



PowerShell v3 入门级教程

PowerShell v3 入门级教程

PowerShell v3 是一个 Windows 任务自动化的框架，它由一个命令行 shell 和内置在这个 .NET 框架上的编程语言组成。PowerShell v3 在 Windows Server 2012 中装载了 Windows Management Framework 3.0。本期《PowerShell v3 入门级教程》旨在让大家充分了解 PowerShell v3。包括 PowerShell v3 新功能、PowerShell v3 的使用、管理技术等。是一部由浅入深、循序渐进的 PowerShell v3 实用教程。

PowerShell v3 新功能介绍

PowerShell v3 真正的美妙之处在于它既不是图形用户界面，也不是命令行界面 (CLI)，而是两者均有兼顾。快来感受 PowerShell v3 五大顶级功能吧！

- ❖ 新型图形界面驾到 PowerShell v3 光芒闪耀
- ❖ 感受 PowerShell v3 五大顶级功能
- ❖ 远程处理在 Microsoft PowerShell v3 中的增强

PowerShell v3 功能使用

PowerShell v3 内部增加了一些新功能或一些功能的增强。如何使用这些新功能开展自己的工作？这里给大家解释常见的 PowerShell v3 使用问题。

- ❖ 如何启用 PowerShell Remoting 之 One-to-one Remoting
- ❖ 如何启用 PowerShell Remoting 之 One-to-many Remoting

PowerShell v3 管理应用

对 IT 管理员来说，PowerShell v3 令他们爱不释手。他们可以用 PowerShell v3 来管理 Windows 桌面、管理 Hyper-V 等等。

- ❖ 如何用 PowerShell 管理 Windows 桌面？
- ❖ 如何使用 PowerShell 管理微软 Hyper-V？
- ❖ 使用 Windows PowerShell 快速安装 WSUS 更新

其他 PowerShell 应用

在其他工作环境中，PowerShell 也时刻发挥着其重要作用。比如如何使用 PowerShell 创建 HTML 条形图等等。

- ❖ 如何用 PowerShell 创建 HTML 条形图？
- ❖ Powershell：为自动化收集 NIC PCI 总线信息
- ❖ 如何用 PowerShell 脚本节省 SharePoint 备份时间？

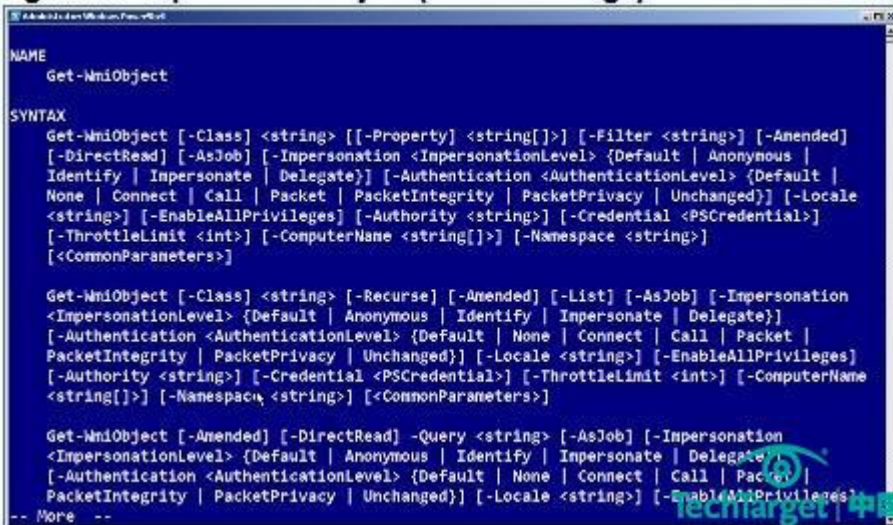
新型图形界面驾到 PowerShell v3 光芒闪耀

当你面对着一个命令行窗口进行工作时，屏幕中有文本，还有光标在闪烁，仅此而已，没有任何的工具栏和标签导航栏。这在你心目中并不是“图形用户界面(GUI)”。

Windows PowerShell v3 是一个特例，它在 Windows 8 开发者预览版中首次出现，现在可以作为微软社区技术预览（Windows 管理框架 3.0 的一部分）安装在 Windows 7 中。但 PowerShell 真正的美妙之处在于它既不是图形用户界面，也不是命令行界面(CLI)，而是两者均有兼顾。这在 v3 中得到了充分的体现。

更好的帮助功能

如果你对命令 Get-WmiObject 的语法不是很了解，可以输入 Help Get-WmiObject 查看帮助信息（如图 1），但这些信息是否有帮助将完全取决于用户对它们的理解。



```

NAME
    Get-WmiObject

SYNTAX
    Get-WmiObject [-Class] <string> [[-Property] <string[]>] [-Filter <string>] [-Anended]
    [-DirectRead] [-AsJob] [-Impersonation <ImpersonationLevel> {Default | Anonymous |
    Identify | Impersonate | Delegate}] [-Authentication <AuthenticationLevel> {Default |
    None | Connect | Call | Packet | PacketIntegrity | PacketPrivacy | Unchanged}] [-Locale
    <string>] [-EnableAllPrivileges] [-Authority <string>] [-Credential <PSCredential>]
    [-ThrottleLimit <int>] [-ComputerName <string[]>] [-Namespace <string>]
    [<CommonParameters>]

    Get-WmiObject [-Class] <string> [-Recurse] [-Anended] [-List] [-AsJob] [-Impersonation
    <ImpersonationLevel> {Default | Anonymous | Identify | Impersonate | Delegate}]
    [-Authentication <AuthenticationLevel> {Default | None | Connect | Call | Packet |
    PacketIntegrity | PacketPrivacy | Unchanged}] [-Locale <string>] [-EnableAllPrivileges]
    [-Authority <string>] [-Credential <PSCredential>] [-ThrottleLimit <int>] [-ComputerName
    <string[]>] [-Namespace <string>] [<CommonParameters>]

    Get-WmiObject [-Anended] [-DirectRead] -Query <string> [-AsJob] [-Impersonation
    <ImpersonationLevel> {Default | Anonymous | Identify | Impersonate | Delegate}]
    [-Authentication <AuthenticationLevel> {Default | None | Connect | Call | Packet |
    PacketIntegrity | PacketPrivacy | Unchanged}] [-Locale <string>] [-EnableAllPrivileges]
    [-Authority <string>] [-Credential <PSCredential>] [-ThrottleLimit <int>] [-ComputerName
    <string[]>] [-Namespace <string>] [<CommonParameters>]

-- More --
  
```

图 1: 命令 Help Get-WmiObject

如果运行命令 Show-Command Get-WmiObject，将会弹出一个基于图形界面的提示框。每个参数集都会对应在自己的选项卡上，每个参数也会分拆到自己的行上。所需参数在其名称旁都有一个简单的“*”及对应的复选框。这使用起来非常容易，当您完成命令时，您可以运行命令或将其复制到剪贴板，使它更易于粘贴到其它地方或更方便地与其它命令结合使用。图 2 就是这条命令的界面。



图 2: Show-Command Get-WmiObject 界面

更加丰富的输出

[PowerShell](#) v3 仍然可以方便地使用命令 Out-GridView (如图 3)，它接收其它命令的输出，并构造一个基于图形界面的表格。可以通过单击列标题进行排序，更可通过使用内置的“加载项”按钮动态地筛选显示的内容，此视图可以显示许多不同类型输出而生成的图形界面。



图 3: Out-GridView 命令界面

更便捷的编辑功能

PowerShell 团队在新的 PowerShell 版本中对 GUI 方面的最大改进就是与脚本环境集成（图 4）。在编写代码时，系统可以智能感知并给出想要的提醒。系统甚至会显示精简版的命令帮助提示，提醒用户命令的功能以及如何使用。

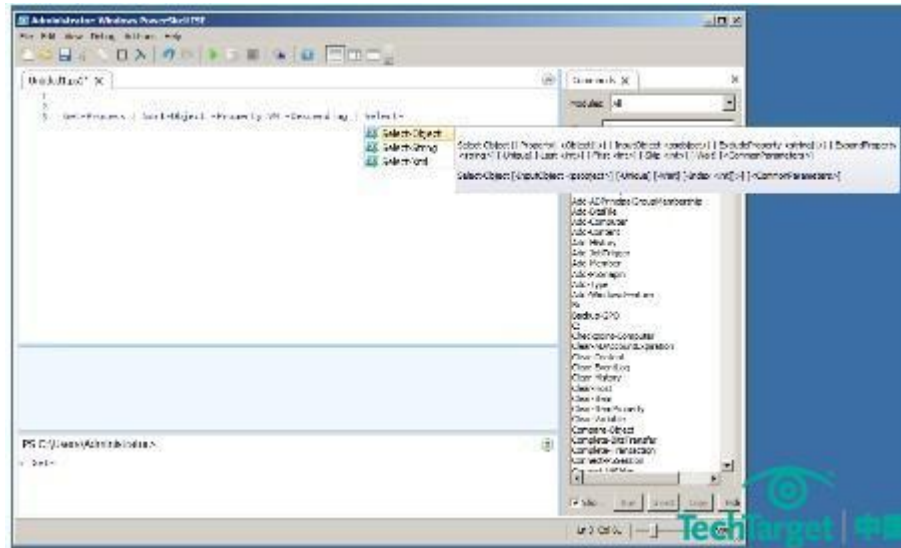


图 4：脚本环境的集成

在右侧的命令浏览器中提供了加载到 shell 中的完整命令列表，并可以通过附加模块（如活动目录）来筛选相应的命令。这里给出的那些命令行环境下最核心的命令实际上就是用户需要的命令，这个新的浏览器能够让用户更容易的发现和使用他们需要的命令。

更受欢迎的网络访问功能

不仅只有这些，在 v3 中还有更多的图形界面，这里介绍的是网络页面。新的 PowerShell 网络访问界面是一个安装在服务器（安装 IIS 是这个特性的前提条件）上的可选功能。配置完成后，您将获得一个完全基于网页的命令行窗口，它更适用于像 iPhone 这样的小型设备。在屏幕底部的文本框中输入命令，并点击“执行”，命令的输出将显示在一个可滚动的类似于命令行界面的窗口中。

想象一下，我们可以在数据中心的“桥头堡”服务器上安装 PowerShell 网页访问应用（PWA）。它支持多个并发连接，因此可以允许多个管理员同时使用它来远程访问数据中心的资源、运行脚本、执行命令，甚至完成他们的工作。在键入命令和参数时，PWA 甚至支持 Tab 功能，这在管理员使用像 Android 手机这样的小型设备上敲入一长串命令时尤其受欢迎！在 PWA 上你甚至可以像使用正常 PowerShell 控制台那样获得前几次执行的命令。

PWA 支持所有常见的 IIS 身份验证机制：摘要式验证、基本验证、集成验证等方式，它还可以使用 HTTPS 连接来保证安全。总的来说，PWA 既可以给予管理员有效的访问，又可以保证环境的安全。

它不只是 CLI！也不只是 GUI！而是它们的集合体

好吧，也许它不会带来一片光明，但 PowerShell v3 已经开始打破传统的命令行 shell 模式。在适当的地方使用图形界面可以更容易学习 Powershell，v3 也继续多个管理模块中提供更强大的管理能力。

感受 PowerShell v3 五大顶级功能

Windows PowerShell v3 即将发布，它将让人难以忽视。[Windows Server 8](#) 将在 [PowerShell](#) 上打造管理的大部分，也提供 GUI 管理选项，以及命令行自动化。shell 的 Version 3 引入了一些相当重要的新功能，下面我们介绍这五大新功能。

更好的远程处理

PowerShell 远程变得更重要，已经逐渐成为在网络上进行管理通信的主要渠道。越来越多的 GUI 管理控制台将依赖远程，因此对微软来说加强这个很重要。现在能够断开远程会话，让你稍后能从同个或不同的计算机重新连接到相同的会话。现在，如果你的客户端计算机崩溃的话，v3 的社区技术预览版不能断开会话。相反，会话会永久关闭。所以这与远程桌面完全不同，远程桌面会话能在客户端崩溃时配置并打开会话。

工作流

这个功能非常了不起。

本质上，PowerShell 新的工作流构建能让你写入与功能类似的东西，使用 PowerShell 翻译你的命令和脚本代码到 Windows 工作流技术 WWF 进程中。WWF 然后能管理整个任务，包括修复网络故障与重启计算机等。它是编排长期运行的、复杂的、多步骤任务的更有效更可靠的一种方式。如果这个功能与下一个版本的 System Center Orchestrator 集成，不用惊讶，如果成为其他服务器产品的标配功能也不用奇怪。

可更新的帮助

PowerShell 老是与帮助文件中的错误斗争。倒不是说错误有太多，而是微软修复起来很困难，因为基本上需要发布一个操作系统补丁。可没有人想给帮助文件打补丁。基于 TechNet 站点的在线帮助的存在减轻了这个问题的，但杯水车薪。在 v3 中，帮助文件能按需更新，从你喜欢的任何微软服务器那都可下载新的 XML 文件。所以微软就能根据找到的问题进行错误修复，不需要操作系统包或补丁了。

预定任务 (Scheduled Job)

PowerShell v2 引入了 job，遵循的是 job 随着时间扩展的理念。在 v3 中，新型 job 即 scheduled job 能被创建并按计划运行，或者相应某个事件。这与

Windows 的 Task Scheduler 的区别只是其中一小点，不过最终你能从 PowerShell 中获得这个功能。不似现在仍然存在的 v2 job，scheduled job 存于 PowerShell 之外，意味着就算 PowerShell 不运行了他们仍然运行。

更好的发现

关于命令行 shell 的一个困难部分在于如何使用它。PowerShell 的帮助系统很有用，需要提供你想知道的命令的名字，并提供命令所在的插件名字，并记得加载附件到内存中。这有太多告警。而 PowerShell v3 在搜索命令时，包含所有安装模块的所有命令，想运行没有装载的命令？那么 shell 会在幕后暗暗装载给你。不过这只能在那些存储在列于 PSModulePath 环境变量中的文件路径中的模块有用，不过只要你想包含额外路径，就可以在任何时候修改变量。

额外功能：CIM

PowerShell 与 Windows 管理规范 WMI 运作很好，WMI 是微软的一项技术，或多或少建立在标准的通用信息模块 CIM 上。在 PowerShell v3 中，WMI cmdlet 发挥余热，但它们加入到新的 CIM cmdlet 集中。起初，功能看起来似乎有重叠，但是有原因的：CIM cmdlet 使用 WS-MAN，这个协议位于 PowerShell 的 Remoting 功能，微软管理功能的新标准的后面。WMI 使用被微软正式弃用的 DCOM，意味着不会再开发新功能，但可一直使用。CIM 是未来的方向，不仅有对已知 WMI 的额外开发，而且在未来还可跨平台管理。

总的说来，PowerShell v3 让人期待。现在你可以立即获得 CTP No. 2 并开始感受它的魅力。

远程处理在 Microsoft PowerShell v3 中的增强

远程处理是 PowerShell v2 中最值得期待的特性。就是它使 PowerShell 到达了一个新的高度。以下摘录所描述的远程控制的形式，是基于由 Don Jones、Richard Siddaway 和 Jeffery Hicks 合著的《PowerShell in Depth》（Manning 出版社出版）中的第 10 章。

如果想要阅读全书，Manning 出版社向 TechTarget 的读者们提供 6 折优惠。购买时你可以在 manning.com 网站上使用 12pidtt 的促销代码。如果你已经购买了纸质的书，同时就可以免费拥有书的电子版。

远程处理是 PowerShell v2 和 Management Framework Core v2 中引入的主要的新技术之一。在 v3 版本中，微软继续在这项重要的基础性技术中进行开发和挖掘。

实际上，PowerShell v2 提供了两种连接到远程计算机的方式：

1. 带有自己的 `-computerName` 参数的 Cmdlets。它利用专有通信协议，最常用的是 DCOM 或 RPC，一般只限于单个任务。它并不使用 PowerShell 的远程处理（也有些例外）。
2. 使用专门的远程处理技术的 Cmdlets：跟有 `-PSSession` 和其它参的 `Invoke-Command`。

这里讨论的主要侧重于第二种方式。它的好处之一是它的任何 cmdlet，不管是否带有 `computerName` 参数，都可以用来进行远程处理。

那么到底什么是远程处理？很简单，它是将一个或多个命令通过网络发送到一个或多个远程计算机的能力。远程计算机使用自己本地的处理资源（也就是该命令必须在远程计算机上存在并加载）运行收到的命令。像所有的 PowerShell 命令一样，命令的结果是一些对象，PowerShell 将它们序列化成 XML 并通过网络传送到命令发起的计算机，在这里这些 XML 又被反序列化成对象并放入 Powershell 的管道中。在整个命令执行过程中，序列化/反序列化的过程至关重要，因为它提供了一种方法可以轻松地复杂的数据结构转换成易于网络传输的文本形式。不要过多的考虑序列化，虽然它并不比使用 `Export-CliXML` 和 `Import-CliXML` 命令来处理运行结果复杂。其实，远程处理所带来的好处主要是可以通过网络获取数据。

远程处理的一些术语让很多人感到困惑，所以让我们先来了解一下它们。

1. **WS-MAN** 是远程处理使用的网络协议。它是一种管理用途的 Web 服务，它或多或少也是一个行业标准协议。虽然它的使用还不是很普遍，但在非 Windows 平台上也可以看到它的影子。WS-MAN 使用的也是 HTTP，跟你使用 Web 浏览器从服务器获取网页使用的协议完全一样。

2. **Windows 远程处理 (WinRM)**，是一个使用 WS-MAN 协议处理连接通讯和身份验证的 Microsoft 服务。WinRM 实际上是用来为任意数量的应用程序提供通信服务的；它并非仅限于 PowerShell。WinRM 接收信息并将信息标记为特定的应用程序，如 PowerShell，WinRM 在收到任何关于此应用程序的反馈或者需要发回的结果时，它就会按照标记进行分类处理。

3. **远程处理 (Remoting)** 是一个用来描述 PowerShell 调用 WinRM 的术语。因此，你只能使用 PowerShell 来进行远程处理，当然，其它应用程序也有它们自己针对 WinRM 的特定用法。

PowerShell v3 中的新功能之一是一套通用信息模型 (CIM) 的 cmdlets。尽管 Windows 管理规范 (WMI) 的 cmdlets 自 PowerShell v1 以来就一直存在，但是随着时间推移，CIM 的 cmdlets 最终会取代传统 WMI 的 cmdlets。现在，WMI 和 CIM 的 cmdlets 在 Powershell V3 中是共存的，而且有很多重叠功能。它们都是利用相同的 WMI 数据资料库；两者之间最主要的区别之一就是它们通过网络进行通信的方式。CIM 使用的是 WinRM，而 WMI 使用的则是远程过程调用 (RPC)。CIM 的 cmdlets 不使用远程处理——它们有自己的 WinRM 调用方式。我们拿这个进行举例只是为了更好的理解这些术语。通常情况下你不必担心它们之间的区别，但在故障排查时，你一定要知道谁正在调用什么。

现在可以通过深挖一些特定的使用细节来解释更多的术语：

1. **端点 (Endpoint)** 是 WinRM 中的一个特定的配置项。端点代表的是 WinRM 能为之接收通信的特定应用程序，通常都是一组设定来决定端点的行为方式。单个应用程序在多个端点有不同的设置是完全可能的，像 Powershell 就是这样的应用。每个端点可能用于不同的目的，可能会有不同的安全、网络设置等等与之关联。

2. **侦听器 (Listener)** 是另一个 WinRM 的配置项，代表服务接受入站网络通信的能力。一个侦听器配置有一个 TCP 端口号，可以接受来自一个或多个 IP 地址的流量等。侦听器也可以用来侦听 HTTP 或 HTTPS 的连接请求，如果你想使用这两种协议，那么就需要安装有两个侦听器。

在接下来的几个章节中，我们将逐个介绍[远程处理](#)的安装和使用全过程，这会特别涵盖“易用场景”，也就是你的计算机和远程计算机都在同一活动目录域中。

如何启用 PowerShell Remoting 之 One-to-one Remoting

Remoting 是大家对于 PowerShell v2 期待已久的功能。因此，PowerShell 功能达到了新的高度。下面内容节选自 Manning 出版社，介绍了如何启用 remoting。选自第十章 PowerShell 深度解析，由 Don Jones、Richard Siddaway 与 Jeffery Hicks 撰写。点击[这里](#)查看 [PowerShell v3](#) 的改进。

Remoting 应该在任何能够接收连接的机器上启用，包括运行 Windows 操作系统的服务器或客户端版本的计算机。设置 remoting 最简单的方式是运行 Enable-PSRemoting。这个命令执行以下几个任务：

1. 启动（或重启）WinRM 服务
2. 设置 WinRM 服务器以后自动运行
3. 创建 WinRM 监听器，用于所有本地 IP 地址端口 5985 上的 HTTP 流量监控

4. 为 WinRM 监听器创建 Windows 防火墙例外。注意，如果网卡配置成“Public”类型，这种在 Windows 客户端版本上的配置将失败，因为防火墙拒绝在这些卡上创建例外。如果不成功，将网卡类型更改为如“Work”或“Private”，然后再次运行 Enable-PSRemoting。或者，如果你有一些公共网卡，添加 SkipNetworkProfileCheck 参数启动 PSRemoting。这样做能成功创建防火墙例外，只允许计算机本地子网的远程处理流量通过。

这个命令也设置以下四个端口中的一个或几个：

1. PowerShell 32-bit
2. PowerShell 64-bit
3. PowerShell Server Manager Workflow
4. PowerShell Workflow

图 1 列出了一些例子和端点配置。在 32 位机器上，这个端点被称为 PowerShell 而不是 PowerShell32。

	PowerShell Version	PowerShell 32 bit	PowerShell 64bit	Server manager	PowerShell workflow
Windows Server 2008 R2	2	Y	Y	Y	
Windows 7 64 bit	3	Y	Y		Y
Windows 8 32 bit client	3	Y			Y
Windows 8 Server	3	Y	Y	Y	Y
Windows 7 client 32 bit standalone	2	Y			

图 1. 端点配置实例

命令运行的时候会提醒你好几次，确认回复“Y” for “Yes”，便于每个步骤完全正确地运行。

One-to-one Remoting

使用 remoting 最直接的方式是叫做 one-to-one remoting，这让你从本质上在远程计算机上提出交互式 PowerShell 提示。如果远程机器上启用了 remoting，这最简单。

```
PS C:\> enter-pssession -ComputerName Win8
```

```
[Win8]: PS C:\Users\Administrator\Documents>
```

注意，如果你想这样做，在计算机上启用 remoting，只需要将本地主机作为计算机名使用。你的计算机将被“远程控制”，但你会获得完整的远程体验。

注意 PowerShell 提示的变化，包括你现在所连接的计算机名。在这里，就如同你站在物理计算机面前，可运行远程计算机包含的任何命令。下面是几点注意事项：

1. 默认下，当 PowerShell prompt 包括任何计算机名的时候，你不能执行任何初始化 Remoting 连接的命令。

2. 不能执行任何启动图形应用的命令。如果这样做，命令将冻结，点击 Ctrl+C 终止命令并重新获得控制权。

3. 不能执行任何本来就拥有过自己的 [shell 命令](#)，如 nslookup or netsh。
4. 如果执行策略允许，那你可以在远程机器上运行脚本。
5. 你没有连到交互式桌面会话，你的连接将被定义为“网络登录”，虽然你能连接到远程机器上的文件共享。由于连接的类型，Windows 不会执行 profile 脚本，即使你连接到远程机器上的 profile 主文件夹。
6. 你所做的在连接到同个机器上的用户是不透明的，即使他们交互登录到桌面控制台上。换句话说，你不能运行相同的应用，使得其跳到正在登录的用户前面。
7. 你必须指定计算机名字，因为它会出现在 Active Directory 或本地受信任的主机列表上。你不能使用 IP 地址或 DNS CNAME 别名。

用完远程机器，那就运行 Exit-PSSession。这将回到本地提示符，关闭到远程机器的连接，释放远程机器的资源。如果关闭 PowerShell 窗口，这个也会自动发生。

```
[Win8]: PS C:\Users\Administrator\Documents> Exit-PSSession
```

```
PS C:\>
```

我们使用 Enter-PSSession 的方式将一直与远程机器的 PowerShell 终端连接。在 64 位操作系统上，将是 64 位版本的 PowerShell。稍后，我们将介绍如何连接到其他端点（请记得 Enable-PSRemoting 总共能创建四个端点。）

如何启用 PowerShell Remoting 之 One-to-many Remoting

上篇文章《[如何启用 PowerShell Remoting 之 One-to-one Remoting](#)》中我们讲解了如何使用 One-to-one remoting 的方式使用 PowerShell remoting。这里是 PowerShell remoting 使用指南的下半部分，继续讲解 One-to-many remoting 方式。

One-to-many Remoting

这是一项强大的技术，真正体现了远程处理的价值。你主要做的就是向远程计算机上传一个命令（或者一些列命令）。远程计算机单独执行这些命令，序列化为 XML 并将结果反馈给你。你的 PowerShell 副本会将这些 XML 反序列化为对象，将其放入流水线。例如，假设我们想从两台不同的计算机上获得所有以字母“s”开头的进程的列表：

```
PS C:\> invoke-command -ScriptBlock { Get-Process -name s* } -  
computername
```

```
localhost,win8
```

```
Handles NPM(K) PM(K) WS(K)
```

```
VM(M) CPU(s) Id ProcessN PSCompu  
ame terName
```

```
-----  
217 11 3200 7080 33 1.23 496 services win8
```

```
50 3 304 980 5 0.13 248 smss win8
```

```
315 16 2880 8372 46 0.03 12 spoolsv win8
```

```
472 36 8908 11540 60 0.31 348 svchost win8
```

```
306 12 2088 7428 36 0.19 600 svchost win8
```

```
295 15 2372 5384 29 0.61 636 svchost win8
```

```
380 15 17368 19428 55 0.56 728 svchost win8
1080 41 12740 25456 120 2.19 764 svchost win8
347 19 3892 8812 93 0.03 788 svchost win8
614 52 13820 18220 1129 2.28 924 svchost win8
45 4 508 2320 13 0.02 1248 svchost win8
211 18 9228 8408 1118 0.05 1296 svchost win8
71 6 804 3540 28 0.00 1728 svchost win8
2090 0 120 292 3 10.59 4 System win8
217 11 3200 7080 33 1.23 496 services loca...
50 3 304 980 5 0.13 248 smss loca...
315 16 2880 8372 46 0.03 12 spoolsv loca...
469 36 8856 11524 59 0.31 348 svchost loca...
306 12 2088 7428 36 0.19 600 svchost loca...
295 15 2372 5384 29 0.61 636 svchost loca...
380 15 17368 19428 55 0.56 728 svchost loca...
1080 41 12740 25456 120 2.19 764 svchost loca...
347 19 3892 8812 93 0.03 788 svchost loca...
607 49 13756 18132 1129 2.28 924 svchost loca...
45 4 508 2320 13 0.02 1248 svchost loca...
211 18 9228 8408 1118 0.05 1296 svchost loca...
```

```
71 6 804 3540 28 0.00 1728 svchost loca...
```

```
2089 0 120 292 3 10.59 4 System loca...
```

这个命令是 `Invoke-Command`。 `-ScriptBlock` 参数接收这些你想要传递给远程机器上的命令（多个命令用分号隔开）； `-ComputerName` 参数确认机器名字。另外，一个脚本块对象可以创建更长的命令：

```
$sb = {Get-Process -Name s*}
```

```
Invoke-Command -ComputerName localhost,win8 -ScriptBlock $sb
```

和 `Enter-PSSession` 一样，你必须指定计算机名字，因为它会出现在 `Active Directory` 或本地受信任的主机列表上。你不能使用 IP 地址或 DNS CNAME 别名。

注意到输出结果有什么有趣的地方了吗？输出结果包含一个叫做 `PSComputerName` 的列，包括了每个结果行来源计算机的名称，方便分离、排序、分组并组织结果。这个属性是通过 PowerShell 添加到传入的结果中；如果你不想在输出结果中看到这个属性，就将 `-HideComputerName` 参数添加到 `Invoke-Command` 中。这个属性仍然存在（可以用来排序等等），但是默认不在输出结果中显示。

同 `Enter-PSSession` 一样，`Invoke-Command` 会用到远程机器上的默认 PowerShell 终端，在 64 位操作系统上，将是 [64 位版本的 PowerShell](#)。

默认的，`Invoke-Command` 只能一次对话 32 个计算机。这样做需要它来保持与之对话的每台远程计算机将 PowerShell 实例放在内存中；32 个是微软努力追赶的数字，并且在各种情况下能够良好运行。如果你指定的计算机多于 32 个，那么多出来的只能排队等待，并且 `Invoke-Command` 完成头 32 个以后才开始工作。你可以使用命令的 `-ThrottleLimit` 参数来改变并行数量，记住，数量越大，你的电脑上的工作负载就越大，而远程计算机上没有额外的负载。

远程处理说明

从远程计算机发送到你的电脑上的数据是打包过的，以便于在网络中传输。我们曾提到过的序列化和反序列化可以完成这个工作——但是会损失一些功能。例如，`Get-Service` 产生的对象的类型：

```
PS C:\> get-service | get-member
```

TypeName: System.ServiceProcess.ServiceController

Name MemberType Definition

Name AliasProperty Name = ServiceName

RequiredServices AliasProperty RequiredServices = ServicesDepe...

Disposed Event System.EventHandler Disposed(Sy...

Close Method System.Void Close()

Continue Method System.Void Continue()

CreateObjRef Method System.Runtime.Remoting.ObjRef ...

Dispose Method System.Void Dispose()

Equals Method bool Equals(System.Object obj)

ExecuteCommand Method System.Void ExecuteCommand(int ...

GetHashCode Method int GetHashCode()

GetLifetimeService Method System.Object GetLifetimeService()

GetType Method type GetType()

InitializeLifetimeService Method System.Object InitializeLifetim...

Pause Method System.Void Pause()

Refresh Method System.Void Refresh()

Start Method System.Void Start(), System.Voi...

Stop Method System.Void Stop()

```
WaitForStatus Method System.Void WaitForStatus(Syste...  
  
CanPauseAndContinue Property bool CanPauseAndContinue {get;}  
  
CanShutdown Property bool CanShutdown {get;}  
  
CanStop Property bool CanStop {get;}  
  
Container Property System.ComponentModel.IContainer...  
  
DependentServices Property System.ServiceProcess.ServiceCo...  
  
DisplayName Property string DisplayName {get;set;}  
  
MachineName Property string MachineName {get;set;}  
  
ServiceHandle Property System.Runtime.InteropServices....  
  
ServiceName Property string ServiceName {get;set;}  
  
ServicesDependedOn Property System.ServiceProcess.ServiceCo...  
  
ServiceType Property System.ServiceProcess.ServiceTy...  
  
Site Property System.ComponentModel.ISite Sit...  
  
Status Property System.ServiceProcess.ServiceCo...  
  
ToString ScriptMethod System.Object ToString();
```

正如你看到的，这些对象的成员包含一些属性，可以让你停止或者暂停等等。现在，试想通过远程处理恢复来自远程机器上的相同类型的对象：

```
PS C:\> invoke-command -ComputerName win8 -ScriptBlock { Get-  
Service } |  
  
>> Get-Member  
  
>>
```

TypeName: Deserialized.System.ServiceProcess.ServiceController

Name	MemberType	Definition
------	------------	------------

-----	-----	-----
-------	-------	-------

ToString	Method	string ToString(), string ToString(str...
----------	--------	-------------------------------------------

Name	NoteProperty	System.String Name=AeLookupSvc
------	--------------	--------------------------------

PSComputerName	NoteProperty	System.String PSComputerName=win8
----------------	--------------	-----------------------------------

PSShowComputerName	NoteProperty	System.Boolean PSShowComputerName=True
--------------------	--------------	-------------------------------------------

RequiredServices	NoteProperty	
------------------	--------------	--

Deserialized.System.ServiceProcess.Ser...

RunspaceId	NoteProperty	System.Guid RunspaceId=00e784f7-6c27-4...
------------	--------------	-------------------------------------------

CanPauseAndContinue	Property	System.Boolean {get;set;}
---------------------	----------	---------------------------

CanShutdown	Property	System.Boolean {get;set;}
-------------	----------	---------------------------

CanStop	Property	System.Boolean {get;set;}
---------	----------	---------------------------

Container	Property	{get;set;}
-----------	----------	------------

DependentServices	Property	Deserialized.System.ServiceProcess.Ser...
-------------------	----------	-------------------------------------------

DisplayName	Property	System.String {get;set;}
-------------	----------	--------------------------

MachineName	Property	System.String {get;set;}
-------------	----------	--------------------------

ServiceHandle	Property	System.String {get;set;}
---------------	----------	--------------------------

ServiceName	Property	System.String {get;set;}
-------------	----------	--------------------------

ServicesDependedOn	Property	Deserialized.System.ServiceProcess.Ser...
--------------------	----------	-------------------------------------------

```
ServiceType Property System.String {get;set;}
```

```
Site Property {get;set;}
```

```
Status Property System.String {get;set;}
```

方法（除了通用的 ToString() 方法）不见了，这是因为你正在关注一个反序列化类型的对象（就在输出顶部的 TypeName 里）。本质上，你得到一个静态版本的只读对象。

这并不一定是，因为序列化和方法的移除不会发生，直到远程命令结束执行，并且输出是打包的。这个对象在远程计算机上仍然是“活的”，所以你只需要简单地在远程计算机上启动、停止、暂停等等。换句话说，你需要发起的任何“行为”需要变成发送至远程计算机执行命令的一部分。

如何用 PowerShell 管理 Windows 桌面？

当微软在 2003 年首次引入 [Windows PowerShell](#) 时，很多 IT 管理员认为 Windows PowerShell 只是执行脚本任务以及管理 Windows 服务器的另一种方式。但是随着时间的推移，Windows PowerShell 已经成为了用于管理、监控，采用脚本处理不同类型软硬件的候选工具。众多厂商已经将 PowerShell 嵌入到他们的产品当中，而且 Windows PowerShell 已经成为 Windows 7 核心的桌面管理平台。

PowerShell 的第一个版本并不能执行脚本也不能从中央工作站或服务器查询远程计算机。然而，默认安装在 Windows 7 以及 [Windows Server 2008 R2](#) 上的 PowerShell 的最新版本 Windows 远程管理 (WinRM) 采用“单一管理平台”架构进行集中式管理。（请注意：最新版的 PowerShell 现在已经支持 Windows XP 和 Vista，可以从官方网站下载。）

尽管有些企业可能至少有三分之一的产品的功能可以使用 PowerShell 实现，但是多数产品需要安装代理。但是，PowerShell 是操作系统内置的而且是一种脚本语言，因此即使和其他产品相比，PowerShell 也可能是一款功能强大的资产。

使用 PowerShell 进行桌面管理

整个管理控制的第一步是确保桌面在运行最新版本的 PowerShell 而且已经启用了 WinRM。

为启用 WinRM，需要以管理员身份在本地计算机的 PowerShell 提示符下执行如下命令：Enable-PSRemoting -force。

该命令将会开启一个恰当的防火墙端口用于和中央管理工作站进行通信。工作站现在已经可以执行远程 PowerShell 命令并进行查询了。

除此之外，您应该以本地管理员身份输入如下命令设置运行脚本的安全性：Set-ExecutionPolicy Unrestricted。

当然，取决于对安全性的要求，您可能会选择一个不同的脚本执行安全级别，比如 RemoteSigned。这一安全级别确保了在交互式会话中运行的所有脚本在执行时不会出错。通过在 PowerShell 提示符下运行 help Set-ExecutionPolicy -detailed 命令可以更多的了解执行策略的设置及其更多的分支。

借助 PowerShell 远程调用，中央工作站可以通过三种方法与远程计算机进行通信。第一种方法允许查询一个或多个计算机；另两种方法更多的是一对一会话。“Invoke-Command”方式是最为常用的一种方法。“Interactive”方式与通过 SSH 或 Telnet 访问远程计算机类似，“Implicit”方式将远程 PowerShell 会话引入到了集中会话当中。

使用内置的 PowerShell cmdlets

最新版的 PowerShell 包括了 30 多种向工作站发送远程命令的 cmdlets。几乎所有内置的 cmdlets 都能接受 - ComputerName 参数，向启用 WinRM 的所有桌面发送远程命令。

例如，`Get-EventLog - Logname Application - EntryType Error - ComputerName mypc` 命令检索包含来自计算机名为 mypc 的错误事件的应用事件日志记录。您可以扩展 - ComputerName 参数以包含多台计算机，比如 - ComputerName mypc, suzipc, tomcp。

您也可以使用 PowerShell 提供的脚本语言功能将参数转变为动态变量。然后就可以运行成百上千个桌面的远程命令了。

我经常使用的 cmdlet 是 `Get-Counter`，该命令允许您近乎实时地查看远程计算机的性能指标。将该命令应用于远程桌面的扩展列表或者通过动态变量查看组织中所有计算机的计数器。试着在 PowerShell 提示符下输入 `help get-counter - full` 命令查找当前环境下可用的语法。

另外，可以在远程计算机上执行清单功能。对于软件清单来说，可以使用 `Get-WMIObject` cmdlet 查询所有已安装 MSI 软件的 WMI 类。对于不是通过 MSI 安装程序进行安装的软件，可以查询所有软件记录的注册表。微软官方网站上的一篇 TechNet 文章是您构建软件清单工具的一个很棒的资源。

对于硬件清单来说，您应该使用另一个 WMI 查询指定硬件类。通过这篇 PowerShellPro 文章获取该脚本，这个脚本我已经使用过几次。

帮助编写 PowerShell 脚本的免费工具

对于生活在脚本世界的管理员来说，使用命令行下的 cmdlets 是很自然的事儿。但是对于没有太多命令行接口（CLI）经验的管理员或者想避免非常长的“单行方式”的管理员来说—Quest 软件公司提供的免费工具 PowerGUI 可能会非常有帮助。

PowerGUI 是 PowerShell 的前端图形用户界面。其最佳特性之一就是使用 PowerPacks，PowerPacks 由脚本专家所创建并编译为单个文件。引入这些脚本后，PowerGUI 将为您键入 PowerShell 命令。您可以在 ScriptEditor 中看到实际的脚本并使用这些脚本创建您自己的脚本。

一旦精通了用于远程桌面的 PowerShell 环境而且在编写脚本时变得自信，我推荐您看一下由 Sapien 开发的 PrimalForms 工具，可以使用该工具将您的脚本合并到图形用户界面当中。使用 PrimalForms，您可以为您自己或者其他人在命令提示符下感到不舒服的用户放置一个图形化前端。例如我创建的一个脚本的 GUI 前端已经成为了帮助桌面管理员的主要工具。可以在 NTPRO.NL 查看很棒的使用 PrimalForms 创建 [PowerShell GUI](#) 的例子。

网络上有大量用于 PowerShell 以及 PowerShell 脚本的资源。在创建您自己的脚本前一定要把这些资源找出来。大多数情况下，已经有人写了一个脚本恰好是您想要的，或者您可以找到能够进行定制脚脚本。

如何使用 PowerShell 管理微软 Hyper-V?

许多管理员喜欢使用 [PowerShell](#) 来自动执行用户创建和文件夹权限管理这类组件功能，但是，[虚拟化](#) 技术也可以通过命令行管理，包括微软 Hyper-V。

虽然有多种方法可以用 PowerShell 来管理 [Hyper-V](#)，但本文将重点介绍如何免费使用 Windows 管理规范 (WMI) 脚本(来自 CodePlex 的开源工具)的方法。

在使用 [WMI](#) 脚本来管理 Hyper-V 之前，了解哪些类可用很重要。微软列出了大量的类。虽然相当完整，但他们不一定易于使用，并且总是不直观。因此，使用 WMI 来管理 Hyper-V 不适合心理承受能力弱的人。

使用 PowerShell 管理 Hyper-V 的比较流行方法之一是使用针对 Hyper-V (PSHyperV) 的 PowerShell 管理库。这是由 James O' Neil 所写的免费且开源的 CodePlex 项目。这是迄今为止最好的选择。它提供一个完整 cmdlet 集给管理员使用，可以处理从虚拟机存储管理到网络管理的所有事情。让我们来了解其中的一些：

Get-VM——返回一个 Hyper-V 服务器上所有的虚拟机（见图 1）。

```
[8]PS> #
[9]PS> # Lots going on here. Lets just look at a few
[10]PS> #
[11]PS> # Lets get the VMs
[12]PS> #
[13]PS> Get-VM | ft -AutoSize
```

Host	VMElementName	State	Up-Time (mS)	Owner
CORE	ADLDS1	Running	19934940	
CORE	XenDesktop	Running	19965066	
CORE	Win2k8R2VM	Running	329047	
CORE	VistaKids	Running	19867031	
CORE	Win7x86	Running	19889945	
CORE	CoreWeb	Running	19965102	

图 1: Get-VM 命令

下面的代码展示了 Get-VM 命令：

```
Function Get-VM
{# .ExternalHelp MAML-VM.XML
param(
```

```
[parameter(ValueFromPipeLine = $true)]
[ValidateNotNullOrEmpty()][Alias("VMName")]
$Name = "%",
[parameter()][ValidateNotNullOrEmpty()]
$Server = ".", #May need to look for VM(s) on Multiple servers
[Switch]$Suspended,
[switch]$Running,
[switch]$Stopped
)
Process {
# In case people are used to the * as a wildcard...
if ($Name.count -gt 1) {[Void]$PSBoundParameters.Remove("Name")}
; $Name | ForEach-object {Get-VM -Name $_ @PSBoundParameters}}
if ($name -is [String]) {
$Name = $Name.Replace("*", "%")
# Note in VI the test was for caption like "Virtual%" which
did not work in languages other than English.
# Thanks to Ronald Beekelaar - we now test for a processID ,
the host has a null process ID, stopped VMs have an ID of 0.
$WQL = "SELECT * FROM MSVM_ComputerSystem WHERE ElementName
LIKE '$Name' AND ProcessID >= 0"
if ($Running -or $Stopped -or $Suspended) {
$state = ""
if ($Running) {$State += " or enabledState = " +
[int][VMState]::Running }
if ($Stopped) {$State += " or enabledState = " +
[int][VMState]::Stopped }
if ($Suspended) {$State += " or enabledState = " +
[int][VMState]::Suspended }
$state = $state.substring(4)
$WQL += " AND ($state)"
}
Get-WmiObject -computername $Server -Namespace $HyperVNamespace -
Query $WQL | Add-Member -MemberType ALIASPROPERTY -Name
"VMElementName" -Value "ElementName" -PassThru
}
elseif ($name.__class) {
Switch ($name.__class) {
```

```
"Msvm_ComputerSystem" {$Name}
"Msvm_VirtualSystemSettingData" {get-wmiobject -
computername $Name.__SERVER -namespace $HyperVNamespace -Query
"associators of {$(($name.__path))} where
resultclass=Msvm_ComputerSystem"}
Default get-wmiobject -
computername $Name.__SERVER -namespace $HyperVNamespace -Query
"associators of {$(($Name.__path))} where
resultclass=Msvm_VirtualSystemSettingData" /
ForEach-Object
{$_ .getRelated("Msvm_ComputerSystem")} | Select-object -unique }
}
}
}
```

如您所见，这段代码包含了 WMI 基本类和 helper 逻辑并报告了结果。

Get-VMSwitch——返回所有在 Hyper-V 服务器上的虚拟交换（见图 2）。

```
[28] PS> #
[29] PS> # Finally lets look at the Virtual Networks
[30] PS> #
[31] PS> Get-VMSwitch
```

Name	Learnable	Status	Addresses
VMNet - Virtual Network	1024	[OK]	

图 2: Get-VMSwitch 命令

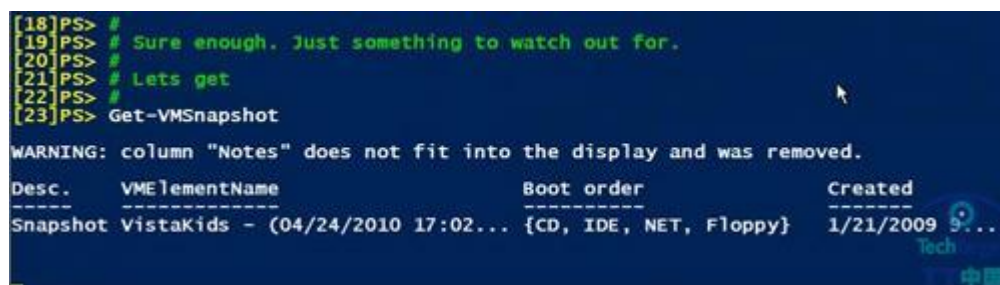
下面的代码展示了 Get-VMSwitch 的命令：

```
Function Get-VMSwitch
{# .ExternalHelp MAML-VMNetwork.XML
param(
[parameter(ValueFromPipeline = $true)][Alias("Name")]
[String]$VirtualSwitchName="",
[parameter()][ValidateNotNullOrEmpty()]
$Server = "." #Can query multiple servers for switches
)
process {
```



```
$VirtualSwitchName=$VirtualSwitchName.replace("*","%")
Get-WmiObject -computerName $server -Namespace $HyperVNamespace
-query "Select * From MsVM_VirtualSwitch Where elementname like
'$VirtualSwitchname' "
}
}
```

Get-VMSnapshot——提供所有在 Hyper-V 服务器上的快照（见图 3）。



```
[18]PS> #
[19]PS> # Sure enough. Just something to watch out for.
[20]PS> #
[21]PS> # Lets get
[22]PS> #
[23]PS> Get-VMSnapshot

WARNING: column "Notes" does not fit into the display and was removed.
Desc.      VMElementName      Boot order      Created
-----
Snapshot VistaKids - (04/24/2010 17:02... {CD, IDE, NET, Floppy} 1/21/2009 9...
```

图 3: Get-VMSnapshot 命令

下面的语句展示了 Get-VMSnapshot 命令:

```
Function Get-VMSnapshot
{# .ExternalHelp MAML-VMSnapshot.XML
Param(
[parameter(Position=0 , ValueFromPipeline = $true)]
$VM = "",
[String]$Name="",
[parameter()][ValidateNotNullOrEmpty()]
$Server=".",
[Switch]$Current,
[Switch]$Newest,
[Switch]$Root
)
process{
if ($VM -is [String]) {$VM=(Get-VM -Name $VM -Server $server) }
if ($VM.count -gt 1) {[Void]$PSBoundParameters.Remove("VM") ; $VM /
ForEach-object { Get-VMSnapshot -VM $_ @PSBoundParameters}}
if ($vm.__CLASS -eq 'Msvm_ComputerSystem') {
if ($current) {Get-wmiobject -computerNam $vm.__server -
Namespace $HyperVNamespace -q "associators of {$( $vm.path)} where
```



```
assocClass=MSvm_PreviousSettingData"}  
else {$Snaps=Get-WmiObject -computerName $vm.__server -Namespace  
$HyperVNamespace -Query "Select * From MsVM_VirtualSystemSettingData  
Where systemName='$($VM.name)' and  
instanceID <> 'Microsoft:$($VM.name)' and elementName like '$name' "  
if ($newest) {$Snaps | sort-object -property  
creationTime | select-object -last 1 }  
elseif ($root) {$snaps | where-object {$_.parent -eq  
$null} }  
else {$snaps}  
}  
}  
}  
}
```

可以从 CodePlex 的网站上找到 PSHyperV 的多种附加功能来帮助管理员执行查找、操作和配置 hypervisor 的不同的组件等相关任务。

编写 WMI 包装器和使用 PSHyperV，只是管理员用 PowerShell 来管理 Hyper-V 的一些方式。请注意，PSHyperV 的最新版本并不是完整的版本，因此，它不像其他软件那么稳定。

使用 Windows PowerShell 快速安装 WSUS 更新

每个 Windows 管理员的职业生涯中很有意义的一件事是命令行自动化的功能完全实现了。

之前，像 [PowerShell](#) 和 VBScript 这类的工具也许不常见。你也许能看到它们在完成小型任务上的价值，但是它们在创建自动化上花费的时间远远大于节省的时间。

总有一天，你会发现一个小小的脚本是如何将你的生活变得更加美好。回到 2007 年，脚本对我来说是很小但功能很强大的 WSUS 黑客。厌烦了每个月都要在漫长的夜晚等待 WSUS 的调度器给服务器打补丁，我最终选择了脚本“你现在的补丁” [VBScript](#)。

它要花上许多天，并且做大量的网络搜索，但是我的努力最终以自己命名为“我的 WSUS Big Red Button”的切实可行的方案结束。双击 VBScript，任何 Windows 计算机都会立刻扫描更新，下载并安装更新，必要的话重启计算机。

更好的是，这个脚本（或者更具体的说，Windows Update Agent）尊重 WSUS 应用配置，不管是手动或通过组策略。因此，任何计算机先前的部分 WSUS 基础设施将只安装有认可标志的更新。

Red Button Mark II, PowerShell 版本

时代在变，脚本语言也在变。VBScript 是长时间形成的手工艺品，现在被更强大的 PowerShell 所代替。因此，似乎是时间该将我的“立即安装 WSUS 更新” Big Red Button 替代掉 PowerShell 了。

下面是代码。远远短于 VBScript，Big Red Button 扫描 Windows 系统，下载并安装任何需要的更新，并依已安装的补丁需求进行重启。

```
#定义更新标准
$Criteria = "IsInstalled=0 and Type='Software'"

#搜索相关更新
$Searcher = New-Object -ComObject Microsoft.Update.Searcher
$SearchResult = $Searcher.Search($Criteria).Updates
```

```
#下载更新
$Session = New-Object -ComObject Microsoft.Update.Session
$Downloader = $Session.CreateUpdateDownloader()
$Downloader.Updates = $SearchResult
$Downloader.Download()

#安装更新
$Installer = New-Object -ComObject Microsoft.Update.Installer
$Installer.Updates = $SearchResult
$Result = $Installer.Install()

#重启，如果安装更新时需要的话
If ($Result.rebootRequired) { shutdown.exe /t 0 /r }
```

对这个“入门”脚本需要解释一下的是，它是我目前使用过的精华产品之一。这个脚本和它需要的同样最小。这句话的意思是，在 Windows 电脑上执行它，这台机器将搜索任何相关的更新，下载这些更新（不论是配置的 WSUS 服务器或是微软的在线服务器），安装它们，并根据需求进行重启。

我特意把这个脚本以最小的形式给你，给你一个机会来扩大对它的使用。我最初发表的 VBScript 是曾经发布的第一个解决这个特定问题的工具，现在在网上搜索“使用 PowerShell 安装 WSUS 更新”会带来更大的选择范围。那些的很大部分已经脱离通知、日志记录、收发邮件的结果文件和所有其他细节，使脚本像这些一样有用。它们掩盖了真正完成的工作。

脚本的第一块提供识别那些你想要安装的更新标准。我列出了几个 \$Criteria 例子，但是你可以在 MSDN 上的文档的帮助下添加自己的。

第二块指导 Windows Update Agent 为本地电脑搜索缺失的更新。第三块利用存储在变量 \$SearchResult 中的结果，开始更新下载。之后这些更新会在第四块进行安装。第五和最后一块查询安装进程，确认并强制进行必要的重启。

因为本地的 Windows Update Agent 不管是手动或是通过组策略，将不更改配置。运行这个将只下载那些你已经批准了的更新并安装在你的 WSUS 控制台。在开始之前，首先在个别机器上启动这个脚本。如果一台机器没有本地的 WSUS 服务配置，Windows Update Agent 将脱离微软互联网服务器查询微软认为适当（和符合你自己添加的标准）的补丁。

你可以往 PowerShell 脚本上添加很多东西，像数据采集和报告、报告邮件和各式各样的 if/then 语句和将一切都联系在一起的验证。

即使你之前从来没有使用过脚本，这样小的自动化提供了一个契机来赚回你的宝贵时间。希望有了它你可以消除又一个令人厌烦的枯燥的工作，让你轻松成为一个更高效的 Windows 管理员。

如何用 PowerShell 创建 HTML 条形图？

很多时候我们可以使用已有的工具实现很多实用的功能，只是我们并没有想到这个点上而已。这几天我一直在研究如何将 PowerShell 控制台当作图形工具使用。当然，有人马上就会想知道如何将它应用到 HTML 上。这里我就给出一个在 HTML 页面中创建彩色条形图的脚本 demo。

下面是 demo 代码：

```
#requires -version 2.0

Param (
    [string[]]$computers=($env:computername),
    [string]$Path="driverreport.htm"
)

$Title="Drive Report"

#embed a stylesheet in the html header
$head = @"
<style>
body { background-color:#FFFFCC;
font-family:Tahoma;
font-size:12pt; }
td, th { border:1px solid #000033;
border-collapse:collapse; }
th { color:white;
background-color:#000033; }
table, tr, td, th { padding: 0px; margin: 0px }
table { margin-left:10px; }
</style>
<Title>$Title</Title>
<br>
"@

#define an array for html fragments
$fragments=@()
```

```
#get the drive data
$data=get-wmiobject -Class Win32_logicaldisk -filter "drivetype=3" -
computer $computers

#group data by computername
$groups=$Data | Group-Object -Property SystemName

#this is the graph character
[string]$g=[char]9608

#create html fragments for each computer
#iterate through each group object

ForEach ($computer in $groups) {

$fragments+="<H2>$($computer.Name)</H2>"

#define a collection of drives from the group object
$Drives=$computer.group

#create an html fragment
$html=$drives | Select @{Name="Drive";Expression={$_.DeviceID}},
@{Name="SizeGB";Expression={$_.Size/1GB -as [int]}}},
@{Name="UsedGB";Expression={"{0:N2}" -f (($_.Size -
$_.Freespace)/1GB) }},
@{Name="FreeGB";Expression={"{0:N2}" -f ($_.FreeSpace/1GB) }},
@{Name="Usage";Expression={
$UsedPer= (($_.Size - $_.Freespace)/$_.Size)*100
$UsedGraph=$g * ($UsedPer/2)
$FreeGraph=$g* ((100-$UsedPer)/2)
#I'm using place holders for the < and > characters
"xopenFont color=Redxclose{0}xopen/FontxclohexopenFont
Color=Greenxclose{1}xopen/fontxclose" -f

$usedGraph, $FreeGraph
}} | ConvertTo-Html -Fragment

#replace the tag place holders. It is a hack but it works.
$html=$html -replace "xopen","<"
```

```
$html=$html -replace "xclose",">"

#add to fragments
$Fragments+=$html

#insert a return between each computer
$fragments+="<br>"

} #foreach computer

#add a footer
$footer=("<br><I>Report run {0} by {1}\{2}<I>" -f (Get-Date -
displayhint date),$env:userdomain,$env:username)
$fragments+=$footer

#write the result to a file
ConvertTo-Html -head $head -body $fragments | Out-File $Path
```

这里的关键点是我是从一个 HTML 片断的集合中创建了最终的 HTML 文件。我认为这个脚本中的大部分都应该都是自解释的。这个脚本的核心部分为计算机收集驱动器，选择定制的属性并将其改变成 HTML 片断。

讲技巧的部分是 Expression scriptblock 中“使用”的数值，我计算了要使用的磁盘空间的数值及其中免费空间的数值。然后我创建了一行代码，它是我的图形字符（[CHAR]9608）乘以一个数值。这个数值是相应比例除以 2，因为我想把用过的图形放在免费图形的右边。我希望用过的图形是红色的，免费的行则是绿色的。我采用快捷方式并使用字体标签。但是我不可以使用 < and > 字符，因为 Convertto-HTML 会翻译它们，所以我选用一个占位符来代替。

```
"xopenFont color=Redxclose{0}xopen/FontxclosexopenFont
Color=Greenxclose{1}xopen/fontxclose" -f $usedGraph,
$FreeGraph
```

我需要做的就是从 HTML 输出中替换占位符。

```
#replace the tag place holders. It is a hack but it works.
$html=$html -replace "xopen","<"
$html=$html -replace "xclose",">"
```



```
#add to fragments
$Fragments+=$html
```

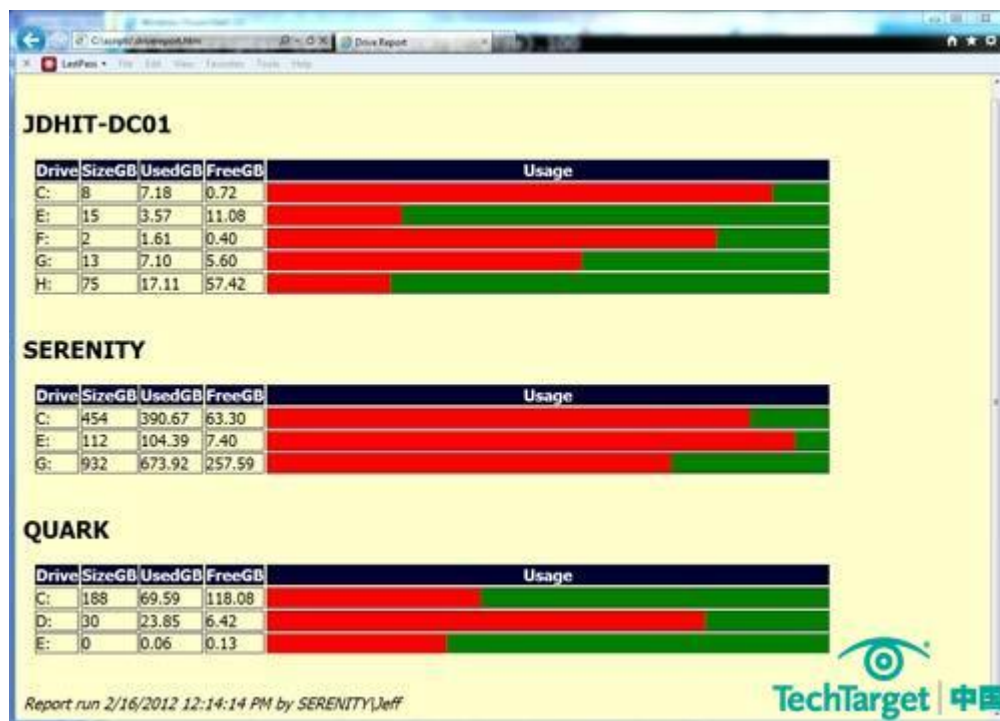
这也许不是最好的解决方法，但是它绝对有效。这个脚本包括所有 HTML 代码并创建最终文件。

```
ConvertTo-Html -head $head -body $fragments | Out-File $Path
```

如果要将它应用到一组计算机上，我可以运行如下命令：

```
PS C:\scripts> .\demo-HtmlBarChart.ps1 "jdhit-dc01","serenity","quark"
```

以下是结果所示：



这只是一个 demo 脚本，因此还有很多不足。也许这可以给大家一个方向，找到更好更完善的方式。如果你已经有了好的想法，不妨和我们来分享一下你的 PowerShell 心得。

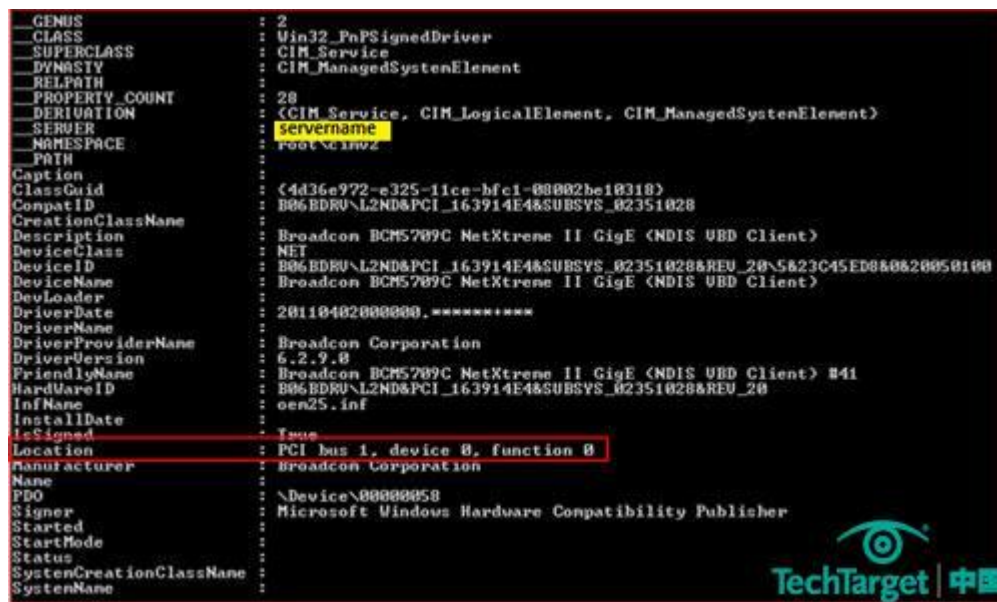
Powershell: 为自动化收集 NIC PCI 总线信息

当你进行一次 [Hyper-V](#) 集群的部署时，配置网络会是一件痛苦的事情。不同的厂商或变更的硬件布局仅仅是自动化部署所遇挑战中的两个举例。这篇文章中我将和你分享用 PowerShell 收集信息的方法，这些信息关于哪个网络适配器位于什么 PCI 总线。你之后可以使用这些信息来重命名网络适配器，组合、更改网络适配器设置等等。

我们首先从收集现有网络适配器的信息开始。完成该过程的 PowerShell 命令如下：

```
Get-WMIObject Win32_PNPSignedDriver | where { $_.DeviceClass -eq "NET" -and $_.HardwareID -like "*PCI*" }
```

结果如下图所示：



```
GENUS : 2
CLASS : Win32_PnPSignedDriver
SUPERCLASS : CIM_Service
DYNASTY : CIM_ManagedSystemElement
RELPATH :
PROPERTY_COUNT : 28
DERIVATION : <CIM_Service, CIM_LogicalElement, CIM_ManagedSystemElement>
SERVER : servername
NAMESPACE : root\wmi
PATH :
Caption :
ClassGuid : {4d36e972-e325-11ce-bfc1-00002be10318}
CompatID : B06BDRU\L2ND&PCI_163914E4&SUBSYS_02351028
CreationClassName :
Description : Broadcom BCM5709C NetXtreme II GigE (NDIS VBD Client)
DeviceClass : NET
DeviceID : B06BDRU\L2ND&PCI_163914E4&SUBSYS_02351028&REV_20\5&23C45ED0&0&20050100
DeviceName : Broadcom BCM5709C NetXtreme II GigE (NDIS VBD Client)
DevLoader :
DriverDate : 20110402000000.0000000
DriverName :
DriverProviderName : Broadcom Corporation
DriverVersion : 6.2.9.0
FriendlyName : Broadcom BCM5709C NetXtreme II GigE (NDIS VBD Client) #41
HardwareID : B06BDRU\L2ND&PCI_163914E4&SUBSYS_02351028&REV_20
InfName : oem25.inf
InstallDate :
IsSigned : True
Location : PCI bus 1, device 8, function 0
Manufacturer : Broadcom Corporation
Name :
PDO : \Device\NPF{...}
Signer : Microsoft Windows Hardware Compatibility Publisher
Started :
StartMode :
Status :
SystemCreationClassName :
SystemName :
```

在输出中我们发现网络适配器的位置。你可能会想，如果服务器上有 12 个网络适配器，那么它就不实用了。那么我们就在这条 [PowerShell](#) 命令中加入 `| ft Location` 来收集 PCI 总线信息。

```
Get-WMIObject Win32_PNPSignedDriver | where { $_.DeviceClass -eq "NET" -and $_.HardwareID -like "*PCI*" } | ft Location
```

```
PS C:\> Get-WMIObject Win32_PNPSignedDriver | where { $_.DeviceClass -eq "NET" -and $_.HardwareID -like "*PCI*" } | ft Location

Location
-----
PCI bus 13, device 0, function 1
PCI bus 13, device 0, function 0
PCI bus 12, device 0, function 1
PCI bus 12, device 0, function 0
PCI bus 8, device 0, function 1
PCI bus 8, device 0, function 0
PCI bus 7, device 0, function 1
PCI bus 7, device 0, function 0
PCI bus 2, device 0, function 1
PCI bus 2, device 0, function 0
PCI bus 1, device 0, function 1
PCI bus 1, device 0, function 0
```

现在我们拥有所有服务器中网络适配器的位置了，但是哪个是哪个呢？

我们需要的是[适配器](#)名称，比如任务管理器。下面的命令会让你得到这些信息。同样对于所有适配器，它就有些不适用了。

```
Get-WMIObject Win32_NetworkAdapter | where { $_.PNPDeviceID -eq $Adapter.DeviceID }
```

让我们将第一条命令放入变量中并且对第二条命令做一个循环。要显示结果，我们做一个简单的 Write-Host 来显示输出。接着脚本会显示如下：

```
$Adapters = Get-WMIObject Win32_PNPSignedDriver | where { $_.DeviceClass -eq "NET" -and
$_HardwareID -like "*PCI*" }
foreach ($Adapter in $Adapters) {
$AdapterName = Get-WMIObject Win32_NetworkAdapter | where { $_.PNPDeviceID -eq $Adapter.DeviceID }
Write-Host 'Adapter Name : ' $AdapterName.NetConnectionID
Write-Host 'PCI BUS : ' $AdapterName.Location
Write-Host 'MAC Address : ' $AdapterName.MACAddress
Write-Host 'GUID : ' $AdapterName.GUID
Write-Host
}
```

结果会怎么样呢？看看下面吧！

```

Administrator: C:\Windows\system32\cmd.exe - powershell
PS C:\Install> .\test.ps1
Adapter Name      : Local Area Connection 11
PCI BUS           : PCI bus 13, device 0, function 1
MAC Address       : 00:1B:21:B5:82:15
GUID              : {8793162F-A66C-4E51-8815-F3D41C72B5E0}

Adapter Name      : LM-TEAM-NIC2
PCI BUS           : PCI bus 13, device 0, function 0
MAC Address       : 00:1B:21:B5:83:1C
GUID              : {CD82BE5A-1CFD-4EAB-BCDA-73A9D2B337AE}

Adapter Name      : TRUNK-TEAM-NIC2
PCI BUS           : PCI bus 12, device 0, function 1
MAC Address       : 00:1B:21:B5:83:19
GUID              : {A14AD1FF-4AC2-464A-A182-02666698BE61}

Adapter Name      : ISCSI-NIC2
PCI BUS           : PCI bus 12, device 0, function 0
MAC Address       : 00:1B:21:B5:82:10
GUID              : {27302C1C-1FD2-44D6-A580-1E7411D3D423}

Adapter Name      : Local Area Connection 7
PCI BUS           : PCI bus 8, device 0, function 1
MAC Address       : 00:1B:21:B5:83:1D
GUID              : {67ADDF5C-17CB-4BA7-962D-9DBE5D12F1F5}

Adapter Name      : LM-TEAM-NIC1
PCI BUS           : PCI bus 8, device 0, function 0
MAC Address       : 00:1B:21:B5:83:1C
GUID              : {83B384B6-C2E9-437E-BCD9-954508FDADC2}

Adapter Name      : TRUNK-TEAM-NIC1
PCI BUS           : PCI bus 7, device 0, function 1
MAC Address       : 00:1B:21:B5:83:19
GUID              : {34988330-EA85-4931-B90E-12D313F2DD29}

Adapter Name      : ISCSI-NIC1
PCI BUS           : PCI bus 7, device 0, function 0
MAC Address       : 00:1B:21:B5:83:18
GUID              : {8B3A74FD-8F37-466D-9563-83CD3EDEC7B2}

Adapter Name      : CSU-HB-TEAM-NIC2
PCI BUS           : PCI bus 2, device 0, function 1
MAC Address       : BC:30:5B:E4:EB:7E
GUID              : {45908E7E-69BD-40CC-8A35-F628D82927FC}

Adapter Name      : SERVER-TEAM-NIC2
PCI BUS           : PCI bus 2, device 0, function 0
MAC Address       : BC:30:5B:E4:EB:7C
GUID              : {11BE9B0B-1905-4D6A-983F-70174994B3A5}

Adapter Name      : CSU-HB-TEAM-NIC1
PCI BUS           : PCI bus 1, device 0, function 1
MAC Address       : BC:30:5B:E4:EB:7E
GUID              : {D0EBFED8-2FBA-4876-8530-C9841577AA84}

Adapter Name      : SERVER-TEAM-NIC1
PCI BUS           : PCI bus 1, device 0, function 0
MAC Address       : BC:30:5B:E4:EB:7C
GUID              : {4B78A442-B1A7-456E-83A6-1AC7292AEF7E}
  
```

就是这样子。我还添加了 MAC 地址和 GUID。这个实例的 MAC 地址还和博通的 BACSccli.exe 命令行工具联合使用，用来配置网络适配器设置。如果需要，GUID 可以用来添加 TcpAckFrequency 到注册表。

复制粘贴会确保所有单双引号都正确。希望这篇文章能对你有用。

如何用 PowerShell 脚本节省 SharePoint 备份时间

企业内微软 SharePoint 越来越受欢迎，这与很多管理员对它的厌烦和嫌弃形成鲜明对比。引起愤怒的一个原因是 SharePoint 在多个不同地方存储所有相关数据的方式，这种方式多少让 SharePoint 特有的备份执行起来有些棘手。要备份每个独立部分并不是不可能，但是这确实很麻烦。

幸运的是，程序员 Jesper M. Christensen 决定就此事做点什么。Christensen 坐下来并写下了一个 PowerShell 脚本，这个脚本让备份 Windows SharePoint Services 或微软 Office SharePoint Server 的整个实例成为可能，这包括了站点、12 贮备和 IIS 元数据。通过使用 stsadm.exe 工具能备份数据库。

在计划任务或其它脚本根据需要发起的任务中，这个叫做 SP 备份的脚本可以手动运行，并且根据微软公共授权 (Ms-PL) 它可以免费获得。这个脚本支持 IIS 6.0 和 IIS 7.0 备份并且在所有支持 SharePoint (客户端和服务端) 且有 PowerShell 的 Windows 版本上工作。

SP 备份由两个部分组成：脚本本身和一个 XML 配置文件，脚本解析这个文件来获得备份目标 (可以是本地路径或服务器共享) 等信息。两个文件需要放在同一个目录，但是属于哪个目录就完全由你决定了。

当然，你会需要编辑 XML 文件来为你的服务器提供合适的参数。参数中的一些是脚本内部的且只在脚本的文件中解释。举例来说，备份目标最大保存期参数让你指定老的备份在自动删除之前可以保留多少天。

你还能设置参数来允许工作结束时一起发送电邮和备份过程中发生事件的描述。其它参数让你指定 IIS 6.0 元数据解密的密码，所以你可以选择在其它服务器上存储或执行一次灾难性备份 (和管理中心路径执行一样的完整 SharePoint 备份)。

还有一个报告也登入到控制台，但是脚本中没有现成的规定来向系统日志中写入报告。说实在的，通过运用脚本和写事件日志命令内的发送邮件功能来写日志结果似乎并不难。另外，用户要求的一个额外功能是不用进行完整推进备份 (一个已经有文件证明的过程) 就能备份 GAC 文件夹和虚拟目录。

注意，脚本暴露的选项没有组成能通过 stsadm.exe (它为 SharePoint 自身执行备份操作) 选项的完整列表。完全用不上一个选项是备份线程，尽管建议说那些运行着的 SharePoint Server 2007 要把它设置到 3

来增效，但它的默认值是 1。你可以根据你的性能要求来向脚本中直接添加这个。如果你的服务器上有多核（现在还有人不是吗？），能用多少就用多少，这有重要意义。

最后，虽然脚本执行备份，它并不执行存储操作，所以你仍然有必要为各个 SharePoint 元素手动并分离地执行它们。