実践テクニックをご紹介

徒然PostgreSQL散策

第4回 ユーザ定義関数の魅力（2）

日本PostgreSQLユーザ会 理事長　石井 達夫
ISHII Tatsuo●ishii@postgresql.jp

PostgreSQL 7.4 リリース！

1年ぶりにPostgreSQLがメジャーバージョンアップされ，PostgreSQL 7.4となりました．PostgreSQL 7.4の改良点や変更点についてはすでに各所で報道されているのでここでは詳細には触れませんが，筆者なりにPostgreSQL 7.4を総括すると「地味ながらも性能や使い勝手の向上した，使い込んでみたいリリース」ということになります．実際に業務で使用するためには，もう少しマイナーバージョンアップして7.4.2とか7.4.3くらいになってからのほうがよいと思いますが，これから開発を始めるようなプロジェクトでは7.4がお勧めです．

ユーザ定義関数の書き方

前回お約束したとおり，今回はまずユーザ定義関数，とりわけC関数の書き方の解説を行います．すでにPostgreSQL 7.4がリリースされているので，今回からはPostgreSQL 7.4を前提に記法や機能を解説することにします．

ユーザ定義C関数とは？

ユーザ定義C関数はC言語で書かれた関数で，PostgreSQLのバックエンド（データベースエンジン）はコンパイル済のオブジェクトを実行時に動的にロードして実行します．ユーザ定義関数を作るためには次のようなステップが必要です．

① CREATE FUNCTION文で関数の定義をデータベースに登録
② ユーザ定義C関数をPostgreSQLの作法にしたがって記述
③ C関数をコンパイルしてオブジェクトを作成

CREATE FUNCTION文

まずは関数を定義するためのSQL文であるCREATE FUNCTION文を詳しく見てみましょう．

CREATE FUNCTIONの構文は図1のようになります．ここで，WITH...以下は過去のバージョンとの互換性のために残されているだけですので，利用は推奨されていません．本稿でも以下，WITHに関しては解説を行いません．

では，構文要素を見ていきましょう．

●OR REPLACE

このオプションを付けると，同じ名前と引数を持つ関数がすでに定義済の場合，すなわち同じシグネチャを持つ関数が定義済の場合，その関数を新しい定義で置き換えます[注1]．

注1）「名前と引数」を「関数のシグネチャ」と言います．つまり，PostgreSQLでは，同じシグネチャを持つ関数は同じものであると見なされます．これは，関数を定義する言語が異なっても同じです．ただし，関数の戻り値の型はシグネチャを構成しませんので，まったく同じ名前と引数を持ち，関数の戻り値の型だけ異なる関数は同じものと見なされます．

●name

（スキーマ名を含む）関数の名前です．引数の型や数が異なっていさえすれば（シグネチャが異なっていれば）他の関数と名前は同じでも構いません．一般に，このような関数の定義機能を関数のオーバーローディングと呼びます．図2にそのような例を示します．この例では，見かけ上，2つの整数型，あるいはTEXT型の引数を受け取り，大きいほうを返すmymaxという同じ名前の関数を定義しています．

●argtype

型を引数の順に並べます．PostgreSQLが型として認識するものならなんでも使えます．「本物」の型の他に，C関数では表1のような仮想的な型が使えます．

●rettype

関数の戻り値の型です．argtypeで説明したのと同様の型が使えます．これに加えて「SETOF」というキーワードを追加できます．SETOFが指定されると，この関数は複数の行を返す関数であると見なされます．たとえば，rettypeがSETOFintegerと定義されたfooという関数は，以下のように呼び出すことができます．

```
SELECT * FROM foo();
```

このように，foo()はテーブルと同様に扱うことができるようになるので，このような関数をテーブル関数と呼ぶことがあります．

●langname

ユーザ定義関数を書くための言語の指定です[注2]．標準では表2のものが使えます．

▼図1　CREATE FUNCTIONの構文

```
CREATE [ OR REPLACE ] FUNCTION name ( [ argtype [, ...] ] )
    RETURNS rettype
  { LANGUAGE langname
    | IMMUTABLE | STABLE | VOLATILE
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ]
```

▼図2　名前が同じで引数の型が異なる関数の例

```
CREATE OR REPLACE FUNCTION mymax(INTEGER, INTEGER) RETURNS
INTEGER
LANGUAGE 'sql' AS '
SELECT CASE WHEN $1 > $2 THEN $1 ELSE $2 END;
';

SELECT mymax(10,1);
 mymax
-------
    10
(1 row)

CREATE OR REPLACE FUNCTION mymax(TEXT, TEXT) RETURNS TEXT
LANGUAGE 'sql' AS '
SELECT CASE WHEN $1 > $2 THEN $1 ELSE $2 END;
';

SELECT mymax('abc','def');
 mymax
-------
 def
(1 row)
```

表1◎仮想的な型

●型名	●内容
CSTRING	C言語の文字列
RECORD	複数のデータからなるデータ（行データ）
ANYELEMENT	どんな型にもマッチ（PostgreSQL 7.4以降で使用可能）
ANYARRAY	anyelementの配列（PostgreSQL 7.4以降で使用可能）

表2◎定義関数で使える言語

●SQL	●SQLで書く関数
PLPGSQL	PL/pgSQLという構造化されたSQLで書く関数
C	C言語で書く関数
INTERNAL	PostgreSQLにあらかじめ組み込まれた関数

●IMMUTABLE | STABLE | VOLATILE

複数回の関数の実行結果を同じとみなしてよいかどうかの指針をオプティマイザに与えます．デフォルトは「VOLATILE」です．

注2）正確には「言語ハンドラ」の指定で，PostgreSQLでは言語ハンドラをユーザが定義することができるので，任意の言語をユーザ定義関数記述用に追加できます．Perl，Tcl，Python，Ruby，PHPなど，多数の言語ハンドラが開発されています．

・IMMUTABLE

この属性を持つ関数は，データベースの内容いかんにかかわらず，同じ結果を返すと見なされます．したがって，同じデータベースセッションの中では，IMMUTABLEな関数は2回目以降常に同じ結果を返します．

・STABLE

この属性を持つ関数は，ある1つのSQL文の中では，関数の実行結果が変らないと見なされます[注3]．

・VOLATILE

この属性を持つ関数は，呼ばれるたびに違う結果を返すと見なされます．setval( )のように副作用を持つ関数もVOLATILEです．

●CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT

NULLに関する扱いを指定します．

・CALLED ON NULL INPUT

引数にNULLが含まれていてもそのまま関数が呼ばれます．NULLをチェックするのは関数側の責任になります（デフォルト）．

・RETURNS NULL ON NULL INPUT | STRICT

どちらも同じ意味で，引数にNULLが含まれている場合，関数が呼び出されることはなく，その代わりに関数の値としてNULLが呼び出し元に返ります．

●[EXTERNAL] SECURITY INVOKER | [EXTERNAL] SECURITY DEFINER

関数の実行権限に関する指定です．なお，[EXTERNAL]は省略可能ですが，このキーワードはSQL標準との互換性のために用意されています．

・[EXTERNAL] SECURITY INVOKER

関数は，関数を呼び出したユーザの権限で実行されます（デフォルト）．

・[EXTERNAL] SECURITY DEFINER

関数は，関数を定義したユーザの権限で実行されます．

● AS 'definition'

関数の定義本体です．C言語以外の関数ではそのまま関数の定義を書きます．

● AS 'obj_file', 'link_symbol'

C言語関数にのみ存在する属性です．これについてはのちほど説明します．

C関数の開発環境

関数の定義方法がわかったので，早速コーディング方法の説明……といきたいところですが，C関数では関数のコンパイルとリンクという関門が立ちはだかっています．PostgreSQLの付属マニュアルの「33.7. C-Language Functions」[注4]にも一応関数のコンパイルとリンク方法が書いてあるのですが，あまりお勧めできません．というのは，プラットフォームごとにコンパイル方法が違っていたり，Makefileの書き方が書かれておらず，実際のプロジェクトではあまり役に立たないであろうと思われるからです．

contribの環境を流用

そこでお勧めしたいのが，contrib以下にある適当なユーザ定義関数を流用することです[注5]．

この方法では，PostgreSQLをconfigureした後のソースツリーを利用します．configureを実行した後，PostgreSQLは移植性を高めるために，そのプラットフォームに最適なMakefileを生成します．ユーザ定義関数を作成するMakefileも基本的にそれをincludeするようにしておけば，プラットフォームによらないポータブルなソースをコーディングできますし，Makefileも手に入るので一石二鳥です．

注3）このような属性を持つ関数の例としては，CURRENT_TIMESTAMPが挙げられます．なぜなら，CURRENT_TIMESTAMPの実行結果は同じトランザクションの中では常に同じであり，1つのSQL文が複数のトランクザションに跨ることはないからです．
注4）本稿ではPostgreSQL 7.4のマニュアルの章番号を採用しています．
注5）実は前回ご紹介したexttableもこの方法で作られています．

exttableを例にとり，この方法で作られる開発環境のファイルを説明します．あらかじめ，ftp://ftp.sra.co.jp/pub/cmd/postgres/exttable/exttable-0.4.tar.gzを取得し，適当なディレクトリに展開しておいてください[注6]．また，PostgreSQLのソースツリーも必要です．ここでは，postgresql-7.4.tar.gzを/usr/local/src/以下に展開，configureまでできているものとします．

exttable-0.4/以下のファイルとカスタマイズ方法を説明します．

● Makefile（図3）

基本的に，“exttable”と書いてあるところを，自分の作りたい関数の名前に置き換えるだけです．

このMakefileは，ソースツリーをPostgreSQLのソースツリーのcontrib/に置くことを前提にしているため，

```
top_builddir = ../..
```

となっていますが，任意の場所でmakeするためには，たとえば

```
top_builddir = /usr/local/src/postgresql-7.4
```

とすればOKです．

● README.exttable，README.exttable.euc_jp

これらは単なる解説書なので，特にフォーマットの指定はありません．ファイル名はMakefileの「DOCS」のところで指定しているものと合っていればどんなものでも構いませんが，慣習として英語版のREADMEはREADME.<関数名>，日本語版はREADME.<関数名>.<エンコーディング>とすることが多いようです．

● exttable.sql

ユーザ定義関数を登録するためのSQL文を格納したファイルです．<関数名>.sqlという名前にしますが，このファイルを直接作成するのではなく，その元ネタになる<関数名>.sql.inという名前のファイルを作ります．makeすると<関数名>.sqlが作成されるしくみになっています．

図4にexttable.sql.inを示します．ここで，MODULE_PATHNAMEが置き換えられる部分です（実際には，「$libdir/exttable」に置き換えられます）．

C関数の書き方

いよいよC関数本体の書き方を解説することにしましょう．ここでは，exttable.c（リスト1）を例にとって解説します．

ヘッダファイル

リスト1-①のヘッダファイルのincludeのうち，必須ヘッダファイルはpostgres.hで，PostgreSQLの最も基本的な定義を含みます．funcapi.hは，行を返すユーザ定義関数を書くための構造体などが定義されて

▼図3　exttableで使われているMakefile

```
#-------------------------------------------------------
#
# exttable Makefile
#
# $Id: Makefile,v 1.4 2003/11/10 08:31:54 t-ishii Exp $
#
#-------------------------------------------------------

subdir = contrib/exttable
top_builddir = ../..
include $(top_builddir)/src/Makefile.global

MODULE_big  := exttable
SRCS        += exttable.c
OBJS        := $(SRCS:.c=.o)
DOCS        := README.exttable README.exttable.euc_jp
DATA_built  := exttable.sql

PG_CPPFLAGS :=
SHLIB_LINK  :=

include $(top_srcdir)/contrib/contrib-global.mk
```

▼図4　exttable.sql.in

```
-- Adjust this setting to control where the objects get created.
SET search_path = public;

SET autocommit TO 'on';

CREATE OR REPLACE FUNCTION exttable(text, text)
RETURNS SETOF RECORD
LANGUAGE C
RETURNS NULL ON NULL INPUT
AS 'MODULE_PATHNAME', 'exttable';
```

注6）前回ご紹介したexttable-0.2には関数を登録するSQL文に間違いがあったので，0.4にバージョンアップしました．

▼リスト1　exttable.c

```
～省略～

#include "postgres.h" ──────────────────────────┐
#include "funcapi.h"                            │
#include "access/heapam.h"                      │
#include "access/tupdesc.h"     /* TupleDesc */ │ ①
#include "catalog/pg_type.h"                    │
#include "nodes/execnodes.h"    /* ReturnSetInfo */
#include "utils/builtins.h"     /* textout() */ │
#include "utils/syscache.h"     /* SearchSysCache() */ ──┘

PG_FUNCTION_INFO_V1(exttable); ────②

extern Datum exttable(PG_FUNCTION_ARGS); ────③

/* ----------
 * exttable
 * read an external tables and returns it
 *
 * C FUNCTION definition
 * exttable(text, char) returns setof record
 *
 * Calling sequence
 * SELECT exttable('/full/path/to/file', '\t') AS (f1 int, f2 text, ....);
 * ----------
 */

typedef struct
{
    int         ntuples;    /* tuple count */
    int         cnt;    /* tuple counter */
    char ***cols;       /* actual data from a tab separated file */
} exttable_status;

static void readfile(char *path, exttable_status *status, int natts, char delimiter);

Datum
exttable(PG_FUNCTION_ARGS)
{
    FuncCallContext *funcctx;
    exttable_status *mystatus;
    ReturnSetInfo   *returnset_info;
    TupleDesc   desc;   /* tuple descriptor expected by caller */

    returnset_info = (ReturnSetInfo *)fcinfo->resultinfo;
    desc = returnset_info->expectedDesc;

    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;
        TupleDesc   tupdesc;    /* result tuple descriptor */
        char *path;
        BpChar *p;
        char delimiter;

        /* create a function context for cross-call persistence */
        funcctx = SRF_FIRSTCALL_INIT();

        /*
         * switch to memory context appropriate for multiple function
         * calls
         */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

        /* build tupdesc for result tuples from expected desc */
        tupdesc = CreateTupleDesc(desc->natts, false, desc->attrs);
        funcctx->slot = TupleDescGetSlot(tupdesc);

        mystatus = (exttable_status *) palloc(sizeof(*mystatus));
        funcctx->user_fctx = (void *) mystatus;
```

▼リスト1　exttable.c（続き）

```
        /* read the file and set actuaul file contents */
        path = DatumGetCString(DirectFunctionCall1(textout, (Datum)PG_GETARG_TEXT_P(0)));
        p = PG_GETARG_BPCHAR_P(1);
        delimiter = (char)*(VARDATA(p));
        readfile(path, mystatus, desc->natts, delimiter);

        mystatus->cnt = 0;      /* reset tuple counter */

        MemoryContextSwitchTo(oldcontext);
    }

    funcctx = SRF_PERCALL_SETUP();
    mystatus = (exttable_status *) funcctx->user_fctx;

    if (mystatus->cnt < mystatus->ntuples)
    {

～省略～

        /* loop for number of attributes specified by
           as t1(i int, j int, k text) etc. */
        for (i=0;i < desc->natts;i++)
        {
            HeapTuple   tuple;
            Form_pg_type typetuple;
            Oid typeoid;

            /* get pg_type tuple oid */
            typeoid = desc->attrs[i]->atttypid;

            /* get a pg_type tuple by oid */
            tuple = SearchSysCache(TYPEOID,
                                   ObjectIdGetDatum(typeoid),
                                   0, 0, 0);
            if (!HeapTupleIsValid(tuple))
                elog(ERROR, "Cannot find type tuple");

            /* call type input function according to it's data type */
            typetuple = (Form_pg_type)GETSTRUCT(tuple);
            values[i] = OidFunctionCall1(typetuple->typinput,
                                         CStringGetDatum(mystatus->cols[mystatus->cnt][i]));
            ReleaseSysCache(tuple);
        }

        tuple = heap_formtuple(funcctx->slot->ttc_tupleDescriptor,
                               values, nulls);
        result = TupleGetDatum(funcctx->slot, tuple);

        pfree(values);
        pfree(nulls);

        mystatus->cnt++;

        SRF_RETURN_NEXT(funcctx, result);
    }
    SRF_RETURN_DONE(funcctx);
}

#include "lib/stringinfo.h"

static void readfile(char *path, exttable_status *status, int natts, char delimiter)
{
～省略～
}
```

います．ここではユーザ定義関数を書くときの必須ヘッダファイルであるfmgr.hをincludeしていませんが，これはfuncapi.hにfmgr.hのincludeが含まれているためです．

これ以外のヘッダファイルについては後で説明します．

ファンクションマネージャとのインタフェース

PostgreSQLでは，すべてのユーザ定義関数はファンクションマネージャというサブシステムが管理しています．ファンクションマネージャへのインタフェースの最新版は「バージョン1」です．ある関数がバージョン1のインタフェースに従う関数であることを宣言するのが，PG_FUNCTION_INFO_V1マクロです（リスト1-②）．このマクロの引数にはこれから定義する関数名を指定します．

このマクロは以下のようにfmgr.hで宣言されています．

```
#define PG_FUNCTION_INFO_V1(funcname) \
extern Pg_finfo_record * CppConcat(pg_finfo_,funcname) (void); \
Pg_finfo_record * \
CppConcat(pg_finfo_,funcname) (void) \
{ \
    static Pg_finfo_record my_finfo = { 1 }; \
    return &my_finfo; \
} \
extern int no_such_variable
```

このままではわかりにくいので，実際にマクロが展開された後のソースコードに見やすく改行やインデントを施したのが以下です．

```
extern Pg_finfo_record *pg_finfo_exttable(void);
Pg_finfo_record *pg_finfo_exttable(void)
{
  static Pg_finfo_record my_finfo = { 1 };
  return &my_finfo;
}
extern int no_such_variable;
```

ご覧のように，「pg_info_関数名」という名前のstatic関数が定義されているのがわかります．この関数は単に1というデータを含む構造体へのポインタを返しているだけですが，ファンクションマネージャは「pg_info_関数名」という関数が存在していること，そしてこの関数が1を返すことをもって，この関数がバージョン1の関数であることを認識しているのです．

では，PG_FUNCTION_INFO_V1が定義されていない場合はどうなるのでしょう？ この場合は「バージョン0」関数として扱われます[注7]．

ユーザ定義関数のプロトタイプ宣言

ユーザ定義関数のプロトタイプ宣言は，すべて同じになります（リスト1-③）．

```
extern Datum exttable(PG_FUNCTION_ARGS);
```

ここでDatumの実体はunsigned longで，マシンに

▼リスト2　FunctionCallInfoData構造体

```
typedef struct FunctionCallInfoData
{
    FmgrInfo    *flinfo;     /* ptr to lookup info used for this call */
    struct Node *context;    /* pass info about context of call */
    struct Node *resultinfo; /* pass or return extra info about result */
    bool        isnull;     /* function must set true if result is
                             * NULL */
    short       nargs;      /* # arguments actually passed */
    Datum       arg[FUNC_MAX_ARGS]; /* Arguments passed to function */
    bool        argnull[FUNC_MAX_ARGS]; /* T if arg[i] is actually NULL */
} FunctionCallInfoData;
```

注7）何かずいぶん回りくどいやり方のような気がしますが，バージョン1以前に作られた関数，すなわちバージョン0関数との互換性を保つためです．「バージョン0」関数は引数や戻り値としてNULLが扱えないなどの致命的な問題を持っていましたが，この問題はバージョン1インタフェースが使えるようになったPostgreSQL 7.1以降でようやく改善されました．今から考えると随分アバウトな話ですが:-)，このあたりの事情はsrc/backend/utils/fmgr/READMEに書いてあります．

よって32ビットまたは64ビット長になります．PostgreSQLでは，関数の戻り値がDatumのサイズに収まる範囲の大きさであれば値自身を（pass by value），そうでなければ値へのポインタを返す（pass by reference）ようになっています（引数に関しても同様の扱いになります）．

PG_FUNCTION_ARGSはマクロで，展開した後はリスト1-③は以下のようになります．

```
extern Datum exttable(FunctionCallInfo fcinfo );
```

つまり，ユーザ定義関数の引数は常に1個です．では，引数が複数個あるexttableのような関数ではどうしたらよいのでしょう？

fcinfoの実体はリスト2のような構造体へのポインタです．ファンクションマネージャは，あらかじめ引数（arg）や引数がNULLかどうかのフラグ（argnull）をセットした状態でユーザ定義関数を呼び出しますので，引数の数にかかわらず，ユーザ定義関数側ではこの構造体をチェックすることによって引数の値を取得するなどの処理が可能です．もっとも，引数の取得や関数の戻り値のセットのための各種マクロが用意されているので，通常FunctionCallInfoData構造体の中身を意識する必要はありません．これについては後で述べます．

最後に

exttable( )の解説の途中ではありますが，予定よりも誌面がオーバーしてしまい，前回約束したdblink( )の説明ができなくなってしまいました．次回こそはexttable( )の残りとdblink( )の解説をしますので，ご容赦ください．

そういえば，本誌が発行されるころにはもうすっかり暮れも押し迫っていますね．それでは皆様よいお年をお迎えください．そして来年もまたよろしくお願いします．WD

C O L U M N

JPUG四国支部設立記念セミナー開催される

広島，北海道，関西，九州に続き，日本PostgreSQLユーザ会四国支部が設立されたのを記念し，11月14日に四国の高松市で記念セミナー『日本PostgreSQLユーザ会四国支部設立記念講演会』が開催されました．

セミナーは，JPUGの理事でもあり，また初代四国支部長に就任された大垣靖男氏が司会を担当されました．講演は主にJPUGの理事が担当し，PostgreSQL用の国産クラスタソフトPGCluster，ビジネスや地方自治体での活用事例，PostgreSQLをベースにした商用データベースの紹介など，興味深い話題が次々と紹介されました．

また，会場に併設された展示コーナーでは，JPUGやPostgreSQL関連企業のブース展示が行われました．特にJPUGのブースでは，マグカップや携帯ストラップなど，できたばかりの5周年記念JPUG特製グッズが配布され，来場者の注目を浴びていました．

セミナーが終わったあとは，いつものようにその土地ならではの味覚の探求です．四国と言えば讃岐うどんということで，大垣氏をはじめ，地元の方々に案内していただいて3軒のお店で讃岐うどんを堪能しました．

四国のゆったりした風土の中で，ごく自然体にPostgreSQLに取り組んでおられる方々との出会いが印象に残った旅でした．

▲講演の様子