

内部を知って業務に活かす

PostgreSQL 研究所

石井 達夫 ISHII Tatsuo
ishii@postgresql.org



第2回 バックエンドの中身を解析(1)

～ パース処理とアナライズ処理



北京出張

4月の中旬に中国は北京に出張で出かけました。反日デモなどの報道があったので内心びくびくしていたのですが、市内は意外と平穏で、特に身の危険を感じることもありませんでした。

筆者は北京は今回で3回目ですが、訪れるたびにどんどん街並みが近代化していくのには驚かされます(筆者が初めて北京を訪れたときは高速道路すらありませんでした)。このままいけば、遅くとも20年後くらいには経済的に日本を追い抜いているのではないのでしょうか。そのくらいのパワーを感じました。

3回目と言えば、今回もまた万里の長城に行ってみました。同行してくれた地元の人に「中国人でも3回も万里の長城に行く人はいない」と呆れられてしまいました。



いよいよPostgreSQLのバックエンドの中身を解析

前回は、PostgreSQLの内部構造を知る上で必要な基礎的事項、すなわち、

- プロセス構造
- ソースツリー
- デバッグを使って実行の流れを追う方法
- tagsの使い方

を説明しました。今回はいよいよPostgreSQLのバックエンドの中身に迫っていきます。



今回からPostgreSQL 8.0.2がターゲット

前回はPostgreSQL 8.0.1を前提にお話しましたが、今回からは5月にリリースされたPostgreSQL 8.0.2のソースを解析の対象にします(もっとも違いはわずかなので、前回の記事内容を特に8.0.2用に変更する必要はないはずです)。

8.0.2は8.0.1に対するバグ修正という位置づけです。実は、8.0.1から8.0.2になるときに1点だけバグ修正以外の変更が入っています。それは、バッファ管理のアルゴリズムがARC(Adaptive Replacement Cache)から2Qに変更されたことです。

PostgreSQL 8.0がリリースされる直前に、IBMがARCの特許を出願していることが発覚しました。特許出願中ということですから、ARCを使ったからといって今ただちに法律上問題になることはありませんが、将来のリスクを避けるに越したことはない、ということで今回の変更になったわけですね。ARCから2Qに変更になったことで気になるのはパフォーマンスですが、開発者によればその差はわずかである、ということでした。なお、次期バージョンの8.1では、さらに改良が加えられたアルゴリズムが採用されることになっています。



問い合わせ処理の流れ

前回はpsqlなどのフロントエンドはまずpostmasterと通信を行い、認証などのセキュリティ関係のチェックを受けたらデータベースエンジンであるpostgresプロセスが起動され、以後フロントエンドはpostmaster

バックエンドの中身を解析 (1)

～ パース処理とアナライズ処理

第2回



ではなく postgres と直接通信を行うということをお話ししました。



バックエンドの処理

以後の流れをバックエンド (postgres) の立場で非常に大雑把に見ると、以下のようになります (図1)。

① 問い合わせの受信

フロントエンドから送られてきた問い合わせ (SQL 文) を受信します。

② パース処理

SQL 文は単なる文字列なので、そのままではコンピュータ処理には向いていません。そこで内部的に扱いやすい「パースツリー」(parse tree) の形に変換します。この段階では、文字通り問い合わせ文字列から得られる情報のみを使用します。したがって、文法的に間違いのない限り、存在しないテーブルを SELECT しようとしてもエラーにはなりません。こうしたことから、この段階のパースツリーは「ローパースツリー」(raw parse tree) とも呼ばれます。

ここでの処理は「パース処理」と呼ばれ、パース処理を行うモジュールを「パーサ」(parser) と呼びます^{注1}。

③ アナライズ処理

パースツリーを解析し、「クエリツリー」(query tree) に変換します。このとき、データベースをアクセスして指定されたテーブルが実際に存在するかどうかチェックし、存在すればテーブル名からOIDに変換するなどの処理が行われます。

ここでの処理は「アナライズ処理」と呼ばれ、アナライズ処理を行うモジュールを「アナライザ」(analyzer) と呼びます^{注2}。

④ リライト処理

PostgreSQL では、クエリを書き換えることによって VIEW や RULE を実装しています。もし必要ならばこの段階でクエリを書き換えます。

ここでの処理は「リライト処理」と呼ばれ、リライト処理を行うモジュールを「リライタ」(rewriter) と呼びます^{注3}。

⑤ プラン処理

クエリツリーを解析し、実際に問い合わせを実行するための「プランツリー」(plan tree) を作成します。

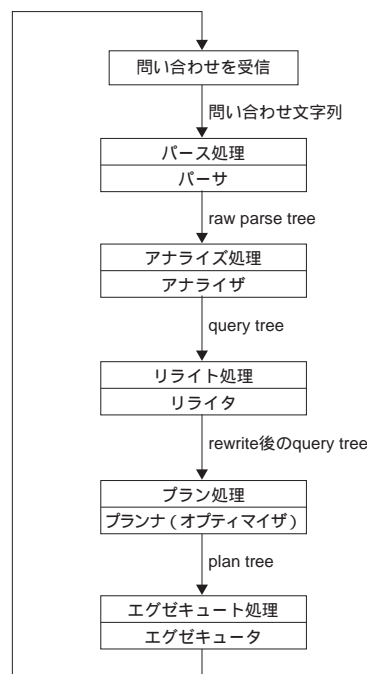
ここでの処理は「プラン処理」と呼ばれますが、実行時間がもっとも短くて済みそうなプランツリーを作成することが非常に大切です。そのため、この処理は「クエリオプティマイズ」(query optimize: 最適化) ないし単に「オプティマイズ」と呼ばれることもあり、こうした処理を行うモジュールをクエリオプティマイザ (query optimizer) ないし、単にオプティマイザと呼びます (あるいはプランナと呼ぶこともあります)^{注4}。

⑥ エグゼキュート処理

プランツリーに従い、問い合わせを実行します。

ここでの処理は「エグゼキュート処理」と呼ばれ、エグゼキュート処理を行うモジュールを「エグゼキュータ」(executor) と呼びます。

図1 問い合わせ処理の流れ



注1) 関連ソースはsrc/backend/parserにあります。

注2) 関連ソースはsrc/backend/parserにあります。

注3) 関連ソースはsrc/backend/rewriteにあります。

注4) 関連ソースはsrc/backend/planにあります。

PostgreSQL 研究所

内部を知って業務に活かす

ータ」(executor)と呼びます^{注5)}。

⑦ 実行結果の送信

実行結果をフロントエンドに送信します。

その後は、再び①に戻ります。

それでは早速、パース処理から順に処理内容を詳しく見ていきましょう。



パース処理



パース処理のソースコード

パース処理のソースコードはsrc/backend/parserにあります。このディレクトリにはアナライズ処理のソースも含まれています。パース処理に関わるソースファイルは以下です。

- gram.c : 構文解析処理
- keywords.c : 予約語表
- parser.c : パース処理メイン
- scan.c : 字句解析処理
- scansup.c : 字句解析処理の補助関数

パース処理のメイン関数はpg_parse_queryで、parser.cで定義されていますが、パース処理のキモは実はgram.cとscan.cです。

パース処理では、文字列としてのSQLを解析する必要があります。もちろん一から文字列処理プログラ

ムを書いていくこともできますが、SQL文法ほどの複雑さのレベルになるとそれはかなり困難なので、専用のツールでSQL文法を表現し、パースするプログラムを自動生成する方法をPostgreSQLでは採用しています。



字句解析処理

字句解析処理とは、文字列から「単語」を切り出す処理のことをさします。たとえば、

```
SELECT * FROM accounts WHERE aid = 100;
```

なら、「」(空白)「SELECT」「*」「FROM」などがすべて単語ということになります。この際に、さらに単語をある程度分類しておきます。たとえば、0~9の数字を組み合わせたものなら整数であるというような感じです。

字句解析処理のソースファイルscan.cは、flexというツール用に書かれたscan.lから自動生成されたものです。scan.lには、flexの文法にしたがって字句解析のルールを書きます。たとえば、先ほどの整数であれば、以下のような記述になります。

- digit : [0-9]
- integer : {digit}+

ご覧のように、基本的にある単語を識別するためのルールを正規表現で書くのがflexの文法である、ということになります。



構文解析処理

字句解析処理によって単語の切り出しができれば、構文解析処理を行います。bisonというツール用に書かれたgram.yから自動生成されたものが構文解析処理のソースファイルgram.cになります。gram.yは構文解析処理の中心になるもので、8000行以上の大きさがあるかなり複雑なものです。すべてを解説するのは誌面の関係でできませんが、ここでは感じだけでも掴んでいただきましょう。

構文解析処理は、基本的にはパターンマッチング処理です。たとえば先ほどのSELECT文は、リスト1のパターンにマッチするはずです。

リスト1 パターンの例

```
simple_select:
    SELECT opt_distinct target_list
    into_clause from_clause where_clause
    group_clause having_clause
    {
        SelectStmt *n = makeNode(SelectStmt);
        n->distinctClause = $2;
        n->targetList = $3;
        n->into = $4;
        n->intoColNames = NIL;
        n->intoHasOids = DEFAULT_OIDS;
        n->fromClause = $5;
        n->whereClause = $6;
        n->groupClause = $7;
        n->havingClause = $8;
        $$ = (Node *)n;
    }
```

注5) 関連ソースはsrc/backend/executorにあります。

バックエンドの中身を解析 (1)

～ パース処理とアナライズ処理

第2回



ここで、「simple_select」はパターンの名称です。右側にはパターンを書きます。このパターンは、

「SELECT」ではじまり、オプションでDISTINCT句 (opt_distinct)、ターゲットリスト (target_list)、INTO句 (into_clause)、FROM句 (from_clause)、WHERE句 (where_clause)、GROUP BY句 (group_clause)、HAVING句 (having_clause) と続く

と定義されています^{注6}。

{ }の中は、パターンマッチが起きたときに実行される「アクション」です。アクションにはC言語の他、bison特有の予約記号を書くことができます。

PostgreSQLのパーサの目的は、パースツリーを作ることです。そこで、まず、

```
SelectStmt *n = makeNode(SelectStmt);
```

によって、SELECT文に対応する構造体SelectStmtを持つノードをmakeNodeという関数で作っています (パースツリーの構造については、後で詳しく述べます)。

以下、構造体のメンバに値をセットしています。たとえば、

```
n->targetList = $3;
```

の\$3は、

```
SELECT opt_distinct target_list
into_clause from_clause where_clause
group_clause having_clause
```

の3番目の部分 (target_list) をtargetListにセットしているわけです。



target_list は別の場所で解析

しかし、このままでは元のSQL文との対応はよくわかりません。というのは、target_listもまたパターンであり、別のところで定義されているからです。その部分はリスト2のようになります。target_listの本体

はtarget_elであり、そのときにlist_make1が呼び出されるというアクションが起こることがわかりました。

では、target_elの定義はどうなっているのでしょうか? (リスト3) いくつかのパターンが「|」で列挙されており、これはどれかのパターンにマッチすればよいことを意味します。

先頭の「a_expr AS ColLabel」は、「i AS this_month」のように、列の別名が付いているパターンです。a_exprやColLabelはさらに別のところで定義されています。

前述のSELECT文のターゲットリストは、「|*」のパターンにマッチします。このときのアクションとしては、ColumnRefという構造体がパースツリーのノードとして作られます。ColumnRefはリスト4 (次ページ) のように定義されます。NodeTagはノードの識別値でenumであり、この場合はT_ColumnRefという値になります^{注7}。

リスト3の「fields」には、列名をリストにしたもの

リスト2 target_listの定義

```
target_list:
    target_el { $$ = list_make1($1); }
```

リスト3 target_elの定義

```
target_el:
    a_expr AS ColLabel
    {
        $$ = makeNode(ResTarget);
        $$->name = $3;
        $$->indirection = NIL;
        $$->val = (Node *)$1;
    }
    | a_expr
    {
        $$ = makeNode(ResTarget);
        $$->name = NULL;
        $$->indirection = NIL;
        $$->val = (Node *)$1;
    }
    | '*'
    {
        ColumnRef *n = makeNode(ColumnRef);
        n->fields = list_make1(makeString(""));

        $$ = makeNode(ResTarget);
        $$->name = NULL;
        $$->indirection = NIL;
        $$->val = (Node *)n;
    }
    ;
```

注6) ここで「SELECT」は予約語であり、keywords.cに定義されています。

注7) makeNodeはマクロであり、引数に「T_」を付けたものをNodeTagにセットするとともに、引数の名前でも構造体のメモリを割り当てます。したがって、構造体の名前にT_を付けたものがノード識別値のenumの値になるという「隠れた規則」があるわけですね。こうした内部的な規則はどこにもドキュメント化されていないので、コードを実際に触りながら推測しなければならないのがちょっとつらいところです。

PostgreSQL 研究所

内部を知って業務に活かす

をセットします。list_make1 は、要素が1個しかないリストを作るマクロです（コラム参照）。また、

```
$$ = makeNode(ResTarget);
```

は、result target（SELECT 文などで検索する列名など）を表現するノードを作り、「上位」のパターンに返却しています。ここで上位のパターンとは、先ほど出てきた simple_select を指します。そのアクションの中に、

```
n->targetList = $$;
```

というのがありましたが、\$\$ でこの部分がアクセスされます。結果として、

```
n->targetList = makeNode(ResTarget);
```

が実行されることになります。

ResTarget はリスト5のような構造体です。ここでは列名は単に「*」なので、name には値なしを表す NULL、indirection は未使用なので NIL、val には先ほど作った ColumnRef 構造体へのポインタをセットしています。



パターンとアクションの組み合わせでパースツリーを構築

あちこちに話が飛んでしまったのでちょっとわかりにくかったかもしれませんが、bison による構文解析では、いきなり詳細な解析を行うのではなく、徐々に詳細な解析を行う「トップダウン」方式な記述が可能

です。また、文法の記述もパターンとアクションの組み合わせなので、手続的にプログラムを書くよりもはるかに見通しがよいことをおわかりいただけたと思います。

こうした構文解析の結果、メモリ上にパースツリーが作成されます。

```
SELECT * FROM accounts WHERE aid = 100;
```

のパースツリーを図2に示します^{注8}。メインになるのは SelectStmt という構造体です。その他の情報はこの構造体にぶら下がる形になっています。



アナライズ処理



生成される情報

アナライズ処理では、実際にシステムカタログなどを検索しながら必要な情報を補ってクエリツリーを作成します。この段階で生成される情報としては、以下のようなものがあります。

① テーブルOID

文字列で指定されたテーブルを、システムカタログの「pg_class」を検索することによって確定し、OID（オブジェクトID）を求めます。

スキーマ名で明示的に修飾されたテーブルは対応するテーブルの実体が確定するので、容易にOIDが求め

られます。スキーマ名で修飾されていない場合は、スキーマサーチパスを検索してスキーマを確定し、OIDを求めます。

テーブルを継承しているテーブルがある場合は、それらのテーブルのOIDリストも作られます。

② 列名リスト

SELECT * FROM ... のように列名が省略されている場合は、①でテーブルOIDを求めた後、システムカタログを検索す

リスト4 ColumnRef の定義

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

リスト5 ResTarget の定義

```
typedef struct ResTarget
{
    NodeTag      type;
    char         *name; /* column name or NULL */
    List         *indirection; /* subscripts and field names, or NIL */
    Node         *val; /* the value expression to compute or
                        * assign */
} ResTarget;
```

注8) postgresql.conf の debug_print_parse を true にすることにより、パースツリーを出力することができるようになっていますが、実際に出力されるのはパースツリーではなく、本稿で言うところのクエリツリーです。本稿を執筆するにあたり、若干ソースコードに手を加えてアナライズ前のパースツリーを表示させて検証を行いました。

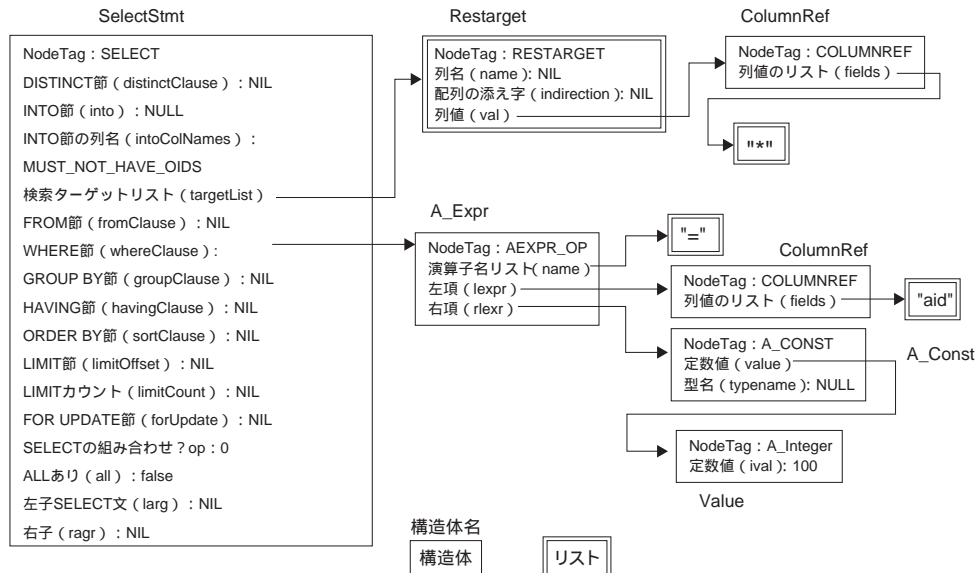
バックエンドの中身を解析 (1)

～ パース処理とアナライズ処理

第2回



図2 SELECT * FROM accounts WHERE aid = 100 のパースツリー



■ PostgreSQL のリストパッケージ

パースツリーなど、複雑なリスト要素を作成するために、PostgreSQL ではリストを扱う機能が充実しています。ソースコードはsrc/backend/nodes/list.cにあり、そのためのインタフェースはsrc/include/nodes/pg_list.hに定義されています。

PostgreSQL のリストは一方方向の単純なリンク付きリストで、3つのタイプがあります。

- T_List : ポインタのリスト
- T_IntList : 整数のリスト
- T_OidList : OIDのリスト

パースツリーなどで使われるのは主にT_Listです。PostgreSQL では、リストで連結されたデータを「セル」(cell) と呼びます。

リストを管理する構造体は以下のようになっており、リストの長さ、最初のセルへのポインタ、最後のセルへのポインタが管理されています。

```
typedef struct List
{
    /* T_List, T_IntList, or T_OidList */
    NodeTag    type;
    int        length;
    ListCell   *head;
```

```
ListCell   *tail;
} List;
```

各セルの構造は以下のようになっています。

```
struct ListCell
{
    union
    {
        void    *ptr_value;
        int     int_value;
        Oid     oid_value;
    }          data;
    ListCell   *next;
};
```

基本的にはデータと次のセルへのポインタが管理されています。なお、空のリストは、List 構造体へのポインタが「NIL」(実際には0)であることで表現されます。したがって、List 構造体が存在するということは、少なくともlengthが1以上であるということになります。

整数値1, 2, 3の3つのセルを持つリストの例を図Aに示します。図Aは、整数1, 2, 3をリストにして

次ページに続く ➡

PostgreSQL 研究所

内部を知って業務に活かす

ることによって明示的な列名のリストに変換します。

③ 型 OID

WHERE 句などで指定されている列や定数の型名を確定し、型 OID を求めます。型が明示的に指定されていないものに関しては、型の推測が行われます。

④ オペレータ OID

オペレータ（演算子）は PostgreSQL では関数呼び出しとして実装されています。同じ「=」といったオペレータでも、その両側に来るデータ型によって、内部的に呼び出される関数が異なってきます。そこでア

ナライザはそれらの型情報を使ってシステムカタログ「pg_operator」を検索し、オペレータの OID を確定して適切な関数を呼び出すことができますようにします。



クエリツリー

アナライズ処理で作成されるメインの構造体は Query です。パース処理では、文の種類（SELECT、INSERT、UPDATE、.....など）の違いによって作成される構造体も分かれていましたが、アナライズ処理の結果は単一の Query 構造体です。

アナライズ処理の結果生成されたクエリツリーを図 3 に示します。

■ PostgreSQL のリストパッケージ（続き）

アクセスインタフェースも示したものです。現在のセルが「1」のセルを指しているものとします。

すべてのインタフェースを解説するのは誌面の関係で無理なので、代表的なものを挙げておきます（表 A）。これらを知っているだけでもソースを読むのが楽になると思います。

- lnext : 次のセルへのポインタ
- lfirst : 現在のセルのデータ（data）
- lsecond : 次のセルのデータ
- lthird : その次のデータ
- linitial : リスト中の最初のセルのデータ
- llast : リスト中の最後のセルのデータ

図 A リストの例

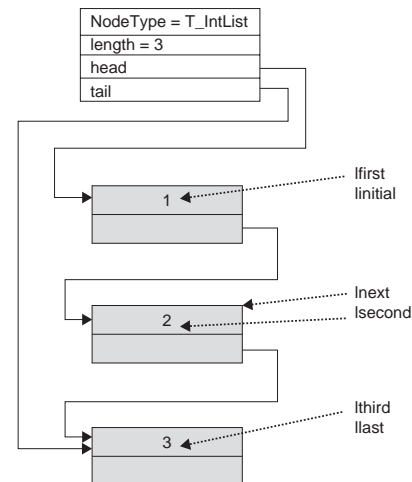


表 A アクセスインタフェース

関数	機能
list_make1(x1)	x1 をセルに持つリストを作る
list_make2(x1,x2)	x1, x2 をセルに持つリストを作る
list_make3(x1,x2,x3)	x1, x2, x3 をセルに持つリストを作る
foreach(cell, l)	リスト l のすべてのセルを順にアクセスする。cell は現在の cell へのポインタ
lappend(List *list, void *datum)	リスト list にデータ datum を持つセルを追加する
list_concat(List *list1, List *list2)	リスト list1 の末尾にリスト list2 を追加する
list_copy(List *list)	リスト list をコピーする
list_free(List *list)	リスト list のメモリを list 自身も含めて解放する。ただし、セルの指すポインタの先のメモリは解放しない
list_free_deep(List *list)	リスト list のメモリを list 自身も含めて解放する。セルの指すポインタの先のメモリも解放する

バックエンドの中身を解析(1) ～ パース処理とアナライズ処理

第2回



次回は？

今回はパース処理とアナライズ処理について解説しました．次回はリライト処理から解説を進めていきま

す．そのほか，PostgreSQL の内部を理解する上で欠かすことのできないメモリ管理サブシステム，システムカタログキャッシュのようなサブシステムについても触れていきたいと思います．**Web**

図3 クエリツリー

