Windows PowerShell 言語による

PostgreSQL Access Class の開発

・PowerShellからのPostgreSQLの利用を容易にする

・プログラミングで，SQL文のデバッグを容易にする

・コマンドシェル，スクリプトプログラム，その他
多彩な利用方法に対応できる

福井精華学園 啓新高等学校
日本PostgreSQLユーザ会 会員
阿　部　哲　也

● 開 発 の 経 緯

・勤務するのは福井県福井市にある私立の高等学校

1）2004年に校内LANの工事を行った。
（このとき作ったファイル受け渡しの仕組み＝ポストがクラスの利用例の一つである。）

2）教務システムの構築を担うこととなった。当初，Excelベースだった（DBMSは使っていなかった）。

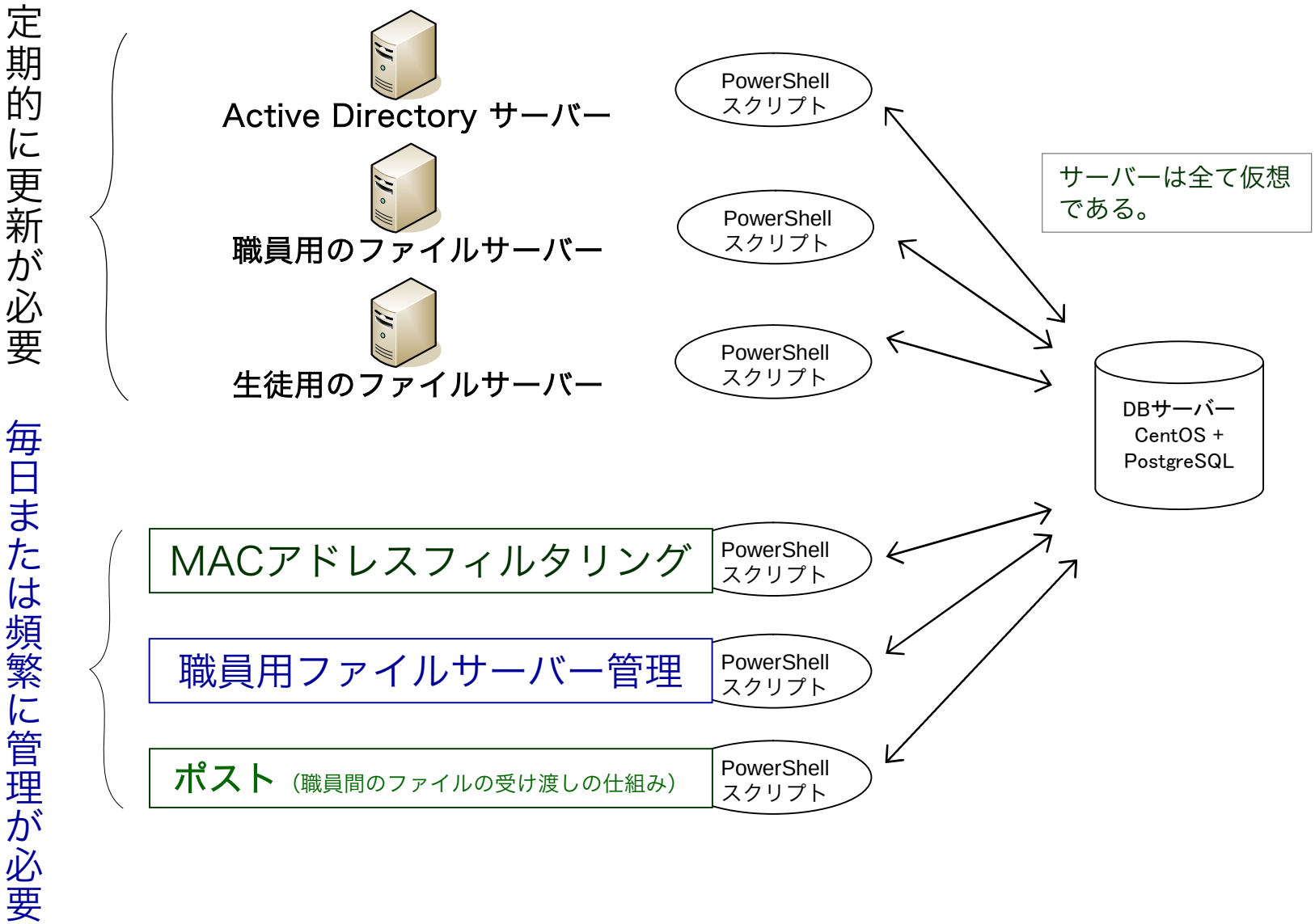
3）2006年Excelベースの機能拡張に限界がきて，DBMSの導入が必要となり，６月 PostgreSQL 8.1 Windows版 を導入する。

4）導入時からクライアント側はExcelで，ADO（ActiveX Data Objects）をいちいち操作するプログラムを書いていた。

5）プログラム作成の負担を減らすため，2007年度終わり頃（2008年初め）より，PostgreSQLアクセス用のクラスをVBAで作成し利用するようになる。

6）2011年LANサーバーのリプレースと仮想化を機に，サーバー管理用途にPowerShellを使うようになり，PostgreSQLアクセス用のクラスの開発を企画する。

●PostgreSQLアクセスクラス開発の目的



●なぜPowerShellか？ コマンドプロンプトやWSHは？

・Microsoft社はシステム管理，サーバー管理に特化したシェル／スクリプト言語としてPowerShellを提供したので，システム・サーバー管理に必要な機能，モジュール，ライブラリが充実している。

・以前から提供しているコマンド・プロンプト（コマンドシェル）やWSH（スクリプト言語）に今後PowerShellが取って代わる予定である。

例：http://technet.microsoft.com/ja-jp/library/cc731774.aspx

・PowerShellは，コマンドシェル（非定型の処理）と，プログラム作成用のスクリプト言語（定型処理）の２つの側面を持つので，管理用途には使い易い。

・PowerShellは，.NET Frameworkの広範なライブラリを直接利用できる他，Comオブジェクトや WMIオブジェクトの利用も容易である。GUIのプログラム開発も可能で（ .NET Framework），以前からあるスクリプトの移植も難しくない（Com，WMI）。

・データベースとの連携に適したコマンドを備えている。

例：・・・・・・・・・・・・・・・・・・

```
$htbl = $pg.read();   ← データベースからの読み込み，ハッシュテーブル $htbl へ
$recs = $htbl | where { $_.姓  -eq  “阿部” } | sort  id;   ← レコードの選択とソート
```

（ハッシュテーブル $recs には列名”姓”が阿部で，列名idでソートしたレコードが入る）

●プログラム開発におけるPowerShellの難点

1）言語のスコープルールがダイナミックスコープ（動的スコープ）である。

・子（呼び出された側）ブロックでは親（呼出側）ブロックで参照可能な全ての変数を参照できる。

・子（呼び出された側）ブロックでの変数への代入が変数宣言となり、そのブロックのローカル変数となる。

・$ErrorActionPreference のようなシェル変数もダイナミックスコープである。

2）関数やスクリプトブロック（メソッド等）からの複数の戻り値を許し，その場合，動的に戻り値が配列となる。

例：$adapter.Fill($dtbl); ← SELECT文による読み込み。戻り値として行数を返す。

戻り値としないためには
$n = $adapter.Fill($dtbl);

読み込んだレコードは $dtbl 変数に入るので，このようについ書いてしまうが，関数等のブロックの中で使われた場合、このままでは行数はブロック(関数)の戻り値となる。

3）デフォルト設定では，$ErrorActionPreference の値は Continue であり，多くのエラーでプログラムが停止しない。

例：ADO.NETを使ったデータベースの操作では，エラーで停止しない。

バグが取れたライブラリが揃えば PowerShell の良さが生きてくる！

● PostgreSQL Access Class の重点仕様

1）接続のOpenやClose，SQL文の実行などの基本的な操作をできるだけ一つのメソッドにまとめ，ADO.NETを意識せずに簡単なコマンド（メソッド）でPostgreSQLとのやり取りをできるようにする。

⬇

誤解を恐れずに言えば，PowerShellからPostgreSQLに**安直**につなぐことを目指す

```
例：$pg = new  pgclass;                    ← 変数 $pg にPG Access Object を代入し初期設定を実行
    $pg.open();                            ← 初期設定ファイルの定義にしたがって、PostgreSQLに接続
    $pg.sql = “SELECT  …  FROM  …  WHERE  …  ; ”;  ← SQL文の設定
    $dtbl = $pg.read();                    ← ハッシュテーブル $dtbl にPostgreSQLからレコードを取得
    $pg.close();                           ← PostgreSQLとの接続を閉じる
```

2）PowerShellのベースであり，移植性が高い，.NET Framework（ADO.NET）を利用して接続する。

3）SELECT文で取得するレコードは，キーが列名のハッシュテーブルで出力する（PowerShellで扱いやすいデータ形式で出力する）。

4）コマンドシェル上での対話的な使用にも，プログラムでの利用にも違和感なく使えるように両方に必要な機能を備えるほか，クラス拡張機能を設け，多彩な利用方法に対応できるようにする。

拡張機能例： $dtbl = $pg.GetStdInf(2, 3); ← ２年３組の生徒の情報を取得する
ユーザー定義メソッド

5）プログラムでの利用では，SQL文のエラーで実行を停止する。 その際，SQL文のデバッグを助ける機能を設ける。

●PostgreSQL Access Class 実装上の問題点と工夫

★PowerShellにはクラス定義の機能がない

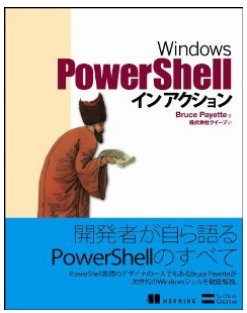
開発者が教える PowerShell の CustomClass

PowerShellの開発リーダーBruce Payetteが，著書「PowerShell インアクション」の８章で紹介している方法

(1) メソッドやプロパティーを持たない空のPSObjectを作成する。

```
$object = New-Object  Management.Automation.PSObject;
```

(2) 作成したオブジェクトにメンバ（メソッド，プロパティー，他）を追加する。



開発の主要ポイント

クラス定義の機能を拡張する改良

1）クラスのプログラムコードを，コンポーネントとして分離することができる。

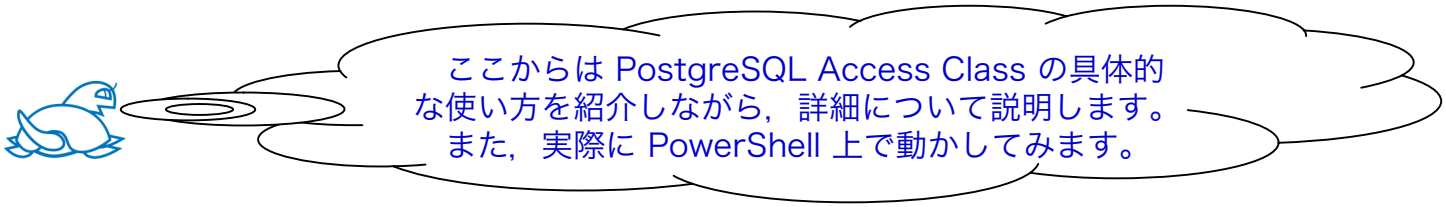
・クラスのメンバ定義の中にプログラム（スクリプト）を書かなくてもよいように，クラス定義とプログラムを分離することで，大きなサイズのクラス定義の問題点を克服する。

2）複数のクラス定義を一つのクラスとしてまとめることができる。

・別に書いたクラス定義を，拡張定義やユーザー定義としてクラス本体に結合する → 動的な機能拡張

3）Constructorを定義できる。

・初期設定ファイルによる，データプロバイダー（ODBCかNpgsql），データベース，ユーザー等の動的な設定，および，拡張メンバ，ユーザー定義メンバによる動的な機能拡張を可能にする。



★PostgreSQL Access Class の詳細説明を「コマンドシェル上で対話的に利用する場合」「プログラムで利用する場合」「両方の利用方法に共通する内容」の３つの視点に分類して説明を進めます。説明の内容および順序は以下の通りです。

１）両方の利用方法に共通する内容

・コマンド一つ，簡単に行える利用設定。
・利用上の問題点となる場合がある，接続パスワードのデータ形式。
・コマンド入力の省力化 = エイリアスの設定 において，ユーザーのカスタマイズを容易にする。
・メソッドの実行後，２つの利用方法に適した反応（メッセージと戻り値）を返す。
・多彩なADO.NETの機能の中で，このクラスに適したものだけを実装する。
・実行時エラーが起きた場合に，データベース接続が残る問題の対処。
・PowerShell が持っている機能と連携し，使いやすさを目指す。

２）コマンドシェル上で対話的に利用する場合

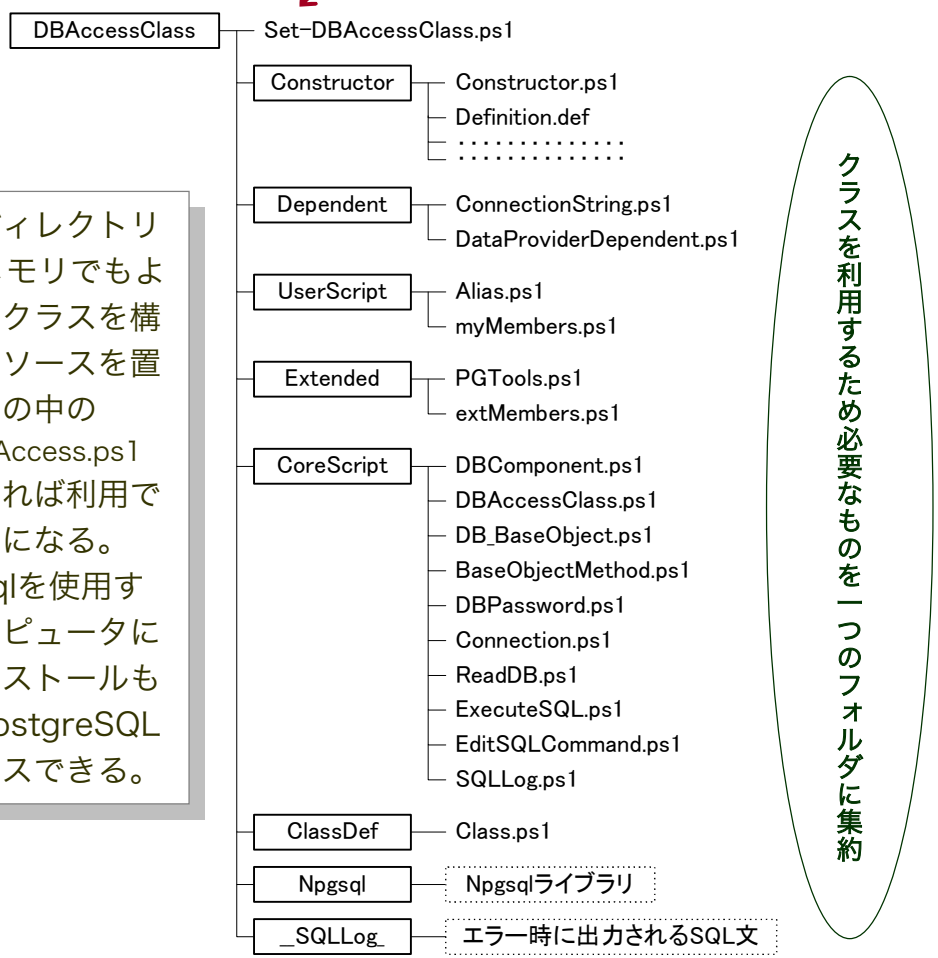
・コマンドシェル上で対話的に利用するのに役立つ便利な機能を提供する。
・仕事に応じて使いたいメソッドやプロパティーを，ユーザー拡張機能として定義できる。

３）プログラムで利用する場合

・動的なSQL文の作成を容易にする，行単位のSQL文の作成機能。
・呼び出し側のプログラムが，デバッグ中，クラスの使用設定を繰り返すことに対処する。
・SQL文の実行エラーで，呼び出し側の実行を強制的に止める機能を設ける。
・SQL文のエラーで実行が止まった場合、エラー以前に実行した10個のSQL文を出力してデバッグを助ける。

●コマンド一つ，簡単に行える利用設定

このスクリプトを実行すれば，利用設定は終了！



クラスを利用するため必要なものを一つのフォルダに集約

適当なディレクトリ（USBメモリでもよい）に，クラスを構成するリソースを置いて，その中の Set-DBAccess.ps1 を実行すれば利用できるようになる。
　Npgsqlを使用すればコンピュータに何のインストールもなしにPostgreSQLにアクセスできる。

デフォルトの初期設定ファイル

初期設定ファイル Definition.def の内容

```
コンストラクターの初期設定のための定義
Definition for Constructor
   DBMS            : PostgreSQL
   DataProvider    : ODBC
   odbcDriver      : {PostgreSQL Unicode}
   DBServer        : 192.168.203.240
   Port            : 5432
   Database        : test_powershell
   DBUser          : adminUser
   DBPassword      : adminPassword
   PasswordFormat  : PlaneText
   SQLErrorAction  : Stop
   Extended        : PGTools.ps1
   UserDef         : Alias.ps1
   UserDef         : myMembers.ps1
End Definition
```

$pg = new pgclass;
で変数 $pg にPostgreSQL Access オブジェクトが生成される。コンストラクタは上記デフォルトの初期設定ファイルを読み込んで設定を行う。
　new の第２引数に初期設定ファイル名を指定すれば，それを使う。

○USBメモリにリソースを置く場合、ルートディレクトリに DB.ps1を置いて，コンソールから以下の通り実行。

PS C:¥Documents and Settings¥t-abe> F:¥DB

DB.ps1 の内容

```
$tPath = Split-Path  $myInvocation.MyCommand.path -Parent;
&( $tPath + "¥DBAccessClass¥Set-DBAccessClass.ps1" );
```

●利用上の問題点となる場合がある接続パスワードのデータ形式

疑問：DB接続のパスワードを平文で初期設定ファイルに書いておくのですか？

回答：PostgreSQL Access Class は３つのパスワード形式を持っています。

１）PlaneText　　平文パスワード

２）KeyboadInput　　コマンドシェル上で対話的な使用の場合

３）Encryption　　パスワードの暗号化。復号時に鍵を入力する方法は？

PowerShellインアクション　　Bruce Payette 著

13-5　安全なスクリプトの記述　　P.528 ～ P.531

・・・・・。PowerShellは，.NETを通じて，機密データを安全に処理するための機能を提供します。・・・・・。

・・・・・。.NETプログラムを作成する際には，機密データが含まれた文字列を安全に操作するために，System.Security.SecureStringクラスを使用します。このクラスは，.NETランタイムがテキストデータを暗号化した状態でメモリ上に格納するためのコンテナです。・・・・・。

・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・。

・・・・・。PowerShellには安全な文字列を操作するためのコマンドレットとして，ConvertTo-SecureStringとConvertFrom-SecureStringの２つが用意されています。・・・・・。

・・・・・。・・・暗号鍵は，Windowsログオン認証に基づきます。つまり，データの暗号化や復号のための鍵を指定する必要はありません。これはログオン認証に基づいて自動的に生成されます。・・・・・

注　ConvertFrom-SecureString … SecureStringから暗号化文字列に変換
　　ConvertTo-SecureString　　… 暗号化文字列からSecureStringに変換

☆パスワード形式がEncryptionの場合

ａ）パスワードを左記説明の暗号化文字列に変換して初期設定ファイルに保存する。

ｂ）左記説明の通り，暗号化文字列の復号に鍵を必要としないが，ログオンユーザーやログオンコンピュータが変わると復号できない。

↓

タスクスケジューラで自動実行する場合でも利用できる。さらに，他のユーザーが復号するのを防ぐことができる。

ログオン情報が違うと、このコマンドの実行でエラーとなる。

●コマンド入力の省力化 = エイリアスの設定 において，ユーザーのカスタマイズを容易にする

○PostgreSQL Access Class で設定されているエイリアス

(1)プロパティー

SQLCommand	→	sql
TableName	→	tbl
AddSQL	→	add
‥‥‥‥‥	→	‥‥‥

(2)メソッド

OpenConnection	→	open
ReadDB	→	read, load, rd, ld
_Execute_SQL_Statement_	→	ExecuteSQL, exe, ex
‥‥‥‥‥	→	‥‥‥

エイリアス名を書き替えれば，ユーザー設定エイリアスを定義できる →

○UserScript フォルダにある MergeClass myAlias

```
MergeClass myAlias {
 # Alias of SQLCommand Property
   Alias     sql          SQLCommand;
 # Alias of TableName Property
   Alias     tbl          TableName;
   ‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥
 # Ailias of Open Connection Method
   Method   open   { param  ( [string]$conStr );
                     return  `
                     ( $this.OpenConnection( $conStr ) ); };
    ‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥‥
}; # end of myAlias
```

●メソッドの実行後，２つの利用方法に適した反応（メッセージと戻り値）を返す

☆コマンドシェルコンソールで OpenConnection メソッドを実行すると

```
PS C:¥> $pg.open()
Npgsql で接続しました。   ←  コンソールへの表示（コマンドシェル上での実行で利用）
True                      ←  メソッドの戻り値（プログラムで利用）
PS C:¥>
```

メソッドの実行では，コマンドシェル上で対話的な利用の場合もプログラムからの利用の場合も，処理の成功・不成功等の実行結果を得たい。その際，コマンドシェル上での利用ではメッセージで返る方がよいが，プログラムでは戻り値として返る方がよい。このオブジェクトはその両方を返す。

PowerShellでは戻り値も画面表示されるため，クラスのプログラミング時に注意が必要であった。また，実行結果によって戻り値の個数が変わらないように細心の注意を払わなければ，バグの原因となる。

●多彩なADO.NETの機能の中で，このクラスに適したものだけを実装する

ー 余計なプロパティーやメソッドを持たずにできるだけ単純化する ー

☆例えばSELECT文でデータベースから読み込む場合の，DataAdapterオブジェクトとDataReaderオブジェクトの違い

○DataAdapterオブジェクトを使った読み込み ← PG Access Class

```
$conStr = "Server=…; Database=…; ……";
$con  = New-Object Npgsql.NpgsqlConnection($conStr);
$con.Open();
$sqlStr = "SELECT * FROM …;";
$adpt = New-Object Npgsql.NpgsqlDataAdapter($sqlStr, $con);
$tbl  = New-Object System.Data.DataTable( “myTable" );
$RowNum = $adpt.Fill($tbl);  #← 読み込んだレコード数を返す
$con.Close();
# DataTableオブジェクト(ハッシュテーブル) $tbl にデータベース
# から読み込んだ全レコードが入る。
```

DataAdapterオブジェクトの出力はハッシュテーブルで，PowerShellと相性が良い。DataReaderオブジェクトによる読み込みは，このクラスの仕様では相性が悪い。

○DataReaderオブジェクトを使った読み込み ← 軽い

```
$conStr = "Server=…; Database=…; ……";
$con  = New-Object Npgsql.NpgsqlConnection($conStr);
$con.Open();
$sqlStr = "SELECT * FROM …;";
$cmd = New-Object Npgsql.NpgsqlCommand($sqlStr, $con);
$reader = $cmd.ExecuteReader();
while ( $reader.Read() ) {
   # $reader.getName(…)が列名を，$reader[…]がデータを返す
   # 読み込んだ 1 レコードを使った処理
};
$con.Close();
# データベースから 1 レコードずつ読み込む。途中で接続を切ると
# それ以上は読み込むことができない。
```

●実行時エラーが起きた場合に，データベース接続が残る問題の対処

a）プログラムで利用していて接続が開いている時，実行時エラーで処理が停止すると，接続は開いたままとなる。

b）オブジェクトを保持する変数が大域変数でなければ，接続を閉じることができなくなる。

・本オブジェクト実行時のエラーによる停止なら，接続を閉じてから処理を停止することができるが，停止の原因はそれだけではない。

・また，コマンドシェル上の対話的な処理では，そもそも接続を閉じたくない。

c）プログラムが停止した後，接続を保持するConnectionオブジェクトはガベージ（ゴミ）となる。

d）調べてみたところ，PowerShellを終了するか，ガベージコレクションが起こると，接続はその時点で閉じる。

★PostgreSQL Access オブジェクトを生成し，最初にデータベースと接続する時点をトリガとして，強制的にガベージコレクションを起こすことで，接続が残る問題に対処しています。

●PowerShell が持っている機能と連携し，使いやすさを目指す

－ PowerShellの機能をできるだけ生かすような仕様とする －

1）PowerShellのヒア文字列 → SQL文の入力

```
PS C:¥> $pg.sql = @"
>> SELECT
>>     cla.grade, cla.class, cla.num, std.sei, std.mei
>> FROM
>>     Student std
>> INNER JOIN
>>     cla_and_num cla
>> ON
.....................................................
>> ;
>> "@
>>
PS C:¥>
```

ヒア文字列

・PowerShellコンソールで，対話的な利用でのSQL文の入力には，PowerShellのヒア文字列が便利。

・左記のように入力できて，整形しながらSQL文を入力できるので，長いSQL文の入力が容易になる。

・コンソールには行単位の編集機能しかないが，エディタでSQL文を編集して，コピー＆ペーストで流し込めばＯＫ！

・SQL文の実行でエラーとなっても，エディタ上のSQLを訂正してもう一度コピー＆ペースト。もう一度実行！

2）DataTableオブジェクト（ハッシュテーブル）→ SELECT文の出力

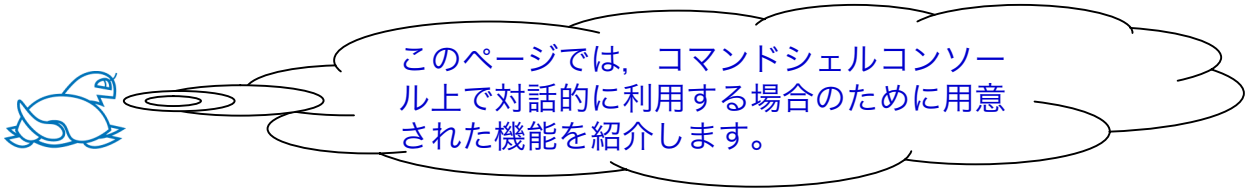
★PowerShellはデータベースとの連携に適したコマンドを持っている

a）$htbl = $pg.read(); ← データベースからの読み込み，ハッシュテーブル $htbl へ
$recs = $htbl | where { $_.姓 -eq “阿部” } | sort id; ← レコードの選択とソート
（ハッシュテーブル $recs には列名”姓”が阿部で，列名idでソートしたレコードが入る）

b）ConvertTo-CsvでCSVデータに，Export-CsvでCSVファイルに出力できる
$csv = $pg.read() | ConvertTo-Csv; ← 変数 $csv にCSVに変換したデータを代入
$pg.read() | Export-Csv D:¥…¥data.csv; ← data.csv ファイルを出力

c）出力をOut-GridViewコマンドに渡せば，GUIのグリッドビューに結果を表示できる。
$pg.read() | Out-GridView ← GUIのグリッドビューへの出力はPowerShellの威力を感じる

SQL文の簡略化や，問い合わせの回数を減らすことに貢献する



●コマンドシェル上で対話的に利用するのに役立つ便利な機能を提供する

・接続情報(データベース名，ユーザー名等）や，接続状況（接続がOpenかClosedか）をいつでも取得できる。

・テーブル名（含スキーマ名）や，テーブル定義情報（今のところ全てではない）を取得できる。

テーブル名の取得　　GetTableName メソッド（スキーマ名とテーブル名の組を取得）

テーブル定義情報の取得　　GetTableSchema メソッド（列名，データ型，プライマリーキー情報，その他を取得）

・GetSQLLogメソッドを設け，直前またはそれ以前（10個まで）に実行したSQL文を取得できる。

・GetSQLLogメソッドで取得したSQL文をクリップボードにコピーする Clipboard 関数がある（作成した）。
（エイリアス名 cb で，PostgreSQL Access Class とは独立した関数）

PS C:¥> $pg.gsl(3) | cb　　３つ前に実行したSQL文をGetSQLLogメソッドで出力し，クリップボードに送ってエディタで編集

●仕事に応じて使いたいメソッドやプロパティーを，ユーザー拡張機能として定義できる

・日頃の業務の中で必要となる，データベースを使うちょっとしたプログラムを，ユーザー定義メソッドとして定義し，クラスを拡張できる。

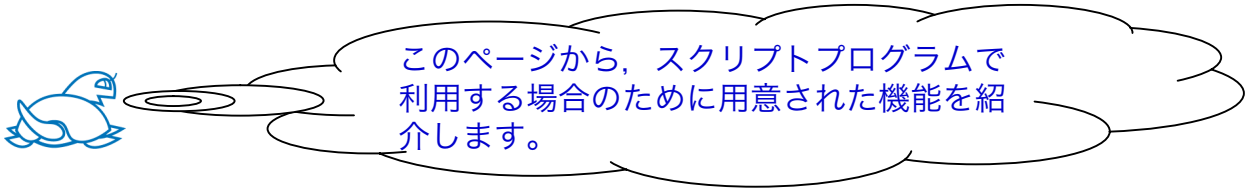
a）仕事の内容によって必要になるスクリプトを書いて，UserScriptフォルダーに保存する。

b）Constructor の初期設定に使う ….def ファイルの中で，UserScript : myMembers.ps1 のように，UserScript 項目として指定する。指定されたスクリプトファイルの定義によってクラスが拡張される。

（私の場合のユーザー定義メソッドの例）

$dtbl = $pg.GetStdInf(2, 3);　← 変数$dtblに，２年３組の生徒の情報を取得する
サンプルとしてこのメソッドの実装例を myMembers.ps1 に収めてあり，簡単な解説もある。

☆仕事によって使い分けができるように，複数のスクリプトファイル（ ….ps1ファイル）と複数の初期設定ファイルを用意する。初期設定ファイルを変えてオブジェクトを作成すれば，仕事の内容に応じた PostgreSQL Access オブジェクトを作成することができる。また，初期設定ファイルの中で UserScript 項目は複数指定できる。



●動的なSQL文の作成を容易にする，行単位のSQL文の作成機能

★プログラムでは値の指定に変数や関数を使ったり，条件分岐を使ってSQL文を作成したい場合があるが，そのような場合、１行ずつSQL文を追加していける機能が便利。

１）ScriptProperty　AddSQL

```
$pg.clr();     #← $pg.sql = “”; でもよい。
  $pg.add = “SELECT”;
  $pg.add = “    *”;
  $pg.add = “FROM”;
・・・・・・・・・・・・・・・・・・
  $pg.add = “WHERE”;
if ( … ) { #← 条件分岐を使ってSQLを作成できる
  $pg.add = “    column1 = “ + (functionA);
} else {
  $pg.add = “    column2 = “ + (functionB);
};
・・・・・・・・・・・・・・・・・・
```

２）ScriptProperty　AddInsertValue

```
$pg. ClrInsCmd();
$pg.tbl = “student”;          ※AddInsertColumn プロパティー
$pg.AddInsVal = 12098;          で対応する列名の指定も可能
$pg.AddInsVal = “’阿部’”;
・・・・・・・・・・・・・・・・・・
$pg.InsNewRec();
$pg.AddInsVal = 12100;
$pg.AddInsVal = “’田中’”;
・・・・・・・・・・・・・・・・・・
$pg.ins(); #← student テーブルに，(複数)レコード INSERT
```

※ここでも，if 文等の条件分岐が使える。また，オブジェクトが整形された INSERT 文を作成してくれる。

☆その他，UPDATE 文や DELETE 文にも２）のようなプロパティー，メソッドがある。また，テーブルの削除（DropTable）やトランザクション（BeginTransaction，CommitTransaction，RollBackTransaction）関係のSQL文はメソッド一つで実行できる。

●呼び出し側のプログラムが，デバッグ中，クラスの使用設定を繰り返すことに対処する

★インタプリタによる実行なので，プログラムのデバッグ中や作成中に実行を繰り返すことがある。このクラスはプログラムの中で利用設定することを前提としているので，繰り返すとメモリを圧迫するアセンブリの読み込みや，上書きが好ましくない大域変数の作成は，利用設定プログラムが一々チェックしている。

●SQL文の実行エラーで，呼び出し側の実行を強制的に止める機能を設ける

a）PowerShellの標準設定では，ADO.NET の接続オブジェクトやSQL文実行オブジェクトのエラーで，例外が発生しない（プログラムの実行が停止しない）。$ErrorActionPreference のデフォルトは Continue である。

b）PostgreSQL Access Class では，SQL文の実行エラーについては，オブジェクトの動作として，プログラムの実行を停止したかった。SQL文のバグがエラーの原因の多くを占め，その時点でデバッグが必要だから。

c）そこで，PostgreSQL Access Class ではSQL文の実行エラーが起こると，それをトラップして強制的に例外を発生（throw 文）させている。接続に関するメソッドでは，エラーが必ずしもバグとはいえないので，実行結果を返す仕様となっている。

d）テストの結果，PostgreSQL Access オブジェクト内で例外が起こっても，呼出側のプログラムでは例外として受け取れないことがわかった。そこで，以下のような処理を行っている。

ダイナミックスコープを利用したコード

```
# メソッドの入り口で，呼出側の $ErrorActionPreference 変数のポインタを取得
$pEAP = [ref]$ErrorActionPreference;
・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・
$ErrorActionPreference = “Stop”;  #← $ErrorActionPreference は ローカル変数に
・・・・・・・・・・・・ SQL文実行処理 ・・・・・・・・・・・・   エラー発生
# エラー処理ブロック（catch ブロック）で，呼出側の $ErrorActionPreference
# 変数の値を強制的に Stop に変更
$pEAP.Value = “Stop”;  #←呼出側の $ErrorActionPreference 変数への代入
☆これで，呼出側のプログラムはストップする。ただし，大域変数の
$ErrorActionPreference の値を書き替えてしまう可能性がある。
```

←呼出側のプログラムがSQL文を実行する前に$ErrorActionPreference の値を Stop にすればよいのですから，「ここまでする必要があるの？」とのご批判が出るのではないかと思います。私としてはPostgreSQLに安直につなげるという基本方針をつらぬきたかったのです。

他にも，SQLErrorAction初期設定値の説明で，PowerShellのエラー処理について解説できるメリットもあります。

e）初期設定に，SQLErrorAction：の項目を設けて，SQL文のエラーによる実行停止がないモードも持つ。

●SQL文のエラーで実行が止まった場合、エラー以前に実行した10個のSQL文を出力する
ー SQL文のデバッグを助ける機能 ー

1）SQL文の実行がエラーになると，コンソールに実行したSQL文を書き出してプログラムをストップする。

2）エラー以前の10個のSQL文について，大域変数 _SQLLog_ にSQLLogオブジェクトを書き出す。

3）SQLLogオブジェクトの内容（SQL文，実行オブジェクトと実行状況に関する情報）をファイルに出力。

● PostgreSQL Access Class の利用例

1）ポストメタファ

本校では以前からファイルサーバの共有フォルダにポストメタファを作成し利用しています。ポストメタファとは、ユーザー間のファイルのやり取りを、郵便を模して共有フォルダ上に実現した仕組みです。

1）クライアントがネットワークドライブを開くと，図１のように「_差出ポスト」「_受取ポスト」フォルダがあらわれる。
2）「_差出ポスト」フォルダを開くと、図２，図３のように職員室フォルダ、個人用ポストフォルダが現れ、目的の相手にファイルを渡すことができる仕組みである。
3）ファイルの受け取りは図１の「_受取ポスト」を使う。

注：差出ポストフォルダと受取ポストフォルダは同じ実フォルダをジャンクション（シンボリックリンク）を使って参照している。

図 1

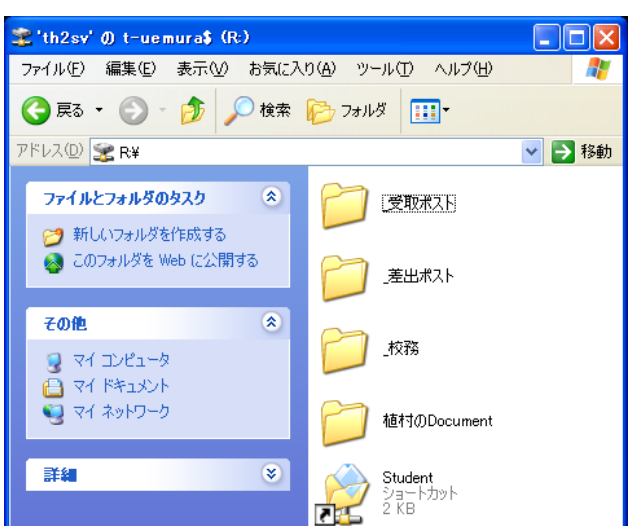
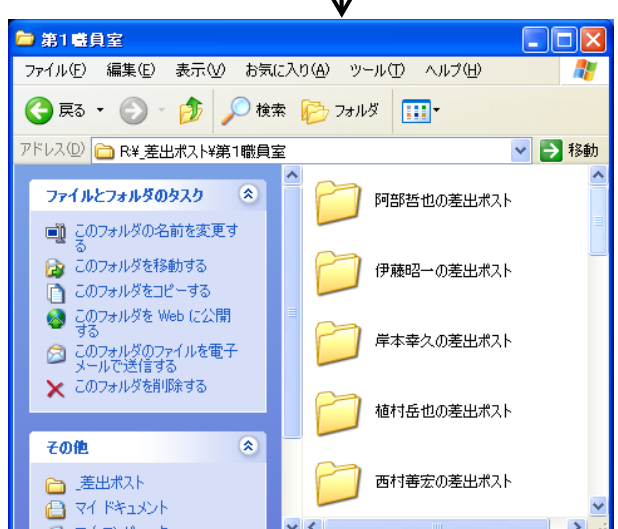


図 2

図 3



☆ポストメタファの工夫

PostgreSQL（データベース）とは関係ありませんが，ポストメタファで行ったちょっとした工夫を聞いて下さい。

a）ポストメタファといっても、当初は単なる共有フォルダを、参照（ジャンクション）を使ってユーザーを錯覚させるものであった。

b）しかし、利用者から「自分のポストの中のファイルを他人に見られるのは何とかしてもらえないか、これではポストと呼べないではないか」との指摘と，改善の要望が出た。

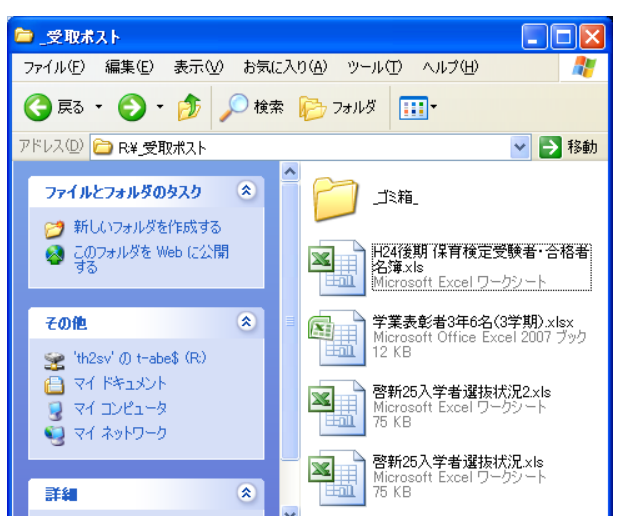
c）この要望は実現不可能と思っていたが，ある時アクセス権の設定をしていて「CREATOR OWNER」というユーザーが存在することに気づいた。これを使えば，だれもがポストにファイルやフォルダは作れるが、受取ユーザーと（ポストへの）書き込みユーザー以外は、ファイルの閲覧やコピーはできないという特殊なアクセス権の環境を作ることができる。

⇩

ポストメタファの完成です

★ PostgreSQL Access Class の利用

・PostgreSQL Access Classはポストメタファの運用管理に利用している。
・各ユーザのポストフォルダを開くと右図のようになる。ポストフォルダは「_ゴミ箱_」フォルダを備えていて、ポストの中で古くなったファイルやフォルダは「_ゴミ箱_」フォルダに強制的に移動させている。
・ファイル名やサイズ、更新日時をチェックして、ポスト内で古くなったアイテムを「_ゴミ箱_」フォルダに移動する仕様とした。
・「_ゴミ箱_」フォルダにはアイテム名の頭に移動日を付けて移動し、必要になった時に容易に探せるように工夫した。
・この機能を果たすポストフォルダ管理プログラムをPowerShell＋PostgreSQLで書き、タスクスケジューラで１日１回動かしている。

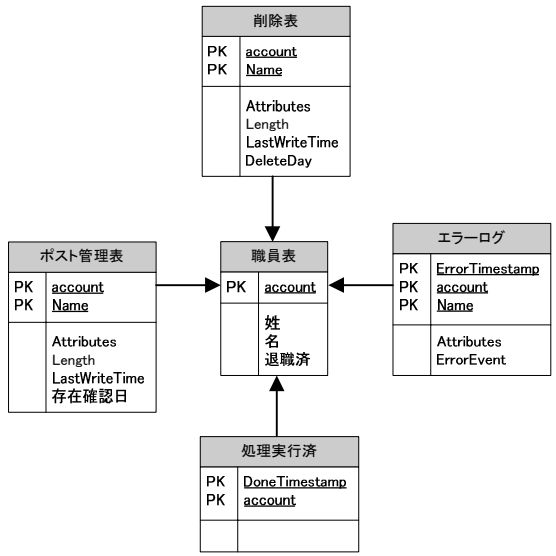


このような機能を付けたのには理由があります。せっかく作った個人間のファイル受け渡しシステムでしたが、多くのユーザーがフォルダ内の整理をしないので、フォルダ内のアイテムが多くなりすぎて十分機能を果たさなくなりました。そこで考えた工夫です。

★ PostgreSQL 上のポストメタファ管理表と PowerShell コード

○ポストメタファは，PostgreSQLで，以下の表を使って管理している

職員表	ポストを持つ職員のアカウントと姓，名，その他
ポスト管理表	ポストに存在するアイテム表（「_ゴミ箱_」フォルダを除く）
削除表	「_ゴミ箱_」フォルダに移動したアイテム表
エラーログ	処理中エラーを起こした場合，エラーを起こした処理，アイテム，時間等を記録する表
処理実行済	毎日１回の処理で，正常に処理を終わったアカウントと時間を記録する。正常に運用されていることを，一目で確認するために作成した表。



○ポストフォルダ管理プログラム PostCleaner.ps1（一部を改変して抜粋）

```
・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・
$pg.sql = “SELECT * FROM ポスト管理表;”;
$dbItemAll = $pg.read();                          # 「ポスト管理表」にある全職員のポストアイテムを取得
foreach ( $fld in $userPostFldrs ) {              # 一人一人の職員のポストフォルダーについて
  $dbItem = $dbItemAll | where { $_.account –eq $fld.Name };   # 「ポスト管理表」内の特定アカウントレコードを取得
  $Item_in_Post = Get-ChildItem $fld.FullName;  # 各ポストフォルダー内のアイテムを取得
  foreach ( $item in $Item_in_Post ) {            #各ポストフォルダー内のアイテムの一つ一つについて
    $target = $item | where { $_.Nmae -eq $dbItem.Name };      #「ポスト管理表」の同じアイテム名のレコードを取得
    if ( $target ) {  #同じアイテム名のレコードが見つかったら
      #「ポスト管理表」のデータと比較して必要な処理を行う
    } else {
      #「ポスト管理表」にレコードを追加
    };
    #「ポスト管理表」にあるレコードのアイテム名と，同じ名のアイテムがフォルダー内になければ，レコードを削除
    # この処理も上記と同じような処理となる
};
・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・
```

注：内容は説明のため改変してあるので，PostCleaner.ps1 そのもののコードではありません。

● PostgreSQL Access Class の利用例（その２）

２）MACアドレスフィルタリング管理

ａ）本校ではWindows Server 2008 R2 の Active Directory ドメインサービスに，NPS サーバーを登録して（RADIUS認証が可能になる），以下の２つの用途で，MACアドレス認証を行っている。

(1) 無線LAN接続（職員室の無線LANと，タブレット・スマートフォン用のWi-Fi）

(2) ファイルサーバーアクセス用の職員系ネットワーク接続（有線も無線もある）← 予定

ｂ）認証は，無線LANアクセスポイントやL3スイッチのRADIUS連携機能を利用する。

ｃ）登録MACアドレス数が多いので，学校管理データベースと連携して，有効な管理を行いたい。

ｄ）PowerShell の Active Directory ドメインサービスの管理用モジュールと PostgreSQL Access Class をを利用して管理ツールの作成に取りかかっている。

ｅ）当初の予定では，現時点でα版完成を目指していたが，大幅に遅れ，AD管理用モジュールのテストを始めるところである。今年度末にはなんとか使える状態にもっていきたい。

３）PostgreSQL Access Class を利用して，これから開発したいツールは

(1) Active Directory ドメインサービス，および，ファイルサーバーは定期的（特に年度更新）にメンテナンスが必要である。この用途のツールには，学校管理データベースとの連携が有効である。

・データベースのユーザーインターフェースには Excel を利用する（GUIの設定）

・PostgreSQL Access Class を利用してデータベースと連携し，サーバーの更新を行う

以上の流れの更新ツールの作成を来年度おこないたい。現時点で一部（ポストのアクセス権設定と運用管理）はできている。

(2) 学校において管理が必要で，合計が大きなサイズとなってファイルサーバーを圧迫するデータに写真ファイル（今後動画も考えておかなければならない）がある。専用のNASを用意してそちらに保存させたいが，ファイルサーバーを監視して，問題点をデータベースに登録し，管理を行いたい。そのためのツール作成に PostgreSQL Access Class を利用する予定である。

● 終 わ り に

1）PostgreSQL Access Class は以下の Webpage で公開します

・スクリプト言語ですので，当然オープンソースです。

・ライセンスはBSDライセンス規定に従います。

・公開日は本日（11月８日）で，URLは以下の通りです。

http://www.mitene.or.jp/~tetsuya/

2）今後，もう少し内容を充実させていく予定です

・マニュアルはまだ必要最低限の内容です。２～３ヶ月かけて充実させます。

・PostgreSQL Access オブジェクト自身の簡単なヘルプ機能も，拡張機能として設ける予定です。

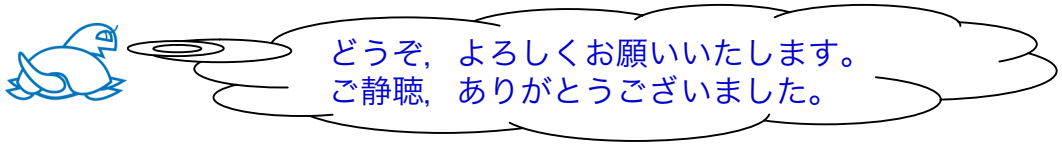
・DataReaderオブジェクトを使った，データベースから１レコードずつ読み込むメソッドを，拡張機能として作成する予定です。

・CSVファイルや連想配列のデータによって，レコードの追加（INSERT）と訂正（UPDATE）を同時に実行するメソッドを，拡張機能として作成したいと考えております。

・PREPARE文の設定で，変数を $1，$2 のような数値ではなく，:param のような変数名で指定できるスクリプトプロパティーを考案中です。実行は，変数名をKeyとした連想配列を引数に持つメソッドで行います。 →　$pg.ExecutePrepareStatement($hashAry);

3）お　願　い

PostgreSQL Access Class は，私個人の利用としては十分な機能を有していますが，PostgreSQLについても，PowerShellや.NETについてもプロではありませんので，欠けていたり不十分な点があると思います。公開を機にご批判を仰ぎながら改良を進めていくことができればと考えております。



Appendix 1　PowerShell 言語のダイナミックスコープ（動的スコープ）

PowerShell プログラム（DynamicScope.ps1）

```
# ダイナミックスコープを検証する

function Abc {
   Write-Host  ( "       function abc では " + $var + " です。" );
   $var = "abc";
};

function Def {
   Write-Host  ( "代入前 function def では " + $var + " です。" );
   $var = "def";
   Abc;  #← 関数 Abc の呼出
   Write-Host  ( "呼出後 function def では " + $var + " です。" );
};

function Ghi {
   Write-Host  ( "代入前 function ghi では " + $var + " です。" );
   $var = "ghi";
   Abc;  #← 関数 Abc の呼出
   Write-Host  ( "呼出後 function ghi では " + $var + " です。" );
};

# Start of DynamicScope.ps1
$var = "xyz";

Write-Host  ( "関数の外で出力すると      " + $var + " です。" );
Def;  #← 関数 Def の呼出
Ghi;  #← 関数 Ghi の呼出
Write-Host  ( "関数の外で出力すると      " + $var + " です。" );
```

実行結果

```
PS C:¥> ./DynamicScope.ps1
関数の外で出力すると      xyz です。
代入前 function def では xyz です。
       function abc では def です。
呼出後 function def では def です。
代入前 function ghi では xyz です。
       function abc では ghi です。
呼出後 function ghi では ghi です。
関数の外で出力すると      xyz です。
```

- 子（呼び出された側）ブロックでは親（呼出側）ブロックで参照可能な全ての変数を参照できる
- 子（呼び出された側）ブロックでの変数への代入が変数宣言となり、そのブロックのローカル変数となる
- 親（呼出側）ブロックで参照可能な変数を全て値引数で渡すのと同じ
- $ErrorActionPreferenceのようなシェル変数もダイナミックスコープである

Appendix 2　PoweShellでCustomClassを定義する

```
# カスタムクラスの型を格納する変数
  $Global : __ClassTable__ = @{};

function Global : CustomClass {
   param ([string] $type, [ScriptBlock] $definition);

   __New_Instance  $definition > $null;
   $Global:__ClassTable__[$type] = $definition;
}; # end of CustomClass function

function Global : __New_Instance( $definition )  {
   ..........................................

   function ScriptProperty( $name, $get, $set) {
      $sbStr = $set.ToString().Trim();
    # ScriptProperty オブジェクトの出力
      if ( $sbStr -eq '' ) {
         New-Object Management.Automation.PSScriptProperty `
            $name, $get;
      } else {
         New-Object Management.Automation.PSScriptProperty `
            $name, $get, $set;
      };
   }; # end of ScriptProperty function
   ..........................................

 # 空のオブジェクトを作成する
   $object = New-Object  Management.Automation.PSObject;
   $members = &$definition; # スクリプトブロック$definitionを実行
   foreach ($member in $members) {
      $Object.PSObject.Members.Add( $member ); # メンバを追加
   };
   $Object; # メンバを追加したオブジェクトを出力
}; # end of __New_Instance function
```

```
function Global : new ( $type ) {
   $definition = $__ClassTable__[ $type ];
   __New_Instance  $definition;
}; # end of new function
```

CustomClass定義用のスクリプトを実行しておいて

↓

```
# CustomClass myClass の定義
CustomClass myClass {

   ScriptProperty  myProperty  { $this.myName; } `
                           { param ( $myName );
                             if ( $myName -is [string] ) {
                                $this.myName = $myName;
                             } else {
                                Write-Host  "このプロパティーには …";
                             }; # end if
                           };
};

# CustomClass myClass の定義にしたがって，オブジェクトを取得
$myObj = new  myClass;
```

注：左記および上記スクリプトはCustomClass定義の基本アルゴリズムを示したもので，開発したスクリプトコードではありません。