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**>

1. 执行计划
2. 桥接器

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, flink-table-api-java-bridge：brid ge桥接器，**将表转为流，流转为表。**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。

### 1.2.1 牛刀小试

Flink SQL可以在流上写SQL

public class \_01\_小试牛刀 {  
 //*todo 读取端口数据 用SQL形式写出* public static void main(String[] args) throws Exception {  
  
 StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
 //*todo 1.获取tableAPI的执行环境* StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);  
  
 //*todo 流转换，将读取端口数据转为javaBean 方便flink SQL使用* SingleOutputStreamOperator<SensorReading> source = env.socketTextStream("hadoop102", 9999).map(  
 new MapFunction<String, SensorReading>() {  
 @Override  
 public SensorReading map(String value) throws Exception {  
 String[] fields = value.split(",");  
 return new SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  
 }  
 }  
 );  
  
  //*todo 2 SQL方式* // *2.1 从DataStream注册表* // *没有返回值，参数1是注册的表名，参数2是DataStream的流名* // *注意：DataStream中的数据结构要封装成JavaBean，这样属性对应表的字段* tableEnv.createTemporaryView("sensor",source);  
 // *todo 2.2 用SQL查询，结果是一个Table对象* Table sqlTable1 = tableEnv.sqlQuery("select id,temp from sensor where id = 'sensor\_1'");  
  
 //*todo 2 TableAPI方式* // *从DataStream中创建表* //Table table = tableEnv.fromDataStream(source);//*todo 从流中直接获取table* Table table = tableEnv.from("sensor"); // *todo 从table中创建table 前提是必须tableEnv中注册了table* Table sqlTable2 = table.where("id='sensor\_1'").select("id,temp");  
  
  
 //*todo SQL和TableAPI的区别* // *SQL: step1 在tableEnv中注册表* // *step2 sqlQuery 写sql语句* // *flink SQL 都是基于tableEnv来调用的* // *TableAPI : step1 在tableEnv中创建表，获取Table对象做原始表* // *step2. 基于Table对象调用API* //*todo 3.将表转换为流输出* // *toAppendStream(表名，按照什么类型输出） row表示不是一个普通javaBean,是一行数据* // *获取Row中数据的时候 通过 .getString(字段对应的索引)来获得* tableEnv.toAppendStream(sqlTable1, Row.class).print("SQL");  
 tableEnv.toAppendStream(sqlTable2, Row.class).print("table");  
  
 env.execute();  
  
 }  
}

|  |
| --- |
| **import** com.atguigu.bean.SensorReading; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.types.Row;  **public class** FlinkSQL\_Test01 {   **public static void** main(String[] args) **throws** Exception {   *//1.获取执行环境并设置并行度* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);   *//2.读取文本数据创建流* DataStreamSource<String> readTextFile = env.readTextFile(**"sensor"**);   *//3.将每一行数据转换为JavaBean* SingleOutputStreamOperator<SensorReading> sensorDataStream = readTextFile.map(line -> {  String[] fields = line.split(**","**);  **return new** SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  });   *//4.创建TableAPI执行环境* StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//5.从流中创建表* **Table table = tableEnv.fromDataStream(sensorDataStream);**   *//6.转换数据  //6.1 使用TableAPI转换数据* Table result = table.select(**"id,temp"**).filter(**"id = 'sensor\_1'"**);   *//6.2 使用FlinkSQL转换数据* tableEnv.createTemporaryView(**"sensor"**, sensorDataStream);  Table sqlResult = tableEnv.sqlQuery(**"select id,temp from sensor where id='sensor\_1'"**);   *//7.转换为流输出数据* tableEnv.toAppendStream(result, Row.**class**).print(**"result"**);  tableEnv.toAppendStream(sqlResult, Row.**class**).print(**"sql"**);   *//8.启动任务* env.execute();  }  } |

### 1.2.2 flink SQL实现wordCount

（1）tableAPi调用函数的顺序遵循SQL的执行顺序，因此group()要在select()前面

（2）撤回流和追加流的应用场景不同：追加流是来一条数据计算结果就要输出去，而撤回流适合聚合场景，下一条数据的计算结果要覆盖上一条的计算结果

public class \_01\_小试牛刀2\_分组count {  
  
 public static void main(String[] args) throws Exception {  
  
  
 StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  
  
 StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);  
  
 SingleOutputStreamOperator<SensorReading> source = env.socketTextStream("hadoop102", 9999).map(  
 new MapFunction<String, SensorReading>() {  
 @Override  
 public SensorReading map(String value) throws Exception {  
 String[] fields = value.split(",");  
 return new SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  
 }  
 }  
 );  
  
 //*todo 1.SQL方式 通过流来创建表* tableEnv.createTemporaryView("sensor",source); //path是表名,后面是DataStream流名  
 Table sqlTable1 = tableEnv.sqlQuery("select id,count(id) from sensor group by id");  
  
 //*todo 2.TableAPI方式* Table table = tableEnv.fromDataStream(source);  
 Table sqlTable2 = table.groupBy("id").select("id,id.count");  
 //*todo 注意：tableAPi的调用顺序遵循SQL的执行顺序，分组是在select前面的* // tableEnv.toAppendStream(sqlTable1, Row.class).print("SQL");  
 // tableEnv.toAppendStream(sqlTable2, Row.class).print("table");  
 //*todo 执行报错 table is not an append-only table (表不是appendOnly表)* // *第一条数据进来 sensor\_1 1 第二条数据进来 sensor\_1 2* // *如果两条计算结果都直接放进流中两个都输出，造成数据重复，* // *因此不能用追加流，用撤回流，将1撤回，再将2输出* tableEnv.toRetractStream(sqlTable1, Row.class).print("SQL");  
 tableEnv.toRetractStream(sqlTable2, Row.class).print("table");  
  
 //*todo 打印结果 用标记位显示过期和有效数据* // *用true表示当前有效数据，false表示过期数据将来要撤回* // *printSink 无法过滤掉过期数据，用Mysql、ES 遇到false就能删掉过期数据* // *=>* // *输入：sensor\_1,1547718209,15* // *table:1> (true,sensor\_1,1)* // *SQL:1> (true,sensor\_1,1)* // *再输入：sensor\_1,1547718209,15* // *SQL:1> (false,sensor\_1,1)* // *SQL:1> (true,sensor\_1,2)* // *table:1> (false,sensor\_1,1)* // *table:1> (true,sensor\_1,2)* // *在输入：sensor\_1,1547718209,15* // *SQL:1> (false,sensor\_1,2)* // *SQL:1> (true,sensor\_1,3)* // *table:1> (false,sensor\_1,2)* // *table:1> (true,sensor\_1,3)* env.execute();  
 }  
}

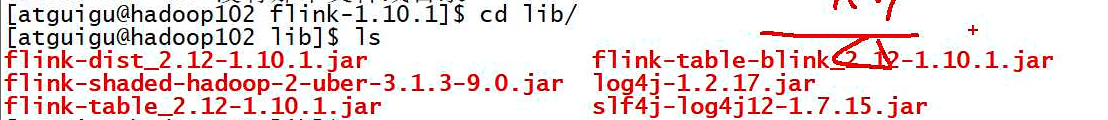
撤回流：如果没有更新，就不进行操作，就没有输出。

比如求max，上一条是36 这一条是35，那么不更新，没有撤回也没有输出

## 1.3 两种planner（old & blink）的区别

（1）blink是新的flinkSql运行环境

（2）blink是阿里修改源码后增加了一些功能，blink相当于一个新的客户端，在flink1.8就已经有了，在flink/lib目录下，能够看到blink的jar包，**1.11版本功能完善了，才推荐使用，1.10环境创建默认是老的，也可以手动创建blink环境**



（3）blink **批流统一**

**blink执行计划只有一个，将批处理作业，视为流式处理的特殊情况。**

**如果用flink做批处理，基本的数据流是DataSet，做流处理，数据流为DataStream；而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。

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

（4）Blink planner只支持全新的目录，不支持已弃用的ExternalCatalog(负责建表的)。

（5）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。（sql执行的优化方式发生改变）

A join B on xxx where xxx => 执行顺序是先走on过滤，再join，最后where筛选；join之前过滤会提高性能，将where过滤on中，先对A表进行过滤，然后再做join。如果是A.id和b.id做join，where中尽管只写了A.id > 10 ,照样也会根据id对B表进行过滤。 这就是谓词下推，就是优化。

在Hive中用explain可以看sql的执行计划

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。

9. 使用Blink需额外指定依赖

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

# 第二章 API调用

## [1] FlinkSQL的开发流程

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

//第一步：创建表的执行环境

StreamTableEnvironment tableEnv = ...

//第二步：创建一张表，用于读取数据tableEnv.connect(...).createTemporaryTable(**"inputTable"**);  
// 第三步：创建一张表，用于把计算结果输出tableEnv.connect(...).createTemporaryTable(**"outputTable"**);  
//第四步：查询获得结果表（可以用SQl也可以用TableAPI）

*//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 ..."**);  
//第五步：将结果表写入输出表中result.**insertInto**(**"outputTable"**);

**（1）tableEnv.connect(...).createTemporaryTable("inputTable");** connect()连接，参数是个连接器，可以连接Kafka\mysql等各个外部系统，不同的外部系统有不同的连接器，连接器有的只可读，有的只可写，有的可读可写，而且读写的数据类型还有限制。

（2）获取结果表：tableEnv.from，返回Table 将结果表插入出去:table.insertInto，没有返回值

## [2] flink SQL 执行环境的创建

**（1）最常用的创建方式**

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);

根据版本自己创建环境。1.11版本自动获取blink环境，1.10获取old环境

**（2）表环境的功能**

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

* 注册catalog (一个目录，相当于元数据表)
* 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(tableEnv的原始环境，用于批处理或者流处理)

**说明：标红的表示在flinkSQL开发流程中，需要我们自己写的代码内容。**

**（3）不同的表环境的创建**

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：

（1）配置老版本的流式处理环境（Flink-Streaming-Query）：

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

EnvironmentSettingssettings = EnvironmentSettings.*newInstance*()  
 .useOldPlanner() *// 使用老版本planner(提供运行时环境和执行计划)* .inStreamingMode() *// 流处理模式* .build();  
StreamTableEnvironmenttableEnv = StreamTableEnvironment.*create*(env, settings);

**额外 说明：在flink1.10版本中，默认使用就是老版本的流处理环境，因此可以不配置settings，如下代码所示：**

**StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env);**

（2）基于老版本的批处理环境（Flink-Batch-Query）：

ExecutionEnvironmentbatchEnv = ExecutionEnvironment.*getExecutionEnvironment();*

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

流批环境的区别：

flink环境 env 流是StreamExecution... 批是Execution...

（3）基于blink版本的流处理环境（Blink-Streamin g-Query）：

StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

EnvironmentSettings bsSettings = EnvironmentSettings.*newInstance*()

.**useBlinkPlanner()**

.inStreamingMode()

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

（4）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);

额外说明：Blink的这种方式实际上是用有界流的处理方式。

flink开发，关注流处理。

## [3]在catalog中注册表（输入表）

注册表就是在flink SQL环境中获取数据源source

### 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。

* **表有两种：可以是常规的（Table，表），或者虚拟的（View，视图）。**

**常规表（Table）：一般从外部系统直接获取，或者从 DataStream转换而来。**

**视图：从现有的表中创建**，通常是table API或者SQL查询的一个结果。

**表和视图在flinkSQL环境下，都是Table类**

### 表的创建

Flink SQL创建表有两种方式，一种是从dataStream中注册表，还可以直接对接外部系统，来注册表。

从dataStream中注册表是实际生产中最常用的方式。

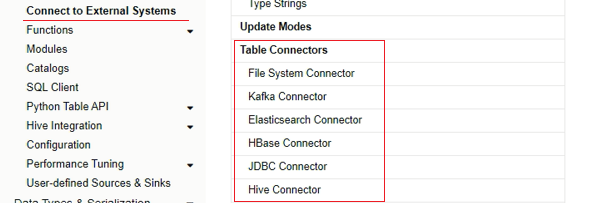
如果通过连接外部系统在Catalog中注册表，直接调用 tableEnv.connect()就可以，里面参数要传入一个ConnectorDescriptor，也就是connector描述器。

tableEnv.connect(...).createTemporaryTable(**"inputTable"**);

#### 从外部系统来创建表

对接外部系统直接获取表，需要用connector

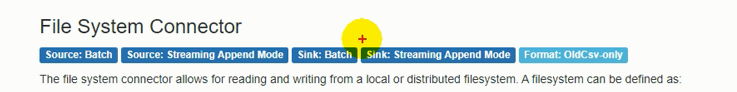
官网：https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/connectors/



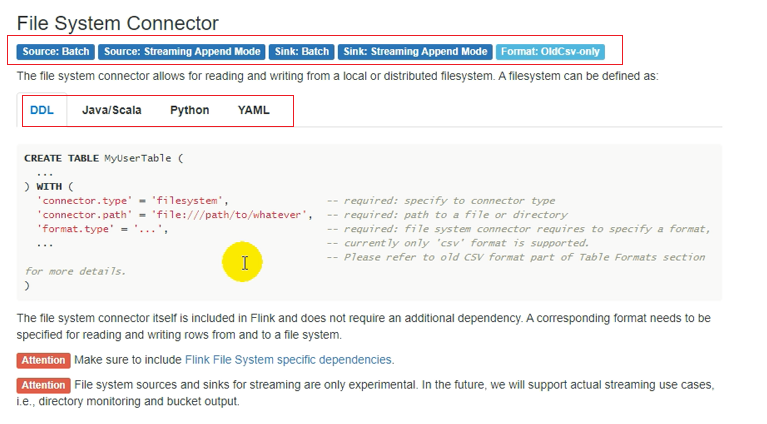
**Connector主要性能在如下四个方面：**

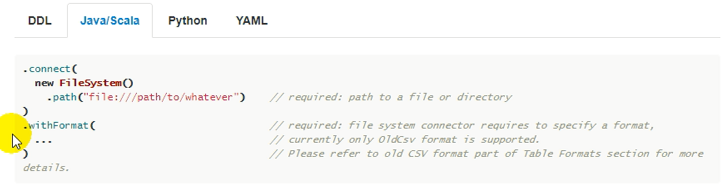
1. Source&Soucemode
2. Sink&sinkMode
3. 支持的数据格式format
4. Connector创建方式

以文件系统Connector为例：



能做批处理的source，也可以做追加流的source，也可以做批处理的sink和追加流的sink；格式只能为OldCsv





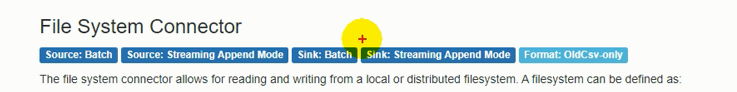
Connector直接连接外部系统，然后创建表，由于有格式限制，所以一般都要先处理成javaBean的形式再创建表

Upsert Mode就是更新流

如果要用哪个Connector，一般都要导包；

用connector来连接的方式有很多种，DDL是通用的，也就是在flinkSQL中写SQL连接；对于MYSQL来说，只能用DDL方式连接；还可以用Java代码来连接

##### 示例1：对接外部文件系统



只能支持OldCsv格式

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：

|  |
| --- |
| **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.DataTypes; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.table.descriptors.FileSystem; **import** org.apache.flink.table.descriptors.OldCsv; **import** org.apache.flink.table.descriptors.Schema; **import** org.apache.flink.types.Row;  **public class** FlinkSQL\_Test02 {   **public static void** main(String[] args) **throws** Exception {   *//1.获取执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();   *//2.设置并行度* env.setParallelism(1);   *//3.创建* StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//4.读取文件数据创建表* tableEnv.connect(**new** FileSystem().path(**"input/sensor.txt"**))  .withFormat(**new** OldCsv())  .withSchema(**new** Schema().field(**"id"**, DataTypes.*STRING*())  .field(**"ts"**, DataTypes.*BIGINT*())  .field(**"temp"**, DataTypes.*DOUBLE*()))   **.createTemporaryTable("inputTable");**   *//5.执行SQL查询数据* Table table = tableEnv.sqlQuery(**"select id,temp from inputTable where id='sensor\_1'"**);   *//6.将表转换为追加流并打印* tableEnv.toAppendStream(table, Row.**class**).print();   *//7.执行任务* env.execute();  }  } |

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。

##### 示例2：对接Kafka



Avro是自定义格式，支持source和sink，只支持追加刘

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。

|  |
| --- |
| **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.DataTypes; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.table.descriptors.Csv; **import** org.apache.flink.table.descriptors.Kafka; **import** org.apache.flink.table.descriptors.Schema; **import** org.apache.flink.types.Row;  **public class** FlinkSQL\_Test03 {   **public static void** main(String[] args) **throws** Exception {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2.定义Kafka连接描述器* Kafka kafka = **new** Kafka()  .version(**"0.11"**) //kafka连接器的版本，指适合kafka的0.11及以上  .topic(**"test"**)  .property(**"bootstrap.servers"**, **"hadoop102:9092"**);   *//3.定义表的Schema信息* Schema schema = **new** Schema()  .field(**"id"**, DataTypes.*STRING*())  .field(**"ts"**, DataTypes.*BIGINT*())  .field(**"temp"**, DataTypes.*DOUBLE*());   *//4.读取Kafka数据创建临时表* tableEnv.connect(kafka).withFormat(**new** Csv()).withSchema(schema).createTemporaryTable(**"KafkaTable"**);   *//5.执行SQL查询数据* Table table = tableEnv.sqlQuery(**"select id,temp from KafkaTable where id='sensor\_1'"**);   *//6.将表转换为追加流进行打印* tableEnv.toAppendStream(table, Row.**class**).print();   *//7.执行任务* env.execute();   }  } |

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。

#### 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。

##### 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.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。

## [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。

### 2.5.1 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'"**);

### 2.5.2 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"**);

## [5]创建临时视图（Temporary View）

创建临时视图有两种方式，第一种方式，就是直接从DataStream转换而来。

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"**);

第二种方式：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。

## [6]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 将表转换成DataStream打印

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。

代码实现如下：

|  |
| --- |
| **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.DataTypes; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.table.descriptors.FileSystem; **import** org.apache.flink.table.descriptors.OldCsv; **import** org.apache.flink.table.descriptors.Schema;  **import** org.apache.flink.types.Row;  **public class** FlinkSQL\_Test04 {   **public static void** main(String[] args) **throws** Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);   *//1. 创建表环境* StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2. 连接外部系统，读取数据，创建表  //2.1 读取文件* String filePath = **"D:\\Projects\\BigData\\FlinkTurtorial\\src\\main\\resources\\sensor.txt"**;  tableEnv.connect(**new** FileSystem().path(filePath))  .withFormat(**new** OldCsv())  .withSchema(  **new** Schema()  .field(**"id"**, DataTypes.*STRING*())  .field(**"ts"**, DataTypes.*BIGINT*())  .field(**"temp"**, DataTypes.*DOUBLE*())  )  .createTemporaryTable(**"inputTable"**);   *//3. 查询转换  //3.1 简单查询  //3.1.1 Table API* Table inputTable = tableEnv.from(**"inputTable"**);  Table resultTable = inputTable.select(**"id, tem"**)  .filter(**"id === 'sensor\_1'"**);   *//3.1.2 SQL* Table resultSqlTable = tableEnv.sqlQuery(**"select id, temperature from inputTable where id = 'sensor\_1'"**);   *//3.2 聚合统计  //3.2.1 Table API* Table aggTable = inputTable  .groupBy(**"id"**)  .select(**"id, id.count as count"**);    *//3.2.2 SQL* Table aggSqlTable = tableEnv.sqlQuery(**"select id, count(id) as ct from inputTable group by id"**);   *// 4. 转换成流打印输出* tableEnv.toAppendStream(resultTable, Row.**class**).print(**"result"**);  tableEnv.toAppendStream(resultSqlTable, Row.**class**).print(**"resultSQL"**);  tableEnv.toRetractStream(aggTable, Row.**class**).print(**"agg"**);  tableEnv.toRetractStream(aggSqlTable, Row.**class**).print(**"aggSQL"**);   env.execute();  } } |

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。

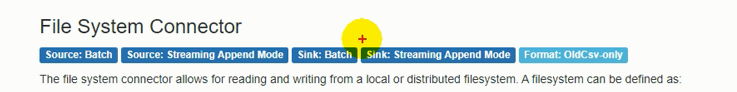
* **撤回模式（Retract Mode）**

**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）。

**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。**

### 2.7.2 输出到文件



**只能用追加SQL**

代码如下：

|  |
| --- |
| **import** com.atguigu.bean.SensorReading; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.DataTypes; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.table.descriptors.Csv; **import** org.apache.flink.table.descriptors.FileSystem; **import** org.apache.flink.table.descriptors.Schema;  **public class** FlinkSQL\_Test05 {   **public static void** main(String[] args) **throws** Exception {   *//1.获取执行环境并设置并行度* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);   *//2.读取文本数据创建流* DataStreamSource<String> readTextFile = env.readTextFile(**"sensor"**);   *//3.将每一行数据转换为JavaBean* SingleOutputStreamOperator<SensorReading> sensorDataStream = readTextFile.map(line -> {  String[] fields = line.split(**","**);  **return new** SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  });   *//4.创建TableAPI执行环境* StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//5.从流中创建表* Table table = tableEnv.fromDataStream(sensorDataStream);   *//6.转换数据* Table result = table.select(**"id,temperature"**).filter(**"id = 'sensor\_1'"**);   *//7.将数据写入本地文件系统* tableEnv.connect(  **new** FileSystem().path(**"output/out.txt"**)) *//定义文件系统连接,文件不能存在* .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* tableEnv.insertInto(**"outputTable"**, result);   *//8.执行任务* env.execute();   } } |

### 2.7.3 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。

#### 撤回模式（Retract Mode）

用于打印

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。

#### Upsert（更新插入）模式

**用于对接外部系统有幂等性的，无则插入，有则更新**

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。

### 2.7.4 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：

|  |
| --- |
| **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.DataTypes; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.table.descriptors.Csv; **import** org.apache.flink.table.descriptors.Kafka; **import** org.apache.flink.table.descriptors.Schema; **import** org.apache.flink.types.Row;  **public class** FlinkSQL\_Test06 {   **public static void** main(String[] args) **throws** Exception {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2.定义Kafka输入描述器* Kafka kafkaInput = **new** Kafka()  .version(**"0.11"**)  .topic(**"test"**)  .property(**"bootstrap.servers"**, **"hadoop102:9092"**);  Schema schemaInput = **new** Schema()  .field(**"id"**, DataTypes.*STRING*())  .field(**"ts"**, DataTypes.*BIGINT*())  .field(**"temp"**, DataTypes.*DOUBLE*());   *//3.读取Kafka数据创建临时表* tableEnv.connect(kafkaInput).withFormat(**new** Csv()).withSchema(schemaInput).createTemporaryTable(**"KafkaInput"**);   *//4.执行SQL查询数据* Table table = tableEnv.sqlQuery(**"select id,temp from KafkaInput where id='sensor\_1'"**);   *//5.定义Kafka输出连接器* Kafka kafkaOutPut = **new** Kafka()  .version(**"0.11"**)  .topic(**"sinkTest"**)  .property(**"bootstrap.servers"**, **"hadoop102:9092"**);  Schema schemaOutPut = **new** Schema()  .field(**"id"**, DataTypes.*STRING*())  .field(**"temp"**, DataTypes.*DOUBLE*());  tableEnv.connect(kafkaOutPut).withFormat(**new** Csv()).withSchema(schemaOutPut).createTemporaryTable(**"KafkaOutPut"**);   *//6.将数据输出到Kafka的另一个主题* tableEnv.insertInto(**"KafkaOutPut"**, table);   *//7.执行任务* env.execute();   }  } |

### 2.7.5 输出到ElasticSearch



**Es有两种输入模式：追加和更新或插入**

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。

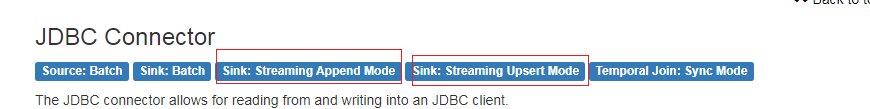
es目前支持的数据格式，只有Json，而flink本身并没有对应的支持，所以还需要引入依赖：

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

代码实现如下：

|  |
| --- |
| **import** com.atguigu.bean.SensorReading; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.DataTypes; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.table.descriptors.Elasticsearch; **import** org.apache.flink.table.descriptors.FormatDescriptor; **import** org.apache.flink.table.descriptors.Json; **import** org.apache.flink.table.descriptors.Schema;  **import** java.util.Map;  **public class** FlinkSQL\_Test07 {   **public static void** main(String[] args) **throws** Exception {   *//1.获取执行环境并设置并行度* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);   *//2.读取文本数据创建流* DataStreamSource<String> readTextFile = env.readTextFile(**"sensor"**);   *//3.将每一行数据转换为JavaBean* SingleOutputStreamOperator<SensorReading> sensorDataStream = readTextFile.map(line -> {  String[] fields = line.split(**","**);  **return new** SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  });   *//4.创建TableAPI执行环境* StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//5.从流中创建表* Table table = tableEnv.fromDataStream(sensorDataStream);   *//6.转换数据* tableEnv.createTemporaryView(**"sensor"**, table);  Table sqlQuery = tableEnv.sqlQuery(**"select id,count(id) as ct from sensor group by id"**);   *//7.构建Es连接器* tableEnv.connect(**new** Elasticsearch()  .host(**"hadoop102"**, 9200, **"http"**)  .version(**"6"**)  .index(**"flink\_index"**)  .documentType(**"\_doc"**))  **.inUpsertMode()**  .withFormat(**new** Json())  .withSchema(**new** Schema()  .field(**"id"**, DataTypes.*STRING*())  .field(**"ct"**, DataTypes.*BIGINT*()))  .createTemporaryTable(**"EsIndex"**);   *//8.将数据写入ES* tableEnv.insertInto(**"EsIndex"**, sqlQuery);   *//9.开启任务* env.execute();   } }  额外说明：  ES 用追加流模式，那么添加进去的数据的doc\_id是随机分配的    用更新流的话会**按照groupBy字段作为doc\_id**，因为更新模式必须要有key才能实现无则插入有则更新    （2）如果分组字段是多个，那么doc\_id也是多个，分隔符默认是“\_”，可以在connector中配置，如下所示          //*todo 注意：* // *(1)更新模式，可以用聚合函数 ES会将分组字段作为doc\_id 实现有则更新无则插入* // *(2)不能同时用两个流插入* // *(3)更新模式 用的是撤销流，ES的更新是取决于撤销流前面的标志位 true/false* // *(4) 如果用新的撤销流进行插入（比如任务停止再重启），第一条肯定为true，那么肯定是插入的（或者更新）* // *比如说：ES中现有数据 sensor\_1 15* // *然后用别的流插入数据：sesor\_1 10 会发现即使求的是max也会更新成10* // *因为撤销流第一条数据肯定为true* |

### 2.7.6 输出到MySql



1.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：

|  |
| --- |
| **import** com.atguigu.bean.SensorReading; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment;  **public class** FlinkSQL\_Test08 {   **public static void** main(String[] args) **throws** Exception {   *//1.获取执行环境并设置并行度* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);   *//2.读取文本数据创建流* DataStreamSource<String> readTextFile = env.readTextFile(**"sensor"**);   *//3.将每一行数据转换为JavaBean* SingleOutputStreamOperator<SensorReading> sensorDataStream = readTextFile.map(line -> {  String[] fields = line.split(**","**);  **return new** SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  });   *//4.创建TableAPI执行环境* StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//5.从流中创建表* Table table = tableEnv.fromDataStream(sensorDataStream);   *//6.转换数据* tableEnv.createTemporaryView(**"sensor"**, table);  Table sqlQuery = tableEnv.sqlQuery(**"select id,count(\*) as ct from sensor group by id"**);   *//7.定义MySQL连接器* String sinkDDL = **"create table jdbcOutputTable ("** +  **" id varchar(20) not null, "** +  **" ct bigint not null "** +  **") with ("** +  **" 'connector.type' = 'jdbc', "** +  **" 'connector.url' = 'jdbc:mysql://hadoop102:3306/test', "** +  **" 'connector.table' = 'sensor\_count', "** +  **" 'connector.driver' = 'com.mysql.jdbc.Driver', "** +  **" 'connector.username' = 'root', "** +  **" 'connector.password' = '000000' )"**;  //**jdbcOutputTable是flinkSQL中的虚拟表格，在Mysql中建立的表是sensor\_count**  //**不会帮我们自动建表，插入前需要先在Mysql中建表，否则报错**  tableEnv.sqlUpdate(sinkDDL);   *//8.将数据写入MySQL* sqlQuery.insertInto(**"jdbcOutputTable"**);   *//9.执行任务* env.execute();   } }  //写入mysql需要设置下面的参数    **指定刷写时机： 默认是5000条刷写，interval默认值为0（不按照时间刷写）**    更新流演示：  //todo 1.给mysql中表建立主键  //和追加流的写法是一样的，flinkSQL会自动根据groupby的字段作为主键，无则插入，有责更新    额外说明：DDL方式写入，在写好DDL语句后，要执行tableEnv.sqlUpdate(sinkDDL) |

## 2.8 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。

通过TableEnvironment.explain（table）方法或TableEnvironment.explain（）方法完成的。

执行计划：先查看执行的计划，用于优化再执行SQL

**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

flink SQL中 Table API和SQL，本质上还是基于关系型表的操作方式；

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。

有count，sum等聚合函数，流处理看的结果都是当下的，需要不断地更新，传统的表查询数据都是死的

所以flink中引入了动态表，表中数据是不断更新的

## 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【这个过程本质上还是状态编程】

3. 生成的动态表被转换回流。【聚合操作只能用撤回流，没有聚合可以用追加流】

**流转表，查询，表转流**

### 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：

[

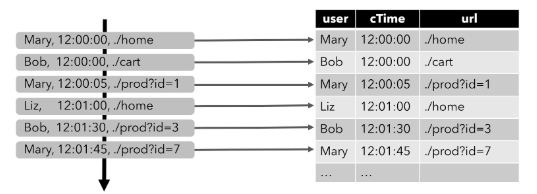
user: VARCHAR, // 用户名

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。

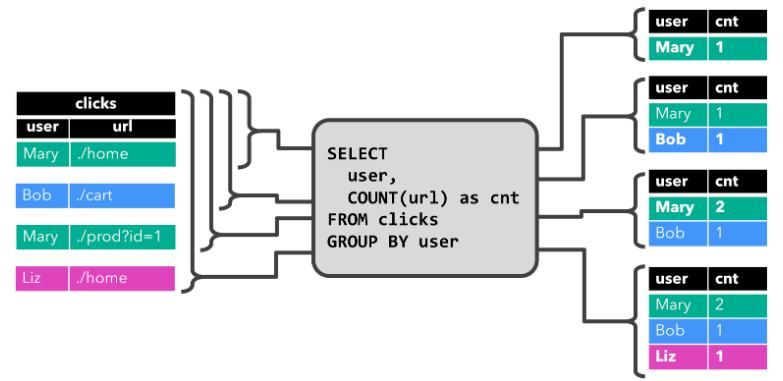
### 3.3.2 持续查询（Continuous Query）

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：**

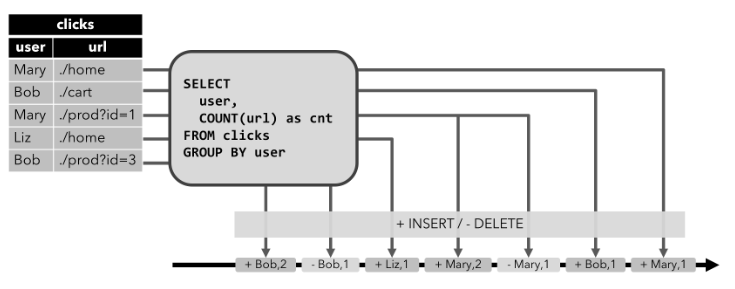
#### 1）仅追加（Append-only）流

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，就是在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。**

下图显示了将动态表转换为Retract流的过程：



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**

**Insert=>add**

**Delete=>retract**

**Update => retract + add**

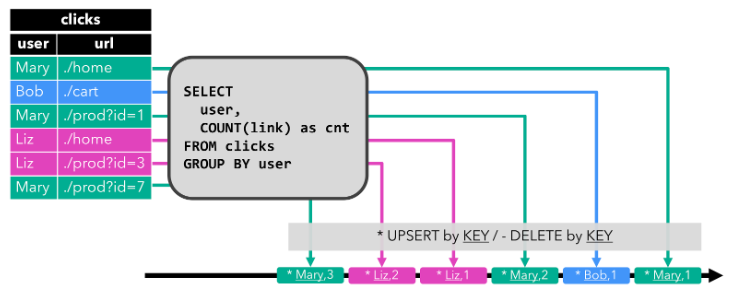
撤回流执行过程：主键没有重复的直接insert,遇到重复的，先delete再insert

#### 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。**

**文件系统和kafka只能用追加流**

#### Upsert流和Restract流的主要区别

（1）update操作，Upsert流效率更高：The main difference to a retract stream is that UPDATE changes are encoded with a single message and hence more efficient.

（2）更新流依赖于外部系统需要有唯一键。比如ES的Connector中可以设置模式，会根据group by字段自动设置docID，而Mysql来说在Mysql中建表的时候就要设置唯一键（或者主键）和flink SQL中group by字段保持一致。

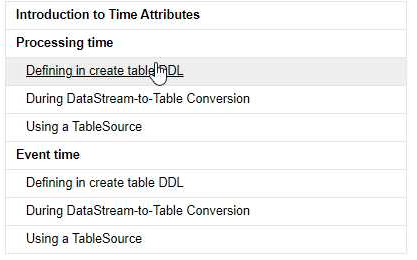
## 3.4 时间特性

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。

如下图所示：



（1）引入时间属性的三种方式：

在用DDL创建表的时候（只在blink中使用）；在流转表的时候（可用）；在用连接器连接外部系统创建表的时候（有问题，不用）；**[其实可以发现，时间属性的引入就是在获取flinkSQL中初始Table的时候]**

（2）时间字段有两种类型：处理时间和事件时间

### 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转化为表时指定（fromDataStream）；**

**在定义Table Schema时指定（连接器中）；**

**在创建表的DDL中指定。**

#### 1）在DataStream转化成Table时指定处理时间

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：

|  |
| --- |
| **import** com.atguigu.bean.SensorReading; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment;  **public class** FlinkSQL\_Test09 {   **public static void** main(String[] args) {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2.读取端口数据创建流并转换为JavaBean* SingleOutputStreamOperator<SensorReading> sensorDataStream = env.socketTextStream(**"hadoop102"**, 7777)  .map(line -> {  String[] fields = line.split(**","**);  **return new** SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  });   *//3.将流转换为Table,并设定处理时间语义* Table table = tableEnv.fromDataStream(sensorDataStream, **"id, ts, temp, pt.proctime"**);   *//4.打印schema信息* table.printSchema();   }  }  //执行结果 TIMESTAMP(3)表示秒后面三位 毫秒 |

#### 2）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：

|  |
| --- |
| **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.DataTypes; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.table.descriptors.FileSystem; **import** org.apache.flink.table.descriptors.OldCsv; **import** org.apache.flink.table.descriptors.Schema;  **public class** FlinkSQL\_Test10 {   **public static void** main(String[] args) **throws** Exception {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2.创建文件连接器* tableEnv.connect(**new** FileSystem().path(**"sensor"**))  .withSchema(**new** Schema()  .field(**"id"**, DataTypes.*STRING*())  .field(**"ts"**, DataTypes.*BIGINT*())  .field(**"temp"**, DataTypes.*DOUBLE*())   **.field("pt", DataTypes.*TIMESTAMP*(3)).proctime())**  .withFormat(**new** OldCsv())  .createTemporaryTable(**"fileInput"**);   *//3.打印schema信息* Table table = tableEnv.from(**"fileInput"**);  table.printSchema();   } }  ***//在flink 1.10中有问题，打印信息没问题***  ***TIMESTAMP*(3):毫秒**  ***Proctime():处理时间语义***  **执行结果：**    打印信息中，TIMESTAMP(3)后面少了 proctime  在flinkSQL中，无法使用时间字段 |

#### 3）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：

**注意：运行这段DDL，必须使用Blink Planner。**

|  |
| --- |
| **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.EnvironmentSettings; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment;  **public class** FlinkSQL\_Test11 {   **public static void** main(String[] args) **throws** Exception {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  EnvironmentSettings blinkStreamSettings = EnvironmentSettings.*newInstance*()  .useBlinkPlanner()  .inStreamingMode()  .build();  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env, blinkStreamSettings);   *//2.创建文件连接器* String sinkDDL = **"create table dataTable ("** +  **" id varchar(20) not null, "** +  **" ts bigint, "** +  **" temp double, "** +  **" pt AS PROCTIME() "** +  **") with ("** +  **" 'connector.type' = 'filesystem', "** +  **" 'connector.path' = 'sensor', "** +  **" 'format.type' = 'csv')"**;  tableEnv.sqlUpdate(sinkDDL);   *//3.打印schema信息* Table table = tableEnv.from(**"dataTable"**);  table.printSchema();   } } |

### 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）。

#### 1）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：

|  |
| --- |
| **import** com.atguigu.bean.SensorReading; **import** org.apache.flink.streaming.api.TimeCharacteristic; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.timestamps.BoundedOutOfOrdernessTimestampExtractor; **import** org.apache.flink.streaming.api.windowing.time.Time; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment;  **public class** FlinkSQL\_Test12 {   **public static void** main(String[] args) {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);   **env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*);**  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2.读取端口数据创建流并转换为JavaBean* SingleOutputStreamOperator<SensorReading> sensorDataStream = env.socketTextStream(**"hadoop102"**, 7777)  .assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor<String>(Time.*seconds*(1)) {  @Override  **public long** extractTimestamp(String element) {  **return** Long.*parseLong*(element.split(**","**)[1]) \* 1000L;  }  })  .map(line -> {  String[] fields = line.split(**","**);  **return new** SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  });   *//3.将流转换为Table,并设定处理时间语义* Table table = tableEnv.fromDataStream(sensorDataStream, **"id, ts, temp, rt.rowtime"**);   *//4.打印schema信息* table.printSchema();   }  }  **说明：rowtime表示事件时间，proctime表示处理时间** |

#### 2）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：

|  |
| --- |
| **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.DataTypes; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.table.descriptors.Csv; **import** org.apache.flink.table.descriptors.FileSystem; **import** org.apache.flink.table.descriptors.Rowtime; **import** org.apache.flink.table.descriptors.Schema;  **public class** FlinkSQL\_Test13 {   **public static void** main(String[] args) **throws** Exception {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  **env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);**  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2.创建文件连接器* tableEnv.connect(**new** FileSystem().path(**"sensor"**))  .withSchema(**new** Schema()  .field(**"id"**, DataTypes.*STRING*())  .field(**"ts"**, DataTypes.*BIGINT*()).rowtime(**new** Rowtime()  .timestampsFromField(**"ts"**) *// 从字段中提取时间戳* .watermarksPeriodicBounded(1000) *// watermark延迟1秒* )  .field(**"temp"**, DataTypes.*DOUBLE*())  )  .withFormat(**new** Csv())  .createTemporaryTable(**"fileInput"**);   *//3.打印schema信息* Table table = tableEnv.from(**"fileInput"**);  table.printSchema();   }  }  **//这种方式也会报错，暂时用不了**    //缺点：无法将时间转为毫秒，只能读到什么时间戳就是什么时间戳 |

#### 3）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：

|  |
| --- |
| **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.EnvironmentSettings; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment;  **public class** FlinkSQL\_Test14 {   **public static void** main(String[] args) **throws** Exception {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);  EnvironmentSettings blinkStreamSettings = EnvironmentSettings.*newInstance*()  .useBlinkPlanner()  .inStreamingMode()  .build();  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env, blinkStreamSettings);   *//2.创建文件连接器* String sinkDDL = **"create table dataTable ("** +  **" id varchar(20) not null, "** +  **" ts bigint, "** +  **" temp double, "** +  **" rt AS TO\_TIMESTAMP( FROM\_UNIXTIME(ts) ), "** +  **" watermark for rt as rt - interval '1' second"** +  **") with ("** +  **" 'connector.type' = 'filesystem', "** +  **" 'connector.path' = 'sensor', "** +  **" 'format.type' = 'csv')"**;  tableEnv.sqlUpdate(sinkDDL);   *//3.打印schema信息* Table table = tableEnv.from(**"dataTable"**);  table.printSchema();   } }    **FROM\_UNIXTIME(ts)：将字段转为时间**  **TO\_TIMESTAMP（）：时间转为时间戳，变为毫秒** |

**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。**

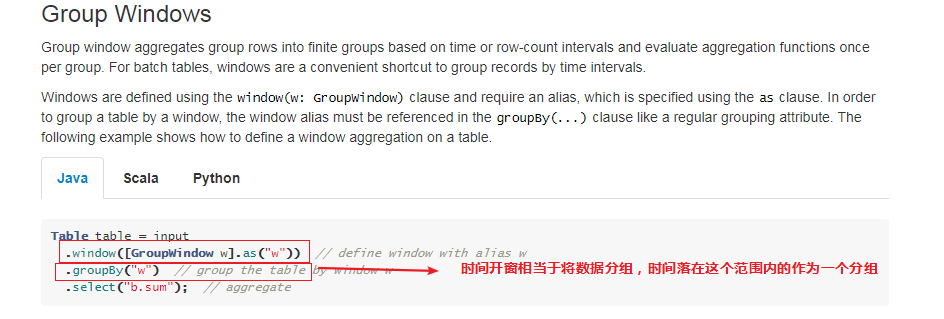
# 第四章 窗口（Windows）

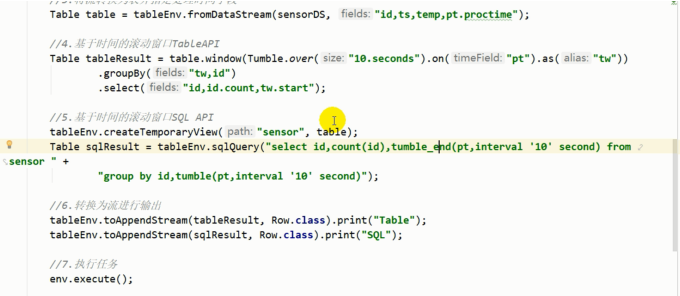
时间语义，要配合窗口操作才能发挥作用。最主要的用途，当然就是开窗口、根据时间段做计算了。下面我们就来看看Table API和SQL中，怎么利用时间字段做窗口操作。

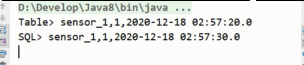
**在Table API和SQL中，主要有两种窗口：Group Windows和Over Windows**

## Group Windows

<https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/table/tableApi.html#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TableAPI方式

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 事件时间滑动窗口*.window(Tumble.*over*(**"10.minutes"**).on(**"rowtime"**).as(**"w"**))  
*// Tumbling Processing-time Window 时间滑动窗口*.window(Tumble.*over*(**"10.minutes"**).on(**"proctime"**).as(**"w"**))  
*// Tumbling Row-count Window 计数窗口*.window(Tumble.*over*(**"10.rows"**).on(**"proctime"**).as(**"w"**))

### 4.1.2 滑动窗口TableAPI方式

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TableAPI方式

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**"))**

### 4.1.4 FlinkSQL方式

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。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)

## 4.2 Group Windows案例实操

### 4.2.1 滚动窗口（时间）

|  |
| --- |
| **import** com.atguigu.bean.SensorReading; **import** org.apache.flink.streaming.api.TimeCharacteristic; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.Tumble; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.types.Row;  **public class** FlinkSQL\_Test15 {   **public static void** main(String[] args) **throws** Exception {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2.读取端口数据创建流并转换为JavaBean* SingleOutputStreamOperator<SensorReading> sensorDataStream = env.socketTextStream(**"hadoop102"**, 7777)  .map(line -> {  String[] fields = line.split(**","**);  **return new** SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  });   *//3.将流转换为Table* Table table = tableEnv.fromDataStream(sensorDataStream, **"id, ts, temp, pt.proctime"**);   *//4.TableAPI实现滚动时间窗口* Table tableResult = table  .window(Tumble.*over*(**"10.seconds"**).on(**"pt"**).as(**"tw"**))  .groupBy(**"tw,id"**)  .select(**"id, id.count as ct, temp.avg as avgTemp, tw.end"**);   *//5.SQL实现滚动时间窗口* tableEnv.createTemporaryView(**"sensor"**,table);  Table sqlResult = tableEnv.sqlQuery(**"select id ,count(id) ct, avg(temp),tumble\_end(pt,interval '10' second) "** +  **" from sensor group by id,tumble(pt,interval '10' second)"**);   *//6.打印数据* tableEnv.toAppendStream(tableResult, Row.**class**).print(**"tableResult"**);  tableEnv.toAppendStream(sqlResult, Row.**class**).print(**"sqlResult"**);   *//7.开启任务* env.execute();   }  } |

### 4.2.2 滑动窗口（时间）

|  |
| --- |
| **import** com.atguigu.bean.SensorReading; **import** org.apache.flink.streaming.api.TimeCharacteristic; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.timestamps.BoundedOutOfOrdernessTimestampExtractor; **import** org.apache.flink.streaming.api.windowing.time.Time; **import** org.apache.flink.table.api.Slide; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.types.Row;  **public class** FlinkSQL\_Test16 {   **public static void** main(String[] args) **throws** Exception {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  env.setStreamTimeCharacteristic(TimeCharacteristic.***EventTime***);  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2.读取端口数据创建流并转换为JavaBean* SingleOutputStreamOperator<SensorReading> sensorDataStream = env.socketTextStream(**"hadoop102"**, 7777)  .assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor<String>(Time.*seconds*(1)) {  @Override  **public long** extractTimestamp(String element) {  **return** Long.*parseLong*(element.split(**","**)[1]) \* 1000L;  }  })  .map(line -> {  String[] fields = line.split(**","**);  **return new** SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  });   *//3.将流转换为Table,并设定处理时间语义* Table table = tableEnv.fromDataStream(sensorDataStream, **"id, ts, temp, rt.rowtime"**);   *//4.TableAPI实现滑动时间窗口* Table tableResult = table  .window(Slide.*over*(**"15.second"**).every(**"5.second"**).on(**"rt"**).as(**"tw"**))  .groupBy(**"tw,id"**)  .select(**"id, id.count as ct, temp.avg as avgTemp, tw.end"**);   *//5.SQL实现滑动时间窗口* tableEnv.createTemporaryView(**"sensor"**,table);  Table sqlResult = tableEnv.sqlQuery(**"select id ,count(id) ct, avg(temp) "** +  **" from sensor group by id,hop(rt,interval '5' second,interval '15' second)"**);   *//6.打印数据* tableEnv.toAppendStream(tableResult, Row.**class**).print(**"tableResult"**);  tableEnv.toAppendStream(sqlResult, Row.**class**).print(**"sqlResult"**);   *//7.开启任务* env.execute();   }  } |

### 4.2.3 会话窗口（时间）

|  |
| --- |
| **import** com.atguigu.bean.SensorReading; **import** org.apache.flink.streaming.api.TimeCharacteristic; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.timestamps.BoundedOutOfOrdernessTimestampExtractor; **import** org.apache.flink.streaming.api.windowing.time.Time; **import** org.apache.flink.table.api.Session; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.Tumble; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.types.Row;  **public class** FlinkSQL\_Test17 {   **public static void** main(String[] args) **throws** Exception {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  env.setStreamTimeCharacteristic(TimeCharacteristic.***EventTime***);  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2.读取端口数据创建流并转换为JavaBean* SingleOutputStreamOperator<SensorReading> sensorDataStream = env.socketTextStream(**"hadoop102"**, 7777)  .assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor<String>(Time.*seconds*(1)) {  @Override  **public long** extractTimestamp(String element) {  **return** Long.*parseLong*(element.split(**","**)[1]) \* 1000L;  }  })  .map(line -> {  String[] fields = line.split(**","**);  **return new** SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  });   *//3.将流转换为Table,并设定处理时间语义* Table table = tableEnv.fromDataStream(sensorDataStream, **"id, ts, temp, rt.rowtime"**);   *//4.TableAPI实现会话窗口* Table tableResult = table.window(Session.*withGap*(**"5.seconds"**).on(**"rt"**).as(**"tw"**))  .groupBy(**"tw,id"**)  .select(**"id, id.count as ct, temp.avg as avgTemp, tw.end"**);   *//5.SQL实现会话窗口* tableEnv.createTemporaryView(**"sensor"**,table);  Table sqlResult = tableEnv.sqlQuery(**"select id ,count(id) ct, avg(temp) "** +  **" from sensor group by id,session(rt,interval '5' second)"**);   *//6.打印数据* tableEnv.toAppendStream(tableResult, Row.**class**).print(**"tableResult"**);  tableEnv.toAppendStream(sqlResult, Row.**class**).print(**"sqlResult"**);   *//7.开启任务* env.execute();   }  } |

### 4.2.4 滚动窗口（计数）

计数窗口只有滚动和滑动，而且只能基于处理时间。

计数窗口不能用SQL方式，只能用tableAPI

|  |
| --- |
| **import** com.atguigu.bean.SensorReading; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.Tumble; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.types.Row;  **public class** FlinkSQL\_Test18 {   **public static void** main(String[] args) **throws** Exception {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2.读取端口数据创建流并转换为JavaBean* SingleOutputStreamOperator<SensorReading> sensorDataStream = env.socketTextStream(**"hadoop102"**, 7777)  .map(line -> {  String[] fields = line.split(**","**);  **return new** SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  });   *//3.将流转换为Table,并设定处理时间语义* Table table = tableEnv.fromDataStream(sensorDataStream, **"id, ts, temp, pt.proctime"**);   *//4.TableAPI实现滚动计数窗口* Table tableResult = table.window(Tumble.*over*(**"5.rows"**).on(**"pt"**).as(**"tw"**))  .groupBy(**"tw,id"**)  .select(**"id, id.count as ct, temp.avg as avgTemp"**);   *//5.打印数据* tableEnv.toAppendStream(tableResult, Row.**class**).print(**"tableResult"**);   *//6.开启任务* env.execute();   }  } |

### 4.2.5 滑动窗口（计数）

|  |
| --- |
| **import** com.atguigu.bean.SensorReading; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.table.api.Slide; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.Tumble; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.types.Row;  **public class** FlinkSQL\_Test19 {   **public static void** main(String[] args) **throws** Exception {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2.读取端口数据创建流并转换为JavaBean* SingleOutputStreamOperator<SensorReading> sensorDataStream = env.socketTextStream(**"hadoop102"**, 7777)  .map(line -> {  String[] fields = line.split(**","**);  **return new** SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  });   *//3.将流转换为Table,并设定处理时间语义* Table table = tableEnv.fromDataStream(sensorDataStream, **"id, ts, temp, pt.proctime"**);   *//4.TableAPI实现滑动计数窗口* Table tableResult = table.window(Slide.*over*(**"5.rows"**).every(**"2.rows"**).on(**"pt"**).as(**"tw"**))  .groupBy(**"tw,id"**)  .select(**"id, id.count as ct, temp.avg as avgTemp"**);   *//5.打印数据* tableEnv.toAppendStream(tableResult, Row.**class**).print(**"tableResult"**);   *//6.开启任务* env.execute();   }  } |

## Over Windows

### 4.3.2 FlinkSQL方式

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）。

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**)

### 4.3.1 TableAPI方式

Over window聚合是和标准SQL中已有的（Over子句）是一样的，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。

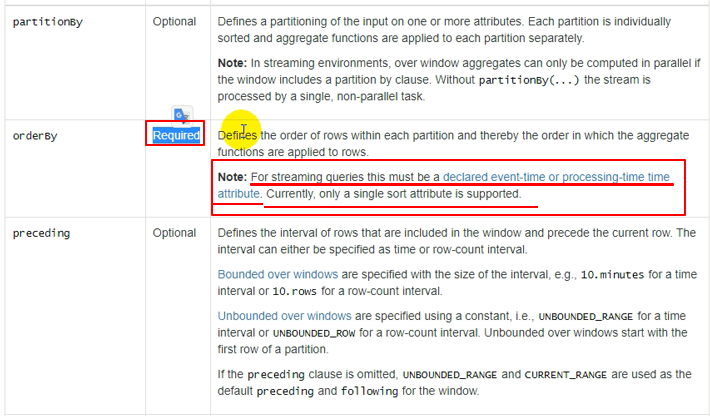
**（1）Over窗口的定义：**

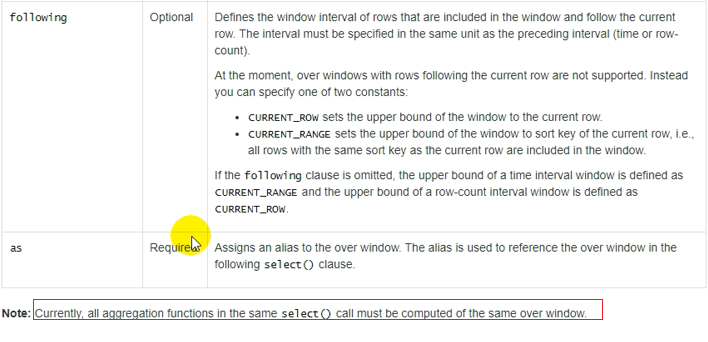
Over windows使用.window（w:overwindows\*）子句定义，并在select（）方法中通过别名来引用。

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。

比如这样：

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





**注意：**over窗口的定义中，orderBy函数是必须要写的，而且只能有单一字段，并且该字段必须是处理时间或者是事件时间。（不仅仅是TABLEAPI，SQL也是如此）

As起别名也是必须写的

**over window中，如果preceding中指定为UNBOUNDED\_RANGE表示是无界时间窗口，如果是UNBOUNDED\_ROW表示是无界计数窗口。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

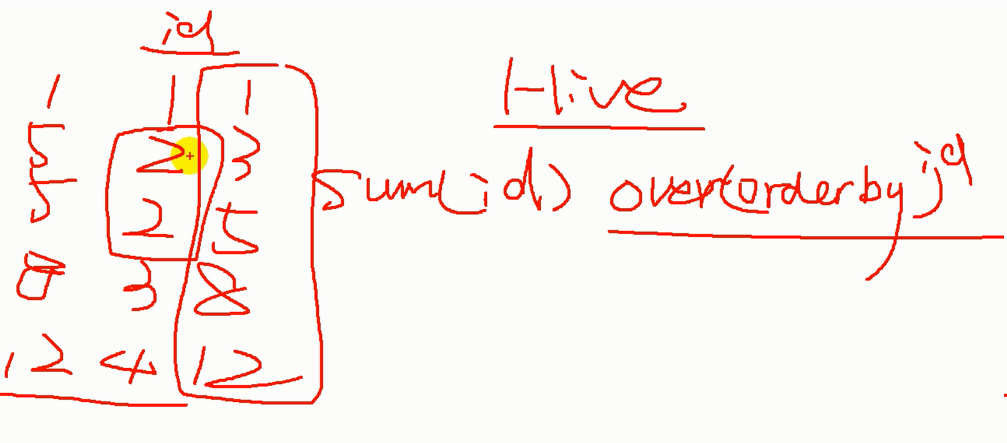
*// 有界的事件时间 over window*.window(Over.*partitionBy*(**"a"**).orderBy(**"rowtime"**).preceding(**"1.minutes"**).as(**"w"**))  
*// 有界的处理时间 over window*.window(Over.*partitionBy*(**"a"**).orderBy(**"proctime"**).preceding(**"1.minutes"**).as(**"w"**))  
*// 有界的事件时间 Row-count over window*.window(Over.*partitionBy*(**"a"**).orderBy(**"rowtime"**).preceding(**"10.rows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

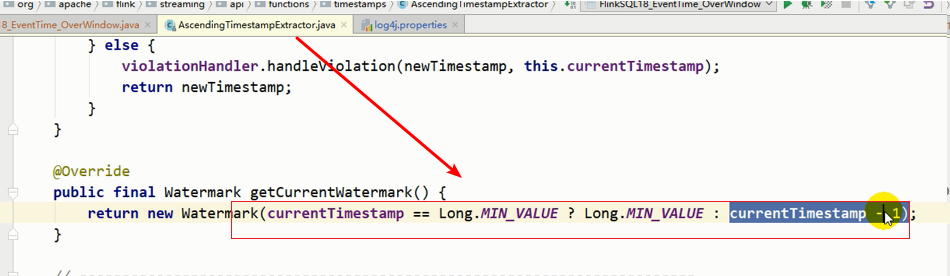
## 4.4 Over Windows案例实操

|  |
| --- |
| **import** com.atguigu.bean.SensorReading; **import** org.apache.flink.streaming.api.TimeCharacteristic; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.timestamps.BoundedOutOfOrdernessTimestampExtractor; **import** org.apache.flink.streaming.api.windowing.time.Time; **import** org.apache.flink.table.api.Over; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.java.StreamTableEnvironment; **import** org.apache.flink.types.Row;  **public class** FlinkSQL\_Test28 {   **public static void** main(String[] args) **throws** Exception {   *//1.创建执行环境* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  env.setStreamTimeCharacteristic(TimeCharacteristic.***EventTime***);  StreamTableEnvironment tableEnv = StreamTableEnvironment.*create*(env);   *//2.读取端口数据创建流并转换为JavaBean* SingleOutputStreamOperator<SensorReading> sensorDataStream = env.socketTextStream(**"hadoop102"**, 7777)  .assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor<String>(Time.*seconds*(1)) {  @Override  **public long** extractTimestamp(String element) {  **return** Long.*parseLong*(element.split(**","**)[1]) \* 1000L;  }  })  .map(line -> {  String[] fields = line.split(**","**);  **return new** SensorReading(fields[0], Long.*parseLong*(fields[1]), Double.*parseDouble*(fields[2]));  });   *//3.将流转换为Table,并设定处理时间语义* Table table = tableEnv.fromDataStream(sensorDataStream, **"id, ts, temp, rt.rowtime"**);   *//4.TableAPI实现over窗口* Table tableResult = table.window(Over.*partitionBy*(**"id"**).orderBy(**"rt"**).preceding(**"1.rows"**).as(**"w"**))  .select(**"id, id.count over w as ct, temp.avg over w as avgTemp"**);   *//5.SQL* tableEnv.createTemporaryView(**"sensor"**,table);  Table sqlResult = tableEnv.sqlQuery(**"select id ,count(id) over w as ct, avg(temp) over w"** +  **" from sensor window w as (partition by id order by rt rows between 1 preceding and current row)"**);   *//6.打印数据* tableEnv.toRetractStream(tableResult, Row.**class**).print(**"tableResult"**);  tableEnv.toRetractStream(sqlResult, Row.**class**).print(**"sqlResult"**);   *//7.开启任务* env.execute();   }  } |

**Sum(id) over(order by id ) 对于 id 1 2 2 3 4**

**输出为 1 5 5 8 12**

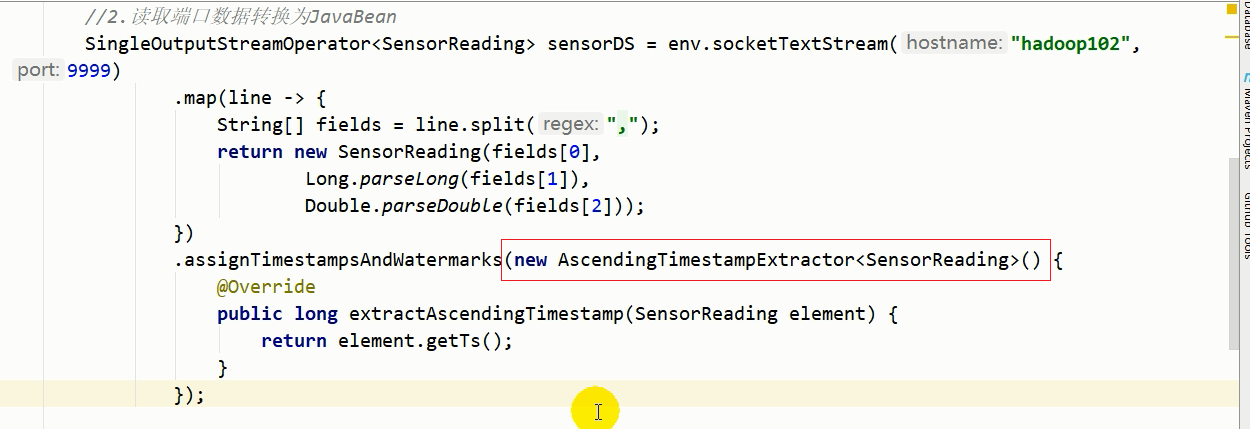




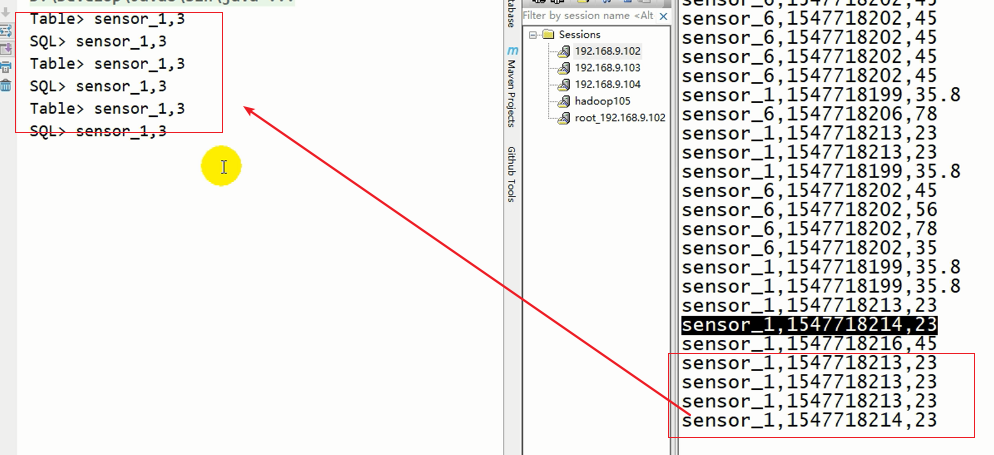


Id相同的时候，按照事件时间rt进行排序，那么over()窗口就是从id中rt最小的到达当前位置，假设id=1的数据来个rt是19，那么开窗是自己，如果再来一条18，求count应该在18是1,19是2。 为了防止这种情况发生，直接将19的waterMark往前推移2s，这样19进来了不会执行计算，18能够进入19的窗口，只有等到21进来了，19才会执行计算。

对于AccendingTimestampExtractor来说，waterMark为0中获取waterMark为现有时间戳-1ms,目的是防止后面有一样的数据，结果提前关了窗，就不能算计来了。





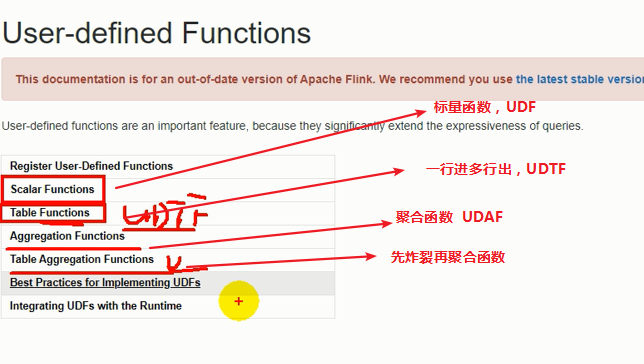


这就说明over窗口中相同时间戳，不会立即计算，而是等到将窗口关闭了才执行，防止后来的时间戳小于当前时间戳，会导致计算结果不对。

看到事件事件首先想到watermark

# 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。



UDF:1行到1行

UDTF:1行到多行

UDAF:多行到1行

炸裂聚合：输入多行输出多行；

**只限定行，不限定列的个数**

## 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"**;  
  
 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：

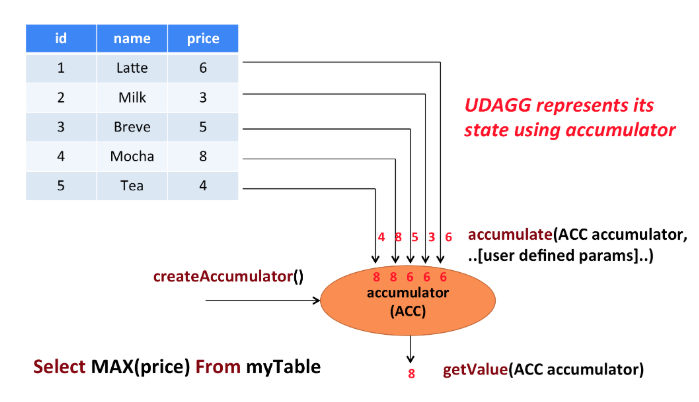
Split split = **new** Split(**"\_"**);  
tableEnv.registerFunction(**"split"**, split);  
Table resultTable = sensorTable  
 .joinLateral( **"split(id) as (word, length)"**)  
 .select(**"id, ts, word, length"**);

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;  
 }  
}

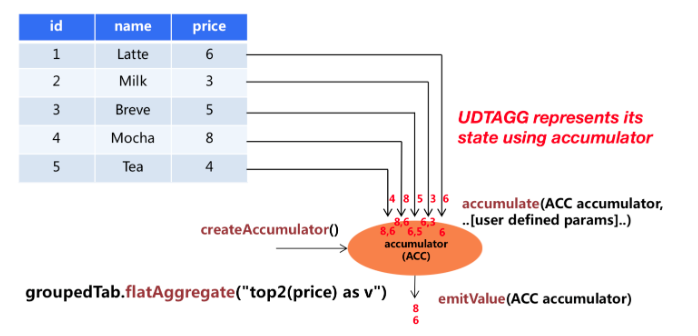
接下来就可以在代码中调用了。

*// 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"**);