> 行百里者，半九十

FlinkSql之函数

> 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。

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
INTERVAstring range

Table API:
STRING.toDate
STRING.toTimestamp
currentTime()
NUMERIC.days
NUMERIC.minutes
```

聚合函数

```
SQL:
COUNT(*)
SUM([ AL| DISTINCT ] expression)
RANK()
ROW_NUMBER()

Table API:
FIELD.count
FIELD.sum0
```

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。

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。

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

主函数中调用，计算sensor id的哈希值

```
def main(args: Array[String]): Unit = {
  vaenv = StreamExecutionEnvironment.getExecutionEnvironment
  env.setParallelism(1)
  env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

  vasettings = EnvironmentSettings
    .newInstance()
    .useOldPlanner()
    .inStreamingMode()
    .build()
  vatableEnv = StreamTableEnvironment.create( env, settings )

  // 定义好 DataStream
  vainputStream: DataStream[String] = env.readTextFile("..\\sensor.txt")
  vadataStream: DataStream[SensorReading] = inputStream
    .map(data => {
      vadataArray = data.split(",")
      SensorReading(dataArray(0), dataArray(1).toLong, dataArray(2).toDouble)
    })
    .assignAscendingTimestamps(_.timestamp * 1000L)

  // 将 DataStream转换为 Table，并指定时间字段
  vasensorTable = tableEnv.fromDataStream(dataStream, 'id, 'timestamp.rowtime,
'temperature)

  // Table API中使用
  vahashCode = new HashCode(10)

  varesultTable = sensorTable
    .select( 'id, hashCode('id) )

  // SQ中使用
  tableEnv.createTemporaryView("sensor", sensorTable)
  tableEnv.registerFunction("hashCode", hashCode)
  varesultSqlTable = tableEnv.sqlQuery("select id, hashCode(id) from sensor")
```

```
  // 转换成流，打印输出
  resultTable.toAppendStream[Row].print("table")
  resultSqlTable.toAppendStream[Row].print("sql")

  env.execute()
}
```

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

接下来，就是在代码中调用。首先是Table API的方式：

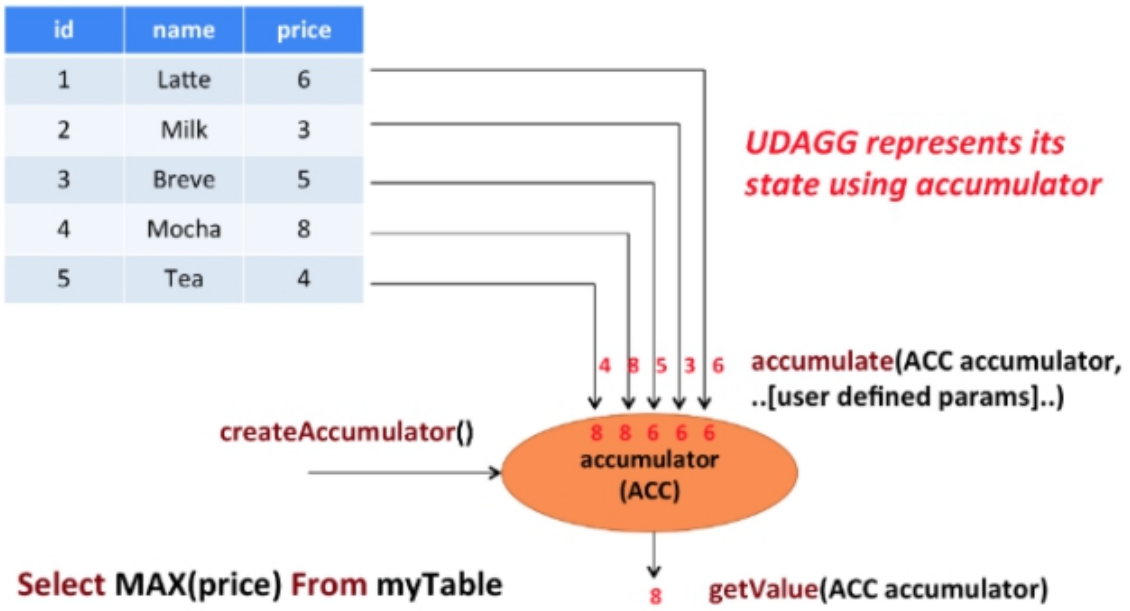
```
// Table API中调用，需要用joinLateral
    varesultTable = sensorTable
      .joinLateral(split('id) as ('word, 'length))   // as对输出行的字段重命名
      .select('id, 'word, 'length)

// 或者用leftOuterJoinLateral
    varesultTable2 = sensorTable
      .leftOuterJoinLateral(split('id) as ('word, 'length))
      .select('id, 'word, 'length)

    // 转换成流打印输出
    resultTable.toAppendStream[Row].print("1")
    resultTable2.toAppendStream[Row].print("2")
```

SQL方式

```
  tableEnv.createTemporaryView("sensor", sensorTable)
    tableEnv.registerFunction("split", split)

    varesultSqlTable = tableEnv.sqlQuery(
      """
        |select id, word, length
        |from
        |sensor, LATERATABLE(split(id)) AS newsensor(word, length)
      """.stripMargin)

    // 或者用左连接的方式
    varesultSqlTable2 = tableEnv.sqlQuery(
      """
        |SELECT id, word, length
        |FROM
        |sensor
|  LEFT JOIN
|  LATERATABLE(split(id)) AS newsensor(word, length)
|  ON TRUE
      """.stripMargin
    )

    // 转换成流打印输出
    resultSqlTable.toAppendStream[Row].print("1")
    resultSqlTable2.toAppendStream[Row].print("2")
```

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。

```
1.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。
2.随后，对每个输入行调用函数的accumulate（）方法来更新累加器。
3.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
class AvgTempAcc {
  var sum: Double = 0.0
  var count: Int = 0
}
```

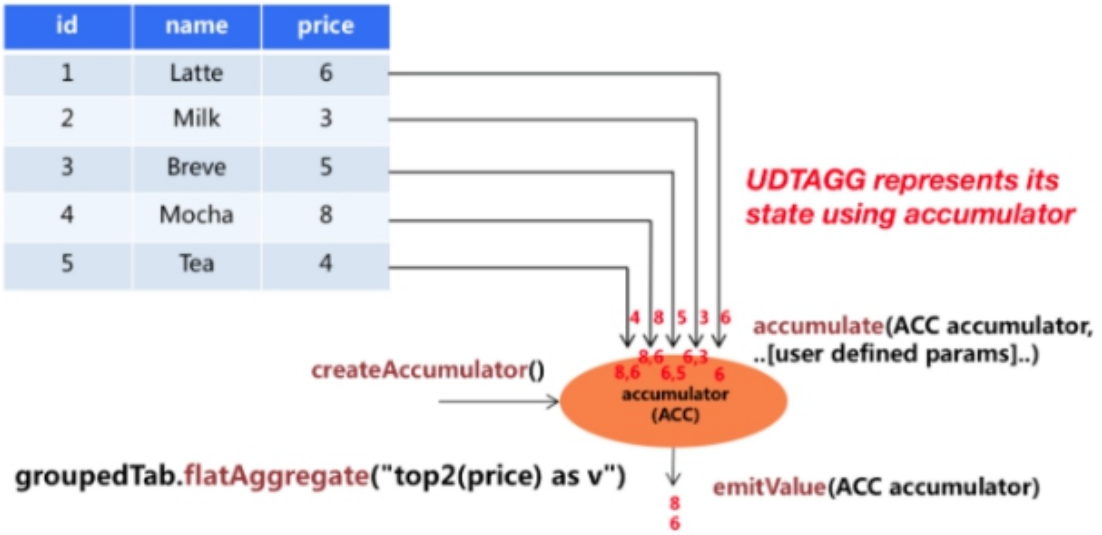
```
class AvgTemp extends AggregateFunction[Double, AvgTempAcc] {
  override def getValue(accumulator: AvgTempAcc): Double =
    accumulator.sum / accumulator.count

  override def createAccumulator(): AvgTempAcc = new AvgTempAcc

  def accumulate(accumulator: AvgTempAcc, temp: Double): Unit ={
    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
vaavgTemp = new AvgTemp()

// Table API的调用
varesultTable = sensorTable.groupBy('id)
  .aggregate(avgTemp('temperature) as 'avgTemp)
  .select('id, 'avgTemp)

// SQL的实现
tableEnv.createTemporaryView("sensor", sensorTable)
tableEnv.registerFunction("avgTemp", avgTemp)
varesultSqlTable = tableEnv.sqlQuery(
  """
    |SELECT
    |id, avgTemp(temperature)
    |FROM
    |sensor
    |GROUP BY id
  """.stripMargin)

// 转换成流打印输出
resultTable.toRetractStream[(String, Double)].print("agg temp")
resultSqlTable.toRetractStream[Row].print("agg temp sql")
```

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。【有问题都可以私聊我WX：focusbigdata，或者关注我的公众号：FocusBigData，注意大小写】



> 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。

```
1.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。

2.随后，对每个输入行调用函数的accumulate（）方法来更新累加器。

3.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
class Top2TempAcc{
  var highestTemp: Double = Int.MinValue
  var secondHighestTemp: Double = Int.MinValue
}

// 自定义 TableAggregateFunction
class Top2Temp extends TableAggregateFunction[(Double, Int), Top2TempAcc]{

  override def createAccumulator(): Top2TempAcc = new Top2TempAcc
```

```
  def accumulate(acc: Top2TempAcc, temp: Double): Unit ={
    if( temp > acc.highestTemp ){
      acc.secondHighestTemp = acc.highestTemp
      acc.highestTemp = temp
    } else if( temp > acc.secondHighestTemp ){
      acc.secondHighestTemp = temp
    }
  }

  def emitValue(acc: Top2TempAcc, out: Collector[(Double, Int)]): Unit ={
    out.collect(acc.highestTemp, 1)
    out.collect(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
vatop2Temp = new Top2Temp()

// Table API的调用
varesultTable = sensorTable.groupBy('id)
  .flatAggregate( top2Temp('temperature) as ('temp, 'rank) )
  .select('id, 'temp, 'rank)

// 转换成流打印输出
resultTable.toRetractStream[(String, Double, Int)].print("agg temp")
resultSqlTable.toRetractStream[Row].print("agg temp sql")
```