主

要架构如图 1-8 所示。

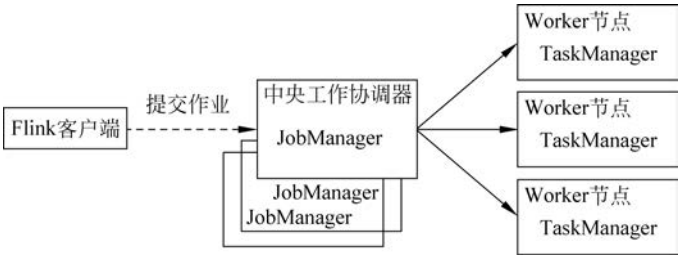

图 1-8　Apache Flink 运行时层的主要架构

图 1-8 展示了一个 Flink 集群的基本结构。在部署 Flink 时，每个组件通常有多个可用选项，见表 1-2。

表 1-2　Flink 程序部署选项

组件	作　　用	实　　现
Flink Client	将批处理或流应用程序编译成数据流图，然后提交给 JobManager	命令行接口 REST Endpoint SQL 客户端 Python REPL Scala REPL
JobManager	JobManager 是 Flink 的中心工作协调组件的名称。它有针对不同资源提供者的实现，这些提供者在高可用性、资源分配行为和支持的作业提交模式上有所不同。用于作业提交的 JobManager 模式有： Application Mode：仅为一个应用程序运行集群。作业的 main 方法（或 client）在 JobManager 上执行。支持在应用程序中多次调用 'execute'/'executeAsync'； Per-Job Mode：仅为一个作业运行集群。作业的 main 方法（或 client）仅在创建集群之前运行； Session Mode：一个 JobManager 实例管理共享同一个 TaskManager 集群的多个作业	Standalone Kubernetes YARN Mesos
TaskManager	TaskManager 是实际执行 Flink Job 工作的服务	

Flink 运行时由两种类型的进程组成：一个 JobManager 和一个或多个 TaskManager。客户端不是运行时和程序执行的一部分，而是用于准备数据流并将数据流发送到 JobManager。在此之后，客户端可以断开连接（分离模式），或者保持连接以接收进度报告（附加模式）。客户端可以作为触发执行的 Java/Scala 程序的一部分运行，也可以在命令行进程（./bin/flink run）中运行。

对 Flink 运行时架构更为详细的描述如图 1-9 所示。

JobManager 和 TaskManager 可以通过多种方式启动，例如，可以直接在机器上作为独

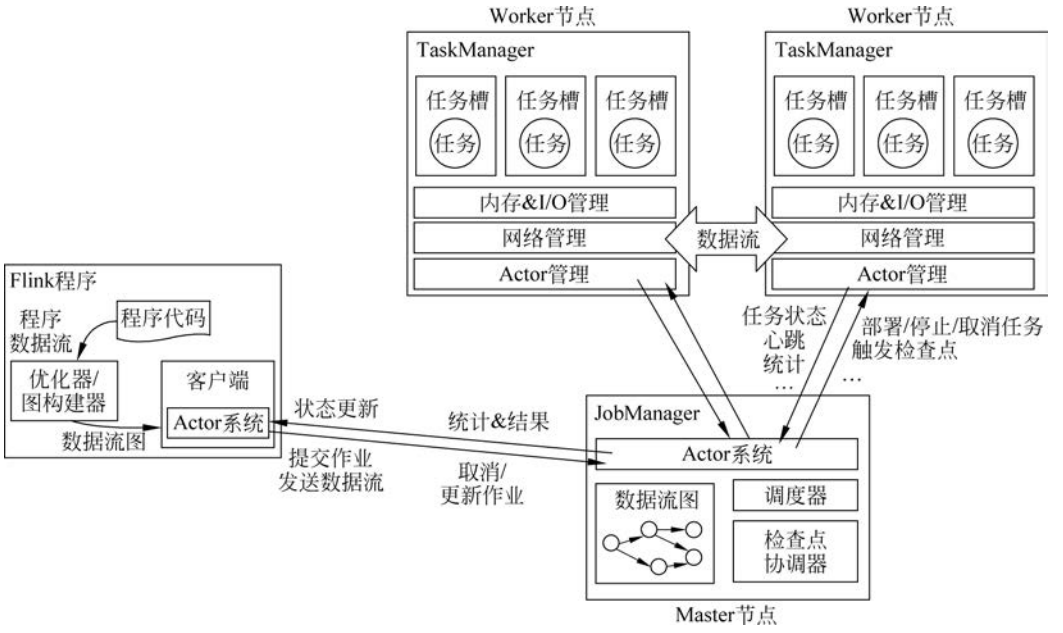

图 1-9 Apache Flink 运行时架构详细描述

立集群启动，也可以在容器中启动，或者由 YARN 或 Mesos 等资源框架管理。TaskManager 连接到 JobManager，宣布自己可用，并被分配工作。

1）JobManager

JobManager 有许多与协调 Flink 应用程序的分布式执行相关的职责：它决定什么时候安排下一个任务（或一组任务），对完成的任务或执行失败做出反应，协调检查点，协调故障恢复等。始终至少要有一个 JobManager。高可用性集群可能有多个 JobManager，其中一个始终是 leader，其他的都是 standby。

JobManager 是 Flink 集群的主进程，它由 3 个不同的组件组成：

（1）ResourceManager。ResourceManager 负责 Flink 集群中的资源分配和释放。它管理任务槽（Task Slots），任务槽是 Flink 集群中的资源调度单元。Flink 为不同的环境和资源提供者（如 YARN、Mesos、Kubernetes 和独立部署）实现了多个 ResourceManager。在 Standalone 独立集群中，ResourceManager 只能分发可用的 TaskManager 的任务槽，不能自己启动新的 TaskManager。

（2）Dispatcher。Dispatcher 提供了一个 REST 接口来提交 Flink 应用程序以供执行，并为每个提交的作业（通过 WebUI 或命令行）启动一个新的 JobMaster。它还运行 Flink WebUI 来提供关于作业执行的信息。

（3）JobMaster。JobMaster 负责管理单个 JobGraph 的执行。多个作业可以在一个 Flink 集群中同时运行，每个作业都有自己的 JobMaster。

2）TaskManager

TaskManagers是一个Flink集群的工作(worker)进程，负责执行数据流的任务，并缓冲和交换数据流。关于TaskManager，有以下几点要求：

(1) 必须始终至少有一个TaskManager。

(2) TaskManager中最小的资源调度单位是任务槽。

(3) TaskManager的任务槽数量表示并发处理的任务数。注意，多个操作符可以在一个任务槽中执行。

(4) TaskManager启动后，TaskManager将其槽位注册到ResourceManager。当得到ResourceManager的指示时，TaskManager会将一个或多个它的槽位提供给JobMaster。

(5) JobMaster可以将任务分配给这些槽以执行它们。

(6) 在执行过程中，一个TaskManager与运行同一应用程序任务的其他TaskManager交换数据。

1.3.3 Flink资源管理

Apache Flink是一个分布式系统，需要计算资源才能执行应用程序。Flink集成了所有常见的集群资源管理器，如Hadoop YARN、Apache Mesos和Kubernetes，不过也可以设置作为Standalone独立集群运行。

注意：在部署Flink应用程序时，Flink根据应用程序配置的并行性自动标识所需的资源，并从资源管理器中请求这些资源。如果发生故障，Flink则可通过请求新的资源来替换失败的容器。所有提交或控制应用程序的通信都是通过REST调用进行的，因此简化了Flink在许多环境中的集成。

在Flink中，资源是由TaskManager上的Slot来表示的，每个Slot可以用来执行不同的任务(Task)，而Job中实际的Task包含了待执行的用户逻辑代码。作业调度的主要目的就是给Task找到匹配的Slot。实际上，Flink作业调度可以看作对资源和任务进行匹配的过程。

注意：从逻辑上来讲，每个Slot都应该有一个向量来描述它所能提供的各种资源的量，每个Task也需要相应地说明它所需要的各种资源的量，但是实际上在Flink 1.9之前，Flink是不支持细粒度的资源描述的，而是统一地认为每个Slot提供的资源和Task需要的资源都是相同的。从Flink 1.9开始，Flink开始增加对细粒度资源匹配的支持和实现，但这部分功能目前仍在完善中。

---

1. Task执行

这是很难解释和理解的部分。在深入讲解之前，读者应该记住什么是Flink中的操作

符和任务。一般情况下，需要记住：Operator＝(Task1＋Task2＋…＋Task*N*)。

(1) Operator in Flink =>将一个数据流转换为另一个数据流(可以是相同类型的数据流，也可以是不同类型的数据流)。Operator(操作符)是逻辑数据流图(也称为 JobGraph)的节点。

(2) Task in Flink =>是由 Flink 的运行时执行的基本工作单元。任务是物理数据流图(也称为 Execution Graph)的节点。

(3) 任务是操作符或操作符链的一个并行实例(两个或多个连续操作符之间没有任何重分区)。

(4) 在 Apache Flink 中，每个 Task 只有一个线程，Task 之间没有共享知识。例如，Task1 不知道另一个 Task 正在发生什么。这意味着 API 可以从一个 Task 中访问每种状态段，但没有办法访问其他线程中的状态。

(5) 还有子任务(Sub-Task)的概念。子任务是在数据流的一部分上工作的任务。子任务指的是同一个操作符或操作符链有多个并行任务(这实际上意味着存在数据并行性)。

了解了术语操作符(Operator)、任务(Task)和子任务(Sub-Task)之后，接下来了解任务执行部分，包括：

(1) TaskManager(worker/slave 进程)可以同时执行多个任务，并发数通常与 TaskManager 所在机器的 CPU 数有关。例如，如果机器的 CPU 数是 16，则一个 TaskManager 可以同时运行 16 个任务。这是 Apache Flink 提供的最佳解决方案，可以同时运行 16 个任务，但这可能会导致一些问题，如 Flink 集群可能会经常重新启动自己等。

(2) 一个 TaskManager 在同一个 JVM 进程中以多线程方式执行它的任务。

(3) 任务可以是相同操作符(记住数据并行性)的子任务，也可以是不同操作符(记住任务并行性)的子任务，甚至可以是来自不同应用程序的子任务。

(4) TaskManager 提供一定数量的槽位(与机器的 CPU 数量相关)来控制它能够并发执行的任务。换句话说，如果机器有 16 个 CPU，则 TaskManager 可以有 16 个槽，每个槽用于处理一个任务。

注意：一个 Slot 槽可以包含一个特定的任务或多个关联的任务。

---

2. 操作符链

对于分布式执行，Flink 将操作子任务连成 Tasks。每个 Task 由一个线程执行。将操作符连接到 Tasks 中是一种优化行为，它减少了线程到线程切换和缓冲的开销，并在降低时延的同时增加了总体吞吐量。

例如，一个数据流使用 5 个子任务执行，因此使用 5 个并行线程，如图 1-10 所示。

操作符链允许非 shuffle 操作在同一个线程中共存，完全避免了序列化和反序列化。

3. 任务槽

每个 TaskManager(Worker 进程)都是一个 JVM 进程，可以在单独的线程中执行一个

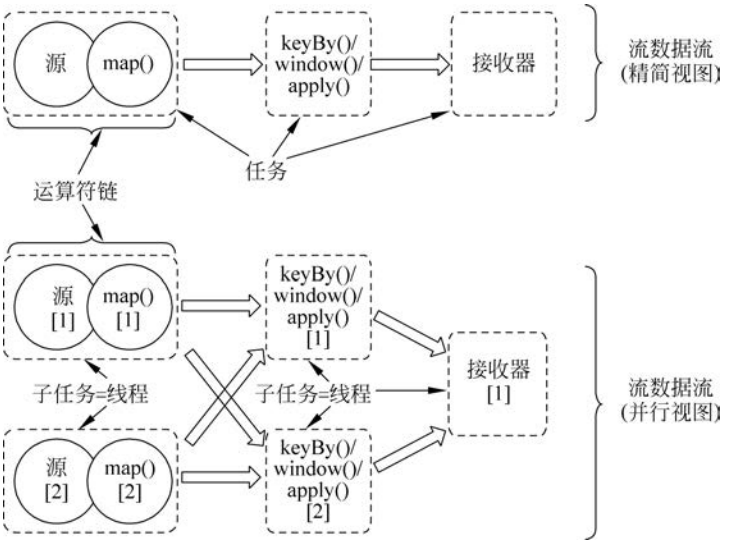

图 1-10　Apache Flink 使用操作符链来优化执行

或多个子任务。为了控制一个 worker 接受多少任务，一个 worker 具有至少一个“任务插槽”。

每个 Task Slot 表示 TaskManager 资源的一个固定子集。例如，一个 TaskManager 有 3 个插槽，它会将其 1/3 的托管内存分配给每个插槽。对资源进行插槽化意味着子任务不会与来自其他作业的子任务争夺托管内存，而是拥有一定数量的预留托管内存。注意，这里没有发生 CPU 隔离；当前插槽只分隔任务的托管内存，如图 1-11 所示。

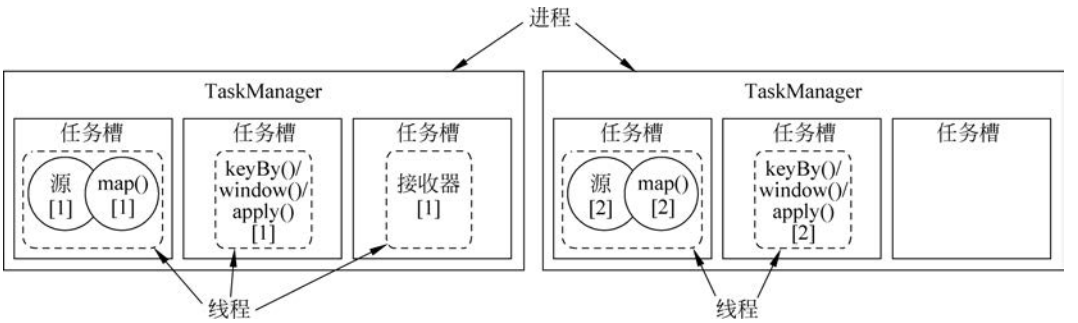

图 1-11　Apache Flink 任务槽

通过调整任务槽的数量，用户可以定义子任务如何彼此隔离。每个 TaskManager 有一个插槽(Slot)意味着每个任务组运行在各自的 JVM 中(例如，可以在单独的容器中启动 JVM)。拥有多个插槽意味着更多的子任务共享同一个 JVM。相同 JVM 中的任务共享 TCP 连接(通过多路复用)和心跳消息。它们还可以共享数据集和数据结构，从而减少每个任务的开销。

默认情况下，Flink 允许子任务共享槽位，即使它们是不同任务的子任务，只要它们来自相同的作业。结果是一个槽位可以容纳作业的整个管道，如图 1-12 所示。

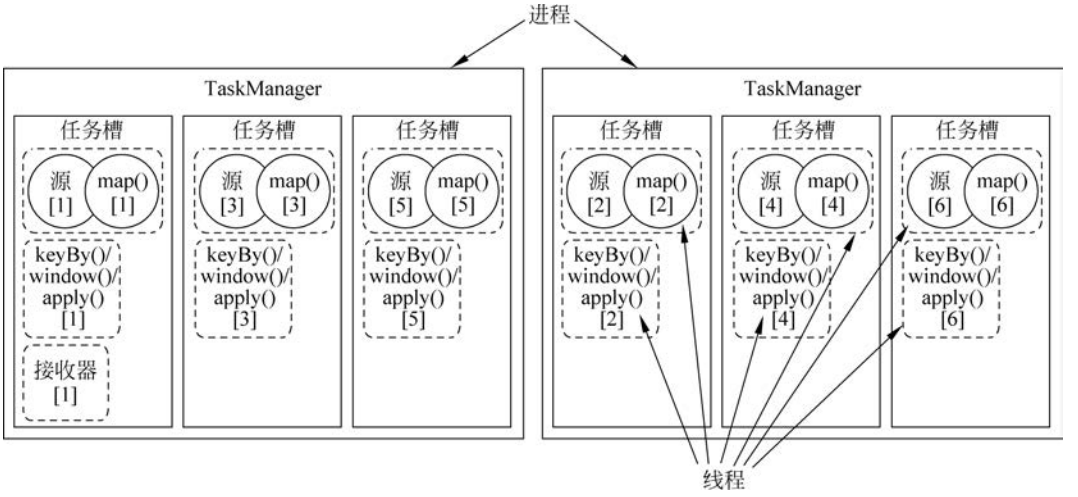

图 1-12 Apache Flink 允许子任务共享槽位

这种槽位共享有以下两个主要好处：

(1) Flink 集群需要的任务插槽与作业中使用的最高并行度一样多。不需要计算一个程序总共包含多少任务(具有不同的并行度)。

(2) 更容易得到更好的资源利用。如果没有槽位共享，则非密集型 source/map()子任务将阻塞与资源密集型窗口子任务一样多的资源。使用插槽共享，将示例中的基本并行度从 2 提高到 6，可以充分利用插槽资源，同时确保繁重的子任务在 TaskManager 中得到公平分配。

API 还包括一个资源组(Resource Group)机制，可用于防止不需要的槽位共享。

根据经验，恰当的默认任务槽位数应该是 CPU 核的数量。使用超线程，每个槽位将接受 2 个或更多的硬件线程上下文。

4. 资源申请

在 ResourceManager 中，有一个子组件叫作 SlotManager，SlotManager 用于维护当前集群中所有 TaskManager 上的 Slot 的信息与状态，例如该 Slot 在哪个 TaskManager 中，该 Slot 当前是否空闲等，如图 1-13 所示。

当 JobMaster 为特定 Task 申请资源时，根据当前作业部署模式(关于作业部署模式，可参阅 1.5 节)的区别，TaskManager 可能已经启动或者尚未启动。如果 TaskManager 尚未启动，则 ResourceManager 会去申请资源来启动新的 TaskManager。当 TaskManager 启动之后，它会通过服务找到当前活跃的 ResourceManager 并进行注册。在注册信息中，会包含该 TaskManager 中所有 Slot 的信息。ResourceManager 收到注册信息后，其中的 SlotManager 就会记录下相应的 Slot 信息。当 JobMaster 为某个 Task 来申请资源时，

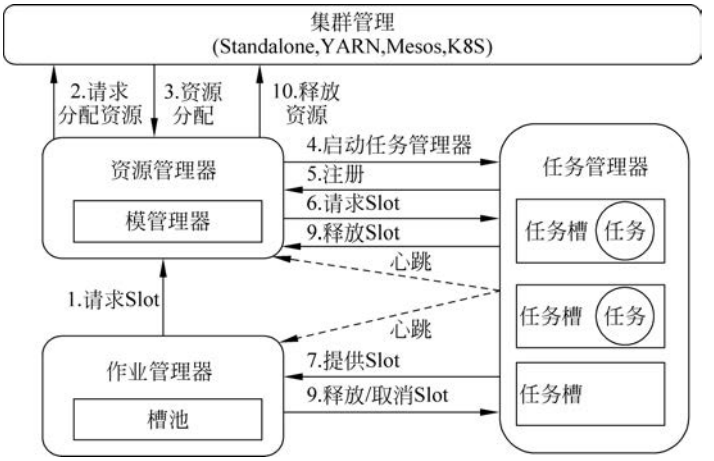

图 1-13　Apache Flink 资源申请流程

SlotManager 就会从当前空闲的 Slot 中按一定规则选择一个空闲的 Slot 进行分配。当分配完成后，ResourceManager 会首先向 TaskManager 发送 RPC 要求将选定的 Slot 分配给特定的 JobManager。TaskManager 如果还没有执行过该 JobMaster 的 Task，则它首先需要与相应的 JobMaster 建立连接，然后发送提供 Slot 的 RPC 请求。在 JobMaster 中，所有 Task 的请求会缓存到 SlotPool 中。当有 Slot 被提供之后，SlotPool 会从缓存的请求中选择相应的请求并结束相应的请求过程。

当 Task 结束之后，无论是正常结束还是异常结束，都会通知 JobMaster 相应的结束状态，然后在 TaskManager 端将 Slot 标记为已占用但未执行任务的状态。JobMaster 会首先将相应的 Slot 缓存到 SlotPool 中，但不会立即释放。这种方式避免了如果将 Slot 直接还给 ResourceManager，在任务异常结束之后需要重启时，则需要立刻重新申请 Slot 的问题。通过延时释放，容错的 Task 可以尽快调度回原来的 TaskManager，从而加快故障切换的速度。当 SlotPool 中缓存的 Slot 超过指定的时间仍未使用时，SlotPool 就会发起释放该 Slot 的过程。与申请 Slot 的过程对应，SlotPool 会首先通知 TaskManager 来释放该 Slot，然后 TaskManager 通知 ResourceManager 该 Slot 已经被释放，从而最终完成释放的逻辑。

5. 心跳报告

除了正常的通信逻辑外，在 ResourceManager 和 TaskManager 之间还存在定时的心跳消息同步 Slot 的状态。在分布式系统中，消息的丢失、错乱不可避免，这些问题会在分布式系统的组件中引入不一致状态，如果没有定时消息，则组件无法从这些不一致状态中恢复。此外，当组件之间长时间未收到对方的心跳时，就会认为对应的组件已经失效，并进入容错的流程。

6. 共享槽位

在 Slot 管理基础上，Flink 可以将 Task 调度到相应的 Slot 当中。如上文所述，Flink 尚

未完全引入细粒度的资源匹配，默认情况下，每个 Slot 可以分配给一个 Task，但是，这种方式在某些情况下会导致资源利用率不高，如图 1-14 所示，假如 A、B、C 依次执行计算逻辑，那么给 A、B、C 分配单独的 Slot 就会导致资源利用率不高。为了解决这一问题，Flink 提供了共享槽位（Share Slot）的机制，如图 1-14 所示。基于共享槽位，每个 Slot 中可以部署来自不同 JobVertex（作业向量）的多个任务，但是不能部署来自同一个 JobVertex 的 Task，如图中所示，每个 Slot 中最多可以部署同一个 A、B 或 C 的 Task，但是可以同时部署 A、B 和 C 的各一个 Task。当单个 Task 占用资源较少时，共享槽位可以提高资源利用率。此外，共享槽位也提供了一种简单的保持负载均衡的方式。

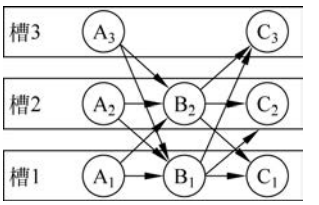

图 1-14 Apache Flink 共享槽位

1.3.4 Flink 作业调度

参与 Flink 程序执行的有多个进程，包括 JobManager、TaskManager 及 JobClient，其中 JobManager 充当 Master 角色，TaskManager 充当 Worker 角色。JobClient 不是 Flink 任务执行过程的内部组件，而是执行过程的起始点。JobClient 负责接收用户提交的应用程序，创建对应的数据流，然后将数据流提交到 JobManager 上执行。一旦执行完毕，JobClient 将执行结果发回给用户。

Flink 程序的执行过程如图 1-15 所示。

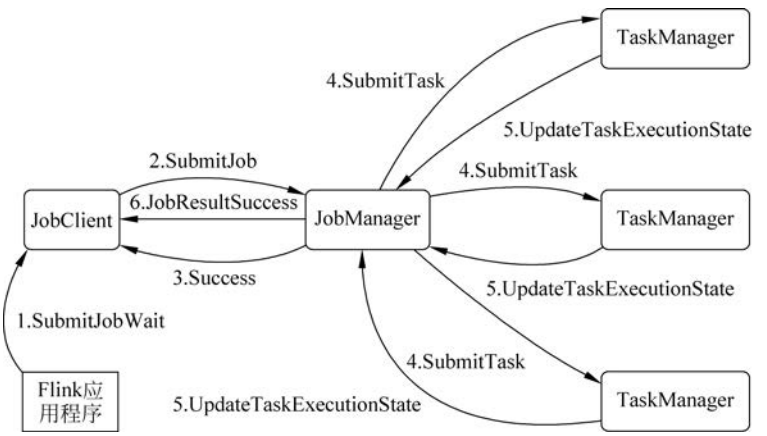

图 1-15 Apache Flink 程序的执行过程

Flink 应用程序首先被提交到 JobClient 上，随后 JobClient 将它提交到 JobManager 上，JobManager 负责安排资源的分配和作业的执行。首先是资源的分配，然后将作业划分为若干任务后提交到对应的 TaskManager 上。TaskManager 在接收到任务后，初始化一个线程并开始执行程序。执行过程中 TaskManager 持续地将状态的变化情况报告给

JobManager，这些状态包括开始执行（starting the execution）、正在执行（in progress）及完成（finished）。一旦作业的执行彻底完成，JobManager 就将结果发回 JobClient 端。

Flink 的分布式作业执行过程包含两个重要的角色：Master 和 Worker，如图 1-16 所示。

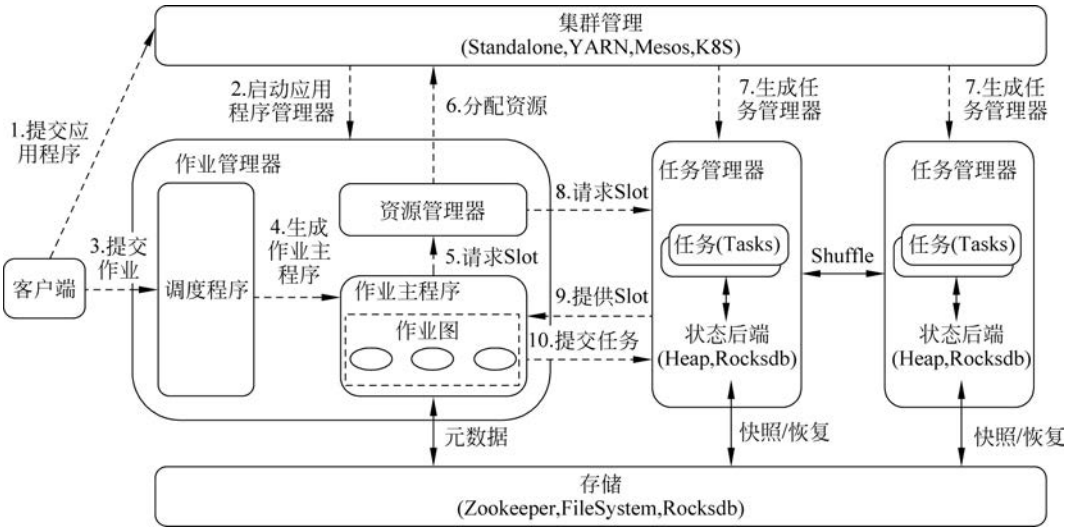

图 1-16　Apache Flink 分布式作业执行过程中的两个重要角色：Master 和 Worker

其中左侧的 JobManager 部分即是 Master，它负责管理整个集群中的资源并处理作业提交、作业监督；右侧的两个 TaskManager 则是 Worker，这是工作进程，负责提供具体的资源并实际执行作业。

当用户提交作业时，提交脚本会先启动一个 JobClient 进程负责作业的编译与提交。JobClient 首先将用户编写的代码编译为一个 JobGraph，在这个过程中，它还会进行一些检查或优化等工作，例如判断哪些运算符（算子）可以连接到同一个任务中，然后，JobClient 将产生的 JobGraph 提交到集群中执行。

当作业提交到 Dispatcher 后，Dispatcher 会首先启动一个 JobMaster 组件，然后 JobMaster 会向 ResourceManager 申请资源来启动作业中具体的任务。这时根据作业部署模式（关于作业部署模式，可参阅 1.3.6 节）的区别，TaskManager 可能已经启动或者尚未启动。

（1）如果是前者，此时 ResourceManager 中已有记录了 TaskManager 注册的资源，可以直接选取空闲资源进行分配。

（2）否则 ResourceManager 首先需要向外部资源管理系统申请资源来启动 TaskManager，然后等待 TaskManager 注册相应资源后再继续选择空闲资源进程分配。

目前 Flink 中 TaskManager 的资源是通过 Slot 来描述的，一个 Slot 一般可以执行一个具体的任务，但在一些情况下也可以执行多个相关联的任务。ResourceManager 选择到空

闲的 Slot 之后，就会通知相应的 TaskManager“将该 Slot 分配给 JobMaster XX”，然后 TaskManager 进行相应的记录后会向 JobManager 进行注册。JobManager 收到 TaskManager 注册的 Slot 后，就可以实际提交任务了。

TaskManager 收到 JobManager 提交的任务之后会启动一个新的线程来执行该任务。该任务启动后就会开始进行预先指定的计算，并通过数据 shuffle 模块互相交换数据。

基于 1.3.3 节中 Slot 管理和分配的逻辑，JobMaster 负责维护作业中任务执行的状态。如上文所述，客户端会向 JobMaster 提交一个 JobGraph，它代表了作业的逻辑结构。JobMaster 会根据 JobGraph 按并发展开，从而得到 JobMaster 中关键的 ExecutionGraph。ExecutionGraph 的结构如图 1-17 所示。

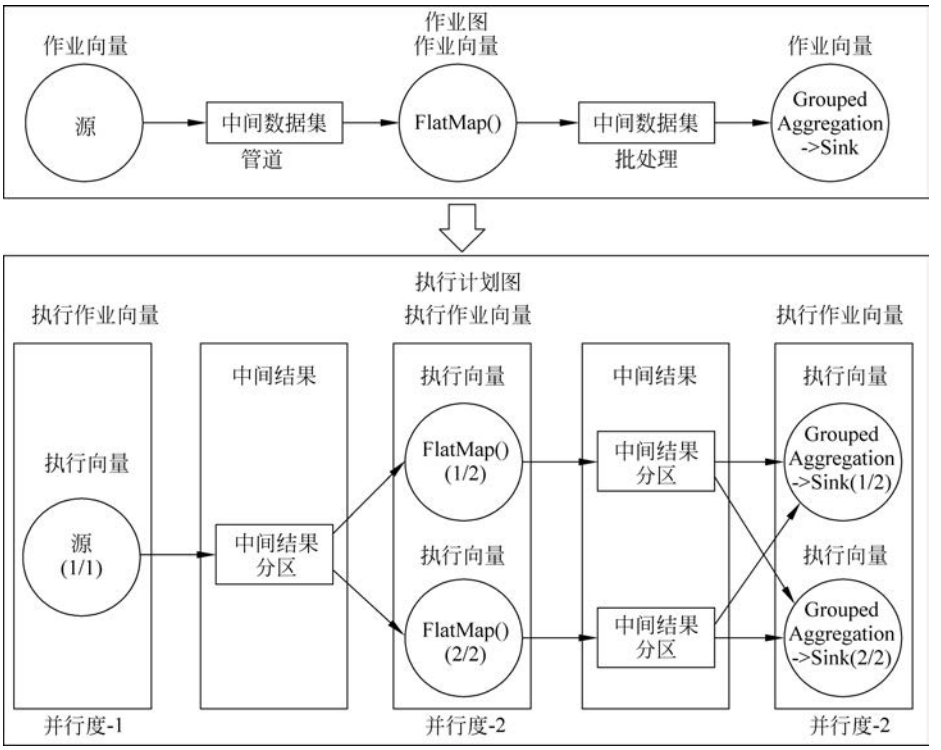

图 1-17 ExecutionGraph 是 JobMaster 中的核心数据结构

与 JobGraph 相比，ExecutionGraph 中对于每个 Task 和中间结果等均创建了对应的对象，从而可以维护这些实体的信息与状态。

在一个 Flink Job 中包含多个 Task，因此另一个关键的问题是在 Flink 中按什么顺序来调度这些 Task。目前 Flink 提供了两种基本的调度策略，即时延调度和即时调度，如图 1-18 所示。

即时调度会在作业启动时申请资源将所有的任务调度起来。这种调度算法主要用来调度可能没有终止的流作业。与之对应，时延调度则是从源开始，按拓扑顺序进行调度。简单

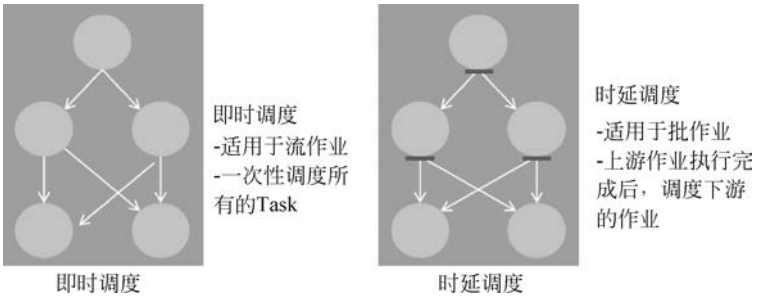

图 1-18　Flink 中两种基本的调度策略

来讲，时延调度会先调度没有上游任务的源任务，当这些任务执行完成时，它会将输出数据缓存到内存或者写入磁盘中，然后，对于后续的任务，当它的前驱任务全部执行完成后，Flink 就会将这些任务调度起来。这些任务会从读取上游缓存的输出数据进行自己的计算。这一过程继续进行直到所有的任务完成计算。

1.3.5　Flink 故障恢复

在 Flink 作业的执行过程中，除正常执行的流程外，还有可能由于环境等原因导致各种类型的错误。整体来讲，错误可分为两大类：Task 执行出现错误或 Flink 集群的 Master 出现错误。由于错误不可避免，为了提高可用性，Flink 需要提供自动错误恢复机制进行重试。

对于第一类 Task 执行错误，Flink 提供了多种不同的错误恢复策略。

（1）第一种错误恢复策略是 restart-all，即直接重启所有的 Task。对于 Flink 的流任务，由于 Flink 提供了 checkpoint 机制，因此当任务重启后可以直接从上次的 checkpoint 开始继续执行，因此这种方式更适合于流作业。

（2）第二种错误恢复策略是 restart-individual，它只适用于 Task 之间没有数据传输的情况。在这种情况下，可以直接重启出错的 Task。

由于 Flink 的批作业没有 checkpoint 机制，因此对于需要数据传输的作业，直接重启所有的 Task 会导致作业从头计算，从而导致一定的性能问题。为了增强对批作业的处理，Flink 在 1.9 版本中引入了一种新的基于分区的错误恢复策略。

在一个 Flink 的批作业中，Task 之间存在两种数据传输方式，一种是管道（pipeline）类型的方式，这种方式上下游 Task 之间直接通过网络传输数据，因此需要上下游同时运行；另外一种是 blocking 类型的方式，这种方式下，上游的 Task 会首先将数据进行缓存，因此上下游的 Task 可以单独执行。基于这两种类型的传输，Flink 将 ExecutionGraph 中使用管道方式传输数据的 Task 的子图叫作分区（region），从而将整个 ExecutionGraph 划分为多个子图。分区内的 Task 必须同时重启，而不同分区的 Task 由于在分区边界存在 blocking 的边，因此，可以单独重启下游分区中的 Task。

基于这一思路，如果某个分区中的某个 Task 执行出现错误，可以分两种情况进行

考虑。

(1) 如果是由于 Task 本身的问题发生错误,则可以只重启该 Task 所属的分区中的所有 Task,这些 Task 重启之后,可以直接拉取上游分区缓存的输出结果继续进行计算,这个过程如图 1-19 所示。

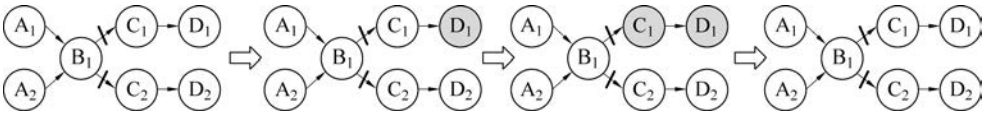

图 1-19 只重启出错 Task 所属的分区中的所有 Task

(2) 另一方面,如果错误是由于读取上游结果出现问题,如网络连接中断、缓存上游输出数据的 TaskManager 异常退出等,则还需要重启上游分区来重新产生相应的数据。在这种情况下,如果上游分区输出的数据分发方式不是确定性的(如 keyBy、broadcast 是确定性的分发方式,而 rebalance、random 则不是,因为每次执行会产生不同的分发结果),为了保证结果的正确性,还需要同时重启上游分区所有的下游分区,这个过程如图 1-20 所示。

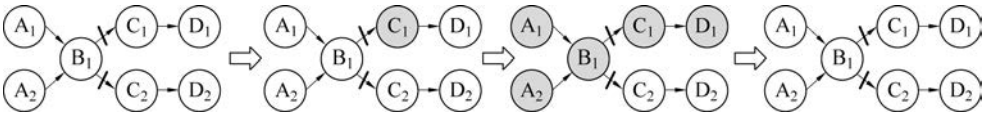

图 1-20 需要同时重启上游分区的故障恢复过程

另一类异常是 Flink 集群的 Master 进程发生异常。目前 Flink 支持启动多个 Master 作为备份,这些 Master 可以通过 ZooKeeper 进行选主 Master,从而保证某一时刻只有一个 Master 在运行。当前活跃的 Master 发生异常时,某个备份的 Master 可以接管协调的工作。为了保证 Master 可以准确维护作业的状态,Flink 目前采用了一种最简单的实现方式,即直接重启整个作业。实际上,由于作业本身可能仍在正常运行,因此这种方式存在一定的改进空间。

1.3.6 Flink 程序执行模式

Flink 可以通过以下 3 种方式来执行应用程序:①以 Application 模式;②以 Per-Job 模式;③以 Session 模式。这 3 种方式如图 1-21 所示。

以上 3 种模式的区别在于:

(1) 集群生命周期和资源隔离保证。

(2) 应用程序的 main()方法是在客户端上执行还是在集群上执行。

1. Per-Job 模式

为了提供更好的资源隔离保证,Per-Job 模式使用可用的资源提供者框架(例如 YARN、Kubernetes)为每个提交的作业启动一个集群。该集群仅对该作业可用。当作业完成时,将关闭集群,并清除所有滞留的资源(文件等)。

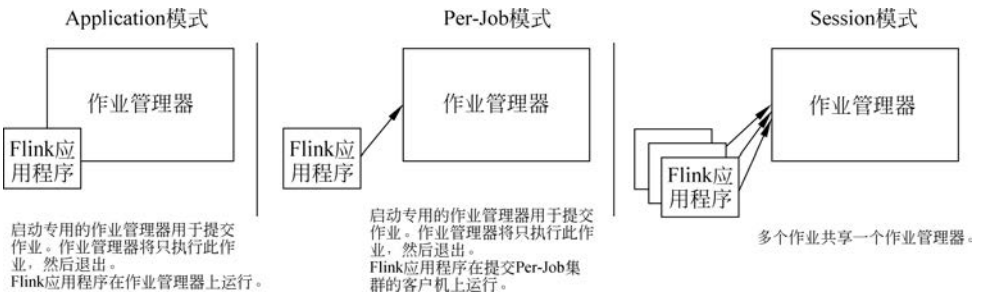

图 1-21　Flink 程序的 3 种执行模式

(1) 集群生命周期：在 Flink Job Cluster 中，可用的集群管理器(如 YARN)用于为每个提交的作业启动集群，该集群仅对该作业可用。在这里，客户端首先从集群管理器请求资源以启动 JobManager，并将作业提交给运行在此进程中的 Dispatcher，然后根据作业的资源需求惰性地分配 TaskManager。一旦作业完成，Flink 作业集群就会被拆除。集群的生命周期与作业的生命周期绑定在一起。

(2) 资源隔离：更好的隔离保证，因为资源不会跨作业共享。JobManager 中的致命错误只会影响 Flink Job Cluster 中运行的一个作业。此外，它将负载分散到多个 JobManager，因为每个作业有一个 JobManager。

由于 ResourceManager 需要申请并等待外部资源管理组件启动 TaskManager 进程并分配资源，所以 Flink Job Clusters 更适合长时间运行、对稳定性要求高、对启动时间不敏感的大型任务。Per-Job 资源分配模型是许多生产环境的首选模式(注：Kubernetes 和 Standalone 集群不支持此模式)。

采用这种模式的有 Flink On YARN，JobManager 不会预先启动，此时 Client 将首先向资源管理系统(如 YARN、K8S)申请资源来启动 JobManager，然后向 JobManager 中的 Dispatcher 提交作业。

例如，可以在 YARN 上使用 Per-Job 模式来部署并运行 Flink 自带的示例程序，命令如下：

```
$ /bin/flink run -t yarn-per-job
--detached ./examples/streaming/TopSpeedWindowing.jar
```

这将在 YARN 上启动一个 Flink 集群，然后在本地运行提供的应用程序 JAR，最后将 JobGraph 提交给 YARN 上的 JobManager。如果传递--detached 参数，则客户端将在提交被接受后停止。一旦作业停止，YARN 集群就会停止 Flink 集群。

一旦部署了 Per-Job Cluster，就可以与它进行交互了，以执行取消或获取保存点等操作。常用命令如下：

```
#列出集群上正在运行的作业
$ ./bin/flink list -t yarn-per-job
-Dyarn.application.id=application_XXXX_YY
```