存储过程设计文档

修订历史

版本	修订日期	修订描述	作者	备注
0.1	2015-9-1	存储过程设计文档	祝君	初稿
0.2	2016-1-3	增加存储过程详细设计	祝君	定稿

1 系统设计

1.1 综述

OceanBase是可扩展的关系数据库，实现了巨大数据量上的跨行跨表事务。但OB 0.4.2.8版本中缺乏对存储过程的支持。OceanBase的存储过程是一组为了完成特定功能的SQL 语句集，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。

本文档的编写目的，是实现对存储过程的支持，读者通过对本文档的阅读和学习，能够理解存储过程的设计方案，以及该设计方案在现有OceanBase系统中的技术实现流程。

1.2 名词解释

无

1.3 功能

存储过程是数据库系统的一个功能拓展，将一组带有特定功能的SQL语句以及控制流程语句存储在服务器端，用户通过存储过程的名称以及相应参数进行调用。存储过程支持变量，顺序，条件和循环等控制结构，存储过程可以定义参数这使得存储过程具有很强的模块性同时具有批处理性。虽然存储过程没有返回值，但是存储过程可以通过定义输出参数来实现返回结果的功能。这些特性都使得存储过程成为数据库系统必不可少的功能。

由于存储过程的源码是存储在服务器的，因此客户端可以通过存储过程名以及参数调用，不需要把这些SQL语句集合发送到服务器，这样的话可以减少网络中传输的数据量，降低网络负荷。

1.4 性能指标

无

2 模块设计

在OceanBase中实现存储过程我们参考的是PostgreSQL数据库，PostgreSQL支持使用内置过程语言PL/pgSQL、脚本语言、编译语言C和C++等编写存储过程。我们主要参考的是它的流程和PL/pgSQL语言的定义。OceanBase存储过程引擎的架构如图2.1所示

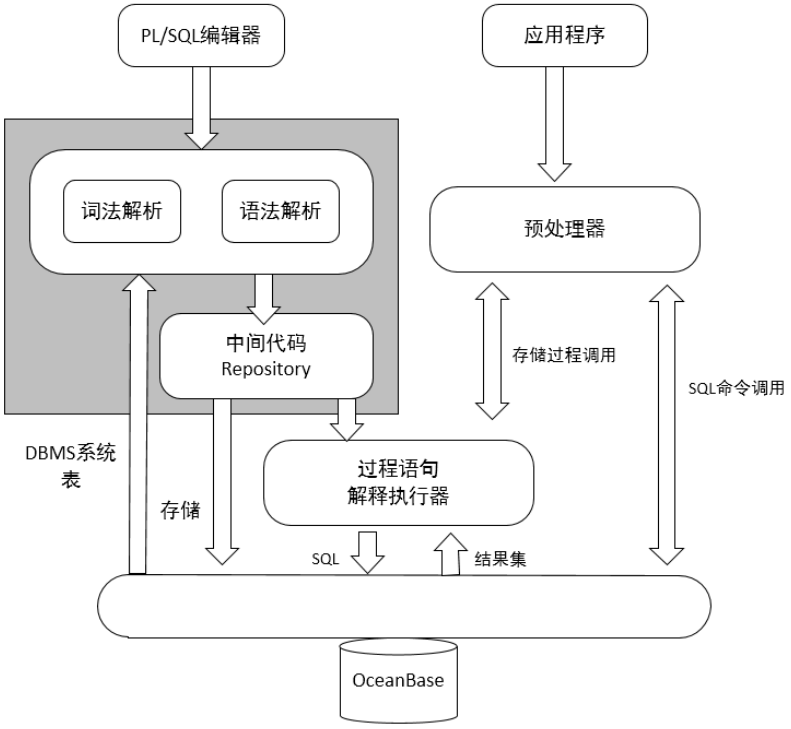

图2.1 存储过程引擎

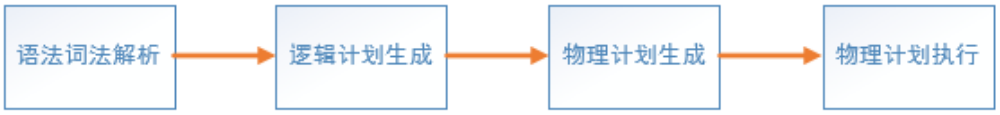

图2.2 SQL语句的基本处理框架

2.1 词法语法解析子模块设计

在存储过程的编译阶段需要将过程源代码编译为系统内部指令树。但需要注意的是，在函数内使用到的独立的SQL 表达式和 SQL 命令的编译则是在过程首次调用阶段完成，其由OceanBase的SQL解析模块负责编译，并产生执行计划。

正如传统的SQL一样，对过程源码的编译也需要经过词法解析和语法解析。在OceanBase中采用开源软件Flex与Bison完成词法解析与语法解析。Flex进行词法分析，这需要我们设计过程语言的语法后，为其设计正则表达式来匹配源码中出现的标记（常量，数学符号，括号，中括号，变量名，保留字等），并定义规则将标记映射为内部标志符。之后，Bison来对词法分析后的内部标志符序列进行语法分析，形成抽象语义树。我们需要为每个语义单元设计识别规则，并为其创建一个相应的结构保存。

2.1.1 词法解析

词法分析的主要任务是从程序源码(ASCII码)中提取分析标识符(Token)，以提供给后续的语法分析程序使用，同时词法分析还要对符号表操作。

词法分析程序完成的是编译第一阶段的工作，在语法解析器中它被设计成一个子程序，每当词法分析需要一个单词时，就调用该子程序。词法分析程序从源程序文件中读入一些字符，直到识别出一个单词，或者直到下一个单词的第一个字符为止。这种方式，使得词法分析程序和语法分析程序放在同一遍中，即在此两个阶段中，只需要对源输入语句进行一次扫描。

作为编译程序的输入，无论是SQL语句还是过程语句都仅仅是由一个个常数组成的字符串，词法分析程序需要将这种形式的语句转化为便于编译程序其余部分进行处理的内部格式。因此词法分析程序需要完成的任务如下：

- 识别出源程序的各个语法单位；
- 滤除无用的空白字符、制表符、回车字符以及其他与输入介质相关的非实质性字符；
- 过滤掉源程序的注释；
- 进行词法检查，如果出现错误，记录出错信息并报告；

其中提到的语法单位即是前面提到的标识符(Token)。一般来说，程序设计语言单词符号可以分为五种，它们分别是基本字（也称为关键字）、标识符、常数、运算符以及界符。经由词法分析程序识别出的单词符号输出时常常采用二元式表示：（单词种别，单词自身的值）。单词种类是语法分析需要的信息，而单词自身的值则是编译其它阶段需要的信息。

一般来说，程序设计语言单词的语法都能用正则表达式来描述。因此，词法分析大都是将用来说明单词结构的正则表达式编译或转化为相应的有限状态自动机(FSA:Finite State Automation)，最终通过构造的有限状态自动机识别出被正则表达式说明的单词。上述过程可用图2.3表示：

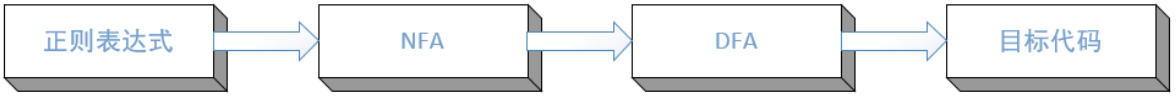

图2.3 正则表达式与有穷自动机

2.1.2 语法分析

语法分析程序是存储过程预编译阶段的核心部分。语法分析程序的作用是识别由词法分析程序给出的单词符号（Token）序列是否复合存储过程给定的文法的正确句子。具体来说，它的任务包括在由词法分析程序产生的符号序列中确定存储过程的语法结构，以及构造出表示该语法结构的语法树。由此可以看出语法分析也可以看做一个函数，该函数把由词法分析生成的符号序列作为输入，并以生成的语法树作为它的输出。这里提到的符号序列不是显示的输入参数，而是当语法分析过程需要下一个符号的时候，调用设计成函数的词法分析程序，从而得到所需的符号。

语法树的结构在很大程度上依赖与存储过程特定的语法结构，它被定义为动态数据结构。该结构的每个节点都有一个记录组成，而这个记录的域包括了编译后面过程所需的特性。

2.1.3 语义分析

语义分析的主要任务就是对语法分析所识别出的各类语法范畴，分析其含义。在预编译阶段，编译系统只能完成静态语义分析（Static Semantic Analysis）。在存储过程模块中，语义分析包括构造符号表、记录函数参数、变量声明中建立的名字含义，在表达式和语句中进行类型推断和类型检查以及在变量的类型转换作用域内判断它们的正确性。

语法分析和语义处理可以通过使用编译程序自动产生工具Bison来实现。Bison输入用户提供的语言的语法描述规格说明，基于LALR语法分析的原理，自动构造一个改语言的语法分析器，同时它还能根据规格文件中给出的语法规范建立相应的语义检查。

借助于Flex与Bison，可以构造出一个完整、规范、高效的编译程序。Flex生成的词法扫描器从输入文本流中找到一种语言的词汇模式，然后在动作代码中返回代表该词汇模式的标记（Token）；Bison生成的语法解析器程序则接收由词法扫描器返回的标记，多个标记可以形成标记序列，这些标记序列就可以表述出输入文本流中的词法模式。Flex与Bison相结合，流程图如图2.4所示，构造并输出一颗表示语法结构的语法树。

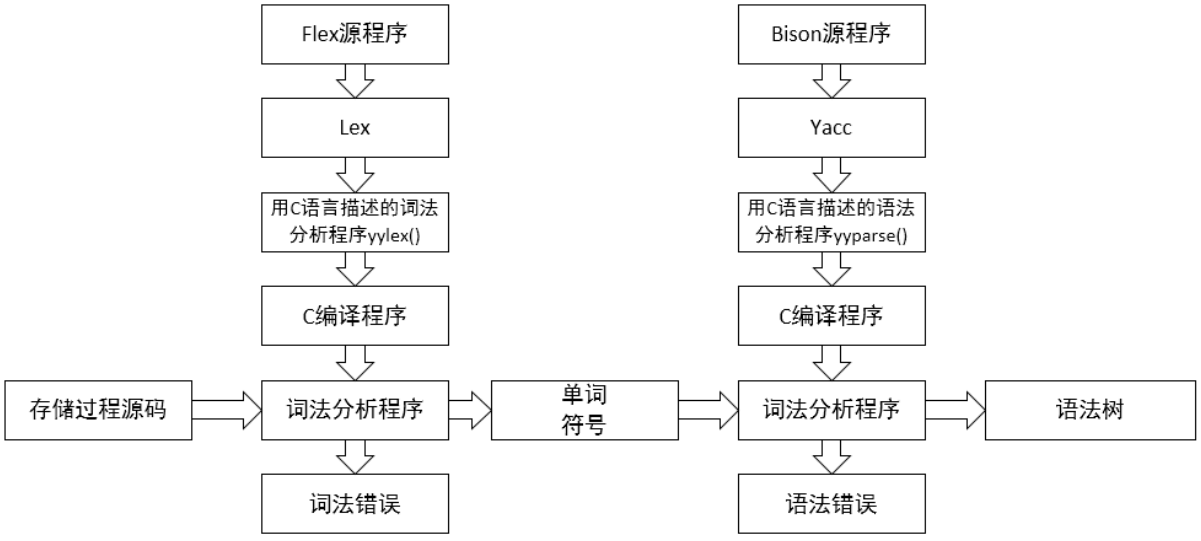

图2.4 flex和bison进行语法词法解析

2.1.4 OceanBase中的词法语法解析

存储过程在Oceanbase中是一组新加入的语法，因此我们需要为存储过程进行语法词法解析，在Oceanbase中SQL语句的解析使用的是Flex&Bison[1]，因此法语法的解析这一部分就同样使用Flex&Bison来实现。OceanBase 的语法树节点结构体只有一个,该结构体包括一个枚举类型变量type,和PostgreSQL一样，代表该结构体对应的类型。还有两组属性，对应终止符节点的64位整型值和字符串值；非终止符节点使用了后两个字段一个表示子节点数量，一个指向子节点数组的首地址。OceanBase的语法树节点数据结构如代码L1所示。

```
typedef struct _ParseNode
{
  ObItemType   type_;

  /* 终止符节点的真实的值 */
  int64_t      value_;
  const char*  str_value_;

  /* 非终止符节点的孩子节点*/
  int32_t      num_child_; /*孩子节点的个数*/
  struct _ParseNode** children_;

  // BuildPlanFunc m_fnBuildPlan;
} ParseNode;
```

对应一个节点而言，要么是终止符节点要么是非终止符节点，它只会使用两组属性中的一组。int,long,float,double,string等都是终止符类型，可以看出int,long都是用64位整形int64表示。float,double,string则用char *字符串表示。终止符的`num_child_`为0,`children_`为null.

语法树的节点的设计，主要是为了解决如何表达语法结构。不同的数据库有不同的具体实现。OceanBase采用终止符和非终止符分类，使OceanBase的设计极具灵活性，但使用时需要仔细验证，避免出错。存储过程在OceanBase中进行词法语法解析过后生成一颗语法树，图2.5是创建存储过程的语法树。

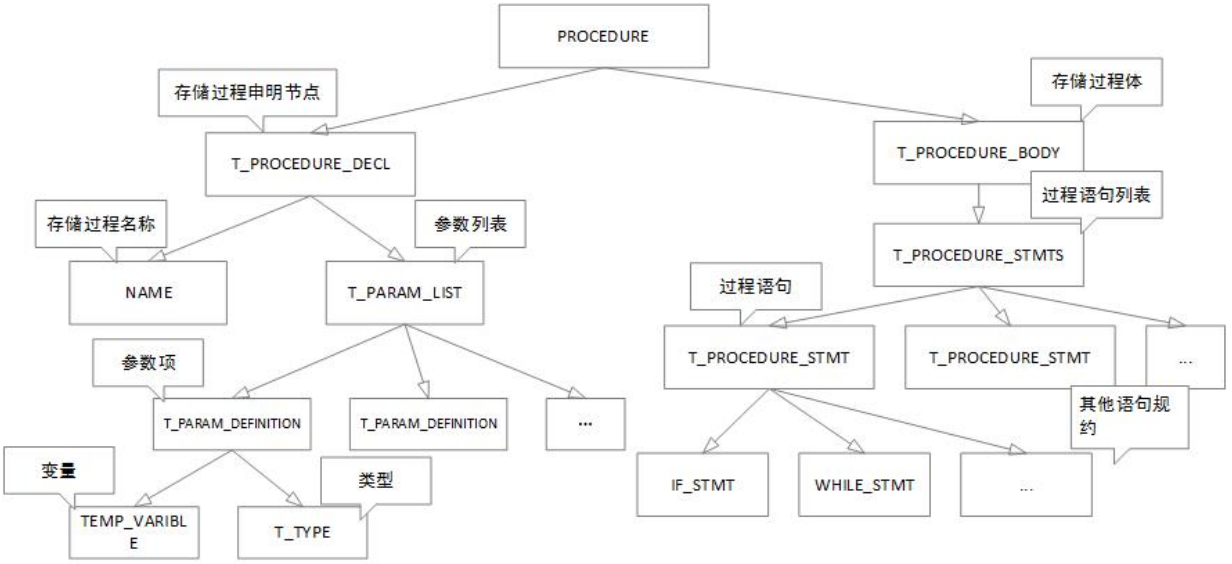

图2.5 存储过程的语法树

> 注意: 其中T_PROCEDURE_STMT是各种类型语句的结构，有声明语句、赋值语句、IF语句、CASE语句、WHILE语句、SQL语句。

一个存储过程的语法树包含两个孩子节点，T_PROCEDURE_DECL节点是对存储过程的定义进行描述的节点，主要包括存储过程名称，存储过程的参数名以及类型。其中存储过程的T_PROCEDURE_BODY是存储过程的过程体节点，T_PROCEDURE_STMT是各种类型语句的结构，有声明语句、赋值语句、IF语句、CASE语句、WHILE语句、SQL语句以及CURSOR相关的语句。对于存储过程中不同的流程控制语句的节点也都不一样，但是都包含了一个能够进行规约的T_PROCEDURE_STMT节点，这使得存储过程的流程控制语句是可以相互嵌套的，也可以嵌套非流程控制语句在里面。例如，IF语句的语法树节点包含三个孩子节点，一个是EXPR节点表示IF语句的判定表达式，另一个是ELSE IF节点表示else if语句的列表，最后一个是表示ELSE语句的节点，每个分支语句内是一个规约的T_PROCEDURE_STMT节点，IF语句的结构如图2.6所示：

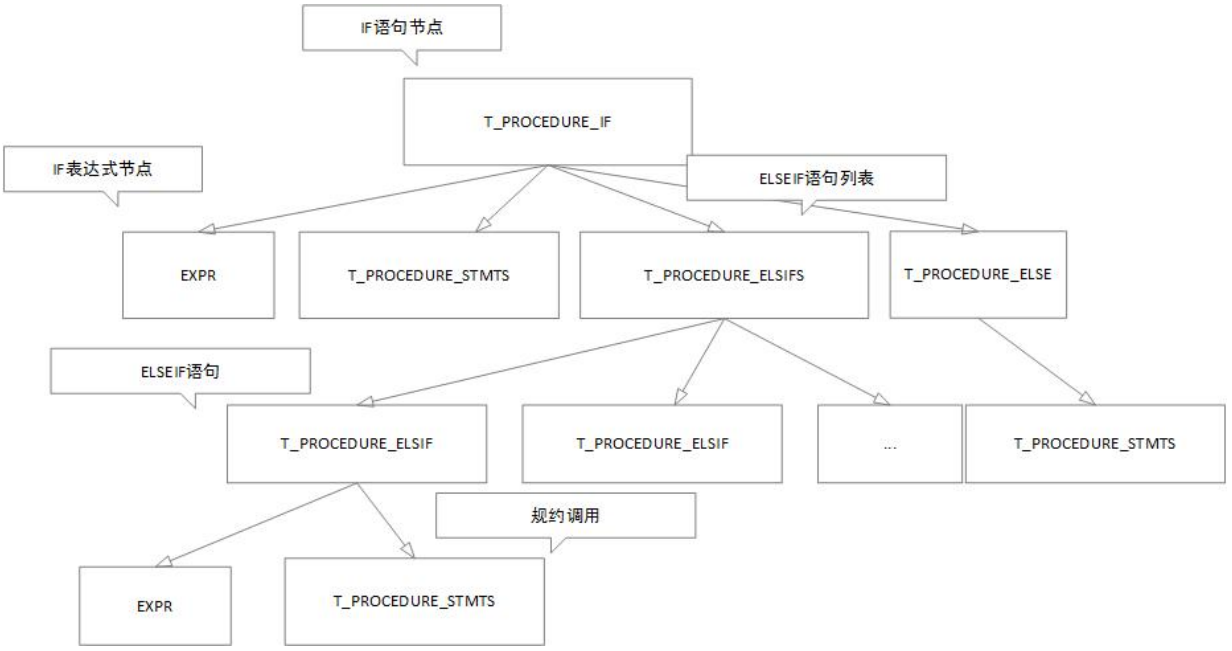

图2.6 IF语句语法树节点

> 注意: 其中T_PROCEDURE_STMT是各种类型语句的结构，有声明语句、赋值语句、IF语句、CASE语句、WHILE语句、SQL语句。

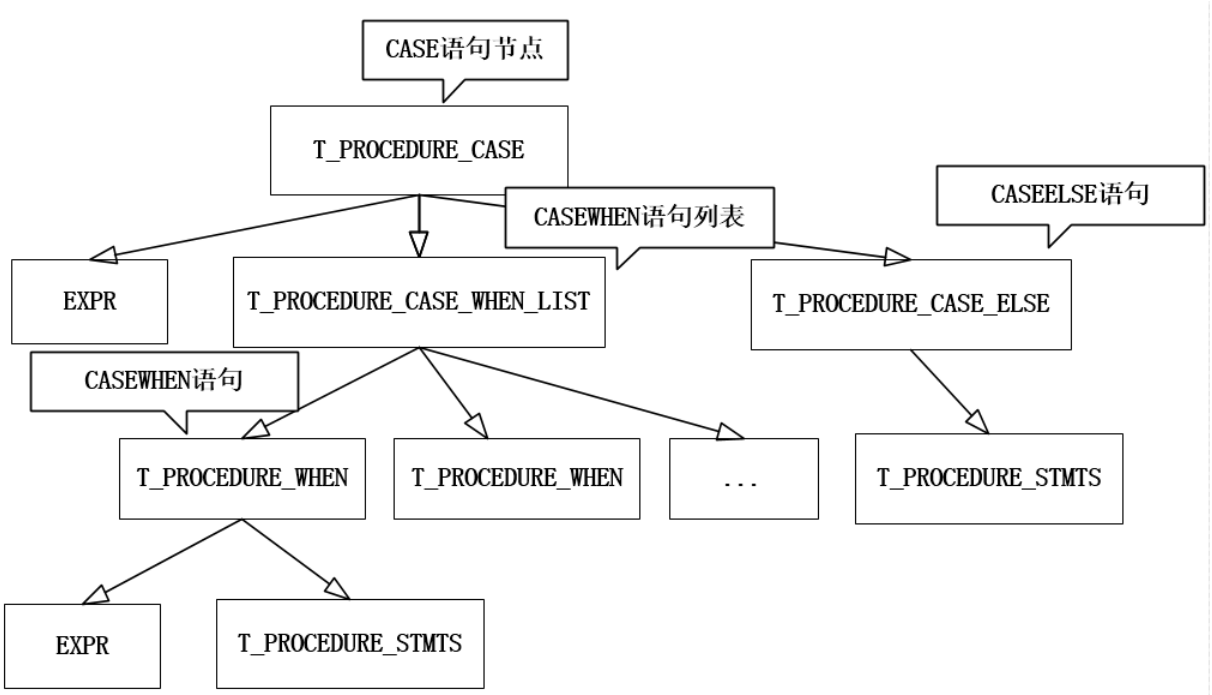

图2.7 CASE语句语法树节点

注意: 其中T_PROCEDURE_STMT是各种类型语句的结构，有声明语句、赋值语句、IF语句、CASE语句、WHILE语句、SQL语句。

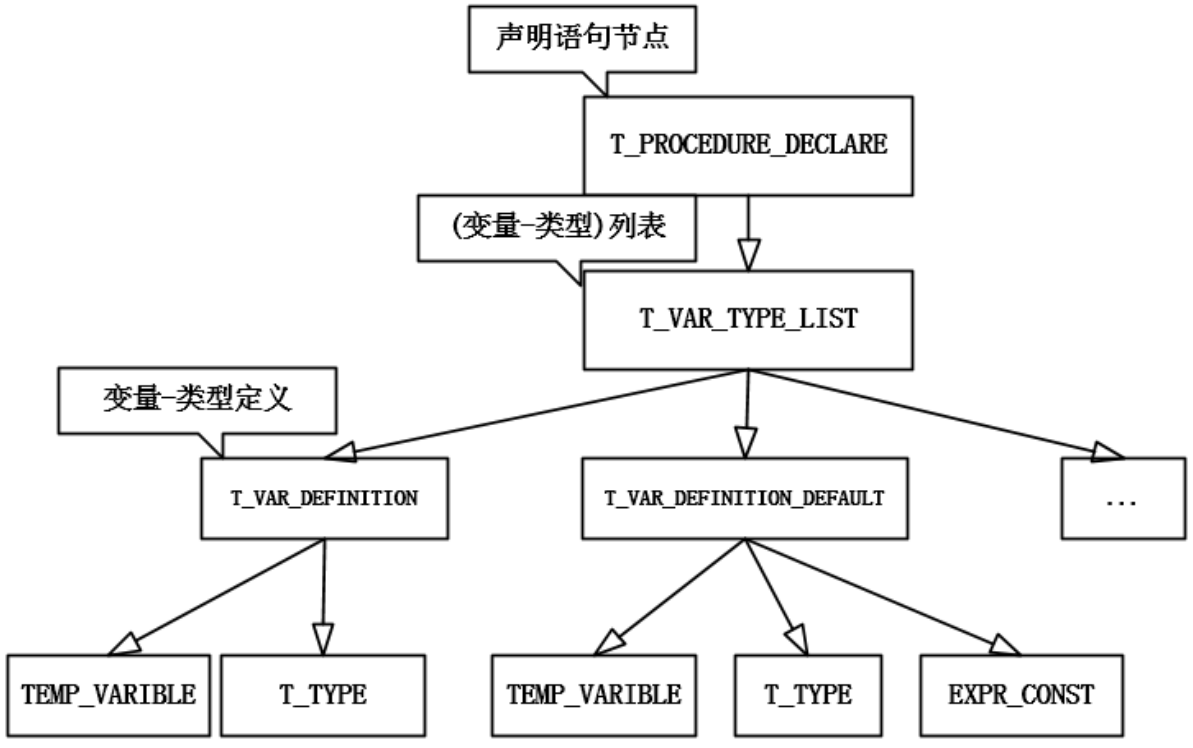

图2.8 DECLARE语句语法树节点

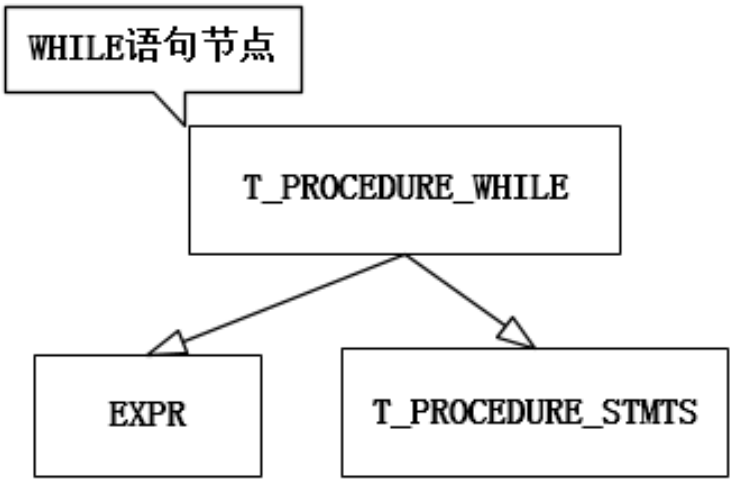

图2.9 WHILE语句语法树节点

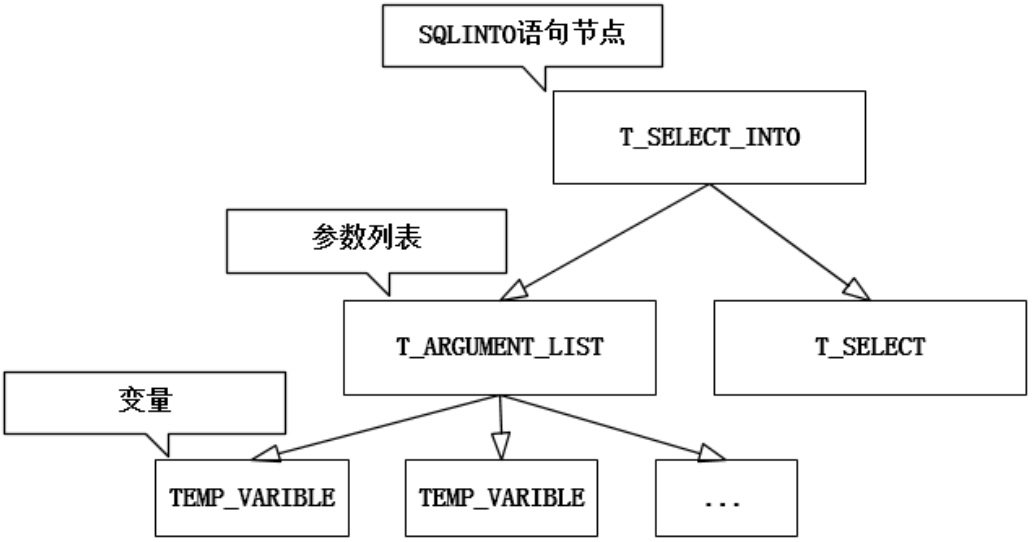

图2.10 WHILE语句语法树节点

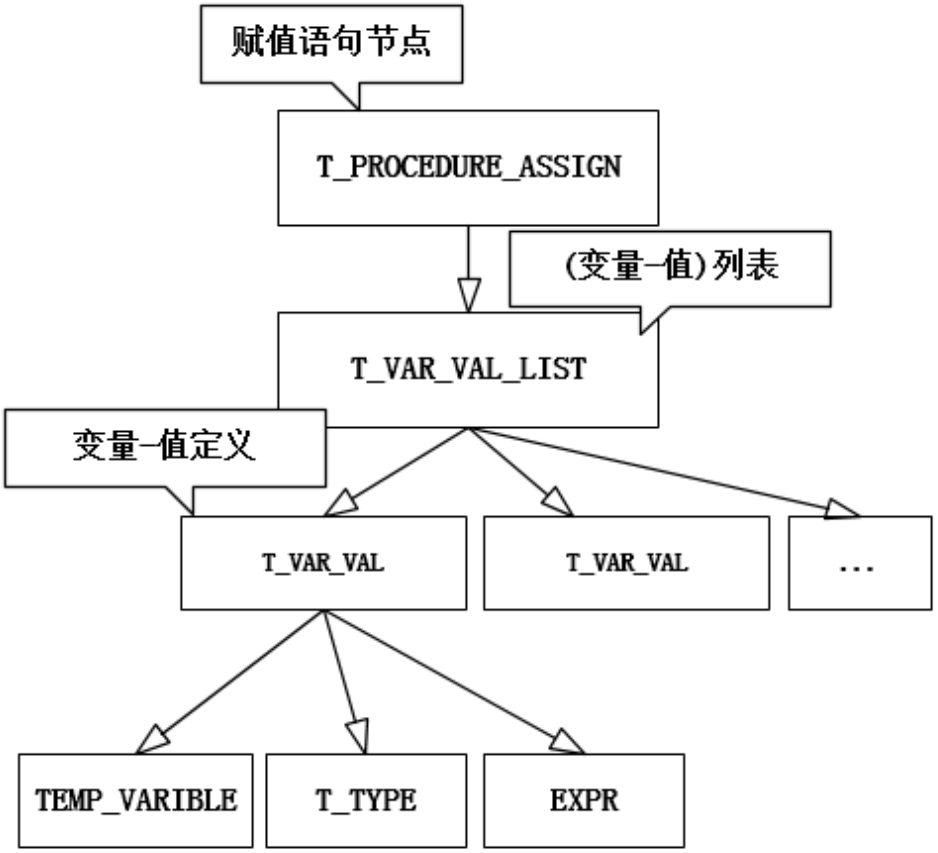

图2.11 ASSGIN语句语法树节点

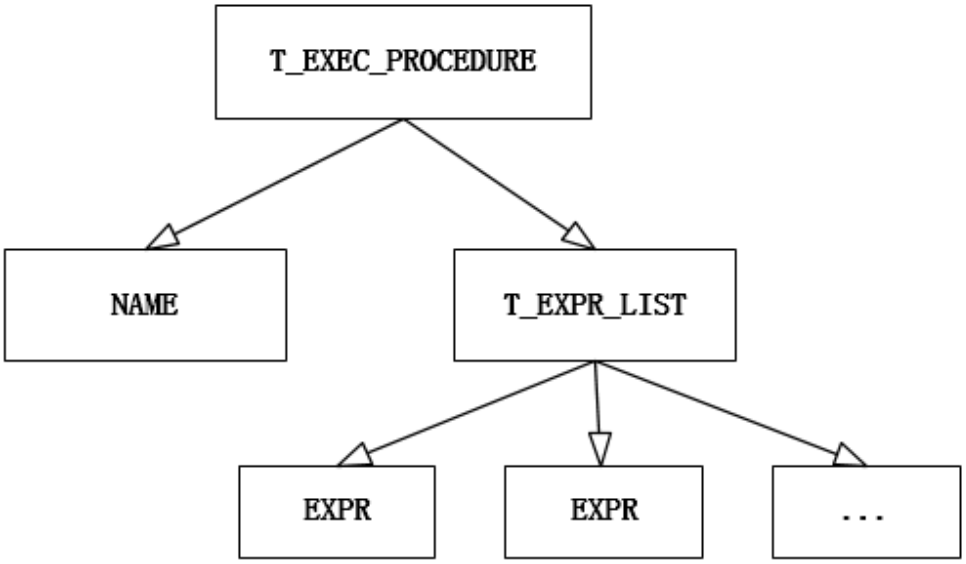

图2.12 CALL语句语法树节点

2.2 逻辑计划生成子模块设计

SQL语句在经过词法语法解析构建了语法树后仅仅只能判断这个SQL语句的写法是否正确，不能确定SQL语句是否可以执行。存储过程语句中包含了SQL语句，所以在存储过程构建语法树后需要生成逻辑计划，逻辑计划需要明确SQL语句中所涉及到的表，字段，表达式等是否有效，OceanBase中的逻辑计划与在《数据库系统实现》等书中描述的逻辑查询计划不同的是OceanBase中的逻辑计划只是查找或生成涉及到的表的ID，字段ID的表达式ID等，而不能将SQL语句转为可运算的关系表达式。

2.2.1 逻辑计划结构

在OceanBase中每一个语句的逻辑计划都是继承ObStmt这个类，ObStmt表示一个单独的查询所包含的内容，一个逻辑计划可以包含多个ObStmt。代码L2所示是ObStmt类的基本结构。

```
class ObStmt
{
    /*省略部分内容*/
    protected:
        common::ObVector<TableItem> table_items_;
        common::ObVector<ColumnItem> column_items_;

    private:
        StmtType type_;
        unit64_t query_id_;
        common::ObVector<unit64_t> where_expr_ids_;
}
```

ObStmt包括了一个查询所有的表`table_items_`,列`column_items_`,表达式`where_expr_ids`和一个唯一的查询标识`query_id_`。注意这里存储的只有表达式的id，而不是表达式的实际内容。

从上述的定义总结来看，一个逻辑计划拥有多条查询实例ObStmt和多个表达式，一个查询实例ObStmt包含了多个表和多个列及所需表达式的引用。表，列，表达式，查询实例都有唯一的标识符进行标记。

2.2.2 逻辑计划生成算法

生成逻辑计划的步骤是在存储过程语句构建了语法树后，依次的从语法树上取出节点，根据节点的类型调用不同的函数处理，存储过程的语法树中有流程控制语句和SQL语句节点，在生成逻辑计划的时候需要递归的调用函数生成规约节点的逻辑计划。在OceanBase中我们增加了一系列的resolve函数来生成存储过程对应的逻辑计划，如代码L3所示。

```
int resolve_procedure_stmt(...) {...}

int resolve_procedure_create_stmt(...) {...}

int resolve_procedure_execute_stmt(...) {...}

int resolve_procedure_declare_stmt(...) {...}

int resolve_procedure_assgin_stmt(...) {...}

int resolve_procedure_if_stmt(...) {...}

int resolve_procedure_elseif_stmt(...) {...}

int resolve_procedure_else_stmt(...) {...}

int resolve_procedure_while_stmt(...) {...}

int resolve_procedure_case_stmt(...) {...}

int resolve_procedure_casewhen_stmt(...) {...}

int resolve_procedure_select_into_stmt(...) {...}
```

入口函数

```
int resolve(ResultPlan* result_plan, ParseNode* node)
{
    /*此处省略部分代码*/
    switch (node->type_) /*根据节点的类型来生成逻辑计划*/
    {
      case T_SELECT:
      {
        ret = resolve_select_stmt(result_plan, node, query_id);
        break;
      }
      case T_INSERT:
      {
        ret = resolve_insert_stmt(result_plan, node, query_id);
        break;
      }
      case T_PROCEDURE_CREATE:
      {
         ret = resolve_procedure_create_stmt(result_plan, node, que
ry_id);
         break;
      }
      case T_PROCEDURE_DROP:
      {
           ret = resolve_procedure_drop_stmt(result_plan, node, que
ry_id);
           break;
      }
      case T_PROCEDURE_EXEC:
      {
         ret = resolve_procedure_execute_stmt(result_plan, node, qu
ery_id);

         break;
      }
      /*此处省略部分代码*/
}
```

2.2.3 逻辑计划生成流程

生成语法树过后调用`resolve(ResultPlan* result_plan, ParseNode* node)`函数对生成的语法树进行遍历，并调用与当前节点类型匹配的resolve函数生成对应节点的逻辑计划，resolve系列函数中也是会对一些有递归结构的节点进行递归的生成。

2.3 物理计划生成子模块设计

物理查询计划能够直接执行并返回数据结果数据。它包含了一系列的基本操作，比如选择，投影，聚集，排序等。因此，本质上，物理查询计划是一系列数据操作的有序集合。在OceanBase中的物理查询计划由一系列的运算符构成。SQL语句在经过构建语法树和逻辑计划生成后，还需要进一步处理，即生成物理执行计划。在OceanBase中生成物理执行计划的时候，首先获取对应的逻辑计划树，从逻辑计划树中取出一个Statement，根据Statement类型调用相应处理函数生成对应的物理操作符，一个物理操作符包含一个或多个孩子节点，每个物理操作符有open、close、get_next_row等函数组成，open函数是具体执行的入口函数。

存储过程中有declare、assign、if、case、while等物理操作符，每个操作符下面有一个或多个物理操作符，它们的嵌套关系和语法树中一样，可能会有规约的节点。当我们在生成物理计划的过程中如果需要生成SQL语句的物理计划，我们直接调用系统中原有的函数生成物理操作符，并把它作为父流程控制语句物理操作符的子操作符。最终存储过程生成的物理计划是一个包含多个物理操作符的物理计划树。

2.3.1 物理计划结构

在OceanBase中，物理运算符接口为ObPhyOperator。其定义如代码L4所示。

```
/// 物理运算符接口
class ObPhyOperator
{
    public:
        ///打开物理运算符，申请资源，打开子运算符，构造row description
        virtual int open();
        ///关闭物理运算符，释放资源，关闭子运算符等。
        virtual int close();
        ///获取下一行的引用
        virtual int get_next_row(const common::ObRow *&row);
}
```

ObPhyOperator定义了open,close,get_next_row等3个函数用于实现运算符的流水化操作。并根据子节点的个数定义了几种类型的运算符，它们都继承自ObPhyOperator。

- ObNoChildrenPhyOperator:无子节点的运算符
- ObSingleChildPhyOperator:只有一个子节点的运算符
- ObDoubleChildrenPhyOperator：有两个子节点的运算符
- ObMultiChildrenPhyOperator:有多个子节点的运算符（0.4版本才出现的）
  此外还有:
- ObRowkeyPhyOperator:带返回RowKey的运算符,也就是返回的时候不是返回Row，而是返回RowKey。 磁盘表扫描运算符ObSstableScan继承自该类。

ObNoRowsPhyOperator:无返回列的运算符,如插入运算符ObInsert继承自该类
以上几个运算符依然是接口部分，真正使用时的运算符如同在关系代数中所说的一般，但SQL语句并不是完全的关系代数运算，为了方便实现时都会定义更多的运算符。

存储过程的物理操作符主要是继承自ObMultiChildrenPhyOperator操作符，存储过程中多种类型语句生成的物理操作符作为root操作符的child，执行的时候循环打开root操作符的子操作符。

2.3.2 物理计划生成算法

在OceanBase中逻辑计划转换成物理计划，主要是通过ObTransformer类来进行转换，辑计划生成物理查询计划或物理操作符的操作由下面gen_physical系列函数完成,如代码L5所示。

```
int gen_physical_procedure(...) {...}

int gen_physical_procedure_create(...) {...}

int gen_physical_procedure_execute(...) {...}

int gen_physical_procedure_if(...) {...}

int gen_physical_procedure_elseif(...) {...}

int gen_physical_procedure_else(...) {...}

int gen_physical_procedure_dclare(...) {...}

int gen_physical_procedure_assgin(...) {...}

int gen_physical_procedure_while(...) {...}

int gen_physical_procedure_case(...) {...}

int gen_physical_procedure_casewhen(...) {...}

int gen_physical_procedure_select_into(...) {...}
```

生成物理计划的主函数

```
int generate_physical_plan(ObLogicalPlan *logical_plan,ObPhysicalPl
an*& physical_plan,...)
{
    /*省略部分代码*/
    switch (stmt->get_stmt_type())
    {
    case ObBasicStmt::T_SELECT:
        ret = gen_physical_select(logical_plan, physical_plan ...,
index);
        break;
        /*省略部分代码*/
```

```
    case ObBasicStmt::T_PROCEDURE_CREATE:
        ret=gen_physical_procedure_create(logical_plan, physical_pl
an ..., index);
        break;
    case ObBasicStmt::T_PROCEDURE_DROP:
        ret=gen_physical_procedure_drop(logical_plan, physical_plan
..., index);
        break;
    case ObBasicStmt::T_PROCEDURE_EXEC:
        ret=gen_physical_procedure_execute(logical_plan, physical_p
lan ..., index);
        break;
    case ObBasicStmt::T_PROCEDURE_IF:
        ret=gen_physical_procedure_if(logical_plan, physical_plan
..., index);
            break;
    case ObBasicStmt::T_PROCEDURE_DECLARE:
        ret=gen_physical_procedure_declare(logical_plan, physical_p
lan, ..., index);
        break;
    case ObBasicStmt::T_PROCEDURE_ASSGIN:
        ret=gen_physical_procedure_assgin(logical_plan, physical_pl
an, ..., index);
        break;
    case ObBasicStmt::T_PROCEDURE_WHILE:
        ret=gen_physical_procedure_while(logical_plan, physical_pla
n, ..., index);
        break;
    case ObBasicStmt::T_PROCEDURE_CASE:
        ret=gen_physical_procedure_case(logical_plan, physical_pla
n, ..., index);
        break;
    case ObBasicStmt::T_PROCEDURE_SELECT_INTO:
        ret=gen_physical_procedure_select_into(logical_plan, physic
al_plan, ..., index);
        break;
}
```

2.3.3 物理计划生成流程

在生成语法树和逻辑计划过后过后调用`generate_physical_plan(...)`函数对生成的逻辑计划进行遍历，根据`stmt_id`从逻辑计划中取出对应的ObStmt对象，并根据ObStmt的类型调用相应的函数生成物理运算符，并将生成的ObPhyOperator运算符添加到物理计划中，或者作为当前ObPhyOperator的子运算符，gen_physcial系列函数会对有递归结构的ObStmt对象递归的调用物理计划生成函数。

3 详细设计

3.1 OceanBase中存储过程的基本处理框架

OceanBase中我们设计的存储过程是按照mysql规范来设计语法规则的，主要包含的语句是CREATE、DELETE、CALL，每一个都是单独作为一条SQL语句处理的。

CREATE语句表示声明一个存储过程和对应的存储过程体，存储过程体可以包含多条语句例如select、insert、update等原生sql，也可以是存储过程支持的if、while、case、declare、assign等逻辑语句，我们会把声明的存储过程存入一张系统表proc_store进行保存。CALL语句根据存储过程名称从数据库中查询出改存储过程的sql语句进行语法解析，生成逻辑计划，生成物理执行计划，然后在send_resonpse的时执行物理计划。这几种语句的处理流程大致相同，下面我以创建存储过程的CREATE语句为例说明存储过程的处理框架，如图3.1所示。

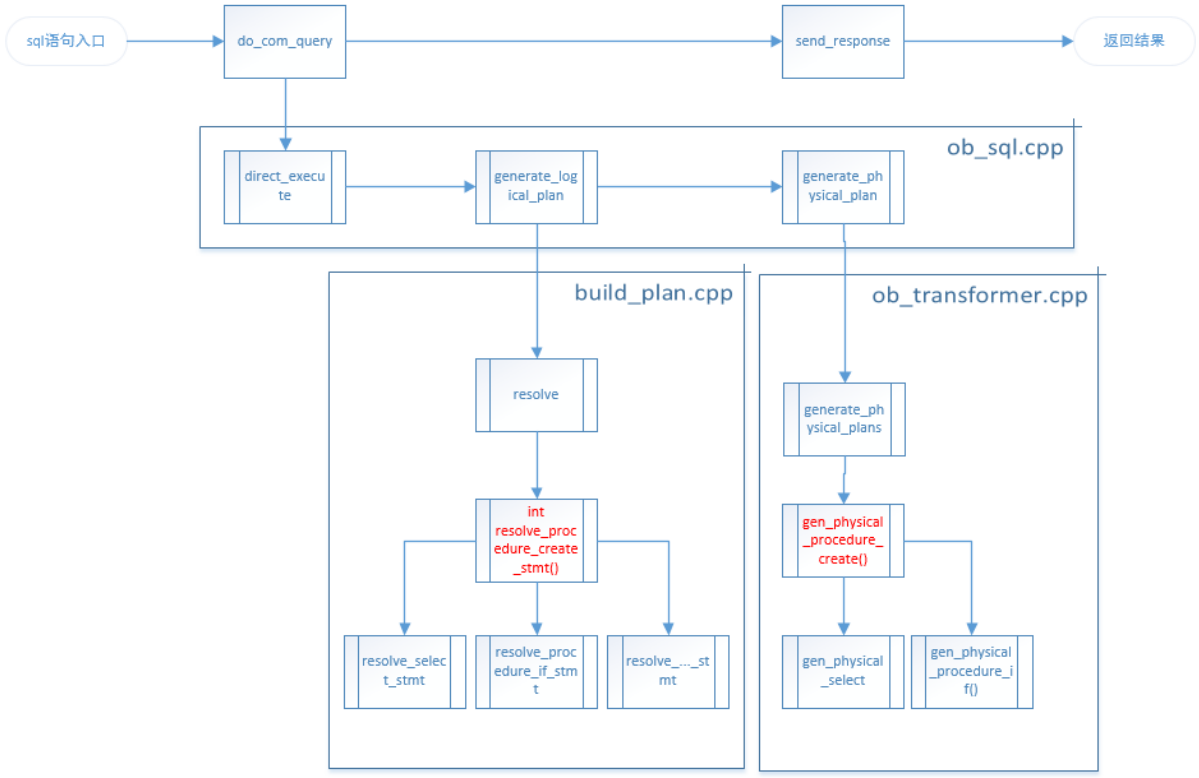

图3.1 存储过程处理框架图

如上图所示，以CREATE语句为例，介绍了存储过程整个处理流程，以及所调用的函数。生成逻辑执行计划部分的函数是需要新增的函数，sql语句传输到服务器之后，由do_com_query函数进行查询处理，它会调用direct_execute函数，direct_execute再调用generate_logical_plan、generate_physical_plan来生成逻辑计划和物理计划，我们分别新增resolve_procedure_系列函数和generate_physical_procedure_系列函数来生成逻辑执行计划和物理执行计划。然后do_com_query调用send_response函数执行物理计划并返回结果。

3.2 CREATE PROCEDURE的实现

本节主要介绍创建存储的整个处理过程，包括词法语法编译，逻辑计划生成和逻辑计划树的结构，物理计划生成和物理计划树的结构。

3.2.1 CREATE语句语法

```
CREATE PROCEDURE 存储过程名([参数#1,…参数#1024])
BEGIN
    [statement_list]
END;
```

> Tips:具体的详细的语法可以参考《存储过程使用手册》

3.2.2 CREATE语句的语法树结构

如图3.2所示，OceanBase在处理CREATE语句的时候首先会把符合上述所示语法的sql语句进行词法语法分析，并解析成为T_PROCEDURE_CREATE类型的node，它有两个节点，child[0]是用来存放存储过程名称以及参数的，child[1]是存储statement_list的，statement_list是由存储过程的if，while，case，select into等过程语句以及普通sql语句构成。

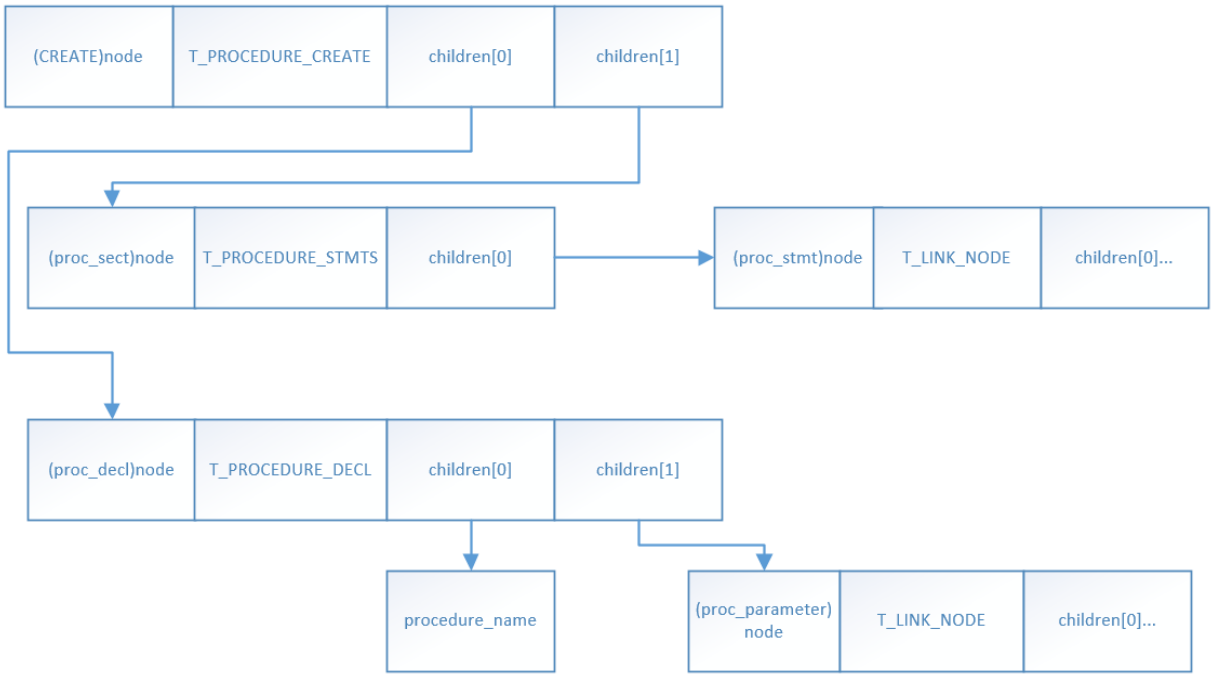

图3.2 存储过程CREATE语法树

> Tips：图*3.1*与图*2.5*中是一样的只是图的反映方式不一样

3.2.3 CREATE逻辑计划

OceanBase中，Logical_plan类用来存储逻辑计划，它运用ObVector结构来存储各个statement类，每个stmt类都存储有自己statement的逻辑计划相关信息，如query_id，stmt_type等。

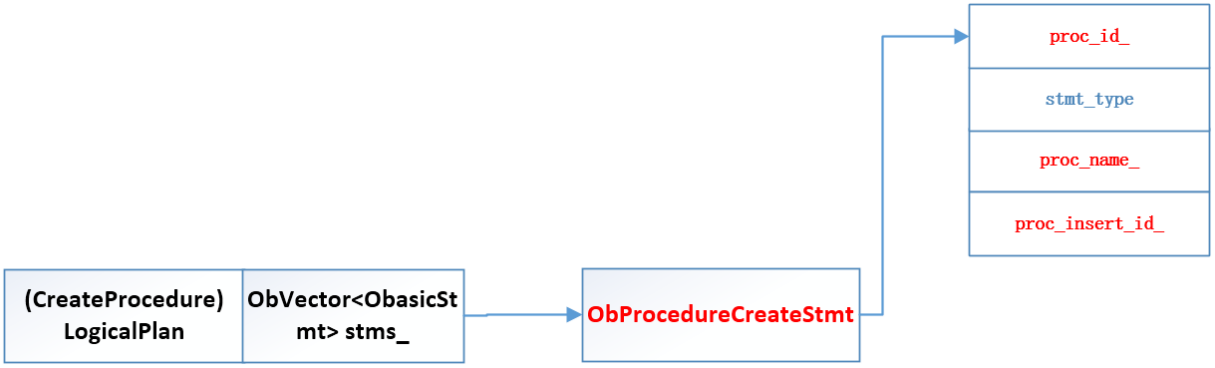

图3.3 存储过程CREATE逻辑计划树

> Tips：图中红色部分是所要创建的类或者需要新建的变量。

上述图3.3所示CREATE的逻辑计划树主要包含三个字段，`proc_id_`是存储过程体的一个标识符，其中存储过程体ProcedureStmt的逻辑计划树如图5.1-b所示。`proc_name_`是表示要创建的存储过程的名称，`prco_insert_id_`是将存储过程插入系统表的insert语句对应的逻辑计划树的标识。

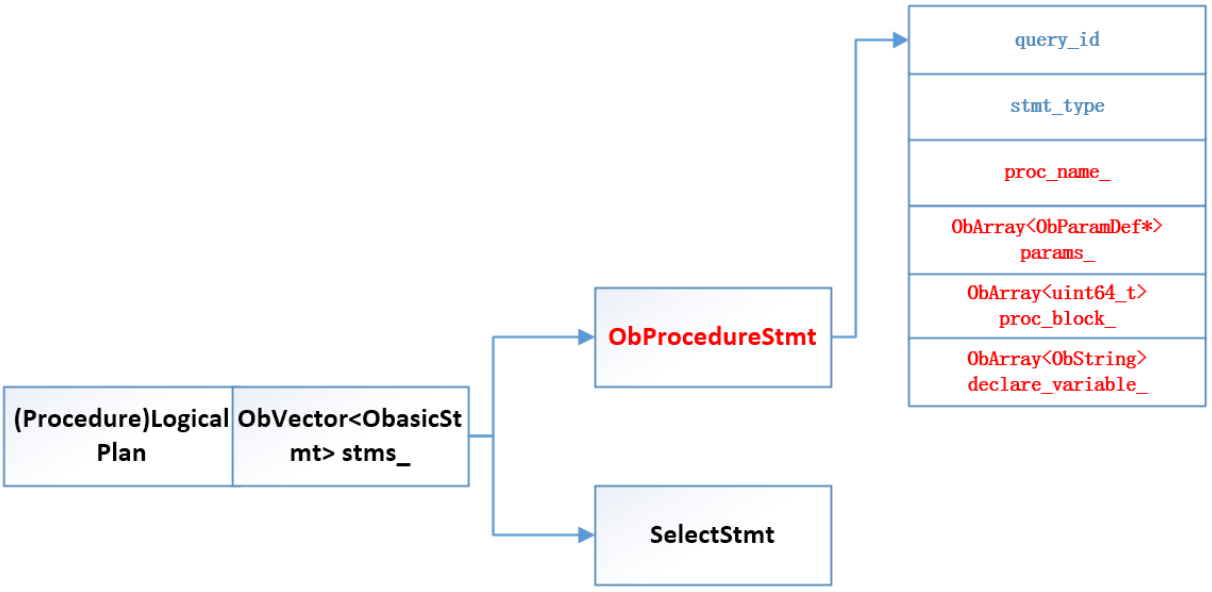

图3.4 存储过程逻辑计划树

上述图3.4所示的是存储过程体ProcedureStmt的逻辑计划树，存储过程体是由多条逻辑语句以及sql语句构成的，所以prco_block_存储的是对应语句的标识符，params_是存储创建存储过程时声明的参数。

> Tips：图中红色部分是所要创建的类或者需要新建的变量。

3.2.4 CREATE物理计划

如图3.5所示，CREATE语句的物理计划包含多个物理操作符，ObProcedure、ObInsert等。其中ObInsert是创建存储过程的时候把存储过程的sql语句存储到系统表prco_store的。ObProcedure是存储过程其他的逻辑语句以及sql语句的物理操作符的父节点，它是一个包含多个操作的符的物理操作符。

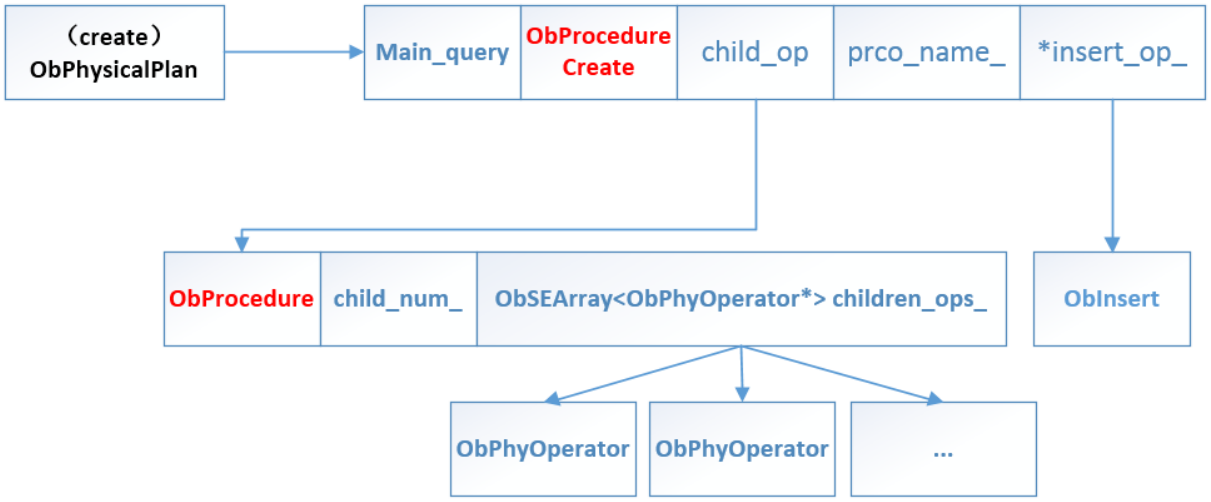

图3.5 CREATE物理计划树

3.2.5 ObProcedureCreate类

主要成员变量

```
ObString proc_name_;
ObPhyOperator *insert_op_;
```

主要成员函数

```
int set_proc_name(ObString &proc_name);
int set_insert_op(ObPhyOperator &insert_op);
```

> Tips：只列举出了主要的成员变量和成员函数

3.2.6 ObProcedure类

主要成员变量

```
int32_t child_num_;/*子操作符的数量*/
ObString proc_name_;/*存储过程名称*/
ObArray<ObParamDef*> params_;/*存储过程参数*/
ObArray<ObString> declare_variable_;/*存储过程中定义的变量*/
```

主要成员函数

```
int set_proc_name(ObString &proc_name);/*设置存储过程名*/
int add_param(ObParamDef &proc_param);/*添加一个存储参数*/
int set_params(ObArray<ObParamDef*> &params);/*存储过程参数*/
int add_declare_var(ObString &var);/*添加一个变量*/
ObArray<ObParamDef*>& get_params();
ObParamDef* get_param(int64_t index);
ObString& get_declare_var(int64_t index);
int64_t get_param_num();
int64_t get_declare_var_num();
```

3.3 存储过程IF语句的实现

本节主要介绍创建存储中的if-else-elseif语句处理过程，包括词法语法编译，逻辑计划生成和逻辑计划树的结构，物理计划生成和物理计划树的结构。

3.3.1 IF语句语法编译

```
IF search_condition THEN statement_list
    [ELSEIF search_condition THEN statement_list] ...
    [ELSE statement_list]
END IF
```

> Tips:具体的详细的语法可以参考《存储过程使用手册》

3.3.2 IF语法树结构

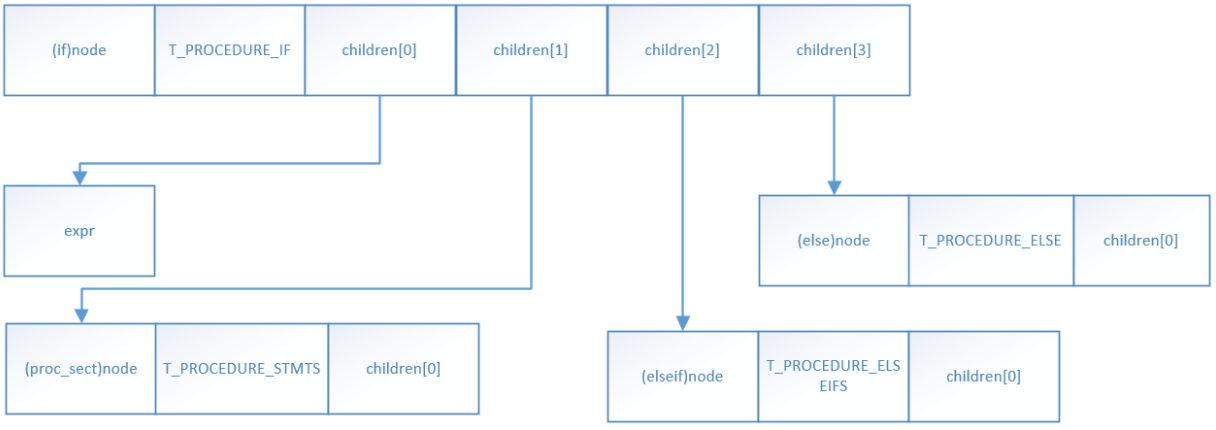

图3.6 存储过程CREATE语法树

> Tips：图*3.6*与图*2.6*中是一样的只是图的反映方式不一样

如图3.6所示，对于存储过程中的IF语句，我们生成T_PROCEDURE_IF类型的node做为父节点，父节点有4个子节点child[0]是IF语句的表达式节点，里面存储的是一个表达式，child[1] 是statement_list的，statement_list是由存储过程的if，while，case，select into等过程语句及普通sql语句构成。child[2]是一个T_PROCEDURE_ELSEIFS节点构成的，它是一个链表每个节点是一个ESLEIF节点，child[3]是一个T_PROCEDURE_ELSE节点。上面的节点除了第一个外之外，每一个节点的内部有可以嵌入其它类型语句的节点。

3.3.3 IF逻辑计划

如图3.7所示，IF语句的逻辑计划树是比较复杂的，LogicalPlan里面的ObProcedureIfStmt类记录了一个if-else-elseif语句的表达式id，elseif语句，else语句等信息。

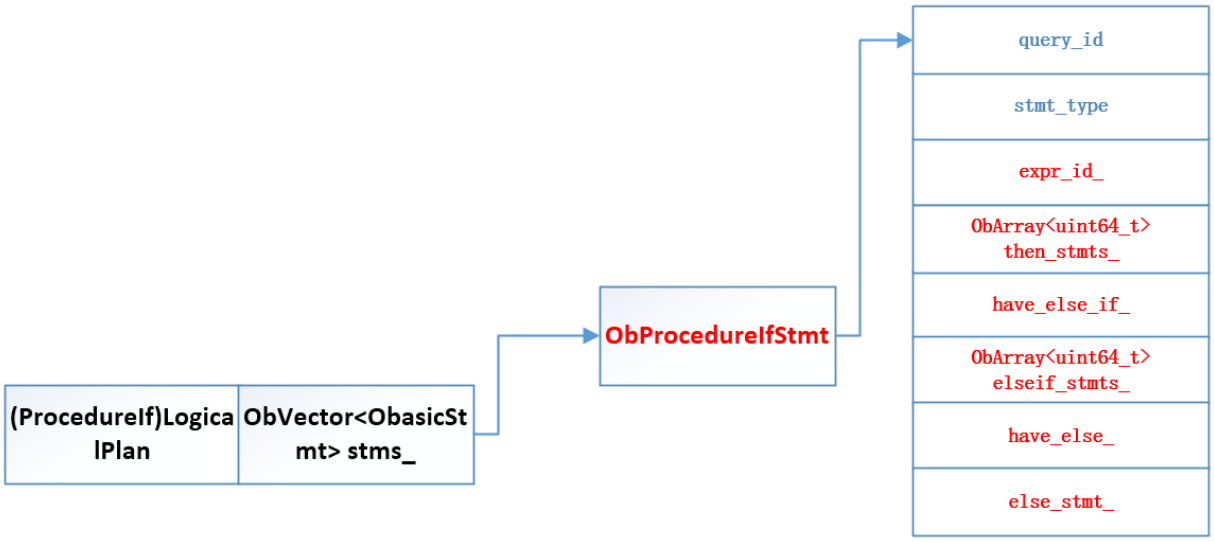

图3.7 IF逻辑计划树

3.3.4 IF物理计划

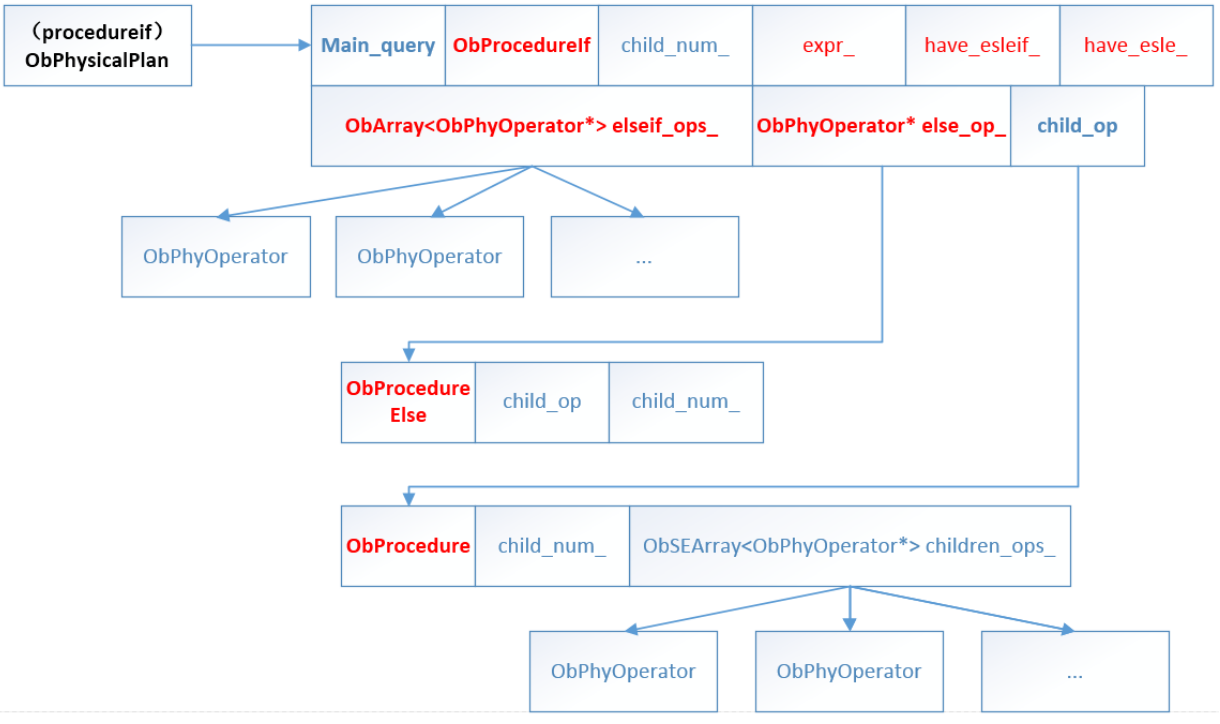

图3.8 IF物理计划树

如图3.8所示，I计划生成IF的物理计划，其中ObProcedureIf物理操作符中expr_是一个ObSqlExpression类型的表达式，have_elseif_标识是否拥有elseif分支，have_else_标识是否有else语句分支，elseif_ops_是存储if语句的所有elseif分支的物理操作符，else_op_是else语句分支的物理操作符。

3.3.5 ObProcedureIf类

主要成员变量

```
int32_t child_num_;     /*子操作符的数量*/
ObSqlExpression expr_; /*if语句的表达式*/
bool have_elseif_;      /* 是否有else if物理操作符*/
ObArray<ObPhyOperator*> else_ops_;/*else if物理操作符列表 */
bool have_else_;         /*是否有else物理操作符*/
ObPhyOperator else_op_; /*else语句的物理操作符*/
```

主要成员函数

```
int set_expr(ObSqlExpression& expr); /*设置if语句的表达式*/
int set_have_elseif(bool flag);/*设置是否有elseif语句*/
int add_elseif_op(ObPhyOperator &elseif_op);/*加个elseif物理操作符*/
int set_have_else(bool flag);/*设置是否else语句*/
int set_else_op(ObPhyOperator &else_op);/*设置else语句的物理操作符*/
bool is_have_elseif();/*获取是否有elseif语句*/
bool is_have_else();/*获取是否有else语句*/
```

3.3.6 ObProcedureElseIf类

主要成员变量

```
int32_t child_num_;     /*子操作符的数量*/
ObSqlExpression expr_; /*elseif语句的表达式*/
```

主要成员函数

```
int set_expr(ObSqlExpression& expr); /*设置elseif语句的表达式*/
```

3.4 存储过程WHILE语句的实现

本节主要介绍创建存储过程中的while循环语句处理过程，包括词法语法编译，逻辑计划生成和逻辑计划树的结构，物理计划生成和物理计划树的结构。

3.4.1 WHILE语句语法编译

```
WHILE expr DO
    statement_list
END WHILE;
```

3.4.2 WHILE语法树结构

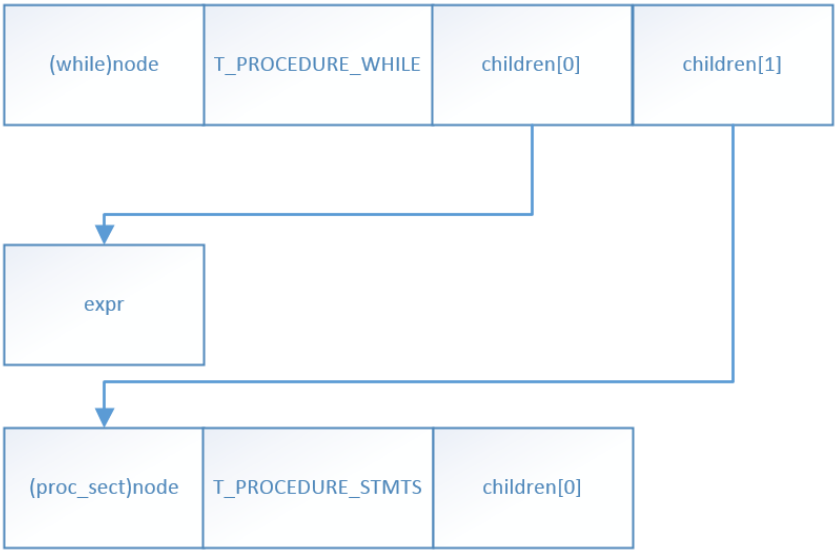

图3.9 WHILE语法树结构

如图3.9所示，对于存储过程中的WHILE语句，我们生成T_PROCEDURE_WHILE类型的node作为父节点，父节点有2个子节点child[0]是WHILE语句的表达式节点，里面存储的是一个表达式，child[1] 是statement_list的，statement_list是由存储过程的if，while，case，select into等过程语句及普通sql语句构成。

3.4.3 WHILE逻辑计划

如图3.10所示，WHILE语句的逻辑计划树LogicalPlan里面的ObProcedureWhileStmt类记录了一个while语句的表达式id，while的循环体语句等信息。

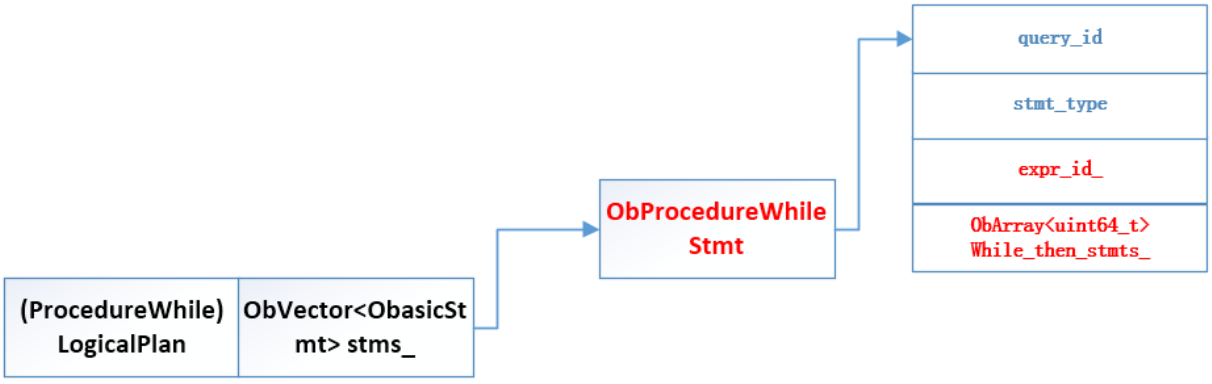

图3.10 WHILE逻辑计划树

3.4.4 WHILE物理计划

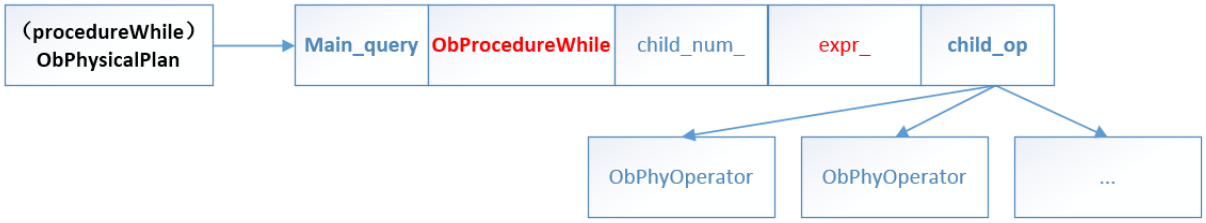

图3.11 WHILE物理计划

如图3.11所示，逻辑计划生成WHILE的物理计划，其中ObProcedureWhile物理操作符中expr_是一个ObSqlExpression类型的表达式。

3.4.5 ObProcedureWhile类

主要成员变量

```
int32_t child_num_;     /*子操作符的数量*/
ObSqlExpression expr_; /*while语句的表达式*/
```

主要成员函数

```
int set_expr(ObSqlExpression& expr); /*设置while语句的表达式*/
```

3.5 存储过程CASE-WHEN语句的实现

本节主要介绍创建存储中的case-when选择开关语句处理过程，包括词法语法编译，逻辑计划生成和逻辑计划树的结构，物理计划生成和物理计划树的结构。

3.5.1 CASE-WHEN语法编译

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list] ...
    [ELSE statement_list]
END CASE
```

3.5.2 CASE-WHEN语法树结构

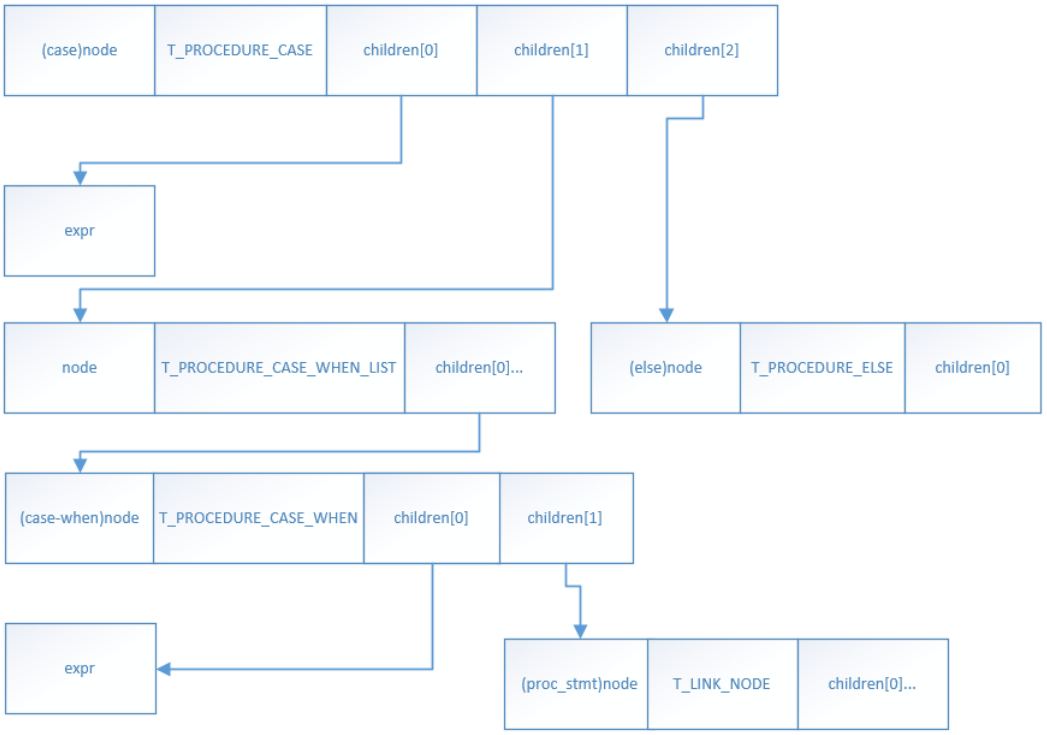

图3.12 CASE-WHEN语法树结构

如图3.12所示，对于存储过程中的case-when语句，我们生成T_PROCEDURE_CASE类型的node做为父节点，父节点有3个子节点child[0]是case-when语句的表达式节点，里面存储的是一个表达式，child[1] 是由T_PROCEDURE_CASE_WHEN_LIST构成的，它是许多T_PROCEDURE_CASE_WHEN节点构成的，每个T_PROCEDURE_CASE_WHEN包含两个节点一个是expr表达式和一个过程语句组成的list，child[2]是一个普通的T_PROCEDURE_ELSE语句节点。

3.5.3 CASE-WHEN逻辑计划

如图3.13所示，WHILE语句的逻辑计划树LogicalPlan里面的ObProcedureCaseStmt类记录了一个CASE语句的表达式id，和对应的casewhen语句列表。

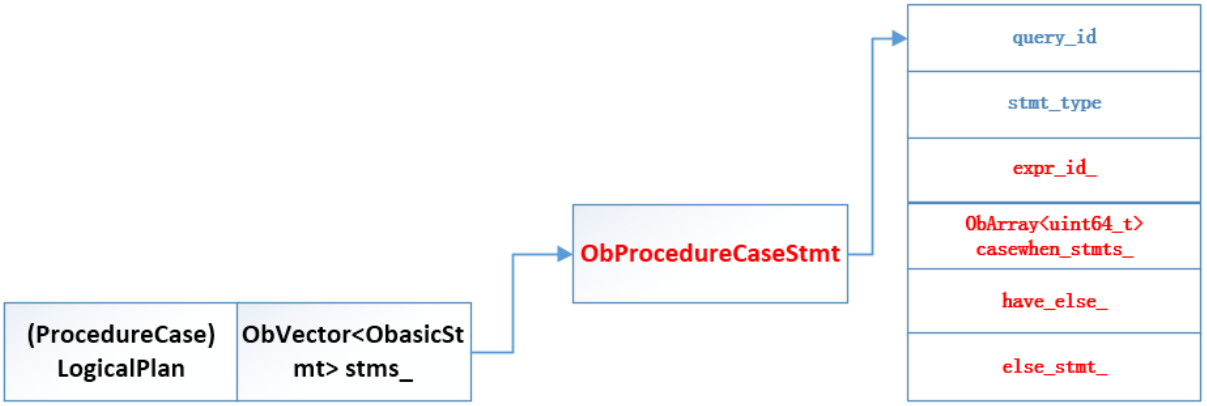

图3.13 CASE-WHEN语法树结构

3.5.4 CASE-WHEN物理计划

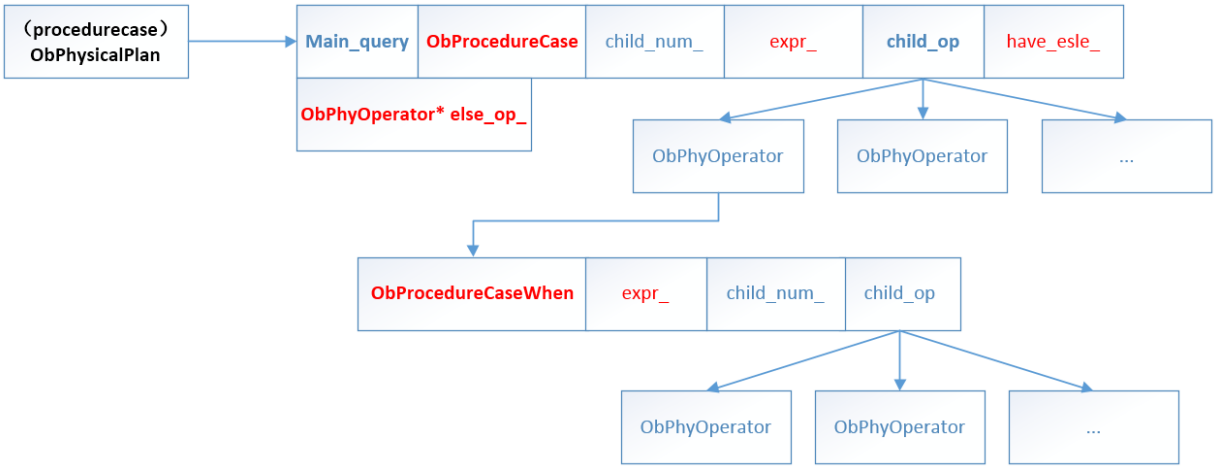

图3.14 CASE-WHEN物理计划树结构

如图3.14所示，CASE计划生成CASE的物理计划，其中ObProcedureCase物理操作符中expr_是一个ObSqlExpression类型的表达式，have_else_标识是否拥有else分支，else_op_是存储CASE语句的else分支的物理操作符，else_op_是else语句分支的物理操作符。CASE的物理操作符是ObProcedureCaseWhen的物理操作符构成的，ObProcedureCaseWhen操作符包含一个expr_和statement_list的操作符。

3.5.5 ObProcedureCase类

主要成员变量

```
int32_t child_num_;     /*子操作符的数量*/
ObSqlExpression expr_; /*case语句的表达式*/
bool have_else_;         /*标识是否有else物理操作符*/
ObPhyOperator *else_op_;/*else语句物理操作符*/
```

主要成员函数

```
int set_expr(ObSqlExpression& expr); /*设置while语句的表达式*/
int have_else(bool flag); /*设置是否有else语句*/
int set_else_op(ObPhyOperator &else_op); /*设置else物理操作符*/
bool is_have_else();/*获取是否有else语句*/
```

3.5.6 ObProcedureCaseWhen类

主要成员变量

```
int32_t child_num_;     /*子操作符的数量*/
ObSqlExpression expr_; /*case when语句的表达式*/
ObSqlExpression case_expr_; /*case语句要比较的那个表达式*/
```

主要成员函数

```
int set_expr(ObSqlExpression& expr); /*设置casewhen语句的表达式*/
int set_case_expr(ObSqlExpression& expr); /*设置case语句的表达式*/
```

3.6 存储过程SELECT INTO语句的实现

本节主要介绍创建存储中的SELECT INTO语句处理过程，包括词法语法编译，逻辑计划生成和逻辑计划树的结构，物理计划生成和物理计划树的结构。

3.6.1 SELECT INTO语法编译

```
SELECT col_name[,...] INTO var_name[,...] table_expr
```

3.6.2 SELECT INTO语句的语法树结构

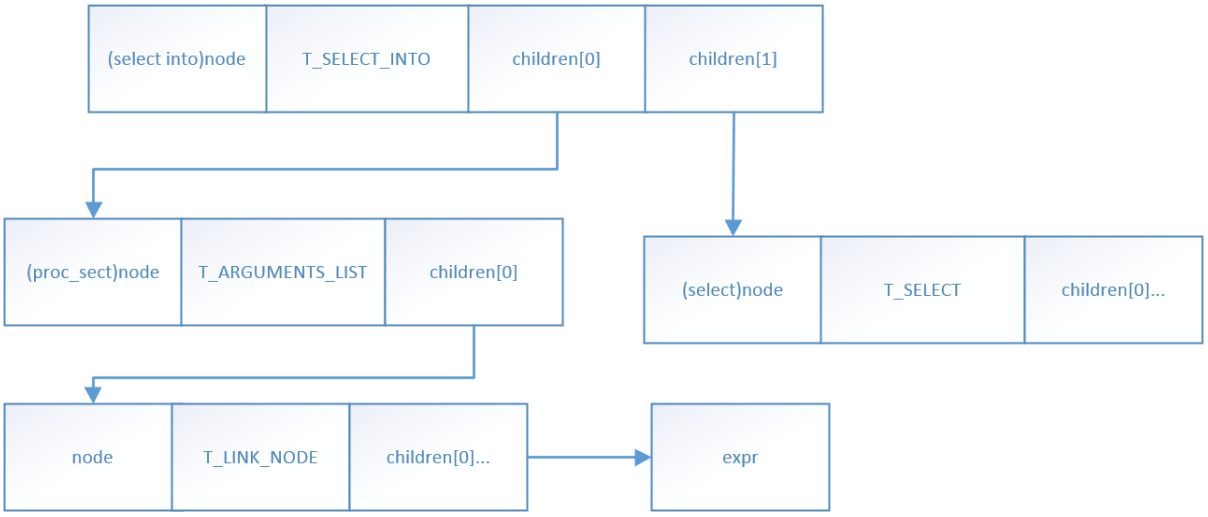

图3.15 SELECT INTO语句的语法树结构

如图3.15所示，对于存储过程中的select into语句，我们生成T_SELECT_INTO类型的node做为父节点，父节点有2个子节点child[0]是select into语句的目标参数列表节点，里面存储的是一个表达式，child[1] 是由T_SELECT构成的，它是一个普通的select语句节点。

3.6.3 SELECT INTO逻辑计划

如图3.16所示，SELECT INTO语句的逻辑计划树LogicalPlan里面的ObProcedureSelectInto类记录了一个select语句的id，目标参数变量列表等信息。

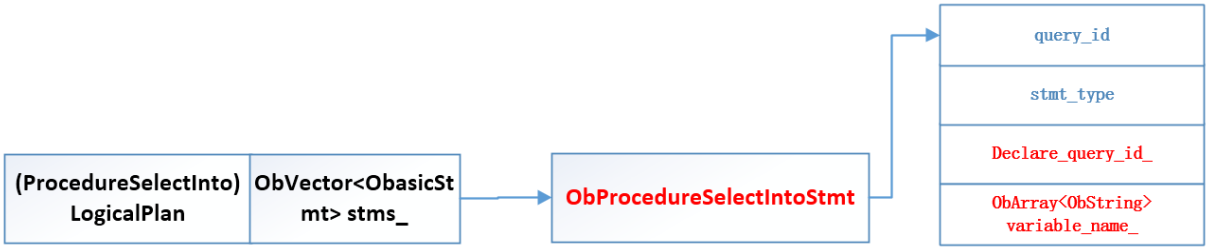

图3.16 SELECT INTO逻辑计划树

3.6.4 SELECT INTO物理计划

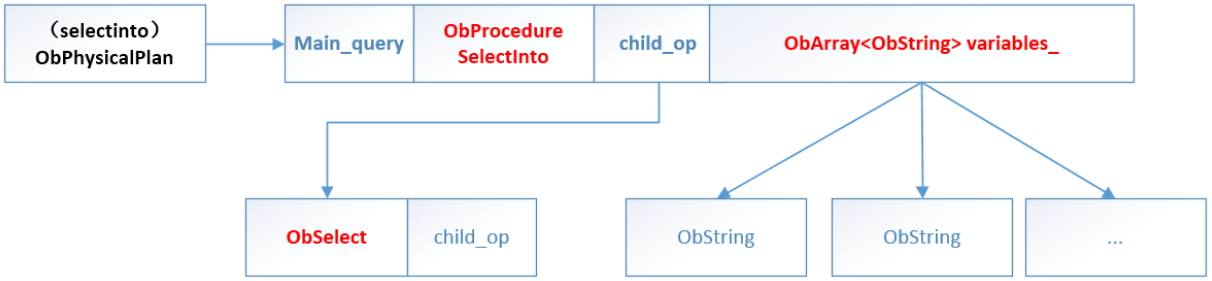

图3.17 SELECT INTO物理计划树

如图3.17所示，逻辑计划生成SELECT INTO的物理计划，其中ObSelect物理操作符是一个普通的select语句的物理操作符，variables_是一个存储目标变量的数组。

3.6.5 ObProcedureSelectInto类

主要成员变量

```
ObArray<ObString> variables_; /*select into语句的目标变量名*/
```

主要成员函数

```
int add_variable(ObString& var); /*添加一个变量名*/
```

3.7 存储过程DECLARE语句的实现

本节主要介绍创建存储中的DECLARE语句处理过程，包括词法语法编译，逻辑计划生成和逻辑计划树的结构，物理计划生成和物理计划树的结构。

3.7.1 DECLARE语法编译

```
DECLARE var_name[,...] type [DEFAULT value]
```

这个语句被用来声明局部变量。要给变量提供一个默认值，请包含一个DEFAULT子句。值可以被指定为一个表达式，不需要为一个常数。如果没有DEFAULT子句，初始值为NULL。

> Tips：具体的例子可以看《存储过程使用手册》

3.7.2 DECLARE 语法树结构图

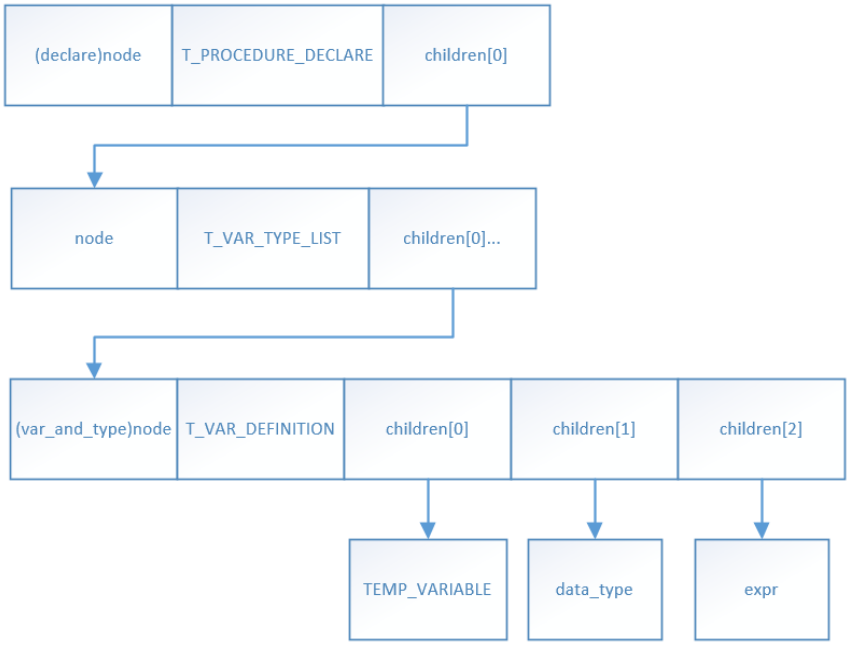

图3.18 DECLARE 语法树结构图

如图3.18所示，对于存储过程中的DECLARE语句，我们生成T_PROCEDURE_DECLARE类型的node做为父节点，父节点有1个子节点child[0]是由多个T_VAR_DEFINITION构成的列表。每个T_VAR_DEFINITION由三个child构成，分别是变量名，类型，和默认值。

3.7.3 DECLARE 逻辑计划

如图3.19所示，DECLARE语句的逻辑计划树LogicalPlan里面的ObProcedureDeclare类记录了一个包含变量定义的列表。

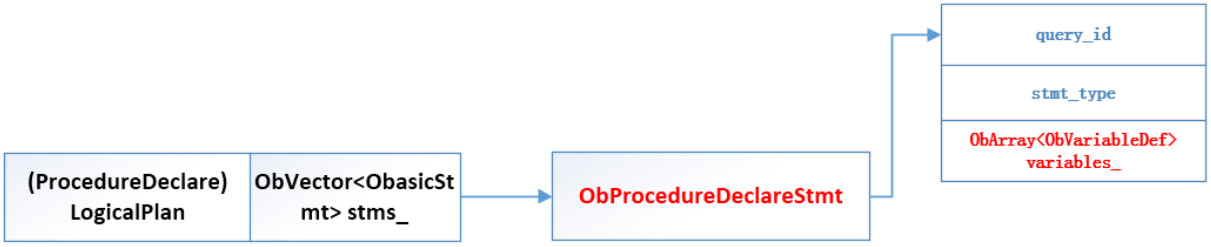

图3.19 DECLARE逻辑计划树

3.7.4 DECLARE物理计划

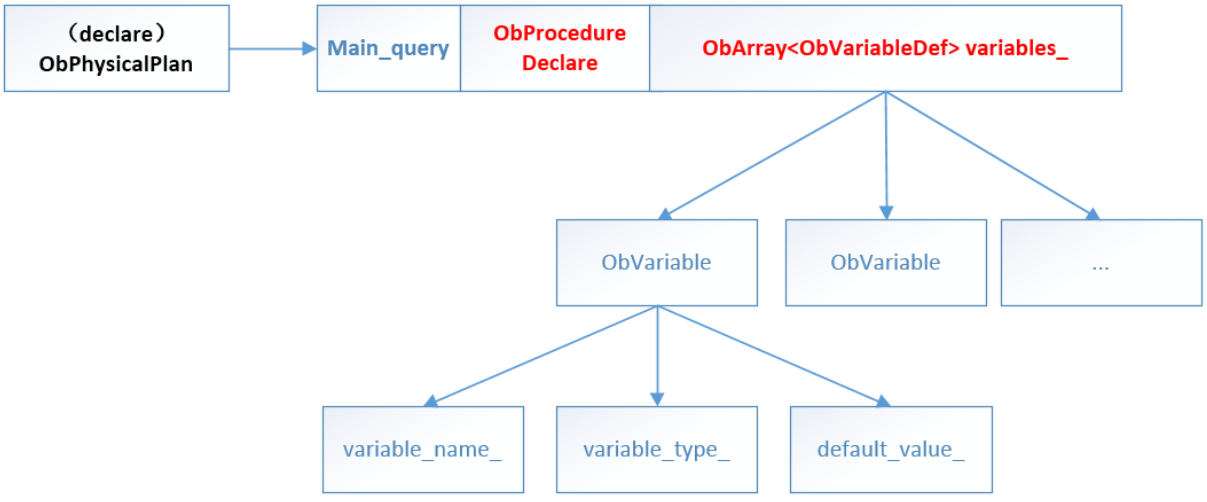

图3.20 DECLARE物理计划树

如图3.20所示，逻辑计划生成DECLARE的物理计划，其中variables_物理操作符是一个由ObVariable构成的数组。每个ObVariable是由变量名称，变量类型和默认值构成的。

3.8 存储过程ASSGIN语句的实现

本节主要介绍创建存储中的Assgin赋值语句处理过程，包括词法语法编译，逻辑计划生成和逻辑计划树的结构，物理计划生成和物理计划树的结构。

3.8.1 ASSGIN语法编译

```
SET var_name = expr [, var_name = expr] ...
```

> Tips：*SET*操作不支持+=运算符

3.8.2 ASSGIN语法树结构图

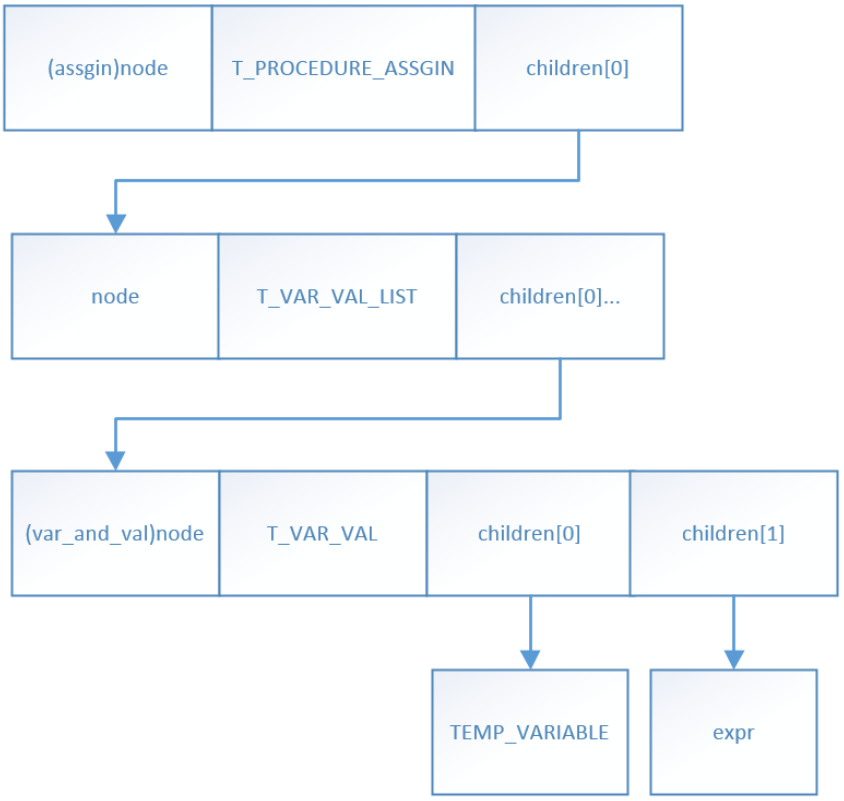

图3.21 ASSGIN语法树

如图3.21所示，对于存储过程中的ASSGIN语句，我们生成T_PROCEDURE_ASSGIN类型的node做为父节点，父节点有1个子节点child[0]是由多个T_VAR_VAL构成的列表。每个T_VAR_VAL由三个child构成，分别是变量名，表达id，和要赋的值。

3.8.3 ASSGIN逻辑计划

如图3.22所示，DECLARE语句的逻辑计划树LogicalPlan里面的ObProcedureAssgin类记录了一个包含变量名称和值的列表。

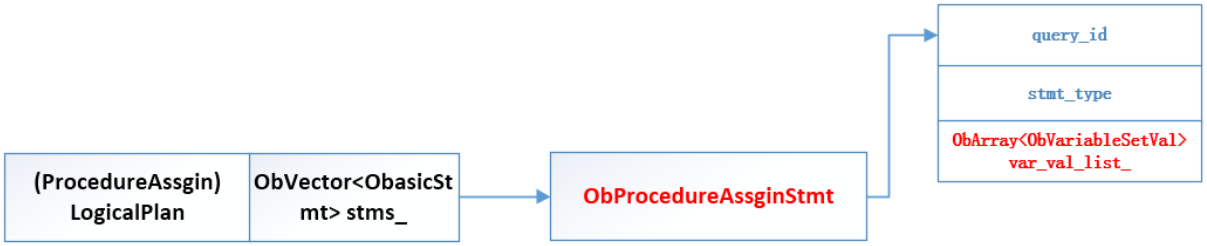

图3.22 ASSGIN逻辑计划树

3.8.4 ASSGIN物理计划

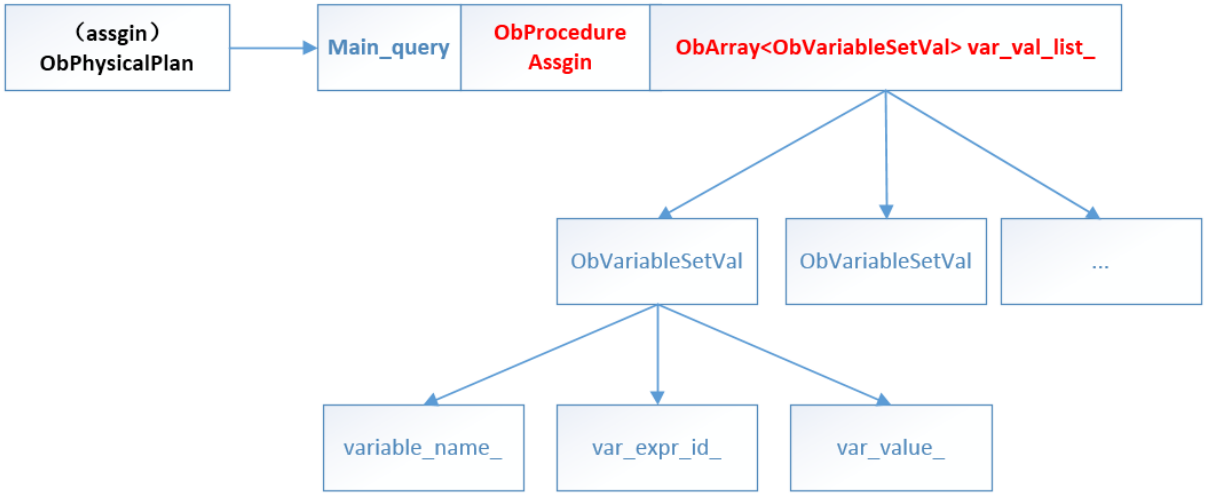

图3.23 ASSGIN物理计划树

如图3.23所示，逻辑计划生成ASSGIN的物理计划，其中var_val_list_物理操作符是一个由ObVariableSetVal构成的数组。每个ObVariableSetVal是由变量名称，变量类型和值构成的。

3.9 CALL PROCEDURE的实现

3.9.1 CALL PROCEDURE语法编译

```
CALL sp_name([parameter[,...]])
```

3.9.2 CALL语法树

如图3.24，对于CALL语句，我们生成T_PROCEDURE_EXEC类型的node作为父节点，有两个子节点，children[0]里记录了存储过程名proc_name，children[1]存储的是参数列表。

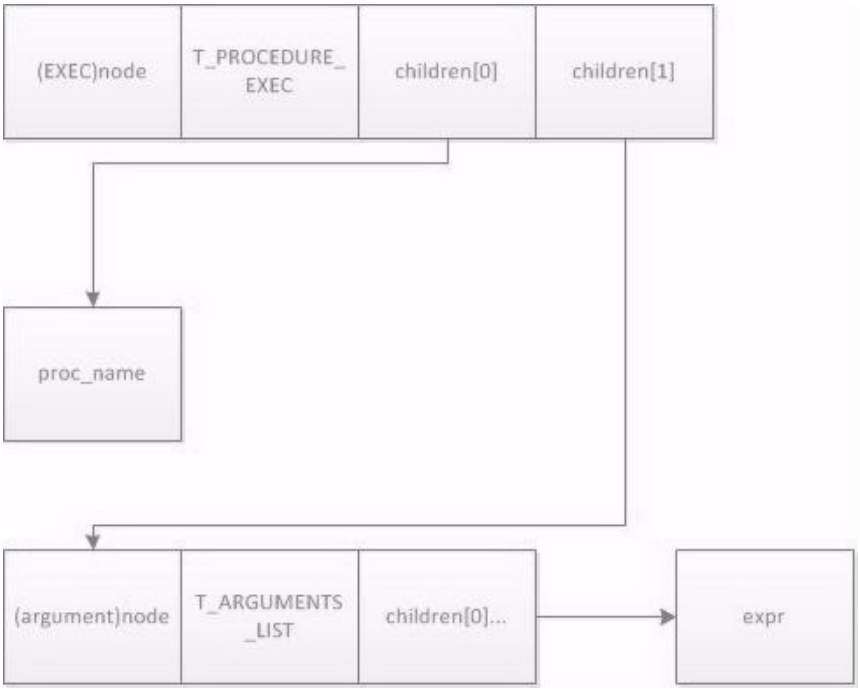

图3.24 CALL语法树

3.9.3 CALL逻辑计划

如图3.25，CALL的逻辑计划树里面的ObProcedureExecuteStmt类里记录了EXEC语句的proc_name,proc_stmt_id,参数名，参数列表等信息。

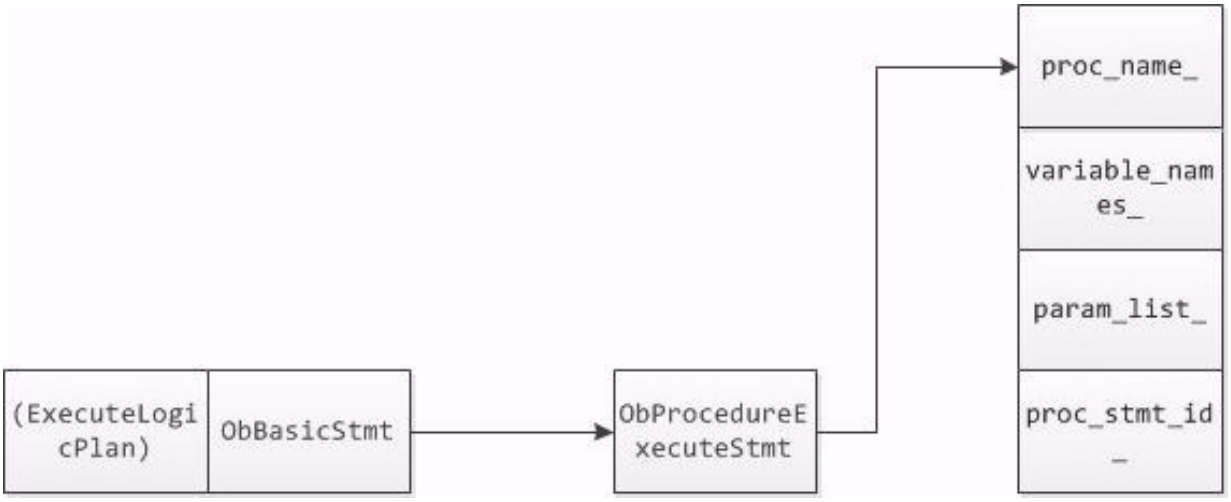

图3.25 CALL逻辑计划树

3.9.4 CALL物理计划

如图3.26，计划生成ObProcedureExecute物理操作符。

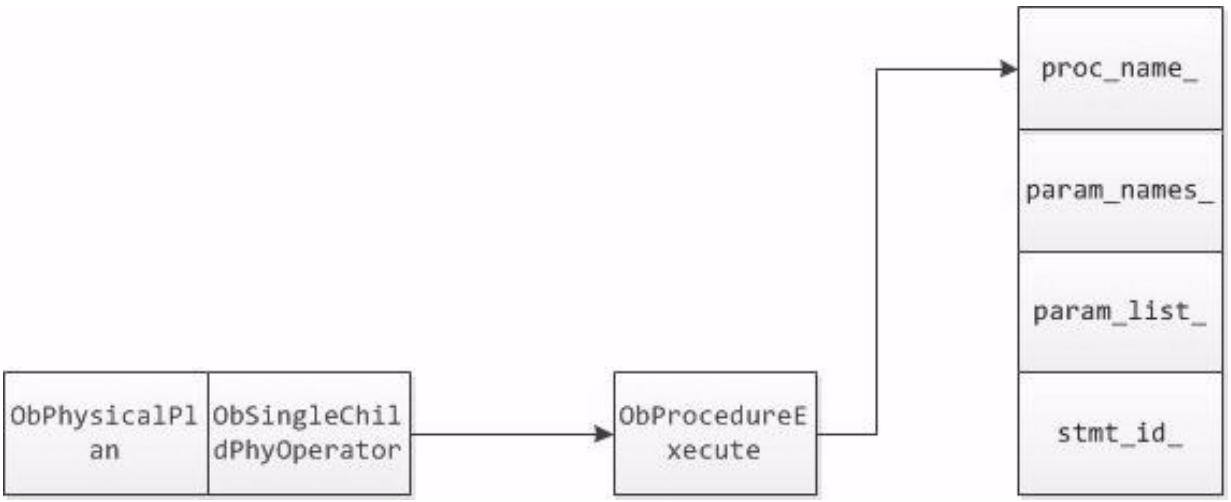

图3.26 CALL物理计划树

3.10 DROP PROCEDURE的实现

3.10.1 DROP PROCEDURE语法编译

```
DROP sp_name
```

3.10.2 DROP语法树

如图3.27，对于DROP语句，我们生成T_PROCEDURE_DROP类型的node作为父节点，有1个子节点，children[0]里记录了存储过程名proc_name。

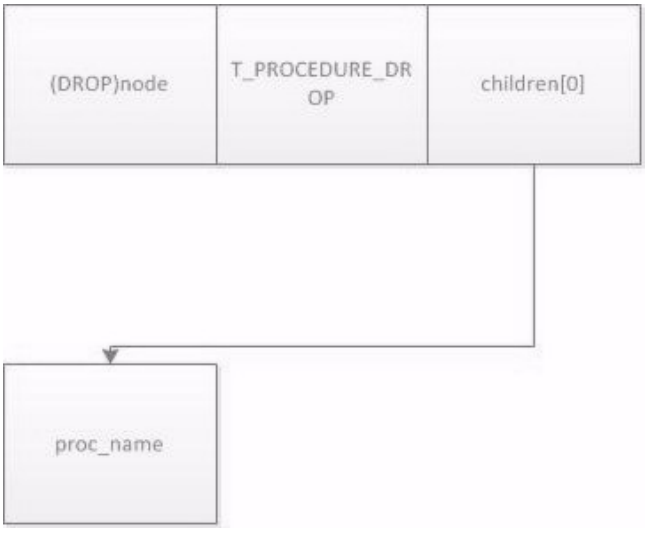

图3.27 DROP语法树

3.10.3 DROP逻辑计划

如图3.28，DROP的逻辑计划树里面的ObProcedureDropStmt类里记录了DROP语句的proc_name,delete_id等信息。

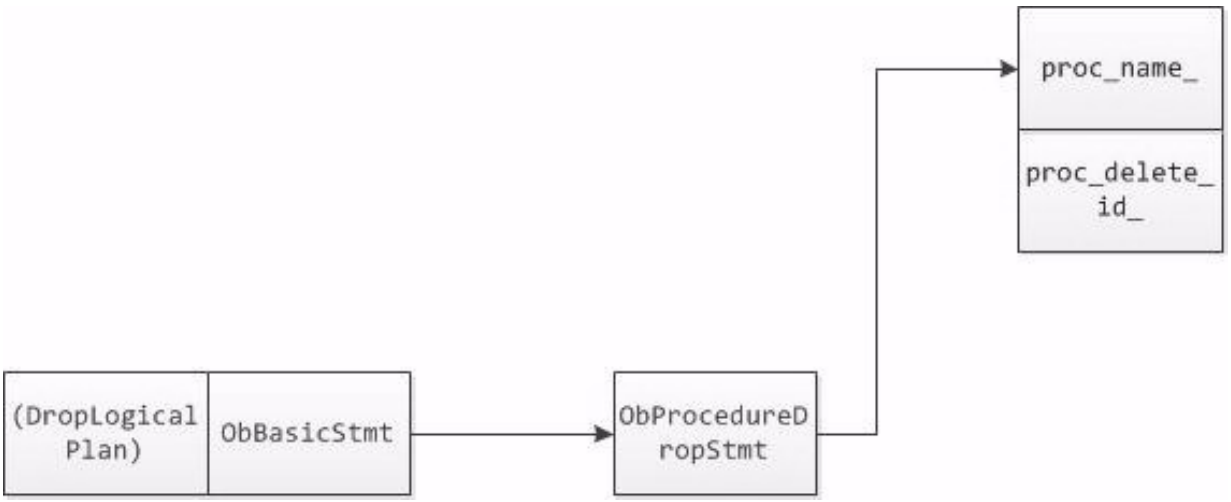

图3.28 DROP逻辑计划树

3.10.4 DROP物理计划

如图3.29，计划生成ObProcedureDrop物理操作符。

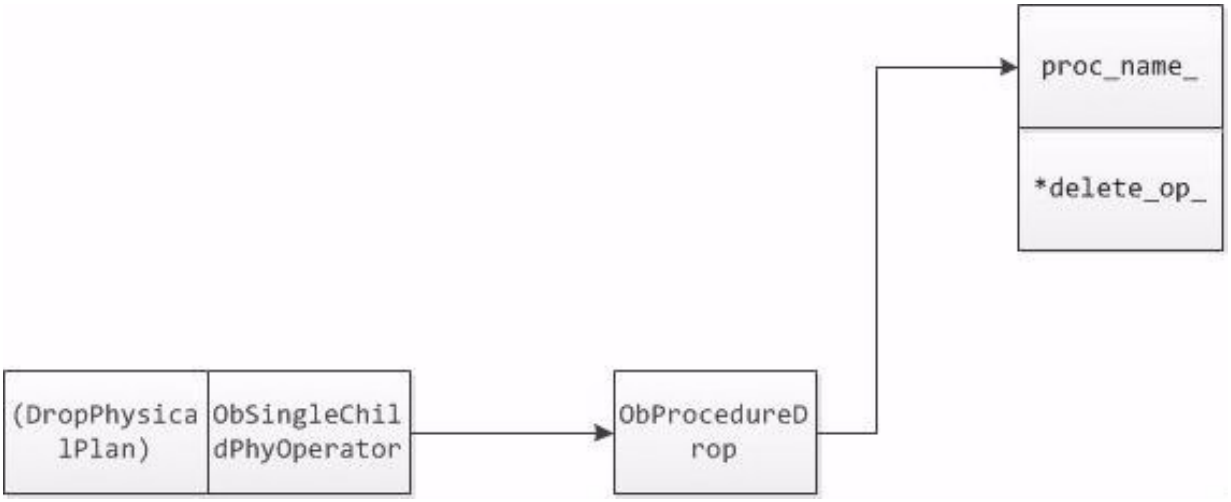

图3.29 DROP物理计划树

3.10.5 ObProcedureDrop类

主要成员变量

```
ObString proc_name;/*存储过程名称*/
unit64_t delete_id;
ObPhysical *delete_op;
```

主要成员函数

```
int add_variable(ObString& var); /*添加一个变量名*/
```

4 系统表__all_procedure的创建

在`bootstrap_sys_tables(...)`函数中增加代码L6，如下所示。

```
if (OB_SUCCESS == ret)
{
    if (OB_SUCCESS != (ret = ObExtraTablesSchema::all_procedure_sch
ema(table_schema)))
    {
        TBSYS_LOG(WARN, "failed to get schema for __all_procedure,
err=%d", ret);
    }
    else if (OB_SUCCESS != (ret = create_sys_table(table_schema)))
    {
        TBSYS_LOG(WARN, "failed to create empty tablet for __all_pr
ocedure, err=%d", ret);
    }
}
```

在ObExtraTablesSchema类中的*all_procedure_schema*函数是构建__all_procedure表的schema结构。

```
ADD_COLUMN_SCHEMA("proc_name", //column_name
        column_id ++, //column_id
        1, //rowkey_id
        ObVarcharType,  //column_type
        128, //column length
        false); //is nullable
ADD_COLUMN_SCHEMA("source", //column_name
        column_id ++, //column_id
        0, //rowkey_id
        ObVarcharType,  //column_type
        10240, //column length
        false); //is nullable
```

其他功能的使用方法及规则

1. flex和bison是进行语法解析的一组工具。 ↩