Table API 和 Flink SQL

# 第一章 整体介绍

## 1.1 什么是 Table API 和 Flink SQL

Flink本身是批流统一的处理框架，所以Table API和SQL，就是批流统一的上层处理API。

目前功能尚未完善，处于活跃的开发阶段。

Table API是一套内嵌在Java和Scala语言中的查询API，它允许我们以非常直观的方式，组合来自一些关系运算符的查询（比如select、filter和join）。而对于Flink SQL，就是直接可以在代码中写SQL，来实现一些查询（Query）操作。Flink的SQL支持，基于实现了SQL标准的Apache Calcite（Apache开源SQL解析工具）。

无论输入是批输入还是流式输入，在这两套API中，指定的查询都具有相同的语义，得到相同的结果。

## 1.2 需要引入的依赖

Table API和SQL需要引入的依赖有两个：planner和bridge。

<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-table-planner\_2.12</**artifactId**>  
 <**version**>1.10.1</**version**>  
</**dependency**>

<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-table-api-java-bridge\_2.12</**artifactId**>  
 <**version**>1.10.1</**version**>  
</**dependency**>

flink-table-planner：planner计划器，是table API最主要的部分，提供了运行时环境和生成程序执行计划的planner；

flink-table-api-scala-bridge：bridge桥接器，主要负责table API和 DataStream/DataSet API的连接支持，按照语言分java和scala。

这里的两个依赖，是IDE环境下运行需要添加的；如果是生产环境，lib目录下默认已经有了planner，就只需要有bridge就可以了。

当然，如果想使用用户自定义函数，或是跟kafka做连接，需要有一个SQL client，这个包含在flink-table-common里。

## 1.3 两种planner（old & blink）的区别

1. 批流统一：Blink将批处理作业，视为流式处理的特殊情况。所以，blink不支持表和DataSet之间的转换，批处理作业将不转换为DataSet应用程序，而是跟流处理一样，转换为DataStream程序来处理。

2. 因为批流统一，Blink planner也不支持BatchTableSource，而使用有界的StreamTableSource代替。

3. Blink planner只支持全新的目录，不支持已弃用的ExternalCatalog。

4. 旧planner和Blink planner的FilterableTableSource实现不兼容。旧的planner会把PlannerExpressions下推到filterableTableSource中，而blink planner则会把Expressions下推。

5. 基于字符串的键值配置选项仅适用于Blink planner。

6. PlannerConfig在两个planner中的实现不同。

7. Blink planner会将多个sink优化在一个DAG中（仅在TableEnvironment上受支持，而在StreamTableEnvironment上不受支持）。而旧planner的优化总是将每一个sink放在一个新的DAG中，其中所有DAG彼此独立。

8. 旧的planner不支持目录统计，而Blink planner支持。

# 第二章 API调用

## 2.1 基本程序结构

Table API 和 SQL 的程序结构，与流式处理的程序结构类似；也可以近似地认为有这么几步：首先创建执行环境，然后定义source、transform和sink。

具体操作流程如下：

StreamTableEnvironment tableEnv = ... *// 创建表的执行环境  
  
// 创建一张表，用于读取数据*tableEnv.connect(...).createTemporaryTable(**"inputTable"**);  
*// 注册一张表，用于把计算结果输出*tableEnv.connect(...).createTemporaryTable(**"outputTable"**);  
  
*// 通过 Table API 查询算子，得到一张结果表*Table result = tableEnv.from(**"inputTable"**).select(...);  
*// 通过 SQL查询语句，得到一张结果表*Table sqlResult = tableEnv.sqlQuery(**"SELECT ... FROM inputTable ..."**);  
  
*// 将结果表写入输出表中*result.insertInto(**"outputTable"**);

## 2.2 创建表环境

创建表环境最简单的方式，就是基于流处理执行环境，调create方法直接创建：

StreamTableEnvironmenttableEnv = StreamTableEnvironment.*create*(env);

表环境（TableEnvironment）是flink中集成Table API & SQL的核心概念。它负责:

* 注册catalog
* 在内部 catalog 中注册表
* 执行 SQL 查询
* 注册用户自定义函数
* 将 DataStream 或 DataSet 转换为表
* 保存对 ExecutionEnvironment 或 StreamExecutionEnvironment 的引用

在创建TableEnv的时候，可以多传入一个EnvironmentSettings或者TableConfig参数，可以用来配置 TableEnvironment的一些特性。

比如，配置老版本的流式查询（Flink-Streaming-Query）：

EnvironmentSettingssettings = EnvironmentSettings.*newInstance*()  
 .useOldPlanner() *// 使用老版本planner* .inStreamingMode() *// 流处理模式* .build();  
StreamTableEnvironmenttableEnv = StreamTableEnvironment.*create*(env, settings);

基于老版本的批处理环境（Flink-Batch-Query）：

ExecutionEnvironmentbatchEnv = ExecutionEnvironment.*getExecutionEnvironment;*

BatchTableEnvironment batchTableEnv = BatchTableEnvironment.*create*(batchEnv);

基于blink版本的流处理环境（Blink-Streaming-Query）：

EnvironmentSettings bsSettings = EnvironmentSettings.*newInstance*()

.useBlinkPlanner()

.inStreamingMode().build();  
StreamTableEnvironment bsTableEnv = StreamTableEnvironment.*create*(env, bsSettings);

基于blink版本的批处理环境（Blink-Batch-Query）：

EnvironmentSettings bbSettings = EnvironmentSettings.*newInstance*()

.useBlinkPlanner()

.inBatchMode().build();  
TableEnvironment bbTableEnv = TableEnvironment.*create*(bbSettings);

## 2.3 在Catalog中注册表

### 2.3.1 表（Table）的概念

TableEnvironment可以注册目录Catalog，并可以基于Catalog注册表。它会维护一个Catalog-Table表之间的map。

表（Table）是由一个“标识符”来指定的，由3部分组成：Catalog名、数据库（database）名和对象名（表名）。如果没有指定目录或数据库，就使用当前的默认值。

表可以是常规的（Table，表），或者虚拟的（View，视图）。常规表（Table）一般可以用来描述外部数据，比如文件、数据库表或消息队列的数据，也可以直接从 DataStream转换而来。视图可以从现有的表中创建，通常是table API或者SQL查询的一个结果。

### 2.3.2 连接到文件系统（Csv格式）

连接外部系统在Catalog中注册表，直接调用tableEnv.connect()就可以，里面参数要传入一个ConnectorDescriptor，也就是connector描述器。对于文件系统的connector而言，flink内部已经提供了，就叫做FileSystem()。

代码如下：

tableEnv

.connect( **new** FileSystem().path(**"sensor.txt"**)) *// 定义表数据来源，外部连接* .withFormat(**new** OldCsv()) *// 定义从外部系统读取数据之后的格式化方法* .withSchema( **new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"timestamp"**, DataTypes.*BIGINT*())  
 .field(**"temperature"**, DataTypes.*DOUBLE*())  
 ) *// 定义表结构* .createTemporaryTable(**"inputTable"**); *// 创建临时表*

这是旧版本的csv格式描述器。由于它是非标的，跟外部系统对接并不通用，所以将被弃用，以后会被一个符合RFC-4180标准的新format描述器取代。新的描述器就叫Csv()，但flink没有直接提供，需要引入依赖flink-csv：

<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-csv</**artifactId**>  
 <**version**>1.10.1</**version**>  
</**dependency**>

代码非常类似，只需要把withFormat里的OldCsv改成Csv就可以了。

### 2.3.3 连接到Kafka

kafka的连接器flink-kafka-connector中，1.10版本的已经提供了Table API的支持。我们可以在 connect方法中直接传入一个叫做Kafka的类，这就是kafka连接器的描述器ConnectorDescriptor。

tableEnv.connect(  
 **new** Kafka()  
 .version(**"0.11"**) *// 定义kafka的版本* .topic(**"sensor"**) *// 定义主题* .property(**"zookeeper.connect"**, **"localhost:2181"**)   
 .property(**"bootstrap.servers"**, **"localhost:9092"**)  
)  
 .withFormat(**new** Csv()).withSchema(**new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"timestamp"**, DataTypes.*BIGINT*())  
 .field(**"temperature"**, DataTypes.*DOUBLE*())  
)  
 .createTemporaryTable(**"kafkaInputTable"**);

当然也可以连接到ElasticSearch、MySql、HBase、Hive等外部系统，实现方式基本上是类似的。

## 2.4 表的查询

利用外部系统的连接器connector，我们可以读写数据，并在环境的Catalog中注册表。接下来就可以对表做查询转换了。

Flink给我们提供了两种查询方式：Table API和 SQL。

### 2.4.1 Table API的调用

Table API是集成在Scala和Java语言内的查询API。与SQL不同，Table API的查询不会用字符串表示，而是在宿主语言中一步一步调用完成的。

Table API基于代表一张“表”的Table类，并提供一整套操作处理的方法API。这些方法会返回一个新的Table对象，这个对象就表示对输入表应用转换操作的结果。有些关系型转换操作，可以由多个方法调用组成，构成链式调用结构。例如table.select(…).filter(…)，其中select（…）表示选择表中指定的字段，filter(…)表示筛选条件。

代码中的实现如下：

TablesensorTable = tableEnv.from(**"inputTable"**);

TableresultTable = senorTable

.select(**"id, temperature"**)

.filter(**"id ='sensor\_1'"**);

### 2.4.2 SQL查询

Flink的SQL集成，基于的是ApacheCalcite，它实现了SQL标准。在Flink中，用常规字符串来定义SQL查询语句。SQL 查询的结果，是一个新的 Table。

代码实现如下：

TableresultSqlTable = tableEnv.sqlQuery(**"select id, temperature from inputTable where id ='sensor\_1'"**);

当然，也可以加上聚合操作，比如我们统计每个sensor温度数据出现的个数，做个count统计：

TableaggResultTable = sensorTable

.groupBy(**"id"**)

.select(**"id**, **id**.count as **count"**);

SQL的实现：

Table aggResultSqlTable = tableEnv.sqlQuery(**"select id, count(id) as cnt from inputTable group by id"**);

这里Table API里指定的字段，前面加了一个单引号’，这是Table API中定义的Expression类型的写法，可以很方便地表示一个表中的字段。

字段可以直接全部用双引号引起来，也可以用半边单引号+字段名的方式。以后的代码中，一般都用后一种形式。

## 2.5 将DataStream 转换成表

Flink允许我们把Table和DataStream做转换：我们可以基于一个DataStream，先流式地读取数据源，然后map成POJO，再把它转成Table。Table的列字段（column fields），就是POJO里的字段，这样就不用再麻烦地定义schema了。

### 2.5.1 代码表达

代码中实现非常简单，直接用tableEnv.fromDataStream()就可以了。默认转换后的 Table schema 和 DataStream 中的字段定义一一对应，也可以单独指定出来。

这就允许我们更换字段的顺序、重命名，或者只选取某些字段出来，相当于做了一次map操作（或者Table API的 select操作）。

代码具体如下：

DataStream<String>inputStream = env.readTextFile(**"sensor.txt"**);

DataStream<SensorReading> dataStream = inputStream  
 .map( line -> {  
 String[] fields = line.split(**","**);  
 **return new** SensorReading(fields[0], **new** Long(fields[1]), **new** Double(fields[2]));  
 } );

Table sensorTable = tableEnv.fromDataStream(dataStream, **"id, timestamp.rowtime as ts, temperature"**);

### 2.5.2 数据类型与 Table schema的对应

在上节的例子中，DataStream 中的数据类型，与表的 Schema 之间的对应关系，是按照类中的字段名来对应的（name-based mapping），所以还可以用as做重命名。

基于名称的对应：

Table sensorTable = tableEnv.fromDataStream(dataStream, **"timestamp** as **ts**, **id** as **myId**, **temperature"**);

Flink的DataStream和 DataSet API支持多种类型。

组合类型，比如元组（内置Scala和Java元组）、POJO、Scala case类和Flink的Row类型等，允许具有多个字段的嵌套数据结构，这些字段可以在Table的表达式中访问。其他类型，则被视为原子类型。

## 2.6. 创建临时视图（Temporary View）

创建临时视图的第一种方式，就是直接从DataStream转换而来。同样，可以直接对应字段转换；也可以在转换的时候，指定相应的字段。

代码如下：

tableEnv.createTemporaryView(**"sensorView"**, dataStream);  
tableEnv.createTemporaryView(**"sensorView"**, dataStream, **"id**, **temperature**, **timestamp** as **ts"**);

另外，当然还可以基于Table创建视图：

tableEnv.createTemporaryView(**"sensorView"**, sensorTable);

View和Table的Schema完全相同。事实上，在Table API中，可以认为View和Table是等价的。

## 2.7. 输出表

表的输出，是通过将数据写入 TableSink 来实现的。TableSink 是一个通用接口，可以支持不同的文件格式、存储数据库和消息队列。

具体实现，输出表最直接的方法，就是通过 Table.insertInto() 方法将一个 Table 写入注册过的 TableSink 中。

### 2.7.1 输出到文件

代码如下：

*// 注册输出表*tableEnv.connect(  
 **new** FileSystem().path(**"…\\resources\\out.txt"**)  
) *// 定义到文件系统的连接* .withFormat(**new** Csv()) *// 定义格式化方法，Csv格式* .withSchema(**new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"temp"**, DataTypes.*DOUBLE*())  
) *// 定义表结构* .createTemporaryTable(**"outputTable"**); *// 创建临时表*resultSqlTable.insertInto(**"outputTable"**);

### 2.7.2 更新模式（Update Mode）

在流处理过程中，表的处理并不像传统定义的那样简单。

对于流式查询（Streaming Queries），需要声明如何在（动态）表和外部连接器之间执行转换。与外部系统交换的消息类型，由**更新模式**（update mode）指定。

Flink Table API中的更新模式有以下三种：

#### 1）追加模式（Append Mode）

在追加模式下，表（动态表）和外部连接器只交换插入（Insert）消息。

#### 2）撤回模式（Retract Mode）

在撤回模式下，表和外部连接器交换的是：添加（Add）和撤回（Retract）消息。

* 插入（Insert）会被编码为添加消息；
* 删除（Delete）则编码为撤回消息；
* 更新（Update）则会编码为，已更新行（上一行）的撤回消息，和更新行（新行）的添加消息。

在此模式下，不能定义key，这一点跟upsert模式完全不同。

#### 3）Upsert（更新插入）模式

在Upsert模式下，动态表和外部连接器交换Upsert和Delete消息。

这个模式需要一个唯一的key，通过这个key可以传递更新消息。为了正确应用消息，外部连接器需要知道这个唯一key的属性。

* 插入（Insert）和更新（Update）都被编码为Upsert消息；
* 删除（Delete）编码为Delete信息。

这种模式和Retract模式的主要区别在于，Update操作是用单个消息编码的，所以效率会更高。

### 2.7.3 输出到Kafka

除了输出到文件，也可以输出到Kafka。我们可以结合前面Kafka作为输入数据，构建数据管道，kafka进，kafka出。

代码如下：

*// 输出到 kafka*tableEnv.connect(  
 **new** Kafka()  
 .version(**"0.11"**)  
 .topic(**"sinkTest"**)  
 .property(**"zookeeper.connect"**, **"localhost:2181"**)  
 .property(**"bootstrap.servers"**, **"localhost:9092"**)  
)  
 .withFormat( **new** Csv() )  
 .withSchema( **new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"temp"**, DataTypes.*DOUBLE*())  
 )  
 .createTemporaryTable(**"kafkaOutputTable"**);  
  
resultTable.insertInto(**"kafkaOutputTable"**);

### 2.7.4 输出到ElasticSearch

ElasticSearch的connector可以在upsert（update+insert，更新插入）模式下操作，这样就可以使用Query定义的键（key）与外部系统交换UPSERT/DELETE消息。

另外，对于“仅追加”（append-only）的查询，connector还可以在append 模式下操作，这样就可以与外部系统只交换insert消息。

es目前支持的数据格式，只有Json，而flink本身并没有对应的支持，所以还需要引入依赖：

<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-json</**artifactId**>  
 <**version**>1.10.1</**version**>  
</**dependency**>

代码实现如下：

*// 输出到es*tableEnv.connect(  
 **new** Elasticsearch()  
 .version(**"6"**)  
 .host(**"localhost"**, 9200, **"http"**)  
 .index(**"sensor"**)  
 .documentType(**"temp"**)  
)  
 .inUpsertMode() *// 指定是 Upsert 模式*  
 .withFormat(**new** Json())  
 .withSchema( **new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"count"**, DataTypes.*BIGINT*())  
 )  
 .createTemporaryTable(**"esOutputTable"**);  
  
aggResultTable.insertInto(**"esOutputTable"**);

### 2.7.5 输出到MySql

Flink专门为Table API的jdbc连接提供了flink-jdbc连接器，我们需要先引入依赖：

<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-jdbc\_2.12</**artifactId**>  
 <**version**>1.10.1</**version**>  
</**dependency**>

jdbc连接的代码实现比较特殊，因为没有对应的java/scala类实现ConnectorDescriptor，所以不能直接tableEnv.connect()。不过Flink SQL留下了执行DDL的接口：tableEnv.sqlUpdate()。

对于jdbc的创建表操作，天生就适合直接写DDL来实现，所以我们的代码可以这样写：

*// 输出到 Mysql*String sinkDDL= **"create table jdbcOutputTable ("** +  
 **" id varchar(20) not null, "** +  
 **" cnt bigint not null "** +  
 **") with ("** +  
 **" 'connector.type' = 'jdbc', "** +  
 **" 'connector.url' = 'jdbc:mysql://localhost:3306/test', "** +  
 **" 'connector.table' = 'sensor\_count', "** +  
 **" 'connector.driver' = 'com.mysql.jdbc.Driver', "** +  
 **" 'connector.username' = 'root', "** +  
 **" 'connector.password' = '123456' )"**;

tableEnv.sqlUpdate(sinkDDL); *// 执行 DDL创建表*aggResultSqlTable.insertInto(**"jdbcOutputTable"**);

## 2.8 将表转换成DataStream

表可以转换为DataStream或DataSet。这样，自定义流处理或批处理程序就可以继续在 Table API或SQL查询的结果上运行了。

将表转换为DataStream或DataSet时，需要指定生成的数据类型，即要将表的每一行转换成的数据类型。通常，最方便的转换类型就是Row。当然，因为结果的所有字段类型都是明确的，我们也经常会用元组类型来表示。

表作为流式查询的结果，是动态更新的。所以，将这种动态查询转换成的数据流，同样需要对表的更新操作进行编码，进而有不同的转换模式。

Table API中表到DataStream有两种模式：

* 追加模式（Append Mode）

用于表只会被插入（Insert）操作更改的场景。

* 撤回模式（Retract Mode）

用于任何场景。有些类似于更新模式中Retract模式，它只有Insert和Delete两类操作。

得到的数据会增加一个Boolean类型的标识位（返回的第一个字段），用它来表示到底是新增的数据（Insert），还是被删除的数据（老数据， Delete）。

代码实现如下：

DataStream<Row> resultStream = tableEnv.toAppendStream(resultTable, Row.**class**);

DataStream<Tuple2<Boolean, Row>> aggResultStream = tableEnv.toRetractStream(aggResultTable, Row.**class**);

resultStream.print(**"result"**);

aggResultStream.print(**"aggResult"**);

所以，没有经过groupby之类聚合操作，可以直接用 toAppendStream 来转换；而如果经过了聚合，有更新操作，一般就必须用 toRetractDstream。

## 2.9 Query的解释和执行

Table API提供了一种机制来解释（Explain）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方法会返回一个字符串，描述三个计划：

* 未优化的逻辑查询计划
* 优化后的逻辑查询计划
* 实际执行计划

我们可以在代码中查看执行计划：

String explaination = tableEnv.explain(resultTable);  
System.***out***.println(explaination);

Query的解释和执行过程，老planner和blink planner大体是一致的，又有所不同。整体来讲，Query都会表示成一个逻辑查询计划，然后分两步解释：

1. 优化查询计划

2. 解释成 DataStream 或者 DataSet程序

而Blink版本是批流统一的，所以所有的Query，只会被解释成DataStream程序；另外在批处理环境TableEnvironment下，Blink版本要到tableEnv.execute()执行调用才开始解释。

# 第三章 流处理中的特殊概念

Table API和SQL，本质上还是基于关系型表的操作方式；而关系型表、关系代数，以及SQL本身，一般是有界的，更适合批处理的场景。这就导致在进行流处理的过程中，理解会稍微复杂一些，需要引入一些特殊概念。

## 3.1 流处理和关系代数（表，及SQL）的区别



可以看到，其实关系代数（主要就是指关系型数据库中的表）和SQL，主要就是针对批处理的，这和流处理有天生的隔阂。

## 3.2 动态表（Dynamic Tables）

因为流处理面对的数据，是连续不断的，这和我们熟悉的关系型数据库中保存的“表”完全不同。所以，如果我们把流数据转换成Table，然后执行类似于table的select操作，结果就不是一成不变的，而是随着新数据的到来，会不停更新。

我们可以随着新数据的到来，不停地在之前的基础上更新结果。这样得到的表，在Flink Table API概念里，就叫做“**动态表**”（Dynamic Tables）。

动态表是Flink对流数据的Table API和SQL支持的核心概念。与表示批处理数据的静态表不同，动态表是随时间变化的。动态表可以像静态的批处理表一样进行查询，查询一个动态表会产生持续查询（Continuous Query）。连续查询永远不会终止，并会生成另一个动态表。查询（Query）会不断更新其动态结果表，以反映其动态输入表上的更改。

## 3.3 流式持续查询的过程

下图显示了流、动态表和连续查询的关系：



流式持续查询的过程为：

1. 流被转换为动态表。

2. 对动态表计算连续查询，生成新的动态表。

3. 生成的动态表被转换回流。

### 3.3.1 将流转换成表（Table）

为了处理带有关系查询的流，必须先将其转换为表。

从概念上讲，流的每个数据记录，都被解释为对结果表的插入（Insert）修改。因为流式持续不断的，而且之前的输出结果无法改变。本质上，我们其实是从一个、只有插入操作的changelog（更新日志）流，来构建一个表。

为了更好地说明动态表和持续查询的概念，我们来举一个具体的例子。

比如，我们现在的输入数据，就是用户在网站上的访问行为，数据类型（Schema）如下：

[

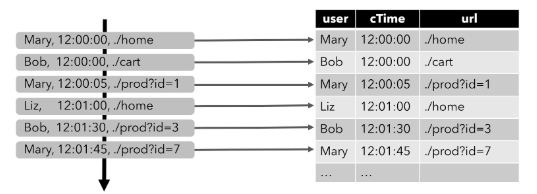
user: VARCHAR, // 用户名

cTime: TIMESTAMP, // 访问某个URL的时间戳

url: VARCHAR // 用户访问的URL

]

下图显示了如何将访问URL事件流，或者叫点击事件流（左侧）转换为表（右侧）。



随着插入更多的访问事件流记录，生成的表将不断增长。

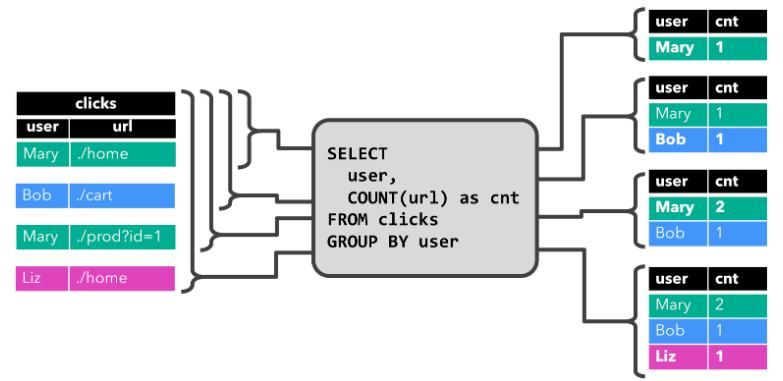
### 3.3.2 持续查询（Continuous Query）

持续查询，会在动态表上做计算处理，并作为结果生成新的动态表。与批处理查询不同，连续查询从不终止，并根据输入表上的更新更新其结果表。

在任何时间点，连续查询的结果在语义上，等同于在输入表的快照上，以批处理模式执行的同一查询的结果。

在下面的示例中，我们展示了对点击事件流中的一个持续查询。

这个Query很简单，是一个分组聚合做count统计的查询。它将用户字段上的clicks表分组，并统计访问的url数。图中显示了随着时间的推移，当clicks表被其他行更新时如何计算查询。



### 3.3.3 将动态表转换成流

与常规的数据库表一样，动态表可以通过插入（Insert）、更新（Update）和删除（Delete）更改，进行持续的修改。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支持三种方式对动态表的更改进行编码：

#### 1）仅追加（Append-only）流

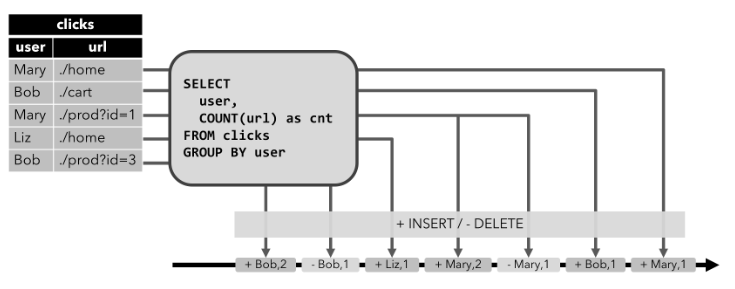
仅通过插入（Insert）更改，来修改的动态表，可以直接转换为“仅追加”流。这个流中发出的数据，就是动态表中新增的每一行。

#### 2）撤回（Retract）流

Retract流是包含两类消息的流，添加（Add）消息和撤回（Retract）消息。

动态表通过将INSERT 编码为add消息、DELETE 编码为retract消息、UPDATE编码为被更改行（前一行）的retract消息和更新后行（新行）的add消息，转换为retract流。

下图显示了将动态表转换为Retract流的过程。

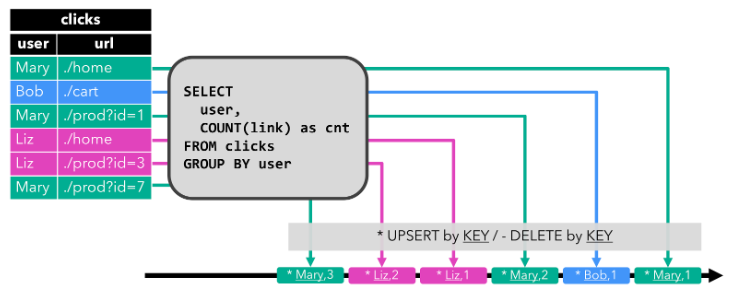


#### 3）Upsert（更新插入）流

Upsert流包含两种类型的消息：Upsert消息和delete消息。转换为upsert流的动态表，需要有唯一的键（key）。

通过将INSERT和UPDATE更改编码为upsert消息，将DELETE更改编码为DELETE消息，就可以将具有唯一键（Unique Key）的动态表转换为流。

下图显示了将动态表转换为upsert流的过程。



这些概念我们之前都已提到过。需要注意的是，在代码里将动态表转换为DataStream时，仅支持Append和Retract流。而向外部系统输出动态表的TableSink接口，则可以有不同的实现，比如之前我们讲到的ES，就可以有Upsert模式。

## 3.4 时间特性

基于时间的操作（比如Table API和SQL中窗口操作），需要定义相关的时间语义和时间数据来源的信息。所以，Table可以提供一个逻辑上的时间字段，用于在表处理程序中，指示时间和访问相应的时间戳。

时间属性，可以是每个表schema的一部分。一旦定义了时间属性，它就可以作为一个字段引用，并且可以在基于时间的操作中使用。

时间属性的行为类似于常规时间戳，可以访问，并且进行计算。

### 3.4.1 处理时间（Processing Time）

处理时间语义下，允许表处理程序根据机器的本地时间生成结果。它是时间的最简单概念。它既不需要提取时间戳，也不需要生成watermark。

定义处理时间属性有三种方法：在DataStream转化时直接指定；在定义Table Schema时指定；在创建表的DDL中指定。

#### DataStream转化成Table时指定

由DataStream转换成表时，可以在后面指定字段名来定义Schema。在定义Schema期间，可以使用.proctime，定义处理时间字段。

注意，这个proctime属性只能通过附加逻辑字段，来扩展物理schema。因此，只能在schema定义的末尾定义它。

代码如下：

*// 定义好 DataStream*DataStream<String> inputStream = env.readTextFile(**"\\sensor.txt"**)  
DataStream<SensorReading> dataStream = inputStream  
 .map( line -> {  
 String[] fields = line.split(**","**);  
 **return new** SensorReading(fields[0], **new** Long(fields[1]), **new** Double(fields[2]));  
 } );

*// 将 DataStream转换为 Table，并指定时间字段*Table sensorTable = tableEnv.fromDataStream(dataStream, **"id**, **temperature**, **timestamp**, **pt**.proctime**"**);

#### 定义Table Schema时指定

这种方法其实也很简单，只要在定义Schema的时候，加上一个新的字段，并指定成proctime就可以了。

代码如下：

tableEnv.connect(  
 **new** FileSystem().path(**"..\\sensor.txt"**))  
 .withFormat(**new** Csv())  
 .withSchema(**new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"timestamp"**, DataTypes.*BIGINT*())  
 .field(**"temperature"**, DataTypes.*DOUBLE*())  
 .field(**"pt"**, DataTypes.*TIMESTAMP*(3))  
 .proctime() *// 指定 pt字段为处理时间* ) *// 定义表结构* .createTemporaryTable(**"inputTable"**); *// 创建临时表*

#### 创建表的DDL中指定

在创建表的DDL中，增加一个字段并指定成proctime，也可以指定当前的时间字段。

代码如下：

String sinkDDL = **"create table dataTable ("** +  
 **" id varchar(20) not null, "** +  
 **" ts bigint, "** +  
 **" temperature double, "** +  
 **" pt AS PROCTIME() "** +  
 **") with ("** +  
 **" 'connector.type' = 'filesystem', "** +  
 **" 'connector.path' = '/sensor.txt', "** +  
 **" 'format.type' = 'csv')"**;  
  
tableEnv.sqlUpdate(sinkDDL);

注意：运行这段DDL，必须使用Blink Planner。

### 3.4.2 事件时间（Event Time）

事件时间语义，允许表处理程序根据每个记录中包含的时间生成结果。这样即使在有乱序事件或者延迟事件时，也可以获得正确的结果。

为了处理无序事件，并区分流中的准时和迟到事件；Flink需要从事件数据中，提取时间戳，并用来推进事件时间的进展（watermark）。

#### DataStream转化成Table时指定

在DataStream转换成Table，schema的定义期间，使用.rowtime可以定义事件时间属性。注意，必须在转换的数据流中分配时间戳和watermark。

在将数据流转换为表时，有两种定义时间属性的方法。根据指定的.rowtime字段名是否存在于数据流的架构中，timestamp字段可以：

* 作为新字段追加到schema
* 替换现有字段

在这两种情况下，定义的事件时间戳字段，都将保存DataStream中事件时间戳的值。

代码如下：

DataStream<String> inputStream = env.readTextFile(**"\\sensor.txt"**)  
DataStream<SensorReading> dataStream = inputStream  
 .map( line -> {  
 String[] fields = line.split(**","**);  
 **return new** SensorReading(fields[0], **new** Long(fields[1]), **new** Double(fields[2]));  
 } )  
 .assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor<SensorReading>(Time.*seconds*(1)) {  
 **@Override  
 public long** extractTimestamp(SensorReading element) {  
 **return** element.getTimestamp() \* 1000L;  
 }  
 });

Table sensorTable = tableEnv.fromDataStream(dataStream, **"id, timestamp.rowtime as ts, temperature"**);

#### 定义Table Schema时指定

这种方法只要在定义Schema的时候，将事件时间字段，并指定成rowtime就可以了。

代码如下：

tableEnv.connect(  
 **new** FileSystem().path(**"sensor.txt"**))  
 .withFormat(**new** Csv())  
 .withSchema(**new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"timestamp"**, DataTypes.*BIGINT*())  
 .rowtime(  
 **new** Rowtime()  
 .timestampsFromField(**"timestamp"**) *// 从字段中提取时间戳*  
 .watermarksPeriodicBounded(1000)  *// watermark延迟1秒*  
 )  
 .field(**"temperature"**, DataTypes.*DOUBLE*())) *// 定义表结构* .createTemporaryTable(**"inputTable"**); *// 创建临时表*

#### 创建表的DDL中指定

事件时间属性，是使用CREATE TABLE DDL中的WARDMARK语句定义的。watermark语句，定义现有事件时间字段上的watermark生成表达式，该表达式将事件时间字段标记为事件时间属性。

代码如下：

String sinkDDL = **"create table dataTable ("** +  
 **" id varchar(20) not null, "** +  
 **" ts bigint, "** +  
 **" temperature double, "** +  
 **" rt AS TO\_TIMESTAMP( FROM\_UNIXTIME(ts) ), "** +  
 **" watermark for rt as rt - interval '1' second"** +  
 **") with ("** +  
 **" 'connector.type' = 'filesystem', "** +  
 **" 'connector.path' = '/sensor.txt', "** +  
 **" 'format.type' = 'csv')"**;  
  
tableEnv.sqlUpdate(sinkDDL);

这里*FROM\_UNIXTIME*是系统内置的时间函数，用来将一个整数（秒数）转换成“YYYY-MM-DD hh:mm:ss”格式（默认，也可以作为第二个String参数传入）的日期时间字符串（date time string）；然后再用*TO\_TIMESTAMP*将其转换成Timestamp。

# 第四章 窗口（Windows）

时间语义，要配合窗口操作才能发挥作用。最主要的用途，当然就是开窗口、根据时间段做计算了。下面我们就来看看Table API和SQL中，怎么利用时间字段做窗口操作。

在Table API和SQL中，主要有两种窗口：Group Windows和Over Windows

## 4.1 分组窗口（Group Windows）

分组窗口（Group Windows）会根据时间或行计数间隔，将行聚合到有限的组（Group）中，并对每个组的数据执行一次聚合函数。

Table API中的Group Windows都是使用.window（w:GroupWindow）子句定义的，并且必须由as子句指定一个别名。为了按窗口对表进行分组，窗口的别名必须在group by子句中，像常规的分组字段一样引用。

Table *table* = input  
 .window([w: GroupWindow] as **"w"**) *// 定义窗口，别名 w* .groupBy(**"w**, **a"**) *// 以属性a和窗口w作为分组的key* .select(**"a**, **b**.sum**"**) *// 聚合字段b的值，求和*

或者，还可以把窗口的相关信息，作为字段添加到结果表中：

Table *table* = input  
 .window([w: GroupWindow] as **"w"**) .groupBy(**"w**, **a"**)

.select(**"a**, **w**.start, **w**.end, **w**.rowtime, **b**.count**"**)

Table API提供了一组具有特定语义的预定义Window类，这些类会被转换为底层DataStream或DataSet的窗口操作。

Table API支持的窗口定义，和我们熟悉的一样，主要也是三种：滚动（Tumbling）、滑动（Sliding）和会话（Session）。

### 4.1.1 滚动窗口

滚动窗口（Tumbling windows）要用Tumble类来定义，另外还有三个方法：

* over：定义窗口长度
* on：用来分组（按时间间隔）或者排序（按行数）的时间字段
* as：别名，必须出现在后面的groupBy中

代码如下：

*// Tumbling Event-time Window*.window(Tumble.*over*(**"10.minutes"**).on(**"rowtime"**).as(**"w"**))  
  
*// Tumbling Processing-time Window*.window(Tumble.*over*(**"10.minutes"**).on(**"proctime"**).as(**"w"**))  
  
*// Tumbling Row-count Window*.window(Tumble.*over*(**"10.rows"**).on(**"proctime"**).as(**"w"**))

### 4.1.2 滑动窗口

滑动窗口（Sliding windows）要用Slide类来定义，另外还有四个方法：

* over：定义窗口长度
* every：定义滑动步长
* on：用来分组（按时间间隔）或者排序（按行数）的时间字段
* as：别名，必须出现在后面的groupBy中

代码如下：

*// Sliding Event-time Window*.window(Slide.over(**"10.minutes"**).every(**"5.minutes"**).on(**"rowtime"**).as(**"w"**))  
  
*// Sliding Processing-time window*.window(Slide.over(**"10.minutes"**).every(**"5.minutes"**).on(**"proctime"**).as(**"w"**))  
  
*// Sliding Row-count window*.window(Slide.over(**"10.rows"**).every(**"5.rows"**).on(**"proctime"**).as(**"w"**))

### 4.1.3 会话窗口

会话窗口（Session windows）要用Session类来定义，另外还有三个方法：

* withGap：会话时间间隔
* on：用来分组（按时间间隔）或者排序（按行数）的时间字段
* as：别名，必须出现在后面的groupBy中

代码如下：

*// Session Event-time Window*.window(Session.withGap.(**"10.minutes"**).on(**"rowtime"**).as(**"w"**))  
  
*// Session Processing-time Window*.window(Session.withGap.(**"10.minutes"**).on(“proctime**").as("**w**"))**

## 4.2 Over Windows

Over window聚合是标准SQL中已有的（Over子句），可以在查询的SELECT子句中定义。Over window 聚合，会针对每个输入行，计算相邻行范围内的聚合。Over windows

使用.window（w:overwindows\*）子句定义，并在select（）方法中通过别名来引用。

比如这样：

Table *table* = input  
 .window([w: OverWindow] as **"w"**)  
 .select(**"a**, **b**.sum over **w**, **c**.min over **w"**)

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、或行计数的范围内，定义Over windows。

无界的over window是使用常量指定的。也就是说，时间间隔要指定UNBOUNDED\_RANGE，或者行计数间隔要指定UNBOUNDED\_ROW。而有界的over window是用间隔的大小指定的。

实际代码应用如下：

1） 无界的 over window

*// 无界的事件时间 over window*.window(Over.*partitionBy*(**"a"**).orderBy(**"rowtime"**).preceding.(UNBOUNDED\_RANGE).as(**"w"**))  
  
*// 无界的处理时间 over window*.window(Over.*partitionBy*(**"a"**).orderBy(**"proctime"**).preceding.(UNBOUNDED\_RANGE).as(**"w"**))  
  
*// 无界的事件时间 Row-count over window*.window(Over.*partitionBy*(**"a"**).orderBy(**"rowtime"**).preceding.(UNBOUNDED\_ROW).as(**"w"**))  
  
*//无界的处理时间 Row-count over window*.window(Over.*partitionBy*(**"a"**).orderBy(**"proctime"**).preceding.(UNBOUNDED\_ROW).as(**"w"**))

2） 有界的over window

*// 有界的事件时间 over window*.window(Over.*partitionBy*(**"a"**).orderBy(**"rowtime"**).preceding(**"1.minutes"**).as(**"w"**))  
   
*// 有界的处理时间 over window*.window(Over.*partitionBy*(**"a"**).orderBy(**"proctime"**).preceding(**"1.minutes"**).as(**"w"**))  
   
*// 有界的事件时间 Row-count over window*.window(Over.*partitionBy*(**"a"**).orderBy(**"rowtime"**).preceding(**"10.rows"**).as(**"w"**))  
   
*// 有界的处理时间 Row-count over window*.window(Over.*partitionBy*(**"a"**).orderBy(**"procime"**).preceding(**"10.rows"**).as(**"w"**))

## 4.3 SQL中窗口的定义

我们已经了解了在Table API里window的调用方式，同样，我们也可以在SQL中直接加入窗口的定义和使用。

### 4.3.1 Group Windows

Group Windows在SQL查询的Group BY子句中定义。与使用常规GROUP BY子句的查询一样，使用GROUP BY子句的查询会计算每个组的单个结果行。

SQL支持以下Group窗口函数:

* TUMBLE(time\_attr, interval)

定义一个滚动窗口，第一个参数是时间字段，第二个参数是窗口长度。

* HOP(time\_attr, interval, interval)

定义一个滑动窗口，第一个参数是时间字段，第二个参数是窗口滑动步长，第三个是窗口长度。

* SESSION(time\_attr, interval)

定义一个会话窗口，第一个参数是时间字段，第二个参数是窗口间隔（Gap）。

另外还有一些辅助函数，可以用来选择Group Window的开始和结束时间戳，以及时间属性。

这里只写TUMBLE\_\*，滑动和会话窗口是类似的（HOP\_\*，SESSION\_\*）。

* TUMBLE\_START(time\_attr, interval)
* TUMBLE\_END(time\_attr, interval)
* TUMBLE\_ROWTIME(time\_attr, interval)
* TUMBLE\_PROCTIME(time\_attr, interval)

### 4.3.2 Over Windows

由于Over本来就是SQL内置支持的语法，所以这在SQL中属于基本的聚合操作。所有聚合必须在同一窗口上定义，也就是说，必须是相同的分区、排序和范围。目前仅支持在当前行范围之前的窗口（无边界和有边界）。

注意，ORDER BY必须在单一的时间属性上指定。

代码如下：

**SELECT** **COUNT**(amount) OVER (

PARTITION **BY** **user**

**ORDER** **BY** proctime

**ROWS** **BETWEEN** 2 PRECEDING **AND** **CURRENT** **ROW**)

**FROM** Orders

*// 也可以做多个聚合*

**SELECT** **COUNT**(amount) OVER w, **SUM**(amount) OVER w

**FROM** Orders

WINDOW w **AS** (

PARTITION **BY** **user**

**ORDER** **BY** proctime

**ROWS** **BETWEEN** 2 PRECEDING **AND** **CURRENT** **ROW**)

### 4.4 代码练习（以分组滚动窗口为例）

我们可以综合学习过的内容，用一段完整的代码实现一个具体的需求。例如，可以开一个滚动窗口，统计10秒内出现的每个sensor的个数。

代码如下：

**public static void** main(String[] args) **throws** Exception {  
 *// 1. 创建环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 env.setParallelism(1);  
 *// 设置事件时间* env.setStreamTimeCharacteristic(TimeCharacteristic.***EventTime***);  
  
 StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);  
  
 *// 2. 读取文件，得到 DataStream* String filePath = **"..\\sensor.txt"**;  
 DataStream<String> inputStream = env.readTextFile(filePath);  
  
 *// 3. 转换成 Java Bean，并指定timestamp和watermark* DataStream<SensorReading> dataStream = inputStream  
 .map( line -> {  
 String[] fields = line.split(**","**);  
 **return new** SensorReading(fields[0], **new** Long(fields[1]), **new** Double(fields[2]));  
 } )  
 .assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor<SensorReading>(Time.*seconds*(1)) {  
 **@Override  
 public long** extractTimestamp(SensorReading element) {  
 **return** element.getTimestamp() \* 1000L;  
 }  
 });  
  
 *// 4. 将 DataStream 转换为 Table* Table sensorTable = tableEnv.fromDataStream(dataStream, **"id, timestamp.rowtime as ts, temperature"**); *// 5. 开窗聚合* Table resultTable = sensorTable  
 .window(Tumble.*over*(**"10.seconds"**).on(**"ts"**).as(**"tw"**))  
 *// 每10秒统计一次，滚动时间窗口* .groupBy(**"id, tw"**)  
 .select(**"id, id.count, temperature.avg, tw.end"**);  
  
 *// sql* tableEnv.createTemporaryView(**"sensor"**, sensorTable);  
 Table resultSqlTable = tableEnv.sqlQuery(  
 **"select id, count(id), avg(temperature), tumble\_end(ts, interval '10' second) "** +  
 **"from sensor group by id, tumble(ts, interval '10' second)"**);  
  
 *// 转换成流打印输出* tableEnv.toAppendStream(resultTable, Row.**class**).print(**"result"**);  
 tableEnv.toAppendStream(resultSqlTable, Row.**class**).print(**"sql"**);  
  
 env.execute(**"time and window test"**);  
 }

# 第五章 函数（Functions）

Flink Table 和 SQL内置了很多SQL中支持的函数；如果有无法满足的需要，则可以实现用户自定义的函数（UDF）来解决。

## 5.1 系统内置函数

Flink Table API 和 SQL为用户提供了一组用于数据转换的内置函数。SQL中支持的很多函数，Table API和SQL都已经做了实现，其它还在快速开发扩展中。

以下是一些典型函数的举例，全部的内置函数，可以参考官网介绍。

* 比较函数

SQL：

value1 = value2

value1 > value2

Table API：

ANY1 === ANY2

ANY1 > ANY2

* 逻辑函数

SQL：

boolean1 OR boolean2

boolean IS FALSE

NOT boolean

Table API：

BOOLEAN1 || BOOLEAN2

BOOLEAN.isFalse

!BOOLEAN

* 算术函数

SQL：

numeric1 + numeric2

POWER(numeric1, numeric2)

Table API：

NUMERIC1 + NUMERIC2

NUMERIC1.power(NUMERIC2)

* 字符串函数

SQL：

string1 || string2

UPPER(string)

CHAR\_LENGTH(string)

Table API：

STRING1 + STRING2

STRING.upperCase()

STRING.charLength()

* 时间函数

SQL：

DATE string

TIMESTAMP string

CURRENT\_TIME

INTERVAL string range

Table API：

STRING.toDate

STRING.toTimestamp

currentTime()

NUMERIC.days

NUMERIC.minutes

* 聚合函数

SQL：

COUNT(\*)

SUM([ ALL | DISTINCT ] expression)

RANK()

ROW\_NUMBER()

Table API：

FIELD.count

FIELD.sum0

## 5.2 UDF

用户定义函数（User-defined Functions，UDF）是一个重要的特性，因为它们显著地扩展了查询（Query）的表达能力。一些系统内置函数无法解决的需求，我们可以用UDF来自定义实现。

### 5.2.1 注册用户自定义函数UDF

在大多数情况下，用户定义的函数必须先注册，然后才能在查询中使用。不需要专门为Scala 的Table API注册函数。

函数通过调用registerFunction（）方法在TableEnvironment中注册。当用户定义的函数被注册时，它被插入到TableEnvironment的函数目录中，这样Table API或SQL解析器就可以识别并正确地解释它。

### 5.2.2 标量函数（Scalar Functions）

用户定义的标量函数，可以将0、1或多个标量值，映射到新的标量值。

为了定义标量函数，必须在org.apache.flink.table.functions中扩展基类Scalar Function，并实现（一个或多个）求值（evaluation，eval）方法。标量函数的行为由求值方法决定，求值方法必须公开声明并命名为eval（直接def声明，没有override）。求值方法的参数类型和返回类型，确定了标量函数的参数和返回类型。

在下面的代码中，我们定义自己的HashCode函数，在TableEnvironment中注册它，并在查询中调用它。

*// 自定义一个标量函数***public static class** HashCode **extends** ScalarFunction {  
 **private int factor** = 13;  
  
 **public** HashCode(**int** factor) {  
 **this**.**factor** = factor;  
 }  
  
 **public int** eval(String s) {  
 **return** s.hashCode() \* **factor**;  
 }  
}

主函数中调用，计算sensor id的哈希值（前面部分照抄，流环境、表环境、读取source、建表）：

**public static void** main(String[] args) **throws** Exception {  
 *// 1. 创建环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 env.setParallelism(1);  
  
 StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);  
  
 *// 2. 读取文件，得到 DataStream* String filePath = **"..\\sensor.txt"**;  
  
 DataStream<String> inputStream = env.readTextFile(filePath);  
  
 *// 3. 转换成 Java Bean，并指定timestamp和watermark* DataStream<SensorReading> dataStream = inputStream  
 .map( line -> {  
 String[] fields = line.split(**","**);  
 **return new** SensorReading(fields[0], **new** Long(fields[1]), **new** Double(fields[2]));  
 } );  
  
 *// 4. 将 DataStream 转换为 Table* Table sensorTable = tableEnv.fromDataStream(dataStream, **"id, timestamp as ts, temperature"**);  
  
 *// 5. 调用自定义hash函数，对id进行hash运算* HashCode hashCode = **new** HashCode(23);  
 tableEnv.registerFunction(**"hashCode"**, hashCode);  
 Table resultTable = sensorTable  
 .select(**"id, ts, hashCode(id)"**);  
  
 *// sql* tableEnv.createTemporaryView(**"sensor"**, sensorTable);  
 Table resultSqlTable = tableEnv.sqlQuery(**"select id, ts, hashCode(id) from sensor"**);  
  
 tableEnv.toAppendStream(resultTable, Row.**class**).print(**"result"**);  
 tableEnv.toRetractStream(resultSqlTable, Row.**class**).print(**"sql"**);  
  
 env.execute(**"scalar function test"**);  
}

### 5.2.3 表函数（Table Functions）

与用户定义的标量函数类似，用户定义的表函数，可以将0、1或多个标量值作为输入参数；与标量函数不同的是，它可以返回任意数量的行作为输出，而不是单个值。

为了定义一个表函数，必须扩展org.apache.flink.table.functions中的基类TableFunction并实现（一个或多个）求值方法。表函数的行为由其求值方法决定，求值方法必须是public的，并命名为eval。求值方法的参数类型，决定表函数的所有有效参数。

返回表的类型由TableFunction的泛型类型确定。求值方法使用protected collect（T）方法发出输出行。

在Table API中，Table函数需要与.joinLateral或.leftOuterJoinLateral一起使用。

joinLateral算子，会将外部表中的每一行，与表函数（TableFunction，算子的参数是它的表达式）计算得到的所有行连接起来。

而leftOuterJoinLateral算子，则是左外连接，它同样会将外部表中的每一行与表函数计算生成的所有行连接起来；并且，对于表函数返回的是空表的外部行，也要保留下来。

在SQL中，则需要使用Lateral Table（<TableFunction>），或者带有ON TRUE条件的左连接。

下面的代码中，我们将定义一个表函数，在表环境中注册它，并在查询中调用它。

自定义TableFunction：

*// 自定义TableFunction***public static class** Split **extends** TableFunction<Tuple2<String, Integer>> {  
 **private** String **separator** = **","**;  
  
 **public** Split(String separator) {  
 **this**.**separator** = separator;  
 }  
  
 *// 类似flatmap，没有返回值* **public void** eval(String str) {  
 **for** (String s : str.split(**separator**)) {  
 collect(**new** Tuple2<String, Integer>(s, s.length()));  
 }  
 }  
}

接下来，就是在代码中调用。首先是Table API的方式：

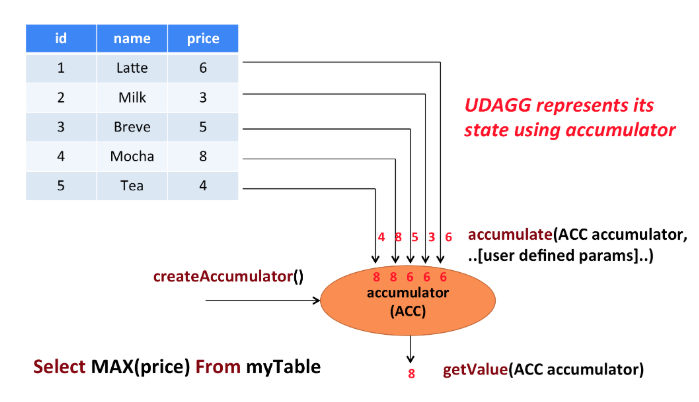
Split split = **new** Split(**"\_"**);  
tableEnv.registerFunction(**"split"**, split);  
Table resultTable = sensorTable  
 .joinLateral( **"split(id) as (word, length)"**)  
 .select(**"id, ts, word, length"**);

然后是SQL的方式：

tableEnv.createTemporaryView(**"sensor"**, sensorTable);  
Table resultSqlTable = tableEnv.sqlQuery(**"select id, ts, word, length "** +  
 **"from sensor, lateral table( split(id) ) as splitId(word, length)"**);

### 5.2.4 聚合函数（Aggregate Functions）

用户自定义聚合函数（User-Defined Aggregate Functions，UDAGGs）可以把一个表中的数据，聚合成一个标量值。用户定义的聚合函数，是通过继承AggregateFunction抽象类实现的。



上图中显示了一个聚合的例子。

假设现在有一张表，包含了各种饮料的数据。该表由三列（id、name和price）、五行组成数据。现在我们需要找到表中所有饮料的最高价格，即执行max（）聚合，结果将是一个数值。

AggregateFunction的工作原理如下。

* 首先，它需要一个累加器，用来保存聚合中间结果的数据结构（状态）。可以通过调用AggregateFunction的createAccumulator（）方法创建空累加器。
* 随后，对每个输入行调用函数的accumulate（）方法来更新累加器。
* 处理完所有行后，将调用函数的getValue（）方法来计算并返回最终结果。

AggregationFunction要求必须实现的方法：

* createAccumulator()
* accumulate()
* getValue()

除了上述方法之外，还有一些可选择实现的方法。其中一些方法，可以让系统执行查询更有效率，而另一些方法，对于某些场景是必需的。例如，如果聚合函数应用在会话窗口（session group window）的上下文中，则merge（）方法是必需的。

* retract()
* merge()
* resetAccumulator()

接下来我们写一个自定义AggregateFunction，计算一下每个sensor的平均温度值。

*// 定义AggregateFunction的Accumulator***public static class** AvgTempAcc {  
 **double sum** = 0.0;  
 **int count** = 0;  
}  
  
*// 自定义一个聚合函数，求每个传感器的平均温度值，保存状态(tempSum, tempCount)***public static class** AvgTemp **extends** AggregateFunction<Double, AvgTempAcc>{  
 **@Override  
 public** Double getValue(AvgTempAcc accumulator) {  
 **return** accumulator.**sum** / accumulator.**count**;  
 }  
  
 **@Override  
 public** AvgTempAcc createAccumulator() {  
 **return new** AvgTempAcc();  
 }  
  
 *// 实现一个具体的处理计算函数，accumulate* **public void** accumulate( AvgTempAcc accumulator, Double temp) {  
 accumulator.**sum** += temp;  
 accumulator.**count** += 1;  
 }  
}

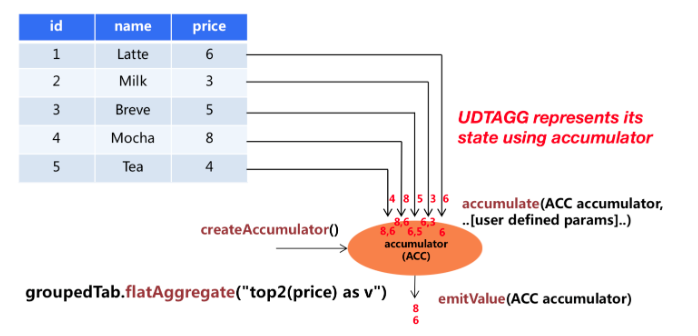
接下来就可以在代码中调用了。

*// 创建一个聚合函数实例*AvgTemp avgTemp = **new** AvgTemp();

*// Table API的调用*   
tableEnv.registerFunction(**"avgTemp"**, avgTemp);  
Table resultTable = sensorTable  
 .groupBy(**"id"**)  
 .aggregate(**"avgTemp(temperature) as avgTemp"**)  
 .select(**"id, avgTemp"**);  
*// sql*tableEnv.createTemporaryView(**"sensor"**, sensorTable);  
Table resultSqlTable = tableEnv.sqlQuery(**"select id, avgTemp(temperature) "** +  
 **"from sensor group by id"**);  
  
tableEnv.toRetractStream(resultTable, Row.**class**).print(**"result"**);  
tableEnv.toRetractStream(resultSqlTable, Row.**class**).print(**"sql"**);

### 5.2.5 表聚合函数（Table Aggregate Functions）

用户定义的表聚合函数（User-Defined Table Aggregate Functions，UDTAGGs），可以把一个表中数据，聚合为具有多行和多列的结果表。这跟AggregateFunction非常类似，只是之前聚合结果是一个标量值，现在变成了一张表。



比如现在我们需要找到表中所有饮料的前2个最高价格，即执行top2（）表聚合。我们需要检查5行中的每一行，得到的结果将是一个具有排序后前2个值的表。

用户定义的表聚合函数，是通过继承TableAggregateFunction抽象类来实现的。

TableAggregateFunction的工作原理如下。

* 首先，它同样需要一个累加器（Accumulator），它是保存聚合中间结果的数据结构。通过调用TableAggregateFunction的createAccumulator（）方法可以创建空累加器。
* 随后，对每个输入行调用函数的accumulate（）方法来更新累加器。
* 处理完所有行后，将调用函数的emitValue（）方法来计算并返回最终结果。

AggregationFunction要求必须实现的方法：

* createAccumulator()
* accumulate()

除了上述方法之外，还有一些可选择实现的方法。

* retract()
* merge()
* resetAccumulator()
* emitValue()
* emitUpdateWithRetract()

接下来我们写一个自定义TableAggregateFunction，用来提取每个sensor最高的两个温度值。

*// 先定义一个 Accumulator***public static class** Top2TempAcc {  
 **double highestTemp** = Double.***MIN\_VALUE***;  
 **double secondHighestTemp** = Double.***MIN\_VALUE***;  
}  
  
*// 自定义表聚合函数***public static class** Top2Temp **extends** TableAggregateFunction<Tuple2<Double, Integer>, Top2TempAcc> {  
 **@Override  
 public** Top2TempAcc createAccumulator() {  
 **return new** Top2TempAcc();  
 }  
  
 *// 实现计算聚合结果的函数accumulate* **public void** accumulate(Top2TempAcc acc, Double temp) {**if** (temp > acc.**highestTemp**) {acc.**secondHighestTemp** = acc.**highestTemp**;  
 acc.**highestTemp** = temp;  
 } **else if** (temp > acc.**secondHighestTemp**) {acc.**secondHighestTemp** = temp;  
 }  
 }  
 *// 实现一个输出结果的方法，最终处理完表中所有数据时调用* **public void** emitValue(Top2TempAcc acc, Collector<Tuple2<Double, Integer>> out) {  
 out.collect(**new** Tuple2<>(acc.**highestTemp**, 1));  
 out.collect(**new** Tuple2<>(acc.**secondHighestTemp**, 2));  
 }  
}

接下来就可以在代码中调用了。

*// 创建一个表聚合函数实例*Top2Temp top2Temp = **new** Top2Temp();  
tableEnv.registerFunction(**"top2Temp"**, top2Temp);  
Table resultTable = sensorTable  
 .groupBy(**"id"**)  
 .flatAggregate(**"top2Temp(temperature) as (temp, rank)"**)  
 .select(**"id, temp, rank"**);  
  
tableEnv.toRetractStream(resultTable, Row.**class**).print(**"result"**);