PostgreSQL 解析資料

～ node 構造 ～

(株) NTT データ
オープンソース開発センタ
井久保 寛明

1. はじめに

本ドキュメントでは、PostgreSQL のクエリコンパイル全般に使われている node 構造について説明している。node 構造は、クエリのコンパイルで木構造が必要なところで使用されている。例えば、SQL 文をパージングしたあとのパーズ木やクエリの実行プランなどに使われている。
node 構造は、クエリコンパイルの他にもメモリコンテキストの木構造の保持にも利用されている。本ドキュメントでは、基本的に、クエリ処理について話を進める。メモリコンテキストの詳細については、「PostgreSQL 解析資料 ～メモリ管理～」を参考にして欲しい。

1.1. 対象バージョン

本ドキュメントは、PostgreSQL8.0.1 を対象にソースコードの調査を行ったものである。従って、他のバージョンでは、内容が異なる場合があるので注意して頂きたい。

1.2. PostgreSQL のクエリコンパイル処理

node 構造を説明する上で、クエリコンパイルに使われる様々な木構造の名称が出てくるので、ここで、PostgreSQL のクエリコンパイル処理の概要を説明する。

PostgreSQLのクエリ処理の全体の流れを表したのが、図 1-1である。
まず、テキストで渡された SQL 文に対して、字句解析と構文解析を行って、パーズ木を生成する。字句解析とは、文字列から意味のある塊（トークン）を切り出すことである。構文解析とは、字句解析で切り出したトークンを、構文規則に合っているかチェックし、構文規則に合わせて木構造のデータを生成していくことである。こうして作成されたのがパーズ木である。
次に、作成されたパーズ木に対して意味解析を行う。意味解析では、指定されたテーブルが実際にあるか確認したり、複数の SQL 文に置き換えて実行する構文の場合、SQL 文の変換を行ったりする。そして、意味解析の結果としてクエリ木が生成される。
続いて、リライト処理として、ユーザ定義のルールが定義してある場合にはクエリの変換処理を行う。リライトの結果もクエリ木である。
最後に、クエリ木は、プランナ（オプティマイザ）に渡される。プランナでは、論理クエリプランを生成して、最終的には１つの物理実行プランが書かれたプラン木を生成する。このときにパスツリーと呼ばれる、同じ実行結果になる複数の異なる実行プランが生成され、その中で最適なものをコスト

ベースで選択する。最終的に選択されたパスツリーをプラン木に変換し、そのプラン木をクエリエグゼキュータに渡して SQL の処理を実行する。

以上が、PostgreSQL のクエリ処理の概要である。

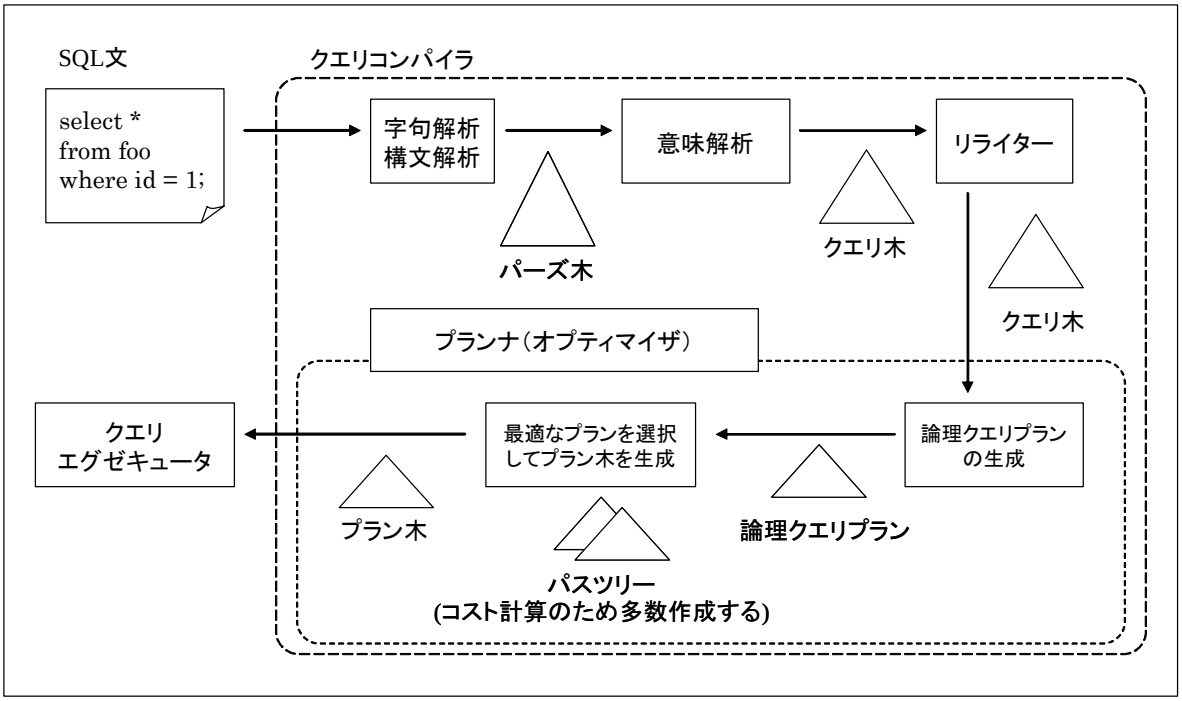

図 1-1 PostgreSQL のクエリコンパイル処理

2. node 構造

PostgreSQL では、SQL のコンパイル処理に使われる木構造を操作するために、src/backend/nodes ディレクトリ以下にある node 構造を使用する。

一番基本的な Node 構造体の構造は、次のようになっている。

```
typedef struct Node
{
    NodeTag     type;
} Node;

#define nodeTag(nodeptr)        (((Node*)(nodeptr))->type)
```

見てのとおり、ノードの型が分かるだけである。ちなみに、define で定義されている nodeTag というマクロで、Node 型の中から type を取り出せるようになっている。
Node 型の使い方は、木構造のデータを Node 型として関数に渡し、関数内で先頭の type をチェックして処理を分岐し、分岐後にキャストを行って処理を継続するというようになる。従って、デバッガなどを使ってソースコードを追っている場合、Node 型を調べる場合には type をチェックして、実際の構造体に当てはめて考える必要がある。

Node 型は、処理分岐の前に使用する型で、実際にノードの処理を行う際には、それぞれ必要な構造体にキャストされる。例えば、パーズ木中のカラム名のノードは次のように定義されている。

```
typedef struct ColumnRef
{
    NodeTag     type;
    List       *fields;         /* field names (list of Value strings) */
    List       *indirection;    /* subscripts (list of A_Indices) */
} ColumnRef;
```

このキャストする様子を図に表すと、図 2-1のようになる。このように、ColumnRef構造体をNode構造体にキャストしても、typeだけは正しく読めるのである。

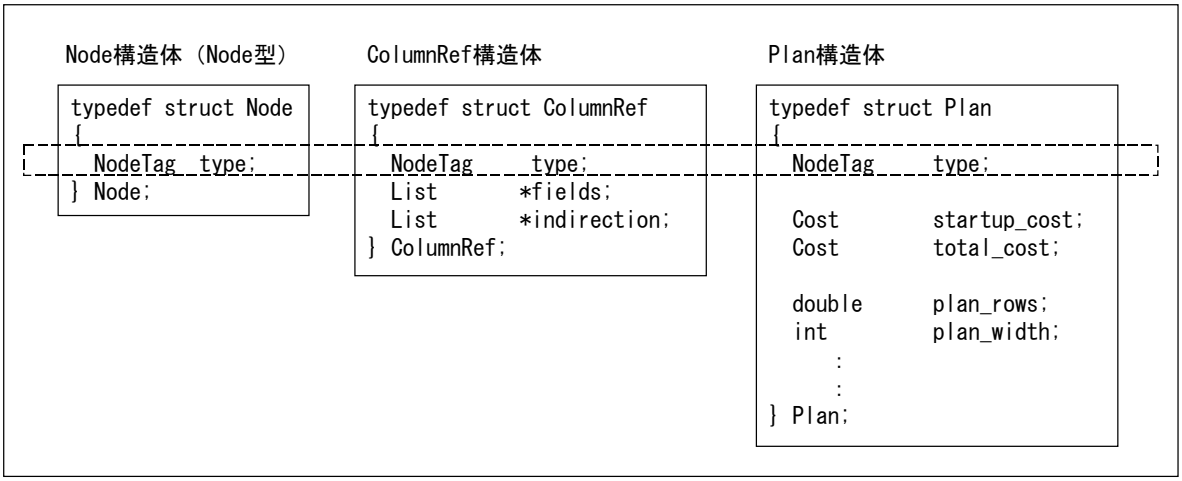

図 2-1 Node 構造体のキャストの仕組み

ColumnRef だけでなく、Plan 構造体やその他多数のノード用の構造体がある。

次に、処理の分岐の例であるが、次のようになっている。まず、ノードの型を取り出すために、前述の nodeTag() マクロにかけ、その結果の型で switch 文で処理を分岐するというようになっている。

```
void *
copyObject(void *from)
{
    void       *retval;

    if (from == NULL)
        return NULL;

    switch (nodeTag(from))
    {
            /*
             * PLAN NODES
             */
        case T_Plan:
            retval = _copyPlan(from);
            break;
        case T_Result:
```

```
            retval = _copyResult(from);
            break;

              :
              :
```

例えば、Node の type が T_Plan であれば、_copyPlan(from) が呼び出される。_copyPlan() では、引数のキャストが行われ、Node 型から Plan 構造体に変換されて処理が継続する。
Plan 構造体は、次のように定義されている。

```
typedef struct Plan
{
    NodeTag     type;

    /*
     * estimated execution costs for plan (see costsize.c for more info)
     */
    Cost        startup_cost;   /* cost expended before fetching any
                                 * tuples */
    Cost        total_cost;     /* total cost (assuming all tuples
                                 * fetched) */

    /*
     * planner's estimate of result size of this plan step
     */
    double      plan_rows;      /* number of rows plan is expected to emit */
    int         plan_width;     /* average row width in bytes */

          :
          :
```

先頭が NodeTag 型の type になっているので、type を見るだけなら Node 型として見ても問題ない訳である。

2.1. NodeTag

ここでは、Node 型の type をもう少し詳しく見ていく。Node 型の type は NodeTag という enum 型で定義されたものである。include/nodes/nodes.h に次のように定義されている。

```
typedef enum NodeTag
{
    T_Invalid = 0,

    /*
     * TAGS FOR EXECUTOR NODES (execnodes.h)
     */
    T_IndexInfo = 10,
    T_ExprContext,
    T_ProjectionInfo,
    T_JunkFilter,
    T_ResultRelInfo,
    T_EState,
    T_TupleTableSlot,

    /*
```

```
        * TAGS FOR PLAN NODES (plannodes.h)
        */
       T_Plan = 100,
       T_Result,

         :
         :
```

Node 型は、様々なモジュールで使われることから、NodeTag に割り当てられる番号が、それぞれの目的毎に分けられている。また、Node 型の type に対応する構造体は、それぞれ次の表のファイルで定義されている。例えば、T_Plan は 100 番であり、T_Plan に対応する Plan 構造体が格納されているのは、plannodes.h ということになる。

Node の種類	構造体の定義されているファイル	type の ID
無効なノード用の ID	nodes.h	0
EXECUTOR 関連	execnodes.h	10～
PLAN 関連	plannodes.h	100 番台
PLAN STATE 関連	execnodes.h	200 番台
PRIMITIVE 関連	primnodes.h	300 番台
EXPRESSION STATE 関連	execnodes.h	400 番台
PLANNER 関連	relation.h	500 番台
MEMORY 関連	memnodes.h	600～649
VALUE 関連	value.h	650～655
リスト関連	pg_list.h	656～658
PARSE TREE 関連	parsenodes.h	700 番台と 800 番台
FUNCTION-CALL CONTEXT と RESULTINFO 関連	fmgr.h	900～

「EXECUTOR 関連」のノードは、実行時にエグゼキュータが使用する。「PLAN 関連」は、プラン木で使われるノードである。「PLAN STATE 関連」については、詳しく調べていないが、エグゼキュータで使われるようである。「PRIMITIVE 関連」は、クエリ木以降、プラン木までの間共通で使われるノードである。「EXPRESSION STATE 関連」のノードは、どこで使われるか調べていない。

「PLANNER 関連」のノードは、パスツリーを作るのに使われる。

「MEMORY 関連」のノードは、クエリ処理とは少し系統が違って、メモリコンテキストの木構造を作るのに使われる。

「VALUE 関連」のノードは、gram.y でクエリ中の定数を入れるためのノードを構築する際に使用される。

「リスト関連」のノードは、クエリ処理で使うリスト構造を構築するのに使われる。例えば、複数のパーズ木をつなぐためのリスト構造に使われる。

「PARSE TREE 関連」のノードは、パーズ木を構成するのに使われる。「PARSE TREE 関連」のノ

ードは、type として 700 番台と 800 番台が使われる。700 番台は SQL の種類にあたるノードが定義されており、800 番台は各 SQL の構成要素に使われるノードが定義されている。
「FUNCTION-CALL CONTEXT と RESULTINFO 関連」も今回は詳しく調べていないが、トリガの実装とエグゼキュータあたりで使われるようである。

2.2. ノード操作の関数

PostgreSQL では、Node 型のデータおよびツリーを操作するために、いくつかの関数が定義されている。まず、Node をサブツリーも含めてコピーする copy 関数群(copyfuncs.c)。Node をサブツリーごと比較するための equal 関数群(equalfuncs.c)。その他に、デバッグ時に Node の内容を表示するための out 関数群(outfuncs.c) である。これらは、それぞれ、src/backend/nodes ディレクトリ以下のファイルとして実装されている。実装の方法もほとんど同じで、例えば、copy を例に挙げると、まず、ローカルに _copyPlan(Plan *from) のような Plan 型用の関数が、type の分だけ定義してある。呼び出し元は 1 箇所で、copyObject(void * from) である。そして、copyObject () 関数の先頭で、nodeType をチェックして、switch 文を使って、型に合わせたローカルなコピー関数を呼び出すようになっている。以下が、copyObject の実際のコードである。

```
void *
copyObject(void *from)
{
    void       *retval;

    if (from == NULL)
        return NULL;

    switch (nodeTag(from))
    {
            /*
             * PLAN NODES
             */
        case T_Plan:
            retval = _copyPlan(from);
            break;
        case T_Result:
            retval = _copyResult(from);
            break;

              :
              :
```

_copyPlan() や _copyResult() などの関数は、copyfuncs.c ファイルの前の方で個別に定義してあり、構造体内のすべてのデータをコピーするように書いてある。その過程で、copyObject が再帰的に呼び出されるところも出てくる。
ここでは、コピーを例に挙げて説明したが、Node 型のオブジェクトを再帰的に表示させる outfuncs は、ほぼ同じ作りである。equalfuncs も引数が２つになることを除けば、ほぼ同じ作りであると言える。

2.3. ソースコード

ノード関連のソースコードは、include/nodes と src/backend/nodes に分かれている。include/nodes 以下には、各 type に対応する Node 型にキャストして読み取れる構造体を定義しているファイルと、node 構造を支援する関数のヘッダという 2 種類のファイルが存在する。

2.3.1. include/nodes 以下のファイル

(※) のついているファイルには、それぞれのフェーズで使われる Node 型に対応する構造体の定義が書かれている。

bitmapset.h	bitmapset.c の外部宣言ファイル。
execnodes.h (※)	エグゼキュータで使われるノードの構造体を定義したファイル。
makefuncs.h	makefuncs.c の外部宣言ファイル。
memnodes.h (※)	メモリコンテキストで使うノードの構造体を定義したファイル。
nodeFuncs.h	nodeFuncs.c の外部宣言ファイル。
nodes.h	NodeTag の enum 型を定義してあるファイル。他にも Node 型の定義や関数の外部宣言を含む。
params.h	プラン木のパラメータに使うデータ構造を定義したファイル。
parsenodes.h (※)	パーザで使うノードの構造体を定義したファイル。
pg_list.h	Node をリストとして操作するためのマクロパッケージ。
plannodes.h (※)	プランナで使われるノードの構造体を定義したファイル。
primnodes.h (※)	パーザ、プランナ、エグゼキュータで共通に使われるプリミティブなノードの構造を定義したファイル。
print.h	print.c の関数の外部宣言用ファイル。
readfuncs.h	readfuncs.c の関数の外部宣言用ファイル。
relation.h (※)	プランナで使用するノードの構造体を定義したファイル。
value.h (※)	定数としてもつ数値関連のノードの構造体を定義したファイル。

2.3.2. src/backend/nodes 以下

node 構造を支援するための関数が定義されたファイルである。

bitmapset.c	Plan ノードや PlanState ノードで使用するビットマップ配列の実装。
copyfuncs.c	Node 型をツリーとして再帰的にコピーするためのパッケージ。
equalfuncs.c	Node 型をツリーとして再帰的に比較するためのパッケージ。
list.c	リスト操作のための実装（マクロを除く）。
makefuncs.c	パーズ時にノードを生成するための関数群。
nodeFuncs.c	ノード操作用の関数。
nodes.c	変数が 1 つ定義されているだけで、関数は 1 つも定義されていない。
outfuncs.c	Node 型を表示可能な文字列に再帰的に置き換えるパッケージ。

params.c	クエリプラン木のパラメータリストの支援関数。
print.c	ツリーの表示の実装（主にデバッグ用）。
read.c	ノード木を文字列で表現したものからトークンを取り出す。readfuncs.c から呼び出される。stringToNode()で、文字列をローカルな pg_strtok_ptr という変数に設定しておき、pg_strtok()を呼び出すと、そこからトークンを取り出す。
readfuncs.c	ノード木を文字列で表現したものからノード木を再構築する。outfuncs.c で生成した nodeToString()の結果からノード木を再構築できる。パスツリーやプラン木に対して使われることはないため、パスツリーやプラン木に対応するコードは実装されていない。
value.c	即値ノードの実装。

主なものは、パーズ木の生成に使われる makefuncs.c、Node 型ツリーの比較に使われる equalfuncs.c、Node 型ツリーのコピーに使われる copyfuncs.c、Node 型ツリーを表示可能な文字列に置き換える outfuncs.c、outfuncs.c で生成した文字列から Node 型ツリーを再構築する readfuncs.c などである。

＜ＥＯＦ＞