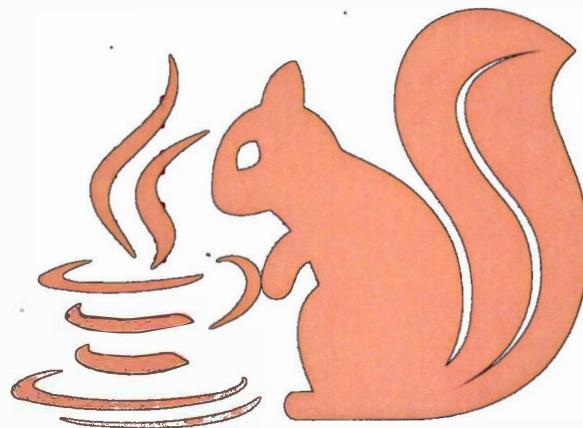


Alink权威指南

基于Flink的机器学习实例入门（Java）

杨旭 著



中国工信出版集团



电子工业出版社
http://www.ptpress.com.cn

Alink权威指南

基于Flink的机器学习实例入门（Java）

杨旭 著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

Alink 是阿里巴巴开源的机器学习算法平台，提供了丰富、高效的算法及简便的使用方式，可帮助用户快速构建业务应用。Alink 非常适合工业级的实际应用，支持在个人计算机上快速进行原型研发，支持分布式计算处理海量的数据，支持流式数据的场景，同时机器学习流程与模型可以方便地嵌入用户的应用系统或预测服务中。

本书是根据机器学习的知识点由浅入深来逐层讲述的，这样可降低阅读的门槛，让读者能对所学的内容有一个清晰的印象，并可熟练地运用到实践中。本书重点介绍算法的使用，每节结合实际的数据和典型的场景，通过 Alink 算法组件形成完整的解决方案，可帮助读者理解各类算法所擅长处理的问题，同时本书的方案还可以被推广、应用到类似的场景中。

本书适合机器学习算法的初学者及中级用户快速入门，也可供数据分析师、算法工程师等专业人员参考阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Alink 权威指南：基于 Flink 的机器学习实例入门：Java / 杨旭著. —北京：电子工业出版社，2021.10
ISBN 978-7-121-42058-0

I. ①A… II. ①杨… III. ①机器学习—指南②JAVA 语言—程序设计—指南 IV. ①TP181-62
②TP312.8-62

中国版本图书馆 CIP 数据核字（2021）第 188761 号

责任编辑：刘皎

印 刷：三河市良远印务有限公司

装 订：三河市良远印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱

邮编：100036

开 本：787×980 1/16 印张：29.5

字数：637 千字

版 次：2021 年 10 月第 1 版

印 次：2021 年 10 月第 1 次印刷

定 价：149.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，
联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

序

算法、算力、数据三个要素对于机器学习的重要性已经深入人心。前几年，深度学习算法突飞猛进，给工业界带来了非常丰富的神经网络算法，尤其是带来了感知类的模型。同时，我们也意识到，在产业互联网中，很多传统的非深度学习算法，比如线性回归、决策树、K 最近邻算法等，依然是智能化改造的必备工具。

如何更容易地将这些算法和实际的生产数据结合起来，如何让机器学习“从算法到模型再到应用”的道路变得更加普惠，是我们开发 Alink 之初就会考虑的问题。基于同样开源的流计算引擎 Flink，Alink 将众多的机器学习算法以标准组件的方式结合在一起，力图让那些对机器学习感兴趣的业务工程师可以迅速将这些算法和生产数据集结合在一起，验证效果，进行参数的调优，并最终将这些算法嵌入业务流程中。

今天我们说“一切业务数据化”，而数据体量的井喷让我们看到了不断增长的算力需求。一方面，机器学习的算法实现需要和大数据体系无缝结合，支持海量数据的分析计算；另一方面，每一位开发者和每一家企业都希望通过云上的 AI 工程体系来更容易地触达更高弹性的算力。从第一天起，Alink 就秉承了大数据与 AI 一体化、云原生的设计模式。作为一家技术企业，同时作为云服务的提供者，阿里巴巴集团希望能够通过开源和云的结合，帮助大家实现更多业务场景的数据化和智能化。

贾扬清
阿里巴巴集团副总裁
阿里巴巴开源技术委员会负责人

前　　言

Alink 是阿里巴巴开源的机器学习平台，在阿里巴巴集团内外经历了各种项目 的实际验证，非常适合工业级的机器学习应用。该机器学习平台降低了我们使用机器学习技术的门槛——其将各个算法封装成组件，即使读者不完全了解背后的理论知识，也可以仿照书中的实例，将组件连接起来解决一些实际问题，在实践中学习。

在 Alink 开源后，陆续有一些网友分享自己学习 Alink 的心得。我自己也写过一系列的文章，每次都聚焦在一个技术点，尽量讲清楚、讲透。在 Alink 的文档中，对每个算法都有专门的介绍，详细说明了参数，并给出了使用示例。但是，各算法是分散讲解的，我们无法窥其全貌。所以，我希望通过本书将分散的知识点串联起来。这样读者读完本书，就可以对 Alink 有一个整体的清晰认识，了解各个算法组件的功能与优缺点，并在面临实际问题的时候，联想到本书的相关章节，能有一个大致的解决问题的思路，知道需要哪方面的组件。

本书内容的脉络如下：

- 第 1 章为 Alink 快速上手，通过示例演示多个场景的应用，可使读者对 Alink 有一个大致的印象。随后的第 2 章介绍了 Alink 的系统概况与核心概念。
- 机器学习需要基于数据，但在实际应用中，数据的来源是五花八门的，我们如何读/写各种文件系统中的数据文件、如何读/写各种类型的数据库？第 3、4 章系统地介绍相关内容。
- 完整的机器学习流程中一定包括数据处理流程，第 5、6、7 章分别介绍了三种重要的方式：使用 SQL 语句、使用用户定义函数，以及使用 Alink 定义好的基本数据处理组件（比如各种采样操作、数据标准化、缺失值填充等）。
- 第 8 章以纸钞的真假判断为例，介绍了线性二分类模型的使用及二分类评估方法。第 9

章以判断蘑菇是否有毒为例，介绍了两种常用的非线性模型：朴素贝叶斯模型与决策树模型。

- 第 10、11 章分别以信用预测及商品购买行为预测为例，介绍了特征的转化和生成方法，介绍了常用的随机森林和 GBDT 算法。
- 第 12、13 章以鸢尾花分类及手写数字识别为例，介绍了常用的多分类算法：二分类器组合算法、Softmax 算法、多层次感知器分类器算法、K 最近邻算法。
- 第 14 章针对在线学习场景，以广告点击率预估为例，介绍了流式训练及预测流程。
- 第 15、16 章从回归的由来讲起，以身高预测及葡萄酒品质预测为例，介绍了常用的回归算法。
- 第 17、18 章以鸢尾花数据、经纬度数据、手写数字数据为例，介绍了常用的聚类算法，并介绍了流式 K-Means (K 均值) 聚类。
- 第 19、20 章介绍了重要的参数降维方法——主成分分析；介绍了超参数搜索的工具，以大幅减少模型调参所需的时间。
- 第 21、22 章介绍文本分析的常用方法以及单词向量化方法，对一些典型文本数据，包括新闻标题数据集、相声《报菜名》的内容、《三国演义》全文进行了分析。
- 第 23 章以电影评论数据集为例，构建情感分析方案，并通过调整算法模型、优化特征工程，不断改进预测效果。
- 第 24 章介绍了常用的推荐算法，并以影片推荐为例，展示了如何构建推荐系统。

本书提供了完整的源代码，读者在个人计算机中就能直接尝试、验证书中的方法和算法。书中所介绍的是业界正在使用的工具，其支持分布式计算处理海量的数据、支持流式数据的场景，同时机器学习流程及模型可以方便地嵌入用户的应用系统或预测服务中。

感谢一直支持 Alink 发展的各位同事和朋友，衷心希望 Alink 能够帮助更多的用户！感谢家人的理解和支持！

杨旭

2021 年 9 月

目 录

第 1 章 Alink 快速上手	1
1.1 Alink 是什么	1
1.2 免费下载、安装	1
1.3 Alink 的功能	2
1.3.1 丰富的算法库	2
1.3.2 多样的使用体验	3
1.3.3 与 SparkML 的对比	3
1.4 关于数据和代码	4
1.5 简单示例	5
1.5.1 数据的读/写与显示	5
1.5.2 批式训练和批式预测	7
1.5.3 流式处理和流式预测	9
1.5.4 定义 Pipeline，简化操作	10
1.5.5 嵌入预测服务系统	12
第 2 章 系统概况与核心概念	14
2.1 基本概念	14
2.2 批式任务与流式任务	15
2.3 Alink=A+link	18
2.3.1 BatchOperator 和 StreamOperator	19

2.3.2 link 方式是批式算法/流式算法的通用使用方式	20
2.3.3 link 的简化	23
2.3.4 组件的主输出与侧输出	23
2.4 Pipeline 与 PipelineModel	24
2.4.1 概念和定义	24
2.4.2 深入介绍	25
2.5 触发 Alink 任务的执行	28
2.6 模型信息显示	29
2.7 文件系统与数据库	34
2.8 Schema String	36
第 3 章 文件系统与数据文件	38
3.1 文件系统简介	38
3.1.1 本地文件系统	39
3.1.2 Hadoop 文件系统	41
3.1.3 阿里云 OSS 文件系统	43
3.2 数据文件的读入与导出	45
3.2.1 CSV 格式	47
3.2.2 TSV、LibSVM、Text 格式	53
3.2.3 AK 格式	56
第 4 章 数据库与数据表	60
4.1 简介	60
4.1.1 Catalog 的基本操作	60
4.1.2 Source 和 Sink 组件	61
4.2 Hive 示例	62
4.3 Derby 示例	65
4.4 MySQL 示例	67
第 5 章 支持 Flink SQL	70
5.1 基本操作	70
5.1.1 注册	70

5.1.2 运行	71
5.1.3 内置函数	74
5.1.4 用户定义函数	74
5.2 简化操作	75
5.2.1 单表操作	76
5.2.2 两表的连接（JOIN）操作	80
5.2.3 两表的集合操作	82
5.3 深入介绍 Table Environment	86
5.3.1 注册数据表名	87
5.3.2 撤销数据表名	88
5.3.3 扫描已注册的表	89
第 6 章 用户定义函数（UDF/UDTF）	90
6.1 用户定义标量函数（UDF）	90
6.1.1 示例数据及问题	91
6.1.2 UDF 的定义	91
6.1.3 使用 UDF 处理批式数据	92
6.1.4 使用 UDF 处理流式数据	93
6.2 用户定义表值函数（UDTF）	95
6.2.1 示例数据及问题	95
6.2.2 UDTF 的定义	96
6.2.3 使用 UDTF 处理批式数据	96
6.2.4 使用 UDTF 处理流式数据	99
第 7 章 基本数据处理	101
7.1 采样	101
7.1.1 取“前”N个数据	102
7.1.2 随机采样	102
7.1.3 加权采样	104
7.1.4 分层采样	105
7.2 数据划分	106
7.3 数值尺度变换	108

7.3.1 标准化	109
7.3.2 MinMaxScale.....	111
7.3.3 MaxAbsScale.....	112
7.4 向量的尺度变换	113
7.4.1 StandardScale、MinMaxScale、MaxAbsScale.....	113
7.4.2 正则化	115
7.5 缺失值填充	116
第 8 章 线性二分类模型.....	119
8.1 线性模型的基础知识	119
8.1.1 损失函数	119
8.1.2 经验风险与结构风险	121
8.1.3 线性模型与损失函数	122
8.1.4 逻辑回归与线性支持向量机（Linear SVM）	123
8.2 二分类评估方法	125
8.2.1 基本指标	126
8.2.2 综合指标	128
8.2.3 评估曲线	131
8.3 数据探索	136
8.3.1 基本统计	138
8.3.2 相关性	140
8.4 训练集和测试集	144
8.5 逻辑回归模型	145
8.6 线性 SVM 模型.....	147
8.7 模型评估	149
8.8 特征的多项式扩展	153
8.9 因子分解机	157
第 9 章 朴素贝叶斯模型与决策树模型.....	160
9.1 朴素贝叶斯模型	160
9.2 决策树模型	162
9.2.1 决策树的分裂指标定义	165

9.2.2 常用的决策树算法	167
9.2.3 指标计算示例	169
9.2.4 分类树与回归树	172
9.2.5 经典的决策树示例	173
9.3 数据探索	176
9.4 使用朴素贝叶斯方法	179
9.5 蘑菇分类的决策树	185
第 10 章 特征的转化	191
10.1 整体流程	195
10.1.1 特征哑元化	197
10.1.2 特征的重要性	198
10.2 减少模型特征的个数	200
10.3 离散特征转化	202
10.3.1 独热编码	202
10.3.2 特征哈希	204
第 11 章 构造新特征	207
11.1 数据探索	208
11.2 思路	210
11.2.1 用户和品牌的各种特征	211
11.2.2 二分类模型训练	212
11.3 计算训练集	213
11.3.1 原始数据划分	213
11.3.2 计算特征	214
11.3.3 计算标签	222
11.4 正负样本配比	224
11.5 决策树	226
11.6 集成学习	227
11.6.1 Bootstrap aggregating	228
11.6.2 Boosting	229
11.6.3 随机森林与 GBDT	232

11.7 使用随机森林算法	233
11.8 使用 GBDT 算法	234
第 12 章 从二分类到多分类	235
12.1 多分类模型评估方法	235
12.1.1 综合指标	237
12.1.2 关于每个标签值的二分类指标	238
12.1.3 Micro、Macro、Weighted 计算的指标	239
12.2 数据探索	241
12.3 使用朴素贝叶斯进行多分类	244
12.4 二分类器组合	246
12.5 Softmax 算法	249
12.6 多层感知器分类器	253
第 13 章 常用多分类算法	256
13.1 数据准备	256
13.1.1 读取 MNIST 数据文件	257
13.1.2 稠密向量与稀疏向量	258
13.1.3 标签值的统计信息	261
13.2 Softmax 算法	262
13.3 二分类器组合	264
13.4 多层感知器分类器	265
13.5 决策树与随机森林	267
13.6 K 最近邻算法	270
第 14 章 在线学习	273
14.1 整体流程	273
14.2 数据准备	275
14.3 特征工程	277
14.4 特征工程处理数据	279
14.5 在线训练	280
14.6 模型过滤	283

第 15 章 回归的由来	286
15.1 平均数	287
15.2 向平均数方向的回归	288
15.3 线性回归	289
第 16 章 常用回归算法	292
16.1 回归模型的评估指标	292
16.2 数据探索	294
16.3 线性回归	297
16.4 决策树与随机森林	300
16.5 GBDT 回归	301
第 17 章 常用聚类算法	303
17.1 聚类评估指标	304
17.1.1 基本评估指标	304
17.1.2 基于标签值的评估指标	306
17.2 K-Means 聚类	308
17.2.1 算法简介	308
17.2.2 K-Means 实例	310
17.3 高斯混合模型	314
17.3.1 算法介绍	314
17.3.2 GMM 实例	316
17.4 二分 K-Means 聚类	317
17.5 基于经纬度的聚类	320
第 18 章 批式与流式聚类	324
18.1 稠密向量与稀疏向量	324
18.2 使用聚类模型预测流式数据	326
18.3 流式聚类	329
第 19 章 主成分分析	331
19.1 主成分的含义	333

19.2 两种计算方式	337
19.3 在聚类方面的应用	339
19.4 在分类方面的应用	343
第 20 章 超参数搜索	347
20.1 示例一：尝试正则系数	348
20.2 示例二：搜索 GBDT 超参数	349
20.3 示例三：最佳聚类个数	350
第 21 章 文本分析	353
21.1 数据探索	353
21.2 分词	355
21.2.1 中文分词	356
21.2.2 Tokenizer 和 RegexTokenizer	359
21.3 词频统计	363
21.4 单词的区分度	365
21.5 抽取关键词	367
21.5.1 原理简介	367
21.5.2 示例	369
21.6 文本相似度	371
21.6.1 文本成对比较	372
21.6.2 最相似的 TopN	375
21.7 主题模型	387
21.7.1 LDA 模型	388
21.7.2 新闻的主题模型	390
21.7.3 主题与原始分类的对比	392
21.8 组件使用小结	396
第 22 章 单词向量化	398
22.1 单词向量预训练模型	399
22.1.1 加载模型	399
22.1.2 查找相似的单词	400

22.1.3 单词向量	402
22.2 单词映射为向量	406
第 23 章 情感分析.....	412
23.1 使用提供的特征	413
23.1.1 使用朴素贝叶斯方法	416
23.1.2 使用逻辑回归算法	419
23.2 如何提取特征	423
23.3 构造更多特征	426
23.4 模型保存与预测	430
23.4.1 批式/流式预测任务	430
23.4.2 嵌入式预测	431
第 24 章 构建推荐系统.....	433
24.1 与推荐相关的组件介绍	434
24.2 常用推荐算法	437
24.2.1 协同过滤	437
24.2.2 交替最小二乘法	438
24.3 数据探索	439
24.4 评分预测	444
24.5 根据用户推荐影片	446
24.6 计算相似影片	452
24.7 根据影片推荐用户	454
24.8 计算相似用户	457



Alink 快速上手

随着大数据时代的到来和人工智能的崛起，机器学习所能处理的场景更加广泛和多样。算法工程师们不单要处理好批式数据的模型训练与预测，也要能处理好流式数据，并需要具备将模型嵌入企业应用和微服务上的能力。为了取得更好的业务效果，算法工程师们需要尝试更多、更复杂的模型，并需要处理更大的数据集，因此他们使用分布式集群已经成为常态。为了及时应对市场的变化，越来越多的业务选用在线学习方式来直接处理流式数据、实时更新模型。

Alink 就是为了更好地满足这些实际应用场景而研发的机器学习算法平台，以帮助数据分析和应用开发人员轻松地搭建端到端的业务流程。

1.1 Alink是什么

Alink 是阿里巴巴计算平台事业部 PAI (Platform of Artificial Intelligence) 团队基于 Flink 计算引擎研发的批流一体的机器学习算法平台，该平台提供了丰富的算法组件库和便捷的操作框架。借此，开发者可以一键搭建覆盖数据处理、特征工程、模型训练、模型预测的算法模型开发全流程。Alink 的名称取自相关英文名称，即 Alibaba、Algorithm、AI、Flink 和 Blink 中的公共部分。Alink 提供了 Java 接口和 Python 接口（PyAlink），开发者不需要 Flink 的技术背景也可以轻松构建算法模型。

Alink 在 2019 年 11 月的 Flink Forward Asia 2019 大会上宣布开源。Alink 所在的 GitHub 地址如链接 1-1 所示，欢迎大家下载使用、反馈意见、提出建议，以及贡献新的算法。

1.2 免费下载、安装

可以在 Alink 开源网站获取其最新版本。为了方便用户查看 Alink 文档，解决 Alink 的本地

安装、使用问题，以及解决 Alink 在集群上部署、运行等方面的问题，我们提供了如下专门的资料供读者参考。相关的网址如下：

- 【主页，参见链接 1-1】完整的开源内容：代码、函数说明、注意事项、安装包、历史版本。
- 【文档，参见链接 1-2】优化文档显示，便于查询、阅读。
- 【指南，参见链接 1-3】侧重介绍 Alink 的安装、运行、部署等方面的内容。
- 【技巧，参见链接 1-4】作者的知乎主页，内容丰富，涉及的话题比较发散。

另外，在开源网站的首页中有 Alink 开源用户钉钉群的二维码，欢迎大家加入该钉钉群。若有问题，可以随时在该钉钉群里与大家沟通、交流。注意：本书提供的额外参考资料，如文中的“链接 1-1”“链接 1-2”等，可从封底的“读者服务”处获取。

1.3 Alink 的功能

1.3.1 丰富的算法库

Alink 拥有丰富的批式算法和流式算法，可帮助数据分析和应用开发人员端到端地完成从数据处理、特征工程、模型训练到预测的整个流程工作。Alink 开源算法如图 1-1 所示，在 Alink 提供的开源算法模块中，每一个模块都包含流式算法和批式算法。比如线性回归，包含批式线性回归训练、流式线性回归预测和批式线性回归预测。

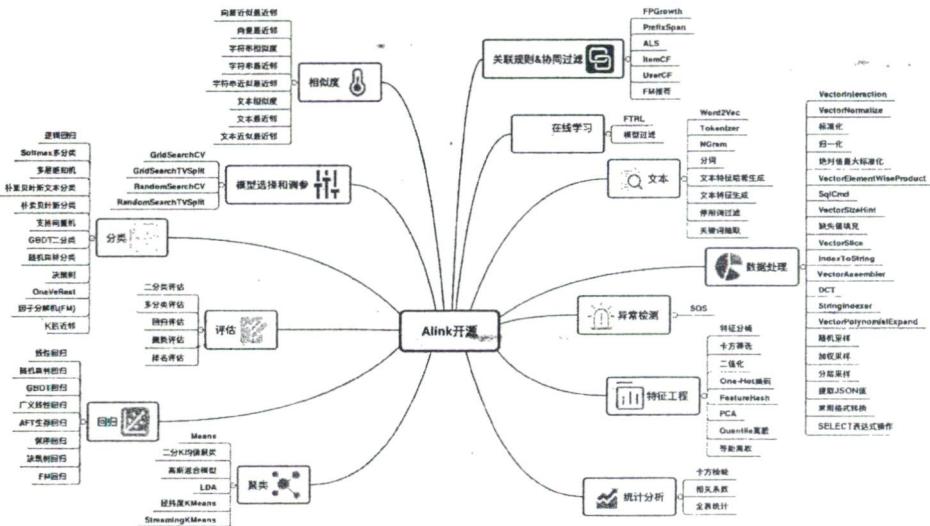
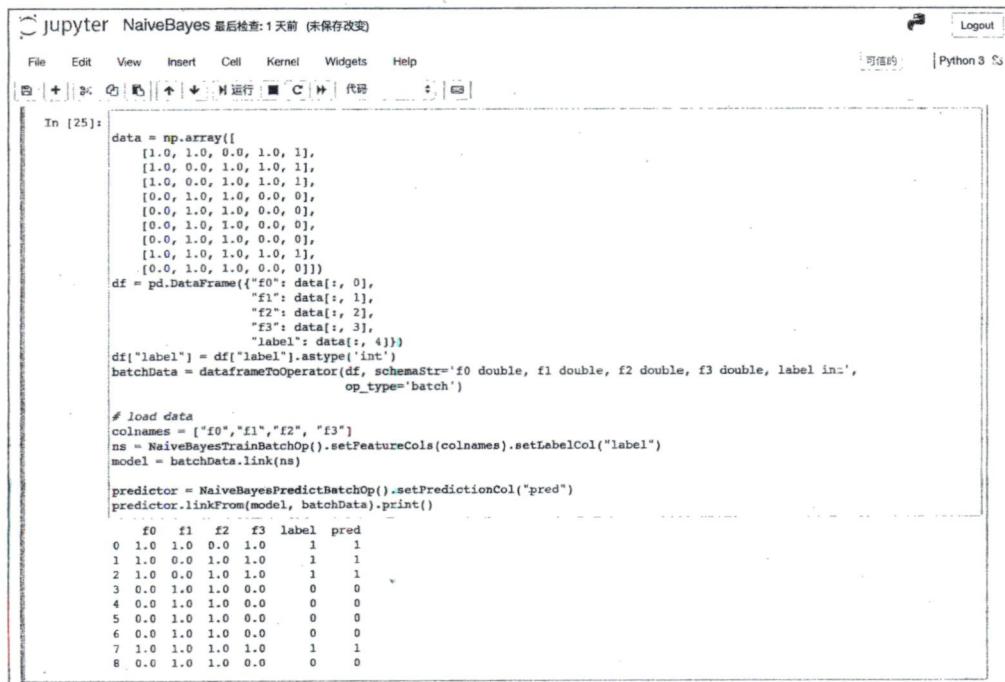


图 1-1

1.3.2 多样的使用体验

Alink 使用 Java 研发，原生提供了整套 Java 调用接口，可以在单机中编辑、调试、运行 Alink 任务，也可以将 Alink 任务编译、发布到 Flink 集群中运行。此外，Alink 也提供了 Python 版本：PyAlink。可以通过 Jupyter 等 Notebook 的方式使用 Alink，提升其交互式和可视化体验；PyAlink 既支持单机运行，又支持集群提交。PyAlink 打通了 Operator（Alink 组件）和 DataFrame 的接口，因此，Alink 的整个算法流程可无缝融入 Python。PyAlink 也提供了使用 Python 函数来调用 UDF 或者 UDTF 的方法。PyAlink 在 Notebook 中的使用如图 1-2 所示。图 1-2 中展示了一个模型训练预测，并打印出了预测结果的过程。



```

jupyter NaiveBayes 最后检查: 1 天前 (未保存改变)
File Edit View Insert Cell Kernel Widgets Help
Logout 可选的 Python 3
In [25]:
data = np.array([
    [1.0, 1.0, 0.0, 1.0, 1],
    [1.0, 0.0, 1.0, 1.0, 1],
    [1.0, 0.0, 1.0, 1.0, 1],
    [0.0, 1.0, 1.0, 0.0, 0],
    [0.0, 1.0, 1.0, 0.0, 0],
    [0.0, 1.0, 1.0, 0.0, 0],
    [0.0, 1.0, 1.0, 0.0, 0],
    [0.0, 1.0, 1.0, 1.0, 1],
    [0.0, 1.0, 1.0, 0.0, 0]
])
df = pd.DataFrame({'f0': data[:, 0],
                    'f1': data[:, 1],
                    'f2': data[:, 2],
                    'f3': data[:, 3],
                    'label': data[:, 4]})
df["label"] = df["label"].astype('int')
batchData = dataframeToOperator(df, schemaStr='f0 double, f1 double, f2 double, f3 double, label int',
                                 op_type='batch')

# load data
colnames = ["f0", "f1", "f2", "f3"]
ns = NaiveBayesTrainBatchOp().setFeatureCols(colnames).setLabelCol("label")
model = batchData.link(ns)

predictor = NaiveBayesPredictBatchOp().setPredictionCol("pred")
predictor.linkFrom(model, batchData).print()

      f0   f1   f2   f3  label  pred
0  1.0  1.0  0.0  1.0     1     1
1  1.0  0.0  1.0  1.0     1     1
2  1.0  0.0  1.0  1.0     1     1
3  0.0  1.0  1.0  0.0     0     0
4  0.0  1.0  1.0  0.0     0     0
5  0.0  1.0  1.0  0.0     0     0
6  0.0  1.0  1.0  0.0     0     0
7  1.0  1.0  1.0  1.0     1     1
8  0.0  1.0  1.0  0.0     0     0

```

图 1-2

1.3.3 与SparkML的对比

在离线学习算法方面，Alink 与 SparkML 的性能基本相当。图 1-3 给出的是一些经典算法的性能对比：对于同一算法，采用相同的数据集、同样的迭代次数等参数。其中的加速比指的

是，SparkML 所用的时间除以 Alink 所用的时间之比。若加速比的值为 1x(1 倍)，则说明 Alink 与 SparkML 的性能相当。加速比的值越大，说明 Alink 的性能越好。

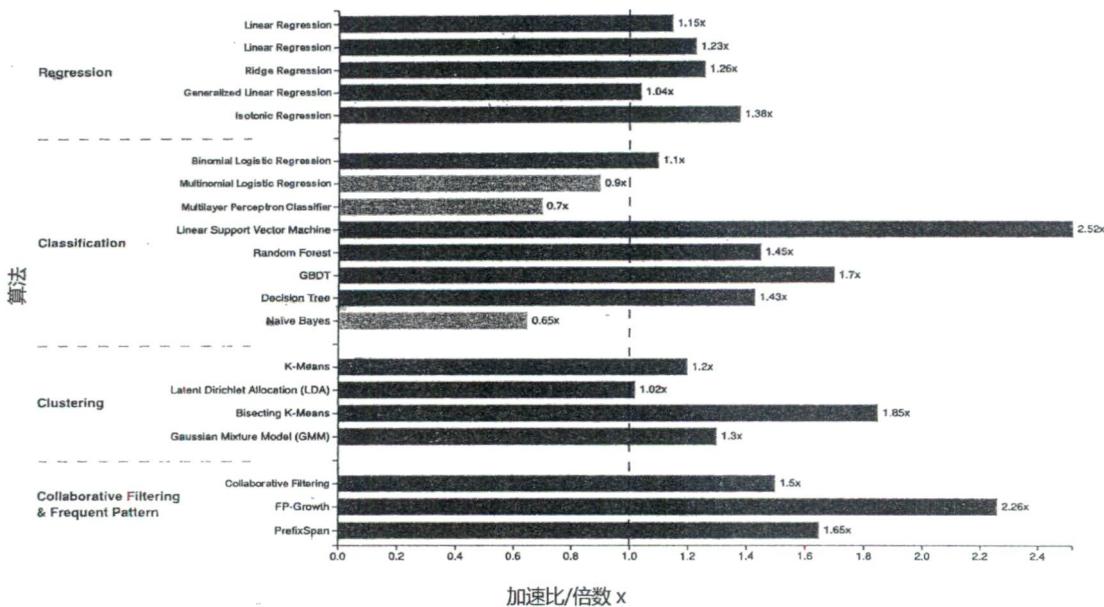


图 1-3

通过图 1-3 可以看出，Alink 在大部分算法方面的性能优于 SparkML，在个别算法方面的性能比 SparkML 弱，二者在整体上是一个相当的水平。但是，在功能的完备性方面，Alink 更有优势，Alink 除了覆盖了 SparkML 的算法，还包含流式算法、流批混跑、在线学习、中文分词等功能。

1.4 关于数据和代码

本书的全部实验都提供了 Java 源代码，所用数据集均可以通过 Web 下载。

Java 源代码的地址：参见链接 1-5。

该代码在 MacBook Pro (Intel Core i7-4770HQ CPU @ 2.20 GHz 四核八线程, 16GB 1600MHz DDR3) 计算机上运行通过，本书记录的实验运行时间也是在该计算机上的运行时间。

本书所使用的数据集都是可以通过 Web 获取的免费数据集。这些免费的数据集在书中都有

相应的介绍，并提供了数据来源。其大部分可以直接点击下载；不过，若要下载来自 Kaggle 的数据集，则需要注册 Kaggle 账户，但下载是免费的。

每个章节的示例代码都在使用不同的数据，建议用户将数据下载到本地使用，并在本地建立一个数据存放的总文件夹。作者在本地的数据总文件夹为“/Users/yangxu/alink/data/”。随后，在数据总文件夹下建立不同的子文件夹，用于存放不同的数据集。子文件夹的命名可以参考示例代码中的命名。这样示例的代码下载到本地后，只需修改一处数据总文件夹路径变量的设置，便可直接运行了。

数据总文件夹的路径变量设置在 Utils 中，为静态变量 ROOT_DIR：

```
public static final String ROOT_DIR = "/Users/yangxu/alink/data/";
```

在各个章节的代码中，所用的数据路径如下：

```
static final String DATA_DIR = Chap01.ROOT_DIR + "iris" + File.separator;
```

该路径由 ROOT_DIR 和当前章节所用的子文件夹名称拼接而成。注意：代码最后使用的 File.separator 是为跨平台而设的。

1.5 简单示例

这里将通过几个简单的 Alink 示例，让读者留下一个初步的 Alink 印象。

1.5.1 数据的读/写与显示

我们以常用的鸢尾花(iris)数据集为例，演示一下如何读取数据。该数据可以通过链接 1-6 直接下载。Alink 的 CSV 格式数据源读取组件，不但可以读取本地文件，还可以直接读取网络文件；在该组件读取文件的时候需要指定各列数据的名称和类型。下面是具体的代码。在此，读取出数据后，选择前 5 条数据进行打印输出：

```
CsvSourceBatchOp source =
new CsvSourceBatchOp()
.setFilePath("http://archive.ics.uci.edu/ml/machine-learning-databases"
+ "/iris/iris.data")
.setSchemaStr("sepal_length double, sepal_width double, petal_length double, "
+ "petal_width double, category string");

source.firstN(5).print();
```

(2) 因为使用批式的模型，这里还是通过 AlinkFileSourceBatchOp 读取文件 gmv_reg.model，得到模型数据，并记为 lr_model。

(3) 定义线性回归的流式预测组件 (LinearRegPredictStreamOp) 为 predictor，指定输出结果列的名称。注意：因为流式任务的流程中无法接入批式的数据，所以将批式模型数据 lr_model 作为 LinearRegPredictStreamOp 构造函数的参数传入。在流式任务执行前，会先完成批式模型数据的导入。

(4) 组装预测流程。由于流式预测组件 predictor 已经在构造函数中导入了模型数据，因此只需连入一个流式预测数据即可，不要使用 linkFrom 方法。整个流程可以写得更简单，可直接把预测数据作为源头，接入流式 SQL SELECT 操作，生成 x2 (注意这里代码的写法，与批式调用时完全一致)，然后连接流式预测组件 predictor，并对预测结果进行打印。

(5) 调用 StreamOperator.execute()，执行流式任务。

流式预测结果，如表 1-3 所示，其与批式预测的结果完全相同。

表 1-3 流式预测的结果

x	x2	pred
2018	4072324	2142.4048
2019	4076361	2682.2263

1.5.4 定义Pipeline，简化操作

前面介绍了 Alink 在批式模型训练、批式预测和流式预测中的例子，可以看到这三个流程中都涉及两个步骤（使用 SQL SELECT 构造新的特征项，以及线性回归的训练或者预测），因此，可以对该功能进行抽象，形成 Pipeline（管道）。其具体概念和用法在本书后面会介绍，这里就不展开说明了。这里只是通过示例，让读者有一个初步的印象：可以通过 Pipeline 简化操作。

示例代码如下：

```
MemSourceBatchOp train_set = ...

Pipeline pipeline = new Pipeline()
    .add(
        new Select()
            .setClause("*, x*x AS x2")
    )
    .add(
        new LinearRegression()
            .setFeatureCols("x", "x2")
```

```

.setLabelCol("gmv")
.setPredictionCol("pred")
);

File file = new File(DATA_DIR + "gmv_pipeline.model");
if (file.exists()) {
    file.delete();
}

pipeline.fit(train_set).save(DATA_DIR + "gmv_pipeline.model");

BatchOperator.execute();

```

这段代码由四部分构成：

- 训练数据 train_set 生成的代码在前面出现过，这里不再重复。
- 核心为 Pipeline 的构成，其需要以下两个操作：
 - Select
设置了 SQL 子语句 “*,**x*** AS x2”。注意第一个字符为“*”，因此会匹配输入数据表中出现的所有列。因为在批式训练场景中，输入的列为“x, gmv”，而在预测场景中输入的只有一列“x”，所以使用“*”会同时适用这两个场景。
 - LinearRegression
设置了训练时需要的参数 FeatureCols 和 LabelCol，也设置了预测时需要的 PredictionCol。
- 有了 Pipeline 的定义后，对训练数据执行 fit 方法，就会得到 PipelineModel（管道模型）。可以直接使用 PipelineModel；也可选择保存 PipelineModel，后面用在不同场景中。Alink 提供了简单的保存 PipelineModel 的方法，提供文件路径，运行 save 方法即可。注意，save 方法将 PipelineModel 对应的模型连接到了 sink 组件，还需要等到执行 BatchOperator.execute() 时，才会真正写出模型。
- 最后使用 BatchOperator.execute()，执行批式任务。

下面我们将通过读取模型文件 gmv_pipeline.model，得到 PipelineModel，并使用其 transform 方法，对批式数据和流式数据进行预测。

读取模型文件，得到 PipelineModel 的代码很简单：

```
PipelineModel pipelineModel = PipelineModel.load(DATA_DIR + "gmv_pipeline.model");
```

对于批式数据的预测代码如下：

```

BatchOperator <?> pred_batch
= new MemSourceBatchOp(new Integer[] {2018, 2019}, "x");

pipelineModel

```

```
.transform(pred_batch)
.print();
```

结果如表 1-4 所示。

表 1-4 Pipeline 批式预测的结果

x	x2	pred
2018	4072324	2142.4048
2019	4076361	2682.2263

对于流式数据的预测代码如下，我们看到 pipelineModel 对于批式数据和流式数据处理所用的方法名称都是一样的，使用方式也是一样的；但 transform 方法的输出结果究竟是批式的还是流式的，取决于输入数据是批式的还是流式的：

```
MemSourceStreamOp pred_stream
= new MemSourceStreamOp(new Integer[] {2018, 2019}, "x");

pipelineModel
.transform(pred_stream)
.print();

StreamOperator.execute();
```

使用 Pipeline 方式，流式预测的结果如表 1-5 所示，与批式预测的结果（见表 1-4）相同。

表 1-5 Pipeline 流式预测的结果

x	x2	pred
2019	4076361	2682.2263
2018	4072324	2142.4048

1.5.5 嵌入预测服务系统

除了使用 Alink 算法组件直接对批式的数据或者流式的数据进行预测，用户也希望我们能提供 SDK 的方式，即，由参数或模型数据直接构建一个本地的 Java 实例，可以对单条数据进行预测，我们称之为 LocalPredictor。如此一来，预测不再必须由 Flink 任务完成，而可以嵌入提供 RestAPI 的预测服务系统中，或者嵌入用户的业务系统里。

有了存储好的 PipelineModel 模型，可以直接使用 LocalPredictor 的构造函数来构建实例。构造函数需要两个参数，一个是 PipelineModel 模型文件的路径，另一个是所要预测数据的各数

据列名称和类型，即输入一个 Alink Schema String 格式（详见 2.8 节的内容）的参数。这里的参数为 "x int"，表示输入的预测数据只有 1 列，名称为 x，类型为整型。具体代码如下：

```
LocalPredictor predictor
= new LocalPredictor(DATA_DIR + "gmv_pipeline.model", "x int");
```

LocalPredictor 输入的预测数据是 Row 格式的，输出的预测结果也是 Row 格式的。Row 格式本身并没有列名和类型的定义，需要通过在构造函数中输入预测数据的 Schema 来获取预测数据各数据列的名称和类型；LocalPredictor 可以根据 PipelineModel 模型的计算过程，推导出预测结果的 Schema（结果数据列名称及类型）。对于我们刚构建的 localPredictor，使用 getOutputSchema 方法获取 Schema 信息并打印显示 Schema 信息，具体代码如下：

```
System.out.println(predictor.getOutputSchema());
```

运行结果如下：

```
root
|-- x: INT
|-- x2: INT
|-- pred: DOUBLE
```

可以看出，LocalPredictor 的预测结果与批式和流式预测结果一样，其预测输出共 3 列，最关键的分类预测结果列 “pred” 在最后。

LocalPredictor 使用 map 方法进行预测，具体代码如下：

```
for (int x : new int[] {2018, 2019}) {
    System.out.println(predictor.map(Row.of(x)));
}
```

计算结果如下：

```
2018, 4072324, 2142.404761955142
2019, 4076361, 2682.2262857556343
```

最右面的数值是预测结果，对其保留小数点后 4 位有效数字，得到的结果与前面批式/流式任务计算的结果相同。

2

系统概况与核心概念

本章将从 Alink 涉及的基本概念开始，逐步深入，帮助读者系统地了解 Alink。在介绍 Alink 基本概念的同时，也介绍了一些小技巧，以帮助读者写出更简练、高效的代码。

2.1 基本概念

数据集（DataSet）与数据流（DataStream）的区别在于用户是否可以假定数据有界。数据集（DataSet）的数据有界就意味着，数据是静止的、确定的，内容和个数都不会变化。

在逐条读取数据时，数据集（DataSet）的数据是有界的、个数确定，我们可以执行某些操作。比如，统计该数据集的记录总数，计算平均值、方差等统计量；在逻辑上，可以看作把该数据集按照某个操作处理完，得到一个新的数据集，然后对该新数据集进行另外一个操作，得到另一个新的数据集。

批式（Batch）数据对应着数据集（DataSet），流式（Stream）数据对应着数据流（DataStream）。

数据处理的基本流程就是三部分：数据源（Source）、算法组件（Operator）和数据导出（Sink），如图 2-1 所示。

考虑到数据集和数据流之间的区别，处理操作被分为批式处理（Batch Processing）和流式处理（Stream

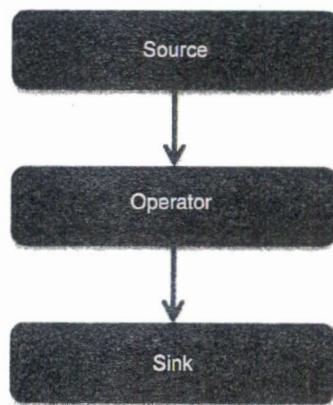


图 2-1

Processing），如图 2-2 所示。在批式处理中，批式数据源组件（Batch Source）读入的数据为数据集；批式导出组件（Batch Sink）将数据集导出到文件系统或数据库；批式算法组件（BatchOperator）的输入和计算结果都为数据集。对于流式处理，流式数据源组件（Stream Source）用来接入数据流；流式导出组件（Stream Sink）负责将数据流导出，即将数据流导出到文件系统或数据库；流式算法组件（StreamOperator）的处理粒度是单条数据，即从输入的数据流中逐条获取数据进行计算，该计算结果为若干条数据，这些数据会进入输出数据流。连接流式数据源组件、各个流式算法组件和流式导出组件，构成数据流的管道。



图 2-2

Alink 将 Flink Table 作为数据集和数据流的统一表示，其中数据行的类型为 org.apache.flink.types.Row。

Alink 为了统一各算法模块间交换数据的格式，确定将 Row 格式作为各算法处理的单条数据的标准类型；相应的批数据与流数据的类型为 Flink Table。如此一来，SQL 操作与算法模块间可以通过 Flink Table 来传递数据，这样 SQL 操作与算法操作就可以出现在同一个工作流中，方便用户使用。

2.2 批式任务与流式任务

批式任务与流式任务看待数据的方式不同，批式任务将数据当作一个整体（数据集）来看待，流式任务将每条数据作为一个基本单位。

我们先看一个典型的批式任务场景，包括读取数据、数据预处理、特征生成、模型训练、

预测、评估环节。各个组件依次对上游的结果数据集进行处理。图 2-3 为实际某算法平台的一个批式任务运行状态截图。

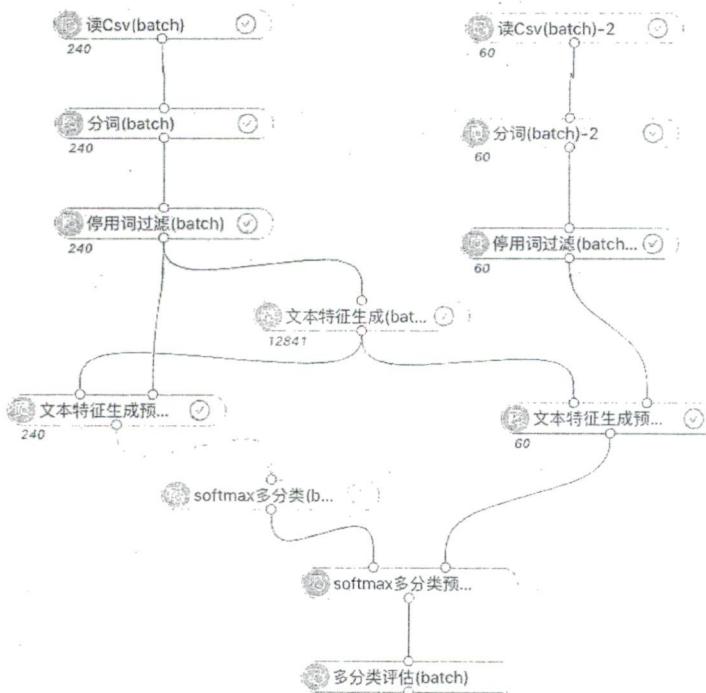


图 2-3

整个流程可用一个有向无环图（Directed Acyclic Graph，缩写为 DAG）来表示。在典型的工作流场景中，每个组件是在其所依赖的前序组件都完成后，才能开始执行的。图 2-4 中正在运行的是“softmax 多分类”组件，关注此处细节。



图 2-4

接入的数据连线是动态的虚线，表明从前序组件获取处理数据进行训练；而计算完成的组件的结果数据集是固定的，在图标左下方显示了该结果数据集的记录条数。“文本特征生成预...

处理”组件生成的带特征的训练数据为 240 条。

我们再来看流式任务，和“流水线”的操作类似，每个组件就相当于一名工人，每经过一名工人，产品上就会多一个组件；各个计算组件是同时启动的，每条数据在前一个组件处理完成后，会自动推到下一个组件。对于流式任务，经常用单位时间处理数据的条数来衡量效率。评估需要并发配置的计算资源，以便对线上请求及时响应。

如图 2-5 所示，我们看到流式任务的所有组件都在运行状态，每个组件左下角显示的是其每秒处理的数据条数。

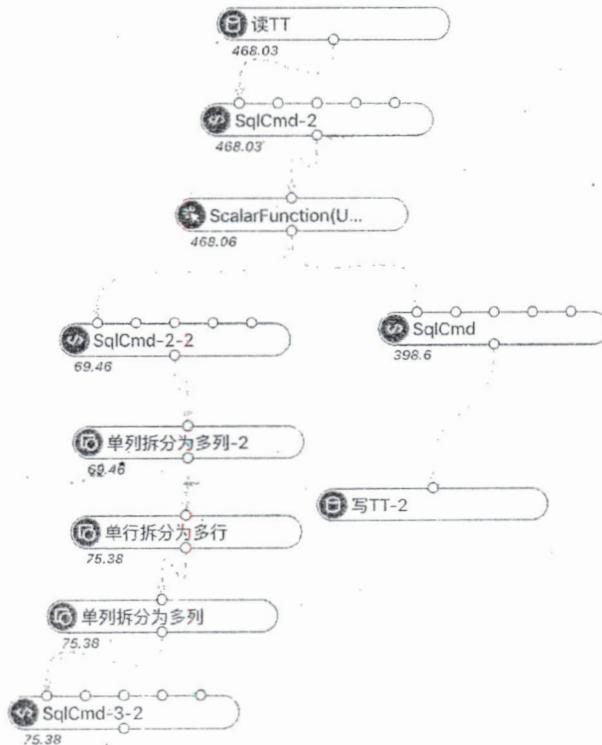


图 2-5

图 2-6 还展示了“流批混跑”的情形，右上方和右侧中部的三个组件为批式任务组件，左侧及下方的组件为流式任务组件。先运行批式任务，其最终输出结果会作为流式任务的初始化数据；批式任务都运行结束后，启动流式任务。

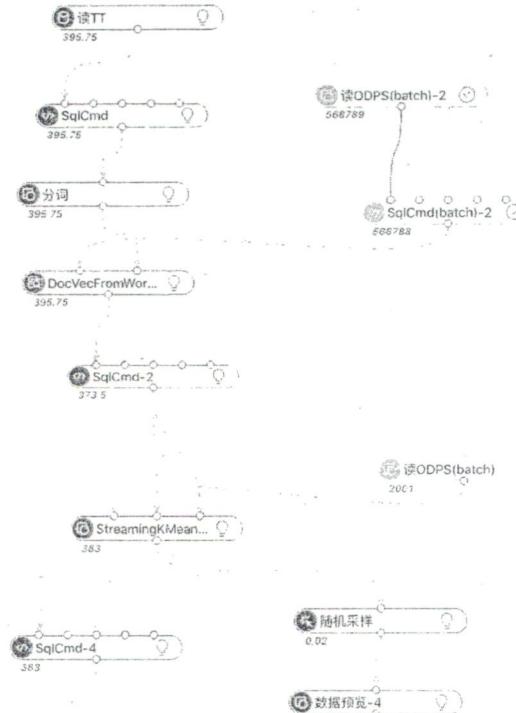


图 2-6

2.3 Alink=A+link

由前面关于批式任务与流式任务的介绍可知，批式任务与流式任务的数据计算及处理操作都发生在组件中，各组件间的连线就是数据的通路。批式任务和流式任务中有关组件与连接的描述是通用的。

Alink 定义了组件的抽象基类 AlgoOperator，规范了组件的基本行为。由 AlgoOperator 派生出了两个基类：用于批式计算及处理场景的批式算法组件（BatchOperator）和用于流式计算及处理场景的流式算法组件（StreamOperator）。

组件间的连接是通过定义 link 方法实现的，比如，组件 algoA 的输出是组件 algoB 的输入，而组件 algoB 的输出又是组件 algoC 的输入，则可以通过组件的 link 方法表示：

```
algoA.link(algoB).link(algoC)
```

Alink的名称可以从这个角度进行解读：“Alink=A+link”。这里的A代表Alink的全部算法组件，其都是由抽象基类 AlgoOperator 派生出来的；link 是 AlgoOperator 各派生组件间的连接方法。

2.3.1 BatchOperator和StreamOperator

由算法组件的抽象基类（AlgoOperator）派生出两个基类：批式算法组件（BatchOperator，或称为批式处理组件、批式组件）和流式算法组件（StreamOperator，或称为流式处理组件、流式组件），它们的UML类图如图 2-7 所示。

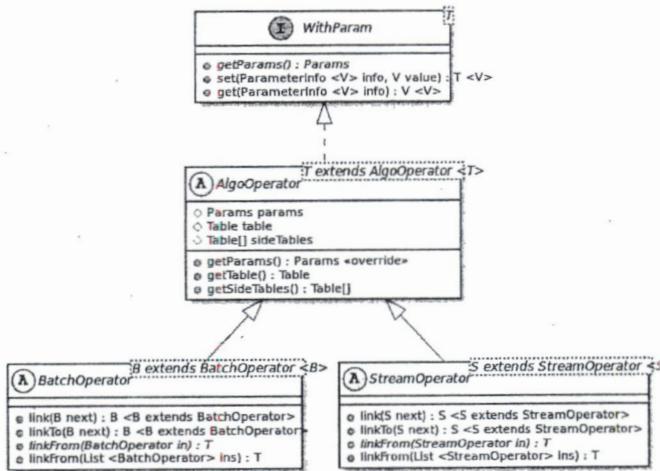


图 2-7

由 UML 类图，可以看到如下信息：

- `WithParam` 定义了参数设定和获取的接口。
- 抽象基类 `AlgoOperator` 中有 `Params` 类型的成员变量 `params`，并实现了参数设定和获取的接口；还定义了 `Table` 类型的变量 `table`，以及 `Table` 类型数组的变量 `sideTables`，用来存放算法的结果，并提供方法供后续组件读取。
- `AlgoOperator` 下面有 2 个派生泛型基类：`BatchOperator`（批式算法组件）和 `StreamOperator`（流式算法组件）。可以看到，这两种算法组件都支持 `link` 的操作；但批式算法组件只能连接一个或多个批式算法组件，流式算法组件只能连接一个或多个流式算法组件。对于需要批式数据和流式数据混合处理的算法，我们会将其作为流式算法组件，一般会将批式数据通过流式算法组件的构造函数传入。

Alink 将每个批式操作定义为一个批式组件（BatchOperator），每个批式组件在命名上都以“BatchOp”为后缀；同样，将每个流式操作定义为一个流式组件（StreamOperator），每个流式组件在命名上都以“StreamOp”为后缀。

通过这样的定义，批式任务和流式任务都可以用相同的方式进行描述，这样就可以大大降低批式任务和流式任务转换的代价。若需要将一个批式任务改写为流式任务，只需要将批式组件后面的“BatchOp”后缀变为“StreamOp”，相应的 link 操作便可转换为针对流式数据的操作。也正是因为 Alink 的批式组件和流式组件有如此密切的联系，所以才能将机器学习的管道（Pipeline）操作推广到流式场景。

算法的输入数据，在大多数情况下可以用一个表（Table）表示，但也有不少情况下需要用多个表（Table）才能表示。比如 Graph 数据，一般包括 Edge Table 和 Node Table，这两个表在一起才是完整的表示。算法的输出也是这样的情况。在大多数情况下可以用一个表（Table）表示，但也有不少情况下需要用多个表（Table）才能表示。比如，Graph 操作的结果还是 Graph，仍然需要用 2 个 Table 分别表示结果图中各条边和各个顶点的信息。自然语言方面的常用算法 LDA（Latent Dirichlet Allocation）的计算结果为 6 个 Table。其算法组件包含了一个 Table 类型的成员变量 table，用来放置该组件的主输出结果（大多数情况下，算法计算的结果只有一个 Table，输出到该变量即可）。该算法组件也定义了一个 Table 类型数组的变量 sideTables，该变量用来存储在多表（Table）输出的情况下，除主表外的所有其他表。

2.3.2 link 方式是批式算法/流式算法的通用使用方式

简单地说，link 方式指的是在工作流中通过连线的方式，串接起不同的组件。link 方式给我们带来的一个简化是，前序组件的产出结果可能比较复杂，比如描述的是一个机器学习模型，我们不必了解其细节、不必详细描述它，只要通过“link”的方式将两个组件建立连接，后面的组件即可通过该连接（link）来获取前序组件的处理结果数据、数据的列数，以及各列的名称和类型。

1. link、linkTo 和 linkFrom

连接（link）是有方向的，组件 A 连接组件 B，即先执行组件 A，然后将计算结果传给组件 B 继续执行，则组件间的关系可以通过以下三种方式表示：

- A.linkTo(B)
- B.linkFrom(A)
- A.link(B)

这里，可将 link 看作 linkTo 的简写。

关于两个组件 A、B 之间的连接 (link) 关系，很容易理解。在实际应用中，我们还会遇到更复杂的情况，但使用 link 方法仍可以轻松处理。

(1) 一对多的情况

组件 B1、B2、B3 均需要组件 A 的计算结果，组件间的关系可以通过以下多种方式表示：

- A.linkTo(B1), A.linkTo(B2), A.linkTo(B3)
- A.link(B1), A.link(B2), A.link(B3)
- B1.linkFrom(A), B2.linkFrom(A), B3.linkFrom(A)
- A.linkTo(B1), A.link(B2), B3.linkFrom(A)

从上述表示方式上可以看出，表示方式可以很灵活。因为组件 B1、B2、B3 与组件 A 的关系是独立的，所以可以分别选用表示方式。

(2) 多对一的情况

组件 B 同时需要组件 A1、A2、A3 的计算结果，表示方式只有一种：

B.linkFrom(A1, A2, A3)

即，linkFrom 可以同时接入多个组件。

2. 深入理解

批式处理组件 BatchOperator 的相关代码如下：

```
public abstract class BatchOperator {
    ...
    public BatchOperator link(BatchOperator f) {
        return linkTo(f);
    }

    public BatchOperator linkTo(BatchOperator f) {
        f.linkFrom(this);
        return f;
    }

    abstract public BatchOperator linkFrom(BatchOperator in);

    public BatchOperator linkFrom(List<BatchOperator> ins) {
        if (null != ins && ins.size() == 1) {
            return linkFrom(ins.get(0));
        } else {
            throw new RuntimeException("Not support more than 1 inputs!");
        }
    }
    ...
}
```

流式处理组件 StreamOperator 的相关代码如下：

```
public abstract class StreamOperator {
    ...
    public StreamOperator link(StreamOperator f) {
        return linkTo(f);
    }

    public StreamOperator linkTo(StreamOperator f) {
        f.linkFrom(this);
        return f;
    }

    abstract public StreamOperator linkFrom(StreamOperator in);

    public StreamOperator linkFrom(List <StreamOperator> ins) {
        if (null != ins && ins.size() == 1) {
            return linkFrom(ins.get(0));
        } else {
            throw new RuntimeException("Not support more than 1 inputs!");
        }
    }
    ...
}
```

从上述代码中，我们可以看出 link、linkTo 与 linkFrom 的关系。首先看看 link 与 linkTo 的关系：

```
public BatchOperator link(BatchOperator f) {
    return linkTo(f);
}

public StreamOperator link(StreamOperator f) {
    return linkTo(f);
}
```

显然，link 等同于 linkTo，可以将 link 看作 linkTo 的简写。

然后，我们再将注意力转向 linkTo 与 linkFrom：

```
public BatchOperator linkTo(BatchOperator f) {
    f.linkFrom(this);
    return f;
}

public StreamOperator linkTo(StreamOperator f) {
    f.linkFrom(this);
    return f;
}
```

显然，A.linkTo(B)等效于 B.linkFrom(A)，在 linkTo 组件具体实现时，只要实现 linkFrom 方法即可。

基类 BatchOperator 和 StreamOperator 均定义了输入参数为一个组件的抽象方法 linkFrom，该方法需要继承类进行实现；抽象类 BatchOperator 和 StreamOperator 同时也实现了一个输入是组件列表的方法 linkFrom，在组件列表中只含有一个组件的时候，该方法会调用前面的抽象方法 linkFrom；在其他情况下，则会抛出异常。

在我们实现的组件中，大部分组件只支持输入一个组件，即只要实现输入参数为一个组件的抽象方法 linkFrom 就可以了；有的组件支持输入多个组件，则需要重载输入为组件列表的方法 linkFrom，并将输入为一个组件的情况看作输入为算法列表时，列表中的组件个数为一个的情况。

2.3.3 link的简化

link 组件是 Alink 的基本使用方式；但对于一些常用的功能，比如取前 N 条数据、随机采样、SQL SELECT、数据过滤等，Alink 定义了相关的方法（方法内部的实现过程也是 link 相应的组件），这样代码写起来会更简练。

对比下面两段代码，执行的是同样的功能；但是很明显，右边的代码更简练，也更易懂：

<pre> source .link(new SelectBatchOp() .setClause("petal_width, category")) .link(new FilterBatchOp() .setClause("category='Iris-setosa'")) .link(new SampleBatchOp() .setRatio(0.3)) .link(new FirstNBatchOp() .setSize(5)) .print(); </pre>	<pre> source .select("petal_width, category") .filter("category='Iris-setosa'") .sample(0.3) .firstN(5) .print(); </pre>
---	--

2.3.4 组件的主输出与侧输出

组件可能需要一个或多个输入，通过 linkFrom 方法便可将多个上游组件的输出连接到该组件。组件也会产生一个或多个输出。对大部分算法组件来说，结果只有一个数据表，输出是唯一的，但是有些算法组件会产生多个数据表。这时，就需要确认一个数据表作为主输出，其余

数据表作为侧输出（Side Output）。

侧输出（Side Output）有两个重要方法：

- `getSideOutputCount()` 获得该组件侧输出的个数。
- `getSideOutput(int index)` 方法通过索引号获取具体的侧输出，每个侧输出是 BatchOperator 或者 StreamOperator。

比如，我们在做机器学习实验的时候，经常要把原始数据分为训练集和测试集，数据划分组件就会对应两个输出：主输出为训练集；侧输出（Side Output）只有一个，即输出测试集。详细的例子可以参考 7.2 节。

2.4 Pipeline与PipelineModel

Alink 提供了 Pipeline/PipelineModel，其在功能和使用方式上与 Scikit-learn 和 SparkML 的 Pipeline/PipelineModel 类似。保持训练和预测过程中数据处理的一致性，这样调用过程清晰、简练，也降低了用户的学习成本。Alink Pipeline/PipelineModel 与批式/流式组件（BatchOperator/StreamOperator）一样，将 Flink Table 作为计算输入和输出结果的数据类型。

Scikit-learn 和 SparkML 的 Pipeline 是针对批式数据的训练和预测设计的。Alink 不仅支持批式数据的训练和预测，也支持将批式训练出的模型用于预测流式数据。

2.4.1 概念和定义

管道（Pipeline）的概念源于 Scikit-learn。可以将数据处理的过程看成数据在“管道”中流动。管道分为若干个阶段（PipelineStage），数据每通过一个阶段就发生一次变换，数据通过整个管道，也就依次经历了所有变换。

如果要在管道中加入分类器，对数据进行类别预测，就涉及模型的训练和预测，这需要分两个步骤完成。所以，管道也会被细分为管道定义与管道模型（PipelineModel）。在管道定义中，每个 PipelineStage 会按其是否需要进行模型训练，分为估计器（Estimator）和转换器（Transformer）。随后，可以对其涉及模型的部分，即对估计器（Estimator）进行估计，从而得到含有模型的转换器。该转换器被称为 Model，并用 Model 替换相应的估计器，从而每个阶段都可以直接对数据进行处理。我们将该 Model 称为 PipelineModel。

上面介绍了几个概念，图 2-8 会用图形表示它们之间的关系，并进行深入介绍。

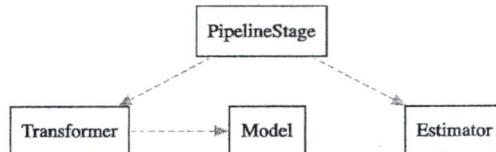


图 2-8

- 转换器（Transformer）：Pipeline 中的处理模块，用于处理 Table。输入的是待处理的数据，输出的是处理结果数据。比如，向量归一化是一个常用的数据预处理操作。它就是一个转换器，输入向量数据，输出的数据仍为向量，但是其范数为 1。
- Model：派生于转换器（Transformer）。其可以存放计算出来的模型。用来进行模型预测。其输入的是预测所需的特征数据，输出的是预测结果数据。
- 估计器（Estimator）：估计器是对输入数据进行拟合或训练的模型计算模块，输出适合当前数据的转换器，即模型。输入的是训练数据，输出的是 Model。

Pipeline 与 PipelineModel 构成了完整的机器学习处理过程，可以分为三个子过程：定义过程，模型训练过程，数据处理过程。

- 定义过程：按顺序罗列 Pipeline 所需的各个阶段。Pipeline 由若干个 Transformer 和 Estimator 构成，按用户指定的顺序排列，并在逻辑上依次执行。
- 模型训练过程：使用 Pipeline 的 fit 方法，对 Pipeline 中的 Estimator 进行训练，得到相应的 Model。Pipeline 执行 fit 方法后得到的结果是 PipelineModel。
- 数据处理过程：该过程指的是通过 PipelineModel 直接处理数据。

比如，LR 分类算法作为一个 Estimator，可以在构建 Pipeline 的时候进行定义，之后在 fit 的过程中，会使用 LR 的训练算法，得到 LR model，并将其作为整个 PipelineModel 的一部分；在使用 PipelineModel 处理数据时，会相应地调用 LR 算法的预测部分。

2.4.2 深入介绍

以使用朴素贝叶斯算法进行多分类为例，演示三个经典场景：批式训练、批式预测与流式预测。为了更好地体现它们之间的关联，我们用一张图表示了出来，如图 2-9 所示。

图 2-9 有如下特点：

- (1) 左边这列组件展示了批式训练的流程，中间那列组件展示了批式预测的流程，右边那列组件展示了流式预测的流程。
- (2) 训练和预测都经历了四个阶段：缺失值填充、MultiStringIndexer（将字符串数据用索

引值替换）、VectorAssembler（将各数据字段组装为向量）、使用朴素贝叶斯算法进行训练/预测。

(3) 在 VectorAssembler 阶段，运行时不需要额外的信息。在批式训练、批式预测和流式预测的过程中，组件均可直接使用。

(4) 除 VectorAssembler 外的三个阶段，都需要对整体数据进行扫描或者迭代训练，得到模型后，才能处理(模型预测)数据。训练过程需要拿到所有的模型，缺失值填充和 MultiStringIndexer 阶段需要在得到模型后，对当前数据进行预测，将预测结果数据传给后面的阶段。

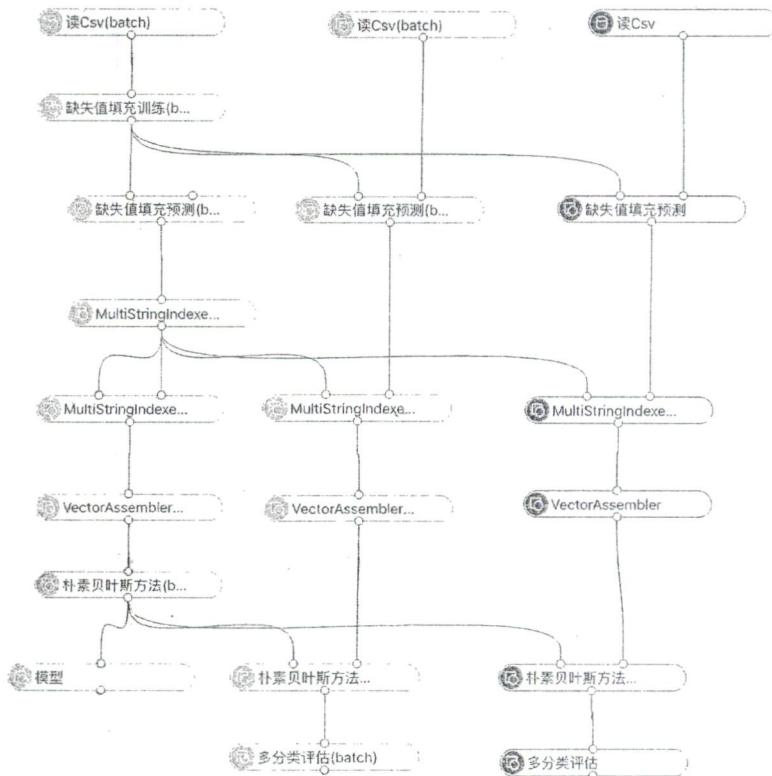


图 2-9

上面的四个阶段分别对应四个 PipelineStage，这些 PipelineStage 构成了 Pipeline。对应的 Java 代码如下：

```
Pipeline pipeline = new Pipeline()
    .add(
```

```

    new Imputer()
    .setSelectedCols("...")
    .setStrategy(Strategy.VALUE)
    .setFillValue("null")
)
.add(
    new MultiStringIndexer()
    .setSelectedCols("...")
)
.add(
    new VectorAssembler()
    .setSelectedCols("...")
    .setOutputCol("vec")
)
.add(
    new NaiveBayesTextClassifier()
    .setVectorCol("vec")
    .setLabelCol("label")
    .setPredictionCol("pred")
);

```

Pipeline 可多次复用，不同的训练数据通过同样的流程可得到不同的模型：

```

CsvSourceBatchOp trainData1 = new CsvSourceBatchOp();
CsvSourceBatchOp trainData2= new CsvSourceBatchOp();

PipelineModel model = pipeline.fit(trainData1);
PipelineModel mode12 = pipeline.fit(trainData2);

```

得到的 PipelineModel 可重复应用于不同的预测数据：

```

CsvSourceBatchOp predictData1= new CsvSourceBatchOp();
CsvSourceBatchOp predictData2= new CsvSourceBatchOp();

BatchOperator<?> predict1 = model.transform(predictData1);
BatchOperator<?> predict2 = model.transform(predictData2);

```

PipelineModel 可以无区别地应用于批式预测和流式预测：

```

CsvSourceBatchOp batchData = new CsvSourceBatchOp();
CsvSourceStreamOp streamData= new CsvSourceStreamOp();

BatchOperator<?> predictBatch = model.transform(batchData);
StreamOperator<?> predictStream = model.transform(streamData);

```

从 PipelineModel 可以得到用于本地预测的 LocalPredictor，直接嵌入 Java 应用服务：

```

String inputSchemaStr = "...";
LocalPredictor localPredictor = model.collectLocalPredictor(inputSchemaStr);

Row inputRow = Row.of("...");
Row pred = localPredictor.map(inputRow);

```

2.5 触发Alink任务的执行

Alink 的批式任务/流式任务本质上是通过触发 Flink 的 ExecutionEnvironment/StreamExecutionEnvironment 的 execute 方法来执行的。

为了调用起来更简练、方便，Alink 包装了如下两个方法：

- BatchOperator.execute()触发批式任务的执行。
- StreamOperator.execute()触发流式任务的执行。

在实际使用中，经常需要触发一些小的批式任务，获取执行的结果。Flink 的 DataSet 提供了 print 方法、collect 方法和 count 方法，Alink 的批式组件也提供了相应的方法。这些方法为用户了解中间过程的数据提供了便利，但是其会被多次触发执行，各个任务之间会有许多重复执行部分。尤其当数据量较大时，每个任务的执行时间较长，而重复计算会消耗大量的计算资源。针对这样的问题，Alink 引入了 Lazy 方式，并相应地定义了 lazyPrint()、lazyCollect() 等方法。

Lazy方式

“Lazy”的概念与“Eager”的概念相对，在许多开发语言中都会应用。Lazy 方式被称作懒操作方式，Eager 方式被称作急操作或者实时操作方式。Eager 方式与 Lazy 方式的区别在于用户发出数据请求（显示、获取、加载等）时，是立即行动，还是可以慢一些。

对于 Alink 的批式处理场景，我们在执行任务时，需要看到原始数据、中间数据，以及统计、评估指标等。若每一个任务均采用 Eager 方式，则每次数据请求都会触发一个 Flink 任务，而且由于 Flink 机制的原因，后续的 Flink 任务还会重复执行前序任务的部分计算。如果采用 Lazy 方式，则每个任务并不急于显示，可以通过一个 Flink 任务完成整个流程，在任务结束后，再返回各项数据。从数据的获取速度上看，采用 Lazy 方式比较慢，且有延迟；但是从整体的计算时间和资源的消耗方面看，采用 Lazy 方式更优。

Flink 提供了三种方法执行批式任务：print、collect 和 execute。它们都是 Eager 方式的。比如，运行到 print 方法，立即会启动 Flink 任务进行执行，并将结果打印出来。Alink 的批式组件（形如***BatchOp）提供了打印的方法；统计和评估类组件提供了 collect 方法，用户可以立即获取统计和评估结果，这些方法的命名规则为 collect+结果名称，比如 collectSummary。执行批式组件，可以使用 BatchOperator.execute()。

Alink 的 Lazy 方式需要在操作前加上“lazy”前缀，主要有如下两类操作：

- 采用 Lazy 方式打印数据、统计结果或模型信息的操作，方法定义为 lazyPrint***()。

- 采用 Lazy 方式获取结果的操作，方法定义为 lazyCollect***()。

图 2-10 详细列举了在各个使用场景下对应的方法。

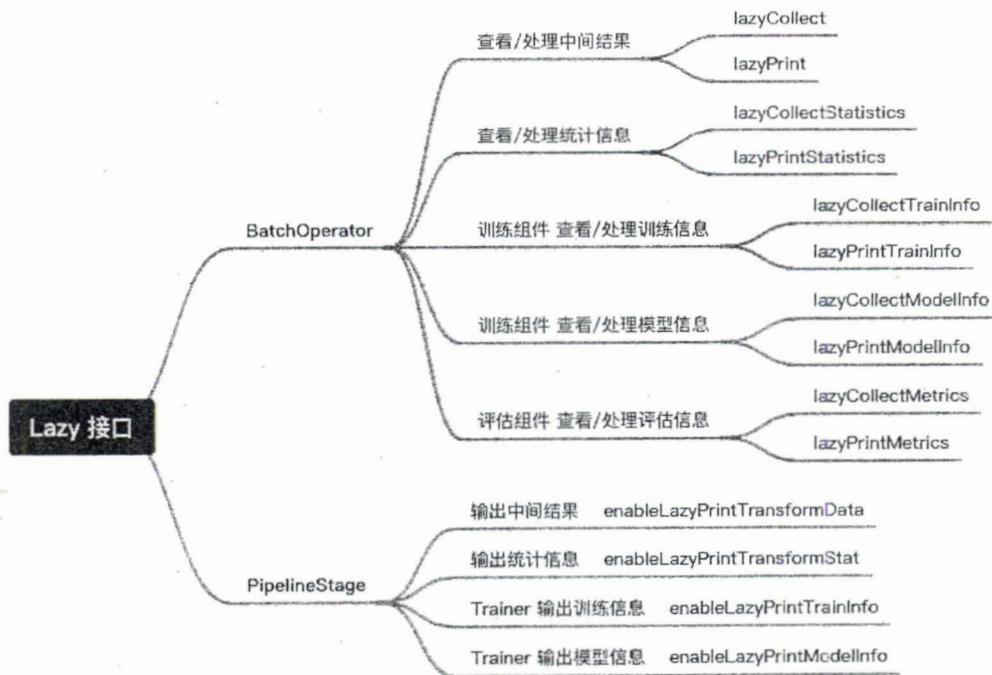


图 2-10

2.6 模型信息显示

我们在训练过程中需要了解训练所得模型的信息；对于已经存储的模型数据或者 `PipelineModel` 数据，需要采用某种方法来了解其具体内容。Alink 对各场景提供了相应的方案，如表 2-1 所示。

表 2-1 模型显示的场景

	训练过程	模型数据
单个批式训练组件	<p>【场景】在训练出来模型的同时，显示模型信息</p> <p>【用法】 调用组件相应的方法：</p> <ul style="list-style-type: none"> • <code>lazyPrintModelInfo</code>: 打印模型信息概要 • <code>lazyCollectModelInfo</code>: 获取完整模型信息，并调用回调方法做处理 	<p>【场景】从单个模型数据中提取模型信息</p> <p>【用法】 选择相应的模型信息组件（形如<code>***ModelInfoBatchOp</code>），随后调用模型信息组件相应的方法：</p> <ul style="list-style-type: none"> • <code>lazyPrintModelInfo</code>: 打印模型信息概要 • <code>lazyCollectModelInfo</code>: 获取完整模型信息，并调用回调方法做处理
Pipeline (多个批式训练组件)	<p>【场景】在 Pipeline 进行 fit 处理的时候，允许某些阶段显示相应的模型信息</p> <p>【用法】 设置 <code>PipelineStage</code> 的行为。若需要输出模型信息，则必须调用如下方法，并设置布尔型参数值为 True： <code>enableLazyPrintModelInfo(True)</code>。调用 Trainer 的 fit 方法时，输出模型信息</p>	<p>【场景】获取显示 <code>PipelineModel</code> 中各个阶段的信息、各个模型的信息</p> <p>【用法】 (1) 显示各个 <code>PipelineStage</code> 的信息 (2) 获取 <code>PipelineStage</code> 的模型数据 (3) 根据 <code>PipelineStage</code> 的信息，确定对应的模型信息组件（形如<code>***ModelInfoBatchOp</code>），随后调用模型信息组件相应的方法：</p> <ul style="list-style-type: none"> • <code>lazyPrintModelInfo</code>: 打印模型信息概要 • <code>lazyCollectModelInfo</code>: 获取完整的模型信息，并调用回调方法做处理

在训练过程中如何显示模型信息呢？可参阅本书的 8.5 节、8.6 节、9.2.5 节、9.4 节等处的示例。本节着重介绍如何针对存储的模型文件，提取模型信息。

比如，已知某个模型文件为单棵决策树模型文件，我们可以使用文件数据源读取模型数据，再连接决策树模型信息组件（`DecisionTreeModelInfoBatchOp`），并使用 `lazyPrintModelInfo` 方法打印模型信息。使用 `lazyCollectModelInfo` 方法，自定义抽取模型信息，并将决策树可视化导出到图片。具体代码如下：

```

new AkSourceBatchOp()
.setFilePath(DATA_DIR + TREE_MODEL_FILE)
.link(
    new DecisionTreeModelInfoBatchOp()
        .lazyPrintModelInfo()
        .lazyCollectModelInfo(new Consumer<DecisionTreeModelInfo>() {
            @Override
            public void accept(DecisionTreeModelInfo decisionTreeModelInfo) {
                try {
                    decisionTreeModelInfo.saveTreeAsImage(
                        DATA_DIR + "tree_model.png", true);
                }
            }
        })
);

```

```

        } catch (IOException e) {
            e.printStackTrace();
        }
    })
);
BatchOperator.execute();

```

运行结果如下：

Classification trees modelInfo:

Number of trees: 1
 Number of features: 4
 Number of categorical features: 2
 Labels: [no, yes]

Categorical feature info:

feature	number of categorical value
outlook	3
Windy	2

Table of feature importance Top 4:

feature	importance
Humidity	0.4637
Windy	0.4637
outlook	0.0725
Temperature	0

导出的决策树可视化结果如图 2-11 所示。

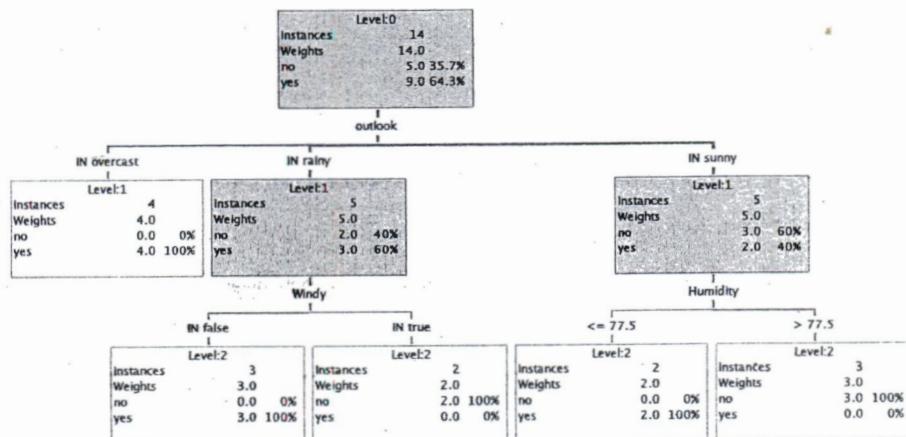


图 2-11

我们再看一下如何从 PipelineModel 中提取信息，显示模型。首先载入 PipelineModel，然后获取 Pipeline 中各阶段（PipelineStage）的信息，相关代码如下：

```
PipelineModel pipelineModel = PipelineModel.load(DATA_DIR + PIPELINE_MODEL_FILE);
TransformerBase <?>[] stages = pipelineModel.getTransformers();
for (int i = 0; i < stages.length; i++) {
    System.out.println(String.valueOf(i) + "\t" + stages[i]);
}
```

运行结果如下：

```
0 com.alibaba.alink.pipeline.sql.Select@19c1f6f4
1 com.alibaba.alink.pipeline.regression.LinearRegressionModel@46fa2a7e
```

这里共有两个阶段，第一个阶段执行 Select 操作，第二个阶段为使用 LinearRegressionModel 进行预测操作。这样，我们就确定了可以使用线性回归（Linear Regression）对应的模型信息组件来查看索引号为 1 的 PipelineStage 的信息，具体代码如下：

```
((LinearRegressionModel) stages[1]).getModelData()
    .link(
        new LinearRegModelInfoBatchOp()
            .lazyPrintModelInfo()
    );
BatchOperator.execute();
```

运行结果如下。这里显示了模型的 meta 信息，在此还可以看到各个权重参数的取值：

```
----- model meta info -----
{hasInterception: true, model name: Linear Regression, num feature: 2}
----- model weight info -----
| intercept | x | x2 |
|-----|-----|-----|
| 122194787.2123 | -121612.28370990 | 30.25813853 |
```

最后，我们将模型信息（ModelInfo）组件、批式训练组件和 PipelineStage 的对照关系整理为表格，详见表 2-2。

表 2-2 模型信息组件、批式训练组件和 PipelineStage 对照表

	批式训练组件	PipelineStage	模型信息组件
分 类 算 法	LinearSvmTrainBatchOp	LinearSvm	LinearSvmModelInfoBatchOp
	LogisticRegressionTrainBatchOp	LogisticRegression	LogisticRegressionModelInfoBatchOp
	SoftmaxTrainBatchOp	Softmax	SoftmaxModelInfoBatchOp
	NaiveBayesTrainBatchOp	NaiveBayes	NaiveBayesModelInfoBatchOp

续表

	批式训练组件	PipelineStage	模型信息组件
分类算法	NaiveBayesTextTrainBatchOp	NaiveBayesTextClassifier	NaiveBayesModelInfoBatchOp
	FmClassifierTrainBatchOp	FmClassifier	FmClassifierModelInfoBatchOp
	DecisionTreeTrainBatchOp	DecisionTreeClassifier	DecisionTreeModelInfoBatchOp
	GbdtTrainBatchOp	GbdtClassifier	GbdtModelInfoBatchOp
	RandomForestTrainBatchOp	RandomForestClassifier	RandomForestModelInfoBatchOp
	C45TrainBatchOp	C45	C45ModelInfoBatchOp
	CartTrainBatchOp	Cart	CartModelInfoBatchOp
回归算法	Id3TrainBatchOp	Id3	Id3ModelInfoBatchOp
	CartRegTrainBatchOp	CartReg	CartRegModelInfoBatchOp
	DecisionTreeRegTrainBatchOp	DecisionTreeRegressor	DecisionTreeRegModelInfoBatchOp
	RandomForestRegTrainBatchOp	RandomForestRegressor	RandomForestRegModelInfoBatchOp
	GbdtRegTrainBatchOp	GbdtRegressor	GbdtRegModelInfoBatchOp
	LassoRegTrainBatchOp	LassoRegression	LassoRegModelInfoBatchOp
	RidgeRegTrainBatchOp	RidgeRegression	RidgeRegModelInfoBatchOp
聚类算法	LinearRegTrainBatchOp	LinearRegression	LinearRegModelInfoBatchOp
	AftSurvivalRegTrainBatchOp	AftSurvivalRegression	AftSurvivalRegModelInfoBatchOp
	FmRegressorTrainBatchOp	FmRegressor	FmRegressorModelInfoBatchOp
	KMeansTrainBatchOp	KMeans	KMeansModelInfoBatchOp
	GmmTrainBatchOp	GaussianMixture	GmmModelInfoBatchOp
	BisectingKMeansTrainBatchOp	BisectingKMeans	BisectingKMeansModelInfoBatchOp
	LdaTrainBatchOp	Lda	LdaModelInfoBatchOp
数据处理	ImputerTrainBatchOp	Imputer	ImputerModelInfoBatchOp
	MaxAbsScalerTrainBatchOp	MaxAbsScaler	MaxAbsScalerModelInfoBatchOp
	MinMaxScalerTrainBatchOp	MinMaxScaler	MinMaxScalerModelInfoBatchOp
	StandardScalerTrainBatchOp	StandardScaler	StandardScalerModelInfoBatchOp
	VectorImputerTrainBatchOp	VectorImputer	VectorImputerModelInfoBatchOp
	VectorMaxAbsScalerTrainBatchOp	VectorMaxAbsScaler	VectorMaxAbsScalerModelInfoBatchOp
	VectorMinMaxScalerTrainBatchOp	VectorMinMaxScaler	VectorMinMaxScalerModelInfoBatchOp
	VectorStandardScalerTrainBatchOp	VectorStandardScaler	VectorStandardScalerModelInfoBatchOp
	EqualWidthDiscretizerTrainBatchOp	EqualWidthDiscretizer	EqualWidthDiscretizerModelInfoBatchOp
	OneHotTrainBatchOp	OneHotEncoder	OneHotModelInfoBatchOp
	PcaTrainBatchOp	PCA	PcaModelInfoBatchOp
	QuantileDiscretizerTrainBatchOp	QuantileDiscretizer	QuantileDiscretizerModelInfoBatchOp

2.7 文件系统与数据库

我们处理的数据主要存储在文件系统和数据库中。本书的第3章将专门介绍文件系统，第4章会详细介绍与数据库、数据表相关的操作。Alink采用插件的方式来管理各种文件系统、数据库以及各个版本所需的函数库。本节还会专门介绍如何利用Alink的插件（Plugin）工具自动下载相关的函数库。

图 2-12 为 Alink 文件系统（File System）与数据库（Catalog）的架构图，具体说明如下：

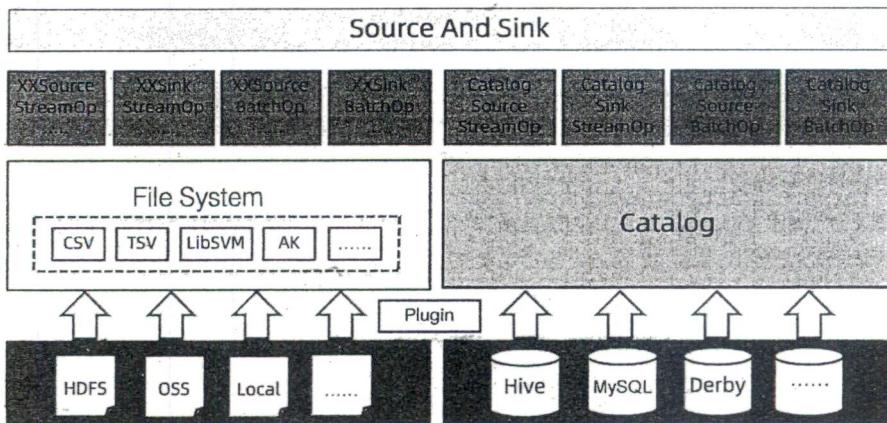


图 2-12

- Alink 通过定义统一的 File System，规范常用的文件操作；Alink 定义统一的 Catalog，抽象常用的数据库（Database）和表（Table）操作。
- 我们在实际应用中，不会同时用到所有文件系统和数据库。另外，我们还要考虑版本的问题。**使用插件机制**，方便大家选择适合自己的方式与版本。
- 在数据源和导出方面，数据文件和数据库均利用统一定义的 File System 和 Catalog 来统一操作流程，避免逐个定义带来大量的类似接口。
- 数据库方面的操作比较直接。确定好 Catalog 以及数据表的路径信息（数据库→表）。此外，还要统一定义数据表的批式/流式数据源组件（CatalogSourceBatchOp/CatalogSourceStreamOp），以及定义数据表的批式/流式导出组件（CatalogSinkBatchOp/CatalogSinkStreamOp）。
- 对于**数据**文件，需要考虑两个方面：文件的格式和所在文件系统的路径。由于每种文件格式所需的参数不同，因此 Alink 按文件格式定义相应的数据源和导出组件。对于 XX

文件格式，定义批式/流式数据源组件（`XXSourceBatchOp/ XXSourceStreamOp`），同样定义数据表的批式/流式导出组件（`XXSinkBatchOp/ XXSinkStreamOp`）。

- 各组件的相同之处是，文件路径参数的设置方式相同；基本形式是使用 `FilePath` 类进行设置，包括两类信息：所在文件系统的信息和文件路径的信息。另外，对于本地路径，可以直接设置路径字符串。

插件下载

Alink 能够支持不同第三方库（例如 OSS、Hive、Derby、MySQL 等）的不同版本（例如 Hive 的 2.3.4 版本、2.3.6 版本等）。为了更好地管理插件（外部的第三方库），我们提供了插件下载器（`PluginDownloader`）来管理不同插件的多个版本。

插件下载器封装了插件的常见功能，如下所示：

- 枚举仓库中的所有插件。
- 枚举某个插件的所有版本。
- 下载某个插件的特定版本/默认版本。
- 下载所有插件的默认版本。
- 升级所有的插件。

在 Java 代码中可以这样使用插件下载器：

```
// 设置插件下载的位置。当路径不存在时，会自行创建路径
AlinkGlobalConfiguration.setPluginDir("/Users/xxx/alink_plugins");

// 获得 Alink 插件下载器
PluginDownloader pluginDownloader = AlinkGlobalConfiguration.getPluginDownloader();

// 从远程加载插件的配置项
pluginDownloader.loadConfig();

// 展示所有可用的插件名称
List<String> plugins = pluginDownloader.listAvailablePlugins();
// 输出结果: [oss, hive, derby, mysql, hadoop, sqlite]

// 显示第 0 个插件的所有版本
String pluginName = plugins.get(0); // oss
List<String> availableVersions = pluginDownloader.listAvailablePluginVersions(pluginName);
// 输出结果: [3.4.1]

// 下载某个插件的特定版本
String pluginVersion = availableVersions.get(0);
pluginDownloader.downloadPlugin(pluginName, pluginVersion);
// 运行结束后，插件会被下载到 "/Users/xxx/alink_plugins/" 目录中
```

```
// 下载某个插件的默认版本
pluginDownloader.downloadPlugin(pluginName);
// 运行结束后，插件会被下载到 "/Users/xxx/alink_plugins/" 目录中

// 下载配置文件中所有插件的默认版本
pluginDownloader.downloadAll();

// 插件升级
// 在升级的过程中，会先对旧的插件进行备份，备份文件名称的后缀为.old。等到插件更新完毕后，
// 会统一删除旧的插件包
// 若插件更新中断，则用户可以从.old 文件中恢复旧版插件
pluginDownloader.upgrade();
```

2.8 Schema String

Alink 在进行表数据的读取和转换时，有时需要显式声明数据表的列名和列类型信息，即 Schema 信息。Schema String 就是将此信息使用字符串的方式进行描述的，这样便于将该信息作为 Java 函数或者 Python 函数的参数输入。

其定义格式如下：

```
colname coltype[, colname2 coltype2[, ...]]
```

例如，“f0 string, f1 bigint, f2 double”，表明数据表共有 3 列，名称分别为 f0、f1 和 f2；其对应的类型分别为字符串类型、长整型和双精度浮点型。熟悉 SQL 的读者会发现，该格式与 CREATE TABLE 的数据列名称和类型的定义格式相同。

关于各种列类型的写法，可以参照 Flink Type 与 Type String 的对应表（见表 2-3）。注意：为了适应不同用户的习惯，同一个 Flink Type 可能对应着多种 Type String 的写法。

表 2-3 Flink Type 与 Type String 的对应表

Flink Type	Type String
Types.STRING	string 或者 varchar
Types.BOOLEAN	boolean
Types.BYTE	byte 或者 tinyint
Types.SHORT	short 或者 smallint
Types.INT	int
Types.LONG	long 或者 bigint

续表

Flink Type	Type String
Types.FLOAT	float
Types.DOUBLE	double
Types.SQL_DATE	sql_date 或者 date
Types.SQL_TIMESTAMP	sql_timestamp 或者 timestamp
Types.SQL_TIME	sql_time 或者 time



文件系统与数据文件

基于 Flink 的 FileSystem 类，Alink 进一步定义和实现了常用的文件系统，比如，本地文件系统、Hadoop 分布式文件系统、阿里云 OSS（Object Storage Service）文件系统。本章将先介绍各文件系统的通用操作，随后通过具体的示例，详细介绍常用的 LocalFileSystem、HadoopFileSystem、OssFileSystem 等。

我们使用的数据常常以数据文件的形式存在，而借助统一的文件系统接口，我们就可以方便地读/写各文件系统的数据文件。但要知道具体的文件内容，还需要了解数据文件的格式，比如，CSV、TSV、LibSVM 等都是机器学习领域常用的数据文件格式。本章的后半部分将介绍 Alink 针对常用数据文件格式提供的组件，以帮助用户轻松地在批式/流式场景中读入和写出数据。

3.1 文件系统简介

Alink 的 FileSystem 类是文件系统的抽象父类，定义了一些文件系统共有的功能。文件操作如表 3-1 所示。

表 3-1 文件操作

函数名	返回类型	说明
getFileStatus(Path f)	FileStatus	返回指定路径文件的元数据
listStatus(Path f)	FileStatus[]	列出给定路径文件夹下的所有子文件夹和子文件的元数据

续表

函数名	返回类型	说明
exists(Path f)	boolean	检查文件或目录是否存在
rename(Path src, Path dst)	boolean	重命名文件或文件夹
delete(Path f, boolean recursive)	boolean	删除文件或文件夹。如果 Path 对应的是文件，则无论 recursive 为 true 或者 false，该文件都会被直接删除；如果 Path 对应的是文件夹，则只有在 recursive 为 true 的情况下该文件夹才会被删除，否则会抛出异常
mkdirs(Path f)	boolean	建立文件夹（如果父文件夹不存在，则建立父文件夹）
create(Path f, boolean overwrite)	FSDataOutputStream	创建指定 Path 的一个文件，返回用于写入数据的输出流，参数 overwrite 控制是否强制覆盖已有的文件
open(Path f)	FSDaInputStream	返回读取指定文件的数据输入流

其中，FileStatus 类型封装了文件或目录的元数据，包括文件长度、块大小、文件访问、文件修改时间、是否为文件夹等信息。通过 create 方法获取输出流，以及通过 open 方法获取输入流，可以实现在同一文件系统内部进行文件的 I/O 操作，也可以做到跨文件系统进行文件的 I/O 操作。

下面将通过示例详细介绍本地文件系统、Hadoop 分布式文件系统和阿里云 OSS 文件系统。

3.1.1 本地文件系统

Alink 定义了 LocalFileSystem，用来操作本地文件系统。想必大家对本地文件系统的操作比较熟悉。这里直接通过 Java 示例来演示文件接口的用法。

先定义一个操作文件夹，后面的操作都会在该文件夹下进行：

```
static final String LOCAL_DIR = Utils.ROOT_DIR + "filesys" + File.separator;
```

下面的代码演示了如何查看本地文件系统的一些信息，其中 getKind 方法返回的是文件系统的类型：

```
LocalFileSystem local = new LocalFileSystem();
System.out.println(local.getHomeDirectory());
```

```
System.out.println(local.getKind());
System.out.println(local.getWorkingDirectory());
```

运行结果如下：

```
file:/Users/yangxu/
FILE_SYSTEM
file:/Users/yangxu/Code/alink/
```

实现一个常用操作：判断目标文件夹是否存在。若该文件夹不存在，则新建一个文件夹，并显示目标文件夹下子文件夹和子文件的信息。具体代码如下：

```
if (!local.exists(LOCAL_DIR)) {
    local.mkdirs(LOCAL_DIR);
}

for (FileStatus status : local.listStatus(LOCAL_DIR)) {
    System.out.println(status.getPath().toUri()
        + " \t" + status.getLen()
        + " \t" + new Date(status.getModificationTime()))
}
}
```

接下来演示与数据读/写相关的操作，设定目标文件为"hello.txt"，可以通过创建文件输出流的方式，将字符串"Hello Alink!"写入其中。随后显示该文件的状态。具体代码如下：

```
String path = LOCAL_DIR + "hello.txt";

OutputStream outputStream = local.create(path, WriteMode.OVERWRITE);
outputStream.write("Hello Alink!".getBytes());
outputStream.close();

FileStatus status = local.getFileStatus(path);
System.out.println(status);
System.out.println(status.getLen());
System.out.println(new Date(status.getModificationTime()));
```

注意：在创建文件输出流的时候选择了“WriteMode.OVERWRITE”，即覆盖写。如果目标文件存在，则清除其原有数据，之后写入该次执行的结果数据。

获得的输出结果如下：

```
LocalFileStatus{file=/Users/yangxu/alink/data/temp/hello.txt, path=file:/Users/yangxu/alink/data/temp/hello.txt}
12
Thu Jul 23 11:33:20 CST 2020
```

最后，我们使用以下文件输入流，将文件内容读取到一个字符串：

```
InputStream inputStream = local.open(path);
String readString = IOUtils.toString(inputStream);
```

```
System.out.println(readString);
```

其输出结果为“Hello Alink!”。

3.1.2 Hadoop文件系统

Alink 定义了 HadoopFileSystem，用来操作 Hadoop 分布式文件系统。

首先看如何构建 HadoopFileSystem 实例。构建该实例需要 HDFS 服务的 IP 地址和 Port（端口），且使用 URI 方式给出。

```
static final String HDFS_URI = "hdfs://10.*.*.*:9000/";
```

注意：这里用“*”略去了具体的数字，读者在尝试此代码时，可以根据自己 HDFS 服务的 IP 地址和 Port（端口）进行替换。

使用 HadoopFileSystem 的构建实例，具体代码如下：

```
HadoopFileSystem hdfs = new HadoopFileSystem(HDFS_URI);
```

随后，调用 getKind 方法；并判断目标文件夹是否存在。若该文件夹不存在，则新建一个文件夹。

```
System.out.println(hdfs.getKind());
if (!hdfs.exists(hdfsDir)) {
    hdfs.mkdirs(hdfsDir);
}
```

设定目标文件为“hello.txt”，并判断该文件是否存在。若该文件存在，则删除该文件；然后通过创建文件输出流的方式，将字符串“Hello Alink!”写入其中。具体代码如下：

```
String path = hdfsDir + "hello.txt";
if (hdfs.exists(path)) {
    hdfs.delete(path, true);
}
OutputStream outputStream = hdfs.create(path, WriteMode.NO_OVERWRITE);
outputStream.write("Hello Alink!".getBytes());
outputStream.close();
```

注意：因为判断和删除操作，保证了目标文件不存在，所以在创建文件输出流的时候，可以使用“WriteMode.NO_OVERWRITE”。

最后，我们使用以下文件输入流，将文件内容读取到一个字符串：

```
InputStream inputStream = hdfs.open(path);
String readString = IOUtils.toString(inputStream);
System.out.println(readString);
```

其输出结果为“Hello Alink!”，说明整套写入、读取流程运行正常。

有了文件系统的基本功能，很容易包装出新的方法。比如下面的函数，通过输入/输出流，完成了文件的复制操作：

```
static void copy(InputStream in, OutputStream out) throws IOException {
    byte[] buffer = new byte[1024 * 1024];
    int len = in.read(buffer);
    while (len != -1) {
        out.write(buffer, 0, len);
        len = in.read(buffer);
    }
    in.close();
    out.close();
}
```

我们使用这个函数，就可完成 Hadoop 文件系统和本地文件系统的文件复制操作，并显示文件夹中的内容。具体代码如下：

```
LocalFileSystem local = new LocalFileSystem();
HadoopFileSystem hdfs = new HadoopFileSystem(HDFS_URI);

copy(
    hdfs.open(HDFS_URI + "user/yangxu/alink/data/temp/hello.txt"),
    local.create(LOCAL_DIR + "hello_1.txt", WriteMode.OVERWRITE)
);

copy(
    local.open(LOCAL_DIR + "hello_1.txt"),
    hdfs.create(HDFS_URI + "user/yangxu/alink/data/temp/hello_2.txt", WriteMode.OVERWRITE)
);

for (FileStatus status : hdfs.listStatus(HDFS_URI + "user/yangxu/alink/data/temp/")) {
    System.out.println(status.getPath().toUri()
        + " \t" + status.getLen()
        + " \t" + new Date(status.getModificationTime()))
}
}
```

运行结果如下：

```
hdfs://10.*.*:9000/user/yangxu/alink/data/temp/hello.txt      12    Thu Jul 23 13:24:45 CST 2020
hdfs://10.*.*:9000/user/yangxu/alink/data/temp/hello_2.txt    12    Thu Jul 23 13:24:46 CST 2020
```

现在我们就可以使用 Hadoop Web UI，查看目标文件夹下的数据情况了。如图 3-1 所示，可以看到新增了两个文件：hello.txt 和 hello_2.txt。

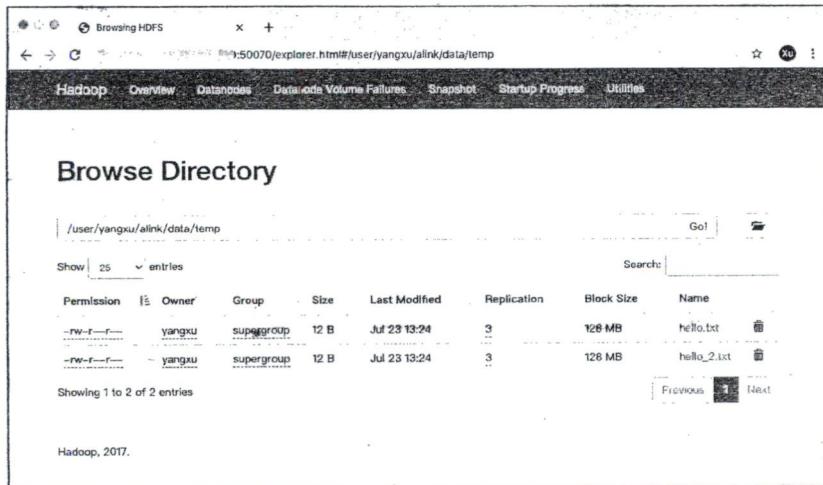


图 3-1

3.1.3 阿里云OSS文件系统

Alink 定义了 OssFileSystem，用来操作阿里云 OSS 文件系统。

首先看如何构建 OssFileSystem 实例，代码如下。构建该实例需要 4 个参数：OSS-END-POINT、OSS-BUCKET-NAME、OSS-ACCESS-ID、OSS-ACCESS-KEY。OSS-ACCESS-ID、OSS-ACCESS-KEY 相当于用户名和密码；OSS-END-POINT 为所要使用的 OSS 服务地址。用户可以设置若干个 Bucket，每个 Bucket 都可以被看作一个根目录，在使用 OSS 时需要选定一个 OSS-BUCKET-NAME 进行。在本示例中，选定的 OSS-BUCKET-NAME 名称为“yangxu-bucket”。

```
OssFileSystem oss =
new OssFileSystem(
    OSS_END_POINT,
    OSS_BUCKET_NAME,
    OSS_ACCESS_ID,
    OSS_ACCESS_KEY
);
```

调用 getKind 方法，代码如下，得到的结果为“OBJECT_STORE”：

```
System.out.println(oss.getKind());
```

OSS URI 都是以“oss://” + Bucket 开头的，设置 OSS URI 的前缀如下：

```
static final String OSS_PREFIX_URI = "oss://" + OSS_BUCKET_NAME + "/";
```

定义目标文件夹，判断目标文件夹是否存在。若该文件夹不存在，则新建一个文件夹。具体代码如下：

```
final String ossDir = OSS_PREFIX_URI + "alink/data/temp/";
if (!oss.exists(new Path(ossDir))) {
    oss.mkdirs(new Path(ossDir));
}
```

设定目标文件为"hello.txt"，并判断该文件是否存在。若该文件存在，则删除该文件；然后通过创建文件输出流的方式，将字符串“Hello Alink!”写入其中。相关代码如下：

```
String path = ossDir + "hello.txt";
OutputStream outputStream = oss.create(path, WriteMode.OVERWRITE);
outputStream.write("Hello Alink!".getBytes());
outputStream.close();
```

最后，我们使用以下文件输入流，将文件内容读取到一个字符串：

```
InputStream inputStream = oss.open(path);
String readString = IOUtils.toString(inputStream);
System.out.println(readString);
```

其输出结果为“Hello Alink!”，说明整套写入、读取流程运行正常。

下面演示文件复制操作的例子。使用前面定义的文件复制函数 copy(InputStream in, OutputStream out)来进行 OSS 文件系统和本地文件系统之间的文件复制操作，并显示文件夹内的内容。代码如下：

```
LocalFileSystem local = new LocalFileSystem();

OssFileSystem oss =
new OssFileSystem(
    OSS_END_POINT,
    OSS_BUCKET_NAME,
    OSS_ACCESS_ID,
    OSS_ACCESS_KEY
);

copy(
    oss.open(OSS_PREFIX_URI + "alink/data/temp/hello.txt"),
    local.create(LOCAL_DIR + "hello_1.txt", WriteMode.OVERWRITE)
);

copy(
    local.open(LOCAL_DIR + "hello_1.txt"),
    oss.create(OSS_PREFIX_URI + "alink/data/temp/hello_2.txt", WriteMode.OVERWRITE)
);
```

```

);
for (FileStatus status : oss.listStatus(new Path(OSS_PREFIX_URI + "alink/data/temp/"))) {
    System.out.println(status.getPath().toUri()
        + " \t" + status.getLen()
        + " \t" + new Date(status.getModificationTime()))
}
}

```

运行结果如下：

```

oss://yangxu-bucket/alink/data/temp/hello.txt 12 Thu Jul 23 14:30:00 CST 2020
oss://yangxu-bucket/alink/data/temp/hello_2.txt 12 Thu Jul 23 14:30:01 CST 2020

```

现在，我们就可以使用 OSS 的浏览器软件来查看目标文件夹下的数据情况了。如图 3-2 所示，可以看到新增了两个文件：hello.txt 和 hello_2.txt。



图 3-2

打开文件 hello_2.txt，如图 3-3 所示，这里显示的内容与预期结果一致。



图 3-3

3.2 数据文件的读入与导出

基于统一的文件系统，数据文件的访问更加便捷。在本节中，我们将深入介绍如何从各种文件系统中获取和导出机器学习计算所需的结构化数据（数据表）。

使用数据文件会涉及以下四点：

- 使用场景：批式处理和流式处理。
- 打开方式：从文件中读取数据（用 Source 标记），将数据导出到文件中（用 Sink 标记）。
- 文件格式：常见的 CSV、TSV、LibSVM 等格式；以及 Alink 专门定义的 AK 格式。
- 文件存储路径及所在文件系统的读/写权限。

上面提到的四点，彼此间是相互独立的，组合起来就对应一个具体的功能。Alink 将前面三点体现在了组件名称上，最后一点是通过参数输入的。

先介绍与 Alink 数据文件相关的组件名称，由三部分构成：

【文件格式】+【打开方式】+【使用场景】

比如，`CsvSourceBatchOp` 就是以批式处理的形式，从数据文件中读取 CSV 格式数据的。文件数据读入与导出组件的构成如表 3-2 所示。

表 3-2 数据文件读入与导出组件的构成

	可选值
文件格式	CSV、TSV、LibSVM、Text、AK 等
打开方式	Source（从文件中读取数据）、Sink（将数据导出到文件中）
使用场景	BatchOp（批式处理）、StreamOp（流式处理）

类 `FilePath`，包含文件存储路径及所在文件系统的读/写权限信息。`FilePath` 的基本构造函数如下：

```
public FilePath(String path, BaseFileSystem fileSystem)
```

文件存储路径可以用字符串表示，本地文件可以直接写为“`/Users/yangxu/iris.csv`”；也可以使用 URI 方式，比如“`hdfs://namenode:50070/data/iris.csv`”就对应着分布式文件系统 HDFS 中的文件；而阿里云 OSS 中的文件路径可以写为“`oss://yangxu-bucket/data/iris.csv`”。执行阿里云 OSS 的文件访问，除了需要文件路径，还需要 `OSS_ACCESS_ID`、`OSS_ACCESS_KEY` 等信息，这些内容是在 `OssFileSystem` 的实例中，从第二个参数传给 `FilePath` 的构造函数的。

对于本地文件系统和 Hadoop 文件系统，由于无须指定权限等信息，并且可以直接从字符串判断出其属于哪种文件系统，因此可以使用更简单的 `FilePath` 构造函数：

```
public FilePath(String path)
```

与 Alink 数据文件相关的组件都有如下参数设置接口，以便用来设置 `FilePath`；也就是设置文件存储路径及所在文件系统的读/写权限信息：

```
setFilePath(String value)
```

```
setFilePath(FilePath filePath)
```

综上，组件名称和参数设置有明显的规律性，便于用户记忆和使用。

注意，有的读者可能会有疑问：为什么不把文件格式也作为参数输入呢？这样组件的个数会减少很多。这是因为每种格式都有自己独特的参数，整合为一个组件时，该组件中要包含所有的参数，用户在使用该组件时就需要区分具体要填入哪些参数，这样容易有困扰。

3.2.1 CSV格式

CSV (Comma-Separated Values) 格式的最基本含义是，每条记录以逗号分隔各字段值，记录之间使用换行符分隔，即每条记录为一行。在实际使用中，CSV 格式被赋予了更广泛的含义：字段间的分隔符可以使用其他分隔符替换，各记录间的分隔符也可以设置，第一行可能记录的是各字段的名称。CSV 格式在机器学习领域中被广泛使用，比如，UCI 数据集、Kaggle 中的数据集，很多都是 CSV 格式的。

1. 批式/流式数据源

CSV 格式的批式和流式数据源 (Source) 组件的参数是一样的，如表 3-3 所示。

表 3-3 CSV 格式数据源组件的参数

参数名称	参数说明
filePath	【必填】CSV 格式文件的路径信息
schemaStr	【必填】数据各字段的名称和类型。格式为“name1 type1, name2 type2, name3 type3”，名称和类型间以空格分隔，各组名称类型对之间使用逗号分隔（详见 2.8 节的内容）
fieldDelimiter	【可选】每行数据各字段的分隔符，默认为逗号，即","
rowDelimiter	【可选】行数据分隔符，默认为换行符，即"\n"
quoteChar	【可选】引号字符，默认为双引号，即"
skipBlankLine	【可选】是否忽略空行，默认值为 true
ignoreFirstLine	【可选】是否忽略第一行数据，默认值为 false

我们先下载一个 CSV 文件用作后面的测试数据。将数据文件(参见链接 3-1)下载到本地，使用文本编辑器打开该文件。如图 3-4 所示，每行为一条数据，每条数据包括四个数值字段和一个字符串字段，各字段间使用逗号分隔。

iris.csv	
6.5	2.8,4.6,1.5,Iris-versicolor
6.1	3.0,4.9,1.8,Iris-virginica
7.3	2.9,6.3,1.8,Iris-virginica
5.7	2.8,4.5,1.3,Iris-versicolor
6.4	2.8,5.6,2.1,Iris-virginica
6.7	2.5,5.8,1.8,Iris-virginica
6.3	3.3,4.7,1.6,Iris-versicolor
7.2	3.6,6.1,2.5,Iris-virginica
7.2	3.0,5.8,1.6,Iris-virginica
4.9	2.4,3.3,1.0,Iris-versicolor
7.4	2.8,6.1,1.9,Iris-virginica
6.5	3.2,5.1,2.0,Iris-virginica
6.6	2.9,4.6,1.3,Iris-versicolor
7.9	3.8,6.4,2.0,Iris-virginica
5.2	2.7,3.9,1.4,Iris-versicolor
6.4	2.7,5.3,1.9,Iris-virginica
6.8	3.0,5.5,2.1,Iris-virginica
5.7	2.5,5.0,2.0,Iris-virginica
7.7	3.0,6.1,2.3,Iris-virginica
5.8	2.7,3.9,1.2,Iris-versicolor
5.6	2.9,3.6,1.3,Iris-versicolor

图 3-4

首先使用 CsvSourceBatchOp 读取本地数据，并取前 5 条数据打印出来。具体代码如下：

```
CsvSourceBatchOp source_local = new CsvSourceBatchOp()
.setFilePath(LOCAL_DIR + "iris.data")
.setSchemaStr("sepal_length double, sepal_width double, "
+ "petal_length double, petal_width double, category string");

source_local.firstN(5).print();
```

注意： CsvSourceBatchOp 组件参数 filePath 和 schemaStr 为必填的。filePath 为本地文件的存储路径 LOCAL_DIR + "iris.data"。schemaStr 为 iris 数据集的列名和类型信息，共有 5 个字段： sepal_length、sepal_width、petal_length、petal_width、category，其数据类型分别为 double、double、double、double、string。

与上面介绍的读取本地 CSV 数据相比，我们只需将数据的存储路径参数 filePath 赋值为 http 路径地址，即可直接读取网络数据：

```
CsvSourceBatchOp source_url = new CsvSourceBatchOp()
.setFilePath("http://archive.ics.uci.edu/ml/machine-learning-databases"
+ "/iris/iris.data")
.setSchemaStr("sepal_length double, sepal_width double, "
+ "petal_length double, petal_width double, category string");

source_url.firstN(5).print();
```

我们看到两次的结果相同，如下所示：

sepal_length	sepal_width	petal_length	petal_width	category
--------------	-------------	--------------	-------------	----------

```
5.1000|3.5000|1.4000|0.2000|Iris-setosa
4.9000|3.0000|1.4000|0.2000|Iris-setosa
4.7000|3.2000|1.3000|0.2000|Iris-setosa
4.6000|3.1000|1.5000|0.2000|Iris-setosa
5.0000|3.6000|1.4000|0.2000|Iris-setosa
```

我们再以流的方式读取数据，只需将组件名称换为 CsvSourceStreamOp，参数设置不动，就得到了流式数据源。流式数据无法指定前几条数据，但为了控制打印显示的数据量，我们可以对数据进行过滤，选取满足条件“sepal_length < 4.5”的数据，并打印输出，具体代码如下：

```
CsvSourceStreamOp source_stream = new CsvSourceStreamOp()
.setFilePath("http://archive.ics.uci.edu/ml/machine-learning-databases"
+ "/iris/iris.data")
.setSchemaStr("sepal_length double, sepal_width double, "
+ "petal_length double, petal_width double, category string");

source_stream.filter("sepal_length < 4.5").print();
StreamOperator.execute();
```

得到的结果如下：

sepal_length	sepal_width	petal_length	petal_width	category
4.4000	3.2000	1.3000	0.2000	Iris-setosa
4.4000	2.9000	1.4000	0.2000	Iris-setosa
4.4000	3.0000	1.3000	0.2000	Iris-setosa
4.3000	3.0000	1.1000	0.1000	Iris-setosa

接下来，我们尝试更复杂的例子。对于葡萄酒品质数据集（参见链接 3-2），我们将其下载到本地，可以看到其文件内容，如图 3-5 所示。注意：第一行的数据太长，右边的显示被截断了。

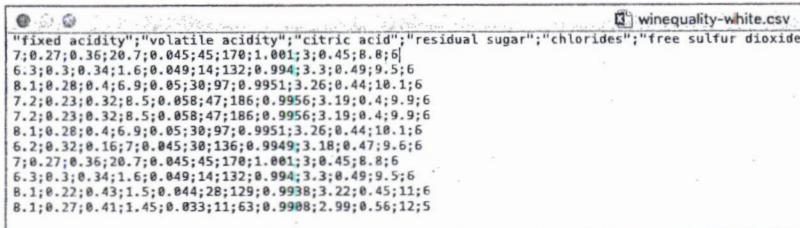


图 3-5

第一行为数据列名的说明，从第二行开始是数据，可以看到这些数据都是数值类型的，各个数值之间用分号“;”进行分隔。

我们可以通过将参数 ignoreFirstLine 设置为 true，略过第一行；并且可以将字段分隔符参数 fieldDelimiter 设置为分号“;”。另外，因为列名不能包含空格，所以由文件第一行转化而来的

列名需要进行相应的处理。在此，我们将列名写成驼峰形式；在每个列名后加上数据类型，这里的数据都是 double 类型的，所有的列名和类型构成了数据集的 schemaStr。具体的脚本如下：

```
CsvSourceBatchOp wine_url = new CsvSourceBatchOp()
    .setFilePath(LOCAL_DIR + "winequality-white.csv")
    .setSchemaStr("fixedAcidity double,volatileAcidity double,citricAcid double,"
        + "residualSugar double, chlorides double,freeSulfurDioxide double,"
        + "totalSulfurDioxide double,density double, pH double,"
        + "sulphates double,alcohol double,quality double")
    .setFieldDelimiter(";")
    .setIgnoreFirstLine(true);

wine_url.firstN(5).print();
```

数据打印的结果如表 3-4 所示。

表 3-4 数据打印的结果

fixedAcidity	volatileAcidity	citricAcid	residualSugar	chlorides	freeSulfurDioxide	totalSulfurDioxide	density	pH	sulphates	alcohol	quality
7.0000	0.2700	0.3600	20.7000	0.0450	45.0000	170.0000	1.0010	3.0000	0.4500	8.8000	6.0000
6.3000	0.3000	0.3400	1.6000	0.0490	14.0000	132.0000	0.9940	3.3000	0.4900	9.5000	6.0000
8.1000	0.2800	0.4000	6.9000	0.0500	30.0000	97.0000	0.9951	3.2600	0.4400	10.1000	6.0000
7.2000	0.2300	0.3200	8.5000	0.0580	47.0000	186.0000	0.9956	3.1900	0.4000	9.9000	6.0000
7.2000	0.2300	0.3200	8.5000	0.0580	47.0000	186.0000	0.9956	3.1900	0.4000	9.9000	6.0000

2. 批式/流式数据导出

CSV 格式的批式和流式数据导出（Sink）组件的参数是一样的，CSV 格式导出组件的参数如表 3-5 所示。

表 3-5 CSV 格式导出组件的参数

参数名称	参数说明
filePath	【必填】CSV 格式文件的路径信息
fieldDelimiter	【可选】每行数据各字段的分隔符，默认为逗号，即","
rowDelimiter	【可选】行数据分隔符，默认为换行符，即"\n"
quoteChar	【可选】引号字符，默认为双引号，即"
numFiles	【可选】保存文件的个数，默认值为 1
overwriteSink	【可选】是否覆盖写已有的数据。默认值为 false，即报错并退出

其中，最后两个参数是导出(Sink)操作所特有的。当参数numFiles大于1时，参数filePath对应的就是文件夹，在该文件夹中将数据写到numFiles个文件。如果目标数据文件(夹)不存在，则新建该数据表或文件(夹)，然后将其整体导出。如果目标数据文件(夹)存在，则需要考虑参数“overwriteSink”，默认值为false，即报错并退出。如果用户将其设为true，则会删除已存在的目标数据文件(夹)，并新建该数据表或文件(夹)，然后将其整体导出，即该导出操作有覆盖写的效果。

我们结合前面介绍的批式/流式数据源(Source)及文件系统，做个综合性的实验。

首先，定义三个目标文件路径，分别位于以下三个文件系统中：本地文件系统、Hadoop分布式文件系统、阿里云OSS文件系统。具体代码如下：

```
OssFileSystem oss =
    new OssFileSystem(
        OSS_END_POINT,
        OSS_BUCKET_NAME,
        OSS_ACCESS_ID,
        OSS_ACCESS_KEY
    );

FilePath[] filePaths = new FilePath[] {
    new FilePath(LOCAL_DIR + "iris.csv"),
    new FilePath(HDFS_URI + "user/yangxu/alink/data/temp/iris.csv"),
    new FilePath(OSS_PREFIX_URI + "alink/data/temp/iris.csv", oss)
};
```

随后，针对批式处理场景设计实验。从网络的HTTP数据源中读取iris数据，将其导出(Sink)到某个目标文件路径，然后读取该目标文件路径的数据，使用count方法统计其记录总条数，验证此新生成的数据文件是否正确(正确的总记录数为150条)。对三个不同文件系统的目标文件路径均执行此操作。具体代码如下：

```
for (FilePath filePath : filePaths) {
    new CsvSourceBatchOp()
        .setFilePath(IRIS_HTTP_URL)
        .setSchemaStr(IRIS_SCHEMA_STR)
        .link(
            new CsvSinkBatchOp()
                .setFilePath(filePath)
                .setOverwriteSink(true)
        );
    BatchOperator.execute();

    System.out.println(
        new CsvSourceBatchOp()
            .setFilePath(filePath)
            .setSchemaStr(IRIS_SCHEMA_STR)
```

```

    .count()
);
}

```

运行结果如下，均为 150，说明在这三个文件系统中的操作都正确完成：

```

150
150
150

```

最后，针对流式场景设计实验。从网络的 HTTP 数据源中流式读取 iris 数据，将其导出(Sink)到某个目标文件路径，然后读取该目标文件路径的数据。因为从该数据源读取的是流式数据，而流式数据无法直接使用 count 方法进行验证，所以这里就选取满足条件 "sepal_length < 4.5" 的数据，并打印输出，以此来进行验证。对三个不同文件系统的目标文件路径均执行此操作。代码如下：

```

for (FilePath filePath : filePaths) {
    new CsvSourceStreamOp()
        .setFilePath(IRIS_HTTP_URL)
        .setSchemaStr(IRIS_SCHEMA_STR)
        .link(
            new CsvSinkStreamOp()
                .setFilePath(filePath)
                .setOverwriteSink(true)
        );
    StreamOperator.execute();

    new CsvSourceStreamOp()
        .setFilePath(filePath)
        .setSchemaStr(IRIS_SCHEMA_STR)
        .filter("sepal_length < 4.5")
        .print();
    StreamOperator.execute();
}

```

三次运行结果如下。进行对比后可发现，其结果一致：

sepal_length	sepal_width	petal_length	petal_width	category
4.4000	2.9000	1.4000	0.2000	Iris-setosa
4.4000	3.2000	1.3000	0.2000	Iris-setosa
4.3000	3.0000	1.1000	0.1000	Iris-setosa
4.4000	3.0000	1.3000	0.2000	Iris-setosa

sepal_length	sepal_width	petal_length	petal_width	category
4.4000	3.0000	1.3000	0.2000	Iris-setosa
4.4000	2.9000	1.4000	0.2000	Iris-setosa
4.3000	3.0000	1.1000	0.1000	Iris-setosa
4.4000	3.2000	1.3000	0.2000	Iris-setosa

sepal_length	sepal_width	petal_length	petal_width	category
4.4000	2.9000	1.4000	0.2000	Iris-setosa
4.3000	3.0000	1.1000	0.1000	Iris-setosa
4.4000	3.2000	1.3000	0.2000	Iris-setosa
4.4000	3.0000	1.3000	0.2000	Iris-setosa

3.2.2 TSV、LibSVM、Text格式

我们先分别介绍这三种格式的定义，随后通过具体的示例进行讲解。

1. TSV格式

TSV (Tab-Separated Values) 为“制表符分隔值”格式，其中的每条记录以制表符（对应键盘上的 Tab 键，字符串表示为“\t”）来分隔各字段值，而记录之间使用换行符分隔，即每条记录为一行。也可以使用 CSV 格式组件执行操作，只要将字段分隔符 fieldDelimiter 设置为“\t”即可。为了方便用户使用，Alink 提供了单独的 TSV 格式组件。

2. LibSVM格式

LibSVM 格式就是 LibSVM（参见[链接 3-3](#)）使用的数据格式，是机器学习领域中比较常见的一种形式。其格式定义如下：

```
<label> <index1>:<value1> <index2>:<value2> ...
```

第一项<label>是训练集的目标值。对于分类问题，用整数作为类别的标识（对于 2 分类，多用{0,1}或者{-1,1}表示；对于多分类问题，常用连续的整数，比如用{1,2,3}表示 3 分类的各个类别）；对于回归问题，目标值是实数。其后由若干索引<index>和数值<value>对（索引和数值间以冒号“:”作为分隔符）构成，各索引/数值对间以空格作为分隔符。索引<index>是从 1 开始的整数，索引可以是不连续的整数；数值<value>为实数。

下面是几条符合 LibSVM 格式的数据：

```
1 1:-0.555556 2:0.5 3:-0.79661 4:-0.916667
1 1:-0.833333 3:-0.864407 4:-0.916667
1 1:-0.444444 2:0.416667 3:-0.830508 4:-0.916667
1 1:-0.611111 2:0.0833333 3:-0.864407 4:-0.916667
2 1:0.5 3:0.254237 4:0.0833333
2 1:0.166667 3:0.186441 4:0.166667
2 1:0.444444 2:-0.0833334 3:0.322034 4:0.166667
```

注意这条数据：

```
2 1:0.5 3:0.254237 4:0.0833333
```

没有索引值为 2 的项，表明第 2 个特征值为 0。

3. Text 格式

Text 格式是 Alink 从使用方式的角度命名的。在处理一些文本数据，或者我们暂时无须区分各字段值，而可以将整条记录看作一个字符串时，每条记录只有一个字符串类型字段，记录之间使用换行符分隔。

下面运行 TSV 示例。MovieLens 数据集中就有 TSV 格式的，相应的链接地址为链接 3-4。使用浏览器打开该链接，如图 3-6 所示，这里有 4 个字段，并以制表符分隔各字段值。

user_id	item_id	rating	ts
196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923
166	346	1	886397596
298	474	4	884182806
115	265	2	881171488
253	465	5	891628467
305	451	3	886324817
6	86	3	883603013
62	257	2	879372434
286	1014	5	879781125
200	222	5	876042340
210	40	3	891035994
224	29	3	888104457
303	785	3	879485318
122	387	5	879270459
194	274	2	879539794
291	1042	4	874834944
234	1184	2	892079237
119	392	4	886176814

图 3-6

使用 TsvSourceBatchOp 读取该数据的代码如下。除了数据路径，还需要 Schema 信息。

```
new TsvSourceBatchOp()
    .setFilePath(LOCAL_DIR + "u.data")
    .setSchemaStr("user_id long, item_id long, rating float, ts long")
    .firstN(5)
    .print();
```

运行结果如下：

user_id	item_id	rating	ts
196	242	3.0000	881250949
186	302	3.0000	891717742
22	377	1.0000	878887116
244	51	2.0000	880606923
166	346	1.0000	886397596

在 LibSVM 网站提供了该格式的数据（参见链接 3-5）。使用浏览器打开该链接，如图 3-7 所示。

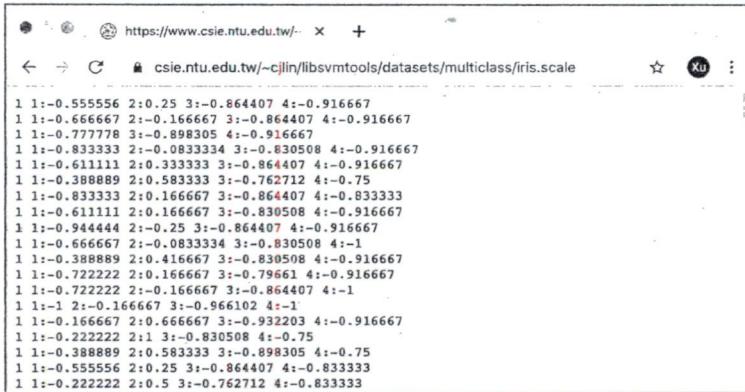


图 3-7

首先，忽略其格式定义，将其当作字符串来读取。使用 TextSourceBatchOp 组件，可以指定输出的字符串列的列名，默认值为“text”。

```

new TextSourceBatchOp()
    .setFilePath(LOCAL_DIR + "iris.scale")
    .firstN(5)
    .print();

```

运行结果如下：

```

text
_____
1 1:-0.555556 2:0.25 3:-0.864407 4:-0.916667
1 1:-0.666667 2:-0.166667 3:-0.864407 4:-0.916667
1 1:-0.777778 3:-0.898305 4:-0.916667
1 1:-0.833333 2:-0.0833334 3:-0.830508 4:-0.916667
1 1:-0.611111 2:0.333333 3:-0.864407 4:-0.916667

```

然后，我们使用 LibSvmBatchOp 进行读取，得到两个数据列：标签列（列名被自动命名为 label）和特征列（列名被自动命名为 features）。其中，特征列为向量格式，我们可以接 VectorNormalizeBatchOp（向量正则化）组件，使每个向量的 2-范数为 1。

```

new LibSvmSourceBatchOp()
    .setFilePath(LOCAL_DIR + "iris.scale")
    .firstN(5)
    .lazyPrint(5, "< read by LibSvmSourceBatchOp >")
    .link(

```

```

new VectorNormalizeBatchOp()
.setSelectedCol("features")
)
.print();

```

运行结果如下：

```

< read by LibSvmSourceBatchOp >
label|features
-----|-----
1.0000|0:-0.555556 1:0.25 2:-0.864407 3:-0.916667
1.0000|0:-0.666667 1:-0.166667 2:-0.864407 3:-0.916667
1.0000|0:-0.777778 2:-0.898305 3:-0.916667
1.0000|0:-0.833333 1:-0.0833334 2:-0.830508 3:-0.916667
1.0000|0:-0.611111 1:0.333333 2:-0.864407 3:-0.916667
label|features
-----|-----
1.0000|0:-0.3969654545518278 1:0.17863431164087318 2:-0.617650997690209 3:-0.6549927141956171
1.0000|0:-0.4645226632894226 1:-0.11613084001826729 2:-0.6023046615566992 3:-0.6387185749250003
1.0000|0:-0.5182689351202653 2:-0.5985815692436788 3:-0.6108170068449973
1.0000|0:-0.5578646805775191 1:-0.05578653500154036 2:-0.5559735185538965 3:-0.6136516172417902
1.0000|0:-0.42454181112608214 1:0.23156807114925168 2:-0.6005077855415192 3:-0.6368130640415773

```

从文件中读取的数据，标签列的类型为 double（因为 LibSVM 格式的数据也可以进行回归计算，标签值可能为浮点值），特征向量的索引是从 0 开始的（Alink 中的稀疏向量索引值都是从 0 开始的）；而在原始数据中，按 LibSVM 的定义，索引值是从 1 开始的。LibSvmBatchOp 组件会在读取和导出时，自动进行索引值的转换。

3.2.3 AK格式

AK 格式是专门为 Alink 定义的数据格式，在 AK 格式的数据文件中包含了各数据列名称和类型等元数据，并对数据内容进行了压缩。

AK 格式的批式和流式数据源（Source）组件的参数是一样的，都只有一个参数，即 AK 格式文件的路径信息，如表 3-6 所示。

表 3-6 AK 格式数据源组件的参数

参数名称	参数说明
filePath	【必填】AK 格式文件的路径信息

AK 格式的批式和流式数据导出（Sink）组件的参数是一样的，具体参数如表 3-7 所示。

表3-7 AK格式导出组件的参数

参数名称	参数说明
filePath	【必填】CSV格式文件的路径信息
numFiles	【可选】保存文件的个数，默认值为1
overwriteSink	【可选】是否覆盖写已有的数据，默认值为False，即报错并退出

其中，最后两个参数是导出（Sink）操作所特有的，其与 CSV 格式的数据导出（Sink）组件的参数含义一样。当参数 numFiles 大于 1 时，参数 filePath 对应的就是文件夹，在该文件夹中指定写到 numFiles 个文件。如果目标数据文件（夹）不存在，则新建该数据表或文件（夹），然后将其整体导出。如果目标数据文件（夹）存在，则需要考虑参数“overwriteSink”，默认值为 false，即报错并退出。如果用户将其设为 true，则会删除已存在的目标数据文件（夹），并新建该数据表或文件（夹），然后将其整体导出，即该导出操作有覆盖写的效果。

与 CSV 格式相比，AK 格式有如下特点：

- 不用额外记录数据的 schemaStr。
- 参数的个数大幅减少。
- 避免了各种数据类型与文本格式转换过程中出现的兼容问题。
- 支持数据压缩，文件占用的空间更小。

示例

本节的实验与对 CSV 格式数据进行的综合实验相似。首先，定义三个目标文件路径，分别位于三个文件系统：本地文件系统、Hadoop 分布式文件系统、阿里云 OSS 文件系统。具体代码如下：

```
OssFileSystem oss =
    new OssFileSystem(
        OSS_END_POINT,
        OSS_BUCKET_NAME,
        OSS_ACCESS_ID,
        OSS_ACCESS_KEY
    );
FilePath[] filePaths = new FilePath[] {
    new FilePath(LOCAL_DIR + "iris.ak"),
    new FilePath(HDFS_URI + "user/yangxu/alink/data/temp/iris.ak"),
    new FilePath(OSS_PREFIX_URI + "alink/data/temp/iris.ak", oss)
};
```

随后，针对批式处理场景设计实验。从网络的 HTTP 数据源中读取 iris 数据，将其导出（Sink）到某个目标文件路径，然后读取该目标文件路径的数据，使用 count 方法统计其记录总条数，验证此新生成的数据文件是否正确（正确的总记录数为 150 条）。对三个不同文件系统的目

文件路径均执行此操作，代码如下：

```

for (FilePath filePath : filePaths) {
    new CsvSourceBatchOp()
        .setFilePath(IRIS_HTTP_URL)
        .setSchemaStr(IRIS_SCHEMA_STR)
        .link(
            new AkSinkBatchOp()
                .setFilePath(filePath)
                .setOverwriteSink(true)
        );
    BatchOperator.execute();

    System.out.println(
        new AkSourceBatchOp()
            .setFilePath(filePath)
            .count()
    );
}

```

运行结果如下：

```

150
150
150

```

最后，针对流式场景设计实验。从网络的 HTTP 数据源中流式读取 iris 数据，将其导出(Sink)到某个目标文件路径，然后读取该目标文件路径的数据。因为从该数据源读取的是流式数据，而流式数据无法直接使用 count 方法进行验证，所以这里就选取满足条件"sepal_length < 4.5"的数据，并打印输出，以此来进行验证。对三个不同文件系统的目标文件路径均执行此操作，代码如下：

```

for (FilePath filePath : filePaths) {
    new CsvSourceStreamOp()
        .setFilePath(IRIS_HTTP_URL)
        .setSchemaStr(IRIS_SCHEMA_STR)
        .link(
            new AkSinkStreamOp()
                .setFilePath(filePath)
                .setOverwriteSink(true)
        );
    StreamOperator.execute();

    new AkSourceStreamOp()
        .setFilePath(filePath)
        .filter("sepal_length < 4.5")
        .print();
    StreamOperator.execute();
}

```

三次运行结果如下。进行对比后可发现，其结果一致。

sepal_length	sepal_width	petal_length	petal_width	category
4.4000	3.2000	1.3000	0.2000	Iris-setosa
4.4000	3.0000	1.3000	0.2000	Iris-setosa
4.4000	2.9000	1.4000	0.2000	Iris-setosa
4.3000	3.0000	1.1000	0.1000	Iris-setosa
sepal_length	sepal_width	petal_length	petal_width	category
4.3000	3.0000	1.1000	0.1000	Iris-setosa
4.4000	3.2000	1.3000	0.2000	Iris-setosa
4.4000	2.9000	1.4000	0.2000	Iris-setosa
4.4000	3.0000	1.3000	0.2000	Iris-setosa
sepal_length	sepal_width	petal_length	petal_width	category
4.4000	2.9000	1.4000	0.2000	Iris-setosa
4.4000	3.2000	1.3000	0.2000	Iris-setosa
4.4000	3.0000	1.3000	0.2000	Iris-setosa
4.3000	3.0000	1.1000	0.1000	Iris-setosa



数据库与数据表

数据库（Database）是重要的数据存储方式，Alink 使用 Flink Catalog 进行数据库和数据表的基本操作。Flink Catalog 提供了元数据，如数据库、表、分区、视图以及访问数据库或其他外部系统中存储的数据所需的函数和信息。同时 Alink 提供了针对数据表的批式/流式数据源（Source）组件及数据导出（Sink）组件。

4.1 简介

4.1.1 Catalog的基本操作

Catalog 允许用户引用数据库系统中现有的元数据，并自动将它们映射到 Flink 的相应元数据中。例如，Flink 可以自动将 JDBC（Java Database Connectivity）表映射到 Flink 表中，用户不必在 Flink 中手动重写 DDL（Data Definition Language）内容。Catalog 极大地简化了用户现有系统开始使用 Flink 所需的步骤，并大大改善了用户体验。关于 Catalog API 的详细介绍，可参见 Flink 官网的说明（参见链接 4-1）。

Catalog 提供了一个统一的 API 来管理元数据，元数据包括数据库（Database）、数据表（Table）、表结构等信息。图 4-1 显示了 Catalog、Database 与 Table 间的层次关系。

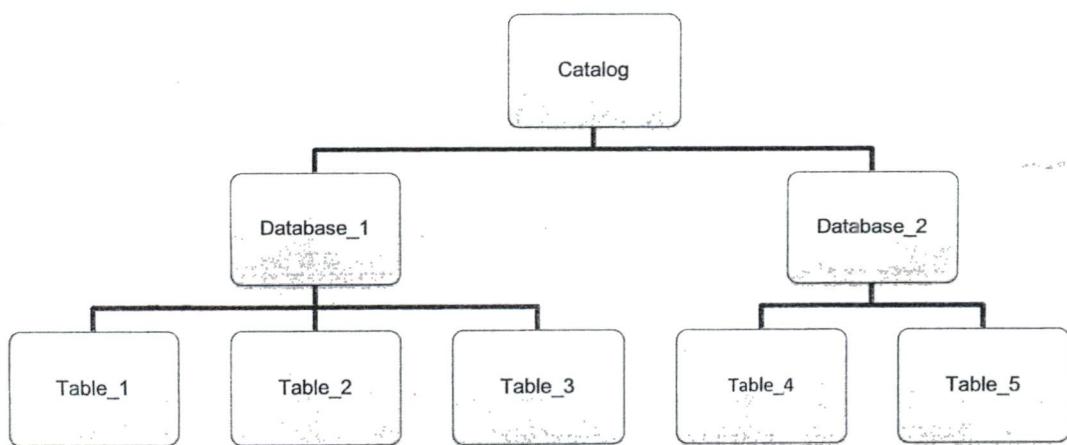


图 4-1

表 4-1 列出了与数据库（Database）和数据表（Table）相关的常用操作。

表 4-1 与数据库和数据表相关的常用操作

与数据库相关的操作	与数据表相关的操作
listDatabases: 列出所有的数据库	listTables: 列出所有的数据表
databaseExists: 判断数据库是否存在	tableExist: 判断数据表是否存在
dropDatabases: 删除数据库	dropTable: 删除数据表
getDefalutDataBase: 获取默认的数据库	renameTable: 重命名数据表
getDatabase: 获取特定的数据库	getTable: 获取指定的数据表
createDatabases: 创建数据库	createTable: 创建数据表
alterDatabases: 修改数据库	alterTable: 修改数据表

4.1.2 Source和Sink组件

本节所要讨论的数据源（Source）组件与导出（Source）组件都是以单个数据表为基本单位的。这里需要用到 Catalog，以及路径信息：

```
public ObjectPath(String databaseName, String objectName)
```

Alink 定义了类 CatalogObject，包括 Catalog 数据源（Source）组件与寻出（Source）组件所需的全部元数据，其构造函数有如下两种形式：

```
public CatalogObject(BaseCatalog catalog, ObjectPath objectPath)
public CatalogObject(BaseCatalog catalog, ObjectPath objectPath, Params params)
```

显然，指定 Catalog 和路径信息（数据库→表）是必需的。

Catalog 的批式/流式数据源与导出组件，共有四个，都使用 CatalogObject 类型参数；而且每个组件都只有一个参数，参数的用法也相同，如表 4-2 所示。

表 4-2 数据表的批式/流式数据源与导出组件参数

参数名称	参数类型	参数说明
catalogObject	CatalogObject	包括 Catalog 和路径信息（数据库→表）

四个组件的含义如下：

- 批式数据源组件（CatalogSourceBatchOp）
- 流式数据源组件（CatalogSourceStreamOp）
- 批式数据导出组件（CatalogSinkBatchOp）

批式数据导出的原则是数据要进行整体导出，不进行追加导出。如果目标数据表不存在，则新建该数据表，然后将其整体导出；如果目标数据表存在，则报错并退出。所以，在执行数据的导出操作前，需要判断目标数据表是否存在，并执行相应的删除操作，以保证数据的导出过程顺利进行。

- 流式数据导出组件（CatalogSinkStreamOp）

在目标数据表不存在的情况下，流式数据导出与批式数据导出的行为是相似的（新建数据表，然后将数据导出到目标数据表）。需要特别注意的是，在目标数据表存在的情况下，批式导出会报错并退出；而流式导出不会报错，会继续执行下去。流式导出的数据会追加在原有数据上。

4.2 Hive示例

Hive 是一个基于 Hadoop 的数据仓库平台，在大数据场景中应用广泛。在本节中，我们会演示 Alink 任务如何从 Hive 读取数据，如何将结果写入 Hive。

首先设置常量如下。其中，IRIS_URL 和 IRIS_SCHEMA_STR 用于读取 HTTP 数据，定义 Hive Catalog 中的数据库名称为 DB_NAME，使用批式读/写的数据表名称为 BATCH_TABLE_NAME，使用流式读/写的数据表名称为 STREAM_TABLE_NAME；这里所用的 Hive 版本为 HIVE_VERSION，其配置文件目录为 HIVE_CONF_DIR。

```

static final String IRIS_URL =
    "http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data";
static final String IRIS_SCHEMA_STR =
    "sepal_length double, sepal_width double, petal_length double, petal_width double, category
string";
static final String DB_NAME = "test_db";
static final String BATCH_TABLE_NAME = "batch_table";
static final String STREAM_TABLE_NAME = "stream_table";
static final String HIVE_VERSION = "2.3.4";
static final String HIVE_CONF_DIR = ... ;

```

创建 HiveCatalog 对象，创建实验用的数据库 DB_NAME：

```

HiveCatalog hive = new HiveCatalog("hive_catalog", null, HIVE_VERSION, HIVE_CONF_DIR);
hive.open();
hive.createDatabase(DB_NAME, new CatalogDatabaseImpl(new HashMap <>(), ""), true);

```

分别以批式/流式方式读取 iris 数据，并将这些数据以批式/流式方式导出到 Hive 的数据表 BATCH_TABLE_NAME 和 STREAM_TABLE_NAME 中。代码如下：

```

new CsvSourceBatchOp()
.setFilePath(IRIS_URL)
.setSchemaStr(IRIS_SCHEMA_STR)
.lazyPrintStatistics("< origin data >")
.link(
    new CatalogSinkBatchOp()
        .setCatalogObject(new CatalogObject(hive, new ObjectPath(DB_NAME, BATCH_TABLE_NAME)))
);
BatchOperator.execute();

new CsvSourceStreamOp()
.setFilePath(IRIS_URL)
.setSchemaStr(IRIS_SCHEMA_STR)
.link(
    new CatalogSinkStreamOp()
        .setCatalogObject(new CatalogObject(hive, new ObjectPath(DB_NAME, STREAM_TABLE_NAME)))
);
StreamOperator.execute();

```

随后，我们分别以批式/流式方式读取数据内容，并计算统计信息，打印采样数据。详细代码如下：

```

new CatalogSourceBatchOp()
.setCatalogObject(new CatalogObject(hive, new ObjectPath(DB_NAME, BATCH_TABLE_NAME)))
.lazyPrintStatistics("< batch catalog source >");
BatchOperator.execute();

```

```

new CatalogSourceStreamOp()
.setCatalogObject(new CatalogObject(hive, new ObjectPath(DB_NAME, STREAM_TABLE_NAME)))
.sample(0.02)
.print();
StreamOperator.execute();

```

运行结果如下：

```

< batch catalog source >
Summary:
+-----+-----+-----+-----+-----+-----+
| colName | count | missing | sum | mean | variance | min | max |
+-----+-----+-----+-----+-----+-----+
| SEPAL_LENGTH | 150 | 0 | 876.5 | 5.8433 | 0.6857 | 4.3 | 7.9 |
| SEPAL_WIDTH | 150 | 0 | 458.1 | 3.054 | 0.188 | 2 | 4.4 |
| PETAL_LENGTH | 150 | 0 | 563.8 | 3.7587 | 3.1132 | 1 | 6.9 |
| PETAL_WIDTH | 150 | 0 | 179.8 | 1.1987 | 0.5824 | 0.1 | 2.5 |
| CATEGORY | 150 | 0 | NaN | NaN | NaN | NaN | NaN |
+-----+-----+-----+-----+-----+-----+
sepal_length|sepal_width|petal_length|petal_width|category
-----+-----+-----+-----+-----+
4.4000|2.9000|1.4000|0.2000|Iris-setosa
5.5000|4.2000|1.4000|0.2000|Iris-setosa
6.8000|3.0000|5.5000|2.1000|Iris-virginica

```

最后，演示一下文件操作。下面列出所有的数据表。根据前面的操作，我们知道共有两个表。首先判断数据表 BATCH_TABLE_NAME 是否存在，然后执行删除操作。也可以直接使用 dropTable 方法删除数据表 STREAM_TABLE_NAME。注意将第二个参数设为 true，代表忽略存在的表。最后显示数据表列表。如果操作成功的话，最后的列表应该为空。

```

System.out.println("< tables before drop >");
System.out.println(JsonConverter.toJson(hive.listTables(DB_NAME)));

if (hive.tableExists(new ObjectPath(DB_NAME, BATCH_TABLE_NAME))) {
    hive.dropTable(new ObjectPath(DB_NAME, BATCH_TABLE_NAME), false);
}
hive.dropTable(new ObjectPath(DB_NAME, STREAM_TABLE_NAME), true);

System.out.println("< tables after drop >");
System.out.println(JsonConverter.toJson(hive.listTables(DB_NAME)));

```

运行结果如下。执行删除操作之前，有两个数据表；执行删除操作之后，没有数据表，显然删除操作达到了预期效果：

```

< tables before drop >
["batch_table", "stream_table"]
< tables after drop >
[]

```

4.3 Derby示例

Derby 是完全用 Java 编写的内存数据库，属于 Apache 的一个开源项目，且基于 Apache License 2.0 分发。Derby 的核心部分只有 2MB，非常小巧，适合在单机上运行。

首先需要设置常量如下。这里所用的 Derby 版本为 DERBY_VERSION，其数据所在目录为 DERBY_DIR：

```
static final String DERBY_VERSION = "10.6.1.0";
static final String DERBY_DIR = ... ;
```

创建 DerbyCatalog 对象，创建实验用的数据库 DB_NAME：

```
DerbyCatalog derby = new DerbyCatalog("derby_catalog", null, DERBY_VERSION, DATA_DIR + DERBY_DIR);
derby.open();
derby.createDatabase(DB_NAME, new CatalogDatabaseImpl(new HashMap<>(), ""), true);
```

分别以批式/流式方式读取 iris 数据，并将这些数据以批式/流式方式导出到 Derby 的数据表 BATCH_TABLE_NAME 和 STREAM_TABLE_NAME 中。代码如下：

```
new CsvSourceBatchOp()
.setFilePath(IRIS_URL)
.setSchemaStr(IRIS_SCHEMA_STR)
.lazyPrintStatistics("< origin data >")
.link(
    new CatalogSinkBatchOp()
        .setCatalogObject(new CatalogObject(derby, new ObjectPath(DB_NAME, BATCH_TABLE_NAME)))
);
BatchOperator.execute();

new CsvSourceStreamOp()
.setFilePath(IRIS_URL)
.setSchemaStr(IRIS_SCHEMA_STR)
.link(
    new CatalogSinkStreamOp()
        .setCatalogObject(new CatalogObject(derby, new ObjectPath(DB_NAME, STREAM_TABLE_NAME)))
);
StreamOperator.execute();
```

随后，我们再分别以批式/流式方式读取数据内容，并计算统计信息，打印采样数据。详细代码如下：

```
new CatalogSourceBatchOp()
.setCatalogObject(new CatalogObject(derby, new ObjectPath(DB_NAME, BATCH_TABLE_NAME)))
.lazyPrintStatistics("< batch catalog source >");
BatchOperator.execute();
```

```

new CatalogSourceStreamOp()
.setCatalogObject(new CatalogObject(derby, new ObjectPath(DB_NAME, STREAM_TABLE_NAME)))
.sample(0.02)
.print();
streamOperator.execute();

```

运行结果如下：

```

< batch catalog source >
Summary:
+-----+-----+-----+-----+-----+-----+
| colName | count | missing | sum | mean | variance | min | max |
+-----+-----+-----+-----+-----+-----+
| SEPAL_LENGTH | 150 | 0 | 876.5 | 5.8433 | 0.6857 | 4.3 | 7.9 |
| SEPAL_WIDTH | 150 | 0 | 458.1 | 3.054 | 0.188 | 2 | 4.4 |
| PETAL_LENGTH | 150 | 0 | 563.8 | 3.7587 | 3.1132 | 1 | 6.9 |
| PETAL_WIDTH | 150 | 0 | 179.8 | 1.1987 | 0.5824 | 0.1 | 2.5 |
| CATEGORY | 150 | 0 | NaN | NaN | NaN | NaN | NaN |
+-----+-----+-----+-----+-----+-----+
SEPAL_LENGTH|SEPAL_WIDTH|PETAL_LENGTH|PETAL_WIDTH|CATEGORY
+-----+-----+-----+-----+
5.5000|3.5000|1.3000|0.2000|Iris-setosa
6.3000|2.5000|4.9000|1.5000|Iris-versicolor
6.4000|2.9000|4.3000|1.3000|Iris-versicolor
6.7000|3.0000|5.2000|2.3000|Iris-virginica

```

最后，演示一下文件操作。下面列出所有的数据表。根据前面的操作，我们知道共有两个表。首先判断数据表 BATCH_TABLE_NAME 是否存在，然后执行删除操作。也可以直接使用 dropTable 方法删除数据表 STREAM_TABLE_NAME，注意将第二个参数设为 true，代表忽略存在的表。最后显示数据表列表。如果操作成功的话，最后的列表应该为空。

```

System.out.println("< tables before drop >");
System.out.println(JsonConverter.toJson(derby.listTables(DB_NAME)));

if (derby.tableExists(new ObjectPath(DB_NAME, BATCH_TABLE_NAME))) {
    derby.dropTable(new ObjectPath(DB_NAME, BATCH_TABLE_NAME), false);
}
derby.dropTable(new ObjectPath(DB_NAME, STREAM_TABLE_NAME), true);

System.out.println("< tables after drop >");
System.out.println(JsonConverter.toJson(derby.listTables(DB_NAME)));

```

运行结果如下，显然文件操作达到了预期效果：

```

< tables before drop >
["BATCH_TABLE", "STREAM_TABLE"]
< tables after drop >
[]

```

4.4 MySQL示例

MySQL 是最流行的关系型数据库之一。本节将对 MySQL 进行数据源与导出操作，执行显示数据表列表、判断数据表是否存在、删除数据表等操作。

首先设置常量如下。这里所用的 MySQL 版本为 MYSQL_VERSION，其数据服务的链接地址为 MYSQL_URL，端口号为 MYSQL_PORT，用户名和密码分别为 MYSQL_USER_NAME 和 MYSQL_PASSWORD。

```
static final String MYSQL_VERSION = "5.1.27";
static final String MYSQL_URL = ...;
static final String MYSQL_PORT = ...;
static final String MYSQL_USER_NAME = ...;
static final String MYSQL_PASSWORD = ...;
```

创建 MySqlCatalog 对象，创建实验用的数据库 DB_NAME：

```
MySqlCatalog mySql = new MySqlCatalog("mysql_catalog", null, MYSQL_VERSION,
    MYSQL_URL, MYSQL_PORT, MYSQL_USER_NAME, MYSQL_PASSWORD);

mySql.open();
mySql.createDatabase(DB_NAME, new CatalogDatabaseImpl(new HashMap<>(), ""), true);
```

分别以批式/流式方式读取 iris 数据，并将这些数据以批式/流式方式导出到 MySQL 的数据表 BATCH_TABLE_NAME 和 STREAM_TABLE_NAME 中。代码如下：

```
new CsvSourceBatchOp()
.setFilePath(IRIS_URL)
.setSchemaStr(IRIS_SCHEMA_STR)
.lazyPrintStatistics("< origin data >")
.link(
    new CatalogSinkBatchOp()
        .setCatalogObject(new CatalogObject(mySql, new ObjectPath(DB_NAME, BATCH_TABLE_NAME)))
);
BatchOperator.execute();

new CsvSourceStreamOp()
.setFilePath(IRIS_URL)
.setSchemaStr(IRIS_SCHEMA_STR)
.link(
    new CatalogSinkStreamOp()
        .setCatalogObject(new CatalogObject(mySql, new ObjectPath(DB_NAME, STREAM_TABLE_NAME)))
);
StreamOperator.execute();
```

随后，我们再分别以批式/流式方式读取数据内容，并计算统计信息，打印采样数据。详细代码如下：

```
new CatalogSourceBatchOp()
    .setCatalogObject(new CatalogObject(mySql, new ObjectPath(DB_NAME, BATCH_TABLE_NAME)))
    .lazyPrintStatistics("< batch catalog source >");
BatchOperator.execute();

new CatalogSourceStreamOp()
    .setCatalogObject(new CatalogObject(mySql, new ObjectPath(DB_NAME, STREAM_TABLE_NAME)))
    .sample(0.02)
    .print();
StreamOperator.execute();
```

运行结果如下：

```
< batch catalog source >
Summary:
+-----+-----+-----+-----+-----+-----+
| colName | count | missing | sum | mean | variance |
+-----+-----+-----+-----+-----+-----+
| SEPAL_LENGTH | 150 | 0 | 876.5 | 5.8433 | 0.6857 | 4.3 | 7.9 |
| SEPAL_WIDTH | 150 | 0 | 458.1 | 3.054 | 0.188 | 2 | 4.4 |
| PETAL_LENGTH | 150 | 0 | 563.8 | 3.7587 | 3.1132 | 1 | 6.9 |
| PETAL_WIDTH | 150 | 0 | 179.8 | 1.1987 | 0.5824 | 0.1 | 2.5 |
| CATEGORY | 150 | 0 | NaN | NaN | NaN | NaN | NaN |
+-----+-----+-----+-----+-----+-----+
sepal_length|sepal_width|petal_length|petal_width|category
-----+-----+-----+-----+-----+
6.3000|2.9000|5.6000|1.8000|Iris-virginica
5.0000|3.6000|1.4000|0.2000|Iris-setosa
```

最后，演示一下文件操作。下面列出所有的数据表。根据前面的操作，我们知道共有两个表。首先判断数据表 BATCH_TABLE_NAME 是否存在。然后执行删除操作。也可以直接使用 dropTable 方法删除数据表 STREAM_TABLE_NAME。注意将第二个参数设为 true，代表忽略存在的表。最后显示数据表列表。如果操作成功的话，最后的列表应该为空。

```
System.out.println("< tables before drop >");
System.out.println(JsonConverter.toJson(mySql.listTables(DB_NAME)));

if (mySql.tableExists(new ObjectPath(DB_NAME, BATCH_TABLE_NAME))) {
    mySql.dropTable(new ObjectPath(DB_NAME, BATCH_TABLE_NAME), false);
}
mySql.dropTable(new ObjectPath(DB_NAME, STREAM_TABLE_NAME), true);

System.out.println("< tables after drop >");
System.out.println(JsonConverter.toJson(mySql.listTables(DB_NAME)));
```

运行结果如下，显然文件操作达到了预期效果：

```
< tables before drop >
["batch_table", "stream_table"]
< tables after drop >
[]
```



支持 Flink SQL

Flink 的 SQL 操作基于实现了 SQL 标准的 Apache Calcite (参见链接 5-1)。Flink SQL 还有一个非常大的优势：对于批式数据表和流式数据表，查询都使用相同的语义，并产生相同的结果。

Alink 批式组件 BatchOperator 的数据格式采用的是批式 Flink Table；同样，Alink 流式组件 StreamOperator 的数据格式采用的是流式 Flink Table。对于 Flink Table 使用 Flink SQL 语句，无须数据格式的转换，只需要在数据表环境 (Table Environment) 中进行相应的注册。这是非常自然和高效的。

Alink 对 Flink SQL 进行了封装，使其和 Alink 组件间的衔接更加自然、方便，而在使用方式上仍延续了 Flink SQL 的风格。

本章以 MovieLens 数据集为例，演示各种 SQL 功能。注意：本书的第 24 章对 MovieLens 数据集有详细的介绍。

5.1 基本操作

5.1.1 注册

SQL 语句中对数据的操作都是通过表名进行的。要对某个 Alink 批式组件 BatchOperator 输出的 Table 类型数据进行操作，需要先为其注册一个名称，具体代码如下：

```
BatchOperator <?> ratings = ...
```

```
BatchOperator <?> users = ...
BatchOperator <?> items = ...

ratings.registerTableName("ratings");
items.registerTableName("items");
users.registerTableName("users");
```

5.1.2 运行

Alink 封装了 Flink 的 `sqlQuery` 方法，将其输出结果包装成了 `BatchOperator`，这样就可以使用 `link` 等方法，连接其他 Alink 组件。

我们使用 SQL 语句做一个示例，从 MovieLens ratings 数据表中计算出某视频网站观众的评价次数最多的 10 部电影，并从 items 数据表中根据观众的 ID 匹配出电影名称。具体代码如下：

```
BatchOperator.sqlQuery(
    "SELECT title, cnt, avg_rating"
    + " FROM ( SELECT item_id, COUNT(*) AS cnt, AVG(rating) AS avg_rating"
    + "        FROM ratings "
    + "       GROUP BY item_id "
    + "      ORDER BY cnt DESC LIMIT 10 "
    + "     ) AS t"
    + " JOIN items"
    + " ON t.item_id=items.item_id"
    + " ORDER BY cnt DESC"
).print();
```

输出结果如下，第一列为影片名称，第二列为观众的评论次数，第三列为电影获得的平均评分：

title	cnt	avg_rating
Star Wars (1977)	583	4.3585
Contact (1997)	509	3.8035
Fargo (1996)	508	4.1555
Return of the Jedi (1983)	507	4.0079
Liar Liar (1997)	485	3.1567
English Patient, The (1996)	481	3.6570
Scream (1996)	478	3.4414
Toy Story (1995)	452	3.8783
Air Force One (1997)	431	3.6311
Independence Day (ID4) (1996)	429	3.4382

为了让读者对影片名有更深刻的印象，下面显示前四名的电影海报，如图 5-1 所示。

看到这里，大家可能会有疑问：排在第 2、3 位的电影似乎没有很大的知名度，它们为什么会排名靠前呢？

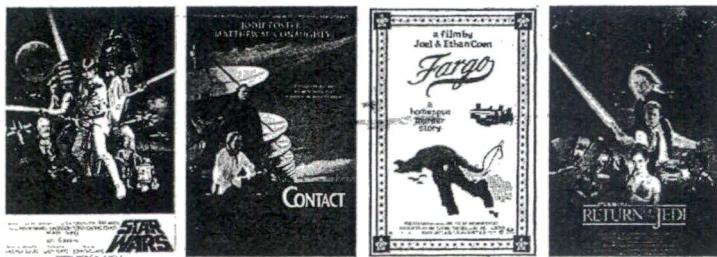


图 5-1

稍后，我们会通过使用 UDF（用户定义函数），得到观众每个评分所发生的时间段，具体分析一下其中的原因。

下面我们再举一个更有挑战性的 SQL 操作示例，计算一下男士与女士评分差异最大的电影。具体代码如下，其中定义了 `m_rating` 和 `f_rating` 来分别记录对于一部影片的男、女平均评分，最后按评分从大到小排列各影片，并获取前 20 部影片的数据：

```
BatchOperator.sqlQuery(
    "SELECT title, cnt, m_rating, f_rating, ABS(m_rating - f_rating) AS diff_rating"
    + " FROM ( SELECT item_id, COUNT(rating) AS cnt, "
    + "           AVG(CASE WHEN gender='M' THEN rating ELSE NULL END) AS m_rating, "
    + "           AVG(CASE WHEN gender='F' THEN rating ELSE NULL END) AS f_rating "
    + "      FROM (SELECT item_id, rating, gender FROM ratings "
    + "            JOIN users ON ratings.user_id=users.user_id)"
    + "            GROUP BY item_id "
    + "        ) AS t"
    + " JOIN items"
    + " ON t.item_id=items.item_id"
    + " ORDER BY diff_rating DESC LIMIT 20"
).print();
```

结果如下：

title	cnt	m_rating	f_rating	diff_rating
Delta of Venus (1994)	2	5.0000	1.0000	4.0000
Two or Three Things I Know About Her (1966)	4	4.6667	1.0000	3.6667
Sliding Doors (1998)	4	4.5000	1.0000	3.5000
Paths of Glory (1957)	33	4.4194	1.0000	3.4194
Magic Hour, The (1998)	5	4.2500	1.0000	3.2500
Love and Death on Long Island (1997)	2	1.0000	4.0000	3.0000
Killer (Bulletproof Heart) (1994)	4	4.0000	1.0000	3.0000
Rough Magic (1995)	2	1.0000	4.0000	3.0000
Visitors, The (Visiteurs, Les) (1993)	2	2.0000	5.0000	3.0000
Little City (1998)	2	5.0000	2.0000	3.0000
.....				

我们看到排名靠前的几部影片的男女评分差异的确很大，排在第一位的电影只有 2 个评分，

分别是最高分和最低分。若评分的人数非常少，则样本的偏差就很大，也就无法代表各影片间的真实评分差异。所以，需要在统计的过程中加入评分个数的限制，具体代码如下：

```
BatchOperator.sqlQuery(
    "SELECT title, `cnt`, `m_rating`, `f_rating`, ABS(`m_rating` - `f_rating`) AS `diff_rating`"
    + " FROM ( SELECT item_id, COUNT(rating) AS `cnt`, "
    + "           AVG(CASE WHEN gender='M' THEN rating ELSE NULL END) AS `m_rating`, "
    + "           AVG(CASE WHEN gender='F' THEN rating ELSE NULL END) AS `f_rating` "
    + "         FROM (SELECT item_id, rating, gender FROM ratings "
    + "                 JOIN users ON ratings.user_id=users.user_id)"
    + "                 GROUP BY item_id "
    + "                 HAVING COUNT(rating)>=50 "
    + "             ) AS t"
    + " JOIN items"
    + " ON t.item_id=items.item_id"
    + " ORDER BY `diff_rating` DESC LIMIT 10"
).print();
```

对评分个数的限制语句为“HAVING COUNT(rating)>=50”，得到的结果如下：

title	cnt	m_rating	f_rating	diff_rating
Ran (1985)	70	4.3214	3.2143	1.1071
Jane Eyre (1996)	63	3.0526	4.1200	1.0674
Postman, The (1997)	58	2.8378	3.9048	1.0669
First Knight (1995)	86	2.7460	3.7826	1.0366
Cook the Thief His Wife & Her Lover, The (1989)	82	3.1940	2.2667	0.9274
Dirty Dancing (1987)	98	2.7742	3.6667	0.8925
Nosferatu (Nosferatu, eine Symphonie des Grauens) (1922)	54	3.6596	2.8571	0.8024
To Wong Foo, Thanks for Everything! Julie Newmar (1995)	57	2.6216	3.4000	0.7784
Daylight (1996)	57	2.5814	3.3571	0.7757
Big Sleep, The (1946)	73	4.2500	3.4762	0.7738

这些影片的评分差异明显变小了。我们关注排在前两位的影片，这两部影片分别被男性观众或女性观众看好。这两部影片的海报及其说明如图 5-2 所示。



该影片讲述的是一个虚构的日本战国时期的
故事，某家族因自相残杀而走向灭亡。



著名小说《简·爱》的电影版

图 5-2

5.1.3 内置函数

Flink SQL 提供了丰富的内置函数，比如字符串函数、算术函数、逻辑函数、比较函数等。内置函数的功能列表随着 Flink 版本而变化，该列表基本上是不断增加的。建议读者根据自己所用的 Flink 版本，到 Flink 网站寻找相应版本的内置函数说明。

5.1.4 用户定义函数

Flink SQL 虽然提供了丰富的内置函数，但是仍不能覆盖所有应用场景，还需要用户根据具体情况定义函数。我们通过一个简单的示例来串联函数的定义和使用过程。

在前面的示例中，我们计算了 ratings 数据表中某视频网站观众的评价次数最多的 10 部电影，但其中排在第 2、3 位的电影似乎没有很大的知名度，我们自然会有疑问：它们为什么会排名靠前呢？

我们注意到 ratings 数据表中有一个数据列，名称为 ts，记录的是评分的 UNIX 时间戳，该时间戳用自 UTC 的 1970 年 1 月 1 日以来的 UNIX 秒数来表示，为长整型数值。我们看到的是一个非常大的整数，无法将其与一个时间关联起来。接下来，我们会通过使用 UDF（用户定义函数），将 UNIX 时间戳转换为我们容易理解的日期时间格式，之后进一步分析上述电影排名靠前的原因。

定义函数 FromUnixTimestamp 如下：

```
public static class FromUnixTimestamp extends ScalarFunction {
    public java.sql.Timestamp eval(Long ts) {
        return new java.sql.Timestamp(ts * 1000);
    }
}
```

在使用前需要注册函数，相关代码如下：

```
BatchOperator.registerFunction("from_unix_timestamp", new FromUnixTimestamp());
```

然后就可以在 SQL 语句中调用 from_unix_timestamp 了，我们将 UNIX 时间戳类型转换为日期类型，并统计评分的起始时间和结束时间。

```
BatchOperator.sqlQuery(
    "SELECT MIN(dt) AS min_dt, MAX(dt) AS max_dt "
    + " FROM ( SELECT from_unix_timestamp(ts) AS dt, 1 AS grp FROM ratings ) "
    + " GROUP BY grp "
).print();
```

输出结果如下：

```
min_dt|max_dt
-----|-----
1997-09-20 11:05:10.0|1998-04-23 07:10:38.0
```

前面介绍了基本的 SQL 语句用法，我们在实际使用中还有更简便的用法。比如对于上面的功能，我们使用如下代码，也可以得到相同的结果：

```
ratings
  .select("from_unix_timestamp(ts) AS dt, 1 AS grp")
  .groupBy("grp", "MIN(dt) AS min_dt, MAX(dt) AS max_dt")
  .print();
```

显然，其在写法上简洁了很多，而且还会省去注册 ratings 的步骤。接下来介绍 SQL 语句的简化操作。

5.2 简化操作

Alink 对 SQL 语句的常用操作进行了包装。BatchOperator 和 StreamOperator 提供了一些方法，可以简化使用 SQL 语句的流程。比如，对单个数据表进行 SELECT 操作，使用 SQL 语句实现的代码如下：

```
ratings.registerTableName("ratings");
BatchOperator ratings_select = BatchOperator.sqlQuery(
    "SELECT user_id, item_id AS movie_id FROM ratings");
```

而使用 Alink 提供的简化方法，一行就可以搞定，代码如下：

```
BatchOperator ratings_select = ratings.select("user_id, item_id AS movie_id");
```

再举一个例子，对两个数据表进行合并操作，合并的过程中不对数据进行去重操作，使用 SQL 语句实现的代码如下：

```
users_1_4.registerTableName("users_1_4");
users_3_6.registerTableName("users_3_6");
BatchOperator.sqlQuery(
    "SELECT * FROM"
    + " ( (SELECT * FROM users_1_4) "
    + " UNION ALL "
    + " (SELECT * FROM users_3_6)"
    + " )"
).print();
```

使用 Alink 提供的简化方法，同样一行就可以搞定，代码如下：

```
new UnionAllBatchOp().linkFrom(users_1_4, users_3_6).print();
```

与标准的 SQL 操作相比，Alink 的简化方法省去了注册数据表的步骤，略去了一些标准样式，只需输入关键语句即可。下面将详细介绍 Alink 封装的一些简化方法。

5.2.1 单表操作

下面将介绍几个针对单个数据表的常用操作。

1. 选择 (select)

其类似于 SQL SELECT 语句。执行选择操作，返回的结果还是 Alink 的批式组件。

```
BatchOperator ratings_select = ratings.select("user_id, item_id AS movie_id");
```

也可以直接使用 Alink 提供的简化方法来打印输出结果：

```
ratings.select("user_id, item_id AS movie_id").firstN(5).print();
```

输出结果如下：

user_id	movie_id
196	242
186	302
22	377
244	51
166	346

注意：可以使用星号 (*) 充当通配符，选择表中的所有列。

```
BatchOperator ratings_select = ratings.select("*");
```

2. 重命名 (as)

重命名字段：

```
ratings.as("f1,f2,f3,f4").firstN(5).print();
```

运行结果如下，可以看到第一行数据列的名称都改变了：

f1	f2	f3	f4
196	242	3.0000	881250949
186	302	3.0000	891717742

```
22|377|1.0000|878887116
244|51|2.0000|880606923
166|346|1.0000|886397596
```

3. 过滤 (filter/where)

filter(where)方法类似于SQL WHERE子句，可筛选出满足条件的行。

```
ratings.filter("rating > 3").firstN(5).print();
```

或者

```
ratings.where("rating > 3").firstN(5).print();
```

运行结果如下：

user_id	item_id	rating	ts
298	474	4.0000	884182806
253	465	5.0000	891628467
286	1014	5.0000	879781125
200	222	5.0000	876042340
122	387	5.0000	879270459

4. 不同记录集 (distinct)

distinct方法类似于SQL DISTINCT子句，可返回具有不同值组合的记录。

```
users.select("gender").distinct().print();
```

输出结果如下：

gender
F
M

5. 分组聚合 (groupBy)

groupBy方法类似于SQL GROUP BY子句。其指定包含分组键值的列，将数据集的所有行按键值划分为若干个组；然后对每组数据执行聚合运算（求总数、最大值、平均值等）；最后汇合各组的结果进行输出。groupBy方法共有两个参数：第一个参数用来指定包含分组键值的列，第二个参数定义使用的聚合函数及输出列名。示例代码如下：

```
users.groupBy("gender","gender", COUNT(*) AS cnt").print();
```

输出结果如下：

gender	cnt
F	273
M	670

6. 排序并输出片段 (orderBy)

orderBy 方法类似于 SQL ORDER BY 子句。必须输入的参数为用来排序的数据列名称。在选取输出数据方面，提供了两种方式：

- (1) 输入一个参数，指定选择输出数据集中最前面的多少行。
- (2) 输入两个参数，表示从某个偏移量开始，选择多少行。

我们通过示例来展示这两种方式的不同，代码如下：

```
users.orderBy("age", 5).print();
users.orderBy("age", 1, 3).print();
```

使用 Java 编辑器可以清晰地显示各参数的含义，如图 5-3 所示。

```
users.orderBy( fieldName: "age", limit: 5).print();
users.orderBy( fieldName: "age", offset: 1, fetch: 3).print();
```

图 5-3

运行结果如下：

user_id	age	gender	occupation	zip_code
30	7	M	student	55436
471	10	M	student	77459
289	11	M	none	94619
142	13	M	other	48118
880	13	M	student	83702

user_id	age	gender	occupation	zip_code
471	10	M	student	77459
289	11	M	none	94619
142	13	M	other	48118

这里共显示了 2 个结果数据：第 1 个结果数据为“age”从小到大排序的前 5 名；第 2 个结果数据仍以“age”从小到大排序，但偏移了 1 个位置，从第 2 位开始，取了 3 行。

在前面的例子中，数据都是从小到大排序的。如果我们想要查找最年长的几位用户的数据，该怎么处理呢？

orderBy 方法还提供了一个布尔类型的参数：是否按升序排列，其默认值为 true，即按升序排列——从小到大排列。该参数放在方法的最后，下面有两个示例：

```
users.orderBy("age", 5, false).print();
users.orderBy("age", 1, 3, false).print();
```

使用 Java 编辑器可以清晰地显示各参数的含义，如图 5-4 所示。

<code>users.orderBy(fieldName: "age", limit: 5, isAscending: false).print();</code>
<code>users.orderBy(fieldName: "age", offset: 1, fetch: 3, isAscending: false).print();</code>

图 5-4

运行结果分别如下：

user_id	age	gender	occupation	zip_code
481	73	M	retired	37771
860	70	F	retired	48322
803	70	M	administrator	78212
767	70	M	engineer	00000
559	69	M	executive	10022

user_id	age	gender	occupation	zip_code
860	70	F	retired	48322
803	70	M	administrator	78212
767	70	M	engineer	00000

这里共显示了 2 个结果数据：第 1 个结果数据为“age”从大到小排序的前 5 名；第 2 个结果数据仍以“age”从大到小排序，但偏移了 1 个位置，从第 2 位开始，取了 3 行。

如果读者想更深入地了解各参数组合之间的关系，可以参见其 Java 接口的代码。

```
public BatchOperator orderBy(String fieldName, int limit) {
    return orderBy(fieldName, limit, true);
}

public BatchOperator orderBy(String fieldName, int limit, boolean isAscending) {
    // ...
}

public BatchOperator orderBy(String fieldName, int offset, int fetch) {
    return orderBy(fieldName, offset, fetch, true);
}

public BatchOperator orderBy(String fieldName, int offset, int fetch, boolean isAscending)
{
    // ...
}
```

5.2.2 两表的连接（JOIN）操作

两表的连接（JOIN）操作细分为以下 4 种情况：

- INNER JOIN：如果左表和右表中的数据都匹配，则返回匹配的行。其经常被简写为“JOIN”。
- LEFT OUTER JOIN：即使右表中没有匹配的数据，也从左表中返回所有的行。
- RIGHT OUTER JOIN：即使左表中没有匹配的数据，也从右表中返回所有的行。
- FULL OUTER JOIN：为 LEFT OUTER JOIN 与 RIGHT OUTER JOIN 结果的并集。

我们从 ratings 中过滤出一些数据，保留少量的 item_id 和 user_id。从 items 数据表中选取少量数据，为了让生成的两个数据集中的 item_id 互不包含，且数量较少，便于查看，我们选取了奇数编号的数据。还有一点，JOIN 类组件在描述匹配关系的时候，由于不能像 SQL 语句那样加数据表名称的前缀，两个数据集中列的名称不应相同，因此，我们要将第 2 个数据集中的列名“item_id”转换为“movie_id”。具体代码如下：

```
BatchOperator left_ratings
= ratings
.filter("user_id<3 AND item_id<4")
.select("user_id, item_id, rating");

BatchOperator right_movies
= items
.select("item_id AS movie_id, title")
.filter("movie_id < 6 AND MOD(movie_id, 2) = 1");
```

我们将这两个数据表打印出来，以便观察其特点。

```
System.out.println("# left_ratings #");
left_ratings.print();
System.out.println("\n# right_movies #");
right_movies.print();
```

显示结果如下：

```
# left_ratings #
user_id|item_id|rating
-----|-----|-----
1|2|3.0000
2|1|4.0000
1|1|5.0000
1|3|4.0000
```

```
# right_movies #
movie_id|title
-----|-----
```

```
1|Toy Story (1995)
3|Four Rooms (1995)
5|Copycat (1995)
```

在左表 left_ratings 中含有 item_id 的值为{1,2,3}，在右表 right_movies 中含有 movie_id 的值为{1,3,5}，两边有交叉，且不互相包含，该示例可以用来演示出 JOIN 操作的 4 种情况。

下面 4 个示例，都是针对数据表 left_ratings 和 right_movies 进行的操作，匹配条件是“item_id = movie_id”，选择左表 left_ratings 的全部列，以及右表的 title 列，即将左表的数据集加上电影名称。具体代码如下：

```
System.out.println("# JOIN #");
new JoinBatchOp()
    .setJoinPredicate("item_id = movie_id")
    .setSelectClause("user_id, item_id, title, rating")
    .linkFrom(left_ratings, right_movies)
    .print();

System.out.println("\n# LEFT OUTER JOIN #");
new LeftOuterJoinBatchOp()
    .setJoinPredicate("item_id = movie_id")
    .setSelectClause("user_id, item_id, title, rating")
    .linkFrom(left_ratings, right_movies)
    .print();

System.out.println("\n# RIGHT OUTER JOIN #");
new RightOuterJoinBatchOp()
    .setJoinPredicate("item_id = movie_id")
    .setSelectClause("user_id, item_id, title, rating")
    .linkFrom(left_ratings, right_movies)
    .print();

System.out.println("\n# FULL OUTER JOIN #");
new FullOuterJoinBatchOp()
    .setJoinPredicate("item_id = movie_id")
    .setSelectClause("user_id, item_id, title, rating")
    .linkFrom(left_ratings, right_movies)
    .print();
```

打印结果如下：

```
# JOIN #
user_id|item_id|title|rating
_____|_____|_____|_____
2|1|Toy Story (1995)|4.0000
1|1|Toy Story (1995)|5.0000
1|3|Four Rooms (1995)|4.0000
```

```
# LEFT OUTER JOIN #
user_id|item_id|title|rating
```

```

-----|-----|-----|-----
1|1|Toy Story (1995)|5.0000
2|1|Toy Story (1995)|4.0000
1|2|null|3.0000
1|3|Four Rooms (1995)|4.0000

#_RIGHT OUTER JOIN #
user_id|item_id|title|rating
-----|-----|-----|-----
2|1|Toy Story (1995)|4.0000
1|1|Toy Story (1995)|5.0000
1|3|Four Rooms (1995)|4.0000
null|null|Copycat (1995)|null

# FULL OUTER JOIN #
user_id|item_id|title|rating
-----|-----|-----|-----
1|1|Toy Story (1995)|5.0000
2|1|Toy Story (1995)|4.0000
1|2|null|3.0000
1|3|Four Rooms (1995)|4.0000
null|null|Copycat (1995)|null

```

在此可以看到：

- 执行 JOIN 操作，返回的是左右表中能匹配上的数据。
- 执行 LEFT OUTER JOIN 操作，除了返回左右表中能匹配上的数据，还输出了左表中没有匹配上的数据。对于没有匹配上的电影名称，赋为 null。
- 执行 RIGHT OUTER JOIN 操作，除了返回左右表中能匹配上的数据，还输出了右表中没有匹配上的数据。对于没有匹配上的电影名称，赋为 null。
- 执行 FULL OUTER JOIN 操作，包括了三部分内容：左右表中能匹配上的数据、左表中没有匹配上的数据，以及右表中没有匹配上的数据。

5.2.3 两表的集合操作

以数据表的行作为基本单位，将行看作集合的元素。集合支持的操作可以分为下面两类：

(1) 严格的集合操作。集合中没有重复元素，数据表中的相同元素会被看作同一个元素。

严格的集合操作有三种：合并 (UNION)、相交 (INTERSECT) 及减 (MINUS) 操作。

(2) 扩展的集合操作。考虑到元素重复的个数，在相应操作名称的后面加上“ALL”，以便与严格的集合操作有所区分。扩展的集合操作也有 3 种：全体合并 (UNION ALL)、全体相交 (INTERSECT ALL) 及全体减 (MINUS ALL) 操作。

为了更好地演示下面的操作，我们先生成 2 个小数据表，从 users 中分别取 user_id 为 1~4 号的数据，及 user_id 为 3~6 号的数据，具体代码如下：

```
BatchOperator users_1_4 = users.filter("user_id<5");
System.out.println("# users_1_4 #");
users_1_4.print();

BatchOperator users_3_6 = users.filter("user_id>2 AND user_id<7");
System.out.println("\n# users_3_6 #");
users_3_6.print();
```

输出结果如下：

```
# users_1_4 #
user_id|age|gender|occupation|zip_code
-----|---|----|-----|-----
1|24|M|technician|85711
2|53|F|other|94043
3|23|M|writer|32067
4|24|M|technician|43537

# users_3_6 #
user_id|age|gender|occupation|zip_code
-----|---|----|-----|-----
3|23|M|writer|32067
4|24|M|technician|43537
5|33|F|other|15213
6|42|M|executive|98101
```

这 2 个数据集各有 4 行数据，并且都包含 user_id 为 3 和 4 的两行数据。

1. 全体合并 (UNION ALL)

执行 UNION ALL 操作可将两个表合并起来，并要求两个表的列（包括列类型、列顺序）完全一致。执行 UNION ALL 操作只是一种直接的合并，并不会判断两个表是否有相同的行，不会进行去重操作。Alink 提供了两个组件 UnionAllBatchOp 和 UnionAllStreamOp，它们是分别针对批式数据和流式数据设计的。示例代码如下：

```
new UnionAllBatchOp().linkFrom(users_1_4, users_3_6).print();
```

输出结果如下：

```
user_id|age|gender|occupation|zip_code
-----|---|----|-----|-----
1|24|M|technician|85711
2|53|F|other|94043
3|23|M|writer|32067
```

```
4|24|M|technician|43537
3|23|M|writer|32067
4|24|M|technician|43537
5|33|F|other|15213
6|42|M|executive|98101
```

可以看到，`user_id` 为 3、4 的数据有重复。

2. 合并 (UNION)

执行 UNION 操作可将两个表合并起来，并要求两个表的列（包括列类型、列顺序）完全一致。如果两个表有相同的行，会有去重操作。Alink 提供了 UnionBatchOp 组件处理批式数据，示例代码如下：

```
new UnionBatchOp().linkFrom(users_1_4, users_3_6).print();
```

输出结果如下：

user_id	age	gender	occupation	zip_code
1 24 M technician 85711				
2 53 F other 94043				
3 23 M writer 32067				
4 24 M technician 43537				
5 33 F other 15213				
6 42 M executive 98101				

在此可以看到，结果中没有相同的数据行，这是去重操作的结果。

3. 相交 (INTERSECT)

执行该操作类似于执行 SQL INTERSECT 子句。执行该操作，可返回两个表中都存在的记录。如果一个记录在一个或两个表中存在一次以上，则仅返回一次，即结果表中没有重复的记录。两个表必须具有相同的字段类型。

```
new IntersectBatchOp().linkFrom(users_1_4, users_3_6).print();
```

输出结果如下：

user_id	age	gender	occupation	zip_code
3 23 M writer 32067				
4 24 M technician 43537				

这里输出了 `users_1_4` 和 `users_3_6` 中都出现的 3 号和 4 号数据。

4. 全体相交 (INTERSECT ALL)

执行该操作类似于执行 SQL INTERSECT ALL 子句。执行该操作，可返回两个表中都存在

的记录。如果一个记录在两个表中均多次出现，则执行该操作所返回的记录次数，与在两个表中均多次出现该记录时所返回的记录次数一样，即结果表中可能有重复的记录。两个表必须具有相同的字段类型。

我们构造两个带有重复数据的数据集：一个数据集是 users_1_4 的数据重复一次，每个 ID 号（1~4 号）都有两条数据；另外一个数据集是 users_1_4 与 users_3_6 进行 UNION ALL 操作，该结果集包含 1~6 号的数据，其中 3 号、4 号数据都为两条。然后对这两个数据集进行 INTERSECT ALL 操作，具体代码如下：

```
new IntersectAllBatchOp()
    .linkFrom(
        new UnionAllBatchOp().linkFrom(users_1_4, users_1_4),
        new UnionAllBatchOp().linkFrom(users_1_4, users_3_6)
    )
    .print();
```

输出结果如下：

user_id	age	gender	occupation	zip_code
1	24	M	technician	85711
2	53	F	other	94043
3	23	M	writer	32067
3	23	M	writer	32067
4	24	M	technician	43537
4	24	M	technician	43537

两个数据集中的相同数据都会在结果中出现。3 号、4 号数据在原先的两个数据集中均出现了 2 次，所以其在结果中也会出现 2 次。

5. 减 (MINUS)

执行该操作类似于执行 SQL EXCEPT 子句。执行减操作，可从左表中返回在右表中不存在的记录。左表中的重复记录仅返回一次，即删除了重复项。两个表必须具有相同的字段类型。

```
new MinusBatchOp().linkFrom(users_1_4, users_3_6).print();
```

输出结果如下：

user_id	age	gender	occupation	zip_code
1	24	M	technician	85711
2	53	F	other	94043

在此，左边的数据集 users_1_4，去掉了其与 users_3_6 都有的 3 号、4 号数据。

6. 全体减（MINUS ALL）

执行该操作类似于执行 SQL EXCEPT ALL 子句。执行该操作，可返回右表中不存在的记录。如果某条记录在左表中出现 n 次且在右表中出现 m 次，则将返回 $(n-m)$ 次该记录，即，删除与右表中存在的重复项一样多的记录。两个表必须具有相同的字段类型。

采用 INTERSECT ALL 例子中构造的数据集，即 2 个带有重复数据的数据集。其中，一个数据集 users_1_4 的数据重复一次，每个 ID 号（1~4 号）都有两条数据；另外一个数据集是 users_1_4 与 users_3_6 进行 UNION ALL 操作，该结果集包含 1~6 号的数据，其中 3 号、4 号数据都为两条。然后对这两个数据集进行 MINUS ALL 操作，具体代码如下：

```
new MinusAllBatchOp()
    .linkFrom(
        new UnionAllBatchOp().linkFrom(users_1_4, users_1_4),
        new UnionAllBatchOp().linkFrom(users_1_4, users_3_6)
    )
    .print();
```

输出结果如下：

user_id	age	gender	occupation	zip_code
1	24	M	technician	85711
2	53	F	other	94043

注意：1 号、2 号数据在左边的数据集中出现了 2 次，在右边的数据集中出现了 1 次；3 号、4 号数据在左右两个数据集中出现的次数相同，均为 2 次。所以，最终的结果为 1 号和 2 号数据各出现了 1 次。

5.3 深入介绍Table Environment

Flink SQL 语句是通过 Batch Table Environment 和 Stream Table Environment 执行的。Alink 的运行环境中包括了 Batch/Stream Table Environment，并提供了接口。

- 获取 Alink 默认使用的 Batch Table Environment，使用 Java 代码：

```
MLEnvironmentFactory.getDefault().getBatchTableEnvironment()
```

- 获取 Alink 默认使用的 Stream Table Environment，使用 Java 代码：

```
MLEnvironmentFactory.getDefault().getStreamTableEnvironment()
```

通过下面的代码可获取 Alink 默认的 Batch Table Environment，并打印出当前 Table Environment 中所有注册的数据表名称列表。

```
String[] tableNames = MLEnvironmentFactory.getDefault().getBatchTableEnvironment().listTables();
System.out.println("Table Names : ");
for (String name : tableNames) {
    System.out.println(name);
}
```

同样的语句，将 getBatchTableEnvironment 换成 getStreamTableEnvironment，就可对流式数据表进行同样的操作。

Flink Table Environment 提供了两种执行方法，即 sqlQuery 和 sqlUpdate：

- sqlQuery 方法会对已在 TableEnvironment 中注册的数据表进行 SQL 查询，结果为数据表。
- sqlUpdate 方法可用来执行 SQL 语句，如 INSERT、UPDATE 或 DELETE；或执行 DDL 语句。

简单来说，sqlQuery 会返回数据表的结果，sqlUpdate 没有返回值。

Alink 对 sqlQuery 进行了简单的封装，定义了静态方法 batchSQL，具体代码如下：

```
/** 
 * Evaluates a SQL query on registered tables and retrieves the result as a 
<code>BatchOperator</code>. 
* 
* @param query The SQL query to evaluate. 
* @return The result of the query as <code>BatchOperator</code>. 
*/ 
public BatchOperator <?> batchSQL(String query) { 
    return new TableSourceBatchOp(getBatchTableEnvironment().sqlQuery(query)); 
}
```

Alink 将其输出结果包装成 BatchOperator，这样就可以使用 link 等方法，连接其他 Alink 组件了。

5.3.1 注册数据表名

Alink BatchOperator 的 registerTableName 方法，可在 Alink 默认的 Batch Table Environment 中，将批式组件的主输出注册为相应的名称。

```
BatchOperator <?> ratings = ... 
BatchOperator <?> users = ... 
BatchOperator <?> items = ... 

ratings.registerTableName("ratings");
```

```
items.registerTableName("items");
users.registerTableName("users");
```

检查一下 Batch Table Environment 的数据表名称列表，代码如下：

```
String[] tableNames
    = MLEnvironmentFactory.getDefault().getBatchTableEnvironment().listTables();
System.out.println("Table Names : ");
for (String name : tableNames) {
    System.out.println(name);
}
```

执行结果如下，前面设置过的 3 个数据表名都显示了出来，说明其已经可以在 SQL 语句中使用了。

```
Table Names :
ratings
items
users
```

5.3.2 撤销数据表名

与数据表名的注册功能相对应，撤销数据表名可以使用如下操作：

```
BatchTableEnvironment batchTableEnvironment
    = MLEnvironmentFactory.getDefault().getBatchTableEnvironment();

System.out.println("Table Names : ");
for (String name : batchTableEnvironment.listTables()) {
    System.out.println(name);
}

batchTableEnvironment.sqlUpdate("DROP TABLE IF EXISTS users");

System.out.println("\nTable Names After DROP : ");
for (String name : batchTableEnvironment.listTables()) {
    System.out.println(name);
}
```

执行结果如下：

```
Table Names :
ratings
items
users

Table Names After DROP :
ratings
items
```

5.3.3 扫描已注册的表

前面介绍的注册操作，指的是将数据表注册到 Table Environment 中；而扫描操作则是反向的操作，可通过扫描的方式来获取 Table Environment 中已注册的表。示例如下：

```
BatchOperator ratings_scan  
= BatchOperator.fromTable(batchTableEnvironment.scan("ratings"));  
ratings_scan.firstN(5).print();
```

其中，batchTableEnvironment.scan("ratings")得到的是 Flink Table 类型对象。还需要使用 BatchOperator.fromTable()转化为 Alink 批式组件。

运行结果如下：

user_id	item_id	rating	ts
196	242	3.0000	881250949
186	302	3.0000	891717742
22	377	1.0000	878887116
244	51	2.0000	880606923
166	346	1.0000	886397596



用户定义函数（UDF/UDTF）

虽然 Alink 提供了丰富的算法组件，也支持 Flink SQL 的丰富操作，但对复杂的实际场景，用户还需要自己能够灵活地定义函数功能，处理批式/流式数据。

用户定义函数分为标量函数（Scalar-Valued Function）和表值函数(Table-Valued Function)，两者的关系如表 6-1 所示。

- 在标量函数中将零个、一个或多个标量值作为输入参数，函数的输出结果为单个标量值。
- 与标量函数相似，在表值函数中也将零个、一个或多个标量值作为输入参数。但是，其输出结果与标量函数有很大差异。该函数可以返回任意数量的行作为输出，而不是单个值。其返回的行可能包含一列或多列数据。

表 6-1 标量函数和表值函数对比表

	输入参数	输出结果
标量函数	零个、一个或多个标量值	单个标量值
表值函数	零个、一个或多个标量值	返回任意数量的行作为输出，返回的行可能包含一列或多列数据

6.1 用户定义标量函数（UDF）

用户定义标量函数（User Defined scalar-valued Function），简称 UDF。自定义标量函数是用户最常用的方式。下面将演示如何定义 UDF，并应用到批式处理和流式处理场景中。

6.1.1 示例数据及问题

我们选择 MovieLens 的一个数据集（参见链接 6-1），该数据集包含了用户对电影评分的具体数据。该数据集包含 943 个用户对 1682 个物品（电影）进行的 100 000 次评分数据；每个用户至少针对 20 部电影进行了评分；用户和物品（电影）都是从 1 开始编号的。图 6-1 是数据文件的部分内容。

			u.data
196	242	3	881250949
186	302	3	891717742
22	377	1	878887116
244	51	2	880606923
166	346	1	886397596
298	474	4	884182806
115	265	2	881171488
253	465	5	891628467
305	451	3	886324817
6	86	3	883683013

图 6-1

在该文件中，每个属性之间用制表符\t分隔。该文件共包含 4 个属性：

- user_id：用户 ID。
- item_id：物品（电影）ID。
- rating：评分。
- ts：UNIX 时间戳，是自 UTC 的 1970 年 1 月 1 日以来的 UNIX 秒数。

在此可以看到，第 4 列 ts 显示的 UNIX 秒数虽然精确地表示了时间，但是我们却很难从这一串数字中知道其具体表示哪年哪月。下面我们就用自定义方式实现一个这样的函数：将 UNIX 秒数转换为标准格式的时间信息。

6.1.2 UDF的定义

定义 Java 版本标量函数需要注意下面三点：

- 必须继承基类：

org.apache.flink.table.functions.ScalarFunction

- 必须实现 eval 方法，标量函数的行为是由 eval 方法定义的。
- 可以选择是否重载 getResultType 方法。一般情况下，不必重载该方法。在如下定义中去掉重载，得到的计算结果也是一样的。

示例定义如下：

```

import org.apache.flink.api.common.typeinfo.TypeInformation;
import org.apache.flink.api.common.typeinfo.Types;
import org.apache.flink.table.functions.ScalarFunction;
...
public static class FromUnixTimestamp extends ScalarFunction {
    ...
    public java.sql.Timestamp eval(Long ts) {
        return new java.sql.Timestamp(ts * 1000);
    }
    @Override
    public TypeInformation <?> getResultType(Class <?>[] signature) {
        return Types.SQL_TIMESTAMP;
    }
}

```

6.1.3 使用UDF处理批式数据

定义好 UDF 后，可以通过 UDFBatchOp 组件调用 UDF，该组件需要通过 setFunc 方法指定 UDF，并设置输入和输出的数据列名称参数。注意，如果我们将输出名称与输入名称设置成一样的，则数据结果会替换掉输入列的原始数据。

```

ratings
    .link(
        new UDFBatchOp()
            .setFunc(new FromUnixTimestamp())
            .setSelectedCols("ts")
            .setOutputCol("ts")
    )
    .firstN(5)
    .print();

```

运行结果如下：

user_id	item_id	rating	ts
196	242	3.0000	1997-12-04 23:55:49.0
186	302	3.0000	1998-04-05 03:22:22.0
22	377	1.0000	1997-11-07 15:18:36.0
244	51	2.0000	1997-11-27 13:02:03.0
166	346	1.0000	1998-02-02 13:33:16.0

进一步，我们注册此 UDF，注册代码如下：

```
BatchOperator.registerFunction("from_unix_timestamp", new FromUnixTimestamp());
```

这样就可以在 SQL 语句中调用此函数了。一个简单的示例如下：

```
ratings
  .select("user_id, item_id, rating, from_unix_timestamp(ts) AS ts")
  .firstN(5)
  .print();
```

运行结果如下：

user_id	item_id	rating	ts
196	242	3.0000	1997-12-04 23:55:49.0
186	302	3.0000	1998-04-05 03:22:22.0
22	377	1.0000	1997-11-07 15:18:36.0
244	51	2.0000	1997-11-27 13:02:03.0
166	346	1.0000	1998-02-02 13:33:16.0

与前面使用 UDFBatchOp 组件调用 UDF 的结果相同。我们还可以在 sqlQuery() 中使用 UDF，相关代码如下，得到的结果与使用 UDFBatchOp 组件得到的结果仍然一致。需要注意的是，使用 SQL 语句前，要先注册数据表 ratings。

```
ratings.registerTableName("ratings");

BatchOperator
  .sqlQuery("SELECT user_id, item_id, rating, from_unix_timestamp(ts) AS ts FROM ratings")
  .firstN(5)
  .print();
```

6.1.4 使用UDF处理流式数据

为了便于打印显示结果，我们先对流式数据进行过滤，减少数据条数，代码如下：

```
ratings = ratings.filter("user_id=1 AND item_id<5");
ratings.print();
StreamOperator.execute();
```

运行结果如下，只有 4 条数据：

user_id	item_id	rating	ts
1	1	5.0000	874965758
1	3	4.0000	878542960
1	2	3.0000	876893171
1	4	3.0000	876893119

与批式处理的情况类似，可以通过 UDFStreamOp 组件调用 UDF，该组件需要通过 setFunc 方法指定 UDF，并设置输入和输出的数据列名称参数。注意，在这个示例中，我们仍然将输出名称与输入名称设置成一样的，数据结果会替换掉输入列的原始数据。

```

ratings
  .link(
    new UDFStreamOp()
      .setFunc(new FromUnixTimestamp())
      .setSelectedCols("ts")
      .setOutputCol("ts")
  )
  .print();

StreamOperator.execute();

```

运行结果如下：

user_id	item_id	rating	ts
1	1	5.0000	1997-09-23 06:02:38.0
1	3	4.0000	1997-11-03 15:42:40.0
1	2	3.0000	1997-10-15 13:26:11.0
1	4	3.0000	1997-10-15 13:25:19.0

进一步，我们注册此 UDF，注册代码如下。注意，这里需要使用的是 StreamOperator 的 registerFunction 方法。

```
StreamOperator.registerFunction("from_unix_timestamp", new FromUnixTimestamp());
```

随后，就可以在 SQL 语句中调用此函数了。一个简单的示例如下：

```

ratings
  .select("user_id, item_id, rating, from_unix_timestamp(ts) AS ts")
  .print();

StreamOperator.execute();

```

运行结果如下：

user_id	item_id	rating	ts
1	1	5.0000	1997-09-23 06:02:38.0
1	3	4.0000	1997-11-03 15:42:40.0
1	2	3.0000	1997-10-15 13:26:11.0
1	4	3.0000	1997-10-15 13:25:19.0

与前面使用 UDFStreamOp 组件调用 UDF 的结果相同。我们还可以在 sqlQuery() 中使用 UDF，相关代码如下，得到的结果与使用 UDFStreamOp 组件得到的结果仍然一致。需要注意的是，使用 SQL 语句前，要先注册 ratings。

```
ratings.registerTableName("ratings");

StreamOperator
    .sqlQuery("SELECT user_id, item_id, rating, from_unix_timestamp(ts) AS ts FROM ratings")
    .print();

StreamOperator.execute();
```

6.2 用户定义表值函数(UDTF)

用户定义表值函数(User Defined Table-valued Function, UDTF)，可以用来解决输入一行、输出多行的问题。

6.2.1 示例数据及问题

我们仍然使用 MovieLens 数据，使用影片信息数据集并选择 item_id 和 title 两个字段。在此选择前 10 条数据并打印：

item_id	title
1	Toy Story (1995)
2	GoldenEye (1995)
3	Four Rooms (1995)
4	Get Shorty (1995)
5	Copycat (1995)
6	Shanghai Triad (Yao a yao yao dao waipo qiao) (1995)
7	Twelve Monkeys (1995)
8	Babe (1995)
9	Dead Man Walking (1995)
10	Richard III (1995)

接下来，我们统计标题(title)字段的词个数，该操作需要两步：首先将标题(title)按词切割开并记录各词在当前标题中出现的个数。然后，进行 groupBy 操作，汇总出每个单词出现的总次数。第一步操作会将一条数据按其包含单词的情况分为多条数据，这不符合前面讲的标量函数(UDF)的输出形式要求，必须通过用户定义表值函数(UDTF)进行处理。

6.2.2 UDTF的定义

定义 Java 版本的表值函数（UDTF）需要注意下面三点：

- 必须继承基类：

```
org.apache.flink.table.functions.TableFunction
```

- 必须实现 eval 方法。与标量函数不同的是，eval()没有返回值，但是在运行过程中，其可以多次调用 TableFunction 的 collect 方法，将所要输出的数据逐条提交给 collect 方法进行输出。
- 必须重载 getResultType 方法，详细定义函数输出结果的各数据列类型。

具体代码如下：

```
public static class WordCount extends TableFunction <Row> {

    private HashMap <String, Integer> map = new HashMap <>();

    public void eval(String str) {
        if (null == str || str.isEmpty()) {
            return;
        }
        for (String s : str.split(" ")) {
            if (map.containsKey(s)) {
                map.put(s, 1 + map.get(s));
            } else {
                map.put(s, 1);
            }
        }
        for (Entry <String, Integer> entry : map.entrySet()) {
            collect(Row.of(entry.getKey(), entry.getValue()));
        }
        map.clear();
    }

    @Override
    public TypeInformation <Row> getResultType() {
        return Types.ROW(Types.STRING, Types.INT);
    }
}
```

6.2.3 使用UDTF处理批式数据

定义好 UDTF 后，可以通过 UDTFBatchOp 组件调用 UDTF，该组件需要通过 setFunc 方法指定 UDTF，并设置输入和输出的数据列名称参数。具体代码如下：

```

BatchOperator <?> words = items.link(
    new UDTFBatchOp()
        .setFunc(new WordCount())
        .setSelectedCols("title")
        .setOutputCols("word", "cnt")
        .setReservedCols("item_id")
);
words.lazyPrint(10, "<- after word count ->");

```

运行结果如下，可以看到每条数据用空格分隔单词，年份和外面的括号也被看作“单词”进行了个数统计。读者可能会有点疑问：第一条数据为“Toy Story (1995)”，但为什么单词输出的顺序却是反过来的？这是因为在实现过程中使用HashMap进行判重和计数操作，HashMap是按字母升序方式获取结果的。

item_id	word	cnt
1	(1995)	1
1	Story	1
1	Toy	1
2	(1995)	1
2	GoldenEye	1
3	(1995)	1
3	Four	1
3	Rooms	1
4	Shorty	1
4	(1995)	1
4	Get	1
5	(1995)	1
5	Copycat	1
6	a	1
6	(1995)	1
6	dao	1
6	waipo	1
6	Shanghai	1
6	yao	2
6	qiao)	1

接下来，使用SQL语句进行汇总统计。具体代码如下：

```

words.groupBy("word", "word", SUM(cnt) AS cnt")
    .orderBy("cnt", 20, false)
    .print();

```

结果如下面左栏所示，右栏是对统计结果的一些说明。

word	cnt	说明
The	356	查看单词出现次数的排名，很容易发现： • 出现次数最多的是“The”。“The”大多数时候出现在片名的开头，有356次。

The 356
(1996) 298
(1995) 296
(1994) 237
(1997) 235
the 145
of 137
(1993) 130
and 66
in 63
(1998) 53
A 50
(1992) 40
a 31
to 31
Love 27
(1991) 24
(1990) 24
Man 23
My 22

- 紧随其后的是片名中出现的年份，而年份中又以 1996 年居多。（1996）出现了 298 次。
- 之后是小写的“the”。“the”出现了 145 次，若加上前面首字母大写的“The”，该单词（大小写形式均包括在内）出现的总次数要远超其他单词。
- 介词“of”出现了 137 次。
- 连词“and”出现的次数为 66 次。
- 除以上这些年份词、介词、连词、冠词等外，出现次数最多的是“Love”。“Love”出现了 27 次。
- “Man”和“My”出现的次数也不少，其分别排在第 19 位和第 20 位。

进一步，我们换一种 UDTF 的调用方式。注册 UDTF，注册代码如下：

```
BatchOperator.registerFunction("word_count", new WordCount());
```

注册的函数 word_count 可以在 sqlQuery() 中使用，相关代码如下，得到的结果与前一种调用方式所得结果仍然一致。需要注意的是，在使用 SQL 语句前，还要先注册数据表 ratings。具体代码如下：

```
items.registerTableName("items");

BatchOperator
  .sqlQuery("SELECT item_id, word, cnt FROM items, "
    + "LATERAL TABLE(word_count(title)) as T(word, cnt)")
  .firstN(10)
  .print();
```

运行结果如下，其与使用 UDTFBatchOp 所调用的结果相同：

item_id	word	cnt
1 (1995)	1	
1 Story	1	
1 Toy	1	
2 (1995)	1	
2 GoldenEye	1	
3 (1995)	1	
3 Four	1	
3 Rooms	1	

```
4|Shorty|1
4|(1995)|1
4|Get|1
5|(1995)|1
5|Copycat|1
6|a|1
6|(1995)|1
6|dao|1
6|waipo|1
6|Shanghai|1
6|yao|2
6|qiao)|1
```

6.2.4 使用UDTF处理流式数据

为了便于打印显示结果，我们先对流式数据进行过滤，减少数据条数。具体代码如下：

```
items = items.select("item_id, title").filter("item_id<4");
items.print();
StreamOperator.execute();
```

运行结果如下，只有3条数据：

```
item_id|title
-----|-----
1|Toy Story (1995)
2|GoldenEye (1995)
3|Four Rooms (1995)
```

与批式处理情况类似，通过 UDTFStreamOp 组件调用 UDTF，该组件需要通过 setFunc 方法指定 UDTF，并设置输入和输出的数据列名称参数。具体代码如下：

```
StreamOperator <?> words = items.link(
    new UDTFStreamOp()
        .setFunc(new WordCount())
        . setSelectedCols("title")
        .setOutputCols("word", "cnt")
        .setReservedCols("item_id")
);
words.print();
StreamOperator.execute();
```

运行结果如下：

```
item_id|word|cnt
-----|---|---
```

```

1|(1995)|1
1|Story|1
1|Toy|1
2|(1995)|1
2|GoldenEye|1
3|(1995)|1
3|Four|1
3|Rooms|1

```

在此可以看到每条数据用空格分隔单词，年份和外面的括号也被看作“单词”进行了个数统计。

下面，我们换一种 UDTF 的调用方式。注册此 UDTF，注册代码如下。注意，这里需要使用的是 StreamOperator 的 registerFunction 方法。

```
StreamOperator.registerFunction("word_count", new WordCount());
```

随后，就可以在 sqlQuery() 中调用注册的函数 word_count 了，具体代码如下。需要注意的是，使用 SQL 语句前，要先注册数据表 items。

```

items.registerTableName("items");

StreamOperator.sqlQuery("SELECT item_id, word, cnt FROM items,
+ "LATERAL TABLE(word_count(title)) as T(word, cnt)")
.print();

StreamOperator.execute();

```

运行结果如下，与前面使用 UDTFStreamOp 所调用 UDTF 的结果相同：

```

item_id|word|cnt
-----|---|---
1|(1995)|1
1|Story|1
1|Toy|1
2|(1995)|1
2|GoldenEye|1
3|(1995)|1
3|Four|1
3|Rooms|1

```

7

基本数据处理

Alink 集成了强大的数据处理能力，包括以下几个方面：

- 支持 Flink SQL 操作（本书第 5 章专门进行了介绍），比如，选取列、数据过滤、类型转换、基本数值计算、判断逻辑，分组聚合、表连接、表合并以及字符串操作等。
- 支持用户定义函数 UDF/UDTF（本书第 6 章专门进行了介绍）。
- 针对机器学习领域一些常用的基本操作，Alink 提供的数据处理组件。本章将重点介绍这些组件。

7.1 采样

本节会先介绍一个和采样比较相似的常用操作：取“前”几条数据。该操作可以被看作不考虑各个数据的获取概率的“采样”。

三种常用的采样方式包括随机采样、加权采样和分层采样，如表 7-1 所示。

表 7-1 常用的采样方式

名称	描述
随机采样	所有数据样本被选中的概率是一样的
加权采样	每个数据样本都有一个权值，该权值与样本被选中的概率成正比
分层采样	每个数据样本都有一个“标签”，可以据此将数据进行“分层”。可为每个标签（即其对应的一“层”数据）分别设置采样的比例

7.1.1 取“前”N个数据

如果我们只想看几条数据，以便对数据集有一个直观的认识，就可以指定取“前”几条数据（注意：“前”是由双引号括起来的，在多并发的情况下，Flink 任务不保证数据的顺序，不能严格地说我们所取出的一定是数据文件的头几条数据）。Alink 提供了 FirstNBatchOp 组件，可通过设置参数 size 控制所取数据的数量。也可以使用包装的方法 firstN(int size)，设置所取数据的数量。这两种方式是等价的。下面的示例展示了这两种使用方式：

```
source
    .link(
        new FirstNBatchOp()
            .setSize(5)
    )
    .print();

source.firstN(5).print();
```

运行结果如下。两次均显示了 5 条数据，但数据并不一致。大家在使用时如果需要确定的顺序，建议采用 SQL Orderby 的方式。

sepal_length	sepal_width	petal_length	petal_width	category
5.8000	2.8000	5.1000	2.4000	Iris-virginica
6.4000	3.2000	5.3000	2.3000	Iris-virginica
6.5000	3.0000	5.5000	1.8000	Iris-virginica
7.7000	3.8000	6.7000	2.2000	Iris-virginica
7.7000	2.6000	6.9000	2.3000	Iris-virginica

sepal_length	sepal_width	petal_length	petal_width	category
5.0000	2.0000	3.5000	1.0000	Iris-versicolor
5.9000	3.0000	4.2000	1.5000	Iris-versicolor
6.0000	2.2000	4.0000	1.0000	Iris-versicolor
6.1000	2.9000	4.7000	1.4000	Iris-versicolor
5.6000	2.9000	3.6000	1.3000	Iris-versicolor

7.1.2 随机采样

Alink 提供了随机采样组件 SampleBatchOp 和 SampleWithSizeBatchOp，但是实际应用中经常使用其包装的方法 sample 或者 sampleWithSize。从名称上很容易分辨出两者的区别，前者通过指定采样的比例来确定采样数据的个数，而后者则直接指定确定的采样数据个数。示例代码如下：

```
source
    .sampleWithSize(50)
    .lazyPrintStatistics("< after sample with size 50 >")
    .sample(0.1)
    .print();
```

运行结果如下。从统计结果上能够看出，sampleWithSize(50)的运行结果的确只有 50 条数据。在此基础上的 10%采样，实际获得了 4 条数据。

```
< after sample with size 50 >
+-----+-----+-----+-----+-----+-----+
| colName | count | missing | sum | mean | variance | min | max |
+-----+-----+-----+-----+-----+-----+
| sepal_length | 50 | 0 | 291.4 | 5.828 | 0.6347 | 4.4 | 7.7 |
| sepal_width | 50 | 0 | 151.7 | 3.034 | 0.208 | 2.2 | 4.4 |
| petal_length | 50 | 0 | 186.9 | 3.738 | 2.8991 | 1 | 6.7 |
| petal_width | 50 | 0 | 57.7 | 1.154 | 0.4597 | 0.2 | 2.5 |
| category | 50 | 0 | NaN | NaN | NaN | NaN | NaN |

sepal_length|sepal_width|petal_length|petal_width|category
-----|-----|-----|-----|-----
4.8000|3.4000|1.9000|0.2000|Iris-setosa
5.8000|2.7000|5.1000|1.9000|Iris-virginica
5.0000|3.6000|1.4000|0.2000|Iris-setosa
6.1000|2.6000|5.6000|1.4000|Iris-virginica
```

在随机采样时可以选择是否“放回采样”。放回采样指的是逐个抽取样本时，选中的样本还会被放回总体样本中，随后该样本可能被多次选中。若不放回采样，则可保证每个样本最多被选中一次。Alink 的随机采样默认选择“不放回采样”的方式。示例代码如下：

```
source
    .lazyPrintStatistics("< origin data >")
    .sampleWithSize(150, true)
    .lazyPrintStatistics("< after sample with size 150 >")
    .sample(0.03, true)
    .print();
```

结果如下，原始数据为 150 条，选择“放回采样”方式采样时也得到了 150 条数据。虽然我们没有逐条比对，但从统计结果上看，原始数据和采样数据的求和、均值等指标不一致，表明这两个数据表虽然同为 150 条数据，但数据内容不完全一致。其原因是选择“放回采样”方式采样导致个别数据被选到多次。数据的总条数不变，说明有的数据还没被选到，从而导致了统计指标的变化。后面在 150 条数据的基础上，按 3%的采样率，得到 3 条数据。由于该采样率较低，因此不容易出现相同的数据。

```
< origin data >
+-----+-----+-----+-----+-----+
| colName | count | missing | sum | mean | variance | min | max |
+-----+-----+-----+-----+-----+
```

sepal_length	150	0 876.5 5.8433	0.6857 4.3 7.9
sepal_width	150	0 458.1 3.054	0.188 2 4.4
petal_length	150	0 563.8 3.7587	3.1132 1 6.9
petal_width	150	0 179.8 1.1987	0.5824 0.1 2.5
category	150	0 NaN NaN	NaN NaN NaN

```
< after sample with size 150 >
+-----+-----+-----+-----+-----+-----+
| colName | count | missing | sum | mean | variance | min | max |
+-----+-----+-----+-----+-----+-----+
| sepal_length | 150 | 0|863.1| 5.754 | 0.6455|4.3|7.7 |
| sepal_width | 150 | 0|461.6|3.0773 | 0.2295|2|4.4 |
| petal_length | 150 | 0|537.1|3.5807 | 3.1585|1|6.9 |
| petal_width | 150 | 0|169.8|1.132 | 0.6119|0.1|2.5 |
| category | 150 | 0|NaN|NaN | NaN|NaN|NaN
```

sepal_length	sepal_width	petal_length	petal_width	category
5.000	2.4000	3.7000	1.0000	Iris-versicolor
4.5000	2.3000	1.3000	0.3000	Iris-setosa
6.5000	3.2000	5.1000	2.0000	Iris-virginica

对于流式数据，Alink 同样提供了随机采样组件，但只支持采用按比例采样的方式。示例代码如下：

```
source_stream.sample(0.1).print();
StreamOperator.execute();
```

运行结果如下：

sepal_length	sepal_width	petal_length	petal_width	category
5.7000	3.8000	1.7000	0.3000	Iris-setosa
6.1000	2.8000	4.7000	1.2000	Iris-versicolor
6.2000	2.9000	4.3000	1.3000	Iris-versicolor
4.9000	2.5000	4.5000	1.7000	Iris-virginica
4.9000	3.1000	1.5000	0.1000	Iris-setosa
4.4000	3.0000	1.3000	0.2000	Iris-setosa
5.0000	3.5000	1.3000	0.3000	Iris-setosa
5.4000	3.0000	4.5000	1.5000	Iris-versicolor

7.1.3 加权采样

加权采样指的是每个数据样本都有一个权值，该权值与该样本被选中的概率成正比。比如，对于样本 1 和样本 2，权值分别为 50 和 10，则样本 1 被选中的概率是样本 2 的 5 倍。Alink 的加权采样组件为 `WeightSampleBatchOp`。与随机采样相比，其多了一个必选的参数：权重列的

名称 WeightCol。

示例代码如下。在采样前使用 SQL 语句新建了权重列，根据花的类别不同而赋予不同的权重：Iris-versicolor 类别的权重为 1，Iris-setosa 类别的权重为 2，Iris-virginica 类别的权重为 4。在采样结束后，统计花中各个类别样本的个数。在理想状态下，这三类花采样后的比例为 1 : 2 : 4。

```
source
  .select("*, CASE category WHEN 'Iris-versicolor' THEN 1 "
    + "WHEN 'Iris-setosa' THEN 2 ELSE 4 END AS weight")
  .link(
    new WeightSampleBatchOp()
      .setRatio(0.4)
      .setWeightCol("weight")
  )
  .groupBy("category", "category", COUNT(*) AS cnt")
  .print();
```

运行结果如下。其中，Iris-versicolor 类别的数据量最少，Iris-setosa 类别的数据量居中，Iris-virginica 类别的数据量最多。这三个类别的采样数据量之比为 13 : 17 : 30，接近理想状态的 1 : 2 : 4。

category	cnt
Iris-setosa	17
Iris-virginica	30
Iris-versicolor	13

7.1.4 分层采样

分层采样，对于含有“分层列”的数据，各“层”数据可以设置不同的采样比例。在处理分类问题时，对于正负样本数量相差悬殊的情况，常用该采样方法，并对拥有数量较多“标签”的样本设置较小的采样比例。Alink 的加权采样组件为 StratifiedSampleBatchOp。需要设定分层列 StrataCol，并设置分层列中各属性值的采样比例。

示例代码如下。其中，根据花的类别不同而赋予不同的采样比例，Iris-versicolor 类别的比例为 0.2，Iris-setosa 类别的比例为 0.4，Iris-virginica 类别的比例为 0.8。在采样结束后，会统计花中各个类别样本的个数。花中三个类别的原始数据量都为 50，理想的采样结果数据量之比为 10 : 20 : 40。

```
source
  .link(
    new StratifiedSampleBatchOp()
      .setStrataCol("category")
      .setStrataRatios("Iris-versicolor:0.2,Iris-setosa:0.4,Iris-virginica:0.8")
```

```

    )
    .groupBy("category", "category, COUNT(*) AS cnt")
    .print();

```

运行结果如下。其中，Iris-versicolor 类别的数据量最少，Iris-setosa 类别的数据量居中，Iris-virginica 类别的数据量最多。这三个类别的采样数据量之比为 8 : 19 : 43，接近理想状态的 10 : 20 : 40。

category	cnt
Iris-setosa	19
Iris-virginica	43
Iris-versicolor	8

对于流式数据，同样可以使用分层采样方式。对应的组件为 StratifiedSampleStreamOp，其参数设置与批式任务相同。示例代码如下：

```

source_stream
    .link(
        new StratifiedSampleStreamOp()
            .setStrataCol("category")
            .setStrataRatios("Iris-versicolor:0.2,Iris-setosa:0.4,Iris-virginica:0.8")
    )
    .print();

StreamOperator.execute();

```

该示例打印的数据较多，这里就不进行展示了。从显示的采样数据中，能够感觉到 Iris-versicolor 类别的数据量较少，Iris-virginica 类别的数据量较多。

7.2 数据划分

我们在做机器学习实验的时候，经常要把原始数据分为训练集和测试集，数据划分组件（SplitBatchOp）就会对应着两个输出：主输出为训练集；侧输出（Side Output）只有一个，输出的是测试集。

首先构建 SplitBatchOp 的实例 spliter。设置其切分比例为 0.9，即主输出占总样本的 90%；侧输出为其余的数据，占总样本的 10%。将原始数据 source 连接到 spliter。具体代码如下：

```

SplitBatchOp spliter = new SplitBatchOp().setFraction(0.9);

source.link(spliter);

```

下面获取一些组件输出的相关信息。直接使用 getSchema 方法，即可获得组件的主输出的

Schema。通过 `getSideOutputCount()` 获得该组件侧输出的个数，随后便可使用 `getSideOutput(index)` 方法通过索引号来获取具体的侧输出。每个侧输出是 `BatchOperator` 或者 `StreamOperator`。仍然可以继续使用 `getSchema` 方法。具体代码如下：

```
System.out.println("schema of spliter's main output:");
System.out.println(spliter.getSchema());

System.out.println("count of spliter's side outputs:");
System.out.println(spliter.getSideOutputCount());

System.out.println("schema of spliter's side output :");
System.out.println(spliter.getSideOutput(0).getSchema());
```

输出结果如下。其中，侧输出只有一个；主输出的 Schema 与侧输出的 Schema 相同。

schema of spliter's main output:

```
root
|--- sepal_length: DOUBLE
|--- sepal_width: DOUBLE
|--- petal_length: DOUBLE
|--- petal_width: DOUBLE
|--- category: STRING
```

count of spliter's side outputs:

```
1
schema of spliter's side output :
root
|--- sepal_length: DOUBLE
|--- sepal_width: DOUBLE
|--- petal_length: DOUBLE
|--- petal_width: DOUBLE
|--- category: STRING
```

接下来，我们将分别针对主输出和侧输出数据进行操作，使用 `lazyPrintStatistics` 方法打印当前数据统计结果，并将数据保存到数据文件中。具体代码如下：

```
spliter
    .lazyPrintStatistics("< Main Output >")
    .link(
        new AkSinkBatchOp()
            .setFilePath(DATA_DIR + TRAIN_FILE)
            .setOverwriteSink(true)
    );
spliter.getSideOutput(0)
    .lazyPrintStatistics("< Side Output >")
    .link(
        new AkSinkBatchOp()
            .setFilePath(DATA_DIR + TEST_FILE)
```

```

    .setOverwriteSink(true)
);
BatchOperator.execute();

```

结果打印如下。在此看到主输出有 135 条数据，侧输出有 15 条数据。我们在检查本地文件夹时可发现新生成的两个数据文件，这表明操作成功。

< Main Output >						
	colName	count	missing	sum	mean	variance
	sepal_length	135	0	790.5	5.8556	0.662
	sepal_width	135	0	413.6	3.0637	0.1923
	petal_length	135	0	508.3	3.7652	3.1132
	petal_width	135	0	163	1.2074	0.5931
	category	135	0	NaN	NaN	NaN

< Side Output >						
	colName	count	missing	sum	mean	variance
	sepal_length	15	0	86	5.7333	0.9467
	sepal_width	15	0	44.5	2.9667	0.151
	petal_length	15	0	55.5	3.7	3.3314
	petal_width	15	0	16.8	1.12	0.5146
	category	15	0	NaN	NaN	NaN

7.3 数值尺度变换

数值尺度变换是围绕统计量进行的。我们先回顾几个统计概念，然后再介绍各个变换。
对于数据 X_1, X_2, \dots, X_n ，有如下定义。

均值 (Mean)

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$$

可用均值来描述数据取值的平均位置。

方差 (Sample Variance)

$$S_n^2 = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2$$

标准差 (Standard Deviation)

$$S_n = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2}$$

标准差在数值上等于方差的开方，二者都是用来反映数据取值的离散（变异）程度的。标准差的量纲与数据的量纲相同。

最大值 (Maximum)

$$X_{\max} = \max(X_1, X_2, \dots, X_n)$$

最小值 (Minimum)

$$X_{\min} = \min(X_1, X_2, \dots, X_n)$$

7.3.1 标准化

标准化操作的定义很简单，一个表的某一列数据就是 X_1, X_2, \dots, X_n ，对其进行变换

$$X'_i = \frac{X_i - \bar{X}}{S_n}$$

得到的数据 X'_1, X'_2, \dots, X'_n 就是标准化处理的结果。将此结果数据存入结果表中。

标准化操作的结果，也被称作标准分数 (Standard Score) 或 Z 分数 (Z Score)。

对一个服从一般正态分布的随机变量 $X \sim N(\mu, \sigma^2)$ ，使用标准化操作转换成

$$Z = \frac{X - \mu}{\sigma}$$

则 Z 为一个服从标准正态分布的随机变量，即 $Z \sim N(0,1)$ 。

标准正态分布的密度分布函数如图 7-1 所示。这里有一些特点值得我们注意：

(1) 密度函数相对于平均值所在的直线对称。平均值与它的众数 (statistical mode) 以及中位数 (median) 为同一数值，标准差为 1。

(2) 68.268949% 的面积在平均数左右的一个标准差范围内，即区间为 $[-1, 1]$ 。

(3) 95.449974% 的面积在平均数左右的两个标准差范围内，即区间为 $[-2, 2]$ 。

(4) 99.730020% 的面积在平均数左右的三个标准差范围内，即区间为 $[-3, 3]$ 。

(5) 99.993666% 的面积在平均数左右的四个标准差范围内，即区间为 $[-4, 4]$ 。