# Flink-Table与SQL

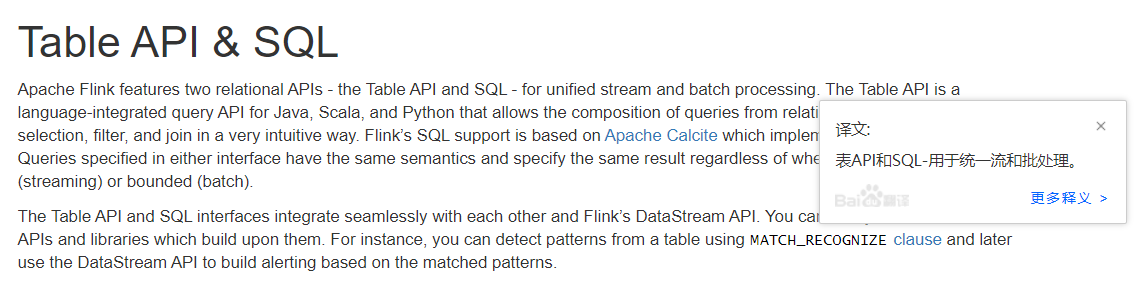
## 课程目标

* 了解Flink Table&SQL发展历史
* 掌握Flink Table&SQL进行批处理开发
* 掌握Flink Table&SQL进行流处理开发
* 掌握Flink Table&SQL常用函数

## Table API & SQL 介绍

### 为什么需要Table API & SQL

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/>



Flink的Table模块包括 Table API 和 SQL：

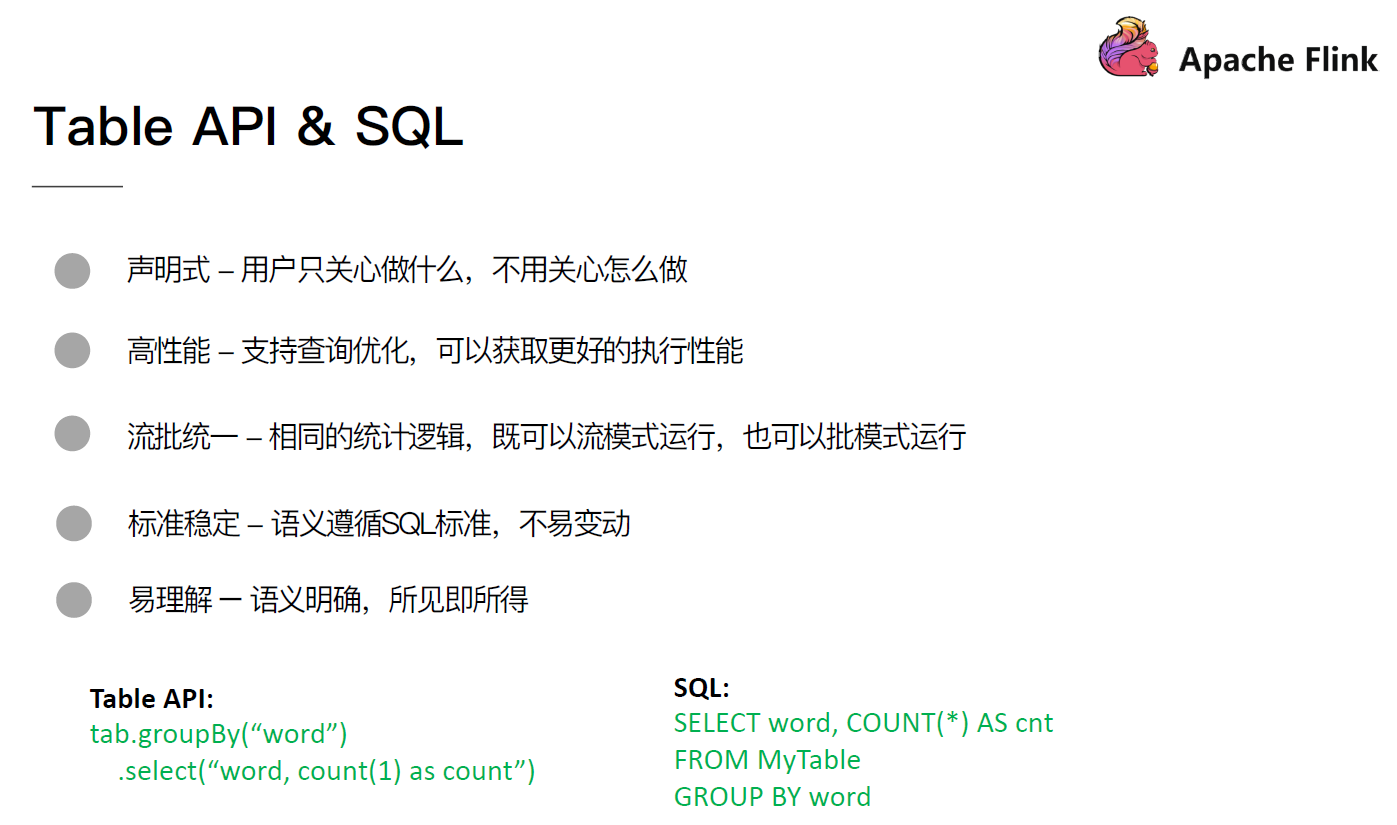
Table API 是一种类SQL的API，通过Table API，用户可以像操作表一样操作数据，非常直观和方便

SQL作为一种声明式语言，有着标准的语法和规范，用户可以不用关心底层实现即可进行数据的处理，非常易于上手

Flink Table API 和 SQL 的实现上有80%左右的代码是公用的。作为一个流批统一的计算引擎，Flink 的 Runtime 层是统一的。

* Table API & SQL的特点

Flink之所以选择将 Table API & SQL 作为未来的核心 API，是因为其具有一些非常重要的特点：



1. 声明式:属于设定式语言，用户只要表达清楚需求即可，不需要了解底层执行；

2. 高性能:可优化，内置多种查询优化器，这些查询优化器可为 SQL 翻译出最优执行计划；

3. 简单易学:易于理解，不同行业和领域的人都懂，学习成本较低；

4. 标准稳定:语义遵循SQL标准，非常稳定，在数据库 30 多年的历史中，SQL 本身变化较少；

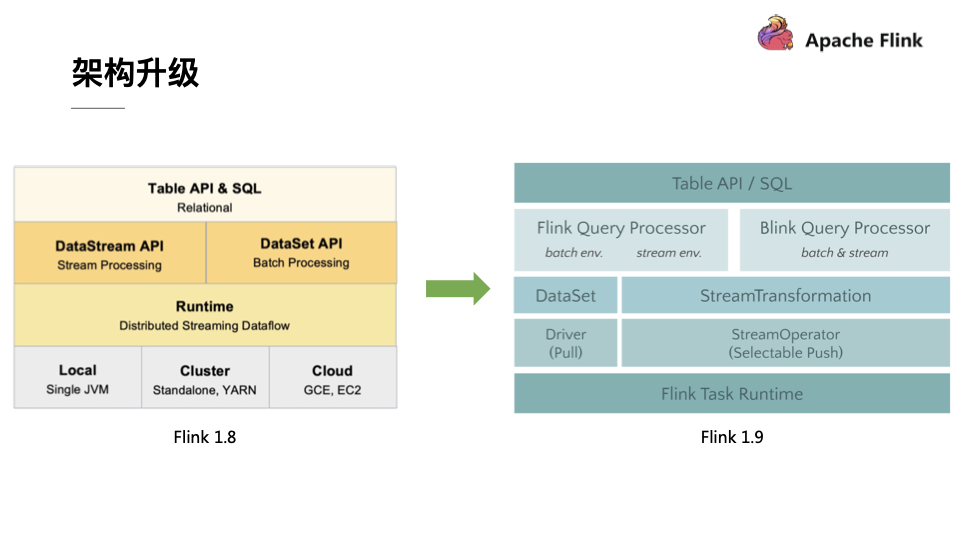
5. 流批统一:可以做到API层面上流与批的统一，相同的SQL逻辑，既可流模式运行，也可批模式运行，Flink底层Runtime本身就是一个流与批统一的引擎

### Table API& SQL发展历程

* 架构升级

自 2015 年开始，阿里巴巴开始调研开源流计算引擎，最终决定基于 Flink 打造新一代计算引擎，针对 Flink 存在的不足进行优化和改进，并且在 2019 年初将最终代码开源，也就是Blink。Blink 在原来的 Flink 基础上最显著的一个贡献就是 Flink SQL 的实现。随着版本的不断更新，API 也出现了很多不兼容的地方。

在 Flink 1.9 中，Table 模块迎来了核心架构的升级，引入了阿里巴巴Blink团队贡献的诸多功能



在Flink 1.9 之前，Flink API 层 一直分为DataStream API 和 DataSet API，Table API & SQL 位于 DataStream API 和 DataSet API 之上。可以看处流处理和批处理有各自独立的api (流处理DataStream，批处理DataSet)。而且有不同的执行计划解析过程，codegen过程也完全不一样，完全没有流批一体的概念，面向用户不太友好。

在Flink1.9之后新的架构中，有两个查询处理器：Flink Query Processor，也称作Old Planner和Blink Query Processor，也称作Blink Planner。为了兼容老版本Table及SQL模块，插件化实现了Planner，Flink原有的Flink Planner不变，后期版本会被移除。新增加了Blink Planner，新的代码及特性会在Blink planner模块上实现。批或者流都是通过解析为Stream Transformation来实现的，不像Flink Planner，批是基于Dataset，流是基于DataStream。

* 查询处理器的选择

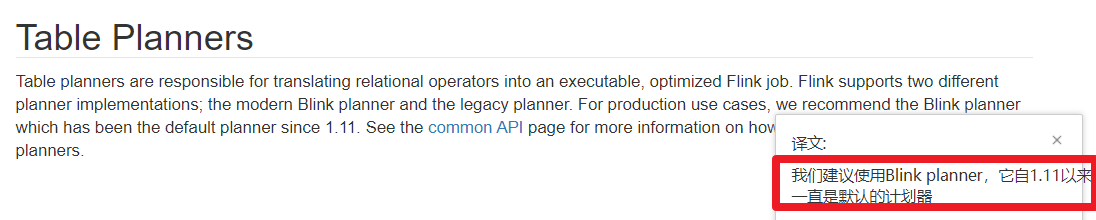
查询处理器是 Planner 的具体实现，通过parser、optimizer、codegen(代码生成技术)等流程将 Table API & SQL作业转换成 Flink Runtime 可识别的 Transformation DAG，最终由 Flink Runtime 进行作业的调度和执行。

Flink Query Processor查询处理器针对流计算和批处理作业有不同的分支处理，流计算作业底层的 API 是 DataStream API， 批处理作业底层的 API 是 DataSet API

Blink Query Processor查询处理器则实现流批作业接口的统一，底层的 API 都是Transformation，这就意味着我们和Dataset完全没有关系了

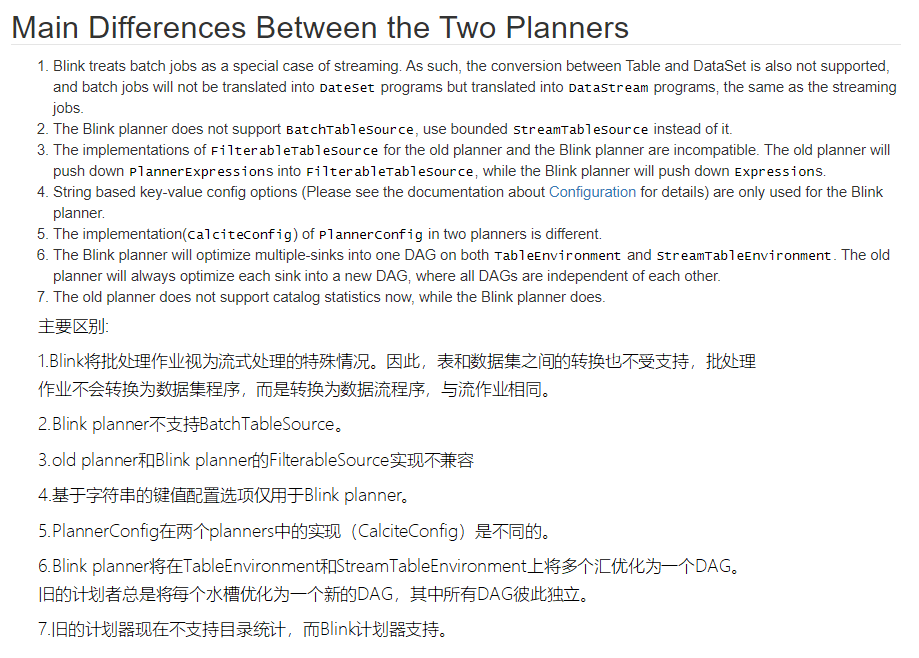
Flink1.11之后Blink Query Processor查询处理器已经是默认的了

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/>



* 了解-Blink planner和Flink Planner具体区别如下：

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/common.html>



### 注意：

<https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/table/common.html>

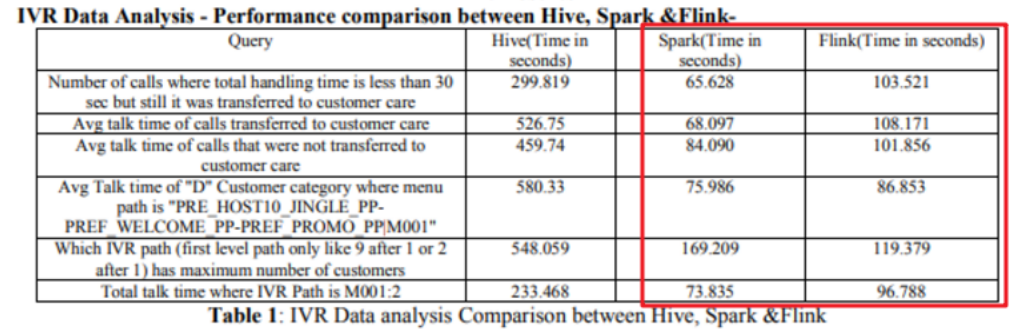
* API稳定性



* 性能对比

注意：目前FlinkSQL性能不如SparkSQL，未来FlinkSQL可能会越来越好

下图是Hive、Spark、Flink的SQL执行速度对比：



## 案例准备

### 依赖

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/>

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-api-scala-bridge\_2.12</**artifactId**>  <**version**>${flink.version}</**version**>  <**scope**>provided</**scope**> </**dependency**> <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-api-java-bridge\_2.12</**artifactId**>  <**version**>${flink.version}</**version**>  <**scope**>provided</**scope**> </**dependency**> *<!-- flink执行计划,这是1.9版本之前的-->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-planner\_2.12</**artifactId**>  <**version**>${flink.version}</**version**> </**dependency**> *<!-- blink执行计划,1.11+默认的-->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-planner-blink\_2.12</**artifactId**>  <**version**>${flink.version}</**version**>  <**scope**>provided</**scope**> </**dependency**> <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-common</**artifactId**>  <**version**>${flink.version}</**version**>  <**scope**>provided</**scope**> </**dependency**> |

● flink-table-common：这个包中主要是包含 Flink Planner 和 Blink Planner一些共用的代码。

● flink-table-api-java：这部分是用户编程使用的 API，包含了大部分的 API。

● flink-table-api-scala：这里只是非常薄的一层，仅和 Table API 的 Expression 和 DSL 相关。

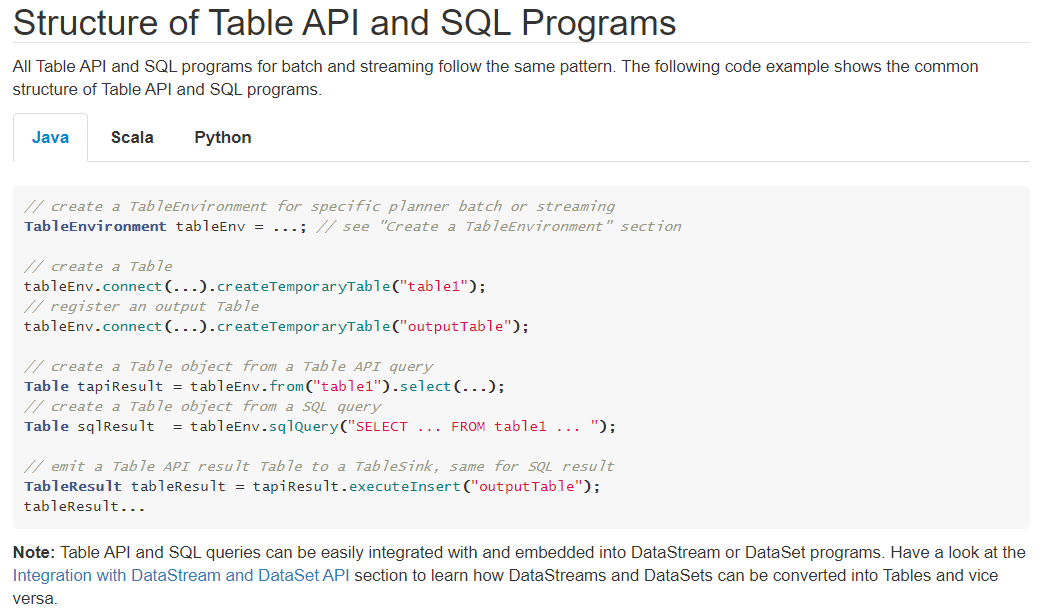
● 两个 Planner：flink-table-planner 和 flink-table-planner-blink。

● 两个 Bridge：flink-table-api-scala-bridge 和 flink-table-api-java-bridge，

Flink Planner 和 Blink Planner 都会依赖于具体的 JavaAPI，也会依赖于具体的 Bridge，通过 Bridge 可以将 API 操作相应的转化为Scala 的 DataStream、DataSet，或者转化为 JAVA 的 DataStream 或者Data Set

### 程序结构

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/common.html#structure-of-table-api-and-sql-programs>



|  |
| --- |
| *// create a TableEnvironment for specific planner batch or streaming*  **TableEnvironment** tableEnv **=** **...;** *// see "Create a TableEnvironment" section*  *// create a Table*  tableEnv**.**connect**(...).**createTemporaryTable**(**"table1"**);**  *// register an output Table*  tableEnv**.**connect**(...).**createTemporaryTable**(**"outputTable"**);**  *// create a Table object from a Table API query*  **Table** tapiResult **=** tableEnv**.**from**(**"table1"**).**select**(...);**  *// create a Table object from a SQL query*  **Table** sqlResult **=** tableEnv**.**sqlQuery**(**"SELECT ... FROM table1 ... "**);**  *// emit a Table API result Table to a TableSink, same for SQL result*  **TableResult** tableResult **=** tapiResult**.**executeInsert**(**"outputTable"**);**  tableResult**...** |

### API

#### 获取环境

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/common.html#create-a-tableenvironment>

|  |
| --- |
| *// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *// FLINK STREAMING QUERY*  *// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment**;**  import org.apache.flink.table.api.EnvironmentSettings**;**  import org.apache.flink.table.api.bridge.java.StreamTableEnvironment**;**  **EnvironmentSettings** fsSettings **=** **EnvironmentSettings.**newInstance**().**useOldPlanner**().**inStreamingMode**().**build**();**  **StreamExecutionEnvironment** fsEnv **=** **StreamExecutionEnvironment.**getExecutionEnvironment**();**  **StreamTableEnvironment** fsTableEnv **=** **StreamTableEnvironment.**create**(**fsEnv**,** fsSettings**);**  *// or TableEnvironment fsTableEnv = TableEnvironment.create(fsSettings);*  *// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *// FLINK BATCH QUERY*  *// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  import org.apache.flink.api.java.ExecutionEnvironment**;**  import org.apache.flink.table.api.bridge.java.BatchTableEnvironment**;**  **ExecutionEnvironment** fbEnv **=** **ExecutionEnvironment.**getExecutionEnvironment**();**  **BatchTableEnvironment** fbTableEnv **=** **BatchTableEnvironment.**create**(**fbEnv**);**  *// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *// BLINK STREAMING QUERY*  *// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment**;**  import org.apache.flink.table.api.EnvironmentSettings**;**  import org.apache.flink.table.api.bridge.java.StreamTableEnvironment**;**  **StreamExecutionEnvironment** bsEnv **=** **StreamExecutionEnvironment.**getExecutionEnvironment**();**  **EnvironmentSettings** bsSettings **=** **EnvironmentSettings.**newInstance**().**useBlinkPlanner**().**inStreamingMode**().**build**();**  **StreamTableEnvironment** bsTableEnv **=** **StreamTableEnvironment.**create**(**bsEnv**,** bsSettings**);**  *// or TableEnvironment bsTableEnv = TableEnvironment.create(bsSettings);*  *// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *// BLINK BATCH QUERY*  *// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  import org.apache.flink.table.api.EnvironmentSettings**;**  import org.apache.flink.table.api.TableEnvironment**;**  **EnvironmentSettings** bbSettings **=** **EnvironmentSettings.**newInstance**().**useBlinkPlanner**().**inBatchMode**().**build**();**  **TableEnvironment** bbTableEnv **=** **TableEnvironment.**create**(**bbSettings**);** |

#### 创建表

|  |
| --- |
| *// get a TableEnvironment*  **TableEnvironment** tableEnv **=** **...;** *// see "Create a TableEnvironment" section*  *// table is the result of a simple projection query*  **Table** projTable **=** tableEnv**.**from**(**"X"**).**select**(...);**  *// register the Table projTable as table "projectedTable"*  tableEnv**.**createTemporaryView**(**"projectedTable"**,** projTable**);** |

|  |
| --- |
| tableEnvironment  **.**connect**(...)**  **.**withFormat**(...)**  **.**withSchema**(...)**  **.**inAppendMode**()**  **.**createTemporaryTable**(**"MyTable"**)** |

#### 查询表

* Table API

|  |
| --- |
| *// get a TableEnvironment*  **TableEnvironment** tableEnv **=** **...;** *// see "Create a TableEnvironment" section*  *// register Orders table*  *// scan registered Orders table*  **Table** orders **=** tableEnv**.**from**(**"Orders"**);***// compute revenue for all customers from France*  **Table** revenue **=** orders  **.**filter**(**$**(**"cCountry"**)**  **.**isEqual**(**"FRANCE"**))**  **.**groupBy**(**$**(**"cID"**),** $**(**"cName"**)**  **.**select**(**$**(**"cID"**),** $**(**"cName"**),** $**(**"revenue"**)**  **.**sum**()**  **.**as**(**"revSum"**));**  *// emit or convert Table*  *// execute query* |

* SQL

|  |
| --- |
| *// get a TableEnvironment*  **TableEnvironment** tableEnv **=** **...;** *// see "Create a TableEnvironment" section*  *// register Orders table*  *// compute revenue for all customers from France*  **Table** revenue **=** tableEnv**.**sqlQuery**(**  "SELECT cID, cName, SUM(revenue) AS revSum " **+**  "FROM Orders " **+**  "WHERE cCountry = 'FRANCE' " **+**  "GROUP BY cID, cName"  **);**  *// emit or convert Table*  *// execute query* |

|  |
| --- |
| *// get a TableEnvironment*  **TableEnvironment** 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**.**executeSql**(**  "INSERT INTO RevenueFrance " **+**  "SELECT cID, cName, SUM(revenue) AS revSum " **+**  "FROM Orders " **+**  "WHERE cCountry = 'FRANCE' " **+**  "GROUP BY cID, cName"  **);** |

#### 写出表

|  |
| --- |
| *// get a TableEnvironment*  **TableEnvironment** tableEnv **=** **...;** *// see "Create a TableEnvironment" section*  *// create an output Table*  **final** **Schema** schema **=** **new** **Schema()**  **.**field**(**"a"**,** **DataTypes.**INT**())**  **.**field**(**"b"**,** **DataTypes.**STRING**())**  **.**field**(**"c"**,** **DataTypes.**BIGINT**());**  tableEnv**.**connect**(new** **FileSystem().**path**(**"/path/to/file"**))**  **.**withFormat**(new** **Csv().**fieldDelimiter**(**'|'**).**deriveSchema**())**  **.**withSchema**(**schema**)**  **.**createTemporaryTable**(**"CsvSinkTable"**);**  *// compute a result Table using Table API operators and/or SQL queries*  **Table** result **=** **...**  *// emit the result Table to the registered TableSink*  result**.**executeInsert**(**"CsvSinkTable"**);** |

#### 与DataSet/DataStream集成

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/common.html#integration-with-datastream-and-dataset-api>



* Create a View from a DataStream or DataSet

|  |
| --- |
| *// get StreamTableEnvironment*  *// registration of a DataSet in a BatchTableEnvironment is equivalent*  **StreamTableEnvironment** tableEnv **=** **...;**  *// see "Create a TableEnvironment" section*  **DataStream<Tuple2<Long,** **String>>** stream **=** **...**  *// register the DataStream as View "myTable" with fields "f0", "f1"*  tableEnv**.**createTemporaryView**(**"myTable"**,** stream**);**  *// register the DataStream as View "myTable2" with fields "myLong", "myString"*  tableEnv**.**createTemporaryView**(**"myTable2"**,** stream**,** $**(**"myLong"**),** $**(**"myString"**));** |

* Convert a DataStream or DataSet into a Table

|  |
| --- |
| *// get StreamTableEnvironment// registration of a DataSet in a BatchTableEnvironment is equivalent*  **StreamTableEnvironment** tableEnv **=** **...;**  *// see "Create a TableEnvironment" section*  **DataStream<Tuple2<Long,** **String>>** stream **=** **...**  *// Convert the DataStream into a Table with default fields "f0", "f1"*  **Table** table1 **=** tableEnv**.**fromDataStream**(**stream**);**  *// Convert the DataStream into a Table with fields "myLong", "myString"*  **Table** table2 **=** tableEnv**.**fromDataStream**(**stream**,** $**(**"myLong"**),** $**(**"myString"**));** |

* Convert a Table into a DataStream or DataSet
  + Convert a Table into a DataStream

Append Mode: This mode can only be used if the dynamic Table is only modified by INSERT changes, i.e, it is append-only and previously emitted results are never updated.

追加模式：只有当动态表仅通过插入更改进行修改时，才能使用此模式，即，它是仅追加模式，并且以前发出的结果从不更新。

Retract Mode: This mode can always be used. It encodes INSERT and DELETE changes with a boolean flag.

撤回模式：此模式始终可用。它使用布尔标志对插入和删除更改进行编码。

|  |
| --- |
| *// get StreamTableEnvironment.*  **StreamTableEnvironment** tableEnv **=** **...;** *// see "Create a TableEnvironment" section*  *// Table with two fields (String name, Integer age)*  **Table** table **=** **...**  *// convert the Table into an append DataStream of Row by specifying the class*  **DataStream<Row>** dsRow **=** tableEnv**.**toAppendStream**(**table**,** **Row.**class**);**  *// convert the Table into an append DataStream of Tuple2<String, Integer>*  *// via a TypeInformation*  **TupleTypeInfo<Tuple2<String,** **Integer>>** tupleType **=** **new** **TupleTypeInfo<>(**  **Types.**STRING**(),**  **Types.**INT**());**  **DataStream<Tuple2<String,** **Integer>>** dsTuple **=**  tableEnv**.**toAppendStream**(**table**,** tupleType**);**  *// convert the Table into a retract DataStream of Row.*  *// A retract stream of type X is a DataStream<Tuple2<Boolean, X>>.*  *// The boolean field indicates the type of the change.*  *// True is INSERT, false is DELETE.*  **DataStream<Tuple2<Boolean,** **Row>>** retractStream **=**  tableEnv**.**toRetractStream**(**table**,** **Row.**class**);** |

* Convert a Table into a DataSet

|  |
| --- |
| *// get BatchTableEnvironment*  **BatchTableEnvironment** tableEnv **=** **BatchTableEnvironment.**create**(**env**);**  *// Table with two fields (String name, Integer age)*  **Table** table **=** **...**  *// convert the Table into a DataSet of Row by specifying a class*  **DataSet<Row>** dsRow **=** tableEnv**.**toDataSet**(**table**,** **Row.**class**);**  *// convert the Table into a DataSet of Tuple2<String, Integer> via a TypeInformation***TupleTypeInfo<Tuple2<String,** **Integer>>** tupleType **=** **new** **TupleTypeInfo<>(**  **Types.**STRING**(),**  **Types.**INT**());**  **DataSet<Tuple2<String,** **Integer>>** dsTuple **=**  tableEnv**.**toDataSet**(**table**,** tupleType**);** |

#### TableAPI

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/tableApi.html>

#### SQLAPI

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/sql/>

### 相关概念

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/streaming/dynamic_tables.html>

#### Dynamic Tables & Continuous Queries

在Flink中，它把针对无界流的表称之为Dynamic Table（动态表）。它是Flink Table API和SQL的核心概念。顾名思义，它表示了Table是不断变化的。

我们可以这样来理解，当我们用Flink的API，建立一个表，其实把它理解为建立一个逻辑结构，这个逻辑结构需要映射到数据上去。Flink source源源不断的流入数据，就好比每次都往表上新增一条数据。表中有了数据，我们就可以使用SQL去查询了。要注意一下，流处理中的数据是只有新增的，所以看起来数据会源源不断地添加到表中。

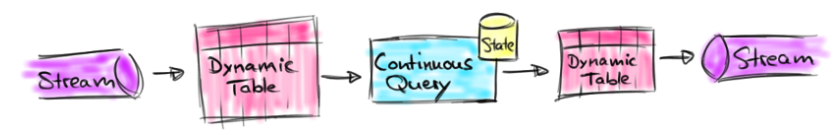
动态表也是一种表，既然是表，就应该能够被查询。我们来回想一下原先我们查询表的场景。

打开编译工具，编写一条SQL语句

* 将SQL语句放入到mysql的终端执行
* 查看结果
* 再编写一条SQL语句
* 再放入到终端执行
* 再查看结果

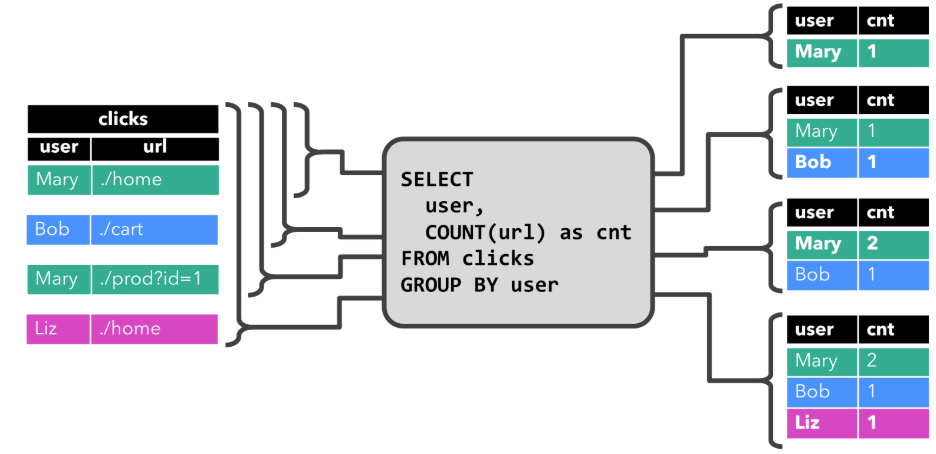
…..如此反复

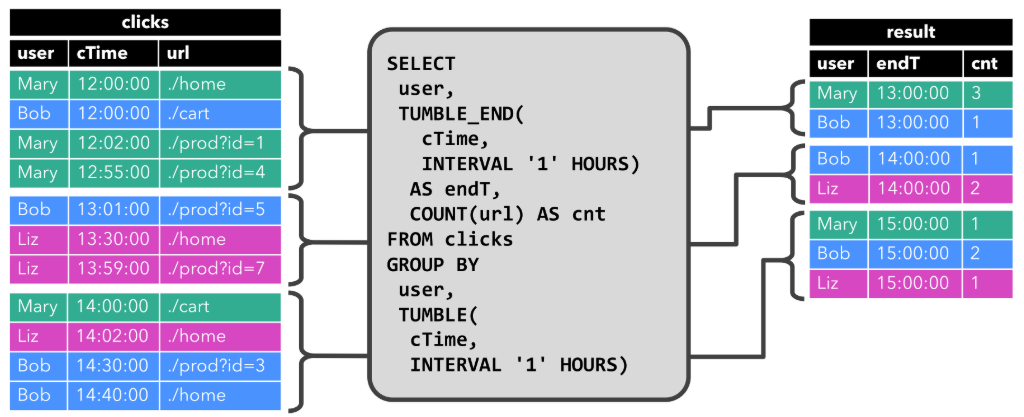
而针对动态表，Flink的source端肯定是源源不断地会有数据流入，然后我们基于这个数据流建立了一张表，再编写SQL语句查询数据，进行处理。这个SQL语句一定是不断地执行的。而不是只执行一次。注意：针对流处理的SQL绝对不会像批式处理一样，执行一次拿到结果就完了。而是会不停地执行，不断地查询获取结果处理。所以，官方给这种查询方式取了一个名字，叫Continuous Query，中文翻译过来叫连续查询。而且每一次查询出来的数据也是不断变化的。



这是一个非常简单的示意图。该示意图描述了：我们通过建立动态表和连续查询来实现在无界流中的SQL操作。大家也可以看到，在Continuous上面有一个State，表示查询出来的结果会存储在State中，再下来Flink最终还是使用流来进行处理。

所以，我们可以理解为Flink的Table API和SQL，是一个逻辑模型，通过该逻辑模型可以让我们的数据处理变得更加简单。





#### Table to Stream Conversion

* 表中的Update和Delete

我们前面提到的表示不断地Append，表的数据是一直累加的，因为表示对接Source的，Source是不会有update的。但如果我们编写了一个SQL。这个SQL看起来是这样的：

|  |
| --- |
| SELECT user, sum(money) FROM order GROUP BY user; |

当执行一条SQL语句之后，这条语句的结果还是一个表，因为在Flink中执行的SQL是Continuous Query，这个表的数据是不断变化的。新创建的表存在Update的情况。仔细看下下面的示例，例如：

第一条数据，张三,2000，执行这条SQL语句的结果是，张三,2000

第二条数据，李四,1500，继续执行这条SQL语句，结果是，张三,2000 | 李四,1500

第三条数据，张三,300，继续执行这条SQL语句，结果是，张三,2300 | 李四,1500

….

大家发现了吗，现在数据结果是有Update的。张三一开始是2000，但后面变成了2300。

那还有删除的情况吗？有的。看一下下面这条SQL语句：

|  |
| --- |
| SELECT t1.`user`, SUM(t1.`money`) FROM t\_order t1  WHERE  NOT EXISTS (SELECT T2.`user`AS TOTAL\_MONEY FROM t\_order t2 WHERE T2.`user` = T1.`user` GROUP BY t2.`user` HAVING SUM(T2.`money`) > 3000)  GROUP BY t1.`user`GROUP BY t1.`user` |

第一条数据，张三,2000，执行这条SQL语句的结果是，张三,2000

第二条数据，李四,1500，继续执行这条SQL语句，结果是，张三,2000 | 李四,1500

第三条数据，张三,300，继续执行这条SQL语句，结果是，张三,2300 | 李四,1500

第四条数据，张三,800，继续执行这条SQL语句，结果是，李四,1500

惊不惊喜？意不意外？

因为张三的消费的金额已经超过了3000，所以SQL执行完后，张三是被处理掉了。从数据的角度来看，它不就是被删除了吗？

通过上面的两个示例，给大家演示了，在Flink SQL中，对接Source的表都是Append-only的，不断地增加。执行一些SQL生成的表，这个表可能是要UPDATE的、也可能是要INSERT的。

* 对表的编码操作

我们前面说到过，表是一种逻辑结构。而Flink中的核心还是Stream。所以，Table最终还是会以Stream方式来继续处理。如果是以Stream方式处理，最终Stream中的数据有可能会写入到其他的外部系统中，例如：将Stream中的数据写入到MySQL中。

我们前面也看到了，表是有可能会UPDATE和DELETE的。那么如果是输出到MySQL中，就要执行UPDATE和DELETE语句了。而DataStream我们在学习Flink的时候就学习过了，DataStream是不能更新、删除事件的。

如果对表的操作是INSERT，这很好办，直接转换输出就好，因为DataStream数据也是不断递增的。但如果一个TABLE中的数据被UPDATE了、或者被DELETE了，如果用流来表达呢？因为流不可变的特征，我们肯定要对这种能够进行UPDATE/DELETE的TABLE做特殊操作。

我们可以针对每一种操作，INSERT/UPDATE/DELETE都用一个或多个经过编码的事件来表示。

例如：针对UPDATE，我们用两个操作来表达，[DELETE] 数据+ [INSERT]数据。也就是先把之前的数据删除，然后再插入一条新的数据。针对DELETE，我们也可以对流中的数据进行编码，[DELETE]数据。

总体来说，我们通过对流数据进行编码，也可以告诉DataStream的下游，[DELETE]表示发出MySQL的DELETE操作，将数据删除。用 [INSERT]表示插入新的数据。

* 将表转换为三种不同编码方式的流

Flink中的Table API或者SQL支持三种不同的编码方式。分别是：

Append-only流

Retract流

Upsert流

分别来解释下这三种流。

* Append-only流

跟INSERT操作对应。这种编码类型的流针对的是只会不断新增的Dynamic Table。这种方式好处理，不需要进行特殊处理，源源不断地往流中发送事件即可。

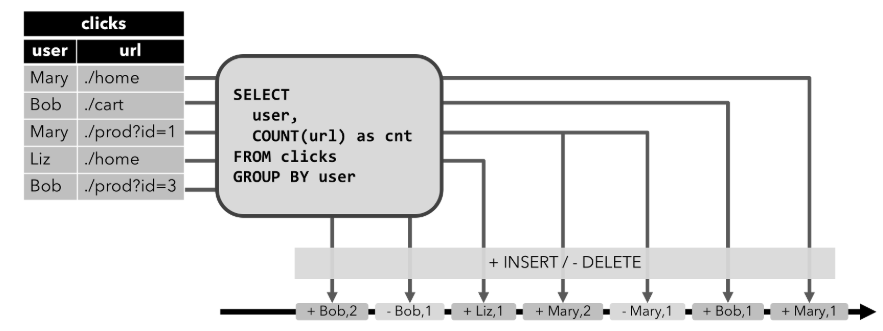
* Retract流

这种流就和Append-only不太一样。上面的只能处理INSERT，如果表会发生DELETE或者UPDATE，Append-only编码方式的流就不合适了。Retract流有几种类型的事件类型：

ADD MESSAGE：这种消息对应的就是INSERT操作。

RETRACT MESSAGE：直译过来叫取消消息。这种消息对应的就是DELETE操作。

我们可以看到通过ADD MESSAGE和RETRACT MESSAGE可以很好的向外部系统表达删除和插入操作。那如何进行UPDATE呢？好办！RETRACT MESSAGE + ADD MESSAGE即可。先把之前的数据进行删除，然后插入一条新的。完美~



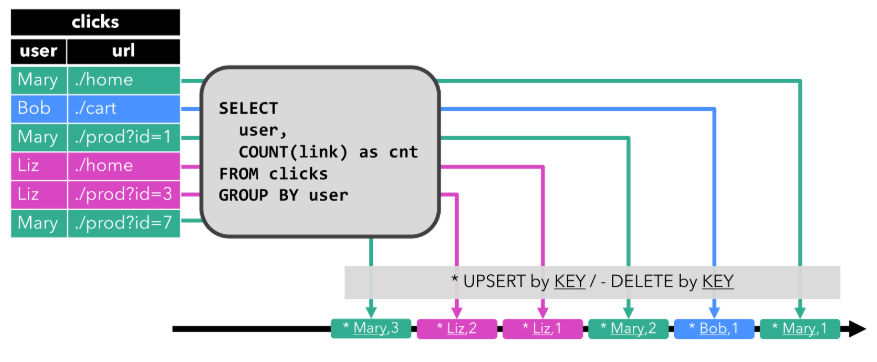
* Upsert流

前面我们看到的RETRACT编码方式的流，实现UPDATE是使用DELETE + INSERT模式的。大家想一下：在MySQL中我们更新数据的时候，肯定不会先DELETE掉一条数据，然后再插入一条数据，肯定是直接发出UPDATE语句执行更新。而Upsert编码方式的流，是能够支持Update的，这种效率更高。它同样有两种类型的消息：

UPSERT MESSAGE：这种消息可以表示要对外部系统进行Update或者INSERT操作

DELETE MESSAGE：这种消息表示DELETE操作。

Upsert流是要求必须指定Primary Key的，因为Upsert操作是要有Key的。Upsert流针对UPDATE操作用一个UPSERT MESSAGE就可以描述，所以效率会更高。



## 案例1

### 需求

将DataStream注册为Table和View并进行SQL统计

### 代码实现

|  |
| --- |
| **package** cn.itcast.sql;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.bridge.java.StreamTableEnvironment;  **import** java.util.Arrays;  **import static** org.apache.flink.table.api.Expressions.*$*;  */\*\*  \* Author itcast  \* Desc  \*/* **public class** FlinkSQL\_Table\_Demo01 {  **public static void** main(String[] args) **throws** Exception {  *//1.准备环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//EnvironmentSettings settings = EnvironmentSettings.newInstance().useBlinkPlanner().inStreamingMode().build();  //StreamTableEnvironment tEnv = StreamTableEnvironment.create(env, settings);* StreamTableEnvironment tEnv = StreamTableEnvironment.*create*(env);   *//2.Source* DataStream<Order> orderA = env.fromCollection(Arrays.*asList*(  **new** Order(1L, **"beer"**, 3),  **new** Order(1L, **"diaper"**, 4),  **new** Order(3L, **"rubber"**, 2)));   DataStream<Order> orderB = env.fromCollection(Arrays.*asList*(  **new** Order(2L, **"pen"**, 3),  **new** Order(2L, **"rubber"**, 3),  **new** Order(4L, **"beer"**, 1)));   *//3.注册表  // convert DataStream to Table* Table tableA = tEnv.fromDataStream(orderA, *$*(**"user"**), *$*(**"product"**), *$*(**"amount"**));  *// register DataStream as Table* tEnv.createTemporaryView(**"OrderB"**, orderB, *$*(**"user"**), *$*(**"product"**), *$*(**"amount"**));   *//4.执行查询* System.***out***.println(tableA);  *// union the two tables* Table resultTable = tEnv.sqlQuery(  **"SELECT \* FROM "** + tableA + **" WHERE amount > 2 "** +  **"UNION ALL "** +  **"SELECT \* FROM OrderB WHERE amount < 2"** );   *//5.输出结果* DataStream<Order> resultDS = tEnv.toAppendStream(resultTable, Order.**class**);  resultDS.print();   env.execute();  }   @Data  @NoArgsConstructor  @AllArgsConstructor  **public static class** Order {  **public** Long **user**;  **public** String **product**;  **public int amount**;  } } |

## 案例2

### 需求

使用SQL和Table两种方式对DataStream中的单词进行统计

### 代码实现-SQL

|  |
| --- |
| **package** cn.itcast.sql;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.bridge.java.StreamTableEnvironment;  **import static** org.apache.flink.table.api.Expressions.*$*;  */\*\*  \* Author itcast  \* Desc  \*/* **public class** FlinkSQL\_Table\_Demo02 {  **public static void** main(String[] args) **throws** Exception {  *//1.准备环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  StreamTableEnvironment tEnv = StreamTableEnvironment.*create*(env);   *//2.Source* DataStream<WC> input = env.fromElements(  **new** WC(**"Hello"**, 1),  **new** WC(**"World"**, 1),  **new** WC(**"Hello"**, 1)  );   *//3.注册表* tEnv.createTemporaryView(**"WordCount"**, input, *$*(**"word"**), *$*(**"frequency"**));   *//4.执行查询* Table resultTable = tEnv.sqlQuery(**"SELECT word, SUM(frequency) as frequency FROM WordCount GROUP BY word"**);   *//5.输出结果  //toAppendStream doesn't support consuming update changes which is produced by node GroupAggregate  //DataStream<WC> resultDS = tEnv.toAppendStream(resultTable, WC.class);* DataStream<Tuple2<Boolean, WC>> resultDS = tEnv.toRetractStream(resultTable, WC.**class**);   resultDS.print();   env.execute();  }   @Data  @NoArgsConstructor  @AllArgsConstructor  **public static class** WC {  **public** String **word**;  **public long frequency**;  } } |

### 代码实现-Table

|  |
| --- |
| **package** cn.itcast.sql;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.bridge.java.StreamTableEnvironment;  **import static** org.apache.flink.table.api.Expressions.*$*;  */\*\*  \* Author itcast  \* Desc  \*/* **public class** FlinkSQL\_Table\_Demo03 {  **public static void** main(String[] args) **throws** Exception {  *//1.准备环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  StreamTableEnvironment tEnv = StreamTableEnvironment.*create*(env);   *//2.Source* DataStream<WC> input = env.fromElements(  **new** WC(**"Hello"**, 1),  **new** WC(**"World"**, 1),  **new** WC(**"Hello"**, 1)  );   *//3.注册表* Table table = tEnv.fromDataStream(input);   *//4.执行查询* Table resultTable = table  .groupBy(*$*(**"word"**))  .select(*$*(**"word"**), *$*(**"frequency"**).sum().as(**"frequency"**))  .filter(*$*(**"frequency"**).isEqual(2));   *//5.输出结果* DataStream<Tuple2<Boolean, WC>> resultDS = tEnv.toRetractStream(resultTable, WC.**class**);   resultDS.print();   env.execute();  }   @Data  @NoArgsConstructor  @AllArgsConstructor  **public static class** WC {  **public** String **word**;  **public long frequency**;  } } |

## 案例3

### 需求

使用Flink SQL来统计5秒内 每个用户的 订单总数、订单的最大金额、订单的最小金额

也就是每隔5秒统计最近5秒的每个用户的订单总数、订单的最大金额、订单的最小金额

上面的需求使用流处理的Window的基于时间的滚动窗口就可以搞定!

那么接下来使用FlinkTable&SQL-API来实现

### 编码步骤

1.创建环境

2.使用自定义函数模拟实时流数据

3.设置事件时间和Watermaker

4.注册表

5.执行sql-可以使用sql风格或table风格(了解)

6.输出结果

7.触发执行

### 代码实现-方式1

|  |
| --- |
| **package** cn.itcast.sql;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.api.common.eventtime.WatermarkStrategy; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.source.RichSourceFunction; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.bridge.java.StreamTableEnvironment; **import** org.apache.flink.types.Row;  **import** java.time.Duration; **import** java.util.Random; **import** java.util.UUID; **import** java.util.concurrent.TimeUnit;  **import static** org.apache.flink.table.api.Expressions.*$*;  */\*\*  \* Author itcast  \* Desc  \*/* **public class** FlinkSQL\_Table\_Demo04 {  **public static void** main(String[] args) **throws** Exception {  *//1.准备环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  StreamTableEnvironment tEnv = StreamTableEnvironment.*create*(env);   *//2.Source* DataStreamSource<Order> orderDS = env.addSource(**new** RichSourceFunction<Order>() {  **private** Boolean **isRunning** = **true**;  @Override  **public void** run(SourceContext<Order> ctx) **throws** Exception {  Random random = **new** Random();  **while** (**isRunning**) {  Order order = **new** Order(UUID.*randomUUID*().toString(), random.nextInt(3), random.nextInt(101), System.*currentTimeMillis*());  TimeUnit.***SECONDS***.sleep(1);  ctx.collect(order);  }  }   @Override  **public void** cancel() {  **isRunning** = **false**;  }  });   *//3.Transformation* DataStream<Order> watermakerDS = orderDS  .assignTimestampsAndWatermarks(  WatermarkStrategy.<Order>*forBoundedOutOfOrderness*(Duration.*ofSeconds*(2))  .withTimestampAssigner((event, timestamp) -> event.getCreateTime())  );   *//4.注册表* tEnv.createTemporaryView(**"t\_order"**, watermakerDS,  *$*(**"orderId"**), *$*(**"userId"**), *$*(**"money"**), *$*(**"createTime"**).rowtime());   *//5.执行SQL* String sql = **"select "** +  **"userId,"** +  **"count(\*) as totalCount,"** +  **"max(money) as maxMoney,"** +  **"min(money) as minMoney "** +  **"from t\_order "** +  **"group by userId,"** +  **"tumble(createTime, interval '5' second)"**;   Table ResultTable = tEnv.sqlQuery(sql);   *//6.Sink  //将SQL的执行结果转换成DataStream再打印出来  //toAppendStream → 将计算后的数据append到结果DataStream中去  //toRetractStream → 将计算后的新的数据在DataStream原数据的基础上更新true或是删除false* DataStream<Tuple2<Boolean, Row>> resultDS = tEnv.toRetractStream(ResultTable, Row.**class**);  resultDS.print();   env.execute();   }   @Data  @AllArgsConstructor  @NoArgsConstructor  **public static class** Order {  **private** String **orderId**;  **private** Integer **userId**;  **private** Integer **money**;  **private** Long **createTime**;  } } |

toAppendStream → 将计算后的数据append到结果DataStream中去

toRetractStream → 将计算后的新的数据在DataStream原数据的基础上更新true或是删除false

### 代码实现-方式2

|  |
| --- |
| **package** cn.itcast.sql;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.api.common.eventtime.WatermarkStrategy; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.source.RichSourceFunction; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.Tumble; **import** org.apache.flink.table.api.bridge.java.StreamTableEnvironment; **import** org.apache.flink.types.Row;  **import** java.time.Duration; **import** java.util.Random; **import** java.util.UUID; **import** java.util.concurrent.TimeUnit;  **import static** org.apache.flink.table.api.Expressions.*$*; **import static** org.apache.flink.table.api.Expressions.*lit*;  */\*\*  \* Author itcast  \* Desc  \*/* **public class** FlinkSQL\_Table\_Demo05 {  **public static void** main(String[] args) **throws** Exception {  *//1.准备环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  StreamTableEnvironment tEnv = StreamTableEnvironment.*create*(env);   *//2.Source* DataStreamSource<Order> orderDS = env.addSource(**new** RichSourceFunction<Order>() {  **private** Boolean **isRunning** = **true**;   @Override  **public void** run(SourceContext<Order> ctx) **throws** Exception {  Random random = **new** Random();  **while** (**isRunning**) {  Order order = **new** Order(UUID.*randomUUID*().toString(), random.nextInt(3), random.nextInt(101), System.*currentTimeMillis*());  TimeUnit.***SECONDS***.sleep(1);  ctx.collect(order);  }  }   @Override  **public void** cancel() {  **isRunning** = **false**;  }  });   *//3.Transformation* DataStream<Order> watermakerDS = orderDS  .assignTimestampsAndWatermarks(  WatermarkStrategy.<Order>*forBoundedOutOfOrderness*(Duration.*ofSeconds*(2))  .withTimestampAssigner((event, timestamp) -> event.getCreateTime())  );   *//4.注册表* tEnv.createTemporaryView(**"t\_order"**, watermakerDS,  *$*(**"orderId"**), *$*(**"userId"**), *$*(**"money"**), *$*(**"createTime"**).rowtime());   *//查看表约束* tEnv.from(**"t\_order"**).printSchema();   *//5.TableAPI查询* Table ResultTable = tEnv.from(**"t\_order"**)  *//.window(Tumble.over("5.second").on("createTime").as("tumbleWindow"))* .window(Tumble.*over*(*lit*(5).second())  .on(*$*(**"createTime"**))  .as(**"tumbleWindow"**))  .groupBy(*$*(**"tumbleWindow"**), *$*(**"userId"**))  .select(  *$*(**"userId"**),  *$*(**"userId"**).count().as(**"totalCount"**),  *$*(**"money"**).max().as(**"maxMoney"**),  *$*(**"money"**).min().as(**"minMoney"**));    *//6.将SQL的执行结果转换成DataStream再打印出来* DataStream<Tuple2<Boolean, Row>> resultDS = tEnv.toRetractStream(ResultTable, Row.**class**);  resultDS.print();   *//7.excute* env.execute();  }   @Data  @AllArgsConstructor  @NoArgsConstructor  **public static class** Order {  **private** String **orderId**;  **private** Integer **userId**;  **private** Integer **money**;  **private** Long **createTime**;  } } |

## 案例4

### 需求

从Kafka中消费数据并过滤出状态为success的数据再写入到Kafka

|  |
| --- |
| {"user\_id": "1", "page\_id":"1", "status": "success"}  {"user\_id": "1", "page\_id":"1", "status": "success"}  {"user\_id": "1", "page\_id":"1", "status": "success"}  {"user\_id": "1", "page\_id":"1", "status": "success"}  {"user\_id": "1", "page\_id":"1", "status": "fail"} |

|  |
| --- |
| /export/server/kafka/bin/kafka-topics.sh --create --zookeeper node1:2181 --replication-factor 2 --partitions 3 --topic input\_kafka  /export/server/kafka/bin/kafka-topics.sh --create --zookeeper node1:2181 --replication-factor 2 --partitions 3 --topic output\_kafka  /export/server/kafka/bin/kafka-console-producer.sh --broker-list node1:9092 --topic input\_kafka  /export/server/kafka/bin/kafka-console-consumer.sh --bootstrap-server node1:9092 --topic output\_kafka --from-beginning |

### 代码实现

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/>

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/connectors/kafka.html>

|  |
| --- |
| **package** cn.itcast.sql;  **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.TableResult; **import** org.apache.flink.table.api.bridge.java.StreamTableEnvironment; **import** org.apache.flink.types.Row;  */\*\*  \* Author itcast  \* Desc  \*/* **public class** FlinkSQL\_Table\_Demo06 {  **public static void** main(String[] args) **throws** Exception {  *//1.准备环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  StreamTableEnvironment tEnv = StreamTableEnvironment.*create*(env);   *//2.Source* TableResult inputTable = tEnv.executeSql(  **"CREATE TABLE input\_kafka (\n"** +  **" `user\_id` BIGINT,\n"** +  **" `page\_id` BIGINT,\n"** +  **" `status` STRING\n"** +  **") WITH (\n"** +  **" 'connector' = 'kafka',\n"** +  **" 'topic' = 'input\_kafka',\n"** +  **" 'properties.bootstrap.servers' = 'node1:9092',\n"** +  **" 'properties.group.id' = 'testGroup',\n"** +  **" 'scan.startup.mode' = 'latest-offset',\n"** +  **" 'format' = 'json'\n"** +  **")"** );  TableResult outputTable = tEnv.executeSql(  **"CREATE TABLE output\_kafka (\n"** +  **" `user\_id` BIGINT,\n"** +  **" `page\_id` BIGINT,\n"** +  **" `status` STRING\n"** +  **") WITH (\n"** +  **" 'connector' = 'kafka',\n"** +  **" 'topic' = 'output\_kafka',\n"** +  **" 'properties.bootstrap.servers' = 'node1:9092',\n"** +  **" 'format' = 'json',\n"** +  **" 'sink.partitioner' = 'round-robin'\n"** +  **")"** );   String sql = **"select "** +  **"user\_id,"** +  **"page\_id,"** +  **"status "** +  **"from input\_kafka "** +  **"where status = 'success'"**;   Table ResultTable = tEnv.sqlQuery(sql);   DataStream<Tuple2<Boolean, Row>> resultDS = tEnv.toRetractStream(ResultTable, Row.**class**);  resultDS.print();   tEnv.executeSql(**"insert into output\_kafka select \* from "**+ResultTable);    *//7.excute* env.execute();  }   } |

## 总结-Flink-SQL常用算子

### SELECT

SELECT 用于从 DataSet/DataStream 中选择数据，用于筛选出某些列。

示例：

SELECT \* FROM Table；// 取出表中的所有列

SELECT name，age FROM Table；// 取出表中 name 和 age 两列

与此同时 SELECT 语句中可以使用函数和别名，例如我们上面提到的 WordCount 中：

SELECT word, COUNT(word) FROM table GROUP BY word;

### WHERE

WHERE 用于从数据集/流中过滤数据，与 SELECT 一起使用，用于根据某些条件对关系做水平分割，即选择符合条件的记录。

示例：

SELECT name，age FROM Table where name LIKE ‘% 小明 %’；

SELECT \* FROM Table WHERE age = 20；

WHERE 是从原数据中进行过滤，那么在 WHERE 条件中，Flink SQL 同样支持 =、<、>、<>、>=、<=，以及 AND、OR 等表达式的组合，最终满足过滤条件的数据会被选择出来。并且 WHERE 可以结合 IN、NOT IN 联合使用。举个例子：

SELECT name, age

FROM Table

WHERE name IN (SELECT name FROM Table2)

### DISTINCT

DISTINCT 用于从数据集/流中去重根据 SELECT 的结果进行去重。

示例：

SELECT DISTINCT name FROM Table;

对于流式查询，计算查询结果所需的 State 可能会无限增长，用户需要自己控制查询的状态范围，以防止状态过大。

### GROUP BY

GROUP BY 是对数据进行分组操作。例如我们需要计算成绩明细表中，每个学生的总分。

示例：

SELECT name, SUM(score) as TotalScore FROM Table GROUP BY name;

### UNION 和 UNION ALL

UNION 用于将两个结果集合并起来，要求两个结果集字段完全一致，包括字段类型、字段顺序。

不同于 UNION ALL 的是，UNION 会对结果数据去重。

示例：

SELECT \* FROM T1 UNION (ALL) SELECT \* FROM T2；

### JOIN

JOIN 用于把来自两个表的数据联合起来形成结果表，Flink 支持的 JOIN 类型包括：

JOIN - INNER JOIN

LEFT JOIN - LEFT OUTER JOIN

RIGHT JOIN - RIGHT OUTER JOIN

FULL JOIN - FULL OUTER JOIN

这里的 JOIN 的语义和我们在关系型数据库中使用的 JOIN 语义一致。

示例：

JOIN(将订单表数据和商品表进行关联)

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

LEFT JOIN 与 JOIN 的区别是当右表没有与左边相 JOIN 的数据时候，右边对应的字段补 NULL 输出，RIGHT JOIN 相当于 LEFT JOIN 左右两个表交互一下位置。FULL JOIN 相当于 RIGHT JOIN 和 LEFT JOIN 之后进行 UNION ALL 操作。

示例：

SELECT \* FROM Orders LEFT JOIN Product ON Orders.productId = Product.id

SELECT \* FROM Orders RIGHT JOIN Product ON Orders.productId = Product.id

SELECT \* FROM Orders FULL OUTER JOIN Product ON Orders.productId = Product.id

### Group Window

根据窗口数据划分的不同，目前 Apache Flink 有如下 3 种 Bounded Window：

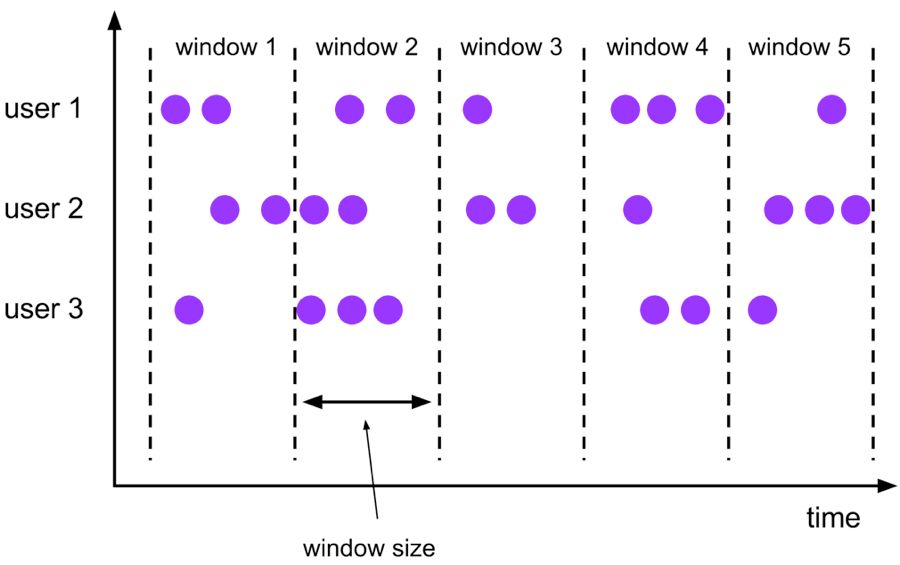
Tumble，滚动窗口，窗口数据有固定的大小，窗口数据无叠加；

Hop，滑动窗口，窗口数据有固定大小，并且有固定的窗口重建频率，窗口数据有叠加；

Session，会话窗口，窗口数据没有固定的大小，根据窗口数据活跃程度划分窗口，窗口数据无叠加。

#### Tumble Window

Tumble 滚动窗口有固定大小，窗口数据不重叠，具体语义如下：



Tumble 滚动窗口对应的语法如下：

SELECT

[gk],

[TUMBLE\_START(timeCol, size)],

[TUMBLE\_END(timeCol, size)],

agg1(col1),

...

aggn(colN)

FROM Tab1

GROUP BY [gk], TUMBLE(timeCol, size)

其中：

[gk] 决定了是否需要按照字段进行聚合；

TUMBLE\_START 代表窗口开始时间；

TUMBLE\_END 代表窗口结束时间；

timeCol 是流表中表示时间字段；

size 表示窗口的大小，如 秒、分钟、小时、天。

举个例子，假如我们要计算每个人每天的订单量，按照 user 进行聚合分组：

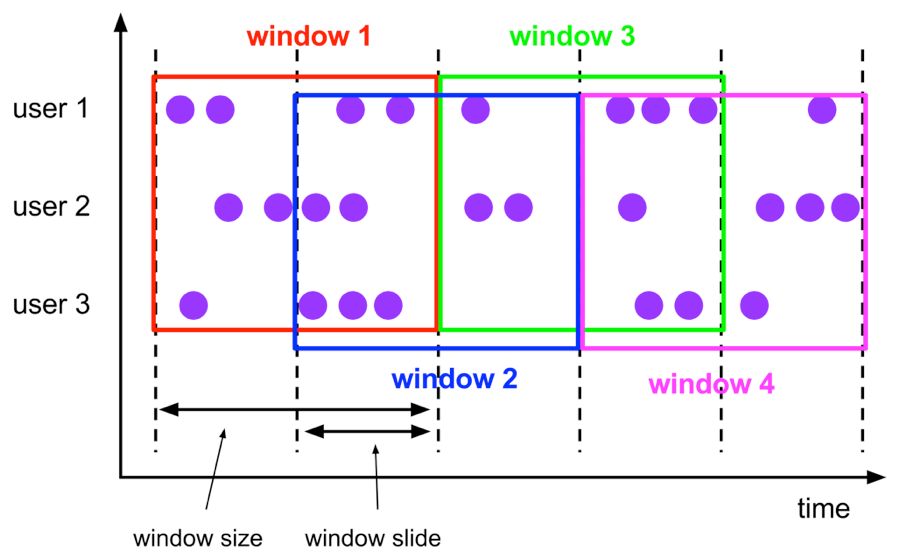
SELECT user, TUMBLE\_START(rowtime, INTERVAL ‘1’ DAY) as wStart, SUM(amount)

FROM Orders

GROUP BY TUMBLE(rowtime, INTERVAL ‘1’ DAY), user;

#### Hop Window

Hop 滑动窗口和滚动窗口类似，窗口有固定的 size，与滚动窗口不同的是滑动窗口可以通过 slide 参数控制滑动窗口的新建频率。因此当 slide 值小于窗口 size 的值的时候多个滑动窗口会重叠，具体语义如下：



Hop 滑动窗口对应语法如下：

SELECT

[gk],

[HOP\_START(timeCol, slide, size)] ,

[HOP\_END(timeCol, slide, size)],

agg1(col1),

...

aggN(colN)

FROM Tab1

GROUP BY [gk], HOP(timeCol, slide, size)

每次字段的意思和 Tumble 窗口类似：

[gk] 决定了是否需要按照字段进行聚合；

HOP\_START 表示窗口开始时间；

HOP\_END 表示窗口结束时间；

timeCol 表示流表中表示时间字段；

slide 表示每次窗口滑动的大小；

size 表示整个窗口的大小，如 秒、分钟、小时、天。

举例说明，我们要每过一小时计算一次过去 24 小时内每个商品的销量：

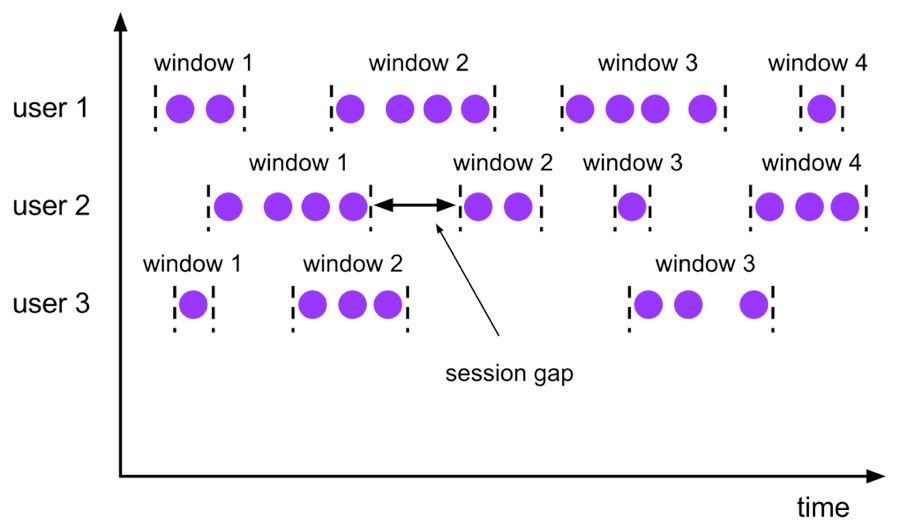
SELECT product, SUM(amount)

FROM Orders

GROUP BY product,HOP(rowtime, INTERVAL '1' HOUR, INTERVAL '1' DAY)

#### Session Window

会话时间窗口没有固定的持续时间，但它们的界限由 interval 不活动时间定义，即如果在定义的间隙期间没有出现事件，则会话窗口关闭。



Seeeion 会话窗口对应语法如下：

SELECT

[gk],

SESSION\_START(timeCol, gap) AS winStart,

SESSION\_END(timeCol, gap) AS winEnd,

agg1(col1),

...

aggn(colN)

FROM Tab1

GROUP BY [gk], SESSION(timeCol, gap)

[gk] 决定了是否需要按照字段进行聚合；

SESSION\_START 表示窗口开始时间；

SESSION\_END 表示窗口结束时间；

timeCol 表示流表中表示时间字段；

gap 表示窗口数据非活跃周期的时长。

例如，我们需要计算每个用户访问时间 12 小时内的订单量：

SELECT user, SESSION\_START(rowtime, INTERVAL ‘12’ HOUR) AS sStart, SESSION\_ROWTIME(rowtime, INTERVAL ‘12’ HOUR) AS sEnd, SUM(amount)

FROM Orders

GROUP BY SESSION(rowtime, INTERVAL ‘12’ HOUR), user