|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | |  |
|  | **Huawei StreamCQL 语法手册** | |  |
|  | |
| **文档版本** | **01** |
| **发布日期** | **2016-08-15** |
| **华为技术有限公司** | |

目 录

[1 前言 1](#_Toc498932413)

[2 系统功能列表 2](#_Toc498932414)

[3 关键概念定义 5](#_Toc498932415)

[4 CQL应用程序开发流程 6](#_Toc498932416)

[5 CQL语法约定 8](#_Toc498932417)

[6 数据类型 9](#_Toc498932418)

[6.1 基本数据类型 9](#_Toc498932419)

[6.2 空值 12](#_Toc498932420)

[7 CQL语法定义 13](#_Toc498932421)

[7.1 应用程序操作语句 13](#_Toc498932422)

[7.1.1 Drop 13](#_Toc498932423)

[7.1.2 Submit 14](#_Toc498932424)

[7.1.3 Explain 15](#_Toc498932425)

[7.1.4 Load 16](#_Toc498932426)

[7.1.5 Deactive 16](#_Toc498932427)

[7.1.6 Active 17](#_Toc498932428)

[7.1.7 Rebalance 17](#_Toc498932429)

[7.2 流操作语句 18](#_Toc498932430)

[7.2.1 Create Input Stream 18](#_Toc498932431)

[7.2.2 Create Output Stream 21](#_Toc498932432)

[7.3 查询语句 23](#_Toc498932433)

[7.3.1 Insert 23](#_Toc498932434)

[7.3.2 SELECT 25](#_Toc498932435)

[7.3.2.1 Select Clause 26](#_Toc498932436)

[7.3.2.2 From Clause 27](#_Toc498932437)

[7.3.2.3 Where Clause 32](#_Toc498932438)

[7.3.2.4 Group By Clause 33](#_Toc498932439)

[7.3.2.5 Having Clause 34](#_Toc498932440)

[7.3.2.6 Order By Clause 34](#_Toc498932441)

[7.3.2.7 Limit Clause 35](#_Toc498932442)

[7.3.2.8 Parallel Clause 36](#_Toc498932443)

[7.3.3 Subquery Clause 36](#_Toc498932444)

[7.3.4 DataSource 37](#_Toc498932445)

[7.3.4.1 Create DataSource 37](#_Toc498932446)

[7.3.4.2 DataSourceQuery 38](#_Toc498932447)

[7.3.5 User Defined Operator 40](#_Toc498932448)

[7.3.5.1 Create User Operator 40](#_Toc498932449)

[7.3.5.2 User Operator Query 42](#_Toc498932450)

[7.4 命令语句 42](#_Toc498932451)

[7.4.1 Set/Get 43](#_Toc498932452)

[7.4.2 Add Jar 44](#_Toc498932453)

[7.4.3 Add File 44](#_Toc498932454)

[7.4.4 Create Function 44](#_Toc498932455)

[7.4.5 Drop Function 45](#_Toc498932456)

[7.4.6 Show Applications 46](#_Toc498932457)

[8 窗口 48](#_Toc498932458)

[8.1 元组窗口 52](#_Toc498932459)

[8.2 范围窗口 54](#_Toc498932460)

[8.3 自然天窗口 57](#_Toc498932461)

[9 表达式 59](#_Toc498932462)

[9.1 通用表达式 63](#_Toc498932463)

[9.2 表达式常量 64](#_Toc498932464)

[9.3 特殊表达式 64](#_Toc498932465)

[9.3.1 Cast 64](#_Toc498932466)

[9.3.2 Case when 65](#_Toc498932467)

[9.3.3 Like 66](#_Toc498932468)

[9.3.4 Between 67](#_Toc498932469)

[9.3.5 In 67](#_Toc498932470)

[9.4 表达式数据类型转换规则 68](#_Toc498932471)

[9.4.1 算术表达式 68](#_Toc498932472)

[9.4.2 CAST表达式 68](#_Toc498932473)

[10 用户自定义接口 70](#_Toc498932474)

[10.1 数据序列化/反序列化 70](#_Toc498932475)

[10.2 流数据读取 77](#_Toc498932476)

[10.3 流数据写入 83](#_Toc498932477)

[10.4 UDF函数 88](#_Toc498932478)

[10.5 数据源 90](#_Toc498932479)

[10.6 自定义算子 95](#_Toc498932480)

[11 系统内置接口实现 101](#_Toc498932481)

[11.1 序列化/反序列化 101](#_Toc498932482)

[11.1.1 SimpleSerde 101](#_Toc498932483)

[11.1.2 KeyValueSerDe 102](#_Toc498932484)

[11.1.3 CSVSerDe 103](#_Toc498932485)

[11.1.4 BinarySerDe 104](#_Toc498932486)

[11.2 数据读取 106](#_Toc498932487)

[11.2.1 KafkaInput 106](#_Toc498932488)

[11.2.2 TCPClientInput 107](#_Toc498932489)

[11.2.3 RamdomGen 108](#_Toc498932490)

[11.3 数据写入 109](#_Toc498932491)

[11.3.1 KafkaOutput 109](#_Toc498932492)

[11.3.2 TCPClientOutput 110](#_Toc498932493)

[11.3.3 ConsoleOutput 111](#_Toc498932494)

[11.4 函数 112](#_Toc498932495)

[11.4.1 字符处理函数 112](#_Toc498932496)

[11.4.2 时间处理函数 114](#_Toc498932497)

[11.4.3 数学函数 122](#_Toc498932498)

[11.4.4 类型转换函数 123](#_Toc498932499)

[11.4.5 聚合函数 128](#_Toc498932500)

[11.5 数据源 135](#_Toc498932501)

[11.5.1 RDB数据源 135](#_Toc498932502)

[12 CQL其他配置参数 138](#_Toc498932503)

[13 CQL异常码 141](#_Toc498932504)

[14 CQL Java开发API说明 147](#_Toc498932505)

[15 CQL关键字列表 148](#_Toc498932506)

# 前言

CQL（Continuous Query Language），持续查询语言，用于数据流上的查询语言。相对于传统的SQL，CQL加入了窗口的概念，使得数据可以一直保存在内存中，由此可以快速进行大量内存计算，CQL的输出结果为数据流在某一时刻的计算结果。

CQL是建立在Storm基础上的类SQL查询语言，它解决了Storm原生API使用复杂，上手难度高，很多基本功能缺失的问题，提升了流处理产品的易用性。

在CQL设计语法之初，通过参考市面上现有的CEP产品的语法，发现这些产品都不算是全部的SQL语句，即仅仅使用SQL语句还不能运行，还必须依靠一些客户端的代码。 这样就给使用带来了一些不便， 用户必须学习客户端API，比较繁琐，上手难度也比较大。

所以，CQL设计目标就是，用纯粹的SQL语句再加上一些命令，就可以完成所有的任务发布以及执行，这样，就可以通过SQL接口，直接进行任务的下发，统一了客户端接口。对于有一定SQL基础的用户，只需要掌握一些CQL比较特殊的语法，比如窗口或者流定义的语法，就可以写出可运行的CQL程序，大大降低了上手难度。

# 系统功能列表

下表列出了系统的功能列表。

CQL功能列表

| 模块 | 功能 | 说明 |
| --- | --- | --- |
| 数据类型 | DataType | 支持int、String、long、float、double、boolean基本类型和time，date，timestamp时间类型以及decimal数字类型。 |
| 应用程序操作语句 | Create Input Stream | 创建输入流，包含数据读取方式和数据反序列化方法。 |
| Create Output Stream | 创建输出流，包含数据发送方式和数据序列化方法。 |
| Explain | 查看并导出执行计划。 |
| Load Application | 加载执行计划。 |
| Submit Application | 提交应用程序。 |
| Drop Application | 删除应用程序。 |
| Deactive Application | 去活应用程序。 |
| Active Application | 激活应用程序。 |
| Rebalance Application | 重分配应用程序。 |
| 查询语句 | Select | 查询语句。 |
| Insert as select | 查询结果重定向到另外一个流。 |
| multiInsert | 将一个流的数据经过不同的处理，发送到其他多个流当中。 |
| Window | 窗口。 |
| Join | 支持inner，left，right，full，cross五种Join类型。 |
| Group by | 分组。 |
| Order by | 对相同批次的输出数据进行排序。 |
| Filter Before Window | 数据进入窗口之前的过滤。 |
| Where | 数据进入窗口之后的过滤。 |
| Having | 聚合之后的过滤。 |
| Limit | 限制输出数量。 |
| DataSource | 数据源。 |
| User Operator | 用户自定义算子 |
| subQuery | 子查询。 |
| 命令 | Show applications | 查看已经提交的应用程序。 |
| Set/Get | 设置或读取配置属性。 |
| Add jar | 添加Jar包。 |
| Add file | 添加文件。 |
| Create function | 注册函数。 |
| Drop Function | 删除函数。 |
| 操作符 | 表达式 | 支持常见And，OR，IS NULL，NULL等，  支持'<','>','<=','>=', '=','==','!=',like, case when。 |
| 自定义数据序列化规则 | 通过自定义数据序列化规则， 实现对不同数据的解析。 |
| 自定义数据读取规则 | 拓展数据读取方式，实现不同文件或者数据类型的读取。 |
| 自定义数据写入规则 | 拓展数据发送方式，实现计算结果向不同的设备发送。 |

# 关键概念定义

* 流：流是一组（无穷）元素的集合，流上的每个元素都属于同一个schema；每个元素都和逻辑时间有关；即流包含了元组和时间的双重属性。留上的任何一个元素，都可以用Element<tuple,Time>的方式来表示，tuple是元组，包含了数据结构和数据内容，Time就是该数据的逻辑时间。
* Window：窗口（window）是流处理中解决事件的无边界（unbounded）及流动性的一种重要手段，把事件流在某一时刻变成静态的视图，以便进行类似数据库表的各种查询操作。在stream上可以定义window，窗口有两种类型，时间窗口（time-based）和记录窗口（row-based）。两种窗口都支持两种模式，滑动（slide）和跳动（tumble）。
* 表达式：符号和运算符的一种组合，CQL解析引擎处理该组合以获取单个值。简单表达式可以是常量、变量或者函数，可以用运算符将两个或者多个简单表达式联合起来构成更复杂的表达式。

# CQL应用程序开发流程

CQL应用程序的开发不同于传统的SQL语句。传统的SQL语句遵循先建表，再导入数据，再查询的顺序，每句SQL提交之后，都会得到SQL引擎的执行并获得返回结果；CQL语句是以应用程序作为一个整体运行的，一个Submit命令，代表一个应用程序的结束。一个应用程序中可以包含用户自定义参数、文件、jar包、函数等，还应该包含至少一个输入流、至少一个的输出流、以及至少一条输入数据计算语句。CQL中，一旦执行了Submit命令，系统就会在本地编译并提交应用程序，提交之后，会清空本地所有的用户操作记录，包含前期设置的参数、提交的用户jar包等。CQL语句提交过程中，如果发生语法错误，用户可以修改之后重新提交，如果发生了语义错误，那么输入内容都会被清空，用户必须从头提交整个应用程序。语义错误一般发生在explain语句或者submit语句执行过程中。

CQL语句并不严格要求各个不同语句的提交顺序，但是用户必须保证流（或者函数、参数等）在使用之前被定义。

CQL应用程序的编写顺序一般是：

| 序号 | 动作 | 说明 |
| --- | --- | --- |
| 1 | 添加自定义文件  添加自定义jar包 | 提交用户自定义功能用到的文件和jar包。  该步骤仅在包含用户自定义内容的时候存在。 |
| 2 | 创建自定义函数 | 定义用户的函数定义。  该步骤仅在包含用户自定义内容的时候存在。 |
| 3 | 创建输入流  创建输出流  创建用户自定义输入流，输出流，序列化，反序列化 | 创建输入输出流，既可以使用系统内置功能，也可以使用自定义功能。 |
| 4 | 编写逻辑计算语句 | 流处理业务计算逻辑核心。 |
| 5 | Submit提交应用程序 | 提交应用程序。 |

下面是一个CQL应用程序示例。

|  |
| --- |
| *--添加用户自定义接口实现的jar包*  add jar "/opt/streaming/example/example.jar";  *--使用用户自定义接口实现创建输入流*  CREATE INPUT STREAM S1(C1 STRING, C2 STRING)  SOURCE "com.huawei.streaming.example.userdefined.operator.input.FileInput"  PROPERTIES ("fileinput.path" = "/opt/streaming/example/input.txt");  *--创建输出流*  CREATE OUTPUT STREAM rs(C1 STRING,C2 STRING)  SINK kafkaOutput  PROPERTIES("operator.kafka.topic"="ttopic");  *--应用程序逻辑部分*  insert into stream rs select \* from S1;  *--提交应用程序*  submit application force example; |



CQL内，除了字符串(用双引号或者单引号括起来)和应用程序名称之外，其他的语法都不区分大小写，比如各种关键字，参数名称等，都不区分大小写。

# CQL语法约定

| 约定 | 作用 | 示例 |
| --- | --- | --- |
| 大写 | CQL语法关键字 | CREATE |
| *斜体* | 用户提供的CQL语法参数 | *stream\_name* |
| | (竖线) | 分隔语法项,只能使用其中一项。 | INT | STRING |
| *[]*（斜体方括号） | CQL语法中的符号，在窗口定义和数据源等地方用到。用斜体表示，以区分可选语法项。 | S *[*ROWS *row\_number* SLIDE [ EXCLUDE NOW ] *]* |
| [ ]（方括号） | 可选语法项。语法输入不包含方括号。 | *column\_name* [ AS *alias* ] |
| ,... | 指示前面的项可以重复多次。各项之间以逗号分隔。 | {*column\_name* <data\_type> [ <cql\_comment>]} ,… |
| … | 指示前面的项可以重复多次。各项之间以空格分隔。 | <multi\_insert>… |
| ; | CQL语法块终止符，代表一段语法定义的结束。 | DROP APPLICATION [IF EXISTS] *application\_name;* |
| <label>::= | 语法块的名称。此约定用于对可在语句中的多个位置使用的过长语法段或语法单元进行分组和标记。 | <data\_type>::=  INT|STRING; |
| <label> | 代表对一个语法块的引用 | <column\_list> |
| *--注释* | 语法定义和CQL示例中的说明文字，描述该语法块作用或者CQL示例程序功能。 | *--输入流完整语法格式定义* |

# 数据类型

[6.1 基本数据类型](#_ZH-CN_TOPIC_0020775486)

[6.2 空值](#_ZH-CN_TOPIC_0020775511)

## 基本数据类型

CQL基本数据类型

| 类型 | 范围 | 说明 |
| --- | --- | --- |
| Boolean | True或者False | 常量表示为：true或者false。  字符串输出的序列化反序列化规则见序列化、反序列化章节。 |
| Int | -231 ~ 231-1  即-2147483648~2147483647 | Int类型直接用数字表示 1,2,3… ，负数表示 -1,-2。 |
| Long | -263 ~ 263-1  即  -9223372036854775808~9223372036854775807 | Long类型表现和Int类型一致，常量表示中要添加后缀L(不区分大小写)，1L,-1l。 |
| Float | 1.4E-45~3.4028235E38 | Float类型和Int类型一致，但是要添加后缀F(不区分大小写)，如-1.0F,2f。 |
| Double | 4.9E-324~1.7976931348623157E308 | Double类型和Int类型一致，但是要添加后缀D(不区分大小写)，如-1.0d,2D。 |
| String |  | 字符串类型将字符用""或''包围起来，如"abc"或'abc'。 |
| Timestamp | Java.lang.Long数据类型取值范围 | 输入格式：yyyy-[M]M-[d]d [H]H:[m]m:[s]s[.SSS] [ Z]。  [.SSS]可以进行省略，也可以输入1-3位任意多数字,最多可以精确到纳秒。  输出格式：yyyy-MM-dd HH:mm:ss.SSS Z，统一精确到毫秒。  相关格式化方法和参数代表意义参照：  http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html。  字符串格式表示为 : '1970-01-01 00:00:00', '1970-01-01 00:00:00.001', '1970-01-01 00:00:00.001 +0800', CQL支持字符串格式的直接输入，系统内部会直接将字符串格式转为指定类型。  TimeStamp类型不支持常量类型输入。 |
| Date | Java.lang.Long数据类型取值范围 | 日期类型，包括年(四位)，月和日。  输入格式：yyyy-[M]M-[d]d。  输出格式：yyyy-MM-dd  相关格式化方法和参数代表意义参照：  http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html。  字符串表示为：'1970-01-01', CQL支持字符串格式的直接输入，系统内部会直接将字符串格式转为指定类型。  Date类型在CQL系统内部计算统一使用GMT+00:00时区。  Date类型不支持常量类型输入。 |
| Time | Java.lang.Long数据类型取值范围 | 一天中的时间，包括小时，分和秒。  输入格式：[H]H:[m]m:[s]s。  输出格式：HH:mm:ss  字符串表示范围从  '00:00:00'到’23:59:59’。  相关格式化方法和参数代表意义参照：  http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html。  字符串表示为：'00:00:01'，CQL支持字符串格式的直接输入，系统内部会直接将字符串格式转为指定类型。  Time类型在CQL系统内部计算统一使用GMT+00:00时区。  Time类型不支持常量类型输入。 |
| Decimal | Decimal 由任意精度的整数非标度值 和 32 位的整数标度 (scale) 组成。如果为零或正数，则标度是小数点后的位数；如果为负数，则将该数的非标度值乘以 10 的负 scale 次幂，因此，Decimal 表示的数值是 (unscaledValue × 10-scale)。 | 金额数据表示类型，金额数据类型运算时不会造成精度损失。  常量表示中要添加后缀BD(不区分大小写)，如1BD,-1bd。 |



所有数据类型在CQL中定义的时候，不区分大小写，允许大小写混合，即：int=Int=iNt=INT

所有的数据类型，在输入和输出中如果使用CQL内置的序列化类，输入数字不用添加'F'，'D'，'BD'等字符后缀。在CQL表达式中表示常量的时候，字符后缀必须添加。

Float、Double的范围表示方法为“科学记数法”，如1.99714E13=19971400000000。

系统不区分+0，-0,+0和+0.0,-0.0的区别，这些都等于0或者0.0。

系统在将字符串 格式化成Float,Double类型的时候,允许数字后面带'F'，'D'字符。但是Float，Double等类型输出的时候不会带'F'，'D'字符。

系统在进行数反序列化的时候不会检查数据是否溢出，由用户自己保证输入的数据在指定类型的范围内。

在时区表示中，+0000和-0000表示同一个时区。



Float类型和double类型属于浮点数类型，运算和表示均会导致精度丢失，所以不建议使用float或者double类型来进行高精度运算；高精度运算推荐使用decimal类型。

## 空值

NULL：

空值不属于任何数据类型，任何数据类型的值都会有Null值。

# CQL语法定义

[7.1 应用程序操作语句](#_ZH-CN_TOPIC_0020775506)

[7.2 流操作语句](#_ZH-CN_TOPIC_0020775513)

[7.3 查询语句](#_ZH-CN_TOPIC_0020775472)

[7.4 命令语句](#_ZH-CN_TOPIC_0020775473)

## 应用程序操作语句

### Drop

* **语法**

|  |
| --- |
| DROP APPLICATION [IF EXISTS] *application\_name;* |

* **参数**

*application\_name*：应用程序名称，可以是字母、数字或者下划线。

* **功能**

删除系统内的应用程序。

如果指定了IF EXISTS，那么删除之前就会检查该应用程序是否存在， 如果存在，则删除，如果不存在，就不会删除。

如果没有指定IF EXISTS, 那么在删除应用程序的时候，如果应用程序不存在，则会抛出异常。



用户可以删除所有自己能够看到的应用程序，不论安全还是非安全环境下。

* **示例**

|  |
| --- |
| DROP APPLICATION transformEvents; |
| DROP APPLICATION IF EXISTS transformEvents; |

### Submit

* **语法**

|  |
| --- |
| SUBMIT APPLICATION [FORCE] *application\_name [ path ]*; |

* **参数**

*application\_name*：应用程序名称, 可以是字母、数字或者下划线。

*path*：执行计划文件路径，可以用单引号或者双引号围起来。可选参数，如果使用，则会从指定路径获取执行计划并提交；*application\_name* 名称会覆盖执行计划中的应用程序名称。

* **功能**

SUBMIT APPLICATION主要进行应用程序的提交，一旦应用程序开始提交，系统就开始进行应用程序的解析和发布。

如果包含FORCE选项，就会在提交应用程序之前，先检查该应用程序是否已经存在，如果已经存在，则会先停止原来的应用程序，再提交当前应用程序。 如果没有包含Force选项，系统就会不做任何检查，直接提交，如果该应用程序已经存在，就会抛出异常。



应用程序一旦submit，CQL当前用户线程中，所有用户设定的参数、提交的 SQL语句都会被清空，当前客户端连接线程会被重置，以进行下个应用程序的提交。

如果submit应用程序的时候，系统剩余资源不足以满足应用程序进程需要，应用程序会提交成功，状态会变为active，但是执行应用程序的进程数目不足，剩余资源需求会加入到资源申请队列中，等待系统资源释放之后再加入新的进程执行该应用程序。

如果submit应用程序的时候，系统剩余资源为0，应用程序会提交成功，状态会变为active，但是不会有任何进程执行该应用程序，该应用程序资源会加入到资源申请队列中，等待系统资源释放。

在安全环境下，用户只能看到自己提交的应用程序，看不到其他人提交的应用程序，所以提交应用程序的时候可能会遇到提示应用程序名称已经存在，而用户又看不到的问题；管理员用户可以看到任何用户提交的应用程序。哪些用户属于管理员取决于Storm集群配置。

path参数一旦指定，会先清空用户之前的所有输入，再进行任务提交，所以会导致用户之前输入被清空。

path参数支持相对路径，该相对路径一般在应用程序的启动路径下，即在启动CQL命令行客户端或者CQL进程的当前路径下。

* **示例**

|  |
| --- |
| SUBMIT APPLICATION transformEvents; |
| SUBMIT APPLICATION FORCE transformEvents; |

### Explain

* **语法**

|  |
| --- |
| EXPLAIN APPLICATION *application\_name* [*path*]; |

* **参数**

*application\_name*：应用程序名称, 可以是字母、数字或者下划线。

*path*：执行计划文件路径，可以用单引号或者双引号围起来。



1、path参数支持相对路径，该相对路径一般在应用程序的启动路径下， 即在启动CQL命令行客户端或者CQL进程的当前路径下。

* **功能**

Explain语句用于显示当前所有已经提交的语句的查询计划。

查询计划显示出一个查询总共要分成多少个处理单元。并显示数据的输出和输出，可以用来显示CQL解析结果，并用于查询优化。

通过指定path文件地址，可以把查询计划导出到指定文件中。

Explain之后，可以继续提交新的CQL语句；只要没有执行submit application，所有的命令都可以下发。但是不允许下发重复的CQL语句，比如查询语句、插入语句等，这样可能会导致错误的解析结果。

* **示例**

|  |
| --- |
| EXPLAIN APPLICATION transform '/opt/cqlplan/transform.xml'; |

### Load

* **语法**

|  |
| --- |
| LOAD APPLICATION *path;* |

* **参数**

*path*： 执行计划文件路径，可以用单引号或者双引号围起来。

* **功能**

该语句用于加载执行计划。将查询计划加载到内存中，会默认清空在查询计划之前所提交的所有的SQL语句和参数设置。

* **示例**

|  |
| --- |
| LOAD APPLICATION '/opt/cqlplan/transform.xml'; |

### Deactive

* **语法**

|  |
| --- |
| DEACTIVE APPLICATION *application\_name;* |

* **参数**

*application\_name*：应用程序名称， 可以是字母、数字或者下划线。

* **功能**

DEACTIVE APPLICATION主要用来进行应用程序的去激活，即暂停当前应用程序的运行。



Deactive操作并不会导致应用程序进程退出，应用程序仍然在运行中，直到收到下一条数据消息处理之后，输入算子才会停止接收后续消息，这个时候，如果应用程序中包含了时间滑动窗、时间跳动窗等以系统时间且驱动的窗口的时候，仍然有可能输出数据。如果该应用程序进程异常退出，Streaming集群会自动拉起该进程并仍然处于Inactive状态。

* **示例**

|  |
| --- |
| DEACTIVE APPLICATION transformEvents; |

### Active

* **语法**

|  |
| --- |
| ACTIVE APPLICATION *application\_name;* |

* **参数**

*application\_name*：应用程序名称， 可以是字母、数字或者下划线。

* **功能**

ACTIVE APPLICATION主要用来进行应用程序的激活，即将当前应用程序从暂停状态恢复到运行状态，重新开始运行该应用程序。

* **示例**

|  |
| --- |
| ACITVE APPLICATION transformEvents; |

### Rebalance

* **语法**

|  |
| --- |
| REBALANCE APPLICATION *application\_name* SET WORKER *int\_number*; |

* **参数**

*application\_name*：应用程序名称， 可以是字母、数字或者下划线。

*int\_number*： 正整型数字。

* **功能**

CQL提供语法可以对应用程序进行重分配操作，在修改了应用程序的worker数量之后，重新启动应用程序。该命令只能用来调整worker数量，不能用来调整各类算子的并发度。



剩余资源不足，rebalance失败，会抛出异常提示用户资源不足。

申请资源超过该应用程序的总并发度，rebalance失败，会抛出异常提示用户资源不足。

Rebalance过程中，如果资源申请队列中有其他优先级更高的应用程序,可能会导致资源被其他应用程序抢占，Rebalance仍会成功执行，但结果会与申请数目不一致； 应用程序状态为Active，剩余资源需求会加入资源申请队列中，等待系统资源释放。

* **示例**

|  |
| --- |
| *--设置应用程序的进程数量为4。*  REBALANCE APPLICATION transformEvents SET WORKER 4; |

## 流操作语句

### Create Input Stream

* **语法**

|  |
| --- |
| *--输入流完整语法格式定义*  CREATE INPUT STREAM *stream\_name*  <column\_list> [ <cql\_comment> ]  [ <serde\_clause> ] <source\_clause>  [ <parallel\_clause> ]  ;  *--数据项列表定义*  <column\_list>::=  ( <column\_defined> ,… )  ;  *--单个数据列名称、列类型和列注释*  <column\_defined>::=  *column\_name* <data\_type> [ <cql\_comment>]  ;  *--反序列化组件及相关配置属性定义*  <serde\_clause>::=  SERDE *serde\_name* [<property\_list>]  ;  *--源数据读取组件及相关配置属性定义*  <source\_clause>::=  SOURCE *source\_name* [<property\_list>]  ;  *--并发度设置*  <parallel\_clause>::=  PARALLEL *int\_number*  ;  *--数据类型*  <data\_type>::=  INT  | LONG  | FLOAT  | DOUBLE  | BOOLEAN  | STRING  | DATE  | TIME  | TIMESTAMP  | DECIMAL  ;  *--配置属性列表*  <property\_list>::=  PROPERTIES ( <property\_pair> , ... )  ;  <property\_pair>::=  *property\_name* = *property\_value*  ;  *--输入流或者输出流的注释*  <cql\_comment>::=  COMMENT *comment\_string*  ; |

* **参数**

*stream\_name*：流名称，字符串形式，无需用单引号或者双引号围起来。

*column\_name*：列名称，字符串形式，无需用单引号或者双引号围起来。

*serde\_name*：序列化类名称，如果属于CQL内部实现，可以使用简写，比如SimpleSerDe。如果是用户自定义实现，则必须用双引号或者单引号围起来。

*source\_name*：输入数据读取类名称，如果属于CQL内部实现，可以使用简写，比如KafkaInput。如果是用户自定义实现，则必须用双引号或者单引号围起来。

*int\_number*：正整型数字。

*property\_name*：配置属性名称，字符串形式，必须用单引号或者双引号围起来。

*property\_value*：配置属性值，字符串形式，必须用单引号或者双引号围起来。

*comment\_string*：注释内容，字符串形式，必须用单引号或者双引号围起来。

* **功能**

Create Input Stream语句定义了输入流的数据列名称，数据读取方式和数据反序列化方式。

SERDE定义了数据的反序列化方式，即如何将从inputStream中读取的数据解析成对应的完整的流的Schema。系统配置了默认的序列化和反序列化类，当使用系统默认反序列化类时，SERDE子句可以省略，同时，SERDE的相关配置属性也可以省略。

SOURCE定义了数据从什么地方读取，比如从Kafka消息队列中读取或者文件等其他方式读取。SOURCE语句一定不能省略。

<parallel\_clause>语句指定了该输入算子的并发数量。

* **示例**

|  |
| --- |
| *--使用系统内置反序列化类读取算子示例*  CREATE INPUT STREAM example  (  eventId INT,  eventDesc STRING  )  COMMENT "this is an example of create input stream."  SERDE simpleSerDe  PROPERTIES ("separator" = "|")  SOURCE TCPClientInput  PROPERTIES ( "server" = "127.0.0.1",  "port" = "9999" )  Parallel 2; |
| CREATE INPUT STREAM example  (  eventId INT,  eventDesc STRING  )  COMMENT "this is an example of create input stream."  SOURCE "com.huawei.streaming.example.source.localFileReader"  PROPERTIES ("reader.path" = "/opt/streaming/example"); |

### Create Output Stream

* **语法**

|  |
| --- |
| *--输出流完整语法格式定义*  CREATE OUTPUT STREAM *stream\_name*  <column\_list> [ <cql\_comment> ]  [ <serde\_clause> ] <sink\_clause>  [ <parallel\_clause> ]  ;  *--数据项列表定义*  <column\_list>::=  ( <column\_defined> ,… )  ;  *--单个数据列名称、列类型和列注释*  <column\_defined>::=  *column\_name* <data\_type> [ <cql\_comment>]  ;  *--反序列化组件及相关配置属性定义*  <serde\_clause>::=  SERDE *serde\_name* [<property\_list>]  ;  *--流数据统计结果写入组件及相关配置属性定义*  <sink\_clause>::=  SINK *sink\_name* [<property\_list>]  ;  *--并发度设置*  <parallel\_clause>::=  PARALLEL *int\_number*  ;  *--数据类型*  <data\_type>::=  INT  | LONG  | FLOAT  | DOUBLE  | BOOLEAN  | STRING  | DATE  | TIME  | TIMESTAMP  | DECIMAL  ;  *--配置属性列表*  <property\_list>::=  PROPERTIES ( <property\_pair> , ... )  ;  <property\_pair>::=  *property\_name* = *property\_value*  ;  *--输入流或者输出流的注释*  <cql\_comment>::=  COMMENT *comment\_string*  ; |

* **参数**

*stream\_name*：流名称，字符串形式，无需用单引号或者双引号围起来。

*column\_name*：列名称，字符串形式，无需用单引号或者双引号围起来。

*serde\_name*：序列化类名称，如果属于CQL内部实现，可以使用简写，比如SimpleSerDe。如果是用户自定义实现，则必须用双引号或者单引号围起来。

*sink\_name*： 输出数据写入类名称，如果属于CQL内部实现，可以使用简写，比如KafkaOutput。如果是用户自定义实现，则必须用双引号或者单引号围起来。

*int\_number*：正整型数字。

*property\_name*：配置属性名称，字符串形式，必须用单引号或者双引号围起来。

*property\_value*：配置属性值，字符串形式，必须用单引号或者双引号围起来。

*comment\_string*：注释内容，字符串形式，必须用单引号或者双引号围起来。

* **功能**

Create Output Stream语句定义了输出流的数据列名称，数据写入方式和数据序列化方式。

SERDE定义了数据的序列化方式，将 系统计算完毕的流数据如何序列化成指定格式，比如CSV格式或者二进制格式等。各个数据列和output Schema中的定义一致。系统配置了默认的序列化和反序列化类，当使用系统默认反序列化类时，SERDE子句可以省略，同时，SERDE的相关配置属性也可以省略。

SINK定义了处理完毕的数据要写入哪里，比如写入Kafka或者写入文件等。Sink语句一定不能省略。

<parallel\_clause>语句指定了该输入算子的并发数量。

* **示例**

|  |
| --- |
| *--使用系统内置Kafka消息队列作为事件输出源。*  *--将数据格式化成CSV格式并输出*  CREATE OUTPUT STREAM transformOutput  (  cnt INT  )  SERDE csvSerDe  SINK kafkaOutput  PROPERTIES ("operator.kafka.topic" = "example")  parallel 2; |
| *--用户自定义序列化类*  *--必须在使用之前先通过add jar的方式提交用户自定义程序*  CREATE OUTPUT STREAM transformOutput  (  cnt INT  )  SERDE "com.huawei.streaming.example.serde.binarySerDe"  PROPERTIES ("binary.compress" = "true")  SINK kafkaOutput  PROPERTIES ("operator.kafka.topic" = "example")  parallel 2; |

## 查询语句

### Insert

* **语法**

|  |
| --- |
| *--单个Insert语法定义*  <insert\_clause> <select\_statement>;  <insert\_clause>::=  INSERT INTO STREAM *stream\_name*  ;  *--select语法块定义*  <selectStatement>::=  [ <select\_clause> ]  [ <from\_clause> ]  [ <where\_clause> ]  [ <groupby\_clause> ]  [ <having\_clause> ]  [ <orderby\_clause> ]  [ <limit\_clause> ]  [ <parallel\_clause> ]  ; |
| *-- multiInsert语句*  *--将一个流的数据按照不同的处理规则发送至不同的流*  *--from语句中只允许简单的流存在，不允许出现窗口等其他语法*  <from\_clause> <multi\_insert>… [ <parallel\_clause> ]  ;  *--MultiInsert语法定义*  <multi\_insert>::=  <insert\_clause> <multi\_select>  ;  *--MultiInsert场景下的select字句语法定义*  *--和系统其它select语句对比，只少了From子句和parallel子句*  <multi\_select>::=  [ <select\_clause> ]  [ <where\_clause> ]  [ <groupby\_clause> ]  [ <having\_clause> ]  [ <orderby\_clause> ]  [ <limit\_clause> ]  ; |

* **参数**

*stream\_name*：流名称，字符串形式，无需用单引号或者双引号围起来。

<select\_statement>：select子句, 详细语法定义见7.3.2 SELECT。

<parallel\_clause>：并发度设置子句，可选, 详细语法定义见7.3.2.8 Parallel Clause。

<select\_clause>：select查询字句，必选，用来表达查询内容, 详细语法定义见7.3.2.1 Select Clause。

<where\_clause>：where条件过滤子句，可选, 详细语法定义见7.3.2.3 Where Clause。

<groupby\_clause>：group by分组子句，可选, 详细语法定义见7.3.2.4 Group By Clause。

<having\_clause>： having聚合之后过滤子句，可选, 详细语法定义见7.3.2.5 Having Clause。

<orderby\_clause>：order by排序子句，可选, 详细语法定义见7.3.2.6 Order By Clause。

<limit\_clause>：输出限制子句，可选, 详细语法定义见7.3.2.7 Limit Clause。

* **功能**

可以将数据导入一个未定义的流中，但是如果有要将多个流的数据导入一个新流，这么几个导入语句生成的schema列名称和列类型必须相同。

允许将select结果导入到一个不存在的流中，只要这个流不是输入和输出流，系统就会自动创建该流。

MultiInsert语句一般用来进行单流数据的分割，它可以将一个流的数据，按照不同的处理规则，在处理完毕之后，发送至指定流。从而达到单流变多流的目的。

MultiInsert语句只有一个From字句，该字句中，只允许进行简单流的定义，不允许出现窗口等复杂语法。

* **示例**

|  |
| --- |
| INSERT INTO STREAM transformTemp SELECT \* FROM transform; |
| *--不包含子查询的MultiInsert*  FROM teststream  INSERT INTO STREAM s1 SELECT \*  INSERT INTO STREAM s2 SELECT a  INSERT INTO STREAM s3 SELECT id, name WHERE id> 10  Parallel 4; |
| *--包含子查询的MultiInsert*  FROM  (  SELECT count(id) as id, 'sss' as name  FROM testStream(id>5 )[RANGE 1 SECONDS SLIDE]  GROUP BY ss  )  INSERT INTO STREAM s1 SELECT \*  INSERT INTO STREAM s2 SELECT a  INSERT INTO STREAM s3 SELECT id,name WHERE id> 10; |

### SELECT

Select的语法结构由下面几个子句构成。

|  |
| --- |
| *--select\_statement语句必须结合insert语句一起使用，不能独立使用，*  *--所以这里采用语法块的方式定义。*  *--下面的其它子句定义也都采用语法块定义方式。*  <select\_statement>::=  [ <select\_clause> ]  [ <from\_clause> ]  [ <where\_clause> ]  [ <groupby\_clause> ]  [ <having\_clause> ]  [ <orderby\_clause> ]  [ <limit\_clause> ]  [ <parallel\_clause> ]  ; |

#### Select Clause

* **语法**

|  |
| --- |
| <select\_clause>::=  SELECT <select\_list>,…  ;  <select\_list>::=  [ *stream\_name* | *stream\_alias.*]\*  | [ *stream\_name* | *stream\_alias.*]*column\_name* [ [ AS ] *column\_alias* ]  | <expression> [ [ AS ] *column\_alias* ]  ; |

* **参数**

*stream\_name*：流名称，字符串形式，无需用单引号或者双引号围起来。

*column\_name*：列名称，字符串形式，无需用单引号或者双引号围起来。

*stream\_alias：*流别名，该别名在from子句中定义，在select子句中引用。字符串形式，无需用单引号或者双引号围起来。

*column\_alias：*流别名，该别名在select子句中定义，在where子句、having子句或者group by子句中引用。字符串形式，无需用单引号或者双引号围起来。

<expression>：CQL表达式子句。详见9 表达式。

* **功能**

Select子句包含了对输出数据计算方式的定义。

Distinct语法不支持直接在select子句中使用，只能放在UDAF函数中，比如count(distinct id)。

* **示例**

|  |
| --- |
| SELECT \* FROM transform[RANGE UNBOUNDED SLIDE]; |
| SELECT eventid FROM transform[RANGE UNBOUNDED]; |
| SELECT eventid, eventcode1 + transform.eventcde2 FROM transform; |
| SELECT eventid, sum(DISTINCT cde) FROM transform[RANGE UNBOUNDED SLIDE] GROUP BY eventtype; |

#### From Clause

* **语法**

|  |
| --- |
| *--From子句语法定义*  <from\_clause>::=  FROM <common\_from\_clause> | <join\_clause> |  <datasource\_clause> | <combine\_clause>  ;  *--一般的from子句定义*  <common\_from\_clause>::=  <common\_from\_stream> [ <filter\_before\_window> ]  [ <window\_source> ] [ [ AS ] *stream\_alias* ]  ;  *--join子句定义*  *--cross join不包含on条件*  *--只支持双流Join*  <join\_clause>::=  <common\_with\_unidirection> <common\_join> <common\_with\_unidirection> ON <expression>  | <common\_with\_unidirection> <corss\_join> <common\_with\_unidirection>  ;  *--单流Join，Unidirection语法定义*  <common\_with\_unidirection>::=  <common\_from\_clause> [UNIDIRECTION]  ;  *--数据源引用定义*  <datasource\_clause>::=  <common\_from\_clause> , <datasource\_body>  ;  *--combine子句定义*  <combine\_clause>::=  <combine\_body> <combine\_condition>  ;  <common\_from\_stream>::=  *stream\_name* | <subquery>  ;  <common\_join>::=  <inner\_join>  | <left\_join>  | <right\_join>  | <full\_join>  ;  <inner\_join>::=  [ INNER ] JOIN  ;  <left\_join>::=  LEFT [ OUTER ] JOIN  ;  <right\_join>::=  RIGHT [ OUTER ] JOIN  ;  <full\_join>::=  FULL [ OUTER ] JOIN  ;  <corss\_join>::=  , | CROSS JOIN  ;  *--数据源引用*  <datasource\_body>::=  DATASOURCE *datasource\_name* <datasource\_arguments> *datasource\_alias*  ;  <datasource\_arguments>::=  ( <datasource\_schema> , <datasource\_query> )  ;  *--数据源中使用到的schema*  <datasource\_schema>::=  SCHEMA ( <column\_defined> ,… )  ;  *--数据源查询语句，所有的CQL参数都在这个query中定义*  <datasource\_query>::=  QUERY ( <expression> ,… )  ;  *--单个数据列名称、列类型和列注释*  <column\_defined>::=  *column\_name* <data\_type> [ <cql\_comment> ]  ;  <combine\_body>::=  *stream\_name* ,...  ;  *-- Combine语句条件定义*  <combine\_condition>::=  COMBINE ( <expression>, ... )  ;  *--子查询定义*  <subquery>::=  ( <select\_statement> )  ;  *--输入流或者输出流的注释*  <cql\_comment>::=  COMMENT *comment\_string*  ; |

* **参数**

*stream\_name*：流名称，字符串形式，无需用单引号或者双引号围起来。

*stream\_alias：*流别名，该别名在from子句中定义，在select子句中引用。字符串形式，无需用单引号或者双引号围起来。

*datasource\_name*：数据源名称，和输入流中定义的数据源名称一致。

*datasource\_alias*：数据源别名。字符串形式，无需用单引号或者双引号围起来。

*column\_name*：列名称，字符串形式，无需用单引号或者双引号围起来。

*comment\_string*：注释内容，字符串形式，必须用单引号或者双引号围起来。

<window\_source>：window定义语法块，见8 窗口。

<expression>：CQL表达式子句。见9 表达式。

<select\_statement>：select子句, 见7.3.2 SELECT。

<data\_type>：数据类型定义语法块，见7.2 流操作语句。

* **功能**

From子句中包含对流的引用和对子查询的引用。如果包含子句，子句需要重命名。

Fiterbeforewindow子句是对进入窗口之前的数据进行过滤，比where子句的过滤发生的更早。使用FilterBeforeWindow子句，可以有效减少进入窗口的数据量。

Combine语句是对多个流的输出结果的合并。它会按照各个流的指定列进行匹配合并输出，通常用于有一个输入流产生了多个中间流，然后需要将这些中间流计算结果进行合并的场景。Combine的时候，各个中间流都必须包含一个可以进行匹配的字段。Combine的时候，各个表必须用逗号连接，不能使用Join等关键词。

Join支持内连接（inner join）、左连接（left outer join）、右连接（right outer join）、全连接（full outer join），交叉连接（cross join）。如果两个流以逗号相连，即隐含join条件，默认按照Cross join执行。Join的匹配规则和输出结果和SQL类似。

Join是对当前左流窗口内数据和右流窗口内数据按照条件进行连接处理。

各类Join类型描述

| 类型 | 描述 | 示例 |
| --- | --- | --- |
| [INNER] JOIN | 返回两侧流同时满足条件的组合 | From S1[range 20 seconds batch]  join S2[range unbounded]  on s1.id=s2.type;  from S1[range 20 seconds batch]  inner join S2[range unbounded]  on s1.id=s2.type; |
| LEFT [OUTER] JOIN | 显示符合条件的数据行，同时显示左边数据流不符合条件的数据行，右边没有对应的条目显示NULL | from S1[range 20 seconds batch]  left outer join S2[range unbounded]  on s1.id=s2.type;  from S1[range 20 seconds batch]  left join S2[range unbounded]  on s1.id=s2.type; |
| RIGHT [OUTER] JOIN | 显示符合条件的数据行，同时显示右边数据流不符合条件的数据行，左边没有对应的条目显示NULL | from S1[range 20 seconds batch]  right outer join S2[range unbounded]  on s1.id=s2.type;  from S1[range 20 seconds batch]  right join S2[range unbounded]  on s1.id=s2.type; |
| FULL [OUTER] JOIN | 显示符合条件的数据行，同时显示左右不符合条件的数据行，相应的左右两边显示NULL，即显示左连接、右连接和内连接的并集 | from S1[range 20 seconds batch]  full outer join S2[range unbounded]  on s1.id=s2.type;  from S1[range 20 seconds batch]  full join S2[range unbounded]  on s1.id=s2.type; |
| CROSS JOIN | 它将会返回被连接的两个表的笛卡尔积，返回结果的行数等于两个表行数的乘积 | From S1[range 20 seconds batch],  S2[range unbounded];  from S1[range 20 seconds batch]  cross join S2[range unbounded]; |



CQL目前只支持双流Join，并且只支持等值Join，且只支持属性表达式。比如s1 inner join s2 on s1.id=s2.id.

Unidirection关键词用来指定单向Join。在流处理系统内部Join的情况下，两个流都有数据产生，双方都会触发Join，如果两个流中，有一边包含了unidirection，那么另外一边，在有数据产生的时候，并不会触发Join，只有包含了Unidirection的这一端，才会触发Join。

From子句中还包含了对DataSource语法的定义，关于DataSource，可以看DataSource章节。

* **示例**

|  |
| --- |
| *--子查询示例*  SELECT \*  FROM  (  SELECT id, name, type  FROM transform  ) AS transformbak; |
| *--combine语句示例*  INSERT INTO STREAM realresult  SELECT transform.\*, transDetail.describe  FROM transform, transDetail  COMBINE( transform.eventid, transDetail.eventid ); |
| *--流前过滤示例*  SELECT \*  FROM transform ( evnetid> 10 )[range UNBOUNDED]  WHERE eventtype =1; |
| *--数据源示例*  insert into rs select rdb.id,s.id,count(rdb.id),sum(s.id) from S[rows 10 slide],  DATASOURCE rdbdatasource  [  SCHEMA (id int,name String,type int),  QUERY("select rid as id,rname,rtype from rdbtable where id = ? ",s.id)  ] rdb  where rdb.name like '%hdd%'  group by rdb.id,s.id; |
| *--inner join示例*  insert into stream rs select \* from S1[range 20 seconds batch] join S2[range unbounded] on s1.id=s2.type where s1.id> 5;  insert into stream rs select \* from S1[range 20 seconds batch] inner join S2[range unbounded] on s1.id=s2.type where s1.id> 5; |
| *--left join示例*  insert into stream rs select \* from S1[range 20 seconds batch] left outer join S2[range unbounded] on s1.id=s2.type where s1.id> 5;  insert into stream rs select \* from S1[range 20 seconds batch] left join S2[range unbounded] on s1.id=s2.type where s1.id> 5; |
| *--right join示例*  insert into stream rs select \* from S1[range 20 seconds batch] right outer join S2[range unbounded] on s1.id=s2.type where s1.id> 5;  insert into stream rs select \* from S1[range 20 seconds batch] right join S2[range unbounded] on s1.id=s2.type where s1.id> 5; |
| *--full join示例*  insert into stream rs select \* from S1[range 20 seconds batch] full outer join S2[range unbounded] on s1.id=s2.type where s1.id> 5;  insert into stream rs select \* from S1[range 20 seconds batch] full join S2[range unbounded] on s1.id=s2.type where s1.id> 5; |
| *--corss join示例*  insert into stream rs select \* from S1[range 20 seconds batch], S2[range unbounded] where s1.id> 5;  insert into stream rs select \* from S1[range 20 seconds batch] cross join S2[range unbounded] where s1.id> 5; |

#### Where Clause

* **语法**

|  |
| --- |
| <where\_clause>::=  WHERE <expression>  ; |

* **参数**

<expression>：CQL表达式子句。详见9 表达式。

**功能**

Where中的过滤条件一定是发生在数据进入窗口之后。聚合之前。所以where的条件表达式中一般不会出现聚合类的函数。

Expression可以是表达式或者其他用户自定义表达式，但是返回值必须是布尔类型。比如is null之类。

多个表达式之间可以使用AND或者其他的二元表达式。

* **示例**

|  |
| --- |
| WHERE a.size = 1 and a.size + b.size =2; |
| WHERE a.size = 1 and ( a.size + b.size )>= 2; |
| WHERE a.size = 1 and cast(a.isnative as boolean)=true; |

#### Group By Clause

* **语法**

|  |
| --- |
| <groupby\_clause>::=  GROUP BY <expression> , ...  ; |

* **参数**

<expression>：CQL表达式子句。详见9 表达式。

* **功能**

GroupBy的作用在于保证数据按照groupBy的字段进行分发，以及在算子内部，按照group by的字段进行分组以进行聚合计算。

Group by的操作在窗口之前，即先group by，再执行针对分组的属性的window。

允许group by的条件是一个UDF的表达式结果，即按照udf函数结果进行分组。



Group by中的表达式目前只支持流中的数据列，不支持其他表达式，比如函数等。

* **示例**

|  |
| --- |
| SELECT s.age,count(1) as cc  FROM schagetransformEvent[RANGE 1 SECOND BATCH] as s  GROUP BY s.age,s.departmentid; |

#### Having Clause

* **语法**

|  |
| --- |
| <having\_clause>::=  HAVING <expression>  ; |

* **参数**

<expression>：CQL表达式子句。详见9 表达式。

* **功能**

Having语句用来对聚合之后的结果数据进行过滤。

允许定义单独的having子句（即不使用group by，直接定义having）， 因为CQL的流式查询中默认已经包含了聚合的窗口。

Having中一般存放聚合操作的过滤条件，比如having count(\*)> 10。



Having条件过滤发生在聚合之后，这个时候系统内已经是计算之后的结果，所以过滤条件中不应该出现From子句中输入流的流名称和列名称。

* **示例**

|  |
| --- |
| SELECT  schages.age,  count(1) as cnt  FROM schagetransformEvent[RANGE 20 SECONDS BATCH]  GROUP BY schages.age  HAVING schages.age=10 AND cnt=2;  *--having中包含了s.age=10的条件，但是由于这个过滤条件放在了having中，所以这--个条件会在执行了count之后，再来执行*。 |

#### Order By Clause

* **语法**

|  |
| --- |
| <orderby\_clause>::=  ORDER BY <orderby\_column>, ...  ;  <orderby\_column>::=  <expression> [ ASC | DESC ]  ; |

* **参数**

<expression>：CQL表达式子句。详见9 表达式。

* **功能**

针对一个或者多个字段的排序。默认升序。

ASC升序排列。

DESC降序排列。

使用Orderby的时候，如果并发度不为1，那么则进行单个算子内部的排序。



Order by发生在聚合之后，这个时候系统内已经是计算之后的结果，所以过滤条件中不应该出现From子句中输入流的流名称和列名称。

* **示例**

|  |
| --- |
| SELECT \*  FROM schagetransformEvent[RANGE 20 SECONDS BATCH]  ORDER BY id DESC; |

#### Limit Clause

* **语法**

|  |
| --- |
| LIMIT *int\_number;* |

* **参数**

*int\_number*：正整型数字。

* **说明**

Limit主要用来限制输出，限制同一批次输出数据的数量。

如果同一批输出多条数据，则从前面开始去limit数量的数据。如果同一批次数据不足limit限制大小，则输出整个批次数据。

* **示例**

|  |
| --- |
| SELECT \* FROM ticketsRecorder[RANGE 20 SECONDS BATCH] LIMIT 10; |

#### Parallel Clause

* **语法**

|  |
| --- |
| PARALLEL *int\_number*; |

* **参数**

*int\_number*：正整型数字。

* **说明**

Parallel主要用设置该语句的并发数量，如果该语句中包含了子查询，该并发度设置也会影响到子查询的并发度；外部子查询的并发度会覆盖子查询的并发度。

该语句可选，默认并发度为1。

* **示例**

|  |
| --- |
| SELECT \* FROM ticketsRecorder[RANGE 20 SECONDS BATCH] LIMIT 10 Parallel 2; |
| INSERT INTO STREAM rs  SELECT \*, count(a)  FROM (  SELECT id as a,id as b, id c from S  )[RANGE 20 SECONDS BATCH]  WHERE a> 5  GROUP BY c  PARALLEL 4; |

### Subquery Clause

* **语法**

|  |
| --- |
| *--子查询定义*  <subquery>::=  ( <select\_statement> )  ; |

* **参数**

<select\_statement>：select子句, 见7.3.2 SELECT。

* **功能**

子查询可以在From子句中引用。

From语句中的子查询，必须显示标注别名才可以引用。

* **示例**

|  |
| --- |
| SELECT a.id  FROM (  SELECT count(tid) as id  FROM schagetransformEvent[RANGE UNBOUNDED]  ) AS a; |

### DataSource

数据源是存储数据的容器，其数据不是以流的形式往外吐的，而是按照触发的方式，根据条件，从数据源中进行查询。

可以把数据源想象成一个数据库，在每条事件到来的时候，都会进行一次查询，并返回查询结果。

数据源的使用分为数据源定义和数据源查询两部分。

#### Create DataSource

* **语法**

|  |
| --- |
| *--数据源定义语法*  CREATE DATASOURCE *datasource\_name*  SOURCE *class\_name*  <property\_list>  ;  *--配置属性列表*  <property\_list>::=  PROPERTIES ( <property\_pair> , ... )  ;  <property\_pair>::=  *property\_name* = *property\_value*  ; |

* **参数**

*datasource\_name*：数据源名称，和输入流中定义的数据源名称一致

*class\_name：*数据源类，字符串形式，必须用单引号或者双引号围起来, 如果属于CQL内部实现，可以使用简写,比如RDBDatasource。

*property\_name*：配置属性名称，字符串形式，必须用单引号或者双引号围起来。

*property\_value*：配置属性值，字符串形式，必须用单引号或者双引号围起来。

* **功能**

Create DataSource提供了数据源定义的语法。数据源的使用分为定义阶段和使用两个阶段。这里描述的是数据源的定义语法，数据源的使用语法见From子句定义。

数据源提供了一种访问外部数据的方式，比如实时访问关系型数据库，访问HBase等组件，支持从一行原始数据通过查询获取多行数据。

数据源的定义和具体使用无关，可以在一个拓扑中定义一个数据源，然后在其他地方多次使用该数据源。比如定义一个数据库数据源，然后可以在不同的地方使用不同的查询语句进行查询。

* **示例**

|  |
| --- |
| CREATE DATASOURCE rdbdatasource  SOURCE RDBDataSource  PROPERTIES (  "url" = "jdbc:postgresql://127.0.0.1:1521/streaming",  "userName" = "55B5B07CF57318642D38F0CEE0666D26",  "password" = "55B5B07CF57318642D38F0CEE0666D26"  ); |

#### DataSourceQuery

* **语法**

数据源和From子句一起结合使用，datasource\_body语法在from字句中也进行了定义。

|  |
| --- |
| *--数据源引用*  <datasource\_body>::=  DATASOURCE *datasource\_name* <datasource\_arguments> *datasource\_alias*  ;  <datasource\_arguments>::=  ( <datasource\_schema> , <datasource\_query> )  ;  *--数据源中使用到的schema*  <datasource\_schema>::=  SCHEMA ( <column\_defined> ,… )  ;  *--数据源查询语句，所有的CQL参数都在这个query中定义*  <datasource\_query>::=  QUERY ( <expression> ,… )  ;  *--单个数据列名称、列类型和列注释*  <column\_defined>::=  *column\_name* <data\_type> [ <cql\_comment> ]  ;  *--输入流或者输出流的注释*  <cql\_comment>::=  COMMENT *comment\_string*  ; |

* **参数**

*datasource\_name： 数据源名称*。字符串形式，无需用单引号或者双引号围起来。

*datasource\_alias*：数据源别名。字符串形式，无需用单引号或者双引号围起来。

*column\_name*：列名称，字符串形式，无需用单引号或者双引号围起来。

*comment\_string*：注释内容，字符串形式，必须用单引号或者双引号围起来。

<expression>：CQL表达式子句。见9 表达式。

<data\_type>：数据类型定义语法块，见7.2 流操作语句。

* **功能**

数据源的查询参数分为查询Schema定义和数据源查询两个部分。

Schema的定义同Create Input Stream中的schema定义，主要用来指定数据源查询结果的列数量、名称、类型，便于进行下一步处理。

* **示例**

|  |
| --- |
| *--RDB数据读取，支持多行数据读取，同时支持CQL UDF以及窗口和聚合运算*  *--QUERY内部的参数顺序固定，不同的数据源，有不同的参数。*  *--RDB的SQL中，如果不包含Where，就会一次查询出多行记录。和原始流做了Join之后，最终输出多条结果。*  insert into rs select rdb.id,s.id,count(rdb.id),sum(s.id) from S[rows 10 slide],  DATASOURCE rdbdatasource  [  SCHEMA (id int,name String,type int),  QUERY("select rid as id,rname,rtype from rdbtable where id = ? ", s.id)  ] rdb  where rdb.name like '%hdd%'  group by rdb.id,s.id; |

### User Defined Operator

当用户的业务逻辑比较复杂，无法用SQL表达出来的时候，就应该使用自定义算子来解决。用户只要实现了自定义算子的接口，然后就可以将这个算子和输入输出算子或者其他聚合算子一起结合使用。

用户自定义算子只支持单个输入schema和单个输出schema。

数据源的使用分为数据源定义和数据源查询两部分。

#### Create User Operator

* **语法**

|  |
| --- |
| CREATE OPERATOR *operator\_name* AS *class\_name*  <input\_schema\_statement> <output\_schema\_statement>  [ <property\_list>]  ;  *--自定义算子的输入schema*  <input\_schema\_statement>::=  INPUT ( <column\_list> )  ;  *--自定义算子的输出schema*  <output\_schema\_statement>::=  OUTPUT ( <column\_list> )  ;  *--数据项列表定义*  <column\_list>::=  ( <column\_defined> ,… )  ;  *--单个数据列名称、列类型和列注释*  <column\_defined>::=  *column\_name* <data\_type> [ <cql\_comment>]  ;  *--数据类型*  <data\_type>::=  INT  | LONG  | FLOAT  | DOUBLE  | BOOLEAN  | STRING  | DATE  | TIME  | TIMESTAMP  | DECIMAL  ;  *--配置属性列表*  <property\_list>::=  PROPERTIES (<property\_pair> , ... )  ;  <property\_pair>::=  *property\_name* = *property\_value*  ;  *--输入流或者输出流的注释*  <cql\_comment>::=  COMMENT *comment\_string*  ; |

* **参数**

*operator\_name：*自定义算子名称，字符串形式，无需用单引号或者双引号围起来。

*class\_name：*数据源类，字符串形式，必须用单引号或者双引号围起来, 如果属于CQL内部实现，可以使用简写,比如RDBDatasource。

*column\_name*：列名称，字符串形式，无需用单引号或者双引号围起来。

*property\_name*：配置属性名称，字符串形式，必须用单引号或者双引号围起来。

*property\_value*：配置属性值，字符串形式，必须用单引号或者双引号围起来。

*comment\_string*：注释内容，字符串形式，必须用单引号或者双引号围起来。

* **功能**

Create user operator提供了用户自定义算子的定义语法，在该语法中，可以定义算子的输入schema和输出schema，以及用到的参数，使用的类。

用户schema会在拓扑提交之前进行校验，系统会校验输出schema和输出schema的数据类型，如果数据类型不一致或者列数量不一致，都会导致检查失败，拓扑会提交失败。

* **示例**

|  |
| --- |
| create operator userOp as"com.huawei.streaming.example.userdefined.operator.UserOperator"  input (id int, name string)  output (newID string, name string, type int)  properties("userop.filename" = "/home/omm/kv.properties"); |

#### User Operator Query

* **语法**

|  |
| --- |
| *--自定义算子查询语法*  <insert\_clause> <using\_statement>  ;  <using\_statement>::=  USING OPERATOR *operator\_name* FROM *stream\_name*[ <distribute\_clause> ] [ <parallel\_clause> ]  ;  <distribute\_clause>::=  DISTRIBUTE BY *column\_name*  ;  *--并发度设置*  <parallel\_clause>::=  PARALLEL *int\_number*  ; |

* **参数**

*operator\_name：*自定义算子名称，字符串形式，无需用单引号或者双引号围起来。

*stream\_name*：流名称，字符串形式，无需用单引号或者双引号围起来。

*column\_name*：列名称，字符串形式，无需用单引号或者双引号围起来。

*int\_number*：正整型数字。

* **功能**

将用户自定义算子和其他输入输出算子或者其他功能算子集成，构成一个大拓扑，以实现用户功能。

* **示例**

|  |
| --- |
| insert into rs using operator userOp from s distribute by id parallel 2; |

## 命令语句

所有的系统命令，都会下发之后马上生效，马上可以查看到结果。

但是应用程序提交之后（即执行submit application之后），所有修改的参数、自定义函数和窗口等，都会被重置。

### Set/Get

* **语法**

|  |
| --- |
| SET *property\_name*= *property\_value;*  GET *property\_name;* |

* **参数**

*property\_name*：配置属性名称，字符串形式，必须用单引号或者双引号围起来。

*property\_value*：配置属性值，字符串形式，必须用单引号或者双引号围起来。

* **功能**

Set 设置系统内变量或者参数，get获取系统内变量或者参数。

配置参数的名称统一用字符形式表示，字符形式是不区分大小写的；但是对于特殊的名称，可以用双引号或者单引号包起来的字符串形式，字符串内容本身是区分大小写的。

变量值统一用字符串表示，字符串本身是区分大小写的。

可以在set和get命令中使用系统变量或者环境变量以及引用其他已经定义的配置属性。

* **示例**

Set "x"="1" 可以在系统内设置变量X，这个变量x可以在任何包含系统配置属性对象的地方使用。

Get "x" 可以获取变量x的值，如果不包含该变量，则返回NULL。

|  |
| --- |
| *--一般的set命令*  set 'streaming.serde.default' = 'com.huawei.streaming.serde.SimpleSerDe'; |
| *--set命令的字符串既可以用单引号，也可以用双引号*  set "serde.simpleserde.separator"=','; |
| *--set的配置属性值中引用其他配置属性的值*  set "x" = "abc";  set "x" = "${conf:x}def";  --get的值等于abcdef  get "x"; |
| *--set中使用系统变量*  set "streaming.template.directory" = "${system:java.io.tmpdir}"; |

### Add Jar

* **语法**

|  |
| --- |
| ADD JAR *path*; |

* **参数**

*path*：执行计划文件路径，可以用单引号或者双引号围起来。

* **功能**

在系统使用中，经常要用到用户接口，比如用户在CQL中使用自定义的序列化规则、自定义的输入输出或者自定义函数之类，这些程序，可以由用户自己打包成Jar包，再通过add jar的命令进行提交，然后就可以使用了。

连接关闭之后，用户手工添加的Jar文件会从本地进程中卸载，无需担心会影响到下一次连接的操作。

* **示例**

|  |
| --- |
| Add jar '/opt/huawei/streaming/test/test.jar'; |

### Add File

* **语法**

|  |
| --- |
| ADD FILE *path*; |

* **参数**

*path*：执行计划文件路径，可以用单引号或者双引号围起来。

* **功能**

在系统使用中，经常要用到用户接口，比如用户在CQL中使用自定义的序列化规则、自定义的输入输出或者自定义函数之类，在这些类中有时会用到配置文件或者资源文件，这些文件，可以通过add File的方式进行提交，在整个应用程序提交到远端集群之后访问。

* **示例**

|  |
| --- |
| ADD FILE '/opt/huawei/streaming/test/test.conf'; |

### Create Function

* **语法**

|  |
| --- |
| CREATE FUNCTION *function\_name* AS *class\_name* [ <property\_list> ]  *--配置属性列表*  <property\_list>::=  PROPERTIES (<property\_pair> , ... )  ;  <property\_pair>::=  *property\_name* = *property\_value*  ; |

* **参数**

*function\_name：*函数名称，不带双引号或则和单引号的字符串形式。

*class\_name：*数据源类，字符串形式，必须用单引号或者双引号围起来, 如果属于CQL内部实现，可以使用简写，比如RDBDatasource。

*property\_name*：配置属性名称，字符串形式，必须用单引号或者双引号围起来。

*property\_value*：配置属性值，字符串形式，必须用单引号或者双引号围起来。

* **功能**

该功能用于注册用户自定义函数，该函数名称不能和系统函数名称重复，如果重复，系统会抛出异常，给以提示。 用户自定义函数则允许通过重复定义来覆盖上次定义的函数。

所有的函数名称都不区分大小写。

函数无需刻意注销，在应用程序提交之后， 该函数即会失效，注册的Jar包和文件都会从本地进程中卸载。

* **示例**

|  |
| --- |
| CREATE FUNCTION rdbCompare AS 'com.huawei.streaming.example.udf.RDBCompare'; |
| create function udf1 as "com.huawei.streaming.cql.functions.DirectOutputUDF"  properties ("cql.test.direct.output" = "a"); |
| create function udf2 as "com.huawei.streaming.cql.functions.DirectOutputUDF"  properties ("cql.test.direct.output" = "b"); |
| create function udf3 as "com.huawei.streaming.cql.functions.DirectOutputUDF"; |

### Drop Function

* **语法**

|  |
| --- |
| DROP FUNCTION [IF EXISTS] *function\_name;* |

* **参数**

*function\_name*: 函数名称，不带双引号或则和单引号的字符串形式。

* **功能**

从系统中删除函数，函数名称不区分大小写。

在不包含IF EXISTS的场景下，如果函数不存在，则会抛出异常。

* **示例**

|  |
| --- |
| DROP FUNCTION rdbCompare; |

### Show Applications

* **定义**

|  |
| --- |
| SHOW APPLICATIONS [ *matched\_application\_name* ]; |

* **参数**

*matched\_application\_name*：应用程序名称，可选，可以通过双引号或者单引号围起来进行模糊查询。

* **功能**

查询应用程序信息。

查询结果由底层运行平台决定，一般包含应用程序名称，应用程序状态，应用程序启动时间等信息。

可以通过添加应用程序名称来进行应用程序过滤。

在安全环境下，一般用户只能看到自己提交的应用程序，管理员用户可以看到所有应用程序，哪些用户属于管理员取决于Storm集群的配置。

* **示例**

|  |
| --- |
| Show applications;  *--查询系统内正在运行的所有的应用程序* |
| Show applications filter;  *--查询系统内正在运行的，名称包含filter的应用程序* |
| Show applications 'filter\*';  *--查询系统内正在运行的，名称以filter开头的应用程*序 |
| Show applications '\*filter\*';  *--查询系统内正在运行的，名称包含filter的应用程序* |
| Show applications '\*filter';  *--查询系统内正在运行的，名称以filter结尾的应用程序* |

# 窗口

* **语法**

|  |
| --- |
| *--窗口语法定义*  *stream\_name* ( <filter\_before\_window> ) *[* <window\_source> *]*  *--不同类型窗口定义*  <window\_source>::=  <range\_window>  | <rows\_window>  | <range\_today>  ;  *--行组窗*  <rows\_window>::=  ROWS *int\_number* <window\_properties> <window\_determiner>  ;  *--时间范围窗口*  <range\_window>::=  RANGE <range\_bound> [<window\_properties>] <window\_determiner>  ;  *--时间范围为当天的窗口*  <range\_today>::=  RANGE TODAY <expression> <window\_determiner>  ;  *--窗口时间范围定义*  <range\_bound>::=  <range\_time>  | <range\_unbound>  ;  *--没有大小限制的窗口*  <range\_unbound>::=  UNBOUNDED  ;  *--窗口时间范围定义*  *--默认单位毫秒，允许组合，但是必须至少有一个值。*  <range\_time>::=  [<range\_day>] [<range\_hour>] [<range\_minutes>] [<range\_seconds>] [<range\_milliseconds>]  ;  *--单位为天的时间范围*  <range\_day>::=  *int\_number* DAY  | *int\_number* DAYS  ;  *--单位为小时的时间范围*  <range\_hour>::=  *int\_number* HOUR  | *int\_number* HOURS  ;  *--单位为分钟的时间范围*  <range\_minutes>::=  *int\_number* MINUTE  | *int\_number* MINUTES  ;  *--单位为秒的时间范围*  <range\_seconds>::=  *int\_number* SECOND  | *int\_number* SECONDS  ;  *--单位为毫秒的时间范围*  <range\_milliseconds>::=  *int\_number* MILLISECOND  | *int\_number* MILLISECONDS  ;  *--窗口属性，可以是滑动窗或者跳动窗*  <window\_properties>::=  SLIDE  | BATCH  ;  *--窗口的限定词，分区，排序，分组，触发方式或者是否排除当前事件。*  <window\_determiner>::=  [<partitionby\_determiner>]  [<sortby\_determiner>]  [<triggerby\_determiner>]  [<excludenow\_determiner>]  ;  *--窗口事件分组方式*  <partitionby\_determiner>::=  PARTITION BY <expression>  ;  *--事件排序方式*  <sortby\_determiner>  SORT BY <expression>  ;  *--窗口事件触发方式*  <triggerby\_determiner>::=  TRIGGER BY <expression>  ;  *--是否在计算的时候排除当前事件*  <excludenow\_determiner>::=  EXCLUDE NOW  ; |

* **参数**

*stream\_name*：流名称，字符串形式，无需用单引号或者双引号围起来。

*int\_number*：正整型数字。

<filter\_before\_window>：数据进入窗口之前的过滤表达式语法定义块。见7.3.2.2 From Clause。

<expression>：CQL表达式子句。见9 表达式。

* **功能**

窗口用来保存流上一段时间内的数据，并且由此产生过期事件。

窗口名称不区分大小写。

整个窗口由窗体、属性、限定词三部分组成。窗体主要描述窗口属于时间窗还是行组窗。属性主要描述窗口的数据输出是滑动还是跳动方式。限定词用来对窗口内的数据更新方式作出限定，比如按照指定字段进行分组，聚合运算是否包含当前事件等。

时间窗口中的时间单位可以表示为millisecond(s), second(s), minute(s), hour(s), day(s),既可以用单数，也可以用复数。

限定词可以定义多个，但是需要按照顺序进行定义。使用顺序如下：Partition by, Sort by, Trigger by, Exclude Now。

CQL的窗口有 窗体，属性和限定词三部分构成，一个窗口只能属于一种窗体，或者最多拥有一种属性，但是可以拥有多个限定词。

窗体分类说明

| 窗体名称 | 语法 | 说明 |
| --- | --- | --- |
| 行组窗 | ROWS *n1* | 数量为*n1*的元组窗。窗口内最多包含*n1*行数据 |
| 范围窗 | RANGE *t1* | *t1*时间单位范围内的时间窗。 |
| 自然天窗 | RANGE TODAY *exp1* | 窗口内保存当天的数据，格式为跳动窗。表达式*exp1*的返回值为时间类型(数字类型(int,long),timestamp, Date) |

窗口属性说明

| 窗口属性 | 语法 | 说明 |
| --- | --- | --- |
| 滑动窗 | SLIDE | 窗口数据以滑动方式输出，每次输出一条数据 |
| 跳动窗 | BATCH | 窗口数据以跳动方式输出，每次输出的数量，取决于窗口的容量。 |

窗口限定词说明

| 窗口限定词 | 语法 | 说明 |
| --- | --- | --- |
| 分组窗 | PARTITION BY *exp1* | 对窗口的数据按照表达式*exp1*进行分组。  分组窗中只支持单个表达式。 |
| 排序窗 | SORT BY *column\_name* | 对窗口内的数据按照时间表达式*exp1*的时间先后顺序进行排序  SORT BY只支持对列进行排序，不支持函数等表达式 |
| 事件驱动窗 | TRIGGER BY *exp1* | 窗口内的数据更新不依赖于系统内的规则，而依赖于外部事件更新。表达式*exp1*类型支持数字类型(int,long), timestamp, Date |
| 排除当前数据窗 | EXCLUDE NOW | 是否排除当前事件  从窗口中排除当前行数据。 |



所有的batch窗口都不支持EXCLUDE NOW和sort by功能。

TRIGGER BY不支持time数据类型。

EXCLUDE NOW数据排除当前行，所以查询语句中不能出现当前行，所以有EXCLUDE NOW的窗口，一般都只有count，sum等聚合函数或者分组列。

窗口数据输出分为新数据和旧数据，CQL中的窗口一般只输出新数据。

SORT BY排序窗口主要解决分布式数据乱序问题，该窗口只输出旧数据。

Join的时候，如果排序窗口和Unidirection在同一侧，那么统一输出旧数据，如果不在一侧，则输出新数据。

Join不支持一侧是排序窗口，一侧是其他非排序窗口的场景，因为新数据和旧数据不能一起计算。

分组窗会对数据进行分组，所以数据在输出的时候是按照分组的顺序进行输出，而不是数据进入窗口的顺序。分组的遍历顺序在系统内部使用hash方式遍历，和数据分组排序方式无关。

在实际使用过程中，分组窗口一般建议和Group by语法一起结合使用。

[8.1 元组窗口](#_ZH-CN_TOPIC_0020775501)

[8.2 范围窗口](#_ZH-CN_TOPIC_0020775525)

[8.3 自然天窗口](#_ZH-CN_TOPIC_0020775544)

## 元组窗口

* **语法**

|  |
| --- |
| ROWS *int\_number* <window\_properties> <window\_determiner>; |

* **参数**

*int\_number*：正整型数字。

<window\_properties>：窗口属性，滑动窗或者跳动窗，详见8 窗口。

<window\_determiner>：窗口限定词， 包含partition by，tirigger by，sort by等，详见8 窗口。

* **说明**

元组窗的主要作用在于在窗口内保存一定数量的事件，然后依据窗口的不同特性，在不同的时间过期，产生新旧事件。窗口内只能保存一定数量的事件。

元组窗口

| 语法 | 名称 | 说明 |
| --- | --- | --- |
| S*[* ROWS *n1* SLIDE [ EXCLUDE NOW ] *]* | 长度滑动窗 | 窗口内最大保存*n1*个事件，当有新事件产生的时候，窗口内的事件依次过期。每个过期事件的批次都不同。 |
| S*[* ROWS *n1* BATCH *]* | 长度跳动窗 | 窗口内最大保存*n1*个事件，当有新事件产生的时候，窗口内每攒满*n1*个事件，就过期依次。同时过期的所有事件处于同一批次。  该窗口不支持EXCLUDE NOW。 |
| S*[* ROWS *n1* SLIDE PARTITION BY *exp1* [ EXCLUDE NOW ] *]* | 分组长度滑动窗 | 同长度滑动窗，但是加入了分组的概念，事件归属于不同的分组，每个分组的长度为*n1*，逐个过期。 |
| S*[* ROWS *n1* BATCH PARTITION BY *exp1]* | 分组长度跳动窗 | 同长度跳动窗，但是加入了分组的概念，事件归属于不同的分组，每个分组的长度为*n1*，每组一个批次，同时过期。  该窗口不支持EXCLUDE NOW。 |
| S*[* ROWS *n1* SLIDE SORT BY *exp1* *]* | 元素排序窗 | 可以按照某一个字段的大小进行排序，也可以按照某一个时间戳字段的从前到后的固定顺序进行排序。窗口内数据依据排序结果逐个过期。 |

* **示例**

|  |
| --- |
| Select \* from transformEvent[rows 10 slide];  *--容量为10的滑动窗。* |
| Select \* from transformEvent[rows 10 slide partition by type];  *--按照type对窗口内数据进行分组，每组容量为10。* |
| Select \* from transformEvent[rows 1000 batch partition by type]; |
| Select \* from transformEvent[rows 1000 slide sort by ts]; |

## 范围窗口

* **语法**

|  |
| --- |
| RANGE <range\_bound> [<window\_properties>] <window\_determiner>; |

* **参数**

<range\_bound>：窗口事件范围，详见窗口定义语法。

<window\_properties>：窗口属性，滑动窗或者跳动窗，详见8 窗口。

<window\_determiner>：窗口限定词， 包含partition by，tirigger by，sort by等，详见8 窗口。

* **功能**

范围窗口的主要作用在于在窗口内保存一定数量的事件，然后依据窗口的不同特性，在不同的时间过期，产生新旧事件。窗口内只能保存指定时间范围的事件。

范围窗口

| 语法 | 名称 | 说明 |
| --- | --- | --- |
| S*[* RANGE UNBOUNDED [ EXCLUDE NOW ] *]* | 无限范围窗 | 窗口内可以保存任意多的数据，数据永远不会过期。 |
| S*[* RANGE *t1* SLIDE *]* | 时间滑动窗 | 窗口内保存最近*t1*时间范围内的数据，*t1*是一个时间单位，可以加入seconds等时间单位。窗口内的事件依次过期。每个过期事件的批次都不同。  该窗口不支持EXCLUDE NOW。 |
| S*[* RANGE *t1* BATCH *]* | 时间跳动窗 | 窗口内保存最近*t1*时间范围内的数据，*t1*是一个时间单位，可以加入Seconds等时间单位。窗口内的事件分批次过期，每次过期的数据属于同一个批次。  该窗口不支持EXCLUDE NOW。 |
| S*[* RANGE *t1* SLIDE PARTITION BY *exp1* *]* | 分组时间滑动窗 | 同时间滑动窗，但是加入了分组的概念。事件属于不同的分组，每个分组的时间范围为*t1*。  该窗口不支持EXCLUDE NOW。 |
| S*[* RANGE *t1* BATCH PARTITION BY *exp1* *]* | 分组时间跳动窗 | 同时间跳动窗，但是加入了分组的概念。事件属于不同的分组，每个分组的时间范围为*t1*。  该窗口不支持EXCLUDE NOW。 |
| S*[* RANGE *t1* SLIDE SORT BY *exp1* *]* | 时间排序窗口 | 窗口内只保存一定时间范围的的数据，每个新产生的数据都会和窗口内的数据做比较，将过期的数据输出，然后将新产生的数据按照顺序插入到窗口指定位置。  该窗口不支持EXCLUDE NOW。 |
| S*[* RANGE *t1* SLIDE TRIGGER BY *exp1* [ EXCLUDE NOW ] *]* | 事件驱动时间滑动窗 | 窗口内保存最近T1时间单位的数据，*exp1*是一个返回值为时间类型(long,timestamp,Date)的表达式，每次产生数据之后，都会和窗口内的数据做对比，然后将大于*t1*时间单位的数据吐出，每次只吐出一个数据。 |
| S*[* RANGE *t1* BATCH TRIGGER BY *exp1* *]* | 事件驱动时间跳动窗 | 窗口内保存最近*t1*时间单位的数据，*exp1*是一个返回值为时间类型(long,timestamp,Date)的表达式，每次产生数据之后，都会和窗口内的数据做对比，将大于*t1*时间范围的事件收集在一起，一起吐出。  该窗口不支持EXCLUDE NOW。 |
| S*[* RANGE *t1* SLIDE PARTITION BY *exp1* TRIGGER BY EXP2 [ EXCLUDE NOW ] *]* | 事件驱动分组时间滑动窗 | 同事件驱动时间滑动窗，不过加入了分组的概念，事件的滑动发生在每个分组上。 |
| S*[* RANGE *t1* BATCH PARTITION BY *exp1* TRIGGER BY *exp2* *]* | 事件驱动分组时间跳动窗 | 同事件驱动时间跳动窗，不过加入了分组的概念，事件的跳动，发生在每个分组上。  该窗口不支持EXCLUDE NOW。 |



时间排序窗在计算的时候，是和集群内时间进行比较，如果满足：【事件时间+窗口时间>系统时间】， 才会输出；如果：【事件时间+窗口时间<系统时间】，说明事件已经发生了很久了，就会马上输出。

1. **示例**

|  |
| --- |
| Select \* from transformEvent[range unbounded] as s1 inner join transformBak s2 on S1.id=S2.id; |
| Select \* from transformEvent[range 10 seconds slide]; |
| Select \* from transformEvent[range 1 day 6 hours slide]; |
| Select \* from transformEvent[range 10 seconds slide partition by type]; |
| Select \* from transformEvent[range 1000 milliseconds batch];  *--定义一个长度为1秒的时间跳动窗* |
| Select \* from transformEvent[range 1000 milliseconds batch partition by id]; |
| Select \* from transformEvent[range 1000 milliseconds sort by dte]; |
| insert into stream output\_where\_event\_tbatch  select sum(OrderPrice),avg(OrderPrice),count(OrderPrice)  from input\_where\_event\_tbatch[range 10 seconds slide trigger by TS exclude now]  where OrderPrice=100 or OrderPrice>=700; |
| insert into stream output\_where\_event\_tbatch  select sum(OrderPrice),avg(OrderPrice),count(OrderPrice)  from input\_where\_event\_tbatch[range 10 seconds slide partition by id trigger by TS exclude now]  where OrderPrice=100 or OrderPrice>=700; |
| insert into stream output\_where\_event\_tbatch  select sum(OrderPrice),avg(OrderPrice),count(OrderPrice)  from input\_where\_event\_tbatch[range 10 seconds batch trigger by TS exclude now]  where OrderPrice=100 or OrderPrice>=700; |
| insert into stream output\_where\_event\_tbatch  select sum(OrderPrice),avg(OrderPrice),count(OrderPrice)  from input\_where\_event\_tbatch[range 10 seconds batch partition by type trigger by TS exclude now]  where OrderPrice=100 or OrderPrice>=700; |

## 自然天窗口

* **语法**

|  |
| --- |
| RANGE TODAY <expression> <window\_determiner>; |

* **参数**

<expression>：CQL表达式子句。详见9 表达式。

<window\_determiner>：窗口限定词， 包含partition by，tirigger by，sort by等，详见8 窗口。

* **功能**

自然天窗口的主要作用在于在窗口内保存一定数量的事件，然后依据窗口的不同特性，在不同的时间过期，产生新旧事件。窗口内只能保存一定数量的事件。

自然天窗口

| 语法 | 名称 | 说明 |
| --- | --- | --- |
| S*[*RANGE TODAY *exp1* [ EXCLUDE NOW ] *]* | 自然天窗口 | 窗口内保存一整天的数据，根据新产生的事件判断时间是否已经超过一个自然天，如果超过，将窗口内保存的上一个自然天的数据全部过期。本身属于一个滑动窗口。  *exp1*是一个时间戳表达式，用于判断某个时间点是否是当天。 |
| S*[*RANGE TODAY *exp1* PARTITION BY *exp2* [ EXCLUDE NOW ] *]* | 分组自然天窗口 | 同自然天窗口，不过加入分组概念，事件分组过期。  *exp1*是一个时间戳表达式，用于判断某个时间点是否是当天。 |

* **示例**

|  |
| --- |
| insert into stream rs  select id,name,count(id)  from transformEvent[range today ts]  where id> 5  group by type  having id>10; |
| insert into stream rs  select id,name,count(id)  from transformEvent[range today ts partition by type]  where id> 5  group by type  having id>10; |

# 表达式

* **语法**

|  |
| --- |
| *--表达式定义*  <expression>::=  <logic\_or\_expression>  ;  *--or逻辑表达式定义*  <logic\_or\_expression>::=  <logic\_and\_expression> OR <logic\_and\_expression> ...  ;  *--and逻辑表达式定义*  <logic\_and\_expression>::=  <logic\_not\_expression> AND <logic\_not\_expression> ...  ;  *--not逻辑表达式定义*  <logic\_not\_expression>::=  [NOT ] <is\_null\_like\_in\_expressions>  ;  *--is null，like，in表达式定义*  <is\_null\_like\_in\_expressions>::=  <binary\_expression>  [IS <null\_condition>  <expression\_like>  <expression\_between>  |<expression\_in>  ]  ;  *--between表达式定义*  <expression\_between>::=  [NOT ] BETWEEN <bit\_expression> AND <bit\_expression>  ;  *--二元表达式定义*  <binary\_expression>::=  <bit\_expression> <relation\_expression>\*  ;  *--关系表达式定义*  <relation\_expression>::=  <relation\_operator> <bit\_expression>  ;  *--关系符号定义*  <relation\_operator>::=  =  |==  |<=>  |<>  |!=  |>  |>  |>=  |<=  ;  *--previous表达式定义*  <expression\_previous>::=  PREVIOUS ( <expression>)  ;  *--in表达式定义*  <expression\_in>::=  [NOT] IN ( <expression>,... )  ;  *--like表达式定义*  <expression\_like>::=  [NOT] LIKE <bit\_expression>  ;  *--算术加运算*  <bit\_expression>::=  <arithmetic\_star\_expression> <arithmetic\_plus\_operator> <arithmetic\_star\_expression> ...  ;  *--加减算数运算符号定义*  <arithmetic\_plus\_operator>::=  + |-  ;  -*-乘除算术运算定义*  <arithmetic\_star\_expression>::=  <field\_expression> <arithmetic\_star\_operator> <field\_expression> ...  ;  *--算术乘除运算符定义*  <arithmetic\_star\_operator>::=  \* |/ | %  ;  *--字段表达式*  <field\_expression>::=  [*stream\_name\_or\_stream\_alias*.]<atom\_expression>  ;  *--原子表达式定义*  <atom\_expression>::=  NULL  |<constant>  |<expression\_previous>  |<function>  |<cast\_expression>  |<case\_expression>  |<when\_expression>  |*column\_name*  | (<expression> )  ;  *--常量定义*  <constant>::=  <unary\_constant>  | *const\_string\_value*  | <boolean\_value>  ;  *--包含正负的数字常量定义*  <unary\_constant>::=  [ +| - ]  *const\_integer\_value*  | *const\_long\_value*  | *const\_float\_value*  | *const\_double\_value*  | *const\_bigdecimal\_value*  ;  *--boolean类型常量定义*  <boolean\_value>::=  TRUE  |FALSE  ;  *--常量定义*  <function>::=  *function\_name* ( [ DISTINCT <expression> ,... ] )  ;  *--cast表达式定义*  <cast\_expression>::=  CAST ( <expression> AS<data\_type> )  ;  *--case表达式定义*  <case\_expression>::=  CASE <expression> <when\_then\_expression>... [ ELSE <expression> ] END  ;  *--when表达式定义*  <when\_expression>::=  CASE <when\_then\_expression>... [ ELSE <expression>] END  ;  *--case when表达式中的when then表达式定义*  <when\_then\_expression>::=  WHEN <expression> THEN <expression>  ; |

* **参数**

*function\_name*： 函数名称，不带双引号或则和单引号的字符串形式。

*column\_name*：列名称，字符串形式，无需用单引号或者双引号围起来。

*const\_integer\_value*： 整型常量，Int类型直接用数字表示1,2,3… ，负数表示 -1,-2。

*const\_long\_value*：long类型常量，Long类型表现和Int类型一致，常量表示中要添加后缀L(不区分大小写)，1L,-1l。

*const\_float\_value*：float类型常量，Float类型和Int类型一致，但是要添加后缀F(不区分大小写)，如-1.0F,2f。

*const\_double\_value*：double类型常量，Double类型和Int类型一致，但是要添加后缀D(不区分大小写)，如-1.0d,2D。

*const\_bigdecimal\_value*：decimal类型常量，decimal数据类型运算时不会造成精度损失，常量表示中要添加后缀BD(不区分大小写)，如1BD,-1bd。

*const\_string\_value*：字符串类型常量，需要用单引号或者双引号围起来。

*stream\_name\_or\_stream\_alias*：流名称或者别名，字符串形式，无需用单引号或者双引号围起来。

<data\_type>：数据类型定义语法块，见7.2 流操作语句。

* **功能**

表达式：符号和运算符的一种组合，CQL解析引擎处理该组合以获取单个值。简单表达式可以是常量、变量或者函数，可以用运算符将两个或者多个简单表达式联合起来构成更复杂的表达式。

[9.1 通用表达式](#_ZH-CN_TOPIC_0020775516)

[9.2 表达式常量](#_ZH-CN_TOPIC_0020775474)

[9.3 特殊表达式](#_ZH-CN_TOPIC_0020775519)

[9.4 表达式数据类型转换规则](#_ZH-CN_TOPIC_0020775502)

## 通用表达式

通用表达式可以出现在select子句中，也可以出现在where或者having子句中。

表达式的优先级从高到底如下表所示。

通用表达式列表

| 表达式 | 说明 |
| --- | --- |
| +, - | 正数，负数之类一元操作符 |
| \*, / | 乘、除 |
| +, - | 二元操作符 |
| =, <>, <,>, <=,>=， != | 比较操作符 |
| IS [NOT] NULL, [NOT] LIKE, [NOT] BETWEEN,[NOT] IN | 是否为空之类判断表达式 |
| AND、OR | 逻辑表达式：多个条件之间是"且"或者"或"的关系 |



Date，time，timestamp数据类型不支持加减乘除比较等运算。

不支持字符串的加减乘除比较等运算。

正数、负数等一元操作符只限于常量数字类型，比如-1,-0.1f等。

Null和除And,OR之外任何通用表达式计算返回值都是null。

Null And True 返回 Null. Null Or true 返回true. Null Or false 返回null, Null and false 返回false。

## 表达式常量

CQL语句中的常量只能用在表达式中，这些表达式可以使用在select子句、where子句、having子句等其他地方。

String：用双引号或者单引号包围起来。比如"Streaming"。

Int：正常表示，比如1或者-1.

Long：以字母L结尾，不区分大小写。比如1l，2L。

Float：以字母F结尾，不区分大小写。比如1.0f,-1.0F。

Double：以字母D结尾，不区分大小写。比如1.0D，-1.0D。

Decimal: 以字母BD结尾，不区分大小写。比如1.0BD，-1.0BD。

Boolean: true或者false。

Time: 不支持，只能通过UDF函数进行转换。

Date: 不支持，只能通过UDF函数进行转换。

Timestamp: 不支持，只能通过UDF函数进行转换。

## 特殊表达式

### Cast

* **语法**

|  |
| --- |
| *--cast表达式定义*  <cast\_expression>::=  CAST( <expression> AS <data\_type> )  ; |

* **参数**

<expression>：CQL表达式子句。见9 表达式。

<data\_type>：数据类型定义语法块，见7.2 流操作语句。

* **返回类型**

和入参中datatype的数据类型一致。

* **功能**

将原始数据转化成其他数据类型。

* **示例**

|  |
| --- |
| INSERT INTO STREAM rs  SELECT  CAST(id AS int),  CAST(id AS long),  CAST(id AS float),  CAST(id AS double),  CAST(id AS String),  CAST(id AS boolean)  FROM s  WHERE NOT id != 'a'  AND type != 1; |

### Case when

* **语法**

|  |
| --- |
| *--case表达式定义*  <case\_expression>::=  CASE <expression> <when\_then\_expression>... [ ELSE <expression> ] END  ;  *--when表达式定义*  <when\_expression>::=  CASE <when\_then\_expression>... [ ELSE <expression>] END  ;  *--case when表达式中的when then表达式定义*  <when\_then\_expression>::=  WHEN <expression> THEN <expression>  ; |

* **参数**

<expression>：CQL表达式子句。见9 表达式。

THEN和ELSE表达式返回数据类型必须一致。

* **返回类型**

返回then或者else中表达式的计算结果。

* **功能**

条件判断表达式，根据条件，输出不同的结果。

* **示例**

|  |
| --- |
| CASE WHEN sex = 1 THEN Population ELSE 0 END; |
| case key when '1' then 1 when '2' then 2 when '3' then 3 else 99 end; |

### Like

* **语法**

|  |
| --- |
| *--like表达式定义*  <expression\_like>::=  [NOT] LIKE <bit\_expression>  ; |

* **参数**

<bit\_expression>: 算术加运算语法块，详见9 表达式。

* **返回类型**

Boolean

* **说明**

判断字符串是否匹配，支持正则,如果匹配，返回true，如果不匹配，返回false。

如果包含NOT，则取反。

* **示例**

1 % 包含零个或更多字符的任意字符串。

示例：WHERE title LIKE '%computer% ' 将查找处于书名任意位置的包含单词 computer 的所有书名。

2 \_（下划线） 任何单个字符。

示例：WHERE au\_fname LIKE '\_ean ' 将查找以 ean 结尾的所有 4 个字母的名字（Dean、Sean 等）。

3 [] 指定范围中的任何单个字符。

示例：WHERE au\_lname LIKE '[C-P]arsen ' 将查找以arsen 结尾且以介于 C 与 P 之间的任何单个字符开始的 作者姓氏，例如，Carsen、Larsen、Karsen 等。

4 [^] 不属于指定范围中的任何单个字符，与 [] 相反。

示例：WHERE au\_lname LIKE 'de[^l]% ' 将查找以 de 开始且其后的字母不为 l 的所有作者的姓氏。

### Between

* **语法**

|  |
| --- |
| *--between表达式定义*  <expression\_between>::=  [NOT ] BETWEEN <bit\_expression> AND <bit\_expression>  ; |

* **参数**

<bit\_expression>: 算术加运算语法块，详9 表达式。

* **返回类型**

boolean

* **功能**

判断某个字段是否在一定范围内。Between必须是小的值在前面，大的值在后面，返回值为boolean类型。

列的数据类型，AND前后两个值的数据类型必须一致。

如果包含NOT，则取反。

* **示例**

|  |
| --- |
| Size between 1 and 30; |

### In

* **语法**

|  |
| --- |
| *--in表达式定义*  <expression\_in>::=  [NOT] IN ( <expression>,... )  ; |

* **参数**

<expression>：CQL表达式子句。见9 表达式。

* **返回类型**

Boolean

* **功能**

判断列的值是否在范围内，列的数据类型和IN中所有常量的数据类型必须一致。

如果包含NOT，则取反。

* **示例**

|  |
| --- |
| Type in (1,2,3);  Type in ('a','b','c'); |

## 表达式数据类型转换规则

### 算术表达式

算术表达式包含表达式的加、减、乘、除以及比较操作。

两个不同的数据类型进行算数操作的时候，会自动向上转型, 不用经过特殊的函数转换。

CQL算术表达式数据类型转换规则从下到上依次为：

Int < Long < Float < Double < Decimal。

String、Boolean、Date、Time、TimeStamp数据类型不支持算术运算。

### CAST表达式

CAST表达式可以完成多种类型的数据转换。下面的表格展示了CQL的 CAST表达式从原始数据类型到目标数据类型的转换规则。

CAST表达式类型转换支持列表

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | String | Boolean | Int | Long | Float | Double | Decimal | Time | Date | TimeStamp |
| String | Y | Y | Y | Y | Y | Y | Y | Y | Y | Y |
| Boolean | Y | Y |  |  |  |  |  |  |  |  |
| Int | Y | Y | Y | Y | Y | Y | Y |  |  |  |
| Long | Y | Y | Y | Y | Y | Y | Y |  |  |  |
| Float | Y | Y | Y | Y | Y | Y | Y |  |  |  |
| Double | Y | Y | Y | Y | Y | Y | Y |  |  |  |
| Decimal | Y | Y | Y | Y | Y | Y |  |  |  |  |
| Time | Y | Y | Y | Y | Y | Y |  |  |  |  |
| Date | Y | Y | Y | Y | Y | Y |  |  |  |  |
| TimeStamp | Y | Y | Y | Y | Y | Y |  |  |  |  |



CAST的类型转换功能同类型转换函数一致。

时间类型的转换，一律使用Streaming集群默认时区，CQL目前不支持指定时区。

大于1的数字类型转为boolean类型都是true，小于1的数字类型转为boolean类型都是false。

# 用户自定义接口

[10.1 数据序列化/反序列化](#_ZH-CN_TOPIC_0020775547)

[10.2 流数据读取](#_ZH-CN_TOPIC_0020775470)

[10.3 流数据写入](#_ZH-CN_TOPIC_0020775518)

[10.4 UDF函数](#_ZH-CN_TOPIC_0020775454)

[10.5 数据源](#_ZH-CN_TOPIC_0020775481)

[10.6 自定义算子](#_ZH-CN_TOPIC_0020775550)

## 数据序列化/反序列化

* **说明**

数据的反序列化，必须实现com.huawei.streaming.serde.StreamSerDe接口。

也可以继承自com.huawei.streaming.serde.BaseSerDe类，该类提供了序列化的一些通用实现，比如创建空对象，数据类型转换等通用方法。

数据序列化反序列化接口说明

| 方法名 | 方法说明 | 参数 | 返回值 |
| --- | --- | --- | --- |
| setConfig | 设置序列化用到的配置属性。  编译期调用。 | StreamingConfig配置属性。 | Void |
| getConfig | 读取序列化类中的配置属性。  编译期调用。 | 无 | StreamingConfig配置属性 |
| initialize | 初始化接口。  编译期调用。 | StreamingConfig 配置参数。 | Void |
| setSchema | 设置输入或者输出schema。  编译期调用。  注意，该方法调用的比setConfig晚，所以不要在setConfig方法中读取Schema。 | TupleEventType  输入或者输出的schema。 | Void |
| deSerialize | 反序列化，将外部输入数据转化为系统可以识别的数据类型。  运行时调用。 | Object  外部第三方系统可以识别数据，比如纯字符串形式。 | List<Object[]>  内部可识别数据类型。  list的大小代表数据行的数量。  Object数组的大小和数据行中列的数量和顺序一致。 |
| serialize | 序列化，将系统计算结果转化为外部其他系统可以识别的数据。  运行时调用。 | List<Object[]>  内部计算结果  list的大小代表数据行的数量。  Object数组的大小和数据行中列的数量和顺序一致。 | Object  外部第三方系统可以识别数据，比如纯字符串形式。 |

* **示例**

|  |
| --- |
| package com.huawei.streaming.example.userdefined.serde;  import java.util.List;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import com.google.common.base.Splitter;  import com.google.common.collect.Lists;  import com.huawei.streaming.config.StreamingConfig;  import com.huawei.streaming.event.TupleEventType;  import com.huawei.streaming.exception.StreamSerDeException;  import com.huawei.streaming.exception.StreamingException;  import com.huawei.streaming.serde.StreamSerDe;  import com.huawei.streaming.util.DataTypeUtils;  /\*\*  \* 序列化和反序列化的例子  \*/  public class ExampleSerDe implements StreamSerDe  {  private static final long serialVersionUID = -8447913584480461044L;  private static final Logger LOG = LoggerFactory.getLogger(ExampleSerDe.class);  private TupleEventType schema;  private String separator = ",";  private StringBuilder sb = new StringBuilder();  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setConfig(StreamingConfig arg0)  throws StreamingException  {  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public StreamingConfig getConfig()  {  //读取配置属性  return null;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setSchema(TupleEventType outputSchema)  {  schema = outputSchema;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public TupleEventType getSchema()  {  return schema;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void initialize()  throws StreamSerDeException  {  }  /\*\*  \* 反序列化  \* 将从输入流中读取的数据解析成系统可识别数据类型  \*  \* @param data 输入流中读取的数据。  \* 输入算子不同，反序列化类中的data数据类型也不同，  \* 一般是字符串类型  \* 在示例程序中，按照字符串进行解析  \* @return 解析好的数据  \* 由于可能从一行衍生出多行数据，所以返回值类型是数组类型  \* @throws StreamSerDeException 序列化异常  \* @see [类、类#方法、类#成员]  \*/  @Override  public List<Object[]> deSerialize(Object data)  throws StreamSerDeException  {  if (data == null)  {  return null;  }  String stringValue = (String)data;  List<Object[]> splitResults = Lists.newArrayList();  Object[] values = Lists.newArrayList(Splitter.on(separator).split(stringValue)).toArray();  splitResults.add(values);  return createEventsInstance(splitResults);  }  /\*\*  \* 序列化方法  \* 将传入的事件进行序列化，转成output可以识别的对象  \* 或者字符串，或者其他类型  \*  \* @param events 系统产生或者处理过后的事件，是一个数组  \* @return 序列化完毕的数据，可以是字符串之类  \* @throws StreamSerDeException 序列化异常  \* @see [类、类#方法、类#成员]  \*/  @Override  public Object serialize(List<Object[]> events)  throws StreamSerDeException  {  if (events == null)  {  LOG.info("Input event is null");  return null;  }  clearTmpString();  for (Object[] event : events)  {  serializeEvent(event);  }  return removeLastChar();  }  /\*\*  \* 根据schema中的数据列，生成对应的数据类型  \*/  private List<Object[]> createEventsInstance(List<Object[]> events)  throws StreamSerDeException  {  if (events == null || events.size() == 0)  {  return Lists.newArrayList();  }  List<Object[]> list = Lists.newArrayList();  for (Object[] event : events)  {  Object[] eventInstance = createEventInstance(event);  list.add(eventInstance);  }  return list;  }  /\*\*  \* 创建单个事件实例  \*/  private Object[] createEventInstance(Object[] event)  throws StreamSerDeException  {  validateColumnSize(event);  Object[] arr = new Object[schema.getAllAttributes().length];  for (int i = 0; i <schema.getAllAttributeTypes().length; i++)  {  arr[i] = createInstance(schema.getAllAttributeTypes()[i], event[i]);  }  return arr;  }  private String removeLastChar()  {  return sb.substring(0, sb.length() - 1);  }  /\*\*  \* 根据指定类型创建数据实例  \*/  private Object createInstance(Class<?> clazz, Object value)  throws StreamSerDeException  {  try  {  return DataTypeUtils.createValue(clazz, value.toString());  }  catch (StreamingException e)  {  throw new StreamSerDeException(e.getMessage(), e);  }  }  /\*\*  \* 序列化单行事件  \*/  private void serializeEvent(Object[] event)  {  for (Object column : event)  {  appendToTmpString(column);  }  sb.replace(sb.length() - 1, sb.length() - 1, "");  }  /\*\*  \* 添加值到临时字符串最后面  \*/  private void appendToTmpString(Object val)  {  if (val != null)  {  sb.append(val.toString() + separator);  }  else  {  sb.append(separator);  }  }  /\*\*  \* 在序列化之前，先清空临时字符串  \*/  private void clearTmpString()  {  sb.delete(0, sb.length());  }  /\*\*  \* 检查列的数量和schema中列的数量是否一致  \*/  private void validateColumnSize(Object[] columns)  throws StreamSerDeException  {  if (columns.length != schema.getAllAttributeTypes().length)  {  LOG.error("deserializer result array size is not equal to the schema column size, "  + "schema size :{}, deserializer size :{}", schema.getAllAttributeTypes().length, columns.length);  throw new StreamSerDeException(  "deserializer result array size is not equal to the schema column size, schema size :"  + schema.getAllAttributeTypes().length + ", deserializer size :" + columns.length);  }  }  } |
| *--使用例子：*  *--打包成jar包，放在任意目录，比如/opt/streaming/example.jar*  *--执行如下CQL*  add jar "/opt/streaming/example.jar ";  create input stream S1  (  C1 STRING,  C2 STRING  )  SERDE "com.huawei.streaming.example.userdefined.serde.ExampleSerDe"  SOURCE kafkainput  PROPERTIES ( "operator.kafka.groupid" = "gidkpi\_1\_1","operator.kafka.topic"="agg\_1\_1\_1"  );  create output stream rs  (  C1 STRING,  C2 STRING  )  SERDE "com.huawei.streaming.example.userdefined.serde.ExampleSerDe"  SINK ConsoleOutput  PROPERTIES("printFrequence" = "10");  insert into stream rs select \* from S1;  submit application force read; |

## 流数据读取

* **说明**

从外部事件源中读取数据,需要实现com.huawei.streaming.operator.IInputStreamOperator接口。

流数据读取接口说明

| 方法名 | 方法说明 | 参数 | 返回值 |
| --- | --- | --- | --- |
| setConfig | 设置算子用到的配置属性。  编译时调用。 | StreamingConfig | Void |
| getConfig | 获取StreamingConfig配置参数。  编译时调用。 | 无 | StreamingConfig |
| setSerDe | 设置序列化/反序列化类。  运行时调用。 | StreamSerDe | Void |
| getSerDe | 获取设置序列化/反序列化类。  运行时调用。 | 无 | StreamSerDe |
| setEmitter | 设置Emitter对象。  运行时调用。 | IEmitter | Void |
| initialize | 初始化接口，运行时调用。 | 无 | Void |
| execute | 执行接口，运行时调用。 | 无 | Void |
| destroy | 关闭接口，运行时调用。 | 无 | Void |

* **示例**

|  |
| --- |
| package com.huawei.streaming.example.userdefined.operator.input;  import java.io.File;  import java.io.IOException;  import java.nio.charset.Charset;  import java.util.List;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import com.google.common.base.Strings;  import com.google.common.io.Files;  import com.google.common.io.LineProcessor;  import com.huawei.streaming.config.StreamingConfig;  import com.huawei.streaming.exception.StreamSerDeException;  import com.huawei.streaming.exception.StreamingException;  import com.huawei.streaming.exception.StreamingRuntimeException;  import com.huawei.streaming.operator.IEmitter;  import com.huawei.streaming.operator.IInputStreamOperator;  import com.huawei.streaming.serde.StreamSerDe;  /\*\*  \* 文件读取示例  \*/  public class FileInput implements IInputStreamOperator  {  private static final long serialVersionUID = 1145305812403368160L;  private static final Logger LOG =LoggerFactory.getLogger(FileInput.class);  private static final String CONF\_FILE\_PATH = "fileinput.path";  private static final Charset CHARSET =Charset.forName("UTF-8");  /\*\*  \* 文件路径  \*/  private String filePath;  private IEmitter emitter;  private StreamSerDe serde;  private StreamingConfig config;  /\*\*  \* file在initialize接口中被实例化  \* 在运行时被调用  \* 所以不用进行序列化传输  \*/  private transient File file;  private transient FileLineProcessor processor;  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setConfig(StreamingConfig conf)  throws StreamingException  {  this.filePath = conf.getStringValue(CONF\_FILE\_PATH);  this.config = conf;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setEmitter(IEmitter iEmitter)  {  this.emitter = iEmitter;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setSerDe(StreamSerDe streamSerDe)  {  this.serde = streamSerDe;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public StreamingConfig getConfig()  {  return this.config;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public StreamSerDe getSerDe()  {  return this.serde;  }  /\*\*  \* 初始化  \*  \* @throws StreamingException 初始化异常  \*/  @Override  public void initialize()  throws StreamingException  {  if (Strings.isNullOrEmpty(filePath))  {  LOG.error("file path is null.");  throw new StreamingException("file path is null.");  }  file = new File(filePath);  if (!file.exists())  {  LOG.error("file in path is not exists.");  throw new StreamingException("file in path is not exists.");  }  if (!file.isFile())  {  LOG.error("file in path is not a file type.");  throw new StreamingException("file in path is not a file type.");  }  processor = new FileLineProcessor();  }  /\*\*  \* 运行时的销毁接口  \*  \* @throws StreamingException 流处理异常  \* @see [类、类#方法、类#成员]  \*/  @Override  public void destroy()  throws StreamingException  {  }  /\*\*  \* 输入算子执行接口  \*  \* @throws StreamingException 流处理异常  \* @see [类、类#方法、类#成员]  \*/  @Override  public void execute()  throws StreamingException  {  try  {  Files.readLines(file, CHARSET, processor);  }  catch (IOException e)  {  throw StreamingException.wrapException(e);  }  }  private class FileLineProcessor implements LineProcessor<Object>  {  /\*\*  \* {@inheritDoc}  \*/  @Override  public boolean processLine(String line)  throws IOException  {  List<Object[]> deserResults = null;  try  {  deserResults =serde.deSerialize(line);  }  catch (StreamSerDeException e)  {  //忽略反序列化异常  LOG.warn("Ignore a serde exception.", e);  return false;  }  for (Object[] event : deserResults)  {  try  {  emitter.emit(event);  }  catch (StreamingException e)  {  //emit异常直接抛出，由外部异常处理机制自己处理  throw new StreamingRuntimeException(e);  }  }  return true;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public Object getResult()  {  return null;  }  }  } |
| *--使用步骤：*  *--打包成jar包，放在任意目录，比如/opt/streaming/example.jar*  *--在CQL语句中使用。*  add jar "/opt/streaming/example/example.jar";  create input stream S1  (  C1 STRING,  C2 STRING  )  SOURCE "com.huawei.streaming.example.userdefined.operator.input.FileInput"  PROPERTIES (  "fileinput.path" = "/opt/streaming/example/input.txt"  );  create output stream rs  (  C1 STRING,  C2 STRING  )  SINK "com.huawei.streaming.example.userdefined.operator.output.FileOutput"  PROPERTIES (  "fileOutput.path" = "/opt/streaming/example/output.txt"  );  insert into stream rs select \* from S1;  submit application force example; |

## 流数据写入

* **说明**

将计算好的结果写入外部事件源,需要实现com.huawei.streaming.operator. IOutputStreamOperator接口。

流数据写入接口说明

| 方法名 | 方法说明 | 参数 | 返回值 |
| --- | --- | --- | --- |
| setConfig | 设置算子用到的配置属性。  编译时调用。 | StreamingConfig | Void |
| getConfig | 获取StreamingConfig配置参数。  编译时调用。 | 无 | StreamingConfig |
| setSerDe | 设置序列化/反序列化类。  运行时调用。 | StreamSerDe | Void |
| getSerDe | 获取设置序列化/反序列化类。  运行时调用。 | 无 | StreamSerDe |
| initialize | 初始化接口，运行时调用。 | 无 | Void |
| execute | 执行接口，运行时调用。 | String streamName：流名称。  TupleEvent：元组数据。 | Void |
| destroy | 关闭接口，运行时调用。 | 无 | Void |

* **示例**

|  |
| --- |
| package com.huawei.streaming.example.userdefined.operator.output;  import java.io.File;  import java.io.IOException;  import java.nio.charset.Charset;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import com.google.common.base.Strings;  import com.google.common.io.Files;  import com.huawei.streaming.config.StreamingConfig;  import com.huawei.streaming.event.TupleEvent;  import com.huawei.streaming.exception.StreamSerDeException;  import com.huawei.streaming.exception.StreamingException;  import com.huawei.streaming.operator.IOutputStreamOperator;  import com.huawei.streaming.serde.BaseSerDe;  import com.huawei.streaming.serde.StreamSerDe;  /\*\*  \* 用户自定义接口示例  \* 文件输出示例  \*/  public class FileOutput implements IOutputStreamOperator  {  private static final long serialVersionUID = 153729627538127379L;  private static final Logger LOG =LoggerFactory.getLogger(FileOutput.class);  private static final String CONF\_FILE\_PATH ="fileOutput.path";  private static final Charset CHARSET =Charset.forName("UTF-8");  /\*\*  \* 文件路径  \*/  private String filePath;  private StreamSerDe serde;  private StreamingConfig config;  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setConfig(StreamingConfig conf)  throws StreamingException  {  this.filePath = conf.getStringValue(CONF\_FILE\_PATH);  this.config = conf;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setSerDe(StreamSerDe streamSerDe)  {  this.serde = streamSerDe;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public StreamingConfig getConfig()  {  return this.config;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public StreamSerDe getSerDe()  {  return this.serde;  }  /\*\*  \* 初始化  \*  \* @throws StreamingException 初始化异常  \*/  @Override  public void initialize()  throws StreamingException  {  }  /\*\*  \* {@inheritDoc}  \*  \* @param streamName  \* @param event  \*/  @Override  public void execute(String streamName, TupleEvent event)  throws StreamingException  {  if (Strings.isNullOrEmpty(filePath))  {  LOG.error("file path is null.");  throw new StreamingException("file path is null.");  }  File file = new File(filePath);  if (!file.exists())  {  LOG.error("file in path is not exists.");  throw new StreamingException("file in path is not exists.");  }  if (!file.isFile())  {  LOG.error("file in path is not a file type.");  throw new StreamingException("file in path is not a file type.");  }  String result = serializeEvent(event);  writeEvent(file, result);  }  /\*\*  \* 运行时的销毁接口  \*  \* @throws com.huawei.streaming.exception.StreamingException 流处理异常  \* @see [类、类#方法、类#成员]  \*/  @Override  public void destroy()  throws StreamingException  {  }  private String serializeEvent(TupleEvent event)  {  String result = null;  try  {  result = (String)serde.serialize(BaseSerDe.changeEventsToList(event));  }  catch (StreamSerDeException e)  {  LOG.error("failed to serialize data ", e);  }  return result;  }  private void writeEvent(File file, String result)  throws StreamingException  {  try  {  Files.append(result, file, CHARSET);  }  catch (IOException e)  {  throw StreamingException.wrapException(e);  }  }  } |
| *--打包成jar包，放在任意目录，比如/opt/streaming/example.jar*  *--在CQL语句中使用。*  add jar "/opt/streaming/example/example.jar";  create input stream S1  (  C1 STRING,  C2 STRING  )  SOURCE "com.huawei.streaming.example.userdefined.operator.input.FileInput"  PROPERTIES (  "fileinput.path" = "/opt/streaming/example/input.txt"  );  create output stream rs  (  C1 STRING,  C2 STRING  )  SINK "com.huawei.streaming.example.userdefined.operator.output.FileOutput"  PROPERTIES (  "fileOutput.path" = "/opt/streaming/example/output.txt"  );  insert into stream rs select \* from S1;  submit application force example; |

## UDF函数

* **说明**

用户自定义函数，可以在CQL的表达式中使用，可以将一个或者多个列进行处理之后，输出指定结果。

UDF函数接口说明

| 方法名 | 方法说明 | 参数 | 返回值 |
| --- | --- | --- | --- |
| 构造方法 | UDF函数构造方法 | Map<String, String>  配置参数。 | 无 |
| evaluate | 函数执行方法 | 不确定，UDF函数自定义，参数类型必须使用CQL支持数据类型，CQL中应用的时候，参数顺序必须和evaluate方法定义的顺序一致。 | 不确定，UDF函数自定义，但是必须是CQL支持的数据类型，且不能为Object。 |

用户自定义的UDF函数,一般继承自com.huawei.streaming.udfs.UDF这个类，构造参数只有一个Config，函数需要的所有的参数，都在这里放着。

参数可以通过set命令设置。

Udf函数必须实现evaluate方法，方法参数和返回值由用户依据使用确定，但是数据类型只能是系统支持的数据类型。

* **示例**

|  |
| --- |
| package com.huawei.streaming.example.udf;  import java.util.Map;  import com.huawei.streaming.udfs.UDF;  /\*\*  \* 移除字符串前后特殊字符  \*/  public class UDFTrim extends UDF  {  private static final long serialVersionUID = 4793756788804334850L;  /\*\*  \* <默认构造函数>  \*  \* @param config udf函数中需要的参数，  \* 这些参数要在cql中通过create function xx properties  \* 语法进行设置  \*/  public UDFTrim(Map<String, String> config)  {  super(config);  }  /\*\*  \* UDF函数的执行方法  \* 方法名称必须是evaluate。  \*  \* @param s 字符串  \* @return 移除空格之后的字符串  \* @see [类、类#方法、类#成员]  \*/  public String evaluate(String s)  {  if (s == null)  {  return null;  }  return s.trim();  }  } |

## 数据源

* **说明**

从外部数据源中读取数据，比如数据库。需要实现com.huawei.streaming.datasource IDataSource接口。

数据源接口说明

| 方法名 | 方法说明 | 参数 | 返回值 |
| --- | --- | --- | --- |
| setConfig | 设置数据源参数，编译期调用。 | StreamingConfig  查询参数 | void |
| setSchema | 设置数据源查询schema，编译时调用。 | TupleEventType  数据源查询结果对应schema | void |
| initialize | 初始化，运行时调用。 |  | void |
| execute | 执行数据源查询，运行时调用。 | List<Object>  完成了参数替换的查询参数。  该数组参数就是在From子句中定义的查询参数，该参数按照定义顺序保存在list列表中，并且在调用的时候已经完成了参数替换。 | List<Object[]>  查询结果，list的大小为查询出的结果行数，object数组为每行的数据结果，数据类型和setSchema方法中的schema数据类型一一对应。 |
| destroy | 销毁数据源内部对象，运行时调用。 |  | void |

* **示例**

|  |
| --- |
| package com.huawei.streaming.example.userdefined.datasource;  import java.io.BufferedReader;  import java.io.File;  import java.io.IOException;  import java.nio.charset.Charset;  import java.util.List;  import java.util.Properties;  import org.apache.storm.shade.org.apache.commons.io.IOUtils;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import com.google.common.base.Strings;  import com.google.common.collect.Lists;  import com.google.common.io.Files;  import com.huawei.streaming.config.StreamingConfig;  import com.huawei.streaming.datasource.IDataSource;  import com.huawei.streaming.event.TupleEventType;  import com.huawei.streaming.exception.ErrorCode;  import com.huawei.streaming.exception.StreamingException;  /\*\*  \* 属性匹配数据源  \* 作为示例，从本地文件中获取配置文件  \* 读取配置文件中的key，value对应关系  \* 返回对应的查询值  \*/  public class PropertyMatchDataSource implements IDataSource  {  public static final Charset CHARSET = Charset.forName("UTF-8");  private static final long serialVersionUID = 8056232432674642637L;  private static final Logger LOG = LoggerFactory.getLogger(PropertyMatchDataSource.class);  private static final String CONF\_FILE\_PATH = "example.datasource.path";  private String propertyFilePath;  private Properties properties;  private StreamingConfig config;  /\*\*  \* 这里的schema，对应查询语句中的schema定义，指的是数据源查询的输出schema  \* 在这个示例中并没有用到  \*/  private TupleEventType schema;  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setConfig(StreamingConfig conf)  throws StreamingException  {  config = conf;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setSchema(TupleEventType tuple)  {  schema = tuple;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void initialize()  throws StreamingException  {  initConfig();  if (Strings.isNullOrEmpty(propertyFilePath))  {  LOG.error("file path is null.");  throw new StreamingException("file path is null.");  }  File file = new File(propertyFilePath);  validateFile(file);  loadProperties(file);  }  private void initConfig()  throws StreamingException  {  //初始化，用于读取配置属性  if (config.containsKey(CONF\_FILE\_PATH))  {  this.propertyFilePath = config.get(CONF\_FILE\_PATH).toString();  }  else  {  LOG.error("can not found {} from configuration.", CONF\_FILE\_PATH);  throw new StreamingException(ErrorCode.CONFIG\_NOT\_FOUND, CONF\_FILE\_PATH);  }  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public List<Object[]> execute(List<Object> replacedQueryArguments)  throws StreamingException  {  validateArgs(replacedQueryArguments);  return evaluateValue(replacedQueryArguments.get(0));  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void destroy()  throws StreamingException  {  //用作打开资源的销毁，比如关闭流，关闭连接等  }  private List<Object[]> evaluateValue(Object replacedQueryArgument)  {  String key = replacedQueryArgument.toString();  if (properties.containsKey(key))  {  Object[] values = {properties.get(key)};  List<Object[]> results = Lists.newArrayList();  results.add(values);  return results;  }  return null;  }  private void validateFile(File file)  throws StreamingException  {  if (!file.exists())  {  LOG.error("file in path is not exists.");  throw new StreamingException("file in path is not exists.");  }  if (!file.isFile())  {  LOG.error("file in path is not a file type.");  throw new StreamingException("file in path is not a file type.");  }  }  private void loadProperties(File file)  throws StreamingException  {  properties = new Properties();  BufferedReader reader = null;  try  {  reader = Files.newReader(file, CHARSET);  properties.load(Files.newReader(file, CHARSET));  }  catch (IOException e)  {  LOG.error("failed to read property files.", e);  throw new StreamingException("failed to read property files.");  }  finally  {  IOUtils.closeQuietly(reader);  }  }  private void validateArgs(List<Object> replacedQueryArguments)  throws StreamingException  {  if (replacedQueryArguments.size() != 1)  {  LOG.error("rdb dataSource query arguments are not equal to 1, args size : {}", replacedQueryArguments.size());  throw new StreamingException("rdb dataSource query arguments are not equal to 1, args size : "  + replacedQueryArguments.size());  }  }  } |
| *--使用步骤：*  *--打包成jar包，放在任意目录，比如/opt/streaming/example.jar*  *--执行add jar '/opt/streaming/example.jar';*  *--在CQL语句中使用。*  add jar '/opt/streaming/example.jar';  create input stream S  (id int,name String)  SOURCE KafkaInput  PROPERTIES (  "operator.kafka.groupid" = "gidkpi\_1\_1",  "operator.kafka.topic" = "0912"  );  CREATE DATASOURCE propertyDataSource  SOURCE "com.huawei.streaming.example.datasource.PropertyMatchDataSource"  PROPERTIES (  "example.datasource.path" = "/opt/example.properties");  create output stream rs  (type String, id String, tag String)  SINK consoleOutput;  insert into rs select s.id,s.name,ds.value from S,  DATASOURCE propertyDataSource  [  SCHEMA (value String),  QUERY(s.id)  ] ds;  submit application datasourcetest; |

## 自定义算子

* **说明**

用户自定义处理逻辑，目前仅支持单流输入和单流输出。需要实现com.huawei.streaming.operator.IFunctionStreamOperator接口。

自定义算子接口说明

| 方法名 | 方法说明 | 参数 | 返回值 |
| --- | --- | --- | --- |
| setConfig(StreamingConfig conf) | StreamingConfig：配置 参数。 | Void | 设置配置参数，编译时接口。 |
| getConfig() |  | Void | StreamingConfig配置参数。  编译时接口。 |
| setEmitter(Map<String, IEmitter> emitterMap) | Map中的key为流名称，value为emitter实例。 | void | 设置多个emiter，每个流对应一个emiter实例。运行时接口。 |
| Initialize() |  | Void | 运行时初始化接口。 |
| execute(String streamName, TupleEvent event) | StreamingName：元组数据所在流名称  TupleEvent：元组数据 | Void | 运行接口，数据的处理逻辑就发生在该接口内。 |
| destroy() | 关闭算子。 | Void | 销毁算子内部创建对象。  运行时接口。 |

* **示例**

|  |
| --- |
| package com.huawei.streaming.example.userdefined.datasource;  import java.io.BufferedReader;  import java.io.File;  import java.io.IOException;  import java.nio.charset.Charset;  import java.util.List;  import java.util.Properties;  import org.apache.storm.shade.org.apache.commons.io.IOUtils;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import com.google.common.base.Strings;  import com.google.common.collect.Lists;  import com.google.common.io.Files;  import com.huawei.streaming.config.StreamingConfig;  import com.huawei.streaming.datasource.IDataSource;  import com.huawei.streaming.event.TupleEventType;  import com.huawei.streaming.exception.ErrorCode;  import com.huawei.streaming.exception.StreamingException;  public class UserOperator implements IFunctionStreamOperator  {  private static final Logger LOG =LoggerFactory.getLogger(UserOperator.class);  public static final Charset CHARSET =Charset.forName("UTF-8");  private static final long serialVersionUID = -4438239751340766284L;  private static final String CONF\_FILE\_NAME ="userop.filename";  private String fileName;  private Properties properties;  private Map<String, IEmitter> emitters = null;  private StreamingConfig config;  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setConfig(StreamingConfig conf)  throws StreamingException  {  if (!conf.containsKey(CONF\_FILE\_NAME))  {  LOG.error("can not found config value {}.", CONF\_FILE\_NAME);  throw new StreamingException("can not found config value " +CONF\_FILE\_NAME + ".");  }  fileName = conf.getStringValue(CONF\_FILE\_NAME);  this.config = conf;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public StreamingConfig getConfig()  {  return this.config;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void setEmitter(Map<String, IEmitter> emitterMap)  {  if (emitterMap == null || emitterMap.isEmpty())  {  LOG.error("can not found emitter.");  throw new StreamingRuntimeException("can not found config value" + CONF\_FILE\_NAME + ".");  }  emitters = emitterMap;  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void initialize()  throws StreamingException  {  File file = new File(fileName);  validateFile(file);  loadProperties(file);  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void execute(String streamName, TupleEvent event)  throws StreamingException  {  Object[] values = event.getAllValues();  Object[] result = new Object[3];  if (properties.containsKey(String.valueOf(values[0])))  {  result[0] = properties.get(String.valueOf(values[0]));  result[1] = 1;  result[2] = 1.0f;  }  else  {  result[0] = "NONE";  result[1] = 1;  result[2] = 1.0F;  }  for (IEmitter emitter : emitters.values())  {  emitter.emit(result);  }  }  /\*\*  \* {@inheritDoc}  \*/  @Override  public void destroy()  throws StreamingException  {  }  private void validateFile(File file)  throws StreamingException  {  if (!file.exists())  {  LOG.error("file in path is not exists.");  throw new StreamingException("file in path is not exists.");  }  if (!file.isFile())  {  LOG.error("file in path is not a file type.");  throw new StreamingException("file in path is not a file type.");  }  }  private void loadProperties(File file)  throws StreamingException  {  properties = new Properties();  BufferedReader reader = null;  try  {  reader = Files.newReader(file, CHARSET);  properties.load(Files.newReader(file, CHARSET));  }  catch (IOException e)  {  LOG.error("failed to read property files.", e);  throw new StreamingException("failed to read property files.");  }  finally  {  IOUtils.closeQuietly(reader);;  }  }  } |
| *--使用步骤：*  *-打包成jar包，放在任意目录，比如/opt/streaming/example.jar*  *--执行add jar '/opt/streaming/example.jar';*  *--在CQL语句中使用。*  add jar '/opt/streaming/example.jar';  create input stream s (id int, name string)  SOURCE KafkaInput  PROPERTIES ( "operator.kafka.groupid" = "gidkpi\_1\_1","operator.kafka.topic"="agg\_1\_1");  create output stream rs  (id string, name string, type int)  SINK KafkaOutput properties("operator.kafka.topic"="agg\_2");  create operator userOp as "com.huawei.streaming.example.userdefined.operator.UserOperator"  input (id int, name string)  output (newID string, name string, type int)  properties ("userop.filename" = "/home/omm/kv.properties");  insert into rs using operator userOp from s distribute by id parallel 2;  submit application force tt; |

# 系统内置接口实现

本章描述各类系统内置的实现及其参数，以供在CQL中使用。

各种内置接口参数的默认值仅供参考，可能会和实际使用场景有所不同。



系统内置实现接口内所有参数名称不区分大小写

[11.1 序列化/反序列化](#_ZH-CN_TOPIC_0020775468)

[11.2 数据读取](#_ZH-CN_TOPIC_0020775542)

[11.3 数据写入](#_ZH-CN_TOPIC_0020775532)

[11.4 函数](#_ZH-CN_TOPIC_0020775503)

[11.5 数据源](#_ZH-CN_TOPIC_0020775543)

## 序列化/反序列化

### SimpleSerde

* **功能**

按照指定分隔符拆分数据并根据schema类型进行反序列化。



Boolean类型的反序列化，如果字符串是true（不区分大小写），则转为true，如果字符串是false（不区分大小写）则转为false，除此之外，其他字符都是null。

* **参数**

系统默认序列化反序列化接口参数

| 名称 | 默认值 | 参数说明 |
| --- | --- | --- |
| separator | , | 每个列之间的分隔符。 |
| timezoneForTimestamp | ${system:user.timezone} | 数据序列化和反序列化的时区设定， 默认使用客户端时区。  参数设置格式支持如下两种方式：  GMT+08:00, Asia/Shanghai。 |

* **示例**

|  |
| --- |
| CREATE INPUT STREAM S  (id INT, name STRING )  SERDE simpleSerde  PROPERTIES ( "separator" = "," )  SOURCE KafkaInput  PROPERTIES (  "operator.kafka.groupid" = "gidkpi\_1\_1",  "operator.kafka.topic" = "simple\_1"  ); |

### KeyValueSerDe

* **功能**

将数据序列化成KeyValue的形式，类似key1=value1,key2=value2。



Boolean类型的反序列化，如果字符串是true(不区分大小写),则转为true，如果字符串是false(不区分大小写),则转为false，除此之外，其他字符都是null。

Keyvalue序列化反序列化中,键和值使用等'='进行连接，等号不可更改，这就要求在进行keyvalue格式序列化反序列化的时候，键或者值中不能包含等号，否则会导致结果错误。

* **参数**

KeyValue格式序列化反序列化参数

| 名称 | 默认值 | 参数说明 |
| --- | --- | --- |
| separator | , | 每个列之间的分隔符。 |
| timezoneForTimestamp | ${system:user.timezone} | 数据序列化和反序列化的时区设定， 默认使用客户端时区。  参数设置格式支持如下两种方式：  GMT+08:00, Asia/Shanghai。 |

* **示例**

|  |
| --- |
| CREATE INPUT STREAM S  (id INT, name STRING )  SERDE keyValueSerde  PROPERTIES ( "separator" = "," )  SOURCE KafkaInput  PROPERTIES (  "operator.kafka.groupid” = "gidkpi\_1\_1",  "operator.kafka.topic" = "simple\_1"  ); |

### CSVSerDe

* **功能**

CSV格式的序列化的反序列化。



Boolean类型的反序列化，如果字符串是true（不区分大小写），则转为true，如果字符串是false（不区分大小写），则转为false，除此之外，其他字符都是null。

整个列里面如果包含分隔符，则用引号将整个列包围；一个引号在CSV格式中用两个引号代替。

CSV格式举例

| 原始数据 | CSV格式 |
| --- | --- |
| 1,a,b | 1,a,b |
| 2,"a","b" | 2,"""a""","""b""" |
| 3,a,b,"a,b" | 3,"a,b","""a,b""" |

* **参数**

| 名称 | 默认值 | 参数说明 |
| --- | --- | --- |
| timezoneForTimestamp | ${system:user.timezone} | 数据序列化和反序列化的时区设定， 默认使用客户端时区。  参数设置格式支持如下两种方式：  GMT+08:00, Asia/Shanghai。 |

* **示例**

|  |
| --- |
| CREATE INPUT STREAM S  (id INT, name STRING )  SERDE CSVSerDe  SOURCE KafkaInput  PROPERTIES (  "operator.kafka.groupid” = "gidkpi\_1\_1",  "operator.kafka.topic" = "simple\_1"  ); |

### BinarySerDe

* **功能**

字节码的序列化和反序列化，多和TCP相关输入输出一起使用，可以按照长度将字节码中的每个属性取出。

每种数据类型都有其固定长度，如果小于这个固定长度，就会导致数据读取错误。

各数据类型Binary长度

| 数据类型 | 长度(单位：字节byte,一个字节长度为8位) |
| --- | --- |
| Boolean | 1 |
| Int | 4 |
| Long | 8 |
| Float | 4 |
| Double | 8 |
| Decimal | 最小长度5。 |
| Time | 字符串格式最小长度8，long类型8。 |
| Date | 字符串格式最小长度10，long类型8。 |
| Timestamp | 字符串格式最小长度19，long类型8。 |
| String | 最小1。 |



1、Boolean类型的反序列化，如果字符串是true（不区分大小写），则转为true，如果字符串是false（不区分大小写），则转为false，除此之外，其他字符都是null。

2、Int,long,flat,double等数据类型，采用的是有符号类型，和C有所不同，所以在跨平台访问的时候要注意正负关系。

* **参数**

二进制格式序列化反序列化参数

| 名称 | 默认值 | 参数说明 |
| --- | --- | --- |
| attributesLength | 无 | 每个属性字段长度。 |
| timeType | Long | 时间类型表示方法。  String或者Long类型，默认Long类型。  不区分大小写。  如果设置了该参数，会影响该算子内所有binaryserde的时间类型(date,time,timestamp)序列化、反序列化方式，对其他算子不会造成影响。 |
| decimalType | decimal | decimal类型表示方法。  String或者decimal类型，默认decimal类型。  不区分大小写。  如果设置了该参数，会影响该算子内所有binaryserde的decimal类型序列化、反序列化方式，对其他算子不会造成影响。 |
| timezoneForTimestamp | ${system:user.timezone} | 数据序列化和反序列化的时区设定， 默认使用客户端时区。  参数设置格式支持如下两种方式：  GMT+08:00, Asia/Shanghai。 |

* **示例**

|  |
| --- |
| CREATE INPUT STREAM call  (  CallingNumber STRING,  CalledNumber STRING,  InitTime LONG  )  SERDE BinarySerDe  PROPERTIES ( "attributesLength" = "26,26,8" )  SOURCE TCPClientInput  PROPERTIES ( "server" = "127.0.0.1", "port" = "9999" ); |

## 数据读取

### KafkaInput

* **功能**

从Kafka消息队列中读取数据的算子。

* **参数**

Kafka输入算子参数

| 名称 | 默认值 | 参数说明 |
| --- | --- | --- |
| operator.kafka.groupid | 无 | 分组Id，相当于客户端ID。 |
| operator.kafka.topic | 无 | 消息的分类，不同类型的消息必须使用不同的topic。 |
| operator.kafka.zookeepers | 默认配置同集群内Kafka使用的zookeeper地址 | kafka读取数据的zookeeper地址，地址加端口，多个之间用逗号分隔，比如：192.168.0.20:2181,192.168.0.21:2181/kafka。 |
| operator.kafka.zksessiontimeout | 20000 | zookeeper连接超时时间。 |
| operator.kafka.zksynctime | 20000 | kafka zookeeper同步时间参数。 |
| operator.kafka.messageserializerclass | kafka.serializer.StringEncoder | kafka消费者中获取到数据之后的序列化类。 |
| operator.kafka.read.frombeginning | false | 是否从topic开始位置读取数据，如果是false，则从kafkainput启动之后在kafka中找到的最新位置读起。 |

* **示例**

|  |
| --- |
| CREATE INPUT STREAM S  (id INT, name STRING )  SERDE simpleSerde  PROPERTIES ( separator = "," )  SOURCE KafkaInput  PROPERTIES (  "operator.kafka.groupid"= "gidkpi\_1\_1",  "operator.kafka.topic" = "simple\_1",  "operator.kafka.zookeepers" = "localhost:2181,10.0.0.21:2181"  ); |

### TCPClientInput

* **功能**

通过TCP客户端从远程服务端读取数据。

* **参数**

TCP客户端输入参数

| 名称 | 默认值 | 参数说明 |
| --- | --- | --- |
| server | 无 | TCP服务端的IP地址。 |
| port | 无 | TCP服务端监听端口。 |
| sessionTimeout | 2000 | TCP客户端连接超时时间，单位毫秒。 |

* **示例**

|  |
| --- |
| CREATE INPUT STREAM transaction  (  C1 LONG,  C2 STRING  )  SOURCE TCPClientInput  PROPERTIES (  "server" = "127.0.0.1",  "port" = "9999" ); |

### RamdomGen

* **功能**

生成随机数的输入源。

* **参数**

随机数生成算子参数

| 名称 | 默认值 | 参数说明 |
| --- | --- | --- |
| timeUnit | SECONDS | 时间单位，默认秒,配置参数，可以配置如下： MILLISECONDS，SECONDS，MINUTES，HOURS，DAYS。 |
| period | 1 | 时间周期，默认1秒，配置参数。 |
| eventNumPerPeriod | 1 | 每时间周期发送事件数据，默认1个。即发送数据 1个/秒，配置参数。 |
| isSchedule | False | 是否定期发送，配置参数 |
| totalNumber | 0 | 发送个数，如果为0表示个数无限，配置参数。 |
| delayTime | 0 | 延迟发送时间，如果为0表示立即发送，配置参数。 |

* **示例**

|  |
| --- |
| CREATE INPUT STREAM S  (id INT, name STRING, type INT)  SOURCE randomgen  PROPERTIES (  "timeUnit" = "SECONDS",  "period" = "1",  "eventNumPerPeriod" = "1",  "isSchedule" = "true",  "totalNumber" = "20000",  "delayTime" = "0"); |

## 数据写入

### KafkaOutput

* **功能**

将数据写入kafka消息队列的算子。

* **参数**

Kafka输出算子参数

| 名称 | 默认值 | 参数说明 |
| --- | --- | --- |
| operator.kafka.topic | 无 | 消息的分类，不同类型的消息必须使用不同的topic。 |
| operator.kafka.zookeepers | 默认配置同集群内Kafka使用的zookeeper地址。 | kafka读取数据的zookeeper地址，格式为地址加端口，多个地址之间用逗号分隔，比如：10.0.0.5:2181, 10.0.0.6:2181/kafka。 |
| operator.kafka.brokers | 默认配置同集群内Kafka使用的brokers地址。 | 消费者获取消息元信息(topics, partitions and replicas)的地址,配置格式是：host1:port1,host2:port2。 |
| operator.kafka.zksessiontimeout | 20000 | zookeeper连接超时时间。 |
| operator.kafka.zksynctime | 20000 | kafka zookeeper同步时间参数。 |
| operator.kafka.messageserializerclass | kafka.serializer.StringEncoder | kafka消费者中获取到数据之后的序列化类。 |

* **示例**

|  |
| --- |
| CREATE OUTPUT STREAM rs  (id STRING, name STRING)  SINK KafkaOutput  PROPERTIES (  "operator.kafka.topic" = "simple\_2",  "operator.kafka.zookeepers" = "10.0.0.12:2181",  "operator.kafka.brokers" = "10.0.0.12:9092"); |

### TCPClientOutput

* **功能**

通过TCP客户端将计算结果发送至指定服务端。

* **参数**

TCP客户端输出算子参数

| 名称 | 默认值 | 参数说明 |
| --- | --- | --- |
| server | 无 | TCP服务端的IP地址。 |
| port | 无 | TCP服务端监听端口。 |
| sessionTimeout | 2000 | TCP客户端连接超时时间，单位毫秒。 |

* **示例**

|  |
| --- |
| CREATE OUTPUT STREAM FilterHO  (  IMSI STRING,  EventID STRING,  HOCount INT  )  SINK TCPClientOutput  PROPERTIES(  "server" = "127.0.0.1",  "port" = "6001",  "sessionTimeout"='2000' ); |

### ConsoleOutput

* **功能**

将收到的数据直接在控制台打印，不输出。

* **参数**

控制台输出算子参数

| 名称 | 默认值 | 参数说明 |
| --- | --- | --- |
| printFrequence | 1 | 计数频率，默认每条都会打印，如果设置成10，那么就会每隔10条消息打印一次 |

* **示例**

|  |
| --- |
| CREATE OUTPUT STREAM FilterHO  (  IMSI STRING,  EventID STRING,  HOCount INT  )  SINK ConsoleOutput  PROPERTIES("printFrequence" = "10"); |

## 函数

函数可以使用在select子句，where子句，having中。目前group by，order by子句不支持使用函数。

系统会内嵌一些经常使用到的函数，如果用户有特殊需求，可以手工定义。

如果用于定义的函数名和系统内置函数名一致，则用户的函数会覆盖当前线程内的系统函数，但是不会对其他用户造成影响，进程结束之后，该覆盖效果也会消失。

### 字符处理函数

字符处理函数用于完成字符方面的处理。如：获取参数的长度、对参数进行类型转换、对参数进行拼接，以及对参数的部分内容进行替换等。

字符串处理函数

| 函数 | 返回值 | 描述 | 示例 |
| --- | --- | --- | --- |
| substr(string s, int from) | String | 从参数string中抽取子字符串。  from表示抽取的起始位置。  字符串索引计数从1开始。  from为0时，按1处理。  from为正数时，抽取从from到末尾的所有字符。  from为负数时，抽取字符串的后n个字符，n为from的绝对值。 | substr("abc",2) --bc  substr("abc",0) --abc  substr("abc",1) --abc  substr("abc",-1) --c |
| substr(string s, int from, int count) | String | 从参数string中抽取子字符串。  from表示抽取的起始位置。  字符串索引计数从1开始。  count表示抽取的子字符串长度。  from为0时，按1处理。  from为正数时，抽取从 from开始的count个字符。  from为负数时，抽取从倒数第n个开始的count 个字符，n为from 的绝对值。  count小于1时，返回空字符串。 | substr("abc",0,2) --ab  substr("abc",1,2) --ab  substr("abc",-1,1) --c  substr("abc",2,-1) --空字符串 |
| strlength(string s) | Int | 返回字符串的长度  如果字符串为null，返回0。 | strlength(null) --0  strlength("") --0  strlength("abc") --3 |
| trim(String s) | String | 移除字符串开始和结尾的特殊字符。  特殊字符指ASCII码值小于33的特殊字符，包含空格，制表符、退格、回车等等，详情可以参照ASCII码表。 | trim(null) --空字符串  trim("") --空字符串  trim(" ") --空字符串  trim("abc") --abc  trim("abc ") –abc  trim("abc") –abc  trim("abc\n") –abc 最后一位是回车符，trim截去该符号 |
| concat(String s1, String s2) | String | 字符串拼接，按照参数出现的顺序进行拼接，中间任何一个参数为null，都会导致所有结果为null。 | concat("a","b") –ab  concat("ab","") --ab  concat("ab",null) --null |
| concat(String s1, String s2, String s3) | String | 字符串拼接，按照参数出现的顺序进行拼接，中间任何一个参数为null，都会导致所有结果为null。 | concat("a","b","c") --abc  concat("ab","","c") --abc  concat("ab",null,"c") --null |
| concat(String s1, String s2, String s3, String s4) | String | 字符串拼接，按照参数出现的顺序进行拼接，中间任何一个参数为null，都会导致所有结果为null。 | concat("a","b","c","d") --abcd  concat("ab","","c","d") --abcd  concat("ab",null,"c","d") --null |
| concat(String s1, String s2, String s3, String s4, String s5) | String | 字符串拼接，按照参数出现的顺序进行拼接，中间任何一个参数为null，都会导致所有结果为null。 | concat("a","b","c","d","e") --abcde  concat("ab","","c","d","e") --abcde  concat("ab",null,"c","d","e") --null |
| concat(String s1, String s2, String s3, String s4, String s5, String s6) | String | 字符串拼接，按照参数出现的顺序进行拼接，中间任何一个参数为null，都会导致所有结果为null。 | concat("a","b","c","d","e","f") --abcdef  concat("ab","","c","d","e","f") --abcdef  concat("ab",null,"c","d","e","f") --null |
| concat(String s1, String s2, String s3, String s4, String s5, String s6, String s7) | String | 字符串拼接，按照参数出现的顺序进行拼接，中间任何一个参数为null，都会导致所有结果为null。 | concat("a","b","c","d","e","f","g") --abcdefg  concat("ab","","c","d","e","f","g") --abcdefg  concat("ab",null,"c","d","e","f","g") --null |
| upper(String s) | String | 字符串转大写。  语言环境为US。 | upper("abc") --ABC  upper("aBc") --ABC |
| lower(String s) | String | 字符串转小写。  语言环境为US。 | lower("ABC") --abc  lower("aBc") --abc |

### 时间处理函数

时间处理函数主要包含对时间数据的操作，包含加减以及获取年月日等操作。

时间处理函数

| 函数 | 返回值 | 描述 | 示例 |
| --- | --- | --- | --- |
| currenttimemillis() | Long | 返回以毫秒为单位的当前时间。  从1970-00-00 00:00:00.000开始的毫秒时间。 | currenttimemillis() –1414036004463  --可以通过 from\_unixtime函数对该结果进行验证，注意：from\_unixtime入参精确到秒；Currenttimemillis返回时间单位为毫秒。 |
| day(String dateString) | Int | 返回指定时间字符串的当前月所在天数。  时间字符串可以是date类型或者timestmap类型，具体格式同CQL数据类型一致。 | day("2013-02-28") --28  day("2013-02-29") --null  day("2013-02-29 09:00:01") --null  day("2013-02-28 09:00:01") --28  day("2013-02-28 09:00:01.0") –28  day("2013-02-28 09:00:01.0 +0800") --28  day("2013-02-28 09:00:01.0 -0800") --1 |
| dayofmonth(String dateString) | Int | 返回指定时间字符串的当前月所在天数。  时间字符串可以是date类型或者timestmap类型，具体格式同CQL数据类型一致。 | dayofmonth("2013-02-28") --28  dayofmonth("2013-02-29") --null  dayofmonth("2013-02-29 09:00:01") --null  dayofmonth("2013-02-28 09:00:01") --28  dayofmonth("2013-02-28 09:00:01.0") –28  dayofmonth("2013-02-28 09:00:01.0 +0800") --28  dayofmonth("2013-02-28 09:00:01.0 -0800") --1 |
| month(String dateString) | Int | 返回指定时间字符串的所在月。  时间字符串可以是date类型或者timestmap类型，具体格式同CQL数据类型一致。 | month("2013-02-28") --2  month("2013-02-29") --null  month("2013-02-29 09:00:01") --null  month("2013-02-28 09:00:01") --2  month("2013-02-28 09:00:01.0") –2  month("2013-02-28 09:00:01.0 +0800") --2  month("2013-02-28 09:00:01.0 -0800") --3 |
| year(String dateString) | Int | 返回指定时间字符串的所在年。  时间字符串可以是date类型或者timestmap类型，具体格式同CQL数据类型一致。 | year("2013-10-17 09:58:00.111") --2013  year("2013-10-17 09:58:00") –2013  year("2012-12-31 17:58:00.111 -0800") --2013  year("2013-10-17 25:62:00") --null  year("2013-00-17") --null  year("2013-10-17") --2013  year("09:58:00") --null |
| hour(String dateString) | Int | 返回指定时间字符串的所在小时。  时间字符串可以是time类型或者timestmap类型，具体格式同CQL数据类型一致。 | hour("2013-10-17 09:58:00.111") --9  hour("2013-10-17 09:58:00") –9  hour("2013-10-17 09:58:00.111 +0800") --9  hour("2013-10-17 09:58:00.111 -0800") --1  hour("09:58:00") --9  hour("24:58:00") --null  hour("00:58:00") --0  hour("2013-10-17") --null |
| minute(String dateString) | Int | 返回指定时间字符串的所在分钟。  时间字符串可以是time类型或者timestmap类型，具体格式同CQL数据类型一致。 | minute("2013-10-17 09:58:00.111") -- 58  minute("2013-10-17 09:58:00") – 58  minute("2013-10-17 09:58:00.111 +0030") --28  minute("2013-10-17 09:58:00.111 -0030") --28  minute("09:58:00") -- 58  minute("09:60:00") -- null  minute("09:00:00") -- 0  minute("2013-10-17") -- null |
| second(String dateString) | Int | 返回指定时间字符串的所在秒。  时间字符串可以是time类型或者timestmap类型，具体格式同CQL数据类型一致。 | second("2013-10-17 09:58:00.111") -- 0  second("2013-10-17 09:58:00") – 0  second("2013-10-17 09:58:01.111 +0800") -- 1  second("09:58:00") -- 0  second("24:58:60") -- null  second("23:59:59") -- 59  second("24:00:00") -- null  second("2013-10-17") -- null |
| weekofyear(String dateString) | Int | 返回指定时间字符串的所在年中的星期数。  时间字符串可以是date类型或者timestmap类型，具体格式同CQL数据类型一致。 | weekofyear("2013-01-01 09:58:00.111") -- 1  weekofyear("2013-01-01 09:58:00") – 1  weekofyear("2013-01-01 02:58:20 +0800") -- 1  weekofyear("2013-01-01 22:58:20 -0800") -- 1  weekofyear("2013-12-28") -- 52  weekofyear("2013-12-52") -- null  weekofyear("2013-01-01 09:58:70") -- null |
| weekofyear(Date date) | Int | 返回指定时间的所在年中的星期数。 | weekofyear(todate("2013-1-1")) -- 1  weekofyear(todate("2013-01-01")) -- 1  weekofyear(todate("2013-12-28")) -- 52  weekofyear(todate("2013-12-52")) -- null |
| weekofyear(TimeStamp timestamp) | Int | 返回指定时间的所在年中的星期数。 | weekofyear(totimestamp("2013-01-01 09:58:00.111")) – 1  weekofyear(totimestamp("2013-01-01 02:58:20 +0800")) -- 1  weekofyear(totimestamp("2013-01-01 22:58:20 -0800")) -- 1  weekofyear(totimestamp("2013-01-01 09:58:00")) -- 1  weekofyear(totimestamp("2013-01-01 09:58:70")) -- null |
| from\_unixtime(int unixTime, String format) | String | 时间格式化，转化时间为对应格式  时间参数精度为秒  时间格式可以为：  "yyyy-MM-dd HH:mm:ss"  "yyyy-MM-dd"  "HH:mm:ss"  "yyyy-MM-dd HH:mm:ss.SSS" | from\_unixtime(0, "yyyy-MM-dd") -- "1970-01-01" |
| from\_unixtime(long unixTime) | String | 时间格式化，转化时间为对应格式  时间参数精度为秒  默认时间格式为：  "yyyy-MM-dd HH:mm:ss"。 | from\_unixtime(0L) -- "1970-01-01 08:00:00"  from\_unixtime(1L) -- "1970-01-01 08:00:01"  from\_unixtime(100L) -- "1970-01-01 08:01:40"  from\_unixtime(-999999L) -- "1969-12-20 18:13:21" |
| from\_unixtime(long unixTime,String format) | String | 时间格式化，转化时间为对应格式  时间参数精度为秒  时间格式可以为：  "yyyy-MM-dd HH:mm:ss"  "yyyy-MM-dd"  "HH:mm:ss"  "yyyy-MM-dd HH:mm:ss.SSS" | from\_unixtime(0L, "yyyy-MM-dd") -- "1970-01-01"  from\_unixtime(0L, "HH:mm:ss") -- "08:00:00"  from\_unixtime(0L, "yyyy-MM-dd HH:mm:ss") -- "1970-01-01 08:00:00"  from\_unixtime(0L, "yyyy-MM-dd HH:mm:ss.SSS") -- "1970-01-01 08:00:00.000" |
| datediff(String dateString1,String dateString2) | Int | 比较两个时间之间相差的天数。  时间字符串可以是date类型或者timestmap类型，具体格式同CQL数据类型一致。输出数据格式和输入数据类型的字符串形式一致。 | datediff("2013-10-17 09:42:00", "2013-10-17 09:42:01") – 0  datediff("2013-10-17 00:00:00", "2013-10-16 23:59:59") -- 1  datediff("2013-10-17 09:42:00", "2013-10-16 09:42:01") -- 1  datediff("2013-10-17 09:42:00", "2013-10-16 09:41:01") – 1  datediff("2013-10-17 22:42:00 +0800", "2013-10-17 22:41:01 -0800") -- -1  datediff("2013-10-17 09:42:00", "2013-10-16 09:100:01") – null  datediff("2013-10-17", "2013-10-16 09:41:01") -- 1  datediff("2013-10-17", "2013-10-16") -- 1  datediff("2013-10-16", "2013-10-17") -- -1  datediff("2013-10-16", "2013") -- null |
| dateadd(String dateString,int days) | String | 给指定日期添加指定天数。  时间字符串可以是date类型或者timestmap类型，具体格式同CQL数据类型一致。  输出数据格式和输入数据类型的字符串形式一致。 | dateadd("2013-10-17 09:42:00.111", 1) -- "2013-10-18 09:42:00.111 +0800"  dateadd("2013-10-17 09:42:00", 1) -- "2013-10-18 09:42:00.000 +0800"  dateadd("2013-10-17 22:42:00 -0800", 1) -- "2013-10-19 14:42:00.000 +0800"  dateadd("2013-10-17", 1) -- "2013-10-18"  dateadd("2013-10-17", -1) -- "2013-10-16"  dateadd("2013-10-17 09:100:00",1) -- null  dateadd("2013-10-17 09:100:00",10000) -- null |
| dateadd(TimeStamp timestamp,int days) | String | 给指定日期添加指定天数。  输出数据格式同CQLTimestamp数据类型输出格式。 | dateadd(totimestamp("2013-10-17 09:42:00"), 1) -- "2013-10-18 09:42:00.111 +0800"  dateadd(totimestamp("2013-10-1709:42:00"), -1) -- "2013-10-16 09:42:00.000 +0800" |
| datesub(String dateString,int days) | String | 给指定日期减去指定天数。  输出数据格式和输入数据类型的字符串形式一致。 | datesub("2013-10-17 09:42:00.111", 1) -- "2013-10-16 09:42:00.111 +0800"  datesub("2013-10-17 09:42:00", 1)-- "2013-10-16 09:42:00.000 +0800"  datesub("2013-10-17 09:42:00-0800", 1) -- "2013-10-17 01:42:00.000 +0800"  datesub("2013-10-17", 1) --"2013-10-16"  datesub("2013-10-17", -1) --"2013-10-18"  datesub("2013-10-17 09:100:00",1)-- null  datesub("2013-10-17 09:100:00",10000) -- null |
| datesub(TimeStamp timestamp,int days) | String | 给指定日期减去指定天数。  返回Timestamp类型字符串输出格式。 | datesub(totimestamp("2013-10-17 09:42:00"), 1) -- "2013-10-16 09:42:00.000 +0800"  datesub(totimestamp("2013-10-1709:42:00"), -1) -- "2013-10-18 09:42:00.000 +0800" |

### 数学函数



系统不会检查所有的数学运算是否溢出，需要用户自己保证。

数学函数

| 函数 | 返回值 | 描述 | 示例 |
| --- | --- | --- | --- |
| abs(int number) | Int | 返回数字类型的绝对值。  说明：系统不会检查是否溢出，需要用户自己保证数据不会溢出。 | abs(1) -- 1  abs(-1) -- 1  abs(0) -- 0  abs(-0) -- 0  abs(2147483647) -- 2147483647  abs(-2147483648) -- 2147483648 |
| abs(long number) | Long | 返回数字类型的绝对值。  说明：系统不会检查是否溢出，需要用户自己保证数据不会溢出。 | abs(1L) -- 1  abs(-1L) -- 1  abs(0L) -- 0  abs(-0L) -- 0 |
| abs(float number) | float | 返回数字类型的绝对值。  说明：系统不会检查是否溢出，需要用户自己保证数据不会溢出。 | abs(1.0f) -- 1.0  abs(-1.0f) - 1.0  abs(0.0F) -- 0.0  abs(-0.0F) -- 0.0 |
| abs(double number) | double | 返回数字类型的绝对值。  说明：系统不会检查是否溢出，需要用户自己保证数据不会溢出。 | abs(1.0d) -- 1.0  abs(-1.0d) -- 1.0  abs(0.0D) -- 0.0  abs(-0.0D) -- 0.0 |
| abs(decimal number) | decimal | 返回数字类型的绝对值。  说明：系统不会检查是否溢出，需要用户自己保证数据不会溢出 | abs(1.0BD) -- 1.0  abs(-1.0BD) -- 1.0  abs(0.0BD) -- 0.0  abs(-0.0BD) -- 0.0 |

### 类型转换函数

类型转换函数函数

| 函数 | 返回值 | 描述 | 示例 |
| --- | --- | --- | --- |
| tostring(object obj) | String | 将传入的数据转为字符串格式。  支持任何数据类型。 | tostring(true) -- "true"  tostring(false) -- "false"  tostring(null) -- null  tostring(1) -- "1"  tostring(1L) -- "1"  tostring(1.0F) -- "1.0"  tostring(1.0D) -- "1.0"  tostring(toDecimal("1.0")) --"1.0"  tostring(toTime("15:40:00")) --"15:40:00"  tostring(toDate("2013-10-17")) --"2013-10-17"  tostring(toDate("2013-1-17")) --"2013-01-17"  tostring(toTimeStamp("2013-10-17 15:40:00.000000")) -- "2013-10-17 15:40:00.000 +0800"  tostring(toTimeStamp("2013-10-17 15:40:00")) -- "2013-10-17 15:40:00.000.000 +0800" |
| toint(string number)  toint(int number)  toint(long number)  toint(float number)  toint(double number)  toint(date number)  toint(time number)  toint(timestamp number)  toint(decimal number) | int | 将指定类型的数据转换为int类型。 | toint(1) -- 1  toint(1L) -- 1  toint(1.0f) -- 1  toint(1.4f) -- 1  toint(1.5f) -- 1  toint(1.6f) -- 1  toint(1.9f) -- 1  toint(1.9d) -- 1  toint("1") -- 1  toint("1.9") -- null  toint(toDecimal("1.9")) -- 1  toint(toDate("1970-01-01")) -- 0  toint(toTime("15:40:00")) -- 56400000  toint(toTimeStamp("1970-01-01 15:40:00.000")) -- 27600000 |
| tolong(string number)  tolong(int number)  tolong(long number)  tolong(float number)  tolong(double number)  tolong(date number)  tolong(time number)  tolong(timestamp number)  tolong(decimal number) | long | 将指定类型的数据转换为long类型。 | tolong(1) -- 1L  tolong(1L) -- 1L  tolong(1.0f) -- 1L  tolong(1.4f) -- 1L  tolong(1.5f) -- 1L  tolong(1.6f) -- 1L  tolong(1.9f) -- 1L  tolong(1.9d) -- 1L  tolong("1") -- 1L  tolong("1.9") -- null  tolong(toDecimal("1.9")) -- 1L  tolong(toDate("1970-01-01")) -- 0L  tolong(toTime("15:40:00")) -- 56400000L  tolong(toTimeStamp("1970-01-01 15:40:00.000")) -- 27600000L |
| tofloat(string number)  tofloat(int number)  tofloat(long number)  tofloat(float number)  tofloat(double number)  tofloat(date number)  tofloat(time number)  tofloat(timestamp number)  tofloat(decimal number) | float | 将指定类型的数据转换为float类型。 | tofloat(1) -- 1.0F  tofloat(1F) -- 1.0F  tofloat(1.0f) -- 1.0F  tofloat(1.4f) -- 1.4F  tofloat(1.5f) -- 1.5F  tofloat(1.6f) -- 1.6F  tofloat(1.9f) -- 1.9F  tofloat(1.9d) -- 1.9F  tofloat("1") -- 1.0F  tofloat("1.9") -- 1.9F  tofloat(toDecimal("1.9")) -- 1.9F  tofloat(toDate("1970-01-01")) -- 0.0F  tofloat(toTime("15:40:00")) -- 56400000.0F  tofloat(toTimeStamp("1970-01-01 15:40:00.000")) -- 27600000.0F |
| todouble(string number)  todouble(int number)  todouble(long number)  todouble(float number)  todouble(double number)  todouble(date number)  todouble(time number)  todouble(timestamp number)  todouble(decimal number) | double | 将指定类型的数据转换为double类型。 | todouble(1) -- 1.0D  todouble(1F) -- 1.0D  todouble(1.0f) -- 1.0D  todouble(1.9d) -- 1.9D  todouble("1") -- 1.0D  todouble("1.9") -- 1.9D  todouble(toDecimal("1.9")) --1.9D  todouble(toDate("1970-01-01")) --0.0D  todouble(toTime("15:40:00")) -- 56400000.0D  todouble(toTimeStamp("1970-01-01 15:40:00.000")) -- 27600000.0D |
| toboolean(string value)  toboolean(int value)  toboolean(long value)  toboolean(float value)  toboolean(double value)  toboolean(date value)  toboolean(time value)  toboolean(timestamp value)  toboolean(decimal value)  toboolean(boolean value) | boolean | 将指定类型的数据转换为boolean类型。  数字类型和时间类型，只要最后转为long类型的值不为0，就都返回true。  非空字符串都返回true。 | toboolean(0) -- false  toboolean(1) -- true  toboolean(0l) -- false  toboolean(1l) -- true  toboolean(0F) -- false  toboolean(1f) -- true  toboolean(0d) -- false  toboolean(1D) -- true);  toboolean(0.9d) -- false  toboolean(0.9f) -- false  toboolean(0BD) -- false  toboolean(0.0BD) -- false  toboolean(1BD) -- true  toboolean("") -- false  toboolean("0") -- true  toboolean("1") -- true  toboolean("true") -- true  toboolean("false") -- true  toboolean(totime("17:06:00")) --true  toboolean(todate("2013-10-17"))-- tru  toboolean(totimestamp("2013-10-17 17:06:00.000")) -- true |
| todate(string s) | date | 将指定类型的数据转换为date类型。  支持yyyy-[M]M-[d]d形式的字符串。 | todate("2014-09-25 17:07:00") -- null  todate("2014-09-25") – "2014-09-25"  todate("17:07:00") -- -- null |
| todate(string s, string format) | date | 将指定类型的数据转换为date类型。  时间格式可以为：  "yyyy-MM-dd HH:mm:ss"  "yyyy-MM-dd"  "yyyy-MM-dd HH:mm:ss.SSS" | todate("2014-09-25 17:07:00", "yyyy-MM-dd HH:mm:ss") -- "2014-09-25"  todate("2014-09-25 17:07:01", "yyyy-MM-dd HH:mm:ss") -- "2014-09-25"  todate("2014-09-25", "yyyy-MM-dd") -- "2014-09-25" |
| totime(string s) | time | 将指定类型的数据转换为time类型。 | totime("2014-09-25 17:07:00") -- null  totime("17:07:00") -- "17:07:00"  totime("17:07:90") -- null  totime("16:44:00") -- "16:44:00" |
| totimestamp(string s) | timestamp | 将指定类型的数据转换为timestamp类型。 | totimestamp("2014-09-25 17:07:00") -- "2014-09-25 17:07:00.000 +0800"  totimestamp("2014-09-25 17:07:00.056") -- "2014-09-25 17:07:00.056.000 +0800"  totimestamp("abc") -- null  totimestamp("2014-09-25 17:07:00.56") -- "2014-09-25 17:07:00.56.000 +0800"  totimestamp("2014-09-25 17:07:00.999")-- "2014-09-25 17:07:00.56.999 +0800"  totimestamp("2014-09-25 17:07:00.9999999999") -- null  totimestamp("2014-9-9 7:7:5") --"2014-09-09 07:07:05.000 +0800"  totimestamp("2014-9-9 7:7:5.9") -- "2014-09-09 07:07:05.009 +0800" |
| todecimal(string number)  todecimal(int number)  todecimal(long number)  todecimal(double number)  todecimal(float number) | decimal | 将指定类型的数据转换为decimal类型。 | todecimal(1) -- "1"  todecimal(1F) -- "1.0"  todecimal(1.0f) -- "1.0"  todecimal(1.9d) -- "1.9"  todecimal("1") -- "1.0"  todecimal("1.9") -- "1.9" |
| todecimal(long number, int scale) | decimal | 将指定类型的数据转换为decimal类型。  可以通过设置scale参数指定标度。 | todecimal(1L,2) -- "0.01"  todecimal(100000L,2) -- "1000.00" |

### 聚合函数

聚合处理类函数，有多个输入，一个输出，将多个值合并成一个输出。

UDAF函数

| 函数 | 返回值 | 描述 | 示例 |
| --- | --- | --- | --- |
| count(Object value) | long | 计算窗口内数据的数量，Null会被忽略，不会参与计算。 | Count(1)  Count(\*)  Count(id) |
| count(Object value, expression) | long | 计算窗口内满足表达式条件数据的数量。  Null会被忽略，不会参与计算。 | Count(id, type = "1") |
| sum(int number) | int | 计算窗口内数据的总和，数据列必须是int类型。  Null会被忽略，不会参与计算。 | Sum(intprice) |
| sum(long number) | long | 计算窗口内数据的总和，数据列必须是long类型。  Null会被忽略，不会参与计算。 | Sum(longprice) |
| sum(float number) | float | 计算窗口内数据的总和，数据列必须是float类型。  Null会被忽略，不会参与计算。 | Sum(floatprice) |
| sum(double number) | double | 计算窗口内数据的总和，数据列必须是double类型。  Null会被忽略，不会参与计算。 | Sum(doubleprice) |
| sum(decimal number) | Decimal | 计算窗口内数据的总和，数据列必须是decimal类型。  Null会被忽略，不会参与计算。 | Sum(decimalprice) |
| sum(int number, expression) | Int | 计算窗口内满足表达式条件数据的总和，数据列必须是int类型。  Null会被忽略，不会参与计算。 | Sum(intprice, type = "1") |
| sum(long number, expression) | long | 计算窗口内满足表达式条件数据的总和，数据列必须是long类型。  Null会被忽略，不会参与计算。 | Sum(longprice, type = "1") |
| sum(float number, expression) | float | 计算窗口内满足表达式条件数据的总和，数据列必须是float类型。  Null会被忽略，不会参与计算。 | Sum(floatprice, type = "1") |
| sum(double number, expression) | double | 计算窗口内满足表达式条件数据的总和，数据列必须是double类型。  Null会被忽略，不会参与计算。 | Sum(doubleprice, type = "1") |
| Sum(decimal number, expression) | Decimal | 计算窗口内满足表达式条件数据的总和，数据列必须是decimal类型。  Null会被忽略，不会参与计算。 | Sum(decimalprice, type = "1") |
| avg(int number) | long | 计算窗口内数据的平均值，数据列必须是int类型。  Null会被忽略，不会参与计算。 | avg(intprice) |
| avg(long number) | long | 计算窗口内数据的平均值，数据列必须是long类型。  Null会被忽略，不会参与计算。 | avg(longprice) |
| avg(float number) | double | 计算窗口内数据的平均值，数据列必须是float类型。  Null会被忽略，不会参与计算。 | avg(floatprice) |
| avg(double number) | double | 计算窗口内数据的平均值，数据列必须是double类型。  Null会被忽略，不会参与计算。 | avg(doubleprice) |
| avg(decimal number) | decimal | 计算窗口内数据的平均值，数据列必须是decimal类型。  Null会被忽略，不会参与计算。 | avg(decimalprice) |
| avg(int number, expression) | long | 计算窗口内满足表达式条件数据的平均值，数据列必须是int类型。  Null会被忽略，不会参与计算。 | avg(intprice, type = "1") |
| avg(long number, expression) | long | 计算窗口内满足表达式条件数据的平均值，数据列必须是long类型。  Null会被忽略，不会参与计算。 | avg(longprice, type = "1") |
| avg(float number, expression) | double | 计算窗口内满足表达式条件数据的总和，数据列必须是float类型。  Null会被忽略，不会参与计算。 | avg(floatprice, type = "1") |
| avg(double number, expression) | double | 计算窗口内满足表达式条件数据的平均值，数据列必须是double类型。  Null值会被忽略，不会计入平均值中。 | avg(doubleprice, type = "1") |
| avg(decimal number, expression) | decimal | 计算窗口内满足表达式条件数据的平均值，数据列必须是decimal类型。  Null值会被忽略，不会计入平均值中。 | avg(decimalprice, type = "1") |
| max(int number) | Int | 计算窗口内数据的最大值，数据列必须是int类型。  Null会被忽略，不会参与计算。 | max(intprice) |
| max(long number) | Long | 计算窗口内数据的最大值，数据列必须是long类型。  Null会被忽略，不会参与计算。 | max(longprice) |
| max(float number) | Float | 计算窗口内数据的最大值，数据列必须是float类型。  Null会被忽略，不会参与计算。 | max(floatprice) |
| max(double number) | Double | 计算窗口内数据的最大值，数据列必须是double类型。  Null会被忽略，不会参与计算。 | max(doubleprice) |
| max(decimal number) | Decimal | 计算窗口内数据的最大值，数据列必须是decimal类型。  Null会被忽略，不会参与计算。 | max(decimalprice) |
| max(int number, expression) | Int | 计算窗口内满足表达式条件数据的最大值，数据列必须是int类型。  Null会被忽略，不会参与计算。 | max(intprice, type = "1") |
| max(long number, expression) | Long | 计算窗口内满足表达式条件数据的最大值，数据列必须是long类型。  Null会被忽略，不会参与计算。 | max(longprice, type = "1") |
| max(float number, expression) | Float | 计算窗口内满足表达式条件数据的最大值，数据列必须是float类型。  Null会被忽略，不会参与计算。 | max(floatprice, type = "1") |
| max(double number, expression) | Double | 计算窗口内满足表达式条件数据的最大值，数据列必须是double类型。  Null会被忽略，不会参与计算。 | max(doubleprice, type = "1") |
| max(decimal number, expression) | Decimal | 计算窗口内满足表达式条件数据的最大值，数据列必须是decimal类型。  Null会被忽略，不会参与计算。 | max(decimalprice, type = "1") |
| min(int number) | Int | 计算窗口内数据的最小值，数据列必须是int类型。  Null会被忽略，不会参与计算。 | min(intprice) |
| min(long number) | Long | 计算窗口内数据的最小值，数据列必须是long类型。  Null会被忽略，不会参与计算。 | min(longprice) |
| min(float number) | Float | 计算窗口内数据的最小值，数据列必须是float类型。  Null会被忽略，不会参与计算。 | min(floatprice) |
| min(double number) | Double | 计算窗口内数据的最小值，数据列必须是double类型。  Null会被忽略，不会参与计算。 | min(doubleprice) |
| min(decimal number) | Decimal | 计算窗口内数据的最小值，数据列必须是decimal类型。  Null会被忽略，不会参与计算。 | min(decimalprice) |
| min(int number, expression) | Int | 计算窗口内满足表达式条件数据的最小值，数据列必须是int类型。  Null会被忽略，不会参与计算。 | min(intprice, type = "1") |
| min(long number, expression) | Long | 计算窗口内满足表达式条件数据的最小值，数据列必须是long类型。  Null会被忽略，不会参与计算。 | min(longprice, type = "1") |
| min(float number, expression) | Float | 计算窗口内满足表达式条件数据的最小值，数据列必须是float类型。  Null会被忽略，不会参与计算。 | min(floatprice, type = "1") |
| min(double number, expression) | Double | 计算窗口内满足表达式条件数据的最小值，数据列必须是double类型。  Null会被忽略，不会参与计算。 | min(doubleprice, type = "1") |
| min(decimal number, expression) | Decimal | 计算窗口内满足表达式条件数据的最小值，数据列必须是decimal类型。  Null会被忽略，不会参与计算。 | min(decimalprice, type = "1") |
| PREVIOUS(number,column) | 和Column数据类型一致 | 从窗口中取出数据列的前几次值，如果没有取到，则返回Null。  Number: previous前进个数，从0开始。0代表当前行，1代表上一行。  Column: 向前查询的列名称。 | Previous(0,columnName)代表当前行。  Previous(1,columnName)代表当前窗口中的当前数据的上一行数据ColumnName列。 |



Previous函数不支持Join，因为会导致结果不满足Join条件。

Previous函数不支持排序窗口，包含长度排序窗口和时间排序窗口，因为排序窗口中的数据会发生变化，导致窗口数据和缓存数据不一致。

## 数据源

### RDB数据源

* **功能**

从数据库中获取数据。

RDB数据源支持PreparedStatement，查询参数数据类型必须是CQL支持的数据类型。

用户可以将用户名或者密码进行加密，系统默认支持AES256加密，通过执行CQL客户端cqlEncrypt.sh脚本可以对用户名或者密码进行加密。

数据源参数分为定义参数和查询参数，定义参数是在create datasource语法中用到的参数，查询参数是在From子句中，正式查询的时候用到的参数。

RDB数据源的查询参数支持多个，第一个默认是查询SQL语句，之后都是查询参数。



CQL不包含任何数据库的JDBC驱动，需要用户通过add jar命令自己添加JDBC驱动程序。

* **参数**

数据源定义参数

| 名称 | 默认值 | 参数说明 |
| --- | --- | --- |
| driver | 无 | 数据库驱动。 |
| url | 无 | 数据库的URL，具体内容和数据库相关。 |
| userName | 无 | 数据库访问用户名。 |
| password | 无 | 数据库访问密码。 |
| decryptClass | com.huawei.streaming.encrypt. DefaultDecrypt | 数据解密实现，用户也可以自己实现com.huawei.streaming.encrypt. StreamingDecrypt 接口，通过add jar的方式生效。 |
| decryptType | ALL | 数据库连接参数加密方式  USER,PASSWORD,NONE,ALL。 |

表格 36 数据源查询参数

|  |  |  |
| --- | --- | --- |
| 名称 | 默认值 | 参数说明 |
| QueryString |  | 查询SQL语句，需要用到的变量用?的方式填充，CQL会自动根据参数定义顺序进行填充，同JDBC Prestatemnt接口定义 |
| queryArgs |  | 可变参数，参数可以0个或者多个，该参数会按照定义的顺序填充到queryString中。 |

* **示例**

|  |
| --- |
| add jar "/opt/software/Test/gsjdbc4-V100R003C10SPC050.jar";  create input stream S  (id int,name String)  SOURCE KafkaInput  PROPERTIES (  "operator.kafka.groupid" = "gidkpi\_1\_1",  "operator.kafka.topic" = "0915"  );  CREATE DATASOURCE ds  SOURCE RDBDataSource  PROPERTIES (  "driver" = "org.postgresql.Driver",  "url" = "jdbc:postgresql://127.0.0.0:1521/streaming",  "userName" = "55B5B07CF57318642D38F0CEE0666D26",  "password"="55B5B07CF57318642D38F0CEE0666D26"  );  create output stream rs  (type String, id String, tag String)  SINK consoleOutput;  insert into rs select "datasoruceColumn", rdb.id, rdb.name from S,  DATASOURCE ds  [  SCHEMA (id String, name String),  QUERY("select id, name from testtb where id = ?", s.id)  ] rdb; |

# CQL其他配置参数

这里的配置参数，是指没有在前面序列化/反序列化、输入输出以及UDF函数中出现过的配置参数，主要分为系统配置参数，平台相关配置参数等。这类参数都没有简写，在使用的时候，只能按照全称进行设置。

CQL其它配置参数

| 参数名称 | 默认值 | 参数说明 |
| --- | --- | --- |
| streaming.killapplication.overtime | 60 | CQL kill application命令执行的超时时间，单位：秒。 |
| streaming.common.istestmodel | False | 是否是测试模式。  如果是测试模式，系统就会只构建拓扑，但是并不提交。  该参数主要在CQL测试情况下使用，正常场景无需配置此参数。 |
| streaming.common.parallel.number | 1 | 默认的并发度，如果在CQL语句中没有体现，则使用系统默认值。 |
| streaming.adaptor.application | com.huawei.streaming.storm.StormApplication | 底层应用程序平台。 |
| streaming.template.directory | ${system:java.io.tmpdir} | 系统临时目录路径，默认值使用的是系统的tmp路径。 |
| streaming.storm.nimbus.host | 127.0.0.1 | nimbus的地址，在Storm HA场景下无效。 |
| streaming.storm.nimbus.port | 6627 | nimbus的端口，在Storm HA场景下无效。 |
| streaming.storm.ha.zkaddress | 127.0.0.1 | storm HA的zookeeper集群地址，如果有多个地址，使用','分隔。 |
| streaming.storm.ha.zkport | 24000 | storm HA的zookeeper端口。 |
| streaming.storm.ha.zksessiontimeout | 30000 | HA连接zookeeper的session 超时时间。 |
| streaming.storm.ha.zookeepersiontimeout | 30000 | HA连接zookeeper 建立连接的超时时间。 |
| streaming.storm.thrift.transport.plugin | backtype.storm.security.auth.SimpleTransportPlugin | 客户端和服务端连接的thrift协议，如果是安全连接，使用：  backtype.storm.security.auth.kerberos.KerberosSaslTransportPlugin。 |
| streaming.security.authentication | NONE | 是否启用安全。  可以使用如下几个值：NONE, KERBEROS。  不区分大小写。  默认值是：NONE 不启用安全。 |
| streaming.security.zookeeper.principal | zookeeper/hadoop | Zookeeper server端的principal。 |
| streaming.security.storm.principal | streaming/hadoop@HADOOP.COM | Storm server端的principal。 |
| streaming.security.user.principal | NONE | 用户principal。 |
| streaming.security.keytab.path | NONE | 用户keytab文件地址。 |
| streaming.security.krbconf.path | NONE | krb5文件地址。  如果为空，则从系统默认地址获取。  默认地址获取顺序为：  1、${JAVA\_HOME}/lib/security/krb5.conf  2、${JAVA\_HOME}/krb5.ini  3、windows: %windir%\krb5.ini  linux: /etc/krb5.conf |
| streaming.storm.submit.islocal | False | 任务提交方式，是否是本地提交，本地提交就意味着CQL任务在本地进程内执行，一般用来测试。 |
| streaming.localtask.alivetime.ms | 5000 | 本地任务存活时间，过了这个时间之后，就会被kill掉。单位毫秒 |
| streaming.storm.killtask.waitseconds | 3 | kill 任务的时候的等待时间，单位秒，Storm 自身kill任务的参数，不同于CQL参数。 |
| streaming.storm.worker.number | 1 | CQL提交任务的进程数量。 |
| streaming.serde.default | com.huawei.streaming.serde.SimpleSerDe | CQL默认的序列化/反序列化类。 |
| serde.simpleserde.separator | , | SimpleSerDe配置参数。  因为系统默认的序列化类为simpleSerDe，所以配置参数中必须包含该序列化类的参数。 |
| streaming.userfile.maxsize | 30 | 用户文件最大大小，单位MB。 |
| streaming.storm.rebalance.waitseconds | 10 | rebalance任务等待时间。 |
| streaming.operator.timezone | ${system:user.timezone} | CQL任务执行过程中的时区设定，默认使用客户端时区。  参数设置格式支持如下两种方式：  GMT+08:00, Asia/Shanghai。 |

# CQL异常码

CQL异常码的设计

异常码分为三部分：

前缀：以CQL-开头。

分类：两位数字编号，从00开头。

细分：三位数字编号，从000开头。

错误码 异常描述

CQL-00 成功完成。

CQL-01 警告。

CQL-02 配置属性异常。

CQL-03 流处理平台异常。

CQL-04 语义分析异常，包含语法解析异常。

CQL-05 拓扑异常，包含Streaming算子解析异常。

CQL-06 函数解析异常。

CQL-07 窗口解析异常。

CQL-08 安全异常。

CQL-99 未知异常。

CQL异常码列表

| 异常码 | 异常描述 |
| --- | --- |
| CQL-00： 成功完成。 | |
| CQL-00000 | 成功完成 |
| CQL-01： 警告。 | |
| CQL-01000 | 警告 |
| CQL-02： 配置属性异常。 | |
| CQL-02000 | 找不到配置属性。 |
| CQL-02001 | 配置属性数据类型转换失败。 |
| CQL-02002 | 错误的配置属性值，通常发生在数据类型正确，但是值超出范围或者不符合要求。 |
| CQL-03： 流处理平台异常。 | |
| CQL-03000 | 删除应用程序超时。  该时间取决于streaming.killapplication.overtime参数配置。 |
| CQL-03001 | 应用程序已经存在。 |
| CQL-03002 | 应用程序不存在。 |
| CQL-03003 | 应用程序的算子拓扑异常。 |
| CQL-03004 | 找不到输入算子。 |
| CQL-03005 | 应用程序状态异常，会导致执行active，deactive，rebalance等命令失败。 |
| CQL-03006 | 应用程序worker数量超过系统可用数量。 |
| CQL-03007 | 无效的nimbus服务异常。 |
| CQL-03008 | 服务端通信异常。 |
| CQL-04： 语义分析异常，包含语法解析异常。 | |
| CQL-04000 | 语法解析异常。 |
| CQL-04001 | 不支持的语法。 |
| CQL-04002 | order by 子句中只支持属性表达式。 |
| CQL-04003 | 表达式在select子句中找不到。 |
| CQL-04004 | 地址文件不存在。 |
| CQL-04005 | 地址必须是一个文件。 |
| CQL-04006 | 文件不是jar类型。 |
| CQL-04007 | Jar文件地址无法解析。 |
| CQL-04008 | 找不到对应流。 |
| CQL-04009 | 常量类型转换失败。 |
| CQL-04010 | 不支持的数据类型。 |
| CQL-04011 | 无法从指定流中找到对应列。 |
| CQL-04012 | 无法从所有的流中找到对应列。 |
| CQL-04013 | 无法唯一确定列， 该列可能存在在多个流中。 |
| CQL-04014 | 无法根据名称找到对应流。 |
| CQL-04015 | 输出列数量和select子句的列数量不一致。 |
| CQL-04016 | like表达式的子表达式必须是String类型。 |
| CQL-04017 | not表达式中所有的表达式返回值必须是boolean类型。 |
| CQL-04018 | IN和BETWEEN表达式中的参数只支持常量。 |
| CQL-04019 | CASE-WHEN表达式必须至少包含一个WHEN-THEN表达式。 |
| CQL-04020 | WHEN-THEN表达式必须同时包含WHEN和THEN表达式。 |
| CQL-04021 | WHEN-THEN表达式的WHEN表达式返回值必须是boolean类型。 |
| CQL-04022 | CASE表达式必须包含输入的表达式。 |
| CQL-04023 | 算术表达式只支持number的数据类型。 |
| CQL-04024 | 逻辑表达式中的表达式必须是boolean类型。 |
| CQL-04025 | 无法找到对应的数据源。 |
| CQL-04026 | 从数据源参数中找不到对应表达式。 |
| CQL-04027 | Combine操作中不允许出现窗口。 |
| CQL-04028 | Combine操作的输入流和combine条件中的流数量不匹配。 |
| CQL-04029 | combine算子只支持属性表达式。 |
| CQL-04030 | combine语句中，一个流的输出列在select语句中必须放在一起。 |
| CQL-04031 | MultiInsert中不允许出现数据源。 |
| CQL-04032 | MultiInsert中不允许出现GroupBy。 |
| CQL-04033 | MultiInsert中不允许出现窗口。 |
| CQL-04034 | MultiInsert中不允许出现多个输入流。 |
| CQL-04035 | 不支持多流Join。 |
| CQL-04036 | NATURAL JOIN 不支持。 |
| CQL-04037 | Join中，只允许其中一个流设置unidirection属性。 |
| CQL-04038 | join语句中找不到On条件。 |
| CQL-04039 | join的条件中的表达式在在输入流中找不到对应列。 |
| CQL-04040 | 不支持的Join条件。 |
| CQL-04041 | 找不到序列化和反序列化类。 |
| CQL-04042 | 找不到类。 |
| CQL-04043 | 序列化或者反序列化类 初始化失败。 |
| CQL-04044 | 流名称已经存在。 |
| CQL-04045 | 表达式比较异常，一般是两个表达式类型不匹配，无法比较。 |
| CQL-04046 | 用户文件总大小超过最大限制。  系统对于用户可以上传的总文件大小做了限制，取决于streaming.userfile.maxsize。  参数，默认30MB。 |
| CQL-04047 | 语法解析结果为空。 |
| CQL-04048 | 该窗口不支持Exclude Now功能。。 |
| CQL-04049 | Previous表达式不能和排序窗口一起用 |
| CQL-04050 | 该窗口不支持输出RStream(旧数据)输出  该场景一般发生在左流和右流一个窗口要输出新数据，一个窗口要输出旧数据。 |
| CQL-04051 | 从左流或者右流中找不到unidirection定义。  这个一般发生在selfjoin中，selfjoin要求必须单向Join。 |
| CQL-04052 | 找不到自定义算子。 |
| CQL-04053 | 自定义算子只允许一个输入schema。 |
| CQL-04054 | 自定义算子只允许一个输出schema。 |
| CQL-04055 | 自定义算子输入schema校验失败。 |
| CQL-04056 | 自定义算子输出schema校验失败。 |
| CQL\_40057 | 算子不匹配 |
| CQL-05： 拓扑异常，包含Streaming算子解析异常。 | |
| CQL-05000 | 一个CQL文件只允许提交一个应用程序。 |
| CQL-05001 | 算子找不到输入流。 |
| CQL-05002 | 算子找不到输出流。 |
| CQL-05003 | 应用程序名称为空。 |
| CQL-05004 | 找不到物理执行计划文件。 |
| CQL-05005 | 物理执行计划文件内容格式错误。 |
| CQL-05006 | 算子初始化失败。 |
| CQL-06： 函数解析异常。 | |
| CQL-06000 | 不支持的函数。 |
| CQL-06001 | 系统函数不能移除。 |
| CQL-06002 | 系统函数不能被覆盖。 |
| CQL-06003 | 用户自定义函数必须继承自指定类。 |
| CQL-06004 | previous函数只能是PREVIOUS(number, column)这样的形式。 |
| CQL-06005 | UDF函数中不支持的参数类型。 |
| CQL-07： 窗口解析异常。 | |
| CQL-07000 | 无法识别的窗口。 |
| CQL-07001 | exclude now中的select列表不能包含其他非group by列。 |
| CQL-07002 | 错误的窗口参数，该错误会导致窗口创建失败。 |
| CQL-07003 | 排序窗只允许滑动类型。 |
| CQL-08： 安全异常。 | |
| CQL-08000 | 不支持的安全类型。 |
| CQL-08001 | 没有授权。 |
| CQL-08002 | keytab文件地址错误。 |
| CQL-08003 | 安全内部错误。 |
| CQL-08004 | 客户端用户鉴权失败。 |
| CQL-99： 未知异常。 | |
| CQL-99000 | 服务端通用异常。 |
| CQL-99999 | 未知异常。 |

# CQL Java开发API说明

通过调用"com.huawei.streaming.cql.Driver"类进行CQL语句提交。

Driver方法说明：

CQL Java开发接口

| 方法名 | 方法说明 | 参数 | 返回值 |
| --- | --- | --- | --- |
| Run | 执行CQL语句 | String CQL语句 | Void |
| getResult | 获取执行结果 | 无 | CQLResult  查询结果根据语句有所不同，get命令返回查询参数的值。 |
| Clean | 清空driver中初始化的用户信息。  该方法调用一般发生在用户完成submit之后，或者重新提交 应用程序之前。 | 无 | Void |

# CQL关键字列表

CQL关键字列表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CREATE | SHOW | EXPLAIN | SET | GET |
| LOAD | EXPORT | DROP | ADD | SELECT |
| COMMENT | FORCE | SERDE | WITH | PROPERTIES |
| SOURCE | INPUT | STREAM | OUTPUT | SINK |
| SUBMIT | APPLICATION | DISTINCT | AND | OR |
| BETWEEN | IN | LIKE | RLIKE | REGEXP |
| CASE | WHEN | THEN | ELSE | END |
| CAST | EXISTS | IF | FALSE | AS |
| NULL | IS | TRUE | ALL | NOT |
| ASC | DESC | SORT | ORDER | GROUP |
| BY | HAVING | WHERE | FROM | ON |
| JOIN | FULL | PRESERVE | OUTER | CROSS |
| SEMI | LEFT | INNER | NATURAL | RIGHT |
| INTO | INSERT | OVERWRITE | LIMIT | UNION |
| APPLICATIONS | WINDOWS | EXTENDED | FUNCTIONS | FILE |
| INPATH | WINDOW | JAR | FUNCTION | COMBINE |
| UNIDIRECTION | PARALLEL | TRIGGER | PARTITION | SLIDE |
| BATCH | RANGE | ROWS | TODAY | UNBOUNDED |
| NOW | PREVIOUS | DATASOURCE | SCHEMA | QUERY |
| DEACTIVE | ACTIVE | WORKER | REBALANCE | DAYS |
| HOUR | HOURS | MINUTE | MINUTES | SECOND |
| SECONDS | MILLISECOND | MILLISECONDS | BOOLEAN | INT |
| LONG | FLOAT | DOUBLE | STRING | DATE |
| TIME | DECIMAL |  |  |  |