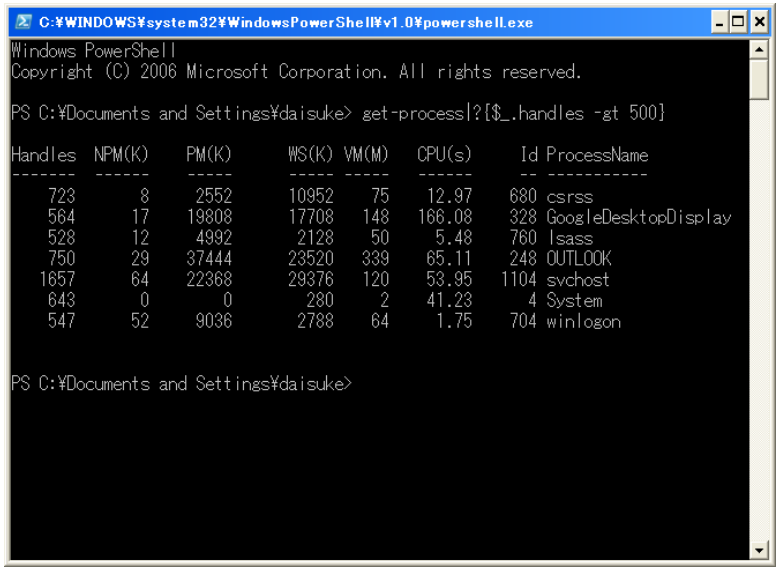
Windows PowerShell
～.NETベースのシェル・スクリプト実行環境～
Part 2

むたぐち＠わんくま同盟

PowerShell

- .NET Frameworkベースの新しいシェル・スクリプト実行環境
  それが、Windows PowerShellです。
  （開発コードMonad、旧称MSH(Microsoft Command Shell)）



PowerShellのダウンロード

- PowerShell v1.0の正式版が先日(2006/11/14)リリースされました！
  - Windows Server 2003 Service Pack 1 および Windows XP Service Pack 2 用の Windows PowerShell 1.0 ローカライズ版インストール パッケージ
  - http://support.microsoft.com/kb/926140
  - .NET Framework Version 2.0が必須
  - Vista版は1/31にリリース予定

PowerShellのPowerの源　コマンドレット

- PowerShellにはデフォルトで120種を超えるCmdlet（コマンドレット）が含まれている。
  - コマンドプロンプトで言うところの「内部コマンド」に相当
- コマンドレットを単独で、あるいは組み合わせることで様々な処理を実現可能。
- コマンドレットの引数も戻り値もみな.NETのオブジェクトである。
  - コマンドレット自体も…

コマンドレットの基本(1)　命名法

- コマンドレット命名法は

  ”Verb-Noun”（動詞-名詞）

- 例：ディレクトリを移動するSet-Locationコマンドレット(コマンドプロンプトのcdに相当)

```
Windows PowerShell
Copyright (C) 2006 Microsoft Corporation. All rights reserved.

PS C:¥Documents and Settings¥daisuke> Set-Location -Path C:¥
PS C:¥>
```

コマンドレットの基本(2)　ヘルプ

- どんなコマンドレットがあるのかを調べるには

  Get-Command

- コマンドレットのヘルプを引くには

  Get-Help コマンドレット名

  または

  コマンドレット名 -?

- .NETオブジェクトのメンバ(プロパティ、メソッドなど)を調べるには

  コマンドレットなどの後に|Get-Member

コマンドレットの基本(3)　パラメータ

- コマンドレットのパラメータはすべて

  -パラメータ名　または　-パラメータ名 パラメータ

  （c.f. Cmdlet -param1 value -param2 value -param3）
- パラメータによってはパラメータ名を省略できる。
- 文字列はスペースを含まない限り""で括らなくて良い。
- コマンドレットに共通のパラメータがある。
  - -?, -Verbose, -Debug, -ErrorAction, -ErrorVariable, -OutVariable, -OutBuffer, -WhatIf, -Confirm

統一的なコマンド体系

コマンドレットの基本(4)　省力化

- コマンドレットにエイリアスが定義可能。デフォルトでもいくつか定義されている。
  （Get-Aliasで一覧を取得可能）

Set-Location	sl, cd ,chdir
Get-ChildItem	gci, dir, ls
Get-Process	gps

etc

- コマンドレット、パラメータ名、パラメータ、すべて、大文字と小文字を区別しない。（変数、メソッド名なども）
- パラメータ名の省略、一部省略
  （-path→省略、-exclude→-ex）
- タブ補完
  （set-<Tab>→Set-Acl→Set-Alias→ Set-AuthenticodeSignature）

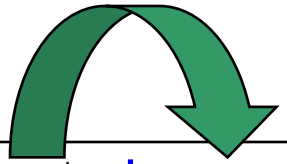
PSドライブ(3) デモ

DEMO1-1

従来のシェルにおけるパイプ

- カレントのファイルをファイル名で逆順ソート

dir / bの出力＝テキストがパイプを通る



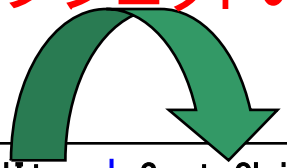
```
C:¥WINDOWS¥system32¥drivers¥etc>dir /b | sort /r
services
protocol
networks
lmhosts.sam
hosts
```

- 上の例は単純なテキストなのでソートできるが、ではサイズでソートするには？

？？？

オブジェクトが渡るパイプ(1) 概要

- ファイルサイズでソート、PowerShellなら可能です。
  Get-ChildItemの出力＝オブジェクトの配列がパイプを通る



```
PS C:¥WINDOWS¥system32¥drivers¥etc> Get-ChildItem | Sort-Object -property Length


    ディレクトリ: Microsoft.PowerShell.Core¥FileSystem::C:¥WINDOWS¥system32¥dri
    vers¥etc


Mode                 LastWriteTime     Length Name
----                 -------------     ------ ----
-a---          2004/08/05     21:00        407 networks
-a---          2004/08/05     21:00        734 hosts
-a---          2004/08/05     21:00        799 protocol
-a---          2004/08/05     21:00       3683 lmhosts.sam
-a---          2004/08/05     21:00       7116 services
```

FileInfoオブジェクトのLengthプロパティを元にソートされる。

オブジェクトが渡るパイプ(2) フィルタと列挙

- Where-Objectを使うと細かくフィルタ可能

?またはwhereでも可

省略可

スクリプトブロック。$_にはパイプに渡されたオブジェクトが格納

```
PS C:¥> Get-Process | Where-Object -filterScript {$_.handles -gt 500}
```

稼働中のプロセスからハンドル数が500より多いものを列挙

- Foreach-Objectでパイプを渡ったオブジェクト配列の要素それぞれに対してコマンド実行可能。

%またはforeachでも可

```
PS C:¥Documents and Settings¥daisuke> Get-ChildItem | Foreach-Object -
process{Write-Host $_.FullName}
```

省略可

スクリプトブロック。$_にはパイプに渡されたオブジェクトが格納

カレントにあるファイルのフルパスの一覧を表示

WMIも自由自在(1) Before & After

- WMI (Windows Management Instrumentation)のクラスのインスタンスを簡単に呼び出せる。

WSH with VBScriptでは…

```
Set wbemServices = GetObject("winmgmts:¥¥.")
Set wbemObjectSet = wbemServices.InstancesOf("Win32_LogicalMemoryConfiguration")

For Each wbemObject In wbemObjectSet
    WScript.Echo "物理メモリの合計 (kb): " & wbemObject.TotalPhysicalMemory
Next
```

長ったらしい…

PowerShellではこんなに簡単！

```
PS C:¥> Get-WMIObject -class Win32_LogicalMemoryConfiguration -property TotalPhy
sicalMemory
```

省略すればさらに簡単！

```
PS C:¥> gwmi Win32_LogicalMemoryConfiguration -p TotalPhysicalMemory
```

パイプの扱いとWMIのデモ

DEMO1-2

PowerShellスクリプティング　導入編

- スクリプトは*.ps1ファイル（テキストファイル）に記述。
  - 日本語を使う場合はencodingにShift-JISやUTF-8などを用いる。
- 関連付けをする場合は手動で（レジストリを直接操作し）行う。
- スクリプト実行ポリシーの設定。
  - デフォルトは"Restricted"（スクリプトの実行不可）
  - **Set-ExecutionPolicy**で"RemoteSigned"（ローカルのスクリプトは無制限に実行可）などにする。

スクリプトの実行

- *.ps1ファイルのフルパスまたは相対パスを指定。

✕
```
PS C:¥script> test.ps1
```

○
```
PS C:¥script> C:¥script¥test.ps1
```

○
```
PS C:¥script> .¥test.ps1
```

- ドットの後にスペースを入れ、その後にスクリプトパスを指定すると、「ドットソース」となる。（スクリプトソースをグローバルスコープに読み込む）

```
PS C:¥script> . .¥test.ps1
```

基本的なスクリプト

- 基本はシェル操作の延長。コマンドラインに入力していたコマンドを複数行に記述する。

C:¥script¥log_windir.ps1

```
$path = "C:¥script¥log.txt"
"日付" >> $path
Get-Date >> $path
Get-ChildItem $env:windir >> $path
"フォルダのサイズ" >> $path
Get-ChildItem $env:windir -recurse -force | measure-object length -sum >>
$path
```

C:¥script¥log.txtというファイルに、現在の日付と時刻、Windowsフォルダ内のファイルとフォルダの一覧、Windowsフォルダのサイズを書き込む。

デモ　～スクリプトの作成から実行まで

DEMO2-1

豊富な演算子

- 数値演算子
  - +, -, *, /, %
- 代入演算子、単項演算子
  - =, +=, -=, *=, /=, %=, ++, --
- 比較演算子（()内は多言語での表記。それぞれcをつけるとcase-sensitiveに）
  - -lt(<), -le(<=), -gt(>), -ge(>=), -eq(=), -ne(!=)
  - -contains, -notcontains, -like -notlike, -match, -notmatch
- 論理演算子、ビット演算子
  - -not, !, -and, -or, -xor, -bnot, -band, -bor, -bxor
- その他、置換演算子(-replace)、型演算子(-is, -as)、範囲演算子(..)、呼び出し演算子(&)、フォーマット演算子(-f)、リダイレクト演算子(>, >>)など

簡素化された配列の取り扱い

- 固定長配列を簡単に作成できる。

```
$arr1 = 1,3,5,7,9          #5個の要素を持つ配列
$arr2 = 1..10              #1～10までの要素を持つ配列
$arr3 = @(1)               #1要素の配列
$arr4 = @()                #空の配列

$arr2[3]                   #4番目の要素を返す
$arr2[5..8]                #6～9番目の要素を含んだ配列を返す
$arr2[0..3+7]              #1～4番目と8番目の要素を含んだ配列を返す
```

- 演算子を使った配列操作。

```
$arr2 -contains 2          #配列にある要素が含まれるかどうか(ここではTrue)
$arr2 -lt 3                #3より小さな要素を含んだ配列を返す

$arr2 += 50                #配列に要素を加える
$arr5 = $arr1 + $arr2      #配列を結合し新しい配列を作成
```

ハッシュテーブル

- ハッシュテーブル(連想配列)を容易に扱える。

```
$hash1 = @{}                          #空のハッシュ
$hash2 = @{a=1;b=2;c=3}               #3つの要素を持つハッシュ

$hash2.a                              #1を返す
$hash2["a"]                           #1を返す（上と同じ）

$hash2.d = 4                          #ハッシュに要素を追加
$hash2.Add("e",5)                     #ハッシュに要素を追加

#ハッシュの要素を列挙
foreach ($key in $hash2.Keys)
{
        $key + ":" + $hash2[$key]
}

$hash2.Contains("b")                  #キーの存在確認
```

各種制御構文

- C#ライクな各種制御構文を使って複雑なスクリプトが記述可能。

条件判別	
if/elseif/elseステートメント	switchステートメント

繰り返し	
forステートメント	foreachステートメント
do/untilステートメント	whileステートメント

エラー処理	
trapステートメント	throwステートメント

他の言語とは少し雰囲気の違うswitchステートメント(1)

- 基本はシンプル。"case"は書かない。

```
$a = 3
switch ($a) {
    1 {"これは1です。";break}
    2 {"これは2です。";break}
    3 {"これは3です。";break}
    4 {"これは4です。";break}
    default {"その他の数です。";break}
}
```

- パラメータを指定すると正規表現マッチなどが可能。

```
$a = "abcdefg2"
switch -regex ($a) {
    "¥d"        {"数値が含まれています。"}
    "[a-zA-Z]"  {"アルファベットが含まれています。"}
    "¥s"        {"空白文字が含まれています。"}
    default     {"その他の文字です。"}
}
```

- 他にもワイルドカードにマッチさせたり(-wildcard)、大文字小文字を区別する(-casesensitive)ことが可能。

他の言語とは少し雰囲気の違うswitchステートメント(2)

- 条件判定にスクリプトブロックを指定可能。

```
$a = 4

switch ($a) {
    {$_ -lt 5} {"$_ は5未満"}
    {$_ -gt 5} {"$_ は5より大きい"}
    {$_ -eq 5} {"$_ は5"}
}
```

- 配列に対して繰り返し処理をさせることも可能。

```
$a = 1..10

switch ($a) {
    {$_ -lt 5} {"$_ は5未満"}
    {$_ -gt 5} {"$_ は5より大きい"}
    {$_ -eq 5} {"$_ は5"}
}
```

function構文(1)

- 独自の関数を定義可能。
  - 基本はfunction 関数名を書くだけ。

```
function func1                                  # 関数の定義
{
    Write-Host "func1を実行しました。"          # 文字列を表示させる
}
func1                                           # 関数の呼び出し
```

  - 引数を指定する場合は、paramキーワードを用いる。

```
function RepeatWord                             # 関数の定義
{
    param([string]$word, [int]$count)           # パラメータ指定
    return $word * $count                       # 値を返却
}
RepeatWord "a" 5                                # 関数の呼び出し（パラメータ付加）
RepeatWord -count 15 -word あ                   # コマンドレットライクな指定も可
```

function構文(2)

- パイプラインを通して関数を呼び出す際、begin、process、endキーワードが使用可能。

```
function func2                              # 関数の定義
{
    begin                                   # begin節：最初だけ呼ばれる
    {
        "最初の1回呼ばれます"
    }
    process                                 # process節：毎回呼ばれる
    {
        "複数回呼ばれます　$input"          # $inputはパイプに渡されたオブジェクト
    }                                       #  （process節中では$_でもよい）
    end                                     # end節：最後だけ呼ばれる
    {
        "最後の1回呼ばれます"
    }
}

1..10 | func2                               # 配列をパイプラインを通じて関数に渡す
```

filter構文

- filter構文もfunction構文と並んで独自関数を記述するものだが、filter構文はパイプラインに渡されたオブジェクトをフィルタするのに用いる。

```
filter Select-ScriptFile                                    #filterの定義
{
    if(".ps1",".vbs",".js" -contains $_.Extension)  # 拡張子が特定の物の場合
    {
        return $_                                           # 入力をそのまま出力する
    }
}

Get-ChildItem | Select-ScriptFile                           # オブジェクトを入力する
```

- functionとの違いは、パイプラインに渡した配列を一度に処理するか(function)個別に処理するか(filter)。

基本構文を使ったスクリプトのデモ

DEMO2-2

.NET Frameworkのオブジェクトを作成する

- **New-Objectコマンドレットを用い、.NET Frameworkに含まれるクラスをインスタンス化**することができる。
  - **New-Object** [-typeName] クラスのフルネーム [[-argumentList] コンストラクタ(配列も可)]
    - 名前空間を含めたクラスのフルネームを指定するのが基本だが、「System.」は省略可能。

```
# オブジェクトの作成
$SmtpClient = New-Object System.Net.Mail.SmtpClient "smtp.example.com"

# プロパティの設定
$SmtpClient.Port = 25

# メソッドの実行
$SmtpClient.Send("from<from@example.com>","to<to@example.com>","件名","本文")
```

smtp.example.comというSMTPサーバーに接続してメールを送信

クラスのスタティックメンバ呼び出し

- [クラスまたは構造体のフルネーム]::メソッド名()
- [クラスまたは構造体のフルネーム]::プロパティ名
  で呼び出し可能。
  - New-Objectのときと同様「System.」は省略可。

```
# メソッドの実行
[System.DateTime]::IsLeapYear(2007)        # 指定年がうるう年か否か

#プロパティの参照
[System.DateTime]::Now                     # 現在時刻取得
```

- スタティックメンバの一覧を表示するには
  [クラスまたは構造体のフルネーム]|**Get-Member** -static
  とする。

型のキャスト

- 型のキャストを行う方法は二通りある。
  - [型名]キャストする変数またはリテラル

```
[DateTime]"2007/1/12"                #文字列（string）型をSystem.DateTime型に変換
```

    - 変換に失敗すると例外がスローされる。
  - -as演算子を用いる。

```
"2007/1/12" -as [DateTime]           #文字列（string）型をSystem.DateTime型に変換
```

    - 変換に失敗するとnullを返す。

COMオブジェクトの呼び出し

- **New-Objectコマンドレットを用い、COMオブジェクトをインスタンス化することができる。**
  - **New-Object [-comObject] ProgID [-strict]**

```
# COMオブジェクトの作成
$IE = New-Object -comObject InternetExplorer.Application

# プロパティの設定
$IE.Visible = $true

# メソッドの実行
$IE.Navigate("http://blogs.wankuma.com/mutaguchi/")
```

InternetExplorerを起動し、指定のサイトにアクセスする。

オブジェクトを使ったスクリプトのデモ

DEMO2-3