PowerShell 真香

2019-03-02 | 2019-03-17 | linux | 0

> 我就是饿死, 死外面, 从外面跳下去, 也不会用 M$ 的垃圾 PowerShell !
>
> ...
>
> 真香!

NOTE: 这是一篇黑 bash 吹 PowerShell(以下简称 pwsh) 的文章, bash 死忠粉以及软黑请退散.

起因 (为什么要使用 pwsh)

因为用 bash 写脚本太痛苦了!

语法诡异, 内置功能弱地1B. zsh 好了那么一点, 然而还是很痛苦, 而且那惜字如金的命名风格导致我的脚本过段时间自己都不认得写的是啥了.

什么? 为什么不用 Python ? Python 倒是过段时间也能认得自己写的是啥, 可是写起来太麻烦了.

我需要一个写起来爽, 还看得懂我写了啥的 shell.

于是我就想起了 pwsh. (其实在用 Linux 之前学过一点, 毕竟 cmd 都亡了, 要紧跟时代 (谁知道我竟然叛逃 Linux 了

于是我就试用了一下, 然后发现: woc, 真香!

优点

语法简单

比 bash 不知道高到哪里去了

多么直观的语法啊, 麻麻再也不用担心我几个星期没写就忘记语法了

```
# if 语句
$n = Read-Host "请输入你的年龄"
if ($n -ge 18) {
    echo "是成年人"
} elseif ($n -ge 13) {
    echo "是青少年"
```

```powershell
} else {
    echo "是小屁孩"
}

# switch 语句
switch ($n) {
    {$_ -gt 18} { echo "是成年人"; break }
    18 { echo "是半步成年"; break }
    Default { echo "不是成年人"; break }
}

# for 语句
$sum = 0
for ($i = 0; $i -le 100; $i++) {
    $sum += $i
}
echo $sum

# while 循环, do while 差不多
$sum = 0
$i = 0
while ($i -le 100) {
    $sum += $i
    $i++
}
echo $sum

# Hash 表
$language = @{
    Name = "PowerShell"
    Company = "Microsoft"
}
echo "$($language.Name)"

# 定义函数
function pow($x, $y=2) {
    return [Math]::Pow($x, $y)
}
```

内置功能强大

pwsh 不需要借助外部命令就能完成大量操作.

> 此处应强调一下, 你愿意的话也可以在 pwsh 里 sed, grep, awk 走起, 写成 bash 味的 pwsh.
>
> 因为这些玩意儿不是 bash 的一部分, 是个 shell 都能调用.

> 你能用的我也能用, 我能用的你却不能用, 岂不美哉?
>
> 有些人可能会觉得这违反了 Unix 哲学, 很多违反 Unix 哲学的东西都很火, 并且干趴了遵循 Unix 哲学的同行 (

在这一点上我要点名批评 bash,

为啥 Linux 上需要 sed, awk, grep 等等工具? 因为 bash 它太 tm 弱鸡了.

刚用 Linux 的时候我觉得 bash 比它在 Windows 上的小兄弟 cmd 还是要强了不少的——cmd 特么连数组都没有, 只能靠变量延迟扩展这种神仙设定来模拟数组.
然而用了一段时间后我发现: nnd, 强个屁, 这破 bash 干啥事儿都要外部命令, 连 mv, cp 这些都是外部命令.

> 这主要是从 cmd 带过来的习惯——高性能批处理指南第一条就是少用外部命令 ~~(其实根本没有这个指南, 不过这话是对的, 除此之外还要少用 for /f %%i in ('') 和管道~~

比如说以空格为分隔符, 截取第 3 列

```batch
:: batch
for /f "tokens=3" %i in ("1 2 3 4 5") do echo %i

# bash
echo 1 2 3 4 5 | cut -d' ' -f3
echo 1 2 3 4 5 | awk '{print $3}'

# pwsh
# 首先 bash 的方法 pwsh 也能用
('1 2 3 4 5' -split ' ')[2]
```

你说 bash 你丢不丢人, 连 cmd 都能不借助外部命令自己处理

再来看正则表达式, bash 不用说了大家都知道的, cmd 也不用说了, findstr 那垃圾正则有和没有都一样

我们来看看 pwsh

```
> '<p>WOWOWOWO</p>' -match '<p>.*</p>'
True
> '<p>WOWOWOWO</p>' -replace '<p>(.*)</p>','$1'
WOWOWOWO
> '<div><div></div>' -match "<div[^>]*>[^<>]*(((?'Open'<div[^>]*>)[^<>]*)+((?'
> $Matches.Values
<div></div>
```

真是妙啊!

符合直觉的通配符

我曾遇到过一个需求, 根据通配符判断文件是否存在

判断文件是否存在我还是会的, bash 的写法 `[ -f ~/.zshrc ]`, pwsh 的写法 `Test-Path ~/.zshrc`

> 看, pwsh 的写法多清晰, 即使从没学过 pwsh 的人也能看出右边的命令做了什么, 然而从没学过 bash 的人估计想破头也看不懂左边的代码. 好歹搞个长命令啊, 比如 `[ -exists ~/.zshrc ]` 什么的

那加上通配符该怎么办呢?

这里又需要点名批评一下 bash, bash 里的通配符是由 shell 展开的. 是的, 是由 shell 展开的!!

乍一看好像也没什么问题? 让我来为你演示一下这个神奇特性

```
[/tmp]$ mkdir test
[/tmp]$ cd test
[/tmp/test]$ touch -- -f
[/tmp/test]$ *
bash: -f：未找到命令
[/tmp/test]$ [ * ~/.zshrc ] && echo 1
1
```

看懂了吗! 竟然 TM 有这种操作!! (想象一下你 `rm *` 的时候目录下有个叫 `-rf` 的文件). (我记得好像有一道 CTF 题就是用了这种神奇操作.)

了解这个神奇特性以后, 你应该知道, `[ -f ~/*.c ]` 这个代码是肯定行不通的.

该怎么办呢? 问了G娘以后得知是 `compgen -G "<glob-pattern>"`, 这个命令在有匹配时会输出匹配, 这样就能判断啦 ~
当然缺点就是你得 `>/dev/null` 屏蔽掉输出.

并且, 注意到了吗, 这个地方使用了双引号来避免通配符被提前展开.
这又带来了另一个问题!! `~` 这玩意儿也是由 shell 展开的!! 也就是说如果你这样写 `compgen -G "~/*"`, 是不会有任何输出的, 因为它期望在一个名为 `~` 的文件夹下寻找文件. 累了累了, 不管了

再看 pwsh 的解法——直接写就行了... `Test-Path "~/*.c"`, 毫无任何奇技淫巧, 可读性倍儿高.

类型

一开始学编程的时候, 我觉得类型这玩意儿好像也没啥用, 弱类型 && 动态类型真香,
后来学了 Rust, 我又觉得强类型 && 静态类型实在是太棒了! 错误就是应该尽早被发现!

pwsh, 作为给系统管理员用的脚本语言当然肯定不会是强类型 && 静态类型的, 那样太残忍了.

不过 pwsh 的类型系统比 bash 就不知道高到哪里去了.

基于 .Net 的 pwsh 默认就支持了大量类型 `[array],[bool],[byte],[char],[datetime],[decimal],[double],[guid],[hashtable],[int16],[int32],[int],[int64],[long],[nullable],[psobject],[regex],[sbyte].[scriptblock],[single],[float],[string],[switch],[timespan],[type],[uint16],[uint32],[uint64],[xml]`, 如果还嫌不够的话, .Net 里面还有大量好东西可以掏, 什么 HashSet 啊, List 啊, Queue 啊, Stack 啊, 应有尽有.

比 bash 不知道高到哪里去了! (强调

而且 pwsh 还有一个很棒的功能, 就是类型标注. 当然不是 Python 那样纯粹标一下给人看 (逃, 当然现在很多 IDE & lint 工具都能识别 type hint

pwsh 的类型标注可以固定某个变量的类型. 比如

```
[int]$n = 2233
$n = 233   # OK
$n = "asd" # ERROR!
$n = "233" # 比较艹蛋的一点是这样又可以, 隐式类型转换坑爹啊

[int[]]$list = @()
$list += 1   # OK
$list += "a" # ERROR!

function add([int]$a, [int]$b) {
    return $a + $b
}
add 1 2     # OK
add "a" "b" # ERROR!
```

注意到上面指定了函数参数的类型, 其实这还可以更棒

```
function add() {
    param(
        [ValidateNotNull]
        [int]
        $a,
        [ValidateNotNull]
```

```
        [int]
        $b
    )
    process {
        return $a + $b
    }
}
add $null 2 # ERROR!
```

这就是 pwsh 的 advanced_parameters

可读性高

这个其实和内置功能那一节的内容紧密相关, 正是因为很多功能都内置了, 语法才会统一.
不过我还是想单独拆一个标题出来. 因为 bash 脚本的可读性是真 tm 糟糕.

Linux 上的这些 shell, 充斥着大量匪夷所思的缩写&符号.

bash	pwsh
`[[ "$string" == *"$substring"* ]]`	`"$string".Contains("$substring")`
`${\#array[@]}`	`$array.Length`
`${string#substring}`	`$string -replace "^$substring"`
`${string%substring}`	`$string -replace "$substring$"`
`$@` OR `$*`	`$args`

可能以前存储空间很宝贵吧, 然而现在机械硬盘起步1T, 省这几个字符有啥用啊!!

而且 pwsh 的内部命令(一般称为 cmdlet), 非常有规律. 都是 动词+名词 的组合, 而且常用命令都有别名(alias), 允许你少打几个字

other/alias	pwsh
rm	Remove-Item
md	New-Item
cd	Set-Location

other/alias	pwsh
cp	Copy-Item
mv	Move-Item

看, pwsh 的命令是何等的规律! 总有人说啥 pwsh 的学习成本高, 不如学 xxx. 这我是不信的, pwsh 的这种一致性加上流行度, 学习 pwsh 远比你去学一个没多少人用的"用户友好"的 shell 要好.

而且 pwsh 对内部命令的补全非常棒棒, 像补全函数名, 补全参数什么的都是小 case, pwsh 甚至能根据管道前的命令来补全

举例来说, `xxx | ForEach-Object { $_. }` 这样的命令, 在 `.` 处按 TAB, pwsh 就会根据 XXX 的返回值类型来补全. (类型系统万岁!)

面向对象

这是一个非常棒的特点, pwsh 是一个面向对象的 shell. cmdlets 的输出是对象, 而不是字符串.

举例来说, 我想获得当前系统中内存占用超过 200 mb 的进程 (这里我又要点名批评一下 C++ 写的 telegram-desktop telegramdesktop/tdesktop#2464, ~~请立即使用 Rust~~

```
> Get-Process | Where-Object { $_.WS -gt 200mb }
# 使用别名的话可以写成 gps | ? { $_.WS -ge 200mb }

 NPM(K)    PM(M)      WS(M)     CPU(s)      Id  SI ProcessName
 ------    -----      -----     ------      --  -- -----------
      0     0.00     206.59     334.50    1033 032 plasmashell
      0     0.00     226.43     773.39    1027 977 kwin_x11
      0     0.00     226.46     135.83    8116 058 chromium
      0     0.00     236.60      97.72   14990 977 Typora
      0     0.00     392.42     394.97    6366 030 telegram-deskto
      0     0.00     402.94     878.92    1381 058 chromium
```

> 眼尖的小朋友可能会发现, "telegram-deskto" 好像有点奇怪.....默默甩 issue dotnet/corefx#34437

反观 bash, 这个时候只能上 awk 了

```
➤ ps aux | awk '$4 * 8192 / 100 > 200' # 为了让输出好看点我删掉了命令行参数
aloxaf    1027  4.9  2.9 3279432 232108 ?      Sl   11:04  14:04 /usr/bin/kwin
aloxaf    1033  2.1  2.6 1395968 212200 ?      Sl   11:04   6:06 /usr/bin/plas
aloxaf    1381  5.3  5.2 1423580 414932 ?      Sl   11:06  15:05 /usr/lib/chro
```

```
aloxaf    6366  3.0  5.0 1956732 404088 ?      Sl   11:50   7:09 /usr/bin/tele
aloxaf    8116  1.0  2.8 883064 225856 ?       Sl   11:59   2:28 /usr/lib/chro
aloxaf   14990  5.2  3.1 1792528 249992 ?      Sl   14:46   3:11 /usr/share/ty
```

如果再按照内存使用量从大到小排序呢? pwsh 再接个 `Sort-Object -Descending -Property WS` 就行了, bash 的话接个 `sort -rk 4`

有没有发现什么. bash 的可读性太 TM 糟糕了——充斥着魔术数字. 这个 4 究竟代表了什么? 从代码中看不出来, 输出以后更是 van 全看不出来!

为啥 bash 需要 sed, awk, grep 等工具? 因为它太 TM 弱了, 也因为大家都只输出字符串.
这玩意儿对人倒是友好, 但是对机器一点都不友好.
这设计实在是太糟糕了, 一旦想让机器来处理这些给人看的格式, 代码就会变成不是给人看的.
我也不是说大家一定要面向对象, 好歹大家一起定义一个数据交换格式, 这样岂不美哉?

举例来说, 大家都用 JSON 交换数据, 然后就可以这样写 `ps aux --export | where '_["mem"] * 8192 / 100 > 200' | to-table`, 这不是很好么?? 比字符串传来传去不知道高到哪里去了!!

缺点

垃圾管道

pwsh 的管道, 一方面非常棒, 传递的是对象, 由此搭配 `Select-Object`, `ForEach-Object` 这类命令能够实现令 bash 望尘莫及的行云流水的操作.

另一方面, 管道是真滴慢. 无数 pwsh 优化指南第一句肯定就是少用管道 (这点和 cmd 挺像的, cmd 的管道慢是因为每次管道都会开启一个新的 cmd 进程, 写批处理的时候为了性能很多时候都会选择重定向到文件再读入 (M$ 这是什么垃圾实现

而且, pwsh 的 `>` 其实是 `| Out-File` 的别名, 所以不要以为重定向就能提高速度.

那该怎么办呢? 内部命令好说, 一般有参数可以让你直接传对象过去, 然而都是外部命令就呵呵了.

距离来说 `seq 12345 | rg 12345` 这样一个简单的命令竟然花了 0.76 s

```
> Measure-Command -Expression {seq 12345 | rg 12345} | select TotalMillisecond

TotalMilliseconds
-----------------
          759.6631
```

反观 bash , 简直就是吊打!

```
➤ time seq 12345 | rg 12345
12345
seq 12345  0.00s user 0.00s system 84% cpu 0.002 total
rg 12345  0.00s user 0.00s system 92% cpu 0.005 total
```

心累啊, 大概 M$ 一开始觉得反正 Windows 上又没啥工具给你调用...结果怎么突然就跨平台了呢?

与 native 程序交互

pwsh 是面向对象的, 这是好的. cmdlets 的返回值是一个对象, 这是 pwsh 自己可以保证的. 然而外部命令呢?

显然, 外部命令的输出只能当成字符串来处理了...

等等? 你确定是字符串??

`cat /usr/bin/ls` ——这玩意儿会是一个字符串吗? 不, 不可能的.

然而 pwsh 真的就是按我上面所说的来处理的...

这就导致了你让 pwsh 处理外部命令给出的任何它不能完美解释为字符串的玩意儿时的迷のbug, 包括二进制数据, 非 UTF-8 编码.

大部分情况下, 我们可以通过 `-o FILE` 输出到文件, 然后再用指定参数的 `Get-Content` 读入来解决, 然而这太不爽了, 而且有些命令根本没有这个功能! 这个时候就要点名批评 `xclip` , 它倒是有 `-o` , 然而这只会把数据输出到 stdout. 当我在 pwsh 里试图用 xclip 从剪贴板里面导出一张图片时, bug 就这样发生了(亏我能立马排查出是管道的问题...

上面两个问题的解决方案

1. sh -c 'command1 | command2 > file', 简单粗暴, 然而不雅观
2. Start-Process 然后手动读 raw byte, 这个方法我至今还没成功过
3. Use-RawPipeline(Windows) 和 Use-PosixPipeline(Linux), 大概是目前情况下最好的解决方案了, 绕过了 pwsh 的管道传递数据, 不过后者尚处于 develop 阶段 (快 PR

启动斯必得

pwsh 的启动速度真的太慢了, 和 zsh 差 20 倍啊.

```
➤ time zsh -c 'exit'
zsh -c 'exit'  0.01s user 0.00s system 94% cpu 0.018 total
➤ time pwsh -nop -c 'exit'
pwsh -nop -c 'exit'  0.38s user 0.05s system 118% cpu 0.360 total
```

而且这可是没有任何配置的 pwsh 啊!!

我本来试了试 oh-my-posh, 发现颜值还真挺高的, 然而这样一来启动速度就到了秒级. 吓得我赶紧卸载掉了.
所以虽然我大赞了一通 pwsh, 我日常交互式的话应该还是不会使用它, 只是写脚本用用.

对了, 此处还要强调一下, pwsh 即使是执行脚本也会加载 $PROFILE 里的内容, 除非指定了 `-nop` (-NoProfile) 参数. 所以如果你的 pwsh 用了重量级配置, 脚本的 shebang 最好加上 -nop 参数

linux # powershell