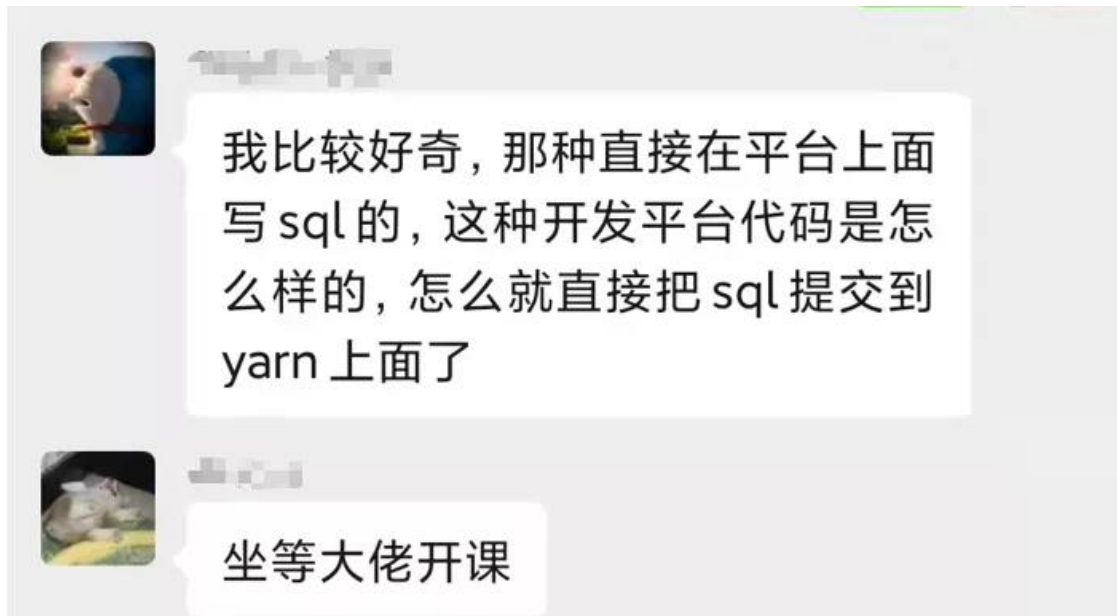


前言

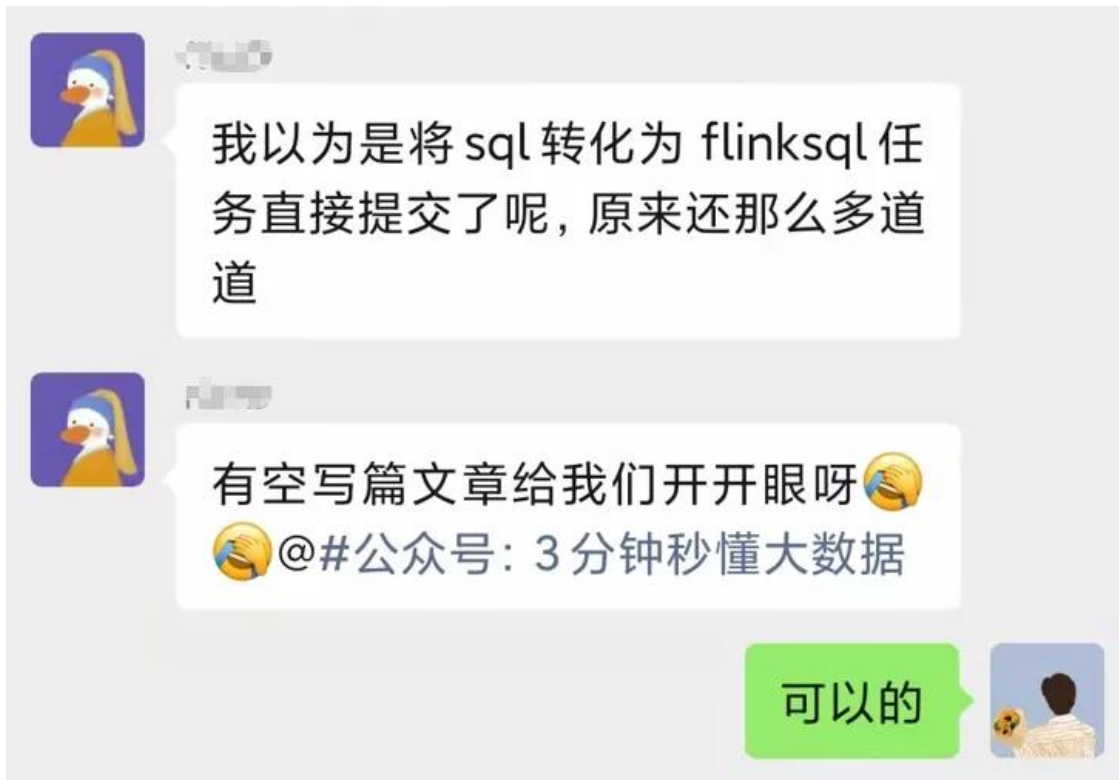
大家好，我是土哥。

这已经是我为读者写的第 21 篇 **Flink** 系列文章了。

上周有粉丝在群里问，在流计算平台编写完 **Flink sql** 后，为什么通过一键提交按钮，就可以将 **sql** 提交到 **yarn** 集群上面了？



由于现在各大厂对业务分层特别清晰，平台方向和底层技术开发会被单独划分，所以好多大数据同学编写完 **Flink Sql** 后，只需通过提交按钮将其提交到集群上，对背后的提交原理也许不太清楚。



下面土哥将为大家揭开这层神秘的面纱，挖掘 Flink Sql 背后的提交原理和源码设计。（硬核文章，建议收藏！）

熟悉平台

故事

小笨猪阿土刚入职某大数据公司担任实习生，然后主管交给阿土一个任务，让其熟悉公司的 **Flink** 流计算平台。



小笨猪阿土

阿土登录流计算平台后，看到平台上面可以编写 Sql 语法，于是就写了一个简单的 sql。



他发现旁边有个**校验功能**、于是就点击了一下，这时平台弹出 SQL 语法校验正确。阿土心中暗自自喜，看来我的 sql 功底还是不错嘛。

SQL 语法校验完成后，阿土点击提交按钮，流计算平台提示，SQL 语法校验正确，已成功提交集群。

Flink sql 代码居然提交到 yarn 集群上了???



应用id	应用名称	主机	端口	clusterId	启动时间	应用状态
flink_app_1631590902264_0003	lystest	10.3.68.204	35247	application_1631425049177_0023	2021-09-14 11:43:43	运行中

小笨猪阿土感到很惊讶，**sql** 就这样直接提交到集群了哇，这时候小笨猪的导师猴哥过来了，看到小笨猪的操作后，表扬了几句。



阿土，完成的不错啊，已经可以提交 sql 代码啦。但是你可别小看这简单的提交，这背后的门路可不浅哟。

这样吧，你好好探索一下这个 **sql** 提交的原理，然后写一篇分析报告，在咱们组分享一下。

啊.....啊.....

小笨猪阿土听到猴哥的要求后，一下就蔫了。从此之后，阿土就和 **Flink sql** 走在了一起。

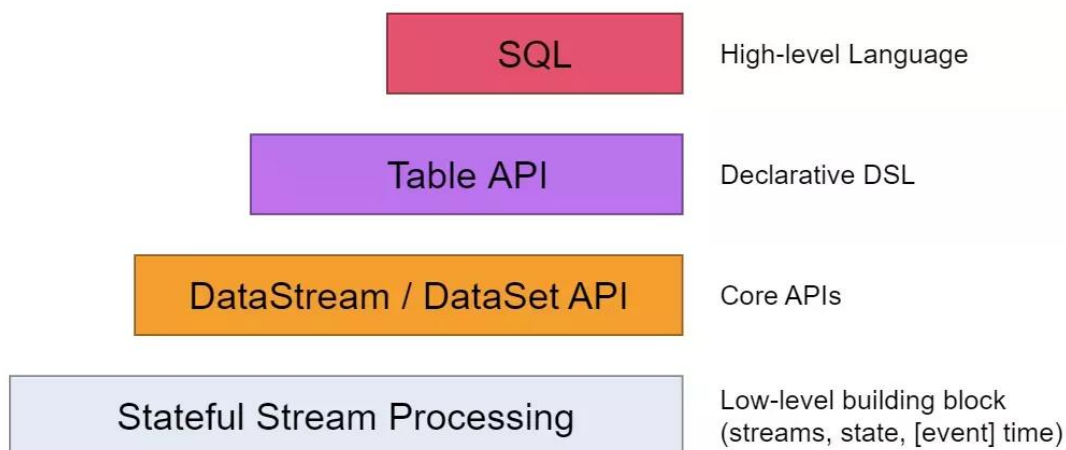
刚开始阿土很懵，于是就开始搜查 **Flink sql** 相关文章，过了几天，终于理清了一些思路。小笨猪将其流程总结为以下几个点：

1. **Flink Sql** 解析器
2. **Flink Planner** 和 **Blink Planner**
3. **Blink Sql** 提交流程

1. **Flink Sql** 解析器

1.1、了解 **Calcite**

为方便用户使用 **Flink** 流计算组件，**Flink** 社区设计了四种抽象，在这些抽象中，**Sql API** 属于 **Flink** 的最上层抽象，是 **Flink** 的一等公民，这就方便用户或者开发者直接通过 **Sql** 编写来提交任务。



但经过阿土的调查后发现，**Flink sql** 在提交任务时，并不是向 **DataStream API** 那样，直接被转为 **StreamGraph**，经过优化生成 **JobGraph** 提交到集群的,而是需要对编写的 **Sql** 进行解析、验证、优化等操作，在这中间，社区引入了一个强大的解析器，那就是 **Calcite**。

阿土好好调研了一番 **Calcite**

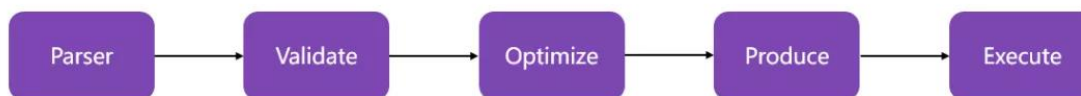
Calcite 属于 Apache 旗下的一个动态数据管理框架，具备很多数据库管理系统的功能，它可以对 SQL 进行 **SQL 解析**，**SQL 校验**，**SQL 查询优化**，**SQL 生成**以及数据连接查询等操作，它不存储元数据和基本数据，不包含处理数据的算法。而是作为一个中介的角色，**将上层 SQL 和底层处理引擎打通**，将其 SQL 转为底层处理引擎需要的数据格式。

它不受上层编程语言的限制，前端可以使用 SQL、Pig、Cascading 等语言，只要通过 Calcite 提供的 SQL Api 将它们转化成关系代数的抽象语法树即可，并根据一定的规则和成本对抽象语法树进行优化，最后推给各个数据处理引擎来执行。

所以 Calcite 不涉及物理规划层，它通过扩展适配器来连接多种后端的数据源和数据处理引擎，如 Hive，Drill，Flink，Phoenix 等。

1.2、Calcite 执行步骤

小笨猪阿土简单画了一下 Calcite 的执行流程，主要涉及 5 个部分 SQL 解析、SQL 校验、SQL 查询优化、SQL 生成、执行等。

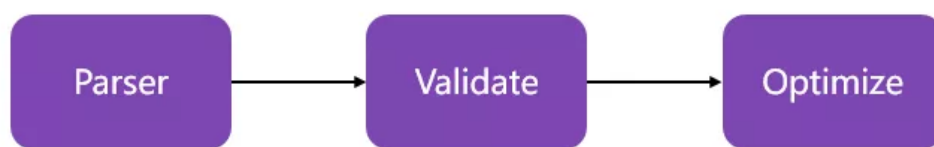


Calcite执行流程

在这个流程中，Calcite 各阶段扮演的角色如下：

1. SQL 解析。通过 JavaCC 实现，使用 JavaCC 编写 SQL 语法描述文件，将 SQL 解析成未经校验的 AST 语法树。
2. SQL 校验。通过与元数据结合验证 SQL 中的 Schema、Field、Function 是否存在，输入输出类型是否匹配等。
3. SQL 优化。对上个步骤的输出(RelNode ，逻辑计划树)进行优化，使用两种规则：基于规则优化 和 基于代价优化，得到优化后的物理执行计划。
4. SQL 生成。将物理执行计划生成为在特定平台/引擎的可执行程序，如生成符合 MySQL 或 Oracle 等不同平台规则的 SQL 查询语句等。
5. 执行。执行是通过各个执行平台执行查询，得到输出结果。

其中，Calcite 再与其他处理引擎结合时，到 SQL 优化阶段就已经结束。所以流程图简化为：



Calcite执行流程

2. Flink Planner 和 Blink Planner

阿土看完 Calcite 的原理后，开始想，那 Calcite 是怎么在 Flink 中扮演的角色呢？

这时猴哥过来给阿土说，单纯的看一些理论文章，是搞不清楚底层设计实现的，阿土啊，你可以看看源码。

听了猴哥的一番话后，阿土开始啃起了 Flink1.13.2 的 Flink Sql 源码

2.1 Flink Planner 和 Blink Planner

在 1.9.0 版本以前，社区使用 Flink Planner 作为查询处理器，通过与 Calcite 进行连接，为 Table/SQL API 提供完整的解析、优化和执行环境，使其 SQL 被转为 DataStream API 的 Transformation，然后再经过 StreamJraph -> JobGraph -> ExecutionGraph 等一系列流程，最终被提交到集群。

在 1.9.0 版本，社区引入阿里巴巴的 Blink，对 Flink Table & SQL 模块做了重大的重构，保留了 Flink Planner 的同时，引入了 Blink Planner，没引入以前，Flink 没考虑流批作业统一，针对流批作业，底层实现两套代码，引入后，基于流批一体理念，重新设计算子，以流为核心，流作业和批作业最终都会被转为 transformation。

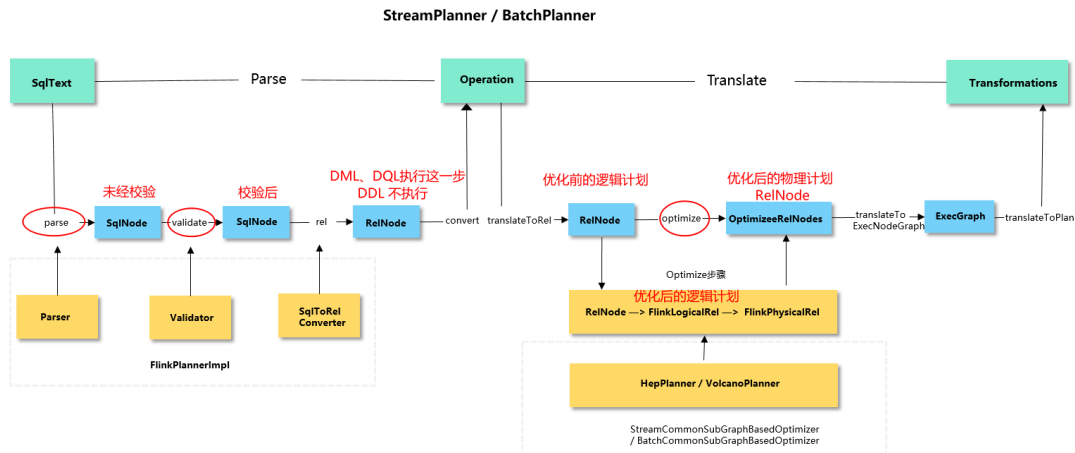
2.2 Blink Planner 与 Calcite 关系

在之后的版本，为了实现 Flink 流批一体的愿景，通过 Blink Planner 与 Calcite 进行对接,对接流程如下：

1. 在 Table/SQL 编写完成后，通过 Calcite 中的 parse、validate、rel 阶段，以及 Blink 额外添加的 convert 阶段,将其先转为 Operation；
2. 通过 Blink Planner 的 translateToRel、optimize、translateToExecNodeGraph 和 translateToPlan 四个阶段，将 Operation 转换成 DataStream API 的 Transformation；

- 再经过 StreamJraph -> JobGraph -> ExecutionGraph 等一系列流程，SQL 最终被提交到集群。

小笨猪根据查询后的资料以及查看 Flink 1.13.2 版本源码后，画出如下 SQL 执行流程图。



3. Blink Sql 提交流程（源码分析）

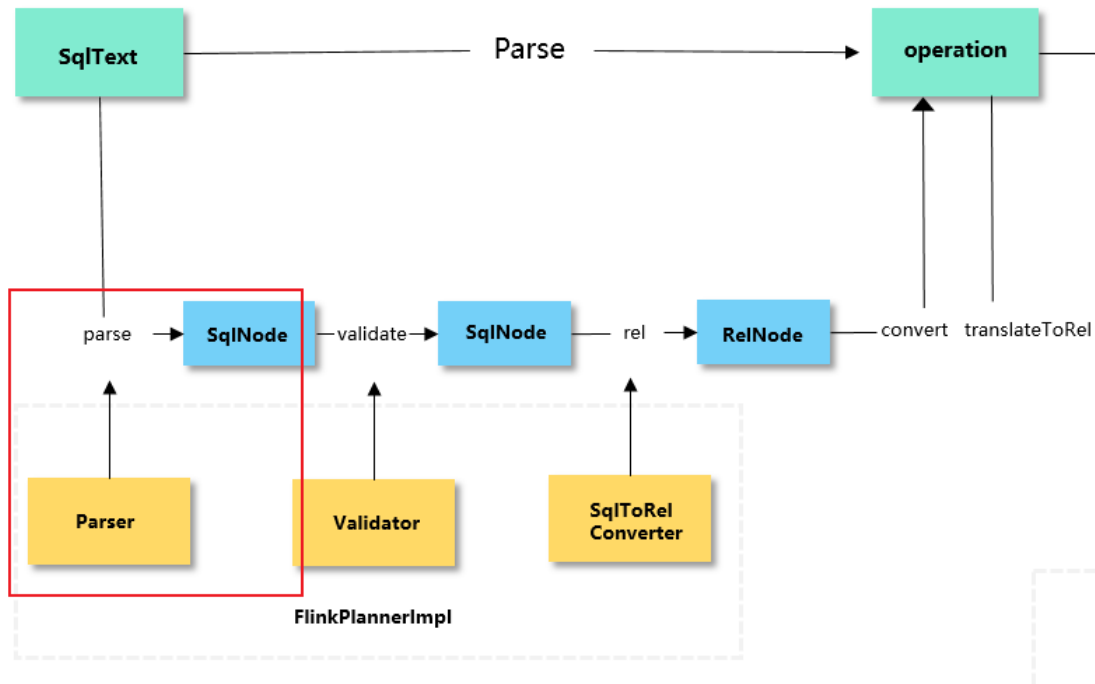
阿土根据对源码的分析后，发现无论是 Flink SQL 执行 DDL 操作、还是 DQL 操作或者 DML 操作、最终都可以将其总结为两个阶段：

SQL 语句到 Operation 过程，即 Parse 阶段； Operation 到 Transformations 过程，即 Translate 阶段。

3.1、Parse 阶段

在 Parse 阶段一共包含 parse、validate、rel、convert 部分

1、parse: SqlText => SqlNode



Calcite 的 parse 解析模块是基于 javacc 实现的。javacc 是一个词法分析生成器和语法分析生成器。词法分析器于将输入字符流解析成一个一个的 token，以下面这段 SQL 语句为例：

示例 1：

```
SELECT id, CAST(score AS INT), 'hello' FROM T WHERE id < 10
```

在 parse 部分，上面的 SQL 语句最后会被解析为如下一组 token：

```
SELECT `id`, `CAST`(`score` AS `INT`)\n, `hello` `FROM` `T`\nWHERE `id` < `10`
```

接下来语法分析器会以词法分析器解析出来的 **token** 序列作为输入来进行语法分析。分析过程使用递归下降语法解析，LL(k)。

其中，第一个 L 表示从左到右扫描输入；第二个 L 表示每次都进行最左推导(在推导语法树的过程中每次都替换句型中最左的非终结符为终结符。类似还有最右推导)；

k 表示的是每次向前探索(lookahead)k 个终结符。

分析所依赖的词法法则定义在一个 **parser.jj** 文件中。

```

/**
 * Parses a leaf SELECT expression without ORDER BY.
 */
SqlSelect SqlSelect() :
{
    final List<SqlLiteral> keywords = new ArrayList<SqlLiteral>();
    final SqlNodeList keywordList;
    List<SqlNode> selectList;
    final SqlNode fromClause;
    final SqlNode where;
    final SqlNodeList groupBy;
    final SqlNode having;
    final SqlNodeList windowDecls;
    final Span s;
}
{
    <SELECT>
    {
        s = span();
    }
    SqlSelectKeywords(keywords)
    (
        <STREAM> {
            keywords.add(SqlSelectKeyword.STREAM.symbol(getPos()));
        }
    )?
    (
        <DISTINCT> {
            keywords.add(SqlSelectKeyword.DISTINCT.symbol(getPos()));
        }
        |
        <ALL> {
            keywords.add(SqlSelectKeyword.ALL.symbol(getPos()));
        }
    )?
    {
        keywordList = new SqlNodeList(keywords, s.addAll(keywords).pos());
    }
    selectList = SelectList()
    (
        <FROM> fromClause = FromClause()
        where = WhereOpt()
        groupBy = GroupByOpt()
        having = HavingOpt()
        windowDecls = WindowOpt()
        |
        E() {
            fromClause = null;
            where = null;
        }
    )
}

```

在经过词法分析和语法分析后，一段 SQL 语句会被解析成一颗抽象语法树（Abstract Syntax Tree, AST），树的节点类型在 Calcite 中以 `SqlNode` 来表示，不同节点以不同子类型的 `SqlNode` 来表示。

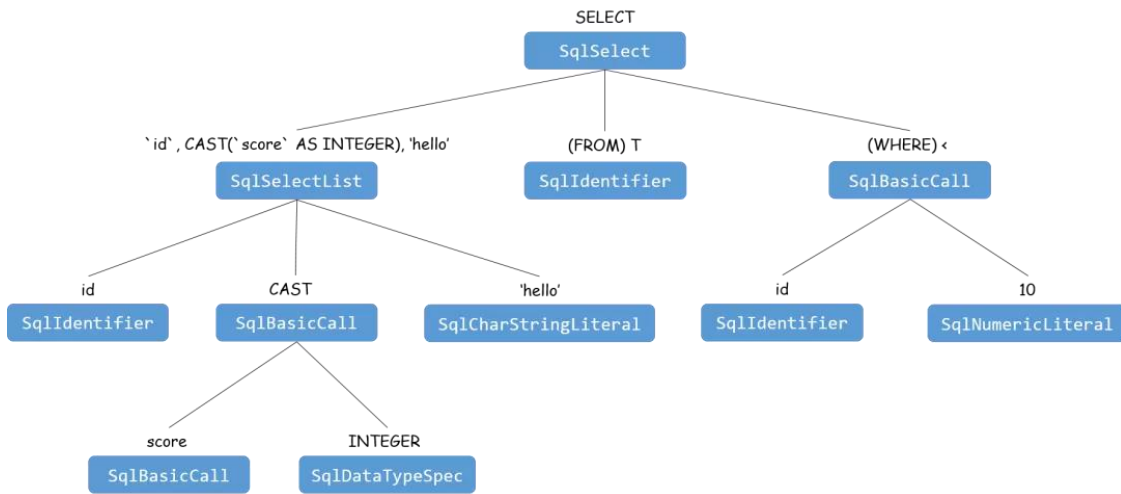
同样以上面的 SQL 为例，在这段 SQL 中：

1. `id`, `score`, `T` 等为 `SqlIdentifier`，表示一个字段名或表名的标识符；
2. `select` 和 `cast()` 为 `SqlCall`，表示一个行为或动作，其中 `cast()` 为一个 `SqlBasicCall`，表示一个函数调用，具体调用的是什么函数，由其内部的

SqlOperator 决定，比如这里是一个二元操作符“<”，对应 SqlBinaryOperator，operator 的名字是“<”，类别是 SqlKind.LESS_THAN;

3. int 为 SqlDataTypeSpec，表示一个类型定义;
4. 'hello'和 10 为 SqlLiteral，表示一个常量;

在 Calcite 中，所有的操作都是一个 SqlCall, 如查询是一个 SqlSelect, 删除是一个 SqlDelete 等，它们都是 SqlCall 的子类型。select 的查询条件等为 SqlCall 中的参数。示例 1 的 SQL 语句最终生成的语法树形式如下：



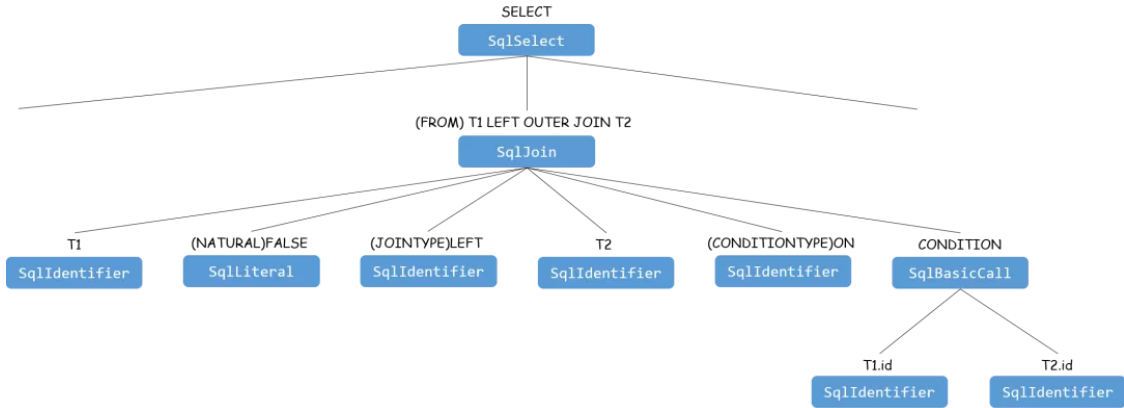
如果把示例 1 中的直接从一个表查询数据，改为从两张表的关联结果中查询数据，例如：

示例 2：

```

SELECT id, CAST(score AS INT), 'hello' FROM T1
LEFT OUTER JOIN T2 ON T1.id = T2.id WHERE T1.id < 10
    
```

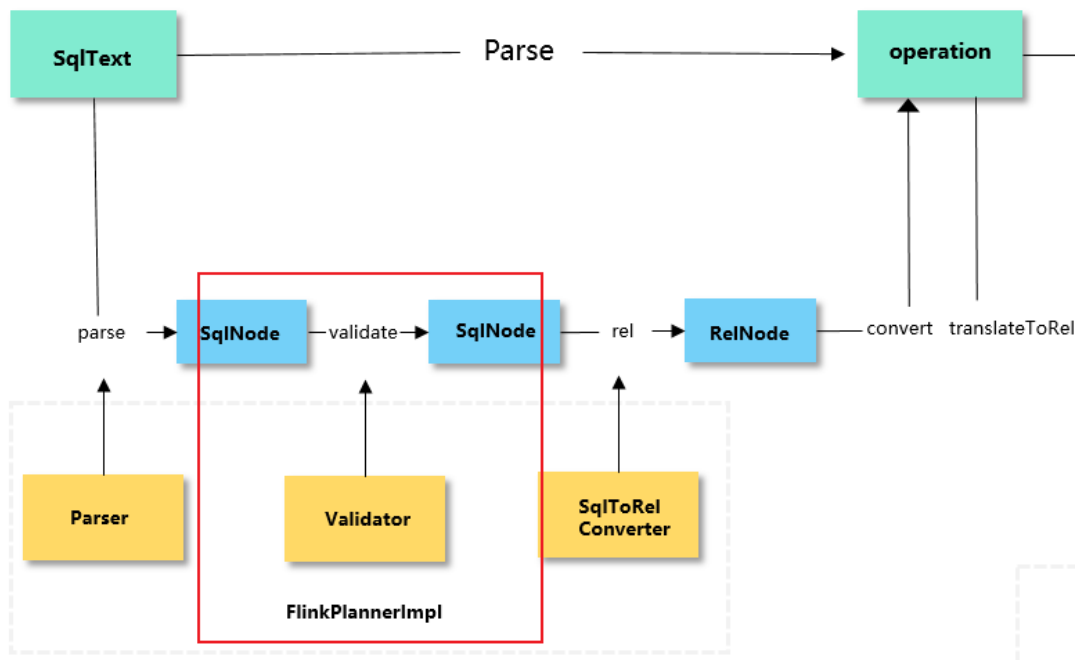
则相应的 AST 形式如下：



其中只有 FROM 子树部分由原来的 SqlIdentifier 节点变成了一棵 SqlJoin 子树，其他部分与示例 1 相同所以在图中省略了。

2、validata: SqlNode => SqlNode

校验（validate）阶段



对经过 parser 解析出的 AST 进行有效性验证，验证的方面主要包括以下两方面：

1. 表名、字段名、函数名是否正确，如在某个查询的字段在当前 SQL 位置上是否存在或有歧义（当前可见的多个数据源中同时存在该名称的字段）

2. 特定类型操作自身的合法性，如 `group by` 聚合中的聚合函数是否存在嵌套调用，使用 `AS` 重命名时，新名字是否是 `x.y` 的形式等

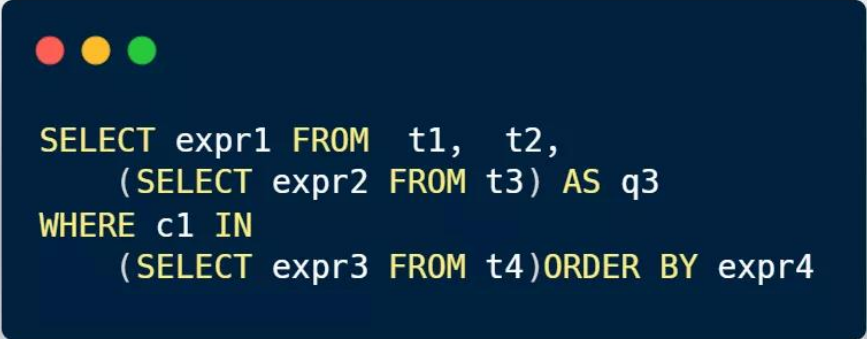
针对上面的第一种情况，在校验过程中首先需要明确两个最重要的概念：`Namespace` 和 `Scope`。

`Namespace` 代表一个逻辑上的数据源，可以是一张表，也可以是一个子查询，而 `Scope` 则代表了在 SQL 的某个位置，表和字段的可见范围。

从概念中可以看出，在某个 SQL 位置上，某个字段所对应的 `scope` 可能包含多个 `namespace`。在 `validate` 阶段解析出来的 `scope` 和 `namespace` 信息会被保存下来，在后面转换成逻辑执行计划的时候还会用到。

通过一个示例来看什么是 `Namespace` 和 `scope`

示例 3



```
SELECT expr1 FROM t1, t2,  
       (SELECT expr2 FROM t3) AS q3  
WHERE c1 IN  
       (SELECT expr3 FROM t4)ORDER BY expr4
```

在上面这样一段 SQL 语句中包含四个 `namespace`：

```
t1
t2
(SELECT expr2 FROM t3) AS q3
(SELECT expr3 FROM t4)
```

对于 SQL 中的不同表达式，根据它们所在的位置，它们所对应的 scope 如下：

```
expr1 数据来源于 t1, t2, q3
expr2 数据来源于 t3
expr3 数据来源于 t1, t2, t4
expr4 数据来源于 t1, t2, q3 加上
      (取决于方言) SELECT子句中定义的任何别名
```

在 Calcite 中，validator 的具体实现类是 `SqlValidatorImpl`，`namespace` 和 `scope` 分别由接口 `SqlValidatorNamespace` 和 `SqlValidatorScope` 表示，图中涉及到的 `xxxNamespace` 和 `xxxScope` 分别是这两个类的子类。

```

graph TD
    subgraph SqlValidatorImpl
        validate() --> validateSqlExpression()
        validateSqlExpression() --> performanceConditionalResources
        validateSqlExpression() --> registerQuery
        validateSqlExpression() --> alterTable
        registerQuery --> validate()
        alterTable --> validate()
    end

    subgraph SqlSelect
        validate() --> validateQuery()
        validateQuery() --> validateNamespace()
        validateNamespace() --> namespaceValidate()
    end

    subgraph SelectNamespace
        validate() --> validateSqlSelect()
        validateSqlSelect() --> validate()
    end

    subgraph IdentifierNamespace
        validate() --> validateIdentifier()
        validateIdentifier() --> resolveTable()
    end

    subgraph EmptyScope
        resolveTable() --> resolveTable()
    end

    subgraph Expander
        resolveTable() --> visitIdentifierId()
        visitIdentifierId() --> getScopeFullyQualified()
    end

    subgraph SelectScope
        getScopeFullyQualified() --> fullyQualified
        fullyQualified --> resolveQualifiedTableName
        resolveQualifiedTableName --> resolve
        resolve --> resolveNamespace
        resolveNamespace --> resolveTable
    end

    validate() --> validateQuery()
    validateQuery() --> validateNamespace()
    validateNamespace() --> namespaceValidate()
    namespaceValidate() --> validateSqlSelect()
    validateSqlSelect() --> validate()
    validate() --> validateIdentifier()
    validateIdentifier() --> resolveTable()
    resolveTable() --> resolveTable()
    resolveTable() --> visitIdentifierId()
    visitIdentifierId() --> getScopeFullyQualified()
    getScopeFullyQualified() --> fullyQualified
    fullyQualified --> resolveQualifiedTableName
    resolveQualifiedTableName --> resolve
    resolve --> resolveNamespace
    resolveNamespace --> resolveTable
    resolveTable --> validate()

```

The flowchart illustrates the SQL validation process in MyBatis, organized into seven vertical sections:

- SqlValidatorImpl**: Contains the main `validate()` method, which branches into `performanceConditionalResources`, `registerQuery`, and `alterTable`. `registerQuery` and `alterTable` both call `validate()` recursively.
- SqlSelect**: Shows the flow from `validate()` to `validateQuery()`, then `validateNamespace()`, and finally `namespace.validate()`.
- SelectNamespace**: Shows the flow from `validate()` to `validateSqlSelect()`, which then calls `validate()` recursively.
- IdentifierNamespace**: Shows the flow from `validate()` to `validateIdentifier()`, which then calls `resolveTable()`.
- EmptyScope**: Shows the flow from `resolveTable()` to `resolveTable()`.
- Expander**: Shows the flow from `resolveTable()` to `visitIdentifierId()`, which then calls `getScopeFullyQualified()`.
- SelectScope**: Shows the flow from `getScopeFullyQualified()` to `fullyQualified`, then `resolveQualifiedTableName`, `resolve`, `resolveNamespace`, and finally `resolveTable`, which calls `validate()` recursively.

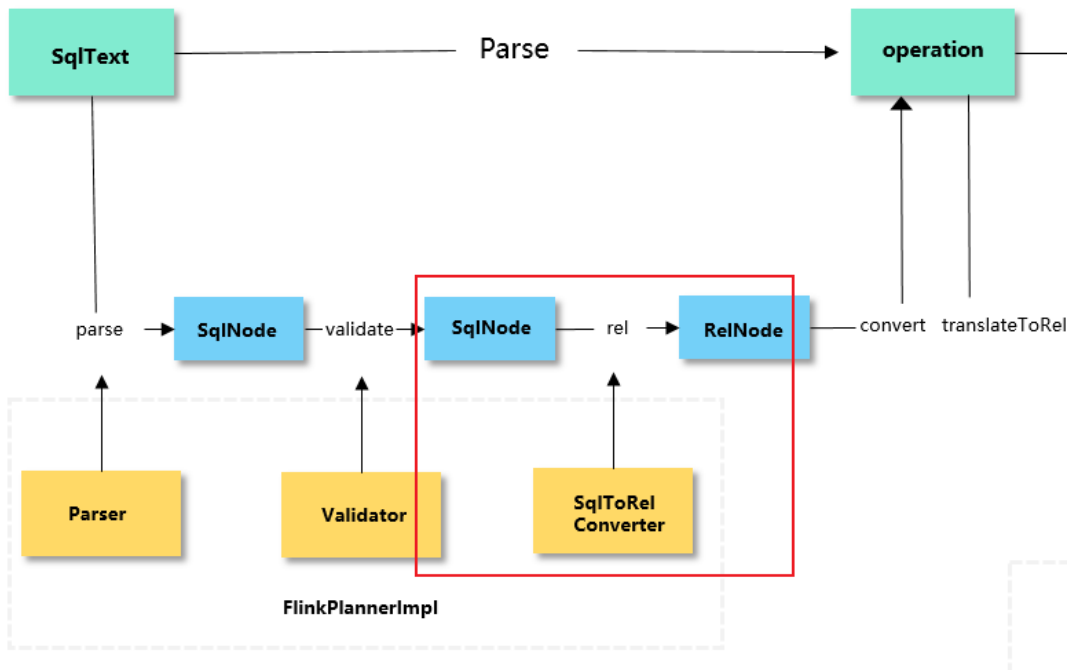
Key annotations in the diagram include:

- 递归调用validate()**: Recursive call to `validate()` (appearing multiple times).
- 递归调用resolveTable()**: Recursive call to `resolveTable()`.
- 递归调用validateSqlSelect()**: Recursive call to `validateSqlSelect()`.
- 递归调用validateIdentifier()**: Recursive call to `validateIdentifier()`.
- 递归调用resolveQualifiedTableName**: Recursive call to `resolveQualifiedTableName`.
- 递归调用resolve**: Recursive call to `resolve`.
- 递归调用resolveNamespace**: Recursive call to `resolveNamespace`.
- 递归调用resolveTable**: Recursive call to `resolveTable`.

大体过程都已经在图中的注解里进行了说明，需要补充的一点是，在通过 `emptyScope.resolve` 解析表名时，表信息是通过具体的 `catalogReader` 从 `catalog` 的 `schema` 中查找出来的。

在 flink 中，根据用户的配置，catalog 可能是 GenericInMemoryCatalog（基于内存的 catalog）或 HiveCatalog（基于 hive metastore 的 catalog）。

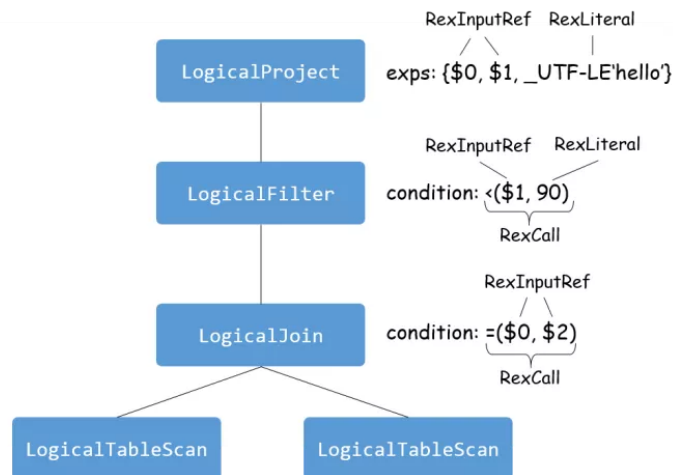
如下图所示:



rel 阶段是将 **SqlNode** 组成的一棵抽象语法树转化为一棵由 **RelNode** 和 **RexNode** 组成的关系代数树，或者称为执行计划。**RelNode** 表示关系表达式，如投影（Project），即 **SELECT**，和连接（JOIN）等；

RexNode 表示行表达式，如示例中的 **CAST(score AS INT)**、**T1.id < 10**。

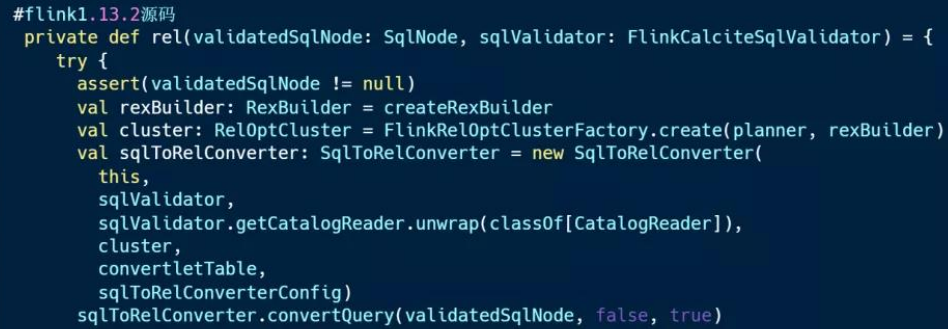
以示例 2 的语法树为例，在经过 rel 阶段转换后会生成下图所示的执行计划：



rel 阶段只处理 DML 和 DQL

因为 DDL 实际上可以认为是对元数据的修改，不涉及复杂关系查询，也就不需要进行关系代数转换来优化执行，所以也无需转换为 `RelNode` 表示，根据对应的 `SqlNode` 中保存的信息已经可以直接执行了。

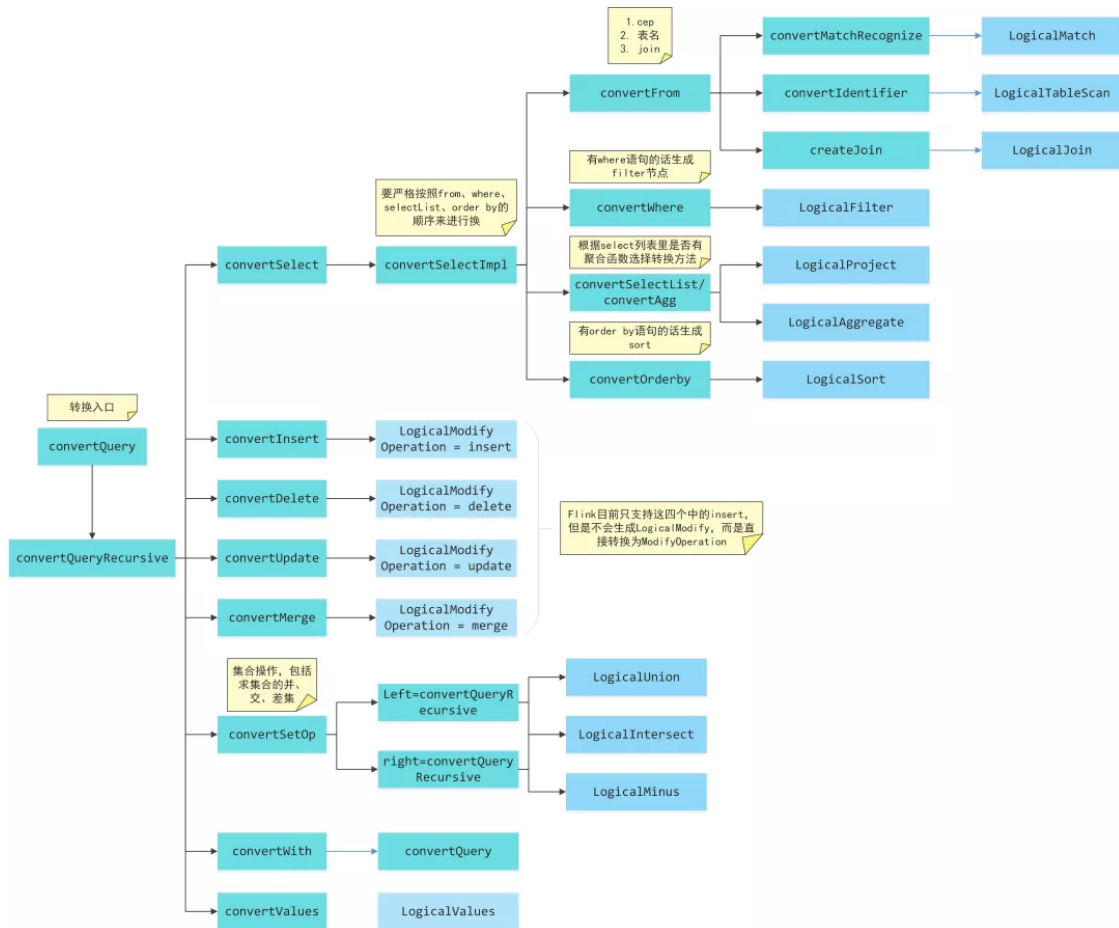
在 `calcite` 中，`SqlToRelConverter` 用于对关系表达式进行转换。Flink 中通过如下方式使用 `calcite` 将 AST 转换成逻辑执行计划，如下图源码所示。



```
#flink1.13.2源码
private def rel(validatedSqlNode: SqlNode, sqlValidator: FlinkCalciteSqlValidator) = {
  try {
    assert(validatedSqlNode != null)
    val rexBuilder: RexBuilder = createRexBuilder
    val cluster: RelOptCluster = FlinkRelOptClusterFactory.create(planner, rexBuilder)
    val sqlToRelConverter: SqlToRelConverter = new SqlToRelConverter(
      this,
      sqlValidator,
      sqlValidator.getCatalogReader.unwrap(classOf[CatalogReader]),
      cluster,
      convertLetTable,
      sqlToRelConverterConfig)
    sqlToRelConverter.convertQuery(validatedSqlNode, false, true)
  }
}
```

从 Flink1.13.2 源码中可以看到转换的入口是 `convertQuery` 方法。

`SqlToRelConverter` 中的简单的转换流程如下图所示：



针对每种可能的根节点类型都有对应的转换方法。其中 DELETE、UPDATE、MERGE、WITH 和 VALUES 这几种语法在 flink 流式 SQL 中还不支持，并且其转换过程也比较简单，后文不再详细分析。

对于一棵转换后得到的逻辑执行计划树中的节点，其实在 AST 中都是可以一一对应的找到对应的节点的，所以转换过程本身并不涉及很复杂的算法，大部分过程是提取已有 SqlNode 节点中记录的信息，然后生成对应的 RelNode 和 RexNode，并设置 RelNode 间的父子关系。

从图中也可以看出在 calcite 里最终都会生成一个 LogicalModify 节点，通过节点内的 operation 属性来标识不同的含义。但是目前 flink 支持的 DML 只有 insert 语句，而且并不会生成 LogicalModify 节点，而是直接转换成了 ModifyOperation，并在需要的时候转换成 flink 内部自己定义的节点类型 LogicalSink。也因为这个原因，对于 DML 的转换流程图中是略有简化的，insert、delete、update 和 merge 本身都可以带查询语句，因此实际转换的时候都会递归地先对查询部分进行转换。

上图所示流程中只展示了对关系表达式的转换，但是每个关系节点（RelNode）中的行表达式同样需要经过转换得来。

Calcite 中行表达式的转换依赖于两个对象：BlackBoard 和 SqlNodeToRelConverter。

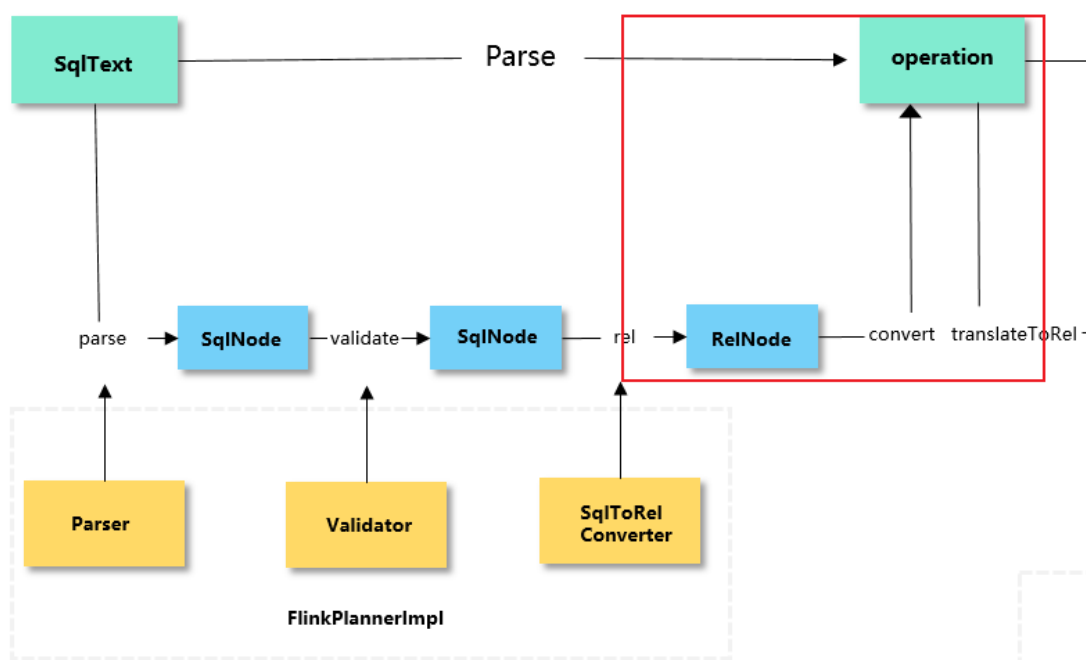
BlackBoard 是对 select 进行转换时的一个临时工作空间，它就像一块“黑板”一样，可以临时记录下转换过程中需要的信息，比如 select 依赖的 scope、当前的 root 节点、当前节点是否是 top 节点等。

BlackBoard 本身还是一个 shuttle，针对不同类型的 SqlNode，其内部都有对应的 visit 方法。其中除 SqlCall、SqlLiteral、SqlIntervalQualifier 外，都可由 BlackBoard 和 SqlToRelConverter 中定义的各种 convertXXX 方法进行转换，这三种类型的 SqlNode 则需要借助 SqlNodeToRelConverter 来进行转换。

SqlLiteral、SqlIntervalQualifier 的转换比较简单，就是从原来的 SqlNode 中提取信息进行简单的处理和转换，然后生成对应的 RexNode。

4、convert:RelNode => Operation

重点：这一步负责将 RelNode tree 转换成 operation



这里只有 DQL 以及 DML 的查询子句会首先通过 planner.rel 转换为 RelNode

RelNode 转换成 Operation 的过程很简单，针对四种类型的操作，其各自的转换过程如下：

1. CreateTable @convertCreateTable

如果 AST 的根节点是 `SqlCreateTable`，提取节点中记录的 `schema`、`properties`、`comment`、`primary keys`、`if not exists` 信息，创建 `CatalogTable` 对象，然后创建 `CreateTableOperation`

1. DropTable @convertDropTable

如果 AST 的根节点是 `SqlDropTable`，提取节点中记录的 `full table name`、`if exists` 信息，创建 `DropTableOperation` 对象

1. Insert @convertInsert

如果 AST 的根节点是 `RichSqlInsert`，提取节点中记录的目标表的完整路径和查询表达式，先将查询表达式通过 `convertSqlQuery` 转换成 `QueryOperation`，然后以转换后的 `QueryOperation` 为子节点创建 `ModifyOperation` 对象。

这里分两种情况：

(1)使用 SQL API 执行了 `insert into` 语句，将数据写入已经通过 `TableEnvironment` 注册过的表中，此时创建的是 `CatalogSinkModifyOperation`

(2)使用 Table API 的 `toXXXStream` 将 table 对象转换成了 `DataStream`，创建的是 `OutputConversionModifyOperation`

1. Query @convertSqlQuery

如果根节点的 `SqlKind` 是 `SqlKind.Query`，先通过 `FlinkPlannerImpl.rel` 将 `SqlNode` 转换成 `RelNode`，然后创建 `PlannerQueryOperation` 对象

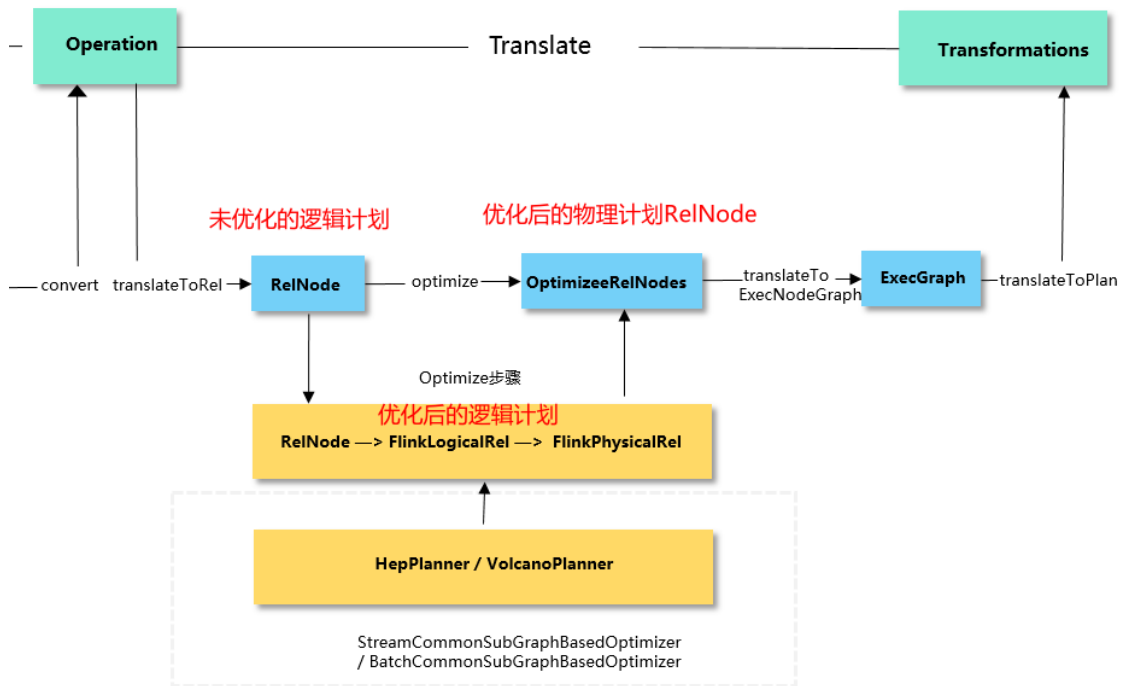
3.2、Translate 阶段

在 Translate 阶段，通过 Blink Planner 的 `translateToRel`、`optimize`、`translateToExecNodeGraph` 和 `translateToPlan` 四个阶段:将 `Operation` 转换成 `Transformations`。

重点：

1. 从 `operation` 开始，先将 `ModifyOperation` 通过 `translateToRel` 方法转换成 `Calcite RelNode` 逻辑计划树，在对应转换成 `FlinkLogicalRel`（`RelNode` 逻辑计划树）；
2. 然后经过 调用 `optimize` 方法将 `FlinkLogicalRel` 优化成 `FlinkPhysicalRel`。
3. 再调用 `translateToExecNodeGraph` 方法将 `FlinkPhysicalRel` 转为 `execGraph`
4. 最后调用 `translateToPlan` 方法将 `execGraph` 转为 `transformations`

从逻辑计划变成物理计划（`RelNode`），



Flink1.13.2 源码如下：

```

//调用translate方法
override def translate(
  modifyOperations: util.List[ModifyOperation]): util.List[Transformation[_]] = {
  validateAndOverrideConfiguration()
  if (modifyOperations.isEmpty) {
    return List.empty[Transformation[_]]
  }
  //将ModifyOperation转换为Calcite RelNode逻辑计划
  val relNodes = modifyOperations.map(translateToRel)
  // 优化Calcite逻辑计划,调用optimize方法将FlinkLogicalRel 优化成FlinkPhysicalRel.
  val optimizedRelNodes = optimize(relNodes)
  //调用translateToExecNodeGraph方法将FlinkPhysicalRel转为execGraph
  val execGraph = translateToExecNodeGraph(optimizedRelNodes)
  //调用translateToPlan方法将execGraph转为transformation
  val transformations = translateToPlan(execGraph)
  cleanupInternalConfigurations()
  transformations
}

```

3.2.1、translateToRel

这个过程可以看成是 **convert: RelNode => Operation** 的逆过程。

逻辑也很简单，无论是使用 SQL API 还是 Table API，最终生成的 operation 的根节点一定是 **ModifyOperation**，因为只有 insert 语句或者将 Table 转换成 **DataStream** 后，在 **DataStream** 结果上面写入 sink 才能触发执行。

前文提到过 **ModifyOperation** 最终都会被转换成 flink 内自定义的 **LogicalSink** 节点，该节点主要记录数据输出信息，核心在于需要创建出表示数据输出的 sink。所以针对三种 **ModifyOperation** 类型分别创建 sink 的过程如下：

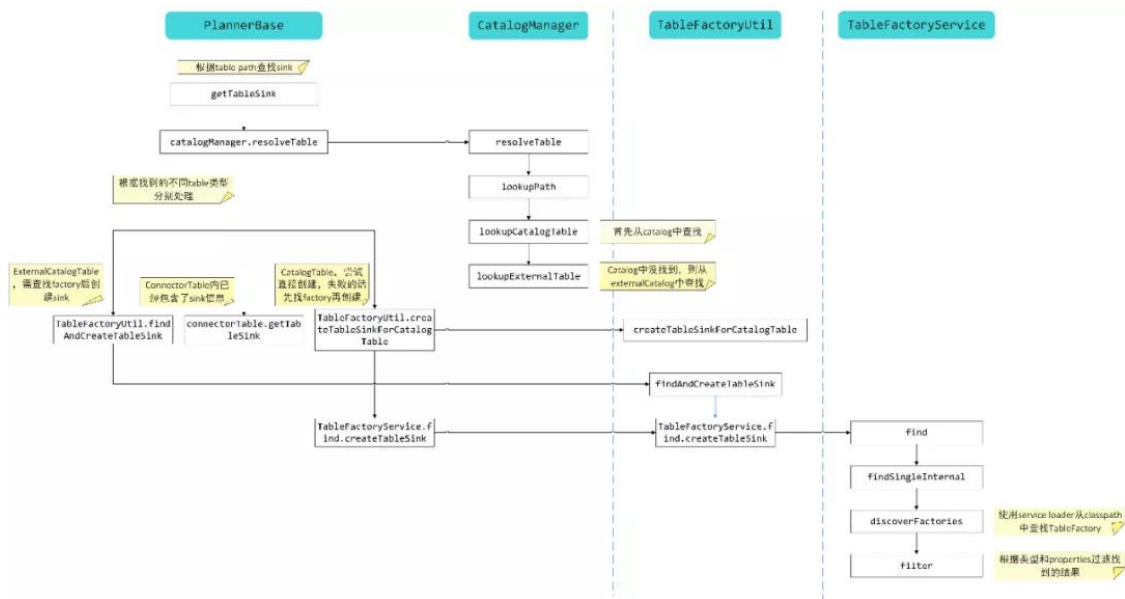
1. UnregisteredSinkModifyOperation:

这个 operation 中直接记录了 sink 信息，因此直接提取出来创建 **LogicalSink** 即可。

1. CatalogSinkModifyOperation:

根据 operation 中记录的 table path 找到对应的 table，然后根据 table 创建出 table sink，最后使用 table sink 创建出 **LogicalSink** 节点。

这个过程中涉及到了在 catalog 中解析 table 和使用 **ServiceLoader** 根据 table 信息在 classpath 中查找并用于创建 table sink 的 **TableSinkFactory** 的过程，具体如下图所示。



3.2.2、optimize

会使用两个优化器:RBO(基于规则的优化器) 和 CBO(基于代价的优化器)

RBO(基于规则的优化器)会将原有表达式裁剪掉，遍历一系列规则（Rule），只要满足条件就转换，生成最终的执行计划。一些常见的规则包括分区裁剪（Partition Prune）、列裁剪、谓词下推（Predicate Pushdown）、投影下推（Projection Pushdown）、聚合下推、limit 下推、sort 下推、常量折叠（Constant Folding）、子查询内联转 join 等。2.CBO(基于代价的优化器)会将原有表达式保留，基于统计信息和代价模型，尝试探索生成等价关系表达式，最终取代价最小的执行计划。CBO 的实现有两种模型，Volcano 模型，Cascades 模型。这两种模型思想很是相似，不同点在于 Cascades 模型一边遍历 SQL 逻辑树，一边优化，从而进一步裁剪掉一些执行计划。

源码如下：

```
@VisibleForTesting
private[flink] def optimize(relNodes: Seq[RelNode]): Seq[RelNode] = {
  val optimizedRelNodes = getOptimizer.optimize(relNodes)
  require(optimizedRelNodes.size == relNodes.size)
  optimizedRelNodes
}
```

translateToExecNodeGraph

调用 translateToExecNodeGraph 方法将 FlinkPhysicalRel 转为 execGraph


```
@VisibleForTesting
private[flink] def translateToExecNodeGraph(optimizedRelNodes: Seq[RelNode]): ExecNodeGraph = {
  val nonPhysicalRel = optimizedRelNodes.filterNot(_.asInstanceOf[FlinkPhysicalRel])
  if (nonPhysicalRel.nonEmpty) {
    throw new TableException("The expected optimized plan is FlinkPhysicalRel plan, " +
      s"actual plan is ${nonPhysicalRel.head.getClass.getSimpleName} plan.")
  }

  require(optimizedRelNodes.forall(_.asInstanceOf[FlinkPhysicalRel]))
  // Rewrite same rel object to different rel objects
  // in order to get the correct dag (dag reuse is based on object not digest)
  val shuttle = new SameRelObjectShuttle()
  val relsWithoutSameObj = optimizedRelNodes.map(_.accept(shuttle))
  // reuse subplan
  val reusedPlan = SubplanReuser.reuseDuplicatedSubplan(relsWithoutSameObj, config)
  // convert FlinkPhysicalRel DAG to ExecNodeGraph
  val generator = new ExecNodeGraphGenerator()
  val execGraph = generator.generate(reusedPlan.map(_.asInstanceOf[FlinkPhysicalRel]))

  // process the graph
  val context = new ProcessorContext(this)
  val processors = getExecNodeGraphProcessors
  processors.foldLeft(execGraph)((graph, processor) => processor.process(graph, context))
}
```

3.2.4、translateToExecNodeGraph

调用 translateToPlan 方法将 execGraph 转为 transformations

```
override protected def translateToPlan(execGraph: ExecNodeGraph): util.List[Transformation[_]] = {
  val planner = createDummyPlanner()

  execGraph.getRootNodes.map {
    case node: StreamExecNode[_] => node.translateToPlan(planner)
    case _ =>
      throw new TableException("Cannot generate DataStream due to an invalid logical plan. " +
        "This is a bug and should not happen. Please file an issue.")
  }
}
```

通过上述四个步骤，实现将 Operation 转换成 Transformations。

小笨猪通过完整的流程分析后，终于搞懂了 Flink sql 的解析和转换过程，最终 SQL 被转为 Transformations,后面的步骤就变成了 Flink DataStream 的提交流程，小笨猪还是比较了解的。

分享

时间一眨眼就到了分享的那天，小笨猪在讲台上对 Flink sql 应用提交做了深入讲解，得到了导师猴哥和主管的一致认可。



猴哥

阿土，讲的很棒呦！



小笨猪

本文原创者：土哥、一名大数据算法工程师。文章首发平台：微信公众号【3 分钟秒懂大数据】

土哥用 Flink1.13.2 源码对 Flink sql 应用提交进行讲解、原创不易，各位觉得文章不错的话，不妨点赞（在看）、留言、转发三连走起！谢谢大家！

扫码加入Flink流计算群 群若过期，加博主微信，拉你进群



3分钟秒懂大数据



扫一扫上面的二维码图案，加我微信



Flink技术交流群



该二维码7天内(9月29日前)有效，重新进入将更新