

Flink Table Api & SQL

目录

【翻译】Flink Table Api & SQL —— Overview	8
依赖结构	9
表程序依赖性	10
扩展依赖	11
接下来要去哪里?	12
两个 planner 之间的主要区别	13
表 API 和 SQL 程序的结构	13
创建一个 TableEnvironment	14
在 catalog 中注册表	16
注册表格	16
注册 TableSource	17
注册 TableSink	17
注册外部 catalog	18
查询表	18
表 API	18
SQL	19
混合 Table API 和 SQL	20
发出 Table	20
翻译并执行查询	21
DataStream 和 DataSet API 集成	22
Scala 的隐式转换	23
将 DataStream 或 DataSet 注册为表	23
将 DataStream 或 DataSet 转换为表	23
将表转换为 DataStream 或 DataSet	24
数据类型到表结构的映射	25
查询优化	30

Old planner	30
Blink planner.....	30
解释表	30
【翻译】 Flink Table Api & SQL ——Streaming 概念	35
接下来要去哪里?	35
【翻译】 Flink Table Api & SQL ——Streaming 概念 ——动态表	35
数据流上的关系查询	36
动态表和持续查询.....	36
在流上定义表	37
持续查询.....	37
更新和 Append 查询	39
查询限制.....	39
表到流的转换	39
【翻译】 Flink Table Api & SQL ——Streaming 概念 ——时间属性	41
时间属性简介	41
处理时间	42
在数据流到表的转换期间	42
使用 TableSource	42
事件时间	43
在 DataStream 到 Table 的转换期间.....	43
使用 TableSource	44
【翻译】 Flink Table Api & SQL ——Streaming 概念 ——在持续查询中 Join	46
Regular Joins.....	46
Time-windowed Joins	46
Join with a Temporal Table Function	47
用法.....	49
Processing-time Temporal Joins	49
Event-time Temporal Joins	49
与时态表 Join.....	50
用法.....	52
【翻译】 Flink Table Api & SQL ——Streaming 概念 —— 时态表	52
设计初衷	53

与表的修改历史相关.....	53
与维表变化相关.....	54
时态表函数.....	55
定义时态表函数.....	55
时态表.....	56
定义时态表.....	57
【翻译】 Flink Table Api & SQL —Streaming 概念 —— 表中的模式匹配 Beta 版	57
简介与范例.....	59
安装指南.....	59
SQL 语义	59
例子	60
分区	61
事件顺序	62
定义和度量.....	62
聚合	62
定义模式.....	63
Greedy & Reluctant 的量词	64
时间限制.....	65
输出模式.....	67
模式导航.....	68
模式变量引用	68
逻辑偏移.....	69
匹配后策略.....	71
时间属性	74
控制内存消耗.....	74
已知局限性.....	75
【翻译】 Flink Table Api & SQL —Streaming 概念 —— 查询配置.....	75
空闲状态保留时间.....	76
【翻译】 Flink Table Api & SQL —— 连接到外部系统	77
依赖	78
连接器	78
格式.....	79

总览	79
表结构	81
行时间属性	82
类型字符串	84
更新模式	85
表连接器	85
文件系统连接器	86
Kafka 连接器	86
Elasticsearch 连接器	87
HBase 连接器	89
JDBC 连接器	90
表格格式	93
CSV 格式	93
JSON 格式	95
Apache Avro 格式	98
旧的 CSV 格式	100
更多 TableSources 和 TableSinks	101
OrcTableSource	101
CsvTableSink	102
JDBCAppendTableSink	102
CassandraAppendTableSink	103
【翻译】 Flink Table Api & SQL —— Table API	104
概述与范例	105
Operations	106
Scan, Projection, and Filter	106
Column Operations	107
Aggregations	107
Joins	110
Set Operations	112
OrderBy, Offset & Fetch	114
Insert	115
Group Windows	115

Over Windows.....	118
Row-based Operations.....	120
Data Types.....	125
Expression Syntax.....	125
【翻译】 Flink Table Api & SQL — SQL.....	128
查询.....	128
指定查询.....	129
Supported Syntax.....	129
Operations.....	133
指定 DDL.....	148
Create Table.....	149
Drop Table.....	149
Data Types.....	149
保留关键字.....	150
【翻译】 Flink Table Api & SQL — 内置函数.....	152
标量函数.....	152
比较功能.....	153
逻辑函数.....	154
算术函数.....	155
字符串函数.....	159
时间函数.....	162
条件函数.....	164
类型转换函数功能.....	164
Collection 函数功能.....	165
Value Construction 构造函数.....	165
Value Construction 函数.....	165
Value Access 函数.....	166
Value Access 函数.....	166
分组函数.....	166
hash 函数.....	166
辅助函数.....	167
汇总函数.....	167

时间间隔和点单位说明符	168
列函数.....	170
【翻译】 Flink Table Api & SQL — 自定义 Source & Sink	171
定义 TableSource.....	172
定义 BatchTableSource	173
定义 StreamTableSource	173
使用时间属性定义 TableSource.....	174
使用投影下推定义 TableSource.....	176
使用过滤器下推定义 TableSource	176
定义用于查找的 TableSource	177
定义 Table Sink	178
BatchTableSink	178
AppendStreamTableSink	179
RetractStreamTableSink.....	179
UpsertStreamTableSink.....	179
定义一个 TableFactory	180
在 SQL 客户端中使用 TableFactory.....	182
在 Table & SQL API 中使用 TableFactory	182
【翻译】 Flink Table Api & SQL — 用户定义函数	183
注册用户定义的函数	183
标量函数	184
Table Function.....	185
聚合函数	187
表聚合函数.....	193
实施 UDF 的最佳做法.....	202
将 UDF 与 Runtime 集成	203
【翻译】 Flink Table Api & SQL — Catalog Beta 版.....	204
Catalogs 类型	204
GenericInMemory Catalog	204
Hive Catalog.....	205
用户定义的 Catalog	205
Catalog API.....	205

注册 Catalog	205
更改当前 Catalog 和数据库	205
列出可用 Catalog	205
列出可用的数据库	205
列出可用表	205
【翻译】 Flink Table Api & SQL — SQL 客户端 Beta 版	206
入门	207
启动 SQL 客户端 CLI	207
运行 SQL 查询	207
配置	208
环境文件	209
依赖关系	213
用户定义的函数	214
Catalogs	216
分离的 SQL 查询	217
SQL 视图	218
时态表	219
局限与未来	220
【翻译】 Flink Table Api & SQL — Hive Beta	220
依赖	221
连接到 Hive	223
支持的类型	223
局限性	224
【翻译】 Flink Table Api & SQL — Hive —— 读写 Hive 表	225
从 Hive 读数据	225
写数据到 hive	227
局限性	227
【翻译】 Flink Table Api & SQL — Hive —— Hive 函数	227
使用 Hive 自定义的函数	228
局限性	230
【翻译】 Flink Table Api & SQL — Hive —— 在 scala shell 中使用 Hive 连接器	231
【翻译】 Flink Table Api & SQL — 配置	231

总览.....	232
执行配置选项	232
优化器选项	236
【翻译】 Flink Table Api & SQL — 性能调优 — 流式聚合	238
小批量聚合	238
局部全局聚合	240
分割不同的聚合	241
在不同的聚合上使用 FILTER 修饰符	242

【翻译】 Flink Table Api & SQL —— Overview

本文翻译自官网: <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/>

Flink Table Api & SQL 翻译目录

一直没有用 flink 的 table 或 sql api, 最近开始要使用这部分功能了, 先把官网对应的文档翻译一遍, 方便自己慢慢查看

Table API & SQL

Overview

Concepts & Common API

Data Types

Streaming Concepts ▼

Connect to External Systems

Table API

SQL

Built-In Functions

User-defined Sources & Sinks

User-defined Functions

Catalogs

SQL Client

Hive ▼

Configuration

Performance Tuning ▼

Apache Flink 具有两个关联 API-Table API 和 SQL - 用于统一流和批处理。Table API 是用 于 Scala 和 Java 的语言集成查询 API，它允许以非常直观的方式组合来自关系运算符（例如选择，过滤和联接）的查询。Flink 的 SQL 基于实现 SQL 标准的 [Apache Calcite](#)。无论输入是批处理输入（DataSet）还是流输入（DataStream），在两个接口中指定的查询都具有相同的语义并指定相同的结果。

Table API 和 SQL 接口与 Flink 的 DataStream 和 DataSet API 紧密集成在一起。您可以轻松地所有 API 和基于 API 的库之间切换。例如，您可以使用 [CEP 库](#)从 DataStream 中提取模式，然后再使用 Table API 分析模式，或者您可以在预处理程序上运行 [Gelly 图算法](#)之前，使用 SQL 查询、扫描、过滤和聚合批处理表数据。

请注意，Table API 和 SQL 尚未完成功能，正在积极开发中。**[Table API, SQL]**和 **[stream, batch]**输入的每种组合都不支持所有操作。

依赖结构

从 Flink 1.9 开始，Flink 提供了两种不同的计划程序实现来评估 Table & SQL API 程序：**Blink planner** 和 Flink 1.9 之前可用的 **old planner**。**planner** 负责将关系运算符转换为可执行的、优化的 Flink 作业。两种 **planner** 带有不同的优化规则和运行时类。它们在支持的功能方面也可能有所不同。

注意对于生产用例，建议使用 Flink 1.9 之前的 **old planner。**

所有 Table API 和 SQL 组件都捆绑在 `flink-table` 或 `flink-table-blink` Maven 组件中。

以下依赖关系与大多数项目有关：

- `flink-table-common`：用于通过自定义功能，格式等扩展表生态系统的通用模块。
- `flink-table-api-java`：适用于使用 **Java** 编程语言的纯表程序的 Table & SQL API（处于开发初期，不建议使用！）。
- `flink-table-api-scala`：使用 **Scala** 编程语言的纯表程序的 Table & SQL API（处于开发初期，不建议使用！）。
- `flink-table-api-java-bridge`：使用 **Java** 编程语言支持带有 `DataStream / DataSet` API 的 Table & SQL API。
- `flink-table-api-scala-bridge`：使用 **Scala** 编程语言支持带有 `DataStream / DataSet` API 的 Table & SQL API。
- `flink-table-planner`：表程序 **planner** 和运行时。这是 1.9 版本之前 Flink 的唯一 **planner**。现在仍然是推荐的。
- `flink-table-planner-blink`：新的 **Blink planner**。
- `flink-table-runtime-blink`：新的 **Blink runtime**。
- `flink-table-uber`：将上述 API 模块以及 **old planner** 打包到大多数 Table & SQL API 用例的分发中。默认情况下，超级 JAR 文件 `flink-table-*.jar` 位于 Flink 版本的目录 `/lib` 中。
- `flink-table-uber-blink`：将上述 API 模块以及特定于 **Blink** 的模块打包到大多数 Table & SQL API 用例的分发中。默认情况下，**uber JAR** 文件 `flink-table-blink-*.jar` 位于 `/lib` Flink 版本的目录中。

有关如何在表程序中的新旧 **planner** 之间进行切换的更多信息，请参见[通用 API](#) 页面。

表程序依赖性

根据目标编程语言，您需要将 **Java** 或 **Scala** API 添加到项目中，以便使用 Table API 和 SQL 定义管道：

```
  
<!-- Either... -->  
<dependency>  
  <groupId>org.apache.flink</groupId>  
  <artifactId>flink-table-api-java-bridge_2.11</artifactId>  
  <version>1.9.0</version>  
  <scope>provided</scope>  
</dependency>  
<!-- or... -->
```

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-scala-bridge_2.11</artifactId>
  <version>1.9.0</version>
  <scope>provided</scope>
</dependency>
```



此外，如果要在 IDE 中本地运行 Table API 和 SQL 程序，则必须添加以下一组模块之一，具体取决于要使用的 planner：

```
<!-- Either... (for the old planner that was available before Flink
1.9) -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner_2.11</artifactId>
  <version>1.9.0</version>
  <scope>provided</scope>
</dependency>
<!-- or.. (for the new Blink planner) -->
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner-blink_2.11</artifactId>
  <version>1.9.0</version>
  <scope>provided</scope>
</dependency>
```



在内部，表生态系统的一部分在 **Scala** 中实现。因此，请确保为批处理和流应用程序都添加以下依赖项：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-streaming-scala_2.11</artifactId>
  <version>1.9.0</version>
  <scope>provided</scope>
</dependency>
```

扩展依赖

如果实现与 **Kafka** 或一组用户定义的函数进行交互的自定义格式，则以下依赖关系就足够了，并且可以用于 SQL Client 的 JAR 文件：

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-common</artifactId>
```

```
<version>1.9.0</version>
<scope>provided</scope>
</dependency>
```

当前，该模块包括以下扩展点：

- `SerializationSchemaFactory`
- `DeserializationSchemaFactory`
- `ScalarFunction`
- `TableFunction`
- `AggregateFunction`

接下来要去哪里？

- 概念和通用 API：表 API 和 SQL 的共享概念和 API。
- 数据类型：列出预定义的数据类型及其属性。
- 流概念：Table API 或 SQL 的流特定文档，例如时间属性的配置和更新结果的处理。
- 连接到外部系统：可用的连接器和格式，用于向外部系统读取和写入数据。
- Table API：Table API 支持的操作和 API。
- SQL：SQL 支持的操作和语法。
- 内置函数：Table API 和 SQL 支持的函数。
- SQL 客户端：使用 Flink SQL 并在没有编程知识的情况下将表程序提交给集群。

【翻译】Flink Table Api & SQL —— 概念与通用 API

本文翻译自官网：<https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/common.html>

Flink Table Api & SQL 翻译目录

Table API 和 SQL 集成在共同 API 中。该 API 的中心概念是 Table，用作查询的输入和输出。本文档介绍了使用 Table API 和 SQL 查询的程序的通用结构，如何注册 Table，如何查询 Table 以及如何发出 Table（数据）。

- 两个 planner 之间的主要区别
- 表 API 和 SQL 程序的结构
- 创建一个 TableEnvironment
- 在 Catalog 中注册表
 - 注册表格
 - 注册一个 TableSource
 - 注册一个 TableSink
- 注册扩展 Catalog
- 查询表
 - 表 API
 - SQL
 - 混合 表 API 和 SQL

- 发发表(数据)
- 翻译并执行查询
- 与 DataStream 和 DataSet API 集成
 - Scala 的隐式转换
 - 将 DataStream 或 DataSet 注册为表
 - 将 DataStream 或 DataSet 转换为表
 - 将表转换为 DataStream 或 DataSet
 - 数据类型到表结构的映射
- 查询优化
 - 解释表

两个 **planner** 之间的主要区别

1. Blink 将批处理作业视为流的特殊情况。因此，还不支持 Table 和 DataSet 之间的转换，并且批处理作业不会转换成 DataSet，而是像流作业一样转换为 DataStream 程序。
2. Blink planner 不支持 BatchTableSource，而是使用 bounded StreamTableSource 代替。
3. Blink planner 仅支持新的 Catalog，不支持 ExternalCatalog 它是不推荐使用的。
4. 为 old planner 和 blink planner 实现的 FilterableTableSource 是不相容的。old planner 会将 PlannerExpressions 下推到 FilterableTableSource，而 blink planner 将下推 Expressions（不懂什么意思：The old planner will push down PlannerExpressions into FilterableTableSource, while the Blink planner will push down Expressions.）。
5. 基于字符串的键值配置选项（有关详细信息，请参阅有关[配置](#)的文档）仅用于 Blink planner。
6. 两个 planner 的实现（CalciteConfig）PlannerConfig 不同。
7. Blink planner 会将多个接收器优化为一个 DAG（仅在 TableEnvironment 上支持，而不在 StreamTableEnvironment 上支持）。old planner 将始终将每个接收器优化为一个新的 DAG，其中所有 DAG 彼此独立。
8. old planner 现在不支持 catalog 统计信息，而 Blink planner 则支持。

表 **API** 和 **SQL** 程序的结构

用于批处理和流式传输的所有 Table API 和 SQL 程序都遵循相同的模式。以下代码示例显示了 Table API 和 SQL 程序的通用结构。

```
// create a TableEnvironment for specific planner batch or streaming
val tableEnv = ... // see "Create a TableEnvironment" section

// register a Table
tableEnv.registerTable("table1", ...) // or
```

```

tableEnv.registerTableSource("table2", ...)    // or
tableEnv.registerExternalCatalog("extCat", ...)
// register an output Table
tableEnv.registerTableSink("outputTable", ...);

// create a Table from a Table API query
val tapiResult = tableEnv.scan("table1").select(...)
// create a Table from a SQL query
val sqlResult  = tableEnv.sqlQuery("SELECT ... FROM table2 ...")

// emit a Table API result Table to a TableSink, same for SQL result
tapiResult.insertInto("outputTable")

// execute
tableEnv.execute("scala_job")

```



注意：表 API 和 SQL 查询可以轻松地与 `DataStream` 或 `DataSet` 程序集成并嵌入其中。请参阅[与 `DataStream` 和 `DataSet` API 集成](#)，以了解如何将 `DataStream` 和 `DataSet` 转换为 `Tables`，反之亦然。

创建一个 `TableEnvironment`

`TableEnvironment` 是 `Table API` 和 `SQL` 集成的中心概念。它负责：

- `Table` 在内部 `catalog` 中注册
- 注册外部 `catalog`
- 执行 `SQL` 查询
- 注册用户定义的（标量，表或聚合）函数
- 将 `DataStream` 或 `DataSet` 转换为 `Table`
- 持有对 `ExecutionEnvironment` 或 `StreamExecutionEnvironment` 的引用

`Table` 始终绑定到特定的 `TableEnvironment`。不可能在同一查询中组合不同 `TableEnvironments` 的表，例如，`join` 或 `union` 它们。

`TableEnvironment` 是通过调用 `StreamExecutionEnvironment` 或 `ExecutionEnvironment` 的静态方法 `BatchTableEnvironment.create()` 或 `StreamTableEnvironment.create()` 与可选的 `TableConfig` 创建的。该 `TableConfig` 可用于配置 `TableEnvironment` 或定制查询优化和翻译过程（参见[查询优化](#)）。

请务必选择特定的 `planner` `BatchTableEnvironment` / `StreamTableEnvironment` 与你的编程语言相匹配。

如果两个 **planner jar** 都在类路径上（默认行为），则应明确设置要在当前程序中使用的 **planner**。



```
// *****
// FLINK STREAMING QUERY
// *****

import
org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.table.api.EnvironmentSettings
import org.apache.flink.table.api.scala.StreamTableEnvironment

val fsSettings =
EnvironmentSettings.newInstance().useOldPlanner().inStreamingMode().build()
val fsEnv = StreamExecutionEnvironment.getExecutionEnvironment
val fsTableEnv = StreamTableEnvironment.create(fsEnv, fsSettings)
// or val fsTableEnv = TableEnvironment.create(fsSettings)

// *****
// FLINK BATCH QUERY
// *****

import org.apache.flink.api.scala.ExecutionEnvironment
import org.apache.flink.table.api.scala.BatchTableEnvironment

val fbEnv = ExecutionEnvironment.getExecutionEnvironment
val fbTableEnv = BatchTableEnvironment.create(fbEnv)

// *****
// BLINK STREAMING QUERY
// *****

import
org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.table.api.EnvironmentSettings
import org.apache.flink.table.api.scala.StreamTableEnvironment

val bsEnv = StreamExecutionEnvironment.getExecutionEnvironment
val bsSettings =
EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build()
val bsTableEnv = StreamTableEnvironment.create(bsEnv, bsSettings)
// or val bsTableEnv = TableEnvironment.create(bsSettings)

// *****
// BLINK BATCH QUERY
```

```
// *****
import org.apache.flink.table.api.{EnvironmentSettings,
TableEnvironment}

val bbSettings =
EnvironmentSettings.newInstance().useBlinkPlanner().inBatchMode().build()
val bbTableEnv = TableEnvironment.create(bbSettings)
```

注意：如果/lib 目录中只有一个计划器 jar，则可以使用 useAnyPlanner（use_any_planner 对于 python）创建 specific EnvironmentSettings。

[回到顶部](#)

在 catalog 中注册表

TableEnvironment 维护按名称注册的表的 catalog。表有两种类型，输入表和输出表。可以在 Table API 和 SQL 查询中引用输入表并提供输入数据。输出表可用于将表 API 或 SQL 查询的结果发送到外部系统。

输入表可以从各种 source 进行注册：

- 现有 Table 对象，通常是 Table API 或 SQL 查询的结果。
- TableSource，用于访问外部数据，例如文件，数据库或消息系统。
- DataStream 或 DataSet 从数据流（仅适用于流作业）或数据集（仅适用于 old planner 转换批处理作业）程序。[DataStream 和 DataSet API 的集成部分](#)中讨论了注册 DataStream 或 DataSet。

可以使用 TableSink 来注册输出表。

注册表格

在 TableEnvironment 中注册 Table 如下：

```
// get a TableEnvironment
val tableEnv = ... // see "Create a TableEnvironment" section

// table is the result of a simple projection query
val projTable: Table = tableEnv.scan("X").select(...)

// register the Table projTable as table "projectedTable"
tableEnv.registerTable("projectedTable", projTable)
```

注意：注册 Table 的处理方式与关系数据库系统中的 VIEW 相似，即，定义的查询 Table 未经过优化，但是当另一个查询引用已注册的查询时将内联 Table。如果多个查询引用同一个已


注册的查询 Table，则将为每个引用查询内联该查询并执行多次，即 Table 将不会共享已注册的结果。

注册 TableSource

TableSource 提供对存储在存储系统（例如数据库（MySQL，HBase 等）具有特定编码的文件（CSV，Apache [Parquet，Avro，ORC]等）或消息系统（Apache Kafka，RabbitMQ 等）中的外部数据的访问。

Flink 旨在为常见的数据格式和存储系统提供 TableSources。请查看“[表源和接收器](#)”页面，以获取受支持的 [TableSource](#) 的列表以及如何构建自定义的 TableSource。


在 TableEnvironment 中注册 TableSource 如下：



```
// get a TableEnvironment
val tableEnv = ... // see "Create a TableEnvironment" section

// create a TableSource
val csvSource: TableSource = new CsvTableSource("/path/to/file", ...)

// register the TableSource as table "CsvTable"
tableEnv.registerTableSource("CsvTable", csvSource)
```




注：TableEnvironment 用于 **blink planner** 只接受 StreamTableSource，LookupableTableSource 和 InputFormatTableSource，StreamTableSource 用于 **blink planner** 必须是有界的。

注册 TableSink

TableSink 可以使用注册的表将 [Table API 或 SQL 查询的结果](#) 发送到外部存储系统，例如数据库，键值存储(系统)，消息队列或文件系统（采用不同的编码，例如 CSV，Apache [Parquet，Avro，ORC]，...）。

Flink 旨在为常见的数据格式和存储系统提供 TableSink。请参阅有关“[表源和接收器](#)”页面的文档，以获取有关可用接收器的详细信息以及如何实现自定义的 TableSink。

在 TableEnvironment 中注册 TableSink 如下：



```
// get a TableEnvironment
val tableEnv = ... // see "Create a TableEnvironment" section

// create a TableSink
val csvSink: TableSink = new CsvTableSink("/path/to/file", ...)

// define the field names and types
val fieldNames: Array[String] = Array("a", "b", "c")
```

```
val fieldTypes: Array[TypeInformation[_]] = Array(Types.INT,
Types.STRING, Types.LONG)

// register the TableSink as table "CsvSinkTable"
tableEnv.registerTableSink("CsvSinkTable", fieldNames, fieldTypes,
csvSink)
```



注册外部 catalog

外部 **catalog** 可以提供有关外部数据库和表的信息，例如它们的名称，结构，统计信息，以及有关如何访问存储在外部数据库、表或文件中的数据的信息。

可以通过实现 **ExternalCatalog** 接口来创建外部 **catalog**，并在 **TableEnvironment** 中对其进行注册，如下所示：

```
// get a TableEnvironment
val tableEnv = ... // see "Create a TableEnvironment" section

// create an external catalog
val catalog: ExternalCatalog = new InMemoryExternalCatalog

// register the ExternalCatalog catalog
tableEnv.registerExternalCatalog("InMemCatalog", catalog)
```



在 **TableEnvironment** 中注册后，可以通过指定表的完整路径（例如 **catalog.database.table**）从 **Table API** 或 **SQL** 查询中访问在 **ExternalCatalog** 中定义的所有表。

目前，Flink 提供了一个 **InMemoryExternalCatalog** 用于演示和测试目的的工具。但是，该 **ExternalCatalog** 接口还可用于将 **HCatalog** 或 **Metastore** 之类的 **catalog** 连接到 **Table API**。

注意：blink planner 不支持外部 **catalog**。

查询表

表 API

Table API 是用于 **Scala** 和 **Java** 的语言集成查询 **API**。与 **SQL** 相反，查询未指定为字符串，而是以宿主语言逐步构成。

该 **API** 基于 **Table** 类，**Table** 类代表一个表（流或批的），并提供应用关系操作的方法。这些方法返回一个新的 **Table** 对象，该对象表示对输入 **Table** 应用关系操作的结果。某些关系操作由多个方法调用组成，例如 **table.groupBy (...)** **select ()**，其中 **groupBy (...)** 指定表的分组，并 **select (...)** 分组的投影表。

Table API 文档描述了流和批处理表支持的所有 Table API 操作。

以下示例显示了一个简单的 Table API 聚合查询：



```
// get a TableEnvironment
val tableEnv = ... // see "Create a TableEnvironment" section

// register Orders table

// scan registered Orders table
val orders = tableEnv.scan("Orders")
// compute revenue for all customers from France
val revenue = orders
    .filter('cCountry === "FRANCE")
    .groupBy('cID, 'cName)
    .select('cID, 'cName, 'revenue.sum AS 'revSum)

// emit or convert Table
// execute query
```



注意：Scala Table API 使用 Scala 符号，这些符号以单个记号（'）开头来引用的 Table 属性。Table API 使用 Scala 隐式转换。为了使用 Scala 隐式转换确保导入了 `org.apache.flink.api.scala._` 和 `org.apache.flink.table.api.scala._`。

SQL

Flink 的 SQL 集成基于实现 SQL 标准的 Apache Calcite。SQL 查询被指定为常规字符串。

SQL 文件描述 flink SQL 支持的流和批的表。

以下示例说明如何指定查询并以返回用表 表示的结果。



```
// get a TableEnvironment
val tableEnv = ... // see "Create a TableEnvironment" section

// register Orders table

// compute revenue for all customers from France
val revenue = tableEnv.sqlQuery("""
|SELECT cID, cName, SUM(revenue) AS revSum
|FROM Orders
|WHERE cCountry = 'FRANCE'
|GROUP BY cID, cName
```

```
"".stripMargin)

// emit or convert Table
// execute query
```



下面的示例演示如何指定将更新查询的结果插入到已注册表中。

```
// get a TableEnvironment
val tableEnv = ... // see "Create a TableEnvironment" section

// register "Orders" table
// register "RevenueFrance" output table

// compute revenue for all customers from France and emit to
"RevenueFrance"
tableEnv.sqlUpdate("""
  |INSERT INTO RevenueFrance
  |SELECT cID, cName, SUM(revenue) AS revSum
  |FROM Orders
  |WHERE cCountry = 'FRANCE'
  |GROUP BY cID, cName
  |""").stripMargin)

// execute query
```



混合 Table API 和 SQL

Table API 和 SQL 查询可以轻松混合，因为它们都返回 Table 对象：

- 可以在 Table API 的查询可以定义在 SQL 查询返回的 Table 对象上。
- 通过在 TableEnvironment 中注册结果表并在 SQL 查询的 FROM 子句中引用它，可以对 Table API 查询的结果定义 SQL 查询。（好绕：A SQL query can be defined on the result of a Table API query by registering the resulting Table in the TableEnvironment and referencing it in the FROM clause of the SQL query）

发出 Table


通过将表写入 TableSink 来发出表。 TableSink 是通用接口，用于支持各种文件格式（例如 CSV, Apache Parquet, Apache Avro），存储系统（例如 JDBC, Apache HBase, Apache Cassandra, Elasticsearch）或消息系统（例如 Apache Kafka, RabbitMQ）。

批处理表只能写入 `BatchTableSink`，而流表则需要 `AppendStreamTableSink`、`RetractStreamTableSink` 或 `UpsertStreamTableSink`。

请参阅有关表源和接收器的文档，以获取有关可用接收器的详细信息以及有关如何实现自定义 `TableSink` 的说明。

`Table.insertInto (String tableName)` 方法将表发射到已注册的 `TableSink`。该方法通过名称从 `catalog` 中查找 `TableSink`，并验证 `Table` 的结构与 `TableSink` 的结构是否相同。

以下示例显示如何发出表：



```
// get a TableEnvironment
val tableEnv = ... // see "Create a TableEnvironment" section


// create a TableSink
val sink: TableSink = new CsvTableSink("/path/to/file", fieldDelim =
"|")

// register the TableSink with a specific schema
val fieldNames: Array[String] = Array("a", "b", "c")
val fieldTypes: Array[TypeInformation] = Array(Types.INT,
Types.STRING, Types.LONG)
tableEnv.registerTableSink("CsvSinkTable", fieldNames, fieldTypes,
sink)

// compute a result Table using Table API operators and/or SQL
queries
val result: Table = ...

// emit the result Table to the registered TableSink
result.insertInto("CsvSinkTable")

// execute the program
```



翻译并执行查询

对于两个 `planner` 来说，翻译和执行查询的行为是不同的。

Old planner

根据 `Table API` 和 `SQL` 查询的输入是流输入还是批处理输入，它们将转换为 `DataStream` 或 `DataSet` 程序。查询被内部表示为一个逻辑查询计划，并在两个阶段被转换：

1. 优化逻辑计划
2. 转换为 `DataStream` 或 `DataSet` 程序

在以下情况下，将转换 Table API 或 SQL 查询：

- 将 Table 发射到 TableSink，即当 Table.insertInto() 被调用。
- 指定 SQL 更新查询，即当 TableEnvironment.sqlUpdate() 调用。
- 将 Table 转换为 DataStream 或 DataSet（请参阅[与 DataStream 和 DataSet API 集成](#)）。

转换后，将像常规 DataStream 或 DataSet 程序一样处理 Table API 或 SQL 查询，并在调用 StreamExecutionEnvironment.execute() 或 ExecutionEnvironment.execute() 时执行。

Blink planner

无论 Table API 和 SQL 查询的输入是流传输还是批处理，都将转换为 [DataStream](#) 程序。查询被内部表示为一个逻辑查询计划，并在两个阶段被转换：

1. 优化逻辑计划，
2. 转换为 DataStream 程序。

翻译查询的行为 TableEnvironment 和 StreamTableEnvironment 是不同的。

对于 TableEnvironment，Table API 或 SQL 查询在 TableEnvironment.execute() 调用时被转换，因为 TableEnvironment 将优化多个接收器为一个 DAG。

而对于 StreamTableEnvironment，在以下情况下会转换 Table API 或 SQL 查询：

- 将 Table 发射到 TableSink，即当 Table.insertInto() 被调用。
- 指定 SQL 更新查询，即当 TableEnvironment.sqlUpdate() 被调用。
- 将 Table 转换为 DataStream。

转换后，将像常规的 DataStream 程序一样处理 Table API 或 SQL 查询，并在调用 TableEnvironment.execute() 或 StreamExecutionEnvironment.execute() 时执行。

DataStream 和 DataSet API 集成

以下示例显示如何发出表：流上的两个 planner 都可以与 DataStream API 集成。只有 old planner 才能与 DataSet API 集成，批量 blink planner 不能同时与两者结合。注意：下面讨论的 DataSet API 仅与批量使用的 old planner 有关。

Table API 和 SQL 查询可以轻松地与 DataStream 和 DataSet 程序集成并嵌入其中。例如，可以查询外部表（例如从 RDBMS），进行一些预处理，例如过滤，投影，聚合或与元数据联接，然后使用 DataStream 或 DataSet API（以及在这些 API 之上构建的任何库，例如 CEP 或 Gelly）。相反，也可以将 Table API 或 SQL 查询应用于 DataStream 或 DataSet 程序的结果。


可以通过将 DataStream 或 DataSet 转换为 Table 来实现这种交互，反之亦然。在本节中，我们描述如何完成这些转换。

Scala 的隐式转换

Scala Table API 具有对 `DataSet`、`DataStream` 和 `Table` 类的隐式转换。 通过为 Scala `DataStream` API 导入 `org.apache.flink.table.api.scala._` 以及 `org.apache.flink.api.scala._`，可以启用这些转换。

将 `DataStream` 或 `DataSet` 注册为表

可以在 `TableEnvironment` 中将 `DataStream` 或 `DataSet` 注册为表。 结果表的模式取决于已注册的 `DataStream` 或 `DataSet` 的数据类型。 请检查有关将[数据类型映射到表模式](#)的部分，以获取详细信息。




```
// get TableEnvironment
// registration of a DataSet is equivalent
val tableEnv: StreamTableEnvironment = ... // see "Create a
TableEnvironment" section

val stream: DataStream[(Long, String)] = ...

// register the DataStream as Table "myTable" with fields "f0", "f1"
tableEnv.registerDataStream("myTable", stream)


// register the DataStream as table "myTable2" with fields "myLong",
"myString"
tableEnv.registerDataStream("myTable2", stream, 'myLong, 'myString)
```



注意: `DataStream` 表的名称不能与 `^_DataStreamTable_[0-9]+` 模式匹配，并且 `DataSet` 表的名称不能与 `^_DataSetTable_[0-9]+` 模式匹配。 这些模式仅供内部使用。

将 `DataStream` 或 `DataSet` 转换为表

除了在 `TableEnvironment` 中注册 `DataStream` 或 `DataSet` 之外，还可以将其直接转换为 `Table`。 如果要在 `Table` API 查询中使用 `Table`，这将很方便。



```
// get TableEnvironment
// registration of a DataSet is equivalent
val tableEnv = ... // see "Create a TableEnvironment" section

val stream: DataStream[(Long, String)] = ...

// convert the DataStream into a Table with default fields '_1, '_2
val table1: Table = tableEnv.fromDataStream(stream)

// convert the DataStream into a Table with fields 'myLong, 'myString
val table2: Table = tableEnv.fromDataStream(stream, 'myLong,
'myString)
```



将表转换为 **DataStream** 或 **DataSet**

可以将表转换为 **DataStream** 或 **DataSet**。这样，可以在 **Table API** 或 **SQL** 查询的结果上运行自定义 **DataStream** 或 **DataSet** 程序。

将表转换为 **DataStream** 或 **DataSet** 时，需要指定生成的 **DataStream** 或 **DataSet** 的数据类型，即，将表的行转换为的数据类型。最方便的转换类型通常是 **Row**。以下列表概述了不同选项的功能：

- **Row**：字段按位置，任意数量的字段，支持 **null** 值，无类型安全访问的方式映射。
- **POJO**：字段按名称映射（**POJO** 字段必须命名为 **Table** 字段），任意数量的字段，支持 **null** 值，类型安全访问。
- **case class**：字段按位置映射，不支持 **null** 值，类型安全访问。
- **tuple**：按位置映射字段，限制为 22（**Scala**）或 25（**Java**）字段，不支持 **null** 值，类型安全访问。
- **原子类型**：**Table** 必须具有单个字段，不支持 **null** 值，类型安全访问。

将 *Table* 转换为 *DataStream*

流式查询结果产生的表将动态更新，即随着新记录到达查询的输入流不断变化。因此，将这种动态查询转换成的 **DataStream** 需要对表的更新进行编码。

有两种模式可以将 **Table** 转换为 **DataStream**：

1. **追加模式**：仅当动态表仅通过 **INSERT** 更改进行修改时才可以使用此模式，即，它仅是追加操作，并且以前发出的结果不更新。
2. **撤回模式**：始终可以使用此模式。它使用布尔标志对 **INSERT** 和 **DELETE** 更改进行编码。



```
// get TableEnvironment.
// registration of a DataSet is equivalent
val tableEnv: StreamTableEnvironment = ... // see "Create a
TableEnvironment" section

// Table with two fields (String name, Integer age)
val table: Table = ...

// convert the Table into an append DataStream of Row
val dsRow: DataStream[Row] = tableEnv.toAppendStream[Row](table)

// convert the Table into an append DataStream of Tuple2[String, Int]
val dsTuple: DataStream[(String, Int)] dsTuple =
    tableEnv.toAppendStream[(String, Int)](table)
```



```
// convert the Table into a retract DataStream of Row.  
//   A retract stream of type X is a DataStream[(Boolean, X)].  
//   The boolean field indicates the type of the change.  
//   True is INSERT, false is DELETE.  
val retractStream: DataStream[(Boolean, Row)] =  
tableEnv.toRetractStream[Row](table)
```



注意：有关动态表及其属性的详细讨论，请参见“[动态表](#)”文档。

将 *Table* 转换为 *DataSet*

Table 转换为 DataSet 如下：



```
val tableEnv = BatchTableEnvironment.create(env)  
  
// Table with two fields (String name, Integer age)  
val table: Table = ...  
  
// convert the Table into a DataSet of Row  
val dsRow: DataSet[Row] = tableEnv.toDataSet[Row](table)  
  
// convert the Table into a DataSet of Tuple2[String, Int]  
val dsTuple: DataSet[(String, Int)] = tableEnv.toDataSet[(String,  
Int)](table)
```



数据类型到表结构的映射

Flink 的 `DataStream` 和 `DataSet` API 支持非常多种类型。元组（内置 Scala 和 Flink Java 元组），POJO，Scala case class 和 Flink 的 `Row` 类型等复合类型，允许嵌套的数据结构具有多个字段，这些字段可在表达式中访问。其他类型被视为原子类型。在下文中，我们描述 `Table` API 如何将这些类型转换为内部行表示形式，并显示将 `DataStream` 转换的 `Table` 示例。

数据类型到表模式的映射可以使用两种方式：基于字段位置或基于字段名称。

基于位置的映射

基于位置的映射可用于在保持字段顺序的同时为字段提供更有意义的名称。此映射可用于具有定义的字段顺序的复合数据类型以及原子类型。元组，行和案例类等复合数据类型具有这样的字段顺序。但是，必须根据字段名称映射 POJO 的字段（请参阅下一节）。字段可以投影出来，但不能用别名 `as` 重命名。

定义基于位置的映射时，输入数据类型中一定不能存在指定的名称，否则 API 会假定应该基于字段名称进行映射。如果未指定任何字段名称，则使用复合类型的默认字段名称和字段顺序，或者原子类型使用 `f0`。



```
// get a TableEnvironment
val tableEnv: StreamTableEnvironment = ... // see "Create a
TableEnvironment" section

val stream: DataStream[(Long, Int)] = ...

// convert DataStream into Table with default field names "_1" and
"_2"
val table: Table = tableEnv.fromDataStream(stream)

// convert DataStream into Table with field "myLong" only
val table: Table = tableEnv.fromDataStream(stream, 'myLong)

// convert DataStream into Table with field names "myLong" and
"myInt"
val table: Table = tableEnv.fromDataStream(stream, 'myLong, 'myInt)
```



基于名称的映射

基于名称的映射可用于任何数据类型，包括 **POJO**。这是定义表模式映射的最灵活的方法。映射中的所有字段均按名称引用，并且可以使用别名 **as** 重命名。字段可以重新排序和投影。

如果未指定任何字段名称，则使用复合类型的默认字段名称和字段顺序，或者原子类型使用 **f0**。



```
// get a TableEnvironment
val tableEnv: StreamTableEnvironment = ... // see "Create a
TableEnvironment" section

val stream: DataStream[(Long, Int)] = ...

// convert DataStream into Table with default field names "_1" and
"_2"
val table: Table = tableEnv.fromDataStream(stream)

// convert DataStream into Table with field "_2" only
val table: Table = tableEnv.fromDataStream(stream, '_2)

// convert DataStream into Table with swapped fields
val table: Table = tableEnv.fromDataStream(stream, '_2, '_1)

// convert DataStream into Table with swapped fields and field names
"myInt" and "myLong"
```

```
val table: Table = tableEnv.fromDataStream(stream, '_2 as 'myInt, '_1
as 'myLong)
```



原子类型

Flink 将基本类型（整数，双精度型，字符串）或泛型（无法分析和分解的类型）视为原子类型。原子类型的 **DataStream** 或 **DataSet** 转换为具有单个属性的表。从原子类型推断出属性的类型，并且可以指定属性的名称。



```
// get a TableEnvironment
val tableEnv: StreamTableEnvironment = ... // see "Create a
TableEnvironment" section

val stream: DataStream[Long] = ...

// convert DataStream into Table with default field name "f0"
val table: Table = tableEnv.fromDataStream(stream)

// convert DataStream into Table with field name "myLong"
val table: Table = tableEnv.fromDataStream(stream, 'myLong)
```



元组 (Scala 和 Java) 和 case 类 (仅 Scala)

Flink 支持 Scala 的内置元组，并为 Java 提供了自己的元组类。两种元组的 **DataStream**s 和 **DataSet** 都可以转换为表。可以通过提供所有字段的名称来重命名字段（根据位置进行映射）。如果未指定任何字段名称，则使用默认字段名称。如果引用了原始字段名称（Flink 元组为 `f0`, `f1`, ..., Scala 元组为 `_1`, `_2`, ...），则 API 会假定映射是基于名称的，而不是基于位置的。基于名称的映射允许使用别名（`as`）对字段和投影进行重新排序。



```
// get a TableEnvironment
val tableEnv: StreamTableEnvironment = ... // see "Create a
TableEnvironment" section

val stream: DataStream[(Long, String)] = ...

// convert DataStream into Table with renamed default field names
'_1, '_2
val table: Table = tableEnv.fromDataStream(stream)

// convert DataStream into Table with field names "myLong",
"myString" (position-based)
val table: Table = tableEnv.fromDataStream(stream, 'myLong,
'myString)
```

```

// convert DataStream into Table with reordered fields "_2", "_1"
(name-based)
val table: Table = tableEnv.fromDataStream(stream, '_2, '_1)

// convert DataStream into Table with projected field "_2" (name-
based)
val table: Table = tableEnv.fromDataStream(stream, '_2)

// convert DataStream into Table with reordered and aliased fields
"myString", "myLong" (name-based)
val table: Table = tableEnv.fromDataStream(stream, '_2 as 'myString,
'_1 as 'myLong)

// define case class
case class Person(name: String, age: Int)
val streamCC: DataStream[Person] = ...

// convert DataStream into Table with default field names 'name, 'age
val table = tableEnv.fromDataStream(streamCC)

// convert DataStream into Table with field names 'myName, 'myAge
(position-based)
val table = tableEnv.fromDataStream(streamCC, 'myName, 'myAge)

// convert DataStream into Table with reordered and aliased fields
"myAge", "myName" (name-based)
val table: Table = tableEnv.fromDataStream(stream, 'age as 'myAge,
'name as 'myName)

```

POJO (Java and Scala)

Flink 支持 POJO 作为复合类型。确定 POJO 的规则[在此文档](#)。

在不指定字段名称的情况下将 POJO DataStream 或 DataSet 转换为 Table 时，将使用原始 POJO 字段的名称。名称映射需要原始名称，并且不能按位置进行。可以使用别名（使用 as 关键字）对字段进行重命名，重新排序和投影。

```

// get a TableEnvironment
val tableEnv: StreamTableEnvironment = ... // see "Create a
TableEnvironment" section

// Person is a POJO with field names "name" and "age"
val stream: DataStream[Person] = ...

```

```
// convert DataStream into Table with default field names "age",
"name" (fields are ordered by name!)
val table: Table = tableEnv.fromDataStream(stream)

// convert DataStream into Table with renamed fields "myAge",
"myName" (name-based)
val table: Table = tableEnv.fromDataStream(stream, 'age as 'myAge,
'name as 'myName)

// convert DataStream into Table with projected field "name" (name-
based)
val table: Table = tableEnv.fromDataStream(stream, 'name)

// convert DataStream into Table with projected and renamed field
"myName" (name-based)
val table: Table = tableEnv.fromDataStream(stream, 'name as 'myName)
```



Row

该 Row 数据类型支持字段和字段与任意数量的 null 值。字段名称可以通过指定 RowTypeInfo 或转化时 Row DataStream 或 DataSet 成 Table。行类型支持按位置和名称映射字段。可以通过提供所有字段的名称（基于位置的映射）来重命名字段，也可以为投影/排序/重命名（基于名称的映射）单独选择字段。



```
// get a TableEnvironment
val tableEnv: StreamTableEnvironment = ... // see "Create a
TableEnvironment" section

// DataStream of Row with two fields "name" and "age" specified in
`RowTypeInfo`
val stream: DataStream[Row] = ...

// convert DataStream into Table with default field names "name",
"age"
val table: Table = tableEnv.fromDataStream(stream)

// convert DataStream into Table with renamed field names "myName",
"myAge" (position-based)
val table: Table = tableEnv.fromDataStream(stream, 'myName, 'myAge)

// convert DataStream into Table with renamed fields "myName",
"myAge" (name-based)
```

```
val table: Table = tableEnv.fromDataStream(stream, 'name as 'myName,
'age as 'myAge)

// convert DataStream into Table with projected field "name" (name-
based)
val table: Table = tableEnv.fromDataStream(stream, 'name)

// convert DataStream into Table with projected and renamed field
"myName" (name-based)
val table: Table = tableEnv.fromDataStream(stream, 'name as 'myName)
```



查询优化

Old planner

Apache Flink 利用 Apache Calcite 来优化和翻译查询。当前执行的优化包括投影和过滤器下推，子查询去相关以及其他类型的查询重写。Old Planner 尚未优化联接的顺序，而是按照查询中定义的顺序执行它们（FROM 子句中的表顺序和/或 WHERE 子句中的连接谓词顺序）。

Blink planner

Apache Flink 利用并扩展了 Apache Calcite 来执行复杂的查询优化。这包括一系列基于规则和成本的优化，例如：

- 基于 Apache Calcite 的子查询解相关
- 计划修剪
- 分区修剪
- 过滤器下推
- 子计划重复数据删除避免重复计算
- 特殊的子查询重写，包括两个部分：
 - 将 IN 和 EXISTS 转换为左半联接
 - 将 NOT IN 和 NOT EXISTS 转换为左反联接
- 可选 join 重新排序
 - 通过启用 `table.optimizer.join-reorder-enabled`

注意：IN / EXISTS / NOT IN / NOT EXISTS 当前仅在子查询重写的结合条件下受支持。

优化器不仅基于计划，而且还基于可从数据源获得的丰富统计信息以及每个 operator（例如 io, cpu, 网络和内存）的细粒度成本来做出明智的决策。

高级用户可以通过 CalciteConfig 对象提供自定义优化，该对象可以通过调用提供给表环境 `TableEnvironment#getConfig#setPlannerConfig`。

解释表

Table API 提供了一种机制来解释计算表的逻辑和优化查询计划。 这是通过 `TableEnvironment.explain(table)` 方法或 `TableEnvironment.explain()` 方法完成的。 `explain(table)` 返回给定表的计划。 `describe()` 返回多接收器计划的结果，主要用于 Blink planner。 它返回一个描述三个计划的字符串：

1. 关系查询的抽象语法树，即未优化的逻辑查询计划
2. 优化的逻辑查询计划
3. 实际执行计划

以下代码显示了一个示例以及使用 `explain(table)` 给定 Table 的相应输出：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tEnv = StreamTableEnvironment.create(env)

val table1 = env.fromElements((1, "hello")).toTable(tEnv, 'count,
'word)
val table2 = env.fromElements((1, "hello")).toTable(tEnv, 'count,
'word)
val table = table1
    .where('word.like("F%"))
    .unionAll(table2)

val explanation: String = tEnv.explain(table)
println(explanation)
```

```
== Abstract Syntax Tree ==
LogicalUnion(all=[true])
  LogicalFilter(condition=[LIKE($1, _UTF-16LE'F%')])
    FlinkLogicalDataStreamScan(id=[1], fields=[count, word])
    FlinkLogicalDataStreamScan(id=[2], fields=[count, word])

== Optimized Logical Plan ==
DataStreamUnion(all=[true], union all=[count, word])
  DataStreamCalc(select=[count, word], where=[LIKE(word, _UTF-
16LE'F%')])
    DataStreamScan(id=[1], fields=[count, word])
    DataStreamScan(id=[2], fields=[count, word])

== Physical Execution Plan ==
Stage 1 : Data Source
  content : collect elements with CollectionInputFormat

Stage 2 : Data Source
```

```

    content : collect elements with CollectionInputFormat

    Stage 3 : Operator
      content : from: (count, word)
      ship_strategy : REBALANCE

    Stage 4 : Operator
      content : where: (LIKE(word, _UTF-16LE'F%')), select:
(count, word)
      ship_strategy : FORWARD

    Stage 5 : Operator
      content : from: (count, word)
      ship_strategy : REBALANCE

```



以下代码显示了一个示例以及使用 `explain ()` 的多 sink 计划的相应输出:

```

val settings =
EnvironmentSettings.newInstance.useBlinkPlanner.inStreamingMode.build
val tEnv = TableEnvironment.create(settings)

val fieldNames = Array("count", "word")
val fieldTypes = Array[TypeInformation[_]](Types.INT, Types.STRING)
tEnv.registerTableSource("MySource1", new
CsvTableSource("/source/path1", fieldNames, fieldTypes))
tEnv.registerTableSource("MySource2", new
CsvTableSource("/source/path2", fieldNames, fieldTypes))
tEnv.registerTableSink("MySink1", new
CsvTableSink("/sink/path1").configure(fieldNames, fieldTypes))
tEnv.registerTableSink("MySink2", new
CsvTableSink("/sink/path2").configure(fieldNames, fieldTypes))

val table1 = tEnv.scan("MySource1").where("LIKE(word, 'F%')")
table1.insertInto("MySink1")

val table2 = table1.unionAll(tEnv.scan("MySource2"))
table2.insertInto("MySink2")

val explanation = tEnv.explain(false)
println(explanation)

```



多 sink 计划的结果是



```
== Abstract Syntax Tree ==
LogicalSink(name=[MySink1], fields=[count, word])
+- LogicalFilter(condition=[LIKE($1, _UTF-16LE'F%')])
   +- LogicalTableScan(table=[[default_catalog, default_database,
MySource1, source: [CsvTableSource(read fields: count, word)]]])

LogicalSink(name=[MySink2], fields=[count, word])
+- LogicalUnion(all=[true])
   :- LogicalFilter(condition=[LIKE($1, _UTF-16LE'F%')])
   : +- LogicalTableScan(table=[[default_catalog, default_database,
MySource1, source: [CsvTableSource(read fields: count, word)]]])
   +- LogicalTableScan(table=[[default_catalog, default_database,
MySource2, source: [CsvTableSource(read fields: count, word)]]])

== Optimized Logical Plan ==
Calc(select=[count, word], where=[LIKE(word, _UTF-16LE'F%')],
reuse_id=[1])
+- TableSourceScan(table=[[default_catalog, default_database,
MySource1, source: [CsvTableSource(read fields: count, word)]]],
fields=[count, word])

Sink(name=[MySink1], fields=[count, word])
+- Reused(reference_id=[1])

Sink(name=[MySink2], fields=[count, word])
+- Union(all=[true], union=[count, word])
   :- Reused(reference_id=[1])
   +- TableSourceScan(table=[[default_catalog, default_database,
MySource2, source: [CsvTableSource(read fields: count, word)]]],
fields=[count, word])

== Physical Execution Plan ==
Stage 1 : Data Source
  content : collect elements with CollectionInputFormat

  Stage 2 : Operator
    content : CsvTableSource(read fields: count, word)
    ship_strategy : REBALANCE

    Stage 3 : Operator
      content : SourceConversion(table:Buffer(default_catalog,
default_database, MySource1, source: [CsvTableSource(read fields:
count, word)]), fields:(count, word))
```

```

        ship_strategy : FORWARD

    Stage 4 : Operator
        content : Calc(where: (word LIKE _UTF-16LE'F%'),
select: (count, word))
        ship_strategy : FORWARD

    Stage 5 : Operator
        content : SinkConversionToRow
        ship_strategy : FORWARD

    Stage 6 : Operator
        content : Map
        ship_strategy : FORWARD

Stage 8 : Data Source
    content : collect elements with CollectionInputFormat

    Stage 9 : Operator
        content : CsvTableSource(read fields: count, word)
        ship_strategy : REBALANCE

    Stage 10 : Operator
        content : SourceConversion(table:Buffer(default_catalog,
default_database, MySource2, source: [CsvTableSource(read fields:
count, word)]), fields:(count, word))
        ship_strategy : FORWARD

    Stage 12 : Operator
        content : SinkConversionToRow
        ship_strategy : FORWARD

    Stage 13 : Operator
        content : Map
        ship_strategy : FORWARD

    Stage 7 : Data Sink
        content : Sink: CsvTableSink(count, word)
        ship_strategy : FORWARD

    Stage 14 : Data Sink
        content : Sink: CsvTableSink(count, word)
        ship_strategy : FORWARD

```



【翻译】Flink Table Api & SQL —Streaming 概念

本文翻译自官网：Streaming 概念 <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/streaming/>

Flink Table Api & SQL 翻译目录

Flink 的 Table API 和 SQL 支持 是用于批处理和流处理的统一 API。这意味着 Table API 和 SQL 查询具有相同的语义，无论它们的输入是有界批处理输入还是无界流输入。因为关系代数和 SQL 最初是为批处理而设计的，所以对无边界流输入的关系查询不如对有边界批输入的关系查询好。

以下页面介绍了 Flink 的关系 API 在流数据上的概念，实际限制和流特定的配置参数。

接下来要去哪里？

- 动态表：描述动态表的概念。
- 时间属性：说明时间属性以及如何在 Table API 和 SQL 中处理时间属性。
- 连续查询中的 join：连续查询中支持的不同连接类型。
- 临时表：描述临时表的概念。
- 查询配置：列出 Table API 和 SQL 特定的配置选项。

【翻译】Flink Table Api & SQL —Streaming 概念 ——动态表

本文翻译自官网：Flink Table Api & SQL 动态

表 https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/streaming/dynamic_tables.html

Flink Table Api & SQL 翻译目录

SQL 和关系代数在设计时并未考虑流数据。所以，关系代数（和 SQL）与流处理之间在概念上有一些差距。

本页讨论了这些差异，并说明了 Flink 如何在无界数据上实现与常规数据库引擎在有界数据上相同的语义。

- 数据流上的关系查询
- 动态表和连续查询
- 在流上定义表
 - 连续查询
 - 更新和追加查询
 - 查询限制
- 表到流的转换

数据流上的关系查询

下表针对输入数据、执行和输出结果，比较了传统的关系代数和流处理之间的差异。

关系代数/ SQL	流处理
关系（或表）是有界的（多个）元组。	流是无限的元组序列。
对批处理数据（例如，关系数据库中的表）执行的查询可以访问完整的输入数据。	流查询在启动时无法访问所有数据，而必须“等待”以流式传输数据。
批处理查询产生固定大小的结果后终止。	流查询根据接收到的记录不断更新其结果，并且永远不会完成。

尽管存在这些差异，使用关系查询和 SQL 处理流并不是不可能的。先进的关系数据库系统提供了称为“物化视图”的功能（将视图的结果作为表存储起来，并定时更新，Oracle 有，百度 Oracle 物化视图）。物化视图被定义为 SQL 查询，就像常规虚拟视图一样。与虚拟视图相反，物化视图缓存查询结果，以便在访问视图时无需评估查询。缓存的一个常见挑战是防止缓存提供过时的结果。修改其定义查询的基表时，物化视图将过时。Eager View Maintenance 是一种在其基表更新后立即更新物化视图的技术。

如果考虑以下因素，那么 Eager View Maintenance 和对流进行 SQL 查询之间的联系将变得显而易见：

- 数据库表是一个结果流的 INSERT，UPDATE 和 DELETE DML 语句，通常被称为更新日志流。
- 物化视图定义为 SQL 查询。为了更新视图，查询会连续处理视图基本关系的变更日志流。
- 实例化视图是流式 SQL 查询的结果。

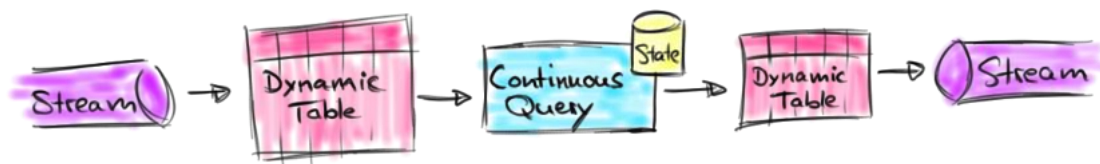
考虑到这些要点，我们将在下一节介绍动态表的以下概念。

动态表和持续查询

动态表是 Flink 的 Table API 和 SQL 对流数据支持的核心概念。与代表批处理数据的静态表相反，动态表随时间而变化。可以像静态批处理表一样查询它们。查询动态表会产生一个持续查询。持续查询永远不会终止并产生动态表作为结果。查询不断更新其（动态）结果表以反映其（动态）输入表上的更改。本质上，对动态表的持续查询与定义物化视图的查询非常相似。

重要的是要注意，持续查询的结果在语义上始终等效于在输入表的快照上以批处理方式执行的同一查询的结果。

下图显示了流，动态表和持续查询之间的关系：



1. 流将转换为动态表。
2. 在动态表上评估持续查询，生成新的动态表。
3. 生成的动态表将转换回流。

注意：动态表首先是一个逻辑概念。在查询执行过程中不一定（完全）实现动态表。

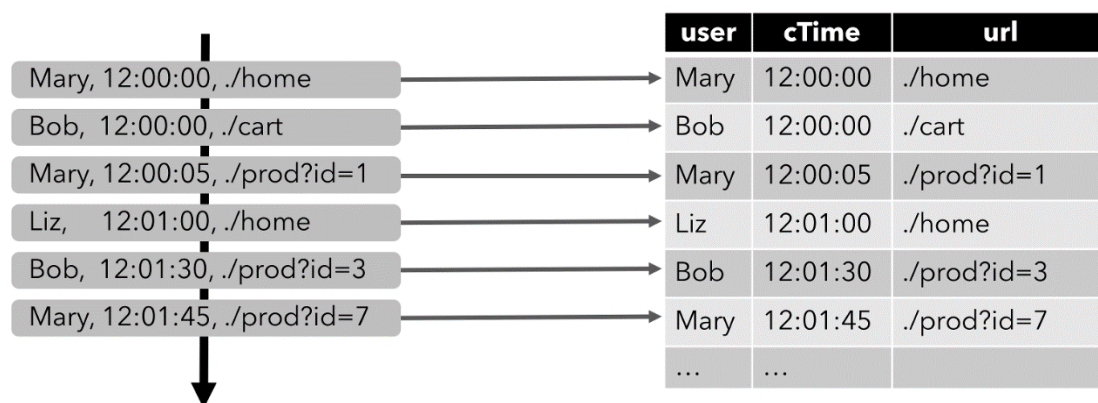
在下文中，我们将通过具有以下模式的单击事件流来解释动态表和持续查询的概念：

```
[
  user:  VARCHAR,    // the name of the user
  cTime: TIMESTAMP,  // the time when the URL was accessed
  url:   VARCHAR     // the URL that was accessed by the user
]
```

在流上定义表

为了使用关系查询处理流，必须将其转换为 Table。从概念上讲，流的每个记录都被解释为 INSERT 对结果表的修改。本质上，我们是从仅 INSERT 的 changelog 流中构建表。

下图可视化了点击事件流（左侧）如何转换为表格（右侧）。随着插入更多点击流记录，结果表将持续增长。



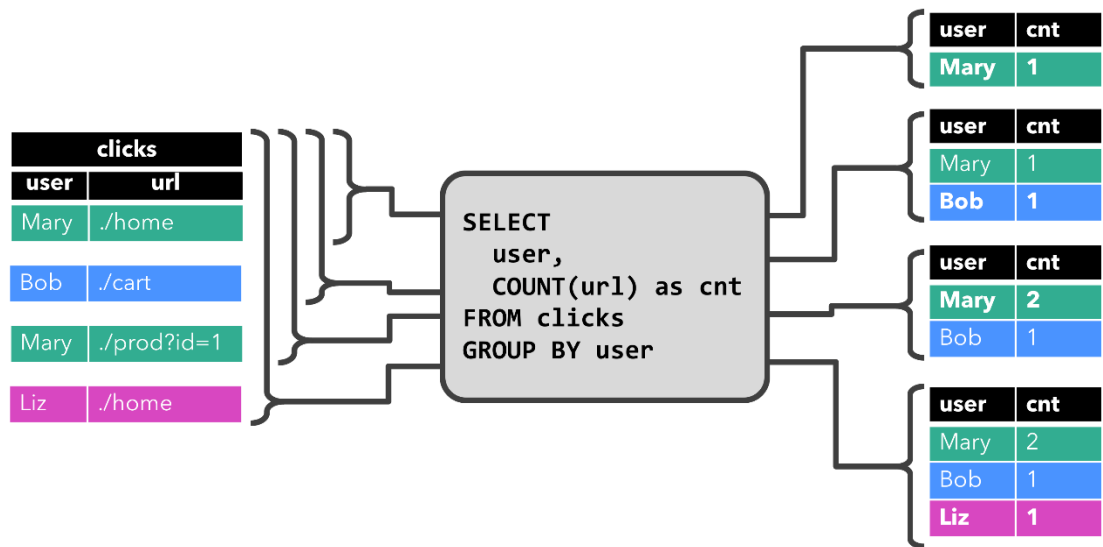
注意：在流上定义的表在内部未实现。

持续查询

在动态表上评估持续查询，并生成一个新的动态表作为结果。与批处理查询相反，持续查询永远不会终止并根据其输入表的更新来更新其结果表。在任何时间点，持续查询的结果在语义上均等同于在输入表的快照上以批处理模式执行同一查询的结果。

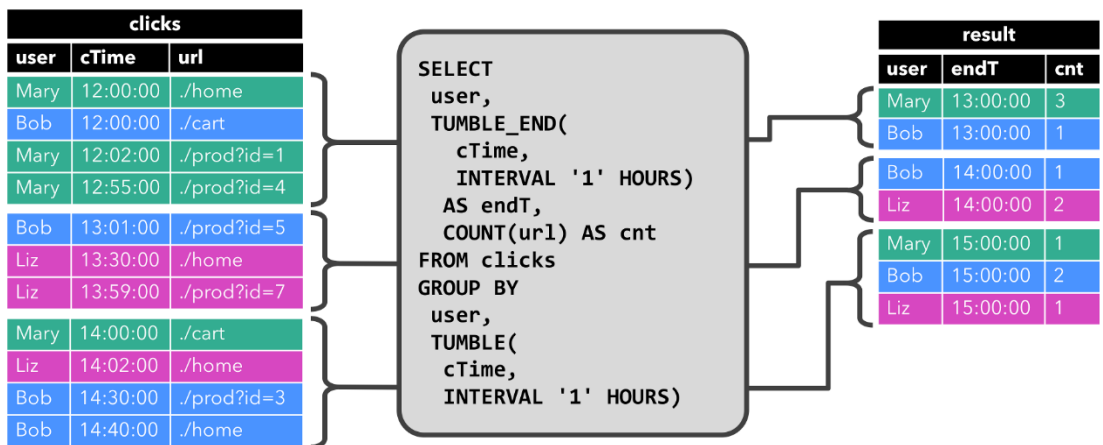
在下面的示例中，我们显示了对单击事件流中定义的表的两个示例查询。

第一个查询是一个简单的 GROUP-BY COUNT 聚合查询。它将 clicks 表中的字段 user 分组，并计算访问的 URL 数量。下图显示了随着 clicks 表中其他行的更新，随着时间的推移如何评估查询。



启动查询后，clicks 表（左侧）为空。当第一行插入到 clicks 表中时，查询开始计算结果表。插入第一行[Mary, ./home]后，结果表（右侧，顶部）由单行[Mary, 1]组成。将第二行[Bob, ./cart]插入到 clicks 表中时，查询将更新结果表并插入新行[Bob, 1]。第三行[Mary, ./prod?id=1]产生已计算结果行的更新，从而将[Mary, 1]更新为[Mary, 2]。最后，当第四行附加到 clicks 表时，查询将第三行[Liz, 1]插入结果表。

第二个查询与第一个查询类似，但是在对 clicks 表进行计数之前，除了将 user 属性归类之外，表还在小时滚动的窗口中进行分组（在基于 URL 的计数之前，基于时间的计算（例如，窗口基于特殊的时间属性），稍后将进行讨论）。同样，该图显示了在不同时间点的输入和输出，以可视化动态表的变化过程。



和以前一样，输入表 clicks 显示在左侧。该查询每小时持续计算结果并更新结果表。clicks 表包含四行，其时间戳（cTime）在 12:00:00 和 12:59:59 之间。该查询从输入计算出两个结果行（每个用户一个），并将它们附加到结果表中。对于 13:00:00 和 13:59:59 之间

的下一个窗口，该 `clicks` 表包含三行，这将导致另外两行追加到结果表中。结果表将更新，因为 `clicks` 随着时间的推移会添加更多行。

更新和 **Append** 查询

尽管两个示例查询看起来非常相似（都计算分组计数汇总），但是它们在一个重要方面有所不同：

- 第一个查询更新先前发出的结果，即结果表包含 `INSERT` 和 `UPDATE` 更改的变更日志流。
- 第二个查询仅附加到结果表，即结果表的 `changelog` 流仅包含 `INSERT` 更改。

查询是生成仅追加表还是更新表具有一些含义：

- 产生更新更改的查询通常必须维护更多状态（请参阅以下部分）。
- 仅追加表到流的转换与更新表的转换不同（请参阅[表到流转换](#)部分）。

查询限制

可以将许多但不是全部的语义有效查询评估为流中的持续查询。某些查询由于需要维护的状态大小或计算更新过于昂贵（注：计算成本）而无法计算。

- 状态大小：持续查询是在无限制的流上评估的，通常应该运行数周或数月。因此，持续查询处理的数据总量可能非常大。必须更新先前发出的结果的查询需要维护所有发出的行，以便能够更新它们。例如，第一个示例查询需要存储每个用户的 `URL` 计数，以便能够增加计数并在输入表接收到新行时发出新结果。如果仅跟踪注册用户，则要维护的计数数量可能不会太高。但是，如果未注册的用户获得分配的唯一用户名，则要维护的计数数量将随着时间的推移而增加，并最终可能导致查询失败。

```
SELECT user, COUNT(url)
FROM clicks
GROUP BY user;
```

- 计算更新：即使只添加或更新一条输入记录，某些查询也需要重新计算和更新很大一部分发射结果行。显然，这样的查询不太适合作为持续查询执行。下面的查询是一个示例，该查询根据最终点击的时间为每个用户计算排名。`clicks` 表格收到新行后，`lastAction` 用户的身份将更新，并且必须计算新排名。但是，由于两行不能具有相同的排名，因此所有排名较低的行也需要更新。

```
SELECT user, RANK() OVER (ORDER BY lastLogin)
FROM (
    SELECT user, MAX(cTime) AS lastAction FROM clicks GROUP BY user
);
```

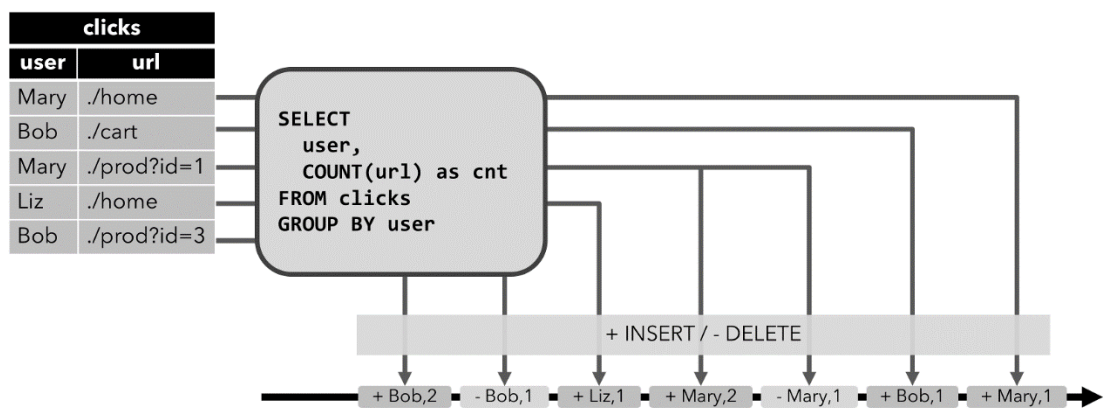
“[查询配置](#)”章节讨论了用于控制持续查询的执行的参数。某些参数可用于权衡维护状态的大小以提高结果的准确性。

表到流的转换

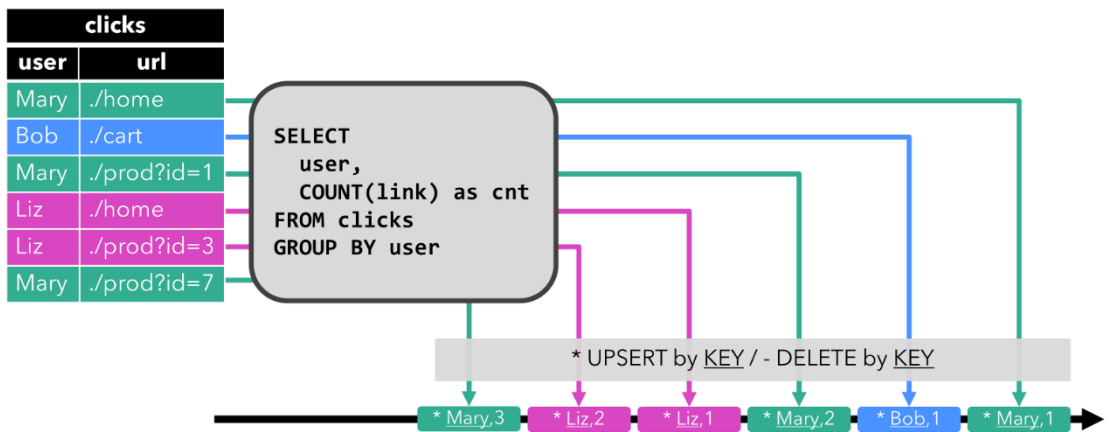
动态表可以通过 INSERT, UPDATE 以及 DELETE 不断修改, 就像一个普通的数据库表。它可能是具有单行的表, 该表会不断更新; 可能是一个仅插入的表, 没有 UPDATE 和 DELETE 修改, 或者介于两者之间。

将动态表转换为流或将其写入外部系统时, 需要对这些更改进行编码。Flink 的 Table API 和 SQL 支持三种方式来编码动态表的更改:

- **Append-only 流:** 可以通过发出插入的行将仅通过 INSERT 更改修改的动态表转换为流。
- **Retract 流:** 撤回流是具有两种消息类型的流, 添加消息和撤回消息。通过将 INSERT 更改编码为添加消息, 将 DELETE 更改编码为撤回消息, 将 UPDATE 更改编码为更新 (先前) 行的更新消息, 并将更新消息编码为更新 (新) 行, 将动态表转换为撤回流。下图可视化了动态表到撤回流的转换。



- **Upsert 流:** Upsert 流是具有两种消息类型的流, Upsert 消息和 Delete 消息。转换为 upsert 流的动态表需要一个 (可能是复合的) 唯一键。通过将 INSERT 和 UPDATE 更改编码为 upsert 消息并将 DELETE 更改编码为 delete 消息, 将具有唯一键的动态表转换为流。流消耗操作需要知道唯一键属性, 以便正确应用消息。流消费算子需要知道唯一键属性, 以便正确应用消息。撤回流的主要区别在于 UPDATE 更改使用单个消息进行编码, 因此效率更高。下图可视化了动态表到 upsert 流的转换。



在 [Common Concepts](#) 页面上讨论了将动态表转换为 `DataStream` 的 API。请注意，将动态表转换为 `DataStream` 时仅支持添加和撤消流。在 [TableSources](#) 和 [TableSinks](#) 页面上讨论了向外部系统发送动态表的 `TableSink` 接口。

【翻译】Flink Table Api & SQL —Streaming 概念 ——时间属性

本文翻译自官网：Time Attributes https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/streaming/time_attributes.html

Flink Table Api & SQL 翻译目录

Flink 能够根据不同的 *时间* 概念处理流数据。

- *Process time* 是指正在执行相应操作的机器的系统时间（也称为“挂钟时间”）。
- *Event time* 是指基于附在每行上的时间戳对流数据进行处理。时间戳可以在事件发生时进行编码。
- *Ingestion time* 是事件进入 Flink 的时间；在内部，它的处理类似于事件时间。

有关 Flink 中时间处理的更多信息，请参见有关[事件时间和水印](#)的介绍。

本页说明如何在 Flink 的 Table API 和 SQL 中为基于时间的操作定义时间属性。

- [时间属性简介](#)
- [处理时间](#)
 - [在数据流到表的转换期间](#)
 - [使用 TableSource](#)
- [事件时间](#)
 - [在数据流到表的转换期间](#)
 - [使用 TableSource](#)

时间属性简介

[Table API](#) 和 [SQL](#) 中的基于时间的操作（例如窗口）都需要有关时间概念及其起源的信息。因此，表可以提供 *逻辑时间属性*，以指示时间并访问表程序中的相应时间戳。

时间属性可以是每个表结构的一部分。它们是从 `DataStream` 创建表时定义的，或者是在使用 `TableSource` 时预定义的。一旦在开始定义了时间属性，就可以将其作为字段引用，并可以在基于时间的操作中使用。

只要时间属性没有被修改并且只是从查询的一部分转发到另一部分，它仍然是有效的时间属性。时间属性的行为类似于常规时间戳，可以进行访问以进行计算。常规时间戳记不能与 Flink 的时间和水印系统配合使用，因此不能再用于基于时间的操作。

表程序要求已为流环境指定了相应的时间特征：



```
val env = StreamExecutionEnvironment.getExecutionEnvironment

env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime) //
default

// alternatively:
// env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime)
// env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
```

处理时间

处理时间允许表程序根据本地计算机的时间产生结果。这是最简单的时间概念，但不提供确定性。它既不需要时间戳提取也不需要水印生成。

有两种定义处理时间属性的方法。

在数据流到表的转换期间

在结构定义期间，使用 `.proctime` 属性定义了处理时间属性。时间属性只能通过其他逻辑字段扩展物理结构。因此，只能在结构定义的末尾定义它。

```
val stream: DataStream[(String, String)] = ...

// declare an additional logical field as a processing time attribute
val table = tEnv.fromDataStream(stream, 'UserActionTimestamp,
'Username, 'Data, 'UserActionTime.proctime)

val windowedTable = table.window(Tumble over 10.minutes on
'UserActionTime as 'userActionWindow)
```

使用 TableSource

处理时间属性由实现 `DefinedProctimeAttribute` 接口的 `TableSource` 定义。逻辑时间属性附加到由 `TableSource` 的返回类型定义的物理结构。

```
class UserActionSource extends StreamTableSource[Row] with
DefinedProctimeAttribute {

  override def getReturnType = {
    val names = Array[String]("Username" , "Data")
    val types = Array[TypeInformation[_]](Types.STRING,
Types.STRING)
    Types.ROW(names, types)
  }
}
```

```

    override def getDataStream(execEnv: StreamExecutionEnvironment):
DataStream[Row] = {
    // create stream
    val stream = ...
    stream
}

    override def getProctimeAttribute = {
    // field with this name will be appended as a third field
    "UserActionTime"
    }
}

// register table source
tEnv.registerTableSource("UserActions", new UserActionSource)

val windowedTable = tEnv
    .scan("UserActions")
    .window(Tumble over 10.minutes on 'UserActionTime as
'userActionWindow)

```



事件时间

事件时间允许表程序根据每个记录中包含的时间来产生结果。即使在无序事件或迟发事件的情况下，这也可以提供一致的结果。从持久性存储中读取记录时，还可以确保表程序的可重播结果。

此外，事件时间允许批处理和流环境中的表程序使用统一语法。流环境中的时间属性可以是批处理环境中记录的常规字段。

为了处理乱序事件并区分流中的按时事件和延迟事件，Flink 需要从事件中提取时间戳并及时进行某种处理（就是水印）。

可以在 `DataStream` 到 `Table` 的转换期间或使用 `TableSource` 定义事件时间属性。


在 `DataStream` 到 `Table` 的转换期间

在结构定义期间，事件时间属性是使用 `.rowtime` 属性定义的。必须在转换的 `DataStream` 中分配时间戳和水印。

将 `DataStream` 转换为 `Table` 时，有两种定义时间属性的方法。根据指定的 `.rowtime` 字段名称是否存在于 `DataStream` 的结构中，`timestamp` 字段为

- 作为新字段附加到结构
- 替换现有字段。

无论哪种情况，事件时间时间戳字段都将保留 `DataStream` 事件时间 时间戳的值。



```
// Option 1:

// extract timestamp and assign watermarks based on knowledge of the
stream
val stream: DataStream[(String, String)] =
inputStream.assignTimestampsAndWatermarks(...)

// declare an additional logical field as an event time attribute
val table = tEnv.fromDataStream(stream, 'Username, 'Data,
'UserActionTime.rowtime)


// Option 2:

// extract timestamp from first field, and assign watermarks based on
knowledge of the stream
val stream: DataStream[(Long, String, String)] =
inputStream.assignTimestampsAndWatermarks(...)

// the first field has been used for timestamp extraction, and is no
longer necessary
// replace first field with a logical event time attribute
val table = tEnv.fromDataStream(stream, 'UserActionTime.rowtime,
'Username, 'Data)

// Usage:

val windowedTable = table.window(Tumble over 10.minutes on
'UserActionTime as 'userActionWindow)
```



使用 **TableSource**

事件时间属性由实现了 `DefinedRowtimeAttributes` 接口的 `TableSource` 定义。
`getRowtimeAttributeDescriptors`（）方法返回用于描述时间属性最终名称的
`RowtimeAttributeDescriptor` 列表，用于导出属性值的时间戳提取器以及与该属性关联的水
印策略。

请确保由 `getDataStream`（）方法返回的 `DataStream` 与定义的时间属性对齐。仅当定义了
`StreamRecordTimestamp` 时间戳提取器时，才考虑 `DataStream` 的时间戳（由
`TimestampAssigner` 分配的时间戳）。仅当定义了 `PreserveWatermarks` 水印策略时，才
会保留 `DataStream` 的水印。 否则，仅 `TableSource` 的 `rowtime` 属性的值相关。



```

// define a table source with a rowtime attribute
class UserActionSource extends StreamTableSource[Row] with
DefinedRowtimeAttributes {

    override def getReturnType = {
        val names = Array[String]("Username" , "Data",
"UserActionTime")
        val types = Array[TypeInformation[_]](Types.STRING,
Types.STRING, Types.LONG)
        Types.ROW(names, types)
    }

    override def getDataStream(execEnv: StreamExecutionEnvironment):
DataStream[Row] = {
        // create stream
        // ...
        // assign watermarks based on the "UserActionTime" attribute
        val stream = inputStream.assignTimestampsAndWatermarks(...)
        stream
    }

    override def getRowtimeAttributeDescriptors:
util.List[RowtimeAttributeDescriptor] = {
        // Mark the "UserActionTime" attribute as event-time
attribute.
        // We create one attribute descriptor of "UserActionTime".
        val rowtimeAttrDescr = new RowtimeAttributeDescriptor(
            "UserActionTime",
            new ExistingField("UserActionTime"),
            new AscendingTimestamps)
        val listRowtimeAttrDescr =
Collections.singletonList(rowtimeAttrDescr)
        listRowtimeAttrDescr
    }
}

// register the table source
tEnv.registerTableSource("UserActions", new UserActionSource)

val windowedTable = tEnv
    .scan("UserActions")
    .window(Tumble over 10.minutes on 'UserActionTime as
'userActionWindow)

```



【翻译】Flink Table Api & SQL —Streaming 概念 ——在持续查询中 Join

本文翻译自官网： Joins in Continuous

Queries <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/streaming/joins.html>

Flink Table Api & SQL 翻译目录

Join 是批量数据处理中连接两个关系行的常见且易于理解的操作。但是，动态表上的 join 语义不那么明显，甚至令人困惑。

因此，有一些方法可以使 Table API 或 SQL 实际执行 join 。

有关语法的更多信息，请检查 Table API 和 SQL 中的 join 部分。

- 常规 join (Regular Joins)
- 时间窗口 join
- 与时态表函数 join
 - 用法
 - 处理时间时态 (表函数) join
 - 事件时间时态 (表函数) join
- 时态表 join
 - 用法

Regular Joins

常规 join 是最通用的 join 类型，在该 join 中，任何新记录的更改对 join 输入两侧都是可见的，并且会影响整个 join 结果。例如，如果左侧有一个新记录，则它将与右侧的所有以前和将来的记录合并在一起。

```
SELECT * FROM Orders
INNER JOIN Product
ON Orders.productId = Product.id
```

这些语义允许进行任何类型的更新（插入，更新，删除）的输入表。

但是，此操作有一个重要的含义：它要求将 join 输入的两端始终保持在 Flink 的状态。因此，如果一个或两个输入表持续增长，资源使用也将无限期增长。

Time-windowed Joins

时间窗口 join 由 join 谓词定义，该 join 谓词检查输入记录的时间属性是否在某些时间限制（即时间窗口）内。

```

SELECT *
FROM
    Orders o,
    Shipments s
WHERE o.id = s.orderId AND
      o.ordertime BETWEEN s.shiptime - INTERVAL '4' HOUR AND
      s.shiptime

```


与常规 `join` 操作相比，这种 `join` 仅支持具有时间属性的 `append-only` 表。由于时间属性是准单调递增的，因此 Flink 可以从其状态中删除旧值，而不会影响结果的正确性。

Join with a Temporal Table Function

具有时态表函数的 `join` 将 `append-only` 表（左侧输入/探针侧，注：输入流）与临时表（右侧输入/构建侧，注：维表）`join`，即随时间变化并跟踪其变化的表（维表）。请查看相应的页面以获取有关时态表的更多信息。

以下示例显示了 `append-only` 表 `Orders`，该表与不断变化的货币汇率表 `RatesHistory` 结合在一起。

`Orders` 是一个 `append-only` 表，代表给定 `amount` 和给定货币（`currency`）的付款。例如，在 10:15 有一笔金额为 2 欧元的订单。




```

SELECT * FROM Orders;

```

rowtime	amount	currency
10:15	2	Euro
10:30	1	US Dollar
10:32	50	Yen
10:52	3	Euro
11:04	5	US Dollar

`RatesHistory` 表示日元汇率（汇率为 1）不断变化的 `append-only` 表。例如，从 09:00 到 10:45 欧元对日元的汇率为 114。从 10:45 到 11:15 为 116。



```

SELECT * FROM RatesHistory;

```

rowtime	currency	rate
09:00	US Dollar	102
09:00	Euro	114
09:00	Yen	1
10:45	Euro	116
11:15	Euro	119

11:49 Pounds 108



假设我们要计算所有订单转换为通用货币（日元）的金额。

例如，我们要使用给定 **rowtime**（114）的适当转换率转换以下订单。

```
rowtime amount currency
=====
10:15      2 Euro
```

如果不使用时态表的概念，则需要编写如下查询：



```
SELECT
    SUM(o.amount * r.rate) AS amount
FROM Orders AS o,
    RatesHistory AS r
WHERE r.currency = o.currency
AND r.rowtime = (
    SELECT MAX(rowtime)
    FROM RatesHistory AS r2
    WHERE r2.currency = o.currency
    AND r2.rowtime <= o.rowtime);
```



借助时态表函数 使 **RatesHistory** 的汇率变化，我们可以在 SQL 中将查询表示为：

```
SELECT
    o.amount * r.rate AS amount
FROM
    Orders AS o,
    LATERAL TABLE (Rates(o.rowtime)) AS r
WHERE r.currency = o.currency
```

探针端记录的相关时间属性时，来自探针端的每个记录将与构建端表的版本关联。为了支持生成侧表上先前值的更新（覆盖），该表必须定义一个主键。

在我们的示例中，**from Orders** 中的每个记录都将与 **Rates** 的 **o.rowtime** 时间版本 结合在一起。该 **currency** 字段已 提前定义为 **Rates** 的主键，并且在我们的示例中用于连接两个表。如果查询使用的是处理时间概念，则执行操作时，新添加的订单将始终与 **Rates** 的 最新版本结合在一起。

与常规 join 相反，这意味着如果在构建端有新记录，则不会影响 join 的先前结果。这又使 **Flink** 可以限制必须保持状态的元素数量。


与时间窗口连接相比，时态表 **join** 未定义时间范围，（所有）关联的数据将被 **join**。探测端的记录总是在 **time** 属性指定的时间与构建端的版本连接在一起。因此，构建端（时态表）的记录可能是任意旧的。随着时间的流逝，该记录的先前版本和不再需要的版本（对于给定的主键）将从状态中删除。

这种行为使临时表成为一个很好的候选者，可以用关系术语来表示流的 `join`。

用法

定义时态表函数 后，我们就可以开始使用它。可以使用与普通表函数相同的方式来使用时态表函数。


以下代码段解决了我们从 `Orders` 表中转换货币的初衷问题：



```
## SQL
SELECT
  SUM(o_amount * r_rate) AS amount
FROM
  Orders,
  LATERAL TABLE (Rates(o_proctime))
WHERE
  r_currency = o_currency

## Java
Table result = orders
  .joinLateral("rates(o_proctime)", "o_currency = r_currency")
  .select("(o_amount * r_rate).sum as amount");

## Scala
val result = orders
  .joinLateral(rates('o_proctime), 'r_currency === 'o_currency)
  .select(('o_amount * 'r_rate).sum as 'amount)
```



注意：对于时态表 `join`，尚未实现在[查询配置中](#)定义的状态保留。这意味着，计算查询结果所需的状态可能会无限增长，具体取决于历史记录表的不同主键数量。

Processing-time Temporal Joins

基于处理时间的时间属性，是不可能通过 *过去的* 时间的属性作为参数的时间表函数（注：处理时间只会增长）。根据定义，它始终是当前时间戳。因此，处理时间时态表函数的调用将始终返回基础表的最新已知版本，基础历史表中的任何更新也将立即覆盖当前值。

仅将构建侧记录的最新版本（相对于定义的主键）保持在该状态。构建端的更新将不会影响先前发出的 `join` 结果。

可以将处理时间的 时态表 视为一种简单的 `HashMap<K, V>`，它可以存储构建侧的所有记录。当来自构建端的新记录与先前的记录具有相同的键时，旧值将被简单地覆盖。总是根据 `HashMap` 的最新/当前状态评估来自探测器端的每个记录（注：与输入数据 `join`）。

Event-time Temporal Joins

使用事件时间 的时间属性（即行时间属性），可以将*过去*的时间属性传递给时态表函数。这允许在公共时间点将两个表（时态表的两个时间状态）连接在一起。

与处理时间 时态表 **join** 相比，时态表不仅将构建侧记录的最新版本（相对于定义的主键）保持在状态中，而且还存储自上次水印以来的所有版本（按时间标识）。

例如，根据时态表的概念，将附加到探针侧表的事件时间 时间戳为 **12:30:00** 的输入行与构建侧表 在时间 **12:30:00** 的版本 进行连接。因此，传入行仅与时间戳小于或等于 **12:30:00** 的行连接，并根据主键应用更新直到该时间点。

根据事件时间的定义，水印允许 **join** 操作及时向前移动，并丢弃不再需要的构建表版本，因为不会输入具有更低或相等时间戳的行。

与时态表 **Join**

与时态表的 **join** 将任意表（左侧输入/探针侧）与时态表（右侧输入/构建侧）**join**，即随时间变化的外部数据表。请查看相应的页面以获取有关时态表的更多信息。

注意：用户不能将任意表用作时态表，需要使用 **LookupableTableSource** 支持的表。

LookupableTableSource 只能作为时间表用于时间联接。有关[如何定义 LookupableTableSource](#) 的更多详细信息，请参见页面。

下面的示例显示了一个 **Orders** 流，该流与不断变化的货币汇率表 **LatestRates** 结合在一起。

LatestRates 是物化最新汇率的维度表。 在时间 **10:15**、**10:30**、**10:52**，**LatestRates** 的内容如下：

```
10:15> SELECT * FROM LatestRates;

currency  rate
=====
US Dollar 102
Euro      114
Yen       1

10:30> SELECT * FROM LatestRates;

currency  rate
=====
US Dollar 102
Euro      114
Yen       1

10:52> SELECT * FROM LatestRates;

currency  rate
=====
US Dollar 102
```

```
Euro      116      <==== changed from 114 to 116
Yen        1
```

时间 10:15 和 10:30 的 LatestRates 的内容相等。欧元汇率在 10:52 从 114 更改为 116。

Orders 是一个 append-only 表，代表给定金额和给定货币的付款。例如，在 10:15 时有一笔 2 欧元的订单。

```
SELECT * FROM Orders;

amount currency
=====
      2 Euro      <== arrived at time 10:15
      1 US Dollar <== arrived at time 10:30
      2 Euro      <== arrived at time 10:52
```

假设我们要计算所有 Orders 折算成通用货币（日元）的金额。

例如，我们想使用 LatestRates 中的最新汇率转换以下订单。结果将是：

amount	currency	rate	amout*rate	
2	Euro	114	228	<== arrived at time 10:15
1	US Dollar	102	102	<== arrived at time 10:30
2	Euro	116	232	<== arrived at time 10:52

借助时态表联接，我们可以在 SQL 中将查询表示为：

```
SELECT
  o.amout, o.currency, r.rate, o.amount * r.rate
FROM
  Orders AS o
  JOIN LatestRates FOR SYSTEM_TIME AS OF o.proctime AS r
  ON r.currency = o.currency
```

探针端的每个记录都将与构建端表的当前版本关联。在我们的示例中，查询使用的是处理时间概念，因此在执行操作时，新附加的订单将始终与最新版本的 LatestRates 结合在一起。注意，结果对于处理时间不是确定的。

与常规 join 相反，尽管在构建方面进行了更改（注：数据修改了），但时态表 Join 的先前结果将不会受到影响。而且，时态表 join 运算符非常轻，并且不保留任何状态。

与时间窗口 join 相比，时态表 join 没有定义将要在其中 join 记录的时间窗口。在处理时，探测端的记录总是与构建端的最新版本结合在一起。因此，构建方面的记录可能是任意旧的。

时态表函数 join 和 时态表 join 都来自相同的初衷，但是具有不同的 SQL 语法和运行时实现：

- 时态表函数 join 的 SQL 语法是 join UDTF，而时态表联接使用 SQL: 2011 中引入的标准时态表语法。
- 时态表函数 join 的实现实际上 join 了两个流并使它们保持状态，而时态表 join 仅接收唯一的输入流并根据记录中的键查找外部数据库。
- 时态表函数 join 通常用于 join 变更日志流，而临时表 join 通常用于 join 外部表（即维表）。

这种行为使时态表成为一个很好的候选者，可以用关系术语来表示流的 join。

将来，时态表联接将支持时态表功能 join 的功能，即支持时态 join 变更日志流。

用法

临时表 join 的语法如下：

```
SELECT [column_list]
FROM table1 [AS <alias1>]
[LEFT] JOIN table2 FOR SYSTEM_TIME AS OF table1.proctime [AS
<alias2>]
ON table1.column-name1 = table2.column-name1
```

当前，仅支持 INNER JOIN 和 LEFT JOIN。临时表之后应遵循 FOR SYSTEM_TIME AS OF table1.proctime。proctime 是 table1 的处理时间属性。这意味着在连接左表中的每个记录时，它会在处理时为时态表做快照。

例如，在定义时态表之后，我们可以如下使用它。

```
SELECT
    SUM(o_amount * r_rate) AS amount
FROM
    Orders
    JOIN LatestRates FOR SYSTEM_TIME AS OF o_proctime
    ON r_currency = o_currency
```

注意：仅在 Blink planner 中支持。

注意：仅在 SQL 中支持，而在 Table API 中尚不支持。

注意：Flink 当前不支持事件时间时态表 join。

【翻译】Flink Table Api & SQL —Streaming 概念 —— 时态表

本文翻译自官网：Temporal Tables https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/streaming/temporal_tables.html

Flink Table Api & SQL 翻译目录

时态表（注：Temporal Table，我翻译为时态表，可以访问表在不同时间的内容）表示一直在修改的表上的（参数化）视图的概念，该视图返回表在特定时间点的内容。

更改表可以是跟踪表的修改历史（例如，数据库更改日志），也可以是维表的具体修改（例如，数据库表）。

对于表的历史修改，Flink 可以跟踪修改，并允许在查询中访问表的特定时间点的内容。在 Flink 中，这种表由 *Temporal Table Function* 表示。


对于变化的维表，Flink 允许在查询中的处理时访问表的内容。在 Flink 中，这种表由 *Temporal Table* 表示。

- 设计初衷
 - 与表的修改历史相关
 - 与维表(内容)变化相关
- 时态表函数
 - 定义时态表函数
- 时态表
 - 定义时态表

设计初衷


与表的修改历史相关

假设我们有下表 RatesHistory。




```
SELECT * FROM RatesHistory;
```

rowtime	currency	rate
09:00	US Dollar	102
09:00	Euro	114
09:00	Yen	1
10:45	Euro	116
11:15	Euro	119
11:49	Pounds	108



RatesHistory 表示一个不断增长的关于日元的货币汇率的附加表（汇率为 1）。例如，汇率期间从 09:00 到 10:45 的欧元到日元的汇率为 114。从 10:45 到 11:15 是 116。

假设我们要在 10:58 的时间输出所有当前汇率，则需要以下 SQL 查询来计算结果表：



```
SELECT *  
FROM RatesHistory AS r
```

```
WHERE r.rowtime = (
    SELECT MAX(rowtime)
    FROM RatesHistory AS r2
    WHERE r2.currency = r.currency
    AND r2.rowtime <= TIME '10:58');
```



子查询确定对应货币的最大时间小于或等于所需时间。外部查询列出具有最大时间戳的汇率。

下表显示了这种计算的结果。 在我们的示例中，考虑了 10:45 时欧元的更新，但是 10:58 时表的版本中未考虑 11:15 时欧元的更新和新的英镑输入。

rowtime	currency	rate
09:00	US Dollar	102
09:00	Yen	1
10:45	Euro	116

*时态表*的概念旨在简化此类查询，加快其执行速度，并减少 Flink 的状态使用率。时态表是 **append-only** 表上的参数化视图，该视图将 **append-only** 表的行解释为表的变更日志，并在特定时间点提供该表的版本。将 **append-only** 表解释为变更日志需要指定主键属性和时间戳属性。主键确定覆盖哪些行，时间戳确定行有效的的时间。

在上面的示例中，currency 是 RatesHistory 表的主键，并且 rowtime 是 timestamp 属性。

在 Flink 中，这由[*时态表函数表示*](#)。

与维表变化相关

另一方面，某些用例需要连接变化的维表，该表是外部数据库表。

假设 LatestRates 是一个以最新汇率实现的表（例如，存储在其中）。LatestRates 是物化的 RatesHistory 历史。那么时间 10:58 的 LatestRates 表的内容将是：

```
10:58> SELECT * FROM LatestRates;
currency    rate
=====
US Dollar   102
Yen         1
Euro        116
```

12:00 时 LatestRates 表的内容将是：



```
12:00> SELECT * FROM LatestRates;
currency    rate
=====
US Dollar   102
Yen         1
Euro        119
```



在 Flink 中，这由 *[Temporal Table](#)* 表示。

时态表函数

为了访问时态表中的数据，必须传递一个[时间属性](#)，该[属性](#)确定将要返回的表的版本。Flink 使用[表函数](#)的 SQL 语法提供一种表达它的方法。

定义后，时态表函数将使用单个时间参数 `timeAttribute` 并返回一组行。该集合包含相对于给定时间属性的所有现有主键的行的最新版本。

假设我们 `Rates(timeAttribute)` 基于 `RatesHistory` 表定义了一个时态表函数，我们可以通过以下方式查询该函数：



```
SELECT * FROM Rates('10:15');
```

rowtime	currency	rate
09:00	US Dollar	102
09:00	Euro	114
09:00	Yen	1

```
SELECT * FROM Rates('11:00');
```

rowtime	currency	rate
09:00	US Dollar	102
10:45	Euro	116
09:00	Yen	1



对 `Rates(timeAttribute)` 的每个查询都将返回给定 `timeAttribute` 的 `Rates` 状态。

注意：当前 Flink 不支持使用常量时间属性参数直接查询时态表函数。目前，时态表函数只能在 `join` 中使用。上面的示例用于提供有关函数 `Rates(timeAttribute)` 返回内容的直观信息。

另请参阅有关[用于持续查询的 join](#)的页面，以获取有关如何与时态表 `join` 的更多信息。

定义时态表函数

以下代码段说明了如何从 `append-only` 表中创建时态表函数。



```
// Get the stream and table environments.  
val env = StreamExecutionEnvironment.getExecutionEnvironment  
val tEnv = StreamTableEnvironment.create(env)
```

```

// Provide a static data set of the rates history table.
val ratesHistoryData = new mutable.MutableList[(String, Long)]
ratesHistoryData.+=(("US Dollar", 102L))
ratesHistoryData.+=(("Euro", 114L))
ratesHistoryData.+=(("Yen", 1L))
ratesHistoryData.+=(("Euro", 116L))
ratesHistoryData.+=(("Euro", 119L))

// Create and register an example table using above data set.
// In the real setup, you should replace this with your own table.
val ratesHistory = env
    .fromCollection(ratesHistoryData)
    .toTable(tEnv, 'r_currency, 'r_rate, 'r_proctime.proctime)

tEnv.registerTable("RatesHistory", ratesHistory)

// Create and register TemporalTableFunction.
// Define "r_proctime" as the time attribute and "r_currency" as the
primary key.
val rates = ratesHistory.createTemporalTableFunction('r_proctime,
'r_currency) // <==== (1)
tEnv.registerFunction("Rates", rates)
// <==== (2)

```

Line (1) 创建了一个 时态表函数 `rates`，使我們可以在 Table API 中使用 `rates` 函数。

Line (2) 在表环境中以 `Rates` 名称注册此函数，这使我們可以在 SQL 中使用 `Rates` 函数。

时态表

注意：仅 `Blink planner` 支持此功能。

为了访问时态表中的数据，当前必须使用 `LookupableTableSource` 定义一个 `TableSource`。Flink 使用 `FOR SYSTEM_TIME AS OF` 的 SQL 语法查询时态表，这在 SQL: 2011 中提出。

假设我们定义了一个时态表 `LatestRates`，我们可以通过以下方式查询此类表：

```

SELECT * FROM LatestRates FOR SYSTEM_TIME AS OF TIME '10:15';

currency    rate
=====
US Dollar   102
Euro        114

```



```

Yen            1

SELECT * FROM LatestRates FOR SYSTEM_TIME AS OF TIME '11:00';

currency    rate
=====
US Dollar   102
Euro        116
Yen         1

```

注意：当前，Flink 不支持以固定时间直接查询时态表。目前，时态表只能在 `join` 中使用。上面的示例用于提供有关时态表 `LatestRates` 返回内容的直觉。

另请参阅有关[用于持续查询的 join](#)的页面，以获取有关如何与时态表 `join` 的更多信息。

定义时态表

```

// Get the stream and table environments.
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tEnv = TableEnvironment.getTableEnvironment(env)

// Create an HBaseTableSource as a temporal table which implements
// LookableTableSource
// In the real setup, you should replace this with your own table.
val rates = new HBaseTableSource(conf, "Rates")
rates.setRowKey("currency", String.class) // currency as the
primary key
rates.addColumn("rate", Double.class)

// register the temporal table into environment, then we can query it
in sql
tEnv.registerTableSource("Rates", rates)

```

另请参阅有关[如何定义 LookableTableSource](#)的页面。

【翻译】Flink Table Api & SQL —Streaming 概念 —— 表中的模式匹配 Beta 版

本文翻译自官网：Detecting Patterns in Tables Beta https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/streaming/match_recognize.html

[Flink Table Api & SQL 翻译目录](#)

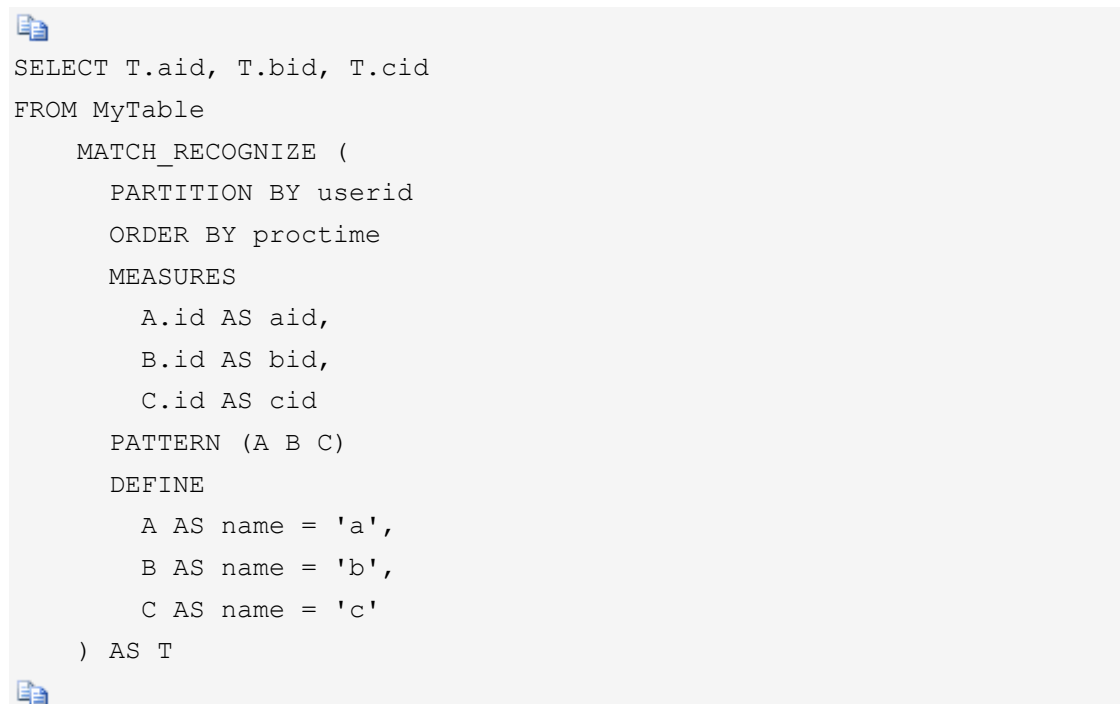
搜索一组事件模式是一种常见的用例，尤其是在数据流的情况下。Flink 带有一个[复杂的事件处理（CEP）库](#)，该库允许在事件流中进行模式检测。此外，Flink 的 SQL API 提供了一种关系查询方式，该查询具有一系列内置函数和基于规则的优化来表达查询，这些查询可以直接使用。

2016 年 12 月，国际标准化组织（ISO）发布了 SQL 版本的新版本，其中包括 SQL 中的[行模式识别（ISO / IEC TR 19075-5: 2016）](#)。它允许 Flink 使用该 MATCH_RECOGNIZE 子句合并 CEP 和 SQL API，以在 SQL 中进行复杂的事件处理。

MATCH_RECOGNIZE 子句启用以下任务：

- 使用 PARTITION BY 和 ORDER BY 子句一起逻辑分区和排序数据。
- 使用 PATTERN 子句定义要搜索的行模式。这些模式使用与正则表达式相似的语法。
- 行模式变量的逻辑组件在 DEFINE 子句中指定。
- 在 MEASURES 子句中定义度量，这些度量是在 SQL 查询的其他部分中可用的表达式。

以下示例说明了基本模式识别的语法：



```
SELECT T.aid, T.bid, T.cid
FROM MyTable
  MATCH_RECOGNIZE (
    PARTITION BY userid
    ORDER BY proctime
    MEASURES
      A.id AS aid,
      B.id AS bid,
      C.id AS cid
    PATTERN (A B C)
    DEFINE
      A AS name = 'a',
      B AS name = 'b',
      C AS name = 'c'
  ) AS T
```

该页面将更详细地说明每个关键字，并说明更复杂的示例。

注意：Flink 对 MATCH_RECOGNIZE 子句的实现是完整标准的子集。仅支持以下各节中记录的那些功能。由于开发仍处于早期阶段，因此请同时查看 [已知的局限性](#)。

- [简介与范例](#)
 - [安装指南](#)
 - [SQL 语义](#)
 - [例子](#)
- [分区](#)
- [事件顺序](#)

- [定义和度量](#)
 - [聚合](#)
- [定义模式](#)
 - [Greedy & Reluctant 量词](#)
 - [时间限制](#)
- [输出模式](#)
- [模式导航](#)
 - [模式变量引用](#)
 - [逻辑偏移](#)
- [匹配后策略](#)
- [时间属性](#)
- [控制内存消耗](#)
- [已知局限性](#)

简介与范例

安装指南

模式识别功能在内部使用 Apache Flink 的 CEP 库。为了能够使用该 MATCH_RECOGNIZE 子句，需要将库作为依赖项添加到您的 Maven 项目中。

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-cep_2.11</artifactId>
  <version>1.9.0</version>
</dependency>
```

或者，您也可以将依赖项添加到群集类路径中（有关更多信息，请参见 [依赖项部分](#)）。

如果要在 [SQL Client](#) 中使用 MATCH_RECOGNIZE 子句，则无需执行任何操作，因为默认情况下包括所有依赖项。

SQL 语义

每个 MATCH_RECOGNIZE 查询都包含以下子句：

- [PARTITION BY](#) - 定义表的逻辑分区；类似于 GROUP BY 操作。
- [ORDER BY](#) - 指定应如何对进入的行进行排序；这是必不可少的，因为模式取决于顺序。
- [MEASURES](#) - 定义该条款的输出；类似于 SELECT 条款。
- [ONE ROW PER MATCH](#) - 输出模式，定义每个匹配项应产生多少行。
- [AFTER MATCH SKIP](#) - 指定下一次匹配应该在哪里开始；这也是控制单个事件可以属于多少个不同匹配项的方法。
- [PATTERN](#) - 允许构造使用类似正则表达式的语法进行搜索的模式。
- [定义](#) - 本部分定义了模式变量必须满足的条件。

注意：当前，该 MATCH_RECOGNIZE 子句只能应用于追加表。此外，它也总是产生一个追加表。


例子

对于我们的示例，我们假设一个表 Ticker 已被注册。该表包含特定时间点的股票价格。


该表具有以下结构：

```
Ticker
  |-- symbol: String          # symbol of the
stock
  |-- price: Long            # price of the
stock
  |-- tax: Long              # tax liability of
the stock
  |-- rowtime: TimeIndicatorTypeInfo(rowtime) # point in time
when the change to those values happened
```

为了简化，我们仅考虑单个 ACME 股票的传入数据。 股票代码看起来类似于下表，其中的行被连续附加。



symbol	rowtime	price	tax
=====	=====	=====	=====
'ACME'	'01-Apr-11 10:00:00'	12	1
'ACME'	'01-Apr-11 10:00:01'	17	2
'ACME'	'01-Apr-11 10:00:02'	19	1
'ACME'	'01-Apr-11 10:00:03'	21	3
'ACME'	'01-Apr-11 10:00:04'	25	2
'ACME'	'01-Apr-11 10:00:05'	18	1
'ACME'	'01-Apr-11 10:00:06'	15	1
'ACME'	'01-Apr-11 10:00:07'	14	2
'ACME'	'01-Apr-11 10:00:08'	24	2
'ACME'	'01-Apr-11 10:00:09'	25	2
'ACME'	'01-Apr-11 10:00:10'	19	1



现在的任务是找到单个股票价格不断下降的时期。为此，可以编写如下查询：



```
SELECT *
FROM Ticker
  MATCH_RECOGNIZE (
    PARTITION BY symbol
    ORDER BY rowtime
    MEASURES
```

```

        START_ROW.rowtime AS start_tstamp,
        LAST(PRICE_DOWN.rowtime) AS bottom_tstamp,
        LAST(PRICE_UP.rowtime) AS end_tstamp
ONE ROW PER MATCH
AFTER MATCH SKIP TO LAST PRICE_UP
PATTERN (START_ROW PRICE_DOWN+ PRICE_UP)
DEFINE
    PRICE_DOWN AS
        (LAST(PRICE_DOWN.price, 1) IS NULL AND
PRICE_DOWN.price < START_ROW.price) OR
        PRICE_DOWN.price < LAST(PRICE_DOWN.price, 1),
    PRICE_UP AS
        PRICE_UP.price > LAST(PRICE_DOWN.price, 1)
) MR;

```

该查询按 **symbol** 列对 **Ticker** 表进行分区，并按 **rowtime** 时间属性对其进行排序。

PATTERN 子句指定我们对以下模式感兴趣：该模式的起始事件为 **START_ROW**，后跟一个或多个 **PRICE_DOWN** 事件，并以 **PRICE_UP** 事件结束。 如果可以找到这样的模式，则将在最后一个 **PRICE_UP** 事件中寻找下一个模式匹配，如 **AFTER MATCH SKIP TO LAST** 子句所示。

DEFINE 子句指定 **PRICE_DOWN** 和 **PRICE_UP** 事件需要满足的条件。 尽管不存在 **START_ROW** 模式变量，但它具有一个隐式条件，该条件始终被评估为 **TRUE**。

模式变量 **PRICE_DOWN** 定义为价格小于满足 **PRICE_DOWN** 条件的最后一行价格的行。 对于初始情况或当没有满足 **PRICE_DOWN** 条件的最后一行时，该行的价格应小于该模式中前一行的价格（由 **START_ROW** 引用）。

模式变量 **PRICE_UP** 定义为价格大于满足 **PRICE_DOWN** 条件的最后一行价格的行。

该查询为股票价格连续下降的每个时期产生一个汇总行。

输出行的确切表示在查询的 **MEASURES** 部分中定义。 输出行数由“每行一次匹配”输出模式定义。

symbol	start_tstamp	bottom_tstamp	end_tstamp
=====	=====	=====	=====
ACME	01-APR-11 10:00:04	01-APR-11 10:00:07	01-APR-11 10:00:08

结果行描述了一个价格下跌时期，该时期始于 **01-APR-11 10:00:04**，并在 **01-APR-11 10:00:07** 达到了最低价格，并在 **01-APR-11 10:00:08** 再次上涨。

分区

可以在分区数据中查找模式，例如单个股票或特定用户的趋势。这可以使用 **PARTITION BY** 子句来表达。该子句类似于 **GROUP BY** 用于聚合。

注意：强烈建议对传入的数据进行分区，因为否则该 `MATCH_RECOGNIZE` 子句将转换为非并行运算符以确保全局排序。

事件顺序

Apache Flink 允许根据时间搜索模式：[处理时间或事件时间](#)。

如果使用事件时间，则在将事件传递到内部模式状态机之前对其进行排序。因此，无论将行附加到表的顺序如何，生成的输出都是正确的。而且按照每行中包含的时间指定的顺序评估模式。

`MATCH_RECOGNIZE` 子句假定一个以升序排列的[时间属性](#)作为 `ORDER BY` 子句的第一个参数。

对于示例 `Ticker` 表，`ORDER BY rowtime ASC, price DESC` 的定义有效，但是 `ORDER BY price, rowtime` 或 `ORDER BY rowtime DESC, price ASC` 无效。

定义和度量

在简单的 SQL 查询中，`DEFINE` 和 `MEASURES` 关键字的含义与 `WHERE` 和 `SELECT` 子句的含义相似。

`MEASURES` 子句定义匹配模式的输出中将包含的内容。它可以投影列并定义用于评估的表达式。产生的行数取决于输出模式设置。

`DEFINE` 子句指定行必须满足的条件才能被分类为相应的模式变量。如果未为模式变量定义条件，则将使用默认条件，每一行的条件都为 `true`。


有关可以在这些子句中使用的表达式的更详细说明，请查看[事件流导航](#)部分。

聚合

可以在 `DEFINE` 和 `MEASURES` 子句中使用聚合。内置和自定义用户定义函数均受支持。

聚合函数应用于映射到匹配项的行的每个子集。为了了解如何评估这些子集，请查看[事件流导航](#)部分。

以下示例的任务是找到报价器的平均价格未低于特定阈值的最长时间段。它显示了可表达的 `MATCH_RECOGNIZE` 如何通过聚合来实现。可以通过以下查询执行此任务：



```
SELECT *
FROM Ticker
  MATCH_RECOGNIZE (
    PARTITION BY symbol
    ORDER BY rowtime
    MEASURES
      FIRST(A.rowtime) AS start_tstamp,
      LAST(A.rowtime) AS end_tstamp,
      AVG(A.price) AS avgPrice
```

```
ONE ROW PER MATCH
AFTER MATCH SKIP PAST LAST ROW
PATTERN (A+ B)
DEFINE
    A AS AVG(A.price) < 15
) MR;
```

给定此查询和以下输入值：

symbol	rowtime	price	tax
=====	=====	=====	=====
'ACME'	'01-Apr-11 10:00:00'	12	1
'ACME'	'01-Apr-11 10:00:01'	17	2
'ACME'	'01-Apr-11 10:00:02'	13	1
'ACME'	'01-Apr-11 10:00:03'	16	3
'ACME'	'01-Apr-11 10:00:04'	25	2
'ACME'	'01-Apr-11 10:00:05'	2	1
'ACME'	'01-Apr-11 10:00:06'	4	1
'ACME'	'01-Apr-11 10:00:07'	10	2
'ACME'	'01-Apr-11 10:00:08'	15	2
'ACME'	'01-Apr-11 10:00:09'	25	2
'ACME'	'01-Apr-11 10:00:10'	25	1
'ACME'	'01-Apr-11 10:00:11'	30	1

只要事件的平均价格不超过 **15**，该查询就会将事件作为模式变量 **A** 的一部分进行累积。例如，这种限制发生在 **01-Apr-11 10:00:04**。接下来的时间段在 **01-Apr-11 10:00:11** 再次超过平均价格 **15**。因此，所述查询的结果将是：

symbol	start_tstamp	end_tstamp	avgPrice
=====	=====	=====	=====
ACME	01-APR-11 10:00:00	01-APR-11 10:00:03	14.5
ACME	01-APR-11 10:00:05	01-APR-11 10:00:10	13.5

注意：聚合可以应用于表达式，但前提是它们引用单个模式变量。因此 `SUM(A.price * A.tax)` 是有效的，但 `AVG(A.price * B.tax)` 不是。

注意：不支持 **DISTINCT** 聚合。

定义模式

使用 **MATCH_RECOGNIZE** 子句，用户可以使用功能强大且表现力强的语法来搜索事件流中的模式，该语法有点类似于广泛的正则表达式语法。

每个模式都由称为 *模式变量* 的基本构建块构建而成，可以将运算符（量词和其他修饰符）应用于该基本变量。整个模式必须放在方括号中。

模式示例如下所示：

```
PATTERN (A B+ C* D)
```

可以使用以下运算符：

- **串联** - 类似的模式 (A B) 表示 A 和 B 之间严格连续。因此，不能有未映射到其间 A 或 B 之间的行。
- **量词** - 修改可以映射到模式变量的行数。
 - * — 0 行或更多行
 - + — 1 行或更多行
 - ? — 0 或 1 行
 - { n } — 正好 n 行 ($n > 0$)
 - { n, } — n 行或更多行 ($n \geq 0$)
 - { n, m } — 在 n 和 m (含) 行之间 ($0 \leq n \leq m, 0 < m$)
 - { , m } — 0 至 m (含) 行之间 ($m > 0$)

注意：不支持可能产生空匹配的模式。这样的模式的实例是 PATTERN (A*), PATTERN (A? B*), PATTERN (A{0,} B{0,} C*), 等。

Greedy & Reluctant 的量词

每个量词可以是 *贪婪的*（默认行为）或 *懒惰的*。贪婪的量词尝试匹配尽可能多的行，而懒惰的量词则尝试匹配尽可能少的行。

为了说明差异，可以通过查询查看以下示例，其中将贪婪量词应用于变量 B：

```
SELECT *
FROM Ticker
  MATCH_RECOGNIZE (
    PARTITION BY symbol
    ORDER BY rowtime
    MEASURES
      C.price AS lastPrice
    ONE ROW PER MATCH
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (A B* C)
    DEFINE
      A AS A.price > 10,
      B AS B.price < 15,
      C AS C.price > 12
  )
```

假设我们有以下输入：

```
symbol  tax  price      rowtime
```


=====	=====	=====	=====
XYZ	1	10	2018-09-17 10:00:02
XYZ	2	11	2018-09-17 10:00:03
XYZ	1	12	2018-09-17 10:00:04
XYZ	2	13	2018-09-17 10:00:05
XYZ	1	14	2018-09-17 10:00:06
XYZ	2	16	2018-09-17 10:00:07

上面的模式将产生以下输出：

symbol	lastPrice
=====	=====
XYZ	16

将 **B ***修改为 **B *?** 的同一查询，这意味着 **B ***应该是懒惰的，将生成：

symbol	lastPrice
=====	=====
XYZ	13
XYZ	16

模式变量 **B** 仅匹配价格为 **12** 的行，而不是吞噬价格为 **12**、**13** 和 **14** 的行。

注意：不可以对模式的最后一个变量使用贪婪量词。 因此，不允许使用类似（**A B ***）的模式。 可以通过引入条件为 **B** 的人工状态（例如 **C**）来轻松解决此问题。因此，您可以使用类似以下的查询：

```
PATTERN (A B* C)
DEFINE
    A AS condA(),
    B AS condB(),
    C AS NOT condB()
```

注意：目前不支持可选的懒惰量词（**A??**或 **A{0,1}?**）。

时间限制

特别是对于流使用情况，通常需要在给定的时间段内完成模式。这允许限制 **Flink** 必须在内部保持的总体状态大小，即使在贪婪的量词的情况下也是如此。

因此，**Flink SQL** 支持附加的（非标准 **SQL**）**WITHIN** 子句，用于定义模式的时间约束。可以在 **PATTERN** 子句之后定义该子句，该子句的间隔为毫秒。

如果潜在匹配的第一个事件和最后一个事件之间的时间长于给定值，则此类匹配将不会附加到结果表中。

注意：通常推荐使用该 **WITHIN** 子句，因为它有助于 **Flink** 进行有效的内存管理。一旦达到阈值，即可修剪基础状态。

注意：但是，**WITHIN** 子句不是 **SQL** 标准的一部分。建议的处理时间限制的方法将来可能会更改。

以下示例查询说明了 **WITHIN** 子句的用法：

```
SELECT *
FROM Ticker
  MATCH_RECOGNIZE (
    PARTITION BY symbol
    ORDER BY rowtime
    MEASURES
      C.rowtime AS dropTime,
      A.price - C.price AS dropDiff
    ONE ROW PER MATCH
    AFTER MATCH SKIP PAST LAST ROW
    PATTERN (A B* C) WITHIN INTERVAL '1' HOUR
    DEFINE
      B AS B.price > A.price - 10
      C AS C.price < A.price - 10
  )
```

该查询检测到在 1 小时的间隔内价格下降了 10。

假设该查询用于分析以下行情清单数据：

symbol	rowtime	price	tax
=====	=====	=====	=====
'ACME'	'01-Apr-11 10:00:00'	20	1
'ACME'	'01-Apr-11 10:20:00'	17	2
'ACME'	'01-Apr-11 10:40:00'	18	1
'ACME'	'01-Apr-11 11:00:00'	11	3
'ACME'	'01-Apr-11 11:20:00'	14	2
'ACME'	'01-Apr-11 11:40:00'	9	1
'ACME'	'01-Apr-11 12:00:00'	15	1
'ACME'	'01-Apr-11 12:20:00'	14	2
'ACME'	'01-Apr-11 12:40:00'	24	2
'ACME'	'01-Apr-11 13:00:00'	1	2
'ACME'	'01-Apr-11 13:20:00'	19	1

该查询将产生以下结果：

symbol	dropTime	dropDiff
=====	=====	=====
'ACME'	'01-Apr-11 13:00:00'	14

结果行表示价格从 15（在 01-Apr-11 12:00:00）下降到 1（在 01-Apr-11 13:00:00）。该 dropDiff 列包含价格差距。

请注意，即使价格也下降了较高的值，例如 11（01-Apr-11 10:00:00 和 01-Apr-11 11:40:00 之间的），这两个事件的时间差大于 1 小时。因此，他们不会产生匹配。

输出模式


该输出模式描述了每一个找到的匹配应发出多少行。SQL 标准描述了两种模式：

- ALL ROWS PER MATCH
- ONE ROW PER MATCH。


当前，唯一受支持的输出模式是 ONE ROW PER MATCH 始终为每个找到的匹配项生成一个输出摘要行。

输出行的模式将是特定顺序的[partitioning columns] + [measures columns] 串联。

以下示例显示了定义为的查询的输出：



```
SELECT *
FROM Ticker
  MATCH_RECOGNIZE (
    PARTITION BY symbol
    ORDER BY rowtime
    MEASURES
      FIRST(A.price) AS startPrice,
      LAST(A.price) AS topPrice,
      B.price AS lastPrice
    ONE ROW PER MATCH
    PATTERN (A+ B)
    DEFINE
      A AS LAST(A.price, 1) IS NULL OR A.price > LAST(A.price,
1),
      B AS B.price < LAST(A.price)
  )
```



对于以下输入行：

symbol	tax	price	rowtime
=====	=====	=====	=====
XYZ	1	10	2018-09-17 10:00:02
XYZ	2	12	2018-09-17 10:00:03
XYZ	1	13	2018-09-17 10:00:04
XYZ	2	11	2018-09-17 10:00:05

该查询将产生以下输出：

symbol	startPrice	topPrice	lastPrice
=====	=====	=====	=====
XYZ	10	13	11

模式识别按 `symbol` 列划分。即使未在 `MEASURES` 子句中明确提及，分区列也将添加到结果的开头。

模式导航

`DEFINE` 和 `MEASURES` 子句允许对于（可能）匹配的模式的行为列表中进行导航。

本节讨论用于声明条件或产生输出结果的导航。

模式变量引用

模式变量引用允许引用 `DEFINE` 或 `MEASURES` 子句中映射到特定模式变量的一组行。

例如，如果我们尝试将当前行与 `A` 匹配，则表达式 `A.price` 将描述一组到目前为止已映射到 `A` 的行，再加上当前行。如果 `DEFINE / MEASURES` 子句中的表达式需要一行（例如 `A. price` 或 `A.price > 10`），**它将选择属于相应集合的最后一个值。**

如果未指定任何模式变量（例如 `SUM(price)`），则表达式引用默认模式变量 `*`，该默认模式变量引用模式中的所有变量。换句话说，它创建了一个列表，列出了到目前为止映射到任何变量的所有行以及当前行。

如果未指定任何模式变量（例如 `SUM (price)`），则表达式引用默认模式变量 `*`，该默认模式变量引用模式中的所有变量。 换句话说，它创建了一个列表，列出了到目前为止映射到任何变量的所有行以及当前行。

例子

对于更详尽的示例，可以看看以下模式和相应条件：

```
PATTERN (A B+)
DEFINE
  A AS A.price > 10,
  B AS B.price > A.price AND SUM(price) < 100 AND SUM(B.price) < 80
```

下表描述了如何为每个传入事件评估这些条件。

该表包括以下列：

- `#` - 行标识符唯一地识别在列表中的传入行 `[A.price]/ [B.price]/ [price]`。
- `price` - 传入行的价格。
- `[A.price]/ [B.price]/ [price]` - 描述在 `DEFINE` 子句中用于评估条件的行的列表。
- `Classifier` - 当前行的分类器，指示该行映射到的模式变量。
- `A.price/ B.price/ SUM(price)/ SUM(B.price)` - 描述这些表达式求值后的结果。

#	price	Classifier	[A.price]	[B.price]	[price]	A.price	B.price	SUM(price)	SUM(B.price)
#1	10	-> A	#1	-	-	10	-	-	-
#2	15	-> B	#1	#2	#1, #2	10	15	25	15
#3	20	-> B	#1	#2, #3	#1, #2, #3	10	20	45	35
#4	31	-> B	#1	#2, #3, #4	#1, #2, #3, #4	10	31	76	66
#5	35		#1	#2, #3, #4, #5	#1, #2, #3, #4, #5	10	35	111	101

从表中可以看出，第一行映射到模式变量 **A**，随后的行映射到模式变量 **B**。但是，最后一行不满足 **B** 条件，因为所有映射行 **SUM (price)** 和 **B** 中所有行的总和超过指定的阈值。

逻辑偏移

*逻辑偏移*使您可以在映射到特定模式变量的事件中进行导航。这可以用两个相应的函数表示：

偏移功能	描述
LAST(variable.field, n)	返回字段的从被映射到该事件的值 \tilde{N} 个 最后的可变的元件。计数从映射的最后一个元素开始。
FIRST(variable.field, n)	返回事件的字段值，该事件已映射到变量的第 n 个元素。计数从映射的第一个元素开始。

偏移功能	描述

例子

对于更详尽的示例，可以看看以下模式和相应条件：

```
PATTERN (A B+)
DEFINE
  A AS A.price > 10,
  B AS (LAST(B.price, 1) IS NULL OR B.price > LAST(B.price, 1)) AND
      (LAST(B.price, 2) IS NULL OR B.price > 2 * LAST(B.price, 2))
```

下表描述了如何为每个传入事件评估这些条件。

该表包括以下列：

- price - 传入行的价格。
- Classifier - 当前行的分类器，指示该行映射到的模式变量。
- LAST(B.price, 1)/ LAST(B.price, 2) - 描述对这些表达式求值后的结果。

price	Classifier	LAST(B.price, 1)	LAST(B.price, 2)	Comment
10	-> A			
15	-> B	null	null	Notice that LAST(A.price, 1) is null because there is still nothing mapped to B.
20	-> B	15	null	
31	-> B	20	15	
35		31	20	Not mapped because $35 < 2 * 20$.

将默认模式变量与逻辑偏移量一起使用也可能很有意义。

在这种情况下，偏移量会考虑到目前为止映射的所有行：

```
PATTERN (A B? C)
DEFINE
  B AS B.price < 20,
```

C AS LAST(price, 1) < C.price			
price	Classifier	LAST(price, 1)	Comment
10	-> A		
15	-> B		
20	-> C	15	LAST(price, 1) 被评估为映射到变量 B 的行的价格。

如果第二行未映射到 B 变量，我们将得到以下结果：

price	Classifier	LAST(price, 1)	Comment
10	-> A		
20	-> C	10	LAST(price, 1) 被评估为映射到变量 A 的行的价格。

也可以在 FIRST / LAST 函数的第一个参数中使用多个模式变量引用。 这样，可以编写访问多个列的表达式。 但是，它们都必须使用相同的模式变量。 换句话说，必须在单个行中计算 LAST / FIRST 函数的值。

因此，可以使用 LAST(A.price * A.tax)，但 LAST(A.price * B.tax) 不允许类似的表达式。

匹配后策略

AFTER MATCH SKIP 子句指定在找到完全匹配项之后从何处开始新的匹配过程。

有四种不同的策略：


- SKIP PAST LAST ROW - 在当前匹配的最后一行之后的下一行恢复模式匹配。
- SKIP TO NEXT ROW - 继续搜索新的 MATCH，MATCH 从 MATCH 开始行的下一行开始。
- SKIP TO LAST variable - 在映射到指定模式变量的最后一行恢复模式匹配。
- SKIP TO FIRST variable - 在映射到指定模式变量的第一行中恢复模式匹配。

这也是指定单个事件可以属于多少个匹配项的方法。例如，使用该 SKIP PAST LAST ROW 策略，每个事件最多只能属于一个 MATCH。


例子

为了更好地理解这些策略之间的差异，可以看一下以下示例。


对于以下输入行：




symbol	tax	price	rowtime
=====	=====	=====	=====
XYZ	1	7	2018-09-17 10:00:01
XYZ	2	9	2018-09-17 10:00:02
XYZ	1	10	2018-09-17 10:00:03
XYZ	2	5	2018-09-17 10:00:04
XYZ	2	17	2018-09-17 10:00:05
XYZ	2	14	2018-09-17 10:00:06



我们使用不同的策略评估以下查询：



```
SELECT *
FROM Ticker
    MATCH_RECOGNIZE (
        PARTITION BY symbol
        ORDER BY rowtime
        MEASURES
            SUM(A.price) AS sumPrice,
            FIRST(rowtime) AS startTime,
            LAST(rowtime) AS endTime
        ONE ROW PER MATCH
        [AFTER MATCH STRATEGY]
        PATTERN (A+ C)
        DEFINE
            A AS SUM(A.price) < 30
    )
```



该查询返回映射到 **A** 的所有行的价格的总和以及整体匹配的第一个和最后一个时间戳。

该查询将根据 **AFTER MATCH** 所使用的策略产生不同的结果：


AFTER MATCH SKIP PAST LAST ROW

symbol	sumPrice	startTime	endTime
=====	=====	=====	=====
XYZ	26	2018-09-17 10:00:01	2018-09-17 10:00:04
XYZ	17	2018-09-17 10:00:05	2018-09-17 10:00:06


第一个结果与 #1, #2, #3, #4 行匹配。

第二个结果与 # 5, # 6 行匹配。

AFTER MATCH SKIP TO NEXT ROW



symbol	sumPrice	startTime	endTime
=====	=====	=====	=====
XYZ	26	2018-09-17 10:00:01	2018-09-17 10:00:04
XYZ	24	2018-09-17 10:00:02	2018-09-17 10:00:05
XYZ	15	2018-09-17 10:00:03	2018-09-17 10:00:05
XYZ	22	2018-09-17 10:00:04	2018-09-17 10:00:06
XYZ	17	2018-09-17 10:00:05	2018-09-17 10:00:06



同样，第一个结果与行 # 1, # 2, # 3, # 4 相匹配。

与之前的策略相比，下一个匹配项在下一个匹配项中再次包含了第 2 行。因此，第二个结果与行 # 2, # 3, # 4, # 5 相匹配。

第三个结果与行 # 3, # 4, # 5 相匹配。

第四结果与行 # 4, # 5, # 6 相匹配。

最后的结果与 # 5, # 6 行匹配。

AFTER MATCH SKIP TO LAST A

symbol	sumPrice	startTime	endTime
=====	=====	=====	=====
XYZ	26	2018-09-17 10:00:01	2018-09-17 10:00:04
XYZ	15	2018-09-17 10:00:03	2018-09-17 10:00:05
XYZ	22	2018-09-17 10:00:04	2018-09-17 10:00:06
XYZ	17	2018-09-17 10:00:05	2018-09-17 10:00:06

同样，第一个结果与行 # 1, # 2, # 3, # 4 相匹配。

与之前的策略相比，下一个匹配项仅包含下一个匹配项的第 3 行（映射到 A）。因此，第二个结果与行 # 3, # 4, # 5 相匹配。

第三个结果与 # 4, # 5, # 6 行匹配。

最后的结果与 # 5, # 6 行匹配。

AFTER MATCH SKIP TO FIRST A

这种组合将产生运行时异常，因为总是会尝试从上一个 MATCH 开始的地方开始一个新比赛。这将产生无限循环，因此被禁止。

必须记住，在采用该 SKIP TO FIRST/LAST variable 策略的情况下，可能没有行映射到该变量（例如，针对模式 A*）。在这种情况下，将抛出运行时异常，因为标准要求有效行才能继续匹配。

时间属性

为了在上进行一些后续查询，MATCH_RECOGNIZE 可能需要使用[时间属性](#)。要选择这些功能，有两个功能：

功能	描述
MATCH_ROW TIME()	返回映射到给定模式的最后一行的时间戳。 结果属性是一个行时间属性，可以在随后的基于时间的操作（例如带时间窗口的 join 和分组窗口或整个窗口聚合）中使用。
MATCH_PRO CTIME()	返回一个 proctime 属性，该属性可以在随后的基于时间的操作（例如带时间窗口的 join 和分组窗口或整个窗口聚合）中使用。

控制内存消耗

在编写 MATCH_RECOGNIZE 查询时，内存消耗是一个重要的考虑因素，因为潜在匹配的空间是以类似于广度优先的方式构建的。考虑到这一点，必须确保模式可以完成。最好将合理数量的行映射到匹配项，因为它们必须适合内存。

例如，模式必须没有接受每行的上限的量词。这样的模式可能看起来像这样：

```
PATTERN (A B+ C)
DEFINE
  A as A.price > 10,
  C as C.price > 20
```

该查询会将每个传入的行映射到该 B 变量，因此将永远不会完成。该查询可以解决，例如，通过否定条件 c：

```
PATTERN (A B+ C)
DEFINE
  A as A.price > 10,
  B as B.price <= 20,
  C as C.price > 20
```

或使用[懒惰的量词](#)：

```
PATTERN (A B+? C)
DEFINE
  A as A.price > 10,
  C as C.price > 20
```

注意：请注意，`MATCH_RECOGNIZE` 子句不使用配置的状态保留时间。为此，可能需要使用 `WITHIN` 子句。

已知局限性

Flink 对 `MATCH_RECOGNIZE` 子句的实现是一项持续的工作，并且尚不支持 SQL 标准的某些功能。

不支持的功能包括：

- 模式表达式：
 - **Pattern groups** - 这意味着例如：量词不能应用于模式的子序列。因此，`(A (B C) +)` 不是有效的模式。
 - **Alterations** - 像 `PATTERN ((A B | C D) E)` 这样的模式，这意味着在寻找 E 行之前必须找到子序列 A B 或 C D。
 - **PERMUTE operator** - 这等同于它应用于例如的所有变量的排列 模式 `(PERMUTE (A, B, C))` = 模式 `(A B C | A C B | B A C | B C A | C A B | C B A)`。
 - **Anchors** - `^`, `$`, 表示分区的开始/结束，在流上下文中没有意义，将不被支持。
 - **Exclusion** - `PATTERN ({-A-} B)` 表示将查找 A，但不参与输出。这仅适用于“每行所有行”模式。
 - **Reluctant optional quantifier** - `PATTERNA??` 仅支持贪婪的可选量词。
- `ALL ROWS PER MATCH` output mode - 为参与创建匹配项的每一行产生一个输出行。这也意味着：
 - 该 `MEASURES` 子句唯一受支持的语义是 `FINAL`
 - `CLASSIFIER` 尚不支持该参数返回将行映射到的模式变量。
- `SUBSET` - 这允许创建模式变量的逻辑组，并在 `DEFINE` 和 `MEASURES` 子句中使用这些组。
- **hysical offsets** - `PREV / NEXT`，它索引所有可见的事件，而不是仅索引那些映射到模式变量的事件（在逻辑偏移情况下）。
- **Extracting time attributes** - 当前没有可能为后续的基于时间的操作获取时间属性。
- `MATCH_RECOGNIZE` 仅 SQL 支持。Table API 中没有对应功能。
- **Aggregations**:
 - 不支持不同的聚合。

【翻译】Flink Table Api & SQL —Streaming 概念 —— 查询配置

本文翻译自官网：Query Configuration https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/streaming/query_configuration.html

Flink Table Api & SQL 翻译目录

表 API 和 SQL 查询具有相同的语义，无论其输入是有界批处理输入还是无界流输入。在许多情况下，对流输入的连续查询能够计算与脱机计算的结果相同的准确结果。但是，这在一般情况下是不可能的，因为连续查询必须限制它们所维护的状态的大小，以避免存储空间用完并能

够长时间处理无限制的流数据。结果，根据输入数据和查询本身的特征，连续查询可能只能提供近似结果。

Flink 的 Table API 和 SQL 界面提供了用于调整连续查询的准确性和资源消耗的参数。通过 QueryConfig 对象指定参数。QueryConfig 可以从 TableEnvironment 获取，并在转换表时（即，将其转换为 [DataStream](#) 或通过 [TableSink](#) 发出时）传入。



```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tableEnv = StreamTableEnvironment.create(env)

// obtain query configuration from TableEnvironment
val qConfig: StreamQueryConfig = tableEnv.queryConfig
// set query parameters
qConfig.withIdleStateRetentionTime(Time.hours(12), Time.hours(24))

// define query
val result: Table = ???

// create TableSink
val sink: TableSink[Row] = ???

// register TableSink
tableEnv.registerTableSink(
    "outputTable",           // table name
    Array[String](...),     // field names
    Array[TypeInformation[_]](...), // field types
    sink)                   // table sink

// emit result Table via a TableSink
result.insertInto("outputTable", qConfig)

// convert result Table into a DataStream[Row]
val stream: DataStream[Row] = result.toAppendStream[Row](qConfig)
```



在下文中，我们描述的参数 QueryConfig 以及它们如何影响查询的准确性和资源消耗。

空闲状态保留时间

许多查询在一个或多个关键属性上聚合或 join 记录。在流上执行这种查询时，连续查询需要收集记录或维护每个键的部分结果。如果输入流的密钥域正在发展，即，活动密钥值随时间而变化，则随着观察到越来越多的不同密钥，连续查询将累积越来越多的状态。但是，一段时间后，密钥通常变得不活动，并且它们的相应状态变得陈旧且无用。

例如，以下查询计算每个会话的点击次数。

```
SELECT sessionId, COUNT(*) FROM clicks GROUP BY sessionId;
```

`sessionId` 属性用作分组键，连续查询会为其观察到的每个 `sessionId` 保持计数。`sessionId` 属性会随着时间的推移而发展，并且 `sessionId` 值仅在会话结束之前（即一段有限的时间段内）才有效。但是，连续查询无法了解 `sessionId` 的此属性，并且期望每个 `sessionId` 值都可以在任何时间出现。它为每个观察到的 `sessionId` 值维护一个计数。因此，随着观察到越来越多的 `sessionId` 值，查询的总状态大小不断增长。

空闲状态保留时间参数定义密钥状态保留多长时间而不被更新，然后再将其删除。对于上一个示例查询，`sessionId` 的计数将在配置的时间段内未更新时立即删除。

通过删除键的状态，连续查询完全忘记了它之前已经看过该键。如果处理了带有键的记录（其状态之前已被删除），则该记录将被视为具有相应键的第一条记录。对于上面的示例，这意味着 `sessionId` 的计数将再次从 0 开始。

有两个参数可配置空闲状态保留时间：

- 最小空闲状态保留时间定义了非活动密钥的状态至少要保留多长时间才能被删除。
- 最大空闲状态保留时间定义非活动密钥的状态在被移除之前最多保持多长时间。

参数说明如下：

```
val qConfig: StreamQueryConfig = ???

// set idle state retention time: min = 12 hours, max = 24 hours
qConfig.withIdleStateRetentionTime(Time.hours(12), Time.hours(24))
```

清理状态需要额外的状态，这对于 `minTime` 和 `maxTime` 的差异越大变得越便宜。`minTime` 和 `maxTime` 之间的差异必须至少为 5 分钟。

【翻译】Flink Table Api & SQL —— 连接到外部系统

本文翻译自官网：Connect to External

Systems <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/connect.html>

Flink Table Api & SQL 翻译目录

注：本文对应代码段为多种格式，影响文章篇幅，所以只选取其中一种类似列入，全部内容见官网对应页面

Flink 的 Table API 和 SQL 程序可以连接到其他外部系统，以读取和写入批处理表和流式表。表源提供对存储在外部系统（例如数据库，键值存储，消息队列或文件系统）中的数据的访问。表接收器将表发送到外部存储系统。根据源和接收器的类型，它们支持不同的格式，例如 CSV，Parquet 或 ORC。

本页介绍如何声明内置表源/表接收器以及如何在 Flink 中注册它们。注册源或接收器后，可以通过 Table API 和 SQL 语句对其进行访问。

注意如果要实现自己的 *定制* 表源或接收器，请查看 [用户定义的源和接收器页面](#)。

- [依赖](#)
 - [连接器](#)
 - [格式](#)
- [总览](#)
- [表结构](#)
 - [行时间属性](#)
 - [类型字符串](#)
- [更新模式](#)
- [表连接器](#)
 - [文件系统连接器](#)
 - [Kafka 连接器](#)
 - [Elasticsearch 连接器](#)
 - [HBase 连接器](#)
 - [JDBC 连接器](#)
- [表格格式](#)
 - [CSV 格式](#)
 - [JSON 格式](#)
 - [Apache Avro 格式](#)
 - [旧的 CSV 格式](#)
- [更多 TableSources 和 TableSinks](#)
 - [OrcTableSource](#)
 - [CsvTableSink](#)
 - [JDBCAppendTableSink](#)
 - [CassandraAppendTableSink](#)

依赖

下表列出了所有可用的连接器和格式。它们的相互兼容性在[表连接器](#)和[表格式](#)的相应部分中进行了标记。下表提供了使用构建自动化工具（例如 **Maven** 或 **SBT**）和带有 **SQL JAR** 捆绑包的 **SQL Client** 的两个项目的依赖项信息。

连接器

Name	Version	Maven dependency	SQL Client JAR
Filesystem		Built-in	Built-in
Elasticsearch	6	flink-connector-elasticsearch6	Download
Apache Kafka	0.8	flink-connector-kafka-0.8	Not available

Name	Version	Maven dependency	SQL Client JAR
Apache Kafka	0.9	flink-connector-kafka-0.9	Download
Apache Kafka	0.10	flink-connector-kafka-0.10	Download
Apache Kafka	0.11	flink-connector-kafka-0.11	Download
Apache Kafka	0.11+ (universal)	flink-connector-kafka	Download
HBase	1.4.3	flink-hbase	Download
JDBC		flink-jdbc	Download

格式

Name	Maven dependency	SQL Client JAR
Old CSV (for files)	Built-in	Built-in
CSV (for Kafka)	flink-csv	Download
JSON	flink-json	Download
Apache Avro	flink-avro	Download

总览

从 Flink 1.6 开始，与外部系统的连接声明与实际实现分开了。

可以指定连接

- 以编程方式使用 org.apache.flink.table.descriptors 下的 Table & SQL API 的 Descriptor。
- 通过用于 SQL 客户端的 [YAML 配置文件](#) 声明。

这不仅可以更好地统一 API 和 SQL Client，还可以在自定义实现的情况下更好地扩展而不更改实际声明。

每个声明都类似于 SQL CREATE TABLE 语句。可以定义表的名称，表的结构，连接器以及用于连接到外部系统的数据格式。

连接器描述了存储表数据的外部系统。 可以在此处声明诸如 **Apache Kafka** 之类的存储系统或常规文件系统。 连接器可能已经提供了带有字段和结构的固定格式。

某些系统支持不同的数据格式。 例如，存储在 **Kafka** 或文件中的表可以使用 **CSV**，**JSON** 或 **Avro** 对行进行编码。 数据库连接器可能需要此处的表结构。 每个连接器都记录了存储系统是否需要格式的定义。 不同的系统还需要不同类型的格式（例如，面向列的格式与面向行的格式）。 该文档说明了哪些格式类型和连接器兼容。

表结构定义了公开给 **SQL** 查询的表的结构。 它描述了源如何将数据格式映射到表模式，反之亦然。 该模式可以访问由连接器或格式定义的字段。 它可以使用一个或多个字段来提取或插入时间属性。 如果输入字段没有确定性的字段顺序，则该结构将明确定义列名称，其顺序和来源。

随后的部分将更详细地介绍每个定义部分（连接器，格式和结构）。 以下示例显示如何传递它们：




```
Java、Scala 定义
tableEnvironment
    .connect(...)
    .withFormat(...)
    .withSchema(...)
    .inAppendMode()
    .registerTableSource("MyTable")
```

表格的类型（源，接收器或两者）决定了表格的注册方式。 对于两种表类型，表源和表接收器都以相同的名称注册。 从逻辑上讲，这意味着我们可以像读取常规 **DBMS** 中的表一样读取和写入该表。

对于流查询，更新模式声明了如何在动态表和存储系统之间进行通信以进行连续查询。

以下代码显示了如何连接到 **Kafka** 以读取 **Avro** 记录的完整示例。

Java、Scala 定义



```
tableEnvironment
    // declare the external system to connect to
    .connect(
        new Kafka()
            .version("0.10")
            .topic("test-input")
            .startFromEarliest()
            .property("zookeeper.connect", "localhost:2181")
            .property("bootstrap.servers", "localhost:9092")
    )

    // declare a format for this system
```



```

.withFormat(
    new Avro()
        .avroSchema(
            "{" +
            "  \"namespace\": \"org.myorganization\",\" +
            "  \"type\": \"record\",\" +
            "  \"name\": \"UserMessage\",\" +
            "    \"fields\": [\" +
            "      {\"name\": \"timestamp\", \"type\": \"string\"},\" +
            "      {\"name\": \"user\", \"type\": \"long\"},\" +
            "      {\"name\": \"message\", \"type\": [\"string\",
\"null\"]}\" +
            "    ]\" +
            "  }"
        )
    )

// declare the schema of the table
.withSchema(
    new Schema()
        .field("rowtime", Types.SQL_TIMESTAMP)
        .rowtime(new Rowtime()
            .timestampsFromField("timestamp")
            .watermarksPeriodicBounded(60000)
        )
        .field("user", Types.LONG)
        .field("message", Types.STRING)
    )

// specify the update-mode for streaming tables
.inAppendMode()

// register as source, sink, or both and under a name
.registerTableSource("MyUserTable");

```



两种方式都将所需的连接属性转换为标准化的基于字符串的键值对。所谓的表工厂根据键值对创建已配置的表源，表接收器和相应的格式。搜索完全匹配的表工厂时，会考虑到所有可通过 Java 服务提供商接口（SPI）找到的表工厂。



如果找不到给定属性的工厂或多个工厂匹配，则将引发异常，并提供有关考虑的工厂和支持的属性的其他信息。

表结构

表结构定义列的名称和类型，类似于 SQL CREATE TABLE 语句的列定义。此外，可以指定如何将列与表数据编码格式的字段进行映射。如果列名应与输入/输出格式不同，则字段的来源可能很重要。例如，一列 `user_name` 应该引用 JSON 格式的 `$$-user-name` 字段。此外，需要使用该结构将类型从外部系统映射到 Flink 的表示形式。如果是表接收器，则可确保仅将具有有效结构的数据写入外部系统。

以下示例显示了一个没有时间属性的简单架构，并且输入/输出到表列的一对一字段映射。

Java、Scala 定义

```
.withSchema(  
    new Schema()  
        .field("MyField1", Types.INT)           // required: specify the fields  
of the table (in this order)  
        .field("MyField2", Types.STRING)  
        .field("MyField3", Types.BOOLEAN)  
    )  

```

对于每个字段，除列的名称和类型外，还可以声明以下属性：

Java、Scala 定义

```
.withSchema(  
    new Schema()  
        .field("MyField1", Types.SQL_TIMESTAMP)  
        .proctime()           // optional: declares this field as a  
processing-time attribute  
        .field("MyField2", Types.SQL_TIMESTAMP)  
        .rowtime(...)         // optional: declares this field as a event-  
time attribute  
        .field("MyField3", Types.BOOLEAN)  
        .from("mf3")          // optional: original field in the input that  
is referenced/aliased by this field  
    )  

```

使用无界流表时，时间属性至关重要。因此，处理时间和事件时间（也称为“行时间”）属性都可以定义为架构的一部分。

有关 Flink 中时间处理（尤其是事件时间）的更多信息，我们建议使用常规[事件时间部分](#)。

行时间属性

为了控制表的事件时间行为，Flink 提供了预定义的时间戳提取器和水印策略。

支持以下时间戳提取器：

Java、Scala



```
// Converts an existing LONG or SQL_TIMESTAMP field in the input into
the rowtime attribute.
.rowtime(
    new Rowtime()
        .timestampsFromField("ts_field")    // required: original field
name in the input
)

// Converts the assigned timestamps from a DataStream API record into
the rowtime attribute
// and thus preserves the assigned timestamps from the source.
// This requires a source that assigns timestamps (e.g., Kafka
0.10+).
.rowtime(
    new Rowtime()
        .timestampsFromSource()
)

// Sets a custom timestamp extractor to be used for the rowtime
attribute.
// The extractor must extend
`org.apache.flink.table.sources.tsextractors.TimestampExtractor`.
.rowtime(
    new Rowtime()
        .timestampsFromExtractor(...)
)

```



支持以下水印策略:

Java、Scala



```
// Sets a watermark strategy for ascending rowtime attributes. Emits
a watermark of the maximum
// observed timestamp so far minus 1. Rows that have a timestamp
equal to the max timestamp
// are not late.
.rowtime(
    new Rowtime()
        .watermarksPeriodicAscending()
)

// Sets a built-in watermark strategy for rowtime attributes which
are out-of-order by a bounded time interval.

```



```

ROW<fieldtype, ...>          # unnamed row; e.g. ROW<VARCHAR,
                              #   with indexed fields names f0,
                              #   f1, ...
ROW<fieldname fieldtype, ...> # named row; e.g., ROW<myField
                              #   VARCHAR, myOtherField INT> that
                              #   is mapped to Flink's RowTypeInfo
POJO<class>                  # e.g.,
POJO<org.mycompany.MyPojoClass> that is mapped to Flink's
                              #   PojoTypeInfo
ANY<class>                    # e.g., ANY<org.mycompany.MyClass>
                              #   that is mapped to Flink's GenericTypeInfo
ANY<class, serialized>       # used for type information that is
                              #   not supported by Flink's Table & SQL API

```



更新模式

对于流查询，需要声明如何在动态表和外部连接器之间执行转换。更新模式指定应与外部系统交换的消息类型：

Append Mode: 在追加模式下，动态表和外部连接器仅交换 INSERT 消息。

Retract Mode: 在撤回模式下，动态表和外部连接器交换 ADD 和 RETRACT 消息。INSERT 更改被编码为 ADD 消息，DELETE 更改为 RETRACT 消息，UPDATE 更改为更新（先前）行的 RETRACT 消息，而 ADD 消息被更新（新）行的 ADD 消息。在此模式下，与 upsert 模式相反，不得定义密钥。但是，每个更新都包含两个效率较低的消息。

Upsert Mode: 在 upsert 模式下，动态表和外部连接器交换 UPSERT 和 DELETE 消息。此模式需要一个（可能是复合的）唯一密钥，通过该密钥可以传播更新。外部连接器需要了解唯一键属性，才能正确应用消息。INSERT 和 UPDATE 更改被编码为 UPSERT 消息。DELETE 更改为 DELETE 消息。与撤回流的主要区别在于 UPDATE 更改使用单个消息进行编码，因此效率更高。

注意：每个连接器的文档都说明了支持哪些更新模式。

```

Java、Scala
.connect(...)
    .inAppendMode()    // otherwise: inUpsertMode() or inRetractMode()

```

另请参阅[常规流概念](#)文档。

表连接器

Flink 提供了一组用于连接到外部系统的连接器。

请注意，并非所有连接器都可以批量和流式使用。此外，并非每个流连接器都支持每种流模式。因此，每个连接器都有相应的标记。格式标签表示连接器需要某种类型的格式。

文件系统连接器

Source: Batch, Source: Streaming, Append Mode, Sink: Batch, Sink: Streaming, Append Mode Format: CSV-only

文件系统连接器允许从本地或分布式文件系统读取和写入。 文件系统可以定义为：

```
Java、Scala
.connect(
    new FileSystem()
        .path("file:///path/to/whatever")    // required: path to a file
or directory
)
```

文件系统连接器本身包含在 Flink 中，不需要其他依赖项。 需要指定一种相应的格式，以便在文件系统中读取和写入行。

注意：确保包括 Flink File System 特定的依赖项。

注意文件系统源和流接收器仅是实验性的。 将来，我们将支持实际的流传输用例，即目录监视和存储桶输出。

Kafka 连接器

Source: Streaming Append Mode, Sink: Streaming Append Mode, Format: Serialization, Schema Format: Deserialization Schema

Kafka 连接器允许在 Apache Kafka 主题之间进行读写。 可以定义如下：

```

.connect(
    new Kafka()
        .version("0.11")    // required: valid connector versions are
                             // "0.8", "0.9", "0.10", "0.11", and
"universal"
        .topic("...")      // required: topic name from which the table
is read

    // optional: connector specific properties
    .property("zookeeper.connect", "localhost:2181")
    .property("bootstrap.servers", "localhost:9092")
    .property("group.id", "testGroup")

    // optional: select a startup mode for Kafka offsets
    .startFromEarliest()
    .startFromLatest()
    .startFromSpecificOffsets(...)

    // optional: output partitioning from Flink's partitions into
Kafka's partitions
```

```

        .sinkPartitionerFixed()           // each Flink partition ends up
in at-most one Kafka partition (default)
        .sinkPartitionerRoundRobin()     // a Flink partition is
distributed to Kafka partitions round-robin
        .sinkPartitionerCustom(MyCustom.class) // use a custom
FlinkKafkaPartitioner subclass
    )

```

Specify the start reading position: 默认情况下，Kafka 源将从 Zookeeper 或 Kafka 代理中的已提交组偏移中开始读取数据。您可以指定其他起始位置，这些位置与“Kafka Consumers 起始位置配置”部分中的配置相对应。

Flink-Kafka Sink Partitioning: 默认情况下，Kafka 接收器最多可以写入与其自身并行性一样多的分区（每个并行的接收器实例都写入一个分区）。为了将写入内容分配到更多分区或控制行到分区的路由，可以提供自定义接收器分区程序。循环分区器对于避免不平衡分区很有用。但是，这将导致所有 Flink 实例与所有 Kafka 代理之间的大量网络连接。

Consistency guarantees: 默认情况下，如果在启用检查点的情况下执行查询，则 Kafka 接收器会将具有至少一次保证的数据提取到 Kafka 主题中。

Kafka 0.10+ Timestamps: 从 Kafka 0.10 开始，Kafka 消息具有时间戳作为元数据，用于指定何时将记录写入 Kafka 主题。通过选择时间戳，可以将这些时间戳用于 rowtime 属性：分别是 YAML 中的 from-source 和 Java / Scala 中的 timestampsFromSource（）。

Kafka 0.11+ Versioning: 从 Flink 1.7 开始，Kafka 连接器定义独立于硬编码的 Kafka 版本。将通用连接器版本用作 Flink Kafka 连接器的通配符，该连接器与所有版本从 0.11 开始的 Kafka 兼容。

Elasticsearch 连接器

Sink: Streaming Append Mode, Sink: Streaming Upsert Mode, Format: JSON-only

Elasticsearch 连接器允许写入 Elasticsearch 搜索引擎的索引。

连接器可以在 upsert 模式下运行，以使用查询定义的密钥与外部系统交换 UPSERT / DELETE 消息。

对于 appen-only 查询，连接器还可以在追加模式下操作，以仅与外部系统交换 INSERT 消息。如果查询未定义任何键，则 Elasticsearch 自动生成一个键。

连接器可以定义如下：

```

connect (
    new Elasticsearch()
        .version("6")           // required: valid connector
versions are "6"

```

```

    .host("localhost", 9200, "http")    // required: one or more
Elasticsearch hosts to connect to
    .index("MyUsers")                  // required: Elasticsearch
index
    .documentType("user")              // required: Elasticsearch
document type

    .keyDelimiter("$")                 // optional: delimiter for composite
keys ("_" by default)
                                     //   e.g., "$" would result in IDs
"KEY1$KEY2$KEY3"
    .keyNullLiteral("n/a")            // optional: representation for null
fields in keys ("null" by default)

    // optional: failure handling strategy in case a request to
Elasticsearch fails (fail by default)
    .failureHandlerFail()              // optional: throws an exception
if a request fails and causes a job failure
    .failureHandlerIgnore()            //   or ignores failures and drops
the request
    .failureHandlerRetryRejected()     //   or re-adds requests that have
failed due to queue capacity saturation
    .failureHandlerCustom(...)         //   or custom failure handling
with a ActionRequestFailureHandler subclass

    // optional: configure how to buffer elements before sending them
in bulk to the cluster for efficiency
    .disableFlushOnCheckpoint()        // optional: disables flushing on
checkpoint (see notes below!)
    .bulkFlushMaxActions(42)           // optional: maximum number of
actions to buffer for each bulk request
    .bulkFlushMaxSize("42 mb")         // optional: maximum size of
buffered actions in bytes per bulk request
                                     //   (only MB granularity is
supported)
    .bulkFlushInterval(60000L)         // optional: bulk flush interval
(in milliseconds)

    .bulkFlushBackoffConstant()        // optional: use a constant
backoff type
    .bulkFlushBackoffExponential()     //   or use an exponential backoff
type
    .bulkFlushBackoffMaxRetries(3)     // optional: maximum number of
retries

```



```

        .bulkFlushBackoffDelay(30000L) // optional: delay between each
backoff attempt (in milliseconds)

        // optional: connection properties to be used during REST
communication to Elasticsearch
        .connectionMaxRetryTimeout(3) // optional: maximum timeout (in
milliseconds) between retries
        .connectionPathPrefix("/v1") // optional: prefix string to be
added to every REST communication
    )

```



Bulk flushing: 有关可选的刷新参数的特征的更多信息，请参见相应的[低级文档](#)。

Disabling flushing on checkpoint: 禁用后，接收器将不等待 Elasticsearch 在检查点上确认所有阻塞的操作请求。因此，接收器不会为动作请求的至少一次传递提供任何有力的保证。

Key extraction: Flink 自动从查询中提取有效键。例如，查询 `SELECT a, b, c FROM t GROUP BY a, b` 定义了字段 `a` 和 `b` 的组合键。Elasticsearch 连接器通过使用关键字定界符按查询中定义的顺序连接所有关键字字段，为每一行生成一个文档 ID 字符串。可以定义键字段的空文字的自定义表示形式。

注意：JSON 格式定义了如何为外部系统编码文档，因此，必须将其添加为依赖项。

HBase 连接器

Source: Batch, Sink: Batch, Sink: Streaming Append Mode, Sink: Streaming Upsert Mode, Temporal Join: Sync Mode

HBase 连接器允许读取和写入 HBase 群集。

连接器可以在 `upsert` 模式下运行，以使用查询定义的密钥与外部系统交换 `UPSERT` / `DELETE` 消息。

对于 `append-only` 查询，连接器还可以在追加模式下操作，以仅与外部系统交换 `INSERT` 消息。

连接器可以定义如下：



```

connector:
  type: hbase
  version: "1.4.3" # required: currently only support
"1.4.3"

  table-name: "hbase_table_name" # required: HBase table name

  zookeeper:

```

```

    quorum: "localhost:2181"      # required: HBase Zookeeper quorum
configuration
    znode.parent: "/test"        # optional: the root dir in
Zookeeper for HBase cluster.
                                # The default value is "/hbase".

    write.buffer.flush:
        max-size: "10mb"          # optional: writing option,
determines how many size in memory of buffered
                                # rows to insert per round trip.
This can help performance on writing to JDBC
                                # database. The default value is
"2mb".
        max-rows: 1000           # optional: writing option,
determines how many rows to insert per round trip.
                                # This can help performance on
writing to JDBC database. No default value,
                                # i.e. the default flushing is not
depends on the number of buffered rows.
        interval: "2s"           # optional: writing option, sets a
flush interval flushing buffered requesting
                                # if the interval passes, in
milliseconds. Default value is "0s", which means
                                # no asynchronous flush thread will
be scheduled.

```



Columns: HBase 表中的所有列系列必须声明为 ROW 类型，字段名称映射到列 **family** 名称，而嵌套的字段名称映射到列 **qualifier** 名称。无需在结构中声明所有族和限定符，用户可以声明必要的内容。除 ROW type 字段外，原子类型的唯一一个字段（例如 **STRING**，**BIGINT**）将被识别为表的行键。行键字段的名称没有任何限制。

Temporary join: 针对 HBase 的查找联接不使用任何缓存；始终总是通过 HBase 客户端直接查询数据。

Java/Scala/Python API: Java/Scala/Python APIs 还不支持

JDBC 连接器

Source: Batch, Sink: Batch, Sink: Streaming Append Mode, Sink: Streaming Upsert Mode, Temporal Join: Sync Mode

JDBC 连接器允许读取和写入 JDBC 客户端。

连接器可以在 **upsert** 模式下运行，以使用查询定义的密钥与外部系统交换 **UPSERT / DELETE** 消息。

对于 **append-only** 查询，连接器还可以在追加模式下操作，以仅与外部系统交换 **INSERT** 消息。

要使用 JDBC 连接器，需要选择一个实际的驱动程序来使用。 当前支持以下驱动程序：

支持的驱动：

Name	Group Id	Artifact Id	JAR
MySQL	mysql	mysql-connector-java	Download
PostgreSQL	org.postgresql	postgresql	Download
Derby	org.apache.derby	derby	Download

连接器可以定义如下：

```
connector:
  type: jdbc
  url: "jdbc:mysql://localhost:3306/flink-test"      # required: JDBC
DB url
  table: "jdbc_table_name"      # required: jdbc table name
  driver: "com.mysql.jdbc.Driver" # optional: the class name of the
JDBC driver to use to connect to this URL.
                                # If not set, it will automatically
be derived from the URL.

  username: "name"              # optional: jdbc user name and
password
  password: "password"

  read: # scan options, optional, used when reading from table
    partition: # These options must all be specified if any of them
is specified. In addition, partition.num must be specified. They
                # describe how to partition the table when reading in
parallel from multiple tasks. partition.column must be a numeric,
                # date, or timestamp column from the table in
question. Notice that lowerBound and upperBound are just used to
decide
                # the partition stride, not for filtering the rows in
table. So all rows in the table will be partitioned and returned.
                # This option applies only to reading.
    column: "column_name" # optional, name of the column used for
partitioning the input.
    num: 50               # optional, the number of partitions.
```

```

    lower-bound: 500      # optional, the smallest value of the
first partition.
    upper-bound: 1000    # optional, the largest value of the last
partition.
    fetch-size: 100      # optional, Gives the reader a hint as to
the number of rows that should be fetched
                        # from the database when reading per
round trip. If the value specified is zero, then
                        # the hint is ignored. The default value
is zero.

lookup: # lookup options, optional, used in temporary join
  cache:
    max-rows: 5000 # optional, max number of rows of lookup cache,
over this value, the oldest rows will
                # be eliminated. "cache.max-rows" and
"cache.ttl" options must all be specified if any
                # of them is specified. Cache is not enabled as
default.
    ttl: "10s"      # optional, the max time to live for each rows
in lookup cache, over this time, the oldest rows
                # will be expired. "cache.max-rows" and
"cache.ttl" options must all be specified if any of
                # them is specified. Cache is not enabled as
default.
    max-retries: 3   # optional, max retry times if lookup database
failed

write: # sink options, optional, used when writing into table
  flush:
    max-rows: 5000 # optional, flush max size (includes all
append, upsert and delete records),
                # over this number of records, will flush
data. The default value is "5000".
    interval: "2s" # optional, flush interval mills, over this
time, asynchronous threads will flush data.
                # The default value is "0s", which means no
asynchronous flush thread will be scheduled.
    max-retries: 3   # optional, max retry times if writing records
to database failed.

```



Upsert sink: Flink 自动从查询中提取有效键。例如，查询 `SELECT a, b, c FROM t GROUP BY a, b` 定义了字段 `a` 和 `b` 的组合键。如果将 `JDBC` 表用作 `upsert` 接收器，请确保查询的键是基础数据库的唯一键集或主键之一。这样可以保证输出结果符合预期。

Temporary Join: JDBC 连接器可以在临时联接中用作查找源。当前，仅支持同步查找模式。如果指定了查找缓存选项（`connector.lookup.cache.max-rows` 和 `connector.lookup.cache.ttl`），则必须全部指定它们。查找缓存用于通过首先查询缓存而不是将所有请求发送到远程数据库来提高临时连接 JDBC 连接器的性能。但是，如果来自缓存，则返回的值可能不是最新的。因此，这是吞吐量和正确性之间的平衡。

Writing: 默认情况下，`connector.write.flush.interval` 为 0s，`connector.write.flush.max-rows` 为 5000，这意味着对于低流量查询，缓冲的输出行可能不会长时间刷新到数据库。因此，建议设置间隔配置。

表格格式

Flink 提供了一组表格格式，可与表连接器一起使用。

格式标签表示与连接器匹配的格式类型。

CSV 格式


Format: Serialization Schema, Format: Deserialization Schema

CSV 格式旨在符合 Internet 工程任务组（IETF）提出的 RFC-4180（“逗号分隔值（CSV）文件的通用格式和 MIME 类型”）。

该格式允许读取和写入与给定格式模式对应的 CSV 数据。格式结构可以定义为 Flink 类型，也可以从所需的表结构派生。

如果格式模式等于表结构，则也可以自动派生该结构。这仅允许定义一次结构信息。格式的名称、类型和字段的顺序由表的结构确定。如果时间属性的来源不是字段，则将忽略它们。表模式中的 `from` 定义被解释为以该格式重命名的字段。

CSV 格式可以如下使用：

```

.withFormat(
    new Csv()

    // required: define the schema either by using type information
    .schema(Type.ROW(...))

    // or use the table's schema
    .deriveSchema()

    .fieldDelimiter(';')           // optional: field delimiter
character (',' by default)
    .lineDelimiter("\r\n")       // optional: line delimiter ("\n" by
default;
                                   // otherwise "\r" or "\r\n" are
allowed)
```

```

        .quoteCharacter('\\"')          // optional: quote character for
enclosing field values ('"' by default)
        .allowComments()                // optional: ignores comment lines
that start with '#' (disabled by default);
                                        //   if enabled, make sure to also
ignore parse errors to allow empty rows
        .ignoreParseErrors()            // optional: skip fields and rows
with parse errors instead of failing;
                                        //   fields are set to null in case
of errors
        .arrayElementDelimiter("|")    // optional: the array element
delimiter string for separating
                                        //   array and row element values
(";" by default)
        .escapeCharacter('\\"')        // optional: escape character for
escaping values (disabled by default)
        .nullLiteral("n/a")            // optional: null literal string
that is interpreted as a
                                        //   null value (disabled by
default)
    )

```



下表列出了可以读取和写入的受支持类型：

Supported Flink SQL Types
ROW
VARCHAR
ARRAY[_]
INT
BIGINT
FLOAT
DOUBLE
BOOLEAN
DATE
TIME

Supported Flink SQL Types
TIMESTAMP
DECIMAL
NULL (unsupported yet)

Numeric types: 值应该是数字，但字面量“ null”也可以理解。 空字符串被视为 null。 值也被修剪（开头/结尾随空白）。 数字是使用 Java 的 `valueOf` 语义解析的。 其他非数字字符串可能会导致解析异常。

String and time types: 值未修剪。 文字“ null”也可以理解。 时间类型必须根据 Java SQL 时间格式进行格式化，且精度为毫秒。 例如：日期为 2018-01-01，时间为 20:43:59，时间戳为 2018-01-01 20: 43: 59.999。

Boolean type: 值应为布尔值（“ true”，“ false”）字符串或“ null”。 空字符串被解释为 false。 值被修剪（开头/结尾随空白）。 其他值导致异常。

Nested types: 使用数组元素定界符可以为一级嵌套支持数组和行类型。

Primitive byte arrays: 基本字节数组以 Base64 编码表示形式处理。

Line endings: 对于行末未引号的字符串字段，即使对于基于行的连接器（如 Kafka）也应忽略行尾。

Escaping and quoting: 下表显示了使用*进行转义和使用'进行引用的转义和引用如何影响字符串的解析的示例：

CSV Field	Parsed String
123*'4**	123'4*
'123''4**'	123'4*
'a;b*'c'	a;b'c
'a;b''c'	a;b'c

确保将 CSV 格式添加为依赖项。


JSON 格式

Format: Serialization Schema, Format: Deserialization Schema


JSON 格式允许读取和写入与给定格式结构相对应的 JSON 数据。 格式结构可以定义为 Flink 类型，JSON 结构或从所需的表结构派生。 Flink 类型启用了更类似于 SQL 的定义并映射到相应的 SQL 数据类型。 JSON 格式允许更复杂和嵌套的结构。

如果格式结构等于表结构，则也可以自动派生该结构。这仅允许定义一次结构信息。格式的名称，类型和字段的顺序由表的结构确定。如果时间属性的来源不是字段，则将忽略它们。表结构中的 **from** 定义被解释为以该格式重命名的字段。

JSON 格式可以如下使用：



```
.withFormat(  
    new Json()  
        .failOnMissingField(true)    // optional: flag whether to fail if  
a field is missing or not, false by default  
  
    // required: define the schema either by using type information  
which parses numbers to corresponding types  
        .schema(Type.ROW(...))  
  
    // or by using a JSON schema which parses to DECIMAL and  
TIMESTAMP  
        .jsonSchema(  
            "{" +  
            "  type: 'object'," +  
            "  properties: {" +  
            "    lon: {" +  
            "      type: 'number'" +  
            "    }," +  
            "    rideTime: {" +  
            "      type: 'string'," +  
            "      format: 'date-time'" +  
            "    }" +  
            "  }" +  
            "}"  
        )  
  
    // or use the table's schema  
        .deriveSchema()  
    )  
)
```



下表显示了 JSON 模式类型到 Flink SQL 类型的映射：

JSON schema	Flink SQL
object	ROW
boolean	BOOLEAN

JSON schema	Flink SQL
array	ARRAY[_]
number	DECIMAL
integer	DECIMAL
string	VARCHAR
string with format: date-time	TIMESTAMP
string with format: date	DATE
string with format: time	TIME
string with encoding: base64	ARRAY[TINYINT]
null	NULL (unsupported yet)

当前，Flink 仅支持 JSON 模式规范 **draft-07** 的子集。尚不支持联合类型（以及 **allOf**，**anyOf** 和 **not**）。仅支持 **oneOf** 和类型数组用于指定可为空性。

支持链接到文档中通用定义的简单引用，如以下更复杂的示例所示：

```

{
  "definitions": {
    "address": {
      "type": "object",
      "properties": {
        "street_address": {
          "type": "string"
        },
        "city": {
          "type": "string"
        },
        "state": {
          "type": "string"
        }
      },
      "required": [
        "street_address",
        "city",
        "state"
      ]
    }
  }
}

```

```

    },
    "type": "object",
    "properties": {
      "billing_address": {
        "$ref": "#/definitions/address"
      },
      "shipping_address": {
        "$ref": "#/definitions/address"
      },
      "optional_address": {
        "oneOf": [
          {
            "type": "null"
          },
          {
            "$ref": "#/definitions/address"
          }
        ]
      }
    }
  }
}

```



Missing Field Handling: 默认情况下，缺少的 JSON 字段设置为 null。您可以启用严格的 JSON 解析，如果缺少字段，则将取消源（和查询）。

确保将 JSON 格式添加为依赖项。

Apache Avro 格式

Format: Serialization Schema, Format: Deserialization Schema

Apache Avro 格式允许读取和写入与给定格式模式相对应的 Avro 数据。格式结构可以定义为 Avro 特定记录的完全限定的类名，也可以定义为 Avro 架构字符串。如果使用了类名，则在运行时该类必须在类路径中可用。

Avro 格式可以如下使用：



```


.withFormat(
    new Avro()

    // required: define the schema either by using an Avro specific
    record class
    .recordClass(User.class)

    // or by using an Avro schema
    .avroSchema(

```

```
    "{" +
    "  \"type\": \"record\", \" +
    "  \"name\": \"test\", \" +
    "  \"fields\" : [\" +
    "    {\"name\": \"a\", \"type\": \"long\"}, \" +
    "    {\"name\": \"b\", \"type\": \"string\"} \" +
    "  ] \" +
    "\""
  )
)
```



Avro 类型映射到相应的 SQL 数据类型。 仅支持联合类型用于指定可为空性，否则它们将转换为 ANY 类型。 下表显示了映射：

Avro schema	Flink SQL
record	ROW
enum	VARCHAR
array	ARRAY[_]
map	MAP[VARCHAR, _]
union	non-null type or ANY
fixed	ARRAY[TINYINT]
string	VARCHAR
bytes	ARRAY[TINYINT]
int	INT
long	BIGINT
float	FLOAT
double	DOUBLE
boolean	BOOLEAN
int with logicalType: date	DATE
int with logicalType: time-millis	TIME

Avro schema	Flink SQL
int with logicalType: time-micros	INT
long with logicalType: timestamp-millis	TIMESTAMP
long with logicalType: timestamp-micros	BIGINT
bytes with logicalType: decimal	DECIMAL
fixed with logicalType: decimal	DECIMAL
null	NULL (unsupported yet)

Avro 使用 Joda-Time 表示特定记录类中的逻辑日期和时间类型。Joda-Time 依赖性不属于 Flink 的发行版。因此，在运行时，请确保 Joda-Time 和特定的记录类在您的类路径中。通过模式字符串指定的 Avro 格式不需要显示 Joda-Time。

确保添加 Apache Avro 依赖项。

旧的 CSV 格式

注意：仅用于原型制作！

旧的 CSV 格式允许使用文件系统连接器读取和写入以逗号分隔的行。


此格式描述了 Flink 的非标准 CSV 表源/接收器。将来，该格式将被适当的 RFC 兼容版本取代。写入 Kafka 时，请使用符合 RFC 的 CSV 格式。现在，将旧版本用于流/批处理文件系统操作。

```

.withFormat(
    new OldCsv()
        .field("field1", Types.STRING)      // required: ordered format
fields
        .field("field2", Types.TIMESTAMP)
        .fieldDelimiter(",")                // optional: string delimiter
        .lineDelimiter("\n")                // optional: string delimiter
        .quoteCharacter('')                 // optional: single character
for string values, empty by default
        .commentPrefix('#')                 // optional: string to indicate
comments, empty by default
        .ignoreFirstLine()                  // optional: ignore the first
line, by default it is not skipped
        .ignoreParseErrors()                // optional: skip records with
parse error instead of failing by default

```

)



旧的 CSV 格式包含在 Flink 中，不需要其他依赖项。

注意：目前，用于写入行的旧 CSV 格式受到限制。 仅支持将自定义字段定界符作为可选参数。

更多 TableSources 和 TableSinks

下表的源和接收器尚未迁移（或尚未完全迁移）到新的统一接口。

这些是 Flink 随附的其他 TableSources:

Class name	Maven dependency	Batch?	Streaming?	Description
OrcTableSource	flink-orc	Y	N	A TableSource for ORC files.


These are the additional TableSinks which are provided with Flink:

Class name	Maven dependency	Batch?	Streaming?	Description
CsvTableSink	flink-table	Y	Append	A simple sink for CSV files.
JDBCAppendTableSink	flink-jdbc	Y	Append	Writes a Table to a JDBC table.
CassandraAppendTableSink	flink-connector-cassandra	N	Append	Writes a Table to a Cassandra table.

OrcTableSource

OrcTableSource 读取 ORC 文件。 ORC 是用于结构化数据的文件格式，并以压缩的列表示形式存储数据。 ORC 具有很高的存储效率，并支持投影和滤镜下推。

创建一个 OrcTableSource，如下所示：



```
// create Hadoop Configuration
Configuration config = new Configuration();
```

```
OrcTableSource orcTableSource = OrcTableSource.builder()
    // path to ORC file(s). NOTE: By default, directories are
    recursively scanned.
    .path("file:///path/to/data")
    // schema of ORC files
    .forOrcSchema("struct<name:string,addresses:array<struct<street:string,zip:smallint>>>")
    // Hadoop configuration
    .withConfiguration(config)
    // build OrcTableSource
    .build();
```



注意：OrcTableSource 还不支持 ORC 的联合类型。

CsvTableSink

CsvTableSink 发出一个表到一个或多个 CSV 文件。

接收器仅支持 **append-only** 流表。它不能用于发出连续更新的表。有关详细信息，请参见 [表到流转换的文档](#)。发出流表时，行至少写入一次（如果启用了检查点），并且 CsvTableSink 不会将输出文件拆分为存储区文件，而是连续写入相同的文件。



```
CsvTableSink sink = new CsvTableSink(
    path,                // output path
    "|",                 // optional: delimit files by '|'
    1,                   // optional: write to a single file
    WriteMode.OVERWRITE); // optional: override existing files

tableEnv.registerTableSink(
    "csvOutputTable",
    // specify table schema
    new String[]{"f0", "f1"},
    new TypeInformation[]{Types.STRING, Types.INT},
    sink);

Table table = ...
table.insertInto("csvOutputTable");
```




JDBCAppendTableSink

JDBCAppendTableSink 发出到 JDBC 连接的表。接收器仅支持 **append-only** 流表。它不能用于发出连续更新的表。有关详细信息，请参见 [表到流转换的文档](#)。

JDBCAppendTableSink 将每个 **Table** 行至少插入一次到数据库表中（如果启用了检查点）。但是，您可以使用 **REPLACE** 或 **INSERT OVERWRITE** 指定插入查询，以执行对数据库的向上写入。

要使用 **JDBC** 接收器，必须将 **JDBC** 连接器依赖项（**flink-jdbc**）添加到项目中。然后，您可以使用 **JDBCAppendSinkBuilder** 创建接收器：



```
JDBCAppendTableSink sink = JDBCAppendTableSink.builder()
    .setDrivername("org.apache.derby.jdbc.EmbeddedDriver")
    .setDBUrl("jdbc:derby:memory:ebookshop")
    .setQuery("INSERT INTO books (id) VALUES (?)")
    .setParameterTypes(INT_TYPE_INFO)
    .build();

tableEnv.registerTableSink(
    "jdbcOutputTable",
    // specify table schema
    new String[]{"id"},
    new TypeInformation[]{Types.INT},
    sink);

Table table = ...
table.insertInto("jdbcOutputTable");
```


与使用 **JDBCOutputFormat** 相似，您必须显式指定 **JDBC** 驱动程序的名称，**JDBC** URL，要执行的查询以及 **JDBC** 表的字段类型。

CassandraAppendTableSink

CassandraAppendTableSink 向 **Cassandra** 表发出一个表。接收器仅支持 **append** 流表。它不能用于发出连续更新的表。有关详细信息，请参见表到流转换的文档。

如果启用了检查点，则 **CassandraAppendTableSink** 将所有行至少插入一次到 **Cassandra** 表中。但是，您可以将查询指定为 **upsert** 查询。

要使用 **CassandraAppendTableSink**，必须将 **Cassandra** 连接器依赖项（**flink-connector-cassandra**）添加到项目中。以下示例显示了如何使用 **CassandraAppendTableSink**。



```
ClusterBuilder builder = ... // configure Cassandra cluster
connection

CassandraAppendTableSink sink = new CassandraAppendTableSink(
    builder,
    // the query must match the schema of the table
    "INSERT INTO flink.myTable (id, name, value) VALUES (?, ?, ?)");
```

```
tableEnv.registerTableSink(  
    "cassandraOutputTable",  
    // specify table schema  
    new String[]{"id", "name", "value"},  
    new TypeInformation[]{Types.INT, Types.STRING, Types.DOUBLE},  
    sink);  
  
Table table = ...  
table.insertInto(cassandraOutputTable);
```



【翻译】Flink Table Api & SQL —— Table API

本文翻译自官网: Table API <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/tableApi.html>

Flink Table Api & SQL 翻译目录

Table API 是用于流和批处理的统一的关系 API。 Table API 查询可以在批处理或流输入上运行而无需修改。 Table API 是 SQL 语言的超集，是专门为与 Apache Flink 配合使用而设计的。 Table API 是用于 Scala 和 Java 的语言集成的 API。 Table API 查询不是将查询指定为 SQL 常见的 String 值，而是以 Java 或 Scala 中的语言嵌入样式定义，并具有 IDE 支持，例如自动完成和语法验证。

Table API 与 Flink 的 SQL 集成共享其 API 的许多概念和部分。 看一下 Common Concepts & API，了解如何注册表或创建 Table 对象。“流概念”页面讨论了流的特定概念，例如动态表和时间属性。

下面的示例假定具有属性（a, b, c, rowtime）的已注册表 Orders。 rowtime 字段是流中的逻辑时间属性，或者是批处理中的常规时间戳字段。

- 概述与范例
- 操作
 - 扫描,投影和过滤
 - 列操作
 - 聚合
 - joins
 - 设定算子
 - OrderBy, Offset 和 Fetch
 - Insert
 - Group Windows
 - Over Windows
 - 基于行的操作

- [Data Types](#)
- [表达式语法](#)

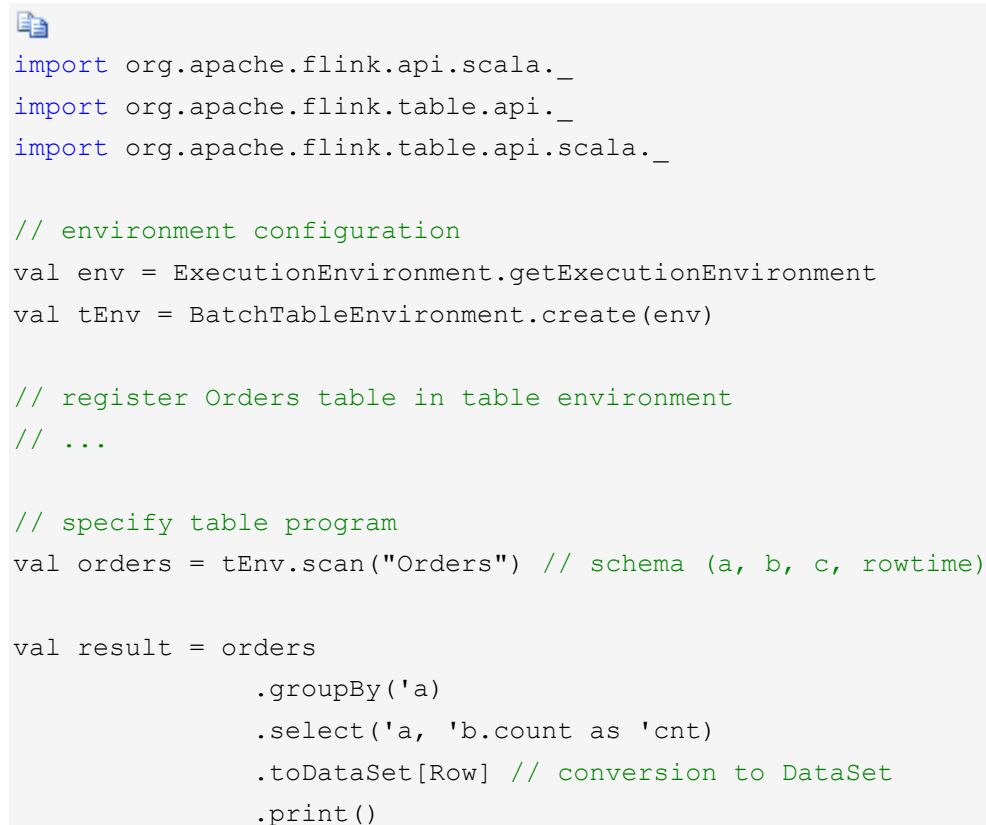
概述与范例

Table API 可用于 Scala 和 Java。Scala Table API 利用 Scala 表达式，Java Table API 基于已解析并转换为等效表达式的字符串。

以下示例显示了 Scala 和 Java Table API 之间的区别。该表程序在批处理环境中执行。它将扫描 Orders 表，按字段 a 进行分组，并计算每组的结果行。该表程序的结果将转换为 Row 类型的数据集并进行打印。

通过导入 `org.apache.flink.api.scala._` 和 `org.apache.flink.table.api.scala._` 来启用 Scala Table API。

以下示例显示了 Scala Table API 程序的构造方式。表属性是使用 Scala 符号引用的，Scala 符号以撇号 (') 开头。



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

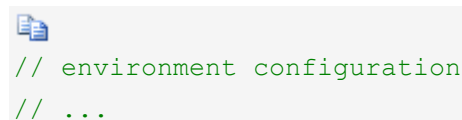
// environment configuration
val env = ExecutionEnvironment.getExecutionEnvironment
val tEnv = BatchTableEnvironment.create(env)

// register Orders table in table environment
// ...

// specify table program
val orders = tEnv.scan("Orders") // schema (a, b, c, rowtime)

val result = orders
    .groupBy('a)
    .select('a, 'b.count as 'cnt)
    .toDataSet[Row] // conversion to DataSet
    .print()
```

下一个示例显示了一个更复杂的 Table API 程序。程序再次扫描 Orders 表。它过滤空值，对 String 类型的字段 a 进行归一化，并针对每个小时计算并产生 a 平均帐单金额 b。



```
// environment configuration
// ...
```

```
// specify table program
val orders: Table = tEnv.scan("Orders") // schema (a, b, c, rowtime)

val result: Table = orders
    .filter('a.isNotNull && 'b.isNotNull && 'c.isNotNull)
    .select('a.lowerCase() as 'a, 'b, 'rowtime)
    .window(Tumble over 1.hour on 'rowtime as 'hourlyWindow)
    .groupBy('hourlyWindow, 'a)
    .select('a, 'hourlyWindow.end as 'hour, 'b.avg as
'avgBillingAmount)
```



由于 Table API 是用于批处理和流数据的统一 API，因此两个示例程序都可以在批处理和流输入上执行，而无需对表程序本身进行任何修改。在这两种情况下，只要流记录不晚，程序都会产生相同的结果（有关详细信息，请参见[流概念](#)）。

Operations

Table API 支持以下操作。请注意，不是所有的操作都可以批量和流式传输。它们被相应地标记。

Scan, Projection, and Filter

Operators	Description
Scan Batch Streaming	类似于 SQL 查询中的 FROM 子句。扫描已注册的表。 <code>val orders: Table = tableEnv.scan("Orders")</code>
Select Batch Streaming	类似于 SQL SELECT 语句。执行选择操作。 <code>val orders: Table = tableEnv.scan("Orders")</code> <code>val result = orders.select('a, 'c as 'd)</code> 您可以使用星号（*）充当通配符，选择表中的所有列。 <code>val orders: Table = tableEnv.scan("Orders")</code> <code>val result = orders.select('*)</code>
As Batch Streaming	Renames fields. <code>val orders: Table = tableEnv.scan("Orders").as('x, 'y, 'z, 't)</code>
Where / Filter Batch Streaming	类似于 SQL WHERE 子句。过滤掉未通过过滤谓词的行。

Operators	Description
ng	<pre>val orders: Table = tableEnv.scan("Orders") val result = orders.filter('a % 2 === 0) or val orders: Table = tableEnv.scan("Orders") val result = orders.where('b === "red")</pre>

Column Operations

Operators	Description
AddColumns Batch Streaming	<p>执行字段添加操作。 如果添加的字段已经存在，它将引发异常。</p> <pre>val orders = tableEnv.scan("Orders"); val result = orders.addColumn(concat('c, "Sunny"))</pre>
AddOrReplaceColumns Batch Streaming	<p>执行字段添加操作。 如果添加列名称与现有列名称相同，则现有字段将被替换。 此外，如果添加的字段具有重复的字段名称，则使用最后一个。</p> <pre>val orders = tableEnv.scan("Orders"); val result = orders.addOrReplaceColumns(concat('c, "Sunny") as 'desc)</pre>
DropColumns Batch Streaming	<p>执行字段删除操作。 字段表达式应该是字段引用表达式，并且只能删除现有字段。</p> <pre>val orders = tableEnv.scan("Orders"); val result = orders.dropColumns('b, 'c)</pre>
RenameColumns Batch Streaming	<p>执行字段重命名操作。 字段表达式应该是别名表达式，并且只能重命名现有字段。</p> <pre>val orders = tableEnv.scan("Orders"); val result = orders.renameColumns('b as 'b2, 'c as 'c2)</pre>

Aggregations

Operators	Description
GroupBy Aggregation Batch Streaming Result Updating	<p>类似于 SQL GROUP BY 子句。 使用以下正在运行的聚合运算符将分组键上的行分组，以逐行聚合行。</p> <pre>val orders: Table = tableEnv.scan("Orders") val result = orders.groupBy('a).select('a, 'b.sum as 'd)</pre> <p>注意：对于流式查询，根据聚合的类型和不同的分组键的数量，计算查询结果所需的状态可能会无限增长。 请提供具有有效保留间隔的查询配置，以防止出现过多的状态。 有关详细信息，请参见查询配置。</p>
GroupBy Window Aggregation Batch Streaming	<p>在 group window 和可能的一个或多个分组键上对表进行分组和聚集。</p> <pre>val orders: Table = tableEnv.scan("Orders") val result: Table = orders .window(Tumble over 5.minutes on 'rowtime as 'w) // define window .groupBy('a, 'w) // group by key and window .select('a, w.start, 'w.end, 'w.rowtime, 'b.sum as 'd) // access window properties and aggregate</pre>
Over Window Aggregation Streaming	<p>类似于 SQL OVER 子句。 基于前一行和后一行的窗口（范围），为每一行计算窗口聚合。 有关更多详细信息，请参见 Windows 部分。</p> <pre>val orders: Table = tableEnv.scan("Orders") val result: Table = orders // define window .window(Over partitionBy 'a orderBy 'rowtime preceding UNBOUNDED_RANGE following CURRENT_RANGE as 'w) .select('a, 'b.avg over 'w, 'b.max over 'w, 'b.min over 'w) // sliding aggregate</pre> <p>注意：必须在同一窗口（即相同的分区，排序和范围）上定义所有聚合。 当前，仅支持 PRECEDING（无边界和有界）到 CURRENT R</p>

Operators	Description
	<p>OW 范围的窗口。 目前尚不支持带有 FOLLOWING 的范围。 必须在单个时间属性上指定 ORDER BY。</p>
Distinct Aggregation Batch Streaming Result Updating	<p>类似于 SQL DISTINCT AGGREGATION 子句，例如 COUNT (DISTINCT a)。 不同的聚合声明聚合函数（内置或用户定义的）仅应用于不同的输入值。 可以将不同应用于 GroupBy 聚合，GroupBy 窗口聚合和 Over Window 聚合。</p> <pre>val orders: Table = tableEnv.scan("Orders"); // Distinct aggregation on group by val groupByDistinctResult = orders .groupBy('a) .select('a, 'b.sum.distinct as 'd) // Distinct aggregation on time window group by val groupByWindowDistinctResult = orders .window(Tumble over 5.minutes on 'rowtime as 'w).groupBy('a, 'w) .select('a, 'b.sum.distinct as 'd) // Distinct aggregation on over window val result = orders .window(Over partitionBy 'a orderBy 'rowtime preceding UNBOUNDED_RANGE as 'w) .select('a, 'b.avg.distinct over 'w, 'b.max over 'w, 'b.min over 'w)</pre> <p>用户定义的聚合函数也可以与 DISTINCT 修饰符一起使用。 要仅针对不同值计算聚合结果，只需向聚合函数添加 distinct 修饰符即可。</p> <pre>val orders: Table = tEnv.scan("Orders"); // Use distinct aggregation for user-defined aggregate functions val myUdagg = new MyUdagg(); orders.groupBy('users).select('users, myUdagg.distinct('points) as 'myDistinctResult);</pre> <p>注意：对于流式查询，计算查询结果所需的状态可能会无限增长，具体取决于不同字段的数量。 请提供具有有效保留间隔的查询配置，以防止出现过多的状态。 有关详细信息，请参见查询配置。</p>

Operators	Description
Distinct Batch Streaming Result Updating	<p>类似于 SQL DISTINCT 子句。 返回具有不同值组合的记录。<code>val orders: Table = tableEnv.scan("Orders")</code> <code>val result = orders.distinct()</code></p> <p>注意：对于流式查询，计算查询结果所需的状态可能会无限增长，具体取决于不同字段的数量。 请提供具有有效保留间隔的查询配置，以防止出现过多的状态。 如果启用了状态清除功能，那么 distinct 必须发出消息，</p> <p>以防止下游运算符过早地退出状态，这会导致 distinct 包含结果更新。 有关详细信息，请参见查询配置。</p>

Joins

Operators	Description
Inner Join Batch Streaming	<p>类似于 SQL JOIN 子句。 连接两个表。 两个表必须具有不同的字段名称，并且至少一个相等的联接谓词必须通过联接运算符或使用 where 或 filter 运算符进行定义。</p> <pre>val left = ds1.toTable(tableEnv, 'a, 'b, 'c) val right = ds2.toTable(tableEnv, 'd, 'e, 'f) val result = left.join(right).where('a === 'd).select('a, 'b, 'e)</pre> <p>注意：对于流式查询，根据不同输入行的数量，计算查询结果所需的状态可能会无限增长。 请提供具有有效保留间隔的查询配置，以防止出现过多的状态。 有关详细信息，请参见查询配置。</p>
Outer Join Batch Streaming Result Updating	<p>类似于 SQL LEFT / RIGHT / FULL OUTER JOIN 子句。 连接两个表。 两个表必须具有不同的字段名称，并且必须至少定义一个相等联接谓词。</p> <pre>val left = tableEnv.fromDataSet(ds1, 'a, 'b, 'c) val right = tableEnv.fromDataSet(ds2, 'd, 'e, 'f)</pre> <pre>val leftOuterResult = left.leftOuterJoin(right, 'a === 'd).select('a, 'b, 'e) val rightOuterResult = left.rightOuterJoin(right, 'a === 'd).select('a, 'b, 'e)</pre>

Operators	Description
	<pre>val fullOuterResult = left.fullOuterJoin(right, 'a === 'd).select('a, 'b, 'e)</pre> <p>注意：对于流式查询，根据不同输入行的数量，计算查询结果所需的状态可能会无限增长。 请提供具有有效保留间隔的查询配置，以防止出现过多的状态。 有关详细信息，请参见查询配置。</p>
Time-windowed Join in Batch Streaming	<p>注意：时间窗口联接是可以以流方式处理的常规联接的子集。</p> <p>时间窗联接需要至少一个等联接谓词和在两侧限制时间的联接条件。 可以通过两个适当的范围谓词（<, <=, >=, >）或比较两个输入表的相同类型的时间属性（即处理时间或事件时间）的单个相等谓词来定义这种条件。</p> <p>例如，以下谓词是有效的窗口连接条件：</p> <ul style="list-style-type: none"> 'ltime === 'rtime 'ltime >= 'rtime && 'ltime < 'rtime + 10.minutes <pre>val left = ds1.toTable(tableEnv, 'a, 'b, 'c, 'ltime.rowtime) val right = ds2.toTable(tableEnv, 'd, 'e, 'f, 'rtime.rowtime) val result = left.join(right) .where('a === 'd && 'ltime >= 'rtime - 5.minutes && 'ltime < 'rtime + 10.minutes) .select('a, 'b, 'e, 'ltime)</pre>
Inner Join with Table Function (UDTF) Batch Streaming	<p>用表函数的结果联接表。 左（外）表的每一行都与表函数的相应调用产生的所有行连接在一起。 如果左表（外部）的表函数调用返回空结果，则该行将被删除。</p> <pre>// instantiate User-Defined Table Function val split: TableFunction[_] = new MySplitUDTF() // join val result: Table = table .joinLateral(split('c) as ('s, 't, 'v)) .select('a, 'b, 's, 't, 'v)</pre>

Operators	Description
Left Outer Join with Table Function (UDTF) Batch Streaming	<p>用表函数的结果联接表。左（外）表的每一行都与表函数的相应调用产生的所有行连接在一起。如果表函数调用返回空结果，则将保留对应的外部行，并用空值填充结果。</p> <p>注意：当前，左外部联接的表函数的谓词只能为空或字面值 true。</p> <pre>// instantiate User-Defined Table Function val split: TableFunction[_] = new MySplitUDTF() // join val result: Table = table .leftOuterJoinLateral(split('c) as ('s, 't, 'v)) .select('a, 'b, 's, 't, 'v)</pre>
Join with Temporal Table Streaming	<p>时态表是跟踪其随时间变化的表。</p> <p>时态表功能提供对特定时间点时态表状态的访问。使用临时表函数联接表的语法与使用表函数进行内部联接的语法相同。</p> <p>当前仅支持使用临时表的内部联接。</p> <pre>val ratesHistory = tableEnv.scan("RatesHistory") // register temporal table function with a time attribute and primary key val rates = ratesHistory.createTemporalTableFunction('r_proctime, 'r_currency) // join with "Orders" based on the time attribute and key val orders = tableEnv.scan("Orders") val result = orders .joinLateral(rates('o_rowtime), 'r_currency === 'o_currency)</pre> <p>有关更多信息，请检查更详细的时态表概念描述。</p>

Set Operations

Operators	Description
Union Batch	<p>类似于 SQL UNION 子句。 合并两个已删除重复记录的表，两个表必须具有相同的字段类型。</p> <pre>val left = ds1.toTable(tableEnv, 'a, 'b, 'c) val right = ds2.toTable(tableEnv, 'a, 'b, 'c) val result = left.union(right)</pre>
UnionAll Batch Streaming	<p>类似于 SQL UNION ALL 子句。 合并两个表，两个表必须具有相同的字段类型。</p> <pre>val left = ds1.toTable(tableEnv, 'a, 'b, 'c) val right = ds2.toTable(tableEnv, 'a, 'b, 'c) val result = left.unionAll(right)</pre>
Intersect Batch	<p>类似于 SQL INTERSECT 子句。 相交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多次出现，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。</p> <pre>val left = ds1.toTable(tableEnv, 'a, 'b, 'c) val right = ds2.toTable(tableEnv, 'e, 'f, 'g) val result = left.intersect(right)</pre>
IntersectAll Batch	<p>类似于 SQL INTERSECT ALL 子句。 IntersectAll 返回两个表中都存在的记录。 如果一个记录在两个表中都存在一次以上，则返回的次数与两个表中存在的次数相同，即，结果表可能具有重复的记录。 两个表必须具有相同的字段类型。</p> <pre>val left = ds1.toTable(tableEnv, 'a, 'b, 'c) val right = ds2.toTable(tableEnv, 'e, 'f, 'g) val result = left.intersectAll(right)</pre>
Minus Batch	<p>类似于 SQL EXCEPT 子句。 减号从左表返回不存在于右表中的记录。 左表中的重复记录仅返回一次，即删除了重复记录。 两个表必须具有相同的字段类型。</p> <pre>val left = ds1.toTable(tableEnv, 'a, 'b, 'c) val right = ds2.toTable(tableEnv, 'a, 'b, 'c) val result = left.minus(right)</pre>
MinusAll Batch	<p>类似于 SQL EXCEPT ALL 子句。 MinusAll 返回右表中不存在的记录。 将返回（n-m）次在左侧表中出现 n 次，在右侧表中出现 m 次的</p>

Operators	Description
	<p>记录，即删除与右侧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。</p> <pre>val left = ds1.toTable(tableEnv, 'a, 'b, 'c) val right = ds2.toTable(tableEnv, 'a, 'b, 'c) val result = left.minusAll(right)</pre>
In Batch Streaming	<p>类似于 SQL IN 子句。 如果给定的表子查询中存在表达式，则 In 返回 true。 子查询表必须由一列组成。 此列必须与表达式具有相同的数据类型。</p> <pre>val left = ds1.toTable(tableEnv, 'a, 'b, 'c) val right = ds2.toTable(tableEnv, 'a) val result = left.select('a, 'b, 'c).where('a.in(right))</pre> <p>注意：对于流查询，该操作将在联接和组操作中重写。 根据不同输入行的数量，计算查询结果所需的状态可能会无限增长。 请提供具有有效保留间隔的查询配置，以防止出现过多的状态。 有关详细信息，请参见查询配置。</p>

OrderBy, Offset & Fetch

Operators	Description
Order By Batch	<p>类似于 SQL ORDER BY 子句。 返回在所有并行分区上全局排序的记录。</p> <pre>val in = ds.toTable(tableEnv, 'a, 'b, 'c) val result = in.orderBy('a.asc)</pre>
Offset & Fetch Batch	<p>与 SQL OFFSET 和 FETCH 子句类似。 偏移量和提取限制从排序结果返回的记录数。 偏移和提取在技术上是 Order By 运算符的一部分，因此必须在其之前。</p> <pre>val in = ds.toTable(tableEnv, 'a, 'b, 'c) // returns the first 5 records from the sorted result val result1: Table = in.orderBy('a.asc).fetch(5) // skips the first 3 records and returns all following records from the sorted result val result2: Table = in.orderBy('a.asc).offset(3)</pre>

Operators	Description
	<pre>// skips the first 10 records and returns the next 5 // records from the sorted result val result3: Table = in.orderBy('a.asc).offset(10).fetch(5)</pre>

Insert

Operators	Description
Insert Into Batch Streaming	<p>与 SQL 查询中的 INSERT INTO 子句相似。 在已插入的输出表中执行插入。</p> <p>输出表必须在 <code>TableEnvironment</code> 中注册（请参阅注册 TableSink）。 此外，已注册表的架构必须与查询的架构匹配。</p> <pre>val orders: Table = tableEnv.scan("Orders") orders.insertInto("OutOrders")</pre>

Group Windows

Group 窗口根据时间或行计数间隔将组行聚合为有限的组，并每组评估一次聚合函数。 对于批处理表，窗口是按时间间隔对记录进行分组的便捷 **shortcut** 。

Windows 是使用 `window (w: GroupWindow)` 子句定义的，并且需要使用 `as` 子句指定的别名。 为了按窗口对表进行分组，必须像常规分组属性一样在 `groupBy (...)` 子句中引用窗口别名。 以下示例显示如何在表上定义窗口聚合。

```
val table = input
    .window([w: GroupWindow] as 'w) // define window with alias w
    .groupBy('w) // group the table by window w
    .select('b.sum) // aggregate
```

在流式传输环境中，如果窗口聚合除对窗口进行分组以外，还对一个或多个属性进行分组，则它们只能并行计算，即 `groupBy (...)` 子句引用窗口别名和至少一个其他属性。 仅引用窗口别名的 `groupBy (...)` 子句（例如上例中的子句）只能由单个非并行任务求值。 以下示例显示如何使用其他分组属性定义窗口聚合。

```
val table = input
    .window([w: GroupWindow] as 'w) // define window with alias w
    .groupBy('w, 'a) // group the table by attribute a and window w
    .select('a, 'b.sum) // aggregate
```

可以在 `select` 语句中将窗口属性（例如时间窗口的开始，结束或行时间戳）添加为窗口别名的属性，分别为 `w.start`，`w.end` 和 `w.rowtime`。 窗口开始和行时间戳是包含窗口的上下边界。 相反，窗口结束时间戳是唯一的窗口上边界。 例如，从下午 2 点开始的 30 分钟

滚动窗口将以 14: 00: 00.000 作为开始时间戳，以 14: 29: 59.999 作为行时间时间戳，以 14: 30: 00.000 作为结束时间戳。

```
val table = input
    .window([w: GroupWindow] as 'w) // define window with alias w
    .groupBy('w, 'a) // group the table by attribute a and window w
    .select('a, 'w.start, 'w.end, 'w.rowtime, 'b.count) // aggregate
and add window start, end, and rowtime timestamps
```


Window 参数定义行如何映射到窗口。窗口不是用户可以实现的接口。相反，Table API 提供了一组具有特定语义的预定义 Window 类，这些类被转换为基础的 DataStream 或 DataSet 操作。支持的窗口定义在下面列出。

Tumble (Tumbling Windows)

滚动窗口将行分配给固定长度的非重叠连续窗口。例如，5 分钟的滚动窗口以 5 分钟为间隔对行进行分组。可以在事件时间，处理时间或行数上定义滚动窗口。

滚动窗口是使用 Tumble 类定义的，如下所示：


Method	Description
over	将窗口的长度定义为时间或行计数间隔。
on	用于分组（时间间隔）或排序（行计数）的时间属性。对于批查询，它可以是任何 Long 或 Timestamp 属性。对于流查询，它必须是声明的 <u>事件时间或处理时间</u> 时间属性。
as	为窗口分配别名。别名用于引用以下 groupBy（）子句中的窗口，并可以选择在 select（）子句中选择窗口属性，例如窗口开始，结束或行时间时间戳。



```
// Tumbling Event-time Window
.window(Tumble over 10.minutes on 'rowtime as 'w)

// Tumbling Processing-time Window (assuming a processing-time
attribute "proctime")
.window(Tumble over 10.minutes on 'proctime as 'w)

// Tumbling Row-count Window (assuming a processing-time attribute
"proctime")
.window(Tumble over 10.rows on 'proctime as 'w)
```



Slide (Sliding Windows)

滑动窗口的大小固定，并以指定的滑动间隔滑动。如果滑动间隔小于窗口大小，则滑动窗口重叠。因此，可以将行分配给多个窗口。例如，一个 15 分钟大小的滑动窗口和 5 分钟的滑

动间隔将每行分配给 3 个 15 分钟大小的不同窗口，它们以 5 分钟的间隔进行评估。 可以在事件时间，处理时间或行数上定义滑动窗口。

滑动窗口是通过使用 `Slide` 类定义的，如下所示：

Method	Description
over	将窗口的长度定义为时间或行计数间隔。
every	将幻灯片间隔定义为时间间隔或行计数间隔。滑动间隔必须与尺寸间隔具有相同的类型。
on	用于分组（时间间隔）或排序（行计数）的时间属性。 对于批查询，它可以是任何 <code>Long</code> 或 <code>Timestamp</code> 属性。对于流查询，它必须是声明的事件时间或处理时间时间属性。
as	为窗口分配别名。 别名用于引用以下 <code>groupBy ()</code> 子句中的窗口，并可以选择在 <code>select ()</code> 子句中选择窗口属性，例如窗口开始，结束或行时间戳。

```
// Sliding Event-time Window
.window(Slide over 10.minutes every 5.minutes on 'rowtime as 'w)

// Sliding Processing-time window (assuming a processing-time
attribute "proctime")
.window(Slide over 10.minutes every 5.minutes on 'proctime as 'w)

// Sliding Row-count window (assuming a processing-time attribute
"proctime")
.window(Slide over 10.rows every 5.rows on 'proctime as 'w)
```

Session (Session Windows)

会话窗口没有固定的大小，但其边界由不活动的时间间隔定义，即，如果在定义的间隔时间段内未出现任何事件，则会话窗口关闭。 例如，间隔 30 分钟的会话窗口在 30 分钟不活动后观察到一行时开始（否则该行将被添加到现有窗口），如果在 30 分钟内未添加任何行，则关闭该窗口。 会话窗口可以在事件时间或处理时间工作。

通过使用 `Session` 类定义会话窗口，如下所示：

Method	Description
withGap	将两个窗口之间的间隔定义为时间间隔。
on	用于分组（时间间隔）或排序（行计数）的时间属性。 对于批查询，它可以是任何 <code>Long</code> 或 <code>Timestamp</code> 属性。对于流查询，它必须是声明的事件时间或处理时间时间

Method	Description
	属性。
<code>as</code>	为窗口分配别名。别名用于引用以下 <code>groupBy()</code> 子句中的窗口，并可以选择在 <code>select()</code> 子句中选择窗口属性，例如窗口开始，结束或行时间戳。

```
// Session Event-time Window
.window(Session withGap 10.minutes on 'rowtime as 'w)

// Session Processing-time Window (assuming a processing-time
attribute "proctime")
.window(Session withGap 10.minutes on 'proctime as 'w)
```

Over Windows

窗口聚合是标准 SQL (`OVER` 子句) 已知的，并在查询的 `SELECT` 子句中定义。与在 `GROUP BY` 子句中指定的组窗口不同，在窗口上方不会折叠行。取而代之的是，在窗口聚合中，为每个输入行在其相邻行的范围内计算一个聚合。

使用 `window(w: OverWindow *)` 子句（在 Python API 中使用 `over_window(* OverWindow)`）定义窗口，并在 `select()` 方法中通过别名引用。以下示例显示了如何在表上定义窗口聚合。

```
val table = input
  .window([w: OverWindow] as 'w) // define over window
with alias w
  .select('a, 'b.sum over 'w, 'c.min over 'w) // aggregate over the
over window w
```

`OverWindow` 定义了计算聚合的行范围。 `OverWindow` 不是用户可以实现的接口。相反，`Table API` 提供了 `Over` 类来配置 `over` 窗口的属性。可以在事件时间或处理时间以及指定为时间间隔或行计数的范围上定义窗口上方。受支持的 `over` 窗口定义作为 `Over`（和其他类）上的方法公开，并在下面列出：

Method	Required	Description
<code>partitionBy</code>	Optional	<p>在一个或多个属性上定义输入的分区。每个分区都经过单独排序，并且聚合函数分别应用于每个分区。</p> <p>注意：在流环境中，仅当窗口包含 <code>partition by</code> 子句时，才可以并行计算整个窗口聚合。没有 <code>partitionBy(...)</code>，流将由单个非并行任务处理。</p>
<code>orderBy</code>	Required	定义每个分区内的行顺序，从而定义将聚合函数应用于行的顺序。

Method	Required	Description
		注意：对于流查询，它必须是声明的事件时间或处理时间时间属性。 当前，仅支持单个 sort 属性。
preceding	Optional	<p>定义窗口中包含的并在当前行之前的行的间隔。 该间隔可以指定为时间间隔或行计数间隔。</p> <p>用时间间隔的大小指定窗口上的边界，例如，时间间隔为 10 分钟，行计数间隔为 10 行。</p> <p>使用常数来指定窗口上的无边界，即对于时间间隔为 UNBOUNDED_RANGE 或对于行计数间隔为 UNBOUNDED_ROW。 Windows 上的无边界从分区的第一行开始。</p> <p>如果省略了前面的子句，则将 UNBOUNDED_RANGE 和 CURRENT_RANGE 用作窗口的默认前后</p>
following	Optional	<p>定义窗口中包含并紧随当前行的行的窗口间隔。 该间隔必须与前面的间隔（时间或行计数）以相同的单位指定。</p> <p>目前，不支持具有当前行之后的行的窗口。 相反，您可以指定两个常量之一：</p> <ul style="list-style-type: none"> CURRENT_ROW 将窗口的上限设置为当前行。 CURRENT_RANGE 将窗口的上限设置为当前行的排序键，即，与当前行具有相同排序键的所有行都包含在窗口中。 <p>如果省略以下子句，则将时间间隔窗口的上限定义为 CURRENT_RANGE，将行计数间隔窗口的上限定义为 CURRENT_ROW。</p>
as	Required	为上方窗口分配别名。 别名用于引用以下 select () 子句中的 over 窗口。

注意：当前，同一 **select** () 调用中的所有聚合函数必须在相同的窗口范围内计算。

Unbounded Over Windows



```
// Unbounded Event-time over window (assuming an event-time attribute "rowtime")
```

```

.window(Over partitionBy 'a orderBy 'rowtime preceding
UNBOUNDED_RANGE as 'w)

// Unbounded Processing-time over window (assuming a processing-time
attribute "proctime")
.window(Over partitionBy 'a orderBy 'proctime preceding
UNBOUNDED_RANGE as 'w)

// Unbounded Event-time Row-count over window (assuming an event-time
attribute "rowtime")
.window(Over partitionBy 'a orderBy 'rowtime preceding UNBOUNDED_ROW
as 'w)

// Unbounded Processing-time Row-count over window (assuming a
processing-time attribute "proctime")
.window(Over partitionBy 'a orderBy 'proctime preceding UNBOUNDED_ROW
as 'w)

```



Bounded Over Windows



```

// Bounded Event-time over window (assuming an event-time attribute
"rowtime")
.window(Over partitionBy 'a orderBy 'rowtime preceding 1.minutes as
'w)

// Bounded Processing-time over window (assuming a processing-time
attribute "proctime")
.window(Over partitionBy 'a orderBy 'proctime preceding 1.minutes as
'w)

// Bounded Event-time Row-count over window (assuming an event-time
attribute "rowtime")
.window(Over partitionBy 'a orderBy 'rowtime preceding 10.rows as 'w)

// Bounded Processing-time Row-count over window (assuming a
processing-time attribute "proctime")
.window(Over partitionBy 'a orderBy 'proctime preceding 10.rows as
'w)

```



Row-based Operations

基于行的操作生成具有多列的输出。

Operators	Description
Map Batch Streaming	<p>使用用户定义的标量函数或内置标量函数执行映射操作。 如果输出类型是复</p> <pre>class MyMapFunction extends ScalarFunction { def eval(a: String): Row = { Row.of(a, "pre-" + a) } override def getResultType(signature: Array[Class[_]]) = Types.ROW(Types.STRING, Types.STRING) } val func = new MyMapFunction() val table = input .map(func('c')).as('a, 'b)</pre>
FlatMap Batch Streaming	<p>使用表格功能执行 flatMap 操作.</p> <pre>class MyFlatMapFunction extends TableFunction[Row] { def eval(str: String): Unit = { if (str.contains("#")) { str.split("#").foreach({ s => val row = new Row(2) row.setField(0, s) row.setField(1, s.length) collect(row) }) } } } override def getResultType: TypeInformation[Row] = { Types.ROW(Types.STRING, Types.INT) } } val func = new MyFlatMapFunction val table = input .flatMap(func('c')).as('a, 'b)</pre>
Aggregate Batch Streaming Result Updating	<p>使用聚合函数执行聚合操作。 您必须使用 select 语句关闭“聚合”，并且 sel 如果输出类型是复合类型，则聚合的输出将被展平.</p> <pre>case class MyMinMaxAcc(var min: Int, var max: Int)</pre>

Operators	Description
	<pre>class MyMinMax extends AggregateFunction[Row, MyMinMaxAcc, Row] { def accumulate(acc: MyMinMaxAcc, value: Int): Unit = { if (value < acc.min) { acc.min = value } if (value > acc.max) { acc.max = value } } override def createAccumulator(): MyMinMaxAcc = MyMinMaxAcc() def resetAccumulator(acc: MyMinMaxAcc): Unit = { acc.min = 0 acc.max = 0 } override def getValue(acc: MyMinMaxAcc): Row = { Row.of(Integer.valueOf(acc.min), Integer.valueOf(acc.max)) } override def getResultType: TypeInformation[Row] = { new RowTypeInfo(Types.INT, Types.INT) } } val myAggFunc = new MyMinMax val table = input .groupBy('key) .aggregate(myAggFunc('a) as ('x, 'y)) .select('key, 'x, 'y)</pre>
Group Window Aggregate Batch Streaming	<p>在组窗口和可能的一个或多个分组键上对表进行分组和聚集。 您必须使用 select 语句，且 select 语句不支持“*”或聚合函数。</p> <pre>val myAggFunc = new MyMinMax val table = input .window(Tumble over 5.minutes on 'rowtime as 'w) // d .groupBy('key, 'w) // group by key and window</pre>

Operators	Description
	<pre>.aggregate(myAggFunc('a) as ('x, 'y)) .select('key, 'x, 'y, 'w.start, 'w.end) // access window aggregate results</pre>
FlatAggregate Streaming Result Updating	<p>类似于 GroupBy 聚合。 使用以下运行表聚合运算符将分组键上的行分组， ateFunction 的区别在于 TableAggregateFunction 可以为一个组返回 0 个 select 语句关闭“ flatAggregate”。 并且 select 语句不支持聚合函数。</p> <p>除了使用 emitValue 输出结果外，还可以使用 emitUpdateWithRetract 方 同，emitUpdateWithRetract 用于发出已更新的值。 此方法在撤消模式下 更新，我们就必须在发送新的更新记录之前撤回旧记录。 如果在表聚合函数 优先使用 emitUpdateWithRetract 方法，因为这两种方法比 emitValue 更 值。 有关详细信息，请参见表聚合函数。</p> <pre>import java.lang.{Integer => JInteger} import org.apache.flink.table.api.Types import org.apache.flink.table.functions.TableAggregateFunction /** * Accumulator for top2. */ class Top2Accum { var first: JInteger = _ var second: JInteger = _ } /** * The top2 user-defined table aggregate function. */ class Top2 extends TableAggregateFunction[JTuple2[JInteger, JInteger], Top2Accum] { override def createAccumulator(): Top2Accum = { val acc = new Top2Accum acc.first = Int.MinValue acc.second = Int.MinValue acc } def accumulate(acc: Top2Accum, v: Int) { if (v > acc.first) { acc.second = acc.first } } }</pre>

Operators	Description
	<pre>acc.first = v } else if (v > acc.second) { acc.second = v } }</pre> <pre>def merge(acc: Top2Accum, its: JIterable[Top2Accum]): Unit = { val iter = its.iterator() while (iter.hasNext) { val top2 = iter.next() accumulate(acc, top2.first) accumulate(acc, top2.second) } }</pre> <pre>def emitValue(acc: Top2Accum, out: Collector[JTuple2[Int, Int]]): Unit = { // emit the value and rank if (acc.first != Int.MinValue) { out.collect(JTuple2.of(acc.first, 1)) } if (acc.second != Int.MinValue) { out.collect(JTuple2.of(acc.second, 2)) } }</pre> <pre>val top2 = new Top2 val orders: Table = tableEnv.scan("Orders") val result = orders .groupBy('key) .flatAggregate(top2('a) as ('v, 'rank)) .select('key, 'v, 'rank)</pre> <p>Note: 对于流查询，根据聚合类型和不同的分组键的数量，计算查询结果所需的状态可能会非常大。因此，对于流查询，请提供具有有效保留间隔的查询配置，以防止出现过多的状态。 有关详细信息，请参阅 保留间隔。</p>
Group Window FlatAggregate Streaming	<p>在组窗口和可能的一个或多个分组键上对表进行分组和聚集。 您必须使用 <code>FlatAggregate</code> 函数。 并且 <code>select</code> 语句不支持聚合函数。</p> <pre>val top2 = new Top2 val orders: Table = tableEnv.scan("Orders")</pre>

Operators	Description
	<pre>val result = orders .window(Tumble over 5.minutes on 'rowtime as 'w) // d .groupBy('a, 'w) // group by key and window .flatAggregate(top2('b) as ('v, 'rank)) .select('a, w.start, 'w.end, 'w.rowtime, 'v, 'rank) // erties and aggregate results</pre>

Data Types

请参阅有关[数据类型](#)的专用页面。

通用类型和（嵌套的）复合类型（例如 POJO，元组，行，Scala case class）也可以是一行的字段。


可以使用值访问功能访问具有任意嵌套的复合类型的字段。

泛型类型被视为黑盒，可以通过用户定义的函数传递或处理。

Expression Syntax

上一节中的某些运算符期望一个或多个表达式。 可以使用嵌入式 Scala DSL 或字符串指定表达式。 请参考上面的示例以了解如何指定表达式。

这是用于表达式的 EBNF 语法：



```
expressionList = expression , { "," , expression } ;

expression = overConstant | alias ;

alias = logic | ( logic , "as" , fieldReference ) | ( logic , "as" ,
"(" , fieldReference , { "," , fieldReference } , ")" ) ;

logic = comparison , [ ( "&&" | "||" ) , comparison ] ;

comparison = term , [ ( "=" | "==" | "===" | "!=" | "!===" | ">" |
">=" | "<" | "<=" ) , term ] ;

term = product , [ ( "+" | "-" ) , product ] ;

product = unary , [ ( "*" | "/" | "%" ) , unary ] ;
```

```

unary = [ "!" | "-" | "+" ] , composite ;

composite = over | suffixed | nullLiteral | prefixed | atom ;

suffixed = interval | suffixAs | suffixCast | suffixIf |
suffixDistinct | suffixFunctionCall | timeIndicator ;

prefixed = prefixAs | prefixCast | prefixIf | prefixDistinct |
prefixFunctionCall ;

interval = timeInterval | rowInterval ;

timeInterval = composite , "." , ( "year" | "years" | "quarter" |
"quarters" | "month" | "months" | "week" | "weeks" | "day" | "days" |
"hour" | "hours" | "minute" | "minutes" | "second" | "seconds" |
"milli" | "millis" ) ;

rowInterval = composite , "." , "rows" ;

suffixCast = composite , ".cast(" , dataType , ")" ;

prefixCast = "cast(" , expression , dataType , ")" ;

dataType = "BYTE" | "SHORT" | "INT" | "LONG" | "FLOAT" | "DOUBLE" |
"BOOLEAN" | "STRING" | "DECIMAL" | "SQL_DATE" | "SQL_TIME" |
"SQL_TIMESTAMP" | "INTERVAL_MONTHS" | "INTERVAL_MILLIS" | ( "MAP" ,
 "(" , dataType , "," , dataType , ")" ) | ( "PRIMITIVE_ARRAY" , "(" ,
 dataType , ")" ) | ( "OBJECT_ARRAY" , "(" , dataType , ")" ) ;

suffixAs = composite , ".as(" , fieldReference , ")" ;

prefixAs = "as(" , expression , fieldReference , ")" ;

suffixIf = composite , "?((" , expression , "," , expression , ")" ;

prefixIf = "?((" , expression , "," , expression , "," , expression ,
 ")" ;

suffixDistinct = composite , "distinct.()" ;

prefixDistinct = functionIdentifier , ".distinct" , [ "(" ,
 [ expression , { "," , expression } ] , ")" ] ;

```

```

suffixFunctionCall = composite , "." , functionIdentifier , [ "(" ,
[ expression , { "," , expression } ] , ")" ] ;

prefixFunctionCall = functionIdentifier , [ "(" , [ expression ,
{ "," , expression } ] , ")" ] ;

atom = ( "(" , expression , ")" ) | literal | fieldReference ;

fieldReference = "*" | identifier ;

nullLiteral = "nullOf(" , dataType , ")" ;

timeIntervalUnit = "YEAR" | "YEAR_TO_MONTH" | "MONTH" | "QUARTER" |
"WEEK" | "DAY" | "DAY_TO_HOUR" | "DAY_TO_MINUTE" | "DAY_TO_SECOND" |
"HOUR" | "HOUR_TO_MINUTE" | "HOUR_TO_SECOND" | "MINUTE" |
"MINUTE_TO_SECOND" | "SECOND" ;

timePointUnit = "YEAR" | "MONTH" | "DAY" | "HOUR" | "MINUTE" |
"SECOND" | "QUARTER" | "WEEK" | "MILLISECOND" | "MICROSECOND" ;

over = composite , "over" , fieldReference ;

overConstant = "current_row" | "current_range" | "unbounded_row" |
"unbounded_row" ;

timeIndicator = fieldReference , "." , ( "proctime" | "rowtime" ) ;

```



Literals: 在这里，文字是有效的 Java 文字。字符串文字可以使用单引号或双引号指定。复制引号以进行转义（例如，'It's me.' or "I ""like"" dogs."）。

Null literals: 空文字必须附加一个类型。使用 `nullOf (type)`（例如 `nullOf (INT)`）创建空值。

Field references: `fieldReference` 指定数据中的一列（如果使用*，则指定所有列），而 `functionIdentifier` 指定受支持的标量函数。列名和函数名遵循 Java 标识符语法。

Function calls: 指定为字符串的表达式也可以使用前缀表示法而不是后缀表示法来调用运算符和函数。

Decimals: 如果需要使用精确的数值或大的十进制数，则 Table API 还支持 Java 的 `BigDecimal` 类型。在 Scala Table API 中，小数可以由 `BigDecimal`（“123456”）定义，而在 Java 中，可以通过附加“p”来精确定义例如 123456p

Time representation: 为了使用时间值，Table API 支持 Java SQL 的日期，时间和时间戳类型。在 Scala Table API 中，可以使用 `java.sql.Date.valueOf`（“2016-06-27”），`java.sql.Time.valueOf`（“10:10:42”）或 `java.sql` 定义文字。 `Timestamp.valueOf`（“2016-06-

27 10: 10: 42.123”)。Java 和 Scala 表 API 还支持调用“2016-06-27”.toDate(), “10:10:42”.toTime() 和“2016-06-27 10: 10: 42.123”.toTimestamp() 用于将字符串转换为时间类型。注意: 由于 Java 的时态 SQL 类型取决于时区, 因此请确保 Flink Client 和所有 TaskManager 使用相同的时区。

Temporal intervals: 时间间隔可以表示为月数 (Types.INTERVAL_MONTHS) 或毫秒数 (Types.INTERVAL_MILLIS)。可以添加或减去相同类型的间隔 (例如 1.小时+ 10 分钟)。可以将毫秒间隔添加到时间点 (例如“2016-08-10”.toDate() + 5.days)。

Scala expressions: Scala 表达式使用隐式转换。因此, 请确保将通配符导入 org.apache.flink.table.api.scala._ 添加到程序中。如果文字不被视为表达式, 请使用.toExpr (如 3.toExpr) 强制转换文字。

【翻译】Flink Table Api & SQL — SQL

本文翻译自官网: SQL <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/sql.html>

Flink Table Api & SQL 翻译目录

这是 Flink 支持的 数据定义语言 (DDL) 和数据操纵语言的完整列表。

- 查询
 - 指定查询
 - 支持语法
 - 操作
- DDL
 - 指定 DDL
 - 创建表
 - 删除表
- Data Types
- Reserved 保留关键字

查询

SQL 查询使用 TableEnvironment 的 sqlQuery() 方法指定。这个方法返回一个表作为 SQL 查询的结果。这个表可以在后续的 SQL 和 Table API 中查询, 可以转换为 DataSet 和 DataStream, 或写到 TableSink 中。SQL 和 Table API 查询可以无缝混合, 可以进行整体优化并将其转换为单个程序。


为了在 SQL 查询中使用一个表, 它必须在 TableEnvironment 中注册。表可以通过 TableSource, Table, CREATE TABLE statement, DataStream, 或 DataSet 注册。或者, 用户还可以在 TableEnvironment 中注册外部目录以指定数据源的位置。

为方便起见，`Table.toString()` 自动在其 `TableEnvironment` 中以唯一名称注册该表并返回该名称。因此，可以将 `Table` 对象直接内联到 SQL 查询中（通过字符串连接），如下面的示例所示。

注意：Flink 的 SQL 支持尚未完成。包含不受支持的 SQL 功能的查询会导致 `TableException`。以下各节列出了批处理表和流表上 SQL 的受支持功能。

指定查询

以下示例显示如何在已注册和内联表上指定 SQL 查询。




```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tableEnv = StreamTableEnvironment.create(env)

// read a DataStream from an external source
val ds: DataStream[(Long, String, Integer)] = env.addSource(...)

// SQL query with an inlined (unregistered) table
val table = ds.toTable(tableEnv, 'user, 'product, 'amount)
val result = tableEnv.sqlQuery(
    s"SELECT SUM(amount) FROM $table WHERE product LIKE '%Rubber%'" )

// SQL query with a registered table
// register the DataStream under the name "Orders"
tableEnv.registerDataStream("Orders", ds, 'user, 'product, 'amount)
// run a SQL query on the Table and retrieve the result as a new
Table
val result2 = tableEnv.sqlQuery(
    "SELECT product, amount FROM Orders WHERE product LIKE '%Rubber%'" )

// SQL update with a registered table
// create and register a TableSink
val csvSink: CsvTableSink = new CsvTableSink("/path/to/file", ...)
val fieldNames: Array[String] = Array("product", "amount")
val fieldTypes: Array[TypeInformation[_]] = Array(Types.STRING,
Types.INT)
tableEnv.registerTableSink("RubberOrders", fieldNames, fieldTypes,
csvSink)
// run a SQL update query on the Table and emit the result to the
TableSink
tableEnv.sqlUpdate(
    "INSERT INTO RubberOrders SELECT product, amount FROM Orders WHERE
product LIKE '%Rubber%'" )
```



Supported Syntax

Flink 使用支持标准 ANSI SQL 的 Apache Calcite 解析 SQL。Flink 不支持 DDL 语句。

以下 BNF 语法描述了批处理和流查询中支持的 SQL 功能的超集。“操作”部分显示了受支持功能的示例，并指示仅批处理或流查询支持哪些功能。



```
insert:
    INSERT INTO tableReference
    query

query:
    values
    | {
        select
        | selectWithoutFrom
        | query UNION [ ALL ] query
        | query EXCEPT query
        | query INTERSECT query
    }
    [ ORDER BY orderItem [, orderItem ]* ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start { ROW | ROWS } ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]

orderItem:
    expression [ ASC | DESC ]

select:
    SELECT [ ALL | DISTINCT ]
    { * | projectItem [, projectItem ]* }
    FROM tableExpression
    [ WHERE booleanExpression ]
    [ GROUP BY { groupItem [, groupItem ]* } ]
    [ HAVING booleanExpression ]
    [ WINDOW windowName AS windowSpec [, windowName AS windowSpec ]* ]

selectWithoutFrom:
    SELECT [ ALL | DISTINCT ]
    { * | projectItem [, projectItem ]* }

projectItem:
    expression [ [ AS ] columnAlias ]
    | tableAlias . *

tableExpression:
```

```

    tableReference [, tableReference ]*
    | tableExpression [ NATURAL ] [ LEFT | RIGHT | FULL ] JOIN
tableExpression [ joinCondition ]

joinCondition:
    ON booleanExpression
    | USING '(' column [, column ]* ')'

tableReference:
    tablePrimary
    [ matchRecognize ]
    [ [ AS ] alias [ '(' columnAlias [, columnAlias ]* ')' ] ]

tablePrimary:
    [ TABLE ] [ [ catalogName . ] schemaName . ] tableName
    | LATERAL TABLE '(' functionName '(' expression [, expression ]*
    ')' ')'
    | UNNEST '(' expression ')'

values:
    VALUES expression [, expression ]*

groupItem:
    expression
    | '(' ')'
    | '(' expression [, expression ]* ')'
    | CUBE '(' expression [, expression ]* ')'
    | ROLLUP '(' expression [, expression ]* ')'
    | GROUPING SETS '(' groupItem [, groupItem ]* ')'

windowRef:
    windowName
    | windowSpec

windowSpec:
    [ windowName ]
    '('
    [ ORDER BY orderItem [, orderItem ]* ]
    [ PARTITION BY expression [, expression ]* ]
    [
        RANGE numericOrIntervalExpression {PRECEDING}
        | ROWS numericExpression {PRECEDING}
    ]
    ')'

```

```

matchRecognize:
    MATCH_RECOGNIZE '('
    [ PARTITION BY expression [, expression ]* ]
    [ ORDER BY orderItem [, orderItem ]* ]
    [ MEASURES measureColumn [, measureColumn ]* ]
    [ ONE ROW PER MATCH ]
    [ AFTER MATCH
        ( SKIP TO NEXT ROW
        | SKIP PAST LAST ROW
        | SKIP TO FIRST variable
        | SKIP TO LAST variable
        | SKIP TO variable )
    ]
    PATTERN '(' pattern ')'
    [ WITHIN intervalLiteral ]
    DEFINE variable AS condition [, variable AS condition ]*
    ')'

measureColumn:
    expression AS alias

pattern:
    patternTerm [ '|' patternTerm ]*

patternTerm:
    patternFactor [ patternFactor ]*

patternFactor:
    variable [ patternQuantifier ]

patternQuantifier:
    '*'
    | '*?'
    | '+'
    | '+?'
    | '?'
    | '??'
    | '{' { [ minRepeat ], [ maxRepeat ] } '}' ['?']
    | '{' repeat '}'

```



Flink SQL 对类似于 Java 的标识符（表，属性，函数名称）使用词法策略：

- 不管是否引用标识符，都保留标识符的大小写。

- 之后，标识符区分大小写。
- 与 Java 不同，反引号允许标识符包含非字母数字字符（例如"SELECT a AS `my field` FROM t"）。

字符串文字必须用单引号引起来（例如 SELECT 'Hello World'）。复制单引号以进行转义（例如 SELECT 'It''s me.'）。字符串文字中支持 **Unicode** 字符。如果需要明确的 **unicode** 代码点，请使用以下语法：

- 使用反斜杠（\）作为转义字符（默认）：SELECT U&'\263A'
- 使用自定义转义字符：SELECT U&'#263A' UESCAPE '#'

Operations

Show and Use

Operation	Description
Show Batch Streaming	<p>Show all catalogs</p> <p><code>SHOW CATALOGS;</code></p> <p>Show all databases in the current catalog</p> <p><code>SHOW DATABASES;</code></p> <p>Show all tables in the current database in the current catalog</p> <p><code>SHOW TABLES;</code></p>
Use Batch Streaming	<p>Set current catalog for the session</p> <p><code>USE CATALOG mycatalog;</code></p> <p>Set current database of the current catalog for the session</p> <p><code>USE mydatabase;</code></p>

Scan, Projection, and Filter

Operation	Description
Scan / Select / As Batch Streaming	<p><code>SELECT * FROM Orders</code></p> <p><code>SELECT a, c AS d FROM Orders</code></p>
Where / Filter Batch Streaming	<p><code>SELECT * FROM Orders WHERE b = 'red'</code></p>

Operation	Description
	<code>SELECT * FROM Orders WHERE a % 2 = 0</code>
User-defined Scalar Functions (Scalar UDF) Batch Streaming	UDFs 必须在 <code>TableEnvironment</code> 中注册了。See the UDF documentation 有关如何指定和注册标量 UDF 的详细信息。 <code>SELECT PRETTY_PRINT(user) FROM Orders</code>

Aggregations

Operation	Description
GroupBy Aggregation Batch Streaming Result Updating	Note: 流表上的 GroupBy 产生更新结果。See the Dynamic Tables Streaming Concepts page for details. <code>SELECT a, SUM(b) as d FROM Orders GROUP BY a</code>
GroupBy Window Aggregation Batch Streaming	使用分组窗口可为每个组计算一个结果行。有关更多详细信息，请参见 Group Windows 部分。 <code>SELECT user, SUM(amount) FROM Orders GROUP BY TUMBLE(rowtime, INTERVAL '1' DAY), user</code>
Over Window aggregation Streaming	Note: 必须在同一窗口（即相同的分区，排序和范围）上定义所有聚合。当前，仅支持 PRECEDING （无边界和有界）到 CURRENT ROW 范围的窗口。 目前尚不支持带有 FOLLOWING 的范围。必须在单个时间属性上指定 ORDER BY <code>SELECT COUNT(amount) OVER (PARTITION BY user ORDER BY proctime ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) FROM Orders</code> <code>SELECT COUNT(amount) OVER w, SUM(amount) OVER w FROM Orders</code>

Operation	Description
	<pre>WINDOW w AS (PARTITION BY user ORDER BY proctime ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)</pre>
Distinct Batch Streaming Result Updating	<pre>SELECT DISTINCT users FROM Orders</pre> <p>Note: 对于流查询，根据不同字段的数量，计算查询结果所需的状态可能会无限增长。</p> <p>请提供具有有效保留间隔的查询配置，以防止出现过多的状态。 有关详细信息，请参见查询配置。。</p>
Grouping sets, Rollup, Cube Batch	<pre>SELECT SUM(amount) FROM Orders GROUP BY GROUPING SETS ((user), (product))</pre>
Having Batch Streaming	<pre>SELECT SUM(amount) FROM Orders GROUP BY users HAVING SUM(amount) > 50</pre>
User-defined Aggregate Functions (UDAGG) Batch Streaming	<p>UDAGG 必须在 TableEnvironment 中注册。 有关如何指定和注册 UDAGG 的详细信息，请参见 UDF 文档。</p> <pre>SELECT MyAggregate(amount) FROM Orders GROUP BY users</pre>

Joins

Operation	Description
Inner Equi-join Batch Streaming	<p>当前，仅支持等值连接，即具有至少一个具有相等谓词的联合条件的连接。 不支持任意交叉或 theta 连接（自连接）。</p> <p>Note: 连接顺序未优化。 表按照在 FROM 子句中指定的顺序进行连接。 确保以不产生交叉联接（笛卡尔乘积）的顺序指定表，该顺序不被支持并会导致查询失败。</p> <pre>SELECT *</pre>

Operation	Description
	<p><code>FROM Orders INNER JOIN Product ON Orders.product Id = Product.id</code></p> <p>Note: 对于流查询，根据不同输入行的数量，计算查询结果所需的状态可能会无限增长。 请提供具有有效保留间隔的查询配置，以防止出现过多的状态。 有关详细信息，请参见查询配置。</p>
Outer Equi-join Batch Streaming Result Updating	<p>当前，仅支持等值连接，即具有至少一个具有相等谓词的联合条件的连接。 不支持任意交叉或 theta 连接（自连接）。</p> <p>Note: 连接顺序未优化。 表按照在 FROM 子句中指定的顺序进行连接。 确保以不产生交叉联接（笛卡尔乘积）的顺序指定表，该顺序不被支持并会导致查询失败。</p> <p><code>SELECT *</code> <code>FROM Orders LEFT JOIN Product ON Orders.productId = Product.id</code></p> <p><code>SELECT *</code> <code>FROM Orders RIGHT JOIN Product ON Orders.productId = Product.id</code></p> <p><code>SELECT *</code> <code>FROM Orders FULL OUTER JOIN Product ON Orders.productId = Product.id</code></p> <p>Note: 对于流查询，根据不同输入行的数量，计算查询结果所需的状态可能会无限增长。 请提供具有有效保留间隔的查询配置，以防止出现过多的状态。 有关详细信息，请参见查询配置。</p>
Time-windowed Join Batch Streaming	<p>Note: 时间窗口连接是常规连接的子集，可以以流方式处理。</p> <p>时间窗连接需要至少一个等联接谓词和在两侧限制时间的连接条件。两个输入表可以通过两个适当的范围谓词（<, <=, >=, >），BETWEEN 谓词或比较相同类型的时间属性（即处理时间或事件时间）的单个相等谓词来定义这样的条件。</p> <p>例如，以下谓词是有效的窗口连接条件：</p> <ul style="list-style-type: none"> <code>ltime = rtime</code> <code>ltime >= rtime AND ltime < rtime + INTERVAL '10' MINUTE</code>

Operation	Description
	<ul style="list-style-type: none"> ltime BETWEEN rtime - INTERVAL '10' SECOND AND rtime + INTERVAL '5' SECOND <pre>SELECT * FROM Orders o, Shipments s WHERE o.id = s.orderId AND o.ordertime BETWEEN s.shiptime - INTERVAL '4' HOUR AND s.shiptime</pre> <p>如果订单在收到订单后四个小时内发货，则上面的示例会将所有订单与其相应的发货合并在一起。</p>
Expanding arrays into a relation Batch Streaming	<p>目前尚不支持使用 ORDINALITY 取消嵌套。</p> <pre>SELECT users, tag FROM Orders CROSS JOIN UNNEST(tags) AS t (tag)</pre>
Join with Table Function (UDTF) Batch Streaming	<p>用表函数的结果连接表。 左（外）表的每一行都与表函数的相应调用产生的所有行连接在一起。</p> <p>必须先注册用户定义的表函数（UDTF）。 有关如何指定和注册 UDTF 的详细信息，请参见 UDF 文档。</p> <p>Inner Join</p> <p>如果左表（外部）的表函数调用返回空结果，则会删除该表的左行。</p> <pre>SELECT users, tag FROM Orders, LATERAL TABLE(unnest_udtf(tags)) t AS tag</pre> <p>Left Outer Join</p> <p>如果表函数调用返回空结果，则保留对应的外部行，并用空值填充结果。</p> <pre>SELECT users, tag FROM Orders LEFT JOIN LATERAL TABLE(unnest_udtf (tags)) t AS tag ON TRUE</pre> <p>Note: 当前，仅支持将文字 TRUE 作为针对横向表的左外部连接的谓词。</p>
Join with Temporal Table Function	<p>时态表是跟踪随时间变化的表。</p>

Operation	Description
Streaming	<p>时态表功能提供对特定时间点时态表状态的访问。使用时态表函数联接表的语法与使用表函数联接的语法相同。</p> <p>Note: 当前仅支持带有时态表的内部联接。</p> <p>假设 Rates 是一个时态表函数，则联接可以用 SQL 表示如下：</p> <pre>SELECT o_amount, r_rate FROM Orders, LATERAL TABLE (Rates(o_proctime)) WHERE r_currency = o_currency</pre> <p>有关更多信息，请检查更详细的时态表概念描述。</p>
Join with Temporal Table Batch Streaming	<p>时态表是跟踪随时间变化的表。临时表提供对特定时间点的时态表版本的访问。</p> <p>仅支持带有处理时间时态表的内部联接和左联接。</p> <p>下面的示例假定 LatestRates 是一个以最新速率物化的时态表。</p> <pre>SELECT o.amout, o.currency, r.rate, o.amount * r.rate FROM Orders AS o JOIN LatestRates FOR SYSTEM_TIME AS OF o.proctime AS r ON r.currency = o.currency</pre> <p>有关更多信息，请检查更详细的时态表概念描述。</p> <p>只有 Blink planner 支持。</p>

Set Operations

Operation	Description
Union Batch	<pre>SELECT * FROM ((SELECT user FROM Orders WHERE a % 2 = 0) UNION</pre>

Operation	Description
	<pre>(SELECT user FROM Orders WHERE b = 0))</pre>
UnionAll Batch Streaming	<pre>SELECT * FROM ((SELECT user FROM Orders WHERE a % 2 = 0) UNION ALL (SELECT user FROM Orders WHERE b = 0))</pre>
Intersect / Except Batch	<pre>SELECT * FROM ((SELECT user FROM Orders WHERE a % 2 = 0) INTERSECT (SELECT user FROM Orders WHERE b = 0)) SELECT * FROM ((SELECT user FROM Orders WHERE a % 2 = 0) EXCEPT (SELECT user FROM Orders WHERE b = 0))</pre>
In Batch Streaming	<p>如果给定表子查询中存在表达式，则返回 true。子查询表必须由一列组成。此列的数据类型必须与表达式相同。</p> <pre>SELECT user, amount FROM Orders WHERE product IN (SELECT product FROM NewProducts)</pre> <p>Note: 对于流查询，该操作将被重写为 join and group 操作。根据不同输入行的数量，计算查询结果所需的状态可能会无限增长。</p> <p>请提供具有有效保留间隔的查询配置，以防止出现过多的状态。有关详细信息，请参见查询配置。。</p>
Exists Batch Streaming	<p>如果子查询返回至少一行，则返回 true。仅在可以在联接和组操作中重写操作时才受支持。</p> <pre>SELECT user, amount FROM Orders</pre>

Operation	Description
	<pre>WHERE product EXISTS (SELECT product FROM NewProducts)</pre> <p>Note: 对于流查询，该操作将被重写为 join and group 操作。 根据不同输入行的数量，计算查询结果所需的状态可能会无限增长。</p> <p>请提供具有有效保留间隔的查询配置，以防止出现过多的状态。 有关详细信息，请参见查询配置。。</p>

OrderBy & Limit

Operation	Description
Order By Batch Streaming	<p>Note: 流查询的结果必须主要按升序时间属性排序。 支持其他排序属性。</p> <pre>SELECT * FROM Orders ORDER BY orderTime</pre>
Limit Batch	<p>Note: LIMIT 子句需要 ORDER BY 子句。</p> <pre>SELECT * FROM Orders ORDER BY orderTime LIMIT 3</pre>

Top-N

注意：仅 Blink planner 仅支持 Top-N。

Top-N 查询要求按列排序的 N 个最小值或最大值。 最小和最大值集都被认为是 Top-N 查询。 在需要只显示批处理/流表中的 N 个最底层记录或 N 个最顶层记录的情况下，Top-N 查询很有用。 此结果集可用于进一步分析。

Flink 使用 OVER 窗口子句和过滤条件的组合来表示 Top-N 查询。 借助 OVER window PARTITION BY 子句的强大功能，Flink 还支持每组 Top-N。 例如，每个类别中实时销量最高的前五种产品。 批处理表和流表上的 SQL 支持 Top-N 查询。

下面显示了 TOP-N 语句的语法：

```
SELECT [column_list]
FROM (
    SELECT [column_list],
        ROW_NUMBER() OVER ([PARTITION BY col1[, col2...]])
```

```
ORDER BY col1 [asc|desc][, col2 [asc|desc]...]) AS rownum
FROM table_name)
WHERE rownum <= N [AND conditions]
```



参数含义：

- `ROW_NUMBER()`：根据分区中各行的顺序，为每一行分配一个唯一的顺序号（从 1 开始）。目前，我们仅支持 `ROW_NUMBER` 作为窗口函数。将来，我们将支持 `RANK()` 和 `DENSE_RANK()`。
- `PARTITION BY col1[, col2...]`：指定分区列。每个分区都有一个 Top-N 结果。
- `ORDER BY col1 [asc|desc][, col2 [asc|desc]...]`：指定排序列。不同列上的排序方向可以不同。
- `WHERE rownum <= N`：Flink 将 `rownum <= N` 识别为该查询是 Top-N 查询。N 代表将保留 N 个最小或最大记录。
- `[AND conditions]`：可以在 `where` 子句中添加其他条件，但是其他条件只能与 `AND` 结合使用 `rownum <= N`。

流模式下的注意：TopN 查询会更新结果。Flink SQL 将根据顺序键对输入数据流进行排序，因此，如果前 N 条记录已更改，则更改后的记录将作为撤消/更新记录发送到下游。建议使用支持更新的存储作为 Top-N 查询的接收器。此外，如果需要将前 N 条记录存储在外部存储中，则结果表应具有与前 N 条查询相同的唯一键。

Top-N 查询的唯一键是分区列和 `rownum` 列的组合。Top-N 查询还可以导出上游的唯一键。以下面的工作为例，假设 `product_id` 是 `ShopSales` 的唯一键，那么 Top-N 查询的唯一键是 `[category, rownum]` 和 `[product_id]`。

以下示例显示如何在流表上使用 Top-N 指定 SQL 查询。这是我们上面提到的“每个类别中实时销量最高的前五种产品”的示例。



```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// read a DataStream from an external source
val ds: DataStream[(String, String, String, Long)] =
env.addSource(...)

// register the DataStream under the name "ShopSales"
tableEnv.registerDataStream("ShopSales", ds, 'product_id, 'category,
'product_name, 'sales)

// select top-5 products per category which have the maximum sales.
val result1 = tableEnv.sqlQuery(
  """
  |SELECT *
  |FROM (
```

```

|    SELECT *,
|        ROW_NUMBER() OVER (PARTITION BY category ORDER BY sales
DESC) as row_num
|    FROM ShopSales)
|WHERE row_num <= 5
""".stripMargin)

```



没有排名输出优化

如上所述，`rownum` 字段将作为唯一键的一个字段写入结果表，这可能导致许多记录被写入结果表。例如，当更新排名 9 的记录（例如 `product-1001`）并将其排名升级到 1 时，排名 1 到 9 的所有记录将作为更新消息输出到结果表。如果结果表接收到太多数据，它将成为 SQL 作业的瓶颈。

优化方法是在 **Top-N** 查询的外部 **SELECT** 子句中省略 `rownum` 字段。这是合理的，因为前 **N** 条记录的数量通常不大，因此消费者可以自己记录进行快速排序。如果没有 `rownum` 字段，则在上面的示例中，仅需要将已更改的记录（`product-1001`）发送到下游，这可以减少结果表的大量 IO。

下面的示例显示如何以这种方式优化上面的 **Top-N** 示例：

```

val env = StreamExecutionEnvironment.getExecutionEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// read a DataStream from an external source
val ds: DataStream[(String, String, String, Long)] =
env.addSource(...)
// register the DataStream under the name "ShopSales"
tableEnv.registerDataStream("ShopSales", ds, 'product_id, 'category,
'product_name, 'sales)

// select top-5 products per category which have the maximum sales.
val result1 = tableEnv.sqlQuery(
    """
        |SELECT product_id, category, product_name, sales -- omit
row_num field in the output
        |FROM (
        |    SELECT *,
        |        ROW_NUMBER() OVER (PARTITION BY category ORDER BY sales
DESC) as row_num
        |    FROM ShopSales)
        |WHERE row_num <= 5
    """.stripMargin)

```



流传输模式中的注意事项：为了将以上查询输出到外部存储并获得正确的结果，外部存储必须具有与 **Top-N** 查询相同的唯一键。 在上面的示例查询中，如果 **product_id** 是查询的唯一键，则外部表也应将 **product_id** 作为唯一键。

重复数据删除

注意：重复数据删除仅在 **Blink planner** 中受支持。

重复数据删除是指删除在一组列上重复的行，仅保留第一个或最后一个。 在某些情况下，上游 **ETL** 作业不是一次精确的端到端，这可能导致在故障转移的情况下，接收器中有重复的记录。但是，重复的记录将影响下游分析作业的正确性（例如 **SUM**，**COUNT**）。 因此，在进一步分析之前需要进行重复数据删除。

Flink 使用 **ROW_NUMBER()** 删除重复项，就像 **Top-N** 查询一样。 从理论上讲，重复数据删除是 **Top-N** 的一种特殊情况，其中 **N** 为 **1**，并按处理时间或事件时间排序。

下面显示了重复数据删除语句的语法：

```
SELECT [column_list]
FROM (
    SELECT [column_list],
        ROW_NUMBER() OVER ([PARTITION BY col1[, col2...]]
            ORDER BY time_attr [asc|desc]) AS rownum
    FROM table_name)
WHERE rownum = 1
```

参数含义：

- **ROW_NUMBER()**：从每一行开始，为每一行分配一个唯一的顺序号。
- **PARTITION BY col1[, col2...]**：指定分区列，即重复数据删除键。
- **ORDER BY time_attr [asc|desc]**：指定排序列，它必须是时间属性。 当前仅支持 **proctime** 属性。 将来将支持行时间属性。 通过 **ASC** 排序意味着保留第一行，通过 **DESC** 排序意味着保留最后一行。
- **WHERE rownum = 1**：Flink 要求 **rownum = 1** 才能识别此查询是重复数据删除。

以下示例说明如何在流表上指定带有重复数据删除功能的 **SQL** 查询。

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val tableEnv = TableEnvironment.getTableEnvironment(env)

// read a DataStream from an external source
val ds: DataStream[(String, String, String, Int)] =
env.addSource(...)

// register the DataStream under the name "Orders"
tableEnv.registerDataStream("Orders", ds, 'order_id, 'user, 'product,
'number, 'proctime.proctime)
```

```
// remove duplicate rows on order_id and keep the first occurrence
row,
// because there shouldn't be two orders with the same order_id.
val result1 = tableEnv.sqlQuery(
    """
    |SELECT order_id, user, product, number
    |FROM (
    |    SELECT *,
    |        ROW_NUMBER() OVER (PARTITION BY order_id ORDER BY
proctime DESC) as row_num
    |    FROM Orders)
    |WHERE row_num = 1
    """.stripMargin)
```



Insert

Operation	Description
Insert Into Batch Streaming	<p>输出表必须在 TableEnvironment 中注册（请参阅注册 TableSink）。此外，已注册表的结构必须与查询的结构匹配。</p> <pre>INSERT INTO OutputTable SELECT users, tag FROM Orders</pre>

Group Windows

分组窗口在 SQL 查询的 **GROUP BY** 子句中定义。就像带有常规 **GROUP BY** 子句的查询一样，带有 **GROUP BY** 子句（包括分组窗口函数）的查询每个组只计算一个结果行。批处理表和流表上的 SQL 支持以下分组窗口功能。

Group Window Function	Description
TUMBLE(time_attribute, interval)	<p>定义滚动时间窗口。滚动时间窗口将行分配给具有固定持续时间（间隔）的非重叠连续窗口。例如，5 分钟的滚动窗口以 5 分钟为间隔对行进行分组。</p> <p>可以在事件时间（流+批处理）或处理时间（流）上定义滚动窗口。</p>

Group Window Function	Description
<p>HOP(time_attr, interval, interval)</p>	<p>定义一个跳跃时间窗口（在 Table API 中称为滑动窗口）。跳跃时间窗口具有固定的持续时间（第二个间隔参数），并按指定的跳跃间隔（第一个间隔参数）跳跃。</p> <p>如果跳跃间隔小于窗口大小，则跳跃窗口重叠。因此，可以将行分配给多个窗口。例如，一个 15 分钟大小和 5 分钟跳跃间隔的跳窗将每行分配给 3 个 15 分钟大小的不同窗口，它们以 5 分钟的间隔进行评估。可以在事件时间（流+批处理）或处理时间（流）上定义跳跃窗口。</p>
<p>SESSION(time_attr, interval)</p>	<p>定义会话时间窗口。会话时间窗口没有固定的持续时间，但其边界由不活动的时间间隔定义，即，如果在定义的间隔时间段内未出现任何事件，则关闭会话窗口。</p> <p>例如，间隔 30 分钟的会话窗口在 30 分钟不活动后观察到一行时开始（否则该行将被添加到现有窗口），如果在 30 分钟内未添加任何行，则关闭该窗口。</p> <p>会话窗口可以在事件时间（流+批处理）或处理时间（流）上工作。</p>

时间属性

对于流表上的 SQL 查询，分组窗口函数的 time_attr 参数必须引用一个有效的时间属性，该属性指定行的处理时间或事件时间。请参阅时间属性文档以了解如何定义时间属性。

对于批处理表上的 SQL，分组窗口函数的 time_attr 参数必须是 TIMESTAMP 类型的属性。

选择分组窗口开始和结束时间戳

可以使用以下辅助功能选择组窗口的开始和结束时间戳以及时间属性：


Auxiliary Function	Description
<p>TUMBLE_START(time_attr, interval)</p> <p>HOP_START(time_attr, interval, interval)</p> <p>SESSION_START(time_attr, interval)</p>	<p>返回相应的滚动，跳动或会话窗口的包含下限的时间戳。</p>
<p>TUMBLE_END(time_attr, interval)</p>	<p>返回相应的滚动，跳跃或会话窗口的排他上限的时间戳。</p>

Auxiliary Function	Description
HOP_END(time_attr, interval, interval) SESSION_END(time_attr, interval)	Note: 排他上限时间戳不能在随后的基于时间的操作（例如带时间窗口的连接和分组窗口或窗口聚合中）中用作行时间属性。
TUMBLE_ROWTIME(time_attr, interval) HOP_ROWTIME(time_attr, interval, interval) SESSION_ROWTIME(time_attr, interval)	返回相应的滚动，跳跃或会话窗口的包含上限的时间戳。 结果属性是一个行时间属性，可以在随后的基于时间的操作（例如带时间窗口的连接和分组窗口或整个窗口聚合）中使用。
TUMBLE_PROCTIME(time_attr, interval) HOP_PROCTIME(time_attr, interval, interval) SESSION_PROCTIME(time_attr, interval)	返回一个 proctime 属性，该属性可以在随后的基于时间的操作（例如带时间窗口的连接和分组窗口或窗口聚合中）中使用。

注意：必须使用与 **GROUP BY** 子句中的分组窗口函数完全相同的参数来调用辅助函数。

以下示例说明如何在流表上使用分组窗口指定 **SQL** 查询。

```


val env = StreamExecutionEnvironment.getExecutionEnvironment
val tableEnv = StreamTableEnvironment.create(env)

// read a DataStream from an external source
val ds: DataStream[(Long, String, Int)] = env.addSource(...)
// register the DataStream under the name "Orders"
tableEnv.registerDataStream("Orders", ds, 'user, 'product, 'amount,
'proctime.proctime, 'rowtime.rowtime)

// compute SUM(amount) per day (in event-time)
val result1 = tableEnv.sqlQuery(
  """
    |SELECT
    |  user,
    |  TUMBLE_START(rowtime, INTERVAL '1' DAY) as wStart,
    |  SUM(amount)
    | FROM Orders
    | GROUP BY TUMBLE(rowtime, INTERVAL '1' DAY), user
  """
)

```

```

""").stripMargin)

// compute SUM(amount) per day (in processing-time)
val result2 = tableEnv.sqlQuery(
  "SELECT user, SUM(amount) FROM Orders GROUP BY TUMBLE(proctime,
INTERVAL '1' DAY), user")

// compute every hour the SUM(amount) of the last 24 hours in event-
time
val result3 = tableEnv.sqlQuery(
  "SELECT product, SUM(amount) FROM Orders GROUP BY HOP(rowtime,
INTERVAL '1' HOUR, INTERVAL '1' DAY), product")

// compute SUM(amount) per session with 12 hour inactivity gap (in
event-time)
val result4 = tableEnv.sqlQuery(
  """)
  |SELECT
  |  user,
  |  SESSION_START(rowtime, INTERVAL '12' HOUR) AS sStart,
  |  SESSION_END(rowtime, INTERVAL '12' HOUR) AS sEnd,
  |  SUM(amount)
  | FROM Orders
  | GROUP BY SESSION(rowtime(), INTERVAL '12' HOUR), user
  """).stripMargin)

```



Pattern Recognition

Operation	Description
MATCH_RECOGNIZE Streaming	<p>根据 MATCH_RECOGNIZE ISO 标准在流表中搜索给定的模式。这样就可以在 SQL 查询中表达复杂的事件处理（CEP）逻辑。</p> <p>有关更详细的描述，请参见用于检测表中模式的专用页面。</p> <pre> SELECT T.aid, T.bid, T.cid FROM MyTable MATCH_RECOGNIZE (PARTITION BY userid ORDER BY proctime MEASURES A.id AS aid, B.id AS bid, C.id AS cid </pre>

Operation	Description
	<pre> PATTERN (A B C) DEFINE A AS name = 'a', B AS name = 'b', C AS name = 'c') AS T </pre>

DDL


DDL 是通过 `TableEnvironment` 的 `sqlUpdate()` 方法指定的。对于成功创建表，该方法不返回任何内容。可以使用 `CREATE TABLE` 语句将表注册到目录中，然后可以在 `TableEnvironment` 的方法 `sqlQuery()` 中的 SQL 查询中引用表。

注意：Flink 的 DDL 支持尚未完成。包含不受支持的 SQL 功能的查询会导致 `TableException`。以下各节列出了批处理表和流表上 SQL DDL 的受支持功能。


指定 DDL

以下示例显示如何指定 SQL DDL。

```


val env = StreamExecutionEnvironment.getExecutionEnvironment
val tableEnv = StreamTableEnvironment.create(env)

// SQL query with a registered table
// register a table named "Orders"
tableEnv.sqlUpdate("CREATE TABLE Orders (`user` BIGINT, product VARCHAR, amount INT) WITH (...");
// run a SQL query on the Table and retrieve the result as a new Table
val result = tableEnv.sqlQuery(
  "SELECT product, amount FROM Orders WHERE product LIKE '%Rubber%'");

// SQL update with a registered table
// register a TableSink
tableEnv.sqlUpdate("CREATE TABLE RubberOrders (product VARCHAR, amount INT) WITH ('connector.path'='/path/to/file' ...)");
// run a SQL update query on the Table and emit the result to the TableSink
tableEnv.sqlUpdate(
  "INSERT INTO RubberOrders SELECT product, amount FROM Orders WHERE product LIKE '%Rubber%'")


```

Create Table

```
CREATE TABLE [catalog_name.][db_name.]table_name
  [(col_name1 col_type1 [COMMENT col_comment1], ...)]
  [COMMENT table_comment]
  [PARTITIONED BY (col_name1, col_name2, ...)]
  WITH (key1=val1, key2=val2, ...)
```

创建具有给定表属性的表。如果数据库中已经存在具有相同名称的表，则会引发异常。

PARTITIONED BY

按指定的列对创建的表进行分区。如果将此表用作文件系统接收器，则会为每个分区创建一个目录。

WITH OPTIONS

用于创建表源/接收器的表属性。这些属性通常用于查找和创建基础连接器。

表达式 **key1 = val1** 的键和值都应为字符串文字。有关不同连接器的所有受支持表属性，请参阅“连接到外部系统”中的详细信息。

注意：表名可以采用三种格式：1. **catalog_name.db_name.table_name** 2.

db_name.table_name 3. **table_name**。对于

catalog_name.db_name.table_name，该表将被注册到 **catalog** 名为“**catalog_name**”和数据库名为“**db_name**”的元存储中；对于 **db_name.table_name**，该表将被注册到执行表环境和名为“**db_name**”的数据库的当前 **catalog** 中；对于 **table_name**，该表将被注册到执行表环境的当前 **catalog** 和数据库中。

注意：用 **CREATE TABLE** 语句注册的表既可以用作表源，也可以用作表接收器，在 **DML** 中引用它之前，我们无法确定是将其用作源还是接收器。

Drop Table

```
DROP TABLE [IF EXISTS] [catalog_name.][db_name.]table_name
```

删除具有给定表名的表。如果要删除的表不存在，则会引发异常。

IF EXISTS

如果该表不存在，则什么也不会发生。

Data Types

请参阅有关[数据类型](#)的专用页面。

通用类型和（嵌套的）复合类型（例如 **POJO**，元组，行，**Scala** 案例类）也可以是一行的字段。

可以使用值访问功能访问具有任意嵌套的复合类型的字段。

泛型类型被视为黑盒，可以通过用户定义的函数传递或处理。

对于 **DDL**，我们支持在“数据类型”页面中定义的完整数据类型。

注意: sql 查询中不支持某些数据类型（强制转换表达式或文字）。例如。 **STRING**, **BYTES**, 不带时区的 **TIME (p)**, 带本地时区的 **TIME (p)**, 不带时区的 **TIMESTAMP (p)**, 带本地时区的 **TIMESTAMP (p)**, 数组, 多集, 行。

保留关键字

尽管尚未实现所有 **SQL** 功能, 但某些字符串组合已作为关键字保留, 以备将来使用。如果您想使用以下字符串之一作为字段名称, 请确保将其用反引号引起来 (例如, “ **value**”, “ **count**”)。



A, ABS, ABSOLUTE, ACTION, ADA, ADD, ADMIN, AFTER, ALL, ALLOCATE, ALLOW, ALTER, ALWAYS, AND, ANY, ARE, ARRAY, AS, ASC, ASENSITIVE, ASSERTION, ASSIGNMENT, ASYMMETRIC, AT, ATOMIC, ATTRIBUTE, ATTRIBUTES, AUTHORIZATION, AVG, BEFORE, BEGIN, BERNOULLI, BETWEEN, BIGINT, BINARY, BIT, BLOB, BOOLEAN, BOTH, BREADTH, BY, BYTES, C, CALL, CALLED, CARDINALITY, CASCADE, CASCADED, CASE, CAST, CATALOG, CATALOG_NAME, CEIL, CEILING, CENTURY, CHAIN, CHAR, CHARACTER, CHARACTERISTICS, CHARACTERS, CHARACTER_LENGTH, CHARACTER_SET_CATALOG, CHARACTER_SET_NAME, CHARACTER_SET_SCHEMA, CHAR_LENGTH, CHECK, CLASS_ORIGIN, CLOB, CLOSE, COALESCE, COBOL, COLLATE, COLLATION, COLLATION_CATALOG, COLLATION_NAME, COLLATION_SCHEMA, COLLECT, COLUMN, COLUMN_NAME, COMMAND_FUNCTION, COMMAND_FUNCTION_CODE, COMMIT, COMMITTED, CONDITION, CONDITION_NUMBER, CONNECT, CONNECTION, CONNECTION_NAME, CONSTRAINT, CONSTRAINTS, CONSTRAINT_CATALOG, CONSTRAINT_NAME, CONSTRAINT_SCHEMA, CONSTRUCTOR, CONTAINS, CONTINUE, CONVERT, CORR, CORRESPONDING, COUNT, COVAR_POP, COVAR_SAMP, CREATE, CROSS, CUBE, CUME_DIST, CURRENT, CURRENT_CATALOG, CURRENT_DATE, CURRENT_DEFAULT_TRANSFORM_GROUP, CURRENT_PATH, CURRENT_ROLE, CURRENT_SCHEMA, CURRENT_TIME, CURRENT_TIMESTAMP, CURRENT_TRANSFORM_GROUP_FOR_TYPE, CURRENT_USER, CURSOR, CURSOR_NAME, CYCLE, DATA, DATABASE, DATE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, DAY, DEALLOCATE, DEC, DECADE, DECIMAL, DECLARE, DEFAULT, DEFAULTS, DEFERRABLE, DEFERRED, DEFINED, DEFINER, DEGREE, DELETE, DENSE_RANK, DEPTH, Deref, DERIVED, DESC, DESCRIBE, DESCRIPTION, DESCRIPTOR, DETERMINISTIC, DIAGNOSTICS, DISALLOW, DISCONNECT, DISPATCH, DISTINCT, DOMAIN, DOUBLE, DOW, DOY, DROP, DYNAMIC, DYNAMIC_FUNCTION, DYNAMIC_FUNCTION_CODE, EACH, ELEMENT, ELSE, END, END-EXEC, EPOCH, EQUALS, ESCAPE, EVERY, EXCEPT, EXCEPTION, EXCLUDE, EXCLUDING, EXEC, EXECUTE, EXISTS, EXP, EXPLAIN, EXTEND, EXTERNAL, EXTRACT, FALSE, FETCH, FILTER, FINAL, FIRST, FIRST_VALUE, FLOAT, FLOOR, FOLLOWING, FOR, FOREIGN, FORTRAN, FOUND, FRAC_SECOND, FREE, FROM, FULL, FUNCTION, FUSION, G, GENERAL, GENERATED, GET,

GLOBAL, GO, GOTO, GRANT, GRANTED, GROUP, GROUPING, HAVING, HIERARCHY, HOLD, HOUR, IDENTITY, IMMEDIATE, IMPLEMENTATION, IMPORT, IN, INCLUDING, INCREMENT, INDICATOR, INITIALLY, INNER, INOUT, INPUT, INSENSITIVE, INSERT, INSTANCE, INSTANTIABLE, INT, INTEGER, INTERSECT, INTERSECTION, INTERVAL, INTO, INVOKER, IS, ISOLATION, JAVA, JOIN, K, KEY, KEY_MEMBER, KEY_TYPE, LABEL, LANGUAGE, LARGE, LAST, LAST_VALUE, LATERAL, LEADING, LEFT, LENGTH, LEVEL, LIBRARY, LIKE, LIMIT, LN, LOCAL, LOCALTIME, LOCALTIMESTAMP, LOCATOR, LOWER, M, MAP, MATCH, MATCHED, MAX, MAXVALUE, MEMBER, MERGE, MESSAGE_LENGTH, MESSAGE_OCTET_LENGTH, MESSAGE_TEXT, METHOD, MICROSECOND, MILLENNIUM, MIN, MINUTE, MINVALUE, MOD, MODIFIES, MODULE, MONTH, MORE, MULTISSET, MUMPS, NAME, NAMES, NATIONAL, NATURAL, NCHAR, NCLOB, NESTING, NEW, NEXT, NO, NONE, NORMALIZE, NORMALIZED, NOT, NULL, NULLABLE, NULLIF, NULLS, NUMBER, NUMERIC, OBJECT, OCTETS, OCTET_LENGTH, OF, OFFSET, OLD, ON, ONLY, OPEN, OPTION, OPTIONS, OR, ORDER, ORDERING, ORDINALITY, OTHERS, OUT, OUTER, OUTPUT, OVER, OVERLAPS, OVERLAY, OVERRIDING, PAD, PARAMETER, PARAMETER_MODE, PARAMETER_NAME, PARAMETER_ORDINAL_POSITION, PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_NAME, PARAMETER_SPECIFIC_SCHEMA, PARTIAL, PARTITION, PASCAL, PASSTHROUGH, PATH, PERCENTILE_CONT, PERCENTILE_DISC, PERCENT_RANK, PLACING, PLAN, PLI, POSITION, POWER, PRECEDING, PRECISION, PREPARE, PRESERVE, PRIMARY, PRIOR, PRIVILEGES, PROCEDURE, PUBLIC, QUARTER, RANGE, RANK, READ, READS, REAL, RECURSIVE, REF, REFERENCES, REFERENCING, REGR_AVGX, REGR_AVGY, REGR_COUNT, REGR_INTERCEPT, REGR_R2, REGR_SLOPE, REGR_SXX, REGR_SXY, REGR_SYY, RELATIVE, RELEASE, REPEATABLE, RESET, RESTART, RESTRICT, RESULT, RETURN, RETURNED_CARDINALITY, RETURNED_LENGTH, RETURNED_OCTET_LENGTH, RETURNED_SQLSTATE, RETURNS, REVOKE, RIGHT, ROLE, ROLLBACK, ROLLUP, ROUTINE, ROUTINE_CATALOG, ROUTINE_NAME, ROUTINE_SCHEMA, ROW, ROWS, ROW_COUNT, ROW_NUMBER, SAVEPOINT, SCALE, SCHEMA, SCHEMA_NAME, SCOPE, SCOPE_CATALOGS, SCOPE_NAME, SCOPE_SCHEMA, SCROLL, SEARCH, SECOND, SECTION, SECURITY, SELECT, SELF, SENSITIVE, SEQUENCE, SERIALIZABLE, SERVER, SERVER_NAME, SESSION, SESSION_USER, SET, SETS, SIMILAR, SIMPLE, SIZE, SMALLINT, SOME, SOURCE, SPACE, SPECIFIC, SPECIFICTYPE, SPECIFIC_NAME, SQL, SQLEXCEPTION, SQLSTATE, SQLWARNING, SQL_TSI_DAY, SQL_TSI_FRAC_SECOND, SQL_TSI_HOUR, SQL_TSI_MICROSECOND, SQL_TSI_MINUTE, SQL_TSI_MONTH, SQL_TSI_QUARTER, SQL_TSI_SECOND, SQL_TSI_WEEK, SQL_TSI_YEAR, SQRT, START, STATE, STATEMENT, STATIC, STDDEV_POP, STDDEV_SAMP, STREAM, STRING, STRUCTURE, STYLE, SUBCLASS_ORIGIN, SUBMULTISSET, SUBSTITUTE, SUBSTRING, SUM, SYMMETRIC, SYSTEM, SYSTEM_USER, TABLE, TABLESAMPLE, TABLE_NAME, TEMPORARY, THEN, TIES, TIME, TIMESTAMP, TIMESTAMPADD, TIMESTAMPDIFF, TIMEZONE_HOUR, TIMEZONE_MINUTE, TINYINT, TO, TOP_LEVEL_COUNT, TRAILING, TRANSACTION, TRANSACTIONS_ACTIVE,

```
TRANSACTIONS_COMMITTED, TRANSACTIONS_ROLLED_BACK, TRANSFORM,
TRANSFORMS, TRANSLATE, TRANSLATION, TREAT, TRIGGER, TRIGGER_CATALOG,
TRIGGER_NAME, TRIGGER_SCHEMA, TRIM, TRUE, TYPE, UESCAPE, UNBOUNDED,
UNCOMMITTED, UNDER, UNION, UNIQUE, UNKNOWN, UNNAMED, UNNEST, UPDATE,
UPPER, UPSERT, USAGE, USER, USER_DEFINED_TYPE_CATALOG,
USER_DEFINED_TYPE_CODE, USER_DEFINED_TYPE_NAME,
USER_DEFINED_TYPE_SCHEMA, USING, VALUE, VALUES, VARBINARY, VARCHAR,
VARYING, VAR_POP, VAR_SAMP, VERSION, VIEW, WEEK, WHEN, WHENEVER,
WHERE, WIDTH_BUCKET, WINDOW, WITH, WITHIN, WITHOUT, WORK, WRAPPER,
WRITE, XML, YEAR, ZONE
```



【翻译】Flink Table Api & SQL — 内置函数

本文翻译自官网：Built-In Functions <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/functions.html>

Flink Table Api & SQL 翻译目录

Flink Table API 和 SQL 为用户提供了一组用于数据转换的内置函数。此页面简要概述了它们。如果尚不支持所需的功能，则可以实现用户定义的功能。如果您认为该功能足够通用，请为此打开 Jira 问题，并提供详细说明。

- 标量函数
 - 比较功能
 - 逻辑功能
 - 算术函数
 - 字符串函数
 - 时间功能
 - 条件函数
 - 类型转换功能
 - 收集功能
 - 价值建构功能
 - 价值访问功能
 - 分组功能
 - 散列函数
 - 辅助功能
- 汇总功能
- 时间间隔和点单位说明符
- 列功能

标量函数

标量函数将零个，一个或多个值作为输入，并返回一个值作为结果。

比较功能

比较功能	描述
<code>ANY1 === ANY2</code>	如果 <i>ANY1</i> 等于 <i>ANY2</i> 返回 true ; 如果 <i>ANY1</i> 或 <i>ANY2</i> 为 NUL L ，则返回 UNKNOWN 。
<code>ANY1 !== ANY2</code>	如果 <i>ANY1</i> 不等于 <i>ANY2</i> 返回 true ; 如果 <i>ANY1</i> 或 <i>ANY2</i> 为 NUL L ，则返回 UNKNOWN 。
<code>ANY1 > ANY2</code>	如果 <i>ANY1</i> 大于 <i>ANY2</i> 返回 TRUE ; 如果 <i>ANY1</i> 或 <i>ANY2</i> 为 NUL L ，则返回 UNKNOWN 。
<code>ANY1 >= ANY2</code>	如果 <i>ANY1</i> 大于或等于 <i>ANY2</i> 返回 TRUE ; 如果 <i>ANY1</i> 或 <i>ANY2</i> 为 NULL ，则返回 UNKNOWN 。
<code>ANY1 < ANY2</code>	如果 <i>ANY1</i> 小于 <i>ANY2</i> 返回 TRUE ; 如果 <i>ANY1</i> 或 <i>ANY2</i> 为 NUL L ，则返回 UNKNOWN 。
<code>ANY1 <= ANY2</code>	如果 <i>ANY1</i> 小于或等于 <i>ANY2</i> 返回 TRUE ; 如果 <i>ANY1</i> 或 <i>ANY2</i> 为 NULL ，则返回 UNKNOWN 。
<code>ANY.isNull</code>	如果 <i>ANY</i> 为 NULL ，则返回 TRUE 。
<code>ANY.isNotNull</code>	如果 <i>ANY</i> 不为 NULL ，则返回 TRUE 。
<code>STRING1.like (STRING2)</code>	如果 <i>STRING1</i> 匹配模式 <i>STRING2</i> ，则返回 TRUE ; 如果 <i>STR I NG1</i> 或 <i>STRING2</i> 为 NULL ，则返回 UNKNOWN 。 例如， <code>"JoKn".like("Jo_n%")</code> 返回 TRUE 。
<code>STRING1.similar (STRING2)</code>	如果 <i>STRING1</i> 与 SQL 正则表达式 <i>STRING2</i> 匹配，则返回 TRU E ; 如果 <i>STRING1</i> 或 <i>STRING2</i> 为 NULL ，则返回 UNKNOW N 。 例如， <code>"A".similar("A+")</code> 返回 TRUE 。

比较功能	描述
<code>ANY1.in(ANY2, ANY3, ...)</code>	<p>如果 ANY1 存在于给定列表 (ANY2, ANY3, ...) 中, 则返回 TRUE。</p> <p>当 (ANY2, ANY3, ...) 包含 NULL, 如果可以找到该元素, 则返回 TRUE, 否则返回 UNKNOWN。</p> <p>如果 ANY1 为 NULL, 则始终返回 UNKNOWN。</p> <p>例如, <code>4.in(1, 2, 3)</code> 返回 FALSE。</p>
<code>ANY.in(TABLE)</code>	<p>如果 ANY 等于子查询 TABLE 返回的行, 则返回 TRUE。</p> <p>注意: 对于流查询, 该操作将在联接和分组操作中重写。</p> <p>根据不同输入行的数量, 计算查询结果所需的状态可能会无限增长。</p> <p>请提供具有有效保留间隔的查询配置, 以防止出现过多的状态。有关详细信息, 请参见查询配置。</p>
<code>ANY1.between(ANY2, ANY3)</code>	<p>如果 ANY1 大于或等于 ANY2 和小于或等于 ANY3 返回 <i>true</i>。当 ANY2 或 ANY3 为 NULL 时, 返回 FALSE 或 UNKNOWN。</p> <p>例如, <code>12.between(15, 12)</code> 返回 FALSE; <code>12.between(10, Null(Types.INT))</code> 返回 UNKNOWN; <code>12.between(Null(Types.INT), 10)</code> 返回 FALSE。</p>
<code>ANY1.notBetween(ANY2, ANY3)</code>	<p>如果 ANY1 小于 ANY2 或大于 ANY3 返回 <i>true</i>。当 ANY2 或 ANY3 为 NULL 时, 返回 TRUE 或 UNKNOWN。</p> <p>例如, <code>12.notBetween(15, 12)</code> 返回 TRUE; <code>12.notBetween(Null(Types.INT), 15)</code> 返回 UNKNOWN; <code>12.notBetween(15, Null(Types.INT))</code> 返回 TRUE。</p>

逻辑函数

逻辑功能	描述
<code>BOOLEAN1 BOOLEAN2</code>	<p>如果 BOOLEAN1 为 TRUE 或 BOOLEAN2 为 TRUE, 则返回 TRUE。支持三值逻辑。</p> <p>例如, <code>true Null(Types.BOOLEAN)</code> 返回 TRUE。</p>

逻辑功能	描述
<code>BOOLEAN1 && BOOLEAN2</code>	如果 <i>BOOLEAN1</i> 和 <i>BOOLEAN2</i> 均为 TRUE，则返回 TRUE。支持三值逻辑。 例如， <code>true && Null (Types.BOOLEAN)</code> 返回未知。
<code>!BOOLEAN</code>	如果 <i>BOOLEAN</i> 为 FALSE，则返回 TRUE ； 如果 <i>BOOLEAN</i> 为 TRUE，则返回 FALSE 。 如果 <i>BOOLEAN</i> 为 UNKNOWN，则返回 UNKNOWN。
<code>BOOLEAN.isTrue</code>	如果 <i>BOOLEAN</i> 为 TRUE，则返回 TRUE； 如果 <i>BOOLEAN</i> 为 FALSE 或 UNKNOWN，则返回 FALSE 。
<code>BOOLEAN.isFalse</code>	如果 <i>BOOLEAN</i> 为 FALSE，则返回 TRUE ； 如果 <i>BOOLEAN</i> 为 TRUE 或 UNKNOWN，则返回 FALSE 。
<code>BOOLEAN.isNotTrue</code>	如果 <i>BOOLEAN</i> 为 FALSE 或 UNKNOWN，则返回 TRUE ； 如果 <i>BOOLEAN</i> 为 TRUE，则返回 FALSE 。
<code>BOOLEAN.isNotFalse</code>	如果 <i>BOOLEAN</i> 为 TRUE 或 UNKNOWN，则返回 TRUE ； 如果 <i>BOOLEAN</i> 为 FALSE，则返回 FALSE。

算术函数

算术函数	描述
<code>+ NUMERIC</code>	返回 <i>NUMERIC</i> 。
<code>- NUMERIC</code>	返回负数 <i>NUMERIC</i> 。
<code>NUMERIC1 + NUMERIC2</code>	返回 <i>NUMERIC1</i> 加 <i>NUMERIC2</i> 。
<code>NUMERIC1 - NUMERIC2</code>	返回 <i>NUMERIC1</i> 减去 <i>NUMERIC2</i> 。

算术函数	描述
<code>NUMERIC1 * NUMERIC2</code>	返回 <i>NUMERIC1</i> 乘以 <i>NUMERIC2</i> 。
<code>NUMERIC1 / NUMERIC2</code>	返回 <i>NUMERIC1</i> 除以 <i>NUMERIC2</i> 。
<code>NUMERIC1.power(NUMERIC2)</code>	返回 <i>NUMERIC1</i> 的 <i>NUMERIC2</i> 次幂。
<code>NUMERIC.abs()</code>	返回 <i>NUMERIC</i> 的绝对值。
<code>NUMERIC1 % NUMERIC2</code>	返回 <i>NUMERIC1</i> 除以 <i>NUMERIC2</i> 的余数（模）。仅当 <i>numeric1</i> 为负数时，结果为负数。
<code>NUMERIC.sqrt()</code>	返回 <i>NUMERIC</i> 的平方根。
<code>NUMERIC.ln()</code>	返回 <i>NUMERIC</i> 的自然对数（以 e 为底）。
<code>NUMERIC.log10()</code>	返回 <i>NUMERIC</i> 的以 10 为底的对数。
<code>NUMERIC.log2()</code>	返回 <i>NUMERIC</i> 的以 2 为底的对数。
<code>NUMERIC1.log()</code> <code>NUMERIC1.log(NUMERIC2)</code>	如果不带参数调用，则返回 <i>NUMERIC1</i> 的自然对数。当使用参数调用时，将 <i>NUMERIC1</i> 的对数返回到基数 <i>NUMERIC2</i> 。 注意：当前， <i>NUMERIC1</i> 必须大于 0，而 <i>NUMERIC2</i> 必须大于 1。
<code>NUMERIC.exp()</code>	返回 e 的 <i>NUMERIC</i> 次幂。
<code>NUMERIC.ceil()</code>	将 <i>NUMERIC</i> 向上舍入，并返回大于或等于 <i>NUMERIC</i> 的最小整数。
<code>NUMERIC.floor()</code>	向下舍入 <i>NUMERIC</i> ，并返回小于或等于 <i>NUMERIC</i> 的最大整数。

算术函数	描述
<code>NUMERIC.sin()</code>	返回 <i>NUMERIC</i> 的正弦值。
<code>NUMERIC.sinh()</code>	返回 <i>NUMERIC</i> 的双曲正弦值。 返回类型为 <i>DOUBLE</i> 。
<code>NUMERIC.cos()</code>	返回 <i>NUMERIC</i> 的余弦值。
<code>NUMERIC.tan()</code>	返回 <i>NUMERIC</i> 的正切。
<code>NUMERIC.tanh()</code>	返回 <i>NUMERIC</i> 的双曲正切值。 返回类型为 <i>DOUBLE</i> 。
<code>NUMERIC.cot()</code>	返回 <i>NUMERIC</i> 的余切。
<code>NUMERIC.asin()</code>	返回 <i>NUMERIC</i> 的反正弦值。
<code>NUMERIC.acos()</code>	返回 <i>NUMERIC</i> 的反余弦值。
<code>NUMERIC.atan()</code>	返回 <i>NUMERIC</i> 的反正切。
<code>atan2(NUMERIC1, NUMERIC2)</code>	返回坐标的反正切 (<i>NUMERIC1</i> , <i>NUMERIC2</i>)。
<code>NUMERIC.cosh()</code>	返回 <i>NUMERIC</i> 的双曲余弦值。 返回值类型为 <i>DOUBLE</i> 。
<code>NUMERIC.degrees()</code>	返回弧度 <i>NUMERIC</i> 的度数表示形式。
<code>NUMERIC.radians()</code>	返回度数 <i>NUMERIC</i> 的弧度表示。

算术函数	描述
<code>NUMERIC.sign()</code>	返回 <i>NUMERIC</i> 的符号。
<code>NUMERIC.round(INT)</code>	返回一个数字，四舍五入为 <i>NUMERIC</i> 的 <i>INT</i> 小数位。
<code>pi()</code>	返回一个比 pi 更接近其他值的值。
<code>e()</code>	返回一个比任何其他值都更接近 e 的值。
<code>rand()</code>	返回介于 0.0 （含）和 1.0 （不含）之间的伪随机双精度值。
<code>rand(INTEGER)</code>	<p>返回带有初始种子 <i>INTEGER</i> 的介于 0.0（含）和 1.0（不含）之间的伪随机双精度值。</p> <p>如果两个 RAND 函数具有相同的初始种子，它们将返回相同的数字序列。</p>
<code>randInteger(INTEGER)</code>	返回介于 0 （含）和 <i>INTEGER</i> （不含）之间的伪随机整数值。
<code>randInteger(INTEGER1, INTEGER2)</code>	<p>返回介于 0（含）和 <i>INTEGER2</i>（不含）之间的伪随机整数值，其初始种子为 <i>INTEGER1</i>。</p> <p>如果两个 randInteger 函数具有相同的初始种子和边界，它们将返回相同的数字序列。</p>
<code>uuid()</code>	<p>根据 RFC 4122 type 4（伪随机生成）UUID 返回 UUID（通用唯一标识符）字符串</p> <p>（例如，" 3d3c68f7-f608-473f-b60c-b0c44ad4cc4e"）。使用加密强度高的伪随机数生成器生成 UUID。</p>
<code>INTEGER.bin()</code>	<p>以二进制格式返回 <i>INTEGER</i> 的字符串表示形式。如果 <i>INTEGER</i> 为 NULL，则返回 NULL。</p> <p>例如，<code>4.bin()</code> 返回 " 100" 并 <code>12.bin()</code> 返回 " 1100"。</p>

算术函数	描述
<code>NUMERIC.hex()</code> <code>STRING.hex()</code>	<p>以十六进制格式返回整数 <i>NUMERIC</i> 值或 <i>STRING</i> 的字符串表示形式。如果参数为 <code>NULL</code>，则返回 <code>NULL</code>。</p> <p>例如，数字 20 导致“ 14”，数字 100 导致“ 64”，字符串“ hello, world”导致“ 68656C6C6F2C776F726C64”。</p>

字符串函数

字符串函数	描述
<code>STRING1 + STRING2</code>	返回 <i>STRING1</i> 和 <i>STRING2</i> 的串联。
<code>STRING.charLength()</code>	返回 <i>STRING</i> 中的字符数。
<code>STRING.upperCase()</code>	以大写形式返回 <i>STRING</i> 。
<code>STRING.lowerCase()</code>	以小写形式返回 <i>STRING</i> 。
<code>STRING1.position(STRING2)</code>	<p>返回 <i>STRING1</i> 在 <i>STRING2</i> 中第一次出现的位置（从 1 开始）；</p> <p>如果在 <i>STRING2</i> 中找不到 <i>STRING1</i>，则返回 0 。</p>
<code>STRING.trim(</code> <code>leading = true,</code> <code>trailing = true,</code> <code>character = " ")</code>	返回一个字符串，该字符串从 <i>STRING</i> 中删除前导和/或结尾字符。
<code>STRING.ltrim()</code>	<p>返回一个字符串，该字符串从 <i>STRING</i> 除去左空格。</p> <p>例如，" This is a test String."<code>.ltrim()</code> 返回 "This is a test String."。</p>
<code>STRING.rtrim()</code>	<p>返回一个字符串，该字符串从 <i>STRING</i> 中删除正确的空格。</p> <p>例如，"This is a test String. "<code>.rtrim()</code> 返回 "This is a test String."。</p>

字符串函数	描述
<code>STRING.repeat(INT)</code>	<p>返回一个字符串，该字符串重复基本 <i>STRING</i> <i>INT</i> 次。</p> <p>例如, "This is a test String.".repeat(2) 返回 "This is a test String.This is a test String."。</p>
<code>STRING1.regexReplace(STRING2, STRING3)</code>	<p>返回字符串 <i>STRING1</i> 所有匹配正则表达式的子串 <i>STRING2</i> 连续被替换 <i>STRING3</i>。</p> <p>例如, "foobar".regexReplace("oo ar", "") 返回 "fb"。</p>
<code>STRING1.overlay(STRING2, INT1)</code> <code>STRING1.overlay(STRING2, INT1, INT2)</code>	<p>从位置 <i>INT1</i> 返回一个字符串，该字符串将 <i>STRING1</i> 的 <i>INT2</i>（默认为 <i>STRING2</i> 的长度）字符替换为 <i>STRING2</i>。</p> <p>例如, "xxxxxtest".overlay("xxxx", 6) 返回 "xxx xxxxxx"; "xxxxxtest".overlay("xxxx", 6, 2) 返回 "xxxxxxxxxst"。</p>
<code>STRING.substring(INT1)</code> <code>STRING.substring(INT1, INT2)</code>	<p>返回字符串 <i>STRING</i> 的子字符串，从位置 <i>INT1</i> 开始，长度为 <i>INT2</i>（默认为结尾）。</p>
<code>STRING1.replace(STRING2, STRING3)</code>	<p>返回一个新字符串替换其中出现的所有 <i>STRING2</i> 与 <i>STRING3</i>（非重叠）从 <i>STRING1</i>。</p> <p>例如, "hello world".replace("world", "flink") 返回 "hello flink"; "ababab".replace("abab", "z") 返回 "zab"。</p>
<code>STRING1.regexExtract(STRING2[, INTEGER1])</code>	<p>从 <i>STRING1</i> 返回一个字符串，该字符串使用指定的正则表达式 <i>STRING2</i> 和正则表达式匹配组索引 <i>INTEGER1</i> 提取。</p> <p>注意: regex 匹配组索引从 1 和 0 开始，表示匹配整个 regex。另外，正则表达式匹配组索引不应超过定义的组数。</p> <p>例如, "foothebar".regexExtract("foo.(*?)(bar)", 2) 返回 "bar"。</p>

字符串函数	描述
<code>STRING.initCap()</code>	<p>返回一种新形式的 <i>STRING</i>，其中每个单词的第一个字符转换为大写，其余字符转换为小写。</p> <p>这里的单词表示字母数字字符序列。</p>
<code>concat(STRING1, STRING2, ...)</code>	<p>返回连接 <i>STRING1</i>, <i>STRING2</i>, ... 的字符串。如果任何参数为 <code>NULL</code>，则返回 <code>NULL</code>。</p> <p>例如，<code>concat("AA", "BB", "CC")</code> 返回 "AABBC C"。</p>
<code>concat_ws(STRING1, STRING2, STRING3, ...)</code>	<p>返回一个字符串，会连接 <i>STRING2</i>, <i>STRING3</i>, 与分离 <i>STRING1</i>。</p> <p>分隔符被添加到要连接的字符串之间。如果 <i>STRING1</i> 为 <code>NULL</code>，则返回 <code>NULL</code>。</p> <p>与相比 <code>concat()</code>，会 <code>concat_ws()</code> 自动跳过 <code>NULL</code> 参数。</p> <p>例如，<code>concat_ws("~", "AA", Null(Types.STRING), "BB", "", "CC")</code> 返回 "AA~BB ~~ CC"。</p>
<code>STRING1.lpad(INT, STRING2)</code>	<p>返回一个新字符串，该字符串从 <i>STRING1</i> 的左侧填充 <i>STRING2</i>，长度为 <i>INT</i> 个字符。</p> <p>如果 <i>STRING1</i> 的长度小于 <i>INT</i>，则返回缩短为 <i>INT</i> 个字符的 <i>STRING1</i>。</p> <p>例如，<code>"hi".lpad(4, "??")</code> 返回 "?? hi"；<code>"hi".lpad(1, "??")</code> 返回 "h"。</p>
<code>STRING1.rpad(INT, STRING2)</code>	<p>返回一个新字符串，该字符串从 <i>STRING1</i> 右侧填充 <i>STRING2</i>，长度为 <i>INT</i> 个字符。</p> <p>如果 <i>STRING1</i> 的长度小于 <i>INT</i>，则返回缩短为 <i>INT</i> 个字符的 <i>STRING1</i>。</p> <p>例如，<code>"hi".rpad(4, "??")</code> 返回 "hi ??"; <code>"hi".rpad(1, "??")</code> 返回 "h"。</p>

字符串函数	描述
<code>STRING.fromBase64()</code>	<p>返回来自 <i>STRING</i> 的 base64 解码结果；如果 <i>STRING</i> 为 NULL，则返回 null。</p> <p>例如，<code>"aGVsbG8gd29ybGQ=".fromBase64()</code> 返回 "hello world"。</p>
<code>STRING.toBase64()</code>	<p>从 <i>STRING</i> 返回 base64 编码的结果；如果 <i>STRING</i> 为 NULL，则返回 NULL。</p> <p>例如，<code>"hello world".toBase64()</code> 返回 "aGVsbG8gd29ybGQ="。</p>

时间函数

时间功能	描述
<code>STRING.toDate</code>	返回以 "yyyy-MM-dd" 形式从 <i>STRING</i> 解析的 SQL 日期。
<code>STRING.toTime</code>	返回以 "HH: mm: ss" 的形式从 <i>STRING</i> 解析的 SQL 时间。
<code>STRING.toTimestamp</code>	返回从 <i>STRING</i> 解析的 SQL 时间戳，格式为 "yyyy-MM-dd HH: mm: ss [.SSS]"。
<code>NUMERIC.year</code> <code>NUMERIC.years</code>	为 <i>NUMERIC</i> 年创建一个时间间隔。
<code>NUMERIC.quarter</code> <code>NUMERIC.quarters</code>	<p>为 <i>NUMERIC</i> 个季度创建一个时间间隔。</p> <p>例如，<code>2.quarters</code> 传回 6。</p>
<code>NUMERIC.month</code> <code>NUMERIC.months</code>	创建间隔 <i>NUMERIC</i> 个月。
<code>NUMERIC.week</code> <code>NUMERIC.weeks</code>	<p>创建 <i>NUMERIC</i> 周的毫秒间隔。</p> <p>例如，<code>2.weeks</code> 传回 1209600000。</p>

时间功能	描述
<code>NUMERIC.day</code> <code>NUMERIC.days</code>	创建 <i>NUMERIC</i> 天的毫秒间隔。
<code>NUMERIC.hour</code> <code>NUMERIC.hours</code>	创建 <i>NUMERIC</i> 小时的毫秒间隔。
<code>NUMERIC.minute</code> <code>NUMERIC.minutes</code>	为 <i>NUMERIC</i> 分钟创建一个毫秒间隔。
<code>NUMERIC.second</code> <code>NUMERIC.seconds</code>	为 <i>NUMERIC</i> 秒创建毫秒间隔。
<code>NUMERIC.milli</code> <code>NUMERIC.millis</code>	创建一个 <i>NUMERIC</i> 毫秒的时间间隔。
<code>currentDate()</code>	返回 UTC 时区中的当前 SQL 日期。
<code>currentTime()</code>	返回 UTC 时区中的当前 SQL 时间。
<code>currentTimestamp()</code>	返回 UTC 时区中的当前 SQL 时间戳。
<code>localTime()</code>	返回本地时区的当前 SQL 时间。
<code>localTimestamp()</code>	返回本地时区的当前 SQL 时间戳。
<code>TEMPORAL.extract(TEMPORAL.TIMEINTERVALUNIT)</code>	<p>返回从 <i>temporal</i> 的 <i>TIMEINTERVALUNIT</i> 部分中提取的长值。</p> <p>例如, <code>"2006-06-05".toDate.extract(TimeIntervalUnit.DAY)</code> 返回 5;</p> <p><code>"2006-06-05".toDate.extract(QUARTER)</code> 返回 2。</p>
<code>TIMEPOINT.floor(TIMEPOINT.TIMEINTERVALUNIT)</code>	<p>返回将 <i>TIMEPOINT</i> 向下舍入为时间单位 <i>TIMEINTERVALUNIT</i> 的值。</p> <p>例如, <code>"12:44:31".toDate.floor(TimeIntervalUnit.MINUTE)</code> 返回 12:44:00。</p>

时间功能	描述
<code>TIMEPOINT.ceil(TIMEINTERVALUNIT)</code>	<p>返回将 <i>TIMEPOINT</i> 舍入为时间单位 <i>TIMEINTERVALUNIT</i> 的值。</p> <p>例如, "12:44:31".toTime.floor(TimeIntervalUnit.MINUTE) 返回 12:45:00。</p>
<code>temporalOverlaps(TIMEPOINT1, TEMPORAL1, TIMEPOINT2, TEMPORAL2)</code>	<p>如果 (<i>TIMEPOINT1</i>, <i>TEMPORAL1</i>) 和 (<i>TIMEPOINT2</i>, <i>TEMPORAL2</i>) 定义的两个时间间隔重叠, 则返回 TRUE。</p> <p>时间值可以是时间点, 也可以是时间间隔。</p> <p>例如, temporalOverlaps("2:55:00".toTime, 1.hour, "3:30:00".toTime, 2.hour) 返回 TRUE。</p>
<code>dateFormat(TIMESTAMP, STRING)</code>	<p>注意此功能存在严重的错误, 暂时不应使用。请改为实施自定义 UDF 或使用 <code>extract()</code> 作为解决方法。</p>
<code>timestampDiff(TIMEPOINTUNIT, TIMEPOINT1, TIMEPOINT2)</code>	<p>返回 <i>TIMEPOINT1</i> 和 <i>TIMEPOINT2</i> 之间的 <i>TIMEPOINTUNIT</i> 的 (带符号) 编号。时间间隔的单位由第一个参数指定, 该参数应为以下值之一: SECOND, MINUTE, HOUR, DAY, MONTH 或 YEAR。</p> <p>另请参见时间间隔和点单位说明符表。</p> <p>例如, timestampDiff(TimePointUnit.DAY, '2003-01-02 10: 00: 00'.toTimestamp, '2003-01-03 10: 00: 00'.toTimestamp) 返回 1。</p> <p>。</p>

条件函数

条件函数	描述
<code>BOOLEAN.?(VALUE1, VALUE2)</code>	<p>如果 <i>BOOLEAN</i> 的计算结果为 TRUE, 则返回 <i>VALUE1</i>; 否则, 返回 <i>VALUE1</i>。否则返回 <i>VALUE2</i>。</p> <p>例如, (42 > 5).?("A", "B") 返回 "A"。</p>

类型转换函数功能

类型转换函数	描述
<code>ANY.cast (TYPE)</code>	<p>返回转换为 <i>TYPE</i> 类型的新 <i>ANY</i>。请在此处查看受支持的类型。</p> <p>例如, <code>"42".cast (Types.INT)</code> 返回 42; <code>Null (Types.STRING)</code> 返回 STRING 类型的 NULL。</p>

Collection 函数功能

Collection 函数	描述
<code>ARRAY.cardinality ()</code>	返回 <i>ARRAY</i> 中的元素数量。
<code>ARRAY.at (INT)</code>	返回位于元素 <i>INT</i> 的 <i>ARRAY</i> 。索引从 1 开始。
<code>ARRAY.element ()</code>	<p>返回 <i>ARRAY</i> 的唯一元素（其基数应为 1）；否则为 <i>false</i>。</p> <p>如果 <i>ARRAY</i> 为空，则返回 NULL。如果 <i>ARRAY</i> 具有多个元素，则引发异常。</p>
<code>MAP.cardinality ()</code>	返回 <i>MAP</i> 中的条目数。
<code>MAP.at (ANY)</code>	返回键指定的值 <i>ANY</i> 在 <i>MAP</i> 。

Value Construction 构造函数

Value Construction 函数	描述
<code>row (ANY1, ANY2, ...)</code>	返回从对象值 (<i>ANY1</i> , <i>ANY2</i> , ...) 的列表创建的行。行是复合类型，可以通过 值访问函数 进行访问。
<code>array (ANY1, ANY2, ...)</code>	返回从对象值 (<i>ANY1</i> , <i>ANY2</i> , ...) 的列表创建的数组。
<code>map (ANY1, ANY2, ANY3, ANY4, ...)</code>	返回从键值对列表 ((<i>ANY1</i> , <i>ANY2</i>), (<i>ANY3</i> , <i>ANY4</i>), ...) 创建的映射。

Value Construction 函数	描述
<code>NUMERIC.rows</code>	创建行的 <i>NUMERIC</i> 间隔（通常在窗口创建中使用）。

Value Access 函数

Value Access 函数	描述
<code>COMPOSITE.get (STRING)</code> <code>COMPOSITE.get (INT)</code>	通过名称或索引从 Flink 复合类型（例如， Tuple ， POJO ）返回字段的值。 例如， <code>'pojo.get("myField")</code> 或 <code>'tuple.get(0)</code> 。
<code>ANY.flatten()</code>	返回 Flink 复合类型（例如 Tuple ， POJO ）的平面表示形式，该类型将其每个直接子类型转换为单独的字段。 在大多数情况下，平面表示形式的字段与原始字段的命名方式相似，但带有美元分隔符（例如， <code>mypojo \$ mytuple \$ f0</code> ）。

分组函数

分组函数	描述
<code>GROUP_ID()</code>	返回一个唯一标识分组键组合的整数。
<code>GROUPING(expression1 [, expression2] *)</code> <code>GROUPING_ID(expression1 [, expression2] *)</code>	返回给定分组表达式的位向量。

hash 函数

hash 函数	描述
<code>STRING.md5()</code>	以 32 个十六进制数字的字符串形式返回 <i>STRING</i> 的 MD5 哈希值；如果 <i>STRING</i> 为 NULL ，则返回 NULL 。

hash 函数	描述
<code>STRING.sha1()</code>	以 40 个十六进制数字的字符串形式返回 <i>STRING</i> 的 SHA-1 哈希值；如果 <i>STRING</i> 为 NULL，则返回 NULL。
<code>STRING.sha224()</code>	以 56 个十六进制数字的字符串形式返回 <i>STRING</i> 的 SHA-224 哈希值；如果 <i>STRING</i> 为 NULL，则返回 NULL。
<code>STRING.sha256()</code>	以 64 个十六进制数字的字符串形式返回 <i>STRING</i> 的 SHA-256 哈希值；如果 <i>STRING</i> 为 NULL，则返回 NULL。
<code>STRING.sha384()</code>	以 96 个十六进制数字的字符串返回 <i>STRING</i> 的 SHA-384 哈希值；如果 <i>STRING</i> 为 NULL，则返回 NULL。
<code>STRING.sha512()</code>	以 128 个十六进制数字的字符串返回 <i>STRING</i> 的 SHA-512 哈希值；如果 <i>STRING</i> 为 NULL，则返回 NULL。
<code>STRING.sha2(INT)</code>	返回由 <i>INT</i> （可能为 224、256、384 或 512）为 <i>STRING</i> 指定的 SHA-2 系列（SHA-224，SHA-256，SHA-384 或 SHA-512）哈希值。如果 <i>STRING</i> 或 <i>INT</i> 为 NULL，则返回 NULL。

辅助函数

辅助函数	描述
<code>ANY.as(NAME1, NAME2, ...)</code>	指定 <i>ANY</i> 的名称（字段）。如果表达式扩展到多个字段，则可以指定其他名称。

汇总函数

聚合函数将所有行中的表达式作为输入，并返回单个聚合值作为结果。

汇总功能	描述
<code>FIELD.count</code>	返回 <i>FIELD</i> 不为 NULL 的输入行数。

汇总功能	描述
<code>FIELD.avg</code>	返回所有输入行中 <i>FIELD</i> 的平均值（算术平均值）。
<code>FIELD.sum</code>	返回所有输入行中数字字段 <i>FIELD</i> 的总和。如果所有值均为 NULL ，则返回 NULL 。
<code>FIELD.sum0</code>	返回所有输入行中数字字段 <i>FIELD</i> 的总和。如果所有值均为 NULL ，则返回 0 。
<code>FIELD.max</code>	返回所有输入行中数字字段 <i>FIELD</i> 的最大值。
<code>FIELD.min</code>	返回所有输入行中数字字段 <i>FIELD</i> 的最小值。
<code>FIELD.stddevPop</code>	返回所有输入行中数字字段 <i>FIELD</i> 的总体标准偏差。
<code>FIELD.stddevSamp</code>	返回所有输入行中数字字段 <i>FIELD</i> 的样本标准偏差。
<code>FIELD.varPop</code>	返回所有输入行中数字字段 <i>FIELD</i> 的总体方差（总体标准差的平方）。
<code>FIELD.varSamp</code>	返回所有输入行中数字字段 <i>FIELD</i> 的样本方差（样本标准差的平方）。
<code>FIELD.collect</code>	返回所有输入行的 <i>FIELD</i> 的多集。

时间间隔和点单位说明符

下表列出了时间间隔和时间单位的说明符。

对于 **Table API**，请使用 `_` 空格（例如 `DAY_TO_HOUR`）。

时间间隔单位	时间点单位
MILLENIUM (仅 SQL)	
CENTURY (仅 SQL)	
YEAR	YEAR
YEAR TO MONTH	
QUARTER	QUARTER
MONTH	MONTH
WEEK	WEEK
DAY	DAY
DAY TO HOUR	
DAY TO MINUTE	
DAY TO SECOND	
HOUR	HOUR
HOUR TO MINUTE	
HOUR TO SECOND	
MINUTE	MINUTE
MINUTE TO SECOND	
SECOND	SECOND
	MILLISECOND
	MICROSECOND
DOY (仅 SQL)	
DOW (仅 SQL)	
	SQL_TSI_YEAR (仅 SQL)
	SQL_TSI_QUARTER (仅 SQL)


时间间隔单位	时间点单位
	SQL_TSI_MONTH (仅 SQL)
	SQL_TSI_WEEK (仅 SQL)
	SQL_TSI_DAY (仅 SQL)
	SQL_TSI_HOUR (仅 SQL)
	SQL_TSI_MINUTE (仅 SQL)
	SQL_TSI_SECOND (仅 SQL)

列函数

列函数用于选择或删除列。

SYNTAX	DESC
<code>withColumns(...)</code>	选择的列
<code>withoutColumns(...)</code>	不选择的列

详细语法如下：




```
columnFunction:
    withColumns(columnExprs)
    withoutColumns(columnExprs)

columnExprs:
    columnExpr [, columnExpr]*

columnExpr:
    columnRef | columnIndex to columnIndex | columnName to columnName

columnRef:
    columnName(The field name that exists in the table) |
columnIndex(a positive integer starting from 1)
```



下表说明了 `column` 函数的用法。（假设我们有一个包含 5 列的表）（a: Int, b: Long, c: String, d:String, e: String）：

api	用法	描述
<code>withColumns (*)</code> <code> *</code>	<code>select(withColumns('*')) select('*) = select('a, 'b, 'c, 'd, 'e)</code>	所有列
<code>withColumns(m 至 n)</code>	<code>select(withColumns(2 to 4)) = select('b, 'c, 'd)</code>	从 m 到 n 的列
<code>withColumns(m, n, k)</code>	<code>select(withColumns(1, 3, 'e)) = select('a, 'c, 'e)</code>	m, n, k 列
<code>withColumns(m, n to k)</code>	<code>select(withColumns(1, 3 to 5)) = select('a, 'c, 'd, 'e)</code>	上面两种表示的混合
<code>withoutColumns(m to n)</code>	<code>select(withoutColumns(2 to 4)) = select('a, 'e)</code>	取消选择从 m 到 n 的列
<code>withoutColumns(m, n, k)</code>	<code>select(withoutColumns(1, 3, 5)) = select('b, 'd)</code>	取消选择列 m, n, k
<code>withoutColumns(m, n to k)</code>	<code>select(withoutColumns(1, 3 to 5)) = select('b)</code>	上面两种表示的混合

列函数可以在所有需要列字段的地方使用，`select`，`groupBy`，`orderBy`，UDFs 等。例如：

```
table
    .groupBy(withColumns(1 to 3))
    .select(withColumns('a to 'b), myUDAgg(myUDF(withColumns(5 to 20))))
```

注意：列函数仅在 **Table API** 中使用。

【翻译】Flink Table Api & SQL — 自定义 Source & Sink

本文翻译自官网：User-defined Sources & Sinks <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/sourceSinks.html>

Flink Table Api & SQL 翻译目录

`TableSource` 提供对存储在外部系统（数据库，键值存储，消息队列）或文件中的数据的访问。在 **TableEnvironment** 中注册 **TableSource** 后，可以通过 **Table API** 或 **SQL** 查询对其进行访问。

`TableSink` 将表发送到外部存储系统，例如数据库，键值存储，消息队列或文件系统（采用不同的编码，例如 `CSV`，`Parquet` 或 `ORC`）。

`TableFactory` 允许将与外部系统的连接的声明与实际实现分开。`TableFactory` 从标准化的基于字符串的属性创建表 `source` 和 `sink` 的已配置实例。可以使用 `Descriptor` 或通过 [SQL Client](#) 的 `YAML` 配置文件以编程方式生成属性。

看一下[通用概念](#)和[API](#)页面，详细了解如何[注册 `TableSource`](#)以及如何[通过 `TableSink` 发出表](#)。有关如何使用工厂的示例，[请参见内置的源，接收器和格式](#)页面。

- [定义 `TableSource`](#)
 - [定义 `BatchTableSource`](#)
 - [定义 `StreamTableSource`](#)
 - [使用时间属性定义 `TableSource`](#)
 - [使用投影下推定义 `TableSource`](#)
 - [使用过滤器下推定义 `TableSource`](#)
 - [定义用于查找的 `TableSource`](#)
- [定义 `TableSink`](#)
 - [BatchTableSink](#)
 - [AppendStreamTableSink](#)
 - [RetractStreamTableSink](#)
 - [UpsertStreamTableSink](#)
- [定义 `TableFactory`](#)
 - [在 `SQL` 客户端中使用 `TableFactory`](#)
 - [在 `Table & SQL API` 中使用 `TableFactory`](#)

定义 `TableSource`

`TableSource` 是一个通用接口，使 `Table API` 和 `SQL` 查询可以访问存储在外部系统中的数据。它提供了表结构以及与该表结构映射到行的记录。根据 `TableSource` 是在流查询还是批处理查询中使用，记录将生成为 `DataSet` 或 `DataStream`。

如果 `TableSource` 在流查询中使用，则必须实现 `StreamTableSource` 接口，如果在批处理查询中使用，则必须实现 `BatchTableSource` 接口。`TableSource` 还可以同时实现两个接口，并且可以在流查询和批处理查询中使用。

`StreamTableSource` 和 `BatchTableSource` 扩展 `TableSource` 定义以下方法的基本接口：

```
TableSource[T] {  
  
  def getTableSchema: TableSchema  
  
  def getReturnType: TypeInformation[T]  
  
  def explainSource: String
```

```
}
```



- `getTableSchema()`: 返回表结构，即表的字段的名称和类型。字段类型是使用 Flink 定义的 `TypeInformation`（请参见 [Table API 类型](#) 和 [SQL 类型](#)）。
- `getReturnType()`: 返回 `DataStream` (`StreamTableSource`) 或 `DataSet` (`BatchTableSource`) 的物理类型以及由产生的记录 `TableSource`。
- `explainSource()`: 返回描述的字符串 `TableSource`。此方法是可选的，仅用于显示目的。

`TableSource` 接口将逻辑表架构与返回的 `DataStream` 或 `DataSet` 的物理类型分开。因此，表结构的所有字段（`getTableSchema()`）必须映射到具有相应物理返回类型（`getReturnType()`）类型的字段。默认情况下，此映射是基于字段名称完成的。例如，一个 `TableSource` 定义具有两个字段[`name: String, size: Integer`]的表结构，它需要 `TypeInformation` 至少具有两个字段，分别名为 `name` 和 `size`，类型分别为 `String` 和 `Integer`。这可以是 `PojoTypeInfo` 或 `RowTypeInfo`，它们具有两个名为 `name` 和 `size` 且具有匹配类型的字段。

但是，某些类型（例如 `Tuple` 或 `CaseClass` 类型）支持自定义字段名称。如果 `TableSource` 返回具有固定字段名称的类型的 `DataStream` 或 `DataSet`，则它可以实现 `DefinedFieldMapping` 接口以将表结构中的字段名称映射到物理返回类型的字段名称。

定义 `BatchTableSource`

`BatchTableSource` 接口扩展了 `TableSource` 接口，并定义一个额外的方法：

```
BatchTableSource[T] extends TableSource[T] {  
  
    def getDataSet(execEnv: ExecutionEnvironment): DataSet[T]  
}
```

- `getDataSet(execEnv)`: 返回带有表数据的 `DataSet`。`DataSet` 的类型必须与 `TableSource.getReturnType()` 方法定义的返回类型相同。可以使用 `DataSet API` 的常规数据源创建 `DataSet`。通常，`BatchTableSource` 是通过包装 `InputFormat` 或 [batch connector](#) 实现的。

定义 `StreamTableSource`

`StreamTableSource` 接口扩展了 `TableSource` 接口，并定义一个额外的方法：

```
StreamTableSource[T] extends TableSource[T] {  
  
    def getDataStream(execEnv: StreamExecutionEnvironment):  
    DataStream[T]  
}
```

- `getDataStream(execEnv)`: 返回带有表格数据的 `DataStream`。`DataStream` 的类型必须与 `TableSource.getReturnType()` 方法定义的返回类型相同。可以使用 `DataSet API` 的常规数据源创建 `DataSet`。通常, `StreamTableSource` 是通过包装 `SourceFunction` 或 [batch connector](#) 实现的。

使用时间属性定义 `TableSource`

流 `Table API` 和 `SQL` 查询的基于时间的操作（例如窗口聚合或 `join`）需要显式指定的时间属性。

`TableSource` 在其表结构中将时间属性定义为 `Types.SQL_TIMESTAMP` 类型的字段。与结构中的所有常规字段相反, 时间属性不得与表 `source` 的返回类型中的物理字段匹配。相反, `TableSource` 通过实现某个接口来定义时间属性。

定义处理时间属性

处理时间属性通常用于流查询中。处理时间属性返回访问该属性的 `operator` 的当前挂钟时间。`TableSource` 通过实现 `DefinedProctimeAttribute` 接口来定义处理时间属性。接口如下所示:

```
DefinedProctimeAttribute {  
  
    def getProctimeAttribute: String  
}
```

- `getProctimeAttribute()`: 返回处理时间属性的名称。指定的属性必须 `Types.SQL_TIMESTAMP` 在表结构中定义为类型, 并且可以在基于时间的操作中使用。`DefinedProctimeAttribute` 表 `source` 无法通过返回 `null` 来定义任何处理时间属性。

注意两者 `StreamTableSource` 和 `BatchTableSource` 可以实现 `DefinedProctimeAttribute` 并定义的处理时间属性。在 `BatchTableSource` 表扫描期间, 使用当前时间戳初始化处理时间字段的情况。

定义行时间属性

行时间属性是类型的属性, `TIMESTAMP` 在流查询和批处理查询中以统一的方式处理。

`SQL_TIMESTAMP` 通过指定以下内容, 可以将类型的表模式字段声明为 `rowtime` 属性:

- 字段名称,
- `TimestampExtractor`, 计算实际值的属性（通常从一个或多个其他字段）
- `WatermarkStrategy` 用于指定如何为 `rowtime` 属性生成水印的。

`TableSource` 通过实现 `DefinedRowtimeAttributes` 接口来定义行时间属性。该接口如下所示:

```
DefinedRowtimeAttributes {
```

```
def getRowtimeAttributeDescriptors:  
    util.List[RowtimeAttributeDescriptor]  
}
```

- `getRowtimeAttributeDescriptors()`: 返回 `RowtimeAttributeDescriptor` 的列表。 `RowtimeAttributeDescriptor` 描述了具有以下属性的行时间属性:
 - `attributeName`: 表结构中的 **rowtime** 属性的名称。该字段必须使用 `Types.SQL_TIMESTAMP` 类型定义。
 - `timestampExtractor`: 时间戳提取器从具有返回类型的记录中提取时间戳。例如, 它可以将 **Long** 字段转换为时间戳, 或者解析 **String** 编码的时间戳。Flink 带有一组 `TimestampExtractor` 针对常见用例的内置实现。也可以提供自定义实现。
 - `watermarkStrategy`: 水印策略定义了如何使用 **rowtime** 属性生成水印。Flink 带有一组 `WatermarkStrategy` 用于常见用例的内置实现。也可以提供自定义实现。

注意: 尽管该 `getRowtimeAttributeDescriptors()` 方法返回一个描述符列表, 但目前仅支持单个 **rowtime** 属性。我们计划将来删除此限制, 并支持具有多个 **rowtime** 属性的表。

注意: 两者, `StreamTableSource` 和 `BatchTableSource`, 可以实现 `DefinedRowtimeAttributes` 并定义 **rowtime** 属性。无论哪种情况, 都使用来提取 **rowtime** 字段 `TimestampExtractor`。因此, 实现 `StreamTableSource` 和 `BatchTableSource` 并定义 **rowtime** 属性的 `TableSource` 为流查询和批处理查询提供了完全相同的数据。

提供的时间戳提取器

Flink 提供 `TimestampExtractor` 了常见用例的实现。

`TimestampExtractor` 当前提供以下实现:

- `ExistingField(fieldName)`: 从现有的 **LONG**, **SQL_TIMESTAMP** 或时间戳格式的 **STRING** 字段中提取 **rowtime** 属性的值。 这样的字符串的一个示例是“2018-05-28 12:34:56.000”。
- `StreamRecordTimestamp()`: 从 `DataStream StreamRecord` 的时间戳中提取 **rowtime** 属性的值。注意, 这 `TimestampExtractor` 不适用于批处理表 `source`。

`TimestampExtractor` 可以通过实现相应的接口来定义自定义。

提供的水印策略

Flink 提供 `WatermarkStrategy` 了常见用例的实现。

`WatermarkStrategy` 当前提供以下实现:

- `AscendingTimestamps`: 提升时间戳的水印策略。 时间戳不正确的记录将被视为较晚。

- `BoundedOutOfOrderTimestamps(delay)`: 用于时间戳的水印策略, 该时间戳最多按指定的延迟乱序。
- `PreserveWatermarks()`: 指示应从基础 `DataStream` 中保留水印的策略。

`WatermarkStrategy` 可以通过实现相应的接口来定义自定义。

使用投影下推定义 **TableSource**

`TableSource` 通过实现 `ProjectableTableSource` 接口来支持投影下推。该接口定义了一个方法:

```
ProjectableTableSource[T] {  
  
    def projectFields(fields: Array[Int]): TableSource[T]  
}
```

- `projectFields(fields)`: 返回具有调整后的物理返回类型的 `TableSource` 的副本。`fields` 参数提供 `TableSource` 必须提供的字段的索引。索引与物理返回类型的 `TypeInformation` 有关, 而不与逻辑表模式有关。复制的 `TableSource` 必须调整其返回类型以及返回的 `DataStream` 或 `DataSet`。复制的 `TableSource` 的 `TableSchema` 不得更改, 即它必须与原始 `TableSource` 相同。如果 `TableSource` 实现了 `DefinedFieldMapping` 接口, 则必须将字段映射调整为新的返回类型。

注意为了使 Flink 可以将投影下推表 `source` 与其原始形式区分开, 必须重写 `explainSource` 方法以包括有关投影字段的信息。

`ProjectableTableSource` 增加了对项目平面字段的支持。如果 `TableSource` 定义了具有嵌套模式的表, 则可以实现 `NestedFieldsProjectableTableSource` 以将投影扩展到嵌套字段。`NestedFieldsProjectableTableSource` 的定义如下:

```
NestedFieldsProjectableTableSource[T] {  
  
    def projectNestedFields(fields: Array[Int], nestedFields:  
Array[Array[String]]): TableSource[T]  
}
```

- `projectNestedField(fields, nestedFields)`: 返回具有调整后的物理返回类型的 `TableSource` 的副本。物理返回类型的字段可以删除或重新排序, 但不得更改其类型。此方法的约定与 `ProjectableTableSource.projectFields()` 方法的约定基本相同。另外, `nestedFields` 参数包含字段列表中每个字段索引的查询到的所有嵌套字段的路径列表。所有其他嵌套字段都不需要在 `TableSource` 生成的记录中读取, 解析和设置。

请注意, 不得更改投影字段的类型, 但未使用的字段可以设置为 `null` 或默认值。

使用过滤器下推定义 **TableSource**

FilterableTableSource 接口添加了对将过滤器下推到 **TableSource** 的支持。扩展此接口的 **TableSource** 能够过滤记录，以便返回的 **DataStream** 或 **DataSet** 返回较少的记录。

该接口如下所示：

```
FilterableTableSource[T] {  
  
    def applyPredicate(predicates: java.util.List[Expression]):  
TableSource[T]  
  
    def isFilterPushedDown: Boolean  
}
```

- `applyPredicate(predicates)`：返回带有添加谓词的 **TableSource** 的副本。谓词参数是“提供”给 **TableSource** 的连接谓词的可变列表。**TableSource** 接受通过从列表中删除谓词来评估谓词。列表中剩余的谓词将由后续的过滤器运算符评估。
- `isFilterPushedDown()`：如果之前调用 `applyPredicate()` 方法，则返回 `true`。因此，对于从 `applyPredicate()` 调用返回的所有 **TableSource** 实例，`isFilterPushedDown()` 必须返回 `true`。



注意：为了使 **Flink** 能够将过滤器下推表源与其原始形式区分开来，必须重写 `explainSource` 方法以包括有关下推式过滤器的信息。

定义用于查找的 **TableSource**

注意这是一项实验功能。将来的版本中可能会更改 接口。仅 **Blink planner** 支持。

LookupableTableSource 接口增加了对通过查找方式通过键列访问表的支持。当用于与维表联接以丰富某些信息时，这非常有用。如果要在查找模式下使用 **TableSource**，则应在[动态表联接语法](#)中使用源。

该接口如下所示：

```
LookupableTableSource[T] extends TableSource[T] {  
  
    def getLookupFunction(lookupKeys: Array[String]): TableFunction[T]  
  
    def getAsyncLookupFunction(lookupKeys: Array[String]):  
AsyncTableFunction[T]  
  
    def isAsyncEnabled: Boolean  
}  

```

- `getLookupFunction(lookupkeys)`：返回一个 **TableFunction**，该函数用于通过查找键查找匹配的行。`lookupkeys` 是联接相等条件下 **LookupableTableSource** 的字段名称。返回的 **TableFunction** 的 `eval` 方法参数应该按照定义的 `lookupkeys` 的顺序。

建议在 `varargs` 中定义参数（例如 `eval (Object ... lookupkeys)`）以匹配所有情况。`TableFunction` 的返回类型必须与 `TableSource.getReturnType ()` 方法定义的返回类型相同。

- `getAsyncLookupFunction(lookupkeys):` 可选的。与 `getLookupFunction` 类似，但是 `AsyncLookupFunction` 异步查找匹配的行。`AsyncLookupFunction` 的基础将通过 `Async I / O` 调用。返回的 `AsyncTableFunction` 的 `eval` 方法的第一个参数应定义为 `java.util.concurrent.CompletableFuture` 以异步收集结果（例如 `eval (CompletableFuture <Collection <String >> result, Object ... lookupkeys)`）。如果 `TableSource` 不支持异步查找，则此方法的实现可能引发异常。
- `isAsyncEnabled():` 如果启用了异步查找，则返回 `true`。如果 `isAsyncEnabled` 返回 `true`，则需要实现 `getAsyncLookupFunction (lookupkeys)`。

定义 Table Sink

`TableSink` 指定如何将表发送到外部系统或位置。该接口是通用的，因此它可以支持不同的存储位置和格式。批处理表和流式表有不同的表接收器。

接口如下所示：

```
TableSink[T] {  
  
    def getOutputType: TypeInformation<T>  
  
    def getFieldNames: Array[String]  
  
    def getFieldTypes: Array[TypeInformation]  
  
    def configure(fieldNames: Array[String], fieldTypes:  
Array[TypeInformation]): TableSink[T]  
}
```

`TableSink#configure` 调用该方法可将 `Table` 的结构（字段名称和类型）传递给 `TableSink`。该方法必须返回 `TableSink` 的新实例，该实例被配置为发出提供的 `Table` 模式。

BatchTableSink

定义外部 `TableSink` 来发出批处理表。

该接口如下所示：

```
BatchTableSink[T] extends TableSink[T] {  
  
    def emitDataSet(dataSet: DataSet[T]): Unit  
}
```

AppendStreamTableSink

定义一个外部 **TableSink** 以发出一个批处理表。

该接口如下所示：

```
AppendStreamTableSink[T] extends TableSink[T] {  
  
    def emitDataStream(dataStream: DataStream[T]): Unit  
}
```

如果还通过更新或删除更改来修改表，则将引发 **TableException**。

RetractStreamTableSink

定义一个外部 **TableSink** 以发出具有插入，更新和删除更改的流表。

该接口如下所示：



```
RetractStreamTableSink[T] extends TableSink[Tuple2[Boolean, T]] {  
  
    def getRecordType: TypeInformation[T]  
  
    def emitDataStream(dataStream: DataStream[Tuple2[Boolean, T]]):  
Unit  
}
```

该表将被转换为累积和撤消消息流，这些消息被编码为 **Java Tuple2**。第一个字段是指示消息类型的布尔标志（**true** 表示插入，**false** 表示删除）。第二个字段保存请求的类型 **T** 的记录。

UpsertStreamTableSink

定义一个外部 **TableSink** 以发出具有插入，更新和删除更改的流表。

该接口如下所示：

```
UpsertStreamTableSink[T] extends TableSink[Tuple2[Boolean, T]] {  
  
    def setKeyFields(keys: Array[String]): Unit  
  
    def setIsAppendOnly(isAppendOnly: Boolean): Unit  
  
    def getRecordType: TypeInformation[T]  
  
    def emitDataStream(dataStream: DataStream[Tuple2[Boolean, T]]):  
Unit  
}  

```

该表必须具有唯一的键字段（原子键或复合键）或仅附加键。如果表没有唯一键并且不是仅追加表，则将引发 `TableException`。该表的唯一键由 `UpsertStreamTableSink#setKeyFields()` 方法配置。

该表将转换为 `upsert` 和 `delete` 消息流，这些消息被编码为 `Java Tuple2`。第一个字段是指示消息类型的布尔标志。第二个字段保存请求的类型 `T` 的记录。

具有 `true` 布尔值字段的消息是已配置密钥的 `upsert` 消息。带有 `false` 标志的消息是已配置密钥的删除消息。如果表是仅追加的，则所有消息都将具有 `true` 标志，并且必须将其解释为插入。

定义一个 `TableFactory`

`TableFactory` 允许从基于字符串的属性中创建与表相关的不同实例。调用所有可用的工厂以匹配给定的属性集和相应的工厂类。

工厂利用 `Java` 的服务提供商接口（`SPI`）进行发现。这意味着每个依赖项和 `JAR` 文件都应在 `META-INF / services` 资源目录中包含一个文件

`org.apache.flink.table.factories.TableFactory`，该文件列出了它提供的所有可用表工厂。

每个表工厂都需要实现以下接口：

```
package org.apache.flink.table.factories;

interface TableFactory {

    Map<String, String> requiredContext();

    List<String> supportedProperties();
}
```

- `requiredContext()`：指定已为此工厂实现的上下文。该框架保证仅在满足指定的属性和值集的情况下才与此工厂匹配。典型的属性可能是 `connector.type`，`format.type` 或 `update-mode`。为将来的向后兼容情况保留了诸如 `connect.property-version` 和 `format.property-version` 之类的属性键。
- `supportedProperties()`：此工厂可以处理的属性键的列表。此方法将用于验证。如果传递了该工厂无法处理的属性，则将引发异常。该列表不得包含上下文指定的键。

为了创建特定实例，工厂类可以实现一个或多个接口，该接口提供

`org.apache.flink.table.factories`：

- `BatchTableSourceFactory`：创建一个批处理表源。
- `BatchTableSinkFactory`：创建一个批处理表接收器。
- `StreamTableSourceFactory`：创建流表源。
- `StreamTableSinkFactory`：创建一个流表接收器。

- `DeserializationSchemaFactory`: 创建反序列化架构格式。
- `SerializationSchemaFactory`: 创建序列化架构格式。

工厂的发现分为多个阶段：

- 发现所有可用的工厂。
- 按工厂类别（例如 `StreamTableSourceFactory`）过滤。
- 通过匹配上下文进行过滤。
- 按支持的属性过滤。
- 验证一个工厂是否完全匹配，否则抛出 `AmbiguousTableFactoryException` 或 `NoMatchingTableFactoryException`。

下面的示例演示如何为自定义流源提供附加的 `connector.debug` 属性标志以进行参数化。

```
import java.util
import org.apache.flink.table.sources.StreamTableSource
import org.apache.flink.types.Row

class MySystemTableSourceFactory extends
StreamTableSourceFactory[Row] {

    override def requiredContext(): util.Map[String, String] = {
        val context = new util.HashMap[String, String]()
        context.put("update-mode", "append")
        context.put("connector.type", "my-system")
        context
    }

    override def supportedProperties(): util.List[String] = {
        val properties = new util.ArrayList[String]()
        properties.add("connector.debug")
        properties
    }

    override def createStreamTableSource(properties: util.Map[String,
String]): StreamTableSource[Row] = {
        val isDebug =
java.lang.Boolean.valueOf(properties.get("connector.debug"))

        # additional validation of the passed properties can also happen
here

        new MySystemAppendTableSource(isDebug)
    }
}
```

```
}
```



在 SQL 客户端中使用 TableFactory

在 SQL Client 环境文件中，先前提供的工厂可以声明为：



```
tables:
  - name: MySystemTable
    type: source
    update-mode: append
    connector:
      type: my-system
      debug: true
```



将 YAML 文件转换为扁平化的字符串属性，并使用描述与外部系统的连接的那些属性来调用表工厂：

```
update-mode=append
connector.type=my-system
connector.debug=true
```

注意：table.#.name 或 table.#.type 之类的属性是 SQL Client 的特定属性，不会传递给任何工厂。根据执行环境，type 属性决定是否需要发现 BatchTableSourceFactory / StreamTableSourceFactory（对于 source），BatchTableSinkFactory / StreamTableSinkFactory（对于 sink）还是同时发现两者（对于两者）。

在 Table&SQL API 中使用 TableFactory

对于使用说明性 Scaladoc / Javadoc 的类型安全的编程方法，Table&SQL API 在 org.apache.flink.table.descriptor 中提供了描述符，这些描述符可转换为基于字符串的属性。请参阅源，接收器和格式的内置描述符作为参考。



```
import org.apache.flink.table.descriptors.ConnectorDescriptor
import java.util.HashMap
import java.util.Map

/**
 * Connector to MySystem with debug mode.
 */
class MySystemConnector(isDebug: Boolean) extends
ConnectorDescriptor("my-system", 1, false) {

  override protected def toConnectorProperties(): Map[String, String]
= {
    val properties = new HashMap[String, String]
```

```
properties.put("connector.debug", isDebug.toString)
properties
}
}
```

然后可以在 API 中使用描述符，如下所示：

```
val tableEnv: StreamTableEnvironment = // ...

tableEnv
  .connect(new MySystemConnector(isDebug = true))
  .withSchema(...)
  .inAppendMode()
  .createTemporaryTable("MySystemTable")
```

【翻译】Flink Table Api & SQL — 用户定义函数

本文翻译自官网：User-defined Functions <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/udfs.html>

Flink Table Api & SQL 翻译目录

用户定义函数是一项重要功能，因为它们显著扩展了查询的表达能力。

- [注册用户定义的函数](#)
- [标量函数](#)
- [表函数](#)
- [聚合函数](#)
- [表格汇总函数](#)
- [实施 UDF 的最佳做法](#)
- [将 UDF 与运行时集成](#)

注册用户定义的函数

在大多数情况下，必须先注册用户定义的函数，然后才能在查询中使用该函数。无需注册 Scala Table API 的函数。

通过调用 `registerFunction()` 方法在 `TableEnvironment` 中注册函数。注册用户定义的函数后，会将其插入 `TableEnvironment` 的函数目录中，以便 Table API 或 SQL 解析器可以识别并正确转换它。


请在以下子会话中找到有关如何注册以及如何调用每种类型的用户定义函数（`ScalarFunction`、`TableFunction` 和 `AggregateFunction`）的详细示例。

标量函数

如果内置函数中未包含所需的标量函数，则可以为 **Table API** 和 **SQL** 定义自定义的，用户定义的标量函数。用户定义的标量函数将零个、一个或多个标量值映射到新的标量值。


为了定义标量函数，必须扩展 `org.apache.flink.table.functions` 中的基类 `ScalarFunction` 并实现（一个或多个）评估方法。标量函数的行为由评估方法确定。评估方法必须公开声明并命名为 `eval`。评估方法的参数类型和返回类型也确定标量函数的参数和返回类型。评估方法也可以通过实现多种名为 `eval` 的方法来重载。评估方法还可以支持可变参数，例如 `eval (String ... str)`。

下面的示例演示如何定义自己的哈希码函数，如何在 `TableEnvironment` 中注册并在查询中调用它。请注意，您可以在构造函数之前注册它的标量函数：

```

// must be defined in static/object context
class HashCode(factor: Int) extends ScalarFunction {
  def eval(s: String): Int = {
    s.hashCode() * factor
  }
}


val tableEnv = BatchTableEnvironment.create(env)

// use the function in Scala Table API
val hashCode = new HashCode(10)
myTable.select('string, hashCode('string))

// register and use the function in SQL
tableEnv.registerFunction("hashCode", new HashCode(10))
tableEnv.sqlQuery("SELECT string, hashCode(string) FROM MyTable")

```

默认情况下，评估方法的结果类型由 Flink 的类型提取工具确定。这对于基本类型或简单的 POJO 就足够了，但对于更复杂，自定义或复合类型可能是错误的。在这些情况下，可以通过覆盖 `ScalarFunction#getResultType()` 来手动定义结果类型的 `TypeInformation`。

下面的示例显示一个高级示例，该示例采用内部时间戳表示，并且还以长值形式返回内部时间戳表示。通过重写 `ScalarFunction#getResultType()`，我们定义了代码生成应将返回的 `long` 值解释为 `Types.TIMESTAMP`。

```

object TimestampModifier extends ScalarFunction {
  def eval(t: Long): Long = {
    t % 1000
  }
}
```



```

    }

    override def getResultType(signature: Array[Class[_]]):
    TypeInformation[_] = {
        Types.TIMESTAMP
    }
}

```



Table Function

与用户定义的标量函数相似，用户定义的表函数将零，一个或多个标量值作为输入参数。但是，与标量函数相比，它可以返回任意数量的行作为输出，而不是单个值。返回的行可能包含一列或多列。

为了定义表函数，必须扩展 `org.apache.flink.table.functions` 中的基类 `TableFunction` 并实现（一个或多个）评估方法。表函数的行为由其评估方法确定。必须将评估方法声明为公开并命名为 `eval`。通过实现多个名为 `eval` 的方法，可以重载 `TableFunction`。评估方法的参数类型确定表函数的所有有效参数。评估方法还可以支持可变参数，例如 `eval (String ... strs)`。返回表的类型由 `TableFunction` 的通用类型确定。评估方法使用受保护的 `collect (T)` 方法发出输出行。

在 Table API 中，表函数与 `.joinLateral` 或 `.leftOuterJoinLateral` 一起使用。`joinLateral` 运算符（叉号）将外部表（运算符左侧的表）中的每一行与表值函数（位于运算符的右侧）产生的所有行进行连接。`leftOuterJoinLateral` 运算符将外部表（运算符左侧的表）中的每一行与表值函数（位于运算符的右侧）产生的所有行连接起来，并保留表函数返回的外部行 一个空桌子。在 SQL 中，使用带有 `CROSS JOIN` 和 `LEFT JOIN` 且带有 `ON TRUE` 连接条件的 `LATERAL TABLE (<TableFunction>)`（请参见下面的示例）。

下面的示例演示如何定义表值函数，如何在 `TableEnvironment` 中注册表值函数以及如何在查询中调用它。请注意，可以在注册表函数之前通过构造函数对其进行配置：



```

// The generic type "(String, Int)" determines the schema of the
// returned table as (String, Integer).
class Split(separator: String) extends TableFunction[(String, Int)] {
    def eval(str: String): Unit = {
        // use collect(...) to emit a row.
        str.split(separator).foreach(x => collect((x, x.length)))
    }
}

val tableEnv = BatchTableEnvironment.create(env)
val myTable = ...           // table schema: [a: String]

```

```
// Use the table function in the Scala Table API (Note: No
registration required in Scala Table API).
val split = new Split("#")
// "as" specifies the field names of the generated table.
myTable.joinLateral(split('a) as ('word, 'length)).select('a, 'word,
'length)
myTable.leftOuterJoinLateral(split('a) as ('word,
'length)).select('a, 'word, 'length)

// Register the table function to use it in SQL queries.
tableEnv.registerFunction("split", new Split("#"))

// Use the table function in SQL with LATERAL and TABLE keywords.
// CROSS JOIN a table function (equivalent to "join" in Table API)
tableEnv.sqlQuery("SELECT a, word, length FROM MyTable, LATERAL
TABLE(split(a)) as T(word, length)")
// LEFT JOIN a table function (equivalent to "leftOuterJoin" in Table
API)
tableEnv.sqlQuery("SELECT a, word, length FROM MyTable LEFT JOIN
LATERAL TABLE(split(a)) as T(word, length) ON TRUE")
```



重要说明：不要将 `TableFunction` 实现为 `Scala` 对象。`Scala` 对象是单例对象，将导致并发问题。

请注意，`POJO` 类型没有确定的字段顺序。因此，您不能使用 `AS` 来重命名表函数返回的 `POJO` 字段。

默认情况下，`TableFunction` 的结果类型由 `Flink` 的自动类型提取工具确定。这对于基本类型和简单的 `POJO` 非常有效，但是对于更复杂，自定义或复合类型可能是错误的。在这种情况下，可以通过重写 `TableFunction#getResultType()` 并返回其 `TypeInformation` 来手动指定结果的类型。

下面的示例显示一个 `TableFunction` 的示例，该函数返回需要显式类型信息的 `Row` 类型。我们通过重写 `TableFunction#getResultType()` 来定义返回的表类型应为 `RowTypeInfo(String, Integer)`。



```
class CustomTypeSplit extends TableFunction[Row] {
  def eval(str: String): Unit = {
    str.split(" ").foreach({ s =>
      val row = new Row(2)
      row.setField(0, s)
      row.setField(1, s.length)
      collect(row)
    })
  }
}
```

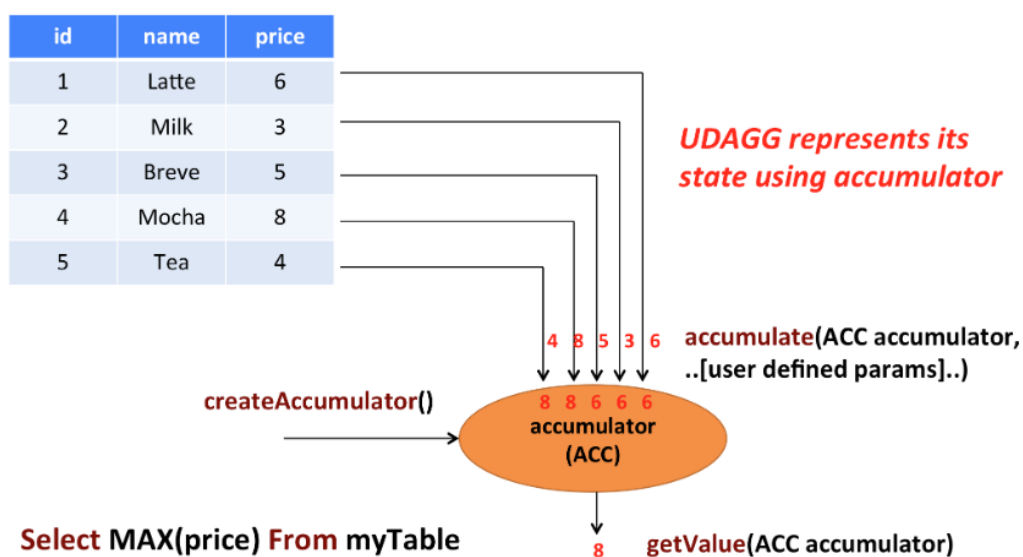
```

override def getResultType: TypeInformation[Row] = {
    Types.ROW(Types.STRING, Types.INT)
}
}

```

聚合函数

用户定义的聚合函数（UDAGG）将表（具有一个或多个属性的一个或多个行）聚合到一个标量值。



上图显示了聚合的示例。假设您有一个包含饮料数据的表。该表由三列组成，即 ID，name 和 price 以及 5 行。假设您需要在表格中找到所有饮料中最高的 price，即执行 max（）汇总。您将需要检查 5 行中的每行，结果将是单个数字值。

用户定义的聚合函数通过扩展 `AggregateFunction` 类来实现。`AggregateFunction` 的工作原理如下。首先，它需要一个累加器，它是保存聚合中间结果的数据结构。通过调用 `AggregateFunction` 的 `createAccumulator()` 方法来创建一个空的累加器。随后，为每个输入行调用该函数的 `accumulate()` 方法以更新累加器。处理完所有行后，将调用该函数的 `getValue()` 方法以计算并返回最终结果。

每种方法都必须使用以下方法 `AggregateFunction`:

- `createAccumulator()`
- `accumulate()`
- `getValue()`

Flink 的类型提取工具可能无法识别复杂的数据类型，例如，如果它们不是基本类型或简单的 POJO。因此，类似于 `ScalarFunction` 和 `TableFunction`，`AggregateFunction` 提供了一些方法来指定结果类型的 `TypeInformation`（通过 `AggregateFunction#getResultType()`）和累加器的类型（通过 `AggregateFunction#getAccumulatorType()`）。


除上述方法外，还有一些可选择性实现的约定方法。尽管这些方法中的某些方法使系统可以更有效地执行查询，但对于某些用例，其他方法是必需的。例如，如果聚合功能应在会话组窗口的上下文中应用，则必须使用 `merge()` 方法（观察到“连接”它们的行时，两个会话窗口的累加器必须合并）。

`AggregateFunction` 根据使用情况，需要以下方法：

- `retract()` 在有界 OVER 窗口上进行聚合是必需的。
- `merge()` 许多批处理聚合和会话窗口聚合是必需的。
- `resetAccumulator()` 许多批处理聚合是必需的。

必须将 `AggregateFunction` 的所有方法声明为 `public`，而不是静态的，并且必须完全按上述名称命名。方法 `createAccumulator`，`getValue`，`getResultType` 和 `getAccumulatorType` 在 `AggregateFunction` 抽象类中定义，而其他方法则是协定方法。为了定义聚合函数，必须扩展基类 `org.apache.flink.table.functions.AggregateFunction` 并实现一个（或多个）累积方法。累加的方法可以重载不同的参数类型，并支持可变参数。

下面给出了 `AggregateFunction` 的所有方法的详细文档。

```

/**
 * Base class for user-defined aggregates and table aggregates.
 *
 * @tparam T the type of the aggregation result.
 * @tparam ACC the type of the aggregation accumulator. The
accumulator is used to keep the
 * aggregated values which are needed to compute an
aggregation result.
 */
abstract class UserDefinedAggregateFunction[T, ACC] extends
UserDefinedFunction {

    /**
     * Creates and init the Accumulator for this (table)aggregate
function.
     *
     * @return the accumulator with the initial value
     */
    def createAccumulator(): ACC // MANDATORY

    /**
```

```

    * Returns the TypeInformation of the (table)aggregate function's
result.
    *
    * @return The TypeInformation of the (table)aggregate function's
result or null if the result
    *         type should be automatically inferred.
    */
    def getResultType: TypeInformation[T] = null // PRE-DEFINED

/**
    * Returns the TypeInformation of the (table)aggregate function's
accumulator.
    *
    * @return The TypeInformation of the (table)aggregate function's
accumulator or null if the
    *         accumulator type should be automatically inferred.
    */
    def getAccumulatorType: TypeInformation[ACC] = null // PRE-DEFINED
}

/**
    * Base class for aggregation functions.
    *
    * @tparam T    the type of the aggregation result
    * @tparam ACC the type of the aggregation accumulator. The
accumulator is used to keep the
    *         aggregated values which are needed to compute an
aggregation result.
    *         AggregateFunction represents its state using
accumulator, thereby the state of the
    *         AggregateFunction must be put into the accumulator.
    */
abstract class AggregateFunction[T, ACC] extends
UserDefinedAggregateFunction[T, ACC] {

    /**
    * Processes the input values and update the provided accumulator
instance. The method
    * accumulate can be overloaded with different custom types and
arguments. An AggregateFunction
    * requires at least one accumulate() method.
    *
    * @param accumulator the accumulator which contains the
current aggregated results

```

```

    * @param [user defined inputs] the input value (usually obtained
from a new arrived data).
    */
    def accumulate(accumulator: ACC, [user defined inputs]): Unit //
MANDATORY

    /**
    * Retracts the input values from the accumulator instance. The
current design assumes the
    * inputs are the values that have been previously accumulated.
The method retract can be
    * overloaded with different custom types and arguments. This
function must be implemented for
    * datastream bounded over aggregate.
    *
    * @param accumulator          the accumulator which contains the
current aggregated results
    * @param [user defined inputs] the input value (usually obtained
from a new arrived data).
    */
    def retract(accumulator: ACC, [user defined inputs]): Unit //
OPTIONAL

    /**
    * Merges a group of accumulator instances into one accumulator
instance. This function must be
    * implemented for datastream session window grouping aggregate
and dataset grouping aggregate.
    *
    * @param accumulator  the accumulator which will keep the merged
aggregate results. It should
    *                      be noted that the accumulator may contain
the previous aggregated
    *                      results. Therefore user should not replace
or clean this instance in the
    *                      custom merge method.
    * @param its           an [[java.lang.Iterable]] pointed to a
group of accumulators that will be
    *                      merged.
    */
    def merge(accumulator: ACC, its: java.lang.Iterable[ACC]): Unit //
OPTIONAL

    /**

```

```

    * Called every time when an aggregation result should be
materialized.
    * The returned value could be either an early and incomplete
result
    * (periodically emitted as data arrive) or the final result of
the
    * aggregation.
    *
    * @param accumulator the accumulator which contains the current
    * aggregated results
    * @return the aggregation result
    */
def getValue(accumulator: ACC): T // MANDATORY

/**
    * Resets the accumulator for this [[AggregateFunction]]. This
function must be implemented for
    * dataset grouping aggregate.
    *
    * @param accumulator the accumulator which needs to be reset
    */
def resetAccumulator(accumulator: ACC): Unit // OPTIONAL

/**
    * Returns true if this AggregateFunction can only be applied in
an OVER window.
    *
    * @return true if the AggregateFunction requires an OVER window,
false otherwise.
    */
def requiresOver: Boolean = false // PRE-DEFINED
}

```

以下示例显示了怎么使用

- 定义一个 `AggregateFunction` 计算给定列上的加权平均值
- 在 `TableEnvironment` 中注册函数
- 在查询中使用该函数。

为了计算加权平均值，累加器需要存储所有累加数据的加权和和计数。 在我们的示例中，我们将一个 `WeightedAvgAccum` 类定义为累加器。 累加器由 Flink 的检查点机制自动备份，并在无法确保一次准确语义的情况下恢复。

我们的 `WeightedAvg AggregateFunction` 的 `accumulate()` 方法具有三个输入。 第一个是 `WeightedAvgAccum` 累加器，其他两个是用户定义的输入：输入值 `ivalue` 和输入

iweight 的权重。 尽管大多数聚合类型都不强制使用 `retract ()`，`merge ()` 和 `resetAccumulator ()` 方法，但我们在下面提供了它们作为示例。 请注意，我们在 `Scala` 示例中使用了 `Java` 基本类型并定义了 `getResultType ()` 和 `getAccumulatorType ()` 方法，因为 `Flink` 类型提取不适用于 `Scala` 类型。

```
import java.lang.{Long => JLong, Integer => JInteger}
import org.apache.flink.api.java.tuple.{Tuple1 => JTuple1}
import org.apache.flink.api.java.typeutils.TupleTypeInfo
import org.apache.flink.table.api.Types
import org.apache.flink.table.functions.AggregateFunction

/**
 * Accumulator for WeightedAvg.
 */
class WeightedAvgAccum extends JTuple1[JLong, JInteger] {
  sum = 0L
  count = 0
}

/**
 * Weighted Average user-defined aggregate function.
 */
class WeightedAvg extends AggregateFunction[JLong, CountAccumulator]
{

  override def createAccumulator(): WeightedAvgAccum = {
    new WeightedAvgAccum
  }

  override def getValue(acc: WeightedAvgAccum): JLong = {
    if (acc.count == 0) {
      null
    } else {
      acc.sum / acc.count
    }
  }

  def accumulate(acc: WeightedAvgAccum, iValue: JLong, iWeight:
JInteger): Unit = {
    acc.sum += iValue * iWeight
    acc.count += iWeight
  }
}
```



```

def retract(acc: WeightedAvgAccum, iValue: JLong, iWeight:
JInteger): Unit = {
    acc.sum -= iValue * iWeight
    acc.count -= iWeight
}

def merge(acc: WeightedAvgAccum, it:
java.lang.Iterable[WeightedAvgAccum]): Unit = {
    val iter = it.iterator()
    while (iter.hasNext) {
        val a = iter.next()
        acc.count += a.count
        acc.sum += a.sum
    }
}

def resetAccumulator(acc: WeightedAvgAccum): Unit = {
    acc.count = 0
    acc.sum = 0L
}

override def getAccumulatorType: TypeInformation[WeightedAvgAccum]
= {
    new TupleTypeInfo(classOf[WeightedAvgAccum], Types.LONG,
Types.INT)
}

override def getResultType: TypeInformation[JLong] = Types.LONG
}

// register function
val tEnv: StreamTableEnvironment = ???
tEnv.registerFunction("wAvg", new WeightedAvg())

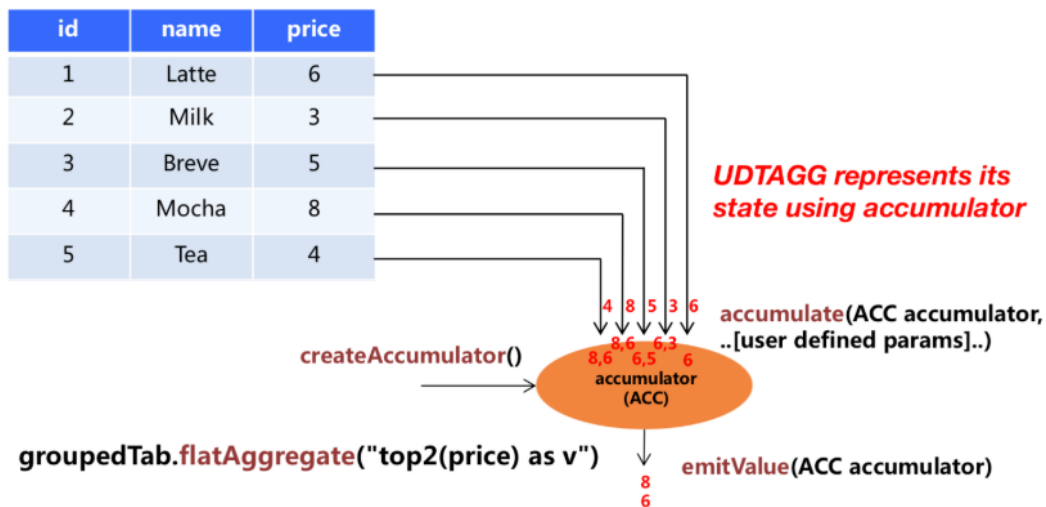
// use function
tEnv.sqlQuery("SELECT user, wAvg(points, level) AS avgPoints FROM
userScores GROUP BY user")

```



表聚合函数

用户定义的表聚合函数（UDTAGG）将一个表（具有一个或多个属性的一个或多个行）聚合到具有多行和多列的结果表中。



上图显示了表聚合的示例。假设您有一个包含饮料数据的表。该表由三列组成，即 ID，name 和 price 以及 5 行。假设您需要在表格中找到所有饮料中 price 最高的前 2 个，即执行 top2() 表汇总。您将需要检查 5 行中的每行，结果将是带有前 2 个值的表。

用户定义的表聚合功能通过扩展 TableAggregateFunction 类来实现。

TableAggregateFunction 的工作原理如下。首先，它需要一个累加器，它是保存聚合中间结果的数据结构。通过调用 TableAggregateFunction 的 createAccumulator() 方法来创建一个空的累加器。随后，为每个输入行调用该函数的 accumulate() 方法以更新累加器。处理完所有行后，将调用该函数的 emitValue() 方法来计算并返回最终结果。

每种方法都必须使用以下方法 TableAggregateFunction:

- createAccumulator()
- accumulate()

Flink 的类型提取工具可能无法识别复杂的数据类型，例如，如果它们不是基本类型或简单的 POJO。因此，类似于 ScalarFunction 和 TableFunction，TableAggregateFunction 提供了一些方法来指定结果类型的 TypeInformation（通过 TableAggregateFunction # getResultType()）和累加器的类型（通过 TableAggregateFunction # getAccumulatorType()）。

除上述方法外，还有一些可选择性实现的约定方法。尽管这些方法中的某些方法使系统可以更有效地执行查询，但对于某些用例，其他方法是必需的。例如，如果聚合功能应在会话组窗口的上下文中应用，则必须使用 merge() 方法（观察到“连接”它们的行时，两个会话窗口的累加器必须合并）。

TableAggregateFunction 根据使用情况，需要以下方法:

- retract() 在有界 OVER 窗口上进行聚合是必需的。
- merge() 许多批处理聚合和会话窗口聚合是必需的。
- resetAccumulator() 许多批处理聚合是必需的。

- `emitValue()` 是批处理和窗口聚合所必需的。

`TableAggregateFunction` 的以下方法用于提高流作业的性能:

- `emitUpdateWithRetract()` 用于发出在撤回模式下已更新的值。

对于 `emitValue` 方法, 它根据累加器发出完整的数据。以 `TopN` 为例, `emitValue` 每次都会发出所有前 `n` 个值。这可能会给流作业带来性能问题。为了提高性能, 用户还可以实现 `emitUpdateWithRetract` 方法来提高性能。该方法以撤回模式增量输出数据, 即, 一旦有更新, 我们必须先撤回旧记录, 然后再发送新的更新记录。如果所有方法都在表聚合函数中定义, 则该方法将优先于 `emitValue` 方法使用, 因为 `emitUpdateWithRetract` 被认为比 `emitValue` 更有效, 因为它可以增量输出值。

必须将 `TableAggregateFunction` 的所有方法声明为 `public`, 而不是静态的, 并且其命名必须与上述名称完全相同。方法 `createAccumulator`, `getResultType` 和 `getAccumulatorType` 在 `TableAggregateFunction` 的父抽象类中定义, 而其他方法则是契约方法。为了定义表聚合函数, 必须扩展基类 `org.apache.flink.table.functions.TableAggregateFunction` 并实现一个 (或多个) 累积方法。累加的方法可以重载不同的参数类型, 并支持可变参数。

下面给出了 `TableAggregateFunction` 的所有方法的详细文档。

```
/**
 * Base class for user-defined aggregates and table aggregates.
 *
 * @tparam T    the type of the aggregation result.
 * @tparam ACC the type of the aggregation accumulator. The
accumulator is used to keep the
 *             aggregated values which are needed to compute an
aggregation result.
 */
abstract class UserDefinedAggregateFunction[T, ACC] extends
UserDefinedFunction {

    /**
     * Creates and init the Accumulator for this (table)aggregate
function.
     *
     * @return the accumulator with the initial value
     */
    def createAccumulator(): ACC // MANDATORY

    /**
     * Returns the TypeInformation of the (table)aggregate function's
result.
     */
}
```

```

    *
    * @return The TypeInformation of the (table)aggregate function's
result or null if the result
    *         type should be automatically inferred.
    */
    def getResultType: TypeInformation[T] = null // PRE-DEFINED

    /**
    * Returns the TypeInformation of the (table)aggregate function's
accumulator.
    *
    * @return The TypeInformation of the (table)aggregate function's
accumulator or null if the
    *         accumulator type should be automatically inferred.
    */
    def getAccumulatorType: TypeInformation[ACC] = null // PRE-DEFINED
}

/**
* Base class for table aggregation functions.
*
* @tparam T    the type of the aggregation result
* @tparam ACC the type of the aggregation accumulator. The
accumulator is used to keep the
*         aggregated values which are needed to compute an
aggregation result.
*         TableAggregateFunction represents its state using
accumulator, thereby the state of
*         the TableAggregateFunction must be put into the
accumulator.
*/
abstract class TableAggregateFunction[T, ACC] extends
UserDefinedAggregateFunction[T, ACC] {

    /**
    * Processes the input values and update the provided accumulator
instance. The method
    * accumulate can be overloaded with different custom types and
arguments. A TableAggregateFunction
    * requires at least one accumulate() method.
    *
    * @param accumulator    the accumulator which contains the
current aggregated results

```

```

    * @param [user defined inputs] the input value (usually obtained
from a new arrived data).
    */
    def accumulate(accumulator: ACC, [user defined inputs]): Unit //
MANDATORY

    /**
    * Retracts the input values from the accumulator instance. The
current design assumes the
    * inputs are the values that have been previously accumulated.
The method retract can be
    * overloaded with different custom types and arguments. This
function must be implemented for
    * datastream bounded over aggregate.
    *
    * @param accumulator          the accumulator which contains the
current aggregated results
    * @param [user defined inputs] the input value (usually obtained
from a new arrived data).
    */
    def retract(accumulator: ACC, [user defined inputs]): Unit //
OPTIONAL

    /**
    * Merges a group of accumulator instances into one accumulator
instance. This function must be
    * implemented for datastream session window grouping aggregate
and dataset grouping aggregate.
    *
    * @param accumulator  the accumulator which will keep the merged
aggregate results. It should
    *                      be noted that the accumulator may contain
the previous aggregated
    *                      results. Therefore user should not replace
or clean this instance in the
    *                      custom merge method.
    * @param its           an [[java.lang.Iterable]] pointed to a
group of accumulators that will be
    *                      merged.
    */
    def merge(accumulator: ACC, its: java.lang.Iterable[ACC]): Unit //
OPTIONAL

    /**

```

```

    * Called every time when an aggregation result should be
materialized. The returned value
    * could be either an early and incomplete result (periodically
emitted as data arrive) or
    * the final result of the aggregation.
    *
    * @param accumulator the accumulator which contains the current
    * aggregated results
    * @param out the collector used to output data
    */
def emitValue(accumulator: ACC, out: Collector[T]): Unit //
OPTIONAL

/**
    * Called every time when an aggregation result should be
materialized. The returned value
    * could be either an early and incomplete result (periodically
emitted as data arrive) or
    * the final result of the aggregation.
    *
    * Different from emitValue, emitUpdateWithRetract is used to emit
values that have been updated.
    * This method outputs data incrementally in retract mode, i.e.,
once there is an update, we
    * have to retract old records before sending new updated ones.
The emitUpdateWithRetract
    * method will be used in preference to the emitValue method if
both methods are defined in the
    * table aggregate function, because the method is treated to be
more efficient than emitValue
    * as it can output values incrementally.
    *
    * @param accumulator the accumulator which contains the current
    * aggregated results
    * @param out the retractable collector used to output
data. Use collect method
    * to output(add) records and use retract
method to retract(delete)
    * records.
    */
def emitUpdateWithRetract(accumulator: ACC, out:
RetractableCollector[T]): Unit // OPTIONAL

/**

```

```

    * Collects a record and forwards it. The collector can output
    retract messages with the retract
    * method. Note: only use it in `emitRetractValueIncrementally`.
    */
    trait RetractableCollector[T] extends Collector[T] {

        /**
         * Retract a record.
         *
         * @param record The record to retract.
         */
        def retract(record: T): Unit
    }
}

```

以下示例显示了怎么使用

- 定义一个 `TableAggregateFunction` 用于计算给定列的前 2 个值
- 在 `TableEnvironment` 中注册函数
- 在 `Table API` 查询中使用该函数 (`Table API` 仅支持 `TableAggregateFunction`)。

要计算前 2 个值，累加器需要存储所有已累加数据中的最大 2 个值。 在我们的示例中，我们定义了一个 `Top2Accum` 类作为累加器。 累加器由 `Flink` 的检查点机制自动备份，并在无法确保一次准确语义的情况下恢复。

我们的 `Top2 TableAggregateFunction` 的 `accumulate()` 方法有两个输入。 第一个是 `Top2Accum` 累加器，另一个是用户定义的输入：输入值 `v`。尽管 `merge()` 方法对于大多数表聚合类型不是强制性的，但我们在下面提供了示例。 请注意，我们在 `Scala` 示例中使用了 `Java` 基本类型并定义了 `getResultType()` 和 `getAccumulatorType()` 方法，因为 `Flink` 类型提取不适用于 `Scala` 类型。

```

import java.lang.{Integer => JInteger}
import org.apache.flink.table.api.Types
import org.apache.flink.table.functions.TableAggregateFunction

/**
 * Accumulator for top2.
 */
class Top2Accum {
    var first: JInteger = _
    var second: JInteger = _
}

/**

```

```

    * The top2 user-defined table aggregate function.
    */
class Top2 extends TableAggregateFunction[JTuple2[JInteger,
JInteger], Top2Accum] {

    override def createAccumulator(): Top2Accum = {
        val acc = new Top2Accum
        acc.first = Int.MinValue
        acc.second = Int.MinValue
        acc
    }

    def accumulate(acc: Top2Accum, v: Int) {
        if (v > acc.first) {
            acc.second = acc.first
            acc.first = v
        } else if (v > acc.second) {
            acc.second = v
        }
    }

    def merge(acc: Top2Accum, its: JIterable[Top2Accum]): Unit = {
        val iter = its.iterator()
        while (iter.hasNext) {
            val top2 = iter.next()
            accumulate(acc, top2.first)
            accumulate(acc, top2.second)
        }
    }

    def emitValue(acc: Top2Accum, out: Collector[JTuple2[JInteger,
JInteger]]): Unit = {
        // emit the value and rank
        if (acc.first != Int.MinValue) {
            out.collect(JTuple2.of(acc.first, 1))
        }
        if (acc.second != Int.MinValue) {
            out.collect(JTuple2.of(acc.second, 2))
        }
    }
}

// init table
val tab = ...

```



```
// use function
tab
  .groupBy('key')
  .flatAggregate(top2('a') as ('v', 'rank'))
  .select('key', 'v', 'rank')
```



以下示例显示如何使用 `emitUpdateWithRetract` 方法仅发出更新。为了仅发出更新，在我们的示例中，累加器同时保留了旧的和新的前 2 个值。注意：如果 `topN` 的 `N` 大，则保留旧值和新值都可能无效。解决这种情况的一种方法是将输入记录以累加方法存储到累加器中，然后在 `emitUpdateWithRetract` 中执行计算。

```
import java.lang.{Integer => JInteger}
import org.apache.flink.table.api.Types
import org.apache.flink.table.functions.TableAggregateFunction

/**
 * Accumulator for top2.
 */
class Top2Accum {
  var first: JInteger = _
  var second: JInteger = _
  var oldFirst: JInteger = _
  var oldSecond: JInteger = _
}

/**
 * The top2 user-defined table aggregate function.
 */
class Top2 extends TableAggregateFunction[JTuple2[JInteger,
JInteger], Top2Accum] {

  override def createAccumulator(): Top2Accum = {
    val acc = new Top2Accum
    acc.first = Int.MinValue
    acc.second = Int.MinValue
    acc.oldFirst = Int.MinValue
    acc.oldSecond = Int.MinValue
    acc
  }

  def accumulate(acc: Top2Accum, v: Int) {
    if (v > acc.first) {
```

```

    acc.second = acc.first
    acc.first = v
  } else if (v > acc.second) {
    acc.second = v
  }
}

def emitUpdateWithRetract(
  acc: Top2Accum,
  out: RetractableCollector[JTuple2[JInteger, JInteger]])
: Unit = {
  if (acc.first != acc.oldFirst) {
    // if there is an update, retract old value then emit new
    value.
    if (acc.oldFirst != Int.MinValue) {
      out.retract(JTuple2.of(acc.oldFirst, 1))
    }
    out.collect(JTuple2.of(acc.first, 1))
    acc.oldFirst = acc.first
  }
  if (acc.second != acc.oldSecond) {
    // if there is an update, retract old value then emit new
    value.
    if (acc.oldSecond != Int.MinValue) {
      out.retract(JTuple2.of(acc.oldSecond, 2))
    }
    out.collect(JTuple2.of(acc.second, 2))
    acc.oldSecond = acc.second
  }
}

// init table
val tab = ...

// use function
tab
  .groupBy('key)
  .flatMapAggregate(top2('a) as ('v, 'rank))
  .select('key, 'v, 'rank)

```



实施 UDF 的最佳做法

Table API 和 SQL 代码生成在内部尝试尽可能多地使用原始值。用户定义的函数可能会通过对象创建，转换和装箱带来很多开销。因此，强烈建议将参数和结果类型声明为基本类型，而不是其框内的类。Types.DATE 和 Types.TIME 也可以表示为 int。Types.TIMESTAMP 可以表示为 long。

我们建议用户定义的函数应使用 Java 而不是 Scala 编写，因为 Scala 类型对 Flink 的类型提取器构成了挑战。

将 UDF 与 Runtime 集成

有时，用户定义的函数可能有必要在实际工作之前获取全局运行时信息或进行一些设置/清理工作。用户定义的函数提供可被覆盖的 open () 和 close () 方法，并提供与 DataSet 或 DataStream API 的 RichFunction 中的方法相似的功能。


open () 方法在评估方法之前被调用一次。最后一次调用评估方法之后调用 close () 方法。

open () 方法提供一个 FunctionContext，其中包含有关在其中执行用户定义的函数的上下文的信息，例如度量标准组，分布式缓存文件或全局作业参数。

通过调用 FunctionContext 的相应方法可以获得以下信息：

方法	描述
getMetricGroup()	此并行子任务的度量标准组。
getCachedFile(name)	分布式缓存文件的本地临时文件副本。
getJobParameter(name, defaultValue)	与给定键关联的全局作业参数值。

以下示例片段显示了如何 FunctionContext 在标量函数中使用它来访问全局 job 参数：

```

object hashCode extends ScalarFunction {

    var hashCode_factor = 12

    override def open(context: FunctionContext): Unit = {
        // access "hashCode_factor" parameter
        // "12" would be the default value if parameter does not exist
        hashCode_factor = context.getJobParameter("hashCode_factor",
"12").toInt
    }

    def eval(s: String): Int = {
        s.hashCode() * hashCode_factor
    }
}
```

```
val tableEnv = BatchTableEnvironment.create(env)

// use the function in Scala Table API
myTable.select('string, hashCode('string))

// register and use the function in SQL
tableEnv.registerFunction("hashCode", hashCode)
tableEnv.sqlQuery("SELECT string, HASHCODE(string) FROM MyTable")
```



【翻译】Flink Table Api & SQL — Catalog Beta 版

本文翻译自官网: Catalogs Beta <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/catalogs.html>

Flink Table Api & SQL 翻译目录

Catalogs 提供元数据，例如数据库，表，分区，视图以及访问存储在数据库或其他外部系统中的数据所需的功能和信息。

数据处理的最关键方面之一是管理元数据。它可能是临时元数据，例如临时表，或者是针对表环境注册的 UDF。或永久性元数据，例如 Hive Metastore 中的元数据。Catalogs 提供了一个统一的 API，用于管理元数据并使其可从 Table API 和 SQL 查询访问。

- Catalogs 类型
 - GenericInMemory Catalogs
 - Hive Catalogs
 - 用户定义的 Catalogs
- Catalogs API
 - 注册 Catalogs
 - 更改当前 Catalogs 和数据库
 - 列出可用 Catalogs
 - 列出可用的数据库
 - 列出可用表

Catalogs 类型

GenericInMemory Catalog

Flink 会话始终具有一个名为 `default_catalog` 的内置 `GenericInMemoryCatalog`，它具有一个名为 `default_database` 的内置默认数据库。所有临时元数据（例如使用 `TableEnvironment#registerTable` 定义的表）均已注册到此目录。

Hive Catalog

HiveCatalog 有两个作用： 作为纯 Flink 元数据的持久存储，以及作为读写现有 Hive 元数据的接口。 Flink 的 Hive 文档提供了有关设置 catalog 以及与现有 Hive 安装接口的完整详细信息。

警告：Hive Metastore 以小写形式存储所有元对象名称。这与 GenericInMemoryCatalog 区分大小写不同。

用户定义的 Catalog

Catalog 是可插入的，用户可以通过实现 Catalog 接口来开发自定义 Catalog。 要在 SQL CLI 中使用自定义 Catalog，用户应通过实现 CatalogFactory 接口来开发 Catalog 及其相应的 CatalogFactory。

catalog factory 定义了一组属性，用于在 SQL CLI 引导时配置 catalog 。 属性集将传递给发现服务，在该服务中，服务将尝试将这些属性与 CatalogFactory 匹配并启动相应的 catalog 实例。

Catalog API

注册 Catalog

用户可以将其他 Catalog 注册到现有的 Flink 会话中。

```
tableEnv.registerCatalog(new CustomCatalog("myCatalog"));
```

更改当前 Catalog 和数据库

Flink 将始终在当前 Catalog 和数据库中搜索表，视图和 UDF。

```
tableEnv.useCatalog("myCatalog");
tableEnv.useDatabase("myDb");
```

通过以 catalog.database.object 形式提供标准名称，可以访问不是当前 catalog 中的元数据。

```
tableEnv.scan("not_the_current_catalog", "not_the_current_db",
"my_table");
```

列出可用 Catalog

```
tableEnv.listCatalogs();
```

列出可用的数据库

```
tableEnv.listDatabases();
```

列出可用表

```
tableEnv.listTables();
```

【翻译】Flink Table Api & SQL — SQL 客户端 Beta 版

本文翻译自官网：SQL Client Beta <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/sqlClient.html>

Flink Table Api & SQL 翻译目录

Flink 的 Table & SQL API 使使用 SQL 语言编写的查询成为可能，但是这些查询需要嵌入用 Java 或 Scala 编写的表程序中。此外，在将这些程序提交给集群之前，需要将它们与构建工具打包在一起。这或多或少地将 Flink 的使用限制为 Java / Scala 程序员。

SQL 客户端旨在提供一种简单的方法来编写，调试和提交表程序到 Flink 集群，而无需一行 Java 或 Scala 代码。SQL Client CLI 允许从命令行上正在运行的分布式应用程序检索和可视化实时结果。



注：[动图](#)，请查看[源网页](#)

注意：SQL Client 处于早期开发阶段。即该应用程序尚未投入生产，它对于原型制作和使用 Flink SQL 还是一个非常有用的工具。将来，社区计划通过提供基于 REST 的 SQL Client Gateway 来扩展其功能。

- 入门
 - 启动 SQL 客户端 CLI
 - 运行 SQL 查询
- 配置
 - 环境文件
 - 依存关系
 - 用户定义的函数
- Catalog

- [分离的 SQL 查询](#)
- [SQL 视图](#)
- [时态表](#)
- [局限与未来](#)

入门

本节介绍如何从命令行设置和运行第一个 Flink SQL 程序。

SQL 客户端捆绑在常规的 Flink 发行版中，因此可以直接运行。它只需要一个运行中的 Flink 集群即可在其中执行表程序。有关设置 Flink 集群的更多信息，请参见“[集群和部署](#)”部分。如果只想试用 SQL Client，也可以使用以下命令以一个工作程序启动本地集群：

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

启动 SQL 客户端 CLI

SQL Client 脚本也位于 Flink 的二进制目录中。[将来](#)，用户可以通过启动嵌入式独立进程或连接到远程 SQL Client Gateway 来启动 SQL Client CLI 的两种可能性。目前仅 embedded 支持该模式。您可以通过以下方式启动 CLI：

```
./bin/sql-client.sh embedded
```

默认情况下，SQL 客户端将从位于中的环境文件中读取其配置 `./conf/sql-client-defaults.yaml`。有关环境文件的结构的更多信息，请参见[配置部分](#)。

运行 SQL 查询

CLI 启动后，您可以使用 `HELP` 命令列出所有可用的 SQL 语句。为了验证您的设置和集群连接，您可以输入第一个 SQL 查询，输入 Enter 按键执行它：

```
SELECT 'Hello World';
```

该查询不需要表源，并且只产生一行结果。CLI 将从群集中检索结果并将其可视化。您可以通过按 Q 键关闭结果视图。

CLI 支持两种用于维护和可视化结果的模式。

表格模式将结果具体化到内存中，并以规则的分页表格表示形式将其可视化。可以通过在 CLI 中执行以下命令来启用它：

```
SET execution.result-mode=table;
```

更改日志模式不会具体化结果，并且无法可视化由包含插入（+）和撤回（-）的连续查询产生的结果流。

```
SET execution.result-mode=changelog;
```

您可以使用以下查询来查看两种结果模式的运行情况：

```
SELECT name, COUNT(*) AS cnt FROM (VALUES ('Bob'), ('Alice'),  
('Greg'), ('Bob')) AS NameTable(name) GROUP BY name;
```

此查询执行一个有限字数示例。

在变更日志模式下，可视化的变更日志类似于：

```
+ Bob, 1
+ Alice, 1
+ Greg, 1
- Bob, 1
+ Bob, 2
```

在表格模式下，可视化结果表格将不断更新，直到表格程序以以下内容结束：

```
Bob, 2
Alice, 1
Greg, 1
```

这两种结果模式在 **SQL** 查询的原型制作过程中都非常有用。在这两种模式下，结果都存储在 **SQL Client** 的 **Java** 堆内存中。为了保持 **CLI** 界面的响应性，更改日志模式仅显示最新的 **1000** 个更改。表格模式允许浏览更大的结果，这些结果仅受可用主存储器和配置的最大行数（最大表结果行）限制。

注意：在批处理环境中执行的查询只能使用表结果模式来检索。

定义查询后，可以将其作为长期运行的独立 **Flink** 作业提交给集群。为此，需要使用 **INSERT INTO** 语句指定存储结果的目标系统。配置部分说明如何声明用于读取数据的表源，如何声明用于写入数据的表接收器以及如何配置其他表程序属性。

配置

可以使用以下可选的 **CLI** 命令启动 **SQL Client**。在随后的段落中将详细讨论它们。

```
./bin/sql-client.sh embedded --help
```

Mode "embedded" submits Flink jobs from the local machine.

Syntax: embedded [OPTIONS]

"embedded" mode options:

- d,--defaults <environment file> The environment properties
with which every new session is
initialized.
- e,--environment <environment file> Properties might be
overwritten by session properties.
The environment properties
to be imported into the session.

It might

environment	overwrite default
<code>-h,--help</code>	properties. Show the help message with descriptions of all
options.	
<code>-j,--jar <JAR file></code>	A JAR file to be imported
into the	session. The file might
contain	user-defined classes
needed for the	execution of statements
such as	functions, table sources,
or sinks.	Can be used multiple
times.	
<code>-l,--library <JAR directory></code>	A JAR file directory with
which every	new session is
initialized. The files	might contain user-defined
classes	needed for the execution
of	statements such as
functions, table	sources, or sinks. Can be
used	multiple times.
<code>-s,--session <session identifier></code>	The identifier for a
session.	'default' is the default
identifier.	



环境文件

SQL 查询需要在其中执行配置环境。所谓的 *环境文件* 定义了可用的目录，表 **source** 和 **sink**，用户定义的函数以及执行和部署所需的其他属性。

每个环境文件都是常规的 [YAML 文件](#)。下面提供了此类文件的示例。



```
# Define tables here such as sources, sinks, views, or temporal tables.
```

```
tables:
```

- name: MyTableSource
 - type: source-table
 - update-mode: append
 - connector:
 - type: filesystem
 - path: "/path/to/something.csv"
 - format:
 - type: csv
 - fields:
 - name: MyField1
 - type: INT
 - name: MyField2
 - type: VARCHAR
 - line-delimiter: "\n"
 - comment-prefix: "#"
 - schema:
 - name: MyField1
 - type: INT
 - name: MyField2
 - type: VARCHAR
- name: MyCustomView
 - type: view
 - query: "SELECT MyField2 FROM MyTableSource"

```
# Define user-defined functions here.
```

```
functions:
```

- name: myUDF
 - from: `class`
 - `class`: foo.bar.AggregateUDF
 - constructor:
 - 7.6
 - `false`

```
# Define available catalogs
```

```
catalogs:
```

- name: catalog_1
 - type: hive
 - property-version: 1

```

    hive-conf-dir: ...
- name: catalog_2
  type: hive
  property-version: 1
  default-database: mydb2
  hive-conf-dir: ...
  hive-version: 1.2.1

# Properties that change the fundamental execution behavior of a
table program.

execution:
  planner: old                                # optional: either 'old'
  (default) or 'blink'
  type: streaming                             # required: execution mode either
  'batch' or 'streaming'
  result-mode: table                          # required: either 'table' or
  'changelog'
  max-table-result-rows: 1000000              # optional: maximum number of
  maintained rows in
                                              # 'table' mode (1000000 by
  default, smaller 1 means unlimited)
  time-characteristic: event-time             # optional: 'processing-time' or
  'event-time' (default)
  parallelism: 1                             # optional: Flink's parallelism
  (1 by default)
  periodic-watermarks-interval: 200           # optional: interval for periodic
  watermarks (200 ms by default)
  max-parallelism: 16                         # optional: Flink's maximum
  parallelism (128 by default)
  min-idle-state-retention: 0                 # optional: table program's
  minimum idle state time
  max-idle-state-retention: 0                 # optional: table program's
  maximum idle state time
  current-catalog: catalog_1                  # optional: name of the current
  catalog of the session ('default_catalog' by default)
  current-database: mydb1                     # optional: name of the current
  database of the current catalog
                                              # (default database of the
  current catalog by default)
  restart-strategy:                           # optional: restart strategy
  type: fallback                              # "fallback" to global restart
  strategy by default

```

```
# Configuration options for adjusting and tuning table programs.

# A full list of options and their default values can be found
# on the dedicated "Configuration" page.
configuration:
  table.optimizer.join-reorder-enabled: true
  table.exec.spill-compression.enabled: true
  table.exec.spill-compression.block-size: 128kb

# Properties that describe the cluster to which table programs are
# submitted to.

deployment:
  response-timeout: 5000
```



配置：

- 使用表源 `MyTableSource` 定义环境，该表源从 CSV 文件读取
- 定义一个视图 `MyCustomView`，该视图使用 SQL 查询声明一个虚拟表
- 定义一个用户定义的函数 `myUDF`，该函数可以使用类名和两个构造函数参数进行实例化，
- 连接到两个 Hive catalog，并使用 `catalog_1` 作为当前 catalog，使用 `mydb1` 作为该 catalog 的当前数据库
- 使用旧 planner 以流模式运行具有事件时间特征和并行度为 1 的语句
- 在表结果模式下运行探索性查询
- 并通过配置选项围绕联接的重新排序和溢出进行一些 planner 调整。

根据使用情况，可以将配置拆分为多个文件。因此，可以出于一般目的（使用 `--defaults` 使用默认环境文件）以及基于每个会话（使用 `--environment` 使用会话环境文件）来创建环境文件。每个 CLI 会话均使用默认属性初始化，后跟会话属性。例如，默认环境文件可以指定在每个会话中都可用于查询的所有表源，而会话环境文件仅声明特定的状态保留时间和并行性。启动 CLI 应用程序时，可以传递默认环境文件和会话环境文件。如果未指定默认环境文件，则 SQL 客户端会在 Flink 的配置目录中搜索 `./conf/sql-client-defaults.yaml`。

注意：在 CLI 会话中设置的属性（例如，使用 `SET` 命令）具有最高优先级：

```
CLI commands > session environment file > defaults environment file
```

重启策略

重启策略控制在发生故障时如何重新启动 Flink 作业。与 Flink 群集的全局重启策略类似，可以在环境文件中声明更细粒度的重启配置。

支持以下策略：



```
execution:
```

```

# falls back to the global strategy defined in flink-conf.yaml
restart-strategy:
  type: fallback

# job fails directly and no restart is attempted
restart-strategy:
  type: none

# attempts a given number of times to restart the job
restart-strategy:
  type: fixed-delay
  attempts: 3      # retries before job is declared as failed
                    (default: Integer.MAX_VALUE)
  delay: 10000     # delay in ms between retries (default: 10 s)

# attempts as long as the maximum number of failures per time
interval is not exceeded
restart-strategy:
  type: failure-rate
  max-failures-per-interval: 1  # retries in interval until
failing (default: 1)
  failure-rate-interval: 60000  # measuring interval in ms for
failure rate
  delay: 10000                  # delay in ms between retries
                                (default: 10 s)

```



依赖关系

SQL 客户端不需要使用 Maven 或 SBT 设置 Java 项目。相反，您可以将依赖项作为常规 JAR 文件传递，然后将其提交给集群。您可以单独指定每个 JAR 文件（使用 `--jar`），也可以定义整个库目录（使用 `--library`）。对于外部系统（例如 Apache Kafka）和相应数据格式（例如 JSON）的连接器，Flink 提供了现成的 JAR 捆绑包。可以从 Maven 中央存储库为每个发行版下载这些 JAR 文件。

提供的 SQL JAR 的完整列表以及有关如何使用它们的文档可以在与[外部系统的连接页面](#)上找到。

以下示例显示了一个环境文件，该文件定义了一个表源，该表源从 Apache Kafka 读取 JSON 数据。



```

tables:
  - name: TaxiRides
    type: source-table
    update-mode: append
    connector:

```

```

property-version: 1
type: kafka
version: "0.11"
topic: TaxiRides
startup-mode: earliest-offset
properties:
  - key: zookeeper.connect
    value: localhost:2181
  - key: bootstrap.servers
    value: localhost:9092
  - key: group.id
    value: testGroup
format:
  property-version: 1
  type: json
  schema: "ROW<rideId LONG, lon FLOAT, lat FLOAT, rideTime
TIMESTAMP>"
  schema:
    - name: rideId
      type: LONG
    - name: lon
      type: FLOAT
    - name: lat
      type: FLOAT
    - name: rowTime
      type: TIMESTAMP
    rowtime:
      timestamps:
        type: "from-field"
        from: "rideTime"
      watermarks:
        type: "periodic-bounded"
        delay: "60000"
    - name: procTime
      type: TIMESTAMP
    proctime: true

```



TaxiRide 表的结果模式包含 JSON 模式的大多数字段。此外，它添加了行时间属性 `rowTime` 和处理时间属性 `procTime`。

连接器和格式都允许定义属性版本（当前为版本 1），以便将来向后兼容。

用户定义的函数

SQL 客户端允许用户创建要在 SQL 查询中使用的自定义用户定义函数。当前，这些函数仅限于以编程方式在 **Java / Scala** 类中定义。

为了提供用户定义的函数，您需要首先实现并编译扩展 **ScalarFunction**，**AggregateFunction** 或 **TableFunction** 的函数类（请参阅用户定义的函数）。然后可以将一个或多个函数打包到 SQL 客户端的依赖项 **JAR** 中。

在调用之前，必须在环境文件中声明所有函数。对于函数列表中的每一项，必须指定

- 函数注册的名称
- 使用的函数源（目前仅限于类）
- 指示函数的完全限定的类名称的类，以及用于实例化的可选构造函数参数列表。

```
functions:
  - name: ...           # required: name of the function
    from: class         # required: source of the function (can
only be "class" for now)
    class: ...          # required: fully qualified class name of
the function
    constructor:        # optimal: constructor parameters of the
function class
      - ...             # optimal: a literal parameter with
implicit type
      - class: ...      # optimal: full class name of the
parameter
        constructor:    # optimal: constructor parameters of the
parameter's class
          - type: ...    # optimal: type of the literal parameter
          value: ...     # optimal: value of the literal parameter
```

确保指定参数的顺序和类型严格匹配函数类的构造函数之一。

函数构造参数

根据用户定义的函数，可能有必要在 SQL 语句中使用实现之前对其进行参数化。

如前面的示例所示，在声明用户定义的函数时，可以通过以下三种方式之一使用构造函数参数来配置类：

具有隐式类型的文字值： SQL Client 将根据文字值本身自动派生类型。目前，仅支持 **BOOLEAN**，**INT**，**DOUBLE** 和 **VARCHAR**。如果自动派生无法按预期进行（例如，您需要 **VARCHAR false**），请改用显式类型。

```
- true           # -> BOOLEAN (case sensitive)
- 42             # -> INT
- 1234.222       # -> DOUBLE
- foo           # -> VARCHAR
```

具有显式类型的文字值：使用类型属性显式声明参数 `type` 和 `value` 属性。

```
- type: DECIMAL
  value: 1111111111111111
```

下表说明了受支持的 **Java** 参数类型和相应的 **SQL** 类型字符串。

Java type	SQL type
java.math.BigDecimal	DECIMAL
java.lang.Boolean	BOOLEAN
java.lang.Byte	TINYINT
java.lang.Double	DOUBLE
java.lang.Float	REAL, FLOAT
java.lang.Integer	INTEGER, INT
java.lang.Long	BIGINT
java.lang.Short	SMALLINT
java.lang.String	VARCHAR

目前尚不支持更多类型（例如 **TIMESTAMP** 或 **ARRAY**），基本类型和 **null**。

（嵌套的）类实例：除了文字值，还可以通过指定 `class` 和 `constructor` 属性来为构造函数参数创建（嵌套的）类实例。可以递归执行此过程，直到所有构造函数参数都用文字值表示为止。

```
- class: foo.bar.paramClass
  constructor:
    - StarryName
    - class: java.lang.Integer
      constructor:
        - class: java.lang.String
          constructor:
            - type: VARCHAR
              value: 3
```

Catalogs

可以将 `catalog` 定义为一组 **YAML** 属性，并在启动 **SQL Client** 时自动将其注册到环境中。

用户可以在 SQL CLI 中指定当前 `catalog`，以及要用作当前数据库的 `catalog` 的数据库。



```
catalogs:
  - name: catalog_1
    type: hive
    property-version: 1
    default-database: mydb2
    hive-version: 1.2.1
    hive-conf-dir: <path of Hive conf directory>
  - name: catalog_2
    type: hive
    property-version: 1
    hive-conf-dir: <path of Hive conf directory>

execution:
  ...
  current-catalog: catalog_1
  current-database: mydb1
```


有关 `catalog` 的更多信息，请参见 [catalog](#)。

分离的 SQL 查询

为了定义端到端的 SQL 管道，可以使用 SQL 的 `INSERT INTO` 语句向 Flink 集群提交长时间运行的分离查询。这些查询将其结果生成到外部系统而不是 SQL Client 中。这允许处理更高的并行度和更大数量的数据。提交后，CLI 本身对分离的查询没有任何控制权。

```
INSERT INTO MyTableSink SELECT * FROM MyTableSource
```

表接收器 `MyTableSink` 必须在环境文件中声明。有关支持的外部系统及其配置的更多信息，请参见连接页面。下面显示了 Apache Kafka table sink 的示例。



```
tables:
  - name: MyTableSink
    type: sink-table
    update-mode: append
    connector:
      property-version: 1
      type: kafka
      version: "0.11"
      topic: OutputTopic
      properties:
        - key: zookeeper.connect
          value: localhost:2181
```

```

- key: bootstrap.servers
  value: localhost:9092
- key: group.id
  value: testGroup
format:
  property-version: 1
  type: json
  derive-schema: true
schema:
  - name: rideId
    type: LONG
  - name: lon
    type: FLOAT
  - name: lat
    type: FLOAT
  - name: rideTime
    type: TIMESTAMP

```



SQL 客户端确保语句已成功提交到群集。提交查询后，CLI 将显示有关 Flink 作业的信息。

```

[INFO] Table update statement has been successfully submitted to the
cluster:
Cluster ID: StandaloneClusterId
Job ID: 6f922fe5cba87406ff23ae4a7bb79044
Web interface: http://localhost:8081

```

注意：提交后，SQL 客户端不会跟踪正在运行的 Flink 作业的状态。提交后可以关闭 CLI 进程，而不会影响分离的查询。Flink 的重启策略可确保容错能力。可以使用 Flink 的 web 界面，命令行或 REST API 取消查询。

SQL 视图

视图允许通过 SQL 查询定义虚拟表。视图定义被立即解析和验证。但是，实际执行是在提交常规 INSERT INTO 或 SELECT 语句期间访问视图时发生的。

可以在环境文件中或在 CLI 会话中定义视图。

以下示例显示如何在一个文件中定义多个视图。按照在环境文件中定义的顺序注册视图。支持诸如视图 A 依赖于视图 B 依赖于视图 C 的引用链。



```

tables:
  - name: MyTableSource
    # ...
  - name: MyRestrictedView
    type: view
    query: "SELECT MyField2 FROM MyTableSource"

```

```
- name: MyComplexView
  type: view
  query: >
    SELECT MyField2 + 42, CAST(MyField1 AS VARCHAR)
    FROM MyTableSource
    WHERE MyField2 > 200
```



与表源和接收器相似，会话环境文件中定义的视图具有最高优先级。

也可以使用 **CREATE VIEW** 语句在 CLI 会话中创建视图：

```
CREATE VIEW MyNewView AS SELECT MyField2 FROM MyTableSource;
```

也可以使用 **DROP VIEW** 语句删除在 CLI 会话中创建的视图：

```
DROP VIEW MyNewView;
```

注意：CLI 中视图的定义仅限于上述语法。将来的版本将支持为视图定义表名或在表名中转义空格。

时态表

时态表允许在变化的历史记录表上进行（参数化）视图，该视图返回表在特定时间点的内容。这对于在特定时间戳将一个表与另一个表的内容连接起来特别有用。在[时态表联接页面](#)中可以找到更多信息。

以下示例显示如何定义时态表 **SourceTemporalTable**：



tables:

```
# Define the table source (or view) that contains updates to a
temporal table
```

```
- name: HistorySource
  type: source-table
  update-mode: append
  connector: # ...
  format: # ...
  schema:
    - name: integerField
      type: INT
    - name: stringField
      type: VARCHAR
    - name: rowtimeField
      type: TIMESTAMP
      rowtime:
        timestamps:
          type: from-field
```

```
        from: rowtimeField
    watermarks:
        type: from-source

    # Define a temporal table over the changing history table with time
    attribute and primary key
    - name: SourceTemporalTable
      type: temporal-table
      history-table: HistorySource
      primary-key: integerField
      time-attribute: rowtimeField # could also be a proctime field
```



如示例中所示，表源，视图和时态表的定义可以相互混合。按照在环境文件中定义的顺序注册它们。例如，时态表可以引用一个视图，该视图可以依赖于另一个视图或表源。

局限与未来

当前的 SQL Client 实现处于非常早期的开发阶段，作为更大的 Flink 改进提案 24（[FLIP-24](#)）的一部分，将来可能会更改。随时加入有关您发现有用的错误和功能的讨论并公开发表问题。

【翻译】Flink Table Api & SQL — Hive Beta

本文翻译自官网：Hive Beta <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/hive/>

Flink Table Api & SQL 翻译目录

Apache Hive 已将自己确立为数据仓库生态系统的焦点。它不仅充当用于大数据分析和 ETL 的 SQL 引擎，而且也是数据发现，定义和演变数据的数据管理平台。

Flink 提供了与 Hive 的双重集成。首先是利用 Hive 的 Metastore 作为持久性 catalog，以跨会话存储 Flink 特定的元数据。第二个是提供 Flink 作为读取和写入 Hive 表的替代引擎。

hive catalog 旨在与现有的 hive 安装程序“开箱即用”兼容。您不需要修改现有的 Hive Metastore 或更改表的数据放置或分区。

- [支持的 Hive 版本](#)
 - [依赖](#)
- [连接到 Hive](#)
- [支持的类型](#)
 - [局限性](#)

Flink 支持 Hive 2.3.4, 1.2.1 并且依赖于 Hive 对其他次要版本的兼容性保证。


如果您使用其他次要 Hive 版本, 例如 1.2.2 或 2.3.1, 则还可以选择最接近的版本 1.2.1 (对于 1.2.2) 或 2.3.4 (对于 2.3.1) 来解决。例如, 您要使用 Flink 在 SQL 客户端中集成 2.3.1 hive 版本, 只需在 YAML 配置中将 `hive-version` 设置为 2.3.4。通过 Table API 创建 HiveCatalog 实例时, 类似地传递版本字符串。

欢迎用户使用此替代方法尝试不同的版本。由于仅测试了 2.3.4 和 1.2.1, 所以可能存在意外问题。我们将在将来的版本中测试并支持更多版本。

依赖

为了与 Hive 集成, 用户在他们的项目中需要以下依赖项。

hive 2.3.4



```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-hive_2.11</artifactId>
  <version>1.9.0</version>
  <scope>provided</scope>
</dependency>

<!-- Hadoop Dependencies -->

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-hadoop-compatibility_2.11</artifactId>
  <version>1.9.0</version>
  <scope>provided</scope>
</dependency>

<!-- Hive 2.3.4 is built with Hadoop 2.7.2. We pick 2.7.5 which
flink-shaded-hadoop is pre-built with, but users can pick their own
hadoop version, as long as it's compatible with Hadoop 2.7.2 -->

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-shaded-hadoop-2-uber</artifactId>
  <version>2.7.5-8.0</version>
  <scope>provided</scope>
</dependency>

<!-- Hive Metastore -->
<dependency>
  <groupId>org.apache.hive</groupId>
```

```
    <artifactId>hive-exec</artifactId>
    <version>2.3.4</version>
</dependency>
```



hive 1.2.1



```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-hive_2.11</artifactId>
  <version>1.9.0</version>
  <scope>provided</scope>
</dependency>

<!-- Hadoop Dependencies -->

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-hadoop-compatibility_2.11</artifactId>
  <version>1.9.0</version>
  <scope>provided</scope>
</dependency>

<!-- Hive 1.2.1 is built with Hadoop 2.6.0. We pick 2.6.5 which
flink-shaded-hadoop is pre-built with, but users can pick their own
hadoop version, as long as it's compatible with Hadoop 2.6.0 -->

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-shaded-hadoop-2-uber</artifactId>
  <version>2.6.5-8.0</version>
  <scope>provided</scope>
</dependency>

<!-- Hive Metastore -->
<dependency>
  <groupId>org.apache.hive</groupId>
  <artifactId>hive-metastore</artifactId>
  <version>1.2.1</version>
</dependency>

<dependency>
  <groupId>org.apache.hive</groupId>
  <artifactId>hive-exec</artifactId>
```

```
<version>1.2.1</version>
</dependency>

<dependency>
  <groupId>org.apache.thrift</groupId>
  <artifactId>libfb303</artifactId>
  <version>0.9.3</version>
</dependency>
```



连接到 Hive

通过表环境或 YAML 配置，使用 Hive catalog 连接到现有的 Hive 安装程序。



```
val name          = "myhive"
val defaultDatabase = "mydatabase"
val hiveConfDir    = "/opt/hive-conf"
val version        = "2.3.4" // or 1.2.1

val hive = new HiveCatalog(name, defaultDatabase, hiveConfDir,
version)
tableEnv.registerCatalog("myhive", hive)
```



支持的类型

当前 HiveCatalog 支持具有以下映射的大多数 Flink 数据类型：

Flink Data Type	Hive Data Type
CHAR(p)	CHAR(p)
VARCHAR(p)	VARCHAR(p)
STRING	STRING
BOOLEAN	BOOLEAN
TINYINT	TINYINT
SMALLINT	SMALLINT
INT	INT
BIGINT	LONG

Flink Data Type	Hive Data Type
FLOAT	FLOAT
DOUBLE	DOUBLE
DECIMAL(p, s)	DECIMAL(p, s)
DATE	DATE
BYTES	BINARY
ARRAY<T>	LIST<T>
MAP<K, V>	MAP<K, V>
ROW	STRUCT

局限性

Hive 数据类型中的以下限制会影响 Flink 和 Hive 之间的映射：

- CHAR(p) 最大长度为 255
- VARCHAR(p) 最大长度为 65535
- Hive MAP 仅支持原始键类型，而 Flink MAP 可以是任何数据类型
- 不支持 Hive 的 UNION 类型
- Flink 的 INTERVAL 类型不能映射到 Hive INTERVAL 类型
- Hive 不支持 Flink TIMESTAMP_WITH_TIME_ZONE 和
TIMESTAMP_WITH_LOCAL_TIME_ZONE
- 由于精度差异，Flink 的 TIMESTAMP_WITHOUT_TIME_ZONE 类型无法映射到 Hive 的
TIMESTAMP 类型。
- Hive 不支持 Flink 的 MULTISET

【翻译】Flink Table Api & SQL — Hive —— 读写 Hive 表

本文翻译自官网：Reading & Writing Hive

Tables https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/hive/read_write_hive.html


Flink Table Api & SQL 翻译目录

使用 HiveCatalog 和 Flink 的 Hive 连接器，Flink 可以读取和写入 Hive 数据，以替代 Hive 的批处理引擎。确保遵循说明在您的应用程序中包括正确的依赖项。

- 从 Hive 读数据
- 写数据到 Hive
 - 局限性

从 Hive 读数据

假设 Hive 在其 default 数据库中包含一个表，该表名为 people，其中包含几行。



```
hive> show databases;
OK
default
Time taken: 0.841 seconds, Fetched: 1 row(s)

hive> show tables;
OK
Time taken: 0.087 seconds

hive> CREATE TABLE mytable(name string, value double);
OK
Time taken: 0.127 seconds

hive> SELECT * FROM mytable;
OK
Tom      4.72
John     8.0
Tom      24.2
Bob       3.14
Bob      4.72
Tom      34.9
Mary     4.79
Tiff     2.72
Bill     4.33
```

```
Mary 77.7
Time taken: 0.097 seconds, Fetched: 10 row(s)
```



准备好数据后，您可以连接到现有的 **Hive** 安装程序并开始查询。



```
Flink SQL> show catalogs;
myhive
default_catalog

# ----- Set the current catalog to be 'myhive' catalog if you
haven't set it in the yaml file -----

Flink SQL> use catalog myhive;

# ----- See all registered database in catalog 'mytable' -----

Flink SQL> show databases;
default

# ----- See the previously registered table 'mytable' -----

Flink SQL> show tables;
mytable

# ----- The table schema that Flink sees is the same that we created
in Hive, two columns - name as string and value as double -----
Flink SQL> describe mytable;
root
|-- name: name
|-- type: STRING
|-- name: value
|-- type: DOUBLE

Flink SQL> SELECT * FROM mytable;
```

name	value
Tom	4.72
John	8.0
Tom	24.2
Bob	3.14

Bob	4.72
Tom	34.9
Mary	4.79
Tiff	2.72
Bill	4.33
Mary	77.7



写数据到 **hive**

同样，可以使用 `INSERT INTO` 子句将数据写入 `hive`。

```
Flink SQL> INSERT INTO mytable (name, value) VALUES ('Tom', 4.72);
```

局限性

以下是 `Hive` 连接器的主要限制列表。我们正在积极努力缩小这些差距。

1. 不支持 `INSERT OVERWRITE`。
2. 不支持插入分区表。
3. 不支持 `ACID` 表。
4. 不支持存储桶的表。
5. 不支持某些数据类型。有关详细信息，请参见[限制](#)。
6. 仅测试了有限数量的表存储格式，即文本，`SequenceFile`，`ORC` 和 `Parquet`。
7. 不支持视图。

【翻译】Flink Table Api & SQL — Hive —— Hive 函数

本文翻译自官网：Hive Functions https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/hive/hive_functions.html

Flink Table Api & SQL 翻译目录

用户可以在 `Flink` 中使用 `Hive` 现有的自定义函数。

支持的 `UDF` 类型包括：

- `UDF`
- `GenericUDF`
- `GenericUDTF`
- `UDAF`
- `GenericUDAFResolver2`

根据查询的计划和执行，`Hive` 的 `UDF` 和 `GenericUDF` 会自动转换为 `Flink` 的 `ScalarFunction`，`Hive` 的 `GenericUDTF` 会自动转换为 `Flink` 的 `TableFunction`，`Hive` 的 `UDAF` 和 `GenericUDAFResolver2` 会转换为 `Flink` 的 `AggregateFunction`。

要使用 `Hive` 用户定义的函数，用户必须

- 设置由 Hive Metastore 支持的 HiveCatalog，其中包含该函数作为会话的当前 catalog
- 在 Flink 的 classpath 中包含该函数的 jar
- 使用 Blink planner

使用 **Hive** 自定义的函数

假设我们在 Hive Metastore 中注册了以下 Hive 函数：

```
/**
 * Test simple udf. Registered under name 'myudf'
 */
public class TestHiveSimpleUDF extends UDF {

    public IntWritable evaluate(IntWritable i) {
        return new IntWritable(i.get());
    }

    public Text evaluate(Text text) {
        return new Text(text.toString());
    }
}

/**
 * Test generic udf. Registered under name 'mygenericudf'
 */
public class TestHiveGenericUDF extends GenericUDF {

    @Override
    public ObjectInspector initialize(ObjectInspector[] arguments)
    throws UDFArgumentException {
        checkArgument(arguments.length == 2);

        checkArgument(arguments[1] instanceof
ConstantObjectInspector);
        Object constant = ((ConstantObjectInspector)
arguments[1]).getWritableConstantValue();
        checkArgument(constant instanceof IntWritable);
        checkArgument(((IntWritable) constant).get() == 1);

        if (arguments[0] instanceof IntObjectInspector ||
            arguments[0] instanceof StringObjectInspector) {
```

```

        return arguments[0];
    } else {
        throw new RuntimeException("Not support argument: " +
arguments[0]);
    }
}

@Override
public Object evaluate(DeferredObject[] arguments) throws
HiveException {
    return arguments[0].get();
}

@Override
public String getDisplayString(String[] children) {
    return "TestHiveGenericUDF";
}
}

/**
 * Test split udtf. Registered under name 'mygenericudtf'
 */
public class TestHiveUDTF extends GenericUDTF {

    @Override
    public StructObjectInspector initialize(ObjectInspector[] argOIs)
throws UDFArgumentException {
        checkArgument(argOIs.length == 2);

        // TEST for constant arguments
        checkArgument(argOIs[1] instanceof ConstantObjectInspector);
        Object constant = ((ConstantObjectInspector)
argOIs[1]).getWritableConstantValue();
        checkArgument(constant instanceof IntWritable);
        checkArgument(((IntWritable) constant).get() == 1);

        return
ObjectInspectorFactory.getStandardStructObjectInspector(
            Collections.singletonList("col1"),

Collections.singletonList(PrimitiveObjectInspectorFactory.javaStringO
bjectInspector));
    }
}


```

```

@Override
public void process(Object[] args) throws HiveException {
    String str = (String) args[0];
    for (String s : str.split(",")) {
        forward(s);
        forward(s);
    }
}

@Override
public void close() {
}
}

```



从 Hive CLI 中，我们可以看到它们已注册：

```

hive> show functions;
OK
.....
mygenericudf
myudf
myudtf

```

然后，用户可以在 SQL 中以如下方式使用它们：

```

Flink SQL> select mygenericudf(myudf(name), 1) as a,
mygenericudf(myudf(age), 1) as b, s from mysourcetable, lateral
table(myudtf(name, 1)) as T(s);

```

局限性

Flink 中现时不支持 Hive 内置内置。要使用 Hive 内置函数，用户必须首先在 Hive Metastore 中手动注册它们。

仅在 Blink planner 中测试了 Flink 批处理对 Hive 功能的支持。

Hive 函数当前不能在 Flink 中的各个 catalog 之间使用。

有关数据类型限制，请参考 [Hive](#)。

【翻译】Flink Table Api & SQL — Hive —— 在 scala shell 中使用 Hive 连接器

本文翻译自官网：Use Hive connector in scala shell https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/hive/scala_shell_hive.html

Flink Table Api & SQL 翻译目录

Flink Scala Shell 是尝试 flink 的便捷方法。您也可以在 scala shell 中使用 hive，而不是在 pom 文件中指定 hive 依赖关系，打包程序并通过 flink run 命令提交。为了在 scala shell 中使用 hive 连接器，您需要将以下 hive 连接器依赖项放在 flink dist 的 lib 文件夹下。

- flink-connector-hive_ {scala_version}-{flink.version} .jar
- flink-hadoop-compatibility_ {scala_version}-{flink.version} .jar
- flink-shaded-hadoop-2-uber- {hadoop.version}-{flink-shaded.version} .jar
- hive-exec-2.x.jar（对于 Hive 1.x，您需要复制 hive-exec-1.x.jar, hive-metastore-1.x.jar, libfb303-0.9.2.jar 和 libthrift- 0.9.2.jar）

然后，您可以在 scala shell 中使用 hive 连接器，如下所示：



```
Scala-Flink> import org.apache.flink.table.catalog.hive.HiveCatalog
Scala-Flink> val hiveCatalog = new HiveCatalog("hive", "default",
"<Replace it with HIVE_CONF_DIR>", "2.3.4");
Scala-Flink> btenv.registerCatalog("hive", hiveCatalog)
Scala-Flink> btenv.useCatalog("hive")
Scala-Flink> btenv.listTables
Scala-Flink> btenv.sqlQuery("<sql query>").toDataSet[Row].print()
```

【翻译】Flink Table Api & SQL — 配置

本文翻译自官网：Configuration <https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/config.html>

Flink Table Api & SQL 翻译目录

默认情况下，Table&SQL API 已预先配置为产生具有可接受性能的准确结果。

根据表程序的要求，可能需要调整某些参数以进行优化。例如，无界流程序可能需要确保所需的状态大小是有上限的（请参阅[流概念](#)）。

- [总览](#)
- [执行选项](#)
- [优化器选项](#)


总览

在每个表环境中，TableConfig 提供了用于配置当前会话的选项。

对于常见或重要的配置选项，TableConfig 提供了具有详细内联文档的 **getter** 和 **setter** 方法。


对于更高级的配置，用户可以直接访问基础键值映射。以下各节列出了可用于调整 Flink Table 和 SQL API 程序的所有可用选项。

注意：由于执行操作时会在不同的时间点读取选项，因此建议在实例化表环境后尽早设置配置选项。



```
// instantiate table environment
val tEnv: TableEnvironment = ...

// access flink configuration
val configuration = tEnv.getConfig().getConfiguration()
// set low-level key-value options
configuration.setString("table.exec.mini-batch.enabled", "true")
configuration.setString("table.exec.mini-batch.allow-latency", "5 s")
configuration.setString("table.exec.mini-batch.size", "5000")
```



主要：当前仅 Blink planner 支持键值对的配置选项

执行配置选项

以下选项可用于调整查询执行的性能。

Key	Default	Description
table.exec.async-lookup.buffer-capacity Batch Streaming	100	async lookup join 可以触发的最大 async i/o 操作的数量
table.exec.async-lookup.timeout	"3 min"	异步操作完成的 超时时间

Key	Default	Description
Batch Streaming		
table.exec.disabled-operators Batch	(none)	<p>主要用于测试。以逗号分隔的运算符名称列表，每个名称代表一种禁用的运算符。</p> <p>可以禁用的运算符包括“NestedLoopJoin”，“ShuffleHashJoin”，“BroadcastHashJoin”，“SortMergeJoin”，“HashAgg”，“SortAgg”。默认情况下，未禁用任何运算符。</p>
table.exec.mini-batch.allow-latency Streaming	"-1 ms"	<p>最大等待时间可用于 MiniBatch 缓冲输入记录。MiniBatch 是用于缓冲输入记录以减少状态访问的优化。</p> <p>MiniBatch 以允许的等待时间间隔以及达到最大缓冲记录数触发。</p> <p>注意：如果将 table.exec.mini-batch.enabled 设置为 true，则其值必须大于零。</p>
table.exec.mini-batch.enabled Streaming	false	<p>指定是否启用 MiniBatch 优化。MiniBatch 是用于缓冲输入记录以减少状态访问的优化。</p> <p>默认情况下禁用此功能。要启用此功能，用户应将此配置设置为 true。</p> <p>注意：如果启用了 mini batch 处理，则必须设置“table.exec.mini-batch.allow-latency”和“table.exec.mini-batch.size”。</p>
table.exec.mini-batch.size Streaming	-1	<p>可以为 MiniBatch 缓冲最大输入记录数。MiniBatch 是用于缓冲输入记录以减少状态访问的优化。</p> <p>MiniBatch 以允许的等待时间间隔以及达到最大缓冲记录数触发。注意：MiniBatch 当前仅适用于非窗口聚合。</p> <p>如果将 table.exec.mini-batch.enabled 设置为 true，则其值必须为正。</p>

Key	Default	Description
table.exec.resource.default-parallelism Batch Streaming	-1	<p>为所有运算符（例如聚合，联接，过滤器）设置默认并行度以与并行实例一起运行。</p> <p>此配置比 <code>StreamExecutionEnvironment</code> 的并行性具有更高的优先级</p> <p>（实际上，此配置优先于 <code>StreamExecutionEnvironment</code> 的并行性）。</p> <p>值-1 表示未设置默认的并行性，则使用 <code>StreamExecutionEnvironment</code> 的并行性将回退。</p>
table.exec.resource.external-buffer-memory Batch	"10 mb"	<p>设置在排序合并联接和嵌套联接以及窗口上使用的外部缓冲存储器大小。</p>
table.exec.resource.hash-agg.memory Batch	"128 mb"	<p>设置哈希聚合运算符的托管内存大小。</p>
table.exec.resource.hash-join.memory Batch	"128 mb"	<p>设置哈希联接运算符的托管内存。 定义下限。</p>
table.exec.resource.sort.memory Batch	"128 mb"	<p>设置排序运算符的托管缓冲区内存大小。</p>
table.exec.shuffle-mode	"batch"	<p>设置执行 <code>shuffle</code> 模式。 只能设置 <code>batch</code> 或 <code>pipeline</code>。 <code>batch</code>: 工作将逐步进行。</p>

Key	Default	Description
Batch		pipeline: 作业将以流模式运行，但是当发送方拥有资源等待将数据发送到接收方时，接收方等待资源启动可能会导致资源死锁。
table.exec.sort.async-merge-enabled Batch	true	是否异步合并排序的溢出文件。
table.exec.sort.default-limit Batch	-1	用户 <code>order</code> 后未设置限制时的默认限制。-1 表示此配置被忽略。
table.exec.sort.max-num-file-handles Batch	128	外部合并排序的最大扇入。它限制了每个运算符的文件句柄数。 如果太小，可能会导致中间合并。 但是，如果太大，将导致同时打开太多文件，占用内存并导致随机读取。
table.exec.source.idle-timeout Streaming	"-1 ms"	当 <code>source</code> 在超时时间内未收到任何元素时，它将被标记为临时空闲。 这样，下游任务就可以前进其水印，而无需在空闲时等待来自该源的水印。
table.exec.spill-compression.block-size Batch	"64 kb"	溢出数据时用于压缩的内存大小。 内存越大，压缩率越高，但是作业将消耗更多的内存资源。

Key	Default	Description
table.exec.spill-compression.enabled Batch	true	<p>是否压缩溢出的数据。</p> <p>目前，我们仅支持对 sort 和 hash-agg 和 hash-join 运算符压缩溢出的数据。</p>
table.exec.window-agg.buffer-size-limit Batch	100000	<p>设置组窗口 agg 运算符中使用的窗口元素缓冲区大小限制。</p>

优化器选项

以下选项可用于调整查询优化器的行为，以获得更好的执行计划。

Key	Default	Description
table.optimizer.agg-phase-strategy Batch Streaming	"AUTO"	<p>汇总阶段的策略。 只能设置 AUTO, TWO_PHASE 或 ONE_PHASE。</p> <p>自动：聚合阶段没有特殊的执行器。</p> <p>选择两阶段汇总还是一阶段汇总取决于成本。 TWO_PHASE: 强制使用具有 localAggregate 和 globalAggregate 的两阶段聚合。</p> <p>请注意，如果聚合调用不支持分为两阶段的优化，我们仍将使用一级聚合。</p> <p>ONE_PHASE: 强制使用仅具有 CompleteGlobalAggregate 的一级聚合。</p>
table.optimizer.distinct-agg.split.bucket-num Streaming	1024	<p>拆分独立聚合时配置存储桶数。</p> <p>该数字在第一级聚合中用于计算存储区密钥“hash_code (distinct_key) %BUCKET_NUM”，该存储区密钥在拆分后用作附加组密钥。</p>

Key	Default	Description
table.optimizer.distinct-agg.split-enabled Streaming	false	<p>告诉优化程序是否将不同的聚合（例如 COUNT（DISTINCT col），SUM（DISTINCT col））分成两个级别。</p> <p>第一次聚合被一个附加 key shuffle，该附加 key 使用 distinct_key 的哈希码和存储桶数计算得出。</p> <p>当不同的聚合中存在数据倾斜时，此优化非常有用，并且可以扩大工作量。默认为 false.</p>
table.optimizer.join-reorder-enabled Batch Streaming	false	<p>在优化器中启用联接重新排序。默认为禁用.</p>
table.optimizer.join.broadcast-threshold Batch	1048576	<p>配置表的最大大小（以字节为单位），该表在执行联接时将广播到所有工作程序节点。</p> <p>通过将此值设置为-1 以禁用广播.</p>
table.optimizer.reuse-source-enabled Batch Streaming	true	<p>如果为 true，则优化器将尝试找出重复的表源并重新使用它们。</p> <p>仅当启用 table.optimizer.reuse-sub-plan 为 true 时，此方法才有效.</p>
table.optimizer.reuse-sub-plan-enabled Batch Streaming	true	<p>当为 true 时，优化器将尝试找出重复的子计划并重用它们。</p>
table.optimizer.source.predicate-pushdown-enabled	true	<p>如果为 true，则优化器会将谓词下推到 FilterableTable Source 中。默认值为 true.</p>

Key	Default	Description
Batch Streaming		

【翻译】Flink Table Api & SQL — 性能调优 — 流式聚合

本文翻译自官网：Streaming Aggregation https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/table/tuning/streaming_aggregation_optimization.html

Flink Table Api & SQL 翻译目录

SQL 是用于数据分析的最广泛使用的语言。Flink 的 Table API 和 SQL 使用户能够以更少的时间和精力定义高效的流分析应用程序。而且，Flink Table API 和 SQL 得到了有效的优化，它集成了许多查询优化和优化的运算符实现。但是并非默认情况下会启用所有优化，因此对于某些工作负载，可以通过打开某些选项来提高性能。

在此页面中，我们将介绍一些有用的优化选项以及流聚合的内部原理，这将在某些情况下带来很大的改进。

注意：当前，仅 Blink 计划程序支持此页面中提到的优化选项。

注意：当前，仅对无边界聚合支持流聚合优化。将来将支持窗口聚合的优化。

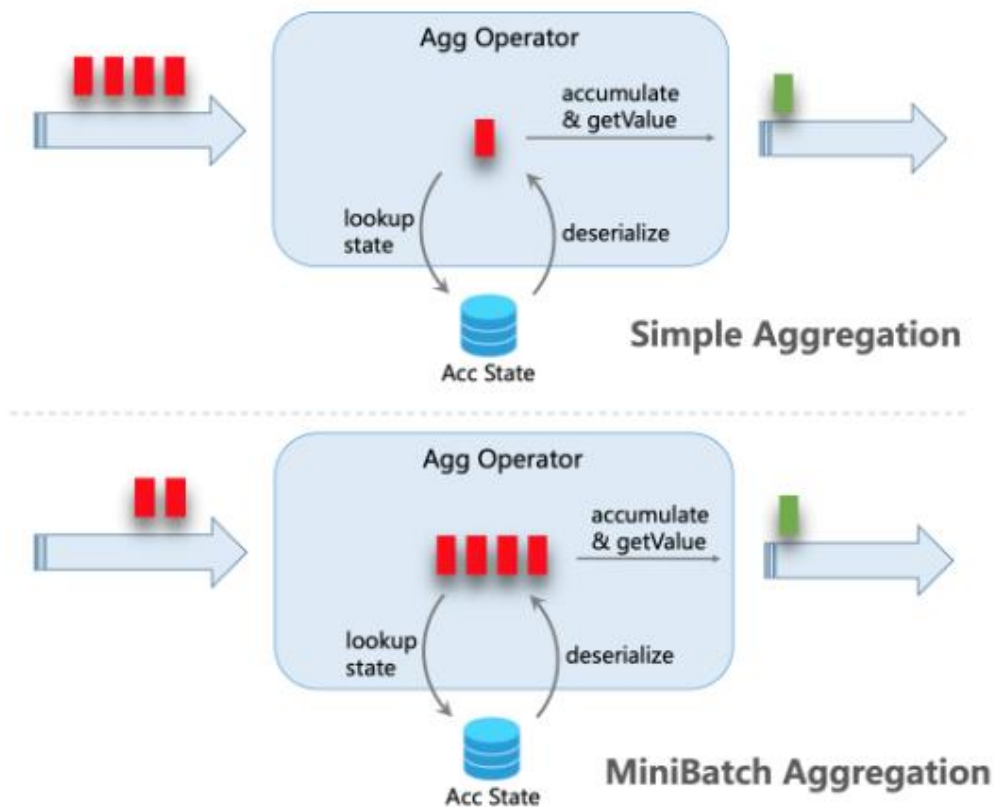
- 小批量聚合
- 局部全局聚合
- 分割不同的聚合
- 在不同的聚合上使用 FILTER 修饰符

默认情况下，无界聚合运算符一个一个地处理输入记录，即（1）从状态读取累加器，（2）将记录累加/缩回到累加器，（3）将累加器写回到状态，（4）下一条记录将从（1）重新进行处理。此处理模式可能会增加 StateBackend 的开销（尤其是对于 RocksDB StateBackend）。此外，生产中非常常见的数据偏斜会使问题恶化，并使工作容易承受背压情况。

小批量聚合

小型批处理聚合的核心思想是将一组输入缓存在聚合运算符内部的缓冲区中。当触发输入以进行处理时，每个键只需一个操作即可访问状态。这样可以大大减少状态开销并获得更好的吞吐量。但是，这可能会增加一些延迟，因为它会缓冲一些记录而不是立即处理它们。这是吞吐量和延迟之间的权衡。

下图说明了小批量聚合如何减少状态操作。



MiniBatch 优化默认情况下处于禁用状态。为了使这种优化，您应该设置 `table.exec.mini-batch.enabled`, `table.exec.mini-batch.allow-latency` 和 `table.exec.mini-batch.size`。请参阅[配置](#)页面以获取更多详细信息。

以下示例显示如何启用这些选项。

```
// instantiate table environment
val tEnv: TableEnvironment = ...

// access flink configuration
val configuration = tEnv.getConfig().getConfiguration()
// set low-level key-value options
configuration.setString("table.exec.mini-batch.enabled", "true") //
enable mini-batch optimization
configuration.setString("table.exec.mini-batch.allow-latency", "5 s")
// use 5 seconds to buffer input records
configuration.setString("table.exec.mini-batch.size", "5000") // the
maximum number of records can be buffered by each aggregate operator
task
```

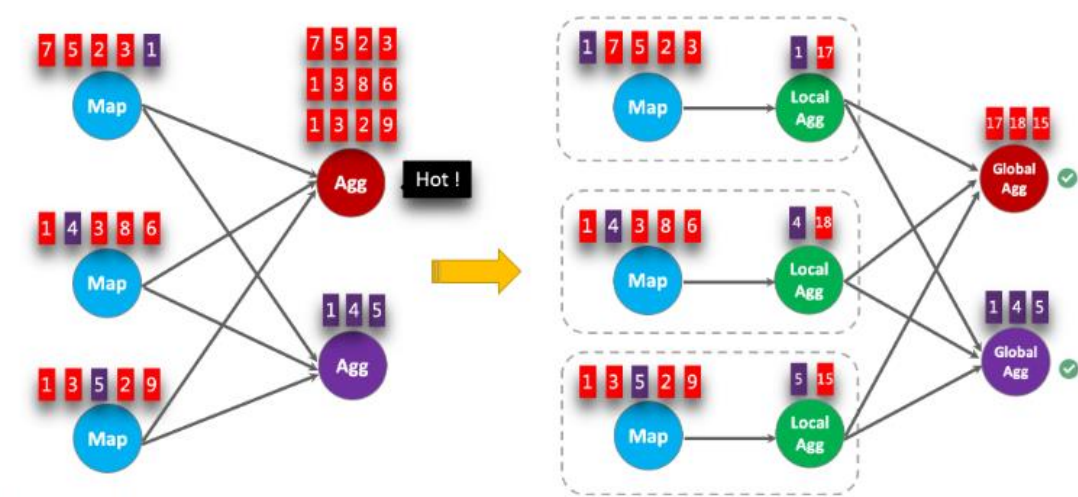
局部全局聚合

提出将局部聚合分为两个阶段来解决数据倾斜问题，即先在上游进行局部聚合，然后在下游进行全局聚合，这类似于 MapReduce 中的 Combine + Reduce 模式。例如，考虑以下 SQL:

```
SELECT color, sum(id)
FROM T
GROUP BY color
```

数据流中的记录可能会倾斜，因此聚合运算符的某些实例会比其他实例处理更多的记录，这会导致热点。本地聚合可以帮助将具有相同密钥的一定数量的输入累加到单个累加器中。全局汇总将仅接收减少的累加器，而不是大量的原始输入。这可以大大减少网络改组 and 状态访问的成本。每次本地聚合累积的输入数量基于最小批处理间隔。这意味着本地-全局聚合取决于启用了小批量优化。

下图显示了本地全局聚合如何提高性能。



以下示例显示了如何启用本地全局聚合。

```
// instantiate table environment
val tEnv: TableEnvironment = ...

// access flink configuration
val configuration = tEnv.getConfig().getConfiguration()
// set low-level key-value options
configuration.setString("table.exec.mini-batch.enabled", "true") //
local-global aggregation depends on mini-batch is enabled
configuration.setString("table.exec.mini-batch.allow-latency", "5 s")
```



```
configuration.setString("table.exec.mini-batch.size", "5000")
configuration.setString("table.optimizer.agg-phase-strategy",
    "TWO_PHASE") // enable two-phase, i.e. local-global aggregation
```



分割不同的聚合

局部全局优化可有效消除常规聚合的数据偏斜，例如 SUM，COUNT，MAX，MIN，AVG。但是，在处理不同的聚合时，其性能并不令人满意。

例如，如果我们要分析今天有多少唯一用户登录。我们可能有以下查询：

```
SELECT day, COUNT(DISTINCT user_id)
FROM T
GROUP BY day
```

如果 **distinct key**（即 **user_id**）的值稀疏，则 **COUNT DISTINCT** 不能减少记录。即使启用了局部全局优化，它也无济于事。因为累加器仍包含几乎所有原始记录，并且全局聚合将成为瓶颈（大多数繁重的累加器由一项任务处理，即在同一天）。

此优化的想法是将不同的聚合（例如 **COUNT(DISTINCT col)**）分为两个级别。第一次聚合由组密钥和其他存储桶密钥混洗。使用来计算存储桶密钥 **HASH_CODE(distinct_key) % BUCKET_NUM**。BUCKET_NUM 默认为 **1024**，可以通过 **table.optimizer.distinct-agg.split.bucket-num** 选项配置。第二次聚合由原始组密钥改组，并用于 SUM 聚合来自不同存储桶的 **COUNT DISTINCT** 值。由于相同的唯一键将仅在同一存储桶中计算，因此转换是等效的。存储桶密钥充当附加组密钥的角色，以分担组密钥中的热点负担。存储桶关键字使工作具有可伸缩性，以解决不同聚合中的数据偏斜/热点。

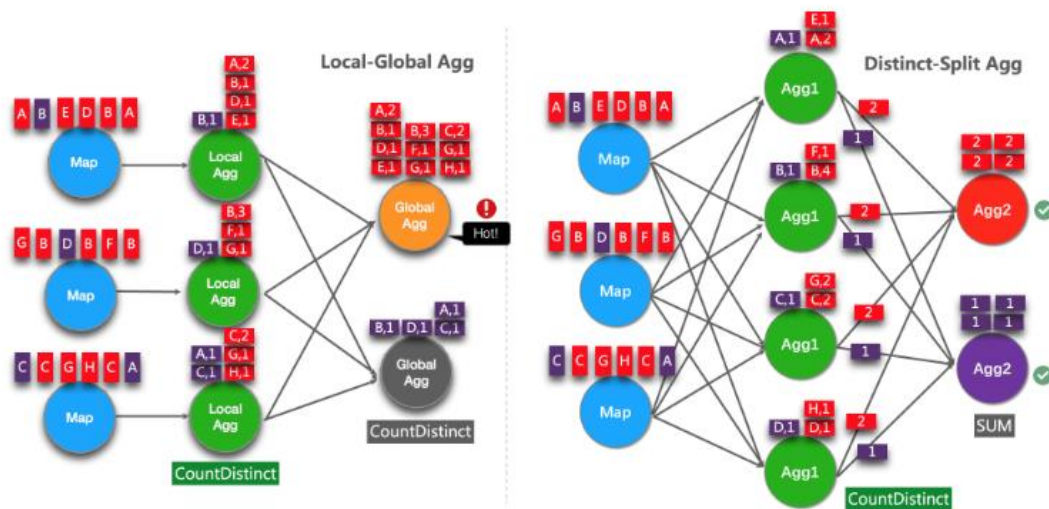
拆分非重复聚合后，上述查询将自动重写为以下查询：



```
SELECT day, SUM(cnt)
FROM (
    SELECT day, COUNT(DISTINCT user_id) as cnt
    FROM T
    GROUP BY day, MOD(HASH_CODE(user_id), 1024)
)
GROUP BY day
```



下图显示了拆分的非重复聚合如何提高性能（假设颜色代表天，字母代表 **user_id**）。



注意：上面是最简单的示例，可以从此优化中受益。除此之外，Flink 支持分裂更复杂的聚集查询，例如，一个以上的具有不同的不同密钥（例如不同的集合 `COUNT(DISTINCT a)`，`SUM(DISTINCT b)`），与其他非重复的聚合工作（例如 `SUM`，`MAX`，`MIN`，`COUNT`）。

注意：但是，当前，拆分优化不支持包含用户定义的 `AggregateFunction` 的聚合。

以下示例显示如何启用拆分非重复聚合优化。

```
// instantiate table environment
val tEnv: TableEnvironment = ...

tEnv.getConfig          // access high-level configuration
  .getConfiguration     // set low-level key-value options
  .setString("table.optimizer.distinct-agg.split.enabled", "true")
// enable distinct agg split
```

在不同的聚合上使用 **FILTER** 修饰符

在某些情况下，用户可能需要从不同维度计算 UV（唯一访客）的数量，例如 Android 的 UV，iPhone 的 UV，Web 的 UV 和总 UV。许多用户将选择 `CASE WHEN` 支持此功能，例如：

```
SELECT
  day,
  COUNT(DISTINCT user_id) AS total_uv,
```

```
COUNT(DISTINCT CASE WHEN flag IN ('android', 'iphone') THEN user_id
ELSE NULL END) AS app_uv,
COUNT(DISTINCT CASE WHEN flag IN ('wap', 'other') THEN user_id ELSE
NULL END) AS web_uv
FROM T
GROUP BY day
```



但是，在这种情况下，建议使用 **FILTER** 语法而不是 **CASE WHEN**。因为 **FILTER** 它更符合 **SQL** 标准，并且将获得更多的性能改进。 **FILTER** 是用于聚合函数的修饰符，用于限制聚合中使用的值。将上面的示例替换为 **FILTER** 修饰符，如下所示：

```
SELECT
  day,
  COUNT(DISTINCT user_id) AS total_uv,
  COUNT(DISTINCT user_id) FILTER (WHERE flag IN ('android', 'iphone'))
AS app_uv,
  COUNT(DISTINCT user_id) FILTER (WHERE flag IN ('wap', 'other')) AS
web_uv
FROM T
GROUP BY day
```

Flink SQL 优化器可以识别同一唯一键上的不同过滤器参数。例如，在上面的示例中，所有三个 **COUNT DISTINCT** 都在 **user_id** 列上。然后 **Flink** 可以只使用一个共享状态实例，而不是三个状态实例，以减少状态访问和状态大小。在某些工作负载中，这可以显着提高性能。