DuckDB源码阅读报告1

一、Duckdb简介

本节的主要内容是对duckdb及OLAP数据库进行简要介绍并提供一个简单示例

1. 什么是OLAP数据库

数据处理一般可以分为两大类，一类是联机分析处理OLAP（OnLine Analytical Processing），另一类是联机事务处理OLTP（OnLine Transaction Processing）

- OLAP为使用多维结构为分析提供对数据的快速访问的技术，OLAP 的源数据通常存储在关系数据库的数据仓库中。OLAP是数据仓库系统的主要应用，支持复杂的分析操作，侧重决策支持，并且提供直观易懂的查询结果。
- OLTP也称为面向交易的处理系统，其基本特征是顾客的原始数据可以立即传送到计算中心进行处理，并在很短的时间内给出处理结果。OLTP是传统的关系型数据库的主要应用，主要是基本的、日常的事务处理，例如银行交易。

2. Duckdb有什么特点

快速分析查询

DuckDB 是一个 OLAP 数据库，因此存储的任何数据都按列组织。与列相关的所有数据在内存中彼此相邻存储，并且数据库针对高效读取和计算列进行了优化。DuckDB 经过优化，在列示矢量化查询引擎上运行，有助于对数据执行快速且复杂的查询。

支持SQL以及其他编程语言的集成

DuckDB提供了多种编程语言的API，并且与Python和R语言深度集成，方便用户进行高效的交互式数据分析。

3. Duckdb使用简例

可以通过命令行安装DuckDB并使用，官方文档中也提供了其他语言的库文件或是包安装方法，DuckDB没有外部依赖，也不需要任何服务器，使用起来比较简单，这也是它的优势之一。

二、主要功能分析与建模

DuckDB的组成

查看src文件夹下的结构层次可以分析出，duckdb主要由以下几个模块组成

Parser

Parser模块主要作用是根据定义的语法规则判断一条SQL语句是否符合对应的语法结构，这是任何进入 DuckDB 的查询的入口点。DuckDB 使用 Postgres 的解析器（libpg_query）。在使用该解析器解析查询之后，标记会被转换为基于 `SQLStatements`、`Expressions` 和 `TableRefs` 的自定义解析树表示。

Planner

规划器负责将`Parser`从查询字符串中提取的标记转换成逻辑查询计划。该计划表示为一棵树，树上有 `LogicalOperator` 类型的节点。

Optimizer

优化器采用 “规划器 ”生成的逻辑查询计划，并将其转换为逻辑上等价但（希望）执行速度更快的逻辑查询计划。优化的例子包括谓词下推、表达式重写和连接排序。基于成本的优化和基于规则的优化都会执行。

Execution

执行层首先获取由 “优化器 ”生成的逻辑查询计划，并将其转换为由 “物理操作器 ”组成的物理查询计划。然后使用基于推送的执行模型来执行 “物理操作符”。

Catalog

目录跟踪数据库中包含的表、模式和函数。在规划阶段，“装订器 ”使用 “目录 ”将符号（如 “table_name”）解析为数据库中实际存在的表和列。

Storage

存储组件负责管理内存和磁盘中的实际物理数据。每当执行层需要访问基础表数据（如执行基础表扫描）或需要更新数据库中存储的信息（如作为 “INSERT ”或 “UPDATE ”命令的一部分）时，就会使用存储组件。

Transaction

事务管理器管理所有当前打开的事务，并负责处理 `COMMIT` 或 `ROLLBACK` 命令。

other

除此之外，duckdb还提供了如并发、验证和第三方兼容的功能。

Parser模块分析

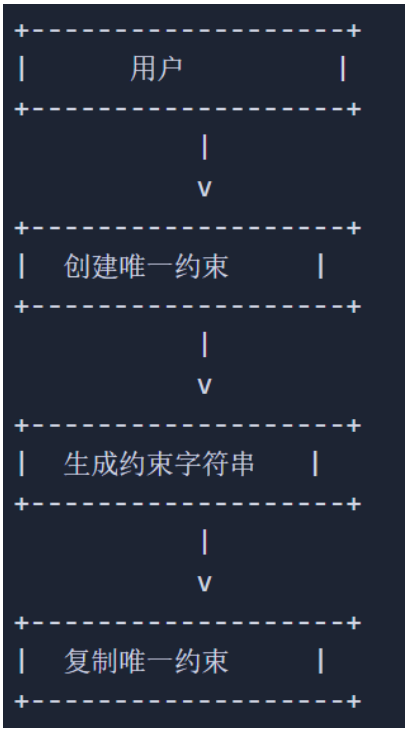
层次结构

parser
|
*--constraints
|
*--expression
|
*--parsed_data
|
*--query_node
|
*--statement
|
*--tableref
|
*--transform
|
*--otherfiles

constraints

- 这部分主要用于定义和实现约束(constraints)，特别是唯一约束(unique constraints)
- 类图

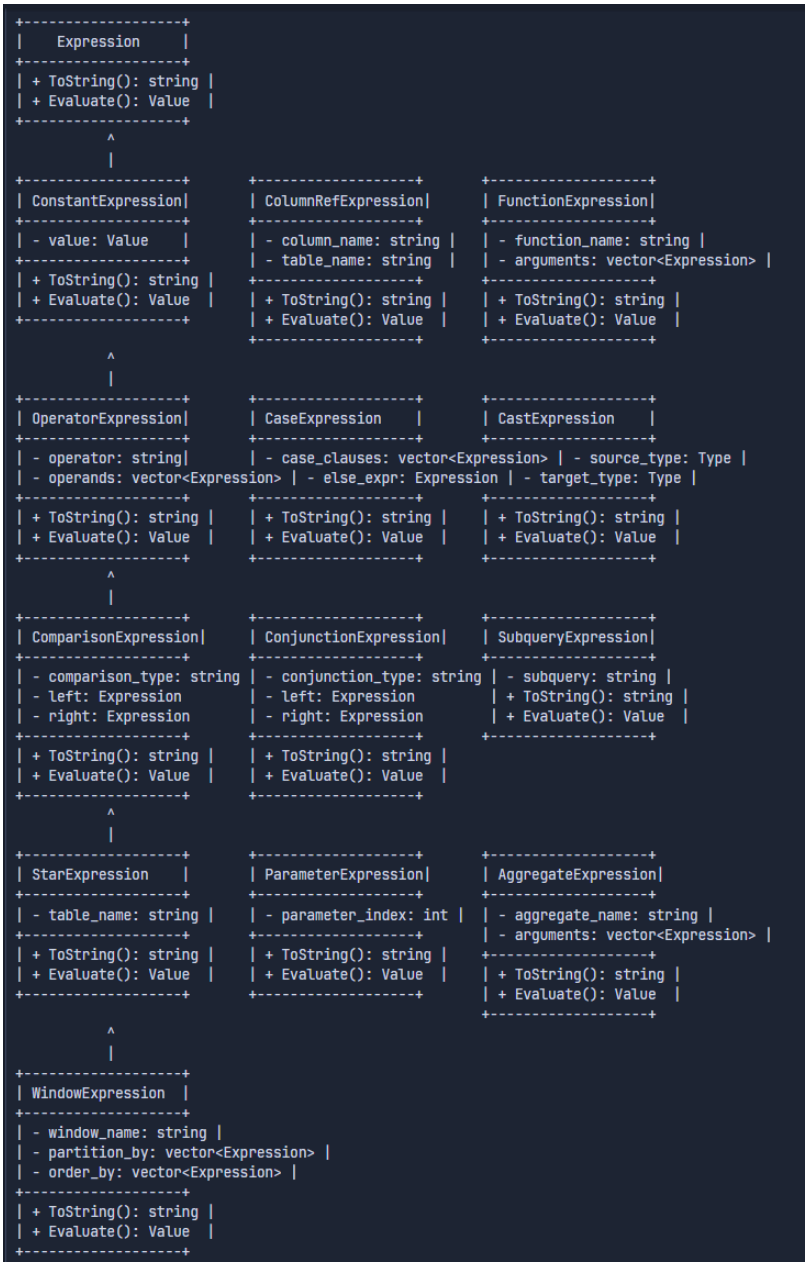
- 需求用例图



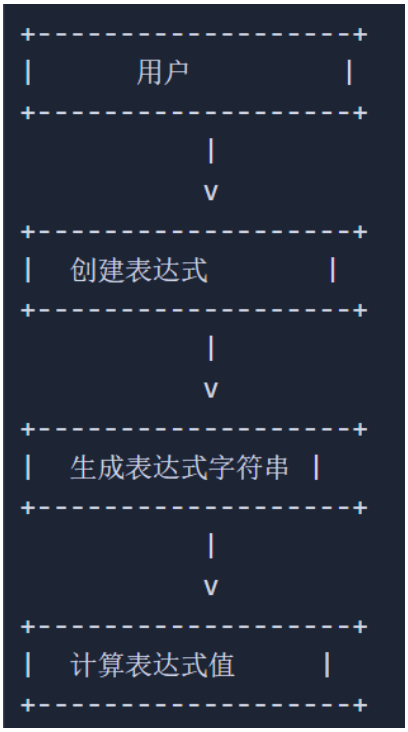
expression

- 该目录下包含了各种表达式类型的定义和实现
- 面向对象思想分析：
  - 每个表达式类都封装了与该表达式相关的数据和方法。
    - 成员变量：存储表达式的具体数据，如常量值、列名、函数名、操作符等。
    - 成员方法：用于操作和访问这些数据，如获取值、转换为字符串等。
  - 父类是`Expression`，子类是`ConstantExpression`等不同的具体表达式类型

- 类图


Expression
+ ToString(): string
+ Evaluate(): Value
ConstantExpression
- value: Value
ColumnRefExpression
- column_name: string
- table_name: string
FunctionExpression
- function_name: string
- arguments: vector<Expression>
OperatorExpression
- operator: string
- operands: vector<Expression>
CaseExpression
- case_clauses: vector<Expression>
- else_expr: Expression
CastExpression
- source_type: Type
- target_type: Type
ComparisonExpression
- comparison_type: string
- left: Expression
- right: Expression
ConjunctionExpression
- conjunction_type: string
- left: Expression
- right: Expression
SubqueryExpression
- subquery: string
StarExpression
- table_name: string
ParameterExpression
- parameter_index: int
AggregateExpression
- aggregate_name: string
- arguments: vector<Expression>
WindowExpression
- window_name: string
- partition_by: vector<Expression>
- order_by: vector<Expression>

- 需求用例图



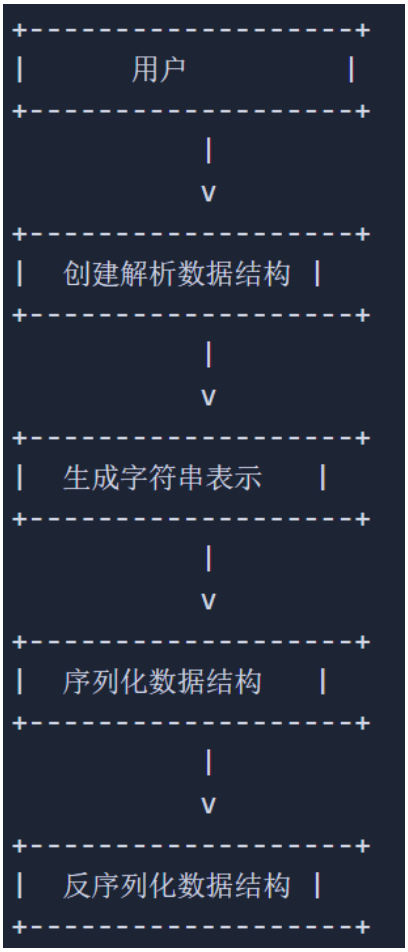
parsed_data

- 这个目录下包含了各种解析后的数据结构，包含对如下类型语句的解析结果
  - 创建`CREATE`
  - 删除`DROP`
  - 修改`ALTER`
  - 数据操作`INSERT`、`UPDATE`、`DELETE`、`SELECT`
- 面向对象思想分析：
  - 每个解析后的数据结构类都封装了与该结构相关的数据和方法。
    - 成员变量：存储解析后的 SQL 语句的具体数据，如表名、列名、约束、条件等。
    - 成员方法：用于操作和访问这些数据，如获取和设置表名、列名、约束、条件等。
  - 所有解析后的数据结构类都继承自一个通用的基类 `ParsedData`，表明它们都是某种类型的解析数据。
    - 基类：`ParsedData`
    - 派生类：`CreateTableInfo`、`CreateIndexInfo`、`CreateViewInfo`、`DropInfo`、`AlterInfo`、`InsertInfo`、`UpdateInfo`、`DeleteInfo`、`SelectInfo` 等。
  - 解析后的数据结构类通过重写 `ParsedData` 类中的虚函数，实现了多态性

- 类图


ParsedData
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
CreateTableInfo
- table_name: string
- columns: vector<Column>
- constraints: vector<Constraint>
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
CreateIndexInfo
- index_name: string
- table_name: string
- columns: vector<string>
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
CreateViewInfo
- view_name: string
- query: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
DropInfo
- object_name: string
- object_type: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
AlterInfo
- object_name: string
- alter_type: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
InsertInfo
- table_name: string
- columns: vector<string>
- values: vector<Value>
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
UpdateInfo
- table_name: string
- updates: vector<Update>
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
DeleteInfo
- table_name: string
- condition: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
SelectInfo
- columns: vector<string>
- table_name: string
- condition: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void

- 需求用例图

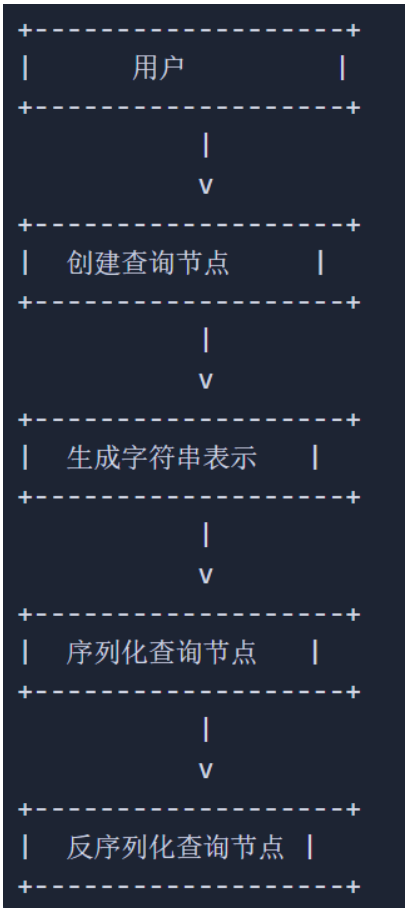


query_node

- 这个目录下的文件定义了各种查询节点，分别用于SQL查询语句的不同部分
  - 选择节点：如 `SelectNode`，表示 `SELECT` 语句的解析结果。
  - 集合操作节点：如 `SetOperationNode`，表示 `UNION`、`INTERSECT`、`EXCEPT` 等集合操作的解析结果。
  - **CTE** 节点：如 `RecursiveCTENode` 和 `CTENode`，表示递归和非递归 CTE 的解析结果。
  - 连接节点：如 `JoinNode`，表示连接操作的解析结果。
  - 基础表引用节点：如 `BaseTableRefNode`，表示对基础表的引用。
  - 子查询节点：如 `SubqueryNode`，表示子查询的解析结果。
  - 表函数节点：如 `TableFunctionNode`，表示表函数的解析结果。
  - 数据操作节点：如 `UpdateNode`、`DeleteNode`、`InsertNode`，表示 `UPDATE`、`DELETE`、`INSERT` 语句的解析结果。
- 面向对象思想分析：
  - 每个查询节点类都封装了与该节点相关的数据和方法。
    - 成员变量：存储查询节点的具体数据，如表名、列名、条件、连接类型等。

        - 成员方法：用于操作和访问这些数据，如获取和设置表名、列名、条件、连接类型等。
    - 所有查询节点类都继承自一个通用的基类 `QueryNode`，表明它们都是某种类型的查询节点。
        - 基类：`QueryNode`
        - 派生类：`SelectNode`、`SetOperationNode`、`RecursiveCTENode`、`CTENode`、`JoinNode`、`BaseTableRefNode`、`SubqueryNode`、`TableFunctionNode`、`UpdateNode`、`DeleteNode`、`InsertNode` 等。
- 类图

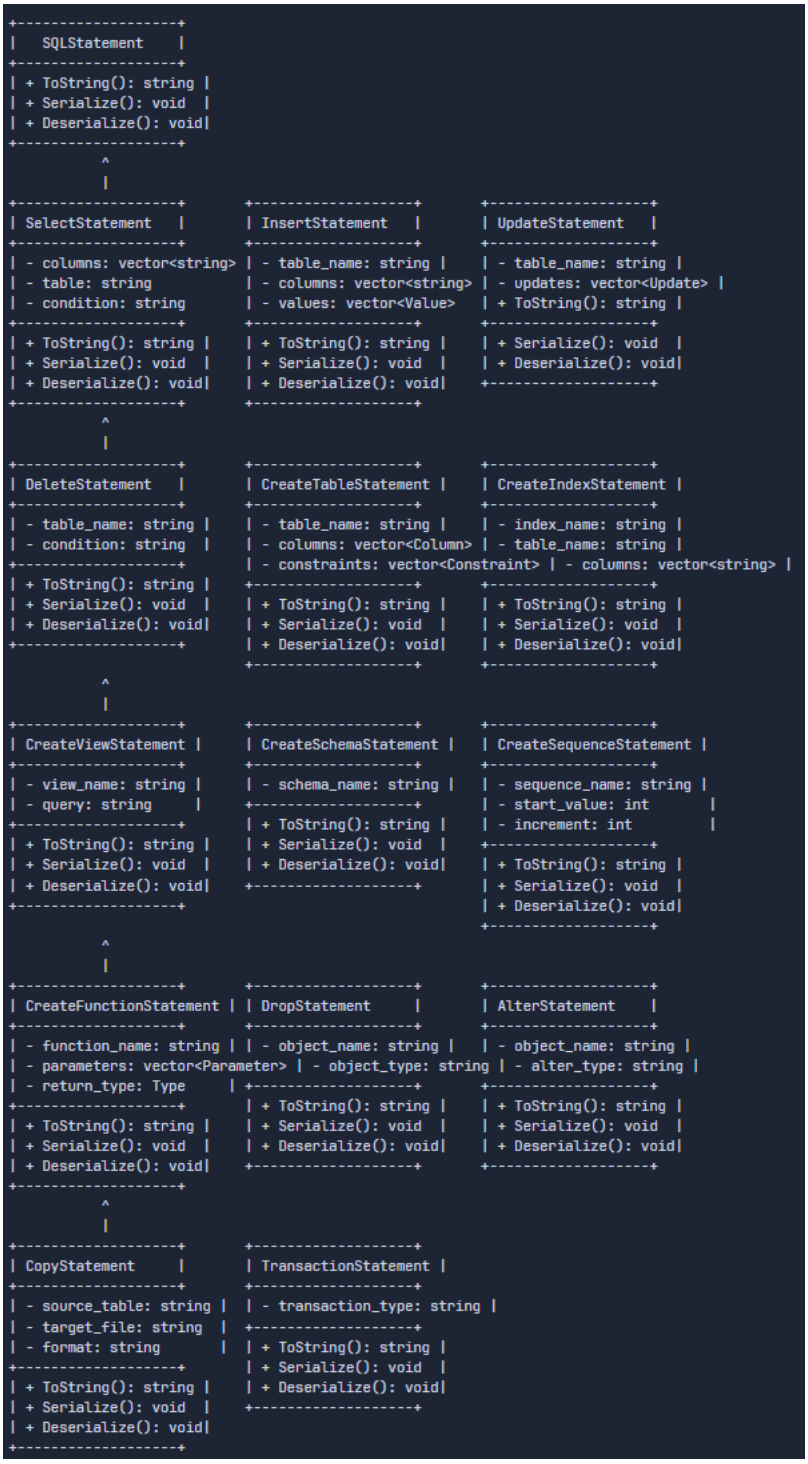
- 需求用例图



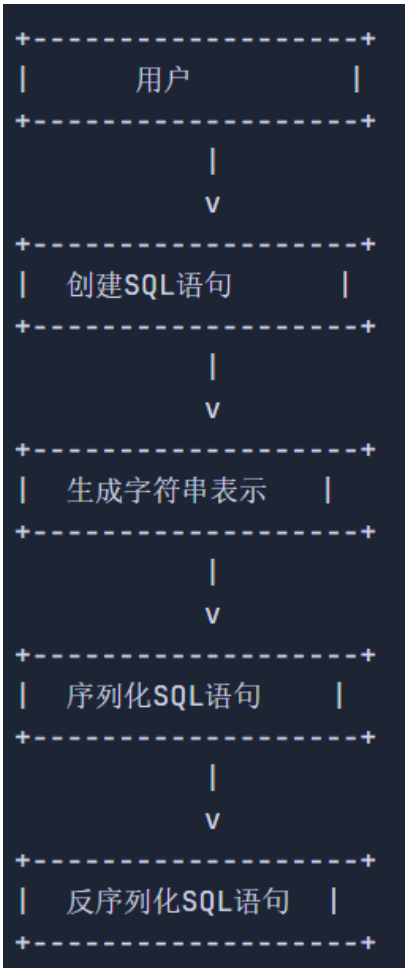
statement

- 该目录下的文件定义了各种SQL语句的解析结果，用于表示不同类型的SQL语句
- 所有 SQL 语句类都继承自一个通用的基类 `SQLStatement`，表明它们都是某种类型的 SQL 语句。
  - 基类：`SQLStatement`
  - 派生类：`SelectStatement`、`InsertStatement`、`UpdateStatement`、`DeleteStatement`、`CreateTableStatement`、`CreateIndexStatement`、`CreateViewStatement`、`CreateSchemaStatement`、`CreateSequenceStatement`、`CreateFunctionStatement`、`DropStatement`、`AlterStatement`、`CopyStatement`、`TransactionStatement` 等。
  - 通过继承，SQL 语句类可以重用 `SQLStatement` 类中的代码，并在此基础上扩展新的功能。

- 类图


SQLStatement
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
SelectStatement
- columns: vector<string>
- table: string
- condition: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
InsertStatement
- table_name: string
- columns: vector<string>
- values: vector<Value>
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
UpdateStatement
- table_name: string
- updates: vector<Update>
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
DeleteStatement
- table_name: string
- condition: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
CreateTableStatement
- table_name: string
- columns: vector<Column>
- constraints: vector<Constraint>
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
CreateIndexStatement
- index_name: string
- table_name: string
- columns: vector<string>
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
CreateViewStatement
- view_name: string
- query: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
CreateSchemaStatement
- schema_name: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
CreateSequenceStatement
- sequence_name: string
- start_value: int
- increment: int
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
CreateFunctionStatement
- function_name: string
- parameters: vector<Parameter>
- return_type: Type
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
DropStatement
- object_name: string
- object_type: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
AlterStatement
- object_name: string
- alter_type: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
CopyStatement
- source_table: string
- target_file: string
- format: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void
TransactionStatement
- transaction_type: string
+ ToString(): string
+ Serialize(): void
+ Deserialize(): void

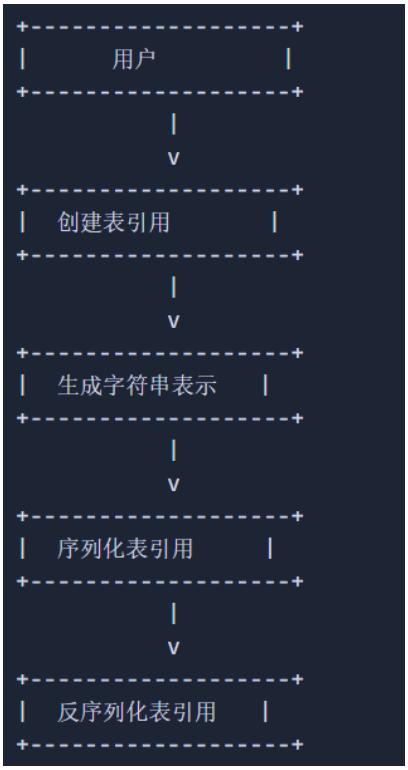
- 需求用例图



tableref

- 这个目录下的文件定义了各种表引用类型，用于表示SQL查询语句中对表的引用
  - 基础表引用：如 `BaseTableRef`，表示对基础表的引用。
  - 笛卡尔积引用：如 `CrossProductRef`，表示两个表的笛卡尔积。
  - 连接引用：如 `JoinRef`，表示两个表的连接操作。
  - 子查询引用：如 `SubqueryRef`，表示一个子查询。
  - 表函数引用：如 `TableFunctionRef`，表示对表函数的引用。
  - 空表引用：如 `EmptyTableRef`，表示一个空的表引用。
  - 表达式列表引用：如 `ExpressionListRef`，表示一个表达式列表。
- 所有表引用类都继承自一个通用的基类 `TableRef`，表明它们都是某种类型的表引用。
  - 基类：`TableRef`
  - 派生类：`BaseTableRef`、`CrossProductRef`、`JoinRef`、`SubqueryRef`、`TableFunctionRef`、`EmptyTableRef`、`ExpressionListRef` 等。

- 需求用例分析

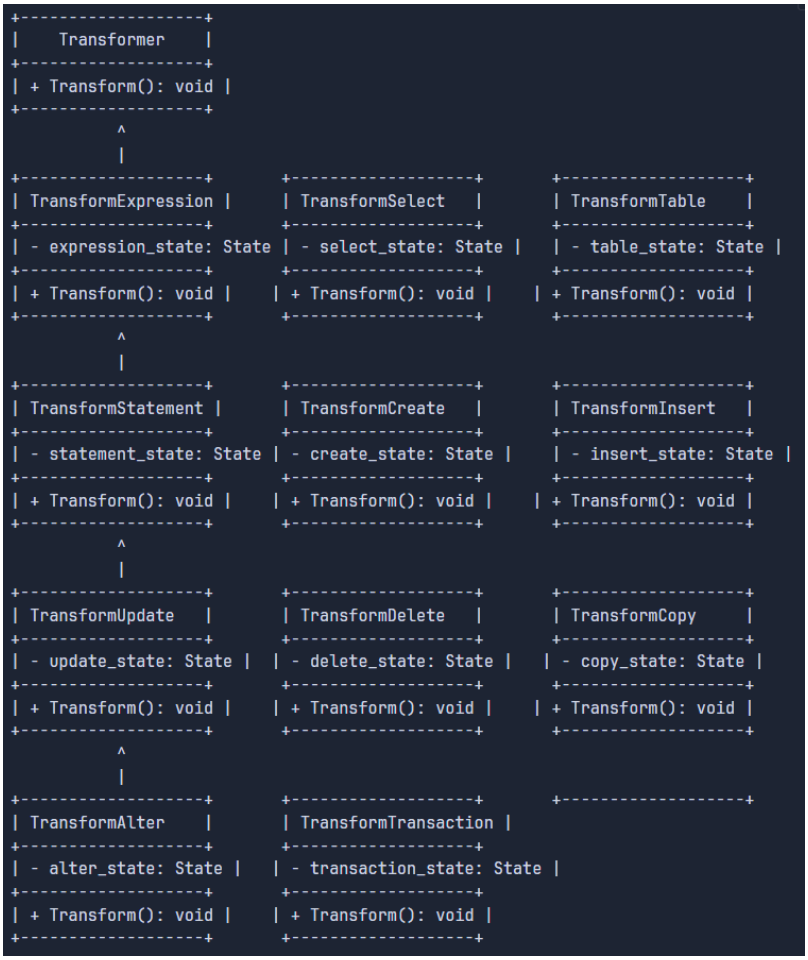


transform

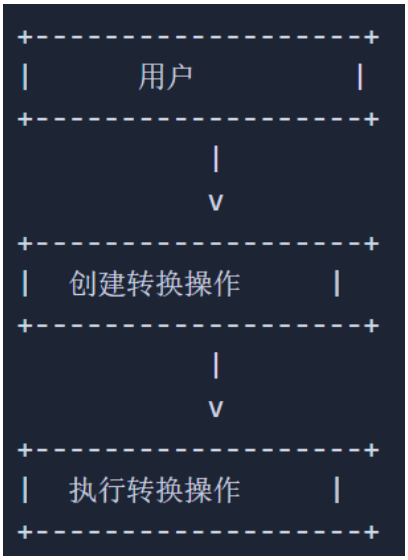
- 该目录下的文件定义了将 SQL 语句转换为内部表示的各种转换操作。
  - 转换器：如 `Transformer`，用于将 SQL 语句转换为内部表示。

    - 表达式转换：如 `TransformExpression`，用于将 SQL 表达式转换为内部表示。
    - 选择语句转换：如 `TransformSelect`，用于将 `SELECT` 语句转换为内部表示。
    - 表转换：如 `TransformTable`，用于将表相关的 SQL 语句转换为内部表示。
    - 语句转换：如 `TransformStatement`，用于将各种 SQL 语句转换为内部表示。
    - 创建语句转换：如 `TransformCreate`，用于将 `CREATE` 语句转换为内部表示。
    - 插入语句转换：如 `TransformInsert`，用于将 `INSERT` 语句转换为内部表示。
    - 更新语句转换：如 `TransformUpdate`，用于将 `UPDATE` 语句转换为内部表示。
    - 删除语句转换：如 `TransformDelete`，用于将 `DELETE` 语句转换为内部表示。
    - 复制语句转换：如 `TransformCopy`，用于将 `COPY` 语句转换为内部表示。
    - 修改语句转换：如 `TransformAlter`，用于将 `ALTER` 语句转换为内部表示。
    - 事务语句转换：如 `TransformTransaction`，用于将事务控制语句转换为内部表示。
- 所有转换类都继承自一个通用的基类 `Transformer`，表明它们都是某种类型的转换操作。
    - 基类：`Transformer`
    - 派生类：`TransformExpression`、`TransformSelect`、`TransformTable`、`TransformStatement`、`TransformCreate`、`TransformInsert`、`TransformUpdate`、`TransformDelete`、`TransformCopy`、`TransformAlter`、`TransformTransaction` 等。

- 类图



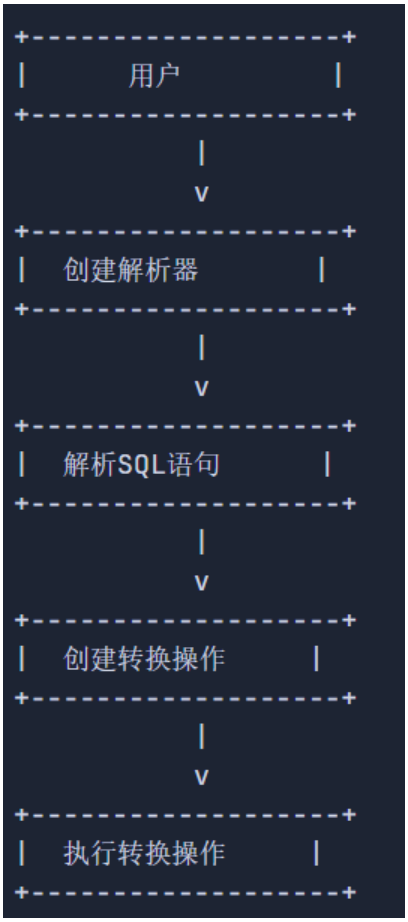
- 需求用例图



other files

- 这些文件主要是实现了解析器的功能，如
  - 解析上下文：如 `parser_context.cpp`，实现了解析上下文的功能。
  - 解析后的表达式基类：如 `parsed_expression.cpp`，实现了解析后的表达式的基类功能。
  - 解析后的数据基类：如 `parsed_data.cpp`，实现了解析后的数据的基类功能。

- 需求用例图



总结

本部分主要分析了`Parser`模块的主要功能，并对其下的具体功能模块进行了分析和建模。由于本次源码阅读报告重点在于分析其中的面向对象思想，所以对于代码细节实现并没有作过多讨论，但DuckDB项目的代码质量过硬也是其广受好评的一点原因。