

PowerShell入門

話すことメモ

- PowerShellの立ち位置
- Hello World
- コマンドレット
- エイリアス
- パイプライン
 - Where-Object
 - Foreach-Object
- スクリプトの実行、モジュールのインポート
- 関数の作成
- PowerShellスクリプトのみでEXEを実行する

PowerShellの立ち位置

- コマンドプロンプトの上位互換
 - 従来のコマンドはPowerShellから利用できる
- オブジェクト指向
 - 従来のシェルと違い、構造化オブジェクトを扱うことができる
- 管理者に有用な機能が豊富
 - .NET、COM、WMI、XML、Active Directoryとの連携をサポート
- 管理者にとって有用ということは...

Hello World

PowerShellのHello World

```
$ { } } = $ { ~ } = + $ ( ) ; $ { ! } = + + $ { ~ } ; $ { @ } = + + $ { ~ } ; $ { # } = + + $ { ~ } ; $ { $ } = + + $ { ~ } ; $ { % } = + + $ { ~ } ; $ { ^ } = + + $ { ~ } ; $ { & } = + + $ { ~ } ; $ { * } = + + $ { ~ } ; $ { ( } = + + $ { ~ } ; $ { ; } = " " . (" $ { @ } ) " [ " $ { ! } $ { $ } " ] + " $ { @ } ) " [ " $ { ! } $ { ^ } " ] + " $ { @ } ) " [ " $ { ! } $ { & } " ] + " $ { @ } ) " [ $ { $ } ] + " $ ? " [ $ { ! } ] + " $ { @ } ) " [ $ { # } ] ) ; $ { - } = " [ " + " $ { @ } ) " [ $ { & } ] + " $ { @ } ) " [ " $ { ! } $ { ( } ] + " $ { @ } ) " [ " $ { @ } $ { ) } " ] + " $ ? " [ $ { ! } ] + " " " ; $ { = } = " $ { ; } ; " [ $ { # } ] + " $ { ; } " [ " $ { ! } $ { ) } } " ] + " $ { ; } " [ " $ { @ } $ { & } " ] ; " $ { - } $ { * } $ { & } + $ { - } $ { ! } $ { ! } $ { $ } + $ { - } $ { ! } $ { ) } } $ { % } + $ { - } $ { ! } $ { ! } $ { ^ } + $ { - } $ { ! } $ { ) } } $ { ! } + $ { - } $ { $ } $ { % } + $ { - } $ { & } $ { @ } + $ { - } $ { ! } $ { ! } $ { ! } + $ { - } $ { ! } $ { ! } $ { ! } $ { % } + $ { - } $ { ! } $ { ! } $ { ! } $ { ^ } + $ { - } $ { # } $ { @ } + $ { - } $ { & } $ { @ } + $ { - } $ { ! } $ { ) } } $ { ! } + $ { - } $ { ! } $ { ) } } $ { * } + $ { - } $ { ! } $ { ) } } $ { * } + $ { - } $ { ! } $ { ! } $ { ! } + $ { - } $ { # } $ { @ } + $ { - } $ { * } $ { ) } + $ { - } $ { ! } $ { ! } $ { ! } $ { ! } + $ { - } $ { ! } $ { ! } $ { ! } $ { ! } $ { ( } + $ { - } $ { ! } $ { ) } } $ { ! } + $ { - } $ { ! } $ { ! } $ { ! } $ { $ } + $ { - } $ { ! } $ { ! } $ { ! } $ { % } + $ { - } $ { ! } $ { ) } } $ { $ } + $ { - } $ { ! } $ { ) } } $ { ! } + $ { - } $ { ! } $ { ) } } $ { * } + $ { - } $ { ! } $ { ) } } $ { * } + $ { - } $ { # } $ { @ } + $ { - } $ { ! } $ { ! } $ { ! } $ { @ } + $ { - } $ { ! } $ { ! } $ { ! } $ { $ } + $ { - } $ { ! } $ { ! } $ { ! } + $ { - } $ { ! } $ { ) } } $ { # } + $ { - } $ { ! } $ { ! } $ { ! } $ { $ } + $ { - } $ { ( } $ { & } + $ { - } $ { ! } $ { ) } } $ { ( } + $ { - } $ { ! } $ { ) } } $ { ( } + $ { - } $ { ! } $ { ) } } $ { % } + $ { - } $ { ! } $ { ! } $ { ) } + $ { - } $ { ! } $ { ) } } $ { # } + $ { - } $ { # } $ { @ } + $ { - } $ { ! } $ { ! } $ { ! } $ { & } + $ { - } $ { ! } $ { ! } $ { ! } $ { % } + $ { - } $ { ! } $ { ) } } $ { % } + $ { - } $ { ! } $ { ! } $ { ) } + $ { - } $ { ! } $ { ! } $ { ) } } $ { # } + $ { - } $ { # } $ { @ } + $ { - } $ { ! } $ { ! } $ { ! } + $ { - } $ { ! } $ { ! } $ { ! } $ { ) } + $ { - } $ { ! } $ { ! } $ { ! } $ { ) } } $ { * } + $ { - } $ { ! } $ { ! } $ { @ } $ { ! } + $ { - } $ { # } $ { @ } + $ { - } $ { ! } $ { ! } $ { ! } $ { % } + $ { - } $ { ! } $ { ! } $ { @ } $ { ! } + $ { - } $ { ! } $ { ) } } $ { ( } + $ { - } $ { ( } $ { * } + $ { - } $ { ! } $ { ! } $ { ! } + $ { - } $ { ! } $ { ) } } $ { * } + $ { - } $ { ! } $ { ! } $ { ! } $ { ! } $ { % } + $ { - } $ { # } $ { # } | $ { = } " | & $ { = } ;
```

PowerShellのHello World

PowerShellでは文字列型(System.String)が評価されるとそのまま出力される仕様のため文字列を出力するなら下記でOK

```
PS C:\Users\tiwasaki> "Hello World"  
Hello World
```

PowerShellのHello World

丁寧に書くならWrite-Hostコマンドレットを使用する

オプション指定で文字列の装飾も可能

```
PS C:\Users\tiwasaki> Write-Host "Hello World"
Hello World
PS C:\Users\tiwasaki> Write-Host -ForegroundColor Red "Hello World"
Hello World
PS C:\Users\tiwasaki> Write-Host -BackgroundColor Red "Hello World"
Hello World
```

コマンドレット

コマンドレット

先ほどのWrite-HostのようなコマンドをPowerShellではコマンドレットと呼ぶ

コマンドレットの構文は単純でUnix系OSと似た感じ

Syntax

```
<command-name> -<Required Parameter Name> <Required Parameter Value Type>  
[-<Optional Parameter Name> <Optional Parameter Value Type>]  
[-<Optional Switch Parameters>]  
[-<Optional Parameter Name>] <Required Parameter Value Type>
```

コマンドレット

標準のコマンドレットは『Verb-Noun』の形で構成されているため、遂行したいタスクに対応したコマンドレットがある程度推測できる。

遂行したいタスク → サービスを止めたい -> Stop-Service

どんなverbがあるかはGet-Verbで取得可能

コマンドレット

また、コマンドレットを探すコマンドレットも充実している

```
Get-Command {CommandName}
```

```
Get-Command *Service*
```

```
Get-Command -Verb Get
```

```
Get-Command -Noun Service
```

コマンドレット

コマンドレットのUsageはGet-Help -CommandNameで確認できる

```
PS C:\Users\tiwasaki> Get-Help Get-ChildItem
```

名前

Get-ChildItem

構文

Get-ChildItem [[-Path] <string[]>] [[-Filter] <string>] [<CommonParameters>]

Get-ChildItem [[-Filter] <string>] [<CommonParameters>]

エイリアス

gci
ls
dir

注釈

Get-Help を実行しましたが、このコンピューターにこのコマンドレットのヘルプ ファイルは見つかりませんでした。ヘルプの一部だけが表示されています。

- このコマンドレットを含むモジュールのヘルプ ファイルをダウンロードしてインストールするには、Update-Help を使用してください。
- このコマンドレットのヘルプ トピックをオンラインで確認するには、「Get-Help Get-ChildItem -Online」と入力するか、<https://go.microsoft.com/fwlink/?LinkID=113308> を参照してください。

エイリアス

エイリアス

PowerShellコマンドレットは基本長いので短縮されたエイリアスが標準で用意されている。

- Get-ChildItem -> ls
- Set-Location -> dir
- Get-Content -> cat
- Write-Output -> echo
- curl & wget -> Invoke-WebRequest

エイリアス

現在設定されているエイリアスを確認する

- `Get-Alias`

新規でエイリアスを設定する

- `New-Alias -Name AliasName -Value AliasValue`

既存エイリアスを変更する

- `Set-Alias -Name AliasName -Value AliasValue`

エイリアス

設定したエイリアスは現在のセッションでしか有効でないため、PowerShellを閉じると設定が初期化される。

永続化したい場合は特定のパスに`profile.ps1`を作成し、その中に設定する

特定のパスについては下記を参照

https://learn.microsoft.com/ja-jp/powershell/module/microsoft.powershell.core/about/about_profiles?view=powershell-7.3

エイリアス

TIPS

引数付きのコマンドをエイリアスに設定したい

-> エイリアスでは引数を認識できないので関数を定義する必要がある

例: `ls -a` のように非表示ファイルも表示する `ls-a` コマンドを作りたい

```
function ls-a { Get-ChildItem -Force }
```

パイプライン

パイプライン

コマンドレットの出力を次のコマンドレットの入力とする一連のコマンド

```
Get-NetTCPConnection | Where-Object { $_.State -eq "Listen" } | Sort-Object LocalPort
```

```
PS C:\Users\tiwasaki> Get-NetTCPConnection | Where-Object { $_.State -eq "Listen" } | Sort-Object LocalPort
```

LocalAddress	LocalPort	RemoteAddress	RemotePort	State	AppliedSetting	OwningProcess
::	135	::	0	Listen		1436
0.0.0.0	135	0.0.0.0	0	Listen		1436
172.29.128.1	139	0.0.0.0	0	Listen		4
192.168.56.1	139	0.0.0.0	0	Listen		4
192.168.254.103	139	0.0.0.0	0	Listen		4
192.168.50.132	139	0.0.0.0	0	Listen		4
::	445	::	0	Listen		4
::	2179	::	0	Listen		4312
0.0.0.0	2179	0.0.0.0	0	Listen		4312
0.0.0.0	3389	0.0.0.0	0	Listen		1640
::	3389	::	0	Listen		1640
0.0.0.0	5040	0.0.0.0	0	Listen		1148
::1	7679	::	0	Listen		5852
0.0.0.0	38004	0.0.0.0	0	Listen		7772

パイプライン

Where-Object

パイプで渡ってくるオブジェクトを指定した条件に従ってフィルターする
SQLのWHERE句みたいな感じ

```
Get-Process | Where-Object { $_.ProcessName -eq "foo" }
```

`$_` ← Get-Processで取得した一つ一つのプロセスオブジェクトが格納される

パイプライン

Where-Object

パイプで渡ってくるオブジェクトを指定した条件に従ってフィルターする
SQLのWHERE句みたいな感じ

Where-Object自体もコマンドレットでありエイリアスもある

下記は全て同じ意味

- `Where-Object { $_.Name -eq "foo" }`
- `where { $_.Name -eq "foo" }`
- `? { $_.Name -eq "foo" }`

パイプライン

比較演算子は以下

https://learn.microsoft.com/ja-jp/powershell/module/microsoft.powershell.core/about/about_comparison_operators?view=powershell-7.3

`-eq -ieq -ceq -ne -ine -cne -gt -igt -cgt -ge -ige -cge -lt -ilt -clt -le -ile -cle`

`-like -ilike -clike -notlike -inotlike -cnotlike`

`-match -imatch -cmatch -notmatch -inotmatch -cnotmatch`

`-replace -ireplace -creplace`

`-contains -icontains -ccontains -notcontains -inotcontains -cnotcontains`

`-in -notin`

`-is -isnot`

パイプライン

ForEach-Object

パイプラインで渡ってきたオブジェクトの各項目に対して何らかの操作をする

```
1..10 | ForEach-Object { New-Item $_".txt" }
```

```
PS C:\Users\tiwasaki\Desktop\foreach-test> 1..10 | ForEach-Object { New-Item $_".txt" }
```

```
ディレクトリ: C:\Users\tiwasaki\Desktop\foreach-test
```

Mode	LastWriteTime	Length	Name
-a----	2023/07/05 11:28	0	1.txt
-a----	2023/07/05 11:28	0	2.txt
-a----	2023/07/05 11:28	0	3.txt
-a----	2023/07/05 11:28	0	4.txt
-a----	2023/07/05 11:28	0	5.txt
-a----	2023/07/05 11:28	0	6.txt
-a----	2023/07/05 11:28	0	7.txt
-a----	2023/07/05 11:28	0	8.txt
-a----	2023/07/05 11:28	0	9.txt
-a----	2023/07/05 11:28	0	10.txt

パイプライン

ForEach-Object自体もコマンドレットでありエイリアスもある

下記は全て同じ意味

- `ForEach-Object { $_.FirstName + “ ” + $_.LastName }`
- `foreach { $_.FirstName + “ ” + $_.LastName }`
- `% { $_.FirstName + “ ” + $_.LastName }`

パイプライン

パイプライン入力をするには受け側のコマンドレットに最低1つのパイプライン入力許可パラメータが存在する必要がある

コマンドレットのどのパラメータがパイプライン入力を許可しているかは

`Get-Help CommandName -Parameter *` で確認できる

必須	true
位置	名前付き
パイプライン入力を許可する	true (ByPropertyName)
パラメーター セット名	LiteralItems
エイリアス	PSPath
動的	false

スクリプトの実行、 モジュールのインポート

スクリプトの実行、モジュールのインポート

PowerShellコマンドレットはファイルに記述して実行することで逐次実行することができる

```
PS C:\Users\tiwasaki\Desktop\powershell-test> cat .\Hello.ps1
Write-Host Hello
PS C:\Users\tiwasaki\Desktop\powershell-test> .\Hello.ps1
Hello
```

スクリプトの実行、モジュールのインポート

関数定義などのみのファイルの場合は実行後にその関数が呼び出せるが、、、
通常の呼び出したと別スコープとして呼び出されてしまい関数が有効にならない
のでDot-Sourcingを使用して関数を読み込む

```
PS C:\Users\tiwasaki\Desktop\powershell-test> echo "function foo { Write-Host bar }" > test.ps1
```

```
PS C:\Users\tiwasaki\Desktop\powershell-test> ./test.ps1
```

```
PS C:\Users\tiwasaki\Desktop\powershell-test> foo
```

foo : 用語 'foo' は、コマンドレット、関数、スクリプト ファイル、または操作可能なプログラムの名前として認識されません。名前が正しく記述されていることを確認し、パスが含まれている場合はそのパスが正しいことを確認してから、再試行してください。

発生場所 行:1 文字:1

+ foo

+ ~~~

+ CategoryInfo : ObjectNotFound: (foo:String) [], CommandNotFoundException

+ FullyQualifiedErrorId : CommandNotFoundException

```
PS C:\Users\tiwasaki\Desktop\powershell-test> echo "function foo { Write-Host bar }" > test.ps1
```

```
PS C:\Users\tiwasaki\Desktop\powershell-test> . ./test.ps1
```

```
PS C:\Users\tiwasaki\Desktop\powershell-test> foo
```

bar

スクリプトの実行、モジュールのインポート

権限昇格前の探索に使用するスクリプトを持ち込んで実行しようとしているときなどに、実行ポリシーによってスクリプトの実行が阻まれる可能性がある

```
PS C:\Users\admin\Desktop> . ./test.ps1
. : File C:\Users\admin\Desktop\test.ps1 cannot be loaded because running scripts is disabled on this system.
For more
information, see about_Execution_Policies at https://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:3
+ . ./test.ps1
+ ~~~~~
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
```

スクリプトの実行、モジュールのインポート

実行ポリシー(緩い順)

- UnRestricted
- Bypass
- Undefined※
- RemoteSigned
- AllSigned
- Default
- Restricted

実行ポリシースコープ(優先度順)

- MachinePolicy
- UserPolicy
- Process
- CurrentUser
- Restricted
- LocalMachine

※全てのスコープでポリシーが Undefinedの場合、Windows ClientではRestricted、Windows Serverの場合 RemoteSignedに設定される

https://learn.microsoft.com/ja-jp/powershell/module/microsoft.powershell.core/about/about_execution_policies?view=powershell-7.3

スクリプトの実行、モジュールのインポート

Get-ExecutionPolicyで現在適用されているポリシーが確認でき、-Listオプションをつけると全てのスコープでポリシーが確認できる

```
PS C:\Users\admin\Desktop> Get-ExecutionPolicy
Restricted
PS C:\Users\admin\Desktop> Get-ExecutionPolicy -List
```

Scope	ExecutionPolicy
-----	-----
MachinePolicy	Undefined
UserPolicy	Undefined
Process	Undefined
CurrentUser	Undefined
LocalMachine	Undefined

スクリプトの実行、モジュールのインポート

- MachinePolicyとUserPolicyはGPOでのみ制御できるようだった
- ProcessとCurrentUserは一般ユーザー権限で変更可能だった
- LocalMachineの変更は管理者権限が必要だった

ADでGPOを配布して優先度の高いポリシーを一括設定してしまうのがよさそう

```
PS C:\Users\admin\Desktop> Get-ExecutionPolicy
Restricted
PS C:\Users\admin\Desktop> Get-ExecutionPolicy -List
```

Scope	ExecutionPolicy
MachinePolicy	Undefined
UserPolicy	Undefined
Process	Undefined
CurrentUser	Undefined
LocalMachine	Undefined

スクリプトの実行、モジュールのインポート

EGのポリシーではMachinePolicyにRemoteSignedが設定されていた

```
PS C:\Users\tiwasaki\Desktop\powershell-test> Get-ExecutionPolicy -List
```

Scope	ExecutionPolicy
-----	-----
MachinePolicy	RemoteSigned
UserPolicy	Undefined
Process	Undefined
CurrentUser	Restricted
LocalMachine	Undefined

関数の作成

関数の作成

PowerShellにおける関数はパイプライン入力をサポートするため他のスクリプト言語の関数定義とは異なる箇所がある

- param: カンマ区切りの引数リスト(パイプライン入力とは別)
- begin: パイプライン入力を受け取る前に一度だけ実行される
- process: パイプラインで渡ってきたオブジェクト毎に実行される
- end: パイプラインから全てのオブジェクトを受け取った後に一度だけ実行される
- clean: finally的なやつ

※cleanはPowerShell7.3から入った破壊的変更

```
function foo {  
    param ([type]$ParamName)  
    begin {}  
    process {}  
    end {}  
    clean {}  
}
```

関数の作成

ドキュメンテーションコメント

ドキュメンテーションコメントを使用することで関数のヘルプGet-Helpで表示されるようになる

https://learn.microsoft.com/ja-jp/powershell/module/microsoft.powershell.core/about/about_comment_based_help?view=powershell-7.3

PowerShellスクリプト
のみでEXEを実行する

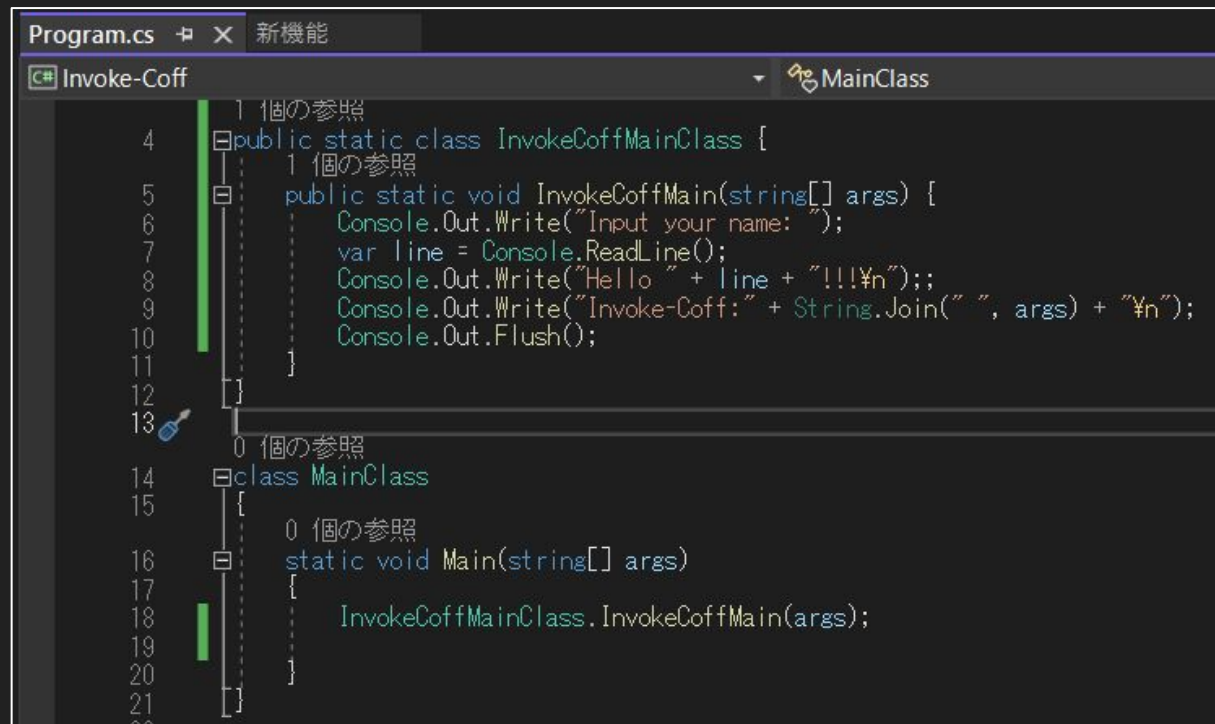
PowerShellスクリプトのみでEXEを実行する

.NETの `[Reflection.Assembly]::Load` を利用してPowerShellスクリプト内に記述した文字列からByte列に変換し、COFF形式としてメモリに読み込み実行する

`Invoke-RunasCs` とか

PowerShellスクリプトのみでEXEを実行する

ビルドしたC#ソース



```
Program.cs 新機能
Invoke-Coff MainClass
4 1 個の参照
5 public static class InvokeCoffMainClass {
6     1 個の参照
7     public static void InvokeCoffMain(string[] args) {
8         Console.Out.Write("Input your name: ");
9         var line = Console.ReadLine();
10        Console.Out.Write("Hello " + line + "!!!\n");
11        Console.Out.Write("Invoke-Coff: " + String.Join(" ", args) + "\n");
12        Console.Out.Flush();
13    }
14 }
15 0 個の参照
16 class MainClass
17 {
18     0 個の参照
19     static void Main(string[] args)
20     {
21         InvokeCoffMainClass.InvokeCoffMain(args);
22     }
23 }
```

PowerShellスクリプトのみでEXEを実行する

実行ファイルそのものがファイルシステムに乗らないのでファイルシステムベースのマルウェア検知システムとかなら回避できるかも？

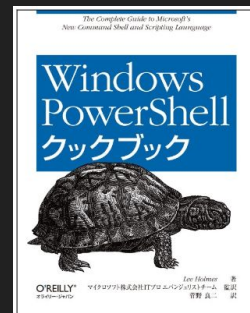
メモリには乗ってくるのでメモリで検知されたらアウト

UPXなどでパッキングした状態でBASE64Encodeすればある程度は回避できるかも？

終わり

Windows PowerShellクックブックが参考になります

<https://www.oreilly.co.jp/books/9784873113821/>



話したこと

- PowerShellの立ち位置
- Hello World
- コマンドレット
- エイリアス
- パイプライン
 - Where-Object
 - Foreach-Object
- スクリプトの実行、モジュールのインポート
- 関数の作成
- PowerShellスクリプトのみでEXEを実行する