

# Windows PowerShell™ 入门

---

Microsoft Corporation  
发布日期：2006 年 9 月

## 摘要

Windows PowerShell™ 是专为系统管理员设计的新 Windows 命令行外壳程序。该外壳程序包括交互式提示和脚本环境，两者既可以独立使用也可以组合使用。本文档介绍了 Windows PowerShell 的基本概念和功能，并提供了一些使用 Windows PowerShell 进行系统管理的建议方法。

**Microsoft®**

---

# 目录

---

Windows PowerShell 入门版权声明.....	
Windows PowerShell 简介.....	
目标受众.....	
关于 Windows PowerShell.....	
可发现特性.....	
一致性.....	
交互式脚本环境.....	
面向对象.....	
易于过渡到脚本.....	
安装和运行 Windows PowerShell.....	
安装要求.....	
安装 Windows PowerShell.....	
运行 Windows PowerShell.....	
Windows PowerShell 基础知识.....	
理解重要的 Windows PowerShell 概念.....	
命令不是基于文本的.....	
命令系列是可扩展的.....	
Windows PowerShell 处理控制台输入和显示.....	
Windows PowerShell 使用某些 C# 语法.....	
了解 Windows PowerShell 名称.....	
Cmdlet 使用“动词-名词”名称以减少命令记忆量.....	
Cmdlet 使用标准参数.....	
帮助参数 (?).....	
通用参数.....	
建议参数.....	
获取摘要命令信息.....	
显示可用命令类型.....	
获取详细帮助信息.....	
使用熟悉的命令名称.....	
解释标准别名.....	

创建新别名.....	
使用 Tab 扩展来自动完成名称.....	
对象管道.....	
了解 Windows PowerShell 管道.....	
查看对象结构 (Get-Member).....	
使用格式命令更改输出视图.....	
使用 Format-Wide 输出单个项目.....	
使用 Column 控制 Format-Wide 显示.....	
使用 Format-List 显示列表视图.....	
使用 Format-List 和通配符来获取详细信息.....	
使用 Format-Table 显示表格格式输出.....	
改进 Format-Table 输出 (AutoSize).....	
Format-Table 输出在列中换行 (Wrap).....	
组织表输出 (-GroupBy).....	
使用 Out-* Cmdlet 重定向数据.....	
对控制台输出进行分页 (Out-Host).....	
放弃输出 (Out-Null).....	
打印数据 (Out-Printer).....	
保存数据 (Out-File).....	
Windows PowerShell 导航.....	
在 Windows Powershell 中管理当前位置.....	
获取当前位置 (Get-Location).....	
设置当前位置 (Set-Location).....	
保存和撤回最近的位置 ( Push-Location 和 Pop-Location ) .....	
管理 Windows PowerShell 驱动器.....	
添加新的 Windows PowerShell 驱动器 (New-PSDrive).....	
删除 Windows PowerShell 驱动器 (Remove-PSDrive).....	
在 Windows PowerShell 外部添加和删除驱动器.....	
处理文件、文件夹和注册表项.....	
枚举文件、文件夹和注册表项 (Get-ChildItem).....	
列出所有包含的项 (-Recurse).....	
按名称筛选项 (-Name).....	
强制列出隐藏项 (-Force).....	
将项名称与通配符匹配.....	

排除项 (-Exclude).....	
混合使用 Get-ChildItem 参数.....	
直接对项进行操作.....	
创建新项 (New-Item).....	
为什么注册表值不属于项.....	
重命名现有项 (Rename-Item).....	
移动项 (Move-Item).....	
复制项 (Copy-Item).....	
删除项 (Remove-Item).....	
执行项 (Invoke-Item).....	
处理对象.....	
获取 WMI 对象 (Get-WmiObject).....	
获取 WMI 对象 (Get-WmiObject).....	
列出 WMI 类.....	
显示 WMI 类详细信息.....	
使用 Format Cmdlet 显示非默认的属性.....	
创建 .NET 和 COM 对象 (New-Object).....	
使用 New-Object 访问事件日志.....	
将构造函数与 New-Object 结合使用.....	
在变量中存储对象.....	
使用 New-Object 访问远程事件日志.....	
使用对象方法清除事件日志.....	
使用 New-Object 创建 COM 对象.....	
使用 WScript.Shell 创建桌面快捷方式.....	
使用 Windows PowerShell 中的 Internet Explorer.....	
获取有关 .NET-Wrapped COM 对象的警告.....	
使用静态类和方法.....	
使用 System.Environment 获取环境数据.....	
引用 System.Environment 静态类.....	
显示 System.Environment 的静态属性.....	
使用 System.Math 进行数学运算.....	
从管道中删除对象 (Where-Object).....	
使用 Where-Object 执行简单测试.....	
根据对象属性进行筛选.....	
对多个对象重复同一任务 (ForEach-Object).....	
选择对象的各个部分 (Select-Object).....	

对对象进行排序.....	
使用变量存储对象.....	
创建变量.....	
对变量进行操作.....	
使用 Cmd.exe 变量.....	
使用 Windows PowerShell 执行管理任务.....	
管理本地进程.....	
列出进程 (Get-Process).....	
停止进程 (Stop-Process).....	
停止所有其他 Windows PowerShell 会话.....	
管理本地服务.....	
列出服务.....	
停止、启动、挂起和重新启动服务.....	
收集有关计算机的信息.....	
列出桌面设置.....	
列出 BIOS 信息.....	
列出处理器信息.....	
列出计算机制造商和型号.....	
列出已安装的修补程序.....	
列出操作系统版本信息.....	
列出本地用户和所有者.....	
获得可用磁盘空间.....	
获得登录会话信息.....	
获得登录到计算机的用户.....	
从计算机获得本地时间.....	
显示服务状态.....	
处理软件安装.....	
列出 Windows Installer 应用程序.....	
列出所有可卸载的应用程序.....	
安装应用程序.....	
删除应用程序.....	
升级 Windows Installer 应用程序.....	
更改计算机状态：锁定、注销、关闭和重新启动.....	
锁定计算机.....	
注销当前会话.....	
关闭或重新启动计算机.....	

处理打印机.....	
列出打印机连接.....	
添加网络打印机.....	
设置默认打印机.....	
删除打印机连接.....	
执行网络任务.....	
列出计算机的 IP 地址.....	
列出 IP 配置数据.....	
对计算机执行 Ping 操作.....	
检索网络适配器属性.....	
为网络适配器指定 DNS 域.....	
执行 DHCP 配置任务.....	
确定启用 DHCP 的适配器.....	
检索 DHCP 属性.....	
在每个适配器上启用 DHCP.....	
对特定适配器解除和续订 DHCP 租约.....	
对所有适配器解除和续订 DHCP 租约.....	
创建网络共享.....	
删除网络共享.....	
连接 Windows 可访问的网络驱动器.....	
处理文件和文件夹.....	
列出文件夹中的所有文件和文件夹.....	
复制文件和文件夹.....	
创建文件和文件夹.....	
删除文件夹中的所有文件和文件夹.....	
将本地文件夹映射为 Windows 可访问驱动器.....	
将文本文件读入数组.....	
处理注册表项.....	
列出注册表项的所有子项.....	
复制项.....	
创建项.....	
删除项.....	
删除特定项下的所有项.....	
处理注册表条目.....	
列出注册表条目.....	
获取单个注册表条目.....	
创建新的注册表条目.....	
重命名注册表条目.....	

删除注册表条目.....	
附录 1 - 兼容性别名.....	
附录 2 - 创建自定义的 PowerShell 快捷方式.....	

# Windows PowerShell 入门版权声明

---

本文档仅供参考，Microsoft 在本文档中不提供任何明示或暗示的保证。本文档中的信息（包括引用的 URL 和其他 Internet 网站）如有变动，恕不另行通知。全部使用风险或使用本文档产生的结果由用户承担。除非另行说明，否则本文档范例中所提及的公司、组织、产品、域名、电子邮件地址、徽标、人员、地点和事件均属虚构，并无有意联系或暗示任何实际的公司、组织、产品、域名、电子邮件地址、徽标、人员、地点或事件。遵守所有适用的版权法是用户的责任。在不限版权许可的权利的情况下，如果没有得到 **Microsoft Corporation** 明确书面许可，本文档的任何部分不得被复制、存储或引进检索系统，或者以任何形式、任何方式（电子、机械、影印、录音或其他）或为任何目的进行传播。

本文档可能涉及 **Microsoft Corporation** 的专利、正在申请的专利、商标、版权或其他知识产权。除非与 **Microsoft Corporation** 签订的书面许可协议中有明确规定，否则使用本文档并不意味着授予使用这些专利、商标、版权或其他知识产权的任何许可。

© 2006 Microsoft Corporation。保留所有权利。

Microsoft、MS-DOS、Windows、Windows NT、Windows 2000、Windows XP 和 Windows Server 2003 是 Microsoft Corporation 在美国和/或其他国家（地区）的注册商标或商标。

此处所提及的实际公司和产品的名称可能是其各自所有者的商标。

## Windows PowerShell 简介

---

Windows PowerShell 是一种命令行外壳程序和脚本环境，使命令行用户和脚本编写者可以利用 .NET Framework 的强大功能。它引入了许多非常有用的新概念，从而进一步扩展了您在 Windows 命令提示符和 Windows Script Host 环境中获得的知识 and 创建的脚本。

### 目标受众

Windows PowerShell 入门主要面向之前没有 Windows PowerShell 背景知识的 IT 专业人员、程序员和高级用户。虽然具备脚本和 WMI 方面的背景知识会有所帮助，但是理解本文档并不假定或要求您具备此方面知识。

## 关于 Windows PowerShell

---

通过解决长期存在的问题并添加一些新的功能，Windows PowerShell 旨在改进命令行和脚本环境。

### 可发现特性

您可轻易发现 Windows Powershell 的功能。例如，若要查找用于查看和更改 Windows 服务的 cmdlet 列表，请键入：

```
get-command *-service
```



在发现可完成任务的 cmdlet 之后，可以使用 Get-Help cmdlet 了解有关该 cmdlet 的详细信息。例如，若要显示有关 Get-Service cmdlet 的帮助，请键入：

```
get-help get-service
```

若要充分理解该 cmdlet 的输出，则可通过管道将其输出传递给 Get-Member cmdlet。例如，以下命令将通过 Get-Service cmdlet 显示有关该对象输出的成员的信息。

```
get-service | get-member
```

## 一致性

管理系统可能是一项复杂的任务，而具有统一接口的工具将有助于控制其固有的复杂性。然而，无论是命令行工具还是可编写脚本的 COM 对象，在一致性方面都乏善可陈。

Windows PowerShell 的一致性是其主要优点中的一项。例如，如果您学会了如何使用 Sort-Object cmdlet，则可利用这一知识对任何 cmdlet 的输出进行排序。而无需了解每个 cmdlet 的不同的排序例程。

此外，cmdlet 开发人员也不必为其 cmdlet 设计排序功能。Windows PowerShell 为他们提供了框架，而该框架可提供基本的功能，并强制他们在接口的许多方面保持一致。该框架虽然消除了通常会留给开发人员的某些选项，但作为回报，开发强健、易于使用的 cmdlet 的工作将更加简单。

## 交互式脚本环境

Windows PowerShell 将交互式环境和脚本环境组合在一起，从而允许您访问命令行工具和 COM 对象，同时还可利用 .NET Framework 类库 (FCL) 的强大功能。

此环境对 Windows 命令提示符进行了改进，后者提供了带有多种命令行工具的交互式环境。此外，还对 Windows Script Host (WSH) 脚本进行了改进，后者允许您使用多种命令行工具和 COM 自动对象，但未提供交互式环境。

通过将对所有这些功能的访问组合在一起，Windows PowerShell 扩展了交互用户和脚本编写者的能力，从而更易于进行系统管理。

## 面向对象

尽管您可以通过以文本方式键入命令与 Windows PowerShell 进行交互，但 Windows PowerShell 是基于对象的，而不是基于文本的。命令的输出即为对象。可以将输出对象发送给另一条命令以作为其输入。因此，Windows PowerShell 为曾使用过其他外壳程序的人员提供了熟悉的界面，同时引入了新的、功能强大的命令行范例。通过允许发送对象（而不是文本），它扩展了在命令之间发送数据的概念。

## 易于过渡到脚本

使用 Windows PowerShell，您可以很方便地从以交互方式键入命令过渡到创建和运行脚本。您可以在 Windows PowerShell 命令提示符下键入命令以找到可执行任务的命令。随后，可将这些命令保存到脚本或历史记录中，然后将其复制到文件中以用作脚本。

# 安装和运行 Windows PowerShell

---

## 安装要求

在安装 Windows PowerShell 之前，请确保您的系统中已安装 Windows PowerShell 要求的软件程序。Windows PowerShell 要求以下程序：

Windows XP Service Pack 2、Windows 2003 Service Pack 1 或 Windows 的更高版本

Microsoft .NET Framework 2.0

如果计算机上已安装任意版本的 Windows PowerShell，则在安装新版本之前，应使用“控制面板”中的“添加或删除程序”将其卸载。

## 安装 Windows PowerShell

若要安装 Windows PowerShell，请执行以下步骤：

1. 下载 Windows PowerShell 安装文件。（该文件名将因平台、操作系统和语言包的不同而异。）
2. 若要开始安装，请单击“打开”。
3. 按照安装向导页上的说明进行安装。

也可将 Windows PowerShell 文件保存到网络共享，以便在多个计算机上进行安装。

若要执行无提示安装，请键入：

```
<PowerShell-exe-file-name> /quiet
```

例如，

```
PowerShellSetup_x86_fre.exe /quiet
```

在 32 位版本的 Windows 中，Windows PowerShell 默认情况下安装在 %SystemRoot%\System32\WindowsPowerShell\v1.0 目录中。在 64 位版本的 Windows 中，32 位版本的 Windows PowerShell 安装在 %SystemRoot%\SystemWow64\WindowsPowerShell\v1.0 目录中，而 64 位版本的 Windows PowerShell 则安装在 %SystemRoot%\System32\WindowsPowerShell\v1.0 目录中。

## 运行 Windows PowerShell

若要从“开始”菜单启动 Windows PowerShell，请依次单击“开始”、“所有程序”、“**Windows PowerShell 1.0**”，以及 **Windows PowerShell** 图标。

若要从“运行”框中启动 Windows PowerShell，请依次单击“开始”、“运行”，键入 **powershell**，然后单击“确定”。

若要从命令提示符 (cmd.exe) 窗口启动 Windows PowerShell，请在命令提示符下键入 **powershell**。由于 Windows PowerShell 运行在控制台会话中，因此可以使用相同技术在远程 telnet 或 SSH 会话中运行它。若要返回到命令提示符会话，请键入 **exit**。

## Windows PowerShell 基础知识

---

图形界面使用了大多数计算机用户所熟知的一些基本概念。用户可以依靠对这些界面的熟悉来完成其任务。操作系统为用户呈现了可浏览的图形化的项目表示形式，通常包括下拉菜单（用于访问特定功能），以及上下文菜单（用于访问特定于上下文的功能）。

命令行接口 (CLI)（例如 Windows PowerShell）必须使用不同的方式来公开信息，因为它没有可方便用户的菜单或图形系统。只有在知道命令的名称后，才可对其进行使用。尽管可以键入相当于 GUI 环境中的功能的复杂命令，但前提是您必须熟悉常用的命令和命令参数。

大多数 CLI 都不提供可帮助用户了解该接口的模式。由于 CLI 是最早的操作系统外壳程序，因此许多命令名称和参数名称都是随意选择的。通常，选择名称的原则是简洁扼要而非清楚明了。尽管大多数 CLI 中都集成了帮助系统和命令设计标准，但是它们通常都是为了与最早的命令兼容而设计的，因此命令集仍然保持着几十年前确定的形式。

而 Windows PowerShell 设计原则是可以利用用户在 CLI 方面的以往知识。在本章中，我们将介绍一些可用于快速了解 Windows PowerShell 的基本工具和概念。其中包括：

- 使用 `Get-Command`

- 使用 `Cmd.exe` 和 UNIX 命令

- 使用外部命令

- 使用 `Tab` 补齐功能

- 使用 `Get-Help`

## 理解重要的 Windows PowerShell 概念

---

Windows PowerShell 在设计上集成了源自众多不同环境的概念。尽管使用过特定外壳程序或编程环境的人可能会熟悉其中几个概念，但很少有人了解所有这些概念。查看这些概念可帮助您概括了解本外壳程序。

### 命令不是基于文本的

与传统的命令行界面命令不同，Windows PowerShell cmdlet 旨在可处理对象 — 结构化的信息，而不只是显示在屏幕上的字符串。命令输出始终提供需要使用的额外信息。我们将在本文中深入讨论此主题。

如果您过去曾使用文本处理工具来处理命令行数据，则会发现，在 Windows PowerShell 中尝试使用这些工具时，其行为会有所不同。在大多数情况下，您不需要使用文本处理工具来提取特定信息。通过使用标准的 Windows PowerShell 对象操纵命令，可直接访问任何数据部分。


### 命令系列是可扩展的

许多接口（例如 `Cmd.exe`）均不能提供直接扩展内置命令集的方法。您可以创建在 `Cmd.exe` 中运行的外部命令行工具，但这些外部工具不能提供服务（例如帮助集成），并且 `Cmd.exe` 不能自动了解它们是有效的命令。

Windows PowerShell 中的本机二进制命令（也称为 cmdlet，读作 `command-let`）可以通过创建并使用管理单元添加到 Windows PowerShell 中的 cmdlet 进行补充。Windows PowerShell 管理

单元已经过编译，与任何其他接口中的二进制工具相同。您可以使用它们将 Windows PowerShell 提供程序以及新的 cmdlet 添加到外壳程序中。

由于 Windows PowerShell 内部命令的特殊性质，因此我们称它们为 cmdlet。

 请注意：

Windows PowerShell 可以运行除 cmdlet 以外的其他命令。“Windows PowerShell 入门”中将不会详细介绍这些命令，但了解这些命令类型类别会很有帮助。Windows PowerShell 支持与 UNIX 外壳程序脚本和 Cmd.exe 批处理文件类似但文件扩展名为 .ps1 的脚本。Windows PowerShell 还允许您创建可直接用于界面或脚本中的内部函数。


## Windows PowerShell 处理控制台输入和显示

键入命令时，Windows PowerShell 始终会直接处理命令行输入。Windows PowerShell 还可以设置屏幕上显示的输出格式。此功能十分重要，因为它可以减少处理每个 cmdlet 所需的工作，并确保您在使用任何一种 cmdlet 时都能始终以相同方式进行操作。这样有助于简化工具开发人员和工作的工作，我们不妨以命令行帮助为例。

对于请求和显示帮助，传统的命令行工具都有其自身的方案。一些命令行工具使用 `/?` 来触发帮助显示；其他命令行工具使用 `-?`、`/H` 甚至 `//`。其中一些会以 GUI 窗口显示帮助，而不以控制台显示。某些复杂工具（例如，应用程序更新程序）会先将内部文件解包，然后才显示其帮助。如果使用的参数不正确，则该工具可能会忽略键入的内容，并自动开始执行任务。

在 Windows PowerShell 中输入命令时，Windows PowerShell 将自动分析并预处理所输入的全部内容。如果将 Windows PowerShell cmdlet 与 `-?` 参数一起使用，则始终表示“显示此命令的帮助”。Cmdlet 开发人员无需分析该命令；他们只需提供帮助文本。

即使是在 Windows PowerShell 中运行传统命令行工具，您也可以使用 Windows PowerShell 的帮助功能，了解这一点十分重要。Windows PowerShell 将对参数进行处理并将结果传递给外部工具。

 请注意：

如果在 Windows PowerShell 中运行图形应用程序，将会打开该应用程序的窗口。只有在处理您提供的命令行输入或返回到控制台窗口的应用程序输出时，Windows PowerShell 才会进行干预；它不会影响该应用程序的内部工作方式。

## Windows PowerShell 使用某些 C# 语法

由于 Windows PowerShell 基于 .NET Framework，因此 Windows PowerShell 的语法功能和关键字与 C# 编程语言中所使用的语法功能和关键字十分类似。如果您对 C# 语言感兴趣，通过了解 Windows PowerShell 将便于您轻松学习该语言。

如果您不是 C# 程序员，这一相似性则无关紧要。不过，如果您已熟悉 C#，则这些相似性可以让您更轻松地了解 Windows PowerShell。

## 了解 Windows PowerShell 名称

使用大多数命令行界面时，了解命令名称及命令参数需要花费大量的时间。问题在于可用模式很少，因此只能通过记住常用的每种命令和每个参数来了解这些命令行界面。

使用新命令或参数时，您通常无法使用已知的名称，而必须查找和了解新的名称。考虑到界面是如何从一个小工具集经过不断添加的新增功能发展而来，就很容易了解为何其结构会是非标

准的。尤其是使用命令名称时，这一点可能是必然的，因为每种命令都是单独的工具。不过，也可以通过更好的方式来处理命令名称。

大多数命令都是为管理操作系统或应用程序的元素（例如，服务或进程）构建的。这些命令具有各种名称，它们可能属于或不属于一个系列。例如，在 Windows 系统上可以使用 **net start** 和 **net stop** 命令来启动和停止服务。此外，还有一个名为 **sc** 的更通用的 Windows 服务控制工具，这一完全不同的名称与 **net** 服务命令的命名模式不一致。对于进程管理，Windows 使用 **tasklist** 命令来列出进程，并使用 **taskkill** 命令来终止进程。

用于接受参数的命令具有不一致的参数规范。不能使用 **net start** 命令来启动远程计算机上的服务。**sc** 命令将启动远程计算机上的服务，但是，若要指定远程计算机，则必须在其名称前添加双反斜杠作为前缀。例如，若要在名为 DC01 的远程计算机上启动后台处理程序服务，您可以键入 **sc \\DC01 start spooler**。若要列出在 DC01 上运行的任务，您需要使用 **/S**（表示“系统”）参数，并提供如下所示的不带反斜杠的名称 DC01: **tasklist /S DC01**。

尽管服务与进程之间有很大的技术差别，但它们都是计算机上具有定义完整的生命周期的可管理元素示例。您可能需要启动或停止服务或进程，或获得所有当前正在运行的服务或进程的列表。换言之，尽管服务和进程并不相同，但从概念上来看，对服务或进程执行的操作通常是相同的。此外，从概念上来看，通过指定参数来自定义操作时所做的选择也可能十分类似。

Windows PowerShell 利用这些相似性来减少要了解和使用的 cmdlet 所需知道的不同名称数。

## Cmdlet 使用“动词-名词”名称以减少命令记忆量

Windows PowerShell 使用“动词-名词”命名系统，其中的每个 cmdlet 名称均由标准动词、连字符和特定名词组成。Windows PowerShell 动词并非一定为英语动词，但其表示 Windows PowerShell 中的特定操作。名词与所有语言中的名词十分类似，它们描述在系统管理中起重要作用的特定对象类型。我们将提供几个动词和名词的示例，以便于向您演示这些由两部分组成的名称是如何减少您了解命令所需的记忆量。

名词所受的限制较少，但它们应始终描述命令的操作对象。Windows PowerShell 包含诸如 **Get-Process**、**Stop-Process**、**Get-Service** 和 **Stop-Service** 之类的命令。

如果仅有两个名词和两个动词，则一致性并不会极大地简化您了解这些命令的过程。但是，假定是由 10 个动词和 10 个名词组成的一组标准命令名称，则您只需记住 20 个单词，而使用这些单词可以构成 100 个不同的命令名称。

通常，您只需通过命令的名称即可识别其用途，而对新命令应使用什么样的名称，这通常也是显而易见的。例如，计算机关机命令可能为 **Stop-Computer**。用于列出网络上所有计算机的命令可能为 **Get-Computer**。用于获取系统日期的命令为 **Get-Date**。

使用 **Get-Command** 和 **-Verb** 参数可以列出所有包含特定动词的命令（我们将在下一节中详细介绍 **Get-Command**）。例如，若要显示所有使用动词 **Get** 的 cmdlet，请键入：

```
PS> Get-Command -Verb Get
CommandType      Name                      Definition
-----
Cmdlet           Get-Acl                  Get-Acl [[-Path] <String[]>]...
Cmdlet           Get-Alias                Get-Alias [[-Name] <String[]>]...
Cmdlet           Get-AuthenticodeSignature Get-AuthenticodeSignature [-...]
Cmdlet           Get-ChildItem            Get-ChildItem [[-Path] <Stri...
```

**-Noun** 参数用处更大，因为使用该参数可以查看影响同一对象类型的一系列命令。例如，若要显示可用于管理服务的命令，请键入以下命令：

```
PS> Get-Command -Noun Service
CommandType      Name                      Definition
-----
```

Cmdlet	Get-Service	Get-Service [-Name] <String>...
Cmdlet	New-Service	New-Service [-Name] <String>...
Cmdlet	Restart-Service	Restart-Service [-Name] <Strin>...
Cmdlet	Resume-Service	Resume-Service [-Name] <String>...
Cmdlet	Set-Service	Set-Service [-Name] <String>...
Cmdlet	Start-Service	Start-Service [-Name] <String>...
Cmdlet	Stop-Service	Stop-Service [-Name] <String>...
Cmdlet	Suspend-Service	Suspend-Service [-Name] <String>...
...		

如果命令具有“动词-名词”命名方案，并不意味着该命令一定为 `cmdlet`。有关具有“动词-名词”名称但不是 `cmdlet` 的命令，一个示例就是本机 Windows PowerShell 命令 `Clear-Host`，该命令用于清除控制台窗口。`Clear-Host` 命令实际上是一个内部函数，如果对其运行 `Get-Command` 您将会发现这一点：

```
PS> Get-Command -Name Clear-Host
```

CommandType	Name	Definition
-----	----	-----
Function	Clear-Host	\$spaceType = [System.Managem...

## Cmdlet 使用标准参数

正如上文所述，传统命令行界面中所使用的命令通常不具有一致的参数名称。有时，参数甚至没有名称。在这种情况下，这些参数通常为可迅速键入的单个字符或缩写单词，但其不易于新用户理解。

与其他大多数传统命令行界面不同，Windows PowerShell 将直接处理参数，并将对参数的这种直接访问和开发人员指南一起用于实现参数名称的标准化。尽管这不保证每个 `cmdlet` 都始终遵从命名标准，但其鼓励这样做。

### 请注意：

使用参数时，参数名称前始终具有“-”符号，以便 Windows PowerShell 能够清楚地将其识别为参数。以 **Get-Command -Name Clear-Host** 为例，其中的参数名称为 **Name**，但输入时应为 **-Name**。

下面介绍了标准参数名称和用法的一些常见特征。

## 帮助参数 (?)

对任何 `cmdlet` 指定 **-?** 参数时，将不执行该 `cmdlet`。而 Windows PowerShell 将显示该 `cmdlet` 的帮助。

## 通用参数

Windows PowerShell 的一些参数称为通用参数。由于这些参数是由 Windows PowerShell 引擎进行控制，因此每次 `cmdlet` 实现这些参数时，它们的行为方式将始终相同。通用参数包括 **WhatIf**、**Confirm**、**Verbose**、**Debug**、**Warn**、**ErrorAction**、**ErrorVariable**、**OutVariable** 和 **OutBuffer**。

## 建议参数

Windows PowerShell 核心 cmdlet 对类似参数使用标准名称。尽管参数名称的使用不是强制的，但存在明确的用法指南以鼓励标准化。

例如，该指南建议在对引用计算机的参数进行命名时采用 **ComputerName** 之类的名称，而不采用 **Server**、**Host**、**System**、**Node** 或其他常见的备选单词。这些重要的建议参数名称包括 **Force**、**Exclude**、**Include**、**PassThru**、**Path** 和 **CaseSensitive**。

## 获取摘要命令信息

Windows PowerShell **Get-Command** cmdlet 用于检索所有可用命令的名称。在 Windows PowerShell 提示符下键入 **Get-Command**，则会显示与下列类似的输出：

```
PS> Get-Command
CommandType      Name                Definition
-----
Cmdlet           Add-Content         Add-Content [-Path] <String[...
Cmdlet           Add-History         Add-History [[-InputObject] ...
Cmdlet           Add-Member          Add-Member [-MemberType] <PS...
...
```

此输出与 **Cmd.exe** 的帮助输出十分相似：表格格式的内部命令摘要。如上所示，在 **Get-Command** 命令输出的摘要中所显示的每个命令的 **CommandType** 都是 **Cmdlet**。**Cmdlet** 是 Windows PowerShell 的固有命令类型，该类型大致对应于 **Cmd.exe** 的 **dir** 和 **cd** 命令以及 UNIX 外壳程序（例如 **BASH**）中的内置命令。

在 **Get-Command** 命令的输出中，所有定义都以省略号 (...) 结尾，以指示 PowerShell 无法在可用空间内显示所有内容。在显示输出时，Windows PowerShell 会将输出格式设置为文本，然后对其进行排列，以使数据整齐地显示在控制台窗口中。我们稍后将在有关格式化程序的一节中对此进行介绍。

**Get-Command** cmdlet 有一个 **Syntax** 参数：使用该参数，您可以仅检索每个 cmdlet 的语法。输入 **Get-Command -Syntax** 命令可显示完整的输出：

```
PS> Get-Command -Syntax
Add-Content [-Path] <String[]> [-Value] <Object[]> [-PassThru] [-Filter <String>]
[-Include <String[]>] [-Exclude <String[]>] [-Force] [Credential <PSCredential>]
[-Verbose] [-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable <String>]
[-OutVariable <String>] [-OutBuffer <Int32>] [-WhatIf] [-Confirm] [-Encoding
<FileSystemCmdletProviderEncoding>]

Add-History [[-InputObject] <PSObject[]>] [-Passthru] [-Verbose] [-Debug]
[-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-OutVariable
<String>] [-OutBuffer <Int32>]...
```


## 显示可用命令类型

**Get-Command** 命令不会列出 Windows PowerShell 中的所有可用命令。实际上，它仅列出 Windows PowerShell 会话中的 cmdlet。Windows PowerShell 实际还支持其他几种类型的命令。尽管在“Windows PowerShell 入门”中未对其进行详细介绍，但别名、函数和脚本也是 Windows PowerShell 命令。作为可执行文件或具有已注册文件类型处理程序的外部文件，也被归类于命令。

通过输入以下命令，您可以返回所有可调用项的列表：

```
PS> Get-Command *
```

由于此列表包括搜索路径中的外部文件，因此它可能包含数千个项目。实际上，仅查看精简的命令集更为有用。若要查找其他类型的本机命令，可以使用 **Get-Command** cmdlet 的 **CommandType** 参数。尽管我们尚未介绍其他这些命令类型，但如果知道某类命令的 **CommandType** 名称，您仍然可以显示这些命令类型。

 请注意：

尽管上文中尚未介绍这一点，但您应了解在 Windows PowerShell 命令参数中是使用星号 (\*) 来进行通配符匹配。\* 表示“匹配一个或多个任意字符”。可以键入 **Get-Command a\*** 来查找以字母 “a” 开头的所有命令。与 **Cmd.exe** 中的通配符匹配不同，Windows PowerShell 的通配符还将匹配句点。

若要显示特殊命令类别的别名（即用作标准命令名称的替代名称的昵称），请输入以下命令：

```
PS> Get-Command -CommandType Alias
```

若要显示所有 Windows PowerShell 函数，请输入以下命令：

```
PS> Get-Command -CommandType Function
```

若要显示 Windows PowerShell 搜索路径中的外部脚本，请输入以下命令：

```
PS> Get-Command -CommandType ExternalScript
```

## 获取详细帮助信息

Windows PowerShell 为所有 cmdlet 提供了详细的帮助文档。若要显示帮助主题，请使用 **Get-Help** cmdlet。例如，若要获得 **Get-Childitem** cmdlet 的帮助，请键入：

```
get-help get-childitem
```

或

```
get-childitem -?
```

通过使用 **man** 和 **help** 函数，还可以逐页显示每个帮助主题。若要使用这些函数，请键入 **man** 或 **help**，后跟 cmdlet 名称。例如，若要显示 **Get-Childitem** cmdlet 的帮助，请键入：

```
man get-childitem
```

或

```
help get-childitem
```

**Get-Help** cmdlet 还会显示有关 Windows PowerShell 中的概念性主题的信息。概念性帮助主题以 “about\_” 前缀开头，例如 **about\_line\_editing**。若要显示概念性主题的列表，请键入：

```
get-help about_*
```

若要显示特定的帮助主题，请键入主题名称，例如：

```
get-help about_line_editing
```



# 使用熟悉的命令名称

使用称为别名的机制，Windows PowerShell 允许用户通过替代名称来引用命令。使用别名，具有其他外壳程序经验的用户可以在 Windows PowerShell 中重用其已知的通用命令名称来执行类似操作。尽管我们不会详细介绍 Windows PowerShell 别名，但在 Windows PowerShell 入门阶段您仍然可以使用这些别名。

别名将所键入的命令名称与另一个命令相关联。例如，Windows PowerShell 有一个名为 **Clear-Host** 的内部函数，用于清除输出窗口。如果在命令提示符下键入 **cls** 或 **clear** 命令，则 Windows PowerShell 会将此命令解释为 **Clear-Host** 函数的别名，从而运行 **Clear-Host** 函数。此功能有助于用户了解 Windows PowerShell。首先，大多数 Cmd.exe 和 UNIX 用户都有其已按名称记忆的大型命令清单，尽管 Windows PowerShell 的等价命令可能不会产生完全相同的结果，但它们在形式上的相似性足以让用户无需先记住 Windows PowerShell 名称即可直接使用这些命令来完成工作。其次，对于已经熟悉其他外壳程序的用户而言，了解新外壳程序的主要困难在于“手指记忆”所导致的错误。假定您已使用 Cmd.exe 多年，在您看到满屏输出并希望将其清除时，您会条件反射地键入 **cls** 命令，然后按 Enter 键。但在 Windows PowerShell 中没有 **Clear-Host** 函数的别名，因而您只会看到一条错误消息“无法将‘cls’识别为 cmdlet、函数、可运行程序或脚本文件”，并且仍然不清楚要清除输出应执行的操作。

下面是可在 Windows PowerShell 内使用的通用 Cmd.exe 和 UNIX 命令的简短列表：

cat	dir	mount	rm
cd	echo	move	rmdir
chdir	erase	popd	sleep
clear	h	ps	sort
cls	history	pushd	tee
copy	kill	pwd	type
del	lp	r	write
diff	ls	ren	

如果您发现自己会条件反射地使用这些命令之一，而又希望了解本机 Windows PowerShell 命令的真实名称，则可以使用 **Get-Alias** 命令：

```
PS> Get-Alias cls
```

CommandType	Name	Definition
Alias	cls	Clear-Host

为使示例更易于阅读，“Windows PowerShell 入门”中通常避免使用别名。但是，如果要处理来自其他来源的任意 Windows PowerShell 代码段，或希望定义自己的别名，则尽早了解有关别名的详细信息仍然十分有用。本节中的其余内容将介绍标准别名以及如何定义您自己的别名。

## 解释标准别名

上述别名是专为实现与其他界面的名称兼容性而设计的，Windows PowerShell 中的内置别名与其不同，它们通常是为了简短易用而设计的。这些简短的名称便于快速键入，但如果您不了解

其含义，则无法解读它们。

通过提供一组基于常用动词和名词的速记名称的标准别名，Windows PowerShell 试图在清晰性与简短性之间取得平衡。这样，在一组常用 cmdlet 的核心别名中，您只需了解速记名称即可解读这些命令。例如，在标准别名中，动词 **Get** 缩写为 **g**，动词 **Set** 缩写为 **s**，名词 **Item** 缩写为 **i**，名词 **Location** 缩写为 **l**，而名词 **Command** 缩写为 **cm**。

以下简短示例说明了这一工作机制。**Get-Item** 的标准别名是通过将表示 **Get** 的 **g** 与表示 **Item** 的 **i** 组合而获得的：**gi**。**Set-Item** 的标准别名是通过将表示 **Set** 的 **s** 与表示 **Item** 的 **i** 组合而获得的：**si**。**Get-Location** 的标准别名是通过将表示 **Get** 的 **g** 与表示 **Location** 的 **l** 组合而获得的：**gl**。**Set-Location** 的标准别名是通过将表示 **Set** 的 **s** 与表示 **Location** 的 **l** 组合而获得的：**sl**。**Get-Command** 的标准别名是通过将表示 **Get** 的 **g** 与表示 **Command** 的 **cm** 组合而获得的：**gcm**。不存在 **Set-Command** cmdlet，但如果其存在，我们很容易即可猜测出其标准别名是通过将表示 **Set** 的 **s** 和表示 **Command** 的 **cm** 组合而获得的：**scm**。此外，如果熟悉 Windows PowerShell 别名的人员遇到 **scm**，他们也会猜测到该别名是指 **Set-Command**。

## 创建新别名

使用 **Set-Alias** cmdlet，可以创建您自己的别名。例如，以下语句将创建在“解释标准别名”中介绍的标准 cmdlet 别名：

```
Set-Alias -Name gi -Value Get-Item
Set-Alias -Name si -Value Set-Item
Set-Alias -Name gl -Value Get-Location
Set-Alias -Name sl -Value Set-Location
Set-Alias -Name gcm -Value Get-Command
```

在内部，Windows PowerShell 在启动期间使用此类命令，但这些别名是不可更改的。如果尝试实际执行这些命令之一，您将获得一条错误消息，表明无法修改该别名。例如：

```
PS> Set-Alias -Name gi -Value Get-Item
Set-Alias: 别名不可写入，因为别名 gi 为只读别名或常量，无法写入。
所在行:1 字符:10
+ Set-Alias <<<< -Name gi -Value Get-Item
```

## 使用 Tab 扩展来自动完成名称

命令行外壳程序通常提供了用于自动完成文件或命令的名称的功能，以便提高输入命令的速度并提供相应提示。Windows PowerShell 允许通过按 **Tab** 键来填充文件名和 cmdlet 名称。

 请注意：

**Tab** 扩展是由内部函数 **TabExpansion** 控制的。由于可以修改或覆盖此函数，因而此处将介绍默认 Windows PowerShell 配置的行为指南。

若要自动根据可用选项来填充文件名或路径，请键入部分名称，然后按 **Tab** 键。Windows PowerShell 会自动将该名称扩展为其找到的第一个匹配项。重复按 **Tab** 键将逐一显示所有可用选项。

cmdlet 名称的 **Tab** 扩展略有不同。若要对 cmdlet 名称使用 **Tab** 扩展，请完整键入名称的第一部分（动词）及其后面的连字符。可以填入名称的更多部分以进行部分匹配。例如，如果键入 **get-co** 然后按 **Tab** 键，则 Windows PowerShell 会将其自动扩展为 **Get-Command** cmdlet（注意，其字母大小写也将更改为标准形式）。如果再次按 **Tab** 键，则 Windows PowerShell 将使用仅有的另一个匹配 cmdlet 名称 **Get-Content** 替换上一名称。

可以在同一行上重复使用 Tab 扩展。例如，可以通过输入以下命令来对 **Get-Content** cmdlet 的名称使用 Tab 扩展：

```
PS> Get-Con<Tab>
```

按 **Tab** 键时，该命令将扩展为：


```
PS> Get-Content
```

您随后可以部分指定智能安装程序日志文件的路径，然后再次使用 Tab 扩展：

```
PS> Get-Content c:\windows\acts<Tab>
```

按 **Tab** 键时，该命令将扩展为：

```
PS> Get-Content C:\windows\actsetup.log
```

 请注意：

Tab 扩展的局限之处在于 Tab 始终被解释为尝试完成单词。如果将命令示例复制并粘贴到 Windows PowerShell 控制台中，请确保该示例不包含 **tab**；如果包含，则结果将是难以预见的，并且几乎肯定不会是您预期的结果。

## 对象管道

管道的作用与一系列相连接的管道段相似。沿管道移动的项目将通过其中的每一段。若要在 Windows PowerShell 中创建管道，只需使用管道运算符 “|” 将命令连接在一起，即可将每个命令的输出用作下一命令的输入。

管道无疑是命令行界面中所使用的最有价值的概念。如果适当使用管道，它们不仅可以减少输入复杂命令所需的工作量，而且便于您查看命令中的工作流。有关管道的一个有用特征是：由于管道对每个项目分别进行操作，因此无论管道中存在零个、一个或多个项目，您都无需基于这一点来修改管道。此外，管道中的每个命令（称为管道元素）通常将其输出中的项目逐个传递给管道中的下一命令。这样通常会减少复杂命令的资源需求，并允许您立即开始获取输出。在本章中，我们将介绍 Windows PowerShell 管道与大多数常见外壳程序中的管道的不同之处，然后为您演示一些可用于帮助控制管道输出以及查看管道运行方式的基本工具。

## 了解 Windows PowerShell 管道

实际上，在 Windows PowerShell 中到处都会用到管道。尽管在屏幕上会看到文本，但 Windows PowerShell 并不通过管道在命令之间传递文本。它实际上通过管道传递对象。

用于管道的表示法与其他外壳程序中所使用的表示法十分类似，因此，乍一看可能不会明显察觉到 Windows PowerShell 引入了新功能。例如，如果使用 **Out-Host** cmdlet 来强行逐页显示其他命令的输出，则该输出的外观将与屏幕上显示的正常文本一样，分为各页显示：

```
PS> Get-ChildItem -Path C:\WINDOWS\System32 | Out-Host -Paging
Directory:Microsoft.Windows PowerShell.Core\FileSystem::C:\WINDOWS\system32
```

Mode	LastWriteTime	Length	Name
----	-----	-----	----

```

-a---      2005-10-22  11:04 PM      315 $winnt$.inf
-a---      2004-08-04   8:00 AM     68608 access.cpl
-a---      2004-08-04   8:00 AM     64512 acctres.dll
-a---      2004-08-04   8:00 AM    183808 accwiz.exe
-a---      2004-08-04   8:00 AM     61952 acelpdec.ax
-a---      2004-08-04   8:00 AM    129536 acledit.dll
-a---      2004-08-04   8:00 AM    114688 aclui.dll
-a---      2004-08-04   8:00 AM    194048 activeds.dll
-a---      2004-08-04   8:00 AM    111104 activeds.tlb
-a---      2004-08-04   8:00 AM      4096 actmovie.exe
-a---      2004-08-04   8:00 AM    101888 actxprxy.dll
-a---      2003-02-21   6:50 PM    143150 admgmt.msc
-a---      2006-01-25   3:35 PM     53760 admparse.dll
<SPACE> 下一页; <CR> 下一行; Q 退出
...

```

如果您希望缓慢显示冗长的输出，则使用 **Out-Host -Paging** 命令这一管道元素将十分有用。在操作占用大量 CPU 时，此命令尤其有用。由于在要显示完整页时，处理权将转到 **Out-Host cmdlet**，因此在管道中继续执行该操作的 **cmdlet** 将暂停操作，直至下一页输出变为可用。如果使用 Windows 任务管理器来监视 Windows PowerShell 的 CPU 和内存使用率，您将会看到这一情况。

运行以下命令：**Get-ChildItem C:\Windows -Recurse**。将 CPU 和内存使用率与以下命令相比较：**Get-ChildItem C:\Windows -Recurse | Out-Host -Paging**。您将在屏幕上看到文本，但这是由于需要在控制台窗口中以文本形式表示对象。实际上，这只是在 Windows PowerShell 内部实际发生的情况的一种表现形式。假定以 **Get-Location cmdlet** 为例。如果当前位置是 C 驱动器的根目录，此时键入 **Get-Location**，将会显示以下输出：

```

PS> Get-Location

Path
----
C:\

```

如果 Windows PowerShell 通过管道传递文本，则发出的命令（例如，**Get-Location | Out-Host**）会将一组字符按其其在屏幕上的显示顺序从 **Get-Location** 传递到 **Out-Host**。换言之之，如果要忽略标题信息，**Out-Host** 会首先接收字符“C”，随后接收字符“:”，然后接收字符“\”。**Out-Host cmdlet** 无法确定与 **Get-Location cmdlet** 输出的字符相关联的含义。

Windows PowerShell 使用对象而不是使用文本来实现管道中的命令通信。从用户的角度来看，对象将相关信息打包为一种便于将信息作为一个单元进行操作的形式，并提取所需的特定项目。

**Get-Location** 命令不返回包含当前路径的文本。它返回一个称为 **PathInfo** 对象的信息包，其中包含当前路径以及其他一些信息。**Out-Host cmdlet** 随后会将此 **PathInfo** 对象发送到屏幕，然后由 Windows PowerShell 决定要显示的信息以及如何基于其格式规则来显示该信息。

实际上，作为设置数据在屏幕上的显示格式的过程的一部分，仅在该过程结尾才添加 **Get-Location cmdlet** 输出的标题信息。屏幕上所显示的内容为信息摘要，而不是输出对象的完整表现形式。

假定 Windows PowerShell 命令输出的信息多于我们在控制台窗口中看到的显示内容，如何才能检索不可见的元素？如何查看额外的数据？此外，如果希望采用非 Windows PowerShell 常用格式的其他格式来查看数据，该怎么办？

本章的其余部分将为您介绍如何发现特定 Windows PowerShell 对象的结构、如何选择特定项目并设置其格式以便于显示，以及如何将此信息发送到备选输出位置（例如，文件和打印机）。

## 查看对象结构 (Get-Member)

由于在 Windows PowerShell 中对象的作用十分重要，因此存在多个专为处理任意对象类型而设计的本机命令。其中最重要的一个即为 **Get-Member** 命令。

用于分析命令所返回对象的最简单的一种方法是通过管道将该命令的输出传递给 **Get-Member cmdlet**。**Get-Member cmdlet** 将显示对象类型的正式名称及其成员的完整列表。有时，所返回的元素数可能十分巨大。例如，进程对象可能有超过 100 个成员。

若要查看进程对象的所有成员并将输出分页显示，以便能够查看完整的该对象，请键入：

```
PS> Get-Process | Get-Member | Out-Host -Paging
```

此命令的输出将与以下所示类似：

```
TypeName: System.Diagnostics.Process

Name            MemberType      Definition
----            -
Handles          AliasProperty   Handles = Handlecount
Name             AliasProperty   Name = ProcessName
NPM              AliasProperty   NPM = NonpagedSystemMemorySize
PM              AliasProperty   PM = PagedMemorySize
VM              AliasProperty   VM = VirtualMemorySize
WS              AliasProperty   WS = WorkingSet
add_Disposed     Method          System.Void add_Disposed(Event...
...
```

通过筛选要查看的元素，您可以更好地使用这一长信息列表。使用 **Get-Member** 命令，您可以仅列出作为属性的成员。存在多种形式的属性。如果将 **Get-MemberMemberType** 参数设置为值 **Properties**，则该 cmdlet 将显示任一类型的属性。所得到的列表仍然很长，但更易于管理：

```
PS> Get-Process | Get-Member -MemberType Properties

TypeName: System.Diagnostics.Process


Name            MemberType      Definition
----            -
Handles          AliasProperty   Handles = Handlecount
Name             AliasProperty   Name = ProcessName
...
ExitCode         Property        System.Int32 ExitCode {get;}
...
Handle           Property        System.IntPtr Handle {get;}
...
CPU              ScriptProperty  System.Object CPU {get=$this.Total...
...
Path             ScriptProperty  System.Object Path {get=$this.Main...
...
```

 请注意：

**MemberType** 允许使用以下值：

**AliasProperty**、**CodeProperty**、**Property**、**NoteProperty**、**ScriptProperty**、**Properties**、**PropertySet**、**Method**、**CodeMethod**、**ScriptMethod**、**Methods**、**ParameterizedProperty**、**MemberSet** 和 **All**。

进程有超过 60 个属性。对于所有常见的对象，Windows PowerShell 通常仅显示其部分属性，这是因为显示其所有属性会生成难以管理的大量信息。

 请注意：

Windows PowerShell 通过使用其名称以 `.format.ps1xml` 结尾的 XML 文件中所存储的信息来确定如何显示对象类型。进程对象（即 `.NET System.Diagnostics.Process` 对象）的格式设置数据存储在 `PowerShellCore.format.ps1xml` 中。

如果需要查看 Windows PowerShell 默认情况下所显示属性之外的其他属性，则您需要自己设置输出数据的格式。可以使用格式 `cmdlet` 来执行此操作。

## 使用格式命令更改输出视图

Windows PowerShell 提供了一组 `cmdlet`，使用这些 `cmdlet` 您可以控制要为特定对象显示的属性。这些 `cmdlet` 的名称均以动词 **Format** 开头。它们允许您选择一个或多个要显示的属性。

**Format** `cmdlet` 包括 **Format-Wide**、**Format-List**、**Format-Table** 和 **Format-Custom**。本文中 will 介绍 **Format-Wide**、**Format-List** 和 **Format-Table** `cmdlet`。

每个格式 `cmdlet` 都具有默认属性；如果未指定要显示的特定属性，则会使用这些默认属性。此外，每个 `cmdlet` 都使用同一参数名称 **Property** 来指定要显示的属性。由于 **Format-Wide** 仅显示单个属性，因此其 **Property** 参数只获取单个值，但 **Format-List** 和 **Format-Table** 的属性参数将接受一组属性名称。

如果对 2 个正在运行的 Windows PowerShell 实例使用命令 **Get-Process -Name powershell**，所得到的输出将与以下所示类似：

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
995	9	30308	27996	152	2.73	2760	powershell
331	9	23284	29084	143	1.06	3448	powershell

本节的其余部分将探讨如何使用 **Format** `cmdlet` 来更改此命令的输出显示方式。

### 使用 **Format-Wide** 输出单个项目

默认情况下，**Format-Wide** `cmdlet` 仅显示对象的默认属性。与每个对象相关联的信息将显示在单独一列中：

```
PS> Get-Process -Name powershell | Format-Wide

powershell                                powershell
```

也可以指定非默认属性：

```
PS> Get-Process -Name powershell | Format-Wide -Property Id

2760                                3448
```

### 使用 **Column** 控制 **Format-Wide** 显示

使用 **Format-Wide** `cmdlet`，每次只能显示一个属性。这样对于显示每行仅显示一个元素的简单列表十分有用。若要获取简单列表，请键入下面命令来将 **Column** 参数的值设置为 1：

```
Get-Command Format-Wide -Property Name -Column 1
```

## 使用 Format-List 显示列表视图

**Format-List** cmdlet 以列表的形式显示对象，并在单独行上标记和显示每个属性：

```
PS> Get-Process -Name powershell | Format-List

Id       : 2760
Handles  : 1242
CPU      : 3.03125
Name     : powershell

Id       : 3448
Handles  : 328
CPU      : 1.0625
Name     : powershell
```

可以根据需要指定任意数量的属性：

```
PS> Get-Process -Name powershell | Format-List -Property ProcessName,FileVersion,
StartTime,Id

ProcessName      :powershell
FileVersion      : 1.0.9567.1
StartTime       : 2006-05-24 13:42:00
Id              : 2760

ProcessName      :powershell
FileVersion      : 1.0.9567.1
StartTime       : 2006-05-24 13:54:28
Id              : 3448
```

## 使用 Format-List 和通配符来获取详细信息

**Format-List** cmdlet 允许您将通配符用作其 **Property** 参数的值。这样可以显示详细信息。通常，对象所包含的信息多于您需要的信息，因此，默认情况下 Windows PowerShell 将不会显示所有属性值。若要显示对象的所有属性，请使用 **Format-List -Property \*** 命令。以下命令将为单个进程生成超过 60 行输出：

```
Get-Process -Name powershell | Format-List -Property *
```

尽管 **Format-List** 命令对于显示详细信息十分有用，但如果希望获得包含多个项目的输出概览，则较简单的表格格式视图通常会更有用。

## 使用 Format-Table 显示表格格式输出

如果使用 **Format-Table** cmdlet（未指定任何属性名）来设置 **Get-Process** 命令的输出格式，所获得的输出效果将与不执行任何格式设置的情况完全相同。这是因为与大多数 Windows PowerShell 对象一样，进程通常是以表格格式显示的。

```
PS> Get-Process -Name powershell | Format-Table
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
1488	9	31568	29460	152	3.53	2760	powershell
332	9	23140	632	141	1.06	3448	powershell

## 改进 Format-Table 输出 (AutoSize)

尽管表格格式的视图适用于显示大量可比较信息，但其可能难以判断显示区域是否过窄而无法容纳数据。例如，如果试图显示进程路径、ID、名称和公司，则会导致进程路径和公司列的输出被截断：

```
PS> Get-Process -Name powershell | Format-Table -Property Path,Name,Id,Company

Path                Name                Id Company
----                -
C:\Program Files... powershell          2836 Microsoft Corpor...
```

如果在运行 **Format-Table** 命令时指定了 **AutoSize** 参数，则 Windows PowerShell 将基于要显示的实际数据来计算列宽。这样会使 **Path** 列可读，但公司列仍处于截断状态：

```
PS> Get-Process -Name powershell | Format-Table -Property Path,Name,Id,Company -
AutoSize

Path                Name                Id Company
----                -
C:\Program Files\Windows PowerShell\v1.0\powershell.exe powershell 2836 Micr...
```

**Format-Table** cmdlet 仍然可能会截断数据，但截断只会发生在屏幕的末尾部分。除最后一个显示的属性外，会将其他属性指定为其最长数据元素所需的宽度以进行正常显示。如果在 **Property** 值列表中将 **Path** 和 **Company** 交换位置，您将会看到公司名称为可见的，但路径被截断：

```
PS> Get-Process -Name powershell | Format-Table -Property Company,Name,Id,Path -
AutoSize

Company                Name                Id Path
-----                -
Microsoft Corporation powershell 2836 C:\Program Files\Windows PowerShell\v1...
```

**Format-Table** 命令假定离属性列表开头越近的属性的重要性越高。因此，它会试图完全显示最靠近开头的属性。如果 **Format-Table** 命令无法显示所有属性，则会从显示中删除部分列并发出相应警告。如果将 **Name** 作为列表中的最后一个属性，则会看到此类行为：

```
PS> Get-Process -Name powershell | Format-Table -Property Company,Path,Id,Name -
AutoSize

警告：列“Name”无法显示，已被删除。

Company                Path                I
d
-----                -
Microsoft Corporation C:\Program Files\Windows PowerShell\v1.0\powershell.exe 6
```

在以上输出中，ID 列被截断以便能容纳在列表中，并且堆叠了列标题。自动调整各列大小，并不一定总会实现您需要的效果。



## Format-Table 输出在列中换行 (Wrap)

通过使用 **Wrap** 参数，可以将冗长的 **Format-Table** 数据强行在其显示列中换行。单独使用 **Wrap** 参数不一定能实现您的预期效果，这是因为如果不同时指定 **AutoSize**，则会使用默认设置：

```
PS> Get-Process -Name powershell | Format-Table -Wrap -Property Name,Id,Company,Path
```

Name	Id	Company	Path
-----	--	-----	----
powershell	2836	Microsoft Corporation	C:\Program Files\Windows PowerShel
on			l\v1.0\powershell.exe

单独使用 **Wrap** 参数的一个优点在于它不会极大降低处理速度。如果在执行大型目录系统的递归文件列表时使用 **AutoSize**，则在显示第一个输出项之前可能需要花费很长时间并占用大量内存。

如果不考虑系统负载，则将 **AutoSize** 与 **Wrap** 参数一起使用会取得不错的效果。与指定 **AutoSize** 而不指定 **Wrap** 参数时的情况一样，始终会为初始列分配所需的宽度以便在一行内显示项目。唯一的不同之处在于：如有必要，最后一列将进行换行：

```
PS> Get-Process -Name powershell | Format-Table -Wrap -AutoSize -Property Name,Id,Company,Path
```

Name	Id	Company	Path
-----	--	-----	----
powershell	2836	Microsoft Corporation	C:\Program Files\Windows PowerShell\v1.0\powershell.exe

如果先指定最宽的列，则可能无法显示某些列，因此，最安全的做法是先指定最小的数据元素。在以下示例中，我们先指定最宽的路径元素，在这种情况下，即使通过换行也仍然无法显示最后的 **Name** 列：

```
PS> Get-Process -Name powershell | Format-Table -Wrap -AutoSize -Property Path,Id,Company,Name
```

警告：列“Name”无法显示，已被删除。

Path	Id	Company
-----	--	-----
C:\Program Files\Windows PowerShell\v1.0\powershell.exe	2836	Microsoft Corporation

## 组织表输出 (-GroupBy)

用于表格格式输出控制的另一个有用参数是 **GroupBy**。越长的表格格式列表可能越难以进行比较。使用 **GroupBy** 参数可以基于属性值对输出进行分组。例如，我们可以按公司对进程进行分组，从而忽略属性列表中的公司值来更轻松地进行检查：

```
PS> Get-Process -Name powershell | Format-Table -Wrap -AutoSize -Property Name,Id,Path -GroupBy Company
```

Company	Name	Id	Path
Microsoft Corporation	powershell	2836	C:\Program Files\Windows PowerShell\v1.0\powershell.exe

Name	Id	Path
powershell	1956	C:\Program Files\Windows PowerShell\powershell.exe
powershell	2656	C:\Program Files\Windows PowerShell\powershell.exe

## 使用 Out-\* Cmdlet 重定向数据

Windows PowerShell 为您提供了多个用于直接控制数据输出的 cmdlet。这些 cmdlet 具有两个重要的共同特征。

第一，它们通常都将数据转换为某种文本形式。这是因为它们要将数据输出到需要接受文本输入的系统组件。这意味着它们需要以文本形式表示对象。因此，文本格式将与 Windows PowerShell 控制台窗口中所显示的格式相同。

第二，这些 cmdlet 都使用 Windows PowerShell 动词 **Out**，这是因为它们要将信息从 Windows PowerShell 中发送到其他位置。**Out-Host** cmdlet 也不例外：主机窗口显示位于 Windows PowerShell 之外。这一特征十分重要，因为从 Windows PowerShell 发出数据时，实际上将删除该数据。如果试图创建将数据分页发送到主机窗口的管道，然后尝试将其格式设置为列表，则会看到此类情况，如下所示：

```
PS> Get-Process | Out-Host -Paging | Format-List
```

您可能希望该命令以列表格式来显示进程信息页面。实际上，它将显示默认的表格格式列表：

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
101	5	1076	3316	32	0.05	2888	alg
...							
618	18	39348	51108	143	211.20	740	explorer
257	8	9752	16828	79	3.02	2560	explorer
...							

<SPACE> 下一页; <CR> 下一行; Q 退出

...

**Out-Host** cmdlet 将数据直接发送到控制台，因此 **Format-List** 命令将不会收到任何要进行格式设置的输入项。

若要构建此命令的结构，正确方式是将 **Out-Host** cmdlet 放在管道末尾，如下所示。这样将导致先在列表中设置进程数据的格式，然后再进行分页和显示。

```
PS> Get-Process | Format-List | Out-Host -Paging
```

```
Id      : 2888
Handles : 101
CPU     : 0.046875
Name    : alg
...

Id      : 740
Handles : 612
CPU     : 211.703125
Name    : explorer
...

Id      : 2560
```

```
Handles : 257
CPU      : 3.015625
Name     : explorer
...
<SPACE> 下一页; <CR> 下一行; Q 退出
...
```

此方式适用于所有 **Out cmdlet**。**Out cmdlet** 应始终出现在管道末尾。

 请注意:

所有 **Out cmdlet** 都以文本形式呈现输出，并使用控制台窗口的有效格式设置（包含行长度限制）进行显示。

## 对控制台输出进行分页 (Out-Host)

默认情况下，Windows PowerShell 会将数据发送到主机窗口，这与 **Out-Host cmdlet** 的作用完全相同。正如上文中所述，**Out-Host cmdlet** 的主要用途是将数据进行分页。例如，以下命令使用 **Out-Host** 对 **Get-Command cmdlet** 的输出进行分页：

```
PS> Get-Command | Out-Host -Paging
```

也可以使用 **more** 函数来对数据进行分页。在 Windows PowerShell 中，**more** 是一个称为 **Out-Host -Paging** 的函数。以下命令演示了如何使用 **more** 函数来对 **Get-Command** 的输出进行分页：

```
PS> Get-Command | more
```

如果将一个或多个文件名用作 **more** 函数的参数，则该函数将读取指定的文件并将其内容分页发送到主机：

```
PS> more c:\boot.ini
[boot loader]
timeout=5
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
...
```

## 放弃输出 (Out-Null)

**Out-Null cmdlet** 是专为立即放弃其接收的任何输入而设计的。如果希望放弃作为运行命令的副产品而获得的不需要的数据，则此命令十分有用。键入以下命令后，将不会返回任何输出：

```
PS> Get-Command | Out-Null
```

**Out-Null cmdlet** 不会放弃错误输出。例如，如果输入以下命令，将显示一条消息，通知您 Windows PowerShell 无法识别 “Is-NotACommand”：

```
PS> Get-Command Is-NotACommand | Out-Null
Get-Command :无法将“Is-NotACommand”识别为 cmdlet、函数、可运行程序
或脚本文件。
所在行:1 字符:12
+ Get-Command <<<< Is-NotACommand | Out-Null
```

## 打印数据 (Out-Printer)

可以使用 **Out-Printer** cmdlet 来打印数据。如果未提供打印机名称，则 **Out-Printer** cmdlet 将使用默认打印机。通过指定打印机的显示名称，您可以使用任何基于 Windows 的打印机。无需指定任何种类的打印机端口映射，甚至无需指定真实的物理打印机。例如，如果安装了 Microsoft Office 文档图像工具，可以通过键入以下命令将数据发送到图像文件：

```
PS> Get-Command Get-Command | Out-Printer -Name "Microsoft Office Document Image Writer"
```

## 保存数据 (Out-File)

通过使用 **Out-File** cmdlet，可以将输出发送到文件而不是控制台窗口。以下命令行将一个进程列表发送到文件 **C:\temp\processlist.txt**：

```
PS> Get-Process | Out-File -FilePath C:\temp\processlist.txt
```

如果习惯于传统的输出重定向，则使用 **Out-File** cmdlet 的结果可能与您的预期结果不同。若要了解此命令的行为，您必须了解运行 **Out-File** cmdlet 的上下文。

默认情况下，**Out-File** cmdlet 将创建 Unicode 文件。从长远的角度来看，这是最佳的默认值，但这意味着预期使用 ASCII 文件的工具将无法正确处理该默认输出格式。可以使用 **Encoding** 参数来将默认输出格式更改为 ASCII：

```
PS> Get-Process | Out-File -FilePath C:\temp\processlist.txt -Encoding ASCII
```

**Out-file** 将文件内容的格式设置为与控制台输出的格式一样。这样将导致在大多数情况下输出将像在控制台窗口中一样发生截断。例如，如果运行以下命令：

```
PS> Get-Command | Out-File -FilePath c:\temp\output.txt
```

输出将与以下所示类似：

CommandType	Name	Definition
-----	----	-----
Cmdlet	Add-Content	Add-Content [-Path] <String[...
Cmdlet	Add-History	Add-History [[-InputObject] ...
...		

若要获得不强行进行换行以匹配屏幕宽度的输出，可以使用 **Width** 参数来指定行宽。由于 **Width** 是 32 位整数参数，因此其最大值为 2147483647。键入以下命令可以将行宽设置为此最大值：

```
Get-Command | Out-File -FilePath c:\temp\output.txt -Width 2147483647
```

如果希望按照控制台上所显示的格式对输出进行保存，则 **Out-File** cmdlet 尤其有用。若要更好地控制输出格式，您需要使用更高级的工具。我们将在下一章介绍这些工具，并提供有关对象操作的一些详细信息。

# Windows PowerShell 导航

文件夹是 Windows 资源管理器界面、Cmd.exe 和 UNIX 工具（例如 BASH）中令人熟悉的组织功能。文件夹（通常称为目录）是用于组织文件和其他目录的有用概念。UNIX 系列操作系统扩展了这一概念，它们将所有可能的项目都视为文件；特定的硬件和网络连接将作为文件显

示在特定文件夹中。此方法无法确保内容是否可由特定应用程序读取或使用，但这样可以更轻松地查找特定项目。用于对文件和文件夹进行枚举或搜索的工具也使用这些设备。此外，还可以使用表示特定项目的文件路径来定位到该项目。

同样，Windows PowerShell 基础结构支持诸如标准 Microsoft Windows 磁盘驱动器或 UNIX 文件系统之类的可进行导航的几乎所有项目都公开为 Windows PowerShell 驱动器。Windows PowerShell 驱动器不一定表示真实的本地驱动器或网络驱动器。本章主要介绍文件系统的导航，但这些概念也适用于与文件系统无关的 Windows PowerShell 驱动器。

## 在 Windows Powershell 中管理当前位置

在 Windows 资源管理器中导航文件夹系统时，通常具有特定的工作位置，即当前打开的文件夹。通过单击当前文件夹中的项目，可以轻松地进行操作。对于 `Cmd.exe` 之类的命令行界面，当您位于某特定文件所在的同一文件夹中时，可以通过指定相对简短的名称来对其进行访问，而无需指定该文件的完整路径。当前目录也称为工作目录。

Windows PowerShell 使用名词 **Location** 来引用工作目录，并实施一系列 `cmdlet` 来对您的位置进行检查和操作。

### 获取当前位置 (Get-Location)

若要确定当前目录位置的路径，请输入 **Get-Location** 命令：

```
PS> Get-Location
Path
----
C:\Documents and Settings\PowerUser
```

 请注意：

`Get-Location cmdlet` 与 BASH 外壳程序中的 `pwd` 命令类似。`Set-Location cmdlet` 与 `Cmd.exe` 中的 `cd` 命令类似。

### 设置当前位置 (Set-Location)

**Get-Location** 命令与 **Set-Location** 命令一起使用。使用 **Set-Location** 命令，您可以指定当前目录位置。

```
PS> Set-Location -Path C:\Windows
```

输入命令之后，您将会发现没有收到有关命令效果的任何直接反馈。大多数执行操作的 Windows PowerShell 命令会产生少量输出或没有输出，因为输出并不总是有用的。若要验证在输入 **Set-Location** 命令时是否已成功进行了目录更改，请在输入 **Set-Location** 命令时包括 **-PassThru** 参数：

```
PS> Set-Location -Path C:\Windows -PassThru
Path
----
C:\WINDOWS
```

在 Windows PowerShell 中，**-PassThru** 参数可以与许多 `Set` 命令一起使用，以便在没有默认输出的情况下返回有关结果的信息。

可以采用与大多数 UNIX 和 Windows 命令外壳程序中所用的相同方式来指定当前位置的相对路径。在标准的相对路径表示法中，句点 (.) 表示当前文件夹，两个句点 (..) 表示当前位置的父目录。

例如，如果您在 **C:\Windows** 文件夹中，则句点 (.) 表示 **C:\Windows**，而双句点 (..) 表示 **C:**。通过键入以下命令可以从当前位置更改为 C: 驱动器的根目录：

```
PS> Set-Location -Path ..-PassThru
Path
----
C:\
```

此方法同样适用于非文件系统驱动器的 Windows PowerShell 驱动器（例如，**HKLM:**）。通过键入以下命令可以将您的位置设置为注册表中的 HKLM\Software 项：

```
PS> Set-Location -Path HKLM:\SOFTWARE -PassThru

Path
----
HKLM:\SOFTWARE
```

然后，可以使用相对路径来将目录位置更改到父目录，即 Windows PowerShell HKLM: 驱动器的根目录：

```
PS> Set-Location -Path ..-PassThru

Path
----
HKLM:\
```

可以键入 **Set-Location** 或使用 **Set-Location** 的任何内置 Windows PowerShell 别名 (**cd**、**chdir**、**sl**)。例如：

```
cd -Path C:\Windows
```

```
chdir -Path ..-PassThru
```

```
sl -Path HKLM:\SOFTWARE -PassThru
```


## 保存和撤回最近的位置（Push-Location 和 Pop-Location）

更改位置时，有必要对曾经处于的位置进行跟踪，并且能够返回以前的位置。Windows PowerShell 中的 **Push-Location** cmdlet 可以创建您曾经处于的目录路径的有序历史记录（“堆栈”），并且可以使用补充的 **Pop-Location** cmdlet 来逐步退回目录路径的历史记录。

例如，Windows PowerShell 会话通常从用户的主目录中开始。

```
PS> Get-Location

Path
----
C:\Documents and Settings\PowerUser
```

 请注意：

在许多编程设置（包括 .NET）中，术语“堆栈”具有特殊的含义。与物料的物理堆栈相似，最后一个放入堆栈的项目将是您可以从该堆栈中取出的第一个项目。如果用通俗语言

来表示向堆栈中添加项目，即称为将项目“压入”堆栈。如果用通俗语言来表示从堆栈中取出项目，即称为将项目“弹出”堆栈。

若要将当前位置压入堆栈，然后转到 **Local Settings** 文件夹，请键入：

```
PS> Push-Location -Path "Local Settings"
```

然后可以通过键入以下命令，将 **Local Settings** 位置压入堆栈，并转到 **Temp** 文件夹：

```
PS> Push-Location -Path Temp
```

通过输入 **Get-Location** 命令，可以验证是否已经更改了目录：

```
PS> Get-Location

Path
----
C:\Documents and Settings\PowerUser\Local Settings\Temp
```

然后，可以通过输入 **Pop-Location** 命令来弹出最近访问过的目录，并通过输入 **Get-Location** 命令来验证该更改：

```
PS> Pop-Location
PS> Get-Location

Path
----
C:\Documents and Settings\me\Local Settings
```

与 **Set-Location** cmdlet 相同，输入 **Pop-Location** cmdlet 时可以包括 **-PassThru** 参数以显示所输入的目录：

```
PS> Pop-Location -PassThru

Path
----
C:\Documents and Settings\PowerUser
```

还可以使用位置 cmdlet 来处理网络路径。如果有一个名为 **FS01** 的服务器，该服务器有一个名为 **Public** 的共享区，则可以通过键入以下命令来更改位置：

```
Set-Location \\FS01\Public
```

或

```
Push-Location \\FS01\Public
```

可以使用 **Push-Location** 和 **Set-Location** 命令将位置更改为任何可用驱动器。例如，如果有一个本地 CD-ROM 驱动器，其驱动器号为 **D**，并且包含数据 **CD**，则可以通过输入 **Set-Location D:** 命令将位置更改为该 CD 驱动器。

如果该驱动器为空，您将获得以下错误消息：

```
PS> Set-Location D:
Set-Location :找不到路径“D:\”，因为该路径不存在。
```

使用命令行界面时，用 **Windows** 资源管理器来检查可用物理驱动器并不太方便。而且，**Wind**

Windows 资源管理器不会为您显示所有 Windows PowerShell 驱动器。Windows PowerShell 提供了一组用于操作 Windows PowerShell 驱动器的命令，稍后我们将介绍这些命令。

## 管理 Windows PowerShell 驱动器

Windows PowerShell 驱动器是一种数据存储位置，您可以像访问 Windows PowerShell 中的文件系统驱动器一样对其进行访问。Windows PowerShell 提供程序会为您自动创建某些驱动器，例如，文件系统驱动器（包括 C: 和 D:）、注册表驱动器（HKCU: 和 HKLM:）以及证书驱动器（Cert:），您还可以创建您自己的 Windows PowerShell 驱动器。这些驱动器十分有用，但它们仅在 Windows PowerShell 中可用。无法使用其他 Windows 工具（例如，Windows 资源管理器或 Cmd.exe）来访问这些驱动器。

Windows PowerShell 使用名词 **PSDrive** 来表示用于处理 Windows PowerShell 驱动器的命令。若要获得会话中的 Windows PowerShell 驱动器列表，请使用 **Get-PSDrive** cmdlet。

```
PS> Get-PSDrive
```

Name	Provider	Root	CurrentLocation
A	FileSystem	A:\	
Alias	Alias		
C	FileSystem	C:\	...And Settings\me
cert	Certificate	\	
D	FileSystem	D:\	
Env	Environment		
Function	Function		
HKCU	Registry	HKEY_CURRENT_USER	
HKLM	Registry	HKEY_LOCAL_MACHINE	
Variable	Variable		

尽管根据您系统中驱动器的不同，此处所显示的驱动器会有所不同，但该列表的外观与上面显示的 **Get-PSDrive** 命令的输出类似。

文件系统驱动器是 Windows PowerShell 驱动器的子集。可以按 Provider 列中的 FileSystem 条目来标识文件系统驱动器。（Windows PowerShell 中的文件系统驱动器由 Windows PowerShell FileSystem 提供程序支持。）

若要查看 **Get-PSDrive** cmdlet 的语法，请键入 **Get-Command** 命令和 **Syntax** 参数：

```
PS> Get-Command -Name Get-PSDrive -Syntax
Get-PSDrive [[-Name] <String[]>] [-Scope <String>] [-PSProvider <String[]>] [-V
erbose] [-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-
OutVariable <String>] [-OutBuffer <Int32>]
```

使用 **PSProvider** 参数，您可以仅显示由特定提供程序支持的 Windows PowerShell 驱动器。例如，若要仅显示由 Windows PowerShell FileSystem 提供程序支持的 Windows PowerShell 驱动器，请键入 **Get-PSDrive** 命令以及 **PSProvider** 参数和 **FileSystem** 值：

```
PS> Get-PSDrive -PSProvider FileSystem
```

Name	Provider	Root	CurrentLocation
A	FileSystem	A:\	
C	FileSystem	C:\	...nd Settings\PowerUser
D	FileSystem	D:\	



若要查看表示注册表配置单元的 Windows PowerShell 驱动器，请使用 **PSPProvider** 参数，以便仅显示由 Windows PowerShell Registry 提供程序支持的 Windows PowerShell 驱动器：  
PS> Get-PSDrive -PSPProvider Registry

Name	Provider	Root	CurrentLocation
-----	-----	----	-----
HKCU	Registry	HKEY_CURRENT_USER	
HKLM	Registry	HKEY_LOCAL_MACHINE	

还可以将标准的位置 cmdlet 用于 Windows PowerShell 驱动器：  
PS> Set-Location HKLM:\SOFTWARE  
PS> Push-Location .\Microsoft  
PS> Get-Location

Path  
-----  
HKLM:\SOFTWARE\Microsoft

添加新的 Windows PowerShell 驱动器 (New-PSDrive)

通过使用 **New-PSDrive** 命令，可以添加您自己的 Windows PowerShell 驱动器。若要获取 **New-PSDrive** 命令的语法，请输入 **Get-Command** 命令和 **Syntax** 参数：

```
PS> Get-Command -Name New-PSDrive -Syntax
New-PSDrive [-Name] <String> [-PSPProvider] <String> [-Root] <String> [-Description <String>] [-Scope <String>] [-Credential <PSCredential>] [-Verbose] [-Debug] [-ErrorAction <ActionPreference>] [-ErrorVariable <String>] [-OutVariable <String>] [-OutBuffer <Int32>] [-WhatIf] [-Confirm]
```

若要创建新的 Windows PowerShell 驱动器，您必须提供三个参数：

驱动器的名称（可以使用任何有效的 Windows PowerShell 名称）


PSPProvider（使用 “FileSystem”表示文件系统位置，使用 “Registry”表示注册表位置）

根目录，即新驱动器的根目录路径

例如，可以创建名为 “Office”的驱动器，该驱动器映射到计算机上包含 Microsoft Office 应用程序的文件夹，例如，C:\Program Files\Microsoft Office\OFFICE11。若要创建该驱动器，请键入以下命令：

```
PS> New-PSDrive -Name Office -PSPProvider FileSystem -Root "C:\Program Files\Microsoft Office\OFFICE11"

Name      Provider      Root      CurrentLocation
-----
Office    FileSystem    C:\Program Files\Microsoft Offic...
```

 请注意：

通常，路径不区分大小写。

引用新 Windows PowerShell 驱动器的方式与引用所有 Windows PowerShell 驱动器的方式一样：该驱动器的名称，后跟冒号 (:)。

Windows PowerShell 驱动器可以简化很多任务。例如，Windows 注册表中某些最重要的注册表项具有很长的路径，从而难以对其进行访问和记忆。关键的配置信息驻留在 **HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion** 下。若要查看和更改 CurrentVersion 注册表项中的项目，您可以通过键入以下命令来创建一个根目录位于该注

册表项中的 Windows PowerShell 驱动器:

```
PS> New-PSDrive -Name cvkey -PSProvider Registry -Root HKLM\Software\Microsoft\Windows\CurrentVersion
```

Name	Provider	Root	CurrentLocation
cvkey	Registry	HKLM\Software\Microsoft\Windows\...	

然后, 可以像对其他任何驱动器一样将其位置更改为 **cvkey:** 驱动器:

```
PS> cd cvkey:
```

或者:

```
PS> Set-Location cvkey:-PassThru
```

```
Path
```

```
-----  
cvkey:\
```

**New-PSDrive** cmdlet 可以将新驱动器仅添加到当前控制台会话中。如果退出控制台或关闭 Windows PowerShell 窗口, 则新驱动器将丢失。若要保存 Windows PowerShell 驱动器, 请使用 **Export-Console** cmdlet 导出当前控制台, 然后使用 **PowerShell.exe PSConsoleFile** 参数将其导入新会话中。或者, 将新驱动器添加到您的 Windows PowerShell 配置文件中。

## 删除 Windows PowerShell 驱动器 (Remove-PSDrive)

通过使用 **Remove-PSDrive** cmdlet, 可以从 Windows PowerShell 中删除驱动器。**Remove-PSDrive** cmdlet 易于使用; 若要删除特定的 Windows PowerShell 驱动器, 您只需提供该 Windows PowerShell 驱动器名称。

例如, 如果按 **New-PSDrive** 主题中所示添加了 **Office:** Windows PowerShell 驱动器, 则可以通过键入以下命令来将其删除:

```
PS> Remove-PSDrive -Name Office
```

若要删除 **New-PSDrive** 主题中出现的 **cvkey:** Windows PowerShell 驱动器, 请使用以下命令:

```
PS> Remove-PSDrive -Name cvkey
```

删除 Windows PowerShell 驱动器十分容易, 但当您位于该驱动器中时, 则无法将其删除。例如:

```
PS> cd office:  
PS Office:\> remove-psdrive -name office  
Remove-PSDrive :无法删除驱动器“Office”, 因为它在使用中。  
所在行:1 字符:15  
+ remove-psdrive <<<< -name office
```

## 在 Windows PowerShell 外部添加和删除驱动器

Windows PowerShell 可检测在 Windows 中添加或删除的文件系统驱动器, 包括映射的网络驱动器、附加的 USB 驱动器以及通过以下方式删除的驱动器: 使用 **net use** 命令或 Windows Script Host (WSH) 脚本中的 **WScript.NetworkMapNetworkDrive** 和 **RemoveNetworkDrive** 方法。

## 处理文件、文件夹和注册表项

Windows PowerShell 使用名词项来表示在 Windows PowerShell 驱动器中找到的数据。当处理 Windows PowerShell FileSystem 提供程序时，项可以是文件、文件夹或 Windows PowerShell 驱动器。在大多数管理设置中，列出并处理这些项是一项重要的基本任务，因此我们应对这些任务进行详细的讨论。

### 枚举文件、文件夹和注册表项 (Get-ChildItem)

由于从特定位置获取项集合是非常常见的任务，因此 **Get-ChildItem** cmdlet 旨在返回容器（例如文件夹）中找到的所有项。

若要返回文件夹 C:\Windows 中直接包含的所有文件和文件夹，请键入：

```
PS> Get-ChildItem -Path C:\Windows
Directory:Microsoft.Windows PowerShell.Core\FileSystem::C:\Windows
Mode                LastWriteTime         Length Name
----                -
-a---             2006-05-16   8:10 AM             0 0.log
-a---             2005-11-29   3:16 PM             97 acc1.txt
-a---             2005-10-23  11:21 PM          3848 actsetup.log
...
```

在 **Cmd.exe** 中输入 **dir** 命令或在 UNIX 命令外壳程序中输入 **ls** 命令时，所列出的内容与您所看到的内容相似。

可以使用 **Get-ChildItem** cmdlet 参数来执行非常复杂的列表操作。我们将看到以下几种情况。

您可以通过键入以下命令来查看 **Get-ChildItem** cmdlet 的语法：

```
PS> Get-Command -Name Get-ChildItem -Syntax
```

可以将这些参数进行混合和匹配以获得较高度度的自定义输出。

### 列出所有包含的项 (-Recurse)

若要查看 Windows 文件夹内部的项及其子文件夹中包含的所有项，请使用 **Get-ChildItem** 的 **Recurse** 参数。列表操作将显示 Windows 文件夹中的所有内容及其子文件夹中的所有项。例如：

```
PS> Get-ChildItem -Path C:\WINDOWS -Recurse

Directory:Microsoft.Windows PowerShell.Core\FileSystem::C:\WINDOWS
Directory:Microsoft.Windows PowerShell.Core\FileSystem::C:\WINDOWS\AppData
Mode                LastWriteTime         Length Name
----                -
-a---             2004-08-04   8:00 AM          1852416 AcGenral.dll
...
```

### 按名称筛选项 (-Name)

若要仅显示项名称，请使用 **Get-Childitem** 的 **Name** 参数：

```
PS> Get-ChildItem -Path C:\WINDOWS -Name
addins
AppPatch
```

```
assembly
...
```

## 强制列出隐藏项 (-Force)

通常，在 Windows 资源管理器或 Cmd.exe 中可见的项均不会显示在 **Get-ChildItem** 命令的输出中。若要显示隐藏项，请使用 **Get-ChildItem** 的 **Force** 参数。例如：

```
Get-ChildItem -Path C:\Windows -Force
```

由于可强制覆盖 **Get-ChildItem** 命令的正常行为，因此此参数命名为 **Force**。**Force** 参数使用广泛，它可强制执行 cmdlet 非正常执行的操作，但该参数不执行危及系统安全的任何操作。

## 将项名称与通配符匹配

**Get-ChildItem** 命令在要列出的项路径中接受通配符。

由于通配符匹配是通过 Windows PowerShell 引擎处理的，因此接受通配符的所有 cmdlet 都可使用相同的表示法，并具有相同的匹配行为。Windows PowerShell 通配符表示法包括：

星号 (\*) 与零或更多出现的字符匹配。

问号 (?) 仅与一个字符匹配。

左方括号 ([]) 字符和右方括号 (]) 字符可括住一组要进行匹配的字符。

此处为如何应用通配符规范的一些示例。

若要查找 Windows 目录中具有后缀 **.log**，且基名称只有五个字符的所有文件，请输入以下命令：

```
PS> Get-ChildItem -Path C:\Windows\?????.log
Directory:Microsoft.Windows PowerShell.Core\FileSystem::C:\Windows
Mode                LastWriteTime         Length Name
----                -
...
-a---             2006-05-11   6:31 PM         204276 ocgen.log
-a---             2006-05-11   6:31 PM         22365  ocmsn.log
...
-a---             2005-11-11   4:55 AM           64  setup.log
-a---             2005-12-15   2:24 PM        17719 VxSDM.log
...
```

若要查找 Windows 目录中以字母 **x** 开头的所有文件，请键入：

```
Get-ChildItem -Path C:\Windows\x*
```

若要查看其文件名以 **x** 或 **z** 开头的所有文件，请键入：

```
Get-ChildItem -Path C:\Windows\[xz]*
```

## 排除项 (-Exclude)

通过使用 **Get-ChildItem** 的 **Exclude** 参数可以排除特定的项。即，在单语句中执行复杂的筛选操作。

例如，假设您要在 System32 文件夹中查找 Windows 时间服务 DLL，但您只记得 DLL 名称是以“W”开头并且其中有数字“32”。

使用类似表达式 **w\*32\*.dll** 即可找到符合该条件的所有 DLL，但该表达式还会返回其名称中包含 “95”或 “16”的与 Windows 95 和 16 位 Windows 兼容的 DLL。您可以使用模式为 **\*[9516]\*** 的 **Exclude** 参数省略名称中具有这些数字的所有文件：

```
PS> Get-ChildItem -Path C:\WINDOWS\System32\w*32*.dll -Exclude *[9516]*
```

Directory:Microsoft.PowerShell.Core\FileSystem::C:\WINDOWS\System32

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	2004-08-04 8:00 AM	174592	w32time.dll
-a---	2004-08-04 8:00 AM	22016	w32topl.dll
-a---	2004-08-04 8:00 AM	101888	win32spl.dll
-a---	2004-08-04 8:00 AM	172032	wldap32.dll
-a---	2004-08-04 8:00 AM	264192	wow32.dll
-a---	2004-08-04 8:00 AM	82944	ws2_32.dll
-a---	2004-08-04 8:00 AM	42496	wsnmp32.dll
-a---	2004-08-04 8:00 AM	22528	wsock32.dll
-a---	2004-08-04 8:00 AM	18432	wtsapi32.dll

混合使用 **Get-ChildItem** 参数

您可以在同一命令中使用 **Get-ChildItem** cmdlet 的多个参数。在混合使用参数之前，请确保您了解通配符匹配。例如，以下命令将不会返回任何结果：

```
PS> Get-ChildItem -Path C:\Windows\*.dll -Recurse -Exclude [a-y]*.dll
```

即使 Windows 文件夹中包含两个以字母 “z”开头 DLL，也不会返回任何结果。不返回任何结果是由于我们将通配符指定为该路径的一部分。即使该命令是递归的，但 **Get-ChildItem** cmdlet 仍将这些项限制为 Windows 文件夹中其名称以 “.dll”结尾的项。若要为其名称与特定模式相匹配的文件指定递归搜索，请使用 **-Include** 参数。

```
PS> Get-ChildItem -Path C:\Windows -Include *.dll -Recurse -Exclude [a-y]*.dll
```

Directory:Microsoft.Windows PowerShell.Core\FileSystem::C:\Windows\System32\Setup

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	2004-08-04 8:00 AM	8261	zoneoc.dll

Directory:Microsoft.Windows PowerShell.Core\FileSystem::C:\Windows\System32

Mode	LastWriteTime	Length	Name
----	-----	-----	----
-a---	2004-08-04 8:00 AM	337920	zipfldr.dll

## 直接对项进行操作

在 Windows PowerShell 驱动器中看到的元素（例如文件系统驱动器中的文件和文件夹）和 Windows PowerShell 注册表驱动器中的注册表项在 Windows PowerShell 中均称为项。用于处理这些项的 cmdlet 的名称中具有名词项。

**Get-Command -Noun Item** 命令的输出显示，存在九个 Windows PowerShell 项 cmdlet。

```
PS> Get-Command -Noun Item
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Clear-Item	Clear-Item [-Path] <String[]>...
Cmdlet	Copy-Item	Copy-Item [-Path] <String[]>...
Cmdlet	Get-Item	Get-Item [-Path] <String[]> ...
Cmdlet	Invoke-Item	Invoke-Item [-Path] <String[]>...
Cmdlet	Move-Item	Move-Item [-Path] <String[]>...
Cmdlet	New-Item	New-Item [-Path] <String[]> ...
Cmdlet	Remove-Item	Remove-Item [-Path] <String[]>...
Cmdlet	Rename-Item	Rename-Item [-Path] <String>...
Cmdlet	Set-Item	Set-Item [-Path] <String[]> ...

## 创建新项 (New-Item)

若要在文件系统中创建新项，请使用 **New-Item** cmdlet。包含带有指向该项的路径的 **Path** 参数和具有“file”或“directory”值的 **ItemType** 参数。

例如，若要在 C:\Temp 目录中创建名为“New.Directory”的新目录，请键入：

```
PS> New-Item -Path c:\temp\New.Directory -ItemType Directory

Directory:Microsoft.Windows PowerShell.Core\FileSystem::C:\temp

Mode                LastWriteTime         Length Name
----                -
d-----          2006-05-18  11:29 AM                New.Directory
```

若要创建文件，请将 **ItemType** 参数值更改为“file”。例如，若要在 New.Directory 目录中创建名为“file1.txt”的文件，请键入：

```
PS> New-Item -Path C:\temp\New.Directory\file1.txt -ItemType file

Directory:Microsoft.Windows PowerShell.Core\FileSystem::C:\temp\New.Directory

Mode                LastWriteTime         Length Name
----                -
-a-----          2006-05-18  11:44 AM              0 file1
```

可以使用相同的技术创建新的注册表项。实际上，由于 Windows 注册表中只有项类型属于项，因此创建注册表项更加容易。（注册表项是项属性。）例如，若要在 CurrentVersion 子项中创建名为“\_Test”的项，请键入：

```
PS> New-Item -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\_Test

Hive:Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
```

SKC	VC Name	Property
---	-----	-----
0	0 _Test	{ }

键入注册表路径时，请确保在 Windows PowerShell 驱动器名称中包含冒号 (:)，例如 HKLM: 和 HKCU:如果没有冒号，则 Windows PowerShell 将无法识别路径中的驱动器名称。

### 为什么注册表值不属于项

在使用 **Get-ChildItem** cmdlet 查找注册表项中的项时，是无法看到实际的注册表项或注册表项的值的。

例如，注册表项 **HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run** 通常包含几个表示系统启动时运行的应用程序的注册表项。

但是，在使用 **Get-ChildItem** 查找该项中的子项时，则所能看到的将是该项的 **OptionalComponents** 子项：

```
PS> Get-ChildItem HKLM:\Software\Microsoft\Windows\CurrentVersion\Run
Hive:Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
SKC VC Name Property
--- --
3 0 OptionalComponents { }
```

尽管可以很容易地将注册表项当作项，但却无法将路径指定给注册表项以确保其唯一性。路径表示法无法将名为 **Run** 的注册表子项与 **Run** 子项中的 **(Default)** 注册表项区分开来。此外，由于注册表项的名称包含反斜杆字符 (\)，如果注册表项是项，则无法使用路径表示法来将名为 **Windows\CurrentVersion\Run** 的注册表项与位于该路径中的子项区分开来。

### 重命名现有项 (Rename-Item)

若要更改文件名或文件夹的名称，请使用 **Rename-Item** cmdlet。以下命令可将 **file1.txt** 文件名更改为 **fileOne.txt**。

```
PS> Rename-Item -Path C:\temp\New.Directory\file1.txt fileOne.txt
```

**Rename-Item** cmdlet 可更改文件名或文件夹的名称，但无法移动项。以下命令失败的原因是，该命令试图将 **New.Directory** 目录中的文件移动至 **Temp** 目录。

```
PS> Rename-Item -Path C:\temp\New.Directory\fileOne.txt c:\temp\fileOne.txt
Rename-Item :无法重命名，因为指定的目标不是路径。
所在行:1 字符:12
+ Rename-Item <<<< -Path C:\temp\New.Directory\fileOne c:\temp\fileOne.txt
```

### 移动项 (Move-Item)

若要移动文件或文件夹，请使用 **Move-Item** cmdlet。

例如，以下命令可以将 **New.Directory** 目录从 **C:\temp** 目录移动至 **C:** 驱动器的根目录中。若要验证某项是否已移动，请在 **Move-Item** cmdlet 中包含 **PassThru** 参数。如果不包含 **PassThru**，则 **Move-Item** cmdlet 将不会显示任何结果。

```
PS> Move-Item -Path C:\temp\New.Directory -Destination C:\ -PassThru
```

```
Directory:Microsoft.Windows PowerShell.Core\FileSystem::C:\

Mode                LastWriteTime         Length Name
----                -
d-----          2006-05-18  12:14 PM                New.Directory
```

## 复制项 (Copy-Item)

如果您熟悉其他外壳程序中的复制操作，则可能会发现 Windows PowerShell 中的 **Copy-Item** cmdlet 的行为有些不同。在您将某项从一个位置复制到另一位置时，默认情况下，Copy-Item 不会复制其内容。

例如，如果将 **New.Directory** 目录从 C: 驱动器复制到 C:\temp 目录，则会执行该命令，但却不会复制 New.Directory 目录中的文件。

```
PS> Copy-Item -Path C:\New.Directory -Destination C:\temp
```

如果显示 **C:\temp\New.Directory** 中的内容，则将发现该目录中不包含任何文件：

```
PS> Get-ChildItem -Path C:\temp\New.Directory
PS>
```

为什么 **Copy-Item** cmdlet 不会将内容复制到新的位置？

**Copy-Item** cmdlet 已设计为通用命令，它不适用于复制文件和文件夹。另外，甚至在复制文件和文件夹时，您也只能复制容器，而不能复制容器中的项。

若要复制文件夹中的所有内容，请在命令中包含 **Copy-Item** cmdlet 的 **Recurse** 参数。如果已复制的目录没有内容，则可以添加 **Force** 参数，而该参数将允许覆盖空文件夹。

```
PS> Copy-Item -Path C:\New.Directory -Destination C:\temp -Recurse -Force
-Passthru
Directory:Microsoft.Windows PowerShell.Core\FileSystem::C:\temp

Mode                LastWriteTime         Length Name
----                -
d-----          2006-05-18   1:53 PM                New.Directory

Directory:Microsoft.Windows PowerShell.Core\FileSystem::C:\temp\New.Directory

Mode                LastWriteTime         Length Name
----                -
-a---          2006-05-18  11:44 AM                0 file1
```

## 删除项 (Remove-Item)

若要删除文件和文件夹，请使用 **Remove-Item** cmdlet。对于进行明显的不可逆更改的 Windows PowerShell cmdlet（例如 **Remove-Item**），若要执行此类命令，通常，Windows PowerShell cmdlet 会提示您进行确认。例如，若要尝试删除 **New.Directory** 文件夹，则系统将提示您确认该命令，因为该文件夹包含有文件：

```
PS> Remove-Item C:\New.Directory

确认
C:\temp\New.Directory 中的项具有子项，并且没有指定 -recurse 参数。
```



如果继续，所有子项均将随该项删除。是否确实要继续？

[Y] 是 [A] 全是 [N] 否 [L] 全否 [S] 挂起 [?] 帮助  
(默认值为“Y”)：

由于 **Yes** 为默认响应，因此若要删除文件夹及其文件，请按 **Enter** 键。若要不进行确认即可删除文件夹，请使用 **-Recurse** 参数。

```
PS> Remove-Item C:\temp\New.Directory -Recurse
```

## 执行项 (Invoke-Item)

Windows PowerShell 使用 **Invoke-Item** cmdlet 来对文件或文件夹执行默认操作。此默认操作是由注册表中默认应用程序的处理程序确定的，其效果与在 Windows 资源管理器中双击该项的效果相同。

例如，假设您运行以下命令：

```
PS> Invoke-Item C:\WINDOWS
```

将出现位于 C:\Windows 中的资源管理器窗口，其效果与双击 C:\Windows 文件夹相同。

如果在 Windows Vista 之前的系统中调用 **Boot.ini** 文件：

```
PS> Invoke-Item C:\boot.ini
```

如果 .ini 文件类型与 Notepad 相关联，则 boot.ini 文件将在 Notepad 中打开。

## 处理对象

我们已经讨论了 Windows PowerShell 如何使用对象在 cmdlet 之间传输数据，并通过使用 Get-Member 和 Format cmdlet 查看对象的特定属性来对查看有关对象详细信息的几种方法进行了说明。

对象的作用是，允许您访问许多复杂的数据，并且它是相关的。通过某些简单的技术，就可以进一步操作对象以进行更多工作。在本章中，我们将讨论可对其进行操作的某些特定对象的类型和操作方法。

## 获取 WMI 对象 (Get-WmiObject)

### 获取 WMI 对象 (Get-WmiObject)

Windows Management Instrumentation (WMI) 是 Windows 系统管理的核心技术，因为它可以按统一的方式公开各种类型的信息。由于 WMI 具有如此强大的功能，因此用于访问 WMI 对象的 Windows PowerShell cmdlet，即 **Get-WmiObject**，为实际工作中最有用的命令之一。我们将讨论如何使用 Get-WmiObject 来访问 WMI 对象，然后讨论如何使用 WMI 对象来执行特定操作。

## 列出 WMI 类

大多数 WMI 用户遇到的第一个问题是：希望了解利用 WMI 可以执行哪些操作。WMI 类可以描述可管理的资源。系统中存在几百个 WMI 类，其中某些类包含几十个属性。

**Get-WmiObject** 可以通过将 WMI 成为可发现的类来解决此问题。您可以通过键入以下命令来获取本地计算机中可用的 WMI 类列表：

```
PS> Get-WmiObject -List

__SecurityRelatedClass          __NTLMUser9X
__PARAMETERS                    __SystemSecurity
__NotifyStatus                  __ExtendedStatus
Win32_PrivilegesStatus           Win32_TSNetworkAdapterSettingError
Win32_TSRemoteControlSettingError Win32_TSEnvironmentSettingError
...
```

通过使用 **ComputerName** 参数，然后指定计算机名称或 IP 地址，可以从远程计算机中检索相同的信息：

```
PS> Get-WmiObject -List -ComputerName 192.168.1.29

__SystemClass          __NAMESPACE
__Provider              __Win32Provider
__ProviderRegistration  __ObjectProviderRegistration
...
```

根据该计算机所运行的特定操作系统和由已安装的应用程序所添加的特定 WMI 扩展的不同，远程计算机返回的类列表也各不相同。

### 请注意：

在使用 **Get-WmiObject** 连接到远程计算机时，该远程计算机必须运行 WMI，并且，在默认配置下，您使用的帐户必须是该远程计算机上本地管理员组的成员。远程系统不需要安装 Windows PowerShell。从而，您可以对未运行 Windows PowerShell 但可使用 WMI 的操作系统进行管理。

甚至可在连接到本地系统时包括 **ComputerName**。可以将本地计算机的名称、其 IP 地址（或环回地址 127.0.0.1）或 WMI 样式 ‘.’ 用作计算机名称。如果名为 Admin01（其 IP 地址为 192.168.1.90）的计算机上运行的是 Windows PowerShell，则以下命令将返回该计算机的 WMI 类列表：

```
Get-WmiObject -List
Get-WmiObject -List -ComputerName .
Get-WmiObject -List -ComputerName Admin01
Get-WmiObject -List -ComputerName 192.168.1.90
Get-WmiObject -List -ComputerName 127.0.0.1
Get-WmiObject -List -ComputerName localhost
```

默认情况下，**Get-WmiObject** 使用 **root/cimv2** 命名空间。若要指定另一个 WMI 命名空间，请使用 **Namespace** 参数并指定相应的命名空间路径：

```
PS> Get-WmiObject -List -ComputerName 192.168.1.29 -Namespace root

__SystemClass          __NAMESPACE
__Provider              __Win32Provider
```

...

### 显示 WMI 类详细信息

如果您已经知道 WMI 类的名称，则可立即使用该类来获取信息。例如，常用于检索计算机相关信息的 WMI 类之一就是 **Win32\_OperatingSystem**。

```
PS> Get-WmiObject -Class Win32_OperatingSystem -Namespace root/cimv2 -ComputerName .

SystemDirectory :C:\WINDOWS\system32
Organization    :Global Network Solutions
BuildNumber     : 2600
RegisteredUser  :Oliver W. Jones
SerialNumber    : 12345-678-9012345-67890
Version         :5.1.2600
```

尽管我们已给出所有参数，但也可采用更简洁的方式来表示命令。在连接到本地系统时，不一定非要使用 **ComputerName** 参数。介绍该参数的目的是为了对最常见的情况予以说明，并帮助您熟悉此参数。**Namespace** 默认为 root/cimv2，且可省略。最后，大多数 cmdlet 都允许省略一些常见参数名称。在使用 Get-WmiObject 时，如果没有为第一个参数指定名称，则 Windows PowerShell 会将其视为 **Class** 参数。这意味着可能已通过键入以下命令发出上一命令：

```
Get-WmiObject Win32_OperatingSystem
```

**Win32\_OperatingSystem** 类所具有的属性数量远远超过此处所显示的属性数量。可以使用 Get-Member 来查看所有属性。就像其他对象属性一样，WMI 类的属性也将自动可用：

```
PS> Get-WmiObject -Class Win32_OperatingSystem -Namespace root/cimv2 -ComputerName . | Get-Member -MemberType Property

TypeName: System.Management.ManagementObject#root\cimv2\Win32_OperatingSystem

Name      MemberType Definition
-----
__CLASS   Property   System.String __CLASS {...
...
BootDevice Property   System.String BootDevic...
BuildNumber Property   System.String BuildNumb...
...
```

### 使用 Format Cmdlet 显示非默认的属性

若要使用默认情况下不显示的 **Win32\_OperatingSystem** 类中包含的信息，则可通过使用 **Format** cmdlet 来显示这些信息。例如，如果希望显示可用的内存数据，请键入：

```
PS> Get-WmiObject -Class Win32_OperatingSystem -Namespace root/cimv2 -ComputerName . | Format-Table -Property
TotalVirtualMemorySize,TotalVisibleMemorySize,FreePhysicalMemory,FreeVirtualMemory,FreeSpaceInPagingFiles

TotalVirtualMemorySize TotalVisibleMem FreePhysicalMem FreeVirtualMemo
```

FreeSpaceInPagingFiles	FreeSpaceInPagingFiles	FreeSpaceInPagingFiles	FreeSpaceInPagingFiles	FreeSpaceInPagingFiles
2097024	785904	305808	2056724	1558232



请注意：

通配符可使用 **Format-Table** 中的属性名称，因此，最后的管道元素可精简为 **Format-Table -Property TotalV\*,Free\***

如果通过键入以下命令将内存数据的格式设置为列表，则将大大提高该内存数据的可读性：

```
PS> Get-WmiObject -Class Win32_OperatingSystem -Namespace root/cimv2 -ComputerName . | Format-List
TotalVirtualMemorySize,TotalVisibleMemorySize,FreePhysicalMemory,FreeVirtualMemory,FreeSpaceInPagingFiles

TotalVirtualMemorySize : 2097024
TotalVisibleMemorySize : 785904
FreePhysicalMemory      : 301876
FreeVirtualMemory       : 2056724
FreeSpaceInPagingFiles  : 1556644
```

## 创建 .NET 和 COM 对象 (New-Object)

某些软件组件具有 .NET Framework 和 COM 接口，因此可执行许多系统管理任务。Windows PowerShell 允许您使用这些组件，因此，您可以使用 cmdlet 执行这些任务，而不会受到任何限制。在最初版本的 Windows PowerShell 中，多数 cmdlet 不适用于远程计算机。我们将在直接使用 Windows PowerShell 中的 .NET **System.Diagnostics.EventLog** 类来管理事件日志时，演示如何消除此限制。

### 使用 New-Object 访问事件日志

.NET Framework 类库包含可用于管理事件日志的名为 **System.Diagnostics.EventLog** 的类。可以使用具有 **TypeName** 参数的 **New-Object** cmdlet 来创建新的 .NET 类实例。例如，以下命令可创建事件日志引用：

```
PS> New-Object -TypeName System.Diagnostics.EventLog

Max(K) Retain OverflowAction Entries Name
-----
```

尽管该命令已创建 EventLog 类实例，但该实例并不包含任何数据。这是由于未指定特定的事件日志。如何可获取实际事件日志？

### 将构造函数与 New-Object 结合使用

若要引用特定的事件日志，则需要指定该日志的名称。**New-Object** 具有 **ArgumentList** 参数。对象的特殊启动方法可使用作为值传递到此参数的参数。由于该方法用于构造对象，因此称为

构造函数。例如，若要获取对 Application 日志的引用，请将字符串 ‘Application’指定为参数：

```
PS> New-Object -TypeName System.Diagnostics.EventLog -ArgumentList Application
```

Max(K)	Retain OverflowAction	Entries	Name
16,384	7 OverwriteOlder	2,160	Application

 请注意：

由于大多数 .NET 核心类包含在系统命名空间中，因此如果 Windows PowerShell 找不到与指定类型名称匹配的项，则将自动尝试查找在系统命名空间中指定的类。这意味着，您可以指定 `Diagnostics.EventLog` 而不是 `System.Diagnostics.EventLog`。

## 在变量中存储对象

您可能需要存储对对象的引用，以便在 Windows PowerShell 会话期间使用它。尽管通过 Windows PowerShell 可对管道执行许多操作，但是，为了减少需要使用的变量数，有时将对象的引用存储在变量中可更方便地对这些对象进行操作。

Windows PowerShell 允许您创建其本质为对象的变量。任何有效的 Windows PowerShell 命令的输出均可存储在变量中。变量名称始终以 \$ 开头。若要在名为 \$AppLog 的变量中存储对应应用程序日志的引用，请键入变量名称、后跟等号，然后键入用于创建应用程序日志对象的命令：

```
PS> $AppLog = New-Object -TypeName System.Diagnostics.EventLog -ArgumentList Application
```

随后，如果键入 \$AppLog，则可看到 \$AppLog 变量中包含该应用程序日志：

```
PS> $AppLog
```

Max(K)	Retain OverflowAction	Entries	Name
16,384	7 OverwriteOlder	2,160	Application

## 使用 New-Object 访问远程事件日志

前一节中使用的命令主要适用于本地计算机，但 `Get-EventLog` cmdlet 也可实现相同的功能。若要访问远程计算机上的应用程序日志，则必须将日志名称以及计算机名称（或 IP 地址）作为参数进行提供。

```
PS> $RemoteAppLog = New-Object -TypeName System.Diagnostics.EventLog Application,192.168.1.81
PS> $RemoteAppLog
```

Max(K)	Retain OverflowAction	Entries	Name
512	7 OverwriteOlder	262	Application

既然我们已将事件日志的引用存储在 \$RemoteAppLog 变量中，那么还可对其执行哪些任务？

## 使用对象方法清除事件日志

对象中经常具有可调用以执行任务的方法。可以使用 `Get-Member` 来显示与对象关联的方

法。以下命令和选定的输出显示了 **EventLog** 类的其中一些方法：

```
PS> $RemoteAppLog | Get-Member -MemberType Method

TypeName: System.Diagnostics.EventLog

Name      MemberType Definition
-----
...
Clear      Method      System.Void Clear()
Close      Method      System.Void Close()
...
GetType    Method      System.Type GetType()
...
ModifyOverflowPolicy Method      System.Void ModifyOverflowPolicy(Overfl...
RegisterDisplayName Method      System.Void RegisterDisplayName(String ...
...
ToString   Method      System.String ToString()
WriteEntry  Method      System.Void WriteEntry(String message),...
WriteEvent  Method      System.Void WriteEvent(EventInstance in...
```

使用 **Clear()** 方法可清除事件日志。在调用方法时，方法名称必须后跟圆括号，即使该方法不需要参数也是这样。这样，Windows PowerShell 可以将方法与可能同名的属性区分开来。键入以下命令可调用 **Clear** 方法：

```
PS> $RemoteAppLog.Clear()
```

键入以下命令可显示日志。此时，您将看到事件日志已清除，并且该日志中有 0 个而不是 262 个条目：

```
PS> $RemoteAppLog

Max(K) Retain OverflowAction      Entries Name
-----
512      7 OverwriteOlder           0 Application
```

## 使用 New-Object 创建 COM 对象

您可以使用 **New-Object** 来处理组件对象模型 (COM) 组件。组件的范围非常广泛，从 Windows Script Host (WSH) 中包含的各种库到 ActiveX 应用程序，例如大多数系统中所安装的 Internet Explorer。

**New-Object** 可使用 .NET Framework 运行库可调用包装来创建 COM 对象，因此它具有的限制与在调用 COM 对象时 .NET 具有的限制相同。若要创建 COM 对象，则需要利用要使用的 COM 类的编程标识符或 ProgId 来指定 **ComObject** 参数。在本教程中，将不对有关 COM 使用限制和确定系统中哪些 ProgId 可用进行完整的讨论，但环境中大多数已知对象（例如 WSH）可以在 Windows PowerShell 中使用。

可以通过指定这些 ProgId 来创建 WSH 对象：**WScript.Shell**、**WScript.Network**、**Scripting.Dictionary** 和 **Scripting.FileSystemObject**。使用以下命令可创建这些对象：

```
New-Object -ComObject WScript.Shell
New-Object -ComObject WScript.Network
New-Object -ComObject Scripting.Dictionary
New-Object -ComObject Scripting.FileSystemObject
```

尽管这些类的大多数功能可在 Windows PowerShell 中以其他方式获得，但使用 WSH 类来执行某些任务（例如创建快捷方式）则更简单易行。

## 使用 WScript.Shell 创建桌面快捷方式

使用 COM 对象可快速执行的一个任务是创建快捷方式。假设您要在链接到 Windows PowerShell 主文件夹的桌面上创建快捷方式。首先，需要创建对 **WScript.Shell** 的引用，后者将存储在名为 **\$WshShell** 的变量中：

```
$WshShell = New-Object -ComObject WScript.Shell
```

**Get-Member** 可处理 COM 对象，因此可键入以下命令来浏览对象成员：

```
PS> $WshShell | Get-Member

TypeName: System.__ComObject#{41904400-be18-11d3-a28b-00104bd35090}

Name                MemberType          Definition
----                -
AppActivate          Method               bool AppActivate (Variant, Va...
CreateShortcut        Method               IDispatch CreateShortcut (str...
...
```

 请注意：

**Get-Member** 具有可选参数 **InputObject**，因此您可以使用该参数而无需通过管道向 **Get-Member** 提供输入。如果改用 **Get-Member -InputObject \$WshShell** 命令，则将得到与上述相同的输出。如果使用 **InputObject**，则它会将其参数视为单独项。这表示，如果一个变量中具有几个对象，则 **Get-Member** 将这些对象视为对象数组。例如：

```
PS> $a = 1,2,"three"
```


```
PS> Get-Member -InputObject $a
```

```
TypeName: System.Object[]
```

```
Name                MemberType          Definition
----                -
Count               AliasProperty       Count = Length
...
```

**WScript.Shell CreateShortcut** 方法可接受单个参数，该参数是指向要创建的快捷方式文件的路径。我们可以键入指向桌面的完整路径，但还有另一种更简单的方法。桌面通常是由当前用户的主文件夹中名为“桌面”的文件夹表示。Windows PowerShell 具有包含指向此文件夹的路径的变量 **\$Home**。可以使用此变量来指定指向主文件夹的路径，然后，键入以下命令来添加“桌面”文件夹的名称以及要创建的快捷方式的名称：

```
$lnk = $WshShell.CreateShortcut("$Home\Desktop\PSHome.lnk")
```

 请注意：

在双引号中使用类似于变量名称的数据时，Windows PowerShell 将尝试替换匹配的值。如果使用单引号，则 Windows PowerShell 将不进行变量值替换。例如，尝试键入以下命令：

```
PS> "$Home\Desktop\PSHome.lnk"

C:\Documents and Settings\aka\Desktop\PSHome.lnk

PS> '$Home\Desktop\PSHome.lnk'

$Home\Desktop\PSHome.lnk
```

现在，我们有了一个名为 **\$lnk** 的变量，变量中包含新的快捷方式引用。若要查看其成员，则可通过管道将其传递给 **Get-Member**。以下输出显示了需要用于完成快捷方式创建的成员：

```
PS> $lnk | Get-Member
```

```
TypeName: System.__ComObject#{f935dc23-1cf0-11d0-adb9-00c04fd58a0b}
```

Name	MemberType	Definition
...		
Save	Method	void Save ()
...		
TargetPath	Property	string TargetPath () {get} {set}
...		

需要指定 **TargetPath**（Windows PowerShell 的应用程序文件夹），然后通过调用 **Save** 方法保存该快捷方式 **\$lnk**。Windows PowerShell 应用程序文件夹路径存储在变量 **\$PSHome** 中，因此可键入以下命令来实现此操作：

```
$lnk.TargetPath = $PSHome
$lnk.Save()
```

## 使用 Windows PowerShell 中的 Internet Explorer

使用 COM 可以自动启动许多应用程序，其中包括 Microsoft Office 系列应用程序和 Internet Explorer。Internet Explorer 演示了在处理基于 COM 的应用程序时涉及的一些典型技巧和问题。通过指定 Internet Explorer ProgId (**InternetExplorer.Application**) 可以创建 Internet Explorer 实例：

```
$ie = New-Object -ComObject InternetExplorer.Application
```

此命令可启动 Internet Explorer，但该应用程序将不可见。如果键入 **Get-Process**，则可看到名为 **ieexplore** 的进程正在运行。事实上，即使退出 Windows PowerShell，该进程仍将继续运行。必须重新启动计算机或使用类似于任务管理器之类的工具才可结束 **ieexplore** 进程。

 请注意：

作为单独进程启动的 COM 对象通常称为 **ActiveX** 可执行文件，在它们启动时可能会显示用户界面窗口，但也可能不显示。如果创建了窗口，但该窗口不可见，例如 **Internet Explorer**，则焦点通常会移至 Windows 桌面，而您必须使该窗口可见才可与其进行交互。

如果键入 **\$ie | Get-Member**，则可查看 Internet Explorer 的属性和方法。若要查看 Internet Explorer 窗口，请通过键入以下命令将 **Visible** 属性设置为 **\$true**：

```
$ie.Visible = $true
```



随后，可以使用导航方法导航至特定的 web 地址：

```
$ie.Navigate("http://www.microsoft.com/technet/scriptcenter/default.aspx")
```

使用 Internet Explorer 对象模型的其他成员，可以从网页中检索文本内容。以下命令将显示当前网页正文中的 HTML 文本：

```
$ie.Document.Body.InnerText
```

若要在 PowerShell 中关闭 Internet Explorer，请调用其 Quit() 方法：

```
$ie.Quit()
```

这将强制关闭 Internet Explorer。\$ie 变量将不再包含有效引用，即使该引用仍为 COM 对象也是如此。若要尝试使用该引用，则将收到自动化错误：

```
PS> $ie | Get-Member
Get-Member :检索属性“Application”的字符串表示形式时发生异常：
“被调用的对象已与其客户端断开连接。（HRESULT 出现异常：0x80010108
(RPC_E_DISCONNECTED)）”
所在行:1 字符:16
+ $ie | Get-Member <<<<
```

您可以使用诸如 \$ie = \$null 之类的命令来删除剩余的引用，也可通过键入以下命令完全删除该变量：

```
Remove-Variable ie
```

 请注意：

在删除对某一项的引用时，对于如何确定是退出还是继续运行 ActiveX 可执行文件没有通用标准。退出或继续运行应用程序取决于以下几种情况：该应用程序是否可见、已编辑的文档是否在其中运行，以及 Windows PowerShell 是否仍在运行。鉴于此原因，您应该对要在 Windows PowerShell 中使用的每个 ActiveX 可执行文件测试其终止行为。

## 获取有关 .NET-Wrapped COM 对象的警告

在某些情况下，COM 对象可能具有关联的 .NET 运行库可调用包装或 RCW，而这将由 **New-Object** 所使用。由于 RCW 的行为可能与标准 COM 对象的行为有所不同，因此 **New-Object** 的 **Strict** 参数将通知您有关 RCW 访问的信息。如果指定 **Strict** 参数，然后创建使用 RCW 的 COM 对象，则将收到一条警告消息：

```
PS> $xl = New-Object -ComObject Excel.Application -Strict
New-Object :被写入管道的对象是该组件的主互操作程序集中类型
为“Microsoft.Office.Interop.Excel.ApplicationClass”的实例。
如果此类型公开的成员不是 IDispatch 成员，则在未安装该主互操作程序集的情况下，为使用此对象而编
写的脚本可能无法运行。
所在行:1 字符:17
+ $xl = New-Object <<<< -ComObject Excel.Application -Strict
```

尽管已创建对象，但仍将警告您它不是标准的 COM 对象。

# 使用静态类和方法

不是所有的 .NET Framework 类都可使用 **New-Object** 进行创建。例如，如果尝试使用 **New-Object** 创建 **System.Environment** 或 **System.Math** 对象，则将收到以下错误消息：

```
PS> New-Object System.Environment
New-Object :找不到构造函数。无法找到适合类型 System.Environment 的构造函数。
所在行:1 字符:11
+ New-Object <<<< System.Environment
PS> New-Object System.Math
New-Object :找不到构造函数。无法找到适合类型 System.Math 的构造函数。
所在行:1 字符:11
+ New-Object <<<< System.Math
```

之所以出现这些错误是因为无法从这些类中创建新的对象。这些类是不能更改状态的方法和属性的引用库。您无法创建这些类，只能使用他们。由于不能创建、销毁或更改这些类和方法，因此将这些类和方法称为静态类。为了清楚了解此部分内容，我们将提供使用静态类的示例。

## 使用 System.Environment 获取环境数据


通常，在 Windows PowerShell 中处理对象的第一步为，使用 **Get-Member** 显示该对象包含的成员。使用静态类，过程可能稍有差异，这是因为实际类不是对象。

### 引用 System.Environment 静态类

通过使用方括号括住类名称，可以引用静态类。例如，可以在方括号中键入名称来引用 **System.Environment**。从而，就可显示一些泛型类型的信息：

```
PS> [System.Environment]

IsPublic IsSerial Name                                     BaseType
-----
True     False     Environment                                     System.Object
```

 请注意：

如上所述，在使用 **New-Object** 时，Windows PowerShell 将自动加上 ‘**System.**’ 以便于键入名称。在使用方括号类型名称时亦即如此，因此，可以将 **[System.Environment]** 指定为 **[Environment]**。

在 Windows PowerShell 中工作时，**System.Environment** 类可包含有关当前进程的工作环境（powershell.exe）的常规信息。

若要通过键入 **[System.Environment] | Get-Member** 来查看此类的详细信息，则对象类型将报告为 **System.RuntimeType**，而不是 **System.Environment**：

```
PS> [System.Environment] | Get-Member

TypeName: System.RuntimeType
```

若要使用 **Get-Member** 查看静态成员，请指定 **Static** 参数：

```
PS> [System.Environment] | Get-Member -Static

TypeName: System.Environment
```

Name	MemberType	Definition
-----	-----	-----
Equals	Method	static System.Boolean Equals(Object ob...
Exit	Method	static System.Void Exit(Int32 exitCode)
...		
CommandLine	Property	static System.String CommandLine {get;}
CurrentDirectory	Property	static System.String CurrentDirectory ...
ExitCode	Property	static System.Int32 ExitCode {get;set;}
HasShutdownStarted	Property	static System.Boolean HasShutdownStart...
MachineName	Property	static System.String MachineName {get;}
NewLine	Property	static System.String NewLine {get;}
OSVersion	Property	static System.OperatingSystem OSVersio...
ProcessorCount	Property	static System.Int32 ProcessorCount {get;}
StackTrace	Property	static System.String StackTrace {get;}
SystemDirectory	Property	static System.String SystemDirectory {...
TickCount	Property	static System.Int32 TickCount {get;}
UserDomainName	Property	static System.String UserDomainName {g...
UserInteractive	Property	static System.Boolean UserInteractive ...
UserName	Property	static System.String UserName {get;}
Version	Property	static System.Version Version {get;}
WorkingSet	Property	static System.Int64 WorkingSet {get;}
TickCount		ExitCode

现在，可以从 `System.Environment` 中选择要查看的属性。

### 显示 `System.Environment` 的静态属性

`System.Environment` 的属性也是静态的，因此必须采用与标准属性的指定方式不同的方式进行指定。从而，使用 `::` 来指示 `Windows PowerShell` 要进行处理的静态方法或属性。若要查看用于启动 `Windows PowerShell` 的命令，则可通过键入以下命令来检查 **CommandLine** 属性：

```
PS> [System.Environment]::CommandLine
"C:\Program Files\Windows PowerShell\v1.0\powershell.exe"
```

若要检查操作系统版本，则可通过键入以下命令显示 **OSVersion** 属性：

```
PS> [System.Environment]::OSVersion

Platform ServicePack      Version      VersionString
-----
Win32NT Service Pack 2    5.1.2600.131072  Microsoft Window...
```

通过显示 **HasShutdownStarted** 属性，可以检查计算机是否正在关闭：

```
PS> [System.Environment]::HasShutdownStarted
False
```

### 使用 `System.Math` 进行数学运算

对于进行某些数学运算，`System.Math` 静态类非常有用。**System.Math** 的重要成员绝大部分为方法，使用 **Get-Member** 可显示这些方法。

 请注意：

`System.Math` 中有几种方法同名，但通过它们使用的参数类型可进行区分。

键入以下命令可列出 **System.Math** 类中的方法:

```
PS> [System.Math] | Get-Member -Static -MemberType Methods

TypeName: System.Math

Name                                     MemberType Definition
----
Abs                                     Method      static System.Single Abs(Single value), static Sy...
Acos                                   Method      static System.Double Acos(Double d)
Asin                                   Method      static System.Double Asin(Double d)
Atan                                   Method      static System.Double Atan(Double d)
Atan2                                  Method      static System.Double Atan2(Double y, Double x)
BigMul                                 Method      static System.Int64 BigMul(Int32 a, Int32 b)
Ceiling                                Method      static System.Double Ceiling(Double a), static Sy...
Cos                                    Method      static System.Double Cos(Double d)
Cosh                                   Method      static System.Double Cosh(Double value)
DivRem                                 Method      static System.Int32 DivRem(Int32 a, Int32 b, Int3...
Equals                                 Method      static System.Boolean Equals(Object objA, Object ...
Exp                                    Method      static System.Double Exp(Double d)
Floor                                  Method      static System.Double Floor(Double d), static Syst...
IEEERemainder                         Method      static System.Double IEEERemainder(Double x, Doub...
Log                                    Method      static System.Double Log(Double d), static System...
Log10                                  Method      static System.Double Log10(Double d)
Max                                    Method      static System.SByte Max(SByte val1, SByte val2), ...
Min                                    Method      static System.SByte Min(SByte val1, SByte val2), ...
Pow                                    Method      static System.Double Pow(Double x, Double y)
ReferenceEquals                       Method      static System.Boolean ReferenceEquals(Object objA...
Round                                  Method      static System.Double Round(Double a), static Syst...
Sign                                   Method      static System.Int32 Sign(SByte value), static Sys...
Sin                                    Method      static System.Double Sin(Double a)
Sinh                                   Method      static System.Double Sinh(Double value)
Sqrt                                   Method      static System.Double Sqrt(Double d)
Tan                                    Method      static System.Double Tan(Double a)
Tanh                                   Method      static System.Double Tanh(Double value)
Truncate                              Method      static System.Decimal Truncate(Decimal d), static...
```

这将显示几种数学方法。此处的命令列表演示了一些常用方法的工作原理:

```
PS> [System.Math]::Sqrt(9)
3
PS> [System.Math]::Pow(2,3)
8
PS> [System.Math]::Floor(3.3)
3
PS> [System.Math]::Floor(-3.3)
-4
PS> [System.Math]::Ceiling(3.3)
4
PS> [System.Math]::Ceiling(-3.3)
-3
PS> [System.Math]::Max(2,7)
7
PS> [System.Math]::Min(2,7)
2
PS> [System.Math]::Truncate(9.3)
9
PS> [System.Math]::Truncate(-9.3)
```


## 从管道中删除对象 (Where-Object)

在 Windows PowerShell 中，与所需的对象数量相比，通常生成的对象数量以及要传递给管道的对象数量要多得多。可以使用 **Format cmdlet** 来指定要显示的特定对象的属性，但这并不能帮您解决从显示中删除整个对象的问题。您可能希望在管道结束之前筛选对象，因此只能在最初生成的对象子集上执行操作。

利用 Windows PowerShell 中的 **Where-Object cmdlet**，可以测试管道中的所有对象，并将符合特定测试条件的对象通过管道进行传递。没有通过测试的对象将从管道中删除。可以将测试条件设置为 **Where-ObjectFilterScript** 参数的值。

### 使用 Where-Object 执行简单测试

**FilterScript** 的值为计算结果为 **True** 或 **False** 的脚本块（由大括号 `{ }` 括住的一个或多个 Windows PowerShell 命令）。这些脚本块非常简单，但创建这些脚本块则需要了解 Windows PowerShell 的另一概念，即，比较运算符。比较运算符可比较该运算符两侧的项。比较运算符以 “-” 字符开头，后跟名称。基本的比较运算符几乎对所有类型的对象适用。更高级的比较运算符只适用于文本或数组。

 请注意：

默认情况下，在处理文本时，Windows PowerShell 比较运算符不区分大小写。

出于分析方面的考虑，诸如 `<`、`>` 和 `=` 之类的符号不能用作比较运算符。因此，比较运算符改由字母组成。基本的比较运算符如下表所示：

比较运算符	含义	示例（返回 <b>True</b> ）
<code>-eq</code>	等于	<code>1 -eq 1</code>
<code>-ne</code>	不等于	<code>1 -ne 2</code>
<code>-lt</code>	小于	<code>1 -lt 2</code>
<code>-le</code>	小于或等于	<code>1 -le 2</code>
<code>-gt</code>	大于	<code>2 -gt 1</code>
<code>-ge</code>	大于或等于	<code>2 -ge 1</code>
<code>-like</code>	类似（用于文本的通配符比较）	<code>"file.doc" -like "f*.do?"</code>
<code>-notlike</code>	不类似（用于文本的通配符比较）	<code>"file.doc" -notlike "p*.doc"</code>
<code>-contains</code>	包含	<code>1,2,3 -contains 1</code>
<code>-notcontains</code>	不包含	<code>1,2,3 -notcontains 4</code>

**Where-Object** 脚本块使用特殊的变量 “`$_`” 来引用管道中的当前对象。此处的示例将演示该变量的工作原理。如果存在一个数字列表，而您只需返回小于 3 的数字，则可通过键入以下命令

令来使用 Where-Object 筛选数字：

```
PS> 1,2,3,4 | Where-Object -FilterScript {$_ -lt 3}
1
2
```

## 根据对象属性进行筛选

由于 \$\_ 引用当前的管道对象，因此可访问其属性以进行测试。

作为示例，我们可以查看 WMI 中的 Win32\_SystemDriver 类。特定系统中可能存在几百个系统驱动程序，而您可能只对某一组特定的系统驱动程序感兴趣，例如。当前正在运行的那些系统驱动程序。如果使用 Get-Member 来查看 Win32\_SystemDriver 成员（**Get-WmiObject -Class Win32\_SystemDriver | Get-Member -MemberType Property**），则将看到的相关属性是“State”，并且该驱动程序运行时，它具有值“Running”。键入以下命令可以只选择正在运行的系统驱动程序以进行筛选操作：

```
Get-WmiObject -Class Win32_SystemDriver | Where-Object -FilterScript {$_.State -eq "Running"}
```

这仍会生成一个很长的列表。您可能还希望通过测试 StartMode 值来进行筛选，以便只选择设置为自动启动的驱动程序：

```
PS> Get-WmiObject -Class Win32_SystemDriver | Where-Object -FilterScript {$_.State -eq "Running"} | Where-Object -FilterScript {$_.StartMode -eq "Auto"}
```

```
DisplayName :RAS Asynchronous Media Driver
Name        :AsyncMac
State       :Running
Status      :OK
Started     :True
```

```
DisplayName :Audio Stub Driver
Name        :audstub
State       :Running
Status      :OK
Started     :True
```

由于我们已了解哪些驱动程序正在运行，因此这将产生许多我们不再需要的信息。实际上，此时我们可能需要的信息仅仅是名称和显示名称。以下命令只包括这两个属性，从而得到更简单的输出：

```
PS> Get-WmiObject -Class Win32_SystemDriver | Where-Object -FilterScript {$_.State -eq "Running"} | Where-Object -FilterScript {$_.StartMode -eq "Manual"} | Format-Table -Property Name,DisplayName
```

Name	DisplayName
AsyncMac	RAS Asynchronous Media Driver
Fdc	Floppy Disk Controller Driver
Flpydisk	Floppy Disk Driver
Gpc	Generic Packet Classifier
IpNat	IP Network Address Translator
mouhid	Mouse HID Driver
MRxDAV	WebDav Client Redirector
mssmbios	Microsoft System Management BIOS Driver

上述命令中存在两个 **Where-Object** 元素，而他们可通过使用 **-and** 逻辑运算符，以单个 **Where-Object** 元素的形式表示出来，如下所示：

```
Get-WmiObject -Class Win32_SystemDriver | Where-Object -FilterScript { ($_.State
-eq "Running") -and ($_.StartMode -eq "Manual") } | Format-Table -Property
Name,DisplayName
```

标准的逻辑运算符如下表所示：

逻辑运算符	含义	示例（返回 <b>True</b> ）
-and	逻辑与；两边都为 <b>True</b> 时值为 <b>True</b>	(1 -eq 1) -and (2 -eq 2)
-or	逻辑或；其中一边为 <b>True</b> 时值为 <b>True</b>	(1 -eq 1) -or (1 -eq 2)
-not	逻辑非；对 <b>True</b> 和 <b>False</b> 取反	-not (1 -eq 2)
!	逻辑非；对 <b>True</b> 和 <b>False</b> 取反	!(1 -eq 2)

## 对多个对象重复同一任务 (ForEach-Object)

**ForEach-Object** cmdlet 可对当前管道对象使用脚本块和 **\$\_** 描述符，以允许您对该管道中每个对象运行命令。这可用于执行某些复杂的任务。

对于以下情况这将非常有用：对数据进行操作以提高数据的实用性。例如，WMI 中的 **Win32\_LogicalDisk** 类可用于返回每个本地磁盘的可用空间信息。数据是以字节形式返回的，因而可读性非常差：

```
PS> Get-WmiObject -Class Win32_LogicalDisk

DeviceID      :C:
DriveType     : 3
ProviderName  :
FreeSpace     :50665070592
Size          :203912880128
VolumeName    :Local Disk
```

将每个值两次与 **1024** 相除，从而将 **FreeSpace** 值转换为兆字节。首次相除之后，数据是以千字节形式表示的，第二次相除之后，数据就以兆字节形式表示了。通过键入以下命令可在 **ForEach-Object** 脚本块中实现此转换：

```
Get-WmiObject -Class Win32_LogicalDisk | ForEach-Object -Process
{ ($_.FreeSpace)/1024.0/1024.0}
48318.01171875
```

但是，现在输出的数据没有任何关联的标签。由于此类 **WMI** 属性是只读的，因此无法直接转换 **FreeSpace**。如果键入以下命令：

```
Get-WmiObject -Class Win32_LogicalDisk | ForEach-Object -Process {$_FreeSpace =
($_FreeSpace)/1024.0/1024.0}
```

则将收到错误消息：

```
“FreeSpace”为只读属性。  
所在行:1 字符:70  
+ Get-WmiObject -Class Win32_LogicalDisk | ForEach-Object -Process {$_.FreeSpace = ($_.FreeSpace)/1024.0/1024.0}
```

您可以使用高级技巧重新组织数据，但更简便的方法是使用 **Select-Object** 创建新对象。

## 选择对象的各个部分 (Select-Object)

可以使用 **Select-Object** cmdlet 来创建新的、自定义的 Windows PowerShell 对象，后者包含的属性是从用于创建他们的对象中选择的。键入以下命令可创建新对象，该对象仅包含 Win32\_LogicalDisk WMI 类的 Name 和 FreeSpace 属性：

```
PS> Get-WmiObject -Class Win32_LogicalDisk | Select-Object -Property  
Name,FreeSpace  
  
Name                               FreeSpace  
----                               -  
C:                                50664845312
```

在发出该命令后，您将无法查看数据的类型，但如果在执行 **Select-Object** 命令后通过管道将结果传递给 **Get-Member**，则可发现，对象的新类型 **PSCustomObject** 已存在：

```
PS> Get-WmiObject -Class Win32_LogicalDisk | Select-Object -Property  
Name,FreeSpace | Get-Member  
  
TypeName: System.Management.Automation.PSCustomObject  
  
Name      MemberType Definition  
----      -  
Equals     Method      System.Boolean Equals(Object obj)  
GetHashCode Method      System.Int32 GetHashCode()  
GetType     Method      System.Type GetType()  
ToString   Method      System.String ToString()  
FreeSpace  NoteProperty FreeSpace=...  
Name       NoteProperty System.String Name=C:
```

**Select-Object** 有很多用途。其中之一就是复制可随后进行修改的数据。现在，我们可以解决上一节中遇到的问题了。我们可以更新最新创建的对象中的 **FreeSpace** 的值，而且输出将包括描述性标签：

```
Get-WmiObject -Class Win32_LogicalDisk | Select-Object -Property Name,FreeSpace |  
ForEach-Object -Process {$_.FreeSpace = ($_.FreeSpace)/1024.0/1024.0; $_}  
Name                               FreeSpace  
----                               -  
C:48317.7265625
```



## 对对象进行排序

通过使用 **Sort-Object** cmdlet 可以组织显示的数据，从而可更轻易地对这些数据进行扫描。**Sort-Object** 可获取排序所依据的一个或多个属性的名称，并返回按这些属性值排序的数据。以 Win32\_SystemDriver 实例列表的问题为例。若要按 **State** 排序，然后按 **Name** 排序，则可通过键入以下命令来实现：

```
Get-WmiObject -Class Win32_SystemDriver | Sort-Object -Property State,Name |
Format-Table -Property Name,State,Started,DisplayName -AutoSize -Wrap
```

尽管这会生成很长的数据显示，但还是可看出具有相同状态的项组合在一起：

Name	State	Started	DisplayName
ACPI	Running	True	Microsoft ACPI Driver
AFD	Running	True	AFD
AmdK7	Running	True	AMD K7 Processor Driver
AsyncMac	Running	True	RAS Asynchronous Media Driver
...			
Abiosdsk	Stopped	False	Abiosdsk
ACPIEC	Stopped	False	ACPIEC
aec	Stopped	False	Microsoft Kernel Acoustic Echo Canceller
...			

还可通过指定 **Descending** 参数以倒序顺序排序对象。这将颠倒排序的顺序，因此将按倒序的字母顺序排序名称，按降序的顺序排序数字。

```
PS> Get-WmiObject -Class Win32_SystemDriver | Sort-Object -Property State,Name
-Descending | Format-Table -Property Name,State,Started,DisplayName -AutoSize
-Wrap
```

Name	State	Started	DisplayName
WS2IFSL	Stopped	False	Windows Socket 2.0 Non-IFS Service Provider Support Environment
wceusbsh	Stopped	False	Windows CE USB Serial Host Driver...
...			
wdmaud	Running	True	Microsoft WINMM WDM Audio Compatibility Driver
Wanarp	Running	True	Remote Access IP ARP Driver
...			

## 使用变量存储对象

Windows PowerShell 可以使用对象。利用 Windows PowerShell，您可创建变量（本质上命名为对象）以保留输出以备后用。如果您已习惯于在其他外壳程序中处理变量，请谨记，Windows PowerShell 变量是对象，而非文本。

变量始终可通过首字符 **\$** 指定，并且在变量名称中可以包含所有的字母数字字符或下划线。

## 创建变量

通过键入有效的变量名称可以创建变量：

```
PS> $loc
PS>
```

由于 **\$loc** 没有值，因此将不会返回任何结果。您可以创建变量，并可同时向其赋值。Windows PowerShell 只能创建目前尚不存在的变量，否则，它会将指定的值赋予已存在的变量。若要在变量 **\$loc** 中存储当前位置，请键入：

```
$loc = Get-Location
```

由于输出已发送到 **\$loc**，因此在键入此命令后将不会显示任何输出。在 Windows PowerShell 中，显示的输出实际上是附加功能，因为未定向的数据始终会发送到屏幕上。键入 **\$loc** 将显示当前位置：

```
PS> $loc

Path
----
C:\temp
```

您可以使用 **Get-Member** 来显示有关变量内容的信息。通过管道将 **\$loc** 传递给 **Get-Member** 表示，**\$loc** 为 **PathInfo** 对象，这类似于 **Get-Location** 的输出：

```
PS> $loc | Get-Member -MemberType Property

TypeName: System.Management.Automation.PathInfo

Name      MemberType Definition
-----
Drive      Property   System.Management.Automation.PSDriveInfo Drive {get;}
Path       Property   System.String Path {get;}
Provider   Property   System.Management.Automation.ProviderInfo Provider {...
ProviderPath Property   System.String ProviderPath {get;}
```

## 对变量进行操作

Windows PowerShell 提供了几条用于对变量进行操作的命令。通过键入以下命令可以查看极具可读性的完整列表：

```
Get-Command -Noun Variable | Format-Table -Property Name,Definition -AutoSize
-Wrap
```

除了当前 Windows PowerShell 会话中创建的变量之外，还存在几个系统定义的变量。您可以使用 **Remove-Variable** cmdlet 来清除所有不受 Windows PowerShell 控制的变量。键入以下命令可清除所有变量：

```
Remove-Variable -Name * -Force -ErrorAction SilentlyContinue
```

这将生成确认提示，如下所示：

```
确认
是否确实要执行此操作?
对目标 "Name: Error" 执行操作 "Remove Variable"。[Y] 是 [A] 全是 [N] 否 [L]
```

```
全否 [S] 挂起 [?] 帮助(默认值为“Y”):A
```

随后，如果运行 **Get-Variable** cmdlet，则可查看剩余的 Windows PowerShell 变量。由于还存在 Windows PowerShell 变量驱动器，因此也可通过键入以下命令来显示所有的 Windows PowerShell 变量：

```
Get-ChildItem variable:
```

## 使用 Cmd.exe 变量

尽管 Windows PowerShell 不是 Cmd.exe，但它也运行于命令外壳程序环境中，并且可以在 Windows 的任意环境中使用相同的可用变量。这些变量是通过名为 **env:** 的驱动器公开的。键入以下命令可查看这些变量：

```
Get-ChildItem env:
```

尽管未设计标准变量 cmdlet 来处理 **env: 变量**，但仍可通过指定 使用这些变量。例如，若要查看操作系统根目录，则可键入以下命令，使用 Windows PowerShell 中的命令外壳程序 **%SystemRoot%** 变量：

```
PS> $env:SystemRoot  
C:\WINDOWS
```

也可创建和修改 Windows PowerShell 中的环境变量。从 Windows PowerShell 访问的环境变量符合 Windows 之外的环境变量标准规则。

## 使用 Windows PowerShell 执行管理任务

Windows PowerShell 的基本目标是使您能够以交互方式或通过脚本更好、更容易地对系统进行管理控制。本章综述了在用 Windows PowerShell 管理 Windows 系统时出现的很多特定问题的解决方案。尽管我们尚未在“Windows PowerShell 入门”中介绍脚本或函数，但随后可以在脚本中或作为函数使用这些解决方案。在提供的示例中，函数将作为解决问题的解决方案的一部分。

在整个解决方案说明中，您将看到使用特定 cmdlet 的解决方案、常规 Get-WmiObject cmdlet、甚至还有作为 Windows 和 .NET 基础结构组成部分的外部工具等诸多方案的混合体。使用外部工具是 Windows PowerShell 的长期设计意图的一部分。甚至随着系统不断发展，用户将继续遇到可用工具集无法完成所有任务的情况。Windows PowerShell 不鼓励仅仅依赖 cmdlet 实现，而是尝试支持将来自所有可能的替代方案的解决方案集成在一起。

## 管理本地进程

核心进程 cmdlet 只有两个：**Get-Process** 和 **Stop-Process**。由于有可能使用参数或对象 cmdlet 来检查和筛选进程，因此可以只使用这两个 cmdlet 来执行一些复杂的任务。

## 列出进程 (Get-Process)

通过无参数运行 **Get-Process**，可以获得正在本地系统中运行的所有进程的列表。

通过使用 ID 参数指定 **ProcessId**，还可以返回单个进程。以下示例将返回系统空闲进程：

```
PS> Get-Process -Id 0
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
0	0	0	16	0	0	Idle	

尽管在某些情况下 **cmdlet** 不返回任何数据是正常的，但按其 **ProcessId** 指定进程时，如果 **Get-Process** 找不到匹配项，它将生成错误，因为它的常见用途是检索已知的正在运行的进程。如果不存在具有该 ID 的进程，则很可能是 ID 不正确，或者感兴趣的进程已经退出：

```
PS> Get-Process -Id 99
Get-Process :找不到 ID 为 99 的进程。
所在行:1 字符:12
+ Get-Process <<<< -Id 99
```

**Name** 参数可以用来基于进程名称指定进程的子集。因为进程可以有相同名称，所以输出可能包括多个进程。如果不存在具有该名称的进程，则 **Get-Process** 将像指定 **ProcessId** 时一样返回错误。例如，如果指定进程名称 **explore** 而不是 **explorer**：

```
PS> Get-Process -Name explore
Get-Process :找不到名称为“explore”的进程。
所在行:1 字符:12
+ Get-Process <<<< -Name explore
```

**Name** 参数支持使用通配符，因此可以键入名称的前几个字符并且后跟星号来获得列表：

```
PS> Get-Process -Name ex*
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
234	7	5572	12484	134	2.98	1684	EXCEL
555	15	34500	12384	134	105.25	728	explorer

### 请注意：

因为 **.NET System.Diagnostics.Process** 类是 Windows PowerShell 进程的基础，因此它遵从 **System.Diagnostics.Process** 所使用的某些约定。这些约定之一是，可执行文件的进程名称永远不包括可执行文件名末尾的 **“.exe”**。

**Get-Process** 还将接受 **Name** 参数的多个值。像指定单个名称一样，如果无法匹配名称，则尽管仍会获得匹配进程的正常输出，但将显示错误：

```
PS> Get-Process -Name exp*,power*,NotAProcess
Get-Process :找不到名称为“NotAProcess”的进程。
所在行:1 字符:12
+ Get-Process <<<< -Name exp*,power*,svchost,NotAProcess
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
540	15	35172	48148	141	88.44	408	explorer
605	9	30668	29800	155	7.11	3052	powershell

## 停止进程 (Stop-Process)

Windows PowerShell 可让您灵活地列出进程，但如何停止进程呢？

**Stop-Process** cmdlet 采用 **Name** 或 **ID** 来指定希望停止的进程。是否能够停止进程取决于您的权限。某些进程不能停止。例如，如果试图停止空闲进程，则将获得错误：

```
PS> Stop-Process -Name Idle
Stop-Process :由于以下错误无法停止进程“Idle (0)”：
访问被拒绝
所在行:1 字符:13
+ Stop-Process <<<< -Name Idle
```

还可以用 **Confirm** 参数强制进行提示。如果指定进程名称时使用通配符，则此参数特别有用，因为您可能意外匹配一些不想停止的进程：

```
PS> Stop-Process -Name t*,e* -Confirm
确认
是否确实要执行此操作？
对目标“explorer (408)”执行操作“Stop-Process”。
[Y] 是 [A] 全是 [N] 否 [L] 全否 [S] 挂起 [?] 帮助
(默认值为“Y”):n
确认
是否确实要执行此操作？
对目标“taskmgr (4072)”执行操作“Stop-Process”。
[Y] 是 [A] 全是 [N] 否 [L] 全否 [S] 挂起 [?] 帮助
(默认值为“Y”):n
```

通过使用某些对象筛选 cmdlet，可以进行复杂的进程操作。由于进程对象有 **Responding** 属性，当进程不再响应时该属性将为 **True**，因此可以用以下命令停止所有无响应的应用程序：

```
Get-Process | Where-Object -FilterScript {$_.Responding -eq $false} | Stop-Process
```

您可以在其他情况下使用相同的方法。例如，假设用户启动一个应用程序时另一个辅助的系统任务栏应用程序自动运行。您可能发现这在终端服务会话中无法正确工作，但仍然需要使它在物理计算机控制台上的会话中持续运行。连接到物理计算机桌面的会话的会话 ID 始终是 0，因此通过使用 **Where-Object** 和进程的 **SessionId**，可以停止在其他会话中的所有进程实例：

```
Get-Process -Name BadApp | Where-Object -FilterScript {$_.SessionId -neq 0} |
Stop-Process
```

## 停止所有其他 Windows PowerShell 会话

可能偶尔需要能够停止除了当前会话以外所有正在运行的 Windows PowerShell 会话。如果会话正在使用太多资源，或者不可访问（它可能正在远程运行，或者在另一个桌面会话中），则可能无法直接停止它。但是，如果试图停止所有正在运行的会话，则可能终止当前会话。

每个 Windows PowerShell 会话都有环境变量 **PID**，其中包含 Windows PowerShell 进程的 ID。可以对照每个会话的 ID 检查该 **\$PID**，并只终止有不同 ID 的 Windows PowerShell 会话。以下管道命令执行此操作，并返回被终止会话的列表（由于使用了 **PassThru** 参数）：

```
PS> Get-Process -Name powershell | Where-Object -FilterScript {$_.Id -ne $PID} |
Stop-Process -
PassThru
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
334	9	23348	29136	143	1.03	388	powershell
304	9	23152	29040	143	1.03	632	powershell

302	9	20916	26804	143	1.03	1116 powershell
335	9	25656	31412	143	1.09	3452 powershell
303	9	23156	29044	143	1.05	3608 powershell
287	9	21044	26928	143	1.02	3672 powershell

## 管理本地服务

共有八个核心服务 **cmdlet**，它们是为众多的服务任务设计的。我们将只介绍如何列出和更改服务的运行状态，但您可以使用 **Get-Help \*-Service** 获得服务 **cmdlet** 的列表，还可以使用 **Get-Help<Cmdlet 名称>**（例如，**Get-Help New-Service**）查找有关每个服务 **cmdlet** 的信息。

### 列出服务

通过使用 **Get-Service**，可以枚举计算机上的本地服务。与 **Get-Process** 一样，无参数使用 **Get-Service** 命令将返回所有服务。可以按名称筛选，甚至使用星号通配符：

```
PS> Get-Service -Name se*
Status  Name          DisplayName
-----  ----
Running seclogon      Secondary Logon
Running SENS         System Event Notification
Stopped ServiceLayer ServiceLayer
```

由于服务的真实名称是什么并非始终是很明显的，因此您可能发现需要按显示名查找服务。可以按具体名称、使用通配符或使用显示名列表来执行该操作：

```
PS> Get-Service -DisplayName se*
Status  Name          DisplayName
-----  ----
Running lanmanserver Server
Running SamSs   Security Accounts Manager
Running seclogon Secondary Logon
Stopped ServiceLayer ServiceLayer
Running wscsvc  Security Center
PS> Get-Service -DisplayName ServiceLayer,Server
Status  Name          DisplayName
-----  ----
Running lanmanserver Server
Stopped ServiceLayer ServiceLayer
```

### 停止、启动、挂起和重新启动服务

所有服务 **cmdlet** 都有相同的常规形式。可以按公用名或显示名指定服务，并且可以用列表和通配符作为值。若要停止后台打印程序，请使用：

```
Stop-Service -Name spooler
```

若要在后台打印程序停止之后启动它，请使用：

```
Start-Service -Name spooler
```

若要挂起后台打印程序，请使用：

```
Suspend-Service -Name spooler
```

**Restart-Service** cmdlet 的工作方式与其他服务 cmdlet 相同，但我们将介绍它的一些更复杂的示例。在最简单的使用中，可以指定服务的名称：

```
PS> Restart-Service -Name spooler
警告:正在等待服务“Print Spooler (Spooler)”完成启动...
警告:正在等待服务“Print Spooler (Spooler)”完成启动...
PS>
```

您将注意到，系统会显示有关后台打印程序正在启动的重复警告消息。在执行需要占用一定时间的服务操作时，Windows PowerShell 将通知您它仍在尝试执行该任务。

如果要重新启动多个服务，可以先获得服务列表，并对它们进行筛选，然后执行重新启动：

```
PS> Get-Service | Where-Object -FilterScript {$_.CanStop} | Restart-Service
警告:正在等待服务“Computer Browser (Browser)”完成停止...
警告:正在等待服务“Computer Browser (Browser)”完成停止...
Restart-Service :无法停止服务“Logical Disk Manager (dmserver)”，因为具有依赖它的服务。只有在设置了 Force 标志时才能停止该服务。
所在行:1 字符:57
+ Get-Service | Where-Object -FilterScript {$_.CanStop} | Restart-Service <<<<
警告:正在等待服务“Print Spooler (Spooler)”完成启动...
警告:正在等待服务“Print Spooler (Spooler)”完成启动...
```

## 收集有关计算机的信息

**Get-WmiObject** 是用于执行常规系统管理任务的最重要 cmdlet。所有关键的子系统设置都是通过 WMI 公开的。此外，WMI 将数据视为有一个或多个项目的集合中的对象。由于 Windows PowerShell 还能处理对象，并且它的管道允许您以相同方式对待单个或多个对象，因此，通用 WMI 访问可让您用非常少的工作量执行一些高级任务。


以下示例演示如何通过对任意计算机使用 **Get-WmiObject** 来收集特定信息。我们用表示本地计算机的点值 (.) 指定 **ComputerName** 参数。您可以指定与可以通过 WMI 访问的任何计算机关联的名称或 IP 地址。若要检索有关本地计算机的信息，可以省略 **-ComputerName**。

### 列出桌面设置

我们首先介绍用于收集本地计算机的桌面相关信息的命令。

```
Get-WmiObject -Class Win32_Desktop -ComputerName .
```

此命令将返回所有桌面的信息，无论它们是否正在使用。

 请注意：

某些 WMI 类返回的信息可能非常详细，并且通常包含有关 WMI 类的元数据。由于这些元数据属性的名称大多数以双下划线开头，因此可以使用 **Select-Object** 筛选这些属性。

请使用 **[a-z]\*** 作为 **Property** 值来仅指定以字母字符开头的属性。例如：

```
Get-WmiObject -Class Win32_Desktop -ComputerName . | Select-Object -Property [a-z]*
```

若要筛选元数据，请使用管道运算符 (|) 将 **Get-WmiObject** 命令的结果发送到 **Select-Object -Property [a-z]\***。

## 列出 BIOS 信息

WMI Win32\_BIOS 类可以返回有关本地计算机系统 BIOS 的相当精简和完整的信息:

```
Get-WmiObject -Class Win32_BIOS -ComputerName .
```

## 列出处理器信息

可以使用 WMI 的 **Win32\_Processor** 类检索常规处理器信息, 但还可能需筛选这些信息:

```
Get-WmiObject -Class Win32_Processor -ComputerName . | Select-Object -Property [a-z]*
```

若要得到处理器系列的一般说明字符串, 只需返回 **Win32\_ComputerSystemSystemType** 属性:

```
PS> Get-WmiObject -Class Win32_ComputerSystem -ComputerName . | Select-Object  
-Property SystemType  
SystemType  
-----  
X86-based PC
```

## 列出计算机制造商和型号

还可以从 **Win32\_ComputerSystem** 获得计算机型号信息。标准的显示输出不需要进行任何筛选, 即可提供 OEM 数据:

```
PS> Get-WmiObject -Class Win32_ComputerSystem  
Domain : WORKGROUP  
Manufacturer : Compaq Presario 06  
Model : DA243A-ABA 6415c1 NA910  
Name : MyPC  
PrimaryOwnerName : Jane Doe  
TotalPhysicalMemory : 804765696
```

像这样的命令输出 (直接从某些硬件返回信息) 实际上只是您拥有的数据。某些信息未被硬件制造商正确配置, 因此可能不可用。

## 列出已安装的修补程序

通过使用 **Win32\_QuickFixEngineering** 可以列出所有已安装的修补程序:

```
Get-WmiObject -Class Win32_QuickFixEngineering -ComputerName .
```

该类返回修补程序的列表, 如下所示:

```
Description : Update for Windows XP (KB910437)  
FixComments : Update  
HotFixID : KB910437  
Install Date :  
InstalledBy : Administrator  
InstalledOn : 12/16/2005  
Name :  
ServicePackInEffect : SP3  
Status :
```



若要得到更简洁的输出，可能需要排除某些属性。尽管可以使用 **Get-WmiObject Property** 参数只选择 **HotFixID**，但这样做实际上将返回更多信息，因为默认情况下将显示所有元数据：

```
PS> Get-WmiObject -Class Win32_QuickFixEngineering -ComputerName .-Property
HotFixId
HotFixID           : KB910437
__GENUS            : 2
__CLASS            : Win32_QuickFixEngineering
__SUPERCLASS       :
__DYNASTY          :
__RELPATH          :
__PROPERTY_COUNT   : 1
__DERIVATION       : {}
__SERVER           :
__NAMESPACE        :
__PATH             :
```

还会返回其他数据，因为 **Get-WmiObject** 中的 **Property** 参数限制从 WMI 类实例返回的属性，而不限限制返回到 Windows PowerShell 的对象。若要减少输出，请使用 **Select-Object**：

```
PS> Get-WmiObject -Class Win32_QuickFixEngineering -ComputerName .-Property Hot
FixId | Select-Object -Property HotFixId
HotFixId
-----
KB910437
```

## 列出操作系统版本信息

**Win32\_OperatingSystem** 类属性包括版本和 **Service Pack** 信息。可以只显式选择这些属性，以便从 **Win32\_OperatingSystem** 获得版本信息摘要：

```
Get-WmiObject -Class Win32_OperatingSystem -ComputerName . | Select-Object
-Property
BuildNumber, BuildType, OSType, ServicePackMajorVersion, ServicePackMinorVersion
```

还可以在 **Select-Object Property** 参数中使用通配符。因为所有以 **Build** 或 **ServicePack** 开头的属性在这里都是重要的，因此可以将该命令缩短为以下形式：

```
PS> Get-WmiObject -Class Win32_OperatingSystem -ComputerName . | Select-Object
-Property Build*, OSType, ServicePack*

BuildNumber           : 2600
BuildType              : Uniprocessor Free
OSType                : 18
ServicePackMajorVersion : 2
ServicePackMinorVersion : 0
```

## 列出本地用户和所有者

通过选择 **Win32\_OperatingSystem** 属性可以查找本地常规用户信息（许可用户数、当前用户数和所有者名称）。可以显式选择要显示的属性，如下所示：

```
Get-WmiObject -Class Win32_OperatingSystem -ComputerName . | Select-Object
-Property NumberOfLicensedUsers, NumberOfUsers, RegisteredUser
```

使用通配符的更简洁版本是：

```
Get-WmiObject -Class Win32_OperatingSystem -ComputerName . | Select-Object  
-Property *user*
```

## 获得可用磁盘空间

若要查看本地驱动器的磁盘空间和可用空间，可以使用 WMI Win32\_LogicalDisk 类。这需要只显示 DriveType 为 3（这是 WMI 为固定硬盘分配的值）的实例。

```
Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" -ComputerName .  
  
DeviceID      :C:  
DriveType     : 3  
ProviderName  :  
FreeSpace     : 65541357568  
Size          : 203912880128  
VolumeName    :Local Disk  
  
DeviceID      :Q:  
DriveType     : 3  
ProviderName  :  
FreeSpace     : 44298250240  
Size          : 122934034432  
VolumeName    :New Volume  
  
PS> Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" -ComputerName . |  
Measure-Object -Property FreeSpace,Size -Sum  
  
Get-WmiObject -Class Win32_LogicalDisk -Filter "DriveType=3" -ComputerName . |  
Measure-Object -Property FreeSpace,Size -Sum | Select-Object -Property  
Property,Sum
```

## 获得登录会话信息

通过 WMI Win32\_LogonSession 类可以获得与用户关联的登录会话的常规信息：

```
Get-WmiObject -Class Win32_LogonSession -ComputerName .
```

## 获得登录到计算机的用户

使用 Win32\_ComputerSystem 可以显示登录到特定计算机系统的用户。此命令只返回登录到系统桌面的用户：

```
Get-WmiObject -Class Win32_ComputerSystem -Property UserName -ComputerName .
```

## 从计算机获得本地时间

使用 WMI Win32\_LocalTime 类可以在特定计算机上检索当前本地时间。因为默认情况下该类显示所有元数据，所以可能需要使用 **Select-Object** 对这些数据进行筛选：

```
PS> Get-WmiObject -Class Win32_LocalTime -ComputerName . | Select-Object -Property  
[a-z]*
```

```

Day           : 15
DayOfWeek     : 4
Hour          : 12
Milliseconds  :
Minute        : 11
Month         : 6
Quarter       : 2
Second        : 52
WeekInMonth   : 3
Year          : 2006

```

## 显示服务状态

若要查看特定计算机上所有服务的状态，可以像前面提到的那样在本地使用 **Get-Service** cmdlet。对于远程系统，可以使用 WMI Win32\_Service 类。如果还使用 **Select-Object** 来筛选 **Status**、**Name** 和 **DisplayName** 的结果，则输出格式将与 **Get-Service** 几乎相同：


```
Get-WmiObject -Class Win32_Service -ComputerName . | Select-Object -Property
Status,Name,DisplayName
```

若要允许完整显示名称非常长的服务的名称，可能需要使用 **Format-Table** 及其 **AutoSize** 和 **Wrap** 参数，以优化列宽并允许长名称换行而不被截断：

```
Get-WmiObject -Class Win32_Service -ComputerName . | Format-Table -Property
Status,Name,DisplayName -AutoSize -Wrap
```

## 处理软件安装

通过 WMI 的 **Win32\_Product** 类可以访问正确设计以使用 Windows Installer 的应用程序，但是并非目前使用的所有应用程序都使用 Windows Installer。由于 Windows Installer 为处理可安装的应用程序提供了范围最广的各种标准技术，因此我们将集中讨论这些应用程序。使用备用安装例程的应用程序通常不受 Windows Installer 管理。处理这些应用程序的具体技术将取决于安装程序软件 and 应用程序开发人员的决定。

 请注意：

对于通过将应用程序文件复制到计算机的方式进行安装的应用程序来说，通常无法使用此处讨论的技术对它们进行管理。可以使用在“处理文件和文件夹”一节中讨论的技术，将这些应用程序作为文件和文件夹进行管理。

### 列出 Windows Installer 应用程序

使用简单的 WMI 查询，可以很容易地枚举在本地或远程系统上用 Windows Installer 安装的应用程序：

```

PS> Get-WmiObject -Class Win32_Product -ComputerName .
IdentifyingNumber : {7131646D-CD3C-40F4-97B9-CD9E4E6262EF}
Name               : Microsoft .NET Framework 2.0
Vendor            : Microsoft Corporation

```


```
Version      : 2.0.50727
Caption      :Microsoft .NET Framework 2.0
```

由于需要使用默认情况下不显示的其他信息，因此您可能仍然需要使用 **Select-Object**。如果尝试查找 **Microsoft .NET Framework 2.0** 的软件包缓存位置，则可以使用以下命令：

```
PS> Get-WmiObject -Class Win32_Product -ComputerName . | Where-Object -FilterScript
{$_Name -eq "Microsoft .NET Framework 2.0"} | Select-Object -Property [a-z]*
Name
:Microsoft .NET Framework 2.0
Version
: 2.0.50727
InstallState
: 5
Caption
:Microsoft .NET Framework 2.0
Description
:Microsoft .NET Framework 2.0
IdentifyingNumber
:{7131646D-CD3C-40F4-97B9-CD9E4E6262EF}
InstallDate
: 20060506
InstallDate2
: 20060506000000.000000-000
InstallLocation
:
PackageCache
:C:\WINDOWS\Installer\619ab2.msi
SKUNumber
:
Vendor
:Microsoft Corporation
```

另外，可以使用 **Get-WmiObject Filter** 参数以便只选择 **Microsoft .NET Framework 2.0**。在此命令中使用的筛选器是 **WMI 筛选器**，因此它不使用 **Windows PowerShell** 筛选语法，而是使用 **WMI 查询语言 (WQL)**：

```
Get-WmiObject -Class Win32_Product -ComputerName . -Filter "Name='Microsoft .NET
Framework 2.0'" | Select-Object -Property [a-z]*
```

 请注意：

**WQL** 查询常用的字符（例如，空格或等于号）在 **Windows PowerShell** 中有特殊含义。因此，谨慎的做法是始终将 **Filter** 参数的值放在一对引号内。还可以使用 **Windows PowerShell** 转义字符，即倒引号 (```)，但它可能不会提高可读性。以下命令相当于前面的命令，并返回相同结果，但是使用倒引号 “```” 会将特殊字符转义，而不是将整个筛选器字符串放在引号内：

```
Get-WmiObject -Class Win32_Product -ComputerName . -Filter Name=`'Microsoft` .NET`
Framework` 2.0`' | Select-Object -Property [a-z]*
```

生成精简列表的另一种方式是显式地选择显示格式。若要标识已安装的特定软件包，则以下输出中显示的属性最有用：

```
Get-WmiObject -Class Win32_Product -ComputerName . | Format-List
Name,InstallDate,InstallLocation,PackageCache,Vendor,Version,IdentifyingNumber
...
Name
:HighMAT Extension to Microsoft Windows XP CD Writing Wizard
InstallDate
: 20051022
InstallLocation
:C:\Program Files\HighMAT CD Writing Wizard\
PackageCache
:C:\WINDOWS\Installer\113b54.msi
Vendor
:Microsoft Corporation
Version
: 1.1.1905.1
IdentifyingNumber
:{FCE65C4E-B0E8-4FBD-AD16-EDCBE6CD591F}
...
```

最后，若要仅查找已安装的应用程序的名称，一个简单的 **Format-Wide** 语句即可完成减少输

出的工作：

```
Get-WmiObject -Class Win32_Product -ComputerName . | Format-Wide -Column 1
```

尽管现在已有几种方式来查看使用 Windows Installer 完成安装的应用程序，但我们尚未考虑其他应用程序。由于大多数标准应用程序会向 Windows 注册它们的卸载程序，因此可以在 Windows 注册表中查找这些程序，以便在本地处理它们。

## 列出所有可卸载的应用程序

尽管不能保证找到系统中的所有应用程序，但可以找到在“添加或删除程序”对话框中列出的所有程序。“添加或删除程序”将从注册表项 **HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall** 下面的列表中查找这些应用程序，并且我们还可以自己检查此项，以查找应用程序。若要使查看 Uninstall 项更容易，可以将 Windows PowerShell 驱动器映射到此注册表位置：

```
PS> New-PSDrive -Name Uninstall -PSProvider Registry -Root
HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall
```

Name	Provider	Root	CurrentLocation
Uninstall	Registry	HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall	

 请注意：


**HKLM:** 驱动器映射到 **HKEY\_LOCAL\_MACHINE** 的根，因此我们在 Uninstall 项的路径中使用该驱动器。如果不使用 **HKLM:**，则可能使用 **HKLM** 或 **HKEY\_LOCAL\_MACHINE** 指定注册表路径。使用现有注册表驱动器的优点是我们可以使用 Tab 补齐功能来填充项名称，所以不需要键入它们。

现在，我们有了一个名为“Uninstall”的驱动器，可以用它来快速和方便地查找应用程序的安装信息。通过统计 Uninstall: Windows PowerShell 驱动器中的注册表项数，可以查找已安装的应用程序个数：

```
PS> (Get-ChildItem -Path Uninstall:).Length
459
```

可以从 **Get-ChildItem** 开始，使用多种技术进一步搜索此应用程序列表。若要应用程序列表放到 **\$UninstallableApplications** 变量中，可以执行以下命令：

```
$UninstallableApplications = Get-ChildItem -Path Uninstall:
```

 请注意：

在此处使用长变量名称是为了清晰起见。在实际使用中，不必使用长名称。可以使用 Tab 补齐功能自动填写变量名称，另外还可以使用 1-2 个字符的名称加快速度。如果正在开发希望重用的代码，则较长的说明性名称最有用。

通过使用以下命令可以在 Uninstall 项中找到应用程序的显示名：

```
PS> Get-ChildItem -Path Uninstall: | ForEach-Object -Process
{ $_.GetValue("DisplayName") }
```

无法保证这些值是唯一的。在以下示例中，两个已安装项都显示为“Windows Media Encoder 9 Series”：

```
PS> Get-ChildItem -Path Uninstall: | Where-Object -FilterScript
{ $_.GetValue("DisplayName") -eq "Windows Media Encoder 9 Series"}

Hive:Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall

SKC  VC Name                                     Property
---  --  ----                                     -
0    3 Windows Media Encoder 9                 {DisplayName, DisplayIcon, UninstallS...
0    24 {E38C00D0-A68B-4318-A8A6-F7...{AuthorizedCDFPrefix, Comments, Conta...
```

## 安装应用程序

可以使用 **Win32\_Product** 类在远程或本地安装 Windows Installer 软件包。远程安装时，应当以经典的通用命名约定 (UNC) 网络路径的格式指定要安装的 .msi 软件包的路径，这是因为 WMI 子系统不理解 Windows PowerShell 路径。例如，若要将位于网络共享 \\AppSrv\dsp 上的 MSI 软件包 NewPackage.msi 安装到远程计算机 PC01 上，可以在 Windows PowerShell 提示符下输入以下命令：

```
(Get-WmiObject -ComputerName PC01 -List | Where-Object -FilterScript {$_.Name -eq
"Win32_Product"}).InvokeMethod("Install","\\AppSrv\dsp\NewPackage.msi")
```

不使用 Windows Installer 技术的应用程序可能有特定于应用程序的方法来进行自动部署。若要查找是否有自动部署的方法，可能需要查看应用程序的文档，或咨询应用程序供应商的支持系统。在某些情况下，即使应用程序供应商没有专门将应用程序设计为自动安装，该安装程序软件制造商仍然可能有某些自动化的技术。

## 删除应用程序

使用 Windows PowerShell 删除 Windows Installer 软件包与使用 **InvokeMethod** 安装软件包的工作方式大体相同。下面是基于名称属性选择要卸载的软件包的示例；在某些情况下，用 **IdentifyingNumber** 进行筛选可能更容易：

```
(Get-WmiObject -Class Win32_Product -Filter "Name='ILMerge'" -ComputerName
.).InvokeMethod("Uninstall",$null)
```

删除其他应用程序并不那么简单，甚至在本地这样做时也如此。我们可以通过提取 **UninstallString** 属性来找到这些应用程序的命令行卸载字符串。此方法对 Windows Installer 应用程序和出现在 Uninstall 项下面的旧程序有效：

```
Get-ChildItem -Path Uninstall: | ForEach-Object -Process
{ $_.GetValue("UninstallString") }
```

如果需要，可以按显示名筛选输出：

```
Get-ChildItem -Path Uninstall: | Where-Object -FilterScript
{ $_.GetValue("DisplayName") -like "Win*" } | ForEach-Object -Process
{ $_.GetValue("UninstallString") }
```

但是，如果不进行某些修改，这些字符串可能无法在 Windows PowerShell 提示符下直接使用。

## 升级 Windows Installer 应用程序

执行应用程序升级涉及两条信息。您需要知道要升级的已安装应用程序的名称，以及应用程序升级软件包的路径。有了该信息，就可以使用单个命令行从 Windows PowerShell 执行升级：

```
(Get-WmiObject -Class Win32_Product -ComputerName .-Filter
"Name='OldAppName']").InvokeMethod("Upgrade", "\\AppSrv\dsp\OldAppUpgrade.msi")
```

## 更改计算机状态：锁定、注销、关闭和重新启动

您可以从 Windows PowerShell 以几种不同方法重置计算机，但是在最初的发行版中必须使用标准命令行工具或 WMI。虽然仅使用 Windows PowerShell 来调用特定工具，但是完成更改计算机电源状态的步骤可说明使用外部工具的一些重要详细信息。

### 锁定计算机

使用标准可用工具直接锁定计算机的唯一方法是，直接调用 **user32.dll** 中的 **LockWorkstation()** 函数：

```
rundll32.exe user32.dll,LockWorkStation
```

此命令将立即锁定工作站。在诸如 Windows XP 之类的操作系统上，由于快速用户切换处于活动状态，计算机将返回到用户登录屏幕，而不是启动当前用户的屏幕保护程序。在可能希望断开特定会话的终端服务器上，也可以使用 **tsshutdn.exe** 命令行工具。

### 注销当前会话

可以使用几种不同方法在本地系统上注销会话。最简单的方法是，使用远程桌面/终端服务命令行工具 **logoff.exe**（在 Windows PowerShell 或命令外壳程序提示符下键入 **logoff /?** 可查看详细使用信息）。若要注销当前的活动会话，请键入不带参数的 **logoff**。

另一种方法是使用 **shutdown.exe** 工具及其注销选项：

```
shutdown.exe -l
```

第三种方法是使用 WMI。Win32\_OperatingSystem 类具有 Win32Shutdown 方法；用参数 0 调用该方法可启动注销：

```
(Get-WmiObject -Class Win32_OperatingSystem -ComputerName
.).InvokeMethod("Win32Shutdown",0)
```

### 关闭或重新启动计算机

关闭和重新启动计算机通常是相同类型的任务。用于关闭计算机的工具通常也可以重新启动它，反之亦然。有两种简单的方法可以从 Windows PowerShell 重新启动计算机。使用带适当参数的 **tsshutdn.exe** 或 **shutdown.exe**。使用 **tsshutdn.exe /?** 或 **shutdown.exe /?** 可以获取详细的用法信息。

直接从 Windows PowerShell 使用 **Win32\_OperatingSystem** 执行关闭和重新启动操作也是可能的。但是，此类实现的详细信息已超出此“Windows PowerShell 入门”的范围。

# 处理打印机

---

在 Windows PowerShell 中，可以使用 WMI 和来自 WSH 的 **WScript.Network** COM 对象执行打印机管理任务。我们将同时使用这两种工具演示特定的任务。

## 列出打印机连接

列出安装在计算机上的打印机的最简单方法是，使用 WMI **Win32\_Printer** 类：

```
Get-WmiObject -Class Win32_Printer -ComputerName .
```

也可以使用 **WScript.Network** COM 对象（通常在 WSH 脚本中使用）列出打印机：

```
(New-Object -ComObject WScript.Network).EnumPrinterConnections()
```

此命令返回端口名称和打印机设备名称的简单字符串集合，不含可区别标签，因此对于轻松检查不太有用。

## 添加网络打印机

使用 **WScript.Network** 可以非常轻松地添加新的网络打印机：

```
(New-Object -ComObject  
WScript.Network).AddWindowsPrinterConnection("\\Printserver01\Xerox5")
```

## 设置默认打印机

若要使用 WMI 设置默认打印机，需要向下筛选 **Win32\_Printer** 集合找到所需的打印机，然后调用 **SetDefaultPrinter** 方法：

```
(Get-WmiObject -ComputerName .-Class Win32_Printer -Filter "Name='HP LaserJet  
5Si'").InvokeMethod("SetDefaultPrinter",$null)
```

**WScript.Network** 使用起来更简单一点；它也具有 **SetDefaultPrinter** 方法，您只需将打印机名称指定为参数即可：

```
(New-Object -ComObject WScript.Network).SetDefaultPrinter('HP LaserJet 5Si')
```

## 删除打印机连接

可以使用 **WScript.Network RemovePrinterConnection** 方法删除打印机连接：

```
(New-Object -ComObject  
WScript.Network).RemovePrinterConnection("\\Printserver01\Xerox5")
```

# 执行网络任务

---

大多数低级网络协议管理任务都涉及 TCP/IP，因为 TCP/IP 是最常用的网络协议。我们将看一看如何使用 WMI 从 Windows PowerShell 完成其中的一些任务。



## 列出计算机的 IP 地址

使用以下命令可以返回计算机正使用的所有 IP 地址：

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=TRUE
-ComputerName . | Select-Object -Property IPAddress
```

此命令的输出与大多数属性列表不同，因为值将括在方括号中：

```
IPAddress
-----
{192.168.1.80}
{192.168.148.1}
{192.168.171.1}
{0.0.0.0}
```

若要了解为什么将它们括在方括号中，可以使用 **Get-Member** 仔细地查看 **IPAddress** 属性：

```
PS> Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=TRUE
-ComputerName . | Get-Member -Name IPAddress
```

```
TypeName: System.Management.ManagementObject#root\cimv2\Win32_NetworkAdapter
Configuration
```

```
Name      MemberType Definition
-----
IPAddress Property      System.String[] IPAddress {get;}
```

每个网络适配器的 **IPAddress** 属性实际上是一个数组。定义中的大括号指示 **IPAddress** 不是 **System.String** 值，而是 **System.String** 值的数组。


可以使用 **Select-Object ExpandProperty** 参数扩展这些值：

```
PS> Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=TRUE
-ComputerName . | Select-Object -ExpandProperty IPAddress
192.168.1.80
192.168.148.1
192.168.171.1
0.0.0.0
```

## 列出 IP 配置数据

若要为每个网络适配器显示详细的 IP 配置数据，可以使用以下命令：

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=TRUE
-ComputerName .
```

 请注意：

网络适配器配置的默认显示是非常精简的一组可用信息。有关深入检查和疑难解答，请使用 **Select-Object** 强制显示更多属性。如果对 **IPX** 或 **WINS** 属性不感兴趣（很可能是在使用现代 **TCP/IP** 网络的情况下），也可以排除以 **WINS** 或 **IPX** 开头的所有属性：

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=TRUE
-ComputerName . | Select-Object -Property [a-z]* -ExcludeProperty IPX*,WINS*
```

此命令将返回有关 **DHCP**、**DNS**、路由和其他次要 **IP** 配置属性的详细信息。

## 对计算机执行 Ping 操作

可以使用 **Win32\_PingStatus** 对计算机执行简单的 Ping 操作。以下命令将执行 Ping 操作，但是返回冗长的输出：

```
Get-WmiObject -Class Win32_PingStatus -Filter "Address='127.0.0.1'" -ComputerName .
```

摘要信息的更有用形式是仅显示 **Address**、**ResponseTime** 和 **StatusCode** 属性：

```
PS> Get-WmiObject -Class Win32_PingStatus -Filter "Address='127.0.0.1'" -ComputerName . | Select-Object -Property Address,ResponseTime,StatusCode
```

Address	ResponseTime	StatusCode
127.0.0.1	0	0

状态代码 0 指示 Ping 操作成功。


可以使用数组对整个系列的计算机执行 Ping 操作。因为此处使用多个地址，所以需要使用 **ForEach-Object** 分别对每个地址执行 Ping 操作：

```
"127.0.0.1","localhost","research.microsoft.com" | ForEach-Object -Process {Get-WmiObject -Class Win32_PingStatus -Filter ("Address='" + $_ + "'") -ComputerName .}| Select-Object -Property Address,ResponseTime,StatusCode
```

可以使用相同的过程对整个子网执行 Ping 操作。例如，若要检查使用网络号 192.168.1.0 的专用网络，并使用标准的 C 类子网掩码 (255.255.255.0)，则只有 192.168.1.1 到 192.168.1.254 的范围内的地址才是合法的本地地址（0 始终是为网络号保留的，255 是子网广播地址）。

可以在 Windows PowerShell 中使用语句 **1..254** 获取从 1 到 254 的数字的数组，因此通过生成该数组并将值添加到 Ping 语句中的部分地址，可以对完整子网执行 Ping 操作：

```
1..254 | ForEach-Object -Process {Get-WmiObject -Class Win32_PingStatus -Filter ("Address='192.168.1.'" + $_ + "'") -ComputerName .}| Select-Object -Property Address,ResponseTime,StatusCode
```

 请注意：

该生成地址范围的方法也可以在别处使用。可以使用此方法生成一组完整地址：

```
$ips = 1..254 | ForEach-Object -Process {"192.168.1." + $_}
```

## 检索网络适配器属性

本入门中前面已经提到，可以使用 **Win32\_NetworkAdapterConfiguration** 检索常规配置属性。虽然不是严格的 TCP/IP 信息，但是诸如 MAC 地址和适配器类型的网络适配器信息对于了解计算机的当前状况是很有用的。可以使用以下命令获取此信息的摘要：

```
Get-WmiObject -Class Win32_NetworkAdapter -ComputerName .
```

## 为网络适配器指定 DNS 域

可以使用 **Win32\_NetworkAdapterConfiguration SetDNSDomain** 方法自动完成要在自动名称解析中使用的 DNS 域的指定。因为独立地为每个网络适配器配置指定 DNS 域，所以需要使

用 **ForEach-Object** 语句来确保逐个选择适配器：

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=true
-ComputerName . | ForEach-Object -Process { $_.InvokeMethod("SetDNSDomain",
"fabrikam.com") }
```

在此处使用了筛选语句 `IPEnabled=true`，因为甚至在仅使用 TCP/IP 的网络上，计算机上的几个网络适配器配置也不是真正的 TCP/IP 适配器；它们是支持用于所有适配器的 RAS、PPTP、QoS 和其他服务的常规软件元素，因此没有它们自己的地址。

可以使用 **Where-Object** 而不是 **Get-WmiObject Filter** 筛选该命令：

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -ComputerName . | Where-
Object -FilterScript {$_.IPEnabled} | ForEach-Object -Process
{ $_.InvokeMethod("SetDNSDomain", "fabrikam.com") }
```

## 执行 DHCP 配置任务

修改 DHCP 详细信息涉及使用一组适配器，就像 DNS 配置一样。有数个可以使用 WMI 执行的不同操作，我们将介绍其中的几个常见任务。

### 确定启用 DHCP 的适配器

使用以下命令，可以在计算机上查找启用 DHCP 的适配器：

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "DHCPEnabled=true"
-ComputerName .
```

如果不查找有 IP 配置问题的适配器，则可以进一步将此限制为仅查找启用 IP 的适配器：

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "IPEnabled=true and
DHCPEnabled=true" -ComputerName .
```

### 检索 DHCP 属性

适配器的 DHCP 相关属性通常以 DHCP 开头，因此可以添加管道元素 **Select-Object -Property DHCP\*** 以查看适配器的摘要 DHCP 信息：

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "DHCPEnabled=true"
-ComputerName . | Select-Object -Property DHCP*
```

### 在每个适配器上启用 DHCP

若要在所有适配器上全局启用 DHCP，请使用

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter IPEnabled=true
-ComputerName . | ForEach-Object -Process { $_.InvokeMethod("EnableDHCP", $null) }
```

可以改用 **Filter** 语句 “`IPEnabled=true and DHCPEnabled=false`”，以免在已经启用 DHCP 处启用它，但是省略此步骤不会导致任何错误。

### 对特定适配器解除和续订 DHCP 租约


**Win32\_NetworkAdapterConfiguration** 具有 **ReleaseDHCPLease** 和 **RenewDHCPLease** 方法。

它们的使用方式是相同的。通常，如果仅需要为特定子网上的某个适配器解除或续订地址，则需要使用这些方法。筛选子网上适配器的最简单方法是，仅选择使用该子网网关的适配器配置。例如，以下命令将对本地计算机上正从 192.168.1.254 获取 DHCP 租约的适配器解除所有 DHCP 租约：

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "IPEnabled=true and DHCPEnabled=true" -ComputerName . | Where-Object -FilterScript {$_.DHCPServer -contains "192.168.1.254"} | ForEach-Object -Process {$_.InvokeMethod("ReleaseDHCPLease",$null)}
```

续订 DHCP 租约时的唯一更改是，调用 **RenewDHCPLease** 而不是 **ReleaseDHCPLease**：

```
Get-WmiObject -Class Win32_NetworkAdapterConfiguration -Filter "IPEnabled=true and DHCPEnabled=true" -ComputerName . | Where-Object -FilterScript {$_.DHCPServer -contains "192.168.1.254"} | ForEach-Object -Process {$_.InvokeMethod("RenewDHCPLease",$null)}
```

 请注意：

对远程计算机使用这些方法时，如果是通过解除或续订租约的适配器连接到远程计算机的，则可能失去对远程计算机的访问。

## 对所有适配器解除和续订 DHCP 租约

使用 **Win32\_NetworkAdapterConfiguration** 的 **ReleaseDHCPLeaseAll** 和 **RenewDHCPLeaseAll** 方法，可以对所有适配器执行全局 DHCP 地址解除或续订。但是，该命令必须适用于 WMI 类，而不是特定的适配器，因为全局解除和续订租约是对类执行的，而不是对特定适配器执行的。

通过列出所有 WMI 类，然后按名称仅选择所需类，可以获取对 WMI 类而不是类实例的引用。例如，以下命令返回 Win32\_NetworkAdapterConfiguration 类：

```
Get-WmiObject -List | Where-Object -FilterScript {$_.Name -eq "Win32_NetworkAdapterConfiguration"}
```

可以将整个命令视为类，然后对它调用 **ReleaseDHCPAdapterLease** 方法。在以下命令中，**Get-WmiObject** 和 **Where-Object** 管道元素括在圆括号中；这将强制首先计算它们：

```
( Get-WmiObject -List | Where-Object -FilterScript {$_.Name -eq "Win32_NetworkAdapterConfiguration"} ) .InvokeMethod("ReleaseDHCPLeaseAll", $null)
```

可以使用相同的命令格式调用 **RenewDHCPLeaseAll** 方法：

```
( Get-WmiObject -List | Where-Object -FilterScript {$_.Name -eq "Win32_NetworkAdapterConfiguration"} ) .InvokeMethod("RenewDHCPLeaseAll", $null)
```

## 创建网络共享

可以使用 **Win32\_Share Create** 方法创建网络共享：

```
(Get-WmiObject -List -ComputerName . | Where-Object -FilterScript {$_.Name -eq "Win32_Share"}).InvokeMethod("Create", ("C:\temp", "TempShare", 0, 25, "test share of the temp folder"))
```

也可以在 Windows PowerShell 中使用 **net share** 创建共享

```
net share tempshare=c:\temp /users:25 /remark:"test share of the temp folder"
```

## 删除网络共享

可以使用 **Win32\_Share** 删除网络共享，但是该过程与创建共享稍有不同，因为需要检索要删除的特定共享，而不是 **Win32\_Share** 类。以下语句将删除共享 “TempShare”：

```
(Get-WmiObject -Class Win32_Share -ComputerName .-Filter
"Name='TempShare'").InvokeMethod("Delete",$null)
```

也可以使用 **Net share** 删除它：

```
PS> net share tempshare /delete
tempshare 已成功删除。
```

## 连接 Windows 可访问的网络驱动器

使用 **New-PSDrive** 可以创建 Windows PowerShell 驱动器，但是这样创建的驱动器仅对 Windows PowerShell 可用。若要创建新的网络驱动器，可以使用 **WScript.Network** COM 对象。以下命令将共享 \\FPS01\users 映射到本地驱动器 B：

```
(New-Object -ComObject WScript.Network).MapNetworkDrive("B:", "\\FPS01\users")
```

也可以使用 **net use** 命令做到这一点：

```
net use B:\\FPS01\users
```

使用 **WScript.Network** 或 **net use** 映射的驱动器立即对 Windows PowerShell 可用。

## 处理文件和文件夹

在 Windows PowerShell 驱动器中导航和操作这些驱动器上的项目，与操作 Windows 物理磁盘驱动器上的文件和文件夹类似。本部分将讨论如何处理特定的文件和文件夹操作任务。

### 列出文件夹中的所有文件和文件夹

可以使用 **Get-ChildItem** 获取文件夹中的所有直接项。添加可选的 **Force** 参数可以显示隐藏项或系统项。例如，以下命令显示 Windows PowerShell 驱动器 C（与 Windows 物理驱动器 C 相同）的直接内容：

```
Get-ChildItem -Force C:\
```

该命令仅列出直接包含的项，与使用 Cmd.exe 的 **DIR** 命令或 UNIX 外壳程序中的 **ls** 非常类似。为了显示包含的项，还需要指定 **-Recurse** 参数。（这可能需要极长的时间才能完成。）

列出 C 驱动器上的所有内容：

```
Get-ChildItem -Force C:\ -Recurse
```

**Get-ChildItem** 可以使用其 **Path**、**Filter**、**Include** 和 **Exclude** 参数筛选项，但是这些操作通常仅基于名称。使用 **Where-Object**，可以基于项的其他属性执行复杂的筛选。

以下命令查找 Program Files 文件夹中上次修改日期晚于 2005 年 10 月 1 日并且既不小于 1 MB 也不大于 10 MB 的所有可执行文件：

```
Get-ChildItem -Path $env:ProgramFiles -Recurse -Include *.exe | Where-Object  
-FilterScript {($_.LastWriteTime -gt "2005-10-01") -and ($_.Length -ge 1m) -and  
($_.Length -le 10m)}
```

## 复制文件和文件夹

复制是使用 **Copy-Item** 进行的。以下命令将 C:\boot.ini 备份到 C:\boot.bak:

```
Copy-Item -Path c:\boot.ini -Destination c:\boot.bak
```

如果目标文件已经存在，则复制尝试将会失败。若要覆盖预先存在的目标，请使用 **Force** 参数：

```
Copy-Item -Path c:\boot.ini -Destination c:\boot.bak -Force
```

甚至在目标为只读时，此命令也有效。

复制文件夹的方法与此相同。以下命令以递归方式将文件夹 C:\temp\test1 复制到新文件夹 c:\temp\DeleteMe:

```
Copy-Item C:\temp\test1 -Recurse c:\temp\DeleteMe
```

也可以复制所选项。以下命令将 c:\data 中任何位置所包含的所有 .txt 文件复制到 c:\temp\text:

```
Copy-Item -Filter *.txt -Path c:\data -Recurse -Destination c:\temp\text
```

仍然可以使用其他工具执行文件系统复制。XCOPY、ROBOCOPY 和 COM 对象（如 **Scripting.FileSystemObject**）均可以在 Windows PowerShell 中使用。例如，可以使用 Windows Script Host **Scripting.FileSystem COM** 类将 C:\boot.ini 备份到 C:\boot.bak:

```
(New-Object -ComObject Scripting.FileSystemObject).CopyFile("c:\boot.ini",  
"c:\boot.bak")
```

## 创建文件和文件夹

在所有 Windows PowerShell 提供程序中，创建新项的方法都是相同的。如果 Windows PowerShell 提供程序具有多种类型的项（例如，FileSystem Windows PowerShell 提供程序区分目录和文件），则需要指定项类型。

以下命令创建新文件夹 C:\temp\New Folder:

```
New-Item -Path 'C:\temp\New Folder' -ItemType "directory"
```

以下命令创建新的空文件 C:\temp\New Folder\file.txt:

```
New-Item -Path 'C:\temp\New Folder\file.txt' -ItemType "file"
```

## 删除文件夹中的所有文件和文件夹

可以使用 **Remove-Item** 删除包含的项，但是，如果该项包含其他内容，将提示您确认删除。例如，如果尝试删除包含其他项的文件夹 C:\temp\DeleteMe，Windows PowerShell 会在删除该文件夹之前提示您进行确认：

```
Remove-Item C:\temp\DeleteMe
```

确认

C:\temp\DeleteMe 中的项具有子项，并且没有指定 **-recurse** 参数。如果继续，所有子项均将随该项删除。是否确实要继续？

[Y] 是 [A] 全是 [N] 否 [L] 全否 [S] 挂起 [?] 帮助

(默认值为“Y”)：

如果对于每个包含的项不希望被提示，请指定 **Recurse** 参数：

```
Remove-Item C:\temp\DeleteMe -Recurse
```

## 将本地文件夹映射为 Windows 可访问驱动器

也可以使用 **subst** 命令映射本地文件夹。以下命令创建位于本地 **Program Files** 目录中的本地驱动器 **P:**：

```
subst p:$env:programfiles
```

就像网络驱动器一样，使用 **subst** 在 Windows PowerShell 中映射的驱动器立即对 Windows PowerShell 会话可见。

## 将文本文件读入数组

文本数据的更常见存储格式之一是，存储在将不同行视为不同数据元素的文件中。可以使用 **Get-Content cmdlet** 一步读取整个文件，如下所示：

```
PS> Get-Content -Path C:\boot.ini
[boot loader]
timeout=5
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
/noexecute=AlwaysOff /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS=" Microsoft Windows XP Professional
with Data Execution Prevention" /noexecute=optin /fastdetect
```

**Get-Content** 已将从文件读取的数据视为一个数组，文件内容的每行上有一个元素。可以通过检查所返回内容的 **Length** 对此进行确认：

```
PS> (Get-Content -Path C:\boot.ini).Length
6
```

对于直接从 Windows PowerShell 获取信息列表，此命令是最有用的。例如，可能将计算机名称或 IP 地址的列表存储在文件 C:\temp\domainMembers.txt 中，文件的每行上有一个名称。可以使用 **Get-Content** 检索文件内容，并将其放置在变量 **\$Computers** 中：

```
$Computers = Get-Content -Path C:\temp\DomainMembers.txt
```

**\$Computers** 现在是其每个元素中都包含计算机名称的数组。

## 处理注册表项

由于注册表项是 Windows PowerShell 驱动器上的项，因此处理它们的方式与处理文件和文件

夹非常类似。一个关键差异是，基于注册表的 Windows PowerShell 驱动器上的每个项都是一个容器，就像文件系统驱动器上的文件夹一样。但是，注册表条目及其关联值是项的属性，而不是不同的项。

## 列出注册表项的所有子项

使用 **Get-ChildItem** 可以显示直接在注册表项中的所有项。添加可选的 **Force** 参数可以显示隐藏项或系统项。例如，此命令显示直接在 Windows PowerShell 驱动器 HKCU:（它对应于 HKEY\_CURRENT\_USER 注册表配置单元）中的项：

```
PS> Get-ChildItem -Path hkcu:\

Hive:Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER

SKC  VC Name                                     Property
---  --
2    0 AppEvents                                {}
7    33 Console                                {ColorTable00, ColorTable01, ColorTab...
25   1 Control Panel                          {Opened}
0    5 Environment                            {APR_ICONV_PATH, INCLUDE, LIB, TEMP...}
1    7 Identities                             {Last Username, Last User ...
4    0 Keyboard Layout                        {}
...
```

这些是在注册表编辑器 (Regedit.exe) 中 HKEY\_CURRENT\_USER 下可见的顶级项。也可以通过指定注册表提供程序的名称后跟 “::” 来指定此注册表路径。注册表提供程序的全名为 **Microsoft.PowerShell.Core\Registry**，但是它只需简写为 **Registry** 即可。下列任一命令将列出直接位于 HKCU 下的内容：

```
Get-ChildItem -Path Registry::HKEY_CURRENT_USER
Get-ChildItem -Path Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER
Get-ChildItem -Path Registry::HKCU
Get-ChildItem -Path Microsoft.PowerShell.Core\Registry::HKCU
Get-ChildItem HKCU:
```

这些命令仅列出直接包含的项，与使用 Cmd.exe 的 **DIR** 命令或 UNIX 外壳程序中的 **ls** 非常类似。若要显示包含的项，需要指定 **Recurse** 参数。若要列出 HKCU 中的所有注册表项，请使用以下命令（此操作可能需要极长的时间。）：

```
Get-ChildItem -Path hkcu:\ -Recurse
```

**Get-ChildItem** 可以通过其 **Path**、**Filter**、**Include** 和 **Exclude** 参数执行复杂的筛选功能，但是这些参数通常仅基于名称。使用 **Where-Objectcmdlet** 可以基于项的其他属性执行复杂的筛选。以下命令查找 HKCU:\Software 中具有不超过一个子项且正好具有四个值的所有项：

```
Get-ChildItem -Path HKCU:\Software -Recurse | Where-Object -FilterScript
{($_.SubKeyCount -le 1) -and ($_.ValueCount -eq 4) }
```

## 复制项

复制是使用 **Copy-Item** 进行的。以下命令将 HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion 及其所有属性复制到 HKCU:\，从而创建一个名为 “CurrentVersion” 的新项：



```
Copy-Item -Path 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion' -Destination
hkcu:
```

如果在注册表编辑器中或使用 **Get-ChildItem** 检查此新项，您将注意到新位置中没有所包含子项的副本。为了复制容器的所有内容，需要指定 **Recurse** 参数。若要使前面的复制命令是递归的，请使用以下命令：

```
Copy-Item -Path 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion' -Destination
hkcu:-Recurse
```

仍可以使用已经可用的其他工具执行文件系统复制。任何注册表编辑工具（包括 **reg.exe**、**regini.exe** 和 **regedit.exe**）和支持注册表编辑的 COM 对象（如 **WScript.Shell** 和 **WMI** 的 **StdRegProv** 类）都可以从 Windows PowerShell 内使用。

## 创建项

在注册表中创建新项比在文件系统中创建新项简单。由于所有的注册表项都是容器，因此无需指定项类型；只需提供显式路径即可，例如：

```
New-Item -Path hkcu:\software\_DeleteMe
```

也可以使用基于提供程序的路径指定项：

```
New-Item -Path Registry::HKCU\_DeleteMe
```

## 删除项

在本质上，删除项对于所有提供程序都是相同的。以下命令将以无提示方式删除项：

```
Remove-Item -Path hkcu:\Software\_DeleteMe
Remove-Item -Path 'hkcu:\key with spaces in the name'
```

## 删除特定项下的所有项

可以使用 **Remove-Item** 删除包含的项，但是如果该项包含任何其他内容，则会提示您确认删除。例如，如果尝试删除所创建的 **HKCU:\CurrentVersion** 子项，则会看到以下内容：

```
Remove-Item -Path hkcu:\CurrentVersion
```

确认

HKCU:\CurrentVersion\AdminDebug 中的项具有子项，并且没有指定 **-recurseparameter** 参数。如果继续，所有子项均将随该项删除。是否确实要继续？

[Y] 是 [A] 全是 [N] 否 [L] 全否 [S] 挂起 [?] 帮助

（默认值为“Y”）：

若要删除包含的项而不出现提示，请指定 **-Recurse** 参数：

```
Remove-Item -Path HKCU:\CurrentVersion -Recurse
```

若要删除 **HKCU:\CurrentVersion** 中的所有项但不删除 **HKCU:\CurrentVersion** 本身，则可以改用：

```
Remove-Item -Path HKCU:\CurrentVersion\* -Recurse
```

# 处理注册表条目

因为注册表条目是项的属性（因而无法直接浏览），所以在处理它们时需要采用稍微不同的方法。

## 列出注册表条目

可以使用许多不同的方法检查注册表条目。最简单的方法是获取与项关联的属性名称。例如，若要查看注册表项 **HKEY\_LOCAL\_MACHINE\Software\Microsoft\Windows\CurrentVersion** 中条目的名称，请使用 **Get-Item**。注册表项具有一个通用名称为“Property”的属性，该属性是项中注册表条目的列表。以下命令选择 **Property** 属性，并扩展项以便在列表中显示它们：

```
PS> Get-Item -Path
Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion | Select-
Object -ExpandProperty Property
DevicePath
MediaPathUnexpanded
ProgramFilesDir
CommonFilesDir
ProductId
```

若要以可读性更强的形式查看注册表条目，请使用 **Get-ItemProperty**：

```
PS> Get-ItemProperty -Path
Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion

PSPath           :Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SO
FTWARE\Microsoft\Windows\CurrentVersion
PSParentPath     :Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SO
FTWARE\Microsoft\Windows
PSChildName      :CurrentVersion
PSDrive          :HKLM
PSProvider       :Microsoft.PowerShell.Core\Registry
DevicePath       :C:\WINDOWS\inf
MediaPathUnexpanded :C:\WINDOWS\Media
ProgramFilesDir  :C:\Program Files
CommonFilesDir   :C:\Program Files\Common Files
ProductId        : 76487-338-1167776-22465
WallPaperDir     :C:\WINDOWS\Web\Wallpaper
MediaPath        :C:\WINDOWS\Media
ProgramFilePath  :C:\Program Files
PF_AccessoriesName :Accessories
(default)        :
```

项的 Windows PowerShell 相关属性均以“PS”为前缀，如 **PSPath**、**PSParentPath**、**PSChildName** 和 **PSProvider**。

可以使用“.”符号来表示当前位置。可以使用 **Set-Location** 首先转到 **CurrentVersion** 注册表容器：

```
Set-Location -Path
Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
```

或者，可以将内置 HKLM PSDrive 与 **Set-Location** 一起使用：

```
Set-Location -Path hklm:\SOFTWARE\Microsoft\Windows\CurrentVersion
```

然后，可以使用“.”符号表示当前位置以列出属性，而不指定完整路径：

```
PS> Get-ItemProperty -Path .
...
DevicePath           :C:\WINDOWS\inf
MediaPathUnexpanded  :C:\WINDOWS\Media
ProgramFilesDir      :C:\Program Files
...
```

路径扩展名与它在文件系统中的使用方法相同，因此可以使用 **Get-ItemProperty -Path ..\Help** 从此位置获取 **HKLM:\SOFTWARE\Microsoft\Windows\Help** 的 **ItemProperty** 列表。

## 获取单个注册表条目

若要检索注册表项中的特定条目，请使用下列几种可能的方法之一。以下示例在 **HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion** 中查找 **DevicePath** 的值。

在使用 **Get-ItemProperty** 时，使用 **Path** 参数指定项的名称，并使用 **Name** 参数指定 **DevicePath** 条目的名称。

```
PS> Get-ItemProperty -Path HKLM:\Software\Microsoft\Windows\CurrentVersion -Name
DevicePath

PSPath           :Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\
Microsoft\Windows\CurrentVersion
PSParentPath     :Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\Software\
Microsoft\Windows
PSChildName      :CurrentVersion
PSDrive          :HKLM
PSProvider       :Microsoft.PowerShell.Core\Registry
DevicePath       :C:\WINDOWS\inf
```

此命令返回标准的 Windows PowerShell 属性以及 **DevicePath** 属性。

 请注意：

虽然 **Get-ItemProperty** 具有 **Filter**、**Include** 和 **Exclude** 参数，但是无法使用它们按属性名称进行筛选。这些参数引用注册表项（它们是项路径），而不是引用注册表条目（它们是项属性）。

另一种方法是使用 **Reg.exe** 命令行工具。若要获取 **reg.exe** 的帮助，请在命令提示符下键入 **reg.exe /?**。若要查找 **DevicePath** 条目，请使用 **reg.exe**，如下命令所示：

```
PS> reg query HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion /v DevicePath

! REG.EXE VERSION 3.0

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
DevicePath REG_EXPAND_SZ %SystemRoot%\inf
```

也可以使用 **WshShell COM** 对象查找一些注册表条目，尽管此方法不适用于大二进制数据或包括诸如“\”的字符的注册表条目名称。使用 \ 分隔符将属性名称追加到项路径：

```
PS> (New-Object -ComObject
```

```
WScript.Shell).RegRead("HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\DevicePath"
)
%SystemRoot%\inf
```

## 创建新的注册表条目

若要将名为“PowerShellPath”的新条目添加到 **CurrentVersion** 项，请将 **New-ItemProperty** 与项的路径、条目名称以及条目的值一起使用。在以下示例中，我们将采用 Windows PowerShell 变量 **\$PSHome** 的值，该变量存储 Windows PowerShell 安装目录的路径。

使用以下命令可以将新条目添加到项，而且该命令还返回有关新条目的信息：

```
PS> New-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name
PowerShellPath -PropertyType String -Value $PSHome

PSPath      :Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion
PSParentPath :Microsoft.PowerShell.Core\Registry::HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
PSChildName  :CurrentVersion
PSDrive      :HKLM
PSProvider   :Microsoft.PowerShell.Core\Registry
PowerShellPath :C:\Program Files\Windows PowerShell\v1.0
```

**PropertyType** 必须是下表中 **Microsoft.Win32.RegistryValueKind** 枚举成员的名称：

PropertyType 值	含义
Binary	二进制数据
DWord	一个有效的 UInt32 数字
ExpandString	一个可以包含动态扩展的环境变量的字符串
MultiString	多行字符串
String	任何字符串值
QWord	8 字节二进制数据

 请注意：

通过指定 **Path** 参数的值数组，可以将注册表条目添加到多个位置：

```
New-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion,
HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name PowerShellPath -PropertyType
String -Value $PSHome
```

通过将 **Force** 参数添加到任何 **New-ItemProperty** 命令，也可以覆盖预先存在的注册表条目值。

## 重命名注册表条目

若要将 **PowerShellPath** 条目重命名为“PSHome”，请使用 **Rename-ItemProperty**：

```
Rename-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name
PowerShellPath -NewName PSHome
```

若要显示重命名后的值，请将 **PassThru** 参数添加到该命令。

```
Rename-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name PowerShellPath -NewName PSHome -passthru
```

### 删除注册表条目

若要同时删除 PSHome 和 PowerShellPath 注册表条目，请使用 **Remove-ItemProperty**:

```
Remove-ItemProperty -Path HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name PSHome
Remove-ItemProperty -Path HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion -Name PowerShellPath
```

## 附录 1 - 兼容性别名

Windows PowerShell 具有几个转换别名，允许 Unix 和 Cmd 用户在 Windows PowerShell 中使用熟悉的命令名称。下表列出了最常用的别名以及别名背后的 Windows PowerShell 命令和标准的 Windows PowerShell 别名（如果存在）。

 请注意：

使用 **Get-Alias** 可以从 PowerShell 中查找任何别名所指向的 PowerShell 命令。例如：

```
PS> Get-Alias cls
```

CommandType	Name	Definition
-----	----	-----
Alias	cls	Clear-Host

CMD 命令	Unix 命令	PS 命令	PS 别名
dir	ls	Get-ChildItem	gci
cls	clear	<b>Clear-Host</b> （函数）	不可用
del、erase、rmdir	rm	Remove-Item	ri
copy	cp	Copy-Item	ci
move	mv	Move-Item	mi
rename	mv	Rename-Item	rni
type	cat	Get-Content	gc
cd	cd	Set-Location	sl
md	mkdir	New-Item	ni
不可用	pushd	Push-Location	不可用
不可用	popd	Pop-Location	不可用

## 附录 2 - 创建自定义的 PowerShell 快捷方式

---

以下过程将指导您完成为已自定义若干便利选项的 PowerShell 创建快捷方式的步骤。

1. 创建指向 `powershell.exe` 的快捷方式。
2. 右键单击该快捷方式，然后选择“属性”。
3. 单击“选项”选项卡。
4. 若要启用基于鼠标的选择和复制，请在“编辑选项”下选中“快速编辑”复选框。然后通过拖动鼠标左键在 PowerShell 控制台窗口中选择文本，再通过按 **Enter** 键或单击鼠标右键将文本复制到剪贴板。
5. 在“编辑选项”中，选中“插入模式”复选框。然后可以在控制台窗口中单击鼠标右键，以便自动地从剪贴板粘贴文本。
6. 在“命令记录”的“缓冲区大小”数字显示框中，键入或选择一个介于 1 和 999 之间的数字。此操作用于设置将在控制台缓冲区中保留的键入命令数。
7. 在“命令记录”中，选中“丢弃旧的副本”复选框以便从控制台缓冲区中除去重复的命令。
8. 单击“布局”选项卡。
9. 在“屏幕缓冲区”分组框的“高度”滚动框中，键入一个介于 1 和 9999 之间的数字。高度表示已缓冲的输出行数。这是在控制台窗口中滚动时为查看而保留的最大行数。如果此数字小于“窗口大小”框中显示的高度，则“窗口大小”中的高度将自动减至该值。
10. 在“窗口大小”分组框中，为宽度键入一个介于 1 和 9999 之间的数字。这表示跨控制台窗口显示的字符数。默认宽度是 80，PowerShell 的输出格式是根据此宽度设计的。
11. 如果希望在打开控制台时将其放置在桌面上的特定位置，请在“窗口位置”分组框中，清除“由系统定位窗口”复选框，然后在“窗口位置”中，更改“左”和“上”中的值。
12. 完成后单击“确定”按钮。