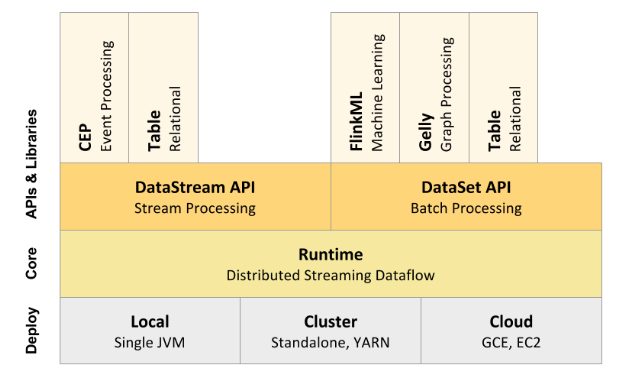
Flink SQL

## Table API

Flink Table API 是Fllink提供的类SQL的SDK API。以Library的形式提供，在流和批上都可以用此API开发业务。



|  |  |
| --- | --- |
| **Operators** | **Description** |
| **Select** | 类似于sql语句中的select  val in = ds.toTable(tableEnv, 'a, 'b, 'c);  val result = in.select('a, 'c as 'd); |
| **As** | Sql中的重命名  val in = ds.toTable(tableEnv).as('a, 'b, 'c); |
| **Where / Filter** | 类似于sql中的where条件选择  val in = ds.toTable(tableEnv, 'a, 'b, 'c);  val result = in.filter('a % 2 === 0);  val in = ds.toTable(tableEnv, 'a, 'b, 'c);  val result = in.where('b === "red"); |
| **GroupBy** | Similar to a SQL GROUPBY clause. Groups rows on the grouping keys, with a following aggregation operator to aggregate rows group-wise.  val in = ds.toTable(tableEnv, 'a, 'b, 'c);  val result = in.groupBy('a).select('a, 'b.sum as 'd); |
| **Join** | Similar to a SQL JOIN clause. Joins two tables. Both tables must have distinct field names and an equality join predicate must be defined using a where or filter operator.  val left = ds1.toTable(tableEnv, 'a, 'b, 'c);  val right = ds2.toTable(tableEnv, 'd, 'e, 'f);  val result = left.join(right).where('a === 'd).select('a, 'b, 'e); |
| **LeftOuterJoin** | Similar to a SQL LEFT OUTER JOIN clause. Joins two tables. Both tables must have distinct field names and at least one equality join predicate must be defined.  val left = tableEnv.fromDataSet(ds1, 'a, 'b, 'c)  val right = tableEnv.fromDataSet(ds2, 'd, 'e, 'f)  val result = left.leftOuterJoin(right, 'a === 'd).select('a, 'b, 'e) |
| **RightOuterJoin** | Similar to a SQL RIGHT OUTER JOIN clause. Joins two tables. Both tables must have distinct field names and at least one equality join predicate must be defined.  val left = tableEnv.fromDataSet(ds1, 'a, 'b, 'c)  val right = tableEnv.fromDataSet(ds2, 'd, 'e, 'f)  val result = left.rightOuterJoin(right, 'a === 'd).select('a, 'b, 'e) |
| **FullOuterJoin** | Similar to a SQL FULL OUTER JOIN clause. Joins two tables. Both tables must have distinct field names and at least one equality join predicate must be defined.  val left = tableEnv.fromDataSet(ds1, 'a, 'b, 'c)  val right = tableEnv.fromDataSet(ds2, 'd, 'e, 'f)  val result = left.fullOuterJoin(right, 'a === 'd).select('a, 'b, 'e) |
| **Union** | Similar to a SQL UNION clause. Unions two tables with duplicate records removed, both tables must have identical field types.  val left = ds1.toTable(tableEnv, 'a, 'b, 'c);  val right = ds2.toTable(tableEnv, 'a, 'b, 'c);  val result = left.union(right); |
| **UnionAll** | Similar to a SQL UNION ALL clause. Unions two tables, both tables must have identical field types.  val left = ds1.toTable(tableEnv, 'a, 'b, 'c);  val right = ds2.toTable(tableEnv, 'a, 'b, 'c);  val result = left.unionAll(right); |
| **Intersect** | Similar to a SQL INTERSECT clause. Intersect returns records that exist in both tables. If a record is present in one or both tables more than once, it is returned just once, i.e., the resulting table has no duplicate records. Both tables must have identical field types.  val left = ds1.toTable(tableEnv, 'a, 'b, 'c);  val right = ds2.toTable(tableEnv, 'e, 'f, 'g);  val result = left.intersect(right); |
| **IntersectAll** | Similar to a SQL INTERSECT ALL clause. IntersectAll returns records that exist in both tables. If a record is present in both tables more than once, it is returned as many times as it is present in both tables, i.e., the resulting table might have duplicate records. Both tables must have identical field types.  val left = ds1.toTable(tableEnv, 'a, 'b, 'c);  val right = ds2.toTable(tableEnv, 'e, 'f, 'g);  val result = left.intersectAll(right); |
| **Minus** | Similar to a SQL EXCEPT clause. Minus returns records from the left table that do not exist in the right table. Duplicate records in the left table are returned exactly once, i.e., duplicates are removed. Both tables must have identical field types.  val left = ds1.toTable(tableEnv, 'a, 'b, 'c);  val right = ds2.toTable(tableEnv, 'a, 'b, 'c);  val result = left.minus(right); |
| **MinusAll** | Similar to a SQL EXCEPT ALL clause. MinusAll returns the records that do not exist in the right table. A record that is present n times in the left table and m times in the right table is returned (n - m) times, i.e., as many duplicates as are present in the right table are removed. Both tables must have identical field types.  val left = ds1.toTable(tableEnv, 'a, 'b, 'c);  val right = ds2.toTable(tableEnv, 'a, 'b, 'c);  val result = left.minusAll(right); |
| **Distinct** | Similar to a SQL DISTINCT clause. Returns records with distinct value combinations.  val in = ds.toTable(tableEnv, 'a, 'b, 'c);  val result = in.distinct(); |
| **Order By** | Similar to a SQL ORDER BY clause. Returns records globally sorted across all parallel partitions.  val in = ds.toTable(tableEnv, 'a, 'b, 'c);  val result = in.orderBy('a.asc); |
| **Limit** | Similar to a SQL LIMIT clause. Limits a sorted result to a specified number of records from an offset position. Limit is technically part of the Order By operator and thus must be preceded by it.  val in = ds.toTable(tableEnv, 'a, 'b, 'c);  val result = in.orderBy('a.asc).limit(3); // returns unlimited number of records beginning with the 4th record or  val in = ds.toTable(tableEnv, 'a, 'b, 'c);  val result = in.orderBy('a.asc).limit(3, 5); // returns 5 records beginning with the 4th record |
| **Window** | Groups the records of a table by assigning them to windows defined by a time or row interval |

## SQL Query

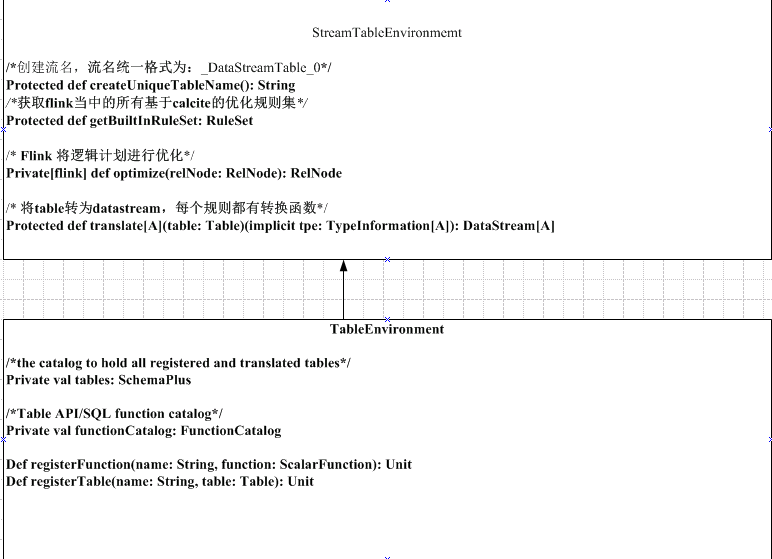
和Spark SQL类似，Flink SQL也可以采用SQL语句的方式执行，由于采用了Calcite的高性能开源框架，同时它还可以以拓展SQL的方式支持流式查询。支持window。

|  |
| --- |
| SELECT STREAM rowtime,  productId,  units,  SUM(units) OVER (ORDER BY rowtime RANGE INTERVAL '1' HOUR PRECEDING) unitsLastHour  FROM Orders; |

## Flink SQL 原理分析

几个重要的类：

1. TableEnvironment,StreamTableEnvironment,BatchTableEnvironment



TableEnvironment中包含

1. 存储calcite 定义的成员table；
2. 存储calcite自定义函数的成员functionCatalog

StreamTableEnvironment中包含基于流的calcite的优化规则

BatchTableEnvironment中包含基于批的calcite的优化规则

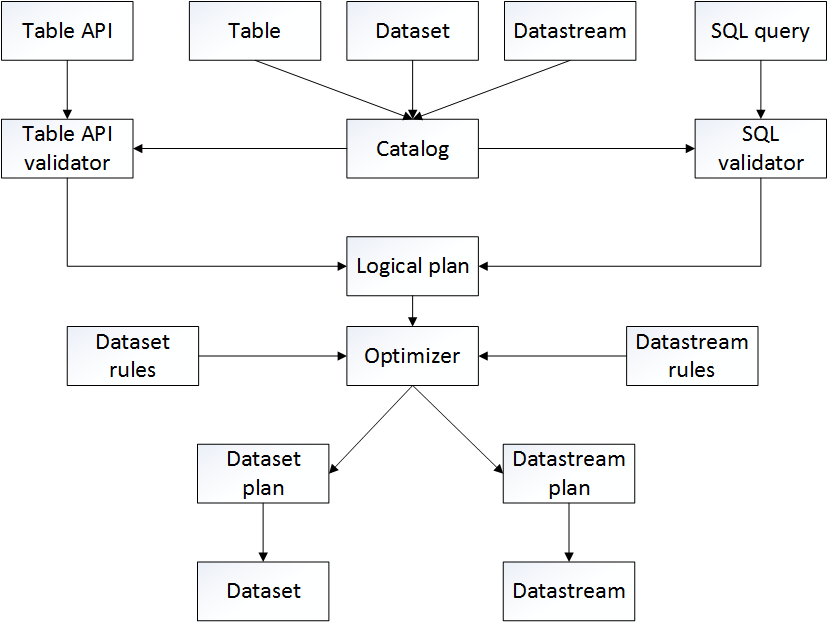
1. Table 类

|  |
| --- |
| class Table(  private[flink] val tableEnv: TableEnvironment,  private[flink] val logicalPlan: LogicalNode) |
| def select()…  def as()…  def join()…  def filter()… |

Table类中包含了

1. calcite的逻辑计划树
2. 公共语法函数

以下为Flink-SQL的执行原理图



***Table API 分析***

1. 获取Table Stream执行环境

|  |
| --- |
| val env = StreamExecutionEnvironment.getExecutionEnvironment  val tEnv = TableEnvironment.getTableEnvironment(env) |

1. 将输入的源数据转化成table

|  |
| --- |
| val orderA = env.fromCollection(Seq(  Order(1L, "beer", 3),  Order(1L, "diaper", 4),  Order(3L, "rubber", 2))).toTable(tEnv)  //toTable函数会将table元数据的定义注册到calcite中，并生成table类对象 |

1. 对table API的语法进行检查

|  |
| --- |
| val result: DataStream[Order] = orderB.select('user, 'product, 'amount)  .where('amount > 3)  .toDataStream[Order] |

1. 将table API的描述翻译成Calcite的逻辑计划树

|  |
| --- |
| LogicalFilter(condition=[>($2, 3)])  LogicalTableScan(table=[[\_DataStreamTable\_1]]) |

1. 对Calcite逻辑计划进行优化，分为dataset优化和datastream优化，生成优化过的calcite逻辑计划树
2. 优化后的逻辑计划树通过7个对应的转换规则，把calcite节点转化为7种flink节点，形成物理计划

|  |
| --- |
| DataStreamAggregate、DataStreamCalc、DataStreamScan、DataStreamUnion、DataStreamValues、DataStreamCorrelate、StreamTableSourceScan |

|  |
| --- |
| DataStreamCalc(select=[user, product, amount], where=[>(amount, 3)])  DataStreamScan(table=[[\_DataStreamTable\_1]]) |

1. Codegen，根据物理计划，每中Flink节点都包含自己的代码生成函数（tranlateToPlan），生成java代码
2. 最后将java代码转为datastream的公共算子，调度到flink进行执行

***SQL Query 分析***

首先看一个来自Flink 官方得到例子:

StreamSQLExample

|  |
| --- |
| **def** main(args: Array[String]): Unit = {   *// set up execution environment* **val** env = StreamExecutionEnvironment.getExecutionEnvironment  **val** tEnv = TableEnvironment.getTableEnvironment(env)   **val** orderA: DataStream[Order] = env.fromCollection(Seq(  Order(1L, **"beer"**, 3),  Order(1L, **"diaper"**, 4),  Order(3L, **"rubber"**, 2)))   **val** orderB: DataStream[Order] = env.fromCollection(Seq(  Order(2L, **"pen"**, 3),  Order(2L, **"rubber"**, 3),  Order(4L, **"beer"**, 1)))   *// register the DataStreams under the name "OrderA" and "OrderB"* tEnv.registerDataStream(**"OrderA"**, orderA, **'user**, **'product**, **'amount**)  tEnv.registerDataStream(**"OrderB"**, orderB, **'user**, **'product**, **'amount**)   *// union the two tables* **val** result = tEnv.sql(  **"SELECT \* FROM OrderA WHERE amount > 2 UNION ALL "** +  **"SELECT \* FROM OrderB WHERE amount < 2"**)   result.toDataStream[Order].print()   env.execute() } |

SQL语句方式的步骤类似，将其分为3步：

1. 获取Table Stream执行环境
2. 将源数据注册到tableEnvironments中，其中有一个catalog来记录注册的表名和结构
3. 执行sql函数解析SQL生成Table对象
4. result.toDataStream[Order].print()这句代码通过隐式转换，调用toDataStream[Order]函数，这个就是Flink支持Streaming SQL的入口

从入口函数toDataStream[Order]进去，可以看到Flink通过calcite的SQL最优化生成一个关系表达式对象RelNode，再进一步把calcite的这个表达式翻译成Flink可以执行的函数对象。

实际的过程是：

Source =>

DataStream[Any] =>

SQL Query -> RelNode -> translateToPlan() -> codegen() -> function对象 =>

DataStream[Any]包含function对象

其中有三个过程：

1. 根据源生成一个数据流对象（代表尚未执行的实际的数据流）
2. 根据输入的SQL，做一系列的处理：查询优化、翻译calcite的SQL处理过程为Java源代码、源代码编译成字节码、生成Flink transform中的转换函数对象
3. 把转换对象分装到DataStream中

简言之，就是把Calcite用SQL表达的事情，翻译成Flink框架支持的函数（前面这些都是定义lazy的尚未执行的过程），然后运行Flink作业。

其中第二个过程是重点，又可以分为三个阶段：

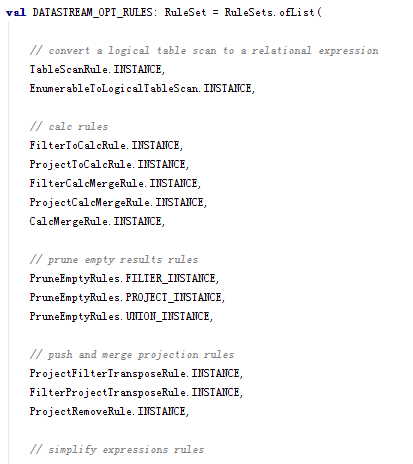
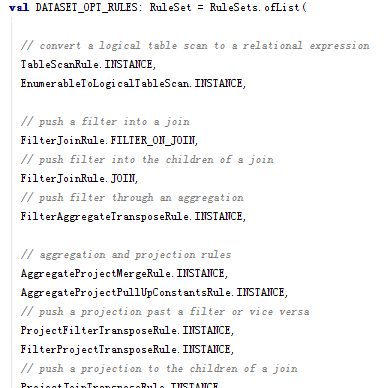
1. SQL解析，验证，生成关系树

|  |
| --- |
| **def** sql(query: String): Table = {  **val** planner = **new** FlinkPlannerImpl(getFrameworkConfig, getPlanner, getTypeFactory)  *// parse the sql query* **val** parsed = planner.parse(query)  *// validate the sql query* **val** validated = planner.validate(parsed)  *// transform to a relational tree* **val** relational = planner.rel(validated)   **new** Table(**this**, LogicalRelNode(relational.rel)) } |

首先生成默认的Calcite配置对象planner，然后把文本的SQL语句用Calcite解析成SQLNode对象parsed，再做SQL验证成SQLNode对象validated，最后转化成一个关系树RelRoot对象relational，即根对象。最后把关系树对象封装为Flink的Table表对象。注意，此阶段没有进行优化。

1. SQL优化

当我们执行toDataStream/toDataSet时，首先会调用optimize来优化逻辑计划，优化规则为FlinkRuleSets，分为DATASET\_OPT\_RULES(dataset)和DATASTREAM\_OPT\_RULES (datastream)两大类（51,25）



1. 翻译Flink逻辑计划为转换函数

将逻辑计划转换为Flink可执行算子,codegen

同样对应几条规则，即7个转化对

|  |
| --- |
| DataStreamAggregateRule.INSTANCE, DataStreamCalcRule.INSTANCE, DataStreamScanRule.INSTANCE, DataStreamUnionRule.INSTANCE, DataStreamValuesRule.INSTANCE, DataStreamCorrelateRule.INSTANCE, StreamTableSourceScanRule.INSTANCE, |

这些对象的核心函数是:translateToPlan()，顾名思义是翻译逻辑计划为Flink可执行算子。

上面7位主要是把逻辑计划翻译成Flink可执行算子，即：把SQL过程翻译成flink的转换函数。