

小白学院 <http://www.xueai8.com>

预览版

Flink 实用教程

基于 Flink 1.13, Scala 和 Java 双语实现

面向零基础小白的大数据入门教程

预览版



只要努力
蜗牛也能爬到金字塔的顶端

前 言

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

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

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

Apache Flink 作为当前最热门的实时计算框架，是从业人员以及希望进入大数据行业人员必须要学习和掌握的大数据技术之一。但是作为大数据的初学者，在学习 Flink 时通常会遇到以下几个难题：

- ☐ 缺少面向零基础小白的 Flink 入门教程。
- ☐ 缺少系统化的 Flink 大数据教程。
- ☐ 现有的 Flink 资料、教程或图书过时陈旧。
- ☐ 官方全英文文档难以阅读和理解。
- ☐ 缺少必要的数据集、可运行的实验案例及学习平台。
- ☐

特别是 Apache Flink 从 2019 年被阿里巴巴收购以后，进入快速版本迭代时代，不但版本更新快，而且 API 变化频繁，笔者在应用和研究 Apache Flink 时，每当遇到 Flink 版本更新，就不得不花费大量精力重构已经的代码。

为此，既是为了自己能更系统更及时地跟进 Flink 的演进和迭代，另一方面也是为了（感同身受地）解决面向零基础小白学习 Flink（以及其他大数据技术）的入门难度，编写了这一本《Flink 实用教程》。个人以为，本书具有以下几个特点：

- ☐ 面向零基础小白，知识点深浅适当，代码完整易懂。
- ☐ 内容全面系统，包括架构原理、开发环境及程序部署、流和批计算等，并特别包含了第 7 章“基于 Flink 构建批流一体数仓”和第 8 章“基于 Flink 和数据湖构建实时数仓”内容。
- ☐ 版本先进，所有代码均基于 Flink 1.13.2。
- ☐ 双语实现，所有示例、案例，均包含 Scala 和 Java 两种语言版本的实现(极个别不支持双语的除外)。

个人认为，本书特别适合想要入门 Apache Flink、流计算的同学、希望系统大数据参考教材的老师以及想要了解最新 Flink 版本应用的从业人员。

当然，因为水平所限，行文以内容难免错误，请大家见谅，并予以反馈，我会在后续的版本重构中不断提升内容质量。

本书导学

为了读者能更好地利用本书，特别给出以下学习建议。

❑ 本书只提供电子版。

一般来说，纸质图书的出版周期较长，而大数据技术更新很快，特别是阿里收购以后的 Apache Flink。因此为了能及时跟进 Flink 的最新版本，本书只提供电子 版本。（当然，如果有幸有哪位出版社的编辑看上本书，作者本人也特别乐意进一步合作）

❑ 本书正式版本提供书中全部代码。

本书正式版会配套提供书中所有经过测试的 Scala 和 Java 代码，以及数据集和教学视频。最好的学习方法就是动手实践。

❑ 配书依赖 www.xueai8.com 提供的个人大数据学习平台 PBLP。

大数据的学习，最大的拦路虎其实是大数据平台。很多初学者花费大量的时间在大数据平台 and 环境的搭建上，并且在运行时经常出现莫名其妙的问题。即使好不容易自己搭建了一个大数据平台，却与教程、教材、学习资料等的运行环境不匹配或不兼容，学习起来磕磕绊绊。

个人大数据学习平台（PBLP, Personal Big Data Learning Platform）是 www.xueai8.com 依据开源大数据框架搭建好的大数据学习平台（目前为 Apache Hadoop 3.2.2 和 Flink 1.13.2），并搭载了本教程中所有代码和案例所使用的数据集，同时本教程的所有代码、案例、数据集路径、操作截图等，均基于此 PBLP 大数据平台测试和运行。读者可以直接下载此平台，在 VMWare 15+ 以上打开，即可轻松运行本书中所有代码和案例。

当然，对于有经验的用户，或者已经熟悉了本书内容之后，可以随意修改此平台的安装配置，因为它是完全使用开源大数据组件搭建而成的。

PBLP 下载地址，请大家访问 <http://www.xueai8.com> 首页相关的下载链接。

❑ 本书提供如下答疑和反馈渠道。

QQ: 185314368。如有任何疑问及问题反馈，都可以加此 QQ 进行咨询。

❑ 如何获取本书的最新版本。

本书会根据 Apache Flink 版本的更新迭代而定期更新。如果想要获取本书的最新版本，请访问 <http://www.xueai8.com>，本书的最新版本会及时在该网站上发布。

❑ 如何获取本书的正式版和配套代码。

如果您拿到的是配的预览版本，并感觉本书对您有一定的帮助，那么可以从 <http://www.xueai8.com> 获取本书正式版及配套代码（Scala + Java）下载地址。请注意：本书正式版是收费版本。

目 录

第 1 章 Flink 架构与集群安装.....	- 1 -
1.1 Flink 简介.....	- 1 -
1.1.1 Flink 发展历程.....	- 1 -
1.1.2 Flink 特性.....	- 2 -
1.2 Flink 应用场景.....	- 3 -
1.2.1 事件驱动应用程序.....	- 3 -
1.2.2 数据分析应用程序.....	- 4 -
1.2.3 数据管道应用程序.....	- 5 -
1.3 Flink 体系架构.....	- 5 -
1.3.1 Flink 系统架构.....	- 5 -
1.3.2 Flink 运行时架构.....	- 6 -
1.3.3 Flink 资源管理.....	- 8 -
1.3.4 Flink 作业调度.....	- 13 -
1.3.5 Flink 故障恢复.....	- 16 -
1.3.6 Flink 程序执行模式.....	- 17 -
1.4 Flink 集群安装.....	- 20 -
1.4.1 Flink 独立集群安装和测试.....	- 20 -
1.4.2 Flink 完全分布式集群安装.....	- 24 -
1.4.3 Flink 常用命令.....	- 30 -
1.5 本章小结.....	- 32 -
1.6 本章习题.....	- 32 -
第 2 章 Flink 开发准备.....	- 33 -
2.1 安装和配置 Maven.....	- 33 -
2.1.1 安装和配置 Maven.....	- 33 -
2.1.2 修改 Maven 本地仓库默认位置.....	- 35 -
2.1.3 创建初始模板项目.....	- 37 -
2.1.4 构建打包项目.....	- 39 -
2.2 使用 IntelliJ IDEA+Maven 开发 Flink 项目.....	- 39 -
2.2.1 在 IntelliJ IDEA 中创建 Flink 项目.....	- 40 -
2.2.2 设置项目基本依赖.....	- 44 -
2.2.3 用于构建具有依赖项的 Jar 的模板.....	- 45 -
2.2.4 编写批处理代码并测试执行.....	- 46 -
2.2.5 项目打包并提交 Flink 集群执行.....	- 48 -
2.3 Flink 相关概念.....	- 49 -
2.3.1 Flink 数据流.....	- 50 -
2.3.2 Flink 分层 API.....	- 50 -
2.4 本章小结.....	- 52 -
2.5 本章习题.....	- 52 -
第 3 章 开发 Flink 实时数据处理程序.....	- 53 -
3.1 Flink 流处理程序编程模型.....	- 53 -
3.1.1 流数据类型.....	- 53 -
3.1.2 应用程序实现.....	- 56 -
3.1.3 应用程序剖析.....	- 59 -

3.2 数据源.....	- 63 -
3.2.1 基于 Socket 的数据源.....	- 63 -
3.2.2 基于文件的数据源.....	- 66 -
3.2.3 基于集合的数据源.....	- 69 -
3.2.4 自定义数据源.....	- 71 -
3.3 数据转换.....	- 88 -
3.3.1 map 转换.....	- 88 -
3.3.2 flatMap 转换.....	- 89 -
3.3.3 filter 转换.....	- 91 -
3.3.4 keyBy 转换.....	- 94 -
3.3.5 reduce 转换.....	- 100 -
3.3.6 聚合转换.....	- 102 -
3.3.7 union 转换.....	- 103 -
3.3.8 connect 转换.....	- 106 -
3.3.9 coMap, coFlatMap 转换.....	- 110 -
3.3.10 iterate 转换.....	- 113 -
3.3.11 project 转换.....	- 116 -
3.4 数据分区.....	- 117 -
3.4.1 流数据分发模式.....	- 117 -
3.4.2 数据分区方法.....	- 119 -
3.4.3 数据分区示例.....	- 135 -
3.4.4 理解操作符链.....	- 147 -
3.5 数据接收器.....	- 148 -
3.5.1 内置数据接收器.....	- 148 -
3.5.2 使用流文件连接器.....	- 151 -
3.5.3 自定义数据接收器.....	- 157 -
3.6 时间和水印.....	- 162 -
3.6.1 时间概念.....	- 163 -
3.6.2 事件时间和水印.....	- 165 -
3.6.3 水印策略.....	- 166 -
3.6.4 处理空闲数据源.....	- 169 -
3.6.5 编写水印生成器.....	- 170 -
3.6.6 内置水印生成器.....	- 175 -
3.6.7 分配时间戳和水印示例.....	- 176 -
3.7 窗口操作.....	- 186 -
3.7.1 理解 Flink 窗口概念.....	- 186 -
3.7.2 窗口分配器.....	- 188 -
3.7.3 窗口函数.....	- 202 -
3.7.4 触发器.....	- 222 -
3.7.5 清除器.....	- 223 -
3.7.6 处理迟到数据.....	- 224 -
3.7.7 处理窗口结果.....	- 225 -
3.7.8 窗口连接.....	- 226 -
3.8 低级操作.....	- 232 -
3.8.1 ProcessFunction.....	- 232 -
3.8.2 KeyedProcessFunction 示例.....	- 235 -
3.8.3 案例：服务器故障检测报警程序.....	- 245 -
3.9 状态和容错.....	- 246 -
3.9.1 状态运算.....	- 246 -
3.9.2 状态的类型.....	- 247 -

3.9.3 使用托管的 Keyed State.....	249
3.9.4 使用托管 Operator State.....	253
3.9.5 广播状态 (Broadcast State)	254
3.9.6 状态后端.....	257
3.9.7 检查点存储.....	259
3.9.8 状态快照.....	262
3.10 侧输出流.....	265
3.10.1 什么是侧输出流.....	265
3.10.2 侧输出流应用示例.....	266
3.11 流连接器.....	269
3.11.1 Kafka 连接器.....	270
3.11.2 JDBC 连接器.....	278
3.12 其他.....	279
3.12.1 在应用程序中使用日志.....	280
3.12.2 使用 ParameterTool.....	280
3.12.3 命名大型 TupleX 类型.....	281
3.13 Flink 流处理案例.....	281
3.13.1 处理 IOT 事件流.....	281
3.13.2 信用卡欺诈检测.....	283
3.13.3 订单延迟支付告警程序.....	289
3.13.4 运输公司车辆超速实时监测.....	290
3.13.5 计算实时热门商品.....	292
第 4 章 开发 Flink 批数据处理程序.....	299
4.1 Flink 批处理程序编程模型.....	299
4.1.1 应用程序实现.....	299
4.1.2 应用程序剖析.....	301
4.2 数据源.....	303
4.2.1 基于文件的数据源.....	303
4.2.2 基于集合的数据源.....	305
4.2.3 通用的数据源.....	306
4.2.4 压缩文件.....	306
4.3 数据转换.....	307
4.3.1 map 转换.....	307
4.3.2 flatMap 转换.....	308
4.3.3 mapPartition 转换.....	309
4.3.4 filter 转换.....	311
4.3.5 reduce 转换.....	312
4.3.6 在分组数据集上的 reduce 转换.....	313
4.3.7 在分组数据集上的 GroupReduce.....	315
4.3.8 在分组数据集上的 GroupCombine.....	317
4.3.9 在分组元组数据集上执行聚合.....	320
4.3.10 在分组元组数据集上执行 minBy.....	322
4.3.11 在分组元组数据集上执行 maxBy.....	323
4.3.12 在全部元组数据集上执行聚合操作.....	324
4.3.13 distinct 转换.....	325
4.3.14 join 连接转换.....	327
4.3.15 union 转换.....	329
4.3.16 project 转换.....	331
4.3.17 first-n 转换.....	332
4.4 数据接收器.....	333

4.4.1 计算结果保存到文本文件.....	333
4.4.2 计算结果保存到 JDBC.....	335
4.4.3 标准 DataSink 方法.....	336
4.4.4 本地排序输出.....	337
4.5 广播变量.....	338
4.6 分布式缓存.....	339
4.7 向函数传递参数.....	343
4.7.1 通过构造函数传参.....	343
4.7.2 通过 withParameters(Configuration)传参.....	344
4.7.3 通过 ExecutionConfig 传递全局参数.....	346
4.8 数据集中的元素拉链操作.....	348
4.8.1 密集索引.....	348
4.8.2 唯一索引.....	350
4.9 Flink 批处理示例.....	352
4.9.1 分析豆瓣热门电影数据集_Scala 实现.....	353
4.9.2 分析豆瓣热门电影数据集_Java 实现.....	353
4.10 小结.....	355
第 5 章 使用 Table API 进行数据处理.....	356
5.1 依赖.....	356
5.2 Table API & SQL 程序编程模式.....	357
5.2.1 TableEnvironment.....	357
5.2.2 Table API & SQL 程序的结构.....	360
5.2.3 在 Catalog 中创建表.....	361
5.2.4 查询表.....	364
5.2.5 发出一个表.....	367
5.2.6 翻译并执行查询.....	369
5.3 Table API.....	374
5.3.1 关系运算.....	374
5.3.2 窗口运算.....	384
5.3.3 基于行的操作.....	390
5.4 Table API 与 DataStream API 集成.....	396
5.4.1 依赖.....	396
5.4.2 在 DataStream 和 Table 之间转换.....	397
5.4.3 处理 insert-only 流.....	402
5.4.4 处理变更日志流.....	412
5.5 Table API 实时流处理案例.....	420
5.5.1 传感器温度实时统计.....	421
5.5.2 车辆超速实时监测.....	422
5.5.3 电商用户行为实时分析.....	423
第 6 章 Flink on YARN.....	427
6.1 Flink on YARN session.....	427
6.1.1 下载 Flink 集成 Hadoop 依赖包.....	427
6.1.2 运行 Flink on YARN session.....	428
6.1.3 提交 Flink 作业.....	430
6.1.4 停止 Flink on YARN session.....	434
6.2 Flink on YARN 支持的部署模式.....	434
6.2.1 Application 模式.....	434
6.2.2 Per-Job 集群模式.....	434
6.2.3 Session 模式.....	435
6.3 小结.....	436

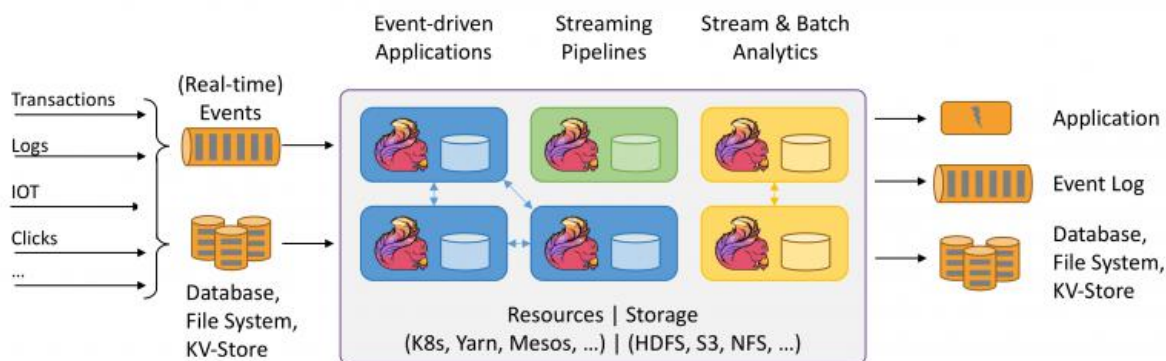
第 7 章 基于 Flink 构建流批一体数仓.....	437 -
7.1 Flink 集成 Hive.....	437 -
7.1.1 Flink 集成 Hive 的方式.....	437 -
7.1.2 Flink 集成 Hive 的步骤.....	437 -
7.1.3 Flink 连接 Hive 模板代码.....	440 -
7.2 批流一体数仓构建实例.....	441 -
7.2.1 数据集说明.....	441 -
7.2.2 创建 Flink 项目.....	442 -
7.2.3 创建执行环境.....	447 -
7.2.4 注册 HiveCatalog.....	448 -
7.2.5 创建 Kafka 流表.....	449 -
7.2.6 创建 Hive 表.....	451 -
7.2.7 流写 Hive 表.....	454 -
7.2.8 动态读取 Hive 流表.....	456 -
7.2.9 完整示例代码.....	456 -
7.2.10 执行步骤.....	457 -
7.3 纯 SQL 构建批流一体数仓.....	458 -
7.3.1 使用 Flink SQL 客户端.....	458 -
7.3.2 集成 Flink SQL Cli 和 Hive.....	461 -
7.3.3 注册 HiveCatalog.....	462 -
7.3.4 使用 SQL Client 提交作业.....	464 -
7.3.5 构建批流一体数仓完整过程.....	468 -
第 8 章 基于 Flink 和数据湖构建实时数仓.....	472 -
8.1 现代数据湖概述.....	472 -
8.1.1 什么是数据湖.....	472 -
8.1.2 数据湖架构.....	475 -
8.1.3 开源数据湖框架.....	475 -
8.2 基于 Flink + Iceberg 构建企业数据湖.....	477 -
8.2.1 Apache Iceberg 的优势.....	477 -
8.2.2 Apache Iceberg 经典业务场景.....	478 -
8.2.3 应用 Apache Iceberg 的准备工作.....	481 -
8.2.4 创建和使用 catalog.....	482 -
8.2.5 Iceberg DDL 命令.....	483 -
8.2.6 Iceberg SQL 查询.....	484 -
8.2.7 Iceberg SQL 写入.....	485 -
8.2.8 使用 DataStream 读取.....	485 -
8.2.9 使用 DataStream 写入.....	486 -
8.2.10 重写文件操作.....	487 -
8.2.11 未来改进.....	487 -
8.3 基于 Flink + Iceberg 构建准实时数仓.....	488 -
8.3.1 实时数仓构建.....	488 -
8.3.2 执行 OLAP 联机分析.....	489 -

第 1 章 Flink 架构与集群安装

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

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

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



1.1 Flink 简介

1.1.1 Flink 发展历程

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

注 1：Data Artisans 公司于 2019 年 1 月被阿里以 9000 万欧元收购。

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



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

Flink 是 Apache 软件基金会最大的 5 个大数据项目之一，在全球拥有超过 200 名开发人员的社区和多个生产安装。作为公认的新一代大数据计算引擎，Flink 已成为阿里、腾讯、滴滴、美团、字节跳动、Netflix、Lyft 等国内外知名公司建设流计算平台的首选！

部分使用 Apache Flink 的成功企业



1.1.2 Flink 特性

Flink 支持流和批处理、复杂的状态管理、事件时间处理语义，以及对状态的一次一致性保证。此外，Flink 可以部署在各种资源提供者(如 YARN、Apache Mesos 和 Kubernetes)上，也可以作为独立集群部署在裸机硬件上。可以将 Flink 集群配置为高可用的以避免单点故障。

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

Apache Flink 为用户提供了更强大的计算能力和更易用的编程接口：

- ❑ 批流统一。Flink 在 Runtime 和 SQL 层批流统一，提供高吞吐低延时计算能力和更强大的 SQL 支持。
- ❑ 生态兼容。Flink 能与 Hadoop Yarn / Apache Mesos / Kubernetes 集成，并且支持单机模式运行。
- ❑ 性能卓越。Flink 提供了性能卓越的批处理与流处理支持。
- ❑ 规模计算。Flink 的作业可被分解成上千个任务，分布在集群中并发执行。

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

1.2 Flink 应用场景

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

- ❑ 事件驱动类型，比如：信用卡交易、刷单、监控等；
- ❑ 数据分析类型，比如：库存分析、双 11 大屏等；
- ❑ 数据管道类型，也就是 ETL 场景，比如一些日志的解析等；

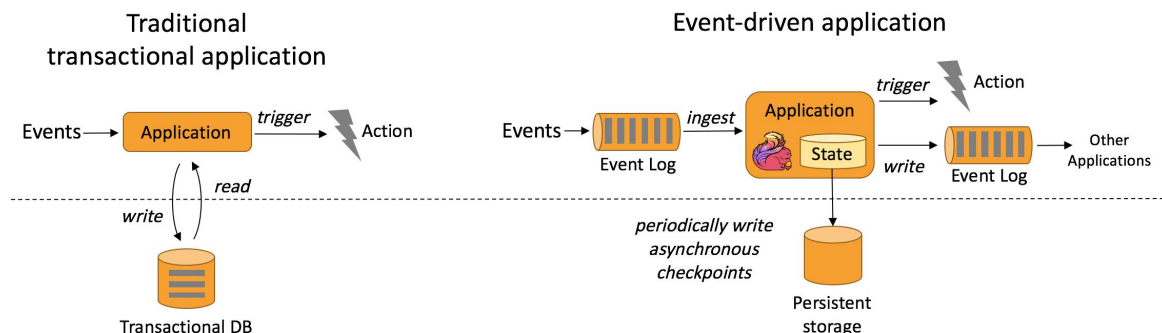
1.2.1 事件驱动应用程序

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

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

事件驱动的应用程序是传统应用程序（具有独立的计算层和数据存储层）设计的演化。在这种传统体系结构中，应用程序从远程事务数据库读取数据并将数据持久存储。

而事件驱动的应用程序基于有状态流来处理应用程序。在这种设计中，数据和计算是共存的，从而产生本地(内存或磁盘)数据访问。通过定期将检查点写入远程持久存储，可以实现容错。下图描述了传统应用程序体系结构和事件驱动应用程序之间的区别。



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

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

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

典型的事件驱动程序包括：

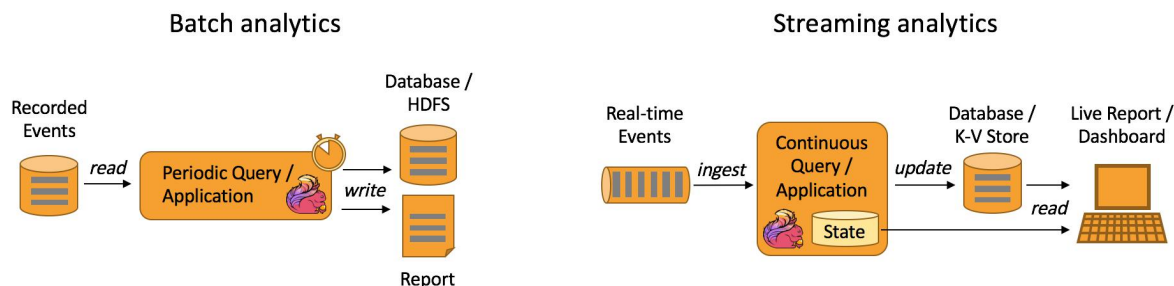
- ☐ 欺诈检测
- ☐ 异常检测
- ☐ 基于规则的提醒
- ☐ 业务流程监控
- ☐ Web 应用程序(社交网络)

1.2.2 数据分析应用程序

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

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

Apache Flink 支持流以及批处理分析应用程序，如下图所示：



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

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

典型的数据分析应用程序包括：

- ☐ 电信网络质量监控
- ☐ 移动应用程序中的产品更新及用户体验分析
- ☐ 消费者技术中实时数据的 Ad-hoc 分析
- ☐ 大规模图分析

1.2.3 数据管道应用程序

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

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

下图描述了周期性 ETL 作业和连续数据管道之间的区别：



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

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

典型的数据管道应用程序包括：

- ☐ 电子商务中的实时搜索索引构建
- ☐ 电子商务中的持续 ETL

1.3 Flink 体系架构

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

表 1.1 三种流计算框架比较

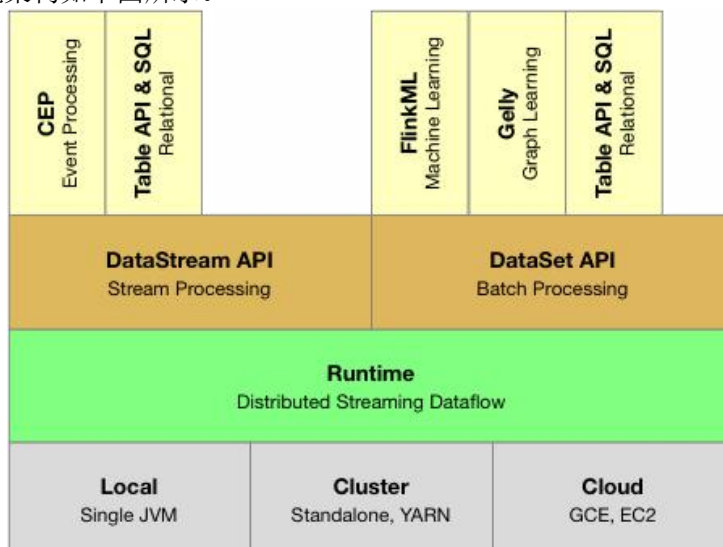
流处理框架	高吞吐量	低延迟	易于使用和表达	正确的时间/窗口语义	压力下保持正确性
Storm	×	√	×	×	×
Spark Streaming	√	×	×	×	√
Flink	√	√	√	√	√

1.3.1 Flink 系统架构

Flink 是可以运行在多种不同的环境中的，例如，它可以通过单进程多线程的方式直接运行，从而

提供调试的能力。它也可以运行在 Yarn 或者 K8S 这种资源管理系统上面，也可以在各种云环境中执行。

Flink 的整体系统架构如下图所示。

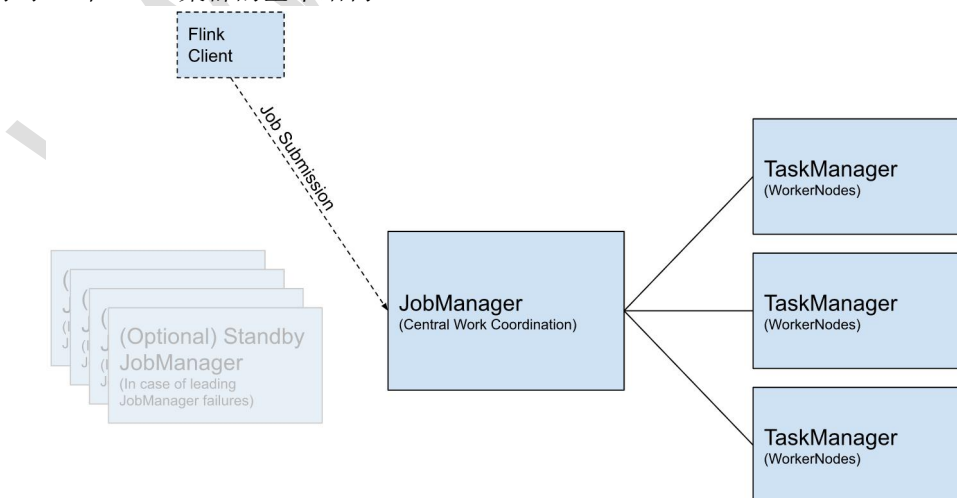


针对不同的执行环境，Flink 提供了一套统一的分布式作业执行引擎，也就是 Flink Runtime（Flink 运行时）这一层。Flink 在 Runtime 层之上提供了 DataStream 和 DataSet 两套 API，分别用来编写流作业与批作业，以及一组更高级的 API 来简化特定作业的编写。

1.3.2 Flink 运行时架构

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

Flink Runtime 层的整个架构采用了标准 Master-Slave 的结构，即总是由一个 Flink Master（即 JobManager）和一个或多个 Flink Slave（即 TaskManager）组成。Flink Runtime 层的主要架构如下图所示，它展示了一个 Flink 集群的基本结构。



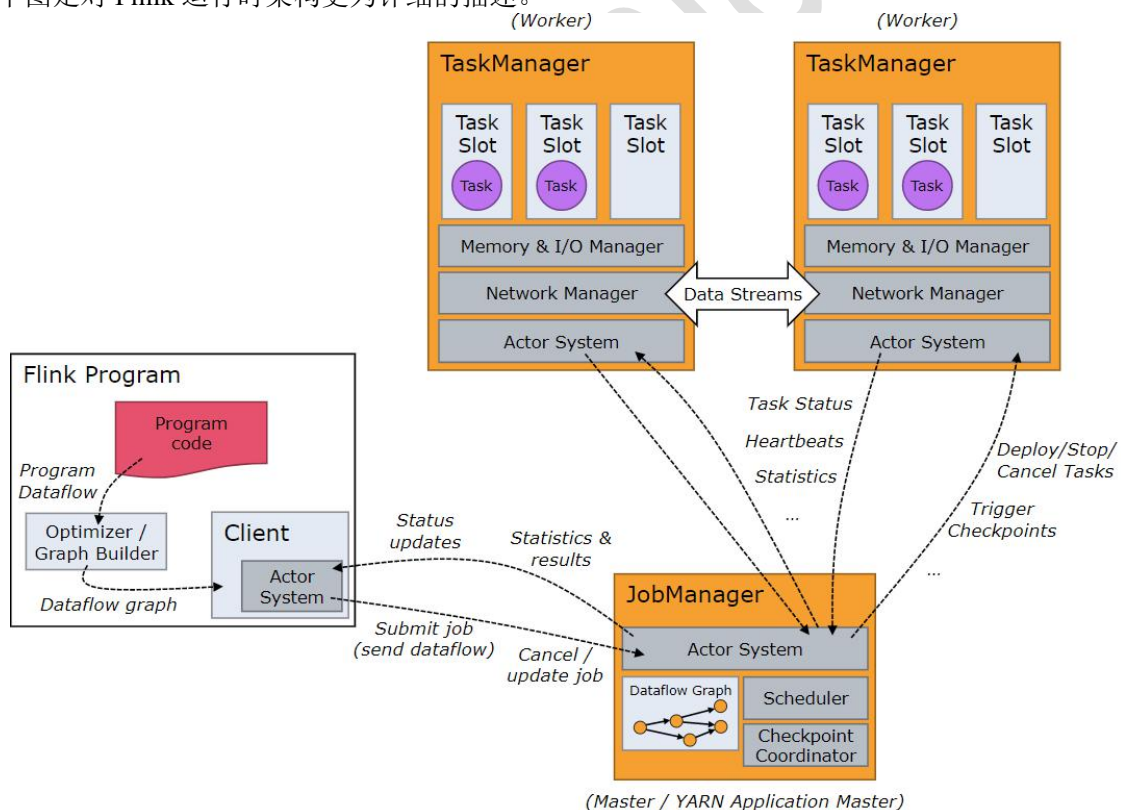
在部署 Flink 时，每个组件通常都有多个可用选项。我们已经在下面的表格中列出了它们。

组件	作用	实现
----	----	----

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

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

下图是对 Flink 运行时架构更为详细的描述。



JobManager 和 TaskManager 可以通过多种方式启动，例如，可以直接在机器上作为独立集群启动，也可以在容器中启动，或者由 YARN 或 Mesos 等资源框架管理。TaskManager 连接到 JobManager，宣布自己可用，并被分配工作。

JobManager

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

JobManager 是 Flink 集群的主进程，它由三个不同的组件组成：

❑ ResourceManager

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

❑ Dispatcher

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

❑ JobMaster

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

TaskManager

TaskManagers 是一个 Flink 集群的工作（worker）进程，负责执行数据流的任务，并缓冲和交换数据流。

- ❑ 必须始终至少有一个 TaskManager。
- ❑ TaskManager 中最小的资源调度单位是任务槽(task slot)。
- ❑ TaskManager 的任务槽(task slot)数量表示并发处理的任务数。注意，多个操作符可以在一个任务槽中执行。
- ❑ TaskManager 启动后，TaskManager 将其槽位注册到 ResourceManager。当得到 ResourceManager 的指示时，TaskManager 会提供一个或多个它的槽位给 JobMaster。
- ❑ 然后，JobMaster 可以将任务分配给这些槽以执行它们。
- ❑ 在执行过程中，一个 TaskManager 与运行同一应用程序任务的其他 TaskManager 交换数据。

1.3.3 Flink 资源管理

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

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

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

补充：逻辑上来说，每个 Slot 都应该有一个向量来描述它所能提供的各种资源的量，每个 Task 也

需要相应的说明它所需要的各种资源的量。但是实际上在 1.9 之前, Flink 是不支持细粒度的资源描述的, 而是统一的认为每个 Slot 提供的资源和 Task 需要的资源都是相同的。从 1.9 开始, Flink 开始增加对细粒度的资源匹配的支持的实现, 但这部分功能目前仍在完善中。

Task 执行

这是很难解释和理解的部分。在深入讲解之前, 我们应该记住什么是 Flink 中的操作符和任务。一般情况下, 请记住: $Operator = (Task_1 + Task_2 + \dots + Task_N)$ 。

- ❑ Operator in Flink => 将一个数据流转换为另一个数据流(可以是相同类型的数据流, 也可以是不同类型的数据流)。Operator (操作符) 是逻辑数据流图(也称为 JobGraph)的节点。
- ❑ Task in Flink => 是由 Flink 的运行时执行的基本工作单元。任务是物理数据流图(也称为 Execution Graph)的节点。
- ❑ 任务是操作符或操作符链的一个并行实例。(两个或多个连续操作符之间没有任何重分区)
- ❑ 在 Apache Flink 中, 每个 Task 只有一个线程, Task 之间没有共享知识。例如, Task₁ 不知道另一个 task 正在发生什么。这意味着 API 提供了可以从一个 Task 中访问每个状态段。没有办法访问其他线程中的状态。
- ❑ 还有子任务 (Sub-Task) 的概念。子任务是在数据流的一部分上工作的任务。子任务指的是同一个操作符或操作符链有多个并行任务。(这实际上意味着存在数据并行性)

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

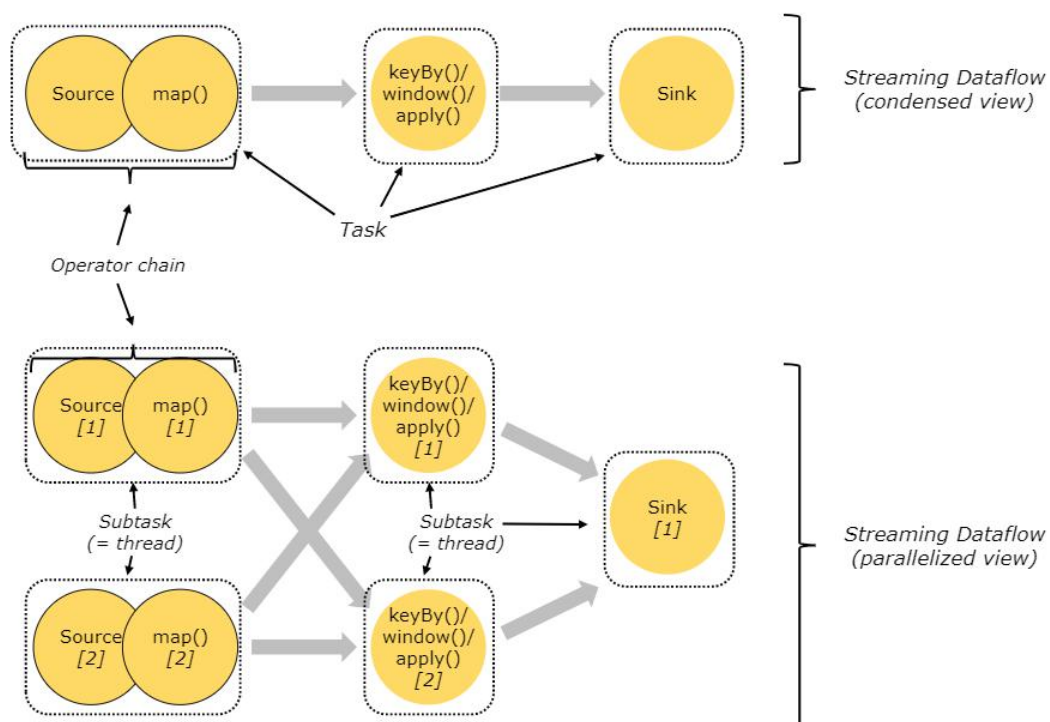
- ❑ TaskManager(worker/slave 进程)可以同时执行多个任务。并发数通常与 TaskManager 所在机器的 CPU 数有关。例如, 如果机器的 CPU 数是 16, 一个 TaskManager 可以同时运行 16 个任务。(这是 Apache Flink 提供的最佳解决方案。你可以同时运行+16 个任务, 但这可能会导致一些问题-Flink 集群可能会经常重新启动自己等等)
- ❑ 一个 TaskManager 在同一个 JVM 进程中以多线程方式执行它的任务。
- ❑ 任务可以是相同操作符(记住数据并行性)的子任务, 也可以是不同操作符(记住任务并行性)的子任务, 甚至可以是来自不同应用程序的子任务。
- ❑ TaskManager 提供一定数量的槽位(与机器的 CPU 数量相关)来控制它能够并发执行的任务。(换句话说, 如果机器有 16 个 CPU, 那么 TaskManager 可以有 16 个槽, 每个槽处理一个任务)

注意: 一个槽可以包含一个特定的任务或多个关联的任务。

操作符链

对于分布式执行, Flink 将操作子任务链成 tasks。每个 task 由一个线程执行。将操作符链接到 tasks 中是一种优化行为: 它减少了线程到线程切换和缓冲的开销, 并在降低延迟的同时增加了总体吞吐量。

下图中的样例数据流使用五个子任务执行, 因此使用五个并行线程。



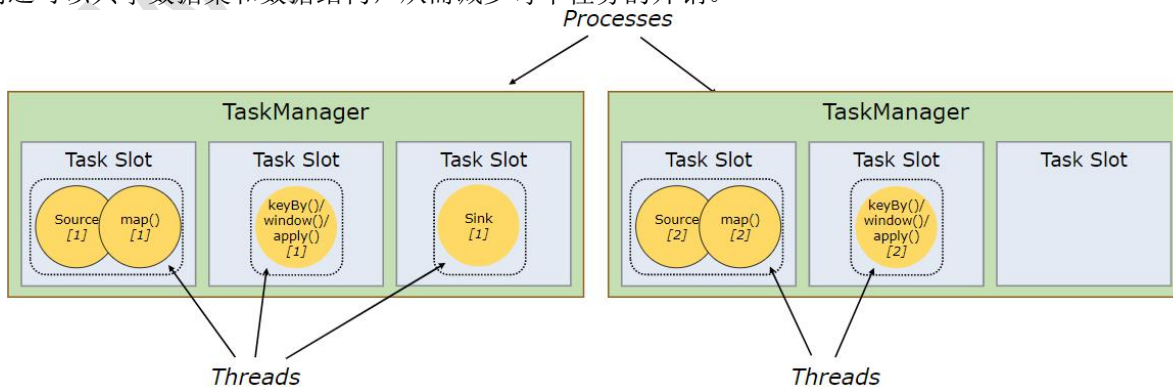
操作符链允许非 shuffle 操作在同一个线程中共存，完全避免了序列化和反序列化。

任务槽

每个 TaskManager (Worker) 都是一个 JVM 进程，可以在单独的线程中执行一个或多个子任务。为了控制一个 worker 接受多少任务，一个 worker 具有至少一个“任务插槽” (task slot)。

每个 task slot 表示 TaskManager 资源的一个固定子集。例如，一个有三个插槽的 TaskManager 会将其 1/3 的托管内存分配给每个插槽。对资源进行插槽化意味着子任务不会与来自其他作业的子任务争夺托管内存，而是拥有一定数量的预留托管内存。注意，这里没有发生 CPU 隔离；当前插槽只分隔任务的托管内存。

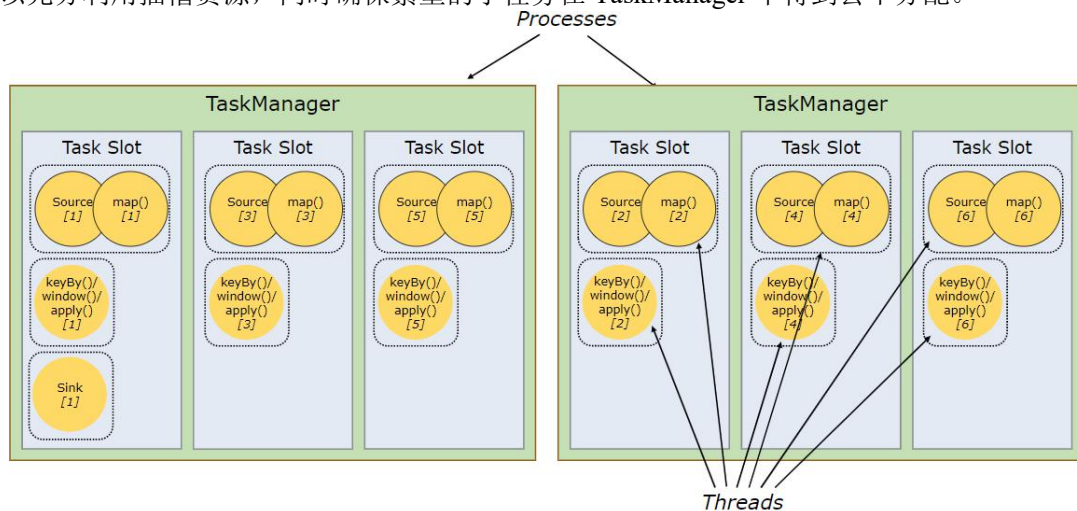
通过调整任务槽的数量，用户可以定义子任务如何彼此隔离。每个 TaskManager 有一个插槽 (Slot) 意味着每个任务组运行在各自的 JVM 中 (例如，可以在单独的容器中启动 JVM)。拥有多个插槽意味着更多的子任务共享同一个 JVM。相同 JVM 中的任务 (task) 共享 TCP 连接 (通过多路复用) 和心跳消息。它们还可以共享数据集和数据结构，从而减少每个任务的开销。



默认情况下，Flink 允许子任务共享槽位，即使它们是不同任务的子任务，只要它们来自相同的作

业。结果是一个槽位可以容纳作业的整个管道。这种槽位共享（slot sharing）有两个主要好处：

- ❑ Flink 集群需要的任务插槽与作业中使用的最高并行度一样多。不需要计算一个程序总共包含多少任务(具有不同的并行度)。
- ❑ 更容易得到更好的资源利用。如果没有槽位共享，非密集型 source/map()子任务将阻塞与资源密集型窗口子任务一样多的资源。使用插槽共享，将我们示例中的基本并行度从 2 提高到 6，可以充分利用插槽资源，同时确保繁重的子任务在 TaskManager 中得到公平分配。

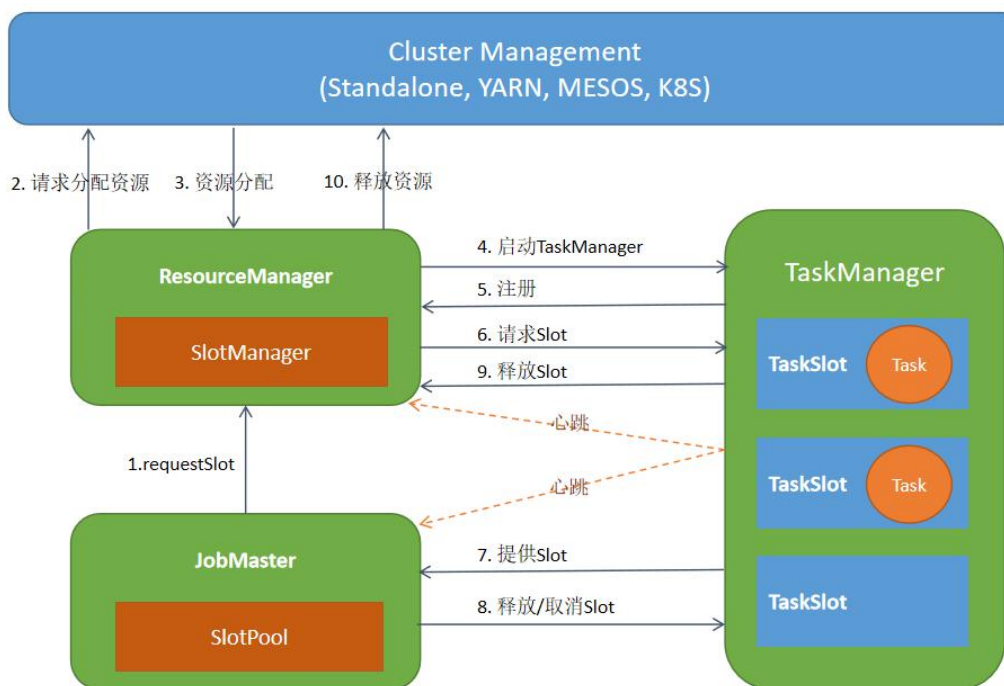


API 还包括一个资源组（resource group）机制，可用于防止不需要的槽位共享。

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

资源申请

在 ResourceManager 中，有一个子组件叫做 SlotManager，它维护了当前集群中所有 TaskManager 上的 Slot 的信息与状态，比如该 Slot 在哪个 TaskManager 中，该 Slot 当前是否空闲等。如下图所示：



当 JobMaster 为特定 Task 申请资源的时候，根据当前作业部署模式（关于作业部署模式，请参阅 1.5）的区别，TaskManager 可能已经启动或者尚未启动。如果 TaskManager 尚未启动，则 ResourceManager 会去申请资源来启动新的 TaskManager。当 TaskManager 启动之后，它会通过服务发现找到当前活跃的 ResourceManager 并进行注册。在注册信息中，会包含该 TaskManager 中所有 Slot 的信息。ResourceManager 收到注册信息后，其中的 SlotManager 就会记录下相应的 Slot 信息。当 JobMaster 为某个 Task 来申请资源时，SlotManager 就会从当前空闲的 Slot 中按一定规则选择一个空闲的 Slot 进行分配。当分配完成后，ResourceManager 会首先向 TaskManager 发送 RPC 要求将选定的 Slot 分配给特定的 JobMaster。TaskManager 如果还没有执行过该 JobMaster 的 Task 的话，它需要首先向相应的 JobMaster 建立连接，然后发送提供 Slot 的 RPC 请求。在 JobMaster 中，所有 Task 的请求会缓存到 SlotPool 中。当有 Slot 被提供之后，SlotPool 会从缓存的请求中选择相应的请求并结束相应的请求过程。

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

心跳报告

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

共享槽位

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

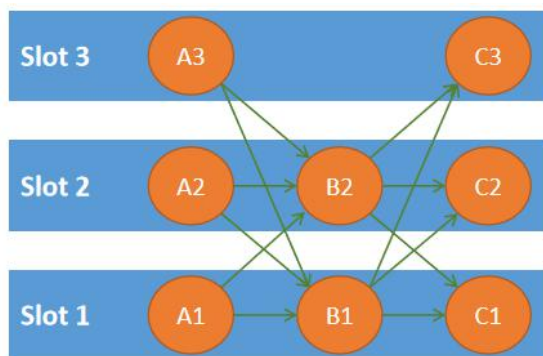
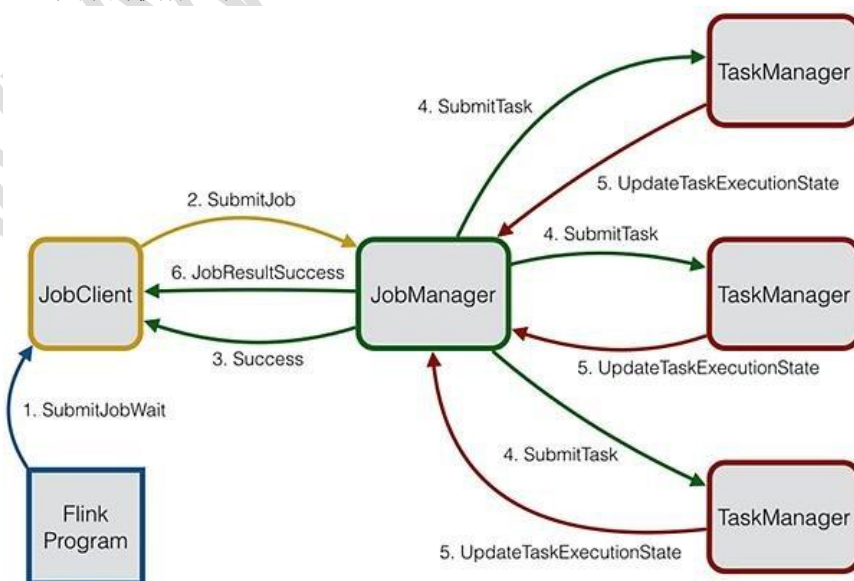


图 共享 Slot

1.3.4 Flink 作业调度

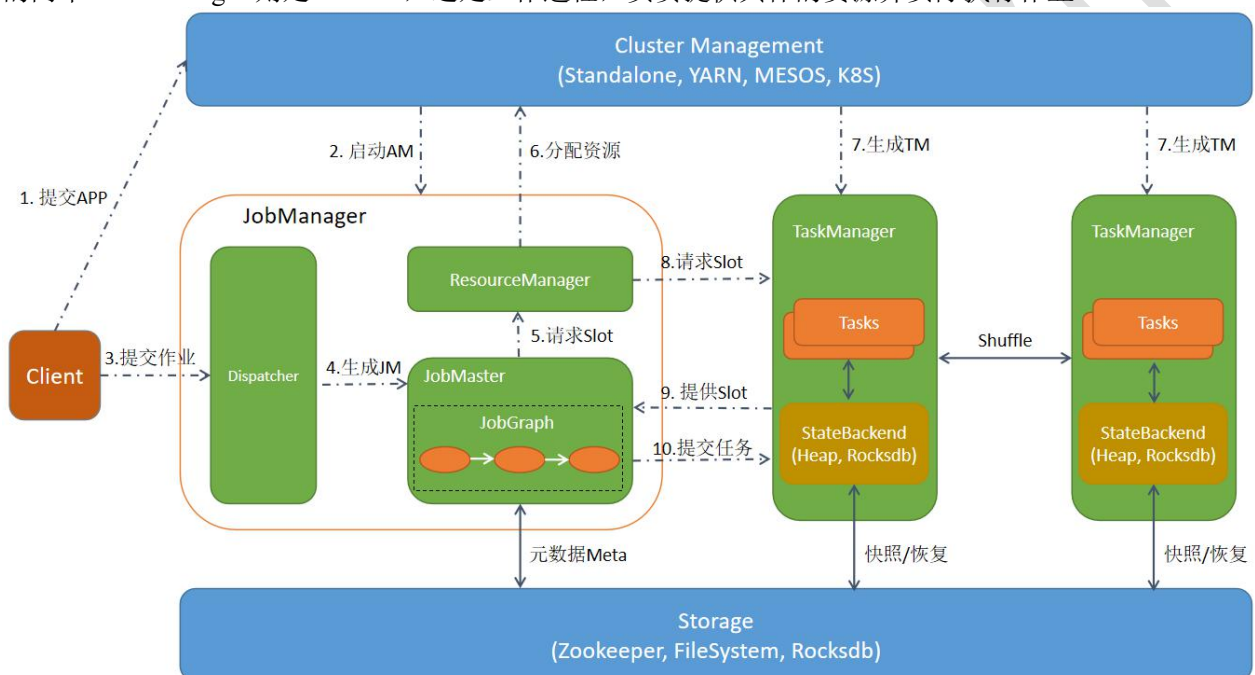
参与 Flink 程序执行的有多个进程，包括 JobManager、TaskManager 以及 JobClient。其中 JobManager 充当 Master 角色，TaskManager 充当 Worker 角色。JobClient 不是 Flink 任务执行过程的内部组件，而是执行过程的起始点。JobClient 负责接收用户提交的应用程序，创建对应的数据流，然后将数据流提交到 JobManager 上执行。一旦执行完毕，JobClient 将执行结果发回给用户。

下图展示了 Flink 程序的执行过程。



Flink 程序首先被提交到 JobClient 上，随后 JobClient 将它提交到 JobManager 上，JobManager 负责安排资源的分配和作业（job）的执行。首先是资源的分配，然后将作业（job）划分为若干任务（task）后提交到对应的 TaskManager 上。TaskManager 在接收到 task 后，初始化一个线程并开始执行程序。执行过程中 TaskManager 持续地将状态的变化情况报告给 JobManager，这些状态包括开始执行（starting the execution）、正在执行（in progress）以及完成（finished）。一旦作业（job）的执行彻底完成，JobManager 就将其结果发回 JobClient 端。

Flink 的分布式作业执行过程包含两个重要的角色，Master 和 Worker。在下面的架构图中，其中左侧的 JobManager 部分即是 Master，它负责管理整个集群中的资源并处理作业提交、作业监督；而右侧的两个 TaskManager 则是 Worker，这是工作进程，负责提供具体的资源并实际执行作业。



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

当作业提交到 Dispatcher 后，Dispatcher 会首先启动一个 JobMaster 组件，然后 JobMaster 会向 ResourceManager 申请资源来启动作业中具体的 task。这时根据作业部署模式（关于作业部署模式，请参阅 1.5）的区别，TaskManager 可能已经启动或者尚未启动。

- ❑ 如果是前者，此时 ResourceManager 中已有记录了 TaskManager 注册的资源，可以直接选取空闲资源进行分配。
- ❑ 否则，ResourceManager 也需要首先向外部资源管理系统申请资源来启动 TaskManager，然后等待 TaskManager 注册相应资源后再继续选择空闲资源进程分配。

目前 Flink 中 TaskManager 的资源是通过 Slot 来描述的，一个 Slot 一般可以执行一个具体的 task，但在一些情况下也可以执行多个相关联的 task。ResourceManager 选择到空闲的 Slot 之后，就会通知相应的 TaskManager “将该 Slot 分配给 JobMaster XX”，然后 TaskManager 进行相应的记录后，会向

JobManager 进行注册。JobManager 收到 TaskManager 注册上来的 Slot 后，就可以实际提交 task 了。

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

基于上一节中 Slot 管理和分配的逻辑，JobMaster 负责维护作业中 Task 执行的状态。如上文所述，客户端会向 JobMaster 提交一个 JobGraph，它代表了作业的逻辑结构。JobMaster 会根据 JobGraph 按并发展开，从而得到 JobMaster 中关键的 ExecutionGraph。ExecutionGraph 的结构如下图所示，与 JobGraph 相比，ExecutionGraph 中对于每个 Task 和中间结果等均创建了对应的对象，从而可以维护这些实体的信息与状态。

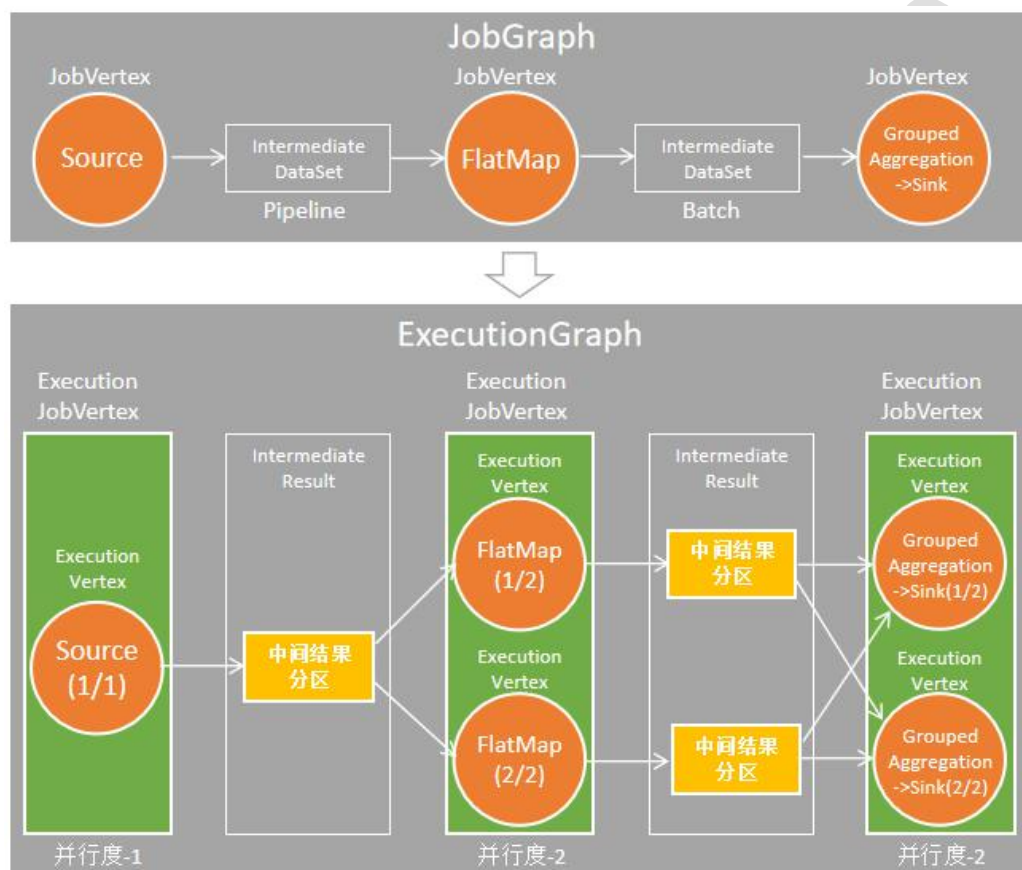


图 ExecutionGraph 是 JobGraph 按并发展开所形成的，它是 JobMaster 中的核心数据结构

在一个 Flink Job 中是包含多个 Task 的，因此另一个关键的问题是在 Flink 中按什么顺序来调度这些 Task。如下图所示，目前 Flink 提供了两种基本的调度策略，即延迟调度（Lazy From Source）和即时调度（Eager 调度）。即时调度会在作业启动时申请资源将所有的 Task 调度起来。这种调度算法主要用来调度可能没有终止的流作业。与之对应，延迟调度则是从 Source 开始，按拓扑顺序来进行调度。简单来说，延迟调度会先调度没有上游任务的 Source 任务，当这些任务执行完成时，它会将输出数据缓存到内存或者写入到磁盘中。然后，对于后续的任务，当它的前驱任务全部执行完成后，Flink 就会将这些任务调度起来。这些任务会从读取上游缓存的输出数据进行自己的计算。这一过程继续进行直到所有的任务完成计算。

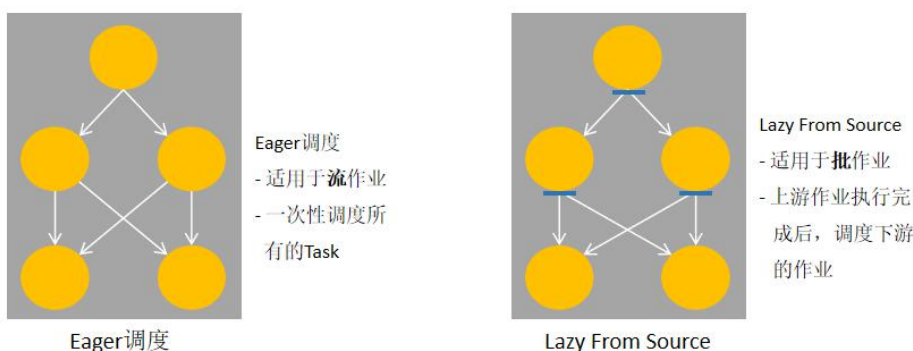


图 Flink 中两种基本的调度策略

1.3.5 Flink 故障恢复

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

对于第一类 **task** 执行错误，Flink 提供了多种不同的错误恢复策略。

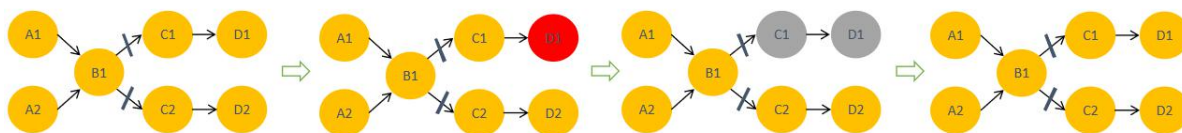
- ❑ 第一种错误恢复策略是 **restart-all**，即直接重启所有的 **task**。对于 Flink 的流任务，由于 Flink 提供了 **checkpoint** 机制，因此当任务重启后可以直接从上次的 **checkpoint** 开始继续执行。因此这种方式更适用于流作业。
- ❑ 第二种错误恢复策略是 **restart-individual**，它只适用于 **task** 之间没有数据传输的情况。这种情况下，可以直接重启出错的 **task**。

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

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

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

（1）如果是由于 **task** 本身的问题发生错误，那么可以只重启该 **task** 所属的分区中的所有 **task**，这些 **task** 重启之后，可以直接拉取上游分区缓存的输出结果继续进行计算。



（2）另一方面，如果错误是由于读取上游结果出现问题，如网络连接中断、缓存上游输出数据的 **TaskManager** 异常退出等，那么还需要重启上游分区来重新产生相应的数据。在这种情况下，如果上游分区输出的数据分发方式不是确定性的（如 **keyBy**、**broadcast** 是确定性的分发方式，而 **rebalance**、

random 则不是，因为每次执行会产生不同的分发结果），为保证结果正确性，还需要同时重启上游分区所有的下游分区。



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

1.3.6 Flink 程序执行模式

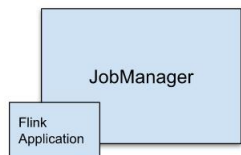
Flink 可以通过以下三种方式来执行应用程序：

- ☐ 以 Application Mode,
- ☐ 以 Per-Job Mode,
- ☐ 以 Session Mode

以上三种模式的区别在于：

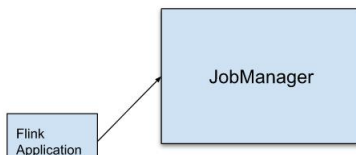
- ☐ 集群生命周期和资源隔离保证；
- ☐ 应用程序的 main()方法是在客户端上执行还是在集群上执行。

Application Mode



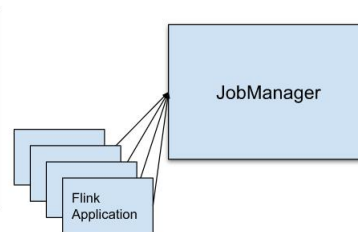
A dedicated JobManager is started for submitting the job. The JobManager will only execute this job, then exit. The Flink Application runs on the JobManager.

Per-Job Mode



A dedicated JobManager is started for submitting the job. The JobManager will only execute this job, then exit. The Flink Application runs on the client submitting the per-job cluster.

Session Mode



Multiple jobs share one JobManager.

Per-Job 模式

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

- ☐ 集群生命周期：在 Flink Job Cluster 中，可用的集群管理器(如 YARN)用于为每个提交的作业启动集群，该集群仅对该作业可用。在这里，客户端首先从集群管理器请求资源以启动 JobManager，并将作业提交给运行在此进程中的 Dispatcher。然后根据作业的资源需求惰性地分配 TaskManager。一旦作业完成，Flink 作业集群就会被拆除。集群的生命周期与作业的生命周期绑定在一起。
- ☐ 资源隔离：更好的隔离保证，因为资源不会跨作业共享。JobManager 中的致命错误只会影响 Flink

Job Cluster 中运行的一个作业。此外，它将簿记（book-keeping）的负载分散到多个 JobManager，因为每个作业有一个 JobManager。

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

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

例如，我们可以在 YARN 上使用 Per-Job 模式来部署运行 Flink 自带的示例程序。

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

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

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

```
# 列出集群上正在运行的作业
```

```
$ ./bin/flink list -t yarn-per-job -Dyarn.application.id=application_XXXX_YY
```

```
# 撤销正在运行的作业
```

```
$ ./bin/flink cancel -t yarn-per-job -Dyarn.application.id=application_XXXX_YY <jobId>
```

请注意，取消 Per-Job Cluster 上的作业将停止集群。

Session 模式

Session Mode 假设一个已经运行的集群（称为 Flink Session Cluster），并使用该集群的资源来执行任何提交的应用程序。在 Session Mode 中，集群的生命周期独立于集群上运行的任何作业的生命周期，并且资源在所有作业之间共享。

- ❑ 集群生命周期：在 Session 模式中，客户端连接到一个预先存在的、长时间运行的集群，该集群可以接受多个作业提交。即使在所有作业完成后，集群(和 JobManager)仍将继续运行，直到手动停止会话。因此，Flink Session Cluster 的生存期不与任何 Flink Job 的生存期绑定。
- ❑ 资源隔离：TaskManager 槽位在作业提交时由 ResourceManager 分配，作业完成后释放。因为所有作业都共享同一个集群，所以在提交作业阶段存在一些对集群资源的竞争—比如网络带宽。这种共享设置的一个限制是，如果一个 TaskManager 崩溃了，那么所有在这个 TaskManager 上运行的任务都会失败；以类似的方式，如果 JobManager 上发生一些致命错误，它将影响集群中运行的所有作业。除了对导致失败的作业产生负面影响外，这还意味着一个潜在的大规模恢复过程，所有重新启动的作业都同时访问文件系统，并使其他服务无法使用它。此外，让一个集群运行多个作业意味着 JobManager 需要承担更多的负载，JobManager 负责对集群中的所有作业进行记账（book-keeping）。

使用预先存在的集群可以节省大量申请资源和启动 TaskManager 的时间。当作业的执行时间非常短，而高启动时间会对端到端用户体验产生负面影响时，这一点非常重要—就像短查询的交互分析那样，作业可以使用现有资源快速执行计算。

这种模式类似于 Standalone，JobManager 会预先启动，此时 Client 直接与 Dispatcher 建立连接并提交作业即可。

例如，我们使用 Session Mode 部署 Flink 自带的 WordCount 程序：

```
# 假设是在 Flink 发行版的根目录下
```

```
# (1)启动 Flink session 集群
$ ./bin/stop-cluster.sh

# (2)现在可以访问 http://localhost:8081 的 Flink Web 界面

# (3)提交作业执行
$ ./bin/flink run ./examples/batch/WordCount.jar

# (4)重新停止集群
$ ./bin/stop-cluster.s
```

Application 模式

在所有其他模式中，应用程序的 `main()` 方法都在客户端执行。这个过程包括在本地下载应用程序的依赖项，执行 `main()` 来提取 Flink 的运行时可以理解的应用程序的表示(即 `JobGraph`)，并将依赖项和 `JobGraph` 发送到集群。这使得客户端成为一个沉重的资源消耗者，因为它可能需要大量的网络带宽来下载依赖项并将二进制文件发送到集群，并需要 CPU 周期来执行 `main()`。当客户端在用户之间共享时，这个问题会更加明显。

在此观察的基础上，`Application Mode` 为每个提交的应用程序创建一个集群，并且应用程序的 `main()` 方法在 `JobManager` 上执行。在 `JobManager` 上执行 `main()` 可以节省所需的 CPU 周期，还可以节省本地下载依赖项所需的带宽。此外，为了下载集群中应用程序的依赖关系，它允许更均匀地分散网络负载，因为每个应用程序有一个 `JobManager`。

- ❑ 集群生命周期：一个 `Flink Application Cluster` 是一个专用的 Flink 集群，它只执行来自一个 Flink 应用程序的作业，其中 `main()` 方法运行在集群而不是客户机上。作业提交是 `one-step` 过程：不需要首先启动一个 Flink 集群，然后将作业提交到现有的集群会话；相反，可以将应用程序逻辑和依赖打包到一个可执行的作业 JAR 中，并且集群入口点(`ApplicationClusterEntryPoint`)负责调用 `main()` 方法来提取 `JobGraph`。这允许我们像在 `Kubernetes` 上部署其他应用程序一样部署 Flink 应用程序。因此，`Flink Application Cluster` 集群的生命周期与 Flink 应用程序的生命周期绑定在一起。
- ❑ 资源隔离：在 `Flink Application Cluster` 中，`ResourceManager` 和 `Dispatcher` 被限定为单个 Flink 应用程序，这比 `Flink Session Cluster` 提供了更好的关注点分离。

与 `Per-Job` 模式相比，`Application Mode` 允许提交由多个作业组成的应用程序。作业执行的顺序不受部署模式的影响，而是受用于启动作业的调用的影响。使用 `execute()` 是阻塞的，它会建立一个顺序，并导致“下一个”作业的执行被推迟到“这个”作业完成。使用 `executeAsync()`，它是非阻塞的，将导致“下一个”任务在“这个”任务完成之前开始。

`Application Mode` 相当于为每个应用程序创建一个会话（`session`）集群，并在集群上执行应用程序的 `main()` 方法。

在这种部署模式下，应用程序 jar 文件需要在类路径中可用。最简单的方法就是把 jar 放到 `lib/` 文件夹中。例如，我们使用 `Application Mode` 部署 Flink 自带的 `TopSpeedWindowing` 程序：

```
# 将 jar 包拷贝到 lib/文件夹下
$ cp ./examples/streaming/TopSpeedWindowing.jar lib/

# 先启动一个 TaskManager（也可以启动多个 TaskManager，如果应用程序需要更多资源的话）
$ ./bin/taskmanager.sh start
```

```
# 启动 JobManager
$ ./bin/standalone-job.sh start --job-classname
org.apache.flink.streaming.examples.windowing.TopSpeedWindowing
该脚本还支持停止服务。如果想停止多个实例，可多次调用它们，或者使用 stop-all:
$ ./bin/taskmanager.sh stop
$ ./bin/standalone-job.sh stop
```

1.4 Flink 集群安装

Flink 运行在 Linux、Mac OS x 和 Windows 上。本教程中我们将 Flink 集群搭建在 Linux 系统上。由于 Flink 是用 Java 和 Scala 实现的，所以所有组件都运行在 JVM 上。

使用 Flink 需要满足以下先决条件：

- ❑ 需要安装 Java 8/Java 11 来运行 Flink 作业/应用程序；
- ❑ Scala API（可选地）依赖于 Scala 2.11；
- ❑ 如果配置为高可用（没有单点故障），需要 Apache ZooKeeper；
- ❑ 如果配置为高可用（可以从故障中恢复）的流处理，Flink 需要某种形式的检查点分布式存储（HDFS / S3 / NFS / SAN / GFS / Kosmos / Ceph / ...）

1.4.1 Flink 独立集群安装和测试

Flink 集群可以运行在单节点上，这称为“Standalone Cluster”模式（在一台机器上，但在不同的进程中）。独立模式是部署 Flink 最简单的方式。Standalone 集群安装步骤如下。

1) 要运行 Flink，要求必须安装好 Java 8。可以使用如下命令检查 Java 是否已经正确安装：

```
$ java -version
```

如果已经正确地安装了 Java 8，那么会输出类似如下的内容：

```
[hduser@hadoop ~]$ java -version
java version "1.8.0_181"
Java(TM) SE Runtime Environment (build 1.8.0_181-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.181-b13, mixed mode)
[hduser@hadoop ~]$
```

2) 下载 Flink 安装包。下载地址：<https://flink.apache.org/downloads.html>。可以选择任何喜欢的 Hadoop/Scala 组合。

Apache Flink 1.13.2

Apache Flink 1.13.2 for Scala 2.11 (asc, sha512)

Apache Flink 1.13.2 for Scala 2.12 (asc, sha512)

Apache Flink 1.13.2 Source Release (asc, sha512)

本教程选择使用这个版本

3) 解压缩安装包。将下载的安装包放在“~/software/”目录下，然后将其解压缩到指定的位置（例如，~/bigdata/目录下）。在终端执行如下的命令。

```
$ cd ~/bigdata
$ tar -zxvf ~/software/flink-1.13.2-bin-scala_2.12.tgz
```

4) 启动一个本地 Flink 集群。

对于单节点设置，Flink 是开箱即用的，即不需要更改默认配置，直接启动即可。

```
$ cd flink-1.13.2
```

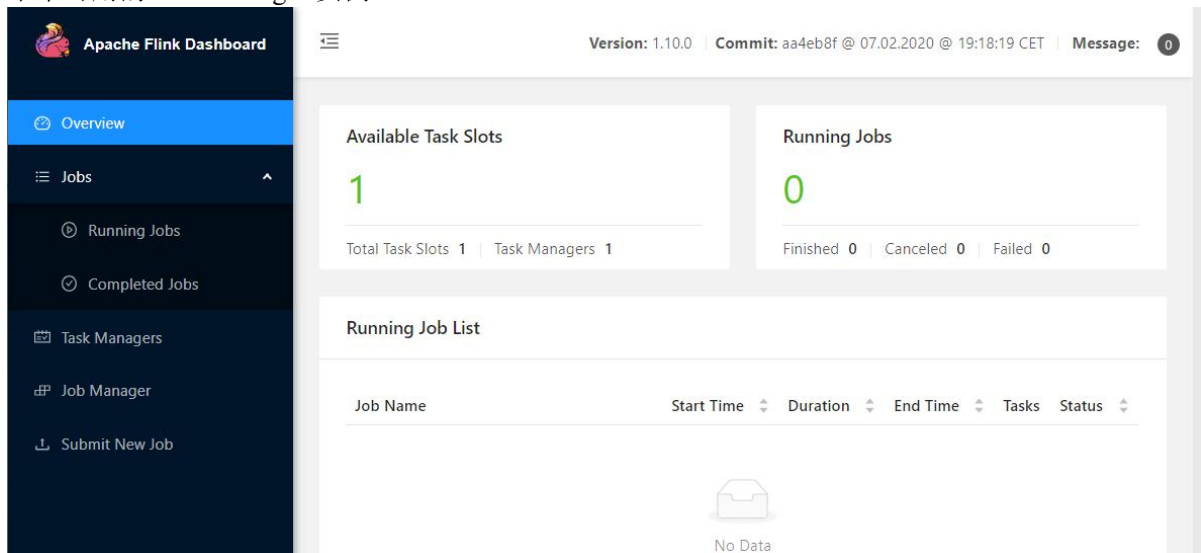
```
$ ./bin/start-cluster.sh
```

使用 `jps` 命令查看，可以看到启动了以下两个进程：

```
2569 StandaloneSessionClusterEntrypoint
```

```
2844 TaskManagerRunner
```

打开浏览器，输入地址：`http://localhost:8081`，可查看检查调度程序的 web 前端。web 前端应该报告有单个可用的 TaskManager 实例。



还可以通过检查 `log` 目录中的日志文件来验证系统是否正在运行：

```
$ tail log/flink-*-standalonesession-*.log
```

5) 要关闭 Flink 集群，使用如下的命令：

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

运行 Flink 自带的实时单词计数程序 - Socket 数据源

Flink 安装包自带了一个以 Socket 作为数据源的实时统计单词计数的程序，位于 Flink 下的“`example/streaming/SocketWindowWordCount.jar`”包中。我们可以通过运行这个程序来测试 Flink 集群的使用方法。请按以下步骤执行。

1) 首先，启动 netcat 服务器，运行在 9000 端口：

```
$ nc -l 9000
```

2) 打开另一个终端，执行以下命令，启动 Flink 集群：

```
$ cd ~/bigdata/flink-1.13.2
```

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

3) 然后启动 Flink 示例程序，监听 netcat 服务器的输入：

```
$ ./bin/flink run examples/streaming/SocketWindowWordCount.jar --hostname localhost --port 9000
```

这个实时单词计数程序将从 Socket 套接字中读取输入的文本内容，并每 5 秒打印前 5 秒内每个不同单词出现的次数，即处理时间的滚动窗口。

4) 在 netcat 控制台，键入一些内容，Flink 将会处理它。

```
good good study
```

```
day day up
```

5) 启动第三个终端窗口，并在该窗口中执行以下命令，查看日志中的输出：

```
$ cd ~/bigdata/flink-1.13.2
```

```
$ tail -f log/flink-*-taskexecutor-*.out
```

可以看到如下输出结果：

```
good : 2
study : 1
day : 2
up : 1
```

6) 还可以检查 Flink Web UI 来查看 job 是怎样执行的。

单击图中的【Running Job List】下正在运行的作业列表，查看某一个正在运行的作业执行情况：

Name	Status	Bytes Received	Records Received	Bytes Sent	Records Sent	Parallelism	Start Time	Tasks
Source: Socket Stream -> Flat Map	RUNNING	0 B	0	0 B	22	1	2020-03-31 14:28:51	1
Window(TumblingProcessingTimeWindows(5000), ProcessingTimeTrigger, ReduceFunction \$1, PassThroughWindowFunction) -> Sink: Print to Std. Out	RUNNING	440 B	22	0 B	0	1	2020-03-31 14:28:51	1

7) 最后停止 Flink 集群，使用如下命令：

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

运行 Flink 自带的单词计数批处理程序 - 文件数据源

Flink 安装包自带了一个以文本文件作为数据源的单词计数批处理程序，位于 Flink 下的“example/batch/”目录下的 WordCount.jar 包中。下面我们在 Flink 集群上执行该程序，读取 HDFS 上的输入数据文件进行处理，并输出计算结果到 HDFS 上。

请按以下步骤执行。

1) 集成 Hadoop。

因为要读取 HDFS 上的源数据文件，所以需要在 Flink 中集成 Hadoop 包。

注：从 flink 1.8 开始，Hadoop 不再包含在 Flink 的安装包中，所以需要单独下载并拷贝到 Flink 的 lib 目录下。

❑ 如果使用的是 Hadoop 2，则可从 Flink 官网下载 flink-shaded-hadoop2-uber-2.7.5-1.10.0.jar。如下

图所示：

Additional Components

These are components that the Flink project develops which are not part of the main Flink release:

[Apache Flink-shaded 14.0 Source Release \(asc, sha512\)](#)

Pre-bundled Hadoop 2.4.1 (asc, sha1)
Pre-bundled Hadoop 2.6.5 (asc, sha1)
Pre-bundled Hadoop 2.7.5 (asc, sha1)
Pre-bundled Hadoop 2.8.3 (asc, sha1)

下载与自己的Hadoop版本相对应的Hadoop包

- ❑ 如果使用的是 Hadoop 3，则从 Maven 库下载。如下图所示：

Home » [org.apache.flink](#) » [flink-shaded-hadoop-3-uber](#)



Flink Shaded Hadoop 3 Uber

Flink Shaded Hadoop 3 Uber

License	Apache 2.0
Tags	hadoop apache
Used By	7 artifacts

选择最新的版本下载即可

Version	Repository	Usages	Date
3.1.1.7.2.9.0-173-9.0	Cloudera	4	Apr, 2021
3.1.1.7.2.8.0-224-9.0	Cloudera	4	Apr, 2021
3.1.1.7.2.1.0-327-9.0	Cloudera	7	Dec, 2020
3.1.1.7.0.3.0-79-7.0	Cloudera	6	Jan, 2020

本教程中使用的是 Hadoop-3.2.2，所以下载 `flink-shaded-hadoop-3-uber-3.1.1.7.2.9.0-173-9.0.jar` 包。
将下载的 Hadoop 包拷贝到 Flink 的 lib 目录下。

2) 准备环境。

在终端窗口中，执行如下的命令，先启动 Flink 集群，再启动 HDFS 集群：

```
# 首先启动 Flink 集群
$ ./bin/start-cluster.sh
```

```
# 再启动 HDFS 集群
$ start-dfs.sh
```

3) 准备数据文件。

编辑数据文件 `wc.txt`，内容如下：

```
good good study
day day up
```

然后将该文件上传到 HDFS 的 `/data/flink/` 目录下。

```
$ hdfs dfs -put wc.txt /data/flink/
```

4) 然后执行 Flink 的批处理程序。

```
# 执行 Flink 单词计数程序
$ ./bin/flink run ./examples/batch/WordCount.jar \
  --input hdfs://xueai8:8020/data/flink/wc.txt \
  --output hdfs://xueai8:8020/data/flink/wc-result
```

执行过程如下图所示：


```
[hduser@xueai8 flink-1.13.2]$ ./bin/start-cluster.sh
Starting cluster.
Starting standalone session daemon on host xueai8.
Starting taskexecutor daemon on host xueai8.
[hduser@xueai8 flink-1.13.2]$ jps
9363 Jps
9028 StandaloneSessionClusterEntrypoint
9308 TaskManagerRunner
[hduser@xueai8 flink-1.13.2]$ start-dfs.sh
Starting namenodes on [xueai8]
Starting datanodes
Starting secondary namenodes [xueai8]
[hduser@xueai8 flink-1.13.2]$ jps
9028 StandaloneSessionClusterEntrypoint
9893 SecondaryNameNode
9308 TaskManagerRunner
9564 NameNode
10012 Jps
9679 DataNode
[hduser@xueai8 flink-1.13.2]$ ./bin/flink run ./examples/batch/WordCount.jar \
> --input hdfs://xueai8:8020/data/flink/wc.txt \
> --output hdfs://xueai8:8020/data/flink/wc-result
Job has been submitted with JobID aa7b5a8aee5bd73e0bc0617cdd205c3c
Program execution finished
Job with JobID aa7b5a8aee5bd73e0bc0617cdd205c3c has finished.
Job Runtime: 5760 ms
[hduser@xueai8 flink-1.13.2]$
```

启动Flink集群

启动HDFS集群

执行Flink自带的单词计数批处理程序，并指定输入路径和输出路径（输出路径事先要不存在）。

上面的命令是在运行 WordCount 时读写 HDFS 中的文件，其中--input 参数指定要处理的输入文件，--output 指定计算结果输出到的结果文件。（注：如果不加 hdfs://前缀，默认使用本地文件系统）

注：在执行程序时，很可能会出现下面这样的错误信息：

```
java.lang.NoSuchMethodError:
org.apache.commons.cli.Option.builder(Ljava/lang/String;)Lorg/apache/commons/cli/Option$Builder;
    at org.apache.flink.client.cli.DynamicPropertiesUtil.<clinit>(DynamicPropertiesUtil.java:39)
    at org.apache.flink.client.cli.GenericCLI.addGeneralOptions(GenericCLI.java:108)
    at org.apache.flink.client.cli.CliFrontend.<init>(CliFrontend.java:136)
    at org.apache.flink.client.cli.CliFrontend.<init>(CliFrontend.java:119)
    at org.apache.flink.client.cli.CliFrontend.main(CliFrontend.java:1129)
```

原因是 Flink 缺少了 commons-cli 的相关 jar 包。从 Maven 仓库下载该 jar 包，拷贝到 Flink 的 lib 目录下即可。

5) 执行以下命令查询输出结果：

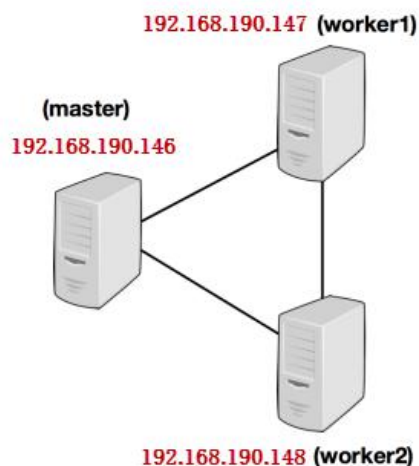
```
$ hdfs dfs -cat /data/flink/wc-result/*
```

可以看到以下计算结果：

```
day 2
good 2
study 1
up 1
```

1.4.2 Flink 完全分布式集群安装

Flink 支持完全分布式安装模式，这时它由一个 master 节点和多个 worker 节点构成。在本节，我们将搭建一个如下的三个节点的 Flink 集群。



Flink 完全分布式集群搭建步骤如下：

1、配置从 master 到 worker 节点的 SSH 无密登录，并保持保节点上相同的目录结构。

SSH 原理：

master 作为客户端，要实现无密码公钥认证，连接到服务端 worker 上时，需要在 master 上生成一个密钥对，包括一个公钥和一个私钥，而后将公钥复制到 worker 上。当 master 通过 ssh 连接 worker 时，worker 就会生成一个随机数并用 master 的公钥对随机数进行加密，并发送给 master。master 收到加密数之后再用私钥进行解密，并将解密数回传给 worker，worker 确认解密数无误之后就允许 master 进行连接了。这就是一个公钥认证过程，其间不需要用户手工输入密码。重要过程是将客户端 master 公钥复制到 worker 上。

(1) 在每台机器上，执行如下命令：

```
$ ssh localhost
$ ssh exit          # 记得最后通过这个命令退出 ssh 连接
```

(2) 在 master 节点上，使用如下命令生成公私钥：

```
$ cd .ssh
$ ssh-keygen -t rsa
```

然后一路回车，在.ssh下生成公私钥。

(3) 将 master 上的公钥分别加入 master、worker1 和 worker2 机器的授权文件中。

在 master 机器上，执行如下命令：

```
$ ssh-copy-id hduser@master
$ ssh-copy-id hduser@worker1
$ ssh-copy-id hduser@worker2
```

(4) 测试。在 master 机器上，使用 ssh 分别连接 master、worker1 和 worker2：

```
$ ssh master
$ ssh worker1
$ ssh worker2
```

这时会发现不需要输入密码，直接就 ssh 连接上了这两台机器。

2、Flink 要求在主节点和所有工作节点上设置 JAVA_HOME 环境变量，并指向 Java 安装的目录。

使用如下命令检查 Java 的安装和版本信息：

```
$ java -version
```

3、下载 Flink 安装包。下载地址：<https://flink.apache.org/downloads.html>。可以选择任何喜欢的 Hadoop/Scala 组合。

Apache Flink 1.13.2

Apache Flink 1.13.2 for Scala 2.11 (asc, sha512)

Apache Flink 1.13.2 for Scala 2.12 (asc, sha512)

Apache Flink 1.13.2 Source Release (asc, sha512)

本教程选择使用这个版本

4、将下载的最新版本的 Flink 压缩包拷贝到 master 节点的 “~/software/” 目录下，并解压缩到 “~/bigdata/” 目录下，步骤如下：

```
$ cd ~/bigdata/  
$ tar xzf ~/software/flink-1.13.2-bin-scala_2.12.tgz  
$ cd flink-1.13.2
```

5、在 master 节点上配置 Flink

所有的配置都在 “conf/flink-conf.yaml” 文件中。在实际应用中，以下几个配置项是非常重要的：

- ☐ jobmanager.heap.mb: 每个 JobManager 的可用内存量，以 MB 为单位。
- ☐ taskmanager.heap.mb: 每个 TaskManager 的可用内存量，以 MB 为单位。
- ☐ taskmanager.numberOfTaskSlots: 每台机器上可用的 CPU 数量，默认为 1。
- ☐ parallelism.default: 集群中 CPU 的总数。
- ☐ io.tmp.dirs: 临时目录。

首先用编辑器 nano 打开该配置文件（你也可以用任何你喜欢的编辑器，如 vim，都可以）。

```
$ nano conf/flink-conf.yaml
```

编辑如下内容（注意，冒号后面一定要有一个空格）：

```
jobmanager.rpc.address: master           // 指向 master 节点  
jobmanager.rpc.port: 6123  
jobmanager.heap.size: 1024m              // 定义允许 JVM 在每个节点上分配的最大主内存量  
taskmanager.memory.process.size: 1024m  
taskmanager.numberOfTaskSlots: 2  
parallelism.default: 6
```

6、每个节点下的 Flink 必须保持相同的目录内容。因此将配置好的 Flink 拷贝到集群中的另外两个节点 work01 和 work02，使用如下的命令：

```
$ scp -r ~/bigdata/flink-1.11.1 hduser@worker01:~/bigdata/  
$ scp -r ~/bigdata/flink-1.11.1 hduser@worker02:~/bigdata/
```

7、最后，必须提供集群中所有用作 worker 节点的列表。在 “conf/slaves” 文件中添加每个 slave 节点信息（IP 或 hostname 均可），每个节点一行，如下所示。每个工作节点稍后将运行一个 TaskManager：

```
master  
worker1  
worker2
```

8、启动 Flink 集群：

```
$ cd ~/bigdata/flink-1.13.2  
$ ./bin/start-cluster.sh
```

这个脚本会在本地节点启动一个 JobManager 并通过 SSH 连接到所有的 worker 节点(在 slaves 文件中列出的) 以启动每个节点上的 TaskManager。注意观察启动过程中的输出信息，如下：

```
Starting cluster.  
Starting standalone-session daemon on host master.  
Starting task-executor daemon on host master.  
Starting task-executor daemon on host worker1.  
Starting task-executor daemon on host worker2.
```

可以看出，Flink 先在 master 上启动 standalone-session 进程，然后依次在 master、worker1 和 worker2

上启动 taskexecutor 进程。

启动以后，分别在 master、worker1 和 worker2 节点上执行 jps 命令，查看各节点上的进程是否正常启动了。

9、关闭集群

```
$ cd ~/bigdata/flink-1.13.2
```

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

也可以分别停止 JobManager 和 TaskManager。例如，

执行以下命令，停止单个的 Job Manager：

```
$ ./bin/jobmanager.sh stop
```

执行以下命令，停止单个的 Task Manager：

```
$ ./bin/taskmanager.sh stop
```

执行 Flink 自带的单词计数流处理程序 - Socket 数据源

1) 首先，启动 netcat 服务器，运行在 9000 端口：

```
$ nc -l 9000
```

2) 在另一个终端，启动 Flink 示例程序，监听 netcat 服务器。它将从套接字中读取文本，并每 5 秒打印前 5 秒内每个不同单词出现的次数，即处理时间的滚动窗口。

```
$ cd ~/bigdata/flink-1.13.2
```

```
$ ./bin/flink run examples/streaming/SocketWindowWordCount.jar --hostname master --port 9000
```

3) 回到第一个正在运行 netcat 的终端窗口，随意输入一些内容，单词之间用空格分隔，Flink 将会处理它。例如，输入以下文本内容：

```
good good study  
day day up
```

4) 分别使用 ssh 登录 master、worker01 和 worker02 节点，并执行以下命令，查看日志中的输出：

```
$ cd ~/bigdata/flink-1.13.2
```

```
$ tail -f log/flink-*-taskexecutor-*.out
```

可以看到如下输出结果：

```
good : 2  
study : 1  
day : 2  
up : 1
```

5) 还可以检查 Flink Web UI 来查看 Job 是怎样执行的。

打开浏览器，输入地址：<http://localhost:8081>，可查看检查调度程序的 web 前端。web 前端应该报告有三个可用的 TaskManager 实例，以及正在执行的作业。Flink WebUI 包含许多关于 Flink 集群及其作业(JobGraph、指标、检查点统计、TaskManager 状态等)的有用而有趣的信息。

The screenshot shows the Apache Flink Dashboard interface. On the left is a sidebar with navigation links: Overview, Jobs, Running Jobs, Completed Jobs, Task Managers, Job Manager, and Submit New Job. The main content area displays the following information:

- Available Task Slots:** 3 (Total Task Slots: 6, Task Managers: 3)
- Running Jobs:** 1 (Finished: 0, Canceled: 0, Failed: 0)
- Running Job List:** A table with columns: Job Name, Start Time, Duration, End Time, Tasks, and Status. One job is listed: "Socket Window WordCount" with a status of "RUNNING". A red callout bubble points to the job name with the text "单击此链接, 查看此Job是怎样执行的".
- Completed Job List:** A table with the same columns as the Running Job List, but it is empty with a "No Data" message.

点击正在运行的作业, 查看作业运行的详细信息, 如下图所示:

This screenshot shows the detailed view of the "Socket Window WordCount" job. At the top, it displays the job name, ID, start time, duration, and a "Cancel Job" button. Below this is a navigation bar with tabs: Overview, Exceptions, TimeLine, Checkpoints, and Configuration. The main part of the screenshot shows a job graph and a table of task metrics.

Job Graph:

```

graph LR
    A[Source: Socket Stream  
Parallelism: 1] -- REBALANCE --> B[Flat Map  
Parallelism: 3]
    B -- HASH --> C[Window(TumblingProcessingTimeWindows(5000), ProcessingTimeTrigger, ReduceFunction$1, PassThroughWindowFunction)  
Parallelism: 3]
    C -- REBALANCE --> D[Sink: Print to Std. Out  
Parallelism: 1]
  
```

Task Metrics Table:

Name	Status	Bytes Received	Records Received	Bytes Sent	Records Sent	Parallelism	Start Time	Duration	End Time	Tasks
Source: Socket Stream	RUNNING	0 B	0	0 B	2	1	2020-03-31 17:45:41	10m 13s	-	1
Flat Map	RUNNING	37 B	2	0 B	6	3	2020-03-31 17:45:41	10m 13s	-	3
Window(TumblingProcessingTimeWindows(5000), ProcessingTimeTrigger, ReduceFunction\$1, PassThroughWindowFunction)	RUNNING	123 B	6	0 B	4	3	2020-03-31 17:45:41	10m 13s	-	3
Sink: Print to Std. Out	RUNNING	114 B	4	0 B	0	1	2020-03-31 17:45:41	10m 13s	-	1

6) 最后停止 Flink 集群, 使用如下命令:

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

运行 Flink 自带的单词计数批处理程序 - 文件数据源

Flink 安装包自带了一个以文本文件作为数据源的单词计数批处理程序, 位于 Flink 下的“example/batch/”目录下的 WordCount.jar 包中。下面我们在 Flink 集群上执行该程序, 读取 HDFS 上的输入数据文件进行处理, 并输出计算结果到 HDFS 上。

请按以下步骤执行。

1) 集成 Hadoop。

因为要读取 HDFS 上的源数据文件, 所以需要在 Flink 中集成 Hadoop 包。

注: 从 flink 1.8 开始, Hadoop 不再包含在 Flink 的安装包中, 所以需要单独下载并拷贝到 Flink 的 lib 目录下。

❑ 如果使用的是 Hadoop 2, 则可从 Flink 官网下载 flink-shaded-hadoop2-uber-2.7.5-1.10.0.jar。如下

图所示：

Additional Components

These are components that the Flink project develops which are not part of the main Flink release:

[Apache Flink-shaded 14.0 Source Release \(asc, sha512\)](#)

Pre-bundled Hadoop 2.4.1 (asc, sha1)
Pre-bundled Hadoop 2.6.5 (asc, sha1)
Pre-bundled Hadoop 2.7.5 (asc, sha1)
Pre-bundled Hadoop 2.8.3 (asc, sha1)

下载与自己的Hadoop版本相对应的Hadoop包

- ❑ 如果使用的是 Hadoop 3，则从 Maven 库下载。如下图所示：

Home » [org.apache.flink](#) » [flink-shaded-hadoop-3-uber](#)



Flink Shaded Hadoop 3 Uber

Flink Shaded Hadoop 3 Uber

License	Apache 2.0
Tags	hadoop apache
Used By	7 artifacts

选择最新的版本下载即可

Version	Repository	Usages	Date
3.1.1.7.2.9.0-173-9.0	Cloudera	4	Apr, 2021
3.1.1.7.2.8.0-224-9.0	Cloudera	4	Apr, 2021
3.1.1.7.2.1.0-327-9.0	Cloudera	7	Dec, 2020
3.1.1.7.0.3.0-79-7.0	Cloudera	6	Jan, 2020

本教程中使用的是 Hadoop-3.2.2，所以下载 `flink-shaded-hadoop-3-uber-3.1.1.7.2.9.0-173-9.0.jar` 包。将下载的 Hadoop 包拷贝到各个节点安装的 Flink 的 lib 目录下。

2) 准备环境。

在终端窗口中，执行如下的命令，先启动 Flink 集群，再启动 HDFS 集群：

首先启动 Flink 集群

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

再启动 HDFS 集群

```
$ start-dfs.sh
```

3) 准备数据文件。

编辑数据文件 `wc.txt`，内容如下：

```
good good study
```

```
day day up
```

然后将该文件上传到 HDFS 的 `/data/flink/` 目录下。

```
$ hdfs dfs -put wc.txt /data/flink/
```

4) 然后执行 Flink 的批处理程序。

执行 Flink 单词计数程序

```
$ ./bin/flink run ./examples/batch/WordCount.jar \
    --input hdfs://master:8020/data/flink/wc.txt \
    --output hdfs://master:8020/data/flink/wc-result
```

执行过程如下图所示：


```
[hduser@xueai8 flink-1.13.2]$ ./bin/start-cluster.sh
Starting cluster.
Starting standalone session daemon on host xueai8.
Starting taskexecutor daemon on host xueai8.
[hduser@xueai8 flink-1.13.2]$ jps
9363 Jps
9028 StandaloneSessionClusterEntrypoint
9308 TaskManagerRunner
[hduser@xueai8 flink-1.13.2]$ start-dfs.sh
Starting namenodes on [xueai8]
Starting datanodes
Starting secondary namenodes [xueai8]
[hduser@xueai8 flink-1.13.2]$ jps
9028 StandaloneSessionClusterEntrypoint
9893 SecondaryNameNode
9308 TaskManagerRunner
9564 NameNode
10012 Jps
9679 DataNode
[hduser@xueai8 flink-1.13.2]$ ./bin/flink run ./examples/batch/WordCount.jar \
> --input hdfs://xueai8:8020/data/flink/wc.txt \
> --output hdfs://xueai8:8020/data/flink/wc-result
Job has been submitted with JobID aa7b5a8aee5bd73e0bc0617cdd205c3c
Program execution finished
Job with JobID aa7b5a8aee5bd73e0bc0617cdd205c3c has finished.
Job Runtime: 5760 ms
[hduser@xueai8 flink-1.13.2]$
```

启动Flink集群

启动HDFS集群

执行Flink自带的单词计数批处理程序，并指定输入路径和输出路径（输出路径事先要不存在）。

上面的命令是在运行 WordCount 时读写 HDFS 中的文件，其中--input 参数指定要处理的输入文件，--output 指定计算结果输出到的结果文件。（注：如果不加 hdfs://前缀，默认使用本地文件系统）

注：在执行程序时，很可能会出现下面这样的错误信息：

```
java.lang.NoSuchMethodError:
org.apache.commons.cli.Option.builder(Ljava/lang/String;)Lorg/apache/commons/cli/Option$Builder;
    at org.apache.flink.client.cli.DynamicPropertiesUtil.<clinit>(DynamicPropertiesUtil.java:39)
    at org.apache.flink.client.cli.GenericCLI.addGeneralOptions(GenericCLI.java:108)
    at org.apache.flink.client.cli.CliFrontend.<init>(CliFrontend.java:136)
    at org.apache.flink.client.cli.CliFrontend.<init>(CliFrontend.java:119)
    at org.apache.flink.client.cli.CliFrontend.main(CliFrontend.java:1129)
```

原因是 Flink 缺少了 commons-cli 的相关 jar 包。从 Maven 仓库下载该 jar 包，拷贝到各个节点 Flink 安装的 lib 目录下即可。

5) 执行以下命令查询输出结果：

```
$ hdfs dfs -cat /data/flink/wc-result/*
```

可以看到以下计算结果：

```
day 2
good 2
study 1
up 1
```

1.4.3 Flink 常用命令

Flink 以 CLI 的方式提供了一些常见的操作任务。下面是常用的 CLI 命令。

1、查看正在运行的作业

CLI 方式：

```
$ flink list
```

注：也可用 REST API 方式查看正在运行的作业：

```
$ curl localhost:8081/jobs
```

2、运行示例程序，不带参数：

```
$ ./bin/flink run ./examples/batch/WordCount.jar
```

3、运行示例程序，带有 input 和 output 参数：

```
$ ./bin/flink run ./examples/batch/WordCount.jar \  
    --input file:///home/user/hamlet.txt \  
    --output file:///home/user/wordcount_out
```

4、运行示例程序，带有 input 和 output 参数，并指定并行度为 16：

```
$ ./bin/flink run -p 16 ./examples/batch/WordCount.jar \  
    --input file:///home/user/hamlet.txt \  
    --output file:///home/user/wordcount_out
```

5、运行示例程序，禁用 flink 日志输出：

```
$ ./bin/flink run -q ./examples/batch/WordCount.jar
```

6、运行示例程序，在分离模式（detached mode）下：

```
$ ./bin/flink run -d ./examples/batch/WordCount.jar
```

7、运行示例程序，在特定的 JobManager 上：

```
$ ./bin/flink run -m myJMHost:8081 ./examples/batch/WordCount.jar \  
    --input file:///home/user/hamlet.txt \  
    --output file:///home/user/wordcount_out
```

8、运行示例程序，以特定的类作为入口点：

```
$ ./bin/flink run -c org.apache.flink.examples.java.wordcount.WordCount ./examples/batch/WordCount.jar \  
    --input file:///home/user/hamlet.txt \  
    --output file:///home/user/wordcount_out
```

9、运行示例程序，使用带有两个 TaskManager 的一个 per-job YARN 集群：

```
$ ./bin/flink run -m yarn-cluster -yn 2 ./examples/batch/WordCount.jar \  
    --input hdfs:///user/hamlet.txt \  
    --output hdfs:///user/wordcount_out
```

10、以 JSON 格式显示 WordCount 示例程序的优化执行计划：

```
$ ./bin/flink info ./examples/batch/WordCount.jar \  
    --input file:///home/user/hamlet.txt \  
    --output file:///home/user/wordcount_out
```

11、列出已调度和正在运行的作业（包括它们的 JobIDs）：

```
$ ./bin/flink list
```

12、列出已调度的作业（包括它们的 JobIDs）：

```
$ ./bin/flink list -s
```

13、列出正在运行的作业（包括它们的 JobIDs）：

```
$ ./bin/flink list -r
```

14、列出所有现有的作业（包括它们的 JobIDs）：

```
$ ./bin/flink list -a
```

15、列出在 Flink YARN session 内正在运行的 Flink 作业：

```
$ ./bin/flink list -m yarn-cluster -yid <yarnApplicationID> -r
```

16、取消一个作业：

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

17、取消一个作业，带有 savepoint:

```
$ ./bin/flink cancel -s [targetDirectory] <jobID>
```

18、停止一个作业（仅限于流作业）:

```
$ ./bin/flink stop <jobID>
```

19、修改一个正在运行的作业（仅限于流作业）:

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

1.5 本章小结

本章介绍了 Flink 的发展历程、特性及应用场景，并详细讲解了 Flink 的整体架构和运行时架构，以及 Flink 的计算资源和资源管理模式。最后演示了如何搭建 Flink 本地集群和完全分布式集群。

下一章，我们将准备 Flink 的开发环境，并开发和部署自己的第一个 Flink 程序。

1.6 本章习题

1、请描述 Flink 适合应用于哪些行业和场景？

2、Flink 是一个主从架构，Master 是 Flink 集群的主进程。它包含三个不同的组件：Resource Manager、Dispatcher 以及每个运行时 Flink 作业的 JobManager。请分别描述这三个组件的作用。

3、Flink 是一个主从架构，TaskManager 是 Flink 集群的工作（worker）进程。请描述 TaskManager 的作用。

4、Flink 是如何表示资源的？怎样尽可能地提高并行度以充分利用资源？

5、Flink 是如何管理资源的？

第 2 章 Flink 开发准备

这一章我们将准备好 Flink 开发环境，并尝试开发自己的第一个 Flink 程序，然后提交到 Flink 集群上运行。最后，我们将深入理解一些 Flink 的核心概念。

Flink 支持 Linux、OS X 和 Windows 作为 Flink 程序和本地执行的开发环境。对于 Flink 开发设置，需要安装以下软件：

- ☐ JDK 8 或以上版本；
- ☐ Apache Maven 3.x；
- ☐ 用于 Java(和/或 Scala)开发的 IDE。建议使用 IntelliJ IDEA。

2.1 安装和配置 Maven

Flink 项目可以使用不同的构建工具来构建，其中最常用的就是 Maven。为了快速开始开发，Flink 为构建工具 Maven 提供了项目模板。这些模板帮助我们设置 Flink 项目结构并创建初始构建文件。

Maven 可以安装在 Linux 和 Windows 等操作系统之上。这里我们在 Windows 下开发 Flink 应用程序，因此将 Maven 安装在 Windows 操作系统上。

（作者注：实际上，IntelliJ IDEA 自带开箱即用的 Maven，所以这一步可以不需要，读者可以直接跳过这一节，并不影响后续的学习。编写这一小节的主要目的是，如果有经验的程序员想要快速创建 Maven 项目，不需要启动 IntelliJ IDEA，通过本机安装配置好的 Maven，可以快速创建、打包和执行 Maven 项目。）

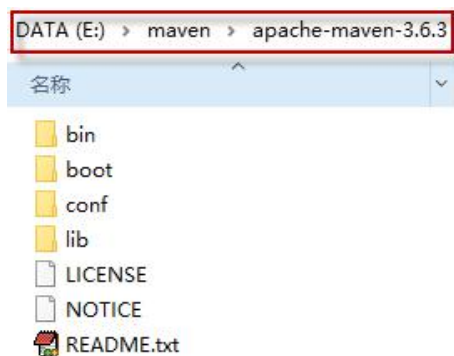
2.1.1 安装和配置 Maven

在 Windows 系统上安装 Maven 的步骤如下：

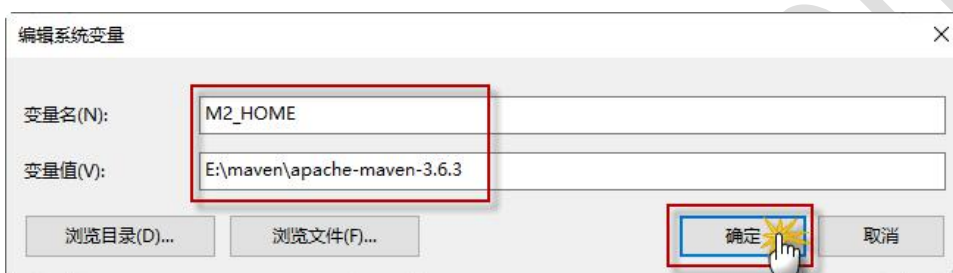
1、首先下载最新的 Maven 安装包。下载地址为：<http://maven.apache.org/download.cgi>。找到 bin.zip 压缩包的下载链接，点击即可下载。如下图所示：

Link	
Binary tar.gz archive	apache-maven-3.6.3-bin.tar.gz
Binary zip archive	apache-maven-3.6.3-bin.zip
Source tar.gz archive	apache-maven-3.6.3-src.tar.gz
Source zip archive	apache-maven-3.6.3-src.zip

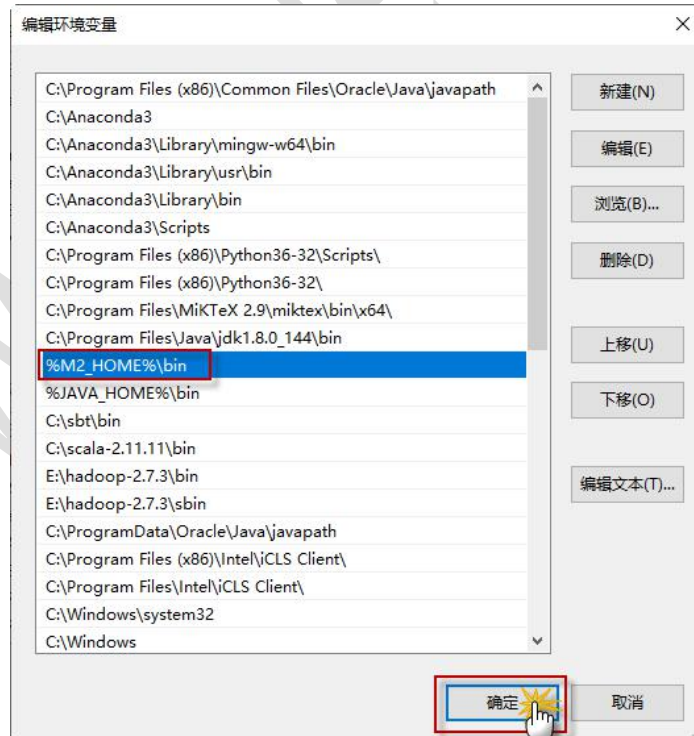
2、将 maven 安装包解压到指定目录。本人解压到了"E:\maven\apache-maven-3.6.3"目录下，该目录下包含有 bin、lib、conf 等文件夹。如下图所示：



3、配置 Maven 系统环境变量。依次选择"我的电脑--属性--高级系统设置--环境变量--系统变量---新建变量", 新建一个系统变量。设置变量名: M2_HOME, 变量值: E:\maven\apache-maven-3.6.3。如下图所示:



(4) 继续配置 Maven 系统环境变量。在系统变量中找到 Path 环境变量, 在变量值尾部加入 "%M2_HOME%\bin" (注意, 如果是 Win7 系统, 则需要在前面加上一个分号, 用来和其他路径值分隔开)。如下图所示:



(5) 检查 maven 的环境变量是否配置成功。打开命令行, 执行以下命令:

```
> mvn -v
```

如果能正确地输出 maven 版本信息, 则说明配置正确。如下图所示:

```
C:\Users\tongyao>mvn -v
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: E:\maven\apache-maven-3.6.3\bin\..
Java version: 1.8.0_241, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-8.0.640\bin\java.exe
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

2.1.2 修改 Maven 本地仓库默认位置

Maven 会将下载类库（jar 包）放置到本地的一个目录下（一般默认情况下 Maven 默认的本地仓库路径为 `${user.home}/.m2/repository`，其中 `${user.home}` 指的是当前用户主目录），如果想重新定义这个仓库目录的位置就需要修改 Maven 本地仓库的配置。

修改本地仓库默认位置的操作步骤如下：

- 1、在自己喜欢的位置创建文件夹，此处本人创建的位置是 "E:\maven\repository"。
- 2、在安装 Maven 的目录下找到 "conf/settings.xml" 文件，打开编辑，更改默认的仓库位置，并指定仓库的镜像位置。编辑内容如下图所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <localRepository>E:/maven/repository</localRepository>

  <mirrors>
    <mirror>
      <id>maven-repository</id>
      <mirrorOf>*</mirrorOf>
      <name>maven repo1</name>
      <url>https://repo.maven.apache.org/maven2/</url>
    </mirror>
  </mirrors>

  <profiles>
    <profile>
      <id>archetype</id>
```

- 3、要基于 Maven 的 Archetype 来快速创建一个新的 Flink 工程，还需要在 `<mirrors/>` 标签后，加上以下 `<profiles>` 内容和 `<activeProfiles>` 内容。如下图所示：（Archetype 是 Maven 工程的模板工具包，会帮助用户创建 Maven 工程模板，并给用户提供生成相关工程模板版本的参数化方法）


```

<profiles>
  <profile>
    <id>archetype</id>
    <repositories>
      <repository>
        <id>archetype</id>
        <url>https://repo1.maven.org/maven2/</url>
        <releases>
          <enabled>true</enabled>
          <checksumPolicy>fail</checksumPolicy>
        </releases>
        <snapshots>
          <enabled>true</enabled>
          <checksumPolicy>warn</checksumPolicy>
        </snapshots>
      </repository>
    </repositories>
  </profile>
</profiles>
1

<activeProfiles>
  <activeProfile>archetype</activeProfile>
</activeProfiles>
2

</settings>

```

4、最后，settings.xml 文件的完整内容如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd">

  <localRepository>E:/maven/repository</localRepository>

  <mirrors>
    <mirror>
      <id>maven-repository</id>
      <mirrorOf>*</mirrorOf>
      <name>maven repo1</name>
      <url>https://repo.maven.apache.org/maven2/</url>
    </mirror>
  </mirrors>

  <profiles>
    <profile>
      <id>archetype</id>
      <repositories>
        <repository>
          <id>archetype</id>
          <url>https://repo1.maven.org/maven2/</url>
          <releases>
            <enabled>true</enabled>
            <checksumPolicy>fail</checksumPolicy>

```

```
</releases>
<snapshots>
  <enabled>true</enabled>
  <checksumPolicy>warn</checksumPolicy>
</snapshots>
</repository>
</repositories>
</profile>
</profiles>

<activeProfiles>
  <activeProfile>archetype</activeProfile>
</activeProfiles>
</settings>
```

5、验证修改是否生效。打开命令行，执行以下命令：

```
> mvn help:system
```

执行完该命令之后，在"E:\maven\repository"下面会出现很多文件，这些文件就是 maven 从中央仓库下载到本地仓库的文件。

2.1.3 创建初始模板项目

在命令行中，使用如下的命令快速创建一个空白 Flink 项目：

```
> cd FlinkProjects // 在指定的工作目录下
> mvn archetype:generate \
  -DarchetypeGroupId=org.apache.flink \
  -DarchetypeArtifactId=flink-quickstart-scala \
  -DarchetypeCatalog=local \
  -DarchetypeVersion=1.10.0
```

注：如果要创建一个 Java 语言的空白 Flink，只需要将上面构建命令中的 flink-quickstart-scala 换成 flink-quickstart-java 即可。其他操作都相同。

如果很慢的话，可以把"http://repo1.maven.org/maven2/archetype-catalog.xml"下载到本地仓库根目录，比如~/.m2下，并在执行"mvn archetype"命令时加上"-DarchetypeCatalog=internal"。

这会在"FlinkProjects"目录下创建一个 Flink 模板项目。在创建过程中，Maven 会交互式地询问 groupId、artifactId 和 package 名称。这里我分别输入以下名称：

- ☐ groupId: com.xueai8
- ☐ artifactId: FlinkJavaBlank
- ☐ package: com.xueai8

创建过程如下图所示：

```

E:\>cd FlinkProjects
E:\FlinkProjects>mvn archetype:generate -DarchetypeGroupId=org.apache.flink -DarchetypeArtifactId=flink-quickstart-java -DarchetypeCatalog=local -DarchetypeVersion=1.10.0
[INFO] Scanning for projects...
[INFO] -----< org.apache.maven:standalone-pom >-----
[INFO] Building Maven Stub Project (No POM) 1
[INFO] -----[ pom ]-----
[INFO] >>> maven-archetype-plugin:3.1.2:generate (default-cli) > generate-sources @ standalone-pom >>>
[INFO] <<< maven-archetype-plugin:3.1.2:generate (default-cli) < generate-sources @ standalone-pom <<<
[INFO] --- maven-archetype-plugin:3.1.2:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype repository not defined. Using the one from [org.apache.flink:flink-quickstart-java:1.10.0] found in catalog local
Define value for property 'groupId': com.xueai8
Define value for property 'artifactId': FlinkJavaBlank
Define value for property 'version': 1.0-SNAPSHOT: :
Define value for property 'package': com.xueai8: :
Confirm properties configuration:
groupId: com.xueai8
artifactId: FlinkJavaBlank
version: 1.0-SNAPSHOT
package: com.xueai8
Y: :
[INFO] Using following parameters for creating project from Archetype: flink-quickstart-java:1.10.0
[INFO] Parameter: groupId, Value: com.xueai8
[INFO] Parameter: artifactId, Value: FlinkJavaBlank
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: package, Value: com.xueai8
[INFO] Parameter: packageInPathFormat, Value: com/xueai8
[INFO] Parameter: package, Value: com.xueai8
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: groupId, Value: com.xueai8
[INFO] Parameter: artifactId, Value: FlinkJavaBlank
[WARNING] CP Don't override file E:\FlinkProjects\FlinkJavaBlank\src\main\resources
[INFO] Project created from Archetype in dir: E:\FlinkProjects\FlinkJavaBlank
[INFO] BUILD SUCCESS
[INFO] Total time: 42.886 s
[INFO] Finished at: 2020-04-02T11:29:49+08:00
[INFO]

```

创建Flink模板项目的命令

交互创建过程中，会询问groupId、artifactId、version和package等信息。除了groupId和artifactId，其他项可直接回车确认使用默认值

工作目录中将有一个新目录，目录名称是 artifactId 的名称。在命令行中可以使用下面的命令查看项目结构：

```
> tree FlinkJavaBlank
```

项目结构如下所示：

```

E:\FlinkProjects>tree FlinkJavaBlank
卷 DATA 的文件夹 PATH 列表
卷序列号为 02D7-8C8C
E:\FLINKPROJECTS\FLINKJAVABLANK
├── src
│   └── main
│       ├── java
│       │   ├── com
│       │   └── xueai8
│       └── resources

```

生成的项目结构

示例项目是一个 Maven 项目，它包含两个类：StreamingJob 和 BatchJob，分别是 DataStream 和 DataSet 程序的基本框架程序。类中的 main 方法是程序的入口点，用于内部测试/执行和适当的部署。这个基本框架程序然后可以导入到 IDE 中进行开发。

2.1.4 构建打包项目

如果想构建/打包 FlinkJavaBlank 项目以便部署，需要运行"mvn clean package"命令。请按以下步骤操作：

1、转到项目目录。在命令行中执行以下命令：

```
> cd E:\FlinkProjects\FlinkJavaBlank
> mvn clean package
```

2、在项目目录下，会生成一个 target 目录。我们打包的结果文件就放在这个目录中，可以转到相应的文件夹去查看，也可以直接在命令行下执行 dir 命令查看，如下所示：

```
E:\FlinkProjects\FlinkJavaBlank>dir
驱动器 E 中的卷是 DATA
卷的序列号是 02D7-8C8C

E:\FlinkProjects\FlinkJavaBlank 的目录
2020/04/02  12:10    <DIR>          .
2020/04/02  12:10    <DIR>          ..
2020/04/02  11:29             6,637 pom.xml
2020/04/02  11:29    <DIR>          src
2020/04/02  12:11    <DIR>          target
                1 个文件          6,637 字节
                4 个目录 114,368,802,816 可用字节
```

2、查看 target 文件夹，将会发现一个 JAR 文件，其中包含我们应用程序以及相关的依赖项（连接器和库）。如下所示：

```
E:\FlinkProjects\FlinkJavaBlank>dir target
驱动器 E 中的卷是 DATA
卷的序列号是 02D7-8C8C

E:\FlinkProjects\FlinkJavaBlank\target 的目录
2020/04/02  12:11    <DIR>          .
2020/04/02  12:11    <DIR>          ..
2020/04/02  12:11    <DIR>          classes
2020/04/02  12:11             5,565 FlinkJavaBlank-1.0-SNAPSHOT.jar
2020/04/02  12:11    <DIR>          generated-sources
2020/04/02  12:11    <DIR>          maven-archiver
2020/04/02  12:11    <DIR>          maven-status
2020/04/02  12:11             5,323 original-FlinkJavaBlank-1.0-SNAPSHOT.jar
                2 个文件          10,888 字节
                6 个目录 114,368,802,816 可用字节
```

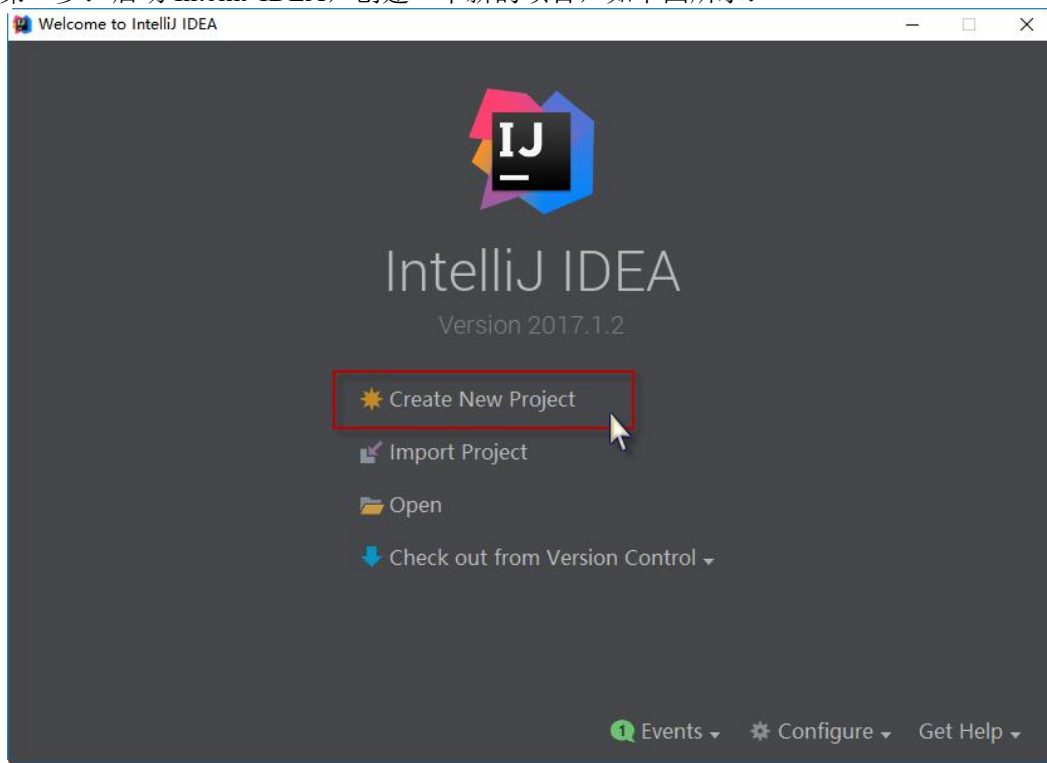
打包后可部署的jar包

2.2 使用 IntelliJ IDEA+Maven 开发 Flink 项目

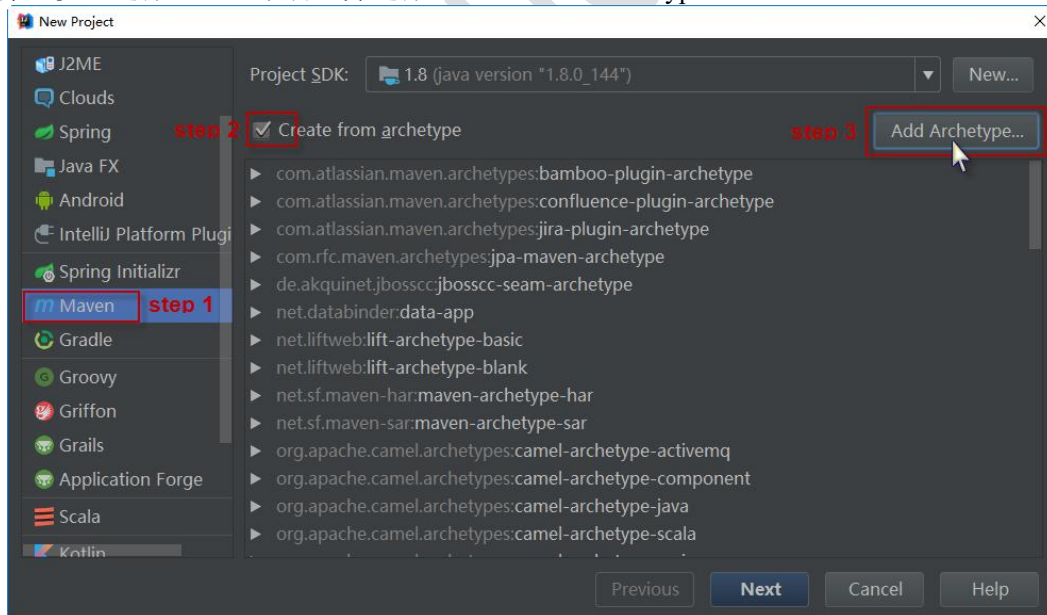
对于熟悉 Maven 操作的读者来说，可以象上一节讲的那样，先在命令行用 Maven 命令快速生成 Flink 基本程序框架，然后导入到 IDE 中进一步开发。如果读者更熟悉 IDE（例如，IntelliJ IDEA）的话，那么可以直接在 IntelliJ IDEA 中开发 Maven 程序。通过结合使用 IntelliJ IDEA 集成开发工具和 Maven 项目构建工具，我们可以方便快速地开发 Flink 项目。

2.2.1 在 IntelliJ IDEA 中创建 Flink 项目

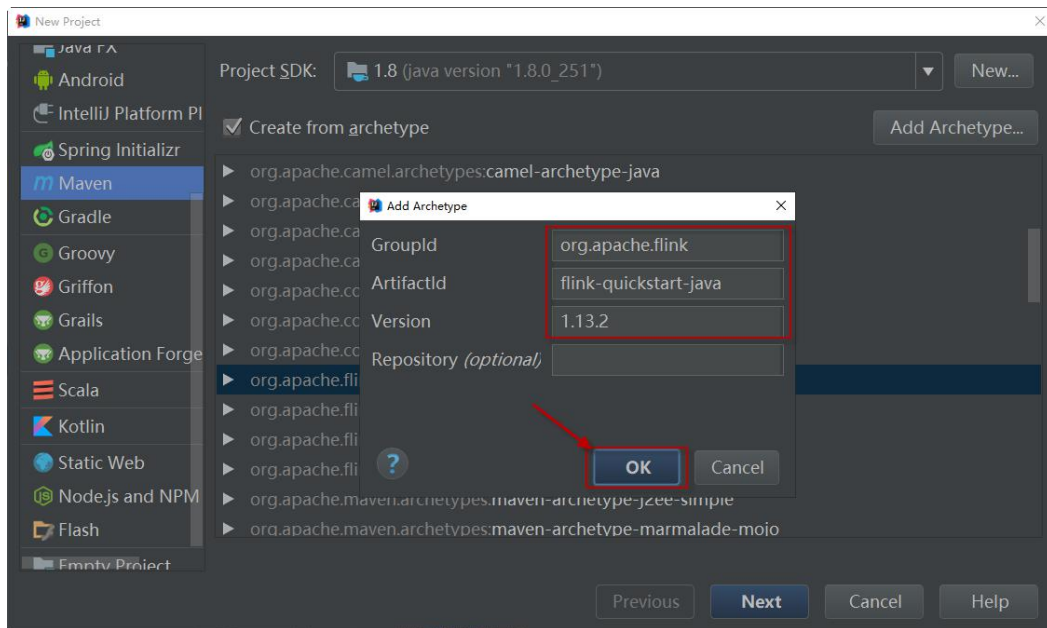
第一步：启动 IntelliJ IDEA，创建一个新的项目，如下图所示：



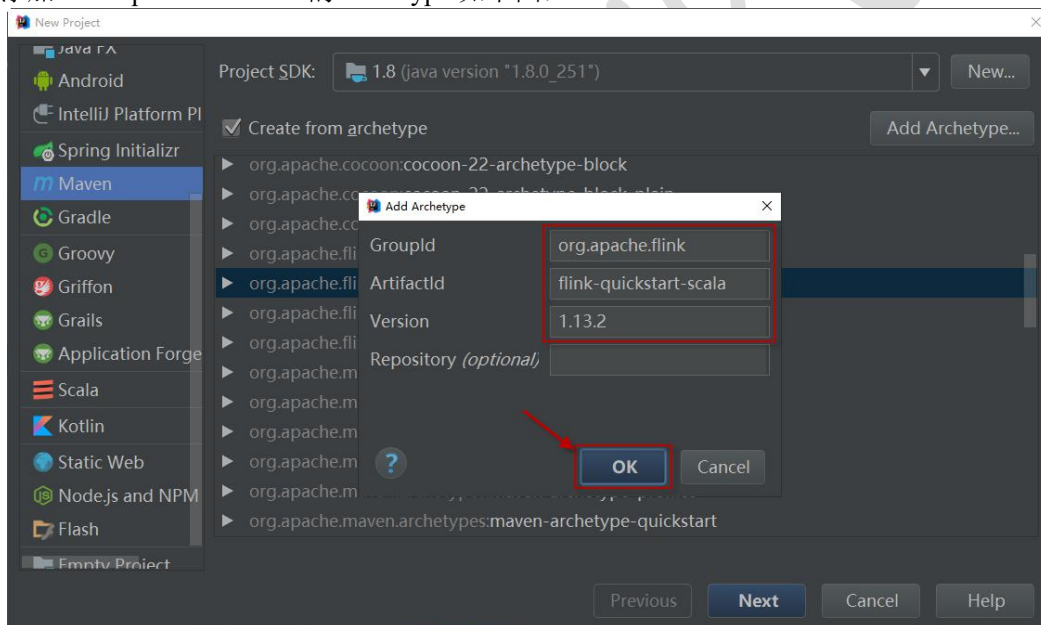
第二步：选择 Maven 项目，并选择 “Create from archetype”：



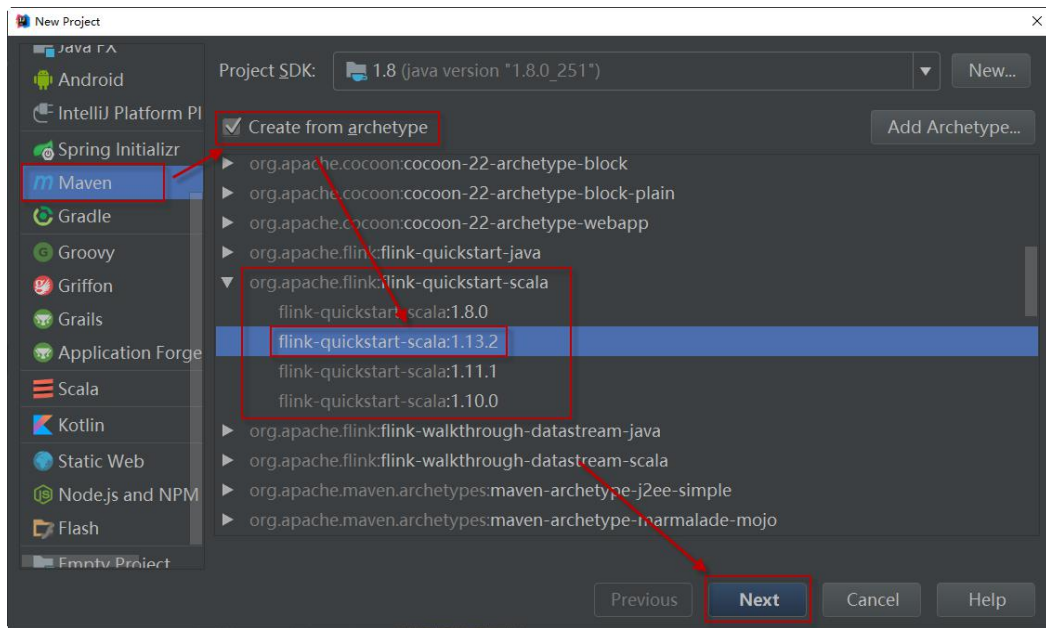
第三步：因为默认没有 Flink 的 archetype，所以需要自己添加。
添加 flink-quickstart-java 的 archetype 如下图：



添加 flink-quickstart-scala 的 archetype 如下图：

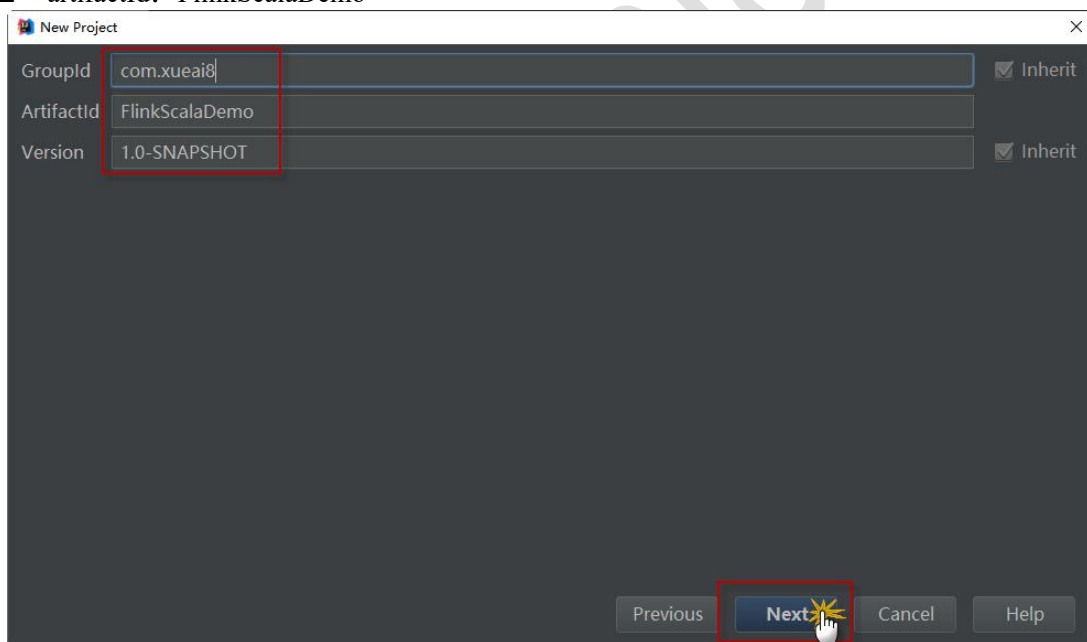


第四步：选择对应的 archetype，比如，这里我选择 flink-quickstart-scala，如下所示：

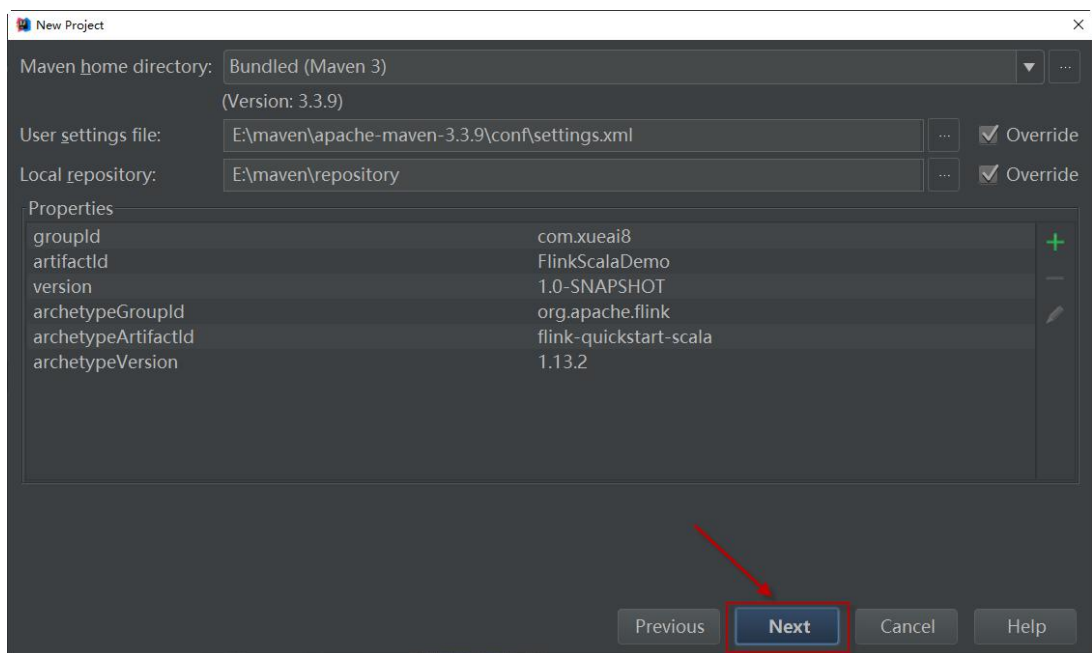


第五步：指定项目的 groupId、artifactId 名称。这里我分别取以下名称：

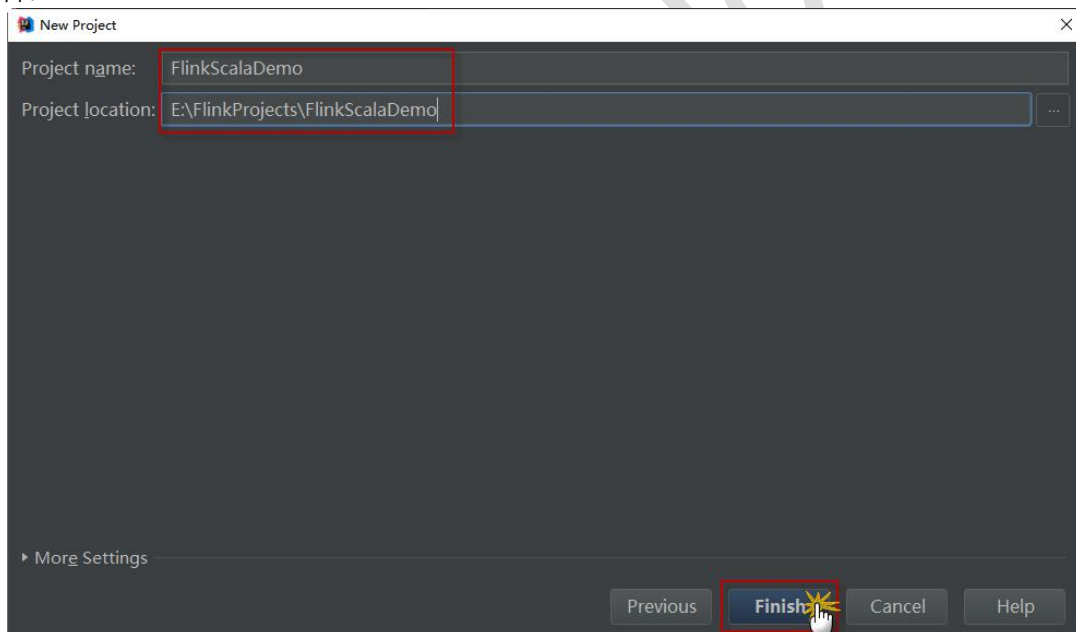
- ☐ groupId: com.xueai8
- ☐ artifactId: FlinkScalaDemo



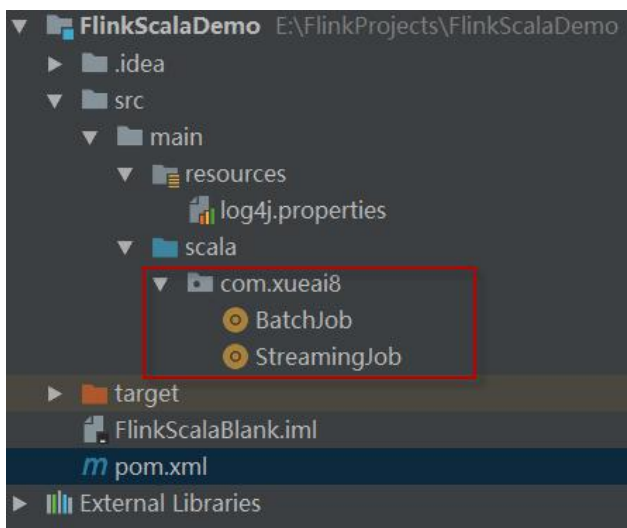
第六步：接下来，指定项目的 Maven 配置，默认就好。



第七步：指定项目的名称和项目文件所在位置。这里保持默认即可。单击【Finish】按钮，开始创建项目：



第八步：Maven 会自动构建项目，最后的项目结构如下所示：



可以看出，flink-quickstart-scala 快速地构建了一个基本的 Flink 项目框架，并生成创建了两个模板程序文件：用于流处理的 StreamingJob 和用于批处理的 BatchJob。

注：同样的步骤，选择 flink-quickstart-java，创建一个基于 Java API 的 Flink 项目框架。请自行尝试。

2.2.2 设置项目基本依赖

每个 Flink 应用程序都依赖于一些核心的 API 库。对于 Maven，我们可以使用 Java 项目模板（flink-quickstart-java）或 Scala 项目模板（flink-quickstart-scala）创建具有这些初始依赖项的程序框架。

如果没有使用项目模板而手动设置项目时，需要为 Java/Scala API 添加以下依赖项(这里以 Maven 语法表示，但是同样的依赖项也适用于其他构建工具(Gradle、SBT 等))。

Scala Maven 依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
```

Java Maven 依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-java</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
```

```
<artifactId>flink-streaming-java_2.12</artifactId>
<version>1.13.2</version>
<scope>provided</scope>
</dependency>
```

重要提示：

请注意，所有这些依赖项的作用域（scope）都被设置为 provided。这意味着需要对它们进行编译，但是不应该将它们打包到项目的最终应用程序 jar 文件中。这些依赖项是 Flink 核心依赖项，Flink 运行时环境已经包含了这些依赖，因此不需要把它们打到 jar 包中。

强烈建议将该依赖关系保持在 provided 范围内。如果没有将它们设置为 provided，那么最好的情况是生成的 JAR 变得非常大，因为它还包含所有 Flink 核心依赖项。最坏的情况是添加到应用程序 jar 文件中的 Flink 核心依赖项与自己的一些依赖项版本冲突(通常通过反向类加载来避免)。

IntelliJ IDEA 注意：

要使 Flink 应用程序在 IntelliJ IDEA 中运行，需要在 scope compile 中声明 Flink 依赖项，而不是 provided。否则 IntelliJ 将不会将它们添加到类路径中，而 IDE 中的执行将会失败，并抛出 NoClassDefFoundError 错误信息。

2.2.3 用于构建具有依赖项的 Jar 的模板

要构建包含声明的连接器和库所需的所有依赖项的应用程序 JAR，可以使用下面的 shade 插件定义：

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>3.1.1</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals>
            <goal>shade</goal>
          </goals>
          <configuration>
            <artifactSet>
              <excludes>
                <exclude>com.google.code.findbugs.jsr305</exclude>
                <exclude>org.slf4j:*</exclude>
                <exclude>log4j:*</exclude>
              </excludes>
            </artifactSet>
            <filters>
              <filter>
                <artifact>*</artifact>
                <excludes>
                  <exclude>META-INF/*.SF</exclude>
                  <exclude>META-INF/*.DSA</exclude>
                  <exclude>META-INF/*.RSA</exclude>
                </excludes>
              </filter>
            </filters>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

```
        </filter>
      </filters>
      <transformers>
        <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
          <mainClass>com.xueai8.StreamingJob</mainClass>
        </transformer>
      </transformers>
    </configuration>
  </execution>
</executions>
</plugin>
</plugins>
</build>
```

2.2.4 编写批处理代码并测试执行

以上一节所创建项目中的 BatchJob 源文件为模板，进一步编写一个简单的批处理代码并执行。这里的目的是为了掌握使用 IntelliJ IDEA 创建 Flink Maven 项目，所以不必理解代码，在后续的章节中会详细讲解。

在 IntelliJ IDEA 中打开 BatchJob 源文件，编辑代码如下。

Scala 代码：

```
package com.xueai8

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

object BatchJob {

  def main(args: Array[String]) {
    // 设置批执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    // 得到输入数据
    val text = env.fromElements("good good study", "day day up")

    // 对数据进行转换
    val counts = text.flatMap { _.toLowerCase.split("\\W+") }
                        .map { (_, 1) }
                        .groupBy(0)
                        .sum(1)

    // 执行并输出结果
    counts.print()
  }
}
```

Java 代码：

```
package com.xueai8;
```

```

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.util.Collector;

/**
 * 要打包应用程序为 JAR 文件，在命令行执行如下命令：
 * $ mvn clean package
 *
 * 如果改变了主类的名称，那么需要在 pom.xml 文件中做相应的修改（简单搜索'mainClass'）
 */
public class BatchJob {

    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // 读取数据源
        DataSet<String> text = env.fromElements("Good good study", "Day day up");

        // 对数据集进行转换
        DataSet<Tuple2<String,Integer>> result = text
            .flatMap(new FlatMapFunction<String, Tuple2<String,Integer>>() {
                @Override
                public void flatMap(String s, Collector<Tuple2<String,Integer>> out) throws Exception {
                    String line = s.toLowerCase();
                    for(String word : line.split(" ")){
                        out.collect(new Tuple2<>(word, 1));
                    }
                }
            })
            .groupBy(0)
            .sum(1);

        result.print();
    }
}

```

在文件内任何空白处，单击右键，在弹出菜单中选择“run BatchJob”，执行该程序，在下方的运行窗口可以看到如下输出结果：

```

(up,1)
(day,2)
(good,2)
(study,1)

```

注 1：

在执行此程序时，有可能会遇到如下异常：

```

.....
Caused by: java.lang.ClassNotFoundException: org.apache.flink.api.scala.typeutils.CaseClassTypeInfo

```



```
at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:335)
at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
... 13 more
```

异常的原因是 maven 引入依赖问题。打开项目中的 pom.xml 文件，将其中依赖部分的 "`<scope>provided</scope>`" 注释（或删除）掉。如下所示：

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

注 2：

在 IDE 中运行项目可能会导致 `java.lang.NoClassDefFoundError` 异常。这可能是因为将所有必需的 Flink 依赖项隐式加载到类路径中。

解决办法：在 IntelliJ IDEA 中，选择 `Run > Edit Configurations > Modify options`，然后选择“include dependencies with "Provided" scope”。这个运行配置现在将包括从 IDE 中运行应用程序所需的所有类。

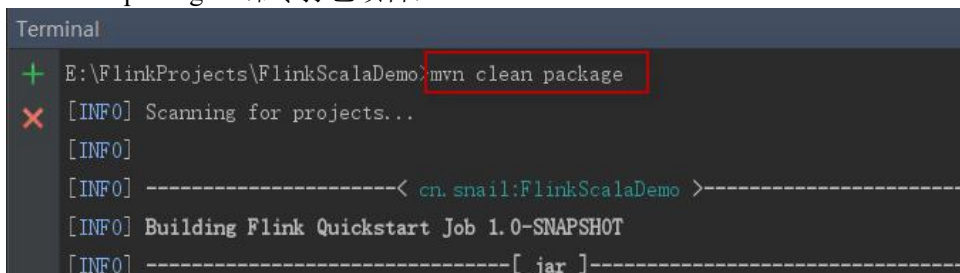
2.2.5 项目打包并提交 Flink 集群执行

项目如果想要部署到生产环境运行，那么就要先打成 jar 包再进行部署。我们使用 Maven 的“`mvn clean package`”命令可以很方便地进行打包。具体操作步骤如下：

1、打开项目中的 pom.xml 文件，找到以下内容并修改 `mainClass`（可 `Ctrl + F`，查找查找 `mainClass`）为当前类的全限定名称：

```
<transformers>
  <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
    <mainClass>com.xueai8.BatchJob</mainClass>
  </transformer>
</transformers>
```

2、在 IntelliJ IDEA 界面最下方左侧，点击 Terminal 选项卡，打开 Terminal 窗口。在 Terminal 窗口中执行命令“`mvn clean package`”命令打包项目：



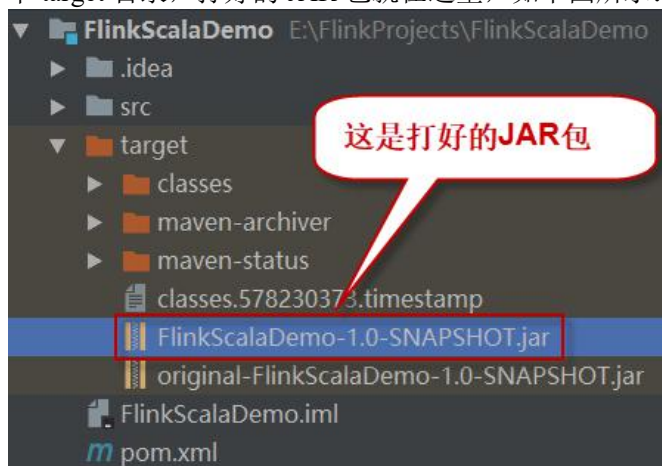
```
Terminal
+ E:\FlinkProjects\FlinkScalaDemo>mvn clean package
[INFO] Scanning for projects...
[INFO]
[INFO] -----< cn.snail:FlinkScalaDemo >-----
[INFO] Building Flink Quickstart Job 1.0-SNAPSHOT
[INFO] -----[ jar ]-----
```

3、会编译打包过程中会输出一系列信息，如下所示：

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.447 s
[INFO] Finished at: 2019-05-14T21:02:59+08:00
[INFO] -----
```

看到这个信息，说明打包成功

4、在项目下会生成一个 target 目录，打好的 JAR 包就在这里，如下图所示：



5、在 Linux 终端中，执行以下命令，启动 Flink 集群：

```
$ cd ~/bigdata/flink-1.13.2/
$ ./bin/start-cluster.sh
```

过程如下图所示：

```
[hduser@hadoop ~]$ cd ~/bigdata/flink-1.10.0/
[hduser@hadoop flink-1.10.0]$ ./bin/start-cluster.sh
Starting cluster.
Starting standalone session daemon on host hadoop.
Starting taskexecutor daemon on host hadoop.
[hduser@hadoop flink-1.10.0]$
```

启动Flink集群

6、将该 JAR 拷贝到 Linux 的指定目录下，例如，“~/flinkdemos/”目录下。然后提交到 Flink 集群上运行 Job 作业：

```
$ cd ~/bigdata/flink-1.13.2/
$ ./bin/flink run --class com.xueai8.BatchJob ~/flinkdemos/FlinkScalaDemo-1.0-SNAPSHOT.jar
```

执行结果如下所示：

```
(day,2)
(good,2)
(study,1)
(up,1)
```

2.3 Flink 相关概念

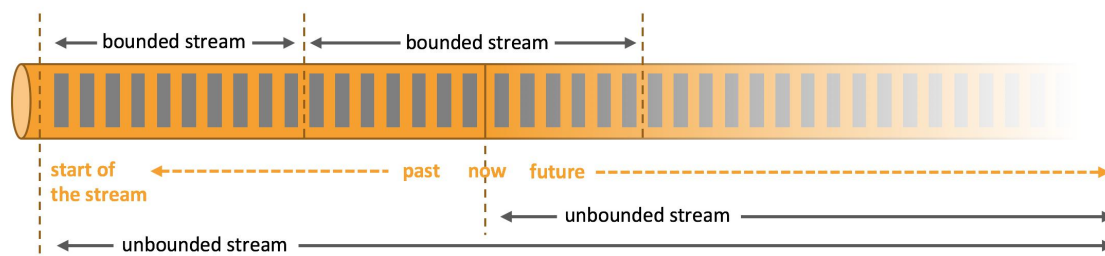
在上一小节，我们准备好了 Flink 应用程序开发环境，尝试编写了自己的第一个 Flink 应用程序，并打包为作业提交给 Flink 集群去执行。在进一步深入学习 Flink 技术之前，我们有必要来了解一些 Flink 的核心概念。

2.3.1 Flink 数据流

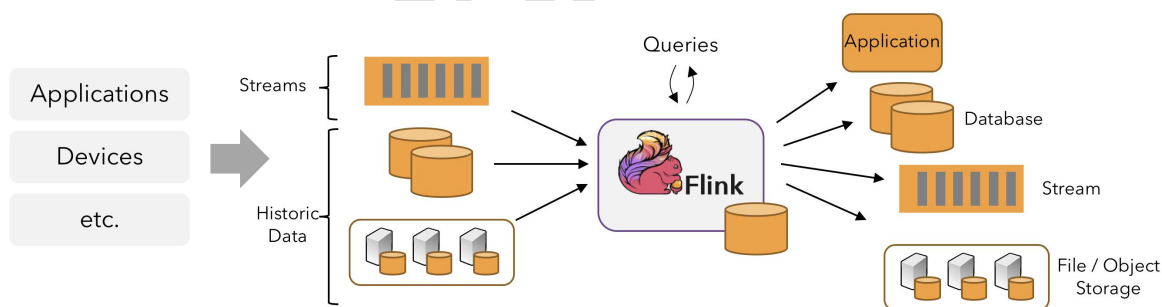
在 Flink 中，应用程序由数据流组成，这些数据流可以由用户定义的运算符（注：有时我们称这些运算符为“算子”）进行转换。这些数据流形成有向图，从一个或多个源开始，以一个或多个输出结束。



Flink 支持流处理和批处理，它是一个分布式的流批结合的大数据处理引擎。在 Flink 中，认为所有的数据本质上都是随时间产生的流数据，把批数据看作是流数据的特例，只不过流数据是一个无界的数据流，而批数据是一个有界的数据流（例如固定大小的数据集）。如下图所示：



因此，Flink 是一个用于在无界和有界数据流上进行有状态计算的通用的处理框架，它既具有处理无界流的复杂功能，也具有专门的运算符来高效地处理有界流。通常我们称无界数据为实时数据，来自消息队列或分布式日志等流源（如 Apache Kafka 或 Kinesis）。而有界数据，通常指的是历史数据，来自各种数据源（如文件、关系型数据库等）。由 Flink 应用程序产生的结果流可以发送到各种各样的系统，并且可以通过 REST API 访问 Flink 中包含的状态。

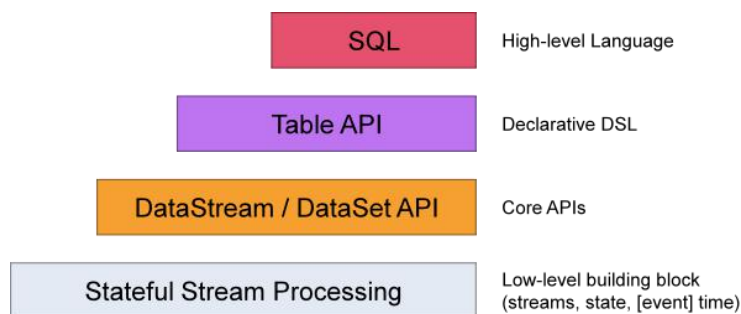


当 Flink 处理一个有界的数据流时，就是采用的批处理工作模式。在这种操作模式中，我们可以选择先读取整个数据集，然后对数据进行排序、计算全局统计数据或生成总结所有输入的最终报告。

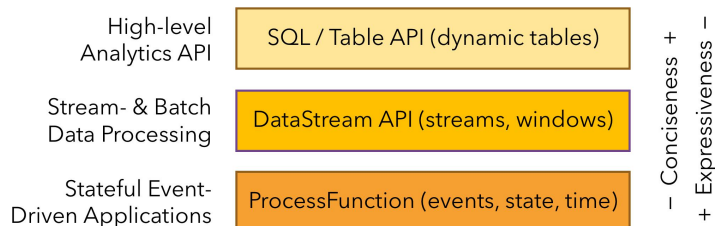
当 Flink 处理一个无界的数据流时，就是采用的流处理工作模式。对于流数据处理，输入可能永远不会结束，因此我们必须在数据到达时持续不断地对这些数据进行处理。

2.3.2 Flink 分层 API

Flink 提供了开发流/批处理应用程序的不同抽象层次。如下图所示：



Flink 提供了三个分层的 API。每个 API 在简洁性和表达性之间提供了不同的权衡，并针对不同的应用场景。



最低级别抽象只提供有状态流，它通过 `ProcessFunction` 嵌入到 `DataStream API` 中。`ProcessFunction` 是 Flink 提供的最富表现力的函数接口，它允许用户自由处理来自一个或多个流的事件，并使用一致的容错状态。`ProcessFunction` 提供了对时间和状态的细粒度控制。`ProcessFunction` 可以任意修改它的状态并注册定时器，从而在将来触发回调函数，以实现复杂的计算。因此，`ProcessFunction` 可以根据许多有状态事件驱动的应用程序的需要实现每个事件的复杂业务逻辑。

实际上，大多数应用程序不需要上面描述的低层抽象，而是根据核心 API（`DataStream API`(流处理)和 `DataSet API`(批处理)）进行编程。`DataStream API` 是数据驱动应用程序和数据管道的主要 API，`DataStream API` 为许多常见的流处理操作提供基本类型，比如窗口化、record-at-a-time 转换和通过查询外部数据存储来丰富事件。`DataStream API` 支持 Java 和 Scala 语言，它基于 `map()`、`reduce()` 和 `aggregate()` 等函数。函数可以通过扩展接口或作为 Java 或 Scala lambda 函数来定义。使用核心 API 开发的 Flink 应用程序可以显式地控制时间和状态(状态、触发器、proc、fun 等)。

低层 `ProcessFunction` 与 `DataStream API` 集成，使得仅对某些操作进行低层抽象成为可能。而 `DataSet API` 在有界数据集上提供了额外的原语，支持循环/迭代运算。

Flink 有两个关系 API，Table API 和 SQL。这两个 API 都是用于批处理和流处理的统一 API，即查询在无界、实时流或有界、记录流上以相同的语义执行，并产生相同的结果。

Table API 是一个以表为中心的声明性 DSL，它可以(在表示流时)动态地更改表。Table API 遵循(扩展的)关系模型：表有一个附加的模式(类似于关系数据库中的表)，API 提供了类似的关系操作，比如 `select`、`project`、`join`、`group-by`、`aggregate` 等。Table API 程序声明应该执行什么逻辑操作，而不是明确指定操作代码的形式。虽然 Table API 可以通过各种类型的用户定义函数(UDF)进行扩展，使用起来更简洁(编写的代码更少)，但是该 API 没有提供对时间和状态的直接访问，与核心 API 相比缺乏表现力。此外，Table API 程序还需要经过一个在执行之前应用优化规则的优化器，从而应用自动优化。可以在 Table 和 DataStream/DataSet 之间无缝转换，Flink 允许程序编程时混合 Table API 和 DataStream/DataSet API。

Flink 提供的最高级抽象是 SQL。这种抽象在语义和表达性上都类似于 Table API，但将程序表示为 SQL 查询表达式。SQL 抽象与 Table API 紧密交互，SQL 查询可以在 Table API 中定义的表上执行。

Table API 和 SQL 利用 Apache Calcite 进行解析、验证和查询优化。它们可以与 DataStream 和 DataSet API 无缝集成，并支持用户定义的标量、聚合和表值（table-valued）函数。Table API / SQL 正在以跨批处理和流的统一方式成为分析应用场景下的主要 API，旨在简化数据分析、数据管道和 ETL 应用程序的定义。

2.4 本章小结

本章首先讲解了如何准备开发环境。一般来说，可以两种方式开发。一种方式是直接使用 Maven 来快速构建和打包项目，本章做了详细的讲解，包括 Maven 的安装和配置、修改本地仓库默认位置、创建初始模板项目以及如何构建打包项目。另一种方式是使用集成开发环境（IDE），例如 IntelliJ IDEA，创建 Maven 项目，本章也 step by step 地讲解整个过程。

本章最后还详细讲解了 Flink 相关的一些重要概念。在 Flink 中，应用程序由数据流组成，这些数据流可以由用户定义的运算符进行转换。这些数据流形成有向图，以一个或多个源开始，以一个或多个输出结束。在 Flink 中，认为所有的数据本质上都是随时间产生的流数据，把批数据看作是流数据的特例，只不过流数据是一个无界的数据流，而批数据是一个有界的数据流。另外还讲解了 Flink 的两种作业提交模式：Session 模式和 Per-Job 模式。Flink 也提供了自动错误恢复（Failover）机制来进行重试。

下一章，我们将介绍如何开发 Flink 实时数据处理程序。

2.5 本章习题

- 1、请阐述 Flink 的“流批合一”概念。
- 2、请描述 Flink 中作业提交和调度的过程，并图示说明。
- 3、请阐述 Flink 的错误恢复机制。

第 3 章 开发 Flink 实时数据处理程序

实时分析是当前非常重要的一个问题。许多不同的领域需要实时处理数据。例如，来自物联网(IoT)的应用程序需要实时或近实时地存储、处理和分析数据。

为了满足对实时数据处理的需求，Apache Flink 提供了一个名为 DataStream API 的流数据处理 API，用于构建健壮的、有状态的流应用程序。它提供了对状态和时间的细粒度控制，允许实现高级事件驱动系统。

Flink 的流处理 API（DataStream API）位于 `org.apache.flink.streaming.api.scala` 包（Scala 语言）或 `org.apache.flink.streaming.api` 包（Java 语言）中。

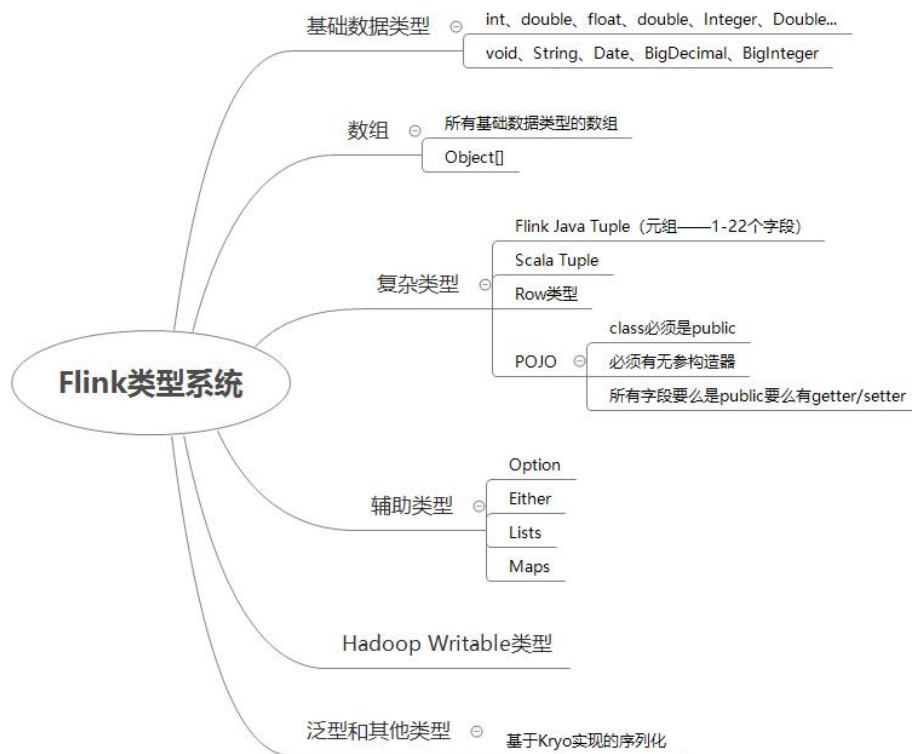
3.1 Flink 流处理程序编程模型

在深入了解 Flink 实时数据处理程序的开发之前，我们先通过一个简单示例来了解使用 Flink 的 DataStream API 构建有状态流应用程序的过程。

3.1.1 流数据类型

Flink 以一种独特的方式处理数据类型和序列化，它包含自己的类型描述符、泛型类型提取和类型序列化框架。基于 Java 和 Scala 语言，Flink 实现了一套自己的类型系统，它支持很多种类的类型：

- ☐ 基本类型
- ☐ 数组类型
- ☐ 复合类型
- ☐ 辅助类型
- ☐ 通用类型



Flink 针对 Java 和 Scala 的 DataStream API 要求流数据的内容必须是可序列化的。Flink 内置了以下类型数据的序列化器：

- ❑ 基本数据类型：String、Long、Integer、Boolean、Array。
- ❑ 复合数据类型：Tuple、POJO、Scala case class。

对于其他类型，Flink 会返回到 Kryo。也可以在 Flink 中使用其他序列化器。Avro 尤其得到了很好的支持。

Java DataStream API 使用的流数据类型

对于 Java API，Flink 定义了自己的 Tuple1 到 Tuple25 类型来表示元组类型：

```
Tuple2<String, Integer> person = new Tuple2<>("王老五", 35);
```

```
// 索引基于 0
```

```
String name = person.f0;
```

```
Integer age = person.f1;
```

在 Java 中，POJO(plain old Java Object)是这样的 Java 类：

- ❑ 有一个无参的默认构造器；
- ❑ 所有的字段要么是 public 的，要么有一个默认的 getter 和 setter。

下面定义的是一个 Person POJO：

```
// 定义一个 Person POJO 类
```

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

```
    public Person() {};
```

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

// 创建一个实例
Person person = new Person("王老五", 35);
```

Scala DataStream API 使用的流数据类型

对于元组，使用 Scala 自己的 Tuple 类型就好。

```
val person = ("王老五", 35)
```

```
// 索引基于 1
```

```
val name = person._1
```

```
val age = person._2
```

对于对象类型，使用 case class（相当于 Java 中的 JavaBean）：

```
case class Person(name: String, age: Int)
```

```
val person = Person("王老五", 35)
```

Flink 类型系统

对于创建的任意一个 POJO 类型，看起来它是一个普通的 Java Bean，在 Java 中，可以使用 Class 来描述该类型。但其实在 Flink 引擎中，它被描述为 PojoTypeInfo，而 PojoTypeInfo 是 TypeInformation 的子类。

TypeInformation 是 Flink 类型系统的核心类。Flink 使用 TypeInformation 来描述所有 Flink 支持的数据类型，就像 Java 中的 Class 类型一样。每一种 Flink 支持的数据类型都对应的是 TypeInformation 的子类。例如：POJO 类型对应的是 PojoTypeInfo、基础数据类型数组对应的是 BasicArrayTypeInfo、Map 类型对应的是 MapTypeInfo、值类型对应的是 ValueTypeInfo。

除了对类型的描述，TypeInformation 还提供了序列化的支持。在 TypeInformation 中有一个方法：createSerializer 方法，它用来创建序列化器，序列化器中定义了一系列的方法。其中，通过 serialize 和 deserialize 方法，可以将指定类型进行序列化。并且，Flink 的这些序列化器会以稠密的方式来将对象写入到内存中。Flink 中也提供了非常丰富的序列化器。在我们基于 Flink 类型系统支持的数据类型进行编程时，Flink 在运行时会推断出数据类型的信息，程序员在基于 Flink 编程时，几乎是不需要关心类型和序列化的。

类型与 Lambda 表达式支持

在编译时，编译器能够从 Java 源代码中读取到完整的类型信息，并强制执行类型的约束，但生成 class 字节码时，会将参数化类型信息删除。这就是类型擦除。类型擦除可以确保不会为泛型创建新的 Java 类，泛型是不会产生额外的开销的。

注意：泛型只是在编译器编译时能够理解该类型。但编译后执行时，泛型是会被擦除掉的。

用以下代码来做一个简单说明：

```
public static <T> boolean hasItems(T [] items, T item){
```

```
    for (T i : items){
        if(i.equals(item)){
            return true;
        }
    }
    return false;
}
```

以上是一段 Java 的泛型方法。但在编译后，编译器会将未绑定类型的 T 擦除掉，替换为 Object。也就是变成下面的样子：

```
public static Object boolean hasItems(Object [] items, Object item){
    for (Object i : items){
        if(i.equals(item)){
            return true;
        }
    }
    return false;
}
```

泛型只是能够防止在运行时出现类型错误。但运行时会出现以下异常，而且 Flink 以非常友好的方式提示：

```
could not be determined automatically, due to type erasure. You can give type information hints by using the
returns(...) method on the result of the transformation call, or by letting your function implement the
'ResultTypeQueryable' interface.
```

就是因为 Java 编译器类型擦除的原因，所以 Flink 根本无法推断出来算子（例如 flatMap）要输出的类型是什么。所以，在 Flink 中使用 Lambda 表达式时，为了防止因类型擦除而出现运行时错误，需要指定 `TypeInformation` 或者 `TypeHint`。

创建 `TypeInformation`：

```
.returns(TypeInformation.of(String.class))
```

创建 `TypeHint`：

```
.returns(new TypeHint<String>() {})
```

3.1.2 应用程序实现

Flink 程序的基本构建块是 stream 和 transformation（流和转换）。从概念上讲，stream 是数据记录的流（可能永远不会结束），transformation 是一个运算，它接受一个或多个流作为输入，经过处理/计算后生成一个或多个输出流。

下面我们实现一个完整的、可工作的 Flink 流应用程序示例。

【示例】将有关人员的记录流作为输入，并从中筛选出未成年人信息。（Scala 实现）

Scala 代码：

```
package com.xueai8.ch03
```

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

```
/**
```

```
 * 在命令行执行'mvn clean package'，可将程序打包为 JAR 文件以执行。
```

```
 * 如果修改了主类的名字，请在 POM.xml 文件中修改相应的入口(查找'mainClass').
```

```
 */
```

```
* 将有关人员的记录流作为输入，并从中筛选出未成年人信息。
*/
object StreamingJobDemo1 {
  // 定义事件类
  case class Person(name:String, age:Integer)

  def main(args: Array[String]) {

    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 读取数据源，构造数据流
    val peoples = env.fromElements(
      Person("张三", 21),
      Person("李四", 16),
      Person("王老五", 35)
    )

    // 对数据流执行 filter 转换
    val adults = peoples.filter(_ .age>18)

    // 输出结果
    adults.print

    // 执行
    env.execute("Flink Streaming Job")
  }
}
```

执行代码，输出结果如下：

```
7> Person(张三,21)
1> Person(王老五,35)
```

Java 代码：

【示例 3-1】将有关人员的记录流作为输入，并从中筛选出未成年人信息。

(1) 在 IntelliJ IDEA 中创建一个 Flink 项目，使用 flink-quickstart-java 项目模板（Flink 项目创建过程，请参见 2.2 节）。

(2) 设置依赖。在 pom.xml 中添加如下依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-java</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
```

(3) 创建一个 POJO 类，用来表示流中的数据。代码如下。

```
package com.xueai8;

// POJO 类，表示人员信息实体
public class Person {
    public String name;        // 存储姓名
    public Integer age;        // 存储年龄

    // 空构造器
    public Person() {}

    // 构造器，初始化属性
    public Person(String name, Integer age) {
        this.name = name;
        this.age = age;
    };

    // 用于调试时输出信息
    public String toString() {
        return this.name.toString() + ": age " + this.age.toString();
    };
}
```

(4) 打开项目中的 StreamingJob 对象文件，编辑流处理代码如下。

```
package com.xueai8;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.api.common.functions.FilterFunction;

public class StreamingJobDemo1 {

    public static void main(String[] args) throws Exception {
        // 获得流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 读取数据源，构造 DataStream
        DataStream<Person> personDS = env.fromElements(
            new Person("张三", 21),
            new Person("李四", 16),
            new Person("王老五", 35)
        );

        // 执行转换运算（这里是过滤年龄不小于 18 岁的人）
        // 注意，这里我们使用了匿名函数
        DataStream<Person> adults = personDS.filter(new FilterFunction<Person>() {
            @Override
            public boolean filter(Person person) throws Exception {
                return person.age >= 18;
            }
        });
    }
}
```

```
});

// 将结果输出到控制台
adults.print();

// 触发程序开始执行
env.execute("stream demo");
}
```

(5) 执行以上程序，输出结果如下。

```
张三: age 21
王老五: age 35
```

注: Flink 将批处理程序作为流程序的一种特殊情况执行, 其中流是有界的(有限数量的元素)。DataSet 在内部被视为数据流。因此, 上述概念同样适用于批处理程序, 也适用于流程序, 只有少数例外:

- ☐ 批处理程序的容错不使用检查点。错误恢复是通过完全重放流来实现的, 这使得恢复的成本更高, 但是因为它避免了检查点, 所以使常规处理更轻量。
- ☐ DataSet API 中的有状态运算使用简化的 in-memory/out-of-core 数据结构, 而不是 key/value 索引。
- ☐ DataSet API 引入了特殊的同步(基于 superstep)迭代, 这只能在有界流上实现。

3.1.3 应用程序剖析

所有的 Flink 应用程序都以特定的步骤来工作。这些工作步骤如下图所示:



也就是说, 每个 Flink 程序都由相同的基本部分组成:

- ☐ 获取一个执行环境;
- ☐ 加载/创建初始数据;
- ☐ 指定对该数据的转换;
- ☐ 指定计算结果放在哪里;
- ☐ 触发程序执行。

1、获取一个执行环境

Flink 应用程序从其 main() 方法中生成一个或多个 Flink 作业 (job)。这些作业可以在本地 JVM (LocalEnvironment) 中执行, 也可以在具有多台机器的集群的远程设置中执行 (RemoteEnvironment)。对于每个程序, ExecutionEnvironment 提供了控制作业执行 (例如设置并行性或容错/检查点参数) 和与外部环境交互 (数据访问) 的方法。

每个 Flink 应用程序都需要一个执行环境(本例中为 env)。流应用程序需要的执行环境使用的是 StreamExecutionEnvironment。为了开始编写 Flink 程序, 我们首先需要获得一个现有的执行环境, 如果没有的话, 就需要先创建一个。根据我们的目的不同, Flink 支持以下几种方式:

- ☐ 获得一个已经存在的 Flink 环境
- ☐ 创建本地环境
- ☐ 创建远程环境

Flink 流程序的入口点是 StreamExecutionEnvironment 类的一个实例, 它定义了程序执行的上下文。

`StreamExecutionEnvironment` 是所有 Flink 程序的基础。可以通过使用以下这些静态方法获得一个 `StreamExecutionEnvironment` 的实例：

```
StreamExecutionEnvironment.getExecutionEnvironment()
StreamExecutionEnvironment.createLocalEnvironment()
StreamExecutionEnvironment.createRemoteEnvironment(String host, int port, String... jarFiles)
```

要获得执行环境，通常只需要调用 `getExecutionEnvironment()` 方法。这将根据我们的上下文选择正确的执行环境。如果正在 IDE 中的本地环境上执行，那么它将启动一个本地执行环境。如果是从程序中创建了一个 JAR 文件，并通过命令行调用它，那么 Flink 集群管理器将执行 `main()` 方法，`getExecutionEnvironment()` 将返回用于在集群上以分布式方式执行程序的执行环境。

在上面的示例程序中，我们使用以下语句来获得流程序的执行环境：

Scala 代码：

```
// 设置流执行环境
```

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
```

Java 代码：

```
// 获得流执行环境
```

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
```

`StreamExecutionEnvironment` 包含 `ExecutionConfig`，可使用它为运行时设置特定于作业的配置值。例如，我们如果要设置设置自动水印发送间隔，可以像下面这样在代码进行配置。

Scala 代码：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.getConfig.setAutoWatermarkInterval(long milliseconds)
```

Java 代码：

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.getConfig().setAutoWatermarkInterval(long milliseconds);
```

2、加载/创建初始数据

执行环境可以从多种数据源读取数据，包括文本文件、CSV 文件、Socket 套接字数据等，也可以使用自定义的数据输入格式。例如，要将文本文件读取为行序列，可以使用以下命令：

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
DataStream<String> text = env.readTextFile("file:///path/to/file");
```

数据被逐行读取到内存后，Flink 会将它们组织到 `DataStream` 中，这是 Flink 中用来表示流数据的特殊类。

在上一节的示例程序中，我们使用 `fromElements()` 方法读取集合数据，并将读取的数据存储为 `DataStream` 类型：

Scala 代码：

```
// 读取数据源，构造数据流
```

```
val personDS = env.fromElements(
    Person("张三", 21),
    Person("李四", 16),
    Person("王老五", 35)
)
```

Java 代码：

```
// 读取数据源，构造 DataStream
```

```
DataStream<Person> personDS = env.fromElements(
```

```
new Person("张三", 21),
new Person("李四", 16),
new Person("王老五", 35));
```

3、对数据进行转换

每个 Flink 程序都对分布式数据集合执行转换。Flink 的 `DataStream` API 提供了多种数据转换功能，包括过滤、映射、连接、分组和聚合。例如，下面是一个 `map` 转换代码，通过将原始集合中的每个字符串转换为整数来创建一个新的 `DataStream`：

```
DataStream<String> input = env.fromElements("12","3","25","5","32","6");

DataStream<Integer> parsed = input.map(new MapFunction<String, Integer>() {
    @Override
    public Integer map(String value) {
        return Integer.parseInt(value);
    }
});
```

在上节的示例程序中，我们使用了 `filter` 过滤转换，将原始数据集转换为只包含成年人信息的新 `DataStream` 流：

Scala 代码：

```
// 对数据流执行 filter 转换
val adults = personDS.filter(_ .age>18)
```

Java 代码：

```
// 对数据流执行 filter 转换
DataStream<Person> adults = flintstones.filter(
    new FilterFunction<Person>() {
        @Override
        public boolean filter(Person person) throws Exception {
            return person.age >= 18;
        }
    });
```

这里不必了解每个转换的具体含义，后面我们会详细介绍它们。需要强调的是，Flink 中的转换是惰性的，在调用 `sink` 操作之前不会真正执行。

4、指定计算结果放在哪里

一旦有了包含最终结果的 `DataStream`，就可以通过创建接收器（`sink`）将其写入外部系统。在上面的示例程序中，我们将计算结果打印输出到屏幕上。

Scala 代码：

```
// 输出结果
adults.print
```

Java 代码：

```
// 输出结果
adults.print();
```

Flink 中的接收器（`sink`）操作触发流的执行，以生成程序所需的结果，例如将结果保存到文件系统或将其打印到标准输出。上面的示例使用 `adults.print()` 将结果打印到任务管理器日志中（在 IDE 中运行时，任务管理器日志将显示在 IDE 的控制台中）。这将对流的每个元素调用其 `toString()` 方法。

5、触发程序执行

一旦写好了程序处理逻辑，就需要通过调用 `StreamExecutionEnvironment` 上的 `execute()` 来触发程序执行。所有的 Flink 程序都是延迟执行的：当程序的主方法执行时，数据加载和转换不会直接发生，而是创建每个运算并添加到程序的执行计划中。当执行环境上的 `execute()` 调用显式触发执行时，这些操作才实际上被执行。程序是在本地执行还是提交到集群中执行取决于 `ExecutionEnvironment` 的类型。延迟计算可以让我们构建复杂的程序，然后 Flink 将其作为一个整体计划的单元执行。

在上一节的示例程序中，我们使用如下代码来触发流处理程序的执行。

Scala 代码：

```
// 触发流程序执行
```

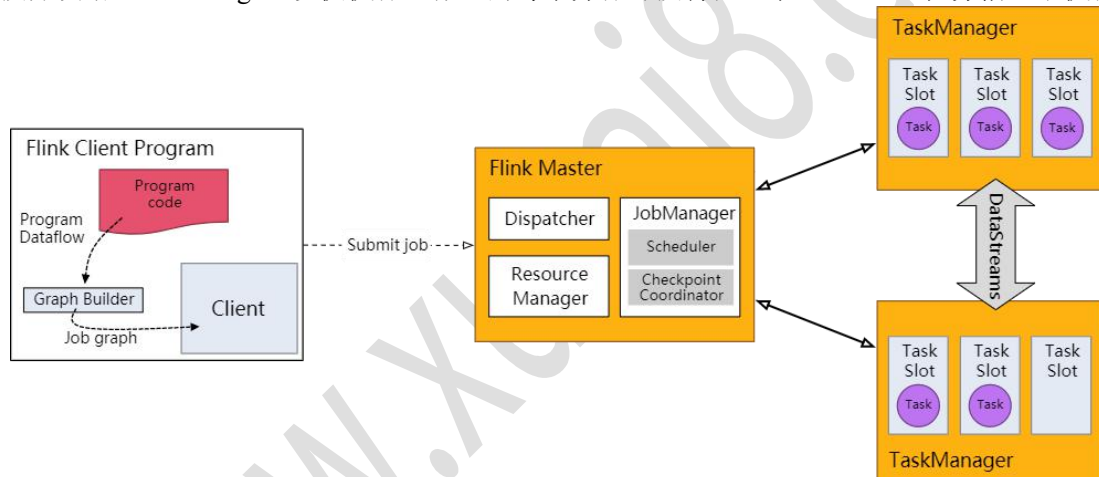
```
env.execute("Flink Streaming Job") // 参数是程序名称，会显示在 Web UI 界面上
```

Java 代码：

```
// 触发流程序执行
```

```
env.execute("Flink Streaming Job"); // 参数是程序名称，会显示在 Web UI 界面上
```

在应用程序中进行的 `DataStream` API 调用将构建一个附加到 `StreamExecutionEnvironment` 的作业图（Job Graph）。调用 `env.execute()` 时，此图被打包并发送到 Flink Master，该 Master 并行化作业并将其片段分发给 `TaskManagers` 以供执行。作业在每个并行片段将在一个 `task slot`（任务槽）中执行。



这个分布式运行时要求 Flink 应用程序是可序列化的。它还要求集群中的每个节点都可以使用所有依赖项。

`StreamExecutionEnvironment` 上的 `execute()` 方法将等待作业完成，然后返回一个 `JobExecutionResult`，其中包含执行时间和累加器结果。注意，如果不调用 `execute()`，应用程序将不会运行。

如果不想等待作业完成，可以通过调用 `StreamExecutionEnvironment` 上的 `executeAsync()` 来触发异步作业执行。它将返回一个 `JobClient`，可以使用它与刚才提交的作业进行通信。例如，下面是如何通过使用 `executeAsync()` 实现 `execute()` 的语义。

Scala 代码：

```
val jobClient = env.executeAsync
```

```
val jobExecutionResult = jobClient.getJobExecutionResult(userClassLoader).get
```

Java 代码：

```
final JobClient jobClient = env.executeAsync();
```

```
final JobExecutionResult jobExecutionResult = jobClient.getJobExecutionResult(userClassLoader).get();
```

3.2 数据源

数据源是 Flink 程序希望从其中获取数据的地方。可以从多个源创建数据集，如 Apache Kafka、CSV、文件或几乎任何其他数据源。Flink 附带了许多预先实现的数据源函数。它还支持编写自定义数据源函数。

首先，让我们理解其内置的数据源函数。可以从 `StreamExecutionEnvironment` 访问预定义的数据源。

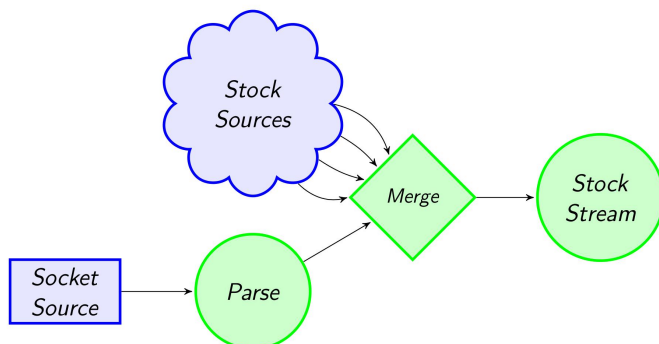
注：在实际应用程序中，最常用的数据源是那些支持低延迟、高吞吐量并行读取以及重放和 `rewind` 的数据源，这是实现高性能和容错的先决条件，比如 Apache Kafka、Kinesis 和各种文件系统。

3.2.1 基于 Socket 的数据源

`DataStream` API 支持从 `Socket` 套接字读取数据。我们只需要指定要从其中读取数据的主机和端口号即可。读取 `Socket` 套接字的数据源函数定义如下：

- ❑ `socketTextStream(hostName, port)`
- ❑ `socketTextStream(hostName, port, delimiter)`：可指定分隔符。
- ❑ `socketTextStream(hostName, port, delimiter, maxRetry)`：还可以指定 API 应该尝试获取数据的最大次数。

Socket 数据源如下图所示：



【示例】Socket 数据源：流应用程序示例，它接收来自 web 套接字的单词。

Scala 实现：

```
package com.xueai8.scala.ch03

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

object SocketSourceDemo {

    val HOST = "192.168.190.133"    // host 主机
    val PORT = 9999                // 端口号

    def main(args: Array[String]): Unit = {
        // 设置流执行环境
        val env = StreamExecutionEnvironment.getExecutionEnvironment
```

```
// 连接 Socket 数据源
val input = env.socketTextStream(HOST, PORT)

// 流数据转换，并打印结果
input
  .map(_.toLowerCase)
  .flatMap(_.\W+)
  .print

// 触发流程序执行
env.execute("Flink Socket Source")
}
```

Java 实现：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

public class SocketSourceDemo {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 源数据流
        String host = "xueai8"; // host 主机或 IP 地址
        int port = 9999; // 端口号
        DataStream<String> input = env.socketTextStream(host, port);

        // 对 DataStream 进行转换，向 map 和 flatMap 传入匿名内容类
        input.map(new MapFunction<String, String>() {
            @Override
            public String map(String s) throws Exception {
                return s.toLowerCase(); // 先转小写
            }
        }).flatMap(new FlatMapFunction<String, String>() {
            @Override
            public void flatMap(String s, Collector<String> collector) throws Exception {
                for(String word : s.split("\\W+")){ // 再分词
                    collector.collect(word); // 向下游发送
                }
            }
        }).print();
    }
}
```

```
// 执行程序
env.execute("Flink Socket Source");
}
}
```

执行过程

1) 打开一个终端窗口，执行如下的命令，启动一个 netcat 服务器，运行在 9999 端口：

```
$ nc -lk 9999
```

2) 运行程序

3) 在 netcat 运行窗口，输入以下内容，并回车：

```
Good good study
```

```
day day up
```

4) 在程序执行窗口，可以看到如下计算输出：

```
5> good
```

```
5> good
```

```
5> study
```

```
6> day
```

```
6> day
```

```
6> up
```

在上面的 Java 示例代码中，我们可以将 map 的逻辑代码合并到 flatMap 函数中，以精减代码。修改后的代码如下所示：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

public class SocketSourceDemo {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 源数据流
        String host = "xueai8"; // host 主机或 IP 地址
        int port = 9999; // 端口号
        DataStream<String> input = env.socketTextStream(host, port);

        // 对 DataStream 进行转换，向 map 和 flatMap 传入匿名内容类
        input.flatMap(new FlatMapFunction<String, String>() {
            @Override
            public void flatMap(String s, Collector<String> collector) throws Exception {
                String line = s.toLowerCase(); // 先转小写
                for(String word : line.split("\\W+")){ // 再分词

```

```
        collector.collect(word);           // 向下游发送
    }
}
}).print();

// 执行程序
env.execute("Flink Socket Source");
}
}
```

在前面的 Java 代码示例中，我们向 `map` 和 `flatMap` 传入的都是匿名内部类。我们也可以代之以 Java 8 引入的 Lambda 函数，以简化代码的编写。使用 Lambda 函数重构的代码如下所示：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

public class SocketSourceDemo {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 源数据流
        String host = "xueai8";           // host 主机或 IP 地址
        int port = 9999;                   // 端口号
        DataStream<String> input = env.socketTextStream(host, port);

        // 对 DataStream 进行转换，向 map 和 flatMap 传入 Lambda 函数
        input.flatMap((FlatMapFunction<String, String>) (s, collector) -> {
            String line = s.toLowerCase(); // 先转小写
            for (String word : line.split("\\W+")) { // 再分词
                collector.collect(word); // 向下游发送
            }
        }).returns(Types.STRING) // 因为类型擦除的原因，所以必须提供返回类型
        .print();

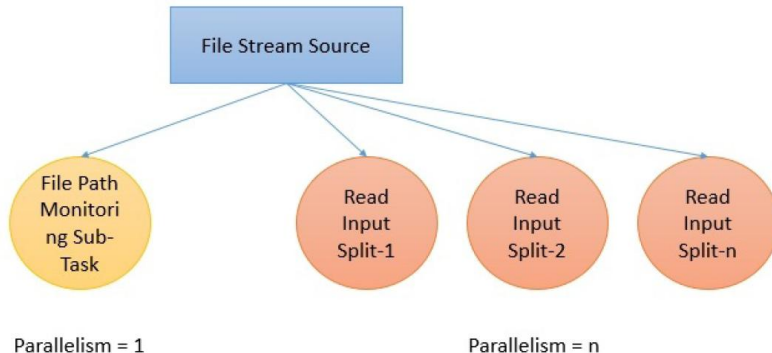
        // 执行程序
        env.execute("Flink Socket Source");
    }
}
```

3.2.2 基于文件的数据源

还可以选择使用基于文件的源函数从文件源中传输数据。从文件源读取数据的源函数定义有：

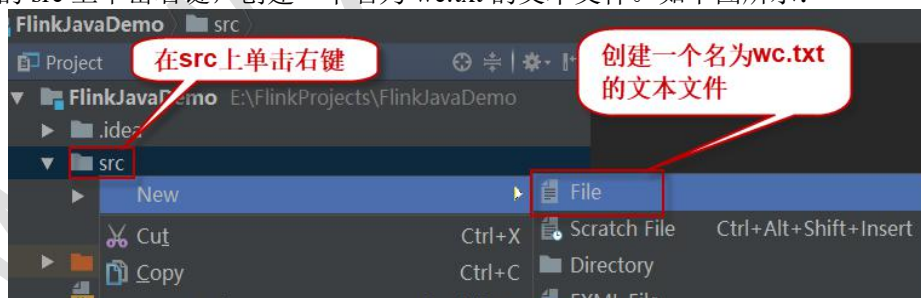
小白学院 <http://www.xueai8.com>

- ❑ `readTextFile(String path)`: 逐行读取路径指定的文本文件，即符合 `TextInputFormat` 规范的文本文件，并以字符串形式返回。
- ❑ `readFile(FileInputFormat inputFormat, String path)`: 根据指定的文件输入格式读取(一次)文件。
- ❑ `readFile(fileInputFormat, path, watchType, interval, pathFilter)`: 这是前两个方法在内部调用的方法。它根据给定的 `fileInputFormat` 读取路径中的文件。根据所提供的 `watchType`，此源可以定期(每隔 `interval` 毫秒)监视新数据的路径 (`FileProcessingMode.PROCESS_CONTINUOUSLY`)，或处理一次当前路径中的数据并退出 (`FileProcessingMode.PROCESS_ONCE`)。使用 `pathFilter` 进一步排除正在处理的文件。在底层，Flink 将文件读取过程分成两个子任务，即目录监视和数据读取。每个子任务都由一个单独的实体实现。监视文件路径的子任务是由单个非并行(并行度 = 1)任务实现的，而文件读取则由多个并行运行的任务执行，并行度等于作业并行度。单个监视任务的作用是扫描目录(定期或仅扫描一次，这取决于 `watchType`)，查找要处理的文件，将它们划分为分段，并将这些分段分配给下游的读取器，如下图所示。读取器将读取实际数据。每个分割只由一个读取器读取，而一个读取器可以逐个读取多个分割。



【示例】读取文件，并统计文件内的单词数量。

- 1、在 IDEA 中创建 Flink 项目；
- 2、在项目的 `src` 上单击右键，创建一个名为 `wc.txt` 的文本文件。如下图所示：



- 3、编辑 `wc.txt`，输入以下内容并保存：

```
good good study
day day up
```

- 4、流处理代码实现。

Scala 代码：

```
package com.xueai8.scala.ch03

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

object FileSourceDemo {
```

```
def main(args: Array[String]): Unit = {  
    // 设置流执行环境  
    val env = StreamExecutionEnvironment.getExecutionEnvironment  
  
    // 加载文件数据源, 构造 DataStream  
    val textPath = "input/wc.txt"  
    val text = env.readTextFile(textPath)  
  
    // 对 DataStream 执行转换操作  
    text  
        .flatMap { _.toLowerCase.split("\\W+").filter( _.nonEmpty ) }  
        .map { (_, 1) }  
        .print()  
  
    // 触发程序执行  
    env.execute("Simple Flink File Source Demo")  
}
```

Java 代码:

```
package com.xueai8.java.ch03;  
  
import org.apache.flink.api.common.functions.FlatMapFunction;  
import org.apache.flink.api.java.tuple.Tuple2;  
import org.apache.flink.streaming.api.datastream.DataStreamSource;  
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
import org.apache.flink.util.Collector;  
  
/**  
 * 文件数据源  
 */  
public class FileSourceDemo {  
    public static void main(String[] args) throws Exception {  
        // 设置流执行环境  
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  
  
        // 加载文件数据源, 构造 DataStream  
        String textPath = "input/wc.txt";  
        DataStreamSource<String> text = env.readTextFile(textPath);  
  
        // 数据流转换  
        text.map(String::toLowerCase)           // 转小写  
            .flatMap(new Splitter())           // flatMap 转换  
            .print();  
  
        // 触发程序执行  
        env.execute("Simple Flink File Source Demo");  
    }  
  
    // 实现 FlatMapFunction 接口的函数
```

```
public static class Splitter implements FlatMapFunction<String, Tuple2<String, Integer>> {  
    @Override  
    public void flatMap(String sentence, Collector<Tuple2<String, Integer>> out) throws Exception {  
        for (String word: sentence.split(" ")) {           // 分词  
            out.collect(new Tuple2<>(word, 1));           // 构造(word,1)发送给下游算子  
        }  
    }  
}
```

执行以上程序，输出结果如下：

```
2> (day,1)  
6> (good,1)  
2> (day,1)  
6> (good,1)  
2> (up,1)  
6> (study,1)
```

3.2.3 基于集合的数据源

也可以直接在内存中将一个数据集合读取为 `DataStream`。Flink 提供了如下几个方法。

- ❑ `fromCollection(Seq)`: 从 Java `Java.util. collection` 创建一个数据流。集合中的所有元素必须具有相同的类型。
- ❑ `fromCollection(Iterator)`: 从迭代器创建数据流。该类指定迭代器返回的元素的数据类型。
- ❑ `fromElements(elements: _*)`: 根据给定的对象序列创建数据流。所有对象必须具有相同的类型。
- ❑ `fromParallelCollection(SplittableIterator)`: 并行地从迭代器创建数据流。该类指定迭代器返回的元素的数据类型。
- ❑ `generateSequence(from, to)`: 并行地生成给定区间内的数字序列。

注：目前，集合数据源要求数据类型和迭代器实现 `Serializable`。此外，收集数据源不能并行执行(并行度=1)。

【示例】使用集合数据源。

- 1、在 IntelliJ IDEA 中创建一个 Flink 项目，使用 `flink-quickstart-scala/flink-quickstart-java` 项目模板。
- 2、打开项目中的 `StreamingJob` 对象文件，编辑流处理代码。

Scala 代码：

```
package com.xueai8.scala.ch03  
  
import org.apache.flink.streaming.api.scala._  
  
object CollectionSourceDemo {  
  
    // 定义事件类  
    case class Person(name:String, age:Integer)  
  
    def main(args: Array[String]): Unit = {  
        // 设置流执行环境  
        val env = StreamExecutionEnvironment.getExecutionEnvironment
```

```
// 首先从环境中获取一些数据, 比如:
val peoples = List(Person("张三", 21), Person("李四", 16), Person("王老五", 35))
val personStream = env.fromCollection(peoples)

// 对 DataStream 执行转换操作, 并输出计算结果
personStream.filter(_age > 18).print()

// 触发流程程序执行
env.execute("Flink Collection Source Demo")
}
```

Java 代码:

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.FilterFunction;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import java.util.ArrayList;
import java.util.List;

public class CollectionSourceDemo {

    // POJO 类
    public static class Person {
        public String name;
        public Integer age;

        public Person() {}

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

        public String toString() {
            return this.name.toString() + ": age " + this.age.toString();
        }
    }

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 读取数据源, 构造 DataStream
        List<Person> peoples = new ArrayList<>();
        peoples.add(new Person("张三", 21));
        peoples.add(new Person("李四", 16));
        peoples.add(new Person("王老五", 35));

        DataStream<Person> personStream = env.fromCollection(peoples);
    }
}
```

```
// 对 DataStream 执行 filter 转换操作
DataStream<Person> adults = personStream.filter(new FilterFunction<Person>() {
    @Override
    public boolean filter(Person person) throws Exception {
        return person.age >= 18;
    }
});

// 输出流计算结果
adults.print();

// 触发流程序执行
env.execute("Flink File Source");
}
```

3、执行以上程序，输出结果如下：

```
张三: age 21
王老五: age 35
```

3.2.4 自定义数据源

除了内置数据源，我们还可以编写自己的定制数据源。对于非并行数据源，实现 `SourceFunction`；对于并行数据源，实现 `ParallelSourceFunction` 接口或扩展（继承）自 `RichParallelSourceFunction`。

SourceFunction 接口分析

`SourceFunction` 是 Flink 中所有流数据源的基本接口。`SourceFunction` 接口的定义如下：

```
package org.apache.flink.streaming.api.functions.source;

import java.io.Serializable;
import org.apache.flink.annotation.Public;
import org.apache.flink.annotation.PublicEvolving;
import org.apache.flink.api.common.functions.Function;
import org.apache.flink.streaming.api.watermark.Watermark;

@Public
public interface SourceFunction<T> extends Function, Serializable {
    void run(SourceFunction.SourceContext<T> var1) throws Exception;

    void cancel();

    @Public
    public interface SourceContext<T> {
        void collect(T var1);

        @PublicEvolving
        void collectWithTimestamp(T var1, long var2);

        @PublicEvolving
```

```
void emitWatermark(Watermark var1);

@PublicEvolving
void markAsTemporarilyIdle();

Object getCheckpointLock();

void close();
}
```

分析:

- ❑ SourceFunction 定义了 run 和 cancel 两个方法和 SourceContext 内部接口。
- ❑ run(SourceContext): 实现数据获取逻辑, 并可以通过传入的参数 ctx 进行向下游节点的数据转发。
- ❑ cancel(): 用来取消数据源, 一般在 run 方法中, 会存在一个循环来持续产生数据, cancel 方法则可以使该循环终止。
- ❑ SourceContext: source 函数用于发出元素和可能的 watermark 的接口, 返回 source 生成的元素的类型。

run 方法

run 方法为数据源向下游发送数据的主要逻辑。编写套路为:

- ❑ 不断调用循环发送数据。
- ❑ 使用一个状态变量控制循环的执行。当 cancel 方法执行后必须能够跳出循环, 停止发送数据。
- ❑ 使用 SourceContext 的 collect 等方法将元素发送至下游。
- ❑ 如果使用 Checkpoint, 在 SourceContext collect 数据的时候必须加锁。防止 checkpoint 操作和发送数据操作同时进行。

cancel 方法:

cancel 方法在数据源停止的时候调用。cancel 方法必须能够控制 run 方法中的循环, 停止循环的运行。并做一些状态清理操作。

SourceContext 类

SourceContext 在 SourceFunction 中使用, 用于向下游发送数据, 或者是发送水印 (watermark)。

SourceContext 的方法包括:

- ❑ collect: 向下游发送数据。有如下三种情况:
 - 如果使用 ProcessingTime, 该元素不携带 timestamp。
 - 如果使用 IngestionTime, 元素使用系统当前时间作为 timestamp。
 - 如果使用 EventTime, 元素不携带 timestamp。需要在数据流后续为元素指定 timestamp (assignTimestampAndWatermark)。
- ❑ collectWithTimestamp: 向下游发送带有 timestamp 的数据。和 collect 方法一样也有如下三种情况:
 - 如果使用 ProcessingTime, timestamp 会被忽略。
 - 如果使用 IngestionTime, 使用系统时间覆盖 timestamp。
 - 如果使用 EventTime, 使用指定的 timestamp。

- ❑ **emitWatermark:** 向下游发送 watermark。watermark 也包含一个 timestamp。向下游发送 watermark 意味着所有在 watermark 的 timestamp 之前的数据已经到齐。如果在 watermark 之后，收到了 timestamp 比该 watermark 的 timestamp 小的元素，该元素会被认为迟到，将会被系统忽略，或者进入到旁路输出（side output）。
- ❑ **markAsTemporarilyIdle:** 标记此数据源暂时闲置。该数据源暂时不会发送任何数据和 watermark。仅对 IngestionTime 和 EventTime 生效。下游任务前移 watermark 的时候将不会再等待被标记为闲置的数据源的 watermark。

CheckpointedFunction

如果数据源需要保存状态，那么就需要实现 CheckpointedFunction 中的相关方法。

CheckpointedFunction 包含如下方法：

- ❑ **snapshotState:** 保存 checkpoint 的时候调用。需要在此方法中编写状态保存逻辑
- ❑ **initializeState:** 在数据源创建或者是从 checkpoint 恢复的时候调用。此方法包含数据源的状态恢复逻辑。

可以使用 StreamExecutionEnvironment.addSource(sourceFunction)将指定数据源附加到程序中。

- ❑ **addSource:** 附加一个新的数据源函数。例如，要从 Apache Kafka 中读取数据，可以使用 addSource(new FlinkKafkaConsumer<>(...))。

【示例】（简单版本）使用自定义数据源，模拟信用卡交易流数据生成器。

- 1、在 IntelliJ IDEA 中创建一个 Flink 项目，使用 flink-quickstart-scala/flink-quickstart-java 项目模板。
- 2、设置依赖。在 pom.xml 中添加如下依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-java</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
```


</dependency>

3、创建 POJO 类，表示信用卡交易数据结构：

```
package com.xueai8.fraud.entity;

import java.util.Objects;

/**
 * 实体类，代表信用卡交易数据
 */
public class Transaction implements Serializable {
    public long accountId;        // 交易账户
    public long timestamp;        // 交易时间
    public double amount;         // 交易金额

    public Transaction() {}

    public Transaction(long accountId, long timestamp, double amount) {
        this.accountId = accountId;
        this.timestamp = timestamp;
        this.amount = amount;
    }

    public long getAccountId() {
        return accountId;
    }

    public void setAccountId(long accountId) {
        this.accountId = accountId;
    }

    public long getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(long timestamp) {
        this.timestamp = timestamp;
    }

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
    }
}
```

```

    } else if (o == null || getClass() != o.getClass()) {
        return false;
    }
    Transaction that = (Transaction) o;
    return accountId == that.accountId &&
        timestamp == that.timestamp &&
        Double.compare(that.amount, amount) == 0;
}

@Override
public int hashCode() {
    return Objects.hash(accountId, timestamp, amount);
}

@Override
public String toString() {
    return "Transaction{" +
        "accountId=" + accountId +
        ", timestamp=" + timestamp +
        ", amount=" + amount +
        '}';
}
}

```

4、创建自定义的数据源类，继承自 `SourceFunction`。实现代码如下所示：

```

package com.xueai8.fraud.source;

import com.xueai8.fraud.entity.Transaction;
import org.apache.flink.streaming.api.functions.source.SourceFunction;

import java.sql.Timestamp;
import java.util.Arrays;
import java.util.List;

/**
 * 自定义数据源，继承自 SourceFunction
 */
public class MyTransactionSource implements SourceFunction<Transaction> {
    private static final long serialVersionUID = 1L;

    private static final Timestamp INITIAL_TIMESTAMP = Timestamp.valueOf("2020-01-01 00:00:00");
    private static final long SIX_MINUTES = 6 * 60 * 1000;

    private final boolean bounded;           // 标志变量，指示生成流数据还是批数据

    private int index = 0;                   // 交易记录的索引
    private long timestamp;                  // 交易发生的时间戳

    private volatile boolean isRunning = true;
    private List<Transaction> data = null;
}

```

```
public MyTransactionSource(boolean bounded){
    this.bounded = bounded;
    this.timestamp = INITIAL_TIMESTAMP.getTime();

    // 事先存储的信用卡交易数据，在实际中来自于外部数据源系统（如 Kafka）
    data = Arrays.asList(
        new Transaction(1, 0L, 188.23),
        new Transaction(2, 0L, 374.79),
        new Transaction(3, 0L, 112.15),
        new Transaction(4, 0L, 478.75),
        new Transaction(5, 0L, 208.85),
        new Transaction(1, 0L, 379.64),
        new Transaction(2, 0L, 351.44),
        new Transaction(3, 0L, 320.75),
        new Transaction(4, 0L, 259.42),
        new Transaction(5, 0L, 273.44),
        new Transaction(1, 0L, 267.25),
        new Transaction(2, 0L, 397.15),
        new Transaction(3, 0L, 0.219),
        new Transaction(4, 0L, 231.94),
        new Transaction(5, 0L, 384.73),
        new Transaction(1, 0L, 419.62),
        new Transaction(2, 0L, 412.91),
        new Transaction(3, 0L, 0.77),
        new Transaction(4, 0L, 22.10),
        new Transaction(5, 0L, 377.54),
        new Transaction(1, 0L, 375.44),
        new Transaction(2, 0L, 230.18),
        new Transaction(3, 0L, 0.80),
        new Transaction(4, 0L, 350.89),
        new Transaction(5, 0L, 127.55),
        new Transaction(1, 0L, 483.91),
        new Transaction(2, 0L, 228.22),
        new Transaction(3, 0L, 871.15),
        new Transaction(4, 0L, 64.19),
        new Transaction(5, 0L, 79.43),
        new Transaction(1, 0L, 56.12),
        new Transaction(2, 0L, 256.48),
        new Transaction(3, 0L, 148.16),
        new Transaction(4, 0L, 199.95),
        new Transaction(5, 0L, 252.37),
        new Transaction(1, 0L, 274.73),
        new Transaction(2, 0L, 473.54),
        new Transaction(3, 0L, 119.92),
        new Transaction(4, 0L, 323.59),
        new Transaction(5, 0L, 353.16),
        new Transaction(1, 0L, 211.90),
        new Transaction(2, 0L, 280.93),
        new Transaction(3, 0L, 347.89),
        new Transaction(4, 0L, 459.86),
```

```
        new Transaction(5, 0L, 82.31),
        new Transaction(1, 0L, 373.26),
        new Transaction(2, 0L, 479.83),
        new Transaction(3, 0L, 454.25),
        new Transaction(4, 0L, 83.64),
        new Transaction(5, 0L, 292.44)
    );
}

@Override
public void run(SourceContext<Transaction> sourceContext) throws Exception {
    while(this.isRunning && this.hasNext()) {
        sourceContext.collect(this.next());
    }
}

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

private boolean hasNext() {
    // 如果还有数据
    if (index < data.size()) {
        return true;
    }
    // 如果是用于生成批数据
    else if(bounded){
        return false;
    }
    // 如果是用于生成流数据，从头循环
    else {
        index = 0;
        return true;
    }
}

// 生成下一个交易数据，交易时间相隔 6 分钟
private Transaction next() {
    try {
        Thread.sleep(100);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    Transaction transaction = data.get(index++);
    transaction.setTimestamp(timestamp);
    timestamp += SIX_MINUTES;
    return transaction;
}
```

```
}
```

5、创建一个测试类（带有 main 方法的主程序），使用 addSource 方法添加自定义数据源。

Scala 代码：

```
package com.xueai8.java.ch03

import com.xueai8.ch03.source.MyTransactionSource
import org.apache.flink.streaming.api.scala._

/**
 * 自定义数据源的测试类
 */
object CustomSourceDemo {

  def main(args: Array[String]): Unit = {
    // 获取流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 添加自定义数据源
    val transactions = env.addSource(new MyTransactionSource(false)).name("transactions")

    // 输出交易记录
    transactions.print

    // 触发程序执行
    env.execute("Flink Custom Source")
  }
}
```

Java 代码：

```
package com.xueai8.java.ch03;

import com.xueai8.ch03.entity.Transaction;
import com.xueai8.ch03.source.MyTransactionSource;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * 自定义数据源测试类
 */
public class CustomSourceDemo {

  public static void main(String[] args) throws Exception {
    // 设置流执行环境
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

    // 设置自定义数据源。参数 false 指定创建的是流数据源
    DataStream<Transaction> transactions =
        env.addSource(new MyTransactionSource(false)).name("transactions");

    // 输出查看
  }
}
```

```

        transactions.print();

        // 执行程序
        env.execute("Transaction Stream");
    }
}

```

6、执行以上程序，查看控制台输出结果如下（部分结果）：

```

1> Transaction{accountId=1, timestamp=1577808000000, amount=188.23}
2> Transaction{accountId=2, timestamp=1577808360000, amount=374.79}
3> Transaction{accountId=3, timestamp=1577808720000, amount=112.15}
4> Transaction{accountId=4, timestamp=1577809080000, amount=478.75}
5> Transaction{accountId=5, timestamp=1577809440000, amount=208.85}
6> Transaction{accountId=1, timestamp=1577809800000, amount=379.64}
7> Transaction{accountId=2, timestamp=1577810160000, amount=351.44}
8> Transaction{accountId=3, timestamp=1577810520000, amount=320.75}
1> Transaction{accountId=4, timestamp=1577810880000, amount=259.42}
2> Transaction{accountId=5, timestamp=1577811240000, amount=273.44}
3> Transaction{accountId=1, timestamp=1577811600000, amount=267.25}
4> Transaction{accountId=2, timestamp=1577811960000, amount=397.15}
5> Transaction{accountId=3, timestamp=1577812320000, amount=0.219}
6> Transaction{accountId=4, timestamp=1577812680000, amount=231.94}
7> Transaction{accountId=5, timestamp=1577813040000, amount=384.73}
8> Transaction{accountId=1, timestamp=1577813400000, amount=419.62}
.....

```

可以对上面的代码加以修改，创建更加通用的自定义数据源类。

【示例】（上一例的复杂版本）使用自定义数据源，模拟信用卡交易流数据生成器。

- 1、在 IntelliJ IDEA 中创建一个 Flink 项目，使用 `flink-quickstart-java` 项目模板。
- 2、设置依赖。在 `pom.xml` 中添加如下依赖：

```

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

```

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

3、创建 POJO 类，表示信用卡交易数据结构：

```
package com.xueai8.fraud.entity;

import java.util.Objects;

/**
 * 实体类，代表信用卡交易
 */
public class Transaction {
    public long accountId;        // 交易账户
    public long timestamp;        // 交易时间
    public double amount;        // 交易金额

    public Transaction() {}

    public Transaction(long accountId, long timestamp, double amount) {
        this.accountId = accountId;
        this.timestamp = timestamp;
        this.amount = amount;
    }

    public long getAccountId() {
        return accountId;
    }

    public void setAccountId(long accountId) {
        this.accountId = accountId;
    }

    public long getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(long timestamp) {
        this.timestamp = timestamp;
    }

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }
}
```



```
@Override
public boolean equals(Object o) {
    if (this == o) {
        return true;
    } else if (o == null || getClass() != o.getClass()) {
        return false;
    }
    Transaction that = (Transaction) o;
    return accountId == that.accountId &&
        timestamp == that.timestamp &&
        Double.compare(that.amount, amount) == 0;
}

@Override
public int hashCode() {
    return Objects.hash(accountId, timestamp, amount);
}

@Override
public String toString() {
    return "Transaction{" +
        "accountId=" + accountId +
        ", timestamp=" + timestamp +
        ", amount=" + amount +
        '}';
}
```

3、定义一个迭代器类，实现 `Iterator` 接口，封装了对 `Transaction` 信用卡交易数据列表的迭代方法。这是一个通用的数据迭代器类，即可以迭代生成批数据，也可以循环迭代生成流数据，由其中的标志变量 `bounded` 来控制。代码如下：

```
package com.xueai8.fraud.source;

import com.xueai8.fraud.entity.Transaction;

import java.io.Serializable;
import java.sql.Timestamp;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;

/**
 * 模拟信用卡交易的交易数据源的迭代器
 */
public class TransactionIterator implements Iterator<Transaction>, Serializable {
    private static final long serialVersionUID = 1L;

    private static final Timestamp INITIAL_TIMESTAMP = Timestamp.valueOf("2020-01-01 00:00:00");
    private static final long SIX_MINUTES = 6 * 60 * 1000;
```

```
private final boolean bounded;           // 标志变量，指示生成流数据还是批数据

private int index = 0;                   // 交易记录的索引
private long timestamp;                   // 交易发生的时间戳

static TransactionIterator bounded() {
    return new TransactionIterator(true);
}

static TransactionIterator unbounded() {
    return new TransactionIterator(false);
}

private TransactionIterator(boolean bounded) {
    this.bounded = bounded;
    this.timestamp = INITIAL_TIMESTAMP.getTime();
}

@Override
public boolean hasNext() {
    // 如果还有数据
    if (index < data.size()) {
        return true;
    }
    // 如果是用于生成流数据，从头循环
    else if (!bounded) {
        index = 0;
        return true;
    }
    // 如果是用于生成批数据
    else {
        return false;
    }
}

// 生成下一个交易数据，交易时间相隔 6 分钟
@Override
public Transaction next() {
    Transaction transaction = data.get(index++);
    transaction.setTimestamp(timestamp);
    timestamp += SIX_MINUTES;
    return transaction;
}

// 事先存储的信用卡交易数据，在实际中来自于外部数据源系统（如 Kafka）
private static List<Transaction> data = Arrays.asList(
    new Transaction(1, 0L, 188.23),
    new Transaction(2, 0L, 374.79),
    new Transaction(3, 0L, 112.15),
    new Transaction(4, 0L, 478.75),
```

```
new Transaction(5, 0L, 208.85),
new Transaction(1, 0L, 379.64),
new Transaction(2, 0L, 351.44),
new Transaction(3, 0L, 320.75),
new Transaction(4, 0L, 259.42),
new Transaction(5, 0L, 273.44),
new Transaction(1, 0L, 267.25),
new Transaction(2, 0L, 397.15),
new Transaction(3, 0L, 0.219),
new Transaction(4, 0L, 231.94),
new Transaction(5, 0L, 384.73),
new Transaction(1, 0L, 419.62),
new Transaction(2, 0L, 412.91),
new Transaction(3, 0L, 0.77),
new Transaction(4, 0L, 22.10),
new Transaction(5, 0L, 377.54),
new Transaction(1, 0L, 375.44),
new Transaction(2, 0L, 230.18),
new Transaction(3, 0L, 0.80),
new Transaction(4, 0L, 350.89),
new Transaction(5, 0L, 127.55),
new Transaction(1, 0L, 483.91),
new Transaction(2, 0L, 228.22),
new Transaction(3, 0L, 871.15),
new Transaction(4, 0L, 64.19),
new Transaction(5, 0L, 79.43),
new Transaction(1, 0L, 56.12),
new Transaction(2, 0L, 256.48),
new Transaction(3, 0L, 148.16),
new Transaction(4, 0L, 199.95),
new Transaction(5, 0L, 252.37),
new Transaction(1, 0L, 274.73),
new Transaction(2, 0L, 473.54),
new Transaction(3, 0L, 119.92),
new Transaction(4, 0L, 323.59),
new Transaction(5, 0L, 353.16),
new Transaction(1, 0L, 211.90),
new Transaction(2, 0L, 280.93),
new Transaction(3, 0L, 347.89),
new Transaction(4, 0L, 459.86),
new Transaction(5, 0L, 82.31),
new Transaction(1, 0L, 373.26),
new Transaction(2, 0L, 479.83),
new Transaction(3, 0L, 454.25),
new Transaction(4, 0L, 83.64),
new Transaction(5, 0L, 292.44)
);
}
```

4、创建自定义的数据源类，继承自 `FromIteratorFunction` 类（这是 Flink 自带的一个 `SourceFunction` 的子类），并封装了一个数据生成器 `RateLimitedIterator`。实现代码如下所示：

小白学院 <http://www.xueai8.com>

```
package com.xueai8.fraud.source;

import com.xueai8.fraud.entity.Transaction;
import org.apache.flink.streaming.api.functions.source.FromIteratorFunction;

import java.io.Serializable;
import java.util.Iterator;

/**
 * 自定义的数据生成器，运行后生成持续的信用卡交易数据（模拟），QPS 为 0.1 秒
 *
 * 这个类继承自 Flink 提供的 org.apache.flink.streaming.api.functions.source.FromIteratorFunction 类
 * 该类中有一个 run 方法，当运行时，会自动连续发送流指定迭代器数据源中的流数据
 */
public class TransactionSource extends FromIteratorFunction<Transaction> {
    private static final long serialVersionUID = 1L;

    public TransactionSource() {
        // 调用超类构造器，默认生成流数据
        super(new RateLimitedIterator<>(TransactionIterator.unbounded()));
    }

    // 泛型数据迭代器
    private static class RateLimitedIterator<T> implements Iterator<T>, Serializable {

        private static final long serialVersionUID = 1L;

        private final Iterator<T> inner;          // 数据源迭代器

        private RateLimitedIterator(Iterator<T> inner) {
            this.inner = inner;
        }

        @Override
        public boolean hasNext() {
            return inner.hasNext();
        }

        @Override
        public T next() {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            return inner.next();
        }
    }
}
```

5、创建一个测试类（带有 main 方法的主程序），使用 addSource 方法添加自定义数据源。

Scala 代码:

```
package com.xueai8.ch03

import com.xueai8.ch03.source.TransactionSource
import org.apache.flink.streaming.api.scala._

/**
 * 自定义数据源的测试类
 */
object CustomSourceDemo2 {

  def main(args: Array[String]): Unit = {
    // 获取流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 添加自定义流数据源
    val transactions = env.addSource(new TransactionSource()).name("transactions")

    // 输出交易记录
    transactions.print

    // 触发程序执行
    env.execute("Flink Custom Source")
  }
}
```

Java 代码:

```
package com.xueai8.ch03;

import com.xueai8.ch03.entity.Transaction;
import com.xueai8.ch03.source.TransactionSource;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * 自定义数据源测试类
 */
public class CustomSourceDemo2 {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 设置自定义流数据源
        DataStream<Transaction> transactions =
            env.addSource(new TransactionSource()).name("transactions");

        // 输出查看
        transactions.print();

        // 执行程序
    }
}
```

```

        env.execute("Transaction Stream");
    }
}

```

6、执行以上程序，查看控制台输出结果如下（部分结果）：

```

1> Transaction{accountId=1, timestamp=1577808000000, amount=188.23}
2> Transaction{accountId=2, timestamp=1577808360000, amount=374.79}
3> Transaction{accountId=3, timestamp=1577808720000, amount=112.15}
4> Transaction{accountId=4, timestamp=1577809080000, amount=478.75}
5> Transaction{accountId=5, timestamp=1577809440000, amount=208.85}
6> Transaction{accountId=1, timestamp=1577809800000, amount=379.64}
7> Transaction{accountId=2, timestamp=1577810160000, amount=351.44}
8> Transaction{accountId=3, timestamp=1577810520000, amount=320.75}
1> Transaction{accountId=4, timestamp=1577810880000, amount=259.42}
2> Transaction{accountId=5, timestamp=1577811240000, amount=273.44}
3> Transaction{accountId=1, timestamp=1577811600000, amount=267.25}
4> Transaction{accountId=2, timestamp=1577811960000, amount=397.15}
5> Transaction{accountId=3, timestamp=1577812320000, amount=0.219}
6> Transaction{accountId=4, timestamp=1577812680000, amount=231.94}
7> Transaction{accountId=5, timestamp=1577813040000, amount=384.73}
8> Transaction{accountId=1, timestamp=1577813400000, amount=419.62}
.....

```

代码说明

在上面的示例代码中，我们并没有直接创建 `SourceFunction` 类的子类，而是从 `FromIteratorFunction` 类继承。`FromIteratorFunction` 是 Flink 的 `org.apache.flink.streaming.api.functions.source` 包中自带的一个类，它已经实现了接口 `SourceFunction<T>`，包装了一个具体的泛型迭代器对象属性 `iterator`。通常使用时从它派生一个子类，并在构造时传入一个具体的迭代数据生成器类即可。其源码如下：

```

package org.apache.flink.streaming.api.functions.source;

import java.util.Iterator;
import org.apache.flink.annotation.PublicEvolving;
import org.apache.flink.streaming.api.functions.source.SourceFunction.SourceContext;

@PublicEvolving
public class FromIteratorFunction<T> implements SourceFunction<T> {
    private static final long serialVersionUID = 1L;
    private final Iterator<T> iterator;
    private volatile boolean isRunning = true;

    public FromIteratorFunction(Iterator<T> iterator) {
        this.iterator = iterator;
    }

    public void run(SourceContext<T> ctx) throws Exception {
        while(this.isRunning && this.iterator.hasNext()) {
            ctx.collect(this.iterator.next());
        }
    }

    public void cancel() {

```

```
        this.isRunning = false;
    }
}
```

【示例】Flink 官方给出的自定义数据源样例。这个数据源会发送 0-999 到下游系统。

```
public class ExampleCountSource implements SourceFunction<Long>, CheckpointedFunction {
```

```
    private long count = 0L;

    // 使用一个 volatile 类型变量控制 run 方法内循环的运行
    private volatile boolean isRunning = true;

    // 保存数据源状态的变量
    private transient ListState<Long> checkpointedCount;

    public void run(SourceContext<T> ctx) {
        while (isRunning && count < 1000) {
            // 这个同步块确保状态检查点，元素的内部状态更新和释放是一个原子操作
            // 此处必须要加锁，防止在 checkpoint 过程中，仍然发送数据
            synchronized (ctx.getCheckpointLock()) {
                ctx.collect(count);
                count++;
            }
        }
    }

    public void cancel() {
        // 设置 isRunning 为 false，终止 run 方法内循环的运行
        isRunning = false;
    }

    public void initializeState(FunctionInitializationContext context) {
        // 获取存储状态
        this.checkpointedCount = context
            .getOperatorStateStore()
            .getListState(new ListStateDescriptor<>("count", Long.class));

        // 如果数据源是从失败中恢复，则读取 count 的值，恢复数据源 count 状态
        if (context.isRestored()) {
            for (Long count : this.checkpointedCount.get()) {
                this.count = count;
            }
        }
    }

    public void snapshotState(FunctionSnapshotContext context) {
        // 保存数据到状态变量
        this.checkpointedCount.clear();
        this.checkpointedCount.add(count);
    }
}
```

```
}
```

3.3 数据转换

数据转换使用操作符（operator）将一个或多个数据流转换为新的数据流。转换输入可以是一个或多个数据流，转换输出也可以是零个、一个或多个数据流。程序可以将多个转换组合成复杂的数据流拓扑。

本节将介绍 Flink 提供的这些基本的转换运算。

3.3.1 map 转换

这是最简单的转换之一，其中输入是一个数据流，输出也是一个数据流。下表是对 map 转换的简单介绍。

转换运算符	描述	用法
Map DataStream → DataStream	获取一个元素并生成一个元素。	<code>dataStream.map { x => x * 2 }</code>

请看下面的示例程序。

Scala 代码:

```
package com.xueai8.ch03

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

object TransformerMap{
  def main(args: Array[String]) {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 首先从环境中获取一些数据，并使用操作符转换 DataStream[String]。比如：
    env.fromElements("Good good study", "Day day up")
      .map(_._toLowerCase)
      .print()

    // 对于流程序，只有执行了下面这个方法，流程序才真正开始执行
    env.execute("flink map transformation")
  }
}
```

Java 代码:

```
package com.xueai8.ch03;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

class TransformerMap{

  public static void main(String[] args) throws Exception {
```



```

// 设置流执行环境
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

// 首先从环境中获取一些数据，比如：
DataStream<String> ds = env.fromElements("Good good study", "Day day up");

// 执行 map 转换
DataStream<String> ds_map = ds.map(String::toLowerCase);

// 输出结果
ds_map.print();

// 对于流程序，只有执行了下面这个方法，流程序才真正开始执行
env.execute("flink map transformation");
}
}

```

输出结果如下所示：

```

good good study
day day up

```

3.3.2 flatMap 转换

flatMap 接受一条记录并输出零条、一条或多条记录。

转换运算符	描述	用法
FlatMap	获取一个元素并生成零个、一个或多个元素。	dataStream.flatMap { str => str.split(" ") }
DataStream → DataStream		

下面是进行 flatMap 转换的示例代码：

Scala 代码：

```

package com.xueai8.ch03

import org.apache.flink.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

object TransformerFlatMap{

  def main(args: Array[String]) {
    // 设置批处理执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 得到输入数据，flatMap 转换：
    env.fromElements("Good good study", "Day day up")
      .map(_._toLowerCase)
      .flatMap(_._split("\\W+"))           // 相当于先执行 map，再执行 flatten
      .print

    // 执行
    env.execute("flink flatmap transformation")
  }
}

```

```
}
```

Java 代码:

```
package com.xueai8.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * flatMap 转换
 */
public class TransformerFlatMap {
    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 首先从环境中获取一些数据，再执行 map 和 flatMap 转换：
        env.fromElements("Good good study", "Day day up")
            .map(String::toLowerCase)
            // 传入一个匿名函数
            .flatMap((FlatMapFunction<String, String>) (value, out) -> {
                for(String word: value.split("\\W+")){
                    out.collect(word);
                }
            }).returns(Types.STRING)
            .print();

        // 对于流程序，只有执行了下面这个方法，流程序才真正开始执行
        env.execute("flink flatmap transformation");
    }
}
```

上面的代码中，flatMap 函数的传入参数是一个 Lambda 表达式，它对于 flatMap 的支持是无法猜测出来类型的，必须通过 returns(Types.STRING) 指定具体的返回值类型。

或者，可以像下面这样，使用匿名内部类（而不是 Lambda 表达式）：

```
package com.xueai8.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.util.Collector;

/**
 * flatMap 转换
 */
public class TransformerFlatMap2 {
    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
```

```
// 首先从环境中获取一些数据，再执行 map 和 flatMap 转换：
DataStream<String> ds = env.fromElements("Good good study", "Day day up")
    .map(String::toLowerCase)
    .flatMap(new FlatMapFunction<String, String>() {
        @Override
        public void flatMap(String value, Collector<String> out) throws Exception {
            for(String word: value.split("\\W+")){
                out.collect(word);
            }
        }
    });

// 输出
ds.print();

// 对于程序，只有执行了下面这个方法，程序才真正开始执行
env.execute("flink flatmap transformation");
}
```

输出结果如下所示：

```
good
good
study
day
day
up
```

3.3.3 filter 转换

`filter` 函数对条件进行评估，如果结果为 `true`，则该条数据输出。`filter` 函数可以输出零个记录。

转换运算符	描述	用法
Filter DataStream → DataStream	对每个元素求布尔函数的值，并保留函数返回 <code>true</code> 的元素	<code>dataStream.filter { _ != 0 }</code>

下面是进行 `filter` 转换的示例代码。

Scala 代码：

```
package com.xueai8.ch03

import org.apache.flink.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

object TransformerFilter{

    def main(args: Array[String]) {
        // 设置批处理执行环境
        val env = StreamExecutionEnvironment.getExecutionEnvironment

        // 得到输入数据，然后执行 filter 转换
        env.fromElements("Good good study", "Day day up")
```

```
        .map(_toLowerCase)
        .filter(_contains("study"))
        .print()

    // 执行
    env.execute("flink filter transformation")
  }
}
```

Java 代码:

```
package com.xueai8.ch03;

import org.apache.flink.api.common.functions.FilterFunction;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * filter 转换
 */
public class TransformerFilter {
    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 首先从环境中获取一些数据，再执行 map 和 filter 转换：
        env.fromElements("Good good study", "Day day up")
            .map(String::toLowerCase)
            .filter((FilterFunction<String> s -> s.contains("study"))
            .print();

        // 对于流程序，只有执行了下面这个方法，流程序才真正开始执行
        env.execute("flink map transformation");
    }
}
```

输出结果如下所示：

```
good good study
```

数据流中的数据也可以封装到 POJO 类中。如下所示：

Scala 代码实现：

```
package com.xueai8.ch03

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

/**
 * filter 转换
 */
object TransformerFilter2 {

    // case class 类，用来表示事件流中的事件
    case class Person(name:String,age:Int)    // 需要定义在 main 方法外部
```

```
def main(args: Array[String]): Unit = {  
  
    // 设置批处理执行环境  
    val env = StreamExecutionEnvironment.getExecutionEnvironment  
  
    // 得到输入数据，然后执行 filter 转换  
    val personDS = env.fromElements(Person("张三", 25), Person("李四", 18), Person("小多米", 2))  
    val adults = personDS.filter(_._age >= 18)  
  
    // 输出过滤后的数据流  
    adults.print  
  
    // 执行  
    env.execute("flink filter transformation")  
}
```

Java 代码实现:

```
package com.xueai8.ch03;  
  
import org.apache.flink.api.common.functions.FilterFunction;  
import org.apache.flink.streaming.api.datastream.DataStream;  
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
  
/**  
 * filter 转换  
 */  
public class TransformerFilter2 {  
  
    // POJO 类，用来表示事件流中的事件  
    public static class Person{  
        String name;  
        Integer age;  
  
        Person() {}  
  
        Person(String name, Integer age) {  
            this.name = name;  
            this.age = age;  
        }  
  
        public String toString() {  
            return this.name + ": 年龄 " + this.age.toString();  
        }  
    }  
  
    public static void main(String[] args) throws Exception {  
        // 设置流执行环境  
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  
  
        // 加载数据源
```

```

DataStream<Person> persons = env.fromElements(
    new Person("张三", 35),
    new Person("李四", 35),
    new Person("小米", 2));

// 执行 filter 转换
DataStream<Person> adults = persons.filter(new FilterFunction<Person>() {
    @Override
    public boolean filter(Person person) throws Exception {
        return person.age >= 18;
    }
});

// 输出结果流
adults.print();

// 对于流程序，只有执行了下面这个方法，流程序才真正开始执行
env.execute("flink filter transformation");
}
}

```

输出结果如下：

张三：年龄 35

李四：年龄 35

3.3.4 keyBy 转换

有一些转换(如 join、coGroup、keyBy、groupBy)要求在元素集合上定义一个 key。还有一些转换(如 reduce、groupReduce、aggregate、windows)可以应用在按 key 分组的数据上。

转换运算符	描述	用法
KeyBy DataStream → KeyedStream	逻辑上将一个流划分为不相连的分区，每个分区包含相同key的元素。在内部，这是通过哈希分区实现的。这个转换返回一个KeyedStream	dataStream.keyBy("someKey") // 按字段 dataStream.keyBy(0) //按元组第一个元素

Flink 的数据模型不是基于 key-value 对的。因此，不需要将数据集类型物理打包为 key 和 value。key 是“虚拟的”：它们被定义一个函数，该函数可指定数据流中实际数据的哪个字段（或属性）用作 key。需要注意的是，如果流元素是 POJO，那么它必须重写 hashCode()方法。

最简单的情况是对元组的一个或多个字段进行分组。请看下面的示例代码：

Scala 代码：

```

package com.xueai8.scala.ch03

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

/**
 * keyBy 转换
 */
object TransformerKeyBy{
    def main(args: Array[String]): Unit = {
        // 设置流执行环境
        val env = StreamExecutionEnvironment

```

```

// 得到输入数据，进行转换：
env.fromElements("Good good study", "Day day up")
  .map(_._toLowerCase)
  .flatMap(_._split("\\W+"))           // 相当于先 map，再 flatten
  .map(_._1)
  .keyBy(0)                            // 按元组索引
  // .keyBy(t => t._1)                 // 或按元组的第一个字段分区
  .print

// 触发程序执行
env.execute("flink keyBy transformation")
}
}

```

Java 代码:

```

package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.tuple.Tuple;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.KeyedStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.util.Collector;

/**
 * keyBy 转换
 */
public class TransformerKeyBy {
    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 加载数据源，并执行 flatMap 转换
        DataStream<String> ds = env.fromElements("good good study", "day day up")
            .flatMap(new FlatMapFunction<String, String>() {
                @Override
                public void flatMap(String value, Collector<String> out)
                    throws Exception {
                    for (String word: value.split("\\W+")){
                        out.collect(word);
                    }
                }
            });

        // 通过 map 转换，将事件流中事件的数据类型变换为(word,1)元组形式
        DataStream<Tuple2<String,Integer>> ds_map =
            ds.map(new MapFunction<String, Tuple2<String,Integer>>() {
                @Override

```

```
        public Tuple2<String, Integer> map(String s) throws Exception {
            return new Tuple2<>(s,1);
        }
    });

    // keyBy 转换，按 key 重分区
    KeyedStream<Tuple2<String,Integer>,Tuple> ds_keyed = ds_map.keyBy(0);

    // 输出
    ds_keyed.print();

    // 执行
    env.execute("flink keyBy transformation");
}
}
```

输出结果如下：

```
(up,1)
(study,1)
(day,1)
(day,1)
(good,1)
(good,1)
```

使用字段表达式来定义 key

在 Flink 1.11 版本之前，也可以使用字段表达式来定义 key（从 flink 1.11 开始已弃用）。可以使用基于字符串的字段表达式来引用嵌套的字段，并为分组、排序、连接或联合分组定义 key。字段表达式使得在(嵌套的)复合类型(如 Tuple 和 POJO 类型)中选择字段变得非常容易。例如，在下面的代码中，我们按成员性别分区。

Scala 代码：

```
package com.xueai8.ch03

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

/**
 * keyBy 转换，使用字段表达式
 */
object TransformerKeyBy2 {

    // 事件流中事件的数据类型采用 case class
    case class Person(name:String, gender:String)

    def main(args: Array[String]): Unit = {
        // 设置流执行环境
        val env = StreamExecutionEnvironment.getExecutionEnvironment

        // 得到输入数据，进行转换：
        val personDS = env.fromElements(Person("张三", "男"), Person("李四", "女"), Person("小多米", "男"))
```



```
    personDS.keyBy("gender").print()

    env.execute("flink keyBy transformatiion")
}
}
```

Java 代码:

```
package com.xueai8.ch03;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

public class TransformerKeyBy2 {

    // POJO 类，用来表示事件流中的事件
    public static class Person {
        public String name;           // 姓名
        public String gender;         // 性别

        public Person() {}

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

        @Override
        public String toString() {
            return this.name + ": 性别 " + this.gender;
        }
    }

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 加载数据源
        DataStream<Person> personDS = env.fromElements(
            new Person("张三", "男"),
            new Person("李四", "女"),
            new Person("小多米", "男"));

        // 执行 keyBy 转换，按性别(gender)分区
        personDS.keyBy("gender").print();

        // 执行
        env.execute("flink keyBy transformatiion");
    }
}
```

执行程序，输出结果如下：

6> 李四: 性别 女

```
1> 张三: 性别 男
1> 小多米: 性别 男
```

注意到，相同的 key 在同一分区内被计算。另外要特别注意的是，对于要充当 key 的 POJO 类，必须满足以下条件：

- ☐ 字段名必须声明为 public 的；
- ☐ 必须有默认的空参构造器；
- ☐ 所有构造器必须声明为 public 的。

使用 “key selector”函数来定义 key

定义 key 的另一种方法是使用 “key selector” 函数。一个 key selector 函数接受单个元素作为输入，并返回该元素的 key。返回的 key 可以是任何类型的，可以从确定性计算中得到。例如，在下面的示例中，我们根据成员的年龄，分为两组：成年人和未成年人。

Scala 代码：

```
package com.xueai8.ch03

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

/**
 * keyBy 转换，使用 key selector
 */
object TransformerKeyBy3 {

    // 事件流中事件的数据类型采用 case class
    case class Person(name:String, age:Integer)

    def main(args: Array[String]): Unit = {
        // 设置流执行环境
        val env = StreamExecutionEnvironment.getExecutionEnvironment

        // 得到输入数据，进行转换：
        val personDS = env.fromElements(
            Person("张三", 16),
            Person("李四", 18),
            Person("王老五", 35),
            Person("赵小六", 23),
            Person("小多米", 12))

        personDS.keyBy(person => if (person.age >= 18) "adult" else "young").print

        env.execute("flink keyBy transformation")
    }
}
```

Java 代码：

```
package com.xueai8.ch03;

import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.streaming.api.datastream.DataStream;
```

```
import org.apache.flink.streaming.api.datastream.KeyedStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * keyBy 转换，使用 key selector
 */

public class TransformerKeyBy3 {

    // POJO 类，用来表示事件流中的事件
    public static class Person {
        public String name;           // 姓名
        public Integer age;           // 年龄

        public Person() {}

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

        @Override
        public String toString() {
            return this.name + ": 年龄 " + this.age;
        }
    }

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 加载数据源
        DataStream<Person> personDS = env.fromElements(
            new Person("张三", 16),
            new Person("李四", 18),
            new Person("王老五", 35),
            new Person("赵小六", 23),
            new Person("小多米", 12)
        );

        // 执行 keyBy 转换，按年龄分区（成年人和未成年人）
        KeyedStream<Person, String> keyed = personDS.keyBy(new KeySelector<Person, String>() {
            @Override
            public String getKey(Person person) throws Exception {
                return person.age >= 18 ? "adult" : "young";
            }
        });

        keyed.print();
    }
}
```

```
// 执行
env.execute("flink keyBy transformation");
}
```

执行以下代码，输出结果如下：

```
6> 李四: 年龄 18
1> 张三: 年龄 16
6> 王老五: 年龄 35
1> 小多米: 年龄 12
6> 赵小六: 年龄 23
```

3.3.5 reduce 转换

转换运算符	描述	用法
Reduce KeyedStream → DataStream	Key-value数据流上的“滚动”约减。将当前元素与最后一个减少的值组合在一起，并发出新的值	keyedStream.reduce { _ + _ }

下面是进行 reduce 转换的示例代码（对 DataStream 进行求和）：

Scala 代码：

```
package com.xueai8.scala.ch03

import org.apache.flink.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

/**
 * reduce 转换
 */
object TransformerReduce {

  def main(args: Array[String]): Unit = {
    // 设置批处理执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 加载数据源，然后执行 DataStream 转换
    env.fromElements("Good good study", "Day day up")
      .map(_._toLowerCase)           // 转换为小写
      .flatMap(_._split("\\W+"))     // 相当于先 map，再 flatten
      .map(_._1)                     // 转换为元组类型
      .keyBy(_._1)                   // 按单词进行分组
      .reduce((a,b) => (a._1, a._2 + b._2)) // reduce
      .print

    // 触发执行
    env.execute("flink reduce transformation")
  }
}
```

Java 代码：

```
package com.xueai8.java.ch03;
```

```

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.ReduceFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.util.Collector;

/**
 * reduce 转换
 */
public class TransformerReduce {
    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 首先加载数据，然后执行转换
        DataStream<Tuple2<String, Integer>> ds = env
            .fromElements("Good good study", "Day day up")
            .map(String::toLowerCase)
            .flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
                @Override
                public void flatMap(String s, Collector<Tuple2<String, Integer>> collector)
                    throws Exception {
                    for(String word : s.split("\\W+")){
                        collector.collect(new Tuple2<>(word,1));
                    }
                }
            });

        ds//.keyBy(0)    // 已弃用
        .keyBy(t -> t.f0)
        .reduce(new ReduceFunction<Tuple2<String, Integer>>() {
            @Override
            public Tuple2<String, Integer> reduce(Tuple2<String, Integer> t1, Tuple2<String, Integer> t2)
                throws Exception {
                return new Tuple2<>(t1.f0, t1.f1 + t2.f1);
            }
        }).print();

        // 对于流程序，只有执行了下面这个方法，流程序才真正开始执行
        env.execute("flink reduce transformiion");
    }
}

```

输出结果如下所示：

```

6> (good,1)
4> (up,1)
6> (good,2)
7> (day,1)
7> (day,2)
5> (study,1)

```

3.3.6 聚合转换

DataStream API 支持各种聚合，比如 min、max、sum 等等。这些函数可以应用于 KeyedDataStream，以获得滚动聚合。

转换运算符	描述	用法
Aggregations KeyedStream → DataStream	在key-value数据流上滚动聚合。 min和minBy之间的区别是，min返回最小值，而minBy返回该字段中具有最小值的元素(max和maxBy也是如此)。	keyedStream.sum(0) keyedStream.sum("key") keyedStream.min(0) keyedStream.min("key") keyedStream.max(0) keyedStream.max("key") keyedStream.minBy(0) keyedStream.minBy("key") keyedStream.maxBy(0) keyedStream.maxBy("key")

下面是进行聚合转换的示例代码：

Scala 代码：

```
package com.xueai8.scala.ch03

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

/**
 * DataStream API 支持各种聚合，比如 min、max、sum 等等。
 * 这些函数可以应用于 KeyedDataStream，以获得滚动聚合。
 */
object TransformerAgg{
  def main(args: Array[String]): Unit = {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 首先从环境中获取一些数据，并使用操作符转换 DataStream[String]。比如：
    val ds_keyed = env.fromElements(("good",1),("good",2),("study",1)).keyBy(_._1)

    ds_keyed.sum(1).print
    ds_keyed.min(1).print
    ds_keyed.max(1).print
    ds_keyed.minBy(1).print
    ds_keyed.maxBy(1).print

    env.execute("flink aggregation transformation")
  }
}
```

Java 代码：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
```

```

* DataStream API 支持各种聚合，比如 min、max、sum 等等。
* 这些函数可以应用于 KeyedDataStream，以获得滚动聚合。
*/
public class TransformerAgg {
    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 首先从环境中获取一些数据，再执行 map 和 flatMap 转换：
        DataStream<Tuple2<String,Integer>> ds = env.fromElements(
            new Tuple2<>("good",1),
            new Tuple2<>("good",2),
            new Tuple2<>("study",1));

        ds.keyBy(t -> t.f0).sum(1).print();           // 参数也可以是字段名
        ds.keyBy(t -> t.f0).min(1).print();
        ds.keyBy(t -> t.f0).max(1).print();
        ds.keyBy(t -> t.f0).minBy(1).print();
        ds.keyBy(t -> t.f0).maxBy(1).print();

        // 对于流程序，只有执行了下面这个方法，流程序才真正开始执行
        env.execute("flink aggregation transformation");
    }
}

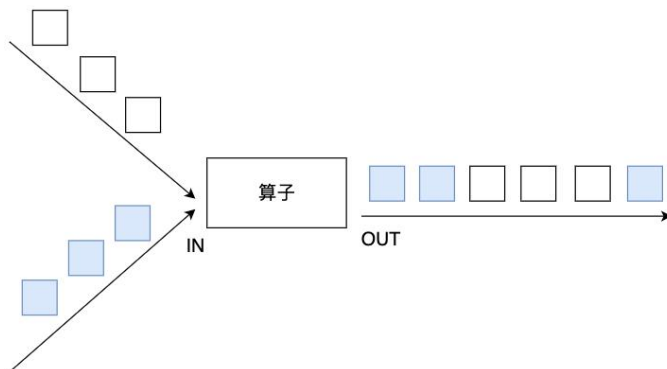
```

3.3.7 union 转换

union 函数执行两个或多个数据流的联合。对两个或者两个以上的 DataStream 进行 union 操作，产生一个包含所有 DataStream 元素的新 DataStream。

转换运算符	描述	用法
Union DataStream* → DataStream	合并两个或多个数据流（并集），创建包含来自所有流的所有元素的新流。 注意：如果将一个数据流与它自己相合并，将得到结果流中的每个元素两次。	dataStream.union(stream1, stream2, ...)

在 DataStream 上使用 union 操作可以合并多个同类型的数据流，并生成同类型的数据流，即将多个 DataStream[T] 合并为一个新的 DataStream[T]。数据将按照先进先出（First In First Out）的模式合并，且不去重。下图 union 对白色和深色两个数据流进行合并，生成一个数据流。



下面是进行 union 转换的示例代码。

Scala 代码:

```
package com.xueai8.scala.ch03

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

/**
 * union 转换
 * Union 函数执行两个或多个数据流的联合。这将并行地组合数据流。
 * 如果我们将一个流与它自己组合起来，那么它将输出每个记录两次。
 */
object TransformerUnion {
  def main(args: Array[String]): Unit = {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // union
    // 第一个数据集
    val ds1 = env.fromElements("good good study").flatMap(_._toLowerCase.split("\\W+")).map(_._, 1))

    // 第二个数据集
    val ds2 = env.fromElements("day day up").flatMap(_._toLowerCase.split("\\W+")).map(_._, 1)

    // 合并两个数据集并输出
    ds1.union(ds2).print()

    // 执行
    env.execute("flink union transformation")
  }
}
```

Java 代码:

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.util.Collector;

/**
 * union 转换
 * Union 函数执行两个或多个数据流的联合。这将并行地组合数据流。
 * 如果我们将一个流与它自己组合起来，那么它将输出每个记录两次。
 */
public class TransformerUnion {
  public static void main(String[] args) throws Exception {
    // 设置流执行环境
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

    // union 转换
```



```
// 第一个数据集
DataStream<String> ds1 = env.fromElements("good good study")
    .map(String::toLowerCase)
    .flatMap(new FlatMapFunction<String, String>() {
        @Override
        public void flatMap(String value, Collector<String> out) throws Exception {
            for(String word: value.split("\\W+")){
                out.collect(word);
            }
        }
    });

DataStream<Tuple2<String, Integer>> ds1_map =
    ds1.map(new MapFunction<String, Tuple2<String, Integer>>() {
        @Override
        public Tuple2<String, Integer> map(String s) throws Exception {
            return new Tuple2<>(s, 1);
        }
    });

// 第二个数据集
DataStream<String> ds2 = env.fromElements("day day up")
    .map(String::toLowerCase)
    .flatMap(new FlatMapFunction<String, String>() {
        @Override
        public void flatMap(String value, Collector<String> out) throws Exception {
            for(String word: value.split("\\W+")){
                out.collect(word);
            }
        }
    });

DataStream<Tuple2<String, Integer>> ds2_map =
    ds2.map(new MapFunction<String, Tuple2<String, Integer>>() {
        @Override
        public Tuple2<String, Integer> map(String s) throws Exception {
            return new Tuple2<>(s, 1);
        }
    });

// 合并两个数据集并输出
DataStream<Tuple2<String, Integer>> ds1_and_ds2 = ds1_map.union(ds2_map);
ds1_and_ds2.print();

// 执行
env.execute("flink union transformation");
}
```

执行以上代码，输出结果如下所示：

```
6> (good,1)
7> (day,1)
7> (day,1)
```

```
7> (up,1)
6> (good,1)
6> (study,1)
```

3.3.8 connect 转换

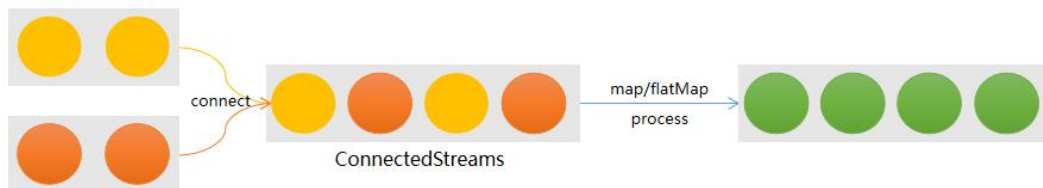
`DataStream` 的 `connect` 操作创建的是 `ConnectedStreams` 或 `BroadcastConnectedStream`，它用了两个泛型，即不要求两个 `dataStream` 的 `element` 是同一类型。

转换运算符	描述	用法
Connect <code>DataStream, DataStream → ConnectedStreams</code>	“连接”两个数据流，保持它们的类型，允许在两个流之间共享状态	<pre>stream1 : DataStream[Int] = ... stream2 : DataStream[String] = ... val connectedStreams = stream1.connect(stream2)</pre>

`union` 虽然可以合并多个数据流，但有一个限制，即多个数据流的数据类型必须相同。`connect` 提供了和 `union` 类似的功能，用来连接两个数据流，它与 `union` 的区别在于：

- ❑ `connect` 只能连接两个数据流，`union` 可以连接多个数据流。
- ❑ `connect` 所连接的两个数据流的数据类型可以不一致，`union` 所连接的两个数据流的数据类型必须一致。
- ❑ 两个 `DataStream` 经过 `connect` 之后被转化为 `ConnectedStreams`，`ConnectedStreams` 会对两个流的数据应用不同的处理方法，且双流之间可以共享状态。

两个输入流经过 `connect` 合并后，可以进一步使用 `CoProcessFunction`、`CoMap`、`CoFlatMap`、`KeyedCoProcessFunction` 等 API 对两个流分别处理。



`ConnectedStreams` 提供了 `keyBy` 方法用于指定两个 stream 的 `keySelector`，提供了 `map`、`flatMap`、`process`、`transform` 操作，其中前三个操作最后都是调用 `transform` 操作。`transform` 操作接收 `TwoInputStreamOperator` 类型的 `operator`，然后转换为 `SingleOutputStreamOperator`。`map` 操作接收 `CoMapFunction`，`flatMap` 操作接收 `CoFlatMapFunction`，`process` 操作接收 `CoProcessFunction`。

下面是进行 `connect` 转换的示例代码。

Scala 代码：

```
package com.xueai8.ch03
```

```
import org.apache.flink.streaming.api.functions.co.{CoMapFunction, CoProcessFunction}
import org.apache.flink.streaming.api.scala._
import org.apache.flink.util.Collector
```

```
/**
```

```
 * connect 转换操作
```

```
 * connect 和 union 都有一个共同的作用，就是将 2 个流或多个流合成一个流。
```

```
 * 但是两者的区别是：union 连接的 2 个流的类型必须一致，connect 连接的流可以不一致，但是可以统一处理。
```

```
 */
```

```
object TransformerConnect {
```

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

```
// 设置流执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

// connect
// 第一个数据集
val ds1:DataStream[(String, Int)] = env.fromElements("good good study").flatMap(_._split("\\W+")).map(_._1)

// 第二个数据集
val ds2:DataStream[String] = env.fromElements("day day up").flatMap(_._split("\\W+"))

// 连接两个数据集
val ds:ConnectedStreams[(String, Int),String] = ds1.connect(ds2)

// 调用 process 方法
// CoProcessFunction 泛型参数: [输入流 1 数据类型, 输入流 2 数据类型, 输出流数据类型]
ds.process(new CoProcessFunction[(String, Int), String, (String, Int)]{
  // 处理输入流 1 的元素
  override def processElement1(in1: (String, Int),
                                context: CoProcessFunction[(String, Int), String, (String, Int)]#Context,
                                out: Collector[(String, Int)]): Unit = {
    out.collect(in1)    // 发送给下游算子
  }

  // 处理输入流 2 的元素
  override def processElement2(in2: String,
                                context: CoProcessFunction[(String, Int), String, (String, Int)]#Context,
                                out: Collector[(String, Int)]): Unit = {
    out.collect((in2,1)) // 将来自输入流 2 的元素转换为元组, 再发送给下游算子
  }
}).print

// map
ds.map(new CoMapFunction[(String, Int), String, (String, Int)] {
  override def map1(in1: (String, Int)): (String, Int) = {
    (in1._1.toUpperCase, in1._2)
  }

  override def map2(in2: String): (String, Int) = {
    (in2, 1)
  }
}).print

// 执行
env.execute("flink connect transformation")
}
```

Java 代码:

```
package com.xueai8.ch03;
```

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.ConnectedStreams;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.co.CoMapFunction;
import org.apache.flink.streaming.api.functions.co.CoProcessFunction;
import org.apache.flink.util.Collector;

/**
 * connect 转换操作
 * connect 和 union 都有一个共同的作用，就是将 2 个流或多个流合成一个流。
 * 但是两者的区别是：union 连接的 2 个流的类型必须一致，connect 连接的流可以不一致，但是可以统一处理。
 */
public class TransformerConnect {
    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // connect 转换操作
        // 第一个数据集
        DataStream<Tuple2<String, Integer>> ds1 = env.fromElements("good good study")
            .map(String::toLowerCase)
            .flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
                @Override
                public void flatMap(String value, Collector<Tuple2<String, Integer>> out)
                    throws Exception {
                    for(String word: value.split("\\W+")){
                        out.collect(new Tuple2<>(word,1));
                    }
                }
            });

        // 第二个数据集
        DataStream<String> ds2 = env.fromElements("day day up")
            .map(String::toLowerCase)
            .flatMap(new FlatMapFunction<String, String>() {
                @Override
                public void flatMap(String value, Collector<String> out) throws Exception {
                    for(String word: value.split("\\W+")){
                        out.collect(word);
                    }
                }
            });

        // 连接两个数据集
        ConnectedStreams<Tuple2<String, Integer>,String> ds = ds1.connect(ds2);

        ds.process(new CoProcessFunction<Tuple2<String,Integer>, String, Tuple2<String,Integer>>() {
            @Override
```

```

        public void processElement1(Tuple2<String, Integer> t,
                                    Context context,
                                    Collector<Tuple2<String,Integer>> out) throws Exception {
            out.collect(t);
        }

        @Override
        public void processElement2(String s, Context context, Collector<Tuple2<String,Integer>> out)
            throws Exception {
            out.collect(new Tuple2<>(s,1));
        }
    }).print("process");

    ds.map(new CoMapFunction<Tuple2<String,Integer>, String, Tuple2<String,Integer>>() {
        @Override
        public Tuple2<String, Integer> map1(Tuple2<String, Integer> t) throws Exception {
            return new Tuple2<>(t.f0.toUpperCase(),t.f1);
        }

        @Override
        public Tuple2<String, Integer> map2(String s) throws Exception {
            return new Tuple2<>(s, 1);
        }
    }).print("map");

    // 执行
    env.execute("flink connect transformation");
}
}

```

执行以上代码，输出结果如下所示：

```

process:1> (day,1)
process:1> (day,1)
process:1> (up,1)
map:1> (day,1)
map:1> (day,1)
map:1> (up,1)
map:5> (GOOD,1)
map:5> (GOOD,1)
map:5> (STUDY,1)
process:5> (good,1)
process:5> (good,1)
process:5> (study,1)

```

另外，在 Scala 实现的 API 中，对于 `ConnectedStreams` 的 `map` 转换，还可以分别执行两个不同的函数，这时两个流元素输出的数据结构可以不同。请看下面的代码：

```

package com.xueai8.ch03

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

```

```
object TransformerConnect {

  def main(args: Array[String]): Unit = {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // connect
    // 第一个数据集
    val ds1:DataStream[(String, Int)] = env.fromElements("good good study").flatMap(_._split("\\W+")).map(_._1)

    // 第二个数据集
    val ds2:DataStream[String] = env.fromElements("day day up").flatMap(_._split("\\W+"))

    // 连接两个数据集
    val ds:ConnectedStreams[(String, Int),String] = ds1.connect(ds2)

    // map, 执行两个不同的函数, 这时两个流元素输出的数据结构可以不同
    ds.map(
      d1 => {(d1._1.toUpperCase, d1._2)},
      d2 => {d2.toUpperCase}
    ).print

    // 执行
    env.execute("flink connect transformation")
  }
}
```

输出结果如下:

```
2> DAY
1> (GOOD,1)
2> DAY
1> (GOOD,1)
2> UP
1> (STUDY,1)
```

3.3.9 coMap, coFlatMap 转换

ConnectedStreams → DataStream: 作用于 ConnectedStreams 上, 功能与 map 和 flatMap 一样, 对 ConnectedStreams 中的每一个 Stream 分别进行 map 和 flatMap 处理。

转换运算符	描述	用法
CoMap, CoFlatMap ConnectedStreams → DataStream	类似于连接数据流上的map和flatMap	<pre>connectedStreams.map((_ : Int) => true, (_ : String) => false) connectedStreams.flatMap((_ : Int) => true, (_ : String) => false)</pre>

其中 map 的处理见上小节的示例。下面是进行 flatMap 转换的示例代码:

Scala 代码:

```
package com.xueai8.ch03
```

```

import org.apache.flink.streaming.api.functions.co.CoFlatMapFunction
import org.apache.flink.streaming.api.scala._
import org.apache.flink.util.Collector

/**
 * connect 转换操作
 */
object TransformerConnect2 {
  def main(args: Array[String]): Unit = {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // union
    // 第一个数据集
    val ds1:DataStream[String] = env.fromElements("good good study","day day up")

    // 第二个数据集
    val ds2:DataStream[Int] = env.fromElements(1,2,3)

    // 连接两个数据集
    val ds:ConnectedStreams[String, Int] = ds1.connect(ds2)

    // flatMap
    ds.flatMap(new CoFlatMapFunction[String,Int,String] {
      override def flatMap1(in1: String, out: Collector[String]): Unit = {
        in1.split(" ").foreach(out.collect)
      }

      override def flatMap2(in2: Int, out: Collector[String]): Unit = {
        1.to(in2).foreach((e) => out.collect(e.toString))
      }
    })
    .print

    // 执行
    env.execute("flink connect transformation")
  }
}

```

Java 代码:

```

package com.xueai8.ch03;

import org.apache.flink.streaming.api.datastream.ConnectedStreams;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.co.CoFlatMapFunction;
import org.apache.flink.util.Collector;

/**
 * connect 转换操作

```

```
*/
public class TransformerConnect2 {
    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // connect 转换操作
        // 第一个数据集
        DataStream<String> ds1 = env.fromElements("good good study","day day up");

        // 第二个数据集
        DataStream<Integer> ds2 = env.fromElements(1,2,3);

        // 连接两个数据集
        ConnectedStreams<String, Integer> ds = ds1.connect(ds2);

        // flatMap
        ds.flatMap(new CoFlatMapFunction<String, Integer, String>() {
            @Override
            public void flatMap1(String s, Collector<String> out) throws Exception {
                for(String word : s.split(" ")){
                    out.collect(word);
                }
            }

            @Override
            public void flatMap2(Integer in, Collector<String> out) throws Exception {
                for(int i = 1;i<=in;i++){
                    out.collect(String.valueOf(i));
                }
            }
        }).print();

        // 执行
        env.execute("flink connect transformation");
    }
}
```

输出结果如下:

```
4> 1
2> good
2> good
2> study
3> day
4> 2
3> day
3> up
4> 3
3> 1
3> 2
2> 1
```


对于 Scala 实现的 API，因为它支持函数作为参数，所以还可以更加简洁的编码：

```
package com.xueai8.ch03

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

/**
 * connect 转换操作
 */
object TransformerConnect2 {
  def main(args: Array[String]): Unit = {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // union
    // 第一个数据集
    val ds1:DataStream[String] = env.fromElements("good good study","day day up")

    // 第二个数据集
    val ds2:DataStream[Int] = env.fromElements(1,2,3)

    // 连接两个数据集
    val ds:ConnectedStreams[String, Int] = ds1.connect(ds2)

    // flatMap
    ds.flatMap(
      (e1:String) => e1.split(" "),
      (e2:Int) => 1 to e2
    ).print

    // 执行
    env.execute("flink connect transformation")
  }
}
```

3.3.10 iterate 转换

在流中创建“反馈（feedback）”循环，通过将算子的输出重定向到某个先前的算子。这对于定义不断更新模型的算法特别有用。

转换运算符	描述	用法
Iterate DataStream → IterativeStream → DataStream	通过将一个操作符的输出重定向到前一个操作符，在流中创建一个“反馈”循环。这对于定义不断更新模型的算法特别有用。 右侧的代码从一个流开始，并持续地应用迭代体。大于0的元素被发送回反馈通道，其余的元素被转发到下游。	<pre>initialStream.iterate { iteration => { val iterationBody = iteration.map { /*do something*/ } (iterationBody.filter(_ > 0), iterationBody.filter(_ <= 0)) } }</pre>

对 DataStream 使用 iterate() 方法创建 IterativeStream，使用 IterativeStream 的 closeWith 方法来关闭 feedbackStream。

DataStream 提供了两个 `iterate` 方法，它们创建并返回 `IterativeStream`，无参的 `iterate` 方法其 `maxWaitTimeMillis` 为 0。如下所示：

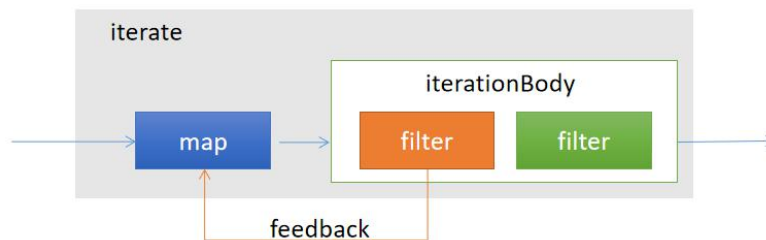
```
@Public
public class DataStream<T> {
    //.....

    @PublicEvolving
    public IterativeStream<T> iterate() {
        return new IterativeStream<>(this, 0);
    }

    @PublicEvolving
    public IterativeStream<T> iterate(long maxWaitTimeMillis) {
        return new IterativeStream<>(this, maxWaitTimeMillis);
    }

    //.....
}
```

`IterativeStream` 主要提供了两个方法，一个是 `closeWith` 方法，用于关闭迭代，它主要用于定义要被 `feedback` 到 `iteration` 头部的这部分 `iteration` (可以理解为回流，或者类似递归的操作，`filter` 控制的是递归的条件，通过 `filter` 的 `elements` 会重新进入 `IterativeStream` 的头部继续参与后面的运算操作)；`withFeedbackType` 方法创建了 `ConnectedIterativeStreams`。



下面我们通过一个示例来理解这个运算。在下面的示例中，我们生成"-2,3,4,5"这样的数据流。我们的目标是只输出非负整数。如果遇到负整数，被发送回反馈通道，并不断地应用迭代体（每次加 1，直到非负为止）；对于遇到的非负整数，则素被向下转发。

Scala 代码：

```
package com.xueai8.scala.ch03
```

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

```
/**
```

```
 * iterate 转换操作
```

```
 * 在流中创建“反馈（feedback）”循环，通过将一个算子的输出重定向到某个先前的算子。
```

```
 * 这对于定义不断更新模型的算法特别有用。
```

```
 *
```

```
 * 在下面的示例中，生成"-2,3,4,5"这样的数据流。
```

```
 * 目标是只输出非负整数。
```

```
 * 如果遇到负整数，被发送回反馈通道，并不断地应用迭代体（每次加 1，直到非负为止）；
```

```
 * 对于遇到的非负整数，则素被向下转发。
```

```
 */
```

```
object TransformerIterate {
```

```

def main(args: Array[String]): Unit = {
  // 设置流执行环境
  val env = StreamExecutionEnvironment.getExecutionEnvironment

  // 加载数据集
  val initialStream = env.fromElements(-2,3,4,5)

  // 迭代操作
  // 使用 iterate 创建 IterativeStream
  initialStream.iterate {
    iteration => {
      // 迭代计算部分
      val iterationBody = iteration.map { value => if(value<0) value+1 else value}
      // 下面元组第一部分发回到上一步迭代计算，第二部分则继续正常的流
      (iterationBody.filter(_ < 0), iterationBody.filter(_ >= 0))
    }
  }.print
  .setParallelism(1)

  // 执行
  env.execute("flink iterate transformation")
}

```

Java 代码:

```

package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.FilterFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.IterativeStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * 在流中创建“反馈（feedback）”循环，通过将一个算子的输出重定向到某个先前的算子。
 * 这对于定义不断更新模型的算法特别有用。
 *
 * 在下面的示例中，生成"-2,3,4,5"这样的数据流。
 * 目标是只输出非负整数。
 * 如果遇到负整数，被发回反馈通道，并不断地应用迭代体（每次加 1，直到非负为止）；
 * 对于遇到的非负整数，则素被向下转发。
 */
public class TransformerIterate {

  public static void main(String[] args) throws Exception {
    // 设置流执行环境
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

    // 加载数据集
    DataStream<Long> initialStream = env.setParallelism(1).fromElements(-2L, 3L, 4L, 5L);

```

```
// 转换为迭代流
IterativeStream<Long> iteration = initialStream.iterate();

// 迭代流中的前运算符（迭代体）
DataStream<Long> iterationBody = iteration.map(new MapFunction<Long, Long>() {
    @Override
    public Long map(Long value) throws Exception {
        return value<0 ? value + 1 : value;
    }
});

// 过滤，凡是小于 0 的元素都被发回前运算符进行迭代计算
DataStream<Long> feedback = iterationBody.filter(new FilterFunction<Long>(){
    @Override
    public boolean filter(Long value) throws Exception {
        return value < 0;
    }
});

// 这里设置 feedback 这个数据流是被反馈的通道，只要是 value<0 的数据都会被重新迭代计算
iteration.closeWith(feedback);

// 迭代流中大于 0 的数挑出来输出
DataStream<Long> output = iterationBody.filter(new FilterFunction<Long>(){
    @Override
    public boolean filter(Long value) throws Exception {
        return value >= 0;
    }
});

output.print();

// 执行
env.execute("flink iterate transformation");
}
```

输出结果如下所示：

```
3
4
5
0
```

默认情况下，带有迭代的 `DataStream` 将永远不会终止，但用户可以使用 `maxWaitTime` 参数设置迭代头的最大等待时间。如果在设置的时间内没有接收到数据，流将终止。

```
public IterativeStream<T> iterate(long maxWaitTimeMillis)
```

3.3.11 project 转换

`Project` 函数从事件流中选择一组属性子集，并且只将选中的元素发送到下一个处理流（相当于 SQL

语句中的投影概念)。下面是进行 project 转换的示例代码:

Scala 代码: 不支持。

Java 代码:

```
package com.xueai8;

import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

public class BatchJob {

    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // project 转换
        Tuple3<Integer, String, Double> user01 = new Tuple3<>(1,"张三",12000.00);
        Tuple3<Integer, String, Double> user02 = new Tuple3<>(2,"李四",22000.00);
        Tuple3<Integer, String, Double> user03 = new Tuple3<>(3,"王老五",18000.00);

        DataStream<Tuple3<Integer, String, Double>> ds = env.fromElements(user01,user02,user03);

        // 选择第 3 列和第 2 列
        DataStream<Tuple3<Integer, String, Double>> ds_select = ds.project(2,1);

        ds_select.print();
    }
}
```

输出结果如下:

```
(张三,12000.0)
(李四,22000.0)
(王老五,18000.0)
```

3.4 数据分区

Flink 应用程序本质上是并行和分布式的。在程序执行期间,一个流被切分为一个或多个流分区,每个运算符有一个或多个运算符任务。运算符任务彼此独立,并在不同的线程中执行,也可能在不同的机器或容器上执行。运算符任务的数量是该运算符的并行度。

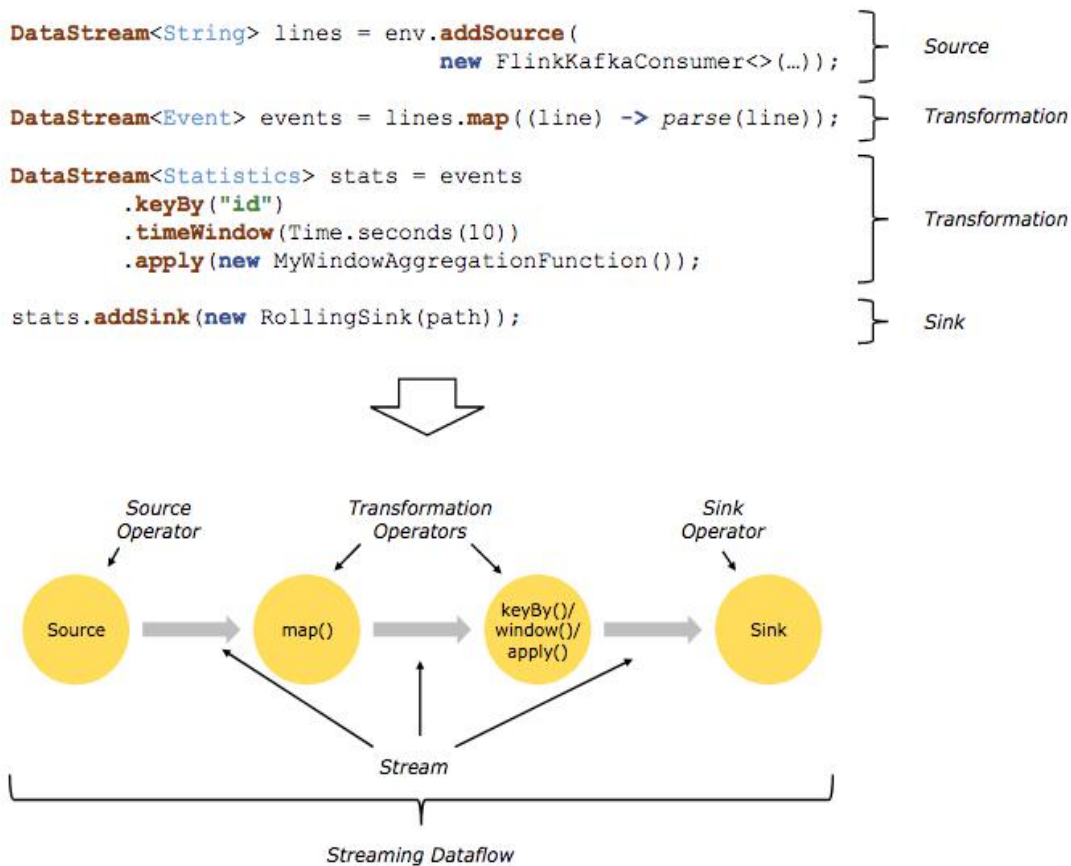
有状态运算符的并行实例集实际上是切分的 key-value 存储。每个并行实例负责处理特定 key 组的事件,这些 key 的状态保存在本地。

3.4.1 流数据分发模式

Flink 程序在执行时被映射到流式数据流,由流和转换运算符组成。每个数据流开始于一个或多个源(source),结束于一个或多个接收器(sinks)。数据流类似于任意有向无环图(DAG),它是一个

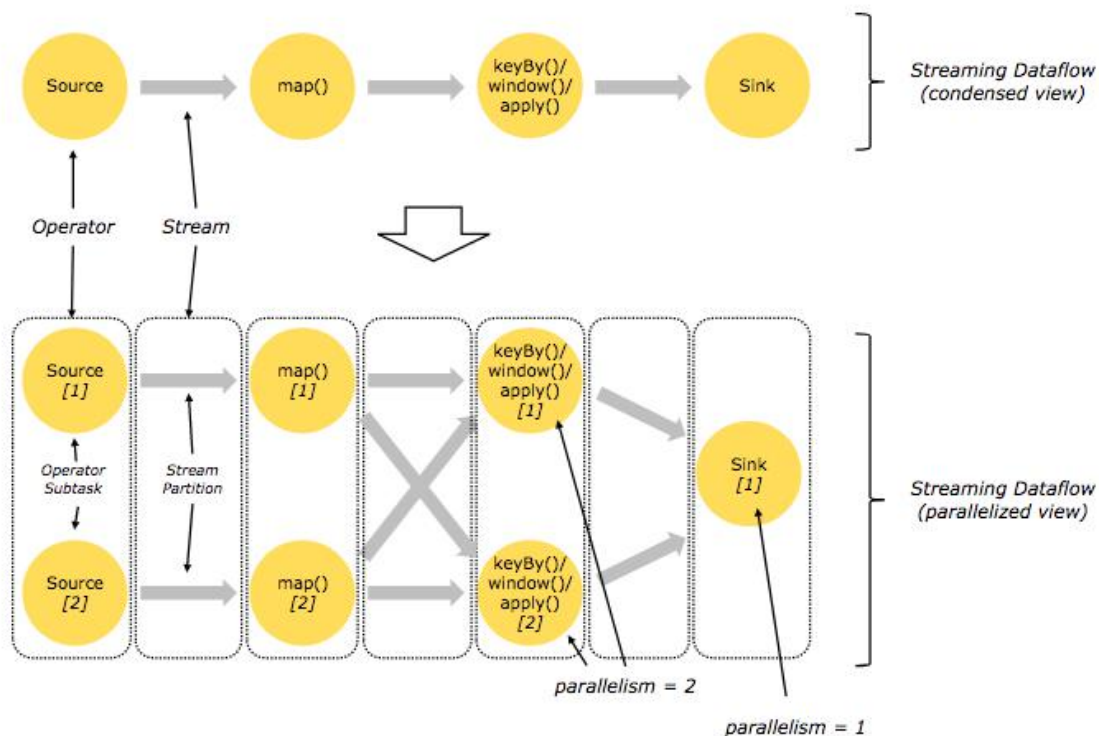
执行计划。

JobClient 将接收的程序转换为对应的数据流的典型过程如下图所示。



通常，程序中的转换与数据流中的运算符之间存在一对一的对应关系。然而，有时一个转换可能包含多个转换运算符。

Flink 数据流默认是并行地分布式地执行，因此实际的转化结果可能更像下面的图所示（其中上半部分为执行简图，下半部分为并行执行视图）。



Flink 流中两个运算符之间的数据分发有两种模式：

- ❑ **one-to-one 流**：一对一分发的模式，例如上图中的 **Source** 运算符和 **map()** 运算符之间。这种方式会保持元素的分区和顺序，也就是说，它保证原来的数据分区和排序不会改变。这意味着 **map()** 运算符的子任务[1]将看到与 **Source** 运算符的子任务[1]生成的元素顺序相同的元素。
- ❑ **redistributing 流**：重新分发的模式，就像上面的 **map()** 和 **keyBy/window** 之间，以及 **keyBy/window** 和 **Sink** 之间。这种方式会打乱流数据原有的划分情况和排序情况。每个运算符子任务根据所选的转换向不同的目标子任务发送数据。例如 **keyBy()**(通过散列 **key** 重新分区)、**broadcast()** 或 **rebalance()**(随机重新分区)。在 **Redistributing** 交换中，元素之间的顺序只保留在每对发送和接收子任务中(例如 **map()** 的子任务[1]和 **keyBy/window** 的子任务[2])。对于 **keyBy** 来说，就是把相同的 **key** 的数据分发到同一个节点上，而对于最终的 **Sink**，由于并行执行，它收到的数据可能不是按照原有的排序情况到达的。

3.4.2 数据分区方法

在 Flink 中，我们可以对转换后的流数据执行物理分区，以进行低级控制（如果需要的话）。Flink 支持以下 8 种分区方法。

分区方法	操作方式
shuffle	随机对数据流进行分区，根据均匀分布随机划分元素。
rebalance	round-robin方式。使用循环分配分区元素的方法，为每个分区创建相等的负载。
rescale	根据上下游运算符的数量，对元素做一个均匀分配。
broadcast	将输出元素被广播到下一个操作(算子)的每个并行实例。
keyBy	按key划分数据流，相同key的元素到一个分区上。
forward	将输出元素被转发到下一个操作的本地子任务。
global	将所有数据都发送到下游 0 号分区中。

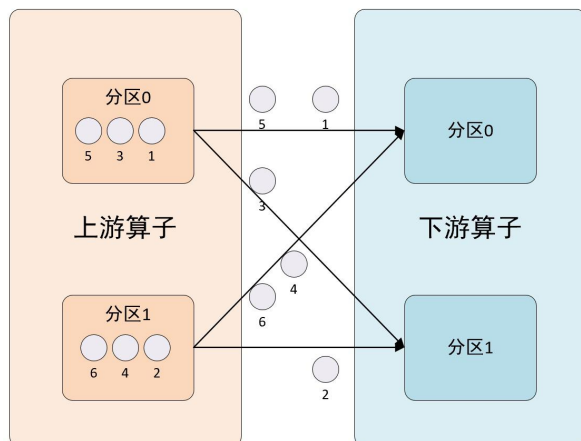
自定义分区

使用用户定义的分区程序（Partitioner）为每个元素选择目标任务。

不同的分区方法代表了 Flink 中不同的数据分区策略，数据的分区策略决定了数据会分发到下游算子的那个分区。下面我们详细了解这 8 种分区方法的原理和应用。

1、shuffle 方法

该方法使用 ShufflePartitioner 分区程序来设置 DataStream 的分区，通过随机选择一个输出通道平均分配数据。该方法会将输出元素均匀随机地打乱到下一个操作（算子），其分区示意图如下所示：



Scala 代码:

```
package com.xueai8.scala.ch03
```

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

```
/**
```

```
 * shuffle 方法随机分区
```

```
 */
```

```
object PartitionShuffle {
```

```
  def main(args: Array[String]) {
```

```
    // 设置流执行环境
```

```
    val env = StreamExecutionEnvironment.getExecutionEnvironment
```

```
    // 从自定义的集合中读取数据
```

```
    val stream = env.fromCollection(List(1,2,3,4,5))
```

```
    // 这里只是为了能够将并行度设置为 2
```

```
    val stream2 = stream
```

```
      .map(v=>{(v%2,v)}) // 偶数 key 为 0，奇数 key 为 1
```

```
      .keyBy(0) // 按奇偶进行分区
```

```
      .map(v=>(v._1,v._2))
```

```
      .setParallelism(2)
```

```
    println(stream2.parallelism) // 查看并行度
```

```
    // 查看随机分区的结果
```

```
    stream2.shuffle.print("shuffle").setParallelism(3)
```



```
// 触发程序执行
env.execute("shuffle 分区示例")
}
}
```

Java 代码:

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.util.Arrays;

/**
 * shuffle 方法随机分区
 */
public class PartitionShuffle {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 从自定义的集合中读取数据
        DataStream<Integer> stream = env.fromCollection(Arrays.asList(1,2,3,4,5));

        // 这里只是为了能够将并行度设置为 2
        DataStream<Tuple2<Integer,Integer>> stream2 = stream
            .map(new MapFunction<Integer, Tuple2<Integer,Integer>>() {
                @Override
                public Tuple2<Integer, Integer> map(Integer input) throws Exception {
                    return new Tuple2<>(input%2, input);
                }
            }) // 偶数 key 为 0, 奇数 key 为 1
            .keyBy(t -> t.f0) // 按奇偶进行分区
            .map(new MapFunction<Tuple2<Integer,Integer>, Tuple2<Integer,Integer>>() {
                @Override
                public Tuple2<Integer, Integer> map(Tuple2<Integer, Integer> t) throws Exception {
                    return new Tuple2<>(t.f0, t.f1);
                }
            })
            .setParallelism(2);
        System.out.println(stream2.getParallelism()); // 查看并行度

        // 查看随机分区的结果
        stream2.shuffle().print("shuffle").setParallelism(3);

        // 触发程序执行
        env.execute("shuffle 分区示例");
    }
}
```

```
}

```

执行以上代码，输出结果如下所示：

```
2

```

```
shuffle:1> (0,2)

```

```
shuffle:3> (1,5)

```

```
shuffle:2> (1,3)

```

```
shuffle:1> (1,1)

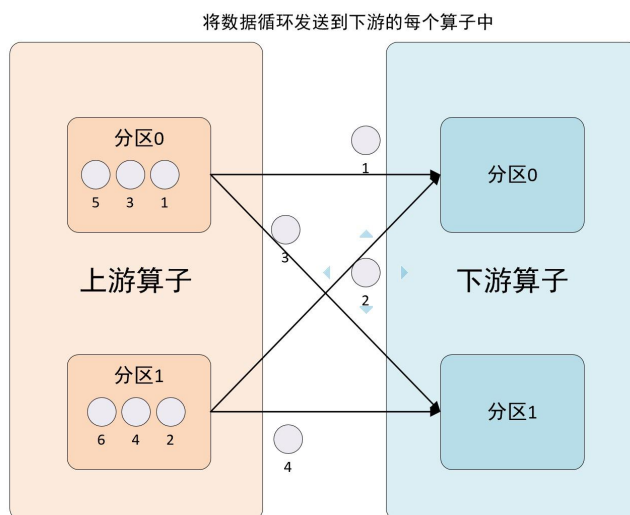
```

```
shuffle:3> (0,4)

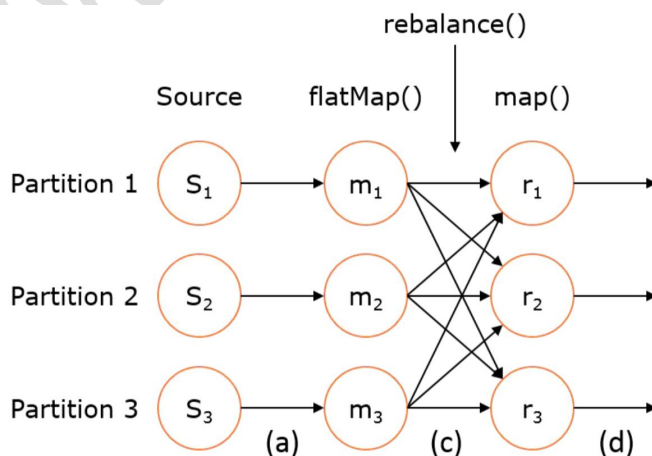
```

2、rebalance 方法

该方法使用 RebalancePartitioner 分区程序来设置 DataStream 的分区，使用循环通过输出通道平均分配数据。该方法会将输出元素以轮循方式均匀地分布到下一个操作（算子）的实例中，其分区示意图如下所示：



这种类型的分区有助于均匀地分布数据。它使用循环分配分区元素的方法，为每个分区创建相等的负载。这种类型的分区对于存在数据倾斜的情况下的性能优化非常有用。



Scala 代码:

```
package com.xueai8.scala.ch03;

```

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

/**
 * 该方法使用 RebalancePartitioner 分区程序来设置 DataStream 的分区，使用循环通过输出通道平均分配数据。
 * 该方法会将输出元素以轮循方式均匀地分布到下一个操作（算子）的实例中。
 */
object RebalancePartitioner extends App{

    // 创建执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 从自定义的集合中读取数据
    val stream = env.fromCollection(List(1,2,3,4,5,6))

    // 直接打印数据
    stream.rebalance().print("rebalance").setParallelism(2)

    env.execute("rebalance 分区示例")
}
```

Java 代码：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.util.Arrays;

/**
 * 该方法使用 RebalancePartitioner 分区程序来设置 DataStream 的分区，使用循环通过输出通道平均分配数据。
 * 该方法会将输出元素以轮循方式均匀地分布到下一个操作（算子）的实例中。
 */
public class PartitionRebalance {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 从自定义的集合中读取数据
        DataStream<Integer> stream = env.fromCollection(Arrays.asList(1,2,3,4,5,6));

        // 直接打印数据
        stream.rebalance().print("rebalance").setParallelism(2);

        // 触发流程执行
        env.execute("rebalance 分区示例");
    }
}
```

执行以上代码，输出结果如下所示：

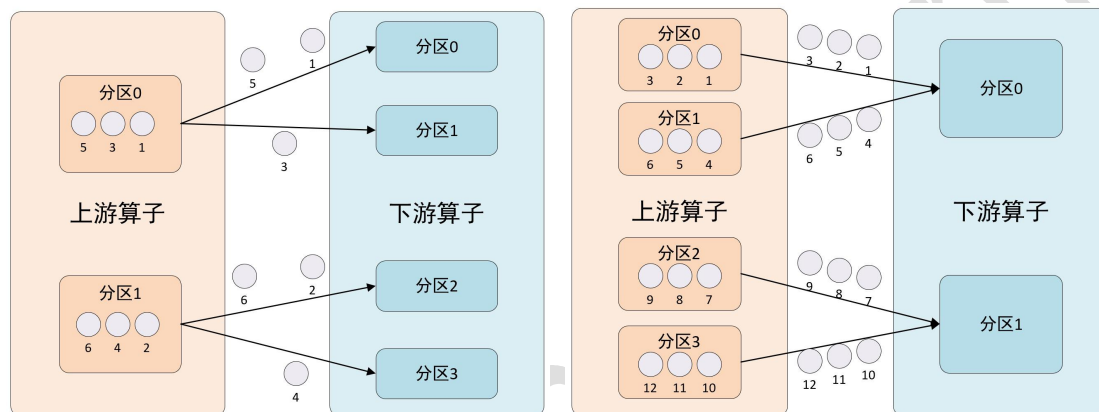
```

rebalance:2> 1
rebalance:1> 2
rebalance:2> 3
rebalance:1> 4
rebalance:2> 5
rebalance:1> 6

```

3、rescale 方法

该方法使用 `RescalePartitioner` 分区程序来设置 `DataStream` 的分区，使用循环通过输出通道平均分配数据。该方法会将输出元素以轮循方式均匀地分布到下一个操作（算子）的实例子集，其分区示意图如下所示：



在这种分区方法中，Flink 循环将元素划分为下游操作的子集。上游操作向其发送元素的下游操作子集取决于上游和下游操作的并行度。例如，如果上游操作的并行度为 2，而下游操作的并行度为 4，那么一个上游操作将把元素分配给两个下游操作，而另一个上游操作将分配给另外两个下游操作。另一方面，如果下游操作的并行度为 2，而上游操作的并行度为 4，那么两个上游操作将分配给一个下游操作，而另外两个上游操作将分配给另一个下游操作。

在上下游算子的并行度不是彼此的倍数的情况下，一个或几个下游操作与上游操作的输入数量不同。

如果我们希望有这样的管道，例如，从一个数据源的每个并行实例分散到几个 mapper 的一个子集来分配负载，但又不希望使用 `rebalance()` 导致完全的重新平衡，那么这个方法是非常有用的。根据其他配置值，比如 `TaskManager` 的槽位数，这将只需要本地数据传输，而不需要通过网络传输数据。

Scala 代码:

```
package com.xueai8.scala.ch03
```

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

```
/**
```

```
 * 该方法使用 RescalePartitioner 分区程序来设置 DataStream 的分区，使用循环通过输出通道平均分配数据。
```

```
 * 该方法会将输出元素以轮循方式均匀地分布到下一个操作（算子）的实例子集。
```

```
 */
```

```
object PartitionRescale {
```

```
  def main(args: Array[String]) {
```

```
// 设置流执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 从自定义的集合中读取数据
val stream = env.fromCollection(List(1,2,3,4,5,6,7,8))
stream.print("before rescale")

// 直接打印数据
stream.rescale().print("rescale").setParallelism(2)

// 触发流程序执行
env.execute("rescale 分区示例")
}
```

Java 代码:

```
package com.xueai8.java.ch03;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.util.Arrays;

/**
 * 该方法使用 RescalePartitioner 分区程序来设置 DataStream 的分区，使用循环通过输出通道平均分配数据。
 * 该方法会将输出元素以轮循方式均匀地分布到下一个操作（算子）的实例子集。
 */
public class PartitionRescale {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 从自定义的集合中读取数据
        DataStream<Integer> stream = env.fromCollection(Arrays.asList(1,2,3,4,5,6,7,8));
        stream.print("before rescale");

        // 直接打印数据
        stream.rescale().print("rescale").setParallelism(2);

        // 触发流程序执行
        env.execute("rescale 分区示例");
    }
}
```

执行以上代码，输出结果如下所示：

```
before rescale:7> 7
before rescale:3> 3
before rescale:1> 1
before rescale:5> 5
before rescale:8> 8
```

```

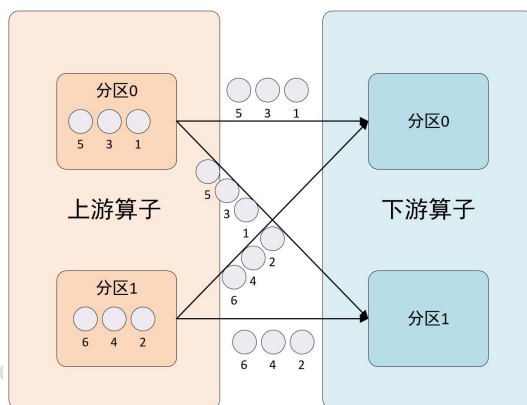
before rescale:4> 4
before rescale:2> 2
before rescale:6> 6
rescale:1> 1
rescale:1> 3
rescale:2> 2
rescale:2> 4
rescale:1> 5
rescale:2> 6
rescale:1> 7
rescale:2> 8

```

简而言之，rescale 用于跨操作(算子)分发数据，对数据子集执行转换并将它们组合在一起。这种重新平衡只发生在单个节点上，因此不需要跨网络进行任何数据传输。

4、broadcast 方法

该方法使用 BroadcastPartitioner 分区程序来设置 DataStream 的分区，通过选择所有输出通道将所有记录分发到每个分区。该方法会将输出元素被广播到下一个操作(算子)的每个并行实例，其分区示意图如下所示：



Scala 代码:

```
package com.xueai8.scala.ch03
```

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

```
/**
```

```

 * 该方法使用 BroadcastPartitioner 分区程序来设置 DataStream 的分区，通过选择所有输出通道将所有记录
 * 分发到每个分区。该方法会将输出元素被广播到下一个操作(算子)的每个并行实例。

```

```
*/
```

```
object PartitionBroadcast {
```

```
  def main(args: Array[String]) {
```

```
    // 设置流执行环境
```

```
    val env = StreamExecutionEnvironment.getExecutionEnvironment
```

```
    // 从自定义的集合中读取数据
```

```
    val stream = env.fromCollection(List(1,2,3,4,5))
```

```
// 直接打印数据
stream.broadcast.print("broadcast").setParallelism(2)

// 触发流程序执行
env.execute("broadcast 分区示例")
}
}
```

Java 代码:

```
package com.xueai8.java.ch03;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.util.Arrays;

/**
 * 该方法使用 BroadcastPartitioner 分区程序来设置 DataStream 的分区，通过选择所有输出通道将所有记录
 * 分发到每个分区。该方法会将输出元素被广播到下一个操作(算子)的每个并行实例。
 */
public class PartitionBroadcast {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 从自定义的集合中读取数据
        DataStream<Integer> stream = env.fromCollection(Arrays.asList(1,2,3,4,5));

        // 直接打印数据
        stream.broadcast().print("broadcast").setParallelism(2);

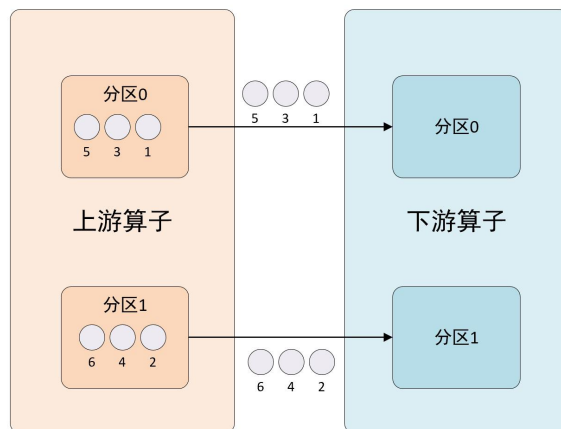
        // 触发流程序执行
        env.execute("broadcast 分区示例");
    }
}
```

执行以上代码，输出结果如下所示：

```
broadcast:1> 1
broadcast:2> 1
broadcast:1> 2
broadcast:2> 2
broadcast:1> 3
broadcast:2> 3
broadcast:1> 4
broadcast:2> 4
broadcast:1> 5
broadcast:2> 5
```

5、forward 方法

该方法使用 ForwardPartitioner 分区程序来设置 DataStream 的分区，仅将元素转发到本地运行的下游操作（算子）。该方法会将输出元素被转发到下一个操作（算子）的本地子任务，其分区示意图如下所示：



在上下游的算子没有指定分区器的情况下，如果上下游的算子并行度一致，则使用 ForwardPartitioner，否则使用 RebalancePartitioner。对于 ForwardPartitioner，必须保证上下游算子并行度一致，即上有算子与下游算子是 1 对 1 的关系，否则会抛出异常。

Scala 代码：

```
package com.xueai8.scala.ch03

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

/**
 * 该方法使用 ForwardPartitioner 分区程序来设置 DataStream 的分区，仅将元素转发到本地运行的
 * 下游操作（算子）。该方法会将输出元素被转发到下一个操作（算子）的本地子任务。
 */
object PartitionForward {
  def main(args: Array[String]) {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 从自定义的集合中读取数据
    val stream = env.fromCollection(List(1,2,3,4,5))

    // 直接打印数据
    stream.map(v=>{v * v}).setParallelism(2).forward.print().setParallelism(2)

    // 触发程序执行
    env.execute("forward 分区示例")
  }
}
```

Java 代码：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
```



```
import java.util.Arrays;

/**
 * 该方法使用 ForwardPartitioner 分区程序来设置 DataStream 的分区，仅将元素转发到本地运行的下游操作（算子）。
 * 该方法会将输出元素被转发到下一个操作（算子）的本地子任务。
 */
public class PartitionForward {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 从自定义的集合中读取数据
        DataStream<Integer> stream = env.fromCollection(Arrays.asList(1,2,3,4,5));

        // 直接打印数据
        stream.map(
            new MapFunction<Integer, Integer>() {
                @Override
                public Integer map(Integer input) throws Exception {
                    return input * input;
                }
            })
            .setParallelism(2)
            .forward()
            .print("forward")
            .setParallelism(2);

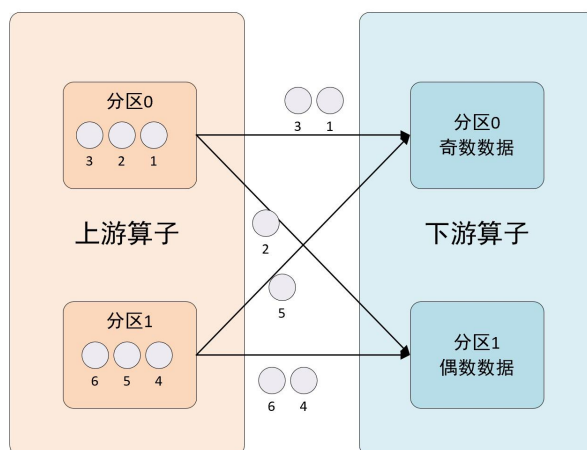
        // 触发流程序执行
        env.execute("forward 分区示例");
    }
}
```

执行以上代码，输出结果如下所示：

```
forward:1> 1
forward:2> 4
forward:1> 9
forward:2> 16
forward:1> 25
```

6、keyBy 方法

该方法使用 KeyGroupStreamPartitioner 分区程序来设置 DataStream 的分区，根据 key 的分组索引选择目标通道，将输出元素发送到相对应的下游分区。该方法的分区示意图如下所示：



Scala 代码:

```
package com.xueai8.scala.ch03
```

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

```
/**
```

```
 * 该方法使用 KeyGroupStreamPartitioner 分区程序来设置 DataStream 的分区，
 * 根据 key 的分组索引选择目标通道，将输出元素发送到相对应的下游分区。
 * 该方法会创建一个新的 KeyedStream，使用提供的 key 来划分其操作符（算子）状态。
 */
```

```
object PartitionKeyBy {
```

```
  def main(args: Array[String]) {
```

```
    // 设置流执行环境
```

```
    val env = StreamExecutionEnvironment.getExecutionEnvironment
```

```
    // 从自定义的集合中读取数据
```

```
    val stream = env.fromCollection(List(1,2,3,4,5,6))
```

```
    // 先转换为(k,v)对，再执行 keyBy，然后打印数据
```

```
    val stream2 = stream.map(v => {(v%3,v)})
```

```
    stream2.setParallelism(2).keyBy(0).print("key")
```

```
    // 触发流程序执行
```

```
    env.execute("keyBy 分区示例")
```

```
  }
```

```
}
```

Java 代码:

```
package com.xueai8.java.ch03;
```

```
import org.apache.flink.api.common.functions.MapFunction;
```

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

```
import org.apache.flink.streaming.api.datastream.DataStream;
```

```
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
```

```
import java.util.Arrays;
```

```
/**
 * 该方法使用 KeyGroupStreamPartitioner 分区程序来设置 DataStream 的分区，
 * 根据 key 的分组索引选择目标通道，将输出元素发送到相对应的下游分区。
 * 该方法会创建一个新的 KeyedStream，使用提供的 key 来划分其操作符（算子）状态。
 */
public class PartitionKeyBy {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 从自定义的集合中读取数据
        DataStream<Integer> stream = env.fromCollection(Arrays.asList(1,2,3,4,5,6));

        // 直接打印数据
        DataStream<Tuple2<Integer, Integer>> stream2 = stream.map(
            new MapFunction<Integer, Tuple2<Integer, Integer>>() {
                @Override
                public Tuple2<Integer, Integer> map(Integer input) throws Exception {
                    return new Tuple2<>(input % 3, input);
                }
            })
            .setParallelism(2);
        stream2.keyBy(t -> t.f0).print("key");

        // 触发流程序执行
        env.execute("keyBy 分区示例");
    }
}
```

该方法会创建一个新的 KeyedStream，使用提供的 key 来划分其操作符（算子）状态。

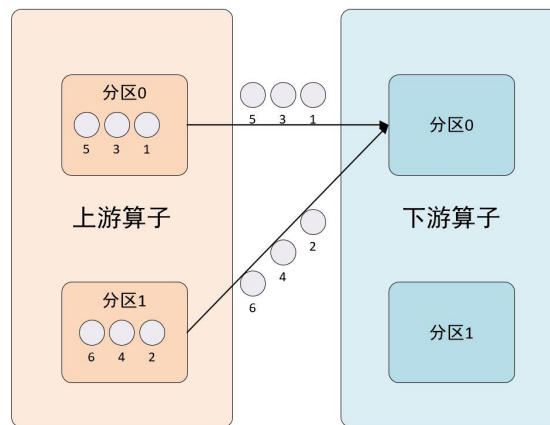
执行以上代码，输出结果如下所示：

```
key:8> (2,2)
key:6> (1,1)
key:6> (0,3)
key:8> (2,5)
key:6> (1,4)
key:6> (0,6)
```

7、global 方法

该方法使用 GlobalPartitioner 分区程序来设置 DataStream 的分区，以便将输出值都转到下一个处理操作符(算子)的第一个实例。使用此设置时要小心，因为它可能会在应用程序中造成严重的性能瓶颈。

该方法的分区示意图如下所示：



Scala 代码:

```
package com.xueai8.scala.ch03
```

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

```
/**
```

```
 * 该方法使用 GlobalPartitioner 分区程序来设置 DataStream 的分区，
 * 以便将输出值都转到下一个处理操作符(算子)的第一个实例。
 * 使用此设置时要小心，因为它可能会在应用程序中造成严重的性能瓶颈。
 */
```

```
object PartitionGlobal {
```

```
  def main(args: Array[String]) {
```

```
    // 设置流执行环境
```

```
    val env = StreamExecutionEnvironment.getExecutionEnvironment
```

```
    // 从自定义的集合中读取数据
```

```
    val stream = env.fromCollection(List(1,2,3,4,5))
```

```
    // 直接打印数据
```

```
    stream.print()
```

```
    // 使用 GlobalPartitioner 之后打印数据
```

```
    stream.global.print("global")
```

```
    // 触发流程序执行
```

```
    env.execute("global 分区示例")
```

```
  }
```

```
}
```

Java 代码:

```
package com.xueai8.java.ch03;
```

```
import org.apache.flink.streaming.api.datastream.DataStream;
```

```
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
```

```
import java.util.Arrays;
```

```
/**
```

```
* 该方法使用 GlobalPartitioner 分区程序来设置 DataStream 的分区，
* 以便将输出值都转到下一个处理操作符(算子)的第一个实例。
* 使用此设置时要小心，因为它可能会在应用程序中造成严重的性能瓶颈。
*/
public class PartitionGlobal {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 从自定义的集合中读取数据
        DataStream<Integer> stream = env.fromCollection(Arrays.asList(1,2,3,4,5));

        // 直接打印数据
        stream.print();

        // 使用 GlobalPartitioner 之后打印数据
        stream.global().print("global");

        // 触发流程序执行
        env.execute("global 分区示例");
    }
}
```

执行以上代码，输出结果如下所示：

```
global:1> 2
global:1> 3
global:1> 4
global:1> 5
```

8、自定义分区程序

通过继承 `org.apache.flink.api.common.functions.Partitioner`，用户也可以自定义分区程序为每个元素选择目标分区。下面我们提供一个分区器的自定义实现，将数据流中的整数按奇偶性分发到不同的子分区中。

Scala 代码：

```
package com.xueai8.scala.ch03

import org.apache.flink.api.common.functions.Partitioner
import org.apache.flink.streaming.api.scala._

/**
 * 通过继承 org.apache.flink.api.common.functions.Partitioner，
 * 用户也可以自定义分区程序为每个元素选择目标分区。
 * 下面我们提供一个分区器的自定义实现，将数据流中的整数按奇偶性分发到不同的子分区中。
 */
object PartitionUDF {

    // 自定义分区程序
    object CustomPartitioner extends Partitioner[String]{
```

```
// 重写 partition 方法
override def partition(key: String, numPartitions: Int): Int = {
  // 根据 key 值的 奇偶 返回不同的分区 id
  key.toInt % 2
}
}

def main(args: Array[String]) {
  // 设置流执行环境
  val env = StreamExecutionEnvironment.getExecutionEnvironment

  // 从自定义的集合中读取数据
  val stream = env.fromCollection(List("1","2","3","4","5"))

  val stream2 = stream.map(value=>{((value.toInt%2).toString,value)})
  stream2.partitionCustom(CustomPartitioner,0).print().setParallelism(2)

  // 触发流程序执行
  env.execute("自定义分区示例")
}
```

Java 代码:

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.functions.Partitioner;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.util.Arrays;

/**
 * 通过继承 org.apache.flink.api.common.functions.Partitioner,
 * 用户也可以自定义分区程序为每个元素选择目标分区。
 * 下面我们提供一个分区器的自定义实现，将数据流中的整数按奇偶性分发到不同的子分区中。
 */
public class PartitionUDF {

  public static void main(String[] args) throws Exception {
    // 设置流执行环境
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

    // 从自定义的集合中读取数据
    DataStream<String> stream = env.fromCollection(Arrays.asList("1","2","3","4","5"));

    // 先转换为<k,v>
    DataStream<Tuple2<String,String>> stream2 = stream.map(
      new MapFunction<String, Tuple2<String,String>>() {
        @Override
```

```
        public Tuple2<String,String> map(String s) throws Exception {
            return new Tuple2<>(String.valueOf(Integer.parseInt(s)%2), s);
        }
    };

    // 应用自定义分区器
    stream2.partitionCustom(new Partitioner<String>() {
        @Override
        public int partition(String key, int i) {
            return Integer.parseInt(key) % 2;
        }
    }, t -> t.f0).print().setParallelism(2);

    // 触发流程序执行
    env.execute("自定义分区示例");
}
```

执行以上代码，输出结果如下所示：

```
2> (1,1)
1> (0,2)
2> (1,5)
2> (1,3)
1> (0,4)
```

在编写自定义分区程序时，需要确保实现了有效的哈希函数。

3.4.3 数据分区示例

默认情况下，Flink 应用程序是使用 **key** 的哈希分区或随机分区。但在有的情况下，我们需要自定义分区规则，这就需要自己来定义分区器（分区程序）。

【示例】（Scala 实现）使用自定义分区程序，按年龄对数据流元素划分分区。

1、在 IntelliJ IDEA 中创建一个 Flink 项目，使用 `flink-quickstart-scala` 项目模板。（Flink 项目创建过程，请参见 2.2 节）

2、设置依赖。在 `pom.xml` 中添加如下依赖（根据项目模板创建的话，这个依赖会自动添加，此步可省略）：

Scala Maven 依赖：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-scala_2.12</artifactId>
    <version>1.13.2</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-scala_2.12</artifactId>
    <version>1.13.2</version>
    <scope>provided</scope>
</dependency>
```

3、创建一个 case class 类，用来表示流中的数据类型。代码如下：

```
package com.xueai8.scala.ch03

import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

/**
 * 自定义分区
 */
object PartitionDemo1 {

    // 定义 case class 类，表示流数据类型
    case class Person(name:String, age:Int)

    def main(args: Array[String]): Unit = {
        // 设置批处理执行环境
        val env = StreamExecutionEnvironment.getExecutionEnvironment

        // 触发流程序开始执行
        env.execute("stream demo")
    }
}
```

4、自定义分区类，实现了 Partitioner 接口，按年龄（age 字段）划分分区。这里我们按年龄把流数据分为三个分区：20 岁以下的、20~30 岁以及 30 岁以上的。代码如下：

```
package com.xueai8.scala.ch03

import org.apache.flink.api.common.functions.Partitioner
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

/**
 * 自定义分区
 */
object PartitionDemo1 {

    // 定义 case class 类，表示流数据类型
    case class Person(name:String, age:Int)

    // 自定义分区器，以年龄为 key
    class AgePartitioner extends Partitioner[Int] {
        // 重写 partition 方法
        override def partition(key: Int, i: Int): Int = {
            key match {
                case age if age<20 => 0
                case age if age>30 => 2
                case _ => 1
            }
        }
    }

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



```
// 设置批处理执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 触发流程序开始执行
env.execute("stream demo")
}
}
```

5、测试自定义分区逻辑。编辑流处理代码如下：

```
package com.xueai8.scala.ch03

import org.apache.flink.api.common.functions.Partitioner
import org.apache.flink.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

/**
 * 自定义分区
 */
object PartitionDemo1 {

    // 定义 case class 类，表示流数据类型
    case class Person(name:String, age:Int)

    // 自定义分区器，以年龄为 key
    class AgePartitioner extends Partitioner[Int] {
        // 重写 partition 方法
        override def partition(k: Int, i: Int): Int = {
            k match {
                case age if age<20 => 0
                case age if age>30 => 2
                case _ => 1
            }
        }
    }

    def main(args: Array[String]): Unit = {
        // 设置批处理执行环境
        val env = StreamExecutionEnvironment.getExecutionEnvironment

        // 读取数据源，构造 DataStream
        val peoples = List(
            Person("张三", 21),
            Person("李四", 16),
            Person("王老五", 35),
            Person("张三 2", 22),
            Person("李四 2", 17),
            Person("王老五 2", 36)
        )
        val personDS = env.fromCollection(peoples)

        // 应用自定义分区器，按年龄字段进行分组
    }
}
```

```
// 注：通过字段位置指定 key 只对元组数据类型有效
val adults = personDS.partitionCustom(new AgePartitioner, "age")

// 将结果输出到控制台
adults.print

// 触发流程序开始执行
env.execute("stream demo")
}
```

6、执行以上程序，输出结果如下：

分区之前分区数：8

.....

2> Person(张三,21)

1> Person(李四,16)

3> Person(王老五,35)

1> Person(李四 2,17)

2> Person(张三 2,22)

3> Person(王老五 2,36)

从上面的输出结果可以看到，数据在三个分区中分别并行地被处理。

注意：输出结果前面的整数是分区号。

对于简单的自定义分区器，也可以使用匿名内部类，以简化代码。如下所示：

```
package com.xueai8.ch03

import org.apache.flink.api.common.functions.Partitioner
import org.apache.flink.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

/**
 * 自定义分区
 * 对于简单的自定义分区器，也可以使用匿名内部类，以简化代码。
 */
object PartitionDemo2 {

  // 定义 case class 类，表示流数据类型
  case class Person(name:String, age:Int)

  def main(args: Array[String]): Unit = {
    // 设置批处理执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 读取数据源，构造 DataStream
    val peoples = List(
      Person("张三", 21),
      Person("李四", 16),
      Person("王老五", 35),
      Person("张三 2", 22),
      Person("李四 2", 17),
      Person("王老五 2", 36)
    )
  }
}
```

```
)  
val personDS = env.fromCollection(peoples)  
  
// 应用自定义分区器，按年龄字段进行分组  
// 注：通过字段位置指定 key 只对元组数据类型有效  
val adults = personDS.partitionCustom(new Partitioner[Int] {  
    // 重写 partition 方法  
    override def partition(k: Int, i: Int): Int = {  
        k match {  
            case age if age < 20 => 0  
            case age if age > 30 => 2  
            case _ => 1  
        }  
    }  
}, "age")    // 第二个分组字段参数，也可以使用 _age 或 person => person.age  
  
// 将结果输出到控制台  
adults.print  
  
// 触发流程序开始执行  
env.execute("stream demo")  
}
```

【示例】（Java 实现）使用自定义分区程序，按年龄对数据流元素划分分区。

1、在 IntelliJ IDEA 中创建一个 Flink 项目，使用 flink-quickstart-java 项目模板。（Flink 项目创建过程，请参见 2.2 节）

2、设置依赖。在 pom.xml 中添加如下依赖（根据项目模板创建的话，这个依赖会自动添加，此步可省略）：

Java Maven 依赖：

```
<dependency>  
    <groupId>org.apache.flink</groupId>  
    <artifactId>flink-java</artifactId>  
    <version>1.13.2</version>  
    <scope>provided</scope>  
</dependency>  
<dependency>  
    <groupId>org.apache.flink</groupId>  
    <artifactId>flink-streaming-java_2.12</artifactId>  
    <version>1.13.2</version>  
    <scope>provided</scope>  
</dependency>
```

3、创建一个 POJO 类，用来表示流中的数据。代码如下：

```
package com.xueai8.java.ch03;  
  
import org.apache.flink.api.common.functions.Partitioner;  
import org.apache.flink.streaming.api.datastream.DataStream;  
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
```

```
public class PartitionDemo1 {  
  
    // POJO 类，表示人员信息实体  
    public static class Person {  
        public String name;           // 存储姓名  
        public Integer age;           // 存储年龄  
  
        // 空构造器  
        public Person() {}  
  
        // 构造器，初始化属性  
        public Person(String name, Integer age) {  
            this.name = name;  
            this.age = age;  
        }  
  
        // 用于调试时输出信息  
        public String toString() {  
            return this.name.toString() + ": age " + this.age.toString();  
        }  
    }  
  
    public static void main(String[] args) throws Exception {  
        // 设置流执行环境  
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  
  
        // 触发流程序开始执行  
        env.execute("stream demo");  
    }  
}
```

4、自定义分区类，实现了 `Partitioner` 接口，按年龄（age 字段）划分分区。这里我们按年龄把流数据分为三个分区：20 岁以下的、20~30 岁以及 30 岁以上的。代码如下：

```
package com.xueai8.java.ch03;  
  
import org.apache.flink.api.common.functions.Partitioner;  
import org.apache.flink.streaming.api.datastream.DataStream;  
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
  
public class PartitionDemo1 {  
  
    // POJO 类，表示人员信息实体  
    public static class Person {  
        public String name;           // 存储姓名  
        public Integer age;           // 存储年龄  
  
        // 空构造器  
        public Person() {}  
  
        // 构造器，初始化属性
```

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

// 用于调试时输出信息
public String toString() {
    return this.name.toString() + ": age " + this.age.toString();
};
}

// 自定义分区器，以年龄为 key
public static class AgePartitioner implements Partitioner<Integer> {
    @Override
    public int partition(Integer key, int numPartitions) {
        if(key < 20){
            return 0;
        }else if(key > 30){
            return 2;
        }else{
            return 1;
        }
    }
}

public static void main(String[] args) throws Exception {
    // 设置流执行环境
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

    // 触发流程序开始执行
    env.execute("stream demo");
}
}
```

5、测试自定义分区逻辑。编辑流处理代码如下：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.Partitioner;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

public class PartitionDemo1 {

    // POJO 类，表示人员信息实体
    public static class Person {
        public String name;           // 存储姓名
        public Integer age;           // 存储年龄

        // 空构造器
        public Person() {};
    }
}
```

```
// 构造器，初始化属性
public Person(String name, Integer age) {
    this.name = name;
    this.age = age;
};

// 用于调试时输出信息
public String toString() {
    return this.name.toString() + ": age " + this.age.toString();
};
}

// 自定义分区器，以年龄为 key
public static class AgePartitioner implements Partitioner<Integer> {
    @Override
    public int partition(Integer key, int numPartitions) {
        if(key < 20){
            return 0;
        }else if(key > 30){
            return 2;
        }else{
            return 1;
        }
    }
}

public static void main(String[] args) throws Exception {
    // 设置流执行环境
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

    // 读取数据源，构造 DataStream
    DataStream<Person> personDS = env.fromElements(
        new Person("张三", 21),
        new Person("李四", 16),
        new Person("王老五", 35),
        new Person("张三 2", 22),
        new Person("李四 2", 17),
        new Person("王老五 2", 36));

    // 应用自定义分区器，按年龄字段进行分组
    // 注：通过字段位置指定 key 只对元组数据类型有效
    DataStream<Person> adults = personDS.partitionCustom(new AgePartitioner(), p -> p.age);

    // 将结果输出到控制台
    adults.print();

    // 触发流程序开始执行
    env.execute("stream demo");
}
```

6、执行以上程序，输出结果如下：

```
1> 李四: age 16
2> 张三: age 21
3> 王老五: age 35
1> 李四 2: age 17
3> 王老五 2: age 36
2> 张三 2: age 22
```

从上面的输出结果可以看到，数据在三个分区中分别并行地被处理。

注意：输出结果前面的整数是分区号。

对于简单的自定义分区器，也可以直接使用匿名内部类，以简化代码。如下所示：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.Partitioner;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * 自定义分区
 * 对于简单的自定义分区器，也可以直接使用匿名内部类，以简化代码。
 */
public class PartitionDemo2 {

    // POJO 类，表示人员信息实体
    public static class Person {
        public String name;           // 存储姓名
        public Integer age;           // 存储年龄

        // 空构造器
        public Person() {}

        // 构造器，初始化属性
        public Person(String name, Integer age) {
            this.name = name;
            this.age = age;
        };

        // 用于调试时输出信息
        public String toString() {
            return this.name.toString() + ": age " + this.age.toString();
        };
    }

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 读取数据源，构造 DataStream
        DataStream<Person> personDS = env.fromElements(
```

```

        new Person("张三", 21),
        new Person("李四", 16),
        new Person("王老五", 35),
        new Person("张三 2", 22),
        new Person("李四 2", 17),
        new Person("王老五 2", 36));

// 应用自定义分区器，按年龄字段进行分组
// 注：通过字段位置指定 key 只对元组数据类型有效
DataStream<Person> adults = personDS.partitionCustom(new Partitioner<Integer>(){
    @Override
    public int partition(Integer key, int i) {
        if(key < 20){
            return 0;
        }else if(key>=20 && key<30){
            return 1;
        }else{
            return 2;
        }
    }
}, person -> person.age);

// 将结果输出到控制台
adults.print();

// 触发程序开始执行
env.execute("stream demo");
}
}

```

在上面的示例中，指定按流数据元素 `Person` 的字段 “age” 来作为 key，即将 “age” 字段作为 `partitionCustom` 方法的第二个参数。如下所示：

```
val adults = personDS.partitionCustom(new AgePartitioner, person -> person.age)
```

实际上，这第二个参数，是通过 key selector 来指定，这在流数据类型是复杂结构时非常有用。下面是对上一示例的改造。

Scala 代码：

```

package com.xueai8.ch03

import org.apache.flink.api.common.functions.Partitioner
import org.apache.flink.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

/**
 * 自定义分区，使用 key selector
 */
object PartitionDemo3 {

    // 定义 case class 类，表示流数据类型
    case class Person(name:String, age:Int)

```



```
def main(args: Array[String]): Unit = {  
    // 设置批处理执行环境  
    val env = StreamExecutionEnvironment.getExecutionEnvironment  
  
    // 读取数据源，构造 DataStream  
    val peoples = List(  
        Person("张三", 21),  
        Person("李四", 16),  
        Person("王老五", 35),  
        Person("张三 2", 22),  
        Person("李四 2", 17),  
        Person("王老五 2", 36)  
    )  
    val personDS = env.fromCollection(peoples)  
  
    // 应用自定义分区器，按 person.age 字段进行分组  
    val adults = personDS.partitionCustom(  
        new Partitioner[Int] {  
            override def partition(k: Int, i: Int): Int = {  
                k match {  
                    case age if age < 20 => 0  
                    case age if age > 30 => 2  
                    case _ => 1  
                }  
            }  
        },  
        person => person.age  
    )  
  
    // 将结果输出到控制台  
    adults.print  
  
    // 触发流程序开始执行  
    env.execute("stream demo")  
}
```

Java 代码:

```
package com.xueai8.ch03;  
  
import org.apache.flink.api.common.functions.Partitioner;  
import org.apache.flink.api.java.functions.KeySelector;  
import org.apache.flink.streaming.api.datastream.DataStream;  
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  
  
/**  
 * Created by www.xueai8.com  
 *  
 * 自定义分区  
 * 对于简单的自定义分区器，也可以使用匿名内部类，以简化代码。 */
```

```
*/
public class PartitionDemo3 {

    // POJO 类，表示人员信息实体
    public static class Person {
        public String name;           // 存储姓名
        public Integer age;           // 存储年龄

        // 空构造器
        public Person() {};

        // 构造器，初始化属性
        public Person(String name, Integer age) {
            this.name = name;
            this.age = age;
        };

        // 用于调试时输出信息
        public String toString() {
            return this.name.toString() + ": age " + this.age.toString();
        };
    }

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        System.out.println("分区之前分区数: " + env.getParallelism());

        // 读取数据源，构造 DataStream
        DataStream<Person> personDS = env.fromElements(
            new Person("张三", 21),
            new Person("李四", 16),
            new Person("王老五", 35),
            new Person("张三 2", 22),
            new Person("李四 2", 17),
            new Person("王老五 2", 36));

        // 应用自定义分区器，按年龄字段进行分组
        // 注：通过字段位置指定 key 只对元组数据类型有效
        DataStream<Person> adults = personDS.partitionCustom(new Partitioner<Integer>() {
            @Override
            public int partition(Integer key, int i) {
                if (key < 20) {
                    return 0;
                } else if (key >= 20 && key < 30) {
                    return 1;
                } else {
                    return 2;
                }
            }
        });
    }
}
```

```

    }
    }, new KeySelector<Person, Integer>() {
        @Override
        public Integer getKey(Person person) throws Exception {
            return person.age;
        }
    });

    // 将结果输出到控制台
    adults.print();

    // 触发程序开始执行
    env.execute("stream demo");
}
}

```

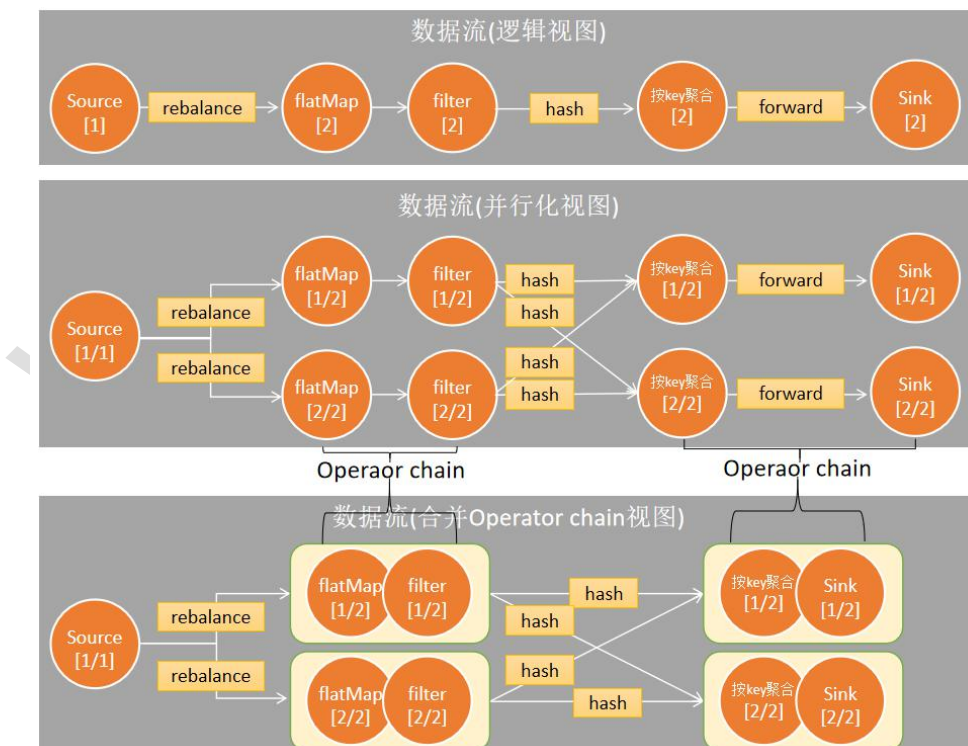
3.4.4 理解操作符链

在 1.3.3 小节中，我们曾经介绍过操作符链，这是 Flink 采用的一种称为 Operator chain 的优化技术，可以在特定条件下减少本地通信的开销。为了满足操作符链的要求，必须将两个及以上的操作符（算子）设为相同的并行度，并通过本地转发（local forward）的方式进行连接。

将操作符（算子）合并为运算符链有两个必须条件：

- ❑ 操作符（算子）的并行度相同；
- ❑ 合并操作符链的运算符（算子）都是 one-to-one 分发模式。

操作符（算子）合并为操作符链之后，原来的操作符（算子）成为里面的子任务。请看下图：



如果我们要为流操作符禁用操作符链，可以使用如下的方法：

```
// 禁止全局操作符链合并
env.disableOperatorChaining()

// 禁止某个算子的操作符链合并
filter().disableChaining()

// 断开前面的操作符链并开始一个新的操作符链
filter().startNewChain()
```

3.5 数据接收器

在使用 Flink 进行数据处理时，数据经数据源流入，然后通过系列转换，最终可以通过 Data Sink 将计算结果进行输出，Flink Data Sinks（数据接收器）就是用于定义数据流最终的输出位置，它消费数据流并将它们转发到文件、套接字、外部系统或打印输出。

3.5.1 内置数据接收器

Flink 提供了多种内置的 Data Sink API 用于日常的开发，具体如下：

- ❑ `writeAsText("/path/to/file")`: 用于将计算结果以字符串的方式并行地写入到指定文件夹下。这些字符串是通过调用每个元素的 `toString()` 方法获得的，使用的输出类是 `TextOutputFormat`。这个方法除了路径参数是必选外，还可以通过指定第二个参数来定义输出模式。输出模式有以下两个可选值：
 - `WriteMode.NO_OVERWRITE`: 当指定路径上不存在任何文件时，才执行写出操作；
 - `WriteMode.OVERWRITE`: 不论指定路径上是否存在文件，都执行写出操作；如果原来已有文件，则进行覆盖。
 - 以上写出是以并行的方式写出到多个文件，如果想要将输出结果全部写出到一个文件，需要设置其并行度为 1:

```
streamSource.writeAsText(path, FileSystem.WriteMode.OVERWRITE).setParallelism(1);
```

- ❑ `writeAsCsv("/path/to/file")`: 用于将计算结果以 CSV 的文件格式写出到指定目录。行和字段分隔符是可配置的。每个字段的值来自对象的 `toString()` 方法。使用的输出类是 `CsvOutputFormat`。该方法除了路径参数是必选外，还支持传入输出模式、行分隔符和字段分隔符三个额外的参数，其方法定义如下：

```
writeAsCsv(String path, WriteMode writeMode, String rowDelimiter, String fieldDelimiter)
```

- ❑ `print()/printToErr()`: 在标准输出/标准错误流上打印输出每个元素的 `toString()` 值。可选地，可以提供输出的前缀，这有助于区分不同的 `print` 调用。如果并行度大于 1，输出也将以产生输出的任务的 id 作为前缀。`print \ printToErr` 是测试当中最常用的方式，用于将计算结果以标准输出流或错误输出流的方式打印到控制台上。
- ❑ `writeUsingOutputFormat()`: 自定义文件输出的基类和方法。支持自定义对象到字节的转换。在定义自定义格式时，需要继承自 `FileOutputFormat`，它负责序列化和反序列化。上面介绍的 `writeAsText` 和 `writeAsCsv` 其底层调用的都是该方法，其方法签名如下：

```
public DataStreamSink<T> writeAsText(String path, WriteMode writeMode) {
    TextOutputFormat<T> tof = new TextOutputFormat<>(new Path(path));
    tof.setWriteMode(writeMode);
}
```

```
        return writeUsingOutputFormat(tof);
    }
}
```

- ❑ `writeToSocket(host, port, SerializationSchema)`: 用于将计算结果以指定的格式写出到指定的 socket 套接字。为了正确的序列化和格式化, 需要定义 `SerializationSchema`。使用示例如下:

```
streamSource.writeToSocket("192.168.0.145", 9999, new SimpleStringSchema());
```

- ❑ `addSink`: 调用自定义接收器函数。Flink 与作为接收器函数实现的其他系统(如 Apache Kafka) 的连接捆绑在一起。

注: 以上方法中, 以 `writeAs*` 开头的方法, 在 Flink API 文档中, 已经标识为 "Deprecated", 即弃用状态, 在未来的版本中有可能被删除, 因此使用时要慎重。原因请参见下一节。

在生产中, 常用的 sinks (接收器) 包括 Kafka 以及各种数据库和文件系统。下面我们通过示例来掌握常用 Flink Data Sink 的用法。

【示例】分析流数据, 并将分析结果写出到 csv 文件中。

- 1、在 IntelliJ IDEA 中创建一个 Flink 项目, 使用 `flink-quickstart-scala/flink-quickstart-java` 项目模板。(Flink 项目创建过程, 请参见 2.2 节)

- 2、设置依赖。在 `pom.xml` 中添加如下依赖:

Scala Maven 依赖:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
```

Java Maven 依赖:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-java</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
```

- 3、创建流应用程序类。代码如下:

Scala 代码:

```
package com.xueai8.scala.ch03

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

```

/**
 * Data Sink: writeAsCSV 方法，将结果写入到 csv 文件
 */
object DataSinkAsCSV {

  def main(args: Array[String]): Unit = {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 得到输入数据，进行转换：
    env.fromElements("Good good study", "Day day up")
      .map(_._toLowerCase)
      .flatMap(_._split("\\W+"))           // 相当于先 map，再 flatten
      .map(_._, 1)                         // 转换为元组
      .writeAsCsv("result.csv")           // 写出到结果文件中
      .setParallelism(1)                  // 结果写到单个文件中

    env.execute("Data Sink Demo")
  }
}

```

Java 代码：

```

package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.util.Collector;

/**
 * Data Sink: writeAsCSV 方法，将结果写入到 csv 文件
 */
public class DataSinkDemo1 {
  public static void main(String[] args) throws Exception {
    // 设置流执行环境
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

    // 获得数据，执行 map 和 flatMap 转换
    env.fromElements("Good good study", "Day day up")
      .map(String::toLowerCase)
      .flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
        @Override
        public void flatMap(String s, Collector<Tuple2<String, Integer>> out) throws Exception {
          for (String word : s.split("\\W+")) {
            out.collect(new Tuple2<>(word, 1));
          }
        }
      })
      .writeAsCsv("result.csv")           // 写出到指定的结果文件中
      .setParallelism(1);                 // 写出到一个结果文件中
  }
}

```

```
// 执行程序
env.execute("Data Sink Demo");
}
```

4、执行以上程序，可以看到，在项目的根目录下生成了一个结果文件 `result`。查看输出的结果文件 `result`，内容如下所示：

```
good,1
good,1
study,1
day,1
day,1
up,1
```

3.5.2 使用流文件连接器

正如上一节小所提到的，`DataStream` 上的 `writeAs*()` 方法主要用于调试。它们不参与 Flink 的检查点，这意味着这些函数通常具有 `at-least-once` 语义。将数据刷新（flush）到目标系统取决于 `OutputFormat` 的实现。这意味着并不是发送到 `OutputFormat` 的所有元素都立即出现在目标系统中。而且，在失败的情况下，这些记录可能会丢失。

要可靠、准确地将流一次交付到文件系统（精确一次性），需要使用 `StreamingFileSink`，这是一个流文件连接器，该连接器提供了一个 `Sink`，用于将分区文件写入由 Flink `FileSystem` 抽象支持的文件系统。

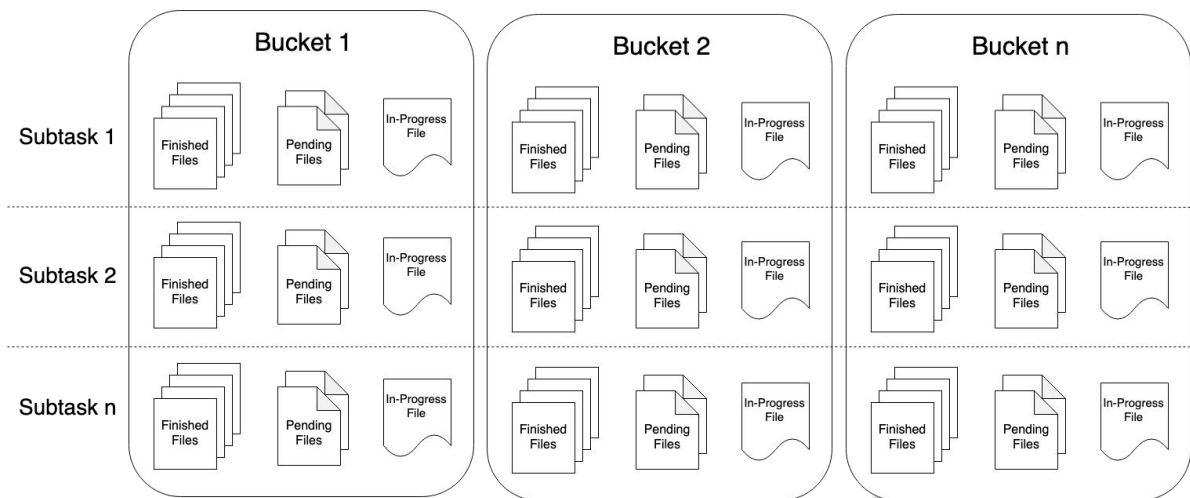
流文件接收器将传入的数据写入桶（bucket）中。假设传入的流可以不受限制，每个桶中的数据被组织成有限大小的 `part file` 文件。桶的行为是完全可配置的，默认的基于时间的桶，每小时开始写一个新的桶。这意味着每个结果桶将包含这样的文件：其包含在 1 小时间隔内从流接收到的记录。

桶目录本身包含几个 `part file` 文件，将数据写入该桶的接收器的每个并行子任务至少有一个 `part file` 文件。这些 `part file` 文件包含实际的输出数据。更多的 `part file` 文件将根据可配置的滚动策略创建。默认策略根据大小、超时(指定文件可以打开的最大持续时间)和最大不活动超时（之后文件被关闭）来滚动 `part file` 文件。

需要注意的是，在流模式下使用 `FileSink` 时需要为流作业启用检查点。流数据流的分布式状态将被定期快照。如果出现故障，流数据流将从最近完成的检查点重新启动。部分文件只能在成功的检查点上完成。如果禁用检查点，部分文件将永远停留在 `in-progress` 或 `pending` 状态，下游系统无法安全地读取。

关于 `StreamingFileSink` 更详细的说明，请参考官方 API 文档：

<https://ci.apache.org/projects/flink/flink-docs-release-1.13/api/java/>



【示例】分析流数据，并将分析结果写出到文本文件中。

- 1、在 IntelliJ IDEA 中创建一个 Flink 项目，使用 `flink-quickstart-scala/flink-quickstart-java` 项目模板。（Flink 项目创建过程，请参见 2.2 节）
- 2、设置依赖。在 `pom.xml` 中添加如下依赖：

Scala Maven 依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
```

Java Maven 依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-java</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
```

- 3、创建流应用程序类。代码如下：

Scala 代码：

```
package com.xueai8.scala.ch03
```



```

import org.apache.flink.api.common.serialization.SimpleStringEncoder
import org.apache.flink.core.fs.Path
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink
import org.apache.flink.streaming.api.scala._

/**
 * Data Sink: 使用 addSink(...)方法
 */
object DataSinkWithStreamingFile {
  def main(args: Array[String]): Unit = {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 为流作业启用检查点
    // 此方法选择 CheckpointingMode.EXACTLY_ONCE 保证
    env.enableCheckpointing(60000) // 每 1 分钟（60 秒）执行一次快照

    // 自定义的 Sink
    val sink: StreamingFileSink[(String,Int)] = StreamingFileSink
      .forRowFormat(new Path("./result"), new SimpleStringEncoder[(String,Int)]("UTF-8"))
      .build()

    // 得到输入数据，进行转换：
    val input: DataStream[(String,Int)] = env.fromElements("Good good study", "Day day up")
      .map(_._toLowerCase) // 转小写
      .flatMap(_._split("\\W+")) // 相当于先 map，再 flatten
      .map(_._1) // 转元组

    // 使用自定义的 Sink
    input.addSink(sink) // 写出到结果文件中
      .name("Flink2File")
      .setParallelism(1) // 结果写到单个文件中

    // 触发流程序执行
    env.execute("Data Sink Demo")
  }
}

```

Java 代码:

```

package com.xueai8.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.serialization.SimpleStringEncoder;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.core.fs.Path;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink;
import org.apache.flink.util.Collector;

```

```
/**
 * Data Sink: 使用 addSink(...)方法
 */
public class DataSinkWithStreamingFile {
    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 为流作业启用检查点
        // 此方法选择 CheckpointingMode.EXACTLY_ONCE 保证
        env.enableCheckpointing(60000) // 每 1 分钟（60 秒）执行一次快照

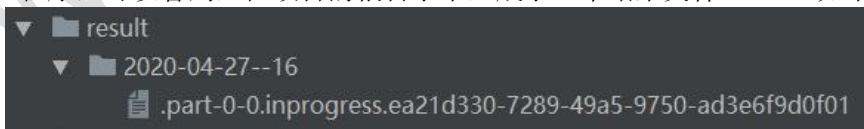
        // 自定义的 Sink
        StreamingFileSink<Tuple2<String,Integer>> sink = StreamingFileSink
            .forRowFormat(new Path("./result"), new SimpleStringEncoder<Tuple2<String,Integer>>())
            .build();

        // 获得数据，执行 map 和 flatMap 转换
        DataStream<Tuple2<String,Integer>> input = env.fromElements("Good good study","Day day up")
            .map(String::toLowerCase)
            .flatMap(new FlatMapFunction<String, Tuple2<String,Integer>>() {
                @Override
                public void flatMap(String s, Collector<Tuple2<String,Integer>> out) throws Exception {
                    for(String word : s.split("\\W+")){
                        out.collect(new Tuple2<>(word,1));
                    }
                }
            });

        // 指定自定义的 Sink，并设置并行度为 1，以便结果写入单个文件中
        // Flink 提供了 addSink 方法用来调用自定义的 Sink 或者第三方的连接器
        input.addSink(sink).name("Flink2File").setParallelism(1);

        // 执行流程序
        env.execute("Data Sink Demo");
    }
}
```

4、执行以上程序，可以看到，在项目的根目录下生成了一个结果文件 result。如下图所示：



查看输出的结果文件 result，内容如下所示：

```
(good,1)
(good,1)
(study,1)
(day,1)
(day,1)
(up,1)
```

文件格式

StreamingFileSink 既支持行编码(row-wise)格式, 也支持批量(bulk)编码格式, 比如 Apache Parquet。这两个变量都附带了它们各自的构建器, 可以使用以下静态方法创建它们:

- ❑ Row-encoded sink: StreamingFileSink.forRowFormat(basePath, rowEncoder)
- ❑ Bulk-encoded sink: StreamingFileSink.forBulkFormat(basePath, bulkWriterFactory)

在创建行编码的接收器或批编码的接收器时, 我们必须指定存储桶的基本路径和数据的编码逻辑。基本路径中每个桶对应一个目录。

行编码格式

行编码格式需要指定一个 Encoder (编码器), 用于将单独的行序列化到 in-progress part file 文件的 OutputStream。

除了桶分配器(bucket assigner), RowFormatBuilder 还允许用户指定:

- ❑ 自定义 RollingPolicy: 滚动策略以覆盖 DefaultRollingPolicy;
- ❑ bucketCheckInterval (default = 1 min) : 检查基于时间的滚动策略的毫秒间隔。

例如, 写字符串流元素到文件中, 使用类似下面这样的代码。

Scala 代码

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

val input: DataStream[String] = ...

val sink: StreamingFileSink[String] = StreamingFileSink
    .forRowFormat(new Path(outputPath), new SimpleStringEncoder[String]("UTF-8"))
    .withRollingPolicy(
        DefaultRollingPolicy.builder()
            .withRolloverInterval(TimeUnit.MINUTES.toMillis(15))
            .withInactivityInterval(TimeUnit.MINUTES.toMillis(5))
            .withMaxPartSize(1024 * 1024 * 1024)
            .build())
    .build()

input.addSink(sink)
```

Java 代码

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

DataStream<String> input = ...;

final StreamingFileSink<String> sink = StreamingFileSink
    .forRowFormat(new Path(outputPath), new SimpleStringEncoder<String>("UTF-8"))
    .withRollingPolicy(
        DefaultRollingPolicy.builder()
```

```
.withRolloverInterval(TimeUnit.MINUTES.toMillis(15))  
.withInactivityInterval(TimeUnit.MINUTES.toMillis(5))  
.withMaxPartSize(1024 * 1024 * 1024)  
.build()  
  
.build();
```

input.addSink(sink);

这个示例创建了一个简单的接收器，它将记录分配给默认的一小时时间桶。它还指定了一个滚动策略，在以下 3 种情况下滚动处理 in-progress 状态的 part file 文件：

- ☐ 它包含至少 15 分钟的数据；
- ☐ 在过去的 5 分钟里，它没有接收到新的记录；
- ☐ 文件大小达到了 1 GB(在写入最后一条记录之后)。

批编码格式

批量编码的接收器与行编码的接收器创建方式类似，但我们必须指定 `BulkWriter.Factory`，而不是指定 `Encoder`（编码器）。`BulkWriter` 逻辑定义了如何添加和刷新新元素，以及如何为进一步的编码目的完成一批记录。

Flink 有四个内置的 `BulkWriter` 工厂：

- ☐ `ParquetWriterFactory`
- ☐ `AvroWriterFactory`
- ☐ `SequenceFileWriterFactory`
- ☐ `CompressWriterFactory`
- ☐ `OrcBulkWriterFactory`

`Bulk Formats` 只能有一个扩展“`CheckpointRollingPolicy`”的滚动策略，它只在每个检查点上滚动。

更为详细的说明，请参考官方文档：

https://ci.apache.org/projects/flink/flink-docs-stable/dev/connectors/file_sink.html

Bucket Assignment

bucket 逻辑定义了如何将数据结构化到基本输出目录中的子目录中。

行格式和批量格式都使用 `DateTimeBucketAssigner` 作为默认分配器。默认情况下，`DateTimeBucketAssigner` 会根据系统默认时区创建每小时的桶，格式如下：yyyy-MM-dd--HH。日期格式(即桶大小)和时区都可以手动配置。

可以通过在格式构建器上调用 `.withBucketAssigner(assigner)` 来指定一个自定义的 `BucketAssigner`。

Flink 有两个内置的 `BucketAssigners`：

- ☐ `DateTimeBucketAssigner`：默认的基于时间的分配器。
- ☐ `BasePathBucketAssigner`：将所有 part file 文件存储在基本路径中(单个全局桶)的分配器。

Rolling Policy

`RollingPolicy` 定义了给定的 in-progress part file 文件何时被关闭，何时被移到 pending（挂起）状态，何时移到 finished（完成）状态。处于“finished”状态的 part file 文件是可以查看的，并且保证包含在故障时不会被还原的有效数据。滚动策略与检查点间隔(挂起的文件在下一个检查点完成)相结合，控制 part file 文件对下游读取器可用的速度，以及这些 part file 的大小和数量。

Flink 自带两个 RollingPolicies:

- ☐ DefaultRollingPolicy
- ☐ OnCheckpointRollingPolicy

Part file 生命周期

为了在下游系统中使用 StreamingFileSink 的输出,我们需要理解生成的输出文件的命名和生命周期。

Part file 文件可以处于以下三种状态之一:

- ☐ in-progress: 正在写入的 part file 文件是 in-progress。
- ☐ pending: 已经关闭(由于指定的滚动策略)正在等待被提交的 in-progress 文件。
- ☐ finished: 在成功的检查点上,挂起的文件转换为“finished”。

只有 finished 的文件才可以被下游系统安全地读取,因为这些文件保证以后不会被修改。

对于任何给定的子任务,part file 文件索引都是严格递增的(按照它们创建的顺序)。然而,这些索引并不总是顺序的。当作业重新启动时,所有子任务的下一个 part 索引将是 $\text{max part index} + 1$, 其中 max 是根据所有子任务计算出来的。

对于每个活动 bucket,每个 writer 子任务在任何给定时间都将有单个 in-progress part file 文件,但也可能有几个 pending 和 finished 的文件。

3.5.3 自定义数据接收器

Flink 还支持使用自定义的 Sink 来满足多样化的输出需求。想要实现自定义的 Sink,需要直接或者间接实现 SinkFunction 接口。通常情况下,我们都是实现其抽象类 RichSinkFunction,相比于 SinkFunction,其提供了更多的与生命周期相关的方法。

下面我们自定义一个 FlinkToMySQLSink,将流计算结果写出到指定的 MySQL 数据库中。

【示例】改写上一示例,将流计算结果保存到 MySQL 数据库中。

1、在 MySQL 的 xueai8 数据库中,创建一个数据表 wc,用来存储 Flink 的计算结果。创建 wc 表的脚本代码如下:

```
mysql> create table wc(word varchar(30), cnt int);
```

2、在 IntelliJ IDEA 中创建一个 Flink 项目,使用 flink-quickstart-scala/flink-quickstart-java 项目模板。(Flink 项目创建过程,请参见 2.2 节)

3、设置依赖。在 pom.xml 中添加如下依赖:

Scala Maven 依赖:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
```

```
<!--JDBC 连接器-->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-jdbc_2.12</artifactId>
  <version>1.13.2</version>
</dependency>

<!--mysql 连接器-->
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.48</version>
</dependency>
```

Java Maven 依赖:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-java</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>

<!--JDBC 连接器-->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-jdbc_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.48</version>
</dependency>
```

4、创建流应用程序类。代码如下:

Scala 代码:

```
package com.xueai8.scala.ch03

import java.sql.{Connection, DriverManager, PreparedStatement}

import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.CheckpointingMode
import org.apache.flink.streaming.api.functions.sink.RichSinkFunction
import org.apache.flink.streaming.api.functions.sink.SinkFunction.Context
```

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

/**
 * Data Sink: 使用 addSink(...)方法, 指定 MySQL 数据库作为 Data Sink
 */
object DataSinkToMysql {

    // 自定义的 Sink 类, 继承自 RichSinkFunction
    class FlinkToMySQLSink extends RichSinkFunction[(String, Int)] {
        // 定义 mysql 数据库连接 url 和驱动程序及账号、密码
        val url = "jdbc:mysql://192.168.190.133:3306/xueai8?characterEncoding=UTF-8&useSSL=false"
        val driver = "com.mysql.jdbc.Driver"
        val username = "root"
        val userpwd = "admin"

        // 声明数据库连接对象和 SQL 预编译语句
        var conn: Connection = _
        var stmt: PreparedStatement = _

        // 重写 open 方法: 加载驱动, 建立连接, 预编译 SQL 语句
        // 先于 invoker 方法用, 仅执行一次
        override def open(parameters: Configuration): Unit = {
            super.open(parameters)
            // 加载驱动程序
            Class.forName(driver)
            // 连接数据库
            conn = DriverManager.getConnection(url, username, userpwd)
            // 执行 SQL 语句
            val sql = "insert into wc(word, cnt) values(?, ?)"
            stmt = conn.prepareStatement(sql)
        }

        // 重写事件驱动方法调用
        override def invoke(value: (String, Int), context: Context): Unit = {
            stmt.setString(1, value._1)
            stmt.setInt(2, value._2)
            stmt.executeUpdate()
        }

        // 关闭资源
        // 后于 invoke 方法调用, 仅执行一次
        override def close(): Unit = {
            super.close()
            if (stmt != null) stmt.close()
            if (conn != null) conn.close()
        }
    }

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

```
// 设置流执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 为流数据启用检查点
env.enableCheckpointing(2000, CheckpointingMode.EXACTLY_ONCE)

// 得到输入数据，进行转换：
val input = env.fromElements("Good good study", "Day day up")
    .map(_._toLowerCase)           // 转小写
    .flatMap(_._split("\\W+"))    // 相当于先 map，再 flatten
    .map(_._, 1)                  // 转换为元组

// 使用自定义的 Sink
input.addSink(new FlinkToMySQLSink) // 写出到数据库中

// 触发程序执行
env.execute("Data Sink Demo")
}
```

Java 代码：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.sink.RichSinkFunction;
import org.apache.flink.util.Collector;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;

/**
 * Data Sink：使用 addSink(...)方法，指定 MySQL 数据库作为 Data Sink
 */
public class DataSinkToMysql {

    // 自定义的 Sink 类，继承自 RichSinkFunction
    public static class FlinkToMySQLSink extends RichSinkFunction<Tuple2<String,Integer>> {
        // 定义 mysql 数据库连接 url 和驱动程序及账号、密码
        private static String JDBC_URL = "jdbc:mysql://192.168.190.133:3306/xueai8" +
            "?characterEncoding=UTF-8&useSSL=false";
        private static String USER_NAME = "root";
        private static String USER_PWD = "admin";

        // 声明数据库连接对象和 SQL 预编译语句
        private PreparedStatement stmt;
        private Connection conn;
```



```
// 重写 open 方法：加载驱动，建立连接，预编译 SQL 语句
@Override
public void open(Configuration parameters) throws Exception {
    Class.forName("com.mysql.jdbc.Driver");
    conn = DriverManager.getConnection(JDBC_URL, USER_NAME, USER_PWD);
    String sql = "insert into wc(word, cnt) values(?, ?)";
    stmt = conn.prepareStatement(sql);
}

// 重写事件驱动方法调用
@Override
public void invoke(Tuple2<String,Integer> t, Context context) throws Exception {
    stmt.setString(1, t.f0);
    stmt.setInt(2, t.f1);
    stmt.executeUpdate();
}

// 关闭资源
@Override
public void close() throws Exception {
    super.close();
    if (stmt != null) {
        stmt.close();
    }
    if (conn != null) {
        conn.close();
    }
}

}

public static void main(String[] args) throws Exception {
    // 设置流执行环境
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

    // 为流数据启用检查点
    env.enableCheckpointing(2000);

    // 获得数据，执行 map 和 flatMap 转换
    DataStream<Tuple2<String,Integer>> stream = env
        .fromElements("Good good study", "Day day up")
        .map(String::toLowerCase)
        .flatMap(new FlatMapFunction<String, Tuple2<String,Integer>>() {
            @Override
            public void flatMap(String s, Collector<Tuple2<String,Integer>> out) throws Exception {
                for(String word : s.split("\\W+")){
                    out.collect(new Tuple2<>(word,1));
                }
            }
        });
};
```

```
// 指定使用自定义的 Sink
stream.addSink(new FlinkToMySQLSink());

// 执行程序
env.execute("Data Sink Demo");
}
```

执行以上程序，然后在 MySQL 中查看表内容：

```
mysql> select * from wc;
```

可以看到，程序计算的结果已经保存到数据库中了：

```
mysql> select * from wc;
+-----+-----+
| word  | cnt  |
+-----+-----+
| good  | 1    |
| day   | 1    |
| day   | 1    |
| good  | 1    |
| up    | 1    |
| study | 1    |
+-----+-----+
6 rows in set (0.00 sec)
```

注：如果想将流计算结果写出到类似 Kafka 这样的第三方消息中间件或者 HDFS 这样的存储系统，请稍后参考 3.12 节的“流连接器”部分。

在上面的代码中，我们自定义 Sink 是继承自 RichSinkFunction 类，它是 SinkFunction 的“Rich”变体。与 SinkFunction 相比，它有一些额外的方法，包括：

- ❑ open(Configuration c)
- ❑ close()
- ❑ getRuntimeContext()

在操作符初始化期间调用 Open() 一次。例如，这是加载一些静态数据或打开到外部服务的连接的机会。而 getRuntimeContext() 提供了对底层环境的访问能力，例如创建和访问由 Flink 管理的状态。

3.6 时间和水印

时间是流应用程序的另一个重要组成部分。大多数事件流都具有固有的时间语义，因为每个事件都在特定的时间点生成。此外，许多常见的流计算都是基于时间的，比如窗口聚合、会话、模式检测和基于时间的连接。

Flink 提供了一组丰富的与时间相关的特性：

- ❑ 事件时间模式：流应用程序（使用事件时间语义处理流）基于事件的时间戳计算结果。因此，事件时间处理允许精确和一致的结果。
- ❑ 处理时间模式：除了事件时间模式，Flink 还支持处理时间语义，它执行由处理机器的挂钟时间触发的计算。处理时间模式可以适用于某些具有严格低延迟要求的应用程序，这些应用程序可以容忍近似结果。
- ❑ 水印支持：Flink 在事件时间应用程序中使用水印来推断时间。水印是一种灵活的机制，用来平衡结果的延迟和完整性。

- ❑ 迟到数据处理：当以事件时间模式处理带有水印的流时，可能会在所有相关事件到达之前完成计算。这样的事件称为迟到事件。Flink 提供了多个处理迟到事件的选项，比如通过侧输出重新路由它们，并更新以前完成的结果。

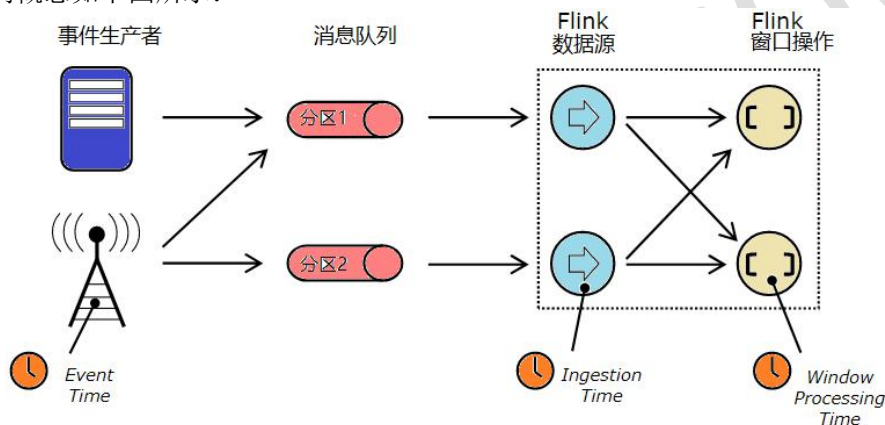
3.6.1 时间概念

Flink Streaming API 借鉴了谷歌数据流模型（Google Data Flow Model），它的流 API 支持不同的时间概念。

Flink 明确支持三个不同的时间概念：

- ❑ 事件时间：事件发生的时间，由产生(或存储)事件的设备记录。
- ❑ 接入时间：Flink 在接入事件时记录的时间戳。
- ❑ 处理时间：管道中特定操作符处理事件的时间。

这几种时间概念如下图所示：



Event Time

事件时间，是事件在其生产设备上发生的时间。例如在物联网项目中，传感器捕获读数的时间。通常，这些事件时间需要在它们进入 Flink 之前嵌入到记录中，由事件中的时间戳描述。在处理时，可以从每个记录中提取这些事件时间戳并考虑用于窗口。Flink 通过时间戳分配器访问事件时间戳。事件时间处理可用于无序事件。

如果要使用事件时间，还需要提供时间戳提取器和水印生成器，Flink 将使用它们来跟踪事件时间的进展。这种水印机制将在下一小节中进行描述。

Processing Time

处理时间，是机器执行数据流处理的系统时间，是执行基于时间的操作的每个操作符的本地时间。处理时间窗口只考虑事件被处理时的时间戳。处理时间是最简单的时间概念，因为它不需要处理流和生产机器之间的任何同步与协调。然而，在分布式异步环境中，处理时间不提供确定性。

当流程序在处理时间上运行时，所有基于时间的操作(如时间窗口)将使用运行各个操作符的机器的系统时钟。每小时处理时间窗口将包括在系统时钟指示完整小时之间到达特定操作符的所有记录。例如，如果一个应用程序在上午 9:15 开始运行，那么第一个每小时处理时间窗口将包括上午 9:15 到 10:00 之间处理的事件，下一个窗口将包括上午 10:00 到 11:00 之间处理的事件，依此类推。

Ingestion Time

摄入时间，这是特定事件在源操作符处进入 Flink 数据流的时间。在源操作符中，每个记录以时间戳的形式获取源的当前时间，基于时间的操作(如时间窗口)引用该时间戳。摄入时间概念上位于事件时间和处理时间之间。由于摄取时间使用稳定的时间戳(在源处分配一次)，对记录的不同窗口操作将引用相同的时间戳，所以摄取时间虽然是比处理时间更昂贵的操作，但它提供可预测的结果。

与事件时间相比，摄取时间程序不能处理任何无序的事件或延迟的数据，因为它只在事件进入 Flink 系统后才分配时间戳。但程序不必指定如何生成水印。在内部，摄取时间处理得很像事件时间，但具有自动时间戳分配和自动水印生成。

处理时间 vs. 事件时间

处理时间是通过直接调用本地机器的时间确定的，而事件时间是由每个处理记录所携带的时间戳确定的。这两种时间在 Flink 的内部处理和用户的实际使用上是不同的。相对来说，处理时间比较容易处理，而事件时间比较麻烦。当使用处理时间时，我们得到的处理结果(或流处理应用程序的内部状态)是不确定的。因为在 Flink 中事件时间是有保证的，所以无论数据重放多少次，使用事件时间都可以获得相对确定且可重现的结果。

因此，在判断应该使用处理时间还是事件时间时，可以遵循一个原则：当应用程序遇到一些问题时，需要从前一个检查点或保存点重放它。希望结果完全一样吗？如果想要结果完全相同，只能使用事件时间；如果接受不同的结果，可以使用处理时间。处理时间的一个常见用途是实时计算整个系统的吞吐量。例如，要计算实时一小时内有多少数据被处理，我们只能使用处理时间。

设置时间特性

时间特性定义了系统如何为依赖时间的顺序和依赖时间的操作（如时间窗口）确定时间。默认情况下，Flink DataStream 程序将使用 EventTime（事件时间）。如果要改用处理时间，那么需要在一开始就设置时间特性。

Scala:

```
// 获得流执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 设置流的时间特性（这里设置为采用处理时间）
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)
```

Java:

```
// 获得流执行环境
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

// 设置流的时间特性（这里设置为采用处理时间）
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);
```

注：在 Flink 1.12 之前，Flink DataStream 默认使用的是处理时间。从 Flink 1.12 开始，默认的流时间特性已被更改为 EventTime，因此不再需要调用此方法来启用事件时间支持。

当然也可以选择设置其他类型时间特性。

Scala:

```
env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime)
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
```

Java:

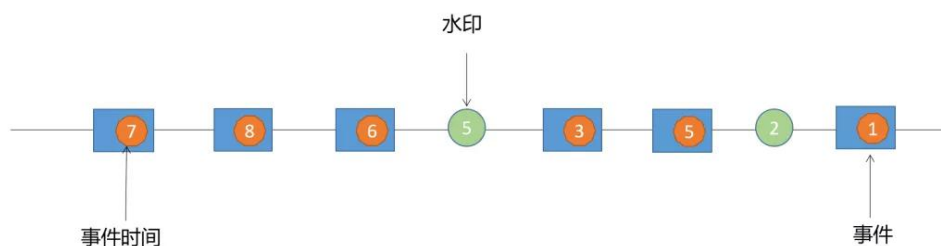
```
env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime);
```

```
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

3.6.2 事件时间和水印

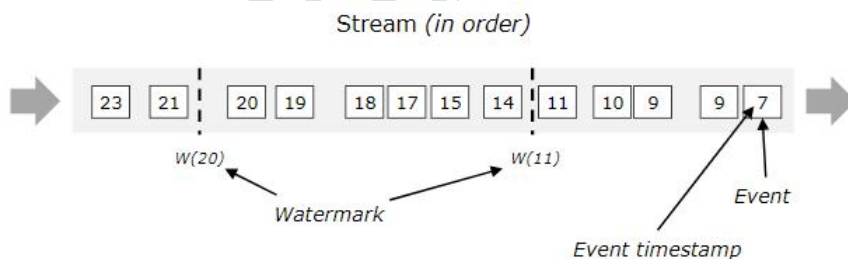
支持事件时间的流处理器需要一种方法来度量事件时间的进度。例如，针对事件时间对数据进行窗口或排序的操作符必须缓冲数据，直到它们能够确保已接收到某个时间间隔的所有时间戳为止。这是由所谓的“时间水印”来处理的。

在 Flink 中测量事件时间进展的机制是水印（watermarks），水印是一种特殊类型的事件，是告诉系统事件时间进度的一种方式。水印流是数据流的一部分，并带有时间戳 t 。水印(t)声明事件时间已经达到该流中的时间 t ，这意味着时间戳 $t' \leq t$ （即时间戳更早或等于水印的事件）的流中不应该有更多的元素。

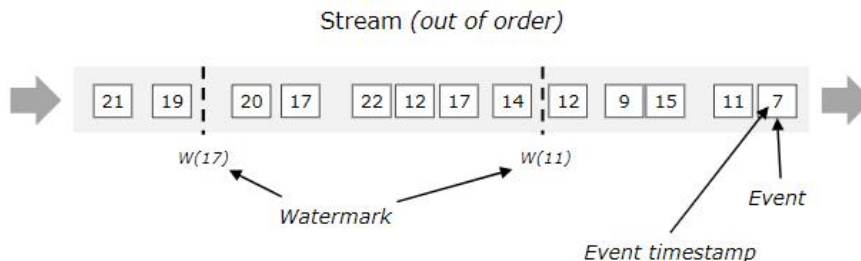


时间 t 的水印标记了数据流中的一个位置，并断言此时的流在时间 t 之前已经完成。为了执行基于事件时间的事件处理，Flink 需要知道与每个事件相关联的时间，它还需要包含水印的流。水印就是系统事件时间的时钟。水印触发基于事件时间的计时器的触发。

下图显示了带有（逻辑的）时间戳的事件流，以及内联流动的水印。在这个例子中，事件是按顺序排列的（相对于它们的时间戳），这意味着水印只是流中的周期标记。



对于无序流，水印是至关重要的，如下图所示，其中事件不是按照它们的时间戳排序的。



例如，当操作符接收到 $w(11)$ 这条水印时，可以认为时间戳小于或等于 11 的数据已经到达，此时可以触发计算。同样，当接收到 $w(17)$ 这条水印时，可以认为时间戳小于或等于 17 的数据已经到达，此时可以触发计算。

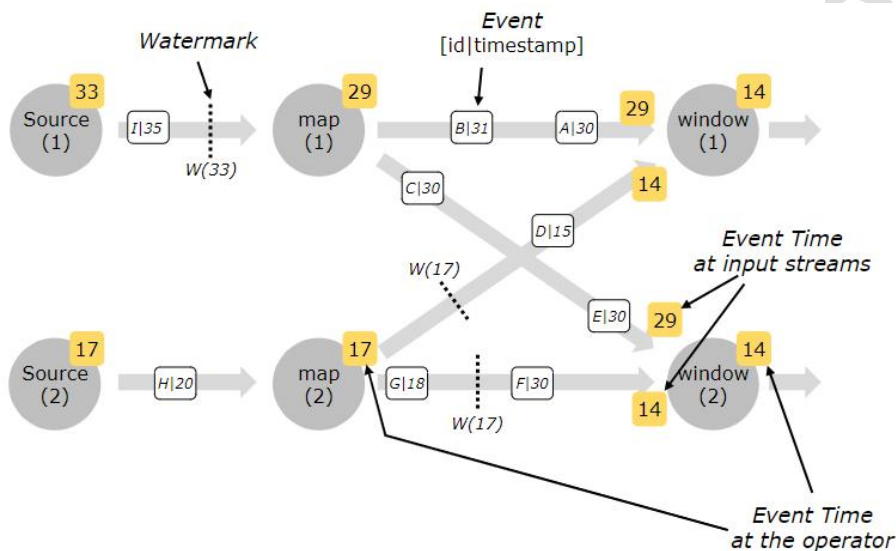
可以看出，水印的时间戳是单调递增的，时间戳为 t 的水印意味着所有后续记录的时间戳将大于 t 。一般来说，水印是一种声明，在流中的那个点之前，在某个时间戳之前的所有事件都应该已经到达。当水印到达运算符（算子）时，运算符可以将其内部事件时间时钟推进到水印的值。

并行流中的水印

水印是在源函数处或直接在源函数之后生成的。源函数的每个并行子任务通常独立地生成其水印。这些水印定义了特定并行源处的事件时间。

当这些水印流经程序时，它们会将事件时间前进到它们到达的操作符（算子）处。每当一个操作符推进它的事件时间，它为它的后继操作符产生一个新的下游水印。

下图显示了事件和水印在并行流中流动的示例，以及操作符（算子）跟踪事件时间的示例。



3.6.3 水印策略

Flink 提供了用于处理事件时间、时间戳和水印的 API。

什么是水印策略？

为了处理事件时间，Flink 流程序需要知道事件的时间戳，这意味着流中的每个元素都需要分配其事件时间戳。这通常是通过使用 `TimestampAssigner` 从元素中的某个字段访问/提取时间戳来实现的。

时间戳分配与生成水印密切相关，生成水印是告诉系统事件时间进度的一种方式。可以通过指定 `WatermarkGenerator` 来配置水印。

`TimestampAssigner` 和 `WatermarkGenerator` 形成了一个水印策略，它定义了如何在流源中生成水印。Flink API 需要一个同时包含 `TimestampAssigner` 和 `WatermarkGenerator` 的 `WatermarkStrategy`。许多常用的策略都可以作为 `WatermarkStrategy` 上的静态方法来使用，但用户也可以在需要时构建自己的策略。

完整的 `WatermarkStrategy` 接口的定义如下：

```
public interface WatermarkStrategy<T>
    extends TimestampAssignerSupplier<T>,
        WatermarkGeneratorSupplier<T>{
```

```

/**
 * Instantiates a {@link TimestampAssigner} for assigning timestamps according to this strategy.
 */
@Override
TimestampAssigner<T> createTimestampAssigner(TimestampAssignerSupplier.Context context);

/**
 * Instantiates a WatermarkGenerator that generates watermarks according to this strategy.
 */
@Override
WatermarkGenerator<T> createWatermarkGenerator(WatermarkGeneratorSupplier.Context context);
}

```

通常不会自己实现这个接口，而是使用 `WatermarkStrategy` 上的静态辅助方法来实现公共水印策略，或者将自定义 `TimestampAssigner` 与 `WatermarkGenerator` 捆绑在一起。例如，要使用有界无序水印和 `lambda` 函数作为时间戳赋值器，可以使用如下代码。

Scala:

```

WatermarkStrategy
  .forBoundedOutOfOrderness[(Long, String)](Duration.ofSeconds(20))
  .withTimestampAssigner(new SerializableTimestampAssigner[(Long, String)] {
    override def extractTimestamp(element: (Long, String), recordTimestamp: Long): Long = element._1
  })

```

Java:

```

WatermarkStrategy
  .<Tuple2<Long, String>>forBoundedOutOfOrderness(Duration.ofSeconds(20))
  .withTimestampAssigner((event, timestamp) -> event.f0);

```

`TimestampAssigner` 是一个简单的函数，它从事件中提取字段。指定一个 `TimestampAssigner` 是可选的，在大多数情况下，实际上我们并不希望指定一个 `TimestampAssigner`。例如，当使用 `Kafka` 或 `Kinesis` 时，我们会直接从 `Kafka/Kinesis` 记录中获得时间戳。

注：时间戳和水印都指定为自 Java 纪元(1970-01-01 00:00:00)以来的毫秒数。

使用水印策略

在 Flink 应用程序中有两个地方可以使用 `WatermarkStrategy`：

- ☐ 1) 直接在源上使用；
- ☐ 2) 在非源操作之后使用。

第一种选择更可取，因为它允许源利用关于水印逻辑中的分片/分区/分割的知识。源通常可以在更细的水平上跟踪水印，源产生的整体水印将更准确。直接在源上指定 `WatermarkStrategy` 通常意味着必须使用一个源特定的接口。

第二个选项（在任意操作后设置 `WatermarkStrategy`）应该只在不能直接在源上设置策略的情况下使用。

1) 直接在源上使用

这种方式指的是数据源本身已经嵌入了事件时间，在处理事件时间流时直接提取其中的事件时间的时间戳即可。因为流数据源可以直接将时间戳分配给它们生成的元素，它们还可以发出水印，因此不需要时间戳分配程序。

注 1：在源文件中存储记录时，最好存储事件时间。

注 2：如果使用了时间戳分配器，则会覆盖源提供的任何时间戳和水印。

要直接向流数据源中的元素分配时间戳，流源必须使用 SourceContext 上的 collectWithTimestamp(...) 方法。要生成水印，源程序必须调用 emitWatermark(Watermark) 函数。

下面是一个简单的（非检查点）的流数据源分配时间戳和产生水印的例子。

Scala 代码：

```
override def run(ctx: SourceContext[MyType]): Unit = {
  while (/* condition */) {
    val next: MyType = getNext()
    ctx.collectWithTimestamp(next, next.eventTimestamp)

    if (next.hasWatermarkTime) {
      ctx.emitWatermark(new Watermark(next.getWatermarkTime))
    }
  }
}
```

Java 代码：

```
@Override
public void run(SourceContext<MyType> ctx) throws Exception {
  while (/* condition */) {
    MyType next = getNext();
    ctx.collectWithTimestamp(next, next.getEventTimestamp());

    if (next.hasWatermarkTime()) {
      ctx.emitWatermark(new Watermark(next.getWatermarkTime()));
    }
  }
}
```

2) 在非源操作之后使用

时间戳分配程序获取一个流并生成一个带有时间戳元素和水印的新数据流。如果原始数据流已经具有时间戳和/或水印，则时间戳分配器将覆盖它们。

时间戳分配程序通常在数据源之后立即指定。例如，一个常见的模式是先对事件进行解析 (MapFunction) 和筛选 (FilterFunction)，然后再分配时间戳。在任何情况下，都需要在事件时间的第一个操作（例如第一个窗口操作）之前指定时间戳分配程序。

注：作为一种特殊情况，当使用 Kafka 作为流作业的源时，Flink 允许在源(或消费者)内部指定一个时间戳分配器/水印发射器。

Scala 代码：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

val stream: DataStream[MyEvent] = env.readFile(
  myFormat,
  myFilePath,
  FileProcessingMode.PROCESS_CONTINUOUSLY,
  100,
  FilePathFilter.createDefaultFilter())
```



```
val withTimestampsAndWatermarks: DataStream[MyEvent] = stream
    .filter( _ .severity == WARNING )
    .assignTimestampsAndWatermarks(<<watermark strategy>>)

withTimestampsAndWatermarks
    .keyBy( _ .getGroup )
    .window(TumblingEventTimeWindows.of(Time.seconds(10)))
    .reduce( (a, b) => a.add(b) )
    .addSink(...)
```

Java 代码:

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

DataStream<MyEvent> stream = env.readFile(
    myFormat,
    myFilePath,
    FileProcessingMode.PROCESS_CONTINUOUSLY,
    100,
    FilePathFilter.createDefaultFilter(),
    typeInfo);

DataStream<MyEvent> withTimestampsAndWatermarks = stream
    .filter( event -> event.severity() == WARNING )
    .assignTimestampsAndWatermarks(<watermark strategy>);

withTimestampsAndWatermarks
    .keyBy( (event) -> event.getGroup() )
    .window(TumblingEventTimeWindows.of(Time.seconds(10)))
    .reduce( (a, b) -> a.add(b) )
    .addSink(...);
```

以这种方式使用 `WatermarkStrategy` 获取一个流，并生成一个带有时间戳的元素和水印的新流。如果原始流已经有时间戳和/或水印，时间戳分配器将覆盖它们。

3.6.4 处理空闲数据源

如果一个输入 `split / partitions / shards` 在一段时间内不携带事件，这意味着 `WatermarkGenerator` 也不会获得任何新信息来作为水印的基础。我们称之为空闲输入或空闲源。这是一个问题，因为可能会发生一些分区仍然携带事件。在这种情况下，水印将被保留，因为它是作为所有不同的并行水印的最小值计算的。

要处理这个问题，可以使用 `WatermarkStrategy` 来检测空闲状态并将输入标记为 `idle`。`WatermarkStrategy` 为此提供了一个方便的辅助方法。

Scala 代码:

```
WatermarkStrategy
    .forBoundedOutOfOrderness[(Long, String)](Duration.ofSeconds(20))
    .withIdleness(Duration.ofMinutes(1))
```

Java 代码:

```
WatermarkStrategy
```

```
.<Tuple2<Long, String>>forBoundedOutOfOrderness(Duration.ofSeconds(20))  
.withIdleness(Duration.ofMinutes(1));
```

`WatermarkStrategy.withIdleness()`方法允许用户在配置的时间内（即超时时间内）没有记录到达时将一个流标记为空闲。这就意味着下游的数据不需要等待水印的到来。当下次有水印生成并发射到下游的时候，这个数据流重新变成活跃状态。

3.6.5 编写水印生成器

在 `WatermarkStrategy` 中，`TimestampAssigner` 是一个简单的函数，它从事件中提取字段，因此我们不需要过多地关注。但是编写 `WatermarkGenerator` 稍微复杂一些，所以接下来我们主要了解如何实现它。

`WatermarkGenerator` 接口的声明如下：

```

/**
 * The {@code WatermarkGenerator} generates watermarks either based on events or
 * periodically (in a fixed interval).
 *
 * <p><b>Note:</b> This WatermarkGenerator subsumes the previous distinction between the
 * {@code AssignerWithPunctuatedWatermarks} and the {@code AssignerWithPeriodicWatermarks}.
 */
@Public
public interface WatermarkGenerator<T> {

    /**
     * Called for every event, allows the watermark generator to examine and remember the
     * event timestamps, or to emit a watermark based on the event itself.
     */
    void onEvent(T event, long eventTimestamp, WatermarkOutput output);

    /**
     * Called periodically, and might emit a new watermark, or not.
     *
     * <p>The interval in which this method is called and Watermarks are generated
     * depends on {@link ExecutionConfig#getAutoWatermarkInterval()}.
     */
    void onPeriodicEmit(WatermarkOutput output);
}

```

有两种不同的水印生成方法：periodic (周期型)和 punctuated(标点型)。

- ☐ 周期生成器通常通过 onEvent()观察传入的事件，然后在框架调用 onPeriodicEmit()时发出水印。
- ☐ 标点生成器将查看 onEvent()中的事件，并等待流中携带水印信息的特殊标记事件或标点。当它看到这些事件之一时，它会立即发出水印。通常，标点生成器不会从 onPeriodicEmit()发出水印。

编写 Periodic WatermarkGenerator

周期性生成器观察流事件并周期性地生成水印（可能取决于流元素，或者纯粹基于处理时间）。生成水印的时间间隔（每 n 毫秒）是通过 ExecutionConfig.setAutoWatermarkInterval(…)定义的，默认是 200ms。例如，通过下面的方式来设置间隔时间。

```

// 每 3 秒生成一个水印
env.getConfig().setAutoWatermarkInterval(3000);

```

每次都会调用生成器的 onPeriodicEmit()方法，如果返回的水印非空且大于前一个水印，则会发出一个新的水印。

这里我们展示了两个使用周期性水印生成的简单水印生成器示例。注意，Flink 附带了 BoundedOutOfOrdernessWatermarks，这是一个与下面所示的 BoundedOutOfOrdernessGenerator 工作类似的水印生成器。

Scala 代码:

```

/**
 * 这个生成器生成水印，假设元素按顺序到达，但只是在一定程度上。
 * 某个时间戳 t 的最新元素最多会在时间戳 t 的最早元素之后 n 毫秒到达。
 */
class BoundedOutOfOrdernessGenerator extends WatermarkGenerator[MyEvent] {

```

```

val maxOutOfOrderness = 3500L           // 3.5 seconds

var currentMaxTimestamp: Long = _

override def onEvent(element: MyEvent, eventTimestamp: Long, output: WatermarkOutput): Unit = {
    currentMaxTimestamp = max(eventTimestamp, currentMaxTimestamp)
}

override def onPeriodicEmit(output: WatermarkOutput): Unit = {
    // 将水印发送为当前最高时间戳减去无序界限 (out-of-orderness bound)
    output.emitWatermark(new Watermark(currentMaxTimestamp - maxOutOfOrderness - 1));
}
}

/**
 * 这个生成器生成的水印比处理时间滞后一定数量。
 * 它假设元素在有界延迟后到达 Flink。
 */
class TimeLagWatermarkGenerator extends WatermarkGenerator[MyEvent] {

    val maxTimeLag = 5000L           // 5 seconds

    override def onEvent(element: MyEvent, eventTimestamp: Long, output: WatermarkOutput): Unit = {
        // 不需要做任何事情，因为我们在处理时间
    }

    override def onPeriodicEmit(output: WatermarkOutput): Unit = {
        output.emitWatermark(new Watermark(System.currentTimeMillis() - maxTimeLag));
    }
}

```

Java 代码:

```

/**
 * 这个生成器生成水印，假设元素按照一定的顺序到达，但只是在一定程度上。
 * 某个时间戳 t 的最新元素最多会在时间戳 t 的最早元素之后 n 毫秒到达。
 */
public class BoundedOutOfOrdernessGenerator implements WatermarkGenerator<MyEvent> {

    private final long maxOutOfOrderness = 3500; // 3.5 seconds

    private long currentMaxTimestamp;

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

    @Override
    public void onPeriodicEmit(WatermarkOutput output) {
        // 将水印发送为当前最高时间戳减去无序界限 (out-of-orderness bound)
    }
}

```

```

        output.emitWatermark(new Watermark(currentMaxTimestamp - maxOutOfOrderness - 1));
    }
}

/**
 * 这个生成器生成的水印比处理时间滞后一定数量。
 * 它假设元素在有界延迟后到达 Flink。
 */
public class TimeLagWatermarkGenerator implements WatermarkGenerator<MyEvent> {

    private final long maxTimeLag = 5000;          // 5 seconds

    @Override
    public void onEvent(MyEvent event, long eventTimestamp, WatermarkOutput output) {
        // 不需要做任何事情，因为我们在处理时间
    }

    @Override
    public void onPeriodicEmit(WatermarkOutput output) {
        output.emitWatermark(new Watermark(System.currentTimeMillis() - maxTimeLag));
    }
}

```

编写 Punctuated WatermarkGenerator

标点水印生成器将观察事件流，并在看到携带水印信息的特殊元素时发出水印。下面的代码实现一个标点生成器，每当事件表明它带有某个标记时就会发出水印。

Scala 代码：

```

class PunctuatedAssigner extends WatermarkGenerator[MyEvent] {

    override def onEvent(element: MyEvent, eventTimestamp: Long): Unit = {
        if (event.hasWatermarkMarker()) {
            output.emitWatermark(new Watermark(event.getWatermarkTimestamp()))
        }
    }

    override def onPeriodicEmit(): Unit = {
        // 不需要做任何事情，因为我们会对上面的事件做出反应
    }
}

```

Java 代码：

```

public class PunctuatedAssigner implements WatermarkGenerator<MyEvent> {

    @Override
    public void onEvent(MyEvent event, long eventTimestamp, WatermarkOutput output) {
        if (event.hasWatermarkMarker()) {
            output.emitWatermark(new Watermark(event.getWatermarkTimestamp()));
        }
    }
}

```

```
}

@Override
public void onPeriodicEmit(WatermarkOutput output) {
    // 不需要做任何事情，因为我们会对上面的事件做出反应
}
}
```

这个方法简单明了，主要是有两个方法：

- ❑ **onEvent**：每个元素都会调用这个方法，如果我们想依赖每个元素生成一个水印，然后发射到下游(可选，就是看是否用 **output** 来收集水印)，我们可以实现这个方法。
- ❑ **onPeriodicEmit**：如果数据量比较大的时候，我们每条数据都生成一个水印的话，会影响性能，所以这里还有一个周期性生成水印的方法。这个水印的生成周期可以这样设置：
`env.getConfig().setAutoWatermarkInterval(5000L);`

注：在每个单独的事件上生成水印是可能的。但是，由于每个水印都会引起下游的一些计算，过多的水印会降低性能。

水印策略和 Kafka 连接器

当使用 Apache Kafka 作为数据源时，每个 Kafka 分区可能有一个简单的事件时间模式（升序时间戳或有界无序）。然而，当使用来自 Kafka 的流时，多个分区经常被并行地使用，交叉使用来自分区的事件并破坏每个分区的模式（这是 Kafka 的消费者客户端固有的工作方式）。

在这种情况下，可以使用 Flink 的 **Kafka-partition-aware** 水印生成。使用该特性，可以在 Kafka 消费者内部生成每个 Kafka 分区的水印，每分区（**per-partition**）的水印合并的方式与在流 **shuffle** 中合并水印的方式相同。

例如，如果事件时间戳按照 Kafka 分区严格升序，那么使用升序时间戳水印生成器生成每分区（**per-partition**）的水印将得到完美的整体水印。注意，我们在示例中没有提供 **TimestampAssigner**，而是使用 Kafka 记录本身的时间戳。

Scala 代码：

```
val kafkaSource = new FlinkKafkaConsumer[MyType]("myTopic", schema, props)
kafkaSource.assignTimestampsAndWatermarks(
    WatermarkStrategy.forBoundedOutOfOrderness(Duration.ofSeconds(20)))
```

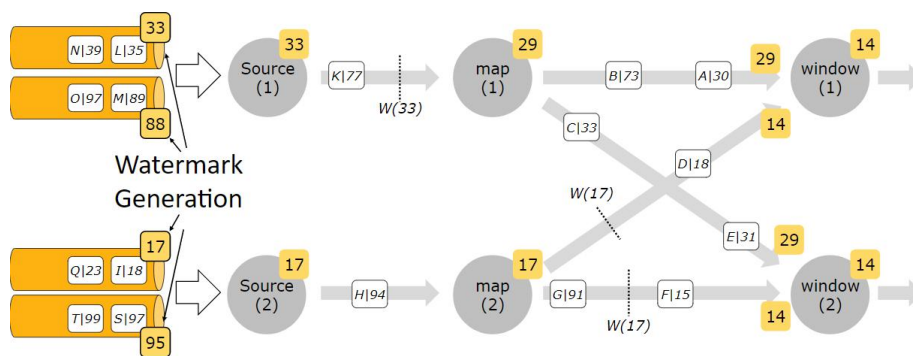
```
val stream: DataStream[MyType] = env.addSource(kafkaSource)
```

Java 代码：

```
FlinkKafkaConsumer<MyType> kafkaSource = new FlinkKafkaConsumer<>("myTopic", schema, props);
kafkaSource.assignTimestampsAndWatermarks(
    WatermarkStrategy.forBoundedOutOfOrderness(Duration.ofSeconds(20)));
```

```
DataStream<MyType> stream = env.addSource(kafkaSource);
```

下图展示了如何使用 **per-Kafka-partition** 的水印生成，以及在这种情况下，水印如何通过流数据流传播。



算子（运算符）如何处理水印？

作为一般规则，算子在将给定的水印转发到下游之前需要完全处理它。例如，WindowOperator 将首先计算应该被触发的所有窗口，并且只有在生成由水印触发的所有输出之后，水印本身才会被发送到下游。换句话说，由于水印的出现而产生的所有元素都将在水印之前发出。

同样的规则也适用于 TwoInputStreamOperator。然而，在这种情况下，算子的当前水印被定义为其两个输入的最小值。该行为的细节由 OneInputStreamOperator#processWatermark、twinputstreamoperator#processWatermark1 和 twinputstreamoperator #processWatermark2 方法的实现定义。

在摄取时间和处理时间的情况下，只需要指定时间特征，水印就会生成。请看下面的代码片段。

Scala 代码：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)
// 或
env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime)
```

Java 代码：

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);
// 或
env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime);
```

已弃用的 AssignerWithPeriodicWatermarks 和 AssignerWithPunctuatedWatermarks

在引入 WatermarkStrategy、TimestampAssigner 和 WatermarkGenerator 的当前抽象之前，Flink 使用的是 AssignerWithPeriodicWatermarks 和 AssignerWithPunctuatedWatermarks。仍然可以在 API 中看到它们，但建议使用新的接口，因为它们提供了更清晰的关注点分离，并统一了生成水印的 periodic 和 punctuated 样式。

3.6.6 内置水印生成器

正如上一节中所述，Flink 提供了抽象，允许程序员分配他们自己的时间戳并发出他们自己的水印。更具体地说，可以通过实现 WatermarkGenerator 接口来实现。

为了进一步简化此类任务的编程工作，Flink 附带了一些预先实现的时间戳赋值器。

- ☐ 单调递增时间戳
- ☐ 固定的迟到时间

单调递增时间戳

周期性水印生成的最简单特例是给定源任务所看到的时间戳按升序出现的情况。在这种情况下，当前时间戳总是可以作为水印，因为不会出现更早的时间戳。

请注意，只有每个并行数据源任务的时间戳是递增的才有必要。例如，如果在一个特定的设置中，一个 Kafka 分区被一个并行数据源实例读取，那么在每个 Kafka 分区中只需要时间戳升序就可以了。Flink 的水印合并机制将在并行流被 shuffle、union、connect 或 merge 时生成正确的水印。

Scala 代码：

```
WatermarkStrategy.forMonotonousTimestamps()
```

Java 代码：

```
WatermarkStrategy.forMonotonousTimestamps();
```

如果时间戳从不乱序，或者我们愿意将所有乱序事件视为迟到，这种情况下使用这个方法为具有单调递增时间戳的情况创建水印策略。

固定的迟到时间

周期性水印生成的另一个例子是当水印滞后于流中看到的最大(事件时间)时间戳时的固定数量的时间。这种情况涵盖在流中可能遇到的最大延迟是预先已知的场景，例如，当创建一个包含元素的自定义源时，这些元素的时间戳分布在一个固定的测试时间段内。对于这些情况，Flink 提供了 `BoundedOutOfOrdernessWatermarksGenerator`，它将 `maxOutOfOrderness` 作为参数。这个参数是一个 `Duration` 类型的时间间隔，也就是我们可以接受的最大的延迟时间，也就是当计算给定窗口的最终结果时，一个元素在被忽略之前允许延迟的最大时间。迟到 `lateness` 对应于 $t - t_w$ 的结果，其中 t 是元素的(事件时间)时间戳， t_w 是前一个水印的时间戳。如果 `lateness > 0`，则该元素被认为是迟来的，默认情况下，在计算其对应窗口的作业结果时被忽略。

例如，我们实现一个延迟 10 秒的固定延迟水印，可以这样做。

Scala 代码：

```
WatermarkStrategy.[T]forBoundedOutOfOrderness(Duration.ofSeconds(10))
```

Java 代码：

```
WatermarkStrategy.<T>forBoundedOutOfOrderness(Duration.ofSeconds(10));
```

如果我们处理的数据流中无序的时间戳是正常的，则使用 `forBoundedOutOfOrderness`，这个方法为记录乱序的情况创建水印策略，但可以设置事件乱序程度的上限。

3.6.7 分配时间戳和水印示例

下面通过一个示例来掌握如何为数据流中的事件分配时间戳和水印。

【示例】（Scala 实现）为数据流中的事件分配时间戳和水印。

请按以下步骤执行。

1、在 IntelliJ IDEA 中创建一个 Flink 项目，使用 `flink-quickstart-scala` 项目模板。（Flink 项目创建过程，请参见 2.2 节）

2、设置依赖。在 pom.xml 中添加如下依赖（根据项目模板创建的话，这个依赖会自动添加，此步可省略）：

Scala Maven 依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
```

3、创建流应用程序类。代码如下：

```
package com.xueai8.scala.ch03

import org.apache.flink.api.common.eventtime.WatermarkStrategy
import org.apache.flink.streaming.api.scala._

/**
 * 功能：为数据流中的事件分配时间戳和水印
 */
内置水印生成策略
为了方便开发，flink 提供了一些内置的水印生成方法供我们使用。
*/
object AssignerDemo2 {

  // case 类，表示流元素数据类型
  case class MessageInfo(hostname: String, status: String)

  // main 方法
  def main(args: Array[String]) {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 数据流源
    val stream = env.fromElements(
      MessageInfo("host1", "1234"),
      MessageInfo("host2", "2234"),
      MessageInfo("host3", "1234")
    )

    // 因为模拟数据没有时间戳，所以用此方法添加递增时间戳和水印
    // 为数据流中的元素分配时间戳，并生成标记以指示事件时间进展
    val withTimestampsAndWatermarks = stream.assignTimestampsAndWatermarks(
      WatermarkStrategy.forMonotonousTimestamps[MessageInfo]()
        .withTimestampAssigner((MessageInfo, ts) => System.currentTimeMillis())
    )
  }
}
```

```
)

// 将结果输出到控制台
withTimestampsAndWatermarks.print

// 执行
env.execute("Flink Watermark Strategy ")
}
```

【示例】（Java 实现）为数据流中的事件分配时间戳和水印。

请按以下步骤执行。

1、在 IntelliJ IDEA 中创建一个 Flink 项目，使用 `flink-quickstart-java` 项目模板。（Flink 项目创建过程，请参见 2.2 节）

2、设置依赖。在 `pom.xml` 中添加如下依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-java</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
```

3、创建事件数据结构。代码如下：

```
package com.xueai8.java.ch03;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * 功能：为数据流中的事件分配时间戳和水印
 */
public class AssignerDemo {

    public static class MessageInfo {
        public String hostname;
        public String status;

        public MessageInfo() {
        }

        public MessageInfo(String hostname, String status){
            this.hostname = hostname;
            this.status = status;
        }
    }
}
```

```

    @Override
    public String toString() {
        return "MessageInfo{" +
            "hostname=" + hostname + "\" +
            ", status=" + status + "\" +
            "}";
    }
}

public static void main(String[] args) throws Exception {
    // 设置流执行环境
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

    // 触发流程序执行
    env.execute("Flink Watermark Strategy");
}
}

```

4、创建流应用程序类。代码如下：

```

package com.xueai8.java.ch03;

import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
 * 功能：为数据流中的事件分配时间戳和水印
 */
public class AssignerDemo {

    public static class MessageInfo {
        public String hostname;
        public String status;

        public MessageInfo() {
        }

        public MessageInfo(String hostname, String status){
            this.hostname = hostname;
            this.status = status;
        }

        @Override
        public String toString() {
            return "MessageInfo{" +
                "hostname=" + hostname + "\" +
                ", status=" + status + "\" +
                "}";
        }
    }
}

```

```
public static void main(String[] args) throws Exception {
    // 设置流执行环境
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

    // 源数据流
    DataStream<MessageInfo> stream = env.fromElements(
        new MessageInfo("host1", "1234"),
        new MessageInfo("host2", "2234"),
        new MessageInfo("host3", "1234")
    );

    // 因为模拟数据没有时间戳，所以用此方法添加单调增加时间戳和水印
    DataStream<MessageInfo> withTimestampsAndWatermarks = stream
        .assignTimestampsAndWatermarks(WatermarkStrategy
            .<MessageInfo>forMonotonousTimestamps()
            .withTimestampAssigner((MessageInfo, ts) -> System.currentTimeMillis())
        );

    // 将结果输出到控制台
    withTimestampsAndWatermarks.print();

    // 触发流程序执行
    env.execute("Flink Watermark Strategy");
}
```

下面这个示例使用 4.2.4 节的自定义信用卡交易数据源，指定交易数据的交易时间 `timestamp` 字段做为事件时间，实现一个延迟 10 秒的固定延迟水印。

【示例】实现一个延迟 10 秒的固定延迟水印。

Scala 代码：

```
package com.xueai8.scala.ch03

import java.time.Duration

import com.xueai8.java.ch03.entity.Transaction
import com.xueai8.java.ch03.source.MyTransactionSource
import org.apache.flink.api.common.eventtime.{SerializableTimestampAssigner, WatermarkStrategy}
import org.apache.flink.streaming.api.scala._

/**
 * 功能：为信用卡交易数据流中的事件分配时间戳和水印
 * 使用自定义的 TransactionSource 数据源
 */
object AssignerDemo2 {

    def main(args: Array[String]) {
        // 设置流执行环境
        val env = StreamExecutionEnvironment
```

```

// 设置自定义数据源。参数 false 指定创建的是流数据源
val stream = env.addSource(new MyTransactionSource(false)).name("transactions")

// 因为模拟数据没有时间戳，所以用此方法添加事件时间戳和水印
/* 使用匿名内部类*/
val withTimestampsAndWatermarks = stream.assignTimestampsAndWatermarks(
    WatermarkStrategy.forBoundedOutOfOrderness[Transaction](Duration.ofMillis(2000))
        .withTimestampAssigner(new SerializableTimestampAssigner[Transaction]() {
            // 抽取交易事件中的 timestamp 字段做为事件时间戳
            override def extractTimestamp(t: Transaction, ts: Long): Long = {
                t.timestamp
            }
        })
)

// 将结果输出到控制台
stream.print

// 执行
env.execute("Flink Watermark Strategy")
}

```

Java 代码:

```

package com.xueai8.java.ch03;

import com.xueai8.java.ch03.entity.Transaction;
import com.xueai8.java.ch03.source.MyTransactionSource;
import org.apache.flink.api.common.eventtime.SerializableTimestampAssigner;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.time.Duration;

/**
 * 功能：为信用卡交易数据流中的事件分配时间戳和水印
 * 使用自定义的 TransactionSource 数据源
 */
public class AssignerDemo2 {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 设置自定义数据源。参数 false 指定创建的是流数据源
        DataStream<Transaction> stream =
            env.addSource(new MyTransactionSource(false)).name("transactions");

        // 因为模拟数据没有时间戳，所以用此方法添加单调增加时间戳和水印
        /* 使用匿名内部类

```

```

        DataStream<Transaction> withTimestampsAndWatermarks =
            stream.assignTimestampsAndWatermarks(
                WatermarkStrategy.<Transaction>forBoundedOutOfOrderness(Duration.ofSeconds(10))
                    .withTimestampAssigner(new SerializableTimestampAssigner<Transaction>() {
                        @Override
                        public long extractTimestamp(Transaction transaction, long ts) {
                            return transaction.timestamp;
                        }
                    })
            );
    }
    /*
    // 或者使用匿名函数 (Lambda 函数)
    DataStream<Transaction> withTimestampsAndWatermarks =
        stream.assignTimestampsAndWatermarks(
            WatermarkStrategy.<Transaction>forBoundedOutOfOrderness(Duration.ofSeconds(10))
                .withTimestampAssigner((transaction, ts) -> transaction.timestamp)
        );

    // 将结果输出到控制台
    withTimestampsAndWatermarks.print();

    // 触发程序执行
    env.execute("Flink Watermark Strategy");
}
}

```

下面这个示例重构 4.2.4 节的自定义信用卡交易数据源，在数据源中分配事件时间戳和生产水印。

【示例】在自定义信用卡交易数据源使用水印策略。(Java 实现)

```

package com.xueai8.java.ch03.source;

import com.xueai8.java.ch03.entity.Transaction;
import org.apache.flink.streaming.api.functions.source.RichSourceFunction;
import org.apache.flink.streaming.api.watermark.Watermark;

import java.sql.Timestamp;
import java.util.Arrays;
import java.util.List;

/**
 * Created by www.xueai8.com
 *
 * 自定义数据源，继承自 SourceFunction
 * 自带时间戳和水印的数据源
 */
public class MyTransactionSource2 extends RichSourceFunction<Transaction> {
    private static final long serialVersionUID = 1L;

    private static final Timestamp INITIAL_TIMESTAMP = Timestamp.valueOf("2020-01-01 00:00:00");
    private static final long SIX_MINUTES = 6 * 60 * 1000;
}

```

```
private final boolean bounded;      // 标志变量，指示生成流数据还是批数据

private int index = 0;              // 交易记录的索引
private long timestamp;             // 交易发生的时间戳

private volatile boolean isRunning = true;
private List<Transaction> data = null;

public MyTransactionSource2(boolean bounded){
    this.bounded = bounded;
    this.timestamp = INITIAL_TIMESTAMP.getTime();

    // 事先存储的信用卡交易数据，在实际中来自于外部数据源系统（如 Kafka）
    data = Arrays.asList(
        new Transaction(1, 0L, 188.23),
        new Transaction(2, 0L, 374.79),
        new Transaction(3, 0L, 112.15),
        new Transaction(4, 0L, 478.75),
        new Transaction(5, 0L, 208.85),
        new Transaction(1, 0L, 379.64),
        new Transaction(2, 0L, 351.44),
        new Transaction(3, 0L, 320.75),
        new Transaction(4, 0L, 259.42),
        new Transaction(5, 0L, 273.44),
        new Transaction(1, 0L, 267.25),
        new Transaction(2, 0L, 397.15),
        new Transaction(3, 0L, 0.219),
        new Transaction(4, 0L, 231.94),
        new Transaction(5, 0L, 384.73),
        new Transaction(1, 0L, 419.62),
        new Transaction(2, 0L, 412.91),
        new Transaction(3, 0L, 0.77),
        new Transaction(4, 0L, 22.10),
        new Transaction(5, 0L, 377.54),
        new Transaction(1, 0L, 375.44),
        new Transaction(2, 0L, 230.18),
        new Transaction(3, 0L, 0.80),
        new Transaction(4, 0L, 350.89),
        new Transaction(5, 0L, 127.55),
        new Transaction(1, 0L, 483.91),
        new Transaction(2, 0L, 228.22),
        new Transaction(3, 0L, 871.15),
        new Transaction(4, 0L, 64.19),
        new Transaction(5, 0L, 79.43),
        new Transaction(1, 0L, 56.12),
        new Transaction(2, 0L, 256.48),
        new Transaction(3, 0L, 148.16),
        new Transaction(4, 0L, 199.95),
        new Transaction(5, 0L, 252.37),
        new Transaction(1, 0L, 274.73),
```

```
        new Transaction(2, 0L, 473.54),
        new Transaction(3, 0L, 119.92),
        new Transaction(4, 0L, 323.59),
        new Transaction(5, 0L, 353.16),
        new Transaction(1, 0L, 211.90),
        new Transaction(2, 0L, 280.93),
        new Transaction(3, 0L, 347.89),
        new Transaction(4, 0L, 459.86),
        new Transaction(5, 0L, 82.31),
        new Transaction(1, 0L, 373.26),
        new Transaction(2, 0L, 479.83),
        new Transaction(3, 0L, 454.25),
        new Transaction(4, 0L, 83.64),
        new Transaction(5, 0L, 292.44)
    );
}

@Override
public void run(SourceContext<Transaction> sourceContext) throws Exception {
    while(this.isRunning && this.hasNext()) {
        Transaction transaction = this.next();
        sourceContext.collectWithTimestamp(transaction, transaction.timestamp); // 带时间戳
        // sourceContext.collect(transaction); // 不带时间戳
        sourceContext.emitWatermark(new Watermark(transaction.timestamp + 20000)); // 发出水印
    }
}

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

private boolean hasNext() {
    // 如果还有数据
    if (index < data.size()) {
        return true;
    }
    // 如果是用于生成批数据
    else if(bounded){
        return false;
    }
    // 如果是用于生成流数据，从头循环
    else {
        index = 0;
        return true;
    }
}

// 生成下一个交易数据，交易时间相隔 6 分钟
```



```
private Transaction next() {
    try {
        Thread.sleep(500);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    Transaction transaction = data.get(index++);
    transaction.timestamp = timestamp;
    timestamp += SIX_MINUTES;    // 交易时间间隔六分钟
    return transaction;
}
```

测试程序。

```
package com.xueai8.java.ch03;

import com.xueai8.java.ch03.entity.Transaction;
import com.xueai8.java.ch03.source.MyTransactionSource;
import com.xueai8.java.ch03.source.MyTransactionSource2;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.time.Duration;

/**
 * 功能：为信用卡交易数据流中的事件分配时间戳和水印
 * 使用自定义带时间戳和水印的 MyTransactionSource2 数据源
 */
public class AssignerDemo3 {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 设置自定义数据源。参数 false 指定创建的是流数据源
        DataStream<Transaction> stream =
            env.addSource(new MyTransactionSource2(false)).name("transactions");

        // 将结果输出到控制台
        stream.print();

        // 触发流程序执行
        env.execute("Flink Watermark Strategy");
    }
}
```

3.7 窗口操作

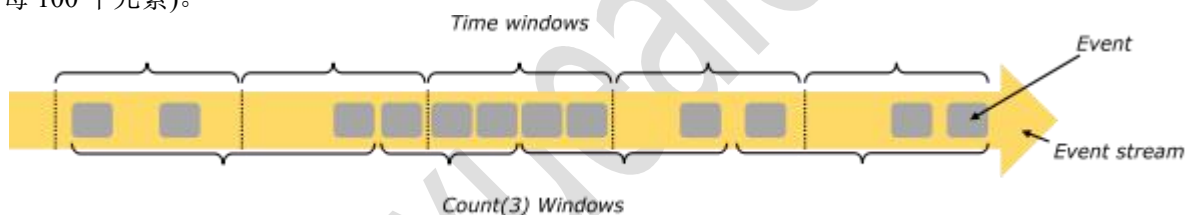
在进行流处理时，很自然的，我们想要计算流的有界子集的聚合分析。例如回答以下问题：

- ❑ 每分钟的页面浏览（pv）次数
- ❑ 每用户每周的会话次数
- ❑ 每分钟每传感器的最高温度
- ❑ 当电商中发布一个秒杀活动时，想要每隔 10 分钟了解流量数据
- ❑

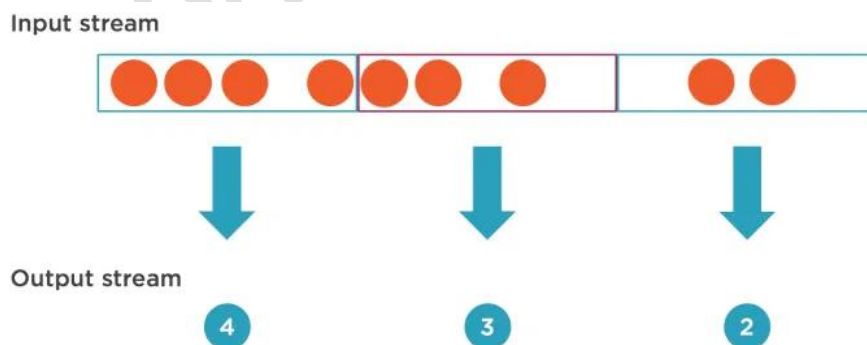
显然，要回答这些问题，我们需要处理元素组，而不是单个的元素。因此，通常使用窗口来限定在数据流上的聚合(count、sum 等)的范围，例如“过去 5 分钟内的计数”或“最后 100 个元素的总和”。因此，在处理流时，通常更有意义的是考虑有限窗口上的聚合，而不是整个流。Flink 具有非常富有表现力的窗口语义。

3.7.1 理解 Flink 窗口概念

窗口(window)是处理无限流的核心，使用窗口计算无界流上的聚合。窗口将流分割为有限大小的组，我们可以在这样的组上面应用计算。窗口可以是时间驱动的(例如，每 30 秒)，也可以是数据驱动的(例如，每 100 个元素)。



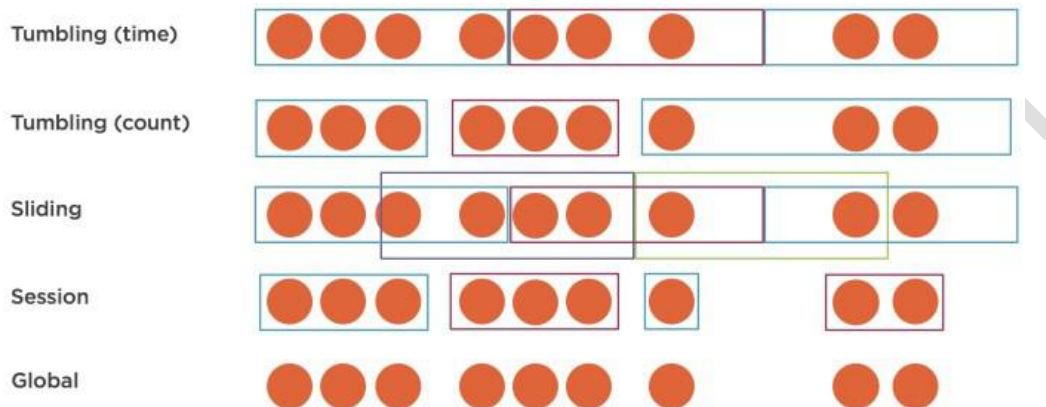
简而言之，流窗口允许我们对流中的元素进行分组，并在每个组上执行用户定义的函数。这个用户定义的函数可以返回 0、1 或多个元素，通过这种方式，它创建了一个新的流，我们可以在单独的系统中处理或存储这个流。



如何对流中的元素进行分组？Flink 支持不同类型的窗口。

- ❑ 滚动窗口：tumbling window，是在流中创建不重叠的相邻窗口。它们是固定长度的窗口，没有重叠。我们可以根据时间对元素进行分组(比如，从 10:00 到 10:05 的所有元素进入一个组)，或者根据计数(前 50 个元素进入一个单独的组)对元素进行分组。例如，我们可以用它来回答这样的问题：“在不重叠的五分钟间隔内计算流中元素的数量”。

- ❑ 滑动窗口: sliding window, 类似于滚动窗口, 但是窗口可以重叠。滑动窗口是固定长度的窗口, 通过用户给定的窗口滑动参数与前面的窗口重叠。例如, 如果需要计算最后 5 分钟的指标, 但我们希望每分钟显示一个输出时。
- ❑ 会话窗口: session window, 当对发生的事件进行分组, 时间接近的分到一组 (即一个窗口中)。还可以提供会话间隔的配置参数, 该参数指示在关闭会话之前需要等待多长时间。
- ❑ 全局窗口: global window, Flink 将所有元素放到一个窗口中。通常在这种情况下, 每个元素都被分配给一个单一的 per-key 全局窗口 (per-key global Window)。如果我们不指定任何触发器, 就不会触发任何计算。这只有在定义自定义触发器时才有用, 该触发器定义了窗口何时结束。



使用 Flink 计算窗口分析依赖于两个主要的抽象: 将事件分配给窗口的窗口赋值器(必要时创建新的窗口对象), 以及应用于分配给窗口的事件的窗口函数。Flink 的窗口 API 还有两个概念: 触发器(Triggers), 它决定何时调用窗口函数; 以及清除器(Evictor), 它可以删除窗口中收集的元素。

当应该属于某个窗口的第一个元素到达时, 就会创建一个窗口, 当时间 (事件时间或处理时间) 超过窗口的结束时间戳加上用户指定的允许延迟时, 该窗口将被完全删除。Flink 保证只针对基于时间的窗口删除, 而不针对其他类型, 例如全局窗口。比如, 一个基于事件时间的窗口策略为每 5 分钟创建一个非重叠窗口, 并且允许迟到 1 分钟, 那么 Flink 将创建一个 12:00-12:05 间隔的新窗口 (当带有落在该时间窗口的时间戳的第一个元素在这个间隔到来时), 当水印通过 12:06 时间戳时它会删除该窗口。

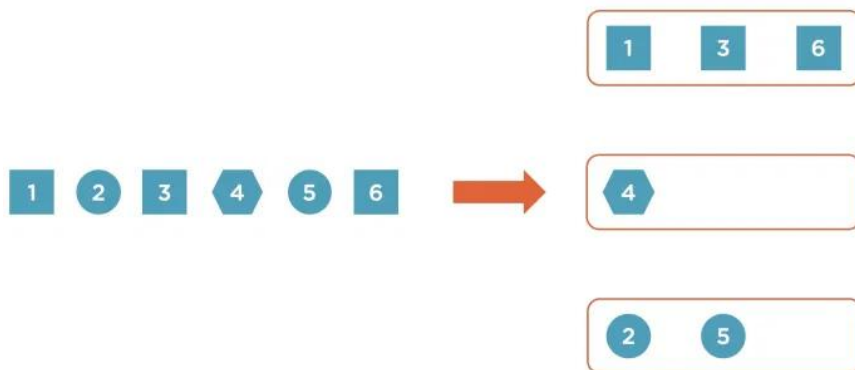
此外, 每个窗口都将有一个触发器 (Trigger) 和一个附加到它的函数 (ProcessWindowFunction、ReduceFunction、AggregateFunction 或 FoldFunction)。附加函数将包含要应用于窗口内容的计算, 而触发器指定触发窗口计算 (应用函数) 的条件。触发策略可能类似于 “当窗口中的元素数量超过 4 个时” 或者 “当水印经过窗口末端时”。触发器还可以决定在创建和删除窗口之间的任何时间清除窗口的内容。在这种情况下, 清除仅指的是窗口中的元素, 而不是窗口元数据。这意味着仍然可以向该窗口添加新数据。

除此之外, 还可以指定一个清除器(Evictor), 它能够在触发器触发之后、函数应用之前和/或之后从窗口中删除元素。

除了选择如何将元素分配给不同的窗口之外, 我们还需要选择一个流类型。Flink 的窗口支持两种流类型:

- ❑ **Non-keyed stream:** 在这种情况下, 流中的所有元素将一起被处理, 我们的用户定义函数将访问流中的所有元素。这种流类型的缺点是它不提供并行性, 集群中只有一台机器能够执行我们的代码。

- ❑ **Keyed stream:** 使用这种流类型，Flink 将通过一个 key（例如，设备的 ID）将单个流划分为多个独立的流。当处理 keyed stream 中的窗口时，定义的函数只能访问具有相同 key 的项。但是使用多个独立的流可以让 Flink 并行化工作。



必须在定义窗口之前指定要处理的流是否应该被赋予 key。使用 `keyBy(...)` 将把无限流分割成逻辑 keyed 流。如果没有调用 `keyBy(...)`，则说明流是 non-keyed 流。

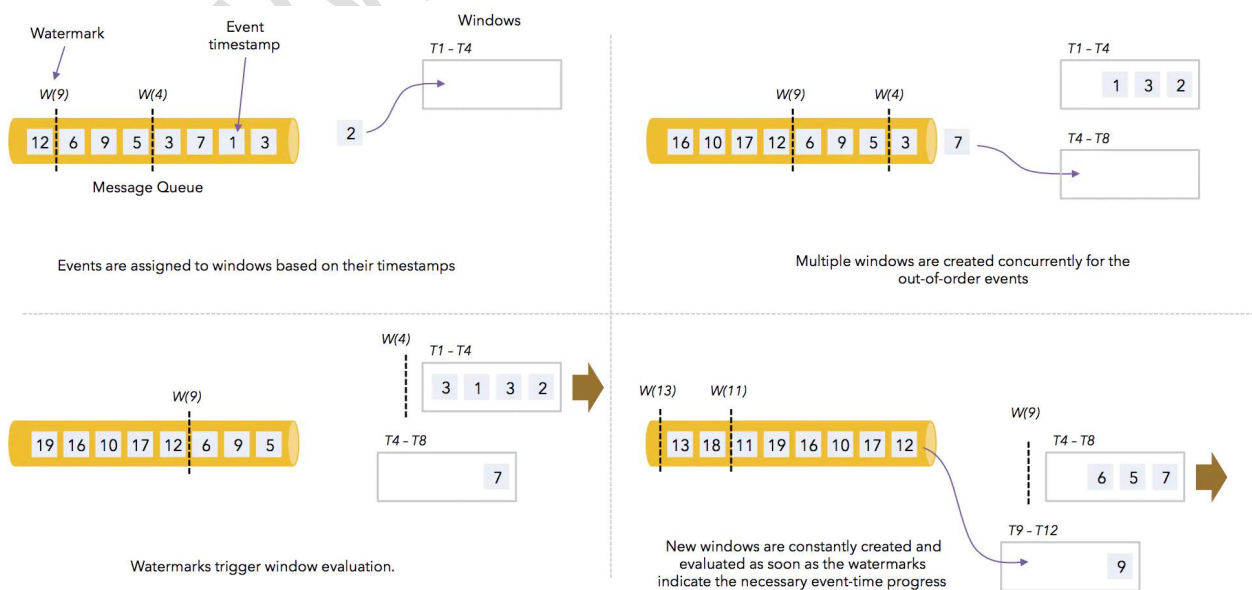
对于 keyed 流，传入事件的任何属性都可以用作 key。这时窗口计算可以由多个任务并行执行，因为每个逻辑 keyed 流都可以独立于其他流进行处理。所有引用相同 key 的元素将被发送到相同的并行任务进行处理。

在 non-keyed 流的情况下，原始流将不会被分割成多个逻辑流，所有的窗口逻辑将由一个任务执行，即并行度为 1。

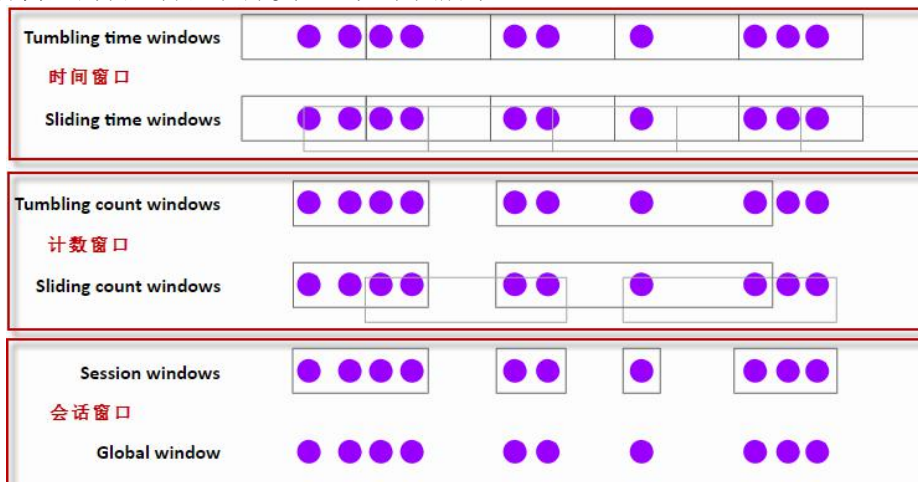
3.7.2 窗口分配器

在指定流是否为 keyed 流之后，下一步是定义一个窗口分配程序（窗口分配器）。窗口分配器定义如何将元素分配给窗口。这是通过在调用 `window(...)`（针对 keyed stream）或 `windowAll()`（针对 non-keyed stream）时指定所选择的 `WindowAssigner` 来实现的。

`WindowAssigner` 负责将每个传入元素分配给一个或多个窗口。下面的图是一个乱序的基于事件时间的数据流窗口分配示例。



Flink 有几种内置的窗口分配程序类型，如下图所示：



Flink 为最常见的场景(即滚动时间窗口、滑动时间窗口、全局窗口和会话窗口)提供了预定义的窗口分配器。

- ❑ 滚动时间窗口：例如，每分钟 PV 数据（浏览量）

```
TumblingEventTimeWindows.of(Time.minutes(1))
```

- ❑ 滑动时间窗口：例如，每 10 秒计算一次每分钟的页面浏览量

```
SlidingEventTimeWindows.of(Time.minutes(1), Time.seconds(10))
```

- ❑ 会话窗口：例如，每个会话的 PV 数据，其中会话定义为会话之间至少 30 分钟的间隔

```
EventTimeSessionWindows.withGap(Time.minutes(30))
```

可以使用 `Time.milliseconds(n)`、`Time.seconds(n)`、`Time.minutes(n)`、`Time.hours(n)` 和 `Time.days(n)` 中的一个来指定持续时间。

所有内置的窗口分配器（全局窗口除外）都根据时间向窗口分配元素。基于时间的窗口分配程序（包括会话窗口）有事件时间和处理时间两种形式。在使用这两种时间窗口时需要进行权衡。如果使用处理时间窗口，则必须受到如下限制：

- ❑ 不能处理历史数据
- ❑ 不能正确处理乱序数据
- ❑ 结果将是不确定的

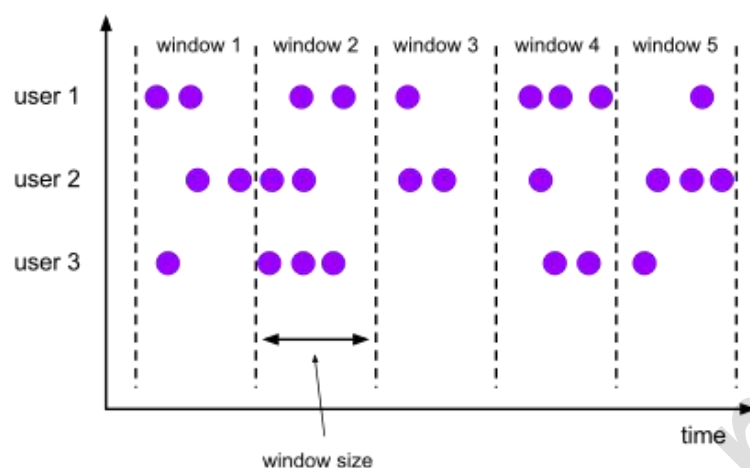
但使用处理时间窗口会具有较低的延迟。

基于时间的窗口有一个开始时间戳(包括)和一个结束时间戳(不包括)，它们共同描述窗口的大小。在处理基于时间的窗口时，Flink 使用 `TimeWindow` 类型，该类型具有查询窗口开始和结束时间戳的方法，以及 `maxTimestamp()` 方法，该方法返回给定窗口允许的最大时间戳。

接下来，我们将展示 Flink 预定义的窗口分配程序是如何工作的，以及如何在 `DataStream` 程序中使用它们。

滚动窗口

滚动窗口分配器将每个元素分配给具有指定窗口大小的窗口。滚动窗口的大小是固定的，不会重叠。例如，如果指定一个大小为 5 分钟的滚动窗口，将计算当前窗口并每 5 分钟启动一个新窗口，如下图所示。



下面的代码片段展示了如何使用滚动窗口。

Scala 代码:

```
val input: DataStream[T] = ...
```

// 滚动事件时间窗口

```
input
```

```
  .keyBy(<key selector>)
  .window(TumblingEventTimeWindows.of(Time.seconds(5)))
  .<windowed transformation>(<window function>)
```

// 滚动处理时间窗口

```
input
```

```
  .keyBy(<key selector>)
  .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
  .<windowed transformation>(<window function>)
```

// 每天滚动事件时间窗口，偏移值为-8 小时

```
input
```

```
  .keyBy(<key selector>)
  .window(TumblingEventTimeWindows.of(Time.days(1), Time.hours(-8)))
  .<windowed transformation>(<window function>)
```

Java 代码:

```
DataStream<T> input = ...;
```

// 滚动事件时间窗口 - 基于事件时间

```
input
```

```
  .keyBy(<key selector>)
  .window(TumblingEventTimeWindows.of(Time.seconds(5)))
  .<windowed transformation>(<window function>);
```

// 滚动处理时间窗口 - 基于处理时间

```
input
```

```
  .keyBy(<key selector>)
```



```

.window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
.<windowed transformation>(<window function>);

// 每天滚动事件时间窗口，偏移值为-8 小时
input
  .keyBy(<key selector>)
  .window(TumblingEventTimeWindows.of(Time.days(1), Time.hours(-8)))
  .<windowed transformation>(<window function>);

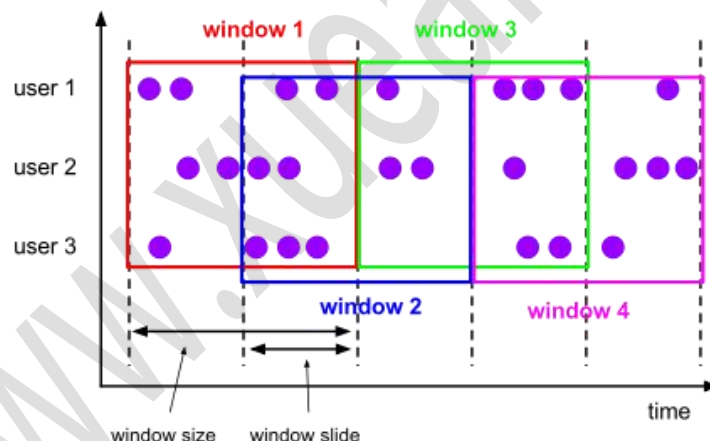
```

如上一个示例所示，滚动窗口分配程序还接受一个可选的偏移参数，该参数可用于更改窗口的对齐方式。例如，如果没有偏移量，每小时滚动窗口都与 epoch 对齐，也就是说我们将得到诸如 1:00:00.000 - 1:59:59.999、2:00:00.000 - 2:59:59.999 等窗口。如果你想改变，你可以给一个偏移量。例如，如果偏移时间为 15 分钟，我们将得到 1:15:00.000 - 2:14:59.999、2:15:00.000 - 3:14:59.999 等等。偏移量的一个重要用例是将窗口调整到 UTC-0 之外的时区。例如，在中国，必须指定 `Time.hours(-8)` 偏移量。

滑动窗口

滑动窗口分配器将元素分配给固定长度的窗口。类似于滚动窗口分配程序，窗口的大小由窗口大小参数配置。附加的窗口滑动参数控制滑动窗口启动的频率。因此，如果滑动窗口小于窗口大小，则滑动窗口可以重叠。在这种情况下，元素被分配到多个窗口。

例如，假设有一个 10 分钟大小的窗口，它可以滑动 5 分钟。这样，每 5 分钟就会得到一个包含过去 10 分钟内到达的事件的窗口，如下图所示。



下面的代码片段展示了如何使用滑动窗口。

Scala 代码:

```

val input: DataStream[T] = ...

// 滑动事件时间窗口
input
  .keyBy(<key selector>)
  .window(SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(5)))
  .<windowed transformation>(<window function>)

// 滑动处理时间窗口
input
  .keyBy(<key selector>)

```

```
.window(SlidingProcessingTimeWindows.of(Time.seconds(10), Time.seconds(5)))
.<windowed transformation>(<window function>)

// 滑动处理时间窗口，偏移-8 个小时
input
  .keyBy(<key selector>)
  .window(SlidingProcessingTimeWindows.of(Time.hours(12), Time.hours(1), Time.hours(-8)))
  .<windowed transformation>(<window function>)
```

Java 代码:

```
DataStream<T> input = ...;

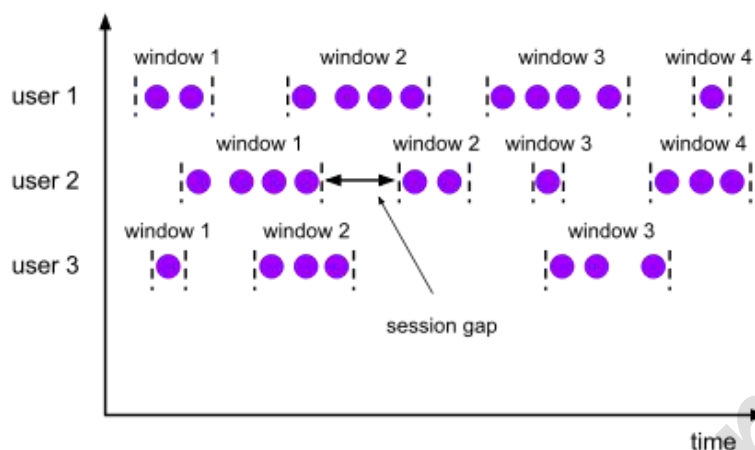
// 滑动事件时间窗口 - 基于事件时间滑动
input
  .keyBy(<key selector>)
  .window(SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(5)))
  .<windowed transformation>(<window function>);

// 滑动处理时间窗口 - 基于处理时间滑动
input
  .keyBy(<key selector>)
  .window(SlidingProcessingTimeWindows.of(Time.seconds(10), Time.seconds(5)))
  .<windowed transformation>(<window function>);

// 滑动处理时间窗口，偏移-8 个小时
input
  .keyBy(<key selector>)
  .window(SlidingProcessingTimeWindows.of(Time.hours(12), Time.hours(1), Time.hours(-8)))
  .<windowed transformation>(<window function>);
```

会话窗口

会话窗口分配器根据活动会话对元素进行分组。与翻滚窗口和滑动窗口相比，会话窗口不会重叠，也没有固定的开始和结束时间。相反，会话窗口在一段时间内不接收元素时关闭，即，当一段不活跃的间隔发生时。会话窗口分配程序可以配置为静态会话间隔，也可以配置会话间隔提取器函数，该函数定义不活动的周期有多长。当此期间过期时，当前会话将关闭，随后的元素将分配给新的会话窗口。



下面的代码片段展示了如何使用会话窗口。

Scala 代码:

```
val input: DataStream[T] = ...

// 事件时间会话窗口，带有静态间隔
input
  .keyBy(<key selector>)
  .window(EventTimeSessionWindows.withGap(Time.minutes(10)))
  .<windowed transformation>(<window function>)

// 事件时间会话窗口，带有动态间隔
input
  .keyBy(<key selector>)
  .window(EventTimeSessionWindows.withDynamicGap(new SessionWindowTimeGapExtractor[String] {
    override def extract(element: String): Long = {
      // 确定并返回会话间隔
    }
  }))
  .<windowed transformation>(<window function>)

// 处理时间会话窗口，带有静态间隔
input
  .keyBy(<key selector>)
  .window(ProcessingTimeSessionWindows.withGap(Time.minutes(10)))
  .<windowed transformation>(<window function>)

// 处理时间会话窗口，带有动态间隔
input
  .keyBy(<key selector>)
  .window(DynamicProcessingTimeSessionWindows.withDynamicGap(new
SessionWindowTimeGapExtractor[String] {
    override def extract(element: String): Long = {
      // 确定并返回会话间隔
    }
  }
  )
  )
```

```
    )))  
    .<windowed transformation>(<window function>)
```

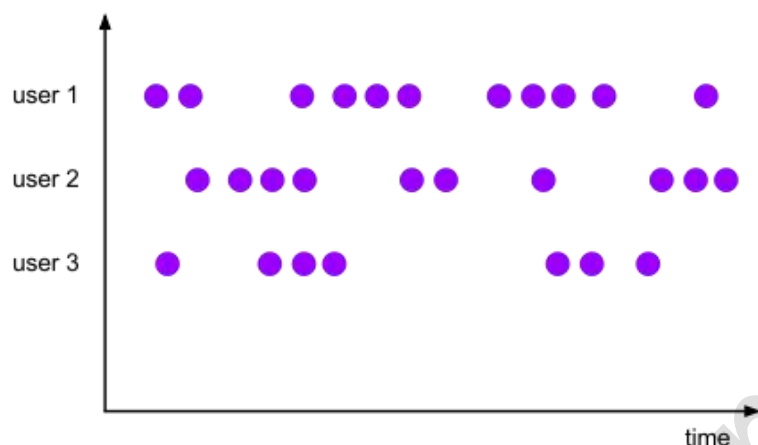
Java 代码:

```
DataStream<T> input = ...;  
  
// 事件时间会话窗口，带有静态间隔  
input  
    .keyBy(<key selector>)  
    .window(EventTimeSessionWindows.withGap(Time.minutes(10)))  
    .<windowed transformation>(<window function>);  
  
// 事件时间会话窗口，带有动态间隔  
input  
    .keyBy(<key selector>)  
    .window(EventTimeSessionWindows.withDynamicGap((element) -> {  
        // 确定并返回会话间隔  
    })))  
    .<windowed transformation>(<window function>);  
  
// 处理时间会话窗口，带有静态间隔  
input  
    .keyBy(<key selector>)  
    .window(ProcessingTimeSessionWindows.withGap(Time.minutes(10)))  
    .<windowed transformation>(<window function>);  
  
// 处理时间会话窗口，带有动态间隔  
input  
    .keyBy(<key selector>)  
    .window(ProcessingTimeSessionWindows.withDynamicGap((element) -> {  
        // 确定并返回会话间隔  
    })))  
    .<windowed transformation>(<window function>);
```

可以使用 `Time.milliseconds(x)`, `Time.seconds(x)`, `Time.minutes(x)`, `Time.hours(n)` 和 `Time.days(n)` 等指定静态间隔。动态间隔是通过实现 `SessionWindowTimeGapExtractor` 接口指定的。

全局窗口

全局窗口分配程序将具有相同 `key` 的所有元素分配给同一个全局窗口。只有在使用自定义触发器执行自定义窗口时，此窗口模式才有用。否则，将不执行任何计算，因为全局窗口没有一个自然的结束，我们不能在此结束处位置处理聚合的元素。在许多情况下，最好使用 `ProcessFunction` 来代替。



下面的代码片段展示了如何使用全局窗口。

Scala 代码:

```
val input: DataStream[T] = ...
```

```
input
```

```
  .keyBy(<key selector>)
  .window(GlobalWindows.create())
  .<windowed transformation>(<window function>)
```

Java 代码:

```
DataStream<T> input = ...;
```

```
input
```

```
  .keyBy(<key selector>)
  .window(GlobalWindows.create())
  .<windowed transformation>(<window function>);
```

特别注意:

1) 滑动窗口分配器可以创建许多窗口对象, 并将每个事件复制到每个相关窗口。例如, 如果每隔 15 分钟就有一个 24 小时的滑动窗口, 那么每个事件都将被复制到 $4 * 24 = 96$ 个窗口中。

2) 窗口时间使用的是 Epoch 时间 (纪元时间), 指的是一个特定的时间: 1970-01-01 00:00:00 UTC。比如, 如果使用了一小时的处理时间窗口, 并且在 12:05 启动应用程序, 那么第一个窗口并不会在 1:05 关闭。第一个窗口将有 55 分钟长, 在 1 点关闭。

3) 窗口后面可以再跟窗口。例如:

```
stream
```

```
  .keyBy(t -> t.key)
  .window(<time specification>)
  .reduce(<reduce function>)
  .windowAll(<same time specification>)
  .reduce(<same reduce function>)
```

4) 空时间窗口没有结果。只有当事件被分配给窗口时, 才会创建窗口。因此, 如果在给定的时间范围内没有事件, 则不会报告结果。

5) 迟来的事件会导致迟来的合并。会话窗口基于可以合并的窗口抽象。每个元素最初被分配给一个新窗口，之后，当窗口之间的间隔足够小时，就会合并窗口。通过这种方式，一个延迟的事件可以填补之前两个独立会话之间的空白，从而产生一个延迟的合并。

6) 还可以通过继承 `WindowAssigner` 类来实现自定义窗口分配程序。

Flink 窗口操作示例

一个 Flink 窗口程序的总体结构如下所示。

Keyed Windows

```
stream
  .keyBy(...)           <- keyed versus non-keyed windows
  .window(...)          <- required: "assigner"
  [.trigger(...)]       <- optional: "trigger" (else default trigger)
  [.evictor(...)]       <- optional: "evictor" (else no evictor)
  [.allowedLateness(...)] <- optional: "lateness" (else zero)
  [.sideOutputLateData(...)] <- optional: "output tag" (else no side output for late data)
  .reduce/aggregate/apply() <- required: "function"
  [.getSideOutput(...)] <- optional: "output tag"
```

Non-Keyed Windows

```
stream
  .windowAll(...)       <- required: "assigner"
  [.trigger(...)]       <- optional: "trigger" (else default trigger)
  [.evictor(...)]       <- optional: "evictor" (else no evictor)
  [.allowedLateness(...)] <- optional: "lateness" (else zero)
  [.sideOutputLateData(...)] <- optional: "output tag" (else no side output for late data)
  .reduce/aggregate/apply() <- required: "function"
  [.getSideOutput(...)] <- optional: "output tag"
```

其中第一个片段代表 `keyed stream`，第二个片段代表 `non-keyed stream`。可以看到，惟一的区别是为 `keyed stream` 调用 `keyBy(...)` 函数，而为 `non-keyed stream` 调用 `windowAll(...)` 函数。

在 `keyed stream` 的情况下，可以使用传入事件的任何属性作为 `key`。在 `keyed stream` 的窗口计算由多个任务并行执行，因为每个逻辑 `keyed stream` 都可以独立于其他流进行处理。所有引用相同 `key` 的元素将被发送到相同的并行任务。在一个 `keyed stream` 流上应用窗口的基本形式如下：

```
stream.keyBy(<key selector>)
  .window(<window assigner>)
  .reduce|aggregate|process(<window function>)
```

在 `non-keyed stream` 的情况下，原始流不会被分割成多个逻辑流，所有的窗口逻辑将由一个任务执行，即并行度为 1（在这种情况下，处理将不会并行进行）。在一个 `non-keyed stream` 流上应用窗口的基本形式如下：

```
stream.windowAll(<window assigner>)
  .reduce|aggregate|process(<window function>)
```

下面我们来看一个滚动窗口示例。

【示例 1】计算每 5 秒内来自 web 套接字的字数。请按以下步骤操作。

1、在 IntelliJ IDEA 中创建一个 Flink 项目，使用 `flink-quickstart-scala/flink-quickstart-java` 项目模板。

2、打开项目中的 StreamingJob 对象文件，编辑流处理代码如下：

Scala 实现：

```
package com.xueai8.scala.ch03

import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, _}
import org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time

/**
 * Created by tongyao on 2019/6/5.
 * 窗口操作：窗口单词计数
 * Socket 数据源：流窗口单词计数应用程序示例，它在 5 秒内计算来自 web 套接字的单词。
 *
 * 要运行这个程序，首先从终端用 netcat 启动输入流： nc -lk 9999
 * 只要输入一些单词，用空格分开，按回车键输入
 */
object WindowOperatorDemo1 {
  def main(args: Array[String]): Unit = {
    val host = "192.168.190.133"
    val port = 9999

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val text = env.socketTextStream(host, port)

    val counts = text.flatMap { _.toLowerCase.split("\\W+").filter { _.nonEmpty } }
      .map { (_, 1) }
      .keyBy(_._1) // keyed stream
      .window(TumblingProcessingTimeWindows.of(Time.seconds(5))) // 5 秒滚动窗口
      .sum(1) // 对元组第二个字段求和

    counts.print()

    env.execute("Window Stream WordCount")
  }
}
```

Java 实现：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

/**
 * Created by tongyao on 2019/6/5.
 * 窗口操作：窗口单词计数
 */
```

```
* Socket 数据源: 流窗口单词计数应用程序示例, 它在 5 秒内计算来自 web 套接字的单词。
*
* 要运行这个程序, 首先从终端用 netcat 启动输入流:  nc -lk 9999
* 只要输入一些单词, 用空格分开, 按回车键输入
*/
public class WindowOperatorDemo1 {
    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 加载数据-转换-执行窗口操作
        // 读取 socket 数据源
        String host = "192.168.190.133";
        int port = 9999;

        DataStream<Tuple2<String, Integer>> dataStream = env
            .socketTextStream(host, port)
            .map(String::toLowerCase)
            .flatMap(new Splitter())
            .keyBy(t -> t.f0)                // keyed stream
            .window(TumblingProcessingTimeWindows.of(Time.seconds(5))) // 5 秒滚动窗口
            .sum(1);                        // 对元组第二个字段求和

        dataStream.print();

        env.execute("Window Stream WordCount");
    }

    // 自定义 flatMap 函数
    public static class Splitter implements FlatMapFunction<String, Tuple2<String, Integer>> {
        @Override
        public void flatMap(String sentence, Collector<Tuple2<String, Integer>> out) throws Exception {
            for (String word: sentence.split("\\W+")) {
                out.collect(new Tuple2<>(word, 1));
            }
        }
    }
}
```

3、另起一个终端窗口, 执行如下命令, 启动一个 Web Socket 服务器:

```
$ nc -l 9999
```

4、回到 Flink 代码文件, 并运行程序。

5、在 Socket 运行窗口, 随便输入一些内容, 以空格分隔。比如:

```
good good study
```

```
day day up
```

6、在 IntelliJ IDEA 的运行窗口, 可以观察到输出如下的统计结果:

```
(good,2)
(study,1)
(up,1)
(day,2)
```

下面我们来看一个滑动窗口示例。

【示例】使用滑动窗口，每 5 秒计算一次前 10 秒内来自 web 套接字的单词计数。

Scala 代码：

```
package com.xueai8.scala.ch03

import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.windowing.assigners.SlidingProcessingTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time

/**
 * Created by tongyao on 2019/6/17.
 * 窗口操作：滑动窗口
 * 每 5 秒钟计算一次过去 10 秒钟内来自 web 套接字的单词计数。
 */
object WindowOperatorDemo2 {
  def main(args: Array[String]): Unit = {
    val host = "192.168.190.133"
    val port = 9999

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val text = env.socketTextStream(host, port)

    val counts = text.flatMap { _.toLowerCase.split("\\W+").filter { _.nonEmpty } }
      .map { (_, 1) }
      .keyBy(_._1) // keyed stream
      .window(SlidingProcessingTimeWindows.of(Time.seconds(10), Time.seconds(5))) // 滑动窗口
      .sum(1) // 对元组第二个字段求和

    counts.print()

    env.execute("Window Stream WordCount")
  }
}
```

Java 代码：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.assigners.SlidingProcessingTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.util.Collector;

/**
 * Created by tongyao on 2019/6/17.
 * 使用滑动窗口，每 5 秒计算一次前 10 秒内来自 web 套接字的单词计数。
 */
```

```

public class WindowOperatorDemo2 {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 读取 socket 数据源
        String host = "192.168.190.133";
        int port = 9999;

        DataStream<Tuple2<String, Integer>> dataStream = env
            .socketTextStream(host, port)
            .map(String::toLowerCase)
            .flatMap(new Splitter())
            .keyBy(t -> t.f0) // keyed stream
            .window(SlidingProcessingTimeWindows.of(Time.seconds(10), Time.seconds(5))) // 滑动窗口
            .sum(1); // 对元组第二个字段求和

        dataStream.print();

        env.execute("Window Stream WordCount");
    }

    // 自定义 FlatMapFunction
    public static class Splitter implements FlatMapFunction<String, Tuple2<String, Integer>> {
        @Override
        public void flatMap(String sentence, Collector<Tuple2<String, Integer>> out) throws Exception {
            for (String word: sentence.split("\\W+")) {
                out.collect(new Tuple2<>(word, 1));
            }
        }
    }
}

```

【示例】使用 3.5.3 小节创建的自定义数据源，请统计每个用户每小时信用卡交易金额。

Scala 代码：

```

package com.xueai8.scala.ch03

import java.time.Duration

import com.xueai8.scala.ch03.entity.Transaction
import com.xueai8.scala.ch03.source.MyTransactionSource
import org.apache.flink.api.common.eventtime.{SerializableTimestampAssigner, WatermarkStrategy}
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows
import org.apache.flink.streaming.api.windowing.time.Time

/**
 * 使用滚动窗口，计算每个用户每小时信用卡交易金额。

```



```

*/
object WindowOperatorDemo3 {

  def main(args: Array[String]) {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 设置自定义数据源。参数 false 指定创建的是流数据源
    val stream = env.addSource(new MyTransactionSource(false)).name("transactions")

    // 因为模拟数据没有时间戳，所以用此方法添加事件时间戳和水印
    /* 使用匿名内部类*/
    stream.assignTimestampsAndWatermarks(
      WatermarkStrategy.forBoundedOutOfOrderness[Transaction](Duration.ofMillis(2000))
        .withTimestampAssigner(new SerializableTimestampAssigner[Transaction]() {
          // 抽取交易事件中的 timestamp 字段做为事件时间戳
          override def extractTimestamp(t: Transaction, ts: Long): Long = {
            t.timestamp
          }
        })
    ).map(t => (t.accountId,t.timestamp,t.amount))
    .keyBy(_._1) // keyed stream
    .window(TumblingEventTimeWindows.of(Time.seconds(60 * 60))) // 滚动窗口
    .sum(1) // 对 amount 字段求和
    .print

    // 执行
    env.execute("Flink Agg Demo")
  }
}

```

Java 代码:

```

package com.xueai8.java.ch03;

import com.xueai8.java.ch03.entity.Transaction;
import com.xueai8.java.ch03.source.MyTransactionSource;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

import java.time.Duration;

/**
 * 使用滚动窗口，计算每个用户每小时信用卡交易金额。
 */
public class WindowOperatorDemo3 {

  public static void main(String[] args) throws Exception {
    // 设置流执行环境

```

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

// 读取 socket 数据源
// 设置自定义数据源。参数 false 指定创建的是流数据源
DataStream<Transaction> stream =
    env.addSource(new MyTransactionSource(false)).name("transactions");

// 先应用水印策略，分配事件时间戳并指定 5 秒水印
stream.assignTimestampsAndWatermarks(
    WatermarkStrategy.<Transaction>forBoundedOutOfOrderness(Duration.ofSeconds(5))
        .withTimestampAssigner((transaction, ts) -> transaction.timestamp))
    .keyBy(t -> t.accountId) // keyed stream
    .window(TumblingEventTimeWindows.of(Time.seconds(60*60))) // 滚动窗口
    .sum("amount") // 对 amount 字段求和
    .print();

env.execute("Window Agg Demo");
}
```

通过前面的讲述和示例程序可知，使用 Flink 计算窗口分析依赖于两个主要的抽象：

- ❑ 将事件分配给窗口的窗口分配程序（Window Assigners，根据需要创建新的窗口对象）
- ❑ 应用于分配给窗口的事件的窗口函数（Window Functions）。

在上面的例子中，我们基于窗口执行的聚合函数来自于 KeyedStream，详细介绍请参考“3.3.6 聚合转换”一节。

3.7.3 窗口函数

定义了窗口分配器之后，我们需要指定要在每个窗口上执行的计算，也就是指定窗口函数。一旦系统确定某个窗口已准备好进行处理，该窗口函数将被用于处理窗口中的每个元素。

Flink 提供了如下几类窗口函数：

- ❑ ReduceFunction
- ❑ AggregateFunction
- ❑ ProcessWindowFunction

前两个窗口函数的执行效率更高，因为 Flink 可以在每个元素到达窗口时增量地聚合元素。而 ProcessWindowFunction 的执行效率没有前两个窗口函数的执行效率高，因为 Flink 在调用函数之前必须在内部缓冲窗口的所有元素，然后再以 Iterable 的形式传给它。

如何处理窗口内容，我们三个基本的选择：

- ❑ 作为一个批处理，使用 ProcessWindowFunction，它将被传递一个带有窗口内容的 Iterable；
- ❑ 增量处理，在每个事件分配给窗口时调用 ReduceFunction 或 AggregateFunction；
- ❑ 或者两者结合使用，其中 ReduceFunction 或 AggregateFunction 的预聚合结果在窗口被触发时提供给 ProcessWindowFunction。

接下来，我们通过对温度传感器数据的分析，来理解和常用 Flink 所提供的这几类窗口函数的应用。首先从准备数据源开始。

【示例】实现一个自定义的温度传感器数据源。

请按以下步骤操作。

1) 定义事件数据类型

首先创建一个 POJO 类，用来封装从传感器采集到的事件数据。

```
package com.xueai8.java.ch03.entity;

/**
 * Java POJO, 传感器读数
 */
public class SensorReading {

    public String id;          // 传感器 id
    public long timestamp;     // 读取时的时间戳
    public double temperature; // 读取到的温度值

    public SensorReading() {}

    public SensorReading(String id, long timestamp, double temperature) {
        this.id = id;
        this.timestamp = timestamp;
        this.temperature = temperature;
    }

    public String toString() {
        return "(" + this.id + ", " + this.timestamp + ", " + this.temperature + ")";
    }
}
```

2) 自定义数据源

然后自定义传感器数据源，继承自 RichSourceFunction 类。

```
package com.xueai8.java.ch03.source;

import com.xueai8.java.ch03.entity.SensorReading;
import org.apache.flink.streaming.api.functions.source.RichSourceFunction;

import java.util.Calendar;
import java.util.Random;

/**
 * 自定义数据源
 */
public class SensorSource extends RichSourceFunction<SensorReading> {

    // 标志变量，指示源是否仍在运行
    private boolean running = true;

    /** run()持续发射 SensorReadings, 通过 SourceContext. */
    @Override
    public void run(SourceContext<SensorReading> srcCtx) throws Exception {
```

```
// 初始化随机数发生器
Random rand = new Random();

// 查找此并行任务（task）的索引
int taskIdx = this.getRuntimeContext().getIndexOfThisSubtask();

// 初始化传感器 id 和温度
String[] sensorIds = new String[10];
double[] curFTemp = new double[10];
for (int i = 0; i < 10; i++) {
    sensorIds[i] = "sensor_" + (taskIdx * 10 + i);
    curFTemp[i] = 65 + (rand.nextGaussian() * 20);
}

while (running) {
    // 获取当前时间
    long curTime = Calendar.getInstance().getTimeInMillis();

    // 放射 SensorReadings
    for (int i = 0; i < 10; i++) {
        // 修改当前温度
        /* nextGaussian()方法
        用于获得下一个伪随机、高斯(“正态”)分布的 double 值，均值 0.0，标准偏差 1.0
        */
        curFTemp[i] += rand.nextGaussian() * 0.5;
        // 发射读数
        srcCtx.collect(new SensorReading(sensorIds[i], curTime, curFTemp[i]));
    }

    // 等待 100 ms
    Thread.sleep(100);
}

/** 取消这个 SourceFunction. */
@Override
public void cancel() {
    this.running = false;
}
}
```

3) 测试数据源

接下来编写流测试程序，输出每个传感器的事件数据。

```
package com.xueai8.java.ch03;

import com.xueai8.java.ch03.entity.SensorReading;
import com.xueai8.java.ch03.source.SensorSource;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

/**
```

```
* 传感器流测试程序
*/
public class SensorSourceTest {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 读取流数据源
        DataStream<SensorReading> input = env.addSource(new SensorSource());

        input.print();

        // 触发流程序执行
        env.execute("Flink Streaming Java API Skeleton");
    }
}
```

执行上面的代码，输出传感器事件如下：

```
7> (sensor_3, 1615348339214, 34.79321478683425)
2> (sensor_6, 1615348339214, 48.92444755311628)
8> (sensor_4, 1615348339214, 96.07384090788655)
5> (sensor_1, 1615348339214, 28.681991445026327)
4> (sensor_0, 1615348339214, 72.66948362432143)
6> (sensor_2, 1615348339214, 43.846775241880266)
1> (sensor_5, 1615348339214, 71.07351152250708)
3> (sensor_7, 1615348339214, 71.6346953171324)
4> (sensor_8, 1615348339214, 81.4481576590041)
5> (sensor_9, 1615348339214, 41.89889773458169)
4> (sensor_6, 1615348339329, 48.850093468561276)
7> (sensor_1, 1615348339329, 28.694854674250678)
1> (sensor_3, 1615348339329, 34.39378107798509)
2> (sensor_4, 1615348339329, 96.46584713976698)
6> (sensor_0, 1615348339329, 72.29067790666107)
3> (sensor_5, 1615348339329, 71.89650023573446)
6> (sensor_8, 1615348339329, 81.58017900080814)
5> (sensor_7, 1615348339329, 71.47770075217043)
7> (sensor_9, 1615348339329, 42.49838055575979)
8> (sensor_2, 1615348339329, 43.78154120804511)
.....
```

1、使用 ReduceFunction

ReduceFunction 指定如何组合输入中的两个元素来生成相同类型的输出元素。Flink 使用 ReduceFunction 递增地聚合窗口的元素。

【示例】找到每个传感器在每 1 分钟的事件时间窗口内的温度峰值。

编写流处理程序如下（注意在窗口操作之前，要先分配事件时间戳）：

```
package com.xueai8.java.ch03;

import com.xueai8.java.ch03.entity.SensorReading;
import com.xueai8.java.ch03.source.SensorSource;
```

```
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.ReduceFunction;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

/**
 * 编写流处理程序，要求在 1 分钟的事件时间窗口内从每个传感器找到峰值，
 * 并生成包含(key, end-of-window-timestamp, max_value)的元组流。
 *
 * 使用 ReduceFunction 窗口处理函数，这是一个增量的窗口函数
 */
class SensorReduceFunDemo {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 读取流数据源
        DataStream<SensorReading> input = env.addSource(new SensorSource());

        input
            // 分配时间戳和水印
            .assignTimestampsAndWatermarks(WatermarkStrategy
                .<SensorReading>forMonotonousTimestamps()
                .withTimestampAssigner((event,timestamp) -> event.timestamp)
            )
            // 按传感器 id 分区
            // 已弃用
            .keyBy("id")
            .keyBy(sensorReading -> sensorReading.id)
            // 划分 1 分钟窗口
            .window(TumblingEventTimeWindows.of(Time.minutes(1)))
            // 聚合
            .reduce(new ReduceMaxTemp())
            // 输出
            .print();

        // 执行
        env.execute("flink streaming job");
    }

    // 窗口处理函数
    public static class ReduceMaxTemp implements ReduceFunction<SensorReading>{

        @Override
        public SensorReading reduce(SensorReading sr1, SensorReading sr2) throws Exception {
            return sr1.temperature>sr2.temperature ? sr1 : sr2;
        }
    }
}
```

```
}
```

执行以上代码，输出结果如下：

```
8> (sensor_9, 1615348669229, 73.46634615051136)
3> (sensor_8, 1615348679677, 75.67541937317398)
1> (sensor_0, 1615348662489, 71.44518522080278)
7> (sensor_4, 1615348671540, 17.85601621623948)
2> (sensor_2, 1615348673449, 60.55270939353984)
7> (sensor_7, 1615348674656, 69.83861401937644)
5> (sensor_1, 1615348669531, 80.92630035619608)
6> (sensor_5, 1615348662690, 87.89684014528792)
1> (sensor_3, 1615348673650, 83.68706216364068)
6> (sensor_6, 1615348669732, 85.60202765682551)
.....
```

2、使用 AggregateFunction

AggregateFunction 是 ReduceFunction 的广义版本，它有三种参数类型：输入类型(IN)、累加器类型(ACC)和输出类型(OUT)。输入类型是输入流中元素的类型，AggregateFunction 有一个方法可以将一个输入元素添加到累加器中。该接口还具有创建初始累加器、将两个累加器合并到一个累加器以及从累加器中提取输出(类型为 OUT)的方法。与 ReduceFunction 相同，Flink 将在窗口的输入元素到达时增量地聚合它们。

【示例】找到每个传感器在每 1 分钟的事件时间窗口内的平均温度，并生成包含(传感器 id、平均温度)的元组流。

编写流处理程序如下（注意在窗口操作之前，要先分配事件时间戳）：

```
package com.xueai8.java.ch03;

import com.xueai8.java.ch03.entity.SensorReading;
import com.xueai8.java.ch03.source.SensorSource;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.AggregateFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

/**
 * 编写流处理程序，要求计算在 1 分钟的事件时间窗口内每个传感器的温度平均值，
 * 并生成包含(sensor_id, end-of-window-timestamp, max_value)的元组流。
 *
 * 使用 AggregateFunction 窗口处理函数，这是一个增量的窗口函数
 * 但无法获取窗口信息（例如，窗口时间）
 */
public class SensorAggregateFunDemo {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
```

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

// 读取流数据源
DataStream<SensorReading> input = env.addSource(new SensorSource());

// 管道
input
    // 分配时间戳和水印
    .assignTimestampsAndWatermarks(WatermarkStrategy
        .<SensorReading>forMonotonousTimestamps()
        .withTimestampAssigner((event,timestamp) -> event.timestamp)
    )
    // 按传感器 id 分区
    // .keyBy("id") // 已弃用
    .keyBy(sensorReading -> sensorReading.id)
    // 划分 1 分钟窗口
    .window(TumblingEventTimeWindows.of(Time.minutes(1)))
    // 聚合
    .aggregate(new AggAvgTemp())
    // 输出
    .print();

// 执行
env.execute("flink streaming job");
}

// 窗口聚合函数
public static class AggAvgTemp implements AggregateFunction<
    SensorReading, // input
    Tuple3<String,Double,Long>, // acc, <id, sum, count>
    Tuple2<String,Double>> { // output, <id, avg>

    // 创建初始 ACC
    @Override
    public Tuple3<String, Double, Long> createAccumulator() {
        return new Tuple3<>("",0.0,0L);
    }

    // 累加每个传感器（每个分区）的事件
    @Override
    public Tuple3<String, Double, Long> add(SensorReading sr, Tuple3<String, Double, Long> t3) {
        return new Tuple3<>(sr.id, t3.f1+sr.temperature,t3.f2+1);
    }

    // 分区合并
    @Override
    public Tuple3<String, Double, Long> merge(
        Tuple3<String, Double, Long> acc1,
        Tuple3<String, Double, Long> acc2) {
        return new Tuple3<>(acc1.f0,acc1.f1+acc2.f1,acc1.f2+acc2.f2);
    }
}
```



```

    }

    // 返回每个传感器的平均温度
    @Override
    public Tuple2<String, Double> getResult(Tuple3<String, Double, Long> t3) {
        return new Tuple2<>(t3.f0, t3.f1/t3.f2);
    }
}

```

执行以上代码，输出结果如下：

```

3> (sensor_8,42.08060591879755)
2> (sensor_2,71.6633424330926)
1> (sensor_0,77.84947694547725)
8> (sensor_9,82.01030900335661)
1> (sensor_3,59.51393327888449)
6> (sensor_5,79.5270624749896)
5> (sensor_1,49.93551500440228)
7> (sensor_4,93.84213670264313)
6> (sensor_6,81.73235148899215)
7> (sensor_7,107.1370199018038)
.....

```

3、使用 ProcessWindowFunction

ProcessWindowFunction 获取一个包含窗口所有元素的迭代器，以及一个具有时间和状态信息访问权的上下文对象（Context），这使它比其他窗口函数提供更大的灵活性，不过这是以性能和资源消耗为代价的，因为元素不能增量地聚合，而是需要在内部进行缓冲，直到认为窗口已经准备好进行处理。

ProcessWindowFunction 的签名如下：

Java 代码：

```

public abstract class ProcessWindowFunction<IN, OUT, KEY, W extends Window> implements Function {

    /**
     * Evaluates the window and outputs none or several elements.
     *
     * @param key The key for which this window is evaluated.
     * @param context The context in which the window is being evaluated.
     * @param elements The elements in the window being evaluated.
     * @param out A collector for emitting elements.
     *
     * @throws Exception The function may throw exceptions to fail the program and trigger recovery.
     */
    public abstract void process(
        KEY key,
        Context context,
        Iterable<IN> elements,
        Collector<OUT> out) throws Exception;

    /**
     * The context holding window metadata.
     */
}

```

```

    */
    public abstract class Context implements java.io.Serializable {
        /**
         * Returns the window that is being evaluated.
         */
        public abstract W window();

        /** Returns the current processing time. */
        public abstract long currentProcessingTime();

        /** Returns the current event-time watermark. */
        public abstract long currentWatermark();

        /**
         * State accessor for per-key and per-window state.
         *
         * <p><b>NOTE:</b>If you use per-window state you have to ensure that you clean it up
         * by implementing {@link ProcessWindowFunction#clear(Context)}.
         */
        public abstract KeyedStateStore windowState();

        /**
         * State accessor for per-key global state.
         */
        public abstract KeyedStateStore globalState();
    }
}

```

Scala 代码:

```

abstract class ProcessWindowFunction[IN, OUT, KEY, W <: Window] extends Function {

    /**
     * Evaluates the window and outputs none or several elements.
     *
     * @param key      The key for which this window is evaluated.
     * @param context  The context in which the window is being evaluated.
     * @param elements The elements in the window being evaluated.
     * @param out      A collector for emitting elements.
     * @throws Exception The function may throw exceptions to fail the program and trigger recovery.
     */
    def process(
        key: KEY,
        context: Context,
        elements: Iterable[IN],
        out: Collector[OUT])

    /**
     * The context holding window metadata
     */
    abstract class Context {

```

```

/**
 * Returns the window that is being evaluated.
 */
def window: W

/**
 * Returns the current processing time.
 */
def currentProcessingTime: Long

/**
 * Returns the current event-time watermark.
 */
def currentWatermark: Long

/**
 * State accessor for per-key and per-window state.
 */
def windowState: KeyedStateStore

/**
 * State accessor for per-key global state.
 */
def globalState: KeyedStateStore
}

```

注:KEY 参数是通过 keyBy()调用指定的 KeySelector 提取的 key。对于 tuple-index keys 或 string-field references, 此 key 类型总是 Tuple, 必须手动将其转换为正确大小的元组, 以提取该 key 字段。

ProcessWindowFunction 被传递了一个 Context 上下文对象, 该对象包含了关于窗口的信息, 我们可以从中获得关于窗口的信息。Context 接口的定义如下:

```

public abstract class Context implements java.io.Serializable {
    public abstract W window();

    public abstract long currentProcessingTime();
    public abstract long currentWatermark();

    public abstract KeyedStateStore windowState();
    public abstract KeyedStateStore globalState();
}

```

可以将 per-key、per-window、或 global per-key 的信息存储到 windowState 和 globalState 中。如果想记录关于当前窗口的信息, 并在处理后续窗口时使用它, 这会非常有用。

【示例】找到在 1 分钟的事件时间窗口内每个传感器的温度峰值, 并生成包含(传感器 id, 窗口结束时间戳, 温度峰值)的元组流。

```

package com.xueai8.java.ch03;

import com.xueai8.java.ch03.entity.SensorReading;
import com.xueai8.java.ch03.source.SensorSource;

```

```
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.windowing.ProcessWindowFunction;
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.api.windowing.windows.TimeWindow;
import org.apache.flink.util.Collector;

/**
 * 找到在 1 分钟的事件时间窗口内每个传感器的温度峰值，并生成包含(传感器 id, 窗口结束时间戳, 温度峰值)的
 * 元组流。
 */
public class SensorProcessWindowFunDemo {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 读取流数据源
        DataStream<SensorReading> input = env.addSource(new SensorSource());

        input
            // 分配事件时间戳
            .assignTimestampsAndWatermarks(WatermarkStrategy
                .<SensorReading>forMonotonousTimestamps()
                .withTimestampAssigner((event,timestamp) -> event.timestamp)
            )
            // .keyBy("id")    // 已弃用
            .keyBy(sensorReading -> sensorReading.id)
            .window(TumblingEventTimeWindows.of(Time.minutes(1)))
            .process(new MyWastefulMax())
            .print();

        // 执行
        env.execute("flink streaming job");
    }

    // 窗口处理函数
    public static class MyWastefulMax extends ProcessWindowFunction<
        SensorReading,                               // input type
        Tuple3<String, Long, Double>,                 // output type
        String,                                       // key type
        TimeWindow> {                                // window type

        @Override
        public void process(
            String id,
            Context context,
```

```
        Iterable<SensorReading> events,
        Collector<Tuple3<String, Long, Double>> out) {
    double max = 0;
    for (SensorReading event : events) {
        if (event.temperature > max) max = event.temperature;
    }
    out.collect(new Tuple3<>(id, context.window().getEnd(), max));
}
}
```

执行上面的程序，输出结果如下：

```
7> (sensor_25,1615343280000,87.72586269722625)
8> (sensor_35,1615343280000,43.142193856943834)
3> (sensor_21,1615343280000,67.3876390193542)
6> (sensor_37,1615343280000,69.2088403952636)
8> (sensor_9,1615343280000,51.63106770197567)
2> (sensor_24,1615343280000,58.968195282098684)
.....
5> (sensor_64,1615343340000,57.68208910452411)
8> (sensor_19,1615343340000,29.283813257290152)
2> (sensor_43,1615343340000,85.09369114076935)
7> (sensor_25,1615343340000,110.71975425459033)
4> (sensor_72,1615343340000,79.27807830250289)
2> (sensor_51,1615343340000,81.77182221437575)
.....
7> (sensor_52,1615343400000,40.71965827483373)
5> (sensor_57,1615343400000,61.80558008285541)
1> (sensor_48,1615343400000,110.79634444720006)
4> (sensor_46,1615343400000,67.34009796907614)
7> (sensor_33,1615343400000,57.52010144672712)
3> (sensor_53,1615343400000,102.57835200189706)
.....
```

4、使用增量 ReduceFunction + ProcessWindowFunction

将 ProcessWindowFunction 用于简单的聚合(比如 count)是非常低效的。ProcessWindowFunction 可以与 ReduceFunction 组合使用，以便在元素到达窗口时对于其进行增量聚合。当窗口关闭时，ProcessWindowFunction 将提供聚合的结果。这样可以在访问 ProcessWindowFunction 的附加窗口元信息的同时递增地计算窗口。

【示例】将增量 ReduceFunction 与 ProcessWindowFunction 组合起来，以返回窗口中最小的事件以及窗口的开始时间。

```
package com.xueai8.java.ch03;

import com.xueai8.java.ch03.entity.SensorReading;
import com.xueai8.java.ch03.source.SensorSource;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.ReduceFunction;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.streaming.api.datastream.DataStream;
```

```
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.windowing.ProcessWindowFunction;
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.api.windowing.windows.TimeWindow;
import org.apache.flink.util.Collector;

/**
 * 编写流处理程序，要将增量 ReduceFunction 与 ProcessWindowFunction 组合起来，
 * 以返回窗口中最高温度的读数以及每个窗口的开始时间。
 */
public class SensorReduceProcessFunDemo {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 读取流数据源
        DataStream<SensorReading> input = env.addSource(new SensorSource());

        input
            // 分配时间戳和水印
            .assignTimestampsAndWatermarks(WatermarkStrategy
                .<SensorReading>forMonotonousTimestamps()
                .withTimestampAssigner((event,timestamp) -> event.timestamp)
            )
            // 按传感器 id 分区
            // .keyBy("id") // 已弃用
            .keyBy(sensorReading -> sensorReading.id)
            // 划分 1 分钟窗口
            .window(TumblingEventTimeWindows.of(Time.minutes(1)))
            // 聚合
            .reduce(new ReduceMaxTemp(), new ProcessMaxTemp())
            // 输出
            .print();

        // 执行
        env.execute("flink streaming job");
    }

    // 窗口增量 reduce 函数
    public static class ReduceMaxTemp implements ReduceFunction<SensorReading> {
        @Override
        public SensorReading reduce(SensorReading sr1, SensorReading sr2) throws Exception {
            return sr1.temperature > sr2.temperature ? sr1 : sr2;
        }
    }

    // 窗口处理函数
    public static class ProcessMaxTemp extends ProcessWindowFunction<
```

```

        SensorReading,           // input type
        Tuple3<String, Long, Double>, // output type
        String,                 // key type
        TimeWindow> {           // window type

    @Override
    public void process(
        String id,
        Context context,
        Iterable<SensorReading> events,
        Collector<Tuple3<String, Long, Double>> out) {
        // 注意, Iterable<SensorReading>将只包含一个读数,
        // 即 MyReduceFunction 计算出的预先聚合的最大值。
        double max = events.iterator().next().temperature;
        out.collect(new Tuple3<>(id, context.window().getEnd(), max));
    }
}

```

注意, `Iterable<SensorReading>` 将只包含一个读数—`MyReduceFunction` 计算出的预先聚合的最大值。执行上面的代码, 输出结果如下:

```

8> (sensor_9,1615355520000,32.37375619031118)
3> (sensor_8,1615355520000,96.42499637334932)
5> (sensor_1,1615355520000,55.060786284631355)
6> (sensor_5,1615355520000,94.91489904737487)
2> (sensor_2,1615355520000,70.12297314504767)
1> (sensor_0,1615355520000,71.17846770042992)
7> (sensor_4,1615355520000,55.549531169074356)
6> (sensor_6,1615355520000,54.348664791183026)
1> (sensor_3,1615355520000,54.64384452116478)
7> (sensor_7,1615355520000,36.501163398755025)
.....

```

5、使用增量 `AggregateFunction` + `ProcessWindowFunction`

`ProcessWindowFunction` 可以与 `AggregateFunction` 组合使用, 以便在元素到达窗口时对于其进行增量聚合。这样可以在访问 `ProcessWindowFunction` 的附加窗口元信息的同时增量地聚合窗口。

【示例】将增量 `AggregateFunction` 与 `ProcessWindowFunction` 组合起来, 计算在 1 分钟的事件时间窗口内每个传感器的温度平均值, 并生成包含(sensor_id, end-of-window-timestamp, max_value)的元组流。

```

package com.xueai8.java.ch03;

import com.xueai8.java.ch03.entity.SensorReading;
import com.xueai8.java.ch03.source.SensorSource;
import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.AggregateFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.tuple.Tuple3;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.windowing.ProcessWindowFunction;

```

```
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.api.windowing.windows.TimeWindow;
import org.apache.flink.util.Collector;

/**
 * 编写流处理程序，要将增量 ReduceFunction 与 ProcessWindowFunction 组合起来，
 * 以返回窗口中最高温度的读数以及每个窗口的开始时间。
 */
public class SensorAggProcessFunDemo {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 读取流数据源
        DataStream<SensorReading> input = env.addSource(new SensorSource());

        input
            // 分配时间戳和水印
            .assignTimestampsAndWatermarks(WatermarkStrategy
                .<SensorReading>forMonotonousTimestamps()
                .withTimestampAssigner((event,timestamp) -> event.timestamp)
            )
            // 按传感器 id 分区
            // 已弃用
            .keyBy(sensorReading -> sensorReading.id)
            // 划分 1 分钟窗口
            .window(TumblingEventTimeWindows.of(Time.minutes(1)))
            // 聚合
            .aggregate(new AggAvgTemp(),new ProcessMaxTemp())
            // 输出
            .print();

        // 执行
        env.execute("flink streaming job");
    }

    // 窗口增量聚合函数
    public static class AggAvgTemp implements AggregateFunction<
        SensorReading, // input
        Tuple2<Double,Long>, // acc, <id, sum, count>
        Double> { // output

        // 创建初始 ACC
        @Override
        public Tuple2<Double,Long> createAccumulator() {
            return new Tuple2<>(0.0,0L);
        }
    }
}
```



```
// 累加每个传感器（每个分区）的事件
@Override
public Tuple2<Double,Long> add(SensorReading sr, Tuple2<Double,Long> t2) {
    return new Tuple2<>(t2.f0+sr.temperature,t2.f1+1);
}

// 分区合并
@Override
public Tuple2<Double,Long> merge(
    Tuple2<Double,Long> acc1,
    Tuple2<Double,Long> acc2) {
    return new Tuple2<>(acc1.f0+acc2.f0,acc1.f1+acc2.f1);
}

// 返回每个传感器的平均温度
@Override
public Double getResult(Tuple2<Double,Long> t2) {
    return t2.f0/t2.f1;
}
}

// 窗口处理函数
public static class ProcessMaxTemp extends ProcessWindowFunction<
    Double,                                     // input type
    Tuple3<String, Long, Double>,              // output type
    String,                                     // key type
    TimeWindow> {                               // window type

    @Override
    public void process(
        String id,
        Context context,
        Iterable<Double> events,
        Collector<Tuple3<String, Long, Double>> out) {
        double average = events.iterator().next();
        out.collect(new Tuple3<>(id,context.window().getEnd(), average));
    }
}
}
```

执行以上代码，输出结果如下所示：

```
5> (sensor_1,1615356240000,79.0445303156929)
1> (sensor_0,1615356240000,80.47726227716672)
7> (sensor_4,1615356240000,42.29616683871286)
6> (sensor_5,1615356240000,38.832404558254375)
2> (sensor_2,1615356240000,80.67415078224867)
7> (sensor_7,1615356240000,81.94299042105301)
6> (sensor_6,1615356240000,50.87279213748181)
8> (sensor_9,1615356240000,92.91812504642569)
3> (sensor_8,1615356240000,62.813793188352854)
1> (sensor_3,1615356240000,55.72952046921107)
```

.....

6、使用 WindowFunction (遗留的旧接口)

在一些可以使用 ProcessWindowFunction 的地方，还可以使用 WindowFunction。这是一个较老版本的 ProcessWindowFunction，它提供的上下文信息较少，并且缺少一些高级特性，例如 per-window keyed state。（注：这个接口将在某个时候被弃用）

WindowFunction 接口的签名如下。

Java 代码：

```
public interface WindowFunction<IN, OUT, KEY, W extends Window> extends Function, Serializable {

    /**
     * Evaluates the window and outputs none or several elements.
     *
     * @param key The key for which this window is evaluated.
     * @param window The window that is being evaluated.
     * @param input The elements in the window being evaluated.
     * @param out A collector for emitting elements.
     *
     * @throws Exception The function may throw exceptions to fail the program and trigger recovery.
     */
    void apply(KEY key, W window, Iterable<IN> input, Collector<OUT> out) throws Exception;
}
```

Scala 代码：

```
trait WindowFunction[IN, OUT, KEY, W <: Window] extends Function with Serializable {

    /**
     * Evaluates the window and outputs none or several elements.
     *
     * @param key The key for which this window is evaluated.
     * @param window The window that is being evaluated.
     * @param input The elements in the window being evaluated.
     * @param out A collector for emitting elements.
     * @throws Exception The function may throw exceptions to fail the program and trigger recovery.
     */
    def apply(key: KEY, window: W, input: Iterable[IN], out: Collector[OUT])
}
```

可以像下面这样来使用 WindowFunction 函数。

Scala 代码：

```
val input: DataStream[(String, Long)] = ...

input
    .keyBy(<key selector>)
    .window(<window assigner>)
    .apply(new MyWindowFunction())
```

Java 代码：

```
DataStream<Tuple2<String, Long>> input = ...;

input
```

```
.keyBy(<key selector>)\n.window(<window assigner>)\n.apply(new MyWindowFunction());
```

【示例】找到在 1 分钟的事件时间窗口内每个传感器的温度峰值，并生成包含(传感器 id, 窗口结束时间戳, 温度峰会)的元组流。

```
package com.xueai8.java.ch03;\n\nimport com.xueai8.java.ch03.entity.SensorReading;\nimport com.xueai8.java.ch03.source.SensorSource;\nimport org.apache.flink.api.common.eventtime.WatermarkStrategy;\nimport org.apache.flink.api.java.tuple.Tuple3;\nimport org.apache.flink.streaming.api.datastream.DataStream;\nimport org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;\nimport org.apache.flink.streaming.api.functions.windowing.WindowFunction;\nimport org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;\nimport org.apache.flink.streaming.api.windowing.time.Time;\nimport org.apache.flink.streaming.api.windowing.windows.TimeWindow;\nimport org.apache.flink.util.Collector;\n\n/**\n * 编写流处理程序，要求在 1 分钟的事件时间窗口内从每个传感器找到温度峰值，\n * 并生成包含(key, end-of-window-timestamp, max_value)的元组流。\n *\n * WindowFunction 是一个较老版本的 ProcessWindowFunction，它提供的上下文信息较少，\n * 并且缺少一些高级特性，例如 per-window keyed state。\n * （注：这个接口将在某个时候被弃用）\n */\npublic class SensorWindowFunctionDemo {\n\n    public static void main(String[] args) throws Exception {\n        // 设置流执行环境\n        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();\n\n        // 读取流数据源\n        DataStream<SensorReading> input = env.addSource(new SensorSource());\n\n        // 处理管道\n        input\n            // 分配时间戳和水印\n            .assignTimestampsAndWatermarks(WatermarkStrategy\n                .<SensorReading>forMonotonousTimestamps()\n                .withTimestampAssigner((event,timestamp) -> event.timestamp)\n            )\n            // 按传感器 id 分区\n            .keyBy("id") // 已弃用\n            .keyBy(sensorReading -> sensorReading.id)\n            // 划分 1 分钟窗口\n            .window(TumblingEventTimeWindows.of(Time.minutes(1)))\n            // 应用自定义窗口函数\n            .apply(new MyWindowFunction());\n    }\n}
```

```
// 输出
.print();

// 触发程序执行
env.execute("Flink Sensor Stream");
}

// 窗口处理函数
public static class MyWindowFunction implements WindowFunction<
    SensorReading,                               // input type
    Tuple3<String, Long, Double>,                 // output type
    String,                                       // key type
    TimeWindow> {                                // window type
    @Override
    public void apply(String id,
        TimeWindow timeWindow,
        Iterable<SensorReading> events,
        Collector<Tuple3<String, Long, Double>> out
    ) throws Exception {
        double max = 0;
        for (SensorReading event : events) {
            if (event.temperature > max) max = event.temperature;
        }
        out.collect(new Tuple3<>(id, timeWindow.getEnd(), max));
    }
}
```

执行以上代码，输出结果如下：

```
1> (sensor_0,1615357080000,78.73390698185324)
2> (sensor_2,1615357080000,96.7693075853522)
3> (sensor_8,1615357080000,74.20232753185753)
6> (sensor_5,1615357080000,33.620243996559225)
7> (sensor_4,1615357080000,87.31423973245688)
5> (sensor_1,1615357080000,80.09551023652499)
8> (sensor_9,1615357080000,76.00900462331602)
6> (sensor_6,1615357080000,92.86722930073302)
1> (sensor_3,1615357080000,123.3866054120469)
7> (sensor_7,1615357080000,76.97441380518192)
.....
```

练习

【练习】对一个元组数据流窗口中所有元素的第二个字段求和。

Java 代码：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.ReduceFunction;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
```

```
/**
 * 窗口聚合，使用 ReduceFunction
 *
 * 对一个元组数据流窗口中所有元组元素的第二个字段求和。
 */
class TupleReduceFunDemo {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 首先从环境中获取一些数据，再执行 reduce 转换：
        DataStream<Tuple2<String,Integer>> ds = env.fromElements(
            new Tuple2<>("good",1),
            new Tuple2<>("good",1),
            new Tuple2<>("study",1)
        );

        ds
            .keyBy(t -> t.f0)
            .reduce(new ReduceFunction<Tuple2<String, Integer>>() {
                @Override
                public Tuple2<String, Integer> reduce(
                    Tuple2<String, Integer> t1,
                    Tuple2<String, Integer> t2) throws Exception {
                    return new Tuple2<>(t1.f0 + "-" + t2.f0, t1.f1 + t2.f1);
                }
            })
            .print();

        // 执行
        env.execute("flink streaming job");
    }
}
```

输出结果如下所示：

```
6> (good,1)
5> (study,1)
6> (good-good,2)
```

Scala 代码：

```
package com.xueai8.scala.ch03

import org.apache.flink.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

object BatchJob {

    def main(args: Array[String]) {
        // 设置批处理执行环境
        val env = StreamExecutionEnvironment
```

```
// 得到输入数据，reduce 转换
env.fromElements("Good good study", "Day day up")
  .map(_._toLowerCase)
  .flatMap(_._split("\\W+")) // 相当于先 map，再 flatten
  .map(_._1)
  .keyBy(_._1)
  .reduce((a,b) => (a._1 + "-" + b._1, a._2 + b._2))
  .print

env.execute("flink reduce transformation")
}
```

输出结果如下所示：

```
(good,1)
(study,1)
(good-good,2)
(day,1)
(day-day,2)
(up,1)
```

3.7.4 触发器

触发器（Trigger）确定一个窗口何时调用窗口函数。每个 WindowAssigner 都带有一个默认触发器。如果默认触发器不适合需要，我们可以使用 trigger(...)指定自定义触发器。

触发器接口有五种方法，允许 Trigger 对不同的事件作出反应：

- ☐ onElement()方法：为添加到窗口的每个元素调用该方法；
- ☐ onEventTime()方法：当已注册的事件时间定时器触发时，将调用该方法；
- ☐ onProcessingTime()方法：当已注册的处理时间定时器触发时，将调用该方法；
- ☐ onMerge()方法：该方法与有状态触发器相关，并在两个触发器的相应窗口合并时合并它们的状态，例如在使用会话窗口时。
- ☐ clear()方法：执行删除相应窗口时所需的任何操作。

以上方法有两点需要注意：

1) 前三种方法通过返回 TriggerResult 来决定如何处理它们的调用事件。可以采取以下行动之一：

- ☐ CONTINUE：什么也不做；
- ☐ FIRE：触发计算；
- ☐ PURGE：清除窗口中的元素；
- ☐ FIRE_AND_PURGE：触发计算，然后清除窗口中的元素。

2) 这些方法中的任何一个都可以用来为将来的操作注册处理时间计时器或事件时间计时器。

触发和清除

一旦触发器确定某个窗口已准备好进行处理，它就会触发，即，它返回 FIRE 或 FIRE_AND_PURGE。这是窗口运算符发出（emit）当前窗口结果的信号。给定一个带有 ProcessWindowFunction 的窗口，所有元素都被传递给 ProcessWindowFunction(可能在将它们传递给一个 evictor 回收器之后)。带有 ReduceFunction、AggregateFunction 或 FoldFunction 的窗口简单地发出（emit）它们 eagerly 聚合的结果。

当触发器触发时，它可以是 FIRE 或 FIRE_AND_PURGE。当 FIRE 时保留窗口的内容，而 FIRE_AND_PURGE 则删除窗口的内容。默认情况下，预实现的触发器简单地 FIRE 而不清除窗口状态。

注：清除将简单地删除窗口的内容，并保留关于窗口和任何触发器状态的任何潜在元信息。

WindowAssigners 的默认触发器

WindowAssigner 的默认触发器适用于许多场景。例如，所有事件时间窗口分配器都有一个 EventTimeTrigger 作为默认触发器。一旦水印经过窗口的末端，这个触发器就会触发。

全局窗口（GlobalWindow）的默认触发器是永不触发的 NeverTrigger。因此，在使用全局窗口时，总是必须定义一个自定义触发器。

注：通过使用 trigger() 指定触发器，会覆盖 WindowAssigner 的默认触发器。例如，如果为 TumblingEventTimeWindows 指定了一个 CountTrigger，那么将不再根据时间进度获得窗口触发，而是只根据 count 计数。在这种情况下，如果想根据时间和计数都做出反应，就必须编写自己的自定义触发器。

内置和自定义触发器

Flink 有几个内置的触发器：

- ❑ EventTimeTrigger：基于由水印度量的事件时间进程触发。
- ❑ ProcessingTimeTrigger：根据处理时间触发。
- ❑ CountTrigger：一旦窗口中的元素数量超过给定的限制，CountTrigger 就会触发。
- ❑ PurgingTrigger：将另一个触发器作为参数并将其转换为一个清除触发器。

注：如果需实现自定义触发器，应该检查抽象 Trigger 类。请注意，该 API 仍在发展中，并且在 Flink 的未来版本中可能会发生变化。

3.7.5 清除器

除 WindowAssigner 和 Trigger 之外，Flink 的窗口模型还允许指定可选的清除器（Evictor）。这可以使用 evictor(...)方法实现。清除器能够在触发器触发之后以及应用窗口函数之前和/或之后从窗口中删除元素。为此，Evictor 接口有两种方法：

```
/**
 * Optionally evicts elements. Called before windowing function.
 *
 * @param elements The elements currently in the pane.
 * @param size The current number of elements in the pane.
 * @param window The {@link Window}
 * @param evictorContext The context for the Evictor
 */
void evictBefore(Iterable<TimestampedValue<T>> elements, int size, Window window, EvictorContext evictorContext);

/**
 * Optionally evicts elements. Called after windowing function.
 *
 * @param elements The elements currently in the pane.
 * @param size The current number of elements in the pane.
 * @param window The {@link Window}
 * @param evictorContext The context for the Evictor
 */
```

```
*/
```

```
void evictAfter(Iterable<TimestampedValue<T>> elements, int size, W window, EvictorContext evictorContext);
```

其中 `evictBefore()` 包含要在窗口函数之前应用的清除逻辑，而 `evictAfter()` 包含要在窗口函数之后应用的清除逻辑。在应用窗口函数之前被移除的元素将不会被该窗口函数处理。

Flink 附带三个预实现的清除器，分别是：

- ❑ **CountEvictor**：从窗口保留用户指定数量的元素，并从窗口缓冲区开始处丢弃剩余的元素。
- ❑ **DeltaEvictor**：接受一个 `DeltaFunction` 和一个阈值，计算窗口缓冲区中的最后一个元素与其余每个元素之间的增量，并删除增量大于或等于阈值的元素。
- ❑ **TimeEvictor**：将一个以毫秒为单位的 `interval` 作为参数，对于给定的窗口，它在其元素中找到最大时间戳 `max_ts`，并删除所有时间戳小于 `max_ts - interval` 的元素。

默认情况下，所有预实现的清除器都在窗口函数之前应用它们的逻辑。

注：指定 `evictor` 可以防止任何预聚合，因为在应用计算之前，窗口的所有元素都必须传递给 `evictor`。这意味着带有 `evictor` 的窗口将创建更多的状态。

注：Flink 不能保证窗口中元素的顺序。这意味着，尽管 `evictor` 可能从窗口的开始处删除元素，但这些元素不一定是最先到达或最后到达的元素。

3.7.6 处理迟到数据

当使用事件时间窗口时，可能会出现元素延迟到达的情况，即 Flink 用来跟踪事件时间进程的水印已经超过了元素所属窗口的结束时间戳。

延迟元素是在系统事件时间时钟(由水印表示)已经通过了延迟元素的时间戳之后到达的元素。

默认情况下，当水印经过窗口末尾时，迟到的数据将被删除。然而，Flink 允许为窗口操作符指定允许的最大延迟。允许延迟指定元素在被删除之前可以延迟多少时间，其默认值为 0。在水印经过窗口末尾之后，但是在水印经过（窗口末尾+允许的延迟）之前到达的元素，仍然被添加到窗口中。根据使用的触发器的不同，延迟但未删除的元素可能会导致窗口再次触发。`EventTimeTrigger` 就是这种情况。

Flink 将会保持窗口的状态，直到它们允许的延迟过期为止。然后，Flink 就会删除窗口并删除它的状态。

默认情况下，允许的延迟设置为 0。也就是说，在水印后面到达的元素将被删除。我们可以像下面这样指定允许的延迟。

Scala 代码：

```
val input: DataStream[T] = ...
```

```
input
```

```
    .keyBy(<key selector>)
    .window(<window assigner>)
    .allowedLateness(<time>)
    .<windowed transformation>(<window function>)
```

Java 代码：

```
DataStream<T> input = ...;
```

```
input
```

```
    .keyBy(<key selector>)
    .window(<window assigner>)
    .allowedLateness(<time>)
```



```
.<windowed transformation>(<window function>);
```

注：在使用 GlobalWindows 窗口分配器时，不会考虑数据延迟，因为全局窗口的结束时间戳是 Long.MAX_VALUE。

当指定允许的延迟大于 0 时，窗口及其内容将在水印通过窗口末端后保留。在这些情况下，当一个延迟但未删除的元素到达时，可能会触发窗口的另一次触发。这些触发被称为延迟触发（late firings），因为它们是由延迟事件触发的，与主触发（main firing）不同，主触发是窗口的第一次触发。

会话窗口基于可以合并的窗口抽象。每个元素最初都被分配给一个新窗口，在此之后，只要它们之间的差距足够小，就会合并窗口。通过这种方式，一个延迟事件可以将两个之前分离的会话连接起来，从而产生一个延迟合并。

获取迟到的数据作为侧输出

使用 Flink 的侧输出特性，可以得到一个延迟丢弃的数据流。

首先需要指定希望在窗口化的流上使用 sideOutputLateData(OutputTag) 获取迟到数据。然后，我们可以在窗口操作的结果上得到侧输出流。

Scala 代码：

```
val lateOutputTag = OutputTag[T]("late-data")

val input: DataStream[T] = ...

val result = input
    .keyBy(<key selector>)
    .window(<window assigner>)
    .allowedLateness(<time>)
    .sideOutputLateData(lateOutputTag)
    .<windowed transformation>(<window function>)
```

```
val lateStream = result.getSideOutput(lateOutputTag)
```

Java 代码：

```
final OutputTag<T> lateOutputTag = new OutputTag<T>("late-data"){

DataStream<T> input = ...;

SingleOutputStreamOperator<T> result = input
    .keyBy(<key selector>)
    .window(<window assigner>)
    .allowedLateness(<time>)
    .sideOutputLateData(lateOutputTag)
    .<windowed transformation>(<window function>);

DataStream<T> lateStream = result.getSideOutput(lateOutputTag);
```

3.7.7 处理窗口结果

窗口操作的结果还是 DataStream，在结果元素中没有保留窗口操作的信息，所以，如果想保持窗口的元信息，就必须在 ProcessWindowFunction 的结果元素中手动编码该信息。在结果元素上设置的惟一

相关信息是元素时间戳。这被设置为已处理窗口的最大允许时间戳，即结束时间戳- 1，因为窗口结束时间戳是不包括在内的。注意，这对于事件时间窗口和处理时间窗口都是一样的。例如，在窗口操作元素之后总是有一个时间戳，但它可以是事件时间时间戳，也可以是一个处理时间时间戳。对于处理时间窗口，这没有特殊的含义，但是对于事件时间窗口，加上水印与窗口的交互方式，可以使用相同的窗口大小实现连续的窗口操作。

水印与窗口的交互

当水印到达窗口操作符时，这将触发两件事：

- ☐ 当最大时间戳（即结束时间戳 - 1）小于新水印时，水印触发所有窗口的计算。
- ☐ 水印按原样转发给下游操作

连续窗口的操作

如前所述，计算窗口化结果的时间戳的方法以及水印与窗口的交互方式允许将连续的窗口化操作串在一起。当我们想要执行两个连续的窗口操作，其中想要使用不同的 key，但仍然希望来自相同上游窗口的元素最终出现在相同的下游窗口中时，这将非常有用。请看下面的示例：

Scala 代码：

```
val input: DataStream[Int] = ...

val resultsPerKey = input
    .keyBy(<key selector>)
    .window(TumblingEventTimeWindows.of(Time.seconds(5)))
    .reduce(new Summer())

val globalResults = resultsPerKey
    .windowAll(TumblingEventTimeWindows.of(Time.seconds(5)))
    .process(new TopKWindowFunction())
```

Java 代码：

```
DataStream<Integer> input = ...;

DataStream<Integer> resultsPerKey = input
    .keyBy(<key selector>)
    .window(TumblingEventTimeWindows.of(Time.seconds(5)))
    .reduce(new Summer());

DataStream<Integer> globalResults = resultsPerKey
    .windowAll(TumblingEventTimeWindows.of(Time.seconds(5)))
    .process(new TopKWindowFunction());
```

在本例中，第一个操作的时间窗[0,5]的结果也将在随后的窗口操作的时间窗[0,5]中结束。这允许计算每个 key 的和，然后在第二个操作中计算同一窗口中的 top-k 元素。

3.7.8 窗口连接

使用窗口连接（window join）将两个流的元素联接起来，这两个流共享一个公共 key，并且位于同一个窗口中。可以使用窗口分配程序定义这些窗口，并对来自这两个流的元素进行计算。

然后将来自两边的元素传递给用户定义的 `JoinFunction` 或 `FlatJoinFunction`，用户可以在其中发出满足联接条件的结果。

一般用法可概括如下：

```
stream.join(otherStream)
    .where(<KeySelector>)
    .equalTo(<KeySelector>)
    .window(<WindowAssigner>)
    .apply(<JoinFunction>)
```

关于语义的一些注意事项：

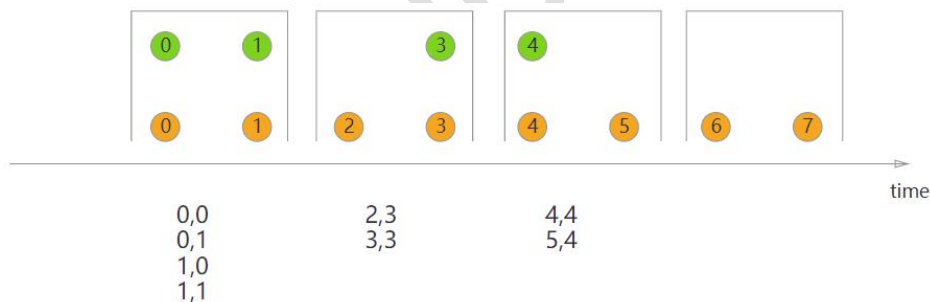
- ❑ 两个流的元素成对组合的创建行为类似于内连接，这意味着如果一个流中的元素没有来自另一个流的相应元素要连接，那么它们将不会被发出。
- ❑ 那些确实被连接的元素将以仍然在各自窗口中的最大时间戳作为它们的时间戳。例如，以[5,10)为边界的窗口将导致合并的元素使用 9 作为时间戳。

下面我们将使用一些示例场景来概述不同类型的窗口连接的行为。

滚动窗口连接

当执行一个滚动窗口连接时，所有具有公共 key 和公共滚动窗口的元素都以成对组合的方式连接，并传递给一个 `JoinFunction` 或 `FlatJoinFunction`。因为这就像一个内连接，一个流的元素如果在其滚动窗口中没有来自另一个流的元素，就不会被发出！

如下图所示，我们定义了一个大小为 2 毫秒的滚动窗口，它产生的窗口形式为[0,1], [2,3],。图中显示了每个窗口中所有元素的成对组合，这些元素将被传递给 `JoinFunction`。注意，在滚动窗口[6,7]中没有发出任何东西，因为绿色的流中不存在与橙色的元素⑥和⑦相连接的元素。



Scala 代码：

```
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

...

val orangeStream: DataStream[Integer] = ...
val greenStream: DataStream[Integer] = ...

orangeStream.join(greenStream)
    .where(elem => /* select key */)
    .equalTo(elem => /* select key */)
    .window(TumblingEventTimeWindows.of(Time.milliseconds(2)))
    .apply { (e1, e2) => e1 + "," + e2 }
```

Java 代码：

```

import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

...

DataStream<Integer> orangeStream = ...
DataStream<Integer> greenStream = ...

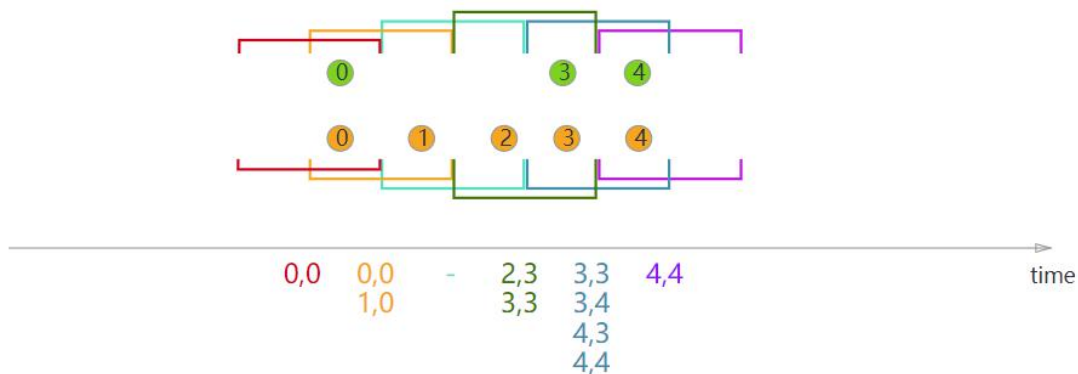
orangeStream.join(greenStream)
    .where(<KeySelector>)
    .equalTo(<KeySelector>)
    .window(TumblingEventTimeWindows.of(Time.milliseconds(2)))
    .apply(new JoinFunction<Integer, Integer, String> (){
        @Override
        public String join(Integer first, Integer second) {
            return first + "," + second;
        }
    });

```

滑动窗口连接

当执行滑动窗口连接时，所有具有公共 key 和公共滑动窗口的元素都作为成对组合连接，并传递给 JoinFunction 或 FlatJoinFunction。在当前滑动窗口中，一个流的元素中没有来自另一个流的元素的元素不会被触发！注意，有些元素可能在一个滑动窗口中被连接，而在另一个窗口中却没有！

在下面这个例子中，我们使用大小为 2 毫秒的滑动窗口，并将其滑动 1 毫秒，结果是滑动窗口[- 1,0], [0,1], [1,2], [2,3], ...。x 轴下面连接的元素是为每个滑动窗口传递给 JoinFunction 的元素。在这里还可以看到，例如，在窗口[2,3]中，橙色的②与绿色的③进行了 join 连接，但与窗口[1,2]中的任何内容都没有与任何元素进行连接。



Scala 代码:

```

import org.apache.flink.streaming.api.windowing.assigners.SlidingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

...

val orangeStream: DataStream[Integer] = ...
val greenStream: DataStream[Integer] = ...

```

```
orangeStream.join(greenStream)
  .where(elem => /* select key */)
  .equalTo(elem => /* select key */)
  .window(SlidingEventTimeWindows.of(Time.milliseconds(2) /* size */, Time.milliseconds(1) /* slide */))
  .apply { (e1, e2) => e1 + "," + e2 }
```

Java 代码:

```
import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.streaming.api.windowing.assigners.SlidingEventTimeWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

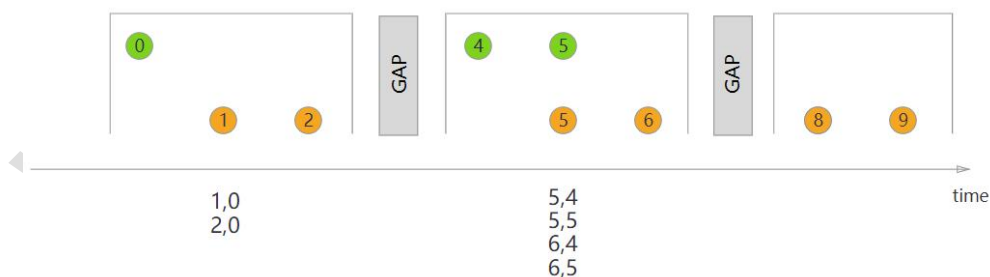
...

DataStream<Integer> orangeStream = ...
DataStream<Integer> greenStream = ...

orangeStream.join(greenStream)
  .where(<KeySelector>)
  .equalTo(<KeySelector>)
  .window(SlidingEventTimeWindows.of(Time.milliseconds(2) /* size */, Time.milliseconds(1) /* slide */))
  .apply (new JoinFunction<Integer, Integer, String> () {
    @Override
    public String join(Integer first, Integer second) {
      return first + "," + second;
    }
  });
```

会话窗口连接

当执行会话窗口连接时，当“组合”满足会话条件时，所有具有相同键的元素都以成对的组合连接，并传递给 JoinFunction 或 FlatJoinFunction。同样，这将执行一个内连接，因此如果会话窗口只包含来自一个流的元素，则不会发出任何输出！



这里我们定义了一个会话窗口连接，其中每个会话至少间隔 1ms。有三个会话，在前两个会话中，来自两个流的连接元素被传递给 JoinFunction。在第三次会话中，绿色流中没有元素，所以⑧和⑨没有连接！

Scala 代码:

```
import org.apache.flink.streaming.api.windowing.assigners.EventTimeSessionWindows;
import org.apache.flink.streaming.api.windowing.time.Time;
```

...

```

val orangeStream: DataStream[Integer] = ...
val greenStream: DataStream[Integer] = ...

orangeStream.join(greenStream)
  .where(elem => /* select key */)
  .equalTo(elem => /* select key */)
  .window(EventTimeSessionWindows.withGap(Time.milliseconds(1)))
  .apply { (e1, e2) => e1 + "," + e2 }

```

Java 代码:

```

import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.streaming.api.windowing.assigners.EventTimeSessionWindows;
import org.apache.flink.streaming.api.windowing.time.Time;

...

DataStream<Integer> orangeStream = ...
DataStream<Integer> greenStream = ...

orangeStream.join(greenStream)
  .where(<KeySelector>)
  .equalTo(<KeySelector>)
  .window(EventTimeSessionWindows.withGap(Time.milliseconds(1)))
  .apply (new JoinFunction<Integer, Integer, String> () {
    @Override
    public String join(Integer first, Integer second) {
      return first + "," + second;
    }
  });

```

间隔连接

间隔连接（Interval join）使用一个公共 key 将两个流(称它们为 A 和 B)的元素连接起来，其中流 B 的元素的时间戳与流 A 中的元素的时间戳相对间隔。

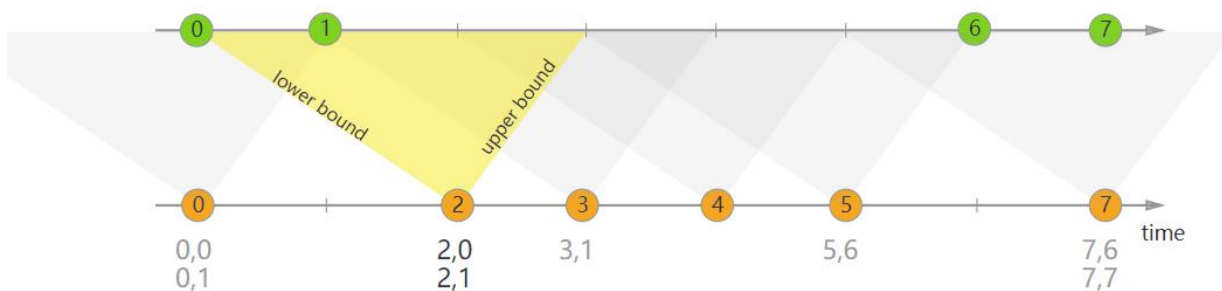
这个也可以更正式地表示如下：

- ❑ $b.timestamp \in [a.timestamp + lowerBound; a.timestamp + upperBound]$ ，或
- ❑ $a.timestamp + lowerBound \leq b.timestamp \leq a.timestamp + upperBound$

其中 a 和 b 是 A 和 B 的元素，它们共用一个 key。只要下界总是小于或等于上界，下界和上界都可以是负的或正的。间隔连接目前仅执行内连接。

当将一对元素传递给 ProcessJoinFunction 时，它们将被分配两个元素的较大时间戳(可以通过 ProcessJoinFunction.Context 访问该时间戳)。

注：interval join 目前只支持事件时间。



在上面的例子中，我们用下界为-2 毫秒和上界为+1 毫秒连接两个流“orange”和“green”。默认情况下，这些边界是包含的，不过可以使用 `lowerBoundExclusive()` 和 `upperBoundExclusive()` 方法来改变默认行为。可以看到，流 A 的每一个元素，都会和流 B 的一定时间范围的元素进行 JOIN。其中，上界和下界可以是负数，也可以是整数。Interval join 目前只支持 INNER JOIN。将连接后的元素传递给 `ProcessJoinFunction` 时，时间戳变为两个元素中最大的那个时间戳。

用公式表示的话，如下，表示图中三角形区域：

$$\square \quad \text{orangeElem.ts} + \text{lowerBound} \leq \text{greenElem.ts} \leq \text{orangeElem.ts} + \text{upperBound}$$

实现连接的代码示例如下。

Scala 代码：

```
import org.apache.flink.streaming.api.functions.co.ProcessJoinFunction;
import org.apache.flink.streaming.api.windowing.time.Time;

...

val orangeStream: DataStream[Integer] = ...
val greenStream: DataStream[Integer] = ...

orangeStream
    .keyBy(elem => /* select key */)
    .intervalJoin(greenStream.keyBy(elem => /* select key */))
    .between(Time.milliseconds(-2), Time.milliseconds(1))
    .process(new ProcessJoinFunction[Integer, Integer, String] {
        override def processElement(left: Integer,
                                    right: Integer,
                                    ctx: ProcessJoinFunction[Integer, Integer, String]#Context,
                                    out: Collector[String]): Unit = {
            out.collect(left + "," + right);
        }
    });
```

Java 代码：

```
import org.apache.flink.api.java.functions.KeySelector;
import org.apache.flink.streaming.api.functions.co.ProcessJoinFunction;
import org.apache.flink.streaming.api.windowing.time.Time;

...

DataStream<Integer> orangeStream = ...
```

```
DataStream<Integer> greenStream = ...

orangeStream
    .keyBy(<KeySelector>)
    .intervalJoin(greenStream.keyBy(<KeySelector>))
    .between(Time.milliseconds(-2), Time.milliseconds(1))
    .process (new ProcessJoinFunction<Integer, Integer, String>() {

        @Override
        public void processElement(Integer left, Integer right, Context ctx, Collector<String> out) {
            out.collect(first + "," + second);
        }
    });
```

3.8 低级操作

从之前的内容我们知道，Flink 的转换操作是无法访问事件的时间戳信息和水印信息的。例如我们常用的 `MapFunction` 转换操作就无法访问时间戳或者当前事件的事件时间。而在一些应用场景下，访问事件的时间戳信息和水印信息极为重要。因此，Flink `DataStream` API 提供了一系列的低级（Low-Level）转换操作，可以访问时间戳、水印以及注册定时事件。还可以输出特定的一些事件，例如超时事件等。这一类的低级 API，被称为 `ProcessFunction`。

`ProcessFunction` 将事件处理与计时器和状态相结合，使其成为流处理应用程序的强大组件。这是使用 Flink 创建事件驱动应用程序的基础。

3.8.1 ProcessFunction

`ProcessFunction` 是一个低级的流处理操作，允许访问所有（非循环）流应用程序的基本构件：

- ❑ **events**：数据流中的元素。
- ❑ **state**：状态，用于容错和一致性，仅用于 `keyed stream`。
- ❑ **timer**：定时器，支持事件时间和处理时间，仅用于 `keyed stream`。

`ProcessFunction` 可以被认为是一个访问 `keyed state` 和定时器的 `FlatMapFunction`。通过为输入流中接收的每个事件调用它来处理事件。

`ProcessFunction` 用来构建事件驱动的应用以及实现自定义的业务逻辑（使用之前的窗口函数和转换算子无法实现）。例如，Flink SQL 就是使用 `ProcessFunction` 实现的。

Flink 提供了 8 个 `Process Function`：

- ❑ **ProcessFunction**：用于 `DataStream`。
- ❑ **KeyedProcessFunction**：用于 `KeyedStream`，`keyBy` 之后的流处理。
- ❑ **CoProcessFunction**：用于 `connect` 连接的流。
- ❑ **ProcessJoinFunction**：用于 `join` 流操作。
- ❑ **BroadcastProcessFunction**：用于广播。
- ❑ **KeyedBroadcastProcessFunction**：`keyBy` 之后的广播。
- ❑ **ProcessWindowFunction**：窗口增量聚合。
- ❑ **ProcessAllWindowFunction**：全窗口聚合。

ProcessFunction 有两个回调函数需要实现，分别是 processElement 和 onTimer。其源码定义如下：

```
/**
 * A function that processes elements of a stream.
 *
 * <p>For every element in the input stream {@link #processElement(Object, Context, Collector)} is
 * invoked. This can produce zero or more elements as output. Implementations can also query the
 * time and set timers through the provided {@link Context}. For firing timers {@link #onTimer(long,
 * OnTimerContext, Collector)} will be invoked. This can again produce zero or more elements as
 * output and register further timers.
 *
 * <p><b>NOTE:</b> Access to keyed state and timers (which are also scoped to a key) is only
 * available if the {@code ProcessFunction} is applied on a {@code KeyedStream}.
 *
 * <p><b>NOTE:</b> A {@code ProcessFunction} is always a {@link
 * org.apache.flink.api.common.functions.RichFunction}. Therefore, access to the {@link
 * org.apache.flink.api.common.functions.RuntimeContext} is always available and setup and teardown
 * methods can be implemented. See {@link
 * org.apache.flink.api.common.functions.RichFunction#open(org.apache.flink.configuration.Configuration)}
 * and {@link org.apache.flink.api.common.functions.RichFunction#close()}.
 *
 * @param <I> Type of the input elements.
 * @param <O> Type of the output elements.
 */
@PublicEvolving
public abstract class ProcessFunction<I, O> extends AbstractRichFunction {

    private static final long serialVersionUID = 1L;

    /**
     * Process one element from the input stream.
     *
     * <p>This function can output zero or more elements using the {@link Collector} parameter and
     * also update internal state or set timers using the {@link Context} parameter.
     *
     * @param value The input value.
     * @param ctx A {@link Context} that allows querying the timestamp of the element and getting a
     *     {@link TimerService} for registering timers and querying the time. The context is only
     *     valid during the invocation of this method, do not store it.
     * @param out The collector for returning result values.
     * @throws Exception This method may throw exceptions. Throwing an exception will cause the
     *     operation to fail and may trigger recovery.
     */
    public abstract void processElement(I value, Context ctx, Collector<O> out) throws Exception;

    /**
     * Called when a timer set using {@link TimerService} fires.
     *
     * @param timestamp The timestamp of the firing timer.
     * @param ctx An {@link OnTimerContext} that allows querying the timestamp of the firing timer,

```

```

*   querying the {@link TimeDomain} of the firing timer and getting a {@link TimerService}
*   for registering timers and querying the time. The context is only valid during the
*   invocation of this method, do not store it.
* @param out The collector for returning result values.
* @throws Exception This method may throw exceptions. Throwing an exception will cause the
*   operation to fail and may trigger recovery.
*/
public void onTimer(long timestamp, OnTimerContext ctx, Collector<O> out) throws Exception {}

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

    /**
     * Timestamp of the element currently being processed or timestamp of a firing timer.
     *
     * <p>This might be {@code null}, for example if the time characteristic of your program is
     * set to {@link org.apache.flink.streaming.api.TimeCharacteristic#ProcessingTime}.
     */
    public abstract Long timestamp();

    /** A {@link TimerService} for querying time and registering timers. */
    public abstract TimerService timerService();

    /**
     * Emits a record to the side output identified by the {@link OutputTag}.
     *
     * @param outputTag the {@code OutputTag} that identifies the side output to emit to.
     * @param value The record to emit.
     */
    public abstract <X> void output(OutputTag<X> outputTag, X value);
}

/**
 * Information available in an invocation of {@link #onTimer(long, OnTimerContext, Collector)}.
 */
public abstract class OnTimerContext extends Context {
    /** The {@link TimeDomain} of the firing timer. */
    public abstract TimeDomain timeDomain();
}
}

```

对每个传入的事件调用 `processElement`；当计时器触发时调用 `onTimer`。这些计时器可以是事件时间定时器，也可以是处理时间定时器。定时器可让应用程序对在处理时间和事件时间中的变化进行响应。`processElement` 和 `onTimer` 都提供了一个上下文对象，可用于与 `TimerService`（以及其他内容）进行交互。这两个回调也都传递给一个 `Collector`，该 `Collector` 可用于向下游发出结果。

要对两个输入流进行低级操作，应用程序可以使用 `CoProcessFunction` 或 `KeyedCoProcessFunction`。此函数绑定到两个不同的输入，并且对于来自这两个不同输入的记录，获取对 `processElement1(…)` 和

processElement2(...)的单独调用。

实现低级别的 join 连接，通常遵循如下的模式：

- ❑ 为每一个输入（或所有输入）创建一个状态对象；
- ❑ 从某个输入中接收元素时更新其状态；
- ❑ 接收到来自其他输入的元素后，探测状态并生成连接的结果。

例如，我们需要将客户数据连接到金融交易数据，并同时保留客户数据的状态。如果希望在面对无顺序事件时有完整和确定性的连接，那么可以使用定时器在客户数据流的水印已经过了该交易的时间时，评估并发出该交易的连接。

3.8.2 KeyedProcessFunction 示例

KeyedProcessFunction 是 RichFunction 的一种。其源码定义如下：

```
/**
 * A keyed function that processes elements of a stream.
 *
 * <p>For every element in the input stream {@link #processElement(Object, Context, Collector)} is
 * invoked. This can produce zero or more elements as output. Implementations can also query the
 * time and set timers through the provided {@link Context}. For firing timers {@link #onTimer(long,
 * OnTimerContext, Collector)} will be invoked. This can again produce zero or more elements as
 * output and register further timers.
 *
 * <p><b>NOTE:</b> Access to keyed state and timers (which are also scoped to a key) is only
 * available if the {@code KeyedProcessFunction} is applied on a {@code KeyedStream}.
 *
 * <p><b>NOTE:</b> A {@code KeyedProcessFunction} is always a {@link
 * org.apache.flink.api.common.functions.RichFunction}. Therefore, access to the {@link
 * org.apache.flink.api.common.functions.RuntimeContext} is always available and setup and teardown
 * methods can be implemented. See {@link
 * org.apache.flink.api.common.functions.RichFunction#open(org.apache.flink.configuration.Configuration)}
 * and {@link org.apache.flink.api.common.functions.RichFunction#close()}.
 *
 * @param <K> Type of the key.
 * @param <I> Type of the input elements.
 * @param <O> Type of the output elements.
 */
@PublicEvolving
public abstract class KeyedProcessFunction<K, I, O> extends AbstractRichFunction {

    private static final long serialVersionUID = 1L;

    /**
     * Process one element from the input stream.
     *
     * <p>This function can output zero or more elements using the {@link Collector} parameter and
     * also update internal state or set timers using the {@link Context} parameter.
     *
     * @param value The input value.
```

```

    * @param ctx A {@link Context} that allows querying the timestamp of the element and getting a
    *   {@link TimerService} for registering timers and querying the time. The context is only
    *   valid during the invocation of this method, do not store it.
    * @param out The collector for returning result values.
    * @throws Exception This method may throw exceptions. Throwing an exception will cause the
    *   operation to fail and may trigger recovery.
    */
    public abstract void processElement(I value, Context ctx, Collector<O> out) throws Exception;

    /**
     * Called when a timer set using {@link TimerService} fires.
     *
     * @param timestamp The timestamp of the firing timer.
     * @param ctx An {@link OnTimerContext} that allows querying the timestamp, the {@link
     *   TimeDomain}, and the key of the firing timer and getting a {@link TimerService} for
     *   registering timers and querying the time. The context is only valid during the invocation
     *   of this method, do not store it.
     * @param out The collector for returning result values.
     * @throws Exception This method may throw exceptions. Throwing an exception will cause the
     *   operation to fail and may trigger recovery.
     */
    public void onTimer(long timestamp, OnTimerContext ctx, Collector<O> out) throws Exception {}

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

        /**
         * Timestamp of the element currently being processed or timestamp of a firing timer.
         *
         * <p>This might be {@code null}, for example if the time characteristic of your program is
         * set to {@link org.apache.flink.streaming.api.TimeCharacteristic#ProcessingTime}.
         */
        public abstract Long timestamp();

        /** A {@link TimerService} for querying time and registering timers. */
        public abstract TimerService timerService();

        /**
         * Emits a record to the side output identified by the {@link OutputTag}.
         *
         * @param outputTag the {@code OutputTag} that identifies the side output to emit to.
         * @param value The record to emit.
         */
        public abstract <X> void output(OutputTag<X> outputTag, X value);

        /** Get key of the element being processed. */
        public abstract K getCurrentKey();
    }

```

```

    }

    /**
     * Information available in an invocation of {@link #onTimer(long, OnTimerContext, Collector)}.
     */
    public abstract class OnTimerContext extends Context {
        /** The {@link TimeDomain} of the firing timer. */
        public abstract TimeDomain timeDomain();

        /** Get key of the firing timer. */
        @Override
        public abstract K getCurrentKey();
    }
}

```

每次调用 `processElement(...)` 函数 都可以获得一个 `Context` 对象, 通过该对象可以访问元素的事件时间戳以及 `TimerService`。可以使用 `TimerService` 为将来的事件时间/处理时间实例注册回调。对于事件时间计时器, 当当前水印前进到或超过定时器的时间戳时, 将调用 `onTimer(...)` 方法, 而对于处理时间计时器, 当挂钟时间达到指定时间时, 将调用 `onTimer(...)` 方法。在调用期间, 所有状态的范围再次限定为创建定时器所用的 `key`, 从而允许定时器操作 `keyed state`。

作为一个 `RichFunction`, 它可以访问使用托管 `keyed state` 所需的 `open` 和 `getRuntimeContext` 方法。对于容错状态, `KeyedProcessFunction` 可以通过 `RuntimeContext` 访问 Flink 的 `keyed state`, 这与其他有状态函数访问 `keyed state` 的方式类似。

作为 `ProcessFunction` 的扩展, `KeyedProcessFunction` 在其 `onTimer(...)` 方法中通过 `OnTimerContext` 提供了对计时器 `key` 的访问。

Java 代码:

```

@Override
public void onTimer(long timestamp, OnTimerContext ctx, Collector<OUT> out) throws Exception {
    K key = ctx.getCurrentKey();
    // ...
}

```

Scala 代码:

```

override def onTimer(timestamp: Long, ctx: OnTimerContext, out: Collector[OUT]): Unit = {
    var key = ctx.getCurrentKey
    // ...
}

```

在下面的示例中, `KeyedProcessFunction` 维护每个 `key` 的计数, 并在每过 10 秒钟 (以事件时间) 而未更新该 `key` 时, 发出一个 `key/count` 对:

- ☐ 监听 `Socket` 数据源, 获取输入字符串。
- ☐ 把 `key` (单词)、计数和最后修改时间戳存储在一个 `ValueState` 状态中, `ValueState` 的作用域是通过 `key` 隐式确定的。
- ☐ 对于每个记录, `KeyedProcessFunction` 递增计数器并设置最后修改时间戳。
- ☐ 该函数还安排了一个 10 秒后的回调 (以事件时间)。
- ☐ 在每次回调时, 它根据存储的计数的最后修改时间检查回调的事件时间时间戳, 并在它们匹配

时发出 key/count（即，在 10 秒钟内这个单词没有再次出现，就把这个单词和它出现的总次数发送到下游算子）。

注：这个简单的示例可以用会话窗口实现。我们在这里使用 KeyedProcessFunction 来说明它提供的基本模式。

【示例】维护数据流中每个 key 的计数，并在每过 10 秒钟（以事件时间）而未更新该 key 时，发出一个 key/count 对。

Scala 代码：

```
package com.xueai8.scala.ch03

import java.text.SimpleDateFormat
import java.time.Duration
import java.util.Date
import org.apache.flink.api.common.eventtime.{SerializableTimestampAssigner, WatermarkStrategy}
import org.apache.flink.api.common.state.{ValueState, ValueStateDescriptor}
import org.apache.flink.streaming.api.functions.KeyedProcessFunction
import org.apache.flink.streaming.api.scala._
import org.apache.flink.util.Collector

/**
 * 维护每个 key 的计数，并在每过 10 秒钟（以事件时间）而未更新该 key 时，发出一个 key/count 对：
 */
object KeyedProcessFunDemo {

  // 存储在状态中的数据类型
  case class CountWithTimestamp(key: String, count: Long, lastModified: Long)

  def main(args: Array[String]) {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    // 并行度 1
    env.setParallelism(1)

    // 源数据流
    val stream = env
      .socketTextStream("192.168.190.133", 9999)
      .flatMap(_.split("\\W+"))
      .map(_._1)

    // 为事件分配时间戳和水印
    stream
      .assignTimestampsAndWatermarks(WatermarkStrategy
        .forMonotonousTimestamps[(String, Int)]()
        .withTimestampAssigner(new SerializableTimestampAssigner[(String, Int)] {
          override def extractTimestamp(t: (String, Int), ts: Long): Long = System.currentTimeMillis()
        })
        .withIdleness(Duration.ofSeconds(5))
      )
      .keyBy(_._1)
```

```

        .process(new CountWithTimeoutFunction())
        .print()

// 执行程序
env.execute("Process Function")
}

/**
 * KeyedProcessFunction 的子类，维护计数和超时。
 * 它的作用是将每个单词最新出现时间记录到 backend，并创建定时器，
 * 定时器触发的时候，检查这个单词距离上次出现是否已经达到 10 秒，如果是，就发射给下游算子
 */
class CountWithTimeoutFunction extends KeyedProcessFunction[String, (String, Int), (String, Long)] {

    /**
     * 首先获得由这个处理函数（process function）维护的状态
     * 通过 RuntimeContext 访问 Flink 的 keyed state
     */
    lazy val state: ValueState[CountWithTimestamp] = getRuntimeContext.getState(
        new ValueStateDescriptor[CountWithTimestamp]("myState", classOf[CountWithTimestamp])
    )

    /**
     * 对于在输入流中接收到的每一个事件，此函数就会被调用以处理该事件
     * @param value      输入元素
     * @param ctx        上下文环境
     * @param out
     * @throws Exception
     */
    override def processElement(value: (String, Int),
                                ctx: KeyedProcessFunction[String, (String, Int), (String, Long)]#Context,
                                out: Collector[(String, Long)]): Unit = {

        // 初始化或检索/更新状态
        val current = state.value match {
            case null =>
                CountWithTimestamp(value._1, 1, ctx.timestamp) // 如果是第一个事件，设初始计数为 1
            case CountWithTimestamp(key, count, lastModified) =>
                // 先撤销之前的定时器
                // ctx.timerService().deleteEventTimeTimer(lastModified + 10000);
                ctx.timerService.deleteProcessingTimeTimer(lastModified + 10000)
                CountWithTimestamp(key, count + 1, ctx.timestamp) // 如果不是第一个事件，则累加
        }

        // 将修改过后的状态写回
        state.update(current)

        // 从当前事件时间开始安排下一个定时器 10 秒
        // 为当前单词创建定时器，十秒后触发
        val timer = current.lastModified + 10000
    }
}

```

```
//      ctx.timerService.registerEventTimeTimer(timer)
      ctx.timerService.registerProcessingTimeTimer(timer)

      // 打印所有信息，用于核对数据正确性
      println(s"process, ${current.key}, ${current.count}, " +
        s"lastModified : ${current.lastModified}, (${time(current.lastModified)}), " +
        s"timer : ${timer} (${time(timer)})\n")
    }

    /** 定时器触发后执行的方法
     * 如果一分钟内没有新来的相同的单词，则发出 key/count 对
     *
     * @param timestamp 这个时间戳代表的是该定时器的触发时间
     * @param ctx
     * @param out
     * @throws Exception
     */
    override def onTimer(timestamp: Long,
      ctx: KeyedProcessFunction[String, (String, Int), (String, Long)]#OnTimerContext,
      out: Collector[(String, Long)]): Unit = {
      // 取得当前单词
      val currentKey = ctx.getCurrentKey

      // 获取调度此计时器的 key 的状态（即此单词的状态）
      val result = state.value

      // 当前元素是否已经连续 10 秒未出现的标志
      var isTimeout = false

      // timestamp 是定时器触发时间
      // 如果等于最后一次更新时间+10 秒，就表示这 10 秒内已经收到过该单词了，
      // 这种连续 10 秒没有出现的元素，被发送到下游算子
      result match {
        case CountWithTimestamp(key, count, lastModified) if timestamp >= lastModified + 10000 =>
          out.collect((key, count)) // 发出 key/count 对
          isTimeout = true
        case _ =>
      }

      // 打印数据，用于核对是否符合预期
      println(s"ontimer, ${currentKey}, ${result.count}, " +
        s"lastModified : ${result.lastModified}, (${time(result.lastModified)}), " +
        s"stamp : ${timestamp} (${time(timestamp)}),isTimeout: ${isTimeout}\n")
    }
  }

  private def time(timeStamp: Long) =
    new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date(timeStamp))
}

```

Java 代码：


```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.eventtime.WatermarkStrategy;
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.state.ValueState;
import org.apache.flink.api.common.state.ValueStateDescriptor;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.KeyedProcessFunction;
import org.apache.flink.util.Collector;
import org.apache.flink.util.StringUtils;

import java.text.SimpleDateFormat;
import java.time.Duration;
import java.util.Date;

/**
 * 维护每个 key 的计数，并在每过 10 秒钟（以事件时间）而未更新该 key 时，发出一个 key/count 对：
 */
public class KeyedProcessFunDemo {

    // 存储在状态中的数据类型，POJO 类
    public static class CountWithTimestamp {
        public String key;
        public long count;
        public long lastModified;
    }

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 并行度 1
        env.setParallelism(1);

        // 源数据流
        DataStream<Tuple2<String, Integer>> stream = env
            .socketTextStream("192.168.190.133", 9999)
            .flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
                @Override
                public void flatMap(String s, Collector<Tuple2<String, Integer>> collector)
                    throws Exception {
                    if (StringUtils.isNullOrWhitespaceOnly(s)) {
                        System.out.println("invalid line");
                        return;
                    }

                    for (String word : s.split("\\W+")){
```

```

        collector.collect(new Tuple2<>(word,1));
    }
}
});

stream
    .assignTimestampsAndWatermarks(WatermarkStrategy
        .<Tuple2<String, Integer>>forMonotonousTimestamps()
        .withTimestampAssigner((event, ts) -> System.currentTimeMillis())
        .withIdleness(Duration.ofSeconds(5))
    )
    .keyBy(t -> t.f0)
    .process(new CountWithTimeoutFunction())
    .print();

// 执行
env.execute("flink streaming job");
}

/**
 * KeyedProcessFunction 的子类，维护计数和超时。
 * 它的作用是将每个单词最新出现时间记录到 backend，并创建定时器，
 * 定时器触发的时候，检查这个单词距离上次出现是否已经达到 10 秒，如果是，就发射给下游算子
 */
public static class CountWithTimeoutFunction extends KeyedProcessFunction<
    String,                                     // key
    Tuple2<String, Integer>,                     // input
    Tuple2<String, Long>> {                       // output

    // 由此函数所维护的存储状态
    private ValueState<CountWithTimestamp> state;

    // 首先获得由这个处理函数维护的状态
    // 通过 RuntimeContext 访问 Flink 的 keyed state
    @Override
    public void open(Configuration parameters) throws Exception {
        // 初始化状态，状态名称是 myState
        state = getRuntimeContext().getState(
            new ValueStateDescriptor<>("myState", CountWithTimestamp.class));
    }

    /**
     * 对于在输入流中接收到的每一个事件，此函数就会被调用以处理该事件
     * @param value          输入元素
     * @param context        上下文环境
     * @param out
     * @throws Exception
     */
    @Override
    public void processElement(

```

```
        Tuple2<String, Integer> value,
        Context context,
        Collector<Tuple2<String, Long>> out) throws Exception {

    // 取得当前是哪个单词
    // String currentKey = context.getCurrentKey();

    // 从 backend 取得当前单词的 myState 状态
    CountWithTimestamp current = state.value();

    // 如果 myState 还从未没有赋值过，就在此初始化
    if (current == null) {
        current = new CountWithTimestamp();
        current.key = value.f0;
    }

    // 先撤销之前的定时器
    // context.timerService().deleteEventTimeTimer(current.lastModified + 10000);
    context.timerService().deleteProcessingTimeTimer(current.lastModified + 10000);

    // 更新状态计数值(单词数量加 1)
    current.count++;

    // 将状态的时间戳设置为记录分配的事件时间戳
    // 取当前元素的时间戳，作为该单词最后一次出现的时间
    current.lastModified = context.timestamp();

    // 重新保存到 backend，包括该单词出现的次数，以及最后一次出现的时间
    state.update(current);

    // 为当前单词创建定时器，10 秒后后触发
    long timer = current.lastModified + 10000;
    context.timerService().registerEventTimeTimer(timer);
    // context.timerService().registerProcessingTimeTimer(timer);

    // 打印所有信息，用于核对数据正确性
    System.out.println(String.format("process, %s, %d, lastModified : %d (%s), timer : %d (%s)\n",
        current.key,
        current.count,
        current.lastModified,
        time(current.lastModified),
        timer,
        time(timer)));
}

/** 定时器触发后执行的方法
 * 如果一分钟内没有新来的相同的单词，则发出 key/count 对
 *
 * @param timestamp 这个时间戳代表的是该定时器的触发时间
 * @param ctx
```

```

    * @param out
    * @throws Exception
    */
    @Override
    public void onTimer(
        long timestamp,
        OnTimerContext ctx,
        Collector<Tuple2<String, Long>> out) throws Exception {

        // 取得当前单词
        String currentKey = ctx.getCurrentKey();

        // 获取调度此计时器的 key 的状态（即此单词的状态）
        CountWithTimestamp result = state.value();

        // 当前元素是否已经连续 10 秒未出现的标志
        boolean isTimeout = false;

        // timestamp 是定时器触发时间
        // 超过 10 秒钟该单词（key）没有再出现，就发送给下游
        if (timestamp >= result.lastModified + 10000) {
            out.collect(new Tuple2<>(result.key, result.count));
            isTimeout = true;
        }

        // 打印数据，用于核对是否符合预期
        System.out.println(String.format("ontimer, %s, %d, lastModified : %d (%s), stamp : %d (%s), isTimeout : %s\n\n",
            currentKey,
            result.count,
            result.lastModified,
            time(result.lastModified),
            timestamp,
            time(timestamp),
            String.valueOf(isTimeout)));
    }

    private static String time(long timeStamp) {
        return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date(timeStamp));
    }
}

```

在 `processElement` 方法中，`state.value()` 可以取得当前单词的状态，`state.update(current)` 可以设置当前单词的状态执行以上代码。

请按以下步骤执行上面的程序：

- 1) 在控制台执行命令 `nc -lk 9999`，这样就可以从控制台向 9999 端口发送字符串了；
- 2) 在 IDEA 上直接执行上同的程序代码，程序运行就开始监听本机的 9999 端口了；
- 3) 在 netcat 的控制台输入 `aaa` 再回车，连续两次，中间间隔不要超过 10 秒；然后再输入 `ttt` 回车。

如下图所示：

```
[hduser@xueai8 ~]$ nc -lk 9999
aaa
aaa
ttt
```

4) 观察 IDEA 控制台输出结果，如下图所示：

```
process, aaa, 1, lastModified : 1629619520806 (2021-08-22 16:05:20), timer : 1629619530806 (2021-08-22 16:05:30)

process, aaa, 2, lastModified : 1629619523038 (2021-08-22 16:05:23), timer : 1629619533038 (2021-08-22 16:05:33)

process, ttt, 1, lastModified : 1629619549431 (2021-08-22 16:05:49), timer : 1629619559431 (2021-08-22 16:05:59)

(aaa, 2)
ontimer, aaa, 2, lastModified : 1629619523038 (2021-08-22 16:05:23), stamp : 1629619533038 (2021-08-22 16:05:33), isTimeout : true
```

3.8.3 案例：服务器故障检测报警程序

对于 Hadoop 运维人员，需要监控生产环境大数据各个组件的状态信息，由于 Regionserver 经常挂掉，这里模拟监控 Regionserver 的状态信息，每次 Regionserver 上线和下线都会发送一个状态信息。这里如果 Regionserver 挂掉后，服务自启动功能在 30 秒以内没有将 Regionserver 拉起（这里只是为了测试使用 30 秒），那么就进行持续的告警，直到收到上线消息，告警取消。

1) 这里消息模拟从 socket 接收服务器告警消息，消息格式包含三个字段分别是：主机名 hostname，告警时间 time，状态 status（RUNNING 服务正常，DEAD 服务停止）；

2) 接收消息之后对数据流按照主机名进行分组，对于状态为 DEAD 的消息，设置定时器 30 秒以内如果状态不恢复为 RUNNING，则定时进行告警，如果 30 秒内恢复 RUNNING 状态，则认为上一条消息是误报，则删除定时器，取消报警。

Scala 代码：

Java 代码：

执行步骤如下：

1) 启动 nc 服务器：

```
$ nc -lk 9999
```

2) 依次输入以下信息，模拟服务器监控事件：

3) 执行结果：

```
currentStatus:RUNNING
lastStatus:null
```

```
currentStatus:DEAD
lastStatus:RUNNING
```

```
currentStatus:DEAD
lastStatus:DEAD
```

主机 IP：192.168.1.101，两次 Regionserver 状态监测宕机，请监测！

3.9 状态和容错

状态函数和运算符在各个元素/事件的处理过程中存储数据,这使得状态成为任何类型的复杂操作的关键组成部分。例如:

- ❑ 当应用程序搜索某些事件模式时, 状态将存储到目前为止遇到的事件序列。
- ❑ 当以每分钟/小时/天的速度聚合事件时, 状态将保留挂起的聚合。
- ❑ 当在一个数据点流上训练一个机器学习模型时, 状态保持模型参数的当前版本。
- ❑ 当需要管理历史数据时, 该状态允许对过去发生的事件进行有效访问。

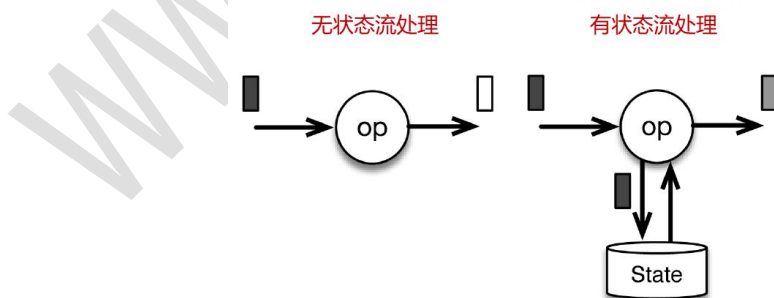
3.9.1 状态运算

Flink 的操作可以是无状态的, 也可以是有状态的。

如果处理事件的结果只与事件本身的内容相关, 则称为无状态操作。无状态程序查看每个单独的事件, 并基于最后一个事件创建一些输出。例如, 流程序可能从传感器接收温度读数, 并在温度超过 90 度时发出警报。

如果结果与先前处理的事件相关, 称为有状态操作。有状态操作这意味着一个事件的处理方式可能取决于它之前发生的所有事件的累积效果。状态可以用于简单的事情, 例如计算仪表盘上显示的每分钟事件数; 也可以用于更复杂的事情, 例如计算欺诈检测模型的功能。有状态程序基于多个事件一起创建输出。任何重要的数据处理, 如基本聚合, 都是有状态的处理。有状态程序的场景包括:

- ❑ 所有类型的窗口。例如, 获取传感器在过去一小时内报告的平均温度是一个有状态的计算。当按分钟/小时/天聚合事件时, 状态保存着挂起的聚合。
- ❑ 用于复杂事件处理(CEP)的各种状态机。例如, 在接收到两个在 1 分钟内相差超过 20 度的温度读数后创建一个警报是一个有状态的计算。
- ❑ 当应用程序搜索某些事件模式时, 状态将存储到目前为止遇到的事件序列。
- ❑ 当通过数据点流训练机器学习模型时, 状态保持模型参数的当前版本。
- ❑ 流之间以及流与静态或缓慢变化的表之间的各种连接。
- ❑ 当需要管理历史数据时, 状态允许对过去发生的事件进行有效访问。

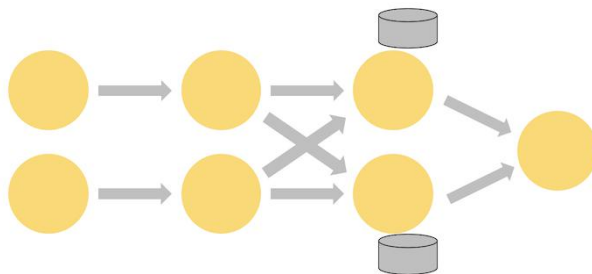


Flink 应用程序在分布式集群上并行运行。给定操作符的不同并行实例将在不同的线程中独立执行, 并且通常运行在不同的机器上。

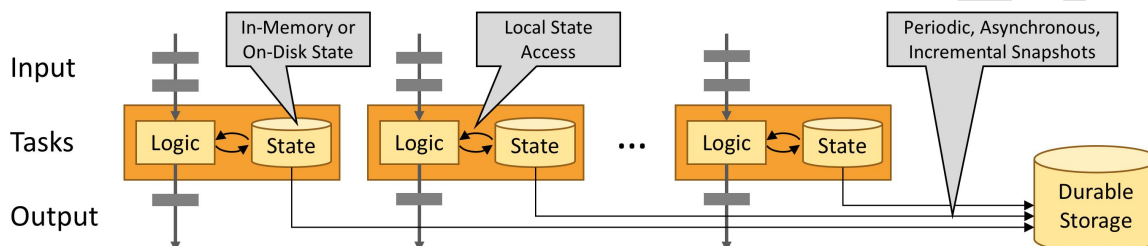
状态操作符的并行实例集实际上是一个分片键值存储。每个并行实例负责处理特定键组的事件, 这些键的状态保存在本地。

下图显示了在作业图的前三个操作符中并行度为 2 的作业运行, 在并行度为 1 的接收器中终止。第三个操作符是有状态的, 可以看到在第二和第三个操作符之间发生了全连接的网络 shuffle。这样做是为

了通过某个键对流进行分区，以便所有需要一起处理的事件都将被一起处理。



状态总是在本地访问，Flink 针对本地状态访问进行了优化，这有助于 Flink 应用程序实现高吞吐量和低延迟。可以选择将状态保存在 JVM 堆中，或者如果它太大，则在访问高效的磁盘数据结构中维护。



3.9.2 状态的类型

Flink 提供了两种基本类型的状态：keyed state 和 operator State。

- ❑ keyed state 始终是相对于 key 的，并且只能在 keyed stream 上的函数和算子中使用。
- ❑ 使用 operator State（或 non-keyed state），每个算子状态都绑定到一个并行算子实例。

keyed state

Keyd state 是专门针对 keyed stream 能够使用的状态。换句话说，就是按照 key 进行分区之后的操作才能够使用状态。Keyed state 被维护在可以被认为是嵌入式键/值存储的地方。状态被严格地与状态操作符读取的流一起划分和分布。因此，访问 key/value 状态只可能在 keyed stream 上，即在执行 keyBy 分区数据交换之后，并且仅限于与当前事件 key 相关的值。

Keyed state 可以确保所有的状态更新都是本地操作，从而保证了状态数据的一致性而不会产生事务开销。因为 keyed state 按照 key 进行维护和访问的，Flink 会为每一个 key 都维护一个状态实例，该状态实例总是位于处理该 key 记录的算子任务上，因此同一个 key 的记录可以访问到一样的状态。这种对流和状态的 key 进行对齐的方式还允许 Flink 重新分配状态并透明地调整流分区。

Flink 提供了不同存储结构类型的 keyed state，这些存储结构官方称为“状态原语”。可用的状态原语有：

这些状态原语具体说明如下：

- ❑ ValueState<T>：这将保留一个可以更新和检索的值(如前所述，作用域为输入元素的键，因此操作看到的每个键可能都有一个值)。可以使用 update(T)设置该值，并使用 T value()检索该值。
- ❑ ListState<T>：它保存了一个元素列表。可以在当前存储的所有元素上追加元素并检索一个

Iterable。使用 `add(T)` 或 `addAll(List<T>)` 添加元素，可使用 `Iterable<T> get()` 检索 Iterable。还可以使用 `update(List<T>)` 覆盖现有列表。

- ❑ **ReducingState<T>**：这将保留一个值，该值表示添加到状态的所有值的聚合。该接口类似于 `ListState`，但使用 `add(T)` 添加的元素使用指定的 `ReduceFunction` 缩减为聚合。
- ❑ **AggregatingState<IN, OUT>**：这将保持单个值，该值表示添加到状态的所有值的聚合，聚合类型可能与添加到该状态的元素的类型不同。这个接口与 `ListState` 类似，但使用 `add(IN)` 添加的元素是使用指定的 `AggregateFunction` 聚合的。
- ❑ **MapState<UK, UV>**：这保存了一个映射列表。可以将 `key-value` 对放入状态，并在所有当前存储的映射上检索一个 `Iterable`。使用 `put(UK, UV)` 或 `putAll(Map<UK, UV>)` 添加映射。可以使用 `get(UK)` 检索与用户 `key` 关联的值。可以使用 `entries()`、`keys()`、`values()` 分别检索映射、`keys` 和 `values` 的可迭代视图。还可以使用 `isEmpty()` 来检查该 `map` 是否包含有任何 `key-value` 映射。

所有类型的状态都有一个方法 `clear()`，该方法为当前活动的 `key`（即输入元素的 `key`）清除状态。

务必记住，这些状态对象仅用于与状态交互。状态不一定存储在内部，但可能驻留在磁盘或其他地方。另外，从状态获取的值取决于输入元素的 `key`。因此，如果涉及的 `key` 不同，那么在一次用户函数调用中得到的值可能与在另一次调用中得到的值不同。

operator state

`operator state` 是一种 `non-keyed state`，与并行的操作算子实例相关联。`operator state` 的作用域是某个算子任务，这意味着所有在同一个并行任务之内的记录都能访问到相同的状态。算子状态不能通过其他任务访问。如下图所示。

而当并行度发生调整时，需要在操作符（算子）的并行度上重新分配状态。在大多数的流处理程序开发中，我们很少用到 `operator state`，因为用户定义的函数通常不需要 `non-keyed state`。这个特性最常用于实现 `source` 和 `sink`，而 `source`、`sink` 都是没有 `key` 的，就需要用到 `operator state` 了。

Keyed State Vs Operator State

`Keyed State` 和 `Operator State` 的对比如下表所示。

原始状态和托管状态

根据状态数据是否被受 Flink 管理，Flink 又将状态（`Keyed State` 和 `Operator State`）分为两种形式：托管状态和原生状态。

托管状态在由 Flink 运行时控制的数据结构中表示，如内部哈希表或 `RocksDB`。例如 `ValueState`、`ListState` 等。Flink 的运行时对状态进行序列化编码并将它们写入检查点。Flink 管理的状态存储在状态后端（`state backend`）。

原生状态是运算符（算子）在自己的数据结构中保持的状态。当 `checkpoint` 时，它们只向检查点写入一个字节序列。Flink 对状态的数据结构一无所知，只看到原生字节。

状态类型	托管状态	原生状态
Keyed state	<code>ValueState<T></code> <code>ListState<T></code>	字节数组: <code>byte[]</code>

	MapState<UK, UV> ReducingState AggregatingState	
Operator state	ListState BroadcastState	字节数组: byte[]

所有 `DataStream` 函数都可以使用托管状态，但是原生状态接口只能在实现自定义运算符（算子）时使用。建议使用托管状态而不是原生状态，因为通过托管状态，当并行性发生变化时，Flink 能够自动重新分配状态，而且还可以进行更好的内存管理。

在下表中列出了托管状态与原生状态的区别。

状态类型	托管状态	原生状态
状态管理方式	Flink Runtime托管，自动存储、自动恢复，内存管理上有优化	用户自己管理，需要自己进行序列化
数据结构	Flink 提供了常用数据结构，如 <code>ValueState</code> 、 <code>ListState</code> 、 <code>MapState</code> 等	字节数组: byte[]
使用场景	所有 <code>DataStream</code> 函数都可以使用	自定义 <code>Operator</code> 时

Flink 为它管理的状态提供了如下一些特性：

- ☐ 本地化：Flink 状态保持在处理它的机器的本地，并且可以以内存速度访问；
- ☐ 持久化：Flink 状态是容错的，也就是说，它是定期自动检查点，并在失败时还原；
- ☐ 垂直扩展：Flink 状态可以存储在嵌入的 RocksDB 实例中，这些实例通过添加更多本地磁盘进行扩展；
- ☐ 水平扩展：Flink 状态会随着集群的增长和收缩而重新分布；
- ☐ 可查询的：可以通过 `Queryable State API` 从外部查询 Flink 状态。

Flink 提供了简单易用的 API 来存储和获取状态。

3.9.3 使用托管的 Keyed State

托管的 `Keyed State` 接口提供对不同类型状态的访问，这些状态的范围都是当前输入元素的 `key`。这意味着这种状态只能在 `Keyed Stream` 上使用，而 `Keyed Stream` 可以通过 `stream.keyBy(...)` 创建。

要获得状态句柄，必须创建一个 `StateDescriptor`，它代表状态描述符。`StateDescriptor` 持有状态的名称（名称必须唯一）、状态的值的类型，另外可能还有一个指定的函数，比如 `ReduceFunction`。每种状态都对应各自的描述符，根据希望检索的状态类型，可以创建 `ValueStateDescriptor`、`ListStateDescriptor`、`ReducingStateDescriptor`、`MapStateDescriptor` 或 `AggregatingStateDescriptor` 等。

通过描述符使用 `RuntimeContext` 访问相应的状态，而 `RuntimeContext` 只有在 `RichFunction` 中才能获取。在 `RichFunction` 中可以使用 `RuntimeContext`，它具有以下访问状态的方法：

- ☐ `ValueState<T> getState(ValueStateDescriptor<T>)`
- ☐ `ReducingState<T> getReducingState(ReducingStateDescriptor<T>)`
- ☐ `ListState<T> getListState(ListStateDescriptor<T>)`
- ☐ `AggregatingState<IN, OUT> getAggregatingState(AggregatingStateDescriptor<IN, ACC, OUT>)`
- ☐ `MapState<UK, UV> getMapState(MapStateDescriptor<UK, UV>)`

所以要想使用 `keyed state`，用户编写的类必须继承 `RichFunction` 或者其子类。

深度

当计算依赖于累积给定 `key` 的数据时，需要将所有这些数据合并在一起。这可以通过两种方式完成：

- ❑ 将数据存储在 `ListState` 容器中，等待会话结束，并在会话结束时将所有数据合并在一起。
- ❑ 使用 `ReducingState` 在每个新事件到达时，将其与之前的事件合并。

使用哪种状态取决于在 `WindowedStream` 上运行的功能：使用 `ProcessWindowFunction` 的 `process` 调用将使用 `ListState`，而使用 `ReduceFunction` 的 `reduce` 调用则将使用 `ReducingState`。

`ReducingState` 的优点非常明显：不存储窗口处理之前的所有数据，而是在单个记录中不断地对其进行聚合。这通常会导致状态更小，取决于在 `reduce` 操作期间会丢弃多少数据。但是，在特定的情况下，它在存储方面几乎没有改善，比如，与历史会话存储的 7 天数据相比，该状态的大小可以忽略不计。这时，通过使用 `ListState` 反而可以提高性能！原因是，每次新事件到来时，连续的 `reduce` 操作都需要对数据进行反序列化和序列化。这可以在 `RocksDBReducingState` 的 `add` 函数中看到，该函数会调用 `getInternal`，从而导致数据反序列化。

【示例 1】数据去重。假设我们有一个想要去重的事件流，要求我们只保留每个 `key` 的第一个事件。在下面这个应用程序中，我们通过名为 `Deduplicator` 的 `RichFlatMapFunction` 来实现去重。

分析：

这需要使用 Flink 的 `keyed state` 接口，来记住每个 `key` 是否已经有过一个事件。像这样使用 `key stream` 时，Flink 将为所管理的每个状态项维护一个 `key/value` 存储（即托管状态）。Flink 支持几种不同类型的 `keyed state`，本例使用最简单的一种，即 `ValueState`。这意味着对于每个键，Flink 将存储单个对象—在本例中是一个 `Boolean` 类型的对象。

实现了 `RichFlatMapFunction` 接口的 `Deduplicator` 类有两个方法：`open()` 和 `flatMap()`。其中 `open` 方法通过定义 `ValueStateDescriptor<Boolean>` 来建立托管状态的使用。构造函数的参数为这个 `keyed state` 项指定了一个名称“`keyHasBeenSeen`”，并提供可用于序列化这些对象的信息（在本例中为 `Types.BOOLEAN`）。

Scala 代码：

Java 代码：

执行以上代码，输出结果如下：

```
6> Event{key='a', timestamp=123}
2> Event{key='b', timestamp=223}
4> Event{key='c', timestamp=523}
```

当 `flatMap` 方法调用 `keyHasBeenSeen.value()` 时，Flink 的运行时在上下文中查找键的这段状态的值，只有当它为 `null` 时，才会继续将事件收集到输出中。在本例中，它还将 `keyHasBeenSeen` 更新为 `true`。

当部署到分布式集群时，将有许多这个 `Deduplicator` 的实例，每个实例负责整个键空间的一个不相交子集。因此，当我们看到 `ValueState` 的单个项时，例如：

```
ValueState<Boolean> keyHasBeenSeen;
```

这不仅表示一个 `Boolean` 值，而且表示一个分布式、分片的 `key/value` 存储。

【示例 2】实现一个简单计数窗口，统计每两个整数元素的平均值。

Scala 代码：

Java 代码：

这个例子实现了一个简单计数窗口。我们通过第一个字段输入元组（在本例中，所有的 `key` 值都是

1)。一旦计数达到 2，它就会发出平均值并清除状态，这样我们就可以从 0 开始。注意，如果我们的元组在第一个字段中有不同的值，那么这将为每个不同的输入 key 保留不同的状态值。

State Time-To-Live (TTL)

对于任何类型 Keyed State 都可以设定状态的生命周期 (TTL, Time-To-Live)，即状态的存活时间，以此来配置 Keyed State 的有效期，以确保能够在规定时间内及时地清理状态数据。如果配置了状态的 TTL，那么当某一个 key 的状态数据过期时，存储的状态会被清除。如果状态是集合类型，那么 TTL 是单独针对每个元素设置的，也就是说每一个 List 元素、或者是 Map 的 entry 都有独立的 TTL。

为了使用状态 TTL，必须首先构建 StateTtlConfig 配置对象。然后，可以在任何状态描述符中通过传递该配置对象来启用 TTL 功能，这是通过将 StateTtlConfig 配置传入 StateDescriptor 中的 enableTimeToLive 方法来实现的。

Scala 代码：

```
import org.apache.flink.api.common.state.StateTtlConfig
import org.apache.flink.api.common.state.ValueStateDescriptor
import org.apache.flink.api.common.time.Time

val ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1)) // time-to-live value, 必须
    .setUpdateType(StateTtlConfig.UpdateType.OnCreateAndWrite) // or OnReadAndWrite
    .setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired) // 未清理的过期值也不返回
    .build

val stateDescriptor = new ValueStateDescriptor[String]("text state", classOf[String])
stateDescriptor.enableTimeToLive(ttlConfig)
```

Java 代码：

```
import org.apache.flink.api.common.state.StateTtlConfig;
import org.apache.flink.api.common.state.ValueStateDescriptor;
import org.apache.flink.api.common.time.Time;

StateTtlConfig ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))
    .setUpdateType(StateTtlConfig.UpdateType.OnCreateAndWrite)
    .setStateVisibility(StateTtlConfig.StateVisibility.NeverReturnExpired)
    .build();

ValueStateDescriptor<String> stateDescriptor = new ValueStateDescriptor<>("text state", String.class);
stateDescriptor.enableTimeToLive(ttlConfig);
```

简单来说就是在创建状态描述符时，添加 StateTtlConfig 配置。

在创建 TTL 配置时，newBuilder 方法的第一个参数是强制性的，它是生存时间值。

可以进行以下配置，指定当状态 TTL 刷新时的更新类型：

- ☐ StateTtlConfig.UpdateType.OnCreateAndWrite: 只在 key 的状态创建和写入时更新 TTL(默认)。
 - ☐ StateTtlConfig.UpdateType.OnReadAndWrite: 读取和写入状态时可以更新 TTL。
- 状态可见性配置是否在读取时返回过期值，如果它还没有被清除（默认为 NeverReturnExpired）：
- ☐ StateTtlConfig.StateVisibility.NeverReturnExpired - 过期的值永远不会返回

❑ `StateTtlConfig.StateVisibility.ReturnExpiredIfNotCleanedUp` - 如果还可用, 返回

在 `NeverReturnExpired` 的情况下, 过期状态的行为就像它不再存在一样, 即使它仍然必须被删除。该选项对于数据在 TTL 之后无法进行严格的读访问的情况非常有用, 例如使用隐私敏感数据的应用程序。另一个选项 `ReturnExpiredIfNotCleanedUp` 允许在清理之前返回过期状态。

注: 目前只支持与处理时间相关的 TTL。

清除过期状态

默认情况下, 在读取时显式地删除过期值(如 `ValueState#value`), 如果配置的状态后端支持, 则在后台定期回收过期值。后台清理可以在 `StateTtlConfig` 中禁用:

Scala 代码:

```
import org.apache.flink.api.common.state.StateTtlConfig

val ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))
    .disableCleanupInBackground
    .build
```

Java 代码:

```
import org.apache.flink.api.common.state.StateTtlConfig;

StateTtlConfig ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))
    .disableCleanupInBackground()
    .build();
```

此外, 可以在获取完整状态快照时激活清理, 这将减少快照的大小。在当前实现下, 本地状态不会被清除, 但当从上一个快照恢复时, 它将不包括已删除的过期状态。可以在 `StateTtlConfig` 中配置。

Scala 代码:

```
import org.apache.flink.api.common.state.StateTtlConfig
import org.apache.flink.api.common.time.Time

val ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))
    .cleanupFullSnapshot
    .build
```

Java 代码:

```
import org.apache.flink.api.common.state.StateTtlConfig;
import org.apache.flink.api.common.time.Time;

StateTtlConfig ttlConfig = StateTtlConfig
    .newBuilder(Time.seconds(1))
    .cleanupFullSnapshot()
    .build();
```

注: 这个选项不适用于 RocksDB 状态后端的增量检查点。

在 Scala DataStream API 中的状态

除了上面描述的接口，对于只有一个 `ValueState` 的 `KeyedStream`，Scala API 还提供了有状态 `map()` 或 `flatMap()` 函数的简洁方式。用户函数获取 `Option` 中的 `ValueState` 的当前值，并且必须返回一个更新后的值，该值将用于更新状态。

```
val stream: DataStream[(String, Int)] = ...

val counts: DataStream[(String, Int)] = stream
  .keyBy(_._1)
  .mapWithState((in: (String, Int), count: Option[Int]) =>
    count match {
      case Some(c) => (in._1, c), Some(c + in._2)
      case None => (in._1, 0), Some(in._2)
    })
```

3.9.4 使用托管 Operator State

要使用托管运算符状态，状态函数可以实现更一般的 `CheckpointedFunction` 接口，也可以实现 `ListCheckpointed<T extends Serializable>` 接口。

CheckpointedFunction

`CheckpointedFunction` 接口通过不同的重分布方案提供对 `non-keyed state` 的访问。它需要实现两种方法：

```
void snapshotState(FunctionSnapshotContext context) throws Exception;

void initializeState(FunctionInitializationContext context) throws Exception;
```

无论何时执行检查点，都会调用 `snapshotState()`。对应的 `initializeState()` 在每次初始化用户定义的函数时都会被调用，无论是在函数第一次初始化时，还是在函数从之前的检查点实际恢复时。因此，`initializeState()` 不仅是初始化不同类型状态的地方，也是包含状态恢复逻辑的地方。

目前，支持列表样式的托管操作符状态。状态应该是一个可序列化对象的 `List` 列表，它们彼此独立，因此可以在重新调节（大小）时进行重新分布。换句话说，这些对象是可以重新分布 `non-keyed state` 的最细粒度。根据状态访问方法，定义了以下重分布方案：

- ❑ **Even-split redistribution:** 每个操作符返回一个状态元素列表。整个状态在逻辑上是所有列表的连接。在恢复/再分配中，只要有并行操作符，列表就被均匀地划分为尽可能多的子列表。每个操作符都有一个子列表，它可以是空的，也可以包含一个或多个元素。例如，如果在 `parallelism 1` 中，操作符的 `checkpoint` 状态包含元素 `element1` 和 `element2`，当将并行度增加到 2 时，`element1` 可能会出现在操作符实例 0 中，而 `element2` 将出现在操作符实例 1 中。
- ❑ **Union redistribution:** 每个操作符返回一个状态元素列表。整个状态在逻辑上是所有列表的连接。在恢复/重新分配时，每个操作符都获得状态元素的完整列表。

下面是一个有状态 `SinkFunction` 的例子，它在将元素发送到外部系统之前使用 `CheckpointedFunction` 来缓冲元素。它演示了基本的 `even-split redistribution` 列表状态：

Scala 代码：

Java 代码:

在上面的代码中, `initializeState` 方法以 `FunctionInitializationContext` 作为参数。这用于初始化 non-keyed state “容器”。这些是 `ListState` 类型的容器, non-keyed state 对象将在检查点时存储在其中。

请注意状态是如何被初始化的, 类似于 keyed state, 使用一个 `StateDescriptor` 状态描述符, 其中包含状态名称和关于状态所持有的值的类型的信息:

Scala 代码:

```
val descriptor = new ListStateDescriptor[(String, Long)](  
    "buffered-elements",  
    TypeInformation.of(new TypeHint[(String, Long)]() {})  
)  
  
checkpointedState = context.getOperatorStateStore().getListState(descriptor)
```

Java 代码:

```
ListStateDescriptor<Tuple2<String, Integer>> descriptor =  
    new ListStateDescriptor<>(  
        "buffered-elements",  
        TypeInformation.of(new TypeHint<Tuple2<Long, Long>>() {}));  
  
checkpointedState = context.getOperatorStateStore().getListState(descriptor);
```

在初始化容器之后, 我们使用上下文的 `isRestore()` 方法来检查在发生故障后是否正在恢复。如果这是真的, 即我们正在恢复, 则应用恢复逻辑。

如修改后的 `BufferingSink` 代码所示, 在状态初始化期间恢复的这个 `ListState` 保存在一个类变量中, 供 `snapshotState()` 将来使用。在那里, `ListState` 被清除了前一个检查点包含的所有对象, 然后填充我们希望检查点的新对象。

作为补充说明, keyed state 也可以在 `initializeState()` 方法中初始化。这可以使用提供的 `FunctionInitializationContext` 来完成。

ListCheckpointed 接口

`ListCheckpointed` 接口是 `CheckpointedFunction` 的一个更受限的变体, 它只支持恢复时带有 even-split redistribution 方案的列表样式状态。它还需要实现两种方法:

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

在 `snapshotState()` 中, 操作符应该向检查点返回一个对象列表, 而 `restoreState` 必须在恢复时处理这样的列表。如果状态不可重分区, 则始终可以在 `snapshotState()` 中返回一个 `Collections.singletonList(MY_STATE)`。

3.9.5 广播状态 (Broadcast State)

广播状态 (Broadcast State) 是 Operator State 的一种特殊类型。广播状态被引入以支持这样的应用场景: 一个流的记录需要广播给所有下游任务, 它们用于在所有子任务之间保持相同的状态, 然后可以在处理第二个流的记录时访问此状态。举个例子, 广播状态可以自然地出现, 我们可以想象一个低吞吐

量的流包含一组规则，我们想要对来自另一个流的所有元素进行评估。考虑到上述类型的应用场景，广播状态与其他操作符状态的区别在于：

- ❑ 它有一个 map 格式
- ❑ 它只对以 broadcasted 流和 non-broadcasted 流作为输入的特定操作符可用，并且
- ❑ 这样的操作符可以有多个名称不同的广播状态

【示例】我们有一个具有不同颜色和形状的对象流，并且我们希望找到遵循特定模式（例如，一个矩形后面跟着一个三角形）的相同颜色的对象对。我们假设一组有趣的模式会随着时间的推移而发展。

在本例中，第一个流将包含带有 Color 和 Shape 属性的 Item 类型的元素。另一个流将包含 Rules。

从 Items 流开始，我们只需要将 Color 定为 key，因为我们想要相同颜色的对。这将确保相同颜色的元素最终出现在相同的物理机上。

```
// 将 Color 设为 key
```

```
KeyedStream<Item, Color> colorPartitionedStream = itemStream.keyBy(new KeySelector<Item, Color>(){...});
```

继续讨论 Rules，包含它们的流应该被广播到所有下游任务，这些下游任务应该将它们存储在本地，以便它们可以根据所有传入的 Items 对它们进行评估。下面的代码片段将 i) 广播规则流，ii) 使用提供的 MapStateDescriptor，它将创建存储规则的广播状态。

```
// 用于存储规则(字符串)名称和规则本身的映射描述符
```

```
MapStateDescriptor<String, Rule> ruleStateDescriptor = new MapStateDescriptor<>(  
    "RulesBroadcastState",  
    BasicTypeInfo.STRING_TYPE_INFO,  
    TypeInfo.of(new TypeHint<Rule>() {}));
```

```
// 广播规则并创建广播状态
```

```
BroadcastStream<Rule> ruleBroadcastStream = ruleStream.broadcast(ruleStateDescriptor);
```

最后，为了针对来自 Item 流的输入元素评估 Rules，我们需要连接两个流，并指定我们的匹配检测逻辑。

可以通过调用非广播流上的 connect() 来连接流(keyed 或 non-keyed)与 BroadcastStream，该 BroadcastStream 作为参数。这将返回一个 BroadcastConnectedStream，我们可以在其上调用 process()，使用特殊类型的 CoProcessFunction。该函数将包含我们的匹配逻辑。函数的确切类型取决于非广播流的类型：

- ❑ 如果它是 keyed 的，那么这个函数就是一个 KeyedBroadcastProcessFunction。
- ❑ 如果它是 non-keyed 的，则该函数是 BroadcastProcessFunction。

鉴于我们的非广播流是 keyed，以下片段包括上述调用：

```
DataStream<String> output = colorPartitionedStream  
    .connect(ruleBroadcastStream)  
    .process(  
        // 我们的 KeyedBroadcastProcessFunction 中的类型参数表示:  
        // 1. keyed stream 的 key  
        // 2. 非广播端元素的类型  
        // 3. 广播端元素的类型  
        // 4. 结果的类型，这里是一个字符串  
        new KeyedBroadcastProcessFunction<Color, Item, Rule, String>(){  
            // 匹配逻辑代码  
        }  
    );
```

注意：连接应该在非广播流上调用，以 `BroadcastStream` 作为参数。

`BroadcastProcessFunction` 和 `KeyedBroadcastProcessFunction`

对于 `CoProcessFunction`，这些函数有两个要实现的处理方法：

- ❑ `processBroadcastElement()`：负责处理广播流中的传入元素
- ❑ `processElement()`：用于非广播流

方法的完整签名如下：

```
public abstract class BroadcastProcessFunction<IN1, IN2, OUT> extends BaseBroadcastProcessFunction {

    public abstract void processElement(IN1 value, ReadOnlyContext ctx, Collector<OUT> out)
        throws Exception;

    public abstract void processBroadcastElement(IN2 value, Context ctx, Collector<OUT> out)
        throws Exception;
}

public abstract class KeyedBroadcastProcessFunction<KS, IN1, IN2, OUT> {

    public abstract void processElement(IN1 value, ReadOnlyContext ctx, Collector<OUT> out)
        throws Exception;

    public abstract void processBroadcastElement(IN2 value, Context ctx, Collector<OUT> out)
        throws Exception;

    public void onTimer(long timestamp, OnTimerContext ctx, Collector<OUT> out) throws Exception;
}
```

这两种方法的不同在于它们所提供的上下文环境。非广播方有一个 `ReadOnlyContext`，而广播方有一个 `Context`。

这两个上下文(以下列举中的 `ctx`)：

- ❑ 给出对广播状态的访问：`ctx.getBroadcastState(MapStateDescriptor<K, V> stateDescriptor)`
- ❑ 允许查询元素的时间戳：`ctx.timestamp()`,
- ❑ 获取当前水印：`ctx.currentWatermark()`
- ❑ 获取当前的处理时间：`ctx.currentProcessingTime()`, and
- ❑ 发射元素到边输出：`ctx.output(OutputTag<X> outputTag, X value)`.

在 `getBroadcastState()` 中的 `stateDescriptor` 状态描述符应该与上面的 `broadcast(ruleStateDescriptor)` 中的状态描述符是同一个。

不同之处在于每一方对广播状态的访问类型不同。广播方对其具有读写访问权，而非广播方具有只读访问权(即名称)。

最后，由于 `KeyedBroadcastProcessFunction` 是在 `keyed stream` 流上操作的，所以它公开了 `BroadcastProcessFunction` 不可用的一些功能。那就是：

- ❑ 在 `processElement()` 方法中的 `ReadOnlyContext` 允许访问 Flink 的底层计时器服务，该服务允许注册事件时间和/或处理时间计时器。当计时器触发时，`onTimer()` 被调用，带有 `OnTimerContext` 作为参数。`OnTimerContext` 公开与 `ReadOnlyContext` plus 相同的功能，再加上：
 - 能够询问触发的计时器是事件时间还是处理时间，和
 - 查询与计时器关联的 `key`

- ❑ 在 `processBroadcastElement()` 方法中的 `Context` 包含方法 `applyToKeyedState(StateDescriptor<S, VS> stateDescriptor, KeyedStateFunction<KS, S> function)`。这允许注册一个 `KeyedStateFunction`，应用到与所提供的状态描述符相关联的所有 key 的所有状态。

注意：只有在“`KeyedBroadcastProcessFunction`”的“`processElement()`”处才能注册计时器，而且只能在那里注册。这在“`processBroadcastElement()`”方法中是不可能的，因为没有与广播元素相关联的 key。

3.9.6 状态后端

在使用状态时，需要了解 Flink 的状态后端（state backends）。

Flink 提供了不同的状态后端，用于指定状态存储的方式和位置。状态可以位于 Java 的堆上，也可以位于堆外。根据用户的状态后端，Flink 也可以管理应用程序的状态，这意味着 Flink 处理内存管理(如果需要，可能会溢出到磁盘)，以允许应用程序保存非常大的状态。默认情况下，配置文件 `flink-conf.yaml` 确定所有 Flink 作业的状态后端。

可用的状态后端

状态后端有两种实现——一种基于 RocksDB，它是一个嵌入式键/值存储，将其工作状态保存在磁盘上；另一种基于堆的状态后端，将其工作状态保存在 Java 堆的内存中。状态在内部是如何表示的，以及它在检查点上是如何以及在哪里持久化的，都取决于所选择的 state Backend。

Flink 打包了这些状态后端：

- ❑ `HashMapStateBackend`
- ❑ `EmbeddedRocksDBStateBackend`

如果没有其他配置，系统将使用 `HashMapStateBackend`。在下表中列出了这两种状态后端的特点。

`HashMapStateBackend` 将数据作为 Java 堆上的对象在内部保存。键/值状态和窗口操作符持有存储值、触发器等哈希表。当使用基于堆的状态后端保存的状态时，访问和更新涉及读和写堆上的对象。

`HashMapStateBackend` 被鼓励用于：

- ❑ 具有大状态、长窗口、大键/值状态的作业。
- ❑ 所有高可用性模式。

还建议将托管内存设置为零。这将确保为 JVM 上的用户代码分配最大的内存量。

`HashMapStateBackend` 总是执行异步快照，这意味着它们可以在不妨碍正在进行的流处理的情况下拍摄快照。

`EmbeddedRocksDBStateBackend` 在 RocksDB 数据库中保存动态数据，RocksDB 数据库默认存储在 TaskManager 本地数据目录中。与在 `HashMapStateBackend` 中存储 Java 对象不同，数据被存储为序列化的字节数组，主要由类型序列化器定义，导致键比较按字节进行，而不是使用 Java 的 `hashCode()` 和 `equals()` 方法。

`EmbeddedRocksDBStateBackend` 总是执行异步快照，这意味着它们可以在不妨碍正在进行的流处理的情况下拍摄快照。

`EmbeddedRocksDBStateBackend` 的限制：由于 RocksDB 的 JNI 桥接 API 基于 `byte[]`，每个键和每个值支持的最大大小为 2^{31} 字节（2GB）。在 RocksDB 中使用 merge 操作的状态（例如 `ListState`）可以

默默地积累值大小 $> 2^{31}$ 字节，然后在下一次检索时失败。这是 RocksDB JNI 目前的局限性。

EmbeddedRocksDBStateBackend 被鼓励用于：

- ☐ 具有大状态、长窗口、大键/值状态的作业。
- ☐ 所有高可用性模式。

注意，EmbeddedRocksDBStateBackend 可以保留的状态量仅受可用磁盘空间的限制。与将状态保存在内存中的 HashMapStateBackend 相比，这允许保持非常大的状态。但是对于在 EmbeddedRocksDBStateBackend 中保存的对象，访问和更新涉及序列化和反序列化，因此开销要大得多。还需要注意的是，只有 EmbeddedRocksDBStateBackend 能够进行增量快照，这对于具有大量缓慢变化状态的应用程序来说是一个显著的好处。

选择正确的状态后端

在 HashMapStateBackend 和 RocksDB 之间进行选择时，需要在性能和可扩展性之间做出选择。

HashMapStateBackend 非常快，因为每个状态访问和更新操作在 Java 堆上的对象上；但是，状态大小受集群内可用内存的限制。另一方面，RocksDB 可以根据可用磁盘空间进行扩展，并且是唯一支持增量快照的状态后端。然而，每个状态访问和更新都需要(反)序列化和可能从磁盘读取，这导致平均性能比内存状态后端慢一个数量级。

在 Flink 1.13 中，统一了 Flink 保存点（savepoint）的二进制格式。这意味着可以获取一个保存点，然后使用不同的状态后端从它恢复。所有的状态后端仅从版本 1.13 开始产生一种通用格式。

配置状态后端

默认状态后端可以在 flink-conf.yaml 中配置，由配置键 state.backend 指定。配置项的可能值是 hashmap (HashMapStateBackend)，rocksdb (EmbeddedRocksDBStateBackend)，或者实现状态后端工厂类 StateBackendFactory 的全限定类名，例如，用于 EmbeddedRocksDBStateBackend 的 org.apache.flink.contrib.streaming.state.EmbeddedRocksDBStateBackendFactory。默认的状态后端可以被代码中的配置覆盖。

state.checkpoints.dir 选项定义了所有后端写入检查点数据和元数据文件的目录。

配置文件中的示例部分如下所示：

```
# 用于存储操作符状态检查点的后端
state.backend: hashmap

# 存储检查点的目录
state.checkpoints.dir: hdfs://xueai8:8020/flink/checkpoints
```

我们可以通过 StreamExecutionEnvironment 为每个 job 设置状态后端。

Scala 代码：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment()
env.setStateBackend(new HashMapStateBackend())
```

Java 代码：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setStateBackend(new HashMapStateBackend());
```

如果想在 IDE 中使用 EmbeddedRocksDBStateBackend 或者在 Flink 工作中以编程方式配置它，需要将以下依赖到 Flink 项目中：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-statebackend-rocksdb_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
```

由于 RocksDB 是默认的 Flink 发行版的一部分，如果在 job 中没有使用任何 RocksDB 代码，并在 `flink-conf.yaml` 中通过 `state.backend` 配置了状态后端以及进一步的 checkpointing 和 RocksDB 特定的参数，那么就不需要这个依赖。

对于 RocksDB 状态后端，虽然我们鼓励在大状态下使用增量检查点，但需要手动启用这个功能，通过在 `flink-conf.yaml` 文件中配置：

```
state.backend.incremental: true
```

也可以直接在代码中配置它（覆盖 `config` 默认值）：

```
EmbeddedRocksDBStateBackend backend = new EmbeddedRocksDBStateBackend(true);
```

注：一旦启用了增量检查点，在 web UI 中显示的 Checkpointed Data Size 仅代表该检查点的增量检查点数据大小，而不是完整状态大小。

当选择 RocksDB State Backend 时，计时器也默认存储在 RocksDB 中。然而，在 RocksDB 中维护计时器可能会有一定的成本，因此 Flink 提供了将计时器存储在 JVM 堆上的选项，即使 RocksDB 用于存储其他状态。当计时器数量较少时，基于堆的计时器可以有更好的性能。

将配置项 `state.backend.rocksdb.timer-service.factory` 设为 `heap`（而不是默认的 `rocksdb`）将计时器存储在堆上。

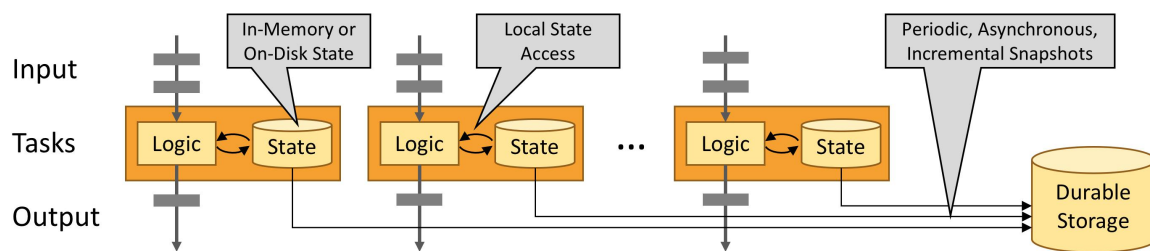
注：RocksDB 状态后端与基于堆的计时器的组合目前不支持计时器状态的异步快照。其他状态(如键态)仍然是异步快照的。

3.9.7 检查点存储

Flink 中的每个函数和操作符都可以是有状态的。有状态函数在处理单个元素/事件时存储数据，使状态成为任何类型更精细操作的关键构建块。

为了使状态容错，Flink 需要对本地生成的状态进行持久化存储，以避免因应用或者节点机器故障等原因导致数据的丢失。Flink 是通过 `checkpoint`（检查点）的方式将状态定期和异步地写入到远程的持久化存储。

Flink 定期在每个操作符中获取所有状态的持久快照，并将这些快照复制到更持久的地方，例如分布式文件系统。在出现故障时，Flink 可以重新启动应用程序，并从最新的检查点加载其状态，从而恢复应用程序的完整状态，并继续处理，就像没有发生任何错误一样，从而确保了在发生故障时的精确一次状态一致性。



存储这些快照的位置是通过作业检查点存储（checkpoint storage）定义的。检查点存储有两种实现，一种是将状态快照持久化到分布式文件系统，另一种使用 JobManager 的堆。

Name	State Backup
FileSystemCheckpointStorage	Distributed file system
• Supports very large state size • Highly durable • Recommended for production deployments	
JobManagerCheckpointStorage	JobManager JVM Heap
• Good for testing and experimentation with small state (locally)	

启用和配置检查点

默认情况下，检查点是禁用的。要启用检查点，在 `StreamExecutionEnvironment` 上调用 `enableCheckpointing(n)`，其中 `n` 是检查点间隔时间（以毫秒为单位）。

检查点的其他参数包括：

- ❑ 检查点存储：可以设置使检查点快照持久的位置。默认情况下，Flink 将使用 JobManager 的堆。对于生产部署，建议使用持久的文件系统。
- ❑ 精确一次性和至少一次性：可以选择将模式传递给 `enableCheckpointing(n)` 方法，以在两个保证级别之间进行选择。对大多数应用程序来说，精确一次性是更好的选择。至少一次可能与某些超低延迟（持续几毫秒）应用程序相关。
- ❑ 检查点超时：进程中的检查点中止的时间，在此之后，如果某个正在进行的检查点尚未完成，则中止该检查点。
- ❑ 检查点之间的最短时间：为了确保流应用程序在检查点之间进行一定的进度，可以定义检查点之间需要多少时间。例如，如果将该值设置为 5000，则下一个检查点将在前一个检查点完成后不超过 5 秒启动，而不考虑检查点持续时间和检查点间隔。注意，这意味着检查点间隔永远不会小于这个参数。注意，这个值还意味着并发检查点的数量为 1。
- ❑ 并发检查点数量：默认情况下，当一个检查点还在进行时，系统不会触发另一个检查点。这可以确保拓扑不会在检查点上花费太多时间，也不会处理流方面取得进展。可以允许多个重叠的检查点，这对有一定处理延迟（例如，因为函数调用外部服务，需要一些时间来回应）的管道很有意义，但是仍然需要频繁的检查点（100 毫秒）来在失败时重新处理很少的数据。当定义了检

检查点之间的最短时间时，不能使用此选项。

- ❑ 外部化的检查点：可以将定期检查点配置为在外部持久化。外部化检查点将其元数据写到持久存储中，并且在作业失败时不会自动清除。这样，如果作业失败了，就会有一个检查点来恢复工作。
- ❑ 检查点错误导致任务失败/继续：这将确定在执行任务的检查点过程中发生错误时任务是否会失败。这是默认的行为。或者，当禁用此选项时，任务将简单地将检查点拒绝给检查点协调器，并继续运行。
- ❑ 选择检查点进行恢复：这决定了一个作业是否会退回到最新的检查点，即使有更多的最近的保存点可用来潜在地减少恢复时间。

下面是启用和配置检查点的代码示例。

Scala 代码：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment()

// 每 1000 毫秒启动一个检查点
env.enableCheckpointing(1000)

// 高级选项:

// 设置模式 exactly-once (这是默认的)
env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE)

// 确保在检查点之间有 500 毫秒的进度
env.getCheckpointConfig.setMinPauseBetweenCheckpoints(500)

// 检查点必须在一分钟内完成，否则将被丢弃
env.getCheckpointConfig.setCheckpointTimeout(60000)

// 防止任务失败如果检查点发生错误，检查点将被拒绝
env.getCheckpointConfig.setFailTasksOnCheckpointingErrors(false)

// 只允许一个检查点在同一时间进行
env.getCheckpointConfig.setMaxConcurrentCheckpoints(1)

// 启用实验性的未对齐检查点
env.getCheckpointConfig().enableUnalignedCheckpoints();

// 设置将写入检查点快照的检查点存储
env.getCheckpointConfig().setCheckpointStorage("hdfs://my/checkpoint/dir")
```

Java 代码：

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

// 每 1000 毫秒启动一个检查点
env.enableCheckpointing(1000);

// 高级选项:

// 将模式设为 exactly-once (这是默认的模式)
```

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

// 确保在检查点之间有 500 毫秒的进度
env.getCheckpointConfig().setMinPauseBetweenCheckpoints(500);

// 检查点必须在一分钟内完成，否则将被丢弃
env.getCheckpointConfig().setCheckpointTimeout(60000);

// 同时只允许一个检查点在运行
env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);

// 启用外部化检查点，外部化检查点在作业取消后会被保留
env.getCheckpointConfig().enableExternalizedCheckpoints(
    ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);

// 启用实验性的未对齐检查点
env.getCheckpointConfig().enableUnalignedCheckpoints();

// 设置将写入检查点快照的检查点存储
env.getCheckpointConfig().setCheckpointStorage("hdfs://my/checkpoint/dir")
可以通过 conf/flink-conf.yaml 设置更多的参数和/或默认值，请参考官网文档。
```

Flink 的检查点机制在计时器和有状态操作符（包括连接器、窗口和任何用户定义的状态）中存储所有状态的一致快照。存储检查点的位置(例如，JobManager 内存、文件系统、数据库)取决于配置的检查点存储。

默认情况下，检查点存储在 JobManager 的内存中。为了正确地持久化大状态，Flink 支持在其他位置检查点状态的各种方法。

可以通过 `StreamExecutionEnvironment.getCheckpointConfig().setCheckpointStorage(...)` 配置选择检查点存储。强烈建议将检查点存储在用于生产部署的高可用文件系统中。

注：Flink 目前仅为没有迭代的作业提供处理保证。在迭代作业上启用检查点会导致异常。为了在迭代程序上强制检查点，用户需要在启用检查点时设置一个特殊的标志：

```
env.enableCheckpointing(interval, CheckpointingMode.EXACTLY_ONCE, force = true);
```

3.9.8 状态快照

Flink 需要知道状态，以便使用检查点和保存点容错。

Flink 能够通过状态快照和流重放的组合提供容错的、精确一次的语义。这些快照捕获分布式管道的整个状态，将偏移量记录到输入队列中，以及在此之前由于摄入数据而导致的整个作业图的状态。当故障发生时，源被重绕（rewind），状态被还原（restore），处理被恢复（resume）。如上所述，这些状态快照是异步捕获的，不会妨碍正在进行的处理。

术语

首先了解几个术语定义。

- ❑ 快照（Snapshot）：一个通用术语，指的是 Flink 作业状态的全局、一致的镜像。快照包含一个

指向每个数据源(例如,一个文件或 Kafka 分区的偏移量)的指针, 以及每个作业的有状态操作符的状态拷贝(这个状态是已经处理了所有数据源中到偏移位置处的事件的结果)。

- ❑ 检查点 (Checkpoint): Flink 为从故障中恢复而自动拍摄的快照。检查点可以是增量的, 并且经过优化可以快速恢复。
- ❑ 外部化的检查点 (Externalized Checkpoint): 通常, 检查点不打算由用户操作。Flink 在作业运行时只保留最近的 n 个检查点(n 是可配置的), 并在作业被取消时删除它们。但是可以将它们配置为保留, 在这种情况下, 可以手动恢复它们。
- ❑ 保存点 (Savepoint): 由用户(或 API 调用)为某些操作目的而手动触发的快照, 例如有状态的重新部署/升级/重新调整等操作。保存点是手动触发的检查点, 它获取程序的快照并将其写入状态后端。为此, 它们依赖于常规检查点机制。

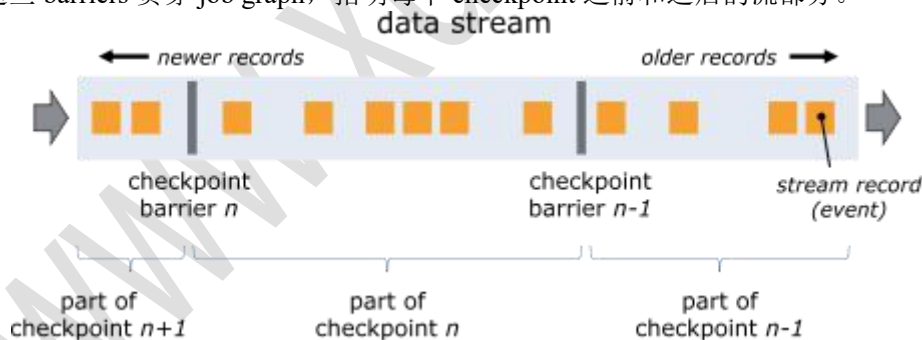
状态快照工作原理

Flink 容错机制的核心部分是绘制分布式数据流和操作符状态的一致快照。这些快照充当一致的检查点, 系统在出现故障时可以回退到这些检查点。

Flink 分布式快照的一个核心元素是 stream barriers。这些 barriers 被注入到数据流中, 并将记录作为数据流的一部分进行流处理。barriers 永远不会超过记录, 它们严格地在一条线上流动。一个 barrier 将数据流中的记录分离为进入当前快照的记录集和进入下一个快照的记录集。每个 barriers 携带着被它把记录推到前面的快照的 ID。barriers 不会中断流的流动, 因此非常轻量。来自不同快照的多个 barriers 可以同时流中, 这意味着各种快照可能同时发生。

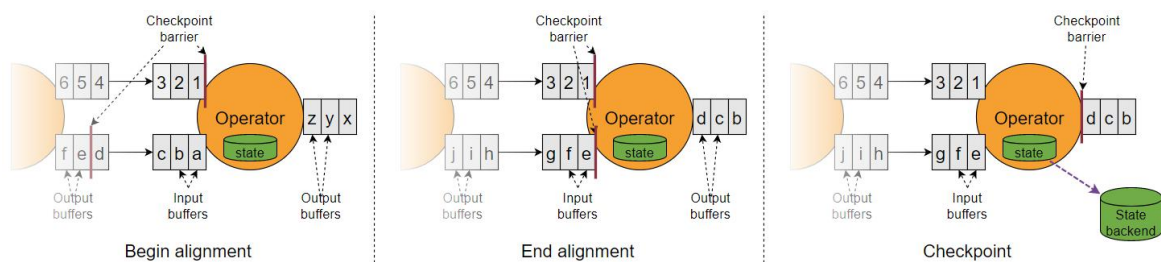
Flink 使用了 Chandy-Lamport 算法的一种变体, 称为 asynchronous barrier snapshotting。

当检查点协调器 (checkpoint coordinator, JobManager 的一部分) 指示一个 TaskManager 开始 checkpoint 时, 这个 TaskManager 让所有源记录它们的偏移量, 并将编号的 checkpoint barriers 插入到它们的流中。这些 barriers 贯穿 job graph, 指明每个 checkpoint 之前和之后的流部分。

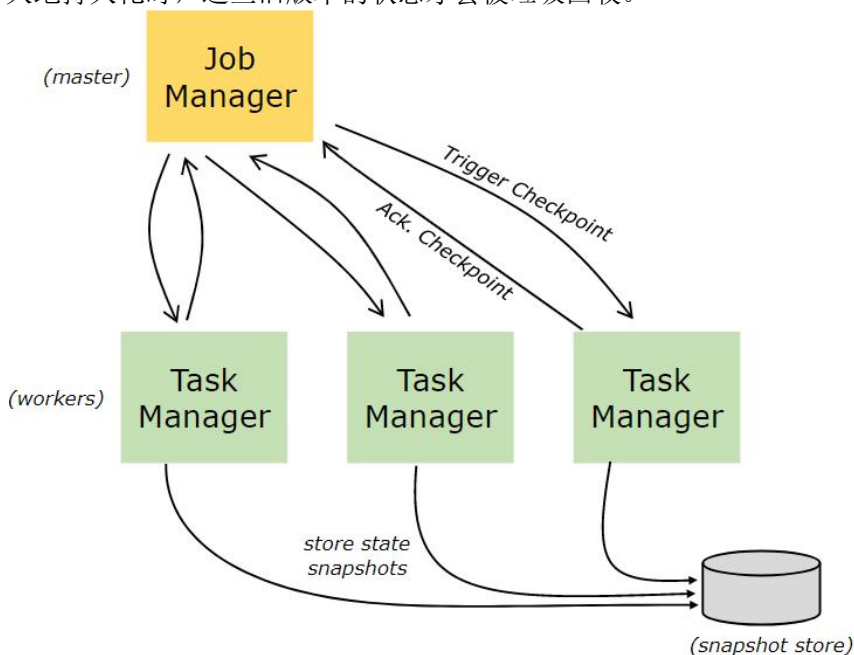


Checkpoint n 将包含每个操作符 (由于在 checkpoint barrier n 之前消费了所有事件, 而在 checkpoint barrier n 之后没有消费任何事件) 的状态。

当 job graph 中的每个操作符 (算子) 接收到其中一个 barrier 时, 它记录其状态 (到状态后端)。具有两个输入流 (如 CoProcessFunction) 的操作符执行 barrier alignment, 以便快照将反映从两个输入流 (但不是超过两个 barrier) 消费事件所产生的状态。



Flink 的状态后端使用 **copy-on-write** 机制，允许在异步快照旧版本状态时不受阻碍地继续流处理。只有当快照被持久地持久化时，这些旧版本的状态才会被垃圾回收。



完全一次保证

当流处理应用程序出现问题时，可能会丢失或重复结果。检查点模式定义了出现故障时系统所保证的一致性。当检查点被激活时，数据流将被重放，从而重复处理过程中丢失的部分。Flink 支持以下两种检查点模式：

- ❑ 不丢失数据，但可能会有重复的结果（至少一次，`CheckpointingMode.AT_LEAST_ONCE`）
- ❑ 既不会丢失数据，也不会有重复的结果（完全一次性，`CheckpointingMode.EXACTLY_ONCE`）

注意，这里所说的精确一次性，并不是说每个事件恰好每处理一次，而是意味着每个事件将只影响一次 Flink 所管理的状态。

只有提供完全一次保证才需要 **barrier alignment**。如果不需要完全一次保证，则可以通过配置 Flink 使用 `CheckpointingMode.AT_LEAST_ONCE` 来获得一些性能，这时会禁用 **barrier alignment**。

// 设置模式

```
env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.AT_LEAST_ONCE)
```

完全一次端到端

为了实现端到端的精确一次，以便来自源的每个事件都精确地影响 sink 一次，以下条件必须成立：

- ❑ 源必须是可重放的（**replayable**）
- ❑ sink 必须是事务性的（必须是事务性的(或幂等的)）

3.10 侧输出流

除了 `DataStream` 操作产生的主流之外，还可以生成任意数量的附加侧输出（side output）结果流。

3.10.1 什么是侧输出流

很多场景下都需要从一个 Flink 操作符输出一个以上的输出流，比如：

- ☐ 异常
- ☐ 格式不正确的事件
- ☐ 延迟事件
- ☐ 操作警报，例如到外部服务的超时连接

除了错误报告之外，侧输出也是实现流的 N-路拆分的好方法。

每个侧输出通道都与一个 `OutputTag<T>` 关联。标记具有与侧输出的 `DataStream` 类型对应的泛型类型，并且具有名称。具有相同名称的两个 `OutputTag` 应该具有相同的类型，并将引用相同的侧输出。结果流中的数据类型不必与主流中的数据类型匹配，不同端输出的类型也可以不同。当希望分割一个数据流时，这个操作非常有用。

当使用侧输出时，首先需要定义一个 `OutputTag`，用于标识侧输出流：

Java 代码：

```
// 这需要一个匿名的内部类，以便我们可以分析类型
OutputTag<String> outputTag = new OutputTag<String>("side-output") {};
```

Scala 代码：

```
val outputTag = OutputTag[String]("side-output")
```

请注意 `OutputTag` 是如何根据侧输出流包含的元素类型进行类型化的。

可以通过以下函数将数据发送到侧输出：

- ☐ `ProcessFunction`
- ☐ `KeyedProcessFunction`
- ☐ `CoProcessFunction`
- ☐ `ProcessWindowFunction`
- ☐ `ProcessAllWindowFunction`

可以使用上面函数中向用户公开的 `Context` 参数，将数据发送到 `OutputTag` 标识的侧输出。下面是一个从 `ProcessFunction` 发出侧输出数据的例子：

Scala 代码：

```
val input: DataStream[Int] = ...
val outputTag = OutputTag[String]("side-output")

val mainDataStream = input
    .process(new ProcessFunction[Int, Int] {
        override def processElement(
            value: Int,
            ctx: ProcessFunction[Int, Int]#Context,
            out: Collector[Int]): Unit = {
            // 将数据发送到常规输出
            out.collect(value)
        }
    })
```

```
// 将数据发送到侧输出
ctx.output(outputTag, "sideout-" + String.valueOf(value))
}
})
```

Java 代码:

```
DataStream<Integer> input = ...;

final OutputTag<String> outputTag = new OutputTag<String>("side-output");

SingleOutputStreamOperator<Integer> mainDataStream = input
    .process(new ProcessFunction<Integer, Integer>() {

        @Override
        public void processElement(
            Integer value,
            Context ctx,
            Collector<Integer> out) throws Exception {
            // 将数据发送到常规输出
            out.collect(value);

            // 将数据发送到侧输出
            ctx.output(outputTag, "sideout-" + String.valueOf(value));
        }
    });
```

要获得侧输出流，可以对 `DataStream` 操作的结果使用 `getSideOutput(OutputTag)`。这将得到一个 `DataStream`，这是输入到侧输出流的结果。

Scala 代码:

```
val outputTag = OutputTag[String]("side-output")

val mainDataStream = ...

val sideOutputStream: DataStream[String] = mainDataStream.getSideOutput(outputTag)
```

Java 代码:

```
final OutputTag<String> outputTag = new OutputTag<String>("side-output");

SingleOutputStreamOperator<Integer> mainDataStream = ...;

DataStream<String> sideOutputStream = mainDataStream.getSideOutput(outputTag);
```

3.10.2 侧输出流应用示例

下面我们通过两个示例来理解侧输出流的应用方法。

【示例】下面这个示例中，我们扩展了经典的单词计数程序，使其只计算至少 5 个字母长的单词，并将较短的单词发送到一个副输出。

Scala 代码:

Java 代码:

```
package com.xueai8.java.ch03;

import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.ProcessFunction;
import org.apache.flink.util.Collector;
import org.apache.flink.util.OutputTag;

/**
 * 侧输出示例：只计算至少 5 个字母长的单词，并将较短的单词发送到一个侧输出
 */
public class SideOutputDemo2 {
    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 定义一个侧输出的标签
        final OutputTag<String> shortWordsTag = new OutputTag<String>("short") {};

        // 分割数据流，返回主输出流
        SingleOutputStreamOperator<Tuple2<String, Integer>> tokenized = env
            .fromElements("Born as the bright summer flowers",
                "Died as the quiet beauty of autumn leaves")
            .process(new ProcessFunction<String, Tuple2<String, Integer>>() {
                @Override
                public void processElement(String value,
                    Context ctx,
                    Collector<Tuple2<String, Integer>> out) throws Exception {
                    // 规范化并分割每一行
                    String[] tokens = value.toLowerCase().split("\\W+");

                    for (String token : tokens) {
                        if (token.length() < 5) {
                            // 发送短的词到一个副输出
                            ctx.output(shortWordsTag, token);
                        } else if (token.length() > 0) {
                            // 发出 key-value 对
                            out.collect(new Tuple2<>(token, 1));
                        }
                    }
                }
            });

        // 输出副输出流（长度小于 5 的词）
        DataStream<String> shortWords = tokenized.getSideOutput(shortWordsTag);
        shortWords.print("short word");
    }
}
```

```
// 对主输出流执行单词计数运算，并输出结果
DataStream<Tuple2<String, Integer>> wordCounts = tokenized
    .keyBy(t -> t.f0)
    .sum(1);
wordCounts.print("long word");

// 执行
env.execute("flink Side Output");
}
```

在上面的代码中，使用传递给 `processElement` 方法的 `Context` 来写入边输出，使用 `OutputTag` 指定要写入哪个边输出。在本例中，短单词被发送到副输出进行收集，而剩余的单词作为元组(典型的单词计数样式)随主收集器一起发送。

执行上面的代码，输出结果如下：

```
short word:8> born
short word:1> died
short word:1> as
short word:1> the
short word:1> of
short word:8> as
short word:8> the
long word:1> (beauty,1)
long word:8> (autumn,1)
long word:6> (quiet,1)
long word:3> (leaves,1)
long word:5> (summer,1)
long word:7> (bright,1)
long word:5> (flowers,1)
```

【示例】下面是一个从 `ProcessFunction` 发出侧输出数据的例子，将数据集中的负数挑出来，输出到侧输出中：

Scala 代码：

```
package com.xueai8.scala.ch03

import org.apache.flink.streaming.api.functions.ProcessFunction
import org.apache.flink.streaming.api.scala._
import org.apache.flink.util.Collector

/**
 * 侧输出示例：将数据集中的负数挑出来，输出到侧输出中
 */
object SideOutputDemo {

  def main(args: Array[String]): Unit = {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment
```

```
// 侧输出
val numbers = List(1, 2, -3, 4, 5, -6, 7, 8, -9, 10)
val ds = env.fromElements(numbers: _*)

// 定义用来标记侧输出的标签, 注意泛型参数是侧输出数据的类型
val outputTag = new OutputTag[Int]("side-output") {}

// 分流处理
val mainDataStream = ds.process(new ProcessFunction[Int, Int] {
  override def processElement(value: Int,
                                ctx: ProcessFunction[Int, Int]#Context,
                                out: Collector[Int]): Unit = {
    if (value > 0) out.collect(value)      // 将数据发送到常规输出
    else ctx.output(outputTag, value)     // 向侧输出发送负数
  }
})

// 获取侧输出结果
val sideOutputStream = mainDataStream.getSideOutput[Int](outputTag)

// 打印主数据流
mainDataStream.print("主数据流")

// 打印侧输出流
sideOutputStream.print("侧输出流")

// 执行
env.execute("flink transformation")
}
```

Java 代码:

输出结果如下所示:

侧输出流:8> -3

侧输出流:3> -6

侧输出流:6> -9

3.11 流连接器

从之前学习的内容中, 我们了解到 Flink 内置了一些基本的数据源和接收器。预定义的数据源包括从文件、目录和 Socket 套接字中读取数据, 以及从集合和迭代器中获取数据。预定义的数据接收器支持向文件、stdout 和 stderr 以及 Socket 套接字写入数据。

除了上述内置 API 外, Flink 中还支持(捆绑)了系列的 Connectors 连接器。连接器提供了用于与各种第三方系统交互的代码, 用于将计算结果输入到常用的存储系统或者消息中间件中。Flink 目前捆绑了如下这些连接器:

- ❑ Apache Kafka (source/sink)

- ☐ Apache Cassandra (sink)
- ☐ Amazon Kinesis Streams (source/sink)
- ☐ Elasticsearch (sink)
- ☐ FileSystem (Hadoop included) - Streaming only (sink)
- ☐ FileSystem (Hadoop included) - Streaming and Batch (sink)
- ☐ RabbitMQ (source/sink)
- ☐ Apache NiFi (source/sink)
- ☐ Twitter Streaming API (source)
- ☐ Google PubSub (source/sink)
- ☐ JDBC (sink)

注意，虽然这里列出的流连接器是 Flink 项目的一部分，并且包含在源版本中，但是它们没有包含在二进制发行版中，因此需要单独下载，或者在 Maven 项目中添加相应的依赖。

Flink 提供了 addSink 方法用来调用自定义的 Sink 或者第三方的连接器。

3.11.1 Kafka 连接器

Kafka 是一个发布-订阅、分布式、消息队列系统，允许用户将消息发布到某个主题，然后将其分发给主题的订阅者。（关于 Kafka 的相关概念，超出了本书的范围，想要了解的话，请参考如下的网络教程 <http://www.xueai8.com/kafka>）

Flink 为读写 Kafka 主题提供了专门的 Kafka 连接器，支持将 Kafka 做为 Flink 流应用程序的数据源和 Data Sink。Flink Kafka 消费者集成了 Flink 的检查点机制来提供精确一次的处理语义。为了实现这一点，Flink 并不完全依赖 Kafka 的消费者组的偏移跟踪，而是在内部跟踪和检查这些偏移。

下图显示了 Flink Kafka 连接器的工作原理：

因为 Flink Kafka 流连接器目前不是二进制发行版的一部分，所以我们需要使用一个特定的 JAR 文件。通常在项目中我们使用以下 Maven 依赖项来加载和管理连接器 JAR 包。在 Maven 项目的 pom.xml 文件中添加以下依赖项以导入指定版本的 Kafka 连接器：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka_2.12</artifactId>
  <version>1.13.2</version>
  <scope>compile</scope>
</dependency>
```

注：用模板生成 Maven 项目的话，已经自带的 Kafka 连接器的配置，只是被注释掉了。取消注释就可以了。

如果使用的是 Kafka source，那么 flink-connector-base 也需要作为依赖项：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-base</artifactId>
  <version>1.13.2</version>
</dependency>
```

Kafka 源

Kafka 源提供了一个构建类来构造 `KafkaSource` 的实例。下面的代码片段展示了如何构建一个 `KafkaSource` 来消费主题“input-topic”的最早偏移量的消息，带有消费组“my-group”，并且只将 message 的值反序列化为字符串。

```
KafkaSource<String> source = KafkaSource.<String>builder()
    .setBootstrapServers(brokers)
    .setTopics("input-topic")
    .setGroupId("my-group")
    .setStartingOffsets(OffsetsInitializer.earliest())
    .setValueOnlyDeserializer(new SimpleStringSchema())
    .build();

env.fromSource(source, WatermarkStrategy.noWatermarks(), "Kafka Source");
```

以下属性是构建 `KafkaSource` 所必需的：

- ☐ Bootstrap 服务器，由 `setBootstrapServers(String)` 配置；
- ☐ 用户组 ID，由 `setGroupId(String)` 配置；
- ☐ 要订阅的主题/分区；
- ☐ 解析 Kafka 消息的反序列化器。

Kafka 源提供了 3 种主题分区订阅方式：

- ☐ 主题列表，从主题列表中的所有分区订阅消息。例如：
`KafkaSource.builder().setTopics("topic-a", "topic-b")`
- ☐ 主题模式，从名称与提供的正则表达式匹配的所有主题订阅消息。例如：
`KafkaSource.builder().setTopicPattern("topic.*")`
- ☐ 分区集，订阅提供的分区集中的分区。例如：

```
final HashSet<TopicPartition> partitionSet = new HashSet<>(Arrays.asList(
    new TopicPartition("topic-a", 0),    // "topic-a"的分区 0
    new TopicPartition("topic-b", 5)));   // "topic-b"的分区 5
KafkaSource.builder().setPartitions(partitionSet)
```

解析 Kafka 消息需要一个反序列化器。

反序列化器(反序列化模式)可以通过 `setDeserializer(KafkaRecordDeserializationSchema)` 配置，其中 `KafkaRecordDeserializationSchema` 定义了如何反序列化一个 Kafka 的 `ConsumerRecord`。如果只需要 Kafka `ConsumerRecord` 的值，那么可以在构建器中使用 `setValueOnlyDeserializer(DeserializationSchema)`，其中 `DeserializationSchema` 定义了如何反序列化 Kafka 消息值的二进制文件。

也可以使用 `KafkaDeserializer` 来反序列化 Kafka 消息值，例如，使用 `StringDeserializer` 将 Kafka 消息值反序列化为 `String` 字符串：

```
import org.apache.kafka.common.serialization.StringDeserializer;

KafkaSource.<String>builder()
    .setDeserializer(KafkaRecordDeserializationSchema.valueOnly(StringSerializer.class));
```

Kafka 源可以通过指定 `OffsetsInitializer` 来消费从不同偏移量开始的消息。内置的初始值设定项包括：

```
KafkaSource.builder()
    // 从消费组的提交偏移量开始，没有重置策略
    .setStartingOffsets(OffsetsInitializer.committedOffsets())
```

```
// 从提交的偏移量开始, 如果提交的偏移量不存在, 也使用 EARLIEST 作为重置策略
.setStartingOffsets(OffsetsInitializer.committedOffsets(OffsetResetStrategy.EARLIEST))
// 从第一个时间戳大于或等于指定时间戳的记录开始
.setStartingOffsets(OffsetsInitializer.timestamp(1592323200L))
// 从最早偏移点开始
.setStartingOffsets(OffsetsInitializer.earliest())
// 从最近的偏移量开始
.setStartingOffsets(OffsetsInitializer.latest())
```

如果上面的内置初始化器不能满足需求, 还可以实现自定义偏移量初始化器。

如果未指定偏移初始化器, 则默认使用 `OffsetsInitializer.earliest()`。

默认情况下, 记录将使用 `Kafka ConsumerRecord` 中嵌入的时间戳作为事件时间。可以定义自己的水印策略来从记录本身提取事件时间, 并向下游发送水印。例如:

```
env.fromSource(kafkaSource, new CustomWatermarkStrategy(), "带有自定义水印策略的 Kafka 源")
```

【示例】编写一个 Flink 流应用程序, 消费 Kafka 中指定主题 `words` 的数据。使用 Kafka 自带的生产者脚本向 `words` 主题写数据。

1、在 IntelliJ IDEA 中创建一个 Flink 项目, 使用 `flink-quickstart-java` 项目模板。(Flink 项目创建过程, 请参见 2.2 节)

2、设置依赖。在 `pom.xml` 中添加如下依赖:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-java</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka_2.12</artifactId>
  <version>1.13.2</version>
  <scope>compile</scope>
</dependency>
```

3、创建流应用程序主类。代码如下:

Scala 代码:

Java 代码:

4、执行程序。请按以下步骤执行:

5、可以得到如下的输出结果:

```
8> good
8> good
```



```
8> study
8> day
8> day
8> up
```

Kafka 生产者

Flink 的 Kafka 生产者被称为 `FlinkKafkaProducer`，通过它可以记录流写入一个或多个 Kafka 主题。

`FlinkKafkaProducer` 的构造函数接受以下参数：

- ❑ 一个默认的输出主题，事件应该在其中写入；
- ❑ `SerializationSchema / KafkaSerializationSchema` 用于将数据序列化到 Kafka 中；
- ❑ Kafka 客户端的属性。以下属性是必需的：
 - “`bootstrap.servers`”（用逗号分隔的 Kafka broker 列表）。
- ❑ 一个容错语义。

Scala 代码：

```
val stream: DataStream[String] = ...

val properties = new Properties
properties.setProperty("bootstrap.servers", "localhost:9092")

val myProducer = new FlinkKafkaProducer[String](
    "my-topic",           // 目标 topic
    new SimpleStringSchema(), // 序列化模式
    properties,           // producer 配置
    FlinkKafkaProducer.Semantic.EXACTLY_ONCE) // 容错性

stream.addSink(myProducer)
```

Java 代码：

```
DataStream<String> stream = ...

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

FlinkKafkaProducer<String> myProducer = new FlinkKafkaProducer<>(
    "my-topic",           // 目标 topic
    new SimpleStringSchema(), // 序列化 schema
    properties,           // producer 配置
    FlinkKafkaProducer.Semantic.EXACTLY_ONCE); // 容错性

stream.addSink(myProducer);
```

【示例】编写一个 Flink 流应用程序，将数据写入到 Kafka 中的指定主题 `words`。使用 Kafka 自带的消费者脚本来查看写入的内容。

请按以下步骤操作。

1、在 IntelliJ IDEA 中创建一个 Flink 项目，使用 `flink-quickstart-java` 项目模板。（Flink 项目创建过程，请参见 2.2 节）

2、设置依赖。在 `pom.xml` 中添加如下依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-java</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_${scala.binary.version}</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka_2.12</artifactId>
  <version>1.13.2</version>
  <scope>compile</scope>
</dependency>
```

3、创建流应用程序主类。代码如下：

Scala 代码：

Java 代码：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer;
import org.apache.flink.streaming.connectors.kafka.KafkaSerializationSchema;
import org.apache.flink.util.Collector;
import org.apache.kafka.clients.producer.ProducerRecord;
import java.nio.charset.StandardCharsets;
import java.util.Properties;

/**
 * 将流数据写入到 Kafka 中的 words 主题
 */
public class KafkaProcuderDemo {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 启用检查点
        env.enableCheckpointing(50000);

        Properties properties = new Properties();
        properties.setProperty("bootstrap.servers", "192.168.190.133:9092");
```

```
// 构造 Flink Kafka Sink, 默认使用 FlinkKafkaProducer.Semantic.AT_LEAST_ONCE 语义
FlinkKafkaProducer<String> myProducer = new FlinkKafkaProducer<String>(
    "words",                                // 目标 topic
    new SimpleStringSchema(),                // 序列化 schema
    properties                              // producer 配置
);

// 将数据写入到 Kafka 的"words"主题
env
    .fromElements("good good study", "day day up")
    .flatMap(new FlatMapFunction<String, String>() {
        @Override
        public void flatMap(String s, Collector<String> out) throws Exception {
            for(String word : s.split("\\W+")){
                out.collect(word);
            }
        }
    })
    .addSink(myProducer);

// 触发流程序执行
env.execute("Kafka Flink Producer Demo");
}
```

默认 FlinkKafkaProducer 使用的是 AT_LEAST_ONCE 语义, 这要求启用检查点。随着 Flink 的检查点启用, Flink Kafka 消费者将消费一个主题的记录, 并定期检查点所有的 Kafka 偏移, 以及其他操作的状态。如果检查点被禁用, Kafka 消费者会定期向 Zookeeper 提交偏移量。

4、执行程序。请按以下步骤执行:

5、在第三个终端窗口 (运行 Kafka 消费者脚本的窗口), 可以看到如下的输出内容:

```
day
day
up
good
good
study
```

在上面的代码中, 我们使用默认的 Semantic.AT_LEAST_ONCE 语义, 这时可以简单地指定序列化模式为 SimpleStringSchema。但是, 如果我们要指定 FlinkKafkaProducer 使用 EXACTLY_ONCE 语义, 那么就需要自定义序列化模式, 它需要实现 KafkaSerializationSchema<T>接口。

关于 SerializationSchema

Flink Kafka Producer 需要知道如何将 Java/Scala 对象转换成二进制数据。KafkaSerializationSchema 允许用户指定这样的模式。对于流中的每条记录, 会调用 ProducerRecord<byte[], byte[]> serialize(T element, @Nullable Long timestamp)方法, 生成一个写入到 Kafka 的 ProducerRecord。

这让用户可以细粒度地控制数据如何写入到 Kafka。通过生产者记录, 可以:

❑ 设置 header 值

- ❑ 为每个记录定义 key
- ❑ 指定自定义数据分区

正则我们重构上面的代码，在构造 `FlinkKafkaProducer` 时，指定使用 `Semantic.EXACTLY_ONCE` 语义。因此，我们需要使用自定义的字符串序列化器 `MyStringSerializationSchema`。重构后的代码如下。

Scala 代码：

Java 代码：

```
package com.xueai8.java.ch03;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer;
import org.apache.flink.streaming.connectors.kafka.KafkaSerializationSchema;
import org.apache.flink.util.Collector;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.nio.charset.StandardCharsets;
import java.util.Properties;

/**
 * 将流数据写入到 Kafka 中的 words 主题
 */
public class KafkaProcuderDemo2 {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // 启用检查点
        env.enableCheckpointing(50000);

        Properties properties = new Properties();
        properties.setProperty("bootstrap.servers", "192.168.190.133:9092");
        properties.setProperty("transaction.timeout.ms",String.valueOf(5*60*1000));

        // 构造 Flink Kafka Sink，默认使用 FlinkKafkaProducer.Semantic.AT_LEAST_ONCE 语义
        FlinkKafkaProducer<String> myProducer = new FlinkKafkaProducer<String>(
            "words", // 目标 topic
            new MyStringSerializationSchema("words"), // 序列化 schema
            properties, // producer 配置
            FlinkKafkaProducer.Semantic.EXACTLY_ONCE // 容错性
        );

        // 将数据写入到 Kafka 的"words"主题
        env
            .fromElements("good good study","day day up")
            .flatMap(new FlatMapFunction<String, String>() {
                @Override
```

```

        public void flatMap(String s, Collector<String> out) throws Exception {
            for(String word : s.split("\\W+")){
                out.collect(word);
            }
        }
    })
    .addSink(myProducer);

    // 触发流程序执行
    env.execute("Kafka Flink Producer Demo");
}

// 自定义字符串序列化器
public static class MyStringSerializationSchema implements KafkaSerializationSchema<String> {
    .....
}
}

```

请注意上面代码中的粗体字部分。Kafka brokers 默认的最大事务超时(transaction.max.timeout.ms)为 15 分钟，生产者设置事务超时不允许大于这个值。但是在默认的情况下，FlinkKafkaProducer 设置事务超时属性(transaction.timeout.ms)为 1 hour，超过了 Kafka brokers 默认的最大事务超时 15 分钟。因此，当使用 Semantic.EXACTLY_ONCE 语义时，会遇到包含下面类似信息的异常：

```

.....
org.apache.kafka.common.KafkaException: org.apache.kafka.common.KafkaException: Unexpected error in
InitProducerIdResponse; The transaction timeout is larger than the maximum value allowed by the broker (as
configured by transaction.max.timeout.ms).
.....

```

因此，需要修改 FlinkKafkaProducer 的事务超时属性 transaction.timeout.ms 来解决这个异常，如代码中所示，我们这里将 transaction.timeout.ms 从 1 hour 降为 5 minutes：

```
properties.setProperty("transaction.timeout.ms",String.valueOf(5*60*1000));
```

如果我们写入 Kafka 的不是字符串而是 Java 对象，则需要自定义一个 SerializatonSchema 序列化器，该序列化需要实现接口 KafkaSerializationSchema。例如，在下面的代码中，我们定义了一个将 Book 对象序列化到 Kafka 中的自定义序列化对象。

【示例】编写一个 Flink 流应用程序，将流数据（Book 对象类型）写入到 Kafka 中的指定主题 books。使用 Kafka 自带的消费者脚本来查看写入的内容。

Scala 代码：

Java 代码：

执行程序。请按以下步骤执行：

在第三个终端窗口（运行 Kafka 消费者脚本的窗口），可以看到如下的输出内容：

```

{"id":103,"title":"Hadoop 大数据处理技术","authors":"zhangsan","year":2017}
{"id":101,"title":"Java 从初学到精通","authors":"xinliwei","year":2019}
{"id":102,"title":"流处理系统","authors":"someone","year":2018}
{"id":104,"title":"Spark 大数据处理技术","authors":"xinliwei","year":2021}

```

Kafka 的生产者和容错

启用 Flink 的检查点后，FlinkKafkaProducer 可以提供精确一次的交付保证。

除了启用 Flink 的检查点，还可以通过传递适当的 semantic 语义参数给 FlinkKafkaProducer 来选择三种不同的操作模式：

- ☐ Semantic.NONE：Flink 将不会做任何保证。生成的记录可能会丢失，也可能会重复。
- ☐ Semantic.AT_LEAST_ONCE：默认设置。这保证了不会丢失任何记录(尽管它们可以被重复)。
- ☐ Semantic.EXACTLY_ONCE：使用 Kafka 事务提供精确一次的语义。当使用事务写入 Kafka 时，不要忘记为任何消费 Kafka 记录的应用程序设置期望的 isolation.level 值（read_committed 或 read_uncommitted，后者是默认值）。

3.11.2 JDBC 连接器

JDBC Connector 提供了一个接收器，用于将数据写入 JDBC 数据库。要使用它，请将以下依赖项添加到项目中：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-jdbc_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.48</version>
</dependency>
```

注：mysql 的版本可以根据自己的实际安装进行修改。

创建的 JDBC 接收器提供了“至少一次”的保证。有效使用 upsert 语句或幂等更新可以实现“精确一次”保证。其用法如下所示：

```
JdbcSink.sink(
    sqlDmlStatement,           // 必须的
    jdbcStatementBuilder,     // 必须的
    jdbcExecutionOptions,     // 可选的
    jdbcConnectionOptions     // 必须的
);
```

SQL DML 语句是用户提供的 SQL 语句，通常带有占位符。

JDBC 语句构建器从用户提供的 SQL 字符串构建一个 JDBC 预编译语句，并重复调用用户指定的函数来使用流中的每个元素值来更新这个预编译语句。

SQL DML 语句是批量执行的，可以选择使用以下实例配置这些语句：

```
JdbcExecutionOptions.builder()
    .withBatchIntervalMs(200) // 可选地: default = 0,这意味着没有基于时间的执行
    .withBatchSize(1000)     // 可选地: default = 5000
    .withMaxRetries(5)       // 可选地: default = 3
    .build()
```

只要满足以下条件之一，就会执行 JDBC 批处理：

- ☐ 已经过配置的批处理间隔时间；

- ❑ 已达到最大批大小；
- ❑ Flink 检查点已经启动。

数据库连接是用 `JdbcConnectionOptions` 实例配置的。

注：从 1.13 开始，Flink JDBC 接收器支持精确一次模式。该实现依赖于 XA 标准的 JDBC 驱动程序支持。

注：在 1.13 中，对于 MySQL 或其他不支持每个连接多个 XA 事务的数据库，Flink JDBC 接收器不支持精确一次模式。

【示例】使用 `JdbcSink` 将流数据写入到 MySQL 数据库。

请按以下步骤执行。

1) 首先，在 MySQL 中执行如下脚本，创建用来接收写入数据的 `books` 数据表。

```
# 创建数据库 xueai8
mysql> create database xueai8;

# 切换到数据库 xueai8
mysql> use xueai8;

# 创建数据表 books
create table books(
  id bigint,
  title varchar(100),
  authors varchar(50),
  year int
);
```

2) 然后，编写 Flink 流处理程序，将数据流中的图书信息写入上面创建的 `books` 数据表中。

Scala 代码：

Java 代码：

3) 执行以上程序。

4) 在 `mysql` 中查询 `books` 表，可以看到数据已经被正确写入了。

```
MariaDB [xueai8]> select * from books;
+-----+-----+-----+-----+
| id   | title                                     | authors                                     | year |
+-----+-----+-----+-----+
| 104  | Spark大数据处理技术                     | xinliwei                                   | 2021 |
| 101  | Java从初学到精通                       | xinliwei                                   | 2019 |
| 103  | Designing Data-Intensive Applications    | Martin Kleppmann                         | 2017 |
| 102  | Streaming Systems                       | Tyler Akidau, Slava Chernyak, Reuven Lax | 2018 |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

3.12 其他

下面是一些不便归于特定主题的内容，包括如何在应用程序中使用日志、如果加载外部配置参数以及如何尽量避免使用大型的元组类型。

3.12.1 在应用程序中使用日志

在任何 Flink 应用程序中使用 SLF4J 时，我们需要导入以下包和类，并用类名初始化日志程序：

```
import org.slf4j.LoggerFactory
import org.slf4j.Logger
```

```
Logger LOG = LoggerFactory.getLogger(MyClass.class)
```

使用占位符机制而不是使用字符串格式化程序进行日志记录也是一种最佳实践。占位符机制有助于避免不必要的字符串格式化，而它只执行字符串连接。下面的代码片段展示了如何使用占位符：

```
LOG.info("Value of a = {}, value of b= {}", myobject.a, myobject.b);
```

我们也可以在异常处理中使用占位符日志：

```
catch(Exception e){
    LOG.error("Error occurred {}", e);
}
```

3.12.2 使用 ParameterTool

从 Flink 0.9 开始，在 Flink 中就有了一个内置的 `ParameterTool`，它可以帮助从外部源（如命令行参数、系统属性或属性文件）获取参数。在内部，它是字符串的 `map` 映射，保留 `key` 作为参数名，`value` 作为参数值。

例如，我们可以考虑在 `DataStream API` 示例中使用 `ParameterTool`，我们需要在其中设置 `Kafka` 属性：

```
String kafkaproperties = "/path/to/kafka.properties";
ParameterTool parameter = ParameterTool.fromPropertiesFile(propertiesFile);
```

从系统属性加载

我们可以读取系统变量中定义的属性。在初始化系统属性文件之前，我们需要通过设置 `Dinput=hdfs://myfile` 来传递系统属性文件。

然后，我们可以在 `ParameterTool` 中读取所有这些属性，如下所示：

```
ParameterTool parameters = ParameterTool.fromSystemProperties();
```

从命令行参数加载

我们还可以从命令行参数中读取参数。我们必须在调用应用程序之前设置 `--elements`。下面的代码展示了如何从命令行参数中读取参数：

```
ParameterTool parameters = ParameterTool.fromArgs(args);
```

从.properties 文件加载

我们还可以从 `.properties` 文件中读取参数。代码如下：

```
String propertiesFile = "/my.properties";
ParameterTool parameters = ParameterTool.fromPropertiesFile(propertiesFile);
```

加载配置参数之后，我们可以在 Flink 程序中读取它们，如下所示：

```
parameter.getRequired("key");
parameter.get("paramterName", "myDefaultValue");
```



```
parameter.getLong("expectedCount", -1L);  
parameter.getNumberOfParameters()
```

3.12.3 命名大型 TupleX 类型

众所周知，元组是一种用于表示复杂数据结构的复杂数据类型。它是各种基本数据类型的组合。一般情况下，建议不要使用大元组；相反，建议使用 Java POJO。如果想使用元组，建议使用一些自定义 POJO 类型来命名它。

为大型元组创建自定义类型非常容易。例如，如果我们想用 Tuple8，然后我们可以这样定义它：

```
// 初始化 Record Tuple  
RecordTuple rc = new RecordTuple(value0, value1, value2, value3, value4, value5, value6, value7);  
  
// 定义 RecordTuple 而不是使用 Tuple8  
public static class RecordTuple extends Tuple8<String, String, Integer, String, Integer, Integer, Integer, Integer> {  
    public RecordTuple() {  
        super();  
    }  
  
    public RecordTuple(String value0, String value1, Integer value2, String value3,  
                        Integer value4, Integer value5, Integer value6, Integer value7) {  
        super(value0, value1, value2, value3, value4, value5, value6, value7);  
    }  
}
```

3.13 Flink 流处理案例

现在我们已经了解了 DataStream API 的各个方面，让我们尝试使用这些概念来解决实际的用例。

3.13.1 处理 IOT 事件流

假设一台机器上安装了传感器，我们希望通过这些传感器收集数据，并每 5 分钟计算每个传感器的平均温度。

其架构如下：

在这个场景中，我们假设传感器将信息发送给 Kafka 主题 temp，信息为（传感器 id、时间戳、温度）。这里假设以字符串的形式接收 Kafka 主题中的事件，部分数据如下所示。

```
sensortemp.csv:  
sensor_1,1629943899014,51.087254019871054  
sensor_9,1629943899014,70.44743245583899  
sensor_7,1629943899014,65.53215956486392  
sensor_0,1629943899014,53.210570822216546  
sensor_8,1629943899014,93.12876931817556  
sensor_3,1629943899014,57.55153052162809
```

```
sensor_2,1629943899014,107.61249366604993
sensor_5,1629943899014,92.02083744773739
sensor_4,1629943899014,95.7688424087137
sensor_6,1629943899014,95.04398353316257
.....
```

现在，我们需要编写 Flink 流处理代码从 Kafka 的 temp 主题读取这些数据，并使用 Flink 转换处理数据。

这里要考虑的是，既然有来自传感器的时间戳值，那么我们可以使用事件时间计算时间因素。这意味着可以处理乱序的传感器数据。

完整的实现代码如下所示。

Scala 代码：

Java 代码：

作业打包

对以上程序打 jar 包。在命令行下，执行以下命令：

```
$ mvn clean package
```

启动 Kafka 服务器并创建 “temp” 主题

要执行作业，首先需要启动 Zookeeper、Kafka 服务，并创建主题 “temp”。请按以下步骤操作：

1) 启动 zookeeper 服务，启动 kafka 服务

打开一个终端窗口，启动 ZooKeeper（不要关闭）

```
$ ./bin/zookeeper-server-start.sh config/zookeeper.properties
```

打开另一个终端窗口，启动 Kafka 服务（不要关闭）

```
$ ./bin/kafka-server-start.sh config/server.properties
```

2) 在 Kafka 中创建一个名为 “temp” 的主题(topic)

```
$ ./bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic temp
```

查看已经创建的 Topic:

```
$ ./bin/kafka-topics.sh --list --bootstrap-server localhost:9092
```

注：要删除一个主题，用以下命令：

```
$ ./bin/kafka-topics.sh --delete --zookeeper localhost:2181 --topic temp
```

在 Flink 集群上执行作业的步骤

要将作业提交到 Flink 集群上运行，请按以下步骤操作：

1) 编写 shell 脚本 streamiot.sh，读取每一行 iot 数据，发送给 kafka 的 “temp” 主题；

2) 然后执行脚本 streamiot.sh:

2) 提交程序 jar 包到集群上运行，抓取 kafka 的 “temp” 主题中消息，并输出在控制台。

观察到输出结果如下：

```
2> (sensor_2,1629943900000,107.4)
```

```
7> (sensor_7,1629943900000,66.11)
1> (sensor_0,1629943900000,52.86)
8> (sensor_9,1629943900000,71.53)
3> (sensor_8,1629943900000,93.09)
1> (sensor_3,1629943900000,57.93)
7> (sensor_4,1629943900000,96.48)
5> (sensor_1,1629943900000,51.64)
6> (sensor_5,1629943900000,92.0)
1> (sensor_0,1629943905000,50.59)
3> (sensor_8,1629943905000,91.96)
8> (sensor_9,1629943905000,71.15)
2> (sensor_2,1629943905000,107.48)
3> (sensor_8,1629943910000,91.13)
1> (sensor_3,1629943905000,60.1)
6> (sensor_6,1629943900000,96.08)
7> (sensor_7,1629943905000,64.93)
5> (sensor_1,1629943905000,53.84)
2> (sensor_2,1629943910000,109.88)
3> (sensor_8,1629943915000,94.77)
1> (sensor_3,1629943910000,62.18)
5> (sensor_1,1629943910000,59.87)
7> (sensor_4,1629943905000,97.99)
5> (sensor_1,1629943915000,63.32)
6> (sensor_6,1629943905000,95.86)
8> (sensor_9,1629943910000,71.55)
7> (sensor_7,1629943910000,67.59)
1> (sensor_0,1629943910000,50.96)
2> (sensor_2,1629943915000,105.35)
7> (sensor_4,1629943910000,100.87)
8> (sensor_9,1629943915000,75.48)
6> (sensor_5,1629943905000,91.47)
7> (sensor_7,1629943915000,69.73)
1> (sensor_0,1629943915000,51.91)
.....
```

3.13.2 信用卡欺诈检测

信用卡诈骗在数字时代是一个越来越令人担忧的问题。罪犯通过运行骗局或侵入不安全的系统来窃取信用卡号码。比如，罪犯通常会先花很小的金额（例如 1 元或更少）尝试购买一件或多件小物品来测试被盗的号码。如果发现购买有效，他们就会进行更大规模的购买，然后进行出售或自己保留使用。

在本案例中，我们将构建一个用于对可疑信用卡交易发出警报的欺诈检测系统。通过使用一组简单的规则，Flink 可以实现高级业务逻辑和实时操作。

实际应用中，数据源将来自外部系统(如 Apache Kafka、Rabbit MQ 或 Apache Pulsar)的数据导入到 Flink 作业中。而在本案例中，我们将创建一个自定义的数据源，用来模拟生成信用卡交易数据流。该源生成一个无限的信用卡交易流供程序处理，每个交易包含一个帐户 ID (accountId)、事务发生时的时间戳(timestamp)和交易金额(amount)。而附加到源的 name 只是为了调试，所以如果出现错误，我们就知道错误来自何处。

交易流包含来自大量用户的交易数据，因此它需要在多个欺诈检测任务中并行处理。由于欺诈发生在每个帐户的基础上，所以必须确保相同帐户的所有交易都由欺诈检测操作符的相同并行任务处理。

数据源

定义信用卡交易事件类 `Transaction`，代码实现如下：

自定义数据源类 `MyTransactionSource`，代码实现如下：

另外，对于检测到的疑似欺诈事件，使用如下的 `Alert` 类来表示：

V1

根据前面的分析，欺诈操作通常所具有的特征是：先花很小的金额（例如 1 元或更少）尝试购买一件或多件小物品来测试被盗的号码。如果发现购买有效，他们就会进行更大规模的购买。因此，针对任何在完成小额交易之后紧接着进行大额交易的账户，欺诈检测器应该输出一个警报。这里我们对小额交易的定义是交易金额小于\$1.00，大额交易的定义是交易金额大于\$500。假设我们的欺诈检测器为某个帐户处理以下交易流。

Txn 1	Txn 2	Txn 3	Txn 4	Txn 5	Txn 6	Txn 7	Txn 8	Txn 9	Txn 10
\$13.01	\$25.00	\$0.09	\$510.00	\$102.62	\$91.50	\$0.02	\$30.01	\$701.83	\$31.92



Fraud



Not-Fraud

要实现这样的功能，欺诈检测器必须记住跨交易的信息：一笔大额交易只有在前一笔交易额很小的情况下才算欺诈。跨交易记忆信息需要状态，所以应该使用 `KeyedProcessFunction`。它提供了对状态和时间的细粒度控制，可根据复杂的需求改进算法。

Flink 提供了容错状态的原语，这些原语几乎与常规成员变量一样易于使用。Flink 中最基本的状态类型是 `ValueState`，这是一种数据类型，它为它包装的任何变量增加了容错能力。`ValueState` 是 `keyed state` 的一种形式，这意味着它只在 `keyed context` 中应用的算子中可用，即在紧跟着 `DataStream#keyBy` 的任何算子中可用。算子的 `keyed state` 作用域自动限定为当前所处理的事件记录的 `key`。在本例中，`key` 是当前交易的帐户 `id` (由 `keyBy()` 声明)，而欺诈检测器为每个帐户维护一个独立的状态。`ValueState` 是使用 `ValueStateDescriptor` 创建的，它包含关于 Flink 应该如何管理该变量的元数据。应该在函数开始处理数据之前注册状态，通常是在 `open()` 方法中实现的。

Scala 代码：

```
@SerialVersionUID(1L)
class FraudDetector extends KeyedProcessFunction[Long, Transaction, Alert] {

    @transient private var flagState: ValueState[java.lang.Boolean] = _

    @throws[Exception]
    override def open(parameters: Configuration): Unit = {
```

```
val flagDescriptor = new ValueStateDescriptor("flag", Types.BOOLEAN)
flagState = getRuntimeContext.getState(flagDescriptor)
}
```

Java 代码:

```
public class FraudDetector extends KeyedProcessFunction<Long, Transaction, Alert> {

    private static final long serialVersionUID = 1L;

    private transient ValueState<Boolean> flagState;

    @Override
    public void open(Configuration parameters) {
        ValueStateDescriptor<Boolean> flagDescriptor = new ValueStateDescriptor<>(
            "flag",
            Types.BOOLEAN);
        flagState = getRuntimeContext().getState(flagDescriptor);
    }
}
```

ValueState 是一个包装类，类似于 Java 标准库中的 AtomicReference 或 AtomicLong。它提供了三种与内容交互的方法：update 设置状态，value 获取当前值，clear 删除其内容。如果某个特定 key 的状态为空，例如在应用程序的开头或调用 ValueState#clear 之后，那么 ValueState#value 将返回 null。对 ValueState#value 返回的对象（即状态值）的修改不能保证被系统识别，因此必须使用 ValueState#update 执行所有修改。另外，容错是由 Flink 底层自动管理的，因此可以像与任何标准变量交互一样与它交互。

在下面的代码中，使用标记状态跟踪潜在欺诈交易。

Scala 代码:

```
override def processElement(
    transaction: Transaction,
    context: KeyedProcessFunction[Long, Transaction, Alert]#Context,
    collector: Collector[Alert]): Unit = {

    // 获取当前 key 的当前状态
    val lastTransactionWasSmall = flagState.value

    // 检查是否设置了标志 flag
    if (lastTransactionWasSmall != null) {
        if (transaction.getAmount > FraudDetector.LARGE_AMOUNT) {
            // 输出下游警报
            val alert = new Alert
            alert.setTid(transaction.getAccountTid)

            collector.collect(alert)
        }
        // 清除状态
        flagState.clear()
    }

    if (transaction.getAmount < FraudDetector.SMALL_AMOUNT) {
        // 设置标志 flag 为 true
    }
}
```

```
        flagState.update(true)
    }
}
```

Java 代码:

```
@Override
public void processElement(
    Transaction transaction,
    Context context,
    Collector<Alert> collector) throws Exception {

    // 获取当前 key 的当前状态
    Boolean lastTransactionWasSmall = flagState.value();

    // 检查是否设置了标志
    if (lastTransactionWasSmall != null) {
        if (transaction.getAmount() > LARGE_AMOUNT) {
            // 向下游输出警报
            Alert alert = new Alert();
            alert.setId(transaction.getAccountId());

            collector.collect(alert);
        }

        // 清楚状态
        flagState.clear();
    }

    if (transaction.getAmount() < SMALL_AMOUNT) {
        // 将 flag 设置为 true
        flagState.update(true);
    }
}
```

对于每个交易，欺诈检测器检查该帐户的标志状态。记住，ValueState的作用域始终是当前key，即账户。如果标志为非空，说明该帐户上一次的交易额很小，因此如果这次交易的金额很大，则该检测器输出一个欺诈警报。

检查之后，该标志状态将被无条件清除。要么是当前交易导致了欺诈警报，并且模式已经结束，要么是当前交易没有导致警报，模式已经被打破，需要重新建立。

最后，检查交易金额，看看它是否小额。如果是，则设置该标志，以便下一个事件可以检查它。注意，ValueState<Boolean>实际上有三种状态，unset (null)、true 和 false。本例中仅使用 unset (null)和 true 来检查是否设置了标志。

V2: 状态 + 时间 = 心跳

在先尝试用小额购买之后，骗子不会等很长时间再来进行大的购买，以减少他们的测试交易被注意到的机会。假设我们想要为欺诈检测器设置1分钟的超时，例如，在前面的例子中，交易3和交易4只有在间隔1分钟内发生时才被认为是欺诈。Flink的KeyedProcessFunction提供了设置定时器的方法，我们可以设置在将来某个时间点调用回调方法的定时器。

我们需要修改之前的作业，以符合新的要求：

- ❑ 每当将标志 `flag` 设置为 `true`，也要设置一个 1 分钟的定时器。
- ❑ 当定时器触发时，通过清除其状态来重置标记。
- ❑ 如果标记被清除，那么定时器应该被取消。

要取消定时器，就必须记住它的设置时间，而记住意味着状态，因此我们从创建计时器状态和标记状态开始。

Scala 代码：

```
@SerialVersionUID(1L)
class FraudDetector extends KeyedProcessFunction[Long, Transaction, Alert] {

    @transient private var flagState: ValueState[java.lang.Boolean] = _    // 标志状态
    @transient private var timerState: ValueState[java.lang.Long] = _      // 计时器状态

    @throws[Exception]
    override def open(parameters: Configuration): Unit = {
        val flagDescriptor = new ValueStateDescriptor("flag", Types.BOOLEAN)
        flagState = getRuntimeContext.getState(flagDescriptor)

        val timerDescriptor = new ValueStateDescriptor("timer-state", Types.LONG)
        timerState = getRuntimeContext.getState(timerDescriptor)
    }
}
```

Java 代码：

```
private transient ValueState<Boolean> flagState;    // 标志状态
private transient ValueState<Long> timerState;     // 计算器状态

@Override
public void open(Configuration parameters) {
    ValueStateDescriptor<Boolean> flagDescriptor = new ValueStateDescriptor<>(
        "flag",
        Types.BOOLEAN);
    flagState = getRuntimeContext().getState(flagDescriptor);

    ValueStateDescriptor<Long> timerDescriptor = new ValueStateDescriptor<>(
        "timer-state",
        Types.LONG);
    timerState = getRuntimeContext().getState(timerDescriptor);
}
```

`KeyedProcessFunction#processElement` 被调用，其传入参数 `Context` 包含一个定时器服务。定时器服务可用于查询当前时间、注册定时器和删除定时器。这样，可以在每次设置标志时设置 1 分钟的定时器，并将时间戳存储在 `timerState` 中。

Scala 代码：

```
if (transaction.getAmount < FraudDetector.SMALL_AMOUNT) {
    // 将标志设为 true
    flagState.update(true)

    // 设置定时器和定时器状态
    val timer = context.timerService.currentProcessingTime + FraudDetector.ONE_MINUTE
}
```

```

    context.timerService.registerProcessingTimeTimer(timer)
    timerState.update(timer)
}

```

Java 代码:

```

if (transaction.getAmount() < SMALL_AMOUNT) {
    // 将标志设为 true
    flagState.update(true);

    // 设置定时器和定时器状态
    long timer = context.timerService().currentProcessingTime() + ONE_MINUTE;
    context.timerService().registerProcessingTimeTimer(timer);    // 注册定时器
    timerState.update(timer);
}

```

处理时间为挂钟时间，由运行操作符的机器的系统时钟决定。当定时器触发时，它调用 `KeyedProcessFunction#onTimer` 方法。重写此方法实现回调以重置标记。

Scala 代码:

```

override def onTimer(
    timestamp: Long,
    ctx: KeyedProcessFunction[Long, Transaction, Alert]#OnTimerContext,
    out: Collector[Alert]): Unit = {
    // 1 分钟后删除定时器
    timerState.clear()
    flagState.clear()
}

```

Java 代码:

```

@Override
public void onTimer(long timestamp, OnTimerContext ctx, Collector<Alert> out) {
    // 1 分钟后删除定时器
    timerState.clear();
    flagState.clear();
}

```

最后，要取消定时器，需要删除已注册的定时器并删除定时器状态。我们将其封装在辅助方法中并调用该方法，而不是直接调用 `flagState.clear()`。

Scala 代码:

```

@throws[Exception]
private def cleanUp(ctx: KeyedProcessFunction[Long, Transaction, Alert]#Context): Unit = {
    // 删除定时器
    val timer = timerState.value
    ctx.timerService.deleteProcessingTimeTimer(timer)

    // 清除所有状态
    timerState.clear()
    flagState.clear()
}

```

Java 代码:

```

private void cleanUp(Context ctx) throws Exception {
    // 删除定时器
}

```



```

Long timer = timerState.value();
ctx.timerService().deleteProcessingTimeTimer(timer);

// 清除所有状态
timerState.clear();
flagState.clear();
}

```

这就完成了一个功能齐全、有状态的分布式流应用程序！

完整代码如下。

Scala 代码：

Java 代码：

测试程序如下。

Java 代码：

使用已准备好的 TransactionSource 数据源运行这个代码，将会检测到账户 3 的欺诈行为，并输出报警信息。能够在 task manager 的日志中看到下边输出：

2019-08-19 14:22:06,220 INFO	org.apache.flink.walkthrough.common.sink.AlertSink	- Alert{id=3}
2019-08-19 14:22:11,383 INFO	org.apache.flink.walkthrough.common.sink.AlertSink	- Alert{id=3}
2019-08-19 14:22:16,551 INFO	org.apache.flink.walkthrough.common.sink.AlertSink	- Alert{id=3}
2019-08-19 14:22:21,723 INFO	org.apache.flink.walkthrough.common.sink.AlertSink	- Alert{id=3}
2019-08-19 14:22:26,896 INFO	org.apache.flink.walkthrough.common.sink.AlertSink	- Alert{id=3}

3.13.3 订单延迟支付告警程序

需求描述

在电商平台中，有效的订单，，是用户真正完成支付动作的时候。用户下单的行为可以表明用户对商品的需求，但在现实中，并不是每次下单都会被用户立刻支付。当拖延一段时间后，用户支付的意愿会降低。

所以为了让用户更有紧迫感从而提高支付转化率，同时也为了防范订单支付环节的安全风险，电商网站往往会对订单状态进行监控，设置一个失效时间（比如 15 分钟），如果下单后一段时间仍未支付，订单就会被取消。

此时需要给用户发送一个信息提醒用户，提高支付转换率！

需求分析

本需求可以使用 process function 原生的状态编程。

问题可以简化成：在 pay 事件超时未发生的情况下，输出超时报警信息。

实现思路

- ❑ 在订单的 create 事件到来后注册定时器，15 分钟后触发；
- ❑ 用一个布尔类型的 ValueState 来作为标识位，表明 pay 事件是否发生过。
- ❑ 如果 pay 事件已经发生，状态被置为 true，那么就不再需要做什么操作；
- ❑ 而如果 pay 事件一直没来，状态一直为 false，到定时器触发时，就应该输出超时报警信息。

示例数据

以下为订单事件数据格式，各字段含义分别为：用户 ID，订单状态，订单 ID，时间戳。

```
34729,create,,1558430842
34730,create,,1558430843
34729,pay,sd76f87d6,1558430844
34730,modify,3hu3k2432,1558430845
34731,create,,1558430846
34731,pay,35jue34we,1558430849
34732,create,,1558430852
34733,create,,1558430855
34734,create,,1558430859
34734,create,,1558431000
34733,pay,,1558431000
34732,pay,,1558449999
```

实现参考

Scala 代码：

Java 代码：

执行：

```
$ cat order.txt | nc -lk 9999
```

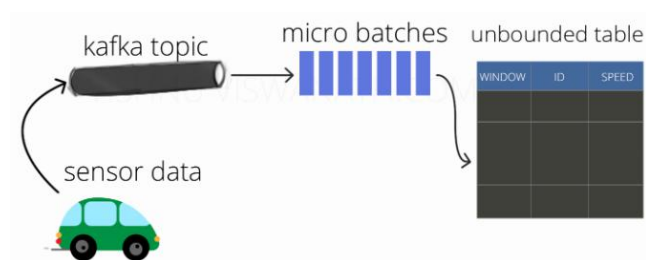
输出内容如下所示：

```
payed> OrderResult(34729,payed successfully)
payed> OrderResult(34731,payed successfully)
payed> OrderResult(34733,payed successfully)
time out order> OrderResult(34732,this order is timeout)
time out order> OrderResult(34730,order timeout)
time out order> OrderResult(34734,order timeout)
time out order> OrderResult(34734,order timeout)
```

3.13.4 运输公司车辆超速实时监测

让我们想象一个物流公司车队管理解决方案，其中车队中的车辆启用了无线网络功能。每辆车定期报告其地理位置和许多操作参数，如燃油水平、速度、加速度、轴承、发动机温度等。物流公司希望利用这一遥测数据流来实现一系列应用程序，以帮助他们管理业务的运营和财务方面。

假设我们开了一家运输公司，需要检查车辆是否超速。我们将创建一个简单的实时的流应用程序来计算车辆每几秒钟的平均速度。



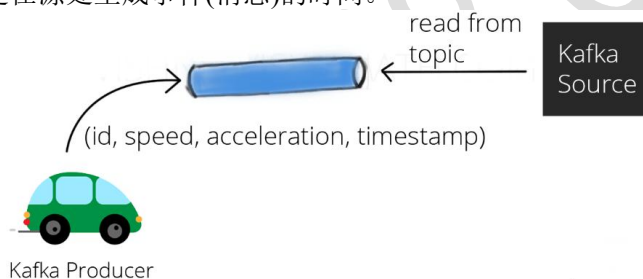
我们使用 Kafka 作为流数据源，将从 Kafka 的“cars”主题来读取这些事件。同时也将 Kafka 作为流的 Data Sink，检测出的超速事件将写入到 Kafka 的“fastcars”主题。

为了模拟车辆向我们发送传感器数据，我们将创建一个 Kafka producer，它将 id、speed、acceleration 和 timestamp 写入 Kafka 的“cars”主题。

Scala 代码：

Java 代码：

注意，这里的时间戳是在源处生成事件(消息)的时间。



下面是流处理管道的实现。

Scala 代码：

Java 代码：

请按以下步骤执行。

1、启动 zookeeper

2、启动 Kafka

3、创建两个 topic:

4、先在一个新的终端窗口中，执行消费者脚本，来拉取 fastcars topic 数据:

5、执行流计算程序

6、执行数据源程序

7、回到消息者脚本执行窗口（第 4 步的窗口），查看超速数据。如果一切正常，应该看到在“fastcars” topic 收到的超速数据如下（部分）：

.....

```

{"carId":"car2","avgSpeed":144.0,"start":"2021-08-27 12:20:48.0","end":"2021-08-27 12:20:53.0"}
{"carId":"car7","avgSpeed":130.0,"start":"2021-08-27 12:20:48.0","end":"2021-08-27 12:20:53.0"}
{"carId":"car7","avgSpeed":130.0,"start":"2021-08-27 12:20:50.0","end":"2021-08-27 12:20:55.0"}
{"carId":"car4","avgSpeed":148.0,"start":"2021-08-27 12:20:56.0","end":"2021-08-27 12:21:01.0"}
{"carId":"car4","avgSpeed":148.0,"start":"2021-08-27 12:20:58.0","end":"2021-08-27 12:21:03.0"}
{"carId":"car4","avgSpeed":148.0,"start":"2021-08-27 12:21:00.0","end":"2021-08-27 12:21:05.0"}
{"carId":"car1","avgSpeed":126.5,"start":"2021-08-27 12:21:00.0","end":"2021-08-27 12:21:05.0"}
{"carId":"car1","avgSpeed":134.0,"start":"2021-08-27 12:21:02.0","end":"2021-08-27 12:21:07.0"}
{"carId":"car1","avgSpeed":134.0,"start":"2021-08-27 12:21:04.0","end":"2021-08-27 12:21:09.0"}
{"carId":"car7","avgSpeed":149.0,"start":"2021-08-27 12:21:12.0","end":"2021-08-27 12:21:17.0"}
{"carId":"car7","avgSpeed":149.0,"start":"2021-08-27 12:21:14.0","end":"2021-08-27 12:21:19.0"}
{"carId":"car7","avgSpeed":149.0,"start":"2021-08-27 12:21:16.0","end":"2021-08-27 12:21:21.0"}
{"carId":"car6","avgSpeed":125.0,"start":"2021-08-27 12:21:16.0","end":"2021-08-27 12:21:21.0"}
{"carId":"car1","avgSpeed":139.0,"start":"2021-08-27 12:21:20.0","end":"2021-08-27 12:21:25.0"}
{"carId":"car1","avgSpeed":139.0,"start":"2021-08-27 12:21:22.0","end":"2021-08-27 12:21:27.0"}
{"carId":"car0","avgSpeed":144.0,"start":"2021-08-27 12:21:40.0","end":"2021-08-27 12:21:45.0"}
{"carId":"car0","avgSpeed":144.0,"start":"2021-08-27 12:21:42.0","end":"2021-08-27 12:21:47.0"}
{"carId":"car0","avgSpeed":144.0,"start":"2021-08-27 12:21:44.0","end":"2021-08-27 12:21:49.0"}
{"carId":"car4","avgSpeed":130.0,"start":"2021-08-27 12:21:46.0","end":"2021-08-27 12:21:51.0"}
{"carId":"car4","avgSpeed":130.0,"start":"2021-08-27 12:21:48.0","end":"2021-08-27 12:21:53.0"}
{"carId":"car4","avgSpeed":130.0,"start":"2021-08-27 12:21:50.0","end":"2021-08-27 12:21:55.0"}
.....

```

3.13.5 计算实时热门商品

某电商网站实时采集用户行为数据，每一条数据表示一条用户行为，由用户 ID、商品 ID、商品类目 ID、行为类型和时间戳组成，并以逗号分隔。其中用户行为包括点击、购买、加购、收藏等行为。

部分数据如下所示：

```

543462,1715,1464116,pv,1511658000
662867,2244074,1575622,pv,1511658000
561558,3611281,965809,pv,1511658000
894923,3076029,1879194,pv,1511658000
834377,4541270,3738615,pv,1511658000
315321,942195,4339722,pv,1511658000

```

数据各字段说明如下：

字段名称	字段说明
用户ID	整数类型，加密后的用户ID
商品ID	整数类型，加密后的商品ID
商品类目ID	整数类型，加密后的商品所属类目ID
行为类型	字符串，枚举类型，包括('pv', 'buy', 'cart', 'fav')
时间戳	行为发生的时间戳，单位秒

注：这里使用来自阿里云天池公开数据集中的淘宝用户行为数据集来模拟实时用户行为，特别感谢！本数据集包含了淘宝上某一天随机一百万用户的所有行为（包括点击、购买、加购、收藏）。

现在我希望实时计算“热门商品”：每隔 5 分钟输出最近一小时内点击量最多的前 N 个商品。将这个需求进行分解，我们大概需要做这么几件事情：

- ❑ 抽取出业务时间戳，告诉 Flink 框架基于事件时间开窗口。
- ❑ 过滤出点击行为数据。

- ❑ 按一小时的窗口大小，每 5 分钟统计一次，基于滑动窗口（Sliding Window）执行聚合操作。
- ❑ 按每个窗口聚合，输出每个窗口中点击量前 N 名的商品。

我们首先分析实现的步骤，并在最后给出完整的代码实现。

首先需要创建一个 POJO 类 UserBehavior，用来表示用户行为的事件数据结构。

```
public class UserBehavior {  
    public long userId;           // 用户 ID  
    public long itemId;          // 商品 ID  
    public int categoryId;        // 商品类目 ID  
    public String behavior;       // 用户行为, 包括("pv", "buy", "cart", "fav")  
    public long timestamp;        // 行为发生的时间戳, 单位秒  
  
    public UserBehavior(){}  
  
    public UserBehavior(long userId, long itemId, int categoryId, String behavior, long timestamp) {  
        this.userId = userId;  
        this.itemId = itemId;  
        this.categoryId = categoryId;  
        this.behavior = behavior;  
        this.timestamp = timestamp;  
    }  
}
```

在本例中，我们需要统计业务时间上的每小时的点击量，所以要基于事件时间（EventTime）来处理。使用如下的代码获取流执行环境并设置使用事件时间：

```
// 获得流执行环境  
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  
  
// 设置基于 EventTime 来处理。  
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

为了模拟实时的用户行为数据流，我们把数据集 UserBehavior.csv 拷贝到项目的 resources 目录下，然后创建一个自定义的数据源，加载该数据集，生成实时数据流。实现代码如下：

```
// UserBehavior.csv 的本地文件路径  
URL fileUrl = HotelTopN.class.getClassLoader().getResource("UserBehavior.csv");  
Path filePath = Path.fromLocalFile(new File(fileUrl.toURI()));  
  
// 抽取 UserBehavior 的 TypeInformation，是一个 PojoTypeInfo  
PojoTypeInfo<UserBehavior> pojoType =  
    (PojoTypeInfo<UserBehavior>) TypeExtractor.createTypeInfo(UserBehavior.class);  
  
// 由于 Java 反射抽取出的字段顺序是不确定的，需要显式指定下文件中字段的顺序  
String[] fieldOrder = new String[]{"userId", "itemId", "categoryId", "behavior", "timestamp"};  
  
// 创建 PojoCsvInputFormat  
PojoCsvInputFormat<UserBehavior> csvInput = new PojoCsvInputFormat<>(filePath, pojoType, fieldOrder);
```

在上面的代码中，我们创建一个 PojoCsvInputFormat，这是一个读取 csv 文件并将每一行转成指定 POJO 类型的输入器。

下一步我们用 PojoCsvInputFormat 创建输入源。

```
// 用 PojoCsvInputFormat 创建输入源
```

```
DataStream<UserBehavior> dataSource = env.createInput(csvInput, pojoType);
```

接下来还需要做的是抽取事件时间并生成 Watermark。由于我们的数据源的数据已经经过整理，没有乱序，即事件的时间戳是单调递增的，所以可以将每条数据的事件时间就当做 Watermark。这里我们用 AscendingTimestampExtractor 来实现时间戳的抽取和 Watermark 的生成。注：真实业务场景一般都是存在乱序的，所以一般使用 BoundedOutOfOrderTimestampExtractor。

```
// 实现时间戳的抽取和 Watermark 的生成。
```

```
DataStream<UserBehavior> usersData = dataSource
```

```
    .assignTimestampsAndWatermarks(new AscendingTimestampExtractor<UserBehavior>() {
```

```
        @Override
```

```
        public long extractAscendingTimestamp(UserBehavior userBehavior) {
```

```
            // 原始数据单位秒，将其转成毫秒
```

```
            return userBehavior.timestamp * 1000;
```

```
        }
```

```
    });
```

由于原始数据中存在点击、加购、购买、收藏各种行为的数据，但是我们只需要统计点击量，所以先使用 FilterFunction 将点击行为数据过滤出来。

```
// 过滤出点击事件
```

```
DataStream<UserBehavior> pvData = usersData
```

```
    .filter(new FilterFunction<UserBehavior>() {
```

```
        @Override
```

```
        public boolean filter(UserBehavior userBehavior) throws Exception {
```

```
            // 过滤出只有点击的数据
```

```
            return userBehavior.behavior.equals("pv");
```

```
        }
```

```
    });
```

由于要每隔 5 分钟统计一次最近一小时每个商品的点击量，所以需要创建一个大小是 1 个小时、每隔 5 分钟滑动一次的滑动窗口，即分别要统计 [09:00, 10:00), [09:05, 10:05), [09:10, 10:10)··· 等窗口的商品点击量。实现代码如下：

```
// 窗口统计点击量
```

```
DataStream<ItemViewCount> windowedData = pvData
```

```
    .keyBy("itemId")
```

```
    .timeWindow(Time.minutes(60), Time.minutes(5))
```

```
    .aggregate(new CountAgg(), new WindowResultFunction());
```

上面代码中，我们使用 .keyBy("itemId") 对商品进行分组，使用 .timeWindow(Time size, Time slide) 对每个商品做滑动窗口（1 小时窗口，5 分钟滑动一次）。然后我们使用 .aggregate(AggregateFunction af, WindowFunction wf) 做增量聚合操作，它能使用 AggregateFunction 提前聚合掉数据，减少状态的存储压力。aggregate() 方法的第一个参数 CountAgg 实现了 AggregateFunction 接口，功能是统计窗口中的条数，即遇到一条数据就加一。

```
// 统计窗口中的条数，即遇到一条数据就加一
```

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

```
    @Override
```

```
    public Long createAccumulator() {
```

```
        return 0L;
```

```
    }
```

```

@Override
public Long add(UserBehavior userBehavior, Long acc) {
    return acc + 1;
}

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

@Override
public Long merge(Long acc1, Long acc2) {
    return acc1 + acc2;
}
}

```

.aggregate()方法的第二个参数 WindowFunction, 将每个 key 每个窗口聚合后的结果带上其他信息进行输出。我们这里实现的 WindowResultFunction 将商品 ID、窗口、点击量封装成了 ItemViewCount 进行输出。

ItemViewCount 是一个 POJO 类:

```

public class ItemViewCount {
    public long itemId;           // 商品 ID
    public long windowEnd;       // 窗口结束时间戳
    public long viewCount;       // 商品的点击量

    public ItemViewCount(){}

    public ItemViewCount (long itemId, long windowEnd, long viewCount) {
        this.itemId = itemId;
        this.windowEnd = windowEnd;
        this.viewCount = viewCount;
    }
}

```

WindowResultFunction 实现了 WindowFunction 接口:

```

// 用于输出窗口的结果
public static class WindowResultFunction
    implements WindowFunction<Long, ItemViewCount, Tuple, TimeWindow> {

    @Override
    public void apply(
        Tuple key,                // 窗口的主键, 即 itemId
        TimeWindow window,       // 窗口
        Iterable<Long> aggregateResult, // 聚合函数的结果, 即 count 值
        Collector<ItemViewCount> collector // 输出类型为 ItemViewCount
    ) throws Exception {
        Long itemId = ((Tuple1<Long>) key).f0;
        Long count = aggregateResult.iterator().next();
        collector.collect(new ItemViewCount(itemId, window.getEnd(), count));
    }
}

```

为了统计每个窗口下最热门的商品，我们需要再次按窗口进行分组，这里根据 ItemViewCount 中的 windowEnd 进行 keyBy() 操作。然后使用自定义的 ProcessFunction 计算点击量排名前 3 名的商品，并将排名结果格式化成字符串，便于后续输出。

```
// 再次按窗口进行分组
DataStream<String> topN = windowedData
    .keyBy("windowEnd")
    .process(new TopNProcessFunction(3)); // 求点击量前 3 名的商品
```

上面代码中的 TopNProcessFunction 继承自 KeyedProcessFunction，实现了一个自定义的 TopN 函数，用来计算点击量排名前 3 名的商品，并使用了 ListState 来存储收到的每条 ItemViewCount 消息，保证在发生故障时，状态数据的不丢失和一致性。我们在 onTimer() 中处理将收集的所有商品及点击量进行排序，选出 TopN，并将排名信息格式化成字符串后进行输出。

```
// 求某个窗口中前 N 名的热门点击商品，key 为窗口时间戳，输出为 TopN 的结果字符串
public static class TopNProcessFunction extends KeyedProcessFunction<Tuple, ItemViewCount, String> {

    private final int topSize;          // top N 中的

    public TopNProcessFunction(int topSize) {
        this.topSize = topSize;
    }

    // 用于存储商品与点击数的状态，待收齐同一个窗口的数据后，再触发 TopN 计算
    private ListState<ItemViewCount> itemState;

    @Override
    public void open(Configuration parameters) throws Exception {
        super.open(parameters);
        // 状态的注册
        ListStateDescriptor<ItemViewCount> itemsStateDesc = new ListStateDescriptor<>(
            "itemState-state",
            ItemViewCount.class);
        itemState = getRuntimeContext().getListState(itemsStateDesc);
    }

    @Override
    public void processElement(
        ItemViewCount input,
        Context context,
        Collector<String> collector) throws Exception {

        // 每条数据都保存到状态中
        itemState.add(input);
        // 注册 windowEnd+1 的 EventTime Timer,
        // 当触发时，说明收齐了属于 windowEnd 窗口的所有商品数据
        context.timerService().registerEventTimeTimer(input.windowEnd + 1);
    }

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



```

// 获取收到的所有商品点击量
List<ItemViewCount> allItems = new ArrayList<>();
for (ItemViewCount item : itemState.get()) {
    allItems.add(item);
}
// 提前清除状态中的数据，释放空间
itemState.clear();
// 按照点击量从大到小排序
allItems.sort(new Comparator<ItemViewCount>() {
    @Override
    public int compare(ItemViewCount o1, ItemViewCount o2) {
        return (int) (o2.viewCount - o1.viewCount);
    }
});
// 将排名信息格式化成 String，便于打印
StringBuilder result = new StringBuilder();
result.append("=====\n");
result.append("时间: ").append(new Timestamp(timestamp-1)).append("\n");
for (int i=0;i<topSize;i++) {
    ItemViewCount currentItem = allItems.get(i);
    // 输出格式为: No1: 商品 ID=12224 浏览量=2413
    result.append("No").append(i).append(":")
        .append(" 商品 ID=").append(currentItem.itemId)
        .append(" 浏览量=").append(currentItem.viewCount)
        .append("\n");
}
result.append("=====\n\n");

out.collect(result.toString());
}
}

```

最后列出完整的代码。

1) POJO 类 UserBehavior，作为用户行为数据结构。

2) POJO 类 ItemViewCount，是窗口操作输出类，并保存在状态中：

3) 主程序类 HotelTopN，实现如下：

执行程序，可以在控制台看到输出结果（部分）如下所示：

```

=====
时间: 2017-11-26 09:05:00.0
No0: 商品 ID=5051027 浏览量=3
No1: 商品 ID=3493253 浏览量=3
No2: 商品 ID=4261030 浏览量=3
=====
=====

```

时间: 2017-11-26 09:10:00.0

No0: 商品 ID=812879 浏览量=5

No1: 商品 ID=2600165 浏览量=4

No2: 商品 ID=2828948 浏览量=4

=====

.....

扩展作业:

扩展 1: 在 Linux 中使用脚本模拟生成流数据, 发送到 Kafka, 然后由 Flink 程序读取实时计算, 并将计算结果实时写入到 Kafka。

扩展 2: 编写 Kafka 客户端程序, 将结果实时写出到 HBase, 再对接 Web 后端实时显示。(Node.js 或 WebSocket)

扩展 3: 在 Linux 中使用脚本模拟生成流数据, 发送到 Kafka, 然后由 Flink 程序读取实时计算, 将结果写出到 ElasticSearch。

第 4 章 开发 Flink 批数据处理程序

对于 Flink，批处理是流处理的一种特殊情况。Flink 支持通过 DataSet API 对数据进行批处理。Flink 中的 DataSet 程序是实现数据集转换(如过滤、映射、连接、分组)的常规程序。

4.1 Flink 批处理程序编程模型

在深入了解 Flink 批数据处理程序开发之前，我们先通过一个简单示例来了解使用 Flink 的 DataSet API 构建批处理应用程序的过程。Flink 程序可以在不同的上下文中运行，可以独立运行，也可以嵌入到其他程序中。执行可以发生在本地 JVM 中，也可以发生在多台机器的集群中。

注：从 Flink 1.12 开始，DataSet API 已经被软弃用。官方建议使用 Table API 和 SQL 在一个完全统一的 API 中运行高效的批处理管道。Table API 与普通批处理连接器和目录集成得很好。或者，还可以使用带有 BATCH 执行模式的 DataStream API。随着开发的进展，DataSet API 最终将被删除。有关这一决定的背景信息，请参阅 FLIP-131。

4.1.1 应用程序实现

下面的程序是一个完整的 WordCount 工作示例。

Scala 代码：

```
package com.xueai8.scala.ch04

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

/**
 * 单词计数
 */
object Wordcount {

  def main(args: Array[String]) {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    val text = env.fromElements(
      "good good study",
      "day day up")

    val counts = text.flatMap { _.toLowerCase.split("\\W+") filter { _.nonEmpty } }
      .map { (_, 1) }
      .groupBy(0)
      .sum(1)
```

```
        counts.print()
    }
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.util.Collector;

/**
 * 单词计数 批处理程序
 */
public class Wordcount {

    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        DataSet<String> text = env.fromElements(
            "good good study",
            "day day up");

        DataSet<Tuple2<String, Integer>> wordCounts = text
            .flatMap(new LineSplitter())
            .groupBy(0)
            .sum(1);

        wordCounts.print();
    }

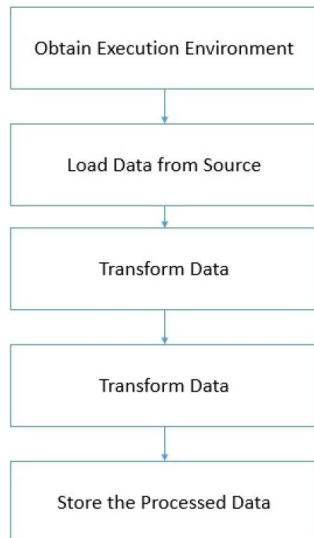
    public static class LineSplitter implements FlatMapFunction<String, Tuple2<String, Integer>> {
        @Override
        public void flatMap(String line, Collector<Tuple2<String, Integer>> out) {
            for (String word : line.split(" ")) {
                out.collect(new Tuple2<String, Integer>(word, 1));
            }
        }
    }
}
```

执行以上程序，输出结果如下所示：

```
(up,1)
(day,2)
(good,2)
(study,1)
```

4.1.2 应用程序剖析

所有的 Flink 程序都以特定的步骤来工作。这些工作步骤如下图所示：



也就是说，每个 Flink 程序都由相同的基本部分组成：

- ☐ 获取一个执行环境；
- ☐ 加载/创建初始数据；
- ☐ 指定对该数据的转换；
- ☐ 指定计算结果放在哪里；
- ☐ 触发程序执行。

1、获取一个执行环境

为了开始编写 Flink 程序，我们首先需要获得一个现有的执行环境，如果没有的话，就需要先创建一个。根据我们的目的不同，Flink 支持以下几种方式：

- ☐ 获得一个已经存在的 Flink 环境
- ☐ 创建本地环境
- ☐ 创建远程环境

Flink 批处理程序的入口点是 `ExecutionEnvironment` 类的一个实例，它定义了程序执行的上下文。可以通过使用以下这些静态方法获得一个 `ExecutionEnvironment` 的实例：

```
ExecutionEnvironment.getExecutionEnvironment()
ExecutionEnvironment.createLocalEnvironment()
ExecutionEnvironment.createRemoteEnvironment(String host, int port, String... jarFiles)
```

要获得执行环境，通常只需要调用 `getExecutionEnvironment()` 方法。这将根据我们的上下文选择正确的执行环境。如果正在 IDE 中的本地环境上执行，那么它将启动一个本地执行环境。如果是从程序中创建了一个 JAR 文件，并通过命令行调用它，那么 Flink 集群管理器将执行 `main()` 方法，`getExecutionEnvironment()` 将返回用于在集群上以分布式方式执行程序的执行环境。

在上面的示例程序中，我们使用以下语句来获得批处理程序的执行环境。

Scala 代码：

```
// 设置批处理程序执行环境
val env = ExecutionEnvironment.getExecutionEnvironment
```

Java 代码:

```
// 设置批处理执行环境
```

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

2、加载/创建初始数据

执行环境可以从多种数据源读取数据, 包括文本文件、CSV 文件等, 也可以使用自定义的数据输入格式。例如, 要将文本文件读取为行序列, 可以使用以下命令。

Scala 代码:

```
val env = ExecutionEnvironment.getExecutionEnvironment
val text = env.readTextFile("file:///path/to/file")
```

Java 代码:

```
final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
DataSet<String> text = env.readTextFile("file:///path/to/file");
```

数据被逐行读取到内存后, Flink 会将它们组织到 `DataSet` 中, 这是 Flink 中用来表示批数据的特殊类。

在上面的示例程序中, 我们使用 `fromElements` 方法读取内存集合数据, 并将读取的数据存储为 `DataSet` 类型。

Scala 代码:

```
val text = env.fromElements("good good study", "day day up")
```

Java 代码:

```
DataSet<String> text = env.fromElements("good good study", "day day up");
```

3、对数据进行转换

每个 Flink 程序都对分布式数据集合执行转换。Flink 的 `DataSet` API 提供了多种数据转换功能, 包括过滤、映射、连接、分组和聚合。在上节的示例程序中, 我们连续使用了多个不同的转换, 将原始数据集转换为新的 `DataSet`。

Scala 代码:

```
// 对读取的每一行文本, 进行分割、过滤、转换等操作, 然后进行汇总统计
val counts = text.flatMap { _.toLowerCase.split("\\W+") filter { _.nonEmpty } }
    .map { (_, 1) }
    .groupBy(0)
    .sum(1)
```

Java 代码:

```
DataSet<Tuple2<String, Integer>> wordCounts = text
    .flatMap(new LineSplitter())
    .groupBy(0)
    .sum(1);
```

这里暂时不必了解每个转换的具体含义, 后面我们会详细介绍每个转换。需要强调的是, Flink 中的转换是惰性的, 在调用 `sink` 操作之前不会真正执行。

4、指定计算结果放在哪里

一旦有了包含最终结果的 `DataSet`, 就可以通过创建接收器 (sink) 将其写入外部系统。在上面的示例程序中, 我们将计算结果打印输出到控制台上。

Scala 代码:

```
counts.print()
```

Java 代码:

```
wordCounts.print()
```

Flink 中的接收器 (sink) 操作触发批计算的执行, 以生成程序所需的结果, 例如将结果保存到文件系统或将其打印到标准输出。上面的示例使用 `counts.print()` 将结果打印到任务管理器日志中(在 IDE 中运行时, 任务管理器日志将显示在 IDE 的控制台中)。这将对批的每个元素调用其 `toString()` 方法。

4.2 数据源

数据源是 DataSet API 希望从其中获取数据的地方。它可以以文件或 Java 集合的形式出现。DataSet API 支持许多内置的数据源函数。它还支持编写自定义数据源函数, 因此不支持的任何东西都可以轻松编程。

DataSet API 支持从多个数据源读取批数据集到 Flink 系统, 并将它们转换为 DataSet 数据集。它主要包括三种类型: 基于文件的数据源、基于集合的数据源和通用数据源。同时, 可以在 DataSet API 中定制 InputFormat / RichInputFormat 接口, 以访问不同数据格式的数据源, 如 CsvInputFormat、TextInputFormat 等。从 ExecutionEnvironment 类提供的方法中, 我们可以看到支持的数据源方法, 如下图所示:

接下来, 我们通过示例程序来分别了解这些数据源函数。

4.2.1 基于文件的数据源

Flink 支持从文件中读取数据。它逐行读取数据并将其作为字符串返回。以下是我们可以用来读取数据的内置函数:

- ❑ `readTextFile(String path)/TextInputFormat`: 从路径指定的文件中传输数据。默认情况下, 它将读取 `TextInputFormat` 并逐行读取字符串。
- ❑ `readTextFileWithValue(String path)/TextValueInputFormat`: 从路径指定的文件中传输数据。它返回可变字符串。
- ❑ `readCsvFile(String path)/CsvInputFormat`: 从逗号分隔的文件中读取数据。它返回 Java POJO 或 tuples 或 case class 对象。
- ❑ `readFileofPrimitives(path,class)/readFileofPrimitives(path,delimiter)/PrimitiveInputFormat`: 这将把新行解析为基本数据类型, 如字符串或整数。
- ❑ `readFileofPrimitives(path,delimiter,class)/PrimitiveInputFormat`
- ❑ `readHadoopFile(FileInputFormat, Key, Value, path)`: 它使用给定的 `FileInputFormat`、`Key` 类和 `Value` 类从指定路径读取文件。它将解析后的值作为元组 `Tuple2<Key,Value>` 返回。
- ❑ `readSequenceFile(Key, Value, path)/SequenceFileInputFormat`: 创建一个 `JobConf` 并使用给定的 `SequenceFileInputFormat`、`Key` 类和 `Value` 类从指定路径读取文件。它将解析后的值作为元组 `Tuple2<Key,Value>` 返回。

注: 对于文件数据源, Flink 支持递归遍历给定路径中指定的文件夹。为了使用这个功能, 我们需要设置一个环境变量, 并在读取数据时将其作为参数传递。要设置的变量是 `recursive.file.enumeration`。我们需要将这个变量设置为 `true`, 以便启用递归遍历。

目前，集合数据源要求数据类型和迭代器实现 `Serializable`。此外，集合数据源不能并行执行 (`parallelism = 1`)。

【示例】读取文本文件数据源中的数据。

为了演示这些数据转换函数，我们先准备好示例数据。在项目的 `input` 目录下，新建一个名为 `wc.txt` 的文本文件，并输入以下内容：

```
Good good study
```

```
Day day up
```

实现代码如下。

Scala 代码：

Java 代码：

执行以上程序，输出结果如下所示：

```
(up,1)
```

```
(day,2)
```

```
(good,2)
```

```
(study,1)
```

【示例】读取.csv 文件数据源中的数据。

1) 在项目的 `input` 目录下创建一个 CSV 文件 `pepoles.csv`，并编辑其内容如下：

```
张三,男,23
```

```
李四,女,18
```

```
王老五,男,35
```

2) 代码实现。

Scala 代码：

Java 代码：

3) 执行程序，输出结果如下所示：

```
Person(李四,女,18)
```

```
Person(王老五,男,35)
```

```
Person(张三,男,23)
```

也可以不使用 POJO，直接将读取的每一行转换为一个元组。

【示例】读取.csv 文件数据源中的数据。

Scala 代码：

Java 代码：

执行以上代码，输出结果如下：

```
(张三,男,23)
```

```
(李四,女,18)
```

```
(王老五,男,35)
```


4.2.2 基于集合的数据源

使用 Flink DataSet API，我们还可以从基于 Java 的集合中读取数据。以下是一些我们可以用来读取数据的函数：

- ❑ `fromCollection(Collection)`：这将从基于 java 的集合创建一个数据集。
- ❑ `fromCollection(Iterable)`：从一个迭代创建一个数据集。由 `Iterable` 返回的所有元素必须是相同类型的。
- ❑ `fromCollection(Iterator, Class)`：这将从迭代器创建数据集。迭代器的元素属于 `Class` 类参数给定的类型。
- ❑ `fromElements(elements: _*)`：从给定的对象序列创建一个数据集。所有对象必须具有相同的类型。
- ❑ `fromParallelCollection(SplittableIterator, Class)`：这将并行地从迭代器创建数据集。`Class` 参数表示对象类型。
- ❑ `generateSequence(from, to)`：这将生成给定区间之间的数字序列，以并行方式。

【示例】使用基于集合的数据源示例。

Scala 代码：

```
package com.xueai8.scala.ch04

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

/**
 * 基于集合的数据源
 */
object CollectSourceDemo {

  def main(args: Array[String]): Unit = {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    // env.fromElements("Good good study", "Day day up")
    //     .map(_._toLowerCase)
    //     .print

    // 或者
    val list = List("Good good study", "Day day up")
    env.fromCollection(list).map(_._toLowerCase).print
  }
}
```

Java 代码：

```
package com.xueai8.java.ch04;

import org.apache.flink.api.java.ExecutionEnvironment;

import java.util.ArrayList;
import java.util.Arrays;
```

```
import java.util.List;

/**
 * 集合数据源
 */
public class CollectSourceDemo {
    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // 从内存数据构造数据源
        // env.fromElements("Good good study", "Day day up")
        //         .map(String::toLowerCase)
        //         .print();

        // 或者
        List<String> list = new ArrayList<>(Arrays.asList("Good good study", "Day day up"));
        env.fromCollection(list)
            .map(String::toLowerCase)
            .print();
    }
}
```

执行以上代码，输出结果如下：

```
good good study
day day up
```

4.2.3 通用的数据源

DataSet API 支持两个读取数据的通用函数：

- ❑ `readFile(inputFormat, path)/FileInputFormat`：这将从给定路径创建 `FileInputFormat` 类型的数据集。
- ❑ `createInput(inputFormat)/InputFormat`：这将创建一个通用输入格式的数据集。

4.2.4 压缩文件

Flink 支持在读取带有适当扩展名的文件时对其进行透明解压缩。我们不需要做任何其他配置来读取压缩文件，任何 `FileInputFormat` 都支持压缩，包括自定义输入格式。如果检测到扩展名正确的文件，那么 Flink 会自动解压并将其发送给进一步处理。

这里需要注意的一件事是，压缩文件可能不会并行读取，从而影响作业的可伸缩性。

下表列出了当前支持的压缩方法。

压缩算法	文件扩展名	是否并行
GZip	.gz, .gzip	不并行
DEFLATE	.deflate	不并行
Bzip2	.bz2	不并行
XZ	.xz	不并行

4.3 数据转换

数据转换（data transformation）将数据集从一种形式转换为另一种形式。输入可以是一个或多个数据集，输出也可以是零个、一个或多个数据集。程序可以将多个转换组合成复杂的程序集。

4.3.1 map 转换

Map 转换在数据集的每个元素上应用用户定义的映射函数。它实现了一对一的映射，也就是说，函数必须返回一个元素。这是最简单的转换之一，其中输入是一个数据集，输出也是一个数据集。下面是进行 map 转换的示例代码，将读取的数据都转换为小写。

Scala 代码：

```
package com.xueai8.scala.ch04

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

/**
 * map 转换
 */
object TransformerDemo01 {
  def main(args: Array[String]): Unit = {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    env.fromElements("Good good study", "Day day up")
      .map(_._toLowerCase)
      .print
  }
}
```

Java 代码：

```
package com.xueai8.java.ch04;

import org.apache.flink.api.java.ExecutionEnvironment;

/**
 * map 转换
 */
public class TransformerDemo01 {
  public static void main(String[] args) throws Exception {
    // 设置批处理执行环境
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // map 转换：
    env.fromElements("Good good study", "Day day up")
      .map(String::toLowerCase) // Java 的方法引用
      .print();
  }
}
```

对于简单的 map, flink 可以推测其类型。复杂的则需要指定 return type, 或者构造一个 MapFunction, 或者 extends 自 Tuple2<Integer, Integer>。

执行以上代码, 输出结果如下:

```
good good study
day day up
```

4.3.2 flatMap 转换

FlatMap 转换在数据集的每个元素上应用用户定义的 flat-map 函数。flatMap 接受一条记录并输出零条、一条或多条记录。map 函数的这种变体可以为每个输入元素返回任意多个结果元素(包括 none)。下面是进行 flatMap 转换的示例代码:

Scala 代码:

```
package com.xueai8.scala.ch04

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

/**
 * flatMap 转换
 */
object TransformerDemo02 {
  def main(args: Array[String]): Unit = {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    env.fromElements("Good good study", "Day day up")
      .map(_._toLowerCase)
      .flatMap(_._split("\\W+")) // 相当于先 map, 再 flatten
      .map(_._1)
      .print
  }
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.util.Collector;

/**
 * Created by tongyao on 2019/6/5.
 * flatMap 转换
 */
public class TransformerDemo02 {
  public static void main(String[] args) throws Exception {
    // 设置批处理执行环境
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
```

```
// 首先从环境中获取一些数据，再执行 map 和 flatMap 转换
/*
// 使用匿名内部类
env.fromElements("Good good study","Day day up")
    .map(String::toLowerCase)
    .flatMap(new FlatMapFunction<String, Tuple2<String,Integer>>() {
        @Override
        public void flatMap(String value, Collector<Tuple2<String, Integer>> out)
            throws Exception {
            for(String word : value.split("\\W+")){
                out.collect(new Tuple2<>(word,1));
            }
        }
    })
    .print();
*/

// 或者使用 Lambda 函数
env.fromElements("Good good study","Day day up")
    .map(String::toLowerCase)
    .flatMap((FlatMapFunction<String, Tuple2<String,Integer>>) (value, out) -> {
        for(String word: value.split("\\W+")){
            out.collect(new Tuple2<>(word,1));
        }
    }).returns(Types.TUPLE(Types.STRING, Types.INT))
    .print();
}
```

对于 flatMap 的支持是无法猜测出来类型的，必须通过 returns(Types.STRING) 指定具体的返回值类型。

执行以上代码，输出结果如下所示：

```
(good,1)
(good,1)
(study,1)
(day,1)
(day,1)
(up,1)
```

4.3.3 mapPartition 转换

MapPartition 在单个函数调用中转换并行分区。该函数以 Iterable 流的形式获取分区，并可以生成任意数量的结果值。每个分区中的元素数量取决于并行度和之前的操作。它类似于 Map，但是一次 Map 一整个并行分区。

以下代码将文本行数据集转换为每个分区的计数数据集。

Scala 代码：

```
package com.xueai8.scala.ch04
```

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

/**
 * mapPartition 转换
 */
object TransformerDemo03 {

  def main(args: Array[String]): Unit = {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    val wordList = env.fromElements(1, 2, 3, 4, 5, 6, 7, 8).setParallelism(2)

    val counts = wordList.mapPartition(in => Some(in.size))

    counts.print()
  }
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.api.common.functions.MapPartitionFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.util.Collector;

/**
 * mapPartition 转换
 */
public class TransformerDemo03 {
    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        DataSet<Integer> wordList = env.fromElements(1, 2, 3, 4, 5, 6, 7, 8).setParallelism(2);

        DataSet<Long> counts = wordList.mapPartition(new MapPartitionFunction<Integer, Long>() {
            @Override
            public void mapPartition(Iterable<Integer> values, Collector<Long> out) throws Exception {
                long count = 0;
                for (Integer v : values) {
                    count++;
                }
                out.collect(count);
            }
        });

        counts.print();
    }
}
```

执行以上代码，输出结果如下：

```
4
4
```

注：mapPartition 比 map 更高效，在性能调优中，经常会被建议尽量用 mappartition 操作去替代 map 操作。

4.3.4 filter 转换

Filter 转换在数据集的每个元素上应用一个用户定义的 filter 函数，并仅保留该函数返回 true 的那些元素。filter 函数对条件进行评估，如果结果为 true，则该条数据输出。filter 函数可以输出零个记录。下面是进行 filter 转换的示例代码。

Scala 代码：

```
package com.xueai8.scala.ch04

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

/**
 * filter 转换
 */
object TransformerDemo04 {
  def main(args: Array[String]): Unit = {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    // 得到输入数据，然后执行 filter 转换
    env.fromElements("Good good study", "Day day up")
      .map(_._toLowerCase)
      .filter(_._contains("study"))
      .print()
  }
}
```

Java 代码：

```
package com.xueai8.java.ch04;

import org.apache.flink.api.java.ExecutionEnvironment;

/**
 * filter 转换
 */
public class TransformerDemo04 {
  public static void main(String[] args) throws Exception {
    // 设置批处理执行环境
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // 得到输入数据，然后执行 filter 转换
    env.fromElements("Good good study", "Day day up")
      .map(String::toLowerCase)
      .filter(s -> s.contains("study"))
  }
}
```

```
        .print();  
    }  
}
```

执行以上代码，输出结果如下所示：

```
good good study
```

4.3.5 reduce 转换

Reduce 转换通过缩减当前值和最后一个 reduced 过的值来滚动输出。下面的代码对 DataSet 进行求和。

Scala 代码：

```
package com.xueai8.scala.ch04  
  
import org.apache.flink.api.scala._  
  
/**  
 * reduce 转换  
 */  
object TransformerDemo05 {  
  
    def main(args: Array[String]): Unit = {  
        // 设置批处理执行环境  
        val env = ExecutionEnvironment.getExecutionEnvironment  
  
        // 得到输入数据，reduce 转换  
        val collection1 = env.fromElements(1,5,7,8)  
        collection1.reduce(_+_).print() // 21  
  
        val collection2 = env.fromElements((1,1),(2,2),(3,3))  
        collection2.reduce((a,b) => (a._1 + b._1, a._2 * b._2)).print() // (6,6)  
    }  
}
```

Java 代码：

```
package com.xueai8.java.ch04;  
  
import org.apache.flink.api.common.functions.ReduceFunction;  
import org.apache.flink.api.java.ExecutionEnvironment;  
import org.apache.flink.api.java.operators.DataSource;  
import org.apache.flink.api.java.tuple.Tuple2;  
  
/**  
 * reduce 转换  
 */  
public class TransformerDemo05 {  
    public static void main(String[] args) throws Exception {  
        // 设置批处理执行环境  
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
```



```

// reduce 转换
DataSource<Integer> collection1 = env.fromElements(1,5,7,8);
collection1
    .reduce((ReduceFunction<Integer>) (t1, t2) -> t1+t2)
    .print();           // 21

DataSource<Tuple2<Integer,Integer>> collection2 = env
    .fromElements(new Tuple2<>(1,1),new Tuple2<>(2,2),new Tuple2<>(3,3));
collection2
    .reduce((ReduceFunction<Tuple2<Integer, Integer>>) (t1, t2) ->
        new Tuple2<>(t1.f0 + t2.f0, t1.f1 * t2.f1))
    .print();           // (6,6)
}
}

```

执行以上代码，输出结果如下所示：

```

21
(6,6)

```

4.3.6 在分组数据集上的 reduce 转换

reduce 操作，既可用在分组的 DataSet 上，也可以用在不分组的 DataSet 上。应用于分组 DataSet 的 Reduce 转换使用用户定义的 reduce 函数将每个组减少为单个元素，相当于 SQL 语言中的分组聚合。对于每组输入元素，reduce 函数连续地将元素对组合成一个元素，直到每个组只剩下一个元素。

注意，对于 ReduceFunction，返回对象的 key 字段应与输入值匹配。这是因为 reduce 是可隐式组合（combine）的，并且从 combine 运算符发出的对象在传递给 reduce 运算符时再次按 key 分组。

Scala 代码：

```

package com.xueai8.scala.ch04

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

/**
 * 在分组数据集上的 reduce 转换
 */
object TransformerDemo07 {
    // case class 类
    case class WC(word:String,count:Int)

    def main(args: Array[String]): Unit = {
        // 设置批处理执行环境
        val env = ExecutionEnvironment.getExecutionEnvironment

        // 用法一
        val words = env.fromElements(("good",1),("good",1),("study",1))
        val wordcounts = words.groupBy(_._1).reduce((a,b) => (a._1,a._2+b._2)) // 按元组索引
        // val wordcounts = words.groupBy(0).reduce((a,b) => (a._1,a._2 + b._2)) // 或者按 key 的位置
        wordcounts.print()

        // 用法二
    }
}

```

```
val words2 = env.fromElements(WC("good", 1), WC("good", 1), WC("study", 1))
val wordCounts2 = words2.groupBy(_._word).reduce((wc1,wc2) => WC(wc1.word,wc1.count + wc2.count))
// val wordCounts2 = words2.groupBy(0).reduce((wc1,wc2) => WC(wc1.word,wc1.count + wc2.count))

wordCounts2.print()
}
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.api.common.functions.ReduceFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;

/**
 * 在分组数据集上的 reduce 转换
 */
public class TransformerDemo07 {
    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // 在分组数据集上的 reduce 转换
        // 用法一
        Tuple2<String, Integer> word01 = new Tuple2<>("good",1);
        Tuple2<String, Integer> word02 = new Tuple2<>("good",1);
        Tuple2<String, Integer> word03 = new Tuple2<>("study",1);

        DataSet<Tuple2<String, Integer>> words1 = env.fromElements(word01,word02,word03);
        DataSet<Tuple2<String, Integer>> wordCounts1 = words1
            .groupBy(0)
            .reduce((ReduceFunction<Tuple2<String, Integer>>) (t1, t2) -> new Tuple2<>(t1.f0,t1.f1 +
t2.f1));

        wordCounts1.print();

        // 用法二
        DataSet<WC> words2 = env.fromElements(new WC("good",1),new WC("good",1),new WC("study",1));
        DataSet<WC> wordCounts2 = words2
            .groupBy(wc -> wc.word)
            .reduce((ReduceFunction<WC>) (wc1, wc2) -> new WC(wc1.word, wc1.count + wc2.count));

        wordCounts2.print();
    }

    public static class WC {
        String word;
        int count;
    }
}
```

```
WC(String word, int count) {  
    this.word = word;  
    this.count = count;  
}  
  
@Override  
public String toString() {  
    return "WC(" + word + "," + count + ")";  
}  
}
```

执行以上代码，输出结果如下所示：

```
(good,2)  
(study,1)  
  
WC(good,2)  
WC(study,1)
```

4.3.7 在分组数据集上的 GroupReduce

将一组元素组合成一个或多个元素。ReduceGroup 可以应用于完整的数据集，也可以应用于分组的数据集。

在按字段位置 key 分组的数据集上执行 GroupReduce（仅适用于 Tuple DataSet）

下面的代码展示了如何从按整数分组的数据集中删除重复的字符串。

Scala 代码：

```
package com.xueai8.scala.ch04  
  
import org.apache.flink.api.scala._  
import org.apache.flink.util.Collector  
  
import scala.collection.mutable  
  
/**  
 * reduceGroup 函数  
 * 下面的代码展示了如何从按整数分组的数据集中删除重复的字符串。  
 */  
object TransformerDemo16 {  
    def main(args: Array[String]): Unit = {  
        // 设置批处理执行环境  
        val env = ExecutionEnvironment.getExecutionEnvironment  
  
        // 聚合操作  
        val ds = env.fromElements(  
            ("河南", "郑州市"),  
            ("河北", "石家庄市"),  
            ("河南", "开封市"),  
            ("河南", "开封市"),  
            ("河北", "邯郸市"),
```

```

        ("河北", "邯郸市")
    )

    val output = ds.groupBy(0)
        .reduceGroup( (in:Iterator[(String, String)], out:Collector[(String, String)]) => {
            val city: mutable.Set[String] = mutable.Set[String]()
            var key: String = null

            for(t <- in){
                key = t._1
                city.add(t._2)
            }

            // 发送给下游
            out.collect((key, "[" + city.toList.mkString(",") + "]"))
        })

    output.print()
}

```

Java 代码:

```

package com.xueai8.java.ch04;

import org.apache.flink.api.common.functions.GroupReduceFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.util.Collector;

import java.util.HashSet;
import java.util.Set;

/**
 * reduceGroup 函数
 * 下面的代码展示了如何从按整数分组的数据集中删除重复的字符串。
 */
public class TransformerDemo16 {
    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // 构造 DataSet
        Tuple2<String, String> word01 = new Tuple2<>("河南", "郑州市");
        Tuple2<String, String> word02 = new Tuple2<>("河北", "石家庄市");
        Tuple2<String, String> word03 = new Tuple2<>("河南", "开封市");
        Tuple2<String, String> word04 = new Tuple2<>("河南", "开封市");
        Tuple2<String, String> word05 = new Tuple2<>("河北", "邯郸市");
        Tuple2<String, String> word06 = new Tuple2<>("河北", "邯郸市");

        DataSet<Tuple2<String, String>> ds =
    
```

```
env.fromElements(word01,word02,word03,word04,word05,word06);

// 转换操作
DataSet
```

执行以上代码，输出结果如下所示：

```
(河南,[开封市, 郑州市])
(河北,[邯郸市, 石家庄市])
```

4.3.8 在分组数据集上的 GroupCombine

GroupCombine 变换是可组合的 GroupReduceFunction 中 combine 步骤的泛化形式。它允许将输入类型 I 与任意输出类型 O 相结合。而 GroupReduce 中的 combine 步骤只允许从输入类型 I 到输出类型 I 的组合。

在某些应用程序中，在执行其他转换（例如，减少数据大小）之前，最好将数据集组合成中间格式。这可以通过一个成本非常低的 CombineGroup 转换来实现。

下面的示例演示如何将 GroupCombine 转换用于另一种 WordCount 实现。

Scala 代码：

```
package com.xueai8.scala.ch04

import org.apache.flink.api.scala._
import org.apache.flink.util.Collector

import scala.collection.mutable

/**
 * combineGroup 转换
```

```
*
* 单词计数的另一种实现
*/
object TransformerDemo17 {

  def main(args: Array[String]): Unit = {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    val combinedWords = env
      .fromElements("Good good study", "Day day up")
      .map(_._toLowerCase)
      .flatMap(_._split("\\W+")) // 相当于先 map，再 flatten
      .groupByKey((s: String) => s)
      // combine
      .combineGroup((in: Iterator[String], out: Collector[(String, Int)]) => {
        var key: String = null
        var count: Int = 0

        for (word <- in) {
          key = word
          count += 1
        }
        // 将带有 word 和 count 的元组发送给下游
        out.collect((key, count))
      })

    val output = combinedWords
      .groupByKey()
      .reduceGroup((in: Iterator[(String, Int)], out: Collector[(String, Int)]) => {
        var key: String = null
        var count: Int = 0

        for (t <- in) {
          key = t._1
          count += t._2
        }

        // 发送给下游
        out.collect((key, count))
      })

    output.print()
  }
}
```

Java 代码:

```
package com.xueai8.java.ch04;
```

```
import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.common.functions.GroupCombineFunction;
```

```
import org.apache.flink.api.common.functions.GroupReduceFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.util.Collector;

/**
 * combineGroup 转换
 *
 * 单词计数的另一种实现
 */
public class TransformerDemo17 {
    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // map 转换:
        DataSet<String> input = env.fromElements("Good good study day", "Day day up study");

        DataSet<Tuple2<String, Integer>> combinedWords = input
            .map(String::toLowerCase)
            .flatMap(new FlatMapFunction<String, String>() {
                @Override
                public void flatMap(String value, Collector<String> out) throws Exception {
                    for (String word : value.split("\\W+")) {
                        out.collect(word);
                    }
                }
            })
            .groupBy(s -> s) // 分组相同的单词
            // combine
            .combineGroup(new GroupCombineFunction<String, Tuple2<String, Integer>>() {
                @Override
                public void combine(Iterable<String> words, Collector<Tuple2<String, Integer>> out) {
                    String key = null;
                    int count = 0;

                    for (String word : words) {
                        key = word;
                        count++;
                    }
                    // 将带有 word 和 count 的元组发送给下游
                    out.collect(new Tuple2(key, count));
                }
            });

        DataSet<Tuple2<String, Integer>> output = combinedWords
            .groupBy(0) // 再次按词分组
            // 具有全部数据交换的 group reduce
            .reduceGroup(new GroupReduceFunction<Tuple2<String, Integer>, Tuple2<String, Integer>>() {
```

```
@Override
public void reduce(Iterable<Tuple2<String, Integer>> iterable,
                  Collector<Tuple2<String, Integer>> collector) throws Exception {
    String key = null;
    int count = 0;

    for (Tuple2<String, Integer> word : iterable) {
        key = word.f0;
        count += word.f1;
    }
    // 将带有 word 和 count 的元组发送给下游
    collector.collect(new Tuple2(key, count));
}

});

output.print();
}
```

执行以上代码，输出结果如下所示：

```
(up,1)
(day,3)
(good,2)
(study,2)
```

4.3.9 在分组元组数据集上执行聚合

聚合转换非常常见。聚合转换提供了如下的内置聚合函数：sum、min 和 max。聚合转换只能应用于元组数据集，并且只支持用于分组的字段位置键。

下面的代码展示了如何对按字段位置键分组的数据集应用聚合转换。

Scala 代码：

```
package com.xueai8.scala.ch04

import org.apache.flink.api.java.aggregation.Aggregations.{MIN, SUM}
import org.apache.flink.api.scala._

/**
 * 分组聚合
 */
object TransformerDemo08 {
    def main(args: Array[String]): Unit = {
        // 设置批处理执行环境
        val env = ExecutionEnvironment.getExecutionEnvironment

        // 聚合操作
        val ds = env.fromElements(
            (1,"财务",12000.00),
            (1,"财务",22000.00),
            (1,"市场",18000.00)
```



```
)  
val output = ds.groupBy(1)           // 按第二个字段分组  
                .aggregate(SUM, 0)    // 对第一个字段求和  
                .and(MIN, 2)          // 计算第三个字段的最小值  
  
output.print()  
}
```

Java 代码:

```
package com.xueai8.java.ch04;  
  
import org.apache.flink.api.java.DataSet;  
import org.apache.flink.api.java.ExecutionEnvironment;  
import org.apache.flink.api.java.tuple.Tuple3;  
  
import static org.apache.flink.api.java.aggregation.Aggregations.MIN;  
import static org.apache.flink.api.java.aggregation.Aggregations.SUM;  
  
/**  
 * 分组聚合  
 */  
public class TransformerDemo08 {  
    public static void main(String[] args) throws Exception {  
        // 设置批处理执行环境  
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();  
  
        // 分组聚合操作  
        Tuple3<Integer, String, Double> user01 = new Tuple3<>(1, "财务", 12000.00);  
        Tuple3<Integer, String, Double> user02 = new Tuple3<>(1, "财务", 22000.00);  
        Tuple3<Integer, String, Double> user03 = new Tuple3<>(1, "市场", 18000.00);  
  
        DataSet<Tuple3<Integer, String, Double>> ds = env.fromElements(user01, user02, user03);  
        DataSet<Tuple3<Integer, String, Double>> output = ds  
            .groupBy(1)           // 按第二个字段分组  
            .aggregate(SUM, 0)    // 对第一个字段求和  
            .and(MIN, 2);          // 计算第三个字段的最小值  
  
        output.print();  
    }  
}
```

执行以上代码，输出结果如下所示：

```
(1,市场,18000.0)  
(2,财务,12000.0)
```

要在一个数据集上应用多个聚合，必须在第一次聚合之后使用`.and()`函数，这意味着`.aggregate(SUM, 0).and(MIN, 2)`生成原始数据集的字段 0 的和以及字段 2 的最小值。与此相对照，`.aggregate(SUM, 0).aggregate(MIN, 2)`将对聚合应用聚合。在给定的例子中，它将在计算按字段 1 分组字段 0 的和后产生字段 2 的最小值。

4.3.10 在分组元组数据集上执行 minBy

MinBy 转换为每组元组选择一个元组。所选的元组是一个或多个指定字段值最小的元组。用于比较的字段必须是有效的 key 字段，即具有可比性。如果多个元组具有最小字段值，则返回这些元组的任意元组。

下面的代码演示了如何从数据集<Tuple3<Integer, String, Double>>>中为每组具有相同字符串值的元组选择 Integer 和 Double 字段的最小值的元组：

Scala 代码：

```
val input: DataSet[(Int, String, Double)] = // [...]
val output: DataSet[(Int, String, Double)] = input
    .groupBy(1) // group DataSet on second field
    .minBy(0, 2) // select tuple with minimum values for first and third field.
```

Java 代码：

```
DataSet<Tuple3<Integer, String, Double>> input = // [...]
DataSet<Tuple3<Integer, String, Double>> output = input
    .groupBy(1) // group DataSet on second field
    .minBy(0, 2); // select tuple with minimum values for first and third field.
```

Scala 代码：

```
package com.xueai8.scala.ch04

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

/**
 * minBy
 */
object TransformerDemo09 {

    def main(args: Array[String]): Unit = {
        // 设置批处理执行环境
        val env = ExecutionEnvironment.getExecutionEnvironment

        // 聚合操作
        val ds = env.fromElements((2, "财务", 12000.00), (1, "财务", 22000.00), (1, "财务", 18000.00))
        val output = ds
            .groupBy(1) // 按第二个字段分组
            .minBy(0, 2) // 选择第 1 列和第 3 列最小值所在的元组

        output.print()
    }
}
```

Java 代码：

```
package com.xueai8.java.ch04;

import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple3;
```

```
/**
 * minBy
 */
public class TransformerDemo09 {
    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // minBy
        Tuple3<Integer, String, Double> user01 = new Tuple3<>(2, "财务", 12000.00);
        Tuple3<Integer, String, Double> user02 = new Tuple3<>(1, "财务", 22000.00);
        Tuple3<Integer, String, Double> user03 = new Tuple3<>(1, "财务", 18000.00);

        DataSet<Tuple3<Integer, String, Double>> ds = env.fromElements(user01, user02, user03);
        DataSet<Tuple3<Integer, String, Double>> output = ds
            .groupBy(1)           // 按第二个字段分组
            .minBy(0, 2);         // 选择第 1 列和第 3 列最小值所在的元组

        output.print();
    }
}
```

执行以上代码，输出结果如下所示：

(1,财务,18000.0)

4.3.11 在分组元组数据集上执行 maxBy

MaxBy 转换为每组元组选择一个元组。所选的元组是一个或多个指定字段值最大值的元组。用于比较的字段必须是有效的 key 字段，即具有可比性。如果多个元组具有最大字段值，则返回这些元组的任意元组。

Scala 代码：

```
package com.xueai8.scala.ch04
```

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

```
object TransformerDemo10 {
```

```
    def main(args: Array[String]) {
```

```
        // 设置批处理执行环境
```

```
        val env = ExecutionEnvironment.getExecutionEnvironment
```

```
        // 聚合操作
```

```
        val ds = env.fromElements((2, "财务", 12000.00), (2, "财务", 22000.00), (1, "财务", 18000.00))
```

```
        val output = ds.groupBy(1)           // 按第二个字段分组
```

```
            .maxBy(0, 2)         // 选择第 1 列和第 3 列最大值所在的元组
```

```
        output.print()
```

```
    }
```

```
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple3;

import static org.apache.flink.api.java.aggregation.Aggregations.MIN;
import static org.apache.flink.api.java.aggregation.Aggregations.SUM;

public class TransformerDemo10{

    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // minBy
        Tuple3<Integer, String, Double> user01 = new Tuple3<>(2,"财务",12000.00);
        Tuple3<Integer, String, Double> user02 = new Tuple3<>(2,"财务",22000.00);
        Tuple3<Integer, String, Double> user03 = new Tuple3<>(1,"财务",18000.00);

        DataSet<Tuple3<Integer, String, Double>> ds = env.fromElements(user01,user02,user03);
        DataSet<Tuple3<Integer, String, Double>> output = ds
            .groupBy(1)          // 按第二个字段分组
            .maxBy(0, 2);        // 选择第 1 列和第 3 列最大值所在的元组
        output.print();
    }
}
```

执行以上代码，输出结果如下所示：

```
(2,财务,22000.0)
```

4.3.12 在全部元组数据集上执行聚合操作

有一些常用的聚合操作。聚合转换提供以下内置聚合功能：`max`、`min` 和 `sum`。聚合转换只能应用于元组数据集。

下面的代码展示了如何在一个完整的数据集上应用聚合转换。

Scala 代码:

```
package com.xueai8.scala.ch04

import org.apache.flink.api.java.aggregation.Aggregations.{MIN, SUM}
import org.apache.flink.api.scala._

object TransformerDemo11 {

    def main(args: Array[String]) {
        // 设置批处理执行环境
        val env = ExecutionEnvironment.getExecutionEnvironment
```

```
// 聚合操作
val ds = env.fromElements((1,"财务",12000.00),(1,"财务",22000.00),(1,"市场",18000.00))
val output = ds.aggregate(SUM, 0)          // 对第一个字段求和
                .and(MIN, 2)                // 计算第三个字段的最小值

output.print()
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple3;

import static org.apache.flink.api.java.aggregation.Aggregations.MIN;
import static org.apache.flink.api.java.aggregation.Aggregations.SUM;

public class TransformerDemo11 {

    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // 在全部数据集上执行聚合操作
        Tuple3<Integer, String, Double> user01 = new Tuple3<>(1,"财务",12000.00);
        Tuple3<Integer, String, Double> user02 = new Tuple3<>(1,"财务",22000.00);
        Tuple3<Integer, String, Double> user03 = new Tuple3<>(1,"市场",18000.00);

        DataSet<Tuple3<Integer, String, Double>> ds = env.fromElements(user01,user02,user03);
        DataSet<Tuple3<Integer, String, Double>> output = ds
                .aggregate(SUM, 0)          // 对第一个字段求和
                .and(MIN, 2);              // 计算第三个字段的最小值

        output.print();
    }
}
```

执行以上代码，输出结果如下所示：

(3,市场,12000.0)

4.3.13 distinct 转换

distinct 转换用来从数据集中删除所有重复的元素。下面是进行 distinct 转换的示例代码。

Scala 代码:

```
package com.xueai8.scala.ch04

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

```
object TransformerDemo12 {  
  
  def main(args: Array[String]) {  
    // 设置批处理执行环境  
    val env = ExecutionEnvironment.getExecutionEnvironment  
  
    // 去重操作  
    val ds = env.fromElements((1, "财务", 12000.00), (1, "财务", 12000.00), (1, "市场", 18000.00))  
    val output = ds.distinct  
  
    output.print()  
  }  
}
```

Java 代码:

```
package com.xueai8.java.ch04;  
  
import org.apache.flink.api.java.DataSet;  
import org.apache.flink.api.java.ExecutionEnvironment;  
import org.apache.flink.api.java.tuple.Tuple3;  
  
import static org.apache.flink.api.java.aggregation.Aggregations.MIN;  
import static org.apache.flink.api.java.aggregation.Aggregations.SUM;  
  
public class TransformerDemo12 {  
  
  public static void main(String[] args) throws Exception {  
    // 设置批处理执行环境  
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();  
  
    // 在全部数据集上执行聚合操作  
    Tuple3<Integer, String, Double> user01 = new Tuple3<>(1, "财务", 12000.00);  
    Tuple3<Integer, String, Double> user02 = new Tuple3<>(1, "财务", 12000.00);  
    Tuple3<Integer, String, Double> user03 = new Tuple3<>(1, "市场", 18000.00);  
  
    DataSet<Tuple3<Integer, String, Double>> ds = env.fromElements(user01, user02, user03);  
    DataSet<Tuple3<Integer, String, Double>> output = ds.distinct();  
  
    output.print();  
  }  
}
```

执行以上代码，输出结果如下所示：

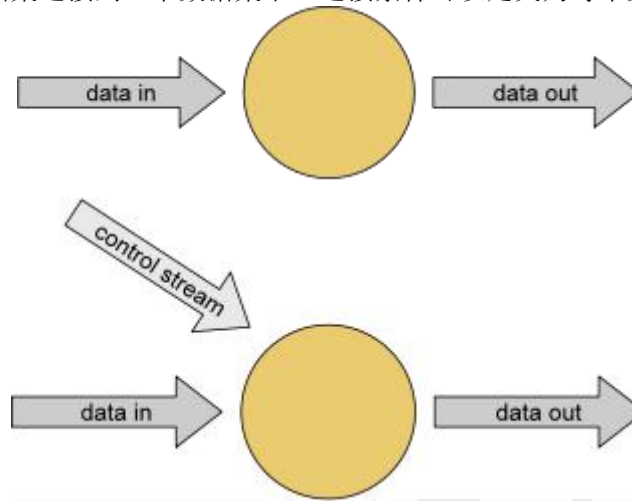
```
(1,市场,18000.0)  
(1,财务,12000.0)
```

也可以改变数据集中元素的区別是如何决定的，使用：

- ☐ 一个或多个字段位置 key(仅为元组数据集)，
- ☐ key 选择器函数，或
- ☐ 一个关键表达式。

4.3.14 join 连接转换

连接转换将两个数据集连接到一个数据集中。连接条件可以定义为每个数据集中的 **key**。



两个数据集的元素在一个或多个 **key** 上进行连接，这些 **key** 可以这样指定：

- ☐ 一个 **key** 表达式
- ☐ 一个 **key-selector** 函数
- ☐ 一个或多个字段位置 **key**(仅限元组数据集)
- ☐ **Case Class** 字段

下面是进行 **join** 转换的示例代码。

Scala 代码：

```
package com.xueai8.scala.ch04

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

/**
 * join 连接
 */
object TransformerDemo13 {

  def main(args: Array[String]) {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    // join 连接
    val employees = env.fromElements(("张三",1),("李四",2))
    val depts = env.fromElements(("市场部",1),("财务部",3))

    val result = employees.join(depts).where(1).equalTo(1)
    result.print()
  }
}
```

Java 代码：

```
package com.xueai8.java.ch04;

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

/**
 * join 连接
 */
public class TransformerDemo13 {

    public static class Employee { public String empName; public int deptId; }
    public static class Dept { public String deptName; public int id; }

    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // 构造员工数据集
        Employee emp01 = new Employee();
        emp01.empName = "张三";
        emp01.deptId = 1;

        Employee emp02 = new Employee();
        emp02.empName = "李四";
        emp02.deptId = 2;

        DataSet<Employee> employees = env.fromElements(emp01, emp02);

        // 构造部门数据集
        Dept dept01 = new Dept();
        dept01.deptName = "市场部";
        dept01.id = 1;

        Dept dept02 = new Dept();
        dept02.deptName = "财务部";
        dept02.id = 3;

        DataSet<Dept> depts = env.fromElements(dept01, dept02);

        // 两个数据集执行 join 连接
        DataSet<Tuple2<Employee, Dept>> result = employees.join(depts)
            .where("deptId")
            .equalTo("id");

        result.map(new MapFunction<Tuple2<Employee, Dept>, String>() {
            @Override
            public String map(Tuple2<Employee, Dept> t) throws Exception {
                return t.f0.empName + "-" + t.f0.deptId + "-" + t.f1.id + "-" + t.f1.deptName;
            }
        })
```



```
    }  
    }).print();  
  }  
}
```

执行以上代码，输出结果如下所示：

张三-1-1-市场部

4.3.15 union 转换

Union 函数执行两个或多个数据集的联合。这将并行地组合数据集。如果我们将一个集与它自己组合起来，那么它将输出每个记录两次。生成两个必须具有相同类型的数据集的并集。一个超过两个数据集的 union 可以通过多个 union 调用来实现，如下所示：

Scala 代码：

```
val vals1: DataSet[(String, Int)] = // [...]  
val vals2: DataSet[(String, Int)] = // [...]  
val vals3: DataSet[(String, Int)] = // [...]  
  
val unioned = vals1.union(vals2).union(vals3)
```

Java 代码：

```
DataSet<Tuple2<String, Integer>> vals1 = // [...]  
DataSet<Tuple2<String, Integer>> vals2 = // [...]  
DataSet<Tuple2<String, Integer>> vals3 = // [...]  
DataSet<Tuple2<String, Integer>> unioned = vals1.union(vals2).union(vals3);
```

下面是一个使用 union 转换运算的示例。

Scala 代码：

```
package com.xueai8.scala.ch04  
  
import org.apache.flink.api.scala._  
  
object TransformerDemo18 {  
  
  def main(args: Array[String]) {  
    // 设置批处理执行环境  
    val env = ExecutionEnvironment.getExecutionEnvironment  
  
    // union 合并  
    val ds1 = env.fromElements("good good study")  
                  .flatMap(_._toLowerCase.split("\\W+"))  
                  .map(_._, 1)  
  
    val ds2 = env.fromElements("day day up")  
                  .flatMap(_._toLowerCase.split("\\W+"))  
                  .map(_._, 1)  
  
    ds1.union(ds2).print()  
  }  
}
```

```
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.util.Collector;

/**
 * union 转换
 */
public class TransformerDemo18 {
    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // union 转换
        DataSet<Tuple2<String,Integer>> ds1 = env
            .fromElements("good good study")
            .flatMap(new FlatMapFunction<String, Tuple2<String,Integer>>() {
                @Override
                public void flatMap(String value, Collector<Tuple2<String,Integer>> out)
                    throws Exception {
                    for(String word: value.split("\\W+")){
                        out.collect(new Tuple2<>(word, 1));
                    }
                }
            });

        DataSet<Tuple2<String,Integer>> ds2 = env
            .fromElements("day day up")
            .flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {
                @Override
                public void flatMap(String value, Collector<Tuple2<String,Integer>> out)
                    throws Exception {
                    for(String word: value.split("\\W+")){
                        out.collect(new Tuple2<>(word, 1));
                    }
                }
            });

        // 合并
        DataSet<Tuple2<String, Integer>> ds1_and_ds2 = ds1.union(ds2);

        // 输出
        ds1_and_ds2.print();
    }
}
```

```
}
```

执行以上代码，输出结果如下所示：

```
(good,1)
(day,1)
(good,1)
(day,1)
(study,1)
(up,1)
```

4.3.16 project 转换

Project 转换删除或移动元组数据集的元组字段。project(int...)方法选择应该由其索引保留的元组字段，并在输出元组中定义它们的顺序。使用 project 转换将一个元组的元素移除或移动到另一个元组中。这可以用来对特定的元素进行选择处理（相当于 SQL 语句中的投影概念）。

下面是进行 project 转换的示例代码。

Scala 代码：

在 Scala 中，不支持这种转换。

Java 代码：

```
package com.xueai8.java.ch04;

import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple3;

/**
 * project 转换
 */
public class TransformerDemo14 {

    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // 构造数据集
        Tuple3<Integer, String, Double> user01 = new Tuple3<>(1,"张三",12000.00);
        Tuple3<Integer, String, Double> user02 = new Tuple3<>(2,"李四",22000.00);
        Tuple3<Integer, String, Double> user03 = new Tuple3<>(3,"王老五",18000.00);
        DataSet<Tuple3<Integer, String, Double>> ds = env.fromElements(user01,user02,user03);

        // project 转换
        DataSet<Tuple3<Integer, String, Double>> ds_select = ds.project(1,2);
        ds_select.print();
    }
}
```

执行以上代码，输出结果如下：

```
(张三,12000.0)
(李四,22000.0)
```

(王老五,18000.0)

4.3.17 first-n 转换

这个转换任意返回数据集的前 n 个元素。返回数据集的前 n 个(任意)元素。First-n 可应用于常规数据集、分组数据集或分组排序数据集。分组键可以指定为键选择器函数或字段位置键。

Scala 代码:

```
package com.xueai8.scala.ch04

import org.apache.flink.api.scala.ExecutionEnvironment

/**
 * first-n 转换
 */
object TransformerDemo15 {

  def main(args: Array[String]): Unit = {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    // first-n 转换:
    env.fromElements("All", "ours", "love", "Flink")
      .map(_._toLowerCase)      // 转换为小写
      .first(3)                // 返回数据集中前 3 个元素
      .print
  }
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.api.java.ExecutionEnvironment;

/**
 * project 转换
 */
public class TransformerDemo15 {

  public static void main(String[] args) throws Exception {
    // 设置批处理执行环境
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // first-n 转换:
    env.fromElements("All","ours","love","Flink")
      .map(String::toLowerCase) // Java 的方法引用
      .first(3)                // 返回数据集中前 3 个元素
      .print();
  }
}
```

```
}
```

执行以上代码，输出结果如下：

```
all  
ours  
love
```

4.4 数据接收器

在完成数据转换之后，我们需要将结果保存到某个地方。数据接收器（sink）消费数据集并用于存储或返回它们。

4.4.1 计算结果保存到文本文件

下面是 Flink DataSet Java API 提供的一些保存结果的选项：

- ❑ **writeAsText()/TextOutputFormat**：将元素按行写入为字符串。这些字符串是通过调用每个元素的 `toString()` 方法获得的。
- ❑ **writeAsFormattedText() / TextOutputFormat**：将元素按行写成字符串。通过为每个元素调用用户定义的 `format()` 方法来获得字符串。
- ❑ **writeAsCsv(...) / CsvOutputFormat**：将元组写入逗号分隔的值文件。行和字段分隔符是可配置的。每个字段的值来自对象的 `toString()` 方法。
- ❑ **print() / printToErr() / print(String msg) / printToErr(String msg)**：在标准输出/标准错误流中打印每个元素的 `toString()` 值。可选地，可以提供前缀(`msg`)作为输出的前缀。这有助于区分不同的打印调用。如果并行度大于 1，输出也将以产生输出的任务的标识符作为前缀。
- ❑ **write() / FileOutputFormat**：自定义文件输出的方法和基类。支持自定义对象到字节的转换。
- ❑ **output()/ OutputFormat**：大多数通用输出方法，用于非基于文件的数据接收器(如将结果存储在数据库中)。

【示例】执行单词计数，并将批处理结果写入到文本文件中。

Scala 代码：

```
package com.xueai8.scala.ch04  
  
import org.apache.flink.api.java.aggregation.Aggregations.SUM  
import org.apache.flink.api.scala._  
  
/**  
 * 单词计数，并将批处理结果写入到文本文件中  
 */  
object DataSinkDemo01 {  
  
  def main(args: Array[String]) {  
    // 设置批处理执行环境  
    val env = ExecutionEnvironment.getExecutionEnvironment  
  
    // 获得数据，执行 map 和 flatMap 转换
```

```
// 获得数据，执行 map 和 flatMap 转换
env
    .fromElements("Good good study", "Day day up")
    .map(_._toLowerCase)
    .flatMap(_._split("\\W+"))
    .map(_._1)
    .groupBy(0)
    .aggregate(SUM,1)
    .setParallelism(1)
    .writeAsText("output/word-output");

// 注意：这时一定要有 execute 方法，来触发写出操作执行
env.execute("write to txt")
}
```

Java 代码：

```
package com.xueai8.java.ch04;

import org.apache.flink.api.common.functions.FlatMapFunction;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.aggregation.Aggregations;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.util.Collector;

/**
 * 单词计数，并将批处理结果写入到文本文件中
 */
public class DataSinkDemo01 {

    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // 获得数据，执行 map 和 flatMap 转换
        env.fromElements("Good good study","Day day up")
            .map(String::toLowerCase)
            .flatMap(new FlatMapFunction<String, Tuple2<String,Integer>>() {
                @Override
                public void flatMap(String s, Collector<Tuple2<String,Integer>> collector)
                    throws Exception {
                    for(String word : s.split("\\W+")){
                        collector.collect(new Tuple2<>(word,1));
                    }
                }
            })
            .groupBy(0)
            .aggregate(Aggregations.SUM,1)
            .setParallelism(1)
            .writeAsText("output/word-output");
    }
}
```

```
// 注意：这时一定要有 execute 方法，来触发写出操作执行
env.execute("write to txt");
}
}
```

注意，当要将批处理结果写出到文件中时，一定要调用 `env.execute()` 方法，来触发写操作执行。上面的程序执行以后，可以在项目根目录下看到生成的结果文件 `result.txt`。打开它，可以看到如下的结果：

```
(up,1)
(day,2)
(good,2)
(study,1)
```

4.4.2 计算结果保存到 JDBC

也可以使用自定义格式。例如，将单词计数统计结果保存到 MySQL 数据库中。

首先，在项目的 `pom.xml` 中添加如下依赖：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-jdbc_2.12</artifactId>
  <version>1.13.2</version>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.46</version>
</dependency>
```

在 MySQL 数据库中，执行如下 SQL 语句，创建表 `wc` 用来接收 Flink 单词计数结果：

```
mysql> create table wc(word varchar(20), cnt int);
```

然后使用 Flink 自带的 `JDBCOutputFormat` 来写入 MySQL。

Scala 代码：

无实现。

Java 代码：

执行以上代码，然后在 MySQL 数据库中执行查询，查看写入结果。如下 图所示：

```
MariaDB [xueai8]> select * from wc;
+-----+-----+
| word  | cnt  |
+-----+-----+
| good  | 2    |
| study | 1    |
| up    | 1    |
| day   | 2    |
+-----+-----+
4 rows in set (0.01 sec)
```

4.4.3 标准 DataSink 方法

下面是标准 DataSink 方法模板代码。

Scala 代码:

```
// 文本数据
val textData: DataSet[String] = // [...]

// 将 DataSet 写入本地文件系统上的文件
textData.writeAsText("file:///my/result/on/localFS")

// 将 DataSet 写入 HDFS 上的文件
textData.writeAsText("hdfs://nnHost:nnPort/my/result/on/localFS")

// 将 DataSet 写入文件, 如果文件存在, 则重写该文件
textData.writeAsText("file:///my/result/on/localFS", WriteMode.OVERWRITE)

// 元组作为行, 管道符号为分隔符 "a|b|c"
val values: DataSet[(String, Int, Double)] = // [...]
values.writeAsCsv("file:///path/to/the/result/file", "\n", "|")

// 将元组以文本格式 "(a, b, c)" 写入, 而不是作为 CSV 行
values.writeAsText("file:///path/to/the/result/file")

// 使用用户定义的格式将值写入字符串
values map { tuple => tuple._1 + " - " + tuple._2 } .writeAsText("file:///path/to/the/result/file")
```

Java 代码:

```
// 文本数据
DataSet<String> textData = // [...]

// 将 DataSet 写入本地文件系统上的文件
textData.writeAsText("file:///my/result/on/localFS");

// 将 DataSet 写入 HDFS 上的文件
textData.writeAsText("hdfs://nnHost:nnPort/my/result/on/localFS");

// 将 DataSet 写入文件, 如果文件存在, 则重写该文件
textData.writeAsText("file:///my/result/on/localFS", WriteMode.OVERWRITE);

// 元组作为行, 管道符号为分隔符 "a|b|c"
DataSet<Tuple3<String, Integer, Double>> values = // [...]
values.writeAsCsv("file:///path/to/the/result/file", "\n", "|");

// 将元组以文本格式 "(a, b, c)" 写入, 而不是作为 CSV 行
values.writeAsText("file:///path/to/the/result/file");

// 使用用户定义的 TextFormatter 对象将值作为字符串写入
values.writeAsFormattedText("file:///path/to/the/result/file",
    new TextFormatter<Tuple2<Integer, Integer>>() {
        public String format (Tuple2<Integer, Integer> value) {
```



```

        return value.f1 + " - " + value.f0;
    }
});

// 使用自定义输出格式
DataSet<Tuple3<String, Integer, Double>> myResult = [...]

// 将元组数据集写入关系数据库
myResult.output(
    // 构建和配置 OutputFormat
    JDBCOutputFormat.buildJDBCOutputFormat()
        .setDrivername("org.apache.derby.jdbc.EmbeddedDriver")
        .setDBUrl("jdbc:derby:memory:persons")
        .setQuery("insert into persons (name, age, height) values (?, ?, ?)")
        .finish()
);

```

4.4.4 本地排序输出

数据接收器的输出可以使用元组字段位置或字段表达式对指定顺序中的指定字段进行本地排序。这适用于每种输出格式。下面的例子展示了如何使用这个功能。

Scala 代码:

```

val tData: DataSet[(Int, String, Double)] = // [...]
val pData: DataSet[(BookPojo, Double)] = // [...]
val sData: DataSet[String] = // [...]

// 对 String 字段的输出按升序排序
tData.sortPartition(1, Order.ASCENDING).print()

// 对 Double 字段按降序排序, Int 字段按升序排序
tData.sortPartition(2, Order.DESCENTING).sortPartition(0, Order.ASCENDING).print()

// 对嵌套 BookPojo 的 "author" 字段的输出按降序排序
pData.sortPartition("_1.author", Order.DESCENTING).writeAsText(...)

// 按升序对完整元组的输出进行排序
tData.sortPartition("_", Order.ASCENDING).writeAsCsv(...)

// 对原子类型(String)输出按降序排序
sData.sortPartition("_", Order.DESCENTING).writeAsText(...)

```

Java 代码:

```

DataSet<Tuple3<Integer, String, Double>> tData = // [...]
DataSet<Tuple2<BookPojo, Double>> pData = // [...]
DataSet<String> sData = // [...]

// 对 String 字段的输出按升序排序
tData.sortPartition(1, Order.ASCENDING).print();

// 对 Double 字段按降序排序, Int 字段按升序排序

```

```
tData.sortPartition(2, Order.DESCEENDING).sortPartition(0, Order.ASCENDING).print();

// 对嵌套 BookPojo 的"author"字段的输出按降序排序
pData.sortPartition("f0.author", Order.DESCEENDING).writeAsText(...);

// 按升序对完整元组的输出进行排序
tData.sortPartition("", Order.ASCENDING).writeAsCsv(...);

// 对原子类型(String)输出按降序排序
sData.sortPartition("", Order.DESCEENDING).writeAsText(...);
```

注：目前还不支持全局排序的输出。

4.5 广播变量

广播变量是分布式计算框架中常用的一种数据共享方法。它的主要功能是通过网络传输小数据集，在每台机器上保持一个只读缓存变量，计算节点的实例可以直接读取本地内存中的广播数据集，以避免在数据计算过程中多次从其他节点读取小数据集，从而提高整个任务的计算效率。

广播变量可以理解为一个公共共享变量，它可以广播数据集，使不同的任务可以读取数据，每个节点上只保存一份广播数据。如果不使用广播变量，则会在任务中在每个节点上复制数据集的副本，从而导致内存浪费。

广播数据集通过 `withBroadcastSet(DataSet, String)` 按名称被注册，并且可以通过目标操作符上的 `getRuntimeContext().getBroadcastVariable(String)` 访问。

下面的代码片段展示了如何广播数据集并根据需要使用它。

Scala 代码：

```
package com.xueai8.scala.ch04

import org.apache.flink.api.common.functions.RichMapFunction
import org.apache.flink.api.scala._
import org.apache.flink.configuration.Configuration
import scala.collection.JavaConverters._

/**
 * 使用广播变量
 */
object BroadcastDemo {

  def main(args: Array[String]) {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    // 1. 获得一个要被广播的数据集
    val toBroadcast = env.fromElements(1, 2, 3)

    // 基本内存集合的数据集
    val data = env.fromElements("India", "USA", "UK")
```

```
val bcData = data.map(new RichMapFunction[String, String]() {
  var toBroadcast: Traversable[Int] = null

  override def open(config: Configuration): Unit = {
    // 将广播数据集作为集合访问
    toBroadcast = getRuntimeContext().getBroadcastVariable[Int]("country").asScala
  }

  override def map(input: String): String = {
    var sum = 0
    for (a <- toBroadcast) {
      sum = a + sum
    }
    input.toUpperCase + sum
  }
}).withBroadcastSet(toBroadcast, "country") // 2. 广播数据集

bcData.print()
}
```

Java 代码:

执行以上代码，输出结果为:

```
INDIA6
USA6
UK6
```

当我们要用于转换的查找条件且查找数据集相对较小时，广播变量非常有用。

注意：由于广播变量的内容保存在每个节点的内存中，所以它不应该太大。对于更简单的内容，比如标量值，可以简单地将参数作为函数闭包的一部分，或者使用 `withParameters(...)` 方法来传递配置。

4.6 分布式缓存

Flink 提供了一个分布式缓存，类似于 Apache Hadoop，使文件在本地可被用户函数的并行实例访问。此功能可用于共享包含静态外部数据（如字典或机器学习的回归模型）的文件。

缓存的工作方式如下。程序将本地或远程文件系统(如 HDFS 或 S3)的文件或目录注册到其执行环境中的特定名称下，作为缓存文件。当程序执行时，Flink 自动将文件或目录复制到所有 `TaskManager` 节点的本地文件系统。用户可以通过注册别名找到该文件或目录，然后在 `TaskManager` 节点的本地文件系统中访问该文件。

获取缓存文件的方法类似于广播变量。它还实现了 `RichFunction` 接口，通过 `RichFunction` 接口获取 `RuntimeContext` 对象，然后通过 `RuntimeContext` 提供的接口获取相应的本地缓存文件。

使用分布式缓存的步骤如下。

Scala 代码:

```
// 在 ExecutionEnvironment 中注册文件或目录。
val env = ExecutionEnvironment.getExecutionEnvironment
```

```
// 从 HDFS 注册一个文件
env.registerCachedFile("hdfs:///path/to/your/file", "hdfsFile")

// 注册一个本地可执行文件(script, executable, ...)
env.registerCachedFile("file:///path/to/exec/file", "localExecFile", true)

// 定义你的程序并执行
...
val input: DataSet[String] = ...
val result: DataSet[Integer] = input.map(new MyMapper())
...
env.execute()

// 在用户函数(这里是 MapFunction)中访问缓存的文件。
// 该函数必须扩展 RichFunction 类, 因为它需要访问 RuntimeContext。
class MyMapper extends RichMapFunction[String, Int] {

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

    // 通过 RuntimeContext 和 DistributedCache 访问缓存的文件
    val myFile: File = getRuntimeContext.getDistributedCache.getFile("hdfsFile")
    // 读取文件(或导航目录)
    ...
  }

  override def map(value: String): Int = {
    // 使用缓存文件的内容
    ...
  }
}
```

Java 代码:

```
// 在 ExecutionEnvironment 中注册文件或目录。
ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

// 从 HDFS 注册一个文件
env.registerCachedFile("hdfs:///path/to/your/file", "hdfsFile")

// 注册一个本地可执行文件(script, executable, ...)
env.registerCachedFile("file:///path/to/exec/file", "localExecFile", true)

// 定义你的程序并执行
...
DataSet<String> input = ...
DataSet<Integer> result = input.map(new MyMapper());
...
env.execute();

// 在用户函数(这里是 MapFunction)中访问缓存的文件。
// 该函数必须扩展 RichFunction 类, 因为它需要访问 RuntimeContext。
public final class MyMapper extends RichMapFunction<String, Integer> {
```

```
@Override
public void open(Configuration config) {

    // 通过 RuntimeContext 和 DistributedCache 访问缓存的文件
    File myFile = getRuntimeContext().getDistributedCache().getFile("hdfsFile");
    // 读取文件(或导航目录)
    ...
}

@Override
public Integer map(String value) throws Exception {
    // 使用缓存文件的内容
    ...
}
}
```

下面的示例演示了分布式缓存的使用方法。

首先在本地创建一个文本文件，例如 E:/目录下，内容如下：

```
1,张三
2,李四
3,王老五
```

然后将其进行分布式缓存。代码实现如下。

Scala 代码：

Java 代码：

```
package com.xueai8.java.ch04;

import org.apache.commons.io.FileUtils;
import org.apache.flink.api.common.functions.RichMapFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.configuration.Configuration;

import java.io.File;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

/**
 * 使用分布式缓存
 */
public class DistributeCacheDemo {

    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();
```

```
/**
 * 注册一个本地文件
 * 文件内容如下:
 * 1,张三
 * 2,李四
 * 3,王老五
 */
String cacheFile = "file:///E:/FlinkProjects/flinkbook/cache/userinfo.txt";
env.registerCachedFile(cacheFile, "localFileUserInfo", true);

ArrayList<Tuple2<Integer,Double>> rawUserAount = new ArrayList<>();

rawUserAount.add(new Tuple2<>(1,1000.00));
rawUserAount.add(new Tuple2<>(2,500.20));
rawUserAount.add(new Tuple2<>(3,800.50));

// 处理数据: 用户 ID, 用户购买数量, [userid, amount]
DataSet<Tuple2<Integer, Double>> userAmount = env.fromCollection(rawUserAount);

DataSet<String> result=userAmount.map(new RichMapFunction<Tuple2<Integer, Double>, String>() {
    // 保存缓存数据
    HashMap<Integer, String> allMap = new HashMap<>();

    @Override
    public void open(Configuration parameters) throws Exception {
        super.open(parameters);
        // 获取分布式缓存的数据
        File userInfoFile = getRuntimeContext().getDistributedCache().getFile("localFileUserInfo");
        List<String> userInfo = FileUtils.readLines(userInfoFile,"utf-8");
        for (String value : userInfo) {
            String[] split = value.split(",");
            allMap.put(Integer.parseInt(split[0]), split[1]);
        }
    }

    @Override
    public String map(Tuple2<Integer, Double> value) throws Exception {
        String userName = allMap.getOrDefault(value.f0,"");
        return "用户 ID:" + value.f0 + "| 用户名:" + userName + "| 购买数量:" + value.f1;
    }
});

result.print();
}
```

执行以上代码, 输出内容如下:

```
用户 ID:1| 用户名:张三| 购买数量:1000.0
用户 ID:2| 用户名:李四| 购买数量:500.2
用户 ID:3| 用户名:王老五| 购买数量:800.5
```

4.7 向函数传递参数

参数可以通过构造函数或 `withParameters(Configuration)` 方法传递给函数。参数被序列化为函数对象的一部分，并传递给所有并行任务实例。

4.7.1 通过构造函数传参

下面的示例演示了如何通过构造函数（构造器）向函数对象传递参数。

Scala 代码：

```
package com.xueai8.scala.ch04

import org.apache.flink.api.common.functions.FilterFunction
import org.apache.flink.api.scala._

/**
 * 通过构造器向函数对象传参
 */
object PassParametersDemo01 {

  def main(args: Array[String]) {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    // 数据源
    val toFilter = env.fromElements(1, 2, 3, 4, 5)

    // FilterFunction 传参：过滤出大于 2 的元素
    toFilter.filter(new MyFilter(2)).print()
  }

  class MyFilter(limit: Int) extends FilterFunction[Int] {
    override def filter(value: Int): Boolean = {
      value > limit
    }
  }
}
```

Java 代码：

```
package com.xueai8.java.ch04;

import org.apache.flink.api.common.functions.FilterFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;

/**
 * 通过构造器向函数对象传参
 */
public class PassParametersDemo01 {
```

```
public static void main(String[] args) throws Exception {
    // 设置批处理执行环境
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // 数据源
    DataSet<Integer> toFilter = env.fromElements(1, 2, 3, 4, 5);

    // FilterFunction 传参：过滤出大于 2 的元素
    toFilter.filter(new MyFilter(2)).print();
}

private static class MyFilter implements FilterFunction<Integer> {

    private final int limit;

    public MyFilter(int limit) {
        this.limit = limit;
    }

    @Override
    public boolean filter(Integer value) throws Exception {
        return value > limit;
    }
}
```

执行以上代码，输出结果如下：

```
3
4
5
```

4.7.2 通过 withParameters(Configuration)传参

下面的示例演示了如何通过 withParameters(Configuration)方法向函数对象传递参数。

Scala 代码：

```
package com.xueai8.scala.ch04

import org.apache.flink.api.common.functions.RichFilterFunction
import org.apache.flink.api.scala._
import org.apache.flink.configuration.Configuration

/**
 * 通过 withParameters(Configuration)方法向函数对象传参
 */
object PassParametersDemo02 {

    def main(args: Array[String]) {
        // 设置批处理执行环境
        val env = ExecutionEnvironment.getExecutionEnvironment
```



```
// 数据源
val toFilter = env.fromElements(1, 2, 3, 4, 5)

// 配置参数
val c = new Configuration()
c.setInteger("limit", 2)

toFilter
  .filter(new RichFilterFunction[Int]() {
    var limit = 0

    override def open(config: Configuration): Unit = {
      limit = config.getInteger("limit", 0)
    }

    override def filter(in: Int): Boolean = {
      in > limit
    }
  }).withParameters(c)
  .print()
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.api.common.functions.RichFilterFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.configuration.Configuration;

/**
 * 通过 withParameters(Configuration)方法向函数对象传参
 */
public class PassParametersDemo02 {

    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // 数据源
        DataSet<Integer> toFilter = env.fromElements(1, 2, 3, 4, 5);

        // 配置参数
        Configuration config = new Configuration();
        config.setInteger("limit", 2);

        // 向 FilterFunction 传参
        toFilter.filter(new RichFilterFunction<Integer>() {
            private int limit;
```

```
        @Override
        public void open(Configuration parameters) throws Exception {
            limit = parameters.getInteger("limit", 0);
        }

        @Override
        public boolean filter(Integer value) throws Exception {
            return value > limit;
        }
    }).withParameters(config)
    .print();
}
```

执行以上代码，输出结果如下：

```
3
4
5
```

4.7.3 通过 ExecutionConfig 传递全局参数

Flink 还允许将自定义配置值传递给环境的 ExecutionConfig 接口。由于执行配置可以在所有(Rich)用户函数中访问，因此自定义配置将在所有函数中全局可用。

Scala 代码：

```
package com.xueai8.scala.ch04

import org.apache.flink.api.common.functions.RichFilterFunction
import org.apache.flink.api.scala._
import org.apache.flink.configuration.Configuration

/**
 * Flink 还允许将自定义配置值传递给环境的 ExecutionConfig 接口。
 * 由于执行配置可以在所有(Rich)用户函数中访问，因此自定义配置将在所有函数中全局可用
 */
object PassParametersDemo03 {

    def main(args: Array[String]) {
        // 设置批处理执行环境
        val env = ExecutionEnvironment.getExecutionEnvironment

        // 数据源
        val toFilter = env.fromElements(1, 2, 3, 4, 5)

        // 1) 设置自定义全局配置
        val config = new Configuration()
        config.setInteger("limit", 2)
        env.getConfig.setGlobalJobParameters(config)

        // 向 FilterFunction 传参
```

```

toFilter
.filter(new RichFilterFunction[Int]() {
    var limit = 0

    // 2) 访问全局配置中的值
    override def open(config: Configuration): Unit = {
        val globalParams = getRuntimeContext.getExecutionConfig.getGlobalJobParameters
        val globConf = globalParams.asInstanceOf[Configuration]
        limit = globConf.getInteger("limit", 0)
    }

    override def filter(in: Int): Boolean = {
        in > limit
    }
})
.print()
}
}

```

Java 代码:

```

package com.xueai8.java.ch04;

import org.apache.flink.api.common.ExecutionConfig;
import org.apache.flink.api.common.functions.RichFilterFunction;
import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.configuration.Configuration;

public class PassParametersDemo03 {

    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // 数据源
        DataSet<Integer> toFilter = env.fromElements(1, 2, 3, 4, 5);

        // 1) 设置自定义全局配置
        Configuration config = new Configuration();
        config.setInteger("limit", 2);
        env.getConfig().setGlobalJobParameters(config);

        // 向 FilterFunction 传参
        toFilter.filter(new RichFilterFunction<Integer>() {
            private int limit;

            // 2) 访问全局配置中的值
            @Override
            public void open(Configuration parameters) throws Exception {
                ExecutionConfig.GlobalJobParameters globalParams = getRuntimeContext()
                    .getExecutionConfig()

```

```
        .getGlobalJobParameters();
        Configuration globConf = (Configuration) globalParams;
        limit = globConf.getInteger("limit", 0);
    }

    @Override
    public boolean filter(Integer value) throws Exception {
        return value > limit;
    }
}
.print();
}
```

执行以上代码，输出结果如下所示：

```
3
4
5
```

注：也可以传递一个扩展自 `ExecutionConfig.GlobalJobParameters` 类的自定义类作为执行配置的全局作业参数。该接口允许实现 `Map<String, String> toMap()` 方法，该方法将在 web 前端显示来自配置的值。

4.8 数据集中的元素拉链操作

某些算法中，可能需要将唯一标识符分配给数据集元素。可以使用 Flink DataSet API 的 `org.apache.flink.api.java.utils` 包中提供的工具类 `DataSetUtils` 实现此目的。该类提供了简单的实用程序方法，用于使用索引或唯一标识符对数据集中的元素执行拉链操作。

4.8.1 密集索引

`DataSetUtils` 类中有一个 `zipWithIndex` 方法，可以为元素分配连续的标签，它接收一个数据集作为输入并返回一个新的数据集，包含(唯一 id, 初始值)二元组。

该方法的签名如下：

```
public static <T> DataSet<Tuple2<Long, T>> zipWithIndex(DataSet<T> input)
```

该方法为输入数据集中的所有元素分配一个唯一的 `Long` 值，生成的值是连续的。这个方法为元素分配连续标签的过程需要两次遍历，第一次是计数，然后第二次标记元素。由于计数的同步，所以不能以流水线的方式进行。方法执行后返回一个由连续的 id 和初始值组成的 `DataSet[(Long, T)]` 2 元组的新数据集。当唯一的标签足够时，它是首选方法。

例如中，在以下代码中，对数据流中元素分配连续密集索引。

Scala 代码：

```
package com.xueai8.scala.ch04

import org.apache.flink.api.scala._
import org.apache.flink.api.scala.utils.DataSetUtils

/**
 * 数据集中的元素拉链操作 - 密集索引
```

```
*/
object ZipWithIndexDemo {

  def main(args: Array[String]) {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    // 设置两个并行分区
    env.setParallelism(2)

    // 初始数据源
    val input: DataSet[String] = env.fromElements("A", "B", "C", "D", "E", "F", "G", "H")

    // 分配索引
    val result: DataSet[(Long, String)] = input.zipWithIndex

    // data sink
    result.print()
  }
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.utils.DataSetUtils;

/**
 * 数据集中的元素拉链操作
 */
public class ZipWithIndexDemo {

  public static void main(String[] args) throws Exception {
    // 设置批处理执行环境
    final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

    // 设置两个并行分区
    env.setParallelism(2);

    // 初始数据源
    DataSet<String> in = env.fromElements("A", "B", "C", "D", "E", "F", "G", "H");

    // 分配索引
    DataSet<Tuple2<Long, String>> result = DataSetUtils.zipWithIndex(in);

    // data sink
    result.print();
  }
}
```

生成的结果如下所示：

```
0,A
1,B
2,C
3,D
4,E
5,F
6,G
7,H
```

4.8.2 唯一索引

在许多情况下，可能不需要分配连续的标签。DataSetUtils 类还提供了另一个 zipWithUniqueId 方法，这个方法以流水线的方式工作。

该方法的签名如下：

```
public static <T> DataSet<Tuple2<Long,T>> zipWithUniqueId(DataSet<T> input)
```

该方法为输入数据集中的所有元素分配一个唯一的 Long 值，如下所述。

- ❑ 一个 map 函数应用于输入数据集
- ❑ 每个 map 任务都有一个计数器 c，该计数器为每个记录增加一个值
- ❑ c 移位 n 位，其中 $n = \log_2(\text{并行任务的数量})$
- ❑ 要在所有任务中创建唯一的 ID，需要将任务 ID 添加到计数器中
- ❑ 对于每个记录，将收集结果计数器

这个方法接收一个数据集作为输入，并返回一个新的返回一个由 id 和初始值组成的 tuple 2 数据集。这个 zipWithUniqueId 方法以流水线的方式工作，加快了标签分配的过程。

例如中，在以下代码中，对数据流中元素分配唯一索引。

Scala 代码：

```
package com.xueai8.scala.ch04

import org.apache.flink.api.scala._
import org.apache.flink.api.scala.utils.DataSetUtils

/**
 * 数据集中的元素拉链操作 - 唯一索引
 */
object ZipWithUniqueIdDemo {

  def main(args: Array[String]) {
    // 设置批处理执行环境
    val env = ExecutionEnvironment.getExecutionEnvironment

    // 设置两个并行分区
    env.setParallelism(2)

    // 初始数据源
    val input:DataSet[String] = env.fromElements("A", "B", "C", "D", "E", "F", "G", "H")
```

```
// 分配索引
val result: DataSet[(Long,String)] = input.zipWithUniqueId

// data sink
result.print()
}
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.api.java.DataSet;
import org.apache.flink.api.java.ExecutionEnvironment;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.api.java.utils.DataSetUtils;

/**
 * 数据集中的元素拉链操作 - 唯一索引
 */
public class ZipWithUniqueIdDemo {

    public static void main(String[] args) throws Exception {
        // 设置批处理执行环境
        final ExecutionEnvironment env = ExecutionEnvironment.getExecutionEnvironment();

        // 设置两个并行分区
        env.setParallelism(2);

        // 初始数据源
        DataSet<String> in = env.fromElements("A", "B", "C", "D", "E", "F", "G", "H");

        // 分配唯一索引
        DataSet<Tuple2<Long, String>> result = DataSetUtils.zipWithUniqueId(in);

        // data sink
        result.print();
    }
}
```

执行以上代码，输出结果如下所示：

```
(0,G)
(1,A)
(2,H)
(3,B)
(5,C)
(7,D)
(9,E)
(11,F)
```

4.9 Flink 批处理示例

我们有一个已经采集了的豆瓣热门电影数据集，以 CSV 格式提供。数据集是使用爬虫采集自豆瓣的热门电影，部分数据如下所示：

```
标题,标题链接,pic,缩略图,bd,inq,hd1,rating_score,rating_num,other
蝙蝠侠：黑暗骑士,https://movie.douban.com/subject/1851857/,26,https://img3.doubanio.com/..., "导演：克里斯托弗·诺兰 Christopher Nolan 主演：克里斯蒂安·贝尔 Christ... 2008 / 美国 英国 / 剧情 动作 科幻 犯罪 惊悚",无尽的黑暗。 / The Dark Knight,9.2,713265 人评价, / 蝙蝠侠前传 2：黑暗骑士 / 黑暗骑士(台)
控方证人,https://movie.douban.com/subject/1296141/,27,https://img1.doubanio.com/..., "导演：比利·怀尔德 Billy Wilder 主演：泰隆·鲍华 Tyrone Power / 玛琳... 1957 / 美国 / 剧情 犯罪 悬疑",比利·怀德满分作品。 / Witness for the Prosecution,9.6,271118 人评价, / 雄才伟略 / 情妇
活着 ,https://movie.douban.com/subject/1292365/,28,https://img3.doubanio.com/..., "导 演：张 艺 谋 Yimou Zhang 主演：葛优 You Ge / 巩俐 Li Gong / 姜武 Wu Jiang 1994 / 中国大陆 中国香港 / 剧情 历史 家庭",张艺谋最好的电影。 / 人生 / Lifetimes,9.2,551959 人评价, / 人生 / Lifetimes
乱世佳人,https://movie.douban.com/subject/1300267/,29,https://img3.doubanio.com/..., "导演：维克多·弗莱明 Victor Fleming / 乔治·库克 George Cukor 主演：费... 1939 / 美国 / 剧情 历史 爱情 战争",Tomorrow is another day. / Gone with the Wind,9.3,473671 人评价, / 飘
寻梦环游记,https://movie.douban.com/subject/20495023/,30,https://img1.doubanio.com/..., "导演：李·昂克里奇 Lee Unkrich / 阿德里安·莫利纳 Adrian Molina 主演：... 2017 / 美国 / 喜剧 动画 奇幻 音乐",死亡不是真的逝去，遗忘才是永恒的消亡。 / Coco,9.1,1033536 人评价, / 可可夜总会(台) / 玩转极乐园(港)
末代皇帝,https://movie.douban.com/subject/1293172/,31,https://img3.doubanio.com/..., "导演：贝纳尔多·贝托鲁奇 Bernardo Bertolucci 主演：尊龙 John Lone / 陈... 1987 / 英国 意大利 中国大陆 法国 / 剧情 传记 历史",“不要跟我比惨，我比你更惨”再适合这部电影不过了。 / The Last Emperor,9.2,492158 人评价, / 末代皇帝溥仪(港)
摔跤吧！爸爸,https://movie.douban.com/subject/26387939/,32,https://img9.doubanio.com/..., "导演：涅提·蒂瓦里 Nitesh Tiwari 主演：阿米尔·汗 Aamir Khan / 法缇玛... 2016 / 印度 / 剧情 传记 运动 家庭",你不是在为你一个人战斗，你要让千千万万的女性看到女生并不是只能相夫教子。 / Dangal,9.0,1079607 人评价, / 我和我的冠军女儿(台) / 打死不离3父女(港)
指环王3：王者无敌,https://movie.douban.com/subject/1291552/,33,https://img3.doubanio.com/..., "导演：彼得·杰克逊 Peter Jackson 主演：维果·莫腾森 Viggo Mortensen / ... 2003 / 美国 新西兰 / 剧情 动作 奇幻 冒险",史诗的终章。 / The Lord of the Rings: The Return of the King,9.2,522845 人评价, / 魔戒三部曲：王者再临(台 / 港)
少年派的奇幻漂流,https://movie.douban.com/subject/1929463/,34,https://img3.doubanio.com/..., "导演：李安 Ang Lee 主演：苏拉·沙玛 Suraj Sharma / 伊尔凡·可汗 Irrfan... 2012 / 美国 中国台湾 英国 加拿大 / 剧情 奇幻 冒险",瑰丽壮观、无人能及的冒险之旅。 / Life of Pi,9.1,1004021 人评价, / 少年 Pi 的奇幻漂流 / 漂流少年 Pi
.....
```

请注意，数据集的第一行是标题行。另外，在所有字段中，我们只关心以下几个字段：

- ☐ 标题：索引为 0，表示电影名称。
- ☐ bd：索引 4，表示导演姓名。
- ☐ rating_score：索引为 7，表示电影评分。
- ☐ rating_num：索引为 8，表示电影评论人数。
- ☐ other：索引为 9，表示电影别名。

现希望通过对该数据集的分析，回答以下问题：

- ☐ 数据集中采集的电影总数量是多少？
- ☐ 这些热门电影平均评分是多少？
- ☐ 找出这些热门电影中评论人数最多的十部电影。

我们将分别以 Scala DataSet API 和 Java DataSet API 来实现这个 Flink 批处理程序。

4.9.1 分析豆瓣热门电影数据集_Scala 实现

下面使用 Flink Scala 批处理 API 分析豆瓣热门电影数据集。请按以下步骤实现。

1) 创建项目，并将源数据集拷贝到项目的"src/data/"目录下。如下图：



2) 新建一个 case class，代表数据集中的每一行电影数据。

```
case class Movie(title:String, director:String, rating:Double, ratingNum:String, other:String)
```

3) 回答第一个问题：数据集中采集的电影总数量是多少？编写代码如下：

运行上面的代码，输出统计结果为：

225

4) 回答第二个问题：这些热门电影平均评分是多少？编写代码如下：

运行上面的代码，输出统计结果为：

8.83

即这 225 部电影的平均评分是 8.83。

5) 回答第三个问题：找出这些热门电影中评论人数最多的十部电影。编写代码如下：

运行上面的代码，输出统计结果为：

```
(我不是药神,1452472)
(让子弹飞,1119027)
(摔跤吧！爸爸,1079607)
(寻梦环游记,1033536)
(绿皮书,1031394)
(少年派的奇幻漂流,1004021)
(头号玩家,987671)
(你的名字。 ,949444)
(飞屋环游记,933034)
(阿凡达,929920)
```

可以看到，“我不是药神”以 1452472 人次评论高居榜首，成为最受关注的电影。

完整代码如下。

4.9.2 分析豆瓣热门电影数据集_Java 实现

下面使用 Flink Java 批处理 API 分析豆瓣热门电影数据集。请按以下步骤实现。

1) 创建项目，并将源数据集拷贝到项目的"src/data/"目录下。如下图：



2) 新建一个 Java POJO 类，取名“Movie.java”，用来封装数据集中的每一行电影数据。

```
package com.xueai8.java.ch04;
```

```
public static class Movie{
    public String title;           // 电影名称
    public String director;       // 导演
    public double rating;         // 评分
    public String ratingNum;      // 评分人数
    public String other;          // 其他名称

    public Movie(){

    }

    public Movie(String title, String director, double rating, String ratingNum, String other) {
        this.title = title;
        this.director = director;
        this.rating = rating;
        this.ratingNum = ratingNum;
        this.other = other;
    }

    @Override
    public String toString() {
        return "Movie{" +
            "title=\"" + title + "\" +
            ", rating=\"" + rating +
            ", ratingNum=\"" + ratingNum + "\" +
            ";
    }
}
```

3) 回答第一个问题：数据集中采集的电影总数量是多少？编写代码如下：

运行上面的代码，输出统计结果为：

225

4) 回答第二个问题：这些热门电影平均评分是多少？编写代码如下：

运行上面的代码，输出统计结果为：

8.83

即这 225 部分电影的平均评分是 8.83。

5) 回答第三个问题：找出这些热门电影中评论人数最多的十部电影。编写代码如下：

运行上面的代码，输出统计结果为：

```
(我不是药神,1452472)
(让子弹飞,1119027)
(摔跤吧！爸爸,1079607)
(寻梦环游记,1033536)
(绿皮书,1031394)
(少年派的奇幻漂流,1004021)
(头号玩家,987671)
(你的名字。,949444)
(飞屋环游记,933034)
(阿凡达,929920)
```

可以看到，“我不是药神”以 1452472 人次评论高居榜首，成为最受关注的电影。

完整代码如下。

4.10 小结

在本章中，我们学习了 DataSet API。它使我们能够进行批处理。为了进行数据处理，我们学习了各种转换。稍后，我们还研究了用于从 HDFS、Amazon S3、MS Azure、Alluxio 等读取/写入数据的各种基于文件的连接器。

在上一节中，我们查看了一个用例，在这个用例中我们应用了前面几节中所学到的知识。

在下一章中，我们将从 Flink 的生态系统角度学习另一个非常重要的 API，即 Table API。

作业：分别统计 9.5 分以上的电影和电影数量。

第 5 章 使用 Table API 进行数据处理

Apache Flink 有两个关系型 API—Table API 和 SQL—用于统一的流和批处理。Table API 是一个用于 Scala 和 Java 的语言集成查询 API，它允许以一种非常直观的方式组合来自关系操作符(如选择、筛选和连接)的查询。Flink 的 SQL 支持基于 Apache Calcite，它实现了 SQL 标准。无论输入是批输入 (DataSet) 还是流输入 (DataStream)，在这两个接口中指定的查询都具有相同的语义，并指定相同的结果。

Table API 和 SQL 接口与 Flink 的 DataStream 和 DataSet API 紧密集成在一起。我们可以轻松地在所有 API 和基于 API 的库之间切换。一旦将 DataSet/DataStream 注册为表，我们就可以自由地应用关系操作，如聚合、连接和选择，还可以像常规 SQL 查询一样查询表。例如，可以使用 CEP 库从 DataStream 中提取模式，然后使用表 API 分析模式，或者在对预处理数据运行 Gelly graph 算法之前，使用 SQL 查询扫描、过滤和聚合批处理表。

5.1 依赖

根据目标编程语言的不同，我们需要将 Java 或 Scala API 添加到项目中，以便使用 Table API 和 SQL 定义管道。

```
<!-- 如果使用的是 java 语言... -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-java-bridge_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>

<!-- 或者，如果使用的是 scala 语言... -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-scala-bridge_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
```

此外，如果想在 IDE 中本地运行 Table API 和 SQL 程序，还必须添加以下一组模块。

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner-blink_2.12</artifactId>
  <version>1.13.2</version>
  <scope>test</scope>
</dependency>

<!-- 注：在内部，部分表生态系统是用 Scala 实现的 -->
<dependency>
  <groupId>org.apache.flink</groupId>
```

```
<artifactId>flink-streaming-scala_2.12</artifactId>
<version>1.13.2</version>
<scope>provided</scope>
</dependency>
```

这些依赖项将下载类路径中所需的所有 jar。下载完成后，我们就可以使用 Table API 了。

扩展依赖

如果想为序列化/反序列化行或一组用户定义函数实现自定义格式或连接器，还需要添加以下依赖项，这可以用于 SQL Client 的 JAR 文件：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-common</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
```

5.2 Table API & SQL 程序编程模式

Table API 和 SQL 集成在一个联合 API 中。这个 API 的核心概念是作为查询的输入和输出的 Table。Table API 和 SQL 查询可以很容易地与 DataStream 程序集成并嵌入其中。

5.2.1 TableEnvironment

TableEnvironment 是 Table API 和 SQL 集成的入口点，一个 Table 总是绑定到一个特定的 TableEnvironment。TableEnvironment 在语言层面上是统一的，适用于所有基于 JVM 的语言(即 Scala 和 Java API 之间没有区别)，也适用于有界和无界数据处理。它负责：

- ☐ 连接外部系统；
- ☐ 注册 catalogs；
- ☐ 在内部 catalog 目录中注册一个 Table；
- ☐ 从 catalog 目录中注册和检索 Tables 和其他元对象；
- ☐ 提供进一步的配置选项；
- ☐ 执行 SQL 查询；
- ☐ 注册用户定义函数（标量函数，表函数，或聚合函数）；
- ☐ DataStream 和 Table 之间的转换(在 StreamTableEnvironment 中)。

在同一个查询中组合中必须使用同一个 TableEnvironments 的表。通过调用 TableEnvironment 的静态 create()方法可以创建一个 TableEnvironment。

Scala 代码：

```
import org.apache.flink.table.api.{EnvironmentSettings, TableEnvironment}

val settings = EnvironmentSettings
  .newInstance()
  .inStreamingMode() //inBatchMode()
  .build()
```

```
val tEnv = TableEnvironment.create(setting)
```

Java 代码:

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

```
EnvironmentSettings settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode()    //inBatchMode()
    .build();
```

```
TableEnvironment tEnv = TableEnvironment.create(settings);
```

或者，用户可以从现有的 `StreamExecutionEnvironment` 创建一个 `StreamTableEnvironment` 来与 `DataStream API` 进行互操作。

Scala 代码:

```
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.table.api.EnvironmentSettings
import org.apache.flink.table.api.bridge.scala.StreamTableEnvironment
```

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tEnv = StreamTableEnvironment.create(env)
```

Java 代码:

```
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
```

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);
```

【示例】给出一些学生各个科目的成绩，请计算每个学生的总成绩。

Scala 代码:

```
package com.xueai8.scala.ch05
```

```
import org.apache.flink.table.api.Expressions.$
import org.apache.flink.table.api._
```

```
object TableDemo00 {
```

```
    def main(args: Array[String]) {
```

```
        // 创建表环境
```

```
        val settings = EnvironmentSettings.newInstance.inBatchMode.build
```

```
        val tEnv = TableEnvironment.create(settings)
```

```
        // 计算每个学生的总成绩
```

```
        tEnv
```

```
            .fromValues(
```

```
                row("张三", "math", 82),
```

```
        row("张三", "english", 92),
        row("李四", "math", 86),
        row("李四", "english", 91)
    )
    .as("name", "subject", "score")
    .groupBy($"name")
    .select($"name", $"score").sum.as("score_sum"))
    .execute()
    .print()
}
}
```

Java 代码:

```
package com.xueai8.java.ch05;

import org.apache.flink.table.api.*;
import static org.apache.flink.table.api.Expressions.$;
import static org.apache.flink.table.api.Expressions.row;

public class TableDemo00 {

    public static void main(String[] args) throws Exception {
        // 创建表环境
        EnvironmentSettings settings = EnvironmentSettings
            .newInstance()
            .inBatchMode()    //inStreamingMode()
            .build();

        TableEnvironment tEnv = TableEnvironment.create(settings);

        // 计算每个学生的总成绩
        tEnv
            .fromValues(
                row("张三", "math", 82),
                row("张三", "english", 92),
                row("李四", "math", 86),
                row("李四", "english", 91)
            )
            .as("name", "subject", "score")
            .groupBy($"name")
            .select($"name", $"score").sum().as("score_sum"))
            .execute()
            .print();
    }
}
```

执行以上代码，输出内容如下：

```
+-----+-----+
|           name | score_sum |
+-----+-----+
|       李四 |       177 |
|       张三 |       174 |
```

```
+-----+
2 rows in set
```

5.2.2 Table API & SQL 程序的结构

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

Scala 代码:

```
// 创建用于批或流执行的 TableEnvironment
val tableEnv = ... // 创建一个 TableEnvironment

// 创建一个输入表
tableEnv.executeSql("CREATE TEMPORARY TABLE table1 ... WITH ( 'connector' = ... )")

// 注册一个输出表
tableEnv.executeSql("CREATE TEMPORARY TABLE outputTable ... WITH ( 'connector' = ... )")

// 从 Table API 查询创建表
val table2 = tableEnv.from("table1").select(...)

// 从 SQL 查询创建一个表
val table3 = tableEnv.sqlQuery("SELECT ... FROM table1 ...")

// 发出 Table API 结果表给一个 TableSink, 同样适用于 SQL 结果
val tableResult = table2.executeInsert("outputTable")
tableResult...
```

Java 代码:

```
// 创建用于批或流执行的 TableEnvironment
TableEnvironment tableEnv = ...; // 创建一个 TableEnvironment

// 创建一个输入表
tableEnv.executeSql("CREATE TEMPORARY TABLE table1 ... WITH ( 'connector' = ... )");

// 注册一个输出表
tableEnv.executeSql("CREATE TEMPORARY TABLE outputTable ... WITH ( 'connector' = ... )");

// 从 Table API 查询创建表
Table table2 = tableEnv.from("table1").select(...);

// 从 SQL 查询创建一个表
Table table3 = tableEnv.sqlQuery("SELECT ... FROM table1 ...");

// 发出 Table API 结果表给一个 TableSink, 同样适用于 SQL 结果
TableResult tableResult = table2.executeInsert("outputTable");
tableResult...
```


5.2.3 在 Catalog 中创建表

TableEnvironment 维护使用标识符创建的表的目录映射。每个标识符由 3 部分组成：目录名、数据库名和对象名。如果没有指定目录或数据库，将使用当前默认值。

表可以是虚拟的(VIEWS)，也可以是常规的(TABLES)。VIEWS 可以从现有的 Table 对象创建，通常是表 API 或 SQL 查询的结果。TABLES 描述外部数据，如文件、数据库表或消息队列。

临时表与永久表

表可以是临时的，并绑定到单个 Flink 会话的生命周期，也可以是永久的，并且跨多个 Flink 会话和集群可见。

永久表需要目录(catalog，如 Hive Metastore)来维护关于表的元数据。一旦创建了永久表，它对于连接到目录的任何 Flink 会话都是可见的，并且将持续存在，直到显式地删除该表。

另一方面，临时表总是存储在内存中，并且只在创建它们的 Flink 会话期间存在。这些表对其他会话不可见。它们没有绑定到任何目录或数据库，但可以在其名称空间(namespace)中创建。如果删除了相应的数据库，则不会删除临时表。

可以使用与现有永久表相同的标识符注册临时表。临时表隐藏了永久表，只要临时表存在，就无法访问永久表。具有该标识符的所有查询都将对临时表执行。

这可能对实验有用。它允许首先对一个临时表运行完全相同的查询，例如只有一个数据子集，或者数据是模糊的。一旦验证了查询是正确的，就可以对实际的生产表运行查询。

创建表

一个 Table API 对象对应于 SQL 术语中的 VIEW(虚拟表)。它封装了一个逻辑查询计划。它可以在一个目录中创建，如下所示：

Scala 代码：

```
// 获得一个 TableEnvironment
import org.apache.flink.table.api.{EnvironmentSettings, TableEnvironment}

val settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode() //inBatchMode()
    .build()

val tEnv = TableEnvironment.create(settings)

// 表是一个简单的投影查询的结果
val projTable: Table = tableEnv.from("X").select(...)

// 将表 projTable 注册为表 "projectedTable"
tEnv.createTemporaryView("projectedTable", projTable)
```

Java 代码：

```
// 获得一个 TableEnvironment
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.TableEnvironment;
```

```
EnvironmentSettings settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode()    //inBatchMode()
    .build();

TableEnvironment tEnv = TableEnvironment.create(setting);

// 表是一个简单的投影查询的结果
Table projTable = tEnv.from("X").select(...);

// 将表 projTable 注册为表 "projectedTable"
tEnv.createTemporaryView("projectedTable", projTable);
```

注：Table 对象类似于关系数据库系统中的 VIEW 对象，即，定义 Table 的查询没有优化，但当另一个查询引用已注册的 Table 时将内联。如果多个查询引用同一个注册 Table，它将内联每个引用查询并执行多次，也就是说，注册 Table 的结果将不会被共享。

也可以通过连接器声明创建关系数据库中的 TABLE。连接器描述存储表数据的外部系统。存储系统（如 Apache Kafka 或常规文件系统）可以在这里声明。

Scala 代码：

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

Java 代码：

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

SQL 代码：

```
tableEnvironment.executeSql("CREATE [TEMPORARY] TABLE MyTable (...) WITH (...)")
```

扩展表标识符

表总是使用由目录、数据库和表名组成的三部分标识符进行注册。

用户可以将其中的一个目录和一个数据库设置为“当前目录”和“当前数据库”。有了它们，上面提到的三部分标识符中的前两个部分可以省略—如果没有提供它们，将引用当前目录和当前数据库。用户可以通过 TABLE API 或 SQL 切换当前目录和当前数据库。

标识符遵循 SQL 要求，这意味着可以使用反号字符(‘)进行转义。

Scala 代码：

```
// 获得一个 TableEnvironment
import org.apache.flink.table.api.{EnvironmentSettings, TableEnvironment}

val settings = EnvironmentSettings
```

```
.newInstance()
.inStreamingMode() //inBatchMode()
.build()

val tEnv = TableEnvironment.create(setting)

// 指定当前目录和当前数据库
tEnv.useCatalog("custom_catalog")
tEnv.useDatabase("custom_database")

val table: Table = tableEnv.from("X").select(...)

// 在名为 custom_catalog 的目录中注册名为 exampleView 的视图
// 在名为 custom_database 的数据库中
tEnv.createTemporaryView("exampleView", table)

// 在名为 custom_catalog 的目录中注册名为 exampleView 的视图
// 在名为 other_database 的数据库中
tEnv.createTemporaryView("other_database.exampleView", table)

// 在名为 custom_catalog 的目录中注册名为 example.View 的视图
// 在名为 custom_database 的数据库中
tEnv.createTemporaryView("example.View", table)

// 在名为 other_catalog 的目录中注册名为 exampleView 的视图
// 在名为 other_database 的数据库中
tEnv.createTemporaryView("other_catalog.other_database.exampleView", table)
```

Java 代码:

```
// 获得一个 TableEnvironment
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.TableEnvironment;

EnvironmentSettings settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode() //inBatchMode()
    .build();

TableEnvironment tEnv = TableEnvironment.create(setting);

// 指定当前目录和当前数据库
tEnv.useCatalog("custom_catalog");
tEnv.useDatabase("custom_database");

Table table = tEnv.from("X").select(...);

// 在名为 custom_catalog 的目录中注册名为 exampleView 的视图
// 在名为 custom_database 的数据库中
tEnv.createTemporaryView("exampleView", table);

// 在名为 custom_catalog 的目录中注册名为 exampleView 的视图
```

```
// 在名为 other_database 的数据库中
tEnv.createTemporaryView("other_database.exampleView", table);

// 在名为 custom_catalog 的目录中注册名为 example.View 的视图
// 在名为 custom_database 的数据库中
tEnv.createTemporaryView("example.View", table);

// 在名为 other_catalog 的目录中注册名为 exampleView 的视图
// 在名为 other_database 的数据库中
tEnv.createTemporaryView("other_catalog.other_database.exampleView", table);
```

5.2.4 查询表

既可以使用 Table API 查询创建的表，也可以使用 SQL 查询创建的表。

使用 Table API

Table API 是用于 Scala 和 Java 的语言集成查询 API，是批处理和流数据的统一 API。

Table API 基于 Table 类，Table 类表示一个表（流或批），并提供应用关系操作的方法。这些方法返回一个新的 Table 对象，它表示在输入 Table 上应用关系操作的结果。一些关系操作由多个方法调用组成，比如 `table.groupBy(...).select()`，其中 `groupBy(...)` 指定一个 table 分组，并 `select(...)` table 分组上的投影。

下面的例子展示了一个简单的表 API 聚合查询。

Scala 代码：

```
// 获得一个 TableEnvironment
import org.apache.flink.table.api.{EnvironmentSettings, TableEnvironment}

val settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode() // inBatchMode()
    .build()

val tEnv = TableEnvironment.create(settings)

// 注册 Orders 表

// 扫描已注册 Orders 表
val orders = tableEnv.from("Orders")

// 计算来自法国的所有客户的收入
val revenue = orders
    .filter($"cCountry" === "FRANCE")
    .groupBy($"cID", $"cName")
    .select($"cID", $"cName", $"revenue".sum AS "revSum")

// emit 或 convert Table
// 执行查询
```

注：Scala Table API 使用了以 \$ 符号开头的 Scala String 插值来引用 Table 的属性。Table API 使用了

Scala 隐式。确保导入：

```
import org.apache.flink.table.api._    // 用于隐式表达式转换

// 如果想与 DataStream 互相转换，还要执行以下两个导入
import org.apache.flink.api.scala._
import org.apache.flink.table.api.bridge.scala._
```

Java 代码：

```
// 获得一个 TableEnvironment
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.TableEnvironment;

EnvironmentSettings settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode()    //inBatchMode()
    .build();

TableEnvironment tEnv = TableEnvironment.create(setting);

// 注册 Orders 表

// 扫描已注册 Orders 表
Table orders = tableEnv.from("Orders");

// 计算来自法国的所有客户的收入
Table revenue = orders
    .filter($"cCountry").isEqual("FRANCE"))
    .groupBy($"cID"), $"cName")
    .select($"cID"), $"cName"), $"(revenue).sum().as("revSum"));

// emit 或 convert Table
// 执行查询
```

使用 SQL

Flink 的 SQL 集成基于 Apache Calcite，它实现了 SQL 标准。SQL 查询被指定为常规字符串。下面的示例显示如何指定查询并将结果作为 Table 返回。

Scala 代码：

```
// 获得一个 TableEnvironment
import org.apache.flink.table.api.{EnvironmentSettings, TableEnvironment}

val settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode()    //inBatchMode()
    .build()

val tEnv = TableEnvironment.create(setting)

// 注册 Orders 表
```

```
// 计算来自法国的所有客户的收入
val revenue = tEnv.sqlQuery("""
|SELECT cID, cName, SUM(revenue) AS revSum
|FROM Orders
|WHERE cCountry = 'FRANCE'
|GROUP BY cID, cName
|""").stripMargin)

// emit 或 convert Table
// 执行查询

Java 代码:
// 获得一个 TableEnvironment
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.TableEnvironment;

EnvironmentSettings settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode()    //inBatchMode()
    .build();

TableEnvironment tEnv = TableEnvironment.create(setting);

// 注册 Orders 表

// 计算来自法国的所有客户的收入
Table revenue = tEnv.sqlQuery(
    "SELECT cID, cName, SUM(revenue) AS revSum " +
    "FROM Orders " +
    "WHERE cCountry = 'FRANCE' " +
    "GROUP BY cID, cName"
);

// emit 或 convert Table
// 执行查询
```

下面的示例显示如何指定将其结果插入已注册表的更新查询。

Scala 代码:

```
// 获得一个 TableEnvironment
import org.apache.flink.table.api.{EnvironmentSettings, TableEnvironment}

val settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode()    //inBatchMode()
    .build()

val tEnv = TableEnvironment.create(setting)

// 注册 Orders 表
// 注册"RevenueFrance"输出表
```

```
// 计算来自法国的所有客户的收入并发送到 "RevenueFrance"
tEnv.sqlUpdate("""
|INSERT INTO RevenueFrance
|SELECT cID, cName, SUM(revenue) AS revSum
|FROM Orders
|WHERE cCountry = 'FRANCE'
|GROUP BY cID, cName
|""").stripMargin)
```

// 执行查询

Java 代码:

// 获得一个 TableEnvironment

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

```
EnvironmentSettings settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode()    //inBatchMode()
    .build();
```

```
TableEnvironment tEnv = TableEnvironment.create(setting);
```

// 注册 Orders 表

// 注册"RevenueFrance"输出表

// 计算来自法国的所有客户的收入并发送到 "RevenueFrance"

```
tableEnv.sqlUpdate(
    "INSERT INTO RevenueFrance " +
    "SELECT cID, cName, SUM(revenue) AS revSum " +
    "FROM Orders " +
    "WHERE cCountry = 'FRANCE' " +
    "GROUP BY cID, cName"
);
```

// 执行查询

混合使用 Table API 和 SQL

Table API 和 SQL 查询很容易混合使用，因为它们都返回 Table 对象：

- ❑ 可以在 SQL 查询返回的 Table 对象上定义 Table API 查询。
- ❑ 可以在 Table API 查询的结果上定义 SQL 查询，方法是将结果表注册到 TableEnvironment 中，并在 SQL 查询的 FROM 子句中引用它。

5.2.5 发出一个表

Table 是通过将其写入 TableSink 而发出的。一个 TableSink 是一个通用接口，支持各种文件格式(如 CSV、Apache Parquet、Apache Avro)，存储系统(如 JDBC、Apache HBase、Apache Cassandra、Elasticsearch)，

或消息系统(如 Apache Kafka、RabbitMQ)。

批处理表只能被写到一个 BatchTableSink 中，而流处理表需要一个 AppendStreamTableSink、一个 RetractStreamTableSink 或一个 UpsertStreamTableSink。

Table.executeInsert(String tableName)方法将 Table 发送到已注册的 TableSink 中。该方法通过名称从目录中查找 TableSink，并验证该 Table 的模式是否与该 TableSink 的模式相同。

下面的例子展示了如何发出一个 Table：

Scala 代码：

```
// 获得一个 TableEnvironment
import org.apache.flink.table.api.{EnvironmentSettings, TableEnvironment}

val settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode() //inBatchMode()
    .build()

val tEnv = TableEnvironment.create(setting)

// 创建输出表
val schema = new Schema()
    .field("a", DataTypes.INT())
    .field("b", DataTypes.STRING())
    .field("c", DataTypes.LONG())

// 注册 TableSink
tEnv.connect(new FileSystem().path("/path/to/file"))
    .withFormat(new Csv().fieldDelimiter('|').deriveSchema())
    .withSchema(schema)
    .createTemporaryTable("CsvSinkTable")

// 使用 Table API 操作符和/或 SQL 查询计算结果表
val result: Table = ...

// 将结果表发送到已注册的 TableSink
result.executeInsert("CsvSinkTable")
```

Java 代码：

```
// 获得一个 TableEnvironment
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.TableEnvironment;

EnvironmentSettings settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode() //inBatchMode()
    .build();

TableEnvironment tEnv = TableEnvironment.create(setting);

// 创建输出表
final Schema schema = new Schema()
```



```
.field("a", DataTypes.INT())
.field("b", DataTypes.STRING())
.field("c", DataTypes.LONG());

// 注册 TableSink
tEnv.connect(new FileSystem().path("/path/to/file"))
    .withFormat(new Csv().fieldDelimiter('|').deriveSchema())
    .withSchema(schema)
    .createTemporaryTable("CsvSinkTable");

// 使用 Table API 操作符和/或 SQL 查询计算结果表
Table result = ...

// 将结果表发送到已注册的 TableSink
result.executeInsert("CsvSinkTable");
```

5.2.6 翻译并执行查询

翻译（translated）和执行查询的行为对于这两个计划器是不同的。Table API 和 SQL 查询被转换成 `DataStream` 程序，无论它们的输入是流还是批。查询在内部表示为逻辑查询计划，并分为两个阶段进行转换：

- ☐ 优化逻辑计划，
- ☐ 转换成 `DataStream` 程序。

Table API 或 SQL 查询在以下情况下被 translated：

- ☐ `TableEnvironment.executeSql()` 被调用。此方法用于执行给定的语句，一旦调用此方法，SQL 查询将立即 translated。
- ☐ `Table.executeInsert()` 被调用。此方法用于将表内容插入到给定的接收器路径，并且一旦调用此方法，Table API 将立即 translated。
- ☐ `Table.execute()` 被调用。此方法用于将表内容收集到本地客户机，并且一旦调用此方法，Table API 将立即 translated。
- ☐ `StatementSet.execute()` 被调用。Table（通过 `StatementSet.addInsert()` 发出到 sink）或 INSERT 语句（通过 `StatementSet.addInsertSql()` 指定）将首先在 `StatementSet` 中缓冲。一旦调用 `StatementSet.execute()`，它们就会被转换。所有 sink 都将被优化成一个 DAG。
- ☐ 当 Table 转换为 `DataStream` 时，它将被 translated。一旦转换，它就是一个常规的 `DataStream` 程序，并在调用 `StreamExecutionEnvironment.execute()` 时执行。

查询优化

Apache Flink 利用并扩展了 Apache Calcite 来执行复杂的查询优化。这包括一系列基于规则和成本的优化，如：

- ☐ 基于 Apache Calcite 的子查询解相关，
- ☐ 投影裁剪，
- ☐ 分区裁剪，
- ☐ 过滤器下推，
- ☐ 子计划去重，避免重复计算，

- ❑ 特殊子查询重写，包括两部分：
 - 将 IN 和 EXISTS 转换为左半连接
 - 将 NOT IN 和 NOT EXISTS 转换为左反连接
- ❑ 可选地 join 重新排序
 - 通过 table.optimizer.join-reorder-enabled 启用

优化器不仅根据计划，还根据数据源提供的丰富统计信息和每个操作符(如 io、cpu、网络 and 内存)的细粒度成本做出智能决策。

高级用户可以通过 CalciteConfig 对象提供自定义优化，CalciteConfig 对象可以通过调用 TableEnvironment#getConfig# setPlanerConfig 提供给表环境。

解释表

Table API 提供了一种机制来解释用于计算 Table 的逻辑和优化的查询计划。这是通过 Table.explain() 方法或 StatementSet.explain() 方法完成的。Table.explain() 返回 Table 的计划。StatementSet.explain() 返回多个接收器的计划。它返回三个计划的一个字符串描述：

- ❑ 关系查询的抽象语法树，即未优化的逻辑查询计划，
- ❑ 优化的逻辑查询计划，和
- ❑ 物理执行计划

TableEnvironment.explainSql() 和 TableEnvironment.executeSql() 支持执行 EXPLAIN 语句来获取计划。下面的代码显示了一个示例和使用 Table.explain() 方法给出的 Table 的相应输出。

Scala 代码：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tEnv = StreamTableEnvironment.create(env)

val table1 = env.fromElements((1, "hello")).toTable(tEnv, $"count", $"word")
val table2 = env.fromElements((1, "hello")).toTable(tEnv, $"count", $"word")
val table = table1
  .where($"word".like("F%"))
  .unionAll(table2)

println(table.explain())
```

Java 代码：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);

DataStream<Tuple2<Integer, String>> stream1 = env.fromElements(new Tuple2<>(1, "hello"));
DataStream<Tuple2<Integer, String>> stream2 = env.fromElements(new Tuple2<>(1, "hello"));

// explain Table API
Table table1 = tEnv.fromDataStream(stream1, $"count", $"word");
Table table2 = tEnv.fromDataStream(stream2, $"count", $"word");
Table table = table1
  .where($"word".like("F%"))
  .unionAll(table2);

System.out.println(table.explain());
```

上面示例的输出结果如下：

```
== Abstract Syntax Tree ==
LogicalUnion(all=[true])
  LogicalFilter(condition=[LIKE($1, _UTF-16LE'F%')])
    FlinkLogicalDataStreamScan(id=[1], fields=[count, word])
    FlinkLogicalDataStreamScan(id=[2], fields=[count, word])

== Optimized Logical Plan ==
DataStreamUnion(all=[true], union all=[count, word])
  DataStreamCalc(select=[count, word], where=[LIKE(word, _UTF-16LE'F%')])
    DataStreamScan(id=[1], fields=[count, word])
    DataStreamScan(id=[2], fields=[count, word])

== Physical Execution Plan ==
Stage 1 : Data Source
  content : collect elements with CollectionInputFormat

Stage 2 : Data Source
  content : collect elements with CollectionInputFormat

Stage 3 : Operator
  content : from: (count, word)
  ship_strategy : REBALANCE

Stage 4 : Operator
  content : where: (LIKE(word, _UTF-16LE'F%')), select: (count, word)
  ship_strategy : FORWARD

Stage 5 : Operator
  content : from: (count, word)
  ship_strategy : REBALANCE
```

下面的代码显示了一个示例和使用 `StatementSet.explain()` 方法的多 sink 计划的相应输出。

Scala 代码：

```
val settings = EnvironmentSettings.newInstance.useBlinkPlanner.inStreamingMode.build
val tEnv = TableEnvironment.create(settings)

val schema = new Schema()
  .field("count", DataTypes.INT())
  .field("word", DataTypes.STRING())

tEnv.connect(new FileSystem().path("/source/path1"))
  .withFormat(new Csv().deriveSchema())
  .withSchema(schema)
  .createTemporaryTable("MySource1")
tEnv.connect(new FileSystem().path("/source/path2"))
  .withFormat(new Csv().deriveSchema())
  .withSchema(schema)
  .createTemporaryTable("MySource2")
```

```
tEnv.connect(new FileSystem().path("/sink/path1"))
  .withFormat(new Csv().deriveSchema())
  .withSchema(schema)
  .createTemporaryTable("MySink1")
tEnv.connect(new FileSystem().path("/sink/path2"))
  .withFormat(new Csv().deriveSchema())
  .withSchema(schema)
  .createTemporaryTable("MySink2")

val stmtSet = tEnv.createStatementSet()

val table1 = tEnv.from("MySource1").where($"word".like("F%"))
stmtSet.addInsert("MySink1", table1)

val table2 = table1.unionAll(tEnv.from("MySource2"))
stmtSet.addInsert("MySink2", table2)
```

```
val explanation = stmtSet.explain()
println(explanation)
```

Java 代码:

```
EnvironmentSettings settings = EnvironmentSettings.newInstance()
  .useBlinkPlanner()
  .inStreamingMode()
  .build();
TableEnvironment tEnv = TableEnvironment.create(settings);

final Schema schema = new Schema()
  .field("count", DataTypes.INT())
  .field("word", DataTypes.STRING());

tEnv.connect(new FileSystem().path("/source/path1"))
  .withFormat(new Csv().deriveSchema())
  .withSchema(schema)
  .createTemporaryTable("MySource1");
tEnv.connect(new FileSystem().path("/source/path2"))
  .withFormat(new Csv().deriveSchema())
  .withSchema(schema)
  .createTemporaryTable("MySource2");
tEnv.connect(new FileSystem().path("/sink/path1"))
  .withFormat(new Csv().deriveSchema())
  .withSchema(schema)
  .createTemporaryTable("MySink1");
tEnv.connect(new FileSystem().path("/sink/path2"))
  .withFormat(new Csv().deriveSchema())
  .withSchema(schema)
  .createTemporaryTable("MySink2");

StatementSet stmtSet = tEnv.createStatementSet();

Table table1 = tEnv.from("MySource1").where($"word".like("F%"));
```

```
stmtSet.addInsert("MySink1", table1);
```

```
Table table2 = table1.unionAll(tEnv.from("MySource2"));
```

```
stmtSet.addInsert("MySink2", table2);
```

```
String explanation = stmtSet.explain();
```

```
System.out.println(explanation);
```

上面代码的输出结果如下：

```
== Abstract Syntax Tree ==
```

```
LogicalLegacySink(name=[MySink1], fields=[count, word])
```

```
+ LogicalFilter(condition=[LIKE($1, _UTF-16LE'F%')])
```

```
  +- LogicalTableScan(table=[[default_catalog, default_database, MySource1, source: [CsvTableSource(read fields: count, word)]]])
```

```
LogicalLegacySink(name=[MySink2], fields=[count, word])
```

```
+ LogicalUnion(all=[true])
```

```
  :- LogicalFilter(condition=[LIKE($1, _UTF-16LE'F%')])
```

```
  :      +- LogicalTableScan(table=[[default_catalog, default_database, MySource1, source: [CsvTableSource(read fields: count, word)]]])
```

```
  +- LogicalTableScan(table=[[default_catalog, default_database, MySource2, source: [CsvTableSource(read fields: count, word)]]])
```

```
== Optimized Logical Plan ==
```

```
Calc(select=[count, word], where=[LIKE(word, _UTF-16LE'F%')], reuse_id=[1])
```

```
+ TableSourceScan(table=[[default_catalog, default_database, MySource1, source: [CsvTableSource(read fields: count, word)]]], fields=[count, word])
```

```
LegacySink(name=[MySink1], fields=[count, word])
```

```
+ Reused(reference_id=[1])
```

```
LegacySink(name=[MySink2], fields=[count, word])
```

```
+ Union(all=[true], union=[count, word])
```

```
  :- Reused(reference_id=[1])
```

```
  +- TableSourceScan(table=[[default_catalog, default_database, MySource2, source: [CsvTableSource(read fields: count, word)]]], fields=[count, word])
```

```
== Physical Execution Plan ==
```

```
Stage 1 : Data Source
```

```
  content : collect elements with CollectionInputFormat
```

```
Stage 2 : Operator
```

```
  content : CsvTableSource(read fields: count, word)
```

```
  ship_strategy : REBALANCE
```

```
Stage 3 : Operator
```

```
  content : SourceConversion(table:Buffer(default_catalog, default_database, MySource1, source: [CsvTableSource(read fields: count, word)]), fields:(count, word))
```

```
  ship_strategy : FORWARD
```

```
Stage 4 : Operator
```

```
content : Calc(where: (word LIKE _UTF-16LE'F%'), select: (count, word))
ship_strategy : FORWARD

Stage 5 : Operator
content : SinkConversionToRow
ship_strategy : FORWARD

Stage 6 : Operator
content : Map
ship_strategy : FORWARD

Stage 8 : Data Source
content : collect elements with CollectionInputFormat

Stage 9 : Operator
content : CsvTableSource(read fields: count, word)
ship_strategy : REBALANCE

Stage 10 : Operator
content : SourceConversion(table:Buffer(default_catalog, default_database, MySource2, source:
[CsvTableSource(read fields: count, word)]), fields:(count, word))
ship_strategy : FORWARD

Stage 12 : Operator
content : SinkConversionToRow
ship_strategy : FORWARD

Stage 13 : Operator
content : Map
ship_strategy : FORWARD

Stage 7 : Data Sink
content : Sink: CsvTableSink(count, word)
ship_strategy : FORWARD

Stage 14 : Data Sink
content : Sink: CsvTableSink(count, word)
ship_strategy : FORWARD
```

5.3 Table API

Table API 是一个用于流和批处理的统一的关系 API。Table API 查询可以在批或流输入上运行，而无需做修改。Table API 是 SQL 语言的一个超级集合，是专门为使用 Apache Flink 而设计的。

5.3.1 关系运算

Table API 支持如下的运算符。（请注意，并不是所有的运算符都同时支持流和批运算）

扫描、投影和过滤

- ❑ **from**: 类似于 SQL 查询中的 **from** 子句, 执行注册表的扫描。

Scala 代码:

```
val orders = tableEnv.from("Orders")
```

Java 代码:

```
Table orders = tableEnv.from("Orders");
```

- ❑ **fromValues**: 类似于 SQL 查询中的 **values** 子句, 根据提供的行生成内联表。可以使用 **row(...)** 表达式来创建复合行。

Scala 代码:

```
val table = tEnv.fromValues(  
  row(1, "ABC"),  
  row(2L, "ABCDE")  
)
```

Java 代码:

```
Table table = tEnv.fromValues(  
  row(1, "ABC"),  
  row(2L, "ABCDE")  
);
```

以上代码片段创建的表具有如下的模式:

```
root  
|-- f0: BIGINT NOT NULL      // 将原始类型 INT 和 BIGINT 泛化为 BIGINT  
|-- f1: VARCHAR(5) NOT NULL // 原始类型 CHAR(3)和 CHAR(5)泛化为 VARCHAR(5)  
                           // 使用 VARCHAR 而不是 CHAR, 因此不应用填充
```

该方法将自动从输入表达式派生类型。如果某个位置的类型不同, 该方法将尝试为所有类型找到一个通用的超类型。如果不存在公共超类型, 将引发异常。

还可以显式指定请求的类型。它可能有助于分配更泛化的类型, 例如 **DECIMAL** 或命名列。

Scala 代码:

```
val table = tEnv.fromValues(  
  DataTypes.ROW(  
    DataTypes.FIELD("id", DataTypes.DECIMAL(10, 2)),  
    DataTypes.FIELD("name", DataTypes.STRING())  
  ),  
  row(1, "ABC"),  
  row(2L, "ABCDE")  
)
```

Java 代码:

```
Table table = tEnv.fromValues(  
  DataTypes.ROW(  
    DataTypes.FIELD("id", DataTypes.DECIMAL(10, 2)),  
    DataTypes.FIELD("name", DataTypes.STRING())  
  ),  
  row(1, "ABC"),  
  row(2L, "ABCDE")  
);
```

以上代码片段创建的表将具有如下的模式：

```
root
|-- id: DECIMAL(10, 2)
|-- name: STRING
```

❑ **select**: 类似于 SQL 语句的 **select** 子句，执行选择操作。

Scala 代码：

```
val orders = tableEnv.from("Orders")
Table result = orders.select($"a", $"c" as "d");
```

Java 代码：

```
Table orders = tableEnv.from("Orders");
Table result = orders.select($"a", $"c").as("d");
```

可以使用星号(*)作为通配符，选择表中的所有列。

Scala 代码：

```
Table result = orders.select($"*")
```

Java 代码：

```
Table result = orders.select($"*");
```

❑ **as**: 重命名字段。

Scala 代码：

```
val orders: Table = tableEnv.from("Orders").as("x", "y", "z", "t")
```

Java 代码：

```
Table orders = tableEnv.from("Orders");
Table result = orders.as("x, y, z, t");
```

❑ **where/filter**: 类似于 SQL 中的 **where** 子句。过滤不通过筛选谓词的行。

Scala 代码：

```
val orders: Table = tableEnv.from("Orders")
val result = orders.where($"a" % 2 === 0)
```

Java 代码：

```
Table orders = tableEnv.from("Orders");
Table result = orders.where($"b").isEqual("red");
```

或者

Scala 代码：

```
val orders: Table = tableEnv.from("Orders")
val result = orders.filter($"a" % 2 === 0)
```

Java 代码：

```
Table orders = tableEnv.from("Orders");
Table result = orders.filter($"b").isEqual("red");
```

列运算符

❑ **addColumnns**: 执行字段添加操作。如果添加的字段已经存在，它将抛出异常。

Scala 代码：

```
val orders = tableEnv.from("Orders");
val result = orders.addColumnns(concat($"c", "Sunny"))
```


Java 代码:

```
Table orders = tableEnv.from("Orders");
Table result = orders.addColumn(concat($"c", "sunny"));
```

- ❑ **addOrReplaceColumns:** 执行字段添加操作。如果添加的列名与现有列名相同,则现有字段将被替换。此外,如果添加的字段有重复的字段名,则使用最后一个字段名。

Scala 代码:

```
val orders = tableEnv.from("Orders");
val result = orders.addOrReplaceColumns(concat($"c", "Sunny") as "desc")
```

Java 代码:

```
Table orders = tableEnv.from("Orders");
Table result = orders.addOrReplaceColumns(concat($"c", "sunny").as("desc"));
```

- ❑ **dropColumns:** 执行字段删除操作。

Scala 代码:

```
val orders = tableEnv.from("Orders");
val result = orders.dropColumns($"b", $"c")
```

Java 代码:

```
Table orders = tableEnv.from("Orders");
Table result = orders.dropColumns($"b", $"c");
```

- ❑ **renameColumns:** 执行字段重命名操作。字段表达式应该是别名表达式,并且只能重命名现有字段。

Scala 代码:

```
val orders = tableEnv.from("Orders");
val result = orders.renameColumns($"b" as "b2", $"c" as "c2")
```

Java 代码:

```
Table orders = tableEnv.from("Orders");
Table result = orders.renameColumns($"b").as("b2"), $"c").as("c2"));
```

聚合操作

- ❑ **groupBy:** 类似于 SQL 的 group by 子句。

Scala 代码:

```
val orders: Table = tableEnv.from("Orders")
val result = orders.groupBy($"a").select($"a", $"b".sum().as("d"))
```

Java 代码:

```
Table orders = tableEnv.from("Orders");
Table result = orders.groupBy($"a").select($"a", $"b".sum().as("d"));
```

- ❑ **groupBy 窗口聚合:** 在组窗口上对表进行分组和聚合,可能还有一个或多个分组键。

Scala 代码:

```
val orders: Table = tableEnv.from("Orders")
val result: Table = orders
  .window(Tumble over 5.minutes on $"rowtime" as "w") // 定义窗口
  .groupBy($"a", $"w") // group by key 和 window
```

```
// 访问窗口属性和聚合
.select(
    $"a",
    $"w".start,
    $"w".end,
    $"w".rowtime,
    $"b".sum as "d")
```

Java 代码:

```
Table orders = tableEnv.from("Orders");
Table result = orders
    .window(Tumble.over(lit(5).minutes()).on($"rowtime")).as("w") // 定义窗口
    .groupBy($"a", $"w") // group by key 和 window
    // 访问窗口属性和聚合
    .select(
        $"a",
        $"w".start(),
        $"w".end(),
        $"w".rowtime(),
        $"b".sum().as("d")
    );
```

- ❑ Over 窗口聚合: 类似于 SQL OVER 子句。基于前一行和后一行的窗口(范围), 计算每一行的窗口聚合。

Scala 代码:

```
val orders: Table = tableEnv.from("Orders")
val result: Table = orders
    // 定义窗口
    .window(
        Over
        partitionBy $"a"
        orderBy $"rowtime"
        preceding UNBOUNDED_RANGE
        following CURRENT_RANGE
        as "w")
    // 滑动聚合
    .select(
        $"a",
        $"b".avg over $"w",
        $"b".max().over($"w"),
        $"b".min().over($"w")
    )
```

Java 代码:

```
Table orders = tableEnv.from("Orders");
Table result = orders
    // 定义窗口
    .window(
        Over
        .partitionBy($"a")
        .orderBy($"rowtime"))
```

```

        .preceding(UNBOUNDED_RANGE)
        .following(CURRENT_RANGE)
        .as("w"))
    // 滑动聚合
    .select(
        $("a"),
        $("b").avg().over($"w"),
        $("b").max().over($"w"),
        $("b").min().over($"w")
    );

```

所有聚合必须在相同的窗口上定义，即相同的分区、排序和范围。目前，只支持“PRECEDING (UNBOUNDED 和 BOUNDED)”到“CURRENT ROW”范围的窗口。目前还不支持 FOLLOWING 范围。ORDER BY 必须在单个时间属性上指定。

- ❑ distinct 聚合：类似于 SQL DISTINCT 聚合子句，如 COUNT(DISTINCT a)。Distinct 可以应用于 GroupBy 聚合、GroupBy 窗口聚合和 Over 窗口聚合。

Scala 代码：

```

val orders: Table = tableEnv.from("Orders");

// group by 上的 distinct 聚合
val groupByDistinctResult = orders
    .groupBy($"a")
    .select($"a", $"b".sum.distinct as "d")

// time window group by 上的 distinct 聚合
val groupByWindowDistinctResult = orders
    .window(Tumble over 5.minutes on $"rowtime" as "w").groupBy($"a", $"w")
    .select($"a", $"b".sum.distinct as "d")

// over window 上的 distinct 聚合
val result = orders
    .window(Over
        partitionBy $"a"
        orderBy $"rowtime"
        preceding UNBOUNDED_RANGE
        as $"w")
    .select($"a", $"b".avg.distinct over $"w", $"b".max over $"w", $"b".min over $"w")

```

Java 代码：

```

Table orders = tableEnv.from("Orders");

// group by 上的 distinct 聚合
Table groupByDistinctResult = orders
    .groupBy($"a")
    .select($"a", $"b".sum().distinct().as("d"));

// time window group by 上的 distinct 聚合
Table groupByWindowDistinctResult = orders
    .window(Tumble

```

```

        .over(lit(5).minutes())
        .on($"rowtime")
        .as("w")
    )
    .groupBy($"a", $"w")
    .select($"a", $"b".sum().distinct().as("d"));

// over window 上的 distinct 聚合
Table result = orders
    .window(Over
        .partitionBy($"a")
        .orderBy($"rowtime")
        .preceding(UNBOUNDED_RANGE)
        .as("w"))
    .select(
        $"a", $"b".avg().distinct().over($"w"),
        $"b".max().over($"w"),
        $"b".min().over($"w")
    );

```

- ❑ **distinct**: 类似于 SQL 的 distinct 子句, 返回具有唯一值组合的记录。

Scala 代码:

```

val orders: Table = tableEnv.from("Orders")
val result = orders.distinct()

```

Java 代码:

```

Table orders = tableEnv.from("Orders");
Table result = orders.distinct();

```

Join 连接

- ❑ **内连接**: 类似于 SQL 的 join 子句, 连接两个表。这两个表必须具有不同的字段名, 并且必须通过连接操作符或使用 where 或 filter 操作符定义至少一个相等连接谓词。

Scala 代码:

```

val left = ds1.toTable(tableEnv, $"a", $"b", $"c")
val right = ds2.toTable(tableEnv, $"d", $"e", $"f")
val result = left.join(right).where($"a" === $"d").select($"a", $"b", $"e")

```

Java 代码:

```

Table left = tableEnv.fromDataSet(ds1, "a, b, c");
Table right = tableEnv.fromDataSet(ds2, "d, e, f");
Table result = left.join(right)
    .where($"a".isEqual($"d"))
    .select($"a", $"b", $"e");

```

- ❑ **外连接**: 类似于 SQL 的 left/right/full outer join 子句, 连接两个表。这两个表必须具有不同的字段名, 并且必须定义至少一个相等连接谓词。

Scala 代码:

```

val left = tableEnv.fromDataSet(ds1, $"a", $"b", $"c")
val right = tableEnv.fromDataSet(ds2, $"d", $"e", $"f")

```

```
val leftOuterResult = left.leftOuterJoin(right, $"a" === $"d").select($"a", $"b", $"e")
val rightOuterResult = left.rightOuterJoin(right, $"a" === $"d").select($"a", $"b", $"e")
val fullOuterResult = left.fullOuterJoin(right, $"a" === $"d").select($"a", $"b", $"e")
```

Java 代码:

```
Table left = tableEnv.fromDataSet(ds1, "a, b, c");
Table right = tableEnv.fromDataSet(ds2, "d, e, f");

Table leftOuterResult = left.leftOuterJoin(right, $("a").isEqual($"d")).select($"a", $"b", $"e");
Table rightOuterResult = left.rightOuterJoin(right, $("a").isEqual($"d")).select($"a", $"b", $"e");
Table fullOuterResult = left.fullOuterJoin(right, $("a").isEqual($"d")).select($"a", $"b", $"e");
```

- 间隔连接: 间隔连接是常规连接的子集, 可以以流方式处理。间隔连接至少需要一个等效连接谓词和一个连接条件, 该连接条件限定了两端的时间。

Scala 代码:

```
val left = ds1.toTable(tableEnv, $"a", $"b", $"c", $"ltime".rowtime)
val right = ds2.toTable(tableEnv, $"d", $"e", $"f", $"rtime".rowtime)

val result = left.join(right)
  .where($"a" === $"d" && $"ltime" >= $"rtime" - 5.minutes && $"ltime" < $"rtime" + 10.minutes)
  .select($"a", $"b", $"e", $"ltime")
```

Java 代码:

```
Table left = tableEnv.fromDataSet(ds1, $("a"), $("b"), $("c"), $"ltime".rowtime());
Table right = tableEnv.fromDataSet(ds2, $("d"), $("e"), $("f"), $"rtime".rowtime());

Table result = left.join(right)
  .where(
    and(
      $("a").isEqual($"d"),
      $("ltime").isGreaterOrEqual($"rtime").minus(lit(5).minutes()),
      $("ltime").isLess($"rtime").plus(lit(10).minutes())
    )
  )
  .select($"a", $"b", $"e", $"ltime");
```

集合运算

- union: 类似于 SQL 中的 union 子句。合并两张表并删除重复记录。这两个表必须具有相同的字段类型。这个运算只作用于批处理。

Scala 代码:

```
val left = tableEnv.from("orders1")
val right = tableEnv.from("orders2")
```

```
left.union(right)
```

Java 代码:

```
Table left = tableEnv.from("orders1");
Table right = tableEnv.from("orders2");
```

```
left.union(right);
```

- ❑ **unionAll**: 类似于 SQL 中的 **union all** 子句。合并两张表。这两个表必须具有相同的字段类型。

Scala 代码:

```
val left = tableEnv.from("orders1")
val right = tableEnv.from("orders2")
```

```
left.unionAll(right)
```

Java 代码:

```
Table left = tableEnv.from("orders1");
Table right = tableEnv.from("orders2");
```

```
left.unionAll(right);
```

- ❑ **intersect**: 类似于 SQL 中的 **intersect** 子句。**Intersect** 返回两个表中都存在的记录。如果一条记录在一个或两个表中出现了不止一次, 它只返回一次, 也就是说, 生成的表中没有重复的记录。这两个表必须具有相同的字段类型。这个运算只作用于批处理。

Scala 代码:

```
val left = tableEnv.from("orders1")
val right = tableEnv.from("orders2")
```

```
left.intersect(right)
```

Java 代码:

```
Table left = tableEnv.from("orders1");
Table right = tableEnv.from("orders2");
```

```
left.intersect(right);
```

- ❑ **intersectAll**: 类似于 SQL 中的 **intersect all** 子句。**IntersectAll** 返回两个表中都存在的记录。如果一条记录在两个表中出现不止一次, 则返回的次数与它在两个表中出现的次数相同, 也就是说, 生成的表可能有重复的记录。这两个表必须具有相同的字段类型。这个运算只作用于批处理。

Scala 代码:

```
val left = tableEnv.from("orders1")
val right = tableEnv.from("orders2")
```

```
left.intersectAll(right)
```

Java 代码:

```
Table left = tableEnv.from("orders1");
Table right = tableEnv.from("orders2");
```

```
left.intersectAll(right);
```

- ❑ **minus**: 类似于 SQL 中的 **except** 子句。**Minus** 返回左表中存在但右表中不存在的记录。左表中的重复记录只返回一次, 即删除重复记录。这两个表必须具有相同的字段类型。这个运算只作用于批处理。

Scala 代码:

```
val left = tableEnv.from("orders1")
```

```
val right = tableEnv.from("orders2")
```

```
left.minus(right)
```

Java 代码:

```
Table left = tableEnv.from("orders1");
```

```
Table right = tableEnv.from("orders2");
```

```
left.minus(right);
```

- ❑ **minusAll**: 类似于 SQL 中的 **except all** 子句。MinusAll 返回不存在于右表中的记录。在左表中出现 n 次在右表中出现 m 次的记录被返回 $(n - m)$ 次, 也就是说, 在右表中出现的重复记录会被删除。这两个表必须具有相同的字段类型。这个运算只作用于批处理。

Scala 代码:

```
val left = tableEnv.from("orders1")
```

```
val right = tableEnv.from("orders2")
```

```
left.minusAll(right)
```

Java 代码:

```
Table left = tableEnv.from("orders1");
```

```
Table right = tableEnv.from("orders2");
```

```
left.minusAll(right);
```

- ❑ **in**: 类似于 SQL 中的 **in** 子句。如果表达式存在于给定的表子查询中, 则返回 **true**。子查询表必须包含一列。此列必须具有与表达式相同的数据类型。

Scala 代码:

```
val left = tableEnv.from("Orders1")
```

```
val right = tableEnv.from("Orders2");
```

```
val result = left.select($"a", $"b", $"c").where($"a".in(right))
```

Java 代码:

```
Table left = tableEnv.from("Orders1")
```

```
Table right = tableEnv.from("Orders2");
```

```
Table result = left.select($"a", $"b", $"c").where($"a".in(right));
```

OrderBy, Offset & Fetch

- ❑ **orderBy**: 类似于 SQL 中的 **order by** 子句。返回所有并行分区的全局排序记录。对于无界表, 此操作需要对时间属性或随后的获取操作进行排序。

Scala 代码:

```
val result = in.orderBy($"a".asc)
```

Java 代码:

```
Table result = in.orderBy($"a".asc());
```

- ❑ **offset & fetch**: 类似于 SQL 中的 **offset** 和 **fetch** 子句。偏移操作限制来自偏移位置的(可能已排序

的结果。fetch 操作将结果限制在前 n 行(可能已经排序)。通常,这两个操作的前面有一个排序操作符。对于无界表,偏移操作需要一个 fetch 操作。

Scala 代码:

```
// 返回排序结果的前 5 条记录
val result1: Table = in.orderBy($"a".asc).fetch(5)

// 跳过前 3 条记录并返回排序结果中所有后面的记录
val result2: Table = in.orderBy($"a".asc).offset(3)

// 跳过前 10 条记录并返回排序结果中的后 5 条记录
val result3: Table = in.orderBy($"a".asc).offset(10).fetch(5)
```

Java 代码:

```
// 返回排序结果的前 5 条记录
Table result1 = in.orderBy($"a".asc()).fetch(5);

// 跳过前 3 条记录并返回排序结果中所有后面的记录
Table result2 = in.orderBy($"a".asc()).offset(3);

// 跳过前 10 条记录并返回排序结果中的后 5 条记录
Table result3 = in.orderBy($"a".asc()).offset(10).fetch(5);
```

插入操作

与 SQL 查询中的 INSERT INTO 子句类似,该方法执行插入到注册输出表的操作。executeInsert() 方法将立即提交一个执行插入操作的 Flink 作业。

输出表必须在 TableEnvironment 中注册。而且,注册表的模式必须与查询的模式匹配。

Scala 代码:

```
val orders = tableEnv.from("Orders")
orders.executeInsert("OutOrders")
```

Java 代码:

```
Table orders = tableEnv.from("Orders");
orders.executeInsert("OutOrders");
```

5.3.2 窗口运算

分组窗口

分组窗口根据时间或行计数间隔将行聚合为有限组,并对每个组计算一次聚合函数。对于批处理表,窗口是按时间间隔对记录进行分组的方便快捷方式。

窗口是使用 window(w: GroupWindow)子句定义的,并且需要使用 as 子句指定的别名。为了根据窗口对表进行分组,必须在 groupBy(...)子句中引用窗口别名,就像常规的分组属性一样。下面的示例演示如何在表上定义窗口聚合。

Scala 代码:

```
val table = input
  .window([w: GroupWindow] as $"w") // 定义别名为 w 的窗口
  .groupBy($"w") // 将表按窗口 w 分组
  .select($"b".sum) // 聚合
```


Java 代码:

```
Table table = input
    .window([GroupWindow w].as("w")) // 定义别名为 w 的窗口
    .groupBy($"w") // 将表按窗口 w 分组
    .select($"b").sum(); // 聚合
```

在流环境中，窗口聚合只能在一个或多个属性上进行分组的情况下并行计算，例如，`groupBy(...)` 子句引用一个窗口别名和至少一个附加属性。只引用窗口别名的 `groupBy(...)` 子句(如上面的示例)只能由单个非并行任务计算。下面的示例显示如何定义带有附加分组属性的窗口聚合。

Scala 代码:

```
val table = input
    .window([w: GroupWindow] as $"w") // 定义别名为 w 的窗口
    .groupBy($"w", $"a") // 根据属性 a 和窗口 w 对表进行分组
    .select($"a", $"b".sum) // 聚合
```

Java 代码:

```
Table table = input
    .window([GroupWindow w].as("w")) // 定义别名为 w 的窗口
    .groupBy($"w", $"a") // 根据属性 a 和窗口 w 对表进行分组
    .select($"a", $"b").sum(); // 聚合
```

窗口属性，例如时间窗口的开始、结束或行时间戳，可以分别作为窗口别名 `w.start`、`w.end` 和 `w.rowtime` 的属性添加到 `select` 语句中。

Scala 代码:

```
val table = input
    .window([w: GroupWindow] as $"w") // 定义别名为 w 的窗口
    .groupBy($"w", $"a") // 根据属性 a 和窗口 w 对表进行分组
    // 聚合并增加窗口的开始、结束和行时间戳
    .select(
        $"a",
        $"w".start,
        $"w".end,
        $"w".rowtime,
        $"b".count
    )
```

Java 代码:

```
Table table = input
    .window([GroupWindow w].as("w")) // 定义别名为 w 的窗口
    .groupBy($"w", $"a") // 根据属性 a 和窗口 w 对表进行分组
    // 聚合并增加窗口的开始、结束和行时间戳
    .select(
        $"a",
        $"w".start(),
        $"w".end(),
        $"w".rowtime(),
        $"b".count()
    );
```

Window 参数定义如何将行映射到窗口。Window 不是一个用户可以实现的接口。相反，Table API 提供了一组具有特定语义的预定义 Window 类，这些类被转换为底层的 `DataStream` 或 `DataSet` 操作。下面列出了受支持的窗口定义。

滚动窗口

滚动窗口将行分配给不重叠的、固定长度的连续窗口。例如，一个 5 分钟的滚动窗口以 5 分钟间隔对行进行分组。滚动窗口可以在事件时间、处理时间或行数上定义。

Scala 代码：

```
// 滚动事件时间窗口
.window(Tumble over 10.minutes on $"rowtime" as $"w")

// 滚动处理时间窗口（假设有一个处理时间属性 "proctime"）
.window(Tumble over 10.minutes on $"proctime" as $"w")

// 滚动行计数窗口（假设有一个处理时间属性 "proctime"）
.window(Tumble over 10.rows on $"proctime" as $"w")
```

Java 代码：

```
// 滚动事件时间窗口
.window(Tumble.over(lit(10).minutes()).on($"rowtime")).as("w");

// 滚动处理时间窗口（假设有一个处理时间属性 "proctime"）
.window(Tumble.over(lit(10).minutes()).on($"proctime")).as("w");

// 滚动行计数窗口（假设有一个处理时间属性 "proctime"）
.window(Tumble.over(rowInterval(10)).on($"proctime")).as("w");
```

滚动窗口是通过使用 `Tumble` 类定义的。其中：

- ❑ **over**：将窗口的长度定义为时间或行计数间隔。
- ❑ **on**：将 `time` 属性分组(时间间隔)或排序(行数)。对于批处理查询，可以是任何 `Long` 或 `Timestamp` 属性。对于流查询，这必须是一个声明的事件时间或处理时间属性。
- ❑ **as**：给窗口分配别名。别名用于在下面的 `groupBy()` 子句中引用窗口，并可选地在 `select()` 子句中选择窗口属性，如窗口开始、结束或行时间戳。

滑动窗口

滑动窗口具有固定的大小，并按指定的滑动间隔滑动。如果滑动间隔小于窗口大小，则滑动窗口重叠。因此，行可以分配给多个窗口。例如，一个 15 分钟大小的滑动窗口和 5 分钟的滑动间隔将每一行分配给 3 个 15 分钟大小的不同窗口，这些窗口在 5 分钟的间隔内进行计算。滑动窗口可以在事件时间、处理时间或行数上定义。

Scala 代码：

```
// 滑动事件时间窗口
.window(Slide over 10.minutes every 5.minutes on $"rowtime" as $"w")

// 滑动处理时间窗口（假设有一个处理时间属性 "proctime"）
.window(Slide over 10.minutes every 5.minutes on $"proctime" as $"w")

// 滑动行计数窗口（假设有一个处理时间属性 "proctime"）
```

```
.window(Slide over 10.rows every 5.rows on $"proctime" as $"w")
```

Java 代码:

// 滑动事件时间窗口

```
.window(Slide.over(lit(10).minutes())
        .every(lit(5).minutes())
        .on($"rowtime"))
        .as("w");
```

// 滑动处理时间窗口 (假设有一个处理时间属性 "proctime")

```
.window(Slide.over(lit(10).minutes())
        .every(lit(5).minutes())
        .on($"proctime"))
        .as("w");
```

// 滑动行计数窗口(假设有一个处理时间属性 "proctime")

```
.window(Slide.over(rowInterval(10)).every(rowInterval(5)).on($"proctime")).as("w");
```

滑动窗口是通过使用 Slide 类定义的。其中:

- ☐ **over:** 将窗口的长度定义为时间或行计数间隔。
- ☐ **every:** 定义滑动间隔, 可以是时间间隔, 也可以是行计数间隔。滑动间隔必须与窗口大小间隔的类型相同。
- ☐ **on:** 将 time 属性分组(时间间隔)或排序(行数)。对于批处理查询, 可以是任何 Long 或 Timestamp 属性。对于流查询, 这必须是一个声明的事件时间或处理时间属性。
- ☐ **as:** 给窗口分配别名。别名用于在下面的 `groupBy()` 子句中引用窗口, 并可选地在 `select()` 子句中选择窗口属性, 如窗口开始、结束或行时间戳。

会话窗口

会话窗口没有固定的大小, 但它们的边界是由一个不活动的间隔定义的, 即, 如果在一个定义的间隔时间内没有事件出现, 会话窗口将关闭。例如, 当在 30 分钟不活动后观察到一行时(否则该行将被添加到现有的窗口), 一个有 30 分钟间隔的会话窗口开始, 如果 30 分钟内没有添加行, 则关闭该会话窗口。会话窗口可以在事件时间或处理时间上工作。

Scala 代码:

// 会话事件时间窗口

```
.window(Session withGap 10.minutes on $"rowtime" as $"w")
```

// 会话处理时间窗口(假设有一个处理时间属性 "proctime")

```
.window(Session withGap 10.minutes on $"proctime" as $"w")
```

Java 代码:

// 会话事件时间窗口

```
.window(Session.withGap(lit(10).minutes()).on($"rowtime")).as("w");
```

// 会话处理时间窗口(假设有一个处理时间属性 "proctime")

```
.window(Session.withGap(lit(10).minutes()).on($"proctime")).as("w");
```

会话窗口通过使用 Session 类定义, 如下所示:

- ☐ **withGap:** 将两个窗口之间的间隔定义为时间间隔。
- ☐ **on:** 将 time 属性分组(时间间隔)或排序(行数)。对于批处理查询, 可以是任何 Long 或 Timestamp 属性。对于流查询, 这必须是一个声明的事件时间或处理时间属性。

- ❑ **as**: 给窗口分配别名。别名用于在下面的 `groupBy()` 子句中引用窗口, 并可选地在 `select()` 子句中
选择窗口属性, 如窗口开始、结束或行时间戳。

Over 窗口

Over 窗口聚合来自标准 SQL (Over 子句), 并在查询的 SELECT 子句中定义。与在 `group by` 子句中指定的组窗口不同, over 窗口不会折叠行。相反, over 窗口聚合计算每个输入行在其相邻行范围内的聚合。

Over 窗口是使用 `window(w: OverWindow*)` 子句定义的, 并通过 `select()` 方法中的别名引用。下面的示例展示了如何在表上定义 over 窗口聚合。

Scala 代码:

```
val table = input
  .window([w: OverWindow] as $"w")           // 用别名 w 定义 over 窗口
  .select(                                     // 在 over 窗口 w 上聚合
    $"a",
    $"b".sum over $"w",
    $"c".min over $"w"
  )
```

Java 代码:

```
Table table = input
  .window([OverWindow w].as($"w"))           // 用别名 w 定义 over 窗口
  .select(                                     // 在 over 窗口 w 上聚合
    $"a",
    $"b".sum().over($"w"),
    $"c".min().over($"w")
  );
```

OverWindow 定义了计算聚合的行范围。Table API 提供了 Over 类来配置 over 窗口的属性。Over 窗口可以在事件时间或处理时间上定义, 也可以在指定为时间间隔或行计数的范围上定义。支持的 over 窗口定义作为 Over(和其他类)上的方法公开, 如下所示:

- ❑ **partitionBy**: 在一个或多个属性上定义输入的分区。对每个分区分别进行排序, 并对每个分区分别应用聚合函数。
- ❑ **orderBy**: 定义每个分区中的行顺序, 从而确定聚合函数应用于行的顺序。注意, 对于流查询, 这必须是一个声明的事件时间或处理时间属性。目前, 只支持单个排序属性。
- ❑ **preceding**: 定义包含在窗口中并位于当前行之前的行的间隔。间隔可以指定为时间间隔或行计数间隔。有界的窗口被指定为间隔的大小, 例如, 10.minutes 时间间隔或 10.rows 行计数间隔。无界窗口使用常量指定, 例如, UNBOUNDED_RANGE 作为一个时间间隔, UNBOUNDED_ROW 作为一个行计数间隔。无界窗口从分区的第一行开始。如果不指定 preceding 子句, 则 UNBOUNDED_RANGE 和 CURRENT_RANGE 将被用作窗口的默认 preceding 和 following。
- ❑ **following**: 定义包含在窗口中并跟随当前行的行的窗口间隔。间隔必须与 preceding 间隔的单位相同(时间或行计数)。目前, 不支持行紧跟在当前行的窗口。相反, 可以指定两个常量之一: CURRENT_ROW 设置窗口的上界为当前行, CURRENT_RANGE 设置窗口的上界来排序当前行的键, 即与当前行具有相同排序键的所有行都包含在窗口中。如果省略 following 子句, 则时间间隔窗口的上限定义为 CURRENT_RANGE, 行数间隔窗口的上限定义为 CURRENT_ROW。
- ❑ **as**: 给 over 窗口分配别名。别名用于在下面的 `select()` 子句中引用 over 窗口。目前, 同一个 `select()`

调用中的所有聚合函数都必须计算同一个 over 窗口。

无界 over 窗口

Scala 代码:

```
// 无界事件时间 over window (假设有一个事件时间属性"rowtime")
.window(Over partitionBy $"a" orderBy $"rowtime" preceding UNBOUNDED_RANGE as "w")

// 无界处理时间 over window (假设有一个处理时间属性"proctime")
.window(Over partitionBy $"a" orderBy $"proctime" preceding UNBOUNDED_RANGE as "w")

// 无界事件时间行计数 over window (假设有一个事件时间属性"rowtime")
.window(Over partitionBy $"a" orderBy $"rowtime" preceding UNBOUNDED_ROW as "w")

// 无界处理时间行计数 over window (假设有一个处理时间属性"proctime")
.window(Over partitionBy $"a" orderBy $"proctime" preceding UNBOUNDED_ROW as "w")
```

Java 代码:

```
// 无界事件时间 over window (假设有一个事件时间属性"rowtime")
.window(Over.partitionBy($"a").orderBy($"rowtime")).preceding(UNBOUNDED_RANGE).as("w");

// 无界处理时间 over window (假设有一个处理时间属性"proctime")
.window(Over.partitionBy($"a").orderBy("proctime").preceding(UNBOUNDED_RANGE).as("w"));

// 无界事件时间行计数 over window (假设有一个事件时间属性"rowtime")
.window(Over.partitionBy($"a").orderBy($"rowtime")).preceding(UNBOUNDED_ROW).as("w"));

// 无界处理时间行计数 over window (假设有一个处理时间属性"proctime")
.window(Over.partitionBy($"a").orderBy($"proctime")).preceding(UNBOUNDED_ROW).as("w"));
```

有界 over 窗口

Scala 代码:

```
// 有界事件时间 over window (假设有一个事件时间属性"rowtime")
.window(Over partitionBy $"a" orderBy $"rowtime" preceding 1.minutes as "w")

// 有界处理时间 over window (假设有一个处理时间属性"proctime")
.window(Over partitionBy $"a" orderBy $"proctime" preceding 1.minutes as "w")

// 有界事件时间行计数 over window (假设有一个事件时间属性"rowtime")
.window(Over partitionBy $"a" orderBy $"rowtime" preceding 10.rows as "w")

// 有界处理时间行计数 over window (假设有一个处理时间属性"proctime")
.window(Over partitionBy $"a" orderBy $"proctime" preceding 10.rows as "w")
```

Java 代码:

```
// 有界事件时间 over window (假设有一个事件时间属性"rowtime")
.window(Over.partitionBy($"a").orderBy($"rowtime")).preceding(lit(1).minutes()).as("w");

// 有界处理时间 over window (假设有一个处理时间属性"proctime")
.window(Over.partitionBy($"a").orderBy($"proctime")).preceding(lit(1).minutes()).as("w"));
```

```
// 有界事件时间行计数 over window (假设有一个事件时间属性"rowtime")
.window(Over.partitionBy($"a")).orderBy($"rowtime").preceding(rowInterval(10)).as("w");

// 有界处理时间行计数 over window (假设有一个处理时间属性"proctime")
.window(Over.partitionBy($"a")).orderBy($"proctime").preceding(rowInterval(10)).as("w");
```

5.3.3 基于行的操作

基于行的操作生成具有多列的输出。

- ❑ **map**: 使用用户定义的标量函数或内置标量函数执行 **map** 操作。如果输出类型是复合类型，则输出将被平展。

Scala 代码:

```
class MyMapFunction extends ScalarFunction {
  def eval(a: String): Row = {
    Row.of(a, "pre-" + a)
  }

  override def getResultType(signature: Array[Class[_]]): TypeInformation[_] =
    Types.ROW(Types.STRING, Types.STRING)
}

val func = new MyMapFunction()
val table = input.map(func($"c")).as("a", "b")
```

Java 代码:

```
public class MyMapFunction extends ScalarFunction {
  public Row eval(String a) {
    return Row.of(a, "pre-" + a);
  }

  @Override
  public TypeInformation<?> getResultType(Class<?>[] signature) {
    return Types.ROW(Types.STRING, Types.STRING);
  }
}

ScalarFunction func = new MyMapFunction();
tableEnv.registerFunction("func", func);

Table table = input.map(call("func", $"c")).as("a", "b");
```

- ❑ **flatMap**: 使用表函数执行 **flatMap** 操作。

Scala 代码:

```
class MyFlatMapFunction extends TableFunction[Row] {
  def eval(str: String): Unit = {
    if (str.contains("#")) {
      str.split("#").foreach({ s =>
        val row = new Row(2)
        row.setField(0, s)
      })
    }
  }
}
```

```

        row.setField(1, s.length)
        collect(row)
    })
}
}

override def getResultType: TypeInformation[Row] = {
    Types.ROW(Types.STRING, Types.INT)
}
}

```

```

val func = new MyFlatMapFunction
val table = input.flatMap(func($"c")).as("a", "b")

```

Java 代码:

```

public class MyFlatMapFunction extends TableFunction<Row> {

    public void eval(String str) {
        if (str.contains("#")) {
            String[] array = str.split("#");
            for (int i = 0; i < array.length; ++i) {
                collect(Row.of(array[i], array[i].length()));
            }
        }
    }

    @Override
    public TypeInformation<Row> getResultType() {
        return Types.ROW(Types.STRING, Types.INT);
    }
}

```

```

TableFunction func = new MyFlatMapFunction();
tableEnv.registerFunction("func", func);

```

```

Table table = input.flatMap(call("func", $"c")).as("a", "b");

```

- ❑ **aggregate:** 使用聚合函数执行聚合操作。必须用一个 `select` 语句来关闭“聚合”，而这个 `select` 语句不支持聚合函数。如果输出类型是复合类型，则聚合的输出将被平展。

Scala 代码:

```

case class MyMinMaxAcc(var min: Int, var max: Int)

class MyMinMax extends AggregateFunction[Row, MyMinMaxAcc] {

    def accumulate(acc: MyMinMaxAcc, value: Int): Unit = {
        if (value < acc.min) {
            acc.min = value
        }
        if (value > acc.max) {
            acc.max = value
        }
    }
}

```

```

    }
}

override def createAccumulator(): MyMinMaxAcc = MyMinMaxAcc(0, 0)

def resetAccumulator(acc: MyMinMaxAcc): Unit = {
    acc.min = 0
    acc.max = 0
}

override def getValue(acc: MyMinMaxAcc): Row = {
    Row.of(Integer.valueOf(acc.min), Integer.valueOf(acc.max))
}

override def getResultType: TypeInformation[Row] = {
    new RowTypeInfo(Types.INT, Types.INT)
}
}

val myAggFunc = new MyMinMax
val table = input
    .groupBy($"key")
    .aggregate(myAggFunc($"a") as ("x", "y"))
    .select($"key", $"x", $"y")

```

Java 代码:

```

public class MyMinMaxAcc {
    public int min = 0;
    public int max = 0;
}

public class MyMinMax extends AggregateFunction<Row, MyMinMaxAcc> {

    public void accumulate(MyMinMaxAcc acc, int value) {
        if (value < acc.min) {
            acc.min = value;
        }
        if (value > acc.max) {
            acc.max = value;
        }
    }

    @Override
    public MyMinMaxAcc createAccumulator() {
        return new MyMinMaxAcc();
    }

    public void resetAccumulator(MyMinMaxAcc acc) {
        acc.min = 0;
        acc.max = 0;
    }
}

```



```

@Override
public Row getValue(MyMinMaxAcc acc) {
    return Row.of(acc.min, acc.max);
}

@Override
public TypeInfo<Row> getResultType() {
    return new RowTypeInfo(Types.INT, Types.INT);
}
}

AggregateFunction myAggFunc = new MyMinMax();
tableEnv.registerFunction("myAggFunc", myAggFunc);
Table table = input
    .groupBy($"key")
    .aggregate(call("myAggFunc", $"a").as("x", "y"))
    .select($"key", $"x", $"y");

```

- ❑ **group window 聚合**：在组窗口（group window）上对表进行分组和聚合，可能有一个或多个分组键。必须用 select 语句关闭“聚合”。该 select 语句不支持“*”或聚合函数。

Scala 代码：

```

val myAggFunc = new MyMinMax
val table = input
    .window(Tumble over 5.minutes on $"rowtime" as "w") // 定义窗口
    .groupBy($"key", $"w") // 按 key 和窗口分组
    .aggregate(myAggFunc($"a") as ("x", "y"))
    .select($"key", $"x", $"y", $"w".start, $"w".end) // 访问窗口属性和聚合结果

```

Java 代码：

```

AggregateFunction myAggFunc = new MyMinMax();
tableEnv.registerFunction("myAggFunc", myAggFunc);

Table table = input
    .window(Tumble.over(lit(5).minutes()).on($"rowtime").as("w")) // 定义窗口
    .groupBy($"key", $"w") // 按 key 和窗口分组
    .aggregate(call("myAggFunc", $"a").as("x", "y"))
    .select($"key", $"x", $"y", $"w".start(), $"w".end()); // 访问窗口属性和聚合结果

```

- ❑ **flatAggregate**：类似于 groupBy 聚合。使用下面的运行表聚合操作符对分组键上的行进行分组，以按组对行进行聚合。与 AggregateFunction 的区别在于，TableAggregateFunction 可以为一个组返回 0 条或多条记录。必须用 select 语句关闭“flatAggregate”，该 select 语句不支持聚合函数。除了使用 emitValue 输出结果，还可以使用 emitUpdateWithRetract 方法。与 emitValue 不同，emitUpdateWithRetract 用于发出已更新的值。该方法在 retract 模式下以增量方式输出数据，也就是说，一旦有了更新，我们必须在发送新的更新记录之前撤销（retract）旧记录。如果两个方法都在表聚合函数中定义，则 emitUpdateWithRetract 方法将优先于 emitValue 方法，因为该方法被视为比 emitValue 更有效，因为它可以增量输出值。

Scala 代码：

```
import java.lang.{Integer => JInteger}
import org.apache.flink.table.api.Types
import org.apache.flink.table.functions.TableAggregateFunction

/**
 * 用于 top2 的累加器
 */
class Top2Accum {
  var first: JInteger = _
  var second: JInteger = _
}

/**
 * top2 用户定义的表聚合函数.
 */
class Top2 extends TableAggregateFunction[JTuple2[JInteger, JInteger], Top2Accum] {

  override def createAccumulator(): Top2Accum = {
    val acc = new Top2Accum
    acc.first = Int.MinValue
    acc.second = Int.MinValue
    acc
  }

  def accumulate(acc: Top2Accum, v: Int) {
    if (v > acc.first) {
      acc.second = acc.first
      acc.first = v
    } else if (v > acc.second) {
      acc.second = v
    }
  }

  def merge(acc: Top2Accum, its: JIterable[Top2Accum]): Unit = {
    val iter = its.iterator()
    while (iter.hasNext) {
      val top2 = iter.next()
      accumulate(acc, top2.first)
      accumulate(acc, top2.second)
    }
  }

  def emitValue(acc: Top2Accum, out: Collector[JTuple2[JInteger, JInteger]]): Unit = {
    // emit the value and rank
    if (acc.first != Int.MinValue) {
      out.collect(JTuple2.of(acc.first, 1))
    }
    if (acc.second != Int.MinValue) {
      out.collect(JTuple2.of(acc.second, 2))
    }
  }
}
```

```

    }
}

val top2 = new Top2
val orders: Table = tableEnv.from("Orders")
val result = orders
    .groupBy($"key")
    .flatAggregate(top2($"a") as ($"v", $"rank"))
    .select($"key", $"v", $"rank")

```

Java 代码:

```

/**
 * 用于 top2 的累加器
 */
public class Top2Accum {
    public Integer first;
    public Integer second;
}

/**
 * top2 用户定义的表聚合函数
 */
public class Top2 extends TableAggregateFunction<Tuple2<Integer, Integer>, Top2Accum> {

    @Override
    public Top2Accum createAccumulator() {
        Top2Accum acc = new Top2Accum();
        acc.first = Integer.MIN_VALUE;
        acc.second = Integer.MIN_VALUE;
        return acc;
    }

    public void accumulate(Top2Accum acc, Integer v) {
        if (v > acc.first) {
            acc.second = acc.first;
            acc.first = v;
        } else if (v > acc.second) {
            acc.second = v;
        }
    }

    public void merge(Top2Accum acc, java.lang.Iterable<Top2Accum> iterable) {
        for (Top2Accum otherAcc : iterable) {
            accumulate(acc, otherAcc.first);
            accumulate(acc, otherAcc.second);
        }
    }

    public void emitValue(Top2Accum acc, Collector<Tuple2<Integer, Integer>> out) {
        // emit the value and rank
    }
}

```

```
        if (acc.first != Integer.MIN_VALUE) {
            out.collect(Tuple2.of(acc.first, 1));
        }
        if (acc.second != Integer.MIN_VALUE) {
            out.collect(Tuple2.of(acc.second, 2));
        }
    }
}

tEnv.registerFunction("top2", new Top2());
Table orders = tableEnv.from("Orders");
Table result = orders
    .groupBy($"key")
    .flatAggregate(call("top2", $"a").as("v", "rank"))
    .select($"key", $"v", $"rank");
```

5. 4 Table API 与 DataStream API 集成

在定义数据处理管道时，Table API 和 DataStream API 同样重要。

DataStream API 在一个相对低级的命令编程 API 中提供了流处理(即时间、状态和数据流管理)的原语。Table API 抽象了许多内部元素，并提供了结构化和声明性的 API。这两种 API 都可以处理有界和无界流。Flink 提供了特殊的桥接功能，使与 DataStream API 的集成尽可能顺利。

5.4.1 依赖

组合 Table API 和 DataStream API 的项目需要添加以下桥接模块之一。它们包括到 flink-table-api-java 或 flink-table-api-scala 和相应的特定于语言的 DataStream API 模块的传递依赖。

```
<!-- 如果是使用 Scala -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-scala-bridge_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>

<!-- 如果是使用 Java -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-java-bridge_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
```

要使用 DataStream API 和 Table API 的 Java 或 Scala 版本声明公共管道，需要以下导入。

Scala 导入：

```
// Scala DataStream API
```

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

// 桥接 Scala DataStream API 的 Table API 导入
import org.apache.flink.table.api._
import org.apache.flink.table.api.bridge.scala._

Java 导入:
// Java DataStream API
import org.apache.flink.streaming.api.*;
import org.apache.flink.streaming.api.environment.*;

// 桥接 Java DataStream API 的 Table API 导入
import org.apache.flink.table.api.*;
import org.apache.flink.table.api.bridge.java.*;
```

5.4.2 在 DataStream 和 Table 之间转换

Flink 提供了一个专门的 `StreamTableEnvironment`（在 Java 和 Scala 中），用于与 `DataStream` API 集成。这些环境使用额外的方法扩展常规的 `TableEnvironment`，并将 `DataStream` API 中使用的 `StreamExecutionEnvironment` 作为参数。

下面的代码展示了如何在两个 API 之间来回切换的示例。`Table` 的列名和类型是由 `DataStream` 的 `TypeInfo` 自动派生的。由于 `DataStream` API 不支持在本地更改日志（changelog）处理，因此在流到表和表到流的转换期间，代码假定 `append-only/insert-only` 语义。

Scala 代码：

```
package com.xueai8.scala.ch04

import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala.StreamTableEnvironment

/**
 * 在 DataStream 和 Table 之间转换
 *
 * 下面的代码展示了如何在两个 API 之间来回切换的示例。
 * Table 的列名和类型是由 DataStream 的 TypeInfo 自动派生的。
 */
object TableDemo01 {
  def main(args: Array[String]) {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // 创建流表执行环境
    val tableEnv = StreamTableEnvironment.create(env)

    // 创建 DataStream
    val dataStream = env.fromElements("张三", "李四", "王老五")

    // 将 insert-only DataStream 解释为一个 Table
```

```
val inputTable = tableEnv.fromDataStream(dataStream)

// 将 Table 对象注册为视图并查询它
tableEnv.createTemporaryView("InputTable", inputTable)
val resultTable = tableEnv.sqlQuery("SELECT f0,CHARACTER_LENGTH(f0) FROM InputTable")

// 再次将 insert-only Table 解释为一个 DataStream
val resultStream = tableEnv.toDataStream(resultTable)

// print sink
resultStream.print()

// 在 DataStream API 中执行
env.execute("mix DataStream & Table API")
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.types.Row;

/**
 * 在 DataStream 和 Table 之间转换
 *
 * 下面的代码展示了如何在两个 API 之间来回切换的示例。
 * Table 的列名和类型是由 DataStream 的 TypeInformation 自动派生的。
 */
public class TableDemo01 {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        // 创建流表执行环境
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // 创建 DataStream
        DataStream<String> dataStream = env.fromElements("张三", "李四", "王老五");

        // 将 insert-only DataStream 解释为一个 Table
        Table inputTable = tableEnv.fromDataStream(dataStream);

        // 将 Table 对象注册为视图并查询它
        tableEnv.createTemporaryView("InputTable", inputTable);
        Table resultTable = tableEnv.sqlQuery("SELECT f0,CHARACTER_LENGTH(f0) FROM InputTable");

        // 再次将 insert-only Table 解释为一个 DataStream
```

```

    DataStream<Row> resultStream = tableEnv.toDataStream(resultTable);

    // print sink
    resultStream.print();

    // 在 DataStream API 中执行
    env.execute();
}

```

在将 Table 转换为 DataStream 或 DataSet 时，需要指定结果 DataStream 或 DataSet 的数据类型，即将 Table 的行转换为 DataStream 的数据类型。通常最方便的转换类型是 Row。以下列表概述了不同选项的特性：

- ❑ Row: 字段按位置映射，任意数量的字段，支持 null 值，没有类型安全的访问。
- ❑ POJO: 字段按名称映射(POJO 字段必须命名为表字段)，任意数量的字段，支持 null 值，类型安全的访问。
- ❑ Case Class: 字段按位置映射，不支持 null 值，类型安全访问。
- ❑ Tuple: 字段按位置映射，限制为 22 (Scala)或 25 (Java)字段，不支持 null 值，类型安全访问。
- ❑ Atomic Type: 表必须有一个单独的字段，不支持 null 值，类型安全访问

执行上面的代码，输出结果如下：

```

7> +[王老五, 3]
5> +[张三, 2]
6> +[李四, 2]

```

下面的示例展示了如何转换更新表。每个结果行表示一个更改日志（changelog）中的一个条目，该条目具有更改标志，可以通过对其调用 row.getKind()进行查询。在这个示例中，张三的第二个分数创建了 update before(-U)和 update after(+U)更改。

Scala 代码：

```

package com.xueai8.scala.ch04

import org.apache.flink.api.scala.typeutils.Types
import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.bridge.scala.StreamTableEnvironment
import org.apache.flink.types.Row

/**
 * 在 DataStream 和 Table 之间转换
 *
 * 下面的示例展示了如何转换更新表。
 * 每个结果行表示一个更改日志（changelog）中的一个条目，该条目具有更改标志，
 * 可以通过对其调用 row.getKind()进行查询。
 * 在这个示例中，张三的第二个分数创建了 update before(-U)和 update after(+U)更改。
 */
object TableDemo02 {
  def main(args: Array[String]) {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // 创建流表执行环境

```

```
val tableEnv = StreamTableEnvironment.create(env)

// 创建 DataStream
val dataStream = env.fromElements(
    Row.of("张三", Int.box(12)),
    Row.of("李四", Int.box(10)),
    Row.of("张三", Int.box(100))
)(Types.ROW(Types.STRING, Types.INT))

// 将 insert-only DataStream 解释为一个 Table
val inputTable = tableEnv.fromDataStream(dataStream).as("name", "score")

// 将 Table 对象注册为视图并查询它
tableEnv.createTemporaryView("InputTable", inputTable)
val resultTable = tableEnv.sqlQuery("SELECT name, SUM(score) FROM InputTable GROUP BY name")

// 再次将 insert-only Table 解释为一个 DataStream
val resultStream = tableEnv.toChangelogStream(resultTable)

// print sink
resultStream.print()

// 在 DataStream API 中执行
env.execute("mix DataStream & Table API")
}
```

Java 代码:

```
package com.xueai8.java.ch04;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.types.Row;

/**
 * 在 DataStream 和 Table 之间转换
 *
 * 下面的代码展示了如何在两个 API 之间来回切换的示例。
 * Table 的列名和类型是由 DataStream 的 TypeInformation 自动派生的。
 */
public class TableDemo02 {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        // 创建流表执行环境
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // 创建 DataStream
```



```
DataStream<Row> dataStream = env.fromElements(
    Row.of("张三", 12),
    Row.of("李四", 10),
    Row.of("张三", 100));

// 将 insert-only DataStream 解释为一个 Table
Table inputTable = tableEnv.fromDataStream(dataStream).as("name", "score");

// 将 Table 对象注册为视图并查询它
tableEnv.createTemporaryView("InputTable", inputTable);
Table resultTable = tableEnv.sqlQuery("SELECT name, SUM(score) FROM InputTable GROUP BY
name");

// 再次将 insert-only Table 解释为一个 DataStream
DataStream<Row> resultStream = tableEnv.toChangelogStream(resultTable);

// print sink
resultStream.print();

// 在 DataStream API 中执行
env.execute();
}
```

执行以上代码，输出结果如下：

```
5> +[李四, 10]
2> +[张三, 12]
2> -U[张三, 12]
2> +U[张三, 112]
```

配置

TableEnvironment 将采用来自传入的 StreamExecutionEnvironment 的所有配置选项。然而，不能保证对 StreamExecutionEnvironment 配置的进一步更改会在实例化后传播到 StreamTableEnvironment。另外，也不支持将选项从 Table API 反向传播到 DataStream API。

建议在切换到 Table API 之前尽早设置 DataStream API 中的所有配置选项。例如。

Scala 代码：

```
import java.time.ZoneId
import org.apache.flink.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.CheckpointingMode
import org.apache.flink.table.api.bridge.scala._

// 创建 Scala DataStream API
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 尽早设置各种配置

env.setMaxParallelism(256)
```

```
env.getConfig.addDefaultKryoSerializer(classOf[MyCustomType], classOf[CustomKryoSerializer])

env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE)

// 然后切换到 Scala Table API

val tableEnv = StreamTableEnvironment.create(env)

// 尽早设置配置

tableEnv.getConfig.setLocalTimeZone(Zoneld.of("Europe/Berlin"))

// 开始以这两种 API 定义管道.....
```

Java 代码:

```
import java.time.Zoneld;
import org.apache.flink.streaming.api.CheckpointingMode;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

// 创建 Scala DataStream API

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

// 尽早设置各种配置

env.setMaxParallelism(256);

env.getConfig().addDefaultKryoSerializer(MyCustomType.class, CustomKryoSerializer.class);

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

// 然后切换到 Scala Table API

StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

// 尽早设置配置

tableEnv.getConfig().setLocalTimeZone(Zoneld.of("Europe/Berlin"));

// 开始以这两种 API 定义管道.....
```

5.4.3 处理 insert-only 流

StreamTableEnvironment 提供了以下方法来进行 DataStream API 转换:

- ❑ `fromDataStream(DataStream)`: 将 insert-only 更改流和任意类型解释为表。默认情况下不传播事件时间和水印。
- ❑ `fromDataStream(DataStream, Schema)`: 将 insert-only 更改流和任意类型解释为表。可选模式允许丰富列数据类型, 并添加时间属性、水印策略、其他计算列或主键。

- ❑ `createTemporaryView(String, DataStream)`: 给流注册一个名字以便在 SQL 中访问它。它是 `createTemporaryView(String, fromDataStream(DataStream))` 的快捷方式。
- ❑ `createTemporaryView(String, DataStream, Schema)`: 给流注册一个名字以便在 SQL 中访问它。Registers the stream under a name to access it in SQL。它是 `createTemporaryView(String, fromDataStream(DataStream, Schema))` 的快捷方式。
- ❑ `toDataStream(DataStream)`: 将表转换为 insert-only 更改流。默认流记录类型为 `org.apache.flink.types.Row`。一个单独的 rowtime 属性列被写回到 DataStream API 的记录中。水印也可以传播。
- ❑ `toDataStream(DataStream, AbstractDataType)`: 将表转换为 insert-only 更改流。此方法接受一个数据类型以表示所需的流记录类型。规划器 (planner) 可以插入隐式强制转换并对列重新排序, 以将列映射到(可能嵌套的)数据类型的字段。
- ❑ `toDataStream(DataStream, Class)`: `toDataStream(DataStream, DataTypes.of(Class))` 的快捷方式, 以快速反射地创建所需的数据类型。

从表 API 的角度来看, 从 DataStream API 转换和转换到 DataStream API 类似于从使用 SQL 中的 CREATE TABLE DDL 定义的虚拟表连接器读取或写入。

虚拟 CREATE TABLE name (schema) WITH (options) 语句中的模式部分可以自动从 DataStream 的类型信息派生出来, 丰富或者完全使用 `org.apache.flink.table.api.Schema` 手工定义。

虚拟 DataStream 表连接器为每一行公开以下元数据:

Key	Data Type	Description	R/W
rowtime	TIMESTAMP_LTZ(3) NOT NULL	流记录的时间戳	R/W

虚拟 DataStream 表源实现了 `SupportsSourceWatermark`, 因此允许调用 `SOURCE_WATERMARK()` 内置函数作为水印策略, 以从 DataStream API 采用水印。

下面的代码展示了如何在不同的场景中使用 `fromDataStream`。

Scala 代码:

```
package com.xueai8.scala.ch05

import java.time.Instant

import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.Schema
import org.apache.flink.table.api.bridge.scala.StreamTableEnvironment

/**
 * Table API
 *
 * 下面的代码展示了如何在不同的场景中使用 fromDataStream。
 */
object TableDemo03 {

  // case class
  case class User(name: String, score: Integer, event_time: Instant)

  def main(args: Array[String]) {
    // 设置流执行环境
```

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
// 创建流表执行环境
val tableEnv = StreamTableEnvironment.create(env)

// 创建一个 DataStream
val dataStream = env.fromElements(
    User("张三", 4, Instant.ofEpochMilli(1000)),
    User("李四", 6, Instant.ofEpochMilli(1001)),
    User("张三", 10, Instant.ofEpochMilli(1002)))

// === EXAMPLE 1 ===
// 自动导出所有物理列
val table1 = tableEnv.fromDataStream(dataStream)
table1.printSchema()

// === EXAMPLE 2 ===
// 自动导出所有物理列
// 但是添加计算列(在本例中用于创建 proctime 属性列)
val table2 = tableEnv.fromDataStream(
    dataStream,
    Schema.newBuilder()
        .columnByExpression("proc_time", "PROCTIME()")
        .build())
table2.printSchema()

// === EXAMPLE 3 ===
// 自动导出所有物理列
// 但是添加计算列(在本例中用于创建 rowtime 属性列)
// 和自定义水印策略
val table3 = tableEnv.fromDataStream(
    dataStream,
    Schema.newBuilder()
        .columnByExpression("rowtime", "CAST(event_time AS TIMESTAMP_LTZ(3))")
        .watermark("rowtime", "rowtime - INTERVAL '10' SECOND")
        .build())
table3.printSchema()

// === EXAMPLE 4 ===
// 自动导出所有物理列
// 但是访问流记录的时间戳以创建一个 rowtime 属性列
// 也依赖于 DataStream API 中生成的水印
// 我们假设之前已经为 "dataStream" 定义了水印策略(不是本例的一部分)
val table4 = tableEnv.fromDataStream(
    dataStream,
    Schema.newBuilder()
        .columnByMetadata("rowtime", "TIMESTAMP_LTZ(3)")
        .watermark("rowtime", "SOURCE_WATERMARK()")
        .build())
table4.printSchema()
```

```
// === EXAMPLE 5 ===
// 手动定义物理列
// 在本例中,
// - 我们可以将时间戳的默认精度从 9 降低到 3
// - 我们还投影列并将 'event_time' 放在开头
val table5 = tableEnv.fromDataStream(
    dataStream,
    Schema.newBuilder()
        .column("event_time", "TIMESTAMP_LTZ(3)")
        .column("name", "STRING")
        .column("score", "INT")
        .watermark("event_time", "SOURCE_WATERMARK()")
        .build()
    table5.printSchema()
}
```

Java 代码:

```
package com.xueai8.java.ch05;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Schema;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

import java.time.Instant;

/**
 * Table API
 *
 * 下面的代码展示了如何在不同的场景中使用 fromDataStream。
 */
public class TableDemo03 {

    // POJO 类
    public static class User {
        public String name;
        public Integer score;
        public Instant event_time;

        // DataStream API 的默认构造函数
        public User() {
        }

        // 表 API 的完全赋值构造函数
        public User(String name, Integer score, Instant event_time) {
            this.name = name;
            this.score = score;
            this.event_time = event_time;
        }
    }
}
```

```
}  
}  
  
public static void main(String[] args) throws Exception {  
    // 设置流执行环境  
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  
    // 创建流表执行环境  
    StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);  
  
    // 创建一个 DataStream  
    DataStream<User> dataStream = env.fromElements(  
        new User("张三", 4, Instant.ofEpochMilli(1000)),  
        new User("李四", 6, Instant.ofEpochMilli(1001)),  
        new User("张三", 10, Instant.ofEpochMilli(1002))  
    );  
  
    // === EXAMPLE 1 ===  
    // 自动导出所有物理列  
    Table table1 = tableEnv.fromDataStream(dataStream);  
    table1.printSchema();  
  
    // === EXAMPLE 2 ===  
    // 自动导出所有物理列  
    // 但是添加计算列(在本例中用于创建 proctime 属性列)  
    Table table2 = tableEnv.fromDataStream(  
        dataStream,  
        Schema.newBuilder()  
            .columnByExpression("proc_time", "PROCTIME()")  
            .build()  
    );  
    table2.printSchema();  
  
    // === EXAMPLE 3 ===  
    // 自动导出所有物理列  
    // 但是添加计算列(在本例中用于创建 rowtime 属性列)  
    // 和自定义水印策略  
    Table table3 = tableEnv.fromDataStream(  
        dataStream,  
        Schema.newBuilder()  
            .columnByExpression("rowtime", "CAST(event_time AS TIMESTAMP_LTZ(3))")  
            .watermark("rowtime", "rowtime - INTERVAL '10' SECOND")  
            .build()  
    );  
    table3.printSchema();  
  
    // === EXAMPLE 4 ===  
    // 自动导出所有物理列  
    // 但是访问流记录的时间戳以创建一个 rowtime 属性列  
    // 也依赖于 DataStream API 中生成的水印  
    // 我们假设之前已经为“dataStream”定义了水印策略(不是本例的一部分)
```

```

        Table table4 = tableEnv.fromDataStream(
            dataStream,
            Schema.newBuilder()
                .columnByMetadata("rowtime", "TIMESTAMP_LTZ(3)")
                .watermark("rowtime", "SOURCE_WATERMARK()")
                .build()
        );
        table4.printSchema();

        // === EXAMPLE 5 ===
        // 手动定义物理列
        // 在本例中,
        // - 我们可以将时间戳的默认精度从 9 降低到 3
        // - 我们还投影列并将 'event_time' 放在开头
        Table table5 = tableEnv.fromDataStream(
            dataStream,
            Schema.newBuilder()
                .column("event_time", "TIMESTAMP_LTZ(3)")
                .column("name", "STRING")
                .column("score", "INT")
                .watermark("event_time", "SOURCE_WATERMARK()")
                .build()
        );
        table5.printSchema();
    }
}

```

`DataStream` 可以直接注册为一个视图(可能使用模式进行充实)。从 `DataStream` 创建的视图只能注册为临时视图。由于它们的内联/匿名性质, 不可能在永久目录中注册它们。

下面的代码展示了如何在不同的场景中使用 `createTemporaryView`。

Scala 代码:

```

package com.xueai8.scala.ch05

import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.Schema
import org.apache.flink.table.api.bridge.scala.StreamTableEnvironment

/**
 * Table API
 *
 * 下面的代码展示了如何在不同的场景中使用 createTemporaryView。
 */
object TableDemo04 {

    def main(args: Array[String]) {
        // 设置流执行环境
        val env = StreamExecutionEnvironment.getExecutionEnvironment
        // 创建流表执行环境
        val tableEnv = StreamTableEnvironment.create(env)
    }
}

```

```

// 创建一个 DataStream
val dataStream: DataStream[(Long, String)] = env.fromElements((12L, "Alice"),(0L, "Bob"))

// === EXAMPLE 1 ===
// 将 DataStream 注册为当前会话中的视图 MyView
// 所有列都是自动派生的
tableEnv.createTemporaryView("MyView1", dataStream)
tableEnv.from("MyView1").printSchema()

// === EXAMPLE 2 ===
// 将 DataStream 注册为当前会话中的视图 MyView
// 提供类似于 'fromDataStream' 的模式来调整列
// 在本例中，派生的 NOT NULL 信息已被删除
tableEnv.createTemporaryView(
    "MyView2",
    dataStream,
    Schema.newBuilder()
        .column("_1", "BIGINT")
        .column("_2", "STRING")
        .build())

tableEnv.from("MyView2").printSchema()

// === EXAMPLE 3 ===
// 如果只是关于重命名列，则在创建视图之前使用 Table API
tableEnv.createTemporaryView(
    "MyView3",
    tableEnv.fromDataStream(dataStream).as("id", "name"))

tableEnv.from("MyView3").printSchema()
}
}

```

Java 代码:

```

package com.xueai8.java.ch05;

import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Schema;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

/**
 * Table API
 *
 * 下面的代码展示了如何在不同的场景中使用 createTemporaryView。
 */
public class TableDemo04 {

    public static void main(String[] args) throws Exception {

```



```
// 设置流执行环境
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
// 创建流表执行环境
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

// 创建一个 DataStream
DataStream<Tuple2<Long, String>> dataStream = env.fromElements(
    Tuple2.of(12L, "张三"),
    Tuple2.of(0L, "李四"));

// === EXAMPLE 1 ===
// 将 DataStream 注册为当前会话中的视图 MyView
// 所有列都是自动派生的
tableEnv.createTemporaryView("MyView", dataStream);
tableEnv.from("MyView").printSchema();

// === EXAMPLE 2 ===
// 将 DataStream 注册为当前会话中的视图 MyView
// 提供类似于 'fromDataStream' 的模式来调整列
// 在本例中，派生的 NOT NULL 信息已被删除
tableEnv.createTemporaryView(
    "MyView2",
    dataStream,
    Schema.newBuilder()
        .column("f0", "BIGINT")
        .column("f1", "STRING")
        .build());

tableEnv.from("MyView2").printSchema();

// === EXAMPLE 3 ===
// 如果只是关于重命名列，则在创建视图之前使用 Table API
tableEnv.createTemporaryView(
    "MyView3",
    tableEnv.fromDataStream(dataStream).as("id", "name"));

tableEnv.from("MyView3").printSchema();
}
```

下面的代码展示了如何在不同的场景中使用 `toDataStream`。

Scala 代码：

```
package com.xueai8.scala.ch05

import org.apache.flink.streaming.api.scala._
import org.apache.flink.table.api.DataTypes
import org.apache.flink.table.api.bridge.scala.StreamTableEnvironment
import org.apache.flink.types.Row
```

```
/**
 * Table API
 *
 * 下面的代码展示了如何在不同的场景中使用 toDataStream。
 */
object TableDemo05 {

  case class User(name: String, score: java.lang.Integer, event_time: java.time.Instant)

  def main(args: Array[String]) {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // 创建流表执行环境
    val tableEnv = StreamTableEnvironment.create(env)

    tableEnv.executeSql(
      """
        CREATE TABLE GeneratedTable (
          name STRING,
          score INT,
          event_time TIMESTAMP_LTZ(3),
          WATERMARK FOR event_time AS event_time - INTERVAL '10' SECOND
        )
        WITH ('connector'='datagen')
      """
    )

    val table = tableEnv.from("GeneratedTable")

    // === EXAMPLE 1 ===
    // 使用默认的 Row 实例转换
    // 因为' event_time '是一个单独的 rowtime 属性，所以它被插入到 DataStream 中
    // 传播元数据和水印
    val dataStream1: DataStream[Row] = tableEnv.toDataStream(table)

    // === EXAMPLE 2 ===
    // 从类' User '中提取数据类型，
    // 规划器对字段重新排序，并在可能的地方插入隐式强制转换，以便将内部数据结构转换为所需的结构化类型
    // 因为' event_time '是一个单独的 rowtime 属性，所以它被插入到 DataStream 中
    // 传播元数据和水印
    val dataStream2: DataStream[User] = tableEnv.toDataStream(table, classOf[User])

    // 数据类型可以如上所示进行反射提取或显式定义
    val dataStream3: DataStream[User] =
      tableEnv.toDataStream(
        table,
        DataTypes.STRUCTURED(
          classOf[User],
          DataTypes.FIELD("name", DataTypes.STRING()),
          DataTypes.FIELD("score", DataTypes.INT()),

```

```
        DataTypes.FIELD("event_time", DataTypes.TIMESTAMP_LTZ(3)))
    }
}
```

Java 代码:

```
package com.xueai8.java.ch05;

import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.table.api.DataTypes;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.types.Row;
import java.time.Instant;

/**
 * Table API
 *
 * 下面的代码展示了如何在不同的场景中使用 toDataStream。
 */
public class TableDemo05 {

    // 带有可变字段的 POJO
    // 由于没有定义完全赋值的构造函数，字段顺序按字母顺序[event_time, name, score]
    public static class User {

        public String name;
        public Integer score;
        public Instant event_time;
    }

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        // 创建流表执行环境
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        tableEnv.executeSql(
            "CREATE TABLE GeneratedTable "
                + "("
                + "  name STRING,"
                + "  score INT,"
                + "  event_time TIMESTAMP_LTZ(3),"
                + "  WATERMARK FOR event_time AS event_time - INTERVAL '10' SECOND"
                + ")"
            + "WITH ('connector'='datagen');");

        Table table = tableEnv.from("GeneratedTable");

        // === EXAMPLE 1 ===
        // 使用默认的 Row 实例转换
    }
}
```

```
// 因为' event_time '是一个单独的 rowtime 属性，所以它被插入到 DataStream 中
// 传播元数据和水印
DataStream<Row> dataStream1 = tableEnv.toDataStream(table);

// === EXAMPLE 2 ===

// 从类' User '中提取数据类型，
// 规划器对字段重新排序，并在可能的地方插入隐式强制转换，以便将内部数据结构转换为所需的结构
化类型
// 因为' event_time '是一个单独的 rowtime 属性，所以它被插入到 DataStream 中
// 传播元数据和水印
DataStream<User> dataStream2 = tableEnv.toDataStream(table, User.class);

// 数据类型可以如上所示进行反射提取或显式定义
DataStream<User> dataStream3 =
    tableEnv.toDataStream(
        table,
        DataTypes.STRUCTURED(
            User.class,
            DataTypes.FIELD("name", DataTypes.STRING()),
            DataTypes.FIELD("score", DataTypes.INT()),
            DataTypes.FIELD("event_time", DataTypes.TIMESTAMP_LTZ(3))));
}
```

注意，toDataStream 只支持非更新表。通常，基于时间的操作(如窗口、间隔连接或 MATCH_RECOGNIZE 子句)非常适用于紧挨着投影和过滤器等简单操作的 insert-only 管道。

带有生成更新操作的管道可以使用到 toChangelogStream。

5.4.4 处理变更日志流

在内部，Flink 的表运行时是一个变更日志处理器(changelog processor)。

StreamTableEnvironment 提供了以下方法来公开这些 change data capture (CDC，变更数据捕获)功能：

- ❑ fromChangelogStream(DataStream)：将变更日志条目流解释为一个表。流记录类型必须是 org.apache.flink.types.Row，因为它的 RowKind 标志是在运行时计算的。默认情况下不传播事件时间和水印。该方法期望一个包含所有类型变更(在 org.apache.flink.types.RowKind 中枚举)的变更日志作为默认的 ChangelogMode。
- ❑ fromChangelogStream(DataStream, Schema)：允许为 DataStream 定义一个模式。
- ❑ fromChangelogStream(DataStream, Schema, ChangelogMode)：给出了如何将流解释为变更日志的完全控制。传递的 ChangelogMode 参数帮助规划器区分 insert-only、upsert 或 retract 行为。
- ❑ toChangelogStream(Table)：生成一个具有 org.apache.flink.types.Row 实例的流，并在运行时为每条记录设置 RowKind 标志。该方法支持各种类型的更新表。如果输入表包含一个 rowtime 列，它将被传播到流记录的时间戳中。水印也将被传播。
- ❑ toChangelogStream(Table, Schema)：该方法可以丰富生成的列数据类型。如果有必要，规划器可以插入隐式强制类型转换。它可以将 rowtime 写成元数据列。

- ❑ `toChangelogStream(Table, Schema, ChangelogMode)`: 提供关于如何将表转换为变更日志流的完全控制。传递的 `ChangelogMode` 帮助规划器区分 `insert-only`、`upsert` 或 `retract` 行为。

从 Table API 的角度来看, 和 `DataStream` API 的相互转换类似于从使用 SQL 中的 `CREATE TABLE` DDL 定义的虚拟表连接器读取或写入。这个虚拟连接器还支持读写流记录的 `rowtime` 元数据。虚表源实现了 `SupportsSourceWatermark`。

下面的代码展示了如何在不同的场景中使用 `fromChangelogStream`。

Scala 代码:

```
package com.xueai8.scala.ch05

import org.apache.flink.api.scala.typeutils.Types
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.table.api.Schema
import org.apache.flink.table.api.bridge.scala.StreamTableEnvironment
import org.apache.flink.table.connector.ChangelogMode
import org.apache.flink.types.{Row, RowKind}

/**
 * Table API
 *
 * 下面的代码展示了如何在不同的场景中使用 fromChangelogStream。
 */
object TableDemo06 {

  def main(args: Array[String]) {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // 创建流表执行环境
    val tableEnv = StreamTableEnvironment.create(env)

    // === EXAMPLE 1 ===
    // 将此流解释为 retract 流
    // 创建一个 changelog DataStream
    val dataStream1 = env.fromElements(
      Row.ofKind(RowKind.INSERT, "张三", Int.box(12)),
      Row.ofKind(RowKind.INSERT, "李四", Int.box(5)),
      Row.ofKind(RowKind.UPDATE_BEFORE, "张三", Int.box(12)),
      Row.ofKind(RowKind.UPDATE_AFTER, "张三", Int.box(100))
    )(Types.ROW(Types.STRING, Types.INT))

    // 将 DataStream 解释为一个 Table
    val table1 = tableEnv.fromChangelogStream(dataStream1)

    // 注册该 table, 并执行一个聚合
    tableEnv.createTemporaryView("InputTable1", table1)
    tableEnv
      .executeSql("SELECT f0 AS name, SUM(f1) AS score FROM InputTable1 GROUP BY f0")
      .print()
  }
}
```

```
// === EXAMPLE 2 ===
// 将流解释为 upsert 流(不需要 UPDATE_BEFORE)
// 创建一个 changelog DataStream
val dataStream2 = env.fromElements(
    Row.ofKind(RowKind.INSERT, "张三", Int.box(12)),
    Row.ofKind(RowKind.INSERT, "李四", Int.box(5)),
    Row.ofKind(RowKind.UPDATE_AFTER, "张三", Int.box(100))
)(Types.ROW(Types.STRING, Types.INT))

// 将 DataStream 解释为一个 Table
val table2 = tableEnv.fromChangelogStream(
    dataStream2,
    Schema.newBuilder().primaryKey("f0").build(),
    ChangelogMode.upsert()
)

// 注册表并执行聚合
tableEnv.createTemporaryView("InputTable2", table2)
tableEnv
    .executeSql("SELECT f0 AS name, SUM(f1) AS score FROM InputTable2 GROUP BY f0")
    .print()
}
```

Java 代码:

```
package com.xueai8.java.ch05;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.Schema;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.table.connector.ChangelogMode;
import org.apache.flink.types.Row;
import org.apache.flink.types.RowKind;

/**
 * Table API
 *
 * 下面的代码展示了如何在不同的场景中使用 fromChangelogStream。
 */
public class TableDemo06 {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        // 创建流表执行环境
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // === EXAMPLE 1 ===
        // 将此流解释为 retract 流
```

```

// 创建一个 changelog DataStream
DataStream<Row> dataStream1 = env.fromElements(
    Row.ofKind(RowKind.INSERT, "张三", 12),
    Row.ofKind(RowKind.INSERT, "李四", 5),
    Row.ofKind(RowKind.UPDATE_BEFORE, "张三", 12),
    Row.ofKind(RowKind.UPDATE_AFTER, "张三", 100)
);

// 将 DataStream 解释为一个 Table
Table table1 = tableEnv.fromChangelogStream(dataStream1);

// 注册该 table，并执行一个聚合
tableEnv.createTemporaryView("InputTable1", table1);
tableEnv
    .executeSql("SELECT f0 AS name, SUM(f1) AS score FROM InputTable1 GROUP BY f0")
    .print();

// === EXAMPLE 2 ===
// 将流解释为 upsert 流(不需要 UPDATE_BEFORE)
// 创建一个 changelog DataStream
DataStream<Row> dataStream2 = env.fromElements(
    Row.ofKind(RowKind.INSERT, "张三", 12),
    Row.ofKind(RowKind.INSERT, "李四", 5),
    Row.ofKind(RowKind.UPDATE_AFTER, "张三", 100)
);

// 将 DataStream 解释为一个 Table
Table table2 = tableEnv.fromChangelogStream(
    dataStream2,
    Schema.newBuilder().primaryKey("f0").build(),
    ChangelogMode.upsert()
);

// 注册表并执行聚合
tableEnv.createTemporaryView("InputTable2", table2);
tableEnv
    .executeSql("SELECT f0 AS name, SUM(f1) AS score FROM InputTable2 GROUP BY f0")
    .print();
}
}

```

执行以上代码，输出结果如下：

op	name	score
+I	李四	5
+I	张三	12
-D	张三	12
+I	张三	100

4 rows in set

```

+---+-----+-----+
| op |                name |      score |
+---+-----+-----+
| +I |                李四 |          5 |
| +I |                张三 |         12 |
| -D |                张三 |         12 |
| +I |                张三 |        100 |
+---+-----+-----+

```

4 rows in set

示例 1 中显示的默认 `ChangelogMode` 应该足以满足大多数用例，因为它接受所有类型的更改。

然而，示例 2 展示了如何通过使用 `upsert` 模式将更新消息的数量减少 50% 来限制传入更改的种类，从而提高效率。通过为 `toChangelogStream` 定义一个主键和 `upsert changelog` 模式，可以减少结果消息的数量。

下面的代码展示了如何在不同的场景中使用 `toChangelogStream`。

Scala 代码：

```

package com.xueai8.scala.ch05

import java.time.Instant

import org.apache.flink.api.scala._
import org.apache.flink.streaming.api.functions.ProcessFunction
import org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment}
import org.apache.flink.table.api._
import org.apache.flink.table.api.bridge.scala.StreamTableEnvironment
import org.apache.flink.types.Row
import org.apache.flink.util.Collector

/**
 * Table API
 *
 * 下面的代码展示了如何在不同的场景中使用 toChangelogStream。
 */
object TableDemo07 {

  def main(args: Array[String]) {
    // 设置流执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    // 创建流表执行环境
    val tableEnv = StreamTableEnvironment.create(env)

    // === EXAMPLE 1 ===
    // 以最简单和最通用的方式转换为 DataStream(no event-time)
    val simpleTable = tableEnv
      .fromValues(row("张三", 12), row("张三", 2), row("李四", 12))
      .as("name", "score")
      .groupBy($"name")
      .select($"name", $"score".sum())
  }
}

```



```

tableEnv
  .toChangelogStream(simpleTable)
  .executeAndCollect()
  .foreach(println)

// === EXAMPLE 2 ===
// 创建带有事件时间的表
tableEnv.executeSql(
  """
    CREATE TABLE GeneratedTable (
      name STRING,
      score INT,
      event_time TIMESTAMP_LTZ(3),
      WATERMARK FOR event_time AS event_time - INTERVAL '10' SECOND
    )
    WITH ('connector'='datagen')
  """)

val table = tableEnv.from("GeneratedTable")

// 以最简单和最通用的方式转换为 DataStream(with event-time)
val dataStream1: DataStream[Row] = tableEnv.toChangelogStream(table)

// 由于 event_time 是模式中的单个时间属性，默认情况下它被设置为流记录的时间戳；
// 然而，与此同时，它仍然是 Row 的一部分
dataStream1.process(new ProcessFunction[Row, Unit] {
  override def processElement(
    row: Row,
    ctx: ProcessFunction[Row, Unit]#Context,
    out: Collector[Unit]): Unit = {
    // prints: [name, score, event_time]
    println(row.getFieldNames(true))

    // timestamp exists twice
    assert(ctx.timestamp() == row.getFieldAs[Instant]("event_time").toEpochMilli)
  }
})
env.execute()

// === EXAMPLE 3 ===
// 转换为 DataStream，但将时间属性作为元数据列写出来，这意味着它不再是物理模式的一部分
val dataStream2: DataStream[Row] = tableEnv.toChangelogStream(
  table,
  Schema.newBuilder()
    .column("name", "STRING")
    .column("score", "INT")
    .columnByMetadata("rowtime", "TIMESTAMP_LTZ(3)")
    .build()

```

```

    )

    // 流记录的时间戳由元数据定义；它不是 Row 的一部分
    dataStream2.process(new ProcessFunction[Row, Unit] {
        override def processElement(
            row: Row,
            ctx: ProcessFunction[Row, Unit]#Context,
            out: Collector[Unit]): Unit = {

            // prints: [name, score]
            println(row.getFieldNames(true))

            // timestamp exists once
            println(ctx.timestamp())
        }
    })
    env.execute()
}

```

Java 代码:

```

package com.xueai8.java.ch05;

import org.apache.flink.streaming.api.datastream.DataStream;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.functions.ProcessFunction;
import org.apache.flink.table.api.DataTypes;
import org.apache.flink.table.api.Schema;
import org.apache.flink.table.api.Table;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;
import org.apache.flink.table.data.StringData;
import org.apache.flink.types.Row;
import org.apache.flink.util.Collector;

import java.time.Instant;

import static org.apache.flink.table.api.Expressions.*;

/**
 * Table API
 *
 * 下面的代码展示了如何在不同的场景中使用 toChangelogStream。
 */
public class TableDemo07 {

    public static void main(String[] args) throws Exception {
        // 设置流执行环境
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
        // 创建流表执行环境
        StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);

        // 创建带有事件时间的表
    }
}

```

```

tableEnv.executeSql(
    "CREATE TABLE GeneratedTable "
        + "("
        + "  name STRING,"
        + "  score INT,"
        + "  event_time TIMESTAMP_LTZ(3),"
        + "  WATERMARK FOR event_time AS event_time - INTERVAL '10' SECOND"
        + ")"
        + "WITH ('connector'='datagen')");

Table table = tableEnv.from("GeneratedTable");

// === EXAMPLE 1 ===
// 以最简单和最通用的方式转换为 DataStream(no event-time)
Table simpleTable = tableEnv
    .fromValues(row("Alice", 12), row("Alice", 2), row("Bob", 12))
    .as("name", "score")
    .groupBy($"name")
    .select($"name", $"score").sum());

tableEnv
    .toChangelogStream(simpleTable)
    .executeAndCollect()
    .forEachRemaining(System.out::println);

// === EXAMPLE 2 ===
// 以最简单和最通用的方式转换为 DataStream(with event-time)
DataStream<Row> dataStream1 = tableEnv.toChangelogStream(table);

// 由于 event_time 是模式中的单个时间属性，默认情况下它被设置为流记录的时间戳；
// 然而，与此同时，它仍然是 Row 的一部分
dataStream1.process(
    new ProcessFunction<Row, Void>() {
        @Override
        public void processElement(Row row, Context ctx, Collector<Void> out) {

            // prints: [name, score, event_time]
            System.out.println(row.getFieldNames(true));

            // timestamp exists twice
            assert ctx.timestamp() == row.<Instant>getFieldAs("event_time").toEpochMilli();
        }
    });
env.execute();

// === EXAMPLE 3 ===
// 转换为 DataStream，但将时间属性作为元数据列写出来，这意味着它不再是物理模式的一部分
DataStream<Row> dataStream2 = tableEnv.toChangelogStream(
    table,

```

```
Schema.newBuilder()
    .column("name", "STRING")
    .column("score", "INT")
    .columnByMetadata("rowtime", "TIMESTAMP_LTZ(3)")
    .build();

// 流记录的时间戳由元数据定义；它不是 Row 的一部分
dataStream2.process(
    new ProcessFunction<Row, Void>() {
        @Override
        public void processElement(Row row, Context ctx, Collector<Void> out) {

            // prints: [name, score]
            System.out.println(row.getFieldNames(true));

            // timestamp exists once
            System.out.println(ctx.timestamp());

        }
    });
env.execute();
```

// === EXAMPLE 4 ===

// 对于高级用户，也可以使用更多的内部数据结构来提高效率

```
DataStream<Row> dataStream3 = tableEnv.toChangelogStream(
    table,
    Schema.newBuilder()
        .column(
            "name",
            DataTypes.STRING().bridgedTo(StringData.class))
        .column(
            "score",
            DataTypes.INT())
        .column(
            "event_time",
            DataTypes.TIMESTAMP_LTZ(3).bridgedTo(Long.class))
        .build());
}
```

toChangelogStream(Table).executeandcollect() 的行为等于调用 Table.execute().collect()。但是，toChangelogStream(Table)可能对测试更有用，因为它允许访问 DataStream API 中后续 ProcessFunction 中生成的水印。

5.5 Table API 实时流处理案例

下面我们使用 Table API 重新实现以下几个案例。

5.5.1 传感器温度实时统计

处理 IOT 事件流

假设一台机器上安装了传感器，我们希望通过这些传感器收集数据，并计算每个传感器每 5 分钟的平均温度。

事件：

```
sensor_1,1629943899014,51.087254019871054
sensor_9,1629943899014,70.44743245583899
sensor_7,1629943899014,65.53215956486392
sensor_0,1629943899014,53.210570822216546
sensor_8,1629943899014,93.12876931817556
sensor_3,1629943899014,57.55153052162809
sensor_2,1629943899014,107.61249366604993
sensor_5,1629943899014,92.02083744773739
sensor_4,1629943899014,95.7688424087137
sensor_6,1629943899014,95.04398353316257
.....
```

首先，项目添加依赖：

然后，编写脚本，调用 Kafka 自带的生产者脚本，每秒 10 条的速度将数据发送给 Kafka。

Scala 代码：

Java 代码：

请按以下步骤执行。

- 1) 启动 zookeeper 服务和 kafka 服务。打开一个终端窗口，启动 ZooKeeper（不要关闭）
- 2) 打开另一个终端窗口，启动 Kafka 服务（不要关闭）
- 3) 打开第三个终端窗口，在 Kafka 中创建一个名为“temp”的主题(topic)
- 4) 查看已经创建的 Topic:
- 5) 运行我们上面编写的流执行程序（相当于 Kafka 的消费者程序）。
- 6) 在第三个终端窗口，执行传感器生成脚本：
- 7) 观察 5) 的控制台输出内容，类似下面这样：

op	id	win_end	avg_temp
+I	sensor_8	2021-08-26 10:11:47.000	91.17537874481508
+I	sensor_2	2021-08-26 10:11:47.000	111.99315044277473
+I	sensor_4	2021-08-26 10:11:47.000	100.14499019450446
+I	sensor_6	2021-08-26 10:11:47.000	93.87981149048566
+I	sensor_3	2021-08-26 10:11:47.000	61.75141384143203
+I	sensor_5	2021-08-26 10:11:47.000	89.54459167996558
+I	sensor_8	2021-08-26 10:11:48.000	90.86763958004099
+I	sensor_3	2021-08-26 10:11:48.000	61.79318995473199
+I	sensor_2	2021-08-26 10:11:48.000	111.22972954270946
+I	sensor_6	2021-08-26 10:11:48.000	94.09083634082995
+I	sensor_7	2021-08-26 10:11:48.000	67.15824053415402
+I	sensor_5	2021-08-26 10:11:48.000	90.44831606878837
+I	sensor_4	2021-08-26 10:11:48.000	100.97907052029765
+I	sensor_1	2021-08-26 10:11:48.000	60.257149639424455
+I	sensor_9	2021-08-26 10:11:48.000	72.35319925509846
+I	sensor_0	2021-08-26 10:11:48.000	49.46435741106236

Process finished with exit code 1

5.5.2 车辆超速实时监测

使用 Table API 和 SQL 实现物流公司运输车辆超速实时监测程序。

监听 Kafka cars 主题的车速信息（csv 格式），将超速信息（平均速度 > 100）发送到 Kafka fastcars 主题（json 格式）。

首先，在项目的 pom.xml 文件中添加如下依赖项：

接下来，编辑 Flink 流处理管道代码。

Scala 代码：

Java 代码：

执行模拟数据源程序

可以看到持续不断生成模拟车辆运行速度数据，如下所示：

```
Writing car6,115,38.44529,1631158426779
Writing car1,29,10.961765,1631158428433
Writing car9,98,9.312117,1631158429434
Writing car8,17,60.017864,1631158430435
Writing car8,18,87.79231,1631158431436
Writing car8,70,84.12387,1631158432437
Writing car4,114,46.86547,1631158433438
Writing car0,55,15.12608,1631158434439
```

```
Writing car8,10,10.488677,1631158435440
```

```
.....
```

回到消息者脚本执行窗口，查看超速数据。可以监测到类似如下这样的超速信息（json 格式）：

```
{"carId":"car6","window_start":"2021-09-09 11:33:42","window_end":"2021-09-09 11:33:47","avgSpeed":115.0}
{"carId":"car6","window_start":"2021-09-09 11:33:44","window_end":"2021-09-09 11:33:49","avgSpeed":115.0}
{"carId":"car6","window_start":"2021-09-09 11:33:46","window_end":"2021-09-09 11:33:51","avgSpeed":115.0}
{"carId":"car4","window_start":"2021-09-09 11:33:50","window_end":"2021-09-09 11:33:55","avgSpeed":114.0}
{"carId":"car4","window_start":"2021-09-09 11:33:52","window_end":"2021-09-09 11:33:57","avgSpeed":114.0}
```

```
.....
```

5.5.3 电商用户行为实时分析

电商平台中的用户行为频繁且较复杂，系统上线运行一段时间后，可以收集到大量的用户行为数据，进而利用大数据技术进行深度挖掘和分析，得到感兴趣的商业指标并增强对风险的控制。

电商用户行为数据多样，整体可以分为用户行为习惯数据和业务行为数据两大类。用户的行为习惯数据包括了用户的登录方式、上线的时间点及时长、点击和浏览页面、页面停留时间以及页面跳转等等，我们可以从中进行流量统计和热门商品的统计，也可以深度挖掘用户的特征；这些数据往往可以从 web 服务器日志中直接读取到。而业务行为数据就是用户在电商平台中针对每个业务（通常是某个具体商品）所作的操作，我们一般会在业务系统中相应的位置埋点，然后收集日志进行分析。业务行为数据又可以简单分为两类：一类是能够明显地表现出用户兴趣的行为，比如对商品的收藏、喜欢、评分和评价，我们可以从中对数据进行深度分析，得到用户画像，进而对用户给出个性化的推荐商品列表，这个过程往往会用到机器学习相关的算法；另一类则是常规的业务操作，但需要着重关注一些异常状况以做好风控，比如登录和订单支付。

基于对电商用户行为数据的基本分类，我们可以发现主要有以下三个分析方向：

- ❑ 热门统计。利用用户的点击浏览行为，进行流量统计、近期热门商品统计等。
- ❑ 偏好统计。利用用户的偏好行为，比如收藏、喜欢、评分等，进行用户画像分析，给出个性化的商品推荐列表。
- ❑ 风险控制。利用用户的常规业务行为，比如登录、下单、支付等，分析数据，对异常情况进行报警提示。

本项目限于数据，我们只实现热门统计和风险控制中的部分内容，将包括以下四大模块：实时热门商品统计、实时流量统计、恶意登录监控和订单支付失效监控。

由于对实时性要求较高，我们会用 flink 作为数据处理的框架。在项目中，我们将综合运用 flink 的各种 API，基于 EventTime 去处理基本的业务需求，并且灵活地使用底层的 processFunction，基于状态编程和 CEP 去处理更加复杂的情形。

数据源说明

我们准备了一份淘宝用户行为数据集，保存为 csv 文件。本数据集包含了 2017 年 11 月 25 日至 2017 年 12 月 3 日之间，约一百万随机用户的所有行为（行为包括点击、购买、加购、喜欢）。数据集的每一行表示一条用户行为，由用户 ID、商品 ID、商品类目 ID、行为类型和时间戳组成，并以逗号分隔。

注：从天池上下载的数据集 UserBehavior.csv 解压缩后为 3.41G，这里我们截取其中一部分（485730 行）用于开发测试，命名为 UserBehavior_part.csv。

关于数据集中每一列的详细描述如下：

列名称	含义	说明
User ID	用户ID	整数类型，序列化后的用户ID
Item ID	商品ID	整数类型，序列化后的商品ID
Category ID	商品类目ID	整数类型，序列化后的商品所属类目ID
Behavior type	行为类型	字符串，枚举类型，包括('pv', 'buy', 'cart', 'fav')
Timestamp	时间戳	长整数，行为发生的时间戳

注意到，用户行为类型共有四种，它们分别是：

- ❑ pv：商品详情页 pv，等价于点击。
- ❑ buy：商品购买。
- ❑ cart：将商品加入购物车。
- ❑ fav：收藏商品。

部分用户行为数据示例如下：

```
543462,1715,1464116,pv,1511658000
662867,2244074,1575622,pv,1511658000
561558,3611281,965809,pv,1511658000
894923,3076029,1879194,pv,1511658000
834377,4541270,3738615,pv,1511658000
.....
```

预备知识

Flink SQL 提供了如下一些与日期处理有关的函数。

❑ TO_TIMESTAMP

将 BIGINT 类型的日期或者 VARCHAR 类型的日期转换成 TIMESTAMP 类型。

语法：

```
TIMESTAMP TO_TIMESTAMP(BIGINT time)           // time: 毫秒
TIMESTAMP TO_TIMESTAMP(VARCHAR date)           // date: yyyy-MM-dd HH:mm:ss
TIMESTAMP TO_TIMESTAMP(VARCHAR date, VARCHAR format)
```

❑ FROM_UNIXTIME

返回值为 VARCHAR 类型的日期值，默认日期格式：yyyy-MM-dd HH:mm:ss，若指定日期格式则按指定格式输出。如果任一输入参数是 NULL，返回 NULL。

语法：

```
VARCHAR FROM_UNIXTIME(BIGINT unixtime[, VARCHAR format])
```

说明：

- 参数 unixtime 为长整型，是以秒为单位的时间戳。
- 参数 format 可选，为日期格式，默认格式为 yyyy-MM-dd HH:mm:ss，表示返回 VARCHAR 类型的符合指定格式的日期，如果有参数为 null 或解析错误，则返回 null。

【示例】使用 Table API 和 SQL，实现如下描述的数据实时处理管道。

- 1) 将用户行为数据采集到 Kafka；
- 2) 使用 Table API 读取 Kafka 并写入 MySQL；
- 3) 然后用 Grafana 实时可视化显示。

首先，项目添加依赖。因为我们要读取的 Kafka 的用户行为事件是 CSV 格式的，所以在项目的 pom.xml 文件中添加如下的依赖：

创建 Kafka 源表的 SQL 语句如下：

如上我们按照数据的格式声明了 5 个字段，除此之外，我们还通过计算列语法和 PROCTIME() 内置函数声明了一个产生处理时间的虚拟列。我们还通过 WATERMARK 语法，在 ts 字段上声明了 watermark 策略（容忍 5 秒乱序），ts 字段因此也成了事件时间列。

接下来，编写流处理代码。

Scala 代码：

Java 代码：

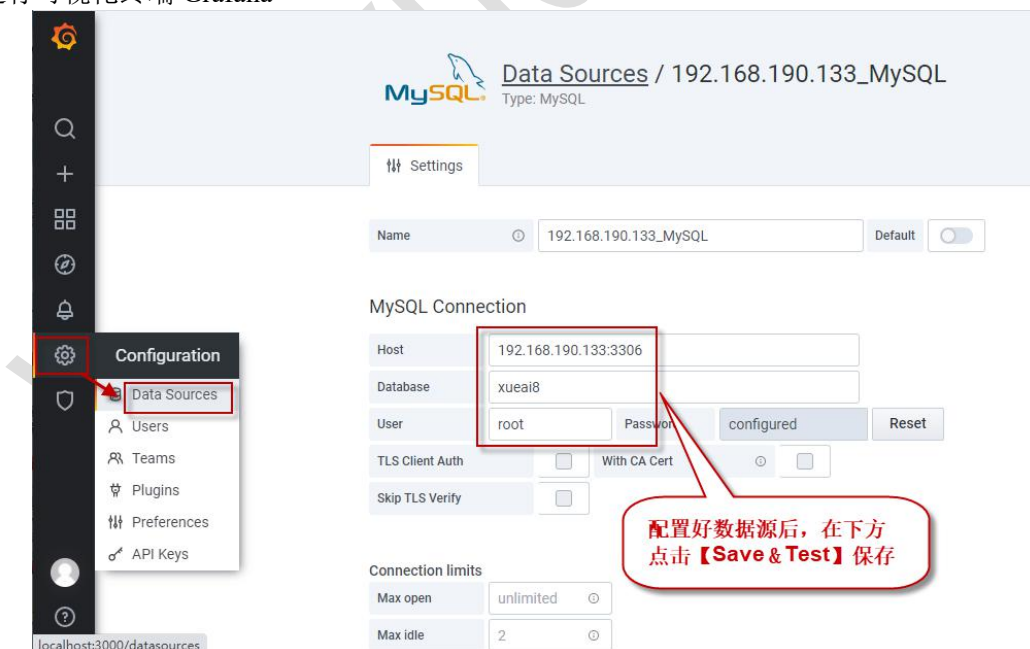
执行过程

1、启动 Kafka

2、在 MySQL 上创建接收表

```
CREATE TABLE buy_cnt_per_hour (  
    hour_of_day timestamp,  
    buy_cnt bigint  
);
```

3、运行可视化终端 Grafana

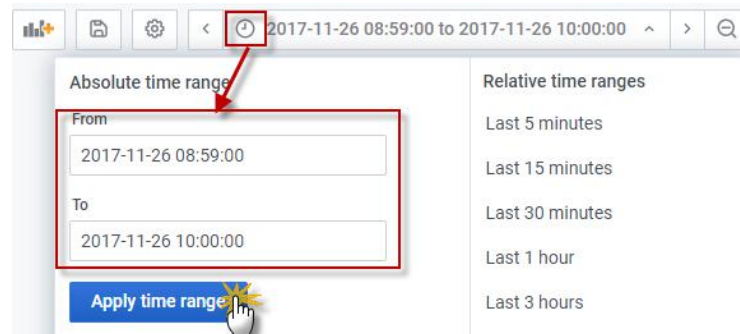


然后在 Grafana 中创建 dashboard，数据使用如下 SQL 语句查询获得：

```
SELECT  
    UNIX_TIMESTAMP(hour_of_day) as time_sec,  
    buy_cnt as value
```

FROM xueai8.buy_cnt_per_hour

为了更好地看到动态变化的效果，我们设置一个仪表板显示的时间范围。



4、运行 Flink 程序。

5、执行数据生产者脚本。

6、观察 Grafana 中数据实时呈现效果。



第 6 章 Flink on YARN

Flink 已内置支持可以在 YARN 上执行。任何使用 Flink API 构建的应用程序都可以在 YARN 上轻松执行。如果用户已经拥有 YARN 集群，则不需要设置或安装任何东西。在本章中，我们将看到如何利用现有的 Hadoop/YARN 集群并行执行 Flink 任务。

6.1 Flink on YARN session

从 Flink 1.8 开始，Hadoop 不再包含在 Flink 的安装包中，所以需要单独下载并拷贝到 Flink 的 lib 目录下。

6.1.1 下载 Flink 集成 Hadoop 依赖包

如果使用的是 Hadoop 2，则可从 Flink 官网下载 `flink-shaded-hadoop2-uber-2.7.5-1.10.0.jar`。如下图所示：

Additional Components

These are components that the Flink project develops which are not part of the main Flink release:

[Apache Flink-shaded 14.0 Source Release \(asc, sha512\)](#)

Pre-bundled Hadoop 2.4.1 (asc, sha1)
Pre-bundled Hadoop 2.6.5 (asc, sha1)
Pre-bundled Hadoop 2.7.5 (asc, sha1)
Pre-bundled Hadoop 2.8.3 (asc, sha1)

下载与自己的Hadoop版本相对应的Hadoop包

如果使用的是 Hadoop 3，则从 Maven 库下载。如下图所示：

Home » [org.apache.flink](#) » [flink-shaded-hadoop-3-uber](#)



Flink Shaded Hadoop 3 Uber

Flink Shaded Hadoop 3 Uber

License	Apache 2.0
Tags	hadoop apache
Used By	7 artifacts

选择最新的版本下载即可

Cloudera (4) Cloudera Libs (1)

Version	Repository	Usages	Date
3.1.1.7.2.9.0-173-9.0	Cloudera	4	Apr, 2021
3.1.1.7.2.8.0-224-9.0	Cloudera	4	Apr, 2021
3.1.1.7.2.1.0-327-9.0	Cloudera	7	Dec, 2020
3.1.1.7.0.3.0-79-7.0	Cloudera	6	Jan, 2020

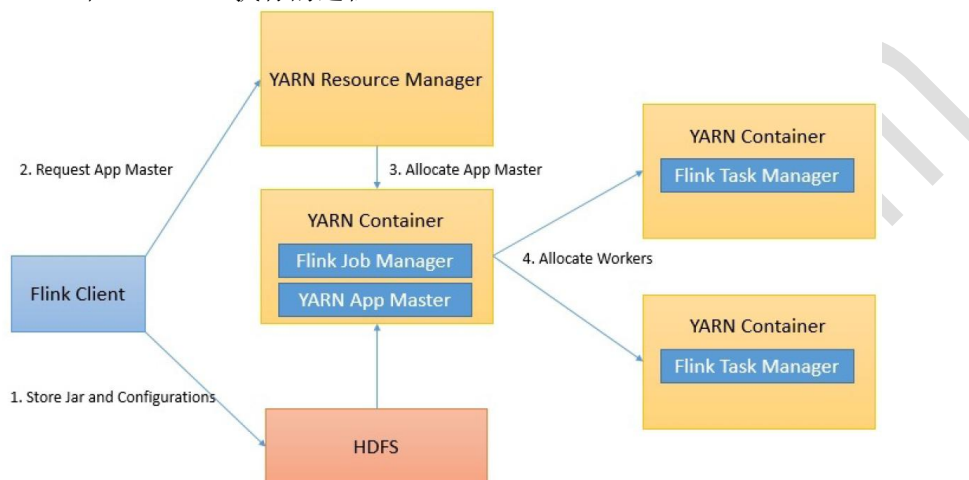
本教程中使用的是 Hadoop-3.2.2，所以下载 `flink-shaded-hadoop-3-uber-3.1.1.7.2.9.0-173-9.0.jar` 这个

包。将下载的 Hadoop 包拷贝到各个节点安装的 Flink 的 lib 目录下。

6.1.2 运行 Flink on YARN session

Apache Hadoop YARN 是一个资源提供者，在许多数据处理框架中很受欢迎。Flink 服务提交给 YARN 的 ResourceManager，后者在由 YARN NodeManager 管理的机器上生成 container 资源容器。Flink 将其 JobManager 和 TaskManager 实例部署到这些 container 资源容器中。

下图展示了 Flink 在 YARN 上执行的过程：



上图显示了 YARN 上的 Flink 的内部工作情况。它经过以下步骤：

- 1、检查是否设置了 Hadoop 和 YARN 配置目录；
- 2、如果是，则联系 HDFS 并将 JAR 和配置存储在 HDFS 上；
- 3、联系 node manager 分配 application master；
- 4、一旦分配了 application master 后，启动 Flink Job Manager；
- 5、稍后，根据给定的配置参数启动 Flink Task Manager。

Flink 可以根据运行在 JobManager 上的作业所需的处理槽的数量动态地分配和解除分配 TaskManager 资源。

注意：确保设置了 HADOOP_CLASSPATH 环境变量(可以通过运行 `echo $HADOOP_CLASSPATH` 来检查它)。如果没有，使用如下的命令来设置：

```
$ export HADOOP_CLASSPATH=`hadoop classpath`
```

一旦确定设置了 HADOOP_CLASSPATH 环境变量，我们就可以在 YARN 上启动一个 Flink 会话 (Flink on YARN session)，并提交 Flink 作业程序。

Flink on YARN session 是一个会话，它在各个节点上启动所有必需的 Flink 服务(Job Manager 和 Task Managers)，以便我们可以开始执行 Flink 作业。

要启动 Flink on YARN session，执行以下命令（确保已经启动了 Hadoop 集群）：

```
# (0) export HADOOP_CLASSPATH
$ export HADOOP_CLASSPATH=`hadoop classpath`

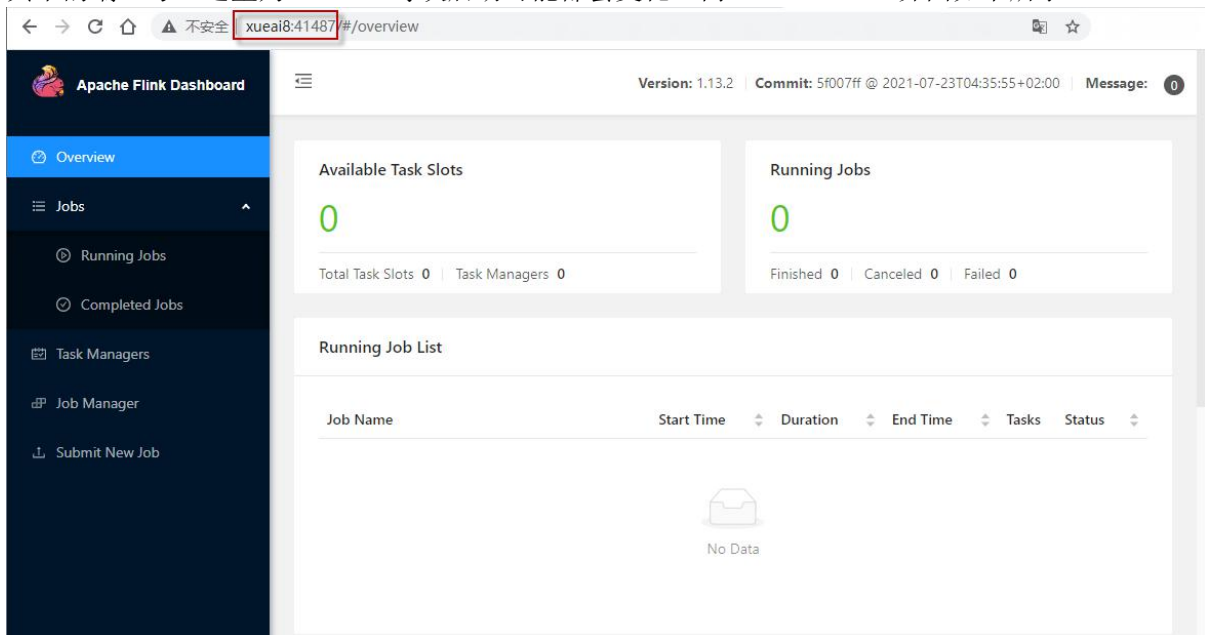
# (1) 启动 YARN session
$ cd ~/bigdata/flink-1.13.2
```

```
$ ./bin/yarn-session.sh --detached # --detached 参数，客户端将在提交被接受后停止
```

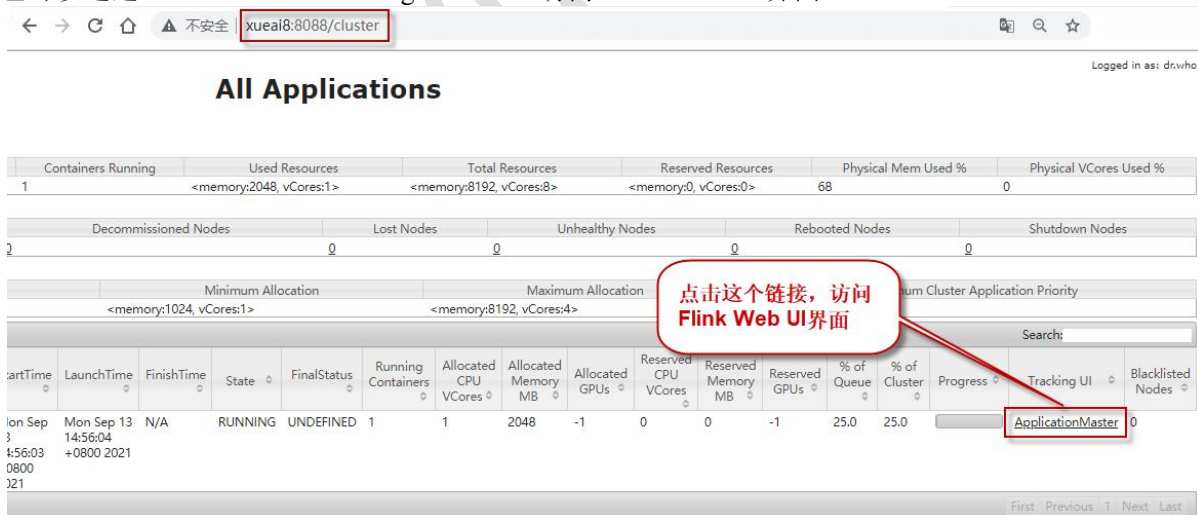
现在，可以通过命令输出最后一行打印的 URL 或通过 YARN ResourceManager Web UI 访问 Flink Web 界面。例如，在命令输出最后一行打印的 URL 信息如下所示：

```
2021-09-13 14:56:30,618 INFO org.apache.flink.yarn.YarnClusterDescriptor
plication 1631516048010 0002'.
JobManager Web Interface: http://xueai8:41487
2021-09-13 14:56:31,044 INFO org.apache.flink.yarn.cli.FlinkYarnSessionCli
```

其中的端口号（这里为 41487）每次启动可能都会变化。访问 Flink Web UI 界面如下所示：



也可以通过 YARN ResourceManager Web UI 访问 Flink Web UI 界面：



这种模式下会启动 YARN session，并且会启动 Flink 的两个必要服务：JobManager 和 TaskManagers，然后就可以向集群提交作业。同一个 session 中可以提交多个 Flink 作业。

注：如果遇到以下启动错误：

```
2021-06-04 18:18:12,867 ERROR org.apache.flink.yarn.cli.FlinkYarnSessionCli
[] - Error while
running the Flink session.
```

```
java.lang.NoSuchMethodError:
```

```
org.apache.commons.cli.Option.builder(Ljava/lang/String;)Lorg/apache/commons/cli/Option$Builder;
```

```
at org.apache.flink.yarn.cli.FlinkYarnSessionCli.<init>(FlinkYarnSessionCli.java:230)
~[flink-dist_2.12-1.13.1.jar:1.13.1]
at org.apache.flink.yarn.cli.FlinkYarnSessionCli.<init>(FlinkYarnSessionCli.java:156)
~[flink-dist_2.12-1.13.1.jar:1.13.1]
at org.apache.flink.yarn.cli.FlinkYarnSessionCli.main(FlinkYarnSessionCli.java:851)
[flink-dist_2.12-1.13.1.jar:1.13.1]
```

The program finished with the following exception:

```
java.lang.NoSuchMethodError:
org.apache.commons.cli.Option.builder(Ljava/lang/String;)Lorg/apache/commons/cli/Option$Builder;
at org.apache.flink.yarn.cli.FlinkYarnSessionCli.<init>(FlinkYarnSessionCli.java:230)
at org.apache.flink.yarn.cli.FlinkYarnSessionCli.<init>(FlinkYarnSessionCli.java:156)
at org.apache.flink.yarn.cli.FlinkYarnSessionCli.main(FlinkYarnSessionCli.java:851)
```

原因是，缺少 commons-cli-1.4.jar 包。从网上下载，拷贝到 flink 的 lib 目录下，重新启动即可。

必须确保设置了 YARN_CONF_DIR 和 HADOOP_CONF_DIR 环境变量，以便 Flink 能够找到所需的配置。也可以在启动 Flink session 时，详细说明任务管理器的数量、每个任务管理器的内存和要使用的插槽：

```
$ ./bin/yarn-session.sh -n 2 -tm 1024 -s 4
```

以上参数说明：

- ☐ -n,--container <arg>: 要分配的 YARN container 数量(=Task Managers 数量)
- ☐ -tm,--taskManagerMemory <arg>: 每个 TaskManager Container 的内存 [in MB]
- ☐ -s,--slots <arg>: 每个 TaskManager 的 slot 的数量

所以上面的命令启动了 2 个 TaskManager，每个 TaskManager 内存为 1G 且占用 4 个资源槽。在启动 YARN session 的时候会加载 conf/flink-config.yaml 配置文件，我们可以根据自己的需求去修改里面的相关参数。

6.1.3 提交 Flink 作业

现在我们有了一个与 YARN 连接的 Flink session，我们准备向其提交一个 Flink 作业。

提交批处理作业

可以使用如下的命令来提交 Flink 批处理示例程序：

```
$ ./bin/flink run ./examples/batch/WordCount.jar --input hdfs://xueai8:8020/data/flink/wc.txt
```

这将调用将在 YARN 集群上执行的 Flink 作业

下面是来自 Flink 应用程序主 UI 的作业执行的屏幕截图。

Apache Flink Dashboard

Version: 1.13.2 | Commit: 5f007ff @ 2021-07-23T04:35:55+02:00 | Message: 0

Overview

Jobs

Running Jobs

Completed Jobs

Task Managers

Job Manager

Submit New Job

Available Task Slots

0

Total Task Slots 0 | Task Managers 0

Running Jobs

0

Finished 1 | Canceled 0 | Failed 0

Running Job List

Job Name	Start Time	Duration	End Time	Tasks	Status
No Data					

Completed Job List

Job Name	Start Time	Duration	End Time	Tasks	Status
Flink Java Job at Mon Sep 13 15:59:35 CST 2021	2021-09-13 15:59:38	13s	2021-09-13 15:59:51	6 / 6	FINISHED

以下是 Flink 执行计划的截图：

Apache Flink Dashboard

Version: 1.13.2 | Commit: 5f007ff @ 2021-07-23T04:35:55+02:00 | Message: 0

Overview

Jobs

Running Jobs

Completed Jobs

Task Managers

Job Manager

Submit New Job

Flink Java Job at Mon Sep 13 15:59:35 CST 2021

ID: 52eba479d02d505175f945f2e005689c | Start Time: 2021-09-13 15:59:38 | End Time: 2021-09-13 15:59:51 | Duration: 13s

Overview | Exceptions | TimeLine | Checkpoints | Configuration

Overview

Execution Plan

Parallelism: 2

Backpressured (max): N/A
Bury (max): N/A
Operation: (none) -> FlatMap -> Sorted Combine

Group/Reduce Reduce (SUM(1), at main/WordCount.java:87)
Parallelism: 2
Backpressured (max): N/A
Bury (max): N/A
Operation: Sorted Group Reduce

Data Sink (collect())
Parallelism: 2
Backpressured (max): N/A
Bury (max): N/A
Operation: (none)

Name	Status	Bytes Received	Records Received	Bytes Sent	Records Sent	Tasks
CHAIN DataSource (at main/WordCount.java:69) [org.apache.flink.ap...	FINISHED	0 B	0	50 B	4	2
Reduce (SUM(1), at main/WordCount.java:87)	FINISHED	66 B	4	50 B	4	2
DataSink (collect())	FINISHED	58 B	4	0 B	0	2

执行结果如下所示：


```
(up,1)
(day,3)
(good,2)
(study,1)
[hduser@xueai8 flink-1.13.2]$
```

如果我们不想将结果输出在终端，而是保存在文件中，可以使用--output 参数指定保存结果的地方：

```
$ ./bin/flink run ./examples/batch/WordCount.jar --input hdfs://xueai8:8020/data/flink/wc.txt
--output hdfs://xueai8:8020/data/flink/out
```

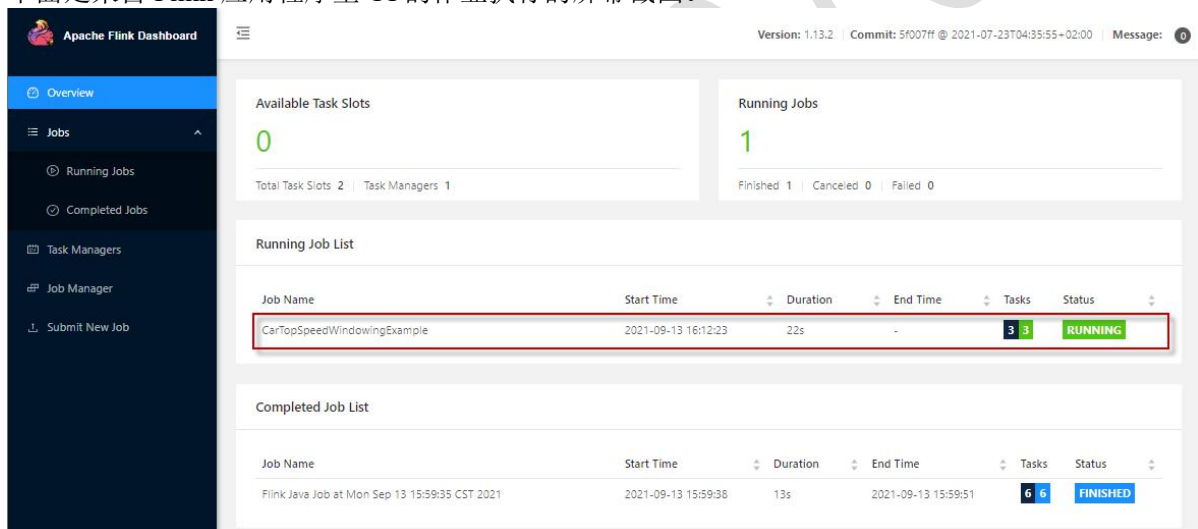
需要注意的是：上面的--input 和--output 参数并不是 Flink 内部的参数，而是 WordCount 程序中定义的。另外在指定路径的时候一定记得需要加上模式，比如上面的 hdfs://，否则程序会在本地寻找文件。

提交流处理作业

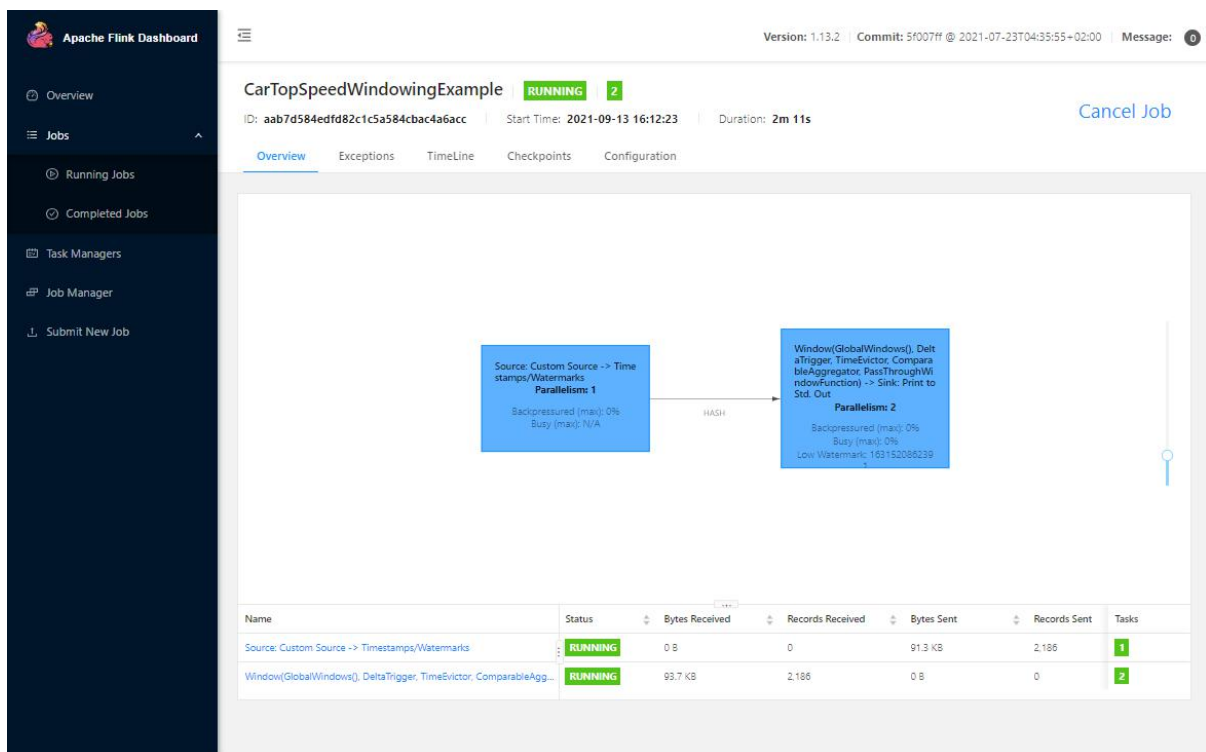
可以使用如下命令来提交 Flink 流处理示例程序：

```
$ ./bin/flink run ./examples/streaming/TopSpeedWindowing.jar
```

下面是来自 Flink 应用程序主 UI 的作业执行的屏幕截图。



以下是 Flink 执行计划的截图：



注：执行过程中容易出现的异常问题及解决方法

异常信息：

Container [pid=28920,containerID=container_1389136889967_0001_01_000121] is running beyond virtual memory limits. Current usage: 1.2 GB of 1 GB physical memory used; 2.2 GB of 2.1 GB virtual memory used. Killing container.

原因：

在 mapred-site.xml（或者 mapred-default.xml）中，以下两个参数配置了 Map 和 Reduce 任务将使用的物理 RAM 的上限：

```
mapreduce.map.java.opts: -Xmx3072m
mapreduce.reduce.java.opts: -Xmx6144m
```

在 YARN 水平上有一个虚拟内存和物理内存使用率的检查。问题不仅仅是 VM 没有足够的内存。而且还因为虚拟内存的使用超出了给定物理内存的预期。注意，这通常是发生在 Centos/RHEL 6 上的，因为它侵略性地分配虚拟内存。

其中 yarn.nodemanager.vmem-pmem-ratio 是 虚拟内存和物理内存比例，在 yarn-site.xml 中设置，默认是 2.1。很明显，异常信息中 container 占用了 2.2G 的虚拟内存，但是分配给 container 的却只有 2.1GB。所以 kill 掉了这个 container。

解决办法：

通过设置 yarn.nodemanager.vmem-check-enabled 的值为 false，禁用虚拟内存使用检查。

通过设置 yarn.nodemanager.vmem-pmem-ratio 为更高的值来增加 VM:PM 比。

在 \$HADOOP_HOME/etc/hadoop/yarn-site.xml 中增加这两个参数的设置，如下所示：

```
<property>
  <name>yarn.nodemanager.vmem-check-enabled</name>
  <value>>false</value>
</property>
```

```
<property>
  <name>yarn.nodemanager.vmem-pmem-ratio</name>
  <value>4</value>
</property>
```

6.1.4 停止 Flink on YARN session

一旦处理完成，我们可以通过两种方式停止 Flink YARN session。

第一种方法是在开始该 YARN 会话的控制台上简单地执行 Ctrl+C，这将发送终止信号并停止 YARN session。

第二种方法是执行以下命令来停止会话（请根据 `yarn-session.sh` 命令的输出信息替换应用程序 id）：

```
$ echo "stop" | ./bin/yarn-session.sh -id application_xxxxx_xxx stop
```

我们可以立即看到 Flink YARN 应用程序被杀死。

6.2 Flink on YARN 支持的部署模式

对于生产使用，建议以 Per-job 或 Application 模式部署 Flink 应用程序，因为这些模式为应用程序提供了更好的隔离。

6.2.1 Application 模式

Application Mode 将在 YARN 上启动一个 Flink 集群，其中应用程序 jar 的 `main()` 方法将在 YARN 中的 JobManager 上执行。应用程序一完成，该集群就会关闭。可以通过 `yarn application -kill <ApplicationId>` 或取消 Flink 作业手动停止集群。

```
$ ./bin/flink run-application -t yarn-application ./examples/streaming/TopSpeedWindowing.jar
```

一旦部署了 Application Mode 集群，就可以与它进行交互，以执行取消或获取保存点等操作。

列出集群上正在运行的作业

```
$ ./bin/flink list -t yarn-application -Dyarn.application.id=application_XXXX_YY
```

取消正在运行的作业

```
$ ./bin/flink cancel -t yarn-application -Dyarn.application.id=application_XXXX_YY <jobId>
```

请注意，取消 Application 集群上的作业将停止该集群。

要释放 Application 模式的全部潜力，可以考虑与 `yarn.provided.lib.dirs` 配置选项一起使用它，并将应用程序 jar 包预先上传到集群中所有节点都可以访问的位置。在本例中，命令看起来像这样：

```
$ ./bin/flink run-application -t yarn-application \
  -Dyarn.provided.lib.dirs="hdfs://myhdfs/my-remote-flink-dist-dir" \
  hdfs://myhdfs/jars/my-application.jar
```

上述操作将使作业提交变得更加轻量级，因为所需的 Flink jar 和应用程序 jar 将由指定的远程位置提取，而不是由客户端发送到集群。

6.2.2 Per-Job 集群模式

Per-Job 集群模式将在 YARN 上启动一个 Flink 集群，然后在本地运行提供的应用程序 jar，最后将

JobGraph 提交给 YARN 上的 JobManager。如果传递--detached 参数，客户端将在提交被接受后停止。一旦作业停止，YARN 集群就会停止。

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

一旦部署了 Per-Job 集群，就可以与它进行交互，以执行取消或获取保存点等操作。

```
# 列出集群上正在运行的作业
```

```
$ ./bin/flink list -t yarn-per-job -Dyarn.application.id=application_XXXX_YY
```

```
# 取消正在运行的作业
```

```
$ ./bin/flink cancel -t yarn-per-job -Dyarn.application.id=application_XXXX_YY <jobId>
```

请注意，取消 Per-Job 集群上的作业将停止集群。

6.2.3 Session 模式

上一节我们已经描述了 session 模式。Session 模式有两种操作模式：

- ❑ attached mode（默认）：yarn-session.sh 客户端将 Flink 集群提交给 YARN，但是客户端继续运行，跟踪集群的状态。如果集群失败，客户端将显示错误。如果客户端被终止，它也会发出关闭集群的信号。
- ❑ detached mode（-d 或--detached）：yarn-session.sh 客户端将 Flink 集群提交给 YARN，然后客户端返回。需要调用另一个客户端或 YARN 工具来停止 Flink 集群。

Session 模式将在/tmp/.yarn-properties-<username>目录下创建一个隐藏的 YARN 属性文件，它将在提交作业时由命令行接口提取用于集群发现。

也可以在提交 Flink 作业时，在命令行接口中手动指定目标 YARN 集群。示例如下：

```
$ ./bin/flink run -t yarn-session \
-Dyarn.application.id=application_XXXX_YY \
./examples/streaming/TopSpeedWindowing.jar
```

可以使用以下命令重新附加到 YARN session：

```
$ ./bin/yarn-session.sh -id application_XXXX_YY
```

除了通过 conf/flink-conf.yaml 文件传递配置外，也可以使用-Dkey=value 参数在提交时将任何配置传递给./bin/yarn-session.sh 客户端。

YARN session 客户端也有一些常用设置的“快捷参数”。它们可以通过./bin/yarn-session.sh -h 列出，如下图所示。

```
Usage:
Optional
-at,--applicationType <arg>    Set a custom application type for the application on YARN
-D <property=value>            use value for given property
-d,--detached                  If present, runs the job in detached mode
-h,--help                      Help for the Yarn session CLI.
-id,--applicationId <arg>      Attach to running YARN session
-j,--jar <arg>                 Path to Flink jar file
-jm,--jobManagerMemory <arg>   Memory for JobManager Container with optional unit (default: MB)
-m,--jobmanager <arg>         Set to yarn-cluster to use YARN execution mode.
-nl,--nodeLabel <arg>          Specify YARN node label for the YARN application
-nm,--name <arg>               Set a custom name for the application on YARN
-q,--query                     Display available YARN resources (memory, cores)
-qu,--queue <arg>              Specify YARN queue.
-s,--slots <arg>               Number of slots per TaskManager
-t,--ship <arg>                Ship files in the specified directory (t for transfer)
-tm,--taskManagerMemory <arg>  Memory per TaskManager Container with optional unit (default: MB)
-yd,--yarnDetached              If present, runs the job in detached mode (deprecated; use non-YARN specific option instead)
-z,--zookeeperNamespace <arg> Namespace to create the Zookeeper sub-paths for high availability mode
[hduser@xueai8 flink-1.13.2]$
```

6.3 小结

在本章中，我们讨论了如何使用现有的 YARN 集群在分布式模式下执行 Flink 作业。我们一步步地查看了执行的细节，并理解了一些相同的实际示例。

www.xueai8.com

第 7 章 基于 Flink 构建流批一体数仓

基于 Flink 构建流批一体的实时数仓是目前数据仓库领域比较火的实践方案。随着 Flink 的不断迭代，其提供的一系列技术特性使得用户构建流批一体的应用变得越来越方便。本文将以 Flink1.13 为例，介绍这些特性的基本使用方式，主要包括以下内容：

- ❑ Flink 集成 Hive
- ❑ Hive Catalog 与 Hive Dialect
- ❑ Flink 读写 Hive
- ❑ Flink upsert-kafka 连接器
- ❑ Flink CDC 的 connector

7.1 Flink 集成 Hive

使用 Hive 构建数据仓库已经成为了比较普遍的一种解决方案。目前，一些比较常见的大数据处理引擎，都无一例外兼容 Hive。Flink 从 1.9 开始支持集成 Hive，在 Flink1.10 版本对 Hive 的集成也达到了生产级别的要求。值得注意的是，不同版本的 Flink 对于 Hive 的集成有所差异，本文将以最新的 Flink1.13 版本为例，阐述 Flink 集成 Hive 的过程和步骤。

7.1.1 Flink 集成 Hive 的方式

Flink 提供了与 Hive 的双重集成。

- ❑ 持久化元数据

第一重集成是 Flink 利用 Hive 的 MetaStore 作为持久化的 Catalog 目录，可通过 HiveCatalog 将不同会话中的 Flink 元数据存储到 Hive Metastore 中。例如，我们可以使用 HiveCatalog 将 Kafka 源表或 ElasticSearch 源表存储在 Hive Metastore 中，这样该表的元数据信息会被持久化到 Hive 的 MetaStore 对应的元数据库中，在后续的 SQL 查询中，就可以重复使用它们。

- ❑ 利用 Flink 来读写 Hive 的表。

第二重集成是提供 Flink 作为读取和写入 Hive 表的替代引擎。

HiveCatalog 的设计提供了与 Hive 良好的兼容性，用户可以“开箱即用”的访问其已有的 Hive 表。不需要修改现有的 Hive Metastore，也不需要更改表的数据位置或分区。

7.1.2 Flink 集成 Hive 的步骤

首先，需要了解 Flink 支持的 Hive 版本信息。Flink 支持如下的 Hive 版本：

大版本	V1	V2	V3	V4	V5	V6	V7
1.0	1.0.0	1.0.1					
1.1	1.1.0	1.1.1					
1.2	1.2.0	1.2.1	1.2.2				
2.0	2.0.0	2.0.1					
2.1	2.1.0	2.1.1					
2.2	2.2.0						
2.3	2.3.0	2.3.1	2.3.2	2.3.3	2.3.4	2.3.5	2.3.6
3.1	3.1.0	3.1.1	3.1.2				

值得注意的是，对于不同的 Hive 版本，可能在功能方面有所差异，这些差异取决于所使用的 Hive 版本，而不取决于 Flink。一些版本的功能差异如下：

- ❑ Hive 内置函数在使用 Hive-1.2.0 及更高版本时支持。
- ❑ 列约束，也就是 PRIMARY KEY 和 NOT NULL，在使用 Hive-3.1.0 及更高版本时支持。
- ❑ 更改表的统计信息，在使用 Hive-1.2.0 及更高版本时支持。
- ❑ DATE 列统计信息，在使用 Hive-1.2.0 及更高版本时支持。
- ❑ 使用 Hive-2.0.x 版本时不支持写入 ORC 表。

依赖

Apache Hive 是基于 Hadoop 之上构建的，所以还需要 Hadoop 的依赖，这需要先配置好 HADOOP_CLASSPATH。配置 HADOOP_CLASSPATH，需要在/etc/profile 文件中配置如下的环境变量：

```
export HADOOP_CLASSPATH=$HADOOP_HOME/bin/hadoop classpath`
```

这一点非常重要，否则在使用 FlinkSQL Cli 查询 Hive 中的表时，会报如下错误：

```
java.lang.ClassNotFoundException: org.apache.hadoop.mapred.JobConf
```

另外，集成 Hive 需要额外添加一些依赖 jar 包，并将其放置在 Flink 安装目录下的 lib 文件夹下，这样我们才能通过 Table API 或 SQL Client 与 Hive 进行交互。

Flink 官网提供了两种方式添加 Hive 的依赖项。

- ❑ 第一种是使用 Flink 提供的 Hive Jar 包（根据使用的 Metastore 的版本来选择对应的 Hive jar）。
- ❑ 如果使用的 Hive 版本与 Flink 提供的 Hive jar 包兼容的版本不一致，可以选择第二种方式，即分别添加每个所需的 jar 包。

建议优先使用 Flink 提供的 Hive jar 包，这种方式比较简单方便。下面列举了可用的 jar 包及其适用的 Hive 版本，我们可以根据使用的 Hive 版本，下载对应的 jar 包即可。比如本书使用的 Hive 版本为 Hive 3.1.2，所以只需要下载 flink-sql-connector-hive-3.1.2 即可，并将其放置在 Flink 安装目录的 lib 文件夹下。Flink 的 Hive 连接器包含了 flink-hadoop 兼容和 flink-orc jar 包。

Metastore version	Maven dependency	SQL Client JAR
1.0.0 - 1.2.2	flink-sql-connector-hive-1.2.2	Download
2.0.0 - 2.2.0	flink-sql-connector-hive-2.2.0	Download
2.3.0 - 2.3.6	flink-sql-connector-hive-2.3.6	Download
3.0.0 - 3.1.2	flink-sql-connector-hive-3.1.2	Download

下载地址: <https://repo.maven.apache.org/maven2/org/apache/flink/>。

上面列举的 jar 包, 是我们在使用 Flink SQL Cli 所需要的 jar 包, 除此之外, 根据不同的 Hive 版本, 还需要添加如下三个 jar 包 (以 Hive 3.1.2 为例):

- ❑ flink-connector-hive_2.12-1.13.2.jar
- ❑ hive-exec-3.1.2.jar

其中, hive-exec-3.1.2.jar 包存在于 Hive 安装路径下的 lib 文件夹。flink-connector-hive_2.11-1.12.2.jar 的下载地址为: https://mvnrepository.com/artifact/org.apache.flink/flink-connector-hive_2.12/1.13.2。

最终, flink 的 lib 下的 jar 包如下图所示:

Name	Size (KB)	Last modified	Owner	Group	Access
..					
antlr-runtime-3.5.2.jar	163	2021-09-14...	hduser	hduser	-rw-rw-r--
commons-cli-1.4.jar	52	2021-08-09...	hduser	hduser	-rw-rw-r--
flink-connector-hive_2.12-1.13.2.jar	7 604	2021-09-13...	hduser	hduser	-rw-rw-r--
flink-csv-1.13.2.jar	90	2021-07-23...	hduser	hduser	-rw-r--r--
flink-dist_2.12-1.13.2.jar	103 645	2021-07-23...	hduser	hduser	-rw-r--r--
flink-json-1.13.2.jar	144	2021-07-23...	hduser	hduser	-rw-r--r--
flink-shaded-hadoop-3-uber-3.1.1.7.2.9.0-173-9.0.jar	58 207	2021-08-09...	hduser	hduser	-rw-rw-r--
flink-shaded-zookeeper-3.4.14.jar	7 529	2021-04-10...	hduser	hduser	-rw-r--r--
flink-sql-connector-hive-3.1.2_2.12-1.13.2.jar	47 700	2021-09-14...	hduser	hduser	-rw-rw-r--
flink-table-blink_2.12-1.13.2.jar	37 644	2021-07-23...	hduser	hduser	-rw-r--r--
flink-table_2.12-1.13.2.jar	34 198	2021-07-23...	hduser	hduser	-rw-r--r--
hive-exec-3.1.2.jar	39 671	2019-08-23...	hduser	hduser	-rw-r--r--
log4j-1.2-api-2.12.1.jar	65	2020-03-25...	hduser	hduser	-rw-r--r--
log4j-api-2.12.1.jar	270	2020-03-25...	hduser	hduser	-rw-r--r--
log4j-core-2.12.1.jar	1 635	2020-03-25...	hduser	hduser	-rw-r--r--
log4j-slf4j-impl-2.12.1.jar	22	2020-03-25...	hduser	hduser	-rw-r--r--

Flink Maven 项目

如果使用 Maven 构建自己的程序, 则需要在 pom.xml 文件中包含以下依赖项。

```

<!-- Flink Dependency -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-hive_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-java-bridge_2.12</artifactId>

```

```
<version>1.13.2</version>
<scope>provided</scope>
</dependency>

<!-- Hive Dependency -->
<dependency>
  <groupId>org.apache.hive</groupId>
  <artifactId>hive-exec</artifactId>
  <version>3.1.2</version>
  <scope>provided</scope>
</dependency>
```

将上面的三个 jar 包添加至 Flink 的 lib 目录下之后, 就可以使用 Flink 操作 Hive 的数据表了。

7.1.3 Flink 连接 Hive 模板代码

Flink 使用 HiveCatalog 可以通过批或者流的方式来处理 Hive 中的表。这就意味着 Flink 既可以作为 Hive 的一个批处理引擎, 也可以通过流处理的方式来读写 Hive 中的表, 从而支持实时数仓的应用和流批一体的落地。

通过表环境或 YAML 配置使用 catalog 接口和 HiveCatalog 连接到现有的 Hive 安装。连接到 Hive 的模板代码如下。

Scala 代码:

```
val settings = EnvironmentSettings.newInstance()
  .useBlinkPlanner()
  .inStreamingMode()
  .build()
val tableEnv = TableEnvironment.create(settings)

val name          = "myhive"
val defaultDatabase = "default"
val hiveConfDir    = "/home/hduser/bigdata/hive-3.1.2/conf"

val hive = new HiveCatalog(name, defaultDatabase, hiveConfDir)
tableEnv.registerCatalog("myhive", hive)

// 将 HiveCatalog 设置为会话的当前目录
tableEnv.useCatalog("myhive")
```

Java 代码:

```
EnvironmentSettings settings = EnvironmentSettings.newInstance()
  .useBlinkPlanner()
  .inStreamingMode()
  .build();
TableEnvironment tableEnv = TableEnvironment.create(settings);

String name          = "myhive";
String defaultDatabase = "default";
String hiveConfDir    = "/home/hduser/bigdata/hive-3.1.2/conf";

HiveCatalog hive = new HiveCatalog(name, defaultDatabase, hiveConfDir);
```



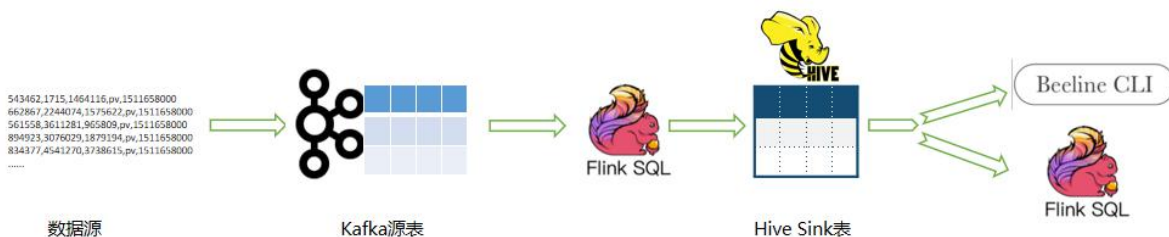
```
tableEnv.registerCatalog("myhive", hive);

// 将 HiveCatalog 设置为会话的当前目录
tableEnv.useCatalog("myhive");
```

7.2 批流一体数仓构建实例

Flink 1.13.2 的 Hive Streaming 功能极大地提高了 Hive 的实时性能，对于 ETL 操作非常有用，也可以以一定的灵活性满足流连续查询的需求。

基于上一节的内容基础，本小节我们以电商用户行为实时分析为例，构建一个批流一体数据仓库案例。实现架构如下图所示：



7.2.1 数据集说明

我们准备了一份淘宝用户行为数据集，保存为 csv 文件。本数据集包含了 2017 年 11 月 25 日至 2017 年 12 月 3 日之间，约一百万随机用户的所有行为（行为包括点击、购买、加购、喜欢）。数据集的每一行表示一条用户行为，由用户 ID、商品 ID、商品类目 ID、行为类型和时间戳组成，并以逗号分隔。

注：从天池上下载的数据集 UserBehavior.csv 解压缩后为 3.41G，这里我们截取其中一部分（485730 行）用于开发测试，命名为 UserBehavior_part.csv。

关于数据集中每一列的详细描述如下：

列名称	含义	说明
User ID	用户ID	整数类型，序列化后的用户ID
Item ID	商品ID	整数类型，序列化后的商品ID
Category ID	商品类目ID	整数类型，序列化后的商品所属类目ID
Behavior type	行为类型	字符串，枚举类型，包括('pv', 'buy', 'cart', 'fav')
Timestamp	时间戳	长整数，行为发生的时间戳

注意到，用户行为类型共有四种，它们分别是：

- ☐ pv：商品详情页 pv，等价于点击。
- ☐ buy：商品购买。
- ☐ cart：将商品加入购物车。
- ☐ fav：收藏商品。

部分用户行为数据示例如下：

```
543462,1715,1464116,pv,1511658000
662867,2244074,1575622,pv,1511658000
561558,3611281,965809,pv,1511658000
894923,3076029,1879194,pv,1511658000
834377,4541270,3738615,pv,1511658000
```

.....

为了模拟源数据流，我们先编写好生产数据的脚本 `streamuserbehavior.sh`，它调用 Kafka 自带的生产者脚本，以每秒 10 条的速度将数据发送给 Kafka 的 `user_behavior` 主题。

`streamuserbehavior.sh`:

```
#!/bin/bash
BROKER=$1
if [ -z "$1" ]; then
    BROKER="localhost:9092"
fi

cat UserBehavior_part.csv | while read line; do
    echo $line
    sleep 0.1
done | ~/bigdata/kafka_2.11-2.4.1/bin/kafka-console-producer.sh --broker-list $BROKER --topic user_behavior
```

预备知识

Flink SQL 提供了如下一些与日期处理有关的函数。

❑ TO_TIMESTAMP

将 BIGINT 类型的日期或者 VARCHAR 类型的日期转换成 TIMESTAMP 类型。

语法:

```
TIMESTAMP TO_TIMESTAMP(BIGINT time)           // time: 毫秒
TIMESTAMP TO_TIMESTAMP(VARCHAR date)          // date: yyyy-MM-dd HH:mm:ss
TIMESTAMP TO_TIMESTAMP(VARCHAR date, VARCHAR format)
```

❑ FROM_UNIXTIME

返回值为 VARCHAR 类型的日期值，默认日期格式：yyyy-MM-dd HH:mm:ss，若指定日期格式则按指定格式输出。如果任一输入参数是 NULL，返回 NULL。

语法:

```
VARCHAR FROM_UNIXTIME(BIGINT unixtime[, VARCHAR format])
```

说明:

- 参数 `unixtime` 为长整型，是以秒为单位的时间戳。
- 参数 `format` 可选，为日期格式，默认格式为 yyyy-MM-dd HH:mm:ss，表示返回 VARCHAR 类型的符合指定格式的日期，如果有参数为 null 或解析错误，则返回 null。

7.2.2 创建 Flink 项目

首先使用 IntelliJ IDEA，创建一个 Flink Maven 项目。修改 `pom.xml`，添加如下依赖项。

Scala Maven 依赖:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-scala_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
```

```
<groupId>org.apache.flink</groupId>
<artifactId>flink-streaming-scala_2.12</artifactId>
<version>1.13.2</version>
<scope>provided</scope>
</dependency>

<!-- 连接器依赖 -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-base</artifactId>
  <version>1.13.2</version>
</dependency>

<!-- Kafka 连接器依赖 -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-sql-connector-kafka_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.4.1</version>
</dependency>

<!-- Hive 连接器依赖 -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-hive_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-sql-connector-hive-3.1.2_2.12</artifactId>
  <version>1.13.2</version>
</dependency>

<!-- Hive 依赖 -->
<dependency>
  <groupId>org.apache.hive</groupId>
  <artifactId>hive-exec</artifactId>
  <version>3.1.2</version>
</dependency>

<!-- 表依赖 -->
```

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-scala-bridge_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner-blink_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-common</artifactId>
  <version>1.13.2</version>
</dependency>

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-csv</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-json</artifactId>
  <version>1.13.2</version>
</dependency>
```

Java Maven 依赖:

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-java</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-java_2.12</artifactId>
  <version>1.13.2</version>
  <scope>provided</scope>
</dependency>

<!-- 连接器依赖 -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-base</artifactId>
  <version>1.13.2</version>
</dependency>

<!-- Kafka 连接器依赖 -->
<dependency>
  <groupId>org.apache.flink</groupId>
```

```
<artifactId>flink-connector-kafka_2.12</artifactId>
<version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-sql-connector-kafka_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>2.4.1</version>
</dependency>

<!-- Hive 连接器依赖 -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-hive_2.12</artifactId>
  <version>1.13.2</version>
</dependency>

<!-- Hive 依赖 -->
<dependency>
  <groupId>org.apache.hive</groupId>
  <artifactId>hive-exec</artifactId>
  <version>3.1.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-sql-connector-hive-3.1.2_2.12</artifactId>
  <version>1.13.2</version>
</dependency>

<!-- 表依赖 -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-java-bridge_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner-blink_2.12</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-common</artifactId>
  <version>1.13.2</version>
</dependency>
```

```

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-csv</artifactId>
  <version>1.13.2</version>
</dependency>
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-json</artifactId>
  <version>1.13.2</version>
</dependency>

<!--兼容 Hadoop-->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-shaded-hadoop-3</artifactId>
  <version>3.1.1.7.2.9.0-173-9.0</version>
  <scope>provided</scope>
</dependency>

```

说明：如果 flink-shaded-hadoop-3-xxx-xx-xx 无法通过 Maven 正确加载，请手工将其添加到项目依赖中。

Flink 集成 Hive 时不支持嵌入式 metastore 时，需要对 Hive 的配置做出修改，在 hive-site.xml 文件中配置 hive.metastore.uris，并运行一个 hive metastore。

配置 hive.metastore.uris 的内容如下：

```

<property>
  <name>hive.metastore.uris</name>
  <value>thrift://localhost:9083</value>
</property>

```

因此，在项目的 resources 目录下，创建 hive-site.xml 文件，并编辑内容如下所示：

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hive.metastore.warehouse.dir</name>
    <value>hdfs://192.168.190.133:8020/user/hive/warehouse</value>
  </property>
  <property>
    <name>spark.sql.hive.metastore.version</name>
    <value>3.1.2</value>
  </property>
  <property>
    <name>hive.metastore.uris</name>
    <value>thrift://192.168.190.133:9083</value>
  </property>

  <!--配置 Hive Metastore:MySQL 连接信息 -->
  <property>
    <name>javax.jdo.option.ConnectionURL</name>
    <!--<value>jdbc:mariadb://192.168.190.133:3306/hive?characterEncoding=UTF-8</value>-->
  </property>

```

```

<value>jdbc:mysql://192.168.190.133:3306/hive?characterEncoding=UTF-8</value>
</property>
<property>
  <name>javax.jdo.option.ConnectionDriverName</name>
  <!--<value>org.mariadb.jdbc.Driver</value>-->
  <value>com.mysql.jdbc.Driver</value>
</property>
<property>
  <name>javax.jdo.option.ConnectionUserName</name>
  <value>root</value>
</property>
<property>
  <name>javax.jdo.option.ConnectionPassword</name>
  <value>admin</value>
</property>
</configuration>

```

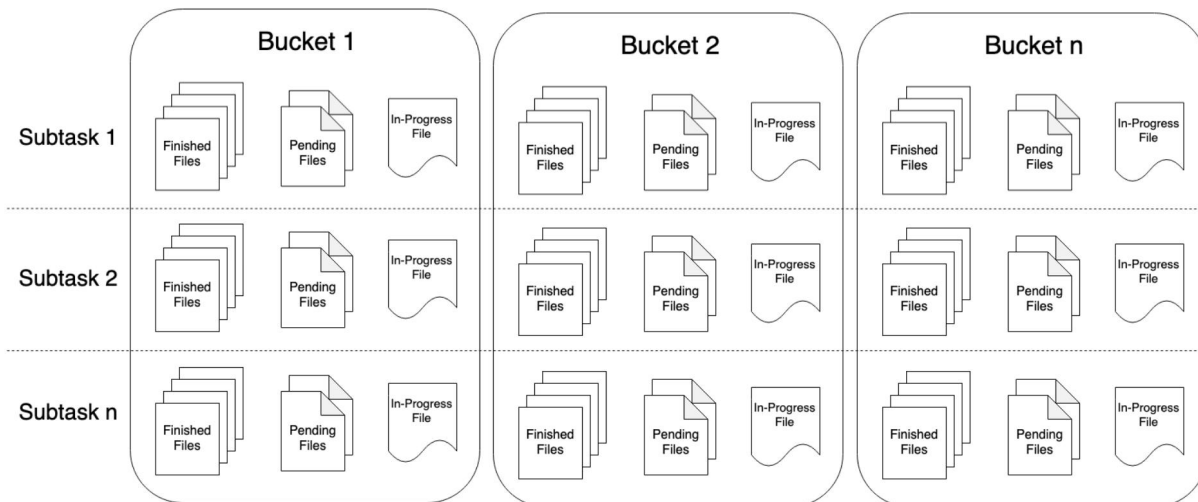
注：请将其中的 IP 地址和数据库连接信息改为你自己的连接信息。如果是使用本教材配置的 PBAP 大数据分析平台，则只需要修改 IP 地址为自己的 IP 即可。

运行 hive metastore 的命令如下：

```
$ ./bin/hive --service metastore
```

7.2.3 创建执行环境

在 Flink 1.13.2 的 Table/SQL API 中，FileSystem 连接器是由 StreamingFileSink 组件的一个增强版本（名为 StreamingFileWriter）实现的。我们知道，只有 checkpoint 检查点成功，StreamingFileSink 写入的文件才会从挂起（pending）变为完成（finished），从而允许它们在下流安全地被读取。所以必须开启检查点并设置一个合理的间隔。



创建执行环境的代码如下所示。

Scala 代码：

```

import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.table.api.EnvironmentSettings
import org.apache.flink.table.api.TableEnvironment

```

```
import org.apache.flink.table.api.bridge.scala.StreamTableEnvironment

val env = StreamExecutionEnvironment.getExecutionEnvironment
env.enableCheckpointing(20000).setParallelism(2)

val settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode()
    .build()

// val tEnv = TableEnvironment.create(setting)
// val tEnv = StreamTableEnvironment.create(env)
val tEnv = StreamTableEnvironment.create(streamEnv, settings)

tEnv.getConfig
    .getConfiguration
    .set(ExecutionCheckpointingOptions.CHECKPOINTING_MODE, CheckpointingMode.EXACTLY_ONCE)
tEnv.getConfig
    .getConfiguration
    .set(ExecutionCheckpointingOptions.CHECKPOINTING_INTERVAL, Duration.ofSeconds(20))
```

Java 代码:

```
import org.apache.flink.table.api.EnvironmentSettings;
import org.apache.flink.table.api.TableEnvironment;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.table.api.bridge.java.StreamTableEnvironment;

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.enableCheckpointing(1000).setParallelism(3);

EnvironmentSettings settings = EnvironmentSettings
    .newInstance()
    .inStreamingMode()
    .build();

// TableEnvironment tEnv = TableEnvironment.create(settings);
// StreamTableEnvironment tEnv = StreamTableEnvironment.create(env);
StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);

tEnv.getConfig()
    .getConfiguration()
    .set(ExecutionCheckpointingOptions.CHECKPOINTING_MODE, CheckpointingMode.EXACTLY_ONCE);
tEnv.getConfig()
    .getConfiguration()
    .set(ExecutionCheckpointingOptions.CHECKPOINTING_INTERVAL, Duration.ofSeconds(20));
```

7.2.4 注册 HiveCatalog

接下来，我们在 Hive MetaStore 中注册 HiveCatalog。

Scala 代码：


```

val catalogName    = "myhive"
val defaultDatabase = "default"
val hiveConfDir    = "/home/hduser/bigdata/hive-3.1.2/conf"

val catalog = new HiveCatalog(catalogName, defaultDatabase, hiveConfDir)
tEnv.registerCatalog(catalogName, catalog)

// 将 HiveCatalog 设置为会话的当前目录
tEnv.useCatalog(catalogName)

```

Java 代码:

```

String catalogName    = "myhive";
String defaultDatabase = "default";
String hiveConfDir    = "/home/hduser/bigdata/hive-3.1.2/conf";

HiveCatalog catalog = new HiveCatalog(catalogName, defaultDatabase, hiveConfDir);
tEnv.registerCatalog(catalogName, catalog);

// 将 HiveCatalog 设置为会话的当前目录
tEnv.useCatalog(catalogName);

```

7.2.5 创建 Kafka 流表

Kafka topic 存储 json 格式的日志信息，并在构建表时计算事件时间和水印列。

Scala 代码:

// 创建 Kafka 源表

```

tEnv.executeSql(
  """
    |CREATE TABLE stream_db.user_behavior_kafka (
    |  user_id bigint,
    |  item_id bigint,
    |  category_id bigint,
    |  behavior string,
    |  ts bigint,
    |  procTime AS PROCTIME(),
    |  eventTime AS TO_TIMESTAMP(FROM_UNIXTIME(ts, 'yyyy-MM-dd HH:mm:ss')),
    |  WATERMARK FOR eventTime AS eventTime - INTERVAL '5' SECONDS
    |)
    |with(
    |  'connector'='kafka',
    |  'topic'='user_behavior',
    |  'properties.bootstrap.servers'='xueai8:9092',
    |  'properties.group.id'='testGroup',
    |  'scan.startup.mode'='latest-offset',
    |  'format'='csv',
    |  'csv.ignore-parse-errors' = 'true',
    |  'csv.field-delimiter'=',',
    |)
  """.stripMargin
)

```

Java 代码:

// 创建 Kafka 源表

```
tEnv.executeSql(
    "CREATE TABLE stream_db.user_behavior_kafka (" +
    "  user_id bigint," +
    "  item_id bigint," +
    "  category_id bigint," +
    "  behavior string," +
    "  ts bigint," +
    "  procTime AS PROCTIME()," +
    "  eventTime AS TO_TIMESTAMP(FROM_UNIXTIME(ts, 'yyyy-MM-dd HH:mm:ss'))," +
    "  WATERMARK FOR eventTime AS eventTime - INTERVAL '5' SECONDS" +
    ") " +
    "with(" +
    "  'connector'='kafka'," +
    "  'topic'='user_behavior'," +
    "  'properties.bootstrap.servers'='xueai8:9092'," +
    "  'properties.group.id'='testGroup'," +
    "  'scan.startup.mode'='latest-offset'," +
    "  'format'='csv'," +
    "  'csv.ignore-parse-errors'='true'," +
    "  'csv.field-delimiter'=''," +
    ")"
);
```

HiveCatalog 之前已经注册过,所以我们可以 在 Hive 中看到创建的 Kafka 流表的元数据。在 Hive Cli 中执行如下命令来查看:

```
hive> DESCRIBE FORMATTED stream_db.user_behavior_kafka;
```

可以看到如下这样的元数据信息 (注意, 该表没有事实上的列):

```
# col_name          data_type          comment

# Detailed Table Information
Database:            stream_db
OwnerType:           USER
Owner:               null
CreateTime:          Wed Sep 15 10:58:44 CST 2021
LastAccessTime:      UNKNOWN
Retention:           0
Location:             hdfs://xueai8:8020/user/hive/warehouse/stream_db.db/user_behavior_kafka
Table Type:          MANAGED_TABLE
Table Parameters:
    flink.connector      kafka
    flink.csv.field-delimiter ,
    flink.csv.ignore-parse-errors true
    flink.format          csv
    flink.properties.bootstrap.servers 192.168.190.133:9092
    flink.properties.group.id      testGroup
    flink.scan.startup.mode latest-offset
    flink.schema.0.data-type      BIGINT
```

```

    flink.schema.0.name      user_id
    flink.schema.1.data-type BIGINT
    flink.schema.1.name      item_id
    flink.schema.2.data-type BIGINT
    flink.schema.2.name      category_id
    flink.schema.3.data-type VARCHAR(2147483647)
    flink.schema.3.name      behavior
    flink.schema.4.data-type BIGINT
    flink.schema.4.name      ts
    flink.schema.5.data-type TIMESTAMP(3) WITH LOCAL TIME ZONE NOT NULL
    flink.schema.5.expr      PROCTIME()
    flink.schema.5.name      procTime
    flink.schema.6.data-type TIMESTAMP(3)
    flink.schema.6.expr      TO_TIMESTAMP(FROM_UNIXTIME(`ts`, 'yyyy-MM-dd HH:mm:ss'))
    flink.schema.6.name      eventTime
    flink.schema.watermark.0.rowtime      eventTime
    flink.schema.watermark.0.strategy.data-type      TIMESTAMP(3)
    flink.schema.watermark.0.strategy.expr `eventTime` - INTERVAL '5' SECOND
    flink.topic      user_behavior
    transient_lastDdlTime 1631674724

# Storage Information
SerDe Library:      org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
InputFormat:        org.apache.hadoop.mapred.TextInputFormat
OutputFormat:       org.apache.hadoop.hive.ql.io.IgnoreKeyTextOutputFormat
Compressed:         No
Num Buckets:        -1
Bucket Columns:     []
Sort Columns:       []
Storage Desc Params:
    serialization.format 1
Time taken: 1.282 seconds, Fetched: 51 row(s)

```

7.2.6 创建 Hive 表

Flink SQL 提供符合 HiveQL 的 DDL，指定 SqlDialect。

为了观察结果，下面的表使用了 days/hours/minutes 的三级分区，在实践中可能没有那么细粒度(10 分钟甚至 1 小时的分区可能更合适)。

Scala 代码：

```

tableEnv.getConfig.setSqlDialect(SqlDialect.HIVE)    // 切换到 HiveQL 方言

tableEnv.executeSql("CREATE DATABASE IF NOT EXISTS hive_tmp")
tableEnv.executeSql("DROP TABLE IF EXISTS hive_tmp.analytics_access_log_hive")

tableEnv.executeSql(
    """
        |CREATE TABLE hive_db.user_behavior_hive (
        |  user_id bigint,
        |  item_id bigint,
    """

```

```

        | category_id bigint,
        | behavior string,
        | ts bigint
    ) PARTITIONED BY (
        | ts_date STRING,
        | ts_hour STRING,
        | ts_minute STRING
    ) STORED AS PARQUET
    TBLPROPERTIES (
        | 'sink.partition-commit.trigger' = 'partition-time',
        | 'sink.partition-commit.delay' = '1 min',
        | 'sink.partition-commit.policy.kind' = 'metastore,success-file',
        | 'partition.time-extractor.timestamp-pattern' = '$ts_date $ts_hour:$ts_minute:00'
    )
    """
    .stripMargin
)

```

Java 代码:

```
tEnv.getConfig().setSqlDialect(SqlDialect.HIVE); // 切换为 HiveQL 方言
```

```
tEnv.executeSql("CREATE DATABASE IF NOT EXISTS hive_db");
```

```
tEnv.executeSql("DROP TABLE IF EXISTS hive_db.user_behavior_hive");
```

```

tEnv.executeSql(
    "CREATE TABLE hive_db.user_behavior_hive (" +
    "    user_id bigint," +
    "    item_id bigint," +
    "    category_id bigint," +
    "    behavior string," +
    "    ts bigint" +
    ") PARTITIONED BY (" +
    "    ts_date STRING," +
    "    ts_hour STRING," +
    "    ts_minute STRING" +
    ") STORED AS PARQUET\n" +
    " TBLPROPERTIES (" +
    "    'sink.partition-commit.trigger' = 'partition-time'," +
    "    'sink.partition-commit.delay' = '1 min'," +
    "    'sink.partition-commit.policy.kind' = 'metastore,success-file'," +
    "    'partition.time-extractor.timestamp-pattern' = '$ts_date $ts_hour:$ts_minute:00'" +
    ")"
);

```

Hive 表的参数复用了 SQL FileSystem Connector 的参数，与分区提交密切相关。下面简单介绍一下上述四个参数。

- ❑ `sink.partition-commit.trigger`。触发分区提交的时间特征。默认值是处理时间。所以这里使用分区时间 `partition-time`，它是根据分区时间戳提交的，分区时间戳是分区中数据的事件时间。
- ❑ `partition.time-extractor.timestamp-pattern`。分区时间戳的抽取格式。需要写成 `yyyy-MM-dd HH:mm:ss`，并替换为 Hive 表中对应的分区字段。显然，Hive 表的分区字段值来自流表中定义的事件时间。

- ❑ `sink.partition-commit.delay`。触发分区提交中的延迟。当时间特征被设置为 `partition-time` 时，分区将不会实际提交，直到水印时间戳大于分区创建时间加上这个延迟。该参数值最好与分区粒度相同，例如 Hive 表按 1 小时进行分区时，可设置为 1 小时，按 10 分钟进行分区时，可设置为 10 分钟。
- ❑ `sink.partition-commit.policy.kind`。分区提交策略可以理解为使分区在下游可见的附加操作。`metastore` 表示更新 Hive metastore 中的表元数据，`success-file` 表示 `create_in partitionSUCCESS` 标签文件。先更新 metastore (添加分区)，然后写入成功文件 `'_SUCCESS'`。

在 hive 中可以查看元数据（程序执行之后才会创建）：

```
hive> DESCRIBE FORMATTED hive_db.user_behavior_hive;
```

可以看到如下这样的元数据信息：

```
# col_name      data_type      comment
user_id         bigint
item_id         bigint
category_id     bigint
behavior        string
ts              bigint

# Partition Information
# col_name      data_type      comment
ts_date         string
ts_hour         string
ts_minute       string

# Detailed Table Information
Database:        hive_db
OwnerType:       USER
Owner:           null
CreateTime:      Wed Sep 15 10:58:46 CST 2021
LastAccessTime:  UNKNOWN
Retention:       0
Location:        hdfs://xueai8:8020/user/hive/warehouse/hive_db.db/user_behavior_hive
Table Type:      MANAGED_TABLE
Table Parameters:
    COLUMN_STATS_ACCURATE  {"BASIC_STATS\":"true"}
    bucketing_version      2
    numFiles                0
    numPartitions           0
    numRows                0
    partition.time-extractor.timestamp-pattern  $ts_date $ts_hour:$ts_minute:00
    rawDataSize            0
    sink.partition-commit.delay  1 min
    sink.partition-commit.policy.kind  metastore,success-file
    sink.partition-commit.trigger  partition-time
    totalSize              0
    transient_lastDdlTime   1631674726

# Storage Information
```

```
SerDe Library:      org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe
InputFormat:        org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat
OutputFormat:       org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat
Compressed:         No
Num Buckets:        -1
Bucket Columns:     []
Sort Columns:       []
Storage Desc Params:
    serialization.format    1
Time taken: 0.558 seconds, Fetched: 46 row(s)
```

7.2.7 流写 Hive 表

接下来，实现读取 Kafka topic 并实时写入 Hive 数据仓库的功能。注意，在本例中，流表中的事件时间被转换为 Hive 的分区。

Scala 代码：

```
tEnv.getConfig.setSqlDialect(SqlDialect.DEFAULT)    // 切换为标准 SQL

tEnv.executeSql(
    """
    |INSERT INTO hive_db.user_behavior_hive
    |SELECT
    |  user_id,item_id,category_id,behavior,ts,
    |  DATE_FORMAT(eventTime,'yyyy-MM-dd') as ts_date,
    |  DATE_FORMAT(eventTime,'HH') as ts_hour,
    |  DATE_FORMAT(eventTime,'mm') as ts_minute
    |FROM stream_db.user_behavior_kafka
    |WHERE item_id > 0
    """).stripMargin
)
```

Java 代码：

```
tEnv.getConfig().setSqlDialect(SqlDialect.DEFAULT);    // 切换为标准 SQL

tEnv.executeSql(
    "INSERT INTO hive_db.user_behavior_hive\n" +
    "SELECT\n" +
    "  user_id,item_id,category_id,behavior,ts," +
    "  DATE_FORMAT(eventTime,'yyyy-MM-dd') as ts_date," +
    "  DATE_FORMAT(eventTime,'HH') as ts_hour," +
    "  DATE_FORMAT(eventTime,'mm') as ts_minute\n" +
    "FROM stream_db.user_behavior_kafka\n" +
    "WHERE item_id > 0"
);
```

看看流接收器的结果。访问 url: <http://192.168.190.133:9870/>。

下面是以小时为分区：

/user/hive/warehouse/hive_db.db/user_behavior_hive/ts_date=2017-11-26/ts_hour=09 Go!

Show 25 entries Search:

☐	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	☐
☐	drwxr-xr-x	hduser	supergroup	0 B	Sep 15 11:42	0	0 B	ts_minute=00	☐
☐	drwxr-xr-x	hduser	supergroup	0 B	Sep 15 11:43	0	0 B	ts_minute=01	☐
☐	drwxr-xr-x	hduser	supergroup	0 B	Sep 15 11:43	0	0 B	ts_minute=02	☐

下面是以分钟为分区：

/user/hive/warehouse/hive_db.db/user_behavior_hive/ts_date=2017-11-26/ts_hour=09/ts_minute=01 Go!

Show 25 entries Search:

☐	Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name	☐
☐	-rw-r--r--	hduser	supergroup	0 B	Sep 15 11:43	3	128 MB	__SUCCESS	☐
☐	-rw-r--r--	hduser	supergroup	1012 B	Sep 15 11:42	3	128 MB	part-92fb0bd7-4584-4f33-ae89-fccc9c7652d1-0-4	☐
☐	-rw-r--r--	hduser	supergroup	5.35 KB	Sep 15 11:42	3	128 MB	part-92fb0bd7-4584-4f33-ae89-fccc9c7652d1-0-5	☐
☐	-rw-r--r--	hduser	supergroup	5.51 KB	Sep 15 11:42	3	128 MB	part-92fb0bd7-4584-4f33-ae89-fccc9c7652d1-0-6	☐
☐	-rw-r--r--	hduser	supergroup	5.4 KB	Sep 15 11:43	3	128 MB	part-92fb0bd7-4584-4f33-ae89-fccc9c7652d1-0-7	☐
☐	-rw-r--r--	hduser	supergroup	3.36 KB	Sep 15 11:43	3	128 MB	part-92fb0bd7-4584-4f33-ae89-fccc9c7652d1-0-8	☐

因为并行度是 2，所以每次写都会生成两个文件。当写入分区中的所有数据时，同时生成一个 __success 文件。如果它是一个正在写入的分区，将看到.inprogress 文件。

检查 Hive 查询以确保数据是正确的（请确保已经启动了 YARN 集群）。

```
hive> SELECT from_unixtime(min(cast(ts AS BIGINT))),from_unixtime(max(cast(ts AS BIGINT)))
> FROM hive_db.user_behavior_hive
> WHERE ts_date = '2017-11-26' AND ts_hour = '09' AND ts_minute = '01';
```

查询结果如下所示：

```
2017-11-26 01:01:00      2017-11-26 01:01:59
Time taken: 135.823 seconds, Fetched: 1 row(s)
```

执行过程如下图所示：

```

hive> SELECT from_unixtime(min(cast(ts AS BIGINT))),from_unixtime(max(cast(ts AS BIGINT)))
> FROM hive_db.user_behavior_hive
> WHERE ts_date = '2017-11-26' AND ts_hour = '09' AND ts_minute = '01';
Query ID = hduser_20210915142122_4c535f3e-f733-4aba-904b-1cd3c0a4d1cd
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1631686931410_0001, Tracking URL = http://xueai8:8088/proxy/application_1631686931410_0001/
Kill Command = /home/hduser/bigdata/hadoop-3.2.2/bin/mapred job -kill job_1631686931410_0001
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2021-09-15 14:23:09,672 Stage-1 map = 0%, reduce = 0%
2021-09-15 14:23:24,907 Stage-1 map = 100%, reduce = 0%, Cumulative CPU 4.51 sec
2021-09-15 14:23:36,206 Stage-1 map = 100%, reduce = 100%, Cumulative CPU 6.93 sec
MapReduce Total cumulative CPU time: 6 seconds 930 msec
Ended Job = job_1631686931410_0001
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1 Reduce: 1 Cumulative CPU: 6.93 sec HDFS Read: 23872 HDFS Write: 139 SUCCESS
Total MapReduce CPU Time Spent: 6 seconds 930 msec
OK
2017-11-26 01:01:00      2017-11-26 01:01:59
Time taken: 135.823 seconds, Fetched: 1 row(s)
hive>

```

执行Hive查询

查询结果

7.2.8 动态读取 Hive 流表

为了使 Hive 表成为流源，需要启用动态表选项，并通过表提示为 Hive 数据流指定参数。下面是一个使用 Hive 简单计算商品 PV 的例子。

Scala 代码：

Java 代码：

上面代码中的三个表提示参数的含义解释如下。

- ❑ streaming-source.enable。设置为 true，表示 Hive 表可以作为 Source 表使用。
- ❑ streaming-source.monitor-interval。感知 Hive 表中新增数据的周期，设置为 1 分钟以上。对于分区表，监视新分区的生成以增量读取数据。
- ❑ streaming-source.consume-start-offset。开始消费的时间戳，也需要写成 yyyy-MM-dd HH:mm:ss。

最后，由于 SQL 语句中有 ORDER BY 和 LIMIT 逻辑，需要调用 toRetractStream() 方法将其转换为一个 undo 流以输出结果。

7.2.9 完整示例代码

到目前为止，我们已经完成了如下两个流程：

- ❑ FlinkHiveJob：读取 Kafka topic 并实时写入 Hive 数仓的 Flink 程序
- ❑ HiveAnalysisJob：将上一个作业写入的 Hive 表作为流源表，进行每分钟聚合统计。

下面是本案例两个程序的完整代码。

FlinkHiveJob

Scala 代码：

Java 代码：

HiveAnalysisJob

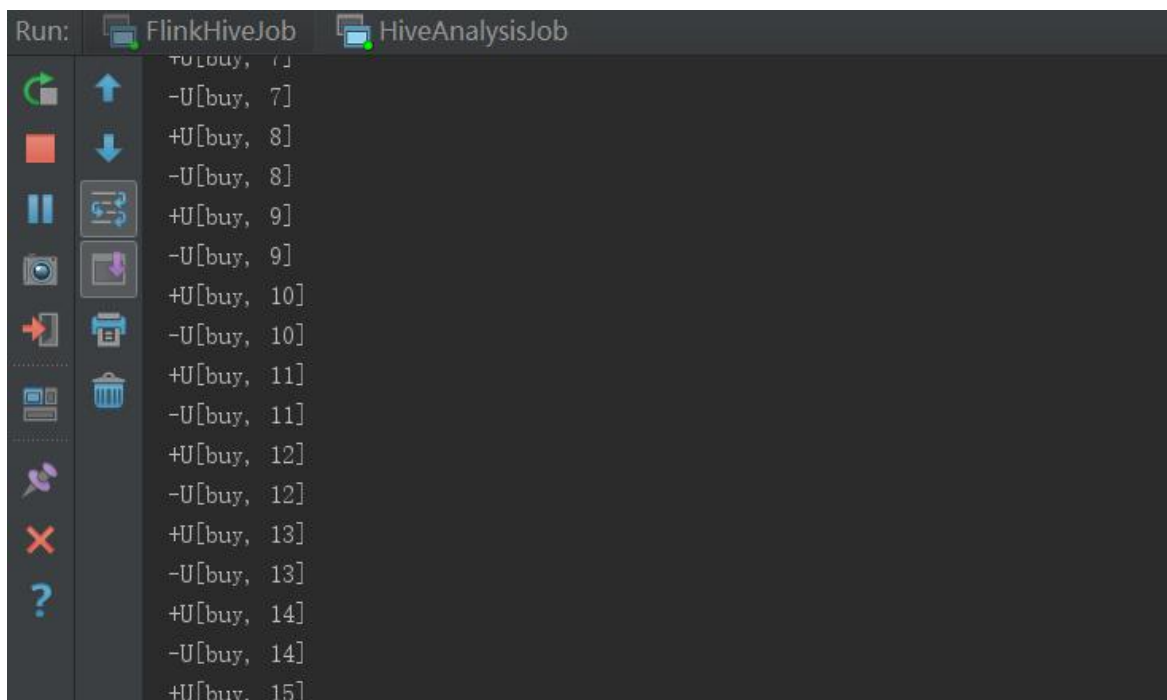
Scala 代码:

Java 代码:

7.2.10 执行步骤

至此，所有的代码已经开发完毕。接下来，请按以下步骤执行。

- 1、启动 Kafka，创建 user_behavior topic。
- 2、启动 Hive Metastore 服务。
- 3、启动 Flink 集群。
- 4、先运行 HiveAnalysisJob 程序，执行连续聚合查询。
- 5、然后运行前面已经开发好的 Flink 流 ETL 程序 FlinkHiveJob，读取 Kafka topic 实时用户购买行为数据，实时入仓。
- 6、执行数据生产者脚本，模拟产生实时用户购买行为数据。
- 7、查看 HiveAnalysisJob 程序统计结果。



7.3 纯 SQL 构建批流一体数仓

在上一节中，我们使用 Flink Table API 集成 Hive 实现了批流一体数据仓库的构建。实际上，对于数据分析人员，我们可以直接使用 Flink SQL CLI 来集成 Hive，实现纯 SQL 无代码构建批流一体数据仓库。

7.3.1 使用 Flink SQL 客户端

Flink 的 Table & SQL API 可以处理使用 SQL 语言编写的查询语句，但是这些查询需要嵌入到用 Java 或 Scala 编写的表程序中。此外，这些程序在提交到集群之前需要用构建工具打包。这或多或少限制了 Java/Scala 程序员使用 Flink。

Flink SQL 客户端的目的是提供一种简单的方式来编写、调试和提交表程序到 Flink 集群上，而无需写一行 Java 或 Scala 代码。Flink 支持的 SQL 语言，包括数据定义语言(DDL)、数据操作语言(DML)和查询语言。Flink 的 SQL 支持基于 Apache Calcite，它实现了 SQL 标准。

Flink SQL 目前支持的 SQL 语句包括：

- ☐ SELECT (Queries)
- ☐ CREATE TABLE, DATABASE, FUNCTION
- ☐ DROP TABLE, DATABASE, FUNCTION
- ☐ ALTER TABLE, DATABASE, FUNCTION
- ☐ INSERT

Flink SQL 客户端命令行界面（CLI）能够在命令行中检索和可视化分布式应用中实时产生的结果。Flink SQL Client 绑定在常规的 Flink 发行版中，因此可以开箱即用。它只需要一个可以执行表程序的正

在运行的 Flink 集群。使用 Flink SQL 客户端的步骤如下。

1) 先启动 Flink 集群:

```
$ cd ~/bigdata/flink-1.13.2
```

```
$ export HADOOP_CLASSPATH=`hadoop classpath`
```

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

2) 再启动 SQL 客户端脚本:

```
$ ./bin/sql-client.sh
```

启动界面如下图所示:



启动 CLI 后, 可以使用 “HELP;” 命令列出所有可用的 SQL 语句。

```
Flink SQL> help;
The following commands are available:

CLEAR          Clears the current terminal.
CREATE TABLE  Create table under current catalog and database.
DROP TABLE    Drop table with optional catalog and database. Syntax: 'DROP TABLE [IF EXISTS] <name>;'
CREATE VIEW    Creates a virtual table from a SQL query. Syntax: 'CREATE VIEW <name> AS <query>;'
DESCRIBE       Describes the schema of a table with the given name.
DROP VIEW      Deletes a previously created virtual table. Syntax: 'DROP VIEW <name>;'
EXPLAIN        Describes the execution plan of a query or table with the given name.
HELP           Prints the available commands.
INSERT INTO    Inserts the results of a SQL SELECT query into a declared table sink.
INSERT OVERWRITE  Inserts the results of a SQL SELECT query into a declared table sink and overwrites.
QUIT          Quits the SQL CLI client.
RESET          Resets a session configuration property. Syntax: 'RESET <key>;'. Use 'RESET;' for reset all session configuration properties.
SELECT         Executes a SQL SELECT query on the Flink cluster.
SET           Sets a session configuration property. Syntax: 'SET <key>=<value>;'. Use 'SET;' for listing all session configuration properties.
SHOW FUNCTIONS Shows all user-defined and built-in functions or only user-defined functions. Syntax: 'SHOW FUNCTIONS [LIKE <pattern>;]'.
SHOW TABLES  Shows all registered tables.
SOURCE         Reads a SQL SELECT query from a file and executes it on the Flink cluster.
USE CATALOG    Sets the current catalog. The current database is set to the catalog's default one. Experimental! Syntax: 'USE CATALOG <name>;'.
USE           Sets the current default database. Experimental! Syntax: 'USE <name>;'.
LOAD MODULE    Load a module. Syntax: 'LOAD MODULE <name> [WITH ('<key1>' = '<value1>' [, '<key2>' = '<value2>'])]'.
UNLOAD MODULE  Unload a module. Syntax: 'UNLOAD MODULE <name>;'.
USE MODULES    Enable loaded modules. Syntax: 'USE MODULES <name1> [, <name2>, ...];'.
BEGIN STATEMENT SET Begins a statement set. Syntax: 'BEGIN STATEMENT SET;'.
END           Ends a statement set. Syntax: 'END;'.

Hint: Make sure that a statement ends with ';' for finalizing (multi-line) statements.

Flink SQL> █
```

CLI 支持三种模式来维护和可视化结果。

- ❑ 表格模式 (table model) 将结果物化到内存中, 并以常规的分页表表示形式将其可视化。可以通过在 CLI 中执行以下命令来启用它 (默认使用表格模式):

```
Flink SQL> SET sql-client.execution.result-mode=table;
```

- ❑ 变更日志模式 (changelog model) 不物化结果, 而是可视化由插入(+)和撤消(-)组成的连续查询产生的结果流。

```
Flink SQL> SET sql-client.execution.result-mode=changelog;
```

- ❑ 表模式 (tableau model) 更像是一种传统的方式, 它将以 tableau 格式直接在屏幕上显示结果。显示内容将受到查询执行类型(execution.type)的影响。

```
Flink SQL> SET sql-client.execution.result-mode=tableau;
```

请注意, 当在流查询中使用 tableau 模式时, 结果将在控制台上连续打印。如果此查询的输入数据是有界的, 则在 Flink 处理完所有输入数据后, 作业将终止, 打印也将自动停止。否则, 如果想终止一个正在运行的查询, 只需键入 CTRL-C, 在本例中, 作业和打印将停止。

在构建 SQL 查询原型期间, 这三种结果模式都很有用。在所有这些模式中, 结果都存储在 SQL 客户端的 Java 堆内存中。为了保持 CLI 界面的响应, changelog 模式只显示最新的 1000 个更改。表模式允许在仅受可用主存和配置的最大行数 (max-table-result-rows) 限制的更大结果中进行导航。

注意: 在批处理环境中执行的查询只能使用 table 或 tableau 结果模式检索。

为了验证设置和集群连接, 可以输入第一个 SQL 查询, 并按 enter 键执行它:

```
Flink SQL> select 'Hello World';
```

这个查询不需要表源, 只产生一个 row 结果。CLI 将从集群中检索结果并对其进行可视化。按 “Q” 键可以关闭结果视图。

执行过程如下图所示:

```
Flink SQL> SET sql-client.execution.result-mode=tableau;
[INFO] Session property has been set.

Flink SQL> select 'Hello World';
+-----+-----+
| op |                               EXPR$0 |
+-----+-----+
| +I |                               Hello World |
+-----+-----+
Received a total of 1 row

Flink SQL> █
```

要退出 Flink SQL CLI 命令行，执行如下命令：

```
Flink SQL> exit;
```

7.3.2 集成 Flink SQL Cli 和 Hive

我们可以使用 Flink SQL Cli 来直接连接 Hive，而无需使用 Table API。

依赖添加

将所有 Hive 依赖项（下面的 jar 列表）添加到 Flink 分布式安装的/lib 目录下。



如下图所示：

编辑 hive-site.xml 文件

将 Hive 安装配置文件 hive-site.xml 拷贝到 Flink 安装的 conf/ 目录下，或者在该位置新建一个 hive-site.xml 配置文件。编辑其内容如下：

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
  <property>
    <name>hive.metastore.warehouse.dir</name>
    <value>hdfs://xueai8:8020/user/hive/warehouse</value>
  </property>
  <property>
    <name>spark.sql.hive.metastore.version</name>
    <value>3.1.2</value>
  </property>
  <property>
    <name>hive.metastore.uris</name>
    <value>thrift://xueai8:9083</value>
  </property>

  <!--配置 Hive Metastore:MySQL 连接信息 -->
  <property>
    <name>javax.jdo.option.ConnectionURL</name>
```

```
<!--<value>jdbc:mariadb://localhost:3306/hive?characterEncoding=UTF-8</value>-->
<value>jdbc:mysql://localhost:3306/hive?characterEncoding=UTF-8</value>
</property>
<property>
  <name>javax.jdo.option.ConnectionDriverName</name>
  <!--<value>org.mariadb.jdbc.Driver</value>-->
  <value>com.mysql.jdbc.Driver</value>
</property>
<property>
  <name>javax.jdo.option.ConnectionUserName</name>
  <value>root</value>
</property>
<property>
  <name>javax.jdo.option.ConnectionPassword</name>
  <value>admin</value>
</property>
</configuration>
```

注：请将其中的 IP 地址和数据库连接信息改为你自己的连接信息。如果是使用本教材配置的 PBAP 大数据分析平台，则只需要修改 IP 地址为自己的 IP 即可。

Flink 集成 Hive 时不支持 embedded metastore 时，需要对 Hive 的配置做出修改，在 hive-site.xml 文件中配置 hive.metastore.uris，并运行一个 hive metastore。

配置 hive.metastore.uris 的内容如下：

```
<property>
  <name>hive.metastore.uris</name>
  <value>thrift://localhost:9083</value>
</property>
```

运行 hive metastore 的命令如下：

```
$ ./bin/hive --service metastore
```

7.3.3 注册 HiveCatalog

HiveCatalog 会连接 Hive metastore 并桥接 Flink 和 Hive 之间的元数据。目前，这个 HiveCatalog 可以提供读取 Hive 元数据的能力，包括数据库（databases），表（tables），表分区（table partitions），简单的数据类型（simple data types），表和列的统计信息（table and column stats）。

Flink 可以通过 HiveCatalog 读取 Hive 的 MetaData。Flink 还实现了 HiveTableSource，使得 Flink job 可以直接读取 Hive 中普通表和分区表的数据，以及做分区的裁剪。

Flink 使用 HiveCatalog 可以通过批或者流的方式来处理 Hive 中的表。这就意味着 Flink 既可以作为 Hive 的一个批处理引擎，也可以通过流处理的方式来读写 Hive 中的表，从而支持实时数仓的应用和流批一体的落地。

Flink 1.13 开始的配置方式

从 Flink 1.13 开始，Flink 推荐使用 SQL 脚本来初始话 session 会话。

Flink SQL 查询需要一个执行它的配置环境。Flink SQL Client 支持 -i 启动选项，在启动 Flink SQL Client 时执行一个初始化 SQL 文件来设置环境。所谓的初始化 SQL 文件可以使用 Flink DDL 来定义可用的 catalog 目录、表源和表 sink、用户定义函数以及执行和部署所需的其他属性。

参考: <https://cwiki.apache.org/confluence/display/FLINK/FLIP-163%3A+SQL+Client+Improvements>

这时, 启动 Flink SQL Client 的语法如下所示:

```
> ./sql-client.sh -i init1.sql,init2.sql
```

其中多个初始化 SQL 文件使用逗号进行分隔。

例如, 我们在 Flink 配置目录 conf 下, 创建一个用于 Flink SQL Client 初始化的 init.sql 文件, 编辑其内容如下:

```
-- 定义可用的 catalogs

CREATE CATALOG MyHiveCatalog
WITH (
  'type' = 'hive',
  'default-database' = 'default',
  'hive-conf-dir' = '/home/hduser/bigdata/flink-1.13.2/conf'
);

USE CATALOG MyHiveCatalog;
```

在上面这个初始化脚本文件中, 连接 Hive catalog 目录, 使用 MyHiveCatalog 作为当前目录, 并指定在 tableau 结果模式下运行探索性查询。

读写 Hive 表

因为 Hive 表对应的数据仓库地址在 HDFS 上, 所以要先启动 HDFS。命令如下:

```
$ start-dfs.sh
```

然后启动 Flink 集群。命令如下:

```
$ export HADOOP_CLASSPATH=`$HADOOP_HOME/bin/hadoop classpath`
```

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

接下来启动 FlinkSQL Cli, 命令如下:

```
$ ./bin/sql-client.sh -i conf/init.sql
```

接下来, 我们可以查看注册的 catalog:

```
> show catalogs;
```

可以看到如下的输出内容:

切换到 MyHiveCatalog:

```
> use catalog MyHiveCatalog;
```

查看 MyHiveCatalog 目录下包含有哪些数据库:

```
> show databases;
```

查看默认当前数据库 (即 default) 中包含有哪些数据表:

```
> show tables;
```

执行过程如下图所示:

在 Flink SQL Client 中执行如下 SQL 语句创建并操作 Hive 表。


```
-- 使用 hive 方言的 DDL 语法
SET table.sql-dialect=hive;

-- 创建表
CREATE TABLE test_hive (
    name string,
    age int
) STORED AS ORC;

-- 查看数据表
show tables;

-- 插入数据
INSERT INTO test_hive SELECT '张三', 20;

-- 设置结果显示模式为 tableau
SET sql-client.execution.result-mode=tableau;

-- 执行查询
select * from test_hive;
```

执行结果如下图所示：

如果要删除 test_hive 表，执行下面的 SQL 语句：

```
> drop table test_hive;
```

7.3.4 使用 SQL Client 提交作业

当执行查询或插入语句时，Flink SQL Client 支持使用进入交互模式或使用 -f 选项提交 SQL 语句。用户在交互式命令行中提交作业，或者使用 -f 选项来执行 SQL 文件。在这两种模式中，Flink SQL Client 都支持解析和执行 Flink 支持的所有类型的 SQL 语句。

交互式命令行

在交互式命令行中，Flink SQL Client 读取用户输入并在获取分号(;)时执行语句。

如果语句执行成功，Flink SQL Client 将打印成功消息。当获取错误时，SQL Client 也将打印错误消息。默认情况下，错误信息只包含错误原因。为了输出完整的异常堆栈进行调试，请通过以下命令设置 sql-client.verbose 为 true。

```
> SET 'sql-client.verbose' = 'true'
```

执行 SQL 文件

SQL Client 支持执行带有 -f 选项的 SQL 脚本文件。SQL Client 将逐个执行 SQL 脚本文件中的语句，并为每个已执行语句打印执行消息。一旦一条语句失败，SQL Client 就会退出，所有剩余的语句都不会被执行。

下面是这样一个文件的示例。


```
CREATE TEMPORARY TABLE users (  
  user_id BIGINT,  
  user_name STRING,  
  user_level STRING,  
  region STRING,  
  PRIMARY KEY (user_id) NOT ENFORCED  
) WITH (  
  'connector' = 'upsert-kafka',  
  'topic' = 'users',  
  'properties.bootstrap.servers' = '...',  
  'key.format' = 'csv',  
  'value.format' = 'avro'  
);  
  
-- 设置同步模式  
SET 'table.dml-sync' = 'true';  
  
-- 设置作业名称  
SET 'pipeline.name' = 'SqlJob';  
  
-- 设置作业提交的队列  
SET 'yarn.application.queue' = 'root';  
  
-- 设置作业并行性  
SET 'parallism.default' = '100';  
  
-- 从指定保存点路径恢复  
SET 'execution.savepoint.path' = '/tmp/flink-savepoints/savepoint-cca7bc-bb1e257f0dab';  
  
-- 定义了查询之后，可以将它作为一个长期运行的、分离的 Flink 作业提交给集群。  
-- 为此，需要使用 INSERT INTO 语句指定存储结果的目标系统。  
INSERT INTO pageviews_enriched  
SELECT *  
FROM pageviews AS p  
LEFT JOIN users FOR SYSTEM_TIME AS OF p.proctime AS u  
ON p.user_id = u.user_id;
```

在上面的配置中，我们：

- ☐ 定义从 CSV 文件中读取的时态表源 `users`，
- ☐ 设置属性，例如作业名称，
- ☐ 设置保存点路径，
- ☐ 提交一个从指定保存点路径加载保存点的 SQL 作业。

执行一组 SQL 语句

Flink SQL 客户端作为单个 Flink 作业执行每个 INSERT INTO 语句。然而，这有时不是最佳的，因为管道的某些部分可以重用。Flink SQL Client 支持 STATEMENT SET 语法来执行一组 SQL 语句。这是 Table API 中 StatementSet 的等价特性。STATEMENT SET 语法包含一个或多个 INSERT INTO 语句。STATEMENT SET 块中的所有语句都进行了整体优化，并作为单个 Flink 作业执行。联合优化和执行允

许重用公共的中间结果，因此可以显著提高执行多个查询的效率。

STATEMENT SET 语法如下：

```
BEGIN STATEMENT SET;
-- 一个或多个 INSERT INTO 语句
{ INSERT INTO|OVERWRITE <select_statement>; }+
END;
```

STATEMENT SET 中包含的语句必须用分号(;)分隔。

在交互式模式下，执行 STATEMENT SET 的示例如下：

```
Flink SQL> CREATE TABLE pageviews (
>   user_id BIGINT,
>   page_id BIGINT,
>   viewtime TIMESTAMP,
>   proctime AS PROCTIME()
> ) WITH (
>   'connector' = 'kafka',
>   'topic' = 'pageviews',
>   'properties.bootstrap.servers' = '...',
>   'format' = 'avro'
> );
[INFO] Execute statement succeed.

Flink SQL> CREATE TABLE pageview (
>   page_id BIGINT,
>   cnt BIGINT
> ) WITH (
>   'connector' = 'jdbc',
>   'url' = 'jdbc:mysql://localhost:3306/mydatabase',
>   'table-name' = 'pageview'
> );
[INFO] Execute statement succeed.

Flink SQL> CREATE TABLE uniqueview (
>   page_id BIGINT,
>   cnt BIGINT
> ) WITH (
>   'connector' = 'jdbc',
>   'url' = 'jdbc:mysql://localhost:3306/mydatabase',
>   'table-name' = 'uniqueview'
> );
[INFO] Execute statement succeed.

Flink SQL> BEGIN STATEMENT SET;
[INFO] Begin a statement set.

Flink SQL> INSERT INTO pageviews
> SELECT page_id, count(1)
> FROM pageviews
> GROUP BY page_id;
[INFO] Add SQL update statement to the statement set.
```

```
Flink SQL> INSERT INTO uniqueview
> SELECT page_id, count(distinct user_id)
> FROM pageviews
> GROUP BY page_id;
[INFO] Add SQL update statement to the statement set.

Flink SQL> END;
[INFO] Submitting SQL update statement to the cluster...
[INFO] SQL update statement has been successfully submitted to the cluster:
Job ID: 6b1af540c0c0bb3fcfcad50ac037c862
```

在 SQL 文件中，执行 STATEMENT SET 的示例如下：

```
CREATE TABLE pageviews (
  user_id BIGINT,
  page_id BIGINT,
  viewtime TIMESTAMP,
  proctime AS PROCTIME()
) WITH (
  'connector' = 'kafka',
  'topic' = 'pageviews',
  'properties.bootstrap.servers' = '...',
  'format' = 'avro'
);

CREATE TABLE pageview (
  page_id BIGINT,
  cnt BIGINT
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://localhost:3306/mydatabase',
  'table-name' = 'pageview'
);

CREATE TABLE uniqueview (
  page_id BIGINT,
  cnt BIGINT
) WITH (
  'connector' = 'jdbc',
  'url' = 'jdbc:mysql://localhost:3306/mydatabase',
  'table-name' = 'uniqueview'
);

BEGIN STATEMENT SET;

INSERT INTO pageviews
SELECT page_id, count(1)
FROM pageviews
GROUP BY page_id;
```

```
INSERT INTO uniqueview
SELECT page_id, count(distinct user_id)
FROM pageviews
GROUP BY page_id;

END;
```

默认情况下，SQL Client 异步执行 DML 语句。这意味着，SQL Client 将向 Flink 集群提交 DML 语句的作业，而不是等待作业完成。因此，SQL Client 可以同时提交多个作业。这对于流作业非常有用，这些作业通常是长时间运行的。

SQL Client 确保一条语句成功提交到集群。一旦提交了语句，CLI 将显示关于 Flink 作业的信息。

```
Flink SQL> INSERT INTO MyTableSink SELECT * FROM MyTableSource;
[INFO] Table update statement has been successfully submitted to the cluster:
Cluster ID: StandaloneClusterId
Job ID: 6f922fe5c8a87406ff23ae4a7bb79044
```

SQL Client 在提交后不会跟踪正在运行的 Flink 作业的状态。CLI 进程可以在提交后关闭，而不影响分离查询。Flink 的 restart strategy（重启策略）解决了容错问题。可以使用 Flink 的 web 界面、命令行或 REST API 取消查询。

然而，对于批处理用户，更常见的情况是，下一个 DML 语句需要等待，直到前一个 DML 语句完成。为了同步执行 DML 语句，可以设置表。在 SQL Client 中设置 table.dml-sync 选项为 true。

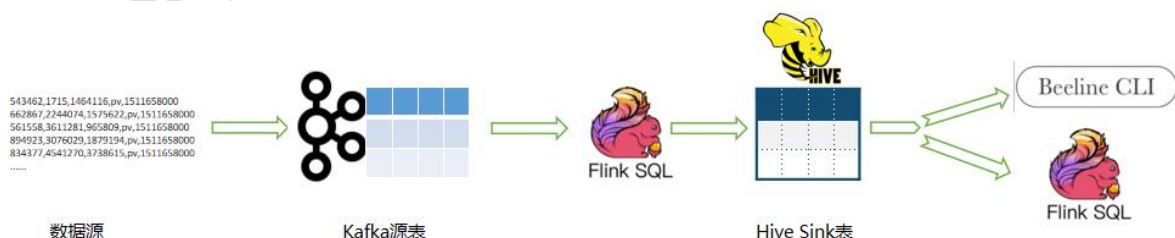
```
Flink SQL> SET 'table.dml-sync' = 'true';
[INFO] Session property has been set.

Flink SQL> INSERT INTO MyTableSink SELECT * FROM MyTableSource;
[INFO] Submitting SQL update statement to the cluster...
[INFO] Execute statement in sync mode. Please wait for the execution finish...
[INFO] Complete execution of the SQL update statement.
```

如果想终止作业，只需键入 CTRL-C 来取消执行。

7.3.5 构建批流一体数仓完整过程

至此，我们已经掌握了如何通过 Flink SQL Client 通过 HiveCatalog 来连接和访问 Hive。接下来，我们重构上一节的项目，通过 Flink SQL Client 集成 Hive 来实现纯 SQL 无代码构建批流一体数据仓库。实现架构如下图所示：



因为这里 Flink SQL Client 要访问 Kafka 源表，所以需要将 Flink Kafka 相关的连接器也添加到 Flink 安装的 lib 目录下。包括以下依赖 jar 包：

□

接下来，请按以下步骤执行。

步骤一 环境准备

首先准备好运行环境，包括 HADOOP 集群、Hive MetaStore、Kafka 集群、Flink on YARN session。

步骤二 创建 Flink SQL 执行的脚本文件

步骤三 提交 Flink 作业 1

步骤四 提交 Flink 作业 2

步骤五 执行数据源数据生成脚本

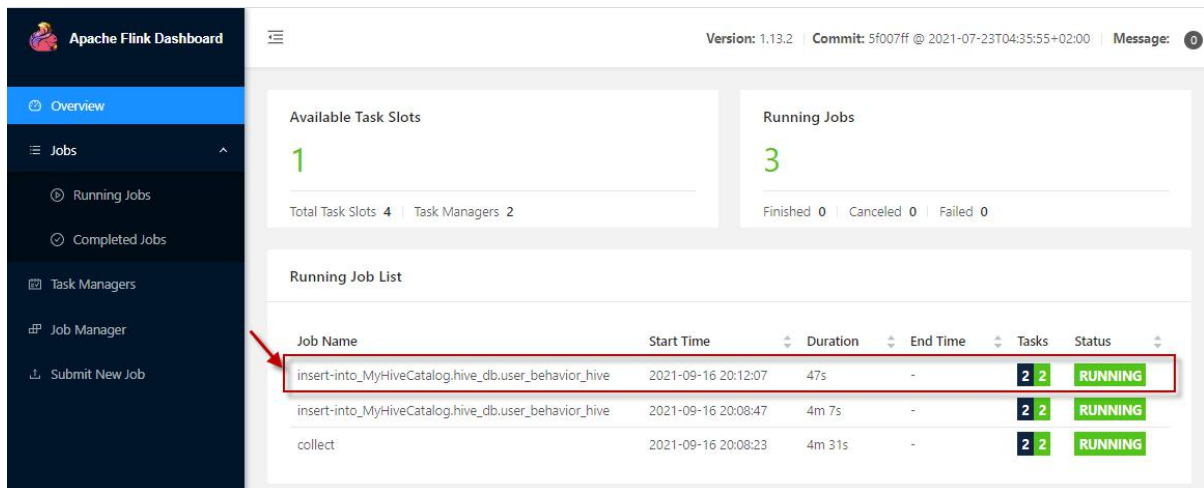
步骤六 观察作业 1 统计结果

查看作业 1 的执行窗口，可以看到类似下面这样的输出内容：

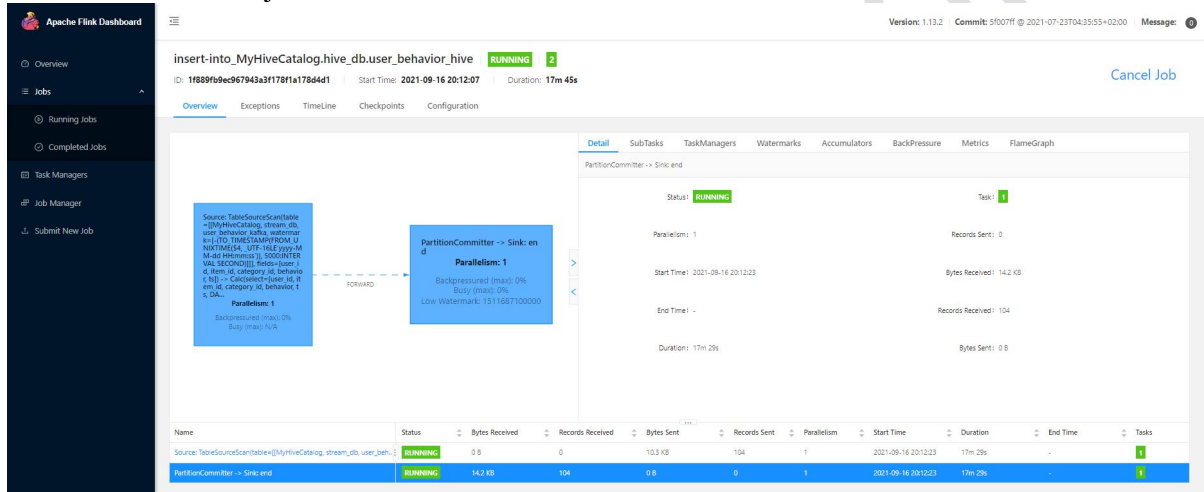
op	behavior	behavior_cnt
+I	buy	1
-U	buy	1
+U	buy	2
-U	buy	2
+U	buy	3
-U	buy	3
+U	buy	4
-U	buy	4
+U	buy	5
-U	buy	5
+U	buy	6
-U	buy	6
+U	buy	7
-U	buy	7
+U	buy	8
-U	buy	8
+U	buy	9
-U	buy	9
+U	buy	10

步骤七 观察作业 2 执行过程

打开浏览器，访问 Flink Web UI 界面，可以观察到 Flink 作业的执行情况。如下图所示：



点击图中红框标示的 job，查看作业执行详情，如下图所示：



（可选地）使用 Hive CLI 进行批量查询

当然，也可以使用 Hive CLI 进行批量查询。例如：

```
hive> select behavior,count(*) as cnt
> from user_behavior_hive
> group by behavior;
```

可以得到类似下面这样的统计结果：

```
buy      115
cart     300
fav      172
pv       5004
```

（可选地）使用 Flink SQL Client 进行批量查询

或者，使用 Flink SQL Client 连接 Hive 进行批量查询。

这时，我们创建另一个 SQL 脚本文件 `analysis_hive_batch.sql`，用于将 Hive 的 `user_behavior_hive` 表作为源表，以批处理的方式对其进行聚合统计。

`analysis_hive_batch.sql`：

```
set table.dynamic-table-options.enabled=true;
```

```
set execution.runtime-mode=batch;
```

```
SELECT behavior,count(1) AS behavior_cnt
FROM hive_db.user_behavior_hive
GROUP BY behavior;
```

注意，因为 checkpoint 不支持 batch 批处理，所以需要先在 Flink 配置文件 flink-conf.yaml 中注释掉检查点配置，如下图所示：

```
# #####
# 配置Java运行时
env.java.home: /opt/java/jdk1.8.0_281

#execution.checkpointing.interval: 10s
#execution.checkpointing.tolerable-failed-checkpoints: 10
```

执行批处理时，需要将 checkpoint 注释掉

然后，通过 Flink SQL Client 连接 Hive 执行批量聚合统计，代码如下：

```
$.bin/sql-client.sh -i conf/init.sql -f ~/data/flink/sql/analysis_hive_batch.sql
```

执行过程如下图所示：

```
[hduser@xueai8 ~]$ cd bigdata/flink-1.13.2/
[hduser@xueai8 flink-1.13.2]$ ./bin/sql-client.sh -i conf/init.sql -f ~/data/flink/sql/analysis_hive_batch.sql
2021-09-17 10:33:22,050 INFO org.apache.flink.yarn.cli.FlinkYarnSessionCli [] - Found Yarn property
2021-09-17 10:33:22,050 INFO org.apache.flink.yarn.cli.FlinkYarnSessionCli [] - Found Yarn property
No default environment specified.
Searching for '/home/hduser/bigdata/flink-1.13.2/conf/sql-client-defaults.yaml'...not found.
Successfully initialized from sql script: file:/home/hduser/bigdata/flink-1.13.2/conf/init.sql
[INFO] Executing SQL from file.
Flink SQL> set table.dynamic-table-options.enabled=true;
[INFO] Session property has been set.
Flink SQL>
set execution.runtime-mode=batch;
[INFO] Session property has been set.
Flink SQL>
SELECT behavior,count(1) AS behavior_cnt
FROM hive_db.user_behavior_hive
GROUP BY behavior;
2021-09-17 10:33:27,616 INFO org.apache.hadoop.mapred.FileInputFormat [] - Total input files t
2021-09-17 10:33:27,639 INFO org.apache.hadoop.mapred.FileInputFormat [] - Total input files t
2021-09-17 10:33:27,655 INFO org.apache.hadoop.mapred.FileInputFormat [] - Total input files t
2021-09-17 10:33:27,675 INFO org.apache.hadoop.mapred.FileInputFormat [] - Total input files t
2021-09-17 10:33:28,219 WARN org.apache.flink.yarn.configuration.YarnLogConfigUtil [] - The configuration d
nf') already contains a LOG4J config file.If you want to use logback, then please delete or rename the log configur
2021-09-17 10:33:28,254 INFO org.apache.hadoop.yarn.client.RMPProxy [] - Connecting to Resou
2021-09-17 10:33:28,366 INFO org.apache.flink.yarn.YarnClusterDescriptor [] - No path for the fl
apache.flink.yarn.YarnClusterDescriptor to locate the jar
2021-09-17 10:33:28,395 INFO org.apache.flink.yarn.YarnClusterDescriptor [] - Found Web Interface
43658815_0003'.
+-----+
| behavior | behavior_cnt |
+-----+
| pv | 5004 |
| cart | 300 |
| buy | 115 |
| fav | 172 |
+-----+
4 rows in set
```

批量查询结果

第 8 章 基于 Flink 和数据湖构建实时数仓

这一章，我们将了解下一代大数据技术的变革技术-数据湖技术，并基于 Flink 流计算和 Iceberg 数据湖框架来构建实时数仓。

8.1 现代数据湖概述

随着数据量增长到前所未有的新水平，处理这种增长的新工具和技术也出现了。其中一个演变的领域是数据湖。传统的数据仓库工具用于从数据驱动业务智能。业界随后认识到，数据仓库通过执行写模式限制了智能的潜力。显然，在收集数据时不能考虑所收集数据集的所有维度。这意味着，强制执行模式或删除很少的条目（因为它们看起来似乎是毫无用处），从长远来看可能会损害业务智能。此外，数据仓库技术无法跟上数据增长的步伐。由于数据仓库一般基于数据库和结构化数据格式，它被证明不足以应对当前数据驱动领域所面临的挑战。

这导致了数据湖的出现，这些数据湖针对非结构化和半结构化数据进行了优化，可以轻松扩展到 PB 级规模，并允许更好地集成各种工具，以帮助企业最大限度地利用其数据。

8.1.1 什么是数据湖

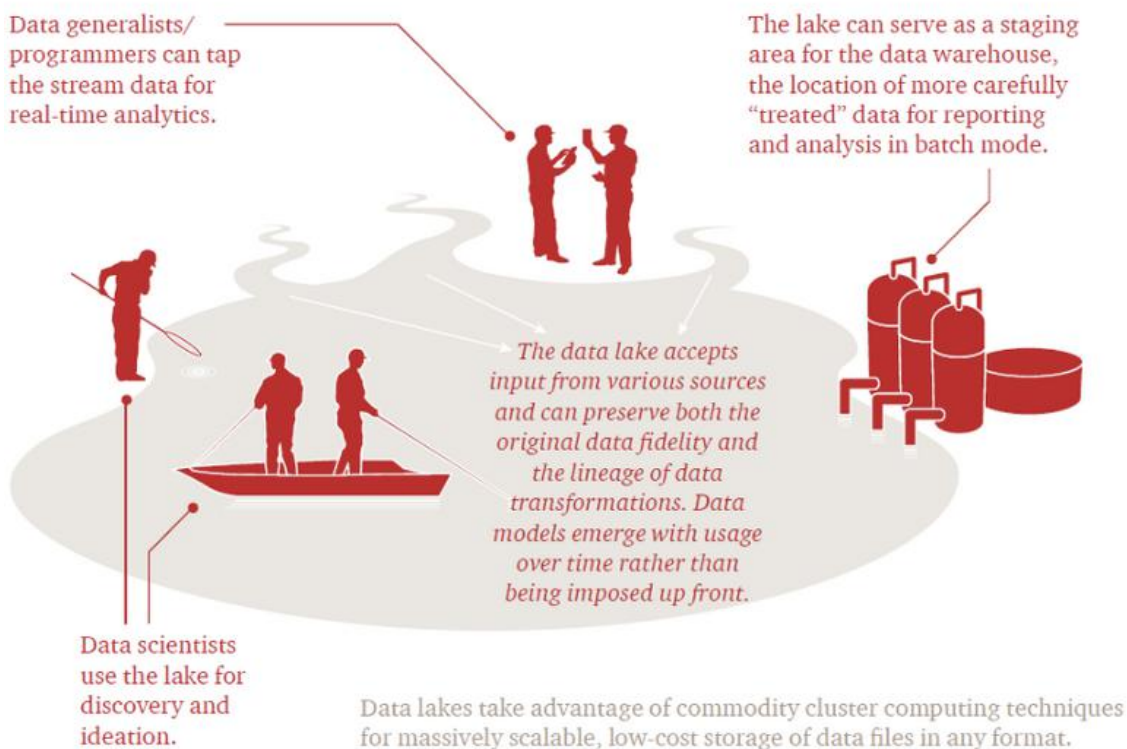
数据湖的概念是什么？一般来说，我们将企业产生的数据维护在一个平台上，我们称之为“数据湖”。通常在不同的上下文环境中术语“数据湖”具有不同含义，但在所有数据湖定义中有几个重要属性是一致的：

- ☐ 支持非结构化和半结构化数据。
- ☐ 可扩展到 PB 级或更高。
- ☐ 使用类 SQL 的接口与存储的数据交互。
- ☐ 能够尽可能无缝地连接各种分析工具。
- ☐ 最后，现代数据湖通常是分离存储和分析工具的组合。

请看下面的图片，它解释了什么是数据湖。图中这个湖的数据源是多种多样的，有些可能是结构化数据，有些可能是非结构化数据，有些甚至可能是二进制数据。有一波人站在湖口，用设备检测水质，对应数据湖上的流处理操作；有多台泵从数据湖抽水，对应数据湖的批处理操作；也有一群人在船头或岸上钓鱼，对应的是数据科学家从中提取数据价值（数据挖掘）。

What is a data lake?

A repository for large quantities and varieties of data, both structured and unstructured.



综上所述，数据湖有四个主要特征。

- ❑ 第一个特征是原始数据的存储，原始数据来源丰富；
- ❑ 第二个特征是支持多种计算模型；
- ❑ 第三个特征是具有完善的数据管理能力。能够访问多个数据源，实现不同数据之间的连接，支持模式管理和权限管理；
- ❑ 第四个特征是灵活的底层存储。一般采用 DS3、OSS 和 HDFS 作为廉价的分布式文件系统，采用特定的文件格式和缓存来满足相应场景的数据分析需求。

存储原始数据	多种计算模型	完善的数据管理	灵活的底层存储
- 结构化数据 - 半结构化数据 - 非结构化数据 - 二进制数据(图片等)	- 批处理 - 流计算 - 交互式分析 - 机器学习	- 多种数据源接入 - 数据连接 - Schema管理 - 权限管理	- S3/OSS/HDFS - Parquet/Avro/Orc - 数据缓存加速

过去几年见证了 Hadoop 作为大数据平台的崛起和随后的衰落。最初，HDFS 作为存储层，Hive 作为分析层。Hadoop 存储能够达到 100TB，可以使用类 SQL 方式对半结构化数据进行查询，并且在它的时间内足够快。

后来，数据量增长到新的规模，企业的需求变得更加雄心勃勃，例如，用户现在期望更快的查询时间、更好的可扩展性、易于管理等等。这时 Hive 和 HDFS 开始为新的更好的技术平台让路。

为了解决 Hadoop 的复杂性和扩展挑战，业界现在正在转向一种**分解的架构**，使用 REST API 将存储层和分析层松散地耦合在一起。这使得每一层都更加独立（在扩展和管理方面），并可以为每个工作使用更完美的工具。例如，在这个分解模型中，用户可以选择使用 Spark 处理批量工作负载进行分析，而 Presto 处理 SQL 繁重工作负载，Spark 和 Presto 都使用相同的后端存储平台。

这种方法现在正迅速成为标准。常用的存储平台是**对象存储平台**，如 AWS S3、Azure Blob Storage、GCS、Ceph、MinIO 等。而分析平台则不同，从简单的基于 Python 和 R 的笔记本到 Tensorflow 到 Spark、Presto 到 Splunk、Vertica 等。

这种新的分解模式，即为每项工作使用正确的工具，通常在可扩展性和易于管理方面表现得更好。这种方法在架构上也非常适合。但是，仍然存在数据一致性和管理方面的挑战需要解决。

1) 文件还是表？

分解模型是指存储系统将数据视为对象或文件的集合。但是最终用户对数据的物理安排不感兴趣，他们希望看到数据的更有逻辑性的视图。RDBMS 数据库在实现这种抽象方面做得很好。随着大数据平台成为未来的数据平台，现在人们期望这些系统以一种用户友好的方式行事，即不强制用户了解任何关于物理存储的信息。

2) SQL 接口

如上所述，用户不再愿意考虑底层平台的低效率。例如，数据湖现在也被期望与 ACID（Atomicity、Consistency、Isolation、Durability）兼容，这样最终用户就不会有额外的开销来确保与数据相关的保证。

3) 变更管理

在这种规模下管理数据的另一个非常重要的方面是能够回滚、查看何时发生了什么更改并检查特定的细节。目前，使用对象存储的版本管理可以实现这一点，但正如我们前面看到的，这是在较低的物理细节层，在较高的逻辑层可能没有用处。现在，用户期望在逻辑层看到版本控制。

业界趋势

构建统一、高效的数据存储以满足不同数据处理场景的需求已成为趋势。

- ☐ ETL 作业和 OLAP 分析—需要高性能的结构化存储，分布式能力。
- ☐ 机器学习训练和推理—需要海量的非结构存储，容器挂载能力。

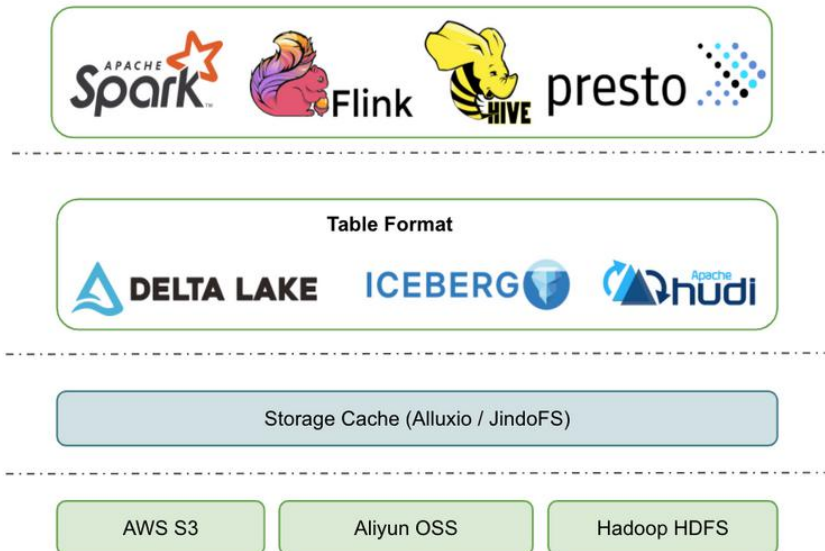
通用数仓（Hive、Spark）在向数据湖分析泛化，而数仓则向高性能架构演进。

现代数据湖的能力要求

- ☐ 支持流批计算
- ☐ Data Mutation
- ☐ 支持事务
- ☐ 计算引擎抽象
- ☐ 存储引擎抽象
- ☐ 数据质量
- ☐ 元数据支持扩展

8.1.2 数据湖架构

那么，开源数据湖的架构是什么样的呢？请看下面的架构图：



这个数据湖架构主要分为四个层次：

- ❑ 底部是分布式文件系统。云上的用户将更多地使用 S3、OSS 等对象存储。非云用户通常使用自己的 HDFS。
- ❑ 第二层是数据加速层。数据湖架构是一个完全分离存储和计算的架构。如果所有数据访问都是远程读取文件系统上的数据，性能和成本将非常高。如果我们能将一些频繁访问的热点数据缓存到本地计算节点中，那么实现冷热分离就很自然了。一方面可以获得良好的本地读取性能，另一方面可以节省远程访问的带宽。在这一层，通常选择云上开源的 Alluxio 或者 JindoFS。
- ❑ 第三层是表格式层，主要是将一批数据文件封装成具有业务意义的表，并提供如 ACID、快照、模式和分区等表级语义。表格式通常与 Delta、Iceberg 和 Hudi 等开源项目相对应。
- ❑ 顶层是不同计算场景的计算引擎。开源计算引擎一般包括 Spark、Flink、Hive、Presto、Hive MR 等。这些计算引擎可以同时访问同一数据湖的表。

在上面的架构图中，第三层表格式层是因可用性挑战和数据一致性要求而导致的一个全新的软件项目类别。这些项目位于存储平台和分析平台之间，并以本地方式处理对象存储平台，同时向最终用户提供强大的 ACID 保证。让我们从更高的层次来看看这些项目，并对它们进行比较。

8.1.3 开源数据湖框架

目前有三大开源数据湖框架，分别是 Delta Lake、Apache Iceberg 和 Hudi。另外，Apache Hive 也正在努力向数据湖演化。

Delta Lake

Delta Lake 是一个开源平台，它为 Apache Spark 带来了 ACID 事务。Delta Lake 由 Databricks 开发，它运行在现有的存储平台（S3, HDFS, Azure）上，并完全兼容 Apache Spark API。特别提供了以下特性：

- ❑ Spark 上的 ACID 事务。可序列化的隔离级别确保读取器不会看到不一致的数据。
- ❑ 可扩展元数据处理：利用 Spark 的分布式处理能力，轻松处理具有数十亿文件的 PB 级表的所有元数据。
- ❑ 流和批统一：Delta Lake 中的表既是一个批表，又是一个流源（source）和流接收器（sink）。流数据采集、批量历史回填、交互式查询都是开箱即用的。
- ❑ 模式强制：自动处理模式变化，以防止在摄入期间插入坏记录。
- ❑ 时间旅行：数据版本控制支持回滚、完整的历史审计跟踪和可重现的机器学习实验。
- ❑ upsert 和 delete：支持合并、更新和删除操作，以支持复杂的用例，如更改数据捕获、缓慢更改维度(SCD)操作、流式 Upsert 等等。

Apache Iceberg

Apache Iceberg 是一种用于巨大分析数据集的开放表格式。Iceberg 向 Presto 和 Spark 添加了使用高性能格式的表，其工作方式与 SQL 表类似。Iceberg 的重点是避免不愉快的意外，帮助模式演化和避免无意的数据删除。用户不需要了解分区来获得快速查询。

- ❑ 模式演进支持添加、删除、更新或重命名，并且没有副作用。
- ❑ 隐藏分区防止用户错误导致不正确的结果或极慢的查询。
- ❑ 分区布局演变可以随着数据量或查询模式的改变而更新表的布局。
- ❑ 时间旅行可以实现使用完全相同的表快照的可重复查询，或者让用户方便地检查更改。
- ❑ 版本回滚允许用户通过将表重置为良好状态来快速纠正问题。

Apache Hive

Apache Hive 数据仓库软件已经有一段时间了。随着新的挑战的到来，Hive 现在正在努力解决一致性和可用性问题。它便于使用 SQL 读取、写入和管理分布在分布式存储中的大型数据集。结构可以投影到已经存储的数据上。Hive 中添加了带有 ACID 语义的事务，以解决以下问题：

- ❑ 流数据摄取：使用工具如 Apache Flume、Apache Storm 或 Apache Kafka 等将数据流入到 Hadoop 集群。这些工具可以以每秒数百行或更多的速度写入数据，而 Hive 只能每 15 分钟到 1 小时增加一个分区。更频繁地添加分区会很快导致表中出现大量分区。这些工具可以将数据流到现有的分区中，但这会导致读取器脏读（也就是说，他们会在开始查询之后看到写入的数据），并在其目录中留下许多小文件，这会给 NameNode 带来压力。Hive 现在支持这种用法，即允许读取器获得一致的数据视图，避免过多的文件。
- ❑ 变化缓慢的维度（SCD）：在典型的星型模式数据仓库中，维度表会随着时间缓慢变化。例如，零售商将开设新的商店，这些商店需要添加到 stores 表中，或者现有的商店可能会更改其面积或其他一些可跟踪的特征。这些更改将导致插入单个记录或更新记录(取决于所选择的策略)。从 0.14 开始，Hive 能够支持这个功能。
- ❑ 数据重述：有时发现收集的数据不正确，需要更正。从 Hive 0.14 开始，这些用例可以通过 INSERT、UPDATE 和 DELETE 来支持。
- ❑ 使用 SQL MERGE 语句进行批量更新。

比较

以下是我们对上述工具的较高层次上的比较：

Platform	Analytics Tools Supported	File Format	Rollbacks	Compaction & Cleanup	Time Travel
Delta Lake	Apache Spark	Parquet	Yes	Yes, manual	Yes
Apache Iceberg	Apache Spark, Presto	Parquet, ORC	Yes	No	Yes
Apache Hive	Hive, Apache Spark	ORC	No	Manual & Automatic	No

Apach Hudi

Apache Hudi（简称 Hudi）是 Uber 创建的一个数据湖框架。Hudi 于 2019 年 1 月加入 Apache 孵化器进行孵化，并于 2020 年 5 月晋升为 Apache 顶级项目。它是最流行的数据湖框架之一。

Hudi 支持以下特征：

- ☐ Upsert 支持快速、可插入的索引。
- ☐ 支持回滚的原子发布数据。
- ☐ 写入器和查询之间的快照隔离。
- ☐ 保存点数据恢复。
- ☐ 使用统计管理文件大小，布局。
- ☐ 异步压缩行和列数据。
- ☐ 跟踪沿袭（lineage）的时间线元数据。
- ☐ 利用聚类优化数据湖布局。

8.2 基于 Flink + Iceberg 构建企业数据湖

随着大数据存储和处理需求的多样化，如何构建一个统一的数据湖存储，并在其上进行多种形式的数据分析成了企业构建大数据生态的一个重要方向。Netflix 发起的 Apache Iceberg 项目具备 ACID 能力的表格式中间件成为了大数据、数据湖领域炙手可热的方向。

8.2.1 Apache Iceberg 的优势

Apache Iceberg 作为一个数据湖解决方案，它支持 ACID 事务、修改和删除（基于 Spark 2.4.5+ 以上）、独立于计算引擎、支持表结构和分区方式动态变更等特性，同时，为了支持流式数据的写入，引入 Flink 作为流式处理框架，并将 Iceberg 作为 Flink sink 的终表解决方案。

Apache Iceberg 优势总结如下：

- ☐ 写入：支持事务，写入即可见；并提供 upset/merge into 的能力；
- ☐ 读取：支持以流的方式读取增量数据：Flink Table Source 以及 Spark Struct streaming 都支持；不惧怕 Schema 的变更；
- ☐ 计算：通过抽象不绑定任何引擎。提供原生的 Java Native API，生态上来说，目前支持 Spark、Flink、Presto、Hive；
- ☐ 存储：对底层存储进行了抽象，不绑定于任何底层存储；支持隐藏分区和分区进化，方便业务进行数据分区策略；支持 Parquet, ORC, Avro 等格式来兼容行存储和列存储。

当然了，Iceberg 并不支持行级 update，仅支持 insert into/overwrite，Iceberg 团队在未来将致力于解

决这个问题。

Apache Iceberg 同时支持 Apache Flink 的 DataStream API 和 Table API，以将记录写入 Iceberg 表。目前，只集成了 Iceberg 和 Apache Flink 1.11.x。

Feature support	Flink 1.11.0 Notes
SQL create catalog	✓
SQL create database	✓
SQL create table	✓
SQL create table like	✓
SQL alter table	✓ Only support altering table properties, Columns/PartitionKey changes are not supported now
SQL drop_table	✓
SQL select	✓ Support both streaming and batch mode
SQL insert into	✓ Support both streaming and batch mode
SQL insert overwrite	✓
DataStream read	✓
DataStream append	✓
DataStream overwrite	✓
Metadata tables	Support Java API but does not support Flink SQL
Rewrite files action	✓

目前已知在用的 Iceberg 的大厂：

- ❑ 国外：Netflix、Apple、Linkined、Adobe、Dremio
- ❑ 国内：腾讯、网易、阿里云

8.2.2 Apache Iceberg 经典业务场景

那么，Flink 可以与数据湖结合哪些经典应用场景呢？在这里，当我们讨论业务场景时，我们默认选择 Apache Iceberg 作为数据湖模型。

场景一 构建实时数据管道

首先，Flink + Iceberg 最经典的场景是构建一个实时数据管道，如下图所示。



业务端产生的大量日志数据被导入到消息队列中，如 Kafka。在使用 Flink 流计算引擎执行 ETL 之后，它被导入到 Apache Iceberg 原始表中。一些业务场景需要直接运行分析作业来分析原始表的数据，而另一些场景则需要进一步净化数据。然后，我们可以启动一个新的 Flink 作业来使用来自 Apache Iceberg 表的增量数据，并在处理后将其写入经过处理的 Iceberg 表。此时，可能有业务需要进一步聚合数据，因此我们继续在 Iceberg 表上启动增量 Flink 作业，并将聚合的数据结果写入聚合表。

通过 Flink+Iceberg 构建的实时数据管道，可以借助 Flink 来实现数据的精确一次性（exactly-once）语义地入湖和出湖，并且新定稿的数据可以 checkpoint 周期内可见。

当然，这个场景也可以通过 Flink + Hive 实现。但是在 Flink + Hive 实现方案中，写入 Hive 的数据

更多的是用于数据仓库的数据分析，而不是增量拉取。一般情况下，Hive 按分区增量写时间大于 15 分钟。长时间高频率的 Flink 写入会导致分区膨胀。

而 Iceberg 允许 1 分钟甚至 30 秒的增量写入，这可以极大地提高端到端数据的实时性能。上层的分析作业可以看到更新的数据，而下游的增量作业可以读取更新的数据。

场景二 实时分析 RDBMS 的变量日志

第二个经典场景是，可以使用 Flink + Iceberg 来分析来自关系型数据库(如 mysql)的 binlog。



一方面，Apache Flink 本身支持 CDC 数据解析。通过 Flink CDC 连接器拉出 binlog 数据后，自动转换为 insert、delete 和 update，Flink 运行时可以识别这些消息，供用户进行进一步的实时计算。

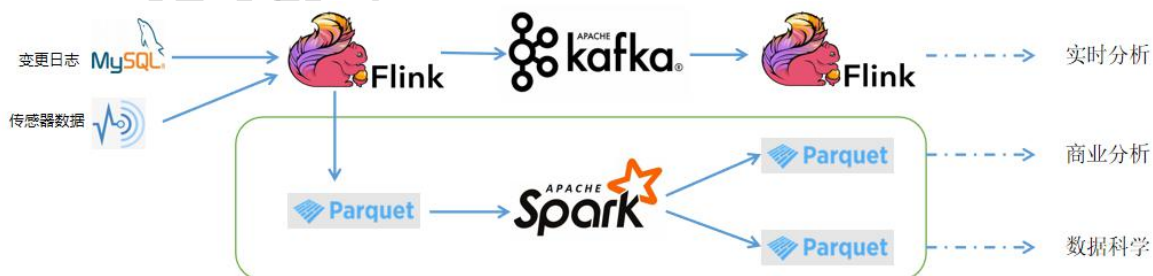
另一方面，Apache Iceberg 完美地实现了相等删除的功能，即用户定义要删除的记录，直接写入 Apache Iceberg 表中，删除相应的行，实现了数据湖的流删除。在未来的 Iceberg 版本中，用户将不需要设计任何额外的业务领域，只需要编写几行代码就可以将 binlog 流传输到 Apache Iceberg。

此外，当 CDC 数据成功输入到 Iceberg 之后，我们还可以通过常见的计算引擎，如 Presto、Spark、Hive 等，实时读取 Iceberg 表中的最新数据。

场景三 近实时场景的流批统一

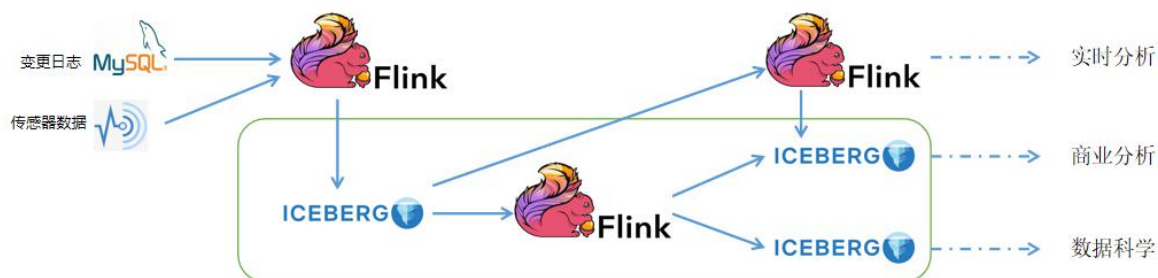
第三个经典场景是近实时场景的流批统一。

在常用的 lambda 架构中，有实时处理模块和离线处理模块。实时模块一般由 Flink、Kafka 和 HBase 构建，离线模块一般由 Parquet 和 Spark 等组件构建。如下图所示：



这种架构涉及的计算组件和存储组件很多，系统维护和业务开发成本非常高。

在实际应用中，有很多场景的实时性要求不是很苛刻，例如，可以放宽到分钟级的水平。这种场景称为近实时场景。我们可以使用 Flink + Iceberg 来优化整个架构，如下图所示。



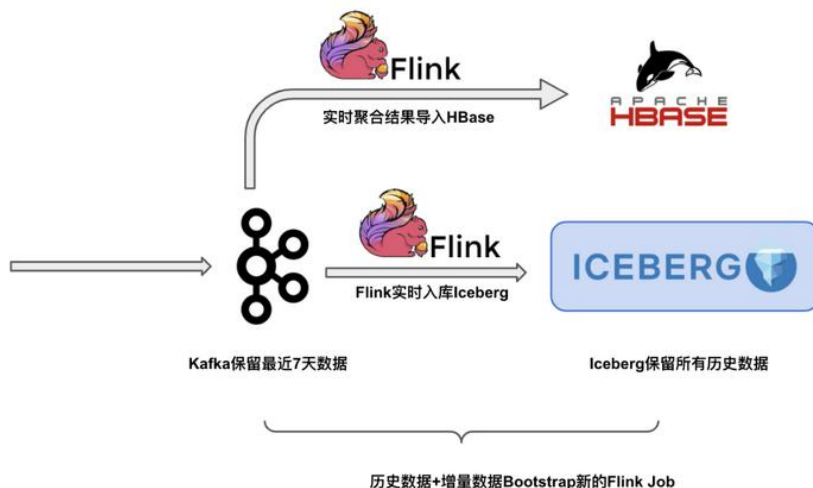
在这个优化后的架构中，实时数据通过 Flink 写入 Iceberg 表。近实时链路仍然可以通过 Flink 计算增量数据。离线链路还可以通过 Flink 批计算读取快照进行全局分析，并得到相应的分析结果，供用户在不同场景下进行读取和分析。经过这样的改进，我们将计算引擎统一到 Flink 中，将存储组件统一到 Iceberg 中，大大降低了整个系统的维护和开发成本。

场景四 使用 Iceberg 全量数据和 Kafka 增量数据引导新的 Flink 作业

第四个场景使用 Apache Iceberg 全量数据和 Kafka 增量数据引导新的 Flink 作业。

例如，我们现有的流作业是在线运行的。突然有一天，一个客户提出他们遇到了一个新的计算场景，需要设计一个新的 Flink 作业。他们浏览了去年的历史数据，并接收了正在生成的 Kafka 增量数据。那么这个时候我们应该怎么做呢？

我们仍然可以使用通用的 lambda 架构。离线链路通过 Kafka > Flink > Iceberg 同步写入数据湖。因为 Kafka 的成本较高，所以可以保留最近 7 天的数据。Iceberg 存储成本低，可以存储全部历史数据（根据检查点划分多个数据间隔）。当启动一个新的 Flink 作业时，只需要拉出 Iceberg 中的数据，然后在运行后顺利地接收 Kafka 的数据。架构如下图所示：



场景五 数据订正

第五个场景和第四个场景有点相似。还是在 lambda 架构中，实时处理链路由于事件丢失或到达顺序的问题可能导致流计算的结果不一定完全准确，此时一般需要充分的历史数据来纠正实时计算的结果。而 Apache Iceberg 可以很好地发挥这个作用，通过它可以以高性价比管理历史数据。



8.2.3 应用 Apache Iceberg 的准备工作

在使用 Flink 集成 Iceberg 之前，有几项准备工作要做。

Flink 集成 Iceberg

首先要下载 Iceberg。下载地址：<https://iceberg.apache.org/releases/#downloads>

Downloads

The latest version of Iceberg is 0.12.0.

- [0.12.0 source tar.gz - signature - sha512](#)
- [0.12.0 Spark 3.0 runtime Jar](#)
- [0.12.0 Spark 2.4 runtime Jar](#)
- [0.12.0 Flink runtime Jar](#)
- [0.12.0 Hive runtime Jar](#)

Apache Iceberg 使用嵌入式程序的方式工作，因此只需要将下载的 iceberg-flink-runtime-0.12.0.jar 包拷贝到 Flink 安装的 lib 目录下即可在 Flink 中使用 Iceberg，不需要特定的安装。

/home/hduser/bigdata/flink-1.13.2/lib/						
Name	Size (KB)	Last modified	Owner	Group	Access	
..						
antlr-runtime-3.5.2.jar	163	2021-09-14...	hduser	hduser	-rw-rw-r--	
commons-cli-1.4.jar	52	2021-08-09...	hduser	hduser	-rw-rw-r--	
flink-connector-base-1.13.2.jar	44	2021-09-16...	hduser	hduser	-rw-rw-r--	
flink-connector-hive_2.12-1.13.2.jar	7 604	2021-09-13...	hduser	hduser	-rw-rw-r--	
flink-connector-kafka_2.12-1.13.2.jar	347	2021-09-16...	hduser	hduser	-rw-rw-r--	
flink-csv-1.13.2.jar	90	2021-07-23...	hduser	hduser	-rw-r--r--	
flink-dist_2.12-1.13.2.jar	103 645	2021-07-23...	hduser	hduser	-rw-r--r--	
flink-json-1.13.2.jar	144	2021-07-23...	hduser	hduser	-rw-r--r--	
flink-shaded-hadoop-3-uber-3.1.1.7.2.9.0-173-9.0.jar	58 207	2021-08-09...	hduser	hduser	-rw-rw-r--	
flink-shaded-zookeeper-3.4.14.jar	7 529	2021-04-10...	hduser	hduser	-rw-r--r--	
flink-sql-connector-hive-3.1.2_2.12-1.13.2.jar	47 700	2021-09-14...	hduser	hduser	-rw-rw-r--	
flink-sql-connector-kafka_2.12-1.13.2.jar	3 584	2021-09-16...	hduser	hduser	-rw-rw-r--	
flink-table-blink_2.12-1.13.2.jar	37 644	2021-07-23...	hduser	hduser	-rw-r--r--	
flink-table_2.12-1.13.2.jar	34 198	2021-07-23...	hduser	hduser	-rw-r--r--	
hive-exec-3.1.2.jar	39 671	2019-08-23...	hduser	hduser	-rw-r--r--	
iceberg-flink-runtime-0.12.0.jar	22 720	2021-09-17...	hduser	hduser	-rw-rw-r--	
kafka-clients-2.4.1.jar	3 193	2021-07-09...	hduser	hduser	-rw-rw-r--	
log4j-1.2-api-2.12.1.jar	65	2020-03-25...	hduser	hduser	-rw-r--r--	
log4j-api-2.12.1.jar	270	2020-03-25...	hduser	hduser	-rw-r--r--	
log4j-core-2.12.1.jar	1 635	2020-03-25...	hduser	hduser	-rw-r--r--	
log4j-slf4j-impl-2.12.1.jar	22	2020-03-25...	hduser	hduser	-rw-r--r--	
mariadb-java-client-2.1.2.jar	482	2021-09-16...	hduser	hduser	-rw-rw-r--	

8.2.4 创建和使用 catalog

Apache Iceberg 支持三种 catalog，分别是 hive catalog、hadoop catalog 和自定义 catalog。

Hive catalog

使用 hive catalog，可以将表结构持久存储到 Hive 的 MetaStore 中。下面是创建一个名为 hive_catalog 的 Iceberg hive catalog 的示例代码：

```
CREATE CATALOG hive_catalog WITH (  
  'type'='iceberg',  
  'catalog-type'='hive',  
  'uri'='thrift://xueai8:9083',  
  'clients'='5',  
  'property-version'='1',  
  'warehouse'='hdfs://xueai8:8020/user/hive/warehouse'  
);
```

各个参数解释如下：

- ☐ type: 对于 iceberg 表格式，只使用 iceberg。（必选）
- ☐ catalog-type: Iceberg 当前支持 hive 或 hadoop catalog 类型。（必选）
- ☐ uri: Hive metastore 的 thrift URI。（必选）
- ☐ clients: Hive metastore 客户端池大小，默认值为 2。（可选）
- ☐ property-version: 版本号，用于描述属性版本。如果属性格式改变，此属性可用于向后兼容。当前的属性版本是 1。（可选）
- ☐ warehouse: Hive warehouse 位置，如果既没有设置 hive-conf-dir 来指定包含 hive-site.xml 配置文件的位置，也没有将正确的 hive-site.xml 添加到 classpath，则用户应该指定此路径。
- ☐ hive-conf-dir: 包含 hive-site.xml 配置文件的目录路径，该配置文件将用于提供定制的 Hive 配置值。当创建 Iceberg catalog 时如果在 hive-conf-dir 和 warehouse 都设置了，那么来自 <hive-conf-dir>/hive-site.xml（或 classpath 的 hive 配置）的 hive.metastore.warehouse.dir 值将被 warehouse 的值覆盖。
- ☐ cache-enabled: 是否启用 catalog 缓存，默认值为 true。

创建之后，可以使用“show catalogs”命令查看当前所有的 catalogs。

Hadoop catalog

Apache Iceberg 还支持 HDFS 中基于目录的 catalog，可以使用'catalog-type'='hadoop'来配置。例如：

```
CREATE CATALOG hadoop_catalog WITH (  
  'type'='iceberg',  
  'catalog-type'='hadoop',  
  'warehouse'='hdfs://xueai8:8020/user/hive/warehouse',  
  'property-version'='1'  
);
```

- ☐ warehouse: 用于存放元数据文件和数据文件的 HDFS 目录。（必选）

自定义 catalog

Flink 还支持通过指定 `catalog-impl` 属性来加载自定义的 Iceberg Catalog 实现。当设置 `catalog-impl` 时，将忽略 `catalog-type` 的值。例如：

```
CREATE CATALOG my_catalog WITH (  
  'type'='iceberg',  
  'catalog-impl'='com.my.custom.CatalogImpl',  
  'my-additional-catalog-config'='my-value'  
);
```

通过 YAML 配置创建 catalog

可以在启动 Flink SQL Client 之前，在 `conf/sql-client-defaults.yaml` 配置文件中注册 catalog。例如：

```
catalogs:  
  - name: my_catalog  
    type: iceberg  
    catalog-type: hadoop  
    warehouse: hdfs://xueai8:8020/warehouse/path
```

8.2.5 Iceberg DDL 命令

Apache Iceberg 支持如下常用的 DDL 命令。

❑ CREATE DATABASE

默认情况下，Iceberg 将使用 Flink 中的 `default` 数据库。如果我们不想在 `default` 数据库下创建表，可以创建一个单独的数据库，例如，以下代码创建一个名为 `iceberg_db` 的数据库：

```
create database iceberg_db;  
use iceberg_db;
```

❑ CREATE TABLE

表创建命令。使用方法如下所示：

```
CREATE TABLE hive_catalog.default.sample (  
  id BIGINT COMMENT 'unique id',  
  data STRING  
);
```

创建表的 `create` 命令现在支持最常用的 Flink `create` 子句，包括：

- ❑ `PARTITION BY (column1, column2, ...)` 配置分区，Flink 还不支持隐藏分区。
- ❑ `COMMENT 'table document'` 设置一个表描述。
- ❑ `WITH ('key'='value', ...)` 设置将存储在 Apache Iceberg 表属性中的表配置。

目前不支持计算列、主键、水印定义等。

❑ PARTITIONED BY

要创建分区表，使用 `PARTITIONED BY`。例如：

```
CREATE TABLE hive_catalog.default.sample (  
  id BIGINT COMMENT 'unique id',  
  data STRING
```

```
) PARTITIONED BY (data);
```

Apache Iceberg 支持隐藏分区，但 Apache Flink 不支持通过列上的函数进行分区，所以我们现在无法在 Flink DDL 中支持隐藏分区。

❑ CREATE TABLE LIKE

要创建与另一个表具有相同模式、分区和表属性的表，可以使用 CREATE TABLE LIKE。例如：

```
CREATE TABLE hive_catalog.default.sample (  
    id BIGINT COMMENT 'unique id',  
    data STRING  
);
```

```
CREATE TABLE hive_catalog.default.sample_like LIKE hive_catalog.default.sample;
```

❑ ALTER TABLE

现在，Iceberg 只支持在 Flink 1.11 中修改表属性。（Flink 1.13.2 中待测试）

```
ALTER TABLE hive_catalog.default.sample SET ('write.format.default'='avro')
```

❑ ALTER TABLE .. RENAME TO

对表重命名。例如：

```
ALTER TABLE hive_catalog.default.sample RENAME TO hive_catalog.default.new_sample;
```

❑ DROP TABLE

删除表，执行以下命令：

```
DROP TABLE hive_catalog.default.sample;
```

8.2.6 Iceberg SQL 查询

Iceberg 现在支持 Flink 中的流和批读取。我们可以执行以下 SQL 命令，将执行类型从 “streaming” 模式切换到 “batch” 模式，反之亦然：

```
-- 以流模式为当前会话上下文执行 flink 作业  
SET execution.type = streaming
```

```
-- 以批处理模式为当前会话上下文执行 flink 作业  
SET execution.type = batch
```

Flink 批量读取

如果想通过提交 Flink 批处理作业检查 Iceberg 表中的所有行，可以执行以下句子：

```
SET execution.type = batch ;  
SELECT * FROM sample ;
```

Flink 流式读取

Iceberg 支持 Flink 流作业中从历史快照 id 开始的增量数据处理。例如：

```
-- 设置当前会话以流模式提交 flink 作业  
SET execution.type = streaming ;
```

```
-- 启用此开关，因为流式读取 SQL 将在 Flink SQL 提示选项中提供的一些的作业选项
SET table.dynamic-table-options.enabled=true;

-- 从 iceberg 当前快照读取所有记录，然后从该快照开始读取增量数据。
SELECT * FROM sample /*+ OPTIONS('streaming'='true', 'monitor-interval'='1s')*/;

-- 从 id 为“3821550127947089987”的快照开始读取所有增量数据(将排除该快照的记录)。
SELECT * FROM sample /*+ OPTIONS('streaming'='true', 'monitor-interval'='1s',
'start-snapshot-id'='3821550127947089987')*/;
```

这些是可以设置在 Flink SQL 提示选项中用于流作业的选项：

- ❑ **monitor-interval**: 连续监控新提交数据文件的时间间隔（默认值：“1s”）。
- ❑ **start-snapshot-id**: 流作业开始的快照 id。

8.2.7 Iceberg SQL 写入

现在，Iceberg 在 Flink 1.11 中支持 insert into 和 insert overwrite。（Flink 1.13.2 待测试）

❑ INSERT INTO

要使用 Flink 流作业将新数据追加到表中，使用 INSERT INTO:

```
INSERT INTO hive_catalog.default.sample VALUES (1, 'a');
INSERT INTO hive_catalog.default.sample SELECT id, data from other_kafka_table;

insert into sample values (1,'test1');
insert into sample values (2,'test2');

insert into sample_partition PARTITION(data='city') SELECT 86;
```

❑ INSERT OVERWRITE

若要用查询结果替换表中的数据，请在批作业中使用 INSERT OVERWRITE (Flink 流作业不支持 INSERT OVERWRITE)。对于 Iceberg 表来说，覆盖是原子操作。

有由 SELECT 查询产生的行的分区将被替换，例如：

```
INSERT OVERWRITE sample VALUES (1, 'a');
```

Iceberg 还支持通过 select 值覆盖给定的分区，例如：

```
INSERT OVERWRITE hive_catalog.default.sample PARTITION(data='a') SELECT 6;
```

对于 Iceberg 分区表，当所有的分区列在 partition 子句中设置了一个值，它是插入一个静态分区，否则如果部分分区列（所有分区列的前缀部分）在 partition 子句中设置了一个值，它是将查询结果写入一个动态分区。对于一个未分区的 Iceberg 表，它的数据将被 INSERT OVERWRITE 完全覆盖。

8.2.8 使用 DataStream 读取

Iceberg 现在在 Java API 中支持流或批读取。

批读取

下面的示例将从 Iceberg 表中读取所有记录，然后在 Flink 批作业中将其打印到标准输出控制台：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironment();
TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://xueai8:8020/warehouse/path");

DataStream<RowData> batch = FlinkSource.forRowData()
    .env(env)
    .tableLoader(loader)
    .streaming(false)
    .build();

// 打印所有记录到标准输出
batch.print();

// 提交并执行此批读作业
env.execute("Test Iceberg Batch Read");
```

流读取

下面的示例将读取从 snapshot-id '3821550127947089987' 开始的增量记录，并在 Flink 流作业中打印到标准输出控制台：

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironment();
TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://xueai8:8020/warehouse/path");

DataStream<RowData> stream = FlinkSource.forRowData()
    .env(env)
    .tableLoader(loader)
    .streaming(true)
    .startSnapshotId(3821550127947089987)
    .build();

// 打印所有记录到标准输出
stream.print();

// 提交并执行此流读作业
env.execute("Test Iceberg Batch Read");
```

8.2.9 使用 DataStream 写入

Iceberg 支持从不同的 DataStream 输入写入 Iceberg 表。

追加数据

支持将 DataStream<RowData> 和 DataStream<Row> 写入到 Iceberg sink 表中。

```
StreamExecutionEnvironment env = ...;
```

```
DataStream<RowData> input = ...;
```

```
Configuration hadoopConf = new Configuration();
TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://xueai8:8020/warehouse/path",hadoopConf);

FlinkSink.forRowData(input)
    .tableLoader(tableLoader)
    .build();

env.execute("Test Iceberg DataStream");
```

Iceberg API 还允许用户将泛型化的 `DataStream<T>` 写入 Iceberg 表。

覆盖数据

为了动态地覆盖现有 Iceberg 表中的数据，我们可以在 FlinkSink builder 中设置 `overwrite` 标志。

```
StreamExecutionEnvironment env = ...;

DataStream<RowData> input = ... ;
Configuration hadoopConf = new Configuration();
TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://xueai8:8020/warehouse/path",hadoopConf);

FlinkSink.forRowData(input)
    .tableLoader(tableLoader)
    .overwrite(true)
    .build();

env.execute("Test Iceberg DataStream");
```

8.2.10 重写文件操作

通过提交 Flink 批作业，Iceberg 提供了将小文件改写为大文件的 API。这个 Flink 操作的行为与 Spark 的 `rewriteDataFiles` 相同。

```
import org.apache.iceberg.flink.actions.Actions;

TableLoader tableLoader = TableLoader.fromHadoopTable("hdfs://xueai8:8020/warehouse/path");
Table table = tableLoader.loadTable();
RewriteDataFilesActionResult result = Actions.forTable(table)
    .rewriteDataFiles()
    .execute();
```

8.2.11 未来改进

在当前的 Flink Iceberg 集成工作中，有一些特性还不支持：

- ☐ 不支持创建带有隐藏分区的 iceberg 表。
- ☐ 不支持创建包含计算列的 iceberg 表。
- ☐ 不支持创建带水印的 iceberg 表。
- ☐ 不支持添加列、删除列、重命名列、更改列。

8.3 基于 Flink + Iceberg 构建准实时数仓

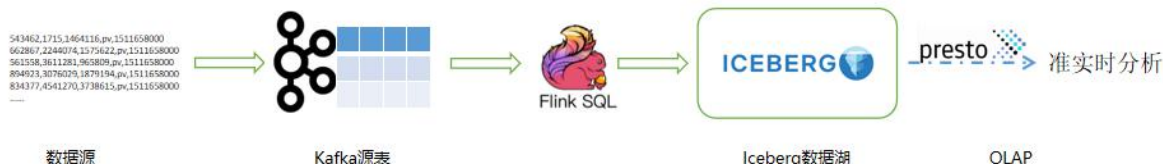
Apache Flink 是大数据领域流行的流批处理计算引擎。数据湖是云时代的一种新技术架构。这导致了基于 Iceberg、Hudi 和 Delta Lake 的解决方案的兴起。Apache Iceberg 目前支持 Flink 通过 DataStream API/Table API 将数据写入 Iceberg 表，并为 Apache Flink 1.11.x 提供集成支持。

在这一章，我们将解释并演示如何使用 Apache Flink 和 Apache Iceberg 构建实时数据仓库。

8.3.1 实时数仓构建

接下来，我们集成 Flink + Iceberg + Presto 实现基于数据湖的准实时数仓案例。整体设计架构如下图所示。我们继续使用淘宝用户行为数据集 UserBehavior_part.csv 来实时生成生成用户行为数据，使用 Kafka 作为流数据入口，使用 Flink SQL Client 执行实时入库操作，使用 Presto 执行分钟级的 OLAP 联机分析处理。

（备注：Apache Iceberg 目前支持到 Flink 1.11，所以这一章如果无法运行，请降低 Flink 版本）



因为这里 Flink SQL Client 要访问 Kafka 源表以及使用 Iceberg 表格式，所以需要将以下相关的 jar 包添加到 Flink 安装路径的 lib 目录下。

□

如下图所示：

接下来，请按以下步骤执行。

步骤一 环境准备

步骤二 启动 Flink SQL 客户端

步骤三 创建 Kafka 流表（源表）

步骤四 创建 Iceberg Hive Catalog，以及 Iceberg Sink 表。

Flink 实时读取 Kafka 的用户购买行为记录并写入到 Iceberg Sink 表中，因此，下面我们先定义一个可用的 Iceberg Catalog，然后在该 catalog 中定义数据库和接收表。继续在 Flink SQL Client 命令行，执行以下语句：

8.3.2 执行 OLAP 联机分析

接下来，我们使用 Presto 执行 OLAP 联机分析处理，对 iceberg_db.user_behavior_hive 表执行聚合操作，每 10 分钟统计一次用户各种行为计数。

Apache Presto 是一个分布式并行查询执行引擎，为低延迟和交互式查询分析进行了优化。Presto 可以轻松地运行查询，而且不需要停机时间，甚至可以从 GB 扩展到 PB 级。

Apache Presto 是由 Facebook 开发的。Facebook 是世界上领先的数据驱动和最大的数据仓库公司之一。最初 Facebook 仓库数据存储在 Hadoop 中，便于大规模计算。后来，当仓库数据增长到 PB 规模时，他们决定开发一个低延迟的新系统。在 2012 年，Facebook 团队成员设计了“Presto”，用于交互式查询分析，即使有 PB 大小的数据也能快速运行。

请按以下步骤操作。

步骤一 配置 Iceberg 连接器

步骤二 运行 Presto 服务器

在终端窗口中，执行以下命令，在前台运行 Presto 服务器：

步骤三 运行 Presto CLI 客户端

在终端窗口中，执行以下命令，通过 Presto CLI 连接 Presto 服务器：

然后在 Presto CLI 中执行如下 SQL 语句：

执行过程和结果如下图所示：

步骤四 执行聚合操作

接下来，执行聚合查询，统计各种用户购买行为。在 Presto CLI 中执行 Presto SQL 语句如下：

执行过程和执行结果如下图所示：

步骤五 退出 Presto CLI

要退出 Presto CLI，执行如下命令：

```
presto> exit;
```

步骤六 关闭 Presto 服务器

要关闭 Presto 服务器，执行如下命令：

执行完所有命令后，可以使用以下命令停止 presto 服务器：