作者是资深架构师和流式计算专家，第四范式AI项目架构师，曾就职于顺丰数据
从动机、原理、实战和调优4个维度循序渐进讲解利用Flink进行分布式流式应用开发，指导读者从零基础入门到进阶

技术丛书

Principle, Practice and Performance Optimization about Flink

Flink原理、实战与性能优化

张利兵◎著

机械工业出版社
China Machine Press

作者是资深架构师和流式计算专家，第四范式AI项目架构师，曾就职于明略数据

从功能、原理、实战和调优4个维度循序渐进讲解利用Flink进行分布式流式应用开发，指导读者从零基础入门到进阶

大数据技术丛书

Principle, Practice and Performance Optimization about Flink

Flink原理、实战与性能优化

张利兵◎著

机械工业出版社
China Machine Press

版权信息
COPYRIGHT INFORMATION

书名：Flink原理、实战与性能优化
作者：张利兵
排版：skip
出版社：机械工业出版社
出版时间：2019-04-01
ISBN：9787111623533

前言

为什么要写这本书

记得在几年前刚开始做流式计算相关的项目时，发觉项目对实时性和数据量的要求很高，无奈求助于Flink开源社区（后文简称“社区”），在社区中发现可以使用的流式框架有很多，例如比较主流的框架Apache Storm、Spark Streaming等，Apache Flink（简称“Flink”）也在其中。于是笔者开始对各种流式框架进行详细研究，最后发现能同时支持低延迟、高吞吐、Excactly-once的框架只有Apache Flink，从那时起笔者就对Flink这套框架充满兴趣，不管是其架构还是接口，都可以发现其中包含了非常优秀的设计思想。虽然当时Flink在社区的成熟度并不是很高，但笔者还是决定将Flink应用在自己的项目中，自此开启了Flink分布式计算技术应用之旅。

刚开始学习Flink，对于没有分布式处理技术和流式计算经验的人来说会相对比较困难，因为其很难理解有状态计算、数据一致性保障等概念。尤其在相关中文资源比较匮乏的情况下，需要用户在官网以及国外的技术网站中翻阅大量的外文资料，这在一定程度上对学习和应用Flink造成了阻碍。笔者在2018年参加了一场由Flink中文社区组织的线下交流活动，当时听了很多领域内专家将Flink应用在不同业务场景中的分享，发现Flink这项技术虽然优秀，但是国内尚未有一本能够全面介绍Flink的中文书籍，于是笔者决定结合自己的实际项目经验来完成一本Flink中文书籍，以帮助他人学习和使用Flink这项优秀的分布式处理技术。

阿里巴巴在2018年1月开源了其内部Flink的分支项目Blink，并推动社区将Blink中优秀的特性合并到Flink主干版本中，一时间Flink在国内的发展被推向了高潮，成为很多公司想去尝试使用的新技术。因此笔者相信未来会有更多的开发者参与到Flink社区中来，Flink也将在未来的大数据生态中占据举足轻重的位置。

读者对象

本书从多个方面对Flink进行了深入介绍，包括原理、多种抽象接口的使用，以及Flink的性能监控与调优等方面，因此本书比较适合以下类型的读者。

- 流计算开发工程师
- 大数据架构工程师
- 大数据开发工程师
- 数据挖掘工程师
- 高校研究生以及高年级本科生

如何阅读本书

本书共分为10章，各章节间具有一定的先后关系，对于刚入门的读者，建议从第1章开始循序渐进地学习。

对于有一定经验的读者可以自行选择章节开始学习。如果想使用Flink开发流式应用，则可以直接阅读第4章、第5章，以及第7章之后的内容；如果想使用Flink开发批计算应用，则可以选择阅读第5章以及第7章之后的内容。

勘误和支持

除封面署名外，参加本书编写工作的还有：张再胜、尚越、程龙、姚远等。由于笔者水平有限，编写时间仓促，书中难免会出现一

些错误或者不准确的地方，恳请读者批评指正。由于Flink技术的参考资料相对较少，因此书中有些地方参考了Flink官方文档，读者也可以结合Flink官网来学习。书中的全部源文件可以从GitHub网站下载，地址为https://github.com/zhanglibing1990/learning-flink。同时笔者也会将相应的功能及时更新。如果你有更多宝贵的意见可以通过QQ群686656574或电子邮箱zhanglibing1990@126.com联系笔者，期待能够得到你们的真挚反馈。

致谢

在本书的写作过程中，得到了很多朋友及同事的帮助和支持，在此表示衷心感谢！

感谢我的女朋友，因为有你的支持，我才能坚持将本书顺利完成，谢谢你一直陪伴在我的身边，不断鼓励我前行。

感谢机械工业出版社华章公司的编辑杨福川和张锡鹏，在这半年多的时间中始终支持我的写作，你们的鼓励和帮助引导我顺利完成全部书稿。

谨以此书献给我最亲爱的家人，以及众多热爱Flink的朋友！

总结

本书最开始介绍Flink的发展历史，然后对Flink批数据和流数据的不同处理接口进行介绍，再对Flink的部署与实施、性能优化等方面进行全面讲解。经过系统完整地了解和学习Flink分布式处理技术之后，可以发现Flink有很多非常先进的概念，以及非常完善的接口设计，这些都能让用户更加有效地处理大数据，特别是流式数据处理。随着大数据技术的不断发展，Flink也在大数据的浪潮中奋勇前行。越来越多的用户也参与到Flink社区的开发中，尤其是近年来随着阿里巴巴的推进，Blink的开源在一定程度上推动了Flink在国内大规模的落地。相信在不久的将来，Flink会逐渐成为国内乃至全球不可或缺的分布式处理引擎，笔者也相信Flink在流式数据处理领域会有新的突破，能够改变

目前大部分基于批处理的模式，让分布式数据处理变得更加高效，使得数据处理成本不断降低。

张利兵
2019年

第1章

Apache Flink介绍

本章对Apache Flink从多个方面进行介绍，让读者对Flink这项分布式处理技术能够有初步的了解。1.1节主要介绍了Flink的由来及其发展历史，帮助读者从历史的角度了解Flink这项技术发展的过程。1.2节重点介绍了Flink能够支持的各种实际业务场景、Flink所具备的主要特性、Flink组成部分及其基本概念等内容，最后在1.4节中介绍了Flink的基本架构以及主要组成部分。

1.1 Apache Flink是什么

在当前数据量激增的时代，各种业务场景都有大量的业务数据产生，对于这些不断产生的数据应该如何进行有效的处理，成为当下大多数公司所面临的问题。随着雅虎对Hadoop的开源，越来越多的大数据处理技术开始涌入人们的视线，例如目前比较流行的大数据处理引擎Apache Spark，基本上已经取代了MapReduce成为当前大数据处理的标准。但随着数据的不断增长，新技术的不断发展，人们逐渐意识到对实时数据处理的重要性。相对于传统的数据处理模式，流式数据处理有着更高的处理效率和成本控制能力。Apache Flink就是近年来在开源社区不断发展的技术中的能够同时支持高吞吐、低延迟、高性能的分布式处理框架。

在2010年至2014年间，由柏林工业大学、柏林洪堡大学和哈索普拉特纳研究所联合发起名为“Stratosphere:Information Management on the Cloud”研究项目，该项目在当时的社区逐渐具有了一定的社区知名度。2014年4月，Stratosphere代码被贡献给Apache软件基金会，成为Apache基金会孵化器项目。初期参与该项目的核心成员均是Stratosphere曾经的核心成员，之后团队的大部分创始成员离开学校，共同创办了一家名叫Data Artisans的公司，其主要业务便是将Stratosphere，也就是之后的Flink实现商业化。在项目孵化期间，项目Stratosphere改名为Flink。Flink在德语中是快速和灵敏的意思，用来体现流式数据处理器速度快和灵活性强等特点，同时使用棕红色松鼠图案作为Flink项目的Logo，也是为了突出松鼠灵活快速的特点，由此，Flink正式进入社区开发者的视线。

2014年12月，该项目成为Apache软件基金会顶级项目，从2015年9月发布第一个稳定版本0.9，到目前撰写本书期间已经发布到1.7的版本，更多的社区开发成员逐步加入，现在Flink在全球范围内拥有350

多位开发人员，不断有新的特性发布。同时在全球范围内，越来越多的公司开始使用Flink，在国内比较出名的互联网公司如阿里巴巴、美团、滴滴等，都在大规模使用Flink作为企业的分布式大数据处理引擎。

Flink近年来逐步被人们所熟知，不仅是因为Flink提供同时支持高吞吐、低延迟和exactly-once语义的实时计算能力，同时Flink还提供了基于流式计算引擎处理批量数据的计算能力，真正意义上实现了批流统一，同时随着阿里对Blink的开源，极大地增强了Flink对批计算领域的支持。众多优秀的特性，使得Flink成为开源大数据数据处理框架中的一颗新星，随着国内社区不断推动，越来越多的国内公司开始选择使用Flink作为实时数据处理技术。在不久的将来，Flink也将会成为企业内部主流的数据处理框架，最终成为下一代大数据处理的标准。

1.2　数据架构的演变

近年来随着开源社区的发展，越来越多新的技术被开源，例如雅虎的Hadoop分布式计算框架、UC伯克利分校的Apache Spark等，而伴随着这些技术的发展，促使着企业数据架构的演进，从传统的关系型数据存储架构，逐步演化为分布式处理和存储的架构。

1.2.1 传统数据基础架构

如图1-1所示，传统单体数据架构（Monolithic Architecture）最大的特点便是集中式数据存储，企业内部可能有诸多的系统，例如Web业务系统、订单系统、CRM系统、ERP系统、监控系统等，这些系统的事务性数据主要基于集中式的关系性数据库（DBMS）实现存储，大多数将架构分为计算层和存储层。存储层负责企业内系统的数据访问，且具有最终数据一致性保障。这些数据反映了当前的业务状态，例如系统的订单交易量、网站的活跃用户数、每个用户的交易额变化等，所有的更新操作均需要借助于同一套数据库实现。

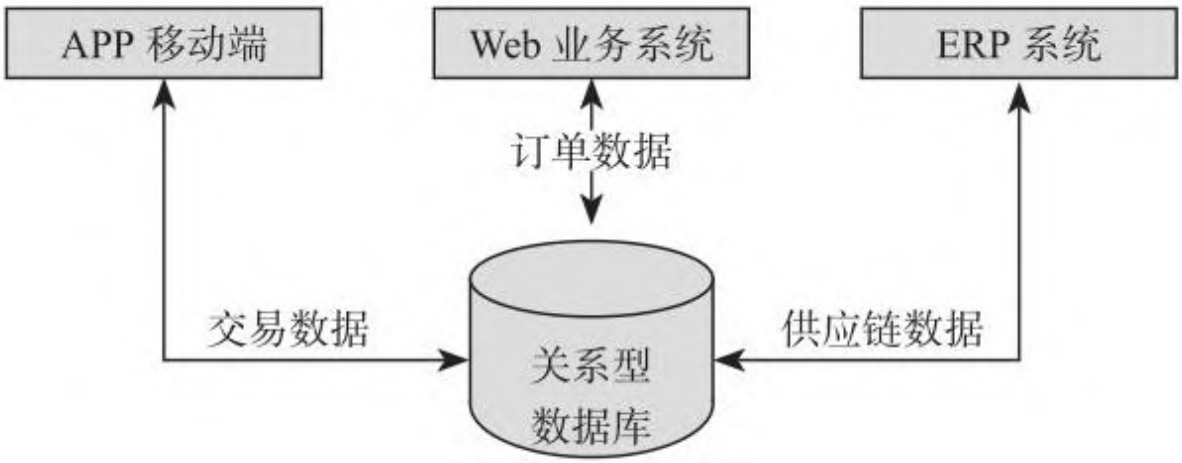

图1-1 传统数据结构

单体架构的初期效率很高，但是随着时间的推移，业务越来越多，系统逐渐变得很大，越来越难以维护和升级，数据库是唯一的准确数据源，每个应用都需要访问数据库来获取对应的数据，如果数据库发生改变或者出现问题，则将对整个业务系统产生影响。

后来随着微服务架构（Microservices Architecture）的出现，企业开始逐渐采用微服务作为企业业务系统的架构体系。微服务架构的核心思想是，一个应用是由多个小的、相互独立的微服务组成，这些服务运行在自己的进程中，开发和发布都没有依赖。不同的服务能依据不同的业务需求，构建的不同的技术架构之上，能够聚焦在有限的业务功能。

如图1-2所示，微服务架构将系统拆解成不同的独立服务模块，每个模块分别使用各自独立的数据库，这种模式解决了业务系统拓展的问题，但是也带来了新的问题，那就是业务交易数据过于分散在不同的系统中，很难将数据进行集中化管理，对于企业内部进行数据分析或者数据挖掘之类的应用，则需要通过从不同的数据库中进行数据抽取，将数据从数据库中周期性地同步到数据仓库中，然后在数据仓库中进行数据的抽取、转换、加载（ETL），从而构建成不同的数据集市和应用，提供给业务系统使用。

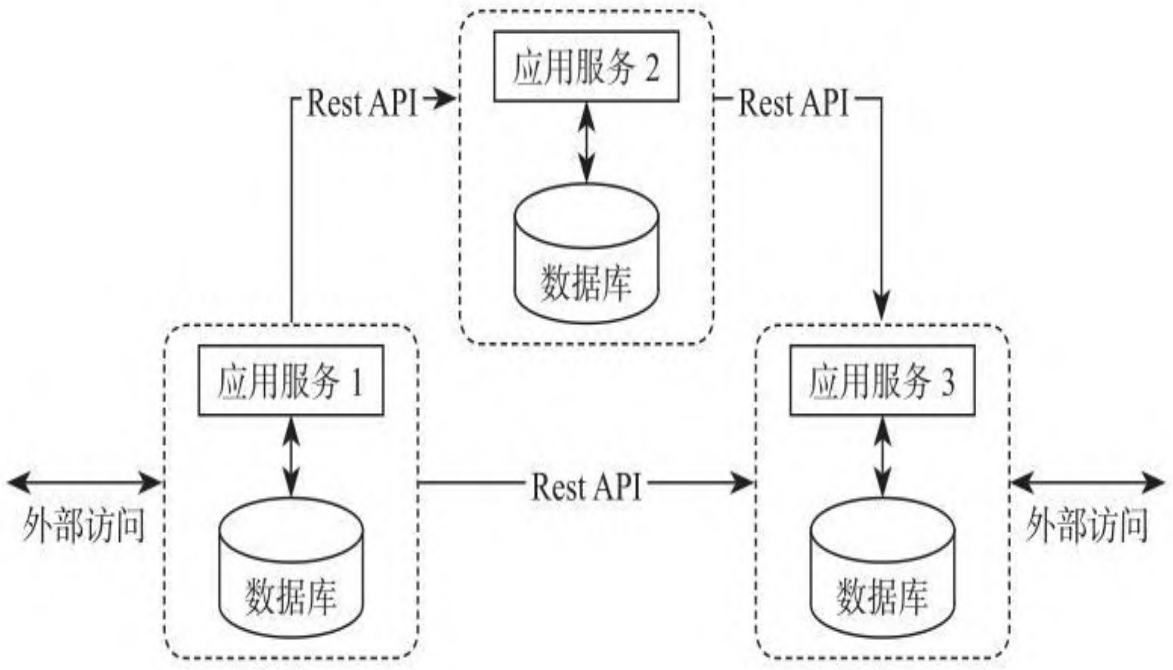

图1-2　微服务架构

1.2.2 大数据数据架构

起初数据仓库主要还是构建在关系型数据库之上，例如Oracle、Mysql等数据库，但是随着企业数据量的增长，关系型数据库已经无法支撑大规模数据集的存储和分析，因此越来越多的企业开始选择基于Hadoop构建企业级大数据平台。同时众多Sql-On-Hadoop技术方案的提出，也让企业在Hadoop上构建不同类型的数据应用变得简单而高效，例如通过使用Apache Hive进行数据ETL处理，通过使用Apache Impala进行实时交互性查询等。

大数据技术的兴起，让企业能够更加灵活高效地使用自己的业务数据，从数据中提取出更多重要的价值，并将数据分析和挖掘出来的结果应用在企业的决策、营销、管理等应用领域。但不可避免的是，随着越来越多新技术的引入与使用，企业内部一套大数据管理平台可能会借助众多开源技术组件实现。例如在构建企业数据仓库的过程中，数据往往都是周期性的从业务系统中同步到大数据平台，完成一系列ETL转换动作之后，最终形成数据集市等应用。但是对于一些时间要求比较高的应用，例如实时报表统计，则必须有非常低的延时展示统计结果，为此业界提出一套Lambda架构方案来处理不同类型的数据。例图1-3所示，大数据平台中包含批量计算的Batch Layer和实时计算的Speed Layer，通过在一套平台中将批计算和流计算整合在一起，例如使用Hadoop MapReduce进行批量数据的处理，使用Apache Storm进行实时数据的处理。这种架构在一定程度上解决了不同计算类型的问题，但是带来的问题是框架太多会导致平台复杂度过高、运维成本高等。在一套资源管理平台中管理不同类型的计算框架使用也是非常困难的事情。总而言之，Lambda架构是构建大数据应用程序的一种很有效的解决方案，但是还不是最完美的方案。

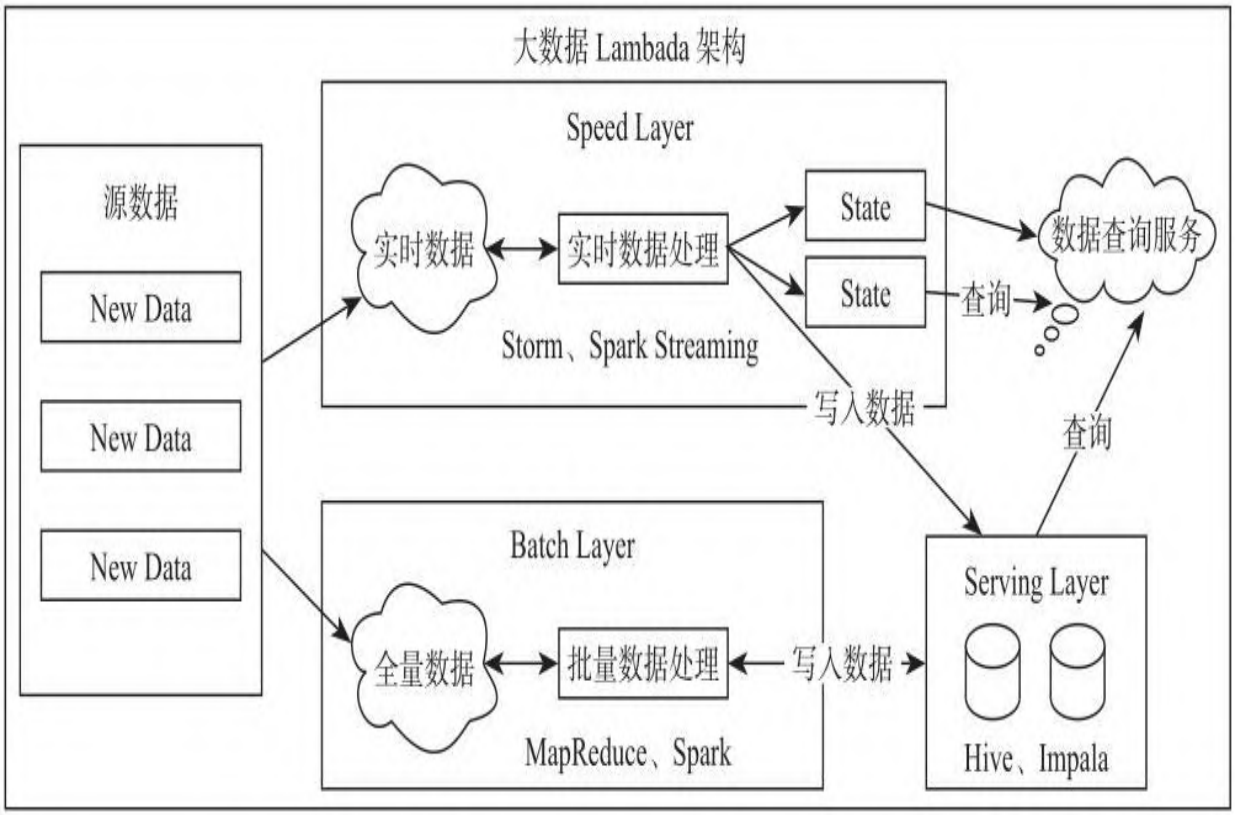

图1-3　大数据Lambada架构

后来随着Apache Spark的分布式内存处理框架的出现，提出了将数据切分成微批的处理模式进行流式数据处理，从而能够在一套计算框架内完成批量计算和流式计算。但因为Spark本身是基于批处理模式的原因，并不能完美且高效地处理原生的数据流，因此对流式计算支持的相对较弱，可以说Spark的出现本质上是在一定程度上对Hadoop架构进行了一定的升级和优化。

1.2.3 有状态流计算架构

数据产生的本质，其实是一条条真实存在的事件，前面提到的不同的架构其实都是在一定程度违背了这种本质，需要通过在一定时延的情况下对业务数据进行处理，然后得到基于业务数据统计的准确结果。实际上，基于流式计算技术局限性，我们很难在数据产生的过程中进行计算并直接产生统计结果，因为这不仅对系统有非常高的要求，还必须要满足高性能、高吞吐、低延时等众多目标。而有状态流计算架构（如图1-4所示）的提出，从一定程度上满足了企业的这种需求，企业基于实时的流式数据，维护所有计算过程的状态，所谓状态就是计算过程中产生的中间计算结果，每次计算新的数据进入到流式系统中都是基于中间状态结果的基础上进行运算，最终产生正确的统计结果。基于有状态计算的方式最大的优势是不需要将原始数据重新从外部存储中拿出来，从而进行全量计算，因为这种计算方式的代价可能是非常高的。从另一个角度讲，用户无须通过调度和协调各种批量计算工具，从数据仓库中获取数据统计结果，然后再落地存储，这些操作全部都可以基于流式计算完成，可以极大地减轻系统对其他框架的依赖，减少数据计算过程中的时间损耗以及硬件存储。

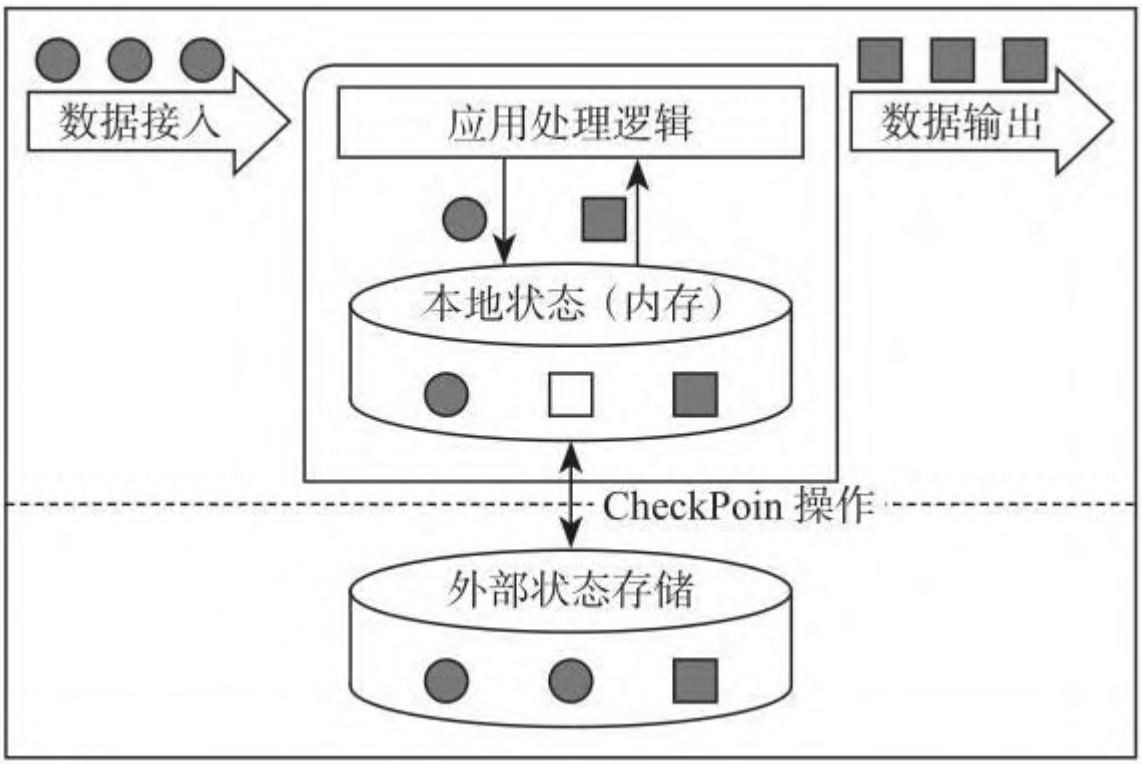

图1-4　有状态计算架构

如果计算的结果能保持一致，实时计算在很短的时间内统计出结果，批量计算则需要等待一定时间才能得出，相信大多数用户会更加倾向于选择使用有状态流进行大数据处理。

1.2.4　为什么会是Flink

可以看出有状态流计算将会逐步成为企业作为构建数据平台的架构模式，而目前从社区来看，能够满足的只有Apache Flink。Flink通过实现Google Dataflow流式计算模型实现了高吞吐、低延迟、高性能兼具实时流式计算框架。同时Flink支持高度容错的状态管理，防止状态在计算过程中因为系统异常而出现丢失，Flink周期性地通过分布式快照技术Checkpoints实现状态的持久化维护，使得即使在系统停机或者异常的情况下都能计算出正确的结果。

Flink具有先进的架构理念、诸多的优秀特性，以及完善的编程接口，而Flink也在每一次的Release版本中，不断推出新的特性，例如Queryable State功能的提出，容许用户通过远程的方式直接获取流式计算任务的状态信息，数据不需要落地数据库就能直接从Flink流式应用中查询。对于实时交互式的查询业务可以直接从Flink的状态中查询最新的结果。在未来，Flink将不仅作为实时流式处理的框架，更多的可能会成为一套实时的状态存储引擎，让更多的用户从有状态计算的技术中获益。

Flink的具体优势有以下几点。

（1）同时支持高吞吐、低延迟、高性能

Flink是目前开源社区中唯一一套集高吞吐、低延迟、高性能三者于一身的分布式流式数据处理框架。像Apache Spark也只能兼顾高吞吐和高性能特性，主要因为在Spark Streaming流式计算中无法做到低延迟保障；而流式计算框架Apache Storm只能支持低延迟和高性能特性，但是无法满足高吞吐的要求。而满足高吞吐、低延迟、高性能这三个目标对分布式流式计算框架来说是非常重要的。

（2）支持事件时间（Event Time）概念

在流式计算领域中，窗口计算的地位举足轻重，但目前大多数框架窗口计算采用的都是系统时间（Process Time），也是事件传输到计算框架处理时，系统主机的当前时间。Flink能够支持基于事件时间（Event Time）语义进行窗口计算，也就是使用事件产生的时间，这种基于事件驱动的机制使得事件即使乱序到达，流系统也能够计算出

精确的结果，保持了事件原本产生时的时序性，尽可能避免网络传输或硬件系统的影响。

（3）支持有状态计算

Flink在1.4版本中实现了状态管理，所谓状态就是在流式计算过程中将算子的中间结果数据保存在内存或者文件系统中，等下一个事件进入算子后可以从之前的状态中获取中间结果中计算当前的结果，从而无须每次都基于全部的原始数据来统计结果，这种方式极大地提升了系统的性能，并降低了数据计算过程的资源消耗。对于数据量大且运算逻辑非常复杂的流式计算场景，有状态计算发挥了非常重要的作用。

（4）支持高度灵活的窗口（Window）操作

在流处理应用中，数据是连续不断的，需要通过窗口的方式对流数据进行一定范围的聚合计算，例如统计在过去的1分钟内有多少用户点击某一网页，在这种情况下，我们必须定义一个窗口，用来收集最近一分钟内的数据，并对这个窗口内的数据进行再计算。Flink将窗口划分为基于Time、Count、Session，以及Data-driven等类型的窗口操作，窗口可以用灵活的触发条件定制化来达到对复杂的流传输模式的支持，用户可以定义不同的窗口触发机制来满足不同的需求。

（5）基于轻量级分布式快照（Snapshot）实现的容错

Flink能够分布式运行在上千个节点上，将一个大型计算任务的流程拆解成小的计算过程，然后将tesk分布到并行节点上进行处理。在任务执行过程中，能够自动发现事件处理过程中的错误而导致数据不一致的问题，比如：节点宕机、网路传输问题，或是由于用户因为升级或修复问题而导致计算服务重启等。在这些情况下，通过基于分布式快照技术的Checkpoints，将执行过程中的状态信息进行持久化存储，一旦任务出现异常停止，Flink就能够从Checkpoints中进行任务的自动恢复，以确保数据在处理过程中的一致性。

（6）基于JVM实现独立的内存管理

内存管理是所有计算框架需要重点考虑的部分，尤其对于计算量比较大的计算场景，数据在内存中该如何进行管理显得至关重要。针

对内存管理，Flink实现了自身管理内存的机制，尽可能减少JVM GC对系统的影响。另外，Flink通过序列化/反序列化方法将所有的数据对象转换成二进制在内存中存储，降低数据存储的大小的同时，能够更加有效地对内存空间进行利用，降低GC带来的性能下降或任务异常的风险，因此Flink较其他分布式处理的框架会显得更加稳定，不会因为JVM GC等问题而影响整个应用的运行。

（7）Save Points（保存点）

对于7*24小时运行的流式应用，数据源源不断地接入，在一段时间内应用的终止有可能导致数据的丢失或者计算结果的不准确，例如进行集群版本的升级、停机运维操作等操作。值得一提的是，Flink通过Save Points技术将任务执行的快照保存在存储介质上，当任务重启的时候可以直接从事先保存的Save Points恢复原有的计算状态，使得任务继续按照停机之前的状态运行，Save Points技术可以让用户更好地管理和运维实时流式应用。

1.3 Flink应用场景

在实际生产的过程中，大量数据在不断地产生，例如金融交易数据、互联网订单数据、GPS定位数据、传感器信号、移动终端产生的数据、通信信号数据等，以及我们熟悉的网络流量监控、服务器产生的日志数据，这些数据最大的共同点就是实时从不同的数据源中产生，然后再传输到下游的分析系统。针对这些数据类型主要包括实时智能推荐、复杂事件处理、实时欺诈检测、实时数仓与ETL类型、流数据分析类型、实时报表类型等实时业务场景，而Flink对于这些类型的场景都有着非常好的支持。

（1）实时智能推荐

智能推荐会根据用户历史的购买行为，通过推荐算法训练模型，预测用户未来可能会购买的物品。对个人来说，推荐系统起着信息过滤的作用，对Web/App服务端来说，推荐系统起着满足用户个性化需求，提升用户满意度的作用。推荐系统本身也在飞速发展，除了算法越来越完善，对时延的要求也越来越苛刻和实时化。利用Flink流计算帮助用户构建更加实时的智能推荐系统，对用户行为指标进行实时计算，对模型进行实时更新，对用户指标进行实时预测，并将预测的信息推送给Wep/App端，帮助用户获取想要的商品信息，另一方面也帮助企业提升销售额，创造更大的商业价值。

（2）复杂事件处理

对于复杂事件处理，比较常见的案例主要集中于工业领域，例如对车载传感器、机械设备等实时故障检测，这些业务类型通常数据量都非常大，且对数据处理的时效性要求非常高。通过利用Flink提供的CEP（复杂事件处理）进行事件模式的抽取，同时应用Flink的Sql进行

事件数据的转换，在流式系统中构建实时规则引擎，一旦事件触发报警规则，便立即将告警结果传输至下游通知系统，从而实现对设备故障快速预警监测，车辆状态监控等目的。

（3）实时欺诈检测

在金融领域的业务中，常常出现各种类型的欺诈行为，例如信用卡欺诈、信贷申请欺诈等，而如何保证用户和公司的资金安全，是来近年来许多金融公司及银行共同面对的挑战。随着不法分子欺诈手段的不断升级，传统的反欺诈手段已经不足以解决目前所面临的问题。以往可能需要几个小时才能通过交易数据计算出用户的行为指标，然后通过规则判别出具有欺诈行为嫌疑的用户，再进行案件调查处理，在这种情况下资金可能早已被不法分子转移，从而给企业和用户造成大量的经济损失。而运用Flink流式计算技术能够在毫秒内就完成对欺诈判断行为指标的计算，然后实时对交易流水进行规则判断或者模型预测，这样一旦检测出交易中存在欺诈嫌疑，则直接对交易进行实时拦截，避免因为处理不及时而导致的经济损失。

（4）实时数仓与ETL

结合离线数仓，通过利用流计算诸多优势和SQL灵活的加工能力，对流式数据进行实时清洗、归并、结构化处理，为离线数仓进行补充和优化。另一方面结合实时数据ETL处理能力，利用有状态流式计算技术，可以尽可能降低企业由于在离线数据计算过程中调度逻辑的复杂度，高效快速地处理企业需要的统计结果，帮助企业更好地应用实时数据所分析出来的结果。

（5）流数据分析

实时计算各类数据指标，并利用实时结果及时调整在线系统相关策略，在各类内容投放、无线智能推送领域有大量的应用。流式计算技术将数据分析场景实时化，帮助企业做到实时化分析Web应用或者App应用的各项指标，包括App版本分布情况、Crash检测和分布等，同时提供多维度用户行为分析，支持日志自主分析，助力开发者实现基于大数据技术的精细化运营、提升产品质量和体验、增强用户黏性。

（6）实时报表分析

实时报表分析是近年来很多公司采用的报表统计方案之一，其中最主要的应用便是实时大屏展示。利用流式计算实时得出的结果直接被推送到前端应用，实时显示出重要指标的变换情况。最典型的案例便是淘宝的双十一活动，每年双十一购物节，除疯狂购物外，最引人注目的就是天猫双十一大屏不停跳跃的成交总额。在整个计算链路中包括从天猫交易下单购买到数据采集、数据计算、数据校验，最终落到双十一大屏上展现的全链路时间压缩在5秒以内，顶峰计算性能高达数三十万笔订单/秒，通过多条链路流计算备份确保万无一失。而在其他行业，企业也在构建自己的实时报表系统，让企业能够依托于自身的业务数据，快速提取出更多的数据价值，从而更好地服务于企业运行过程中。

1.4 Flink基本架构

1.4.1 基本组件栈

在Flink整个软件架构体系中，同样遵循着分层的架构设计理念，在降低系统耦合度的同时，也为上层用户构建Flink应用提供了丰富且友好的接口。

从图1-5中可以看出整个Flink的架构体系基本上可以分为三层，由上往下依次是API&Libraries层、Runtime核心层以及物理部署层。

· API&Libraries层

作为分布式数据处理框架，Flink同时提供了支撑流计算和批计算的接口，同时在此基础之上抽象出不同的应用类型的组件库，如基于流处理的CEP（复杂事件处理库）、SQL&Table库和基于批处理的FlinkML（机器学习库）等、Gelly（图处理库）等。API层包括构建流计算应用的DataStream API和批计算应用的DataSet API，两者都提供给用户丰富的数据处理高级API，例如Map、FlatMap操作等，同时也提供比较低级的Process Function API，用户可以直接操作状态和时间等底层数据。

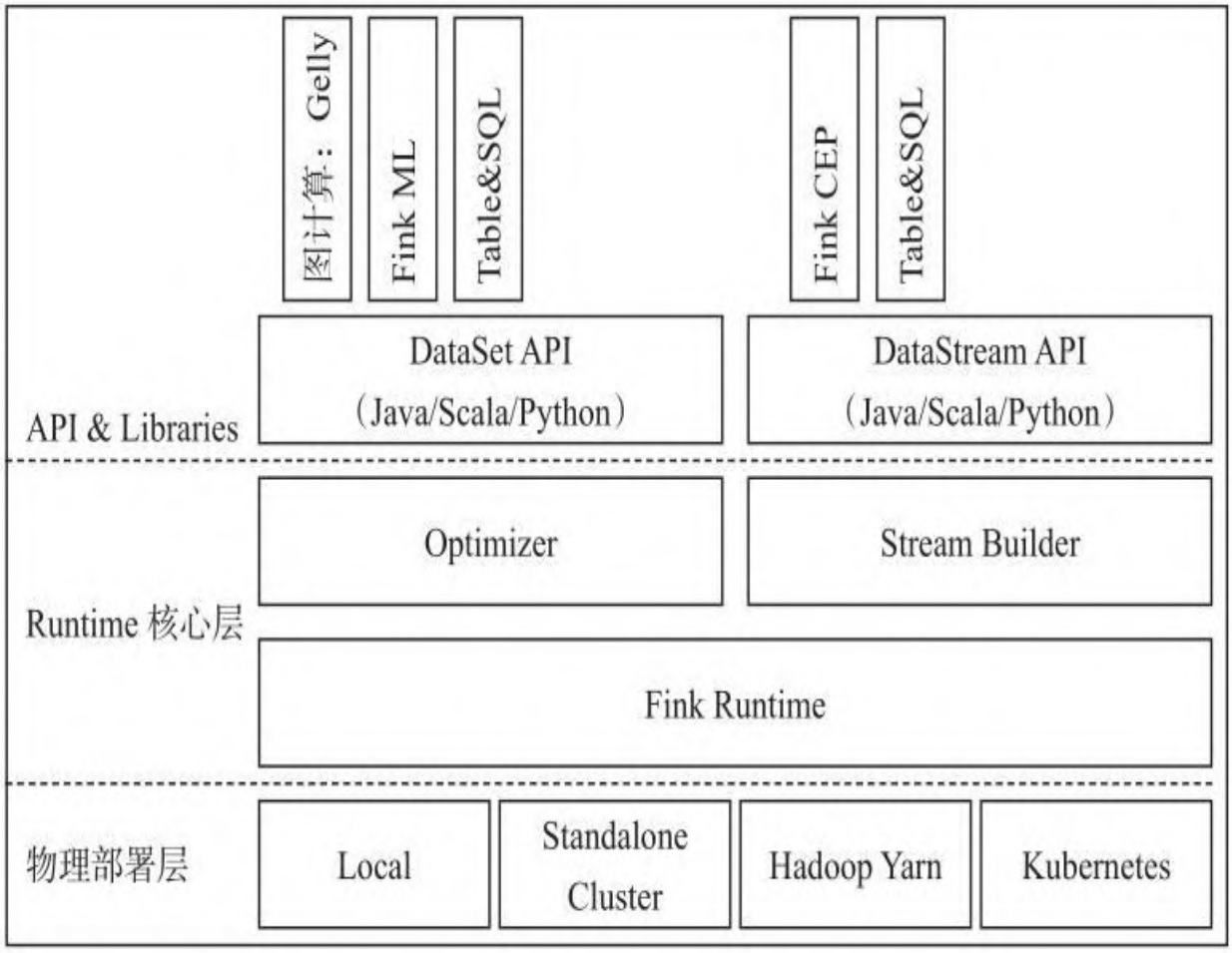

图1-5　Flink基本组件栈

· Runtime核心层

该层主要负责对上层不同接口提供基础服务，也是Flink分布式计算框架的核心实现层，支持分布式Stream作业的执行、JobGraph到ExecutionGraph的映射转换、任务调度等。将DataSteam和DataSet转成统一的可执行的Task Operator，达到在流式引擎下同时处理批量计算和流式计算的目的。

· 物理部署层

该层主要涉及Flink的部署模式，目前Flink支持多种部署模式：本地、集群（Standalone/YARN）、云（GCE/EC2）、Kubenetes。

Flink能够通过该层能够支持不同平台的部署，用户可以根据需要选择使用对应的部署模式。

1.4.2 基本架构图

Flink系统架构设计如图1-6所示，可以看出Flink整个系统主要由两个组件组成，分别为JobManager和TaskManager，Flink架构也遵循Master-Slave架构设计原则，JobManager为Master节点，TaskManager为Worker（Slave）节点。所有组件之间的通信都是借助于Akka Framework，包括任务的状态以及Checkpoint触发等信息。

（1）Client客户端

客户端负责将任务提交到集群，与JobManager构建Akka连接，然后将任务提交到JobManager，通过和JobManager之间进行交互获取任务执行状态。客户端提交任务可以采用CLI方式或者通过使用Flink WebUI提交，也可以在应用程序中指定JobManager的RPC网络端口构建ExecutionEnvironment提交Flink应用。

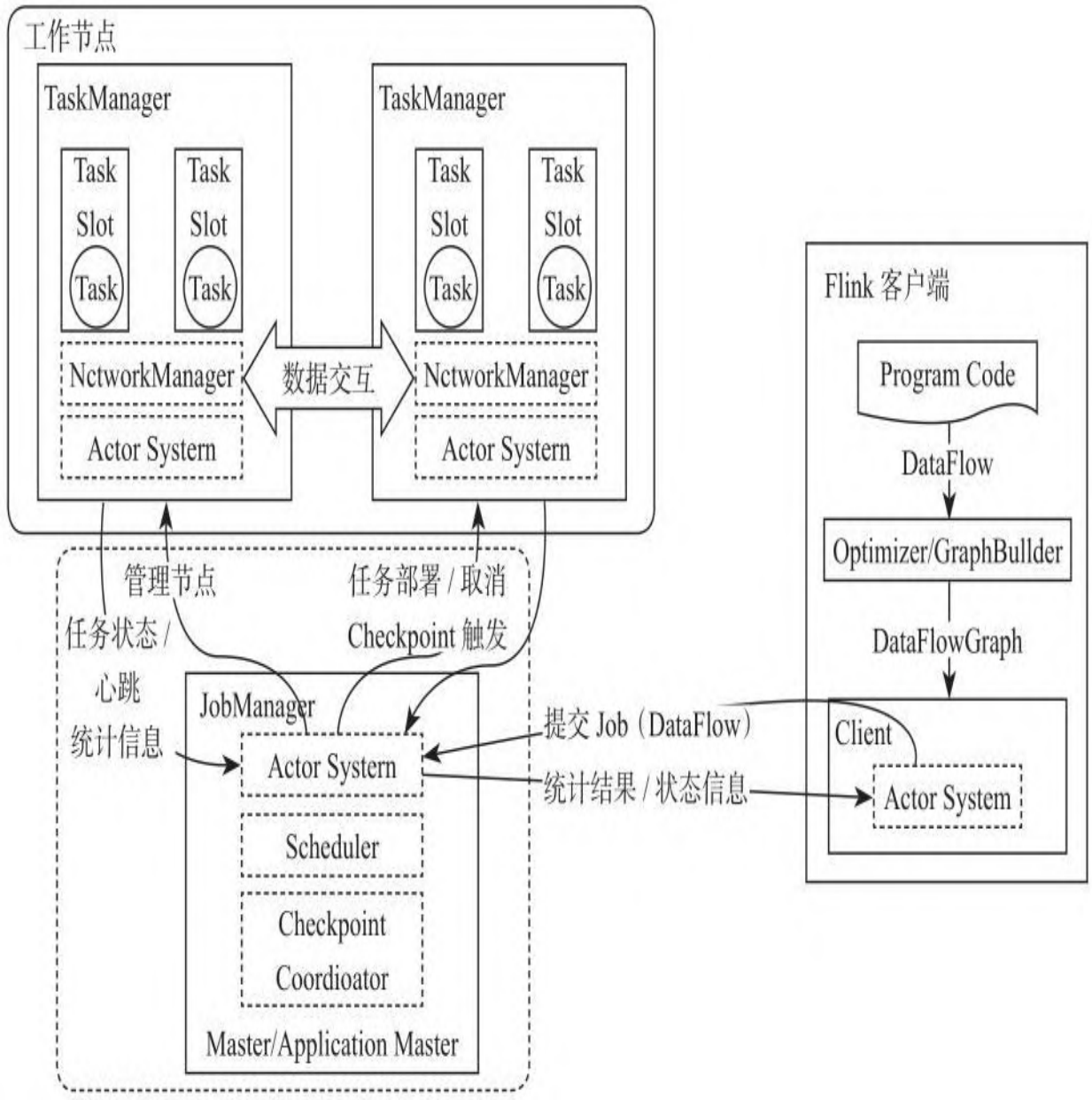

图1-6　Flink基本架构图

（2）JobManager

JobManager负责整个Flink集群任务的调度以及资源的管理，从客户端中获取提交的应用，然后根据集群中TaskManager上TaskSlot的使用情况，为提交的应用分配相应的TaskSlots资源并命令TaskManger启动从客户端中获取的应用。JobManager相当于整个集群的Master节点，且整个集群中有且仅有一个活跃的JobManager，负责整个集群的

任务管理和资源管理。JobManager和TaskManager之间通过Actor System进行通信，获取任务执行的情况并通过Actor System将应用的任务执行情况发送给客户端。同时在任务执行过程中，Flink JobManager会触发Checkpoints操作，每个TaskManager节点收到Checkpoint触发指令后，完成Checkpoint操作，所有的Checkpoint协调过程都是在Flink JobManager中完成。当任务完成后，Flink会将任务执行的信息反馈给客户端，并且释放掉TaskManager中的资源以供下一次提交任务使用。

（3）TaskManager

TaskManager相当于整个集群的Slave节点，负责具体的任务执行和对应任务在每个节点上的资源申请与管理。客户端通过将编写好的Flink应用编译打包，提交到JobManager，然后JobManager会根据已经注册在JobManager中TaskManager的资源情况，将任务分配给有资源的TaskManager节点，然后启动并运行任务。TaskManager从JobManager接收需要部署的任务，然后使用Slot资源启动Task，建立数据接入的网络连接，接收数据并开始数据处理。同时TaskManager之间的数据交互都是通过数据流的方式进行的。

可以看出，Flink的任务运行其实是采用多线程的方式，这和MapReduce多JVM进程的方式有很大的区别Fink能够极大提高CPU使用效率，在多个任务和Task之间通过TaskSlot方式共享系统资源，每个TaskManager中通过管理多个TaskSlot资源池进行对资源进行有效管理。

1.5 本章小结

在本章1.1节对Flink的基本概念及发展历史进行了介绍。1.2节对目前数据架构领域的发展进行了深入的介绍，让读者能够了解传统的数据架构到大数据架构的演变过程，以及在未来支持有状态流计算的实时计算架构会扮演什么样的角色，让用户能够对Flink这项技术的发展中有着更深入的理解。1.3节列举了Flink不同的应用场景，让读者结合自己的业务场景进行技术选型，帮助读者能够更加合理地使用Flink这项技术。1.4节介绍了Flink基本组件栈、基本架构以及Flink所具备的特性，例如支持高吞吐、低延时、强一致性保障等。通过本章的学习可以让读者对Flink有一个初步的认识和了解，接下来的章节我们将逐步深入地了解和掌握Flink分布式计算技术。

第2章

环境准备

本章主要介绍Flink在使用前的环境安装准备，包括必须依赖的环境以及相应的参数，首先从不同运行环境进行介绍，包括本地调试环境、Standalone集群环境，以及在On Yarn环境上。另外介绍Flink自带的Template模板，如何通过该项目模板本地运行代码环境的直接生成，而不需要用户进行配置进行大量的开发环境配置，节省了开发的时间成本。最后介绍Flink源码编译相关的事项，通过对源码进行编译，从而对整个Flink计算引擎有更深入的理解。

2.1 运行环境介绍

Flink执行环境主要分为本地环境和集群环境，本地环境主要为了方便用户编写和调试代码使用，而集群环境则被用于正式环境中，可以借助Hadoop Yarn或Mesos等不同的资源管理器部署自己的应用。

环境依赖

（1）JDK环境

Flink核心模块均使用Java开发，所以运行环境需要依赖JDK，本书暂不详细介绍JDK安装过程，用户可以根据官方教程自行安装，其中包括Windows和Linux环境安装，需要注意的是JDK版本需要保证在1.8以上。

（2）Scala环境

如果用户选择使用Scala作为Flink应用开发语言，则需要安装Scala执行环境，Scala环境可以通过本地安装Scala执行环境，也可以通过Maven依赖Scala-lib来引入。

（3）Maven编译环境

Flink的源代码目前仅支持通过Maven进行编译，所以如果需要对源代码进行编译，或通过IDE开发Flink Application，则建议使用Maven作为项目工程编译方式。Maven的具体安装方法这里不再赘述。

需要注意的是，Flink程序需要Maven的版本在3.0.4及以上，否则项目编译可能会出问题，建议用户根据要求进行环境的搭建。

（4）Hadoop环境

对于执行在Hadoop Yarn资源管理器的Flink应用，则需要配置对应的Hadoop环境参数。目前Flink官方提供的版本支持hadoop2.4、2.6、2.7、2.8等主要版本，所以用户可以在这些版本的Hadoop Yarn中直接运行自己的Flink应用，而不需要考虑兼容性的问题。

2.2 Flink项目模板

Flink为了对用户使用Flink进行应用开发进行简化，提供了相应的项目模板来创建开发项目，用户不需要自己引入相应的依赖库，就能够轻松搭建开发环境，前提是在JDK（1.8及以上）和Maven（3.0.4及以上）的环境已经安装好且能正常执行。在Flink项目模板中，Flink提供了分别基于Java和Scala实现的模板，下面就两套项目模板分别进行介绍和应用。

2.2.1 基于Java实现的项目模板

1. 创建项目

创建模板项目的方式有两种，一种方式是通过Maven archetype命令进行创建，另一种方式是通过Flink提供的Quickstart Shell脚本进行创建，具体实例说明如下。

· 通过Maven Archetype进行创建：

```
$ mvn archetype:generate                                \
    -DarchetypeGroupId=org.apache.flink                  \
    -DarchetypeArtifactId=flink-quickstart-java          \
    -DarchetypeCatalog=https://repository.apache.org/   \
    content/repositories/snapshots/ \
    -DarchetypeVersion=1.7.0
```

通过以上Maven命令进行项目创建的过程中，命令会交互式地提示用户对项目的groupId、artifactId、version、package等信息进行定义，且部分选项具有默认值，用户直接回车即可，如图2-1所示。我们创建了实例项目成功之后，客户端会提示用户项目创建成功，且在当前路径中具有相应创建的Maven项目。

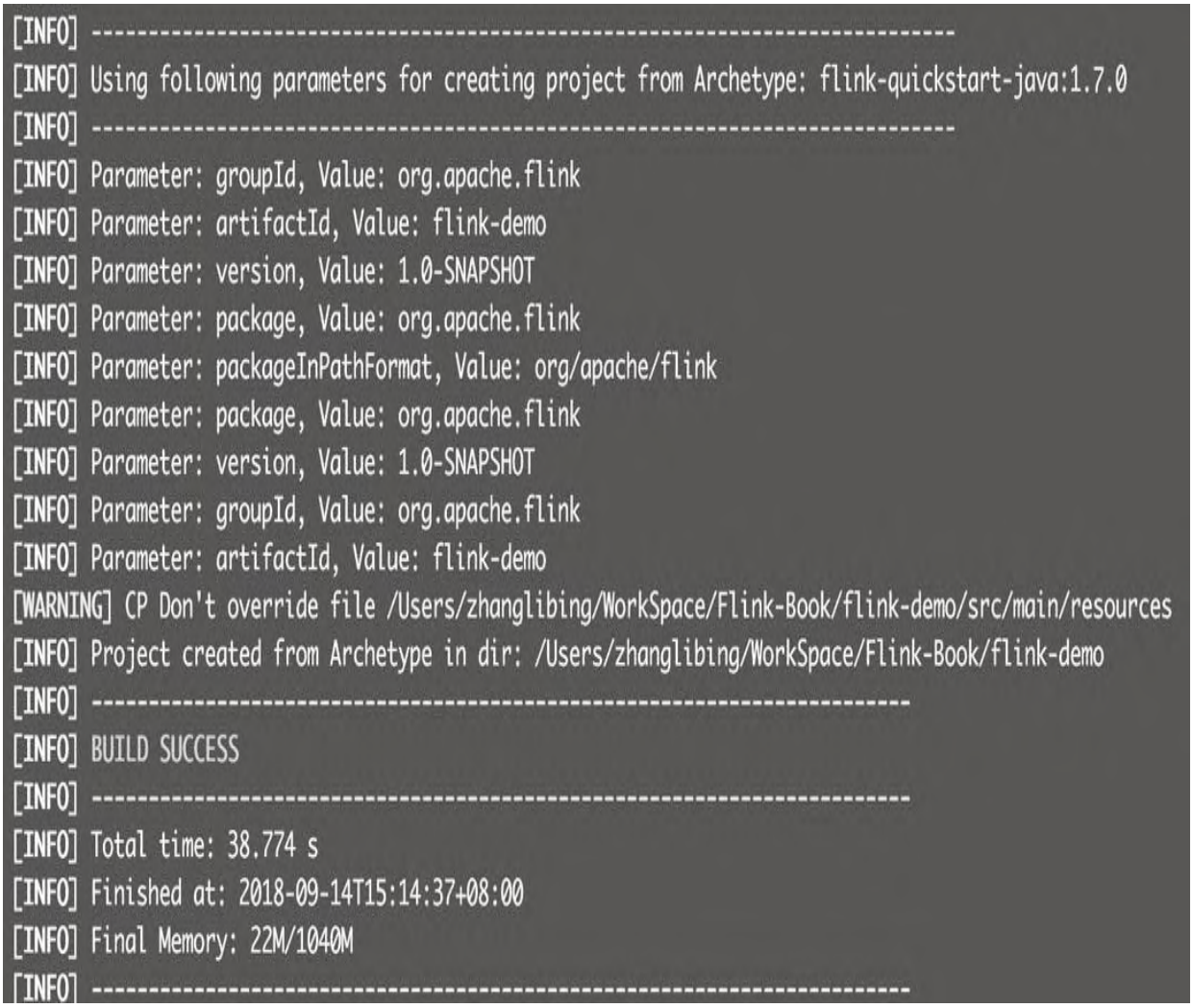

图2-1　Maven创建Java项目

· 通过quickstart脚本创建：

```
$ curl https://flink.apache.org/q/quickstart-SNAPSHOT.sh | bash -s
1.6.0
```

通过以上脚本可以比较简单地创建项目，执行后项目会自动生成，但是项目的名称和一些GAV信息都是自动生成的，用户不能进行交互式重新定义，其中的项目名称为quickstart，gourpid为org.myorg.quickstart，version为0.1。这种方式对于Flink入门相对

比较适合，其他有一定基础的情况下，则不建议使用这种方式进行项目创建。

 注意

在Maven 3.0以上的版本中，DarchetypeCatalog配置已经从命令行中移除，需要用户在Maven Settings中进行配置，或者直接将该选项移除，否则可能造成不能生成Project的错误。

2. 检查项目

对于使用quickstart curl命令创建的项目，我们可以看到的项目结构如代码清单2-1所示，如果用户使用Maven Archetype，则可以自己定义对应的artifactId等信息。

代码清单2-1　Java模板项目结构

```
tree quickstart/
quickstart/
├── pom.xml
└── src
    └── main
        ├── java
        │   └── org
        │       └── myorg
        │           └── quickstart
        │               ├── BatchJob.java
        │               └── StreamingJob.java
        └── resources
            └── log4j.properties
```

从上述项目结构可以看出，该项目已经是一个相对比较完善的Maven项目，其中创建出来对应的Java实例代码，分别是BatchJob.java和Streaming.java两个文件，分别对应Flink批量接口DataSet的实例代码和流式接口DataStream的实例代码。在创建好上述项目后，建议用户将项目导入到IDE进行后续开发，Flink官网推荐使用的是Intellij IDEA或者Eclipse进行项目开发，具体的开发环境配置可以参考下一节中的介绍。

3. 编译项目

项目经过上述步骤创建后，可以使用Maven Command命令mvn clean package对项目进行编译，编译完成后在项目同级目录会生成target/<artifact-id>-<version>.jar，则该可执行Jar包就可以通过Flink命令或者Web客户端提交到集群上执行。

 注意

通过Maven创建Java应用，用户可以在Pom中指定Main Class，这样提交执行过程中就具有默认的入口Main Class，否则需要用户在执行的Flink App的Jar应用中指定Main Class。

4. 开发应用

在项目创建和检测完成后，用户可以选择在模板项目中的代码上编写应用，也可以定义Class调用DataSet API或DataStream API进行Flink应用的开发，然后通过编译打包，上传并提交到集群上运行。具体应用的开发读者可以参考后续章节。

2.2.2　基于Scala实现的项目模板

Flink在开发接口中同样提供了Scala的接口，用户可以借助Scala高效简洁的特性进行Flink App的开发。在创建项目的过程中，也可以像上述Java一样创建Scala模板项目，而在Scala项目中唯一的区别就是可以支持使用SBT进行项目的创建和编译，以下实例，将从SBT和Maven两种方式进行介绍。

1.创建项目

（1）创建Maven项目

1）使用Maven archetype进行项目创建

代码清单2-2是通过Maven archetype命令创建Flink Scala版本的模板项目，其中项目相关的参数同创建Java项目一样，需要通过交互式的方式进行输入，用户可以指定对应的项目名称、groupid、artifactid以及version等信息。

代码清单2-2　使用Maven archetype创建Scala项目

```
mvn archetype:generate                                \
    -DarchetypeGroupId=org.apache.flink                \
    -DarchetypeArtifactId=flink-quickstart-scala       \
    -DarchetypeCatalog=https://repository.apache.org/ \
    content/repositories/snapshots/ \
    -DarchetypeVersion=1.7.0
```

执行完上述命令之后，会显示如图2-2所示的提示，表示项目创建成功，可以进行后续操作。同时可以在同级目录中看到已经创建好的Scala项目模板，其中包括了两个Scala后缀的文件。

2）使用quickstart curl脚本创建

如上节所述，在创建Scala项目模板的过程中，也可以通过quickstart curl脚本进行创建，这种方式相对比较简单，只要执行以

下命令即可：

```
curl https://flink.apache.org/q/quickstart-scala-SNAPSHOT.sh | bash
  -s 1.7.0
```

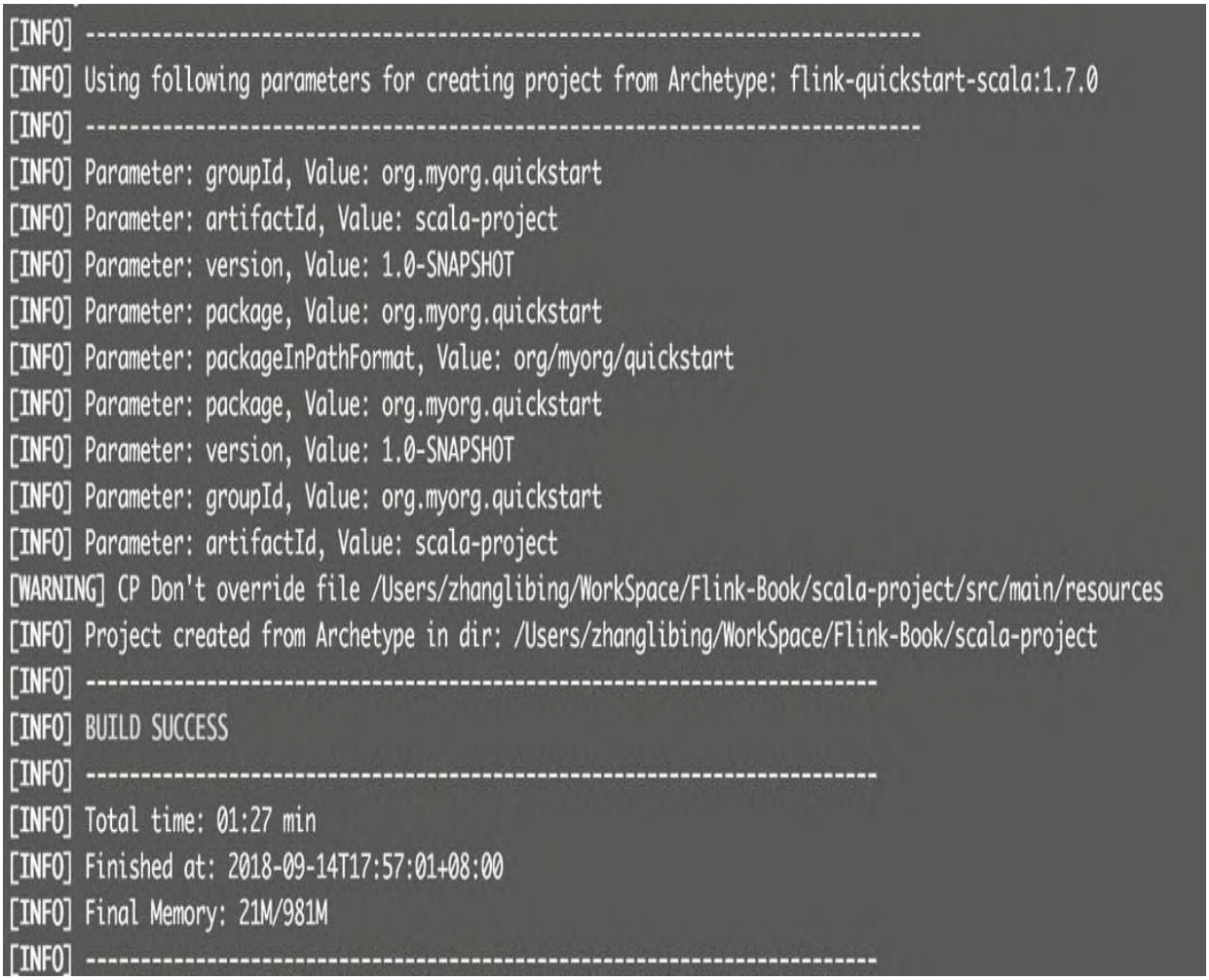

图2-2　Maven创建Scala项目

执行上述命令后就能在路径中看到相应的quickstart项目生成，其目录结构和通过Maven archetype创建的一致，只是不支持修改项目的GAV信息。

（2）创建SBT项目

在使用Scala接口开发Flink应用中，不仅可以使用Maven进行项目的编译，也可以使用SBT（Simple Build Tools）进行项目的编译和管理，其项目结构和Maven创建的项目结构有一定的区别。可以通过SBT命令或者quickstart脚本进行创建SBT项目，具体实现方式如下：

1）使用SBT命令创建项目

```
sbt new path/flink-project.g8
```

执行上述命令后，会在客户端输出创建成功的信息，表示项目创建成功，同时在同级目录中生成创建的项目，其中包含两个Scala的实例代码供用户参考。

2）使用quickstart curl脚本创建项目

可以通过使用以下指令进行项目创建Scala项目：

```
bash <(curl https://flink.apache.org/q/sbt-quickstart.sh)
```

 注意

如果项目编译方式选择SBT，则需要在环境中提前安装SBT编译器，同时版本需要在0.13.13以上，否则无法通过上述方式进行模板项目的创建，具体的安装教程可以参考SBT官方网站https://www.scala-sbt.org/download.html进行下载和安装。

2. 检查项目

对于使用Maven archetype创建的Scala项目模板，其结构和Java类似，在项目中增加了Scala的文件夹，且包含两个Scala实例代码，其中一个是实现DataSet接口的批量应用实例BatchJob，另外一个是实

现DataStream接口的流式应用实例StreamingJob，如代码清单2-3所示。

代码清单2-3　Scala模板项目结构

```
tree quickstart/
quickstart/
├── pom.xml
└── src
    └── main
        ├── resources
        │   └── log4j.properties
        └── scala
            └── org
                └── myorg
                    └── quickstart
                        ├── BatchJob.scala
                        └── StreamingJob.scala
```

3. 编译项目

1）使用Maven编译

进入到项目路径中，然后通过执行mvn clean package命令对项目进行编译，编译完成后产生target/<artifact-id>-<version>.jar。

2）使用Sbt编译

进入到项目路径中，然后通过使用sbt clean assembly对项目进行编译，编译完成后再产生target/scala_your-major-scala-version/project-name-assembly-0.1-SNAPSHOT.jar。

4. 开发应用

在项目创建和检测完成后，用户可以选择在Scala项目模板的代码上编写应用，也可以定义Class调用DataSet API或DataStream API进行Flink应用的开发，然后通过编译打包，上传并提交到集群上运行。

2.3 Flink开发环境配置

我们可以选择IntelliJ IDEA或者Eclipse作为Flink应用的开发IDE，但是由于Eclipse本身对Scala语言支持有限，所以Flink官方还是建议用户能够使用IntelliJ IDEA作为首选开发的IDE，以下将重点介绍使用IntelliJ IDEA进行开发环境的配置。

2.3.1 下载IntelliJ IDEA IDE

用户可以通过IntelliJ IDEA官方地址下载安装程序，根据操作系统选择相应的程序包进行安装。安装方式和安装包请参考https://www.jetbrains.com/idea/download/。

2.3.2 安装Scala Plugins

对于已经安装好的IntelliJ IDEA默认是不支持Scala开发环境的，如果用户选择使用Scala作为开发语言，则需要安装Scala插件进行支持。以下说明在IDEA中进行Scala插件的安装：

· 打开IDEA IDE后，在IntelliJ IDEA菜单栏中选择 Preferences选项，然后选择Plugins子选项，最后在页面中选择Browser Repositories，在搜索框中输入Scala进行检索；

· 在检索出来的选项列表中选择和安装Scala插件，如图2-3所示；

· 点击安装后重启IDE，Scala编程环境即可生效。

图2-3　检索IDE中Scala插件

2.3.3 导入Flink应用代码

开发环境配置完毕之后，下面就可以将2.3.2节中创建好的项目导入到IDE中，具体步骤如下所示：

· 启动IntelliJ IDEA，选择File→Open，在文件选择框中选择创建好的项目（quickstart），点击确定，IDEA将自动进行项目的导入；

· 如果项目中提示没有SDK，可以选择File→Project Structure，在SDKS选项中选择安装好的JDK路径，添加Scala SDK路径，如果系统没有安装Scala SDK，也可以通过Maven Dependence将scala-lib引入；

· 项目正常导入之后，程序状态显示正常，可以通过mvn clean package或者直接通过IDE中自带的工具Maven编译方式对项目进行编译；

完成了在IntelliJ IDEA中导入Flink项目，接下来用户就可以开发Flink应用了。

2.3.4　项目配置

对于通过项目模板生成的项目，项目中的主要参数配置已被初始化，所以无须额外进行配置，如果用户通过手工进行项目的创建，则需要创建Flink项目并进行相应的基础配置，包括Maven Dependences、Scala的Version等配置信息。

1.Flink基础依赖库

对于Java版本，需要在项目的pom.xml文件中配置如代码清单2-4所示的依赖库，其中flink-java和flink-streaming-java分别是批量计算DataSet API和流式计算DataStream API的依赖库，{flink.version}是官方的发布的版本号，用户可根据自身需要进行选择，本书中所有的实例代码都是基于Flink 1.7版本开发。

代码清单2-4　Flink Java项目依赖配置库

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-java</artifactId>
    <version>{flink.version}</version> <!—指定flink版本-->
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-java_2.11</artifactId>
    <version>{flink.version}</version> <!—指定flink版本-->
    <scope>provided</scope>
</dependency>
```

创建Scala版本Flink项目依赖库配置如下，和Java相比需要指定scala的版本信息，目前官方建议的是使用Scala 2.11，如果需要使用特定版本的Scala，则要将源码下载进行指定Scala版本编译，否则Scala各大版本之间兼容性较弱会导致应用程序在实际环境中无法运行的问题。Flink基于Scala语言项目依赖配置库如代码清单2-5所示：

代码清单2-5　Flink Scala项目依赖配置库

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-scala_2.11</artifactId>
    <version>{flink.version} </version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-scala_2.11</artifactId>
    <version>{flink.version} </version>
    <scope>provided</scope>
</dependency>
```

另外在上述Maven Dependences配置中，核心的依赖库配置的Scope为provided，主要目的是在编译阶段能够将依赖的Flink基础库排除在项目之外，当用户提交应用到Flink集群的时候，就避免因为引入Flink基础库而导致Jar包太大或类冲突等问题。而对于Scope配置成provided的项目可能出现本地IDE中无法运行的问题，可以在Maven中通过配置Profile的方式，动态指定编译部署包的scope为provided，本地运行过程中的scope为compile，从而解决本地和集群环境编译部署的问题。

 注意

由于Flink在最新版本中已经不再支持scala 2.10的版本，建议读者使用scala 2.11，同时Flink将在未来的新版本中逐渐支持Scala 2.12。

2.Flink Connector和Lib依赖库

除了上述Flink项目中应用开发必须依赖的基础库之外，如果用户需要添加其他依赖，例如Flink中内建的Connector，或者其他第三方依赖库，需要在项目中添加相应的Maven Dependences，并将这些Dependence的Scope需要配置成compile。

如果项目中需要引入Hadoop相关依赖包，和基础库一样，在打包编译的时候将Scope注明为provided，因为Flink集群中已经将Hadoop

依赖包添加在集群的环境中，用户不需要再将相应的Jar包打入应用中，否则容易造成Jar包冲突。


注意

对于有些常用的依赖库，为了不必每次都要上传依赖包到集群上，用户可以将依赖的包可以直接上传到Flink安装部署路径中的lib目录中，这样在集群启动的时候就能够将依赖库加载到集群的ClassPath中，无须每次在提交任务的时候上传依赖的Jar包。

2.4 运行Scala REPL

和Spark Shell一样，Flink也提供了一套交互式解释器（Scala-Shell），用户能够在客户端命令行交互式编程，执行结果直接交互式地显示在客户端控制台上，不需要每次进行编译打包在集群环境中运行，目前该功能只支持使用Scala语言进行程序开发。另外需要注意的是在开发或者调试程序的过程中可以使用这种方式，但在正式的环境中则不建议使用。

2.4.1 环境支持

用户可以选择在不同的环境中启动Scala Shell，目前支持Local、Remote Cluster和Yarn Cluster模式，具体命令可以参考以下说明：

· 通过start-scala-shell.sh启动本地环境；

```
bin/start-scala-shell.sh local
```

· 可以启动远程集群环境，指定远程Flink集群的hostname和端口号；

```
bin/start-scala-shell.sh remote <hostname> <portnumber>
```

· 启动Yarn集群环境，环境中需要含有hadoop客户端配置文件；

```
bin/start-scala-shell.sh yarn -n 2
```

2.4.2 运行程序

启动Scala Shell交互式解释器后，就可以进行Flink流式应用或批量应用的开发。需要注意的是，Flink已经在启动的执行环境中初始化好了相应的Environment，分别使用“benv”和“senv”获取批量计算环境和流式计算环境，然后使用对应环境中的API开发Flink应用。以下代码实例分别是用批量和流式实现WordCount的应用，读者可以直接在启动Flink Scala Shell客户端后执行并输出结果。

· 通过Scala-Shell运行批量计算程序，调用benv完成对单词数量的统计。

```
scala> val textBatch = benv.fromElements(
    "To be, or not to be,--that is the question:--",
    "Whether 'tis nobler in the mind to suffer")
scala> val counts = textBatch
    .flatMap { _.toLowerCase.split("\\W+") }
    .map { (_, 1) }.groupBy(0).sum(1)
scala> counts.print()
```

· 通过Scala-Shell运行流式计算，调用senv完成对单词数量的统计。

```
scala> val textStreaming = senv.fromElements(
    "flink has Stateful Computations over Data Streams")
scala> val countsStreaming = textStreaming
    .flatMap { _.toLowerCase.split("\\W+") }
    .map { (_, 1) }.keyBy(0).sum(1)
scala> countsStreaming.print()
scala> senv.execute("Streaming Wordcount")
```


注意

用户在使用交互式解释器方式进行应用开发的过程中，流式作业和批量作业中的一些操作（例如写入文件）并不会立即执行，而是需要用户在程序的最后执行env.execute(“appname”)命令，这样整个程序才能触发运行。

2.5 Flink源码编译

对于想深入了解Flink源码结构和实现原理的读者，可以按照本节的内容进行Flink源码编译环境的搭建，完成Flink源码的编译，具体操作步骤如下所示。

Flink源码可以从官方 Git Repository上通过git clone命令下载：

```
git clone https://github.com/apache/flink
```

读者也可以通过官方镜像库手动下载，下载地址为https://archive.apache.org/dist/flink/。用户根据需要选择需要编译的版本号，下载代码放置在本地路径中，然后通过如下Maven命令进行编译，需要注意的是，Flink源码编译依赖于JDK和Maven的环境，且JDK必须在1.8版本以上，Maven必须在3.0版本以上，否则会导致编译出错。

```
mvn clean install -DskipTests
```

（1）Hadoop版本指定

Flink镜像库中已经编译好的安装包通常对应的是Hadoop的主流版本，如果用户需要指定Hadoop版本编译安装包，可以在编译过程中使

用-Dhadoop.version参数指定Hadoop版本，目前Flink支持的Hadoop版本需要在2.4以上。

```
mvn clean install -DskipTests -Dhadoop.version=2.6.1
```

如果用户使用的是供应商提供的Hadoop平台，如Cloudera的CDH等，则需要根据供应商的系统版本来编译Flink，可以指定-Pvendor-repos参数来激活类似于Cloudera的Maven Repositories，然后在编译过程中下载依赖对应版本的库。

```
mvn clean install -DskipTests -Pvendor-repos -
Dhadoop.version=2.6.1-cdh5.0.0
```

（2）Scala版本指定

Flink中提供了基于Scala语言的开发接口，包括DataStream API、DataSet API、SQL等，需要在代码编译过程中指定Scala的版本，因为Scala版本兼容性相对较弱，因此不同的版本之间的差异相对较大。目前Flink最近的版本基本已经支持Scala-2.11，不再支持Scala-2.10的版本，在Flink 1.7开始支持Scala-2.12版本，社区则建议用户使用Scala-2.11或者Scala-2.12的Scala环境。

2.6 本章小结

本章介绍了在使用Flink应用程序开发之前必备的环境要求，分别对基础环境依赖例如JDK、Maven等进行说明。通过借助于Flink提供的项目模板创建不同编程语言的Flink应用项目。在2.3节中介绍了如何构建Flink开发环境，以及如何选择合适的IDE和项目配置。在2.4节中介绍了Flink交互式编程客户端Scala REPL，通过使用交互式客户端编写和执行Flink批量和流式应用代码。2.5节介绍了Flink源码编译和打包，编译中指定不同的Hadoop版本以及Scala版本等。在第3章将重点介绍Flink编程模型，其中包括Flink程序基本构成，以及Flink所支持的数据类型等。

第3章

Flink编程模型

本章将重点介绍Flink编程模型中的基本概念和编写Flink应用程序所遵循的基本模式。其中，包括Flink支持的数据集类型，有界数据集和无界数据集的区别，以及有界数据集和无界数据集之间的转换。同时针对无界和有界数据集的处理，将介绍Flink分别提供对应的开发接口DataStream API和DataSet API的使用。然后介绍Flink程序结构，包括基本的Flink应用所包含的组成模块等。最后介绍Flink所支持的数据类型，包括常用的POJOs、Tuples等数据类型。

3.1 数据集类型

现实世界中，所有的数据都是以流式的形态产生的，不管是哪里产生的数据，在产生的过程中都是一条条地生成，最后经过了存储和转换处理，形成了各种类型的数据集。如图3-1所示，根据现实的数据产生方式和数据产生是否含有边界（具有起始点和终止点）角度，将数据分为两种类型的数据集，一种是有界数据集，另外一种是无界数据集。

1. 有界数据集

有界数据集具有时间边界，在处理过程中数据一定会在某个时间范围内起始和结束，有可能是一分钟，也有可能是一天内的交易数据。对有界数据集的数据处理方式被称为批计算（Batch Processing），例如将数据从RDBMS或文件系统等系统中读取出来，然后在分布式系统内处理，最后再将处理结果写入存储介质中，整个过程就被称为批处理过程。而针对批数据处理，目前业界比较流行的分布式批处理框架有Apache Hadoop和Apache Spark等。

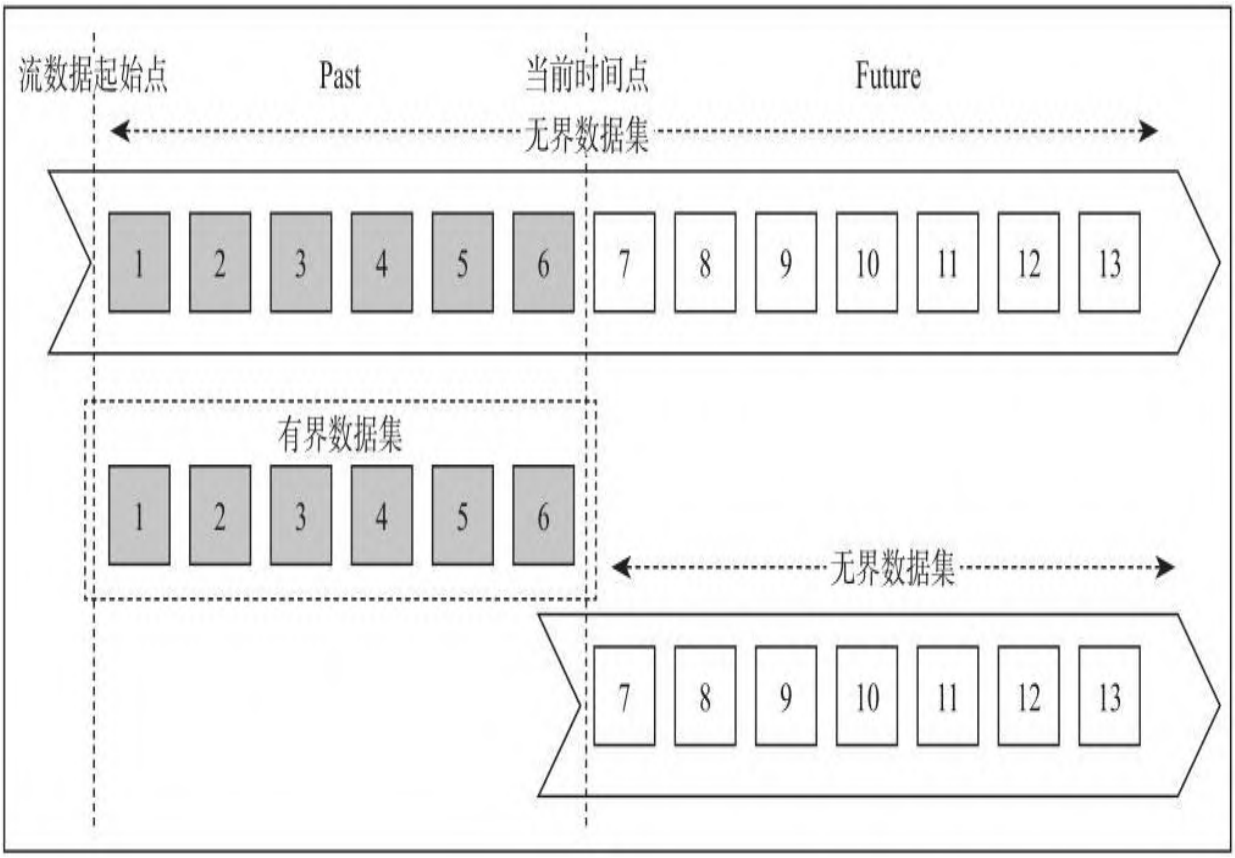

图3-1　有界数据集和无界数据集

2. 无界数据集

对于无界数据集，数据从开始生成就一直持续不断地产生新的数据，因此数据是没有边界的，例如服务器的日志、传感器信号数据等。和批量数据处理方式对应，对无界数据集的数据处理方式被称为流式数据处理，简称为流处理（Streaming Process）。可以看出，流式数据处理过程实现复杂度会更高，因为需要考虑处理过程中数据的顺序错乱，以及系统容错等方面的问题，因此流处理需要借助专门的流数据处理技术。目前业界的Apache Storm、Spark Streaming、Apache Flink等分布式计算引擎都能不同程度地支持处理流式数据。

3. 统一数据处理

有界数据集和无界数据集只是一个相对的概念，主要根据时间的范围而定，可以认为一段时间内的无界数据集其实就是有界数据集，同时有界数据也可以通过一些方法转换为无界数据。例如系统一年的

订单交易数据，其本质上应该是有界的数据集，可是当我们把它一条一条按照产生的顺序发送到流式系统，通过流式系统对数据进行处理，在这种情况下可以认为数据是相对无界的。对于无界数据也可以拆分成有界数据进行处理，例如将系统产生的数据接入到存储系统，按照年或月进行切割，切分成不同时间长度的有界数据集，然后就可以通过批处理方式对数据进行处理。从以上分析我们可以得出结论：有界数据和无界数据其实是可以相互转换的。有了这样的理论基础，对于不同的数据类型，业界也提出了不同的能够统一数据处理的计算框架。

目前在业界比较熟知的开源大数据处理框架中，能够同时支持流式计算和批量计算，比较典型的代表分别为Apache Spark和Apache Flink两套框架。其中Spark通过批处理模式来统一处理不同类型的数据集，对于流数据是将数据按照批次切分成微批（有界数据集）来进行处理。Flink则从另外一个角度出发，通过流处理模式来统一处理不同类型的数据集。Flink用比较符合数据产生的规律方式处理流式数据，对于有界数据可以转换成无界数据统一进行流式，最终将批处理和流处理统一在一套流式引擎中，这样用户就可以使用一套引擎进行批计算和流计算的任务。

前面已经提到用户可能需要通过将多种计算框架并行使用来解决不同类型的数据处理，例如用户可能使用Flink作为流计算的引擎，使用Spark或者MapReduce作为批计算的引擎，这样不仅增加了系统的复杂度，也增加了用户学习和运维的成本。而Flink作为一套新兴的分布式计算引擎，能够在统一平台中很好地处理流式任务和批量任务，同时使用流计算模式更符合数据产生的规律，相信Flink会在未来成为众多大数据处理引擎的一颗明星。

3.2 Flink编程接口

如图3-2所示，Flink根据数据集类型的不同将核心数据处理接口分为两大类，一类是支持批计算的接口DataSet API，另外一类是支持流计算的接口DataStream API。同时Flink将数据处理接口抽象成四层，由上向下分别为SQL API、Table API、DataStream /DataSet API以及Stateful Stream Processing API，用户可以根据需要选择任意一层抽象接口来开发Flink应用。

（1）Flink SQL

从图3-2中可以看出，Flink提供了统一的SQL API完成对批计算和流计算的处理，目前SQL API也是社区重点发展的接口层，对SQL API也正在逐步完善中，其主要因为SQL语言具有比较低的学习成本，能够让数据分析人员和开发人员更快速地上手，帮助其更加专注于业务本身而不是受限于复杂的编程接口。

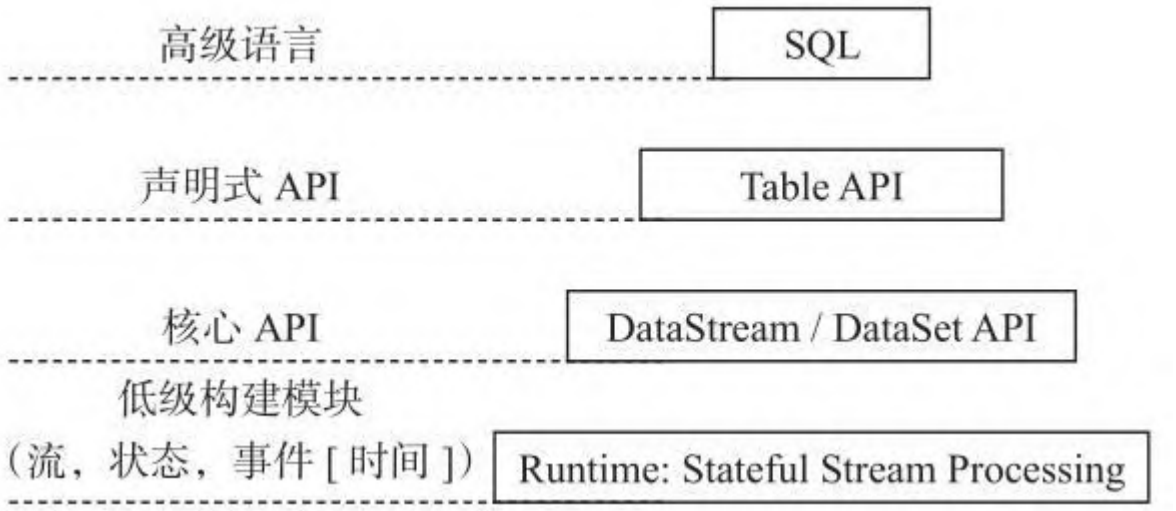

图3-2　Flink接口分层与抽象

（2）Table API

Table API将内存中的DataStream和DataSet数据集在原有的基础之上增加Schema信息，将数据类型统一抽象成表结构，然后通过Table API提供的接口处理对应的数据集。SQL API则可以直接查询Table API中注册表中的数据表。Table API构建在DataStream和DataSet之上的同时，提供了大量面向领域语言的编程接口，例如GroupByKey、Join等操作符，提供给用户一种更加友好的处理数据集的方式。除此之外，Table API在转换为DataStream和DataSet的数据处理过程中，也应用了大量的优化规则对处理逻辑进行了优化。同时Table API中的Table可以和DataStream及DataSet之间进行相互转换。

（3）DataStream API和DataSet API

DataStream API和DataSet API主要面向具有开发经验的用户，用户可以使用DataStream API处理无界流数据，使用DataSet API处理批量数据。DataStream API和DataSet API接口同时提供了各种数据处理接口，例如map，filter、oins、aggregations、window等方法，同时每种接口都支持了Java、Scala及Python等多种开发语言的SDK。

（4）Stateful Stream Process API

Stateful Stream Process API是Flink中处理Stateful Stream最底层的接口，用户可以使用Stateful Stream Process接口操作状态、时间等底层数据。使用Stream Process API接口开发应用的灵活性非常强，可以实现非常复杂的流式计算逻辑，但是相对用户使用成本也比较高，一般企业在使用Flink进行二次开发或深度封装的时候会用到这层接口。

3.3 Flink程序结构

和其他分布式处理引擎一样，Flink应用程序也遵循着一定的编程模式。不管是使用DataStream API还是DataSet API基本具有相同的程序结构，如代码清单3-1所示。通过流式计算的方式实现对文本文件中的单词数量进行统计，然后将结果输出在给定路径中。

代码清单3-1 Streaming WordCount实例代码

```
package com.realtime.flink.streaming
import org.apache.flink.api.java.utils.ParameterTool
import org.apache.flink.streaming.api.scala.{DataStream,
StreamExecution
Environment, _}
object WordCount {
  def main(args: Array[String]) {
    // 第一步：设定执行环境设定
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // 第二步：指定数据源地址，读取输入数据
    val text = env.readTextFile("file:///path/file")
    // 第三步：对数据集指定转换操作逻辑
    val counts: DataStream[(String, Int)] = text
        .flatMap(_.toLowerCase.split(" "))
      .filter(_.nonEmpty)
      .map((_, 1))
      .keyBy(0)
      .sum(1)
    // 第四步：指定计算结果输出位置
    if (params.has("output")) {
      counts.writeAsText(params.get("output"))
    } else {
      println("Printing result to stdout. Use --output to specify
output path.")
      counts.print()
```

```
    }
    // 第五步：指定名称并触发流式任务
    env.execute("Streaming WordCount")
  }
}
```

整个Flink程序一共分为5步，分别为设定Flink执行环境、创建和加载数据集、对数据集指定转换操作逻辑、指定计算结果输出位置、调用execute方法触发程序执行。对于所有的Flink应用程序基本都含有这5个步骤，下面将详细介绍每个步骤。

1.Execution Environment

运行Flink程序的第一步就是获取相应的执行环境，执行环境决定了程序执行在什么环境（例如本地运行环境或者集群运行环境）中。同时不同的运行环境决定了应用的类型，批量处理作业和流式处理作业分别使用的是不同的Execution Environment。例如StreamExecutionEnvironment是用来做流式数据处理环境，ExecutionEnvironment是批量数据处理环境。可以使用三种方式获取Execution Environment，例如StreamExecution-Envirenment。

```
//设定Flink运行环境，如果在本地启动则创建本地环境，如果是在集群上启动，则创建集群环境
StreamExecutionEnvironment.getExecutionEnvironment
//指定并行度创建本地执行环境
StreamExecutionEnvironment.createLocalEnvironment(5)
//指定远程JobManagerIP和RPC端口以及运行程序所在jar包及其依赖包
StreamExecutionEnvironment.createRemoteEnvironment("JobManagerHost",6021,5,"/user/application.jar")
```

其中第三种方式可以直接从本地代码中创建与远程集群的Flink JobManager的RPC连接，通过指定应用程序所在的Jar包，将运行程序远程拷贝到JobManager节点上，然后将Flink应用程序运行在远程的环境中，本地程序相当于一个客户端。

和StreamExecutionEnvironment构建过程一样，开发批量应用需要获取Execution-Environment来构建批量应用开发环境，如以下代码

实例通过调用ExecutionEnvironment的静态方法来获取批计算环境。

```
//设定Flink运行环境，如果在本地启动则创建本地环境，如果是在集群上启动，则创建集群环境
ExecutionEnvironment.getExecutionEnvironment
//指定并行度创建本地执行环境
ExecutionEnvironment.createLocalEnvironment(5)
//指定远程JobManagerIP和RPC端口以及运行程序所在jar包及其依赖包
ExecutionEnvironment.createRemoteEnvironment("JobManagerHost",6021,5,"/user/application.jar")
```

针对Scala和Java不同的编程语言环境，Flink分别制定了不同的语言同时分别定义了不同的Execution Environment接口。StreamExecutionEnvironment Scala开发接口在org.apache.flink.streaming.api.scala包中，Java开发接口在org.apache.flink.streaming.api.java包中；ExecutionEnvironment Scala接口在org.apache.flink.api.scala包中，Java开发接口则在org.apache.flink.api.java包中。用户使用不同语言开发Flink应用时需要引入不同环境对应的执行环境。

2.初始化数据

创建完成ExecutionEnvironment后，需要将数据引入到Flink系统中。Execution-Environment提供不同的数据接入接口完成数据的初始化，将外部数据转换成DataStream<T>或DataSet<T>数据集。如以下代码所示，通过调用readTextFile()方法读取file:///pathfile路径中的数据并转换成DataStream<String>数据集。

```
val text:DataStream[String] =
env.readTextFile("file:///path/file")
```

通过读取文件并转换为DataStream[String]数据集，这样就完成了从本地文件到分布式数据集的转换，同时在Flink中提供了多种从外部读取数据的连接器，包括批量和实时的数据连接器，能够将Flink系统和其他第三方系统连接，直接获取外部数据。

3. 执行转换操作

数据从外部系统读取并转换成DataStream或者DataSet数据集后，下一步就将对数据集进行各种转换操作。Flink中的Transformation操作都是通过不同的Operator来实现，每个Operator内部通过实现Function接口完成数据处理逻辑的定义。在DataStream API和DataSet API提供了大量的转换算子，例如map、flatMap、filter、keyBy等，用户只需要定义每种算子执行的函数逻辑，然后应用在数据转换操作Dperator接口中即可。如下代码实现了对输入的文本数据集通过FlatMap算子转换成数组，然后过滤非空字段，将每个单词进行统计，得到最后的词频统计结果。

```
val counts: DataStream[(String, Int)] = text
    .flatMap(_.toLowerCase.split(" "))//执行FlatMap转换操作
  .filter(_.nonEmpty)//执行Filter操作过滤空字段
  .map((_, 1))//执行map转换操作，转换成key-value接口
  .keyBy(0)//按照指定key对数据重分区
  .sum(1)//执行求和运算操作
```

在上述代码中，通过Scala接口处理数据，极大地简化数据处理逻辑的定义，只需要通过传入相应Lambada计算表达式，就能完成Function定义。特殊情况下用户也可以通过实现Function接口来完成定义数据处理逻辑。然后将定义好的Function应用在对应的算子中即可。Flink中定义Funciton的计算逻辑可以通过如下几种方式完成定义。

（1）通过创建Class实现Funciton接口

Flink中提供了大量的函数供用户使用，例如以下代码通过定义MyMapFunction Class实现MapFunction接口，然后调用DataStream的map()方法将MyMapFunction实现类传入，完成对实现将数据集中字符串记录转换成大写的数据处理。

```
val dataStream: DataStream[String] = env.fromElements("hello",
"flink")
dataStream.map(new MyMapFunction)
```

```
class MyMapFunction extends MapFunction[String, String] {
  override def map(t: String): String = {
    t.toUpperCase()
  }
}
```

（2）通过创建匿名类实现Funciton接口

除了以上单独定义Class来实现Function接口之处，也可以直接在map()方法中创建匿名实现类的方式定义函数计算逻辑。

```
val dataStream: DataStream[String] = env.fromElements("hello",
"flink")
//通过创建MapFunction匿名实现类来定义Map函数计算逻辑
  dataStream.map(new MapFunction[String, String] {
    //实现对输入字符串大写转换
    override def map(t: String): String = {
      t.toUpperCase()
    }
  })
```

（3）通过实现RichFunciton接口

前面提到的转换操作都实现了Function接口，例如MapFunction和FlatMap-Function接口，在Flink中同时提供了RichFunction接口，主要用于比较高级的数据处理场景，RichFunction接口中有open、close、getRuntimeContext和setRuntimeContext等方法来获取状态，缓存等系统内部数据。和MapFunction相似，RichFunction子类中也有RichMap-Function，如下代码通过实现RichMapFunction定义数据处理逻辑，具体的RichFunction的介绍读者可以参考后续章节中心介绍。

```
//定义匿名类实现RichMapFunction接口，完成对字符串到整形数字的转换
data.map (new RichMapFunction[String, Int] {
  def map(in: String):Int = { in.toInt }
})
```

4. 分区Key指定

在DataStream数据经过不同的算子转换过程中，某些算子需要根据指定的key进行转换，常见的有join、coGroup、groupBy类算子，需要先将DataStream或DataSet数据集转换成对应的KeyedStream和GroupedDataSet，主要目的是将相同key值的数据路由到相同的Pipeline中，然后进行下一步的计算操作。需要注意的是，在Flink中这种操作并不是真正意义上将数据集转换成Key-Value结构，而是一种虚拟的key，目的仅仅是帮助后面的基于Key的算子使用，分区人Key可以通过两种方式指定：

（1）根据字段位置指定

在DataStream API中通过keyBy()方法将DataStream数据集根据指定的key转换成重新分区的KeyedStream，如以下代码所示，对数据集按照相同key进行sum()聚合操作。

```
val dataStream: DataStream[(String, Int)] = env.fromElements(("a",
1), ("c",
  2))
//根据第一个字段重新分区，然后对第二个字段进行求和运算
Val result = dataStream.keyBy(0).sum(1)
```

在DataSet API中，如果对数据根据某一条件聚合数据，对数据进行聚合时候，也需要对数据进行重新分区。如以下代码所示，使用DataSet API对数据集根据第一个字段作为GroupBy的key，然后对第二个字段进行求和运算。

```
val dataSet = env.fromElements(("hello", 1), ("flink", 3))
  //根据第一个字段进行数据重分区
  val groupedDataSet:GroupedDataSet[(String,Int)] =
dataSet.groupBy(0)
  //求取相同key值下第二个字段的最大值
  groupedDataSet.max(1)
```

（2）根据字段名称指定

KeyBy和GroupBy的Key除了能够通过字段位置来指定之外，也可以根据字段的名称来指定。使用字段名称需要DataStream中的数据结构类型必须是Tuple类或者POJOs类的。如以下代码所示，通过指定name字段名称来确定groupby的key字段。

```
val personDataSet = env.fromElements(new Persion("Alex", 18),new
Persion("Peter", 43))
//指定name字段名称来确定groupby字段
personDataSet.groupBy("name").max(1)
```

如果程序中使用Tuple数据类型，通常情况下字段名称从1开始计算，字段位置索引从0开始计算，以下代码中两种方式是等价的。

```
val personDataStream = env.fromElements(("Alex", 18),("Peter",
43))
//通过名称指定第一个字段名称
personDataStream.keyBy("_1")
//通过位置指定第一个字段
personDataStream.keyBy(0)
```

如果在Flink中使用嵌套的复杂数据结构，可以通过字段名称指定Key，例如：

```
class CompelexClass(var nested: NestedClass, var tag: String) {
  def this() { this(null, "") }
}
class NestedClass (
    var id: Int,
    tuple: (Long, Long, String)){
  def this() { this(0, (0, 0, "")) }
}
```

通过调用“nested”获取整个NestedClass对象里所有的字段，调用“tag”获取CompelexClass中tag字段，调用“nested.id”获取NestedClass中的id字段，调用“nested.tuple._1”获取NestedClass

中tuple元祖的第一个字段。由此可以看出，Flink能够支持在复杂数据结构中灵活地获取字段信息，这也是非 Key-Value的数据结构所具有的优势。

（3）通过Key选择器指定

另外一种方式是通过定义Key Selector来选择数据集中的Key，如下代码所示，定义KeySelector，然后复写getKey方法，从Person对象中获取name为指定的Key。

```
case class Person(name: String, age: Int)
val person= env.fromElements(Person("hello",1), Person("flink",4))
//定义KeySelector,实现getKey方法从case class中获取Key
val keyed: KeyedStream[WC]= person.keyBy(new KeySelector[Person,
String]() {
  override def getKey(person: Person): String = person.word
})
```

5. 输出结果

数据集经过转换操作之后，形成最终的结果数据集，一般需要将数据集输出在外部系统中或者输出在控制台之上。在Flink DataStream和DataSet接口中定义了基本的数据输出方法，例如基于文件输出writeAsText()，基于控制台输出print()等。同时Flink在系统中定义了大量的Connector，方便用户和外部系统交互，用户可以直接通过调用addSink()添加输出系统定义的DataSink类算子，这样就能将数据输出到外部系统。以下实例调用DataStream API中的writeAsText()和print()方法将数据集输出在文件和客户端中。

```
//将数据输出到文件中
counts.writeAsText("file://path/to/savefile")
//将数据输出控制台
counts.print()
```

6. 程序触发

所有的计算逻辑全部操作定义好之后，需要调用ExecutionEnvironment的execute()方法来触发应用程序的执行，其中execute()方法返回的结果类型为JobExecutionResult，里面包含了程序执行的时间和累加器等指标。需要注意的是，execute方法调用会因为应用的类型有所不同，DataStream流式应用需要显性地指定execute()方法运行程序，如果不调用则Flink流式程序不会执行，但对于DataSet API输出算子中已经包含对execute()方法的调用，则不需要显性调用execute()方法，否则会出现程序异常。

```
//调用StreamExecutionEnvironment的execute方法执行流式应用程序
env.execute("App Name");
```

3.4 Flink数据类型

3.4.1 数据类型支持

Flink支持非常完善的数据类型，数据类型的描述信息都是由TypeInformation定义，比较常用的TypeInformation有BasicTypeInfo、TupleTypeInfo、CaseClassTypeInfo、PojoTypeInfo类等。TypeInformation主要作用是为了在Flink系统内有效地对数据结构类型进行管理，能够在分布式计算过程中对数据的类型进行管理和推断。同时基于对数据的类型信息管理，Flink内部对数据存储也进行了相应的性能优化。Flink能够支持任意的Java或Scala的数据类型，不用像Hadoop中的org.apache.hadoop.io.Writable而实现特定的序列化和反序列化接口，从而让用户能够更加容易使用已有的数据结构类型。另外使用TypeInformation管理数据类型信息，能够在数据处理之前将数据类型推断出来，而不是真正在触发计算后才识别出，这样能够及时有效地避免用户在使用Flink编写应用的过程中的数据类型问题。

1. 原生数据类型

Flink通过实现BasicTypeInfo数据类型，能够支持任意Java原生基本类型（装箱）或 String类型，例如Integer、String、Double等，如以下代码所示，通过从给定的元素集中创建DataStream数据集。

```
//创建Int类型的数据集
val intStream:DataStream[Int] = env.fromElements(3, 1, 2, 1, 5)
//创建String类型的数据集
```

```
val dataStream: DataStream[String] = env.fromElements("hello",
"flink")
```

Flink实现另外一种TypeInfomation是BasicArrayTypeInfo，对应的是Java基本类型数组（装箱）或 String对象的数组，如下代码通过使用Array数组和List集合创建DataStream数据集。

```
//通过从数组中创建数据集
val dataStream: DataStream[Int] = env.fromCollection(Array(3, 1,
2, 1, 5))
//通过List集合创建数据集
val dataStream: DataStream[Int] = env.fromCollection(List(3, 1, 2,
1, 5))
```

2. Java Tuples类型

通过定义TupleTypeInfo来描述Tuple类型数据，Flink在Java接口中定义了元祖类（Tuple）供用户使用。Flink Tuples是固定长度固定类型的Java Tuple实现，不支持空值存储。目前支持任意的Flink Java Tuple类型字段数量上限为25，如果字段数量超过上限，可以通过继承Tuple类的方式进行拓展。如下代码所示，创建Tuple数据类型数据集。

```
//通过实例化Tuple2创建具有两个元素的数据集
val tupleStream2: DataStream[Tuple2[String, Int]] =
env.fromElements(new Tuple2("a",1), new Tuple2("c", 2))
```

3. Scala Case Class类型

Flink通过实现CaseClassTypeInfo支持任意的Scala Case Class，包括Scala tuples类型，支持的字段数量上限为22，支持通过字段名称和位置索引获取指标，不支持存储空值。如下代码实例所示，定义WordCount Case Class数据类型，然后通过fromElements方

法创建input数据集，调用keyBy()方法对数据集根据word字段重新分区。

```
//定义WordCount Case Class数据结构
case class WordCount(word: String, count: Int)
//通过fromElements方法创建数据集
val input = env.fromElements(WordCount("hello", 1),
WordCount("world", 2))
val keyStream1 = input.keyBy("word") // 根据word字段为分区字段，
val keyStream2 = input.keyBy(0) //也可以通过指定position分区
```

通过使用Scala Tuple创建DataStream数据集，其他的使用方式和Case Class相似。需要注意的是，如果根据名称获取字段，可以使用Tuple中的默认字段名称。

```
//通过scala Tuple创建具有两个元素的数据集
val tupleStream: DataStream[Tuple2[String, Int]] =
env.fromElements(("a", 1),
  ("c", 2))
//使用默认字段名称获取字段，其中_1表示tuple这种第一个字段
tupleStream.keyBy("_1")
```

4. POJOs类型

POJOs类可以完成复杂数据结构的定义，Flink通过实现PojoTypeInfo来描述任意的POJOs，包括Java和Scala类。在Flink中使用POJOs类可以通过字段名称获取字段，例如dataStream.join(otherStream).where("name").equalTo("personName")，对于用户做数据处理则非常透明和简单，如代码清单3-2所示。如果在Flink中使用POJOs数据类型，需要遵循以下要求：

· POJOs类必须是Public修饰且必须独立定义，不能是内部类；

· POJOs类中必须含有默认空构造器；

· POJOs类中所有的Fields必须是Public或者具有Public修饰的getter和setter方法；

· POJOs类中的字段类型必须是Flink支持的。

代码清单3-2 Java POJOs数据类型定义

```
//定义Java Person类，具有public修饰符
public class Person {
  //字段具有public修饰符
    public String name;
    public int age;
  //具有默认空构造器
    public Person() {
    }
    public Person(String name, int age) {
      this.name = name;
      this.age = age;
    }
}
```

定义好POJOs Class后，就可以在Flink环境中使用了，如下代码所示，使用fromElements接口构建Person类的数据集。POJOs类仅支持字段名称指定字段，如代码中通过Person name来指定Keyby字段。

```
val persionStream = env.fromElements(new Person("Peter",14),new
Person("Linda",25))
//通过Person.name来指定Keyby字段
persionStream.keyBy("name")
Scala POJOs数据结构定义如下，使用方式与Java POJOs相同。
class Person(var name: String, var age: Int) {
    //默认空构造器
      def this() {
        this(null, -1)
  }
}
```

5.Flink Value类型

Value数据类型实现了org.apache.flink.types.Value，其中包括read()和write()两个方法完成序列化和反序列化操作，相对于通用的序列化工具会有着比较高效的性能。目前Flink提供了内建的Value类型有IntValue、DoubleValue以及StringValue等，用户可以结合原生数据类型和Value类型使用。

6.特殊数据类型

在Flink中也支持一些比较特殊的数据数据类型，例如Scala中的List、Map、Either、Option、Try数据类型，以及Java中Either数据类型，还有Hadoop的Writable数据类型。如下代码所示，创建Map和List类型数据集。这种数据类型使用场景不是特别广泛，主要原因是数据中的操作相对不像POJOs类那样方便和透明，用户无法根据字段位置或者名称获取字段信息，同时要借助Types Hint帮助Flink推断数据类型信息，关于Tyeps Hmt介绍可以参考下一小节。

```
//创建Map类型数据集
val mapStream =
env.fromElements(Map("name"->"Peter","age"->18),Map("name"-
>"Linda",
"age"->25))
//创建List类型数据集
val listStream = env.fromElements(List(1,2,3,5),List(2,4,3,2))
```

3.4.2 TypeInformation信息获取

通常情况下Flink都能正常进行数据类型推断，并选择合适的serializers以及comparators。但在某些情况下却无法直接做到，例如定义函数时如果使用到了泛型，JVM就会出现类型擦除的问题，使得Flink并不能很容易地获取到数据集中的数据类型信息。同时在Scala API和Java API中，Flink分别使用了不同的方式重构了数据类型信息。

1.Scala API类型信息

Scala API通过使用Manifest和类标签，在编译器运行时获取类型信息，即使是在函数定义中使用了泛型，也不会像Java API出现类型擦除的问题，这使得Scala API具有非常精密的类型管理机制。同时在Flink中使用到Scala Macros框架，在编译代码的过程中推断函数输入参数和返回值的类型信息，同时在Flink中注册成TypeInformation以支持上层计算算子使用。

当使用Scala API开发Flink应用，如果使用到Flink已经通过TypeInformation定义的数据类型，TypeInformation类不会自动创建，而是使用隐式参数的方式引入，代码不会直接抛出编码异常，但是当启动Flink应用程序时就会报“could not find implicit value for evidence parameter of type TypeInformation”的错误。这时需要将TypeInformation类隐式参数引入到当前程序环境中，代码实例如下：

```
import org.apache.flink.api.scala._
```

2.Java API类型信息

由于Java的泛型会出现类型擦除问题，Flink通过Java反射机制尽可能重构类型信息，例如使用函数签名以及子类的信息等。同时类型推断在当输出类型依赖于输入参数类型时相对比较容易做到，但是如果函数的输出类型不依赖于输入参数的类型信息，这个时候就需要借

助于类型提示（Ctype Himts）来告诉系统函数中传入的参数类型信息和输出参数信息。如代码清单3-3通过在returns方法中传入TypeHint<Integer>实例指定输出参数类型，帮助Flink系统对输出类型进行数据类型参数的推断和收集。

代码清单3-3　定义Type Hint输出类型参数

```
DataStream<Integer> typeStream = input
  .flatMap(new MyMapFunction<String, Integer>())
  .returns(new TypeHint<Integer>() {//通过returns方法指定返回参数类型
    });
//定义泛型函数，输入参数类型为<T,O>,输出参数类型为O
  class MyMapFunction<T, O> implements MapFunction<T, O> {
    public void flatMap(T value, Collector<O> out) {
      //定义计算逻辑
    }
  }
```

在使用Java API定义POJOs类型数据时，PojoTypeInformation为POJOs类中的所有字段创建序列化器，对于标准的类型，例如Integer、String、Long等类型是通过Flink自带的序列化器进行数据序列化，对于其他类型数据都是直接调用Kryo序列化工具来进行序列化。

通常情况下，如果Kryo序列化工具无法对POJOs类序列化时，可以使用Avro对POJOs类进行序列化，如下代码通过在ExecutionConfig中调用enableForceAvro()来开启Avro序列化。

```
ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
//开启Avro序列化方式
env.getConfig().enableForceAvro();
```

如果用户想使用Kryo序列化工具来序列化POJOs所有字段，则在ExecutionConfig中调用enableForceKryo()来开启Kryo序列化。

```
final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
env.getConfig().enableForceKryo();
```

如果默认的Kryo序列化类不能序列化POJOs对象，通过调用ExecutionConfig的addDefault-KryoSerializer()方法向Kryo中添加自定义的序列化器。

```
env.getConfig().addDefaultKryoSerializer(Class<?> type, Class<?
extends
Serializer<?>> serializerClass)
```

3.自定义TypeInformation

除了使用已有的TypeInformation所定义的数据格式类型之外，用户也可以自定义实现TypeInformation，来满足的不同的数据类型定义需求。Flink提供了可插拔的Type Information Factory让用户将自定义的TypeInformation注册到Flink类型系统中。如下代码所示只需要通过实现org.apache.flink.api.common.typeinfo.TypeInfoFactory接口，返回相应的类型信息。

· 通过@TypeInfo注解创建数据类型，定义CustomTuple数据类型。

```
@TypeInfo(CustomTypeInfoFactory.class)
public class CustomTuple<T0, T1> {
  public T0 field0;
  public T1 field1;
}
```

· 然后定义CustomTypeInfoFactory类继承于TypeInfoFactory，参数类型指定CustomTuple。最后重写createTypeInfo方法，创建的CustomTupleTypeInfo就是CustomTuple数据类型TypeInformation。

```
public class CustomTypeInfoFactory extends
TypeInfoFactory<CustomTuple> {
  @Override
  public TypeInformation<CustomTuple> createTypeInfo(Type t,
Map<String, TypeInformation<?>> genericParameters) {
    return new CustomTupleTypeInfo(genericParameters.get("T0"),
genericParameters.get("T1"));
  }
}
```

3.5 本章小结

本章对Flink编程中模型进行了介绍，在3.1节中介绍了Flink支持的数据集类型，以及有界数据集合无界数据集之间的关系等。3.2节对Flink编程接口进行了介绍与说明，分别介绍了Flink在不同层面的API及相应的使用，使读者能够从接口层面对Flink有一个比较清晰的认识和了解。在3.3节针对Flink程序结构进行了说明，介绍了在编写Flink程序中遵循的基本模式。最后对Flink中支持的数据结构类型进行介绍，包括使用Java的POJOS对象、原始数据类型等。接下来的章节我们将更加深入地介绍Flink在流式计算和批量计算领域中的对应接口使用方式，让读者对Flink编程有更加深入的掌握和理解。

第4章

DataStream API介绍与使用

本章将重点介绍如何利用DataStream API开发流式应用，其中包括基本的编程模型、常用操作、时间概念、窗口计算、作业链等。4.1节将介绍在使用DataStream接口编程中的基本操作，例如如何定义数据源、数据转换、数据输出等操作，以及每种操作在Flink中如何进行拓展。4.2节将重点介绍Flink在流式计算过程中，对时间概念的区分和使用，其中包括事件时间（Event Time）、注入时间（Ingestion Time）、处理时间（Process Time）等时间概念，其中在事件时间中，会涉及Watermark等概念的解释和说明，帮助用户如何通过使用水印技术处理乱序数据。4.3节将介绍Flink在流式计算中常见的窗口计算类型，如滚动窗口、滑动窗口、会话窗口等，以及每种窗口的使用和应用场景。4.4节将介绍Flink应用通过使用作业链条操作，对Flink的任务进行优化，保证资源的合理利用。

4.1 DataStream编程模型

在Flink整个系统架构中，对流计算的支持是其最重要的功能之一，Flink基于Google提出的DataFlow模型，实现了支持原生数据流处理的计算引擎。Flink中定义了DataStream API让用户灵活且高效地编写Flink流式应用。DataStream API主要可为分为三个部分，DataSource模块、Transformation模块以及DataSink模块，其中Sources模块主要定义了数据接入功能，主要是将各种外部数据接入至Flink系统中，并将接入数据转换成对应的DataStream数据集。在Transformation模块定义了对DataStream数据集的各种转换操作，例如进行map、filter、windows等操作。最后，将结果数据通过DataSink模块写出到外部存储介质中，例如将数据输出到文件或Kafka消息中间件等。

4.1.1 DataSources数据输入

DataSources模块定义了DataStream API中的数据输入操作，Flink将数据源主要分为的内置数据源和第三方数据源这两种类型。其中内置数据源包含文件、Socket网络端口以及集合类型数据，其不需要引入其他依赖库，且在Flink系统内部已经实现，用户可以直接调用相关方法使用。第三方数据源定义了Flink和外部系统数据交互的逻辑，包括数据的读写接口。在Flink中定义了非常丰富的第三方数据源连接器（Connector），例如Apache kafka Connector、Elatic Search Connector等。同时用户也可以自定义实现Flink中数据接入函数SourceFunction，并封装成第三方数据源的Connector，完成Flink与其他外部系统的数据交互。

1.内置数据源

（1）文件数据源

Flink系统支持将文件内容读取到系统中，并转换成分布式数据集DataStream进行数据处理。在StreamExecutionEnvironment中，可以使用readTextFile方法直接读取文本文件，也可以使用readFile方法通过指定文件InputFormat来读取特定数据类型的文件，其中InputFormat可以是系统已经定义的InputFormat类，如CsvInputFormat等，也可以用户自定义实现InputFormat接口类。代码清单4-1分别描述了直接读取文本文件和使用CSVInputFormat读取CSV文件。

代码清单4-1　实现File-Based输入流

```
  //直接读取文本文件
  val textStream =
env.readTextFile("/user/local/data_example.log")
  //通过指定CSVInputFormat读取CSV文件
  val csvStream = env.readFile(new CsvInputFormat[String](new
Path("/user/local/data_example.csv")) {
    override def fillRecord(out: String, objects: Array[AnyRef]):
String = {
    return null
```

```
    }
}, "/user/local/data_example.csv")
```

在DataStream API中，可以在readFile方法中指定文件读取类型（WatchType）、检测文件变换时间间隔（interval）、文件路径过滤条件（FilePathFilter）等参数，其中WatchType共分为两种模式——PROCESS_CONTINUOUSLY和PROCESS_ONCE模式。在PROCESS_CONTINUOUSLY模式下，一旦检测到文件内容发生变化，Flink会将该文件全部内容加载到Flink系统中进行处理。而在PROCESS_ONCE模式下，当文件内容发生变化时，只会将变化的数据读取至Flink中，在这种情况下数据只会被读取和处理一次。

 注意

可以看出，在PROCESS_CONTINUOUSLY模式下是无法实现Excatly Once级别数据一致性保障的，而在PROCESS_ONCE模式，可以保证数据Excatly Once级别的一致性保证。但是需要注意的是，如果使用文件作为数据源，当某个节点异常停止的时候，这种情况下Checkpoints不会更新，如果数据一直不断地在生成，将导致该节点数据形成积压，可能需要耗费非常长的时间从最新的checkpoint中恢复应用。

（2）Socket数据源

Flink支持从Socket端口中接入数据，在StreamExecutionEnvironment调用socket-TextStream方法。该方法参数分别为Ip地址和端口，也可以同时传入字符串切割符delimiter和最大尝试次数maxRetry，其中delimiter负责将数据切割成Records数据格式；maxRetry在端口异常的情况，通过指定次数进行重连，如果设定为0，则Flink程序直接停止，不再尝试和端口进行重连。如下代码是使用socketTextStream方法实现了将数据从本地9999端口中接入数据并转换成DataStream数据集的操作。

```
val socketDataStream = env.socketTextStream("localhost", 9999)
```

在Unix系统环境下，可以执行nc -lk 9999命令启动端口，在客户端中输入数据，Flink就可以接收端口中的数据。

（3）集合数据源

Flink可以直接将Java或Scala程序中集合类（Collection）转换成DataStream数据集，本质上是将本地集合中的数据分发到远端并行执行的节点中。目前Flink支持从java.util.Collection和java.util.Iterator序列中转换成DataStream数据集。这种方式非常适合调试Flink本地程序，但需要注意的是，集合内的数据结构类型必须要一致，否则可能会出现数据转换异常。

· 通过fromElements从元素集合中创建DataStream数据集：

```
val dataStream = env.fromElements(Tuple2(1L, 3L), Tuple2(1L, 5L),
Tuple2(1L,
  7L), Tuple2(1L, 4L), Tuple2(1L, 2L))
```

· 通过fromCollection从数组转创建DataStream数据集：

```
String[] elements = new String[]{"hello", "flink"};
DataStream<String> dataStream =
env.fromCollection(Arrays.asList(elements));
```

· 将java.util.List转换成DataStream数据集：

```
List<String> arrayList = new ArrayList<>();
  arrayList.add("hello flink");
DataStream<String> dataList = env.fromCollection(arrayList);
```

2. 外部数据源

（1）数据源连接器

前面提到的数据源类型都是一些基本的数据接入方式，例如从文件、Socket端口中接入数据，其实质是实现了不同的SourceFunction，Flink将其封装成高级API，减少了用户的使用成本。对于流式计算类型的应用，数据大部分都是从外部第三方系统中获取，为此Flink通过实现SourceFunction定义了非常丰富的第三方数据连接器，基本覆盖了大部分的高性能存储介质以及中间件等，其中部分连接器是仅支持读取数据，例如Twitter Streaming API、Netty等；另外一部分仅支持数据输出（Sink），不支持数据输入（Source），例如Apache Cassandra、Elasticsearch、Hadoop FileSystem等。还有一部分是既支持数据输入，也支持数据输出，例如Apache Kafka、Amazon Kinesis、RabbitMQ等连接器。

以Kafka为例，用户在Maven编译环境中导入如代码清单4-2所示的环境配置，主要因为Flink为了尽可能降低用户在使用Flink进行应用开发时的依赖复杂度，所有第三方连接器依赖配置放置在Flink基本依赖库以外，用户在使用过程中，根据需要将需要用到的Connector依赖库引入到应用工程中即可。

代码清单4-2 Kafka Connector Maven依赖配置

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka-0.8_2.11</artifactId>
  <version>1.7.1</version>
</dependency>
```

在引入Maven依赖配置后，就可以在Flink应用工程中创建和使用相应的Connector，在kafka Connector中主要使用的其中参数有kafka topic、bootstrap.servers、zookeeper.connect。另外Schema参数的主要作用是根据事先定义好的Schema信息将数据序列化成该Schema定义的数据类型，默认是SimpleStringSchema，代表从Kafka中接入的数据将转换成String字符串类型处理，如代码清单4-3所示。

代码清单4-3 Kafka Event DataSource数据接入

```
Properties properties = new Properties();
properties.setProperty("bootstrap.servers", "localhost:9092");
// only required for Kafka 0.8
properties.setProperty("zookeeper.connect", "localhost:2181");
properties.setProperty("group.id", "test");
DataStream<String> input = env
.addSource(
new FlinkKafkaConsumer010<>(
properties.getString("input-data-topic"),
new SimpleStringSchema(),properties);
```

用户通过自定义Schema将接入数据转换成制定数据结构，主要是实现Deserialization-Schema接口来完成，代码清单4-4说明了KafkaEventSchema的定义。可以看到在SourceEventSchema代码中，通过实现deserialize方法完成数据从byte[]数据类型转换成SourceEvent的反序列化操作，以及通过实现getProducedType方法将数据类型转换成Flink系统所支持的数据类型，例如以下列代码中的TypeInformation<SourceEvent>类型。

代码清单4-4　DeserializationSchema实例

```
public class SourceEventSchema implements
DeserializationSchema<SourceEvent>{
  private static final long serialVersionUID =
6154188370191669789L;
  @Override
  public SourceEvent deserialize(byte[] message) throws
IOException {
    return SourceEvent.fromString(new String(message));
  }
  @Override
  public boolean isEndOfStream(SourceEvent nextElement) {
    return false;
  }
  @Override
  public TypeInformation< SourceEvent > getProducedType() {
    return TypeInformation.of(SourceEvent.class);
  }
}
```

针对Kafka数据的解析，Flink提供了KeyedDeserializationSchema，其中deserialize方法定义为T deserialize(byte[] messageKey, byte[] message, String topic, int partition, long offset)，支持将Message中的key和value同时解析出来。

同时为了更方便地解析各种序列化类型的数据，Flink内部提供了常用的序列化协议的Schema，例如TypeInformationSerializationSchema、JsonDeserializationSchema和AvroDeserializationSchema等，用户可以根据需要选择使用。

（2）自定义数据源连接器

Flink中已经实现了大多数主流的数据源连接器，但需要注意，Flink的整体架构非常开放，用户也可以自己定义连接器，以满足不同的数据源的接入需求。可以通过实现SourceFunction定义单个线程的接入的数据接入器，也可以通过实现ParallelSource-Function接口或继承RichParallelSourceFunction类定义并发数据源接入器。DataSoures定义完成后，可以通过使用SteamExecutionEnvironment的addSources方法添加数据源，这样就可以将外部系统中的数据转换成DataStream[T]数据集合，其中T类型是Source-Function返回值类型，然后就可以完成各种流式数据的转换操作。

4.1.2 DataSteam转换操作

即通过从一个或多个DataStream生成新的DataStream的过程被称为Transformation操作。在转换过程中，每种操作类型被定义为不同的Operator，Flink程序能够将多个Transformation组成一个DataFlow的拓扑。所有DataStream的转换操作可分为单Single-DataStream、Multi-DaataStream、物理分区三类类型。其中Single-DataStream操作定义了对单个DataStream数据集元素的处理逻辑，Multi-DataStream操作定义了对多个DataStream数据集元素的处理逻辑。物理分区定义了对数据集中的并行度和数据分区调整转换的处理逻辑。

1.Single-DataStream操作

（1）Map [DataStream->DataStream]

调用用户定义的MapFunction对DataStream[T]数据进行处理，形成新的Data-Stream[T]，其中数据格式可能会发生变化，常用作对数据集内数据的清洗和转换。例如将输入数据集中的每个数值全部加1处理，并且将数据输出到下游数据集。

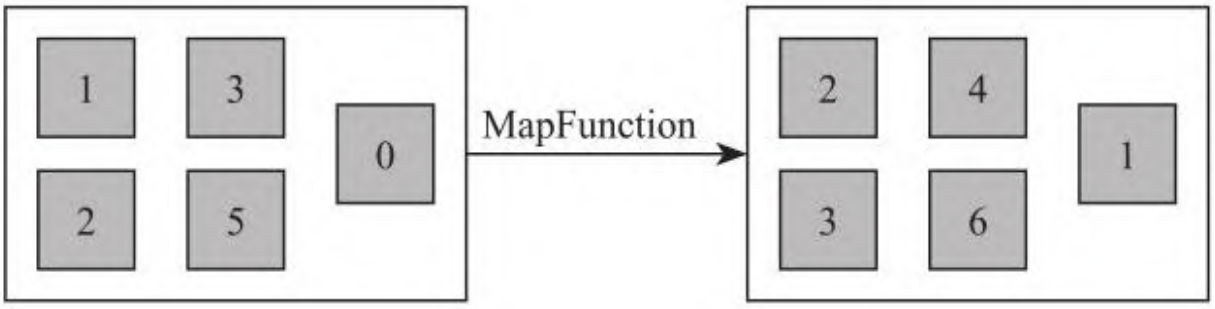

图4-1 DataStream MapFunction定义

图4-1中计算逻辑实现代码如下所示，通过从集合中创建dataStream，并调用DataStream的map方法传入计算表达式，完成对第二个字段加1操作，最后得到新的数据集mapStream。

```
val dataStream = env.fromElements(("a", 3), ("d", 4), ("c", 2),
("c", 5), ("a",
```

```
  5))
  //指定map计算表达式
val mapStream: DataStream[(String, Int)] = dataStream.map(t =>
(t._1, t._2 + 1))
```

除了可以在map方法中直接传入计算表达式，如下代码实现了MapFunction接口定义map函数逻辑，完成数据处理操作。其中MapFunction[(String, Int), (String, Int)]中共有两个参数，第一个参数(String, Int)代表输入数据集数据类型，第二个参数(String, Int)代表输出数据集数据类型。

```
//通过指定MapFunction
val mapStream: DataStream[(String, Int)] = dataStream.map(new
MapFunction[(String, Int), (String, Int)] {
  override def map(t: (String, Int)): (String, Int) = {
    (t._1, t._2 + 1)}
  })
```

以上两种方式得到的结果一样，但是第二种方式在使用Java语言的时候用得较多，用户可以根据自己的需要偏好使用。

（2）FlatMap [DataStream->DataStream]

该算子主要应用处理输入一个元素产生一个或者多个元素的计算场景，比较常见的是在经典例子WordCount中，将每一行的文本数据切割，生成单词序列如在图4-2中对于输入DataStream[String]通过FlatMap函数进行处理，字符串数字按逗号切割，然后形成新的整数数据集。

图4-2 DataStream FlatMapFunction定义

针对上述计算逻辑实现代码如下所示，通过调用resultStream接口中flatMap方法将定义好的FlatMapFunction传入，生成新的数据集。FlatMapFunction的接口定义为FlatMapFunction[T, O] { flatMap(T, Collector[O]): Unit }其中T为输入数据集的元素格式，O为输出数据集的元素格式。

```
val dataStream:DataStream[String] = environment.fromCollections()
val resultStream[String] = dataStream.flatMap { str => str.split(" ") }
```

（3）Filter [DataStream->DataStream]

该算子将按照条件对输入数据集进行筛选操作，将符合条件的数据集输出，将不符合条件的数据过滤掉。如图4-3所示将输入数据集中偶数过滤出来，奇数从数据集中去除。

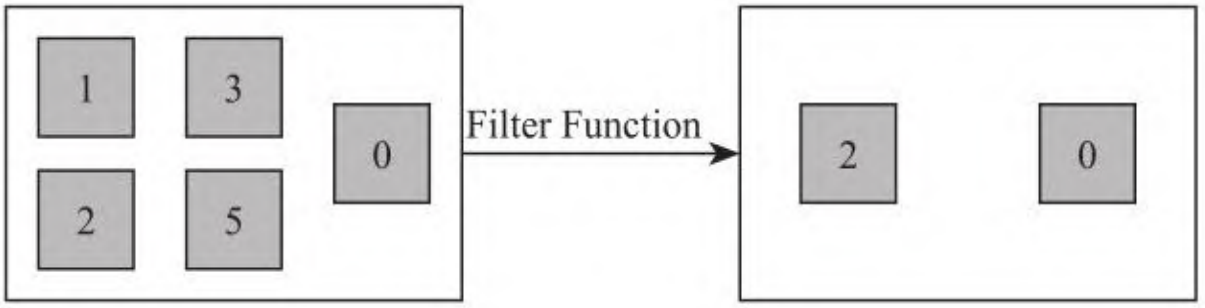

图4-3 DataStream根据奇偶性定义FilterFunction操作

针对图4-3中的计算逻辑代码实现如下，可以使用Scala通配符下划线，用_代替参数名，或者直接使用Scala Lambada表达式，两种方式都是可以的。

```
//通过通配符
val filter:DataStream[Int] = dataStream.filter { _ % 2 == 0 }
```

```
//或者指定运算表达式
val filter:DataStream[Int] = dataStream.filter { x => x % 2 == 0 }
```

（4）KeyBy [DataStream->KeyedStream]

该算子根据指定的Key将输入的DataStream[T]数据格式转换为KeyedStream[T]，也就是在数据集中执行Partition操作，将相同的Key值的数据放置在相同的分区中。如图4-4所示，将白色方块和灰色方块通过颜色的Key值重新分区，将数据集分为具有灰色方块的数据集合。

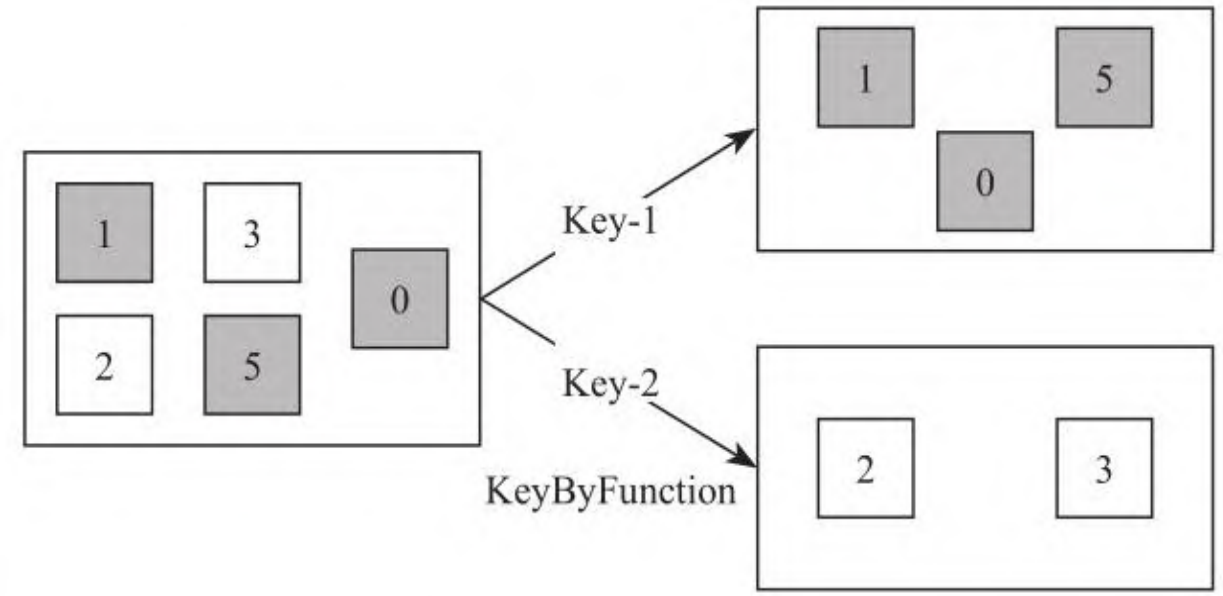

图4-4 DataStream根据颜色进行KeyByFunction操作

如下代码所示，将数据集中第一个参数作为Key，对数据集进行KeyBy函数操作，形成根据id分区的KeyedStream数据集。其中keyBy方法输入为DataStream[T]数据集。

```
val dataStream = env.fromElements((1, 5),(2, 2),(2, 4),(1, 3))
//指定第一个字段为分区Key
val keyedStream: KeyedStream[(String,Int), Tuple] =
dataStream.keyBy(0)
```

注意

以下两种数据类型将不能使用KeyBy方法对数据集进行重分区：1）用户使用POJOs类型数据，但是POJOs类中没有复写hashCode()方法，而是依赖于Object.hasCode()；2）任何数据类型的数组结构。

（5）Reduce [KeyedStream->DataStream]

该算子和MapReduce中Reduce原理基本一致，主要目的是将输入的KeyedStream通过传入的用户自定义的ReduceFunction滚动地进行数据聚合处理，其中定义的ReduceFunciton必须满足运算结合律和交换律。如下代码对传入keyedStream数据集中相同的key值的数据独立进行求和运算，得到每个key所对应的求和值。

```
val dataStream = env.fromElements(("a", 3), ("d", 4), ("c", 2),
("c",
5), ("a", 5))
  //指定第一个字段为分区Key
val keyedStream: KeyedStream[(String,Int), Tuple] =
dataStream.keyBy(0)
//滚动对第二个字段进行reduce相加求和
val reduceStream = keyedStream.reduce { (t1, t2) =>
    (t1._1, t1._2 + t2._2)
}
```

用户也可以单独定义Reduce函数，如下代码所示：

```
//通过实现ReduceFunction匿名类
val reduceStream1 = keyedStream.reduce(new ReduceFunction[(String,
Int)] {
    override def reduce(t1: (String, Int), t2: (String, Int)):
(String, Int)={
    (t1._1, t1._2 + t2._2)
  }})
```

运行代码的输出结果依次为：（c，2）（c，7）（a，3）（d，4）（a，8）。

（6）Aggregations[KeyedStream->DataStream]

Aggregations是DataStream接口提供的聚合算子，根据指定的字段进行聚合操作，滚动地产生一系列数据聚合结果。其实是将Reduce算子中的函数进行了封装，封装的聚合操作有sum、min、minBy、max、maxBy等，这样就不需要用户自己定义Reduce函数。如下代码所示，指定数据集中第一个字段作为key，用第二个字段作为累加字段，然后滚动地对第二个字段的数值进行累加并输出。

```
//指定第一个字段为分区Key
val keyedStream: KeyedStream[(Int, Int), Tuple] =
dataStream.keyBy(0)
//对第二个字段进行sum统计
val sumStream: DataStream[(Int, Int)] = keyedStream.sum(1)
//输出计算结果
sumStream.print()
```

代码执行完毕后结果输出在客户端，其中key为1的统计结果为（1，5）和（1，8），key为2的统计结果为（2，2）和（2，6）。可以看出，计算出来的统计值并不是一次将最终整个数据集的最后求和结果输出，而是将每条记录所叠加的结果输出。

聚合函数中需要传入的字段类型必须是数值型，否则会抛出异常。对应其他的聚合函数的用法如下代码所示。

```
val minStream: DataStream[(Int, Int)] = keyedStream.min(1)
//滚动计算指定key的最大值
val maxStream: DataStream[(Int, Int)] = keyedStream.max(1)
//滚动计算指定key的最小值，返回最大值对应的元素
val minByStream: DataStream[(Int, Int)] = keyedStream.minBy(1)
//滚动计算指定key的最大值，返回最大值对应的元素
val maxByStream: DataStream[(Int, Int)] = keyedStream.maxBy(1)
```

2. Multi-DataStream操作

（1）Union[DataStream ->DataStream]

Union算子主要是将两个或者多个输入的数据集合并成一个数据集，需要保证两个数据集的格式一致，输出的数据集的格式和输入的数据集格式保持一致，如图4-5所示，将灰色方块数据集和黑色方块数据集合并成一个大的数据集。

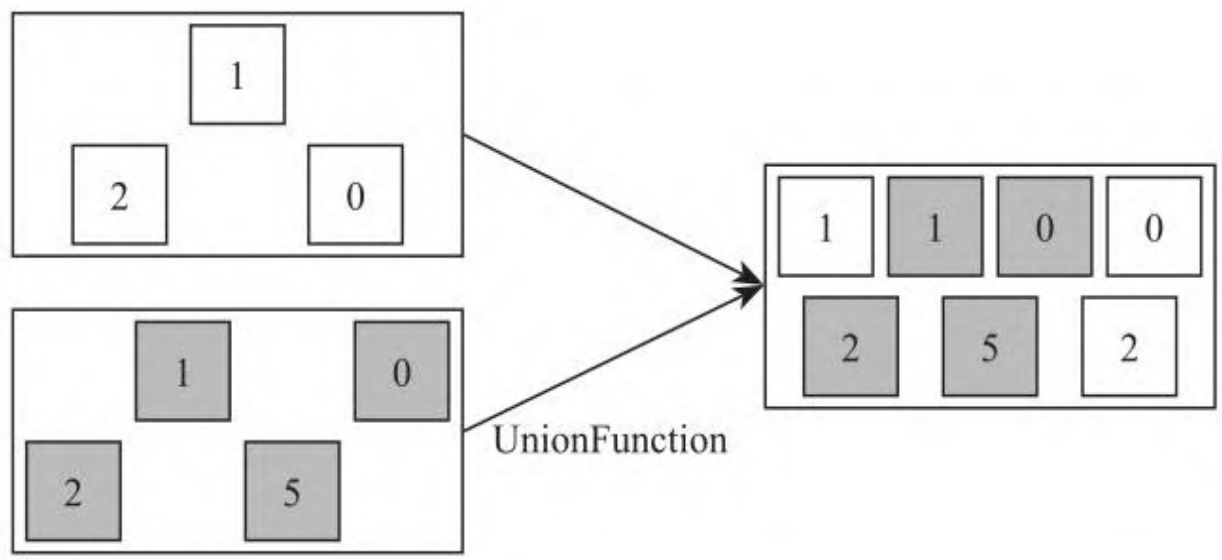

图4-5　DataStream将不同颜色的数据集进行Union操作

可以直接调用DataStream API中的union()方法来合并多个数据集，方法中传入需要合并的DataStream数据集。如下代码所示，分别将创建的数据集dataStream_01和dataStream_02合并，如果想将多个数据集同时合并则在union()方法中传入被合并的数据集的序列即可。

```
//创建不同的数据集
val dataStream1: DataStream[(String, Int)] = env.fromElements(("a",
3), ("d", 4), ("c", 2), ("c", 5), ("a", 5))
val dataStream2: DataStream[(String, Int)] = env.fromElements(("d",
1), ("s", 2), ("a", 4), ("e", 5), ("a", 6))
val dataStream3: DataStream[(String, Int)] = env.fromElements(("a",
2), ("d", 1), ("s", 2), ("c", 3), ("b", 1))
//合并两个DataStream数据集
val unionStream = dataStream1.union(dataStream_02)
```

```
//合并多个DataStream数据集
val allUnionStream = dataStream1.union(dataStream2, dataStream3)
```

（2）Connect，CoMap，CoFlatMap[DataStream ->DataStream]

Connect算子主要是为了合并两种或者多种不同数据类型的数据集，合并后会保留原来数据集的数据类型。连接操作允许共享状态数据，也就是说在多个数据集之间可以操作和查看对方数据集的状态，关于状态操作将会在后续章节中重点介绍。如下代码所示，dataStream1数据集为(String, Int)元祖类型，dataStream2数据集为Int类型，通过connect连接算子将两个不同数据类型的算子结合在一起，形成格式为ConnectedStreams的数据集，其内部数据为[(String, Int), Int]的混合数据类型，保留了两个原始数据集的数据类型。

```
//创建不同数据类型的数据集
val dataStream1: DataStream[(String, Int)] = env.fromElements(("a",
3), ("d", 4), ("c", 2), ("c", 5), ("a", 5))
val dataStream2: DataStream[Int] = env.fromElements(1, 2, 4, 5, 6)
//连接两个DataStream数据集
val connectedStream: ConnectedStreams[(String, Int), Int] =
dataStream1.connect(dataStream2)
```

需要注意的是，对于ConnectedStreams类型的数据集不能直接进行类似Print()的操作，需要再转换成DataStream类型数据集，在Flink中ConnectedStreams提供的map()方法和flatMap() 需要定义CoMapFunciton或CoFlatMapFunction分别处理输入的DataStream数据集，或者直接传入两个MapFunction来分别处理两个数据集。如下代码所示，通过定义CoMapFunction处理ConnectedStreams数据集中的数据，指定的参数类型有三个，其中（String，Int）和Int分别指定的是第一个和第二个数据集的数据类型，（Int，String）指定的是输出数据集的数据类型，在函数定义中需要实现map1和map2两个方法，分别处理输入两个数据集，同时两个方法返回的数据类型必须一致。

```
val resultStream = connectedStream.map(new CoMapFunction[(String,
Int), Int,
  (Int, String)] {
```

```
//定义第一个数据集函数处理逻辑，输入值为第一个DataSteam
    override def map1(in1: (String, Int)): (Int, String) = {
      (in1._2, in1._1)
    }
  //定义第二个函数处理逻辑，输入值为第二个DataStream
    override def map2(in2: Int): (Int, String) = {
      (in2, "default")
    }})
```

在以上实例中，两个函数会多线程交替执行产生结果，最终将两个数据集根据定义合并成目标数据集。和CoMapFunction相似，在flatmap()方法中需要指定CoFlatMapFunction。如下代码所示，通过实现CoFlatMapFunction接口中flatMap1()方法和flatMap2()方法，分别对两个数据集进行处理，同时可以在两个函数之间共享number变量，完成两个数据集的数据合并整合。

```
val resultStream2 = connectedStream.flatMap(new
CoFlatMapFunction[(String,
  Int), Int, (String, Int, Int)] {
    //定义共享变量
    var number = 0
    //定义第一个数据集处理函数
    override def flatMap1(in1: (String, Int), collector:
Collector[(String, Int, Int)]): Unit = {
      collector.collect((in1._1, in1._2, number))
    }
    //定义第二个数据集处理函数
    override def flatMap2(in2: Int, collector: Collector[(String,
Int, Int)]): Unit = {
      number = in2
    }
}
)
```

通常情况下，上述CoMapFunction或者CoFlatMapFunction函数并不能有效地解决数据集关联的问题，产生的结果可能也不是用户想使用的，因为用户可能想通过指定的条件对两个数据集进行关联，然后产生相关性比较强的结果数据集。这个时候就需要借助keyBy函数或broadcast广播变量实现。

```
// 通过keyby函数根据指定的key连接两个数据集
val keyedConnect: ConnectedStreams[(String, Int), Int] =
dataStream1.connect(dataStream2).keyBy(1, 0)
// 通过broadcast关联两个数据集
val broadcastConnect: BroadcastConnectedStream[(String, Int), Int]
= dataStream1.connect(dataStream2.broadcast())
```

通过使用keyby函数会将相同的key的数据路由在一个相同的Operator中，而BroadCast广播变量会在执行计算逻辑之前将dataStream2数据集广播到所有并行计算的Operator中，这样就能够根据条件对数据集进行关联，这其实也是分布式Join算子的基本实现方式。

 注意

CoMapFunction和CoFlaMapFunction中的两个方法，在Paralism>1的情况下，不会按照指定的顺序指定，因此有可能会影响输出数据的顺序和结果，这点用户在使用过程中需要注意。

（3）Split [DataStream->SplitStream]

Split算子是将一个DataStream数据集按照条件进行拆分，形成两个数据集的过程，也是union算子的逆向实现。每个接入的数据都会被路由到一个或者多个输出数据集中。如图4-6所示，将输入数据集根据颜色切分成两个数据集。

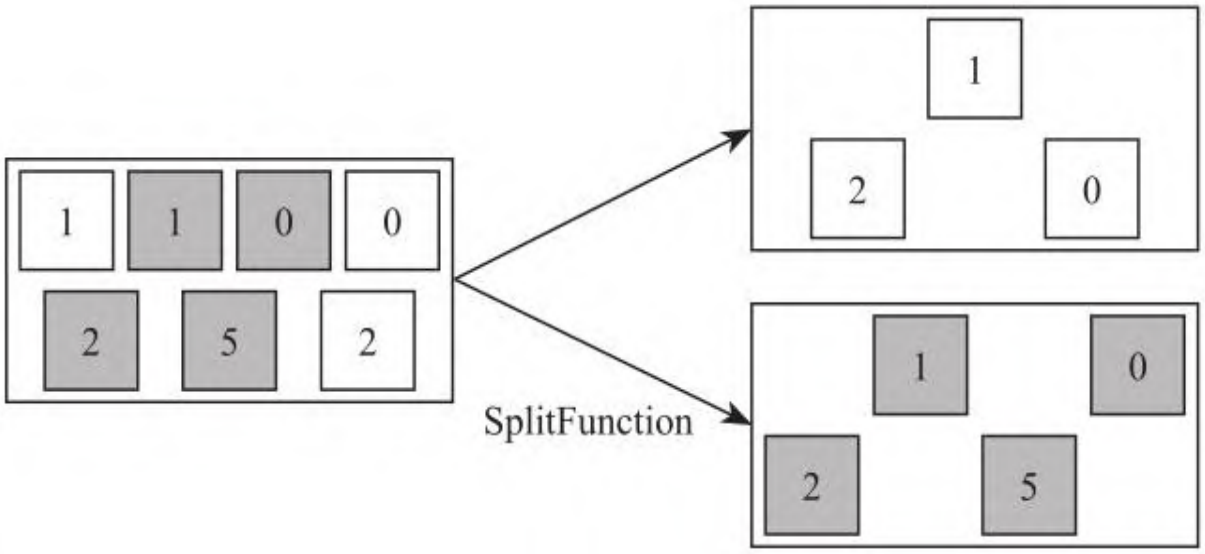

图4-6　通过Split对DataStream数据集进行切分

在使用split函数中，需要定义split函数中的切分逻辑，如下代码所示，通过调用split函数，然后指定条件判断函数，将根据第二个字段的奇偶性将数据集标记出来，如果是偶数则标记为even，如果是奇数则标记为odd，然后通过集合将标记返回，最终生成格式SplitStream的数据集。

```
//创建数据集
val dataStream1: DataStream[(String, Int)] = env.fromElements(("a", 3), ("d", 4), ("c", 2), ("c", 5), ("a", 5))
//合并两个DataStream数据集
val splitedStream: SplitStream[(String, Int)] = dataStream1.split(t => if (t._2 % 2 == 0) Seq("even") else Seq("odd"))
```

（4）Select [SplitStream ->DataStream]

split函数本身只是对输入数据集进行标记，并没有将数据集真正的实现切分，因此需要借助Select函数根据标记将数据切分成不同的数据集。如下代码所示，通过调用SplitStream数据集的select()方法，传入前面已经标记好的标签信息，然后将符合条件的数据筛选出来，形成新的数据集。

```
//筛选出偶数数据集
val evenStream: DataStream[(String, Int)] =
splitedStream.select("even")
//筛选出奇数数据集
val oddStream: DataStream[(String, Int)] =
splitedStream.select("odd")
//筛选出奇数和偶数数据集
val allStream: DataStream[(String, Int)] =
splitedStream.select("even", "odd")
```

（5）Iterate[DataStream->IterativeStream->DataStream]

Iterate算子适合于迭代计算场景，通过每一次的迭代计算，并将计算结果反馈到下一次迭代计算中。如下代码所示，调用dataStream的iterate()方法对数据集进行迭代操作，如果事件指标加1后等于2，则将计算指标反馈到下一次迭代的通道中，如果事件指标加1不等于2则直接输出到下游DataStream中。其中在执行之前需要对数据集做map处理主要目的是为了对数据分区根据默认并行度进行重平衡，在iterate()内参数类型为ConnectedStreams，然后调用ConnectedStreams的方法内分别执行两个map方法，第一个map方法执行反馈操作，第二个map函数将数据输出到下游数据集。

```
val dataStream = env.fromElements(3, 1, 2, 1, 5).map { t: Int => t
}
val iterated = dataStream.iterate((input: ConnectedStreams[Int,
String]) => {
//分别定义两个map方法完成对输入ConnectedStreams数据集数据的处理
    val head = input.map(i => (i + 1).toString, s => s)
    (head.filter(_ == "2"), head.filter(_ != "2"))
  }, 1000)//1000指定最长迭代等待时间，单位为ms，超过该时间没有数据接入则终止迭代
```

---

3. 物理分区操作

物理分区（Physical Partitioning）操作的作用是根据指定的分区策略将数据重新分配到不同节点的Task案例上执行。当使用DataStream提供的API对数据处理过程中，依赖于算子本身对数据的分区控制，如果用户希望自己控制数据分区，例如当数据发生了数据倾

斜的时候，就需要通过定义物理分区策略的方式对数据集进行重新分布处理。Flink中已经提供了常见的分区策略，例如随机分区（Random Partitioning）、平衡分区（Roundobin Partitioning）、按比例分区（Roundrobin Partitioning）等。当然如果给定的分区策略无法满足需求，也可以根据Flink提供的分区控制接口创建分区器，实现自定义分区控制。

Flink内部提供的常见数据重分区策略如下所述。

（1）随机分区（Random Partitioning）：[DataStream ->DataStream]

通过随机的方式将数据分配在下游算子的每个分区中，分区相对均衡，但是较容易失去原有数据的分区结构。

```
//通过调用DataStream API中的shuffle方法实现数据集的随机分区
  val shuffleStream = dataStream.shuffle
```

（2）Roundrobin Partitioning：[DataStream ->DataStream]

通过循环的方式对数据集中的数据进行重分区，能够尽可能保证每个分区的数据平衡，当数据集发生数据倾斜的时候使用这种策略就是比较有效的优化方法。

```
//通过调用DataStream API中rebalance()方法实现数据的重平衡分区
  val shuffleStream = dataStream.rebalance();
```

（3）Rescaling Partitioning：[DataStream ->DataStream]

和Roundrobin Partitioning一样，Rescaling Partitioning也是一种通过循环的方式进行数据重平衡的分区策略。但是不同的是，当使用Roundrobin Partitioning时，数据会全局性地通过网络介质传输到其他的节点完成数据的重新平衡，而Rescaling Partitioning仅仅会对上下游继承的算子数据进行重平衡，具体的分区主要根据上下游

算子的并行度决定。例如上游算子的并发度为2，下游算子的并发度为4，就会发生上游算子中一个分区的数据按照同等比例将数据路由在下游的固定的两个分区中，另外一个分区同理路由到下游两个分区中。

```
//通过调用DataStream API中rescale()方法实现Rescaling Partitioning操作
  val shuffleStream = dataStream.rescale();
```

（4）广播操作（Broadcasting）：[DataStream ->DataStream]

广播策略将输入的数据集复制到下游算子的并行的Tasks实例中，下游算子中的Tasks可以直接从本地内存中获取广播数据集，不再依赖于网络传输。这种分区策略适合于小数据集，例如当大数据集关联小数据集时，可以通过广播的方式将小数据集分发到算子的每个分区中。

```
//可以通过调用DataStream API 的broadcast()方法实现广播分区
val shuffleStream = dataStream.broadcast();
```

（5）自定义分区（Custom Partitioning）：[DataStream ->DataStream]

除了使用已有的分区器之外，用户也可以实现自定义分区器，然后调用DatSstream API上partitionCustom()方法将创建的分区器应用到数据集上。如以下代码所示自定义分区器代码实现了当字段中包含“flink”关键字的数据放在partition为0的分区中，其余数据随机进行分区的策略，其中num Partitions是从系统中获取的并行度参数。

```
object customPartitioner extends Partitioner[String] {
  //获取随机数生成器
  val r = scala.util.Random
  override def partition(key: String, numPartitions: Int): Int = {
    //定义分区策略，key中如果包含a则放在0分区中，其他情况则根据Partitions
num随机分区
    if (key.contains("flink")) 0 else r.nextInt(numPartitions)
```

```
    }
}
```

自定义分区器定义好之后就可以调用DataSteam API的partitionCustom来应用分区器，第二个参数指定分区器使用到的字段，对于Tuple类型数据，分区字段可以通过字段名称指定，其他类型数据集则通过位置索引指定。

```
//通过数据集字段名称指定分区字段
dataStream.partitionCustom(customPartitioner, "filed_name");
//通过数据集字段索引指定分区字段
dataStream.partitionCustom(customPartitioner, 0);
```

4.1.3 DataSinks数据输出

经过各种数据Transformation操作后，最终形成用户需要的结果数据集。通常情况下，用户希望将结果数据输出在外部存储介质或者传输到下游的消息中间件内，在Flink中将DataStream数据输出到外部系统的过程被定义为DataSink操作。在Flink内部定义的第三方外部系统连接器中，支持数据输出的有Apache Kafka、Apache Cassandra、Kinesis、ElasticSearch、Hadoop FileSystem、RabbitMQ、NIFI等，除了Flink内部支持的第三方数据连接器之外，其他例如Apache Bahir框架也支持了相应的数据连接器，其中包括ActiveMQ、Flume、Redis、Akka、Netty等常用第三方系统。用户使用这些第三方Connector将DataStream数据集写入到外部系统中，需要将第三方连接器的依赖库引入到工程中。

1. 基本数据输出

基本数据输出包含了文件输出、客户端输出、Socket网络端口等，这些输出方法已经在Flink DataStream API中完成定义，使用过程不需要依赖其他第三方的库。如下代码所示，实现将DataStream数据集分别输出在本地文件系统和Socket网络端口。

```
val personStream = env.fromElements(("Alex", 18), ("Peter", 43))
//通过writeAsCsv方法将数据转换成CSV文件输出，并执行输出模式为OVERWRITE
personStream.writeAsCsv("file:///path/to/person.csv",WriteMode.OVERWRITE)
//通过writeAsText方法将数据直接输出到本地文件系统
personStream.writeAsText("file:///path/to/person.txt")
//通过writeToSocket方法将DataStream数据集输出到指定Socket端口
personStream.writeToSocket(outputHost, outputPort, new SimpleStringSchema())
```

2. 第三方数据输出

通常情况下，基于Flink提供的基本数据输出方式并不能完全地满足现实场景的需要，用户一般都会有自己的存储系统，因此需要将Flink系统中计算完成的结果数据通过第三方连接器输出到外部系统

中。Flink中提供了DataSink类操作算子来专门处理数据的输出，所有的数据输出都可以基于实现SinkFunction完成定义。例如在Flink中定义了FlinkKafkaProducer类来完成将数据输出到Kafka的操作，需要根据不同的Kafka版本需要选择不同的FlinkKafkaProducer，目前FlinkKafkaProducer类支持Kafka大于1.0.0的版本，FlinkKafkaProducer11或者010支持Kafka0.10.0.x的版本。如代码清单4-5所示，通过使用FlinkKafkaProducer11将DataStream中的数据写入Kafka的Topic中。

代码清单4-5　通过FlinkKafkaProducer11向Kafka Topic中写入数据

```
val wordStream = env.fromElements("Alex", "Peter", "Linda")
//定义FlinkKafkaProducer011 Sink算子
val kafkaProducer = new FlinkKafkaProducer011[String](
    "localhost:9092", // 指定Broker List参数
    "kafka-topic", // 指定目标Kafka Topic名称
    new SimpleStringSchema) // 设定序列化Schema
  /通过addsink添加kafkaProducer到算子拓扑中
  wordStream.addSink(kafkaProducer)
```

在以上代码中使用FlinkKafkaProducer往Kafka中写入数据的操作相对比较基础，还可以配置一些高级选项，例如可以配置自定义properties类，将自定义的参数通过properties类传入FlinkKafkaProducer中。另外还可以自定义Partitioner将DataStream中的数据按照指定分区策略写入Kafka的分区中。也可以使用KeyedSerializationSchema对序列化Schema进行优化，从而能够实现一个Producer往多个Topic中写入数据的操作。

4.2 时间概念与Watermark

4.2.1 时间概念类型

对于流式数据处理，最大的特点是数据上具有时间的属性特征，Flimk根据时间产生的位置不同，将时间区分为三种时间概念，分别为事件生成时间（Event Time）、事件接入时间（Ingestion Time）和事件处理时间（Processing Time）。如图4-7所示，数据从终端产生，或者从系统中产生的过程中生成的时间为事件生成时间，当数据经过消息中间件传入到Flink系统中，在DataSource中接入的时候会生成事件接入时间，当数据在Flink系统中通过各个算子实例执行转换操作的过程中，算子实例所在系统的时间为数据处理时间。Flink已经支持这三种类型时间概念，用户能够根据需要选择时间类型作为对流式数据的依据，这种情况极大地增强了对事件数据处理的灵活性和准确性。

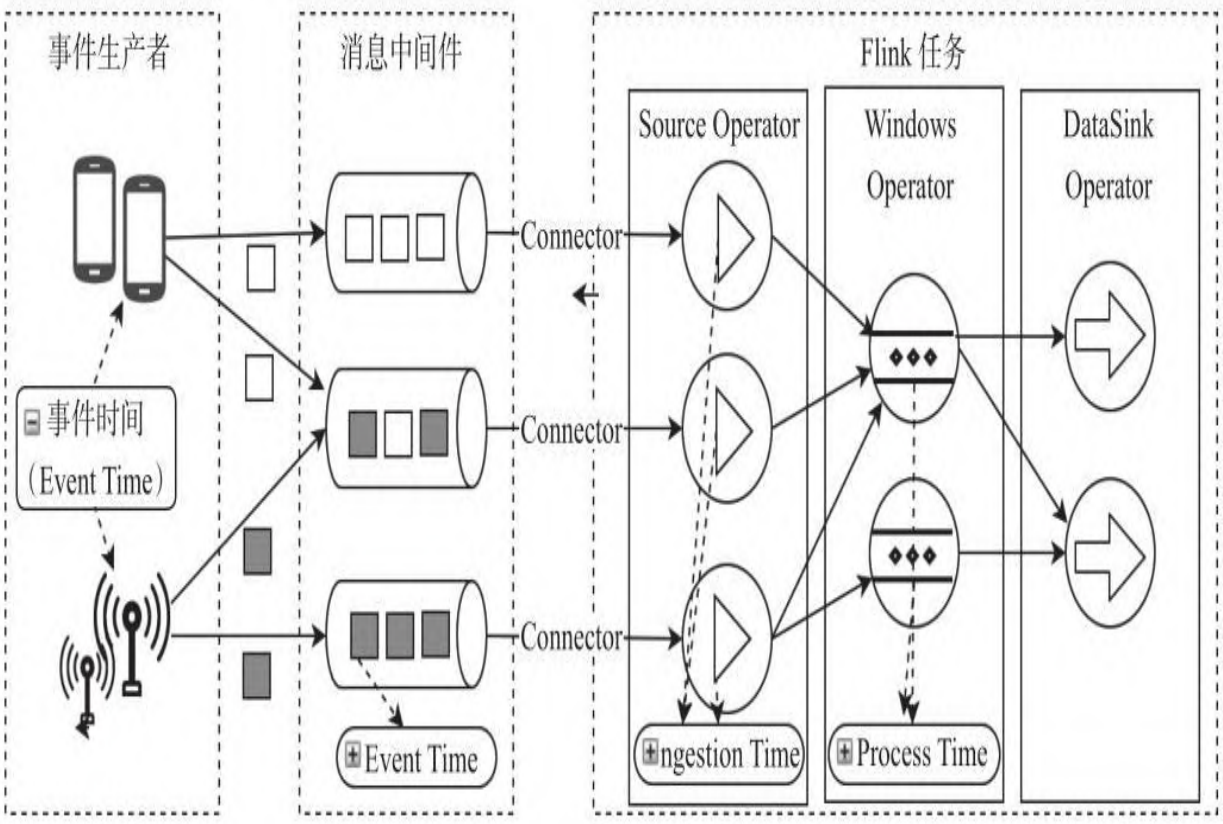

图4-7　Flink EventTime/IngestionTime/ProcessingTime时间概念

1.事件时间（Event Time）

事件时间（Event Time）是每个独立事件在产生它的设备上发生的时间，这个时间通常在事件进入Flink之前就已经嵌入到事件中，时间顺序取决于事件产生的地方，和下游数据处理系统的时间无关。事件数据具有不变的事件时间属性，该时间自事件元素产生就不会改变。通常情况下可以在Flink系统中指定事件时间属性或者设定时间提取器来提取事件时间。

所有进入到Flink流式系统处理的事件，其时间都是在外部系统中产生，经过网络进入到Flink系统内处理的，在理论情况下（所有系统都具有相同系统时钟），事件时间对应的时间戳一定会早于在Flink系统中处理的时间戳，但在实际情况中往往会出现数据记录乱序、延迟到达等问题。基于EventTime的时间概念，数据处理过程依赖于数据本身产生的时间，而不是Flink系统中Operator所在主机节点的系统时间，这样能够借助于事件产生时的时间信息来还原事件的先后关系。

2.接入时间（Ingestion Time）

接入时间（Ingestion Time）是数据进入Flink系统的时间，Ingestion Time依赖于Source Operator所在主机的系统时钟。Ingestion Time介于Event Time和Process Time之间，相对于Process Time，Ingestion Time生成的代价相对较高，Ingestion Time具有一定的可预见性，主要因为Ingestion Time在数据接入过程生成后，时间戳就不再发生变化，和后续数据处理Operator所在机器的时钟没有关系，从而不会因为某台机器时钟不同步或网络时延而导致计算结果不准确的问题。但是需要注意的是相比于Event Time，Ingestion Time不能处理乱序事件，所以也就不用生成对应的Watermarks。

3.处理时间（Processing Time）

处理时间（Processing Time）是指数据在操作算子计算过程中获取到的所在主机时间。当用户选择使用Processing Time时，所有和时间相关的计算算子，例如Windows计算，在当前的任务中所有的算子将直接使用其所在主机的系统时间。Processing Time是Flink系统中最简单的一种时间概念，基于Processing Time时间概念，Flink的程序性能相对较高，延时也相对较低，对接入到系统中的数据时间相关的计算完全交给算子内部决定，时间窗口计算依赖的时间都是在具体算子运行的过程中产生，不需要做任何时间上的对比和协调。但Processing Time时间概念虽然在性能和易用性的角度上具有优势，但考虑到对数据乱序处理的情况，Processing Time就不是最优的选择。同时在分布式系统中，数据本身不乱序，但每台机器的时间如果不同步，也可能导致数据处理过程中数据乱序的问题，从而影响计算结果。总之，Processing Time概念适用于时间计算精度要求不是特别高的计算场景，例如统计某些延时非常高的日志数据等。

4.时间概念指定

在Flink中默认情况下使用是Process Time时间概念，如果用户选择使用Event Time或者Ingestion Time概念，则需要在创建的StreamExecutionEnvironment中调用setStream-TimeCharacteristic()方法设定系统的时间概念，如下代码使用TimeCharacteristic.EventTime作为系统的时间概念，这样对当前的StreamExecutionEnvironment会全局生效。对应的，如果使用

Ingestion Time概念，则通过传入TimeCharacteristic.IngestionTime参数指定。

```
  val env = StreamExecutionEnvironment.getExecutionEnvironment()
//在系统中指定EventTime概念
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

4.2.2 EventTime和Watermark

通常情况下，由于网络或系统等外部因素影响，事件数据往往不能及时传输至Flink系统中，导致数据乱序到达或者延迟到达等问题，因此，需要有一种机制能够控制数据处理的过程和进度，比如基于事件时间的Window创建后，具体该如何确定属于该Window的数据元素已经全部到达。如果确定全部到达，就可以对Window的所有数据做窗口计算操作（如汇总、分组等），如果数据没有全部到达，则继续等待该窗口中的数据全部到达才开始处理。这种情况下就需要用到水位线（WaterMarks）机制，它能够衡量数据处理进度（表达数据到达的完整性），保证事件数据（全部）到达Flink系统，或者在乱序及延迟到达时，也能够像预期一样计算出正确并且连续的结果。Flink会将用读取进入系统的最新事件时间减去固定的时间间隔作为Watermark，该时间间隔为用户外部配置的支持最大延迟到达的时间长度，也就是说理论上认为不会有事件超过该间隔到达，否则就认为是迟到事件或异常事件。

简单来讲，当事件接入到Flink系统时，会在Sources Operator中根据当前最新事件时间产生Watermarks时间戳，记为X，进入到Flink系统中的数据事件时间，记为Y，如果Y < X，则代表Watermark X时间戳之前的所有事件均已到达，同时Window的End Time大于Watermark，则触发窗口计算结果并输出。从另一个角度讲，如果想触发对Window内的数据元素的计算，就必须保证对所有进入到窗口的数据元素满足其事件时间Y >= X，否则窗口会继续等待Watermark大于窗口结束时间的条件满足。可以看出当有了Watermarks机制后，对基于事件时间的流数据处理会变得特别灵活，可以有效地处理乱序事件的问题，保证数据在流式统计中的结果的正确性。

（1）顺序事件中的Watermarks

如果数据元素的事件时间是有序的，Watermark时间戳会随着数据元素的事件时间按顺序生成，此时水位线的变化和事件时间保持一直，也就是理想状态下的水位线。当Watermark时间大于Windows结束时间就会触发对Windows的数据计算，并创建另一个新的Windows将事件时间Y < X的数据元素分配到新的Window中。如图4-8所示，事件按照其原本的顺序进入系统中，Watermark跟随着事件时间之后生成，可以看出Watermarks其实只是对Stream简单地进行周期性地标记，并没

有特别大的意义，也就是说在顺序事件的数据处理过程中，Watermarks并不能发挥太大的价值，反而会因为设定了超期时间而导致延迟输出计算结果。

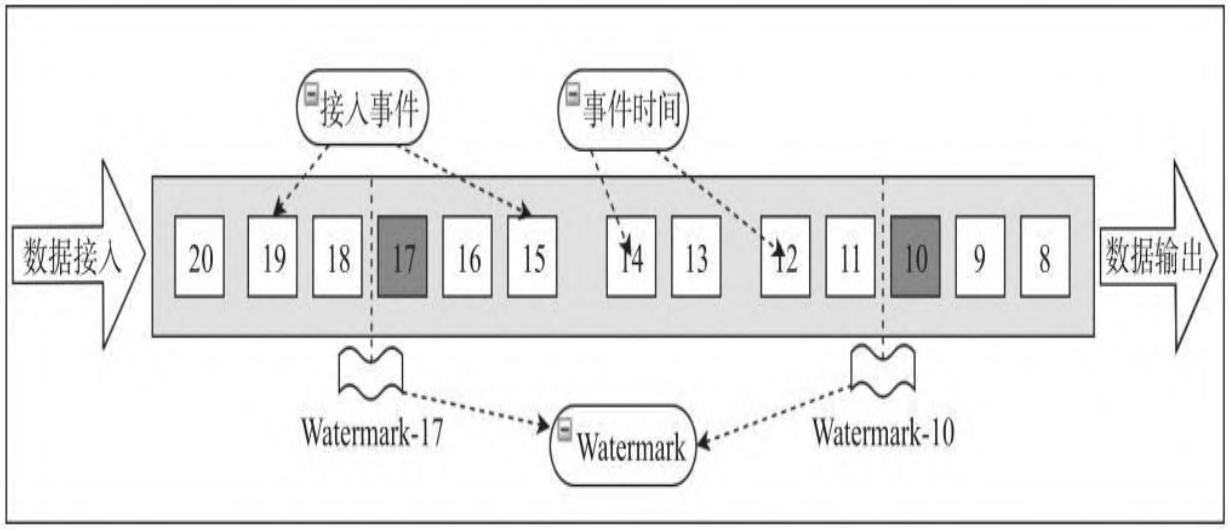

图4-8　顺序时间中的Watermarks

（2）乱序事件中的Watermarks

现实情况下数据元素往往并不是按照其产生顺序接入到Flink系统中进行处理，而频繁出现乱序或迟到的情况，这种情况就需要使用Watermarks来应对。如图4-9所示。事件11和事件17进入到系统中，Flink系统根据设定的延时值分别计算出Watermark W(11)和W(17)，这两个Watermark到达一个Operator中后，便立即调整算子基于事件时间的虚拟时钟与当前的Watermark的值匹配，然后再触发相应的计算以及输出操作。

（3）并行数据流中的Watermarks

Watermark在Source Operator中生成，并且在每个Source Operator的子Task中都会独立生成Watermark。在Source Operator的子任务中生成后就会更新该Task的Watermark，且会逐步更新下游算子中的Watermark水位线，随后一致保持在该并发之中，直到下一次Watermarks的生成，并对前面的Watermarks进行覆盖。如图4-10所示，W(17)水位线已经将Source算子和Map算子的子任务时钟的时间全部更新为值17，并且一直会随着事件向后移动更新下游算子中的事件

时间。如果多个Watermark同时更新一个算子Task的当前事件时间，Flink会选择最小的水位线来更新，当一个Window算子Task中水位线大于了Window结束时间，就会立即触发窗口计算。

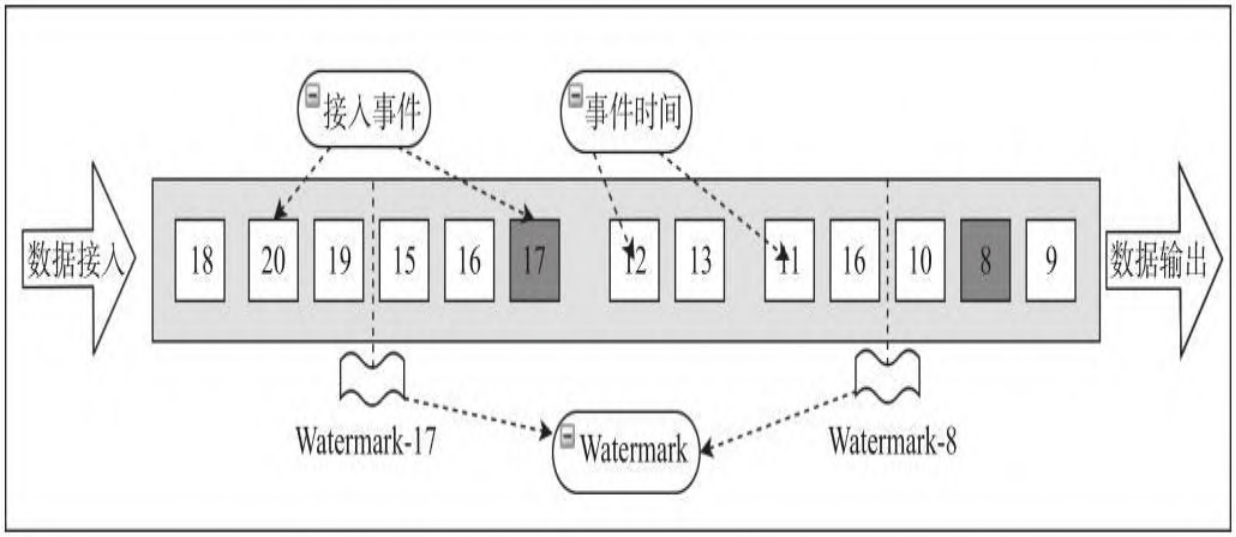

图4-9　乱序事件中的watermark

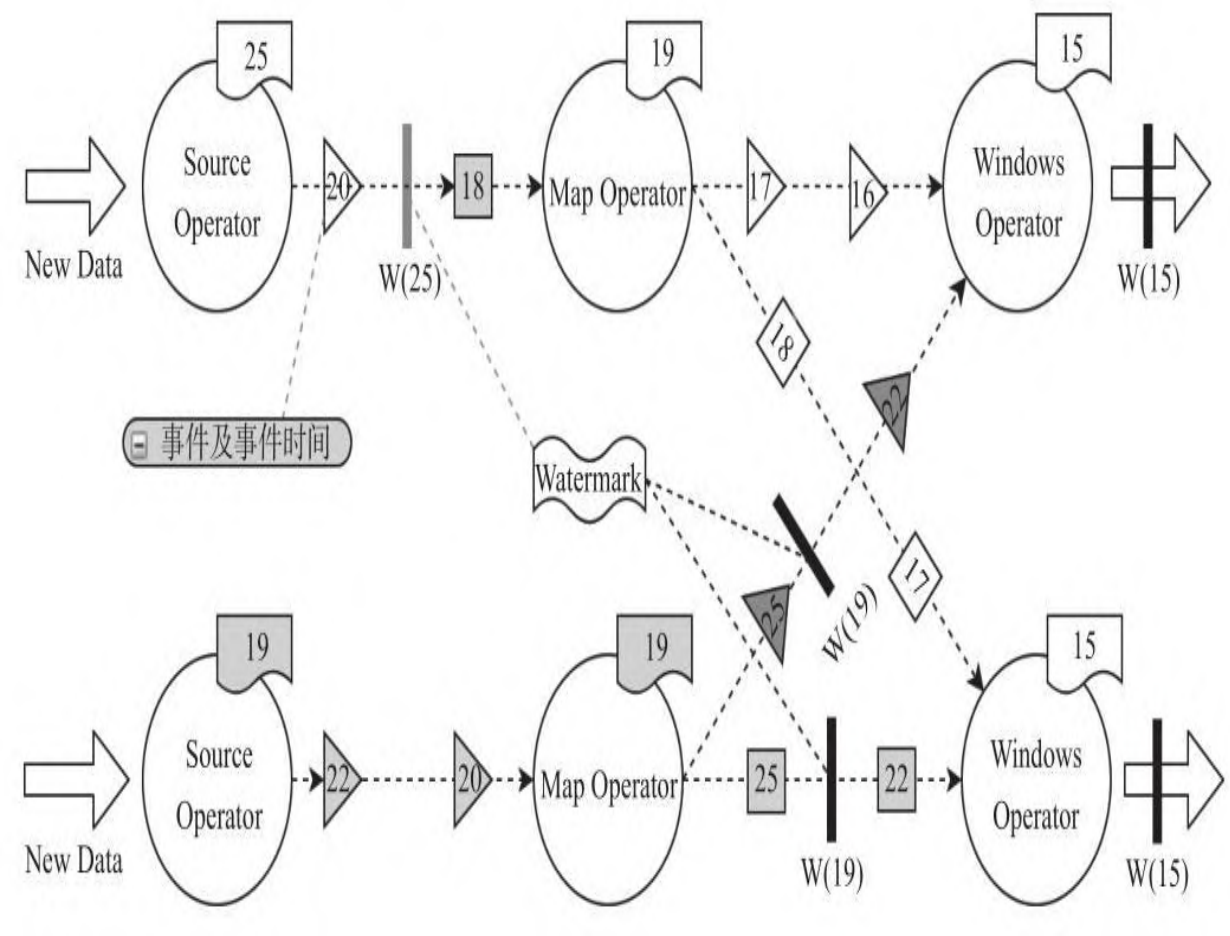

图4-10　Flink并行数据流中的Watermarks

1.指定Timestamps与生成Watermarks

如果使用Event Time时间概念处理流式数据，除了在StreamExecationEviromment中指定TimeCharacteristic外，还需要在Flink程序中指定Event Time时间戳在数据中的字段信息，在Flink程序运行过程中会通过指定字段抽取出对应的事件时间，该过程叫作Timestamps Assigning。简单来讲，就是告诉系统需要用哪个字段作为事件时间的数据来源。另外Timestamps指定完毕后，下面就需要制定创建相应的Watermarks，需要用户定义根据Timestamps计算出Watermarks的生成策略。目前Flink支持两种方式指定Timestamps和生成Watermarks，一种方式在DataStream Source算子接口的Source Function中定义，另外一种方式是通过自定义Timestamp Assigner和Watermark Generator生成。

（1）在Source Function中直接定义Timestamps和Watermarks

在DataStream Source算子中指定EventTime Timestamps，也就是说在数据进入到Flink系统中就直接指定分配EventTime和Watermark。用户需要复写SourceFunciton接口中run()方法实现数据生成逻辑，同时需要调用SourceContext的collectWithTimestamp()方法生成EventTime时间戳，调用emitWatermark()方法生成Watermarks。如代码清单4-6所示，在addSource中通过匿名类实现SourceFunction接口，将本地集合数据读取到系统中，并且分别调用collectWithTimestamp和emitWatermark方法指定EventTime和生成Watermark。

代码清单4-6　通过定义Sourc Function设定Timestamps和Watermarks

```
//创建数组数据集
val input = List(("a", 1L, 1), ("b", 1L, 1), ("b", 3L, 1))
//添加DataSource数据源，实例化SourceFunction接口
val source: DataStream[(String, Long, Int)] = env.addSource(
  new SourceFunction[(String, Long, Int)]() {
    //复写run方法，调用SourceContext接口
    override def run(ctx: SourceContext[(String, Long, Int)]): Unit
= {
      input.foreach(value => {
        //调用collectWithTimestamp增加Event Time抽取
        ctx.collectWithTimestamp(value, value._2)
        //调用emitWatermark,创建Watermark,最大延时设定为1
        ctx.emitWatermark(new Watermark(value._2 - 1))
      })
      //设定默认Watermark
      ctx.emitWatermark(new Watermark(Long.MaxValue))
    }
    override def cancel(): Unit = {}
  })
```

（2）通过Flink自带的Timestamp Assigner指定Timestamp和生成Watermark

如果用户使用了Flink已经定义的外部数据源连接器，就不能再实现SourceFuncton接口来生成流式数据以及相应的Event Time和

Watermark，这种情况下就需要借助Timestamp Assigner来管理数据流中的Timestamp元素和Watermark。Timestamp Assigner一般是跟在Data Source算子后面指定，也可以在后续的算子中指定，只要保证Timestamp Assigner在第一个时间相关的Operator之前即可。如果用户已经在SourceFunciton中定义Timestamps和Watermarks的生成逻辑，同时又使用了Timestamp Assigner，此时Assigner会覆盖Source Function中定义的逻辑。

Flink将Watermarks根据生成形式分为两种类型，分别是Periodic Watermarks和后者。Periodic Watermarks是根据设定时间间隔周期性地生成Watermarks，Punctuated Watermarks是根据接入数据的数量生成，例如数据流中特定数据元素的数量满足条件后触发生成Watermark。在Flink中两种生成Watermarks的逻辑分别借助于AssignerWithPeriodicWatermarks和AssignerWithPunctuatedWatermarks接口定义。

在Flink系统中实现了两种Periodic Watermark Assigner，一种为升序模式，会将数据中的Timestamp根据指定字段提取，并用当前的Timestamp作为最新的Watermark，这种Timestamp Assigner比较适合于事件按顺序生成，没有乱序事件的情况；另外一种是通过设定固定的时间间隔来指定Watermark落后于Timestamp的区间长度，也就是最长容忍迟到多长时间内的数据到达系统。

1）使用Ascending Timestamp Assigner指定Timestamps和Watermarks

如下代码所示，通过调用DataStream API中的assignAscendingTimestamps来指定Timestamp字段，不需要显示地指定Watermark，因为已经在系统中默认使用Timestamp创建Watermark。

```
//指定系统时间概念为EventTime
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
val input = env.fromCollection(List(("a", 1L, 1), ("b", 1L, 1),
("b", 3L, 1)))
//使用系统默认Ascending分配时间信息和Watermark
val withTimestampsAndWatermarks = input.assignAscendingTimestamps(t
=> t._3)
//对数据集进行窗口运算
val result =
```

```
withTimestampsAndWatermarks.keyBy(0).timeWindow(Time.seconds(10)).s
um("_2")
```

2）使用固定时延间隔的Timestamp Assigner指定Timestamps和Watermarks

如下代码所示，通过创建BoundedOutOfOrdernessTimestampExtractor实现类来定义Timestamp Assigner，其中第一个参数Time.seconds(10)代表了最长的时延为10s，第二个参数为extractTimestamp抽取逻辑。在代码中选择使用input数据集中第三个元素作为Event Timestamp，其中Watermarks的创建是根据Timestamp减去固定时间长度生成，如果当前数据中的时间大于Watermarks的时间，则会被认为是迟到事件，具体迟到事件处理策略可以参考后续章节。

```
val withTimestampsAndWatermarks =
input.assignTimestampsAndWatermarks(new
BoundedOutOfOrdernessTimestampExtractor[(String,Long,Int)]
(Time.seconds(10)){
//定义抽取EventTime Timestamp逻辑
  override def extractTimestamp(t: (String, Long, Int)): Long =
t._2
})
```

（3）自定义Timestamp Assigner和Watermark Generator

前面使用Flink系统中已经定义好的两种Timestamp Assigner，用户也可以自定义实现AssignerWithPeriodicWatermarks和AssignerWithPunctuatedWatermarks两个接口来分别生成Periodic Watermarks和Punctuated Watermarks。

1）Periodic Watermarks自定义生成

Periodic Watermarks根据固定的时间间隔，周期性地在Flink系统中分配Timestamps和生成Watermarks，在定义和实现AssignerWithPeriodicWatermarks接口之前，需要先在

ExecutionConfig中调用setAutoWatermarkInterval()方法设定Watermarks产生的时间周期。

```
ExecutionConfig.setAutoWatermarkInterval(...)
```

如代码清单4-7所示，通过创建Class实现AssignerWithPeriodicWatermarks接口，复写extractTimestamp和getCurrentWatermark两个方法，其中extractTimestamp定义了抽取TimeStamps的逻辑，getCurrentWatermark定义了生成Watermark的逻辑。其中getCurrentWatermark生成Watermark依赖于currentMaxTimestamp，getCurrentWatermark()方法每次都会被调用时，如果新产生的Watermark比现在的大，就会覆盖掉现有的Watermark，从而实现对Watermarks数据的更新。

代码清单4-7　通过实现AssignerWithPeriodicWatermarks接口自定义生成Watermark

```
class PeriodicAssigner extends
      AssignerWithPeriodicWatermarks[(String,Long,Int)] {
  val maxOutOfOrderness = 1000L // 1秒时延设定，表示在1秒以内的数据延时有效，超过一秒的数据被认定为迟到事件
  var currentMaxTimestamp: Long = _
  override def extractTimestamp(event: (String,Long,Int), previousEventTimestamp: Long): Long = {
    //复写currentTimestamp方法，获取当前事件时间
    val currentTimestamp = event._2
    //对比当前的事件时间和历史最大事件时间，将最新的时间赋值给currentMaxTimestamp变量
    currentMaxTimestamp = max(currentTimestamp, currentMaxTimestamp)
    currentTimestamp
  }
    //复写getCurrentWatermark方法，生成Watermark
  override def getCurrentWatermark(): Watermark = {
    // 根据最大事件时间减去最大的乱序时延长度，然后得到Watermark
    new Watermark(currentMaxTimestamp - maxOutOfOrderness)
  }
}
```

2）Punctuated Watermarks自定义生成

除了根据时间周期生成Periodic Watermark，用户也可以根据某些特殊条件生成Punctuated Watermarks，例如判断某个数据元素的当前状态，如果接入事件中状态为0则触发生成Watermarks，如果状态不为0，则不触发生成Watermarks的逻辑。生成Punctuated Watermark的逻辑需要通过实现AssignerWithPunctuatedWatermarks接口定义，然后分别复写extractTimestamp方法和checkAndGetNextWatermark方法，完成抽取Event Time和生成Watermark逻辑的定义，具体实现如代码清单4-8所示。

代码清单4-8　通过实现AssignerWithPunctuatedWatermarks接口自定义生成Watermark

```
class PunctuatedAssigner extends
AssignerWithPunctuatedWatermarks[(String,
Long, Int)] {
  //复写extractTimestamp方法，定义抽取Timestamp逻辑
  override def extractTimestamp(element: (String, Long, Int),
previousElementTimestamp: Long): Long = {
    element._2
  }
  //复写checkAndGetNextWatermark方法，定义Watermark生成逻辑
  override def checkAndGetNextWatermark(lastElement: (String, Long,
Int), extractedTimestamp: Long): Watermark = {
    //根据元素中第三位字段状态是否为0生成Watermark
    if (lastElement._3 == 0) new Watermark(extractedTimestamp) else
null
  }
}
```

4.3 Windows窗口计算

Windows计算是流式计算中非常常用的数据计算方式之一，通过按照固定时间或长度将数据流切分成不同的窗口，然后对数据进行相应的聚合运算，从而得到一定时间范围内的统计结果。例如统计最近5分钟内某网站的点击数，此时点击的数据在不断地产生，但是通过5分钟的窗口将数据限定在固定时间范围内，就可以对该范围内的有界数据执行聚合处理，得出最近5分钟的网站点击数。

Flink DataStream API将窗口抽象成独立的Operator，且在Flink DataStream API中已经内建了大多数窗口算子。如下代码展示了如何定义Keyed Windows算子，在每个窗口算子中包含了Windows Assigner、Windows Trigger（窗口触发器）、Evictor（数据剔除器）、Lateness（时延设定）、Output Tag（输出标签）以及Windows Funciton等组成部分，其中Windows Assigner和Windows Funciton是所有窗口算子必须指定的属性，其余的属性都是根据实际情况选择指定。

```
stream.keyBy(...) // 是Keyed类型数据集
.window(...)  //指定窗口分配器类型
[.trigger(...)] //指定触发器类型（可选）
[.evictor(...)]    //指定evictor或者不指定（可选）
[.allowedLateness(...)]   //指定是否延迟处理数据（可选）
[.sideOutputLateData(...)] //指定Output Lag（可选）
.reduce/aggregate/fold/apply()   //指定窗口计算函数
[.getSideOutput(...)]    //根据Tag输出数据（可选）
```

· Windows Assigner：指定窗口的类型，定义如何将数据流分配到一个或多个窗口；

· Windows Trigger：指定窗口触发的时机，定义窗口满足什么样的条件触发计算；

· Evictor：用于数据剔除；

· Lateness：标记是否处理迟到数据，当迟到数据到达窗口中是否触发计算；

· Output Tag：标记输出标签，然后在通过getSideOutput将窗口中的数据根据标签输出；

· Windows Funciton：定义窗口上数据处理的逻辑，例如对数据进行sum操作。

4.3.1 Windows Assigner

1.Keyed和Non-Keyed窗口

在运用窗口计算时，Flink根据上游数据集是否为KeyedStream类型（将数据集按照Key分区），对应的Windows Assigner也会有所不同。上游数据集如果是KeyedStream类型，则调用DataStream API的window()方法指定Windows Assigner，数据会根据Key在不同的Task实例中并行分别计算，最后得出针对每个Key统计的结果。如果是Non-Keyed类型，则调用 WindowsAll()方法来指定Windows Assigner，所有的数据都会在窗口算子中路由到一个Task中计算，并得到全局统计结果。

从业务层面讲，如果用户选择对Key进行分区，就能够将相同key的数据分配在同一个分区，例如统计同一个用户在五分钟内不同的登录IP地址数。如果用户没有根据指定Key，此时需要对窗口上的数据进行去全局统计计算，这种窗口被称为Global Windows，例如统计某一段时间内某网站所有的请求数。

如下代码所示，不同类型的DataStream调用不同的Windows Assigner指定方法。具体的Windows Assigner定义可参考后续章节。

```
val inputStream:DataStream[T] = ...;
//调用KeyBy创建KeyedStream,然后调用window方法指定Windows Assigner
inputStream.keyBy(input=> input.id).window(new MyWindowsAssigner())
//对于DataStream数据集，直接调用windowALL指定Windows Assigner
imputstream.windowAll(new MyAllWindowsAssigner())
```

2.Windows Assigner

Flink支持两种类型的窗口，一种是基于时间的窗口，窗口基于起始时间戳（闭区间）和终止时间戳（开区间）来决定窗口的大小，数据根据时间戳被分配到不同的窗口中完成计算。Flink使用TimeWindow类来获取窗口的起始时间和终止时间，以及该窗口允许进入的最新时间戳信息等元数据。另一种是基于数量的窗口，根据固定的数量定义窗口的大小，例如每5000条数据形成一个窗口，窗口中接入的数据依

赖于数据接入到算子中的顺序，如果数据出现乱序情况，将导致窗口的计算结果不确定。在Flink中可以通过调用DataSteam API中的countWindows()来定义基于数量的窗口。接下来我们重点介绍基于时间窗口的使用，基于数量的窗口读者参考官网资料。

在Flink流式计算中，通过Windows Assigner将接入数据分配到不同的窗口，根据Windows Assigner数据分配方式的不同将Windows分为4大类，分别是滚动窗口(Tumbling Windows)、滑动窗口（Sliding Windows）、会话窗口（Session Windows）和全局窗口（Global Windows）。并且这些Windows Assigner已经在Flink中实现，用户调用DataStream API的windows或windowsAll方法来指定Windows Assigner即可。

（1）滚动窗口

如图4-11所示，滚动窗口是根据固定时间或大小进行切分，且窗口和窗口之间的元素互不重叠。这种类型的窗口的最大特点是比较简单，但可能会导致某些有前后关系的数据计算结果不正确，而对于按照固定大小和周期统计某一指标的这种类型的窗口计算就比较适合，同时实现起来也比较方便。

DataStream API中提供了基于Event Time和Process Time两种时间类型的Tumbling窗口，对应的Assigner分别为TumblingEventTimeWindows和TumblingProcessTimeWindows。调用DataStream API的Window方法来指定相应的Assigner，并使用每种Assigner的of()方法来定义窗口的大小，其中时间单位可以是Time.milliseconds(x)、Time.seconds(x)或Time.minutes(x)，也可以是不同时间单位的组合。如代码清单4-9所示，定义Event Time和Process Time类型的滚动窗口，窗口时间按照10s进行切分，窗口的时间是[1:00:00.000-1:00:09.999]到[1:00:10.000-1:00:19.999]的等固定时间范围。

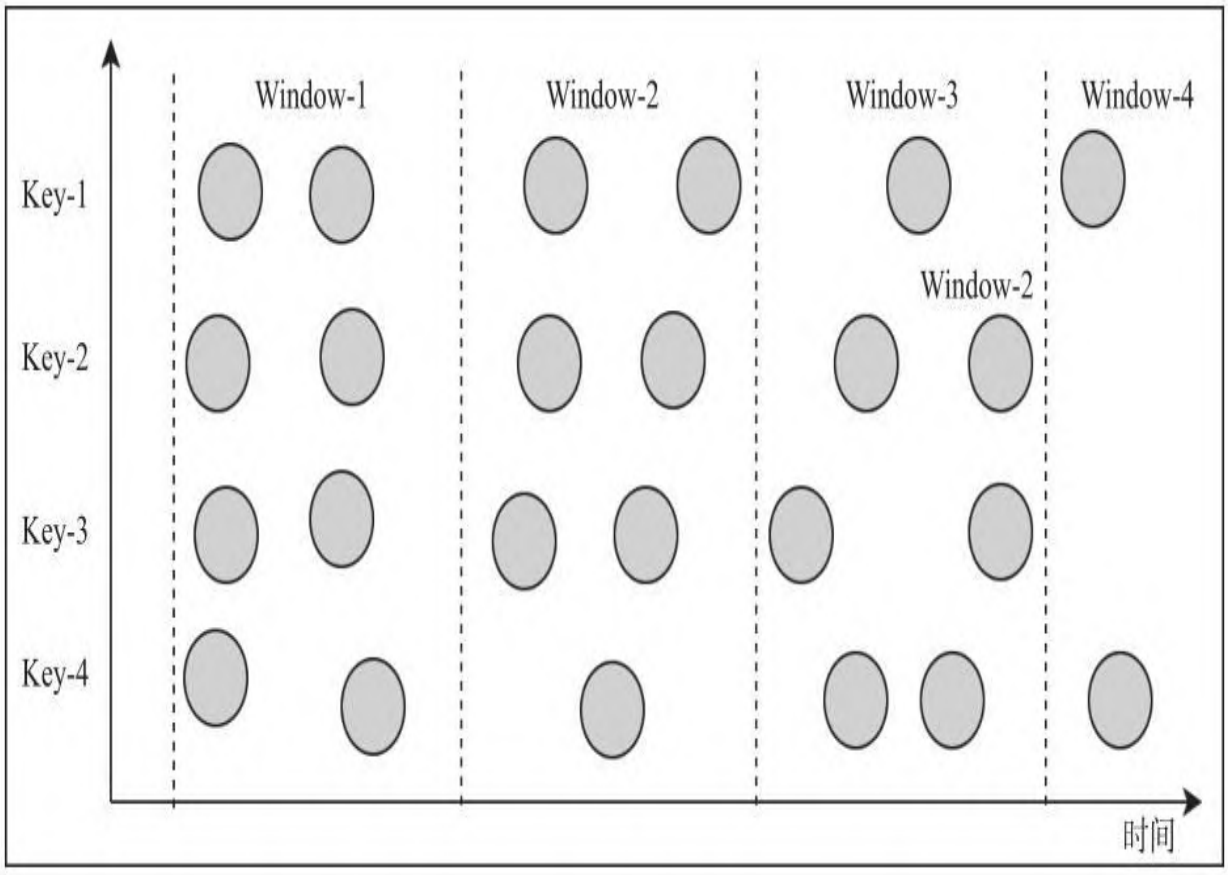

图4-11　数据在Tumbling Windows中的分配过程

代码清单4-9　定义Event Time和Process Time Tumbling Windows

```
val inputStream:DataStream[T] = ...
//定义Event Time Tumbling Windows
val tumblingEventTimeWindows = inputStream
.keyBy(_.id)
//通过使用TumblingEventTimeWindows定义Event Time 滚动窗口
.window(TumblingEventTimeWindows.of(Time.seconds(10)))
.process(...)//定义窗口函数

//定义Process Time Tumbling Windows
val tumblingProcessingTimeWindows = inputStream
  .keyBy(_.id)
//通过使用TumblingProcessTimeWindows定义Event Time 滚动窗口
.window(TumblingProcessTimeWindows.of(Time.seconds(10)))
.process(...)//定义窗口函数
```

上述对滚动窗口定义相对比较常规，用户还可以直接使用DataStream API中timeWindow()快捷方法、定义TumblingEventTimeWindows或TumblingProcessTimeWindows，时间的类型根据用户事先设定的时间概念确定。如下代码使用timeWindow方法来定义滚动窗口：

```
val inputStream:DataStream[T] = ...;
inputStream.keyBy(_.id)
//通过使用timeWindow方式定义滚动窗口，窗口时间类型根据time characteristic确定
.timeWindow(Time.seconds(1))
.process(...)//定义窗口函数
```

 注意

默认窗口时间的时区是UTC-0，因此UTC-0以外的其他地区均需要通过设定时间偏移量调整时区，在国内需要指定Time.hours（-8）的偏移量。

（2）滑动窗口

滑动窗口也是一种比较常见的窗口类型，其特点是在滚动窗口基础之上增加了窗口滑动时间（Slide Time），且允许窗口数据发生重叠。如图4-12所示，当Windows size固定之后，窗口并不像滚动窗口按照Windows Size向前移动，而是根据设定的Slide Time向前滑动。窗口之间的数据重叠大小根据Windows size和Slide time决定，当Slide time小于Windows size便会发生窗口重叠，Slide size大于Windows size就会出现窗口不连续，数据可能不能在任何一个窗口内计算，Slide size和Windows size相等时，Sliding Windows其实就是Tumbling Windows。滑动窗口能够帮助用户根据设定的统计频率计算指定窗口大小的统计指标，例如每隔30s统计最近10min内活跃用户数等。

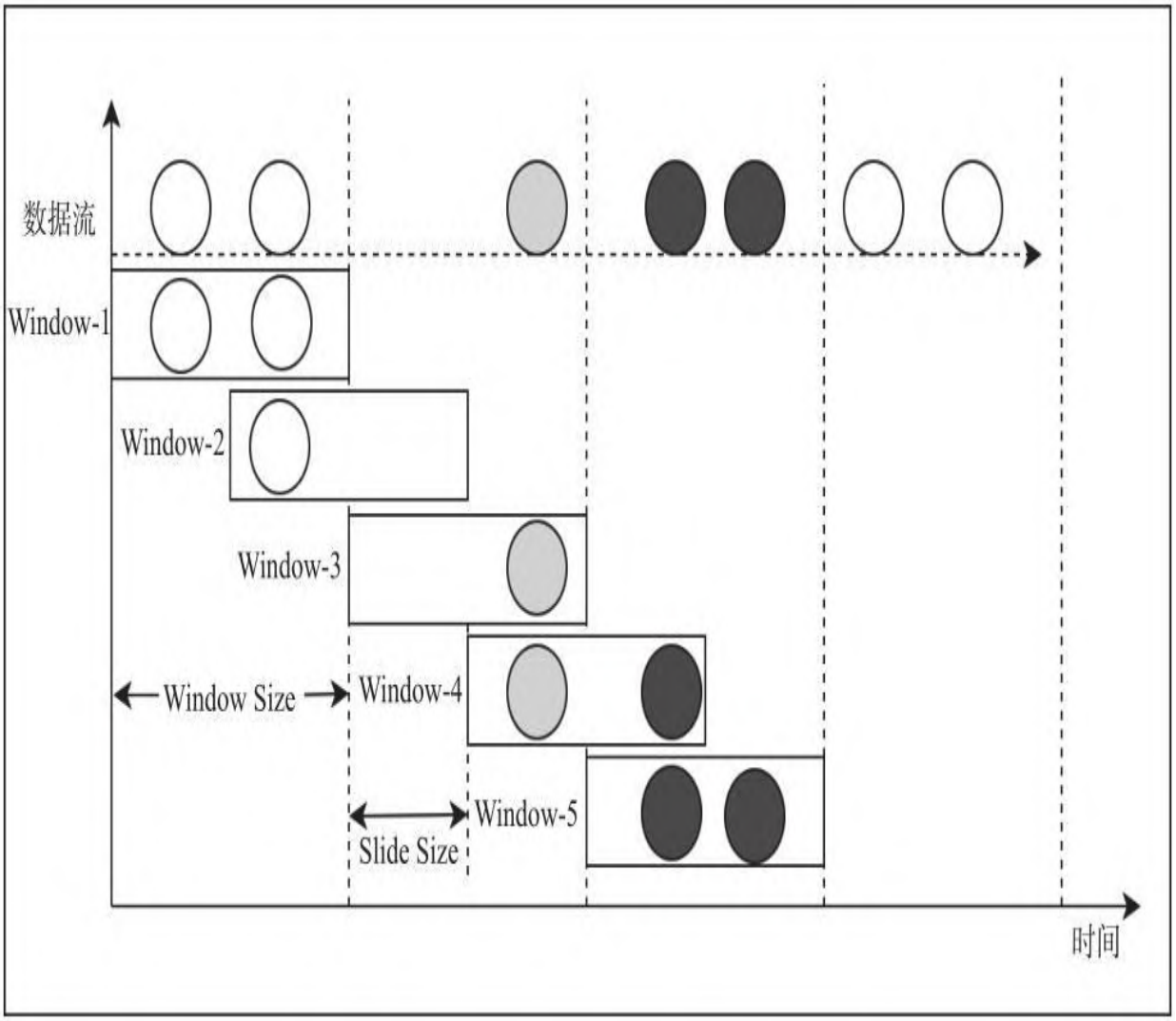

图4-12　数据在Sliding Windows中的分配过程

DataStream API针对Sliding Windows也提供了不同时间类型的Assigner，其中包括基于Event Time的SlidingEventTimeWindows和基于Process Time的SlidingProcessingTime-Windows。代码清单4-10中分别创建了两种时间类型的Sliding Windows，并指定Windows size为1h，Slide time为10s。

代码清单4-10　定义Event Time和Process Time Sliding Windows

```
val inputStream:DataStream[T] = ...
//定义Event Time Sliding Windows
val slidingEventTimeWindows  = inputStream
.keyBy(_.id)
//通过使用SlidingEventTimeWindows定义Event Time 滚动窗口
.window(SlidingEventTimeWindows.of(Time.hours(1),Time.minutes(10)))
.process(...)//定义窗口函数

//定义Process Time Sliding Windows
val slidingProcessingTimeWindows= inputStream
.keyBy(_.id)
//通过使用SlidingProcessingTimeWindows定义Event Time 滚动窗口
.window(SlidingProcessingTimeWindows.of(Time.hours(1),Time.minutes(
10)))
.process(...)//定义窗口计算函数
```

和滚动窗口一样，Flink DataStream API中也提供了创建两种窗口的快捷方式，通过调用DataStream API的timeWindow方法就能够创建对应的窗口。如下代码所示，通过DataStream API timeWindow方法定义滑动窗口，窗口的时间类型根据用户在Execation Enviroment中设定的Time characteristic确定，指定的参数分别Windows Size、Slide Time还有时区偏移量，如果是国内时区则设定为Time.hours(-8)。

```
val inputStream:DataStream[T] = ...;
val slidingEventTimeWindows = inputStream
.keyBy(_.id)
//通过使用timeWindow方式定义滑动窗口，窗口时间类型根据time characteristic
确定
.timeWindow(Time.seconds(10),Time.seconds(1),Time.hours(-8))
.process(...)//定义窗口函数
```

（3）会话窗口

会话窗口（Session Windows）主要是将某段时间内活跃度较高的数据聚合成一个窗口进行计算，窗口的触发的条件是Session Gap，是指在规定的时间内如果没有数据活跃接入，则认为窗口结束，然后触发窗口计算结果。需要注意的是如果数据一直不间断地进入窗口，也会导致窗口始终不触发的情况。与滑动窗口、滚动窗口不同的是，

Session Windows不需要有固定windows size和slide time，只需要定义session gap，来规定不活跃数据的时间上限即可。如图4-13所示，通过session gap来判断数据是否属于同一活跃数据集，从而将数据切分成不同的窗口进行计算。

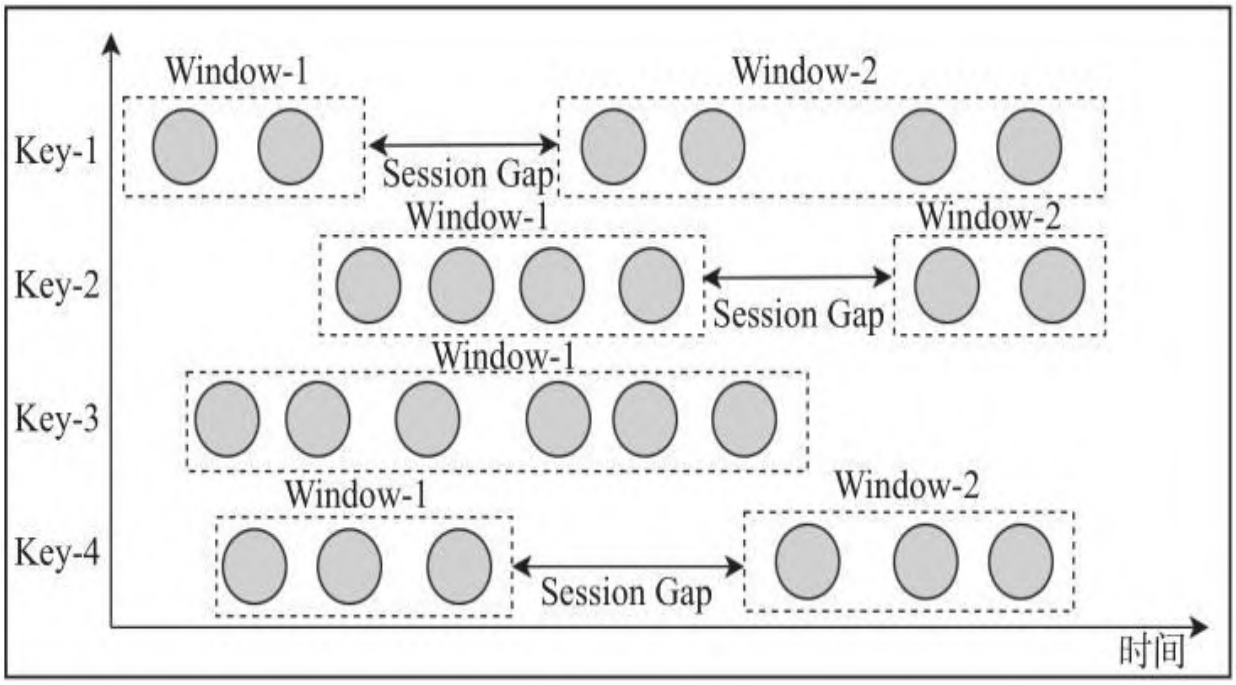

图4-13　数据在Session Windows中的分配过程

Session Windows窗口类型比较适合非连续型数据处理或周期性产生数据的场景，根据用户在线上某段时间内的活跃度对用户行为数据进行统计。和前面两个窗口一样，DataStream API中可以创建基于Event Time和Process Time的Session Windows，对应的Assigner分别为EventTimeSessionWindows和ProcessTimeSessionWindows，如代码清单4-11所示，用户需要调用withGap()方法来指定Session Gap，来规定不活跃数据的时间周期。

代码清单4-11　定义Event Time和Process Time Session Windows

```
val inputStream:DataStream[T] = ...
//定义Event Time Session Windows
val eventTimeSessionWindows  = inputStream
```

```
.keyBy(_.id)
//通过使用EventTimeSessionWindows定义Event Time 滚动窗口
.window(EventTimeSessionWindows.withGap(Time.milliseconds(10)))
.process(...)

//定义Process Time Session Windows
val processingTimeSessionWindows = inputStream
.keyBy(_.id)
//通过使用ProcessingTimeSessionWindows定义Event Time 滚动窗口
.window(ProcessingTimeSessionWindows.withGap(Time.milliseconds(10))
)
.process(...)
```

在创建Session Windows的过程中，除了调用withGap方法输入固定的Session Gap，Flink也能支持动态的调整Session Gap。如代码清单4-12所示，只需要实现SessionWindowTimeGapExtractor接口，并复写extract方法，完成动态Session Gap的抽取，然后将创建好的Session Gap抽取器传入ProcessingTimeSessionWindows.withDynamicGap()方法中即可。

代码清单4-12　通过动态Session Gap定义Event Time和Process Time Session Windows

```
val inputStream:DataStream[T] = ...;
//定义Event Time Session Windows
val eventTimeSessionWindows  = inputStream
.keyBy(_.id)
//通过使用EventTimeSessionWindows定义Event Time 滚动窗口
.window(EventTimeSessionWindows.withDynamicGap(
  //实例化SessionWindowTimeGapExtractor接口
  new SessionWindowTimeGapExtractor[String] {
  override def extract(element: String): Long = {
    // 动态指定并返回Session Gap
}}))
.process(...)

//定义Process Time Session Windows
val processingTimeSessionWindows  = inputStream
.keyBy(_.id)
//通过使用ProcessingTimeSessionWindows定义Event Time 滚动窗口
.window(ProcessingTimeSessionWindows.withDynamicGap(
    //实例化SessionWindowTimeGapExtractor接口
  new SessionWindowTimeGapExtractor[String] {
```

```
  override def extract(element: String): Long = {
    // 动态指定并返回Session Gap
}}))
.process(...)
```

注意

由于Session Windows本质上没有固定的起止时间点，因此底层计算逻辑和Tumbliing窗口及Sliding窗口有一定的区别。Session Windows为每个进入的数据都创建了一个窗口，最后再将距离Session Gap最近的窗口进行合并，然后计算窗口结果。因此对于Session Windows来说需要能够合并的Trigger和Windows Funciton，比如ReduceFunction、AggregateFunction、ProcessWindowFunction等。

（4）全局窗口

全局窗口（Global Windows）将所有相同的key的数据分配到单个窗口中计算结果，窗口没有起始和结束时间，窗口需要借助于Triger来触发计算，如果不对Global Windows指定Triger，窗口是不会触发计算的。因此，使用Global Windows需要非常慎重，用户需要非常明确自己在整个窗口中统计出的结果是什么，并指定对应的触发器，同时还需要有指定相应的数据清理机制，否则数据将一直留在内存中。

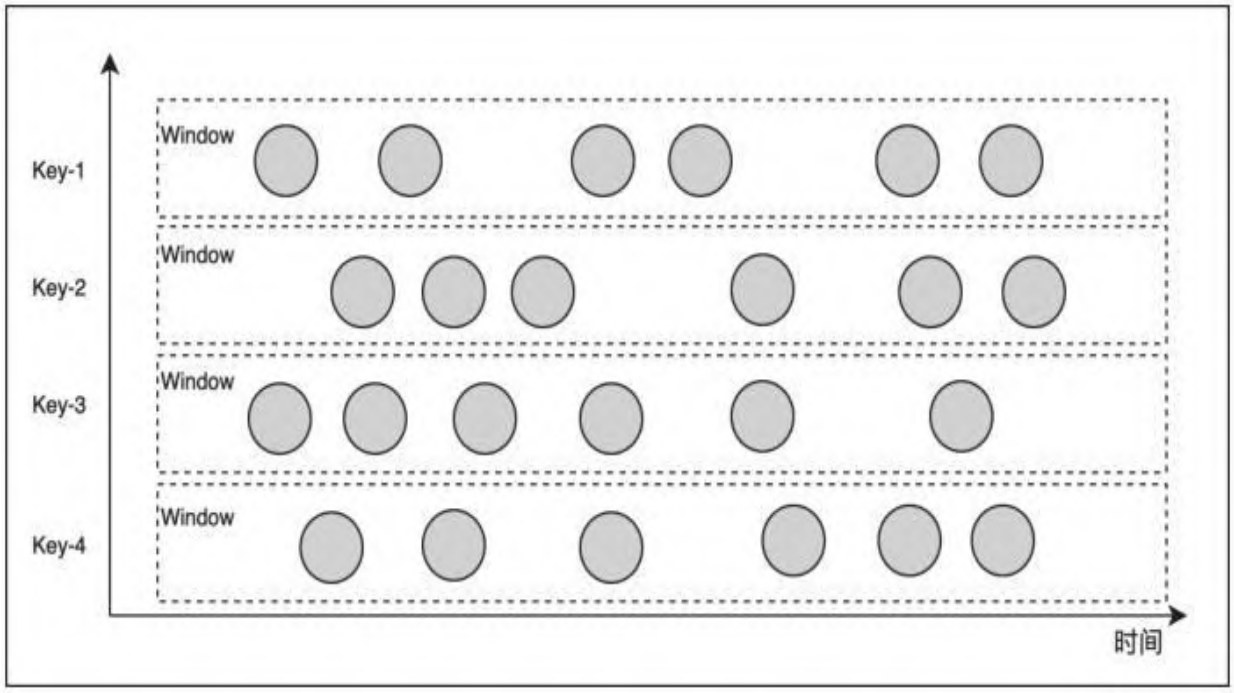

图4-14　数据在Global Windows中的分配过程

如代码清单4-13所示，定义Global Windows相对比较简单，可以通过Global-Windows创建Global Windows的分配器。后面我们会讲到对窗口Trigger的定义，读者可以结合在本节内容学习。

代码清单4-13　定义Global Windows

```
Val inputStream:DataStream[T] = ...;
val globalWindows = inputStream
    .keyBy(_.id)
    .window(GlobalWindows.create())//通过GlobalWindows定义Global
Windows
    .process(...)
```

4.3.2 Windows Function

在上一节的学习我们已经了解Flink支持了不同类型窗口的Assigner，对数据集定义了Window Assigner之后，下一步就可以定义窗口内数据的计算逻辑，也就是Window Function的定义。Flink中提供了四种类型的Window Function，分别为ReduceFunction、AggregateFunction、FoldFunction以及ProcessWindowFunction。

四种类型的Window Fucntion按照计算原理的不同可以分为两大类，一类是增量聚合函数，对应有ReduceFunction、AggregateFunction和FoldFunction；另一类是全量窗口函数，对应有ProcessWindowFunction。增量聚合函数计算性能较高，占用存储空间少，主要因为基于中间状态的计算结果，窗口中只维护中间结果状态值，不需要缓存原始数据。而全量窗口函数使用的代价相对较高，性能比较弱，主要因为此时算子需要对所有属于该窗口的接入数据进行缓存，然后等到窗口触发的时候，对所有的原始数据进行汇总计算。如果接入数据量比较大或窗口时间比较长，就比较有可能导致计算性能的下降。下面将分别对每种Window Function在Flink中的使用进行解释和说明。

1.ReduceFunction

ReduceFunction定义了对输入的两个相同类型的数据元素按照指定的计算方法进行聚合的逻辑，然后输出类型相同的一个结果元素。如代码清单4-14所示，创建好Window Assigner之后通过在reduce()方法中指定ReduceFunciton逻辑，可以使用Scala lambada表达式定义计算逻辑。

代码清单4-14 使用Scala Lambada表达式定义Windows ReduceFunction

```
val inputStream:DataStream[(Int,Long)] = ...;
val reduceWindowStream = inputStream
.keyBy(_._0)
//指定窗口类型
.window(SlidingEventTimeWindows.of(Time.hours(1),Time.minutes(10))
)
```

```
//指定聚合函数逻辑，将根据ID将第二个字段求和
.reduce { (v1, v2) => (v1._1, v1._2 + v2._2) }
```

除了可以直接使用表达式的方式对ReduceFunction逻辑进行定义，也可以创建Class实现ReduceFunction接口来定义聚合逻辑，如代码清单4-15所示。

代码清单4-15　使用Class实现接口的方式表达式定义Windows ReduceFunction

```
val reduceWindowStream = inputStream
  .keyBy(_._1)
  //指定窗口类型
  .window(SlidingEventTimeWindows.of(Time.hours(1),
Time.minutes(10)))
  //定义ReduceFunction实现类定义聚合函数逻辑，将根据ID将第二个字段求和
  .reduce(new ReduceFunction[(Int, Long)] {
  override def reduce(t1: (Int, Long), t2: (Int, Long)): (Int,
Long) = {
    (t1._1, t1._2 + t2._2)
  }
})
```

2. AggregateFunction

和ReduceFunction相似，AggregateFunction也是基于中间状态计算结果的增量计算函数，但AggregateFunction在窗口计算上更加通用。AggregateFunction接口相对ReduceFunction更加灵活，实现复杂度也相对较高。AggregateFunction接口中定义了三个需要复写的方法，其中add()定义数据的添加逻辑，getResult定义了根据accumulator计算结果的逻辑，merge方法定义合并accumulator的逻辑。如代码清单4-16所示，实现了AggregateFunction完成对数据集中字段求取平均值的聚合运算。

代码清单4-16　定义Windows AggregateFunction并在DataStream中使用

```
//定义求取平均值的AggregateFunction
class MyAverageAggregate extends AggregateFunction[(String, Long),
(Long,
  Long), Double] {
    //定义createAccumulator为两个参数的元祖
    override def createAccumulator() = (0L, 0L)
    //定义输入数据累加到accumulator的逻辑
    override def add(input: (String, Long), acc: (Long, Long)) =
      (acc._1 + input._2, acc._2 + 1L)
    //根据累加器得出结果
    override def getResult(acc: (Long, Long)) = acc._1 / acc._2
    //定义累加器合并的逻辑
    override def merge(acc1: (Long, Long), acc2: (Long, Long)) =
    (acc1._1 + acc2._1, acc1._2 + acc2._2)
}
在DataStream API使用定义好的AggregateFunction
val inputStream:DataStream[(String,Long)] = ...
val aggregateWindowStream = inputStream
  .keyBy(_._1)
  //指定窗口类型
  .window(SlidingEventTimeWindows.of(Time.hours(1),
Time.minutes(10)))
  //指定聚合函数逻辑，将根据ID将第二个字段求平均值
  .aggregate(new MyAverageAggregate)
```

3.FoldFunction

FoldFunction定义了如何将窗口中的输入元素与外部的元素合并的逻辑，如代码清单4-17所示将“flink”字符串添加到inputStream数据集中所有元素第二个字段上，并将结果输出到下游DataStream中。

代码清单4-17　Windows FoldFunction定义说明

```
val inputStream:DataStream[(String,Long)] = ...;
val foldWindowStream = inputStream
.keyBy(_._1)
//指定窗口类型
.window(SlidingEventTimeWindows.of(Time.hours(1),
Time.minutes(10)))
//指定聚合函数逻辑，将flink字符串和每个元祖中第二个字段相连并输出
.fold("flink") { (acc, v) => acc + v._2 }
```

FoldFunction已经在Flink DataStream API中被标记为@Deprecated，也就是说很可能会在未来的版本中移除，Flink建议用户使用AggregateFunction来替换使用FoldFunction。

4.ProcessWindowFunction

前面提到的ReduceFunction和AggregateFunction都是基于中间状态实现增量计算的窗口函数，虽然已经满足绝大多数场景，但在某些情况下，统计更复杂的指标可能需要依赖于窗口中所有的数据元素，或需要操作窗口中的状态数据和窗口元数据，这时就需要使用到ProcessWindowsFunction，ProcessWindowsFunction能够更加灵活地支持基于窗口全部数据元素的结果计算，例如统计窗口数据元素中某一字段的中位数和众数。在Flink中ProcessWindowsFunction抽象类定义如代码清单4-18所示，在类中的Context抽象类完整地定义了Window的元数据以及可以操作Window的状态数据，包括GlobalState以及WindowState。

代码清单4-18　Flink中ProcessWindowsFunction抽象类定义

```
  public abstract class ProcessWindowFunction<IN, OUT, KEY, W
extends Window>
  extends AbstractRichFunction {
  //评估窗口并且定义窗口输出的元素
  void process(KEY key, Context ctx, Iterable<IN> vals,
Collector<OUT> out) throws Exception;
  //定义清除每个窗口计算结束后中间状态的逻辑
  public void clear(Context ctx) throws Exception {}
  //定义包含窗口元数据的上下文
public abstract class Context implements Serializable {
    //返回窗口的元数据
    public abstract W window();
    //返回窗口当前的处理时间
    public abstract long currentProcessingTime();
    //返回窗口当前的event-time的Watermark
    public abstract long currentWatermark();
    //返回每个窗口的中间状态
    public abstract KeyedStateStore windowState();
    //返回每个Key对应的中间状态
    public abstract KeyedStateStore globalState();
    //根据OutputTag输出数据
```

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

在实现ProcessWindowFunction接口的过程中，如果不操作状态数据，则只需要实现process()方法即可，该方法中定义了评估窗口和具体数据输出的逻辑。如代码清单4-19所示，通过自定义实现ProcessWindowFunction完成基于窗口上的Key统计包括求和、最小值、最大值，以及平均值的聚合指标，并获取窗口结束时间等元数据信息。

代码清单4-19　自定义ProcessWindowFunction实现指标统计

```
val inputStream:DataStream[(String,Long)] = ...;
//向DataStream数据集指定StaticProcessFunction
val staticStream =
inputStream.keyBy(_._1).timeWindow(Time.seconds(10)).process(new
StaticProcessFunction)
//定义StaticProcessFunction,根据窗口中的数据统计指标
class StaticProcessFunction
    extends ProcessWindowFunction[(String, Long, Int), (String,
Long, Long, Long, Long, Long), String, TimeWindow] {
    override def process(
      key: String,
      ctx: Context,
      vals: Iterable[(String, Long, Int)],
      out: Collector[(String, Long, Long, Long, Long, Long)]):
Unit = {
    //定义求和、最大值、最小值、平均值、窗口时间逻辑
    val sum = vals.map(_._2).sum
    val min = vals.map(_._2).min
    val max = vals.map(_._2).max
    var avg = sum / vals.size
    val windowEnd = ctx.window.getEnd
    //通过out.collect返回计算结果
    out.collect((key, min, max, sum, avg, windowEnd))
  }
}
```

注意

使用ProcessWindowFunction完成简单的聚合运算明显是非常浪费的，用户需要确认自己的业务计算场景，选择合适的WindowFucntion来统计窗口结果。

5. Incremental Aggregation和ProcessWindowsFunction整合

ReduceFunction和AggregateFunction等这些增量聚合函数虽然在一定程度上能够提升窗口计算的性能，但是这些函数的灵活性却不及ProcessWindowsFunction，例如对窗口状态数据的操作以及对窗口中元数据信息的获取等。但是如果使用ProcessWindowsFunction去完成一些基础的增量统计运算相对比较浪费系统资源。此时可以将Incremental Aggregation Function和ProcessWindowsFunction进行整合，以充分利用两种函数各自的优势。在Flink DataStream API也提供了对应的方法，如代码清单4-20所示，将增量ReduceFuction和ProcessWindowFunciton整合，求取窗口中指标最大值以及对应窗口的终止时间。

代码清单4-20　通过ReduceFuction和ProcessWindowFunciton整合求取窗口结束时间和指标最小值

```
ProcessWindowFunciton整合求取窗口结束时间和指标最小值
val inputStream:DataStream[(String,Long)] = ...;
val result = inputStream
.keyBy(_._1)
.timeWindow(Time.seconds(10))
.reduce(
  //定义ReduceFunction,完成求取最小值的逻辑
  (r1: (String, Long, Int), r2: (String, Long, Int)) => {
    if (r1._2 > r2._2) r2 else r1
  }
  , //定义ProcessWindowsFunciton,完成对窗口元数据的采集
  (key: String,
  window: TimeWindow,
  minReadings: Iterable[(String, Long, Int)],
  out: Collector[(Long, (String, Long, Int))]) => {
    val min = minReadings.iterator.next()
    //采集窗口结束时间和最小值对应的数据元素
```

```
    out.collect((window.getEnd, min))
  })
```

从实例中我们可以看出计算过程中需要在reduce方法中定义两个Function，分别是ReduceFunction和ProcessWindowFunction。ReduceFunction中定义了数据元素根据指定Key求取第二个字段对应最小值的逻辑，ProcessWindowFunction定义了从窗口元数据中获取窗口结束时间属性，然后将ReduceFution统计的数据元素的最小值和窗口结束时间共同返回。同理，AggregateFunction和FoldFunction也可以按照这种方式和ProcessWindowsFunction整合，在实现增量聚合计算的同时，也可以操作窗口的元数据信息以及状态数据。

6.ProcessWindowFunction状态操作

除了能够通过RichFunction操作keyed State之外，ProcessWindowFunction也可以操作基于窗口之上的状态数据，这类状态被称为Per-window State。状态数据针对指定的Key在窗口上存储，例如将用户ID作为key，求取每个用户ID最近一个小时的登录数，如果平台中一共有3000个用户，则窗口计算中会创建3000个窗口实例，每个窗口实例中都会保存每个key的状态数据。可以通过ProcessWindowFunction中的Context对象获取并操作Per-window State数据，其中Per-window State在ProcessWindowFunction中分为两种类型：

- globalState：窗口中的keyed state数据不限定在某个窗口中；
- windowState：窗口中的keyed state数据限定在固定窗口中。

获取这些状态数据适合于在同一窗口多次触发计算的场景，或针对迟到的数据来触发窗口计算，例如可以存储每次窗口触发的次数以及最新一次触发的信息，用于下一次窗口触发判断的逻辑。使用Per-window State数据时需要及时清理状态数据，可以调用ProcessWindowFunction的clear()方法完成对状态数据的清理。

4.3.3 Trigger窗口触发器

数据接入窗口后，窗口是否触发WindowFuncition计算，取决于窗口是否满足触发条件，每种类型的窗口都有对应的窗口触发机制，保障每一次接入窗口的数据都能够按照规定的触发逻辑进行统计计算。Flink在内部定义了窗口触发器来控制窗口的触发机制，分别有EventTimeTrigger、ProcessTimeTrigger以及CountTrigger等。每种触发器都对应于不同的Window Assigner，例如Event Time类型的Windows对应的触发器是EventTimeTrigger，其基本原理是判断当前的Watermark是否超过窗口的EndTime，如果超过则触发对窗口内数据的计算，反之不触发计算。以下对Flink自带的窗口触发器进行分类整理，用户可以根据需要选择合适的触发器：

· EventTimeTrigger：通过对比Watermark和窗口EndTime确定是否触发窗口，如果Watermark的时间大于Windows EndTime则触发计算，否则窗口继续等待；

· ProcessTimeTrigger：通过对比ProcessTime和窗口EndTime确定是否触发窗口，如果窗口Process Time大于Windows EndTime则触发计算，否则窗口继续等待；

· ContinuousEventTimeTrigger：根据间隔时间周期性触发窗口或者Window的结束时间小于当前EventTime触发窗口计算；

· ContinuousProcessingTimeTrigger：根据间隔时间周期性触发窗口或者Window的结束时间小于当前ProcessTime触发窗口计算；

· CountTrigger：根据接入数据量是否超过设定的阈值确定是否触发窗口计算；

· DeltaTrigger：根据接入数据计算出来的Delta指标是否超过指定的Threshold，判断是否触发窗口计算；

· PurgingTrigger：可以将任意触发器作为参数转换为Purge类型触发器，计算完成后数据将被清理。

如果已有Trigger无法满足实际需求，用户也可以继承并实现抽象类Trigger自定义触发器，FlinkTrigger接口中共有如下方法需要复写，然后在DataStream API中调用trigger方法传入自定义Trigger。

· OnElement()：针对每一个接入窗口的数据元素进行触发操作；

· OnEventTime()：根据接入窗口的EventTime进行触发操作；

· OnProcessTime()：根据接入窗口的ProcessTime进行触发操作；

· OnMerge()：对多个窗口进行Merge操作，同时进行状态的合并；

· Clear()：执行窗口及状态数据的清除方法。

在自定义触发器时，判断窗口触发方法返回的结果有如下类型，分别是CONTINUE、FIRE、PURGE、FIRE_AND_PURGE。其中CONTINUE代表当前不触发计算，继续等待；FIRE代表触发计算，但是数据继续保留；PURGE代表窗口内部数据清除，但不触发计算；FIRE_AND_PURGE代表触发计算，并清除对应的数据；用户在指定触发逻辑满足时可以通过将以上状态返回给Flink，由Flink在窗口计算过程中，根据返回的状态选择是否触发对当前窗口的数据进行计算。

如代码清单4-21中所示，通过自定义EarlyTriggeringTrigger实现窗口触发的时间间隔，当窗口是Session Windows时，如果用户长时间不停地操作，导致Session Gap一直都不生成，因此该用户的数据会长期存储在窗口中，如果需要至少每隔五分钟统计一下窗口的结果，不想一直等待，此时就需要自定义Trigger来实现。Flink中已经定义了ContinuousEventTimeTrigger窗口触发器来实现相似的功能，以下通过自定义实现类似ContinuousEventTimeTrigger的窗口触发器。

代码清单4-21　自定义实现EarlyTriggeringTrigger

```
class ContinuousEventTimeTrigger(interval: Long) extends
Trigger[Object,
TimeWindow] {
  //重定义Java.lang.Long类型为JLong类型
```

```
  private type JLong = java.lang.Long
  //实现函数，求取2个时间戳的最小值
  private val min = new ReduceFunction[JLong] {
    override def reduce(v1: JLong, v2: JLong): JLong =
Math.min(v1, v2)
  }
  private val stateDesc = new ReducingStateDescriptor[JLong]
("trigger-time", min, Types.LONG)
  //处理接入的元素，每次都会被调用
  override def onElement(element: Object,
                   timestamp: Long,
                   window: TimeWindow,
                   ctx: TriggerContext): TriggerResult =
  //如果当前的Watermark超过窗口的结束时间，则清除定时器内容，直接触发窗口计算
    if (window.maxTimestamp <= ctx.getCurrentWatermark) {
      clearTimerForState(ctx)
      TriggerResult.FIRE
    }
    else {
      //否则将窗口的结束时间注册给EventTime定时器
      ctx.registerEventTimeTimer(window.maxTimestamp)
      //获取当前分区状态中的时间戳
      val fireTimestamp = ctx.getPartitionedState(stateDesc)
      //如果第一次执行，则对元素的timestamp进行floor操作，取整后加上传入的
实例变量interval，得到下一次触发时间并注册，添加到状态中
      if (fireTimestamp.get == null) {
        val start = timestamp - (timestamp % interval)
        val nextFireTimestamp = start + interval
        ctx.registerEventTimeTimer(nextFireTimestamp)
        fireTimestamp.add(nextFireTimestamp)
      }
      //此时继续等待
      TriggerResult.CONTINUE
    }
  //时间概念类型不选择ProcessTime,不会基于rocessing Time 触发，直接返回
CONTINUE
  override def onProcessingTime (time: Long,window: TimeWindow,
ctx:    TriggerContext): TriggerResult = TriggerResult.CONTINUE
  //当Watermark超过注册时间时，就会执行onEventTime方法
  override def onEventTime(time: Long,
                       window: TimeWindow,
                       ctx: TriggerContext): TriggerResult = {
    //如果事件时间等于maxTimestamp时间，则清空状态数据，并触发计算
    if (time == window.maxTimestamp()) {
      clearTimerForState(ctx)
      TriggerResult.FIRE
    } else {
      //否则，获取状态中的值（maxTimestamp和nextFireTimestamp的最小值）
      val fireTimestamp = ctx.getPartitionedState(stateDesc)
```

```
      //如果状态中的值等于事件时间，则清除定时器时间戳，注册下一个interval的
时间戳，并触发窗口计算
      if (fireTimestamp.get == time) {
        fireTimestamp.clear()
        fireTimestamp.add(time + interval)
        ctx.registerEventTimeTimer(time + interval)
        TriggerResult.FIRE
      } else {//否则继续等待
        TriggerResult.CONTINUE}}
  //从TriggerContext中获取状态中的值，并从定时器中清除
  private def clearTimerForState(ctx: TriggerContext): Unit = {
    val timestamp = ctx.getPartitionedState(stateDesc).get()
    if (timestamp != null) {
      ctx.deleteEventTimeTimer(timestamp)}}
  //用于session window的merge,指定可以merge
  override def canMerge: Boolean = true
  //定义窗口状态merge的逻辑
  override def onMerge(window: TimeWindow,
                       ctx: OnMergeContext): TriggerResult = {
    ctx.mergePartitionedState(stateDesc)
    val nextFireTimestamp =
ctx.getPartitionedState(stateDesc).get()
    if (nextFireTimestamp != null) {
      ctx.registerEventTimeTimer(nextFireTimestamp)
    }
    TriggerResult.CONTINUE
  }
  //删除定时器中已经触发的时间戳，并调用Trigger的clear方法
  override def clear(window: TimeWindow,
                     ctx: TriggerContext): Unit = {
    ctx.deleteEventTimeTimer(window.maxTimestamp())
    val fireTimestamp = ctx.getPartitionedState(stateDesc)
    val timestamp = fireTimestamp.get
    if (timestamp != null) {
      ctx.deleteEventTimeTimer(timestamp)
      fireTimestamp.clear()}}
  override def toString: String = s"
ContinuousEventTimeTrigger($interval)"
}
//类中的of方法，传入interval，作为参数传入此类的构造器，时间转换为毫秒
object ContinuousEventTimeTrigger {
  def of(interval: Time) = new
ContinuousEventTimeTrigger(interval.toMilliseconds)
}
```

如以下代码所示，通过调用DataStream API中的trigger()方法，将自定义ContinuousEvent-TimeTrigger应用到窗口上，就能够按照自

定义的窗口触发策略完成对窗口数据的计算。

```
val windowStream = inputStream
  .keyBy(_._1)
  //指定窗口类型
  .window(EventTimeSessionWindows.withGap(Time.milliseconds(10)))
  //指定窗口触发器
  .trigger(ContinuousEventTimeTrigger.of(Time.seconds(5)))
  .process(...)
})
```

从代码中可以看出，重写Trigger接口中的方法非常多，且每个方法中的逻辑也比较复杂，因此建议用户尽可能使用Flink中自带的窗口触发器。另外需要注意的是，当用户使用自定义触发器时，默认触发器将会被覆盖，因此用户在自己定义触发器的时候，需要考虑对默认触发器中的功能是否有依赖，例如使用EventTime Windows时，就需要考虑是否需要借助EventTimeTrigger中Watermark处理乱序数据的处理，并在当前自定义的触发器中实现EventTimeTrigger的对应逻辑。另外Flink中GlobalWindow的默认触发器是NeverTrigger，如果使用GlobalWindow窗口，则必须自定义触发器，否则数据接入Window后将永远不会触发计算，窗口中的数据量会越来越大，最终导致系统内存溢出等问题。

4.3.4 Evictors数据剔除器

Evictors是Flink窗口机制中一个可选的组件，其主要作用是对进入WindowFuction前后的数据进行剔除处理，Flink内部实现CountEvictor、DeltaEvictor、TimeEvitor三种Evictors。在Flink中Evictors通过调用DataStream API中evictor()方法使用，且默认的Evictors都是在WindowsFunction计算之前对数据进行剔除处理。

· CountEvictor：保持在窗口中具有固定数量的记录，将超过指定大小的数据在窗口计算前剔除；

· DeltaEvictor：通过定义DeltaFunction和指定threshold，并计算Windows中的元素与最新元素之间的Delta大小，如果超过threshold则将当前数据元素剔除；

· TimeEvictor：通过指定时间间隔，将当前窗口中最新元素的时间减去Interval，然后将小于该结果的数据全部剔除，其本质是将具有最新时间的数据选择出来，删除过时的数据。

和Trigger一样，用户也可以通过实现Evictor接口完成自定义Evictor，如代码清单4-22所示，Evictor接口需要复写的方法有两个：evictBefore()方法定义数据在进入WindowsFunction计算之前执行剔除操作的逻辑，evictAfter()方法定义数据在WindowsFunction计算之后执行剔除操作的逻辑。其中方法参数中elements是代表在当前窗口中所有的数据元素。

代码清单4-22 Evictor接口在Flink中的定义

```
public interface Evictor<T, W extends Window> extends Serializable
{
//定义WindowFunciton触发之前的数据剔除逻辑
void evictBefore(Iterable<TimestampedValue<T>> elements, int size,
W window, EvictorContext evictorContext);
//定义WindowFunciton触发之后的数据剔除逻辑
void evictAfter(Iterable<TimestampedValue<T>> elements, int size,
W window, EvictorContext evictorContext);
```

```
//定义上下文对象
interface EvictorContext {
  long getCurrentProcessingTime();
  MetricGroup getMetricGroup();
  long getCurrentWatermark();
}}
```

注意

从Evictor接口中能够看出，元素在窗口中其实并没有保持顺序，因此如果对数据在进入WindowFunciton之前或之后进行预处理，其实和数据在进入窗口中的顺序是没有关系的，无法控制绝对的先后关系。

4.3.5　延迟数据处理

基于Event-Time的窗口处理流式数据，虽然提供了Watermark机制，却只能在一定程度上解决了数据乱序的问题。但在某些情况下数据可能延时会非常严重，即使通过Watermark机制也无法等到数据全部进入窗口再进行处理。Flink中默认会将这些迟到的数据做丢弃处理，但是有些时候用户希望即使数据延迟到达的情况下，也能够正常按照流程处理并输出结果，此时就需要使用Allowed Lateness机制来对迟到的数据进行额外的处理。

DataStream API中提供了allowedLateness方法来指定是否对迟到数据进行处理，在该方法中传入Time类型的时间间隔大小(t)，其代表允许延时的最大时间，Flink窗口计算过程中会将Window的Endtime加上该时间，作为窗口最后被释放的结束时间（P），当接入的数据中Event Time未超过该时间（P），但Watermak已经超过Window的EndTime时直接触发窗口计算。相反，如果事件时间超过了最大延时时间（P），则只能对数据进行丢弃处理。

需要注意的是，默认情况下GlobleWindow的最大Lateness时间为Long.MAX_VALUE，也就是说不超时。因此数据会源源不断地累积到窗口中，等待被触发。其他窗口类型默认的最大Lateness时间为0，即不允许有延时数据的情况。

通常情况下用户虽然希望对迟到的数据进行窗口计算，但并不想将结果混入正常的计算流程中，例如用户大屏数据展示系统，即使正常的窗口中没有将迟到的数据进行统计，但为了保证页面数据显示的连续性，后来接入到系统中迟到数据所统计出来的结果不希望显示在屏幕上，而是将延时数据和结果存储到数据库中，便于后期对延时数据进行分析。对于这种情况需要借助Side Output来处理，通过使用sideOutputLateData（OutputTag）来标记迟到数据计算的结果，然后使用getSideOutput（lateOutputTag）从窗口结果中获取lateOutputTag标签对应的数据，之后转成独立的DataStream数据集进行处理，如下代码所示，创建late-data的OutputTag，再通过该标签从窗口结果中将迟到数据筛选出来。

```
//创建延迟数据OutputTag,标记为late-data
val lateOutputTag = OutputTag[T]("late-data")
val input: DataStream[T] = ...
val result = input
  .keyBy(<key selector>)
  .window(<window assigner>)
  .allowedLateness(<time>)
//通过sideOutputLateData方法对迟到数据进行标记
  .sideOutputLateData(lateOutputTag)
  .process(...)
//最后通过lateOutputTag从窗口结果中获取迟到数据产生的统计结果
val lateStream = result.getSideOutput(lateOutputTag)
```

4.3.6　连续窗口计算

对接入的流式数据进行窗口处理的过程，其实是将DataStream在窗口中完成窗口计算逻辑，处理完毕后数据又会被转换为DataStream，对于Windows处理出来的结果，可以继续按照DataStream的方式进行后续处理。

1. 独立窗口计算

如代码清单4-23所示，针对同一个DataStream进行不同的窗口处理，窗口之间相对独立，输出结果在不同的DataStream中，这时在Flink Runtime执行环境中，将分为两个Window Operator在不同的Task中执行，相互之间元数据不会进行共享。

代码清单4-23　独立窗口计算模式

```
val input: DataStream[T] = ...
//定义Session Gap为100s的窗口并计算结果windowStream1
val windowStream1 = inputStream
  .keyBy(_._1)
  //指定窗口类型
  .window(EventTimeSessionWindows.withGap(Time.milliseconds(100)))
  .process(...)
})
//定义Session Gap为100s的窗口并计算结果windowStream2
val windowStream2 = inputStream
  .keyBy(_._1)
  //指定窗口类型
  .window(EventTimeSessionWindows.withGap(Time.milliseconds(10)))
  .process(...)
})
```

2. 连续窗口计算

连续窗口计算表示上游窗口的计算结果是下游窗口计算的输入，窗口算子和算子之间上下游关联，窗口之间的元数据信息能够共享。如代码清单4-24所示，在上游窗口统计最近10种每个Key的最小值，通过下游窗口统计出整个窗口上TopK的值。可以看出，两个窗口的类型

和End-Time都是一致的，上游将窗口元数据（Watermark）信息传递到下游窗口中，真正触发计算是在下游窗口中，窗口的计算结果全部在下游窗口中统计得出，最终完成在同一个窗口中同时计算与Key相关和非key相关的指标。

代码清单4-24　在同一个窗口上计算按Key统计Sum值以及统计TopK结果

```
val input: DataStream[T] = ...
//定义Session Gap为100s的窗口并计算结果windowStream1
val windowStream1 = inputStream
  .keyBy(_._1)
  //指定窗口类型
  .window(TumblingEventTimeWindows.of(Time.milliseconds(10)))
  .reduce(new Min())
})
//在windowStream1上定义Session Gap为10s的窗口并计算结果windowStream2
val windowStream2 = windowStream1
  //指定窗口类型
  .windowAll(TumblingEventTimeWindows.of(Time.milliseconds(10)))
  .process(new TopKWindowFunction())
})
```

4.3.7 Windows多流合并

在Flink中支持窗口上的多流合并，即在一个窗口中按照相同条件对两个输入数据流进行关联操作，需要保证输入的Stream要构建在相同的Window上，并使用相同类型的Key作为关联条件。如下代码所示，先通过join方法将inputStream1数据集和inputStream2数据集关联，形成JoinedStreams类型数据集，调用where()方法指定inputStream1数据集的Key，调用equalTo()方法指定inputStream2对应关联的Key，通过window()方法指定Window Assigner，最后再通过apply()方法中传入用户自定义的JoinFunction或者FlatJoinFunction对输入的数据元素进行窗口计算。

```
inputStream1:DataStream[(Long,String,Int)] = ...
inputStream2:DataStream[(String,Long,Int)] = ...
//通过DataStream Join方法将两个数据流关联
inputStream1.join(inputStream2)
    //指定inputStream1的关联Key
    .where(_._1)
    //指定inputStream2的关联Key
    .equalTo(_._2)/
    //指定Window Assigner
    .window(TumblingEventTimeWindows.of(Time.milliseconds(10)))
  .apply(<JoinFunction>) //指定窗口计算函数
```

在Windows Join过程中所有的Join操作都是Inner-join类型，也就是说必须满足在相同窗口中，每个Stream中都要有Key且Key值相同才能完成关联操作并输出结果，任何一个数据集中的Key不具备或缺失都不会输出关联计算结果。

在Windows Join中，指定不同的Windows Assigner，DataStream的关联过程也相应不同，包括数据计算的方式也会有所不同。在Flink中共有滚动窗口关联（Tumbling Window join）、滑动窗口关联（Sliding Window join）、会话窗口关联（Session Window join）以及间隔关联（Interval Join）四种多流合并的操作可以选择作用。

1. 滚动窗口关联

滚动窗口关联操作是将滚动窗口中相同Key的两个DataStream数据集中的元素进行关联，并应用用户自定义的JoinFunction计算关联结果。如图4-15所示两个DataStream Join的过程中，数据是在相同的窗口中进行关联，如果出现相同的Key值，则应用JoinFunction计算出结果并输出。在窗口中任何一个DataStream没有对应的Key的元素，窗口都不会输出计算结果。

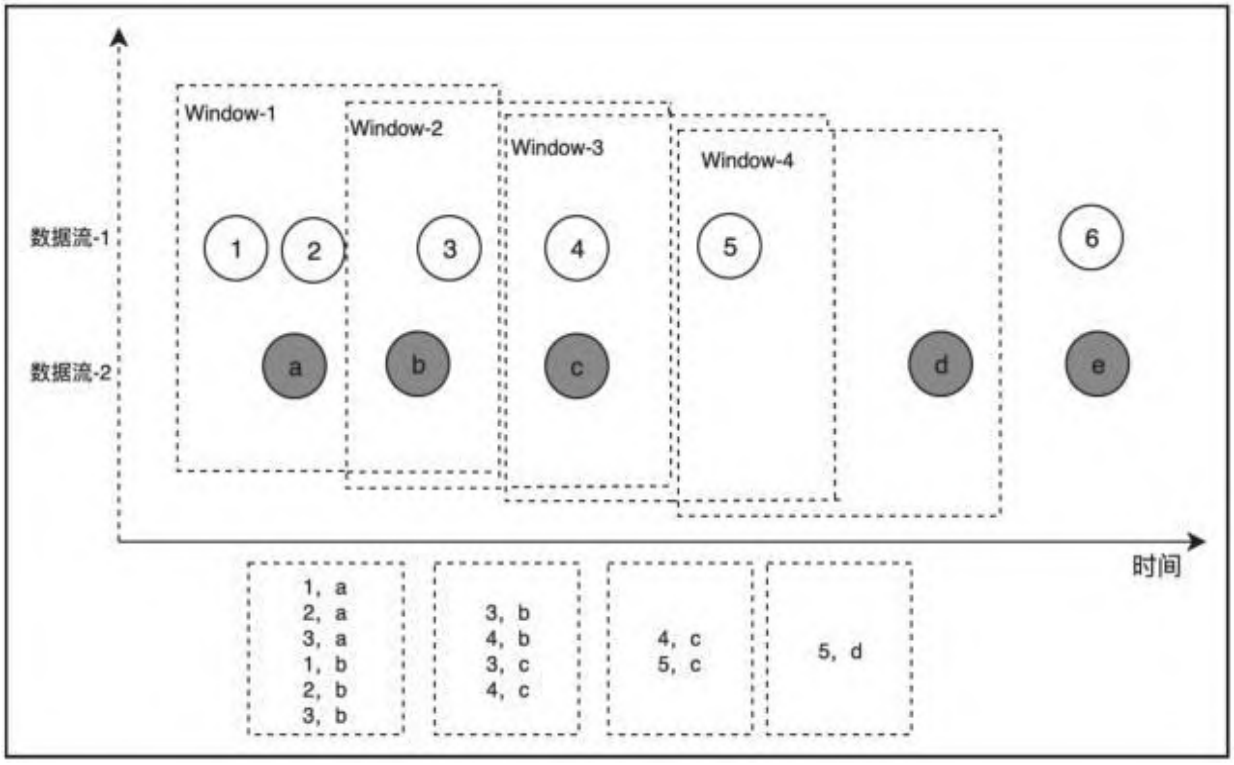

图4-15　滚动窗口关联数据计算过程

图4-15中黑色代表的是一个DataStream数据集中的数据，白色代表另外一个DataStream数据集中的数据，两个Stream在相同的时间窗口中进行内连操作，最后一个窗口因为黑色数据集中没有数据，所以关联操作没有结果输出。图中的滚动窗口关联逻辑可以通过代码清单4-25实现。

代码清单4-25　滚动窗口关联完成对黑白数据集的关联

```
//创建黑色元素数据集
val blackStream: DataStream[(Int, Long)] = ...
```

```
//创建白色元素数据集
val whiteStream: DataStream[(Int, Long)] = ...
//通过Join方法将两个数据集进行关联
val windowStream: DataStream[(Int,Long)] =
blackStream.join(whiteStream)
  .where(_._1)       //指定第一个Stream的关联Key
  .equalTo(_._1)    //指定第二个Stream的关联Key
  .window(TumblingEventTimeWindows.of(Time.milliseconds(10))) //指定
窗口类型
  .apply((black, white) => (black._1,black._2 + white._2))  //应用
JoinFunciton
```

---

2. 滑动窗口关联

滑动窗口是指窗口在指定的SlideTime的间隔内进行滑动，同时允许窗口重叠，在滑动窗口关联操作过程中，就会出现重叠的关联操作，如图4-16所示，两个DataStream数据元素在单个窗口中根据相同的Key进行关联，且关联数据会发生重叠同时滑动窗口关联也是基于内连接，如果一个窗口中只出现了一个DataStream中的Key，则不会输出关联计算结果。

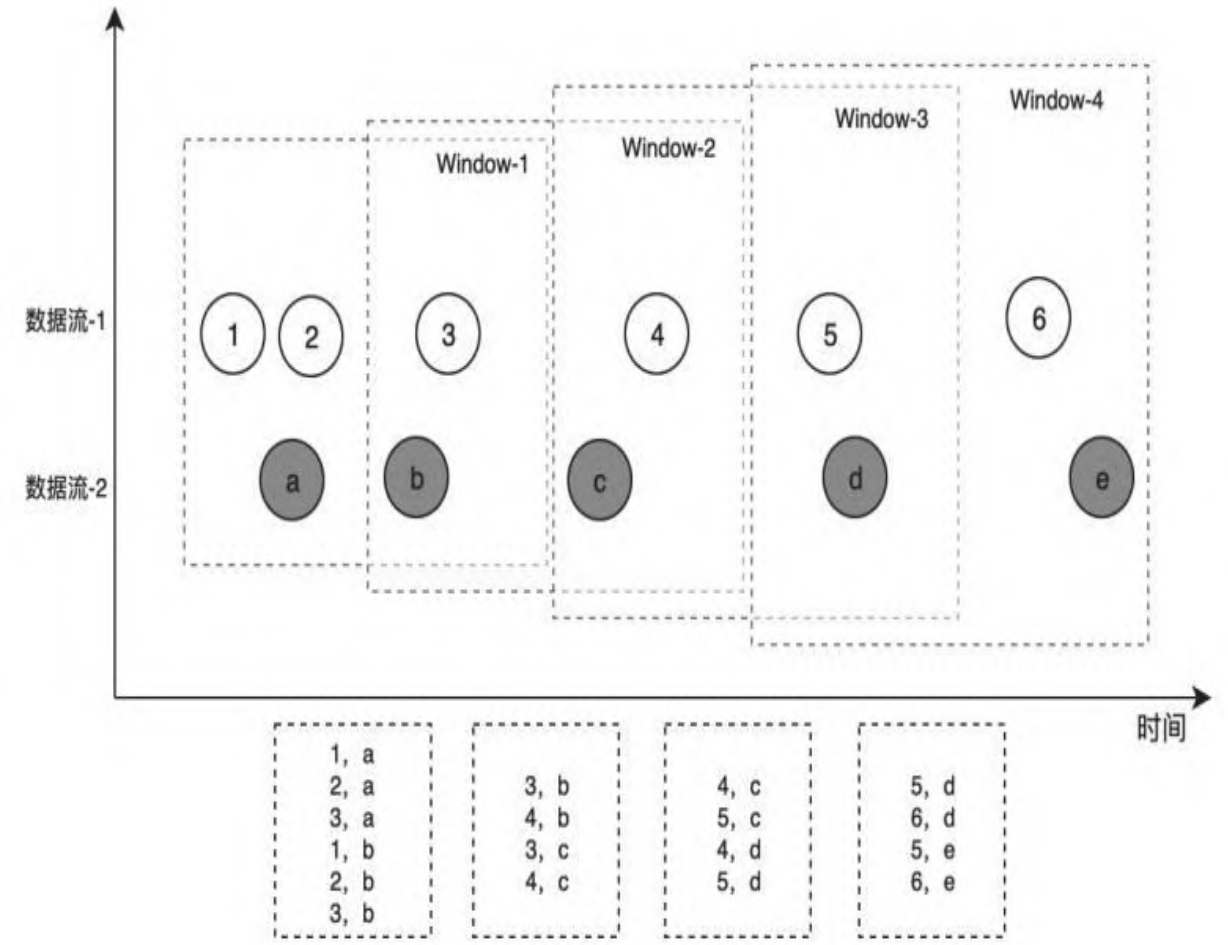

图4-16　滑动窗口关联数据计算过程

随着时间的推后，会发现在第一个窗口中进行Join的0和0元素，同时在第二个窗口中进行关联得出来的结果为（0, 0）和（1, 0），各个窗口关联出来的结果中有可能会出现重复情况，这也是滑动窗口本身的特性允许窗口重叠造成的。图中对应窗口计算的实现如下，和滑动窗口关联相比，窗口的类型变为SlidingEventTimeWindows，产生基于滑动窗口的计算结果。

```
//创建黑色元素数据集
val blackStream: DataStream[(Int, Long)] = ...
//创建白色元素数据集
val whiteStream: DataStream[(Int, Long)] = ...
//通过Join方法将两个数据集进行关联
val windowStream: DataStream[(Int,Long)] =
```

```
blackStream.join(whiteStream)
  .where(_._1) //指定第一个Stream的关联Key
  .equalTo(_._1) //指定第二个Stream的关联Key

.window(SlidingEventTimeWindows.of(Time.milliseconds(10),Time.milli
seconds(2))) //指定窗口类型
  .apply((black, white) => (black._1,black._2 + white._2)) //应用
JoinFunciton
```

3. 会话窗口关联

会话窗口是根据Session Gap将数据集划分成不同的窗口，会话窗口关联对两个Stream的数据元素进行窗口关联操作，窗口中含有两个数据集元素，并且元素具有相同的key，则输出关联计算结果，如图4-17所示，黑色数据元素和白色数据元素在相同的Session窗口中根据相同的key进行关联，并基于每个窗口中计算结果并输出。

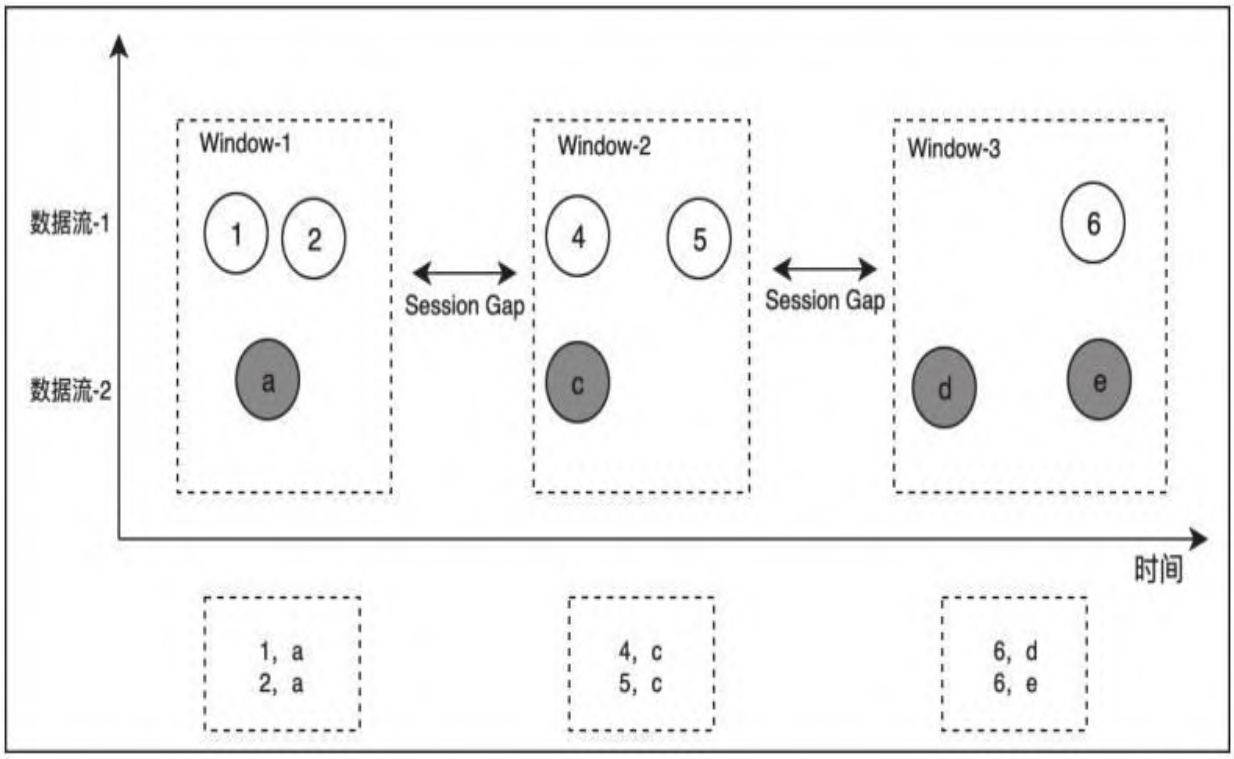

图4-17　会话窗口关联数据计算过程

可以看出会话窗口中数据关联也是内连接，也就是在当窗口中两个DataStream必须同时含有相同key的数据元素，然后进行结果输出否则不输出窗口计算结果。虽然窗口不会重叠，但有可能会出现不连续的情况，例如两个DataStream的会话时间不在一个频道上，可能导致一段时间内两个DataStream都无法连接，这时需要根据情况重新设置Gap的大小。会话窗口关联代码实现如下，只需要在window中指定EventTimeSessionWindows类型即可，其余部分和其他窗口类似。

```
  //创建黑色元素数据集
  val blackStream: DataStream[(Int, Long)] = ...
  //创建白色元素数据集
  val whiteStream: DataStream[(Int, Long)] = ...
  //通过Join方法将两个数据集进行关联
val windowStream: DataStream[(Int,Long)] =
blackStream.join(whiteStream)
  .where(_._1) //指定第一个Stream的关联Key
  .equalTo(_._1) //指定第二个Stream的关联Key
  .window(EventTimeSessionWindows.withGap(Time.milliseconds(10))//
指定窗口类型
  .apply((black, white) =>(black._1,black._2 + white._2)) //应用
JoinFunciton
```

---

4. 间隔关联

和其他窗口关联不同，间隔关联的数据元素关联范围不依赖窗口划分，而是通过DataStream元素的时间加上或减去指定Interval作为关联窗口，然后和另外一个DataStream的数据元素时间在窗口内进行Join操作。

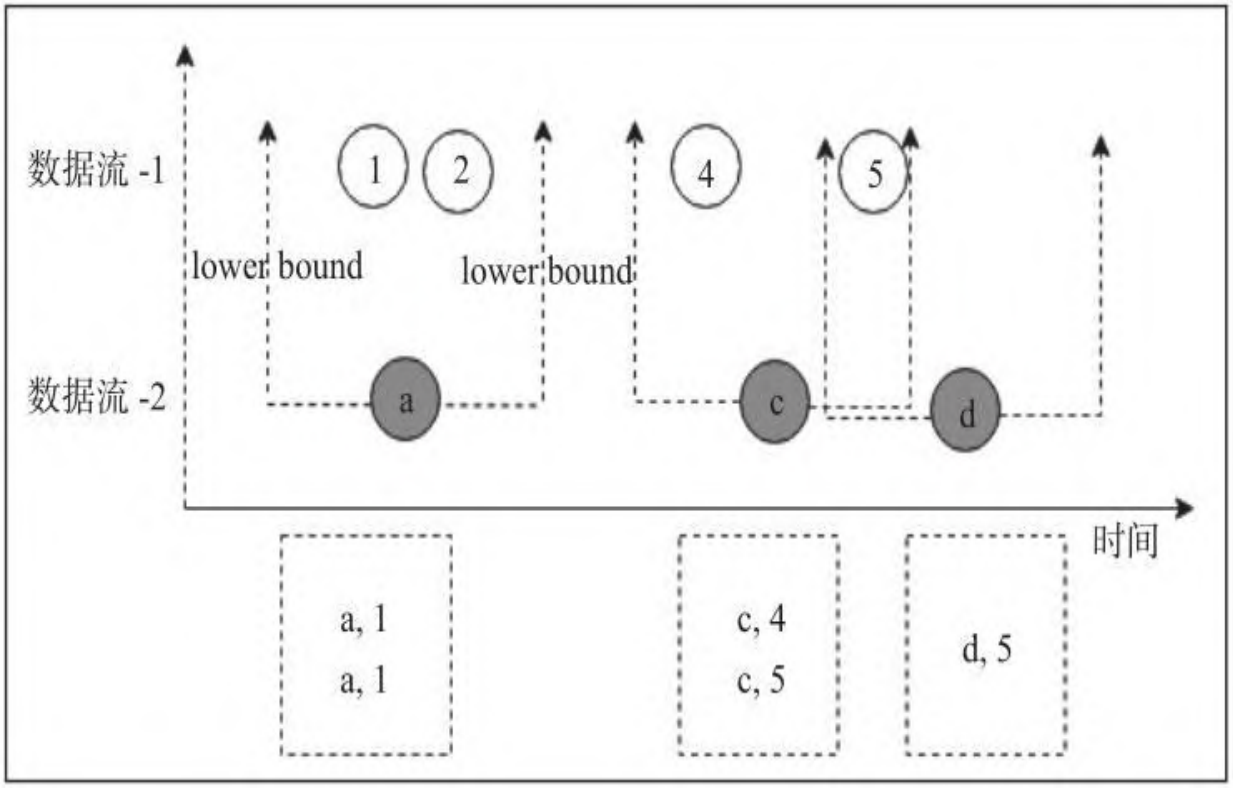

图4-18　间隔关联数据计算过程

间隔关联结果依赖于Inteval的大小，Interval越大，则关联的数据越多，反之亦然。如下代码所示，blackStream调用IntervalJoin关联whiteStream，此时blackStream和whiteStream都是转换成KeyedStream类型，同时在between()方法中指定blackStream数据元素的上下界，然后在process()方法中指定定义的ProcessJoinFunction，完成指定窗口内的数据计算和输出。

```
//创建黑色元素数据集
val blackStream: DataStream[(Int, Long)] = env.fromElements((2,
21L), (4, 1L), (5, 4L))
//创建白色元素数据集
val whiteStream: DataStream[(Int, Long)] = env.fromElements((2,
21L), (1, 1L), (3, 4L))
//通过Join方法将两个数据集进行关联
val windowStream: DataStream[String] = blackStream.keyBy(_._1)
//调用intervalJoin方法关联另外一个DataStream
  .intervalJoin(whiteStream.keyBy(_._1))
//设定时间上限和下限
```

```
  .between(Time.milliseconds(-2), Time.milliseconds(1))
  .process(new ProcessWindowFunciton())
//通过单独定义ProcessWindowFunciton实现ProcessJoinFunction
class ProcessWindowFunciton extends ProcessJoinFunction[(Int,
Long), (Int, Long), String] {
  override def processElement(in1: (Int, Long), in2: (Int, Long),
context: ProcessJoinFunction[(Int, Long), (Int, Long),
String]#Context, collector: Collector[String]): Unit = {
    collector.collect(in1  + ":" + (in1._2 + in2._2))
  }
}
```

4.4 作业链和资源组

在Flink作业中，用户可以指定相应的链条将相关性非常强的转换操作绑定在一起，这样能够让转换过程上下游的Task在同一个Pipeline中执行，进而避免因为数据在网络或者线程间传输导致的开销。一般情况下Flink在例如Map类型的操作中默认开启TaskChain，以提高Flink作业的整体性能。Flink同时也为用户提供细粒度的链条控制，用户能够根据自己的需要创建作业链或禁止作业链。

4.4.1 作业链

（1）禁用全局链条

用户能够通过禁止全局作业链的操作来关闭整个Flink作业的链条，需要注意这个操作会影响整个作业的执行情况，在选择关闭之前，用户需要非常清楚作业的执行过程，否则可能会出现一些意想不到的情况。

```
StreamExecutionEnvironment.disableOperatorChaining()
```

关闭全局作业链后，创建对应Opecator的链条，需要用户事先指定操作符，然后再通过使用startNewChain()方法来创建，且创建的链条只对当前的操作符和之后的操作符有效，不影响其他操作，例如：

```
someStream.filter(...).map(...).startNewChain().map(...)
```

在上述过程中，新创建的作业链只针对两个map操作进行链条绑定，对前面的filter操作无效，如果用户需要创建，则可以在filter操作和map操作之间进行startNewChain()方法即可。

（2）禁用局部链条

如果用户不想关闭整体作业算子上的链条，而是只想关闭某些操作符上的链条，可以通过disableChaining()方法来禁用当前操作符上的链条，例如：

```
someStream.map(...).disableChaining();
```

在上述过程中，只会禁用map操作上的链条，且不会对其他操作符产生影响。

4.4.2 Slots资源组

Slot是整个Flink系统中所提供的资源最小单元，其概念上和Yarn中的Container类似，Flink在TaskManager启动后，会自动管理当前TaskManager上所能提供的Slot，在作业提交到Flink集群后，由JobManager进行统一分配Slots数量。在这里需要介绍的是Slot资源共享的问题，即多个Task计算过程中在同一个Slot中进行，这样能够对特定过程中的Tasks进行物理隔离，使其能够在同一个Slot中执行，对数据转换操作进行隔离。另外，如果当前操作符（operator）中所有input操作均具有相同的slot group，则该操作符会继承前面操作符的slot group，然后在同一个Slot中进行数据处理，如果不是则当前的操作符会选择默认的slot group("default")，然后经作业发送到对应的slot上执行。如果用户不显示指定slot group，则所有的操作符均在default slot group中执行操作，而默认情况slot相互之间没有隔离，故Task上的操作符执行可能会在不同的slot上转换执行。可以通过如下方法指定slot group：

```
someStream.filter(...).slotSharingGroup("name");
```

这样就可以创建一个名为name的slot group，将filter操作指定在slot group中对应的slot上执行计算操作。

4.5 Asynchronous I/O异步操作

在使用Flink处理流式数据的过程中，会经常和外部系统进行数据交互。通常情况下在Flink中可以通过RichMapFunction来创建外部数据库系统的Client连接，然后通过Client连接将数据元素写入外部存储系统中或者从外部存储系统中读取数据。考虑到连接外部系统的网络等因素，这种同步查询和操作数据库的方式往往会影响整个函数的处理效率，用户如果想提升应用的处理效率，就必须考虑增加算子的并行度，这将导致大量的资源开销。Flink在1.2版本中引入了Asynchronous I/O，能够支持通过异步方式连接外部存储系统，以提升Flink系统与外部数据库交互的性能及吞吐量，但前提是数据库本身需要支持异步客户端。

如代码清单4-26所示，自定义实现AsyncFunction，创建数据库异步客户端DBClient，从数据库中异步获取数据，然后通过调用callback方法将结果返回给ResultFuture对象，完成从数据库中查询数据并创建下游DataStream数据集的操作。

代码清单4-26 自定义实现AsyncFunction，实现数据从DB中获取逻辑

```
//通过实例化AsyncFunction接口，实现数据库数据异步查询
class AsyncDBFunction extends AsyncFunction[String, (String,
String)] {
  //创建数据连接的异步客户端，能够支持数据的异步查询
  lazy val dblient: DBClient = new DBClient(host, post)
  //用于future callbacks的context
  implicit lazy val executor: ExecutionContext =
ExecutionContext.fromExecutor(Executors.directExecutor())
```

```
    //复写asyncInvoke方法，实现AsyncFunction触发数据库数据查询
    override def asyncInvoke(str: String, resultFuture:
ResultFuture[(String, String)]): Unit = {
      //通过client查询数据传入字符串，返回Future对象
      val resultFutureRequested: Future[String] =
dblient.query(str)
      //当客户端查询完请求后，通过调用callback将查询结果返回给resultFuture
      resultFutureRequested.onSuccess {
        case result: String =>
resultFuture.complete(Iterable((str, result)))
    }}}
//创建DataStream数据集
val stream: DataStream[String] = ...
// 在现有的DataStream上应用创建好的AsyncDatabaseRequest方法，返回查询结果
后回应的数据集
val resultStream: DataStream[(String, String)] =
  AsyncDataStream.unorderedWait(stream, new
AsyncDatabaseRequest(), 1000, TimeUnit.MILLISECONDS, 100)
```

通过使用Asynchronous I/O的方式会在很大程度上提升Flink系统的性能和吞吐量，主要原因是在异步函数中可以尽可能异步并发地查询外部数据库。在异步IO中需要考虑对数据库查询超时以及并发线程数控制两个因素。Asynchronous I/O提供了Timeout和Capacity两个参数来配置异步数据IO操作，其中Timeout是定义Asynchronous请求最长等待时间，超过该时间Flink将认为查询数据超时而请求失败，避免因请求无法按时返回导致系统长时间等待的情况，对于超时的请求可以通过复写AsyncFunction中的timeout方法来处理。Capacity定义在同一时间点异步请求并发数，通过Capacity参数来控制总的请求数，一旦Capacity定义的请求数被耗尽，Flink会直接触发反压机制来抑制上游数据的接入，从而保证Flink系统的正常运行。

使用异步IO方式进行数据输出，其输出结果的先后顺序有可能并不是按照之前原有数据的顺序进行排序，因此在Flink异步IO中，需要用户显式指定是否对结果排序输出，而是否排序同样影响着结果的顺序和系统性能，下面针对结果是否进行排序输出进行对比：

· 乱序模式：异步查询结果输出中，原本数据元素的顺序可能会发生变化，请求一旦完成就会输出结果，可以使用AsyncDataStream.unorderedWait(...)方法应用这种模式。如果系统同时选择使用Process Time特征具有最低的延时和负载。

· 顺序模式：异步查询结果将按照输入数据元素的顺序输出，原本Stream数据元素的顺序保持不变，这种情况下具有较高的时延和负载，因为结果数据需要在Operator的Buffer中进行缓存，直到所有异步请求处理完毕，将按照原来元素顺序输出结果，这也将对Checkpointing过程造成额外的延时和性能损耗。可以使用AsyncDataStream.orderedWait(...)方法使用这种模式。

另外在使用Event-Time时间概念处理流数据的过程中，Asynchronous I/O Operator总能够正确保持Watermark的顺序，即使使用乱序模式，输出Watermark也会保持原有顺序，但对于在Watermark之间的数据元素则不保持原来的顺序，也就是说如果使用了Watermark，将会对异步IO造成一定的时延和开销，具体取决于Watermark的频率，频率越高时延越高同时开销越大。

4.6 本章小结

本章重点介绍了如何使用Flink DataStream API编写流式应用以及DataStream API所支持的高级特性等。4.1节介绍了DataStream API编程模型以及开发环境准备，并通过简单的实例介绍如何使用DataStream API编写简单的流式应用，同时介绍了DataStream API中DataSource数据源支持，包括基本数据源和外部第三方数据源，以及DataStream API提供的常规的数据处理方法，包括针对单个Stream和多个Stream的转换方法，然后介绍利用DataSink组件将DataStream的数据输出到外部系统中。4.2节介绍了Flink DataStream API中事件驱动以及Watermark机制，了解如何通过EventTime和Watermark处理乱序数据。4.3节介绍了DataStream API支持的窗口计算方法，包括窗口Assigner、Trigger、Evictor、WindowFucntion的使用，并介绍了多流合并的内容，以及如何对延迟数据进行处理。4.5节介绍了Flink在数据输出过程中支持的异步I/O的特性，提升Flink和外部系统交互的性能和效率，降低系统的资源消耗。

第5章

Flink状态管理和容错

本章将重点介绍Flink对有状态计算的支持，其中包括有状态计算和无状态计算的区别，以及在Flink中支持的不同状态类型，分别有Keyed State和Operator State。另外针对状态数据的持久化，以及整个Flink任务的数据一致性保障，Flink提供了Checkpoint机制处理和持久化状态结果数据。最后针对状态数据Flink提供了不同的状态管理器来管理状态数据，例如MemoryStateBackend等。

5.1　有状态计算

在Flink架构体系中，有状态计算可以说是Flink非常重要的特性之一。有状态计算是指在程序计算过程中，在Flink程序内部存储计算产生的中间结果，并提供给后续Function或算子计算结果使用。如图5-1所示，状态数据一致维系在本地存储中，这里的存储可以是Flink的堆内存或者堆外内存，也可以借助第三方的存储介质，例如Flink中已经实现的RocksDB，当然用户也可以自己实现相应的缓存系统去存储状态信息，以完成更加复杂的计算逻辑。和状态计算不同的是，无状态计算不会存储计算过程中产生的结果，也不会将结果用于下一步计算过程中，程序只会在当前的计算流程中实行计算，计算完成就输出结果，然后下一条数据接入，然后再处理。

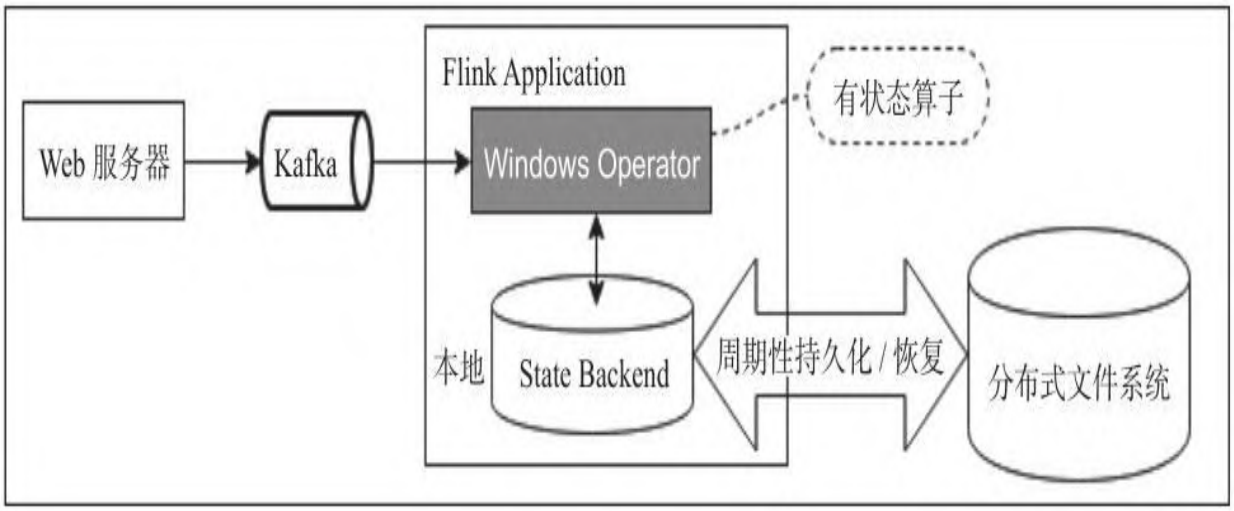

图5-1　Flink状态计算示意图

无状态计算实现的复杂度相对较低，实现起来较容易，但是无法完成提到的比较复杂的业务场景，例如下面的例子：

· 用户想实现CEP（复杂事件处理），获取符合某一特定事件规则的事件，状态计算就可以将接入的事件进行存储，然后等待符合规则的事件触发；

· 用户想按照分钟、小时、天进行聚合计算，求取当前的最大值、均值等聚合指标，这就需要利用状态来维护当前计算过程中产生的结果，例如事件的总数、总和以及最大，最小值等；

· 用户想在Stream上实现机器学习的模型训练，状态计算可以帮助用户维护当前版本模型使用的参数；

· 用户想使用历史的数据进行计算，状态计算可以帮助用户对数据进行缓存，使用户可以直接从状态中获取相应的历史数据。

以上场景充分说明了状态计算在整个流式计算过程中重要性，可以看出，在Flink引入状态这一特性，能够极大地提升流式计算过程中数据的使用范围以及指标计算的复杂度，而不再需要借助类似于Redis外部缓存存储中间结果数据，这种方式需要频繁地和外部系统交互，并造成大量系统性能开销，且不易保证数据在传输和计算过程中的可靠性，当外部存储发生变化，就可能会影响到Flink内部的计算结果。

Flink状态类型及应用

1. 状态类型

在Flink中根据数据集是否根据Key进行分区，将状态分为Keyed State和Operator State（Non-keyed State）两种类型。

（1）Keyed State

表示和key相关的一种State，只能用于KeydStream类型数据集对应的Functions和Operators之上。Keyed State是Operator State的特例，区别在于Keyed State事先按照key对数据集进行了分区，每个Key State仅对应一个Operator和Key的组合。Keyed State可以通过Key

Groups进行管理，主要用于当算子并行度发生变化时，自动重新分布Keyed Sate数据。在系统运行过程中，一个Keyed算子实例可能运行一个或者多个Key Groups的keys。

（2）Operator State

与Keyed State不同的是，Operator State只和并行的算子实例绑定，和数据元素中的key无关，每个算子实例中持有所有数据元素中的一部分状态数据。Operator State支持当算子实例并行度发生变化时自动重新分配状态数据。

同时在Flink中Keyed State和Operator State均具有两种形式，其中一种为托管状态（Managed State）形式，由Flink Runtime中控制和管理状态数据，并将状态数据转换成为内存Hash tables或RocksDB的对象存储，然后将这些状态数据通过内部的接口持久化到Checkpoints中，任务异常时可以通过这些状态数据恢复任务。另外一种是原生状态（Raw State）形式，由算子自己管理数据结构，当触发Checkpoint过程中，Flink并不知道状态数据内部的数据结构，只是将数据转换成bytes数据存储在Checkpoints中，当从Checkpoints恢复任务时，算子自己再反序列化出状态的数据结构。DataStream API支持使用Managed State和Raw State两种状态形式，在Flink中推荐用户使用Managed State管理状态数据，主要原因是Managed State能够更好地支持状态数据的重平衡以及更加完善的内存管理。

2.Managed Keyed State

Flink中有以下Managed Keyed State类型可以使用，每种状态都有相应的使用场景，用户可以根据实际需求选择使用。

· ValueState[T]：与Key对应单个值的状态，例如统计user_id对应的交易次数，每次用户交易都会在count状态值上进行更新。ValueState对应的更新方法是update(T)，取值方法是T value()；

· ListState[T]：与Key对应元素列表的状态，状态中存放元素的List列表。例如定义ListState存储用户经常访问的IP地址。在ListState中添加元素使用add(T)或者addAll(List[T])两个方法，获取元素使用Iterable get()方法，更新元素使用update(List[T])方法；

· ReducingState[T]：定义与Key相关的数据元素单个聚合值的状态，用于存储经过指定ReduceFucntion计算之后的指标，因此，ReducingState需要指定ReduceFucntion完成状态数据的聚合。ReducingState添加元素使用add(T)方法，获取元素使用T get()方法；

· AggregatingState[IN,OUT]:定义与Key对应的数据元素单个聚合值的状态，用于维护数据元素经过指定AggregateFunciton计算之后的指标。和ReducingState相比，AggregatingState输入类型和输出类型不一定是相同的，但ReducingState输入和输出必须是相同类型的。和ListState相似，AggregatingState需要指定AggregateFunciton完成状态数据的聚合操作。AggregatingState添加元素使用add(IN)方法，获取元素使用OUT get()方法。

· MapState[UK,UV]：定义与Key对应键值对的状态，用于维护具有key-value结构类型的状态数据，MapState添加元素使用put(UK,UV)或者putAll(Map[UK,UV]方法，获取元素使用get(UK)方法。和HashMap接口相似，MapState也可以通过entries()、keys()、values()获取对应的keys或values的集合。

在Flink中需要通过创建StateDescriptor来获取相应State的操作类。StateDescriptor主要定义了状态的名称、状态中数据的类型参数信息以及状态自定义函数。每种Managed Keyed State有相应的StateDescriptor，例如ValueStateDescriptor、ListStateDescriptor、ReducingState-Descriptor、FoldingStateDescriptor、MapStateDescriptor等。

（1）Stateful Funciton定义

接下来通过完整的实例来说明如何在RichFlatmapFunction中使用ValueState，完成对接入数据最小值的获取。如代码清单5-1所示，通过定义leastValueState存储系统中指标的最小值，并在每次计算时和当前接入的数据对比，如果当前元素的数值小于状态中的最小值，则更新状态。然后在输出操作中增加对应指标的最小值作为新的数据集的字段。

代码清单5-1　通过创建ValueState来获取指标的最小值

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
//创建元素数据集
val inputStream: DataStream[(Int, Long)] = env.fromElements((2,
21L), (4, 1L), (5, 4L))
inputStream.keyBy(_._1).flatMap {
  //定义和创建RichFlatMapFunction,第一个参数为输入数据类型，第二个参数为输出
数据类型
  new RichFlatMapFunction[(Int, Long), (Int, Long, Long)] {
//
    private var leastValueState: ValueState[Long] = _
    override def open(parameters: Configuration): Unit = {
      //创建ValueStateDescriptor,定义状态名称为leastValue,并指定数据类型
      val leastValueStateDescriptor = new
ValueStateDescriptor[Long]("leastValue", classOf[Long])
      //通过getRuntimeContext.getState获取State
      leastValueState =
getRuntimeContext.getState(leastValueStateDescriptor)
    }
    override def flatMap(t: (Int, Long), collector: Collector[(Int,
Long, Long)]): Unit = {
      //通过value方法从leastValueState中获取最小值
      val leastValue = leastValueState.value()
      //如果当前指标大于最小值，则直接输出数据元素和最小值
      if (t._2 > leastValue) {
        collector.collect((t._1, t._2, leastValue))
      } else {
        //如果当前指标小于最小值，则更新状态中的最小值
        leastValueState.update(t._2)
        //将当前数据中的指标作为最小值输出
        collector.collect((t._1, t._2, t._2))
      }}}}
```

从以上代码实例中可以看出，在定义的RichFlatMapFunction接口中，Flink提供了RuntimeContext用于获取状态数据，同时RuntimeContext提供了常用的Managed Keyd State的获取方式，可以通过创建相应的StateDescriptor并调用RuntimeContext方法来获取状态数据。例如获取ValueState可以调用ValueState[T] getState(ValueStateDescriptor[T])方法，获取ReducingState可以调用ReducingState[T] getReducingState(ReducingStateDescriptor[T])方法。

（2）State生命周期

对于任何类型Keyed State都可以设定状态的生命周期（TTL），以确保能够在规定时间内及时地清理状态数据。状态生命周期功能可以通过StateTtlConfig配置，然后将StateTtlConfig配置传入StateDescriptor中的enableTimeToLive方法中即可。Keyed State配置实例如代码清单5-2所示。

代码清单5-2　状态生命周期配置

```
//创建StateTtlConfig
val stateTtlConfig = StateTtlConfig
  //指定TTL时长为10s
  .newBuilder(Time.seconds(10))
  //指定TTL刷新时只对创建和写入操作有效
  .setUpdateType(StateTtlConfig.UpdateType.OnCreateAndWrite)
  //指定状态可见性为永远不反悔过期数据

.setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpir
ed)
  .build
//创建ValueStateDescriptor
val valueStateDescriptor = new ValueStateDescriptor[String]
("valueState", classOf[Long])
//指定创建好的stateTtlConfig
valueStateDescriptor.enableTimeToLive(stateTtlConfig)
```

在StateTtlConfig中除了通过newBuilder方法中设定过期时间的参数是必需的之外，其他参数都是可选的或使用默认值。其中setUpdateType方法中传入的类型有两种：

· StateTtlConfig.UpdateType.OnCreateAndWrite仅在创建和写入时更新TTL；

· StateTtlConfig.UpdateType. OnReadAndWrite所有读与写操作都更新TTL。

需要注意的是，过期的状态数据根据UpdateType参数进行配置，只有被写入或者读取的时间才会更新TTL，也就是说如果某个状态指标一直不被使用或者更新，则永远不会触发对该状态数据的清理操作，这种情况可能会导致系统中的状态数据越来越大。目前用户可以使用

StateTtlConfig. cleanupFullSnapshot设定当触发State Snapshot的时候清理状态数据，需要注意这个配置不适合用于RocksDB做增量Checkpointing的操作。

另外可以通过setStateVisibility方法设定状态的可见性，根据过期数据是否被清理来确定是否返回状态数据。

StateTtlConfig.StateVisibility.NeverReturnExpired：状态数据过期就不会返回（默认）；

StateTtlConfig.StateVisibility.ReturnExpiredIfNotCleanedUp：状态数据即使过期但没有被清理依然返回。

（3）Scala DataStream API中直接使用状态

除了像上一小节中通过定义RichFlatMapFunction或者FichMapFunction操作状态之外，Flink Scala版本的DataStream API提供了快捷方式来创建和查询状态数据。在KeyedStream接口中提供了filterWithState、mapWithState、flatMapWithState三种方法来定义和操作状态数据，以mapWithState为例，可以在mapWithState指定输入参数类型和状态数据类型，系统会自动创建count对应的状态来存储每次更新的累加值，整个过程不需要像实现RichFunciton那样操作状态数据，如代码清单5-3所示。

代码清单5-3　使用mapuwithState直接操作状态数据

```
//创建元素数据集
val inputStream: DataStream[(Int, Long)] = env.fromElements((2,
21L), (4, 1L), (5, 4L))
val counts: DataStream[(Int, Int)] = inputStream
  .keyBy(_._1)
  //指定输入参数类型和状态参数类型
  .mapWithState((in: (Int, Long), count: Option[Int]) =>
    //判断count类型是否非空
    count match {
        //输出key,count,并在原来的count数据上累加
      case Some(c) => ((in._1, c), Some(c + in._2))
        //如果输入状态为空，则将指标填入
      case None => ((in._1, 0), Some(in._2))
    })
```

3. Managed Operator State

Operator State是一种non-keyed state，与并行的操作算子实例相关联，例如在Kafka Connector中，每个Kafka消费端算子实例都对应到Kafka的一个分区中，维护Topic分区和Offsets偏移量作为算子的Operator State。在Flink中可以实现Checkpointed-Function或者ListCheckpointed<T extends Serializable>两个接口来定义操作Managed Operator State的函数。

（1）通过CheckpointedFunction接口操作Operator State

CheckpointedFunction接口定义如代码清单5-4所示，需要实现两个方法，当checkpoint触发时就会调用snapshotState()方法，当初始化自定义函数的时候会调用initializeState()方法，其中包括第一次初始化函数和从之前的checkpoints中恢复状态数据，同时initializeState()方法中需要包含两套逻辑，一个是不同类型状态数据初始化的逻辑，另外一个是从之前的状态中恢复数据的逻辑。

代码清单5-4　CheckpointedFunction接口定义

```
public interface CheckpointedFunction {
  //每当checkpoint触发时，调用此方法
void snapshotState(FunctionSnapshotContext context) throws
Exception;
  //每次自定义函数初始化的时候，调用此方法初始化状态
void initializeState(FunctionInitializationContext context) throws
Exception;}
```

在每个算子中Managed Operator State都是以List形式存储，算子和算子之间的状态数据相互独立，List存储比较适合于状态数据的重新分布，Flink目前支持对Managed Operator State两种重分布的策略，分别是Even-split Redistribution和Union Redistribution。

· Even-split Redistribution：每个算子实例中含有部分状态元素的List列表，整个状态数据是所有List列表的合集。当触发restore/redistribution动作时，通过将状态数据平均分配成与算子并行度

相同数量的List列表，每个task实例中有一个List，其可以为空或者含有多个元素。

· Union Redistribution：每个算子实例中含有所有状态元素的List列表，当触发restore/redistribution动作时，每个算子都能够获取到完整的状态元素列表。

例如可以通过实现FlatMapFunction和CheckpointedFunction完成对输入数据中每个key的数据元素数量和算子的元素数量的统计。如代码清单5-5所示，通过在initializeState()方法中分别创建keyedState和operatorState两种状态，存储基于Key相关的状态值以及基于算子的状态值。

代码清单5-5　实现CheckpointedFunction接口利用Operator State统计输入到算子的数据量

```
private class CheckpointCount(val numElements: Int)
  extends FlatMapFunction[(Int, Long), (Int, Long, Long)] with
CheckpointedFunction {
  //定义算子实例本地变量，存储Operator数据数量
  private var operatorCount: Long = _
  //定义keyedState，存储和Key相关的状态值
  private var keyedState: ValueState[Long] = _
  //定义operatorState，存储算子的状态值
  private var operatorState: ListState[Long] = _
  override def flatMap(t: (Int, Long), collector: Collector[(Int,
Long, Long)]): Unit = {
    val keyedCount = keyedState.value() + 1
    //更新keyedState数量
    keyedState.update(keyedCount)
    //更新本地算子operatorCount值
    operatorCount = operatorCount + 1
    //输出结果，包括id,id对应的数量统计keyedCount，算子输入数据的数量统计
operatorCount
    collector.collect((t._1, keyedCount, operatorCount))
  }

  //初始化状态数据
  override def initializeState(context:
FunctionInitializationContext): Unit = {
    //定义并获取keyedState
    keyedState = context.getKeyedStateStore.getState(
```

```
      new ValueStateDescriptor[Long](
        "keyedState", createTypeInformation[Long]))
    //定义并获取operatorState
    operatorState = context.getOperatorStateStore.getListState(
      new ListStateDescriptor[Long](
        "operatorState", createTypeInformation[Long]))
    //定义在Restored过程中，从operatorState中恢复数据的逻辑
    if (context.isRestored) {
      operatorCount = operatorState.get().asScala.sum
    }
  }
    //当发生snapshot时，将operatorCount添加到operatorState中
  override def snapshotState(context: FunctionSnapshotContext):
Unit = {
    operatorState.clear()
    operatorState.add(operatorCount)
    }}
```

可以从上述代码中看到的是，在snapshotState()方法中清理掉上一次checkpoint中存储的operatorState的数据，然后再添加并更新本次算子中需要checkpoint的operatorCount状态变量。当系统重启时会调用initializeState方法，重新恢复keyedState和operatorState，其中operatorCount数据可以从最新的operatorState中恢复。

对于状态数据重分布策略的使用，可以在创建operatorState的过程中通过相应的方法指定：如果使用Even-split Redistribution策略，则通过context. getListState(descriptor)获取Operator State；如果使用Union Redistribution策略，则通过context. getUnionList State(descriptor)来获取。实例代码中默认使用的Even-split Redistribution策略。

（2）通过ListCheckpointed接口定义Operator State

ListCheckpointed接口和CheckpointedFunction接口相比在灵活性上相对弱一些，只能支持List类型的状态，并且在数据恢复的时候仅支持even-redistribution策略。在ListCheckpointed接口中需要实现以下两个方法来操作Operator State：

```
List<T> snapshotState(long checkpointId, long timestamp) throws
Exception;
void restoreState(List<T> state) throws Exception;
```

其中snapshotState方法定义数据元素List存储到checkpoints的逻辑，restoreState方法则定义从checkpoints中恢复状态的逻辑。如果状态数据不支持List形式，则可以在snapshotState方法中返回Collections.singletonList(STATE)。如代码清单5-6所示，通过实现FlatMapFunction接口和ListCheckpointed接口完成对输入到FlatMapFunction算子中的数据量统计，同时在函数中实现了snapshotState方法，将本地定义的算子变量numberRecords写入Operator State中，并通过restoreState方法从状态中恢复numberRecords数据。

代码清单5-6 实现ListCheckpointed接口利用Operator State统计算子输入数据量

```
class numberRecordsCount extends FlatMapFunction[(String, Long),
(String,
Long)] with ListCheckpointed[Long] {
  //定义算子中接入的numberRecords数量
  private var numberRecords: Long = 0L
  override def flatMap(t: (String, Long), collector:
Collector[(String, Long)]): Unit = {
    //接入一条记录则进行统计，并输出
    numberRecords += 1
    collector.collect(t._1, numberRecords)
  }
  override def snapshotState(checkpointId: Long, ts: Long):
util.List[Long] = {
    //Snapshot状态的过程中将numberRecords写入
    Collections.singletonList(numberRecords)
  }
  override def restoreState(list: util.List[Long]): Unit = {
    numberRecords = 0L
    for (count <- list) {
      //从状态中恢复numberRecords数据
      numberRecords += count }}}
```

5.2 Checkpoints和Savepoints

5.2.1 Checkpoints检查点机制

Flink中基于异步轻量级的分布式快照技术提供了Checkpoints容错机制，分布式快照可以将同一时间点Task/Operator的状态数据全局统一快照处理，包括前面提到的Keyed State和Operator State。如图5-2所示，Flink会在输入的数据集上间隔性地生成checkpoint barrier，通过栅栏（barrier）将间隔时间段内的数据划分到相应的checkpoint中。当应用出现异常时，Operator就能够从上一次快照中恢复所有算子之前的状态，从而保证数据的一致性。例如在KafkaConsumer算子中维护Offset状态，当系统出现问题无法从Kafka中消费数据时，可以将Offset记录在状态中，当任务重新恢复时就能够从指定的偏移量开始消费数据。对于状态占用空间比较小的应用，快照产生过程非常轻量，高频率创建且对Flink任务性能影响相对较小。checkpoint过程中状态数据一般被保存在一个可配置的环境中，通常是在JobManager节点或HDFS上。

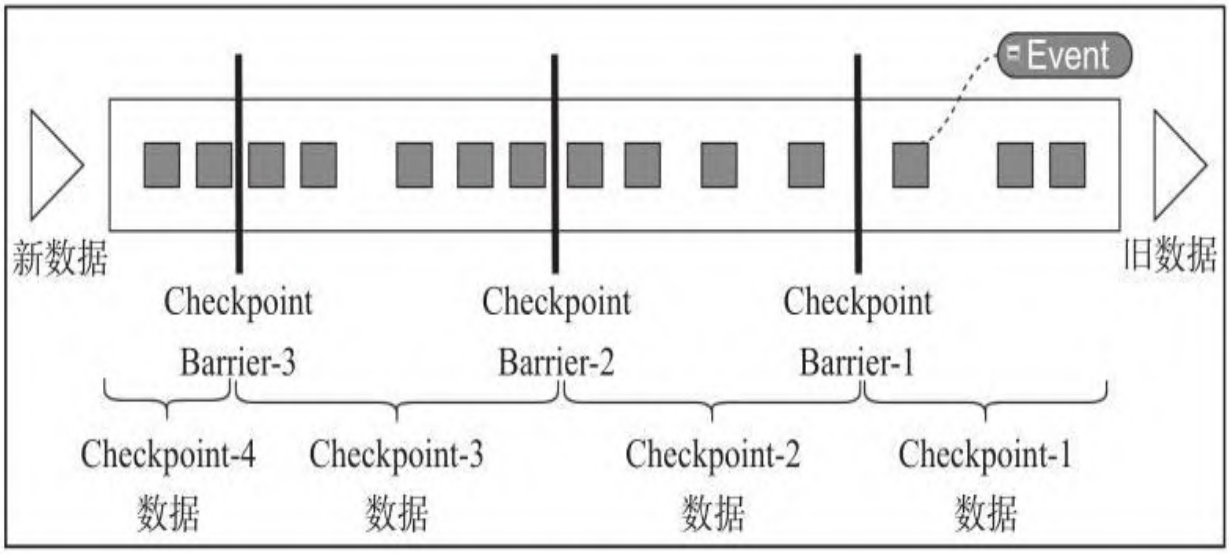

图5-2　Checkpoint机制

默认情况下Flink不开启检查点的，用户需要在程序中通过调用enable-Checkpointing(n)方法配置和开启检查点，其中n为检查点执行的时间间隔，单位为毫秒。除了配置检查点时间间隔，针对检查点配置还可以调整其他相关参数：

（1）Checkpoint开启和时间间隔指定

开启检查点并且指定检查点时间间隔为1000ms，根据实际情况自行选择，如果状态比较大，则建议适当增加该值。

```
env.enableCheckpointing(1000);
```

（2）exactly-ance和at-least-once语义选择

可以选择exactly-once语义保证整个应用内端到端的数据一致性，这种情况比较适合于数据要求比较高，不允许出现丢数据或者数据重复，与此同时，Flink的性能也相对较弱，而at-least-once语义更适合于时廷和吞吐量要求非常高但对数据的一致性要求不高的场景。如下通过setCheckpointingMode()方法来设定语义模式，默认情况下使用的是exactly-once模式。

```
env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);
```

（3）Checkpoint超时时间

超时时间指定了每次Checkpoint执行过程中的上限时间范围，一旦Checkpoint执行时间超过该阈值，Flink将会中断Checkpoint过程，并按照超时处理。该指标可以通过setCheckpointTimeout方法设定，默认为10分钟。

```
env.getCheckpointConfig().setCheckpointTimeout(60000);
```

（4）检查点之间最小时间间隔

该参数主要目的是设定两个Checkpoint之间的最小时间间隔，防止出现例如状态数据过大而导致Checkpoint执行时间过长，从而导致Checkpoint积压过多，最终Flink应用密集地触发Checkpoint操作，会占用了大量计算资源而影响到整个应用的性能。

```
env.getCheckpointConfig().setMinPauseBetweenCheckpoints(500);
```

（5）最大并行执行的检查点数量

通过setMaxConcurrentCheckpoints()方法设定能够最大同时执行的Checkpoint数量。在默认情况下只有一个检查点可以运行，根据用户指定的数量可以同时触发多个Checkpoint，进而提升Checkpoint整体的效率。

```
env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);
```

（6）外部检查点

设定周期性的外部检查点，然后将状态数据持久化到外部系统中，使用这种方式不会在任务正常停止的过程中清理掉检查点数据，而是会一直保存在外部系统介质中，另外也可以通过从外部检查点中对任务进行恢复。

```
env.getCheckpointConfig().enableExternalizedCheckpoints(ExternalizedCheckp
ointCleanup.RETAIN_ON_CANCELLATION);
```

（7）failOnCheckpointingErrors

failOnCheckpointingErrors参数决定了当Checkpoint执行过程中如果出现失败或者错误时，任务是否同时被关闭，默认值为True。

```
env.getCheckpointConfig().setFailOnCheckpointingErrors (false);
```

5.2.2 Savepoints机制

Savepoints是检查点的一种特殊实现，底层其实也是使用Checkpoints的机制。Savepoints是用户以手工命令的方式触发Checkpoint，并将结果持久化到指定的存储路径中，其主要目的是帮助用户在升级和维护集群过程中保存系统中的状态数据，避免因为停机运维或者升级应用等正常终止应用的操作而导致系统无法恢复到原有的计算状态的情况，从而无法实现端到端的Excatly-Once语义保证。

1.Operator ID配置

当使用Savepoints对整个集群进行升级或运维操作的时候，需要停止整个Flink应用程序，此时用户可能会对应用的代码逻辑进行修改，即使Flink能够通过Savepoint将应用中的状态数据同步到磁盘然后恢复任务，但由于代码逻辑发生了变化，在升级过程中有可能导致算子的状态无法通过Savepoints中的数据恢复的情况，在这种情况下就需要通过唯一的ID标记算子。在Flink中默认支持自动生成Operator ID，但是这种方式不利于对代码层面的维护和升级，建议用户尽可能使用手工的方式对算子进行唯一ID标记，ID的应用范围在每个算子内部，具体的使用方式如代码清单5-7所示，可以通过使用Operator中提供的uid方法指定唯一ID，这样就能将算子唯一区分出来。

代码清单5-7 使用Operator ID标计算子

```
DataStream<String> stream = env.
  // Stateful source (e.g. Kafka) with ID
  .addSource(new StatefulSource())
  .uid("source-id") // ID for the source operator
  .shuffle()
  // Stateful mapper with ID
  .map(new StatefulMapper())
  .uid("mapper-id") // ID for the mapper
  // Stateless printing sink
  .print(); // Auto-generated ID
```

2.Savepoints操作

Savepoint操作可以通过命令行的方式进行触发，命令行提供了取消任务、从Savepoints中恢复任务、撤销Savepoints等操作，在Flink1.2版本以后也可以通过Flink Web页面从Savepoints中恢复应用。

（1）手动触发Savepoints

通过在Flink命令中指定“savepoint”关键字来触发Savepoints操作，同时需要在命令中指定jobId和targetDirectory（两个参数），其中jobId是需要触发Savepoints操作的Job Id编号，targetDirectory指定Savepoint数据存储路径，所有Savepoint存储的数据都会放置在该配置路径中。

```
bin/flink savepoint :jobId [:targetDirectory]
```

在Hadoop Yarn上提交的应用，需要指定Flink jobId的同时也需要通过使用yid指定YarnAppId，其他参数和普通模式一样。

```
bin/flink savepoint :jobId [:targetDirectory] -yid :yarnAppId
```

（2）取消任务并触发Savepoints

通过cancel命令将停止Flink任务的同时将自动触发Savepoints操作，并把中间状态数据写入磁盘，用以后续的任务恢复。

```
bin/flink cancel -s [:targetDirectory] :jobId
```

（3）通过从Savepoints中恢复任务

```
bin/flink run -s :savepointPath [:runArgs]
```

通过使用run命令将任务从保存的Savepoint中恢复，其中-s参数指定了Savepoint数据存储路径。通常情况下Flink通过使用savepoint可以恢复应用中的状态数据，但在某些情况下如果应用中的算子和Savepoint中的算子状态可能不一致，例如用户在新的代码中删除了某个算子，这时就会出现任务不能恢复的情况，此时可以通过--allowNonRestoredState（--n）参数来设置忽略状态无法匹配的问题，让程序能够正常启动和运行。

（4）释放Savepoints数据

```
bin/flink savepoint -d :savepointPath
```

可以通过以上--dispose（-d）命令释放已经存储的Savepoint数据，这样存储在指定路径中的savepointPath将会被清除掉。

3.TargetDirectory配置

在前面的内容中我们已经知道Flink在执行Savepoints过程中需要指定目标存储路径，但对目标路径配置除了可以在每次生成Savepoint，通过在命令行中指定之外，也可以在系统环境中配置默认的TargetDirectory，这样就不需要每次在命令行中指定。需要注意，默认路径和命令行中必须至少指定一个，否则无法正常执行Savepoint过程。

（1）默认TargetDirecy配置

在flink-conf.yaml配置文件中配置state.savepoints.dir参数，配置的路径参数需要是TaskManager和JobManager都能够访问到的路径，例如分布式文件系统Hdfs的路径。

```
state.savepoints.dir: hdfs:///flink/savepoints
```

（2）TargetDirectoy文件路径结构

TargetDirectoy文件路径结构如下，其中包括Savepoint directory以及metadata路径信息。需要注意，即便Flink任务的Savepoints数据存储在一个路径中，但目前还不支持将Savepoint路径拷贝到另外的环境中，然后通过Savepoint恢复Flink任务的操作，这主要是因为在_metadata文件中使用到了绝对路径，这点会在Flink未来的版本中得到改善，也就是支持不同环境中的Flink任务的恢复和迁移。

```
# Savepoint target directory
/savepoints/
# Savepoint directory
/savepoints/savepoint-:shortjobid-:savepointid/
# Savepoint file contains the checkpoint meta data
/savepoints/savepoint-:shortjobid-:savepointid/_metadata
# Savepoint state
/savepoints/savepoint-:shortjobid-:savepointid/...
```

5.3 状态管理器

在Flink中提供了StateBackend来存储和管理Checkpoints过程中的状态数据。

5.3.1 StateBackend类别

Flink中一共实现了三种类型的状态管理器，包括基于内存的MemoryStateBackend、基于文件系统的FsStateBackend，以及基于RockDB作为存储介质的RocksDBState-Backend。这三种类型的StateBackend都能够有效地存储Flink流式计算过程中产生的状态数据，在默认情况下Flink使用的是内存作为状态管理器，下面分别对每种状态管理器的特点进行说明。

1.MemoryStateBackend

基于内存的状态管理器将状态数据全部存储在JVM堆内存中，包括用户在使用DataStream API中创建的Key/Value State，窗口中缓存的状态数据，以及触发器等数据。基于内存的状态管理具有非常快速和高效的特点，但也具有非常多的限制，最主要的就是内存的容量限制，一旦存储的状态数据过多就会导致系统内存溢出等问题，从而影响整个应用的正常运行。同时如果机器出现问题，整个主机内存中的状态数据都会丢失，进而无法恢复任务中的状态数据。因此从数据安全的角度建议用户尽可能地避免在生产环境中使用MemoryStateBackend。

Flink将MemoryStateBackend作为默认状态后端管理器，也可以通过如下参数配置初始化MemoryStateBackend，其中“MAX_MEM_STATE_SIZE”指定每个状态值最大的内存使用大小。

```
new MemoryStateBackend(MAX_MEM_STATE_SIZE, false);
```

在Flink中MemoryStateBackend具有如下特点，需要用户在选择使用中注意：

· 聚合类算子的状态会存储在JobManager内存中，因此对于聚合类算子比较多的应用会对JobManager的内存有一定压力，进而对整个集群会造成较大负担。

· 尽管在创建MemoryStateBackend时可以指定状态初始化内存大小，但是状态数据传输大小也会受限于Akka框架通信的“akka.framesize”大小限制（默认：10485760bit），该指标表示在JobManage和TaskManager之间传输数据的最大消息容量。

· JVM内存容量受限于主机内存大小，也就是说不管是JobManager内存还是在TaskManager的内存中维护状态数据都有内存的限制，因此对于非常大的状态数据则不适合使用MemoryStateBackend存储。

因此综上可以得出，MemoryStateBackend比较适合用于测试环境中，并用于本地调试和验证，不建议在生产环境中使用。但如果应用状态数据量不是很大，例如使用了大量的非状态计算算子，也可以在生产环境中使用MemoryStateBackend，否则应该改用其他更加稳定的StateBackend作为状态管理器，例如后面讲到的FsStateBackend和RockDbStateBackend等。

2. FsStateBackend

和MemoryStateBackend有所不同，FsStateBackend是基于文件系统的一种状态管理器，这里的文件系统可以是本地文件系统，也可以是HDFS分布式文件系统。

```
new FsStateBackend(path, false);
```

如以上创建FsStateBackend的实例代码，其中path如果为本地路径，其格式为“file:///data/flink/checkpoints”，如果path为HDFS路径，其格式为“hdfs://nameservice/flink/checkpoints”。FsStateBackend中第二个Boolean类型的参数指定是否以同步的方式进行状态数据记录，默认采用异步的方式将状态数据同步到文件系统中，异步方式能够尽可能避免在Checkpoint的过程中影响流式计算任务。如果用户想采用同步的方式进行状态数据的检查点数据，则将第二个参数指定为True即可。相比于MemoryStateBackend，FsStateBackend更适合任务状态非常大的情况，例如应用中含有时间范围非常长的窗口计算，或Key/value State状态数据量非常大的场

景，这时系统内存不足以支撑状态数据的存储。同时基于文件系统存储最大的好处是相对比较稳定，同时借助于像HDFS分布式文件系统中具有三副本备份的策略，能最大程度保证状态数据的安全性，不会出现因为外部故障而导致任务无法恢复等问题。

3. RocksDBStateBackend

RocksDBStateBackend是Flink中内置的第三方状态管理器，和前面的状态管理器不同，RocksDBStateBackend需要单独引入相关的依赖包到工程中。通过初始化RockDBStateBackend类，使可以得到RockDBStateBackend实例类。

```
//创建RocksDBStateBackend实例类
new RocksDBStateBackend(path);
```

RocksDBStateBackend采用异步的方式进行状态数据的Snapshot，任务中的状态数据首先被写入RockDB中，然后再异步地将状态数据写入文件系统中，这样在RockDB仅会存储正在进行计算的热数据，对于长时间才更新的数据则写入磁盘中进行存储。而对于体量比较小的元数据状态，则直接存储在JobManager的内存中。

与FsStateBackend相比，RocksDBStateBackend在性能上要比FsStateBackend高一些，主要是因为借助于RocksDB存储了最新热数据，然后通过异步的方式再同步到文件系统中，但RocksDBStateBackend和MemoryStateBackend相比性能就会较弱一些。

需要注意的是RocksDB通过JNI的方式进行数据的交互，而JNI构建在byte[]数据结构之上，因此每次能够传输的最大数据量为2^31字节，也就是说每次在RocksDBStateBackend合并的状态数据量大小不能超过2^31字节限制，否则将会导致状态数据无法同步，这是RocksDB采用JNI方式的限制，用户在使用过程中应当注意。

综上可以看出，RocksDBStateBackend和FsStateBackend一样，适合于任务状态数据非常大的场景。在Flink最新版本中，已经提供了基于RocksDBStateBackend实现的增量Checkpoints功能，极大地提高了

状态数据同步到介质中的效率和性能，在后续的社区发展中，RocksDBStateBackend也会作为状态管理器重点使用的方式之一。

5.3.2　状态管理器配置

在StateBackend应用过程中，除了MemoryStateBackend不需要显示配置之外，其他状态管理器都需要进行相关的配置。在Flink中包含了两种级别的StateBackend配置：一种是应用层面配置，配置的状态管理器只会针对当前应用有效；另外一种是整个集群的默认配置，一旦配置就会对整个Flink集群上的所有应用有效。

1. 应用级别配置

在Flink应用中通过StreamExecutionEnvironment提供的setStateBackend()方法配置状态管理器，代码清单5-8通过实例化FsStateBackend，然后在setStateBackend方法中指定相应的状态管理器，这样后续应用的状态管理都会基于HDFS文件系统进行。

代码清单5-8　设定应用层面的StateBackend

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setStateBackend(new
FsStateBackend("hdfs://namenode:40010/flink/checkpoints"));
```

如果使用RocksDBStateBackend则需要单独引入rockdb依赖库，如代码清单5-9所示，将相关的Maven依赖配置引入到本地工程中。

代码清单5-9　RocksDBStateBackend Maven配置

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-statebackend-rocksdb_2.11</artifactId>
  <version>1.7.0</version>
</dependency>
```

经过上述配置后，如代码清单5-10所示就可以使用RocksDBStateBackend作为状态管理器进行算子或者数据的状态管理。其中需要配置的参数和FsStateBackend基本一致。

代码清单5-10　RocksDBStateBackend应用配置

```
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
env.setStateBackend(
new RocksDBStateBackend
("hdfs://namenode:40010/flink/checkpoints"));
```

2. 集群级别配置

前面已经提到除了能够在应用层面对StateBackend进行配置，应用独立使用自己的StateBackend之外，Flink同时支持在集群中配置默认的StateBackend。具体的配置项在flink-conf.yaml文件中，如下代码所示，参数state.backend指明StateBackend类型，state.checkpoints.dir配置具体的状态存储路径，代码中使用filesystem作为StateBackend，然后指定相应的HDFS文件路径作为state的checkpoint文件夹。

```
state.backend: filesystem
# Directory for storing checkpoints
state.checkpoints.dir: hdfs://namenode:40010/flink/checkpoints
```

如果在集群默认使用RocksDBStateBackend作为状态管理器，则对应在flink-conf.yaml中的配置参数如下：

```
state.backend.rocksdb.checkpoint.transfer.thread.num: 1
state.backend.rocksdb.localdir: /var/rockdb/flink/checkpoints
state.backend.rocksdb.timer-service.factory:  HEAP
```

· state.backend.rocksdb.checkpoint.transfer.thread.num：用于指定同时可以操作RocksDBStateBackend的线程数量，默认值为1，用户可以根据实际应用场景进行调整，如果状态量比较大则可以将此参数适当增大。

· state.backend.rocksdb.localdir：用于指定RocksDB存储状态数据的本地文件路径，在每个TaskManager提供该路径存储节点中的状态数据。

· state.backend.rocksdb.timer-service.factory：用于指定定时器服务的工厂类实现类，默认为“HEAP”，也可以指定为“RocksDB”。

5.4 Querable State

在Flink中将算子的状态视为头等公民，状态作为系统流式数据计算的重要数据支撑，借助于状态可以完成了相比于无状态计算更加复杂的场景。而在通常情况下流式系统中基于状态统计出的结果数据必须输出到外部系统中才能被其他系统使用，业务系统无法与流系统直接对接并获取中间状态结果。在Flink新的版本中提出可查询的状态服务，也就是说业务系统可以通过Flink提供的RestfulAPI接口直接查询Flink系统内部的状态数据。

1.Querable State架构

从图5-3中可以看出，Flink可查询状态架构中包含三个重要组件：

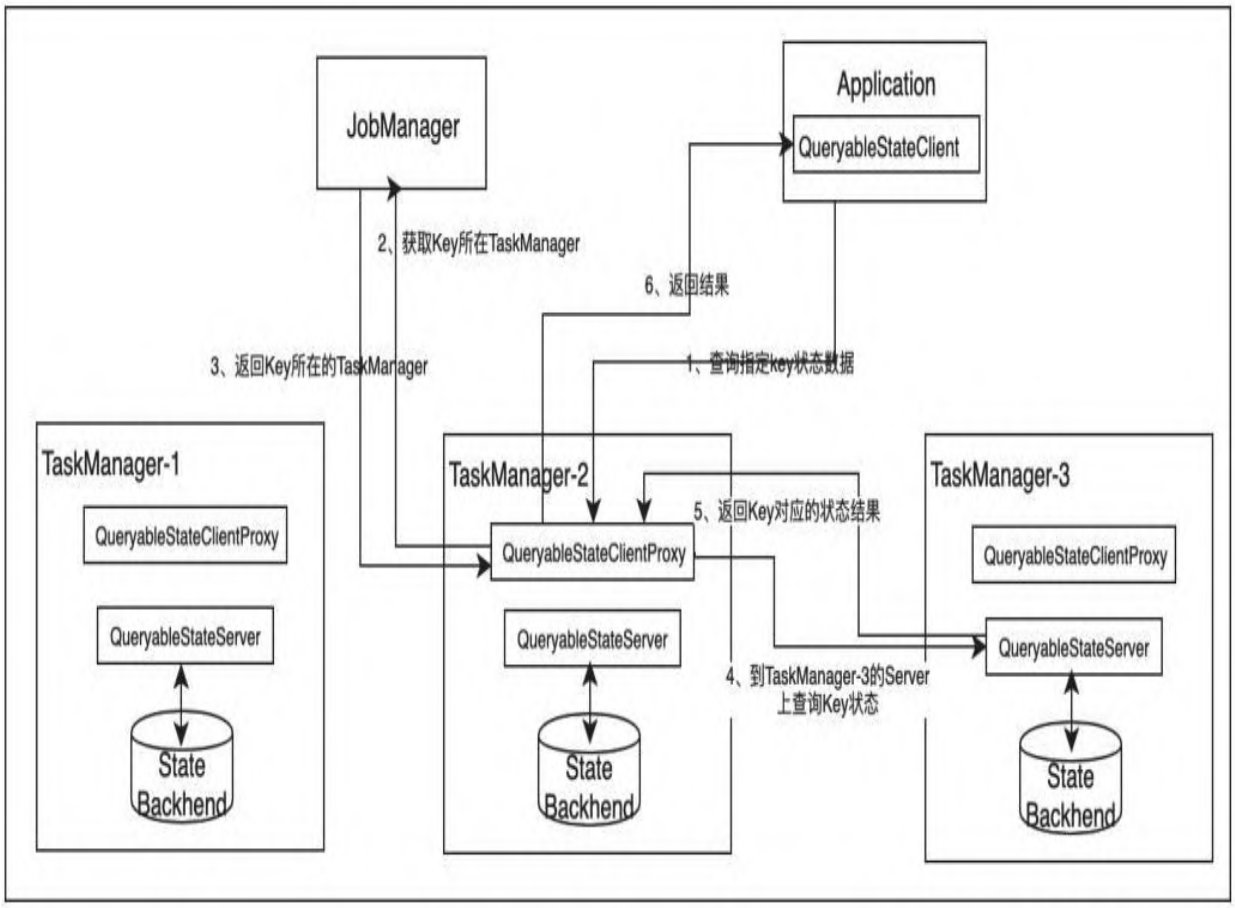

图5-3 Flink Querable State服务架构

· QueryableStateClient：用于外部应用中，作为客户端提交查询请求并收集状态查询结果。

· QueryableStateClientProxy：用于接收和处理客户端的请求，每个TaskManager上运行一个客户端代理。状态数据分布在算子所有并发的实例中，Client Proxy需要通过从JobManager中获取Key Group的分布，然后判断哪一个TaskManager实例维护了Client中传过来的Key对应的状态数据，并且向所在的TaskManager的Server发出访问请求并将查询到的状态值返回给客户端。

· QueryableStateServer：用于接收Client Proxy的请求，每个TaskManager上会运行一个State Server，该Server用于通过从本地的状态后台管理器中查询状态结果，然后返回给客户端代理。

2. 激活可查询状态服务

为了能够开启可查询状态服务，需要在Flink中引入flink-queryable-state-runtime.jar文件，可以通过将flink-queryable-state-runtime.jar从安装路径中./opt拷贝到./lib路径下引入，每次Flink集群启动时就会将flink-queryable-state-runtime.jar加载到TaskManager的环境变量中。在Flink集群启动时，Queryable State的服务就会在TaskManager中被拉起，此时就能够处理由Client发送的请求。可以通过检查TaskManager日志的方式来确认Queryable State服务是否成功启动，如果日志中出现：“Started the Queryable State Proxy Server”，则表明Queryable State服务被正常启动并可以使用。Queryable State Proxy和Server端口以及其他相关的参数均可以在flink-conf.yaml文件中配置。

3. 可查询状态应用配置

除了在集群层面激活Queryable State服务，还需要在Flink应用中修改应用程序的代码，将需要暴露的可查询状态通过配置开放出来。在代码中增加Queryable State功能相对比较简单，在创建状态的StateDescriptor中调用setQueryable(String)方法就能够将需要暴露的状态开发出来。如代码清单5-11所示，在有状态计算的算子上增加可查询状态的功能：

代码清单5-11　在Flink应用中配置可查询状态服务

```
override def open(parameters: Configuration): Unit = {
  //创建ValueStateDescriptor,定义状态名称为leastValue,并指定数据类型
  val leastValueStateDescriptor = new ValueStateDescriptor[Long]
("leastValue", classOf[Long])
  //打开可查询状态功能，让状态可以被外部应用检索
  leastValueStateDescriptor.setQueryable("leastQueryValue")
  //通过getRuntimeContext.getState获取State
  leastValueState =
getRuntimeContext.getState(leastValueStateDescriptor)
}
```

以上是通过创建StateDescripeor来设定状态是否可查询，Flink同时提供了在DataSteam数据集上来配置可查询的状态数据，具体使用

方式如代码清单5-12所示，统计5s窗口上每个key对应的最大值，并设定为可查询状态。

代码清单5-12　直接在DataSteam数据集上配置可查询状态

```
val maxInputStream: DataStream[(Int, Long)] = inputStream
.map(r => (r._1, r._2))
.keyBy(_._1)
.timeWindow(Time.seconds(5))
.max(1)
// 存储每个key在5s窗口上的最大值
maxInputStream
//根据Key进行分区，并设定为可查询状态
.keyBy(_._1).asQueryableState("maxInputState")
```

通过在KeyedStream上使用asQueryableState方法来设定可查询状态，其中返回的QueryableStateStream数据集将被当作DataSink算子，因此后面不能再接入其他算子。根据状态类型的不同，可以在asQueryableState()方法中指定不同的StatDesciptor来设定相应的可查询状态：

（1）ValueState

```
QueryableStateStream asQueryableState(
  String queryableStateName,
  ValueStateDescriptor stateDescriptor)
// ValueState的简易使用方法，不需要传入stateDescriptor
QueryableStateStream asQueryableState(String queryableStateName)
```

（2）FoldingState

```
QueryableStateStream asQueryableState(
  String queryableStateName,
  FoldingStateDescriptor stateDescriptor)
```

（3）ReducingState

```
QueryableStateStream asQueryableState(
  String queryableStateName,
  ReducingStateDescriptor stateDescriptor)
```

开启可查询状态的应用其执行过程和普通类型的应用没有特别大的区别，但需要确认的是集群环境一定要事先打开可查询状态服务，否则提交的应用将不能被正常运行。

4.通过外部应用查询状态数据

对于基于JVM的业务应用，可以通过QueryableStateClient获取Flink应用中的可查询状态数据，QueryableStateClient在flink-queryable-state-client-java.jar包中，可通过添加代码清单5-13中的flink-core和flink-queryable-state-client-java_2.11的Maven依赖配置到项目中将相关库引入。

代码清单5-13　Flink QueryableStateClient客户端Maven配置

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-core</artifactId>
  <version>1.7.0</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-queryable-state-client-java_2.11</artifactId>
  <version>1.7.0</version>
</dependency>
```

如下代码所示，QueryableStateClient初始化参数分别为TaskManager的Hostname以及客户端代理监听端口。通常情况下，Querable State Client Proxy的默认端口是9067，也可以在./conf/flink-conf.yaml文件中配置监听端口。

```
QueryableStateClient client = new QueryableStateClient(tmHostname,
proxyPort);
```

获取到QueryableStateClient之后，就可以通过调用客户端中的getKvState()方法查询Flink流式应用中的状态值。其中getKvState()方法定义如下，参数中包括Flink Job任务的jobID，可查询状态名称queryableStateName以及需要查询的key，还有key对应的数据元素的keyTypeInfo，另外在stateDescriptor参数中包含了可查询状态的信息，如状态名称、类型以及序列化和反序列化等信息。

```
CompletableFuture<S> getKvState(
  JobID jobId,
  String queryableStateName,
  K key,
  TypeInformation<K> keyTypeInfo,
  StateDescriptor<S, V> stateDescriptor)
```

同时getKvState()方法返回值可以指定返回的状态数据类型，目前Flink支持查询ValueState、 ReduceState、ListState、MapState、AggregatingState几种类型的状态值。可以调用相应的get()方法获取状态中的指标，例如对于ValueState可以使用ValueState.get()获取结果。注意可查询状态结果是不可变的，不可以调用update()或者add()对状态进行修改，否则会报出UnsupportedOperationException异常。如代码清单5-14所示，使用QueryableStateClient查询代码清单5-6中定义的可查询状态值。

代码清单5-14　使用QueryableStateClient查询Flink应用中可查询状态值

```
  val tmHostname: String = "localhost"
  val proxyPort: Int = 9069
  val jobId: String = "d1227b1a350d952c372re4c886d2re243"
  val key: Integer = 5
  //创建QueryableStateClient
  val client: QueryableStateClient = new
QueryableStateClient(tmHostname, proxyPort)
  //创建需要查询的状态对应的ValueStateDescriptor
```

```
  val valueDescriptor: ValueStateDescriptor[Long] = new
ValueStateDescriptor[Long]("leastValue", TypeInformation.of(new
TypeHint[Long]() {}))
  //查询key为5的可查询状态值
  val resultFuture: CompletableFuture[ValueState[Long]] =
client.getKvState(
    JobID.fromHexString(jobId),
    "leastQueryValue",
    key,
    Types.INT,
    valueDescriptor)
}
//从resultFuture等待返回结果
resultFuture.thenAccept(response => {
    try
      val res = response.value()
      println(res)
    catch {
      case e: Exception =>
        e.printStackTrace()
    }
})
```

5.5 本章小结

本章重点介绍了Flink所支持的有状态计算相关的知识和概念。在5.1节介绍了状态计算的概念和应用范围。5.2节介绍了Flink中支持的状态类型，包括对Keyed State、Operator State以及每种类型的状态在流数据处理过程中的应用方式和区别。5.3节介绍了Flink容错中对状态数据的持久化操作Checkpoint，以及Checkpoints和Savepoints之间的关系和各自的应用场景。5.4节介绍了Flink中支持的StateBackend，如MemoryStateBackend、FsStateBackend、RocksDBStateBackend等，并比较了每种StateBackend所具有的特点以及各自的应用场景。在5.4节同时介绍了Flink新特性Querable State的使用，了解如何在外部业务系统中查询Flink流式应用中的状态数据。

第6章

DataSet API介绍与使用

目前Flink在批量计算领域的应用不是特别广泛，但并不代表Flink不擅长处理批量数据，从前面的章节中已经知道批量数据其实是流式数据的子集，可以通过一套引擎处理批量和流式数据，而Flink在未来也会重点投入更多的资源到批流融合中。本章将从多个方面介绍Flink在批计算领域的应用，包括Flink提出的针对批处理计算的DataSet API的介绍与使用，以及Flink对迭代计算的支持等。通过本章的学习读者可以了解如何使用DataSet API开发批计算应用。

6.1 DataSet API

和DataStream API一样，Flink提出DataSet API用于处理批量数据。Flink将接入数据转换成DataSet数据集，并行分布在集群的每个节点上，基于DataSet数据集完成各种转换操作（map，filter等），并通过DataSink操作将结果数据输出到外部系统中。

开发环境配置

在使用Flink DataSet API进行批量应用程序开发之前，需要在工程中引入Flink批量计算相关依赖库，可以在项目工程中的pom.xml文件中添加flink-java对应的Dependency配置，引入DataSet API所需要的依赖库，用户可以根据需要选择Java版本或者Scala版本，也可以将两个依赖库同时引入工程。

```
<dependency>
<groupId>org.apache.flink</groupId>
<artifactId>flink-java</artifactId>
<version>1.7.0</version>
</dependency>
// 引入Scala版本的批量计算依赖库
<dependency>
<groupId>org.apache.flink</groupId>
<artifactId>flink-scala_2.11</artifactId>
<version>1.7.0</version>
</dependency>
```

6.1.1 应用实例

通过代码清单6-1可以看出，DataSet API其实和DataStream API相似，都是需要创建ExecutionEnvironment环境，然后使用ExecutionEnvironment提供的方法读取外部数据，将外部数据转换成DataSet数据集，最后在创建好的数据集上应用DataSet API提供的Transformation操作，对数据进行转换，处理成最终的结果，并对结果进行输出。

代码清单6-1 Flink DataSet API wordcount实例

```
//如果使用Scala语言编写DataSet API程序，需要引入相应隐式的方法
import org.apache.flink.api.scala._
object WordCount {
  def main(args: Array[String]) {
    //创建ExecutionEnvironment
    val env = ExecutionEnvironment.getExecutionEnvironment
    //读取数据集
    val text = env.fromElements(
      "Who's there?", "Hello World")
    //对数据集进行转换操作,形成（Word,value）格式,并进行Group操作,统计词频
    val counts = text.flatMap { _.toLowerCase.split("\\W+") filter
{ _.nonEmpty } }
      .map { (_, 1) }
      .groupBy(0)
      .sum(1)
    //输出统计出来的结果
    counts.print()
  }
}
```

6.1.2 DataSources数据接入

DataSet API支持从多种数据源中将批量数据集读到Flink系统中，并转换成DataSet数据集。数据接入接口共有三种类型，分别是文件系统类型、Java Collection类型，以及通用类数据源。同时在DataSet API中可以自定义实现InputFormat/RichInputFormat接口，以接入不同数据格式类型的数据源，常见的数据格式有CsvInputFormat、TextInput-Format TextInputFormat等。

1.文件类数据

（1）readTextFile(path) / TextInputFormat

使用DataSet API中的readTextFile方法读取文本文件，并将文件内容转换成DataSet[String]类型数据集。

```
/读取本地文件
val textFiles: DataSet[String] =
env.readTextFile("file:///path/textfile")
//读取HDFS文件
val hdfsFiles =
env.readTextFile("hdfs://nnHost:nnPort/path/textfile")
```

（2）readTextFileWithValue(path) / TextValueInputFormat

读取文本文件内容，将文件内容转换成DataSet[StringValue]类型数据集。StringValue是一种可变的String类型，通过StringValue存储文本数据可以有效降低String对象创建数量，从而降低系统性能上的开销。

```
//读取本地文件,指定读取字符格式类型为UTF-8
val ds = env.readTextFileWithValue("file:///path/textfile", "UTF-
8")
```

（3）readCsvFile(path) / CsvInputFormat

读取指定分隔符切割的CSV文件，且可以直接转换成Tuple类型、Case Class对象或者POJOs类。在方法中可以指定行切割符、列切割符、字段等信息。

```
val csvInput = env.readCsvFile[(String, Double)](
  " hdfs://nnHost:nnPort/path/to/csvfile ",
  includedFields = Array(0, 3))
```

（4）readSequenceFile(Key, Value, path) / SequenceFileInputFormat

读取SequenceFileInputFormat类型的文件，在参数中指定Key Class和Value Class类型，返回结果为Tuple2[Key,Value]类型。

```
val tuples = env.readSequenceFile(classOf[IntWritable],
classOf[Text],
"hdfs://nnHost:nnPort/path/to/file")
```

2.集合类数据

（1）fromCollection(Seq)

从给定集合中创建DataSet数据集，集合类型可以是数组、List等，也可以从非空Iterable中创建，需要指定数据集的Class类型。

```
//从Seq中创建DataSet数据集
val dataSet: DataSet[String] = env.fromCollection(Seq("flink",
  "hadoop", "spark"))
//从Iterable中创建DataSet数据集
val dataSet: DataSet[String] =
env.fromCollection(Iterable("flink","hadoop", "spark"))
```

（2）fromElements(elements: _*)

从给定数据元素序列中创建DataSet数据集，且所有的数据对象类型必须一致。

```
  val dataSet: DataSet[String] = env.fromElements("flink",
"hadoop",
"spark")
```

（3）generateSequence(from, to)

指定from到to范围区间，然后在区间内部生成数字序列数据集。

```
val numbers: DataSet[Long] = env.generateSequence(1, 10000000)
```

3. 通用数据接口

DataSet API中提供了Inputformat通用的数据接口，以接入不同数据源和格式类型的数据。InputFormat接口主要分为两种类型：一种是基于文件类型，在DataSet API对应readFile()方法；另外一种是基于通用数据类型的接口，例如读取RDBMS或NoSQL数据库中等，在DataSet API中对应createInput()方法。

（1）readFile(inputFormat, path) / FileInputFormat

自定义文件类型输入源，将指定格式文件读取并转成DataSet数据集。

```
// 通过自定义PointInFormat,读取指定格式数据
env.readFile(new PointInFormat(),"file:///path/file")
```

（2）createInput(inputFormat) / InputFormat

自定义通用型数据源，将读取的数据转换为DataSet数据集。如以下实例使用Flink内置的JDBCInputFormat，创建读取mysql数据源的JDBC Input Format，完成从mysql中读取Person表，并转换成DataSet[Row]数据集。

```
//通过创建JDBCInputFormat读取JDBC数据源
val jdbcDataSet: DataSet[Row] =
env.createInput(
  JDBCInputFormat.buildJDBCInputFormat()
    .setDrivername("com.mysql.jdbc.Driver")
    .setDBUrl("jdbc:mysql://localhost:3306/test ")
    .setQuery("select id, name from person")
    .setRowTypeInfo(new RowTypeInfo(BasicTypeInfo.LONG_TYPE_INFO,
BasicTypeInfo.
STRING_TYPE_INFO))
    .finish()
)
```

4. 第三方文件系统

为简化用户和其他第三方文件系统之间的交互，Flink针对常见类型数据源提出通用的FileSystem抽象类，每种数据源分别继承和实现FileSystem类，将数据从各个系统中读取到Flink中。DataSet API中内置了HDFS数据源、Amazon S3，MapR file system，Alluxio等文件系统的连接器，用户可以参考官方文档说明进行使用。

6.1.3 DataSet转换操作

针对DataSet数据集上的转换，Flink提供了非常丰富的转换操作符，从而实现基于DataSet批量数据集的转换。转换操作实质是将DataSet转换成另外一个新的DataSet，然后将各个DataSet的转换连接成有向无环图，并基于Dag完成对批量数据的处理。

1.数据处理

（1）Map

完成对数据集Map端的转换，并行将每一条数据转换成新的一条数据，数据分区不发生变化。

```
val dataSet: DataSet[String] = env.fromElements("flink", "hadoop",
"spark")
val transformDS: DataSet[String] = dataSet.map(x => x.toUpperCase)
```

（2）FlatMap

将接入的每一条数据转换成多条数据输出，包括空值。例如以下实例将文件中每一行文本切割成字符集合。

```
val dataSet: DataSet[String] =
env.fromElements("flink,hadoop,spark")
val words = dataSet.flatMap { _.split(",") }
```

（3）MapPartition

功能和Map函数相似，只是MapPartition操作是在DataSet中基于分区对数据进行处理，函数调用中会按照分区将数据通过Iteator的形式传入，并返回任意数量的结果值。

```
val dataSet: DataSet[String] = env.fromElements("flink", "hadoop",
"spark")
dataSet.mapPartition { in => in map { (_, 1) } }
```

（4）Filter

根据条件对传入数据进行过滤，当条件为True后，数据元素才会传输到下游的DataSet数据集中。

```
val dataSet: DataSet[Long] = env.fromElements(222,12,34,323)
val resultDs = dataSet.filter(x => x > 100)
```

---

2. 聚合操作

（1）Reduce

通过两两合并，将数据集中的元素合并成一个元素，可以在整个数据集上使用，也可以和Group Data Set结合使用。

```
val dataSet: DataSet[Long] = env.fromElements(222,12,34,323)
val result = dataSet.reduce((x, y) => x + y)
```

（2）ReduceGroup

将一组元素合并成一个或者多个元素，可以在整个数据集上使用，也可以和Group Data Set结合使用。

```
val dataSet: DataSet[Long] = env.fromElements(222,12,34,323)
dataSet.reduceGroup { collector => collector.sum }
```

（3）Aggregate

通过Aggregate Function将一组元素值合并成单个值，可以在整个DataSet数据集上使用，也可以和Group Data Set结合使用。如下代码是，在DataSet数据集中根据第一个字段求和，根据第三个字段求最小值。

```
val dataSet: DataSet[(Int, String, Long)] = env.fromElements((12,
"Alice", 34),
(12, "Alice", 34), (12, "Alice", 34))
val result:DataSet[(Int, String, Long)]
= dataSet.aggregate(Aggregations.SUM,
0).aggregate(Aggregations.MIN, 2)
```

也可以使用Aggregation函数的缩写方法，sum()、min()、max()等。

```
val result2: DataSet[(Int, String, Long)] = dataSet.sum(0).min(2)
```

（4）Distinct

求取DataSet数据集中的不同记录，去除所有重复的记录。

```
val dataSet: DataSet[Long] = env.fromElements(222,12,12,323)
val distinct: DataSet[Long] = dataSet.distinct
```

3. 多表关联

（1）Join

根据指定的条件关联两个数据集，然后根据选择的字段形成一个数据集。关联的key可以通过key表达式、Key-selector函数、字段位置以及Case Class字段指定。

对于两个Tuple类型的数据集可以通过字段位置进行关联，左边数据集的字段通过where方法指定，右边数据集的字段通过equalTo()方法指定。

```
val dataSet1: DataSet[(Int, String)] = ...
val dataSet2: DataSet[(Double, Int)] = ...
val result = dataSet1.join(dataSet2).where(0).equalTo(1)
```

对于Case Class类型的数据集可以直接使用字段名称作为关联Key：

```
val dataSet1: DataSet[Person] =
env.fromElements(Person(1,"Peter"),Person(2,"Alice"))
val dataSet2: DataSet[(Double, Int)] = env.fromElements((12.3,1),
(22.3,3))
val result = dataSet1.join(dataSet2).where("id").equalTo(1)
```

可以在关联的过程中指定自定义Join Funciton，Funciton的入参为左边数据集中的数据元素和右边数据集的中的数据元素所组成的元祖，并返回一个经过计算处理后的数据，其中Left和right的Key相同。

```
val result = dataSet1.join(dataSet2).where("id").equalTo(1){
  (left,right) => (left.id,left.name,right._1 + 1)
}
```

FlatMap与Map方法的相似，Join Function中同时提供了FlatJoin Function用来关联两个数据集，FlatJoin函数返回可以是一个或者多个元素，也可以不返回任何结果。

```
val result = dataSet1.join(dataSet2).where("id").equalTo(1) {
  (left, right, collector: Collector[(String, Double)]) =>
    collector.collect(left.name, right._1 + 1)
```

```
    collector.collect("prefix_" + left.name, right._1 + 2)
 }
```

为了能够更好地引导Flink底层去正确地处理数据集，可以在DataSet数据集关联中，通过Size Hint标记数据集的大小，Flink可以根据用户给定的线索调整计算策略，例如可以使用joinWithTiny或joinWithHuge提示第二个数据集的大小。

```
val dataSet1: DataSet[Person] =
env.fromElements(Person(1,"Peter"),Person(2,"Alice"))
val dataSet2: DataSet[(Double, Int)] = env.fromElements((12.3,1),
(22.3,3))
//提示Flink第二个数据集是小数据集
val result = dataSet1. joinWithTiny
(dataSet2).where("id").equalTo(1)
//提示Flink第二个数据集是大数据集
val result =
dataSet1.joinWithHuge(dataSet2).where("id").equalTo(1)
```

除了能够使用joinWithTiny或joinWithHuge方法来提示关联数据集的大小之外，Flink还提供了Join算法提示，可以让Flink更加灵活且高效地执行Join操作。

```
//将第一个数据集广播出去,并转换成HashTable存储,该策略适用于第一个数据集非常小
的情况
ds1.join(ds2,JoinHint.BROADCAST_HASH_FIRST).where("id").equalTo(1)
//将第二个数据集广播出去,并转换成HashTable存储,该策略适用于第二个数据集非常小
的情况
ds1.join(ds2,JoinHint.BROADCAST_HASH_SECOND).where("id").equalTo(1
)
//和不设定Hint相同,将优化的工作交给系统处理
ds1.join(ds2,JoinHint.OPTIMIZER_CHOOSES).where("id").equalTo(1)
//将两个数据集重新分区,并将第一个数据集转换成HashTable存储,该策略适用于第一个
数据集比
第二个数据集小,但两个数据集相对都比较大的情况
ds1.join(ds2,JoinHint.REPARTITION_HASH_FIRST).where("id").equalTo(
1)
//将两个数据集重新分区,并将第二个数据集转换成HashTable存储,该策略适用于第二个
数据集比
```

```
第一个数据集小,但两个数据集相对都比较大的情况
ds1.join(ds2,JoinHint.REPARTITION_HASH_SECOND).where("id").equalTo
(1)
//将两个数据集重新分区,并将每个分区排序,该策略适用于两个数据集已经排好顺序的情
况
ds1.join(ds2,JoinHint.REPARTITION_SORT_MERGE).where("id").equalTo(
1)
```

（2）OuterJoin

OuterJoin对两个数据集进行外关联，包含left、right、full outer join三种关联方式，分别对应DataSet API中的leftOuterJoin、rightOuterJoin以及fullOuterJoin方法。

```
//左外关联两个数据集,按照相同的key进行关联,如果右边数据集中没有数据则会填充空
值
dataSet1.leftOuterJoin(dataSet2).where("id").equalTo(1)
//右外关联两个数据集,按照相同的key进行关联,如果左边数据集中没有数据则会填充空
值
dataSet1.rightOuterJoin(dataSet2).where("id").equalTo(1)
和JoinFunciton一样，OuterJoin也可以指定用户自定义的JoinFunciton。
dataSet1.leftOuterJoin(dataSet2).where("id").equalTo(1){
  (left, right) =>
    if(right == null) {(left.id,1)
    } else{ (left.id,right._1)
    }
 }
```

对于大数据集，Flink也在OuterJoin操作中提供相应的关联算法提示，可以针对左右数据集的分布情况选择合适的优化策略，以提升整体作业的处理效率。

```
//将第二个数据集广播出去,并转换成HashTable存储,该策略适用于第一个数据集非常小
的情况
ds1.leftOuterJoin(ds2,JoinHint.BROADCAST_HASH_SECOND).where("id").
equalTo(
1)
//将两个数据集重新分区,并将第二个数据集转换成HashTable存储,该策略适用于第一个
数据集比
```

```
第二个数据集小,但两个数据集相对都比较大的情况
ds1.leftOuterJoin(ds2,JoinHint.REPARTITION_HASH_SECOND).where("id"
).equalTo(1)
```

和Join操作不同，OuterJoin的操作只能适用于部分关联算法提示。其中leftOuterJoin仅支持OPTIMIZER_CHOOSES、BROADCAST_HASH_SECOND、REPARTITION_HASH_SECOND以及REPARTITION_SORT_MERGE四种策略。rightOuterJoin仅支持OPTIMIZER_CHOOSES、BROADCAST_HASH_FIRST、REPARTITION_HASH_FIRST以及REPARTITION_SORT_MERGE四种策略。fullOuterJoin仅支持OPTIMIZER_CHOOSES、REPARTITION_SORT_MERGE两种策略。

（3）Cogroup

将两个数据集根据相同的Key记录组合在一起，相同Key的记录会存放在一个Group中，如果指定key仅在一个数据集中有记录，则co-group Function会将这个Group与空的Group关联。

```
val dataset = dataSet1.coGroup(dataSet2).where("id").equalTo(1)
```

（4）Cross

将两个数据集合并成一个数据集，返回被连接的两个数据集所有数据行的笛卡儿积，返回的数据行数等于第一个数据集中符合查询条件的数据行数乘以第二个数据集中符合查询条件的数据行数。Cross操作可以通过应用Cross Funciton将关联的数据集合并成目标格式的数据集，如果不指定Cross Funciton则返回Tuple2类型的数据集。

```
val dataSet1: DataSet[(Int, String)] = env.fromElements((12,
"flink"), (22,
"spark"))
val dataSet2: DataSet[String] = env.fromElements("flink")
//不指定Cross Funciton,返回Tuple[T,V],其中T为左边数据集数据类型,V为右边数
据集
val crossDataSet: DataSet[((Int, String), String)] =
dataSet1.cross(dataSet2)
```

4. 集合操作

（1）Union

合并两个DataSet数据集，两个数据集的数据元素格式必须相同，多个数据集可以连续合并。

```
val dataSet1: DataSet[(Long, Int)] = ...
val dataSet2: DataSet[(Long, Int)] = ...
//合并两个数据集
val unioned = dataSet1.union(dataSet2)
```

（2）Rebalance

对数据集中的数据进行平均分布，使得每个分区上的数据量相同。

```
val dataSet: DataSet[String] = env.fromElements("flink","spark")
//将DataSet数据集进行重平衡，然后执行map操作
val result = dataSet.rebalance().map {_.toUpperCase}
```

（3）Hash-Partition

根据给定的Key进行Hash分区，key相同的数据会被放入同一个分区内。

```
val dataSet: DataSet[(String, Int)] = ...
//根据第一个字段进行数据重分区,然后再执行MapPartition操作处理每个分区的数据
val result = dataSet.partitionByHash(0).mapPartition { ... }
```

（4）Range-Partition

根据给定的Key进行Range分区，key相同的数据会被放入同一个分区内。

```
val dataSet: DataSet[(String, Int)] = ...
//根据第一个字段进行数据重分区,然后再执行MapPartition操作处理每个分区的数据
val result = dataSet.partitionByRange(0).mapPartition { ... }
```

（5）Sort Partition

在本地对DataSet数据集中的所有分区根据指定字段进行重排序，排序方式通过Order.ASCENDING以及Order.DESCENDING关键字指定。

```
val dataSet: DataSet[(String, Int)] = ...
//本地对根据第二个字段对分区数据进行逆序排序,
val result = dataSet.sortPartition(1, Order. DESCENDING)
          //根据第一个字段对分区进行升序排序
          .sortPartition(0, Order. ASCENDING)
          //然后在排序的分区上执行MapPartition转换操作
          .mapPartition { ... }
```

5. 排序操作

（1）First-n

返回数据集的n条随机结果，可以应用于常规类型数据集、Grouped类型数据集以及排序数据集上。

```
val dataSet: DataSet[(Int, String)] = ...
// 普通数据集上返回五条记录
val result1 = dataSet.first(5)
// 聚合数据集上返回五条记录
val result2 = dataSet.groupBy(0).first(5)
// Group排序数据集上返回五条记录
val result3 = dataSet.groupBy(0).sortGroup(1,
Order.ASCENDING).first(5)
```

（2）Minby/Maxby

从数据集中返回指定字段或组合对应最小或最大的记录，如果选择的字段具有多个相同值，则在集合中随机选择一条记录返回。

```
val dataSet: DataSet[(Int, Double, String)] = ...
// 返回数据集中第一个字段和第三个字段最小的记录,并产生新的数据集
val result1: DataSet[(Int, Double, String)] = dataSet.minBy(0, 2)
// 根据第一个字段对数据集进行聚合,并返回每个Group内第二个字段最小对应的记录
val result2: DataSet[(Int, Double, String)] =
dataSet.groupBy(1).minBy(1)
```

6.1.4 DataSinks数据输出

通过对批量数据的读取（DataSource）及转换（Transformation）操作，最终形成用户期望的结果数据集，然后需要将数据写入不同的外部介质中进行存储，进而完成整个批量数据处理过程。Flink中对应数据输出功能被称为DataSinks操作，和DataSource Operator操作类似，为了能够让用户更加灵活地使用外部数据，Flink抽象出通用的OutputFormat接口，批量数据输出全部实现于OutputFormat接口，例如文本文件（TextOutputFormat）、CSV文件格式（CSVOutputFormat）。Flink内置了常用数据存储介质对应的OutputFormat，如HadoopOutputFormat、JDBCOutputFormat等。另外用户也可以根据需要自定义实现OutputFormat接口，对接其他第三方系统。

Flink在DataSet API中的数据输出共分为三种类型。第一种是基于文件实现，对应DataSet的write()方法，实现将DataSet数据输出到文件系统中。第二种是基于通用存储介质实现，对应DataSet的output()方法，例如使用JDBCOutputFormat将数据输出到关系型数据库中。最后一种是客户端输出，直接将DataSet数据从不同的节点收集到Client，并在客户端中输出，例如DataSet的print()方法。

1.基于文件输出接口

在DataSet API中，基于文件的输出接口直接在DataSet中完成封装和定义，例如目前支持的writeAsText直接将DataSet数据输出到指定文件中。在使用write相关方法输出文件的过程中，用户也可以指定写入文件的模式，分为OVERWRITE模式和NOT_OVERWRITE模式，前者代表将对文件内容进行覆盖写入，后者代表输出的数据将追加到文件尾端。

（1）writeAsText/TextOutputFormat

将DataSet数据以TextOutputFormat文本格式写入文件系统，其中文件系统可以是本地文件系统，也可以是HDFS文件系统，根据用户指定路径的前缀进行识别，例如<file>前缀代表本地文件系统，<hdfs>前缀代表HDFS分布式文件系统。TextOutputFormat是

FileOutputFormat的子类，而FileOutputFormat则是OutputFormat的实现类，具体实例代码如代码清单6-2所示。

代码清单6-2　DataSet文本格式文件输出实例

```
val dataSet: DataSet[(String, Int, Double)] = ...
// 将DataSet数据输出到本地文件系统
dataSet.writeAsText("file:///my/result/on/localFS");
//将DataSet数据输出到HDFS文件系统
dataSet.writeAsText("hdfs://nnHost:nnPort/my/result/on/localFS");
```

（2）writeAsCsv(...)/CSVOutputFormat

该方法将数据集以CSV文件格式输出到指定的文件系统中，并且可以在输出方法中指定行切割符、列切割符等基于CSO文件配置。

```
val dataSet: DataSet[(String, Int, Double)] = ...
//将DataSet输出为CSV文件,指定行切割符为\n,列切割符为,
dataSet.writeAsCsv("file:///path/file", "\n", ",")
```

2. 通用输出接口

在DataSet API中，除了已经定义在DataSet中的输出方式，也可以使用自定义Output-Format方法来定义介质对应的OutputFormat，例如JDBCOutputFormat、HadoopOutput-Format等。

```
//读取数据集并转换为(word,count)类型数据
val dataSet: DataSet[(String, Long)] = ...
//将数据集的格式转换成[Text,LongWritable]类型
val words = dataSet.map( t => (new Text(t._1), new
LongWritable(t._2)) )
//定义HadoopOutputFormat
val hadoopOutputFormat = new HadoopOutputFormat[Text,
LongWritable](
  new TextOutputFormat[Text, LongWritable],
  new JobConf)
```

```
//指定输出路径
FileOutputFormat.setOutputPath(hadoopOutputFormat.getJobConf, new
Path(resultPath))
//调用Output方法将数据写入Hadoop文件系统
words.output(hadoopOutputFormat)
```

6.2 迭代计算

迭代计算在批量数据处理过程中的应用非常广泛，如常用的机器学习算法KMeans、逻辑回归，以及图计算等，都会用到迭代计算。DataSet API对迭代计算功能的支持相对比较完善，在性能上较其他分布式计算框架也具有非常高的优势。目前Flink中的迭代计算种类有两种模式，分别是Bulk Iteration（全量迭代计算）和Delt Iteration（增量迭代计算）。

6.2.1 全量迭代

全量迭代计算过程如图6-1所示，在数据接入迭代算子过程中，Step Function每次都会处理全量的数据，然后计算下一次迭代的输入，也就是图中的Next Partial Solution，最后根据触发条件输出迭代计算的结果，并将结果通过DataSet API传输到下一个算子中继续进行计算。Flink中迭代的数据和其他计算框架相比，并不是通过在迭代计算过程中不断生成新的数据集完成，而是基于同一份数据集上完成迭代计算操作，因此不需要对数据集进行大量拷贝复制操作，从而避免了数据在复制过程中所导致的性能下降问题。

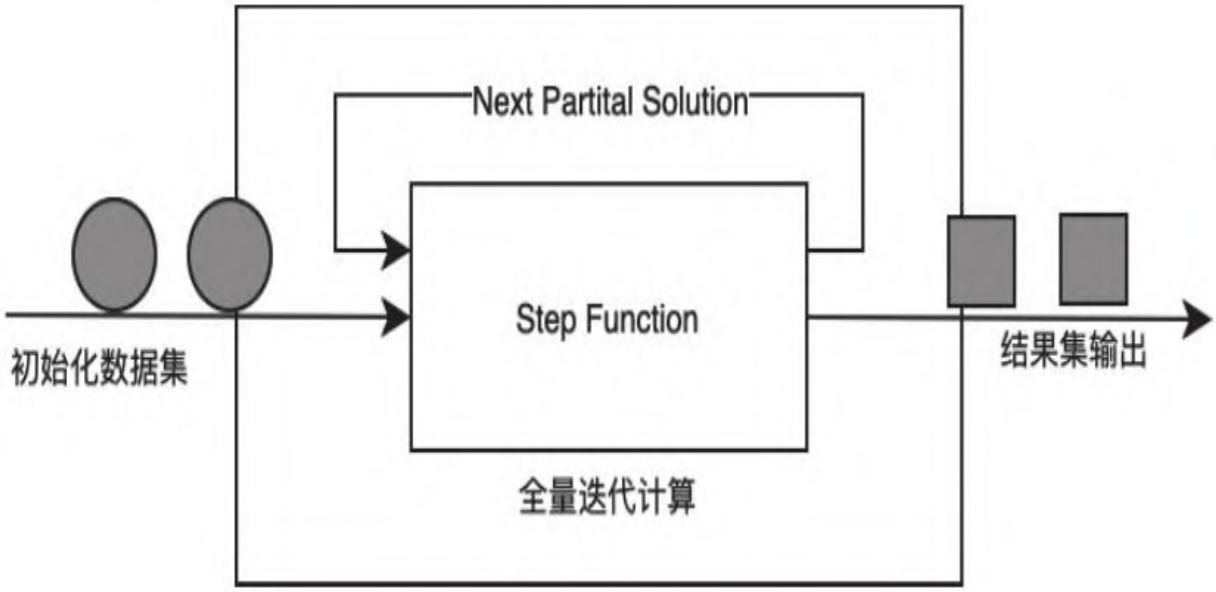

图6-1 全量迭代计算示意图

针对全量迭代计算，一共分为以下几个步骤：

· 首先初始化数据，可以通过从DataSource算子中读取，也可以从其他转换Operators中接入。

· 其次定义Step Function，并在每一步迭代过程使用Step Function，结合数据集以及上一次迭代计算的Solution数据集，进行本次迭代计算。

· 每一次迭代过程中Step Function输出的结果，被称为Next Partital Solution数据集，该结果会作为下一次迭代计算的输入数据集。

· 最后一次迭代计算的结果输出，可通过DataSink输出，或接入到下一个Operators中。迭代终止的条件有两种，分别为达到最大迭代次数或者符合自定义聚合器收敛条件：

· 最大迭代次数：指定迭代的最大次数，当计算次数超过该设定值时，终止迭代。

· 自定义收敛条件：用户自定义的聚合器和收敛条件，例如将终止条件设定为当Sum统计结果小于零则终止，否则继续迭代。

全量迭代计算通过使用DataSet的iterate()方法调用，具体实例如代码清单6-3所示：

代码清单6-3　全量迭代计算示例

```
val env = ExecutionEnvironment.getExecutionEnvironment
  // 创建初始化数据集
  val initial = env.fromElements(0)
  //调用迭代方法,并设定迭代次数为10000次
  val count = initial.iterate(10000) { iterationInput: DataSet[Int]
=>
    val result = iterationInput.map { i =>
      val x = Math.random()
      val y = Math.random()
      i + (if (x * x + y * y < 1) 1 else 0)
    }
    result
  }
  //输出迭代结果
  val result = count map { c => c / 10000.0 * 4 }
  result.print()
  env.execute("Iterative Pi Example")
```

6.2.2 增量迭代

如图6-2所示，增量迭代是通过部分计算取代全量计算，在计算过程中会将数据集分为热点数据和非热点数据集，每次迭代计算会针对热点数据展开，这种模式适合用于数据量比较大的计算场景，不需要对全部的输入数据集进行计算，所以在性能和速度上都会有很大的提升。

针对图6-2中的每一个步骤，解释如下：

· Iteration Input：初始化数据，可以是DataSource生成，也可以是计算算子生成；

· Step Function：在每一步迭代过程中使用的计算方法，可以是类似于map、reduce、join等方法；

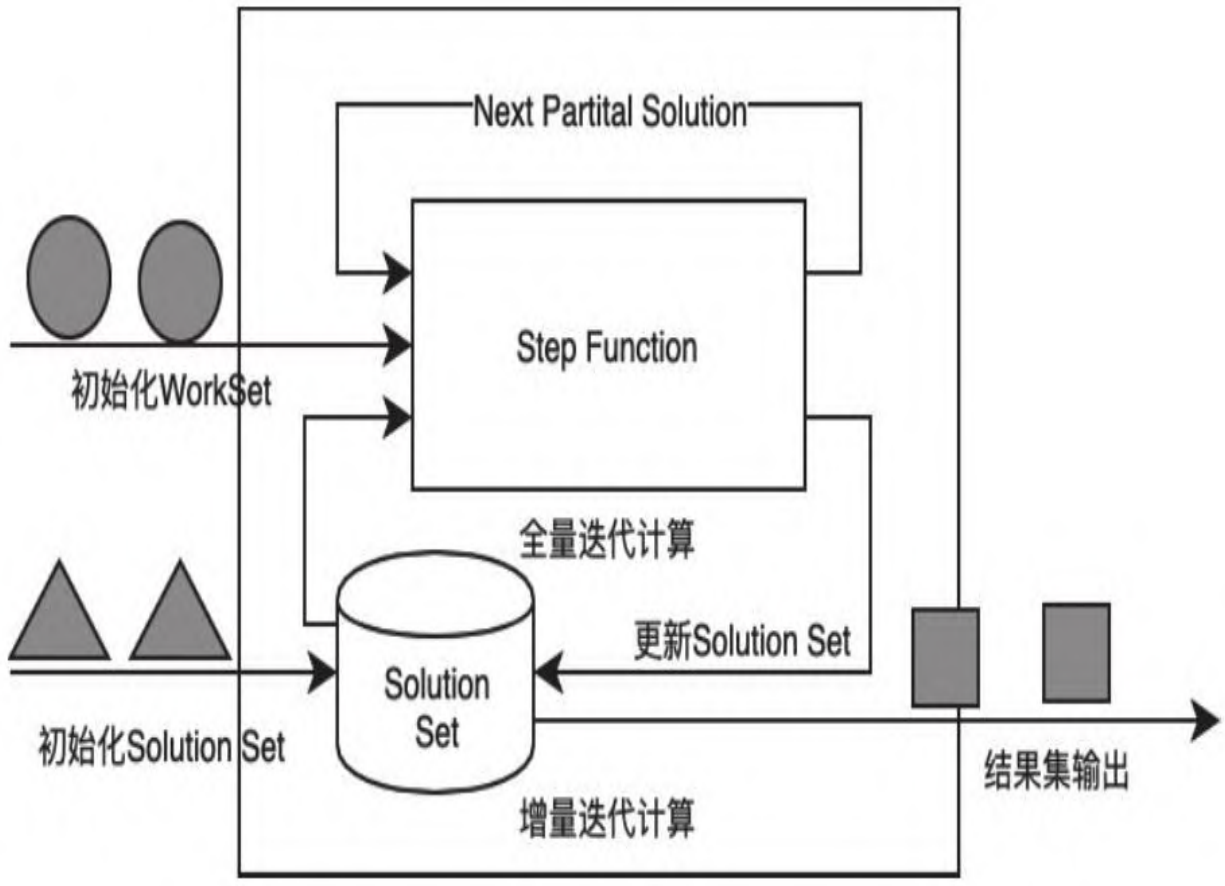

图6-2　增量迭代计算

· Next Partial Solution：在每一次迭代过程中，当前的Step Function输出的结果，该结果会作为下一次迭代计算的输入；

· Iteration Result：最后一次迭代计算的输出，可以通过指定DataSink输出，或者接入下一个Operators中。

增量迭代的终止条件可以指定为：

· WorkSet为空：如果下一次迭代输入WorkSet为空，则终止迭代。

· 最大迭代次数：当计算次数超过指定迭代的最大次数，则终止迭代。

· 增量迭代计算代码实例如代码清单6-4所示

代码清单6-4　增量迭代计算示例

```
// 读取初始化数据集
val initialSolutionSet: DataSet[(Long, Double)] = ...
//读取初始化WorkSet数据集
val initialWorkset: DataSet[(Long, Double)] = ...
//设定迭代参数
val maxIterations = 100
val keyPosition = 0
//通过iterateDelta应用增量迭代方法
val result = initialSolutionSet.iterateDelta(initialWorkset,
maxIterations, Array(keyPosition)) {
  (solution, workset) =>
    val candidateUpdates = workset.groupBy(1).reduceGroup(new
ComputeCandidateChanges())
    val deltas =
candidateUpdates.join(solution).where(0).equalTo(0)(new
CompareChangesToCurrent())
    val nextWorkset = deltas.filter(new FilterByThreshold())
    (deltas, nextWorkset)
}
//输出迭代计算的结果
result.writeAsCsv(outputPath)
env.execute()
```

6.3 广播变量与分布式缓存

6.3.1 广播变量

广播变量是分布式计算框架中经常会用到的一种数据共享方式，目的是对小数据集采用网络传输的方式，在每个并行的计算节点的实例内存中存储一份该数据集，所在的计算节点实例均可以在本地内存中直接读取被广播的数据集，这样能够避免在数据计算过程中多次通过远程的方式从其他节点中读取小数据集，从而提升整体任务的计算性能。

在DataSet API中，广播变量通过DataSet的withBroadcastSet(DataSet, String)方法定义，其中第一个参数为所需要广播的DataSet数据集，需要保证DataSet在广播之前已经创建完毕，第二个参数为广播变量的名称，需要在当前应用中保持唯一，如以下代码将broadcastData数据集广播在data数据集所在的每个实例中。

```
// 创建需要广播的数据集
val broadcastData = env.fromElements(1, 2, 3)
//广播DataSet数据集,指定广播变量名称为broadcastSetName
data.map(...).withBroadcastSet(broadcastData, "broadcastSetName")
```

DataSet API支持在RichFunction接口中通过RuntimeContext读取到广播变量。首先在RichFunction中实现Open()方法，然后调用getRuntimeContext()方法获取应用的RuntimeContext，接着调用getBroadcastVariable()方法通过广播名称获取广播变量。同时Flink

直接通过collect操作将数据集转换为本地Collection。需要注意的是，Collection对象的数据类型必须和定义的数据集的类型保持一致，否则会出现类型转换问题。

如以下代码实例所示，在dataSet2的Map转换中通过withBroadcastSet方法指定dataSet1为广播变量，然后通过实现RichMapFunction接口，在open()方法中调用RuntimeContext对象的getBroadcastVariable()方法，将dataSet1数据集获取到本地并转换成Collection。最后在map方法中访问dataSet1中的数据，完成后续的处理操作。

```
// 创建需要广播的数据集
val dataSet1:DataSet[Int] = ...
//创建输入数据集
val dataSet2:DataSet[String] = ...
dataSet2.map(new RichMapFunction[String, String]() {
  var broadcastSet: Traversable[Int] = null
  override def open(config: Configuration): Unit = {
    // 获取广播变量数据集,并且转换成Collection对象
    broadcastSet = getRuntimeContext().getBroadcastVariable[Int]
("broadcastSet-1").asScala
  }
  def map(input: String): String = {
      input + broadcastSet.toList //获取broadcastSet元素信息
  }
//广播DataSet数据集,指定广播变量名称为broadcastSetName
}).withBroadcastSet(dataSet1, "broadcastSet-1")
```

6.3.2 分布式缓存

在批计算中，需要处理的数据集大部分来自于文件，对于某些文件尽管是放在类似于HDFS之上的分布式文件系统中，但由于Flink并不像MapReduce一样让计算随着数据所在位置上进行，因此多数情况下会出现通过网络频繁地复制文件的情况。因此对于有些高频使用的文件可以通过分布式缓存的方式，将其放置在每台计算节点实例的本地task内存中，这样就能够避免因为读取某些文件而必须通过网络远程获取文件的情况，进而提升整个任务的执行效率。

分存式缓存在ExecutionEnvironment中直接注册文件或文件夹，Flink在启动任务的过程中将会把指定的文件同步到task所在计算节点的本地文件系统中，目前支持本地文件、HDFS、S3等文件系统，另外可以通过Boolean参数来指定文件是否可执行，具体使用方式如下：

```
val env = ExecutionEnvironment.getExecutionEnvironment
// 通过从HDFS文件读取并转换成分布式缓存
env.registerCachedFile("hdfs:///path/file", "hdfsFile")

// 通过从本地文件中读取并注册为分布式缓存,并将可执行设定为True
env.registerCachedFile("file:///path/file", "localFile", true)
```

获取缓存文件的方式和广播变量相似，也是实现RichFunction接口，并通过RichFunction接口获得RuntimeContext对象，然后通过RuntimeContext提供的接口获取对应的本地缓存文件，使用方式如以下代码所示：

```
// 定义RichMapFunction获取分布式缓存文件
class FileMapper extends RichMapFunction[String, Int] {
  var myFile: File = null
  override def open(config: Configuration): Unit = {
    //  通过RuntimeContext和DistributedCache获取缓存文件
    myFile  =
getRuntimeContext.getDistributedCache.getFile("hdfsFile")
  }
  override def map(value: String): Int = {
```

```
    // 使用读取到的文件内容
    val inputFile = new FileInputStream(myFile)
    ...  //定义数据处理逻辑
  }
}
```

通过RuntimeContext和DistributedCache获取缓存文件，且文件为java.io.File类型，然后将文件定义成静态对象中，就可以直接在map方法中读取文件中的内容，进行后续的算子操作，同时使用完缓存文件后Flink会自动将文件从本地文件系统中清除。

6.4 语义注解

在Flink批量数据处理过程中，往往传入函数的对象中可能含有很多字段，其中有些字段是Function计算会用到的，但有些字段在进入Funciton后并没有参与到实际计算过程中。针对这种情况，Flink提出了语义注解的功能，将这些字段在Function中通过注解的形式标记出来，区分出哪些是需要参与函数计算的字段，哪些是直接输出的字段。Flink Runtime在执行算子过程中，会对注解的字段进行判别，对于不需要函数处理的字段直接转发到Output对象中，以减少数据传输过程中所消耗的网络IO或者不必要的排序操作等，以提升整体应用的处理效率。

在DataSet API中将语义注解支持的字段分为三种类型，分别为Forwarded Fields、Non-Forward Fields以及Read Fields，下面详细介绍每种语义注解的使用方式。

6.4.1 Forwarded Fileds注解

转发字段（Forwarded Fileds）代表数据从Function进入后，对指定为Forwarded的Fileds不进行修改，且不参与函数的计算逻辑，而是根据设定的规则表达式，将Fields直接推送到Output对象中的相同位置或指定位置上。

转发字段的规则通过表达式进行指定，表达式中可以指定转发字段的源位置和目标位置。例如“f0->f2”，代表将Input的Tuple对象中的第一位字段转发到Output的Tuple对象中的第三位字段的位置上；单个字符“f2”，代表将Input的Tuple对象中的第三位字段转发到Output的Tuple对象中的相同位置，位置不发生变化；“f1->*”表达式代表将Input的Tuple对象中的第二位字段转发为Output整个字段，其他字段不再输出。多个表达式可以同时使用，表达式中间通过分号分隔，例如混合表达式“f1->f2;f3->f1;f0”。在使用表达式的定义字段转发规则的过程中，用户需要非常清楚哪些字段需要转发，保证所有的定义都是正确的，以免在字段转发过程中出现问题。注意，在Scala环境中一般是通过_1表示元祖中第一个参数，依次类推。

转发字段定义方式有两种，首先可以通过在函数类上添加Java注解的方式指定，其次也可以通过在Operator算子对应的Function后调用类似ForwardedFieldsFirst的方法来指定。

1.函数注解方式

ForwardedFields注解主要用于单输入的Function进行字段转发，例如Map、Reduce等。如下代码所示，定义实现MapFunction接口的MyMap Function Class，完成map方法的定义，最后在MyMap Class上添加ForwardedFields注解，其中f0->f2代表将输入元祖Tuple2中的第一个字段转发到输出元祖Tuple3中的第三个字段的位置上，该字段不参与函数计算，直接转发至输出对象中。

```
//通过函数注解方式配置转发字段,将输入数据集中的第一个字段转发到输出数据集的第二个字段中
@ForwardedFields("_1->_2")
class MyMapper extends MapFunction[(Int, Double), (Double, Int)]{
```

```
  def map(t: (Int, Double)): (Double, Int) = {
//map函数中也定义为将t._1输出到output对象的t_2字段中
    return (t._2 / 2, t._1)
  }
```

对于多输入函数，如Cogroup、Join等函数，可以使用@ForwardedFieldsFirst以及@ForwardedFieldsSecond注解分别对输入的数据集进行转发配置，而且@Forwarded-FieldsFirst和@ForwardedFieldsSecond也可以在函数定义的过程中同时使用。

2.算子参数方式

在单输入Operator算子中，可以调用withForwardFields完成函数的转发字段的定义。例如data.map(myMapFnc).withForwardedFields("f0->f2")，实现数据集data在myMapFnc函数调用中，Input Tuple中的第一个字段转发为Output Tuple中的第三个字段转发定义。针对多输入算子的转发字段定义，例如CoGroup、Join等算子，可以通过withForwardedFieldsSecond方法或withForwardedFieldsSecond方法分别对第一个和第二个输入数据集中的字段进行转发，两个方法也可以同时使用。以下实例就实现了对Join函数中第二个Input对象中第二个字段转发到输出对象中第三个字段逻辑的定义。

```
//创建数据集
val dataSet1: DataSet[Person] = ...
val dataSet2: DataSet[(Double, Int)] = ...
//指定Join函数,并且在算子尾部通过withForwardedFieldsSecond方法指定字段转发逻辑
val result = dataSet1.join(dataSet2).where("id").equalTo(1) {
  (left, right, collector: Collector[(String, Double, Int)]) =>
    collector.collect(left.name, right._1 + 1, right._2)
    collector.collect("prefix_" + left.name, right._1 + 2,
right._2)
}.withForwardedFieldsSecond("_2->_3")//定义转发逻辑
```

6.4.2 Non-Forwarded Fileds注解

和前面提到的Forwarded Fileds相反，Non-Forwarded Fileds用于指定不转发的字段，也就是说除了某些字段不转发在输出Tuple相应的位置上，其余字段全部放置在输出Tuple中相同的字段位置上，对于被Non-Forwarded Fileds指定的字段将必须参与到函数计算过程中，并产生新的结果进行输出。在使用Non-Forwarded Fields注解时，需要对应的Function具有相同类型的Input和Output对象。例如，表达式“f1;f3”代表输入函数的Input对象中，第二个和第四个字段不需要保留在Output对象中，其余字段全部按照原来位置进行输出，其中第二个和第四个字段需要在函数计算过程中产生，然后在输出结果中完成整个Output Tuple对象的整合。

和Forwarded Fileds一样，非转发字段的定义可以通过函数类注解的方式实现，对于不同的函数输入分别有不同的注解方式可以使用：对于单输入的算子，例如Map、Reduce等，可以通过NonForwardedFields注解进行的定义。对于多输入的算子，例如CoGroup、Join等，可以通过NonForwardedFieldsFirst和NonForwardedFieldsSecond分别对第一个对象和第二个对象中的字段进行转发逻辑定义。

```
// 不转发第二个,其余字段转发到输出对象相同位置上
@NonForwardedFields("_2")
class MyMapper extends MapFunction[(String, Long, Int), (String,
Long, Int)] {
  def map(input: (String, Long, Int)): (String, Long, Int) = {
    //第一个和第三个字段不参与函数计算,第二个字段参与到函数计算过程中,并产生新
的结果
    return (input._1, input._2 / 2, input._3)
  }
}
```

6.4.3 Read Fields注解

读取字段（Read Fields）注解用来指定Function中需要读取以及参与函数计算的字段，在注解中被指定的字段将全部参与当前函数结果的运算过程，如条件判断、数值计算等。和前面的字段注解类似，Flink针对读取字段也提供了相应的注解类定义，可以创建Function的Class上部通过使用注解定义转发规则。

对于单输入类型函数，使用@ReadFields完成注解定义，表达式可以是“f0;f2”，表示Input中Tuple的第一个字段和第三个字段参与函数的运算过程，其他字段因为不参与计算不需要读取至函数中，进而减少函数执行过程中数据传输的大小。如下代码实例所示，其中f0和f3参与了函数计算过程，f0参与了条件判断，f3字段参与了数值运算，指定在@ReadFields(“_1; _2”)函数注解指明，f1虽然在函数中引用过，但其并没有涉及运算，因此无须在注解中指明。

```
@ReadFields("_1; _2")
  class MyMapper extends MapFunction[(Int, Int, Double, Int),
(Int, Long)]{
    def map(value: (Int, Int, Double, Int)): (Int, Double) = {
      if (value._1 == 42) {
        return (value._1, value._3)
      } else {
        return (value._2 + 10, value._3)
      }
}}
```

针对多输入的函数，例如Join、CoGroup等函数，可以使用ReadFieldsFirst和ReadFieldsSecond注解来完成对第一个和第二个输入对象读取字段的定义，具体的定义方式和单数入函数定义类似，并且可以同时使用两个注解。

 注意

一旦使用了Read Fields注解，函数中所有参与计算的字段均必须在注解中指明，否则会导致计算函数执行失败等情况，另外如果字段并未参与函数的计算过程，也可以在注解中指定，这种方式不会对程序有太大影响，用户应该尽可能清楚函数中哪些字段参与了计算，哪些字段未参与函数计算过程。

6.5 本章小结

本章重点介绍了如何使用Flink DataSet接口对批量数据进行处理，其中包括对各种数据源的读取，从不同数据源中获取数据转换成DataSet数据集，并利用Flink提供的丰富的转换操作以完成对数据集的批量处理，最终将数据写入外部存储介质中。6.2节介绍了DataSet API提供的迭代运算函数，介绍了如何使用DataSet API进行全量和增量迭代计算，以及每种迭代计算的特点。6.3节介绍了在Flink中如何使用广播变量和分布式缓存进行数据的共享，让每台计算节点都能在本地获取数据或文件，进而提升分布式计算环境下数据处理的效率。6.4节介绍了Flink DataSet API中提供的语义注解特性，了解如何通过使用语义注解对函数进行优化，减少不必要的数据传输，以提升整体应用的计算性能。

第7章

Table API & SQL介绍与使用

对于像DataFrame这样的关系型编程接口，因其强大且灵活的表达能力，能够让用户通过非常丰富的接口对数据进行处理，有效降低了用户的使用成本，近年来逐渐成为主流大数据处理框架主要的接口形式之一。Flink也提供了关系型编程接口Table API以及基于Table API的SQL API，让用户能够通过使用结构化编程接口高效地构建Flink应用。同时Table API以及SQL能够统一处理批量和实时计算业务，无须切换修改任何应用代码就能够基于同一套API编写流式应用和批量应用，从而达到真正意义的批流统一。本章将重点介绍如何使用Flink Table & SQL API来构建流式应用和批量应用。

7.1 TableEnviroment概念

和DataStream API一样，Table API和SQL中具有相同的基本编程模型。首先需要构建对应的TableEnviroment创建关系型编程环境，才能够在程序中使用Table API和SQL来编写应用程序，另外Table API和SQL接口可以在应用中同时使用，Flink SQL基于Apache Calcite框架实现了SQL标准协议，是构建在Table API之上的更高级接口。

7.1.1 开发环境构建

在使用Table API和SQL开发Flink应用之前，通过添加Maven配置到项目中，在本地工程中引入相应的flink-table_2.11依赖库，库中包含了Table API和SQL接口。

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table_2.11</artifactId>
  <version>1.7.0</version>
</dependency>
```

然后需要分别引入开发批量应用和流式应用对应的库，对于批量应用，需要引入以下flink-scala_2.11依赖库：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-scala_2.11</artifactId>
  <version>1.7.0</version>
</dependency>
```

对于构建实时应用，需要引入以下flink-streaming-scala_2.11依赖库：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-scala_2.11</artifactId>
  <version>1.7.0</version>
</dependency>
```

 注意

由于Flink Table接口中引入了Apache Calcite第三方库，会阻止Java虚拟机对用户的Classloaders进行垃圾回收，因此不建议用户在构建Flink应用时将flink-table依赖库打包进fat-jar中，可以在集群环境中将{FLINK_HOME}/opt的对应的flink-table jar复制到{FLINK_HOME}/lib中来解决此类问题，其中FLINK_HOME为Flink安装路径。

7.1.2 TableEnvironment基本操作

使用Table API或SQL创建Flink应用程序，需要在环境中创建TableEnvironment对象，TableEnvironment中提供了注册内部表、执行Flink SQL语句、注册自定义函数等功能。根据应用类型的不同，TableEnvironment创建方式也有所不同，都是通过调用TableEnvironment. getTableEnvironment()方法创建TableEnvironment，但是方法中的参数分别是每种应用类型对应的执行环境。

对于流式应用创建StreamExecutionEnvironment，然后通过TableEnvironment. getTable-Environment()方法获取TableEnvironment对象。

```
val streamEnv =
StreamExecutionEnvironment.getExecutionEnvironment();
// 通过从StreamExecutionEnvironment创建StreamTableEnvironment
val tStreamEnv = TableEnvironment.getTableEnvironment(streamEnv);
```

对于批量应用创建ExecutionEnvironment，然后通过TableEnvironment. getTable-Environment()获取TableEnvironment对象：

```
val batchEnv = ExecutionEnvironment.getExecutionEnvironment();
// 通过从ExecutionEnvironment创建TableEnvironment
val tBatchEnv = TableEnvironment.getTableEnvironment(batchEnv);
```

1.内部CataLog注册

在获取TableEnvironment对象后，然后就可以使用TableEnvironment提供的方法来注册相应的数据源和数据表信息。所有对数据库和表的元数据信息存放在Flink CataLog内部目录结构中，

其存放了Flink内部所有与Table相关的元数据信息，包括表结构信息、数据源信息等。

（1）内部Table注册

通过TableEnvironment中的Register接口完成对数据表的注册，如代码清单7-1所示，registerTable方法中包括两个参数，tableEnv.registerTable（"projectedTable", projTable），第一个参数是注册在CataLog中的表名，第二个参数是对应的Table对象，其中Table可以通过StreamTableEnvironment提供接口生成，或者从DataStream或DataSet中转换而来。

代码清单7-1　通过TableEnvironment注册Table

```
// 获取TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env);
// 通过Select查询生成projTable Table
val projTable = tableEnv.scan("SensorsTable").select(...);
// 将projTable在Catalog中注册成内部表projectedTable
tableEnv.registerTable("projectedTable", projTable);
```

通过以上代码就完成了对Table的注册，在内部CataLog中生成相应的数据表信息，这样用户就可以使用SQL语句对表进行处理，完成各种数据转换操作。注册在CataLog中的Table类似关系统数据库中的视图结构，当注册的表被引用和查询时数据才会在对应的Table中生成。需要注意多个语句同时查询一张表时，表中的数据将会被执行多次，且每次查询出来的结果相互之间不共享。

（2）TableSource注册

在使用Table API时，可以将外部的数据源直接注册成Table数据结构。目前Flink中已经提供了大部分常用的数据源，例如本地、HDFS、S3等文件系统，文件格式类型能够支持例如Text、CSV、Parquet等文件格式，对于流式应用Table API也已经支持了大部分流式数据源，可供用户自行选择使用，例如Kafka等消息中间件。

如代码清单7-2所示，通过CSVTableSource举例说明如何在CataLog中完成Flink TablSource的注册。首先获取StreamingTableEnvironment对象，然后创建CSVTableSource对象，指定CSV文件路径，最后调用StreamingTableEnvironment中registerTableSource方法完成对TableSource的注册。

代码清单7-2　Internal CataLog中注册TableSources

```
// 创建StreamingTableEnvironment,批量应用开发环境相似
val tableEnv = TableEnvironment.getTableEnvironment(env);
// 创建CSV文件类型的TableSource
TableSource csvSource = new CsvTableSource("/path/to/file", ...);
// 将创建好的TableSource注册到tableEnv中
tableEnv.registerTableSource("CsvTable", csvSource);
```

CSV TableSource在TableEnviroment中完成注册后，就能够使用SQL对“CsvTable”这张表进行查询操作，由于创建的是StreamingTableEnvironment，因此CsvTable中的数据会被当成流式数据进行处理。批量数据处理类似，创建对应的BatchTable-Environment，然后完成数据源的注册，再执行SQL语句查询或直接使用TableAPI处理数据源中的数据即可。

（3）TableSink注册

数据处理完成后需要将结果写入外部存储中，在Table API中有对应的Sink模块，被称为TableSink。TableSink操作在TableEnvironment中注册需要输出的表，SQL查询处理之后产生的结果将插入TableSink对应的表中，最终达到数据输出到外部系统目的。和TableSource相同，TableSink也需要事先注册到TableEnvironment中，参数内数据表名以及对应的TableSink对象。如代码清单7-3所示，创建CSVSink并注册到TableEnvironment，首先创建StreamTableEnvironment对象，然后创建CSVTableSink对象并指定字段名称以及字段类型，最后通过TableEnvironment的registerTableSink方法将定义好的CSVTableSink对象注册到TableEnvironment中，接下来就可以使用Insert语句向CsvSinkTable中写入数据。

代码清单7-3　在Flink内部CataLog中注册TableISink

```
//创建TableEnvironment对象
val tableEnv = TableEnvironment.getTableEnvironment(env)
// 创建CsvTableSink,指定CSV文件地址和切割符
val csvSink: CsvTableSink = new CsvTableSink("/path/csvfile", ",")
// 定义fieldNames和fieldTypes
val fieldNames: Array[String] = Array("field1", "field2",
"field3")
val fieldTypes: Array[TypeInformation[_]] = Array(Types.INT,
Types.DOUBLE, Types.LONG)
// 将创建的csvSink注册到TableEnvironment中并指定名称为"CsvSinkTable"
tableEnv.registerTableSink("CsvSinkTable", fieldNames, fieldTypes,
csvSink)
```

2. 外部CataLog

除了能够使用Flink内部的CataLog作为所有Table数据的元数据存储介质之外，也可以使用外部CataLog，外部CataLog需要用户自定义实现，然后在TableEnvironment中完成注册使用。Table API和SQL可以将临时表注册在外部CataLog中。

在TableEnvironment中注册外部CataLog，一共包含两个步骤：第一步是实现Flink内部接口ExternalCatalog来定义外部CataLog，外部CataLog可以是基于内存，也可以是基于其他的存储介质；第二步是将用户定义好的CataLog通过TableEnvironment的registerExternalCatalog方法进行注册，注册完成后就能够在整个TableEnvironment中进行使用。Flink已经在内部实现了InMemoryExternalCatalog，方便用户能够进行测试和开发使用，具体定义如实例代码所示。

```
// 配置TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)
// 创建基于内存的外部catalog
val InmemCatalog: ExternalCatalog = new InMemoryExternalCatalog()
// 向TableEnvironment中注册创建好的InmemCatalog
tableEnv.registerExternalCatalog("InMemCatalog", InmemCatalog)
```

如果用户需要自己定义其他类型ExternalCatalog，可以参考InMemoryExternalCatalog的实现。

3.DataStream或DataSet与Table相互转换

前面已经知道Table API是构建在DataStream API和DataSet API之上的一层更高级的抽象，因此用户可以灵活地使用Table API将Table转换成DataStream或DataSet数据集，也可以将DataSteam或DataSet数据集转换成Table，这和Spark中的DataFrame和RDD的关系类似。

以下我们通过实例进行说明如何在Table和DataStream及DataSet之间完成转换。

（1）DataStream或DataSet转换为Table

目前有两种方式可以将DataStream或DataSet转换为Table。一种是通过注册Table的方式，将DataSet或DataStream数据集注册成Catalog中的表，然后可以直接使用Flink SQL操作注册好的Table，这种方式需要指定表名和包含字段名称，Flink会自动从DataStream或DataSet数据集中推断出Table的字段类型。另外一种是转换方式，将DataSet或DataStream数据集转换成Table结构，然后可以使用Table API操作创建好的Table。

· DataStream注册成Table

调用TableEnvironment中的registerDataStream或registerDataSet就可以分别将相应的数据集注册至TableEnvironment中，注意在注册Table的过程中表名必须唯一，否则会出现表名冲突。以下代码是将DataStream通过TableEnvironment的registerDataStream方法注册成Table，然后用户就能够使用Flink SQL对表中的数据进行查询和处理操作。

```
//配置流计算执行环境
val sEnv = StreamExecutionEnvironment.getExecutionEnvironment
// 配置流式TableEnvironment,与批量环境类似
val tStreamEnv = TableEnvironment.getTableEnvironment(env)
val stream: DataStream[(Long, String)] = sEnv.fromElements((192,
```

```
"foo"), (122, "fun"))
// 将DataStream注册成Table,指定表名为table1并使用默认字段名f0,f1
tStreamEnv.registerDataStream("table1", stream)
// 将DataStream注册成Table,指定表名为table2和字段名称为field1, field2
tStreamEnv.registerDataStream("table2", stream, 'field1, 'field2)
```

· DataStream转换成Table

可以使用fromDataStream方法将DataStream数据集转换成Table，字段名称需要指定，字段类型由Flink自动推断，转换完成后可以使用Table API操作创建好的Table。

```
//将DataStream通过fromDataStream转换成Table
val table1: Table = tStreamEnv.fromDataStream(stream)
//将DataStream通过fromDataStream转换成Table,并指定字段名称
val table2: Table = tStreamEnv.fromDataStream(stream, 'field1,
'field2)
```

· DataSet注册成Table

与Datastream注册成Table的方式类似，只需要调用TableEnvironment中的registerDataSet()方法即可。

```
//配置批计算执行环境 ExecutionEnvironment
val bEnv = ExecutionEnvironment.getExecutionEnvironment
// 配置批计算TableEnvironment,与流计算环境类似
val tBatchEnv = TableEnvironment.getTableEnvironment(bEnv)
//创建DataSet数据集
val dataSet: DataSet[(Long, String)] = bEnv.fromElements((192,
"foo"), (122, "fun"))
// 将DataSet注册成Table,指定表名为table1并使用默认字段名f0,f1
tBatchEnv.registerDataSet("table1", dataSet)
// 将DataSet注册成Table,指定表名为table2和字段名称为field1, field2
tBatchEnv.registerDataSet("table2", dataSet, 'field1, 'field2)
```

· DataSet转换成Table

可以调用fromDataSet方法将DataSet数据集转换成Table，然后使用Table API操作创建好的Table，同时在转换过程中指定属性名称，也可以不指定，默认使用f0、f1作为字段名称。

```
//将DataStream通过fromDataStream转换成Table, 默认使用f0,f1作为字段名称
val table1: Table = tBatchEnv.fromDataSet(dataSet)
//将DataStream通过fromDataStream转换成Table,并指定字段名称
Val table2: Table = tBatchEnv.fromDataSet(dataSet,'field1,
'field2)
```

（2）Table转换为DataStream或DataSet

在Flink应用程序中，也可以将Table转换为DataStream或DataSet数据集，也就是说在Table API或SQL环境中可以同时使用DataStream和DataSet API来处理数据。Table转换为DataStream或DataSet数据集，需要在转换过程指明目标数据集的字段类型，Flink目前支持从Table中转换为Row、POJO、Case Class、Tuple、Atomic Type等数据类型。Row对象类型数据实现最为简单，不需要用户做任何数据结构定义，例如可以直接转换成DataStream[Row]或DataSet[Row]，其他的数据类型需要用户根据实际数据类型进行指定。

· Table转换为DataStream

在流式计算中Table的数据是不断动态更新的，将Table转换成DataStream需要设定数据输出的模式。目前Flink Table API支持Append Model和Retract Model两种输出模式。Append Model采用追加方式仅将Insert更新变化的数据写入DataStream中。Retract Model是一种更高级模式，在这种模式下，数据将会通过一个Boolean类型字段标记当前是Insert操作更新还是Delete操作更新的数据，用户根据具体条件筛选出相应操作类型输出的数据，且在实际使用过程中Retract Model会比较常见。下面通过如下实例说明如何将Table转换成DataStream数据集。

代码清单7-4　Table转换成DataStream数据集

```
//配置流计算执行环境 StreamExecutionEnvironment
val sEnv = StreamExecutionEnvironment.getExecutionEnvironment
// 配置流式TableEnvironment,与批量环境类似
val tStreamEnv = TableEnvironment.getTableEnvironment(env)
val stream: DataStream[(Long, String)] = sEnv.fromElements((192,
"foo"), (122, "fun"))
val table: Table = tStreamEnv.fromDataStream(stream)
// 将table通过toAppendStream方法转换成Row格式的DataStream
val dsRow: DataStream[Row] = tStreamEnv.toAppendStream[Row](table)
// 将table通过toAppendStream方法转换成Tuple2[String, Int]格式的
DataStream
val dsTuple: DataStream[(Long, String)] =
tStreamEnv.toAppendStream[(Long, String)](table)
// 将table通过toRetractStream方法转换成Row格式的DataStream
// 返回结果类型为(Boolean, Row)
// 可以根据第一个字段是否为True判断是插入还是删除引起的更新数据
val retractStream: DataStream[(Boolean, Row)] =
        tStreamEnv.toRetractStream[Row](table)
```

在代码中可以看出，使用toAppendStream和toRetractStream方法将Table转换为DataStream[T]数据集，T可以是Flink自定义的数据格式类型Row，也可以是用户指定的数据格式类型。在使用toRetractStream方法时，返回的数据类型结果为DataStream[(Boolean,T)]，Boolean类型代表数据更新类型，True对应INSERT操作更新的数据，False对应DELETE操作更新的数据。

· Table转换为DataSet

如代码清单7-5所示，Table转换为DataSet数据集的方法和DataSteam类似，但需要注意的是，在批量模式下Table中的数据为静态数据集，Table转换为DataSet的过程中不会涉及动态查询，因此直接通过调用TableEnvironment中的toDataSet方法转换DataSet数据集即可。需要指明目标转换数据集的数据类型，默认可以使用Row类型，也可以自定义数据类型，注意须和Table的数据结构类型一致，否则会出现转换异常。

代码清单7-5　Table转换成DataSet数据集

```
//配置批计算执行环境 ExecutionEnvironment
  val bEnv = ExecutionEnvironment.getExecutionEnvironment
  // 配置批计算TableEnvironment,流计算环境类似
  val tBatchEnv = TableEnvironment.getTableEnvironment(bEnv)
  //从DataSet中创建Table
  val table: Table = tBatchEnv.fromDataSet(dataSet)
    // 将Table转换成Row数据类型DataSet数据集
  val rowDS: DataSet[Row] = tBatchEnv.toDataSet[Row](table)
    //将Table转换成Tuple2(Long,String)类型数据类型DataSet数据集
  val tupleDS: DataSet[(Long, String)] =
tBatchEnv.toDataSet[(Long, String)](table)
```

通过以上代码实例可以看出，Table转换为DataSet相对比较简单，其中对于数据类型的定义，使用到了Row和Tuple2类型，用户也可以选择使用其他数据类型，例如POJOs类等。

（3）Schema字段映射

前面已经了解到Table可以由DataStream或者DataSet数据集转换而来，但是有一点需要注意，Table中的Schema和DataStream或DataSet的字段有时候并不是完全匹配的，通常情况下需要在创建Table的时候修改字段的映射关系。Flink Table Schema可以通过基于字段偏移位置和字段名称两种方式与DataStream或DataSet中的字段进行映射。

· 字段位置映射

字段位置映射（Position-Based）是根据数据集中字段位置偏移来确认Table中的字段，当使用字段位置映射的时候，需要注意数据集中的字段名称不能包含在Schema中，否则Flink会认为映射的字段是在原有的字段之中，将会直接使用原来的字段作为Table中的字段属性。如代码清单7-6所示，如果在映射中没有指定名称，Flink会默认使用索引位置作为Table字段名称，如“_1”，“_2”。

代码清单7-6 Position-Based Mapping

```
// 获取TableEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)
```

```
//创建DataStream数据集
val stream: DataStream[(Long, String)] = ...
// 将DataStream转换成Table,没有指定字段名称则使用默认值"_1","_2"
val table: Table = tStreamEnv.fromDataStream(stream)
//将DataStream转换成Table,并且使用field1,field2作为Table字段名称
val table: Table = tStreamEnv.fromDataStream(stream,
'field1,'field2)
```

· 字段名称映射

字段名称映射（Name-based）是指在DataStream或DataSet数据集中，使用数据中的字段名称进行映射。与使用偏移位置相比，字段名称映射将更加灵活，适用于包括自定义POJOs类的所有数据类型，另外也可以直接使用字段名称构建Table的Schema，或对字段进行重命名，也可以对字段进行重排序和投影输出。如代码清单7-7所示，通过字段名称映射，将Tuple2类型DataStream数据集中的字段映射到Table中的Schema信息中，Tuple2中默认的字段名称是_1和_2。

代码清单7-7　使用名称映射将Tuple类型DataStream数据集转换成Table

```
// 获取TableEnvironment
val tStreamEnv = TableEnvironment.getTableEnvironment(env)
val stream: DataStream[(Long, Int)] = ...
// 将DataStream转换成Table,并且使用默认的字段名称"_1","_2"
val table: Table = tStreamEnv.fromDataStream(stream)
// 将DataStream转换成Table,并且仅获取字段名称"_2"的字段
val table: Table = tStreamEnv.fromDataStream(stream, '_2)
// 将DataStream转换成Table,并且交换两个字段的位置
val table: Table = tStreamEnv.fromDataStream(stream, '_2, '_1)
// 将DataStream转换成Table,并且交换两个字段的位置,分别对两个字段进行重命名
val table: Table = tStreamEnv.fromDataStream(stream, '_2 as
'field1, '_1 as 'field2)
```

POJOs类数据集可以同时使用字段位置映射和名称映射两种映射方式。代码清单7-8是使用名称映射的实例。

代码清单7-8　使用名称映射将POJOs类型DataStream数据集转换成Table

```
// 定义Event Case Class
case class Event(id: String, rowtime: Long, variable: Int)
val stream: DataStream[Event] = ...
// 将DataStream转换成Table,并且使用默认字段名称id,time,variable
val table = tStreamEnv.fromDataStream(stream)
// 将DataStream转换成Table,并且基于位置重新指定字段名称为"field1",
"field2", "field3"
val table = tStreamEnv.fromDataStream(stream, 'field1, 'field2,
'field3)
// 将DataStream转换成Table,并且将字段名称重新成别名
val table: Table = tStreamEnv.fromDataStream(stream, 'rowtime as
'newTime, 'id as 'newId,'variable as 'newVariable)
```

7.1.3 外部连接器

在Table API和SQL中，Flink可以通过Table connector直接连接外部系统，将批量或者流式数据从外部系统中获取到Flink系统中，或者从Flink系统中将数据发送到外部系统中。其中对于数据接入，Table API中已经提供了TableSource从外部系统获取数据，例如常见的数据库、文件系统等外部系统，对应的有OrcTableSource、HBaseTableSource、CSVTableSource等常用的TableSource。对于数据输出，Table API提供了TableSink将Flink Table数据写入外部系统中，同时支持定义不同的文件格式类型，例如CsvTableSink、JDBCAppendTableSink和CassandraAppendTableSink等。

TableSource和TableSink的定义基本已经能够满足Table API和SQL对外部数据源输入和输出的需求，但是还不够灵活和通用，用户无法在编写应用程序的时候通过配置化的方式直接使用已经定义好的数据源。在Flink 1.6版本之后，为了能够让Table API通过配置化的方式连接外部系统，且同时可以在SQL Client中使用，Flink提出了Table Connector的概念，主要目的是将TableSource和TableSink的定义和使用分离。通过Table Connector将不同内建的TableSource和TableSink封装，形成可配置化的组件，在Table API和SQL Client能够同时使用。

如以下在Table API中通过Table Connector注册TableSource所涵盖的步骤，其中connect方法指定了需要连接Table Connector对应的Descriptor，withFormat方法指定了输出或者输入的文件格式，例如JSON或Parquet等，withSchema方法指定了注册在TableEnvironment中的表结构，inAppendMode指定了数据更新模式，最后通过registerTable-Source方法将本次连接的数据源对应的TableSource注册在TableEnvironment中。

```
tableEnvironment
  .connect(...) //指定Table Connector Descriptor
  .withFormat(...) //指定数据格式
  .withSchema(...) //指定表结构
  .inAppendMode() //指定更新模式
  .registerTableSource("MyTable") //注册TableSource
```

接下来分别介绍上述每个步骤中需要定义的对象参数。

1.Table Connector

Table API和SQL中使用org.apache.flink.table.descriptors.Descriptor接口实现类来创建Table Connector实例。在Flink中已经内置的Table Connector有File System Connector、Kafka Connector以及Elasticsearch Connector等。

（1）File System Connector

File System Connector允许用户从本地或者分布式文件系统中读取和写入数据，在Table API中文件系统的Table Connector可以通过FileSystem类来创建，只需指定相应的参数即可，如以下实例是创建本地文件系统连接器。

```
tableEnvironment
.connect(
  new FileSystem()
    .path("file:///path/filename ")     // 可以是文件夹或者文件
)
```

目前File System Connector通过流式的方式对文件内容的读取和写入还处于实验阶段，因此不建议用户在流式场景中使用。

（2）Kafka Connector

Kafka Connector支持从Apache Kafka的Topic中消费和写入数据，如以下实例所示，Kafka Connector可以配置Kafka的版本信息、Topic以及连接Kafka所需要的Properties配置信息，如果是从Kafka消费数据，则可以指定Offset的启动模式，如果是将数据写入Kafka中，则可以指定Flink和Kafka的数据分区策略。

```
.connect(
new Kafka()
  .version("0.11")    // 指定Kafka的版本,支持"0.8", "0.9", "0.10",
"0.11"
  .topic("mytopic")       // 指定Table对应的Kafka Topic
  // 通过property指定Kafka Connector需要的配置信息
  .property("zookeeper.connect", "localhost:2181")
  .property("bootstrap.servers", "localhost:9092")
  .property("group.id", "KafkaGroup")
  // 从Kakfa中读取数据：指定Offset的启动模式（可选）
  .startFromEarliest()//从最早的Offset开始消费
  .startFromLatest()  //从最新的Offset开始消费
  .startFromSpecificOffsets(...)//从指定的Offset开始消费
// 向Kafka中写入数据：指定Flink和Kafka的数据分区策略
  .sinkPartitionerFixed() //每个Flink分区最多被分配到一个Kafka分区上
  .sinkPartitionerRoundRobin() // Flink中分区随机映射到Kafka分区上
  .sinkPartitionerCustom(CustomPartitioner.class) //自定义
KafkaPartitioner
}
```

对于端到端的一致性保障，默认情况下，Kafka Table Connector支持到at-least-once级别，同时Flink也提供了exactly-once级别的一致性保障，前提需要集群打开Checkpointing的功能。

2.Table Format

Flink中提供了常用的Table Format可以在Table Connector中使用，以支持Connector传输不同格式类型不同格式的数据，例如常见的CSV Format、JSON Format以及Apache Avro Format等，Table Format通过使用TableEnvironment的withFormat方法来指定，以下常见的几种Table Format可以直接使用。

（1）CSV Format

CSV Format指定分隔符切分数据记录中的字段，如下代码所示，可以使用field字段来指定字段名称和类型，使用fieldDelimiter方法来指定列切割符，使用lineDelimiter方法来指定行切割符。

```
.withFormat(
  new Csv()
```

```
    .field("field1", Types.STRING)                  // 根据顺序指定字段名
称和类型（必选）
    .field("field2", Types.TIMESTAMP)    // 根据顺序指定字段名称和类型
（必选）
    .fieldDelimiter(",")                            // 指定列切割符,默认
使用","（可选）
    .lineDelimiter("\n")                            // 指定行切割符,默认
使用"\n"（可选）
    .quoteCharacter('"')                            // 指定字符串中的单个
字符,默认为空（可选）
    .commentPrefix('#')                  // 指定Comment的前缀,默认为空
（可选）
    .ignoreFirstLine()                   // 是否忽略第一行（可选）
    .ignoreParseErrors()                            // 是否忽略解析错误的
数据,默认开启（可选）
)
```

（2）JSON Format

JSON Format支持将读取或写入的数据映射成JSON格式，JSON是一种轻量的数据表示方法。在Table API中JSON Format具有三种定义方式，可以通过Flink数据类型定义，也可以直接使用JSON Schema定义，或者将Flink Table Schema转换成JSON Schema的方式来定义。其中，JSON Schema能够定义非常复杂和嵌套的数据结构，而Flink内部数据类型定义比较适合用于简单的Mapping关系，Flink会根据映射关系将Table中的数据类型转换成JSON格式，如Flink中ROW类型对应于JSON中的object结构，String类型对应于JSON中的VARCHAR结构等。如果Flink Table Schema信息和JSON的Schema一致，则可以直接使用deriveSchema从Table中抽取JSON Schema信息，使用这种方式时，用户只需要定义一次Table Schema，字段的名称、类型、位置都是由Table Schema确定的。如下代码所示，通过三种方式来定义JSOn Format。

```
.withFormat(
  new Json()
    .failOnMissingField(true)    // 当字段缺失的时候是否解析失败（可选）
    // 【方式一】使用Flink数据类型定义,然后通过Mapping映射成JSON Schema
    .schema(Type.ROW(...))
    // 【方式二】通过配置jsonSchema构建JSON FORMAT
```

```
    .jsonSchema(
      "{" +
      "  type: 'object'," +
      "  properties: {" +
      "    id: {" +
      "      type: 'number'" +  //定义字段类型
      "    }," +
           name: {" +
      "      type: 'string'" +  //定义字段类型
      "    }," +
      "    rowtime: {" +
      "      type: 'string'," +  //定义字段类型
      "      format: 'date-time'" + //指定时间格式
      "    }" +
      "  }" +
      "}"
    )
    // 【方式三】直接使用Table中的Schema信息,转换成JSON结构
    .deriveSchema()
)
```

目前Flink仅支持例如object、array、number等常用JSON schema的子集数据类型，不支持类似于allof、anyOf、not等复杂JSON数据结构。

（3）Apache Avro Format

Apache Avro Format可以支持读取和写入Avro格式数据，和JSON Format一样，Avro Format数据也具有丰富的数据结构类型，以及快速可压缩的二进制数据形式等特性。Avro Format的结构可以通过定义Avro的SpecificRecord Class来实现，或者通过指定avroSchema的方式来定义。

```
.withFormat(
  new Avro()
    //通过Avro SpecificRecord Class定义
    .recordClass(MyRecord.class)

    //通过avroSchema字符串定义
    .avroSchema(
      "{" +
      "  \"type\": \"record\"," +
```

```
      "  \"name\": \"event\"," +
      "  \"fields\" : [" +
      "    {\"name\": \"id\", \"type\": \"long\"}," +
      "    {\"name\": \"name\", \"type\": \"string\"}" +
      "  ]" +
      "}"
    )
)
```

3. Table Schema

Table Schema定义了Flink Table的数据表结构，包括字段名称、字段类型等信息，同时Table Schema会和Table Format相匹配，在Table数据输入或者输出的过程中完成Schema的转换。但是当Table Input/Output Format和Table Schema不一致的时候，都需要相应的Mapping关系来完成映射。

如以下代码所示，Table Schema可以根据字段顺序生成与数据源中对应的字段映射，需要注意，用户必须要按照Input/Ouput数据源中的字段顺序来定义Table Schema。

```
.withSchema(
  new Schema()
    .field("id", Types.INT)      // 指定第一个字段的名称和类型
    .field("name", Types.STRING) // 指定第二个字段的名称和类型
    .field("value", Types.BOOLEAN) //指定第三个字段的名称和类型
)
```

除了在创建Table Schema时指定名称和类型之外，也支持通过使用proctime和rowtime等方法获取外部数据中的时间属性，其中proctime方法不需要传入参数，rowtime方法需要定义时间字段以及Watermark生成逻辑。同时也可以通过使用from方法从数据集中根据名称映射Table Schema字段信息。

```
.withSchema(
  new Schema()
    .field("Field1", Types.SQL_TIMESTAMP)
```

```
      .proctime()       // 获取Process Time属性
    .field("Field2", Types.SQL_TIMESTAMP)
      .rowtime(...)     // 获取Event Time属性
    .field("Field3", Types.BOOLEAN)
      .from("origin_field_name")      // 从Input/Output数据指定字段中
获取数据
)
```

如果Table API基于Event Time时间概念处理数据，则需要在接入数据中生成事件时间Rowtime信息，以及Watermark的生成逻辑。

```
.rowtime(
  //可以根据字段名称从输入数据中提取
  new Rowtime().timestampsFromField("ts_field")
  //或者从底层DataStream API中转换而来,数据源需要支持分配时间戳（如
Kafka0.10+）
  new Rowtime().timestampsFromSource()
  //或者通过自定义实现timestampsFromExtractor抽取Rowtime
  new Rowtime().timestampsFromExtractor(...)
```

紧接在Rowtime()对象实例后需要指定Watermark策略。

```
.rowtime(
  //延时两秒生成Watermark
  new Rowtime().watermarksPeriodicBounded(2000)
  //和rowtime最大时间保持一致
  new Rowtime().watermarksPeriodicAscending()
  //使用底层DataStream API内建的Watermark
  new Rowtime().watermarksFromSource()
)
```

---

4. Update Modes

对于Stream类型的Table数据，需要标记出是由于INSERT、UPDATE、DELETE中的哪种操作更新的数据，在Table API中通过Update Modes指定数据更新的类型，通过指定不同的Update Modes模式来确定是哪种更新操作的数据来与外部系统进行交互。

```
.connect(...)
  .inAppendMode()    //仅交互INSERT操作更新数据
  .inUpsertMode()    //仅交互INSERT、UNPDATE、DELETE操作更新数据
  .inRetractMode()   //仅交互INSERT和DELETE操作更新数据
```

5. 应用实例

通过实例将Table Connector所有模块组装在一起，如代码清单7-9所示，首先创建Kafka的Table Connector，然后调用withFormat方法指定JSON Format，并使用jsonSchema来定义JSON的Schema信息，紧接着调用withSchema来指定Table的Schema信息，Table Schema和jsonSchema的结构基本上保持一致。通过rowtime方法从Json数据中提取rowtime和watermark信息，最后调用registerTableSource将创建好的Table Source注册到Table Environment中，最终整个Table Connector就完成了定义，后面Table API和SQL语句使用。

代码清单7-9 Kafka Table Connector应用实例

```
// 获取TableEnvironment
val tStreamEnv = TableEnvironment.getTableEnvironment(env)
tableEnvironment
  // 指定需要连接的外部系统,以下指定Kafka Connector
  .connect(
    new Kafka()
      .version("0.10")
      .topic("my-topic")
      .startFromEarliest()
      .property("zookeeper.connect", "localhost:2181")
      .property("bootstrap.servers", "localhost:9092")
  )
  // 指定Table Format信息
  .withFormat(
    new Json()
     .failOnMissingField(true)
     .jsonSchema(
      "{" +
      "  type: 'object'," +
      "  properties: {" +
      "    id: {" +
      "      type: 'number'" +
```

```
      "    },"  +
           name: {" +
      "      type: 'string'" +
      "    }," +
      "    timestamp: {" +
      "      type: 'string'," +  //定义字段类型
      "      format: 'date-time'" + //指定时间格式
      "    }" +
      "  }" +
      "}"
    )
  // 指定Table Schema信息
  .withSchema(
    new Schema()
      .field("id", Types.INT)
      .field("name", Types.STRING)
      .field("rowtime", Types.SQL_TIMESTAMP)
        .rowtime(new Rowtime()
          .timestampsFromField("timestamp")
          .watermarksPeriodicBounded(60000)
        )
  )
  // 指定数据更新模式为AppendMode
  .inAppendMode()
  // 注册TableSource,指定Table名称为KafkaInputTable
  .registerTableSource("KafkaInputTable");
```

7.1.4 时间概念

对于在Table API和SQL接口中的算子，其中部分需要依赖于时间属性，例如GroupBy Windows类算子等，因此对于这类算子需要在Table Schema中指定时间属性。我们已经Flink支持ProcessTime、EventTime和IngestionTime三种时间概念，针对每种时间概念，Flink Table API中使用Schema中单独的字段来表示时间属性，当时间字段被指定后，就可以在基于时间的操作算子中使用相应的时间属性。

1.Event Time指定

和DataStream API中的一样，Table API中的Event Time也是从输入事件中提取而来的，在Table API中EventTime支持两种提取方式，可以在DataStream转换成Table的过程中指定，也可以在定义TableSource函数中指定。

（1）在DataStream转换Table的过程中定义

在Table API中通过使用.rowtime来定义EventTime字段，例如在数据集中，事件时间属性为event_time，此时Table中的EventTime字段中可以通过定义为'event_time.rowtime来指定。目前Flink支持两种方式定义EventTime字段，分别是通过在Table Schema中自动从DataStreamEventTime字段或将EventTime字段提前在DataStream中放在某一字段中，然后通过指定相应位置来定义EventTime字段。两种方式在使用方式上有一定的区别，由以下代码实例可以看出。

```
//获取输入数据集
inputStream: DataStream[String,String] = ...
//调用DataStream API的assignTimestampsAndWatermarks指定EventTime和
Watermark信息
val stream: DataStream[(String, String)] =
          inputStream.assignTimestampsAndWatermarks(...)
// 在Table Schema末尾使用'event_time.rowtime定义EventTime字段
//系统会从TableEnvironment中获取EventTime信息
val table = tEnv.fromDataStream(watermarkStream, 'id, 'var1,
      'event_time.rowtime)
// 调用DataStream API的assignTimestampsAndWatermarks指定EventTime和
// Watermark信息,并在DataStream中将第一个字段提取出来并指定为EventTime字
```

```
段
val watermarkStream: DataStream[(Long, String, String)] =
          inputStream.assignTimestampsAndWatermarks(...)
// 当在第一个字段上定义'event_time.rowtime时,系统使用DataStream中对应字段
作为EventTime字段
val table = tEnv.fromDataStream(stream, 'event_time.rowtime, 'id,
'var1)
```

当EventTime字段在Table API中定义完毕之后，就可以在基于事件时间的操作算子中使用，例如在窗口中使用方式如下：

```
val windowTable = table.window(Tumble over 10.minutes on
'event_time as 'window)
```

（2）通过TableSource函数定义

另外也可以在创建TableSource的时候，实现DefinedRowtimeAttributes接口来定义EventTime字段，在接口中需要实现getRowtimeAttributeDescriptors方法，创建基于EventTime的时间属性信息。

```
// 定义InputEventSource创建外部数据源
// 并实现DefinedRowtimeAttributes接口以定义EventTime时间属性
  class InputEventSource extends StreamTableSource[Row] with
DefinedRowtimeAttributes {
    //定义数据集字段名称和类型
    override def getReturnType = {
      val names = Array[String]("id", "value", "event_time")
      val types = Array[TypeInformation[_]](Types.STRING,
Types.STRING, Types.LONG)
      Types.ROW(names, types)
    }
      //实现StreamTableSource接口中的getDataStream()方法,定义输入数据源
    override def getDataStream(execEnv:
StreamExecutionEnvironment): DataStream[Row] = {
      // 定义获取DataStream数据集的逻辑
      val inputStream:DataStream[(String,String,Long)] = ...
      // ...
      // 指定数据集中的EventTime时间信息和Watermark
```

```
      val stream = inputStream.assignTimestampsAndWatermarks(...)
      stream
    }
    //定义Table API中的时间属性信息
    override def getRowtimeAttributeDescriptors:
util.List[RowtimeAttributeDescriptor] = {
      // 创建基于event_time的RowtimeAttributeDescriptor,确定时间属性信
息
      val rowtimeAttrDescr = new RowtimeAttributeDescriptor(
        "event_time",//时间属性名称
        new ExistingField("event_time"),
        new AscendingTimestamps)
      val rowtimeAttrDescrList =
Collections.singletonList(rowtimeAttrDescr)
      rowtimeAttrDescrList
    }
  }
```

将定义好的StreamTableSource注册到TableEnvironment中之后，然后在Flink Table API应用程序中使用创建好的Table，并且可以基于EventTime属性信息创建时间相关的操作算子。例如以下实例是基于event_time属性创建的滚动窗口，然后再基于窗口统计结果。

```
// 注册输入数据源
tStreamEnv.registerTableSource("InputEvent", new InputEventSource)
//在窗口中使用输入数据源,并基于TableSource中定义的EventTime字段创建窗口
val windowTable = tStreamEnv
  .scan("InputEvent")
  .window(Tumble over 10.minutes on 'event_time as 'window)
```

2.ProcessTime指定

（1）在DataStream转换Table的过程中定义

和EventTime时间属性一样，ProcessTime也可以在DataStream转换成Table的过程中定义。在Table API中，在ProcessTime时间字段名后使用.proctime后缀来指定ProcessTime时间属性，例如'process_time.proctime。和EventTime不同的是，ProcessTime属性只能在Table Schema尾部定义，不能基于指定位置来定义

ProcessTime属性，如以下代码示例，创建字段名称为process_time的ProcessTime属性。

```
//获取DataStream数据集
val stream: DataStream[(String, String)] = ...
// 将DataStream数据转换成Table
val table = tEnv.fromDataStream(stream, 'id, 'value,
'process_time.proctime)
//基于process_time时间属性创建滚动窗口
val windowTable = table.window(Tumble over 10.minutes on
'process_time as 'window)
```

（2）通过TableSource函数定义

和EventTime一样，也可以在创建TableSource的过程中定义ProcessTime字段，通过实现DefinedProctimeAttribute接口中的getRowtimeAttributeDescriptors方法，创建基于ProcessTime的时间属性信息，并在Table API中注册创建好的Table Source，最后便可以创建基于ProcessTime的操作算子。

```
// 定义InputEventSource创建外部数据源
// 并实现DefinedRowtimeAttributes接口以定义EventTime时间属性
class InputEventSource extends StreamTableSource[Row] with
DefinedProctimeAttribute {
  //定义数据集字段名称和类型
    override def getReturnType = {
      val names = Array[String]("id" , "value")
      val types = Array[TypeInformation[_]](Types.STRING,
Types.STRING)
      Types.ROW(names, types)
  }
    // 定义获取DataStream数据集的逻辑
    override def getDataStream(execEnv:
StreamExecutionEnvironment): DataStream[Row] = {
    // 定义获取DataStream数据集的逻辑
    val inputStream:DataStream[(String,String)] = ...
    // ...
    //将数据集转换成DataStream[Row]格式
    stream = inputStream.map(…)
    //不需要指定Watermark信息
    stream
```

```
    }
    //定义Table API中的时间属性信息
    override def getProctimeAttribute = {
    //该字段将会被添加到Schema的尾部
        "process_time"
    }
}
```

将定义好StreamTableSource注册到TableEnvironment中之后，就能够在Flink Table API中使用创建好的Table，并可以基于Process Time属性创建Window等基于时间属性的操作算子。以下实例基于process_time创建滚动窗口，然后基于窗口统计结果。

```
// 注册输入数据源
tStreamEnv.registerTableSource("InputEvent", new InputEventSource)
//在窗口中使用输入数据源,并基于TableSource中定义的Process Time字段创建窗口
val windowTable = tEnv
    .scan("InputEvent")
    .window(Tumble over 10.minutes on 'process_time as 'window)
```

7.1.5 Temporal Tables临时表

在Flink中通过Temporal Tables来表示其实数据元素一直不断变化的历史表，数据会随着时间的变化而发生变化。Temporal Tables底层其实维系了一张Append-Only Table，Flink对数据表的变化进行Track，在查询操作中返回与指定时间点对应的版本的结果。没有临时表时，如果想关联查询某些变化的指标数据，就需要在关联的数据集中通过时间信息将最新的结果筛选出来，显然这种做法需要浪费大量的计算资源。但如果使用临时表，则可直接关联查询临时表，数据会通过不断地更新以保证查询的结果是最新的。Temporal Tables的目的就是简化用户查询语句，加速查询的速度，同时尽可能地降低对状态的使用，因为不需要维护大量的历史数据。

Temporal Table Funciton定义

在Flink中，Temporal Tables使用Temporal Table Funciton来定义和表示。一旦Temporal Table Funciton被定义后，每次调用只需要传递时间参数，就可以返回与当前时间节点包含所有已经存在的Key的最新数据集合。定义Temporal Table Funciton需要主键信息和时间属性，其中主键主要用于覆盖数据记录以及确定返回结果，时间属性用于确定数据记录的有效性，用以返回最新的查询数据。如下代码实例，可以基于Append-Only Table来定义Temporal Table Funciton，在创建好的Table上调用createTemporalTable-Function(timeAttribute,primaryKey)，参数分别为时间属性和主键。

```
// 获取TableEnvironment
val tStreamEnv = TableEnvironment.getTableEnvironment(env)
//创建流式数据集
val ds: DataStream[(Long, Int)] = ...
//将DataStream数据集转换成Table
val tempTable = ds.toTable(tStreamEnv, 't_id, 't_value,
't_proctime.proctime)
//在TableEnvironment中注册表结构
tStreamEnv.registerTable("tempTable", tempTable)
//调用createTemporalTableFunction注册临时表,指定时间属性t_proctime及主键t_id
```

```
//在Table API中可以直接调用tempTableFunction
val tempTableFunction =
tempTable.createTemporalTableFunction('t_proctime, 't_id)
//在TableEnvironment中注册tempTableFunction信息,然后在SQL中通过名称调用
tEnv.registerFunction("tempTable", tempTableFunction)
```

Temporal Table Funciton定义好后，就可以在Table API或SQL中使用了。目前，Flink仅支持在Join算子中关联Temporal Table Funciton和Temporal Table，具体调用的细节可以参考7.2节和7.3节中的相关内容。

7.2 Flink Table API

Table API是Flink构建在DataSet和DataStream API之上的一套结构化编程接口，用户能够使用一套Table API编写流式应用和批量应用，不需要更改任何的代码逻辑。Flink Table API是Flink SQL的底层实现接口，也是Flink SQL的超集。Table API提供了非常丰富的操作算子用于对数据集进行处理。与Flink SQL不同的是，Table API是一种内嵌式语言，可以嵌入Java和Scala等语言中，可以通过IDE进行代码语法检测和自动填充等。Table API覆盖了所有批量和流式处理操作，其中大部分操作都同时支持流式处理和批量处理，只有个别操作是仅支持批量处理或者流式处理在以下介绍中我们会重点说明。

7.2.1 Table API应用实例

如代码清单7-10所示使用Table API构建实时和批量应用，其假设读取的表结构为某传感器信号表，表结构为<id:String, timestamp:Long, var1:Long, var2:Int>，字段分别为信号id、触发时间(t)、指标var1和指标var2。通过Table API统计与每个信号id对应的var2指标。需要注意的是如果使用Scala语言开发Table API类型应用程序，需要事先将org.apache.flink.api.scala._和org.apache.flink.table.api.scala._导入代码环境中，这样才能够使用Scala字符特性'x来获取字段。

代码清单7-10 Table API构建实时流式应用

```
//获取TableEnvironment
val tStreamEnv = TableEnvironment.getTableEnvironment(env)
// 通过scan方法在CataLog中找到Sensors表
val sensors:Table = tStreamEnv.scan("Sensors")
//对sensors 使用Table API进行处理
val result2 = sensors
  .groupBy('id)//根据id进行GroupBy操作
  .select('id, 'var1.sum as 'var1Sum)//查询id和var1的sum指标
  .toAppendStream[(String,Long)] //将处理结果转换成元祖类型DataStream数据集
```

从代码中可以看出，使用Table API构建Flink实时应用非常简单。上述代码可以在不需要修改任何Table API自身的代码的情况下应用在批处理计算上，用户只需要切换一下TableEnvironment即可。

7.2.2 数据查询和过滤

对于已经在TableEnvironment中注册的数据表，可以通过scan方法查询已经在CataLog中注册的表并转换为Table结构，然后在Table上使用select操作符查询需要获取的指定字段。如以下代码所示从TableEnvironment中查询已经注册的Sensors表。

```
val sensors: Table = tableEnv.scan("Sensors")
//可以通过在Table结构上使用select方法查询指定字段,并通过as进行字段重命名
val result = tableEnv.scan("Sensors").select('id, 'var1 as
'myvar1)
//使用select(*)将所有的字段查询出来
val result = tableEnv.scan("Sensors").select('*)
```

Table API中可以使用类似SQL中的as方法对字段进行重命名，例如将Sensors表中的字段按照位置分别命名为a、b、c、d。

```
val sensors: Table = tableEnv.scan("Sensors").as('a, 'b, 'c, 'd)
```

可以使用filter或where方法过滤字段和检索条件，将需要的数据检索出来。注意，在Table API语法中进行相等判断时需要三个等号连接表示。

```
//使用filter方法进行数据筛选
val result = sensors.filter('var1%2 === 0)
//使用where方法进行数据筛选
val result = sensors.where('id === "1001")
```

7.2.3 窗口操作

Flink Table API要将窗口分为GroupBy Window和OverWindow两种类型，具体的说明如下：

1.GroupBy Window

GroupBy Window和DataStream API、DataSet API中提供的窗口一致，都是将流式数据集根据窗口类型切分成有界数据集，然后在有界数据集之上进行聚合类运算。如下代码所示，在Table API中使用window()方法对窗口进行定义和调用，且必须通过as()方法指定窗口别名以在后面的算子中使用。在window()方法指定窗口类型之后，需要紧跟groupBy()方法来指定创建的窗口名称以窗口数据聚合的Key，然后使用Select()方法来指定需要查询的字段名称以及窗口聚合数据进行统计的函数，如以下代码指定为var1.sum，其他还可以为mm.max等。

```
//获取TableEnvironment
val tStreamEnv = TableEnvironment.getTableEnvironment(env)
// 通过scan方法在CataLog中找到Sensors表
val sensors:Table = tStreamEnv.scan("Sensors")
val result = sensors
  .window([w: Window] as 'window)  // 指定窗口类型并对窗口重命名为
window
  .groupBy('window)   // 根据窗口进行聚合,窗口数据会分配到单个Task算子中
  .select('var1.sum)  // 指定对var1字段进行sum求和
```

在流式计算任务中，GroupBy聚合条件中可以以上实例选择使用Window名称，也可是一个（或多个）Key值与Window的组合。如果仅指定Window名称，则和Global Window相似，窗口中的数据都会被汇合到一个Task线程中处理，统计窗口全局的结果；如果指定Key和Window名称的组合，则窗口中的数据会分布到并行的算子实例中计算结果。如下实例所示，GroupBy中指定除窗口名称以外的Key，完成对指定Key在窗口上的数据聚合统计。

```
//获取TableEnvironment
val tStreamEnv = TableEnvironment.getTableEnvironment(env)
// 通过scan方法在CataLog中找到Sensors表
val sensors:Table = tStreamEnv.scan("Sensors")
val result = sensors
  .window([w: Window] as 'window) // 指定窗口类型并将窗口重命名为window
  .groupBy('window, 'id)  // 根据窗口进行聚合,窗口数据会分配到单个Task算
子中
  .select('id, 'var1.sum)  // 指定对var1字段进行sum求和
```

在select语句中除了可以获取数据元素外，还可以获取窗口的元数据信息，例如可以通过window.start获取当前窗口的起始时间，通过window.end获取当前窗口的截止时间（含窗口区间上界），以及通过window.rowtime获取当前窗口截止时间（不含窗口区间上界）。

```
// 通过scan方法在CataLog中找到Sensors表
val sensors:Table = tStreamEnv.scan("Sensors")
val result = sensors
  .window([w: Window] as 'window) // 指定窗口类型并将窗口重命名为window
  .groupBy('window, 'id)  // 根据窗口进行聚合,窗口数据会分配到单个Task算
子中
// 指定对var1字段进行sum求和,并指定窗口起始时间、结束时间及rowtime等元数据信
息
  .select('id, 'var1.sum,'window.start, 'window.end,
'window.rowtime)
```

需要注意的是，在以上window()方法中需要指定的是不同的窗口类型，以确定数据元素被分配到窗口的逻辑。在Table API中支持Tumble、Sliding及Session Windows三种窗口类型，并分别通过不同的Window对象来完成定义。例如Tumbling Windows对应Tumble对象，Sliding Windows对应Slide对象，Session Windows对应Session对象，同时每种对象分别具有和自身窗口类型相关的参数。

（1）Tumbling Windows

前面已经提到滚动窗口的窗口长度是固定的，窗口和窗口之间的数据不会重合，例如每5min统计最近5min内的用户登录次数。滚动窗口可以基于Event Time、Process Time以及Row-Count来定义。如以下

代码实例，Table API中的滚动窗口使用Tumble Class来创建，且分别基于EventTime、ProcessTime以及Row-Count来定义窗口。

```
// 通过scan方法在CataLog中查询Sensors表
val sensors:Table = tStreamEnv.scan("Sensors")
// 基于EventTime时间概念创建滚动窗口,窗口长度为1h
sensors.window(Tumble over 1.hour on 'rowtime as 'window)
// 基于ProcessTime时间概念创建滚动窗口,窗口长度为1h
sensors.window(Tumble over 1.hour on 'proctime as 'window)
//基于元素数量创建滚动窗口,窗口长度为100条记录('proctime没有实际意义)
sensors.window(Tumble over 100.rows on 'proctime as 'window)
```

其中over操作符指定窗口的长度，例如over 10.minutes代表10min创建一个窗口，over 10.rows代表10条数据创建一个窗口。on操作符定义了窗口基于的时间概念类型为EventTime还是ProcessTime，EventTime对应着rowtime，ProcessTime对应着proctime。最后通过as操作符将创建的窗口进行重命名，同时窗口名称需要在后续的算子中使用。

（2）Sliding Windows

滑动窗口的窗口长度也是固定的，但窗口和窗口之间的数据能够重合，例如每隔10s统计最近5min的用户登录次数。滑动窗口也可以基于EventTime、ProcessTime以及Row-Count来定义。如下代码实例所示，Table API中的滑动窗口使用Slide Class来创建，且分别基于EventTime、ProcessTime以及Row-Count来定义窗口。

```
// 通过scan方法在CataLog中查询Sensors表
val sensors:Table = tStreamEnv.scan("Sensors")
// 基于EventTime时间概念创建滑动窗口,窗口长度为10分钟,每隔5s统计一次
sensors.window(Slide over 10.minutes every 5.millis on 'rowtime as
'window)
// 基于ProcessTime时间概念创建滑动窗口,窗口长度为10分钟,每隔5s统计一次
sensors.window(Slide over 10.minutes every 5.millis on 'procetime
as 'window)
//基于元素数量创建滑动窗口,指定10条记录创建一个窗口,窗口每5条记录移动一次
// 注意,'proctime没有实际意义
```

```
sensors.window(Slide over 100.rows every 5.rows on 'procetime as
'window)
```

上述代码中的over、on操作符与Tumpling窗口中的一样，都是指定窗口的固定长度和窗口的时间概念类型。和Tumpling窗口相比，Sliding Windows增加了every操作符，通过该操作符指定窗口的移动频率，例如every 5.millis表示窗口每隔5s移动一次。同时在最后创建Sliding窗口也需要使用as操作符对窗口进行重命名，并在后续操作中通过窗口名称调用该窗口。

（3）Session Windows

与Tumpling、Sliding窗口不同的是，Session窗口不需要指定固定的窗口时间，而是通过判断固定时间内数据的活跃性来切分窗口，例如10min内数据不接入则切分窗口并触发计算。Session窗口只能基于EventTime和ProcessTime时间概念来定义，通过withGap操作符指定数据不活跃的时间Gap，表示超过该时间数据不接入，则切分窗口并触发计算。如以下代码，通过指定EventTime和ProcessTime时间概念来创建Session Window。

```
// 通过scan方法在CataLog中查询Sensors表
val sensors:Table = tStreamEnv.scan("Sensors")
// 基于EventTime时间概念创建会话窗口,Session Gap为10min
sensors.window(Session withGap 10.minutes on 'rowtime as 'window)
// 基于ProcessTime时间概念创建会话窗口,Session Gap为10min
sensors.window(Session withGap 10.minutes on 'proctime as 'window)
```

2.Over Window

Over Window和标准SQL中提供的OVER语法功能类似，也是一种数据聚合计算的方式，但和Group Window不同的是，Over Window不需要对输入数据按照窗口大小进行堆叠。Over Window是基于当前数据和其周围邻近范围内的数据进行聚合统计的，例如基于当前记录前面的20条数据，然后基于这些数据统计某一指标的聚合结果。

在Table API中，Over Window也是在window方法中指定，但后面不需要和groupby操作符绑定，后面直接接select操作符，并在select操作符中指定需要查询的字段和聚合指标。如以下代码使用Over Class创建Over Window并命名为window，通过select操作符指定聚合指标var1.sum和var2.max。

```
// 通过scan方法在CataLog中查询Sensors表
val sensors:Table = tStreamEnv.scan("Sensors")
val table = sensors
    //指定OverWindow并重命名为window
sensors.window(Over partitionBy 'id orderBy 'rowtime preceding
UNBOUNDED_RANGE as 'window)
    // 通过在Select操作符中指定查询字段、窗口上var1求和值和var2最大值
  .select('id, 'var1.sum over 'window, 'var2.max over 'window)
```

上述Over Window的创建需要依赖于partitionBy、orderBy、preceding及following四个参数：

· partitionBy操作符中指定了一个或多个分区字段，Table中的数据会根据指定字段进行分区处理，并各自运行窗口上的聚合算子求取统计结果。需要注意，partitionBy是一个可选项，如果用户不使用partitionBy操作，则数据会在一个Task实例中完成计算，不会并行到多个Tasks中处理。

· orderBy操作符指定了数据排序的字段，通常情况下使用EventTime或ProcessTime进行时间排序。

· preceding操作符指定了基于当前数据需要向前纳入多少数据作为窗口的范围。preceding中具有两种类型的时间范围，其中一种为Bounded类型，例如指定100.rows表示基于当前数据之前的100条数据；也可以指定10.minutes，表示向前推10min，计算在该时间范围以内的所有数据。另外一种为UnBounded类型，表示从进入系统的第一条数据开始，且UnBounded类型可以使用静态变量UNBOUNDED_RANGE指定，表示以时间为单位的数据范围；也可以使用UNBOUNDED_ROW指定，表示以数据量为单位的数据范围。

· following操作符和preceding相反，following指定了从当前记录开始向后纳入多少数据作为计算的范围。目前Table API还不支持从当前记录开始向后指定多行数据进行窗口统计，可以使用静态变量CURRENT_ROW和CURRENT_RANGE来设定仅包含当前行，默认情况下Flink会根据用户使用窗口间隔是时间还是数量来指定following参数。需要注意的是，preceding和following指定的间隔单位必须一致，也就说二者必须是时间和数量中的一种类型。

如以下实例定义了Unbounded类型的Over Window，其中包括了UNBOUNDED_RANGE和UNBOUNDED_ROW两种preceding参数类型。

```
//创建UNBOUNDED_RANGE类型的OverWindow,指定分区字段为id,并根据rowtime排序
  .window([w: OverWindow] as 'window)
//创建UNBOUNDED_RANGE类型的OverWindow,指定分区字段为id,并根据proctime排序
sensors.window(Over partitionBy 'id orderBy 'proctime preceding
UNBOUNDED_RANGE as 'window)
//创建UNBOUNDED_ROW类型的OverWindow,指定分区字段为id,并根据rowtime排序
sensors.window(Over partitionBy 'id orderBy 'rowtime preceding
UNBOUNDED_ROW
as 'window)
//创建UNBOUNDED_ROW类型的OverWindow,指定分区字段为id,并根据proctime排序
sensors.window(Over partitionBy 'id orderBy 'proctime preceding
UNBOUNDED_ROW
as 'window)
```

如下实例定义了Unbounded类型的Over Window，其中包括UNBOUNDED_RANGE和UNBOUNDED_ROW两种preceding参数类型。

```
//创建BOUNDED类型的OverWindow,窗口大小为向前10min,并根据rowtime排序
sensors.window(Over partitionBy 'id orderBy 'rowtime preceding
10.minutes as
'window)
//创建BOUNDED类型的OverWindow,窗口大小为向前10min,并根据proctime排序
sensors.window(Over partitionBy 'id orderBy 'proctime preceding
10.minutes as
'window)
//创建BOUNDED类型的OverWindow,窗口大小为向前100条,并根据rowtime排序
```

```
sensors.window(Over partitionBy 'id orderBy 'rowtime preceding
100.rows as
'window)
//创建BOUNDED类型的OverWindow,窗口大小为向前100条,并根据rowtime排序
sensors.window(Over partitionBy 'id orderBy 'proctime preceding
100.rows as
'window)
```

7.2.4 聚合操作

在Flink Table API中提供了基于窗口以及不基于窗口的聚合类操作符基本涵盖了数据处理的绝大多数场景，和SQL中Group By语句相似，都是对相同的key值的数据进行聚合，然后基于聚合数据集之上统计例如sum、count、avg等类型的聚合指标。

1.GroupBy Aggregation

在全量数据集上根据指定字段聚合，首先将相同的key的数据聚合在一起，然后在聚合的数据集上计算统计指标。需要注意的是，这种聚合统计计算依赖状态数据，如果没有时间范围，在流式应用中状态数据根据不同的key及统计方法，将会在计算过程中不断地存储状态数据，所以建议用户尽可能限定统计时间范围避免因为状态体过大导致系统压力过大。

```
val sensors: Table = tStreamEnv.scan("Sensors")
//根据id进行聚合,求取Var1字段的sum结果
val groupResult = sensors.groupBy('id).select('id, 'var1.sum as
'var1Sum)
```

2.GroupBy Window Aggregation

该类聚合运算是构建在GroupBy Window之上然后根据指定字段聚合并统计结果。与非窗口统计相比，GroupBy Window可以将数据限定在一定范围内，这样能够有效控制状态数据的存储大小。如下代码实例所示，通过window操作符指定GroupBy Window类型之后，紧接着就是使用groupBy操作符指定需要根据哪些key进行数据的聚合，最后在select操作符中查询相关的指标。

```
val sensors: Table = tStreamEnv.scan("Sensors")
val groupWindowResult: Table = orders
  // 定义窗口类型为滚动窗口
  .window(Tumble over 1.hour on 'rowtime as 'window)
  // 根据id和window进行聚合
```

```
  .groupBy('id, 'window)
  //获取字段id,窗口属性start、end、rowtime,以及聚合指标var1Sum
  .select('id, 'window.start, 'window.end, 'window.rowtime,
'var1.sum as 'var1Sum)
```

3.Over Window Aggregation

和GroupBy Window Aggregation类似，但Over Window Aggregation是构建在Over Window之上，同时不需要在window操作符之后接groupby操作符。如以下代码实例所示在select操作符中通过“var1.avg over 'window”来指定需要聚合的字段及聚合方法。需要注意的是，在select操作符中只能使用一个相同的Window，且Over Window Aggregation仅支持preceding定义的UNBOUNDED和BOUNDED类型窗口，对于following定义的窗口目前不支持。同时Over Window Aggregation仅支持流式计算场景。

```
val sensors: Table = tStreamEnv.scan("Sensors")
val overWindowResult: Table = sensors
  // 定义UNBOUNDED_RANGE类型的OverWindow
  .window(Over partitionBy 'id orderBy 'rowtime preceding
UNBOUNDED_RANGE
as 'window)
  //获取字段id以及每种指标的聚合结果
  .select('id, 'var1.avg over 'window, 'var2.max over 'window,
'var3.min over 'window)
```

4.Distinct Aggregation

Distinct Aggregation和标准SQL中的COUNT(DISTINCT a)语法相似，主要作用是将Aggregation Function应用在不重复的输入元素上，对于重复的指标不再纳入计算范围内。Distinct Aggregation可以与GroupBy Aggregation、GroupBy Window Aggregation及Over Window Aggregation结合使用。

```
val sensors: Table = tStreamEnv.scan("Sensors")
// 基于GroupBy Aggregation,对不同的var1指标进行求和
val groupByDistinctResult = sensors
```

```
  .groupBy('id)
  .select('id, 'var1.sum.distinct as 'var1Sum)
// 基于GroupBy Window Aggregation,对不同的var1指标进行求和
val groupByWindowDistinctResult = sensors
  .window(Tumble over 1.minutes on 'rowtime as
'window).groupBy('id, 'window)
  .select('id, 'var1.sum.distinct as 'var1Sum)
// 基于GroupBy Window Aggregation,对不同的var1求平均值,并获取var2的最小
值
val overWindowDistinctResult = sensors
  .window(Over partitionBy 'id orderBy 'rowtime
   preceding UNBOUNDED_RANGE as 'window)
  .select('id, 'var1.avg.distinct over 'window, 'var2.min over
'window)
```

5.Distinct

单个Distinct操作符和标准SQL中的DISTINCT功能一样，用于返回唯一不同的记录。Distinct操作符可以直接应用在Table上，但是需要注意的是，Distinct操作是非常消耗资源的，且仅支持在批量计算场景中使用。

```
val sensors: Table = tStreamEnv.scan("Sensors")
//返回sensors表中唯一不同的记录
val distinctResult = sensors.distinct()
```

7.2.5 多表关联

1. Inner Join

Inner Join和标准SQL的JOIN语句功能一样，根据指定条件内关联两张表，并且只返回两个表中具有相同关联字段的记录，同时两张表中不能具有相同的字段名称。

```
val t1 = tStreamEnv.fromDataStream(stream1, 'id1, 'var1, 'var2)
val t2 = tStreamEnv.fromDataStream(stream2, 'id2, 'var3, 'var4)
val innerJoinresult = t1.join(t2).where('id1 ===
'id2).select('id1, 'var1, 'var3)
```

2. Outer Join

Outer Join操作符和标准SQL中的LEFT/RIGHT/FULL OUTER JOIN功能一样，且根据指定条件外关联两张表中不能有相同的字段名称，同时必须至少指定一个关联条件。如以下代码案例，分别对t1和t2表进行三种外关联操作。

```
//从DataSet数据集中创建Table
val t1 = tBatchEnv.fromDataSet(dataset1, 'id1, 'var1, 'var2)
val t2 = tBatchEnv.fromDataSet(dataset2, 'id2, 'var3, 'var4)
//左外关联两张表
val leftOuterResult = t1.leftOuterJoin(t2, 'id1 ===
'id2).select('id1, 'var1, 'var3)
//右外关联两张表
val rightOuterResult = t1. rightOuterJoin (t2, 'id1 ===
'id2).select('id1, 'var1, 'var3)
//全外关联两张表
val fullOuterResult = t1. fullOuterJoin (t2, 'id1 ===
'id2).select('id1, 'var1, 'var3)
```

3. Time-windowed Join

Time-windowed Join是Inner Join的子集，在Inner Join的基础上增加了时间条件，因此在使用Time-windowed Join关联两张表时，需要至少指定一个关联条件以及两张表中的关联时间，且两张表中的时间属性对应的时间概念必须一致（EventTime或者ProcessTime），时间属性对比使用Table API提供的比较符号（<, <=, >=, >），同时可以在条件中增加或者减少时间大小，例如rtime - 5.minutes，表示右表中的时间减去5分钟。

```
//从DataSet数据集中创建Table,并且两张表都使用EventTime时间概念
val t1= tBatchEnv.fromDataSet(dataset1, 'id1, 'var1,
'var2,'time1.rowtime)
val t2= tBatchEnv.fromDataSet(dataset2, 'id2, 'var3,
'var4,'time2.rowtime)
//将t1和t2表关联,并在where操作符中指定时间关联条件
val result = t1.join(t2)
//指定关联条件
.where('id1 === 'id2 && 'time1 >= 'time2 - 10.minutes && 'time1 <
'time2 + 10.minutes)
//查询并输出结果
.select('id1, 'var1, 'var2, 'time1)
```

4.Join with Table Function

在Inner Join中可以将Table与自定义的Table Funciton进行关联，Table中的数据记录与Table Fuction输出的数据进行内关联，其中如果Table Function返回空值，则不输出结果。

```
val table = tBatchEnv.fromDataSet(dataset, 'id, 'var1,
'var2,'time.rowtime)
// 初始化自定义的Table Function
val upper: TableFunction[_] = new MyUpperUDTF()
// 通过Inner Join关联经过MyUpperUDTF处理的Table,然后形成新的表
val result: Table = table
    .join(upper('var1) as 'upperVar1)
    .select('id, 'var1, 'upperVar1, 'var2)
```

在Left Outer Join中使用Table Funciton与使用Inner Join类似，区别在于如果Table Funciton返回的结果是空值，则在输出结果中对应的记录将会保留且Table Function输出的值为Null。

5.Join with Temporal Table

```
val tempTable = tEnv.scan("TempTable")
val temps = tempTable.createTemporalTableFunction('t_proctime,
't_id)
val table = tEnv.scan("Table")
val result = table.join(temps('o_rowtime), 'table_key ==
'temp_key)
```

7.2.6 集合操作

当两张Table都具有相同的Schema结构，则这两张表就可以进行类似于Union类型的集合操作。注意，以下除了UnionAll和In两个操作符同时支持流计算场景和批量计算场景之外，其余的操作符都仅支持批量计算场景。

```
//从DataSet数据集中创建Table
val t1= tBatchEnv.fromDataSet(dataset1, 'id1, 'var1, 'var2)
val t2= tBatchEnv.fromDataSet(dataset2, 'id2, 'var3, 'var4)
```

· Union：和标准SQL中的UNION语句功能相似，用于合并两张表并去除相同的记录。

```
val unionTable = t1.union(t2)
```

· UnionAll：和标准SQL中的UNIONALL语句功能相似，用于合并两张表但不去除相同的记录。

```
val unionAllTable = t1.unionAll(t2)
```

· Intersect：和标准SQL中的INTERSECT语句功能相似，合并两张变且仅返回两张表中的交集数据，如果记录重复则只返回一条记录。

```
val intersectTable = t1.intersect(t2)
```

· IntersectAll：和标准SQL中的INTERSECT ALL语句功能相似，合并两张表且仅返回两张表中的交集数据，如果记录重复则返回所有重复的记录。

```
val intersectAllTable = t1.intersectAll(t2)
```

· Minus：和标准SQL中EXCEPT语句功能相似，合并两张表且仅返回左表中有但是右表没有的数据差集，如果左表记录重复则只返回一条记录。

```
val minusTable = t1.minus(t2)
```

· MinusAll：和标准SQL中EXCEPT ALL语句功能相似，合并两张表仅返回左表有但是右表没有的数据差集，如果左表中记录重复n次，右表中相同记录出现m次，则返回n-m条记录。

```
val minusAllTable = t1.minusAll(t2)
```

· In：和标准SQL中IN语句功能相似，通过子查询判断左表中记录的某一列是否在右表中或给定的列表中，如果存在则返回True，如果不存在则返回False，where操作符根据返回条件判断是否返回记录。

```
val stream1: DataStream[(Long, String)] = ...
val stream2: DataStream[Long] = ...
val left: Table = tStreamEnv.fromDataStream(stream1,'id,'name)
val right: Table = tStreamEnv.fromDataStream(stream2,'id)
//使用in语句判断left表中记录的id是否在右表中,如过在则返回记录
val result1 = left.where('id in(right))
//使用in语句判断left表中记录的id是否在给定列表中,如过在则返回记录
val result2 = left.where('id in("92","11"))
```

7.2.7 排序操作

· Orderby：和标准SQL中ORDER BY语句功能相似，Orderby操作符根据指定的字段对Table进行全局排序，支持顺序（asc）和逆序（desc）两种方式。可以使用Offset操作符来控制排序结果输出的偏移量，使用fetch操作符来控制排序结果输出的条数。需要注意，该操作符仅支持批量计算场景。

```
val table: Table= ds.toTable(tBatchEnv, 'id,'var1,'var2)
//根据var1对table按顺序方式排序
val result = table.orderBy('var1.asc)
//根据id对table按逆序方式排序
val result1 = table.orderBy('var1.desc)
// 返回排序结果中的前5条
val result2: Table = in.orderBy('var1.asc).fetch(5)
// 忽略排序结果中的前10条数据,然后返回剩余全部数据
val result3: Table = in.orderBy('var1.asc).offset(10)
// 忽略排序结果中的前10条数据,然后返回剩余数据中的前5条数据
val result4: Table = in.orderBy('var1.asc).offset(10).fetch(5)
```

7.2.8 数据写入

通过Insert Into操作符将查询出来的Table写入注册在TableEnvironment的表中，从而完成数据的输出。注意，目标表的Schema结构必须和查询出来的Table的Schema结构一致。

```
val sensors: Table = tableEnv.scan("Sensors")
//将查询出来的表写入OutSensors表中,OutSensors必须已经在TableEnvironment中注册
sensors.insertInto("OutSensors")
```

7.3 Flink SQL使用

SQL作为Flink中提供的接口之一，占据着非常重要的地位，主要是因为SQL具有灵活和丰富的语法，能够应用于大部分的计算场景。Flink SQL底层使用Apache Calcite框架，将标准的Flink SQL语句解析并转换成底层的算子处理逻辑，并在转换过程中基于语法规则层面进行性能优化，比如谓词下推等。另外用户在使用SQL编写Flink应用时，能够屏蔽底层技术细节，能够更加方便且高效地通过SQL语句来构建Flink应用。Flink SQL构建在Table API之上，并含盖了大部分的Table API功能特性。同时Flink SQL可以和Table API混用，Flink最终会在整体上将代码合并在同一套代码逻辑中，另外构建一套SQL代码可以同时应用在相同数据结构的流式计算场景和批量计算场景上，不需要用户对SQL语句做任何调整，最终达到实现批流统一的目的。

7.3.1 Flink SQL实例

以下通过实例来了解Flink SQL整体的使用方式。在前面小节中我们知道如何在Flink TableEnvironment中注册和定义数据库或者表结构，最终是能够让用户方便地使用Table API或者SQL处理不同类型数据的，然后调用TableEnvironment SqlQuery方法执行Flink SQL语句，完成数据处理，如代码清单7-11所示。

代码清单7-11 TableEnvironment中执行Flink SQL

```
// 获取StreamTableEnvironment对象
val tableEnv = TableEnvironment.getTableEnvironment(env)
// 表结构schema (id, type,timestamp, var1, var2)
tableEnv.register("sensors", sensors_table)
val csvTableSink = new CsvTableSink("/path/csvfile", ...)
//定义字段名称
val fieldNames: Array[String] = Array("id", "type")
//定义字段类型
val fieldTypes: Array[TypeInformation[_]] = Array(Types.LONG,
Types.STRING)
//通过registerTableSink将CsvTableSink注册成table
tableEnv.registerTableSink("csv_output_table", fieldNames,
fieldTypes, csvSink)
// 计算与传感器类型为A的每个传感器id对应的var1指标的和
val result: Table = tEnv.sqlQuery(
"select id,
    sum(var1)as sumvar1
    from sensors_table
    where type='speed'
    group by sensor_id)")
// 通过SqlUpdate方法,将类型为温度的数据筛选出来并输出到外部表中
tableEnv.sqlUpdate(
  "INSERT INTO csv_output_table SELECT product, amount FROM
Sensors WHERE type = 'temperature'")
```

以上实例使用SQL语句筛选出sensor_type为“speed”的记录，并根据id对var1指标进行聚合并求和，执行完sqlQuery()方法后生成result Table，并且后续计算中可以直接使用result Table进行。在实例代码中可以看出，整个应用包括了从数据源的注册到具体数据处

理SQL转换，以及结果数据的输出。结果输出使用了Flink自带的CSVSink，然后将SQL执行的结果用INSERT INTO的方式写入对应CSV文件中，完成结果数据的存储落地。

7.3.2 执行SQL

Flink SQL可以借助于TableEnvironment的SqlQuery和SqlUpdate两种操作符使用，前者主要是从执行的Table中查询并处理数据生成新的Table，后者是通过SQL语句将查询的结果写入到注册的表中。其中SqlQuery方法中可以直接通过$符号引用Table，也可以事先在TableEnvironment中注册Table，然后在SQL中使用表名引用Table。

1. 在SQL中引用Table

如以下代码实例所示，创建好Table对象之后，可以在SqlQuery方法中直接使用$符号来引用创建好的Table，Flink会自动将被引用的Table注册到TableEnvironment中，从代码层面将Table API和SQL进行融合。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)
// 从外部数据源中转换形成DataStream数据集
val inputSteam: DataStream[(Long, String, Integer)] = ...
//将DataStream数据集转换成Table
val sensor_table = inputSteam.toTable(tableEnv, 'id, 'type, 'var1)
//在SqlQuery中直接使用$符号来引用创建好的Table
val result = tableEnv.sqlQuery(
  s"SELECT SUM(var1) FROM $sensor_table WHERE product ===
'temperature'")
```

2. 在SQL中引用注册表

如以下代码实例所示，事先调用registerDataStream方法将DataStream数据集在TableEnvironment中注册成Table，然后在sqlQuery()方法中SQL语句就直接可以通过Table名称来引用Table。

```
tableEnv.registerDataStream("Orders", ds, 'user, 'product,
'amount)
// 在SQL中直接引用注册好的Table名称,完成数据的处理,并输出结果
val result = tableEnv.sqlQuery(
```

```
  "SELECT product, amount FROM Sensors WHERE typeLIKE
'temperature'")
```

3. 在SQL中数据输出

如以下代码实例所示，可以调用sqlUpdate()方法将查询出来的数据输出到外部系统中，首先通过实现TableSink接口创建外部系统对应的TableSink，然后将创建好的TableSink实例注册在TableEnvironment中。最后使用sqlUpdate方法指定Insert Into语句将Table中的数据写入CSV文件TableSink对应的Table中，最终完成将Table数据的输出到CSO文件中。

```
val csvTableSink = new CsvTableSink("/path/csvfile", ...)
//定义字段名称
val fieldNames: Array[String] = Array("id", "type")
//定义字段类型
val fieldTypes: Array[TypeInformation[_]] = Array(Types.LONG,
Types.STRING)
//通过registerTableSink将CsvTableSink注册成Table
tableEnv.registerTableSink("csv_output_table", fieldNames,
fieldTypes, csvSink)
// 通过SqlUpdate方法,将类型为温度的数据筛选出并输出到外部表中
tableEnv.sqlUpdate(
  "INSERT INTO csv_output_table SELECT id, type FROM Sensors WHERE
type = 'temperature'")
```

7.3.3 数据查询与过滤

可以通过Select语句查询表中的数据，并使用Where语句设定过滤条件，将符合条件的数据筛选出来。

```
//查询Persons表全部数据
SELECT * FROM Sensors
//查询Persons表中name、age字段数据,并将age命名为d
SELECT id, type AS t FROM Sensors
//将信号类型为'temperature'的数据查询出来
SELECT * FROM Sensors WHERE type = 'temperature'
//将id为偶数的信号信息查询出来
SELECT * FROM Sensors WHERE id % 2 = 0
```

7.3.4 Group Windows窗口操作

Group Window是和GroupBy语句绑定使用的窗口，和Table API一样，Flink SQL也支持三种窗口类型，分别为Tumble Windows、HOP Windows和Session Windows，其中HOP Windows对应Table API中的Sliding Window，同时每种窗口分别有相应的使用场景和方法。

（1）Tumble Windows

滚动窗口的窗口长度是固定的，且窗口和窗口之间的数据不会重合。SQL中通过TUMBLE(time_attr, interval)关键字来定义滚动窗口，其中参数time_attr用于指定时间属性，参数interval用于指定窗口的固定长度。滚动窗口可以应用在基于EventTime的批量计算和流式计算场景中，和基于ProcessTime的流式计算场景中。窗口元数据信息可以通过在Select语句中使用相关的函数获取，且窗口元数据信息可用于后续的SQL操作，例如可以通过TUMBLE_START获取窗口起始时间，TUMBLE_END获取窗口结束时间，TUMBLE_ROWTIME获取窗口事件时间，TUMBLE_PROCTIME获取窗口数据中的ProcessTime。如以下实例所示，分别创建基于不同时间属性的Tumble窗口。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)
// 创建数据集
val ds: DataStream[(Long, String, Int)] = ...
// 注册表名信息并定义字段proctime为Process Time,定义字段rowtime为
rowtime,
tableEnv.registerDataStream("Sensors", ds, 'id, 'type, 'var1,
'proctime.proctime, 'rowtime.rowtime)
//基于proctime创建TUMBLE窗口,并指定10min切分为一个窗口,根据id进行聚合求取
var1的和
tableEnv.sqlQuery(SELECT id, SUM(var1) FROM Sensors GROUP BY
TUMBLE(proctime, INTERVAL '10' MINUTE), id"
//基于rowtime创建TUMBLE窗口,并指定5min切分为一个窗口,根据id进行聚合求取
var1的和
tableEnv.sqlQuery(SELECT id, SUM(var1) FROM Sensors GROUP BY
TUMBLE(proctime, INTERVAL '5’ MINUTE), id"
```

（2）HOP Windows

滑动窗口的窗口长度固定，且窗口和窗口之间的数据可以重合。在Flink SQL中通过HOP(time_attr, interval1, interval2)关键字来定义HOP Windows，其中参数time_attr用于指定使用的时间属性，参数interval1用于指定窗口滑动的时间间隔，参数interval2用于指定窗口的固定大小。其中如果interval1小于interval2，窗口就会发生重叠。HOP Windows可以应用在基于EventTime的批量计算场景和流式计算场景中，以及基于ProcessTime的流式计算场景中。HOP窗口的元数据信息获取的方法和Tumble的相似，例如可以通过HOP_START获取窗口起始时间，通过HOP_END获取窗口结束时间，通过HOP_ROWTIME获取窗口事件时间，通过HOP_PROCTIME获取窗口数据中的ProcessTime。

如以下代码所示，分别创建基于不同时间概念的HOP窗口，并通过相应方法获取窗口云数据。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)
// 创建数据集
val ds: DataStream[(Long, String, Int)] = ...
// 注册表名信息并定义字段proctime为ProcessTime,定义字段rowtime为rowtime,
tableEnv.registerDataStream("Sensors", ds, 'id, 'type, 'var1,
'proctime.proctime, 'rowtime.rowtime)
//基于proctime创建HOP窗口,并指定窗口长度为10min,每1min滑动一次窗口
//然后根据id进行聚合求取var1的和
tableEnv.sqlQuery(SELECT id, SUM(var1) FROM Sensors GROUP BY
HOP(proctime, INTERVAL '1' MINUTE,INTERVAL '10' MINUTE), id"
//基于rowtime创建HOP窗口,并指定窗口长度为10min,每5min滑动一次窗口
//根据id进行聚合求取var1的和
tableEnv.sqlQuery(SELECT id, SUM(var1) FROM Sensors GROUP BY
HOP(rowtime, INTERVAL '5' MINUTE,INTERVAL '10' MINUTE), id"
//基于rowtime创建HOP窗口,并指定5min切分为一个窗口,根据id进行聚合求取var1的
和
tableEnv.sqlQuery(SELECT id,
//获取窗口起始时间并记为wStart字段
HOP_START(rowtime, INTERVAL '5' MINUTE, INTERVAL '10' MINUTE) as
wStart,
//获取窗口起始时间并记为wEnd字段
HOP_START(rowtime, INTERVAL '5' MINUTE, INTERVAL '10' MINUTE) as
wEnd,
SUM(var1)
```

```
FROM Sensors GROUP BY HOP(rowtime, INTERVAL '5’ MINUTE, INTERVAL
'10’ MINUTE), id"
```

（3）Session Windows

Session窗口没有固定的窗口长度，而是根据指定时间间隔内数据的活跃性来切分窗口，例如当10min内数据不接入Flink系统则切分窗口并触发计算。在SQL中通过SESSION(time_attr, interval)关键字来定义会话窗口，其中参数time_attr用于指定时间属性，参数interval用于指定Session Gap。Session Windows可以应用在基于EventTime的批量计算场景和流式计算场景中，以及基于ProcessTime的流式计算场景中。

Session窗口的元数据信息获取与Tumble窗口和HOP窗口相似，通过SESSION_START获取窗口起始时间，SESSION_END获取窗口结束时间，SESSION_ROWTIME获取窗口数据元素事件时间，SESSION_PROCTIME获取窗口数据元素处理时间。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)
// 创建数据集
val ds: DataStream[(Long, String, Int)] = ...
// 注册表名信息并定义字段proctime为ProcessTime,定义字段rowtime为rowtime,
tableEnv.registerDataStream("Sensors", ds, 'id, 'type, 'var1,
'proctime.proctime, 'rowtime.rowtime)
//基于proctime创建SESSION窗口,指定Session Gap为1h
//然后根据id进行聚合求取var1的和
tableEnv.sqlQuery(SELECT id, SUM(var1) FROM Sensors GROUP BY
SESSION(proctime, INTERVAL '1' HOUR), id"
//基于rowtime创建SESSION窗口,指定Session Gap为1h
tableEnv.sqlQuery(SELECT id, SUM(var1) FROM Sensors GROUP BY
SESSION(proctime, INTERVAL '5' HOUR), id"
//基于rowtime创建SESSION窗口,指定Session Gap为1h,
tableEnv.sqlQuery(SELECT id,
//获取窗口起始时间并记为wStart字段
SESSION_START(proctime, INTERVAL '5' HOUR) as wStart,
//获取窗口起始时间并记为wStart字段
SESSION_END(rowtime, INTERVAL '5' HOUR) as wEnd,
  SUM(var1) FROM Sensors
GROUP BY SESSION(rowtime, INTERVAL '5' HOUR), id"
//基于rowtime创建SESSION窗口,指定Session Gap为1h
```

```
tableEnv.sqlQuery(SELECT id, SUM(var1) FROM Sensors GROUP BY
SESSION(rowtime, INTERVAL '5’ HOUR), id"
```

7.3.5 数据聚合

1.GroupBy Aggregation

GroupBy Aggregation在全景数据集上根据指定字段聚合，产生计算指标。需要注意的是，这种聚合统计计算主要依赖于状态数据，如果不指定时间范围，对于流式应用来说，状态数据会越来越大，所以建议用户尽可能在流式场景中使用GroupBy Aggregation。

```
SELECT id, SUM(var1) as d FROM Sensors GROUP BY id
```

2.GroupBy Window Aggregation

和Table API中的GroupBy Window Aggregation功能一致，GroupBy Window Aggregation基于窗口上的统计指定key的聚合结果，在FlinkSQL中通过在窗口之前使用GroupBy语句来定义。

```
//滚动窗口统计求和指标
SELECT id, SUM(var1) FROM Orders
  GROUP BY TUMBLE(rowtime, INTERVAL '1' DAY), user
//滑动窗口上统计最小值指标
SELECT id, MIN(var1) FROM Sensors
  GROUP BY HOP(rowtime, INTERVAL '1' HOUR, INTERVAL '1' DAY), id
//会话窗口上统计最大值指标
SELECT id, MAX(var1) FROM Sensors
  GROUP BY SESSION(rowtime, INTERVAL '1' DAY), id
```

3.Over Window Aggregation

Over Window Aggregation基于Over Window来计算聚合结果，可以使用Over关键字在查询语句中定义Over Window，也可以使用Window w AS()方式定义，在查询语句中使用定义好的window名称。注意Over Window所有的聚合算子必须指定相同的窗口，且窗口的数据范围目前仅支持PRECEDING到CURRENT ROW，不支持FOLLOWING语句。

```
//通过OVER关键字直接定义OVER WINDOW,并统计var1指标的最大值
SELECT MAX(var1) OVER (
//根据id进行聚合
  PARTITION BY id
//根据ProcessTime进行排序
  ORDER BY proctime
//ROWS数据限定在从当前数据向前推10条记录到当前数据
ROWS BETWEEN 10 PRECEDING AND CURRENT ROW)
FROM Sensors

//通过WINDOW关键字定义OVER WINDOW,通过OVER引用定义好的window
SELECT COUNT(var1) OVER window, SUM(var1) OVER window FROM Sensors
//定义WINDOW并重命名为window
WINDOW window AS (
  PARTITION BY id
  ORDER BY proctime
  ROWS BETWEEN 10 PRECEDING AND CURRENT ROW)
```

4.Distinct

和标准SQL中DISTINCT功能一样，Distinct用于返回唯一不同的记录。下面的代码通过Distinct关键子查询数据表中唯一不同的type字段。

```
SELECT DISTINCT type FROM Sensors
```

5.Grouping sets

和GROUP BY语句相比，Grouping sets将不同Key的GROUP BY结果集进行UNION ALL操作。以下代码表示通过指定id和type关键字同时进行聚合生成var1指标的结果。

```
SELECT SUM(var1) FROM Sensors GROUP BY GROUPING SETS ((id),
(type))
```

6.Having

和标准SQL中HAVING功能一样，Flink SQL中Having主要解决WHERE关键字无法与合计函数一起使用的问题，因此可以使用HAVING语句来对聚合结果筛选输出。

```
SELECT SUM(var1) FROM Sensors GROUP BY id HAVING SUM(var1) > 500
```

7.User-defined Aggregate Functions (UDAGG)

在SQL语句中使用自定义聚合函数，UDAGG需要事先在TableEnvironment中注册，然后通过函数名称在查询语句中使用。具体自定义函数相关细节读者可以参考7.4节。

```
SELECT MyAggregate(var1) FROM Sensors GROUP BY id
```

7.3.6 多表关联

1. Inner Join

Inner Join通过指定条件对两张表进行内关联，当且仅当两张表中都具有相同的key才会返回结果。

```
SELECT * FROM Sensors INNER JOIN Sensor_detail ON Sensors.id =
Sensor_detail.id
```

2. Outer Join

SQL外连接包括LEFT JOIN、RIGHT JOIN以及FULL OUTER三种类型，和标准的SQL语法一致，目前Flink SQL仅支持等值连接，不支持任意比较关系的theta连接。以下分别使用三种方式对两张表进行外关联。

```
//左外连接
SELECT * FROM Sensors LEFT JOIN Sensor_detail ON Sensors.id =
Sensor_detail.id
//右外连接
SELECT * FROM Sensors RIGHT JOIN Sensor_detail ON Sensors.id =
Sensor_detail.id
//全外连接
SELECT * FROM Sensors FULL OUTER JOIN Sensor_detail ON Sensors.id
= Sensor_detail.id
```

3. Time-windowed Join

和Inner Join类似，Time-windowed Join在Inner Join的基础上增加了时间属性条件，因此在使用Time-windowed Join关联两张表时，需要至少指定一个关联条件以及绑定两张表中的关联时间字段，且两张表中的时间属性对应的时间概念需要一致（Event Time或者ProcessTime），其中时间比较操作使用SQL中提供的比较符号（<,

<=, >=, >），且可以在条件中增加或者减少时间间隔，例如b.rowtime_INTERVAL'4'HOVR表示右表中的时间减去4h。

```
SELECT * FROM Sensors_1 a, sensors_2 b
WHERE a.id = b.id AND
    a.rowtime BETWEEN AND s.rowtime
```

4.Join with Table Function

在Inner Join或Left outer Join中可以使用自定义Table Funciton作为关联数据源，原始Table中的数据和Table Fuction产生的数据集进行关联，然后生成关联结果数据集。Flink SQL中提供了LATERAL TABLE语法专门应用在Table Funciton以及Table Funciton产生的临时表上，如果Table Function返回空值，则不输出结果。注意Table Function需要事先在TableEnvironment中定义，具体可以参考7.4节。

```
SELECT id, tag FROM Sensors, LATERAL TABLE(my_udtf(type)) t AS t
```

在Flink中临时表借助于Table Function生成，可以通过Flink LATERAL TABLE语法使用自定义的UDTF函数产生临时数据表的，并指定关联条件与临时表进行关联。目前临时表仅支持内关联操作，其他关联操作目前不支持，关于临时表的定义可以参见7.1.5节内容。

```
SELECT id, type FROM Sensors, LATERAL TABLE (my_udtf(o_proctime))
    WHERE type = type2
```

7.3.7 集合操作

· UNION：和标准SQL中的UNION语句一样，合并两张表同时去除相同的记录，注意两张表的表结构必须要一致，且该操作仅支持批量计算场景。

```
SELECT *
FROM (
    (SELECT user FROM Sensors WHERE var1 >= 0)
  UNION
    (SELECT user FROM Sensors WHERE type = 'temperature')
)
```

· UNION ALL：和标准SQL中的UNION ALL语句一样，合并两张表但不去除相同的记录，要求两张表的表结构必须一致。

```
SELECT *
FROM (
    (SELECT user FROM Sensors where var1  < 0)
  UNION ALL
    (SELECT user FROM Sensors where type = 'temperature')
)
```

· INTERSECT / EXCEPT：和标准SQL中的INTERSECT语句功能相似，合并两张变且仅返回两张表中的交集数据，如果记录重复则只返回一条记录。目前该操作仅支持批量计算场景。

```
SELECT *
FROM (
  (SELECT user FROM Sensors WHERE id % 2 = 0)
  INTERSECT
  (SELECT user FROM Sensors WHERE type = 'temperature')
)
```

· IN：和标准SQL中IN语句功能一样，通过子查询表达式判断左表中记录的某一列是否在右表中，如果在则返回True，如果不在则返回False，where语句根据返回条件判断是否返回记录。

```
SELECT id, type
FROM Sensors
WHERE type IN (
    SELECT type FROM Sensor_Types
)
```

· EXISTS：用于检查子查询是否至少返回一行数据，该子查询实际上并不返回任何数据，而是返回值True或False。和IN语句相比，EXISTS适合于子查询生成的表较大的情况。

```
SELECT id, type
FROM Sensors
WHERE type EXISTS (
    SELECT type FROM Sensor_Types
)
```

7.3.8 数据输出

通过INSERT Into语句将Table中的数据输出到外部介质中，需要事先将输出表注册在TableEnvironment中，查询语句对应的Schema和输出表结构必须一致。同时需要注意Insert Into语句只能被应用在SqlUpdate方法中，用于完成对Table中数据的输出。

```
INSERT INTO OutputTable SELECT id, type FROM Sensors
```

7.4 自定义函数

在Flink Table API中除了提供大量的内建函数之外，用户也能够实现自定义函数，这样极大地拓展了Table API和SQL的计算表达能力，使得用户能够更加方便灵活地使用Table API或SQL编写Flink应用。但需要注意的是，自定义函数主要在Table API和SQL中使用，对于DataStream和DataSet API的应用，则无须借助自定义函数实现，只要在相应接口代码中构建计算函数逻辑即可。

通常情况下，用户自定义的函数需要在Flink TableEnvironment中进行注册，然后才能在Table API和SQL中使用。函数注册通过TableEnvironment的registerFunction()方法完成，本质上是将用户自定义好的Function注册到TableEnvironment中的Function CataLog中，每次在调用的过程中直接到CataLog中获取函数信息。Flink目前没有提供持久化注册的接口，因此需要每次在启动应用的时候重新对函数进行注册，且当应用被关闭后，TableEnvironment中已经注册的函数信息将会被清理。

在Table API中，根处理的数据类型以及计算方式的不同将自定义函数一共分为三种类别，分别为Scalar Function、Table Function和Aggregation Function。

7.4.1 Scalar Function

Scalar Function也被称为标量函数，表示对单个输入或者多个输入字段计算后返回一个确定类型的标量值，其返回值类型可以为除TEXT、NTEXT、IMAGE、CURSOR、 TIMESTAMP和TABLE类型外的其他所有数据类型。例如Flink常见的内置标量函数有DATE()、UPPER()、LTRIM()等，同时在自定义标量函数中，用户需要确认Flink内部是否已经实现相应的Scalar Fuction，如果已经实现则可以直接使用；如果没有实现，则在注册自定义函数过程中，需要和内置的其他Scalar Function名称区分，否则会导致注册函数失败，影响应用的正常执行。

定义Scalar Function需要继承org.apache.flink.table.functions包中的ScalarFunction类，同时实现类中的evaluation方法，自定义函数计算逻辑需要在该方法中定义，同时该方法必须声明为public且将方法名称定义为eval。同时在一个ScalarFunction实现类中可以定义多个evaluation方法，只需要保证传递进来的参数不相同即可。如代码清单7-12所示，通过定义AddClass并继承ScalarFunction接口，实现对两个数值相加的功能。然后在Table Select操作符和SQL语句中使用。

代码清单7-12 自定义实现Scalar Function实现字符串长度获取

```
// 注册输入数据源
tStreamEnv.registerTableSource("InputTable", new InputEventSource)
//在窗口中使用输入数据源,并基于TableSource中定义的EventTime字段创建窗口
val table: Table = tStreamEnv.scan("InputTable")
// 在Object或者静态环境中创建自定义函数
class Add extends ScalarFunction {
  def eval(a: Int, b: Int): Int = {//整型数据相加
    if (a == null || b == null) null
    a + b}
  def eval(a: Double, b: Double): Double = {//Double类型数据相加
    if (a == null || b == null) null
    a + b}
}
// 实例化ADD函数
val add = new Add
```

```
// 在Scala Table API中使用自定义函数
val result = table.select('a, 'b, add('a, 'b))
// 在Table Environment中注册自定义函数
tStreamEnv.registerFunction("add", new Add)
//在SQL中使用ADD Scalar函数
tStreamEnv.sqlQuery("SELECT a,b, ADD(a,b) FROM InputTable")
```

在自定义标量函数过程中，函数的返回值类型必须为标量值，尽管Flink内部已经定义了大部分的基本数据类型以及POJOs类型等，但有些比较复杂的数据类型如果Flink不支持获取，此时需要用户通过继承并实现ScalarFunction类中的getResultType实现getResult-Type方法对数据类型进行转换。例如在Table API和SQL中可能需要使用Types.TIMESTAMP数据类型，但是基于ScalarFunction得出的只能是Long类型，因此可以通过实现getResultType方法对结果数据进行类型转换，从而返回Timestamp类型。

```
object LongToTimestamp extends ScalarFunction {
  def eval(t: Long): Long = { t % 1000}
  override def getResultType(signature: Array[Class[_]]):
TypeInformation[_]
    = {
        Types.TIMESTAMP
      }}
```

7.4.2 Table Function

和Scalar Function不同，Table Function将一个或多个标量字段作为输入参数，且经过计算和处理后返回的是任意数量的记录，不再是单独的一个标量指标，且返回结果中可以含有一列或多列指标，根据自定义Table Funciton函数返回值确定，因此从形式上看更像是Table结构数据。

定义Table Function需要继承org.apache.flink.table.functions包中的TableFunction类，并实现类中的evaluation方法，且所有的自定义函数计算逻辑均在该方法中定义，需要注意方法必须声明为public且名称必须定义为eval。另外在一个TableFunction实现类中可以实现多个evaluation方法，只需要保证参数不相同即可。

在Scala语言Table API中，Table Function可以用在Join、LeftOuterJoin算子中，Table Function相当于产生一张被关联的表，主表中的数据会与Table Function所有产生的数据进行交叉关联。其中LeftOuterJoin算子当Table Function产生结果为空时，Table Function产生的字段会被填为空值。

在应用Table Function之前，需要事先在TableEnvironment中注册Table Function，然后结合LATERAL TABLE关键字使用，根据语句结尾是否增加ON TRUE关键字来区分是Join还是leftOuterJoin操作。如代码清单7-13所示，通过自定义SplitFunction Class继承TableFunction接口，实现根据指定切割符来切分输入字符串，并获取每个字符的长度和HashCode的功能，然后在Table Select操作符和SQL语句中使用定义的SplitFunction。

代码清单7-13 自定义Table Funciton实现将给定字符串切分成多条记录

```
// 注册输入数据源
tStreamEnv.registerTableSource("InputTable", new InputEventSource)
// 在Scala Table API中使用自定义函数
val split = new SplitFunction(",")
```

```
//在join函数中调用Table Function,将string字符串切分成不同的Row,并通过as指定字段名称为str,length,hashcode
table.join(split('origin as('string, 'length, 'hashcode)))
.select('origin, 'str, 'length, 'hashcode)
table.leftOuterJoin(split('origin as('string, 'length,
'hashcode)))
.select('origin, 'str, 'length, 'hashcode)
// 在Table Environment中注册自定义函数,并在SQL中使用
tStreamEnv.registerFunction("split", new SplitFunction(","))
//在SQL中和LATERAL TABLE关键字一起使用Table Function
//和Table API的JOIN一样,产生笛卡儿积结果
tStreamEnv.sqlQuery("SELECT origin, str, length FROM InputTable,
LATERAL TABLE(split(origin)) as T(str, length,hashcode)")
//和Table API中的LEFT OUTER JOIN一样,产生左外关联结果
tStreamEnv.sqlQuery("SELECT origin, str, length FROM InputTable,
LATERAL TABLE(split(origin)) as T(str, length,hashcode) ON TRUE")
```

和Scalar Function一样，对于不支持的输出结果类型，可以通过实现TableFunction接口中的getResultType()对输出结果的数据类型进行转换，具体可以参考ScalarFunciton定义。

7.4.3 Aggregation Function

Flink Table API中提供了User-Defined Aggregate Functions (UDAGGs)，其主要功能是将一行或多行数据进行聚合然后输出一个标量值，例如在数据集中根据Key求取指定Value的最大值或最小值。

自定义Aggregation Function需要创建Class实现org.apache.flink.table.functions包中的AggregateFunction类。关于AggregateFunction的接口定义如代码清单7-14所示可以看出AggregateFunction定义相对比较复杂。

代码清单7-14 AggregateFunction定义

```
public abstract class AggregateFunction<T, ACC> extends
UserDefinedFunction {
    // 创建Accumulator（强制）
    public ACC createAccumulator();
    // 累加数据元素到ACC中（强制）
    public void accumulate(ACC accumulator, [user defined
inputs]);
    // 从ACC中去除数据元素（可选）
    public void retract(ACC accumulator, [user defined inputs]);
    // 合并多个ACC（可选）
    public void merge(ACC accumulator, java.lang.Iterable<ACC>
its);
    //获取聚合结果（强制）
    public T getValue(ACC accumulator);
    //重置ACC（可选）
    public void resetAccumulator(ACC accumulator);
    //如果只能被用于Over Window则返回True(预定义)
    public Boolean requiresOver = false;
    //指定统计结果类型(预定义)
    public TypeInformation<T> getResultType = null;
    //指定ACC数据类型（预定义）
    public TypeInformation<T> getAccumulatorType = null;
  }
```

在AggregateFunction抽象类中包含了必须实现的方法createAccumulator()、accumulate()、getValue()。其中，

createAccumulator()方法主要用于创建Accumulator，以用于存储计算过程中读取的中间数据，同时在Accumulator中完成数据的累加操作；accumulate()方法将每次接入的数据元素累加到定义的accumulator中，另外accumulate()方法也可以通过方法复载的方式处理不同类型的数据；当完成所有的数据累加操作结束后，最后通过getValue()方法返回函数的统计结果，最终完成整个AggregateFunction的计算流程。

除了以上三个必须要实现的方法之外，在Aggregation Function中还有根据具体使用场景选择性实现的方法，如retract()、merge()、resetAccumulator()等方法。其中，retract()方法是在基于Bouded Over Windows的自定义聚合算子中使用；merge()方法是在多批聚合和Session Window场景中使用；resetAccumulator()方法是在批量计算中多批聚合的场景中使用，主要对accumulator计数器进行重置操作。

因为目前在Flink中对Scala的类型参数提取效率相对较低，因此Flink建议用户尽可能实现Java语言的Aggregation Function，同时应尽可能使用原始数据类型，例如Int、Long等，避免使用复合数据类型，如自定义POJOs等，这样做的主要原因是在Aggregation Function计算过程中，期间会有大量对象被创建和销毁，将对整个系统的性能造成一定的影响。

7.5 自定义数据源

通过前面小节已经知道，Flink Table API可以支持很多种数据源的接入，除了能够使用已经定义好的TableSource数据源之外，用户也可以通过自定义TableSource完成从其他外部数据介质（数据库，消息中间件等）中接入流式或批量类型的数据。Table Source在TableEnviroment中定义好后，就能够在Table API和SQL中直接使用。

与Table Source相似的在Table API中提供通过TableSink接口定义对Flink中数据的输出操作，包括将数据输出到外部的存储系统中，例如常见的数据库、消息中间件及文件系统等。用户实现TableSink接口并在TableEnvironment中注册，就能够在Table API和SQL中获取TableSink对应的Table，然后将数据输出到TableSink对应的存储介质中。

7.5.1 TableSource定义

TableSource是在Table API中专门针对获取外部数据提出的通用数据源接口。TableSource定义中将数据源分为两类，一种为StreamTableSource，主要对应流式数据源的数据接入；另外一种BatchTableSource，主要对应批量数据源的数据接入。且两者都是TableSource的子类，TableSource定义如代码清单7-15所示。

代码清单7-15 TableSource接口定义

```
TableSource<T> {
  public TableSchema getTableSchema();
  public TypeInformation<T> getReturnType();
  public String explainSource();
}
```

可以看出在TableSource接口中，共有getTableSchema()、getReturnType()和explainSource()三个方法需要实现。其中，getTableSchema()方法用于指定数据源的Table Schema信息，例如字段名称和类型等；getReturnType()方法用于返回数据源中的字段数据类型信息，所有的返回字段必须是Flink TypeInformation支持的类型；explainSource()方法用于返回TableSource的描述信息，功能类似于SQL中的explain方法。

在Table API中，将TableSource分为主要针对流式数据接入的StreamTableSource和主要针对批量数据接入的BatchTableSource。其中StreamTableSource是从DataStream数据集中将数据转换成Table，BatchTableSource是从DataSet数据集中将数据转换成Table，以下分别介绍每种TableSource的定义和使用。

1.StreamTableSource

如以下代码所示StreamTableSource是TableSource的子接口，在StreamTableSource中可以通过getDataStream方法将数据源从外部介质中抽取出来并转换成DataStream数据集，且对应的数据类型必须是

TableSource接口中getReturnType方法中返回的数据类型。StreamTableSource可以看成是对DataStream API中SourceFunciton的封装，并且在转换成Table的过程中增加了Schema信息。

```
StreamTableSource[T] extends TableSource[T] {
//定义获取DataStream的逻辑
  def getDataStream(execEnv: StreamExecutionEnvironment):
DataStream[T]
}
```

如代码清单7-16所示，通过实现StreamTableSource接口，完成了从外部数据源中获取DataStream数据集，并在getTableSchema方法中指定数据源对应的Table的Schema结构信息。

代码清单7-16　自定义实现StreamTableSource的接口完成对流动式数据的接入

```
// 定义InputEventSource
class InputEventSource extends StreamTableSource[Row] {
  override def getReturnType = {
    val names = Array[String]("id", "value")
    val types = Array[TypeInformation[_]](Types.STRING,
Types.LONG)
    Types.ROW(names, types)
  }
//实现getDataStream方法,创建输入数据集
  override def getDataStream(execEnv: StreamExecutionEnvironment):
DataStream[Row] = {
    // 定义获取DataStream数据集
    val inputStream: DataStream[(String, Long)] =
execEnv.addSource(...)
    //将数据集转换成指定数据类型
    val stream: DataStream[Row] = inputStream.map(t =>
Row.of(t._1, t._2))
    stream
  }
  //定义TableSchema信息
  override def getTableSchema: TableSchema = {
    val names = Array[String]("id", "value")
    val types = Array[TypeInformation[_]](Types.STRING,
Types.LONG)
```

```
    new TableSchema(names, types)
  }
}
```

定义好的StreamTableSource之后就可以在Table API和SQL中使用，在Table API中通过registerTableSource方法将定义好的TableSource注册到TableEnvironment中，然后就可以使用scan操作符从TableEnvironment中获取Table在SQL中则直接通过表名引用注册好的Table即可。

```
// 注册输入数据源
  tStreamEnv.registerTableSource("InputTable", new
InputEventSource)
//在窗口中使用输入数据源,并基于TableSource中定义的EventTime字段创建窗口
  val table: Table = tStreamEnv.scan("InputTable")
```

2. BatchTableSource

和StreamTableSource相似，BatchTableSource接口具有了getDataSet()方法，主要将外部系统中的数据读取并转换成DataSet数据集，然后基于对DataSet数据集进行处理和转换，生成BatchTableSource需要的数据类型。其中DataSet数据集中的数据格式也必须要和TableSource中getReturnType返回的数据类型一致。BatchTableSource接口定义如下。

```
BatchTableSource<T> implements TableSource<T> {
  public DataSet<T> getDataSet(ExecutionEnvironment execEnv);
}
```

其中，BatchTableSource本质上也是实现DataSet底层的SourceFunction，如代码清单7-17所示，通过实例化BatchTableSource完成对外部批量数据的接入，然后在Table API中应用之后定义好的BatchTableSource。

代码清单7-17　自定义实现BatchTableSource的接口完成批量数据接入

```
// 创建InputEventSource
class InputBatchSource extends BatchTableSource[Row] {
  //定义结果类型信息
  override def getReturnType = {
    val names = Array[String]("id", "value")
    val types = Array[TypeInformation[_]](Types.STRING,
Types.LONG)
    Types.ROW(names, types)
  }
  //获取DataSet数据集
  override def getDataSet(execEnv: ExecutionEnvironment):
DataSet[Row] = {
    //从外部系统中读取数据
    val inputDataSet = execEnv.createInput(...)
    val dataSet: DataSet[Row] = inputDataSet.map(t => Row.of(t._1,
t._2))
    dataSet
  }
  //定义TableSchema信息
  override def getTableSchema: TableSchema = {
    val names = Array[String]("id", "value")
    val types = Array[TypeInformation[_]](Types.STRING,
Types.LONG)
    new TableSchema(names, types)
  }}}
```

BathTableSource和StreamTableSource的使用方式一样，也需要在TableEnvironment中注册已经创建好的TableSource信息，然后就可以在Table API或SQL中使用TableSource对应的表结构。

7.5.2 TableSink定义

与TableSource接口定义相似，TableSink接口的主要功能是将Table中的数据输出到外部系统中。如代码清单7-18所示TableSink接口中主要包含getOutputType、getFieldNames、getFieldTypes和configure四个方法。其中，getOutputType定义了输出的数据类型，且类型必须是TypeInformation所支持的类型；getFieldNames方法定义了当前Table中的字段名称；getFieldTypes方法和getFieldNames对应，返回了Table中字段的数据类型；configure方法则定义了输出配置，其中fieldNames定义了输出字段名称，fieldTypes定义了输出数据类型。

代码清单7-18 TableSink接口定义

```
TableSink<T> {
  public TypeInformation<T> getOutputType();
  public String[] getFieldNames();
  public TypeInformation[] getFieldTypes();
  public TableSink<T> configure(String[] fieldNames,
TypeInformation[] fieldTypes);
}
```

在TableSink接口中，分别通过BatchTableSink和StreamingTableSink两个子接口定义和实现对批数据和流数据的输出功能。

1.StreamTableSink

可以通过实现StreamTableSink接口将Table中的数据以流的形式输出。在Stream-TableSink接口中emitDataStream方法定义了Table中输出数据的逻辑，实际是将Data-Stream数据集发送到对应的存储系统中。另外根据Table中数据记录更新的方式不同，将StreamTableSink分为AppendStreamTableSink、RetractStreamTableSink以及UpsertStreamTableSink三种类型。

（1）AppendStreamTableSink

AppendStreamTableSink只输出在Table中所有由于INSERT操作所更新的记录，对于类似于DELTE操作更新的记录则不输出。如果用户同时输出了INSERT和DELETE操作的数据，则系统会抛出TableException异常信息。AppendStreamTableSink接口定义如下。

```
AppendStreamTableSink<T> implements TableSink<T> {
  public void emitDataStream(DataStream<T> dataStream);
}
```

（2）RetractStreamTableSink

RetractStreamTableSink同时输出INSERT和DELETE操作更新的记录，输出结果会被转换为Tuple2< Boolean, T>的格式。其中，Boolean类型字段用于对结果进行标记，如果是INSERT操作更新的记录则标记为true，反之DELETE操作更新的记录则标记为false；第二个字段为具体的输出数据。和AppendStreamTableSink相比RetractStreamTableSink则更加灵活，可以将全部操作更新的数据输出，并把筛选和处理的逻辑交给用户控制。RetractStreamTableSink接口定义如以下代码所示，接口中包括getRecordType和emitDataStream两个方法，getRecordType主要返回输出数据集对应的数据类型，emitDataStream定义数据输出到外部系统的逻辑。

```
RetractStreamTableSink<T> implements TableSink<Tuple2<Boolean, T>>
{
  public TypeInformation<T> getRecordType();
  public void emitDataStream(DataStream<Tuple2<Boolean, T>>
dataStream);
}
```

（3）UpsertStreamTableSink

和RetractStreamTableSink相比，UpsertStreamTableSink增加了对与UPDATE操作对应的记录输出的支持。该接口能够输出INSERT、UPDATE、DELETE三种操作更新的记录。使用UpsertStreamTableSink接口，需要指定输出相应的唯一主键keyFields，可以是单个字段的或者

多个字段的组合，如果KeyFields不唯一且AppendOnly为false时，该接口中的方法会抛出TableException。

```
UpsertStreamTableSink<T> implements TableSink<Tuple2<Boolean, T>>
{
  //指定对应的keyFields,需要用户保持唯一性
    public void setKeyFields(String[] keys);
  // 设定Table输出模式是否为AppendOnly
    public void setIsAppendOnly(boolean isAppendOnly);
  //指定Table中的数据类型
    public TypeInformation<T> getRecordType();
  //定义数据输出逻辑
    public void emitDataStream(DataStream<Tuple2<Boolean, T>>
dataStream);
}
```

以上是UpsertStreamTableSink的接口定义，UpsertStreamTableSink接口中emitDataStream方法中的输入数据集的格式为Tuple2<Boolean,T>类型、其中，第一个Boolean字段标记UNPSERT更新为true，DELETE字段更新为false；第二个字段为类型T的输出数据记录。

2.BatchTableSink

BatchTableSink接口主要用于对批量数据的输出，和StreamTableSink不同的是，该接口底层操作的是DataSet数据集。BatchTableSink中没有区分是INSERT还是DELETE等操作更新的数据，而是全部都统一输出。BatchTableSink接口定义如下。在BatchTableSink接口中通过实现emitDataSet方法定义DataSet<T> 数据集来输出外部系统的逻辑。

```
BatchTableSink<T> implements TableSink<T> {
  public void emitDataSet(DataSet<T> dataSet);
}
```

7.5.3 TableFactory定义

TableFactory主要作用是将事先定义好的TableSource和TableSink实现类封装成不同的Factory，然后通过字符参数进行配置，这样在Table API或SQL中就可以使用配置参数来引用定义好的数据源。TableFactory使用Java的SPI机制为TableFactory来寻找接口实现类，因此需要保证在META-INF/services/资源目录中包含所有与TableFactory实现类对应的配置列表，所有的TableFactory实现类都将被加载到Classpath中，然后应用中就能够通过使用TableFactory来读取输出数据集在Flink集群启动过程。

TableFactory接口定义中包含requiredContext和supportedProperties两个方法，其中requiredContext定义了当前实现的TableFactory中的Context上下文，同时通过Key-Value的方式来标记TableFactory，例如connector.type=dev-system，Flink应用中配置参数需要和Context具有相同的Key-Value参数才能够匹配到TableSource并使用，否则不会匹配相应的TableFactory实现类。在supportedProperties方法中定义了当前TableFactory中需要使用到的参数，如果Flink应用中配置的参数不属于当前的TableFactory，便会抛出异常。需要注意的是，Context参数中的Key不能和supportedProperties参数名称相同。如代码清单7-19所示，通过定义SocketTableSourceFactory实现类，完成从Socket端口中接入数据，并在Table API或SQL Client使用。

代码清单7-19 自定义实现SocketTableSourceFactory

```
class SocketTableSourceFactory extends
StreamTableSourceFactory[Row] {
  //指定TableFactory上下文参数
    override def requiredContext(): util.Map[String, String] = {
      val context = new util.HashMap[String, String]()
      context.put("update-mode", "append")
      context.put("connector.type", "dev-system")
      context
    }
      //指定TableFactory用于处理的参数集合
    override def supportedProperties(): util.List[String] = {
      val properties = new util.ArrayList[String]()
```

```
      properties.add("connector.host")
      properties.add("connector.port")
      properties
    }
        //创建SocketTableSource实例
        override def createStreamTableSource(properties:
util.Map[String, String]): StreamTableSource[Row] = {
      val socketHost = properties.get("connector.host")
      val socketPort = properties.get("connector.port")
      new SocketTableSource(socketHost, socketPort)
    }
}
```

1.在SQL Client中使用TableFactory

SQL Client能够支持用户在客户端中使用SQL编写Flink应用，从而可以在SQL Client中查询记录实现和定义好的SocketTableSource中数据的数据源，可以通过代码清单7-20的配置文件进行配置。

代码清单7-20 TableFactory SQL Client Yaml配置

```
tables:
- name: SocketTable
  type: source //指定Table类型是source还是sink
  update-mode: append
  connector:
    type: dev-system
    host: localhost
    port: 10000
```

配置文件通过Yaml文件格式进行配置，需要放置在SQL Client environment文件中，文件中的配置项将直接被转换为扁平的字符配置，然后传输给相应的TableFactory。注意，需要将对应的TableFactory事先注册至Flink执行环境中，然后才能将配置文件项传递给对应的TableFactory，进而完成数据源的构建。

2.在Table & SQL API中使用TableFactory

如果用户想在Table & SQL API中使用TableFactory定义的数据源，也需要将对应的配置项传递给对应的TableFactory。为了安全起见，Flink提供了ConnectorDescriptor接口让用户定义连接参数，然后转换成字符配置项，具体的实现方式如代码清单7-21所示。

代码清单7-21　通过ConnectorDescriptor定义MySocketConnector

```
class MySocketConnector(host:String,port:String) extends
  ConnectorDescriptor("dev-system", 1, false) {
  override protected def toConnectorProperties(): Map[String,
String] = {
      val properties = new HashMap[String, String]
      properties.put("connector.host", host)
      properties.put("connector.port", port)
      properties
    }}
```

创建MySocketConnector之后，在Table & SQL API中通过ConnectorDescriptor连接Connector，然后调用registerTableSource将对应的TableSource注册到TableEnvironment中。接下来就可以在Table & SQL API中正式使用注册的Table，以完成后续的数据处理了。

```
val tableEnv: StreamTableEnvironment = // ...
tableEnv.connect(new MySocketConnector("localhost","10000"))
  .inAppendMode()
  .registerTableSource("MySocketTable")
```

7.6 本章小结

本章重点介绍了Flink Table API & SQL的使用，在7.1节中介绍了Flink Table & SQL API开发环境主要基本操作，其中包括对数据源、函数等信息的注册等。7.2节中重点介绍了Flink Table API的使用方式，包括聚合、多表关联、窗口计算等常用的数据操作。7.3节中重点介绍了Flink SQL API的使用，让用户可以通过SQL来构建Flink应用。7.4节中介绍了如何在Table API & SQL中使用自定义函数，其中自定义函数包括Scalar Function、Table Function、Aggregation Function等常用的函数类型，然后在Table API & SQL中使用自定义函数。7.5节介绍了如何通过自定义TableSource和TableSink完成与外部系统数据源的对接并通过TubleFactory对TableSoure和TableSink进行封装并利用配置调用相应的数据源。

第8章

Flink组件栈介绍与使用

通过对前面章节的学习，我们对Flink的基本编程接口有了一定的认识和了解，在本章将重点介绍Flink在不同的应用领域中所提供的组件栈，其中包括构建复杂事件处理应用的FlinkCEP组件栈，构建机器学习应用的FlinkML组件栈，以及构建图计算应用的Gelly组件栈。这些组件栈本质上都是构建在DataSet或DataStream接口之上的，其主要目的就是方便用户构建不同应用领域的应用。

8.1 Flink复杂事件处理

复杂事件处理（CEP）是一种基于流处理的技术，将系统数据看作不同类型的事件，通过分析事件之间的关系，建立不同的事件关系序列库，并利用过滤、关联、聚合等技术，最终由简单事件产生高级事件，并通过模式规则的方式对重要信息进行跟踪和分析，从实时数据中发掘有价值的信息。复杂事件处理主要应用于防范网络欺诈、设备故障检测、风险规避和智能营销等领域。目前主流的CEP工具有Esper、Jboss Drools和商业版的MicroSoft StreamInsight等，Flink基于DataStrem API提供了FlinkCEP组件栈，专门用于对复杂事件的处理，帮助用户从流式数据中发掘有价值的信息。

8.1.1 基础概念

1. 环境准备

在使用FlinkCEP组件之前，需要将FlinkCEP的依赖库引入项目工程中。与FlinkCEP对应的Maven Dependence如下（将如下配置添加到本地Maven项目工程的Pom.xml文件中即可）。

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-cep-scala_2.11</artifactId>
  <version>1.7.2</version>
</dependency>
```

2. 基本概念

（1）事件定义

· 简单事件：简单事件存在于现实场景中，主要的特点为处理单一事件，事件的定义可以直接观察出来，处理过程中无须关注多个事件之间的关系，能够通过简单的数据处理手段将结果计算出来。例如通过对当天的订单总额按照用户维度进行汇总统计，超过一定数量之后进行报告。这种情况只需要计算每个用户每天的订单金额累加值，达到条件进行输出即可。

· 复杂事件：相对于简单事件，复杂事件处理的不仅是单一的事件，也处理由多个事件组成的复合事件。复杂事件处理监测分析事件流(Event Streaming)，当特定事件发生时来触发某些动作。

（2）事件关系

复杂事件中事件与事件之间包含多种类型关系，常见的有时序关系、聚合关系、层次关系、依赖关系及因果关系等。

· 时序关系：动作事件和动作事件之间，动作事件和状态变化事件之间，都存在时间顺序。事件和事件的时序关系决定了大部分的时序规则，例如A事件状态持续为1的同时B事件状态变为0等。

· 聚合关系：动作事件和动作事件之间，状态事件和状态事件之间都存在聚合关系，即个体聚合形成整体集合。例如A事件状态为1的次数为10触发预警。

· 层次关系：动作事件和动作事件之间，状态事件和状态事件之间都存在层次关系，即父类事件和子类事件的层次关系，从父类到子类是具体化的，从子类到父类是泛化的。

· 依赖关系：事物的状态属性之间彼此的依赖关系和约束关系。例如A事件状态触发的条件前提是B事件触发，则A与B事件之间就形成了依赖关系。

· 因果关系：对于完整的动作过程，结果状态为果，初始状态和动作都可以视为原因。例如A事件状态的改变导致了B事件的触发，则A事件就是因，而B事件就是果。

（3）事件处理

复杂事件处理的目的是通过相应的规则对实时数据执行相应的处理策略，这些策略包括了推断、查因、决策、预测等方面的应用。

· 事件推断：主要利用事物状态之间的约束关系，从一部分状态属性值可以推断出另一部分的状态属性值。例如由三角形一个角为90度及另一个角为45度，可以推断出第三个角为45度。

· 事件查因：当出现结果状态，并且知道初始状态，可以查明某个动作是原因；同样当出现结果状态，并且知道之前发生了什么动作，可以查明初始状态是原因。当然反向的推断要求原因对结果来说必须是必要条件。

· 事件决策：想得到某个结果状态，知道初始状态，决定执行什么动作。该过程和规则引擎相似，例如某个规则符合条件后触发行动，然后执行报警等操作。

· 事件预测：该种情况知道事件初始状态，以及将要做的动作，预测未发生的结果状态。例如气象局根据气象相关的数据预测未来的天气情况等。

8.1.2 Pattern API

FlinkCEP中提供了Pattern API用于对输入流数据的复杂事件规则定义，并从事件流中抽取事件结果。如代码清单8-1所示，通过使用Pattern API构建CEP应用程序，其中包括输入事件流的创建，以及Pattern接口的定义，然后通过CEP.pattern方法将定义的Pattern应用在输入的Stream上，最后使用PatternStream.select方法获取触发事件结果。以下实例是将温度大于35度的信号事件抽取出来，并产生事件报警，最后将结果输出到外部数据集中。

代码清单8-1 Pattern接口应用实例

```
//创建输入事件流
val inputStream: DataStream[Event] = ...
//定义Pattern接口
val pattern = Pattern
  .begin[Event]("start")
  .where(_.getType == "temperature")
  .next("middle")
  .subtype(classOf[TempEvent])
  .where(_.getTemp >= 35.0)
  .followedBy("end")
  .where(_.getName == "end")
//将创建好的Pattern应用在输入事件流上
val patternStream = CEP.pattern(inputStream, pattern)
//获取触发事件结果
val result: DataStream[Result] =
patternStream.select(getResult(_))
```

1. 模式定义

个体Pattern可以是单次执行模式，也可以是循环执行模式。单词执行模式一次只接受一个事件，循环执行模式可以接收一个或者多个事件。通常情况下，可以通过指定循环次数将单次执行模式变为循环执行模式。每种模式能够将多个条件组合应用到同一事件之上，条件组合可以通过where方法进行叠加。

个体Pattern都是通过begin方法定义的，例如以下通过Pattern.begin方法定义基于Event事件类型的Pattern，其中<start_pattern>是指定的PatternName象。

```
val start = Pattern.begin[Event]("start_pattern")
```

下一步通过Pattern.where()方法在Pattern上指定Condition，只有当Condition满足之后，当前的Pattern才会接受事件。

```
start.where(event => event.getType == "temperature")
```

（1）指定循环次数

对于已经创建好的Pattern，可以指定循环次数，形成循环执行的Pattern，且有3种方式来指定循环方式。

· times：可以通过times指定固定的循环执行次数。

```
//指定循环触发4次
start.times(4);
//可以执行触发次数范围,让循环执行次数在该范围之内
start.times(2, 4);
```

· optional：也可以通过optional关键字指定要么不触发要么触发指定的次数。

```
start.times(4).optional();
start.times(2, 4).optional();
```

· greedy：可以通过greedy将Pattern标记为贪婪模式，在Pattern匹配成功的前提下，会尽可能多地触发。

```
//触发2、3、4次,尽可能重复执行
start.times(2, 4).greedy();
//触发0、2、3、4次,尽可能重复执行
start.times(2, 4).optional().greedy();
```

· oneOrMore：可以通过oneOrMore方法指定触发一次或多次。

```
// 触发一次或者多次
start.oneOrMore();
//触发一次或者多次,尽可能重复执行
start.oneOrMore().greedy();
// 触发0次或者多次
start.oneOrMore().optional();
// 触发0次或者多次,尽可能重复执行
start.oneOrMore().optional().greedy();
```

· timesOrMor：通过timesOrMore方法可以指定触发固定次数以上，例如执行两次以上。

```
// 触发两次或者多次
start.timesOrMore(2);
// 触发两次或者多次,尽可能重复执行
start.timesOrMore(2).greedy();
// 不触发或者触发两次以上,尽可能重复执行
start.timesOrMore(2).optional().greedy();
```

（2）定义模式条件

每个模式都需要指定触发条件，作为事件进入到该模式是否接受的判断依据，当事件中的数值满足了条件时，便进行下一步操作。在FlinkCFP中通过pattern.where()、pattern.or()及pattern.until()

方法来为Pattern指定条件，且Pattern条件有Iterative Conditions、Simple Conditions及Combining Conditions三种类型。

· 迭代条件：Iterative Conditions能够对前面模式所有接收的事件进行处理，根据接收的事件集合统计出计算指标，并作为本次模式匹配中的条件输入参数。如代码清单8-2所示，通过subtype将Event事件转换为TempEvent，然后在where条件中通过使用ctx.getEventsForPattern(...)方法获取“middle”模式所有接收的Event记录，并基于这些Event数据之上对温度求取平均值，然后判断当前事件的温度是否小于平均值。

代码清单8-2 Pattern中定义迭代条件

```
middle.oneOrMore()
  .subtype(classOf[TempEvent])
  .where(
    (value, ctx) => {
      lazy val avg =
ctx.getEventsForPattern("middle").map(_.getValue).avg
      value.getName.startsWith("condition") && value.getPrice <
avg
        }
    )
```

· 简单条件：Simple Condition继承于Iterative Condition类，其主要根据事件中的字段信息进行判断，决定是否接受该事件。如以下代码将Sensor事件中Type为temperature的事件筛选出来。

```
start.where(event => event.getType == "temperature"))
```

可以通过subtype对事件进行子类类型转换，然后在where方法中针对子类定义模式条件。

```
start.subtype(classOf[TempEvent]).where(tempEvent =>
event.getValue > 10)
```

· 组合条件：组合条件是将简单条件进行合并，通常情况下也可以使用where方法进行条件的组合，默认每个条件通过AND逻辑相连。如果需要使用OR逻辑，如以下代码直接使用or方法连接条件即可。

```
pattern.where(event => event.getName.startWith("foo").or(event =>
event.getType == "temperature")
```

· 终止条件

如果程序中使用了oneOrMore或者oneOrMore().optional()方法，则必须指定终止条件，否则模式中的规则会一直循环下去，如下终止条件通过until()方法指定。

```
pattern.oneOrMore().until(event => event.getName == "end")
```

注意

需要注意的是，在上述迭代条件中通过调用ctx.getEventsForPattern("middle")的过程中，成本相对较高，会产生比较大的性能开销，因此建议用户尽可能少地使用该方式。

2. 模式序列

将相互独立的模式进行组合然后形成模式序列。模式序列基本的编写方式和独立模式一致，各个模式之间通过邻近条件进行连接即可，其中有严格邻近、宽松邻近、非确定宽松邻近三种邻近连接条

件。如以下代码所示，每个Pattern sequence都必须要有begin Pattern。

```
val start : Pattern[Event, _] = Pattern.begin("start")
```

（1）严格邻近

严格邻近条件中，需要所有的事件都按照顺序满足模式条件，不允许忽略任意不满足的模式。如下代码所示，在start Pattern后使用next方法指定下一个Pattern，生成严格邻近的Pattern。

```
val strict: Pattern[Event, _] = start.next("middle").where(...)
```

（2）宽松邻近

在宽松邻近条件下，会忽略没有成功匹配模式条件，并不会像严格邻近要求得那么高，可以简单理解为OR的逻辑关系。如下代码所示稀松邻近条件通过followby方法指定。

```
val relaxed: Pattern[Event, _] =
start.followedBy("middle").where(...)
```

（3）非确定宽松邻近

和宽松邻近条件相比，非确定宽松邻近条件指在模式匹配过程中可以忽略已经匹配的条件。如以下代码实例，非确定宽松邻近条件通过followedByAny方法指定。

```
val nonDetermin: Pattern[Event, _] =
start.followedByAny("middle").where(...)
```

除了上述条件之外，FlinkCEP还提供了notNext()、NotFollowBy()等连接条件，其中notNext()表示不想让某一模式紧跟在另外一个模式之后发生，NotFollowBy()则强调不想让某一模式触发夹在两个模式之间触发。注意模式序列不能以NotFollowBy()结尾，且Not类型的模式不能和optional关键字同时使用。

3. 模式组

模式序列可以作为begin、 followedBy、followedByAny及next等连接条件的输入参数从而形成械组，在GoupPattern上可以指定oneOrMore、times、optional等循环条件，应用在GoupPattern中的模式序列上，每个模式序列完成自己内部的条件匹配，最后在模式组层面对模型序列结果进行汇总。如代码清单8-3所示，首先通过Pattern的begin方法创建start GroupPattern，其包含一个模式序列，然后通过next条件创建严格邻近模式组next，并设定循环的次数为3。

代码清单8-3　模式组代码实例

```
//创建GroupPattern
val start: Pattern[Event, _] =
Pattern.begin(  Pattern.begin[Event]
("start").where(...).followedBy("start_middle").where(...)
)
// 严格邻近模式组（next）
val strict: Pattern[Event, _] = start.next(
Pattern.begin[Event]
("next_start").where(...).followedBy("next_middle").where(...)
).times(3)//指定循环3次
```

4. AfterMatchSkipStrategy

在给定的Pattern中，当同一事件符合多种模式条件组合之后，需要指定AfterMatchSkipStrategy策略以处理已经匹配的事件。在AfterMatchSkipStrategy配置中有四种事件处理策略，分别为NO_SKIP、SKIP_PAST_LAST_EVENT、SKIP_TO_FIRST及SKIP_TO_LAST。每种策略的定义和使用方式如以下说明，其中SKIP_TO_FIRST和SKIP_TO_LAST在定义过程中需要指定有效的PatternName。

· NO_SKIP：该策略表示将所有可能匹配的事件进行输出，不忽略任何一条。

```
AfterMatchSkipStrategy.noSkip()
```

· SKIP_PAST_LAST_EVENT：该策略表示忽略从模式条件开始触发到当前触发Pattern中的所有部分匹配的事件。

```
AfterMatchSkipStrategy.skipPastLastEvent()
```

· SKIP_TO_FIRST：该策略表示忽略第一个匹配指定PatternName的Pattern其之前的部分匹配事件。

```
AfterMatchSkipStrategy.skipToFirst(patternName)
```

· SKIP_TO_LAST：该策略表示忽略最后一个匹配指定PatternName的Pattern之前的部分匹配事件。

```
AfterMatchSkipStrategy.skipToLast(patternName)
```

· SKIP_TO_NEXT：该策略表示忽略指定PatternName的Pattern之后的部分匹配事件。

```
AfterMatchSkipStrategy.skipToNext(patternName)
```

选择完AfterMatchSkipStrategy之后，可以在创建Pattern时，通过begin方法中指定skipStrategy，然后就可以将AfterMatchSkipStrategy应用到当前的Pattern中。

```
val skipStrategy = ...
Pattern.begin("patternName", skipStrategy)
```

8.1.3 事件获取

对于前面已经定义的模式序列或模式组，需要和输入数据流进行结合，才能发现事件中潜在的匹配关系。如以代码实例所示FlinkCEP提供了CEP.pattern方法将DataStream和Pattern应用在一起，得到PatternStream类型的数据集，且后续事件数据获取都基于PatternStream进行。另外可以选择创建EventComparator，对传入Pattern中的事件进行排序，当Event Time相等或者同时到达Pattern时，EventComparator中定义的排序策略可以帮助Pattern判断事件的先后顺序。

```
//创建Input Stream
val inputEvent : DataStream[Event] = ...
//定义Pattern
val pattern = Pattern.begin[Event]("start_pattern").where(...)
//创建EventComparator（可选）
var comparator : EventComparator[Event] = ...
//使用CEP.pattern方法将三者应用在一起,产生PatternStream
val patternStream: PatternStream[Event] = CEP.pattern(inputEvent,
pattern, comparator)
```

当可以CEP.pattern方法被执行后，会生成PatternStream数据集，该数据集中包含了所有的匹配事件。目前在FlinkCEP中提供select和flatSelect两种方法从PatternStream提取事件结果事件。

1.通过Select Funciton抽取正常事件

可以通过在PatternStream的Select方法中传入自定义Select Funciton完成对匹配事件的转换与输出。其中Select Funciton的输入参数为Map[String, Iterable[IN]]，Map中的key为模式序列中的Pattern名称，Value为对应Pattern所接受的事件集合，格式为输入事件的数据类型。需要注意，Select Funciton将会在每次调用后仅输出一条结果。如以下代码通过创建selectFn从PatternStream中分别提取start_pattern和middle对应的Pattern所匹配的事件。

```
def selectFunction(pattern : Map[String, Iterable[IN]]): OUT = {
  //获取pattern中的startEvent
  val startEvent = pattern.get("start_pattern").get.next
    //获取Pattern中middleEvent
    val middleEvent = pattern.get("middle").get.next
    //返回结果
    OUT(startEvent, middleEvent)
}
```

2.通过Flat Select Funciton抽取正常事件

Flat Select Funciton和Select Function相似，不过Flat Select Funciton在每次调用可以返回任意数量的结果。因为Flat Select Funciton使用Collector作为返回结果的容器，可以将需要输出的事件都放置在Collector中返回。如以下代码所示，定义了flatSelectFn根据startEvent中的Value数量返回StartEvent和middleEvent的合并结果。

```
def flatSelectFn(pattern : Map[String, Iterable[IN]], collector :
Collector[OUT]) = {
    //获取pattern中startEvent
  val startEvent = pattern.get("start_pattern").get.next
    //获取Pattern中middleEvent
  val middleEvent = pattern.get("middle").get.next
    //并根据startEvent的Value数量进行返回
  for (i <- 0 to startEvent.getValue) {
    collector.collect(OUT(startEvent, middleEvent))
  }
}
```

此外对于Pattern中的触发事件，如果没有及时处理或者超过了Pattern within关键字设定的时间限制，就会成为超时事件。例如对系统的性能进行排查或者单独处理超时事件，就需要获取超时事件。在Pattern API中提供了select和flatSelect两个方法获取超时事件，和前面获取正常事件的select和flatSelect方法相似，但是这里使用的是Scala的偏函数，函数中定义了超时事件的处理器，该处理器会根据事件的时间判断事件是否发生超时。

3. 通过Select Funciton抽取超时事件

如以下代码通过PatternStream. Select方法分别获取超时事件和正常事件。首先需要创建OutputTag来标记超时事件，然后在PatternStream.select方法中使用OutputTag，就可以将超时事件从PatternStream中抽取出来。

```
  // 通过CEP.pattern方法创建PatternStream
  val patternStream: PatternStream[Event] = CEP.pattern(input,
pattern)
  //创建OutputTag,并命名为timeout-output
  val timeoutTag = OutputTag[String]("timeout-output")
  //调用PatternStream select()并指定timeoutTag
  val result: SingleOutputStreamOperator[NormalEvent] =
  patternStream.select(timeoutTag){
  //超时事件获取
    (pattern: Map[String, Iterable[Event]], timestamp: Long) =>
      TimeoutEvent()//返回异常事件
  } {
  //正常事件获取
    pattern: Map[String, Iterable[Event]] =>
      NormalEvent()//返回正常事件
  }
//调用getSideOutput方法,并指定timeoutTag将超时事件输出
val timeoutResult: DataStream[TimeoutEvent] =
result.getSideOutput(timeoutTag)
```

4. 通过Flat Select Funciton抽取超时事件

如以下代码一通过使用PatternStream的FlatSelect方法。也获取超时事件和正常事件。和Select方法不同，FlatSelect可以在每次返回中获取任意条事件数据。

```
// 通过CEP.pattern方法创建PatternStream
val patternStream: PatternStream[Event] = CEP.pattern(input,
pattern)
//创建OutputTag,并命名为timeout-output
val outputTag = OutputTag[String](" timeout-output")
  //调用PatternStream flatSelect ()并指定timeoutTag
val result: SingleOutputStreamOperator[NormalEvent] =
```

```
  patternStream.flatSelect(timeoutTag){
  //超时事件获取
(pattern:Map[String,Iterable[Event]],timestamp:Long,out:Collector[
TimeoutEvent])=>
    out.collect(TimeoutEvent())
  } {
  //正常事件获取
  (pattern: mutable.Map[String, Iterable[Event]],
out:Collector[NormalEvent]) =>
    out.collect(NormalEvent())
  }
val timeoutResult: DataStream[NormalEvent] =
result.getSideOutput(outputTag)
```

8.1.4 应用实例

基于前面的FlinkCEP复杂事件处理技术，可以通过完整实例对FlinkCEP的使用方式进行整合。如代码清单8-4所示，首先初始化事件流，然后创建Pattern，并对模式设定超时限制，最后调用PatternStream select()方法将符合条件的startEvent事件筛选出来，整个实例就是一个简单的FlinkCEP应用。

代码清单8-4 FlinkCEP使用实例场景

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
//设定时间概念属性
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
//创建输入事件流
val input: DataStream[Event] = env.fromElement(...)
//根据ID分区
val partitionedInput = input.keyBy(event => event.getId)
  //创建Pattern
    val pattern = Pattern.begin[Event]("start")
  .next("middle").where((event, ctx) => event.getType ==
"temperature")
    .followedBy("end").where((event, ctx) => event.getId >= 1000)
  //指定超时限制
    .within(Time.seconds(10))
// 通过CEP.pattern方法创建PatternStream
val patternStream: PatternStream[Event] = CEP.pattern(input,
pattern)
//调用PatternStream select()
val result: DataStream[Event] =
    patternStream.select(event => selectFn(event))
//创建selectFn,将触发的startEvent筛选出来
def selectFn(pattern: Map[String, Iterable[Event]]): Event = {
  //获取pattern中的startEvent
  val startEvent = pattern.get("start").iterator.next().toList(0)
  startEvent
}
```

8.2 Flink Gelly图计算应用

早在2010年，Google就推出了著名的分布式图计算框架Pregel，之后Apache Spark社区也推出GraphX等图计算组件库，以帮助用户有效满足图计算领域的需求。Flink则通过封装DataSet API，形成图计算引擎Flink Gelly。同时Gelly中的Graph API，基本涵盖了从图创建，图转换到图校验等多个方面的图操作接口，让用户能够更加简便高效地开发图计算应用。本节将重点介绍如何通过Flink Gelly组件库来构建图计算应用。

8.2.1 基本概念

1.环境准备

在使用Flink Gelly组件库之前，需要将Flink Gelly依赖库添加到工程中。用户如果使用Maven作为项目管理工具，需要在本地工程的Pom.xml文件中添加如下Maven Dependency配置。

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-gelly_2.11</artifactId>
  <version>1.7.0</version>
</dependency>
```

对于使用Scala语言开发Flink Gelly应用的用户，需要添加如下Maven Dependency配置。

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-gelly-scala_2.11</artifactId>
  <version>1.7.0</version>
</dependency>
```

2.数据结构

图是通过边和顶点构成的，边和顶点在Flink Gelly中基于DataSet数据集构建，一个顶点具有ID和Value两个属性，且顶点ID必须保持唯一同时实现了Comparable接口，Value可以为任何数据类型，也可以通过NullValue表示为空。

```
// 创建顶点
val v = new Vertex(1001, "foo")
// 创建空值顶点
val v = new Vertex(1001, NullValue.getInstance())
```

顶点创建完毕之后，就可以通过边将顶点关联起来，然后形成Graph。在Gelly中，边的结构用Edge数据结构表示，每条边需要指定Source顶点ID和Target顶点ID，以及边的权重Value。

```
//创建边的结构，指定SourceID、TargetID及Value
val edge = new Edge(1L, 2L, 0.5)
//将边进行反转
val reversed = edge.reverse
```

边上的顶点对应的数据类型必须要和创建好的顶点数据类型保持一致，且一条边总是由一个Source顶点指向一个Target顶点，不存在一个Edge中有多个顶点的情况。

8.2.2 Graph API

Graph API则是Gelly构建在DataSet API之上的更高级API，主要功能是降低使用Flink编写图计算应用的成本。在Graph API中提供了非常丰富的图操作接口，例如对图的生成、转换以及修改等，以下分别介绍Graph API中提供的主要图操作接口。

1.Graph创建

1）通过从DataSet数据集中构建Graph

如以下代码首先构建Vertex[String, Long] DataSet数据集以及Edge[String, Double] DataSet数据集，然后通过Graph.fromDataSet()创建Graph。

```
val env = ExecutionEnvironment.getExecutionEnvironment
//构建顶点集合
val vertices: DataSet[Vertex[String, Long]] = ...
//构建边的结合
val edges: DataSet[Edge[String, Double]] = ...
//将边和顶点通过Graph.fromDataSet方法整合,创建Graph
val graph = Graph.fromDataSet(vertices, edges, env)
```

2）通过从Collection中构建Graph

如以下代码所示，可以通过Graph.fromCollection方法，然后使用Vertex和Edge的本地集合来构建Graph。

```
val env = ExecutionEnvironment.getExecutionEnvironment
  //创建顶点
  val vertices: List[Vertex[Long, Long]] = List[Vertex[Long,
Long]](
    new Vertex[Long, Long](6L, 6L))
  //创建边
  val edges: List[Edge[Long, Long]] = List[Edge[Long, Long]](
    new Edge[Long, Long](6L, 1L, 61L))
```

```
  //创建图
  val graph = Graph.fromCollection(vertices, edges, env)
```

3）通过从CSV文件中构建Graph

从CSV文件中构建Graph时，CSV文件中需要包含顶点和边的数据，可以通过在Graph.fromCsvReader方法中分别指定Vertices的路径以及Edges的路径来获取顶点和边的数据。在读取Edges文件时，默认使用文件中第一个字段作为Edges的Source顶点，第二个字段为Target顶点，如果第三个字段作有值，则作为Edges的value，如果为空则填充为NullValue。当然Vertices路径也可以不指定，默认由Edges中的顶点来构建Vertices数据集。

```
//从CSV文件中创建Graph,指定Vertices和Edges路径
  val graph = Graph.fromCsvReader[String, Long, Double](
    pathVertices = "path/vertex_file",
    pathEdges = "path/edge_file",
    env = env)
//从CSV文件中创建Graph,不指定Vertices路径
  val simpleGraph = Graph.fromCsvReader[Long, NullValue,
NullValue](
    pathEdges = "path/vertex_file ",
    env = env)
```

2.Graph转换操作

1）Map

Flink Gelly提供了mapVertices和mapEdges分别用于对Graph中Vertices和Edges的Value进行修改，在执行过程中，Vertice和Edge中的ID不发生变化，Value的转换逻辑在map Function中定义。如下代码所示，通过Vertices DataSet和Edge DataSet创建Graph，然后使用mapVertices方法对Graph中Vertices的Value进行乘10操作，使得Graph中所有的顶点的Value都会比原来大10倍。

```
val env = ExecutionEnvironment.getExecutionEnvironment
val graph = Graph.fromDataSet(vertices, edges, env)
//使用mapVertices方法对Vertices的Value进行转换
val updatedGraph = graph.mapVertices(v => v.getValue * 10)
```

2）Translate

Flink Gelly提供的Translate方法能够对Graph中的ID以及顶点和边的Value进行类型的转换。如下代码所示，通过使用translateGraphIds方法将Graph中的ID类型从Long类型的ID转换为String类型的ID，这样，Graph中所有和边有关的ID类型将全部发生转换，在Graph API中同时提供了translateVertexValues与translateEdgeValues分别对顶点和边的Value进行类型转换。

```
val env = ExecutionEnvironment.getExecutionEnvironment
val graph = Graph.fromDataSet(vertices, edges, env)
// 将Graph中的ID转换为String类型
val updatedGraph = graph.translateGraphIds(id => id.toString)
```

3）Filter

Flink Gelly中提供filterOnEdges和filterOnVertices方法，分别对边和顶点进行筛选操作。其中filterOnEdges方法对边进行过滤，将满足条件的边筛选出来形成子图。filterOnVertices方法根据条件对顶点进行筛选，Edge中的Source顶点或Target顶点有一个不符合条件，则过滤掉对应的Edge，如果形成符合条件的子图。

```
val graph = Graph.fromDataSet(vertices, edges, env)
// 过滤Edge的Value大于10的Graph
graph.filterOnEdges(edge => edge.getValue > 10)
//过滤Vertices的Value小于10的Graph
graph.filterOnVertices(vertex => vertex.getValue < 10)
```

除了可以使用filterOnEdges和filterOnVertices方法分别对Graph的边和顶点进行过滤之外，也可以使用Graph提供的subgraph方法对边和顶点进行过滤，如以下代码所示，将顶点值为正数和边值为负数的子图筛选出来。

```
graph.subgraph((vertex => vertex.getValue > 0), (edge =>
edge.getValue < 0))
```

4）Join

Flink Gelly提供Join方法用于将Vertex或Edge中根据各自的ID与其他数据集关联，形成关联后的Graph。其中joinWithVertices方法支持将Vertices与Tuple2类型的DataSet关联，关联主键是VertexID和Tuple2的第一个字段，结果类型根据函数返回值确定。

JoinWithEdges是将Edge和数据类型为Tuple3的DataSet进行关联，Key是Souce顶点和Target顶点的ID，Tuple3数据集合中包含两个顶点ID对应的Key。而joinWithEdgesOnSource方法通过Source顶点关联Tuple2类型的DataSet数据集，主键为Source Vertex ID和Tuple2集合中第一个字段。同理joinWithEdgesOnTarget方法通过Target顶点和Tuple2类型的DataSet进行关联。

```
val graph: Graph[Long, Long, Long] = ...
val dataSet: DataSet[(Long, Long)] = ...
// 通过joinWithEdgesOnSource和dataSet数据集关联,并指定Source顶点的Value计算函数
val outputResult = graph.joinWithEdgesOnSource(dataSet, (v1: Long,
v2: Long) => v1 + v2)
```

在使用Join图进行操作的过程中，如果关联的DataSet中含有多个相同的key，则Gelly仅会关联第一条符合条件的记录，其他记录则不再进行关联。

5）Reverse

通过Gelly中的reverse()方法能够将Graph中所有的Source顶点和Target顶点进行位置交换，并形成新的Graph。

```
val reverseGraph = graph.reverse()
```

6）Undirected

虽然在Gelly中所有被创建的Graph都是有向的，但Flink Gelly中提供了getUndirected()方法能够将Graph转换成无向图，其原理是增加从Target顶点指向Source顶点的边，从而将有向图转换为无向图。

```
val undirectedGraph = graph. getUndirected()
```

7）Union

Gelly中提供了union()方法对两个图结构完全相同的Graph进行合并。注意在合并的过程中，顶点和边都会进行判断，对于重复的顶点会进行去除，但对于重复的边则会选择保留。

```
val result: Graph[Long, Long, Long] = graph1.union(graph2)
```

8）Difference

Gelly中提供的difference()方法从顶点和边对两个Graph进行对比，然后返回两个Graph中顶点和边均不相同的子图。

```
val result: Graph[Long, Long, Long] = graph1.difference(graph2)
```

9）Intersect

Intersect方法是对两个Graph求取交集，也就是将两个Graph中边完全相同的子图抽取出来，要求每条边的Source ID和Target ID以及Value必须完全一样，然后才会被视为相同结构，方法详见代码清单8-5所示。

代码清单8-5 Gelly Intersect方法

```
// 获取两个Graph交集,并且将distinctEdges设定为true,去除重复的Edges
val intersect = graph1.intersect(graph2, true)
//获取两个Graph交集,并且将distinctEdges设定为false,不去除重复的Edges
val intersect = graph1.intersect(graph2, false)
```

3. 图突变

Gelly提供了对Graph结构进行修改的方法，包括增加和删除边和顶点等，这些操作会导致图的结构发生变化，故被称为图突变。

1）向图中添加单个顶点，如果顶点已经存在，则不会再添加。

```
graph.addVertex(vertex: Vertex[K, VV]
```

2）向图中添加多个顶点，如果顶点已经存在，则不会再添加。

```
graph.addVertices(verticesToAdd: List[Vertex[K, VV]])
```

3）向图中添加单条边，指定Source顶点、Target顶点以及edgeValue，如果图中不存在添加的顶点，边也会被添加到图中。

```
graph.addEdge(source: Vertex[K, VV], target: Vertex[K, VV],
edgeValue: EV)
```

4）向图中添加多条边，如果边已经存在，则不再添加，如果图中不存在被添加的edges的顶点，边也会被添加到图中。

```
graph.addEdges(edges: List[Edge[K, EV]])
```

5）删除图中的顶点，同时删除该顶点的出度和入度对应的边。

```
graph.removeVertex(vertex: Vertex[K, VV])
```

6）删除图中的多个顶点，并且连带删除相关联的边。

```
graph.removeVertices(verticesToBeRemoved: List[Vertex[K, VV]])
```

7）删除图中给定Edge所有匹配的边，如果图中存在多个边被匹配到则全部删除。

```
graph.removeEdge(edge: Edge[K, EV])
```

8）删除图中给定Edges列表中所有匹配到的边，如果图中多个边被匹配到则全部删除。

```
graph.removeEdges(edgesToBeRemoved: List[Edge[K, EV]])
```

4. 邻方法

邻方法（Neighborhood methods）会基于顶点对每个顶点的邻近点或边进行聚合计算，例如reduceOnEdges是对每个顶点的邻近边进行

聚合计算，需要用户实现ReduceEdgesFunction定义聚合逻辑，参数为顶点边的Weight值，并通过EdgeDirection.IN/ OUT多数来指定是对入度还是出度的边进行聚合。reduceOnNeighbors方法是对每个顶点的邻近顶点进行聚合计算，需要用户实现ReduceNeighborsFunction完成聚合计算逻辑的定义，参数为每个邻近顶点的Value。如以下代码所示，通过实现ReduceEdgesFunction实现统计图中每个顶点的所有出度边的最大权重。

```
val graph: Graph[Long, Long, Long] = ...
val maxWeights = graph.reduceOnEdges(new SelectMaxWeightFunction,
EdgeDirection.OUT)
// 定义ReduceEdgesFunction实现对出度边的权重求和
final class SelectMaxWeightFunction extends
ReduceEdgesFunction[Long] {
override def reduceEdges(firstEdgeValue: Long, secondEdgeValue:
Long): Long = {
    Math.max(firstEdgeValue, secondEdgeValue)
  }}
```

如下代码通过实现ReduceNeighborsFunction将每个顶点邻近点的Value进行求和。

```
val verticesWithSum = graph.reduceOnNeighbors(new
SumValuesFucntion,
EdgeDirection.IN)
// 定义ReduceEdgesFunction实现对每个Target顶点邻近点的和
final class SumValuesFucntion extends
ReduceNeighborsFunction[Long] {
  override def reduceNeighbors(firstNeighbor: Long,
secondNeighbor: Long): Long = {firstNeighbor + secondNeighbor}}
```

5. 图校验

在使用创建好的Graph之前，一般需要对Graph的正确性进行检验，从而确保Graph可用。在Gelly中提供了validate()方法可以对Graph进行校验，方法的参数是GraphValidator接口实现类，通过实现GraphValidator中的validate()方法定义对Graph进行逻辑检验。同时

Flink提供了默认的GraphValidator实现类InvalidVertexIdsValidator，能够简单检测Edges中每条边的顶点ID是否全部存在于Vertex集合中，即所有Edge的ID必须都存在于Vertices IDs集合中，实现方式参考以下代码实例。

```
val env = ExecutionEnvironment.getExecutionEnvironment
// 创建顶点集合,IDs = {1, 2, 3}
val vertices: List[Vertex[Long, Long]] = ...
// 创建边的集合: IDs = {(1, 2) (1, 3), (2, 4)}
val edges: List[Edge[Long, Long]] = ...
val graph = Graph.fromCollection(vertices, edges, env)
//检验图是否有效,返回ID为4的节点不合法
graph.validate(new InvalidVertexIdsValidator[Long, Long, Long])
```

8.2.3 迭代图处理

在图计算的过程中伴随着非常多的大规模迭代计算场景，例如求取有向图的最短路径等问题。而在Gelly中针对迭代图分别提供了Vertex-Centric Iterations、Scatter-Gather Iterations、Gather-Sum-Apply Iterations三种迭代计算方式，每种迭代方式中均使用到了不同的策略及优化手段，极大地增强了迭代类图计算应用的效率。

1.Vertex-Centric迭代

Vertex-Centric迭代是以顶点为中心进行迭代，在计算过程中会以顶点为中心分别计算每个顶点上的边或邻近顶点等指标。其中单次迭代的过程被称为Supersteps，在每次迭代中都会并行执行已经定义的Function，同顶点和顶点之间通过Message进行通信，一个顶点可以向另外一个顶点发送信息，前提是这个顶点知道其他顶点的ID。在使用Vertex-Centric迭代计算时，需要实现ComputeFunction接口，定义顶点的计算函数，然后将定义好的ComputeFunction传入Graph中的runVertexCentricIteration方法中，并通过maxIterations参数来指定最大迭代次数。另外也可以定义MessageCombiner方法对Message进行合并，从而降低Message在顶点之间的传输成本。

可通过配置VertexCentricConfiguration参数对Vertex-Centric迭代算法进行优化和调整，其中主要参数如表8-1所示。

表8-1 VertexCentricConfiguration参数配置

参数名称	参数说明	默认值
Name	对于本次迭代计算的名称，可用于在日志或者 messages 中对迭代进行区分	无
Parallelism	迭代计算的并行度，也就是在每次 superstep 中并行计算的线程数量，适度增加可以有效提升计算效率	1
Solution set in unmanaged memory	是否使用非管理内存存储 Solution set 数据集，参数为 True 代表使用，False 代表不使用	False
Aggregators	迭代累加器可以通过 registerAggregator() 进行注册，然后在 Compute Function 中使用，一个迭代累加器可以全局地统计每次 superstep 的指标，然后将结果传输至下一次 superstep	无
Broadcast Variables	可以使用 addBroadcastSet() 方法向 ComputeFunction 中增加广播变量	无

如代码清单8-6所示，通过Vertex-Centric迭代方法，求取Graph中的最短路径。首先通过Collection创建Graph，定义最大迭代次数为10次，并指定初始顶点ID，接着实现ShortestPathComputeFunction完成对ComputeFunction的定义。在第一个superstep中，Source顶点开始向其他顶点计算距离，在下一次迭代中，每个顶点会检查接收到的message，然后和顶点值进行对比，选择最小指标作为当前顶点值，并接着把message传向邻近点。如果顶点在superstep中不再改变顶点值，将不会再给邻近点产生任何message信息，最后直到Graph不再更新顶点值或者达到最大迭代次数，则停止计算并返回结果。同时在代码中定义了message combiner来减少messages的数量，降低superstep之间的传输成本。

代码清单8-6　Vertex-Centric Iterations应用实例

```
val env = ExecutionEnvironment.getExecutionEnvironment
//通过从Collection中创建Graph
val graph: Graph[Long, Long, Long] =
Graph.fromCollection(getLongLongVertices, getLongLongEdges, env)
// 定义最大迭代次数
val maxIterations = 10
```

```
// 执行vertex-centric iteration
val result = graph.runVertexCentricIteration(new
ShortestPathComputeFunction, new ShortestPathMsgCombiner,
maxIterations)
// 提取最短路径,产生结果
val shortestPaths = result.getVertices
//定义起点ID
val srcId = 1L
//定义ComputeFunction
final class ShortestPathComputeFunction extends
ComputeFunction[Long, Long, Long, Long] {
  override def compute(vertex: Vertex[Long, Long], messages:
MessageIterator
[Long]) = {
    var minDistance = if (vertex.getId.equals(srcId)) 0 else
Long.MaxValue
    while (messages.hasNext) {
      val msg = messages.next
      if (msg < minDistance) minDistance = msg
    }
      //如果顶点的Value大于最小值,则重新设定Value,并按照边的方向将Message发
出
    if (vertex.getValue > minDistance) {
      setNewVertexValue(minDistance)
      getEdges.forEach(edge => sendMessageTo(edge.getTarget,
vertex.getValue + edge.getValue))
    }}}
// 定义Message Combiner,合并Message消息
final class ShortestPathMsgCombiner extends MessageCombiner[Long,
Long] {
  override def combineMessages(messages: MessageIterator[Long]) {
    var minDistance = Long.MaxValue
    while (messages.hasNext) {
      val message = messages.next
      if (message < minDistance) {
        minDistance = message
      }}
    sendCombinedMessage(minDistance)
  }
}
```

---

2. Scatter-Gather Iterations

Scatter-Gather迭代也被称为“signal/collect”迭代，共包含了Scatter和Gather两个阶段，Scatter阶段主要是将一个顶点上的Message发送给其他顶点，Gather阶段根据接收到的Message更新顶点

的Value。和Vertex-Centric迭代相似，Scatter-Gather迭代对图中的顶点进行迭代计算，每次迭代计算的过程也被称为supersteps，在supersteps中计算每个顶点为其他顶点生成Message，同时让接收到其他节点的Message更新自己的Value。

在Gelly中通过调用Graph.runScatterGatherIteration()方法应用Scatter-Gather迭代，需要事先定义好对应scatter和gather阶段的数据处理逻辑的ScatterFunction和GatherFunction。ScatterFunction定义了一个顶点发送给其他顶点的Messages逻辑，以及在相同的superstep中接收其他节点发送的Message信息，而GatherFunction定义顶点接收的Message更新其Value的逻辑。最后将定义好的Function和最小迭代次数传递给runScatterGatherIteration()方法，完成整个Scatter-Gather的应用，同时经过多次迭代后，最终形成迭代更新过的Graph。

针对Scatter-Gather迭代抽象参数配置如表8-2所示，用户可以根据情况进行调整。

表8-2　Scatter-Gather迭代抽象参数配置

参数名称	参数说明	默认值
Name	对于本次迭代计算的名称，可用于在日志或者 messages 中对迭代进行区分	无
Parallelism	迭代计算的并行度，也就是在每次 superstep 中并行计算的线程数量，适度增加可以有效提升计算效率	
Solution set in unmanaged memory	是否使用非管理内存存储 Solution Set 数据集，参数为 True 代表使用，False 代表不使用	False
Aggregators	迭代累加器可以通过 registerAggregator() 进行注册，然后在 Compute-Function 中使用，一个迭代累加器可以全局统计每次 superstep 的指标，然后将结果传输至下一次 superstep	无
Broadcast Variables	可以使用 addBroadcastSet() 方法向 ComputeFunction 中增加广播变量	无
Number of Vertices	设定每次迭代中能够接入和操作的顶点数量，可以通过 setOpt-NumVertices() 方法设定参数。对于顶点数量过多的情况，可以适当增大该参数	无
Degrees	定义每次迭代计算顶点的度数，分为入度和出度两种类型。可以通过 setOptDegrees() 设定该参数，每个顶点入度和出度的数量可以通过 getIn-Degree() 和 getOutDegree() 方法获取	无
Messaging Direction	表示 Message 传输的方向。默认情况下，一个顶点发出 Messages 给它的 out-neighbors，根据它的 in-neighbors 更新其指标。Message 的消息传递方向可以调用 graph 的 setDirection() 进行指定，分别包含 EdgeDirection.IN、EdgeDirection.OUT、EdgeDirection.ALL 三种方向类型	

如代码清单8-7所示，通过最短路径实例对Scatter-Gather迭代应用进行说明，首先定义MinDistanceMessenger实现ScatterFunction，以及定义VertexDistanceUpdater实现GatherFunction，然后通过使用Graph API中的runScatterGatherIteration方法完成对Scatter-Gather的应用，最终计算出给定有向图的最短优化路径。

代码清单8-7　Scatter-Gather迭代最短路径算法应用实例代码

```
val env = ExecutionEnvironment.getExecutionEnvironment
val graph: Graph[Long, Long, Long] = ...
val config = new ScatterGatherConfiguration
//定义最大迭代次数
val maxIterations = 10
// 执行scatter-gather iteration
val result = graph.runScatterGatherIteration(new
MinDistanceMessenger, new VertexDistanceUpdater,maxIterations
,config)
// 获取计算结果
val shortestPaths = result.getVertices
// 定义ScatterFunction
final class MinDistanceMessenger extends ScatterFunction[Long,
Long, Long, Long] {
  override def sendMessages(vertex: Vertex[Long, Long]) = {
    for (edge: Edge[Long, Long] <- getEdges) {
      sendMessageTo(edge.getTarget, vertex.getValue +
edge.getValue)
    }
  }
}
// 定义GatherFunction更新顶点指标
final class VertexDistanceUpdater extends GatherFunction[Long,
Long, Long] {
  override def updateVertex(vertex: Vertex[Long, Long], inMessages:
MessageIterator[Long]) = {
    var minDistance = Long.MaxValue
    while (inMessages.hasNext) {
      val msg = inMessages.next
      if (msg < minDistance) {
        minDistance = msg
      }
    }
//如果顶点的Value 大于minDistance,则更新节点Value
    if (vertex.getValue > minDistance) {
      setNewVertexValue(minDistance)
    }}}
```

3.Gather-Sum-Apply Iterations

与Scatter-Gather迭代不同的是，Gather-Sum-Apply迭代中的superstep共包含了三个阶段，分别为Gather、Sum和Apply阶段，其中Gater阶段并行地在每个顶点上执行自定义GatherFunction，计算边或邻近顶点的指标，形成部分结果值。然后进入到Sum阶段，在Sum阶段

将Gather阶段生成的部分结果进行合并，生成单一指标，这步是通过自定义Reducer函数实现。最后Apply阶段，根据Sum阶段生成的结果，判断并更新Vertex上的指标。在整个迭代过程中，三个阶段需要同步在一个superstep中进行，当全部完成才能进入到下一个superstep。同时执行完最大的迭代次数后，完成整个迭代任务，返回新生成的Graph。

针对Scatter-Gather.Apply迭代参数配置如表8-3所示，读者可以根据实际情况进行调整。

表8-3　Scatter-Gather迭代抽象参数配置

参数名称	参数说明	默认值
Name	对于本次迭代计算的名称，在日志或 messages 中对每次迭代进行区分	无
Parallelism	迭代计算的算子并行度，也就是在每次 superstep 中并行计算的线程数量，适度增加可以有效提升计算效率	1
Solution set in unmanaged memory	是否使用非托管内存来存放 Solution Set 数据集，参数为 True 代表使用，False 代表不使用	False
Aggregators	迭代累加器，通过 registerAggregator() 进行注册，然后用在 ComputeFunction 中。迭代累加器可以全局统计每次 superstep 的指标，并将结果传输至下一次 superstep	无
Broadcast Variables	使用 addBroadcastSet() 方法向 ComputeFunction 中增加广播变量	无
Number of Vertices	设定每次迭代中能够接入的顶点数量，可通过 setOptNumVertices() 方法设定参数。对于顶点数量过多的情况，可适当增大该参数	-1
Neighbor Direction	默认情况下聚合每个顶点的出度节点，可以通过使用 setDirection() 方法进行设定	EdgeDirection.OUT

如代码清单8-8所示，通过最短路径实例对Scatter-Gather-Apply迭代的应用进行说明，在应用程序中需要定义三个函数，分别为GatherFunction、SumFunction、ApplyFunction，其中GatherFunction用于计算顶点邻近点的距离，生成初步的距离值，然后通过SumFunction对GatherFunction产生的结果进行聚合，产生最小距离值，最后通过ApplyFunction使用SumFunction产生的最小距离值判断并更新顶点Value值。当顶点Value不再发生变化或者达到最大迭代次数时，终止计算并输出结果。

代码清单8-8 Gather-Sum-Apply迭代实例

```
val env = ExecutionEnvironment.getExecutionEnvironment
// 初始化Graph
val graph: Graph[Long, Long, Long] =
Graph.fromCollection(getLongLongVertices, getLongLongEdges, env)
// 定义GSAConfiguration
val config = new GSAConfiguration
// 定义最大迭代次数
val maxIterations = 10
// 执行 GSA iteration
val result = graph.runGatherSumApplyIteration(new
CalculateDistancesFun, new minDistanceFun, new UpdateDistanceFun,
maxIterations,config)
// 抽取执行结果
val singleSourceShortestPaths = result.getVertices
// 定义GatherFunction,计算邻近点的距离
final class CalculateDistancesFun extends GatherFunction[Long,
Long, Long] {
  override def gather(neighbor: Neighbor[Long, Long]): Long = {
    neighbor.getNeighborValue + neighbor.getEdgeValue
  }
}
// 定义SumFunction,选择最小的距离
final class minDistanceFun extends SumFunction[Long, Long, Long] {
  override def sum(newValue: Long, currentValue: Long): Long = {
    Math.min(newValue, currentValue)}}
// 定义ApplyFunction,更新顶点上的Value
final class UpdateDistanceFun extends ApplyFunction[Long, Long,
Long] {
  override def apply(newDistance: Long, oldDistance: Long) = {
    if (newDistance < oldDistance) {
      setResult(newDistance)
    }}}
```

8.2.4 图生成器

现实场景中存在着不同类型的图，例如循环图、完整图、回声图等。而在Gelly中定义了非常丰富的图生成器，可以用其创建不同类型的图，以下简单列举几种常见图的创建方法。

1）循环图

Cycle Graph中所有顶点只与其相邻顶点相连，循环图所有的边组成一个唯一的循环且边上不具有方向。在Gelly中可以通过CirculantGraph类创建循环图，同时可以指定顶点的数量，Gelly将根据用户指定顶点数量创建相应的循环图。

```
val vertexCount = 6;
val graph = new CirculantGraph(env.getJavaEnv, 5).addRange(1,
2).generate()
```

2）完整图

完整图中每个顶点与其他顶点都由无向边相连，也就是所有顶点和顶点之间两两相互连接。在Gelly中可以通过CompleteGraph类创建完整图，构造器第二个参数为顶点数量，如下代码中创建6个顶点的完整图。

```
val vertexCount = 6
val graph = new CompleteGraph(env.getJavaEnv,
vertexCount).generate()
```

3）空图

空图只有顶点，没有边，最终呈现形式为多个独立的点。可以通过使用EmptyGraph类创建空图，同时构造器中第二个参数代表顶点数量。

```
val vertexCount = 6
val graph = new EmptyGraph(env.getJavaEnv, vertexCount).generate()
```

4）Star Graph

星图中含有一个中心顶点能够连接到所有的子顶点，且所有的边都是无向的。可以通过StarGraph类创建星图，同样需要指定Graph的顶点数量。

```
val vertexCount = 6
val graph = new StarGraph(env.getJavaEnv, vertexCount).generate();
```

除了提供以上的内置的图生成器之外，Flink Gelly中还提供了其他类型图的创建方式，例如有回声图、超立方体图、路径图、RMat图等，具体的创建方式用户可以参考官方文档。

8.3 FlinkML机器学习应用

机器学习是一门多领域交叉学科，涵盖设计概率论、统计学、逼近轮等多门学科。机器学习主要是用来设计和分析一些可以让机器自动学习的算法，这些算法是从数据中自动分析获取规律，并利用已经学习的规律对未知数据进行预测，产生预测结果。而机器学习共分为几大种类型，分别是监督学习、半监督学习、无监督学习以及强化学习，每种学习类型具有相应的算法集。例如对于无监督学习来说，对应的是聚类算法，有监督学习则对应的是分类与回归算法等。

和其他分布式处理框架一样，Flink基于DataSet API之上，封装出针对机器学习领域的组件栈FlinkML。在FlinkML中提供了诸如分类、聚类、推荐等常用的算法，用户可以直接使用这些算法构建相应的机器学习应用。虽然目前FlinkML中集成的算法相对较少，但是Flink社区会在未来的版本中陆续集成更多的算法。

8.3.1　基本概念

1.环境准备

FlinkML组件库作为Flink的应用库，并没有直接集成在集群环境中，因此需要在工程中单独引入对应的Maven依赖配置，可以将代码清单8-9的配置引入到工程Pom.xml中。

代码清单8-9　FlinkML依赖库Maven环境配置

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-ml_2.11</artifactId>
  <version>1.7.0</version>
</dependency>
```

2.数据结构

和SparkMLib一样，FlinkML也是通过使用Breeze库实现了底层的向量和矩阵等数据结构，用以辅助构建算法模型，其中Breeze库提供了Vector/Matrix的实现以及对应的计算接口。在FlinkML中则使用了自己定义的Vector和Matrix，只是在具体的计算过程中通过转换为Breeze的形式进行运算。

在FlinkML中可以通过两种方式来构建Vector数据，分别是读取LibSVM数据或读取文件数据转换成DataSet[String]数据集，然后再通过Map算子将数据集转换为DataSet[LabelVector]数据集。

（1）通过读取LibSVM数据

FlinkML中提供了MLUtils类的readLibSVM方法，用于读取LIBSVM格式类型的数据，同时提供writeLibSVM方法将Flink中的数据以LIBSVM格式输出到外部文件系统中。

```
import org.apache.flink.ml.MLUtils
//读取LibSVM数据
val trainData: DataSet[LabeledVector] = MLUtils.readLibSVM(env,
"/path/svmfile1")
val predictData: DataSet[LabeledVector] = MLUtils.readLibSVM(env,
"/path/svmfile2")
//写出LibSVM数据
val svmData: DataSet[LabeledVector] = ...
val dataSink:DataSink[String] = MLUtils.
writeLibSVM("/path/svmfile2", svmData)
```

（2）读取CSV文件转换

可以通过DataSet API中提供读取文件数据的方式，将外部数据读取到Flink系统中，并通过Map函数将数据集转换成LabelVector类型。如以下代码通过readCsvFile方法将数据集读取进来并转换为LabeledVector数据结构。

```
val trainCsvData = env.readCsvFile[(String, String, String,
String)]("/path/svm_train.data")
val trainData:DataSet[LabeledVector] = trainCsvData.map { data =>
  val numList =
data.productIterator.toList.map(_.asInstanceOf[String].toDouble)
  LabeledVector(numList(3), DenseVector(numList.take(3).toArray))
}
```

8.3.2　有监督学习算子

目前在Flink有监督算子分类算法中，FlinkML已经支持了SVM算法、多元线性回归算法以及K-Nearest算法等常用分类算法。

1.SVM算法

在机器学习中，支持向量机（SVM）是在分类与回归分析中分析数据的监督式学习模型。SVM模型是将实例表示为空间中的点，通过映射使得单独类别的实例被尽可能宽的明显间隔分开，然后将新的实例映射到同一空间中，并基于它们落在间隔的哪一侧来预测所属类别。除了进行线性分类外，SVM还可以使用所谓的核技巧有效地进行非线性分类，并将其输入隐式映射到高维特征空间中。

在Flink中SVM是一个预测函数，含有fit和predict两个方法，其中fit方法是基于训练数据集进行转化和训练，生成SVM模型；而Predict方法使用已经训练好的模型进行预测，并生成LabelVector类型预测结果。

1）参数配置

针对SVM模型中参数配置如表8-4所示，用户可以根据情况自行调整和优化。

表8-4　SVM参数汇总

参数名称	参数说明	默认值
Blocks	设定接入的数据集将要被切分的 Block 数量，每块数据集上本地执行随机双坐标上升方法，且 Blocks 数量至少要和并行度一样	None
Iterations	最大迭代数量，超过该值则终止迭代	10
LocalIterations	定义随机双坐标上升（SDCA）最大的迭代次数	10
Regularization	定义 SVM 算法正则化参数	1.0
Stepsize	定义更新权重向量的初始化步长，步长越长，则本次权重向量每次更新到下一个权重向量的贡献度将越多	1.0
ThresholdValue	定义正负标签确定函数的阈值，大于该值则认为是正标签，小于该值则认为是负标签	0.0
OutputDecisionFunction	确定预测和评估函数中应该返回的实例类型，设定为 True 返回超平面的原生实例类型，设定为 False 则返回 Lable 标签信息	False
Seed	初始化随机数的种子	随机值

2）应用举例

如代码清单8-10所示，通过给定训练和测试数据集，使用SVM算法构建分类模型并使predict进行预测。

代码清单8-10　SVM算法实例

```
val env = ExecutionEnvironment.getExecutionEnvironment
//指定训练数据集和测试数据集
val trainLibSvmFile: String = ...
val testLibSvmFile: String = ...
// 读取训练LibSVM数据集
val trainingDS: DataSet[LabeledVector] =
env.readLibSVM(trainLibSvmFile)
// 创建SVM算子,并制定Blocks数量为10
val svm = SVM().setBlocks(10)
// 训练SVM模型
```

```
svm.fit(trainingDS)
// 读取SVM测试数据集,对模型进行评估
val testingDS= env.readLibSVM(pathToTestingFile).map(_.vector)
// 通过predict方法对测试数据集进行预测,产生预测结果
val predictionDS: DataSet[(Vector, Double)] =
svm.predict(testingDS)
```

2. 多元线性回归

多元线性回归模型主要用回归方程描述一个因变量与多个自变量的依存关系。和其他预测算法一样，在Flink ML中多元线性回归算法中包含fit和predict两个方法，分别对用户训练模型和基于模型进行预测，其中fit方法传入的是LablePoint数据结构的dataset数据集，并返回模型结果。predict方法中可以有所有Vector接口的实现类，方法结果返回的是含有输入参数和Double类型的打分结果。

1）参数配置

对于多元线性回归参数配置见表8-5所示，用户可根据实际情况进行调整。

表8-5　多元线性回归参数

参数名称	参数说明	默认值
Iterations	该参数代表最大迭代次数，超过该值则终止迭代	10
Stepsize	对于梯度下降算法初始步长，该参数决定在梯度计算过程中向前推移的大小，Stepsize 越大则梯度下降的速度越快，但是也有可能不稳定，用户需要根据实际情况进行调整	S
ConvergenceThreshold	收敛阈值，定义直到迭代终止，残差平方和相对变化的阈值	Node
LearningRateMethod	学习率方法，FlinkML 中提供了常用的学习率方法，每种学习率方法通过 LearningRateMethod 类进行引用	Default

2）应用举例

如代码清单8-11所示，基于给定的训练和测试数据集，通过使用多元线性回归完成模型的构建和预测。

代码清单8-11 Multiple Linear Regression实例

```
// 创建多元线性回归学习器
val mlr = MultipleLinearRegression()
.setIterations(10)
.setStepsize(0.5)
.setConvergenceThreshold(0.001)
// 创建训练集和测试集
val trainingDS: DataSet[LabeledVector] = ...
val testingDS: DataSet[Vector] = ...
// 将定义好的模型适配到数据集上进行模型训练
mlr.fit(trainingDS)
// 使用测试数据集进行模型预测，产生预测结果
val predictions = mlr.predict(testingDS)
```

8.3.3　数据预处理

在FlinkML中实现了基本的数据预处理方法，其中包括多项式特征、标准化、区间化等常用方法，这些算子都继承于Transformer接口，并使用fit方法从训练集学习模型（例如，归一化的平均值和标准偏差）。

1. 多项式特征

特征加工的过程中，通过增加一些输入数据的非线性特征来增加模型的复杂度通常是非常有效的，可以获得特征的更高维度和相互间的关系项。当多项式特征比较少的时候，可以对很少的特征进行多项式变化，产生更多的特征。多项式变换就是把现有的特征排列组合相乘，例如如果是degree为2的变换，则会把现有的特征中，抽取两个相乘，并获得所有组合的结果。

1）参数配置

在多项式变换调优的参数中主要包括Degree，表示多项式最大的维度，默认值为10。

2）应用实例

如代码清单8-12所示，使用Flink中自带的PolynomialFeatures方法对给定训练数据集进行处理。

代码清单8-12　多项式特征抽取

```
val env = ExecutionEnvironment.getExecutionEnvironment
  // 获取训练数据集
  val trainingDS: DataSet[LabeledVector] =
env.fromElements(LabeledVector(1.2,Vector()))
  // 设定多项式转换维度为3
  val polyFeatures = PolynomialFeatures()
    .setDegree(3)
  //使用PolynomialFeatures进行特征转换
  polyFeatures.fit(trainingDS)
```

2. 标准化

标准化处理函数的主要目的是根据用户统计出来的均值和方差对数据集进行标准化处理，防止因为度量单位不同而导致计算出现的偏差。标准化处理函数通过使用均值来对某个特征进行中心化，然后除以非常量特征（non-constant features）的标准差进行缩放。

例如，在机器学习算法的目标函数（比如SVM的RBF内核或线性模型的L1和L2正则化），许多学习算法中目标函数的基础都是假设所有的特征都是零均值并且具有同一阶数上的方差。另外如果某个特征的方差比其他特征大几个数量级，其就会在学习算法中占据主导位置，导致学习器并不能像期望的那样，从其他特征中学习并产出模型。

FlinkML中提供了StandardScaler类帮助用户对数据进行标准化处理，其中包含fit和transform两个方法，其中fit方法通过基于给定数据集中学习平均值和标准偏差fit方法定义如下：

```
fit[T <: Vector]: DataSet[T] => Unit
fit: DataSet[LabeledVector] => Unit
```

transform方法的定义如下，其主要目的是完成对数据集标准化处理。

```
transform[T <: Vector]: DataSet[T] => DataSet[T]
transform: DataSet[LabeledVector] => DataSet[LabeledVector]
```

1）标准化处理的StandardScaler类主要包含2个参数：其中Mean表示缩放数据集的期望值，默认值为0.0；Std表示缩放数据集的标准偏差，默认值为1.0。

2）应用实例

如下代码所示，通过使用StandardScaler函数对给定数据集进行标准化处理。

```
// 创建标准化函数,并设定平均值为5.0,标准偏差为2.0
val scaler = StandardScaler()
.setMean(5.0)
.setStd(2.0)
// 读取数据集
val dataSet: DataSet[Vector] = ...
// 从训练数据集中学习平均值和标准偏差值
scaler.fit(dataSet)
// 对给定数据集进行缩放,使其平均值mean=10.0标准偏差std=2.0
val result = scaler.transform(dataSet)
```

3. 区间缩放

区间缩放是将某一列向量根据最大值和最小值进行区间缩放，将指标转换到指定范围的区间内，从而尽可能地使特征的度量标准保持一致，避免因为某些指标比较大的特征在训练模型过程中占有太大的权重，影响整个模型的效果。

FlinkML中通过MinMaxScaler类实现区间缩放功能，其也实现了Transformer接口，包含了fit方法和transform方法，MinMaxScaler通过fit方法进行训练，训练数据可以是Vector的子类型或者是LabeledVector类型。然后通过transform方法对数据集进行区间缩放操作，并产生新的区间缩放后的数据集。

1）fit方法定义

```
fit[T <: Vector]: DataSet[T] => Unit
fit: DataSet[LabeledVector] => Unit
```

2）transform方法定义

```
transform[T <: Vector]: DataSet[T] => DataSet[T]
transform: DataSet[LabeledVector] => DataSet[LabeledVector]
```

3）参数配置

区间缩放功能的MinMaxScaler类主要包括2个参数：其中Min参数表示数据集数值范围中的最小值，默认值为0.0；Max参数表示数据集数值范围中的最大值，默认值为1.0。

4）应用实例

通过代码清单8-13能够看到，在应用MinMaxScaler进行区间缩放的时候，需要实例化MinMaxScaler类，并通过setMin()和setMax()设定数据集区间大小范围，然后调用fit方法进行训练，最后调用transform方法将接入数据集将标准化转到执行区间范围内，形成缩放后的数据集。

代码清单8-13　MaxMinScaler数据处理实例

```
// 创建MinMax缩放器
val minMaxscaler = MinMaxScaler()
  .setMin(-1.0)
// 创建DataSet输入数据集
val dataSet: DataSet[Vector] = ...
// 学习给定数据集中的最大值和最小值
minMaxscaler.fit(dataSet)
// 将给定的数据集转换成-1到1之间的集合
val scaledDS = minMaxscaler.transform(dataSet)
```

8.3.4 推荐算法

Alternating Least Squares (ALS)

ALS算法也被称为为交替最小二乘算法，是目前业界使用相对广泛协同过滤算法。ALS算法通过观察用户对商品的打分，来判断每个用户的喜好并向用户推荐合适的商品。从协同过滤的角度分析，ALS算法属于User-Item CF，即同时考虑了User和Item两个方面的内容，用户和商品的数据，可以抽象成三元组<User,Item,Rating>。

目前ALS算法已经集成到FlinkML库中，ALS是一个Predictor函数类，具有fit和predict方法。

1）参数配置

ALS算法参数配置如表8-6所示，用户可以根据实际情况进行优化和调整。

表8-6 ALS算法参数配置

参数名称	参数说明	默认值
NumFactors	指定计算 User 和 Item 向量的维度	10
Lambada	规则化参数，通过调整该参数可以有效避免过拟合问题或者过泛化问题	1
Iterations	最大迭代次数	10
Blocks	User 和 Item 矩阵分组的 Blocks 数量。越少的 Block 数量，意味着越少的数据冗余传输，但是 Block 太大会带来内存溢出的风险，需要根据情况进行调整	None
Seed	随机种子用于初始化 Item 矩阵	0
TemporaryPath	用于存放中间计算结果的路径	None

2）应用实例

如代码清单8-14所示，基于给定数据集，通过使用ALS算法构建推荐模型并基于模型进行结果预测。

代码清单8-14　ALS算法应用实例

```
// 读取训练数据集
val trainingDS: DataSet[(Int, Int, Double)] = ...
// 读取测试数据集
val testingDS: DataSet[(Int, Int)] = ...
// 设定ALS学习器
val als = ALS()
.setIterations(100)
.setNumFactors(10)
.setBlocks(100)
.setTemporaryPath("hdfs://temporary/Path")
// 通过ParameterMap计算额外参数
val parameters = ParameterMap()
.add(ALS.Lambda, 0.9)
.add(ALS.Seed, 42L)
// 计算隐式分解
als.fit(trainingDS, parameters)
// 根据测试数据集,计算推荐结果
val result = als.predict(testingDS)
```

8.3.5 Pipelines In FlinkML

机器学习已经被成功应用到多个领域，如智能推荐、自然语言处理、模式识别等。但是不管是什么类型的机器学习应用，其实都基本遵循着相似的流程，包括数据源接入、数据预处理、特征抽取、模型训练、模型预估、模型可视化等步骤。如果能够将这些步骤有效连接并形成流水线式的数据处理模式，将极大地提升机器学习应用构建的效率。受Scikit-Learn开源项目的启发，在FlinkML库中提供了Pipelines的设计，将数据处理和模型预测算子进行连接，以提升整体机器学习应用的构建效率。同时，FlinkML中的算法基本上都实现于Transformers和Predictors接口，主要目的就是为了能够提供一整套的ML Pipelines，帮助用户构建复杂且高效的机器学习应用。

如代码清单8-15所示，将PolynomialFeatures算子和MultipleLinearRegression预测算子通过Pipelines整合在一起，完成复杂的机器学习应用构建。

代码清单8-15 Flink ML Pipelines构建实例

```
// 获取训练数据集
val trainingDS: DataSet[LabeledVector] = ...
// 获取测试数据集,无Label标签
val testingDS: DataSet[Vector] = ...
val polyFeatures = PolynomialFeatures()
val mlr = MultipleLinearRegression()
// 构建Pipelines,将polyFeatures和mlr进行连接
val pipeline = polyFeatures
  .chainPredictor(mlr)
// 将创建的Pipeline应用到训练数据集上
pipeline.fit(trainingDS)
// 对测试数据集进行预测打分,产生预测结果
val result: DataSet[LabeledVector] = pipeline.predict(testingDS)
```

除了可以使用FlinkML中已经定义的Transformers和Predictors之外，用户也可以自定义算子并应用在Pipelines中，完成整个机器学习任务链路的构建。FlinkML中实现对数据的转换操作的接口为Transformer，用于模型训练及预测的接口为Predictor。在整个

Pipelines中，Transformer类型算子后面可以接其他算子，而Predictor类型算子后面则不能接入任何类型算子，也就是说，Predictor算子是整个Pipelines的终点。

Estimator接口是Transformer接口和Predictor接口的父类，如代码清单8-16所示，在Estimator中定义了fit方法，主要负责调用具体的算法实现逻辑，该方法中含有两个参数，分别为传入的训练数据集和Estimator所用到的参数。在FitOperation中定义了fit方法具体计算逻辑，Estimator中的fit方法借由包装类将计算逻辑通过调用隐式方法转换到FitOperation中的fit方法。同理Transformers类和Predictors类也是通过这种方式进行，因此用户在定义相关实现逻辑时需要有Scala隐式转换相关的知识。

代码清单8-16 Estimator接口定义

```
trait Estimator[Self] extends WithParameters with Serializable {
  that: Self =>
  def fit[Training](
      training: DataSet[Training],
      fitParameters: ParameterMap = ParameterMap.Empty)
      (implicit fitOperation: FitOperation[Self, Training]): Unit
= {

FlinkMLTools.registerFlinkMLTypes(training.getExecutionEnvironment
)
    fitOperation.fit(this, fitParameters, training)
  }
}
```

8.4 本章小结

本章介绍了Flink在复杂事件处理（CEP）、图计算、机器学习等不同应用领域的组件库。8.1节介绍了如何使用FlinkCEP组件库解决复杂事件处理的场景问题，包括对简单事件和复杂事件的定义，以及Flink中Patten接口的使用8.2节介绍了Flink中专门用于解决图计算问题的组件库Gelly，其中包括使用GraphAPI对图数据结构的创建和转换等常规操作，同时还有一些比较高级的图处理迭代算法，例如顶点中心迭代等，能够帮助用户更加高效地处理图数据。同时也介绍了Flink Gelly中提供的常见的图的生成及对应的算法。8.3节重点介绍了FlinML在机器学习领域的应用，包括有监督学习和无监督学习算法，以及常用的数据处理方法，Pipelines数据流水线处理模式等。

第9章

Flink部署与应用

本章首先会重点讲解如何将前面章节中不同API编写的Flink应用部署在Flink实际集群环境，以及Flink所支持的不同部署环境与模式，其中包括Standalone cluster，Flink On Yarn以及Flink On Kubernetes三种部署模式。然后介绍Flink在集群环境中如何进行高可用等配置，其中重点包括如何实现JobManager的高可用。除此之外，本章也将介绍Flink集群安全认证管理，帮助用户进行整个Flink集群的安全管理，实现与大数据集群认证系统对接。最后还将介绍Flink集群在升级运维过程中，如何通过Savepoint技术实现数据一致性的保障。

9.1 Flink集群部署

作为通用的分布式数据处理框架，Flink可以基于Standalone集群进行分布式计算，也可以借助于第三方组件提供的计算资源完成分布式计算。目前比较熟知的资源管理器有Hadoop Yarn、Apache Mesos、Kubernetes等，目前Flink已经能够非常良好地支持这些资源管理器。以下将分别介绍Flink基于Standalone Cluster、Yarn Cluster、Kubunetes Cluster等资源管理器上的部署与应用。

9.1.1 Standalone Cluster部署

前面已经知道Flink是Master-Slave架构的分布式处理框架，因此构建Standalone Cluster需要同时配置这两种服务角色，其中Master角色对应的是JobManager，Slave角色对应的是TaskManager。在构建Flink Standalone集群之前，以下服务器基础环境参数必须要符合要求。

1.环境要求

首先Flink集群需要构建在Unix系统之上，例如Linux和Mac OS等系统，如果是Windows系统则需要安装Cygwin构建Linux执行环境，不同的网络和硬件环境需要不同的环境配置。

· JDK环境

其次需要在每台主机上安装JDK，且版本需要保持在1.8及以上，可以设定环境变量JAVA_HOME，也可以在配置文件中指定JDK安装路径，Flink配置文件在安装路径中的conf目录下，修改flink-conf.yaml中的env.java.home参数，指定Java安装路径即可。

· SSH环境

最后需要在集群节点之间配置互信，可以使用SSH对其他节点进行免密登录，因为Flink需要通过提供的脚本对其他节点进行远程管理和监控，同时也需要每台节点的安装路径结构保持一致。

2.集群安装

Flink Standalone集群安装相对比较简单，只需要两步就能够安装成功，具体安装步骤如下：

（1）下载Flink对应版本安装包

如果环境能连接到外部网络，可以直接通过wget命令下载，或在官方镜像仓库中下载，然后上传到系统中的安装路径中，官方安装包下载地址为https://flink.apache.org/downloads.html。注意，如果

仅是安装Standalone集群，下载Apache Flink x.x. only对应的包即可；如果需要Hadoop的环境支持，则下载相应Hadoop对应的安装包，这里建议用户使用基于Hadoop编译的安装包。可以通过如下命令从官网镜像仓库下载安装包：

```
wget http://mirrors.tuna.tsinghua.edu.cn/apache/flink/flink-1.7.2/flink-1.7.0-bin-hadoop27-scala_2.11.tgz
```

（2）安装Flink Standalone Cluster

如下代码所示，通过tar命令解压Flink安装包，并进入到Flink安装路径。至此，Flink Standalone Cluster就基本安装完毕。

```
tar -xvf flink-1.7.0-bin-hadoop27-scala_2.11.tgz
```

进入到Flink安装目录后，所有Flink的配置文件均在conf路径中，启动脚本在bin路径中，依赖包都放置在lib路径中。

3. 集群配置

安装完Standalone集群之后，下一步就需要对Flink集群的参数进行配置，配置文件都在Flink根目录中的conf路径下。

首先配置集群的Master和Slave节点，在master文件中配置Flink JobManager的hostname以及端口。然后在conf/slaves文件中对Slave节点进行配置，如果需要将多个TaskManager添加在Flink Standalone集群中，如以下只需在conf/slaves文件中添加对应的IP地址即可。

```
10.0.0.1
10.0.0.2
```

通过配置以上参数，将10.0.0.1和10.0.0.2节点添加到Flink集群之中，且每台节点的环境参数必须保持一致且符合Flink集群要求，否则会出现集群无法启动的问题。

同时在conf/flink-conf.yaml文件中可以配置Flink集群中JobManager以及TaskManager组件的优化配置项，主要的配置项如以下所示：

· jobmanager.rpc.address

表示Flink Cluster集群的JobManager RPC通信地址，一般需要配置指定的JobManager的IP地址，默认localhost不适合多节点集群模式。

· jobmanager.heap.mb

对JobManager的JVM堆内存大小进行配置，默认为1024M，可以根据集群规模适当增加。

· taskmanager.heap.mb

对TaskManager的JVM堆内存大小进行配置，默认为1024M，可根据数据计算规模以及状态大小进行调整。

· taskmanager.numberOfTaskSlots

配置每个TaskManager能够贡献出来的Slot数量，根据TaskManager所在的机器能够提供给Flink的CPU数量决定。

· parallelism.default

Flink任务默认并行度，与整个集群的CPU数量有关，增加parallelism可以提高任务并行的计算的实例数，提升数据处理效率，但也会占用更多Slot。

· taskmanager.tmp.dirs

集群临时文件夹地址，Flink会将中间计算数据放置在相应路径中。

这些默认配置项会在集群启动的时候加载到Flink集群中，当用户提交任务时，可以通过-D符号来动态设定系统参数，此时flink-conf.yaml配置文件中的参数就会被覆盖掉，例如使用-Dfs.overwrite-files=true动态参数。

4.启动Flink Standalone Cluster

Flink Standalone集群配置完成后，然后在Master节点，通过bin/start-cluster.sh脚本直接启动Flink集群，Flink会自动通过Scp的方式将安装包和配置同步到Slaves节点。启动过程中如果未出现异常，就表示Flink Standalone Cluster启动成功，可以通过https://{JopManagerHost:Port} 访问Flink集群并提交任务，其中JopManagerHost和Port是前面配置JopManager的IP和端口。

5.动态添加JobManager&TaskManager

对于已经启动的集群，可以动态地添加或删除集群中的JobManager和TaskManager，该操作通过在集群节点上执行jobmanager.sh和taskmanager.sh脚本来完成。

· 向集群中动态添加或删除JobManager

```
bin/jobmanager.sh ((start|start-foreground) [host] [webui-port])|stop|stop-all
```

· 向集群中动态添加TaskManager

```
bin/taskmanager.sh start|start-foreground|stop|stop-all
```

9.1.2 Yarn Cluster部署

Hadoop Yarn是 Hadoop 2.x提出的统一资源管理器，也是目前相对比较流行的大数据资源管理平台，Spark和MapReduce等分布式处理框架都可以兼容并运行在Yarn上，Flink也是如此。

需要注意Flink应用提交到Yarn上目前支持两种模式，一种是Yarn Session Model，这种模式中Flink会向Hadoop Yarn申请足够多的资源，并在Yarn上启动长时间运行的Flink Session集群，用户可以通过RestAPI或Web页面将Flink任务提交到Flink Session集群上运行。另外一种为Single Job Model和大多数计算框架的使用方式类似，每个Flink任务单独向Yarn提交一个Application，并且每个任务都有自己的JobManager和TaskManager，当任务结束后对应的组件也会随任务释放，运行流程如图9-1所示。

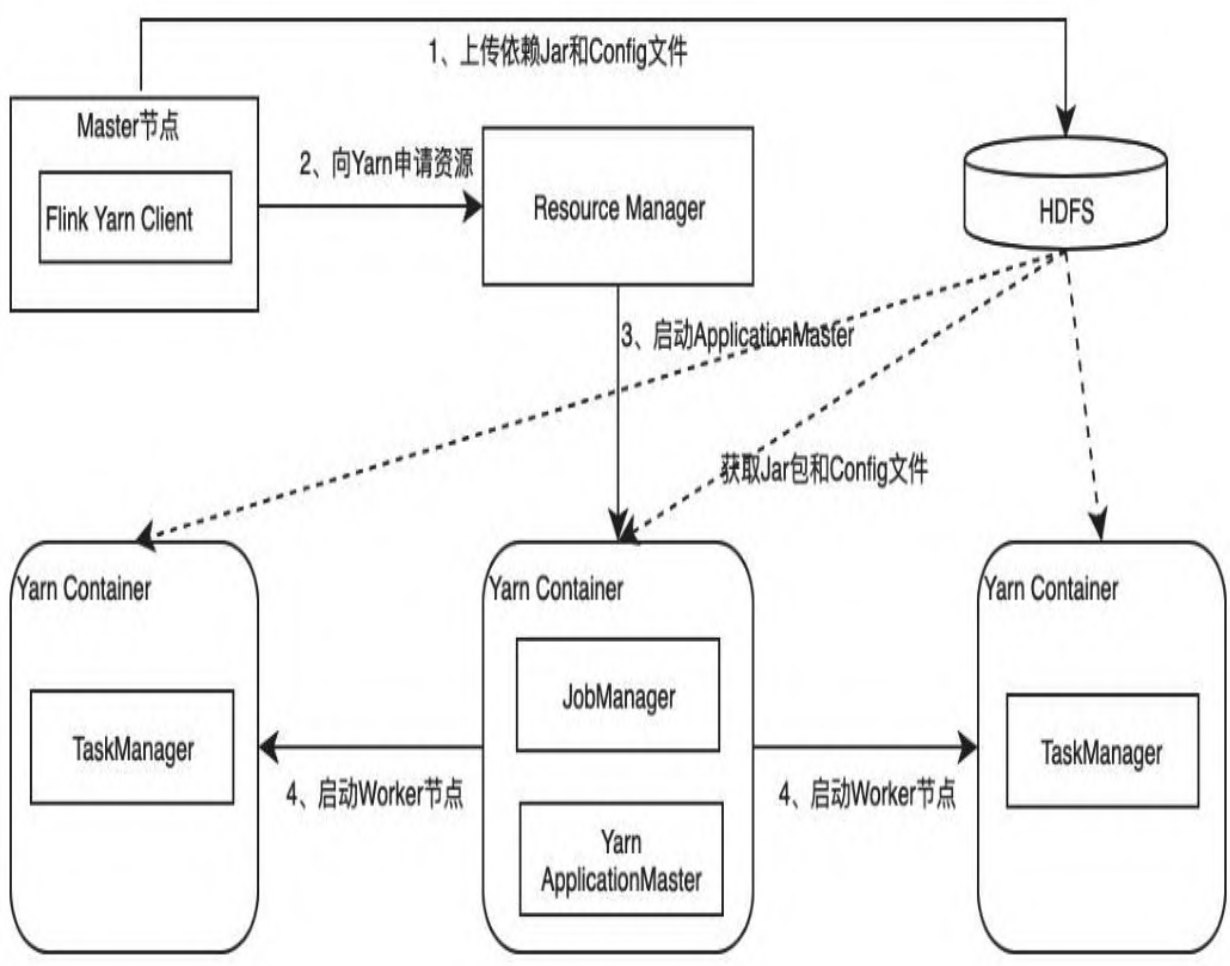

图9-1　Flink On Yarn运行流程图

1. 环境依赖

将Flink任务部署在Yarn Cluster之前，需要确认Hadoop环境是否满足以下两点要求：

· Hadoop版本至少保证在2.2以上，并且集群中安装有HDFS服务。

· 主机中已经配置HADOOP_CONF_DIR变量指定Hadoop客户端配置文件目录，并在对应的路径中含有Hadoop配置文件，其中主要包括hdfs-default.xml、hdfs-site.xml以及yarn-site.xml等。在启动Flink集群的过程中，Flink会通过识别HADOOP_CONF_DIR环境变量读取Hadoop配置参数。

2. 集群安装

通过从Flink官方下载地址下载Flink安装包，选择一台具有Hadoop客户端配置的主机，解压并进入到安装路径中，基本完成Flink集群的安装，下面介绍每种模式对应环境的启动方式。

```
tar xvzf flink-1.7.0-bin-hadoop2.tgz
cd flink-1.7.0/
```

3. Yarn Session模式

前面已经提到Yarn Session模式其实是在Yarn上启动一个Flink Session集群，其中包括JobManager和TaskManager组件。Session集群会一直运行在Hadoop Yarn之上，底层对应的其实是Hadoop的一个Yarn Application应用。当Yarn Session Cluster启动后，用户就能够通过命令行或RestAPI等方式向Yarn Session集群中提交Flink任务，从而不需要再与Yarn进行交互，这样其实也是让Flink应用在相同的集群环境运行，从而屏蔽底层不同的运行环境。

· 启动Yarn Session Cluster

首先启动Yarn Session Cluster之前Flink需要使用Hadoop客户端参数，Flink默认使用YARN_CONF_DIR或者HADOOP_CONF_DIR环境变量获取Hadoop客户端配置文件。如果启动的节点中没有相应的环境变量和配置文件，则可能导致Flink启动过程无法正常连接到Hadoop Yarn集群。

如果节点中没有相应的环境变量，则建议用户在每次启动Yarn Session之前通过手动的方式对环境变量进行赋值。如果启动节点中没有Hadoop客户端配置，用户可以将配置从Hadoop集群中获取出来，然后放置在指定路径中，再通过上述步骤进行环境变量配置。

如下通过yarn-session.sh命令启动Flink Yarn Session集群，其中-n参数配置启动4个Yarn Container，-jm参数配置JobManager的JVM内存大小，-tm参数配置Task-Manager的内存大小，-s表示集群中共启动16个slots来提供给应用以启动task实例。

```
./bin/yarn-session.sh -n 4 -jm 1024m -tm 4096m -s 16
```

集群启动完毕之后就可以在Yarn的任务管理页面查看Flink Session集群状况，并点击ApplicationMaster对应的URL，进入Flink Session Cluster集群中，注意在On YARN的模式中，每次启动JobManager的地址和端口都不是固定的。

· Yarn Session独立模式

通过以上方式启动Yarn Session集群，集群的运行与管理依赖于于本地Yarn Session集群的本地启动进程，一旦进程关闭，则整个Session集群也会终止。此时可以通过，在启动Session过程中指定参数--d或--detached，将启动的Session集群交给Yarn集群管理，与本地进程脱离。通过这种方式启动Flink集群时，如果停止Flink Session Cluster，需要通过Yarn Application -kill [appid]来终止Flink Session集群。

· 本地进程绑定已有的Session

与Detached Yarn Session相反，如果用户想将本地进程绑定到Yarn上已经提交的Session，可以通过以下命令Attach本地的进程到Yarn集群对应的Application，然后Yarn集群上ApplicationID对应的Session就能绑定到本地进程中。此时用户就能够对Session进行本地操作，包括执行停止命令等，例如执行Ctrl+C命令或者输入stop命令，就能将Flink Session Cluster停止。

```
./bin/yarn-session.sh -id [applicationid]
```

· 提交任务到Session

当Flink Yarn Session集群构建好之后，就可以向Session集群中提交Flink任务，可以通过命令行或者RestAPI的方式提交Flink应用到Session集群中。例如通过以下命令将Flink任务提交到Session中，正常情况下，用户就能够直接进入到Flink监控页面查看已经提交的Flink任务。

```
./bin/flink run ./windowsWordCountApp.jar
```

4. 容错配置

在Yarn上执行的Flink Session集群通常情况下需要进行对应的任务恢复策略配置，以防止因为某些系统问题导致整个集群出现异常。针对在Yarn上的容错配置，Flink单独提供了如下几个相关的参数，用户可以根据实际情况进行配置使用。

· yarn.reallocate-failed

该参数表示集群中TaskManager失败后是否被重新拉起，设定为True表示重新分配资源并拉起失败的TaskManager，默认为True，其本质是Yarn是否重新分配TaskManager的Container。

· yarn.maximum-failed-containers

该参数表示集群所容忍失败Container数量的最大值，如果超过该参数，则会直接导致整个Session集群失败并停止，参数默认值为TaskManager数量，也就是用户启动集群提交任务时-n参数对应的值。

· yarn.application-attempts

该参数表示整个Session集群所在的Yarn Application失败重启的次数，如果Session集群所在的整个应用失败，则在该参数范围内，Yarn也会重新拉起相应的Application，但如果重启次数超过该参数，Yarn不会再重启应用，此时整个Flink Session会失败，与此同时Session上提交的任务也会全部停止。

以上三个参数从不同层面保证了Flink任务在Yarn集群上的正常运行，且这些参数可以在conf/flink-default.yaml文件中配置，也可以在启动Session集群时通过-D动态参数指定，如-Dyarn.application-attempts=10。

5.防火墙配置

在生产环境中的集群，一般都具有非常高的安全管控，且网络基本都是通过防火墙进行隔离，有些情况下，用户想要实现在集群之外的机器上远程提交Flink作业，这种在Standalone集群中比较容易实现，因为JopManager的Rpc端口和Rest端口都可以通过防火墙配置打开。但在Yarn Session Cluster模式下用户每启动一次Session集群，Yarn都会给相应的Application分配一个随机端口，这使得Flink Session中的JobManager的Rest和Rpc端口都会发生变化，客户端无法感知远程Session Cluster端口的变化，同时端口也可能被防火墙隔离掉，无法连接到Session Cluster集群，进而导致不能正常提交任务到集群。针对这种情况，Flink提供了相应的解决策略，就是通过端口段实现。

6.Single Job模式

在Single Job模式中，Flink任务可以直接以单个应用提交到Yarn上，不需要使用Session集群提交任务，每次提交的Flink任务一个独立的Yarn Application，且在每个任务中都会有自己的JobManager和TaskManager组件，且应用所有的资源都独立使用，这种模式比较适合批处理应用，任务运行完便释放资源。

可以通过以下命令直接将任务提交到Hadoop Yarn集群，生成对应的Flink Application。注意需要在参数中指定-m yarn-cluster，表示使用Yarn集群提交Flink任务，-yn表示任务需要的TaskManager数量。

```
./bin/flink run -m yarn-cluster -yn 2 ./windowsWordCountApp.jar
```

9.1.3 Kubernetes Cluster部署

容器化部署是目前业界非常流行的一项技术，基于Docker镜像运行能够让用户更加方便地对应用进行管理和运维。随着Docker容器编排工具Kubernetes近几年逐渐流行起来，大多数企业也逐渐使用Kubernetes来管理集群容器资源。Flink也在最近的版本中支持了Kubernetes部署模式，让用户能够基于Kubernetes来构建Flink Session Cluster，也可以通过Docker镜像的方式向Kuernetes集群中提交独立的Flink任务。下面将介绍如何基于Kubernetes部署Flink Session集群，如果用户已经有部署好的Kubernetes集群，则可以直接使用，否则需要搭建Kubernetes集群，具体的搭建步骤可以参考Kubernetes的官方网站，这里不再赘述。

1.Yaml配置

在Kubernetes上构建Flink Session Cluster，需要将Flink集群中的组件对应的Docker镜像分别在Kubernetes集群中启动，其中包括JobManager、TaskManager、JobManager-Services三个镜像服务，其中每个镜像服务都可以从中央镜像仓库中获取，用户也可以构建本地的镜像仓库，针对每个组件所相应Kubernetes Yamb配置如下：

· JobManager yaml配置

主要提供运行JobManager组件镜像的参数配置，包含JobManager自身的参数，例如RPC端口等配置信息，JobManager yaml的配置文件如下：

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: flink-jobmanager
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: flink
        component: jobmanager
```

```
    spec:
      containers:
      - name: jobmanager
        image: flink:latest
        args:
        - jobmanager
        ports:
        - containerPort: 6123
          name: rpc
        - containerPort: 6124
          name: blob
        - containerPort: 6125
          name: query
        - containerPort: 8081
          name: ui
        env:
        - name: JOB_MANAGER_RPC_ADDRESS
          value: flink-jobmanager
```

· TaskManager Yaml配置

主要提供运行TaskManager组件的参数配置，以及TaskManager自身的参数，例如RPC端口等配置信息。

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: flink-taskmanager
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: flink
        component: taskmanager
    spec:
      containers:
      - name: taskmanager
        image: flink:latest
        args:
        - taskmanager
        ports:
        - containerPort: 6121
          name: data
        - containerPort: 6122
```

```
          name: rpc
        - containerPort: 6125
          name: query
        env:
        - name: JOB_MANAGER_RPC_ADDRESS
          value: flink-jobmanager
```

· JobManagerServices配置

主要为Flink Session集群提供对外的RestApi和UI地址，使得用户可以通过Flink UI的方式访问集群并获取任务和监控信息。JobManagerServices Yaml配置文件如下：

```
apiVersion: v1
kind: Service
metadata:
  name: flink-jobmanager
spec:
  ports:
  - name: rpc
    port: 6123
  - name: blob
    port: 6124
  - name: query
    port: 6125
  - name: ui
    port: 8081
  selector:
    app: flink
    component: jobmanager
```

2.启动Flink Sesssion Cluster

当各个组件服务配置文件定义完毕后，就可以通过使用以下Kubectl命令创建Flink Session Cluster，集群启动完成后就可以通过JobJobManagerServices中配置的WebUI端口访问FlinkWeb页面。

```
//启动jobmanager-service服务
kubectl create -f jobmanager-service.yaml
```

```
//启动jobmanager-deployment服务
kubectl create -f jobmanager-deployment.yaml
//启动taskmanager-deployment服务
kubectl create -f taskmanager-deployment.yaml
```

· 获取Flink Session Cluster状态

集群启动后就可以通过kubectl proxy方式访问Flink UI，需要保证kubectl proxy在终端中运行，并在浏览器里输入以下地址，就能够访问FlinkUI，其中JobManagerHost和port是在JobManagerYaml文件中配置的相应参数。

http://{JobManagerHost:Port}/api/v1/namespaces/default/services/flink-jobmanager:ui/proxy

· 停止Flink Session Cluster

可以通过kubectl delete命令行来停止Flink Session Cluster。

```
//停止jobmanager-deployment服务
kubectl delete -f jobmanager-deployment.yaml
//停止taskmanager-deployment服务
kubectl delete -f taskmanager-deployment.yaml
//停止jobmanager-service服务
kubectl delete -f jobmanager-service.yaml
```

9.2 Flink高可用配置

目前Flink高可用配置仅支持Standalone Cluster和Yarn Cluster两种集群模式，同时Flink高可用配置中主要针对JobManager的高可用保证，因为JobManager是整个集群的管理节点，负责整个集群的任务调度和资源管理，如果JobManager出现问题将会导致新Job无法提交，并且已经执行的Job也将全部失败，因此对JobManager的高可用保证尤为重要。

Flink集群默认是不开启JobManager高可用保证的，需要用户自行配置，下面将分别介绍Flink在Standalone Cluster和Yarn Session Cluster两种集群模式下如何进行JobManager高可用配置，以保证集群安全稳定运行。

9.2.1 Standalone集群高可用配置

Standalone集群中的JobManager高可用主要借助Zookeeper来完成，Zookeeper作为大数据生态中的分布式协调管理者，主要功能是为分布式系统提供一致性协调（Coordination）服务。在Flink Standalone集群中，JobManager的服务信息会被注册到Zookeeper中，并通过Zookeeper完成JobManager Leader的选举。Standalone集群会同时存在多个JobManager，但只有一个处于工作状态，其他处于Standby状态，当Active JobManager失去连接后（如系统宕机），Zookeeper会自动从Standby中选举新的 JobManager来接管Flink集群。

如果用户并没有在环境中安装Zookeeper，也可以使用Flink中自带的Zookeeper服务，如以下代码所示需要通过在conf/zoo.cfg文件中配置需要启动的Zookeeper主机，然后Flink就能够在启动的过程中在相应的主机上启动Zookeeper服务，因此不再需要用户独立安装Zookeeper服务。

```
server.X=addressX:peerPort:leaderPort
server.Y=addressY:peerPort:leaderPort
```

在上述配置中，server.X和server.Y分别为Zookeeper服务所在主机对应的唯一ID，配置完成后通过执行bin/start-zookeeper-quorum.sh脚本来启动Zookeeper服务。需要注意的是，Flink方不建议用户在生产环境中使用这种方式，应尽可能使用独立安装的Zookeeper服务，保证生产系统的安全与稳定。

在Standalone Cluster高可用配置中，还需要对masters和flink-conf.yaml两个配置文件进行修改，以下分别介绍如何对每个配置文件的参数进行修改。

首先，在conf/masters文件中添加以下配置，用于指定jobManagerAddress信息，分别为主备节点的JobManager的Web地址和端口。

```
jobManagerAddress1:webUIPort1
jobManagerAddress2:webUIPort2
```

然后在conf/flink-conf.yaml文件中配置如下Zookeeper的相关配置：

· 高可用模式，通过Zookeeper支持系统高可用，默认集群不会开启高可用状态。

```
high-availability: zookeeper
```

· Zookeeper服务地址，多个IP地址可以使用逗号分隔。

```
high-availability.zookeeper.quorum:
zkHost:2181[,...],zkAdress:2181
```

· Zookeeper中Flink服务对应的root路径。

```
high-availability.zookeeper.path.root: /flink
```

· 在Zookeeper中配置集群的唯一ID，用以区分不同的Flink集群。

```
high-availability.cluster-id: /default_ns
```

· 用于存放JobManager元数据的文件系统地址。

```
high-availability.storageDir: hdfs:///flink/recovery
```

以下通过实例来说明如何配置两个JobManager的Standalone集群。

· 首先在conf/flink-conf.yaml文件中配置high-availability:zookeeper以及Zookeeper服务的地址和cluster-id信息等。

```
high-availability: zookeeper
high-availability.zookeeper.quorum: zkHost1:2181, zkHost2:2181,
zkHost3:2181
high-availability.zookeeper.path.root: /flink
high-availability.cluster-id: /standalone_cluster_1
high-availability.storageDir: hdfs://namenode:8020/flink/ha
```

· 其次在JobManager节点的conf/masters文件中配置Masters信息，且分别使用8081和8082端口作为JobManager Rest服务端口。

```
localhost:8081
localhost:8082
```

· 如果系统中没有Zookeeper集群，则需要配置conf/zoo.cfg以启用Flink自带的Zookeeper服务。

```
server.0=localhost:2888:3888
```

· 通过start-zookeeper-quorum.sh启动Flink自带的Zookeeper集群。

```
bin/start-zookeeper-quorum.sh
Starting zookeeper daemon on host localhost.
```

· 最后启动Standalone HA-cluster集群。

```
$ bin/start-cluster.sh
Starting HA cluster with 2 masters and 1 peers in ZooKeeper
quorum.
Starting jobmanager daemon on host localhost.
Starting jobmanager daemon on host localhost.
Starting taskmanager daemon on host localhost.
```

· 可以通过如下命令停止Flink Standalone HA集群以及ZooKeeper服务。

```
./bin/stop-cluster.sh
Stopping taskmanager daemon (pid: 7647) on localhost.
Stopping jobmanager daemon (pid: 7495) on host localhost.
Stopping jobmanager daemon (pid: 7349) on host localhost.
./bin/stop-zookeeper-quorum.sh
Stopping zookeeper daemon (pid: 7101) on host localhost.
```

9.2.2 Yarn Session集群高可用配置

在Flink Yarn Session集群模式下，高可用主要依赖于Yarn协助进行，主要因为Yarn本身对运行在Yarn上的应用具有一定的容错保证。前面已经了解到Flink On Yarn的模式其实是将Flink JobManager执行在ApplicationMaster所在的容器之中，同时Yarn不会启动多个JobManager达到高可用目的，而是通过重启的方式保证JobManager高可用。

可以通过配置yarn-site.xml文件中的yarn.resourcemanager.am.max-attempts参数来设定Hadoop Yarn中ApplicationMaster的最大重启次数。当Flink Session集群中的JobManager因为机器宕机或者重启等停止运行时，通过Yarn对ApplicationMaster的重启来恢复JobManager，以保证Flink集群高可用。

除了能够在Yarn集群中配置任务最大重启次数保证高可用之外，也可以在flink-conf.yaml中通过yarn.application-attempts参数配置应用的最大重启次数，以下配置表示Flink应用最多可以被重启4次，其中1次为集群初始化。需要注意的是，在flink-conf.yaml中配置的次数不能超过Yarn集群中配置的最大重启次数。

```
yarn.application-attempts: 5
```

下面通过实例说明具有两个JobManager的Yarn Session Cluster如何进行HA配置。

（1）首先在conf/flink-conf.yaml配置文件中添加Zookeeper信息，以及应用最大重启次数。

```
high-availability: zookeeper
high-availability.zookeeper.quorum: localhost:2181
high-availability.storageDir: hdfs://namenode:8020/flink/ha
```

```
high-availability.zookeeper.path.root: /flink
yarn.application-attempts: 5
```

（2）系统中如果没有Zookeeper集群，则需要配置conf/zoo.cfg，以启用Flink自带Zookeeper服务。

```
server.0=localhost:2888:3888
```

（3）Zookeeper配置完成后，可通过如下命令启动Flink自带Zookeeper集群。

```
$ bin/start-zookeeper-quorum.sh
Starting zookeeper daemon on host localhost.
```

（4）最后使用以下命令启动Flink Yarn Session HA集群，至此Flink基于Yarn的高可用就配置完成。

```
$ bin/yarn-session.sh -n 2
```

9.3 Flink安全管理

在开启了网络安全认证的集群环境中，往往需要对计算框架进行安全认证，例如Hadoop环境已经开启Kerberos认证，就需要在Flink任务或集群中配置相应的认证参数以通过Kerberos安全认证，认证通过的应用才能获取到集群的资源以及数据，从而完成计算过程。下面将介绍Flink在开启Kerberos安全认证的集群中如何进行配置以提交和执行Flink任务。

9.3.1 认证目标

作为分布式计算框架，Flink需要访问外部数据以及获取计算资源才能完成具体的计算任务。而在开启安全认证的集群中，一般是通过Kerberos服务对用户和服务进行双向认证，Kerberos也是目前大数据生态中比较通用的安全认证协议，Flink也只支持通过Kerberos进行网络安全认证。另外在Flink安全认证过程中，主要包含以下几个方面的安全认证：

· Connector认证

通过Connector获取外部数据源中的数据，如果数据源开启Kerberos认证，则在任务执行过程中需要对数据源进行安全认证，例如从开启Kerberos认证的Kafka集群中消费数据。

· Zookeeper认证

Flink高可用需要借助于Zookeeper集群服务，如果Zookeeper开启了Kerberos认证并使用了SASL网络传输协议，则Flink做为客户端需要对Zookeeper进行网络安全认证。

· Hadoop认证

在OnYarn集群，Flink任务需要Hadoop Yarn上获取计算资源或访问HDFS上的数据，如果Hadoop开启了Kerboros认证，则Flink集群做为客户端需要进行安全认证，才能获取到Hadoop Yarn上的计算资源以及访问HDFS中的数据。

9.3.2 认证配置

上述提到的三种认证在认证方式上也有一定的区别。例如，使用Hadoop Yarn资源或读取HDFS数据，是通过Hadoop提供的UserGroupInformation对象创建进程级别的Login Context进行认证；对于Connector连接到Kafka或其他第三方存储介质的安全认证，是通过JAAS方式，在Flink任务执行过程中，动态将认证信息注入应用中以完成安全认证。

1.Hadoop安全认证

在Hadoop安全管理中，通过Hadoop提供的UserGroupInformation对象对用户进行认证，并建立进程级别的登录上下文，这种方式适用于Flink集群对Hadoop Yarn资源的访问和HDFS数据的访问与存储。例如，Flink集群在启动过程中需要向Yarn申请资源，同时会将中间结果数据存储在HDFS上，在集群开启Kerberos认证后，Flink集群启动之前首先要对Kerberos用户进行初始化，然后才能启动TaskManager和JobManager等服务进程。

可以通过配置Flink集群以访问开启Kerberos认证的Hadoop集群。需要在Standalone集群和Yarn Session集群中每个节点上的conf/flink-conf.yaml文件中增加以下安全配置。

```
#是否使用ticket-cache
security.kerberos.login.use-ticket-cache: true
#指定Flink集群KeyTab认证凭据
security.kerberos.login.keytab: /path/to/kerberos/keytab
#指定kerboros登录凭据,默认flink-user
security.kerberos.login.principal: flink-user
#使用JAAS Login Context配置
security.kerberos.login.contexts: Client,KafkaClient
```

注意，如果配置通过keytab文件进行用户认证，则use-ticket-cache配置就会失效，系统每次会使用keytab文件进行认证。另外，需要确保Flink配置文件中security.kerberos.login.keytab路径中的keytab文件存在，且keytab中所对应的用户具有相应访问集群的权

限。配置完成之后，就可以按照正常方式启动Standalone或Yarn Session集群服务。

另外在Yarn Session集群中，除了可以通过keytab文件进行网路安全认证之外，也可以使用Ticket Cache方式来认证。首先在conf/flink-conf.yaml文件中去除keytab配置，然后将security.kerberos.login.use-ticket-cache配置设定为True。需要在每次启动Flink集群之前使用Kinit命令初始化Kerberos用户，然后按照正常方式启动Flink集群，注意，为避免Cache过期而导致集群无法启动，建议用户使用keytab的方式进行认证。

2.JAAS认证

JAAS（Java认证和授权服务）提供了认证与授权的基础框架与接口定义，适用于在Flink任务中动态地使用Kerberos凭据进行网络安全认证的情况。例如，在应用中创建Connector去访问外部数据源，Flink可以通过动态参数的形式，将Kerberos认证参数信息传递至程序中，然后进行网络安全认证操作。

9.3.3 SSL配置

在集群安全管理中，除了需要考虑网络安全认证之外，还要关注网络中数据传输的安全性，由于Flink本身是分布式计算框架的特点，必然会涉及数据在不同的计算节点中进行网络传输，而对于敏感数据，一旦被非法截获，就可能导致非常严重的后果。

在Flink集群中将网络传输分为两种：一种为Flink集群内部网络传输，主要包括JobManager和TaskManager之间的RPC/BLOB消息通信，例如Reparation操作中的数据传输等；另外一种为外部网络传输，例如Flink JobManager对外提供的RestAPI以及客户端命令行等，如图9-2所示。

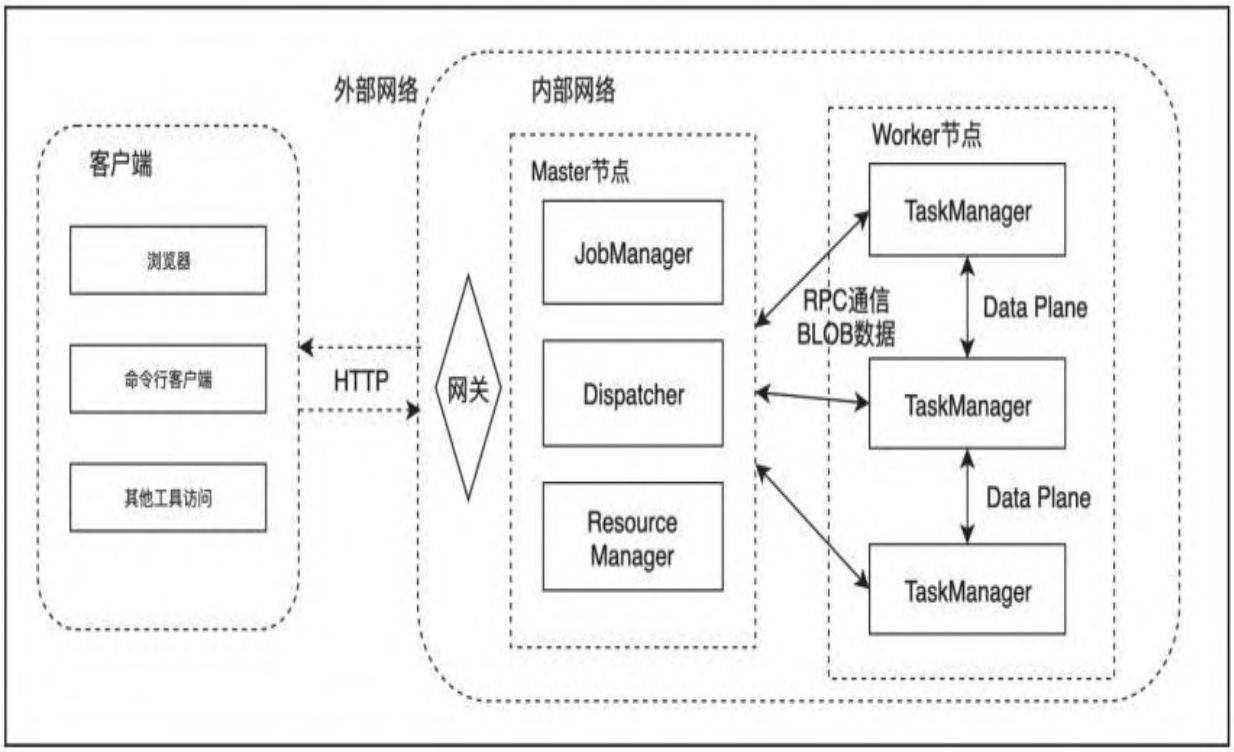

图9-2 Flink集群网络连接结构

针对每种网络传输，Flink支持使用SSL（服务层网络传输协议）对网络连接中的数据进行加密传输。另外可以对内部网络传输和外部网络传输配置不同的加密策略。

以下分别使用不同配置来打开内部网络连接和外部网络连接的SSL配置，为了向早期版本兼容，Flink也提供了security.ssl.enabled参数直接打开与关闭内部和外部网络连接的SSL配置。

· 打开内部连接的SSL配置

```
security.ssl.internal.enabled: true
```

· 打开外部连接的SSL配置

```
security.ssl.rest.enabled: true
```

在内部连接中，除了通过支持使用security.ssl.internal.enabled参数来整体打开内部连接SSL配置之外，也可以更加精细地控制内部不同的网络连接。例如，通过如下配置，仅将TaskManager之间的Data Plane数据传输SSL配置为true，其他内部网络传输不使用SSL加密传输。

```
taskmanager.data.ssl.enabled: true
```

打开JobManager到TaskManager的BLOBs网络传输的SSL配置。

```
blob.service.ssl.enabled: true
```

打开JobManager、TaskManager及ResourceManager之间基于Akka的网络连接的SSL配置。

```
akka.ssl.enabled:
```

以上配置仅是将Flink集群中内部网络或者外部网络的SSL配置激活，集群中使用SSL对数据进行加密转换，也需要分别对内网和外网的SSL配置项进行修改，其中包括存储公钥以及私钥信息的KeyStores以及主要存储可信任证书信息的Truststores。

· 内部网络SSL配置

由于Flink内部网络传输都是相互认证，KeyStore和Truststore包含相同的专用证书，且基于证书就可以在机器和服务之间形成互信机制，因此就可以保证内部网络传输安全。通过在flink-conf.yaml文件中配置如下信息来启用内部网络SSL通信加密，配置中分别包含了keystore以及truststore的相关配置选项。

```
security.ssl.internal.keystore: /path/to/file.keystore
security.ssl.internal.keystore-password: keystore_password
security.ssl.internal.key-password: key_password
security.ssl.internal.truststore: /path/to/file.truststore
security.ssl.internal.truststore-password: truststore_password
```

· 外部网络连接SSL配置

对于外部网络连接，默认情况下keystore在服务端中使用，truststore是在Rest客户端中去接收服务端的认证证书，从而完成服务端和客户端的互信配置。可以通过如下配置对外部Rest网络连接中的SSL进行配置。

```
security.ssl.rest.keystore: /path/to/file.keystore
security.ssl.rest.keystore-password: keystore_password
security.ssl.rest.key-password: key_password
security.ssl.rest.truststore: /path/to/file.truststore
security.ssl.rest.truststore-password: truststore_password
security.ssl.rest.authentication-enabled: false
```

9.4 Flink集群升级

对于长时间运行的Flink流式任务，如何进行有效地升级和运维对整个平台显得至关重要。与批计算场景不同，流式任务是7*24小时不间断运行，一旦任务启动，数据就会源源不断地接入到Flink系统中。如果在每次停止或重启Flink任务的过程中，不能及时保存原有任务中的状态数据，将会导致任务中的数据出现不一致的问题。为了解决此类问题Flink引入了Savepoint，它是Checkpoint的一种特殊实现，目的是在停止任务时，能够将任务中的状态数据落地到磁盘中并持久化，然后当重新拉起任务时，就能够从已经持久化的Savepoint数据中恢复原有任务中的数据，从而保证数据的一致性。

另外，对于某些情况，例如任务中的代码逻辑改变，导致算子中状态计算方式和之前的版本不一致等，Flink也提供了相应的方法对任务的状态数据进行恢复。

9.4.1 任务重启

Flink的任务在停止或启动过程中，可以通过使用savepoint命令将Flink整个任务状态持久化到磁盘中。如以命令对正在执行的Flink任务进行savepoint操作，首先通过指定jobID确定需要重启的应用，然后通过指定Savepoint存储路径，将算子状态数据持久化到指定的本地路径中。

```
./bin/flink savepoint <jobID> [pathToSavepoint]
```

需要注意的是，savepoint命令仅是将任务的状态数据持久化到固定路径中，任务并不会终止。可以通过如下cancel命令将正在运行的Flink任务优雅地停止，在停止过程中外部数据将不再接入，且Flink也不再执行Checkpoint操作，Savepoint中存储着当前任务中最新的一份Snapshot数据。

```
./bin/flink cancel -s [pathToSavepoint] <jobID>
```

可以通过使用flink run命令加-s参数启动Cancel的Flink应用，其中-s参数代表Savepoint数据存储路径，这样Flink任务将可以从之前任务中所持久化的savepoints数据进行恢复，其中包括算子的状态和用户定义的状态数据等。

```
./bin/flink run -d -s [pathToSavepoint] ~/MyStreaming.jar
```

9.4.2 状态维护

前面已经讲过，Flink将计算逻辑构建在不同的Operator中，且Operator主要分为两种类型：一种为有状态算子，例如基于窗口计算的算子；另一种为无状态算子，例如常见的map转换算子等。默认情况下，Flink会对应用中的每个算子都会生成一个唯一的ID，当应用代码没有发生改变，任务重启就能够根据默认算子ID对算子中的数据进行恢复。但如果应用代码发生了比较大的修改，例如增加或者修改了算子的逻辑，将有可能导致算子中状态数据无法恢复的情况。针对这种情况，Flink允许在编写应用代码时对有状态的算子让用户手工指定唯一ID。如以下代码所示，将MyStatefulMapFunc通过“mapper-1”ID标记，这样用户在调整应用代码的过程中只要沿用之前的算子ID，就能够从Savepoint中恢复有状态算子的数据。

```
val mappedEvents= events.map(new MyStatefulMapFunc()).uid("mapper-1")
```

通常情况下，并不是所有的算子都需要指定ID，用户可根据情况对部分重要的算子指定ID，其他算子可以使用系统分配的默认ID。同时Flink官方建议用户尽可能将所有算子进行手工标记，以便于在未来进行系统的升级和调整。另外对于新增加的算子，在Savepoint中没有维护对应的状态数据，Flink会将算子恢复到默认状态。

对于在升级过程中，如果应用接入的数据类型发生变化，也可能会导致有状态算子数据恢复失败。有状态算子从定义的形式上共分为两种类型：一种为用户自定义有状态算子，例如通过实现RichMapFunction函数定义的状态算子；另一种为Flink内部提供的有状态算子，例如窗口算子中的状态数据等。每种算子除了通过ID进行标记以外，当接入数据类型发生变化时，数据恢复策略也有所不同。

· 自定义算子状态

用户自定义状态算子一般是用户自定义RichFunction类型的函数，然后通过接口注册在计算拓扑中，函数中维系了自定义中间状态

数据，在升级和维护过程中，需要用户对算子的状态进行兼容和适配。例如，如果算子状态数据的类型发生变化，则可以通过定义中间算子，间接地将旧算子状态转换为新的算子状态。

· 算子内部状态

默认情况下，Flink算子内部的状态是不向用户开放的，这些有状态算子相对比较固定，对输入和输出数据有非常明确的限制。目前Flink内部算子不支持数据类型的更改，如果输入和输出数据类型发生改变，则可能会破坏算子状态的一致性，从而影响到应用的升级，因此用户在使用这类算子的过程中应尽可能避免输入和输出的数据类型发生变化。

9.4.3　版本升级

前面介绍了Flink单个应用的重启和升级，会涉及对应用中的算子状态进行恢复。随着Flink的发展，社区会不断提供新的稳定版本供用户选择和使用，而这个过程中就会涉及Flink整体集群版本的升级。而在同一套集群之上可能会有非常多Flink应用运行，因此如何做到版本向下兼容以及应用数据的恢复是目前社区要重点解决的问题之一。

在Flink集群版本进行升级的过程中，主要包含了两步操作，分别为：

（1）执行Savepoint命令，将系统数据写入到Savepoint指定路径中。

（2）升级Flink集群至新的版本，并重新启动集群，将任务从之前的Savepoint数据中恢复。

在升级操作中，用户可以根据不同的业务场景选择原地升级或卷影复制升级的策略。其中原地升级表示直接从原有版本的Flink集群位置升级，需要事先将集群停止，并替换新的集群安装包；而卷影复制不需要立即停止原有集群，首先将Savepoint数据获取下来，并在服务器其他位置重新搭建一套新的Flink集群，新的集群升级完毕后，通过在Savepoint数据中恢复，然后将原有Flink集群关闭。

目前随着Flink版本的迭代，形成众多的社区版本，多个版本之间的兼容性会有一定的差别，有的版本能够完全向下兼容，但是个别版本不可以，用户可以在官方网站中查询Flink各个版本的向下兼容情况。

9.5 本章小结

本章重点介绍了如何在集群环境中部署和应用Flink，其中涵盖了常见的Flink部署模式，例如基于Standalone、YarnKubernetes等集群部署模式。在9.2节介绍了如何通过开启Standalone和Yarn集群的高可用配置，从而保障集群在生产环境中的稳定运行。9.3节重点介绍了Flink安全管理方面的内容，包括如果和开启kerboros认证的Hadoop集群进行网络安全认证，以及如何通过Hadoop UserGroupInfmation提供统一认证以及使用JAAS进行Kerberos动态认证。9.4节介绍了Flink集群升级和运维的过程，包含对单个任务以及整个集群的升级，以及如何保证在升级过程中Flink任务中数据的一致性。经过本章的学习，能够让读者从实际使用的角度对Flink有一个更加深入的认识，以及如何将Flink用到真正的生产场景中，解决现实中的实际应用问题。

第10章

Flink监控与性能优化

对于构建好的Flink集群，如何能够有效地进行集群以及任务方面的监控与优化是非常重要的，尤其对于7*24小时运行的生产环境。本章将从多个方面介绍Flink在监控和性能优化方面的内容，其中监控部分包含了Flink系统内部提供的常用监控指标的获取，以及用户如何自定义实现指标的采集和监控。同时，也会重点介绍Checkpointing以及任务反压的监控。然后通过分析各种监控指标帮助用户更好地对Flink应用进行性能优化，以提高Flink任务执行的数据处理性能和效率。

10.1　监控指标

Flink任务提交到集群后，接下来就是对任务进行有效的监控。Flink将任务监控指标主要分为系统指标和用户指标两种：系统指标主要包括Flink集群层面的指标，例如CPU负载，各组件内存使用情况等；用户指标主要包括用户在任务中自定义注册的监控指标，用于获取用户的业务状况等信息。Flink中的监控指标可以通过多种方式获取，例如可以从Flink UI中直接查看，也可以通过Rest Api或Reporter获取。

10.1.1　系统监控指标

如图10-1所示，在FlinkUI Overview页签中，包含对系统中的TaskManager、TaskSlots以及Jobs的相关监控指标，用户可通过该页面获取正在执行或取消的任务数，以及任务的启动时间、截止时间、执行周期、JobID、Task实例等指标。

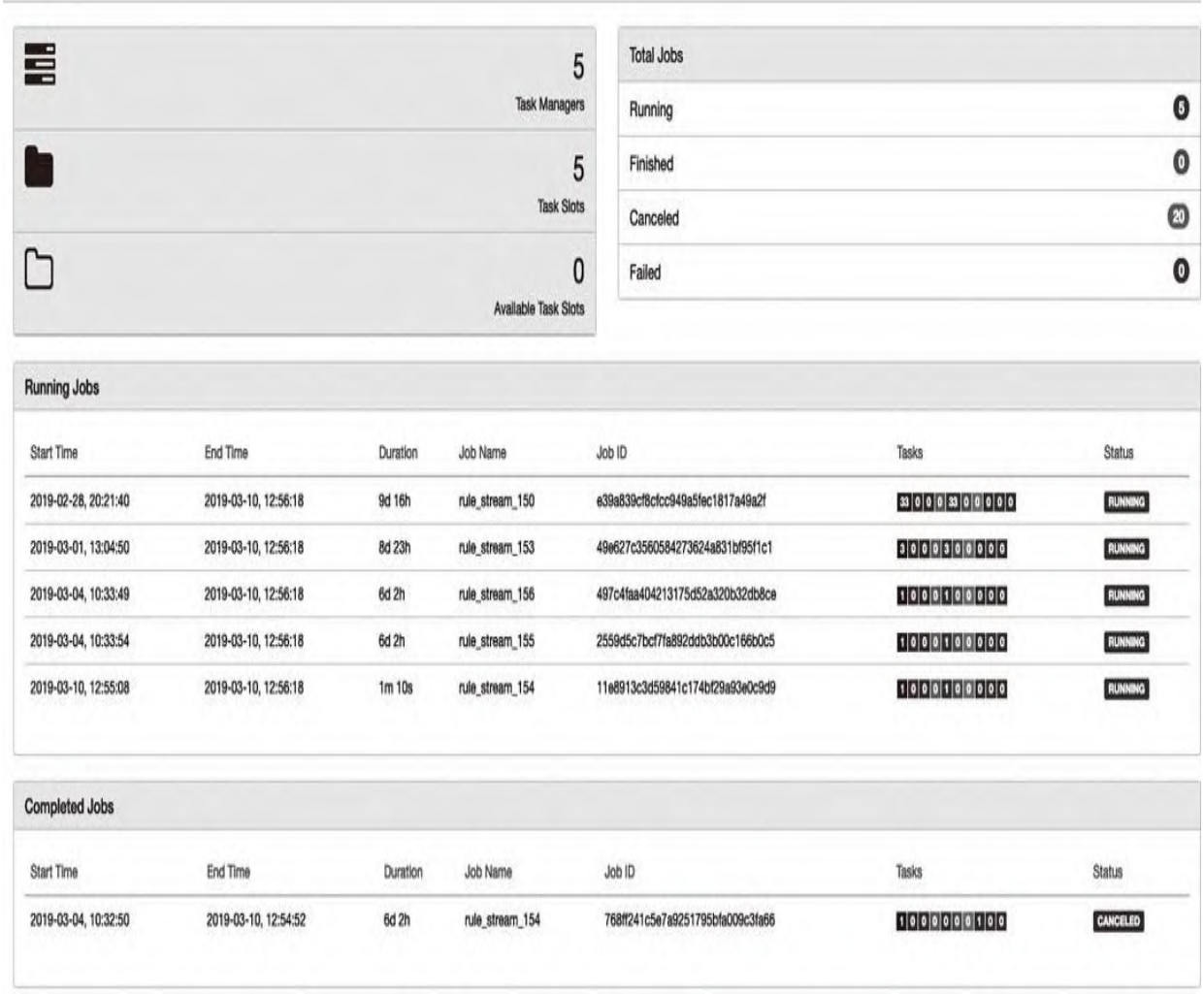

图10-1　Flink监控Overview页面

在Task Manager页签中可以获取到每个TaskManager的系统监控指标，例如JVM内存，包括堆外和堆内存的使用情况，以及NetWork中内存的切片数、垃圾回收的次数和持续时间等。

另外除了可以在FlinkUI中获取指标之外，用户也可以使用Flink提供的RestAPI获取监控指标。通过使用http://hostname:8081拼接需要查询的指标以及维度信息，就可以将不同服务和任务中的Metric信息查询出来，其中hostname为JobManager对应的主机名。

· 例如访问http://hostname:8081/jobmanager/metrics来获取所有JobManager的监控指标名称。

```
GET http://hostname: 8081/jobmanager/metrics
查询结果：
[{
     "id": "Status.JVM.GarbageCollector.PS_MarkSweep.Time"
    },{
     "id": "Status.JVM.Memory.NonHeap.Committed"
    },
    ...
]
```

同时可以在URL中添加get=metric1,metric2参数获取指定Metric的监控指标。例如，获取Jobanager内存CPU占用时间，就可以通过拼接get=Status.JVM.CPU.Time获取，其他的监控指标查询方法相似。

```
GET /jobmanager/metrics?get=Status.JVM.CPU.Time
 查询结果：
[
    {
        "id": "Status.JVM.CPU.Time",
        "value": "7052900000000"
    }
]
```

· 获取taskmanagers中metric1和metric2对应的Value。

```
GET /taskmanagers/metrics?get=metric1,metric2
```

· 获取taskmanagers中metric1和metric2对应的Value，以及所有taskmanagers的指标统计的最大值和最小值。

```
GET /taskmanagers/metrics?get=metric1,metric2&agg=max,min
```

· 获取指定taskmanagerid对应的TaskManager上的全部监控指标的Metric名称。

```
GET /taskmanagers/<taskmanagerid>/metrics
```

· 获取指定JobID对应的监控指标对应的Metric名称。

```
GET /jobs/<jobid>/metrics
```

· 获取指定JobID对应的任务中的metric1和metric2指标。

```
GET /jobs/<jobid>/metrics?get=metric1,metric2
```

· 获取指定JobID以及指定顶点的subtask监控指标。

```
GET /jobs/<jobid>/vertices/<vertexid>/subtasks/<subtaskindex>
```

对于监控指标的metric名称可以在官网中获取，这里不再深入介绍。

10.1.2　监控指标注册

除了使用Flink系统自带的监控指标之外，用户也可以自定义监控指标。可以通过在RichFunction中调用getRuntimeContext().getMetricGroup()获取MetricGroup对象，然后将需要监控的指标记录在MetricGroup所支持的Metric中，然后就可以将自定义指标注册到Flink系统中。目前Flink支持Counters、Gauges、Histograms以及Meters四种类型的监控指标的注册和获取。

· Counters指标

Counters指标主要为了对指标进行计数类型的统计，且仅支持Int和Long数据类型。例如在代码清单10-1中实现了map方法中对进入到算子中的正整数进行计数统计，得到MyCounter监控指标。

代码清单10-1　实现RichMapFunction定义Counters指标

```
class MyMapper extends RichMapFunction[Long, Long] {
  @transient private var counter: Counter = _
  //在Open方法中获取Counter实例化对象
  override def open(parameters: Configuration): Unit = {
    counter = getRuntimeContext()
      .getMetricGroup()
      .counter("MyCounter")}
  override def map(input : Long): Long = {
    if (input> 0) { counter.inc()//如果value>0,则counter自增}
    value
  }}
```

· Gauges指标

Gauges相对于Counters指标更加通用，可以支持任何类型的数据记录和统计，且不限制返回的结果类型。如代码清单10-2所示，通过使用Gauges指标对输入Map函数中的数据量进行累加统计，得到MyGauge统计指标。

代码清单10-2　实现RichMapFunction定义Gauges指标

```
class gaugeMapper extends RichMapFunction[String,String] {
  @transient private var countValue = 0
  //在Open方法中获取gauge实例化对象
  override def open(parameters: Configuration): Unit = {
    getRuntimeContext()
      .getMetricGroup()
      .gauge[Int, ScalaGauge[Int]]("MyGauge", ScalaGauge[Int]( ()
=> countValue ) )
  }
  override def map(value: String): String = {
    countValue += 1 //累加countValue值
    value
  }
}
```

· Histograms指标

Histograms指标主要为了计算Long类型监控指标的分布情况，并以直方图的形式展示。Flink中没有默认的Histograms实现类，可以通过引入Codahale/DropWizard Histograms来完成数据分布指标的获取。注意，DropwizardHistogramWrapper包装类并不在Flink默认依赖库中，需要单独引入相关的Maven Dependency。如代码清单10-3所示，定义histogramMapper类实现RichMapFunction接口，使用DropwizardHistogram-Wrapper包装类转换Codahale/DropWizard Histograms，统计Map函数中输入数据的分布情况。

代码清单10-3　实现RichMapFunction定义Histogram指标

```
class histogramMapper extends RichMapFunction[Long, Long] {
  @transient private var histogram: Histogram = _
 //在Open方法中获取Histogram实例化对象
  override def open(config: Configuration): Unit = {
//使用DropwizardHistogramWrapper包装类转换Codahale/DropWizard
Histograms
    val dropwizardHistogram =
      new com.codahale.metrics.Histogram(new
SlidingWindowReservoir(500))
    histogram = getRuntimeContext()
      .getMetricGroup()
      .histogram("myHistogram", new
```

```
      DropwizardHistogramWrapper(dropwizardHistogram))
  }
  override def map(value: Long): Long = {
    histogram.update(value) //更新指标
    value
  }}
```

· Meters指标

Meters指标是为了获取平均吞吐量方面的统计，与Histograms指标相同，Flink中也没有提供默认的Meters收集器，需要借助Codahale/DropWizard meters实现，并通过DropwizardMeterWrapper包装类转换成Flink系统内部的Meter。如代码清单10-4所示，在实现的RichMapFunction中定义Meter指标，并在Meter中使用markEvent()标记进入到函数中的数据。

代码清单10-4　实现RichMapFunction定义Meter指标

```
class meterMapper extends RichMapFunction[Long,Long] {
  @transient private var meter: Meter = _
  //在Open方法中获取Meter实例化对象
  override def open(config: Configuration): Unit = {
//使用DropwizardMeterWrapper包装类转换Codahale/DropWizard Meter
    val dropwizardMeter = new com.codahale.metrics.Meter()
    meter = getRuntimeContext()
      .getMetricGroup()
      .meter("myMeter", new
DropwizardMeterWrapper(dropwizardMeter))
  }
  override def map(value: Long): Long = {
    meter.markEvent() //更新指标
    value
  }
}
```

当自定义监控指标定义完毕之后，就可以通过使用RestAPI获取相应的监控指标，具体的使用方式读者可参考上一小节内容。

10.1.3 监控指标报表

Flink提供了非常丰富的监控指标Reporter，可以将采集到的监控指标推送到外部系统中。通过在conf/flink-conf.yaml中配置外部系统的Reporter，在Flink集群启动的过程中就会将Reporter配置加载到集群环境中，然后就可以把Flink系统中的监控指标输出到Reporter对应的外部监控系统中。目前Flink支持的Reporter有JMX、Graphite、Prometheus、StatsD、Datadog、Slf4j等系统，且每种系统对应的Reporter均已在Flink中实现，用户可以直接配置使用。

在conf/flink-conf.yaml中Reporter的配置包含以下几项内容，其中reporter-name是用户自定义的报表名称，同时reporter-name用以区分不同的reporter。

· metrics.reporter..：配置reporter-name对应的报表的参数信息，可以通过指定config名称将参数传递给报表系统，例如配置服务器端口.port:9090。

· metrics.reporter..class：配置reporter-name对应的class名称，对应类依赖库需要已经加载至Flink环境中，例如JMX Reporter对应的是org.apache.flink.metrics.jmx.JMXReporter。

· metrics.reporter..interval：配置reporter指标汇报的时间间隔，单位为秒。

· metrics.reporter..scope.delimiter：配置reporter监控指标中的范围分隔符，默认为metrics.scope.delimiter对应的分隔符。

· metrics.reporters：默认开户使用的reporter，通过逗号分隔多个reporter，例如reporter1和reporter2。

如下通过介绍Jmx以及Prometheus两种Reporter来说明如何使用Reporter完成对监控指标的输出。

1. JMX Reporter配置

JMX可以跨越一系列异构操作系统平台、系统体系结构和网络传输协议，灵活地开发无缝集成的系统、网络和服务管理应用。目前大多数的应用都支持JMX，主要因为JMX可以为运维监控提供简单可靠的数据接口。Flink在系统内部已经实现JMX Reporter，并通过配置就可以使用JMX Reporter输出监控指标到JMX系统中。

```
metrics.reporter.jmx.class:
org.apache.flink.metrics.jmx.JMXReporter
metrics.reporter.jmx.port: 8789
```

其中class配置对应的org.apache.flink.metrics.jmx.JMXReporter类已经集成在Flink系统中，用户可以直接配置使用，metrics.reporter.jmx.port配置是JMX服务的监听端口。

2.Prometheus Reporter配置

对于使用Prometheus Reporter将监控指标发送到Prometheus中，首先需要在启动集群前将/opt/flink-metrics-prometheus_2.11-1.7.2.jar移到/lib路径下，并在conf/flink-conf.yaml中配置Prometheus Reporter相关信息。Prometheus Reporter有两种形式，一种方式是通过配置Prometheus监听端口将监控指标输出到对应端口中，也可以不设定端口信息，默认使用9249，对于多个Prometheus Reporter实例，可以使用端口段来设定。

```
metrics.reporter.prometheus.class:
org.apache.flink.metrics.prometheus.PrometheusReporter
metrics.reporter.prometheus.port: 9249
```

另外一种方式是使用PrometheusPushGateway，将监控指标发送到指定网关中，然后Prometheus从该网关中拉取数据，对应的Reporter Class为PrometheusPushGateway-Reporter，另外需要指定Pushgateway的Host、端口以及JobName等信息，通过配置

deleteOnShutdown来设定Pushgateway是否在关机情况下删除metrics指标。

```
metrics.reporter.promgateway.class:
org.apache.flink.metrics.prometheus.PrometheusPushGatewayReporter
metrics.reporter.promgateway.host: localhost
metrics.reporter.promgateway.port: 9091
metrics.reporter.promgateway.jobName: myJob
metrics.reporter.promgateway.randomJobNameSuffix: true
metrics.reporter.promgateway.deleteOnShutdown: false
```

10.2 Backpressure监控与优化

反压在流式系统中是一种非常重要的机制，主要作用是当系统中下游算子的处理速度下降，导致数据处理速率低于数据接入的速率时，通过反向背压的方式让数据接入的速率下降，从而避免大量数据积压在Flink系统中，最后系统无法正常运行。Flink具有天然的反压机制，不需要通过额外的配置就能够完成反压处理。

10.2.1 Backpressure进程抽样

当在FlinkUI中切换到Backpressure页签时，Flink才会对整个Job触发反压数据的采集，反压过程对系统有一定的影响，主要因为JVM进程采样成本较高。如图10-2所示，Flink通过在TaskManager中采样LocalBufferPool内存块上的每个Task的stackTrace实现。默认情况下，TaskManager会触发100次采样，然后将采样的结果汇报给JobManager，最终通过JobManager进行汇总计算，得出反压比例并在页面中展示，反压比例等于反压出现次数/采样次数。

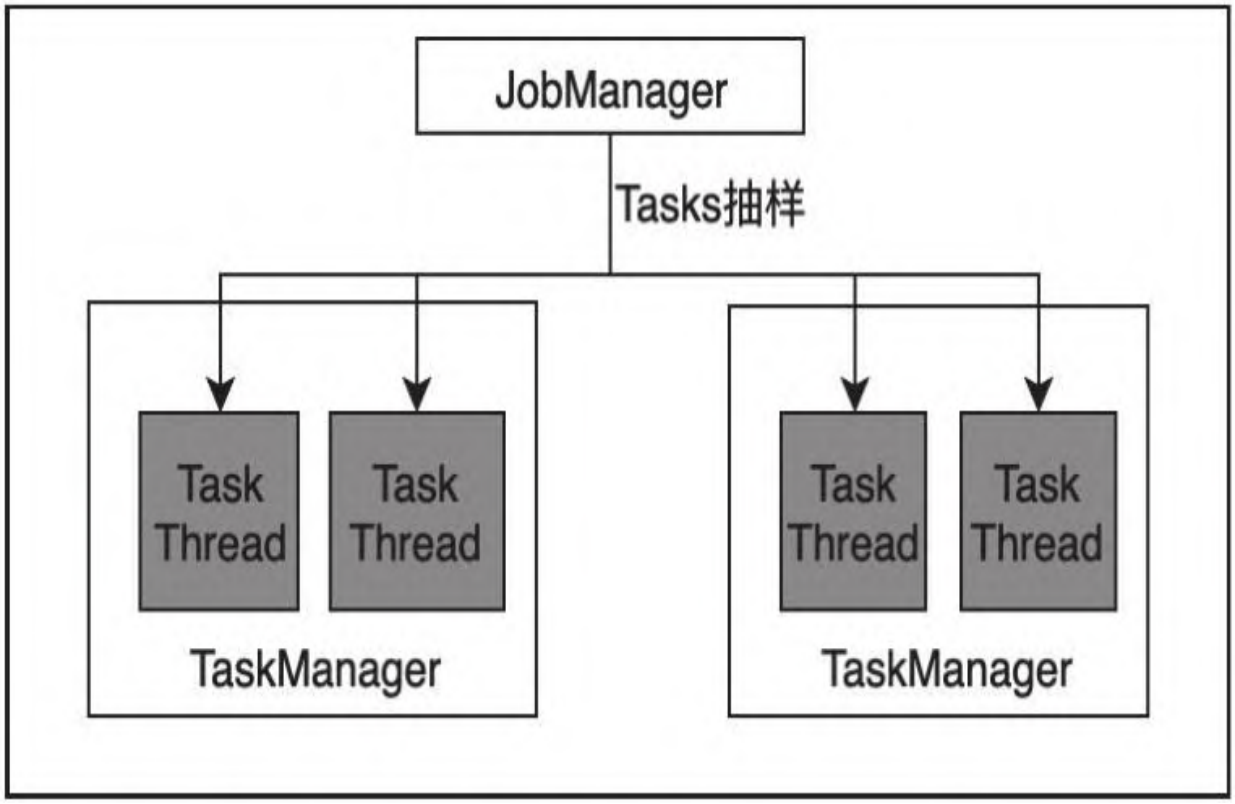

图10-2 Backpressure进程采样

如图10-3所示，通过在页面中点击Back Pressure页签触发反压检测，整个采样过程大约会持续5s中，每次采样的间隔为50ms，持续100次。同时，为了避免让TaskManager过多地采样Stack Trace，即便页面被刷新，也要等待60s后才能触发下一次Sampling过程。

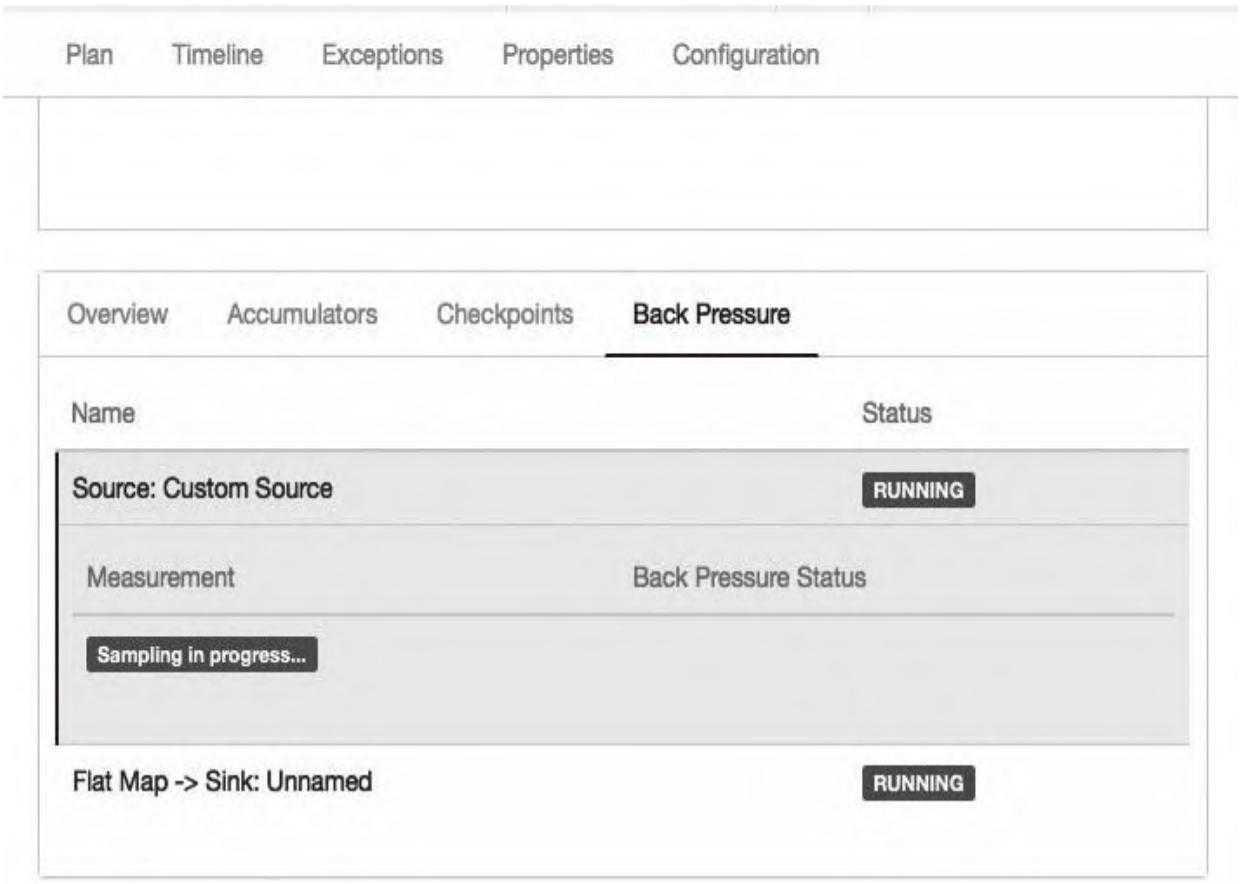

图10-3　页面开启Backpressure进程采样

10.2.2 Backpressure页面监控

通过触发JVM进程采样的方式获取到反压监控数据，同时Flink会将反压状态分为三个级别，分别为OK、LOW、HIGH级别，其中OK对应的反压比例为大于0小于10%，LOW对应的反压比例大于10%小于50%，HIGH对应的反压比例大于50%小于100%。

如图10-4所示，对Task进行抽样显示，所有的Subtasks状态均显示OK，表示未发生发大规模的数据堵塞，系统整体运行正常，不需要做任何调整。

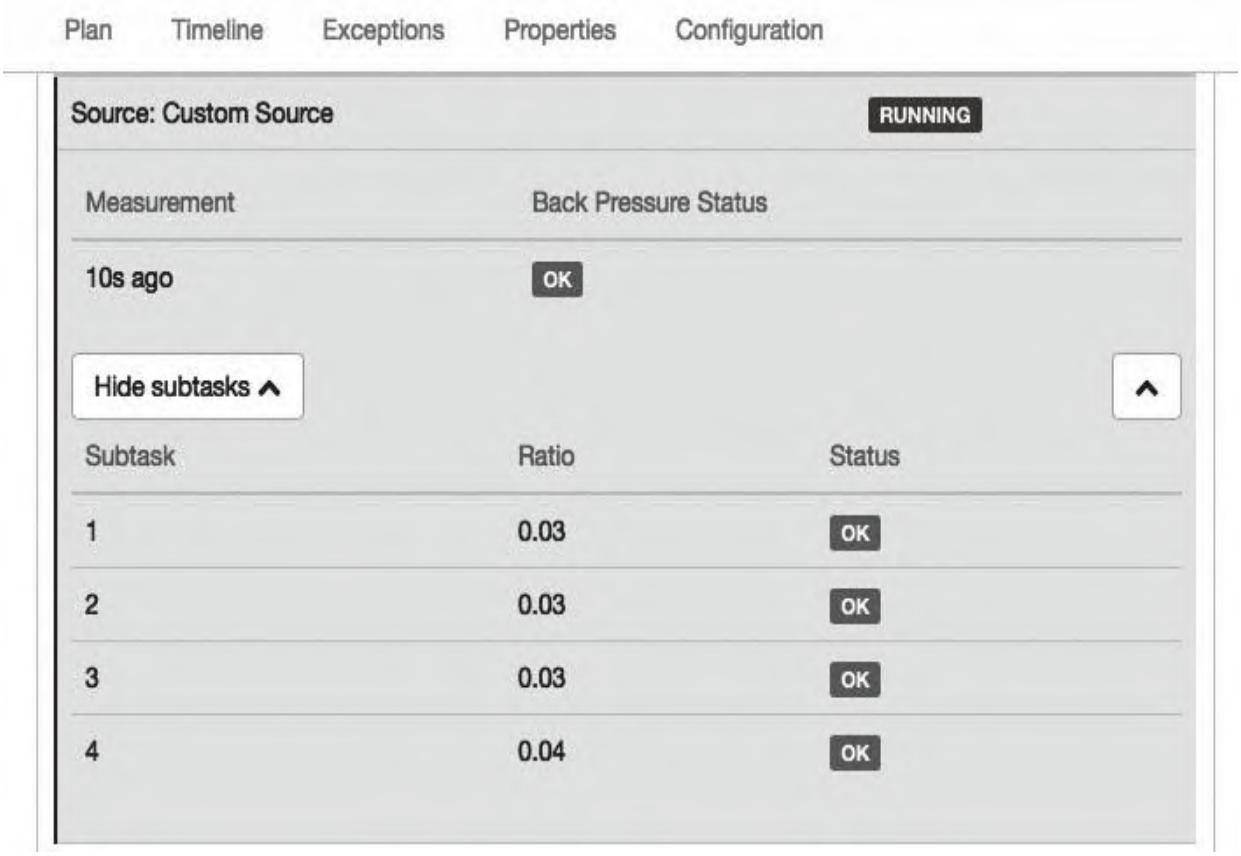

图10-4 反压状态良好

如图10-5所示，对Task进行采样检测，所有的Subtasks状态均显示HIGH，表示系统触发了比较多的反压，需要适当地增加Subtask并发

度或者降低数据生产速度，否则经过长时间的运行后，系统中处理的数据将出现比较严重的超时现象。

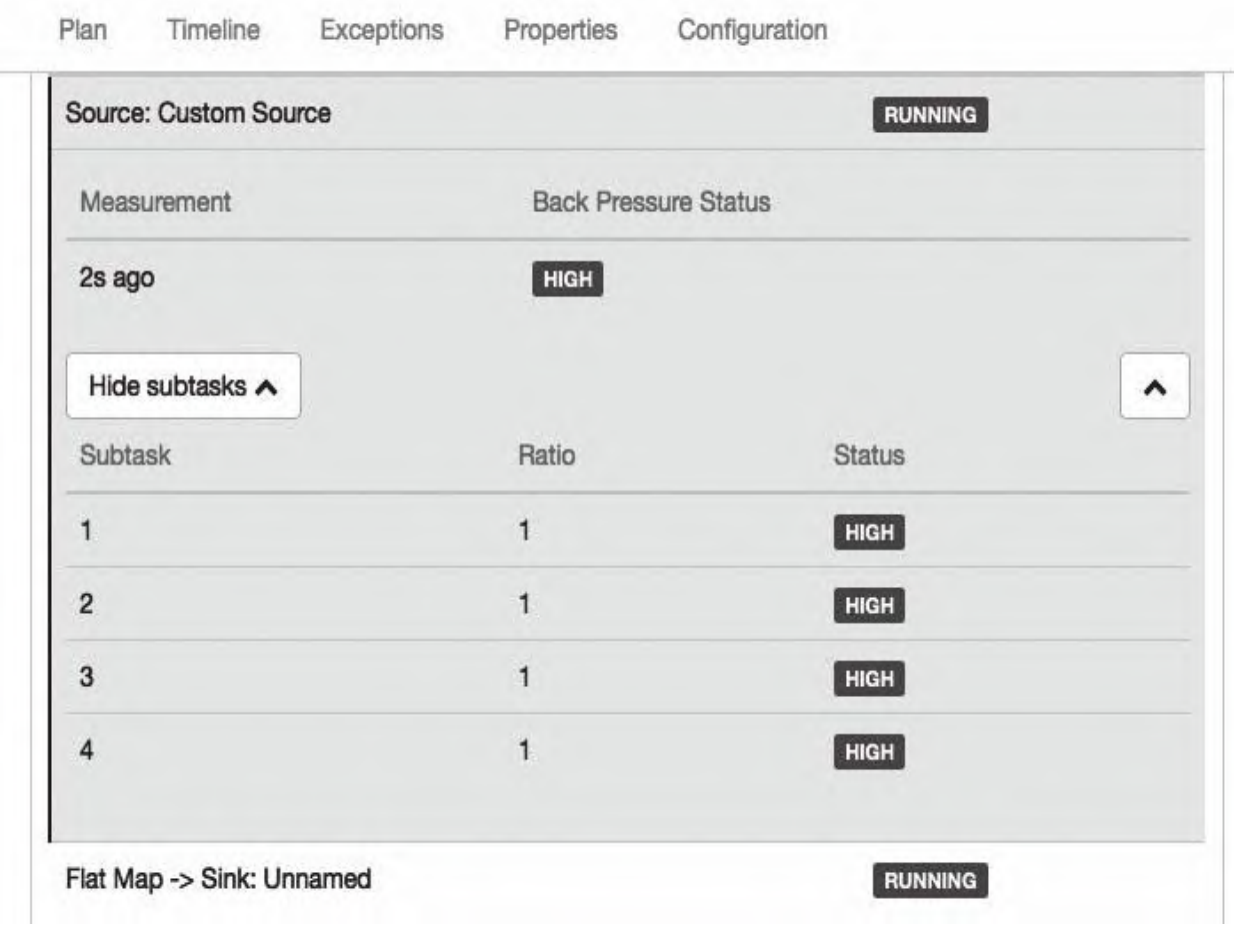

图10-5　反压状态异常

10.2.3 Backpressure配置

针对反压的优化，用户可以调整以下参数：

· web.backpressure.cleanup-interval：当启动反压数据采集后，需要等待页面并获取反压数据的时间长度，默认60s。

· web.backpressure.delay-between-samples：Stack Trace抽样到确认反压状态之间的时延，默认为50ms。

· web.backpressure.num-samples：设定Stack Trace抽样数以确定反压状态，默认为100。

10.3 Checkpointing监控与优化

10.3.1 Checkpointing页面监控

Flink Web页面中也提供了针对Job Checkpointing相关的监控信息，Checkpointing监控页面中共有Overview、History、Summary和Configuration四个页签，分别对Checkpointing从不同的角度进行了监控，每个页面中都包含了与Checkpointing相关的指标。

1.Overview页签

Overview页签中宏观地记录了Flink应用中Checkpoints的数量以及Checkpoint的最新记录，包括失败和完成的Checkpoints记录。

如图10-6所示，Overview页签中包含了以下指标，这些指标会依赖于JobManager的存活，也就是说当JobManager关闭或者重置都会置空这些统计信息。

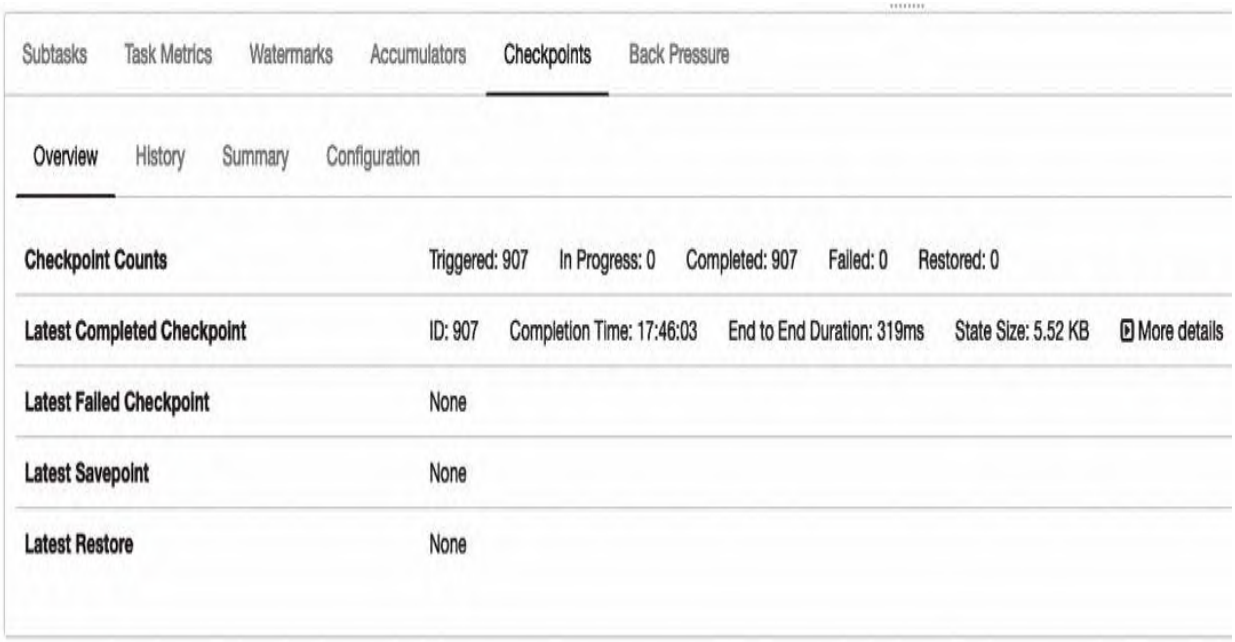

图10-6 Checkpointing Overview页面

· Checkpoint Counts：包含了触发、进行中、完成、失败、重置等Checkpoint状态数量统计。

· Latest Completed Checkpoint：记录了最近一次完成的Checkpoint信息，包括结束时间，端到端时长，状态大小等。

· Latest Failed Checkpoint：记录了最近一次失败的Checkpoint信息。

· Latest Savepoint：记录了最近一次Savepoint触发的信息。

· Latest Restore：记录了最近一次重置操作的信息，包括从Checkpoint和Savepoint两种数据中重置恢复任务。

2.History页签

如图10-7所示，History页面中记录了历史触发Checkpoint的详情，包括Checkpoint的ID、状态、触发时间，最后一次Acknowledgement信息等，通过点击More details对应的链接可以查看子Task对应的Checkpoint数据。

3. Summary页签

Summary页面中记录了所有完成的Checkpoint统计指标的最大值、最小值，以及平均值等，指标中包括端到端的持续时间、状态大小，以及分配过程中缓冲的数据大小。

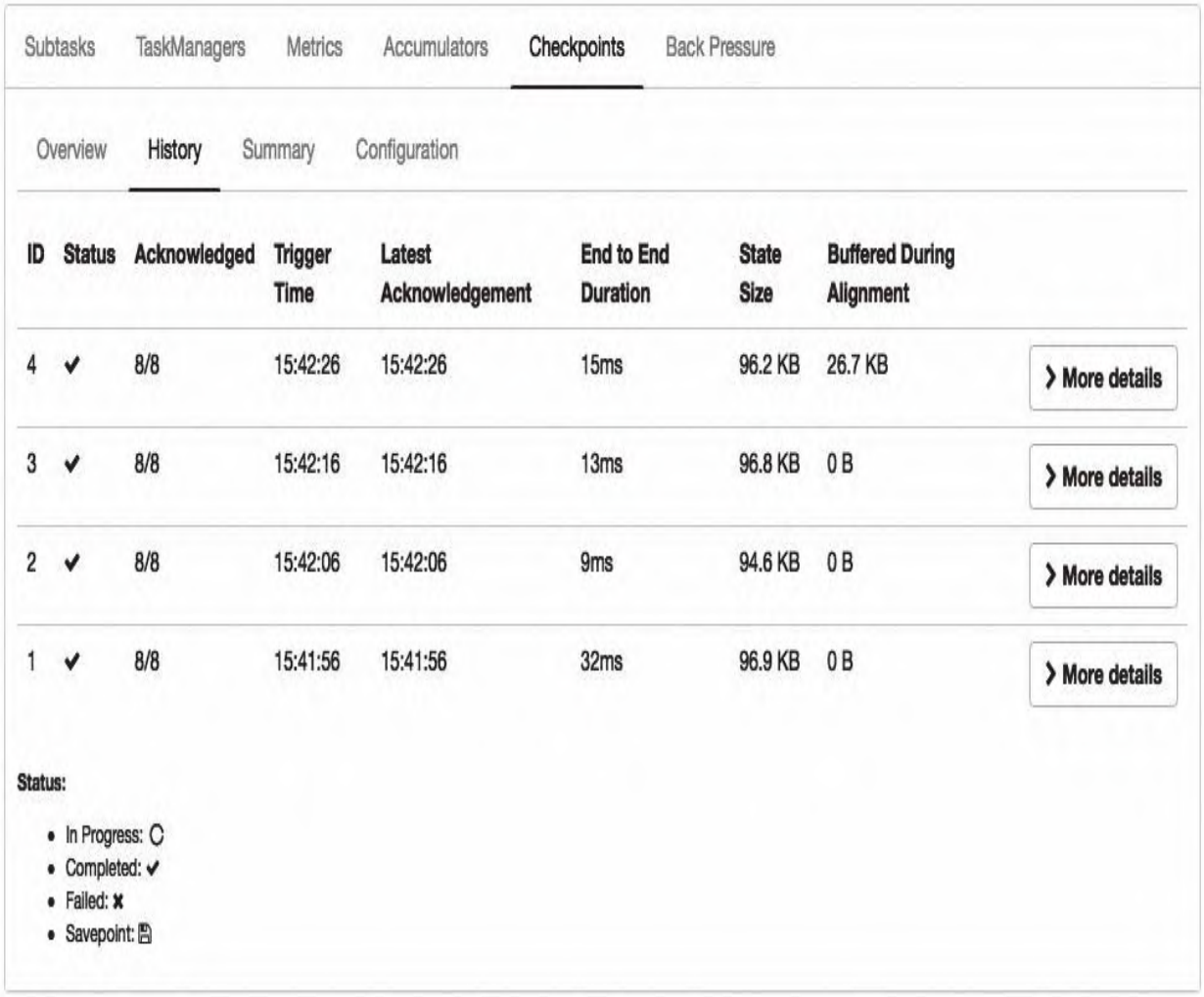

图10-7 Checkpointing History监控

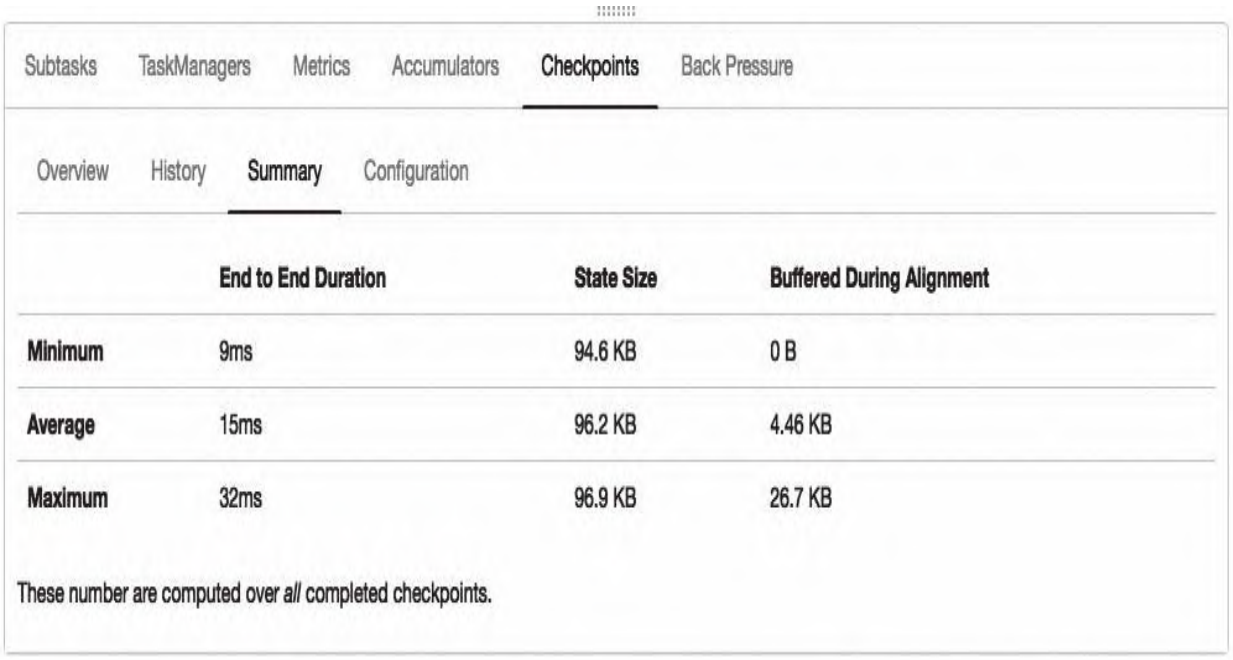

图10-8 Checkpointing Sumary监控

4.Configuration页签

Configuration页签中包含Checkpoints中所有的基本配置，具体的配置解释如下：

· Checkpointing Mode:标记Checkpointing是Exactly Once还是At Least Once的模式。

· Interval: Checkpointing触发的时间间隔，时间间隔越小意味着越频繁的Checkpointing。

· Timeout: Checkpointing触发超时时间，超过指定时间JobManager会取消当次Checkpointing，并重新启动新的Checkpointing。

· Minimum Pause Between Checkpoints:配置两个Checkpoints之间最短时间间隔，当上一次Checkpointing结束后，需要等待该时间间隔才能触发下一次Checkpoints，避免触发过多的Checkpoints导致系统资源被消耗。

· Persist Checkpoints Externally:如果开启Checkpoints，数据将同时写到外部持久化存储中。

10.3.2 Checkpointing优化

1.最小时间间隔

当Flink应用开启Checkpointing功能，并配置Checkpointing时间间隔，应用中就会根据指定的时间间隔周期性地对应用进行Checkpointing操作。默认情况下Checkpointing操作都是同步进行，也就是说，当前面触发的Checkpointing动作没有完全结束时，之后的Checkpointing操作将不会被触发。在这种情况下，如果Checkpointing过程持续的时间超过了配置的时间间隔，就会出现排队的情况。如果有非常多的Checkpointing操作在排队，就会占用额外的系统资源用于Checkpointing，此时用于任务计算的资源将会减少，进而影响到整个应用的性能和正常执行。

在这种情况下，如果大状态数据确实需要很长的时间来进行Checkpointing，那么只能对Checkpointing的时间间隔进行优化，可以通过Checkpointing之间的最小间隔参数进行配置，让Checkpointing之间根据Checkpointing执行速度进行调整，前面的Checkpointing没有完全结束，后面的Checkpointing操作也不会触发。

```
StreamExecutionEnvironment.getCheckpointConfig().setMinPauseBetweenCheckpoints(milliseconds)
```

通过最小时间间隔参数配置，可以降低Checkpointing对系统的性能影响，但需要注意的是，对于非常大的状态数据，最小时间间隔只能减轻Checkpointing之间的堆积情况。如果不能有效快速地完成Checkpointing，将会导致系统Checkpointing频次越来越低，当系统出现问题时，没有及时对状态数据有效地持久化，可能会导致系统丢失数据。因此，对于非常大的状态数据而言，应该对Checkpointing过程进行优化和调整，例如采用增量Checkpointing的方法等。

 注意

用户可以通过配置CheckpointConfig中setMaxConcurrentCheckpoints()方法设定并行执行的Checkpoints数量，这种方式也能有效降低Checkpointing堆积的问题，但会提高Checkpointing的资源占用。同时，如果开启了并行Checkpointing操作，当用户以手动方式触发Savepoint的时候，Checkpoint操作也将继续执行，这将影响到Savepoint过程中对状态数据的持久化。

2. 状态容量预估

除了对已经运行的任务进行Checkpointing优化，对整个任务需要的状态数据量进行预估也非常重要，这样才能选择合适的Checkpointing策略。对任务状态数据存储的规划依赖于如下基本规则：

· 正常情况下应该尽可能留有足够的资源来应对频繁的反压。

· 需要尽可能提供给额外的资源，以便在任务出现异常中断的情况下处理积压的数据。这些资源的预估都取决于任务停止过程中数据的积压量，以及对任务恢复时间的要求。

· 系统中出现临时性的反压没有太大的问题，但是如果系统中频繁出现临时性的反压，例如下游外部系统临时性变慢导致数据输出速率下降，这种情况就需要考虑给予算子一定的资源。

· 部分算子导致下游的算子的负载非常高，下游的算子完全是取决于上游算子的输出，因此对类似于窗口算子的估计也将会影响到整个任务的执行，应该尽可能给这些算子留有足够的资源以应对上游算子产生的影响。

3. 异步Snapshot

默认情况下，应用中的Checkpointing操作都是同步执行的，在条件允许的情况下应该尽可能地使用异步的Snapshotted，这样将大幅度提升Checkpointing的性能，尤其是在非常复杂的流式应用中，如多数据源关联、Co-functions操作或Windows操作等，都会有较好的性能改善。

在使用异步快照前需要确认应用遵循以下两点要求：

· 首先必须是Flink托管状态，即使用Flink内部提供的托管状态所对应的数据结构，例如常用的有ValueState、ListState、ReducingState等类型状态。

· StateBackend必须支持异步快照，在Flink1.2的版本之前，只有RocksDB完整地支持异步的Snapshots操作，从Flink1.3版本以后可以在heap-based StateBackend中支持异步快照功能。

4. 状态数据压缩

Flink中提供了针对Checkpoints和Savepoints的数据进行压缩的方法，目前Flink仅支持通过用Snappy压缩算法对状态数据进行压缩，在未来的版本中Flink将支持其他压缩算法。在压缩过程中，Flink的压缩算法支持Key-Group层面压缩，也是就是不同的Key-Group分别被压缩成不同的部分，因此解压缩过程可以并行进行，这对大规模数据的压缩和解压缩带来非常高的性能提升和较强的可拓展性。Flink中使用的压缩算法在ExecutionConfig中进行指定，通过将setUseSnapshotCompression方法中的值设定为true即可。

```
ExecutionConfig executionConfig = new ExecutionConfig();
executionConfig.setUseSnapshotCompression(true);
```

5. Checkpoint Delay Time

Checkpoints延时启动时间并不会直接暴露在客户端中，而是需要通过以下公式计算得出。如果该时间过长，则表明算子在进行Barries对齐，等待上游的算子将数据写入到当前算子中，说明系统正处于一个反压状态下。Checkpoint Delay Time可以通过整个端到端的计算时间减去异步持续的时间和同步持续的时间得出。

```
checkpoint_start_delay = end_to_end_duration -
synchronous_duration - asynchronous_duration
```

10.4 Flink内存优化

在大数据领域，大多数开源框架（Hadoop、Spark、Storm）都是基于JVM运行，但是JVM的内存管理机制往往存在着诸多类似OutOfMemoryError的问题，主要是因为创建过多的对象实例而超过JVM的最大堆内存限制，却没有被有效回收掉，这在很大程度上影响了系统的稳定性，尤其对于大数据应用，面对大量的数据对象产生，仅仅靠JVM所提供的各种垃圾回收机制很难解决内存溢出的问题。在开源框架中有很多框架都实现了自己的内存管理，例如Apache Spark的Tungsten项目，在一定程度上减轻了框架对JVM垃圾回收机制的依赖，从而更好地使用JVM来处理大规模数据集。

如图10-9所示，Flink也基于JVM实现了自己的内存管理，将JVM根据内存区分为Unmanned Heap、Flink Managed Heap、Network Buffers三个区域。在Flink内部对Flink Managed Heap进行管理，在启动集群的过程中直接将堆内存初始化成Memory Pages Pool，也就是将内存全部以二进制数组的方式占用，形成虚拟内存使用空间。新创建的对象都是以序列化成二进制数据的方式存储在内存页面池中，当完成计算后数据对象Flink就会将Page置空，而不是通过JVM进行垃圾回收，保证数据对象的创建永远不会超过JVM堆内存大小，也有效地避免了因为频繁GC导致的系统稳定性问题。

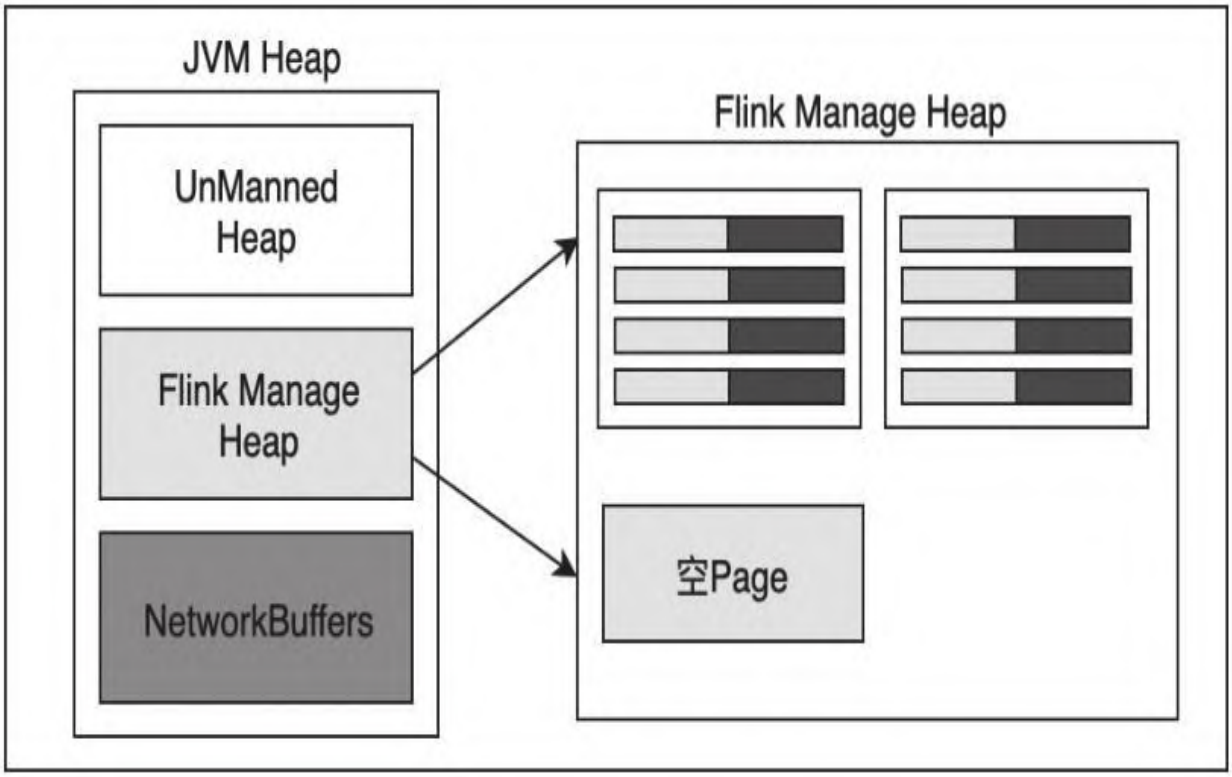

图10-9　Flink内存管理

10.4.1 Flink内存配置

1. JobManager配置

以下分别针对JobManager和TaskManager组件内存配置进行说明。

JobManager在Flink系统中主要承担管理集群资源、接收任务、调度Task、收集任务状态以及管理TaskManager的功能，JobManager本身并不直接参与数据的计算过程中，因此JobManager的内存配置项不是特别多，只要指定JobManager堆内存大小即可。

· jobmanager.heap.size：设定JobManager堆内存大小，默认为1024MB。

2. TaskManager配置

TaskManager作为Flink集群中的工作节点，所有任务的计算逻辑均执行在TaskManager之上，因此对TaskManager内存配置显得尤为重要，可以通过以下参数配置对TaskManager进行优化和调整。

· taskmanager.heap.size：设定TaskManager堆内存大小，默认值为1024M，如果在Yarn的集群中，TaskManager取决于Yarn分配给TaskManager Container的内存大小，且Yarn环境下一般会减掉一部分内存用于Container的容错。

· taskmanager.jvm-exit-on-oom：设定TaskManager是否会因为JVM发生内存溢出而停止，默认为false，当TaskManager发生内存溢出时，也不会导致TaskManager停止。

· taskmanager.memory.size：设定TaskManager内存大小，默认为0，如果不设定该值将会使用taskmanager.memory.fraction作为内存分配依据。

· taskmanager.memory.fraction：设定TaskManager堆中去除Network Buffers内存后的内存分配比例。该内存主要用于TaskManager任务排

序、缓存中间结果等操作。例如，如果设定为0.8，则代表TaskManager保留80%内存用于中间结果数据的缓存，剩下20%的内存用于创建用户定义函数中的数据对象存储。注意，该参数只有在taskmanager.memory.size不设定的情况下才生效。

· taskmanager.memory.off-heap：设置是否开启堆外内存供Managed Memory或者Network Buffers使用。

· taskmanager.memory.preallocate：设置是否在启动TaskManager过程中直接分配TaskManager管理内存。

· taskmanager.numberOfTaskSlots：每个TaskManager分配的slot数量。

10.4.2 Network Buffers配置

Flink将JVM堆内存切分为三个部分，其中一部分为Network Buffers内存。Network Buffers内存是Flink数据交互层的关键内存资源，主要目的是缓存分布式数据处理过程中的输入数据。例如在Repartitioning和Broadcating操作过程中，需要消耗大量的Network Buffers对数据进行缓存，然后才能触发之后的操作。通常情况下，比较大的Network Buffers意味着更高的吞吐量。如果系统出现“Insufficient number of network buffers”的错误，一般是因为Network Buffers配置过低导致，因此，在这种情况下需要适当调整TaskManager上Network Buffers的内存大小，以使得系统能够达到相对较高的吞吐量。

目前Flink能够调整Network Buffer内存大小的方式有两种：一种是通过直接指定Network Buffers内存数量的方式，另外一种是通过配置内存比例的方式。

1.设定Network Buffer内存数量

直接设定Nework Buffer数量需要通过如下公式计算得出：

```
NetworkBuffersNum = total-degree-of-parallelism * intra-node-parallelism * n
```

其中total-degree-of-parallelism表示每个TaskManager的总并发数量，intra-node-parallelism表示每个TaskManager输入数据源的并发数量，n表示在预估计算过程中Repar-titioning或Broadcasting操作并行的数量。intra-node-parallelism通常情况下与Task-Manager的所占有的CPU数一致，且Repartitioning和Broadcating一般下不会超过4个并发。可以将计算公式转化如下：

```
NetworkBuffersNum  = <slots-per-TM>^2 * < TMs>* 4
```

其中slots-per-TM是每个TaskManager上分配的slots数量，TMs是TaskManager的总数量。对于一个含有20个TaskManager，每个TaskManager含有8个Slot的集群来说，总共需要的Network Buffer数量为8^2*20*4=5120个，因此集群中配置Network Buffer内存的大小约为300M较为合适。

计算完Network Buffer数量后，可以通过添加如下两个参数对Network Buffer内存进行配置。其中segment-size为每个Network Buffer的内存大小，默认为32KB，一般不需要修改，通过设定numberOfBuffers参数以达到计算出的内存大小要求。

· taskmanager.network.numberOfBuffers：指定Network堆栈Buffer内存块的数量。

· taskmanager.memory.segment-size.：内存管理器和Network栈使用的内存Buffer大小，默认为32KB。

2. 设定Network内存比例

在1.3版本以前，设定Network Buffers内存大小需要通过上面的方式进行，显然相对比较繁琐。从1.3版本开始，Flink就提供了通过指定内存比例的方式设置Network Buffer内存大小，其涵盖的配置参数如下。

· taskmanager.network.memory.fraction: JVM中用于Network Buffers的内存比例。

· taskmanager.network.memory.min: 最小的Network Buffers内存大小，默认为64MB。

· taskmanager.network.memory.max: 最大的Network Buffers内存大小，默认1GB。

· taskmanager.memory.segment-size: 内存管理器和Network栈使用的Buffer大小，默认为32KB。

注意

目前Flink已经将直接设定Network Buffer内存大小的方式标记为@deprecated，也就是说未来的版本中可能会移除这种配置方式，因此建议用户尽可能采用按比例配置的方式。

10.5　本章小结

本章从Flink集群监控和优化的角度对Flink进行了比较深入的介绍。

其中10.1节介绍了如何对Flink任务进行有效的监控，其中包括Flink所提供的监控指标的获取，以及如何通过使用Reporter将监控指标推送到外部系统，帮助用户更好地了解Flink任务运行状况。10.2节和10.3节分别介绍了Flink流式任务常用的监控和优化策略，其中包括反压过程和Checkpointing过程的监控和配置。10.4节介绍了Flink内存方面的优化，其中包括对JobManager和TaskManager内存以及Network Buffers内存在配置方面的优化。