Oceanbase Cursor设计文档

修订历史

版本	修订日期	修订描述	作者	备注
0.1	2014-9-1	无	周楠	初稿
0.2	2016-1-3	无	祝君	定稿

1 概述

1.1目的

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OB0.4.2.8版本中缺乏游标功能的支持，为满足业务功能上的需求，需在OceanBase_BankComm版本中增加游标功能，使得OB数据库系统能够实现从结果集中逐一读取数据的功能，用户可以用SQL语句逐一从游标中获取记录，并赋给主变量，交由主语言进一步处理。

本文档的编写目的，是对实现游标功能的技术描述，读者通过对本文档的阅读与学习，能够了解游标功能的设计方案，以及该设计方案在现有OB系统中的技术实现流程。

1.2 文档结构

无

1.3 功能

游标是数据库高级语言调用接口的重要组成部分，它与数据库查询引擎的设计紧密相连。一般在开发数据库查询引擎的同时会完成游标的实现。如果数据库系统初始阶段没有实现游标机制，很多的业务需求便无法满足。此时，通常需要在现有的查询处理器上进行二次开发，添加游标机制。OceanBase是面向海量数据查询的分布式数据库系统，它结合了关系数据库和非关系数据库的优势，支持关系查询和跨行跨表事务。但目前OceanBase不支持游标功能，影响其在企业和机构的推广使用。本文在深度分析游标原理和OceanBase查询处理策略的基础上，提出了在OceanBase上实现支持磁盘缓存功能的游标方案，并对实现的游标进行高并发、回归测试验证其可用性。实验验证该游标机制能极大地增加用户访问数据库的灵活性和查询性能。

1.4 性能指标

经过试验，将内存限制大小设置为10M时，性能较好。试验中，结果表为4列，列的类型为int,int,char和char。这种情形下，open大小为10万行的结果集（约20M）大致需要1秒，fetch任意一行不超过0.1秒，如下图：

```
mysql> declare c cursor for select * from test where no<100003;
Query OK, 0 rows affected (0.00 sec)

mysql> open c;
Query OK, 100000 rows affected (1.03 sec)

mysql> fetch c ;
+------+-----------------------------------------------------------------------------------------------------------------+---------+---------
--+
| no   | name                                                                                                            | company | salary
 |
+------+-----------------------------------------------------------------------------------------------------------------+---------+---------
--+
|    1 | 18z478hoo3gw7n0v1i90116z1wuno1b5pb1sjuiwmtk5jd5tc9vqf437525zfoh4l7pt4kj9ap3576qlfxmkbo2fyywbdsq3zizwwzwg0 | cd      | 36685806
 |
+------+-----------------------------------------------------------------------------------------------------------------+---------+---------
--+
1 row in set (0.05 sec)

mysql> close c;
Query OK, 0 rows affected (0.00 sec)

mysql>
```

open大小为100万行的结果集（约200M）大致需要11秒，fetch任意一行不超过0.1秒，如下图：

```
mysql> declare c cursor for select * from test where no<1000004;
Query OK, 0 rows affected (0.00 sec)

mysql> open c;
Query OK, 1000000 rows affected (10.99 sec)

mysql> fetch c;
+------+-----------------------------------------------------------------------------------------------------------------+---------+----------+
| no   | name                                                                                                            | company | salary   |
+------+-----------------------------------------------------------------------------------------------------------------+---------+----------+
|    1 | 18z478hoo3gw7n0v1i90116z1wuno1b5pb1sjuiwmtk5jd5tc9vqf437525zfoh4l7pt4kj9ap3576qlfxmkbo2fyywbdsq3zizwwzwg0 | cd      | 36685806 |
+------+-----------------------------------------------------------------------------------------------------------------+---------+----------+
1 row in set (0.08 sec)

mysql> close c;
Query OK, 0 rows affected (0.01 sec)
```

我对相同的表执行select*语句进行对比：10万行select*花费约1秒，100万行select*花费约11秒。可见，游标的性能还是不错的。

2 功能设计

2.1 外部用户接口

本小节主要介绍，添加Cursor相关新功能后，对于用户使用OB的主要影响，当然影响主要集中于SQL语句的使用。

1) Cursor主要功能

1、查询：能够为查询语句建立游标。查询语句语句以SQL方式输入，要求该语句能够被OB识别接受，并成功为select语句建立游标。

示例SQL：

DECLARE cursor_name CURSOR FOR select_statement

2、开启游标：能够使用已经声明过的游标名称，成功生成结果集

示例SQL：

Open cursor_name;

3、获取结果：在使用游标名获取结果集的时候，成功返回一行结果。
示例SQL：
Fetch cursor_name;

4、关闭游标：在使用游标名关闭由标的时候，能够删除游标和相应的结果集
示例SQL：
Close cursor_name;

2.2 词法语法解析子模块设计

在游标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。

2.2.1 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：
(1)、识别出源程序的各个语法单位；
(2)、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；
(3)、过滤掉源程序的注释；
(4)、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：

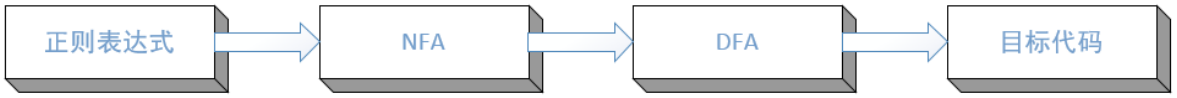

图2.3 正则表达式与有穷自动机

2.2.2 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。

2.2.3 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。

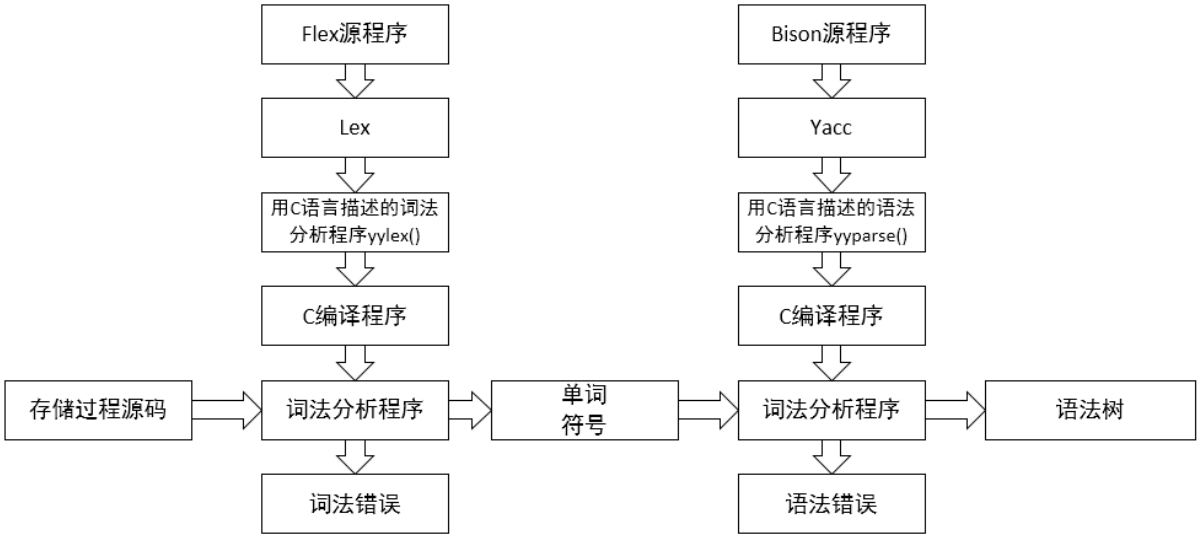

图2.4 flex和bison进行语法词法解析

2.2.4 OceanBase中的词法语法解析

Cursor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。

2.3 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。

2.3.1 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。

2.3.2 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_cursor_declare_stmt(...) {...}

int resolve_cursor_open_stmt(...) {...}

int resolve_cursor_fetch_stmt(...) {...}

int resolve_cursor_fetch_prior_stmt(...) {...}

int resolve_cursor_fetch_first_stmt(...) {...}

int resolve_cursor_fetch_first_into_stmt(...) {...}

int resolve_cursor_fetch_last_stmt(...) {...}

int resolve_cursor_fetch_last_into_stmt(...) {...}

int resolve_cursor_fetch_relative_stmt(...) {...}

int resolve_cursor_fetch_absolute_stmt(...) {...}

int resolve_cursor_fetch_fromto_stmt(...) {...}

int resolve_cursor_close_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_CURSOR_DECLARE:
      {
         ret = resolve_cursor_declare_stmt(result_plan, node, quer
y_id);
         break;
      }
      case T_CURSOR_OPEN:
      {
           ret = resolve_cursor_open_stmt(result_plan, node, quer
y_id);
           break;
      }
      case T_CURSOR_FETCH:
      {
         ret = resolve_cursor_fetch_stmt(result_plan, node, query_i
d);

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

2.3.3 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。

2.4 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。

游标中有declare、open、fetch、close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。

2.4.1 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。

2.4.2 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_cursor_declare(...) {...}

int gen_physical_cursor_open(...) {...}

int gen_physical_cursor_fetch(...) {...}

int gen_physical_cursor_fetch_prior(...) {...}

int gen_physical_cursor_fetch_first(...) {...}

int gen_physical_cursor_fetch_last(...) {...}

int gen_physical_cursor_fetch_relative(...) {...}

int gen_physical_cursor_fetch_absolute(...) {...}

int gen_physical_cursor_fetch_fromto(...) {...}

int gen_physical_cursor_close(...) {...}
```

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_CURSOR_DECLARE:
        ret=gen_physical_cursor_declare(logical_plan, physical_plan
..., index);
        break;
    case ObBasicStmt::T_CURSOR_OPEN:
        ret=gen_physical_cursor_open(logical_plan, physical_plan
..., index);
        break;
    case ObBasicStmt::T_CURSOR_FETCH:
        ret=gen_physical_cursor_fetch(logical_plan, physical_plan
..., index);
        break;
    case ObBasicStmt::T_CURSOR_CLOSE:
        ret=gen_physical_cursor_close(logical_plan, physical_plan
..., index);
            break;
    }
}
```

2.4.3 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SQL语句的基本处理框架

游标中 DECLARE、OPEN、FETCH、CLOSE 语句都是单独作为一条SQL 语句处理的。DECLARE语句表示声明游标和对应的select语句，我们会把所声明的游标名字和select语句的物理执行计划用idplanhash存入session中。OPEN语句根据游标名字hash将物理执行计划存储SESSION，并在send_resonpse时执行物理计划，得到结果集，再用obcursor操作符将得到的结果集缓存起来，如果结果集过大，obcursor操作符会自动flush部分结果集到磁盘中。FETCH语句根据游标名字找到结果集，并返回某一行结果。CLOSE语句根据游标名字找到存储的物理计划和结果集，删除之。这四条语句的处理流程大致相同，下面我以DECLARE语句为例说明OB游标语句的处理框架。

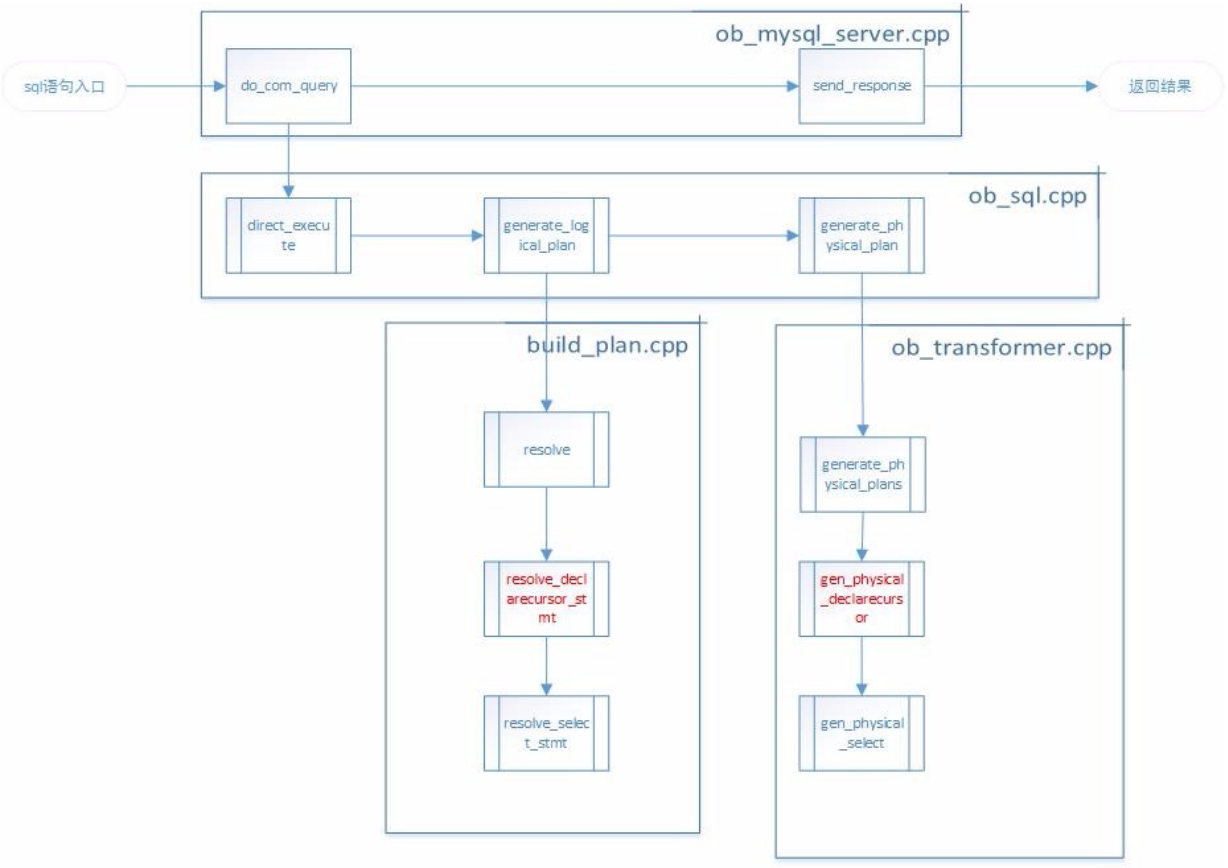

图3.1 Declare Cursor语句处理框架

如图3.1，以declare语句为例，介绍了其整个处理流程，以及所调用的函数。函数名为红色的是需要新增的功能函数。Sql语句传输到服务器之后，由do_com_query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_declarecursor_stmt、gen_physical_declarecursor函数来生成declare语句的逻辑计划和物理计划。然后，do_com_query调用send_response函数执行物理计划并返回结果。

3.2 OceanBase中cursor语句的基本处理框架

Cursor使用顺序为declare cursor，open cursor，fetch cursor，close cursor，不可乱序使用。其中：declare cursor语句经过语法解析、逻辑计划和物理计划生成之后，它的物理计划包含三个操作符，作为main_query的是ObDeclareCursor，用来控制物理操作和记录游标的状态，其他两个是ObCursor和Obselect，分别用来缓存、管理结果集和产生select语句的结果集。执行物理计划open()时，会把自己的物理计划以hashmap的方式存储起来，即建立由cursor_name到物理计划的哈希，以便open的时候能得到相应的物理计划。

open cursor语句经过语法解析、逻辑计划和物理计划生成之后，执行物理计划open()时，用cursor_name哈希取到对应的物理计划，取出ObCursor操作符，并且执行之。于是，得到了所需结果集并以缓存好。如果结果集过大，则会把结果集flush到磁盘。

fetch cursor语句经过语法解析、逻辑计划和物理计划生成之后，执行物理计划open()时，用cursor_name哈希取到对应的物理计划，取出ObCursor操作符，然后get_next_row()时从结果集中取出一行并返回。

close cursor语句经过语法解析、逻辑计划和物理计划生成之后，执行物理计划open()时，用cursor_name哈希取到对应的物理计划，取出ObDeclareCursor操作符，执行close操作删除缓存的结果集，再删除存在session里的物理计划。如图3.2所示：

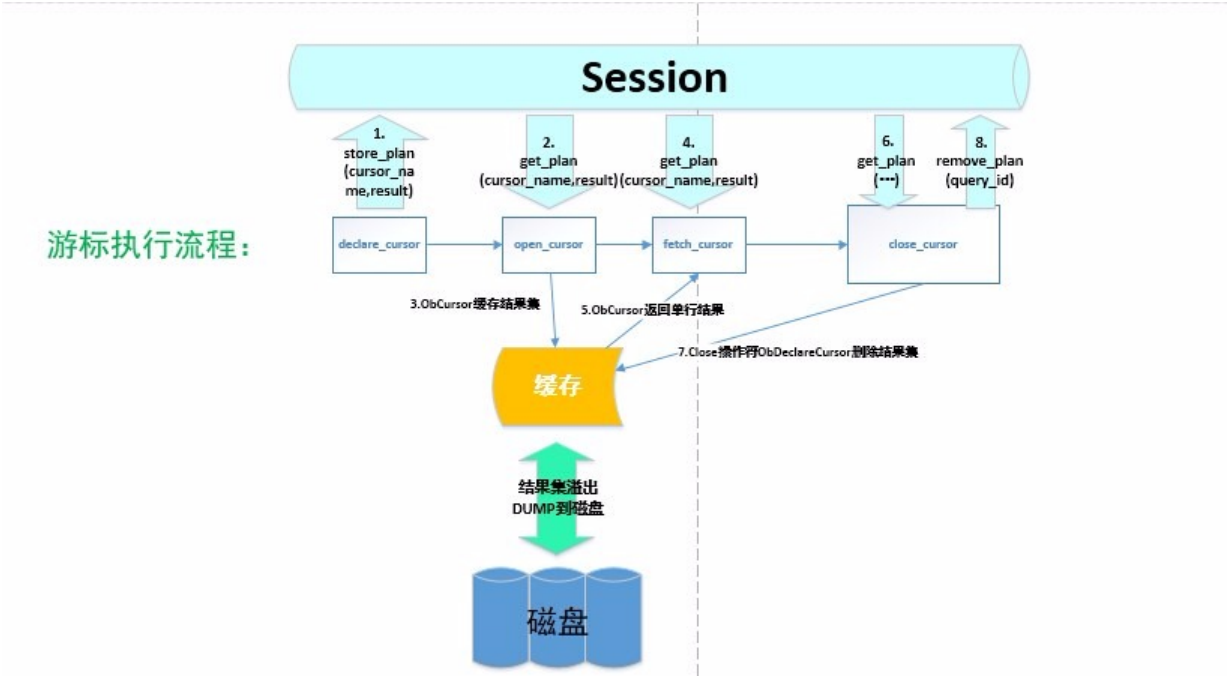

图3.2 OceanBase游标处理框架

3.3 游标声明的实现

本节主要介绍游标declare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。

3.3.1 declare语法

```
Declare cursor_nameCursor For select_stmt
```

3.3.2 declare语法树结构

如图3.3，OB在处理Declare语句的时候，会对sql语句进行词法分析。对于Declare Cursor语句就会将其解析成T_DeclareCursor类型的node，它有两个子节点，children[0]用来存cursor_name，children[1]用来存select语句的语法树。

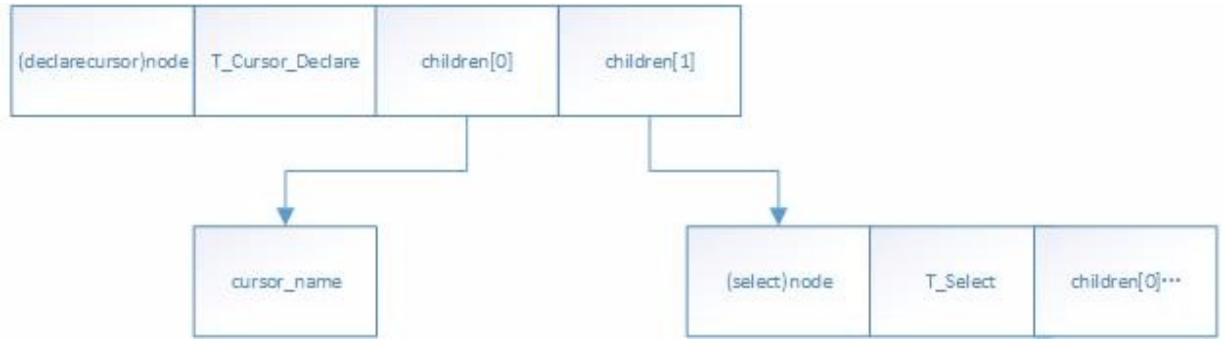

图3.3 Declare语法树

3.3.3 declare逻辑计划

OB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。如图3.4，对于declare cursor语句，生成逻辑计划的时候会将语法树中的T_DeclareCursor节点、T_Select节点分别处理生成ObDeclareCursorStmt和SelectStmt。其中ObDeclareCursorStmt为main_stmt，记录了cursor_name,query_id和select语句的declare_query_id等信息。SelectStmt记录了select语句逻辑计划的各种信息。

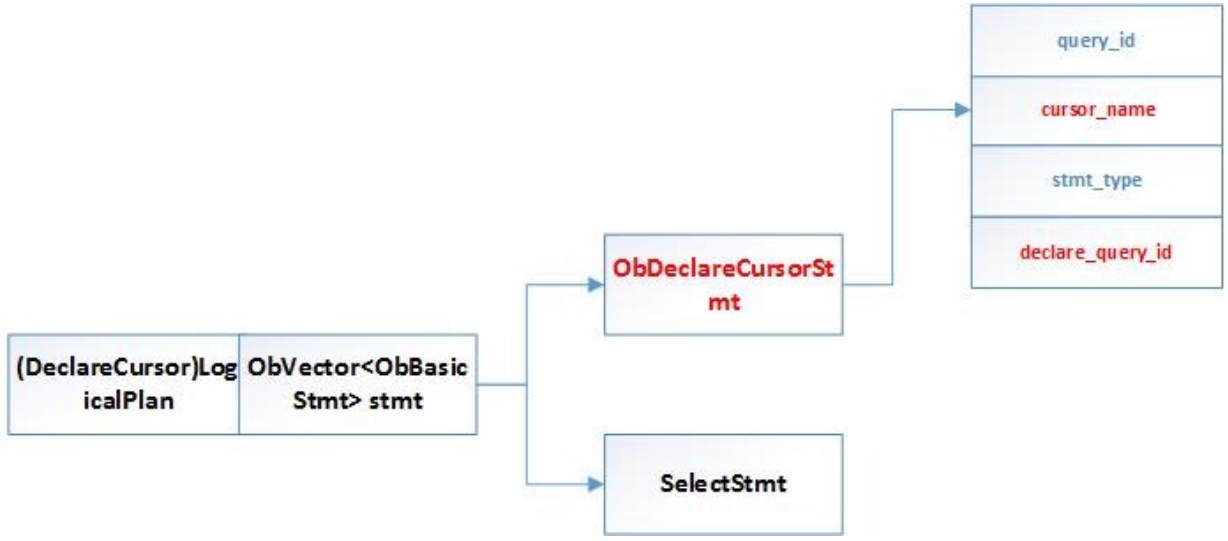

图3.4 Declare逻辑计划

> 注意：图中红色部分是所要创建的类或者需要新建的变量。

3.3.4 declare物理计划

如图3.5，declare物理计划包括三个物理操作符ObDeclareCursor, ObCursor和ObSelect。其中ObDec lareCursor是需要实现和添加，也是main_query，用于设定定义游标所需要的操作。ObCursor物理操作符是ObDeclareCursor的子操作符，用来缓存select语句结果集，当内存不能放下全部数据时，自动将结果刷到外存上。最后，ObCursor操作符的子操作符是Select语句操作符，用来完成select操作，得到结果集。

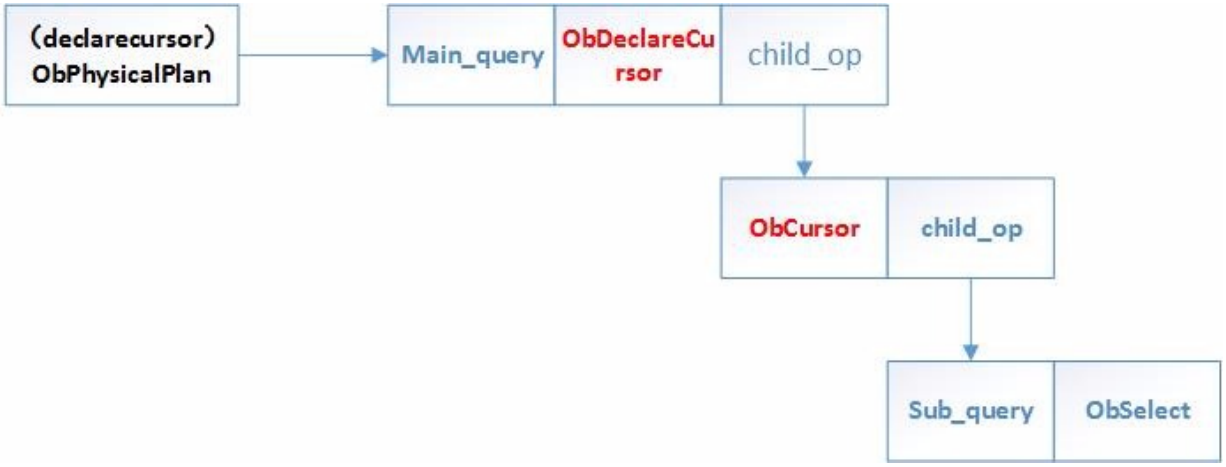

图3.5 Declare物理计划

> 注意：红色的部分是需要实现的类

3.3.5 declare物理计划执行流程

3.3.6 ObCursor类

主要成员变量

```
common::ObArray<int64_t>run_array_;//记录磁盘中每块数据含有的行数
ObInMemoryCursorin_mem_cursor_;//存储和管理内存中的结果集
ObMergeCursormerge_cursor_;//存储和管理磁盘中的结果集
ObCursorHelper *cursor_reader_;// ObInMemoryCursor 和ObMergeCursor的
父类
int64_tmem_size_limit_;//限制占用内存大小
int64_trow_offset_; //内存数据的当前偏移量
int64_trow_count_;//记录每次刷入磁盘的数据行数
int64_trun_idx_;//当前磁盘数据的偏移量
int64_trow_num_;//结果集总行数
```

主要成员函数

```
int do_cursor();//执行open语句时被调用，将结果集缓存起来
int get_run_file();//将磁盘中对应的数据块载入内存
virtual int get_next_row(const common::ObRow *&row);
intget_ab_row(int64_tab_num_,const common::ObRow *&row);
int get_first_row(const common::ObRow *&row);
int get_last_row(const common::ObRow *&row);
int get_prior_row(const common::ObRow *&row);
intget_rela_row(boolis_next,int64_trela_count,constcommon::ObRow
*&row);
```

3.3.7 In_Mem_Cursor类

主要成员变量

```
common::ObRowStore row_store_;//管理存储row的类
common::ObArray<const common::ObRowStore::StoredRow*> cursor_arra
y_;//存储内存结果
int64_t cursor_array_get_pos_;//当前取结果集的位置
common::ObRow curr_row_;//过渡变量
const common::ObRowDesc *row_desc_;//行的描述
```

主要成员函数

```
int add_row(const common::ObRow &row);//向cursor_array_中载入数据
int get_next_row(common::ObRow &row);//从cursor_array_中取出一行
```

3.3.7 Merge_Cursor类

主要成员变量

```
ObRunFile run_file_;//控制文件的磁盘io的类
int64_t run_idx_;//磁盘文件的位置
int64_t dump_run_count_;//数据块的个数
common::ObRow curr_row_;//过渡变量
const common::ObRowDesc *row_desc_;//行的描述
```

主要成员函数

```
int dump_run(ObInMemoryCursor &rows);//执行open语句被调用，将结果集刷入
磁盘
intget_next_row(common::ObRow&row);//从磁盘文件中取一行
intend_get_run();//关闭（还原）数据块
```

3.4 游标Open功能的实现

本节主要介绍游标open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。

3.4.1 Open语句语法

```
Open cursor_name
```

3.4.2 Open语法树结构

如图3.6，对于open语句，我们生成T_OpenCursor类型的node作为父节点，父节点只有一个子节点children[0]，里头记录了cursor_name。

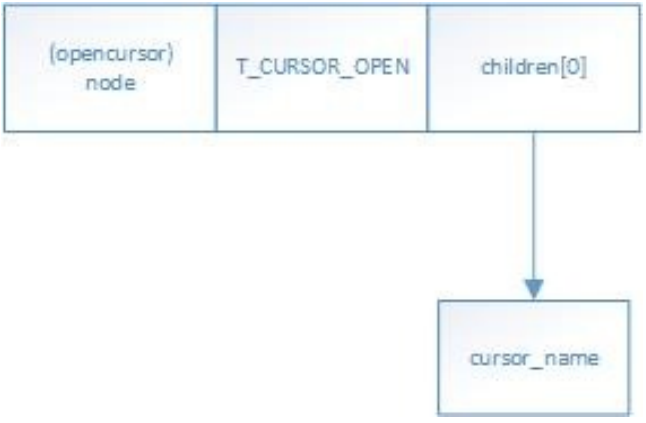

图3.6 Open语法树

3.4.3 Open逻辑计划树

如图3.7，Open的逻辑计划树比较简单，LogicalPlan里面的ObOpenCursorStmt类记录了Open语句的cursor_name、query_id等信息。

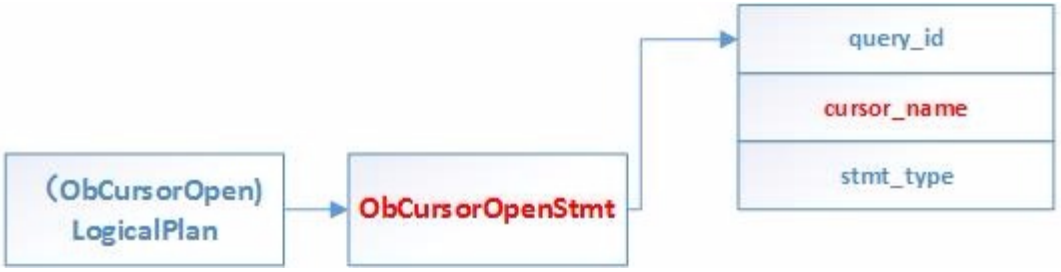

图3.7 Open语法树

3.4.4 Open物理计划

图3.8 Open物理计划树

> 注意：红色的部分是需要实现的类

如图所示：计划生成ObOpenCursor物理操作符，负责打开存在session里的select物理计划，并用ObDeclare物理操作符缓存结果集。
具体执行时，open()函数调用fill_open_iterms()，fill_open_iterms()用cursor_name在session中哈希取出正确的物理计划（即declare语句存入的物理计划），然后执行它，执行完后结果集便存储在内存或磁盘中了。

3.5 游标Fetch功能的实现

本节主要介绍游标fetch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。

3.5.1 Fetch语法

```
Fetch cursor_name
```

3.5.2 Fetch语法树结构

如图3.9，Fetch 语句与open语句类似。

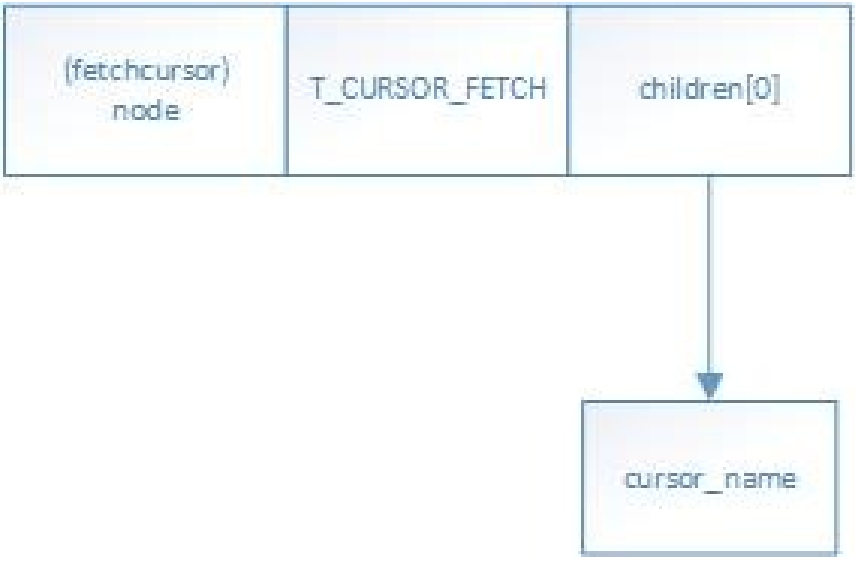

图3.9 Fetch语法树

3.5.3 Fetch逻辑计划

如图3.10，逻辑计划中用ObFetchStmt对象中存入cursor_name、query_id等信息。

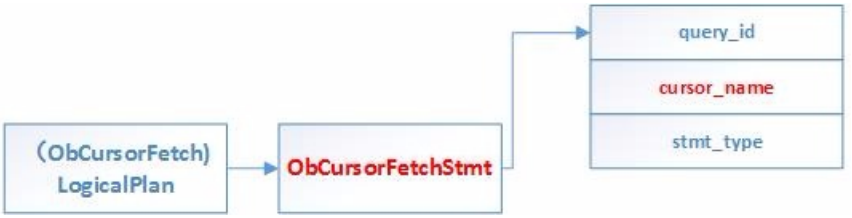

图3.10 Fetch逻辑计划

> 注意：图中红色部分是所要创建的类或者需要新建的变量。

3.5.4 Fetch物理计划

图3.11 Fetch物理计划

如图所示,计划生成ObFetchCursor从结果集中返回单行记录。首先，ObFetchCursor操作符根据stmt_id哈希到declare语句存入session的物理计划，该过程和open语句一样。然后，执行ObFetchCursor操作符的get_next_row()，返回下一行结果集。

3.6 游标Close功能的实现

本节主要介绍游标close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。

3.6.1 Close语法编译

```
Close cursor_name
```

3.6.2 Close语句的语法树结构

如图3.12，Close 语句与open语句类似：

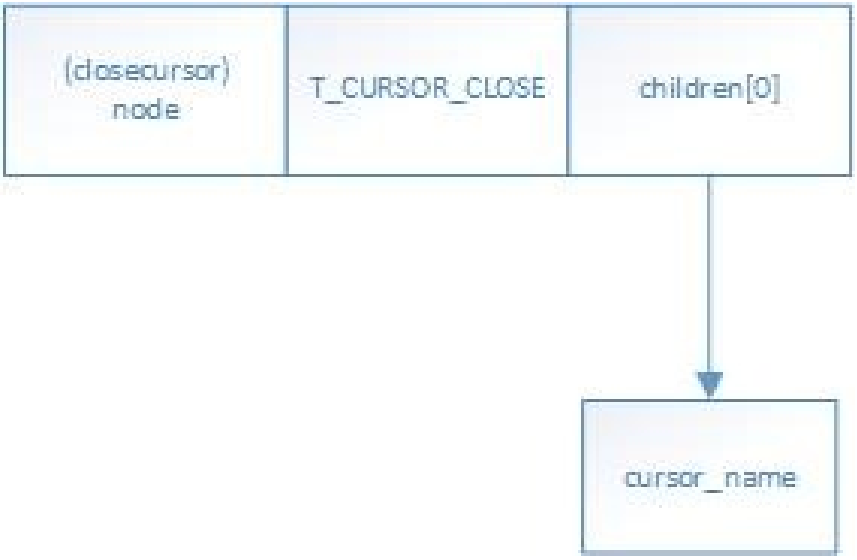

图3.11 Close语法树

3.6.3 Close逻辑计划

如图3.13，close语句逻辑计划与open、fetch语句类似，在ObCloseStmt对象中存进一个cursor_name。

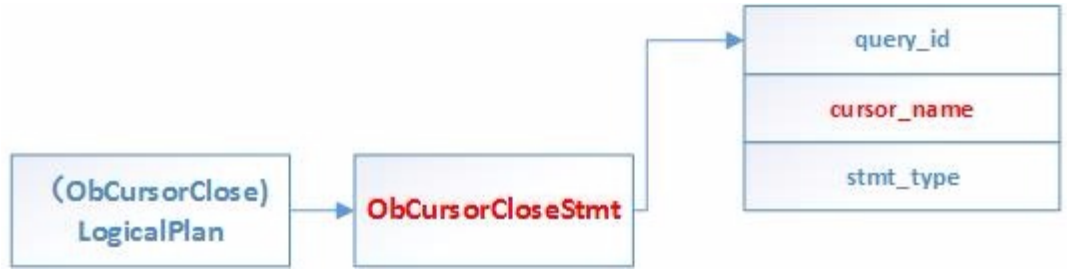

图3.13 Close逻辑计划

> 注意：图中红色部分是所要创建的累或者需要新建的变量。

3.6.4 Close物理计划

图3.14 Close逻辑计划

如图所示，计划生成ObCloseCursor物理操作符，和之前的ObOpenCursor操作符类似，首先负责找到存在session里的物理计划，然后根据删除掉物理计划所打开的结果集，最后删除物理计划本身。

示例

声明一个名为cur1的游标

```
declare cursor cur1 from select * from test;
```

打开游标

```
open cur1;
```

获取数据

```
fetch cur1;
```

关闭游标

```
close cur1;
```

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。 ↩