実践テクニックをご紹介

徒然PostgreSQL散策

第5回 ユーザ定義関数の魅力（3）

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

ニューヨーク大寒波

まだまだ寒い冬が続いていますが，読者の皆さんいかがお過ごしでしょうか．この原稿を執筆しているのは冬真っ盛りの1月中旬，しかも場所は仕事の都合でニューヨークのホテルの一室です．ただでさえ寒いニューヨークは現在大寒波に見舞われており，零下10度を下回ることもあるとんでもない寒さです．ある夜“Blue MAN Group”というショーを見に行ったときに，あまりの寒さに記念品の帽子（写真1）を買ってしまいましたが，これが実に役に立ちました（たぶん日本では二度と使うことはないと思いますけど:-）．

前回に引き続きexttable本体を解説

前回はPostgreSQLのユーザ定義C関数の書き方をexttable（リスト1）という関数を例にとって説明しました．今回はその続きで，いよいよexttable本体の説明を行います．

ReturnSetInfoの取得

exttableのようにRECORD型を返す関数では，関数の呼び出し時に動的に返す行を構成する各データの型が決まります．たとえば，exttableを

```
SELECT * FROM exttable('/tmp/sample.txt',
'\t') AS
交通費(日付 DATE, 目的地 TEXT , 目的 TEXT,
金額 INTEGER);
```

のように呼び出したときは，各列の型がDATE，TEXT，TEXT，INTEGERになるわけです．この情報を取得しているのがリスト1-①で，この結果はリスト2のような構造体になります．これは列の構造を記述する構造体で，列の数（natts），型（attrs），制約（constr）などの情報を含みます．今回使用するのはnattsとattrsです．

初回呼び出し時の処理

exttableのように複数の行を返す関数では，行数分だけ関数が何度も呼ばれます．初回呼び出しのときには，各種初期処理が必要になります．初回呼び出しかどうかの判定はSRF_IS_FIRSTCALLマクロで行います（リスト1-③）．

●初期化処理

まず，SRF_FIRSTCALL_INITを呼び出して初期化

写真1◎寒さに耐えきれずニューヨークで買ってしまった帽子

▼リスト2　tupleDesc構造体

```
typedef struct tupleDesc
{
    int         natts;      /* Number of attributes in the tuple */
    Form_pg_attribute *attrs;
    /* attrs[N] is a pointer to the description of Attribute Number N+1. */
    TupleConstr *constr;
    bool        tdhasoid;   /* Tuple has oid attribute in its header */
}   *TupleDesc;
```

▼リスト1　exttable.c

```
～省略～

typedef struct
{
    int         ntuples;    /* tuple count */
    int         cnt;    /* tuple counter */
    char ***cols;       /* actual data from a tab separated file */
} exttable_status;                                                    ①

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;                              ②

    if (SRF_IS_FIRSTCALL())                                           ③
    {
        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();                               ④

        /*
         * switch to memory context appropriate for multiple function
         * calls
         */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);   ⑤

        /* build tupdesc for result tuples from expected desc */
        tupdesc = CreateTupleDesc(desc->natts, false, desc->attrs);   ⑥
        funcctx->slot = TupleDescGetSlot(tupdesc);                    ⑦

        mystatus = (exttable_status *) palloc(sizeof(*mystatus));     ⑧
        funcctx->user_fctx = (void *) mystatus;                       ⑨



        /* read the file and set actuaul file contents */
        path = DatumGetCString(DirectFunctionCall1(textout, (Datum)PG_GETARG_TEXT_P(0)));   ⑩
        p = PG_GETARG_BPCHAR_P(1);
        delimiter = (char)*(VARDATA(p));
        readfile(path, mystatus, desc->natts, delimiter);             ⑪

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

        MemoryContextSwitchTo(oldcontext);
    }

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

    if (mystatus->cnt < mystatus->ntuples)                            ⑫
    {

～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++)
        {
```

▼リスト1　exttable.c（続き）

```
            HeapTuple   tuple;
            Form_pg_type typetuple;
            Oid typeoid;

            /* get pg_type tuple oid */
            typeoid = desc->attrs[i]->atttypid; ———⑬

            /* 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); ———⑭
            values[i] = OidFunctionCall1(typetuple->typinput, ———┐
                                         CStringGetDatum(mystatus->cols[mystatus->cnt][i])); ┘⑮
            ReleaseSysCache(tuple); ———⑯
        }

        tuple = heap_formtuple(funcctx->slot->ttc_tupleDescriptor, ———┐
                               values, nulls);                         ⑰
        result = TupleGetDatum(funcctx->slot, tuple); ———┘

        pfree(values);
        pfree(nulls);

        mystatus->cnt++;

        SRF_RETURN_NEXT(funcctx, result); ———⑱
    }
    SRF_RETURN_DONE(funcctx); ———⑲
}

#include "lib/stringinfo.h"

static void readfile(char *path, exttable_status *status, int natts, char delimiter) ———⑳
{
～省略～
}
```

処理を行います（リスト1-④）．この際に複数回の関数呼び出しでも使えるメモリ上のデータ領域が確保されます．このメモリ領域は関数コンテキストと呼ばれます．

●関数コンテキストへの切り替え

PostgreSQLのバックエンドで使用するメモリは，メモリマネージャによって管理されています．メモリマネージャはmalloc( )などのOSが用意するメモリ管理機構の上に構築されたサブシステムで，メモリをトランザクションや関数呼び出しなどの目的（メモリコンテキスト）別に管理することができ，コンテキストの終了時に一挙にメモリを解放することにより，メモリの解放忘れを防ぎます．

リスト1-⑤は，初期化処理で作成した関数コンテキストへメモリコンテキストを切り替えています．以後獲得されたメモリは関数コンテキストに含まれることになります．

●返却データ用メモリ領域の構築

ユーザに返却するための行を記述するメモリ上の構造体（タプルディスクリプタスロット）をCreateTupleDesc( )とTupleDescGetSlot( )を呼び出して構築します（リスト1-⑥，⑦）．タプルディスクリプタスロットは，本来データベースの中で実際に問い合わせを実行するエグゼキュータで使われる構造体で，通常はディスク上のデータベースを読み出して処理するのに使われます．ここでは，行を返す関数では，データベースから読み出す代わりにメモリ上に検索結果を模擬的に作り出す必要があり，そのためにエグゼキュータと同じタプルディスクリプタスロット構造体を使用しています．作成したタプルディスクリプタスロットは，

関数コンテキスト（funcctx->slot）の中にしまっておき，以後の関数呼び出しで再度使えるようにしておきます（リスト1-⑦）．

●作業領域の確保

前述のように，exttableはテキストファイルの行数分だけ何度も呼び出されますので，最初にメモリに読み込んだテキストファイルを格納したメモリへのポイントや処理行数のカウンタなどの情報をどこかにしまっておく必要があります．ファンクションマネージャを使う際には，一般に以下のような手順でそうしたデータを扱います．

❶ ユーザデータを格納する構造体を定義（リスト1-①）
❷ ❶の構造体を格納するメモリを確保（リスト1-⑧）．なお，メモリ獲得はmalloc( )ではなくpalloc( )という関数を使うこと，関数メモリコンテキストで行うことがポイントです
❸ ❷で獲得した領域へのポインタをファンクションマネージャが管理する関数メモリコンテキストの中に保存します（リスト1-⑨）

●テキストファイルの読み出し

exttableでは，関数の初回呼び出し時に検索対象のテキストファイルの中身を全部メモリに読み取っています[注1]．

テキストファイルの読み込みはreadfile( )という内部関数で行います（リスト1-⑪）が，そのためにはまずテキストファイルのパス名や区切り文字の情報をexttable( )の引数から取得する必要があります．

●引数の取得

ファンクションマネージャから渡されるパラメータは，前述のようにFunctionCallInfoData構造体のargメンバに渡ってきます．したがって，fcinfo->arg[0]のようにすれば，引数に対する値（call by valueの場合）またはポインタ（call by referenceの場合）が取得できるはずです．さらにPostgreSQLの場合，文字列データ型や配列などではデータの自動圧縮が行われることもあり，それらをすべてユーザが判定するのはあまりにも煩雑であるため，引数を取得するための一連の便利なマクロが用意されています．これらのマクロは，引数として0から始まる引数の位置を取り，データ型に対応するデータまたはデータ型へのポインタを返します．どのようなマクロがあるのかということについての資料はソースコード（fmgr.h）しかないという状況なので，参考までに作成したマクロの一覧表（表

表1◎マクロの一覧

●マクロ名	●SQLでのデータ型	●Cでのデータ型	●call by value?
PG_GETARG_INT32	INTEGER	int	yes
PG_GETARG_INT16	SMALLINT	short	yes
PG_GETARG_BOOL	BOOLEAN	bool	yes
PG_GETARG_OID	OID	Oid	yes
PG_GETARG_CSTRING	CSTRING	char *	no
PG_GETARG_NAME	NAME	name *	no
PG_GETARG_FLOAT4	REAL	float4	yes◆
PG_GETARG_FLOAT8	DOUBLE PRECISION	float8	yes◆
DatumGetByteaP	BYTEA	bytea *	no
DatumGetTextP	TEXT	text *	no
DatumGetBpCharP	CHARACTER	BpChar *	no
DatumGetVarCharP	VARCHAR	VarChar *	no

◆REALやDOUBLE PRECISIONは本来pass by reference扱いですが，これらのマクロを使うとpass by valueとみなすことができます．

注1）そのため，本稿で解説しているバージョンのexttableでは扱えるテキストファイルの大きさは最大1000行という制限がありましたが，現在公開している最新バージョンでは1行読んでは1行返すように改良されています．

1）を示します．

これ以外のデータ型やユーザ定義データ型については専用のマクロがありませんが，マクロを該当するデータ型でキャストすることによって引数の取得が可能です（表2）．

なお，同じデータ型に対してPG_GETARG_FLOAT8とPG_GETARG_INT64の両方が書いてあるものがあります．デフォルトではPG_GETARG_FLOAT8を，--enable-integer-datetimesを指定してPostgreSQLをインストールした場合はPG_GETARG_INT64を使用してください．

●第1引数の読み込み

exttableの第1引数はテキストファイルへのパス名です．これを読み込む処理がリスト1-⑩です．ちょっとごちゃごちゃした感じですが，順番に見ていきましょう．まず，PG_GETARG_TEXT_P(0)で，第1引数からtext型のデータとしてパス名を取り出します．text型，BpChar型，VarChar型は実はすべて同じvarlenaという構造体の別名です．

```
struct varlena
{
    int32   vl_len;
    char    vl_dat[1];
};
```

vl_lenはデータのバイト長で，vl_datに実際のデータが格納されています．このままではCプログラムでは扱いにくいので，PostgreSQLの内部関数であるtextout()を使ってCの文字列型に変換します．textout()はDatum型を受け取り，C文字列データへのポインタをDatumとして返します．textout()はPostgreSQLがすべてのデータ型のために備えている内部表現から外部表現への変換関数の1つですが，ユーザ定義C関数ではこのようなPostgreSQL内部の関数も自由に利用できるところが魅力です[注2]．

textout()のような内部関数もファンクションマネージャによって管理されており，動的にロードする必要がないこと以外はほとんどユーザ定義関数と同じです．つまり，textout()はもともとはPostgreSQLの外部からしか呼び出さない関数なのです．これをPostgreSQL内部から呼び出せるようにしたのがDirectFunctionCall1()です．DirectFunctionCall1()は最初の引数が関数，2番目の引数がその関数の引数です[注3]．

textout()の戻値はDatumなので，これをCの文字列に変換するためのマクロDatumGetCStringを最後に使って，ようやくテキストファイルのパス名を取り

表2◎マクロの一覧（2）

●マクロ名	●SQLでのデータ型	●Cでのデータ型	●call by value?
PG_GETARG_INT32	DATE	DateAdt	yes
PG_GETARG_FLOAT8	TIME	TimeAdt	yes
PG_GETARG_INT64	TIME	TimeAdt	yes
PG_GETARG_POINTER	TIME WITH TIME ZONE	TimeTzAdt *	no
PG_GETARG_FLOAT8	TIMESTAMP	Timestamp	yes
PG_GETARG_INT64	TIMESTAMP	Timestamp	yes
PG_GETARG_FLOAT8	TIMESTAMP WITH TIME ZONE	TimestampTz	yes
PG_GETARG_INT64	TIMESTAMP WITH TIME ZONE	TimestampTz	yes
PG_GETARG_POINTER	INTERVAL	Interval *	no
PG_GETARG_POINTER	TID	ItemPointer *	no

注2）ただし，PostgreSQLの内部関数に関するドキュメントはまったくと言ってよいほど整備されておらず，ソースが読めないとどうしようもないのが辛いところです．チャンスがあれば，この連載でPostgreSQLの内部関数についてもっとご紹介できると嬉しいのですが……．

注3）そのほか，2引数用のDirectFunctionCall2()から19引数用のDirectFunctionCall19()まで用意されています．

出すことができました．

●第2引数の読み込み

第2引数はCHARACTER型の区切り文字です．PostgreSQLでは，CHARACTERはCHARACTER(1)と同じなので，PG_GETARG_BPCHAR_Pマクロを使って引数を取得します．その結果はBpChar型，すなわちvarlena型です．varlena型はvl_datにデータが入っているのでそれをそのまま取り出せそうな気がしますが，前述のように文字列型のデータは圧縮されていることがあるので，直接vl_datにアクセスするのではなく，VARDATAというマクロを使うようにします．

readfile( )

readfile（リスト1-⑳）は指定されたテキストファイルを1,000行以内で読み込みます．palloc( )を使う以外は中身はごく普通のCプログラムです．強いて言えば，PostgreSQLの内部関数のmakeStringInfo( )とappendStringInfoChar( )を使っていることが挙げられます．これらはあらかじめ予測できない長さの文字列を扱うための関数で，こういう便利なものが使えるのもC関数の良いところです．

返却する行データの作成（リスト1-⑫）

heap_formtuple( )とTupleGetDatum( )というPostgreSQL内部関数を使ってメモリ上に行データを作成するのが主たる処理です．

まず，各列のデータ型を求めるために，TupleDesc型のForm_pg_attribute型のメンバであるattrsにデータ型に格納されているOIDを取得します（リスト1-⑬）．

PostgreSQLでは型に関する情報はすべてシステムカタログで管理されているので，型のOIDをキーにしてシステムカタログを検索しなければなりません．こう書くと大変そうな気がしますが，SearchSysCache( )というPostgreSQLの内部関数を使えば簡単かつ高速に処理できます．SearchSysCache( )の第1引数は「キャッシュID」（検索のタイプ），第2引数から第5引数までは検索キーです（キーの数は第1引数によって決まっています）[注4]．今回は検索キーは型のOIDだけですが，キーの型はDatumに変換する必要があるので，ObjectIdGetDatum( )というマクロを使っています．

SearchSysCache( )の戻り値はメモリ上に読み込まれたシステムカタログ（この場合，pg_type）の該当行です．PostgreSQLでは，各システムカタログに対応した構造体があらかじめ定義してあり，システムカタログに何かするときにはGETSTRUCTマクロを使ってこの構造体に変換するのが作法です（リスト1-⑭）[注5]．

後は「外部表現」にテキストファイルのデータを書き換えてあげれば完成です．ここでtextout( )のときのことを思い出してください．PostgreSQLではすべての型に外部表現に変換する関数が用意されているのでしたね．pg_typeにはそのような関数のOIDが登録されていますので，それを呼び出せば簡単に処理できます（リスト1-⑮）．OidFunctionCall1( )は前に出てきたDirectFunctionCall1( )と同じような関数で，ただ関数名の代わりに関数のOIDを渡すところだけが異なります．

SearchSysCache( )が返却したメモリを使い終わったら，必ずReleaseSysCache( )でメモリを解放してください（リスト1-⑯）．

以上で1行のデータをすべて作り終わったので，heap_formtuple( )とTupleGetDatum( )を呼び出してデータを行として作成（リスト1-⑰），関数の戻り値として返却します（リスト1-⑱）．

ここで，SRF_RETURN_NEXTは返却する行がまだあるときに呼び出すマクロで，すべての行を返却し終わったらSRF_RETURN_DONEを呼び出さなければなりません（リスト1-⑲）．

dblink

2回に渡ってユーザ定義C関数の書き方を説明いたしました．ユーザ定義関数の解説の締めくくりとして，擬似的に分散データベース機能を実現するdblink

注4）この他にどのようなキャッシュIDがあるかについては，シーラカンス本（技術評論社刊『PostgreSQL完全攻略ガイド』）のサポートページhttp://www2b.biglobe.ne.jp/~caco/third_edition/index.html の「カタログキャッシュについて」をご覧ください．

注5）構造体の名前は「Form_システムカタログ名」になります．

というC関数をご紹介します．

dblinkとは

PostgreSQLでは，ユーザは同時には1個のデータベースしか使うことができません．たとえば東京と大阪にデータベースが別々に設置されているときに，それぞれのデータベースを切り替えて使うことはできますが，東京と大阪のデータベースの中にあるテーブルをJOINするなどということはできません[注6]．

このような機能は一般に分散データベースと呼ばれており，まだPostgreSQLでは実装されていない機能の一つです．これをユーザ定義C関数を使って（かなり強引な方法で）実現してしまったのがdblinkです．もちろん本物の分散データベースではありませんから，複数のデータベースの間の更新のトランザクション一貫性を保つとか，ネットワーク遅延を考慮して最適な問い合わせプランを作成するといった機能はありません．

dblinkのインストール

dblinkをインストールするためには，PostgreSQLのソースツリーが必要です．以下，すでにPostgreSQL 7.4.1がインストール済であり，かつインストールに使ったソース/usr/local/src/postgresql-7.4.1が/usr/local/src以下に展開済であるものとします．

▼図1　テストデータの登録

```
CREATE TABLE t1(i SERIAL PRIMARY KEY, j INTEGER,
dbname TEXT DEFAULT current_database());
psql:dblink.sql:2: NOTICE:  CREATE TABLE will
create implicit sequence "t1_i_seq" for "serial"
column "t1.i"
psql:dblink.sql:2: NOTICE:  CREATE TABLE /
PRIMARY KEY will create implicit index "t1_pkey"
for table "t1"
CREATE TABLE
INSERT INTO t1(j) VALUES(10);
INSERT 119409 1
INSERT INTO t1(j) VALUES(11);
INSERT 119410 1
INSERT INTO t1(j) VALUES(12);
INSERT 119411 1
SELECT * FROM t1;
 i | j  | dbname
---+----+--------
 1 | 10 | test1
 2 | 11 | test1
 3 | 12 | test1
(3 rows)
```

以下，postgresユーザで実行します．

```
$ cd /usr/local/src/postgresql-7.4.1/contrib/dblink
$ make
$ make install
```

次にdblinkを使用したいデータベースを作ります．ここではtest1とtest2とします．

```
$ createdb -E EUC_JP test1
$ createdb -E EUC_JP test2
```

最後にdblinkのユーザ定義関数群を登録します．

```
$ psql -f dblink.sql test1
$ psql -f dblink.sql test2
```

準備

テストデータとして，あらかじめ図1のものを登録しておきます．データベースの名前がわかるように，dbnameという列を作ってあります．

同じスクリプトをtest2にも流しておきます．以下のような状態になります．

```
SELECT * FROM t1;
 i | j  | dbname
---+----+--------
 1 | 10 | test2
 2 | 11 | test2
 3 | 12 | test2
(3 rows)
```

dblinkの簡単な使い方

さっそくdblinkを使ってtest1データベースに接続したままtest2のt1を検索してみましょう（図2）．

この形のdblinkでは，第1引数で接続するデータベースなどを指定する**接続文字列**（connection string），第2引数にSELECT文を渡します．接続文字列はlibpqやPHPなどでPostgreSQLにアクセスするプログラム

注6）同じデータベースクラスタの中のデータベース同士についても同じことが言えます．

▼図2　test2のt1を検索

```
test1=> SELECT * FROM dblink('dbname=test2',
'SELECT * FROM t1') AS t1(i INTEGER, j
INTEGER, dbname TEXT);
 i | j  | dbname
---+----+--------
 1 | 10 | test2
 2 | 11 | test2
 3 | 12 | test2
(3 rows)
```

を書いたことのある方ならお馴染の，データベースへの接続条件を指定する文字列です．今回の例ではデータベース名だけを指定していますが，ホスト名も指定するとしたら，以下のようになります．

```
hostname=anotherhost dbname=test2
```

これ以外にも接続文字列には表3のようなパラメータが使えます．

dblinkの戻り値は特定のデータ型を持たないレコード型ですから，このようにAS句を使って結果のデータ型を指定します．ビューを使えば図3のように，もう少し普通のSELECT文らしく見せることもできます．もちろんtest1とtest2のテーブルを結合することもできます（図4）．

データを更新するには，dblink_execを使います（図5）．第2引数のSQL文にはもちろんUPDATE文以外にINSERTやDELETE文も使えます．

持続的接続

dblinkはどうやって他のデータベースに接続しているのでしょう？実はdblinkはPostgreSQLのクライアントインタフェースであるlibpqを使っているのです．dblinkはサーバ側で動く関数ですからサーバアプリケーションですが，その一方でlibpqを使うクライアントアプリケーションでもあるのです．

したがって，さきほどの例では問い合わせを実行する都度test2データベースに接続していたわけです．データベースへの接続オーバヘッドを避けたい場合は，持続的接続（persistent connection）を使用することができます．

dblink_conectは持続的接続を開きます．

```
test1=> SELECT dblink_connect('dbname=test2');
 dblink_connect
----------------
 OK
(1 row)
```

持続的接続を利用して検索を行うには，図6のように引数が1個のdblinkを使用します（更新は引数が1個のdblink_execを使います）．

持続的接続はdblink_disconnectを呼び出すか，バ

▼図3　ビューの使用

```
test1=> CREATE VIEW test2_t1 AS SELECT * FROM
dblink('dbname=test2', 'SELECT * FROM t1') AS
t1(i INTEGER, j INTEGER, dbname TEXT);
CREATE VIEW
test1=> SELECT * FROM test2_t1;
 i | j  | dbname
---+----+--------
 1 | 10 | test2
 2 | 11 | test2
 3 | 12 | test2
(3 rows)
```

表3◎接続文字列のパラメータ

●パラメータ	●意味	●例
hostaddr	接続IPアドレス	127.0.0.1
port	ポート番号	5433
user	ユーザ名	foo
password	パスワード	mypassword

▼図4　テーブルの結合

```
test1=> SELECT * FROM t1 JOIN test2_t1
USING(i);
 i | j  | dbname | j  | dbname
---+----+--------+----+--------
 1 | 10 | test1  | 10 | test2
 2 | 11 | test1  | 11 | test2
 3 | 12 | test1  | 12 | test2
(3 rows)
```

▼図5　データの更新

```
test1=> SELECT dblink_exec('dbname=test2',
'UPDATE t1 SET j = j + 1 WHERE i = 1');
 dblink_exec
-------------
 UPDATE 1
(1 row)

test1=> SELECT * FROM test2_t1;
 i | j  | dbname
---+----+--------
 2 | 11 | test2
 3 | 12 | test2
 1 | 11 | test2
(3 rows)
```

▼図6　持続的接続

```
test1=> SELECT * FROM dblink('SELECT * FROM
t1') AS t1(i INTEGER, j INTEGER, dbname TEXT);
 i | j  | dbname
---+----+--------
 1 | 10 | test2
 2 | 11 | test2
 3 | 12 | test2
(3 rows)
```

ックエンドが終了するまで持続します．

```
test1=> SELECT dblink_disconnect();
 dblink_disconnect
-------------------
 OK
(1 row)
```

名前付持続的接続

dblinkの第1引数に接続の名前を指定することにより，複数の持続的接続を使い分けることもできます(図7)．

名前付持続的接続もまたdblink_disconnectを呼び出すか，バックエンドが終了するまで持続します．

```
test1=> SELECT dblink_disconnect('mycon1');
 dblink_disconnect
-------------------
 OK
(1 row)
```

もちろんdblink_execと持続的接続を使って更新を行うこともできます．この場合，dblink_execの第1引数に持続的接続の名前を使うだけです．

dblinkの使い方その他

dblinkにはこのほかカーソルを扱うための関数，主キーを獲得する関数などが用意されています．誌面の都合でここでは紹介できませんが，詳細はdblinkに付属するドキュメントをご覧ください．

README.dblink	：	dblinkの概要に関する解説
doc/connection	：	DB接続に関する関数の解説
doc/cursor	：	カーソルに関する関数の解説
doc/query	：	SELECTに関する関数の解説
doc/execute	：	更新SQLに関する関数の解説
doc/misc	：	その他の関数の解説L
doc/deprecated	：	もう使われなくなった関数

さいごに

C言語によるユーザ定義関数は，PostgreSQLの機能を最大限に発揮させる究極のPostgreSQLアプリケーションとも言えます．3回に渡る連載によって，そのテクニックを少しでも身につけていただけたなら幸いです．WD

▼図7　複数の持続的接続

```
test1=> SELECT dblink_connect('mycon1', 'dbname=test1');
 dblink_connect
----------------
 OK
(1 row)

test1=> SELECT dblink_connect('mycon2', 'dbname=test2');
 dblink_connect
----------------
 OK
(1 row)

test1=> SELECT * FROM dblink('mycon1', 'SELECT * FROM t1') AS t1(i INTEGER, j INTEGER, dbname TEXT)
JOIN dblink('mycon2', 'SELECT * FROM t1') AS t2(i INTEGER, j INTEGER, dbname TEXT) USING(i);
 i | j  | dbname | j  | dbname
---+----+--------+----+--------
 1 | 10 | test1  | 10 | test2
 2 | 11 | test1  | 11 | test2
 3 | 12 | test1  | 12 | test2
(3 rows)
```