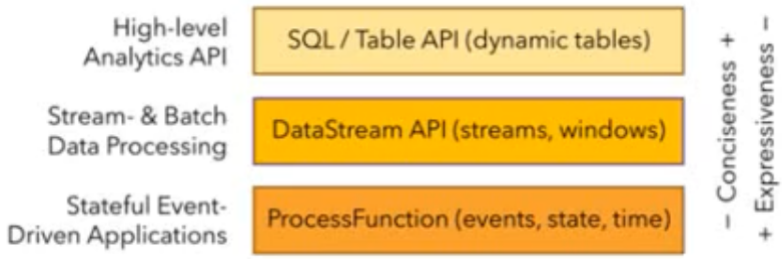
Table API 和 Flink SQL

一、简介

Table API和Flink SQL是什么?

- Flink对批处理和流处理，提供了统一的上层API
- Table API是一套内嵌在Java和Scala语言中的查询API，它允许以非常直观的方式组合来自一些关系运算符的查询
- Flink的SQL支持基于实现了SQL标准的Apache Calcite



需要引入的pom依赖

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table-planner_2.11</artifactId>
    <version>1.10.0</version>
</dependency>

<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-table-api-scala-bridge_2.11</artifactId>
    <version>1.10.0</version>
</dependency>
```

二、表环境定义

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十分类似

```
val 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
val result = tableEnv.from("inputTable").select(...)
// 通过SQL查询语句，得到一张结果表
```

```
val sqlResult = tableEnv.sqlQuery("SELECT ... FROM inputTable ...")

// 将结果表写入输出表中
result.insertInto("outputTable")
```

创建TableEnvironment

创建表的执行环境，需要将flink流处理的执行环境传入

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

TableEnvironment是flink中集成Table API和SQL的核心概念，所有对表的操作都基于TableEnvironment

- 注册Catalog
- 在Catalog中注册表
- 执行SQL查询
- 注册用户自定义函数（UDF）

三、读取文件创建表

表（Table）

TableEnvironment可以注册目录Catalog，并可以基于Catalog注册表

表（Table）是一个“标识符”（identifier）来指定的，由3部分组成：Catalog名、数据库（database）名和对象名

表可以是常规的，也可以是虚拟的（视图，view）

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（View）可以从现有的表中创建，通常是table API或者SQL查询的一个结果集

创建表

TableEnvironment可以调用.connect()方法，连接外部系统，并调用.createTemporaryTable()方法，在Catalog中注册表

```
tableEnv
    .connect(...)    // 定义表的数据来源，和外部系统建立连接
    .withFormat(...)     // 定义数据格式化方法
    .withSchema(...)     // 定义表结构
    .createTemporaryTable("MyTable")     // 创建临时表
```

四、读取kafka数据创建表

```
tableEnv.connect(new Kafka()
                 .version("0.11")
                 .topic("sensor")
                 .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")
```

五、表的查询转换

表的查询 - Table API

Table API是集成在Scala和Java语言内的查询API

Table API基于代表"表"类的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，表示对输入表应用转换操作的结果

有些关系型转换操作，可以有多个方法调用组成，构成链式调用结构

```
val sensorTable: Table = tableEnv.from("intputTable")
val resultTable: Table = sensorTable
    .select("id, temporary")    // .select('id, 'temporary)
    .filter("id = 'sensor_1'")  // .filter('id === "sensor_1")

val aggResultTable = sensorTable
    .groupBy('id)
    .select('id, 'id.count as 'count)
```

六、表和流相互转换

将DataStream转换成表

对于一个DataStream，可以直接转换成Table，进而方便地调用Table API做转换操作

```
val dataStream: DataStream[SensorReading] = ...
val sensorTable: Table = tableEnv.fromDataStream(dataStream)
```

默认转换出来的Table Schema和DataStream中的字段定义一一对应，也可以单独指定出来

```
val dataStream: DataStream[SensorReading] = ...
val sensorTable = tableEnv.fromDataStream(dataStream, 'id, 'timestamp,
'temporature)
```

数据类型与Schema的对应

DataStream中的数据类型，与表的Schema之间的对应关系，可以有两种：基于字段名称，或者基于字段位置

- 基于名称（name-based）

```
val sensorTable = tableEnv.fromDataStream(dataStream, 'timestamp as 'ts, 'id
as         'myId, 'temporature)
```

- 基于位置（position-based）

```
val sensorTable = tableEnv.fromDataStream(dataStream, 'myId, 'ts)
```

创建临时视图（Temporary View）

- 基于DataStream创建临时视图

```
tableEnv.createTemporaryView("sensorView", dataStream)
tableEnv.createTemporaryView("sensorView", dataStream, 'id, 'temporature,
'timestamp as 'ts)
```

- 基于Table创建临时视图

```
tableEnv.createTemporaryView("sensorView", sensorTable)
```

七、输出到文件

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

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

```
tableEnv.connect(...)
    .createTemporaryTable("outputTable")
val resultTable: Table = ...
resultTable.insertInto("outputTable")
```

八、更新模式

对于流式查询，需要声明如何在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

- 追加（Append）模式

  表只做插入操作，和外部连接器只交换插入（Insert）消息

- 撤回（Retract）模式

  表和外部连接器交换添加（Add）和撤回（Retract）消息

  插入操作（Insert）编码为Add消息；删除（Delete）编码为Retract消息；更新（Update）编码为上一条的Retract和下一条的Add消息

- 更新插入（Upsert）模式

  更新和插入都被编码为Upsert消息；删除编码为Delete消息

九、数据管道测试

输出到Kafka

可以创建Table来描述kafka中的数据，作为输入或输出的TableSink

```
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")
```

十、写入数据到其他外部系统

输出到ES

可以创建Table来描述ES中的数据，作为输出的TableSink

```
tableEnv.connect(
    new Elasticsearch()
        .version("6")
        .host("localhost", 9200, "http")
        .index("sensor")
        .documentType("temp")
)
    .inUpsertMode()
    .withFormat(new Json())
    .withSchema(new Schema()
        .field("id", DataTypes.STRING())
        .field("count", DataTypes.BIGINT())
    )
    .createTemporaryTable("esOutputTable")
```

```
aggResultTable.insertInto("esOutputTable")
```

输出到MySql

可以创建Table来描述MySql中的数据，作为输入和输出

```
val sinkDDL: String =
    """
        |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'
        |)
    """.stripMargin

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

十一、时间语义和时间属性的定义

时间特性（Time Attributes）

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。

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。

定义处理时间（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。

由DataStream转换成表时指定。

在定义Schema期间，可以使用.proctime，指定字段名定义处理时间字段。

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。

```
val sensorTable = tableEnv.fromDataStream(dataStream,
    'id, 'temperature, 'timestamp, 'pt.proctime)
```

定义Table Schema时指定

```
.withSchema(new Schema()
    .field("id", DataTypes.STRING())
    .field("timestamp", DataTypes.BIGINT())
    .field("temperature", DataTypes.DOUBLE())
    .field("pt", DataTypes.TIMESTAMP(3))
    .proctime()
)
```

在创建表的DDL中定义

```
val sinkDDL: String =
    """
        |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'
        |)
    """.stripMargin
tableEnv.sqlUpdate(sinkDDL)
```

定义事件时间（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时间时，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。

定义事件时间，同样有三种方法：

- 由DataStream转换成表时指定
- 定义Table Schema时指定
- 在创建表的DDL中定义

由DataStream转换成表时指定：

在DataStream转换成Table，使用rowtime可以定义事件时间属性

```
// 将DataStream转换成Table，并指定时间字段
val sensorTable = tableEnv.fromDataStream(dataStream,
    'id, 'temperature, 'timestamp.rowtime as 'rt)

// 或者，直接追加时间字段
val sensorTable = tableEnv.fromDataStream(dataStream,
    'id, 'temperature, 'timestamp, 'rt.rowtime)
```

定义Table Schema时指定

```
.withSchema(new Schema()
    .field("id", DataTypes.STRING())
    .field("timestamp", DataTypes.BIGINT())
    .rowtime(
        new Rowtime()
            .timestampFromField("timestamp")    // 从字段中提取时间戳
            .watermarksPeriodicBounded(1000)    // watermark延迟1秒
    )
    .field("temperature", DataTypes.DOUBLE())
)
```

在创建表的DDL中定义

```
val sinkDDL: String =
    """
    |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'
    |)
    """.stripMargin
tableEnv.sqlUpdate(sinkDDL)
```

十二、Group Windows

窗口

时间语义，要配合窗口操作才能发挥作用。

在Table API和SQL中，主要有两种窗口：

- Group Windows（分组窗口）

  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。

- Over Windows

  针对每个输入行，计算相邻行范围内的聚合

Group Windows

Group Windows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

```
val table = input
    .window([w:GroupWindow] as 'w)  // 定义窗口，别名为w
    .groupBy('w, 'a)    // 按照字段a和窗口w分组
    .select('a, 'b.sum) // 聚合
```

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。

滚动窗口（Tumbling windows）

滚动窗口要用Tumble类来定义

```
// 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)
```

滑动窗口（Sliding windows）

滑动窗口要用Slide类来定义

```
// 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)
```

会话窗口（Session windows）

```
// 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)
```

十三、Over Windows

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

```
val 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。

无界Over Windows

可以在事件时间或处理时间，以及指定为时间间隔、或行技术的范围内，定义Over windows。

无界的over window是使用常量指定的。

```
// 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)
```

有界Over Windows

有界的over window是用间隔的大小指定的。

```
//
.window(Over partitionBy 'a orderBy 'rowtime preceding 1.minutes as 'w)

//
.window(Over partitionBy 'a orderBy 'proctime preceding 1.minutes as 'w)

//
.window(Over partitionBy 'a orderBy 'rowtime preceding 10.rows as 'w)

//
.window(Over partitionBy 'a orderBy 'proctime preceding 10.rows as 'w)
```

十四、Flink SQL中的窗口实现

SQL中的Group Windows

Group Windows定义在SQL查询的Group By子句中。

- 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。

SQL中的Over Windows

用Over做窗口聚合时，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。

ORDER BY必须在单一的时间属性上指定。

```
SELECT COUNT(amount) OVER (
    PARTITION BY user
    ORDER BY proctime
    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
FROM Orders
```

十五、系统内置函数

函数（Functions）

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。

- 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

- 算数函数

  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(expression)
    - RANK()
    - ROW_NUMBER()

  Table API:

    - FIELD.count
    - FIELD.sum()

十六、自定义标量函数

用户自定义函数（UDF）

用户定义函数（User-defined Functions，UDF）是一个重要的特性，它们显著的扩展了查询的表达能力。

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。

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和SQL解析器就可以识别并正确的解释它。

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ScalarFunction，并实现（一个或多个）求值（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。

```
class HashCode(factor: Int) extends ScalarFunction {
    def eval(s: String): Int = {
        s.hashCode * factor
    }
}
```

十七、自定义表函数

表函数（Table Functions）

用户定义的表函数，也可以将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。

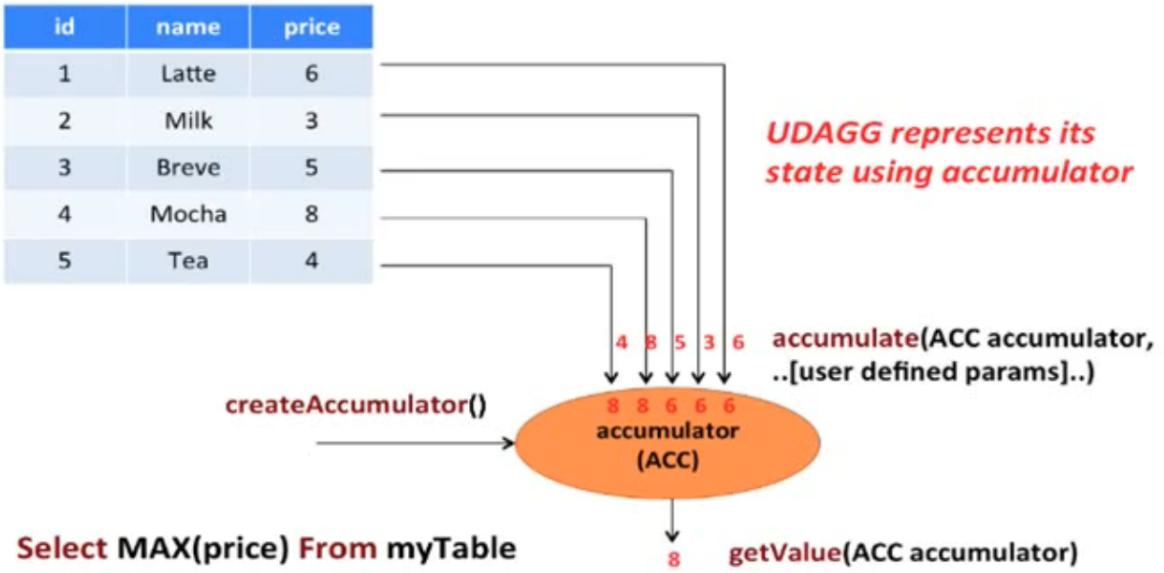
表函数的行为由其求值方法决定，求值方法必须是public的，并命名为eval。

```
class Split(seperator: String) extends TableFunction[(String, Int)] {
    def eval(str: String): Unit = {
        str.split(seperator).foreach(
        word => collect((word, word.length)))
    }
}
```

十八、自定义聚合函数

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。



聚合函数（Aggregate Functions）

AggregateFunction要求必须实现的方法：

- createAccumulator()
- accumulate()
- getValue()

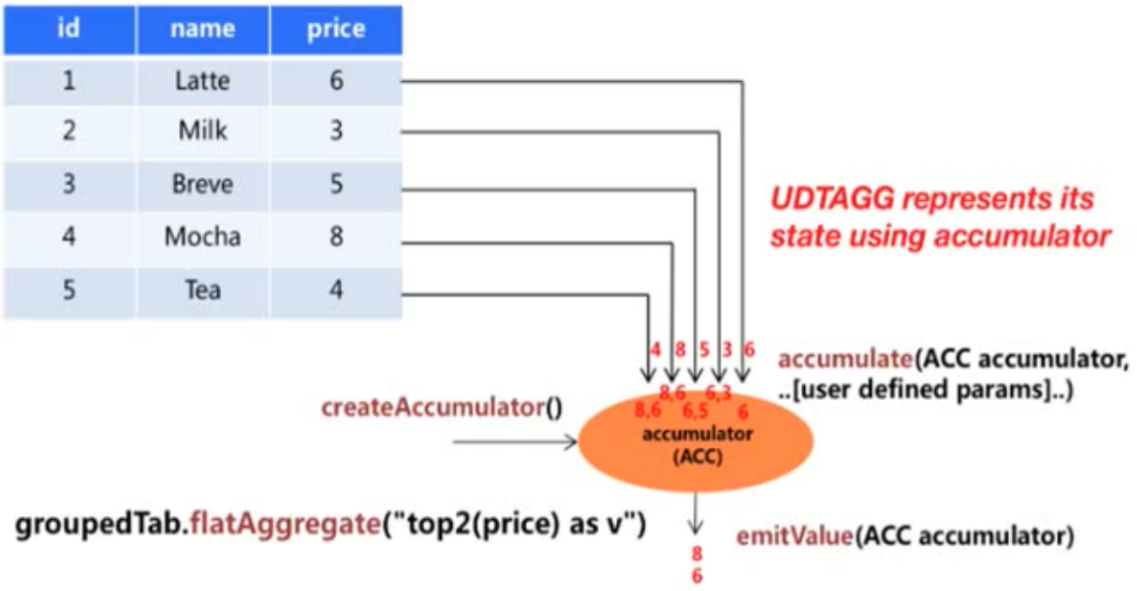
AggregateFunction的工作原理如下：

- 首先，它需要一个累加器（Accumulator），用来保存聚合中间结果的数据结构，可以通过调用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。

十九、自定义表聚合函数

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。

用户定义表聚合函数，是继承了TableAggregateFunction抽象类来实现的。



表聚合函数（Table Aggregate Functions）

AggregationFunction要求必须实现的方法：

- createAccumulator()
- accumulate()
- emitValue()

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。