コマンドプロンプトから PowerShell に乗り換えるための小さな本

2015 年 5 月 初版

内容

著作 http://bonk.red

はじめに

Windows でコマンド操作と言えば伝統的にコマンドプロンプトが使われてきました。このコマンドプロンプトですが、元々は MS-DOS を Windows で使うためのエミュレータ (COMMAND.COM)でした。現在のコマンドプロンプト(CMD.EXE)は、MS-DOS とは直接関係なくなりましたが、画面のデザインから操作方法まで MS-DOS を踏襲しています。

このため、機能不足が指摘されて、Windows Scripting Host (WSH)などで機能強化されましたが、シェルとしては相変わらず MS-DOS を引き継いでいました。

以前はこのコマンドプロンプトしかなかった Windows のコマンドベースのシェルですが、Windows Vista のときに Windows PowerShell 1.0 がリリースされ、現在(2015 年 5 月)は、バージョン 4.0 になっています。最初はドキュメントが不足し、開発に使われることも少なかったと思われますが、さすがにバージョン 4.0 になるとドキュメントや開発情報も豊富になり、機能、安定性も十分なものになっています。

使い方

実行ポリシーの変更

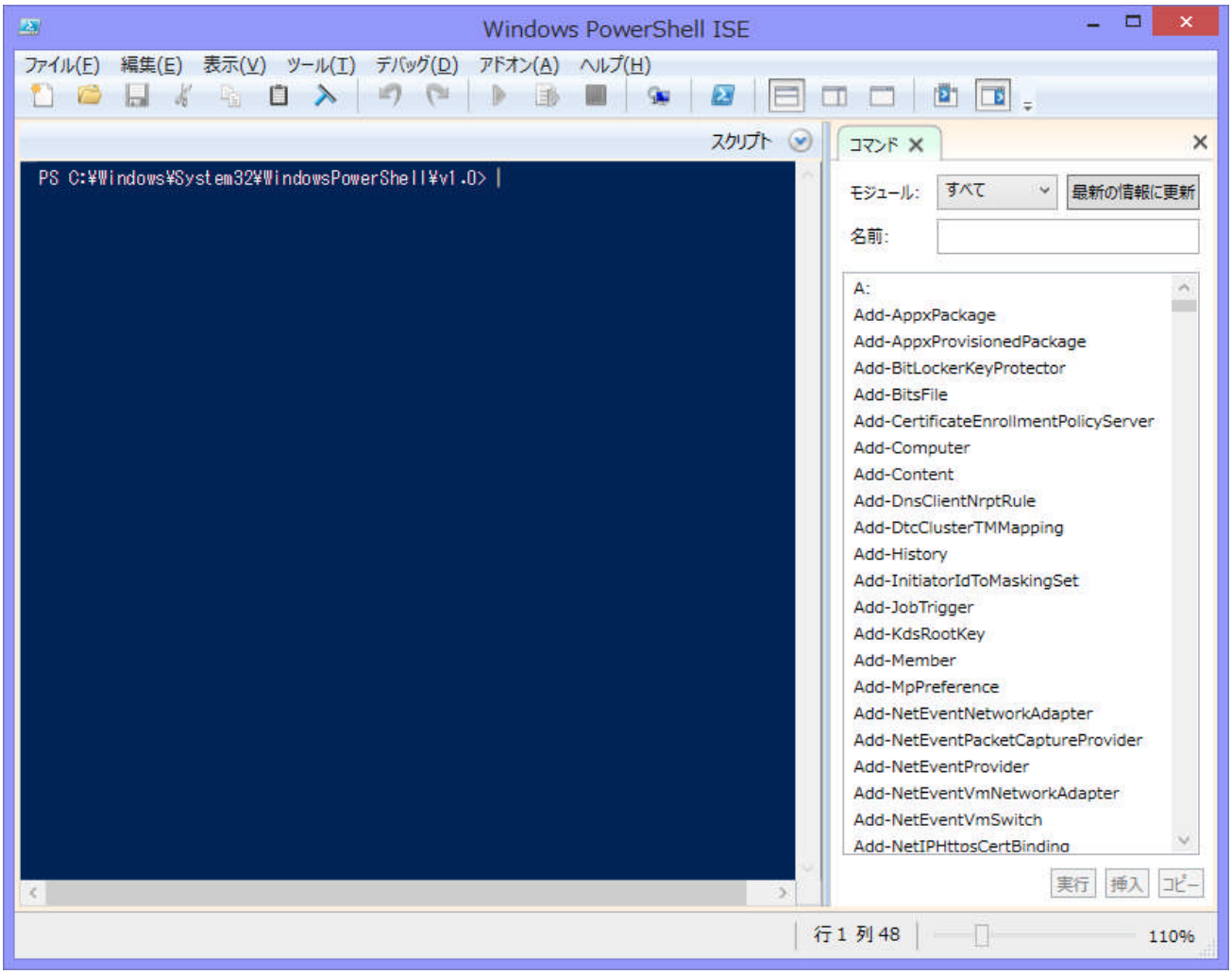
Windows 7 の場合は、スタートメニューから起動できます。Windows 8.1 の場合はスタート画面には登録されていないので、アプリ画面に移動し、横スクロールで「Windows システムツール」へ移動しそこに登録されているアイコンをクリックします。これだと使い勝手が悪いので、よく使うならスタート画面に「ピン留め」したほうがよいでしょう。

ところで、Windows PowerShell と Windows PowerShell ISE と 2 種類のアイコンが登録されていますが、ISE は開発用のバージョンです。つまり、Windows PowerShell スクリプトを開発するとき使うと便利です。

コマンドプロンプトと違って PowerShell は、スクリプトファイルの実行がデフォルトで禁止状態になっています。これは、コンピュータに詳しくない人が、どこかからダウンロードしたり、メールに添付されていたスクリプトを不用意に実行できないようにするためです。したがって、使う前にこれを解除しておく必要があります。

解除方法ですが、管理者として PowerShell を起動し、次のコマンドを実行します。

```
Set-ExecutionPolicy RemoteSigned
```



Windows PowerShell ISE (バージョン 4.0 のもの)

モジュール用ディレクトリの作成とプロファイル

これは必須ではありませんが、モジュール用ディレクトリを作ります。場所ですが、$profile という組み込み変数の内容を表示すると確認できます。

```
PS C:¥> $profile
C:¥Users¥user¥Documents¥WindowsPowerShell¥Microsoft.PowerShell_profile.ps1
PS C:¥>
```

次のようにして、実際にこのディレクトリが存在するか確認します。最初は存在しないはずです。

```
PS C:¥> test-path $profile
False
```

存在していないならそのディレクトリを作成します。

```
PS C:¥> mkdir C:¥Users¥user¥Documents¥WindowsPowerShell
```

ついでにモジュール用ディレクトリも追加しておきます。

```
PS C:¥> mkdir C:¥Users¥user¥Documents¥WindowsPowerShell¥Modules
```

エディタでプロファイル(Microsoft.PowerShell_profile.ps1)を作成します。内容ですが、PowerShell を起動したとき自動的に実行するコマンド(一般に複数)を記述します。とりあえず、ここでは作業用のフォルダに移動するようにしておきます(下記)。

```
Set-location C:¥workspace¥Scripts
```

PowerShell の操作方法

PowerShell の操作方法ですが、コマンドプロンプトと大体同じです。使えるコマンドも同じようになっています。実は、コマンドプロンプトと同じコマンドは「エイリアス」で、PowerShell のネイティブのものは「コマンドレット」と呼ばれる「動詞-対象」形式のものです。

エイリアス一覧は alias (または get-alias)コマンドで表示できます。下に表示例を示します。

```
CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Alias           % -> ForEach-Object
Alias           ? -> Where-Object
Alias           ac -> Add-Content
Alias           asnp -> Add-PSSnapin
Alias           cat -> Get-Content
Alias           cd -> Set-Location
Alias           chdir -> Set-Location
  ....
```

コマンドプロンプト互換のエイリアスだけでなく Unix 互換のものも用意されています。さらに自分でエイリアスを定義することも可能です。

Unix 互換のエイリアスも含め、大文字と小文字の区別はされません。例えば、ls も LS も Ls もフォルダの内容一覧を表示します。

コマンド履歴を表示するには、F7 キーを押します。あるいは history (あるいは get-history)コマンドを実行します。
また、上下矢印キーで履歴バッファのコマンドを現在のプロンプト位置に呼び出すこともできます。

場所を移動するとき便利なエクスプローラからのドラッグ＆ドロップも可能です。(管理者モードでは不可)

PowerShell に組み込まれているコマンドレットだけでなく、一般の Windows アプリケーション、Java アプリケーション、

WSH(VBScript など)や PowerShell のスクリプトなども実行できます。

リダイレクトやパイプラインもコマンドプロンプト同様に使えますが、入力のリダイレクト(<)だけは使えません。

基本的なコマンドレット

操作	コマンドレット	エイリアス
フォルダの内容一覧	Get-ChildItem	dir, ls
場所を移動する。	Set-Location	cd, chdir
文字列を表示する。	Write-Host, Write-Output	echo
テキストファイルの内容を表示する。	Get-Content	type, cat
ファイルのコピー	Copy-Item	copy, cp
ファイルの削除	Remove-Item	del, rm
ファイルのリネーム	Rename-Item	ren, mv
ディレクトリの作成	New-Item	md, mkdir
ディレクトリの削除	Remove-Item	rd, rmdir
履歴の表示	Get-History	history
履歴コマンドの実行	Invoke-History	F7 キー
エイリアスの表示	Get-Alias	alias
ヘルプの表示	Get-Help	help
画面のクリア	Clear-Host	cls

参考 Write-Host, Write-Output の違い

```
PS C:¥> $a=1,2,3
PS C:¥> write-host $a
1 2 3
PS C:¥> write-output $a
1
2
3
PS C:¥>
```

PowerShell スクリプト

PowerShell のスクリプトですが、拡張子が .ps1 です。実行するときは、拡張子は省略できます。カレントディレクトリにあるスクリプトを実行する場合、".¥"を先頭につける必要があります。コマンドプロンプトではこれは不要ですが、Bash などでは必要です。

例えば、Script1.ps1 というスクリプトがカレントディレクトリにあり、それを実行したい場合は次のようにします。

```
PS C:> .¥Script1
```

コマンドレット

コマンドレットは.NET Framework のクラスなので、扱うデータも.NET Framework のオブジェクトです。パラメータも出力データも.NET Framework のオブジェクトになります。

文字列の表示

文字列の表示は echo(Write-Output)や Write-Host コマンドレットを使いますが、文字列（さらにオブジェクト）だけを書いておけばコンソールにその内容が文字列として出力されます。

```
PS C:¥> $s = "Hello"
PS C:¥> $s
Hello
```

コマンドライン引数

コマンドライン引数は$Args という組み込み変数です。PowerShell では変数も大文字・小文字の区別をしないので$args, $ARGS と書いても同じです。これは、オブジェクトの配列なので文字列以外もパラメータとすることができます。また、.NET Framework の配列なので長さは Length プロパティで取得できます。（プロパティやメソッドも大文字・小文字を区別しません）

サンプル

```
# コマンドライン引数
$Args.GetType()
$Args.Length
if ($Args.Length -gt 0) {
  $Args | ForEach-Object { Write-Output $_ }
}
```

実行例

```
PS C:¥> .¥args.ps1 100 120

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
2
100
120
```

スクリプトの終了

スクリプトの最後まで実行すれば終了します。終了させるために特別なステートメントは必要ありません。しかし、途中で終了させる場合は、exit を実行します。

サンプル

```
if ($args.Length -eq 0) {
  "No parameters"
  exit
}
echo $args.length
```

実行例

```
PS C:¥> .¥quit.ps1
No parameters
```

変数

PowerShell では変数も大文字・小文字の区別はしません。そして、変数には$を付けます。また、既定の変数(組み込み変数)があり、それらを他の目的に使うことはできません。例えば、コマンドライン引数は $args ですが、値を代入しても無視されます。

例

```
PS C:¥> $args="))"
PS C:¥> $args
PS C:¥>
```

変数に値を代入するには=を使います。Bash のように=を変数にくっつけなければならないみたいなことはありません。

配列

コマンドプロンプトには配列がありませんが、PowerShell では利用できます。配列は次のようにして初期化します。

```
$a = 0,1,2,3,4
PS C:¥> write-host $a
0 1 2 3 4
```

連続した値なら .. 演算子を使って次のようにすることもできます。

```
$a = 0..4
PS C:¥ > write-host $b
0 1 2 3 4
```

配列の要素は [] 演算子を使ってアクセスします。

```
PS C:¥ > $a[1]
1
```

ところで配列の長さは固定なので、要素を追加したり削除したりはできません。

連想配列

PowerShell では連想配列も利用できます。連想配列は次のようにして初期化します。

```
PS C:¥I> $h = @{"dog"="犬";"cat"="猫";"mouse"="ねずみ"}
```

要素の参照は配列同様[]演算子を使用します。

```
PS C:¥> $h["dog"]
犬
```

配列と違って要素を追加することもできます。

```
PS C:¥> $h["bird"]="鳥"
PS C:¥> $h["bird"]
鳥
```

この例ではキーに文字列を使っていますが、一般にはキーはオブジェクトですので、連続した整数を使うと配列のように使用できます。

```
PS C:¥> $arr=@{0=10;1=11}
PS C:¥> $arr[2]=12
PS C:¥> $arr[0]
10
PS C:¥> $arr[2]
12
```

演算子

PowerShell の演算子は豊富です。算術演算子(+, -, *, /, %)は他の言語とだいたい同じですが、比較演算子は ==, >, < などではありません。>はリダイレクトの意味になります。比較演算子は

- -eq 等しい
- -ne 等しくない
- -lt より小さい
- -le より小さいか等しい
- -gt より大きい
- -ge より大きいか等しい

などになります。

この他、代入演算子(=, += etc)、単項演算子(++, --, -)、正規表現演算子(-match etc)、論理演算子(-and, -or etc)など多彩です。

関数

コマンドプロンプトには関数がありませんが、PowerShell では関数も利用できます。関数は function 文で定義します。

サンプル

```
function root([double] $x) {
  if ($x -lt 0.0) {
    return -1.0
  }
  $y = [System.Math]::sqrt($x)
  return $y
}

root(2)
```

実行例

```
1.4142135623731
```

オブジェクト

PowerShell で扱う変数はすべて.NET Framework のオブジェクトです（System.Object またはその派生クラスのインスタンス）。したがって、今扱っている変数が何なのかを意識して使う必要があります。それは GetType()メソッドを使って知ることができます。

例

```
PS C:¥ > $a.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

変数はオブジェクトなのでプロパティやメソッドを持ちます。上の例で言うと System.Array（つまり配列)なので、長さは Length プロパティで取得できることがわかります。

```
PS C:¥ > $a.length
5
```

PowerShell の変数はオブジェクトなのでしばしば型の変換が必要になる場合があります。型の変換(キャスト)は、[type]演算子を使います。

例： 整数$i を文字列$s に変換する。

```
PS C:¥workspace¥misc¥PowerShell¥syntax> $i = 100
PS C:¥workspace¥misc¥PowerShell¥syntax> $i.gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Int32                                    System.ValueType

PS C:¥workspace¥misc¥PowerShell¥syntax> $s = [string]$i
PS C:¥workspace¥misc¥PowerShell¥syntax> $s.gettype()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
```

```
True        True        String                                   System.Object
```

パイプラインとリダイレクト

コマンドプロンプトや Bash でおなじみのパイプラインやリダイレクトも使えます。ただし、入力のリダイレクト(<)は現在のところ使えません。(PowerShell 4.0)

例 C:¥scripts というディレクトリに含まれるファイルの数を表示する。

```
PS C:¥> get-childitem C:¥scripts -recurse | measure-object
```

例 C:¥scripts というディレクトリに含まれるファイルの一覧をテキストファイルとして保存する。

```
PS C:¥> get-childitem C:¥scripts -name > filelist.txt
```

.NET Framework の利用

PowerShell は.NET Framework ベースのシェルなので、様々な場面で.NET Framework のお世話になることになります。.NET Framework のクラスを使うには、New-Object コマンドレットでクラスをインスタンス化して使用します。

例 System.Text.RegularExpressions.Regex(ファイルリストに含まれる拡張子 jpg のファイルのみを表示する)

```
$re = new-object -TypeName "System.Text.RegularExpressions.Regex" -ArgumentList "¥.jpg$"
$list = "001.jpg", "002.jpg", "003.png", "004.gif", "005.jpg"

foreach ($x in $list) {
  if ($re.ismatch($x) -eq $true) {
    $x
  }
}
```

実行例

```
PS C:¥> .¥regex.ps1
001.jpg
002.jpg
005.jpg
```

例 System.Text.StringBuilder (文字列を連結する。不要な表示をリダイレクトを利用して null に捨てている)

```
$sb = new-object "System.Text.StringBuilder"
$sb.Append("System.") > $null
$sb.Append("Text.") > $null
$sb.Append("StringBuilder")   > $null
$sb.ToString()
```

実行例

```
PS C:¥> .¥stringbuilder.ps1
System.Text.StringBuilder
```

スタティックなメソッドはインスタンス化は不要で、[type]::を使って直接メソッドを呼び出します。

例 System.Convert クラスのメソッドの使用例

```
$n = 100
$bn = [System.Convert]::ToByte($n)
$bn.GetType().Name
$bn
$cn = [System.Convert]::ToChar($n)
$cn.GetType().Name
$cn
```

実行例

```
PS C:¥> .¥convert.ps1
Byte
100
Char
d
```

COM の利用

COM ベースである MS オフィスなどと連携する場合は、New-Object コマンドレットで-ComObject オプションを使ってオブジェクトを作成します。

例 Excel ファイルのセルを読み込んで表示する。

```
$excel = new-object -comobject Excel.Application
$excel.visible = 1    # Excel を表示したくない場合はここをコメントアウト
# Excel ファイルの場所はフルパスで指定する。
$book = $excel.workbooks.open("C:¥workspace¥Misc¥PowerShell¥object¥Book1.xlsx")
$sheet = $book.worksheets.item("sheet1")
echo $sheet.range("A1").text
echo $sheet.cells.item(1, 2).text
# $excel.quit()   # Excel が非表示の場合はこの行を有効にする。
```

既存のスクリプトの利用

既存のスクリプトを実行するにはドット演算子を使います。

例 呼出し側 (call_hello.ps1)

```
# hello.ps1 を実行する。
. .¥hello.ps1
```

例 呼び出される側 (hello.ps1)

```
echo "Hello, world!"
```

実行例

```
PS C:¥> .¥call_hello.ps1
Hello, world!
```

応用例

ショートカットの一括変更

　ショートカットを作っておいたファイルを移動するとショートカットはそのファイルの正しい場所(移動先)がわからないので機能しなくなります。でも、ショートカットの内容を書き換えてやれば正しく動作するようになります。
　次の応用例は、あるディレクトリごとファイルを移動したとき、それらのファイルを参照するショートカットの内容を一括して書き換えるものです。

```
# フォルダ内のショートカットを検索し、ショートカットの内容を変更する。
clear-host

# ショートカットを検索する対象フォルダ
$folder = "C:¥data¥Pictures"
write-host "対象フォルダ："  $folder
# 置換対象(検索)文字列
$search = "D:¥"
# 置換する文字列
$replace = "C:¥data¥"

# ショートカット一覧を得る。
"ショートカットを検索中。しばらくお待ちください ..."
$shortcuts = get-childitem -path $folder -recurse -include *.lnk

$count = $shortcuts.length
if ($count -eq 0) {
  "ショートカットが見つかりませんでした。", "終了します。"
  exit 9
}

$shell = new-object -comobject WScript.Shell

write-host $count "件のショートカットが見つかりました。"
$shortcuts | foreach-object {
  echo $_.fullname
  $wsc = $shell.CreateShortcut($_.fullname)
  if ($wsc.TargetPath.IndexOf("D:¥Pictures") -eq 0) {
    echo $wsc.TargetPath
    $wsc.TargetPath = $wsc.TargetPath.Replace($search, $replace)
    $wsc.Save()
  }
}
```

CSV ファイルの作成

PowerShell には CSV ファイルを扱うための Export-CSV コマンドレットが用意されていますが、これはオブジェクトを CSV として出力するもので、Excel のデータを扱うには便利ではありません。
ここでは、COM オブジェクトとして Excel ブックを取得して、シート 1 のデータを行と列を入れ替えて CSV ファイルとして保存してみます。

```
#
#   CSV ファイルの作成
#

if ($args.length -eq 0) {
  "Excel ファイルを指定してください。"
  exit
}

$filename = $args[0]
$csvfile = $args[1]

# Excel ブックオブジェクトを取得
[reflection.assembly]::LoadWithPartialName("'Microsoft.VisualBasic") > $null
$book= [Microsoft.VisualBasic.Interaction]::GetObject($filename)

# シート 1 を選択する。
$sheet = $book.worksheets.item(1)

# 行数を取得
$i = 0
$val = $sheet.cells.item($i + 1, 1).text
while ($val -ne "") {
  $i++
  $val = $sheet.cells.item($i + 1, 1).text
}

$rows = $i

# 列数を取得
$j = 0
$val = $sheet.cells.item(1, $j + 1).text
while ($val -ne "") {
  $j++
  $val = $sheet.cells.item(1, $j + 1).text
}

$cols = $j

$fs = new-object "System.IO.StreamWriter" -argumentlist $csvfile,$true
for ($i = 1; $i -le $cols; $i++) {
  $line = ""
  for ($j = 1; $j -le $rows; $j++) {
```

```
      $line += $sheet.cells.item($j, $i).text
      if ($j -lt $rows) {
        $line += ","
      }
    }
    $fs.WriteLine($line)
  }
  $fs.close()
  echo "done."
```

ゴミ箱へファイルを捨てる

ファイルを削除するコマンドレットは、Remove-Item ですが、これは直ちにファイルを削除します。削除でなくゴミ箱に入れたい場合は、次のようにシェルの機能を利用します。

```
  # ファイルやフォルダをごみ箱に捨てる。
  #    $args[0]  コマンド引数に捨てるファイルのフルパスを指定する。
  if ($args.length -eq 0) {
    "ごみ箱に捨てるファイルを指定してください。"
    exit
  }

  $filename = [System.IO.Path]::GetFileName($args[0])
  $dirname = [System.IO.Path]::GetDirectoryName($args[0])
  $shell = new-object -comobject Shell.Application;
  $folder = $shell.Namespace($dirname);

  $item = $folder.ParseName($filename)
  $item.InvokeVerb("delete")
```

Thumbs.db の一括削除

フォルダに画像ファイルがあると自動的に隠しファイル Thumbs.db というファイルが作られます。ところがこのファイルを何かのプロセスがロックしてそのフォルダが削除できないことがあります。事前にこのファイルを削除するとそのようなケースが少なくなります。

```
  #
  #   指定したフォルダ内の隠しファイル thumbs.db を削除する。
  #    (サブフォルダを含む)
  #   args[0] フォルダのパス
  #   args[1] 省略したときは削除候補の表示のみを行う。任意の文字を設定すると削除する。
  #

  if ($args.length -eq 0) {
  @"
    指定したフォルダ内の隠しファイル thumbs.db を削除します。
```

```
    フォルダを指定してください。
"@
    exit
}

if ($args.length -eq 1) {
    # 削除候補のみを表示する。
    remove-item $args[0] -include thumbs.db -recurse -whatif -force
}
else {
    # 実際に削除する。
    remove-item $args[0] -include thumbs.db -recurse -force
    "削除しました。"
}
```

フォルダのサイズ

フォルダに含まれるファイルサイズの合計を求めるスクリプトです。

```
#
#   指定したフォルダ内のファイルサイズの合計を求める。
#
if ($args.length -eq 0) {
    "パラメータを指定してください。folder"
    exit
}

$(get-childitem $args[0] -recurse | foreach-object {
    $_.length
} | measure-object -sum).sum
```