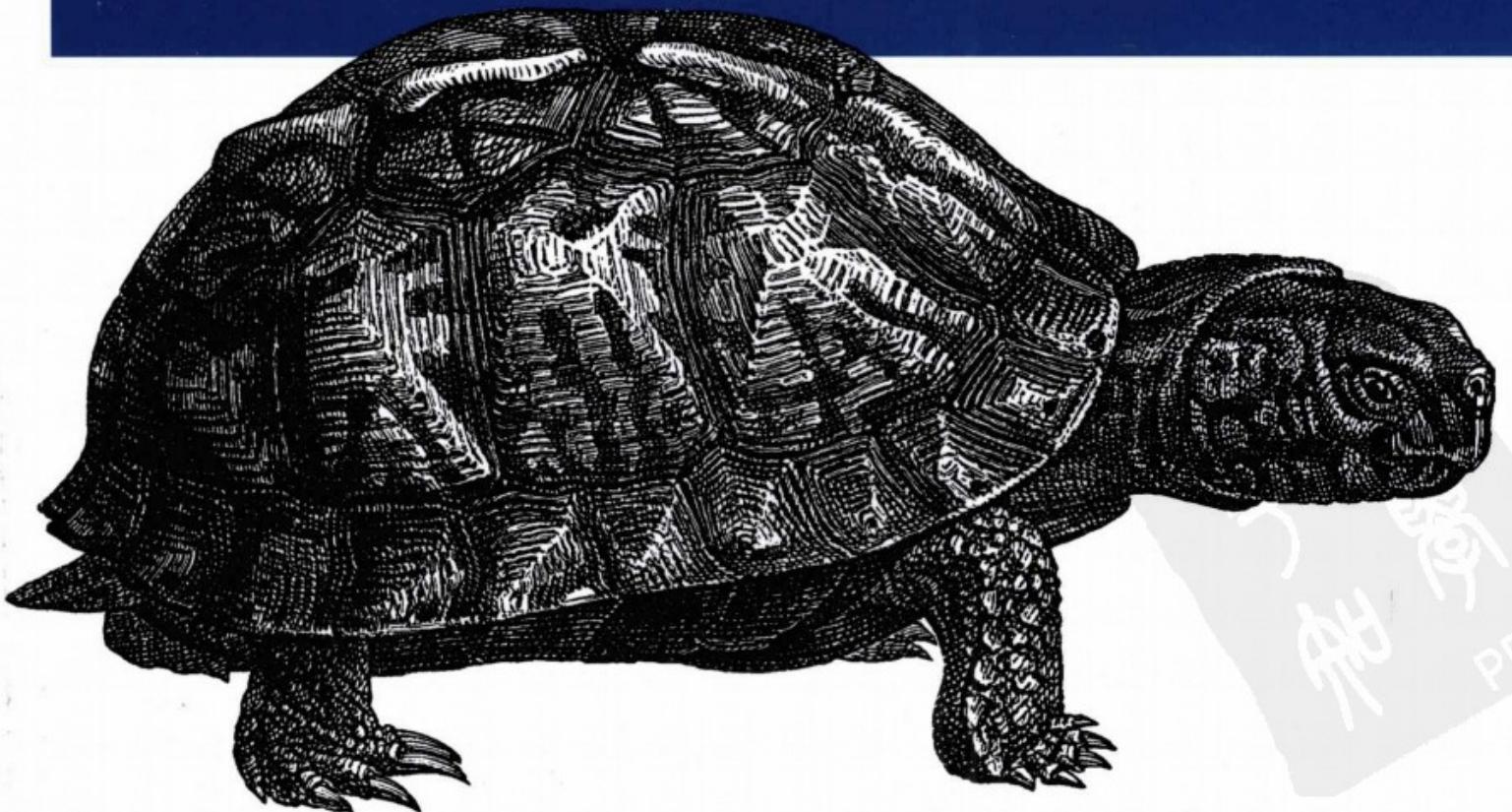


Windows PowerShell Cookbook

Windows PowerShell 应用手册



O'REILLY®

机械工业出版社
China Machine Press



Lee Holmes 著
Dean Tsaltas 序
赵松德 王英群 译

Windows PowerShell应用手册



Windows PowerShell团队开发人员Lee Holmes在本书中提供了上百个测试过的脚本，让你可以使用这个新的工具。本书提供了150个解决方案，并结合一个简洁的基于Windows PowerShell脚本语言和环境的任务进行介绍。当你遇到棘手的问题或需要一个快速解决方案时，可以快速地查找相关的指导。

Lee Holmes为你提供了实用的工具和忠告，使你提高工作效率。你将能够解决所有问题，从自动执行日常任务、使用文件、事件日志、其他形式的结构化数据，到管理用户和复杂的Windows网络资源。每个解决方案都包含一些代码，并讲解了代码是如何工作的，以便你可以将该解决方案应用于类似的任务中。

本书包括：

- 管道、变量、对象、循环和流控制、字符串和非结构化的文本、计算和数学计算。
- 简单文件、结构化文件、支持Internet的脚本、列表、数组、哈希表、错误管理、环境安全意识、脚本签名。
- 文件和目录、注册表操作、比较数据、事件日志、进程cmdlet、服务cmdlet、活动目录、企业计算机管理。

附录包括PowerShell语言快速参考和对管理员来说有用的.NET、WMI和COM对象的介绍。那些管理Microsoft Exchange 2007和系统中心Operations Manager（以前的MOM）的管理员会发现有专门介绍相关内容的章节。

“Lee是PowerShell的核心开发人员，也是PowerShell社区中主要的问题解答者。本书提供解决实际问题的策略和方法，值得每一位PowerShell用户收藏。”

——Jeffrey Snover，Windows PowerShell架构师

“本书填补了PowerShell书库中最大的空缺。虽然有许多书（包括我的书）谈论到PowerShell，但是本书是第一本真正把重点放在应用PowerShell并针对现实问题提供一个解决方案的书。本书值得每个人收藏。”

——Bruce Payette，PowerShell语言设计者和畅销书《Windows PowerShell in Action》作者

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

华章网站：<http://www.hzbook.com>

网上购书：www.china-pub.com



www.oreilly.com

O'Reilly Media, Inc.授权机械工业出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行

This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-111-25362-4



9 787111 253624

定价：89.00元

赵松德 王英群 译

Windows PowerShell 应用手册

Lee Holmes 著

赵松德 王英群 译

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

O'Reilly Media, Inc. 授权机械工业出版社出版

机械工业出版社

赵松德 王英群 译

机械工业出版社

赵松德 王英群 译

图书在版编目 (CIP) 数据

Windows PowerShell 应用手册 / (美) 霍麦斯 (Holmes, L.) 著; 赵松德等译.
—北京: 机械工业出版社, 2009.4

书名原文: Windows PowerShell Cookbook

ISBN 978-7-111-25362-4

I. W… II. ①霍…②赵… III. 窗口软件, Windows — 手册 IV. TP316.7-62

中国版本图书馆 CIP 数据核字 (2008) 第 161265 号

北京市版权局著作权合同登记

图字: 01-2008-1733 号

©2008 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Machine Press, 2009. Authorized translation of the English edition, 2008 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2008。

简体中文版由机械工业出版社出版 2009。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得 ~~到 O'Reilly Media, Inc. 的授权~~ — O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

本书法律顾问

北京市展达律师事务所

书 名 / Windows PowerShell 应用手册

书 号 / ISBN 978-7-111-25362-4

责任编辑 / 周茂辉

封面设计 / Karen Montgomery, 张健

出版发行 / 机械工业出版社

地 址 / 北京市西城区百万庄大街 22 号 (邮编 100037)

印 刷 / 北京京师印务有限公司印刷

开 本 / 178 毫米 × 233 毫米 16 开本 35.5 印张

版 次 / 2009 年 4 月第 1 版 2009 年 4 月第 1 次印刷

定 价 / 89.00 元 (册)

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线: (010)68326294

O'Reilly Media, Inc. 介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权机械工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在 UNIX、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的 *The Whole Internet User's Guide & Catalog* (被纽约公共图书馆评为 20 世纪最重要的 50 本书之一) 到 GNN (最早的 Internet 门户和商业网站)，再到 WebSite (第一个桌面 PC 的 Web 服务器软件)，O'Reilly Media, Inc. 一直处于 Internet 发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc. 是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版阵容。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc. 还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在着手编写著作，O'Reilly Media, Inc. 依靠他们及时地推出图书。因为 O'Reilly Media, Inc. 紧密地与计算机业界相联系，所以 O'Reilly Media, Inc. 知道市场上真正需要什么样的图书。

译者序

PowerShell 自 2006 年 4 月 25 日正式发布以来，时隔两年多，由 PowerShell 开发团队重要核心成员 Lee Holmes 编写的《Windows PowerShell Cookbook》的中文版终于要和广大 PowerShell 爱好者们见面了。Lee 的书填补了 PowerShell 书库中的最大的空缺。本书把重点放在应用 PowerShell，并针对每个问题提供相应的解决方案。在本书中提供了上百个测试过的脚本和 150 个解决方案，让你可以迅速掌握与应用 PowerShell，值得每个人收藏。这是 PowerShell 工具本身的魅力所在，也是本书作者 Lee Holmes 孜孜不倦的创作激情和灵感所结出的硕果。

本书将每个 PowerShell 功能分解为响应“问题”及其“解决方案”，大部分还配以讨论，详尽地讲解问题本身及其解决方案甚至是代码本身是如何工作的。本书覆盖了所有 PowerShell 应用，从日常任务自动执行、使用文件、事件日志、其他形式的结构化数据、管理用户到复杂的 Windows 网络资源等。

本书还介绍了使用其他工具作为 PowerShell 可行性的方法，如：使用 .NET、WMI 和 COM 对象。那些管理 Microsoft Exchange 2007 和系统中心 Operations Manager（以前 MOM）的管理员会发现也有专门介绍相关内容的章节。

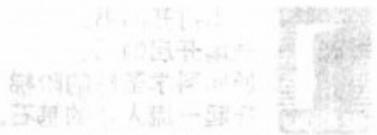
我们在翻译本书的过程中力求忠于原著，真实反映作者思想。对于本书中出现的大量专业术语尽量遵循标准的译法。

全书主要由赵松德翻译，王英群参与了翻译工作。由于水平有限，书中出现错误与不妥之处在所难免，恳请读者批评指正。

译者

2008 年 6 月

作者简介



Lee Holmes 是 Microsoft Windows PowerShell 小组的开发人员，自从 PowerShell 发布 Beta 版本之后，一直是相关的信息的权威发布者。他拥有大量的使用 Windows PowerShell 的经验并将这些经验在讨论中整理成“如何”和“为什么”。Lee 通过新闻、邮件群发和博客参与 PowerShell 和管理社区的讨论。面对各种管理员和用户遇到的问题，他表现出了敏锐的洞察力。

封面介绍

本书封面上的动物是一种盒龟 (box turtle)。这种龟主要产于北美洲，主要分布在美
国和墨西哥的北部地区。

这种雄龟平均长度大约是6英寸长并且有红色眼睛，雌龟比雄龟短一些并有黄色的眼
睛。盒龟在未成年期间什么都吃，但是长大后主要食草。如果盒龟遇到危险，它的
头、尾巴和肢体会缩回壳里。它们通常会在以出生地为圆心，半径一公里的范围内活
动，很少会离开。在交配期间，雄龟有时通过互相推挤来赢取雌龟的注意。在交媾
时，雄龟会身体向后翻倒，而无法自我在翻过身来，有可能会被饿死。

虽然盒龟能活到超过100年，但是由于大陆的开发与修路，它们的栖息地正受到严重
威胁。在漫长的冬眠期间，盒龟需要宽松的、潮湿土壤来下蛋和挖洞。盒龟也非常易
受疾病的影响。专家强烈建议不要将盒龟从他们的栖息地迁出，因为那样不仅将打乱
它们繁殖的机会，还会使盒龟变的压力大并且迅速灭亡。



03 ... 面貌巨变 Performance-based 安全策略
05 ... 从命令行到 PowerShell，从批处理到脚本
06 ... 从命令行到 PowerShell，从批处理到脚本
07 ... 从命令行到 PowerShell，从批处理到脚本
08 ... 从命令行到 PowerShell，从批处理到脚本
09 ... 从命令行到 PowerShell，从批处理到脚本
10 ... 从命令行到 PowerShell，从批处理到脚本
11 ... 从命令行到 PowerShell，从批处理到脚本
12 ... 从命令行到 PowerShell，从批处理到脚本
13 ... 从命令行到 PowerShell，从批处理到脚本
14 ... 从命令行到 PowerShell，从批处理到脚本
15 ... 从命令行到 PowerShell，从批处理到脚本
16 ... 从命令行到 PowerShell，从批处理到脚本
17 ... 从命令行到 PowerShell，从批处理到脚本
18 ... 从命令行到 PowerShell，从批处理到脚本
19 ... 从命令行到 PowerShell，从批处理到脚本
20 ... 从命令行到 PowerShell，从批处理到脚本
21 ... 从命令行到 PowerShell，从批处理到脚本
22 ... 从命令行到 PowerShell，从批处理到脚本
23 ... 从命令行到 PowerShell，从批处理到脚本
24 ... 从命令行到 PowerShell，从批处理到脚本
25 ... 从命令行到 PowerShell，从批处理到脚本

目录

序	1
前言	5
第一部分 教程	
教程 Windows PowerShell 使用指南	13
简介	13
一个交互式的 shell	14
结构化命令	16
与对象深层次的集成	17
作为一流的系统管理员	18
可组合命令	18
防止误操作的技术	19
常用的发现命令	20
无处不在的脚本	21
特殊开发	21
技术的桥梁	22
通过提供程序导航命名空间	24
更多的内容	25

第二部分 基础知识

第1章 Windows PowerShell 交互界面	29
1.0 绪论	29
1.1 运行程序、脚本和已有的工具	29
1.2 运行 PowerShell 命令	31
1.3 自定义 shell、配置文件与提示符	32
1.4 查找实现指定任务的命令	35
1.5 获得命令帮助	36
1.6 编程：搜索帮助	38
1.7 在 PowerShell 之外调用 PowerShell 脚本	39
1.8 编程：保持批处理文件修改的环境变量	40
1.9 获取系统日期与时间	42
1.10 检查最后运行命令的状态	42
1.11 度量命令执行时间	44
1.12 定制 shell 来提升工作效率	45
1.13 编程：学习常用命令的别名	46
1.14 使用与管理控制台历史	48
1.15 将命令的输出保存到文件中	49
1.16 向文件的结尾处加入信息	51
1.17 记录你的会话全文	51
1.18 将某一项的属性显示成列表	52
1.19 将某一项的属性显示成表格	53
1.20 管理命令的错误输出	54
1.21 配置调试、校验和处理输出	56
1.22 通过附加单元扩展 PowerShell	57
1.23 使用控制台文件加载保存 snapin	58
第2章 管道	60
2.0 简介	60
2.1 过滤列表项或命令输出项	61
2.2 编程：简化多数 Where-Object 过滤	62
2.3 编程：交互式过滤对象	63

2.4	处理列表或命令输出的每一项	65
2.5	自动化数据密集型任务	67
第3章 变量与对象		72
3.0	简介	72
3.1	在变量中存储信息	72
3.2	访问环境变量	74
3.3	控制访问和变量的范围与其他项	76
3.4	使用 .NET 对象	78
3.5	创建一个 .NET 对象的实例	82
3.6	编程：创建对象的实例	84
3.7	快速输入较长的类名	85
3.8	使用 COM 对象	87
3.9	了解类型和对象	87
3.10	获得类和对象详细文档	89
3.11	向对象添加自定义的方法和属性	91
3.12	向类添加自定义的方法和属性	93
第4章 循环与流程控制		97
4.0	简介	97
4.1	通过比较和逻辑操作做出决定	97
4.2	使用条件语句控制脚本流程	99
4.3	使用 switch 管理条件语句	100
4.4	使用循环	101
4.5	添加暂停或延迟	103
第5章 字符串与非结构化文本		105
5.0	简介	105
5.1	创建字符串	105
5.2	创建一个多行或格式化的字符串	107
5.3	在字符串中放置特殊字符	108
5.4	向字符串中插入动态信息	109
5.5	禁止字符串包含动态信息	110

5.6	在字符串中插入格式化的信息	111
5.7	根据文本或模式在字符串中查找	113
5.8	替换字符串中的文本	115
5.9	字符串大、小写转换	116
5.10	去掉字符串中的空格	117
5.11	格式化日期的输出	118
5.12	转换文本流为对象	120
5.13	生成大的报告和文本流	124

第 6 章 计算和数学计算 127

6.0	简介	127
6.1	执行简单的算法	127
6.2	执行复杂的算法	129
6.3	度量一个列表的统计属性	131
6.4	使用二进制数	133
6.5	简化管理用的常量	136
6.6	在不同的进制间转换数字	137

第三部分 常见任务

第 7 章 简单文件 143

7.0	简介	143
7.1	获取文件的内容	143
7.2	搜索文件中的文本	145
7.3	分析和管理基于文本的日志	146
7.4	分析和管理二进制文件	149
7.5	创建临时文件	151
7.6	搜索和替换文件中的文本	152

第 8 章 结构化文件 156

8.0	简介	156
8.1	访问 XML 文件中的信息	156
8.2	对 XML 文件执行 XPath 查询	159

8.3	修改 XML 文件中的信息	160
8.4	轻松导入和导出结构化数据	162
8.5	将一个命令的输出存储到 CSV 文件	164
8.6	从 CSV 文件中导入结构化的数据	165
8.7	使用 Excel 管理命令输出	166

第 9 章 支持 Internet 的脚本 169

9.0	简介	169
9.1	从 Internet 下载一个文件	169
9.2	从 Internet 下载一个 Web 页面	170
9.3	编程：获得页面中的超级链接	173
9.4	编程：调用 Web 服务	175
9.5	将命令的输出生成一个 Web 页面	178
9.6	编程：发送电子邮件	179
9.7	编程：与 Internet 协议交互	180

第 10 章 代码复用 185

10.0	简介	185
10.1	编写一个脚本	185
10.2	编写一个函数	188
10.3	编写一个脚本块	189
10.4	从脚本、函数、脚本块返回数据	191
10.5	将常用的函数放到库文件中	193
10.6	脚本、函数或脚本块的访问参数	194
10.7	访问管道输入	196
10.8	用命令关键字（Cmdlet Keywords）编写面向管道的脚本	198
10.9	编写一个面向管道的函数	202

第 11 章 列表、数组和哈希表 204

11.0	简介	204
11.1	创建数组或项的列表	204
11.2	创建交错或多维数组	206
11.3	访问数组中的元素	207
11.4	访问数组的每个元素	208

11.5	对数组或列表中的项进行排序	209
11.6	确定数组是否包含某项	210
11.7	合并数组	210
11.8	从数组中查找匹配一个值的项	212
11.9	从数组中移出元素	212
11.10	从数组中查找大于或小于一个值的项	213
11.11	使用 ArrayList 类完成高级的数组任务	214
11.12	创建哈希表或关联数组	216
11.13	根据键或值对哈希表排序	217
11.14	将哈希表转换为数组	218
第 12 章 用户交互		219
12.0	简介	219
12.1	读取用户输入一行	219
12.2	读取用户输入的按键	220
12.3	编程：向用户显示一个菜单	221
12.4	给用户显示输出和消息	223
12.5	为长时间运行的任务提供进度更新	226
12.6	编写支持区域性的脚本	227
12.7	编程：采用交替的区域性设置调用脚本块	230
12.8	主机的用户界面的访问功能	231
12.9	编程：向你的脚本中添加一个图形用户界面	233
12.10	将命令行参数传递给你的脚本	234
第 13 章 跟踪和错误管理		236
13.0	简介	236
13.1	查看由某一命令生成的错误	236
13.2	处理警告、错误和终止错误	238
13.3	输出警告、错误和终止错误	240
13.4	调试脚本	241
13.5	收集脚本或命令的详细的跟踪信息	244
13.6	编程：分析脚本的性能的配置文件	245
13.7	将命令行参数传递给你的脚本	246
第 14 章 掌握环境		250
14.0	简介	250
14.1	查看和修改环境变量	250

14.2	关于你的命令调用的访问信息	252
14.3	编程：研究请求信息变量	254
14.4	找到脚本的名称	256
14.5	找到你的脚本的位置	257
14.6	查找常见的系统路径的位置	258
14.7	编程：搜索 Windows 开始菜单	261
14.8	获取当前位置	262
14.9	安全地生成程序文件路径	263
14.10	与 PowerShell 的全局环境进行交互	264

第 15 章 Windows PowerShell 的扩展 266

15.0	简介	266
15.1	访问 WMI 数据	266
15.2	编程：确定可用到 WMI 筛选器的属性	268
15.3	编程：搜索 WMI 类	269
15.4	使用 .NET 来执行高级的 WMI 任务	272
15.5	将一个 VBScript WMI 脚本转换为 PowerShell	274
15.6	使用 COM 脚本接口自动化程序	277
15.7	编程：查询 SQL 数据源	278
15.8	访问 Windows 性能计数器	280
15.9	编程：调用 Windows 系统 API	281
15.10	编程：添加 C# 代码到 PowerShell 脚本中	284
15.11	访问 .NET SDK 库	287
15.12	创建你自己的 PowerShell Cmdlet	289
15.13	添加 PowerShell 脚本到你自己的程序	293

第 16 章 安全和脚本签名 296

16.0	简介	296
16.1	通过执行策略启用脚本	297
16.2	PowerShell 脚本或格式文件签名	299
16.3	编程：创建一个自签名的证书	301
16.4	管理企业中的 PowerShell 安全性	302
16.5	验证 PowerShell 脚本的数字签名	305
16.6	安全地处理敏感信息	306

16.7	安全地要求用户名和密码	308
16.8	编程：作为另一个用户启动一个进程	309
16.9	在磁盘上安全地存储凭据	311
16.10	访问用户和计算机证书	312
16.11	编程：搜索证书存储区	314

第四部分 管理员任务

第 17 章 文件和目录 319

17.0	简介	319
17.1	查找一个特定日期之前修改的所有文件	320
17.2	清除或移动文件	321
17.3	管理与改变文件属性	322
17.4	获取目录中的文件列表	323
17.5	使用匹配模式查找文件	324
17.6	管理包含特殊字符的文件	328
17.7	获取磁盘使用情况信息	329
17.8	确定当前的位置	330
17.9	监视文件内容变更	331
17.10	编程：获取一个文件的 MD5 或 SHA1 哈希值	332
17.11	建立目录	334
17.12	删除文件或目录	335
17.13	文件或目录重命名	335
17.14	移动文件或目录	337
17.15	获取文件或目录的访问控制列表	338
17.16	设置文件或目录的访问控制列表	339
17.17	将扩展的文件属性添加到文件	341
17.18	编程：创建文件系统硬链接	343
17.19	编程：创建 ZIP 文档	344

第 18 章 Windows 注册表 347

18.0	简介	347
18.1	注册表导航	347

18.2	查看一个注册表项	348
18.3	修改或删除一个注册表键值	349
18.4	创建一个注册表键值	350
18.5	删除注册表项	351
18.6	将站点添加到 IE 浏览器的安全域中	352
18.7	修改 IE 浏览器设置	354
18.8	编程：搜索 Windows 注册表	356
18.9	获取某个注册表项的访问控制列表	357
18.10	设置一个注册表项的访问控制列表	358
18.11	使用远程计算机的注册表	359
18.12	编程：从远程计算机获取注册表项	361
18.13	编程：获取远程注册表项的属性	362
18.14	编程：设置远程注册表项的属性	364
18.15	程序的注册表设置	365
第 19 章 数据比较		369
19.0	简介	369
19.1	比较两个命令的输出	369
19.2	确定两个文件之间的差异	370
19.3	验证文件集的完整性	371
第 20 章 事件日志		373
20.0	简介	373
20.1	列出所有事件日志	373
20.2	从事件日志中获取最新项	374
20.3	使用特定的文本查找事件日志项	375
20.4	检索一个特定的事件日志项	376
20.5	根据频率查找事件日志记录	378
20.6	备份事件日志	380
20.7	创建或删除事件日志	380
20.8	写入事件日志	381
20.9	访问远程计算机的事件日志	382

第21章 进程	384
21.0 简介	384
21.1 列出当前运行的进程	385
21.2 启动一个进程	386
21.3 停止一个进程	387
21.4 编程：调用远程计算机上的一个 PowerShell 表达式	389
第22章 系统服务	391
22.0 简介	391
22.1 列出所有运行的服务	391
22.2 管理一个正在运行的服务	393
22.3 访问在远程计算机上的服务	394
第23章 活动目录	396
23.0 简介	396
23.1 测试本地安装的活动目录脚本	396
23.2 创建组织单元	399
23.3 获取组织单元的属性	400
23.4 修改组织单元的属性	400
23.5 获取一个活动目录容器的子集	401
23.6 创建用户账户	401
23.7 编程：批量导入活动目录用户	402
23.8 搜索用户账户	404
23.9 获取并列出用户账户的属性	405
23.10 修改用户账户的属性	406
23.11 创建一个安全组或分布组	406
23.12 搜索一个安全组或分布组	408
23.13 获取一个组的属性	409
23.14 查找用户组的所有者	409
23.15 修改安全组或分布组的属性	410
23.16 将用户添加到安全组或分布组	410
23.17 从安全组或分布组中删除用户	411
23.18 列出用户的组成员身份	412

23.19	列出一组的所有成员	412
23.20	列出一个组织单元的所有用户	413
23.21	搜索计算机账户	414
23.22	获取并列出一台计算机账户的属性	415
第 24 章 企业级计算机管理		416
24.0	简介	416
24.1	编程: 列出用户登录或注销的脚本	416
24.2	编程: 列出计算机启动或关机脚本	418
24.3	启用或禁用 Windows 防火墙	419
24.4	打开或关闭 Windows 防火墙中的端口	420
24.5	“编程: 列出所有已安装的软件”	421
24.6	卸载应用程序	422
24.7	管理计算机上的计划任务	423
24.8	检索打印机信息	424
24.9	检索打印机队列统计信息	425
24.10	管理打印机和打印队列	427
24.11	确定是否安装了补丁程序	428
24.12	编程: 汇总系统信息	430
24.13	续订动态主机配置协议租约	431
24.14	分配一个静态 IP 地址	433
24.15	列出计算机的所有 IP 地址	434
24.16	列出网络适配器的属性	435
第 25 章 管理 Exchange 2007 通信服务器		437
25.0	简介	437
25.1	尝试使用 Exchange 管理 shell 程序	437
25.2	自动执行向导任务	438
25.3	管理 Exchange 用户	439
25.4	管理邮箱	441
25.5	管理分发组	442
25.6	管理传输规则	443
25.7	管理 Outlook Web Access	444

第 26 章 管理操作管理器 2007 服务器	445
26.0 简介	445
26.1 体验命令 shell 程序	445
26.2 管理操作管理器代理	446
26.3 计划维护窗口	447
26.4 获取、安装和卸载管理包	448
26.5 启用或禁用规则	449
26.6 列出并启动任务	450
26.7 管理警报	450

第五部分 参考资源

附录 A PowerShell 语言和环境	455
附录 B 正则表达式参考	502
附录 C PowerShell 自动变量	510
附录 D 标准 PowerShell 动词	513
附录 E 选定的 .NET 类和它们的使用	517
附录 F WMI 参考	525
附录 G 选定的 COM 对象和它们的使用	533
附录 H .NET 字符串格式	536
附录 I .NET 日期格式	540

序

当 Lee 邀请我为他的新书写序时，我很惊讶。我一直认为应该由该领域中受人尊敬且有所成就的人来作序。很显然，这不是根本性问题。我猜这无关紧要，我的意思是，谁会花时间去阅读一本关于脚本的书的序，是不是？

Lee 希望微软脚本小子小组 (Scripting Gays) 中的一个成员为他撰写本书的序。他写这本书的目的是供那些使用脚本来努力工作的管理员参考。Lee 认为让脚本小子小组中的人提供一些关于 Windows 管理脚本的现状、发展以及对 PowerShell 的意见是十分有意义的。

自从 Lee 和我第一次谈论这个问题之后，很多事情都发生了变化。我离开了 Microsoft 脚本小子小组，从事 WMI SDK 的开发。老天好像和“脚本小子”名称开了一个玩笑，目前在后面推动该小组的是一个娇小的、名为 Jean Ross 的女士。现在 Jean 让 Greg 做的工作（如打包和传送脚本）使得 Greg 看起来像个劳工。当我们只看到他耗尽精力撰写的“你好，脚本小子”系列文章时，脚本小子有可能将来会变成脚本女孩！

粘合剂、激活器和 WSH

无论何时当我想起“远景”和“脚本”，通常我会想到 Bob Wells。Bob 非常认真地工作，并多年在 Microsoft 内部和外部推广使用脚本。当我加入该脚本小子小组时，Bob 经常提到术语“粘合剂 (glue)”和“激活器 (enabler)”。我花了一些时间来理解为什么他经常提到这些术语，以及为什么正确地理解激活器是如此重要。我现在理解是因为清楚地定义这两个概念，就建立了一个简单、有用的框架，用来理解管理脚本。脚本语言中的“粘合剂”就是 *foreach*、*if* 和 *var* 这些本身。

粘合剂是用于把一些小的任务编排或粘附在一起，完成一项较大的任务。激活器（我们目前还没有找到更合适的术语来形容它）实际是完成每个子任务的指令或手段。

此表为我们列出了作为 Windows 脚本提供的粘合剂及激活器，有的可以应用到很多年后。

粘合剂	激活器
Cmd.exe 批处理语言	命令行工具 (OS、ResKit,) 支持工具
WSH	命令行工具 (OS、ResKit,) 支持工具 自动化 COM 对象 (WMI、ADSI)
Windows PowerShell	命令行工具 (OS、ResKit、支持工具) 自动化 COM 对象 (WMI、ADSI)
	.NET Framework 类库

请注意每个新的环境如何让你使用以前环境的激活器。这一点很重要，因为它使你可以继续使用已有的知识。客观地说，WSH脚本的功能比批处理脚本更强大，因为它提供更多的可访问的对象。它可以使更多的任务自动进行，因此你能访问 COM 对象自然暴露的其他功能。退一步说，你可能认为即使如果您只打算使用命令行工具作为激活器，相对于批处理来说，WSH是一个更好的选择，因为它提供了一些很有用的粘合剂功能；完善的激活器使更多的事情可能，而提升粘合性，有时使操作更加方便。

WSH脚本提供了一个相当舒适的环境。WMI和ADSI COM类库为全世界的系统管理员带来了无数的痛苦和欢欣。但让人始终烦恼的是你不能正确地使用WSH，或要求您在凌晨2点时从一些陌生的站点下载某些工具。当你真地无法决定在你的服务器上安装什么工具时，如果只有VBScript在它的激活器中包含 Win32 API 作为开发人员，只要你喜欢，你可以做任何事情。

对于开发人员来说，.NET Framework 类库 (FCL) 是新的 Win 32 API。因此，我们真正需要的是一个脚本环境，包含 FCL 作为激活器。这正是 Windows PowerShell 要做的事。事实上，Windows PowerShell 在与该库相同的环境中运行并与其无缝连接。我阅读了大量有关的 Windows PowerShell 管道对象的功能。这些功能非常酷，主要表现在粘合性方面，当然，使用 FCL 自然是提前条件。但增加 Jeffrey 等人创建的 FCL 作为激活器使得此脚本比 WSH 功能更强大。你需要正确的方法来解决如何深入了解 FCL 的问题。这是令人鼓舞的，当你开始在熟悉的环境中投资和开发专门的技术时，你能使用开发人员当前有的或可预知将来有的所有激活器，这应该也是令人安慰的。如果你消耗时间学习 Windows PowerShell，你将得到来自微软 .NET Framework 的长久支持。

Windows PowerShell 遵循传统的 WSH，并在其粘合性方面有所提高。在 WSH 中使用 COM对象的真正痛苦之一是很难找出可用的属性和方法。除非你出钱购买一个智能编辑器，否则你要花费大量的时间在编写脚本和查询文档间切换上。使用 Windows

PowerShell 中的对象时不是这样的。你可以在 Windows PowerShell 提示符下键入下面内容：

```
$objShell = New-Object -com Shell.Application  
$objShell | Get-Member
```

关于 Lee

希望我的介绍已经使你深信 Windows PowerShell 是一个很好的工具，值得你花时间学习。现在，谈一下为什么我认为你应该购买和阅读本书呢？

首先，我应该告诉你 Windows PowerShell 开发团队是由一群不固定的人组成的。这些人都很优秀。Jeffrey Snover 那里的教师是如此喜欢并且相信他们的技术，很难阻止这些教师教你！在这些人中，Lee 是尤为突出的。你曾经听过 Exchange 服务器发生问题时发出的声音吗？当 Lee 来工作并在我们的内部 Windows PowerShell 邮件列表上开始回答问题时，我们都很佩服他。Lee 具有收集各方面知识的能力，并很好地利用 Windows PowerShell 来解决现实世界中出现的问题。通过捕获和共享本书中的某些知识，他和 O'Reilly 已为我们完成了一种很好的服务。

Windows 系统管理脚本编写者是地球上最酷的人。很荣幸这本书能够帮助你们继续愉快地工作，衷心地希望你们喜欢本书。

—— Dean Tsaltas
Microsoft Scripting Guy Emeritus



而且大體不著迷於從事與其職業有關的知識。他們只是半懷疑地向其老師們求教。

$\sigma_1(x) = \sigma_1(100) \cdot T_{100}(x) + \sigma_1(0) \cdot (1 - T_{100}(x))$

see 干关

本片同时表现了对普通人命运的同情。通过一个普通家庭的遭遇，展示了社会的不平等和对底层人民的漠视。

。農業的經濟學，與社會主義的經濟學，是完全不同的兩回事。

對敵空襲而被擊落者外，並無死傷。人情頗重。報明呈准，特此轉本報，希悉。

© 2016 Debra T. Miller

Winnipeg Seed Growers' Co-operative

前言

在 2002 年的年底，有人在 Slashdot 上发表了一篇消息，传闻微软正在开发有关“下一代 shell 程序”。作为一名 shell 程序的爱好者，这条消息立即引起了我的注意。此 shell 程序真能提供像 UNIX 系统上的命令行功能和效率吗？

因为我六个月前刚刚加入 Microsoft，我抓住机会查出 Slashdot 来源。消息中提到它与 .NET Framework 进行了强大的集成，所以我向内部 C# 邮件列表中发布了一个消息。我收到的回应是该项目名称为“Monad”，然后我获得一套内部原型版本。

原型是一个大术语。在早期阶段，生成主要是一个概念验证。要清屏？没问题！按住回车键，直到你以前的命令和输出滚动出你的视线！然而即使在早期阶段，Monad 很显然是命令行 Shell 的一个革命性的进步。它的优点是不言自明的。Monad 支持在其命令之间传递真正的 .NET 对象。甚至对于更复杂的命令，Monad 不再需要基于文本的解析器。这个新的模型由简单和功能强大的数据操作工具支持，创建强大和易于使用的 shell 程序。

加入了 Monad 开发工作组不久之后我完成了我负责的部分，帮助完成世界上杰出的技术的其余部分。此后，Monad 不断地成长，变成现在称为 Windows PowerShell 的产品。

那么，为什么编写一本有关它的书呢？

许多用户可能是出于对 PowerShell 的好奇而选择 PowerShell。对于他们来说，真正有效的方面才能真正为他们带来好处。例如，你可能碰巧要学习一种新的技术知识，用它来解决了您的问题。例如：如何使用 PowerShell 导航文件系统？如何管理文件和文件夹？如何检索网页等？

本书着重帮助你学习 PowerShell，通过基于任务的解决方案来解决大多数紧迫的问题。你可以阅读本书中的一个小节、一章或整本书，你都会从中获益。

本书适合的读者

本书可帮助您使用PowerShell来完成工作。它包含了数百个特定的、解决实际问题的方案。对于系统管理，你会发现很多示例，说明如何管理文件系统、Windows注册表、事件日志、进程和更多内容。对于企业管理，你会发现本书中有完整的两章来介绍WMI、活动目录（Active Directory）和其他企业主要任务。

对于 Exchange 2007 或操作管理器 2007 (MOM) 的管理员，你会发现每一章都会专门涵盖到获取和基础信息新的产品来完成上层任务。

同时，你还将学习有关 PowerShell 的很多知识，包括它的功能、命令和它的脚本语言，但最重要的是解决问题。

本书的组织方式

本书包括五个主要部分：PowerShell 教程、PowerShell 基础知识、常见任务、管理任务和详细的参考资料。

第一部分：教程

一个 Windows PowerShell 的指导教程（第一部分）在较高的级别上介绍了 PowerShell。这里介绍了 PowerShell 的核心功能，包括：

- 这是一个交互式 shell 程序
 - 一种新的命令模型
 - 基于对象的管道
 - 重点关注系统管理员
 - 学习和发现的一致模型
 - 无处不在的脚本
 - 与关键管理技术的集成

该指导教程可以让你自己熟悉 PowerShell。这有助于帮你创建一个框架，来理解本书其余部分的详细信息和解决方案。

第二部分：基础知识

第1章到第6章介绍了PowerShell的基础知识，是解决本书中很多解决方案的基础。此部分中的解决方案向你介绍PowerShell的交互式shell程序、基本管道和对象概念以及PowerShell脚本语言的许多功能。

第三部分：常见任务

第7章至16章涵盖了在PowerShell中处理更复杂的问题时你最常运行的任务。这包括使用简单和结构化的文件、连接到Internet的脚本、代码重用、用户交互和更多内容。

第四部分：管理员任务

第17章到第26章关注在系统和企业管理中最常用的任务。从第17章到第22章，关注的重点是各个系统，包括：文件系统、注册表、事件日志、进程、服务和更多内容。第23章和第24章的重点是活动目录以及最常见的管理网络或加入系统域的典型任务。

第25章和第26章分别介绍管理Exchange 2007和操作管理器2007(MOM)的相关内容。

第五部分：参考资料

许多书把无用的信息添加到它们的附录，只是为了增加页数。而在本书中，详细的参考资料是学习和使用PowerShell不可缺少的基础和基本资源。它包括：

- PowerShell语言和环境
- 正则表达式语法和主要的PowerShell示例
- PowerShell的自动和默认变量
- PowerShell的标准动词
- 管理员常用的.NET类及其用法
- 管理员常用的WMI类及其用法
- 管理员常用的COM对象及其用法
- .NET字符串格式化语法和主要的PowerShell示例
- .NET DateTime格式化语法和主要的PowerShell示例

使用本书时需要的内容

本书的大多数解决方案只需要安装Windows PowerShell。如果你还没有安装PowerShell，那么可以到 <http://www.microsoft.com/PowerShell> 下载它。此链接提供了在 Windows XP、Windows Server 2003 和 Windows Vista 下的下载说明。对于 Windows Server 2008，PowerShell 是可选组件，它可以像其他可选组件那样通过控制面板进行安装。

“活动目录”中给出的活动目录脚本在应用于企业环境时是最有用的。但第 23.1 节演示了如何安装其他软件（活动目录应用程序模式），这让你可以区别于本地安装来运行这些脚本。

第 26 章和第 27 章要求你能够访问 Exchange 或操作管理器 2007 的环境。如果你没有这些环境，第 25.1 节和第 26.1 节向你演示了如何对 Exchange 和操作管理器使用 Microsoft 虚拟实验室作为一种可行的替代方案。

本书中使用的约定

本书中使用下列排印惯例：

纯文本 (Plain text)

表示菜单标题、菜单项、菜单按钮和键盘加速键。

斜体 (*Italic*)

表示新术语、URL、电子邮件地址、文件名、文件扩展名、路径名、目录和 UNIX 实用程序。

等宽字体 (Constant width)

表示命令、选项、开关、变量、类型、类、名称空间、方法、模块、属性、参数、值、对象、事件、事件句柄、XML 标记、宏、文件内容或命令输出。

等宽粗体 (Constant width bold)

显示用户输入的命令或其他文本。

等宽斜体 (Constant width italic)

表示应该用用户提供的值替换的文本。

注意：此图标表示一个提示、建议或一般备注。

警告：此图标表明警告。

代码示例

获取示例代码

若要获取本书中给出的程序和示例的电子版本，可以访问下面链接：

<http://www.oreilly.com/catalog/9780596528492>

使用代码示例

本书旨在帮助你完成工作。一般而言，你可以在自己的程序和文档中使用本书中的代码。你不必与我们联系以获得许可。除非你要复制代码的重要部分。例如，编写一个程序，其中使用了本书中的几段代码，无须取得许可；销售或发行包含 O'Reilly 图书中示例的 CD-ROM 时，必须取得许可。引用本书及其示例代码来解答问题时，不需要获得许可；若要将本书中的大量示例代码加入你的产品文档，则必须取得许可。

我们感谢参考资料的提供者但不一定注明出处。在注明出处时，通常应包括书名、作者、出版社和 ISBN。

例如：“*Windows PowerShell Cookbook* by Lee Holmes. Copyright 2007 Lee Holmes, 978-0-596-52849-2”。

如果你感到对代码示例的使用不在公平使用或上述许可范围之内，可通过 permissions@oreilly.com 与我们联系。

本书的评论和建议

请通过以下地址把关于本书的评论和问题发送给出版社：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室 (100035)
奥莱利技术咨询（北京）有限公司

本书的 Web 页上列出了勘误表、示例和其他额外信息。可在以下网址查询：

<http://www.oreilly.com/catalog/9780596528492> (英文版)

<http://www.oreilly.com.cn/book.php?bn=978-7-111-25362-4> (中)

如果想就本书的技术问题发表评论或咨询，请发邮件到：

bookquestions@oreilly.com
info@mail.oreilly.com.cn

关于书本、会议、资源中心和 O'Reilly 网络的更多信息，请访问 O'Reilly 的 Web 站点：

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

第一部分

教程

Windows PowerShell 教程

代碼一票

黑木

必勝用物 (Best Win's Tools)

Windows PowerShell

使用指南

简介

Windows PowerShell 在推出时承诺要对现有的系统管理和命令行管理界面进行彻底的改革。从基于对象的管道，到关注系统管理员工作，再到其他涉及管理的技术，PowerShell 极大地提高了系统管理员和用户的工作效率。

我们在开始学习一项新的技术时，由于它提供了不熟悉的功能，我们会感到迷茫，不知道从哪里开始。尤其是首次接触 Windows PowerShell 的用户，因为他们以前没有使用命令行界面的经验。更糟糕的是，他们听说了 PowerShell 神奇的与脚本集成的能力，就会害怕被动地进入他们一直避免进入的这片领域。

幸运的是，他们的恐惧是多余的：PowerShell 会在你使用的同时与你一同成长。让我们看看它能够完成什么工作：

- PowerShell 与标准的 Windows 命令和应用程序兼容，你可以继续使用已经掌握的知识。
- PowerShell 推出了强大的新命令格式。PowerShell 命令（简称 cmdlet）采用动-名词的语法，比现有的命令行格式更容易操作。
- PowerShell 支持对象。与传统的命令行打交道相比，直接与支持丰富结构的对象打交道极大地简化了工作。
- PowerShell 适合管理员使用。即使是使用它提供的高级功能，PowerShell 也会将更多的关注点集中到与用户的交互：体验在一个运行的 PowerShell 应用程序中输入命令。
- PowerShell 支持发现功能。通过使用三个简单的命令，你可以学习并发掘 PowerShell 所提供的所有功能。

- PowerShell 支持多种脚本语言。任何一种可以直接在命令行下工作的脚本语言，都可以通过 PowerShell 进行自动化的任务处理。
- PowerShell 将为许多技术提供桥梁。
- 在 PowerShell 中使用这些技术，如 .NET、COM、WMI、XML、活动目录等，它比以往单独使用它们更加容易。
- PowerShell 简化了数据存储管理。通过自带的提供者模式，你可以使用已经熟悉的、同样的技术来管理文件与文件夹。

接下来我们逐一介绍 PowerShell 的各项功能。

一个交互式的 shell

作为 PowerShell 的核心，PowerShell 首要的是提供交互式界面。在它提供脚本和其他强大功能的同时，关注的是提供一个交互的平台。启动 PowerShell 非常简单，只需要用 *PowerShell.exe* 替代 *cmd.exe*。它为我们提供了丰富的功能，使你可以立即开始使用。

加载 Windows PowerShell 的步骤如下：

点击 Start (开始) → All Programs (所有程序) → Windows PowerShell 1.0 → Windows PowerShell 或者点击 Start (开始) → Run (运行)，然后键入“PowerShell”。

PowerShell 的提示窗口与传统的 Windows XP、Windows Server 2003 的命令行窗口很相似，提示符“PS C:\Documents and Settings\Lee>”表明 PowerShell 已经准备好接收命令，如图 T-1 所示。

加载 PowerShell 提示符后，你可以输入 DOS 风格和 UNIX 风格的命令来操作文件系统，与你以前在 Windows 和 UNIX 提示符下操作一样，交互的界面参见例 T-1。

例 T-1：输入一些标准的 DOS 和 UNIX 风格的文件管理命令，输出与你平常使用其他 Windows Shell 的效果类似

```
PS C:\Documents and Settings\Lee> function Prompt { "PS >" }
PS >pushd .
PS >cd \
PS >dir
Directory: Microsoft.PowerShell.Core\FileSystem::C:\

        Mode                LastWriteTime     Length Name
<---->                <---->           <---->

```

Mode

LastWriteTime

Length Name



图 T-1：Windows PowerShell，准备好接收输入

```
d---- 11/2/2006 4:36 AM $WINDOWS.~BT
d---- 5/8/2007 8:37 PM Blurpark
d---- 11/29/2006 2:47 PM Boot
d---- 11/28/2006 2:10 PM DECCHECK
d---- 10/7/2006 4:30 PM Documents and Settings
d---- 5/21/2007 6:02 PM F&SC-demo
d---- 4/2/2007 7:21 PM Inetpub
d---- 5/20/2007 4:59 PM Program Files
d---- 5/21/2007 7:26 PM temp
d---- 5/21/2007 8:55 PM Windows
-a--- 1/7/2006 10:37 PM 0 autoexec.bat
-ar-s 11/29/2006 1:39 PM 8192 BOOTSECT.BAK
-a--- 1/7/2006 10:37 PM 0 config.sys
-a--- 5/1/2007 8:43 PM 33057 RUU.log
-a--- 4/2/2007 7:46 PM 2487 secedit.INTEG.RAW
```

```
PS >popd
PS >pwd
```

Path

```
----
```

```
C:\Documents and Settings\Lee
```

如例 T-1 所示，你可以使用 pushd、cd、dir、pwd 和 popd 命令来保存当前位置、导航文件系统、列出当前目录并返回初始位置，你可以试一下！

注意：pushd 命令是 PowerShell 命令 Push-Location 的别名。同样地，命令 cd、dir、popd 和 pwd 也都有相应容易记忆的名称。

尽管导航文件系统很有帮助，你也可以运行你知道的、喜爱的工具，如 ipconfig 和 notepad。输入命令的名称，你会看到如例 T-2 显示的结果。

例 T-2：Windows 工具和应用程序，如 ipconfig 在 PowerShell 中运行与在 cmd 命令下运行一样

```
PS >ipconfig
```

```
Windows IP Configuration
```

```
Ethernet adapter Wireless Network Connection 4:
```

```
    Connection-specific DNS Suffix . : hsd1.wa.comcast.net.  
    IP Address . . . . . : 192.168.1.100  
    Subnet Mask . . . . . : 255.255.255.0  
    Default Gateway . . . . . : 192.168.1.1
```

```
PS >notepad
```

```
(notepad launches)
```

输入 ipconfig 显示你当前使用网络连接情况。输入 notepad 会启动 Windows 自带的文本编辑器记事本，你可以在你的机器上试试。

结构化命令

除了支持传统的 Windows 可执行文件，PowerShell 还推出了一个功能强大的新的命令叫做 *cmdlet*。**所有的 cmdlet 命名规则遵循动词 - 名词这种语法结构，如 Get-Process、Get-Content 和 Stop-Process。**

```
PS >Get-Process -Name lsass
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
668	13	6228	1660	46		932	lsass

在上面这个例子中，你为 ProcessName 参数提供了一个值，来获得指定名称的进程。

注意：一旦你掌握了一些 PowerShell 中常用的动词以后，学习如何使用新的名词就变得非常容易了。虽然你以前可能从来没有和某一个对象打过交道（例如，一个服务），标准的动作如 Get、Set、Start 和 Stop 仍然适用。要获得常用的动词，可以参见表 D-1。

你不用总是键入完整的命令名，PowerShell 会帮助你使用 Tab 键来自动完成命令输入，包括命令的名称和参数的名称：

```
PS >Get-Pr<TAB> -N<TAB> lsass
```

如果想快速地交互使用，以上的方法可能还是需要输入较多的内容。为了帮助你提高效率，PowerShell为所有常用的命令定义了别名机制，并允许你自己定义别名。在这种机制下，PowerShell只需要你输入尽可能少的参数名称，只要不和后面的参数产生冲突就行。PowerShell还区分大小写。使用内置的gps别名（表示Get-Process命令）可以键入如下命令：

```
PS >gps -n lsass
```

进一步，PowerShell支持在 cmdlet 上使用位置参数。位置参数允许你在参数指定的位置输入响应的参数，而不用输入参数名称。Get-Process命令支持名称的参数，该参数位于第一位，这个参数甚至支持通配符：

```
PS >gps l*s
```

与对象深层次的集成

PowerShell 在处理结构化的数据和对象时，向我们展示了强大的能力。例如，以下命令生成简单的文本字符串。由于没有输出，PowerShell直接将显示下面内容：

```
PS >"Hello World"  
Hello World
```

你刚刚生成的字符，事实上调用了.NET框架生成的对象，我们可以访问它的 Length 属性，这可以告诉你字符串中有多少个字符。要访问一个属性，在对象和属性名之间使用点（.），如下：

```
PS >"Hello World".Length  
11
```

所有 PowerShell 命令都将生成的输出作为对象。例如，Get-Process cmdlet 生成一个 System.Diagnostics.Process 对象，你可以把它存储在一个变量中。在 PowerShell 中，**变量的名称都以 \$ 字符开头**。如果你有一个正在运行的记事本的实例，下面的命令存储对它的引用：

```
$process = Get-Process notepad
```

由于这是一个进程对象，在 .NET Framework 中，你可以调用对象上的方法，对其执行操作。下面的命令调用 Kill() 方法，停止一个进程。若要访问一个方法，在对象和它的方法名之间放置一个圆点：

```
$process.Kill()
```

PowerShell 通过 Stop-Process cmdlet 更直接地支持此功能，但此示例演示的一个重要方面是有关你与这些丰富的对象进行交互的能力。

作为一流的系统管理员

虽然 PowerShell 对 .NET Framework 中对象的支持令大多数用户非常振奋，但是 PowerShell 还是将重点放在管理员的任务上。例如，PowerShell 支持 MB（兆字节）和 GB（千兆字节）作为标准的管理常量。例如，若要将一个 40GB 硬盘驱动器备份到 CD-ROM 需要多少磁盘？

```
PS >40GB / 650MB  
63.0153846153846
```

PowerShell 是一个主要用于管理员的 shell 程序，但这并不意味着你不能使用 .NET Framework 用于管理任务！事实上，PowerShell 提供了一个很好的日历。例如，要判断 2008 是闰年？PowerShell 可以告诉你：

```
PS >[DateTime]::IsLeapYear(2008)  
True
```

进一步，可能你要确定到夏季还有多久到来？下面的命令将“06 / 21 / 2008”（夏季的开始）转换为一个日期，然后用它减去当前日期。将结果存储在 \$ result 变量中，然后访问 TotalDays 属性。

```
PS >$result = [DateTime] "06/21/2008" - [DateTime]::Now  
PS >$result.TotalDays
```

283.0549285662616

可组合命令

每当一个命令生成输出时，可以使用一个管道字符 (|) 直接把该输出传递到另一个命令作为输入。如果第二个命令理解第一个命令所产生的对象，它可以操作结果。你可以通过这种方式把许多命令链接在一起，创建功能强大的输出。例如，下面的命令获取 Path1 目录中所有项并将它们移到 Path2 目录：

```
Get-Item Path1\* | Move-Item -Destination Path2
```

通过将其他命令 (cmdlet) 添加到该管道中，你可以创建更复杂的命令。在例 T-3 中，第一个命令获取系统中运行的所有进程。然后将结果传递给 Where-Object 命令，针对每个传入的项进行比较。在这种情况下，比较的条件是 \$ _.Handles -ge 500，检查当前对象（由 \$ _ 变量表示）的 Handles 属性是否大于或等于 500。对于此中比较结果为 true 的对象，将结果传递给 Sort-Object 命令，按它们的 Handles 属性排序。最后，将该对象传递给 Format-Table 命令生成一个表，其中包含进程的 Handles、Name 和 Description。

例 T-3：你可以通过使用管道链接 cmdlet，构建更复杂的 PowerShell 命令，如下所示使用 Get-Process、Where-Object、Sort-Object 和 Format-Table

```
PS >Get-Process |  
>> Where-Object { $_.Handles -ge 500 } |  
>> Sort-Object Handles |  
>> Format-Table Handles,Name,Description -AutoSize  
>>
```

Handles	Name	Description
588	winlogon	
592	svchost	
667	lsass	
725	csrss	
742	System	
964	WINWORD	Microsoft Office Word
1112	OUTLOOK	Microsoft Office Outlook
2063	svchost	

防止误操作的技术

虽然别名、通配符和串组合管道是功能强大的，但是在修改系统信息的命令中使用它们是非常伤脑筋的。毕竟，这个命令有什么作用是要考虑的，但是不能轻易尝试：

```
PS >gps [b-t]*[c-r] | Stop-Process
```

它似乎是要停止所有以字母 b 到 t 为开头和以字母 c 到 r 结尾的进程。你如何确保命令的执行？PowerShell 可以做到。对于那些修改数据的命令，PowerShell 支持 -WhatIf 和 -Confirm 参数，允许你查看一个命令将执行什么操作：

```
PS >gps [b-t]*[c-r] | Stop-Process -whatif  
What if: Performing operation "Stop-Process" on Target "ctfmon (812)".  
What if: Performing operation "Stop-Process" on Target "Ditto (1916)".  
What if: Performing operation "Stop-Process" on Target "dsamain (316)".  
What if: Performing operation "Stop-Process" on Target "ehrecv (1832)".  
What if: Performing operation "Stop-Process" on Target "ehSched (1852)".  
What if: Performing operation "Stop-Process" on Target "EXCEL (2092)".  
What if: Performing operation "Stop-Process" on Target "explorer (1900)".  
(...)
```

在这个交互中，使用命令 Stop-Process 的 -whatif 参数，允许你预览在实际执行该操作之前你的系统上的哪些进程将被停止。

注意不要尝试运行上面的命令：

该命令会尝试结束所有进程，但是在 Vista 中，还会强制执行关机，只留给你 1 分钟时间！

这很有趣吧，至少我有足够的时间来保存全部内容！

常用的发现命令

虽然通过阅读一个指导性的教程很有帮助，但我发现大多数学习是以特殊的方式实现的。

若要查找与给定的通配符匹配的所有命令，使用 Get-Command 命令。例如，通过输入以下命令，你可以找到 PowerShell 命令（和 Windows 应用程序）中包含单词 process 的命令。

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

CommandType	Name	Definition
Cmdlet	Get-Process	Get-Process [-Name] <Str...
Application	qprocess.exe	c:\windows\system32\qproc...
Cmdlet	Stop-Process	Stop-Process [-Id] <Int32...

若要查看一个命令可以做什么，如 Get-Process，可以使用 Get-Help 命令，如下：

```
PS >Get-Help Get-Process
```

由于 PowerShell 允许你使用 .NET Framework 中的对象，所以它提供了 Get-Member 命令来检索一个对象的属性和方法，例如，一个 .NET System.String 支持的信息。将一个字符串用管道传给 Get-Member 命令，将显示它的类型名称和成员：

```
PS >"Hello World" | Get-Member
```

Type Name	Member Type	Definition
System.String		
(...)		
PadLeft	Method	System.String PadLeft(Int32 tota...
PadRight	Method	System.String PadRight(Int32 tot...
Remove	Method	System.String Remove(Int32 start...
Replace	Method	System.String Replace(Char oldCh...
Split	Method	System.String[] Split(Params Cha...
StartsWith	Method	System.Boolean StartsWith(String...
Substring	Method	System.String Substring(Int32 st...
ToCharArray	Method	System.Char[] ToCharArray(), Sys...
ToLower	Method	System.String ToLower(), System...
ToLowerInvariant	Method	System.String ToLowerInvariant()
ToString	Method	System.String ToString(), System...
ToUpper	Method	System.String ToUpper(), System...
ToUpperInvariant	Method	System.String ToUpperInvariant()
Trim	Method	System.String Trim(Params Char[]...
TrimEnd	Method	System.String TrimEnd(Params Cha...
TrimStart	Method	System.String TrimStart(Params C...
Chars	ParameterizedProperty	System.Char Chars(Int32 index) { ...
Length	Property	System.Int32 Length {get;}

无处不在的脚本

在命令行上键入的命令和在脚本中的命令之间没有区别。在脚本中，你喜欢的命令和你喜欢的脚本技术（例如，`foreach`语句）直接作用在命令行上。例如，要统计所有正在运行的进程的句柄数：

```
PS >$handleCount = 0
PS >foreach($process in Get-Process) { $handleCount += $process.Handles }
PS >$handleCount
19403
```

PowerShell 提供一个命令 (`Measure-Object`)，以测量统计集合的信息，下面这个简短的示例演示了 PowerShell 允许你应用的通常需要一种单独的脚本或编程语言完成的技术。

除了使用 PowerShell 脚本关键字之外，你也可以直接创建和处理 .NET Framework 中的对象。**PowerShell 几乎成为了 Visual Studio 中的与 C# 一样的即时模式。**例 T-4 显示了 PowerShell 如何轻松地与 .NET Framework 进行交互。

例 T-4：使用在 .NET Framework 中的对象来检索网页和处理其内容

```
PS >$webClient = New-Object System.Net.WebClient
PS >$content = $webClient.DownloadString("http://blogs.msdn.com/PowerShell/rss.aspx")
PS >$content.Substring(0,1000)
<?xml version="1.0" encoding="UTF-8" ?>
<?xmlstylesheet type="text/xsl" href="http://blogs.msdn.com/utility/FeedS
tyleSheets/rss.xsl" media="screen"?><rss version="2.0" xmlns:dc="http://pu
rл.org/dc/elements/1.1/" xmlns:slash="http://purl.org/rss/1.0/modules/sla
sh/" xmlns:wfw="http://wellformedweb.org/CommentAPI/"><channel><title>Windo
(...)
```

特殊开发

通过模糊交互式管理和编写脚本之间的界线的方法，PowerShell 会话的历史记录缓冲区很快就会变成特殊脚本开发的基础。在此示例中，调用 `Get-History` 命令来检索你的会话的历史记录。对于每一项来说，你都可以获得它的 `CommandLine` 属性（你键入的操作），并将输出内容发送给新的脚本文件。

```
PS >Get-History | Foreach-Object { $_.CommandLine } > c:\temp\script.ps1
PS >notepad c:\temp\script.ps1
(save the content you want to keep)
PS >c:\temp\script.ps1
```

注意：如果这是你第一次在 PowerShell 中运行一个脚本，你需要配置你的执行策略。有关选择一个执行策略的详细信息，请参阅第 16.1 节。

技术的桥梁

我们已经看到 PowerShell 如何让你可以完全利用 .NET Framework，但它对常见的技术提供了进一步的支持。如例 T-5 所示，PowerShell 支持 XML：

例 T-5：使用 PowerShell 中的 XML 内容

```
PS >$xmlContent = [xml] $content
PS >$xmlContent

xml                               xmlstylesheet      rss
---                               -----

  <channel>
    <item>
      <title>Windows 7: How to Create a Log File for a Specific Application</title>
      <link>http://www.microsoft.com/technet/windows7/fixes/w7log.mspx?mfr=true</link>
      <description>Windows 7: How to Create a Log File for a Specific Application</description>
      <pubDate>2009-01-20T10:00:00Z</pubDate>
      <category>Windows 7: How to Create a Log File for a Specific Application</category>
    </item>
  </channel>

PS >$xmlContent.rss

version : 2.0
dc      : http://purl.org/dc/elements/1.1/
slash   : http://purl.org/rss/1.0/modules/slash/
wfw     : http://wellformedweb.org/CommentAPI/
channel : channel

PS >$xmlContent.rss.channel.item | select Title
title
-----
CMD.exe compatibility
Time Stamping Log Files
Microsoft Compute Cluster now has a PowerShell Provider and Cmdlets
The Virtuous Cycle: .NET Developers using PowerShell
(...)
```

Powershell 也允许你使用 Windows 管理设备（Windows Management Instrumentation, WMI）：

```
PS >Get-WmiObject Win32_Bios
SMBIOSBIOSVersion : ASUS A7N8X Deluxe ACPI BIOS Rev 1009
Manufacturer       : Phoenix Technologies, LTD
Name               : Phoenix - AwardBIOS v6.00PG
SerialNumber       : xxxxxxxxxxxx
Version            : Nvidia - 42302e31
```

例 T-6 显示了对活动目录服务接口（Active Directory Service Interfaces, ADSI）的支持：

例 T-6：PowerShell 中使用活动目录

```
PS >[ADSI] "WinNT://./Administrator" | Format-List *
```

正如例 T-7 显示的，甚至可以对传统的 COM 对象编写脚本：

例 T-7：PowerShell 中使用 COM 对象

```
PS >$firewall = New-Object -com HNetCfg.FwMgr  
PS >$firewall.LocalPolicy.CurrentProfile
```

```
Type : 1
FirewallEnabled : True
ExceptionsNotAllowed : False
NotificationsDisabled : False
UnicastResponsesToMulticastBroadcastDisabled : False
RemoteAdminSettings : System.__ComObject
IcmpSettings : System.__ComObject
GloballyOpenPorts : {Media Center Extender Service, Remote Media Center Experience, Adam Test Instance, QWAVE...}
Services : {File and Printer Sharing, UPnP Framework, Remote Desktop}
AuthorizedApplications : {Remote Assistance, Windows Messenger, Media Center, Trillian...}
```

通过提供程序导航命名空间

PowerShell 提供了另一个处理系统的方法是提供程序 (providers)。PowerShell 提供程序允许我们使用与操作文件系统相同的技术来导航和管理数据存储区，如例 T-8 所示。

例 T-8：导航文件系统

```
PS >Set-Location c:\  
PS >Get-ChildItem  
  
Directory: Microsoft.PowerShell.Core\FileSystem::C:\
```

Mode	LastWriteTime	Length	Name
d----	11/2/2006	4:36 AM	\$WINDOWS.~BT
d----	5/8/2007	8:37 PM	Blurpark
d----	11/29/2006	2:47 PM	Boot
d----	11/28/2006	2:10 PM	DECCHECK
d----	10/7/2006	4:30 PM	Documents and Settings
d----	5/21/2007	6:02 PM	F&SC-demo
d----	4/2/2007	7:21 PM	Inetpub
d----	5/20/2007	4:59 PM	Program Files
d----	5/21/2007	11:47 PM	temp
d----	5/21/2007	8:55 PM	Windows
-a---	1/7/2006	10:37 PM	0 autoexec.bat
-ar-s	11/29/2006	1:39 PM	8192 BOOTSECT.BAK
-a---	1/7/2006	10:37 PM	0 config.sys
-a---	5/1/2007	8:43 PM	33057 RUU.log
-a---	4/2/2007	7:46 PM	2487 secedit.INTEG.RAW

这也可用于注册表，如例 T-9 所示：

例 T-9：导航注册表

```

PS >Set-Location HKCU:\Software\Microsoft\Windows
PS >Get-ChildItem
    Name          Value
    --          --
Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Microsoft\Windows
SKC  VC  Name           Property
---  --  ----
 30   1  CurrentVersion {ISC}
   3   1  Shell           {BagMRU Size}
   4   2  ShellNoRoam   {(default), BagMRU Size}

PS >Set-Location CurrentVersion\Run
PS >Get-ItemProperty

```

```
(...)  
FolderShare          : "C:\Program Files\FolderShare\Fol  
                      ckground  
TaskSwitchXP        : d:\lee\tools\TaskSwitchXP.exe  
ctfmon.exe          : C:\WINDOWS\system32\ctfmon.exe  
Ditto               : C:\Program Files\Ditto\Ditto.exe  
(...)
```

甚至可用于计算机的证书存储区，如例 T-10 所示。

例 T-10：浏览证书存储区

```
PS >Set-Location cert:\CurrentUser\Root  
PS >Get-ChildItem
```

Thumbprint	Subject
CDD4EEAE6000AC7F40C3802C171E30148030C072	CN=Microsoft Root Certificate...
BE36A4562FB2EE05DBB3D32323ADF445084ED656	CN=Thawte Timestamping CA, OU...
A43489159A520F0D93D032CCAF37E7FE20A8B419	CN=Microsoft Root Authority, ...
9FE47B4D05D46E8066BAB1D1BFC9E48F1DBE6B26	CN=PowerShell Local Certifica...
7F88CD7223F3C813818C994614A89C99FA3B5247	CN=Microsoft Authenticode(tm)...
245C97DF7514E7CF2DF8BE72AE957B9E04741E85	OU=Copyright (c) 1997 Microso...
(...)	

[更多的內容](#)

这个指导性的教程是令人振奋的，介绍了 PowerShell 可以使你提高工作效率和系统管理技能的一些亮点。有关 PowerShell 入门的更多信息，请参阅第 1 章。

最新最
好發售

示例 01-T 阅读：阅读并理解以下段落。

凶神恶煞五黄临 01-丁酉

the [classic](#) gift: [Customize](#)/Book

<http://www.micromouse.com/micromouse/micromouse.html>

容內附送更

。章一葉參差——惠亮一些的道姓照。有关 Powershell 入门的更多消息，[请参阅](#)

第二部分

基础知识

第1章，Windows PowerShell交互界面

第2章，管道

第3章，变量与对象

第4章，循环与流程控制

第5章 字符串与非结构化文本

第6章，计算和数学计算

代码二集

基础入门

第1章 Windows Powershell 简介

重写 第1章
字体已重写 第1章
储藏库已取消 第1章
本文档非已弃用 第1章
算符与运算符 第1章

Windows PowerShell 交互界面

1.0 绪论

Windows PowerShell 在设计的时候将它的高效率和强大的交互性放在首位。虽然脚本语言起着关键的作用，但是交互式的使用发挥了重要作用。

令大多数人吃惊的是，当他们第一次运行 PowerShell 的时候，感觉与已经存在很长时间的 Windows 命令提示符很相似。熟悉的工具继续使用，熟悉的命令继续使用，甚至连快捷键都一样。通过这个熟悉的用户界面，PowerShell 提供了强大的引擎，可以让你一次轻松地完成繁重的管理与脚本任务。

本章将从交互性的角度来介绍 PowerShell。

1.1 运行程序、脚本和已有的工具

问题

你依赖着那些已经花费你很多精力来学习使用的工具。在日常工作中，你可能已经有一些可执行文件、Perl 脚本、VBScript 等，或者遗留的错综复杂的用于编译的批处理文件。你想使用 PowerShell，但又不想放弃已经有的工具。

解决方案

在系统路径下运行程序、脚本、批处理文件或其他可执行文件，可以键入其文件名。对于可执行类型的文件，可以不用输入扩展名，如：

Program.exe arguments

```
ScriptName.ps1 arguments  
BatchFile.cmd arguments
```

运行那些在命令名中包含空格的命令，需要用引号（'）将命令括起来，同时在前面加上符号（&），这在 PowerShell 中称作调用操作（Invoke operator），如：

```
& 'C:\Program Files\Program\Program.exe' arguments
```

运行当前目录下的命令，在文件名前添加.\，如：

```
.\Program.exe arguments
```

在当前目录下运行那些命令名中包含空格的命令，同时加上符号（&）和.\，如：

```
& '.\Program With Spaces.exe' arguments
```

讨论

在这个案例中，解决方案主要关注的是如何使用你以前使用过的工具。唯一不同的是，你是在 PowerShell 的交互界面中运行程序，而不是以前的 cmd.exe。

解决方案中最后3个技巧值得注意。许多刚刚使用PowerShell的用户在运行程序的时候，会对这部分功能感到困惑。第10个演示了如何处理命令中包含空格的问题。在原来的命令提示符下，运行方法是用引号将命令括起来，如：

```
"C:\Program Files\Program\Program.exe"
```

而在 PowerShell 中，将文本用引号括起来，可以让你在提示符下进行复杂的运算，如例 1-1 所示：

例 1-1：在 PowerShell 提示符下，进行表达式运算

```
PS >1 + 1  
2  
PS >26 * 1.15  
29.9  
PS >"Hello" + " World"  
Hello World  
PS >"Hello World"  
Hello World  
PS >"C:\Program Files\Program\Program.exe"  
C:\Program Files\Program\Program.exe  
PS >
```

所以说，在引号中的程序的名称与其他在引号中的字符没有区别，都是一个表达式。如前面解决方案所示，运行一个包含在字符串中的命令的方法是在字符串前面加上前缀（&）。如果你想运行一个批处理文件来修改环境变量，请参见第 1.8 节。

注意：默认情况下，PowerShell的安全策略会阻止脚本的运行。一旦你开始编写或运行脚本的时候，你就需要配置策略来减少限制。关于如何配置脚本执行策略，可以参见第 16.1 节。

第二个例子有时会令新手迷惑的就是在当前目录下运行命令。在 cmd 提示符下，当前目录会自动被认为是路径的一部分，Windows 会在这个路径下搜索你键入的程序。比如说你目前在 C:\Programs 目录下，cmd.exe 会在 C:\Programs 目录下查找程序来运行。

PowerShell 与大多数 Unix shell 一样，要求你明确地指出你要在当前目录下运行程序。这样做，你需要使用像前面提到的 .\Program.exe 这样的语法。这样做的目的是，当你访问某一目录的时候，可以有效地防止一些用户在你的硬盘上非法散布一些与你要运行的程序名称相近（或相同）的恶意程序。

许多用户将经常用到的工具和 PowerShell 脚本放到系统目录的“tools”目录下，这样就不用在每次运行程序的时候键入路径了。如果 PowerShell 能够在系统路径下找到你键入的命令，你可以不用刻意地输入路径。

注意：本书中用到的脚本与示例可以从 <http://www.oreilly.com/catalog/9780596528492> 下载。

关于学习如何编写 PowerShell 脚本，可以参见 10.1 节。

参考

- 1.8 节
- 10.1 节
- 16.1 节

1.2 运行 PowerShell 命令

问题

你要运行 PowerShell 命令。

解决方案

在命令提示符下键入你要运行的命令的名字，如：

```
PS >Get-Process
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
133	5	11760	7668	46	1112	audiogd	
184	5	33248	508	93	1692	avgamsvr	
143	7	31852	984	97	1788	avgemc	

讨论

Get-Process 命令是一个内置的 PowerShell 命令，我们称之为 *cmdlet*。它与传统命令的区别在于，cmdlet 为管理员和开发人员提供了重要的功能：

- 采用同一的命令语法。
- 支持丰富的管道功能（将一个命令的输出作为另外一个命令的输入）。
- **输出易于管理的对象，而不是容易出错的文本。**

由于 Get-Process 命令生成了丰富的基于对象的输出，所以你可以利用这个输出来执行一些与进程相关的任务。

Get-Process 命令仅仅是 PowerShell 支持的命令之一。要学习如何查找 PowerShell 命令的技术，请参见 1.4 节。要获得如何使用 .NET 框架提供的类的更多信息，请参见 3.4 节。

参考

- 1.4 节
- 3.4 节

1.3 自定义 shell、配置文件与提示符

问题

你希望通过个性化的提示符、别名来定制 PowerShell 的交互 shell。

解决方案

你可以通过将个性化的内容放到你的个人配置脚本文件中来实现 PowerShell shell 的个性化。PowerShell 将脚本文件的地址存放在 \$profile 变量中，你可以通过这个变量来访问你的脚本文件。

注意：默认情况下，PowerShell的安全策略会阻止脚本（包括你的配置脚本）的运行。一旦你开始编写或运行脚本，你就需要配置策略来减少限制。关于如何配置脚本执行策略，可以参考 16.1 节。

创建一个新的配置脚本（或者覆盖已经存在的配置脚本），可以输入下面命令：

```
New-Item -type file -force $profile
```

编辑已经存在的配置脚本，可以输入下面的命令：

```
notepad $profile
```

察看你的配置脚本文件的内容，可以输入下面的命令：

```
Get-ChildItem $profile
```

当你创建了配置脚本文件后，你可以添加一个名为 Prompt 的函数，用来返回一个字符串。PowerShell 会将 Prompt 函数返回的字符串作为提示符。

```
function Prompt
{
    "PS [$env:COMPUTERNAME] >"
```

这个例子会在提示符中显示计算机的名字，看起来像：PS [计算机的名字] >

你会发现在配置文件中加入别名会对你有很大帮助。别名可以让你根据自己选择的名字来引用常用的命令。你可以在个人的配置脚本文件中定义别名、函数、变量或其他你可能在 PowerShell 提示符下设置的参数。别名可能是你在脚本文件中最常配置的项，通过别名，你可以把 PowerShell 的命令（或你的脚本）用容易输入的名字来引用。

注意：如果你在为命令定义别名的同时，需要修改命令的参数，建议你用函数来替代别名。

例如：

```
Set-Alias new New-Object
Set-Alias iexplore 'C:\Program Files\Internet Explorer\iexplore.exe'
```

你对配置文件的修改会在你保存配置文件并重新启动 PowerShell 后生效。如果你希望修改马上生效，可以键入下面命令：

```
.$profile
```

函数也经常用来进行个性化定制，比较受欢迎的是自定义 Prompt 函数。

讨论

虽然上面的 Prompt 函数只是返回一个简单的字符串，但你也可以编写更为复杂的函数。例如，许多用户会更新他们控制窗口的标题（通过修改 \$host.UI.RawUI.WindowTitle 变量），或者使用 Write-Host 命令修改输出提示的颜色。如果你的 Prompt 函数自己处理屏幕输出，那么它需要一个返回字符串（如一个空格）来防止 PowerShell 使用默认的提示符。如果你不想在提示符中显示额外的空格，你可以在 write-Host 命令结尾加入空格，返回退格（“`b”）字符，如例 1-2 所示。

例 1-2：PowerShell 提示符例子

```
function Prompt
{
    $id = 1
    $historyItem = Get-History -Count 1
    if($historyItem)
    {
        $id = $historyItem.Id + 1
    }
    Write-Host -ForegroundColor DarkGray "`n[$(Get-Location)]"
    Write-Host -NoNewLine "PS:$id > "
    $host.UI.RawUI.WindowTitle = "$(Get-Location)"
    "`b"
}
```

[C:\] PS:73 >5 * 5
[C:\] PS:74 >1 + 1
2
[C:\] PS:75 >Invoke-History 73
5 * 5
25
[C:\] PS:76 >

通过引用 \$profile 变量你可以定义你要引用的配置文件，PowerShell 实际上最多支持 4 个配置脚本文件。关于如何使用这些配置脚本的说明（包括其他配置项），可以查阅附录 A。

参考

- 16.1 节
- 附录 A 中的“通用设定的项”

1.4 查找实现指定任务的命令

问题

你要在 PowerShell 中执行一条任务，但是不知道用什么命令来执行。

解决方案

使用 Get-Command 这个命令来检索命令。

如果想获得指定命令的概要信息，可以将命令的名称作为 Get-Command 的参数，如下：

```
Get-Command CommandName
```

如果想获得指定命令的详细信息，将 Get-Command 的输出结果重新定向到 Format-List 命令，如下：

```
Get-Command CommandName | Format-List
```

如果想搜索包括“text”的所有命令，用星号（*）将它括起来，如下：

```
Get-Command *text*
```

如果想搜索所有使用 Get 动词的命令，在 -Verb 后输入参数 Get，如下：

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

如果想搜索所有与服务有关的命令，在 -Noun 后加入参数 Service 如下：

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

讨论

PowerShell 的优点之一是为管理员提供一致的命令名称。所有的 PowerShell 命令（又称作 cmdlets）都遵循动词-名词的模式。比如：Get-Process、Get-Eventlog、Set-Location 等。这些动词取自一个标准的动词集合（参考附录 D），描述了 cmdlet 采取的动作；名词描述了 cmdlet 对谁采取动作。

了解这个规则后，你可以轻松地学习与使用系统提供的这些 cmdlets。如果你要启动本机的一个服务，应该使用的动词是 Start，根据规则，你可能会首先试一下 Start-Service（在这里是正确的）。更为有效的方法是你可以通过 Get-Command -Verb Start 来查看可以开始哪些任务，当然，你也可以通过 Get-Command -Noun Service 来查看在服务上都可以采取哪些动作。

要获得所有命令的列表以及它们的详细使用说明可以参考 1.5 节。

当你在使用 Windows PowerShell 的时候，Get-Command 是你最经常使用的三个命令之一，其他两个命令是 Get-Help 和 Get-Member。

值得注意的是，当我们试图查找 PowerShell 命令来完成某一任务的时候，经常会发现没有相关的命令。实际上，完成一个任务很多时候还是使用已经存在的命令是最佳的选择，例如：我们还是使用 shutdown.exe 来重新启动机器；使用 netstat.exe 来查看 TCP/IP 网络协议统计与连接情况。

要获得更多的关于 Get-Command 命令的信息，可以键入 **Get-Help Get-Command**。

参考

- 1.5 节

1.5 获得命令帮助

问题

你希望学习特定命令是如何工作的，以及如何使用。

解决方案

通过 Get-Help 命令，你可以获得命令的使用帮助。你还可以通过不同的渠道来获得帮助，这取决于你的需要。

如果想获得一个特定命令的概要帮助信息，可以将命令的名字作为参数传给 Get-Help 命令。帮助中主要包括命令的大纲、语法以及细节描述：

```
Get-Help CommandName
```

或

```
CommandName -?
```

如果想获得一个特定命令帮助信息的详情，使用 **Get-Help cmdlet** 的 **-Detailed** 参数。除了可以查看摘要还可以阅读其参数说明和示例：

```
Get-Help CommandName-Detailed
```

如果想获得一个特定命令的详细的帮助信息，就要为 **Get-Help** 提供 **-Full** 标记。在帮助的内容中，包括了该命令的全部参数描述和注释信息：

```
Get-Help CommandName -Full
```

如果想获得一个特定命令的例子信息，就要为 **Get-Help** 提供 **-Examples** 标记：

```
Get-Help CommandName -Examples
```

讨论

在 PowerShell 中，我们主要通过 **Get-Help** 命令来获取帮助。与 **Get-Command** 一样，**Get-Help** 支持通配符。如果你希望列出所有匹配某一模式的命令（比如，***process***），你只需要键入命令 **Get-Help *process***。

注意：如果希望列出所有 cmdlets 的使用大纲，运行下面的命令：

```
Get-Help * | Select-Object Name,Synopsis | Format-Table -Auto
```

如果只匹配到一个命令，PowerShell 会显示那个命令的帮助信息。虽然采用通配符方式可以有效地搜索 PowerShell 的帮助内容，我们仍然可以通过脚本来更准确地搜索帮助内容，具体操作可以参考 1.6 节。

当你在使用 Windows PowerShell 的时候，**Get-Help** 应该是你最经常使用的三个命令之一，其他两个命令是 **Get-Command** 和 **Get-Member**。

获得更多的关于 **Get-Help** 的信息，可以键入 **Get-Help Get-Help**。

参考

- 1.6 节

1.6 编程：搜索帮助

通过Get-Command和Get-Help这两个命令，你可以搜索匹配某一模式的命令名。然而，当你不能确切地知道你要查找的命令名时，你就需要通过搜索帮助的内容（content）来获得答案。在UNIX系统中，这个命令称作Apropos，但是在PowerShell中，没有类似的命令。

当然，我们可以通过编写函数来实现同样的功能。

以下是我们编写的搜索帮助的脚本（例1-3），运行时需要提供一个搜索的字符，这个字符既可以是简单的文本，也可以是正则表达式。在搜索的结果中，显示的是所有匹配的帮助信息的名称与大纲。使用Get-Help查看相关的帮助内容。

例1-3：Search-Help.ps1

```
#####
## Search-Help.ps1
##
## Search the PowerShell help documentation for a given keyword or regular
## expression.
## Example:
##     Search-Help hashtable
##     Search-Help "(datetime|ticks)"
#####
param($pattern = $(throw "Please specify content to search for"))

$helpNames = $(Get-Help * | Where-Object { $_.Category -ne "Alias" })

foreach($helpTopic in $helpNames)
{
    $content = Get-Help -Full $helpTopic.Name | Out-String
    if($content -match $pattern)
    {
        $helpTopic | Select-Object Name, Synopsis
    }
}
```

关于更多运行脚本的信息，请参考1.1节。

参考

- 1.1节

1.7 在 PowerShell 之外调用 PowerShell 脚本

问题

你想从批处理文件、登录脚本、定时任务或其他非PowerShell应用程序中调用PowerShell脚本。

解决方案

可以采用下面的方式运行 PowerShell：

```
PowerShell "& 'full path to script' arguments"
```

例如：

```
PowerShell "& 'c:\shared scripts\Get-Report.ps1' Hello World"
```

讨论

就像在交互界面下运行命令一样，你可以为 *PowerShell.exe* 提供一个字符串参数来运行命令。如果存放脚本的路径包含空格，需要用单引号把它括起来，并在单引号前面加上字符 &；如果路径中没有空格，那么可以忽略单引号和字符 &。这样，你可以在登录脚本、高级的文件操作、定时任务中调用 PowerShell 脚本。

注意：如果你在自己的程序中需要调用 PowerShell 脚本或命令，PowerShell 提供了更简捷的方法让你来调用这些脚本与命令。关于详细的说明，请参考 15.13 节。

如果脚本命令变得很复杂，同时应用程序调用中的特殊字符还要与 PowerShell 交互，比如说在 *cmd.exe* 提示符下，你可以利用 PowerShell 提供的 *EncodedCommand* 参数，把你想要运行的脚本命令用 Unicode 进行 Base64 编码。例 1-4 演示了如何将包含 PowerShell 命令的字符串进行 Base64 编码。

例 1-4：把 PowerShell 命令转换为 Base64 编码格式

```
$commands = '1..10 | % {"PowerShell Rocks"}'  
$bytes = [System.Text.Encoding]::Unicode.GetBytes($commands)  
$encodedString = [Convert]::ToBase64String($bytes)
```

一旦获得编码后的字符串，你就可以将它作为 *EncodedCommand* 参数的值，如例 1-5 所示：

例 1-5：在 cmd 下加载 Powershell，传递编码后的命令

Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

更多关于如何调用 PowerShell 脚本的说明，可以参考 1.1 节。

参考

- ## • 1.1 节

1.8 编程：保持批处理文件修改的环境变量

当一个批处理文件运行时，它可能会修改一个环境变量，即使批处理文件结束后，*cmd.exe*仍然会保持这个变更。这样经常会产生问题，因为批处理文件有可能无意中修改其他程序设置的环境变量。也就是说，批处理文件的作者为了完成某个任务，往往会展全局的环境变量、路径以及其他内容。

当然，环境变量是一个进程所特有的，当进程退出后也随之消失。我们之前提到的修改环境变量的脚本在PowerShell下就失去作用了，同样地，如果你在另外一个cmd窗口运行它们也会无效。

例 1-6 中的脚本可以让你运行批处理文件，修改环境变量，当 cmd 窗口关闭后保持变更的内容。实现的原理是这样的：当批处理文件运行结束后，我们把环境变量保存到文本文件中，然后在 PowerShell 会话中设置这些变量。

你通过键入 `Invoke-CmdScript Scriptname.cmd` 或 `Invoke-CmdScript Scriptname.bat` 运行这个脚本。

注意：如果你是第一次在PowerShell中运行脚本，你需要配置你的执行策略。关于更多执行策略的内容，请参考16.1节。

注意在这个脚本中，所有的cmdlets都使用了完整的名称，如：Get-Content、ForEach-Object、Set-Content和Remove-Item。这样既增加了脚本的可读性，同时也方便其他人阅读。当然，采用更简短的别名（如gc、%、sc、ri等）可以使你工作得更高效。

例1-6：Invoke-CmdScript.ps1

```
#####
## Invoke-CmdScript.ps1
##
## Invoke the specified batch file (and parameters), but also propagate any
## environment variable changes back to the PowerShell environment that
## called it.
##
## i.e., for an already existing 'foo-that-sets-the-FOO-env-variable.cmd':
##
## PS > type foo-that-sets-the-FOO-env-variable.cmd
## @set FOO=%*
## echo FOO set to %FOO%.
##
## PS > $env:FOO
##
## PS > Invoke-CmdScript "foo-that-sets-the-FOO-env-variable.cmd" Test
##
## C:\Temp>echo FOO set to Test.
## FOO set to Test.
##
## PS > $env:FOO
## Test
##
#####
param([string] $script, [string] $parameters)
$tempFile = [IO.Path]::GetTempFileName( )

## Store the output of cmd.exe. We also ask cmd.exe to output
## the environment table after the batch file completes
cmd /c `"$script`" $parameters && set > `"$tempFile`" `

## Go through the environment variables in the temp file.
## For each of them, set the variable in our local environment.
Get-Content $tempFile | ForEach-Object {
    if($_ -match "^(.*)=(.*)$")
    {
        Set-Content "env:\$($matches[1])" $matches[2]
    }
}

Remove-Item $tempFile
```

关于运行脚本的更多信息，请参考 1.1 节。

参考

- **1.1 节** 介绍了如何使用 PowerShell 命令行界面，其中包含有关如何运行脚本的信息。
- **16.1 节** 将介绍如何（通过将命令写入文件）将脚本脚本重用起来，然后将其插入其他脚本中。

1.9 获取系统日期与时间

问题

你要获取系统日期。

解决方案

运行命令 Get-Date。

```
PS>Get-Date  
Wednesday, April 22, 2009 10:45:07 AM  
(UTC+08:00) Beijing, Shanghai, Hong Kong, Taipei, Tokyo
```

讨论

Get-Date 命令生成了强大的基于对象的输出，所以你可以在此基础上进一步执行与日期相关的任务。比如，检查今天是星期几，键入下面命令：

```
PS>$date = Get-Date  
PS>$date.DayOfWeek  
Sunday
```

关于 Get-Date 的更多信息，可以通过命令 **Get-Help Get-Date** 来获取。

关于如何使用 .NET 框架中的类，请参考 3.4 节。

参考

- **3.4 节**

1.10 检查最后运行命令的状态

问题

你希望获得你最后执行命令的状态信息，比如检查命令是否成功执行。

解决方案

PowerShell 提供了两个变量可以用来监测你最后执行的命令是否成功，这两个变量是 \$lastExitCode 变量和 \$? 变量。

变量 \$lastExitCode

数字型，它返回最后脚本或应用程序执行返回的退出码或出错级别；

变量 \$?

布尔型，它返回最后执行命令的成功 (true) 或失败 (false)。

讨论

PowerShell 中的 \$lastExitCode 变量与 DOS 下的 %errorlevel% 变量类似，用来保存最后一个应用程序的退出码。这样的话，我们可以使用退出码作为首要的通信机制继续与传统的可执行程序（如 ping、findstr 和 choice）进行交互。PowerShell 还扩展了 \$lastExitCode 的内容，包含了脚本的退出码，可以用来设置退出的状态。例 1-7 演示了这个交互：

例 1-7：与变量 \$lastExitCode 和 \$? 交互

```
PS >ping localhost

Pinging MyComputer [127.0.0.1] with 32 bytes of data:

Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Ping statistics for 127.0.0.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milliseconds:
        Minimum = 0ms, Maximum = 0ms, Average = 0ms
PS >$?

True
PS >$lastExitCode
0
PS >ping missing-host
Ping request could not find host missing-host. Please check the name and try again.
PS >$?
False
PS >$lastExitCode
1
```

变量 \$? 使用更通用的方式来描述最后的应用程序退出的状态。在以下应用程序发生错误的时候，PowerShell 会设置 \$? 变量为 False：

- 应用程序的退出码非零。
- cmdlet 或脚本输出错误信息。
- cmdlet 或脚本捕获到终止错误或异常。

当命令执行没有错误的时候，PowerShell 设置 \$? 变量为 True。

1.11 度量命令执行时间 问题

你想了解执行一个命令需要多少时间。

解决方案

计算一个命令执行的时间，可以使用 Measure-Command 命令，如下所示：

```
PS >Measure-Command { Start-Sleep -Milliseconds 337 }
```

Days	:	0
Hours	:	0
Minutes	:	0
Seconds	:	0
Milliseconds	:	339
Ticks	:	3392297
TotalDays	:	3.92626967592593E-06
TotalHours	:	9.42304722222222E-05
TotalMinutes	:	0.00565382833333333
TotalSeconds	:	0.3392297
TotalMilliseconds	:	339.2297

讨论

在交互使用情况下，计算一个命令执行的时间是比较常见的。一个常见的例子是对你开发的应用程序进行性能测试。Measure-Command 使这项工作变得十分容易。与此同时，Measure-Command 命令会生成丰富的基于对象的输出，你可以在此基础上进一步执行与时间相关的任务。更多内容请参考 3.4 节，关于 Measure-Command 的更多信息可以键入命令 Get-Help Measure-Command。

参考

- 3.4 节

1.12 定制 shell 来提升工作效率

问题

你希望有效地使用 PowerShell 控制台，进行复制、粘贴、历史纪录管理和滚屏等。

解决方案

运行例1-8中的代码来永久地定制你的PowerShell 控制台窗口，这使许多工作变得容易。

例 1-8：Set-ConsoleProperties.ps1

```
Push-Location
Set-Location HKCU:\Console
New-Item '.\%SystemRoot%\system32\WindowsPowerShell_v1.0_powershell.exe'
Set-Location '..\%SystemRoot%\system32\WindowsPowerShell_v1.0_powershell.exe'

New-ItemProperty . ColorTable00 -type DWORD -value 0x00562401
New-ItemProperty . ColorTable07 -type DWORD -value 0x00f0edee
New-ItemProperty . FaceName -type STRING -value "Lucida Console"
New-ItemProperty . FontFamily -type DWORD -value 0x00000036
New-ItemProperty .FontSize -type DWORD -value 0x000c0000
New-ItemProperty .FontWeight -type DWORD -value 0x00000190
New-ItemProperty .HistoryNoDup -type DWORD -value 0x00000000
New-ItemProperty .QuickEdit -type DWORD -value 0x00000001
New-ItemProperty .ScreenBufferSize -type DWORD -value 0x0bb80078
New-ItemProperty .WindowSize -type DWORD -value 0x00320078
Pop-Location
```

在上面这个例子中，我们定义了控制台的颜色、字体、历史存储属性、快速编辑的模式、缓冲区的大小以及窗口的大小。

通过这些调整，通过学习常用命令的快捷键，你可以提高工作效率。常用命令的快捷键的列表参见表 1-1。PowerShell 使用与 cmd.exe 相似的输入习惯，如果你已经对 cmd.exe 所有的输入功能都很熟悉，那么你对 PowerShell 控制台也会很快上手。

表 1-1：部分 Windows PowerShell 快捷键

快捷键	含义
Up 箭头	向前搜索历史命令
Down 箭头	向后搜索历史命令
PgUp	显示历史命令中第一个命令
PgDown	显示历史命令中最后一个命令
Left 箭头	在命令行中，把光标向左移动一个字符
Right 箭头	在命令行中，把光标向右移动一个字符

表 1-1：部分 Windows PowerShell 快捷键（续）

快捷键	含义
Home	在命令行中，把光标移到开始位置
End	在命令行中，把光标移到结束位置
Ctrl + Left 箭头	在命令行中，把光标向左移动一个单词
Ctrl + Right 箭头	在命令行中，把光标向右移动一个单词

讨论

当你从 Windows 开始菜单中启动 PowerShell 的时候，需要初始化控制窗口的几个选项：

- 前景和背景的颜色，使窗口看起来更吸引人。
- 快速编辑模式，使得用鼠标复制与粘贴更加容易。
- 缓冲区大小，使得 PowerShell 可以在控制台窗口历史记录中保留更多的命令。

默认的情况下，当你从开始（Start）到运行 Run 对话框中运行 PowerShell 的时候，以上定制是不起作用的。通过应用本解决方案中的命令，可以使这些设置生效，从而改善你的使用体验。

当然，快捷键的功能适用于所有的 PowerShell 窗口（包括那些使用 Windows 的传统输入模式的应用程序）。常用的设置都在本解决方案中列出来了，在附录 A 中有完整的列表。

参考

- 附录 A 中的“常用的设置选项”

1.13 编程：学习常用命令的别名

在交互使用 PowerShell 的时候，输入完整的命令的名字是一件很繁琐的事情，而且输入的速度也慢。虽然通过命令的别名可以使事情变得容易，但是你可能需要花费时间来掌握这些别名。为了使学习别名更容易，你可以通过修改提示符来提醒你使用这些命令的别名。

包括以下两个步骤：

- 把下面的程序Get-AliasSuggestion.ps1加到你的工具目录或其他目录中，代码如下所示：

例 1-9: Get-AliasSuggestion.ps1

```
#####
## Get-AliasSuggestion.ps1
##
## Get an alias suggestion from the full text of the last command
##
## ie:
##
## PS > Get-AliasSuggestion Remove-ItemProperty
## Suggestion: An alias for Remove-ItemProperty is rp
##
#####
param($lastCommand)

$helpMatches = @()

## Get the alias suggestions
foreach($alias in Get-Alias)
{
    if($lastCommand -match ("\\b" +
        [System.Text.RegularExpressions]::Escape($alias.Definition) + "\\b"))
    {
        $helpMatches += "Suggestion: An alias for $($alias.Definition) is $($alias.Name)"
    }
}

$helpMatches
```

- 将例 1-10 中的代码复制到你的配置文件中的 Prompt 函数中。如果你的配置文件中还没有 Prompt 函数，可以参考 1.3 节，来建立配置文件。如果你已经有配置文件和函数，那么只需要将下面的内容加到配置文件的 Prompt 函数中即可。

例 1-10: 帮助你学习常用命令的别名的提示符

```
function Prompt
{
    ## Get the last item from the history
    $historyItem = Get-History -Count 1

    ## If there were any history items
    if($historyItem)
    {
        ## Get the training suggestion for that item
        $suggestions = @(Get-AliasSuggestion $historyItem.CommandLine)
        ## If there were any suggestions
        if($suggestions)
        {
            ## For each suggestion, write it to the screen
```

```
foreach($aliasSuggestion in $suggestions)
{
    Write-Host "$aliasSuggestion"
}
Write-Host ""

## Rest of prompt goes here
"PS [$env:COMPUTERNAME] >"
```

更多如何运行脚本的信息可以参考 1.1 节。

参考

- 1.1 节
- 1.3 节

1.14 使用与管理控制台历史

问题

当你在控制台工作一段时间后，你希望从历史命令中调用某一个命令，查看已经输入的命令记录并保存这些命令。

解决方案

在 1.12 节中，我们已经了解了如何管理历史命令。当然，除此之外，PowerShell 还提供了另外几个功能帮助你更好地使用控制台。

在使用过程中，获得最近使用的命令，可以使用 Get-History 命令，如下所示：

```
Get-History
```

在使用过程中，从历史命令中获得指定的某个命令，可以将命令的 *Id* 传给 Invoke-History cmdlet，如下所示：

```
Invoke-History Id
```

通过给变量 \$MaximumHistoryCount 赋值，可以增加或限制会话历史中存储命令的数量，如下所示：

```
$MaximumHistoryCount = Count
```

如果你希望保存你输入的命令的历史记录到一个文件，可以使用管道功能将 Get-History 输出到 Export-CliXml 命令中，如下所示：

```
Get-History | Export-CliXml Filename
```

如果你希望将已经保存的命令的历史加载到当前的控制台，可以使用 Import-CliXml 命令，然后利用管道功能将结果重定向到 Add-History 命令，如下所示：

```
Import-CliXml Filename | Add-History
```

讨论

在 1.12 节中，我们讨论了如何使用快捷键，而 Get-History 命令可以生成内容丰富的结果来显示历史记录中的信息。每个对象包括项 ID、命令行、开始执行的时间和结束的时间。

一旦你知道了历史项的 ID（如 Get-History 的输出所示），你可以通过把 ID 传给 Invoke-History 的方法来再次执行那个命令。1.3 节中的提示函数的例子使得调用历史项变得很容易，只需要在提示符中加入该命令的历史 ID 即可。

注意：Get-History 中返回的 ID 与 Windows 控制台返回的 ID（使用 F7 键）是不一样的，因为两者对历史项管理采用了不同的技术。

默认的情况下，PowerShell 只保留最后 64 个命令。如果你想调整这个值，可以通过设置变量 \$MaximumHistoryCount 来达到目的。如果要这个设置永久生效，可以在 PowerShell 配置文件中设置这个变量。如果要清除历史记录，可以重新运行 PowerShell 或暂时将 \$MaximumHistoryCount 变量设为 1。

参考

- 1.3 节
- 1.12 节

1.15 将命令的输出保存到文件中

问题

你希望将管道输出的结果保存到文件中。

解决方案

可以通过 Out-File 命令或者重定向操作符将命令输出的结果保存到文件中。

```
Out-File:  
Get-ChildItem | Out-File unicodeFile.txt  
Get-Content filename.cs | Out-File -Encoding ASCII file.txt  
Get-ChildItem | Out-File-Width 120 unicodeFile.cs
```

重定向操作符：

```
Get-ChildItem > files.txt  
Get-ChildItem 2> errors.txt
```

讨论

Out-File 命令和重定向符有许多相似之处，在大多数情况下可以使用任意一个。重定向符的独特之处在于可以对输出的文件流进行功能强大的控制；Out-File 的长处是可以轻松地处理输出内容的宽度与编码方式。

默认的输出格式的宽度与输出的编码方式是我们在对命令结果重定向时会遇到的两个困难的地方。

默认的输出格式的宽度有时会引起问题，因为重定向 PowerShell 格式化的输出到文件的功能在设计的时候模拟的是用户在屏幕上看到的效果。如果用户的屏幕设成 80 个字符宽，生成的文件也是每行 80 个字符。经过 PowerShell 格式化了输出的例子包括目录的清单（格式化成表格），还有那些使用 Format-* 参数来明确定义输出格式的命令。如果这里产生问题，你可以通过使用 -Width 参数来定义 Out-File 命令的输出格式的宽度。

PowerShell 默认采用 UTF-16 编码来创建所有的文件，有时候就会产生意外的结果。这种编码方式使得 PowerShell 支持所有国家的字符集、cmdlet 和输出，虽然这对传统的 shell 来说是一个极大的进步，但是在对采用 ASCII 编码的文件进行大量的搜索与替换操作的时候可能产生意外的结果。强制地使 PowerShell 在输出结果到文件的时候采用 ASCII 方式编码，可以在 Out-File 命令上加入 -Encoding 参数。

如果要获得更多关于 Out-File 的信息，可以键入 Get-Help Out-File。在附录 A 中有重定向符的完整说明。

参考

- 附录 A 中的“捕获输出”

1.16 向文件的结尾处加入信息

问题

你希望将管道输出重定向到文件中，并且在文件后面加入信息。

解决方案

把一个命令的输出重定向到文件，可以使用 Out-File 命令的参数 -Append，也可以使用附录 A 中描述的重定向操作符。两者都可以实现向文件尾追加文本的功能。Out-File：

```
Get-ChildItem | Out-File -Append files.txt
```

重定向操作符：

```
Get-ChildItem >> files.txt
```

讨论

Out-File 命令和重定向操作符有许多相似之处，在多数情况下，你可以使用任意一个。参考 1.15 节可以获得更多关于两者比较的信息，以及在当前状况下使用哪一个更合适。

参考

- 1.15 节
- 附录 A 中的“捕获输出”

1.17 记录你的会话全文

问题

你想记录当前会话的日志或全文。

解决方案

生成你当前会话的记录，可以运行 Start-Transcript path，path 参数是可选项，用来指定记录的文件名，它基于当前的系统时间。默认的情况下，PowerShell 将这个文件保存到 My Documents 目录下，如果想停止记录，则运行 Stop-Transcript。

讨论

虽然Get-History可以帮助我们查看历史命令，但是不能记录你当前会话的输出。通过使用Start-Transcript命令可以实现记录当前会话中键入的命令与输出的功能。除了前面提到的Path参数之外，Start-Transcript还支持其他参数，让你可以控制PowerShell如何处理输出文件。

1.18 将某一项的属性显示成列表

问题

你有一个项（如错误记录项、目录项或.NET对象），你想将这个对象的详细信息显示成列表。

解决方案

为了显示一个项的详细信息，可以使用管道功能把项输出给Format-List命令。例如，显示系统捕获到的错误，可以键入下面的命令：

```
$currentError = $error[0]
$currentError | Format-List -Force
```

讨论

Format-List命令是PowerShell三种格式化命令中的一个。这三种命令包括Format-Table、Format-List和Format-Wide。Format-List命令获得输入，然后将它以列表方式显示出来。默认情况下，PowerShell从安装目录下的*.format.ps1xml文件中获得要显示的属性。如果想显示所有的属性，可以键入**Format-List ***。有些时候，你键入了**Format-List ***，但是不能获得项的属性列表，这种情况的出现是由于在*.format.ps1xml中定义了项，但是没有定义任何在列表命令中要显示的条目，这种情况下，你可以键入**Format-List -Force**。

获取更多关于Format-List的信息，键入**Get-Help Format-List**。

1.19 将某一项的属性显示成表格

问题

你有一系列的项（如错误记录集、目录集或.NET对象），你想以表格方式显示它们的概要信息。

解决方案

为了显示系列项的概要信息，你可以将它们传给 Format-Table 命令。在 PowerShell 中，这是默认的系列项的显示格式，它提供了几个有用的功能。

为了使用 PowerShell 的默认格式，你可以将命令（如 *Get-Process*）的输出以管道方式输出给 Format-Table，如下所示：

```
Get-Process | Format-Table
```

为了在表格格式中显示特定的属性（如 *Name* 和 *Workingset*），你需要将这些属性的名字作为参数提供给 Format-Table，如下所示：

```
Get-Process | Format-Table Name,WS
```

为了通知 PowerShell 以最易阅读的方式来格式化表格，可以提供 *-Auto* 参数给 Format-Table。PowerShell 将“WS”定义为进程中的工作集的一个别名，如下所示：

```
Get-Process | Format-Table Name,WS -Auto
```

为了定义一个自定义的列（如：以兆为单位显示一个进程的工作集），你可以提供一个自定义的格式化表达式给 Format-Table，如下：

```
$fields = "Name",@{Label = "WS (MB)"; Expression = {$_._WS / 1mb}; Align = "Right"}  
Get-Process | Format-Table $fields -Auto
```

讨论

Format-Table 是 PowerShell 三个格式化的命令之一，这些命令包括 Format-Table、Format-List 和 Format-Wide。Format-Table 获得输入，以表格方式来显示这些输入。默认情况下，PowerShell 从安装目录下的 *.format.ps1xml 文件中获得要显示的属性。你可以通过键入命令 **Format-Table *** 来显示项的所有属性，尽管这是一种较少使用的形式。

通过给 Format-Table 传递参数 -Auto，可以自动使输出的表格尽可能充分地使用屏幕的空间，当然，这需要花费时间。为了计算最佳的表格布局，PowerShell 需要计算输入项中的每个项，对于数量不多的项来说，不会花费很长时间，但是对于包含大量项的情况来说（例如递归目录显示）要花费较多时间。没有 -Auto 参数的时候，Format-Table 会收到一项显示一项；而有 -Auto 参数时，只有收到所有输入项后才会显示输入内容。

关于 Format-Table 最吸引人的功能应该是可以自定义表格的列。你自定义一个表格的列，以同样的方式，你自定义一个列表。你提供一个哈希表，而不是指定项已有的属性。哈希表包括三个值：列的标签、格式化表达式和对齐方式。Format-Table 将标签显示成列的标题，并使用表达式生成列数据。标签必须是字符串格式，表达式必须是脚本块，对齐方式必须是 "Left"、"Center" 或 "Right"。在脚本表达式中，\$_ 变量表示当前的项正在被格式化。

上面最后一个例子中的表达式中，我们设置当前要处理的项，并以 1MB 来作单位。

关于 Format-Table 的详细内容，请输入 **Get-Help Format-Table**。

关于哈希表的详细内容，请参见 11.12 节。

关于脚本块的详细内容，请参见 10.3 节。

参考

- 10.3 节
- 11.12 节

1.20 管理命令的错误输出

问题

你想显示一条命令执行后的详细的错误信息。

解决方案

为了列出当前会话中发生的所有错误，你可以访问 \$error 数组：

```
$error
```

为了列出当前会话中最后发生的错误，需要访问 \$error 数组中的第一个值：

Serrot [0]

为了列出错误的详细信息，将错误以管道方式输出给 Format-List，注意要加上 -Force 参数：

```
$currentError = $error[0]  
$currentError | Format-List -Force
```

为了列出引起错误的命令的详细信息，可以访问 `InvocationInfo` 属性。

```
$currentError = $error[0]  
$currentErrorInvocationInfo
```

为了以更加简洁的基于分类的方式显示错误，修改变量 \$errorView 的值为 "CategoryView"。

```
$errorView = "CategoryView"
```

为了清除 PowerShell 产生的错误，需要调用 \$error 的 Clear() 方法。

```
$error.Clear()
```

讨论

在管理员的工作中，发生错误是常见的事实，并不是所有的错误都会带来严重的后果。因此，PowerShell 将错误分为两类：非中断型和中断型。

非中断型的错误是最常见的错误，它用来指出命令、脚本、函数或管道捕获到一个错误，这个错误可以恢复或继续。一个非中断型错误的例子是Copy-Item，当你从一处往另一处复制文件的时候，如果复制某个文件发生错误，你仍然可以继续完成剩余文件的复制。

中断型的错误，从另一方面来说，表明在操作中发生的比较严重的、底层的错误。一个中断型错误的例子是当使用 Copy-Item 时你输入了无效的命令参数。

关于如何处理这两种错误的详细信息，请参见第 13 章。

参考

- 第13章

1.21 配置调试、校验和处理输出

问题

你想管理详细的调试、验证信息，以及命令与脚本产生的执行过程的输出。

解决方案

为了启用脚本和生成它的命令的调试信息的输出你可以：

```
$debugPreference = "Continue"  
Start-DebugCommand
```

为了启用命令的验证模式，需要加入 -Verbose 参数：

```
Copy-Item c:\temp\*.txt c:\temp\backup\ -Verbose
```

为了禁用脚本或命令生成的进度信息，你可以：

```
$progressPreference = "SilentlyContinue"  
Get-Progress.ps1
```

讨论

除了产生错误输出（如 1.20 节中的描述）之外，许多脚本和命令还生成其他几种输出，包括：

调试输出 (*debug output*)

它可以帮助你诊断问题，提供更细致的关于命令的工作步骤。你可以在脚本中使用 Write-Debug 命令或调用命令的 WriteDebug() 方法输出这类信息。除非你通过 \$host.PrivateData.Debug* 颜色配置变量来自定义输出信息的颜色，否则 PowerShell 会以黄色来显示这类信息。

验证输出 (*verbose output*)

它帮助你更好地监视命令的动作，你可以在脚本中使用 Write-Verbose 命令或调用命令的 WriteVerbose() 方法来输出这类信息。除非你通过 \$host.PrivateData.Verbose* 颜色配置变量来自定义输出信息的颜色，否则 PowerShell 会以黄色来显示这类信息。

进度输出 (*progress output*)

帮助你监视长时间运行的命令的状态，你可以在脚本中使用 Write-Progress 命令或调用命令的 WriteProgress() 方法来输出这类信息。除非你通过 \$host.

`PrivateData.Progress*`颜色配置变量来自定义输出信息的颜色，否则PowerShell会以黄色来显示这类信息。

一些命令只有在分别指定`-Verbose`或`-Debug`参数后才会产生验证或调试信息。为了配置脚本或命令的调试、验证或进度信息，你可以通过修改shell变量`$debugPreference`、`$verbosePreference`和`$progressPreference`来达到目的，这些变量可以接受下面的值：

安静模式 (silentlyContinue)

不显示输出信息。

停止模式 (stop)

将输出视为错误。

继续模式 (continue)

显示输出。

询问模式 (inquire)

显示继续操作信息。

参考

- 1.20 节

1.22 通过附加单元扩展 PowerShell 问题

你想使用第三方编写的 cmdlet 和提供者 (provider)。

解决方案

在 PowerShell 中，把包含扩展 cmdlet 和提供者的扩展程序称为 snapin。作者可以通过安装程序或独立的 PowerShell 配件来分配这些程序。PowerShell 通过配件的名字来识别每个 snapin。

使用 snapin 的步骤：

1. 获得 snapin 的执行文件。

2. 将文件拷贝到安全的地方，因为是可执行文件，最好选取只有读取权限，没有写入权限（比如 Program Files 目录）的文件夹用户。
3. 注册 snapin。在 snapin 所在文件夹下运行命令 `InstallUtil SnapinFilename.dll`，这样本机所有用户就都可以运行 snapin 提供的命令了。你可以在 .NET 框架安装目录下找到文件 `InstallUtil`，一般是 `C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\InstallUtil.exe`。
4. 添加 snapin。在 PowerShell 提示符下（或在你的配置文件中）运行命令 `Add-PsSnapin SnapinIdentifier`，如果想查看所有 snapin 的标识，你可以运行下面命令来查看名称：

```
Get-PsSnapin -Registered
```

5. 使用 snapin 提供的 cmdlet 和提供者。

为了从系统中移出 snapin，可以通过命令 `InstallUtil/u SnapinFilename.dll`。一旦取消了注册，你就可以删除 snapin 相关的文件了。

讨论

对于交互使用的情况（或在配置文件中），`Add-PsSnapin` 是加载独立的 snapin 的最佳方法。当然，如果想加载多个配置好的 snapin，1.23 节提供了另一种可选的方法。

你可以访问网站 <http://www.codeplex.com/PowerShellCX>，那里提供了更多流行的 snapin。

参考

- 1.23 节

1.23 使用控制台文件加载保存 snapin

问题

你想在启动 PowerShell 的时候加载一些 snapin，同时又不想修改你（或他人）的配置文件。

解决方案

一旦你在系统中注册了一个 snapin，你就可以通过在 PowerShell 控制台文件中添加

snapin的标识符的方法来达到加载目的。通过给PowerShell命令提供-PsConsoleFile参数，可以加载控制台文件中定义的snapin到新的会话中。

通过下面命令，将当前加载的snapin保存到控制台文件中：

```
Export-Console Filename.ps1
```

通过下面命令，加载文件 *Filename.ps1* 中定义的 snapin 到 PowerShell 中：

```
PowerShell -PsConsoleFile Filename.ps1
```

讨论

PowerShell 控制台文件采用 XML 格式存储已经安装的 snapin，典型的文件格式如例 1-11 所示。

例 1-11：一个典型的 PowerShell 控制台文件

```
<?xml version="1.0" encoding="utf-8"?>
<PSConsoleFile ConsoleSchemaVersion="1.0">
  <PSVersion>1.0</PSVersion>
  <PSSnapIns>
    <PSSnapIn Name="CustomSnapin" />
    <PSSnapIn Name="SecondSnapin" />
  </PSSnapIns>
</PSConsoleFile>
```

控制台文件的扩展名是 *.ps1*。

一般情况下，通过 PowerShell 的命令参数（在脚本或自动执行任务中）来运行控制台文件，当然，也可以通过双击控制台文件来运行。

关于如何管理 snapin 的更多信息，参见 1.22 节。

参考

- 1.22 节

第2章

管道

2.0 简介

在 Shell 中一个基本的概念称作管道 (pipeline)。它是 PowerShell 中形成表格的最重要的功能之一。管道的概念比较简单，即在一组命令中，输出的一个命令成为下一个输入。管道的概念与工厂中的生产线比较相似：在不同的生产环节进行连续地再加工，如例 2-1 所示。

例 2-1：一个 PowerShell 管道

```
Get-Process | Where-Object { $_.WorkingSet -gt 500kb } | Sort-Object -Descending Name
```

在 PowerShell 中，使用管道符号 (|) 来划分管道中的每个命令。

在例 2-1 中，`Get-Process` 命令返回系统中存在的进程，这些进程对象包含进程的基本信息，如进程名、内存使用情况、进程的 ID 及其他信息。紧接着，`Where-Object` 命令直接对这些进程对象进行处理，简单处理那些占用内存大于 500kb 的进程对象，然后将结果往后传递，`Sort-Object` 对这些进程对象按名称进行降序排列。这个简略的例子说明了管道的强大功能，PowerShell 在管道中传递的是高保真的对象，不仅仅是文本表述。

与之形成对比的是，所有其他 Shell 在管道中传递的是纯文本的数据。它们从纯文本的输出中提取有用的信息并转换成管道中传递的信息。在 `cmd.exe` 中，以传统的基于 UNIX 的 Shell 来实现上面的例子是十分困难的，而且几乎是不可能的。

传统的基于文本的 Shell 使得管道功能变得十分复杂，因为要对在管道中的每个命令的输出格式十分清楚才行，如例 2-2 所示。

例 2-2：一个传统的基于文本的管道

```
lee@trinity:~$ ps -F | awk '{ if($5 > 500) print }' | sort -r -k 64,70
UID      PID  PPID  C    SZ   RSS PSR STIME TTY          TIME CMD
```

```
lee      8175  7967  0   965  1036  0 21:51 pts/0    00:00:00 ps -f
lee      7967  7966  0  1173  2104  0 21:38 pts/0    00:00:00 -bash
```

在这个例子中，你需要了解每一行的内容，如组号 5 表示的是内存的范围；你还需要了解另外一种语言（awk 工具）来过滤列数据；最后，你还要了解包含进程名的列的范围（此系统中的第 64 列到第 70 列），根据它来排序。这仅仅是一个完成简单功能的例子。

基于对象的管道技术展示了巨大的潜力，为系统管理提供了大量更快捷、更高效的途径。

2.1 过滤列表项或命令输出项

问题

你想过滤一个列表或命令输出的项。

解决方案

使用 Where-Object（标准别名 where 和 ?）从列表（或命令的输出）中来选择你要输出的项。

为了列出所有正在运行进程名称中包含“*search*”的进程，对进程名字属性使用 -like 操作符来比较进程的 Name 属性。

```
Get-Process | Where-Object { $_.Name -like "*Search*" }
```

为了列出当前位置的所有子目录，使用 PsIsContainer 属性：

```
Get-ChildItem | Where-Object { $_.PsIsContainer }
```

为了列出所有已经停止的服务，对服务的 Status 属性使用操作符 -eq：

```
Get-Service | Where-Object { $_.Status -eq "Stopped" }
```

讨论

对于输入的每一项（前一个命令的输出），Where-Object 会根据你定义的脚本块对输入进行估算。如果脚本块返回真，Where-Object 会把结果输出；否则，不输出。在 PowerShell 中，一个脚本块是指用 {} 括起来的一系列 PowerShell 命令。在脚本块中可以输入任意的 PowerShell 命令。**在脚本块中，\$_ 变量代表当前输入的对象。**对于输入对象中的每一个项，PowerShell 将它指派给 \$_ 变量，然后运行脚本块。在上面的例子中，输入的对象代表前一个命令生成的进程、文件或服务。

如果需要，在脚本块中可以包含大量的函数，可以进行组合测试、比较或其他工作。关于脚本块的更多信息，参见 10.3 节；关于有效的比较类型，参见附录 A 中的“比较操作符”。

对于简单的过滤，Where-Object 的语法看起来有点繁琐。在 2.2 节中，通过一个脚本可以使简单的过滤（如上面的例子）更容易实现。

对于复杂的过滤（比如，在 Explorer 窗口中用鼠标对文件进行操作），编写脚本块表达你的愿望是困难的，甚至是不可能的。2.3 节展示了一个使手工过滤更容易完成的脚本。

关于 Where-Object 的更多信息，可以键入 Get-Help Where-Object。

参考

- 2.2 节
- 2.3 节
- 10.3 节
- 附录 A 中的“比较操作符”

2.2 编程：简化多数 Where-Object 过滤

Where-Object 命令是如此的强大，以至于允许你基于任意的表达式来过滤你的输出。对于简单的过滤（如只基于一个简单属性的比较操作），语法看起来可能有些难看：

```
Get-Process | Where-Object { $_.Handles -gt 1000 }
```

对于这种情况，一个简单的方法是编写一个脚本文件（如例 2-3 所示），把脚本块中的表达式直接写在脚本中：

```
Get-Process | Compare-Property Handles gt 1000  
Get-ChildItem | Compare-Property PsIsContainer
```

通过简短的别名，可以简化书写：

```
PS >Set-Alias wheres Compare-Property  
PS >Get-ChildItem | wheres Length gt 100
```

例 2-3 实现了这个“简单位置”功能。注意，如果输入了不存在的操作符作为 \$ 符号参数，会生成一个错误信息。

例 2-3: Compare-Property.ps1

```
#####
## Compare-Property.ps1
##
## Compare the property you provide against the input supplied to the script.
## This provides the functionality of simple Where-Object comparisons without
## the syntax required for that cmdlet.
##
## Example:
##   . Get-Process | Compare-Property Handles gt 1000
##   dir | Compare-Property PsIsContainer
#####
param($property, $operator = "eq", $matchText = "$true")

Begin { $expression = "`$_.{$property} -$operator `"$matchText`"" }
Process { if(Invoke-Expression $expression) { $_ } }
```

关于更多运行脚本的信息。

参考

- 1.1 节

2.3 编程：交互式过滤对象

有些时候，对我们来说 Where-Object 提供了太强大的功能，在这种情况下，2.2 节中的 Compare-Property 脚本文件提供了更简单的方法。有些时候，Where-Object 又太简单了，如表达你的选择逻辑的代码十分麻烦，不如手工选择，在这种情况下，交互式的过滤变得更有效率。

例子 2-4 实现了交互式过滤。使用了一些本书没有提到的概念，你可以认为这是个灵巧的脚本。为了了解脚本中你还不能理解的内容，看看脚本中的注释。

例 2-4: Select-FilteredObject.ps1

```
#####
## Select-FilteredObject.ps1
##
## Provides an interactive window to help you select complex sets of objects.
## To do this, it takes all the input from the pipeline, and presents it in a
## notepad window. Keep any lines that represent objects you want to pass
## down the pipeline, delete the rest, then save the file and exit notepad.
##
## The script then passes the original objects that you kept along the
## pipeline.
##
```

```
## Example:  
##     Get-Process | Select-FilteredObject | Stop-Process -WhatIf  
##  
#####  
  
## PowerShell runs your "begin" script block before it passes you any of the  
## items in the pipeline.  
Begin  
{  
    ## Create a temporary file  
    $filename = [System.IO.Path]::GetTempFileName()  
  
    ## Define a header in a "here-string" that explains how to interact with  
    ## the file  
    $header = @"  
#####  
## Keep any lines that represent objects you want to pass  
## down the pipeline, and delete the rest.  
##  
## Once you finish selecting objects, save this file and  
## exit.  
#####  
  
" @  
  
    ## Place the instructions into the file  
    $header > $filename  
  
    ## Initialize the variables that will hold our list of objects, and  
    ## a counter to help us keep track of the objects coming down the  
    ## pipeline  
    $objectList = @()  
    $counter = 0  
}  
  
## PowerShell runs your "process" script block for each item it passes down  
## the pipeline. In this block, the "$_" variable represents the current  
## pipeline object  
process  
{  
    ## Add a line to the file, using PowerShell's format (-f) operator.  
    ## When provided the output of Get-Process, for example, these lines look  
    ## like:  
    ## 30: System.Diagnostics.Process (powershell)  
    ##     "{0}: {1}" -f $counter,$_.ToString() >> $filename  
  
    ## Add the object to the list of objects, and increment our counter.  
    $objectList += $_  
    $counter++  
}  
  
## PowerShell runs your "end" script block once it completes passing all  
## objects down the pipeline.  
end  
{  
    ## Start notepad, then call the process's WaitForExit() method to
```

```
## pause the script until the user exits notepad.
$processStartInfo = New-Object System.Diagnostics.ProcessStartInfo "notepad"
$processStartInfo.Arguments = $filename
$process = [System.Diagnostics.Process]::Start($processStartInfo)
$process.WaitForExit()

## Go over each line of the file
foreach($line in (Get-Content $filename))
{
    ## Check if the line is of the special format: numbers, followed by
    ## a colon, followed by extra text.
    if($line -match "^(\d+?):.*")
    {
        ## If it did match the format, then $matches[1] represents the
        ## number -- a counter into the list of objects we saved during
        ## the "process" section.
        ## So, we output that object from our list of saved objects.
        $objectList[$matches[1]]
    }
}

## Finally, clean up the temporary file.
Remove-Item $filename
}
```

关于更多运行脚本的信息，参见 1.1 节。

参考

- 1.1 节
- 2.2 节

2.4 处理列表或命令输出的每一项

问题

你有一个列表，想处理其中每项。

解决方案

使用 `Foreach-Object` 命令（标准的别名 `foreach` 和 `%`）来处理列表中的项。

为了对列表中的每项进行运算，在脚本块参数中使用 `$_` 变量：

```
PS >1..10 | Foreach-Object { $_ * 2 }
```

```
6  
8  
10  
12  
14  
16  
18  
20
```

为了处理目录中的文件，在脚本块中使用 `$_` 变量作为参数：

```
Get-ChildItem *.txt | Foreach-Object { attrib -r $_ }
```

为了访问列表中每一项的方法或属性，使用 `$_` 变量作为脚本块的参数。

在这个例子中，获得正在运行的记事本程序的进程列表，然后等待它们退出：

```
$notepadProcesses = Get-Process notepad  
$notepadProcesses | Foreach-Object { $_.WaitForExit() }
```

讨论

与 `Where-Object` 相似，`Foreach-Object` 对输入项中的每一项执行脚本块中的命令。在 PowerShell 中，一个脚本块是指用 {} 括起来的一系列命令，针对输入对象中的每个项，PowerShell 将他指派给 `$_` 变量，每次一个元素。在本案例中，`$_` 变量代表的是前一个命令生成的每个文件或进程。

在脚本块中，如果需要，可以包含大量的函数，可以进行组合测试、比较或其他工作。关于脚本块的更多信息，参见 10.3 “编写一个脚本块”；关于有效的比较类型的更多信息，参见附录 A 中的“比较操作符”。

注意：第一个例子中，演示了一个简洁的方法生成一组数字：

```
1..10
```

这是 PowerShell 声明数组的语法，你可以通过 11.3 “访问数组中的元素” 来了解更多关于数组的信息。

`Foreach-Object` 并不是针对列表中的项进行操作的唯一方法，PowerShell 脚本语言还支持其他一些关键字，如 `for`, `foreach`, `do` 和 `while`。关于如何使用这些关键字，可以参见 4.4 节。

关于 `Foreach-Object` 更多的信息，可以键入 `Get-Help Foreach-Object`。

关于如何处理脚本中的管道输入，函数以及脚本块，参见 10.7 节。

参考

- 4.4 节
 - 10.3 节
 - 10.7 节
 - 11.3 节
 - 附录 A 中的“比较操作符”

2.5 自动化数据密集型任务

问题

你想针对大量的数据调用一个简单的任务。

解决方案

如果仅仅是一部分数据不同（如服务器名字或用户名），可以将它们存储到一个文本文件。使用Get-Content来读取数据，然后使用Foreach-Object（标准的别名foreach或%）来操作列表中的每一项。例 2-5 举例说明这种技术。

例 2-5：使用文本文件中信息自动化数据密集型

```
PS >Get-Content servers.txt
SERVER1
SERVER2
PS >$computers = Get-Content servers.txt
PS >$computers | Foreach-Object { Get-WmiObject Win32_OperatingSystem -Computer $_ }

SystemDirectory : C:\WINDOWS\system32
Organization     :
BuildNumber      : 2600
Version          : 5.1.2600

SystemDirectory : C:\WINDOWS\system32
Organization     :
BuildNumber      : 2600
Version          : 5.1.2600
```

如果任务变得复杂或不清晰，不能确定针对 Foreach-Object 命令中每一项的操作，可以使用 foreach 关键字来实现任务，如例 2-6 所示。

例 2-6：使用 foreach 关键字使得循环语句容易阅读

```
$computers = Get-Content servers.txt
```

```
foreach($computer in $computers)
{
    ## Get the information about the operating system from WMI
    $system = Get-WmiObject Win32_OperatingSystem -Computer $computer

    ## Determine if it is running Windows XP
    if($system.Version -eq "5.1.2600")
    {
        "$computer is running Windows XP"
    }
}
```

如果针对每个任务有多个数据需要变化（如针对大批的计算机的 WMI 类和计算机的名字），创建一个 CSV 文件，为每个任务创建一行。使用 `Import-Csv` 将数据导入 PowerShell，然后使用结果对象的属性作为相关数据的源。例 2-7 演示这个技术：

例 2-7：使用 CSV 文件中的信息来自动化数据密集型任务

```
PS >Get-Content WmiReport.csv
ComputerName,Class
LEE-DESK,Win32_OperatingSystem
LEE-DESK,Win32_Bios
PS >$data = Import-Csv WmiReport.csv
PS >$data

ComputerName          Class
-----              -----
LEE-DESK             Win32_OperatingSystem
LEE-DESK             Win32_Bios

PS >$data |
>>     Foreach-Object { Get-WmiObject $_.Class -Computer $_.ComputerName }
>>

SystemDirectory : C:\WINDOWS\system32
Organization    :
BuildNumber     : 2600
Version         : 5.1.2600

SMBIOSBIOSVersion : ASUS A7N8X Deluxe ACPI BIOS Rev 1009
Manufacturer    : Phoenix Technologies, LTD
Name            : Phoenix - AwardBIOS v6.00PG
SerialNumber    : xxxxxxxxxxxx
Version         : Nvidia - 42302e31
```

讨论

PowerShell 一个主要的优点是具有自动化重复性任务的能力。有时候，这些重复性的任务是动作密集的（如系统维护，通过注册表到文件清理），并且包括复杂的有顺序的互

相调用的命令。在这种情况下，可以通过编写脚本将这些操作组合到一起，这样可以节省时间，降低错误的发生。

其他时候，你可能需要完成一个简单的任务（例如，通过 WMI 进行查询），但是需要重复地调用这个任务，产生大量的数据。这种情况下，PowerShell 脚本声明、管道的支持，以及数据管理的命令可以帮助完成这些任务。

本解决方案中给出的选项是 Import-Csv。Import-Csv 读取 CSV 文件，为每一行自动创建对象，并将列的名字作为对象的属性。例 2-8 演示了导入 CSV 文件的结果，包含 ComputerName 和 Class 标题。

例 2-8：Import-Csv 创建对象，包括属性 ComputerName 和 Class

```
PS >$data = Import-Csv WmiReport.csv
PS >$data

ComputerName          Class
-----              -----
LEE-DESK             Win32_OperatingSystem
LEE-DESK             Win32_Bios

PS >
PS >$data[0].ComputerName
LEE-DESK
```

如解决方案中所示，你可以使用 Foreach-Object 从这些对象中提供数据来进行重复调用。通过指定参数名字，与相应的数据（从当前的 CSV 对象的属性中获得）连接起来。

这是最常见的解决方案，当对象的属性有相同的名字，许多 cmdlet 的参数可以自动从输入对象中获取这些值。这样可以省略 Foreach-Object 和属性映射的步骤。支持这种功能的参数称作通过属性名支持的管道值。Move-Item 就是这样一个例子，接收管道输入行，如例 2-9 所示：

例 2-9：Move-Item 帮助信息显示了一个通过属性名支持管道值的参数

```
PS >Get-Help Move-Item -Full
(...)

PARAMETERS
    -path <string[]>
        Specifies the path to the current location of the items. The default
        is the current directory. Wildcards are permitted.

    Required?          true
    Position?         1
    Default value     <current location>
    Accept pipeline input? true (ByValue, ByPropertyName)
    Accept wildcard characters? true
```

-destination <string>
Specifies the path to the location where the items are being moved.
The default is the current directory. Wildcards are permitted, but
the result must specify a single location.

To rename the item being moved, specify a new name in the value of
Destination.

Required?	false
Position?	2
Default value	<current location>
Accept pipeline input?	true (ByPropertyName)
Accept wildcard characters?	True
(...)	

如果你命名 CSV 文件中的列的名称，使之与管道传来的参数名一致，PowerShell 可以实现这种部分或全部参数的映射，如例 2-10 所示。

例 2-10：使用 Import-Csv 来使 cmdlet 按属性的名称从管道中接收值

```
PS >Get-Content ItemMoves.csv
Path, Destination
test.txt, Test1Directory
test2.txt, Test2Directory
PS >dir test.txt,test2.txt | Select Name
```

Name

test.txt
test2.txt

```
PS >Import-Csv ItemMoves.csv | Move-Item
```

```
PS >dir Test1Directory | Select Name
```

Name

test.txt

```
PS >dir Test2Directory | Select Name
```

Name

test2.txt

关于 Foreach-Object 和 foreach 脚本关键字的更多的信息，参见 2.4 节。关于如何使用 csv 文件的更多信息，参见 8.6 节。关于如何使用 WMI 的更多信息，可以参见 15.1 节。

參考

- 2.4 节
 - 8.6 节
 - 15.1 节

卷之三

食譜

上，方整圓滑下，TAN 滅點或縮減，有內齒音體。而以：堅持不拔，如：堅忍，堅強，堅固，堅韌。

黑虎山要地被燒燬後，則伯靈勢，明陞，則凌雲勢，則平陽勢。

（出事後被逮關押，直至1949年1月2日被槍斃於北平）

新嘉坡中華總船頭

卷之三

資料來源：財政部，〈民國九十二年政府總預算案〉。

第3章

变量与对象

3.0 简介

正如第2章提到的，PowerShell通过管道传递，使用内置的信息格式：对象，使得信息管理变的非常容易。使用传统的shell程序时，用户花费了大量的时间理解shell生成的名文信息，通过工具软件，减轻了这些工作，但是还是不尽人意。

PowerShell是构建在.NET框架之上的，所有的内在信息都是遵循.NET对象的格式，将信息和相关的函数封装。

假设你想获得当前系统中运行进程的列表，在其他shell下，通过命令（如tlist.exe或/bin/ps）生成了当前系统的名文的进程的报告。为了进一步处理输出结果，你可能使用一些文本处理软件对结果进行处理，幸运的话，会得到你想要的结果。

PowerShell通过Get-Process生成当前系统中运行的进程列表。与其他shell不同的是，这些列表是.NET框架下的System.Diagnostics.Process对象，.NET框架的文档中是这样描述这个对象的：“访问本地或远程的进程，使你可以开始或停止本地的系统进程”。有了这些对象，你可以访问他们的属性（如进程的名称或内存使用范围），使用对象的函数（如停止、开始和等待退出）。

3.1 在变量中存储信息

问题

你希望把一个管道或命令的输出保存下来供后续使用，或进一步操作。

解决方案

为了保存输出供后续使用，可以把它存储在一个变量中，然后你可以通过变量访问它，甚至可以将它输出给管道，把它当作原始命令的输出：

```
PS >$result = 2 + 2
PS >$result
4
PS >$processes = Get-Process
PS >$processes.Count
85
PS >$processes | Where-Object { $_.ID -eq 0 }

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

讨论

PowerShell 中的变量（其他脚本语言和编程语言同理）可以让你存储一些命令的输出，供你在后续的过程中使用。一个变量使用美元符号 \$ 作为开始，几乎可以使用任意字符作为变量的名称。在 PowerShell 中，一部分的字符代表特殊的意义，所以需要通过一些途径命名。

你可以保存任意的管道或命令的结果到变量中供后续使用。如果命令生成的是简单的数据（如一个数字或字符串），那么变量中包含简单数据；如果命令生成的是丰富的数据（如 Get-Process 返回的代表系统进程的对象），那么变量中包含丰富数据的列表；如果命令生成的是文本信息（如传统的可执行文件产生的输出），那么变量中包含简单的文本信息。

注意：如果你存储了大量的数据到一个变量中，现在不再需要这些数据了，可以给变量赋值 \$null（或其他值），PowerShell 可以释放变量占用的内存空间。

关于 PowerShell 变量的更多的语法和类型信息，参见附录 A 中的“变量”。

除了你创建的变量，PowerShell 自动定义了几个变量表明一些信息，如配置文件的路径，PowerShell 进程的 ID 等。关于这些自动创建的变量的完整的信息，参见附录 C。

参考

- 附录 A 中的“变量”
- 附录 C 中的“PowerShell 自定义变量”

3.2 访问环境变量

问题

你想在脚本或当前会话中使用一个环境变量（如系统路径或当前用户名）

解决方案

PowerShell 提供了几种方式来访问环境变量。

为了获得所有环境变量，列出 env 驱动器的所有子节点：

```
Get-ChildItem env:
```

为了获得某一个环境变量，可以采用简洁的语法，将他的名字前面加上 \$env::：

```
$env:variablename
```

i.e.: \$env:username

为了通过他的提供者路径获得一个环境变量，将 env: 或 Environment:: 提供给 Get-ChildItem：

```
Get-ChildItem env:variablename  
Get-ChildItem Environment::variablename
```

讨论

PowerShell 通过环境提供者 (environment provider) 让我们可以访问环境变量。提供者让你使用数据的存储（如注册表、环境变量和别名），就像访问文件系统一样。

默认情况下，PowerShell 创建一个驱动器（称作 env），与 *environment provider* 来打交道，使你可以访问环境变量。环境提供者让你访问 env: 驱动器中的项，就像访问其他驱动器一样，可以使用 dir env:\variablename 或 dir env:variablename。如果你想直接访问提供者（不是通过驱动器），你也可以键入 dir Environment::variablename。

当然，最常见的（最简单的）使用环境变量的方法是键入 \$env:variablename，这种方法支持任何提供者，但最典型的还是应用于环境变量。

这是因为环境提供者与其他提供者（支持 *-Content 的）共享一些通用的语法，名称上为核心命令 *-Content 集提供支持（参见例 3-1）。

例 3-1：与不同的提供者打交道

```
PS >"hello world" > test
PS >Get-Content c:test
hello world
PS >Get-Content variable:ErrorActionPreference
Continue
PS >Get-Content function:more
param([string[]]$paths); if(($paths -ne $null) -and ($paths.length -ne 0)) { ...
    Get-Content $local:file | Out-Host -p     } } else { $input | Out-Host ...
PS >Get-Content env:systemroot
C:\WINDOWS
```

对于那些支持内容的 cmdlets，PowerShell 提供了一个特殊的变量语法使你可以和这些内容进行交互（参见例 3-2）。

例 3-2：使用 PowerShell 特殊变量语法访问内容

```
PS >$function:more
param([string[]]$paths); if(($paths -ne $null) -and ($paths.length -ne 0)) { ...
    Get-Content $local:file | Out-Host -p     } } else { $input | Out-Host ...
PS >$variable:ErrorActionPreference
Continue
PS >$c:test
hello world
PS >$env:systemroot
C:\WINDOWS
```

这种内容管理的变量语法可以让你设置或获得内容：

```
PS >$function:more = { $input | less.exe }
PS >$function:more
$input | less.exe
```

当使用这种方法访问复杂的提供者路径时，你会遇到命名的问题（即使文件存在）：

```
PS >$c:\temp\test.txt
Unexpected token '\temp\test.txt' in expression or statement.
At line:1 char:17
+ $c:\temp\test.txt <<<
```

以上问题的解决依赖于 PowerShell 中的对复杂变量命名提供转义支持。为了定义这种复杂的变量名，可以将变量名用大括号括起来：

```
PS >${1234123!@#$!@#$12$!@#$@!} = "Crazy Variable!"
PS >${1234123!@#$!@#$12$!@#$@!}
Crazy Variable!
PS >dir variable:\1*
Name                           Value
----                           -----
1234123!@#$!@#$12$!@#$@!      Crazy Variable!
```

访问内容等效的方法（假设文件存在）：

```
PS >${c:\temp\test.txt}  
hello world
```

如果环境变量的名字中不包含特殊的字符，`Get-Content - 变量语法 (variable syntax)` 是最佳（最简单）的方式来访问环境变量。

关于更多使用 PowerShell 变量的信息，参见附录 A。关于更过使用环境变量的信息，可以键入 `Get-Help About_Environment_Variable`。

参考

- 附录 A 中的“变量”

3.3 控制访问和变量的范围与其他项问题

你想控制如何定义变量、别名、函数和驱动器的可用范围。

解决方案

PowerShell 提供了几种方式访问变量。

为了在指定的范围内创建变量，在变量的名称前面提供 `SCOPE` 关键字：

```
$SCOPE:variable = value
```

为了访问指定范围内的变量，在变量的前面提供范围：

```
$SCOPE:variable
```

为了创建变量，在脚本退出后仍然生效，在变量前提供 `GLOBAL` 关键字：

```
$GLOBAL:variable = value
```

为了从一个函数内修改脚本块内的变量范围，在变量前提供 `SCRIPT` 关键字：

```
$SCRIPT:variable = value
```

讨论

PowerShell 通过定义范围的机制来控制对变量、函数、别名和驱动器的访问，一个项的

范围的另一个说法是他的可见度。你总是在一个范围内（称作当前或本地范围），但是
一些操作会改变这些范围。

当你的代码进入到一个嵌套的提示、脚本、函数或脚本块的时候，PowerShell 创建一个
新的范围，这个范围变成本地的范围。这时候，PowerShell 会记住旧的范围和新的范围
之间的关系，从新的范围角度来说，旧的范围被称作父范围；从旧的范围角度来说，新的
范围被称作子范围。子范围可以访问父范围内所有变量，但是在子范围内对那些变量
的修改不会影响到父范围内的变量版本。

注意：尝试在函数内改变脚本域变量是一个难题，因为一个函数本身是一个新的范围，如前面提
到的，在子范围内（函数）改变变量不会影响到父范围（脚本）。以下我们讨论一些在整个
脚本范围内改变变量值的方法。

当你的代码退出一个嵌套的提示、脚本、函数或脚本块的时候，PowerShell 移走旧的范
围，然后变更本地范围到原始的范围，即原范围的父范围。

一些范围经常使用，PowerShell 给它们特定的命名：

Global

全局范围，在此范围内的变量对其他范围来说都是可见的

Script

代表当前脚本范围，此范围内的变量对于其他脚本内的范围都是可见的

Local

当前范围

当你定义了变量的范围，PowerShell 支持两个附加的范围修饰选项：`Private` 和
`AllScope`。当你定义一个属于 `Private` 范围的变量，PowerShell 不允许他直接对子范
围有效。这个变量对子范围是可见的，子范围可以使用 `Get-Variable` 的 `-Scope` 参数
来从父范围获得变量。当你定义了一个属于 `AllScope` 的变量（通过 `*-Variable`、
`*Alias` 或者 `*-Drive`），子范围修改变量也会影响父范围。

在这种情况下，PowerShell 提供了几种方式，让你可以控制访问和变量与其他项的范围。

变量

为了在特定范围内定义一个变量（或者访问特定范围的变量），在变量前面加上范围名
称，如：

```
$SCRIPT:myVariable = value
```

如附录 A 中“变量”所示，变量集合允许你通过 -Scope 参数来指定范围。

函数

为了在特定范围内定义一个函数（或者访问特定范围的函数），在创建函数的时候指定范围名称，如下所示：

```
function $GLOBAL:MyFunction { ... }  
GLOBAL:MyFunction args
```

别名和驱动器

为了在特定范围内定义一个别名或驱动器，使用可选参数 `*-Alias` 或 `*-Drive`；为了访问特定范围的别名或驱动器，可以使用命令的范围参数 `*-Alias` 或 `*-Drive`。

关于范围的更多信息，键入 **Get-Help About-Scope**。

参考

- 附录 A 中的“变量”

3.4 使用 .NET 对象

问题

你想使用 PowerShell 提供的内在支持的，功能强大的 .NET 对象。

解决方案

PowerShell 提供了几种方式来访问对象的方法（包括静态的和实例化的）和属性。

为了调用一个类的静态的方法，用中括号将类的名称括起来，然后输入两个冒号把类名和方法名分开，然后是方法的名称、参数等。

```
[ClassName]::MethodName(parameter list)
```

为了调用对象的一个方法，在代表对象的变量和方法名之间加入圆点符号，如下所示：

```
$objectReference.MethodName(parameter list)
```

为了访问类的静态的属性，用中括号将类的名称括起来，然后输入两个冒号，然后是属性的名称，如下所示：

[ClassName]::PropertyName

为了访问对象的属性，在代表对象的变量和属性名之间加入圆点符号，如下所示：

\$objectReference.PropertyName

讨论

PowerShell 对庞大的.NET 框架的兼容能力使得他能够完成那些令人难以置信的系统维护与应用程序开发的任务。.NET 框架包含了十分庞大的类，每个类覆盖特定的内容，把相关的函数和信息紧密地结合在一起。使用.NET 框架是 PowerShell 一个方面，对管理外壳的领域介绍出一个重大的变革。

一个.NET 框架的类的例子是 System.Diagnostics.Process，提供一组功能，“可以访问本地或远程的进程，用来开始或停止本地系统进程”。

注意：术语 *type* 和 *class* 经常可以互换使用。

类包含方法（允许进行操作）和属性（允许访问信息）。

如，Get-Process 生成 System.Diagnostics.Process 对象，不是传统 shell 生成的文本报告。管理这些进程变得如此的容易，因为它们包含丰富的信息（属性）和操作（方法），你不再需要在从文本流中对进程的 ID 进行分析了，你可以直接与进程对象打交道。

```
PS >$process = Get-Process Notepad
PS >$process.Id
3872
```

静态方法

[ClassName]::MethodName(parameter list)

有些方法只适用于类所表示的概念。比如，检索一个系统中所有运行的进程涉及到一般的进程的概念，而不是特定的进程。所谓的静态方法（*Static methods*）是指应用于类/类型的方法。

例如：

```
PS >[System.Diagnostics.Process]::GetProcessById(0)
```

使用 Get-Process 可以更好的完成这个任务，这里只是为了演示 PowerShell 的调用.NET 类的方法的能力，通过调用 System.Diagnostics.Process 类的静态方法 GetProcessById 获得 ID 是 0 的进程，生成下面的输出：

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

实例的方法

```
$objectReference.MethodName(parameter list)
```

一些方法涉及到具体的实在的类（称为实例），一个例子是停止当前系统中运行的进程，有别于一般概念的进程。如果 \$objectReference 代表的是特定的 System.Diagnostics.Process (如 GetPorcess 返回的进程)，你可以调用方法开始、停止或等待退出。所谓实例的方法是指作用于类的实例的方法。

注意：术语“对象”和“实例”经常可以互换。

例如：

```
PS >$process = GetProcess Notepad
PS >$process.WaitForExit()
```

保存 ID 等于零的进程到 \$process 变量，然后调用该进程对象的 WaitForExit() 方法来暂停 PowerShell 直到进程结束。

注意：要了解一个给定的方法支持的不同的参数（又称重载），键入方法不带任何参数，可以返回该方法的使用说明：

```
PS >$now = Get-Date
PS >$now.AddDays

MemberType      : Method
OverloadDefinitions : {System.DateTime AddDays(Double value)}
TypeNameOfValue   : System.Management.Automation.PSMETHOD
Value           : System.DateTime AddDays(Double value)
Name            : AddDays
IsInstance      : True
```

静态属性

```
[ClassName]::PropertyName
```

或

```
[ClassName]::PropertyName = value
```

用来设置类的静态属性。如果类没有静态属性，尝试使用此语法将引发错误。

与静态方法相似，一些属性只与类代表的概念信息有关系，如 System.DateTime 类“代表着一种瞬间的时候，通常表示为一个日期和一天的时间”。他提供了 NOW 静态属性，返回当前的时间：

```
PS >[System.DateTime]::Now  
Saturday, June 2, 2007 4:57:20 PM
```

使用 Get-Date 可以更好的完成这个任务，这里只是为了演示 PowerShell 的调用 .NET 类的方法的能力。

虽然比较少见，一些类允许你设置静态属性的值，如 [System.Environment]::CurrentDirectory 属性，这个属性代表进程的当前目录，也就是 PowerShell 的启动目录，有别于你在提示符中看到的路径。

实例属性

```
$objectReference.PropertyName
```

或

```
$objectReference.PropertyName = value
```

与实例方法相似，一些属性涉及到具体的实在的类（称为实例），一个这样的例子是一天中一个实际的瞬间时间，有别于一般概念的日期和时间。如果 \$objectReference 指的是一个特定的 System.DateTime（比如：Get-Date 输出或 [System.DateTime]::Now），你可能要获得他所在的星期、天或月。所谓的实例属性是指返回一个类的实例的信息。

比如：

```
PS >$today = Get-Date  
PS >$today.DayOfWeek  
Saturday
```

这个例子把当前日期保存到 \$today 变量中，然后调用实例属性 DayOfWeek 返回日期对应的星期。

有了这些了解，后面的问题就是“如何学习 .NET 框架中可用的函数”和“如何学习一个对象是什么”。

对于第一个问题，参见附录 E，了解那些对系统管理员来说非常有用的类；对于第二个问题，参见 3.9 节和 3.10 节。

参考

- 3.9 节
- 3.10 节
- 附录 E “选择 .NET 类使用它们”

3.5 创建一个 .NET 对象的实例

问题

你想创建一个 .NET 对象的实例，并使用他的方法与属性。

解决方案

使用 New-Object 来创建一个对象的实例。

为了使用默认的构造函数创建一个对象的实例，使用 New-Object 加上类名作为参数。

```
PS >$generator = New-Object System.Random
PS >$generator.NextDouble()
0.853699042859347
```

为了使用带参数构造函数创建一个对象的实例，把参数传递给 New-Object。有些时候，要实例的类保存在独立的库文件中，PowerShell 没有默认加载，比如 System.Windows.Forms，这个时候，你必须首先加载库文件，然后再实例类：

```
[Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")
$image = New-Object System.Drawing.Bitmap source.gif
$image.Save("source_converted.jpg", "JPEG")
```

为了在创建对象的同时使用它（而不是先保存再使用），用括号把 New-Object 和要实例的类括起来，如：

```
PS >(New-Object Net.WebClient).DownloadString("http://live.com")
```

讨论

许多命令（如 Get-Process 和 Get-ChildItem）可以生成实在的 .NET 对象，表示进程、文件和目录。但是 PowerShell 支持更多 .NET 框架下的对象，而不仅仅是命令生成的对象。这些 .NET 框架下额外的新增对象提供了巨大的函数功能，你可以在脚本或系统管理的任务中来使用。

当你要使用这些类的时候，第一步是创建类的一个实例，把实例保存到一个变量中，然后使用该实例提供的方法和属性。为了创建一个实例，使用 New-Object。New-Object 需要的第一个参数是类的名称，第二个参数是构造函数的参数，如果有的话，New-Object 支持 PowerShell 的类型快捷方式，你可以不必使用完整的类名，关于类型快捷方式的更多信息，参见 3.4 节。

既然第二个参数传递的是类的构造函数参数，当你试图传递的参数本身是一个数组，你可能会遇到麻烦。假设 \$byte 是一个字节的数组：

```
PS >$memoryStream = New-Object System.IO.MemoryStream $bytes
New-Object : Cannot find an overload for ".ctor" and the argument count: "11".
At line:1 char:27
+ $memoryStream = New-Object <<< System.IO.MemoryStream $bytes
```

为了解决这个问题，定义一个数组来包含数组

```
PS >$parameters = , $bytes
PS >$memoryStream = New-Object System.IO.MemoryStream $parameters
```

或者

```
PS >$memoryStream = New-Object System.IO.MemoryStream @($bytes)
```

从另外的文件中加载类

默认情况下，PowerShell 会加载最常用的类型。当然，还有一些类需要你加载对应的库文件后才可用。你可以从 MSDN 中查找与类对应的库文件。

为了加载一个类库，使用 System.Reflection.Assembly 类提供的方法来加载

```
PS >[Reflection.Assembly]::LoadWithPartialName("System.Web")
GAC      Version      Location
---      ---       ---
True    v2.0.50727  C:\WINDOWS\assembly\GAC_32\(...)\System.Web.dll

PS >[Web.HttpUtility]::UrlEncode("http://search.msn.com")
http%3a%2f%2fsearch.msn.com
```

注意： LoadWithPartialName 方法不适合在脚本或产品发布的环境中使用，他加载最新的库文件，可能与你用来开发的版本不一致。比较安全的加载库文件的方法是使用 [Reflection.Assembly]::Load() 加上完整的名称。

参见附录 E 为了了解那些对系统管理员来说非常有用的类。为了了解更多类支持的函数的信息，参见 3.9 节。

关于 New-Object 更多信息，可以键入 **Get-Help New-Object**。

参考

- 3.4 节
- 3.10 节
- 附录 E “选择 .NET 类使用它们”

3.6 编程：创建对象的实例

当使用 .NET 框架的时候，你经常会使用到这样一些类：它们的主要的职责是管理其他的对象。比如，`System.Collections.ArrayList` 类可以用来管理动态的对象列表，你可以在队列中加入对象，移走对象，进行排序等操作。你可以向队列中插入任意类型的对象，字符串、整数、日期对象等。当然，使用这种支持任意对象的类有时会产生问题，其中之一就是类型安全问题：如果你偶尔将一个字符串插入整数的队列，直到程序运行失败时才会发现错误。

如果我们只是在 PowerShell 中使用，这个问题变得无意义。在强型的语言（如 C#）中，一个更常见的问题是，当你再次使用某个对象的时候，要提醒系统（通过明确声明）该对象的类型，如下所示：

```
// This is C# code
System.Collections.ArrayList list =
    new System.Collections.ArrayList();
list.Add("Hello World");

string result = (String) list[0];
```

为了解决上述问题，.NET 框架引入了一个称为泛型的概念：一种支持任意类型的对象的类，允许你指定对象的类型，在下面的例子中，定义了字符串的集合：

```
// This is C# code
System.Collections.ObjectModel.Collection<String> list =
    new System.Collections.ObjectModel.Collection<String>();
list.Add("Hello World");

string result = list[0];
```

尽管 New-Object 命令功能强大，但是目前还不是很方便处理创建泛型。对于一个简单的泛型，可以参照 .NET 框架的语法来创建，如下：

```
$coll=New-Object 'System.Collections.ObjectModel.Collection`1[System.String]'
```

然而，当你试图定义系统提供类型之外的类型，或者创建复杂的泛型（如：互相引用的泛型）的时候，上面的语句会失败。

例 3-3 使创建泛型变得容易。

例 3-3: New-GenericObject.ps1

```
#####
## New-GenericObject.ps1
##
## Creates an object of a generic type:
##
## Usage:
##       # Simple generic collection
##       New-GenericObject System.Collections.ObjectModel.Collection`1[System.Int32]
##
##       # Generic dictionary with two types
##       New-GenericObject System.Collections.Generic.Dictionary`2[System.String,System.Int32]
##
##       # Generic list as the second type to a generic dictionary
##       $secondType = New-GenericObject System.Collections.Generic.List`1[Int32]
##       New-GenericObject System.Collections.Dictionary`1[System.String,$secondType.GetType()]
##
##       # Generic type with a non-default constructor
##       New-GenericObject System.Collections.Generic.LinkedListNode`1[System.String] "Hi"
##
```

3.7 快速输入较长的类名

问题

当你在脚本文件使用具有较长命名的类的时候，希望减少输入多余的信息。

解决方案

为了减少输入静态方法的名称，可以把类名保存到一个变量中，如下：

```
$math = [System.Math]
$math::Min(1,10)
$math::Max(1,10)
```

为了减少在同一命名空间的对象名的输入，你可以使用 -f 操作符，如下：

```
$namespace = "System.Collections.{0}"
$arrayList = New-Object ($namespace -f "ArrayList")
$queue = New-Object ($namespace -f "Queue")
```

为了减少在同一命名空间的多个类型的静态方法名的输入，也使用-f 操作符，如下：

```
$namespace = "System.Diagnostics.{0}"
([Type] ($namespace -f "EventLog"))::GetEventLogs()
([Type] ($namespace -f "Process"))::GetCurrentProcess()
```

讨论

值得注意的是，当你在使用一些.NET 的类（或来自第三方 SDK 的类）的时候，指定它们完整的类名会变得麻烦。比如，在.NET 框架下，许多有用的集合类都是以“System.Collections”开始，我们把这称为类的命名空间 (*namespaces*)。大多数的编程语言都通过using指示来引入命名空间，你可以在程序中通过using来引入多个命名空间，这样，当你在输入类似“ArrayList”的类名称的时候，系统会自动到你引入的命名空间中寻找这个类。PowerShell 不支持 using 指示，但是可以通过其他方式达到相同的效果。

如果你重复使用特定的类的静态的方法，可以把类名保存到变量来减少输入，如下：

```
$math = [System.Math]
$math::Min(1,10)
$math::Max(1,10)
```

如果你是用同一个命名空间来创建不同类的实例，可以把命名空间保存到变量中，然后使用-f 操作符来指定唯一的类名，如下：

```
$namespace = "System.Collections.{0}"
$arrayList = New-Object ($namespace -f "ArrayList")
$queue = New-Object ($namespace -f "Queue")
```

如果你是用一个命名空间下的几个类来调用其静态方法，可以把命名空间保存到变量中，然后使用-f 操作符来指定唯一的类名，然后传递给 type，如下：

```
$namespace = "System.Diagnostics.{0}"
([Type] ($namespace -f "EventLog"))::GetEventLogs()
([Type] ($namespace -f "Process"))::GetCurrentProcess()
```

关于 PowerShell 格式化操作符的更多信息，参见 5.6 节。

参考

- 5.6 节

3.8 使用 COM 对象

问题

你想创建一个 COM 对象，使用他的方法与属性。

解决方案

使用 `New-Object` 命令（加上 `-ComObject` 参数），通过 `ProgID` 来创建一个 COM 对象，然后可以像使用 PowerShell 中其他对象那样，调用他的方法和属性。

```
$object = New-Object -ComObject ProgId
```

比如：

```
PS >$sapi = New-Object -Com Sapi.SpVoice  
PS >$sapi.Speak("Hello World")
```

讨论

历史遗留的问题是，许多应用程序暴露它们的脚本和管理的接口为 COM 对象，虽然 .NET APIs（和 PowerShell 命令）变得越来越流行，但是还是有一些管理的任务需要调用 COM 对象。

与 .NET 框架中的类不同，了解如何使用 COM 对象来完成系统管理的任务是困难的。参见附录 G 和它们的使用，你可以查阅对于系统管理员来说最常用的 COM 对象。

关于 `New-Object` 命令的更多的信息，键入 `Get-help New-Object`。

参考

- 附录 G “选择的 COM 对象和它们的使用”

3.9 了解类型和对象

问题

你创建了一个对象的实例，希望了解对象提供了哪些方法，支持哪些属性。

解决方案

比较通用的了解一个对象和属性的方法是通过 Get-Member 命令。

为了获得保存到变量 \$object 中的一个对象的实例成员，这里把 \$object 变量用管道输出到 Get-Member cmdlet，如下：

```
$object | Get-Member  
Get-Member -InputObject $object
```

为了得到保存到变量 \$object 中一个对象的静态成员，需要把它用管道输出到 Get-Member 命令，同时提供 -Static 标志，如下：

```
$object | Get-Member -Static  
Get-Member -Static -InputObject $object
```

为了得到一个特定类的静态成员，需要把他用管道输出给 Get-Member cmdlet，同时也提供 -Static 标志，如下：

```
[Type] | Get-Member -Static  
Get-Member -InputObject [Type]
```

为了得到保存到变量 \$object 一个对象的指定类型（如方法、属性）的成员，需要把成员的类型传递给 -MemberType 参数，如下：

```
$object | Get-Member -MemberType memberType  
Get-Member -MemberType memberType -InputObject $object
```

讨论

在 Windows PowerShell 中有三个命令是会经常用到的，其中之一就是 Get-Member 命令，另外两个是 Get-Command 和 Get-Help。

如果你通过管道向 Get-Member 传递了一个对象的集合（如队列），PowerShell 会从集合中将它们提出，然后逐个传给 Get-Member，Get-Member 会返回每个对象的成员。当你试图了解集合类本身的成员或属性的时候，会花费大量的时间并产生问题。

如果你希望了解一个集合的属性（不是他包含的元素），需要提供 -InputObject 参数给集合。

有时候，你可能会将集合打包到数组当中（使用，号），这样当 Get-Member 在拆开数组的时候，集合对象仍然存在，如下：

```
PS >$files = Get-ChildItem  
PS >,$files | Get-Member
```

TypeName: System.Object[]		
Name	MemberType	Definition
Count	AliasProperty	Count = Length
Address	Method	System.Object& Address(Int32)
(...)		

关于类型和对象的更详细的信息，可以参见下一章节，3.10节。

关于 Get-Member 更多信息，键入 **Get-Help Get-Member**。

参考

- 3.10 节

3.10 获得类和对象详细文档

问题

你有一类对象，你想了解该对象的方法、属性的详细地信息。

解决方案

通过 .NET 框架的在线帮助文档——MSDN（见 <http://msdn.microsoft.com>），是了解一个对象支持的方法、属性的详细信息的最好方法。一般通过下面两步来查找：

1. 确定对象的类型

确定一个对象的类型，既可以使用 Get-Member 命令，也可以调用对象的 GetType 的方法，前提是该对象必须存在实例，如下：

```
PS >$date = Get-Date
PS >$date.GetType().ToString()
System.DateTime
```

2. 在 msdn 网站 <http://msdn.microsoft.com> 的搜索框中输入类型的名称

讨论

当使用 Get-Member 命令没有返回需要的信息的时候，MSDN 是一个非常好的选择，相对于 Get-Member 返回的帮助信息，MSDN 提供了更为详细的信息，通常包括类型的详细描述，相关的信息，甚至参考代码。MSDN 文档主要面向那些运用 C# 开发语言来使用这些类型的开发人员提供帮助信息，所以一开始你可能会有些不适应。

典型的现象是，一个类的文档首先是关于该类的概述，然后提供了该类成员的超级链接，主要包括支持的方法和属性。

注意：为了快速搜索类成员的文档，在搜索的时候加入术语 members，如：

```
typename members
```

一个类的成员的文档包括他的方法和属性，与 Get-Member 的输出类似。带有 S 图标的代表这是一个静态的方法和属性。你通过点击成员名字来获得更为详细的信息。

公共构造函数 (public constructors)

这部分列出类的构造函数，你通过 New-Object 命令创建一个类的时候会使用到构造函数。当你点击构造函数链接的时候，帮助文档会列出该类提供的所有的构造函数，包括需要传递的参数。

公共字段 / 公共属性 (public fields/public properties)

这部分列出一个对象的域和属性的名称，带有 S 图标的代表一个静态的域或属性。当你点击一个字段或属性的时候，帮助文档会提供这个字段或属性返回值的类型。

比如，在 System.DateTime.Now 的定义中有以下内容：

```
C#
public static DateTime Now { get; }
```

public 表示 Now 属性是公共的，所有成员都可以访问他；Static 表示这个属性是静态的（如 3.4 节中所述）；DateTime 表示这个属性返回的值是 DateTime 对象；Get 表示你可以从这个属性获得信息，但是不能设置属性的信息。当然，许多属性支持 Set（如 System.IO.FileInfo 中的 IsReadOnly 属性），表示你可以修改属性的值。

公共函数 (public methods)

这部分列出对象的方法的名称。S 图标表示这是一个静态方法，当你点击方法的时候，帮助文档列出所有不同的调用该方法的方式，包括在 PowerShell 中调用该方法需要传递的参数。

比如，在 System.DateTime.AddDays() 的定义中有以下内容：

```
C#
public DateTime AddDays (
```

```
        double value  
    }
```

`public` 表示 `AddDays` 方法是公共的，所有成员都可以访问他；`DateTime` 表示这个方法返回的值是 `DateTime` 对象；`double value` 表示这个方法需要一个 `double` 类型的参数。在这个例子中，当你调用该函数的时候，传递的参数决定了要加入到 `DateTime` 对象中的天数。

参考

- 3.4 节
 - 3.9 节

3.11 向对象添加自定义的方法和属性

问题

你有一个对象，希望向该对象添加自定义的属性或方法成员。

解决方案

通过使用 Add-Member 命令向对象中添加自定义的成员

讨论

当你向一个独立的对象添加自定义的成员的时候，Add-Member 命令是非常有用的。比如，想象一下，你想创建当前目录下文件的报告，希望报告中包括每个文件的所有者。在标准的 Get-ChildItem 返回的结果中没有 Owner 这样的属性，但是你可以通过编写脚本来加入这个自定义的属性，如例 3-4 所示：

例 3-4：用于向文件对象的输出中添加自定义属性的一个脚本

```
#####
## Get-OwnerReport.ps1
##
## Gets a list of files in the current directory, but with their owner added
## to the resulting objects.
##
## Example:
##     Get-OwnerReport
##     Get-OwnerReport | Format-Table Name,LastWriteTime,Owner
#####
```

```
$files = Get-ChildItem
foreach($file in $files)
{
    $owner = (Get-Acl $file).Owner
    $file | Add-Member NoteProperty Owner $owner
    $file
}
```

关于运行脚本的更多信息，参见 1.1 节。

尽管添加静态的信息是最常见的，`Add-Member` cmdlet 还支持其他几种类型的属性和方法，包括 `AliasProperty`、`ScriptProperty`、`CodeProperty`、`CodeMethod`、and `ScriptMethod`。关于这些类型的更详细的信息参见附录 A 中“使用 .NET 框架”，以及 `Add-Member` cmdlet 的帮助文档。

尽管 `Add-Member` 可以帮助你定制一些对象，但是不能定制类的所有对象，关于如何实现定制类的所有对象，参见 3.12 节。

计算出的属性 (Calculated properties)

计算的属性是向输出对象中添加信息的另外一个非常有用的方式。如果你的脚本或命令使用了 `Format-Table` 或 `Select-Object` 命令来生成输出，你可以通过提供一个表达式生成创建的附加属性的值，如：

```
Get-ChildItem |
    Select-Object Name,
        @{Name="Size (MB)"; Expression={ "{0,8:0.00}" -f ($_.Length / 1MB) }}
```

在这个例子中，我们获得目录下的文件的列表，然后使用 `Select-Object` 命令选择文件的名字，以及一个名称为 `Size (MB)` 的计算的属性。这个计算的属性以兆为单位，返回的是文件的大小，而不是默认的字节。

关于 `Add-Member` 命令的更多信息，键入 `Get-Help Add-Member`。

关于添加计算属性的更多信息，键入 `Get-Help Select-Object` 或 `Get-Help Format-Table`。

参考

- 1.1 节
- 3.12 节
- 附录 A 中的“使用 .NET 框架”

3.12 向类添加自定义的方法和属性

问题

你想向某一个类中的所有对象添加自定义的属性和方法。

解决方案

使用自定义的类型扩展文件 (*custom type extension files*)，向一个类中的所有对象添加成员。

讨论

尽管 Add-Member 命令在向对象添加自定义的成员的时候非常有用，但是需要你对逐个对象添加这些自定义的成员，不能实现向指定类中的所有对象添加成员，因此 PowerShell 支持另外一种机制——自定义的类型扩展文件来实现这个目的。

类型扩展文件采用 XML 格式，使得向任何类添加功能变得十分容易。如果你的代码只是与一种类型的对象交互，最好使用类型扩展文件。

注意：既然类扩展文件是 XML 格式的，一定要保证你添加的自定义的属性在编码后是有效的，比如 <、> 和 &。

举例来说，假设有一个脚本返回指定驱动器的剩余磁盘空间，脚本运行良好，但是你会发现通过 PowerShell 的 PsDrive 对象就可以实现这个功能，而且更简便。

开始起步

如果你还没有开始，那么创建一个类扩展文件的第一步就是创建一个空的文件，最佳保存路径是和你的配置文件放在同一目录，名称为 Types.Custom.ps1xml，如例 3-5 所示。

例 3-5：Types.Custom.ps1xml 文件示例

```
<?xml version="1.0" encoding="utf-8" ?>
<Types>
</Types>
```

紧接着，向你的配置文件中加入以下代码，可以使 PowerShell 在启动的时候加载你的类扩展文件，如下：

```
$typeFile = (Join-Path (Split-Path $profile) "Types.Custom.ps1xml")
Update-TypeData -PrependPath $typeFile
```

默认地，PowerShell 会根据安装目录下的 Types.ps1xml 文件加载一些类扩展，Update-TypeData cmdlet 通知 PowerShell 在启动的时候搜索你的 Types.Custom.ps1xml 文件，察看是否有自定义的类加载。-PrependPath 参数通知 PowerShell，如果发生冲突，用你定义的类扩展覆盖系统内建的类扩展。

一旦你创建了自定义的类文件，添加成员变得十分容易，下面这些例子将完成我们期望的功能：向 PowerShell 的 PsDrive 类添加功能。

为了完成这个任务，你需要扩展一下自定义的类文件，来指明要扩展的类是 System.Management.Automation.PSDriveInfo，如例 3-6 所示。System.Management.Automation.PSDriveInfo 返回值是 Get-Psdrive cmdlet 生成的类型。

例 3-6：自定义类型文件模板

```
<?xml version="1.0" encoding="utf-8" ?>
<Types>
    <Type>
        <Name>System.Management.Automation.PSDriveInfo</Name>
        <Members>
            add members such as <ScriptProperty> here
        </Members>
    </Type>
</Types>
```

添加脚本属性

你可以向类中添加脚本属性（*ScriptProperty*），使用 PowerShell 脚本作为扩展的语言。包含 3 个子元素：属性的名称、获得属性、设置属性。

在 GetScriptBlock 和 SetScriptBlock 部分，\$this 变量用来表示要被扩展的对象；在 SetScriptBlock 部分，\$args[0] 变量用来表示要设置的值。

例 3-7 向类 PSDriveInfo 添加了一个 AvailableFreeSpace 脚本属性，这部分代码应该放到例 3-6 中的 member 部分中。当你访问这个属性的时候，会返回驱动器上剩余空间；当你设置该属性时，会修改剩余空间的值。

例 3-7：一个 PSDriveInfo 类的脚本属性

```
<ScriptProperty>
    <Name>AvailableFreeSpace</Name>
    <GetScriptBlock>
        ## Ensure that this is a FileSystem drive
        if($this.Provider.ImplementingType -eq
            [Microsoft.PowerShell.Commands.FileSystemProvider])
```

```
{  
    ## Also ensure that it is a local drive  
    $driveRoot = $this.Root  
    $fileZone = [System.Security.Policy.Zone]::CreateFromUrl()  
    $driveRoot).SecurityZone  
    if($fileZone -eq "MyComputer")  
    {  
        $drive = New-Object System.IO.DriveInfo $driveRoot  
        $drive.AvailableFreeSpace  
    }  
}  
</GetScriptBlock>  
<SetScriptBlock>  
    ## Get the available free space  
    $availableFreeSpace = $this.AvailableFreeSpace  
  
    ## Find out the difference between what is available, and what they  
    ## asked for.  
    $spaceDifference = (([long] $args[0]) - $availableFreeSpace) / 1MB  
  
    ## If they want more free space than they have, give that message  
    if($spaceDifference -gt 0)  
    {  
        $message = "To obtain $args bytes of free space, " +  
                  " free $spaceDifference megabytes."  
        Write-Host $message  
    }  
    ## If they want less free space than they have, give that message  
    else  
    {  
        $spaceDifference = $spaceDifference * -1  
  
        $message = "To obtain $args bytes of free space, " +  
                  " use up $spaceDifference more megabytes."  
        Write-Host $message  
    }  
</SetScriptBlock>  
</ScriptProperty>
```

添加别名属性

别名属性（Alias Property）可以为一个属性定义一个可选的名称。在 PowerShell 处理类扩展文件的时候，被定义别名的属性可以不必存在，你可以在稍后的时间里通过 Add-Member cmdlet 添加属性。

例 3-8 为 PSDriveInfo 添加了一个名称为 free 的别名属性，这部分代码应该添加到例 3-6 中的 members 部分。当你访问这个 free 属性时候，实际上返回的是剩余磁盘空间的属性，当你设置这个属性的时候，也是设置剩余磁盘空间的属性。

例 3-8：PsDriveInfo 类的别名属性

```
<AliasProperty>
<Name>Free</Name>
<ReferencedMemberName>AvailableFreeSpace</ReferencedMemberName>
</AliasProperty>
```

添加一个脚本方法

脚本方法（ScriptMethod）允许你为一个对象定义一个动作，使用 PowerShell 脚本作为扩展的语言。脚本方法由 2 个子元素：属性的 Name 与 Script。

在脚本元素中，变量 \$this 代表的是你要扩展属性的对象。与单独的脚本相似的是，变量 \$args 表示向方法传递的参数；与单独的脚本不同的是，脚本方法不支持参数的状态。

例 3-9 为 PSDriveInfo 添加一个名称为 Remove 的脚本方法，与其他添加内容一样，把这部分代码添加到例 3-6 给出的模板的 member 部分。当你不带参数调用这个方法的时候，这个方法模拟删除驱动器（通过带有 -WhatIf 参数的 Remove-PsDrive 命令）。如果你调用这个方法，传递了 \$true 作为第一个参数，那么会从当前的 PowerShell 会话中删除驱动器。

例 3-9：PSDriveInfo 类的脚本方法

```
<ScriptMethod>
<Name>Remove</Name>
<Script>
$force = [bool] $args[0]
## Remove the drive if they use $true as the first parameter
if($force)
{
    $this | Remove-PSDrive
}
## Otherwise, simulate the drive removal
else
{
    $this | Remove-PSDrive -WhatIf
}
</Script>
</ScriptMethod>
```

添加其他扩展点

PowerShell 还支持其他一些针对类型扩展文件的附加功能，包括 CodeProperty、NoteProperty、CodeMethod 和 MemberSet。对于一般用户来说，这些功能可能很少使用；对于那些 PowerShell 提供者和 cmdlet 的开发人员来说，这些功能十分有用。关于这方面更多的信息，参见 Windows PowerShell SDK 或 MSDN 文档。

循环与流程控制

4.0 简介

当你开始编写脚本或命令来操作未知的数据的时候，对数据的循环与流程控制变得日益重要。

PowerShell的循环语句和命令可以允许你执行一个操作，而不用重复命令本身。比如说，执行某项任务特定次数，处理集合中的每一项，或者一直执行某一任务直到满足某一条件。

PowerShell 的流程控制和比较的语句可以让你的脚本或命令更适合操作未知类型的数据，可以基于数据的值来执行命令或跳过命令。

总而言之，循环和流程控制语句使 PowerShell 工具链的功能更加强大。

4.1 通过比较和逻辑操作做出决定 问题

你想对两个数据进行比较，根据比较的结果做出决定。

解决方案

使用 PowerShell 逻辑运算符对数据进行比较，然后根据结果做出决定。

常用比较运算符：

-eq, -ne, -ge, -gt, -lt, -le, -like, -notlike, -match, -notmatch,
-contains, -notcontains, -is, -isnot

常用逻辑符：

-and, -or, -xor, -not

关于这些运算符号的详细信息，参见附录 A 中的“比较运算符”。

讨论

PowerShell 的逻辑和比较运算符允许你对数据进行比较，或者根据条件测试数据。一个运算符要么比较两个数据（二进制），要么对一个数据进行测试。所有的比较运算符都是二进制的，多数逻辑运算符也是这样。唯一的一元逻辑运算符是“-not”，会根据测试数据的结果返回真或假。

比较运算符对两个数据进行比较，根据特定的比较符返回结果。比如，你想检查一个集合是否至少有指定数量的元素，如下：

```
PS >(dir).Count -ge 4
True
```

检查一个字符串是否与指定的正则表达式匹配，如下：

```
PS >"Hello World" -match "H.*World"
True
```

多数的比较运算符也会匹配输入数据。比如，当你对像字符串这样的简单数据进行比较的时候，-like 和 -match 比较符会检查字符串是否与给定的模式匹配。当你对简单数据组成的集合进行比较操作的时候，那些相同的比较符会返回集合中所有与给定模式相匹配的元素。

注意：-match 运算符把正则表达式作为他的参数。一个更常用的正则表达式的符号是 \$ 字符，它代表行的结尾。\$ 字符同时也表示 PowerShell 中变量的开始，为了防止发生冲突，建议将字符串用单引号括起来，如下：

```
PS >"Hello World" -match "Hello"
True
PS >"Hello World" -match 'Hello$'
False
```

关于比较运算符号如何匹配输入的详细内容，参见附录 A 中的“比较运算符”。

逻辑运算符将 true 或 false 语句联合起来，根据指定的逻辑操作返回结果。比如，你想检查一个字符串是否与给定的带通配符的模式相匹配，以及他是否比给定字符长度长，如下：

```
PS >$data = "Hello World"
PS >($data -like "*llo W*") -and ($data.Length -gt 10)
True
PS >($data -like "*llo W*") -and ($data.Length -gt 20)
False
```

一些比较的运算符号实际上已经和逻辑操作合并到一起了。既然使用与比较操作相对立的操作是如此普遍，PowerShell 提供了这样的比较运算符（比如 `-notlike`），这样你就不必使用 `-not` 运算符了。

关于逻辑操作的详细信息，参见附录 A 中“比较操作”。

当与流程控制结合到一起的时候，比较操作和逻辑操作构成了我们编写的脚本或命令的核心内容，来适应数据与输入。

要获得相关的详细信息，请参见附录 A 中“条件语句”。

关于 PowerShell 运算符的更多信息，键入 `Get-Help About_Operator`。

参考

- 附录 A 中“比较操作”
- 附录 A 中“条件语句”

4.2 使用条件语句控制脚本流程

问题

您想在 PowerShell 执行命令或脚本的一部分的情况下进行控制。

解决方案

使用 PowerShell 的 `if`、`elseif` 和 `else` 条件语句来控制脚本执行的流程，比如：

```
$temperature = 90
if($temperature -le 0)
{
    "Balmy Canadian Summer"
}
elseif($temperature -le 32)
{
    "Freezing"
}
```

```
elseif($temperature -le 50)
{
    "Cold"
}
elseif($temperature -le 70)
{
    "Warm"
}
else
{
    "Hot"
}
```

讨论

条件语句包括以下内容：

if 语句 (*statement*)

如果条件为真，执行 if 语句后的脚本程序

elseif 语句

如果条件为真，执行 `elseif` 语句后的脚本程序，在 `if` 或者 `elseif` 语句里没有条件为真。

else 语句

如果在 if 或 elif 语句里没有条件为真，则执行 else 后的脚本

关于这些流程控制的语句的更多信息，键入 `Get-Help About_Flow_Control`。

4.3 使用 switch 管理条件语句

问题

你想找到一条更简单的或者适合的方法来表示大量的if...elseif...else条件语句。

解决方案

可以使用PowerShell 中的 switch语句来表示大量的 if...elseif...else 条件语句。

比如：

```
$temperature = 20  
  
switch($temperature)  
{  
    { $_ -lt 32 }    { "Below Freezing"; break }
```

```
32          { "Exactly Freezing"; break }
{ $ _ -le 50 } { "Cold"; break }
{ $ _ -le 70 } { "Warm"; break }
default        { "Hot" }
}
```

讨论

PowerShell 中的 switch 语句使你更容易的对输入的数据做大量的比较。switch 语句支持一些选项，允许你配置 PowerShell 来把输入数据与条件进行比较，比如说通配符、正则表达式，甚至是任意的脚本块。既然搜索一个文件中的文本是一项常见的任务，PowerShell 中的 switch 语句就可以直接支持与文件中的文本进行比较。这个功能使得 PowerShell 的 switch 语句比起 C 或 C++ 中的 switch 语句更强大。

尽管通过 switch 语句使得表示大量的条件语句更加清晰，但与大量的顺序的 if...elseif...elseif...else 语句相比，一个 switch 语句更像是一个大量顺序排列的 if 语句。PowerShell 会把你输入的条件与每一个 switch 中的条件进行比较，如果比较结果是真，PowerShell 会执行那个 switch 中的脚本块；如果脚本块中不包含 break 语句，则会一直进行比较。

关于 switch 语句的更多信息，参见附录 A 中“条件语句”或键入 **Get-Help About_Switch**。

参考

- 附录 A 中“条件语句”

4.4 使用循环

问题

你想多次执行同样的代码。

解决方案

使用 PowerShell 的循环语句 (for, foreach, while 和 do)，或者 PowerShell 的 foreach-Object 命令来多次运行一个命令或脚本块。关于这些循环语句的更详细的信息，参见附录 A 中的“循环语句”，如下：

```
for 循环 (loop)
    for($counter = 1; $counter -le 10; $counter++)
    {
        "Loop number $counter"
    }
```

foreach 循环

```
foreach($file in dir)
```

{

"File length: " + \$file.Length

1

Foreach-Object 命令

```
Get-ChildItem | Foreach-Object { "File length: " + $_.Length }
```

while 循环

```
$response = ""
```

```
while($response ne "QUIT")
```

f

```
$response = Read-Host "Type something"
```

1

do..while 循环

```
$response = ""
```

do

1

```
$response = Read-Host "Type something"
```

```
} while($response ne "QUIT")
```

讨论

尽管每种循环语句都可以用来实现与其他循环语句相同的功能，每种循环语句都有其适用的场景。

当你需要执行一个已知次数的操作，使用 `for` 循环，这是非常通用的做法，称为 *counted for loop*：

当你拥有一个集合对象，并希望在集合中访问每一项，使用 `foreach` 循环。当你还没有加载整个的集合到内存中的时候（比如，从上面 `foreach` 例子中的 `dir` 集合），使用 `Foreach-Object` 命令是一种更为有效的选择。

与 `foreach` 循环不同的是，`Foreach-Object` 命令允许你在 PowerShell 生成元素的时候处理集合中的元素，这一点非常重要。在一个 `foreach` 循环中处理一个大的命令（如 `Get-Content hugefile.txt`）的整个输出内容会拖垮你的系统。

注意：一个对命令执行重复操作的小技巧如下所示：

```
PS >1..10 | foreach { "Working" }
Working
```

while 和 do..while 循环比较相似，都是当某一条件为真时执行循环。一个 while 循环会在脚本块运行前检查条件；而 do..while 循环会在脚本块运行后检查条件。

关于这些循环语句的详细信息，参见附录 A 中的“循环语句”或键入 **Get-Help About_For** 或键入 **Get-Help About_Foreach**。

参考

- 附录 A 中的“循环语句”

4.5 添加暂停或延迟

问题

你想暂停或延迟对脚本或命令的执行。

解决方案

暂停，直到用户按下回车键才继续，使用 Read-host cmdlet：

```
PS >Read-Host "Press ENTER"
Press ENTER:
```

暂停，直到用户按下某一键才继续，使用 \$host 对象的 Readkey() 方法，如下：

```
PS >$host.UI.RawUI.ReadKey()
```

暂停脚本指定的时间，使用 Start-Sleep cmdlet，如下：

```
PS >Start-Sleep 5
PS >Start-Sleep -Milliseconds 300
```

讨论

当你想暂停脚本的执行，直到用户按键或指定的时间，`Read-Host` 和 `Start-Sleep` 命令是最常用的。关于如何使用 `Read-Host` 命令来读取用户的输入，参见 12.1 节。在其他情况下，有时你可能会在脚本中编写循环，按指定的时间（比如每分钟一次或每秒 30 次）来运行脚本，这通常是一项艰巨的任务，因为命令在循环中可能占用了大量甚至是全部的时间。

在过去，许多电脑游戏遭遇了问题解决不正确的痛苦。为了控制游戏的速度，开发人员添加一些命令。经过了多次的失败，开发人员意识到，在一台电脑上，当他们让计算机每次更新屏幕的时候计数 100 万次，游戏可以正常运行。不幸的是，这些命令（比如计数）的执行严重地依赖于计算机的处理速度。一台运行速度快的计算机能够比一台速度慢的计算机更快的计算 100 万次（快到不可思议的程度），所以游戏的速度也会更快。

为了让你的循环运行在一个稳定的速度，你可以度量一下一个循环中的命令需要运行多长时间，然后根据需要加入延迟，如例 4-1 所示：

例 4-1：以稳定的速度运行循环

```
$loopDelayMilliseconds = 650
while($true)
{
    $startTime = Get-Date

    ## Do commands here
    "Executing"

    $endTime = Get-Date
    $loopLength = ($endTime - $startTime).TotalMilliseconds
    $timeRemaining = $loopDelayMilliseconds - $loopLength

    if($timeRemaining -gt 0)
    {
        Start-Sleep -Milliseconds $timeRemaining
    }
}
```

关于 `Start-Sleep` 命令的更多信息，键入 `Get-Help Start-Sleep`。

参考

- 12.1 节

字符串与非结构化文本

5.0 简介

在传统的 shell 和脚本语言中，创建与操作文本是一项主要的任务。事实上，Perl 是一个用来进行文本的操作简单的起步工具。他发展的良好，超出了预期的设计，受到广大用户的喜爱，但是需要进一步提高能力。

在基于文本的 shell 中，这种关注仍在延续。当你与系统发生的交互是通过操作程序的基于文本的输出的时候，多数情况下强大的文本处理能力变得尤为重要。这些文本分析工具，如 awk、sed 和 grep 构成了基于文本的系统管理的基石。

在 PowerShell 的基于对象的环境中，这个传统的工具链并没有扮演着很重要的角色。你可以使用 PowerShell 的命令更有效的完成这些工具完成的任务。当然，作为面向对象的 shell 并不意味着 PowerShell 放弃了所有的对文本的支持。处理字符串与非结构化的文本仍然是系统管理员日常工作中重要的组成部分。PowerShell 允许你在全域范围内管理系统中多数的任务（使用命令和对象），所以文本处理的工具可以把重点放到实际文本处理任务中。

5.1 创建字符串

问题

你想创建一个变量来保存文本。

解决方案

使用 PowerShell 的字符串变量来保存和操作文本。

为了定义一个字符串，支持变量扩展和转移字符，我们使用双引号把他括起来，如下：

```
$myString = "Hello World"
```

为了定义一个原生的字符串（不支持变量扩展或转义字符），我们使用单引号把他括起来，如下：

本文档讨论非字符串字符

讨论

字符串文本分成两类：按字面 (*literal*) (非扩展 (*nonexpanding*)) 和可扩展 (*expanding*) 的字符串。为了创建一个 literal 字符串，使用单引号将他扩起来 (\$myString = 'Hello World')；为了创建一个可扩展字符串，使用双引号将文本括起来 (\$myString = "Hello World")。

在一个 literal 字符串中，所有的单一号之间的文本都是字符串的一部分；在一个可扩展字符串中，PowerShell 扩展了用他们的值（分别如 \$myString 和新字符）变量的名称（如 \$myString）和转义序列（如 `n）。

关于 PowerShell 中对字符串的转义序列和替换规则的详细信息，参见附录 A 中的“字符串”。

“文本字符串中的所有字面字符都是字面的 (“*all text in a literal string is literal*”)” 规则的一个例外来自引号字符本身。PowerShell 允许你将该字符串的两个引号字符放在一起，以包括引号字符本身，如下：

```
$myString = "This string includes ""double quotes"" because it combined quote  
characters."  
$myString = 'This string includes ''single quotes'' because it combined quote  
characters.'
```

这样，当你试图在一个单引号引用的字符串中包含一个单引号的时候，可以避免转义的发生，如下面例子所示：

```
$myString = 'This string includes ' + "'' + 'single quotes' + "'"
```

注意：这个例子说明了在 PowerShell 中通过添加其他字符串来创建一个新的字符串是多么的容易。这为在脚本中建立格式化的报表提供了一种具有吸引力的方式；但是在使用时要小心，因为 .NET 框架管理字符串的方式可能会导致在向一个大的字符串尾添加信息的时候产生严重的性能问题。如果你打算创建大的报告，参见 5.13 节。

参考

- 5.13 节
- 附录 A 中的“字符串”

5.2 创建一个多行或格式化的字符串

问题

你想创建一个变量，用来保存包括换行的文本或其他明确格式化的字符串。

解决方案

使用 PowerShell 的 *here string* 来保存包括换行的文本和其他格式化的信息，如下：

```
$myString = @"
This is the first line
of a very long string. A "here string"
lets you to create blocks of text
that span several lines.
"@
```

讨论

PowerShell 是这样处理 *here sting* 的：当遇到 @ " 跟随着换行的时候，认为是 *here string* 的开始，当在新行遇到 "@ 字符时，认为是 *here string* 结束。这些看似奇怪的限制允许你创建大的格式化的文本的时候用到引用的字符，换行和其他符号。

注意：这些限制，虽然比较有用，但有时当你从网上拷贝 PowerShell 的例子的时候会产生问题。

Web 页面经常会在每行的结尾加入空格，会干扰对 *here string* 的开始标识的确认。当你的脚本中定义了一个 *here string*，而 PowerShell 又产生一个错误，你可以检查一下 *here string* 中是否在第一个引号后包括一个未知的空格。

与字符串 literals 相似，*here string* 可以是 *literal*（使用单引号）或 *expanding*（使用双引号）。

除了能有效的处理格式化的文本变量，*here string* 还提供了一个有用的途径来临时禁用脚本中的行。因为 PowerShell 没有提供多行的注释，所以你可以使用 *here string* 在其他脚本或程序语言中实现注释功能。如例 5-1 所示：

例 5-1：使用 here string 实现多行注释

```
## This is a regular comment
$null = @"
function MyTest
{
    "This should not be considered a function"
}

$myVariable = 10; 串符字的卦左部
"@"

## This is regular script again
```

为变量名称使用 \$null 告诉 Poweshell 不必为后续的使用而继续保留信息了。

5.3 在字符串中放置特殊字符

问题

你想在一个字符串变量中放置特殊的字符（比如说 tab 和换行）。

解决方案

在一个扩展的字符串中，使用PowerShell的转义字符来包括特殊字符，比如tab 或换行，如下：

```
PS >$myString = "Report for Today`n-----"  
PS >$myString  
Report for Today-----
```

讨论

正如 5.1 节中所讨论的，PowerShell 字符串分为两类：*literal* 和 *expanding* 字符串。一个 *literal* 的字符串使用单引号，而 *expanding* 字符串使用双引号。

在 literal 字符串中，所有引号中间的文本都认为是字符串的一部分；在 expanding 字符串中，PowerShell 可以用对应的值扩展变量的名称（如 \$ENV:SystemRoot）和转义序列（如 `n）。

注意：与其他语言不同的是，PowerShell 使用`字符作为转义的标识，而不是使用反斜线(\)作为转义符号。因为PowerShell 把重点放在系统管理上，而反斜线是在路径名称中是经常使用的。

关于 PowerShell 中的字符串的转义序列和替换原则的详细信息，参见附录 A 中“字符串”。

参考

- 5.1 节
- 附录 A 中的“字符串”

5.4 向字符串中插入动态信息

问题

你想向一个字符串中插入动态的信息，如其他变量的值。

解决方案

在可扩展的字符串中，可以通过在字符串中包含变量名称的方法插入变量的值，如下：

```
PS >$header = "Report for Today"
PS >$myString = "$header`n-----"
PS >$myString
Report for Today
-----
```

为了包含更复杂的信息，可以使用子表达式将内容括起来，如下：

```
PS >$header = "Report for Today"
PS >$myString = "$header`n$(`'-' * $header.Length)"
PS >$myString
Report for Today
-----
```

讨论

在一个扩展字符串中进行变量的替换是件简单的事情，但是其中的子表达式值得我们注意。

子表达式 (*subexpression*) 的语法是这样的，使用美元字符开始，紧跟着用括号括起来的 PowerShell 的命令，如下：

```
$(subexpression)
```

当 PowerShell 在扩展字符串中遇到子表达式的时候，他会处理子表达式，然后将处理

的结果插到扩展字符串中。在本解决方案中，子表达式`'-' * $header.Length`通知 PowerShell 使用变量`$header.Length`这么长写成一行。

另外一种向字符串中插入动态信息的方式是使用 PowerShell 的字符串格式化操作符，它基于 .NET 框架的字符串格式化规则，具体举例如下：

```
PS >$header = "Report for Today"
PS >$myString = "{0}`n{1}" -f $header, ('-' * $header.Length)
PS >$myString
Report for Today
-----
```

关于 PowerShell 的格式化操作符的信息，可以参见 5.6 节。关于 PowerShell 转义符的更多信息，可以键入 `Get-Help About_Escape_Character` 或键入 `Get-Help About_Special_Character`。

参考

- 5.6 节

5.5 禁止字符串包含动态信息

问题

你想防止 Powershell 解释字符串中的特殊字符或变量名。

解决方案

使用非扩展字符串来保障 PowerShell 能正确的解析你输入的字符串。非扩展字符串使用单引号把文本扩起来，如下：

```
PS >$myString = 'Useful PowerShell characters include: $, ` , " and { }'
PS >$myString
Useful PowerShell characters include: $, ` , " and { }
```

如果你想在字符串中包含换行，可以使用非扩展的 here string，如例 5-2 中所示：

例 5-2：一个包含新换行字符的非扩展的 here string

```
PS >$myString = @'
>> Tip of the Day
>> -----
>> Useful PowerShell characters include: $, ` , ' , " and { }
>> '@
```

```
PS >$myString  
Tip of the Day  
-----  
Useful PowerShell characters include: $, ', ', " and { }
```

讨论

在一个 literal 的字符串中，单引号之间的所有文本都被当作字符串的一部分。与扩展的字符串相对比，PowerShell 会展开变量名和转义序列它们的值。

注意：可以使用非扩展字符来管理文件和文件夹，因为它们往往包含特殊的字符，可能会错误的解释成转义字符。关于管理文件和文件夹名的更多信息，可以参见 17.6 节。

正如我们在 5.1 节中提到的，“文本字符串中的所有 literal 字符串都是字面化的”规则的一个例外来自引号字符本身。PowerShell 允许你将该字符串的两个引号字符放在一起，以包括引号字符本身，如下：

```
$myString = "This string includes ""double quotes"" because it combined quote  
characters."  
$myString = 'This string includes ''single quotes'' because it combined quote  
characters.'
```

参考

- 5.1 节
 - 17.6 节

5.6 在字符串中插入格式化的信息

问题

你想在字符串中放置格式化的信息，如右对齐的文本或舍入到特定数量的小数位数的数字。

解决方案

使用 PowerShell 的格式设置符来放置格式化的信息到一个字符串，如下：

```
PS >$formatString = "{0,8:D4} {1:C}`n"
PS >$report  = "Quantity Price`n"
PS >$report += "-----`n"
```

```
PS >$report += $formatString -f 50,2.5677
PS >$report += $formatString -f 3,9
PS >$report
Quantity Price
-----
0050 $2.57
0003 $9.00
```

讨论

PowerShell 的字符串格式设置符 (-f) 使用与 .NET 框架中的 `String.Format()` 方法相同的字符串格式作为规则。它将设置的格式放在左侧，要设置格式的内容放在右侧。

在本解决方案中，你格式化了两个数字：一个数量和一个价格。第一个数字 ({0}) 表示数量，并且在右对齐的 8 个字符框中 (,8)，它被格式设置为用 4 位数字 (:D 4) 表示的十进制数。第二个数 ({1}) 表示价格，它被格式化为货币 (:C)。

有关 PowerShell 的格式设置符的详细说明，请参阅附录 A 中的“简单操作符”。有关格式规则的详细列表，请参阅附录 H。

尽管主要用于控制布局的信息，字符串格式设置符也可以作为字符串的连接符，如下：

```
PS >$number1 = 10
PS >$number2 = 32
PS >"$number2 divided by $number1 is " + $number2 / $number1
32 divided by 10 is 3.2
```

字符串格式设置符使代码更易于阅读，如下：

```
PS >"{0} divided by {1} is {2}" -f $number2, $number1, ($number2 / $number1)
32 divided by 10 is 3.2
```

除了字符串格式设置符，PowerShell 还提供了三种格式命令 (-Table 格式，-Wide 格式，和 -List 格式），这允许您方便地生成格式化的报告。有关这些命令，详细信息请参阅附录 A 中的“格式输出”。

参考

- 附录 A 中的“格式输出”
- 附录 A 中的“简单操作”
- 附录 H 中的“.NET 字符串格式”

5.7 根据文本或模式在字符串中查找问题

你要确定一个字符串是否包含另一个字符串，或要查找一个字符串在另一个字符串中的位置。

解决方案

PowerShell 提供了几个选项以帮助你搜索文本中的字符串。

使用 `-like` 运算符来确定一个字符串是否与一个给定的类似 DOS 下的通配符相匹配，如下：

```
PS >"Hello World" -like "*llo W*"  
True
```

使用 `-match` 运算符来确定一个字符串是否与给定的正则表达式相匹配，如下：

```
PS >"Hello World" -match '.*l[1-z]o W.*$'  
True
```

使用 `Contains()` 方法，以确定一个字符串是否包含特定字符串，如下：

```
PS >"Hello World".Contains("World")  
True
```

使用 `IndexOf()` 方法来确定一个字符串在另一个字符串中的位置，如下：

```
PS >"Hello World".IndexOf("World")  
6
```

讨论

由于 PowerShell 字符串是具有 .NET 对象完全功能，所以它们直接支持很多面向字符串的直接操作。`Contains()` 和 `IndexOf()` 方法是 `String` 类支持的许多功能的两个示例。若要了解其他 `String` 类支持功能，请参阅 3.9 节。

虽然它们使用类似字符，但是简单的通配符和正则表达式可提供显著不同的用途。通配符比正则表达式更简单，而正因为这样，有更多的约束。因为你可以把通配符的使用规则汇总成仅有的四点，所以整本书都在描写如何使用正则表达式。

注意：正则表达式的一个常见用途是跨越多个行搜索一个字符串。默认的，正则表达式不跨行搜索，但你可以使用(?s)选项让它们可以跨行搜索，如下：

```
PS >"Hello `n World" -match "Hello.*World"  
False  
PS >"Hello `n World" -match "(?s)Hello.*World"  
True
```

通配符与简单的匹配项匹配，而正则表达式与更复杂的匹配项匹配。

有关-like运算符的详细说明，请参阅附录A中的“比较运算符”。有关-match运算符的详细说明，请参阅附录A中的“简单操作”。有关正则表达式规则和语法的详细列表，请参阅附录B。

当您尝试将PowerShell命令的结果存储到一个字符串时有时会出现一个困难，如例5-3所示。

例5-3：尝试存储字符串中的PowerShell命令的输出

```
PS >Get-Help Get-ChildItem  
  
NAME  
    Get-ChildItem  
  
SYNOPSIS  
    Gets the items and child items in one or more specified locations.  
(...)  
  
PS >$helpContent = Get-Help Get-ChildItem  
PS >$helpContent -match "location"  
False
```

在这种情况下，你指定的-match运算符搜索字符串的模式似乎失败。这是因为所有PowerShell命令都生成对象。如果您不把该输出存储到另一个变量中或传递到另一个命令，PowerShell会在转换为文本表示之前把结果显示给你。在例5-3中，\$helpContent是一个对象，而不只是它的字符串表示。

```
PS >$helpContent.Name  
Get-ChildItem
```

若要使用基于文本的PowerShell命令的表示形式，你可以显式地将其通过Out-String命令发送出去。Out-String命令将他的输入转换为用于在屏幕看到的基于文本的表单。

```
PS >$helpContent = Get-Help Get-ChildItem | Out-String  
PS >$helpContent -match "location"  
True
```

参考

- 3.9 节
- 附录 A 中的“比较操作符”
- 附录 A 中的“简单操作”
- 附录 B

5.8 替换字符串中的文本

问题

你要把字符串的一部分用另一字符串代替。

解决方案

PowerShell 提供了几个选项以帮助你用其他文本替换字符串中的文本。

使用 Replace() 函数对字符串本身执行简单的替换，如下：

```
PS >"Hello World".Replace("World", "PowerShell")
Hello PowerShell
```

使用 PowerShell 的正则表达式 -replace 运算符来执行更高级的正则表达式替换：

```
PS >"Hello World" -replace '(.*)(.*)','$2 $1'
World Hello
```

讨论

Replace() 方法和 -replace 运算符都提供有用的方法来替换字符串中的文本。Replace() 方法是最快但也是最受约束的。它将根据你指定的确切的替换匹配项来替换每一个与之相匹配的内容。-replace 操作符提供了更多的灵活性，因为它的参数是可以匹配和替换复杂模式的正则表达式。

注意：你使用带 -replace 运算符的正则表达式中通常包含一些字符，PowerShell 正常情况下会将它们解释为变量名称或转义符。若要防止 PowerShell 解释这些字符，使用 nonexpanding 字符串（单引号）。

有关 -replace 运算符的详细信息请参阅附录 A 中的“简单操作”和附录 H。

参考

- 附录 A 中的“简单运算符”
- 附录 H

5.9 字符串大、小写转换

问题

你要将字符串转换为大写或小写。

本文档中字符串替换 8.2

返回

解决方案

使用该字符串的 `ToUpper()` 和 `ToLower()` 方法，分别将其转换为大写和小写。

要将一个字符串转换成大写，使用 `ToUpper()` 方法：

```
PS >"Hello World".ToUpper()  
HELLO WORLD
```

要将一个字符串转换成小写，使用 `ToLower()` 方法：

```
PS >"Hello World".ToLower()  
hello world
```

讨论

由于 PowerShell 中的字符串是具有 .NET 对象完全功能的，它们支持很多面向字符串的直接操作。`ToUpper()` 和 `ToLower()` 方法是 `String` 类支持的许多功能的两个示例。若要了解其他 `String` 类支持功能，请参阅 3.9 节。

注意：PowerShell 或 .NET `String` 类的方法都不直接支持将单词的第一个字母大写。如果你只想把单词或句子的第一个字符转换成大写，请尝试以下命令：

```
PS >$text = "hello"  
PS >$newText = $text.Substring(0,1).ToUpper() +  
>>   $text.Substring(1)  
>> $newText  
>>  
Hello
```

当你将一个字符串转换为大写或小写的时候，要注意的事是你执行转换的原因。一个最常见原因是用于比较字符串，如例 5-4 中所示。

例 5-4：使用 ToUpper() 方法来使字符串规范化

```
## $text comes from the user, and contains the value "quit"
if($text.ToUpper() -eq "QUIT") { ... }
```

不幸的是，当你的脚本在不同的的语言环境下运行的时候，明确改变字符串的大写形式会失败。许多国家遵循不同的大写规则和比较规则，它和你现在使用的不同。例如，土耳其的语言中包括两种类型的字母“İ”：一个带小数点，另外一个不带。大写版本的小写字母 i 是大写的 I 带个点，而不是“QUIT”中的大写的 I。这些规则导致在使用土耳其语言的时候，例 5-4 中比较字符串的代码失效。

在对大小写不重要的情况下，要将一些输入与硬编码的字符串相比较，比较好的解决方案是使用 PowerShell 的 -eq 操作符，不用作任何的修改。默认情况下 -eq 操作符对大小写是不敏感的，是与语言无关的，如下：

```
PS >$text1 = "Hello"
PS >$text2 = "HELLO"
PS >$text1 -eq $text2
True
```

关于编写与文化相关的脚本的更多信息，参见 12.6 节。

参考

- 3.9 节
- 12.6 节

5.10 去掉字符串中的空格

问题

你想要从字符串或用户输入中去掉开始处或尾部处的空格。

解决方案

使用字符串的 Trim() 方法移除所有的开始处和结尾处的空格。

```
PS >$text = " `t    Test String`t  `t"
PS >"|" + $text.Trim() + "|"
|Test String|
```

讨论

Trim()方法可以清除一个字符串的开始位置和末尾的所有空白。如果只是想去除其中的一个，可以调用 TrimStart()或 TrimEnd()方法，去除开头或结尾的空白。如果想要从开始或结尾删除字符串中特定的字符，Trim()、TrimStart()和 TrimEnd()方法提供选项以支持这个功能。若要从一个字符串的末尾开始调整特定字符的列表，把该列表提供给方法，如例 5-5 所示。

例 5-5：从一个字符串的尾部去除指定字符项

```
PS >"Hello World".TrimEnd('d','l','r','o','W',' ')  
He
```

注意：下面的命令试图从字符串的尾部去除“World”，但是实际的结果是错误的，如下：

```
PS >"Hello World".TrimEnd(" World")  
He
```

之所以发生这种情况，是因为 TrimEnd() 方法使用一个字符列表来从一个字符串的末尾删除。如果需要，PowerShell 自动将一个字符串转换成字符列表，因此此命令中实际上和例 5-5 中的命令相同。

如果你想替换字符串中的任意位置的文本（并且不只是从开始处或结束处），请参阅 5.8 节。

参考

- 5.8 节

5.11 格式化日期的输出

问题

你想控制 PowerShell 显示或格式化日期的方式。

解决方案

若要控制一个日期的格式，可以使用下列选项之一：

- 使用 Get-Date 命令的 -Format 参数：

```
PS >Get-Date -Date "05/09/1998 1:23 PM" -Format "dd-MM-yyyy @ hh:mm:ss"  
09-05-1998 @ 01:23:00
```

- PowerShell 的字符串格式化 (-f) 运算符:

```
PS >$date = [DateTime] "05/09/1998 1:23 PM"  
PS >"{0:dd-MM-yyyy @ hh:mm:ss}" -f $date  
09-05-1998 @ 01:23:00
```

- 对象的 ToString() 方法:

```
PS >$date = [DateTime] "05/09/1998 1:23 PM"  
PS >$date.ToString("dd-MM-yyyy @ hh:mm:ss")  
09-05-1998 @ 01:23:00
```

- Get-Date 命令的 -UFormat 参数，它可以支持 Unix 日期格式:

```
PS >Get-Date -Date "05/09/1998 1:23 PM" -UFormat "%d-%m-%Y @ %I:%M:%S"  
09-05-1998 @ 01:23:00
```

讨论

除了 Get-Date 命令的 -UFormat 参数之外，PowerShell 中所有的日期格式都采用标准的 .NET DateTime 格式串。这些格式的字符串可以让你使用多种标准的格式中的一种来显示日期（比如系统中的短日期或长日期格式），或是完全使用自定义的格式。关于如何指定标准的 .NETDateTime 格式字符串，参见附录 I。

如果你已经习惯了使用 UNIX 风格的日期格式字符串（或转换成复杂格式中已存在的脚本），Get-Date 命令的 -UFormat 参数是十分有用的。它接受 UNIX 日期命令接受的格式化的字符串，但不提供任何功能的标准 .NET 日期格式。

当使用日期和时间的字符串版本时，应了解它们是最常见的国际化问题的来源，这就是在一台机器使用不同的语言运行那个书写语言的脚本。在北美 “05 / 09 / 1998” 表示 “1998 年 5 月 9 日”。但是，在很多其他区域，这意味着 “1998 年 9 月 5 日”。只要可能，尽量使用 DateTime 对象来比较（而不是字符串），可以避免这些区域性不同产生的问题。

例 5-6：使用 -gt 运算符比较 DateTime 对象

```
PS >$dueDate = [DateTime] "01/01/2006"  
PS >if([DateTime]::Now -gt $dueDate)  
>> {  
>>     "Account is now due"  
>> }  
>>  
Account is now due
```

注意：PowerShell 始终以北美日期设置格式解释 DateTime 常量，如 [DateTime] "05 / 09 / 1998"。基于同样的原因，所有的语言都按北美的格式解释数值常数（如 12.34）。如果不是这样做，几乎每个处理日期和时间的脚本在国际性系统上都将失败。

有关 `Get-Date` 命令的详细信息，键入 `Get-Help Get-Date`。

参考

- 附录 I .NET DateTime 格式

5.12 转换文本流为对象

PowerShell 的最强大的功能之一是它的基于对象的管道。你不用浪费你的精力来创建、析构和重新创建你的数据的对象表示形式。在其他 shell 中，当管道将其转换为纯文本的时候，你会失去完全保真的数据表示形式，你可以通过大量的文本分析重新获得一些，但并不是全部。

但是，你仍通常必须与来自 PowerShell 外部交互输入进行交互。基于文本的数据文件和旧的程序是两个示例。

PowerShell 为有关的三个文本的分析软件中的两个提供强大支持：

Sed

替换文本，PowerShell 中使用 `-replace` 参数。

Grep

搜索文本，PowerShell 中提供 `Select-String` 命令。

第三个传统的文本分析工具，*Awk*，允许你将一行文本拆分成多个组。PowerShell 提供了 `Split()` 方法来拆分字符串，但是缺少将一个字符串分成组的功能。

例 5-7 中 `Convert-TextObject` 脚本允许你将文本流根据你指定的规则转换成一组代表这些文本元素的对象。这样，你可以使用所有的 PowerShell 的基于对象的工具，这使你获得更好的操作能力。

例 5-7：Convert-TextObject.ps1

```
#####
## Convert-TextObject.ps1 -- Convert a simple string into a custom PowerShell
## object.
##
##     Parameters:
##
##         [string] Delimiter
##             If specified, gives the .NET Regular Expression with which to
##             split the string. The script generates properties for the
##             resulting object out of the elements resulting from this split.
```

```
## If not specified, defaults to splitting on the maximum amount
## of whitespace: "\s+", as long as ParseExpression is not
## specified either.
##
## [string] ParseExpression
## If specified, gives the .NET Regular Expression with which to
## parse the string. The script generates properties for the
## resulting object out of the groups captured by this regular
## expression.
## ** NOTE ** Delimiter and ParseExpression are mutually exclusive.
##
## [string[]] PropertyName
## If specified, the script will pair the names from this object
## definition with the elements from the parsed string. If not
## specified (or the generated object contains more properties
## than you specify,) the script uses property names in the
## pattern of Property1,Property2,...,PropertyN
##
## [type[]} PropertyType
## If specified, the script will pair the types from this list with
## the properties from the parsed string. If not specified (or the
## generated object contains more properties than you specify,) the
## script sets the properties to be of type [string]
##
##
## Example usage:
## "Hello World" | Convert-TextObject
## Generates an Object with "Property1=Hello" and "Property2=World"
##
## "Hello World" | Convert-TextObject -Delimiter "11"
## Generates an Object with "Property1=He" and "Property2=o World"
##
## "Hello World" | Convert-TextObject -ParseExpression "He(11.*o)r(1d)"
## Generates an Object with "Property1=llo Wo" and "Property2=ld"
##
## "Hello World" | Convert-TextObject -PropertyName FirstWord,SecondWord
## Generates an Object with "FirstWord=Hello" and "SecondWord=World"
##
## "123 456" | Convert-TextObject -PropertyType $([string],[int])
## Generates an Object with "Property1=123" and "Property2=456"
## The second property is an integer, as opposed to a string
##
param(
    [string] $delimiter,
    [string] $parseExpression,
    [string[]] $propertyName,
    [type[]] $propertyType
)
function Main(
    $inputObjects, $parseExpression, $propertyType,
    $propertyName, $delimiter)
{
    $delimiterSpecified = [bool] $delimiter
    $parseExpressionSpecified = [bool] $parseExpression
```

```
## If they've specified both ParseExpression and Delimiter, show usage
if($delimiterSpecified -and $parseExpressionSpecified)
{
    Usage
    return
}

## If they enter no parameters, assume a default delimiter of whitespace
if(-not $($delimiterSpecified -or $parseExpressionSpecified))
{
    $delimiter = "\s+"
    $delimiterSpecified = $true
}

## Cycle through the $inputObjects, and parse it into objects
foreach($inputObject in $inputObjects)
{
    if(-not $inputObject) { $inputObject = "" }
    foreach($inputLine in $inputObject.ToString())
    {
        ParseTextObject $inputLine $delimiter $parseExpression ` 
            $propertyType $propertyName
    }
}

function Usage
{
    "Usage: "
    " Convert-TextObject"
    " Convert-TextObject -ParseExpression parseExpression " +
        "-PropertyName propertyName] [-PropertyType propertyType]"
    " Convert-TextObject -Delimiter delimiter " +
        "-PropertyName propertyName] [-PropertyType propertyType]"
    return
}

## Function definition -- ParseTextObject.
## Perform the heavy-lifting -- parse a string into its components.
## for each component, add it as a note to the Object that we return
function ParseTextObject
{
    param(
        $textInput, $delimiter, $parseExpression,
        $propertyTypes, $propertyNames)

    $parseExpressionSpecified = -not $delimiter

    $returnObject = New-Object PSObject

    $matches = $null
    $matchCount = 0
    if($parseExpressionSpecified)
    {
        ## Populates the matches variable by default
        [void] ($textInput -match $parseExpression)
```

```
$matchCount = $matches.Count
}
else
{
    $matches = [Regex]::Split($textInput, $delimiter)
    $matchCount = $matches.Length
}

$counter = 0
if($parseExpressionSpecified) { $counter++ }
for(; $counter -lt $matchCount; $counter++)
{
    $propertyName = "None"
    $propertyType = [string]

    ## Parse by Expression
    if($parseExpressionSpecified)
    {
        $propertyName = "Property$counter"

        ## Get the property name
        if($counter -le $propertyNames.Length)
        {
            if($propertyName[$counter - 1])
            {
                $propertyName = $propertyNames[$counter - 1]
            }
        }

        ## Get the property value
        if($counter -le $propertyTypes.Length)
        {
            if($types[$counter - 1])
            {
                $propertyType = $propertyTypes[$counter - 1]
            }
        }
    }
    ## Parse by delimiter
    else
    {
        $propertyName = "Property$($counter + 1)"

        ## Get the property name
        if($counter -lt $propertyNames.Length)
        {
            if($propertyNames[$counter])
            {
                $propertyName = $propertyNames[$counter]
            }
        }

        ## Get the property value
        if($counter -lt $propertyTypes.Length)
        {
            if($propertyTypes[$counter])
        }
    }
}
```

```
{  
    $propertyType = $propertyTypes[$counter]  
}  
}  
}  
Add-Note $returnObject $propertyName`  
    ($matches[$counter] -as $propertyType)  
}  
$returnObject  
}  
  
## Add a note to an object  
function Add-Note ($object, $name, $value)  
{  
    $object | Add-Member NoteProperty $name $value  
}  
  
Main $input $sparseExpression $propertyType $propertyName $delimiter
```

有关运行脚本的更多信息请参阅 1.1 节。

参考

- 1.1 节

5.13 生成大的报告和文本流

问题

你要编写一个脚本来生成一个大的报表或大量的数据。

解决方案

只要有可能，最好的生成大量的数据的方法是利用 PowerShell 的流行为。本解决方案使用管道在命令之间传递数据，如下：

```
Get-ChildItem C:\ *.txt -Recurse | Out-File c:\temp\AllTextFiles.txt
```

而不是在每一步搜集输出数据：

```
$files = Get-ChildItem C:\ *.txt -Recurse  
$files | Out-File c:\temp\AllTextFiles.txt
```

如果你的脚本生成一个较大文本报告（流不是一个选项），使用 StringBuilder 类：

```
$output = New-Object System.Text.StringBuilder
Get-ChildItem C:\ *.txt -Recurse | 
    Foreach-Object { [void] $output.Append($_.FullName + "`n") }
$output.ToString()
```

而不仅仅是简单的文本串连：

```
$output = ""
Get-ChildItem C:\ *.txt -Recurse | Foreach-Object { $output += $_.FullName }
```

讨论

在 PowerShell 中，用管道组合命令是一个基本概念。当脚本和命令生成输出，只要可能，PowerShell 将该输出传递到管道中的下一个命令。在本解决方案中，`Get-ChildItem` 命令检索 C: 驱动器上的所有文本文件，花了很长时间才完成。当然，因为在开始时候立即生成数据，当 `Get-ChildItem` 命令生成数据的时候，PowerShell 可以立即传递该数据到下一个命令。这种任何命令的生成或使用数据的情况，称为流式处理 (*streaming*)。

因为它能够非常有效地使用内存，所以几乎在 `Get-ChildItem` 命令完成生成它的数据的同时，管道线数据传递也完成。

第二个 `Get-ChildItem` 示例（收集其数据）禁止 PowerShell 充分利用这个 streaming 机会。它首先将所有文件都存储在数组中，由于数据量较大，需要花费很长时间和巨大数量的内存。然后，它将所有这些对象发送到输出文件，这也需要很长时间。

但是，大多数命令可以直接使用管道所产生的数据，如上 `Out-File` 命令所示。对于这些命令，只要你将命令重定向到管道，PowerShell 就可以提供流行为。对于那些不支持直接从管道接收数据的命令，`Foreach-Object` 命令（别名为 `foreach` 和 `%`）允许你使用上面的命令生成每个数据片段，如示例 `StringBuilder` 中所示。

创建较大的文本报告

当你生成较大的报表时，比较常见的做法是把整个报表存储到字符串中，然后当脚本结束的时候把字符串流写到一个文件中。你通常可以通过直接把文本输出到目标（一个文件或屏幕）来更有效地完成这个过程，但有时这是不可能的。

既然 PowerShell 使得将更多的文本添加到字符串的结尾变得如此容易，许多人最初都选择那种方法。对于小型到中型字符串，它的效果非常好，但对于较大的字符串会造成极大的性能问题。



注意：作为此性能差异的示例，比较下面的例子

```
PS >Measure-Command {
>>     $output = New-Object Text.StringBuilder
>>     1..10000 | 
>>         Foreach-Object { $output.Append("Hello World") }
>> }
>>

(...)

TotalSeconds      : 2.3471592

PS >Measure-Command {
>>     $output = ""
>>     1..10000 | Foreach-Object { $output += "Hello World" }
>> }
>>

(...)

TotalSeconds      : 4.9884882
```

在 .NET Framework (和在 PowerShell 中)，当你创建字符串后，它们永远不会发生改变。当你将更多的文本添加到字符串的末尾时，PowerShell 通过合并两个较小的字符串来创建一个新的字符串。对于较大的字符串，这个操作需要很长时间，这正是 .NET Framework 包括 System.Text.StringBuilder 类的原因。与普通字符串不同，StringBuilder 类假定你将修改其数据，这种假设允许它能够有效适应更改。

在本章之后的内容中，你将学习如何使用字符串类来处理文本。在处理文本时，你需要知道字符串类的某些特性。例如，字符串类是不可变的，这意味着一旦一个字符串被创建出来，就不能再被修改。如果尝试修改一个字符串，将返回一个新的字符串。如果想要修改一个字符串，必须使用 StringBuilder 类。在下一节中，你将学习如何使用 StringBuilder 类。

计算和数学计算

6.0 简介

数学是任何脚本语言中的一个重要的特性。一种语言中对数学的支持包括加法、减法、乘法和除法，进一步扩展到更多高级数学运算。因此，PowerShell支持的强大的数学运算功能和面向计算的功能不会让你感到意外。

尽管 PowerShell 支持他的脚本语言从命令行访问，但这将保证总有一个功能强大且有用的命令行计算器在你的手里。

除了对传统的数学运算的支持，PowerShell 也关注于系统管理员，使用一些概念，如兆字节和千兆字节、简单的统计信息（如求总数和平均数）以及在不同制式下进行转换的支持。

6.1 执行简单的算法

问题

你希望使用 PowerShell 来进行简单的数学计算。

解决方案

使用 PowerShell 的算术操作符，如下：

+	加
-	减
*	乘

/	除
%	求余
$+=$, $-=$, $\star=$, $/=$, and $\%=$	上述的操作符组合
()	优先 / 顺序操作

有关这些数学运算符的详细说明, 请参阅附录 A 中的“简单操作”。

讨论

许多编程语言中的一个困难来自它们处理变量中数据的方式。例如, 此 C# 代码段将保存结果变量中“1”的值, 而用户可能希望在结果变量中保存 1.5 的浮点值:

```
double result = 0;  
result = 3/2;
```

这是因为 C# (与许多其他语言) 根据除法运算中的数据类型来确定该除法运算的结果。在上述示例中, 因为该除法运算中使用的是两个整数, 所以结果也是一个整数。

另一方面, PowerShell 也避免了此问题, 即使你在除法中使用两个整数, 如果需要的话, PowerShell 也会返回浮点值的结果, 这称为扩大转换 (*widening*)。

```
PS >$result = 0  
PS >$result = 3/2  
PS >$result  
1.5
```

这种自动的 *widening* 的一个例外情况是你明确地告诉 PowerShell 所需结果的类型的情况。例如, 你可以使用一个整数标识 ([int]) 指出你希望结果是一个整数:

```
PS >$result = [int] (3/2)  
PS >$result  
2
```

当浮点转换成整型的时候, 许多编程语言删除小数点后的部分, 这称为截断 (*truncation*)。另一方面, PowerShell 使用 *banker's rounding* 进行此转换。它将浮点数字转换为与其最接近的整数, 小数部分“舍入”到最接近的偶数。

一些编程技术仍在使用截断, 因此一种脚本语言以某种方式支持它仍然是很重要的。PowerShell 不具有一个内置运算符执行截断的除法, 但它通过 [Math]:: 来支持.NET Framework 中的 Truncate() 方法使用如下:

```
PS >$result = 3/2  
PS >[Math]::Truncate($result)  
1
```

如果该语法看起来非常复杂，下面的示例定义了一个 `trunc` 函数来截断其输入。

```
PS >function trunc($number) { [Math]::Truncate($number) }
PS >$result = 3/2
PS >trunc $result
1
```

参考

- 附录 A 中的“简单操作”

6.2 执行复杂的算法

问题

你希望使用 PowerShell 来进行更复杂的计算或高级的数学运算。

解决方案

PowerShell 主要通过 .NET Framework 中对 `System.Math` 类的支持完成更高级的数学任务。

若要求得一个数字的绝对值，可以使用 `[Math]::Abs()` 方法：

```
PS >[Math]::Abs(-10.6)
10.6
```

若要计算一个数的平方，使用 `[Math]::Pow()` 方法。在下面的例子中，计算 123 的平方：

```
PS >[Math]::Pow(123, 2)
15129
```

若要计算一个数的开方，使用 `[Math]::Sqrt()` 方法：

```
PS >[Math]::Sqrt(100)
10
```

若要计算一个角度（以弧度为单位）的正弦值、余弦值或正切值，使用 `[Math]::Sin()`，`[Math]::Cos()`，或 `[Math]::Tan()` 方法。

```
PS >[Math]::Sin([Math]::PI / 2)
1
```

要计算一个正弦值、余弦值或正切值的角度（以弧度为单位），使用 `[Math]::ASin()`，`[Math]::ACos()` 或 `[Math]::ATan()` 方法：

```
PS >[Math]::ASin(1)  
1.5707963267949
```

要学习如何找出 System.Math 类提供了哪些其他功能, 请参阅 3.9 节。

讨论

一旦你开始使用 System.Math 类, 你可能感觉到他看起来就像设计时遗漏了某些重要的功能。他支持求一个数的平方根, 但不支持求其他根 (例如立方根)。它支持以弧度表示的正弦值、余弦值和正切值 (和它们的反向求值), 不是常用的度为单位的度量。

计算任意的根

去真谛杂乱行进 S.8

若要确定一个数字的任意的根 (如立方根), 你可以使用例 6-1 中给出的函数。

例 6-1: 求根的函数与一些例子

```
PS >function root($number, $root) { [Math]::Exp($([Math]::Log($number) / $root)) }  
PS >root 64 3  
4  
PS >root 25 5  
1.90365393871588  
PS >[Math]::Pow(1.90365393871588, 5)  
25.0000000000001  
PS >[Math]::Pow( $root 25 5), 5)  
25
```

案式央轴

这个函数应用于以下数学事实, 你可以计算一个数的任意次的根 (如立方根)。尽管你可以选择 2 作为基础 (它生成 2 的正方根), System.Math 类中的 [Math]::Exp() 和 [Math]::Log() 方法使用了一种更常用的称为 e 的数学基数。

注意: [Math]::Exp() 和 [Math]::Log() 函数使用 e (大约为 2.718) 作为基础, 在很大程度上是因为该数经常出现在使用这些函数的数学类型中!

该示例还阐释了一个在计算机中的关于数学非常重要的一点。当你使用这个函数的时候 (或任何其他的操作浮点数), 始终应了解的是浮点答案的结果都是实际结果的近似值。如果你在同一语句中组合多个计算, 编程语言和脚本语言有时可以提高其答案 (如在第二个 [Math]::Pow() 尝试) 的准确性, 但这种情况很少见。

有些数学系统通过使用公式和计算符号 (不是数字) 来避免此问题。像人类一样, 这些系统知道你刚刚计算开平方的数的平方, 因此它们实际上不必执行这些操作。这些系统, 通常是非常专用和昂贵的。

使用度而不是弧度为单位

将弧度（数学家通常测量角度的方式）转换为度（大多数人测量角度的方式）比求根函数更直接。如果使用弧度为单位，一个圆包含 2 个 Pi 弧度；如果使用度为单位，一个圆包含 360 度。它提供了以下两个函数：

```
PS >function Convert-RadiansToDegrees($angle) { $angle / (2 * [Math]::Pi) * 360 }
PS >function Convert-DegreesToRadians($angle) { $angle / 360 * (2 * [Math]::Pi) }
```

它们的使用方法如下：

```
PS >Convert-RadiansToDegrees ([Math]::Pi)
180
PS >Convert-RadiansToDegrees ([Math]::Pi / 2)
90
PS >Convert-DegreesToRadians 360
6.28318530717959
PS >Convert-DegreesToRadians 45
0.785398163397448
PS >[Math]::Tan( (Convert-DegreesToRadians 45) )
1
```

参考

- 3.9 节

6.3 度量一个列表的统计属性

问题

你要测量对象组成的一个列表中数值（最小、最大、求和、平均）或文本（字符、单词、行）的特性。

解决方案

使用 Measure-Object 命令来测量列表的这些统计属性。

要测量对象流的数值特性，需要将这些对象管道输出给 Measure-Object 命令，如下：

```
PS >1..10 | Measure-Object -Average -Sum
Count      : 10
Average    : 5.5
Sum        : 55
Maximum   :
```

Minimum :
Property :

要测量对象流中的特定属性的数值特性，需要把属性的名称提供给 Measure-Object 命令的 -Property 参数。例如，测量目录中的文件：

```
PS >Get-ChildItem | Measure-Object -Property Length -Max -Min -Average -Sum  
  
Count : 427  
Average : 10617025.4918033  
Sum : 4533469885  
Maximum : 647129088  
Minimum : 0  
Property : Length
```

要测量对象流的文本特性，需要使用 Measure-Object 命令的 -Character、-Word 和 -Line 参数：

```
PS >Get-ChildItem > output.txt  
PS >Get-Content output.txt | Measure-Object -Character -Word -Line  
  
Lines Words Characters Property  
----- ----- ----- -----  
964 6083 33484
```

讨论

默认情况下 Measure-Object 命令只统计它收到对象的个数。如果你要测量这些对象中的其他属性（如最大值、最小值、平均值、总和、字符、单词或行），则需要命令提供选项。

对于数值属性，你通常不需要测量对象本身。相反，你可能希望从列表中度量指定的属性如文件的 Length 属性的值。为了达到该目的，Measure-Object 命令支持通过 -Property 参数来得到要测量的属性。

有时，你可能需要度量的属性不是一个简单的数字，如文件的 LastWriteTime 属性。由于 LastWriteTime 属性是一个 DateTime 类型，你不能立即确定它的平均值。但是，如果有一种属性使你通过一种可行的方式将其转换为数字（如 DateTime 的 Ticks 属性），然后你就可以计算其统计属性了。例 6-2 演示如何从文件的列表中获取 LastWriteTime 的平均值。

例 6-2：使用 Data Time 类中的 Ticks 属性来获取文件列表中 LastWriteTime 的平均值

```
PS >## Get the LastWriteTime from each file  
PS >$times = dir | Foreach-Object { $_.LastWriteTime }
```

```
PS >## Measure the average Ticks property of those LastWriteTime
PS >$results = $times | Measure-Object Ticks -Average

PS >## Create a new DateTime out of the average Ticks
PS >New-Object DateTime $results.Average

Sunday, June 11, 2006 6:45:01 AM
```

有关 **Measure-Object** 命令的更多信息，请键入 **Get-Help Measure-Object**。

6.4 使用二进制数

问题

你希望使用一个数的单个位或使用一系列的标志组合生成的数。

解决方案

若要直接输入一个十六进制数字，使用 **0x** 前缀：

```
PS >$hexNumber = 0x1234
PS >$hexNumber
4660
```

若要将数字转换为它的二进制表示形式，向 **[Convert]::ToString()** 方法提供 2 为基数：

```
PS >[Convert]::ToString(1234, 2)
10011010010
```

若要将二进制数转换成其十进制表示形式，向 **[Convert]::ToInt32()** 方法提供 2 为基数：

```
PS >[Convert]::ToInt32("10011010010", 2)
1234
```

若要管理一个数的单个位，使用 PowerShell 的二进制运算符。在这种情况下，该存档标志只是文件的众多属性之一，对于一个给定的文件，可能是真：

```
PS >$archive = [System.IO.FileAttributes] "Archive"
PS >attrib +a test.txt
PS >Get-ChildItem | Where { $_.Attributes -band $archive } | Select Name
Name
-----
test.txt
PS >attrib -a test.txt
```

```
PS >Get-ChildItem | Where { $_.Attributes -band $archive } | Select Name
```

讨论

在某些系统管理任务中，经常遇到数字本身似乎没有意义的情况。一个文件的属性是一个很好的示例：

```
PS >(Get-Item test.txt).Encrypt()
PS >(Get-Item test.txt).IsReadOnly = $true
PS >[int] (Get-Item test.txt -force).Attributes
16417
PS >(Get-Item test.txt -force).IsReadOnly = $false
PS >(Get-Item test.txt).Decrypt()
PS >[int] (Get-Item test.txt).Attributes
32
```

数字 16417 和 32 可以告诉我们有关文件的什么内容？

此答案来自于从另一个角度对属性的观察——可以是 True 或 False 的一组功能。例 6-3 显示了一个目录中的某个项可能的属性。

例 6-3：一个文件可能的属性

```
PS >[Enum]::GetNames([System.IO.FileAttributes])
ReadOnly
Hidden
System
Directory
Archive
Device
Normal
Temporary
SparseFile
ReparsePoint
Compressed
Offline
NotContentIndexed
Encrypted
```

如果一个文件是 ReadOnly、Archive 和 Encrypted 的，你可以考虑把这些作为该文件上的属性的简短说明：

```
ReadOnly = True
Archive = True
Encrypted = True
```

计算机有一种表示 true 和 false 值的集的非常简洁的方法称为二进制表示形式。若要以二进制表示一个目录项的属性，你只需将放它们放入一个表中。如果该属性应用到该项，我们给该项赋值为“1”否则为“0”（请参见表 6-1）。

表 6-1：一个目录项的属性

属性	真 (1) 或假 (0)
Encrypted	1
NotContentIndexed	0
Offline	0
Compressed	0
ReparsePoint	0
SparseFile	0
Temporary	0
Normal	0
Device	0
Archive	1
Directory	0
<Unused>	0
System	0
Hidden	0
ReadOnly	1

如果我们将这些功能视为单独的二进制数字，这样我们得到 100000000100001。如果我们将该数字转换为它的十进制形式，它会变成 16417：

```
PS >[Convert]::ToInt32("100000000100001", 2)
16417
```

此方法位于许多属性的核心，可以表示为功能或标志的组合。它不是通过表列出功能，而是通过文档描述数，这将导致该功能只有一个激活的，如 FILE_ATTRIBUTE_REPARSEPOINT = 0x400。例 6-4 演示各种文件属性的表示形式。

例 6-4：可能的文件属性的整数，十六进制和二进制表示形式

```
PS >$attributes = [Enum]::GetValues([System.IO.FileAttributes])
PS >$attributes | Select-Object `n
>>     @{"Name"="Property";
>>         "Expression"= { $_.ToString() };
>>     @{"Name"="Integer";
>>         "Expression"= { [int]$_.Value };
>>     @{"Name"="Hexadecimal";
>>         "Expression"= { [Convert]::ToString([int]$_.Value, 16) } },
>>     @ {"Name"="Binary";
>>         "Expression"= { [Convert]::ToString([int]$_.Value, 2) } } |
>>     Format-Table -auto
>>
```

Property	Integer	Hexadecimal	Binary
ReadOnly	1	1	1
Hidden	2	2	10
System	4	4	100
Directory	16	10	10000
Archive	32	20	100000
Device	64	40	1000000
Normal	128	80	10000000
Temporary	256	100	100000000
SparseFile	512	200	1000000000
ReparsePoint	1024	400	10000000000
Compressed	2048	800	100000000000
Offline	4096	1000	1000000000000
NotContentIndexed	8192	2000	10000000000000
Encrypted	16384	4000	100000000000000

了解了16417是如何形成的，你现在就可以通过有意义的方式使用属性。例如，PowerShell 的 -band 运算符允许你检查某些位是否已设置：

```
PS >$encrypted = 16384
PS >$attributes = (Get-Item test.txt -force).Attributes
PS >($attributes -band $encrypted) -eq $encrypted
True
PS >$compressed = 2048
PS >($attributes -band $compressed) -eq $compressed
False
PS >
```

尽管上面的示例显式使用数字值，但更常见的是通过名称输入数字：

```
PS >$archive = [System.IO.FileAttributes] "Archive"
PS >($attributes -band $archive) -eq $archive
True
```

有关 PowerShell 的二进制运算符的详细信息，请参阅附录 A 中的“简单运算符”。

参考

- 附录 A 中的“简单运算符”

6.5 简化管理用的常量

问题

你希望使用常见的管理单位数字（即千字节、兆字节和千兆字节），而不必记住或计算这些数字。

解决方案

使用 PowerShell 的管理常量（KB，MB 和 GB）可以帮助处理这些常见的数字。

通过连接 215 个千字节（每秒），计算 10.18 MB 文件的下载时间（以秒为单位）：

```
PS >10.18mb / 215kb  
48.4852093023256
```

讨论

PowerShell 的管理常量基于 2 的 n 次幂，这些常量是使用计算机时最常用的。每个都是前一个的 1024 倍：

```
1kb = 1024  
1mb = 1024 * 1 kb  
1gb = 1024 * 1 mb
```

某些硬盘驱动器制造商希望使用 2 的幂次方作为单位，如千字节、兆字节、千兆字节。每个数都是前一个数的 1000 倍。

尽管不由常量管理，PowerShell 仍然要使用这些基于 10 的幂的数字。例如，在 Windows 下算出一个 300 GB 硬盘驱动器有多大，方法如下：

```
PS >$kilobyte = [Math]::Pow(10, 3)  
PS >$kilobyte  
1000  
PS >$megabyte = [Math]::Pow(10, 6)  
PS >$megabyte  
1000000  
PS >$gigabyte = [Math]::Pow(10, 9)  
PS >$gigabyte  
1000000000  
PS >(300 * $gigabyte) / 1GB  
279.396772384644
```

6.6 在不同的进制间转换数字

问题

你希望在不同的进制下转换一个数字。

解决方案

PowerShell 脚本语言允许你直接输入十进制和十六进制数字。他本身并不支持其他制式

的数字，但支持在与 .NET Framework 交互操作时，与二进制、八进制、十进制和十六进制进行的转换。

若要将十六进制数转换为十进制表示形式，需要在数字前加入前缀 0x，表示为十六进制数：

```
PS >$myErrorCode = 0xFE4A
PS >$myErrorCode
65098
```

若要将数字转换为二进制表示形式，向 [Convert]::ToInt32() 方法提供参数 2 为基数：

```
PS >[Convert]::ToInt32("10011010010", 2)
1234
```

若要将数字转换为八进制表示形式，向 [Convert]::ToInt32() 方法提供参数 8 为基数：

```
PS >[Convert]::ToInt32("1234", 8)
668
```

若要将数字转换到十六进制表示形式，使用 [Convert] 类或 PowerShell 的格式运算符：

```
PS >## Use the [Convert] class
PS >[Convert]::ToString(1234, 16)
4d2

PS >## Use the formatting operator
PS >"{0:X4}" -f 1234
04D2
```

若要将数字转换到二进制表示形式，向 [Convert]::ToString() 方法提供参数 2 为基数：

```
PS >[Convert]::ToString(1234, 2)
10011010010
```

若要将数字转换到八进制数表示形式，向 [Convert]::ToString() 方法提供参数 8 为基数：

```
PS >[Convert]::ToString(1234, 8)
2322
```

讨论

当你处理的数字表示二进制数据（如一个文件的属性的组合时），会经常使用基于不同制式之间的数字的转换。有关处理如何二进制数据的更多信息，请参阅 7.4 节。

参考

- 7.4 节



www.123.com
技术文库

卷三

同小行愚用制常之金，（切合臣事君属下情文一个时，既非而臣二宗又不送报止长山中。
。序上。）因奉闻，恩旨准予相留。陈若二同顾恨少武清，既封荫半岁而加三。此

第三部分

常见任务

- 第 7 章，简单文件
- 第 8 章，结构化文件
- 第 9 章，支持 Internet 的脚本
- 第 10 章，代码复用
- 第 11 章，列表、数组和哈希表
- 第 12 章，用户交互
- 第 13 章，跟踪和错误管理
- 第 14 章，掌握环境
- 第 15 章，Windows PowerShell 的扩展
- 第 16 章，安全和脚本签名



代暗三策

丧丑见常

本章单行，章 15
单行对译，章 16
本脚注，章 17
用夏制，布 18
秦齐合璧，共 19
立文立排，章 20
吸管射箭射炮，章 21
象棋对弈，章 22
第 12 章，Windows Powershell 脚本 脚
本 10 章，支付宝本脚本全文

简单文件

7.0 简介

在管理系统时，你可能要花费大量时间来处理该系统上的文件。处理文件内容的许多工作都是很简单的，如：获取它们的内容、以某种模式搜索它们，或替换其中的文本内容。

虽然都是些简单的操作，但是PowerShell的面向对象的风格还是会添加几个统一和强大的功能。

7.1 获取文件的内容

问题

你想获得一个文件的内容。

解决方案

将文件名作为参数提供给 Get-Content 命令，来获得文件的内容，如下：

```
PS >$content = Get-Content c:\temp\file.txt
```

将文件名放在 \${ } 中，这是 Get-Content 的另外一种写法：

```
PS >$content = $(c:\temp\file.txt)
```

你也可以使用 .NET Framework 中的 System.IO.File 类的 ReadAllText() 方法，提供文件名作为参数：

```
PS >$content = [System.IO.File]::ReadAllText("c:\temp\file.txt")
```

讨论

PowerShell 提供了三个主要方法来获取文件的内容。第一个是 Get-Content 命令，为此目的而专门设计的。事实上，Get-Content cmdlet 适用于 PowerShell 中支持有内容的项概念的任何驱动器。这包括 Alias:、Function: 等等。第二个和第三个方法是 Get-Content 变量语法和 ReadAllText() 方法。

在处理文件时，`Get-Content` 命令逐行返回文件的内容。在处理过程中，PowerShell 提供有关该输出行的其他附加信息。这些信息作为属性附加到每个输出行，它包括所在的驱动器和路径等。

注意：如果你希望用你选择的字符串作为换行的标识（而不是默认的新行），可以使用 Get-Content 的 -Delimiter 参数输入你的选项。

虽然这样做有用，但在处理相对较大的文件的时候，如果你不使用这些信息，让 PowerShell 加载这些额外信息有时会降低脚本运行速度。如果你要更快地处理较大的文件，使用 `Get-Content` 命令的 `ReadCount` 参数，可以控制 PowerShell 每次从文件中读取的行数。当 `ReadCount` 等于 1 时（这是默认值），每次返回一行；当 `ReadCount` 等于 2 时，PowerShell 每次返回两行。对于小于 1 的 `ReadCount`，PowerShell 立即返回所有行。

注意：针对非常大的文件使用小于 1 的 ReadCount 的时候要小心。Get-Content 命令的优点之一是它的流式行为。不管如何大的文件，你都仍然能够处理文件中每行而不会用尽系统的内存。当使用的参数 ReadCount 小于 1 的时候，返回任何结果之前一直在读取整个文件，较大的文件有可能用尽你的系统内存。有关如何有效地利用 PowerShell 流功能的更多信息，请参阅 5.13 节。

如果性能是一个主要关注点，.NET Framework 中的 [File]::ReadAllText() 方法能够最快速的从磁盘读取文件。与 Get-Content 命令不同，它不会将文件拆分为换行符、附加任何信息，或对任何其他 PowerShell 驱动器采取动作。与带有小于 1 的 ReadCount 参数的 Get-Content 命令相似，在返回结果之前，它从文件中读取所有的内容。因此在非常大的文件上使用它时要格外小心。

有关 Get-Content 命令的更多信息，可以键入 **Get-Help Get-Content**。有关如何使用更加结构化的文件（如 XML 和 CSV）的信息，请参阅第 8 章。有关如何使用二进制文件的更多信息，请参阅 7.4 节。

参考

- 5.13 节
- 7.4 节
- 第 8 章结构化文件

7.2 搜索文件中的文本

问题

你希望在文件中查找字符串或正则表达式。

解决方案

如果要在一个文件中进行精确的搜索，但不区分大小写，使用带有 `-Simple` 参数的 `Select-String` 命令，如下：

```
PS >Select-String -Simple SearchText file.txt
```

如果要使用正则表达式来搜索文件内容，将匹配的内容提供给 `Select-String` 命令，如下：

```
PS >Select-String "\(...\)" ..... phone.txt
```

如果要使用正则表达式递归搜索所有扩展名是 `.txt` 的文件，可以将 `Get-ChildItem` 的结果通过管道输出给 `Select-String`：

```
PS >Get-ChildItem -Filter *.txt -Recurse | Select-String pattern
```

讨论

`Select-String` 命令是搜索文件中的字符串的最简单的命令。与传统的支持同样功能的文本匹配工具不同，`Select-String` 命令返回的结果包括了匹配内容的详细信息：

```
PS >$matches = Select-String "output file" transcript.txt
PS >$matches | Select LineNumber,Line
LineNumber Line
-----
7 Transcript started, output file...
```

如果你希望搜索指定扩展名的多个文件，`select-string` 命令允许你对文件名使用通配符。对于更复杂的文件列表，通常使用 `Get-ChildItem` 命令来生成。

在默认情况下，`Select-String` 命令会输出找到的匹配项的文件名、行号与匹配的行。对于某些情况来说，这些输出过于详细了，比如你想搜索哪个二进制文件包含一个特殊的字符串，在显示文本的时候二进制文件很少有意义，所以你的屏幕就很快填满了难以理解的字符。

你可以通过使用 `Select-String` 命令的参数 `-Quiet` 来解决上面提到的问题，如果文件中包含指定的字符就返回真，否则返回假。比如，在当前目录下搜索扩展名为 DLL 的文件是否包含文本 "Debug"，可以输入下面的命令：

```
Get-ChildItem | Where { $_ | Select-String "Debug" -Quiet }
```

其他两个经常用来搜索文件中的文本的工具是 `-match` 操作符和带 `-file` 选项的 `switch` 语句。关于这些命令的详细信息，请参见 5.7 节和 4.3 节。关于 `Select-String` 命令的更多信息，请键入 `Get-Help Select-String`。

参考

- 4.3 节
- 5.7 节

7.3 分析和管理基于文本的日志

问题

你想使用 PowerShell 的标准的对象管理命令分析一个基于文本的日志文件。

解决方案

可以使用 5.12 节中编写的 `Convert-TextObject` 脚本来处理基于文本的日志文件。它可以把文本流转换成对象流，这样，你就可以使用 PowerShell 中的标准的命令来处理这些日志。

`Convert-TextObject` 脚本接收 2 个参数：

1. 正则表达式描述如何分割文本。
2. 脚本指定属性的列表名。

作为示例，你可以使用 Windows 目录下的补丁日志文件，这些日志文件记录了本机系统

补丁安装的详细信息（Vista下例外），其中一个细节是在本次安装补丁的时候被修改的文件的名称和版本号，如例 7-1 所示：

例 7-1：获得被补丁程序修改的文件列表

```
PS >cd $env:WINDIR
PS >$parseExpression = "(.*): Destination:(.*) \((.*)\)"
PS >$files = dir kb*.log -Exclude *uninst.log
PS >$logContent = $files | Get-Content | Select-String $parseExpression
PS >$logContent

(...)

0.734: Destination:C:\WINNT\system32\shell32.dll (6.0.3790.205)
0.734: Destination:C:\WINNT\system32\wininet.dll (6.0.3790.218)
0.734: Destination:C:\WINNT\system32\urlmon.dll (6.0.3790.218)
0.734: Destination:C:\WINNT\system32\shlwapi.dll (6.0.3790.212)
0.734: Destination:C:\WINNT\system32\shdocvw.dll (6.0.3790.214)
0.734: Destination:C:\WINNT\system32\digest.dll (6.0.3790.0)
0.734: Destination:C:\WINNT\system32\browseui.dll (6.0.3790.218)
(...)
```

就像大多数日志文件一样，日志文件的格式非常规范但是难于管理，在这个例子中，包含下面的信息：

一个数字（记录从补丁开始到现在的秒）

文本，“: Destination:”

打补丁的文件

左括号

文件补丁的版本

右括号

你对其中的文本不感兴趣，但是其他的包括时间、文件和文件的版本是有用的属性，跟踪它们可以定义如下变量：

```
$properties = "Time", "File", "FileVersion"
```

现在，你可以使用 Convert-TextObject 脚本把文本输出转换成一个对象流，如下：

```
PS >$logObjects = $logContent |
>> Convert-TextObject -ParseExpression $parseExpression -PropertyName $properties
>>
```

我们可以使用 PowerShell 内置的命令来查询这些对象。比如，你可以查找在打补丁的过程中受影响最大的文件，如例 7-2 所示：

例 7-2：查找在打补丁的过程中受影响最大的文件

```
PS >$logObjects | Group-Object file | Sort-Object -Descending Count |
>> Select-Object Count, Name | Format-Table -Auto
>>
```

```
Count Name
-----
152 C:\WINNT\system32\shdocvw.dll
147 C:\WINNT\system32\shlwapi.dll
128 C:\WINNT\system32\wininet.dll
116 C:\WINNT\system32\shell32.dll
92 C:\WINNT\system32\rpcss.dll
92 C:\WINNT\system32\olecli32.dll
92 C:\WINNT\system32\ole32.dll
84 C:\WINNT\system32\urlmon.dll
(...)
```

通过运用这项技术，你可以处理大多数基于文本的日志文件。

讨论

在例7-2中，通过将输入的文本划分成简单的字符串组，你获得了你需要的所有信息。时间、文件和版本信息都有存在的目的。除了例7-2中使用的功能之外，Convert-TextObject脚本还支持一个参数，允许你控制这些属性的类型。如果某一属性需要按数字或日期来处理，而你按字符串处理那个属性的话，你可能会得到错误的结果。关于这个函数的更多信息，请参见Convert-TextObject脚本中关于-PropertyName参数的描述。

尽管多数的日志文件被设计在单行内记录内容，还是有些日志跨越多行。当一个日志文件包含跨越多行的内容，一般会使用特殊的分割符来区分每块内容，如下：

```
PS >Get-Content AddressBook.txt
Name: Chrissy
Phone: 555-1212
-----
Name: John
Phone: 555-1213
```

处理这种日志文件的关键有两点，第一是Get-Content命令的-Delimiter参数，可以用参数提供的符号来分割文件；第二是编写分析的正则表达式，来忽略每个记录中的换行字符。

```
PS >$records = gc AddressBook.txt -Delimiter "-----"
PS >$parseExpression = "(?s)Name: (\S+).*Phone: (\S+).*"
PS >$records | Convert-TextObject -ParseExpression $parseExpression
Property1
-----
Chrissy
John
Property2
-----
555-1212
555-1213
```

这个例子中分析表达式使用了单行(*single line*)选项(?S)，这样正则表达式在遇到(.*)的时候会换行。关于这些正则表达式选项的更多信息，请参见附录B。

对于非常大的日志文件，手工编写的分析工具可能无法满足你的要求，在这种情况下，一些特殊的日志管理工具可能会给你帮助，一个有用的工具是微软的免费的日志分析工具 Log Parser，可以通过网站 <http://www.logparser.com> 来下载。另外一个可选的办法是把日志内容导入到 SQL 数据库，然后执行相应的查询操作。

参考

- 5.12 节
- 附录 B 正则表达式参考

7.4 分析和管理二进制文件

问题

你想处理一个文件中的二进制数据。

解决方案

在处理文件中的二进制数据时主要使用两种技术。第一种是使用字节编码，这样 PowerShell 不会把内容当作文本来处理；第二种是使用 BitConverter 类来翻译这些字节。

例 7-3 显示了一个 Windows 可执行文件的特性。任何可执行文件（包括 DLL、.EXE 及其他），都包含一个二进制的区域称为 PE 头 (*portable executable header*)，这里面包含了文件的一些特性，如该文件是否是一个 DLL 文件。

关于 PE 头信息的详细信息，请参见 <http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx>。

例 7-3：Get-Characteristics.ps1

```
#####
## Get-Characteristics.ps1
##
## Get the file characteristics of a file in the PE Executable File Format.
##
## ie:
##
## PS >Get-Characteristics $env:WINDIR\notepad.exe
## IMAGE_FILE_LOCAL_SYMS_STRIPPED
## IMAGE_FILE_RELOCS_STRIPPED
```

```
## IMAGE_FILE_EXECUTABLE_IMAGE
## IMAGE_FILE_32BIT_MACHINE
## IMAGE_FILE_LINE_NUMS_STRIPPED
##
#####
param([string] $filename = $(throw "Please specify a filename."))

## Define the characteristics used in the PE file file header.
## Taken from http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
$characteristics = @{}
$characteristics["IMAGE_FILE_RELOCS_STRIPPED"] = 0x0001
$characteristics["IMAGE_FILE_EXECUTABLE_IMAGE"] = 0x0002
$characteristics["IMAGE_FILE_LINE_NUMS_STRIPPED"] = 0x0004
$characteristics["IMAGE_FILE_LOCAL_SYMS_STRIPPED"] = 0x0008
$characteristics["IMAGE_FILE.Aggressive_WS_TRIM"] = 0x0010
$characteristics["IMAGE_FILE_LARGE_ADDRESS_AWARE"] = 0x0020
$characteristics["RESERVED"] = 0x0040
$characteristics["IMAGE_FILE_BYTES_REVERSED_LO"] = 0x0080
$characteristics["IMAGE_FILE_32BIT_MACHINE"] = 0x0100
$characteristics["IMAGE_FILE_DEBUG_STRIPPED"] = 0x0200
$characteristics["IMAGE_FILE_Removable_RUN_FROM_SWAP"] = 0x0400
$characteristics["IMAGE_FILE_NET_RUN_FROM_SWAP"] = 0x0800
$characteristics["IMAGE_FILE_SYSTEM"] = 0x1000
$characteristics["IMAGE_FILE_DLL"] = 0x2000
$characteristics["IMAGE_FILE_UP_SYSTEM_ONLY"] = 0x4000
$characteristics["IMAGE_FILE_BYTES_REVERSED_HI"] = 0x8000

## Get the content of the file, as an array of bytes
$fileBytes = Get-Content $filename -ReadCount 0 -Encoding byte

## The offset of the signature in the file is stored at location 0x3c.
$signatureOffset = $fileBytes[0x3c]

## Ensure it is a PE file
$signature = [char[]]$fileBytes[$signatureOffset..($signatureOffset + 3)]
if([String]::Join('', $signature) -ne "PE`0`0")
{
    throw "This file does not conform to the PE specification."
}

## The location of the COFF header is 4 bytes into the signature
$coffHeader = $signatureOffset + 4

## The characteristics data are 18 bytes into the COFF header. The BitConverter
## class manages the conversion of the 4 bytes into an integer.
$characteristicsData = [BitConverter]::ToInt32($fileBytes, $coffHeader + 18)

## Go through each of the characteristics. If the data from the file has that
## flag set, then output that characteristic.
foreach($key in $characteristics.Keys)
{
    $flag = $characteristics[$key]
    if(($characteristicsData -band $flag) -eq $flag)
    {
        $key
```

讨论

对于多数的文件来说，这个技术是处理二进制数据的最简单的方式。如果你修改了二进制的数据，那么在你将它回写到磁盘的时候，就需要使用字节编码，如下：

```
$fileBytes | Set-Content modified.exe -Encoding Byte
```

对于非常大的文件，在处理的时候全部加载到内存中的速度之慢是难以接受的。如果你遇到这样的问题，不妨试试.NET 框架中与文件管理相关的类，包括 BinaryReader, StreamReader 等。关于使用.NET 框架中的类的详细信息，可以参见 3.4 使用.NET 对象。关于运行脚本的更多信息，请参见 1.1 节。

参考

- 1.1 节
- 3.4 节

7.5 创建临时文件

问题

你想临时创建一个文件，并且要确认文件并不存在。

解决方案

使用.NET 框架中的 [System.IO.Path]::GetTempFilename() 方法来创建临时文件，如下：

```
$filename = [System.IO.Path]::GetTempFileName()  
(... use the file ...)  
Remove-Item -Force $filename
```

讨论

一般情况下，为了完成某个任务，需要创建临时的文件。比如说，你可能想搜索一个文件并替换其中的文本，如果文件比较大的话，需要创建一个临时文件（参见 7.6 节）。另外一个例子是在 2.3 节中临时文件的使用。

一般来说，只要可能，我们会在C:\下或脚本所在的位置，或其他地方创建临时文件。这么做在本机没有问题，但是一旦换了环境，就很难正常工作。比如，如果用户没有使用管理员的权限来运行每日的任务，你的脚本在访问C:\的时候会因为没有权限而失败。

另外一个问题是如何保障创建的临时文件的名称是唯一的。如果你在脚本里对文件的名字硬编码，那么当你同时运行两个一样的脚本时会报错。你可能会编写一个脚本来搜索文件名是否存在，如果不存在，创建，然后使用它。不幸的是，在这种情况下仍然可能会出现问题，如果另外一个同样脚本在发现不存在之后先创建了文件，而在你的脚本准备去创建文件之前。

最后涉及的是一些安全方面的问题，你创建的临时文件可能被其他人读取或写入。

幸运的是，.NET框架提供了[`System.IO.Path`]::`GetTempFilename()`方法来解决这个问题。它自动在一个安全的目录创建一个名称唯一的临时文件，该方法返回文件的名称，你可以通过它来访问文件。

注意：当你的脚本不再需要临时文件的时候，记得要删除它，不然的话会浪费磁盘空间，引起不必要的混乱。记住，你的脚本的任务是解决问题，而不是引起错误。

在默认的情况下，`GetTempFilename()`方法返回的是扩展名为`.tmp`的临时文件。在多数情况下，文件的扩展名不是问题，但是在一些特殊情况下，你可能需要创建一个指定扩展名的文件，`[System.IO.Path]::ChangeExtension()`方法允许你修改临时文件的扩展名。下面的例子创建了一个临时文件，它使用`.cs`扩展名，如下：

```
$filename = [System.IO.Path]::GetTempFileName()
$newname = [System.IO.Path]::ChangeExtension($filename, ".cs")
Move-Item $filename $newname
(... use the file ...)
Remove-Item $newname
```

参考

- 2.3 节
- 7.6 节

7.6 搜索和替换文件中的文本

问题

你想搜索文件中的文本，使用其他字符替换那段文本。

解决方案

为了对文件进行搜索与替换操作，首先将文件的内容保存到一个变量，然后对变量进行查找与替换操作，最后将变量回写到文件，如例 7-4 所示：

例 7-4：替换文件中的文本

```
PS >$filename = "file.txt"
PS >$match = "source text"
PS >$replacement = "replacement text"
PS >
PS >$content = Get-Content $filename
PS >$content
This is some source text that we want
to replace. One of the things you may need
to be careful careful about with Source
Text is when it spans multiple lines,
and may have different Source Text
capitalization.
PS >
PS >$content = $content -creplace $match,$replacement
PS >$content
This is some replacement text that we want
to replace. One of the things you may need
to be careful careful about with Source
Text is when it spans multiple lines,
and may have different Source Text
capitalization.
PS >$content | Set-Content $filename
```

讨论

使用 PowerShell 来搜索与替换一个文件或多个文件的内容是一个典型的使用工具自动化完成重复性工作的例子。一些以往需要花费数月手工完成的任务可以压缩到几分钟（或者最多几小时）。

注意：注意在需要区分大小写的情况下使用-creplace操作符。因为你提供的替换的字符串严格地区分大小写，所以在执行替换操作的时候，需要加上-creplace操作符。如果要替换的字符有多种大写的方式，那么搜索与替换的操作也会执行相应的次数。

例 7-4 说明了最常见的一些场景：

- 你使用的是 ASCII 编码的文本文件。
- 你使用纯文本替换纯文本。

- 你不用担心匹配的文本跨多行。
- 你的文本文件相对较小。

如果上述某一条件不为真，那么你可以参考下面的讨论来帮助你在各种条件下进行文本的查找与替换。

使用 Unicode 编码（或其他编码）的文件

默认的，`Set-Content` 假设你希望输出的文件包含的是 ASCII 文本。如果你的文件采用了其他的编码方式，比如 Unicode 或 OEM，可以使用 `Out-File` 命令的 `-Encoding` 参数来指定编码方式，如下：

```
$content | Out-File -Encoding Unicode $filename  
$content | Out-File -Encoding OEM $filename
```

使用模式而不是纯文本来替换文本

比较常见的任务是使用一个遵照原文的字符串来替换另外一个遵照原文的字符串，而在一些其他场合，你可能希望依照一个模式来替换文本。一个例子就是交换姓与名的位置。`PowerShell` 支持在替换操作中使用正则表达式来实现这种替换，如下：

```
PS >$content = Get-Content names.txt  
PS >$content  
John Doe  
Mary Smith  
PS >$content -replace '(.*)(.*)','$2,$1'  
Doe, John  
Smith, Mary
```

替换跨行的文本

本案例中，使用 `Get-Content` 命令从文件中逐行返回文件内容。当你使用 `-replace` 操作符执行替换操作的时候，该操作会逐行进行。如果匹配的内容跨多行，`-replace` 操作符将不会检测到相匹配的内容，从而不执行替换操作。

如果你希望替换跨行的文本，首先要停止把输入的文本当作行的集合来处理；同时你还要在替换的表达式中忽略换行，如例 7-5 所示：

例 7-5：替换跨行的文本

```
$filename = Get-Item file.txt  
$singleLine = [System.IO.File]::ReadAllText($filename.FullName)  
$content = $singleLine -creplace "(?s)Source(\s*)Text", 'Replacement$1Text'
```

例 7-5 中的前两行将整个文件内容当作一个字符串赋给变量。通过调用 .NET 框架中的 [System.IO.File]::ReadAllText() 方法获得文件的内容，而 Get-Content 命令将文件的内容分割成多行。

第三行通过使用正则表达式的模式来替换文本。Source(\s*)Text 表示要搜索单词 Source，后面可以有空白内容，紧跟着是单词 Text。由于正则表达式的空白部分周围有括号，我们想要记住空白是什么。默认情况下正则表达式不会把换行符作为空白，正则表达式的第一部分使用单行选项以便把换行符作为空白来计算。-replace 运算符的替换部分替换匹配 Replacement 的内容，紧跟着匹配我们捕获 (\$1) 的空白内容。有关详细信息，请参阅附录 A 中的“简单操作符”。

替换大文件中的文本

到目前为止使用这些方法，都是将文件的内容读取到内存中进行处理，一旦我们在内存中完成替换，我们将更新的内容回写到磁盘。这适用于替换小型、中型、甚至大型文件中的文本。对于非常大的文件（例如超过几百兆字节），使用此内存可能会加大系统的负担，使你的脚本运行速度下降。若要解决该问题，你可以逐行处理文件，而不是一次处理整个文件。

由于你正在逐行处理文件，当你尝试向其写回替换文本时它仍将逐行写入。通过将替换文本写入到一个临时文件的方法，你可以避免此问题。一旦你已完成扫描文件，你可以删除它，使用临时文件替换文本内容。

```
$filename = "file.txt"
$temporaryFile = [System.IO.Path]::GetTempFileName()

$match = "source text"
$replacement = "replacement text"

Get-Content $filename |
    Foreach-Object { $_ -creplace $match,$replacement | Add-Content $temporaryFile }

Remove-Item $filename
Move-Item $temporaryFile $filename
```

参考

- 附录 A 中的“简单操作符”

第8章

结构化文件

8.0 简介

在纯文本的系统管理的环境下，管理结构化的文件通常是一个问题。例如，使用（或编辑）一个 XML 文件意味着将其加载到一个编辑器来手动修改或者编写一个自定义工具来处理。更糟糕的是，它可能意味着修改该文件，就像处理纯文本文件一样，而我们不希望破坏 XML 文件本身的数据。

同样地，使用 CSV 格式的文件意味着要自己拆分文件，用逗号分隔每一行。它看上去是一种很好的方法，直到你发现自己不止面对最简单的数据时就不是这样了。

结构和结构化的文件不只来自其他程序。在编写脚本时，一个常见的目的是保存结构化的数据，以便以后可以使用它。在大多数脚本（和编程）语言中，这需要你设计一个数据结构以容纳该数据，设计一种方式来在磁盘中存储和检索，并在你想重新使用的时候，可以回写成可用的形式。

幸运的是，PowerShell 使得使用 XML、CSVs 或其他你自定义结构的文件变的更容易了。

8.1 访问 XML 文件中的信息

问题

你想访问一个 XML 文件中的信息。

解决方案

使用 PowerShell 的 XML 转换符将纯文本格式的 XML 转成一种更容易使用的格式。下面的例子从 Windows PowerShell 博客中下载 RSS 源：

```
PS >$xml = [xml] (Get-Content powershell_blog.xml)
```

注意：请参见 9.1 节，了解如何使用 PowerShell 下载文件。

与其他对象相似，当你浏览时，PowerShell 会显示 XML 文件的属性。这些属性包括子节点和属性，如例 8-1 所示：

例 8-1：访问一个 XML 文档的属性

```
PS >$xml

xml          xmlstylesheet      rss
---          -----            ---
                           ---            rss
                           rss

PS >$xml.rss

version : 2.0
dc       : http://purl.org/dc/elements/1.1/
slash    : http://purl.org/rss/1.0/modules/slash/
wfw     : http://wellformedweb.org/CommentAPI/
channel : channel
```

如果有一个以上的节点共用一个名字（就像 RSS 源中的 item 结点），那么属性的名字代表一个节点的集合，如下：

```
PS >($xml.rss.channel.item).Count
15
```

你可以单独访问这些项，就像你处理一个数组一样，如例 8-2 所示。

例 8-2：访问一个 XML 文档的项

```
PS >($xml.rss.channel.item)[0]

description : <P>Since a lot of people have been asking about it, yes - my
               (...)

guid        : guid
title       : "Windows PowerShell in Action" has been released
comment     : http://blogs.msdn.com/powershell/rsscomments.aspx?PostID=171
               8281
link        : http://blogs.msdn.com/powershell/archive/2007/02/19/windows-
               powershell-in-action-has-been-released.aspx
pubDate     : Mon, 19 Feb 2007 20:05:00 GMT
comments    : {4, http://blogs.msdn.com/powershell/comments/1718281.aspx}
```

```
commentRSS : http://blogs.msdn.com/powershell/commentrss.aspx?PostID=1718  
             281  
creator    : PowerShellTeam
```

你可以访问这些元素的属性，就像你可以正常访问一个对象一样，如下：

```
PS >($xml.rss.channel.item)[0].title  
"Windows PowerShell in Action" has been released
```

既然这些元素是对象，你可以使用 PowerShell 中基于对象的高级命令，如排序和过滤。例 8-3 演示如何使用这些命令。

例 8-3：在 XML 文件中对项进行排序和过滤

```
PS >$xml.rss.channel.item | Sort-Object title | Select-Object title
```

```
title  
-----  
"Windows PowerShell in Action" has been released  
Controlling PowerShell Function (Re)Definition  
Execution Policy and Vista  
Executive Demo  
It's All about Economics  
NetCmdlets Beta 2 is now Available.  
Payette Podcast  
Port 25 interview with Bruce Payette  
PowerShell Benefits Over COM Scripting  
PowerShell Cheat Sheet - Now in XPS  
PowerShell Tip: How to "shift" arrays  
Processing text, files and XML  
Virtual Machine Manager's PowerShell Support  
Windows PowerShell 1.0 for Windows Vista  
Working With WMI Events
```

讨论

PowerShell 的内置的对 XML 的支持使得浏览与访问 XML 文件变得容易。通过作为属性公开 XML 层次结构，你可以执行大多数任务，而不必求助于纯文本处理或自定义工具。

事实上，PowerShell 对与 XML 之间的交互的支持不仅仅是以友好的对象方式展示数据。实际上由 [xml] 转换创建的对象代表的是 .NET Framework 中的 System.Xml.XmlDocument 对象；同样的，结果对象中的每个属性代表的是 .NET Framework 中的 System.Xml.XmlElement 对象。这些基础的对象提供了大量的附加功能，这使你可以对 XML 文件执行常见的和复杂的任务。

基础的 System.Xml.XmlDocument 和 System.Xml.XmlElement 对象提供的支持使得你的 XML 文件提供了对自己有用的属性，如 Attributes、Name、OuterXml 等等。由于这些属性可能会干扰你从 XML 文件中访问属性方法，因此 PowerShell 在默认情况

下会隐藏它们。为了访问它们，你可以在任何节点上使用 `Psbase` 属性。`PsBase` 属性适用于 PowerShell 中的任何对象，表示 PowerShell 下对对象抽象的。如：

```
PS >$xml.rss.psbase.Attributes
#text
-----
2.0
http://purl.org/dc/elements/1.1/
http://purl.org/rss/1.0/modules/slash/
http://wellformedweb.org/CommentAPI/
```

关于如何使用 .NET 对象完成高级任务的更多内容，请参阅 8.2 节和 8.3 节。

有关在 PowerShell 中 XML 使用的更多信息，请参阅附录 A 中的“XML”。

参考

- 8.2 节
- 8.3 节
- 9.1 节
- 附录 A 中的“XML”

8.2 对 XML 文件执行 XPath 查询

问题

你想使用 XML 的标准的 *Xpath* 语法执行一个高级查询。

解决方案

使用 PowerShell 的 XML 转换符将纯文本格式的 XML 转成一种更容易使用的格式。下面的例子从 Windows PowerShell 博客中下载 RSS 源：

```
PS >$xml = [xml] (Get-Content powershell_blog.xml)
```

然后使用变量的 `SelectNodes()` 方法来执行查询。比如，要查找所有少于 20 个字符的标题：

```
PS >$query = "/rss/channel/item[string-length(title) < 20]/title"
PS >$xml.SelectNodes($query)
```

```
#text
-----
Payette Podcast
Executive Demo
```

讨论

尽管每种语言都有自己的特点，但是 XPath 的查询语法提供了一种有力的、以 XML 为中心的方法来编写高级的查询。

对于简单的查询，你可以发现使用 PowerShell 的基于对象的 XML 导航内容更容易使用。

关于使用 PowerShell 中的 XML 类型的更多内容，请参见附录 A 中的“XML”。

参考

- 附录 A 中的“XML”

8.3 修改 XML 文件中的信息

问题

你想使用 PowerShell 来修改 XML 文件中的数据。

解决方案

为了修改一个 XML 文件中的数据，你可以将文件加载成 PowerShell 的 XML 数据类型，修改你要改的内容，然后回写到磁盘。例 8-4 演示如何修改文件内容：

例 8-4：从 Power Shell 修改 XML 文件

```
PS >## Store the filename
PS >$filename = (Get-Item phone.xml).FullName
PS >
PS >## Get the content of the file, and load it
PS >## as XML
PS >Get-Content $filename
<AddressBook>
  <Person contactType="Personal">
    <Name>Lee</Name>
    <Phone type="home">555-1212</Phone>
    <Phone type="work">555-1213</Phone>
  </Person>
  <Person contactType="Business">
    <Name>Ariel</Name>
```

```
<Phone>555-1234</Phone>
</Person>
</AddressBook>
PS >$phoneBook = [xml] (Get-Content $filename)
PS >
PS >## Get the part with data we want to change
PS >$person = $phoneBook.AddressBook.Person[0]
PS >
PS >## Change the text part of the information,
PS >## and the type (which was an attribute)
PS >$person.Phone[0].#text" = "555-1214"
PS >$person.Phone[0].type = "mobile"
PS >
PS >## Add a new phone entry
PS >$newNumber = [xml] '<Phone type="home">555-1215</Phone>'
PS >$newNode = $phoneBook.ImportNode($newNumber.Phone, $true)
PS >[void] $person.AppendChild($newNode)
PS >
PS >## Save the file to disk
PS >$phoneBook.Save($filename)
PS >Get-Content $filename
<AddressBook>
  <Person contactType="Personal">
    <Name>Lee</Name>
    <Phone type="mobile">555-1214</Phone>
    <Phone type="work">555-1213</Phone>
    <Phone type="home">555-1215</Phone>
  </Person>
  <Person contactType="Business">
    <Name>Ariel</Name>
    <Phone>555-1234</Phone>
  </Person>
</AddressBook>
```

讨论

在上面的例子中，你将 *Lee* 的电话号码从 555-1212 修改为 555-1214，同时把电话号码的类型从 "home" 修改为 "mobile"。

将新的信息添加到 XML 几乎是一样容易的。要将信息添加到 XML 文件，必须将其作为一个子节点（child node）添加到文件中的另一个节点上。要获取这个子节点，最简单的方法是编写表示 XML 的字符串，然后创建一个临时的 PowerShell XML 文档。从该文档中，你可以使用主 XML 文档的 ImportNode() 函数导入你所关注的节点，特别是在此示例中的 *Phone* 节点。

获得该节点的子节点后，我们需要决定在何处放置它。由于我们希望此 *Phone* 节点是 *Lee* 所在的 *Person* 节点的子级，所以我们将它放置在那。若要将一个子节点（例 8-4 中的 \$newNode）添加到目标节点（在示例中的 \$person），需要在目标节点上使用 AppendChild() 方法。

注意：XML 文档的 `Save()` 方法允许你把文档不仅仅保存到文件。一个快速的把 XML 转换成可阅读的格式的方法是将它保存到控制台，如下：

```
$phoneBook.Save([Console]::Out)
```

最后，我们将 XML 回写到原来的文件中。

8.4 轻松导入和导出结构化数据

问题

你有一组数据（如一个哈希表或数组），并且希望将其保存到磁盘，以便可以在以后使用它。相反地，你已经把结构化的数据保存到一个文件中了，你要导入它，以便可以使用它。

解决方案

你可以使用 PowerShell 的 `Export-CliXml` 命令把结构化的数据保存到磁盘，使用 `Import-CliXml` 命令来从磁盘导入文件。

例如，假设在一个哈希表中存储着你喜欢的目录的列表，你可以使用“收藏夹 CD”函数轻松地定位你的系统。例 8-5 显示了这个函数。

例 8-5：要求持续结构化数据的函数

```
PS >$favorites = @{}
PS >$favorites["temp"] = "c:\temp"
PS >$favorites["music"] = "h:\lee\my music"
PS >function fcd {
>>     param([string] $location) Set-Location $favorites[$location]
>> }
>>
PS >Get-Location
Path
HKLM:\software\Microsoft\Windows\CurrentVersion\Run
PS >fcd temp
PS >Get-Location
Path
C:\temp
```

遗憾的是，当你退出 PowerShell 的时候，`$favorites` 变量也会随之消失。

要解决这个问题，你可以在你的配置文件中重新创建 \$favorites 变量，也可以将它直接导出到一个文件。下面的命令假定你已经创建了一个配置文件，并将该文件放在与配置文件相同的位置：

```
PS >$filename = Join-Path (Split-Path $profile) favorites.clixml
PS >$favorites | Export-CliXml $filename
PS >$favorites = $null
PS >$favorites
PS >
```

一旦将他保存到磁盘上，你就可以使用 Import-CliXml 命令重新加载，如例 8-6 所示。

例 8-6：从磁盘文件中恢复结构化数据

```
PS >$favorites = Import-CliXml $filename
PS >$favorites

Name          Value
----          -----
music         h:\lee\my music
temp          c:\temp

PS >fcd music
PS >Get-Location
```

```
Path
-----
H:\lee\My Music
```

讨论

PowerShell 提供了 Export-CliXml 和 Import-CliXml 命令让你方便地将结构化的数据导入和导出到文件。这些命令只存储名称、值和属性数据的基本数据类型来完成此任务。

注意：默认的，PowerShell 保存一级的数据，即那些可以直接访问的简单的属性，其他任意级别的数据都采用纯文本的方式保存。关于如何控制输出的深度，你可以键入 **Get-Help Export-CliXml** 来了解 -Depth 参数的解释。

当你导入通过 Export-CliXml 保存的数据之后，你就可以访问原始数据的属性和值了。PowerShell 会将某些对象转换回它们的完全对象（如 System.DateTime 对象），但是大部分不保留原始对象的功能（例如方法）。

8.5 将一个命令的输出存储到 CSV 文件

问题

你想将一个命令的输出存储在 CSV 文件中供以后处理。当你要导出数据供以后在 PowerShell 外部处理时，这样做是很有用的。

解决方案

使用 Powershell 的 Export-Csv 命令来保存一个命令的输出到 CSV 文件中。例如，按 KB 的编号创建一个应用到系统的补丁清单（Vista 之前的系统中），如下：

```
cd $env:WINDIR  
Get-ChildItem KB*.log | Export-Csv c:\temp\patch_log.csv
```

然后你可以使用 Excel 来查看这个补丁日志，把它用邮件发给其他人，或使用 CSV 文件执行任何你希望的工作。

讨论

CSV 文件格式是用于程序和系统之间交换半结构化数据的最常用的格式之一。

Powershell 的 Export-Csv 命令提供了一种从 Powershell 环境中导出数据的简单的方式，同时仍然允许你保留你的数据结构。当 Powershell 把数据导出到 CSV 的时候，它将为你提供的每个对象创建一行。对于每一行，PowerShell 会在 CSV 文件中创建表示对象属性的列。

值得注意的一点是，CSV 文件的格式只支持平实的字符串来表示属性的值。如果对象的某个属性实际上不是字符串，Powershell 会将它转为字符串。把多样的属性的值转换成字符串，意味着相当数量的信息的丢失。如果你最终的目的是在 Powershell 中再次加载这个未经修改的数据，Export-CliXml 命令提供了一个更好的选择。关于 Export-CliXml 命令的详细的信息，请参见 8.4 节。

关于如何从 CSV 文件向 Powershell 导入数据的更多信息，请参见 8.6 节。

参考

- 8.4 节
- 8.6 节

8.6 从 CSV 文件中导入结构化的数据

问题

你想从一个CSV文件导入结构化的数据。当你要使用由另一个程序或其他人修改的结构化的数据时，此功能很有用。

解决方案

使用 Powershell 的 Import-Csv 命令来从一个 CSV 文件导入结构化的数据。

举例来说，假设你之前按 KB 编号导出了应用于非 Vista 之前系统的补丁目录，如下：

```
cd $env:WINDIR
Get-ChildItem KB*.log | Export-Csv c:\temp\patch_log.csv
```

其他人收到这个CSV文件，只保留那些他们感兴趣的行。你可以拷贝这些补丁文件到一个目录以便共享。

```
PS >Import-Csv C:\temp\patch_log_reviewed.csv | Foreach-Object {
>>     Copy-Item -LiteralPath $_.FullName -Destination c:\temp\sharedlogs\}
>>
```

讨论

如8.5节中提到的，CSV文件的格式是用来在程序与系统之间交换半结构化数据的一种最常见的方式。

通过 Import-Csv 命令，可以把半结构化的数据从其他程序导入到 PowerShell 环境里。当 PowerShell 从 CSV 文件导入数据的时候，它会为 CSV 文件中的每一行创建一个新的对象，对于每个对象，PowerShell 根据 CSV 文件中的列的值为对象创建属性。

注意：解决方案中使用了 Foreach-Object 来传递每个对象给 Copy-Item 命令，对于每一项，使用对象的 FullName 属性作为源路径，然后使用 c:\temp\sharedlogs\ 作为目标路径。

CSV 包括了一个属性 PSPath 表示源路径，多数的 cmdlets 都支持 PSPath 作为 -LiteralPath 的别名参数。这样，我们可以使用下面的写法：

```
PS >Import-Csv C:\temp\patch_log_reviewed.csv |
>>     Copy-Item -Destination c:\temp\sharedlogs\
```

关于这项功能的更多信息，请参见 2.5 节。

值得注意的一点是，CSV文件只支持文本串作为属性的值。当你从CSV文件导入数据的时候，日期类型的属性被当作字符串；数字类型的属性也被当作字符串；其他类型的属性都被当作字符串。这意味着在任何属性上执行的排序操作，总是按字母顺序排序。

如果你的最终目的是从之前PowerShell导出的文件中加载未经修改的数据，Import-Clixml提供了更好的选择。关于Import-Clixml的更多信息，请参见8.4节。

关于如何从PowerShell把数据导出到一个CSV文件的更多信息，请参见8.5节。

参考

- 2.5节
- 8.4节
- 8.5节

8.7 使用Excel管理命令输出

问题

你需要使用Excel来操纵或可视化一个命令的输出。

解决方案

使用PowerShell的Export-Csv命令来把一个命令的输出保存到一个CSV文件，然后在Excel中打开那个CSV文件。如果系统中已经把.CSV文件与Excel关联起来了，当你给Invoke-Item提供了一个.CSV文件作为参数的时候，会自动启动Excel。

例8-7演示如何为当前目录下的子文件夹生成占用磁盘空间的CSV。

例8-7：使用Excel可视化显示系统磁盘使用率

```
PS >$filename = "c:\temp\diskusage.csv"
PS >
PS >$output = Get-ChildItem | Where-Object { $_.PsIsContainer } |
>     Select-Object Name,
>     @{ Name="Size";
>         Expression={ ($_.Get-ChildItem -Recurse |
>             Measure-Object -Sum Length).Sum + 0 } }
>
PS >$output | Export-Csv $filename
PS >
PS >Invoke-Item $filename
```

在 Excel 中，你可以根据你的意愿对数据进行格式化，如图 8-1 所示，我们创建了一个饼状图：

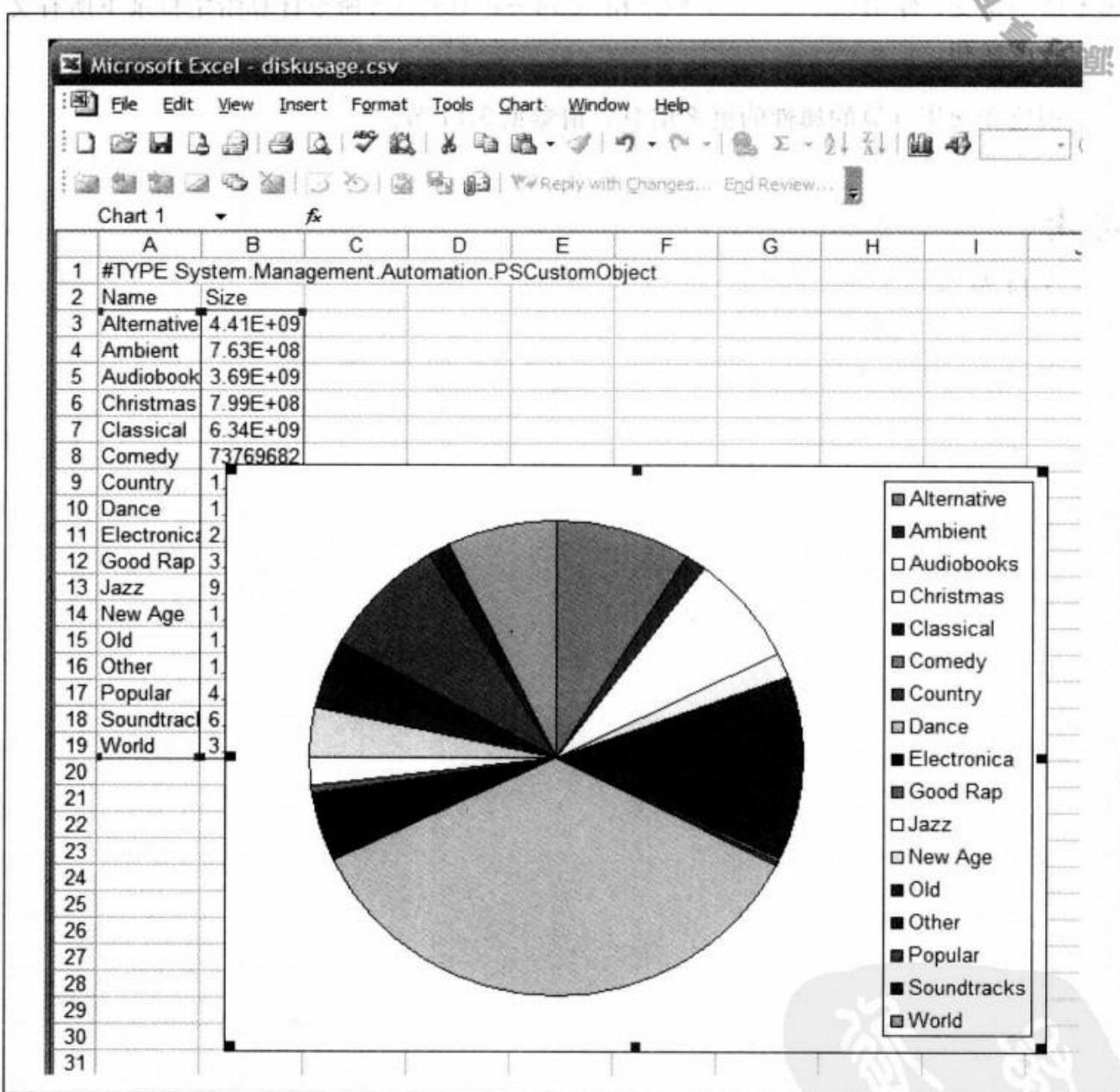


图 8-1：在 Excel 中查看数据

讨论

尽管只用作演示，但例 8-7 把很多内容打包成很少的几行。

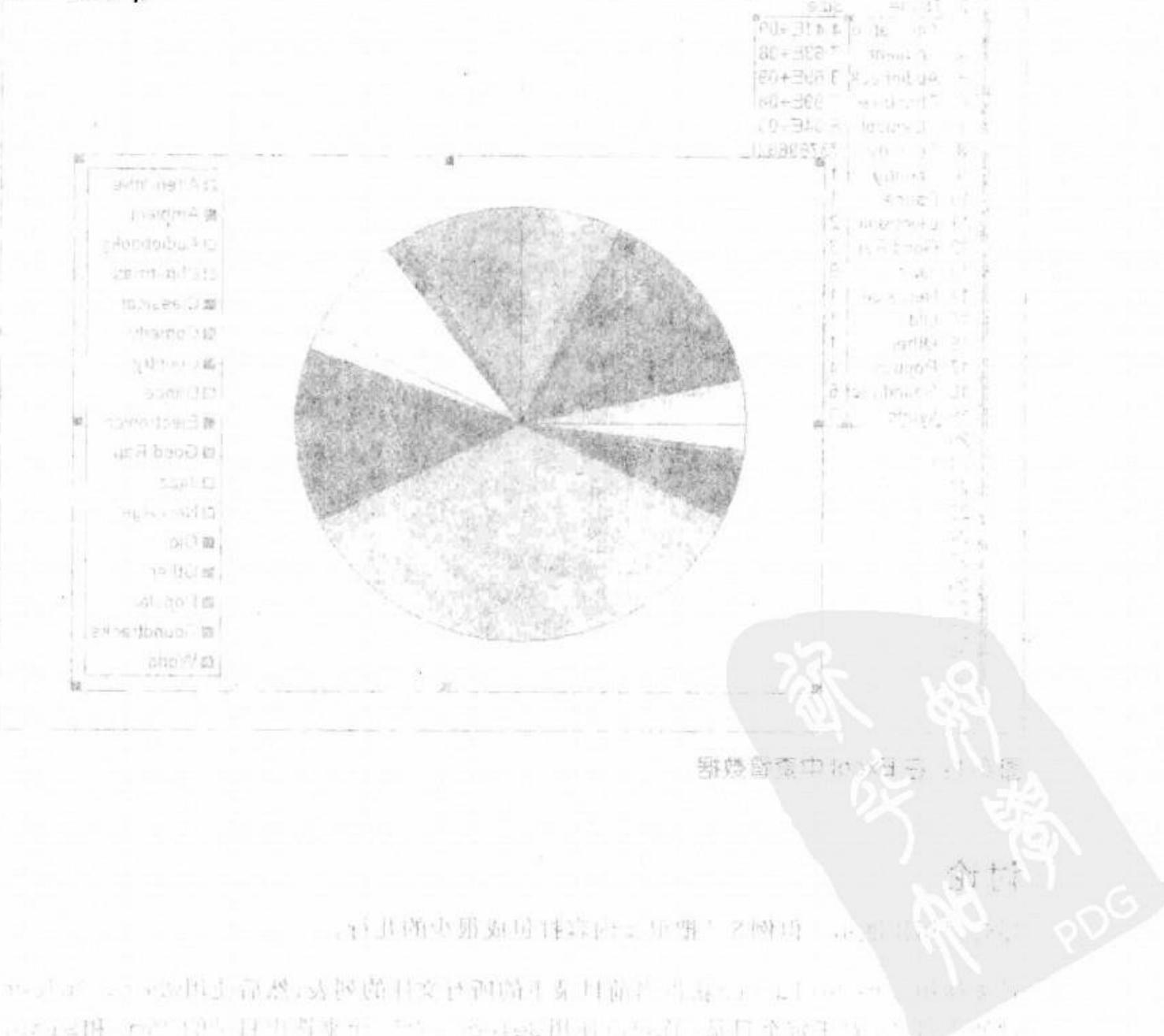
首先使用 `Get-ChildItem` 获得当前目录下的所有文件的列表，然后使用 `Where-Object` 过滤出目录。对于每个目录，你可以使用 `Select-Object` 来选出目录的 `Name` 和 `Size`。

目录本身没有 `Size` 的属性，为了得到这个属性，我们使用 `Select-Object` 哈希表的语法生成一个计算的属性（calculated property）。这个计算的属性（如 Expression 脚本块中定义）使用 `Get-ChildItem` 和 `Measure-Object` 命令计算给定目录下所有文件的长度之和。

关于创建和使用计算的属性的更多信息，请参见 3.11 节。

参考

- 3.11 节



第9章

支持 Internet 的脚本

9.0 简介

当你编写脚本来与本地的系统进行交互操作的时候，PowerShell提供了丰富的功能。同时，它支持操作来自 Internet 的数据，例如，可以从 Internet 下载文件或信息、与 Web 服务进行交互，把你的输出存储为 HTML，甚至以电子邮件形式发送一个长时间运行的脚本的结果。

9.1 从 Internet 下载一个文件

问题

你想从一个 Web 站点下载文件。

解决方案

可以使用.NET框架下的System.Net.WebClient类的DownloadFile()方法来下载文件，如下：

```
PS >$source = "http://www.leeholmes.com/favicon.ico"
PS >$destination = "c:\temp\favicon.ico"
PS >
PS >$wc = New-Object System.Net.WebClient
PS >$wc.DownloadFile($source, $destination)
```

讨论

使用 .NET 框架下的 System.Net.WebClient 类可以非常容易地从 Web 服务器上传与下载文件。

WebClient 类就像是一个浏览器，你可以指定用户类型，代理服务器的信息（如果你的连接需要的话），甚至是认证信息。

所有的浏览器在发送请求的时候，都会发送一个用户代理的标识。这个标识告诉 web 服务器什么样的程序在发送请求，如 Internet Explorer、Firefox，或从搜索引擎发送的请求。许多网站通过这个标识来决定如何显示页面。许多站点因为不能检测到进站请求的用户代理标识而无法正常显示页面。你可以使用 System.Net.WebClient 类标明用户代理标识为 Internet Explorer，如下：

```
$userAgent = "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2;)"  
$wc = New-Object System.Net.WebClient  
$wc.Headers.Add("user-agent", $userAgent)
```

需要注意的是在解决方案中为目标文件使用了完整的路径。这是一个重要的步骤，DownloadFile()方法将文件保存到PowerShell的启动目录中（默认情况下是你的用户配置文件目录的根目录）。

你可以使用 DownloadFile() 方法来下载一个 Web 页面，就像是下载文件一样，你只需要提供一个 URL 地址作为源地址。如果你最终的目的是打算分析或读取下载的页面，DownloadString() 可能是更合适的选择。

关于如何下载 web 页面的详细信息，请参见 9.2 节。

参考

- 9.2 节

9.2 从 Internet 下载一个 Web 页面

问题

你想从 Internet 下载一个 Web 页面，将页面的内容当做文本字符串使用。

解决方案

使用 .Net 框架下的 System.Net.WebClient 类的 DownloadString() 方法来把一个 Web 页面或文本文件下载到一个字符串中，如下：

```
PS >$source = "http://blogs.msdn.com/powershell/rss.xml"
PS >
PS >$wc = New-Object System.Net.WebClient
PS >$content = $wc.DownloadString($source)
```

讨论

尽管 Web 服务正在变得日益流行，但与能够显示有用数据的网页相比，它们还是不常见的。因此，从 Internet 上的服务中检索数据通常出现在屏幕抓取 (*screen scraping*)：下载该网页的 HTML，然后从大多数你不需要的内容中仔细分离出你需要内容。

注意： 屏幕抓取的技术比 Internet 的历史要长。计算机的输出主要是供人使用，屏幕抓取工具使得输出可以被其他计算机程序使用。

遗憾的是，屏幕抓取在提取内容的时候容易产生错误。如果网页作者更改了 HTML，你的代码通常将停止正常工作。如果站点的 HTML 是采用有效的 XHTML 编写的，你可以使用 PowerShell 内置的对 XML 的支持更方便地分析内容。

有关 PowerShell 的内置对 XML 的支持的详细信息，请参阅 8.1 节。

除了其脆弱性，纯屏幕捕获通常是唯一的可选项。在例 9-1 中，你可以使用这种方法来轻松地从 MSN 搜索中获取 Encarta 的即时答案。如果当你运行该脚本时它不再工作，我表示歉意，因为它会演示屏幕捕获的危险。

例 9-1：Get-Answer.ps1

```
#####
## Get-Answer.ps1
##
## Use Encarta's Instant Answers to answer your question
##
## Example:
##   Get-Answer "What is the population of China?"
#####
param([string] $question = $( throw "Please ask a question."))

function Main
{
    ## Load the System.Web.HttpUtility DLL, to let us URLEncode
    [void] [System.Reflection.Assembly]::LoadWithPartialName("System.Web")
```

```
## Get the web page into a single string with newlines between
## the lines.
$encoded = [System.Web.HttpUtility]::UrlEncode($question)
$url = "http://search.live.com/results.aspx?q=$encoded"
$text = (new-object System.Net.WebClient).DownloadString($url)

## Get the answer with annotations
$startIndex = $text.IndexOf('<span class="answer_header">')
$endIndex = $text.IndexOf('function YNC')

## If we found a result, then filter the result
if(($startIndex -ge 0) -and ($endIndex -ge 0))
{
    $partialText = $text.Substring($startIndex, $endIndex - $startIndex)

    ## Very fragile screen scraping here
    $pattern = '<script.+?<div (id="results")|class="answer_fact_body">'
    $partialText = $partialText -replace $pattern, ``n``n
    $partialText = $partialText -replace '<span class="attr.?.?.?">', ``n``n
    $partialText = $partialText -replace '<BR ?/>', ``n``n

    $partialText = clean-html $partialText
    $partialText = $partialText -replace ``n``n, ``n``n
    $partialText = ``n`` + $partialText.Trim()
}
else
{
    $partialText = ``nNo answer found.`n
}

## Clean HTML from a text chunk
function clean-html ($htmlInput)
{
    $tempString = [Regex]::Replace($htmlInput, "<[^>]*>", "")
    $tempString.Replace("&nbsp;&nbsp;", "")
}

. Main
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节
- 8.1 节

9.3 编程：获得页面中的超级链接

当你使用 HTML 的时候，通常需要编写高级的正则表达式来将你需要的内容分离出来。一个典型的例子是从网页中提取所有 HTML 链接。

链接有许多形式，这取决于你希望的形式。它们可能会根据各种 HTML 标准进行组织。它们可能使用相对路径，或使用绝对路径。它们可能会使用双引号或单引号将 URL 引起来。如果你真的不够幸运，可能会遇到只有一侧包括引号的 URL。

例9-2 演示了一些处理这类高级分析任务的方法。这里给出一个你已经从 Internet 下载的 Web 页，它会提取页面中所有链接，返回一个 URL 的列表。它还修复相对的 URL（例如，/file.zip），包含服务器原来所在的地址。

例 9-2：Get-PageUrls.ps1

```
#####
## Get-PageUrls.ps1
##
## Parse all of the URLs out of a given file.
##
## Example:
##     Get-PageUrls microsoft.html http://www.microsoft.com
##
#####
param(
    ## The filename to parse
    [string] $filename = $(throw "Please specify a filename."),
    ## The URL from which you downloaded the page.
    ## For example, http://www.microsoft.com
    [string] $base = $(throw "Please specify a base URL."),
    ## The Regular Expression pattern with which to filter
    ## the returned URLs
    [string] $pattern = ".*"
)

## Load the System.Web DLL so that we can decode URLs
[void] [Reflection.Assembly]::LoadWithPartialName("System.Web")

## Defines the regular expression that will parse an URL
## out of an anchor tag.
$regex = "<\s*a\s*[>]*?href\s*=\s*[`'"]*([`'"]+(>)+)[^>]*?>"

## Parse the file for links
function Main
{
    ## Do some minimal source URL fixups, by switching backslashes to
    ## forward slashes
    $base = $base.Replace("\", "/")
```

```
if($base.IndexOf(":/") -lt 0)
{
    throw "Please specify a base URL in the form of " +
        "http://server/path_to_file/file.html"
}

## Determine the server from which the file originated. This will
## help us resolve links such as "/somefile.zip"
$base = $base.Substring(0,$base.LastIndexOf("/") + 1)
$baseSlash = $base.IndexOf("/", $base.IndexOf(":/") + 3)
$domain = $base.Substring(0, $baseSlash)

## Put all of the file content into a big string, and
## get the regular expression matches
$content = [String]::Join(' ', (get-content $filename))
$contentMatches = @(GetMatches $content $regex)

foreach($contentMatch in $contentMatches)
{
    if(-not ($contentMatch -match $pattern)) { continue }

    $contentMatch = $contentMatch.Replace("\\", "/")

    ## Hrefs may look like:
    ## ./file
    ## file
    ## ../../file
    ## /file
    ## url
    ## We'll keep all of the relative paths, as they will resolve.
    ## We only need to resolve the ones pointing to the root.
    if($contentMatch.IndexOf(":/") -gt 0)
    {
        $url = $contentMatch
    }
    elseif($contentMatch[0] -eq "/")
    {
        $url = "$domain$contentMatch"
    }
    else
    {
        $url = "$base$contentMatch"
        $url = $url.Replace("./", "/")
    }

    ## Return the URL, after first removing any HTML entities
    [System.Web.HttpUtility]::HtmlDecode($url)
}

function GetMatches([string] $content, [string] $regex)
{
    $returnMatches = New-Object System.Collections.ArrayList

    ## Match the regular expression against the content, and
    ## add all trimmed matches to our return list
```

```
$resultingMatches = [Regex]::Matches($content, $regex, "IgnoreCase")
foreach($match in $resultingMatches)
{
    $cleanedMatch = $match.Groups[1].Value.Trim()
    [void] $returnMatches.Add($cleanedMatch)
}

$returnMatches
}

Main
```

关于运行脚本的更多信息，请参见 1.1 节。

参考

- 1.1 节

9.4 编程：调用 Web 服务

尽管屏幕抓取（分析网页的 HTML）是最常用的从 Internet 获得数据的方法，Web 服务正在成为日益常见的服务类型。在对 HTML 进行分析的时候，Web 服务显示出了重要的优势，因为当网站的设计者在设计时候有微小改动，不会影响分析的过程。

当然，使用 Web 服务唯一的好处不是它的更稳定的接口。当使用 web 服务的时候，.NET 框架允许你生成代理（proxies），让你和 web 服务的交互变得和使用传统的.NET 对象一样容易。对于你来说，这些代理的操作几乎完全与任何其他.NET 对象相同。想要调用 web 服务的一个方法，只需要调用代理上的方法即可。

注意：当你使用一个 web 服务的时候，你会注意到与使用一个传统的.NET 对象之间的主要的差异是速度和联网要求。根据实际情况，调用一个 web 服务的代理可能要几秒钟完成。如果你的计算机联网的速度慢，调用可能会返回一个网络错误（如超时），而不是返回你期望的信息。

如果你知道 web 服务描述文件（WSDL）的地址，例 9-3 使你可以与远端的 web 服务建立连接。它会为你生成 web 服务的代理，然后你可以与代理进行交互，就像与其他任意的.NET 对象交互一样。

例 9-3：Connect-WebService.ps1

```
#####
## Connect-WebService.ps1
##
```

```
## Connect to a given web service, and create a type that allows you to
## interact with that web service.
##
## Example:
##
##     $wsdl = "http://terraserver.microsoft.com/TerraService2.asmx?WSDL"
##     $terraServer = Connect-WebService $wsdl
##     $place = New-Object Place
##     $place.City = "Redmond"
##     $place.State = "WA"
##     $place.Country = "USA"
##     $facts = $terraServer.GetPlaceFacts($place)
##     $facts.Center
#####
param(
    [string] $wsdlLocation = $(throw "Please specify a WSDL location"),
    [string] $namespace,
    [Switch] $requiresAuthentication)

## Create the web service cache, if it doesn't already exist
if(-not (Test-Path Variable:\Lee.Holmes.WebServiceCache))
{
    ${GLOBAL:Lee.Holmes.WebServiceCache} = @{}
}

## Check if there was an instance from a previous connection to
## this web service. If so, return that instead.
$oldInstance = ${GLOBAL:Lee.Holmes.WebServiceCache}[$wsdlLocation]
if($oldInstance)
{
    $oldInstance
    return
}
## Load the required Web Services DLL
[void] [Reflection.Assembly]::LoadWithPartialName("System.Web.Services")

## Download the WSDL for the service, and create a service description from
## it.
$wc = New-Object System.Net.WebClient
if($requiresAuthentication)
{
    $wc.UseDefaultCredentials = $true
}
$wsdlStream = $wc.OpenRead($wsdlLocation)

## Ensure that we were able to fetch the WSDL
if(-not (Test-Path Variable:\wsdlStream))
{
    return
}
$serviceDescription =
    [Web.Services.Description.ServiceDescription]::Read($wsdlStream)
$wsdlStream.Close()
```

```
## Ensure that we were able to read the WSDL into a service description
if (-not(Test-Path Variable:\serviceDescription))
{
    return
}

## Import the web service into a CodeDom
$serviceNamespace = New-Object System.CodeDom.CodeNamespace
if($namespace)
{
    $serviceNamespace.Name = $namespace
}

$codeCompileUnit = New-Object System.CodeDom.CodeCompileUnit
$serviceDescriptionImporter =
    New-Object Web.Services.Description.ServiceDescriptionImporter
$serviceDescriptionImporter.AddServiceDescription(
    $serviceDescription, $null, $null)
[void] $codeCompileUnit.Namespaces.Add($serviceNamespace)
[void] $serviceDescriptionImporter.Import(
    $serviceNamespace, $codeCompileUnit)

## Generate the code from that CodeDom into a string
$generatedCode = New-Object Text.StringBuilder
$stringWriter = New-Object IO.StringWriter $generatedCode
$provider = New-Object Microsoft.CSharp.CSharpCodeProvider
$provider.GenerateCodeFromCompileUnit($codeCompileUnit, $stringWriter, $null)

## Compile the source code.
$references = @("System.dll", "System.Web.Services.dll", "System.Xml.dll")
$compilerParameters = New-Object System.CodeDom.Compiler.CompilerParameters
$compilerParameters.ReferencedAssemblies.AddRange($references)
$compilerParameters.GenerateInMemory = $true

$compilerResults =
    $provider.CompileAssemblyFromSource($compilerParameters, $generatedCode)

## Write any errors if generated.
if($compilerResults.Errors.Count -gt 0)
{
    $errorLines = ""
    foreach($error in $compilerResults.Errors)
    {
        $errorLines += "`n`t" + $error.Line + ":"`t" + $error.ErrorText
    }

    Write-Error $errorLines
    return
}
## There were no errors. Create the web service object and return it.
else
{
    ## Get the assembly that we just compiled
    $assembly = $compilerResults.CompiledAssembly

    ## Find the type that had the WebServiceBindingAttribute.
```

```
## There may be other "helper types" in this file, but they will
## not have this attribute
$type = $assembly.GetTypes() |
    Where-Object { $_.GetCustomAttributes(
        [System.Web.Services.WebServiceBindingAttribute], $false) }

if(-not $type)
{
    Write-Error "Could not generate web service proxy."
    return
}

## Create an instance of the type, store it in the cache,
## and return it to the user.
$instance = $assembly.CreateInstance($type)

## Many services that support authentication also require it on the
## resulting objects
if($requiresAuthentication)
{
    if(@( $instance.PSObject.Properties |
        where { $_.Name -eq "UseDefaultCredentials" }).Count -eq 1)
    {
        $instance.UseDefaultCredentials = $true
    }
}
${GLOBAL:Lee.Holmes.WebServiceCache}[$wsdlLocation] = $instance
```

关于运行脚本的更多信息，请参见 1.1 节。

参考

- ## • 1.1 节

9.5 将命令的输出生成一个 Web 页面

你得把一个命令的结果输出为一个 web 页面，这样可以把它发布到一个 web 服务器上。

解决方案

使用 PowerShell 的 `ConvertTo-HTML` 命令来把命令的输出转换到一个 Web 页面上。比如，创建一个 PowerShell 命令的概要的 HTML 文件，可以输入：

```
PS >$filename = "c:\temp\help.html"
PS >
PS >$commands = Get-Command | Where { $_. CommandType -eq "Alias" }
PS >$summary = $commands | Get-Help | Select Name, Synopsis
PS >$summary | ConvertTo-HTML | Set-Content $filename
```

讨论

当你使用 `ConvertTo-HTML` 命令把输出命令的结果导出到一个文件时，PowerShell 会生成一个HTML表格来代表命令的输出。表格为你提供的每个对象创建一行。在每行中，PowerShell 会创建列代表对象属性的值。

`ConvertTo-HTML` 命令允许你输入一些参数来自定义表格的样式，来向文件的头和正文中添加自定义的内容。

关于 `ConvertTo-HTML` 命令的更多信息，可以键入 `Get-Help ConvertTo-HTML`。

9.6 编程：发送电子邮件

例 9-4 中展示了通过脚本中发送电子邮件是如此的简单。

除了脚本中使用的字段，`System.Net.Mail.MailMessage` 类还支持其他一些属性，它可以让你添加附件，设置信息优先级及其他。关于在 .NET Framework 中使用类的更多信息，请参阅 3.4 节。

例 9-4：Send-MailMessage.ps1

```
#####
## Send-MailMessage.ps1
## Illustrate the techniques used to send an email in PowerShell.
## Example:
## PS >$body = @"
## >> Hi from another satisfied customer of The PowerShell Cookbook!
## >> "@
## >>
## PS >$to = "guide_feedback@leeholmes.com"
## PS >$subject = "Thanks for all of the scripts."
## PS >$mailHost = "mail.leeholmes.com"
## PS >Send-MailMessage $to $subject $body $mailHost
##
param()
```

```
[string[]] $to = $(throw "Please specify the destination mail address"),
[string] $subject = "<No Subject>",
[string] $body = $(throw "Please specify the message content"),
[string] $smtpHost = $(throw "Please specify a mail server."),
[string] $from = "$($env:UserName)@example.com"
}

## Create the mail message
$email = New-Object System.Net.Mail.MailMessage

## Populate its fields
foreach($mailTo in $to)
{
    $email.To.Add($mailTo)
}

$email.From = $from
$email.Subject = $subject
$email.Body = $body

## Send the mail
$client = New-Object System.Net.Mail.SmtpClient $smtpHost
$client.UseDefaultCredentials = $true
$client.Send($email)
```

关于运行脚本的更多信息，请参见 1.1 节。

参考

- 1.1 节
- 3.4 节

9.7 编程：与 Internet 协议交互

虽然工作在 Web 站点和 Web 服务这样的抽象级别是常见的。但有时也需要编写一个完全独立的 Internet 脚本来与远程计算机在更低的级别进行交互。这个级别（称作 TCP 级，传输控制协议，即 *Transmission Control Protocol*）是多数 Internet 协议——如 Telnet、SMTP（发送邮件）、POP3（接收邮件）、HTTP 的通信基础。

.NET 框架提供了一些类，允许你和一些 internet 协议直接交互：如 `System.Web.Mail.SmtpMail` 类支持 SMTP 协议；`System.Net.WebClient` 类支持 HTTP 协议及其他一些协议。当.NET 框架不支持一个你需要的 Internet 协议的时候，如果你知道它的工作原理，你也可以在脚本中与应用程序协议直接交互。

例 9-5 展示了如何接收在一个远程 POP3 中等待的邮件的信息，使用了例 9-6 中的 `Send-TcpRequest` 脚本。

例 9-5：与远程的 POP3 邮箱交互

```
## Get the user credential
if(-not (Test-Path Variable:\mailCredential))
{
    $mailCredential = Get-Credential
}
$address = $mailCredential.UserName
$password = $mailCredential.GetNetworkCredential().Password

## Connect to the remote computer, send the commands, and receive the
## output
$pop3Commands = "USER $address", "PASS $password", "STAT", "QUIT"
$output = $pop3Commands | Send-TcpRequest mail.myserver.com 110
$inbox = $output.Split("`n")[3]

## Parse the output for the number of messages waiting and total bytes
$status = $inbox |
    Convert-TextObject -PropertyName "Response", "Waiting", "BytesTotal", "Extra"
"{0} messages waiting, totaling {1} bytes." -f $status.Waiting, $status.BytesTotal
```

在例 9-5 中，你连接到远端的邮件服务器的 110 端口。你可以以一种邮件服务器可以理解的形式发送命令来查看邮箱的状态。这个网络会话的格式是标准的 POP3 协议规定好的。例 9-5 中使用的 Convert-TextObject 命令是在 5.12 节中提供的。

例 9-6 提供了核心函数，这让你可以轻松的使用纯文本格式的 TCP 协议。

例 9-6：Send-TcpRequest.ps1

```
#####
## Send-TcpRequest.ps1
##
## Send a TCP request to a remote computer, and return the response.
## If you do not supply input to this script (via either the pipeline, or the
## -InputObject parameter,) the script operates in interactive mode.
##
## Example:
##
##     $http = @"
##         GET / HTTP/1.1
##         Host:search.msn.com
##         `n`n
##         "@
##
##     $http | Send-TcpRequest search.msn.com 80
#####
param(
    [string] $remoteHost = "localhost",
    [int] $port = 80,
    [string] $inputObject,
    [int] $commandDelay = 100
)
```

```
[string] $output = ""

## Store the input into an array that we can scan over. If there was no input,
## then we will be in interactive mode.
$currentInput = $inputObject
if(-not $currentInput)
{
    $SCRIPT:currentInput = @($input)
}
$scriptedMode = [bool] $currentInput

function Main
{
    ## Open the socket, and connect to the computer on the specified port
    if(-not $scriptedMode)
    {
        Write-Host "Connecting to $remoteHost on port $port"
    }
    trap { Write-Error "Could not connect to remote computer: $_"; exit }
    $socket = New-Object System.Net.Sockets.TcpClient($remoteHost, $port)
    if(-not $scriptedMode)
    {
        Write-Host "Connected. Press ^D followed by [ENTER] to exit.\n"
    }

    $stream = $socket.GetStream()
    $writer = New-Object System.IO.StreamWriter($stream)

    ## Create a buffer to receive the response
    $buffer = New-Object System.Byte[] 1024
    $encoding = New-Object System.Text.AsciiEncoding

    while($true)
    {
        ## Receive the output that has buffered so far
        $SCRIPT:output += GetOutput

        ## If we're in scripted mode, send the commands,
        ## receive the output, and exit.
        if($scriptedMode)
        {
            foreach($line in $currentInput)
            {
                $writer.WriteLine($line)
                $writer.Flush()
                Start-Sleep -m $CommandDelay
                $SCRIPT:output += GetOutput
            }

            break
        }
        ## If we're in interactive mode, write the buffered
        ## output, and respond to input.
```

```
else
{
    if($output)
    {
        foreach($line in $output.Split("`n"))
        {
            Write-Host $line
        }
        $SCRIPT:output = ""
    }

    ## Read the user's command, quitting if they hit ^D
    $command = Read-Host
    if($command -eq ([char] 4)) { break; }

    ## Otherwise, write their command to the remote host
    $writer.WriteLine($command)
    $writer.Flush()
}
}

## Close the streams
$writer.Close()
$stream.Close()

## If we're in scripted mode, return the output
if($scriptedMode)
{
    $output
}

## Read output from a remote host
function GetOutput
{
    $outputBuffer = ""
    $foundMore = $false

    ## Read all the data available from the stream, writing it to the
    ## output buffer when done.
    do
    {
        ## Allow data to buffer for a bit
        Start-Sleep -m 1000

        ## Read what data is available
        $foundmore = $false
        while($stream.DataAvailable)
        {
            $read = $stream.Read($buffer, 0, 1024)
            $outputBuffer += ($encoding.GetString($buffer, 0, $read))
            $foundmore = $true
        }
    } while($foundmore)
```

SoutputBuffer

)

Main**关于运行脚本的更多信息，请参见 1.1 节。****参考**

- **1.1 节** *从命令行运行脚本，或从 Java 程序中调用它们* [参见第 4 章](#)
从命令行运行脚本 [参见第 1.1 节](#)
- **5.12 节** *从命令行运行脚本，或从 Java 程序中调用它们* [参见第 4 章](#)
从命令行运行脚本 [参见第 1.1 节](#)

从命令行运行脚本 [参见第 1.1 节](#)**从命令行运行脚本** [参见第 4 章](#)**从命令行运行脚本** [参见第 1.1 节](#)**从命令行运行脚本** [参见第 4 章](#)

代码复用

10.0 简介

令人惊奇的是，在PowerShell的交互提示符下，大多数人都能完成任务，自PowerShell可以非常容易地将其强大的命令合并成功能更加强大的命令以来，交互式意味着要简洁。实际上，大多数脚本交互式的核心为多数简洁的表达式：*one-liners*。

尽管交互的效率很高，但是你显然不想在每次需要它们时候重新输入，当你要保存或重新使用你已经编写的命令的时候，PowerShell提供许多途径支持进行代码的重用，包括：脚本、库、函数、脚本块和更多其他方式。

10.1 编写一个脚本

问题

你想将命令存储在一个脚本中，这样可以与其他人共享或在以后使用。

解决方案

你可以用你喜欢的文本编辑器创建一个纯文本的文件，然后把命令（与你在交互式外壳中使用的命令相同）加入到文件中，保存文件的扩展名为`.ps1`。

讨论

有关PowerShell，要记住的最重要的一件事情是运行脚本和在命令行中运行命令本质上是等效的操作。你可以把脚本中的命令拷贝到命令行中运行，也可以把命令行中的命令粘贴到脚本中执行。

当你编写完脚本后，你可以在 PowerShell 中像调用其他程序一样调用脚本。运行一个脚本与逐条运行脚本中的命令是一样的。

注意：PowerShell 引入了几个与运行脚本和工具有关系的功能，如果你不知道，它们可能会在开始混淆你。有关如何调用脚本和现有的工具的更多信息，请参见 1.1 节。

当你在 PowerShell 中第一次运行脚本的时候，PowerShell 会提示以下的错误信息：

```
File c:\tools\myFirstScript.ps1 cannot be loaded because the execution of scripts is disabled on this system. Please see "get-help about_signing" for more details.  
At line:1 char:12  
+ myFirstScript <<<<
```

既然很少的计算机用户会编写脚本，那么 PowerShell 默认的安全策略会阻止脚本运行。当然，在你开始编写脚本后，你就可以配置这个策略来减少限制。关于如何配置你机器上的执行策略，可以参见 16.1 节。

当你给你的脚本命名的时候，最好挑选一个能够表达脚本内容的名字，这样当你一看到名称的时候就可以大概了解脚本所要完成的功能。PowerShell 采用的命名方式值得借鉴，它采用动-名词的模式，动词代表要执行的操作，名词代表要操作的项。作为这一理念的用途的示例，考虑例 10-1 中给出的典型的 Windows 命令的名字：

例 10-1：一些标准的 Windows 命令的名字

```
PS >dir $env:WINDIR\System32\*.exe | Select-Object Name
```

Name
accwiz.exe
actmovie.exe
ahui.exe
alg.exe
append.exe
arp.exe
asr_fmt.exe
asr_ldm.exe
asr_pfu.exe
at.exe
atadm.exe
attrib.exe
(...)

把上面例子中的名字与例 10-2 中一些标准的 Windows PowerShell 命令 比较一下

例 10-2：一些标准的 Windows PowerShell 命令的名字

```
PS >Get-Command | Select-Object Name
```

```
Name
-----
Add-Content
Add-History
Add-Member
Add-PSSnapin
Clear-Content
Clear-Item
Clear-ItemProperty
Clear-Variable
Compare-Object
ConvertFrom-SecureString
Convert-Path
ConvertTo-HTML
(...)
```

这样的命令命名方式加快了查找命令的速度，PowerShell 采用这样的命名原则使得你可以用少于 50 个的动词来管理你 80% 的系统任务。一旦你理解了一个动词的含义（比如作为读取、打开等标准的动词 Get），你通常就可以猜到该命令的含义。

当你给你的脚本命名的时候，尤其是要与其他人共享这个脚本的时候，你需要尽一切努力选择一个符合这些规则的名称。附录 D 中的“标准的 PowerShell 动词”中提供了一个 PowerShell 的标准动词的列表，可以帮助你准确的给你的脚本命名。作为证据，可以思考一下本书中包含的一些脚本的命名：

```
PS >dir | select Name

Name
-----
Compare-Property.ps1
Connect-WebService.ps1
Convert-TextObject.ps1
Get-AliasSuggestion.ps1
Get-Answer.ps1
Get-Characteristics.ps1
Get-OwnerReport.ps1
Get-PageUrls.ps1
Invoke-CmdScript.ps1
New-GenericObject.ps1
Select-FilteredObject.ps1
(...)
```

像 PowerShell 中的命令的命名一样，这些脚本的名称清楚，很容易了解，并且使用 PowerShell 的标准动词列表中的动词。

参考

- 1.1 节

- 16.1 节
- 附录 D 的标准 PowerShell 动词

10.2 编写一个函数

问题

你想多次调用脚本或你认为脚本中的部分可以作为工具重复使用。

解决方案

将这些常用的代码放在一个函数中，然后调用该函数。下面脚本中摄氏的转换代码

```
param([double] $fahrenheit)

## Convert it to Celsius
$celsius = $fahrenheit - 32
$celsius = $celsius / 1.8
## Output the answer
"$fahrenheit degrees Fahrenheit is $celsius degrees Celsius."
```

可以放在脚本的一个函数中：

```
param([double] $fahrenheit)

## Convert Fahrenheit to Celsius
function ConvertFahrenheitToCelsius([double] $fahrenheit)
{
    $celsius = $fahrenheit - 32
    $celsius = $celsius / 1.8
    $celsius
}

$celsius = ConvertFahrenheitToCelsius $fahrenheit

## Output the answer
"$fahrenheit degrees Fahrenheit is $celsius degrees Celsius."
```

尽管使用函数使得脚本变得更长和更难理解了，但是这个技术是非常有价值的。

讨论

当你定义好一个函数后，任何在这个定义后面的命令都可以使用它。也就是说，你必须在使用函数的脚本之前来定义你的函数。你会发现如果你的脚本中定义了很多函数，一

些函数的定义可能会影响脚本的主逻辑部分。如果出现这类问题，你可以把主逻辑部分放到“Main”函数中，就像附录A中“编写脚本，复用函数”中所描述的一样。

注意：对于那些习惯于使用批处理命令的人来说，一个常见的问题就是在PowerShell中是否有与GOTO等效的命令。如果GOTO是用于调用子程序或批处理文件中的其他独立的帮助器的话，你可以使用PowerShell函数来完成该任务。如果GOTO是用在循环中，那么PowerShell的循环机制就更合适了。

在PowerShell中，调用一个函数感觉上就像调用一个命令或一个脚本。作为用户，你不用了解一个小的帮助程序是用cmdlet、脚本还是函数写的。当你调用一个函数的时候，你只需要在函数名后面输入相应的参数，并使用空格分隔每个参数。这与你调用许多编程语言（如C#）中的函数方式形成对比，在C#中，函数名后使用括号，每个参数之间用逗号分隔。

此外，注意从函数返回的值可以是任何他要写入到输出管道的内容（如解决方案中的\$celsius）。如果你想，你可以写return\$celsius，但那是不需要的。

关于如何编写函数的更多信息，请参见附录A中的“编写脚本，复用函数”。关于PowerShell中循环语句的更多信息，请参见4.4节。

参考

- 4.4节
- 附录A中的“编写脚本，复用函数”

10.3 编写一个脚本块

问题

你有一个脚本片断，除了在处理逻辑上有点不同之外，输入基本相同。

解决方案

如例10-3中所示，把这点不同的逻辑放到一个脚本块中，然后将脚本块作为一个参数传给需要它的代码。使用调用操作符（&）来执行这个脚本块。

例 10-3：一个在管道中应用脚本块的脚本

```
#####
## Map-Object.ps1
##
## Apply the given mapping command to each element of the input
##
## Example:
## 1,2,3 | Map-Object{ $_ * 2 }
## param([ScriptBlock] $mapCommand)
```

```
process
{
    & $mapCommand
}
```

讨论

假设在脚本中需要对一个列表中的所有元素进行 *2 的操作，如下：

```
function MultiplyInputByTwo
{
    process
    {
        $_ * 2
    }
}
```

同时，他还需要执行一个更复杂的计算，如下：

```
function MultiplyInputComplex
{
    process
    {
        ($_ + 2) * 3
    }
}
```

这两个函数除了执行计算的那行代码不同之外其余的都非常相似。随着我们添加的计算的增多，这个问题变得更加明显：添加每个新的七行函数仅仅是为了返回一行值。

```
PS >1,2,3 | MultiplyInputByTwo
2
4
6
PS >1,2,3 | MultiplyInputComplex
9
12
15
```

如果我们用脚本块来替换这部分未知的计算，我们则不需要添加新的函数，如下：

```
PS >1,2,3 | Map-Object { $_. * 2 }
2
4
6
PS >1,2,3 | Map-Object {($_ + 2) * 3 }
9
12
15
PS >1,2,3 | Map-Object {($_ + 3) * $_ }
4
10
18
```

事实上，`Map-Object`提供的功能是非常有用的，它是一个标准的PowerShell命令，称为`Foreach-Object`。有关脚本块的更多信息，请参阅附录A中的“编写脚本，复用函数”。有关运行脚本的详细信息，请参阅1.1节。

参考

- 第1.1节
- 附录A中的“编写脚本，复用函数”

10.4 从脚本、函数、脚本块返回数据

问题

你需要调用脚本或函数，并返回调用它的数据。

解决方案

为了从一个脚本或函数返回数据，需要把该数据写入输出管道，如下：

```
## Get-Tomorrow.ps1
## Get the date that represents tomorrow

function GetDate
{
    Get-Date
}

$tomorrow = (GetDate).AddDays(1)
$tomorrow
```

讨论

在PowerShell中，除非有函数捕获输出，否则所有函数与脚本生成的数据都发送到输出管道。GetDate函数可以生成数据（一个日期值）但并不捕获它，这使它成为该函数的输出。脚本中的命令调用了GetDate函数并获得输出，然后对其进行操作。

最后，脚本把变量\$tomorrow输出给管道，这样变量\$tomorrow的值就成为了脚本的返回值。

注意：一些.NET类的方法会产生输出，如System.Collections.ArrayList类。如果你不希望该类的方法返回值，可以在该类的方法前面加入[void]来避免返回值，如下：

```
PS >$collection = New-Object System.Collections.ArrayList
PS >$collection.Add("Hello")
0
PS >[void] $collection.Add("Hello")
```

虽然遵守“管道的输出成为返回值”的规律，PowerShell仍然支持使用传统的关键字return来从一个函数或脚本中返回值（比如return "hello"）。PowerShell会将"hello"作为一个语句来处理。

注意：如果你想让其他人清楚地了解你的脚本的意图，可以使用Write-Output来明确地把数据发送给管道。两者的输出结果是一样的，不同的只是使用的习惯而已。

如果你向输出管道写入一个集合（如数组），PowerShell实际上会分别把每个元素输出到管道。为了保障集合在输出时保持原样，需要使用逗号前缀，这样会返回一个，保持了原样的集合，如下所示：

```
function WritesObjects
{
    $arrayList = New-Object System.Collections.ArrayList
    [void] $arrayList.Add("Hello")
    [void] $arrayList.Add("World")

    $arrayList
}

function WritesArrayList
{
    $arrayList = New-Object System.Collections.ArrayList
    [void] $arrayList.Add("Hello")
    [void] $arrayList.Add("World")

    , $arrayList
}
```

```
$objectOutput = WritesObjects  
  
# The following command would generate an error  
# $objectOutput.Add("Extra")  
  
$arrayListOutput = WritesArrayList  
$arrayListOutput.Add("Extra")
```

有时候你可能希望使用退出代码来表示脚本执行的成功与否,你可以使用exit关键字。

关于return和exit语句的更多信息,请参见附录A中的“编写脚本,复用函数”和1.10节。

参考

- 1.10节
- 附录A中的“编写脚本,复用函数”

10.5 将常用的函数放到库文件中

问题

你开发了一套有用的函数,想在多个脚本里共享这些函数。

解决方案

首先,把这些常用的函数定义和内容放到一个脚本文件中,文件的名字使用Library作为前缀。实际上Library前缀不是必须的,这么写只是一种好的命名习惯。例10-4演示了这个方法:

例10-4: 温度函数库

```
## LibraryTemperature.ps1  
## Functions that manipulate and convert temperatures  
  
## Convert Fahrenheit to Celsius  
function ConvertFahrenheitToCelsius([double] $fahrenheit)  
{  
    $celsius = $fahrenheit - 32  
    $celsius = $celsius / 1.8  
    $celsius  
}
```

下一步,在需要使用这个函数的脚本中dot-source(使用.(.)格式)引入这个库文件,如例10-5所示:

例 10-5：使用库的脚本

```
param([double] $fahrenheit)

$scriptDirectory = Split-Path $myInvocation.MyCommand.Path
. (Join-Path $scriptDirectory LibraryTemperature.ps1)

$celsius = ConvertFahrenheitToCelsius $fahrenheit

## Output the answer
"$fahrenheit degrees Fahrenheit is $celsius degrees Celsius."
```

讨论

尽管多数用于库文件，你也可以使用 dot-source 用于任何脚本和函数。当你 dot-source 一个脚本或函数的时候，PowerShell 的行为就像调用脚本本身的函数一样。

关于 dot-sourcing 的更多信息，请参见附录 A 中的“编写脚本，复用函数”。

参考

- 附录 A 中的“编写脚本，复用函数”

10.6 脚本、函数或脚本块的访问参数

问题

你想访问脚本、函数或脚本块提供的参数。

解决方案

要按名称访问参数，使用 param 语句，如下：

```
param($firstNamedArgument, [int] $secondNamedArgument = 0)

"First named argument is: $firstNamedArgument"
"Second named argument is: $secondNamedArgument"
```

要按位置访问无名参数，使用 \$args 数组，如下：

```
"First positional argument is: " + $args[0]
"Second positional argument is: " + $args[1]
```

这种方法同样适用于脚本、函数和脚本块，如例 10-6 所示：

例 10-6：带参数的脚本、函数或脚本块

```
#####
## Get-Arguments.ps1
##
## Use command-line arguments
#####
param($firstNamedArgument, [int] $secondNamedArgument = 0)

## Display the arguments by name
"First named argument is: $firstNamedArgument"
"Second named argument is: $secondNamedArgument"

function GetArgumentsFunction
{
## We could use a param statement here, as well
## param($firstNamedArgument, [int] $secondNamedArgument = 0)

## Display the arguments by position
"First positional function argument is: " + $args[0]
"Second positional function argument is: " + $args[1]
}

GetArgumentsFunction One Two

$scriptBlock =
{
    param($firstNamedArgument, [int] $secondNamedArgument = 0)

    ## We could use $args here, as well
    "First named scriptblock argument is: $firstNamedArgument"
    "Second named scriptblock argument is: $secondNamedArgument"
}

& $scriptBlock -First One -Second 4.5
```

例 10-6 执行结果如下：

```
PS >Get-Arguments First 2
First named argument is: First
Second named argument is: 2
First positional function argument is: One
Second positional function argument is: Two
First named scriptblock argument is: One
Second named scriptblock argument is: 4
```

讨论

尽管 PowerShell 同时支持 `param` 关键字和 `$args` 数组两种方式来访问参数，但你通常都会使用 `param` 关键字来定义和访问脚本、函数和脚本块的参数。

注意：在多数编程语言中，使用\$args风格的数组访问参数的最常见的原因是确定当前运行的脚本的名字。关于如何实现这项技术，可以参见 14.2 节。

当你使用 param 关键字来定义参数的时候，PowerShell 为你的脚本或函数提供了许多有用的功能，它允许用户像使用 cmdlets 一样使用你的脚本，这些功能包括：

- 用户可以通过参数名称来区分参数
- 用户可以清楚了解参数的含义
- 你可以指定参数的类型，这样 PowerShell 可以在需要的时候转换输入值
- 你可以指定参数的默认值

有时候，\$args 数组也是有用的，比如说可以一次处理所有的参数，例如：

```
function Reverse
{
    $argsEnd = $args.Length - 1
    $args[$argsEnd..0]
}
```

输出结果：

```
PS >Reverse 1 2 3 4
4
3
2
1
```

关于 param 语句的更多信息，可以参见附录 A 中的“编写脚本，复用函数”。关于运行脚本的更多信息，可以参见 1.1 节。

参考

- 14.2 节
- 1.1 节
- 附录 A 中的“编写脚本，复用函数”

10.7 访问管道输入 问题

你想与通过管道发送给你的函数、脚本或脚本块的用户输入之间进行交互。

解决方案

要访问管道输入的数据，可以使用例 10-7 中的 \$input 变量，如下：

例 10-7：访问管道输入数据

```
function InputCounter
{
    $count = 0

    ## Go through each element in the pipeline, and add up
    ## how many elements there were.
    foreach($element in $input)
    {
        $count++
    }

    $count
}
```

如果是在 Windows 的系统目录下运行，结果如下（视实际情况而定）：

```
PS >dir $env:WINDIR | InputCounter
295
```

讨论

在你的脚本、函数或脚本块中，变量 \$input 代表一个用户通过管道输入的枚举值 (enumerator)（与简单数组相对而言）。枚举值可以让你使用 foreach 语句有效地扫描输入的元素（如例 10-7 所示），但是不允许你直接访问其中某项的值（比如，输入中的第 5 个元素）。

注意：一个枚举值只允许你向前浏览它的内容。一旦你访问了枚举中的一个元素，PowerShell 会自动移到下一个元素。如果你需要访问一个已经访问过的元素，你必须调用 \$input.Reset() 来从头开始搜索，或者把输入值保存到一个数组中。

如果你需要访问输入值中指定的元素（或多次访问某项），最佳的做法是把输入值保存到数组当中。这样就避免了脚本使用 \$input 枚举流的行为，有时这也是唯一的选择。要把输入保存到一个数组，使用 PowerShell 的列表赋值语法 (@()) 来强制 PowerShell 将它视为一个数组，例如：

```
function ReverseInput
{
    $inputArray = @($input)
    $inputEnd = $inputArray.Count - 1
```

```
$inputArray[$inputEnd..0]  
}
```

输出结果如下：

```
PS > 1,2,3,4 | ReverseInput  
4  
3  
2  
1
```

如果管道输入的处理在你的脚本、函数或脚本块中扮演很重要的角色，PowerShell 还提供了一个可选的方法来处理管道输入，这样可以使你的脚本更容易书写理解。更多信息请参见 10.8 节。

参考

- 10.8 节

10.8 用命令关键字 (Cmdlet Keywords) 编写面向管道的脚本

问题 你在编写脚本时希望将脚本分成几个部分，每部分的功能各不相同，但又希望它们能很好地协作。解决方案 你可以使用 begin、process 和 end 关键字，将你的脚本分成初始化、处理和清除几个区域。比如，可以把 10.7 节中的例子转换成面向管道的方式，如例 10-8 所示：

例 10-8：面向管道的使用命令关键字的脚本

```
function InputCounter  
{  
    begin  
    {  
        $count = 0  
    }  
    ## Go through each element in the pipeline, and add up  
    ## how many elements there were.  
    process
```

```
{  
    Write-Debug "Processing element $_"  
    $count++  
}  
  
end  
{  
    $count  
}  
}
```

输出结果如下：

```
PS >$debugPreference = "Continue"  
PS >dir | InputCounter  
DEBUG: Processing element Compare-Property.ps1  
DEBUG: Processing element Connect-WebService.ps1  
DEBUG: Processing element Convert-TextObject.ps1  
DEBUG: Processing element ConvertFrom-FahrenheitWithFunction.ps1  
DEBUG: Processing element ConvertFrom-FahrenheitWithLibrary.ps1  
DEBUG: Processing element ConvertFrom-FahrenheitWithoutFunction.ps1  
DEBUG: Processing element Get-AliasSuggestion.ps1  
(...)  
DEBUG: Processing element Select-FilteredObject.ps1  
DEBUG: Processing element Set-ConsoleProperties.ps1
```

讨论

如果脚本、函数或脚本块主要处理管道的输入，那么 `begin`、`process` 和 `end` 关键字可以使你清楚地表达你的解决方案。阅读你脚本的人（包括你自己）可以清晰地了解你的脚本哪部分处理初始化，哪部分处理记录。此外，把你的代码分隔成这些块，使得你的脚本可以使用来自该管道的元素。

比如说例 10-9 中的两个函数，`Get-InputWithForeach` 和 `Get-InputWithKeyword`。前者使用 `foreach` 语句来访问输入管道中的每个元素，而后者使用 `begin`、`process` 和 `end` 关键字。

例 10-9：两个函数采用不同的方法处理管道输入

```
## Process each element in the pipeline, using a  
## foreach statement to visit each element in $input  
function Get-InputWithForeach($identifier)  
{  
    Write-Host "Beginning InputWithForeach (ID: $identifier)"  
  
    foreach($element in $input)  
    {  
        Write-Host "Processing element $element (ID: $identifier)"  
        $element  
    }  
}
```



```
        Write-Host "Ending InputWithForeach (ID: $identifier)"  
    }  
  
## Process each element in the pipeline, using the  
## cmdlet-style keywords to visit each element in $input  
function Get-InputWithKeyword($identifier)  
{  
    begin  
    {  
        Write-Host "Beginning InputWithKeyword (ID: $identifier)"  
    }  
  
    process  
    {  
        Write-Host "Processing element $_ (ID: $identifier)"  
        $_  
    }  
    end  
    {  
        Write-Host "Ending InputWithKeyword (ID: $identifier)"  
    }  
}
```

这两个函数运行的结果是一样的，但是当我们把它们与其他脚本或函数联合使用的时候，不同之处就变得清楚了。当使用 \$input 变量的时候，必须要等前一个脚本完成输出后才能开始执行，如果前一个脚本要花费较长的时间才能输出全部记录（比如，大目录的列表），那么用户必须要等所有的目录列举结束后才能看到结果，而不是在脚本每生成一项时就能看到结果。

注意：如果一个脚本、函数或脚本块使用了 cmdlet 风格的关键字，它必须要把其全部的代码放到这三块中的一块里。如果你的代码需要定义变量或定义函数，就把这部分代码放到 begin 区域中。包含在 begin、process 和 end 块中的代码可以访问定义在它们前面的变量和函数。

当我们将通过使用 begin、process 和 end 关键字来处理输入的两个脚本串在一起的时候，第二个脚本会在第一个脚本输出一项结果的同时开始处理。

```
PS >1,2,3 | Get-InputWithKeyword 1 | Get-InputWithKeyword 2  
Beginning InputWithKeyword (ID: 1)  
Beginning InputWithKeyword (ID: 2)  
Processing element 1 (ID: 1)  
Processing element 1 (ID: 2)  
1  
Processing element 2 (ID: 1)  
Processing element 2 (ID: 2)  
2  
Processing element 3 (ID: 1)  
Processing element 3 (ID: 2)
```

```
3
Ending InputWithKeyword (ID: 1)
Ending InputWithKeyword (ID: 2)
```

当我们将通过 \$input 变量处理输入的两个脚本串在一起的时候，第二个脚本要等第一个脚本运行完毕后才开始运行。

```
PS >1,2,3 | Get-InputWithForeach 1 | Get-InputWithForeach 2
Beginning InputWithForeach (ID: 1)
Processing element 1 (ID: 1)
Processing element 2 (ID: 1)
Processing element 3 (ID: 1)
Ending InputWithForeach (ID: 1)
Beginning InputWithForeach (ID: 2)
Processing element 1 (ID: 2)
1
Processing element 2 (ID: 2)
2
Processing element 3 (ID: 2)
3
Ending InputWithForeach (ID: 2)
```

当第一个脚本使用 cmdlet 风格关键字，第二个脚本使用 \$input 变量的时候，第二个脚本直到第一个脚本结束后才开始运行。

```
PS >1,2,3 | Get-InputWithKeyword 1 | Get-InputWithForeach 2
Beginning InputWithKeyword (ID: 1)
Processing element 1 (ID: 1)
Processing element 2 (ID: 1)
Processing element 3 (ID: 1)
Ending InputWithKeyword (ID: 1)
Beginning InputWithForeach (ID: 2)
Processing element 1 (ID: 2)
1
Processing element 2 (ID: 2)
2
Processing element 3 (ID: 2)
3
Ending InputWithForeach (ID: 2)
```

当第一个脚本使用 \$input 变量，第二个脚本使用 cmdlet 风格关键字的时候，第二个脚本会在第一个脚本产生输出后马上开始工作。

```
PS >1,2,3 | Get-InputWithForeach 1 | Get-InputWithKeyword 2
Beginning InputWithKeyword (ID: 2)
Beginning InputWithForeach (ID: 1)
Processing element 1 (ID: 1)
Processing element 1 (ID: 2)
1
Processing element 2 (ID: 1)
Processing element 2 (ID: 2)
2
```

```
Processing element 3 (ID: 1)
Processing element 3 (ID: 2)
3
Ending InputWithForeach (ID: 1)
Ending InputWithKeyword (ID: 2)
```

关于处理管道输入的更多信息，请参见附录 A 中的“编写脚本，复用函数”。

参考

- #### • 附录 A 中的“编写脚本，复用函数”

10.9 编写一个面向管道的函数

问题

你的函数主要从管道获取输入，你希望对输入中的每项元素执行相同的操作。

哪个方案好

要编写一个面向管道的函数，需要使用 `filter` 关键字来定义函数，而不是用 `function` 关键字。PowerShell 会将当前的管道对象赋值给变量 `$_ variable`。

```
filter Get-PropertyValue($property)
{
    $_.$property
}
```

讨论

过滤器和函数一样使用 cmdlet 风格的关键字，所有的代码都写在 process 这部分里。

本解决方案演示了一个非常有用的过滤器：返回管道中每项的属性的值，如下：

```
PS >Get-Process | Get-PropertyValue Name  
audiogd  
avgamsrv  
avgemc  
avgrssvc  
avgrssvc  
avgupsvc  
{...}
```

关于 cmdlet 风格关键字的更多信息，请参见 10.8 节。

参考

• 10.8 节

秦始皇帝陵，秦代

简介 0.1

位于西安市临潼区，又名“兵马俑”，被誉为“世界第八大奇迹”。秦始皇陵是秦始皇嬴政的陵墓，位于今陕西省西安市临潼区东5公里。陵园占地近800万平方米，是世界上最大的帝王陵园。陵园内有兵马俑坑、陪葬坑、陵寝区等。陵寝区有铜车马、玉衣、金缕玉衣等珍贵文物。陪葬坑内有陶俑、陶马、陶车等。陵园外有秦始皇陵博物馆。

秦代皇帝陵墓复原 0.1

概况

秦代皇帝陵墓复原工程于1980年完成。

秦代皇帝陵墓复原工程

秦代皇帝陵墓复原工程于1980年完成，展示了秦始皇陵的雄伟壮观。

秦代皇帝陵墓复原工程于1980年完成，展示了秦始皇陵的雄伟壮观。

第 11 章

列表、数组和哈希表

11.0 简介

多数的脚本可以同时处理几件事情，列出服务器、文件，查找代码等。通过它的语言功能和实用程序，PowerShell 支持很多功能，它可以帮助你完成更多的任务。

在PowerShell中，使用数组和列表就像使用其他数据类型一样。你可以轻松的创建一个数组或列表，然后向其中添加元素或从中移出元素，也可以对它进行排序、搜索或与其他数组合并。当你希望存储一段数据与另一段数据之间的一个映射时，哈希表是很好的选择。

11.1 创建数组或项的列表

问题

你想创建一个数组或项的列表。

解决方案

要创建一个数组来保存给定的项，使用逗号将这些项分开，如下：

```
PS >$myArray = 1,2,"Hello World"
PS >$myArray
1
2
Hello World
```

要创建指定大小的数组，使用 New-Object 命令，如下：

```
PS >$myArray = New-Object string[] 10
PS >$myArray[5] = "Hello"
PS >$myArray[5]
Hello
```

要把一个命令的输出生成列表，使用变量赋值，如下：

```
PS >$myArray = Get-Process
PS >$myArray
```

Handles	NPM (K)	PM (K)	WS (K)	VM (M)	CPU(s)	Id	ProcessName
274	6	1316	3908	33		3164	alg
983	7	3636	7472	30		688	csrss
69	4	924	3332	30	0.69	2232	ctfmon
180	5	2220	6116	37		2816	dllhost
(...)							

要创建一个经常修改的数组，使用 ArrayList，如例 11-1 所示。

例 11-1：使用 ArrayList 管理动态的项的集合

```
PS >$myArray = New-Object System.Collections.ArrayList
PS >[void] $myArray.Add("Hello")
PS >[void] $myArray.AddRange( ("World", "How", "Are", "You") )
PS >$myArray
Hello
World
How
Are
You
PS >$myArray.RemoveAt(1)
PS >$myArray
Hello
How
Are
You
```

讨论

除了基本的数据类型（如字符串、整数和小数）之外，项的列表是你的脚本和你书写的命令中常见的内容。多数的命令会生成数据的列表：Get-Content 读取文件内容，生成字符串列表；Get-Process 生成当前系统正在运行的进程的列表；Get-Command 生成命令的列表等等。

注意：本解决方案中展示了如何保存一个命令的输出以形成一个列表。如果一个命令的输出只有一项（如文件只有一行，一个进程或一个命令），那么输出不再是列表。如果你想强制输出为列表，则使用列表赋值语法 (@())），如下：

```
$myArray = @(Get-Process Explorer)
```

关于列表和数组的更多信息，请参见附录 A 中的“数组和列表”。

参考

- 附录 A 中的“数组和列表”

11.2 创建交错或多维数组

问题

你要创建数组的数组或多维数组。

解决方案

若要创建一个交错的多维数组（一个数组的数组），可以使用 @() 数组语法，如下：

```
PS >$jagged = @(  
>>      (1,2,3,4),  
>>      (5,6,7,8)  
>>    )  
>>  
PS >$jagged[0][1]  
>>2  
PS >$jagged[1][3]  
>>8
```

若要创建一个（非交错）多维数组，使用 New-Object 命令，如下：

```
PS >$multidimensional = New-Object "int32[,]" 2,4  
PS >$multidimensional[0,1] = 2  
PS >$multidimensional[1,3] = 8  
PS >  
PS >$multidimensional[0,1]  
>>2  
PS >$multidimensional[1,3]  
>>8
```

讨论

交错的多维数组可用于保存列表或数组的数组的列表。交错的数组更易于使用（使用较少的内存），非交错的多维数组有时也可用于处理大型网格的数据。

由于一个交错的数组是由数组组成的数组，在交错数组中创建项与在一个正常的数组中创建项遵循相同的规则。如果数组的任何项是单个元素的数组，则使用一元逗号运算符。例如，创建一个交错的数组嵌套一个只有一个元素的数组，如下：

```
PS >$oneByOneJagged = @()
>> , (,1)
>>
PS >$oneByOneJagged[0][0]
```

关于列表和数组的更多信息，请参见附录 A 中的“数组和列表”。

参考

- 附录 A 中的“数组和列表”

11.3 访问数组中的元素

问题

你要访问数组中的元素。

解决方案

若要访问数组中的特定元素，使用 PowerShell 的数组访问机制，如下：

```
PS >$myArray = 1,2,"Hello World"
PS >$myArray[1]
2
```

若要访问数组中某个范围的元素，使用数组区域和数组截取，如下：

```
PS >$myArray = 1,2,"Hello World"
PS >$myArray[1..2 + 0]
2
Hello World
1
```

讨论

PowerShell 的数组访问机制提供了一种方便的方式来访问数组的特定元素或数组中更复杂的组合元素。在 PowerShell 中（和大多数其他脚本和编程语言一样），索引 0 处的项表示数组中的第一个项。

尽管可以通过数组的数字索引来访问数组中的元素，但也可以通过名称，甚至是自定义的标签等其他方法来访问。这种类型的数组也称为关联数组（associative array）（或哈希表（hashtable））。关于使用哈希表和关联数组的更多信息，请参见 11.12 节。

关于 PowerShell 中列表和数组的更多信息（包括数组范围和截取），请参见附录 A 中的“数组和列表”。

参考

- 11.12 节
- 附录 A 中的“数组和列表”

11.4 访问数组的每个元素

问题

你希望使用数组的每个元素。

解决方案

要逐个访问数组中的项，使用 `Foreach-Object` 命令，如下：

```
PS >$myArray = 1,2,3
PS >$sum = 0
PS >$myArray | Foreach-Object { $sum += $_ }
PS >$sum
6
```

要以更脚本化的方式访问数组中的项，使用 `foreach` 脚本关键字，如下：

```
PS >$myArray = 1,2,3
PS >$sum = 0
PS >foreach($element in $myArray) { $sum += $element }
PS >$sum
6
```

若要按位置访问数组中的各项，使用一个 `for` 循环，如下：

```
PS >$myArray = 1,2,3
PS >$sum = 0
PS >for($counter = 0; $counter -lt $myArray.Count; $counter++) {
>>     $sum += $myArray[$counter]
>> }
>>
PS >$sum
6
```

讨论

PowerShell提供了三个主要的备选方案来处理一个数组中的元素。Foreach-Object命令和foreach脚本关键字一次访问数组中的一项，而for循环（和相关的循环结构）可以让你在一种不太结构化的方法中使用数组中的项。

关于 Foreach-Object 的更多信息，请参见 2.4 节。

关于 foreach、for 和循环构造等脚本关键字的更多信息，请参见 4.4 节。

参考

- 2.4 节
- 4.4 节

11.5 对数组或列表中的项进行排序

问题

你要对数组或列表中的元素排序。

解决方案

要对列表项进行排序，使用 Sort-Object 命令，如下：

```
PS >Get-ChildItem | Sort-Object -Descending Length | Select Name,Length
```

Name	Length
Convert-TextObject.ps1	6868
Connect-WebService.ps1	4178
Select-FilteredObject.ps1	3252
Get-PageUrls.ps1	2878
Get-Characteristics.ps1	2515
Get-Answer.ps1	1890
New-GenericObject.ps1	1490
Invoke-CmdScript.ps1	1313

讨论

Sort-Object为你提供了一种按指定的属性进行排序的方便的方法。如果你不指定一个属性，那么 Sort-Object 遵循默认的排序规则。

除了可以按某个属性进行升序或降序的排序，Sort-Object 可以使用 -Unique 选项从已排序的集合中移除重复项。

关于 Sort-Object 的更多的信息，可以键入 **Get-Help Sort-Object**。

11.6 确定数组是否包含某项

问题

你需要确定数组或列表是否包含特定项。

解决方案

要检查一个列表是否包含指定的项，使用 -contains 运算符，如下：

```
PS >"Hello", "World" -contains "Hello"
True
PS >"Hello", "World" -contains "There"
False
```

讨论

-contains 运算符是一个有用的方法来快速确定列表是否包含特定元素。若要搜索与某个模式匹配的项目列表，使用 -match 或 -like 运算符。

关于 -contains, -match 和 -like 操作符的更多信息，参见附录 A 中的“比较操作符”。

参考

- 附录 A 中的“比较操作符”

11.7 合并数组

问题

你有两个数组并想将它们合成一个。

解决方案

要合并数组，需要使用加法运算符 (+)，如下：

```
PS >$firstArray = "Element 1", "Element 2", "Element 3", "Element 4"
PS >$secondArray = 1, 2, 3, 4
PS >
PS >$result = $firstArray + $secondArray
PS >$result
Element 1
Element 2
Element 3
Element 4
1
2
3
4
```

讨论

合并两个数组的一个常见原因是你希望将数据添加到一个数组的末尾。例如：

```
PS >$array = 1,2
PS >$array = $array + 3,4
PS >$array
1
2
3
4
```

你可以写得更加清晰，如下：

```
PS >$array = 1,2
PS >$array += 3,4
PS >$array
1
2
3
4
```

当使用第二种方式合并数组的时候，你可能会认为 PowerShell 添加项到数组末尾的同时保持数组本身不变。这其实是错误的，因为在 PowerShell 中，一旦创建了数组，就要保持数组的长度不变。在合并两个数组的时候，PowerShell 创建一个足够大的数组，以便容纳两个数组的内容，然后将这两个数组的内容复制到目标数组。

如果您打算经常向数组添加和从数组删除数据，那么 `System.Collections.ArrayList` 类提供了一个更加动态的替代方法。有关使用 `ArrayList` 类的更多信息，请参阅 11.11 节。

参考

- 11.11 节

11.8 从数组中查找匹配一个值的项

问题

你有一个数组，希望查找所有的与给定的项或条件（包括完全相同、某一模式或正则表达式）相匹配的项。

解决方案

若要查找与项目相匹配的所有元素，使用 `-eq`、`-like` 和 `-match` 比较运算符，如下：

```
PS >$array = "Item 1", "Item 2", "Item 3", "Item 1", "Item 12"
PS >$array -eq "Item 1"
Item 1
Item 1
PS >$array -like "*1*"
Item 1
Item 1
Item 12
PS >$array -match "Item .."
Item 12
```

讨论

操作符 `-eq`、`-like` 和 `-match` 是非常有用的，它们可以让你从一个集合中查找匹配的元素。操作符 `-eq` 返回所有与你输入的条件相等的元素；操作符 `-like` 返回所有与你输入的模式相似的元素；操作符 `-match` 返回所有与你输入的正则表达式相匹配的元素。

关于操作符 `-eq`、`-like` 和 `-match` 的更多信息，请参见附录 A 中的“比较操作符”。

参考

- 附录 A 中的“比较操作符”

11.9 从数组中移出元素

问题

你想根据给定的条件从数组中移出元素，这些条件可以是完全相同，某一模式或正则表达式。

解决方案

要从一个数组中移除所有匹配某一模式的元素，可以使用`-ne`、`-notlike`和`-notmatch`比较运算符，如例 11-2 中所示。

例 11-2：使用运算符`-ne`、`-notlike`和`-notmatch`从数组中移出元素

```
PS >$array = "Item 1", "Item 2", "Item 3", "Item 1", "Item 12"
PS >$array -ne "Item 1"
Item 2
Item 3
Item 12
PS >$array -notlike "*1*"
Item 2
Item 3
PS >$array -notmatch "Item .."
Item 1
Item 2
Item 3
Item 1
```

要从该数组中实际移除项，需要将结果存储数组中：

```
PS >$array = "Item 1", "Item 2", "Item 3", "Item 1", "Item 12"
PS >$array = $array -ne "Item 1"
PS >$array
Item 2
Item 3
Item 12
```

讨论

在一个集合中查找符合给定条件的元素，可以使用`-eq`、`-like`和`-match`运算符；与之相对的运算符`-ne`、`-notlike`和`-notmatch`会返回不匹配的元素。

第二个例子演示了如何从一个数组中移除所有与给定的模式匹配的元素，然后保存所有与该模式不匹配的元素。

关于运算符`-ne`、`-notlike`和`-notmatch`的更多信息，可以参见附录 A 中的“比较运算符”。

11.10 从数组中查找大于或小于一个值的项

问题

你有一个数组，要查找所有大于或小于一个给定值的项。

解决方案

要查找所有大于或小于一个给定值的项，可以使用 -gt、-lt 和 -le 比较运算符，如下：

```
PS >$array = "Item 1", "Item 2", "Item 3", "Item 1", "Item 12"  
PS >$array -ge "Item 3"  
Item 3  
PS >$array -lt "Item 3"  
Item 1  
Item 2  
Item 1  
Item 12
```

讨论

从集合中查找大于或小于一个给定值的项，可以使用 -ge、-lt 和 -le 运算符。像所有其他的 PowerShell 的比较运算符一样，这些运算符都使用集合中的项的比较规则。由于解决方案中的数组是一个字符串数组，所以这个结果可能让你感到意外，如下：

```
PS >$array -lt "Item 2"  
Item 1  
Item 1  
Item 12
```

返回这样的结果是因为数组是按字母顺序(alphabetically)，"Item 12"是在"Item 2"之前，这是 PowerShell 进行字符串数组比较的方法。

```
PS >$array | Sort-Object  
Item 1  
Item 1  
Item 12  
Item 2  
Item 3
```

关于 -gt、-ge、-lt 和 -le 运算符的更多信息，请参见附录 A 中的“比较运算符”。

参考

- 附录 A 中的“比较运算符”

11.11 使用 ArrayList 类完成高级的数组任务

问题

你想要频繁地对数组执行添加、搜索、移除和修改的操作。

解决方案

如果你定义数组之后要经常使用它，可以使用System.Collections.ArrayList类，如下：

```
PS >$myArray = New-Object System.Collections.ArrayList
PS >[void] $myArray.Add("Hello")
PS >[void] $myArray.AddRange( ("World", "How", "Are", "You") )
PS >$myArray
Hello
World
How
Are
You
PS >$myArray.RemoveAt(1)
PS >$myArray
Hello
How
Are
You
```

讨论

像大多数其他语言一样，PowerShell 中的数组在被创建后保持不变的长度。PowerShell 允许你在数组中添加项、删除项和搜索项，但当你处理大批量的数据时这些操作可能会耗费时间。例如，为了合并两个数组，PowerShell 要创建一个足够大的新数组来容纳两个数组的内容，然后将这两个数组复制到目标数组。

在比较的时候，`ArrayList` 类旨在使你轻松地在集合中添加、删除和搜索项。

注意：默认地，PowerShell会传递你的脚本生成的任何数据，除非你捕获它或将其强制转换为 [void]。因为主要的设计目标是作为一种编程语言，所以 System.Collections.ArrayList 类自动会生成输出，即使你不希望那样。要防止将数据发送到输出管道，要么捕获数据，要么将其强制转换为 [void]：

```
PS >$collection = New-Object System.Collections.ArrayList  
PS >$collection.Add("Hello")  
0  
PS >[void] $collection.Add("World")
```

如果你经常对某个数组进行添加和删除操作，`System.Collections.ArrayList` 类会提供一个更加动态的替代方法。

关于使用 .NET 框架中的类的更多信息，请参见 3.4 节。

参考

- 3.4 节 “使用 PowerShell cmdlet 里常用的哈希表” 中讲解了如何使用哈希表。

11.12 创建哈希表或关联数组

问题

你想要通过你提供的一个标签访问集合中的项。

解决方案

要定义标签与项之间的一个映射，可以使用一个哈希表（关联数组）：

```
PS >$myHashtable = @{}  
PS >  
PS >$myHashtable = @{ "Key1" = "Value1"; "Key 2" = 1,2,3 }  
PS >$myHashtable["New Item"] = 5  
PS >  
PS >$myHashTable  


| Name     | Value     |
|----------|-----------|
| Key 2    | {1, 2, 3} |
| New Item | 5         |
| Key1     | Value1    |


```

讨论

哈希表与数组相似，两者之间不同的是哈希表允许你通过定义的标签来访问数组中的项，而数组通过索引来访问。由于使用方便灵活，哈希表是构成大量脚本的技术基础。因为它们允许你将名称映射到值，所以它们是构成如 ZIP 代码和区号查找表的自然基础。因为它们允许你将名称映射到对象和脚本块，所以它们通常可以取代自定义对象。因为你可以在丰富的对象映射到其他丰富的对象，它们甚至可以作为更高级的数据结构，如缓存和图表对象的基础。

在与支持高级的配置参数的 cmdlet 交互的时候，这种标签和值之间的映射是有帮助的，如 Format-Table 和 Select-Object 命令支持的计算属性参数。

关于哈希表的更多信息，请参见附录 A 中的“哈希表（关联数组）”。

参考

- 附录 A 中的“哈希表（关联数组）”

11.13 根据键或值对哈希表排序

问题

你有一个哈希表，并且你想根据键值对值的列表进行排序。

解决方案

若要排序一个哈希表，可以在哈希表上使用 `GetEnumerator()` 方法来访问它的各个元素。然后，使用 `Sort-Object` 命令来按名称或值对其进行排序。

```
foreach($item in $myHashtable.GetEnumerator() | Sort Name)
{
    $item.Value
}
```

讨论

由于一个哈希表的重点是简单地把键映射到值，所以你不应依赖于它来保留任何排序顺序，比如你添加项的顺序、键的排序顺序或值的排序顺序等等。

例 11-3 清楚的说明了这一点。

例 11-3：哈希表不保留顺序的例子

```
PS >$myHashtable = @{}
PS >$myHashtable["Hello"] = 3
PS >$myHashtable["Ali"] = 2
PS >$myHashtable["Alien"] = 4
PS >$myHashtable["Duck"] = 1
PS >$myHashtable["Hectic"] = 11
PS >$myHashtable
```

Name	Value
Hectic	11
Duck	1
Alien	4
Hello	3
Ali	2

但是，哈希表对象支持 `GetEnumerator()` 方法，允许你使用独立的哈希表项，每项都有一个 `name` 和 `value` 属性。一旦我们获得了这些，我们就可以对它们排序，就像我们对任何其他的 PowerShell 数据进行排序那样轻松。例 11-4 演示了这项技术。

例 11-4：按键名或值对哈希表排序

```
PS >$myHashtable.GetEnumerator() | Sort Name
```

Name	Value
Ali	2
Alien	4
Duck	1
Hectic	11
Hello	3

Name	Value
Duck	1
Ali	2
Hello	3
Alien	4
Hectic	11

关于哈希表的更多信息，请参见附录 A 中的“哈希表（关联数组）”

参考

- 附录 A 中的“哈希表（关联数组）”

用户交互

12.0 简介

大多数脚本设计为自动运行，因此你经常会发现让脚本与用户交互是十分有用的。

从用户处获得输入的最好的方法是通过参数给脚本或函数传递参数。这样用户在运行脚本的时候可以不必在现场！

如果脚本大大受益于（或需要）互动的体验，那么 PowerShell 提供了多种可能性。这可能只是等待一个按键，提示输入，或显示一个更丰富的基于选择的提示。

用户的输入并不是交互的唯一内容，除了输入设备，PowerShell 还支持从显示简单的文本字符串到更详细进度报告的输出以及与用户界面框架之间的交互。

12.1 读取用户输入一行

问题

如何在脚本中使用用户输入的内容。

解决方案

要获得用户的输入内容，需要使用 **Read-Host** 命令，如下：

```
PS >$directory = Read-Host "Enter a directory name"  
Enter a directory name: C:\MyDirectory  
PS >$directory  
C:\MyDirectory
```

讨论

`Read-Host` 命令从用户输入的内容中读取一行。如果输入内容包含敏感的信息，`Read-Host` 支持使用参数 `-AsSecureString` 来读取输入，作为一个安全字符串来处理。

如果用户的输入表示日期、时间或编号，那么要注意大多数地区以不同的方式表示这些数据类型。有关编写地区性的脚本的详细信息，请参阅 12.6 节。

想得到 `Read-Host` 命令的更多信息，请输入 `Get-Help Read-Host`。

参考

- 12.6 节

12.2 读取用户输入的按键

问题

如何在脚本中获得用户输入的按键。

解决方案

在大多数情况下，使用 `[Console]::ReadKey()` 方法来获得用户按键，如下：

```
PS >$key = [Console]::ReadKey($true)  
PS >$key
```

KeyChar	Key	Modifiers
h	H	Alt

在交互比较频繁的情况下（比如，你关心用户按下的是向上键还是向下键），使用下面的语句：

```
PS >$key = $host.UI.RawUI.ReadKey("NoEcho,IncludeKeyDown")  
PS >$key
```

VirtualKeyCode	Character	ControlKeyState	KeyDown
16	...ssed, NumLockOn		True

```
PS >$key.ControlKeyState  
ShiftPressed, NumLockOn
```

讨论

大多数情况下，使用 [Console]::ReadKey() 是获得用户按键的最好的办法。因为它既可以接受简单的按键，也可以接受复杂的按键，其中可能包括 Ctrl、Alt 和 Shift 键。

下面的函数模拟了 DOS 下的 pause 命令：

```
function Pause
{
    Write-Host -NoNewLine "Press any key to continue . . . "
    [Console]::ReadKey($true) | Out-Null
    Write-Host
}
```

如果你要捕获单个键按下和松开事件（包括 Ctrl、Alt 和 Shift），使用 \$host.UI.RawUI.ReadKey() 方法。

12.3 编程：向用户显示一个菜单

问题

我们经常会让用户从我们给定的选项范围内选择某一项。下面的脚本让你以一种比默认方式更友好的方式访问 PowerShell 的提示功能。它从你提供的选项的列表中返回一个表示它们选择位置的数字。

PowerShell 的提示需要你包含一个加速键（描述选项中字母前面键入 &）来定义表示该按键的选项。因为你不总是控制选项的列表（例如，可能的目录的一个列表），例 12-1 自动为缺少的描述生成合适的加速键字符。

例 12-1：Read-HostWithPrompt.ps1

```
#####
## Read-HostWithPrompt.ps1
##
## Read user input, with choices restricted to the list of options you
## provide.
##
## ie:
##
## PS >$caption = "Please specify a task"
## PS >$message = "Specify a task to run"
## PS >$option = "&Clean Temporary Files","&Defragment Hard Drive"
## PS >$helptext = "Clean the temporary files from the computer",
## >>           "Run the defragment task"
## >>
## PS >$default = 1
```

```

## PS >Read-HostWithPrompt $caption $message $option $helpText $default
##
## Please specify a task
## Specify a task to run
## [C] Clean Temporary Files [D] Defragment Hard Drive [?] Help
## (default is "D")?:?
## C - Clean the temporary files from the computer
## D - Run the defragment task
## [C] Clean Temporary Files [D] Defragment Hard Drive [?] Help
## (default is "D")?:C
## 0
#####

param(
    $caption = $null,
    $message = $null,
    $option = $(throw "Please specify some options."),
    $helpText = $null,
    $default = 0
)

## Create the list of choices
[Management.Automation.Host.ChoiceDescription[]] $choices = @()

## Create a list of possible key accelerators for their options
$accelerators = New-Object System.Collections.ArrayList

## First, add a the list of numbers as possible choices
$startNumber = [int][char] '0'
$endNumber = [int][char] '9'
foreach($number in $startNumber..$endNumber)
{
    [void] $accelerators.Add([char] $number)
}

## Then, a list of characters as possible choices
$startLetter = [int][char] 'A'
$endLetter = [int][char] 'Z'
foreach($letter in $startLetter..$endLetter)
{
    [void] $accelerators.Add([char] $letter)
}

## Go through each of the options, and add them to the choice collection
for($counter = 0; $counter -lt $option.Length; $counter++)
{
    $optionText = $option[$counter]

    ## If they didn't provide an accelerator, generate new option
    ## text for them
    if($optionText -notmatch '&')
    {
        $optionText = "&{0} - {1}" -f $accelerators[0], $optionText
    }
}

```

```
## Now, remove their option character from the list of possibilities
$acceleratorIndex = $optionText.IndexOf('&')
$optionCharacter = $optionText[$acceleratorIndex + 1]
$accelerators.Remove($optionCharacter)

## Create the choice
$choice = New-Object Management.Automation.Host.ChoiceDescription $optionText
if($helpText -and $helpText[$counter])
{
    $choice.HelpMessage = $helpText[$counter]
}

## Add the choice to the list of possible choices
$choices += $choice
}

## Prompt for the choice, returning the item the user selected
$host.UI.PromptForChoice($caption, $message, $choices, $default)
```

关于运行脚本的更多信息，请参见 1.1 节。

参考

- 1.1 节

12.4 给用户显示输出和消息

问题

如何向用户显示消息和其他信息。

解决方案

要确保输出到达屏幕，需要使用 Write-Host 或 Out-Host 命令，如下：

```
PS >function Get-DirectorySize
>> {
>>     $size = (Get-ChildItem | Measure-Object -Sum Length).Sum
>>     Write-Host ("Directory size: {0:N0} bytes" -f $size)
>> }
PS >Get-DirectorySize
Directory size: 46,581 bytes
PS >$size = Get-DirectorySize
Directory size: 46,581 bytes
```

如果你需要一些信息来帮助你（或用户）调试你的脚本，可以使用 Write-Debug 命令；

如果要提供详细的跟踪类型的输出，可以使用 Write-Verbose 命令。例 12-2 演示了这些 cmdlet。

例 12-2：提供调试和验证输出的函数

```
PS >function Get-DirectorySize
>> {
>>     Write-Debug "Current Directory: $(Get-Location)"
>>
>>     Write-Verbose "Getting size"
>>     $size = (Get-ChildItem | Measure-Object -Sum Length).Sum
>>     Write-Verbose "Got size: $size"
>>
>>     Write-Host ("Directory size: {0:N0} bytes" -f $size)
>> }
PS >$DebugPreference = "Continue"
PS >Get-DirectorySize
DEBUG: Current Directory: D:\lee\OReilly\Scripts\Programs
Directory size: 46,581 bytes
PS >$DebugPreference = "SilentlyContinue"
PS >$VerbosePreference = "Continue"
PS >Get-DirectorySize
VERBOSE: Getting size
VERBOSE: Got size: 46581
Directory size: 46,581 bytes
PS >$VerbosePreference = "SilentlyContinue"
```

讨论

你编写的多数脚本会输出丰富的结构化数据，比如一个目录的实际大小。这样，其他脚本可以使用该脚本的输出作为它们功能的生成块。

当确实需要专门为用户提供输出时，可以使用 Write-Host、Write-Debug 和 Write-Verbose 命令。

但是，应注意这种输出的类型会绕过正常的文件重定向，因此很难让用户捕获。只有当你的脚本已经生成其他，可以被用户在文件或变量中捕获结构化的数据时，才使用 Write-Host。

当多数作者的脚本尝试着将带格式的数据输出给用户的时候，他们都会遇到例 12-3 中的问题。

例 12-3：格式语句引起的错误消息

```
PS >## Get the list of items in a directory, sorted by length
PS >function Get-ChildItemSortedByLength($path = (Get-Location))
>> {
>>     Get-ChildItem $path | Format-Table | Sort Length
```

```
>> }
>>
PS >Get-ChildItemSortedByLength
out-lineoutput : Object of type "Microsoft.PowerShell.Commands.Internal.Format.FormatEntryData" is not legal or not in the correct sequence. This is likely caused by a user-specified "format-*" command which is conflicting with the default formatting.
```

发生这种情况是因为实际上 `format-*` 命令生成的格式设置信息是供给 `Out-Host` 命令来使用的。`Out-Host` 命令（PowerShell 将自动添加到你的管道的末尾）接下来使用此信息来生成带格式的输出。若要解决此问题，始终要确保格式命令是管道中的最后一个命令。如例 12-4 中所示：

例 12-4：不产生格式错误的函数

```
PS >## Get the list of items in a directory, sorted by length
PS >function Get-ChildItemSortedByLength($path = (Get-Location))
>> {
>>     ## Problematic version
>>     ## Get-ChildItem $path | Format-Table | Sort Length
>>
>>     ## Fixed version
>>     Get-ChildItem $path | Sort Length | Format-Table
>> }
>>
PS >Get-ChildItemSortedByLength
(...)

Mode           LastWriteTime      Length Name
----           -----          ---- -
-a--- 3/11/2007 3:21 PM        59 LibraryProperties.ps1
-a--- 3/6/2007 10:27 AM       150 Get-Tomorrow.ps1
-a--- 3/4/2007 3:10 PM       194 ConvertFrom-FahrenheitWithoutFunction.ps1
-a--- 3/4/2007 4:40 PM       257 LibraryTemperature.ps1
-a--- 3/4/2007 4:57 PM       281 ConvertFrom-FahrenheitWithLibrary.ps1
-a--- 3/4/2007 3:14 PM       337 ConvertFrom-FahrenheitWithFunction.ps1
(...)
```

当运行脚本产生输出给用户的时候，一个常见的问题是提供进度信息。PowerShell 实际通过一种更丰富的方式，即通过 `Write-Progress` 命令来支持进度信息。有关 `Write-Progress` 的更多信息，请参见 12.5 节。

参考

- 12.5 节

12.5 为长时间运行的任务提供进度更新

问题

如何在运行长时间的任务的时候，向用户显示状态信息。

解决方案

要提供状态的更新信息，需要使用 Write-Progress，如例 12-5 所示：

例 12-5：使用 Write-Progress 命令来显示状态更新

```
$activity = "A long-running operation"

$status = "Initializing"
## Initialize the long-running operation
for($counter = 0; $counter -lt 100; $counter++)
{
    $currentOperation = "Initializing item $counter"
    Write-Progress $activity $status -PercentComplete $counter
        -CurrentOperation $currentOperation
    Start-Sleep -m 20
}

$status = "Running"
## Initialize the long-running operation
for($counter = 0; $counter -lt 100; $counter++)
{
    $currentOperation = "Running task $counter"
    Write-Progress $activity $status -PercentComplete $counter
        -CurrentOperation $currentOperation
    Start-Sleep -m 20
}
```

讨论

当你的脚本运行长时间的操作（如图 12-1）的时候，Write-Progress 为用户提供结构化的状态信息。

像其他详细信息命令（Write-Debug、Write-Verbose 和其他 Write-* 命令）一样，PowerShell 允许用户控制它们看到此信息的大小。

有关 Write-Progress 命令的更多信息，请键入 **Get-help Write-Progress**。

The screenshot shows a Windows PowerShell window titled 'C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe'. The window displays a list of processes and their details. At the top, it shows the following header information:

	91	3	6576	2096	49		1144	SLsvc
	28	1	244	332	4		356	smss

Below this, the text 'A long running operation' and 'Initializing' is displayed, followed by a series of zeros: [oooooooooooooooooooooooooooo]. The main body of the window lists numerous processes with their respective details:

683	15	40960	36188	151		1008	svchost
1468	43	57984	20528	208		1020	svchost
574	21	11768	7272	91		1176	svchost
638	21	19344	7076	106		1368	svchost
273	23	13404	8920	75		1564	svchost
105	4	1888	1912	35		1928	svchost
272	8	3592	2888	48		3432	svchost
415	0	0	1304	9		4	System
129	4	2024	2328	55		1308	taskeng
333	10	9760	3960	80	5.34	2832	taskeng
67	3	2100	1468	38	6.30	3716	unsecapp
97	4	1176	388	34		476	wininit
128	3	1816	1784	39		520	winlogon
916	32	60460	27356	743	347.05	3948	WINWORD
212	5	2264	1824	63	0.55	3312	wmdSync
110	4	3356	2484	40		3776	WmiPrvSE
347	10	6896	4940	109		5888	wmpnetwk
103	3	1556	4140	44	0.19	2656	wmpnscfg

At the bottom of the window, the command PS > .\Invoke-LongRunningOperation.ps1 is visible.

图 12-1：长时间操作的输出

12.6 编写支持区域性的脚本

问题

如何确保你的脚本适用于世界各地的计算机。

解决方案

若要编写支持区域性的脚本，在你开发脚本的时候，要记住以下原则：

- 使用 PowerShell 的原始语言命令来创建日期、时间和数字。
- 使用 PowerShell 的内置运算符比较字符串。
- 避免将用户的输入视为字符的集合。
- 使用 Parse() 方法来将用户输入转换成日期、时间和数字。

讨论

在专业软件开发人员的领域里，编写支持区域性的程序在很长时间内是孤立的。简单的程序和脚本并不能从区域性意识中受益。对于非专业程序员来说遵循最佳做法经常太难。但是 PowerShell 使得这比传统的编程语言更容易了。

你的脚本在不同的文化环境之间传输的时候，一些事情发生了改变。

日期、时间和数字的格式

大多数区域有特定的日期、时间和数字格式。若要确保你的脚本在所有区域工作正常，无论你的脚本在哪里运行，PowerShell 都要首先确保其语言基元保持一致性。即使你的脚本运行在一台在法国的机器上（使用逗号为它的小数位分隔符），你还是可以依靠语句 \$myDouble = 3.5 来创建一个处于 3 和 4 中间的数字。同样的，即使在区域中日期的顺序是日、月、年，你仍然可以使用赋值语句 \$christmas = [DateTime] "12/25/2007" 来创建表示 2007 年圣诞节的一个日期。

区域性的程序会使用该区域的文化作为显示日期、时间和数字的首选项。当脚本在区域之间移动的时候它不会中断脚本，并且是编写支持区域性的脚本的一个重要方面。PowerShell 会为你处理这些问题，因为每当显示数据的时候，它都会使用当前区域的首选项。

注意：如果你的脚本要求用户输入日期、时间或数字，则必须确保你执行操作的时候考虑到了受影响的用户的区域性的格式。

若要将用户输入转换成一种特定类型的数据，需要使用该类型的 Parse() 方法。

```
$userInput = Read-Host "Please enter a date"  
$enteredDate = [DateTime]::Parse($userInput)
```

因此，若要确保你的脚本对日期、时间和数字格式保持区域性支持，只需在脚本中定义它们的时候使用 PowerShell 的语言基元即可。当你从用户处读取它们的时候，当你把它们从字符串转换成其他形式的时候，使用 Parse() 方法。

用户输入和文件内容的复杂程度

英语之所以是一种极好的语言，是在于它的字母非常简单。这将导致所有类型的编程算法，将用户输入和文件内容视为字节或简单的纯文本（ASCII）字符的数组；而在多数的国际语言中，这些技巧会失败。许多国际符号实际上占用两个字符的长度。

对于所有基于字符串的操作，PowerShell 都使用标准的 Unicode 格式：从用户那里获得输入，向用户显示输出、通过管道发送数据以及使用文件。

注意：尽管 PowerShell 完全支持 Unicode，但由于 Windows 控制台系统中的限制，powershell.exe 命令行主机不能正确输出某些字符。图形化的 PowerShell 主机（如多个第三方 PowerShell IDE）却不受这些限制的影响。

当处理用户输入的时候，如果你使用 PowerShell 的标准功能，你就不必担心它的复杂性；如果你要使用单个字符或输入中的单词，那么你就需要采取特殊的预防措施。System.Globalization.StringInfo 类允许你在一种区域性的方法中执行此操作。关于 System.Globalization.StringInfo 类的更多的信息，请参见[http://msdn2.microsoft.com/en-us/library/7h9tk6x8\(vs.71\).aspx](http://msdn2.microsoft.com/en-us/library/7h9tk6x8(vs.71).aspx)。

因此，要确保你的脚本对于用户的输入保持区域性，就要尽可能地使用 PowerShell 的字符串操作符。

大小写规则

在脚本中常见的要求是将用户的输入与一些预定义文本（如一个菜单选择）进行比较。你通常希望这个比较是不区分大小写的，这样一来，“QUIT”和“qUiT”就表示相同的含义了。

若要完成此任务，最常见的方法是将用户输入转换成大写或小写：

```
## $text comes from the user, and contains the value "quit"
if($text.ToUpper() -eq "QUIT") { ... }
```

遗憾的是，在不同的文化环境中显式更改大写字符串的操作会失败，因为很多区域具有不同的大小写和比较规则。例如，土耳其语言包括两种类型的字母“İ”：一个带点，另一个不带。小写字母“i”的大写版本对应于大写字母“İ”带点，而不是 QUIT 中的“I”。

该例使上面的字符串比较在土耳其语系统上就会失败。

要将某些输入与一个硬编码的字符串以不区分大小写的方式进行比较，更好的解决方案是使用 PowerShell 的 -eq 运算符，而不更改任何字符串。默认情况下，-eq 运算符是不区分大小写和区域性的，如下：

```
PS >$text1 = "Hello"
PS >$text2 = "HELLO"
PS >$text1 -eq $text2
True
```

因此，要确保你的脚本相对于大小写规则保持区域性，只要有可能，就尽量使用 PowerShell 中不区分大小写的比较运算符。

排序规则

排序规则在区域之间频繁地更改。例如，12.7 节中，给出的比较英语和丹麦语的脚本。

```
PS >Use-Culture en-US { "Apple", "Æble" | Sort-Object }
Æble
Apple
PS >Use-Culture da-DK { "Apple", "Æble" | Sort-Object }
Apple
Æble
```

要确保你的脚本相对于排序规则保持区域性，就要假定在你排序后排序结果正确输出，但不要依赖于已排序的输出的实际顺序。

其他指南

关于影响编写区域性程序的因素的其他资源，请参见 [http://msdn2.microsoft.com/en-us/library/h6270d0z\(vs.71\).aspx](http://msdn2.microsoft.com/en-us/library/h6270d0z(vs.71).aspx) 和 <http://www.microsoft.com/globaldev/getwr/steps/wrguide.mspx>。

参考

- 12.7 节

12.7 编程：采用交替的区域性设置调用脚本块

在PowerShell的用户社区中，你共享的脚本将通常运行在设置为英语之外的其他语言系统上。要确保你的脚本能够正常运行在其他语言中，在该语言环境中进行测试是很有帮助的。例 12-6 允许你在你选择的区域中运行你提供的脚本块。

例 12-6：Use-Culture.ps1

```
#####
## Use-Culture.ps1
##
## Invoke a scriptblock under the given culture
##
## ie:
##
## PS >Use-Culture fr-FR { [DateTime]::Parse("25/12/2007") }
```

```
##  
## mardi 25 décembre 2007 00:00:00##  
##  
##### param([  
[System.Globalization.CultureInfo] $culture =  
$(throw "Please specify a culture"),  
[ScriptBlock] $script = $(throw "Please specify a scriptblock")  
)  
  
## A helper function to set the current culture  
function Set-Culture([System.Globalization.CultureInfo] $culture)  
{  
    [System.Threading.Thread]::CurrentThread.CurrentCulture = $culture  
    [System.Threading.Thread]::CurrentThread.CurrentCulture = $culture  
}  
  
## Remember the original culture information  
$oldCulture = [System.Threading.Thread]::CurrentThread.CurrentCulture  
  
## Restore the original culture information if  
## the user's script encounters errors.  
trap { Set-Culture $oldCulture }  
  
## Set the current culture to the user's provided  
## culture.  
Set-Culture $culture  
  
## Invoke the user's scriptblock  
& $script  
  
## Restore the original culture information.  
Set-Culture $oldCulture
```

关于运行脚本的更多信息，请参见 1.1 节。

参考

- 1.1 节

12.8 主机的用户界面的访问功能

问题

你希望与宿主应用程序的用户界面中的功能进行交互，但 PowerShell 不直接提供这些命令。

解决方案

若要访问主机的用户界面功能，需要使用 \$host.UI.RawUI 变量，如下：

```
$host.UI.RawUI.WindowTitle = (Get-Location)
```

讨论

PowerShell 本身包含两个重要的组件：第一个是解释命令、执行管道，并执行其他类似操作的引擎；第二个是主应用程序，提供用户与 PowerShell 引擎交互的方式。

默认的 shell 程序，如 PowerShell.exe，基于的是传统的 Windows 控制台的用户界面。其他应用程序存在于该主 PowerShell 的一个图形用户界面中。事实上，PowerShell 使得开发人员生成其自己的主应用程序，或甚至将 PowerShell 引擎功能嵌入到它们自己的应用程序里变得相对简单。

你（和你的脚本）可以通过变量 \$host.UI 使函数功能在所有的主机上保持不变。例 12-7 显示了对你来说所有主机的可用功能。

例 12-7：通过 \$host.UI 属性可用的函数

```
PS >$host.UI | Get-Member | Select Name,MemberType | Format-Table -Auto
```

Name	MemberType
(...)	
Prompt	Method
PromptForChoice	Method
PromptForCredential	Method
ReadLine	Method
ReadLineAsSecureString	Method
Write	Method
WriteDebugLine	Method
WriteErrorLine	Method
WriteLine	Method
WriteProgress	Method
WriteVerboseLine	Method
WriteWarningLine	Method
RawUI	Property

如果你（或你的脚本）要与特定于当前宿主的用户界面部分进行交互，那么 PowerShell 可以提供通过 \$host.UI.RawUI 变量的访问功能。例 12-8 给你展示了 PowerShell 控制台主机中的可用功能。

例 12-8：默认的控制台主机中的可用功能

```
PS >$host.UI.RawUI | Get-Member |
>>     Select Name,MemberType | Format-Table -Auto
>>
```

Name	MemberType
----	-----
(...)	
FlushInputBuffer	Method
GetBufferContents	Method
GetHashCode	Method
GetType	Method
LengthInBufferCells	Method
NewBufferCellArray	Method
ReadKey	Method
ScrollBufferContents	Method
SetBufferContents	Method
BackgroundColor	Property
BufferSize	Property
CursorPosition	Property
CursorSize	Property
ForegroundColor	Property
KeyAvailable	Property
MaxPhysicalWindowSize	Property
MaxWindowSize	Property
WindowPosition	Property
WindowSize	Property
WindowTitle	Property

如果你的脚本依赖于 \$host.UI.RawUI 中主机特定的功能，那么当你的脚本要在其他机器上运行的时候，就需要修改了。

12.9 编程：向你的脚本中添加一个图形用户界面

虽然这一章的其余部分中提供的技术通常是你需要的，但有时提供图形用户界面与用户进行交互也是很有帮助的。

由于 PowerShell 完全支持传统的可执行文件，所以简单的程序通常可以满足要求。如果在 Visual Studio 这样的环境中创建一个简单的程序不方便，经常可以使用 PowerShell 直接创建这些应用程序。

例 12-9 演示了单独使用 PowerShell 脚本开发 Windows 窗体应用程序的技术。

例 12-9：Select-GraphicalFilteredObject.ps1

```
#####
##          .NET Framework 4.0+ required
## Select-GraphicalFilteredObject.ps1
##
## Display a Windows Form to help the user select a list of items piped in.
## Any selected items get passed along the pipeline.
##
```

```
## ie:  
##  
## PS >dir | Select-GraphicalFilteredObject  
##  
##     Directory: Microsoft.PowerShell.Core\FileSystem::C:\  
##  
##  
## Mode           LastWriteTime      Length Name  
## ----          -----          ----- ----  
## d----  10/7/2006 4:30 PM          Documents and Settings  
## d----  3/18/2007 7:56 PM          Windows  
##  
#####  
  
$objectArray = @($input)  
  
## Ensure that they've piped information into the script  
if($objectArray.Count -eq 0)  
{  
    Write-Error "This script requires pipeline input."  
    return  
}  
  
## Load the Windows Forms assembly  
[void] [Reflection.Assembly]::LoadWithPartialName("System.Windows.Forms")  
  
## Create the main form  
$form = New-Object Windows.Forms.Form  
$form.Size = New-Object Drawing.Size @(600,600)  
  
## Create the listbox to hold the items from the pipeline  
$listbox = New-Object Windows.Forms.CheckedListBox  
$listbox.CheckOnClick = $true  
$listbox.Dock = "Fill"  
$form.Text = "Select the list of objects you wish to pass down the pipeline"  
$listBox.Items.AddRange($objectArray)  
  
## Create the button panel to hold the OK and Cancel buttons  
$buttonPanel = New-Object Windows.Forms.Panel  
$buttonPanel.Size = New-Object Drawing.Size @(600,30)  
$buttonPanel.Dock = "Bottom"  
  
## Create the Cancel button, which will anchor to the bottom right  
$cancelButton = New-Object Windows.Forms.Button  
$cancelButton.Text = "Cancel"  
$cancelButton.DialogResult = "Cancel"  
$cancelButton.Top = $buttonPanel.Height - $cancelButton.Height - 5  
$cancelButton.Left = $buttonPanel.Width - $cancelButton.Width - 10  
$cancelButton.Anchor = "Right"  
  
## Create the OK button, which will anchor to the left of Cancel  
$okButton = New-Object Windows.Forms.Button  
$okButton.Text = "Ok"  
$okButton.DialogResult = "Ok"  
$okButton.Top = $cancelButton.Top
```

```
$okButton.Left = $cancelButton.Left - $okButton.Width - 5
$okButton.Anchor = "Right"

## Add the buttons to the button panel
$buttonPanel.Controls.Add($okButton)
$buttonPanel.Controls.Add($cancelButton)

## Add the button panel and list box to the form, and also set
## the actions for the buttons
$form.Controls.Add($listBox)
$form.Controls.Add($buttonPanel)
$form.AcceptButton = $okButton
$form.CancelButton = $cancelButton
$form.Add_Shown( { $form.Activate() } )

## Show the form, and wait for the response
$result = $form.ShowDialog()
```

```
## If they pressed OK (or Enter,) go through all the
## checked items and send the corresponding object down the pipeline
if($result -eq "OK")
{
```

```
    foreach($index in $listBox.CheckedIndices)
    {
        $objectArray[$index]
    }
}
```

关于运行脚本的更多信息，请参见 1.1 节。

参考 参见第 1 章，了解更多关于如何使用 PowerShell 的信息。有关如何使用 PowerShell 与 Windows API 进行交互的信息，请参见第 1.1 节。

如果想要在命令行中使用 PowerShell，可以在命令提示符窗口中输入 powershell -command "你的命令"。例如：

1.8 | 命令行工具

返回

案式光碟

示例 1-11 回取、显示和处理命令行输入。这段脚本从命令行读取输入，然后将其输出到文件中。

第 13 章

跟踪和错误管理

13.0 简介

如果脚本没有按计划执行该怎么办？这是任何系统中的错误管理的核心问题，它在 PowerShell 脚本中占据大部分内容。

PowerShell 对错误管理的支持提供了几个唯一的功能可以使你的工作更容易：主要优点是区分中断式和非中断式错误。

当运行复杂的脚本或任务的时候，你的脚本可能因为打不开正在操作的 1000 个文件中的一个而崩溃，虽然它应该使你了解失败，但是脚本仍会继续操作下一个文件，这是一个非中断式错误；当脚本在运行备份任务的时候，如果磁盘空间不够了，脚本就会退出，这是一个中断式错误。

通过这种区分，PowerShell 可以让你管理脚本或程序中的错误，也允许你生成错误。

13.1 查看由某一命令生成的错误

问题

你想查看当前会话中发生的错误。

解决方案

若要访问到目前为止生成错误的列表，需要使用 \$error 变量，如例 13-1 所示。

例 13-1：查看变量 \$error 中的内容

```
PS >1/0
Attempted to divide by zero.
At line:1 char:3
+ 1/0 <<<
PS >$error[0] | Format-List -Force

ErrorRecord      : Attempted to divide by zero.
StackTrace       :      at System.Management.Automation.Parser.ExpressionNode.A
                     (...)

Message          : Attempted to divide by zero.
Data             : {}

InnerException   : System.DivideByZeroException: Attempted to divide by zero.
                  at System.Management.Automation.ParserOps.polyDiv(Execu
                  val, Object rval)
TargetSite        : System.Collections.ObjectModel`1[System.Manag
                     ements.IEnumerable]

HelpLink         : 
Source           : System.Management.Automation
```

讨论

PowerShell 的 \$error 变量会保存当前会话中到目前为止发生的错误的列表。其中包括中断错误和非中断错误。

默认情况下，PowerShell 在自定义的视图中显示错误记录。如果你想在表或列表中查看错误（使用 Format-Table 或 Format-List 命令），你必须指定 -Force 以重写自定义的视图。

如果你想以更加紧凑方式显示错误，PowerShell 还支持通过 \$errorView 首选项变量来设置称为 CategoryView 的视图的方法，如下：

```
PS >Get-ChildItem IDoNotExist
Get-ChildItem : Cannot find path 'C:\IDoNotExist' because it does not exist.
At line:1 char:4
+ Get-ChildItem <<< IDoNotExist
PS >$errorView = "CategoryView"
PS >Get-ChildItem IDoNotExist
ObjectNotFound: (C:\IDoNotExist:String) [Get-ChildItem], ItemNotFoundException
```

要清除错误，调用 \$error 的 Clear() 方法，如下：

```
PS >$error.Count
2
PS >$error.Clear()
PS >$error.Count
0
```

关于 PowerShell 的首选变量的更多信息，请参见附录 C “PowerShell 自动变量”。如果你要检测最后一个命令运行成功与失败，请参见 1.10 节。

参考

- 附录 C “PowerShell 自动变量”
- 1.10 节

13.2 处理警告、错误和终止错误

问题

你希望处理警告、错误和终止你调用的脚本或其他工具生成的错误。

解决方案

若要控制你的脚本如何响应警告消息，需要设置 \$warningPreference 变量。在下面的示例中，要忽略警告消息：

```
$warningPreference = "SilentlyContinue"
```

若要控制你的脚本如何响应非终止错误，需要设置 \$errorActionPreference 变量。在下面的示例中，要忽略警告消息：

```
$errorActionPreference = "SilentlyContinue"
```

若要控制你的脚本如何响应终止错误，需要使用错误捕获语句。在下面的示例中，输出一条消息并继续此脚本：

```
trap [DivideByZeroException] { "Don't divide by zero!"; continue }
```

讨论

PowerShell 定义了多个首选项变量，帮助你控制脚本如何响应警告、错误和中断式错误。

作为这些错误管理技术的示例，请思考下面的脚本：

```
#####
## Get-WarningsAndErrors.ps1
##
## Demonstrates the functionality of the Write-Warning, Write-Error, and
```

```

throw
## statements
##
#####
Write-Warning "Warning: About to generate an error"
Write-Error "Error: You are running this script"
throw "Could not complete operation."

```

关于运行脚本的更多信息，请参见 1.1 节。

你现在可以看到一个脚本是如何管理这些不同类型的错误的，例如：

```

PS >$warningPreference = "Continue"
PS >Get-WarningsAndErrors.ps1
WARNING: Warning: About to generate an error
.. Get-WarningsAndErrors.ps1 : Error: You are
running this script
At line:1 char:27
+ Get-WarningsAndErrors.ps1 <<<
Could not complete operation.
At .. Get-WarningsAndErrors.ps1:12 char:6
+ throw <<< "Could not complete operation."

```

一旦你修改警告首选项，原始的警告消息将被取消，如下：

```

PS >$warningPreference = "SilentlyContinue"
PS >Get-WarningsAndErrors.ps1
.. Get-WarningsAndErrors.ps1 : Error: You are
running this script
At line:1 char:27
+ Get-WarningsAndErrors.ps1 <<<
Could not complete operation.
At .. Get-WarningsAndErrors.ps1:12 char:6
+ throw <<< "Could not complete operation."

```

当你修改错误首选项时，你取消了错误和例外，如下：

```

PS >$errorActionPreference = "SilentlyContinue"
PS >Get-WarningsAndErrors.ps1
PS >

```

除了 \$errorActionPreference 变量，所有命令都允许你在单独调用过程中指定一个首选项，如下：

```

PS >$errorActionPreference = "Continue"
PS >Get-ChildItem IDoNotExist
Get-ChildItem : Cannot find path '...\\IDoNotExist' because it does not
exist.
At line:1 char:14
+ Get-ChildItem <<< IDoNotExist
PS >Get-ChildItem IDoNotExist -ErrorAction SilentlyContinue
PS >

```

如果你重置错误首选项为 Continue，你可以看到错误捕获语句的结果。Write-Error 调用的消息能够顺利通过，但异常却不能。

```
PS >$errorActionPreference = "Continue"
PS >trap { "Caught an error"; continue }; Get-WarningsAndErrors
.. Get-WarningsAndErrors.ps1 : Error: You are
running this script
At line:1 char:61
+ trap { "Caught an error"; continue }; Get-WarningsAndErrors <<<
Caught an error
```

关于 PowerShell 中错误管理的更多信息，请参见附录 A 中的“管理错误”。关于这些首选变量的有效设置的更多信息，请参见附录 C。

参考

- 附录 A 中的“管理错误”
- 附录 C “PowerShell 内置变量”
- 1.1 节

13.3 输出警告、错误和终止错误

问题

如何使你的脚本通知它的调用方发生了警告、错误或终止错误。

```
#####
## 
## Get-WarningsAndErrors.ps1
##
## Demonstrates the functionality of the Write-Warning, Write-Error, and throw
## statements
##
#####
Write-Warning "Warning: About to generate an error"
Write-Error "Error: You are running this script"
throw "Could not complete operation."
```

解决方案

若要写入警告和错误，分别使用 Write-Warning 和 Write-Error 命令。使用 throw 语句生成一个终止错误。

讨论

当你的脚本发生异常的时候，你可以通过 Write-Warning、Write-Error 和 throw 语句通知脚本的调用者。如果你希望向用户发送多个警告信息，则使用 Write-Warning 命令。如果你的脚本遇到的错误（但可以合理地跳过该错误），则使用 Write-Error 命令。如果该错误是致命的并且使你的脚本不能继续运行，则使用 throw 语句抛出错误。

有关如何处理由脚本引发的这些错误信息，请参阅 13.2 节。

有关在 PowerShell 中错误管理的详细信息，请参阅附录 A 中的“管理错误”。有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节
- 13.2 节
- 附录 A 中的“管理错误”

13.4 调试脚本

问题

如何在脚本中以交互方式诊断故障或意外的行为。

解决方案

要在你的脚本中生成调试语句，需要使用 Write-Debug。如果要逐步调试一个区域，则要用 Set-PsDebug -Step 把它包围起来。若要了解在执行的特定点处的环境，要添加一行对 \$host.EnterNestedPrompt() 的调用。

讨论

默认情况下，PowerShell 可以将数据分配给你尚未创建的变量（从而创建这些变量）。它还允许你从不存在的变量中检索数据，这通常会发生意外情况并会导致错误。为了帮助你摆脱此问题，PowerShell 提供了一种严格（strict）的模式，如果你尝试访问一个不存在的变量就会产生错误。例 13-2 演示了这个模式。

例 13-2：PowerShell 以 strict 模式操作

```
PS >$testVariable = "Hello"  
PS >$tsetVariable += " World"  
PS >$testVariable  
Hello  
PS >Remove-Item Variable:\tsetvariable  
PS >Set-PsDebug -Strict  
PS >$testVariable = "Hello"  
PS >$tsetVariable += " World"  
The variable $tsetVariable cannot be retrieved because it has not been set  
yet.  
At line:1 char:14  
+ $tsetVariable <<< += " World"
```

为了保证你的脚本调试运行状况的稳定，严格的模式应该第一个添加到你的 PowerShell 配置文件中。

在交互式调试（相对于错误保护）时，你可能会习惯于使用几个 PowerShell 支持的最有用的调试功能，包括：跟踪（通过 Set-PsDebug -Trace 语句），逐步（通过 Set-PsDebug -Step 语句）和环境检查（通过 \$host.EnterNestedPrompt() 调用）。

作为这些技术的演示，请参见例 13-3。

例 13-3：一个与 PowerShell 调试功能相交互的脚本

```
#####
##  
## Invoke-ComplexScript.ps1  
##  
## Demonstrates the functionality of PowerShell's debugging support.  
##  
#####  
  
Write-Host "Calculating lots of complex information"  
  
$runningTotal = 0  
$runningTotal += [Math]::Pow(5 * 5 + 10, 2)  
  
Write-Debug "Current value: $runningTotal"  
  
Set-PsDebug -Trace 1  
$dirCount = @(Get-ChildItem $env:WINDIR).Count  
  
Set-PsDebug -Trace 2  
$runningTotal -= 10  
$runningTotal /= 2  
  
Set-PsDebug -Step  
$runningTotal *= 3  
$runningTotal /= 2
```

```
$host.EnterNestedPrompt()
```

```
Set-PsDebug -off
```

有关运行脚本的更多信息，请参阅 1.1 节。

当你想确定为什么此脚本不像你期望的那样工作时，调试会话可能看上去像例 13-4。

例 13-4：调试一个复杂的脚本

```
PS >$debugPreference = "Continue"
```

```
PS >Invoke-ComplexScript.ps1
```

```
Calculating lots of complex information
```

```
DEBUG: Current value: 1225
```

```
DEBUG: 17+ $dirCount = @(Get-ChildItem $env:WINDIR).Count
```

```
DEBUG: 17+ $dirCount = @(Get-ChildItem $env:WINDIR).Count
```

```
DEBUG: 19+ Set-PsDebug -Trace 2
```

```
DEBUG: 20+ $runningTotal -= 10
```

```
DEBUG: ! SET $runningTotal = '1215'.
```

```
DEBUG: 21+ $runningTotal /= 2
```

```
DEBUG: ! SET $runningTotal = '607.5'.
```

```
DEBUG: 23+ Set-PsDebug -Step
```

Continue with this operation?

```
24+ $runningTotal *= 3
```

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y") :y

```
DEBUG: 24+ $runningTotal *= 3
```

```
DEBUG: ! SET $runningTotal = '1822.5'.
```

Continue with this operation?

```
25+ $runningTotal /= 2
```

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y") :y

```
DEBUG: 25+ $runningTotal /= 2
```

```
DEBUG: ! SET $runningTotal = '911.25'.
```

Continue with this operation?

```
27+ $host.EnterNestedPrompt()
```

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y") :y

```
DEBUG: 27+ $host.EnterNestedPrompt()
```

```
DEBUG: ! CALL method 'System.Void EnterNestedPrompt()'
```

```
PS >$dirCount
```

```
296
```

```
PS >$dirCount + $runningTotal
```

```
1207.25
```

```
PS >exit
```

Continue with this operation?

```
29+ Set-PsDebug -off
```

[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help
(default is "Y") :y

```
DEBUG: 29+ Set-PsDebug -off
```

虽然没有内置的图形用户界面，但 PowerShell 的交互式调试功能还是可以帮助你诊断并迅速解决问题。

有关 Set-PsDebug 命令的更多信息，请键入 **Get-Help Set-PsDebug**。

参考

- 1.1 节

13.5 收集脚本或命令的详细的跟踪信息

问题

你希望访问有关脚本或命令执行的详细的调试或诊断信息。

解决方案

若要在脚本执行时进行跟踪，可以使用 Set-PsDebug 命令的 -Trace 参数。

若要查看 PowerShell 引擎和其命令的详细跟踪输出信息，可以使用 Trace-Command 命令。

讨论

Set-PsDebug 命令使你在脚本执行的过程中可以配置调试明细的数据量。通过设置 -Trace 参数，PowerShell 还可以让你在执行脚本的时候查看脚本的行。有关 Set-PsDebug 命令的更多信息，请参阅 13.4 节。

当想要检查你的代码与 PowerShell 的交互情况，或使用 PowerShell 命令的情况等问题时，可以使用 Trace-Command 命令。Trace-Command 命令提供了大量明细数据，可以用来进行深入的故障分析。

例如，要获得一些类似于 PowerShell 无法查找你的脚本的可能原因等深入的信息，可以输入命令：

```
Trace-Command CommandDiscovery -PsHost { ScriptInTheCurrentDirectory.ps1 }
```

Trace-Command 命令需要一个跟踪源（trace source）（例如 *CommandDiscovery*）、一个目标位置（通常是 -PsHost 或 -File）和一个脚本块来进行跟踪。这个命令的输出表

明了 PowerShell 永远不会为你的脚本实际地搜索当前目录，因此你需要显式指明脚本的位置：`.\ScriptInTheCurrentDirectory.ps1`。

有关 `Trace-Command` 命令的更多信息，请键入 `Get-Help Trace-Command`。若要了解哪些跟踪源可用，请参阅 `Get-TraceSource` 命令。

参考

- 13.4 节

13.6 编程：分析脚本的性能的配置文件

当你编写的脚本与用户大量交互的时候，你的脚本可能会得益于更好的性能。

当处理性能问题的时候，第一个规则是测量该问题。除非你可以使用硬盘性能数据来指导你的优化工作，否则你肯定会把自己的工作引向错误点。随机的性能方面的改进会很快将你的代码变成一个不可读的混乱的甚至没有任何性能可言的代码堆积！低级别的性能优化应该始终以硬盘数据的支持为指导。

你可以通过事件探查器来获取硬盘性能数据。PowerShell 本身不附带事件探查器，但你可以通过例 13-5 来进行实现。

例 13-5: Get-ScriptPerformanceProfile.ps1

```
#####
## .beignce snli-ed oI heignsa.com to inidun efe sige pse uo nse
## Get-ScriptPerformanceProfile.ps1 ew esiqmca vram wot tuo eriit, zelA se
## (jE = 1000000)
## Computes the performance characteristics of a script, based on the transcript
## of it running at trace level 1.
##
## To profile a script:
##   1) Turn on script tracing in the window that will run the script:
##       Set-PsDebug -trace 1
##   2) Turn on the transcript for the window that will run the script:
##       Start-Transcript
##       (Note the filename that PowerShell provides as the logging destination.)
##   3) Type in the script name, but don't actually start it.
##   4) Open another PowerShell window, and navigate to the directory holding
##       this script. Type in 'Get-ScriptPerformanceProfile <transcript>',
##       replacing <transcript> with the path given in step 2. Don't
##       press <Enter> yet.
##   5) Switch to the profiled script window, and start the script.
##       Switch to the window containing this script, and press <Enter>
##   6) Wait until your profiled script exits, or has run long enough to be
##       representative of its work. To be statistically accurate, your script
```

```

## should run for at least ten seconds.
## 7) Switch to the window running this script, and press a key.
## 8) Switch to the window holding your profiled script, and type:
##     Stop-Transcript
## 9) Delete the transcript.
##
## Note: You can profile regions of code (ie: functions) rather than just lines
## by placing the following call at the start of the region:
##     write-debug "ENTER <region_name>"
## and the following call and the end of the region:
##     write-debug "EXIT"
## This is implemented to account exclusively for the time spent in that
## region, and does not include time spent in regions contained within the
## region. For example, if FunctionA calls FunctionB, and you've surrounded
## each by region markers, the statistics for FunctionA will not include the
## statistics for FunctionB.
##
#####
param($logFilePath = $(throw "Please specify a path to the transcript log file."))
function Main
{
    ## Run the actual profiling of the script. $uniqueLines gets
    ## the mapping of line number to actual script content.
    ## $samples gets a hashtable mapping line number to the number of times
    ## we observed the script running that line.
    $uniqueLines = @{}
    $samples = GetSamples $uniqueLines

    "Breakdown by line:"
    -----
    ## Create a new hash table that flips the $samples hashtable --
    ## one that maps the number of times sampled to the line sampled.
    ## Also, figure out how many samples we got altogether.
    $counts = @{}
    $totalSamples = 0;
    foreach($item in $samples.Keys)
    {
        $counts[$samples[$item]] = $item
        $totalSamples += $samples[$item]
    }

    ## Go through the flipped hashtable, in descending order of number of
    ## samples. As we do so, output the number of samples as a percentage of
    ## the total samples. This gives us the percentage of the time our script
    ## spent executing that line.
    foreach($count in ($counts.Keys | Sort-Object -Descending))
    {
        $line = $counts[$count]
        $percentage = "{0:#0}" -f ($count * 100 / $totalSamples)
        "{0,3}%: Line {1,4} -{2}" -f $percentage,$line,
        $uniqueLines[$line]
    }
}

```

```

## Go through the transcript log to figure out which lines are part of any
## marked regions. This returns a hashtable that maps region names to
## the lines they contain.
##
"Breakdown by marked regions:"$functionMembers = GenerateFunctionMembers
"-----"
$functionMembers = GenerateFunctionMembers

## For each region name, cycle through the lines in the region. As we
## cycle through the lines, sum up the time spent on those lines and output
## the total.
foreach($key in $functionMembers.Keys)
{
    $totalTime = 0
    foreach($line in $functionMembers[$key])
    {
        $totalTime += ($samples[$line] * 100.0 / $totalSamples)
    }

    $percentage = "{0:#0}" -f $totalTime
    "{0,3}%: {1}" -f $percentage,$key
}

## Run the actual profiling of the script. $uniqueLines gets
## the mapping of line number to actual script content.
## Return a hashtable mapping line number to the number of times
## we observed the script running that line.
function GetSamples($uniqueLines)
{
    ## Open the log file. We use the .Net file I/O, so that we keep monitoring
    ## just the end of the file. Otherwise, we would make our timing inaccurate
    ## as we scan the entire length of the file every time.
    $logStream = [System.IO.File]::Open($logFilePath, "Open", "Read", "ReadWrite")
    $logReader = New-Object System.IO.StreamReader $logStream

    $random = New-Object Random
    $samples = @{}
    $lastCounted = $null

    ## Gather statistics until the user presses a key.
    while(-not $host.UI.RawUI.KeyAvailable)
    {
        ## We sleep a slightly random amount of time. If we sleep a constant
        ## amount of time, we run the very real risk of improperly sampling
        ## scripts that exhibit periodic behaviour.
        $sleepTime = [int] ($random.NextDouble() * 100.0)
        Start-Sleep -Milliseconds $sleepTime

        ## Get any content produced by the transcript since our last poll.
        ## From that poll, extract the last DEBUG statement (which is the last
        ## line executed.)
        $rest = $logReader.ReadToEnd()
        $lastEntryIndex = $rest.LastIndexOf("DEBUG: ")
}

```

```

## If we didn't get a new line, then the script is still working on the
## last line that we captured.
if($lastEntryIndex -lt 0)
{
    if($lastCounted) { $samples[$lastCounted] ++ }
    continue;
}

## Extract the debug line.
$lastEntryFinish = $rest.IndexOf("\n", $lastEntryIndex)
if($lastEntryFinish -eq -1) { $lastEntryFinish = $rest.length }
$scriptLine = $rest.Substring(
    $lastEntryIndex, ($lastEntryFinish -
$lastEntryIndex)).Trim()
if($scriptLine -match 'DEBUG:[ \t]*([0-9]*)\+([.*])')
{
    ## Pull out the line number from the line
    $last = $matches[1]

    $lastCounted = $last
    $samples[$last] ++

    ## Pull out the actual script line that matches the line number
    $uniqueLines[$last] = $matches[2]
}

## Discard anything that's buffered during this poll, and start waiting
## again
$logReader.DiscardBufferData()

}

## Clean up
$logStream.Close()
$logReader.Close()

$samples
}

## Go through the transcript log to figure out which lines are part of any
## marked regions. This returns a hashtable that maps region names to
## the lines they contain.
function GenerateFunctionMembers
{
    ## Create a stack that represents the callstack. That way, if a marked
    ## region contains another marked region, we attribute the statistics
    ## appropriately.
    $callstack = New-Object System.Collections.Stack
    $currentFunction = "Unmarked"
    $callstack.Push($currentFunction)

    $functionMembers = @{}

    ## Go through each line in the transcript file, from the beginning
    foreach($line in (Get-Content $logFilePath))
    {
        if($line -match 'DEBUG:[ \t]*([0-9]*)\+([.*])')
        {
            $lastCounted = $matches[1]
            $scriptLine = $matches[2]
            $callstack.Push($scriptLine)
            $functionMembers[$scriptLine] = $callstack
        }
        else
        {
            $scriptLine = $callstack.Pop()
            $functionMembers[$scriptLine] = $callstack
        }
    }
}

```

```
## Check if we're entering a monitor block
## If so, store that we're in that function, and push it onto
## the callstack.
if($line -match 'write-debug "ENTER (.*)"')
{
    $currentFunction = $matches[1]
    $callstack.Push($currentFunction)
}

## Check if we're exiting a monitor block
## If so, clear the "current function" from the callstack,
## and store the new "current function" onto the callstack.
elseif($line -match 'write-debug "EXIT"')
{
    [void] $callstack.Pop()
    $currentFunction = $callstack.Peek()
}

## Otherwise, this is just a line with some code.
## Add the line number as a member of the "current function"
else
{
    if($line -match 'DEBUG:[ \t]*([0-9]*)\+')
    {
        ## Create the arraylist if it's not initialized
        if(-not $functionMembers[$currentFunction])
        {
            $functionMembers[$currentFunction] =
                New-Object System.Collections.ArrayList
        }

        ## Add the current line to the ArrayList
        if(-not $functionMembers[$currentFunction].Contains($matches[1]))
        {
            [void] $functionMembers[$currentFunction].Add($matches[1])
        }
    }
}

$functionMembers
}

.Main
```

关于运行脚本的更多的信息，请参见 1.1 节。

参考

- 1.1 节

第 14 章

掌握环境

14.0 简介

你的许多脚本在设计的时候工作在隔离的环境中，但是你会发现了解一些与脚本执行环境有关的信息是非常有用的，如：脚本的名称、当前工作目录、环境变量、常见的系统路径等。

PowerShell 提供了几种方法来获得此信息，包括从它的 cmdlet，到内置的变量，再到 .NET Framework 中提供的功能。

14.1 查看和修改环境变量

问题

如何与系统的环境变量进行交互。

解决方案

要与环境变量交互，可以采用与访问常规的 PowerShell 变量相同的方式访问它们。

唯一的区别是你在 (\$) 符号和变量名之间放置了 env:，如下：

```
PS >$env:Username  
Lee
```

你也可以采用这种以下方式来修改环境变量。例如，要暂时将当前目录添加到路径变量中：

```
PS >Invoke-DemonstrationScript
The term 'Invoke-DemonstrationScript' is not recognized as a cmdlet, function, operable program, or script file. Verify the term and try again.
At line:1 char:26
+ Invoke-DemonstrationScript <<<
PS >$env:PATH = $env:PATH + ";".
PS >Invoke-DemonstrationScript.ps1
The script ran!
```

讨论

在批处理文件中，环境变量的主要用途是用来存储临时信息，或在批处理文件之间传送信息。PowerShell中的变量和脚本参数是更有效的解决这些问题的方式，但是环境变量提供了一种有用的方法来访问常见的系统设置，例如系统的路径、临时目录、域名、用户名和更多内容。

PowerShell 通过其环境提供程序（environment provider）来访问环境变量，使你可以像使用文件系统或注册表提供程序中的项那样使用环境变量。默认情况下 PowerShell 定义一个 `env:`（如 `c:` 或 `d:`）提供对此类信息的访问：

```
PS >dir env:
Name          Value
----          -----
Path          c:\program files\ruby\bin;C:\WINDOWS\system32;C:\DOCUME~1\Lee\LOCALS~1\Temp
TEMP          Console
SESSIONNAME
PATHEXT      .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;
(...)
```

既然是一个规则的 PowerShell 驱动，那么获得一个环境变量的值的完整方法就如下所示：

```
PS >Get-Content Env:\Username
Lee
```

当然，作为环境变量来说，你几乎可以不使用这种方式，因为 PowerShell 支持 `Get-Content` 和 `SetContent` 变量语法（variable syntax），可以简化语法如下：

```
PS >$env:Username
Lee
```

此语法适用于所有驱动，但最常用于访问环境变量。有关此语法的详细信息，请参阅 14.2 节。

实际上，某些环境变量从两个位置获取它们的值的组合：机器范围的设置和当前用户的设置。

如果你要访问的环境变量值是在计算机或用户级别配置的，可以使用 `Environment::GetEnvironmentVariable()` 方法。例如，如果在你的路径中定义了一个工具目录，你可能会看到：

```
PS >[Environment]::GetEnvironmentVariable("Path", "User")
d:\lee\tools
```

若要永久性地设置这些计算机或特定于用户的环境变量，可以使用 `[Environment]::SetEnvironmentVariable()` 方法：

```
[Environment]::SetEnvironmentVariable(<name>, <value>, <target>)
```

`Target` 参数定义了存储该变量的位置：`User` 为当前用户；`Machine` 为计算机上所有用户。

例如，要永久性地将你的工具目录添加到你的路径中：

```
PS >$oldPersonalPath = [Environment]::GetEnvironmentVariable("Path", "User")
PS >$oldPersonalPath += ";d:\tools"
PS >[Environment]::SetEnvironmentVariable("Path", $oldPersonalPath, "User")
```

有关 `Get-content` 和 `Set-content` 语法的详细信息，请参阅第 3 章中的“变量”一节。有关环境提供程序的详细信息，请键入 `Get-help About_Environment`。

参考

- 第 3 章中的“变量”中的“变量和对象”
- 14.2 节

14.2 关于你的命令调用的访问信息

问题

你想了解用户如何调用你的脚本、函数或脚本块。

解决方案

若要访问有关用户如何调用你的命令的信息，可以使用 `$myInvocation` 变量：

```
"You invoked this script by typing: " + $myInvocation.Line
```

讨论

\$myInvocation 变量提供了有关当前脚本、函数或脚本块的大量信息，以及它被调用的上下文：

MyCommand

关于命令（脚本、函数或脚本块）本身的信息。

ScriptLineNumber

脚本中调用该命令的行号。

ScriptName

当出现在函数或脚本块中时，代表调用该命令的脚本名称。

Line

在脚本（或命令行）的行中调用该命令的文本。

InvocationName

用户提供的用来调用该命令的名称。如果用户为命令定义了别名，那么这是不同于 MyCommand 提供的信息。

PipelineLength

调用该命令的管道中的命令数。

PipelinePosition

调用该命令的管道中该命令的位置。

在使用 \$myInvocation 变量的时候需要注意的一点是，它的改变取决于调用它的命令的类型。如果你是从一个函数中访问此信息，那么它提供特定于该函数的信息而不是特定于调用它的脚本的信息。因为脚本、函数和脚本块都是唯一的，所以 \$myInvocation. MyCommand 变量中的信息会在不同的命令类型之间发生变化。

脚本

Definition 和 Path

当前运行的脚本的完整路径

Name

当前运行的脚本的名称

CommandType

总是 ExternalScript

函数

Definition 和 ScriptBlock

当前正在运行的函数的源代码

Options

应用于当前正在运行的函数的选项（无、只读、常量、私有、全局）

Name

当前正在运行的函数的名称

CommandType

总是 Function

脚本块

Definition 和 ScriptBlock

当前正在运行的脚本块的源代码

Name

空

CommandType

总是 Script

14.3 编程：研究请求信息变量

当尝试通过 \$myInvocation 变量访问有效信息的时候，了解此信息如何在脚本、函数和脚本块之间更改会有所帮助。为了更好的了解由变量 \$myInvocation 提供的资源，可以查看例 14-1 的输出结果。

例 14-1：Get-InvocationInfo.ps1

```
#####
## Get-InvocationInfo.ps1
##
## Display the information provided by the $myInvocation variable
##
param([switch] $preventExpansion)

## Define a helper function, so that we can see how $myInvocation changes
## when it is called, and when it is dot-sourced
function HelperFunction
{
    " MyInvocation from function:"
```

```
'''*50
$myInvocation

    " Command from function:"
'''*50
$myInvocation.MyCommand
}

## Define a script block, so that we can see how $myInvocation changes
## when it is called, and when it is dot-sourced
$myScriptBlock = {
    " MyInvocation from script block:"
'''*50
$myInvocation

    " Command from script block:"
'''*50
$myInvocation.MyCommand
}

## Define a helper alias
Set-Alias gii Get-InvocationInfo

## Illustrate how $myInvocation.Line returns the entire line that the
## user typed.
"You invoked this script by typing: " + $myInvocation.Line

## Show the information that $myInvocation returns from a script
"MyInvocation from script:"
'''*50
$myInvocation

"Command from script:"
'''*50
$myInvocation.MyCommand

## If we were called with the -PreventExpansion switch, don't go
## any further
if($preventExpansion)
{
    return
}

## Show the information that $myInvocation returns from a function
"Calling HelperFunction"
'''*50
HelperFunction

## Show the information that $myInvocation returns from a dot-sourced
## function
"Dot-Sourcing HelperFunction"
'''*50
. HelperFunction

## Show the information that $myInvocation returns from an aliased script
"Calling aliased script"
```

```
--*50
gii -PreventExpansion

## Show the information that $myInvocation returns from a script block
"Calling script block"
--*50
& $myScriptBlock

## Show the information that $myInvocation returns from a dot-sourced
## script block
"Dot-Sourcing script block"
--*50
. $myScriptBlock

## Show the information that $myInvocation returns from an aliased script
"Calling aliased script"
--*50
gii -PreventExpansion
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

14.4 找到脚本的名称

问题

你想知道当前正在运行的脚本的名称。

解决方案

要确定当前正在执行的脚本的完整的路径和文件名，可以使用下面的函数：

```
function Get-ScriptName
{
    $myInvocation.ScriptName
}
```

若要确定用户调用你脚本（例如，在一个“Usage”消息中时）实际键入的名称，可以使用 \$myInvocation.InvocationName 变量。

讨论

通过在函数中放置 \$myInvocation.ScriptName 语句，我们大大简化了用来确定当前

正在运行脚本的名称的逻辑。如果你不希望使用函数，你也可以直接调用一个脚本块，同样可以简化确定当前正在运行脚本的名称的逻辑，如下：

```
$scriptName = & { $myInvocation.ScriptName }
```

虽然这是获得当前脚本名称的一个相当复杂的方法，另一种方法却更容易。如你正处于一个脚本的正文中，可以通过键入下面的命令直接获取当前脚本的名称：

```
$myInvocation.Path
```

如果你正在一个函数或脚本块中，那么你必须使用下面的命令来获得脚本名称：

```
$myInvocation.ScriptName
```

使用 \$myInvocation.InvocationName 变量有时比较棘手，因为在脚本中直接使用时返回的是脚本名称，而在该脚本中的函数中调用的时候不是返回脚本名称。

有关使用 \$myInvocation 变量的详细信息，请参阅 14.2 节。

参考

- 14.2 节

14.5 找到你的脚本的位置

问题

如何知道当前正在运行的脚本的位置。

解决方案

要确定当前正在执行的脚本的位置，可以使用下面的函数：

```
function Get-ScriptPath
{
    Split-Path $myInvocation.ScriptName
}
```

讨论

一旦我们了解到脚本的完整路径，就可以通过 Split-Path 命令方便地确定其位置。它的同级，可以使用 Join-Path 命令组织形成新的路径。

通过访问函数中的 \$myInvocation.ScriptName 变量，大大简化了确定当前正在运行脚本的位置的逻辑。有关为此目的而使用函数的备选方案的讨论，请参阅 14.4 节。

有关使用 \$myInvocation 变量的详细信息，请参阅 14.2 节。有关 Join-Path 命令的更多信息，请参阅 14.9 节。

参考

- 14.2 节
- 14.4 节
- 14.9 节

14.6 查找常见的系统路径的位置

问题

如何知道常见的系统路径和特殊文件夹，如我的文档（My Documents）及程序文件夹（Program Files）的位置。

解决方案

若要确定常见的系统路径和特殊文件夹的位置，可以使用 [Environment]::GetFolderPath() 方法，如下：

```
PS >[Environment]::GetFolderPath("System")
C:\WINDOWS\system32
```

对于不支持此方法（如所有用户开始菜单）的路径，可以使用 WScript.Shell COM 对象：

```
$shell = New-Object -Com WScript.Shell
$allStartMenu = $shell.SpecialFolders.Item("AllUsersStartMenu")
```

讨论

[Environment]::GetFolderPath() 方法允许你访问许多在 Windows 中常用的位置。在使用的时候，需要提供位置的代名词（如 System 或 Personal）。因为你可能不具备所有提供的这些代名词，所以查看所有这些值的一种方法是使用 [Enum]::GetValues() 方法，如例 14-2 所示。

例 14-2:[Environment]::GetFolderPath()方法支持的文件夹

```
PS >[Enum]::GetValues([Environment+SpecialFolder])
Desktop
Programs
Personal
Favorites
Startup
Recent
SendTo
StartMenu
MyMusic
DesktopDirectory
MyComputer
Templates
ApplicationData
LocalApplicationData
InternetCache
Cookies
History
CommonApplicationData
System
ProgramFiles
MyPictures
CommonProgramFiles
```

由于这是一项常见的任务，所以 PowerShell 如果不能转换你的输入，就会在错误信息中提供可能的值，如下：

```
PS >[Environment]::GetFolderPath("aouaoue")
Cannot convert argument "0", with value: "aouaoue", for "GetFolderPath" to
type "System.Environment+SpecialFolder": "Cannot convert value "aouaoue"
to type "System.Environment+SpecialFolder" due to invalid enumeration values.
Specify one of the following enumeration values and try again. The possible
enumeration values are "Desktop, Programs, Personal, MyDocuments, Favorites,
Startup, Recent, SendTo, StartMenu, MyMusic, DesktopDirectory, MyComputer,
Templates, ApplicationData, LocalApplicationData, InternetCache,
Cookies, History, CommonApplicationData, System, ProgramFiles, MyPictures,
CommonProgramFiles"."
At line:1 char:29
+ [Environment]::GetFolderPath( <<< "aouaoue")
```

尽管此方法提供了对最常见的常见系统路径的访问，但还是有些路径无法通过此方法被访问到。对于那些无法使用 [Environment]::GetFolderPath() 方法获得的路径，可以使用 WScript.Shell COM 对象来获得。WScript.Shell COM 对象支持以下路径：AllUsersDesktop、AllUsersStartMenu、AllUsersPrograms、AllUsersStartup、桌面 (Desktop)、收藏夹 (Favorites)、字体 (Fonts)、我的文档 (MyDocuments)、网上邻居 (NetHood)、PrintHood、程序 (Programs)、最近 (Recent)、发送到 (Send TO)、开始菜单 (StartMenu)、启动 (Startup) 和模板 (Templates)。

你可以使用 [Environment]::GetFolderPath() 方法或 WScript.Shell COM 对象来获得系统的目录，但两者支持的内容存在差异，如例 14-3 所示。

例 14-3：[Environment]::GetFolderPath() 和 Wscript.Shell COM object 之间的差异

```
PS >$shell = New-Object -Com WScript.Shell
PS >$shellPaths = $shell.SpecialFolders | Sort-Object
PS >
PS >$netFolders = [Enum]::GetValues([Environment+SpecialFolder])
PS >$netPaths = $netFolders |
>>     Foreach-Object { [Environment]::GetFolderPath($_) } | Sort-Object
>>
PS >## See the shell-only paths
PS >Compare-Object $shellPaths $netPaths |
>>     Where-Object { $_.SideIndicator -eq "<=" }.
>>

InputObject                               SideIndicator
-----
C:\Documents and Settings\All Users\Desktop           <=
C:\Documents and Settings\All Users\Start Menu         <=
C:\Documents and Settings\All Users\Start Menu\Programs <=
C:\Documents and Settings\All Users\Start Menu\Programs\... <=
C:\Documents and Settings\Lee\NetHood                 <=
C:\Documents and Settings\Lee\PrintHood                <=
C:\Windows\Fonts                                     <=
```



```
PS >## See the .NET-only paths
PS >Compare-Object $shellPaths $netPaths |
>>     Where-Object { $_.SideIndicator -eq ">" }
>>

InputObject                               SideIndicator
-----
C:\Documents and Settings\All Users\Application Data      =>
C:\Documents and Settings\Lee\Cookies                   =>
C:\Documents and Settings\Lee\Local Settings\Application... =>
C:\Documents and Settings\Lee\Local Settings\History       =>
C:\Documents and Settings\Lee\Local Settings\Temporary I... =>
C:\Program Files                                =>
C:\Program Files\Common Files                      =>
C:\WINDOWS\system32                            =>
d:\lee                                         =>
D:\Lee\My Music                                 =>
D:\Lee\My Pictures                            =>
```

有关使用 .NET Framework 中的类的详细信息，请参阅 3.4 节。

参考

- 3.4 节

14.7 编程：搜索 Windows 开始菜单

在使用命令行时，你可能要启动通常只位于你的“开始”菜单中的程序。当然也可以通过单击开始菜单来查找，也可以使用脚本来搜索开始菜单，如例 14-4 所示。

例 14-4：Search-StartMenu.ps1

```
#####
## Search-StartMenu.ps1
##
## Search the Start Menu for items that match the provided text. This script
## searches both the name (as displayed on the Start Menu itself,) and the
## destination of the link.
##
## ie:
##
## PS >Search-StartMenu "Character Map" | Invoke-Item
## PS >Search-StartMenu "network" | Select-FilteredObject | Invoke-Item
##
#####
param(
    $pattern = $(throw "Please specify a string to search for.")
)
## Get the locations of the start menu paths
$myStartMenu = [Environment]::GetFolderPath("StartMenu")
$shell = New-Object -Com WScript.Shell
$allStartMenu = $shell.SpecialFolders.Item("AllUsersStartMenu")

## Escape their search term, so that any regular expression
## characters don't affect the search
$escapedMatch = [Regex]::Escape($pattern)

## Search for text in the link name
dir $myStartMenu *.lnk -rec | ? { $_.Name -match "$escapedMatch" }
dir $allStartMenu *.lnk -rec | ? { $_.Name -match "$escapedMatch" }

## Search for text in the link destination
dir $myStartMenu *.lnk -rec |
    Where-Object { $_ | Select-String "\\\\"*$escapedMatch\.\" -Quiet }
dir $allStartMenu *.lnk -rec |
    Where-Object { $_ | Select-String "\\\\"*$escapedMatch\.\" -Quiet }
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

14.8 获取当前位置

问题

你希望确定当前位置。

解决方案

若要确定当前位置，可以使用 Get-Location 命令，如下所示：

```
PS >Get-Location  
  
Path  
----  
C:\temp  
PS >$currentLocation = (Get-Location).Path  
PS >$currentLocation  
C:\temp
```

讨论

在使用 .Net 框架的时候，会影响脚本运行的一个问题是 PowerShell 中的“当前位置”的概念与 *PowerShell.exe* 的进程的“当前目录”是不同的，举例来说：

```
PS >Get-Location  
  
Path  
----  
C:\temp  
  
PS >Get-Process | Export-CliXml processes.xml  
PS >$reader = New-Object Xml.XmlTextReader processes.xml  
PS >$reader.BaseURI  
file:///C:/Documents and Settings/Lee/processes.xml
```

PowerShell 单独保留这些概念是因为它支持多个执行的管道。当你在几个 shell 程序间切换的时候，如果其中的某一个修改了进程的当前目录，可能会造成所有的后台任务环境的混乱。

当你在大多数的 .NET 方法中使用文件名时，最好的做法是使用完整的路径名。Resolve-Path 命令简化了这项工作：

```
PS >Get-Location  
  
Path  
----  
C:\temp
```

```
PS >Get-Process | Export-CliXml processes.xml
PS >$reader = New-Object Xml.XmlTextReader (Resolve-Path processes.xml)
PS >$reader.BaseURI
file:///C:/temp/processes.xml
```

如果你想要访问并不存在的路径，就要把 Join-Path 与 Get-Location 结合起来使用，如下：

```
PS >Join-Path (Get-Location) newfile.txt
C:\temp\newfile.txt
```

有关 Join-Path 命令的更多信息，请参阅下面的 14.9 节。

参考

- 14.9 节

14.9 安全地生成程序文件路径

问题

你想要生成一个子路径合成之外的新的路径。

解决方案

要把一个路径的元素加在一起，使用 Join-Path 命令：

```
PS >Join-Path (Get-Location) newfile.txt
C:\temp\newfile.txt
```

讨论

要创建新的路径，通常的方法是组合组件，并在它们之间放置一个路径分隔符字符串：

```
PS >"$(Get-Location)\newfile.txt"
C:\temp\newfile.txt
```

遗憾的是，这种方法会产生一些问题：

- 如果 Get-Location 返回的目录已经有一个反斜杠结尾该怎么办？
- 如果路径中包含正斜杠而不是反斜杠该怎么办？
- 如果我们谈论的是注册表路径而不是文件系统路径该怎么办？

幸运的是，Join-Path 命令解决了这些问题。

有关 Join-Path 命令的更多信息，请键入 **Get-help Join-Path**。

14.10 与 PowerShell 的全局环境进行交互

问题

如何将信息存储在 PowerShell 环境中，以便其他脚本可以访问它。

解决方案

若要使一个变量在整个 PowerShell 会话中都可用，那么当你向变量存储信息的时候，要使用一个 \$GLOBAL: 前缀，如下：

```
## Create the web service cache, if it doesn't already exist
if(-not (Test-Path Variable:\Lee.Holmes.WebServiceCache))
{
    ${GLOBAL:Lee.Holmes.WebServiceCache} = @{}
}
```

如果你的脚本的主要目的是为它的调用方提供永久的函数和变量，那么可以把该脚本作为一个库来处理，让调用者 dot-source 这个脚本：

```
PS > . LibraryDirectory
PS >Get-DirectorySize
Directory size: 53,420 bytes
```

讨论

在存储信息时需要注意的是，只要可能，应该避免在该会话的全局环境中存储信息。在全局作用域中存储信息的脚本都容易破坏其他脚本的执行，更易于被其他脚本破坏。

在批处理文件编程中这是通用的方法，但脚本的参数和返回值通常会提供一个更好的选择。

如果你发现自己确实需要将变量写入全局作用域，那么请确保它们在被创建的时候具有唯一名称，以防止与其他脚本冲突，如解决方案中所示。用于命名脚本前缀的好方法包括脚本名、作者的名称或公司名称。

有关在全局作用域（和其他范围里）设置变量的详细信息，请参阅 3.3 节。更多有关 dot-sourcing 创建库的更多信息，请参阅 10.5 节。

参考

- 3.3 节
 - 10.5 节

Windows PowerShell

金面 0.2

新发现的物种，其中 *Phrynosoma modestum* 的首次使用，是“新物种”，即没有对物种，物种尚未“高置入分类，等到尚未被识别时，它们就可能被归类于未被识别的物种”。

¹⁰ 例如，高爾基的《童年》，由最初編號[1]是被認為太低了，是因為

聯發 IMW 向古

卷之三

精英读物·新编世界文学名著典藏·第二辑

卷之三

（原刊于1981年3月《中国青年报》，略有删节，原文标题为“中国青年报采访记”）

第 15 章

Windows PowerShell 的扩展

15.0 简介

PowerShell 的操作环境是全面的。它提供了一些功能强大的命令来帮助你管理你的系统；它提供了一种很好的脚本语言，让你自动化这些任务，直接访问你已经知道的所有实用程序和工具。

命令、脚本语言和已经存在的工具仅仅是 PowerShell 中的一部分。除了这些功能，PowerShell 提供的其他大量的技术，也大大提高了它的性能，这些技术包括：.Net 框架、WMI、COM 自动化和 API 调用等。

对于使用这些技术，PowerShell 提供了统一的访问方式。你可以像使用 PowerShell 对象的属性和方法那样来使用 .NET 对象的属性和方法。使用 WMI 和 COM 对象可以采用同样的方式。

使用这些技术的另一个巨大的好处是可以轻松的使用 .NET 的编程语言，如 C#。

15.1 访问 WMI 数据

问题

如何使用 Windows 中 WMI 工具提供的功能和数据。

解决方案

若要检索 WMI 类的所有实例，可以使用 Get-WmiObject 命令：

```
Get-WmiObject -ComputerName Computer -Class Win32_Bios
```

要检索一个 WMI 类的特定实例，可以使用 WMI 筛选器，提供 -Filter 参数给 Get-WmiObject 命令：

```
Get-WmiObject Win32_Service -Filter "StartMode = 'Auto'"
```

若要使用 WMI 的 WQL 语言检索 WMI 类的实例，可以使用 [WmiSearcher] 类型的快捷方式：

```
$query = [WmiSearcher] "SELECT * FROM Win32_Service WHERE StartMode = 'Auto'"  
$query.Get()
```

若要使用 WMI 筛选器检索一个 WMI 类的特定实例，可以使用 [Wmi] 类型的快捷方式：

```
[Wmi] 'Win32_Service.Name="winmgmt"'
```

若要检索一个 WMI 实例的属性，你可以像访问 .NET 属性那样访问它：

```
$service = [Wmi] 'Win32_Service.Name="winmgmt"'  
$service.StartMode
```

若要调用 WMI 实例的一个方法，你可以像调用一个 .NET 方法那样调用它：

```
$service = [Wmi] 'Win32_Service.Name="winmgmt"'  
$service.ChangeStartMode("Manual")  
$service.ChangeStartMode("Automatic")
```

若要调用 WMI 类上的一个方法，你可以使用 [WmiClass] 类型的快捷方式来访问该 WMI 类。然后，你可以像调用一个 .NET 方法那样调用它：

```
$class = [WmiClass] "Win32_Process"  
$class.Create("Notepad")
```

讨论

一直以来，人们使用 WMI 进行 Windows 系统管理，尤其是管理公司或企业的系统。WMI 支持大量的 Windows 管理任务，但是没有提供友好的访问方式。

传统上，管理员需要使用 VBScript 或 WMIC 命令行工具通过 WMI 来访问和管理这些系统。尽管它们功能强大且有用，但这些方法仍然存在很多需要改进的地方。VBScript 不支持特殊的调查方法，而 WMIC 未能提供（或利用）适用于 WMIC 外部的一些知识。

与它们相比，PowerShell 可以让你就像你使用 shell 程序的其余部分一样来使用 WMI。你可以像使用 PowerShell 中的其他对象的方法和属性那样来使用 WMI 实例提供的方法和属性。

一旦你掌握了这个方法，你会发现使用 WMI 实例和类会使工作变得更轻松，但同时，

PowerShell 还提供了另一种在开始位置访问它们的全新方法。对于大多数的任务来说，只需要使用解决方案中所示的简单的 [Wmi]、[WmiClass] 或 [WmiSearcher] 语法就可以完成。

虽然 WMI 提供了丰富的功能，但是随之而来的问题是：如何快速查找完成你的任务的 WMI 类。如果你要学习可用的 WMI 类，可以参见附录 F，那里提供了最常见的 WMI 类的列表。你也可以使用 15.3 节中提供的脚本，按名称、描述、属性名或属性说明来搜索 WMI 类。

某些高级的 WMI 任务要求你启用安全权限或调整你的请求中使用数据包的隐私设置。解决方案提供的语法不直接支持这些任务，但是 PowerShell 通过提供对表示你的 WMI 查询的基础对象的访问支持来这些选项。有关使用这些基础对象的详细信息，请参阅 15.4 节。

当你使用 [Wmi] 快捷方式访问特定的 WMI 实例时，你可能首先要确定哪些属性是 WMI 可以搜索的。这些属性称为类的关键属性。你可以参阅下面的 15.2 节中的脚本来列出关键属性。

有关 Get-WmiObject 命令的更多信息，可以键入 **Get-help Get-WmiObject**。

参考

- 15.2 节
- 15.3 节
- 15.4 节
- 附录 F “WMI 参考”

15.2 编程：确定可用到 WMI 筛选器的属性

当你要使用 PowerShell 的 [Wmi] 快捷方式访问特定的 WMI 实例的时候，你可能首先要确定 WMI 允许你搜索哪些属性。这些属性称为类的关键属性。例 15-1 获取了给定的类在 WMI 筛选器中可以使用的所有属性。

例 15-1：Get-WmiClassKeyProperty.ps1

```
#####
##  
## Get-WmiClassKeyProperty.ps1  
##
```

```
## Get all of the properties that you may use in a WMI filter for a given
class.
##
## ie:
##
## PS >Get-WmiClassKeyProperty Win32_Process
##
#####
param( [WmiClass] $wmiClass )

## WMI classes have properties
foreach($currentProperty in $wmiClass.PsBase.Properties)
{
    ## WMI properties have qualifiers to explain more about them
    foreach($qualifier in $currentProperty.Qualifiers)
    {
        ## If it has a 'Key' qualifier, then you may use it in a filter
        if($qualifier.Name -eq "Key")
        {
            $currentProperty.Name
        }
    }
}
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

15.3 编程：搜索 WMI 类

问题

虽然 WMI 提供了丰富的功能，但是随之而来的问题是：如何快速查找完成你的任务的 WMI 类。如果你要学习可用的 WMI 类，可以参见附录 F，那里提供了最常见的 WMI 类的列表。如果你想更深入的了解 WMI 类，例 15-2 中的脚本可以让你按 WMI 类的名称、说明、属性名或属性说明来搜索 WMI 类。

例 15-2：Search-WmiNamespace.ps1

```
#####
##
## Search-WmiNamespace.ps1
##
## Search the WMI classes installed on the system for the provided match text.
##
## ie:
```

```

## PS >Search-WmiNamespace Registry
## PS >Search-WmiNamespace Process ClassName,PropertyName
## PS >Search-WmiNamespace CPU -Detailed
##
#####
param(
    [string] $pattern = $(throw "Please specify a search pattern."),
    [switch] $detailed,
    [switch] $full,
    ## Supports any or all of the following match options:
    ## ClassName, ClassDescription, PropertyName, PropertyDescription
    [string[]] $matchOptions = ("ClassName","ClassDescription")
)

## Helper function to create a new object that represents
## a Wmi match from this script
function New-WmiMatch
{
    param( $matchType, $className, $propertyName, $line )

    $wmiMatch = New-Object PsObject
    $wmiMatch | Add-Member NoteProperty MatchType $matchType
    $wmiMatch | Add-Member NoteProperty ClassName $className
    $wmiMatch | Add-Member NoteProperty PropertyName $propertyName
    $wmiMatch | Add-Member NoteProperty Line $line

    $wmiMatch
}

## If they've specified the -detailed or -full options, update
## the match options to provide them an appropriate amount of detail
if($detailed)
{
    $matchOptions = "ClassName", "ClassDescription", "PropertyName"
}

if($full)
{
    $matchOptions =
    "ClassName", "ClassDescription", "PropertyName", "PropertyDescription"
}

## Verify that they specified only valid match options
foreach($matchOption in $matchOptions)
{
    $fullMatchOptions =
    "ClassName", "ClassDescription", "PropertyName", "PropertyDescription"

    if($fullMatchOptions -notcontains $matchOption)
    {
        $error = "Cannot convert value {0} to a match option. " +
                 "Specify one of the following values and try again.
                 "The possible values are ""{1}""."
    }
}

```

```
$ofs = ", " + $matchOptions
throw ($error -f $matchOption, ([string] $fullMatchOptions))
}

## Go through all of the available classes on the computer
foreach($class in Get-WmiObject -List)
{
    ## Provide explicit get options, so that we get back descriptions
    ## as well
    $managementOptions = New-Object System.Management.ObjectGetOptions
    $managementOptions.UseAmendedQualifiers = $true
    $managementClass =
        New-Object Management.ManagementClass $class.Name,$managementOptions

    ## If they want us to match on class names, check if their text
    ## matches the class name
    if($matchOptions -contains "ClassName")
    {
        if($managementClass.Name -match $pattern)
        {
            New-WmiMatch "ClassName"
            $managementClass.Name $null $managementClass.__PATH
        }
    }

    ## If they want us to match on class descriptions, check if their text
    ## matches the class description
    if($matchOptions -contains "ClassDescription")
    {
        $description =
            $managementClass.PsBase.Qualifiers |
                foreach { if($_.Name -eq "Description") { $_.Value } }
        if($description -match $pattern)
        {
            New-WmiMatch "ClassDescription"
            $managementClass.Name $null $description
        }
    }

    ## Go through the properties of the class
    foreach($property in $managementClass.PsBase.Properties)
    {
        ## If they want us to match on property names, check if their text
        ## matches the property name
        if($matchOptions -contains "PropertyName")
        {
            if($property.Name -match $pattern)
            {
                New-WmiMatch "PropertyName"
                $managementClass.Name $property.Name $property.Name
            }
        }
    }
}
```

```
## If they want us to match on property descriptions, check if
## their text matches the property name
if($matchOptions -contains "PropertyDescription")
{
    $propertyDescription =
        $property.Qualifiers |
            foreach { if($_.Name -eq "Description") { $_.Value } }
    if($propertyDescription -match $pattern)
    {
        New-WmiMatch "PropertyDescription" `
            $managementClass.Name $property.Name $propertyDescription
    }
}
}
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节
 - 附录 F “WMI 参考”

15.4 使用 .NET 来执行高级的 WMI 任务

问题

你要使用 WMI 的高级功能，但 PowerShell 对它们的访问不能直接支持（通过 [Wmi]、[WmiClass] 和 [WmiSearcher] 加速器）。

解决方案

若要通过 .NET 接口与它们的高级功能进行交互，可以使用所生成的对象的 `PsBase` 属性。

高级的实例功能

若要获取与给定的实例相关的 WMI 实例，可以调用 GetRelated()方法：

```
$instance = [Wmi] 'Win32_Service.Name="winmgmt"'  
$instance.PsBase.GetRelated()
```

若要启用命令（例如更改系统时间）的安全权限，要将 `EnablePrivileges` 属性设置为 `$true`：

```
$system = Get-WmiObject Win32_OperatingSystem
$system.PsBase.Scope.Options.EnablePrivileges = $true
$system.SetDateTime($class.ConvertFromDateTime("01/01/2007"))
```

高级的类功能

要检索 WMI 属性和类的限定符，可以访问 `PsBase.Properties` 属性：

```
$class = [WmiClass] "Win32_Service"
$class.PsBase.Properties
```

高级的查询功能

要配置连接选项，如数据包隐私和身份验证，需要设置 `Scope` 属性上的选项：

```
$credential = Get-Credential
$query = [WmiSearcher] "SELECT * FROM IISWebServerSetting"
$query.Scope.Path = "\\\$REMOTE_COMPUTER\Root\MicrosoftIISV2"
$query.Scope.Options.Username = $credential.Username
$query.Scope.Options.Password = $credential.GetNetworkCredential().Password
$query.Scope.Options.Authentication = "PacketPrivacy"
$query.get() | Select-Object AnonymousUserName
```

讨论

[Wmi]、[WmiClass] 和 [WmiSearcher] 类型的快捷方式分别返回 .NET `System.Management.ManagementObject`、`System.Management.ManagementClass` 和 `System.Management.ManagementObjectSearcher` 类的实例。

正如期待的那样，.NET Framework 全面的支持 WMI 查询，而 PowerShell 提供了更易于使用的支持 WMI 查询的接口。如果需要对 PowerShell 直接提供的功能以外的支持，可以考虑使用 .NET Framework 中的这些类提供的高级的接口。

有关使用 .NET Framework 中的类的详细信息，请参阅 3.4 节。

参考

- 3.4 节

15.5 将一个VBScript WMI 脚本转换为 PowerShell

问题

你想在 PowerShell 中执行一个 WMI 任务，但只能找到解决该任务的 VBScript 示例。

解决方案

若要完成从计算机中检索数据的脚本任务，可以使用 `Get-WmiObject` 命令：

```
foreach($printer in Get-WmiObject -Computer COMPUTER Win32_Printer)
{
    ## Work with the properties
    $printer.Name
}
```

若要完成调用实例方法的脚本任务，可以使用 `[Wmi]` 或 `[WmiSearcher]` 快捷方式来检索实例，然后像调用任何其他 PowerShell 方法一样调用该实例上的方法：

```
$service = [Wmi] 'Win32_Service.Name="winmgmt"'
$service.ChangeStartMode("Manual")
$service.ChangeStartMode("Automatic")
```

若要完成调用类方法的脚本任务，可以使用 `[WmiClass]` 快捷方式来检索类，然后像调用任何其他 PowerShell 方法一样调用该类上的方法：

```
$class = [WmiClass] "Win32_Process"
$class.Create("Notepad")
```

讨论

多年以来，VBScript 是管理员用来访问 WMI 数据的首选的语言。因此，可以在书籍或 Internet 上找到大多数有关的脚本。

这些脚本通常采取以下三种形式之一：检索数据和访问属性、调用实例的方法以及调用类的方法。

注意： 大多数来自 Internet 的 WMI 脚本完成唯一的任务，PowerShell 本身支持许多传统的 WMI 任务。如果你打算把一个 WMI 脚本示例转换成 PowerShell，首先检查是否有任何 PowerShell 命令可以直接完成该任务。

检索数据

WMI 的最常见用途之一是数据收集和系统详细目录的任务。一个典型的检索数据的 VBScript 看起来类似于例 15-3。

例 15-3：使用 VBScript 从 WMI 中检索打印机信息

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" &
    & "{impersonationLevel=impersonate}!\" & strComputer & "\root\cimv2")

Set colInstalledPrinters = objWMIService.ExecQuery(_
    ("Select * from Win32_Printer"))

For Each objPrinter in colInstalledPrinters
    Wscript.Echo "Name: " & objPrinter.Name
    Wscript.Echo "Location: " & objPrinter.Location
    Wscript.Echo "Default: " & objPrinter.Default
Next
```

前三行代码准备与一个给定的计算机和命名空间建立 WMI 连接。接下来的两行代码准备一个 WMI 查询，该查询请求一个类的所有实例。然后 For Each 循环遍历所有实例，并显示这些实例上的属性。

在 PowerShell 中，Get-WmiObject 命令负责从计算机和你指定的命名空间检索类的所有实例。这样的话，上面 VBScript 脚本中的前五个行代码可以用下面的语句实现：

```
$installedPrinters = Get-WmiObject Win32_Printer
```

如果你要指定不同的计算机、命名空间或查询的限制，GetWmiObject 命令通过可选参数来支持这些选项。如果你需要指定高级的连接选项（如身份验证级别），请参阅 15.4 节。

在 PowerShell 中，For Each 循环语句如下：

```
foreach($printer in $installedPrinters)
{
    $printer.Name
    $printer.Location
    $printer.Default
}
```

注意我们演示了如何把 VBScript 脚本转换为 PowerShell 语句来访问属性。如果你实际上不处理属性（只想显示这些作为报表），PowerShell 的格式命令可进一步简化为：

```
Get-WmiObject Win32_Printer | Format-List Name,Location,Default
```

有关使用 Get-WmiObject 命令的详细信息，请参阅 15.1 节。

调用一个实例的方法

尽管数据检索的脚本形成了大量的 WMI 管理示例，但另一个常见的任务是调用实例的方法来执行操作。例如，例 15-4 更改服务的启动类型。

例 15-4：使用 VBScript 更改服务的启动类型

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:" _ 
    & "{impersonationLevel=impersonate}!\" & strComputer & "\root\cimv2")

Set colServiceList = objWMIService.ExecQuery _ 
    ("Select * from Win32_Service where StartMode = 'Manual'")

For Each objService in colServiceList
    errReturnCode = objService.ChangeStartMode("Disabled")
Next
```

前三行代码准备与一个给定的计算机和命名空间建立 WMI 连接。接下来的两行代码准备一个 WMI 事件查询，请求一个类的所有实例并把一个额外的筛选器 (StartMode = 'Manual') 添加到查询中。For Each 语句块循环了所有服务的实例，并调用 objService 的 ChangeStartMode() 方法。

在 PowerShell 中，Get-WmiObject 命令负责从计算机和你指定的命名空间中检索类的所有实例。这样的话，上面 VBScript 脚本中的前五行代码可以用下面的语句实现：

```
$services = Get-WmiObject Win32_Service -Filter "StartMode = 'Manual'"
```

如果你要指定不同的计算机、命名空间或查询的限制，GetWmiObject 命令通过可选参数来支持这些选项。如果你需要指定高级的连接选项（如身份验证级别），请参阅 15.4 节。

在 PowerShell 中，For Each 语句块变为：

```
foreach($service in $services)
{
    $service.ChangeStartMode("Disabled")
}
```

有关使用 Get-WmiObject 命令的详细信息，请参阅 15.1 节。

调用类的方法

尽管很少使用，但有时调用一个 WMI 类的方法也是非常有帮助的。PowerShell 使此项工作几乎和调用实例的方法完全一样。例如，在远程计算机创建一个进程的脚本如下所示：

```
strComputer = "COMPUTER"
Set objWMIService = GetObject _
    ("winmgmts:\\" & strComputer & "\root\cimv2:Win32_Process")
objWMIService.Create("notepad.exe")
```

前三行代码准备与某个给定的计算机和命名空间建立 WMI 连接，最后一行调用类的 Create() 方法。

在 PowerShell 中，[WmiClass] 快捷方式可让你轻松地访问 WMI 类。这样前三行代码可以写成下面的样子：

```
$processClass = [WmiClass] "\\\$COMPUTER\Root\Cimv2:Win32_Process"
```

如果你需要指定高级的连接选项（如身份验证级别），请参阅 15.4 节。

在 PowerShell 中，调用类方法与实例调用类的方法几乎是相同的：

```
$processClass.Create("notepad.exe")
```

有关使用 [WmiClass] 快捷方式的详细信息，请参阅 15.1 节。

参考

- 15.1 节
- 15.4 节

15.6 使用 COM 脚本接口自动化程序

问题

你希望通过 COM 自动化接口来自动运行程序或执行系统任务。

解决方案

若要实例化并使用 COM 对象，可以使用 New-Object 命令的 -ComObject 参数。

```
$shell = New-Object -ComObject "Shell.Application"
$shell.Windows() | Format-Table LocationName,LocationUrl
```

讨论

与 WMI 相似，COM 自动化接口是一个标准的工具，可用于脚本编写和系统管理。当

应用程序公开管理或使任务自动化时，COM 对象是第二个最常见的接口（自定义命令行工具）。

PowerShell 会公开 COM 对象，如同它在系统中公开的大多数其他管理对象一样。一旦你创建了一个 COM 对象后，你就可以使用其属性和方法，就像使用 PowerShell 中的对象的方法和属性一样。

除了自动化任务，许多 COM 对象的存在完全是为了改善语言的使用体验，如 VBScript。一个示例是使用文件，或对数组进行排序。

在使用这些 COM 对象的时候，要注意的是 PowerShell 通常提供了更好的备选方案。很多情况下，PowerShell 中的命令、脚本语言或 .NET Framework 提供了与你可能使用的 COM 对象相同或相似的功能。

有关使用 COM 对象的详细信息，请参阅 3.8 节。有关最有用的 COM 对象的列表，请参阅附录 E 中“选中的 .NET 类和它们的使用”。

参考

- 3.8 节
- 附录 E 中“选中的 .NET 类和它们的使用”

15.7 编程：查询 SQL 数据源

针对如 SQL 服务器、Access 数据库或甚至 Excel 电子表格等数据源执行特殊查询和命令通常很有帮助的。尤其是当你从一个系统中取出数据并将其放在另一个系统中，或者想要将数据放到交互操作中处理的时候。

尽管在 PowerShell 中可以直接访问这些数据源（通过 .NET Framework 的支持），但每个数据源仍需要一个唯一的和比较难记的语法。

例 15-5 将使用这些基于 SQL 的数据源的工作变得一致同时又功能强大。

例 15-5：Invoke-SqlCommand.ps1

```
#####
## Invoke-SqlCommand.ps1
## Return the results of a SQL query or operation
## ie: $x = Invoke-SqlCommand -Query "SELECT * FROM sys.tables"
```

```
##  
##    ## Use Windows authentication  
##    Invoke-SqlCommand.ps1 -Sql "SELECT TOP 10 * FROM Orders"  
##  
##    ## Use SQL Authentication  
##    $cred = Get-Credential  
##    Invoke-SqlCommand.ps1 -Sql "SELECT TOP 10 * FROM Orders" -Cred $cred  
##  
##    ## Perform an update  
##    $server = "MYSERVER"  
##    $database = "Master"  
##    $sql = "UPDATE Orders SET EmployeeID = 6 WHERE OrderID = 10248"  
##    Invoke-SqlCommand $server $database $sql  
##  
##    $sql = "EXEC SalesByCategory 'Beverages'"  
##    Invoke-SqlCommand -Sql $sql  
##  
##    ## Access an access database  
##    Invoke-SqlCommand (Resolve-Path access_test.mdb) -Sql "SELECT * from Users"  
##  
##    ## Access an excel file  
##    Invoke-SqlCommand (Resolve-Path xls_test.xls) -Sql 'SELECT * from [Sheet1]'  
##  
#####  
  
param(  
    [string] $dataSource = ".\SQLEXPRESS",  
    [string] $database = "Northwind",  
    [string] $sqlCommand = $(throw "Please specify a query."),  
    [System.Management.Automation.PsCredential] $credential  
)  
  
## Prepare the authentication information. By default, we pick  
## Windows authentication  
$authentication = "Integrated Security=SSPI;"  
  
## If the user supplies a credential, then they want SQL  
## authentication  
if($credential)  
{  
    $plainCred = $credential.GetNetworkCredential()  
    $authentication =  
        ("uid={0};pwd={1};"-f $plainCred.Username,$plainCred.Password)  
}  
  
## Prepare the connection string out of the information they  
## provide  
$connectionString = "Provider=sqloledb; " +  
                    "Data Source=$dataSource; " +  
                    "Initial Catalog=$database; " +  
                    "$authentication; "  
  
## If they specify an Access database or Excel file as the connection  
## source, modify the connection string to connect to that data source  
if($dataSource -match '\.xls\$|\.mdb\$')
```

```
{  
    $connectionString = "Provider=Microsoft.Jet.OLEDB.4.0; Data Source=$dataSource;  
  
    if($dataSource -match '\.xls$')  
    {  
        $connectionString += 'Extended Properties="Excel 8.0;"';  
  
        ## Generate an error if they didn't specify the sheet name properly  
        if($sqlCommand -notmatch '\[.+\$\\\]')  
        {  
            $error = 'Sheet names should be surrounded by square brackets, and ' +  
                    'have a dollar sign at the end: [Sheet1$]'  
            Write-Error $error  
            return  
        }  
    }  
}  
  
## Connect to the data source and open it  
$connection = New-Object System.Data.OleDb.OleDbConnection $connectionString  
$command = New-Object System.Data.OleDb.OleDbCommand $sqlCommand,$connection  
$connection.Open()  
  
## Fetch the results, and close the connection  
$adapter = New-Object System.Data.OleDb.OleDbDataAdapter $command  
$dataset = New-Object System.Data.DataSet  
[void] $adapter.Fill($dataset)  
$connection.Close()  
  
## Return all of the rows from their query  
$dataset.Tables | Select-Object -Expand Rows
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

15.8 访问 Windows 性能计数器

问题

你希望从 PowerShell 中访问系统性能计数器的信息。

解决方案

若要检索有关一个特定的性能计数器的信息，可以使用 .NET Framework 中的 System.Diagnostics.PerformanceCounter 类，如例 15-6 所示：

例 15-6：通过 System.Diagnostics 访问性能计数器数据

```
PS >$arguments = "System", "System Up Time"
PS >$counter = New-Object System.Diagnostics.PerformanceCounter $arguments
PS >
PS >[void] $counter.NextValue()
PS >New-Object TimeSpan 0,0,0,$counter.NextValue()
```

```
Days          : 0
Hours         : 18
Minutes       : 51
Seconds       : 17
Milliseconds  : 0
Ticks          : 678770000000
TotalDays     : 0.785613425925926
TotalHours    : 18.8547222222222
TotalMinutes  : 1131.28333333333
TotalSeconds  : 67877
TotalMilliseconds : 67877000
```

或者，使用 WMI 的 Win32_Perf * 设置这些类支持的许多最常见的性能计数器：

```
Get-WmiObject Win32_PerfFormattedData_Tcpip_NetworkInterface
```

讨论

.NET Framework 中的 System.Diagnostics.PerformanceCounter 类提供了对不同的性能计数器的访问。例 15-6 说明了如何从一个具体类别上使用性能计数器。另外，PerformanceCounter 类的构造器可以允许在访问性能计数器后指定实例名字，甚至是机器的名字。

当你第一次访问一个性能计数器的时候，NextValue()方法返回的是 0。从那时起，系统开始抽样采集性能信息，当下一次你调用NextValue()方法的时候就会返回一个当前值。

有关使用 .NET Framework 中的类的详细信息，请参阅 3.4 节。

参考

- 3.4 节

15.9 编程：调用 Windows 系统 API

有些时候，PowerShell 提供的命令和脚本都无法直接支持某一项功能的需要。在大多数这种情况下，PowerShell 支持使用 .NET Framework 提供的功能帮助你完成任务。但

是，在某些情况下，甚至是 .NET Framework 也不能解决问题，这时候，解决问题的唯一的方法就是访问核心的 Windows API。

对于复杂的 API 调用（那些需要高度结构化的数据），解决方案是使用 .NET Framework 中支持的 P/Invoke（Platform Invoke）编写一个 PowerShell 命令。.NET Framework 中的 P/Invoke 允许你直接访问核心 Windows API。

尽管你可以自己通过查询获得这些 P/Invoke 的定义，但是最简单的方法是基于其他人的工作结果之上获得。如果想了解在 .NET 语言中如何调用一个 Windows API，那么你可以从网站 <http://pinvoke.net> 上获得许多相关的信息。

如果你要访问的 API 是简单的（调用和返回的是简单的数据类型），例 15-7 允许你直接从 PowerShell 调用这些 Windows API。有关此脚本的操作的示例，请参见 17.18 节。

例 15-7：Invoke-WindowsApi.ps1

```
#####
## Invoke-WindowsApi.ps1
##
## Invoke a native Windows API call that takes and returns simple data types.
##
## ie:
##
## ## Prepare the parameter types and parameters for the
## CreateHardLink function
## $parameterTypes = [string], [string], [IntPtr]
## $parameters = [string] $filename, [string] $existingFilename, [IntPtr]::Zero
##
## ## Call the CreateHardLink method in the Kernel32 DLL
## $result = Invoke-WindowsApi "kernel32" ([bool]) "CreateHardLink" `

## $parameterTypes $parameters
##
#####
param(
    [string] $dllName,
    [Type] $returnType,
    [string] $methodName,
    [Type[]] $parameterTypes,
    [Object[]} $parameters
)

## Begin to build the dynamic assembly
$domain = [AppDomain]::CurrentDomain
$name = New-Object Reflection.AssemblyName 'PInvokeAssembly'
$assembly = $domain.DefineDynamicAssembly($name, 'Run')
$module = $assembly.DefineDynamicModule('PInvokeModule')
$type = $module.DefineType('PInvokeType', 'Public,BeforeFieldInit')
```

```
## Go through all of the parameters passed to us. As we do this,
## we clone the user's inputs into another array that we will use for
## the P/Invoke call.
$inputParameters = @()
$refParameters = @()

for($counter = 1; $counter -le $parameterTypes.Length; $counter++)
{
    ## If an item is a PSReference, then the user
    ## wants an [out] parameter.
    if($parameterTypes[$counter - 1] -eq [Ref])
    {
        ## Remember which parameters are used for [Out] parameters
        $refParameters += $counter

        ## On the cloned array, we replace the PSReference type with the
        ## .Net reference type that represents the value of the PSReference,
        ## and the value with the value held by the PSReference.
        $parameterTypes[$counter - 1] =
            $parameters[$counter - 1].Value.GetType().MakeByRefType()
        $inputParameters += $parameters[$counter - 1].Value
    }
    else
    {
        ## Otherwise, just add their actual parameter to the
        ## input array.
        $inputParameters += $parameters[$counter - 1]
    }
}

## Define the actual P/Invoke method, adding the [Out]
## attribute for any parameters that were originally [Ref]
## parameters.
$method = $type.DefineMethod($methodName,
'Public,HideBySig,Static,PinvokeImpl',
$returnType, $parameterTypes)
foreach($refParameter in $refParameters)
{
    [void] $method.DefineParameter($refParameter, "Out", $null)
}

## Apply the P/Invoke constructor
$ctor = [Runtime.InteropServices.DllImportAttribute].GetConstructor([string])
$attr = New-Object Reflection.Emit.CustomAttributeBuilder $ctor, $dllName
$method.SetCustomAttribute($attr)

## Create the temporary type, and invoke the method.
$realType = $type.CreateType()

$realType.InvokeMember($methodName, 'Public,Static,InvokeMethod', $null, $null,
$inputParameters)

## Finally, go through all of the reference parameters, and update the
## values of the PSReference objects that the user passed in.
foreach($refParameter in $refParameters)
{
```

```
$parameters[$refParameter - 1].Value = $inputParameters[$refParameter - 1]  
}
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节
- 17.18 节

15.10 编程：添加 C# 代码到 PowerShell 脚本中

在你学习完PowerShell之后，应该掌握的一门语言就是C#。它采用了许多和PowerShell相同的编程技术，同样使用.NET框架提供的类。此外，C#在某些方面还提供了更多的语言特性和性能方面的提升，这些是PowerShell无法提供的。

在这种情况下，你不用完全切换到C#，例15-8可以让你在脚本中直接编写与调用C#。

例 15-8：Invoke-Inline.ps1

```
#####
## Invoke-Inline.ps1
## Library support for inline C#
##
## Usage
## 1) Define just the body of a C# method, and store it in a string. "Here
## strings" work great for this. The code can be simple:
##
## $codeToRun = "Console.WriteLine(Math.Sqrt(337));"
##
## or more complex:
##
## $codeToRun = @"
##     string firstArg = (string) ((System.Collections.ArrayList) arg)[0];
##     int secondArg = (int) ((System.Collections.ArrayList) arg)[1];
##     Console.WriteLine("Hello {0} {1}", firstArg, secondArg );
##
##     returnValue = secondArg * 3;
## "@
##
## 2) (Optionally) Pack any arguments to your function into a single object.
## This single object should be strongly-typed, so that PowerShell does
## not treat it as a PsObject.
## An ArrayList works great for multiple elements. If you have only one
## argument, you can pass it directly.
##
```

```
##      [System.Collections.ArrayList] $arguments =
##      New-Object System.Collections.ArrayList
##      [void] $arguments.Add("World")
##      [void] $arguments.Add(337)
##
## 3) Invoke the inline code, optionally retrieving the return value. You can
##    set the return value in your inline code by assigning it to the
##    "returnValue" variable as shown above.
##
##      $result = Invoke-Inline $codeToRun $arguments
##
##
## If your code is simple enough, you can even do this entirely inline:
##
##      Invoke-Inline "Console.WriteLine(Math.Pow(337,2));"
##
#####
##### param(
#####     [string] $code = $(throw "Please specify the code to invoke"),
#####     [object] $arg,
#####     [string[]] $reference = @()
##### )
#####
##### Stores a cache of generated inline objects. If this library is dot-sourced
##### from a script, these objects go away when the script exits.
if(-not (Test-Path Variable:\Lee.Holmes.inlineCache))
{
    ${GLOBAL:Lee.Holmes.inlineCache} = @{}
}

## The main function to execute inline C#.
## Pass the argument to the function as a strongly-typed variable. They will
## be available from C# code as the Object variable, "arg".
## Any values assigned to the "returnValue" object by the C# code will be
## returned to the caller as a return value.

function main
{
    ## See if the code has already been compiled and cached
    $cachedObject = ${Lee.Holmes.inlineCache}{$code}

    ## The code has not been compiled or cached
    if($cachedObject -eq $null)
    {
        $codeToCompile =
@"
using System;

public class InlineRunner
{
    public Object Invoke(Object arg)
    {
        Object returnValue = null;
        $code
    }
}
```

```
        return returnValue;
    }
}

## Obtains an ICodeCompiler from a CodeDomProvider class.
$provider = New-Object Microsoft.CSharp.CSharpCodeProvider

## Get the location for System.Management.Automation DLL
$dllName = [PsObject].Assembly.Location

## Configure the compiler parameters
$compilerParameters = New-Object System.CodeDom.Compiler.CompilerParameters

$assemblies = @("System.dll", $dllName)
$compilerParameters.ReferencedAssemblies.AddRange($assemblies)
$compilerParameters.ReferencedAssemblies.AddRange($reference)
$compilerParameters.IncludeDebugInformation = $true
$compilerParameters.GenerateInMemory = $true

## Invokes compilation.
$compilerResults =
    $provider.CompileAssemblyFromSource($compilerParameters, $codeToCompile)

## Write any errors if generated.
if($compilerResults.Errors.Count -gt 0)
{
    $errorLines = "`n$codeToCompile"
    foreach($error in $compilerResults.Errors)
    {
        $errorLines += "`n`t" + $error.Line + ":"`t" + $error.ErrorText
    }
    Write-Error $errorLines
}
## There were no errors. Store the resulting object in the object "cache".
else
{
    ${Lee.Holmes.inlineCache}[$code] =
        $compilerResults.CompiledAssembly.CreateInstance("InlineRunner")
}

$cachedObject = ${Lee.Holmes.inlineCache}[$code]
}

## Finally invoke the C# code
if($cachedObject -ne $null)
{
    return $cachedObject.Invoke($arg)
}
}

.Main
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

15.11 访问 .NET SDK 库

问题

你想访问一个被 .NET DLL 公开的功能，但是该 DLL 已经打包到一个面向开发人员的软件开发工具包（SDK）中。

解决方案

要创建一个包含在 DLL 中的对象，可以使用 [System.Reflection.Assembly]::LoadFile()方法来加载 DLL 到内存中，然后使用 New-Object 命令来创建该对象。例 15-9 阐释了这种技术。

例 15-9：与 SharpZipLib SDK 中的 DLL 进行交互

```
[Reflection.Assembly]::LoadFile("d:\bin\ICSharpCode.SharpZipLib.dll")
$namespace = "ICSharpCode.SharpZipLib.Zip.{0}"

$zipName = Join-Path (Get-Location) "PowerShell_TDG_Scripts.zip"
$zipFile = New-Object ($namespace -f "ZipOutputStream") ([IO.File]::Create($zipName))

foreach($file in dir *.ps1)
{
    $zipEntry = New-Object ($namespace -f "ZipEntry") $file.Name
    $zipFile.PutNextEntry($zipEntry)
}

$zipFile.Close()
```

讨论

C# 和 VB.NET 开发人员通常是 .NET Framework SDK 的主要使用者，PowerShell 允许你轻松地访问 SDK 功能。为了访问 SDK，首先使用 [Reflection.Assembly]::LoadFile() 方法加载 SDK 的程序集，然后使用该程序集中的类。

注意：尽管 PowerShell 允许你轻松的访问面向开发人员的 SDK，但这个 SDK 是面向开发人员的，而不是面向管理员的。SDK 和编程接口在设计的时候很少会考虑管理员的使用，所以要准备好数个使用这些编程模型，可能需要多个步骤来完成你的任务。

在使用 SDK 中的类的时候，需要注意的是那些令人头痛的类的名称。例如，在 SharpZipLibSDK 中与 zip 相关的类都是以 ICSharpCode.SharpZipLib.Zip 开头的。这称为该类的命名空间 (*namespace*)。大多数编程语言使用 using 语句来解决此问题。当你输入一个类的名称如 ZipEntry 的时候，系统会自动提示你该类下的命名空间，供你检索。PowerShell 中缺少 using 语句，本解决方案演示了几个方法来实现这个功能。

有关如何管理这些较长类名的更多信息，请参阅 3.7 节。

重新打包的 SDK 也不是仅有的加载这种方法的 DLL。一个 SDK 库文件是由某些人编写的，经过编译、打包最后发布的 DLL 文件。如果你对 .NET 语言很熟悉，你也可以编写自己的 DLL、编译它，并采用同样的方式使用。

以例 15-10 给出的简单数学库为例。它提供了一个静态求和方法和实例化乘法方法。

例 15-10：一个简单的 C# 数学库

```
namespace MyMathLib
{
    public class Methods
    {
        public Methods()
        {
        }

        public static int Sum(int a, int b)
        {
            return a + b;
        }

        public int Product(int a, int b)
        {
            return a * b;
        }
    }
}
```

例 15-11 演示了在你的 PowerShell 系统中为了使上述代码正常工作需要做的事。

例 15-11：编译、加载和使用一个简单的 C# 库

```
PS >notepad MyMathLib.cs
<add the above code to MyMathLib.cs>

PS >Set-Alias csc $env:WINDIR\Microsoft.NET\Framework\v2.0.50727\csc.exe
PS >csc /target:library MyMathLib.cs

Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.
```

```
PS >[Reflection.Assembly]::LoadFile("c:\temp\MyMathLib.dll")
GAC      Version      Location
---      -----
False    v2.0.50727  c:\temp\MyMathLib.dll

PS >[MyMathLib.Methods]::Sum(10, 2)
12
PS >$mathInstance = New-Object MyMathLib.Methods
PS >$mathInstance.Product(10, 2)
20

有关使用 .NET Framework 中的类的更多信息，请参阅 3.5 节。
```

参考

- 3.5 节
- 3.7 节

15.12 创建你自己的 PowerShell Cmdlet

问题

如何编写你自己的 PowerShell cmdlet。

讨论

正如在 Windows PowerShell 指导教程中提到的，“结构化的命令 (cmdlets)”，PowerShell 命令提供了一些比传统的可执行程序更重要的优点。从用户的角度来看，cmdlet 支持输入强类型对象，这使其功能强大有着令人难以置信的一致。从 cmdlet 作者的角度来看，与他们提供的强大的功能相比，cmdlet 非常容易编写。创建和公开一个新的命令行参数与在某个类上创建一个新的公用属性一样容易。支持一个丰富的管道模型，可以将你的实现逻辑放到三个标准方法之一中。

有关如何实现一个完整 cmdlet 的讨论超出了这本书的范围，以下步骤说明了如何实现一种简单的 cmdlet 的过程。

有关如何编写一个 PowerShell cmdlet 的更多信息，请参阅 MSDN 主题，“How to Create a Windows PowerShell Cmdlet” 它位于 <http://msdn2.microsoft.com/en-us/library/ms714598.aspx>。

步骤 1：下载 Windows SDK

Windows SDK 包含示例、工具、引用程序集、模板、文档和开发 PowerShell cmdlet 时使用的其他信息。你可以通过在网站 <http://download.microsoft.com> 上搜索“Microsoft Windows SDK”来下载最新的 Windows Vista SDK。

步骤 2：创建一个文件用于保存 cmdlet 和管理单元源代码

创建一个名为 `InvokeTemplateCmdletCommand.cs` 的文件，内容如例 15-12 所示，并把它保存到硬盘。

例 15-12：`InvokeTemplateCmdletCommand.cs`

```
using System;
using System.ComponentModel;
using System.Management.Automation;

/*
To build and install:

1)
Set-Alias csc $env:WINDIR\Microsoft.NET\Framework\v2.0.50727\csc.exe

2)
Set-Alias installutil `
    $env:WINDIR\Microsoft.NET\Framework\v2.0.50727\installutil.exe
3)
$ref = [PsObject].Assembly.Location
csc /out:TemplateSnapin.dll /t:library InvokeTemplateCmdletCommand.cs /r:$ref

4)
installutil TemplateSnapin.dll
5)
Add-PSSnapin TemplateSnapin

To run:
PS >Invoke-TemplateCmdlet

To uninstall:
installutil /u TemplateSnapin.dll
*/
namespace Template.Commands
{
    [Cmdlet("Invoke", "TemplateCmdlet")]
    public class InvokeTemplateCmdletCommand : Cmdlet
    {
        [Parameter(Mandatory=true, Position=0, ValueFromPipeline=true)]
        public string Text
```

```
        {
            get
            {
                return text;
            }
            set
            {
                text = value;
            }
        }
        private string text;
    }

    protected override void BeginProcessing()
    {
        WriteObject("Processing Started");
    }

    protected override void ProcessRecord()
    {
        WriteObject("Processing " + text);
    }

    protected override void EndProcessing()
    {
        WriteObject("Processing Complete.");
    }
}

[RunInstaller(true)]
public class TemplateSnapin : PSSnapIn
{
    public TemplateSnapin()
        : base()
    {
    }

    ///<summary>The snapin name which is used for registration</summary>
    public override string Name
    {
        get
        {
            return "TemplateSnapin";
        }
    }

    /// <summary>Gets vendor of the snapin.</summary>
    public override string Vendor
    {
        get
        {
            return "Template Vendor";
        }
    }

    /// <summary>Gets description of the snapin. </summary>
    public override string Description
    {

```

```
get
{
    return "This is a snapin that provides a template cmdlet.";
}
}

}
```

步骤 3：编译管理单元

一个 PowerShell cmdlet 是一个简单的 .NET 类。包含已编译的 cmdlet 的 DLL 被称为一个管理单元 (*snapin*)。

```
Set-Alias csc $env:WINDIR\Microsoft.NET\Framework\v2.0.50727\csc.exe
$ref = [PsObject].Assembly.Location
csc /out:TemplateSnapin.dll /t:library InvokeTemplateCmdletCommand.cs /r:$ref
```

步骤 4：安装并注册管理单元

一旦你已编译管理单元，下一步就是将其注册。注册管理单元为 PowerShell 提供了需要的信息。此命令需要管理员的权限。

```
Set-Alias installutil `
$env:WINDIR\Microsoft.NET\Framework\v2.0.50727\installutil.exe
installutil TemplateSnapin.dll
```

步骤 5：将管理单元添加到你的会话

尽管第 4 步注册了管理单元，但 PowerShell 不会将命令添加到你的活动会话中，除非你调用外接程序 PsSnapin 命令来将管理单元添加到你的会话中。

```
Add-PsSnapin TemplateSnapin
```

步骤 6：使用管理单元

一旦你已经将管理单元添加到你的会话中，你就可以从该管理单元调用命令，就像你可以调用任何其他 cmdlet 一样。

```
PS >"Hello World" | Invoke-TemplateCmdlet
Processing Started
Processing Hello World
Processing Complete.
```

15.13 添加 PowerShell 脚本到你自己的程序

问题

你想为你的用户提供简便的方法来自动化地运行你的程序，但你不想自己编写一种脚本语言。

解决方案

PowerShell吸引人的方面之一是让你可以轻松地将其许多的功能添加到你自己的程序中。这是因为 PowerShell 的核心是一个可以使用任何应用程序功能强大的引擎。PowerShell 控制台应用程序实际上只是用于该引擎的一个基于文本的界面。

虽然 PowerShell 宿主模型的全面讨论超出了本书的范围，但下面的示例向你的用户阐释了公开应用程序的功能到脚本之后的技术。

例 15-13，假设当你收到一封电子邮件时，电子邮件应用程序运行某一规则。你需要设计一个标准接口，允许用户可以创建简单的规则，你还想为用户提供一种方法来编写非常复杂的规则。相对于自己设计一种脚本语言，你只需使用 PowerShell 的脚本语言即可。在下面的示例中，我们为用户编写的脚本提供一个称为 \$message 的变量，该变量表示当前消息，然后程序再运行命令。

```
PS >Get-Content VerifyCategoryRule.ps1
if($message.Body -match "book")
{
    [Console]::WriteLine("This is a message about the book.")
}
else
{
    [Console]::WriteLine("This is an unknown message.")
}
PS >.\RulesWizardExample.exe (Resolve-Path VerifyCategoryRule.ps1)
This is a message about the book.
```

有关如何在你自己的应用程序中调用 PowerShell 的详细信息，请参见 MSDN 主题，“How to Create a Windows PowerShell Hosting Application”它位于 <http://msdn2.microsoft.com/en-us/library/ms714661.aspx>。

步骤 1：下载 Windows SDK

Windows SDK 包含示例、工具、引用程序集、模板、文档和开发 PowerShell cmdlet 时使用的其他信息。通过在网站 <http://download.microsoft.com> 上搜索“Microsoft Windows SDK”来下载最新的 Windows Vista SDK。

步骤 2：创建一个文件用于保存托管源代码

创建一个名为 RulesWizardExample.cs 的文件，其内容如例 15-13 所示，并将其保存在硬盘上。

例 15-13：RulesWizardExample.cs

```
using System;
using System.Management.Automation;
using System.Management.Automation.Runspaces;

namespace Template
{
    // Define a simple class that represents a mail message
    public class MailMessage
    {
        public MailMessage(string to, string from, string body)
        {
            this.To = to;
            this.From = from;
            this.Body = body;
        }

        public String To;
        public String From;
        public String Body;
    }

    public class RulesWizardExample
    {
        public static void Main(string[] args)
        {
            // Ensure that they've provided some script text
            if(args.Length == 0)
            {
                Console.WriteLine("Usage:");
                Console.WriteLine(" RulesWizardExample <script text>");
                return;
            }

            // Create an example message to pass to our rules wizard
            MailMessage mailMessage =
                new MailMessage(
                    "guide_feedback@LeeHolmes.com",
                    "guide_reader@example.com",
                    "This is a message about your book.");

            // Create a runspace, which is the environment for
            // running commands
            Runspace runspace = RunspaceFactory.CreateRunspace();
            runspace.Open();

            // Create a variable, called "$message" in the Runspace, and populate
            // it with a reference to the current message in our application.
            // Pipeline commands can interact with this object like any other
        }
    }
}
```

```
// .Net object.  
runspace.SessionStateProxy.SetVariable("message", mailMessage);  
  
// Create a pipeline, and populate it with the script given in the  
// first command line argument.  
Pipeline pipeline = runspace.CreatePipeline(args[0]);  
  
// Invoke (execute) the pipeline, and close the runspace.  
pipeline.Invoke();  
runspace.Close();  
}  
}
```

步骤 3：编译并运行该示例

尽管该示例本身提供了很少功能，但它阐释了如何把PowerShell脚本添加到你自己的程序的核心概念。

```
Set-Alias csc $env:WINDIR\Microsoft.NET\Framework\v2.0.50727\csc.exe  
$dll = [PsObject].Assembly.Location  
Csc RulesWizardExample.cs /reference:$dll  
RulesWizardExample.exe <script commands to run>
```

```
PS >.\RulesWizardExample.exe '[Console]::WriteLine($message.From)  
guide.reader@example.com'
```

参考

- Windows PowerShell 使用指南中的“结构化命令”

第 16 章

安全和脚本签名

16.0 简介

安全在 PowerShell 中扮演着两个重要的角色。第一个角色是 PowerShell 本身的安全性：在 Windows 中，脚本语言一直是基于电子邮件的恶意软件的传播者，所以 PowerShell 在设计的时候应尽量避免这种安全威胁。第二个角色是在使用你的计算机时可能会遇到的与安全相关的任务的一些设置，包括：脚本签名、证书和凭据等。

当我们谈论脚本和命令行世界中的安全性时，一个是传说中的处理大师，一个是迷信无需图形界面。最常见的误解之一是，脚本语言和命令行解释器由于某种原因允许用户绕过 Windows 图形用户界面的安全保护。

Windows 的安全模型（就像任何实际提供安全性的安全模型一样）保护资源的方式与你获得这些资源的方式是不同的。实际上，那是因为你运行程序时，如果安全机制允许，那么你可以运行；如果安全机制不允许，你就不能运行这个程序。例如，请思考更改 Windows 注册表中的关键数据的操作。如果你使用 Windows 注册表编辑器的图形用户界面，当你尝试执行某项你没有权限的操作时，它会弹出一个错误消息对话框，如图 16-1 中所示。

注册表编辑器提供这个信息是因为它无法删除那个键，而不是它本身要阻止你删除这个键值。Windows 本身会保护注册表项，而不是你运行的程序去保护注册表项。

同样，当你尝试执行一个你没有权限的操作时，PowerShell 也会提示错误信息。不是因为 PowerShell 为该操作进行了额外的安全检查，而是因为它本身也无法执行该操作：

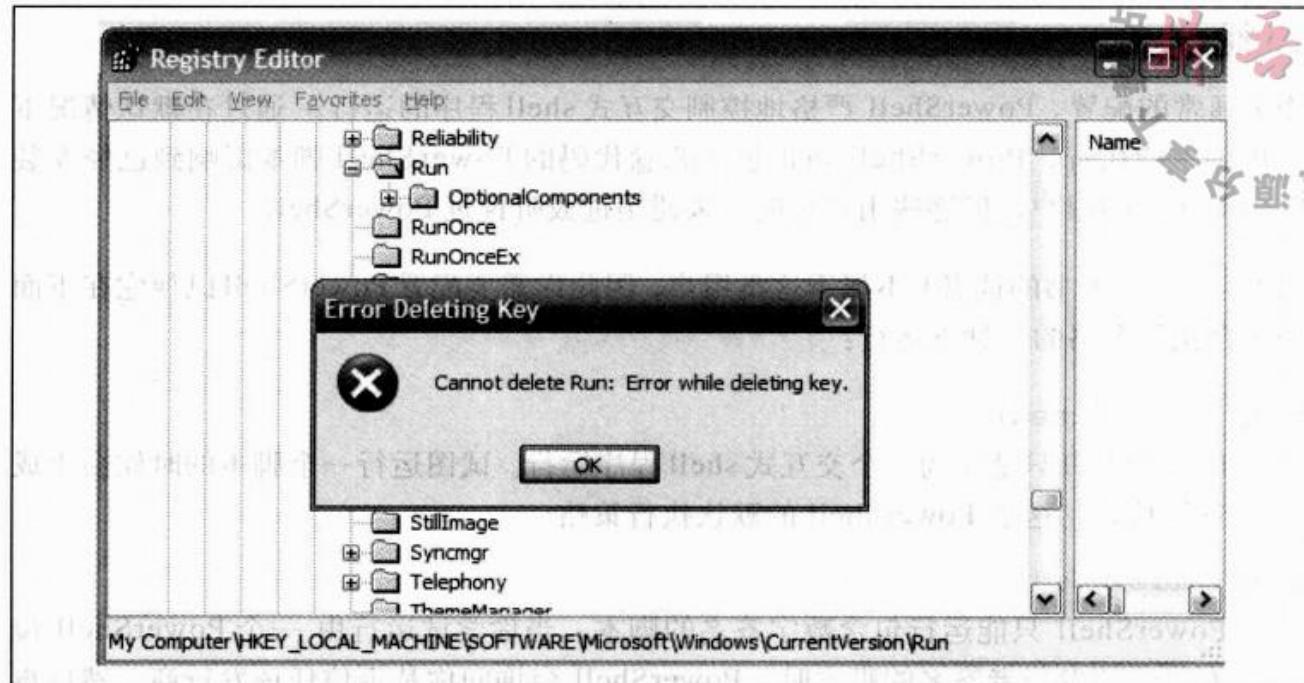


图 16-1: Windows 注册表编辑器提示的错误信息

```
PS >New-Item "HKLM:\Software\Microsoft\Windows\CurrentVersion\Run\New"
New-Item : Requested registry access is not allowed.
At line:1 char:9
+ New-Item <<< "HKLM:\Software\Microsoft\Windows\CurrentVersion\Run\New"
```

经过上面的解释后，你可能会清楚并不是运行的命令 shell 或脚本语言阻止你的操作。

16.1 通过执行策略启用脚本问题

当你尝试运行脚本的时候，PowerShell 可能会提示如下的错误信息：

```
PS>.\Test.ps1
File C:\temp\test.ps1 cannot be loaded because the execution of scripts is disabled on this system. Please see "get-help about_signing" for more details.
At line:1 char:10
+ .\Test.ps1 <<<
```

解决方案

要防止这个错误信息，你可以使用 `Set-ExecutionPolicy` 命令来更改 PowerShell 的执行策略，以允许运行脚本：

```
Set-ExecutionPolicy RemoteSigned
```

讨论

作为通常的配置，PowerShell 严格地控制交互式 shell 程序的运行。通过在默认情况下禁用脚本的执行，PowerShell 防止包含恶意代码的 PowerShell 脚本影响到已经安装 PowerShell 的用户，但这些用户可能从未使用过或听说过 PowerShell。

当然了，你（本书的读者）不属于这类用户，因此你需要配置 PowerShell 以便它在下面的四个执行策略的一种下运行：

受限 (Restricted)

PowerShell 只是作为一个交互式 shell 程序运行。试图运行一个脚本的时候会生成错误信息。这是 PowerShell 的默认执行策略。

签名 (Allsigned)

PowerShell 只能运行包含数字签名的脚本。当你尝试运行由一个 PowerShell 没有见过的发行者签名的脚本时，PowerShell 会询问你是否信任该发行商，然后再在你的系统上运行脚本。

远程签名 (推荐) (Remote Signed)

PowerShell 运行大多数脚本而不提示，但要求源自 Internet 的脚本包含数字签名。在签名 (Allsigned) 模式下，当你运行一个由它没有见过的发行者签名的脚本时，PowerShell 会询问你是否信任该发行商，并在你的系统上运行脚本。当你使用一个流行的通信程序如 Internet Explorer、Outlook 或 Messenger 下载一个 ps 脚本到你的计算机时，PowerShell 会认为它是一个来自互联网的脚本。

不受限制的 (Unrestricted)

PowerShell 不需要任何脚本包含数字签名，但当一个脚本是来自互联网的时候（像 Windows 资源管理器），PowerShell 会警告你。

你可以以管理员的身份运行 Set-ExecutionPolicy 命令来配置系统的执行策略。如果你要在 Windows Vista 上配置你的执行策略，你可以右键单击 Windows PowerShell 链接选项以管理员身份来启动 PowerShell。

注意：脚本已经签过名，并不意味着该脚本是安全的！脚本上的签名可以让你验证脚本的来源，但并不意味着你可以信任它的作者，并在你的系统上运行命令。你需要亲自决定是否运行脚本，这也是 PowerShell 会询问你的原因。

此外，你可能会直接修改注册表项来修改 PowerShell 的执行策略。这是在注册表路径下的 ExecutionPolicy 属性。HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell。

在企业的设置中，PowerShell 还可以让你通过组策略优先于本地重写此设置。有关 PowerShell 的组策略支持的详细信息，请参阅 16.4 节。

当使用执行策略检测基于 Internet 的脚本时，你可能要阻止 PowerShell 将这些脚本视为远程脚本，为此，在 Windows 资源管理器中用鼠标右键单击该文件，选择属性，然后单击取消阻止。

在企业的环境中，PowerShell 有时会警告基于 Internet 的脚本是危险的，即使它们位于网络共享中。如果解除锁定该文件不能解决该问题，你的计算机可能已配置为限制对网络共享的访问。这在 Internet Explorer 的增强的安全配置模式（Enhanced Security Configuration）下是常见的。要防止这个消息，可以将指向网络共享的路径添加到 Internet Explorer 的 Intranet 或受信任（trusted）的站点区域。有关管理 Internet Explorer 的区域映射的更多信息，请参阅 18.6 节。

有关脚本签名的更多信息，请键入 `Get-help about_signing`。有关 `Set-ExecutionPolicy` 命令的更多信息，请键入 `Get-help set-ExecutionPolicy`。

参考

- 16.4 节
- 18.6 节

16.2 PowerShell 脚本或格式文件签名

问题

你希望对一个 PowerShell 脚本进行签名，以便它可以在执行策略被设置为需要脚本签名的系统上运行。

解决方案

若要用你的标准代码签名的证书对脚本签名，可以使用 `Set-AuthenticodeSignature` 命令：

```
$cert = @(Get-ChildItem cert:\CurrentUser\My -CodeSigning)[0]
Set-AuthenticodeSignature file.ps1 $cert
```

此外，你也可以使用其他传统的应用程序（如 `signtool.exe`）来签名 `PowerShell.ps1` 和 `.ps1xml` 文件。

讨论

签名脚本或格式化的文件为你和你的客户提供了两个主要优点：发行者标识和文件完整性。当你签名脚本或格式化文件的时候，PowerShell 将你的数字签名追加到该文件的末尾。此签名可以用来验证该文件来自你，还可以确保没有人可以篡改该文件中的内容。如果你尝试加载已经损坏的文件，PowerShell 会提供以下的错误信息：

```
File C:\temp\test.ps1 cannot be loaded. The contents of file  
C:\temp\test.ps1  
may have been tampered because the hash of the file does not match the hash  
stored in the digital signature. The script will not execute on the system. Please  
see "get-help about_signing" for more details..  
At line:1 char:10  
+ .\test.ps1 <<<
```

当对脚本和格式化文件进行签名的时候，PowerShell 参与了标准的 Windows 验证码基础结构的制定。因此，你已经了解的签名文件和使用它们的签名技术，可以继续用于 PowerShell 脚本和格式化文件。Set-AuthenticodeSignature 命令设计的主要目的是，支持脚本和格式化文件，它还支持 DLL 和其他标准的 Windows 可执行文件类型。

要签名文件，Set-AuthenticodeSignature 命令要求你给他提供具有有效的代码签名的证书。大多数证书颁发机构提供付费的可信的代码签名的证书。通过从著名的证书颁发机构（如 VeriSign 或 Thawte）获得一个可信的代码签名证书，你可以确保所有的用户都能够验证你的脚本上的签名。某些联机服务提供了非常便宜的代码签名的证书，但要注意的是许多计算机可能无法验证由这些证书创建的数字签名。

注意：你仍可以在你的计算机上通过生成你自己的代码签名的证书来获得许多代码签名的好处。尽管其他计算机不能够识别该签名，它仍然可以确保你计算机上的内容不会被恶意篡改。有关此方法的详细信息，请参阅下面的 16.3 节。

通过 -TimeStampServer 参数，你可以在代码签名的证书过期后使你的脚本或格式文件上的签名仍然有效。

有关 Set-AuthenticodeSignature 命令的更多信息，请键入 **Get-help set-AuthenticodeSignature**。

参考

- 16.3 节

16.3 编程：创建一个自签名的证书

讨论

我们可以受益于确保签名脚本的内容不会被恶意篡改，而不必支付一个正式的代码签名的证书的费用。你可以通过创建一个自签名的证书来执行此操作。使用自签名证书签名的脚本将无法在其他计算机识别，但仍允许你在自己的计算机上使用该签名脚本。

当示例 16-1 运行的时候，它会提示你输入一个密码。Windows 使用此密码来防止恶意程序文件以你的名义自动签名。

例 16-1：New-SelfSignedCertificate.ps1

```
#####
## New-SelfSignedCertificate.ps1
##
## Generate a new self-signed certificate. The certificate generated by these
## commands allow you to sign scripts on your own computer for protection
## from tampering. Files signed with this signature are not valid on other
## computers.
##
## ie:
##
## PS >New-SelfSignedCertificate.ps1
##
#####

if(-not (Get-Command makecert.exe -ErrorAction SilentlyContinue))
{
    $errorMessage = "Could not find makecert.exe. " +
        "This tool is available as part of Visual Studio, or the Windows SDK."
    Write-Error $errorMessage
    return
}

$keyPath = Join-Path ([IO.Path]::GetTempPath()) "root.pvk"

## Generate the local certification authority
makecert -n "CN=PowerShell Local Certificate Root" -a sha1 -
    -eku 1.3.6.1.5.5.7.3.3 -r -sv $keyPath root.cer -
    -ss Root -sr localMachine

## Use the local certification authority to generate a self-signed
## certificate
makecert -pe -n "CN=PowerShell User" -ss MY -a sha1 -
    -eku 1.3.6.1.5.5.7.3.3 -iv $keyPath -ic root.cer

## Remove the private key from the filesystem.
Remove-Item $keyPath
```

```
## Retrieve the certificate
Get-ChildItem cert:\currentuser\my -codesign |
    Where-Object { $_.Subject -match "PowerShell User" }
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

16.4 管理企业中的 PowerShell 安全性问题

你希望控制企业环境中的 PowerShell 的安全功能。

解决方案

若要在企业中管理 PowerShell 的安全功能，可以执行下列操作：

- 应用 PowerShell 的组策略模板，来通过组策略控制 PowerShell 的执行策略。
- 部署 Microsoft 证书服务来为域账户自动生成可信的代码签名的证书。
- 应用软件限制策略，来防止 PowerShell 信任特定的脚本发行商。

讨论

应用 PowerShell 的组策略模板

Windows PowerShell 的管理模板可以在计算机和每个用户的级别上重写计算机的本地执行策略。要获取 PowerShell 的管理模板，请访问 <http://www.microsoft.com/downloads> 并搜索“Administrative templates for Windows PowerShell”。

注意：尽管组策略设置可以覆盖本地首选项，PowerShell 的执行策略不应视为一种防止用户在系统中破坏的安全措施。它只是一个安全措施，可以防止在系统上运行不受信任的脚本。就像介绍中提到的，PowerShell 只是一个允许用户执行它们已有权限的程序的手段。

一旦安装了 Windows PowerShell 的管理模板，你就可以启动组策略对象编辑器 MMC 管理单元。右键单击管理模板（Administrative Templates），然后选择添加或删除管理

模板 (Add/Remove Administrative Templates)。你会在安装位置中发现 Windows PowerShell 管理模板。组策略编辑器 MMC 管理单元提供 PowerShell 作为它的管理模板节点下的选项，如图 16-2 中所示。

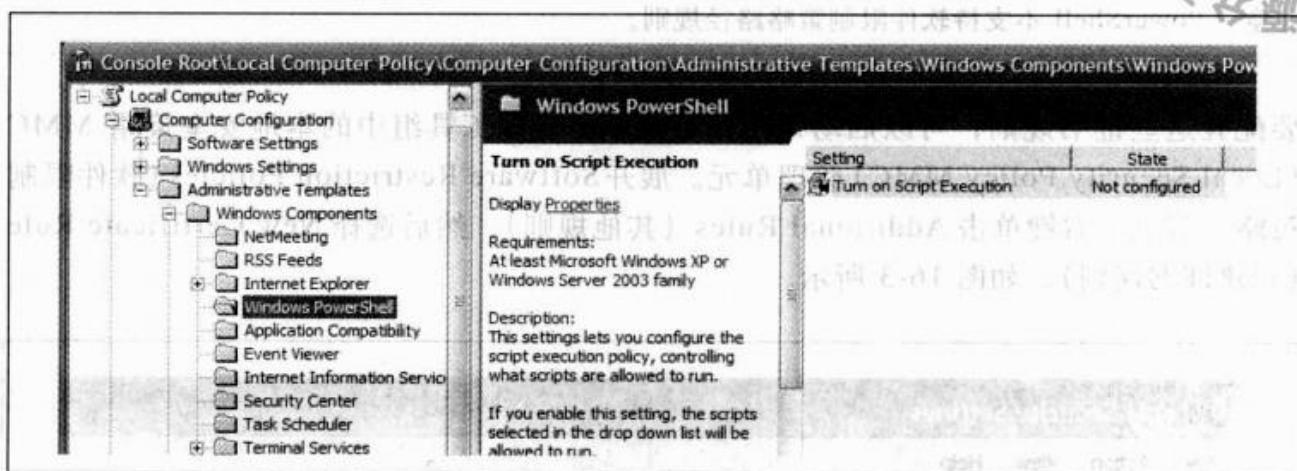


图 16-2：PowerShell 组策略配置

PowerShell 组策略默认状态是未配置。在这种情况下，PowerShell 将本地计算机执行策略作为首选项（如 16.1 节中所述）。如果将状态更改为启用选项（或禁用），PowerShell 将使用这种配置，而不是使用该计算机的本地执行策略作为首选项。

注意：PowerShell 无论在什么情况下都遵循这些组策略设置。这其中包括的设置可能会使计算机的管理员觉得降低了系统的安全性，如一个无限制的策略替代本地签名的组策略。

每个用户组策略设置会覆盖本地计算机的组策略，每个机器的组策略设置将覆盖每个用户设置。

部署 Microsoft Certificate 服务

尽管超出了本书的范围，Microsoft 证书服务允许你自动将代码签名的证书部署到任何或所有域用户。它可以帮助用户防止意外或恶意脚本篡改。

有关本主题介绍，可以访问 <http://technet.microsoft.com> 和搜索“Enterprise Design for Certificate Services”。有关脚本签名的详细信息，请参阅 16.2 节。

应用软件限制策略

虽然不常见，但你有时可能需要防止 PowerShell 运行由特定的发布者签名的脚本。如果该脚本属于签名验证（例如，它是一个远程脚本或者 PowerShell 的执行策略被设置为

AllSigned), PowerShell就可以通过计算机的软件限制策略中的证书规则进行配置来阻止脚本的运行。

注意：PowerShell 不支持软件限制策略路径规则。

要配置这些证书规则，可以启动列在开始菜单中管理工具组中的本地安全策略 MMC (Local Security Policy MMC) 管理单元。展开 Software Restriction Policies (软件限制策略) 节点，右键单击 Additional Rules (其他规则)，然后选择 New Certificate Rule (新建证书规则)，如图 16-3 所示。

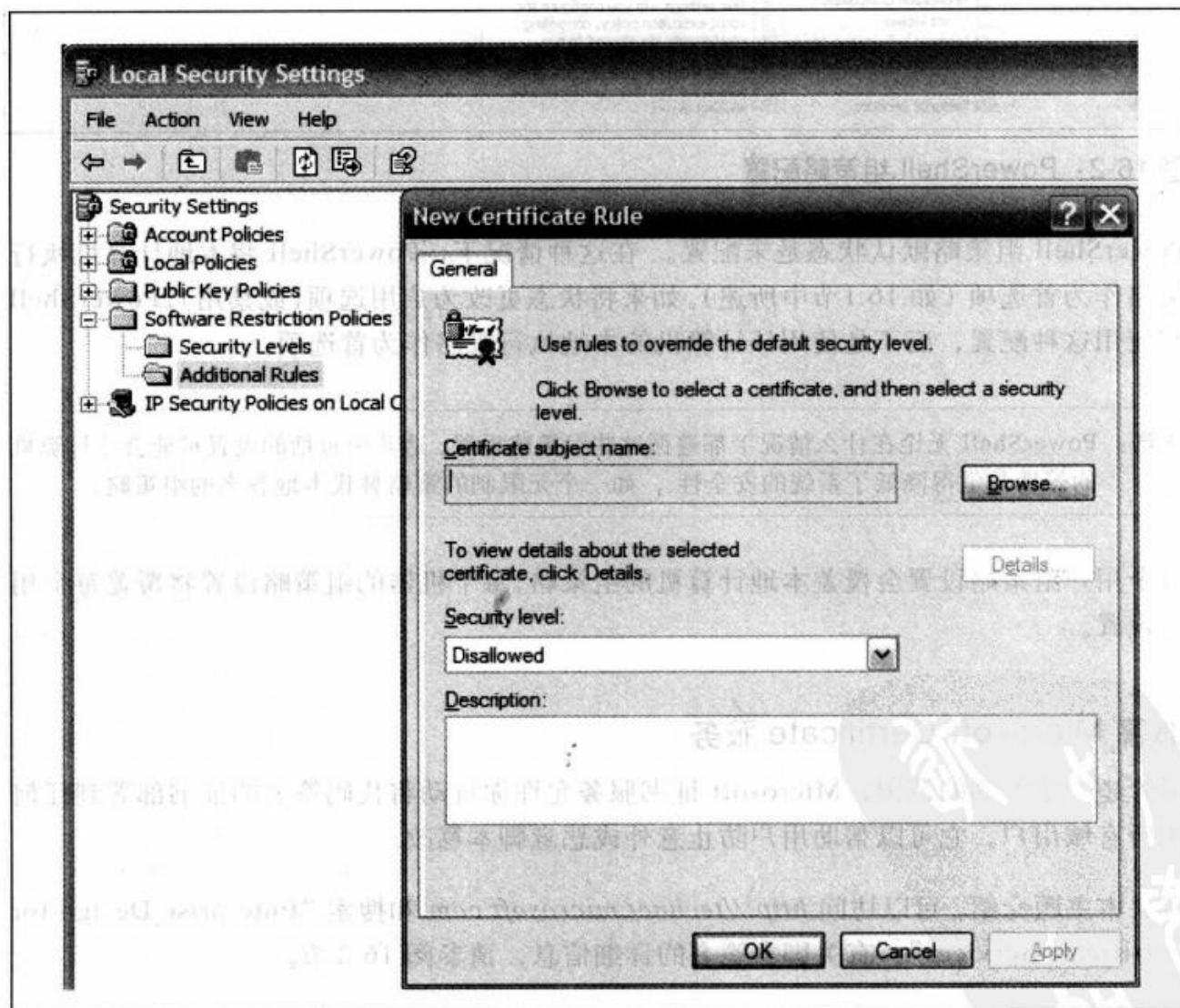


图 16-3：添加一个新的证书规则

浏览到你希望阻止的发行商证书，然后单击确定（OK）以阻止该发行商。

你还可以创建仅允许从一个集中管理的安全列表证书中选择证书的策略。为此，要么选择只允许所有管理员来管理可靠发行商 (Allow only all administrators to manage Trusted Publishers)，要么选择只允许企业管理员可以从受信任的发布服务商 (Allow only enterprise administrators to manage Trusted Publishers) 管理对话框中选择受信任的发布商。

参考

- 16.1 节
- 16.2 节

16.5 验证 PowerShell 脚本的数字签名

问题

你希望验证 PowerShell 脚本或格式文件的数字签名。

解决方案

若要验证一个脚本或格式文件的签名，可以使用 Get-AuthenticodeSignature 命令：

```
PS >Get-AuthenticodeSignature .\test.ps1

Directory: C:\temp

SignerCertificate                               Status      Path
-----                                         Valid       test.ps1
FD48FAA9281A657DBD089B5A008FAFE61D3B32FD
```

讨论

Get-AuthenticodeSignature 命令从一个文件中获取验证码签名。该文件可以是一个 PowerShell 脚本或格式文件，这个命令还支持 DLL 和其他 Windows 标准的可执行文件类型。

默认情况下，PowerShell 显示该签名证书的概述和它的状态。有关签名的详细信息，使用 Format-List 命令，如例 16-2 中所示。

例 16-2：PowerShell 显示有关验证码签名的详细信息

```
PS >Get-AuthenticodeSignature .\test.ps1 | Format-List
```

```
SignerCertificate : [Subject]
                    CN=PowerShell User

[Issuer]
CN=PowerShell Local Certificate Root

[Serial Number]
454D75B8A18FBDB445D8FCEC4942085C

[Not Before]
4/22/2007 12:32:37 AM

[Not After]
12/31/2039 3:59:59 PM

[Thumbprint]
FD48FAA9281A657DBD089B5A008FAFE61D3B32FD

TimeStamperCertificate :
Status          : Valid
StatusMessage   : Signature verified.
Path            : C:\temp\test.ps1
```

有关 `Get-AuthenticodeSignature` 命令的更多信息，请键入 **Get-help Get-AuthenticodeSignature**。

16.6 安全地处理敏感信息

问题

你希望从用户处请求敏感的信息，但要尽可能安全地执行此操作。

解决方案

若要安全地处理敏感信息，只要有可能，都要将其存储在安全字符串（`SecureString`）中。`Read-host` 命令（使用 `-AsSecureString` 参数）可以通过把 `SecureString` 作为返回用户的响应来提示用户输入（和处理）的是敏感信息：

```
PS >$secureInput = Read-Host -AsSecureString "Enter your private key"
Enter your private key: *****
PS >$secureInput
System.Security.SecureString
```

讨论

当你使用 .NET Framework 中的任何字符串时，它都会保持该字符串，以便可以在稍后有效地重复使用它。与大多数 .NET 数据不同，未使用的字符串会在完成使用它们后保

持在内存里。当此数据在内存中时，始终有机会可能将它捕获到一个故障转储中，或在换页操作中交换到磁盘。因为某些数据（如密码和其他机密信息）可能是比较敏感的，.NET Framework 提供了 `SecureString` 类，可以对存储在内存中的文本数据进行加密。需要与在 `SecureString` 中的纯文本格式的数据进行交互的代码会很安全地执行。

当一个 cmdlet 发出者向你要求敏感数据（例如加密密钥）的时候，最佳的做法是将该参数指定为 `SecureString`，以确保你的信息保密。你可以提供该参数使用 `SecureString` 变量作为输入，如果你不提供，主机会提示您输入 `SecureString`。PowerShell 还支持两个 cmdlet (`Convert To-SecureString` 和 `Convert From-SecureString`)，它们使你可以安全地将数据保存到磁盘。有关安全地在磁盘上存储信息的更多信息，请参阅 16.9 节。

注意：凭据是一种常见的敏感信息。关于如何安全地管理 PowerShell 中的凭据，请参阅 16.7 节。

默认情况下，`SecureString` cmdlet 使用 Windows 数据保护 API 对文本进行转换。它使用你的 Windows 登录凭据来加密数据，因此，只有你可以解密已加密的数据。如果你希望导出的数据能在另一个系统或单独的用户账户下使用，你可以使用 cmdlet 选项允许你提供一个明确的密钥。PowerShell 视此敏感数据为不透明的。

但是，有很多情况下你可能会想要自动提供一个 `SecureString` 输入，而不用主机提示你输入。在这种情况下，理想的解决方案是使用 `ConvertTo-SecureString` cmdlet 来从磁盘导入一个之前导出的 `SecureString`，这样将保留你的数据的机密性并仍允许你自动输入。

如果数据是动态的（例如，来自一个 CSV），可以使用 `Convert To-SecureString` cmdlet 支持的 `-AsPlainText` 参数：

```
$secureString = ConvertTo-SecureString "Kinda Secret" -AsPlainText -Force
```

在本例子中，由于你已经提供了纯文本输入，再将此数据放在 `SecureString` 中就不能提供一个安全的保障。为防止安全性的错误，cmdlet 要求使用 `-force` 参数来将纯文本格式的数据转换成一个 `SecureString`。

在 `SecureString` 中有数据后，你可能要访问它的纯文本形式。PowerShell 没有提供直接的方式来完成此功能，那违背了 `SecureString` 的目的。如果你仍想将 `SecureString` 转换为纯文本，你有两种选择：

1. 使用 `PsCredential` 类的 `GetNetworkCredential()` 方法

```
$secureString = Read-Host -AsSecureString  
$temporaryCredential = New-Object `
```

```
System.Management.Automation.PsCredential "TempUser", $secureString  
$unsecureString = $temporaryCredential.GetNetworkCredential().Password
```

2. 使用 .NET Framework 的 Marshal 类

```
$secureString = Read-Host -AsSecureString  
$unsecureString = [Runtime.InteropServices.Marshal]::PtrToStringAuto([  
    Runtime.InteropServices.Marshal]::SecureStringToBSTR($secureString))
```

参考

- 16.7 节
- 16.9 节

16.7 安全地要求用户名和密码

问题

如何在尽可能安全的状态下去要求用户提供用户名和密码。

解决方案

若要从用户处请求一个凭据，可以使用 Get-Credential cmdlet：

```
$credential = Get-Credential
```

讨论

Get-Credential cmdlet 尽可能安全地读取来自用户的凭据，并确保用户的密码始终保持高度受保护。有关在脚本中有效地使用 Get-Credential cmdlet 的示例，请参阅 16.8 节。

有了用户名和密码后，你可以将该信息传递给任何其他可以接受 PowerShell 凭据对象的命令，而不必担心泄露敏感信息。如果命令不能接受一个 PowerShell 凭据对象（但它支持 SecureString 来存放敏感信息），那么生成的 PsCredential 对象提供一个用户名 (Username) 属性，该属性返回在凭据中的用户名称；一个密码 (Password) 属性，该属性返回一个包含用户的密码的 SecureString。

遗憾的是，不是所有需要凭据的命令都可以接受 PowerShell 凭据或 SecureString。如果你要向这些命令或 API 调用提供一个凭据，PsCredential 对象提供了一个 GetNetworkCredential() 方法，可以将 PowerShell 凭据转换为安全性较低的 NetworkCredential 对象。一旦你已经将凭据转换为 NetworkCredential，你可以从

原始的凭据中 Username 属性和 Password 属性访问解密的用户名和口令。.NET Framework 中许多与网络相关的类直接支持 NetworkCredential 类。

注意： NetworkCredential 类与 PsCredential 类相比是不太安全的，因为它以纯文本格式存储用户的密码。有关以纯文本形式存储敏感信息的安全含义的更多信息，请参阅 16.6 节。

如果频繁地运行脚本需要凭据，你可以考虑在内存中缓存这些凭据以提高该脚本的可用性。例如，在脚本调用 Get-Credential cmdlet 的区域中，你可以使用例 16-3 显示的技术来缓存凭据。

例 16-3：在内存中缓存凭据来提高可用性

```
$credential = $null
if(Test-Path Variable:\Lee.Holmes.CommonScript.CachedCredential)
{
    $credential = ${GLOBAL:Lee.Holmes.CommonScript.CachedCredential}
}

${GLOBAL:Lee.Holmes.CommonScript.CachedCredential} =
Get-Credential $credential

$credential = ${GLOBAL:Lee.Holmes.CommonScript.CachedCredential}
```

该脚本在凭据第一次调用它的时候，提示用户输入凭据，但在后续调用的时候使用缓存的凭据。有关 Get-Credential cmdlet 的更多信息，请键入 **Get-help Get-Credential**。

参考

- 16.6 节
- 16.8 节

16.8 编程：作为另一个用户启动一个进程

讨论

如果你的脚本要求用户凭据，那么你希望在 PowerShell PsCredential 对象中存储这些凭据。它允许你安全地存储这些凭据，或将其传递给其他可以接受 PowerShell 凭据的命令。在你编写一个接受凭据的脚本时，可以考虑让用户提供一个用户名或一个预先存在的凭据。例 16-4 演示这样的一种有用的方法。为演示此框架，该脚本允许你作为其他用户来启动进程。

例 16-4: Start-ProcessAsUser.ps1

```
#####
## Start-ProcessAsUser.ps1
## Launch a process under alternate credentials, providing functionality
## similar to runas.exe.
##
## ie:
## PS >$file = Join-Path ([Environment]::GetFolderPath("System")) certmgr.msc
## PS >Start-ProcessAsUser Administrator mmc $file
##
##
#####
param(
    [System.Management.Automation.Credential]$credential,
    [string]$process = $(throw "Please specify a process to start."),
    [string]$arguments = ""
)

## Create a real credential if they supplied a username
if($credential -is "String")
{
    $credential = Get-Credential $credential
}

## Exit if they canceled out of the credential dialog
if(-not ($credential -is "System.Management.Automation.PsCredential"))
{
    return
}

## Prepare the startup information (including username and password)
$startInfo = New-Object Diagnostics.ProcessStartInfo
$startInfo.Filename = $process
$startInfo.Arguments = $arguments

## If we're launching as ourselves, set the "runas" verb
if(($credential.Username -eq "$ENV:Username") -or
    ($credential.Username -eq "\$ENV:Username"))
{
    $startInfo.Verb = "runas"
}
else
{
    $startInfo.UserName = $credential.Username
    $startInfo.Password = $credential.Password
    $startInfo.UseShellExecute = $false
}
```

```
## Start the process
[Diagnostics.Process]::Start($startInfo)
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

16.9 在磁盘上安全地存储凭据

问题

你的脚本执行的操作需要凭据，但你不希望在运行时与用户交互。

解决方案

若要安全地存储凭据的密码到磁盘，以便你的脚本可以自动加载，需要使用 ConvertFrom-SecureString 和 ConvertTo-SecureString 命令。

将凭据的密码保存到磁盘

将密码存储在磁盘上的第一步通常是手动。提供的一个已经存储在 \$credential 变量中的凭据，你可以使用以下命令安全地把密码导出到 password.txt：

```
PS >$credential.Password | ConvertFrom-SecureString |
    Set-Content c:\temp\password.txt
```

通过存储在磁盘上的密码重新创建凭据

在你想要自动运行的脚本中，添加下面的命令：

```
$password = Get-Content c:\temp\password.txt | ConvertTo-SecureString
$credential = New-Object System.Management.Automation.PsCredential
    "CachedUser", $password
```

这些命令创建一个新的凭据对象（用于 *CachedUser* 用户），并将该对象存储在 \$credential 变量中。

讨论

当你阅读该解决方案的时候，你首先可能会谨慎地考虑是否在磁盘上保存密码。这是很正常的，要谨慎地在你的硬盘驱动器上保存敏感的信息，`ConvertFrom-SecureString` cmdlet 使用 Windows 标准数据保护 API 来加密此数据。这样可以确保只有你的用户账户可以正确地解密其内容。

虽然保持密码安全是一项重要的安全功能，但你可能有时需要在磁盘上存储密码（或其他敏感信息），以便其他账户可以访问它。这通常是用于为服务账户运行脚本或为计算机之间进行传输设计的脚本。`ConvertFrom-SecureString` 和 `ConvertTo-SecureString` 命令允许你指定加密密钥，从而支持此功能。

注意：与一个硬编码加密键一起使用的时候，这种技术不再作为一种安全措施。如果用户可以访问自动运行的脚本的内容，则它们有权访问加密密钥。如果用户有权访问加密密钥，则它们具有你试图保护数据的访问权。

尽管该解决方案将密码存储在一个特定的命名文件中，但更常见的是存储在一个更为通用的位置，例如包含脚本的目录或包含你的配置文件的目录。

要从与你的配置文件相同的位置加载 `password.txt`, 可以使用下面的命令:

```
$passwordFile = Join-Path (Split-Path $profile) password.txt  
$password = Get-Content $passwordFile | ConvertTo-SecureString
```

若要了解如何从与你的脚本相同的位置加载它，请参阅 14.5 节。

有关ConvertTo-SecureString和ConvertFrom-SecureString cmdlet的更多信息，请键入 **Get-Help ConvertTo-SecureString** 或 **Get-Help ConvertFrom-SecureString**。

参考

- 14.5 节

16.10 访问用户和计算机证书

问题

如何检索当前用户或本地计算机证书的信息。

解决方案

若要浏览并检索本地计算机上的证书，可以使用 PowerShell 的证书驱动器。这个驱动器由证书提供商创建，如例 16-5 中所示。

例 16-5：了解证书提供商中的证书

```
PS >Set-Location cert:\CurrentUser\  
PS >$cert = Get-ChildItem -Recurse -CodeSign  
PS >$cert | Format-List  
  
Subject      : CN=PowerShell User  
Issuer       : CN=PowerShell Local Certificate Root  
Thumbprint    : FD48FAA9281A657DBD089B5A008FAFE61D3B32FD  
FriendlyName :  
NotBefore    : 4/22/2007 12:32:37 AM  
NotAfter     : 12/31/2039 3:59:59 PM  
Extensions   : {System.Security.Cryptography.Oid, System.Security.Cryptography.Oid}
```

讨论

该证书驱动器提供了一种有用的方法来导航并查看当前用户或本地计算机中的证书。例如，如果你的执行策略要求使用数字签名，那么下面的命令告诉你哪些发行商是受信任的可以在你的系统上运行脚本：

```
Get-ChildItem cert:\CurrentUser\TrustedPublisher
```

证书提供者可能最常用于为 Set-AuthenticodeSignature cmdlet 选择一个代码签名的证书。

下面的命令选择最后一个过期的“最佳”代码签名证书：

```
$certificates = Get-ChildItem Cert:\CurrentUser\My -CodeSign  
$signingCert = $($certificates | Sort -Descending NotAfter)[0]
```

在此 -CodeSign 参数，让你可以在证书存储区中搜索支持代码签名的证书。若要搜索用于其他目的的证书，请参阅 16.11 节。

虽然证书提供者可用于从计算机的证书存储区中浏览和检索信息，但不允许你从这些位置添加或删除项目。如果你想管理存储区中的证书，.NET Framework 中 System.Security.Cryptography.X509Certificates.X509Store 类（和其他在命名空间 System.Security.Cryptography.X509Certificates 中相关的类）支持该功能。

有关证书提供程序的详细信息，请键入 **Get-Help Certificate**。有关使用 .NET Framework 中的类的详细信息，请参阅 3.4 节。

参考

- 第三章 3.4 节

16.11 编程：搜索证书存储区

讨论

一个证书提供程序的非常有用的功能是它提供了一个 -CodeSign 参数，可以在证书存储中搜索支持代码签名的证书。当然，代码签名的证书，不是唯一的证书类型；其他经常使用的证书类型包括加密文件系统、客户端身份验证等等。

例 16-6 可以让你通过证书的提供者搜索一个支持给定的增强型密钥用法 (EKU) 的证书。

例 16-6：Search-CertificateStore.ps1

```
#####
## Search-CertificateStore.ps1
##
## Search the certificate provider for certificates that match the specified
## Enhanced Key Usage (EKU.)
##
## ie:
## PS >Search-CertificateStore "Encrypting File System"
##
#####
param(
    $ekuName = $(throw "Please specify the friendly name of an " +
                      "Enhanced Key Usage (such as 'Code Signing')")
)
##
## Go through every certificate in the current user's "My" store
foreach($cert in Get-ChildItem cert:\CurrentUser\My)
{
    ## For each of those, go through its extensions
    foreach($extension in $cert.Extensions)
    {
        ## For each extension, go through its Enhanced Key Usages
        foreach($certEku in $extension.EnhancedKeyUsages)
        {
            ## If the friendly name matches, output that certificate
            if($certEku.FriendlyName -eq $ekuName)
            {
                $cert
```

有关银行账户及多信息，首参见上文。

参考

· 上市

在股票市场上，上市是指一家公司首次向公众出售其股票。上市是企业融资的重要途径之一，也是企业走向资本市场的重要一步。上市后，公司可以利用股票市场的资金优势，扩大生产规模，提高经济效益。同时，上市也有助于提升企业的知名度和信誉度，为企业的发展提供良好的平台。然而，上市也存在一定的风险，如市场竞争激烈、股价波动大等。因此，在决定是否上市时，企业需要综合考虑自身的实际情况和市场环境，做出明智的决策。

第四部分

管理员任务

- 第 17 章，文件和目录
- 第 18 章，Windows 注册表
- 第 19 章，数据比较
- 第 20 章，事件日志
- 第 21 章，进程
- 第 22 章，系统服务
- 第 23 章，活动目录
- 第 24 章，企业级计算机管理
- 第 25 章，管理 Exchange 2007 通信服务器
- 第 26 章，管理操作管理器 2007 服务器

代暗四策
卷之二

新竹縣竹東市
秀樹路 250 號
郵政信箱 11-1
郵局編號 300
新竹市
赤林里 10
郵局編號 300
新竹市
翠華里 10
郵局編號 300
新竹市
翠華里 10
郵局編號 300

第17章

文件和目录

在日常生活中，我们常常会遇到一些问题，比如要遍历某个项目的文件夹，或者要查找某个文件，这时就需要使用到命令行工具。PowerShell 提供了强大的命令来处理这些问题。

17.0 简介

管理系统中文件和目录作业是管理系统时最常见的任务之一。当你在命令行管理计算机时你可以编写脚本去自动地管理它。

所幸，PowerShell 使得在命令行编写脚本管理文件和目录变得很容易，许多经验丰富的程序员编写脚本通常会错过。常见的例子是当你用尽了磁盘空间时你需要查找哪些文件占用了最多的磁盘空间。

通常程序员通过编写功能函数指定系统中某一具体的目录作为原始检索目录。遍历该目录中的每一个文件，检查其是否占用了很大的磁盘空间，如果是则将其加入清单中。再将该目录中的每一子目录依次作为原始目录进行遍历，程序员重复此过程（直到遍历完所有的目录为止）。

俗话说，“你能在任何语言中使用 C 语言编写程序”，然而，习惯与偏好某种语言通常会直接影响你在另外一种语言中的提高。

开始一个管理命令，PowerShell 直接支持遍历一个子目录中的所有文件，或从一个目录移动一个文件到另一个目录。这样复杂的面向程序员的脚本就变成了一条命令：

```
Get-ChildItem -Recurse | Sort-Object -Descending Length | Select -First 10
```

深入了解您喜欢的程序员的工具包之前，请检查 PowerShell 支持那些功能。在多数情况下，编写它不需要过多的编程技巧。

17.1 查找一个特定日期之前修改的所有文件

问题

如何查找一个特定日期之前最后修改的所有文件。

解决方案

若要查找一个特定日期之前修改的所有文件，先使用 `Get-ChildItem` cmdlet 列出一个目录下的所有文件，然后使用 `Where-Object` cmdlet 来筛选最后修改时间属性与你指定的日期一的文件。例如，要查找一年前创建的所有文件：

```
Get-ChildItem -Recurse | Where-Object { $_.LastWriteTime -lt "01/01/2007" }
```

讨论

通常的做法是去比较相对于某一特定日期来说最近修改过(或最近没有修改过)的文件。这与解决方案中的例子几乎一样，但是你编写脚本时可能又不知道要比较的确切的日期。

这种情况下，.NET Framework 的 `DateTime` 类中的 `AddDays()` 方法为您提供了一些简单的日期运算。如你有一个时间对象，你可以通过加、减日期操作得到全新的日期。例如：查找在过去 30 天里修改的所有文件：

```
$compareDate = (Get-Date).AddDays(-30)  
Get-ChildItem -Recurse | Where-Object { $_.LastWriteTime -ge $compareDate }
```

同样，要查找 30 天前修改的所有文件：

```
$compareDate = (Get-Date).AddDays(-30)  
Get-ChildItem -Recurse | Where-Object { $_.LastWriteTime -lt $compareDate }
```

在这个例子中，`Get-Date` cmdlet 返回一个对象，该对象代表当前日期和时间。您调用 `AddDays()` 方法从该日期中减去 30 天，则 `$compareDate` 变量获得的日期为“30 天前”的日期对象。接下来则通过 `ChildItem` cmdlet 返回用该日期与文件的最后修改时间属性比较的结果集。

注意：这个时间是管理员的幸运日！

```
PS >[DateTime]::IsLeapYear(2008)  
True  
PS >$daysTillSummer = [DateTime] "06/21/2008" - (Get-Date)  
PS >$daysTillSummer.Days  
283
```

要想获取有关Get-ChildItem cmdlet的更多信息,请键入**Get-Help Get-ChildItem**。
有关Where-Object cmdlet 的更多信息, 请参阅 2.1 节。

参考

- 2.1 节

17.2 清除或移动文件

问题

你想要清除文件内容或移动文件。

解决方案

清除文件内容可以使用 Clear-Content cmdlet, 可以使用 Remove-Item cmdlet 去移动文件。如例 17-1 所示。

例 17-1：清除文件内容并移动它

```
PS >Get-Content test.txt
Hello World
PS >Clear-Content test.txt
PS >Get-Content test.txt
PS >Get-Item test.txt

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

Mode          LastWriteTime    Length Name
----          -----        -
-a---  4/23/2007 8:05 PM      0 test.txt

PS >Remove-Item test.txt
PS >Get-Item test.txt
Get-Item : Cannot find path 'C:\temp\test.txt' because it does not exist.
At line:1 char:9
+ Get-Item <<< test.txt
```

讨论

正如名字描述的Clear-Content 和 Remove-Item cmdlet, 它们分别用来清除内容和移动项。尽管这个解决方案事例只对文件系统中的文件进行了操作, 但实际上它支持任何 PowerShell 提供程序的“内容”和“项”。例如: 支持驱动器的示例, “内容”和“项”

可以是函数 Function、别名 Alias、变量 Variable、驱动器、注册表或环境变量。驱动器不支持内容清除的功能，但你可以用 Remove-Item 去移动目录。

注意： Remove-Item cmdlet 所涵盖的标准别名有：ri、rm、rmdir、del、erase 和 rd。

使用 Get-Help Remove-Item 和 Get-Help Clear-Content 可以得到 Remove-Item 和 Clear-Content 命令的更多信息。

17.3 管理与改变文件属性

问题

你希望去更新文件的只读、隐藏或系统属性。

解决方案

大部分时间，您需要使用熟悉的 attrib.exe 程序来更改文件的属性：

```
attrib +r test.txt
attrib -s test.txt
```

若要设置只读 ReadOnly 属性，您可以选择设置该文件的 IsReadOnly 属性：

```
$file = Get-Item test.txt
$file.IsReadOnly = $true
```

要应用一组特定的属性，要在文件上使用 Attributes 属性：

```
$file = Get-Item test.txt
$file.Attributes = "ReadOnlyNotContentIndexed"
```

你可以通过 Mode 或 Attributes 属性来直接访问一个文件的属性：

```
PS >$file.Attributes = "ReadOnly", "System", "NotContentIndexed"
PS >$file.Mode
--r-s
PS >$file.Attributes
ReadOnly, System, NotContentIndexed
```

讨论

Get-Item 或 Get-ChildItem cmdlet 检索文件时，生成的文件有一个 attributes 属性。即“属性”，尚未被修改或未被删除。“属性”即“属性”而“属性”即“属性”

性。此属性不能提供 *attrib.exe* 程序提供的更多的常规属性，但它更易于将属性设置为特定的状态。

注意：在文件上设置 `Hidden` 属性将其从大多数默认视图列表中移除。如果要显示它，大多数命令需要加 `-Force` 参数。同样，在文件上设置 `ReadOnly` 的属性使该文件不能进行写入操作，除非在调用该命令时加 `-Force` 参数。

如：要怎样做才能使用 `Attributes` 属性为文件添加一个属性：

```
$file = Get-Item test.txt
$readOnly = [IO.FileAttributes]::"ReadOnly"
$file.Attributes = $file.Attributes -bor $readOnly
```

有关使用 .NET Framework 类的更多信息，请参阅 3.4 节。

要设置 `Hidden` 属性于单个文件，将其从文件夹或目录树中移除，可以使用 `Set-Item` 命令。若要设置并重写所有文件，如果想要将所有文件都设置为 `Hidden`，则必须使用 `Set-Item` 命令。

参考

- 3.4 节

有关使用 .NET Framework 类的更多信息，请参阅 3.4 节。

17.4 获取目录中的文件列表

问题

你想获取目录中的文件列表。

解决方案

若要检索目录中的文件的列表，可以使用 `Get-ChildItem cmdlet`。若要获取特定项，可以使用 `Get-Item cmdlet`：

- 使用 `Get-ChildItem cmdlet` 列出当前目录中的所有项：

```
Get-ChildItem
```

- `Get-ChildItem cmdlet` 支持使用通配符，你可以列出所有匹配通配符的项：

```
Get-ChildItem *.txt
```

- 列出当前目录（及其所有子目录），所有与通配符匹配的文件使用 `Get-ChildItem cmdlet` 的 `-Include` 和 `-Recurse` 参数：

```
Get-ChildItem -Include *.txt -Recurse
```

- 如果列出在当前目录中所有子目录，使用 `Where-Object` cmdlet 来检验其是否具备 `PsIsContainer` 属性：

```
Get-ChildItem | Where { $_.PsIsContainer }
```

- 使用 `Get-Item` cmdlet 得到一个特定项的信息：

```
Get-Item test.txt
```

讨论

尽管 `Get-ChildItem` 和 `Get-Item` cmdlet 最常用于文件系统，实际上可以用于 PowerShell 驱动器中任何项目。除了 A~Z 驱动器（标准的文件系统驱动器），它们还能用于 Alias、Cert、Env、Function、HKLM、HKCU 和 Variable 的处理。

注意：例如：列出目录中所有与通配符匹配的文件及其子目录。该示例适用于任何 PowerShell 提供程序。但是，如果想要 PowerShell 更快地检索到您想要的结果，您应该使用其提供的过滤器功能，如以下的 17.5 节中所述。

该解决方案演示了一些支持 `Get-ChildItem` cmdlet 的简单的通配符方案，但 PowerShell 实际上有高级的匹配方案。有关这些方案的详细信息，请参阅下面的 17.5 节。

在文件系统中，这些 cmdlet 返回在 .NET Framework 中分别表示文件和目录的 `System.IO.FileInfo` 和 `System.IO.DirectoryInfo` 类的实例化对象。每一个都提供了大量的有用的信息：属性、修改时间、完整的名称等。尽管默认的目录列表也提供了大量的信息，但 PowerShell 会提供更多的信息。有关使用 .NET Framework 中的类的详细信息，请参阅 3.4 节。

参考

- 3.4 节
- 17.5 节

17.5 使用匹配模式查找文件

问题

如何得到使用匹配模式筛选的文件列表

解决方案

Get-ChildItem cmdlet 的简单与高级的通配符应用：

- 要使用 PowerShell 通配符查找当前目录中所有与之匹配的项，需给出通配符在 Get-ChildItem cmdlet 中的应用：

```
Get-ChildItem *.txt
```

- 要在当前目录中查找与 *provider-specific* 的过滤器匹配的所有项目，使用 -Filter 参数进行筛选器匹配：

```
Get-ChildItem -Filter *~2*
```

- 要在当前目录中查找所有与 PowerShell 通配符不匹配的项目，使用 -Exclude 参数：

```
Get-ChildItem -Exclude *.txt
```

- 要在子目录中查找所有与 PowerShell 通配符匹配的项目使用 -Include 和 -Recurse 参数：

```
Get-ChildItem -Include *.txt -Recurse
```

- 要在子目录中查找与 *Provider-specific* 的过滤器匹配的所有项目，使用 -Filter 和 -Recurse 参数：

```
Get-ChildItem -Filter *.txt -Recurse
```

- 要在子目录中查找与 PowerShell 通配符不匹配的所有项目，使用 -Exclude 和 -Recurse 参数：

```
Get-ChildItem -Exclude *.txt -Recurse
```

Where-Object cmdlet 对正则表达式的高级支持：

- 要查找所有文件名与正则表达相匹配的项，使用 Where-Object cmdlet 比较正则表达式与文件名的属性：

```
Get-ChildItem | Where-Object { $_.Name -match '^KB[0-9]+\log$' }
```

- 要查找所有目录名与正则表达相匹配的项，使用 Where-Object cmdlet 比较正则表达式与目录名的属性：

```
Get-ChildItem -Recurse | Where-Object { $_.DirectoryName -match 'Release' }
```

- 要查找与正则表达式匹配的所有目录或文件项目，使用 Where-Object cmdlet 来比较正则表达式与 FullName（全名）属性：

```
Get-ChildItem -Recurse | Where-Object { $_.FullName -match 'temp' }
```

讨论

Get-ChildItem cmdlet 通过三个参数支持通配符的使用：

Path

这个 -Path 参数是首要（也是默认）的参数。默认位置是当前目录，您可以输入简单的路径，如，C:\ 或 D:\Documents；还可以使用含通配符的路径，如 *、*.txt、[a-z] ????.log，甚至 C:\win**.N[a-f]?F*\v2*\csc.exe。

Include/Exclude

Path 参数可与 -Include 和 -Exclude 参数组合进行过滤。如您指定 -Recurse 参数，那么 -Include 和 -Exclude 将适用于所有符合相应过滤条件的返回项目。

注意： -Include 参数最常见的错误是，当您使用相对路径而没有用通配符指定要检索的指定内容时，返回的结果会与预想结果不一样：

```
Get-ChildItem $env:WINDIR -Include *.log
```

该命令没有返回任何结果，因为您没有用通配符指定要检索的指定内容。正确的命令是：

```
Get-ChildItem $env:WINDIR\* -Include *.log
```

Filter

-Filter 参数允许您使用基于 *provider-specific* 的提供程序的语言过滤器过滤您想要的结果。PowerShell 的通配符与文件系统通配符相似，它与大多数用户在文件系统上使用的 -Filter 参数似乎是相同（和等效）的参数。在 -Filter 参数中使用程序提供者的 SQL 语法进行过滤。同样的活动目录提供程序将在 -Filter 参数中使用 LDAP（Lightweight Directory Access Protocol）轻量目录访问协议。

虽然不显著，但 PowerShell 通配符语法与文件系统提供程序的过滤语言还是有少量不完全相同的语法。例如，-Filter 参数只与短文件名匹配：

```
PS >Get-ChildItem | Select-Object Name
Name
-----
A Long File Name With Spaces Also.txt
A Long File Name With Spaces.txt

PS >Get-ChildItem *1* | Select-Object Name
PS >Get-ChildItem -Filter *1* | Select-Object Name

Name
-----
A Long File Name With Spaces.txt
```

另一方面，PowerShell 的通配符语法支持更多文件系统的本地化过滤语言。有关 PowerShell 的通配符语法的详细信息，请键入 Get-Help About_WildCard。

Where-Object cmdlet 提供执行筛选时比 PowerShell 的通配符语法高级的更多筛选操作。例如，从搜索中排除某些目录：

```
Get-ChildItem -Rec | Where-Object { $_.DirectoryName -notmatch "Debug" }
```

或者列出所有目录：

```
Get-ChildItem | Where-Object { $_.PsIsContainer }
```

Where-Object cmdlet 的语法有时可以非常复杂，对于简单的查询，2.2 节中的 Compare-Property 脚本文件提供了更简单的方法：

```
Get-ChildItem -Rec | Compare-Property DirectoryName notmatch Debug
```

对于很难（或不可能）以编程方式指定的筛选器来说，2.3 节中的 Select-FilteredObject 脚本允许您以交互方式筛选输出。

由于 PowerShell 的管道模式，由 Get-ChildItem 生成的高级文件组自动转换为供其他 cmdlet 操作的高级文件组：

```
PS >Get-ChildItem -Rec | Where-Object { $_.Length -gt 20mb } |
>> Sort-Object -Descending Length | Select-FilteredObject |
>> Remove-Item -WhatIf
>>
What if: Performing operation "Remove File" on Target "C:\temp\backup092300.zip".
What if: Performing operation "Remove File" on Target "C:\temp\sp-tricking_iT2.zip".
What if: Performing operation "Remove File" on Target "C:\temp\slime.mov".
What if: Performing operation "Remove File" on Target "C:\temp\hello-world.mov".
```

有关 Get-ChildItem cmdlet 的更多信息，请键入 **Get-Help Get-ChildItem**。

有关 Where-Object cmdlet 的更多信息，请键入 **Get-Help Where-Object**。

参考

- 2.2 节
- 2.3 节

17.6 管理包含特殊字符的文件

问题

如何在文件名中包含通配符的情况下使用支持通配符的 cmdlet。

解决方案

若要防止 PowerShell 将这些字符视为通配符字符，如果它定义了一个参数，使用 cmdlet 的 `-LiteralPath`（或类似命名的）参数：

```
Get-ChildItem -LiteralPath '[My File].txt'
```

讨论

PowerShell 高级通配符支持下的一种结果可能是用来指定字符范围的方括号有时与实际的文件名冲突。请考虑下面的示例：

```
PS >Get-ChildItem | Select-Object Name
Name
-----
[My File].txt

PS >Get-ChildItem '[My File].txt' | Select-Object Name
PS >Get-ChildItem -LiteralPath '[My File].txt' | Select-Object Name

Name
-----
[My File].txt
```

第一个命令清楚地说明了我们有一个称为 `[My File].txt` 的文件。当我们尝试检索它（将它的名称传递给 `Get-ChildItem` cmdlet）时，我们看到无结果。由于方括号是 PowerShell 中的通配符字符（如 * 和 ?），所以我们提供的文本转换成了一个搜索表达式而不是一个文件名。

`-LiteralPath` 参数（或其他 cmdlet 中的一个类似的命名的参数）会告知 PowerShell 这个文件名被确切地指定，而不是通配符检索项。

除了通配符匹配，文件名有时可能与另一个主题冲突——PowerShell 转义字符。例如，字符 “\” 在 PowerShell 中意味着转义序列的开始，如 “\t”（制表符）、“\n”（换行符）或 “\a”（警报）。为防止 PowerShell 把后面的字符解释为一个转义序列，要使用单引号而不是双引号把字符串括起来。

有关 Get-ChildItem cmdlet 的更多信息，请键入 **Get-Help Get-ChildItem**。

有关 PowerShell 的特殊字符的详细信息，请键入 **Get-Help Get-Help About_Special_Characters**。

17.7 获取磁盘使用情况信息

讨论

在磁盘空间不足的情况下，你会自然而然地要找出清理工作的重点所在。有时，可以通过查找占用空间最多的目录（包括在目录子项）来处理。其他情况下，只需查找它们中包含大文件的目录来解决即可。

例 17-2 收集了这两种类型的检索。它还阐释了如何能有效地使用计算属性。如 Add-Member cmdlet 使你可以通过指定生成数据的表达式来添加属性到输出对象。有关 Add-Member cmdlet 添加自定义成员的详细信息，请参阅 3.11 节。

有关运行脚本的更多信息，请参阅 1.1 节。

例 17-2: Get-DiskUsage.ps1

```
#####
## Get-DiskUsage.ps1
##
## Retrieve information about disk usage in the current directory and all
## subdirectories. If you specify the -IncludeSubdirectories flag, this
## script accounts for the size of subdirectories in the size of a directory.
##
## ie: PS >Get-DiskUsage
##      PS >Get-DiskUsage -IncludeSubdirectories
##
#####
param(
    [switch] $includeSubdirectories
)

## If they specify the -IncludeSubdirectories flag, then we want to account
## for all subdirectories in the size of each directory
if($includeSubdirectories)
{
    Get-ChildItem | Where-Object { $_.PsIsContainer } |
        Select-Object Name,
            @{ Name="Size";
            Expression={ ($_.Get-ChildItem -Recurse |
```

```
        Measure-Object -Sum Length).Sum + 0 } }

## Otherwise, we just find all directories below the current directory,
## and determine their size
else
{
    Get-ChildItem -Recurse | Where-Object { $_.PsIsContainer } |
        Select-Object FullName,
            @{ Name="Size";
              Expression={ ($_ | Get-ChildItem |
                  Measure-Object -Sum Length).Sum + 0 } }
}
```

有关运行脚本的更多信息，请参考 [1.1 节](#)。

参考

- [1.1 节](#)
- [3.11 节](#)

17.8 确定当前位置

问题

如何在脚本或命令中确定当前位置。

解决方案

若要得到当前位置，使用 `Get-Location` cmdlet。`Get-Location` cmdlet 提供 `Drive` 和 `Path` 作为两个常见的属性：

```
$currentLocation = (Get-Location).Path
```

作为 `(Get-Location).Path` 的一个简单形式，要使用 `$pwd` 自动变量。

讨论

`Get-Location` cmdlet 返回有关当前位置的信息。从它的返回信息中，你可以访问当前的驱动器、提供程序和路径。这个当前位置会影响 PowerShell 命令和从 PowerShell 启动程序。在你与 .NET Framework 进行交互时并不会使用它，但是，如果您要调用一个与文件系统进行交互的 .NET 方法就要明确地提供完整路径：

```
[System.Reflection.Assembly]::LoadFile("d:\documents\path_to_library.dll")
```

如果您确信该文件存在，那么 `Resolve-Path` cmdlet 可以把相对路径转换为绝对路径：

```
$filePath = (Resolve-Path library.dll).Path
```

如果该文件不存在，那么 `Join-Path` cmdlet 与 `Get-Location` cmdlet 结合起来用来指定文件位置：

```
$filePath = Join-Path (Get-Location) library.dll
```

将这两种方法组合是另一种更为高级的功能，但也要求您指定相对位置。它来自 `PowerShell` `$ExecutionContext` 变量中的方法，该方法通常用于提供 cmdlet 和提供者使用的功能：

```
$ExecutionContext.SessionState.Path.  
GetUnresolvedProviderPathFromPSPPath("..\\library.dll")
```

有关 `Get-Location` cmdlet 的更多信息，请键入 **Get-Help Get-Location**。

17.9 监视文件内容变更

问题

你希望当文件内容改变时监视到其改变内容。

解决方案

若要当文件内容改变时监视到其改变内容，要使用 `Get-Content` cmdlet 的 `-Wait` 参数。

```
Get-Content log.txt -Wait
```

讨论

`Get-Content` cmdlet 的 `-Wait` 参数有点像传统的 Unix 命令尾部的 `-follow` 参数。如果你使用 `-Wait` 参数，`Get-Content` cmdlet 会读取文件的内容但不会退出。程序将新的内容追加到文件的末尾时，`Get-Content` cmdlet 返回该内容并继续等待。

注意：与 Unix 尾部命令不同，`Get-Content` cmdlet 不支持从一个文件的末尾开始读取的功能。

如果您需要监视非常大的文件末尾，一个专用的文件监视实用工具是一个有效的选择。

有关 `Get-Content` cmdlet 的更多信息，请键入 **Get-Help Get-Content**。有关 `-Wait` 参数的更多信息，请键入 **Get-Help FileSystem**。

17.10 编程：获取一个文件的 MD5 或 SHA1 哈希值

讨论

文件哈希值提供了一种有效的方式来检查一个文件的损坏或修改。一个数字哈希就像文件的指纹，它甚至可以检测到微小的修改。如果文件的内容改变了，哈希值也会改变。很多联机下载服务提供文件下载页上的文件的哈希值，以便您可以确定在传输中文件是否有损坏（请参阅图 17-1）。

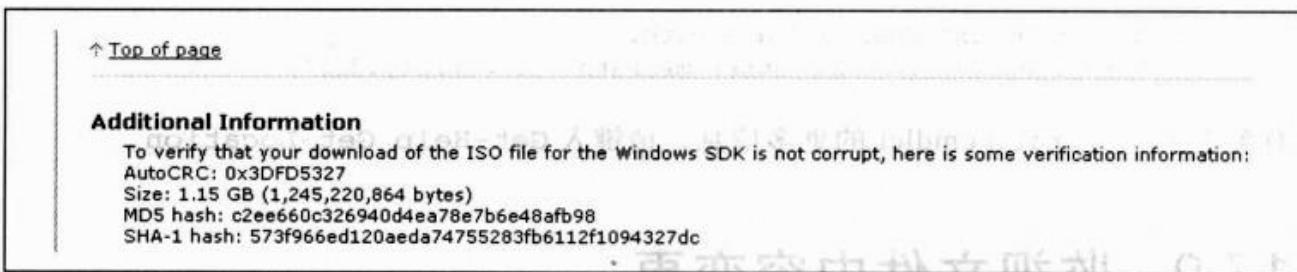


图 17-1：文件哈希值作为验证机制

有三种常见的方法来生成一个文件的哈希值：MD5，SHA1，SHA256。最常用的是 MD5，其次是 SHA1。这些流行的哈希值类型可以被信任用以检测文件的意外修改。如果有人想要篡改该文件而不更改其哈希值，这是无效的。SHA256 算法可用于防止文件不被有意篡改。

例 17-3 允许您确定文件的（或由该管道提供的多个文件）哈希值。

例 17-3：Get-FileHash.ps1

```
#####
## Get-FileHash.ps1
##
## Get the hash of an input file.
##
## ie:
## PS >Get-FileHash myFile.txt
## PS >dir | Get-FileHash
## PS >Get-FileHash myFile.txt -Hash SHA1
## #####
param(
    $path,
    $hashAlgorithm = "MD5"
)
```

```
## Create the hash object that calculates the hash of our file. If they
## provide an invalid hash algorithm, provide an error message.
if($hashAlgorithm -eq "MD5")
{
    $hasher = [System.Security.Cryptography.MD5]::Create()
}
elseif($hashAlgorithm -eq "SHA1")
{
    $hasher = [System.Security.Cryptography.SHA1]::Create()
}
elseif($hashAlgorithm -eq "SHA256")
{
    $hasher = [System.Security.Cryptography.SHA256]::Create()
}
else
{
    $errorMessage = "Hash algorithm $hashAlgorithm is not valid. Valid " +
        "algorithms are MD5, SHA1, and SHA256."
    Write-Error $errorMessage
    return
}

## Create an array to hold the list of files
$files = @()

## If they specified the file name as a parameter, add that to the list
## of files to process
if($path)
{
    $files += $path
}
## Otherwise, take the files that they piped in to the script.
## For each input file, put its full name into the file list
else
{
    $files += @{$input | Foreach-Object { $_.FullName }}
}

## Go through each of the items in the list of input files
foreach($file in $files)
{
    ## Convert it to a fully-qualified path
    $filename = (Resolve-Path $file -ErrorAction SilentlyContinue).Path

    ## If the path does not exist (or is not a file,) just continue
    if((-not $filename) -or (-not (Test-Path $filename -Type Leaf)))
    {
        continue
    }

    ## Use the ComputeHash method from the hash object to calculate
    ## the hash
    $inputStream = New-Object IO.StreamReader $filename
    $hashBytes = $hasher.ComputeHash($inputStream.BaseStream)
    $inputStream.Close()
```

```
## Convert the result to hexadecimal
$builder = New-Object System.Text.StringBuilder
$hashBytes | Foreach-Object { [void] $builder.Append($_.ToString("X2")) }

## Return a custom object with the important details from the
## hashing
$output = New-Object PsObject
$output | Add-Member NoteProperty Path ([IO.Path]::GetFileName($file))
$output | Add-Member NoteProperty HashAlgorithm $hashAlgorithm
$output | Add-Member NoteProperty HashValue ([string] $builder.ToString())
$output
}
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

17.11 建立目录

问题

如何建立目录或文件夹。

解决方案

使用 `md` 或 `mkdir` 命令建立目录：

```
PS >md NewDirectory
```

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

Mode	LastWriteTime	Length	Name
d----	4/29/2007 7:31 PM		NewDirectory

讨论

`md` 和 `mkdir` 功能是复杂的 `New-Item` cmdlet 的简化。可能与您猜的一样，`New-Item` cmdlet 会在您提供的位置创建一个项。它不只对文件系统起作用，支持项概念的任何提供程序也都自动支持此 cmdlet。

有关 `New-Item` cmdlet 的更多信息，请键入 `Get-Help New-Item`。

17.12 删除文件或目录

问题

如何删除文件或目录。

解决方案

使用 `Remove-Item` cmdlet 删除文件或目录：

```
PS >Test-Path NewDirectory
True
PS >Remove-Item NewDirectory
PS >Test-Path NewDirectory
False
```

讨论

`Remove-Item` cmdlet 从您提供的位置删除一个项目。`Remove-Item` cmdlet 不只对文件系统起作用，支持项概念的任何提供程序同样自动支持此 cmdlet。

注意： `Remove-Item` cmdlet 允许您通过其 `Path`、`Include`、`Exclude` 和 `Filter` 参数指定多个文件。有关如何有效地使用这些参数的更多内容，请参阅在本章前面的 17.5 节。

如果该项是一个容器（例如一个目录），那么 PowerShell 会警告您，您的操作也将删除在该容器内的任何内容。如果要防止出现此信息，您可以使用 `-Recurse` 标记。

有关 `Remove-Item` cmdlet 的更多信息，请键入 `Get-Help Remove-Item`。

参考

- 17.5 节

17.13 文件或目录重命名

问题

如何为文件或目录重命名。

解决方案

使用 `Rename-Item` cmdlet 为项目重命名：

```
PS > Rename-Item example.txt example2.txt
```

讨论

使用 `Rename-Item` cmdlet 更改项的名称。这时常见的错误提示：

```
PS > Rename-Item c:\temp\example.txt c:\temp\example2.txt
Rename-Item: 不能重命名，因为指定的目标不是路径。
在 line:1 char:12
+ Rename-Item <<< c:\temp\example.txt c:\temp\example2.txt
```

在这种情况下，PowerShell 提供（不是非常有帮助的）错误消息，因为我们为一个项制定了新的路径，而不只是其名称。

某些解释器允许你在同一时间重命名多个文件。在这些解释器中的命令如下所示：

```
ren *.gif *.jpg
```

PowerShell 不支持此语法，但对其 `-replace` 运算符提供更强功能。作为一个简单的示例，我们可以模拟上面的命令：

```
Get-ChildItem *.gif | Rename-Item -NewName { $_.Name -replace '.gif$', '.jpg' }
```

此语法功能强大。请思考从文件名中删除下划线并用空格替换它们：

```
Get-ChildItem *_* | Rename-Item -NewName { $_.Name -replace '_' ' ' }
```

或重建目录中与命名约定 `Report_Project_Quarter.txt` 相符的文件：

```
PS > Get-ChildItem | Select Name
Name
-----
Report_Project1_Q3.txt
Report_Project1_Q4.txt
Report_Project2_Q1.txt
```

可能要使用的高级样式替换来更改 `Quarter_Project.txt`：

```
PS > Get-ChildItem |
>> Rename-Item -NewName { $_.Name -replace '.*_(.*)_(.*)\.txt', '$2_$1.txt' }
>>
PS > Get-ChildItem | Select Name
```

```
Name
-----
Q1_Project2.txt
Q3_Project1.txt
Q4_Project1.txt
```

有关 `-replace` 运算符的详细信息，请参阅 5.8 节。

与其他 `*-Item` cmdlet 一样，`Rename-Item` 不只对文件系统起作用。支持项概念的任何提供程序同样自动支持此 cmdlet。有关 `Rename-Item` cmdlet 的更多信息，请键入 `Get-Help Rename-Item`。

参考

- 5.8 节

17.14 移动文件或目录

问题

如何移动文件或目录。

解决方案

使用 `Move-Item` cmdlet 移动文件或目录，如下：

```
PS >Move-Item example.txt c:\temp\example2.txt
```

讨论

`Move-Item` cmdlet 将项从一个位置移动到另一个位置。像其他 `*-Item` cmdlet 一样，`Move-Item` 不只对文件系统起作用，支持项概念的任何提供程序同样自动支持此 cmdlet。

注意： `Move-Item` cmdlet 允许您通过其 `Path`、`Include`、`Exclude` 和 `Filter` 参数指定多个文件。有关如何有效地使用这些参数的更多信息，请参阅 17.5 节。

尽管在每个提供程序中都引进了 `Move-Item` cmdlet，但不能在提供程序之间移动项。有关 `Move-Item` cmdlet 的更多信息，请键入 `Get-Help Move-Item`。

参考

- 17.5 节

17.15 获取文件或目录的访问控制列表

问题

如何获取文件或目录的访问控制列表（Access Control List, ACL）。

解决方案

使用 Get-Acl cmdlet 获取文件的控制列表：

```
PS >Get-Acl example.txt

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

Path          Owner          Access
----          ----          -----
example.txt   LEE-DESK\Lee  BUILTIN\Administrator...
```

讨论

Get-Acl cmdlet 检索一个项的安全描述。但此 cmdlet 不是只对文件系统工作，支持安全描述概念的任何提供程序（例如注册表提供程序）也支持 Get-Acl cmdlet。

Get-Acl cmdlet 返回一个对象，该对象代表基于所在特定提供程序的项的安全描述。在文件系统中，它返回一个.NET System.Security.AccessControl.FileSecurity 对象，您可以通过这个对象了解有关的更多信息。例如，例 17-4 通过确保每个文件包含一个完全控制“访问控制列表”的管理员来搜索目录中的可能的访问控制配置错误。

例 17-4：Get-AclMisconfiguration.ps1

```
#####
## Get-AclMisconfiguration.ps1
##
## Demonstration of functionality exposed by the Get-Acl cmdlet. This script
## goes through all access rules in all files in the current directory, and
## ensures that the Administrator group has full control of that file.
##
```

```
## Get all files in the current directory
foreach($file in Get-ChildItem)
{
    ## Retrieve the ACL from the current file
    $acl = Get-Acl $file
    if(-not $acl)
    {
        continue
    }

    $foundAdministratorAcl = $false

    ## Go through each access rule in that ACL
    foreach($accessRule in $acl.Access)
    {
        ## If we find the Administrator, Full Control access rule,
        ## then set the $foundAdministratorAcl variable
        if(($accessRule.IdentityReference -like "*Administrator*") -and
            ($accessRule.FileSystemRights -eq "FullControl"))
        {
            $foundAdministratorAcl = $true
        }
    }

    ## If we didn't find the administrator ACL, output a message
    if(-not $foundAdministratorAcl)
    {
        "Found possible ACL Misconfiguration: $file"
    }
}
```

有关Get-Acl命令的详细信息，请键入**Get-Help Get-Acl**。有关在.NET Framework中使用类的详细信息，请参见3.4节。有关运行脚本的更多信息，请参阅1.1节。

参考

- 1.1节
- 3.4节

17.16 设置文件或目录的访问控制列表

问题

你希望去改变文件与目录的访问控制列表。

解决方案

使用 Set-Acl cmdlet 改变一个文件的访问控制列表。本例阻止来宾账户访问一个文件：

```
$acl = Get-Acl example.txt
$arguments = "LEE-DESK\Guest", "FullControl", "Deny"
$accessRule =
    New-Object System.Security.AccessControl.FileSystemAccessRule $arguments
$acl.SetAccessRule($accessRule)
$acl | Set-Acl example.txt
```

讨论

Set-Acl cmdlet 设置项的安全策略。但这个 cmdlet 不仅只应用于文件系统。任何支持安全策略的提供程序（例如注册表提供程序）也支持 Set-Acl cmdlet。

Set-Acl cmdlet 要求您提供它的访问控制列表应用于该项。虽然可以从头开始构建访问控制列表，但它通常简便的方法是预先从项中检索它（如该解决方案中所示）。若要检索访问控制列表，要使用 Get-Acl cmdlet。一旦您已经修改了访问控制列表的访问控制规则，只需应用到 Set-Acl cmdlet 管道，以使它们成为永久的即可。

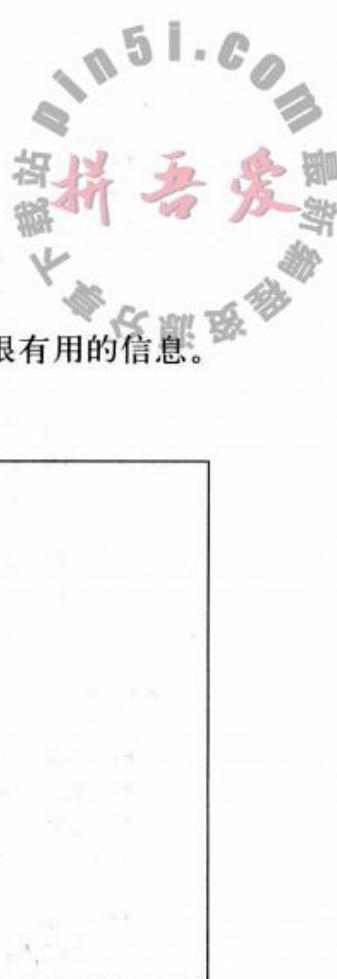
在此解决方案中，我们提供的 FileSystemAccessRule 构造函数 \$arguments 列表显式，设置了 LEE-DESK 计算机的来宾账户上的拒绝规则来阻止全部控制权限。有关使用 .NET Framework 中类（如 FileSystemAccessRule 类）的详细信息，请参阅 3.4 节。

尽管 Set-Acl 命令功能强大，但您可能也已经熟悉过类似的提供功能（如 cacls.exe）的命令行工具。尽管这些工具通常无法工作在注册表（或其他支持 PowerShell 安全描述的提供程序）中，你仍然可以继续在 PowerShell 中使用这些工具。

有关 Set-Acl cmdlet 的更多信息，请键入 **Get-Help Set-Acl**。有关 Get-Acl cmdlet 的详细信息，请参阅 17.15 节。

参考

- 3.4 节
- 17.15 节



17.17 将扩展的文件属性添加到文件

讨论

在资源管理中单击文件，浏览器在右键的属性对话框中提供了有关文件的很有用的信息。这包括创作信息、图像信息、音乐信息等等（见图 17-2）。

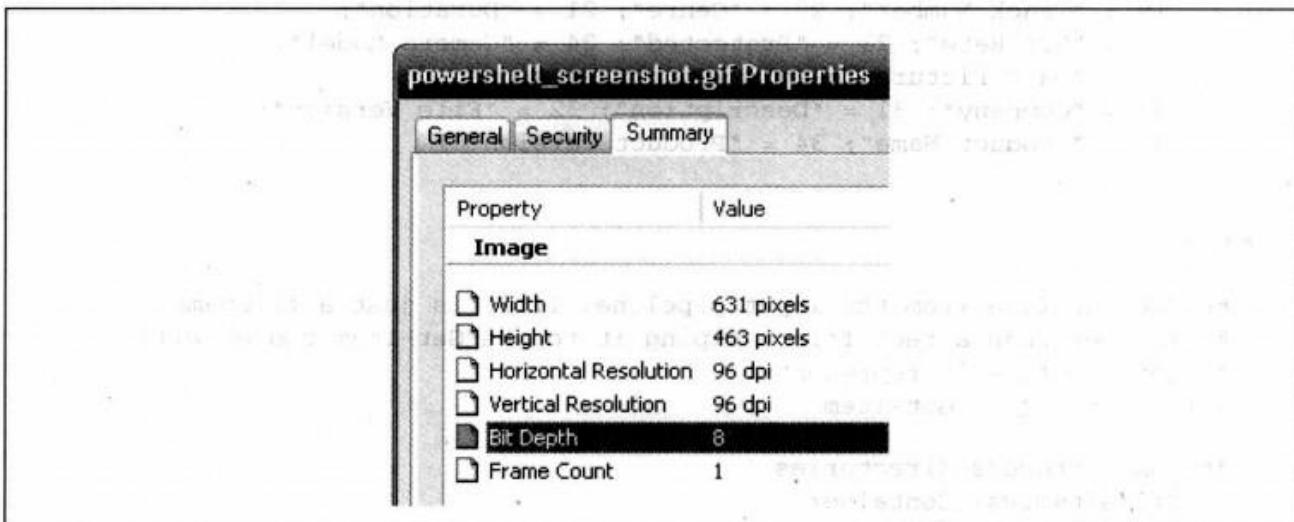


图 17-2：在 Windows 资源管理器中展开的文件属性

默认情况下 PowerShell 不公开此信息，但可以从 `Shell.Application` COM 对象获取这些属性。例 17-5 就是这样，并将此扩展的信息添加为由 `Get-ChildItem` cmdlet 返回的文件的属性。

例 17-5：Add-ExtendedFileProperties.ps1

```
#####
## Add-ExtendedFileProperties.ps1
## Add the extended file properties normally shown in Explorer's
## "File Properties" tab.
## ie:
## PS >Get-ChildItem | Add-ExtendedFileProperties.ps1 |
## Format-Table Name, "Bit Rate"
##
#####
begin
{
    ## Create the Shell.Application COM object that provides this
    ## functionality
    $shellObject = New-Object -Com Shell.Application
```

```

## Store the property names and identifiers for all of the shell
## properties
$itemProperties = @{
    1 = "Size"; 2 = "Type"; 3 = "Date Modified";
    4 = "Date Created"; 5 = "Date Accessed";
    7 = "Status"; 8 = "Owner";
    9 = "Author"; 10 = "Title"; 11 = "Subject";
    12 = "Category"; 13 = "Pages"; 14 = "Comments";
    15 = "Copyright"; 16 = "Artist"; 17 = "Album Title";
    19 = "Track Number"; 20 = "Genre"; 21 = "Duration";
    22 = "Bit Rate"; 23 = "Protected"; 24 = "Camera Model";
    25 = "Date Picture Taken"; 26 = "Dimensions";
    30 = "Company"; 31 = "Description"; 32 = "File Version";
    33 = "Product Name"; 34 = "Product Version" }
}

process
{
    ## Get the file from the input pipeline. If it is just a filename
    ## (rather than a real file,) piping it to the Get-Item cmdlet will
    ## get the file it represents.
    $fileItem = $_ | Get-Item

    ## Don't process directories
    if($fileItem.PSIsContainer)
    {
        $fileItem
        return
    }

    ## Extract the file name and directory name
    $directoryName = $fileItem.DirectoryName
    $filename = $fileItem.Name

    ## Create the folder object and shell item from the COM object
    $folderObject = $shellObject.NameSpace($directoryName)
    $item = $folderObject.ParseName($filename)

    ## Now, go through each property and add its information as a
    ## property to the file we are about to return
    foreach($itemProperty in $itemProperties.Keys)
    {
        $fileItem | Add-Member NoteProperty $itemProperties[$itemProperty]
        $folderObject.GetDetailsOf($item, $itemProperty)
    }

    ## Finally, return the file with the extra shell information
    $fileItem
}

```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

17.18 编程：创建文件系统硬链接

讨论

有时由两个不同的名称或位置指向同一个文件是非常有用的。您不能通过复制该项目来解决此问题，因为对一个文件的修改并不会自动影响另一个文件。

它的解决方案被称为硬链接（*hardlink*），一个指向另一个文件的数据的新名称的项。Windows 操作系统支持硬链接，但只有 Windows Vista 包括了一个用以创建它们的实用工具。

例 17-6 允许您创建硬链接，而不必安装其他工具。它使用（和要求）15.9 节提供的 *Invoke-WindowsApi.ps1* 脚本。

例 17-6：New-FilesystemHardLink.ps1

```
#####
## New-FilesystemHardLink.ps1
##
## Create a new hard link, which allows you to create a new name by which you
## can access an existing file. Windows only deletes the actual file once
## you delete all hard links that point to it.
##
## ie:
##
## PS >"Hello" > test.txt
## PS >dir test* | select name
##
## Name
## ----
## test.txt
##
## PS >New-FilesystemHardLink.ps1 test2.txt test.txt
## PS >type test2.txt
## Hello
## PS >dir test* | select name
## Name
## ----
## test.txt
## test2.txt
##
```

```
param(
    ## The new filename you want to create
    [string] $filename,
    ## The existing file that you want the new name to point to
    [string] $existingFilename
)
## Ensure that the provided names are absolute paths
$filename = $ExecutionContext.SessionState.Path.`
    GetUnresolvedProviderPathFromPSPath($filename)
$existingFilename = Resolve-Path $existingFilename

## Prepare the parameter types and parameters for the CreateHardLink function
$parameterTypes = [string], [string], [IntPtr]
$parameters = [string] $filename, [string] $existingFilename, [IntPtr]::Zero

## Call the CreateHardLink method in the Kernel32 DLL
$currentDirectory = Split-Path $myInvocation.MyCommand.Path
$invokeWindowsApiCommand = Join-Path $currentDirectory Invoke-WindowsApi.ps1
$result = & $invokeWindowsApiCommand "kernel32" `

    ([bool]) "CreateHardLink" $parameterTypes $parameters

## Provide an error message if the call fails
if(-not $result)
{
    $message = "Could not create hard link of $filename to " +
        "existing file $existingFilename"
    Write-Error $message
}
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节
- 15.9 节

17.19 编程：创建 ZIP 文档

讨论

传输或存档文件时打包文档是非常有用的。ZIP 文档是最常见的文档类型，因此如果有一个脚本来帮助管理它们会很有用。

传统的命令行 ZIP 存档工具可以满足您的需要。但如果它们不支持详细信息提供或与管理任务的交互，一个能提供更多功能的编程方式便极具吸引力了。

例 17-7 通过文件管道创建 ZIP 存档。它要求您安装 SharpZipLib，可以从 <http://www.icsharpcode.net/OpenSource/SharpZipLib/> 下载。

例 17-7: New-ZipFile.ps1

```
#####
## New-ZipFile.ps1
##
## Create a Zip file from any files piped in. Requires that
## you have the SharpZipLib installed, which is available from
## http://www.icsharpcode.net/OpenSource/SharpZipLib/
##
## ie:
##
## PS >dir *.ps1 | New-ZipFile scripts.zip d:\bin\ICSharpCode.SharpZipLib.dll
## PS >"readme.txt" | New-ZipFile docs.zip d:\bin\ICSharpCode.SharpZipLib.dll
##
#####

param(
    $zipName = $(throw "Please specify the name of the file to create."),
    $libPath = $(throw "Please specify the path to ICSharpCode.SharpZipLib.dll.")
)

## Load the Zip library
[void] [Reflection.Assembly]::LoadFile($libPath)
$namespace = "ICSharpCode.SharpZipLib.Zip.{0}"

## Create the Zip File
$zipName =
    $ExecutionContext.SessionState.Path.GetUnresolvedProviderPathFromPSPPath($zipName)
$zipFile = New-Object ($namespace -f "ZipOutputStream") ([IO.File]::Create($zipName))
$zipFullName = (Resolve-Path $zipName).Path

[byte[]] $buffer = New-Object byte[] 4096

## Go through each file in the input, adding it to the Zip file
## specified
foreach($file in $input)
{
    ## Skip the current file if it is the zip file itself
    if($file.FullName -eq $zipFullName)
    {
        continue
    }

    ## Convert the path to a relative path, if it is under the current location
    $replacePath = [Regex]::Escape( (Get-Location).Path + "\\" )
    $zipName = ([string] $file) -replace $replacePath,""

    ## Create the zip entry, and add it to the file
    $zipEntry = New-Object ($namespace -f "ZipEntry") $zipName
    $zipFile.PutNextEntry($zipEntry)
```

```
$fileStream = [IO.File]::OpenRead($file.FullName)
[ICSharpCode.SharpZipLib.Core.StreamUtils]::Copy($fileStream, $zipFile, $buffer)
$fileStream.Close()
}

## Close the file
$zipFile.Close()
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

第 18 章

Windows 注册表

18.0 简介

作为绝大多数应用程序的配置存储，注册表在系统管理中扮演了一个重要角色，同样也难以掌握。尽管命令行工具（例如 `reg.exe`）可以帮助您处理注册表，但它们的接口通常不一致而且混乱。尽管注册表编辑器图形用户界面使用起来很方便，但它不支持脚本的管理。

PowerShell通过将Windows注册表发布为一个导航提供程序来解决这个问题—这与您使用文件系统进行浏览和管理数据源是完全相同的方法。

18.1 注册表导航

问题

如何导览 Windows 注册表。

解决方案

使用 Set-Location 可以使你像使用文件系统一样来浏览注册表：

```
PS >Set-Location HKCU:  
PS >Set-Location \Software\Microsoft\Windows\CurrentVersion\Run  
PS >Get-Location  
  
Path  
----  
HKCU:\Software\Microsoft\Windows\CurrentVersion\Run
```

讨论

PowerShell 使你能够以与浏览文件系统、驱动器和其他基于导航的提供程序中完全相同的方式来浏览 Windows 注册表。像这些其他提供程序一样，注册表提供程序支持 Set-Location cmdlet（带有 sl、cd 和 chdir 的标准命令）、Push-Location（标准命令入栈 pushd）、Pop-Location（标准命令出栈 popd）等。

有关当您选定一个注册表项时如何更改注册表键值的信息，请参阅 18.3 节。有关注册表提供程序的详细信息，请键入 **Get-Help Registry**。

参考

- ### • 18.3 节

18.2 查看一个注册表项

问题

如何查看某个特定的注册表项的值。

解决方案

若要检索某个注册表项的值，可以使用 `Get-ItemProperty` cmdlet，如例 18-1 所示。

例 18-1：检索注册表项的属性

```
PS >Set-Location HKCU:  
PS >Set-Location \Software\Microsoft\Windows\CurrentVersion\Run  
PS >Get-ItemProperty
```

```
PSPATH : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run  
PSPARENTPATH : Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion  
PSCHILDNAME : Run  
PSDRIVE : HKCU  
PSPROVIDER : Microsoft.PowerShell.Core\Registry  
FOLDERSHARE : "C:\Program Files\FolderShare\FolderShare.exe" /background  
TASKSWITCHXP : d:\lee\tools\TaskSwitchXP.exe  
CTFMON.EXE : C:\WINDOWS\system32\ctfmon.exe  
DITTO : C:\Program Files\itto\itto.exe  
QUICKTIME TASK : "C:\Program Files\QuickTime Alternative\qttask.exe"  
` -atboottime  
H/PC CONNECTION AGENT : "C:\Program Files\Microsoft ActiveSync\wcescomm.exe"
```

讨论

在注册表提供程序中，PowerShell 将注册表项视为项以及为这些项的属性值。若要获取项的属性，可以使用 `Get-ItemProperty` cmdlet。`Get-ItemProperty` cmdlet 具有标准别名 `gp`。

例 18-1 列出了与该特定的键关联的所有属性值。若要检索特定项的值，访问它就像访问一个 .NET 对象，或访问 PowerShell 中的其他任何地方的属性时一样：

```
PS >$item = Get-ItemProperty .  
PS >$item.TaskSwitchXp  
d:\lee\tools\TaskSwitchXP.exe
```

如果想一次完成，则执行下面命令

```
PS >$runKey = "HKCU:\Software\Microsoft\Windows\CurrentVersion\Run"
PS >(Get-ItemProperty $runKey).TaskSwitchXp
d:\lee\tools\TaskSwitchXP.exe
```

有关**Get-ItemProperty** cmdlet 的更多信息，请键入 **Get-Help Get-ItemProperty**。
有关注册表提供程序的更多信息，请键入 **Get-Help Registry**。

18.3 修改或删除一个注册表键值

问题

如何修改或删除特定的注册表键的属性。

解决方案

若要设置一个注册表键值，可以使用 `Set-ItemProperty` cmdlet：

```
PS >(Get-ItemProperty .).MyProgram  
c:\temp\MyProgram.exe  
PS >Set-ItemProperty . MyProgram d:\Lee\tools\MyProgram.exe  
PS >(Get-ItemProperty .).MyProgram  
d:\Lee\tools\MyProgram.exe
```

若要移除一个注册表键值，可以使用 Remove-ItemProperty cmdlet：

```
PS >Remove-ItemProperty . MyProgram  
PS >(Get-ItemProperty .).MyProgram
```

讨论

在注册表提供程序中，PowerShell 将注册表项视为项以及这些项的属性值。若要更改一个键属性的值，可以使用 Set-ItemProperty cmdlet。Set-ItemProperty cmdlet 具有标准别名 sp。若要完全移除一个键属性，使用 Remove-ItemProperty cmdlet。

注意：一般更改注册表中的信息要谨慎。误删或误改项很容易使你的系统无法启动。

键入 **Get-Help Get-ItemProperty** 获取有关 Get-ItemProperty cmdlet 的更多信息。分别键入 **Get-Help Set-ItemProperty** 或 **Get-Help Remove-ItemProperty** 获取有关 Set-ItemProperty 和 Remove-ItemProperty cmdlet 信息。有关注册表提供程序的详细信息，请键入 **Get-Help Registry**。

18.4 创建一个注册表键值

问题

你想给现有的注册表项添加新键值。

解决方案

若要给注册表键添加一个值，可以使用 New-ItemProperty cmdlet。例 18-2 将 MyProgram.exe 添加到当前用户在登录时启动的程序列表中。

例 18-2：给注册表键创建新的属性

```
PS >New-ItemProperty . -Name MyProgram -Value c:\temp\MyProgram.exe
PSPath          : Microsoft.PowerShell.Core\Registry:::HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
PSParentPath    : Microsoft.PowerShell.Core\Registry:::HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion
PSChildName     : Run
PSDrive         : HKCU
PSProvider      : Microsoft.PowerShell.Core\Registry
MyProgram       : c:\temp\MyProgram.exe

PS >Get-ItemProperty .

PSPath          : Microsoft.PowerShell.Core\Registry:::HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
PSParentPath    : Microsoft.PowerShell.Core\Registry:::HKEY_CURRENT_USER
```

```
SER\Software\Microsoft\Windows\CurrentVersion
PSChildName      : Run
PSDrive          : HKCU
PSPrinter        : Microsoft.PowerShell.Core\Registry
FolderShare      : "C:\Program Files\FolderShare\FolderShare.exe" /ba
                  ckground
TaskSwitchXP     : d:\lee\tools\TaskSwitchXP.exe
ctfmon.exe       : C:\WINDOWS\system32\ctfmon.exe
Ditto            : C:\Program Files\itto\itto.exe
QuickTime Task   : "C:\Program Files\QuickTime Alternative\qtask.exe"
                  " -atboottime
H/PC Connection Agent : "C:\Program Files\Microsoft ActiveSync\wcescomm.ex
e"
MyProgram        : c:\temp\MyProgram.exe
```

讨论

在注册表提供程序中，PowerShell 将注册表项视为项以及这些项的属性项值。若要创建一个键属性，请使用 `New-ItemProperty` cmdlet。有关 `New-ItemProperty` cmdlet 的更多信息，请键入 `Get-Help New-ItemProperty`。有关注册表提供程序的详细信息，请键入 `Get-Help Registry`。

18.5 删 除注册表项

问题

你想删除注册表键及其属性。

解决方案

若要移除一个注册表键，可以使用 `Remove-Item` cmdlet：

```
PS >dir
Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\
Microsoft\Windows\CurrentVersion\Run
SKC VC Name           Property
--- -- -----
0 0 Spyware          {}
```

```
PS >Remove-Item Spyware
```

讨论

如 18.4 节中所述，注册表提供程序允许您使用 Remove-Item cmdlet 删除项和容器。 Remove-Item cmdlet 具有标准命令 rm、 rmdir、 del、 erase 和 rd。

注意：更改注册表中的信息时始终都会有警告。误删或误改项很容易使你的系统无法启动。

与文件系统一样，Remove-Item cmdlet 可以通过其 Path、Include、Exclude 和 Filter 参数来指定多个项。有关如何有效地使用这些参数的信息，请参阅 17.5 节。

有关 Remove-Item cmdlet 的更多信息，请键入 **Get-Help Remove-Item**。有关注册表提供程序的详细信息，请键入 **Get-Help Registry**。

参考

- 17.5 节：有关将文件夹添加到注册表的详细信息。
- 18.4 节

18.6 将站点添加到 IE 浏览器的安全域中

问题

你要将站点添加到一个特定的 IE 浏览器的安全域中。

解决方案

若要将站点添加到特定的安全域中所需的注册表项和属性分别是 New-Item 和 New-ItemProperty cmdlet。例 18-3 将 *www.example.com* 添加到 IE 浏览器的受信任站点列表中。

例 18-3：将 *www.example.com* 添加到 IE 浏览器的受信任站点列表中

```
Set-Location "HKCU:\Software\Microsoft\Windows\CurrentVersion\Internet
Settings"
Set-Location ZoneMap\Domains
New-Item example.com
Set-Location example.com
New-Item www
Set-Location www
New-ItemProperty . -Name http -Value 2 -Type DWORD
```

讨论

IE 浏览器为不同的安全区添加和删除站点，需要在注册表中修改相应的数据。

IE 浏览器将它的区域映射信息存储在在注册表 `HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\Domains` 中。在这个键下，资源管理器存储域名（如 `leeholmes.com`）与主机名（如 `www`）作为某个键的子项（请参阅图 18-1）。在子项中，资源管理器使用 DWORD 值来存储对应于区域的标识符的属性（如 `http`）。

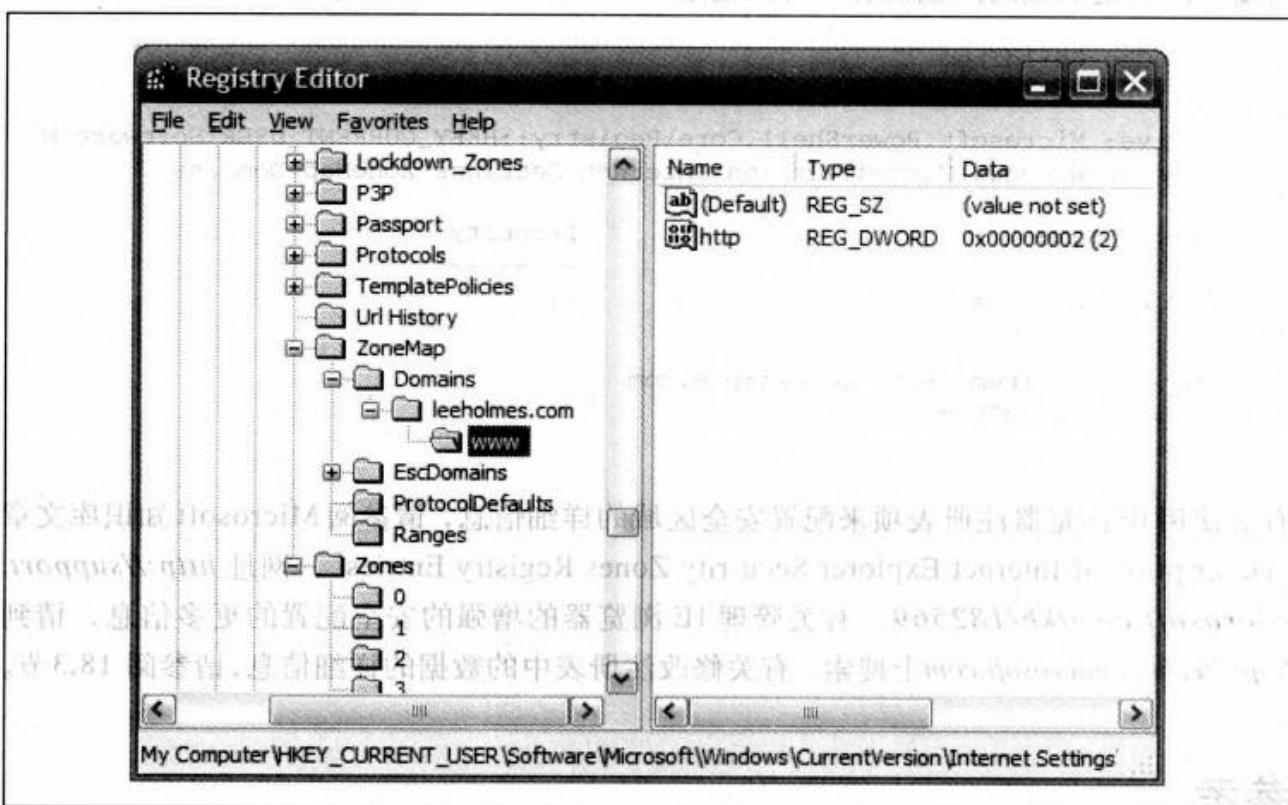


图 18-1：IE 浏览器区域的配置

IE 浏览器区域标识符是：

0. 我的电脑
1. 本地 Intranet
2. 受信任的站点
3. Intranet
4. 受限制的站点

如果 IE 浏览器被配置成增强的安全配置模式，您还必须修改 `EscDomains` 键下的项。

注意：一旦一台计算机已启用IE浏览器的增强的安全配置，那么即使在删除增强的安全配置后这些设置依旧会保持。下面的命令允许您的计算机重新信任UNC路径：

```
Set-Location "HKCU:\Software\Microsoft\Windows\"  
Set-Location "CurrentVersion"  
Set-Location "Internet Settings"  
Set-ItemProperty ZoneMap UNCAsIntranet -Type DWORD 1  
Set-ItemProperty ZoneMap IntranetName -Type DWORD 1
```

若要移除特定的域的区域映射，可以使用 Remove-Item cmdlet：

```
PS >Get-ChildItem
```

```
Hive: Microsoft.PowerShell.Core\Registry::HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\Domains
```

SKC	VC	Name
1	0	example.com

Property
{}

```
PS >Remove-Item -Recurse example.com  
PS >Get-ChildItem  
PS >
```

有关使用IE浏览器注册表项来配置安全区域的详细信息，请参阅 Microsoft 知识库文章“Description of Internet Explorer Security Zones Registry Entries”，网址 <http://support.microsoft.com/kb/182569>。有关管理 IE 浏览器的增强的安全配置的更多信息，请到 <http://technet.microsoft.com> 上搜索。有关修改注册表中的数据的详细信息，请参阅 18.3 节。

参考

- 18.3 节

18.7 修改 IE 浏览器设置

问题

你想要修改 IE 浏览器配置选项。

解决方案

若要修改 IE 浏览器配置的注册表项，可以使用 Set-ItemProperty cmdlet。例如，要修改代理服务器：

```
Set-Location "HKCU:\Software\Microsoft\Windows\CurrentVersion\Internet Settings"
Set-ItemProperty . -Name ProxyServer -Value http://proxy.example.com
Set-ItemProperty . -Name ProxyEnable -Value 1
```

讨论

IE浏览器将它的主要配置信息作为属性存储在该注册表项HKCU:\Software\Microsoft\Windows\CurrentVersion\Internet Settings上。若要更改这些属性，可以使用Set-ItemProperty cmdlet，如解决方案中所示。

要调整的另一组常见属性是定义安全区域的配置参数。一个示例是为了防止在受限制的站点区域中运行脚本。对每个区域，IE 浏览器都将此信息作为属性存储在注册表项HKCU:\Software\Microsoft\Windows\CurrentVersion\Internet Settings\Zones\<Zone>中，其中 <Zone> 表示用来管理区域的标识符(0、1、2、3 或 4)。

IE 浏览器区域标识符是：

0. 我的电脑
1. 本地 Internet
2. 信任的站点
3. Internet
4. 受限制的站点

这个键的属性名并不是面向用户设计的，如1A04 和 1809 这样。虽然是不好理解的命名，但你编写脚本时仍可以使用它们。

有关使用Internet Explorer的注册表设置配置安全区域的详细信息，请参阅 Microsoft 知识库文章“Internet Explorer 安全区域注册表项的描述”，网址 <http://support.microsoft.com/kb/182569>。

有关修改注册表中的数据的详细信息，请参阅 18.3 节。

参考

- 18.3 节

18.8 编程：搜索 Windows 注册表

讨论

虽然 Windows 注册表编辑器提供了方便的搜索功能，但它有时可能提供不了你所需要的更多功能。例如，注册表编辑器不支持使用通配符或正则表达式搜索。

在文件系统中，我们可以使用 `Select-String` cmdlet 来搜索文件内容。虽然 PowerShell 没有为其他存储设备提供这种功能，但我们可以编写脚本来执行该操作。其主旨是像过滤文件内容一样过滤注册表键值：

- 目录有项，项有内容。
- 注册表项有属性，属性具有值。

例 18-4 遍历所有注册表项（和它们的值）并返回与搜索条件最匹配的信息。

例 18-4：Search-Registry.ps1

```
#####
## Search-Registry.ps1
##
## Search the registry for keys or properties that match a specific value.
##
## ie:
## PS >Set-Location HKCU:\Software\Microsoft\
## PS >Search-Registry Run
##
#####
param([string] $searchText = ${throw "Please specify text to search for."})

## Helper function to create a new object that represents
## a registry match from this script
function New-RegistryMatch
{
    param( $matchType, $keyName, $propertyName, $line )

    $registryMatch = New-Object PsObject
    $registryMatch | Add-Member NoteProperty MatchType $matchType
    $registryMatch | Add-Member NoteProperty KeyName $keyName
    $registryMatch | Add-Member NoteProperty PropertyName $propertyName
    $registryMatch | Add-Member NoteProperty Line $line

    $registryMatch
}

## Go through each item in the registry
foreach($item in Get-ChildItem -Recurse -ErrorAction SilentlyContinue)
```

```
    ## Check if the key name matches
    if($item.Name -match $searchText)
    {
        New-RegistryMatch "Key" $item.Name $null $item.Name
    }

    ## Check if a key property matches
    foreach($property in (Get-ItemProperty $item.PsPath).PsObject.Properties)
    {
        ## Skip the property if it was one PowerShell added
        if(($property.Name -eq "PSPATH") -or ($property.Name -eq "PSChildName"))
        {
            continue
        }

        ## Search the text of the property
        $propertyText = "$($property.Name)=$($property.Value)"
        if($propertyText -match $searchText)
        {
            New-RegistryMatch "Property" $item.Name $property.Name $propertyText
        }
    }
}
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

18.9 获取某个注册表项的访问控制列表

问题

如何检索一个注册表项的访问控制列表。

解决方案

若要检索一个注册表项的 ACL，可以使用 Get-Acl cmdlet 代码：

```
PS >Get-Acl HKLM:\Software

Path          Owner          Access
----          -----          -----
Microsoft.PowerShell.... BUILTIN\Administrators  CREATOR OWNER Allow ...
```

讨论

正如17.15节中所述的Get-Acl cmdlet检索某项的安全描述一样，此cmdlet不只针对注册表。支持安全描述概念的任何提供程序（例如文件系统提供程序）也支持Get-Acl cmdlet。Get-Acl cmdlet返回一个代表该项的安全描述的对象，并且是基于包含该项的提供程序。在注册表提供程序中，此方法返回一个.NET System.Security.AccessControl.RegistrySecurity对象，您可以了解有关的更多信息。例如，在脚本中使用访问控制列表，请参阅17.15节。

有关Get-Acl命令的详细信息，请键入**Get-Help Get-Acl**。有关.NET Framework中使用类的详细信息，请参阅3.4节。

参考

- 3.4节
- 17.15节

18.10 设置一个注册表项的访问控制列表

问题

你想更改某个注册表项的访问控制列表。

解决方案

使用Set-Acl cmdlet来设置一个注册表项上的访问控制列表。本示例授予一个账户进行对HKLM:\Software下的注册表项的写访问权限。此功能对程序特别有用，将它写入注册表中的管理员区后，程序便无法在一个非管理员账户下运行。

```
cd HKLM:\Software\MyProgram
$acl = Get-Acl .
$arguments = "LEE-DESK\Lee","FullControl","Allow"
$accessRule = New-Object System.Security.AccessControl.RegistryAccessRule
$arguments
$acl.SetAccessRule($accessRule)
$acl | Set-Acl .
```

讨论

正如17.16节中所述，Set-Acl cmdlet可以设置某项的安全描述。并且此cmdlet不能

只针对注册表。支持安全描述概念的任何提供程序（例如文件系统提供程序）也支持 Set-Acl cmdlet。

Set-Acl cmdlet 要求您提供它的访问控制列表使之应用于该项目。虽然可以从头开始构建访问控制列表，但通常最简便的方法是预先（如该解决方案中所示）在该项目中检索它。若要检索访问控制列表，可以使用 Get-Acl cmdlet。一旦您已经修改访问控制列表上的访问控制规则，只需应用于 Set-Acl cmdlet 的设置来保存你的设置即可。

在解决方案中，我们为 RegistryAccessRule（注册表访问规则）构造函数提供的 \$arguments 参数为 LEE-DESK 计算机的账户 Lee 显式设定了允许完全控制规则来获取许可。有关 .NET Framework 中类（例如 RegistryAccessRule 类）的使用的详细信息，请参阅 3.4 节。

尽管 Set-Acl 命令功能强大，但您可能已经熟悉提供类似功能（如 SubInAcl.exe）的命令行工具了。您当然可以在 PowerShell 中继续使用这些工具。

有关 Set-Acl cmdlet 的更多信息，请键入 **Get-Help Set-Acl**。有关 Get-Acl cmdlet 的更多信息，请参阅 18.9 节。

参考

- 3.4 节
- 17.16 节
- 18.9 节

18.11 使用远程计算机的注册表

问题

你想使用远程计算机的注册表项和值。

解决方案

若要使用远程计算机的注册表，可以使用这一章中提供的脚本：Get-RemoteRegistryChildItem、Get-RemoteRegistryProperty 和 Set-RemoteRegistryProperty。这些脚本要求该远程计算机的远程注册表服务已启用并运行。例 18-5 更新 PowerShell 在远程计算机的执行策略。

例 18-5：设置 PowerShell 在远程计算机的执行策略

```
PS >$registryPath = "HKLM:\Software\Microsoft\PowerShell\1"
PS >Get-RemoteRegistryChildItem LEE-DESK $registryPath

SKC VC Name                               Property
--- -- ----
0   1  1033                                {Install}
0   5  PowerShellEngine                   {ApplicationBase, ConsoleHostAss...
2   0  PowerShellSnapIns                {}
1   0  ShellIds                           {}

PS >Get-RemoteRegistryChildItem LEE-DESK $registryPath\ShellIds

SKC VC Name                               Property
--- -- ----
0   2  Microsoft.PowerShell             {Path, ExecutionPolicy}

PS >
PS >$registryPath = "HKLM:\Software\Microsoft\PowerShell\1\" +
>>      "ShellIds\Microsoft.PowerShell"
>>
PS >Get-RemoteRegistryKeyProperty LEE-DESK $registryPath ExecutionPolicy

ExecutionPolicy
-----
Unrestricted

PS >Set-RemoteRegistryKeyProperty LEE-DESK $registryPath ` 
>>      "ExecutionPolicy" "RemoteSigned"
>>
PS >Get-RemoteRegistryKeyProperty LEE-DESK $registryPath ExecutionPolicy

ExecutionPolicy
-----
RemoteSigned
```

讨论

尽管此特定的任务通过 PowerShell 的组策略支持可能会被更好地解决，但它演示了一个包括远程注册表的浏览和修改的一种有用方案。

有关 Get-RemoteRegistryChildItem、Get-RemoteRegistryProperty 和 Set-RemoteRegistryProperty 脚本的更多信息，请参阅 18.12 节、18.13 节和 18.14 节。

参考

- 18.12 节

- 18.13 节
- 18.14 节

18.12 编程：从远程计算机获取注册表项

讨论

尽管 PowerShell 不直接允许您访问和操纵远程计算机的注册表，但它仍可以通过使用 .NET Framework 来支持这一功能。由于 .NET Framework 提供的功能更面向开发人员，所以我们可以改为使用操作简便的 PowerShell 脚本。

例 18-6 允许您列出在远程注册表项中的子项，这更像是您在本地计算机上执行一样。为了成功执行此脚本，目标计算机必须启用并运行远程注册表服务。

例 18-6: Get-RemoteRegistryChildItem.ps1

```
#####
## Get-RemoteRegistryChildItem.ps1
##
## Get the list of subkeys below a given key.
##
## ie:
##
## PS >Get-RemoteRegistryChildItem LEE-DESK HKLM:\Software
##
#####
param(
    $computer = $(throw "Please specify a computer name."),
    $path = $(throw "Please specify a registry path")
)

## Validate and extract out the registry key
if($path -match "^HKLM:\\(.*)")
{
    $baseKey = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey(
        "LocalMachine", $computer)
}
elseif($path -match "^HKCU:\\(.*)")
{
    $baseKey = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey(
        "CurrentUser", $computer)
}
else
{
    Write-Error ("Please specify a fully-qualified registry path " +
        "(i.e.: HKLM:\Software) of the registry key to open.")
    return
}
```

```
## Open the key
$key = $baseKey.OpenSubKey($matches[1])

## Retrieve all of its children
foreach($subkeyName in $key.GetSubKeyNames())
{
    ## Open the subkey
    $subkey = $key.OpenSubKey($subkeyName)

    ## Add information so that PowerShell displays this key like regular
    ## registry key
    $returnObject = [PsObject] $subkey
    $returnObject | Add-Member NoteProperty PsChildName $subkeyName
    $returnObject | Add-Member NoteProperty Property $subkey.GetValueNames()

    ## Output the key
    $returnObject

    ## Close the child key
    $subkey.Close()
}

## Close the key and base keys
$key.Close()
$baseKey.Close()
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

18.13 编程：获取远程注册表项的属性

讨论

尽管 PowerShell 不直接允许您访问和操纵远程计算机的注册表，但它仍可以通过使用 .NET Framework 来支持这一功能。由于 .NET Framework 提供的功能更侧重于面向开发人员，所以我们可以改为使用操作简便的 PowerShell 脚本。

例 18-7 可以从给定的远程注册表项中获取该属性（或特定的属性）。为了成功执行此脚本，目标计算机必须启用并运行远程注册表服务。

例 18-7：Get-RemoteRegistryKeyProperty.ps1

```
#####
## Get-RemoteRegistryKeyProperty.ps1
##
```

```
## Get the value of a remote registry key property
##
## ie:
##
## PS >$registryPath =
##      "HKLM:\software\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell"
## PS >Get-RemoteRegistryKeyProperty LEE-DESK $registryPath "ExecutionPolicy"
##
#####
param(
    $computer = $(throw "Please specify a computer name."),
    $path = $(throw "Please specify a registry path"),
    $property = "*"
)
## Validate and extract out the registry key
if($path -match "^HKLM:\\(.*)")
{
    $baseKey = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey(
        "LocalMachine", $computer)
}
elseif($path -match "^HKCU:\\(.*)")
{
    $baseKey = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey(
        "CurrentUser", $computer)
}
else
{
    Write-Error ("Please specify a fully-qualified registry path " +
        "(i.e.: HKLM:\Software) of the registry key to open.")
    return
}

## Open the key
$key = $baseKey.OpenSubKey($matches[1])
$returnObject = New-Object PsObject

## Go through each of the properties in the key
foreach($keyProperty in $key.GetValueNames())
{
    ## If the property matches the search term, add it as a
    ## property to the output
    if($keyProperty -like $property)
    {
        $returnObject |
            Add-Member NoteProperty $keyProperty $key.GetValue($keyProperty)
    }
}

## Return the resulting object
$returnObject

## Close the key and base keys
$key.Close()
$baseKey.Close()
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

18.14 编程：设置远程注册表项的属性

讨论

尽管 PowerShell 不直接允许您访问和操纵远程计算机的注册表，但它仍可以通过使用 .NET Framework 来支持这一功能。由于 .NET Framework 提供的功能更面向开发人员，所以我们可以改为使用操作简便的 PowerShell 脚本。

例 18-8 允许您将一个给定的远程注册表项上设置一个属性的值。为了成功执行此脚本，目标计算机必须启用并运行远程注册表服务。

例 18-8：Set-RemoteRegistryKeyProperty.ps1

```
#####
## Set-RemoteRegistryKeyProperty.ps1
##
## Set the value of a remote registry key property
##
## ie:
##
## PS >$registryPath =
##     "HKLM:\software\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell"
## PS >Set-RemoteRegistryKeyProperty LEE-DESK $registryPath ` 
##     "ExecutionPolicy" "RemoteSigned"
##
#####
#
param(
    $computer = $(throw "Please specify a computer name."),
    $path = $(throw "Please specify a registry path"),
    $property = $(throw "Please specify a property name"),
    $PropertyValue = $(throw "Please specify a property value")
)

## Validate and extract out the registry key
if($path -match "^HKLM:\\(.*)")
{
    $baseKey = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey
        ("LocalMachine", $computer)
}
elseif($path -match "^HKCU:\\(.*)")
{
    $baseKey = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey
        ("CurrentUser", $computer)
}
```

```
$baseKey = [Microsoft.Win32.RegistryKey]::OpenRemoteBaseKey
    ("CurrentUser", $computer)
}
else
{
    Write-Error ("Please specify a fully-qualified registry path " +
        "(i.e.: HKLM:\Software) of the registry key to open.")
    return
}

## Open the key and set its value
$key = $baseKey.OpenSubKey($matches[1], $true)
$key.SetValue($property, $PropertyValue)

## Close the key and base keys
$key.Close()
$baseKey.Close()
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

18.15 程序的注册表设置

问题

您希望程序的配置自动执行，但该程序不会记录它的注册表配置设置。

解决方案

若要了解程序的注册表设置，可以使用 Sysinternals 的进程监视器来观察该程序的注册表访问。进程监视器位于 <http://www.microsoft.com/technet/sysinternals/FileAndDisk/processmonitor.mspx>。

讨论

在理想世界中，所有的程序将完全支持命令行管理和 PowerShell cmdlet 的配置。但是许多程序都不支持，因此该解决方案是浏览它们的文档，列出用于控制它们的设置的注册表项和属性。许多程序都记录了它们的注册表配置设置，而许多仍然没有。

虽然这些程序可能不记录它们的注册表设置，但您通常可以看到它们的注册表访问活动，以确定它们使用的注册表路径。为了说明这个问题，我们将使用 Sysinternals 的进程监

视器来发现 PowerShell 的执行策略配置项。尽管 PowerShell 记录了这些注册表项并轻松使其自动配置，但实际上它使用了一种通用技术。

启动并配置进程监视器

一旦您已经下载了进程监视器，第一步是要对其筛选，使其只输出包含您感兴趣的程序。默认情况下进程监视器将记录系统几乎所有的注册表和文件活动。

首先，启动进程监视器，然后按 Ctrl-E（或单击放大镜图标）以暂时防止捕获任何数据（请参见图 18-2）。接下来，按 Ctrl-X（或单击带有橡皮擦图标的空白工作表）以清除它自动捕获的额外信息。最后，拖动目标图标并将其释放放在监视器应用程序上。您可以按 Ctrl-L（或单击漏斗图标）查看当前用于它的输出的进程监视器的过滤器。

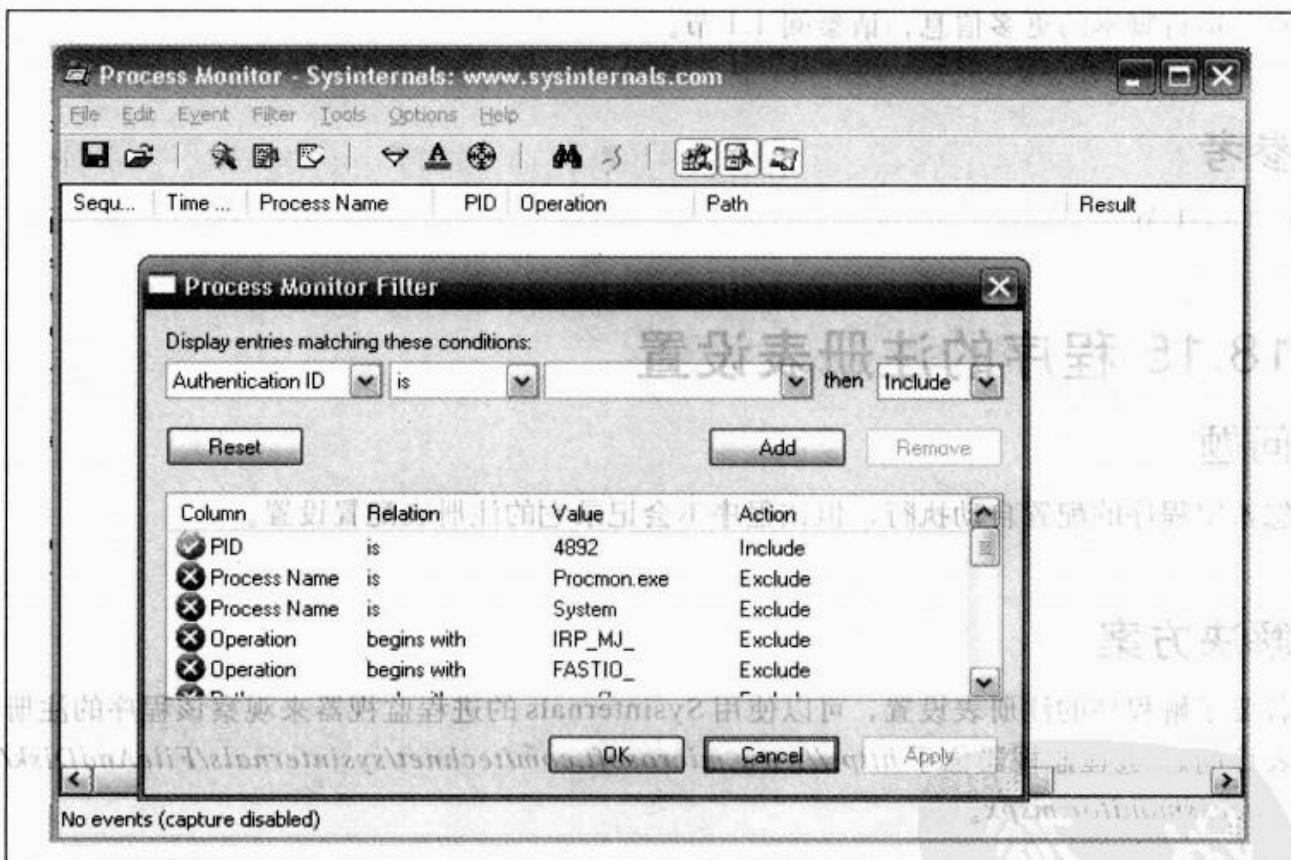


图 18-2：进程监视器读取捕获进程

准备手动设置配置选项

接下来，准备手动设置程序的配置选项。通常，这意味着键入并单击所有属性设置，但不只是单击确定 (OK) 或应用 (Apply)。对于本 PowerShell 示例，请键入 Set-ExecutionPolicy 命令行，但不要按 Enter 键（见图 18-3）。

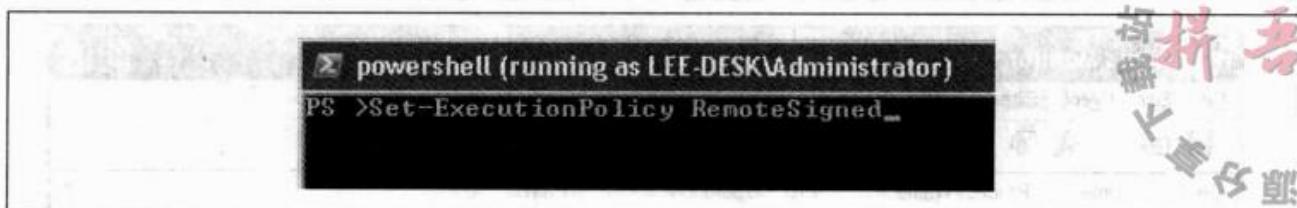


图 18-3：准备应用配置选项

告知进程监视器开始捕获信息

切换到进程监视器窗口，然后按 Ctrl-E（或单击放大镜图标）。进程监视器现在开始捕获程序的所有注册表的访问。

手动设置配置选项

单击确定、应用，或任何完成程序的配置所需的操作。对于该 PowerShell 示例，这意味着按 Enter 键。

告知进程监视器停止捕获信息

再次切换到进程监视器窗口，然后按 Ctrl-E（或单击放大镜图标）。进程监视器现在将不再捕获应用程序的活动。

查看捕获日志以修改注册表

进程监视器窗口现在显示了它配置设置应用后与该应用程序交互的所有注册表项。

按 Ctrl-F（或单击望远镜图标），然后搜索 RegSetValue。进程监视器突出显示对注册表项的第一个修改，如图 18-4 中所示。按 Enter 键（或双击所在的突出显示的行）查看有关此特定的注册表修改详细信息。在此示例中，我们可以看到 PowerShell 把 ExecutionPolicy 属性（在 HKLM:\Software\Microsoft\PowerShell\1\ShellIds\Microsoft.PowerShell 目录下）更改为 RemoteSigned。按 F3 键以查看对应于注册表修改的下一个条目。

自动完成注册表写入

现在，您已经知道更新设置时注册表写入的应用程序，判断和试验，将帮助您确定哪些修改真正表示这个设置。由于 PowerShell 仅执行一次注册表写入（对明显表示执行策略的项的写操作），所以在此示例中的选择很清晰。

The screenshot shows the Process Monitor application window titled "Process Monitor - Sysinternals: www.sysinternals.com". The main pane displays a list of registry events for the process "powershell.exe" (PID 4892). The columns are "Sequence", "Time", "Process Name", "PID", "Operation", and "Path". The operations listed are primarily Registry Query and Open operations on keys under "HKLM\Software\Microsoft\PowerShell\1\ShellIds" and "HKLM\Software\Microsoft\PowerShell\1\ShellIds\". There are also a few "QueryOpen" and "RegSetValue" operations. The bottom status bar indicates "Showing 59 of 16,827 events (0.35%)".

Seq...	Time ...	Process Name	PID	Operation	Path
9799	3:04:2...	powershell.exe	4892	RegQueryValue	HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\
9800	3:04:2...	powershell.exe	4892	RegQueryValue	HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\
9801	3:04:2...	powershell.exe	4892	RegCloseKey	HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\
9802	3:04:2...	powershell.exe	4892	QueryOpen	C:\Windows\assembly\GAC_MSIL\Microsoft.PowerSh
9803	3:04:2...	powershell.exe	4892	RegOpenKey	HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\
9804	3:04:2...	powershell.exe	4892	RegQueryValue	HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\
9807	3:04:2...	powershell.exe	4892	RegSetValue	HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\
9808	3:04:2...	powershell.exe	4892	RegCloseKey	HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\
9809	3:04:2...	powershell.exe	4892	RegOpenKey	HKLM\Software\Policies\Microsoft\Windows\PowerS
9810	3:04:2...	powershell.exe	4892	RegOpenKey	HKCU\Software\Policies\Microsoft\Windows\PowerS
9811	3:04:2...	powershell.exe	4892	RegOpenKey	HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\
9812	3:04:2...	powershell.exe	4892	RegQueryValue	HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\
9813	3:04:2...	powershell.exe	4892	RegQueryValue	HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\
9814	3:04:2...	powershell.exe	4892	RegCloseKey	HKLM\SOFTWARE\Microsoft\PowerShell\1\ShellIds\
9815	3:04:2...	powershell.exe	4892	RegOpenKey	HKCU\Control Panel\International

图 18-4：注册表监视器访问详情

一旦您已经发现应用程序用来存储其配置数据的注册表项、属性和值，你就可以使用在 18.3 节中讨论的方法来自动完成这些配置设置。

例如：

```
PS >$key = "HKLM:\Software\Microsoft\PowerShell\1\" +
>>      "ShellIds\Microsoft.PowerShell"
>>
PS >Set-ItemProperty $key ExecutionPolicy AllSigned
PS >Get-ExecutionPolicy
AllSigned
PS >Set-ItemProperty $key ExecutionPolicy RemoteSigned
PS >Get-ExecutionPolicy
RemoteSigned
```

参考

- 18.3 节

数据比较

19.0 简介

在使用 PowerShell 工作时，通常要使用对象集合。与您使用 .NET Framework 中的许多方法一样，大多数 PowerShell 命令能生成对象。PowerShell 引入了 Compare-Object cmdlet 来帮助使用这些对象集合。Compare-Object cmdlet 与众所周知的 diff 命令的功能相似，使用了面向对象的风格。

19.1 比较两个命令的输出

问题

你想比较两个命令的输出。

解决方案

要比较两个命令的输出，要将每个命令的输出存储在变量中，然后使用 Compare-Object cmdlet 来比较这些变量：

```
PS >notepad
PS >$processes = Get-Process
PS >Stop-Process -ProcessName Notepad
PS >$newProcesses = Get-Process
PS >Compare-Object $processes $newProcesses
InputObject           SideIndicator
-----  
System.Diagnostics.Process (notepad) <=
```

讨论

该解决方案显示了如何确定在对 Get-Process 的两次调用中有哪些进程已退出了。SideIndicator <= 告诉我们该进程存在于左集合 (\$processes) 中而不在右集合 (\$newProcesses) 中。

有关 Compare-Object cmdlet 的更多信息，请键入 **Get-Help Compare-Object**。

19.2 确定两个文件之间的差异

问题

你想确定两个文件之间的差异。

解决方案

为了确定每个文件的内容上的小差异，要将它们的内容存储在变量中，然后使用 Compare-Object cmdlet 来比较这些变量：

```
PS >"Hello World" > c:\temp\file1.txt
PS >"Hello World" > c:\temp\file2.txt
PS >"More Information" >> c:\temp\file2.txt
PS >$content1 = Get-Content c:\temp\file1.txt
PS >$content2 = Get-Content c:\temp\file2.txt
PS >Compare-Object $content1 $content2

InputObject           SideIndicator
-----              -----
More Information     =>
```

讨论

Compare-object cmdlet 的要点是要比较两个非排序的对象集。尽管这些对象集可以是字符串（作为两个文件的内容），但文件在运行时内容通常是无规则的，因此 Compare-Object 实际上是在对无序对象进行比较。

在比较大文件（或顺序文件）时，仍然可以使用传统的文件比较工具，如 Windows 工具和 Visual Studio 附带的 diff.exe 和 WinDiff 应用程序。

有关 Compare-Object cmdlet 的更多信息，请键入 **Get-Help Compare-Object**。

19.3 验证文件集的完整性

问题

你想确定一组文件中的任何文件是否被修改或损坏。

解决方案

若要验证文件集的完整性，请使用 17.10 节中提供的 Get-FileHash 脚本来给正在讨论中的那些文件生成唯一标识。与已知的文件系统做法相同，最后使用 Compare-Object cmdlet 来比较这两个集。

讨论

要给问题中的文件生成唯一标识，使用如下命令：

```
dir C:\Windows\System32\WindowsPowerShell\v1.0 | Get-FileHash |  
Export-CliXml c:\temp\PowerShellHashes.clixml
```

此命令从 *C:\Windows\System32\WindowsPowerShell\v1.0* 获取这些文件的哈希值，并使用 Export-CliXml cmdlet 将这些数据存储在一个文件。

在文件系统传输此文件，然后从该文件导入数据。

```
$otherHashes = Import-CliXml c:\temp\PowerShellHashes.clixml
```

注意：您还可以将网络驱动器映射到问题中的文件上并跳过导出、传输和完全导入这些步骤：

```
net use x: \\lee-desk\c$\Windows\System32\WindowsPowerShell\v1.0  
$otherHashes = dir x: | Get-FileHash
```

从你知道的这个完整文件中生成信息：

```
$knownHashes = dir C:\Windows\System32\WindowsPowerShell\v1.0 |  
Get-FileHash
```

最后，使用 Compare-Object cmdlet 来检测可能存在的差异：

```
Compare-Object $otherHashes $knownHashes -Property Path,HashValue
```

如果有任何差异，Compare-Object cmdlet 会在列表中显示出来，如例 19-1 所示。

例 19-1：Compare-Object cmdlet 显示两个文件的不同

```
PS >Compare-Object $otherHashes $knownHashes -Property Path,HashValue
```

```
Path          HashValue      SideIndicator
----          -----          -----
system.management.aut... 247F291CCDA8E669FF9FA... =>
system.management.aut... 5A68BC5819E29B8E3648F... <=
```

```
PS >Compare-Object $otherHashes $knownHashes -Property Path,HashValue |  
>>   Select-Object Path  
>>
```

```
Path
----  
system.management.automation.dll-help.xml  
system.management.automation.dll-help.xml
```

有关 Compare-Object cmdlet 的更多信息，请键入 **Get-Help Compare-Object**。有关 Export-CliXml 和 Import-CliXml cmdlets 的更多信息，请分别键入 **Get-Help Export-CliXml** 和 **Get-Help Import-CliXml**。

参考

- 17.10 节

如果希望将所有与系统管理相关的命令组织到一个单独的模块中，可以使用 **Import-Module** 命令从一个单独的模块加载所有与系统管理相关的命令。

如果希望在单个命令中包含所有与系统管理相关的命令，可以使用 **Import-Module** 命令从一个单独的模块加载所有与系统管理相关的命令。

事件日志

20.0 简介

Windows 中大多数监视和诊断的内核都会形成事件日志。为了支持此行为，PowerShell 提供 Get-EventLog cmdlet 来使您能够查询和使用系统事件日志数据。除了 PowerShell 内置的 Get-EventLog cmdlet，同样的它也支持 .NET Framework。这意味着您可以访问远程计算机上的事件日志、给事件日志添加条目以及创建和删除事件日志。

20.1 列出所有事件日志

问题

你想确定系统中存在哪些事件日志。

解决方案

若要列出系统中的事件日志，可以使用 Get-EventLog cmdlet 的 -List 参数：

```
PS >Get-EventLog -List
```

Max(K)	Retain	OverflowAction	Entries	Name
512	0	OverwriteAsNeeded	2,157	ADAM (Test)
512	7	OverwriteOlder	2,090	Application
512	7	OverwriteOlder	0	Internet Explorer
8,192	45	OverwriteOlder	0	Media Center
512	7	OverwriteOlder	0	ScriptEvents
512	7	OverwriteOlder	2,368	System
15,360	0	OverwriteAsNeeded	0	Windows PowerShell

讨论

Get-EventLog cmdlet 的 -List 参数生成了系统注册事件的日志列表。与几乎所有的 PowerShell 命令的输出一样，这些事件日志是完全具有.NET 对象特色的，其具有.NET 系统的 `Diagnostics.EventLog` 对象类型。有关如何使用这些对象来把项目写入到事件日志信息，请参阅 20.8 节。

注意：虽然 Get-EventLog 的输出显示了一个表头，而你实际需要的是 Where-Object (和类似的命令) 中使用的日志属性。

有关 Get-EventLog cmdlet 的更多信息，请键入 **Get-Help Get-EventLog**。

参考

- 20.8 节

20.2 从事件日志中获取最新项

问题

如何从事件日志中检索最新项。

解决方案

若要从事件日志中检索最新条目，要使用如例中 20-1 所示的 Get-EventLog cmdlet 的 -Newest 参数。

例 20-1：从系统事件日志中检索最新 10 条日志

```
PS >Get-EventLog System -Newest 10 | Format-Table Index,Source,Message -A
```

Index	Source	Message
2922	Service Control Manager	The Background Intelligent Transfer Servi...
2921	Service Control Manager	The Background Intelligent Transfer Servi...
2920	Service Control Manager	The Logical Disk Manager Administrative S...
2919	Service Control Manager	The Logical Disk Manager Administrative S...
2918	Service Control Manager	The Logical Disk Manager Administrative S...
2917	TermServDevices	Driver Microsoft XPS Document Writer requi...
2916	Print	Printer Microsoft Office Document Image W...
2915	Print	Printer Microsoft Office Document Image W...
2914	Print	Printer Microsoft Office Document Image W...
2913	TermServDevices	Driver Microsoft Shared Fax Driver requir...

讨论

Get-EventLog cmdlet 的 -Newest 参数从您指定的事件日志中检索最新条目。若要列出系统中可用的事件日志，请参阅 20.1 节。

有关 Get-EventLog cmdlet 的更多信息，请键入 **Get-Help Get-EventLog**。

参考

- 20.1 节

20.3 使用特定的文本查找事件日志项

问题

如何检索包含给定条件的所有事件日志项。

解决方案

若要查找特定的事件日志项，可以使用 Get-EventLog cmdlet 来检索项，然后把它们传到 Where-Object cmdlet 来进行筛选，如例 20-2 所示。

例 20-2：检索记录磁盘的事件日志

```
PS >Get-EventLog System | Where-Object { $_.Message -match "disk" }
```

Index	Time	Type	Source	EventID	Message
2920	May 06 09:18	Info	Service Control M...	7036	The Logical Disk...
2919	May 06 09:17	Info	Service Control M...	7036	The Logical Disk...
2918	May 06 09:17	Info	Service Control M...	7035	The Logical Disk...
2884	May 06 00:28	Error	sr	1	The System Resto...
2333	Apr 03 00:16	Error	Disk	11	The driver detect...
2332	Apr 03 00:16	Error	Disk	11	The driver detect...
2131	Mar 27 13:59	Info	Service Control M...	7036	The Logical Disk...
2127	Mar 27 12:48	Info	Service Control M...	7036	The Logical Disk...
2126	Mar 27 12:48	Info	Service Control M...	7035	The Logical Disk...
2123	Mar 27 12:31	Info	Service Control M...	7036	The Logical Disk...
2122	Mar 27 12:29	Info	Service Control M...	7036	The Logical Disk...
2121	Mar 27 12:29	Info	Service Control M...	7035	The Logical Disk...

讨论

Get-EventLog cmdlet 检索出代表事件日志项的多个对象，您可以把它们传到 Where-Object cmdlet 来进行筛选。

默认情况下PowerShell的默认表格显示事件日志项的摘要。如果你正在搜索事件日志消息，您可能还想查看更多的详细信息。在这种情况下，使用 `Format-List` cmdlet 来把这些项的格式显示在更详细的列表视图中。例 20-3 显示了这个视图。

例 20-3：一个事件日记的详细列表视图

```
PS >Get-EventLog System | Where-Object { $_.Message -match "disk" } |
```

>> Format-List

2

```
Index          : 2920
EntryType      : Information
EventID        : 7036
Message        : The Logical Disk Manager Administrative Service service entered the stopped state.
Category       : {0}
CategoryNumber : 0
ReplacementStrings : {Logical Disk Manager Administrative Service, stopped}
Source         : Service Control Manager
TimeGenerated   : 5/6/2007 9:18:25 AM
TimeWritten     : 5/6/2007 9:18:25 AM
UserName        :
Index          : 2919
(...)
```

有关 `Get-EventLog` cmdlet 的更多信息，请键入 `Get-Help Get-EventLog`。有关筛选命令输出的详细信息，请参阅 2.1 节。

参考

- 2.1 节

20.4 检索一个特定的事件日志项

问题

你想检索一个特定的事件日志项。

解决方案

若要检索一个特定的事件日志项，可以使用 `Get-EventLog` cmdlet 来检索事件日志中的项目，然后管道输出到 `Where-Object` cmdlet 来筛选您所要查找的事件日志项。

```
PS >Get-EventLog System | Where-Object { $_.Index -eq 2920 }  
Index Time Type Source EventID Message  
----- ---- ---- ----- -----  
2920 May 06 09:18 Info Service Control M... 7036 The Logical Disk...
```

讨论

当你列出具有特定的文本的事件日志的项或搜索的结果时，你通常想进一步获取某个特定的事件日志项的详细资料。

因为 Get-EventLog cmdlet 检索出丰富的代表事件日志项的对象，所以你可以把它们管道输出到 Where-Object cmdlet 用于同样丰富的过滤。

默认情况下 PowerShell 的表格格式显示事件日志项的摘要。如果您正在检索一个特定的项，您可能还想查看更多的详细信息。在这种情况下，使用 Format-List cmdlet 来把这些项格式显示在更详细的列表视图中。例 20-4 显示了这个视图。

例 20-4：事件日志项的详细列表视图

```
PS > Get-EventLog System | Where-Object { $_.Index -eq 2920 } |  
>> Format-List  
>>  
  
Index : 2920  
EntryType : Information  
EventID : 7036  
Message : The Logical Disk Manager Administrative Service service entered the stopped state.  
Category : (0)  
CategoryNumber : 0  
ReplacementStrings : {Logical Disk Manager Administrative Service, stopped}  
Source : Service Control Manager  
TimeGenerated : 5/6/2007 9:18:25 AM  
TimeWritten : 5/6/2007 9:18:25 AM  
UserName :  
  
Index : 2919  
(...)
```

有关 Get-EventLog cmdlet 的更多信息，请键入 **Get-Help Get-EventLog**。有关筛选命令输出的详细信息，请参阅 2.1 节。

参考

- 2.1 节

20.5 根据频率查找事件日志记录

问题

你想查找发生频率最高的事件日志记录。

解决方案

若要根据频率查找事件日志记录，可以使用 `Get-EventLog` cmdlet 来检索事件日志中的项，然后把它们管道输出到 `Group-Object` cmdlet 来按其信息对它们进行分组。

```
PS >Get-EventLog System | Group-Object Message
```

Count	Name	Group
23	The Background Intelli...	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
23	The Background Intelli...	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
3	The Logical Disk Manag...	{LEE-DESK, LEE-DESK, LEE-DESK}
3	The Logical Disk Manag...	{LEE-DESK, LEE-DESK, LEE-DESK}
3	The Logical Disk Manag...	{LEE-DESK, LEE-DESK, LEE-DESK}
161	Driver Microsoft XPS D...	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
(...)		

讨论

`Group-Object` cmdlet 是确定您的系统中哪些事件最常发生的一种实用的方法。这里还提供了一种非常有用的方法来汇总在事件日志中的信息。

若要了解有关特定组中的项的更多信息，可以使用 `Where-Object` cmdlet。由于我们使用 `Group-Object` cmdlet 的 `Message` 属性，因此需要对 `Where-Object` cmdlet 的消息进行筛选。例如，要了解 Microsoft XPS 驱动程序的相关项（解决方案中的方法）的更多信息，如下：

```
PS >Get-EventLog System |  
>> Where-Object { $_.Message -like "Driver Microsoft XPS*" }  
>>
```

Index	Time	Type	Source	EventID	Message
2917	May 06 09:13	Erro	TermServDevices	1111	Driver Microsoft...
2883	May 05 10:40	Erro	TermServDevices	1111	Driver Microsoft...
2877	May 05 08:10	Erro	TermServDevices	1111	Driver Microsoft...
(...)					

如果信息分组不能提供有用的信息，也可以按任何其他属性进行分组，如源代码：

```
PS >Get-EventLog Application | Group-Object Source
Count Name          Group
-----
4 Application      {LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK}
191 Media Center Scheduler {LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
1082 MSSQL$SQLEXPRESS {LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
(...)
```

当你列出具有特定文本内容的事件日志的项或搜索的结果时，你通常想进一步获取某个特定的事件日志项的详细资料。

默认情况下 PowerShell 的表格可以显示事件日志项的摘要信息。如果您正在检索一个特定的项目，您可能还想查看更多的详细信息。在这种情况下，可以使用 Format-List cmdlet 来把这些项格式显示在更详细的列表视图中。如例 20-5 中所示。

例 20-5：事件日志项的详细列表视图

```
PS >Get-EventLog System | Where-Object { $_.Index -eq 2917 } |
>>   Format-List
>>
Index          : 2917
EntryType      : Error
EventID        : 1111
Message        : Driver Microsoft XPS Document Writer required for pri
                  nter Microsoft XPS Document Writer is unknown. Contac
                  t the administrator to install the driver before you
                  log in again.
Category       : (0)
CategoryNumber : 0
ReplacementStrings : {Microsoft XPS Document Writer, Microsoft XPS Documen
                     t Writer}
Source         : TermServDevices
TimeGenerated   : 5/6/2007 9:13:31 AM
TimeWritten    : 5/6/2007 9:13:31 AM
UserName       :
```

有关 Get-EventLog cmdlet 的更多信息，请键入 **Get-Help Get-EventLog**。有关筛选命令输出的详细信息，请参阅 2.1 节。有关 Group-Object cmdlet 的更多信息，请键入 **Get-Help Group-Object**。

参考

- 2.1 节

20.6 备份事件日志

问题

你想将事件日志中的信息存储在文件中来作为备份或审查之用。

解决方案

要在文件中存储事件日志项，可以使用 `Get-EventLog` cmdlet 检索事件日志中的项，然后把它们管道输出到 `Export-CliXml` cmdlet 来将其存储在一个文件中。

```
Get-EventLog System | Export-CliXml c:\temp\SystemLogBackup.clixml
```

讨论

一旦您已经从事件日志中导出了事件，您就可以把它们存档，或使用 `Import-CliXml` cmdlet 在任何安装了 PowerShell 的计算机上查看它们：

```
PS >$archivedLogs = Import-CliXml c:\temp\SystemLogBackup.clixml
PS >$archivedLogs | Group Source
```

Count	Name	Group
856	Service Control Manager	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
640	TermServDevices	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
91	Print	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
100	WMPNetworkSvc	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
123	Tcpip	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
(...)		

有关 `Get-EventLog` cmdlet 的更多信息，请键入 `Get-Help Get-EventLog`。有关 `Export-CliXml` 和 `Import-CliXml` cmdlet 的详细信息，请分别键入 `Get-Help Export-CliXml` 和 `Get-Help Import-CliXml` 查看。

20.7 创建或删除事件日志

问题

你想创建或删除事件日志。

解决方案

若要创建事件日志，可以使用.NET Framework 的 [System.Diagnostics.EventLog]::CreateEventSource() 方法：

```
$newLog =  
    New-Object Diagnostics.EventSourceCreationData  
    "PowerShellCookbook", "ScriptEvents"  
  
[Diagnostics.EventLog]::CreateEventSource($newLog)
```

要删除事件日志，可以使用.NET Framework 的 [System.Diagnostics.EventLog]::Delete() 方法：

```
[Diagnostics.EventLog]::Delete("ScriptEvents")
```

讨论

.NET Framework 中的 [System.Diagnostics.EventLog]::CreateEventSource() 方法注册了一个新的事件源（解决方案中的 PowerShellCookbook）并将项目写入事件日志中（解决方案中的 ScriptEvents）。如果在事件日志中不存在，CreateEventSource() 方法将创建事件日志。

.NET Framework 的 [System.Diagnostics.EventLog]::Delete() 方法连同与事件日志关联的任何事件源一起完全删除。若只要删除一个特定的事件源，可以使用.NET Framework 中的 [System.Diagnostics.EventLog]::DeleteEventSource() 方法。

注意：请小心操作，如果误删错删了事件日志便很难重新创建该事件源了。

有关使用 .NET Framework 中的类的详细信息，请参阅 3.4 节。

参考

- 3.4 节

20.8 写入事件日志

问题

你想向事件日志中添加项。

解决方案

若要写入一个事件日志，可以使用 Get-EventLog cmdlet 的 -List 参数来检索正确的事件日志。然后将其源设置为一个已注册的事件日志源并调用它的 WriteEntry() 方法：

```
PS >$log = Get-EventLog -List | Where-Object { $_.Log -eq "ScriptEvents" }
PS >$log.Source = "PowerShellCookbook"
PS >$log.WriteEntry("This is a message from my script.")
PS >
PS >Get-EventLog ScriptEvents -Newest 1 | Select Source,Message
Source                               Message
-----
PowerShellCookbook                  This is a message from my script.
```

讨论

如该解决方案提到的，您必须在信息写入日志之前将事件日志的来源属性设置为一个已注册的事件日志源。如果您还没有在系统中注册事件日志源，请参阅 20.7 节。

有关在 .NET Framework 中使用类的更多信息，请参阅 3.4 节。

参考

- 3.4 节
- 20.7 节

20.9 访问远程计算机的事件日志

问题

你想访问远程计算机的事件日志项。

解决方案

若要访问在远程计算机上的事件日志，可以使用日志名称和计算机名称创建一个新的 System.Diagnostics.EventLog 类。然后访问它的属性项：

```
PS >$log = New-Object Diagnostics.EventLog "System", "LEE-DESK"
PS >$log.Entries | Group-Object Source
```

Count	Name	Group
91	Print	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
640	TermServDevices	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
148	W32Time	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
100	WMPNetworkSvc	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
856	Service Control Manager	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
123	Tcpip	{LEE-DESK, LEE-DESK, LEE-DESK, LEE-DESK...}
(...)		

讨论

该解决方案演示了访问远程计算机上事件日志的一种方法。除了检索事件日志项，`System.Diagnostics.EventLog`类还可以对远程计算机进行如创建事件日志、删除事件日志、编写事件日志项等其他操作。

`System.Diagnostics.EventLog`类提供了通过另外的参数支持管理事件日志的方法。例如，从远程机器获取事件日志。

[`Diagnostics.EventLog`]::`GetEventLogs("LEE-DESK")`在远程计算机上创建事件日志或事件源：

```
$newLog =  
    New-Object Diagnostics.EventSourceCreationData  
    "PowerShellCookbook", "ScriptEvents"  
    $newLog.MachineName = "LEE-DESK"  
    [Diagnostics.EventLog]::CreateEventSource($newLog)
```

在远程计算机上写事件日志：

```
$log = New-Object Diagnostics.EventLog "ScriptEvents", "LEE-DESK"  
$log.Source = "PowerShellCookbook"  
$log.WriteEntry("Test event from a remote machine.")
```

有关如何获取事件日志的信息，请参阅 20.1 节；有关如何创建或删除事件日志的信息，请参阅 20.7 节；要获取有关如何编写事件日志项目的更多信息，请参阅 20.8 节。

有关使用 .NET Framework 中的类的详细信息，请参阅 3.4 节。

参考

- 3.4 节
- 20.1 节
- 20.7 节
- 20.8 节

第 21 章

进程

金书

使用系统进程是系统管理中很平常的事情。令系统管理员自豪的技能大多数是书写出神入化的正则表达式功夫。毕竟，用户不会使用 Unix one-liner 去停止使用超过 100 MB 内存的所有进程：

```
ps -el | awk '{ if ( $6 > (1024*100) ) { print $3 } }' | grep -v PID | xargs kill
```

帮助中阐释了纯文本进程本身易损的性质。为命令能成功执行，必须执行下列步骤：

- 依赖 ps 命令在第 6 列中显示内存使用情况。
- 依赖 ps 命令的输出的第 6 列表示内存的使用状况以 kB 为单位。
- 依赖 ps 命令的输出第 3 列表示进程 ID。
- 从 ps 命令的输出中移除列头。

因为 PowerShell 的 Get-Process cmdlet 返回的信息为 .NET 对象结构，易损的文本进程已成为过去：

```
Get-Process | Where-Object { $_.WorkingSet -gt 100mb } | Stop-Process -WhatIf
```

简写很重要，PowerShell 定义了别名，以使大多数命令更易于键入：

```
gps | ? { $_.WS -gt 100mb } | kill -WhatIf
```

21.1 列出当前运行的进程

问题

你想查看当前系统中正在运行的进程。

解决方案

若要检索当前正在运行的进程列表，可以使用 Get-Process cmdlet：

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
274	6	1328	3940	33		1084	alg
85	4	3816	6656	57	5.67	3460	AutoHotkey
50	2	2292	1980	14	384.25	1560	BrmfRsmg
71	3	2520	4680	35	0.42	2592	cmd
946	7	3676	6204	32		848	csrss
84	4	732	2248	22		3144	csrss
68	4	936	3364	30	0.38	3904	ctfmon
243	7	3648	9324	48	2.02	2892	Ditto
(...)							

讨论

Get-Process cmdlet 检索系统中运行的所有进程的相关信息。因为它们是 .NET 对象 (System.Diagnostics.Process 类型)，所以高级过滤和操作要比以往任何时候都更容易。

例如，要查找使用的内存超过 100 MB 的所有进程：

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
1458	29	83468	105824	273	323.80	3992	BigBloatedApp

按公司进行进程分组：

Count	Name	Group
39	{alg, csrss, csrss, dllhost...}	
4	{AutoHotkey, Ditto, gnuserv, mafwTray}	
1	Brother Industries, Ltd.	{BrmfRsmg}
19	Microsoft Corporation	{cmd, ctfmon, EXCEL, explorer...}

```
1 Free Software Foundation {emacs}
1 Microsoft (R) Corporation {FwcMgmt}
(...)
```

或可能按开始时间进行排序（最先的进程排在最前面）：

```
PS >Get-Process | Sort -Descending StartTime | Select-Object -First 10
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
1810	39	53616	33964	193	318.02	1452	iTunes
675	6	41472	50180	146	49.36	296	powershell
1240	35	48220	58860	316	167.58	4012	OUTLOOK
305	8	5736	2460	105	21.22	3384	WindowsSearch...
464	7	29704	30920	153	6.00	3680	powershell
1458	29	83468	105824	273	324.22	3992	iexplore
478	6	24620	23688	143	17.83	3548	powershell
222	8	8532	19084	144	20.69	3924	EXCEL
14	2	396	1600	15	0.06	2900	logon.scr
544	18	21336	50216	294	180.72	2660	WINWORD

这些高级的任务因 PowerShell 因每个进程返回的详细的信息而变得非常简单。有关 `Get-Process` cmdlet 的更多信息，请键入 `Get-Help Get-Process`。有关 PowerShell 命令中的筛选、分组和排序的更多信息，请参阅 2.1 节。

有关使用 .NET Framework 中的类的详细信息，请参阅 3.4 节。

参考

- 2.1 节
- 3.4 节

21.2 启动一个进程

问题

你想在系统中启动一个新的进程，并配置其启动环境。

解决方案

若要启动一个新的进程，可以使用 `[System.Diagnostics.Process]::Start()` 方法。若要控制其启动环境，需要提供一个 `System.Diagnostics.ProcessStartInfo` 对象准备给它，如例 21-1 所示。

例 21-1：为新进程配置启动环境

```
$credential = Get-Credential
## Prepare the startup information (including username and password)
$startInfo = New-Object Diagnostics.ProcessStartInfo
$startInfo.UserName = $credential.Username
$startInfo.Password = $credential.Password
$startInfo.Filename = "powershell"

## Start the process
$startInfo.UseShellExecute = $false
[Diagnostics.Process]::Start($startInfo)
```

讨论

正常情况下，在 PowerShell 中启动一个进程与键入程序名一样简单：

```
PS >notepad c:\temp\test.txt
```

但是，您有时可能需要对进程详细信息进行详细控制，如它的凭据、启动目录、环境变量等。在这些情况下，`[System.Diagnostics.Process]::Start()`方法提供这些功能。

注意：以下函数与 `cmd.exe` 的 `start` 命令类似，如 Windows 中的 Start | Run 对话框：

```
PS >function start { [Diagnostics.Process]::Start($args) }
PS >start www.msn.com
```

有关从 PowerShell 启动程序的更多信息，请参阅 1.1 节。有关使用 .NET Framework 中的类的详细信息，请参阅 3.4 节。

参考

- 1.1 节
- 3.4 节

21.3 停止一个进程

问题

你想停止（或终止）系统中的一个进程。

解决方案

若要停止一个进程，使用 Stop-Process cmdlet，如例 21-2 中所示。

例 21-2：使用 Stop-Process cmdlet 停止一个进程

```
PS >notepad
```

```
PS >Get-Process Notepad
```

Handles	NPM (K)	PM (K)	WS (K)	VM (M)	CPU (s)	Id	ProcessName
42	3	1276	3916	32	0.09	3520	notepad

```
PS >Stop-Process -ProcessName notepad
```

```
PS >Get-Process Notepad
```

```
Get-Process : Cannot find a process with the name 'Notepad'. Verify the  
process name and call the cmdlet again.
```

```
At line:1 char:12
```

```
+ Get-Process <<<< Notepad
```

讨论

虽然 Stop-Process cmdlet 的参数在它们自己的权限中有用，但 PowerShell 的管道模型可以做到更精确。Stop-Process cmdlet 停止您管道中的任意进程，因此 Get-Process 生成的一个高级进程集自动变成 Stop-Process cmdlet 要操作的的高级进程集：

```
PS >Get-Process | Where-Object { $_.WorkingSet -lt 10mb } |  
>> Sort-Object -Descending Name | Stop-Process -WhatIf  
>>  
What if: Performing operation "Stop-Process" on Target "svchost (1368)".  
What if: Performing operation "Stop-Process" on Target "sqlwriter (1772)".  
What if: Performing operation "Stop-Process" on Target "qttask (3672)".  
What if: Performing operation "Stop-Process" on Target "Ditto (2892)".  
What if: Performing operation "Stop-Process" on Target "ctfmon (3904)".  
What if: Performing operation "Stop-Process" on Target "csrss (848)".  
What if: Performing operation "Stop-Process" on Target "BrmfRsmg (1560)".  
What if: Performing operation "Stop-Process" on Target "AutoHotkey (3460)".  
What if: Performing operation "Stop-Process" on Target "alg (1084)".
```

注意：请注意本示例在 Stop-Process cmdlet 上使用了 -WhatIf 参数。如果运行该命令，此参数可以看到将会产生的结果，但实际上并不执行该操作。

有关 Stop-Process cmdlet 的更多信息，请键入 **Get-Help Stop-Process**。有关 Where-Object cmdlet 的更多信息，请键入 **Get-Help Where-Object**。

21.4 编程：调用远程计算机上的一个 PowerShell 表达式

例 21-4 可以在远程计算机上启动进程和调用 PowerShell 表达式。它使用 PsExec (从 <http://www.microsoft.com/technet/sysinternals/utilities/psexec.mspx>) 支持远程的命令执行。

这些脚本提供了比远程命令执行更多的能力。例 21-3 演示它利用了 PowerShell 的导入和导出强结构的数据，这样您可以使用于本地处理命令输出系统许多相同的技术。例 21-3 演示在远程系统进行筛选命令输入，但在本地系统对其进行排序的能力。

例 21-3：在远程计算机调用一个 PowerShell 表达式。

```
PS >$command = { Get-Process | Where-Object { $_.Handles -gt 1000 } }
```

```
PS >Invoke-RemoteExpression \\LEE-DESK $command | Sort Handles
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
1025	8	3780	3772	32	134.42	848	csrss
1306	37	50364	64160	322	409.23	4012	OUTLOOK
1813	39	54764	36360	321	340.45	1452	iTunes
2316	273	29168	41164	218	134.09	1244	svchost

由于此强结构化的数据来自另一个系统上对象，PowerShell 不重新生成这些对象的功能（极少数情况下除外）。有关导入和导出结构化的数据详细信息，请参阅 8.4 节。

例 21-4：Invoke-RemoteExpression.ps1

```
#####
## Invoke-RemoteExpression.ps1
##
## Invoke a PowerShell expression on a remote machine. Requires PsExec from
## http://www.microsoft.com/technet/sysinternals/utilities/psexec.mspx
##
## ie:
##
## PS >Invoke-RemoteExpression \\LEE-DESK { Get-Process }
## PS >(Invoke-RemoteExpression \\LEE-DESK { Get-Date }).AddDays(1)
## PS >Invoke-RemoteExpression \\LEE-DESK { Get-Process } | Sort Handles
##
#####

param(
    $computer = "\\$ENV:ComputerName",
    [ScriptBlock] $expression = $(throw "Please specify an expression to invoke."),
    [switch] $noProfile
)
```

```
## Prepare the command line for PsExec. We use the XML output encoding so
## that PowerShell can convert the output back into structured objects.
$CommandLine = "echo . | powershell -Output XML"

if($noProfile)
{
从) $CommandLine += "-NoProfile"
}

## Convert the command into an encoded command for PowerShell
$CommandBytes = [System.Text.Encoding]::Unicode.GetBytes($expression)
$EncodedCommand = [Convert]::ToBase64String($CommandBytes)
$CommandLine += "-EncodedCommand $EncodedCommand"

## Collect the output and error output
$errorOutput = [IO.Path]::GetTempFileName()
$output = psexec /acceptEula $computer cmd /c $CommandLine 2>$errorOutput

## Check for any errors
$errorContent = Get-Content $errorOutput
Remove-Item $errorOutput
if($errorContent -match "Access is denied")
{
    $OFS = "`n"
    $errorMessage = "Could not execute remote expression."
    $errorMessage += "Ensure that your account has administrative "
    "privileges on the target machine.`n"
    $errorMessage += ($errorContent -match "psexec.exe :")

Write-Error $errorMessage
}

## Return the output to the user
$output
```

关于运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节
- 8.4 节

系统服务

22.0 简介

作为 Windows 的许多管理任务的支持机制，处理和使用系统服务自然地整合进入管理员的工具箱。

PowerShell 提供了一系列 cmdlet 来使系统服务的使用更加容易：从目录服务到生命周期管理，甚至是服务安装。

22.1 列出所有运行的服务

问题

你想查看系统中有哪些服务在运行。

解决方案

要列出所有正在运行的服务，可以使用 Get-Service cmdlet：

```
PS >Get-Service
```

Status	Name	DisplayName
Running	ADAM_Test	Active Directory Application Mode - ADAM
Stopped	Alerter	Event Log Service
Running	ALG	Application Layer Gateway Service
Stopped	AppMgmt	Application Management
Stopped	aspnet_state	ASP.NET State Service
Running	AudioSrv	Windows Audio
Running	BITS	Background Intelligent Transfer Ser...
Running	Browser	Computer Browser
(...)		

讨论

`Get-Service` cmdlet 检索所有在系统上运行的服务的相关信息。因为它们是.NET 对象 (`System.ServiceProcess.ServiceController` 类型), 所以你可以采用高级的筛选和操作来使管理服务变得简单。

例如, 要查找所有正在运行的服务:

```
PS >Get-Service | Where-Object { $_.Status -eq "Running" }
```

Status	Name	DisplayName
Running	ADAM_Test	Test
Running	ALG	Application Layer Gateway Service
Running	AudioSrv	Windows Audio
Running	BITS	Background Intelligent Transfer Ser...
Running	Browser	Computer Browser
Running	COMSysApp	COM+ System Application
Running	CryptSvc	Cryptographic Services

或者, 对服务进行特定项目的排序:

```
PS >Get-Service | Sort-Object -Descending { $_.DependentServices.Count }
```

Status	Name	DisplayName
Running	RpcSs	Remote Procedure Call (RPC)
Running	PlugPlay	Plug and Play
Running	lanmanworkstation	Workstation
Running	SSDPSRV	SSDP Discovery Service
Running	TapiSrv	Telephony
(...)		

因为 PowerShell 返回代表系统服务的.NET 对象, 所以这些甚至更多的任务由于 PowerShell 为每个服务返回的丰富信息而变得非常简单。有关 `Get-Service` cmdlet 更多信息, 请键入 **Get-Help Get-Service**。更多有关 PowerShell 命令中的筛选、分组和排序信息, 请参阅 2.1 节。

注意: `Get-Services` cmdlet 显示了运行服务的大多数 (但不是全部) 信息。要显示其他信息 (如服务的启动模式), 请使用 `Get-WmiObject` cmdlet:

```
$service = Get-WmiObject Win32_Service |  
    Where-Object { $_.Name -eq "AudioSrv" }  
$service.StartMode
```

有关使用 .NET Framework 中的类的详细信息, 请参阅 3.4 节。有关使用 `Get-WmiObject` cmdlet 的详细信息, 请参阅 15.1 节。

参考

- 2.1 节
- 3.4 节
- 15.1 节

22.2 管理一个正在运行的服务

问题

你想管理一个正在运行的服务。

解决方案

若要停止某项服务，可以使用 **Stop-Service cmdlet**：

```
PS >Stop-Service AudioSrv -WhatIf
What if: Performing operation "Stop-Service" on Target "Windows Audio (AudioSrv)".
```

同样，分别使用 **Suspend-Service**, **Restart-Service** 和 **Resume-Service cmdlet** 来实现挂起、重新启动和恢复服务。

对于其他任务（如设置启动模式），可以使用 **Get-WmiObject cmdlet**：

```
$service = Get-WmiObject Win32_Service
Where-Object { $_.Name -eq "AudioSrv" }
$service.ChangeStartMode("Manual")
$service.ChangeStartMode("Automatic")
```

讨论

Stop-Service cmdlet 使你可以通过名称或显示名称来停止服务。

注意：请注意该解决方案中使用了 **Stop-Service cmdlet** 的 **-WhatIf** 参数。此参数使你能够看到运行后会发生的情况，但实际上并不执行该操作。

有关 **Stop-Service cmdlet** 的更多信息，请键入 **Get-Help Stop-Service**。如果你希望挂起、重新启动或继续服务，请分别参阅 **Suspend-Service**、**Restart-Service** 和 **Resume-Service cmdlet**。

有关使用 Get-WmiObject cmdlet 的详细信息，请参阅 15.1 节。

参考

- 15.1 节

22.3 访问在远程计算机上的服务

问题

你想列出或管理远程计算机上的服务。

解决方案

若要从远程计算机检索服务，可以使用.NET Framework 的 [System.ServiceProcess.ServiceController]::GetServices() 方法。

```
PS >[void] ([Reflection.Assembly]::LoadWithPartialName("System.ServiceProcess"))
PS >[System.ServiceProcess.ServiceController]::GetServices("LEE-DESK")
```

Status	Name	DisplayName
Running	ADAM_Test	Test
Stopped	Alerter	Alerter
Running	ALG	Application Layer Gateway Service
Stopped	AppMgmt	Application Management
Stopped	aspnet_state	ASP.NET State Service
Running	AudioSrv	Windows Audio
Running	BITS	Background Intelligent Transfer Ser...
Running	Browser	Computer Browser
Stopped	CiSvc	Indexing Service

要控制一个对象，可以使用 Where-Object cmdlet 来专门检索对象并调用该对象方法来管理它：

```
[void] ([Reflection.Assembly]::LoadWithPartialName("System.ServiceProcess"))
$service =
[System.ServiceProcess.ServiceController]::GetServices("LEE-DESK") |
Where-Object { $_.Name -eq "Themes" }

$service.Stop()
$service.WaitForStatus("Stopped")
Start-Sleep 2
$service.Start()
```

讨论

如果你有远程计算机管理员权限, .NET Framework 中的 [System.ServiceProcess.ServiceController]::GetServices() 方法便能允许您控制该计算机上的服务。

注意: 当这样做时,请注意解决方案中的这两种示例都要求先加载包含管理服务的 .NET 类的程序集。*-Service cmdlet 会自动加载此 DLL。

有关使用 .NET Framework 中类的详细信息,请参阅 3.4 节。

参考

- 3.4 节

介面 0.82

本文档将介绍如何使用 System.Diagnostics.Process 来管理进程。首先，将介绍如何启动和停止进程，以及如何在进程中注入线程。然后，将探讨如何从进程中读取和写入数据，以及如何在进程中运行命令行。最后，将介绍如何在进程中执行文件操作，如打开、读取和写入文件。

NET 提供了多种方法来启动新进程。最常用的方法是通过 ProcessStartInfo 对象。该对象包含启动新进程所需的所有信息，包括要运行的命令行、工作目录、启动方式（如正常启动或以管理员权限启动）等。

本章目录

回顾

本章将回顾前几章中学习过的概念和术语。

案例分析

Process class (MAUI) 为用户提供了对进程的基本操作。本章将详细介绍如何使用 Process class 来启动新进程。通过演示一个名为 "ProcessExample" 的示例，读者将了解如何在 MAUI 应用程序中使用 Process class。

简介

本章将向读者介绍如何使用 Process class 在 MAUI 应用程序中启动新进程。通过示例代码，读者将了解如何设置启动参数、读取进程输出并处理错误。

第23章

活动目录

23.0 简介

到目前为止，在Windows平台上的最独特的系统管理之一是用活动目录进行信息交互。作为Windows网络的集中式的授权、身份验证和信息存储，活动目录自动形成了许多企业管理任务的核心。

而PowerShell既不包含活动目录cmdlet，也不包含活动目录提供程序，但可以通过.NET Framework提供对活动目录的管理来对其进行访问。

23.1 测试本地安装的活动目录脚本

问题

你想测试本地安装的活动目录脚本。

解决方案

为了测试本地系统的脚本，要安装活动目录应用程序模式（ADAM，Active Directory Application Mode）和它的示例配置。

讨论

若要测试本地化的脚本，你需要安装ADAM，然后建立一个测试用例。

安装 ADAM

要安装 ADAM，第一步先要下载它。微软下载中心提供 ADAM 免费下载。在 <http://download.microsoft.com> 上你能通过搜索“Active Directory Application Mode”来获得。

一旦你下载了它，就可以运行安装 ADAM 程序了。图 23-1 显示 ADAM 安装过程。

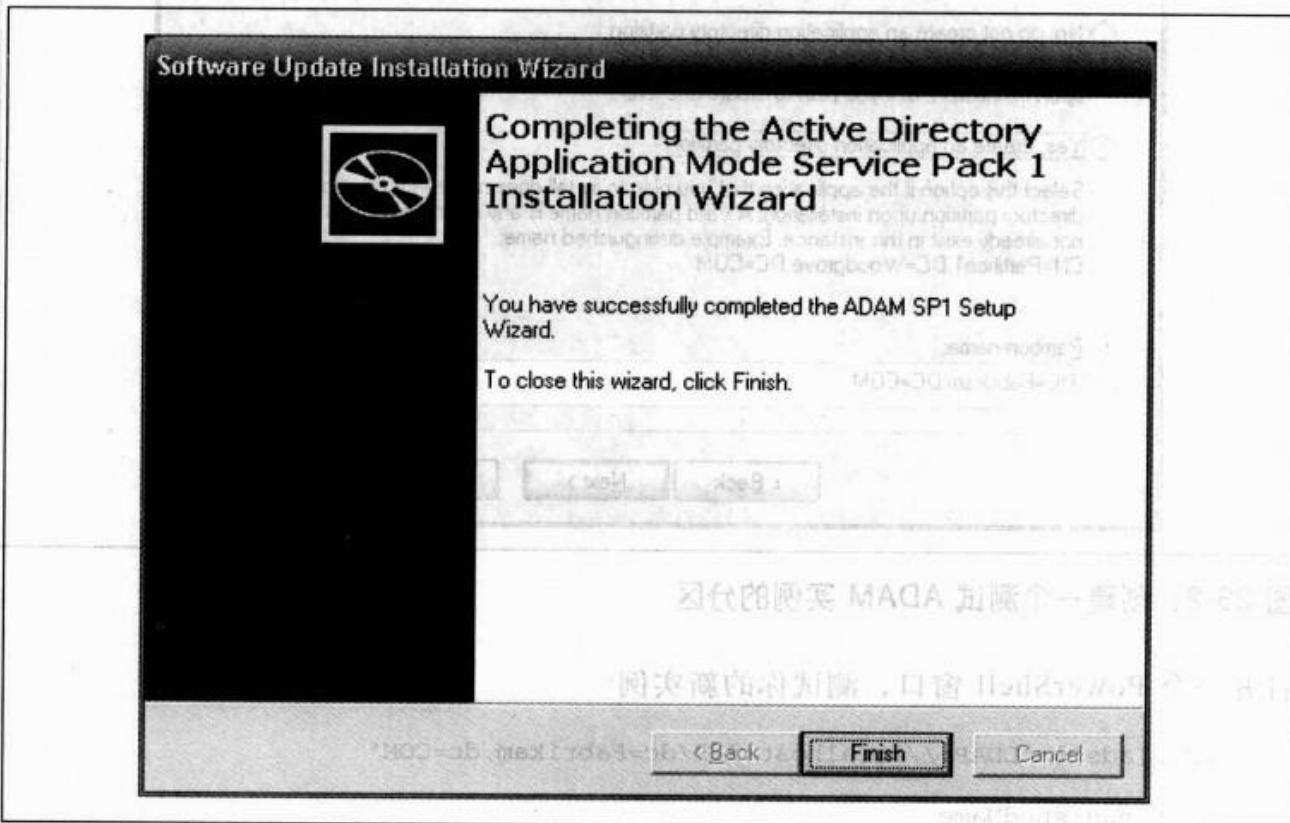


图 23-1：ADAM 的发布安装界面

对壁类样具曲其侧中 Head2now 已 (sectorisq) 为底壁特深深个一深起

建立测试用例

从 Windows 开始菜单中的 ADAM 菜单中选择建立 ADAM 实例，在设置页点下一步，选择唯一的创建实例，在实例名页面，输入 **Test** 作为实例名。端口选择默认的，然后点确定，在下一页上选“是，创建一个应用程序目录分区”，键入 **DC = Fabrikam, DC = COM** 作为分区名称。如图 23-2 中所示。

在下一页中，使用默认文件位置、服务账户和管理员。

如果安装向导为你提供了导入 LDIF 文件的选项，那么你可以导入所有可用的除了用于 **MS-AZMan.LDF** 的文件。单击下一步，确认页完成该实例安装。

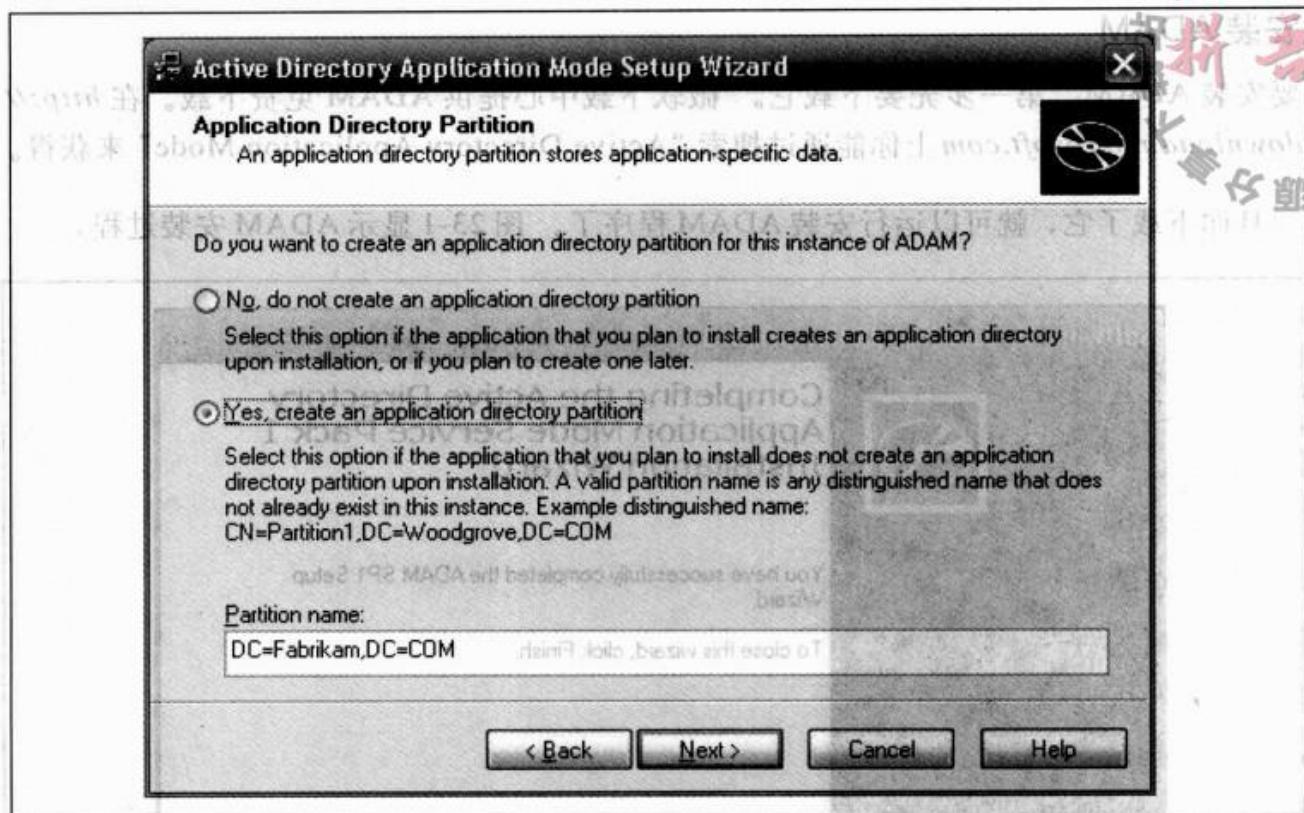


图 23-2：创建一个测试 ADAM 实例的分区

打开一个 PowerShell 窗口，测试你的新实例：

```
PS >[adsi] "LDAP://localhost:389/dc=Fabrikam,dc=COM"  
distinguishedName  
-----  
{DC=Fabrikam,DC=COM}
```

[adsi] 标记是一个类型快捷方式 (typeshotcat)，与 PowerShell 中的其他几种类型快捷方式类似。[adsi] 类型快捷方式可以通过活动目录服务接口快速创建和使用目录。

尽管 ADAM 操作脚本测试环境与直接操作活动目录几乎相同的，但也有些细微差别。ADAM 脚本在绑定字符串中指定主机和端口（即 localhost:389/），而活动目录脚本则不是这样做。

有关类型快捷方式在 PowerShell 中的详细资料，请参阅附录 A 中的“使用 .net 框架”。

参考

- 附录 A 中的“使用 .net 框架”

23.2 创建组织单元

问题

你想在活动目录中创建一个组织单元 (OU)。

解决方案

若要在容器中创建组织单元，可以使用 [adsi] 类型快捷方式来绑定活动目录，然后调用 Create() 方法。

```
$domain = [adsi] "LDAP://localhost:389/dc=Fabrikam,dc=COM"
$salesOrg = $domain.Create("OrganizationalUnit", "OU=Sales")
$salesOrg.Put("Description", "Sales Headquarters, SF")
$salesOrg.Put("wwwHomePage", "http://fabrikam.com/sales")
$salesOrg.SetInfo()
```

讨论

在解决方案的示例中显示了创建一个销售组织单元 (OU) 机构的根。你可以使用相同的语法规来创建其他 OU 及子 OU。例 23-1 演示如何创建东部和西部的销售部门。

例 23-1：创建东部和西部的销售部门

```
$sales =[adsi ] ""LDAP://localhost:389/ou=Sales,dc=Fabrikam,dc=COM"

$east =$sales.Create("OrganizationalUnit", "OU=East")
$east.Put("wwwHomePage", "http://fabrikam.com/sales/east")
$east.SetInfo()

$west =$sales.Create("OrganizationalUnit", "OU=West")
$west.Put("wwwHomePage", "http://fabrikam.com/sales/west")
$west.SetInfo()
```

若要查看已创建的这些 OU，请参阅 23.5 节。

参考

- 23.5 节

23.3 获取组织单元的属性

问题

你想获取并列出一个特定 OU 的属性。

解决方案

列出OU的属性使用[adsi]类型快捷方式。若要列出一个OU的属性，可以使用[adsi]类型快捷方式将其绑定到活动目录的 OU 中，并且把该OU传递给Format-List cmdlet：

```
$organizationalUnit =  
[adsi] "LDAP://localhost:389/ou=West,ou=Sales,dc=Fabrikam,dc=COM"  
  
$organizationalUnit | Format-List *
```

讨论

该解决方案检索西部销售的 OU。默认情况下 Format-List cmdlet 仅显示有代表性的信息，当我们键入 Format-List * 时则显示所有属性。

如果你想知道某个属性的值，可以按名称检索：

```
PS >$organizationalUnit.WWWHomePage  
http://fabrikam.com/sales/west
```

与OU不同，某些活动目录对象类型不可以使用按名称检索它们的属性这种方式。而是你必须调用Get()方法来检索特定的属性：

```
PS >$organizationalUnit.Get("name")  
West
```

23.4 修改组织单元的属性

问题

你想修改一个特定的 OU 的属性。

解决方案

若要修改一个 OU 的属性，可以使用[adsi]类型快捷方式将其绑定到活动目录的 OU 中，然后调用Put()方法。最后调用SetInfo()方法来适用于所做的更改。

```
$organizationalUnit =  
    [adsi] "LDAP://localhost:389/ou=West,ou=Sales,dc=Fabrikam,dc=COM"  
  
$organizationalUnit.Put("Description", "Sales West Organization")  
$organizationalUnit.SetInfo()
```

讨论

该解决方案检索西部销售的 OU：先说明设置为西部销售组织，然后将这些更改应用于该活动目录。

23.5 获取一个活动目录容器的子集

问题

你想列出某个活动目录容器的所有子级。

解决方案

若要列出某个容器中的项，可以使用[adsi]类型快捷方式将其绑定到活动目录的 OU 中，然后访问该容器的 PsBase.Children 属性：

```
$sales =  
    [adsi] "LDAP://localhost:389/ou=Sales,dc=Fabrikam,dc=COM"  
$sales.PsBase.Children
```

讨论

该解决方案列出销售 OU 的所有子级。你通常得到的信息的级别是在 ADSIEDIT MMC 管理单元中选择的一个节点。如果你想筛选的信息仅包括显示用户、其他组织单位或更复杂的查询，请参阅 23.8 节。

参考

- 23.8 节（如何在活动目录中筛选特定类型的对象，如向导和驱动器）

23.6 创建用户账户

问题

你想在一个特定的 OU 中创建用户账户。

解决方案

若要在容器中创建用户，请使用 [adsi] 类型快捷方式绑定到该活动目录的 OU 中，然后调用 Create() 方法：

```
$salesWest =  
[adsi] "LDAP://localhost:389/ou=West,ou=Sales,dc=Fabrikam,dc=COM"  
  
$user = $salesWest.Create("User", "CN=MyerKen")  
$user.Put("userPrincipalName", "Ken.Myer@fabrikam.com")  
$user.Put("displayName", "Ken Myer")  
$user.SetInfo()
```

讨论

该解决方案创建在西部销售组织单位下的一个用户。它设置 userPrincipalName (用户的一个唯一标识符)，以及该用户的显示名称。

注意：当您运行脚本对一个真正的活动目录进行部署（而不是一个 ADAM 实例），务必更新 SAMAccountName 属性，或你将得到一个自动生成的默认值。

若要查看已创建这些用户，请参阅 23.5 节。

如果你要批量创建用户，请参阅下面的 23.7 节。

参考

- 23.5 节
- 23.7 节

23.7 编程：批量导入活动目录用户

批量导入活动目录用户，改变了令人厌恶的手写操作（或甚至要一个一个地编写脚本添加每个用户）。若要解决此问题，可以将我们所有的数据存入 CSV，然后执行从 CSV 中批量导入信息。

例 23-2 支持这种灵活的方式。你提供一个容器来保存用户账户和一个 CSV 保存账户信息。脚本对于 CSV 中的每一行数据，都会相应地创建一个用户。只要求 CSV 列定义常见的用户名。任何其他列，如果显示，就表示要为该用户定义其他的活动目录属性。

例 23-2: Import-ADUser.ps1

```
#####
## Import-AdUser.ps1
##
## 在 Active Directory 中创建用户从一个 CSV 的内容。
##
## 例如
## $container = "LDAP://localhost:389/ou=West,ou=Sales,dc=Fabrikam,dc=COM"
## Import-ADUser.ps1 $container .\users.csv
##
## In the user CSV, One column must be named "CN" for the user name.
## All other columns represent properties in Active Directory for that user.
##
## For example:
## CN,userPrincipalName,displayName,manager
## MyerKen,Ken.Myer@fabrikam.com,Ken Myer,
## DoeJane,Jane.Doe@fabrikam.com,Jane Doe,"CN=MyerKen,OU=West,OU=Sales,DC=..."
## SmithRobin,Robin.Smith@fabrikam.com,Robin Smith,"CN=MyerKen,OU=West,OU=..."
##
#####
param(
    $container = $(throw "Please specify a container (such as " +
        "LDAP://localhost:389/ou=West,ou=Sales,dc=Fabrikam,dc=COM")",
    $csvPath = $(throw "Please specify the path to the users CSV")
)

## Bind to the container
$userContainer = [adsi] $container

## Ensure that the container was valid
if(-not $userContainer.Name)
{
    Write-Error "Could not connect to $container"
    return
}

## Load the CSV
$users = @(Import-Csv $csvPath)
if($users.Count -eq 0)
{
    return
}

## Go through each user from the CSV
foreach($user in $users)
{
    ## Pull out the name, and create that user
    $username = $user.CN
    $newUser = $userContainer.Create("User", "CN=$username")

    ## Go through each of the properties from the CSV, and set its value
    ## on the user
    foreach($property in $user.PsObject.Properties)
```

```
    {
        ## Skip the property if it was the CN property that sets the
        ## user name
        if($property.Name -eq "CN")
        {
            continue
        }

        ## Ensure they specified a value for the property
        if(-not $property.Value)
        {
            continue
        }

        ## Set the value of the property
        $newUser.Put($property.Name, $property.Value)
    }

    ## Finalize the information in Active Directory
    $newUser.SetInfo()
}
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节

23.8 搜索用户账户

问题

你想搜索特定的用户账户，但不知道用户的具体名称（DN, distinguished name）。

解决方案

要搜索活动目录中的用户，请使用 [adsi] 类型快捷方式将其绑定到一个包含该用户账户的容器，然后使用 .NET Framework 中的 System.DirectoryServices.DirectorySearcher 类搜索用户：

```
$domain = [adsi] "LDAP://localhost:389/dc=Fabrikam,dc=COM"
$searcher = New-Object System.DirectoryServices.DirectorySearcher $domain
$searcher.Filter = '(&(objectClass=User)(displayName=Ken Myer))'

$userResult = $searcher.FindOne()
$user = $userResult.GetDirectoryEntry()
```

讨论

当你不知道用户账户的完整 DN 时, .NET Framework 中的 System.DirectoryServices.DirectorySearcher 类允许你对其进行搜索。

你提供了 LDAP 筛选器 (在此例中搜索显示名为 ken Myer 的用户), 然后调用 FindOne() 方法。该 FindOne() 方法返回与筛选器相匹配的首次搜索结果, 以便我们检索其实际的活动目录入口。尽管解决方案根据用户的显示名进行搜索, 但你还可以搜索活动目录中的任何字段, userPrincipalName 和 sAMAccountName 是两个不错的选择。当你执行此搜索时, 你会始终尝试缩小搜索域的范围。如果我们知道 Ken Myer 在销售 OU 中, 那么绑定到该 OU 会好一些:

```
$domain = [adsi] "LDAP://localhost:389/ou=Sales,dc=Fabrikam,dc=COM"
```

有关 LDAP 搜索筛选器语法的更多信息, 在 <http://msdn.microsoft.com> 搜索“Search Filter Syntax”。

23.9 获取并列出用户账户的属性

问题

你想获取并列出一个特定的用户账户的属性。

解决方案

若要列出一个用户账户的属性, 请使用 [adsi] 类型快捷方式将其绑定到在活动目录中的用户, 然后把用户传递到 Format-List cmdlet:

```
$user=
[adsi]"LDAP://localhost:389/cn=MyerKen,ou=West,ou=Sales,dc=Fabrikam,dc=COM"
$user | Format-List *
```

讨论

该解决方案从西部销售 OU 中检索 MyerKen 用户。默认情况下 Format-List cmdlet 只显示用户的主要名称 (DN), 我们可以键入 Format-List * 来显示其所有属性。

如果你知道所需值的属性, 就可以按名称指定它:

```
PS >$user.DirectReports
CN=SmithRobin,OU=West,OU=Sales,DC=Fabrikam,DC=COM
CN=DoeJane,OU=West,OU=Sales,DC=Fabrikam,DC=COM
```

与用户不同，某些类型的活动目录对象不允许你按名称这种方式检索它们的属性。你可以调用 `Get()` 方法来获得特定属性：

```
PS >$user.Get("userPrincipalName")
Ken.Myer@fabrikam.com
```

23.10 修改用户账户的属性

问题

如何修改特定用户账户属性。

解决方案

若要修改用户账户，可以使用 `[adsi]` 类型快捷方式绑定到在活动目录中的用户，再调用 `Put()` 方法修改属性。最后，调用 `SetInfo()` 方法，应用所做的更改。

```
$user =
[adsi] "LDAP://localhost:389/cn=MyerKen,ou=West,ou=Sales,dc=Fabrikam,dc=COM"

$user.Put("Title", "Sr. Exec. Overlord")
$user.SetInfo()
```

讨论

该解决方案从西部销售OU中检索 MyerKen 用户。然后设置用户的标题为 Sr. Exec. Overlord 并将这些更改应用于活动目录。

23.11 创建一个安全组或分布组

问题

你想创建一个安全组或分布组。

解决方案

若要创建一个安全组或分布组，可以使用 `[adsi]` 类型快捷方式来绑定到活动目录中的一个容器中，然后调用 `Create()` 方法：

```
$salesWest =
[adsi] "LDAP://localhost:389/ou=West,ou=Sales,dc=Fabrikam,dc=COM"
```

```
$management = $salesWest.Create("Group", "CN=Management")
$management.SetInfo()
```

讨论

该解决方案在西部销售 OU 中创建一个名为 Management 的组。

注意：当你对一个真正的活动目录部署（相对 A D A M 实例）运行脚本时，务必更新 sAMAccountName 属性，否则你将得到一个自动生成的默认值。

当你在活动目录中创建一个组时，通常设置 groupType 属性来定义该组的类型。若要指定组的类型，使用 -bor 运算符来合并组标志，并使用其结果为 groupType 的属性。例 23-3 定义组作为一个全球化的启用安全机制的组。

例 23-3：建立一个自定义的活动目录组类型

```

$ADS_GROUP_TYPE_GLOBAL_GROUP = 0x00000002
$ADS_GROUP_TYPE_DOMAIN_LOCAL_GROUP = 0x00000004
$ADS_GROUP_TYPE_LOCAL_GROUP = 0x00000004
$ADS_GROUP_TYPE_UNIVERSAL_GROUP = 0x00000008
$ADS_GROUP_TYPE_SECURITY_ENABLED = 0x80000000

$salesWest =
    [adsi] "LDAP://localhost:389/ou=West,ou=Sales,dc=Fabrikam,dc=COM"

$groupType = $ADS_GROUP_TYPE_SECURITY_ENABLED -bor
    $ADS_GROUP_TYPE_GLOBAL_GROUP

$management = $salesWest.Create("Group", "CN=Management")
$management.Put("groupType", $groupType)
$management.SetInfo()

```

如果你要从CSV中的数据批量创建组，23.7节中的Import-ADUser脚本提供了一个不错的开始。若要使该脚本创建组（而不是用户），请更改下面这行：

```
$newUser = $userContainer.Create("User", "CN=$username")
```

改为：

```
$newUser = $userContainer.Create("Group", "CN=$username")
```

如果你更改脚本来批量创建组，更改该变量名称也很有帮助的，names（\$user、\$users、\$username 和 \$newUser）对应于组相关的 names：\$group、\$groups、\$groupname 和 \$newgroup。

参考

- 23.7 节

23.12 搜索一个安全组或分布组

问题

你想搜索特定的组，但不知道它的 DN。

解决方案

要搜索一个安全组或分布组，可以使用 [adsi] 类型快捷方式将其绑定到活动目录中包含组的容器，然后使用 .NET Framework 中的 System.DirectoryServices.DirectorySearcher 类搜索该组：

```
$domain = [adsi] "LDAP://localhost:389/dc=Fabrikam,dc=COM"
$searcher = New-Object System.DirectoryServices.DirectorySearcher $domain
$searcher.Filter = '(&(objectClass=Group)(name=Management))'

$groupResult = $searcher.FindOne()
$group = $groupResult.GetDirectoryEntry()
```

讨论

当你不知道组的完整 DN，.NET Framework 中的 System.DirectoryServices.DirectorySearcher 类允许你对其进行搜索。

提供 LDAP 筛选器（在此例，搜索具有 Management 名称的组），然后调用 FindOne() 方法。该 FindOne() 方法返回与筛选器相匹配第一次搜索的结果，以便我们检索其实际的活动目录入口。尽管解决方案只搜索组的名称，但你可以搜索任何活动目录中的字段， mailNickname 和 sAMAccountName 是两个不错的选择。

当你执行此搜索时，始终尝试缩小搜索域的范围。如果我们知道 Management 组是在销售 OU 中，将它改为绑定到该 OU 会好一些：

```
$domain = [adsi] "LDAP://localhost:389/ou=Sales,dc=Fabrikam,dc=COM"
```

有关 LDAP 搜索筛选器语法的更多信息，请搜索 <http://msdn.microsoft.com> 有关“Search Filter Syntax”的内容。

23.13 获取一个组的属性

问题

你想获取并列出一个特定的安全组或分布组的属性。

解决方案

若要列出组的属性，可以使用 [adsi] 类型快捷方式将其绑定到活动目录中的组，然后将组传递给 Format-List cmdlet：

```
$group=  
[adsi]"LDAP://localhost:389/cn=Management,ou=West,ou=Sales,dc=Fabrikam,dc=COM"  
  
$group | Format-List *
```

讨论

该解决方案从西部销售 OU 中检索 Management 组。默认情况下 Format-List cmdlet 只显示组的 DN，因此我们键入 Format-List * 就可以显示所有属性。

如果你知道所需值的属性，可以按名称指定它：

```
PS >$group.Member  
CN=SmithRobin,OU=West,OU=Sales,DC=Fabrikam,DC=COM  
CN=MyerKen,OU=West,OU=Sales,DC=Fabrikam,DC=COM
```

与组不同，某些类型的活动目录对象不支持属性名检索这种方式。你必须调用 Get() 方法检索特定的属性：

```
PS >$group.Get("name")  
Management
```

23.14 查找用户组的所有者

问题

你想获得一个安全组或分布组的所有者。

解决方案

若要确定一个用户组的所有者，可以使用 [adsi] 类型快捷方式绑定到活动目录中的组，然后检索该 ManagedBy 属性：

```
$group= [adsi]"LDAP://localhost:389/cn=Management,ou=West,ou=Sales,dc=Fabrikam,dc=COM"  
$group.ManagedBy
```

讨论

该解决方案从西部销售 OU 中检索 Management 组的所有者。为此，它访问该组的 ManagedBy 属性。此属性只有组填充管理员时存在，但它这么做是最优的方法。与组不同，活动对象的某些类型不允许你通过属性名检索它们，而是必须调用 Get() 方法来检索特定的属性：

```
PS >$group.Get("name")  
Management
```

23.15 修改安全组或分布组的属性

问题

你想修改一个特定的安全组或分布组的属性。

解决方案

若要修改一个安全组或分布组，可以使用 [adsi] 类型快捷方式绑定到活动目录中的组，然后调用 Put() 方法修改属性。最后调用 SetInfo() 方法来应用于所做的更改。

```
$group =
    [adsi] "LDAP://localhost:389/cn=Management,ou=West,ou=Sales,dc=Fabrikam,dc=COM"
PS >$group.Put("Description", "Managers in the Sales West Organization")
PS >$group.SetInfo()
```

讨论

该解决方案从西部销售 OU 中检索 Management 组。然后为西部销售组中的 Managers 设置描述，然后应用于对活动目录的更改。

23.16 将用户添加到安全组或分布组

问题

你想将用户添加到安全组或分布组。

解决方案

若要将用户添加到一个安全组或分布组，可以使用 [adsi] 类型快捷方式绑定到活动目录中的组，然后调用 Add() 方法：

```
$management =  
[adsi]"LDAP://localhost:389/cn=Management,ou=West,ou=Sales,dc=Fabrikam,dc=COM"  
  
$user = "LDAP://localhost:389/cn=MyerKen,ou=West,ou=Sales,dc=Fabrikam,dc=COM"  
$management.Add($user)
```

讨论

该解决方案将用户 MyerKen 添加到西部销售 OU 中的一个名为 Management 的组中。若要查看你是否已成功地添加该用户，请参阅 23.18 节。

参考

- 23.18 节

23.17 从安全组或分布组中删除用户

问题

你想从安全组或分布组中删除用户。

解决方案

若要从一个安全组或分布组中移除用户，使用 [adsi] 类型快捷方式绑定到活动目录中的组，然后调用 Remove() 方法：

```
$management =  
[adsi]"LDAP://localhost:389/cn=Management,ou=West,ou=Sales,dc=Fabrikam,dc=COM"  
  
$user = "LDAP://localhost:389/cn=MyerKen,ou=West,ou=Sales,dc=Fabrikam,dc=COM"  
$management.Remove($user)
```

讨论

该解决方案从西部销售 OU 中名为 Management 的组中移除 MyerKen 用户。

若要查看你是否已成功地删除该用户，请参阅下面的 23.18 节。

参考

- 23.18 节

23.18 列出用户的组成员身份

问题

如何列出用户的组成员身份。

解决方案

要列出用户的组成员身份，可以使用 [adsi] 类型快捷方式将其绑定到活动目录中的用户，然后访问 MemberOf 属性：

```
$user =  
[adsi] "LDAP://localhost:389/cn=MyerKen,ou=West,ou=Sales,dc=Fabrikam,dc=COM"  
$user.MemberOf
```

讨论

该解决方案列出用户 MyerKen 所属的所有组。由于活动目录把此信息存储为用户属性，这只是检索用户特定信息的一个示例。有关检索有关用户的信息的详细信息，请参阅 23.9 节。

参考

- 23.9 节

23.19 列出一组的所有成员

问题

你想列出一组的所有成员。

解决方案

要列出组的成员，使用 [adsi] 类型快捷方式将其绑定到的活动目录中的组，然后访问 Member 属性：

```
$group= [adsi]"LDAP://localhost:389/cn=Management,ou=West,ou=Sales,dc=Fabrikam,dc=COM"  
$group.Member
```

讨论

该解决方案列出西部销售 OU 中组 Management 的所有成员。由于活动目录把此信息存储为组的属性，这只是检索组特定信息的一个示例。有关组检索的详细信息，请参阅 23.13 节。

参考

- 23.13 节

23.20 列出一个组织单元的所有用户

问题

你想列出一个 OU 中的所有用户。

解决方案

若要列出一个 OU 中的用户，使用 [adsi] 类型快捷方式将其绑定到活动目录中的 OU。为该 OU 创建一个新的 System.DirectoryServices.DirectorySearcher 然后将其筛选器属性设置为 (objectClass = User)。最后，调用搜索器的 FindAll() 方法以执行搜索。

```
$sales =  
[adsi]"LDAP://localhost:389/ou=Sales,dc=Fabrikam,dc=COM"  
  
$searcher = New-Object System.DirectoryServices.DirectorySearcher $sales  
$searcher.Filter = '(objectClass=User)'  
$searcher.FindOne()
```

讨论

该解决方案列出销售 OU 中的所有用户。它通过 .NET Framework 的 System.DirectoryServices.DirectorySearcher 类执行此操作，为了方便你查询活动目录，该筛选器属性指定了一个 LDAP 筛选器字符串。

注意：默认情况下，一个 DirectorySearcher 搜索对给定的容器和它下面的所有容器。设置 SearchScope 属性可以更改此默认设置。Base 只搜索当前容器，OneLevel 只搜索直接子容器。

有关在 .NET Framework 中使用类的更多信息，请参阅 3.4 节。

参考

- 3.4 节

23.21 搜索计算机账户

问题

你想搜索特定的计算机账户，但不知道它的 DN。

解决方案

要搜索计算机账户，可以使用 [adsi] 类型快捷方式将绑定到一个存放活动目录中账户的容器，然后使用 .NET Framework 中的 System.DirectoryServices.DirectorySearcher 类来搜索账户：

```
$domain = [adsi] "LDAP://localhost:389/dc=Fabrikam,dc=COM"
$searcher = New-Object System.DirectoryServices.DirectorySearcher $domain
$searcher.Filter = '(&(objectClass=Computer)(name=kenmyer_laptop))'
$computerResult = $searcher.FindOne()
$computer = $computerResult.GetDirectoryEntry()
```

讨论

当你不知道计算机账户的完整 DN 时，.NET Framework 中的 System.DirectoryServices.DirectorySearcher 类允许您对其进行搜索。

你提供 LDAP 篩选器（在此例，搜索名为 kenmyer_laptop 的计算机），然后调用 FindOne() 方法。该 FindOne() 方法返回与筛选器相匹配的第一个搜索结果，以便我们检索其实际的活动目录项。尽管解决方案是根据名称搜索计算机，但你可以搜索活动目录中的任何字段—— sAMAccountName 和操作系统特征 (operatingSystem, operatingSystemVersion, operatingSystemServicePack) 是不错的选择。

当你执行此搜索时，始终尝试缩小搜索域的范围。如果你知道该计算机是在销售 OU 中，将它绑定到该 OU 会好一些：

```
$domain = [adsi] "LDAP://localhost:389/ou=Sales,dc=Fabrikam,dc=COM"
```

有关 LDAP 搜索筛选器语法的更多信息，搜索 <http://msdn.microsoft.com> 有关“Search Filter Syntax”。

23.22 获取并列出一台计算机账户的属性

问题

你想获取并列出一个特定的计算机账户的属性。

解决方案

若要列出一个计算机账户的属性，使用[adsi]类型快捷方式将其绑定到活动目录中的计算机，然后传递该计算机到Format-List cmdlet：

```
$computer = [adsi] "LDAP://localhost:389/cn=kenmyer_laptop,ou=West,ou=Sales,dc=Fabrikam,dc=COM"
```

```
$computer | Format-List *
```

讨论

该解决方案从西部销售 OU 中检索 kenmyer_laptop 计算机。默认的，Format-List cmdlet 只显示计算机的名称，因此我们可以键入 Format-List * 来显示所有属性。

如果你知道所选值的属性，可以按名称指定它：

```
PS >$computer.OperatingSystem  
Windows Server 2003
```

与用户不同，某些类型的活动目录对象不支持按名称的这种方式检索它们的属性，而必须调用 Get() 方法来检索特定属性：

```
PS >$user.Get("operatingSystem")  
Windows Server 2003
```

第 24 章

企业级计算机管理

24.0 简介

当你在企业中用 Windows 管理系统时，经常会遇到的问题是“如何在 PowerShell 中完成这项任务”。管理员心目中理想的工作环境应该是这样的，为用户设计的管理功能还应该可以（通过 PowerShell cmdlet）管理相应的功能。许多管理任务都比 PowerShell 长，因此有时回答是，在 PowerShell 之前使用了相同的方法。

但这并不是说有了 PowerShell 就可以提高你作为管理员的工作效率。Pre-PowerShell 管理任务通常属于下面的几种模型之一：命令行实用程序、Windows Management Instrumentation (WMI) 交互、注册表操作、文件操作、与 COM 对象交互或与 .NET 对象之间的交互。

PowerShell 可以更容易地与所有这些任务模型进行交互，并因此使管理它们相应的功能更加容易。

24.1 编程：列出用户登录或注销的脚本

Windows 组策略系统在注册表下的键 HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Group Policy\State\<User SID>\Scripts\Logon 和 HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Group Policy\State\<User SID>\Scripts\Logoff 中存储登录和注销脚本。每个键有一个子项应用于组策略对象。这些子键都有另一个级别的键对应于该计算机的单个脚本。不知道用户的 SID 可能使访问这些脚本变得困难。例 24-1 自动将用户名和 SID 以及要访问此信息所需要的注册表操作任务进行映射。

例 24-1：Get-UserLogonLogoffScript.ps1

```
#####
## Get-UserLogonLogoffScript.ps1
##
## Get the logon or logoff scripts assigned to a specific user
##
## ie:
## PS >Get-UserLogonLogoffScript LEE-DESK\LEE Logon
## Get-UserLogonLogoffScript.ps1 -User LEE-DESK\LEE -Type Logon
#####
param(
    $username = $(throw "Please specify a username"),
    $scriptType = $(throw "Please specify the script type")
)

## Verify that they've specified a correct script type
$scriptOptions = "Logon", "Logoff"
if($scriptOptions -notcontains $scriptType)
{
    $error = "Cannot convert value {0} to a script type. " +
        "Specify one of the following values and try again. " +
        "The possible values are ""{1}""."
    $ofs = ","
    throw ($error -f $scriptType, ([string] $scriptOptions))
}

## Find the SID for the username
$account = New-Object System.Security.Principal.NTAccount $username
$sid =
    $account.Translate([System.Security.Principal.SecurityIdentifier]).Value

## Map that to their group policy scripts
$registryKey = "HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\" +
    "Group Policy\State\$sid\Scripts"

## Go through each of the policies in the specified key
foreach($policy in Get-ChildItem $registryKey\$scriptType)
{
    ## For each of the scripts in that policy, get its script name
    ## and parameters
    foreach($script in Get-ChildItem $policy.PsPath)
    {
        Get-ItemProperty $script.PsPath | Select Script,Parameters
    }
}
```

有关在 PowerShell 中使用 Windows 注册表的更多信息，请参阅第 18 章。有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节
- 第 18 章 Windows 注册表

24.2 编程：列出计算机启动或关机脚本

Windows 组策略系统在注册表项 HKLM:\SOFTWARE\Policies\Microsoft\Windows\System\Scripts\Startup 和 HKLM:\SOFTWARE\Policies\Microsoft\Windows\System\Scripts\Shutdown 下存储启动和关机脚本。每个键有一个子键应用于组策略对象。每个子键都有另一个级别的键对应于该计算机的单个脚本。例 24-2 允许你轻松地检索和访问计算机启动和关机脚本。

例 24-2: Get-MachineStartupShutdownScript.ps1

```
#####
## Get-MachineStartupShutdownScript.ps1
## Get the startup or shutdown scripts assigned to a machine
## ie:
## PS >Get-MachineStartupShutdownScript Startup
#####

param(
    $scriptType = $(throw "Please specify the script type")
)
## Verify that they've specified a correct script type
$scriptOptions = "Startup", "Shutdown"
if($scriptOptions -notcontains $scriptType)
{
    $error = "Cannot convert value {0} to a script type. " +
        "Specify one of the following values and try again. " +
        "The possible values are ""{1}""."
    $ofs = ","
    throw ($error -f $scriptType, ([string] $scriptOptions))
}

## Store the location of the group policy scripts for the machine
$registryKey = "HKLM:\SOFTWARE\Policies\Microsoft\Windows\System\Scripts"
## Go through each of the policies in the specified key
foreach($policy in Get-ChildItem $registryKey\$scriptType)
{
    ## For each of the scripts in that policy, get its script name
```

```
## and parameters
foreach($script in Get-ChildItem $policy.PsPath)
{
    Get-ItemProperty $script.PsPath | Select Script,Parameters
}
}
```

有关在 PowerShell 中使用 Windows 注册表的更多信息，请参阅第 18 章。有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节
- 第 18 章 Windows 注册表

24.3 启用或禁用 Windows 防火墙

问题

你想启用或禁用 Windows 防火墙。

解决方案

若要管理 Windows 防火墙，可以使用 HNetCfg.FwMgr COM 对象的 LocalPolicy.CurrentProfile.FirewallEnabled 属性：

```
PS >$firewall = New-Object -com HNetCfg.FwMgr
PS >$firewall.LocalPolicy.CurrentProfile.FirewallEnabled = $true
PS >$firewall.LocalPolicy.CurrentProfile.FirewallEnabled
True
```

讨论

在 Windows XP SP 2 和更高版本中，HNetCfg.FwMgr COM 对象提供编程方式访问 Windows 防火墙。LocalPolicy.CurrentProfile 属性提供了大部分的功能。

关于通过 COM API 管理 Windows 防火墙的更多信息，可以访问 <http://msdn.microsoft.com>，然后搜索“Using Windows Firewall API”。文档中的例子是用 VBScript 编写的，是对可用的功能的一个有用的概述。

如果你不熟悉文档中的 VBScript 的特定部分，Microsoft 脚本中心会提供有用的指导来帮助你将 VBScript 转换为 PowerShell。你可以在 <http://www.microsoft.com/technet/scriptcenter/topics/winpsn/convert/default.mspx> 找到该文档。

有关使用 PowerShell 中的 COM 对象的详细信息，请参阅 15.6 节。

参考

- 15.6 节

24.4 打开或关闭 Windows 防火墙中的端口

问题

你想打开或关闭 Windows 防火墙中的端口。

解决方案

若要打开或关闭 Windows 防火墙中的端口，可以使用 HNetCfg.FwMgr COM 对象的 LocalPolicy.CurrentProfile.GloballyOpenPorts 集合。

要添加一个端口，创建一个 HNetCfg.FWOpenPort COM 对象来表示该端口，然后将其添加到 GloballyOpenPorts 集合：

```
$PROTOCOL_TCP = 6
$firewall = New-Object -com HNetCfg.FwMgr
$port = New-Object -com HNetCfg.FWOpenPort
$port.Name = "Webserver at 8080"
$port.Port = 8080
$port.Protocol = $PROTOCOL_TCP

$firewall.LocalPolicy.CurrentProfile.GloballyOpenPorts.Add($port)
```

若要关闭一个端口，从 GloballyOpenPorts 集合中删除他：

```
$PROTOCOL_TCP = 6
$firewall.LocalPolicy.CurrentProfile.GloballyOpenPorts.Remove(8080, $PROTOCOL_TCP)
```

讨论

在 Windows XP SP 2 和更高版本中，HNetCfg.FwMgr COM 对象提供编程访问 Windows 防火墙。LocalPolicy.CurrentProfile 属性提供了大部分的功能。

关于通过 COM API 管理 Windows 防火墙的更多信息，可以访问 <http://msdn.microsoft.com> 然后搜索“Using Windows Firewall API”。文档中的例子是用 VBScript 编写的，是对可用的功能的一个有用的概述。

如果你不熟悉文档中的 VBScript 的特定部分，Microsoft 脚本中心提供有用的指导来帮助你将 VBScript 转换为 PowerShell。你可以在 <http://www.microsoft.com/technet/scriptcenter/topics/winps/convert/default.mspx> 找到该文档。

有关使用 PowerShell 中的 COM 对象的详细信息，请参阅 15.6 节。

参考

- 15.6 节

24.5 “编程：列出所有已安装的软件”

查找有关当前已经安装的软件的信息的最佳位置是 HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall 注册表项。这里存储了有关如何卸载该软件的信息。

该注册表项的每个子项代表一个传统上可以通过在控制面板中添加/删除程序项卸载的软件。除了应用程序的显示名，通常还存在其他有用的属性（具体取决于应用程序）。示例包括 Publisher、UninstallString 和 HelpLink。

若要查看所有你的系统上安装的软件的属性，可以键入以下命令：

```
$properties = Get-InstalledSoftware |  
    Foreach-Object { $_.PsObject.Properties }  
  
$properties | Select-Object Name | Sort-Object -Unique Name
```

这将列出至少一个已安装的应用程序的所有属性（尽管非常有限且被所有已安装的应用程序共享）。

若要使用此数据，你首先要检索它。例 24-3 提供了一个脚本以列出当前系统所有的安装软件，返回的所有信息作为 PowerShell 对象的属性。

例 24-3：Get-InstalledSoftware.ps1

```
#####  
## Get-InstalledSoftware.ps1  
##  
## List all installed software on the current computer.
```

```
## 本脚本的目的是显示所有已安装的 Windows API。MO 块由作者编写。
## ie:
## PS >Get-InstalledSoftware PowerShell
##
##### param(
## $displayName = ".*"
## )
## Get all the listed software in the Uninstall key
$keys = Get-ChildItem HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall
## Get all of the properties from those items
$item = $keys | Foreach-Object { Get-ItemProperty $_.PsPath }
## For each of those items, display the DisplayName and Publisher
foreach($item in $items)
{
    if(($item.DisplayName) -and ($item.DisplayName -match $displayName))
    {
        $item
    }
}
```

有关在 PowerShell 中使用 Windows 注册表的更多信息，请参阅第 18 章。有关运行脚本的更多信息，请参见 1.1 节。

参考

- 1.1 节
- 第 18 章 Windows 注册表

24.6 卸载应用程序

问题

你想卸载一个指定的软件。

要卸载一个应用程序，可以使用 24.5 节中提供的 Get-InstalledSoftware 脚本来检索要卸载软件的命令。由于 UninstallString 使用批处理文件的语法，因此可以使用 cmd.exe 来启动某个程序的卸载：

```
PS > $software = Get-InstalledSoftware UnwantedProgram  
PS > cmd /c $software.UninstallString
```

或者，可以使用 Win32_Product WMI 类进行全自动的安装：

```
$application = Get-WmiObject Win32_Product -filter "Name='UnwantedProgram'"  
$application.Uninstall()
```

讨论

应用程序提供的UninstallString启动的交互式体验与你通过控制面板中添加/删除程序项卸载该应用程序是一致的。如果你要以全自动的方式删除该软件，你有两种选择：使用应用程序的卸载程序的“安静模式”（例如，/quiet切换到msiexec.exe），或使用如解决方案中所示的 Win32_Product WMI 类的软件移除功能。

有关使用 PowerShell 中 WMI 的详细信息，请参阅 15.1 节。

参考

- 15.1 节
- 24.5 节

24.7 管理计算机上的计划任务

问题

如何管理计算机上的计划任务。

解决方案

若要管理计划任务，可以使用 schtasks.exe 应用程序。

若要查看计划任务的列表，可以使用下面的命令：

```
PS >schtasks  
  
TaskName          Next Run Time      Status  
=====            ======           =====  
Defrag C          03:00:00, 5/21/2007  
User_Feed_Synchronization-{CA4D6D9C- 18:34:00, 5/20/2007  
User_Feed_Synchronization-{CA4D6D9C- 18:34:00, 5/20/2007
```

如果要在每天 3 点进行磁盘碎片整理的任务，可以使用下面的命令：

```
schtasks /create /tn "Defrag C" /sc DAILY  
/st 03:00:00 /tr "defrag c:" /ru Administrator
```

如要按名称删除计划的任务，可以使用下面的命令：

```
schtasks /delete /tn "Defrag C"
```

讨论

该解决方案中的示例告诉系统每一天在上午 3:00 对磁盘 C: 进行碎片整理。由于 *defrag.exe* 命令需要管理特权运行，所以此命令运行在管理员账户下。除了可以安排在本地计算机上的任务，*schtasks.exe* 应用程序还允许你在远程计算机上安排任务。

在 Windows Vista 中，*schtasks.exe* 应用程序已被增强，可以支持事件触发器、条件和其他设置。有关 *schtasks.exe* 应用程序的详细信息，请键入 **schtasks /?**。

24.8 检索打印机信息

问题

如何获取当前系统上有关打印机的信息。

解决方案

若要检索连接到系统的有关打印机的信息，可以使用 Win32_Printer WMI 类：

```
PS >Get-WmiObject Win32_Printer | Select-Object Name,PrinterStatus  
Name  
----  
Microsoft Office Document Image Wr...  
Microsoft Office Document Image Wr...  
CutePDF Writer  
Brother DCP-1000
```

若要检索有关特定打印机的信息，可以根据打印机的名称来应用一个筛选器：

```
PS >$device = Get-WmiObject Win32_Printer -Filter "Name='Brother DCP-1000'"  
PS >$device | Format-List *  
  
Status : Unknown  
Name : Brother DCP-1000  
Attributes : 588  
Availability :  
AvailableJobSheets :  
AveragePagesPerMinute : 0
```

```
    Capabilities           : {4, 2, 5}
    CapabilityDescriptions : {Copies, Color, Collate}
    Caption                : Brother DCP-1000
    (...)
```

若检索特定的属性，你可以像访问其他 PowerShell 对象上的属性那样访问：

```
PS >$device.VerticalResolution  
600  
PS >$device.HorizontalResolution  
600
```

讨论

本解决方案中的例子使用Win32_Printer WMI类来检索计算机上安装的打印机信息。虽然Win32_Printer类提供了访问大多数常用信息的功能，但同时WMI还支持其他几个与打印机相关的类：Win32_TCPIPPrinterPort、Win32_PrinterDriver、CIM_Printer、Win32_PrinterConfiguration、Win32_PrinterSetting、Win32_PrinterController、Win32_PrinterShare和Win32_PrinterDriverDll。

有关使用 PowerShell 中的 WMI 的详细信息，请参阅 15.1 节。

参考

- ## • 15.1 节

24.9 检索打印机队列统计信息

问题

如何获取有关当前系统上的打印机中打印队列的信息。

解决方案

若要检索有关连接到系统的打印机的信息，可以使用 Win32_PerfFormattedData_Spooler_PrintQueue WMI 类：

```
PS >Get-WmiObject Win32_PerfFormattedData_Spooler_PrintQueue |  
    >> Select Name,TotalJobsPrinted  
    >>
```

Name	TotalJobsPrinted
Microsoft Office Document Image Wr...	0
Microsoft Office Document Image Wr...	0
CutePDF Writer	0
Brother DCP-1000	2
_Total	2

若要检索特定打印机的信息，可以应用基于其名称的筛选器，如例 24-4 所示。

例 24-4：检索特定打印机的信息

```
PS >$queueClass = "Win32_PerfFormattedData_Spooler_PrintQueue"
PS >$filter = "Name='Brother DCP-1000'"
PS >$stats = Get-WmiObject $queueClass -Filter $filter
PS >$stats | Format-List *
```

AddNetworkPrinterCalls	: 129
BytesPrintedPersec	: 0
Caption	:
Description	:
EnumerateNetworkPrinterCalls	: 0
Frequency_Object	:
Frequency_PerfTime	:
Frequency_Sys100NS	:
JobErrors	: 0
Jobs	: 0
JobsSpooling	: 0
MaxJobsSpooling	: 1
MaxReferences	: 3
Name	: Brother DCP-1000
NotReadyErrors	: 0
OutofPaperErrors	: 0
References	: 2
Timestamp_Object	:
Timestamp_PerfTime	:
Timestamp_Sys100NS	:
TotalJobsPrinted	: 2
TotalPagesPrinted	: 0

若要检索特定的属性，可以像访问其他 PowerShell 对象上的属性那样进行访问：

```
PS >$stats.TotalJobsPrinted
2
```

讨论

`Win32_PerfFormattedData_Spooler_PrintQueue` WMI 类提供了对各种与打印队列相关联的 Windows 性能计数器的访问。因此，你也可以通过 .NET Framework 访问它们，如 15.8 节中所述：

```
$printer = "Brother DCP-1000"
$pc = New-Object Diagnostics.PerformanceCounter "Print Queue", "Jobs", $printer
$pc.NextValue()
```

有关使用 PowerShell 中的 WMI 的详细信息，请参阅 15.1 节。

参考

- 15.1 节
- 15.8 节

24.10 管理打印机和打印队列

问题

如何从打印机中清除挂起的作业。

解决方案

若要管理连接到系统上的打印机，可以使用 Win32_Printer WMI 类。默认情况下，WMI 类列出所有打印机：

```
PS >Get-WmiObject Win32_Printer | Select-Object Name,PrinterStatus
```

Name	PrinterStatus
Microsoft Office Document Image Wr...	3
Microsoft Office Document Image Wr...	3
CutePDF Writer	3
Brother DCP-1000	3

若要清除特定打印机的打印队列，可以根据它的名称进行筛选，并调用 CancelAllJobs() 方法：

```
PS >$device = Get-WmiObject Win32_Printer -Filter "Name='Brother DCP-1000'"
PS >$device.CancelAllJobs()
```

```
__GENUS      : 2
__CLASS     : __PARAMETERS
__SUPERCLASS: 
__DYNASTY   : __PARAMETERS
__RELPATH   : 
__PROPERTY_COUNT: 1
__DERIVATION: {}
__SERVER    :
```

```
__NAMESPACE      : 
__PATH          : 
ReturnValue     : 5
```

讨论

该解决方案中的例子使用 Win32_Printer WMI 类取消一个打印机的所有作业。除了取消所有打印作业，Win32_Printer 类还支持其他任务：

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

TypeName: System.Management.ManagementObject#root\cimv2\Win32_Printer

Name           MemberType Definition
----           -----
CancelAllJobs Method   System.Management.ManagementBaseObject Can...
Pause          Method   System.Management.ManagementBaseObject Pau...
PrintTestPage Method   System.Management.ManagementBaseObject Pri...
RenamePrinter Method   System.Management.ManagementBaseObject Ren...
Reset          Method   System.Management.ManagementBaseObject Res...
Resume         Method   System.Management.ManagementBaseObject Res...
SetDefaultPrinter Method System.Management.ManagementBaseObject Set...
SetPowerState   Method   System.Management.ManagementBaseObject Set...
```

有关使用 PowerShell 中的 WMI 的详细信息，请参阅 15.1 节。

参考

- 15.1 节

24.11 确定是否安装了补丁程序

问题

如何确定是否在系统上安装了特定的补丁程序。

解决方案

若要检索应用于系统的修复程序的列表，可以使用 Win32_QuickfixEngineering WMI 类：

```
PS >Get-WmiObject Win32_QuickfixEngineering -Filter "HotFixID='KB925228'"

Description      : Windows PowerShell(TM) 1.0
FixComments     :
```

```
HotFixID      : KB925228
Install Date   :
InstalledBy    :
InstalledOn    :
Name          :
ServicePackInEffect : SP3
Status        :
```

若要确定是否应用了一个特定的补丁程序，可以使用例 24-5 中提供的脚本 Test-HotfixInstallation:

```
PS >Test-HotfixInstallation KB925228 LEE-DESK
True
PS >Test-HotfixInstallation KB92522228 LEE-DESK
False
```

讨论

例 24-5 可以确定是否在特定的系统上安装了修补程序。它使用 Win32_QuickfixEngineering WMI 类来检索此信息。

例 24-5：Test-HotfixInstallation.ps1

```
#####
## Test-HotfixInstallation.ps1
##
## Determine if a hotfix is installed on a computer
##
## ie:
## PS >Test-HotfixInstallation KB925228 LEE-DESK
## True
##
param(
    $hotfix = $(throw "Please specify a hotfix ID"),
    $computer = "."
)

## Create the WMI query to determine if the hotfix is installed
$filter = "HotFixID='$hotfix'"
$results = Get-WmiObject Win32_QuickfixEngineering ` 
    -Filter $filter -Computer $computer

## Return the results as a boolean, which tells us if the hotfix is installed
[bool] $results
```

有关使用 PowerShell 中的 WMI 的详细信息，请参阅 15.1 节。有关运行脚本的详细信息，请参阅 1.1 节。

参考

- 1.1 节
- 15.1 节

24.12 编程：汇总系统信息

Windows 管理规范（WMI，Windows Management Instrumentation）提供了有关当前系统或远程系统的大量的信息。事实上，传统的用来收集系统信息的 msinfo32.exe 应用程序很大程度上是基于 WMI 的。

例脚本 24-6 中汇总显示了最常见的信息，但 WMI 提供了更多的内容。有关其他常用的 WMI 类的列表，请参阅附录 F。有关使用 PowerShell 中的 WMI 的详细信息，请参阅 15.1 节。

例 24-6：Get-DetailedSystemInformation.ps1

```
#####
## Get-DetailedSystemInformation.ps1
## Get detailed information about a system.
##
## ie:
## PS >Get-DetailedSystemInformation LEE-DESK > output.txt
##
param(
    $computer = "."
)
#####
[System Information Summary]
Generated $(Get-Date)
#####
[Computer System Information]
Get-WmiObject Win32_ComputerSystem -Computer $computer | Format-List *
#####
[Operating System Information]
Get-WmiObject Win32_OperatingSystem -Computer $computer | Format-List *
```

```
"#""*80
"BIOS Information"
"#""*80
Get-WmiObject Win32_Bios -Computer $computer | Format-List *

"#""*80
"Memory Information"
"#""*80
Get-WmiObject Win32_PhysicalMemory -Computer $computer | Format-List *

"#""*80
"Physical Disk Information"
"#""*80
Get-WmiObject Win32_DiskDrive -Computer $computer | Format-List *

"#""*80
"Logical Disk Information"
"#""*80
Get-WmiObject Win32_LogicalDisk -Computer $computer | Format-List *
```

有关运行脚本的更多信息，请参阅 1.1 节。

参考

- 1.1 节
- 15.1 节
- 附录 F “WMI 参考”

24.13 续订动态主机配置协议租约

问题

如何为计算机上的连接续订 DHCP (Dynamic Host Configuration Protocol) 租约。

解决方案

若要续订 DHCP 租约，可以使用 ipconfig 应用程序。续订所有连接的租约：

```
PS >ipconfig /renew
```

要续订特定的连接租约：

```
PS >ipconfig /renew "Wireless Network Connection 4"
```

讨论

标准ipconfig应用程序很好地管理本地计算机上的网络配置选项。若要续订远程计算机上的租约，你有两个选项。

使用 Win32_NetworkAdapterConfiguration WMI 类

若要续订远程计算机上的租约，可以使用Win32_NetworkAdapterConfiguration WMI类。WMI类需要你知道网络适配器的说明，因此，你要通过查看Get-WmiObject Win32_NetworkAdapterConfiguration -Computer <ComputerName>的输出获得网络适配器的说明：

```
PS >Get-WmiObject Win32_NetworkAdapterConfiguration -Computer LEE-DESK  
(...)  
DHCPEnabled      : True  
IPAddress        : {192.168.1.100}  
DefaultIPGateway : {192.168.1.1}  
DNSDomain        : hsd1.wa.comcast.net.  
ServiceName       : USB_RNDIS  
Description       : Linksys Wireless-G USB Network Adapter with (...)  
Index             : 13  
(...)
```

现在，你已经了解了要续订的适配器，所以你就可以调用它的RenewDHCPLease()方法：

```
$description = "Linksys Wireless-G USB"  
$adapter = Get-WmiObject Win32_NetworkAdapterConfiguration -Computer LEE-DESK |  
    Where-Object { $_.Description -match $description }  
$adapter.RenewDHCPLease()
```

在远程计算机上运行 ipconfig

续订远程计算机上的DHCP租约的另一种方法是使用21.4节提供的解决方案。

```
PS >Invoke-RemoteExpression \\LEE-DESK { ipconfig /renew }
```

有关使用PowerShell中的WMI的详细信息，请参阅15.1节。

参考

- 15.1节
- 21.4节

24.14 分配一个静态 IP 地址

问题

你要将一个静态的 IP 地址分配给一台计算机。

解决方案

使用 Win32_NetworkAdapterConfiguration WMI 类来管理计算机的网络设置：

```
$description = "Linksys Wireless-G USB"
$staticIp = "192.168.1.100"
$subnetMask = "255.255.255.0"
$gateway = "192.168.1.1"

$adapter = Get-WmiObject Win32_NetworkAdapterConfiguration -Computer LEE-DESK |
    Where-Object { $_.Description -match $description}
$adapter.EnableStatic($staticIp, $subnetMask)
$adapter.SetGateways($gateway, [UInt16] 1)
```

讨论

在管理一台计算机的网络设置时，Win32_NetworkAdapter Configuration WMI 类要求你知道网络适配器的说明。可以通过检查的 Get-WmiObject Win32_NetworkAdapterConfiguration -Computer <ComputerName> 的输出来获得该信息：

```
PS >Get-WmiObject Win32_NetworkAdapterConfiguration -Computer LEE-DESK
(...)

DHCPEnabled      : True
IPAddress        : {192.168.1.100}
DefaultIPGateway : {192.168.1.1}
DNSDomain        : hsd1.wa.comcast.net.
ServiceName       : USB_RNDIS
Description       : Linksys Wireless-G USB Network Adapter with (...)

Index            : 13
(...)
```

现在，你已经知道你希望续订哪个适配器了，你就可以调用该对象上的方法，如解决方案中所示。若要再次在一个适配器上启用 DHCP，可以使用 EnableDHCP() 方法：

```
PS >$adapter.EnableDHCP()
```

有关使用 PowerShell 中的 WMI 的详细信息，请参阅 15.1 节。

参考

- 15.1 节

24.15 列出计算机的所有 IP 地址

问题

如何列出计算机上的所有 IP 地址。

解决方案

若要列出分配给一台计算机的 IP 地址，请使用 ipconfig 应用程序：

```
PS >ipconfig
```

讨论

标准 ipconfig 应用程序可以很好地管理本地计算机上的网络配置选项。要查看远程计算机的 IP 地址，你有两个选项。

使用 Win32_NetworkAdapterConfiguration WMI 类

若要查看远程计算机的 IP 地址，可以使用 Win32_NetworkAdapterConfiguration WMI 类。由于列出了所有网络适配器，因此就可以使用 Where-object cmdlet 来只显示那些已经分配 IP 地址的网络适配器：

```
PS >Get-WmiObject Win32_NetworkAdapterConfiguration -Computer LEE-DESK |  
>>     Where-Object { $_.IpEnabled }  
>>  
  
DHCPEnabled      : True  
IPAddress       : {192.168.1.100}  
DefaultIPGateway : {192.168.1.1}  
DNSDomain       : hsd1.wa.comcast.net.  
ServiceName      : USB_RNDIS  
Description      : Linksys Wireless-G USB Network Adapter with SpeedBooster v2 - Packet Scheduler Miniport  
Index            : 13
```

在远程计算机上运行 ipconfig

另一种查看远程计算机的 IP 地址的方法是使用 21.4 节提供的解决方案。

```
PS >Invoke-RemoteExpression \\LEE-DESK {ipconfig}
```

有关使用 PowerShell 中的 WMI 的详细信息，请参阅 15.1 节。

参考

- 15.1 节
- 21.4 节

24.16 列出网络适配器的属性

问题

如何检索计算机上的有关网络适配器的信息。

解决方案

若要检索计算机上的有关网络适配器的信息，可以使用 Win32_NetworkAdapterConfiguration WMI 类：

```
Get-WmiObject Win32_NetworkAdapterConfiguration -Computer <ComputerName>
```

若要列出那些已经分配 IP 地址的网卡的信息，可以使用 Where-object cmdlet 在 IpEnabled 属性上进行筛选：

```
PS >Get-WmiObject Win32_NetworkAdapterConfiguration -Computer LEE-DESK |  
>>     Where-Object { $_.IpEnabled }  
>>  
  
DHCPEnabled      : True  
IPAddress        : {192.168.1.100}  
DefaultIPGateway : {192.168.1.1}  
DNSDomain        : hsd1.wa.comcast.net.  
ServiceName       : USB_RNDIS  
Description       : Linksys Wireless-G USB Network Adapter with SpeedBooster v2 - Packet Scheduler Miniport  
Index             : 13
```

讨论

该解决方案使用 Win32_NetworkAdapterConfiguration WMI 类在给定的系统上检索有关网络适配器的信息。默认情况下，PowerShell 只显示有关网络适配器的最重要的信息，但也提供了对一些其他的信息的访问。

要查看所有可用的信息，可以使用 Format-List cmdlet，如例 24-7 所示。

例 24-7：使用 Format-List cmdlet 查看有关网络适配器的详细的信息

```
PS >$adapter = Get-WmiObject Win32_NetworkAdapterConfiguration |  
>>     Where-Object { $_.IpEnabled }  
>>  
PS >$adapter  
DHCPEnabled      : True  
IPAddress       : {192.168.1.100}  
DefaultIPGateway : {192.168.1.1}  
DNSDomain        : hsd1.wa.comcast.net.  
ServiceName       : USB_RNDIS  
Description       : Linksys Wireless-G USB Network Adapter with SpeedBooster v2 - Packet Scheduler Miniport  
Index            : 13  
  
PS >$adapter | Format-List *  
  
DHCPLeaseExpires          : 20070521221927.000000-420  
Index                     : 13  
Description               : Linksys Wireless-G USB Network Adapter with SpeedBooster v2 - Packet Scheduler Miniport  
DHCPEnabled               : True  
DHCPLeaseObtained         : 20070520221927.000000-420  
DHCPServer                : 192.168.1.1  
DNSDomain                 : hsd1.wa.comcast.net.  
DNSDomainSuffixSearchOrder :  
DNSEnabledForWINSResolution : False  
DNSHostName               : Lee-Desk  
DNSServerSearchOrder      : {68.87.69.146, 68.87.85.98}  
DomainDNSRegistrationEnabled : False  
FullDNSRegistrationEnabled : True  
IPAddress                 : {192.168.1.100}  
IPConnectionMetric         : 25  
IPEnabled                  : True  
IPFilterSecurityEnabled    : False  
WINSEnableLMHostsLookup    : True  
(...)
```

如果要检索特定的属性，可以像访问其他 PowerShell 对象上的属性那样访问其属性：

```
PS >$adapter.MacAddress  
00:12:17:77:B4:EB
```

有关在 PowerShell 中使用的 WMI 的详细信息，请参阅 15.1 节。

参考

- 15.1 节

第25章

管理 Exchange 2007 通信服务器

25.0 简介

Windows 管理一直以来都是由指向与单击操作来完成的。未来这些操作有可能被用于命令行，用于管理 Windows 的某些部分（或其他应用程序）。一旦所有管理都已被添加到用户界面，如果幸运的话，应用程序的程序员可以使用 COM API 或命令行工具来快速地完成某项任务。如果不那么幸运，它们仅公布一些快捷键设置，或可能什么也没提供。

这些大多数来自不同的管理工具模型：与一个提供充足完善的用户功能接口相比，少数应用程序的开发人员认为一个充足可编写的脚本功能管理是更为重要的，它就是 Exchange 2007。

与这些仅仿效用户接口的管理模型相比，Exchange 2007 开发团队首先编写管理 Exchange 的架构，Exchange 2007，不仅仅是一大特色更是有生命力的。Exchange 2007 包括了近 400 个让你管理 Exchange 系统的 Cmdlet，不仅仅数量大，而且具有很好的深度与广度。这些都保证了覆盖到 PowerShell 所有的 Cmdlet。这个 Exchange 管理控制用户建立在 PowerShell cmdlet 上。

Exchange 环境的命令基本上都是通过 Exchange 管理 cmdlet 管理的。

这种模型的好处是巨大的。与你是否要使用用户、组、邮箱或任何其他 Exchange 中的操作没有关系：你可以自动处理它。

25.1 尝试使用 Exchange 管理 shell 程序

问题

你希望通过 Exchange 管理 shell 程序的功能和特性，而不需要一个真实的环境。

解决方案

若要了解 Exchange 如何管理 shell 程序，可以使用基于 shell 程序的 Exchange 2007 Microsoft 虚拟实验室。

讨论

Microsoft 最近引入虚拟实验室作为一种核心技术来帮助你体验新技术。Exchange 2007 提供了几个这样的虚拟实验室。其中一个适用于 Exchange 管理 shell 程序，被称为“Exchange Server 2007：使用 Exchange Server 2007 管理控制台和 shell 程序的虚拟实验室”。

要启动此实验室，可以访问在 <http://www.microsoft.com/technet/traincert/virtuallab/default.mspx> 上的 Technet 的虚拟实验室主页。在按产品区分虚拟实验室的选项中，单击 Exchange Server；然后选择 Exchange Server 2007：使用 Exchange Server 2007 管理控制台和 shell 程序。

注册后，会启动虚拟实验室，它可让你来体验环境（请参阅图 25-1）。单击开始→所有程序→Microsoft Exchange Server 2007→Exchange 管理 shell 程序来启动 Exchange 管理 shell 程序。

25.2 自动执行向导任务

问题

你希望自动完成通常要在 Exchange 管理控制台中通过向导来完成的任务。

解决方案

要自动执行一项向导任务，在 Exchange 管理 shell 程序中完成该向导，然后保存显示的脚本以供将来参考。

讨论

由于 Exchange 管理控制台用户界面使用 PowerShell cmdlet 来完成所有操作，所以每个向导都显示了你可以运行的用来完成相同的任务的 PowerShell 脚本。

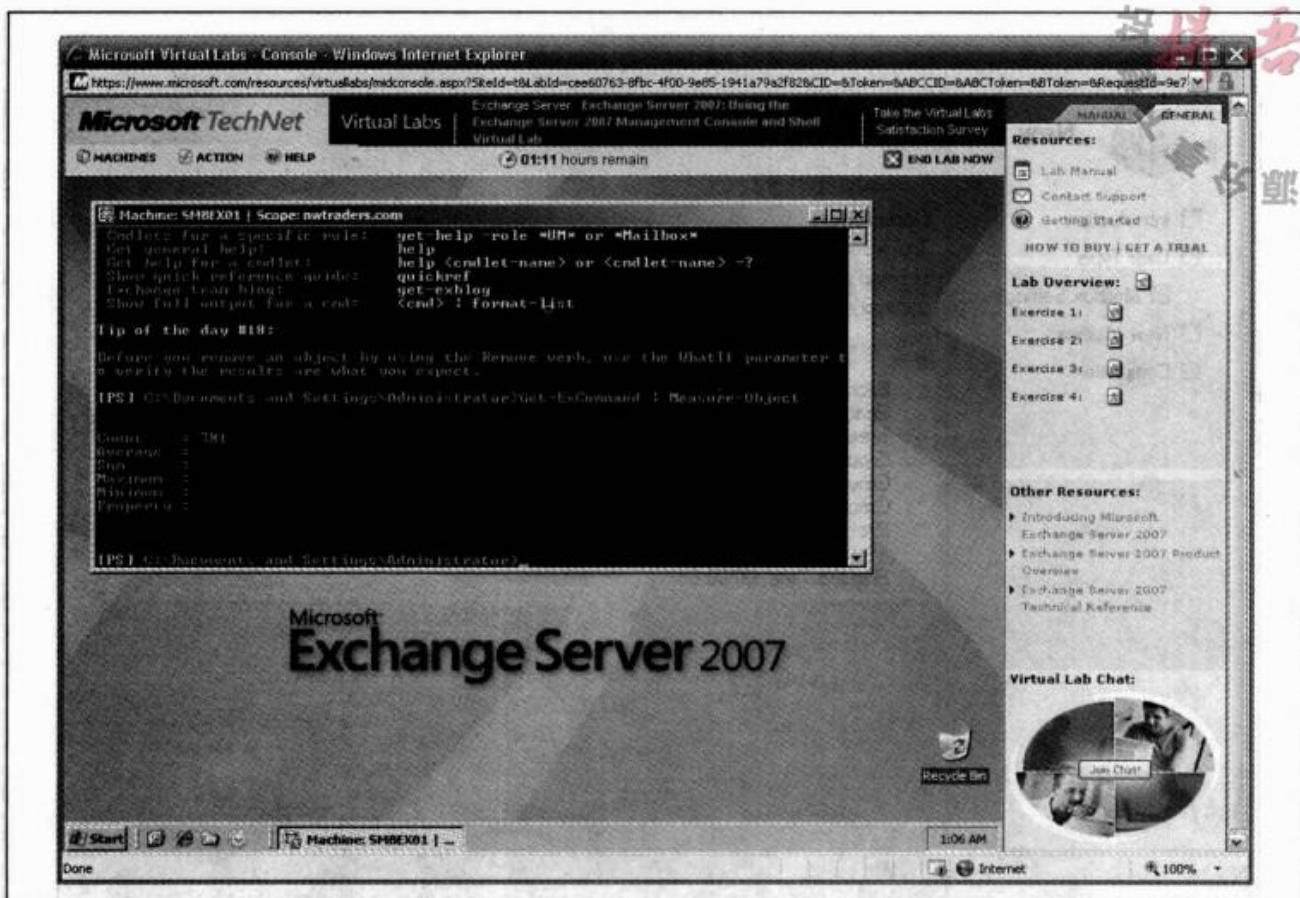


图 25-1：Exchange Server 2007 虚拟实验室

启动 Exchange 管理控制台，然后单击配置收件人→邮箱。在操作窗格中，单击新建邮箱。选择用户邮箱→存在用户→Preeda Ola。键入 **preeda** 作为别名，然后完成该向导。最后一步提供的 PowerShell 命令你可以在下一次使用，如图 25-2 中所示。

25.3 管理 Exchange 用户

问题

你希望从 Exchange 管理 shell 程序中获取并修改有关用户域账户的信息。

解决方案

若要获取和设置有关活动目录中用户的信息，可以分别使用 Get-User 和 Set-User cmdlet：

```
$user = Get-User *preeda*
$user | Format-List *
```

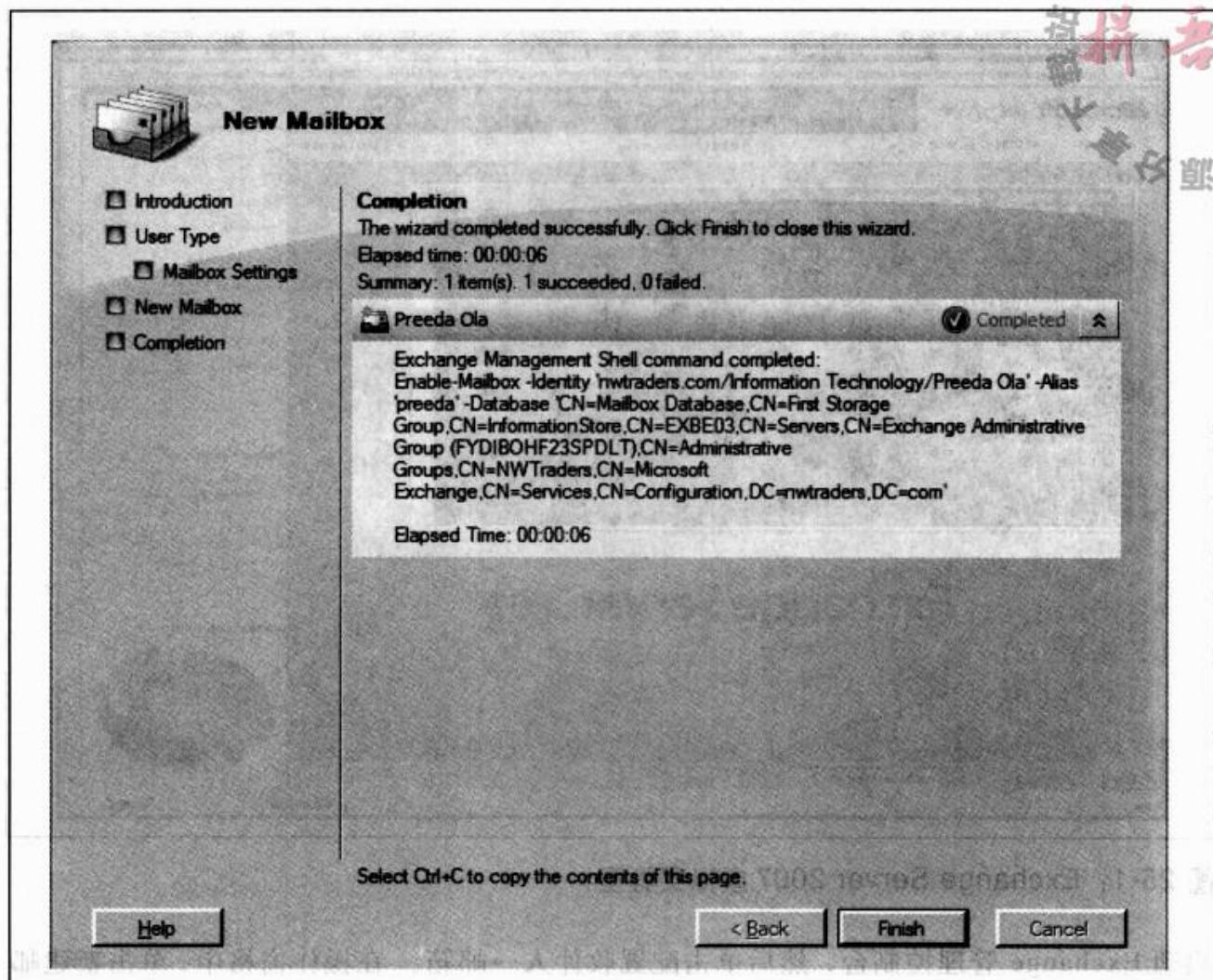


图 25-2：Exchange 2007 中的脚本向导

如果要把 Preeda 的标题更新为“Senior Vice President”，可以使用下面的命令：

```
$user | Set-User -Title "Senior Vice President"  
$user.Title
```

讨论

在使用 Exchange 2007 的时候，活动目录的账户是它的主要组成部分。你可能要经常修改域用户的邮箱，Get-User 和 Set-User 让你可以使用账户本身。

有关 Get-User cmdlet 的更多信息，请键入 **Get-Help get-user**。有关 Set-User cmdlet 的更多信息，请键入 **Get-Help Set-user**。

25.4 管理邮箱

问题

你要通过 Exchange 管理 shell 程序来获取并修改有关邮箱的信息。

解决方案

若要检索有关邮箱（或多个邮箱）的信息，可以使用 **Get-Mailbox cmdlet**：

```
$user = Get-Mailbox *preeda*
$user | Format-List *
```

若要修改一个邮箱信息，可以使用 **Set-mailbox cmdlet**。下面的命令完成功能是：当 Preeda 的邮箱大于 2GB 的时候，防止 Preeda 通过她的邮箱发送邮件，然后进行验证：

```
$user | Set-Mailbox -ProhibitSendQuota 2GB
$user | Get-Mailbox
```

讨论

除了检索和修改邮箱信息这种常见的任务，另一个有用的与邮箱的命令相关的是 **Move-Mailbox cmdlet**。例如，要将所有用户从一个存储组移动到另一个数据库：

```
Get-Mailbox |
    Where-Object { $_.Database -like "*SMBEX01\First Storage Group" } |
    Move-Mailbox -TargetDatabase "Mailbox Database 3"
```

过了一段时间之后，该命令会显示大容量邮箱的移动进度，如图 25-3 所示。

有关 **Get-Mailbox cmdlet** 的更多信息，请键入 **Get-Help Get-Mailbox**。有关 **Set-Mailbox cmdlet** 的更多信息，请键入 **Get-Help Set-Mailbox**。有关 **Move-Mailbox cmdlet** 的更多信息，请键入 **Get-Help Move-Mailbox**。

```

Machine: SMBEX01 | Scope: nwtraders.com
TargetNTAccountDNToCreate : 
TargetManager : 
FinanceArchive
  Moving messages.
  [too]
JimC
  % Preparing mailbox to be moved.
KimA
  Preparing mailbox to be moved.
  []

RsgMailboxGuid : 
RsgMailboxLegacyExchangeDN : 
RsgMailboxDisplayName : 
RsgDatabaseGuid : 
MoveType : IntraOrg
MoveStage : Completed
StartTime : 5/21/2007 2:13:55 AM
EndTime : 5/21/2007 2:14:28 AM
StatusCode : 0
StatusMessage : This mailbox has been moved to the target database.

```

图 25-3：操作中的 Move-Mailbox cmdlet

25.5 管理分发组

问题

如何在 Exchange 管理 shell 程序中获取并修改有关通讯组的信息。

解决方案

若要检索有关通讯组或通讯组的信息，可以使用 Get-DistributionGroup cmdlet 代码：

```
$group = Get-DistributionGroup "Stock Traders"
$group | Format-List *
```

若要修改有关组的信息，可以使用 Set-DistributionGroup cmdlet。下面的示例更新了 Stock Traders 通讯组使之只接受来自 Stock Traders 通讯组的其他成员的邮件：

```
$group | Set-DistributionGroup -AcceptMessagesOnlyFromDLMembers "Stock Traders"
```

若要将用户添加到通讯组，可以使用 Add-DistributionGroupMember cmdlet：

```
$group | Add-DistributionGroupMember -Member *preeda*
```

若要列出通讯组中的成员，可以使用 Get-DistributionGroupMember cmdlet：

```
$group | Get-DistributionGroupMember
```

讨论

有关 **Get-DistributionGroup** cmdlet 的更多信息，请键入 **Get-Help Get-DistributionGroup**。有关 **Set-DistributionGroup** cmdlet 的更多信息，请键入 **Get-Help Set-DistributionGroup**。有关 **Add-DistributionGroupMember** cmdlet 的更多信息，键入 **Get-Help Add-DistributionGroupMember**。有关 **Get-DistributionGroupMember** cmdlet 的更多信息，请键入 **Get-Help Get-DistributionGroupMember**。

25.6 管理传输规则

问题

如何管理应用于传入或传出邮件的传输规则。

解决方案

若要在服务器上创建传输规则，可以使用 **New-TransportRule** cmdlet，如例 25-1 所示。

例 25-1：创建新的传输规则

```
$from = Get-TransportRulePredicate FromScope
$from.Scope = "NotInOrganization"

$attachmentSize = Get-TransportRulePredicate AttachmentSizeOver
$attachmentSize.Size = 0

$action = Get-TransportRuleAction ApplyDisclaimer
$action.Text = "Warning: Only open attachments you were already expecting."

New-TransportRule -Name "Attachment Warning"
-Conditions $from,$attachmentSize -Action $action -Enabled:$true
```

讨论

New-TransportRule cmdlet 在服务器上添加传输规则。传输规则包括谓词、操作条件和应用于该规则的操作的集合。

在该解决方案示例中，我们添加的规则警告内部用户来自外部世界的附件是危险的。有关在 **New-TransportRule** cmdlet 的更多信息，请键入 **Get-Help New-TransportRule**。

25.7 管理 Outlook Web Access

问题

如何在 Exchange 管理 shell 程序中获取并修改 Outlook Web Access 设置。

解决方案

若要检索有关Outlook Web Access虚拟目录的信息，可以使用Get-OwaVirtualDirectory cmdlet：

```
$owa = Get-OwaVirtualDirectory "owa (Default Web Site)"
```

若要修改该虚拟目录的信息，可以使用 Set-OwaVirtualDirectory cmdlet。本示例防止用户通过 Outlook Web Access 更改密码：

```
$owa | Set-OwaVirtualDirectory -ChangePasswordEnabled:$false
```

讨论

有关 `Get-OwaVirtualDirectory` cmdlet 的更多信息，请键入 **`Get-Help Get-OwaVirtualDirectory`**。有关 `Set-OwaVirtualDirectory` cmdlet 的详细信息，请键入 **`Get-Help Set-OwaVirtualDirectory`**。

管理操作管理器 2007 服务器

26.0 简介

与 Exchange 2007 相似, System Center 操作管理器 2007(以前称为 Microsoft Operations Manager) 广泛地采用 PowerShell 作为一种自动化管理的技术。Operations Manager 2007 采用一个双路的方法。它包括 70 多个 PowerShell cmdlet 以及 PowerShell 提供程序, 以便你把命令分配(和导航)到管理组中。

操作管理器 2007 中最常用的管理任务可以划分为以下多个类别之一: 管理代理、管理包、规则、任务、通知和维护窗口。

26.1 体验命令 shell 程序

问题

你希望使用操作管理器命令 shell 的特性和功能, 而不需要一个真实的环境。

解决方案

若要浏览操作管理器命令 shell 程序, 可以使用 Microsoft Forefront 和系统中心演示工具包。

讨论

可以下载的虚拟机映像和联机虚拟实验室最近已经成为尝试使用新技术的非常有效的方法。Microsoft System Center 应用程序套件提供了几个虚拟机镜像的下载, 其中一个安

装了 Forefront Security，可以与 System Center 系列的其余部分进行交互。在虚拟实验室中的 Oxford 计算机代表正在运行的 System Center 2007 操作管理器，它为新技术的研究提供了一个优秀的途径。

要下载此演示工具包，请访问 Microsoft 的下载页面 <http://download.microsoft.com>，然后搜索“System Center demonstration toolkit”。

注册后，要下载该演示工具包的所有文件并启动安装程序。一旦安装完成，就可以访问安装目录中 *VMImages* 目录，并启动 Oxford 演示虚拟机。

一旦虚拟计算机加载完毕，就单击“开始→所有程序→系统中心操作管理器 2007→命令 shell 程序”来启动操作管理器命令 shell，如图 26-1 中所示。

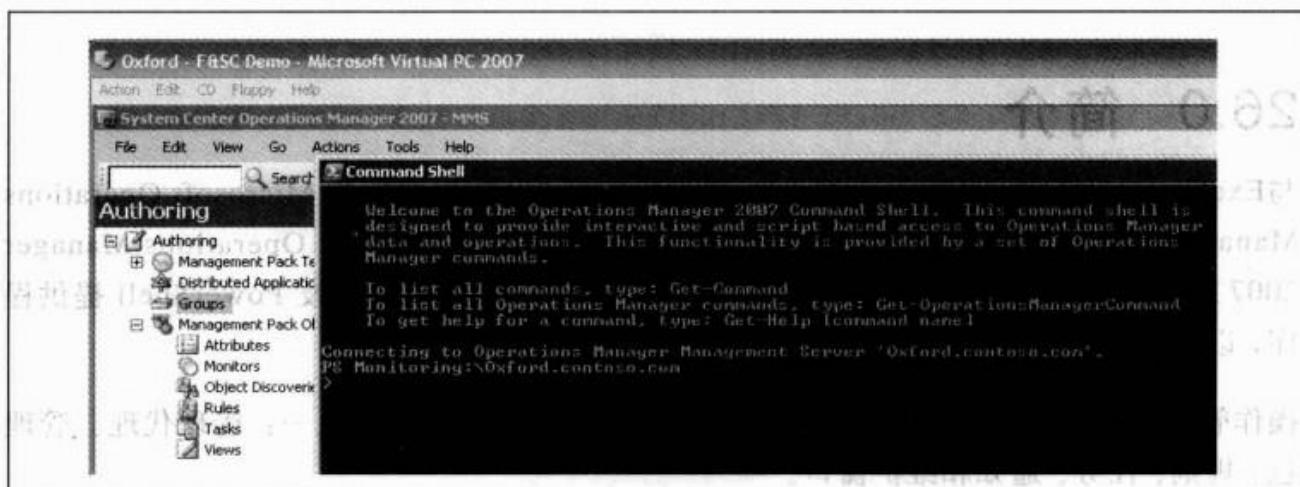


图 26-1：系统中心操作管理器命令 shell 程序

26.2 管理操作管理器代理

问题

你想管理远程计算机上的操作管理代理。

解决方案

若要检索关于已安装的代理的信息，可以使用 *Get-Agent* cmdlet 代码：

```
PS Monitoring:\Oxford.contoso.com  
>Get-Agent | Select-Object DisplayName
```

若要从远程计算机上卸载操作管理代理，可以使用 *Remove-ManagementAgent* cmdlet 代码：

```
DisplayName  
-----  
Ibiza.contoso.com  
Denver.contoso.com  
Sydney.contoso.com
```

若要移除一个代理程序，使用 **Uninstall-Agent cmdlet**：

```
PS Monitoring:\Oxford.contoso.com  
>Get-Agent | Where-Object { $_.DisplayName -match "Denver" } |  
>> Uninstall-Agent
```

若要在特定的计算机上安装代理程序，使用 **Install-AgentByName 函数**：

```
PS Monitoring:\Oxford.contoso.com  
>Install-AgentByName Oxford.contoso.com
```

讨论

Get-Agent cmdlet 返回了大量的有关检索每个代理的信息。解决方案中的示例对此处进行筛选只为了显示 **DisplayName**，但你可以忽略 **Select-Object cmdlet** 来检索有关该代理的所有信息。

如果你需要更好地控制代理安装的过程，请检查 **Install-AgentByName 函数** 的内容：

```
PS Monitoring:\Oxford.contoso.com  
>Get-Content Function:\Install-AgentByName
```

该函数简化了用于安装代理程序最常见的情形。**Install-Agent cmdlet** 支持它提供的附加功能。

有关 **Get-Agent cmdlet** 的更多信息，请键入 **Get-help Get-Agent**。有关 **Install-Agent cmdlet** 的更多信息，请键入 **Install-Agent**。有关 **Uninstall-Agent cmdlet** 的更多信息，请键入 **Get-help Uninstall-Agent**。

26.3 计划维护窗口

问题

你想将服务器配置在维护模式下，以防止它生成不正确的警报。

解决方案

若要在计算机上安排一个维护窗口，可以使用 **new-MaintenanceWindow cmdlet**：

```
$computer = Get-Agent | Where-Object { $_.Name -match "Denver" }
$computer.HostComputer | New-MaintenanceWindow
    -StartTime (Get-Date)
    -EndTime   (Get-Date).AddMinutes(5)
    -Comment "Security updates"
```

若要检索有关该维护窗口的信息，可以使用 **Get-MaintenanceWindow cmdlet**：

```
>$computer.HostComputer | Get-MaintenanceWindow

MonitoringObjectId : a542ffe8-91a2-84a6-37b0-0555b15513bd
StartTime          : 5/22/2007 9:27:23 AM
ScheduledEndTime   : 5/22/2007 9:32:23 AM
EndTime            :
Reason             : PlannedOther
Comments           : Security updates
User               : CONTOSO\Administrator
LastModified       : 5/22/2007 9:27:23 AM
ManagementGroup    : MMS
ManagementGroupId : 846c0974-7fd0-58f2-8020-400f125beb67
```

要停止维护模式，可以使用 **Set-MaintenanceWindow cmdlet** 立即结束维护窗口：

```
$computer.HostComputer | Set-MaintenanceWindow -EndTime (Get-Date)
```

讨论

有关 **New-MaintenanceWindow cmdlet** 的更多信息，请键入 **Get-help New-MaintenanceWindow**。有关 **Get-MaintenanceWindow cmdlet** 的更多信息，请键入 **Get-help Get-MaintenanceWindow**。有关 **Set-MaintenanceWindow cmdlet** 的更多信息，请键入 **Get-help Set MaintenanceWindow**。

26.4 获取、安装和卸载管理包

问题

如何自动执行部署管理包的配置。

解决方案

若要检索有关已安装的管理包的信息，可以使用 **Get-ManagementPack cmdlet**，如例 26-1 所示：

例 26-1：使用 **Get-ManagementPack cmdlet**

```
PS Monitoring:\Oxford.contoso.com
>$mp = Get-ManagementPack | Where-Object { $_.DisplayName -eq "Health Internal Library" }
```

```
PS Monitoring:\Oxford.contoso.com  
>$mp
```

```
Name          : System.Health.Internal  
TimeCreated   : 5/22/2007 9:38:40 AM  
LastModified  : 5/22/2007 9:38:40 AM  
KeyToken      : 31bf3856ad364e35  
Version       : 6.0.5000.0  
Id            : 9395a1eb-6322-1c8b-71ad-7ab6955c7e11  
VersionId     : dfaeece9-437e-7d46-edce-260bd77a8667  
References    : {System.Library, System.Health.Library}  
Sealed        : True  
ContentReadable : False  
FriendlyName  : System Health Internal Library  
DisplayName   : Health Internal Library  
Description   : System Health Internal Library: This Management Pack (...)  
DefaultLanguageCode : ENU  
LockObject    : System.Object
```

可以使用 **Uninstall-ManagementPack cmdlet** 来删除一个管理包：

```
$mp = Get-ManagementPack | Where-Object { $_.DisplayName -eq "Management Pack Name" }  
$mp | Uninstall-ManagementPack
```

若要安装一个管理包，可以把它的路径提供给 **Install-ManagementPack cmdlet**：

```
Install-ManagementPack <PathToManagementPack>
```

讨论

有关 **Get-ManagementPack cmdlet** 的更多信息，请键入 **Get-help Get-ManagementPack**。有关 **Install-ManagementPack cmdlet**，请键入 **Get-help Install-ManagementPack**。有关 **Uninstall-ManagementPack cmdlet** 的更多信息，请键入 **Get-help Uninstall-ManagementPack**。

26.5 启用或禁用规则

问题

如何启用或禁用一个管理包中的规则。

解决方案

若要检索一个规则，可以使用 **Get-Rule**：

```
$rule = Get-Rule | Where-Object { $_.DisplayName -match "RuleName" }
```

然后，使用 `Enable-Rule` 或者 `Disable-Rule` cmdlet 来启用或禁用那条规则，如下：

```
$rule | Enable-Rule  
$rule | Disable-Rule
```

讨论

有关 `Get-Rule` cmdlet 的更多信息，请键入 `Get-help Get-Rule`。有关 `Disable-Rule` cmdlet 的更多信息，请键入 `Get-help Disable-Rule`。有关 `Enable-Rule` cmdlet 的更多信息，请键入 `Get-help Enable-Rule`。

26.6 列出并启动任务

问题

你想列出一个监视对象中的所有允许的任务，然后调用其中的一个。

解决方案

若要检索当前监视对象允许使用的 cmdlet，可以使用 `Get-Task` cmdlet，如下：

```
$task = Get-Task | Where-Object { $_.DisplayName -match "TaskName" }
```

然后，使用 `Start-Task` cmdlet 来启动任务：

```
$task | Start-Task
```

讨论

`Get-Task` cmdlet 检索适用于当前路径监视对象的任务。若要更改返回的 cmdlet 任务列表，可以定位到你要管理的监视对象目录。

有关 `Get-Task` cmdlet 的更多信息，请键入 `Get-help Get-Task`。有关 `Start-Task` cmdlet 的更多信息，请键入 `Get-help Start-Task`。

26.7 管理警报

问题

如何检索和管理当前监视对象中的警报。

解决方案

若要检索当前监视对象上的警报，使用 **Get-Alert cmdlet**。若要只检索活动的警报，可以应用一个只包括那些 **ResolutionState** 为 0 的警报的筛选器。

```
>Get-Alert | Where-Object { $_.ResolutionState -eq 0 } | Select-Object Description  
  
Description  
-----  
The process started at 2:15:56 AM failed to create System.Discovery.Data, no error  
The process started at 2:05:23 PM failed to create System.Discovery.Data. Errors  
MSEExchangeIS service is stopped. This may be caused by missing patch KB 915786.  
The computer Ibiza.contoso.com was not pingable.  
The computer Sydney.contoso.com was not pingable.
```

若要解除一个警报，要把它管道输出给 **Resolve-Alert cmdlet**。例如，要批量的清除警报，可以使用下面的命令：

```
Get-Alert | Where-Object { $_.ResolutionState -eq 0 } | Resolve-Alert
```

讨论

有关 **Get-Alert cmdlet** 的更多信息，请键入 **Get-help Get-Alert**。有关 **Resolve-Alert cmdlet** 的更多信息，请键入 **Get-Help Resolve-Alert**。

案式夫鋪

，器类前的“鼎彝”或“尊彝”等字，即指此。

For more information, please contact www.fcc.gov/encyclopedia.

WTF is on (SW). You would have to be blind or deaf to not see what's going on. It's like you're trying to tell us that we're not important. I mean, I'm not saying that you're not important, but I think it's important to remember that we're not the only ones who care about this issue.

刺青頭量掛獎，成國。Resolving-Art competition，掛營个一綱繩要當；全命頭面不用繩以挂，掛繩

For more information about the study, please contact Dr. Michael J. Hwang at (319) 335-1151 or via email at mhwang@uiowa.edu.

卷十七

有关 `get-After` 和 `get-Left` 的更多详细信息，参见 [GET-LEFT 和 GET-AFTER](#)。

第五部分

参考资源

- 附录 A, PowerShell 语言和环境
- 附录 B, 正则表达式参考
- 附录 C, PowerShell 自动变量
- 附录 D, 标准 PowerShell 动词
- 附录 E, 选定的 .NET 类和它们的使用
- 附录 F, WMI 参考
- 附录 G, 选定的 COM 对象和它们的使用
- 附录 H, .NET 字符串格式
- 附录 I, .NET 日期格式



代驾正集

真資參

皇根味吉告 HerPisach / 東南
卷零為法貴陣有 / 日本國
量變極自 HerSrawap / 泰國
同極 HerSrawap / 泰國
出對酒 / 古味美 TEK 韓家 / 日本國
卷零 BMW / 德國
甲對酒 / 古味美 MOG 韓家 / 日本國
左對串 / 宝工 / 日本國
右對酒 / 宝工 / 日本國

PowerShell 语言和环境

命令和表达式

PowerShell 将你输入的内容拆分成独立的单元，然后把每个单元翻译成一个命令或一个表达式。两者的区别是微小的，表达式支持逻辑和流程控制语句（如 if、foreach 和 throw），而命令不支持。

你可能想控制 Windows PowerShell 翻译你的语句的方式，所以表 A-1 列出一些可选的方式。

表 A-1：Windows PowerShell 赋值控制语句

声明	示例	解释
优先控制：()	PS >5 * (1 + 2) 15 PS >(dir).Count 2276	强制一个命令或表达式的赋值，使用圆括号控制赋值的顺序
表达式内解析：\$()	PS >"The answer is \$(2+2)" The answer is (2+2) PS >"The answer is \$((2+2))" The answer is 4 PS >\$value = 10 PS >\$result = \$(>> if(\$value -gt 0) { \$true } > else { \$false } >>) >> 义含 PS >\$result True	内解析表达式就像是一小段程序代码，只有在表达式中包含逻辑和流程控制语句时候才使用。 也可以在字符串中进行动态的复制

表 A-1：Windows PowerShell 赋值控制语句（续）

声明	示例	解释
列表赋值：@()	<pre>PS > "Hello".Length 5 PS > @("Hello").Length 1 PS > (Get-ChildItem).Count 12 PS > (Get-ChildItem *.txt).Count PS > @((Get-ChildItem *.txt).Count 1</pre>	强制一个表达式当作列表来处理。如果表达式已经是一个列表，则保持不变；如果表达式不是一个列表，PowerShell会临时将它作为列表来处理

注释

为了创建单行的注释，使用 # 作为行的开头，PowerShell 不支持多行的注释，但是你可以通过将脚本放到 *here* 字符串中来注释大段的脚本。

```
# This is a regular comment

# Start of the here string
$null = @"
function MyTest
{
    "This should not be considered a function"
}

$myVariable = 10;
@"
# End of the here string
# This is regular script again
```

注意：可以阅读后边的字符串章节的内容，来了解更 *here* 字符串的信息

变量

PowerShell 提供了几种方式来定义和访问变量，如表 A-2 中总结

表 A-2：Windows PowerShell 变量语法

语法	含义
\$simpleVariable = "Value"	一个简单的变量名。变量名必须由数字与字符组成，不区分大小写

表 A-2: Windows PowerShell 变量语法 (续)

语法	含义
<code>\$[arbitrary#@#@`{var`}iable] = "Value"</code>	一个任意的变量名。变量名必须用大括号括起来, 可以包含任意的字符, 大括号内的括号必须用`来转义
<code>\$(c:\filename.extension)</code>	变量 Get/Set 内容语法。这与任意变量名的语法相似, 如果变量中的路径是有效的, 那么你可以通过变量读取或设置文件的内容
<code>[datatype] \$variable = "Value"</code>	强类型变量。保证变量内容与变量的类型一致, 如果给变量赋予了错误的数值, PowerShell 会抛出异常
<code>\$SCOPE:variable</code>	在指定的范围内获得或设置变量。有效的范围的名称包括: Global: 变量对整个 Shell 可用 Script: 变量在当前脚本中可用 Local: 变量在当前范围与子范围可用 Private: 变量只对当前范围可用 当在交互的 Shell 中定义时, 变量的默认范围是 Global; 当在函数或脚本块外定义变量, 默认的范围是 Script; 其他处默认的是 Local
<code>New-Item Variable:\variable -Value value</code>	使用变量提供者创建变量
<code>Get-Item Variable:\variable</code>	通过变量提供者获得变量或通过
<code>Get-Variable variable</code>	Get-Variable cmdlet 获得。可以让你访问变量的额外的信息, 如它的选项和描述
<code>New-Variable variable -Option option -Value value</code>	使用 New-Variable cmdlet 创建变量。可以让你为变量提供额外的信息, 如它的选项和描述

注意: 与其他语言不同, 当转换数字为 int 型时候, PowerShell 会进行四舍五入。

```
PS >(3/2)
1.5
PS >[int] (3/2)
2
```

第 6 章讲述了 PowerShell 截断数字的内容。

布尔值

(卷) 变量与表达式 PowerShell 教程

布尔值变量是比较常用的,正如它的名称所示,它经常会初始化值为 \$true 和 \$false。当作为一个布尔表达式的一部分来赋值的时候(比如 if 表达式中),PowerShell 会把它们映射成一个合适的表示法,如表 A-3 所示。

表 A-3: Windows PowerShell 布尔表达式解释

结果	布尔值
\$true	True
\$false	False
\$null	False
Nonzero number	True
Zero	False
Nonempty string	True
Empty string	False
Nonempty array	True
Empty array	False
Hashtable (无论 empty 与否)	True

字符串

PowerShell 提供了几种简单的方式来处理简单文本数据。

原本和扩展的字符串

为了定义一个完全遵循原样的字符串(没有变量或有转义表达式),可以使用单引号把它括起来,如下:

```
$myString = 'hello `t $ENV:SystemRoot'
```

变量 \$myString 实际的值是 hello `t \$ENV:SystemRoot。

为了定义一个扩展的字符串(包括变量和转义表达式),可以使用双引号把它括起来,如下:

```
$myString = "hello `t $ENV:SystemRoot"
```

变量 \$mystring 实际的值是 hello C:\WINDOWS。

为了在一个用单引号引用的字符串中包括一个单引号引用，或者在一个双引号引用的字符串中包括一个双引号引用，可以在一行中包含两个引用符号，如下：

```
PS >"Hello ""There""!"  
Hello "There"!  
PS >'Hello ''There'''!  
Hello 'There'!
```

注意：为了在一个扩展的字符串中包含一个复杂的表达式，使用子表达式，比如：

```
$prompt = "$(get-location) >"
```

访问一个对象的属性需要一个子表达式，如下：

```
$output = "Current script name is: $($MyInvocation.MyCommand.Path)"
```

Here 字符串

为了定义一个 *here* 字符串（跨多行），在字符串开始加入 @” 符号，在字符串的结尾加入 “@ 符号。

如下所示：

```
$myHereString = @"
This text may span multiple lines, and may
contain "quotes".
"@
```

Here 字符串中可能包含单一的字符或扩展的字符。

转义顺序

PowerShell 支持在字符串中加入转义符号，如表 A-4 所示：

表 A-4: Windows PowerShell 转义序列

序列	含义
`0	空字符。经常用作记录的分隔符
`a	在控制台下显示的时候生成嘟嘟声

表 A-4: Windows PowerShell 转义序列 (续)

序列	含义
`b	回格字符。当在控制台显示时候，前一个字符被覆盖
`f	在多数打印机上打印的时候产生分页
`n	产生新行
`r	回车
`t	Tab
`v	垂直 Tab
'` (Two single quotes)	单引号
" " (Two double quotes)	双引号
\<any other character>	字符原样

数字

PowerShell 提供了一些选项来与数字和数值交互。

简单任务

为了定义一个变量来保存数字的数据，可以像定义其他变量一样。PowerShell 自动采用相应的格式来保存数据。

```
$myInt = 10
```

变量 \$myint 的值是 10，类型是 32 位整型。

```
$myDouble = 3.14
```

变量 \$myDouble 的值是 3.14，类型是 32 位浮点。

为了明确的声明一个数字为 64 位长整型或小数，使用 Long 或 decimal 下标，如下：

```
$myLong = 2147483648L
```

变量 \$mylong 的值是 2147483648，类型长整型。

```
$myDecimal = 0.999D
```

变量 \$myDecimal 的值是 0.999。

PowerShell 还支持科学符号，如下：

\$myPi = 3141592653e-9

变量 \$myPi 的值是 3.141592653。

PowerShell 中的数据类型（整型、长整型、浮点、小数）建立在 .NET 的数据类型上。

管理用的数字常量

管理员很少有机会与数字打交道，PowerShell 提供了数字的常量，包括 gb、mb 和 kb，分别用来表示千兆、兆和千字节。

```
PS >$downloadTime = (1gb + 250mb) / 120kb  
PS >$downloadTime  
10871.4666666667
```

十六进制和其他进制

为了直接输入一个十六进制数字，可以使用前缀 0X，如下：

\$mvErrorCode = 0xFE4A

变量 \$myErrorCode 的值是 65098。

PowerShell 脚本语言本身不支持其他进制的数字，但是通过与 .NET 框架的交互，就可以支持转换成二进制、八进制、十进制和十六进制。

```
$myBinary = [Convert]::ToInt32("101101010101", 2)
```

这是二进制转换，变量 \$myBinary 的值是整型 2901

```
$myOctal = [Convert]::ToInt32("1334567", 8)
```

这是八进制转换，\$myOctal 变量的值是 342391

```
$myHexString = [Convert]::ToString(65098, 16)
```

这是十六进制转换，\$myHexString 变量的值是 fe4a

```
$myBinaryString = [Convert]::ToString(12345, 2)
```

\$myBinaryString 变量的值是 11000000111001

注意：参见李章稍后的使用 .NET 框架部分，来了解 PowerShell 与 .NET 框架的交互。

数组和列表

数组的定义

PowerShell 使用 @() 语法来定义一个数组，为了创建一个空的数组，键入：

```
$myArray = @()
```

为了定义一个非空的数组，可以使用逗号将数组中的元素分开，如下：

```
$mySimpleArray = 1, "Two", 3.14
```

数组中可以只包含一个元素，如下：

```
$myList = , "Hello"
```

或者：

```
$myList = @("Hello")
```

除非你定义强类型的数组，否则数组中的元素可以存放任意类型的数据。在下面这个例子中，通过 [] 定义强类型的变量，int[] 表示这是一个整型的数组，如下：

```
[int[]] $myArray = 1,2,3.14
```

在这种情况下，如果不能把数组中的元素转换成定义的数据类型，PowerShell 会抛出错误，本例中，把 3.14 进行四舍五入。

```
PS >$myArray[2]  
3
```

注意：为了保证 PowerShell 将不确定长度的集合视为列表，要使用列表赋值语法 @()。

数组可以是多维的交错型数组——数组中包含数组

```
$multiDimensional = @(  
    (1,2,3,4),  
    (5,6,7,8)  
)
```

\$multiDimensional[0][1] 返回 2，来自 0 行，1 列。

\$multiDimensional[1][3] 返回 8，来自 1 行，3 列。

为了定义一个多维的非交错型的数组，要使用下面的语法：

```
$multidimensional = New-Object "Int32[,]" 2,4  
$multidimensional[0,1] = 2  
$multidimensional[1,3] = 8
```

访问数组

为了访问数组中的某一元素，要使用`[]`操作符。PowerShell 对数组中的元素编码从 0 开始，在下面的数组`$myArray = 1, 2, 3, 4, 5, 6`中，变量`$myArray[0]`返回数组中的第一个元素，值是 1。

`$myArray[2]` 返回数组中的第三个元素，值是 3；`$myArray[-1]` 返回数组中最后一个元素，值是 6；`$myArray[-2]` 返回数组中的倒数第 2 个元素，值是 5。

你也可以访问数组中的指定范围内的数据，如下：

```
PS >$myArray[0..2]  
1  
2  
3
```

返回数组中从 0 到 2 的元素：

```
PS >$myArray[-1..2]  
6  
1  
2  
3
```

返回最后一个元素，然后从头开始，返回 0 到 2 的元素。PowerShell 会自动循环，因为范围内一个是正数，第二个是负数。

```
PS >$myArray[-1..-3]  
6  
5  
4
```

返回数组中从第三到结束的最后的元素，PowerShell 没有自动循环，因为范围内两个数符号相同，都是负号。

数组限制

你可以把上面几种情况组合起来，形成更复杂的表达式。可以使用加号来将数组从指定的索引处分开。

```
$myArray[0,2,4]
```

返回值 0,2,4。

```
$myArray[0,2+4..5]
```

返回值是 0,2,4,5。

```
$myArray[ , 0+2 .. 3+0, 0]
```

返回值是 0,2,3,0,0。

注意：你可以使用数组切片语法来创建数组，如：

```
$myArray = ,0+2..3+0,0
```

哈希表（联合数组）

哈希表的定义

PowerShell 中的哈希表可以允许你定义成对的键/值，为了定义一个哈希表，使用下面的语法：

```
$myHashtable = @{}  
# Create a new HashTable object
```

你可以通过键 / 值方式方式创建哈希表。PowerShell 设定键是字符型的，值可以是任意类型的数据，如下：

```
$myHashtable = @{ Key1 = "Value1"; "Key 2" = 1,2,3; 3.14 = "Pi" }
```

访问哈希表

为了访问哈希表中的元素，你既可以使用数组项访问或数组属性访问的语法，如下：

```
$myHashtable["Key1"]
```

返回值 "value1"。

```
$myHashtable."Key 2"
```

返回值是数组 1, 2, 3。

```
$myHashtable["New Item"] = 5
```

向哈希表中添加 "new_item"，值是 5

Searchable "New Item" - E

也是向哈希表中添加 "new_item"，值是 5

XML

PowerShell 支持将 XML 作为内置的数据类型。为了创建一个 XML 变量，将字符串赋给 [xml] 类型，如下：

```
$myXml = [xml] @"
<AddressBook>
    <Person contactType="Personal">
        <Name>Lee</Name>
        <Phone type="home">555-1212</Phone>
        <Phone type="work">555-1213</Phone>
    </Person>
    <Person contactType="Business">
        <Name>Ariel</Name>
        <Phone>555-1234</Phone>
    </Person>
</AddressBook>
"@
```

PowerShell 暴露了所有子节点和属性作为属性，当它这么做时，PowerShell 自动将子节点分组共享同一节点类型，如下：

```
$myXml.AddressBook
```

返回一个包含 Person 属性的对象：

```
$myXml.AddressBook.Person
```

返回 Person 节点的列表，每个 Person 节点包括 contactType、Name 和 Phone 属性：

```
$myXml.AddressBook.Person[0]
```

返回第一个 Person 节点：

```
$myXml.AddressBook.Person[0].ContactType
```

返回第一个 Person 节点的 ContactType 是 Personal。

注意： XML 数据类型封装了 .NET 框架下的 XmlDocument 和 XmlElement 类。与其他多数 PowerShell 下的 .NET 封装不同，XML 封装不会从子类中暴露属性，因为这样可能会与 PowerShell 添加的节点名称的动态属性冲突。

为了访问子类的属性，使用 PsBase 属性，如下所示：

```
$myXml.PsBase.InnerXml
```

参见本章下面的“使用 .NET 框架”来了解 PowerShell 如何和 .NET 框架交互。



简单操作符号

当你定义好了数据后，下一步开始使用这些数据。

算术运算符

算术运算符可以允许你对数据进行数学操作，如表 A-5 所示。

注意：除了PowerShell内置的操作符之外，还可以使用.NET框架中的System.Math类提供的丰富的功能，如：

```
PS >[Math]::Pow([Math]::E, [Math]::Pi)  
23.1406926327793
```

参见本章下面的“使用 .NET 框架”来了解 PowerShell 如何和 .NET 框架交互。

表 A-5: PowerShell 数学操作符

操作符	含义
+	<p>加法操作符</p> <p>语法: <code>\$leftValue + \$rightValue</code></p> <p>当变量都是数字时, 返回它们的和</p> <p>当变量都是字符串时, 返回由两个字符串组成的新的字符串</p> <p>当变量都是数组时, 返回由两个数组组成的新的数组</p> <p>当变量都是哈希表时, 返回由两个哈希表合并的新的哈希表, 因为哈希表中的键必须唯一, 所以当两个哈希表中如果有重复的键, PowerShell 会抛出异常</p> <p>当使用其他类型的数据时, 如果那种数据类型支持加操作, PowerShell 会根据他定义的方法进行加操作。</p>
-	<p>减法操作</p> <p>语法: <code>\$leftValue - \$rightValue</code></p> <p>当变量都是数字型时, 会返回它们的差</p> <p>本操作符不支持字符串、数组、哈希表</p> <p>当使用其他类型的数据时, 如果那种数据类型支持减操作, PowerShell 会根据他定义的方法进行减操作</p>
*	<p>乘法操作符</p> <p>语法: <code>\$leftValue * \$rightValue</code></p> <p>当变量类型是数字型时, 返回它们的积</p>

表 A-5: PowerShell 数学操作符 (续)

操作符	含义
<code>*</code>	当变量类型是字符串时, 如 (<code>"=" * 80</code>), 会返回一个新的字符串, 包括指定数字次数的字符串
<code>..*</code>	当变量类型是数组时, 如 (<code>1..3 * 7</code>), 会返回一个新的数组, 包括指定数字次数的数组
<code>-notcontains</code>	不支持哈希表
<code>*</code>	当使用其他类型的数据时, 如果那种数据类型支持乘法操作, PowerShell 会根据定义的方法进行乘操作
<code>/</code>	除法操作 语法: <code>\$leftValue / \$rightValue</code> 当变量都是数字型时, 会返回它们的商 本操作符不支持字符串、数组、哈希表 当使用其他类型的数据时, 如果那种数据类型支持除操作, PowerShell 会根据定义的方法进行除操作
<code>%</code>	求余操作 语法: <code>\$leftValue % \$rightValue</code> 当变量都是数字型时, 会返回它们的商 本操作符不支持字符串、数组、哈希表 当使用其他类型的数据时, 如果那种数据类型支持除操作, PowerShell 会根据定义的方法进行除操作
<code>+=</code>	赋值操作符:
<code>-=</code>	<code>\$variable operator =value</code>
<code>*=</code>	这些操作符与简单的算术操作符相似, 不同之处是将结果保存到操作符左侧的变量中, 如下:
<code>/=</code>	<code>\$variable = \$variable operator value</code>
<code>%=</code>	

逻辑操作符

逻辑操作符可以让你比较布尔类型的值, 如表 A-6 所示。

表 A-6: Windows PowerShell 逻辑操作符

操作符	含义
-and	逻辑与： 语法：\$leftValue -and \$rightValue 如果左、右两侧的条件都为真，那么就返回真；否则返回假。 你可以在一个表示式中使用多个 -and 操作符： \$value1 -and \$value2 -and \$value3 ... PowerShell 将每个 -and 操作符当作一个小的循环操作符，只有当它前面的条件为真时，才会进行操作
-or	逻辑或： 语法：\$leftValue -or \$rightValue 如果左、右两侧的条件之一为真，那么就返回真；否则返回假。 你可以在一个表示式中使用多个 -or 操作符： \$value1 -or \$value2 -or \$value3 ... PowerShell 将每个 -or 操作符当作一个小的循环操作符，只有它前面的条件为假时，才会进行操作
-xor	逻辑异或： 语法：\$leftValue -xor \$rightValue 如果左、右侧的参数有一个为真，那么返回真；否则都返回假
-not	逻辑非： ! -not \$value 如果右侧参数为假，则返回真；否则返回假

二进制操作符

表 A-7 中列出的二进制操作符，可以允许你对操作的语句按位进行逻辑操作。按位比较的时候，1 代表真，0 代表假。

表 A-7: Windows PowerShell 二进制操作符

操作符	含义
-band	二进制与： 语法：\$leftValue -band \$rightValue 如果左右两侧的相同位置的位都是 1，则该位被设为 1，否则设为 0。比如：

表 A-7: Windows PowerShell 二进制操作符 (续)

操作符	含义
-band	二进制与操作符。示例中 & & 符号表示两个数的二进制表示进行按位与操作。
PS >\$boolean1 = "110110110" PS >\$boolean2 = "010010010" PS >\$int1 = [Convert]::ToInt32(\$boolean1, 2) PS >\$int2 = [Convert]::ToInt32(\$boolean2, 2) PS >\$result = \$int1 -band \$int2 PS >[Convert]::ToString(\$result, 2)	结果为 10010010
-bor	二进制或操作符。示例中 符号表示两个数的二进制表示进行按位或操作。
语法: \$leftValue -bor \$rightValue	
如果左右两侧的相同位置的位有一个是1，则该位被设为1，否则设为0。比如：	
PS >\$boolean1 = "110110110" PS >\$boolean2 = "010010010" PS >\$int1 = [Convert]::ToInt32(\$boolean1, 2) PS >\$int2 = [Convert]::ToInt32(\$boolean2, 2) PS >\$result = \$int1 -bor \$int2 PS >[Convert]::ToString(\$result, 2)	结果为 110110110
-bxor	二进制独占或操作符。
语法: \$leftValue -bxor \$rightValue	
如果左右两侧的相同位置的位有一个是1，则该位被设为1，但是如果都是1，则设为0。比如：	
PS >\$boolean1 = "110110110" PS >\$boolean2 = "010010010" PS >\$int1 = [Convert]::ToInt32(\$boolean1, 2) PS >\$int2 = [Convert]::ToInt32(\$boolean2, 2) PS >\$result = \$int1 -bor \$int2 PS >[Convert]::ToString(\$result, 2)	结果为 100100100
-bnot	二进制非操作符。
语法: -bnot \$value	
将操作符右侧的值按位进行翻转操作，1 变为 0，0 变为 1，比如：	
PS >\$boolean1 = "110110110" PS >\$int1 = [Convert]::ToInt32(\$boolean1, 2) PS >\$result = -bnot \$int1 PS >[Convert]::ToString(\$result, 2)	结果 11111111111111111111111001001001

其他操作符

(英) 詹姆斯·斯科特著
PowerShell 附录 A

PowerShell 还支持一些简单的操作符，如表 A-8 中所示：

表 A-8：其他的 PowerShell 操作符

操作符	含义
-replace	替换操作符 语法： <code>"target" -replace "pattern", "replacement"</code> 返回一个新的字符串，使用 pattern 中的正则表达式与 target 中的内容进行匹配，用 replacement 中的文本替换匹配的内容。 默认地，PowerShell 在进行比较的时候忽略大小写。可以使用 -ireplace 操作符进行严格的大小写比较；使用 -creplace 操作符忽略大小写。 如： <pre>PS >"Hello World" -replace "(.) (.*)", '\$2 \$1' World Hello</pre> 如果 target 代表的是一个数组，那么 -replace 会对数组中的每个元素进行操作。关于正则表达式的更多信息，参见附录 B。
-f	格式化操作符 语法： <code>"Format String" -f Values</code> 返回一个字符串，使用提供的格式化的值替换字符串中的内容，例如： <pre>PS >"{0:n0}" -f 1000000000 1,000,000,000</pre> 格式化字符串中的内容就是 .NET 中 <code>String.Format</code> 方法支持的内容。关于格式化字符串的详细的信息，参见附录 H。
-as	类型转换操作符： 语法： <code>\$value -as [Type]</code> 根据提供的类型对 \$value 进行转换。如果转换是无效的，返回空，例如： <pre>PS >3/2 -as [int] 2 PS >\$result = "Hello" -as [int] PS >\$result -eq \$null True</pre>

比较操作符

PowerShell 提供的比较操作符如表 A-9 中所示，允许你对表达式进行比较。PowerShell 的比较操作符是不区分大小写的。对于需要严格区分大小写的地方，使用 -I 前缀；不需要区分大小写的，使用 -c 前缀。

表 A-9: Windows PowerShell 比较操作符

操作符	含义
-eq	相等符 语法: \$leftValue -eq \$rightValue 对于所有的简单的类型, 如果 \$leftValue 和 \$rightValue 相等, 返回真。 当使用数组的时候, 返回两个数组中相等的元素。 对于其他类型的数据, 如果该类型提供了 Equals() 的方法, 使用这个方法。
-ne	不相等操作符 语法: \$leftValue -ne \$rightValue 对于所有的简单的类型, 如果 \$leftValue 和 \$rightValue 不相等, 返回真。 当使用数组的时候, 返回两个数组中不相等的元素。 对于其他类型的数据, 如果该类型提供了 Equals() 的方法, 使用这个方法。
-ge	大于等于操作符 语法: \$leftValue -ge \$rightValue 对于所有的简单的类型, 如果 \$leftValue 大于等于 \$rightValue, 返回真。 当使用数组的时候, 返回左侧数组中大于等于右侧数组中的元素。 对于其他类型的数据, 如果该类型提供了 Compare() 的方法, 使用这个方法。 如果该方法返回值大于等于零, 则返回真。
-gt	大于操作符 语法: \$leftValue -gt \$rightValue 对于所有的简单的类型, 如果 \$leftValue 大于 \$rightValue, 返回真。 当使用数组的时候, 返回左侧数组中大于右侧数组中的元素。 对于其他类型的数据, 如果该类型提供了 Compare() 的方法, 使用这个方法。 如果该方法返回值大于零, 则返回真。
-lt	小于操作符 语法: \$leftValue -lt \$rightValue 对于所有的简单的类型, 如果 \$leftValue 小于 \$rightValue, 返回真。 当使用数组的时候, 返回左侧数组中小于右侧数组中的元素。 对于其他类型的数据, 如果该类型提供了 Compare() 的方法, 使用这个方法。 如果该方法返回值小于零, 则返回真。
-le	小于等于操作符 语法: \$leftValue -le \$rightValue 对于所有的简单的类型, 如果 \$leftValue 小于等于 \$rightValue, 返回真。

表 A-9: Windows PowerShell 比较操作符 (续)

操作符	含义
-lt	当使用数组的时候，返回左侧数组中小于等于右侧数组中的元素。
-gt	对于其他类型的数据，如果该类型提供了 Compare() 的方法，使用这个方法。
-le	如果该方法返回值小于等于零，则返回真。
-ge	如果该方法返回值大于零，则返回假。
-like	相似操作符
	语法: \$leftValue -like Pattern
	根据提供的模式进行比较，如果匹配上了，返回真。
	当使用数组的时候，返回数组中所有匹配的项。
	-like 操作符支持下面这些简单的通配符：
	? 任意单个字符
	* 零个或多个字符
	[a-b] 指定范围中的任何字符
	[ab] 指定的字符是或 b
	例如：
	PS >"Test" -like "[A-Z]e?[tr]" True
-notlike	非类似运算符
	-like 运算符返回 \$false 时，返回 \$true。
-match	匹配运算符
	语法: "Target" -match Regular Expression
	将目标内容与正则表达式进行匹配，如果匹配成功，返回真。
	当使用数组的时候，返回数组中所有匹配的元素。
	匹配的结果保存到 \$matches 变量中，\$matches 是一个哈希表，用来保存匹配的结果。\$matches[0] 中保存匹配的所有的内容。
	例如：
	PS >"Hello World" -match "(.*)" (.*)" True PS >\$matches[1] Hello
	关于正则表达式的更多信息，参见附录 B
-notmatch	非匹配操作符
	-match 运算符返回 \$false 时，返回 \$true。
	-notmatch 运算符用匹配的结果填充 \$matches 变量。

表 A-9: Windows PowerShell 比较操作符 (续)

操作符	含义
-contains	包含操作符 语法: \$list -contains \$value 如果变量\$list 中包含变量\$value 的值, 返回真。也就是说只要\$list 中有一项与\$value 相等, 返回真。
-not-contains	非包含操作符 -contains 运算符返回 \$false 时, 返回 \$true。
-is	类型操作符 语法: \$leftValue -is [type] 如果 \$leftValue 是 .NET 中定义的类型, 返回真。
-isnot	非类型操作符 -is 运算符返回 \$false 时, 返回 \$true。

条件语句

PowerShell 中的条件语句允许你更改你的脚本中执行的顺序。

if、elseif 和 else 语句

语法:

```
if(condition)
{
    statement block
}
elseif(condition)
{
    statement block
}
else
{
    statement block
}
```

如果条件的计算结果为 \$true, PowerShell 执行该语句块。然后, 它执行 if / elseif / else 语句后面的语句。即使该语句块只包含一个语句, PowerShell 也要求用大括号语句块封闭语句。

注意：有关 PowerShell 如何将表达式作为一个条件请参阅上文中的“简单操作符”和“比较操作符”。

如果条件计算为 \$false，那么 PowerShell 会计算下面（可选）的 elseif 条件，直到有一个匹配为止。如果遇到一个匹配条件，PowerShell 执行与该条件关联的语句块，然后继续执行 if / elseif / else 处语句。

例如：

```
$textToMatch = Read-Host "Enter some text"
$matchType = Read-Host "Apply Simple or Regex matching?"
$pattern = Read-Host "Match pattern"
if($matchType -eq "Simple")
{
    $textToMatch -like $pattern
}
elseif($matchType -eq "Regex")
{
    $textToMatch -match $pattern
}
else
{
    Write-Host "Match type must be Simple or Regex"
}
```

如果没有任何条件计算为 \$true，PowerShell 就执行 else 中的关联语句块，然后执行 if / elseif / else 尾部的其他语句。

switch 语句

```
switch options expression
{
    comparison value          { statement block }
    -or-
    { comparison expression } { statement block }
    ...
    default                  { statement block }
}
```

或：

```
switch options -file filename
{
    comparison value          { statement block }
    -or-
    { comparison expression } { statement block }
    ...
    default                  { statement block }
}
```

当 PowerShell 遇到 switch 语句的时候，会将 switch 中的表达式与正文中的语句进行比较。如果表达式是一个值的列表，PowerShell 会将每个项与 switch 正文中的语句进行比较。如果你指定了 -file 选项，PowerShell 将文件中的每一行当作表达式中的列表的项。

comparison value 语句允许你将当前输入的项与所指定模式进行匹配。默认情况下 PowerShell 不区分大小写完全匹配，但你可以提供给 switch 语句添加选项来更改它，如表 A-10 中所示。

表 A-10：PowerShell 中 switch 语句可选项

选项	含义
-casesensitive	区分大小写的匹配项
-c	使用此选项，只有当前输入的项完全匹配比较值所指定值的时候，PowerShell 才执行关联的语句块。如果当前输入对象是字符串，匹配是区分大小写的。
-exact	完全匹配
-e	使用此选项，只有当前输入的项完全匹配比较值所指定值的时候，PowerShell 才执行关联的语句块。这个匹配是区分大小写的，也是 PowerShell 默认地操作模式。
-regex	正则表达式匹配项
-r	使用此选项，只有当前输入项匹配正则表达式指定的比较值的时候，PowerShell 才执行关联的语句块。这种类型的匹配不区分大小写。
-wildcard	通配符匹配
-w	使用此选项，只有当前输入项与指定的通配符比较值匹配的时候，PowerShell 才执行关联的语句块。 通配符匹配支持下列简单的通配符字符： ? 任何单个未指定的字符 * 零个或多个未指定的字符 [a-b] 指定的范围中的任何字符 [ab] 指定的字符 这种类型的匹配是不区分大小写的。

{比较表达式} 语句可以处理当前输入的项，它存储在任意脚本块的 \$_ 变量中。当处理一个{比较表达式} 语句的时候，只有在{比较表达式} 计算结果为 \$true 的时候，PowerShell 才执行关联的语句块。

如果在 switch 正文中没有其他语句，PowerShell 将执行（可选）默认的关联语句块。

当处理 `switch` 语句的时候, PowerShell 尝试将当前输入的对象与正文中的每个语句进行匹配, 即使一个或多个语句已经匹配, 仍然要执行到下一个语句。如果要 PowerShell 在匹配一个语句后退出 `switch` 语句, 那么在语句块的最后要包含一个 `break` 语句。

例如:

```
$myPhones = "(555) 555-1212", "555-1234"
switch -regex ($myPhones)
{
    { $_.Length -le 8 } { "Area code was not specified"; break }
    { $_.Length -gt 8 } { "Area code was specified" }
    "\((555)\).*" { "In the $($matches[1]) area code" }
}
```

输出如下:

```
Area code was specified
In the 555 area code
Area code was not specified
```

注意: 有关 `break` 语句的详细信息, 请参阅下面的部分“**循环语句**”。

默认情况下 PowerShell 不区分大小写完全匹配, 但你可以提供添加选项给 `switch` 语句来更改它。

循环语句

PowerShell 中的循环语句可以多次执行一组语句。

for 语句

语法:

```
:loop_label for(initialization; condition; increment)
{
    statement block
}
```

当 PowerShell 执行一个 `for` 语句的时候, 它首先执行给定的 `initialization` 语句。接着执行给定的 `condition` 语句, 如果 `condition` 语句执行的结果为 `$true`, PowerShell 会执行 `increment` 中的语句。只要 `condition` 语句为 `$true`, PowerShell 会继续执行 `for` 循环中的语句与 `increment` 语句。

例如：

```
for($counter = 0; $counter -lt 10; $counter++)
{
    Write-Host "Processing item $counter"
}
```

break 和 continue 语句可作为结束循环语句的标记。

foreach 语句

```
:loop_label foreach(variable in expression)
{
    statement block
}
```

当 PowerShell 执行一个 foreach 语句时，它先执行 expression 中的管道命令，如 Get-Process | Where-Object { \$_.Handles -gt 500 } or 1..10。对于 expression 所生成的每一项，指派给 variable 变量，然后执行给定的语句，例如：

```
$handleSum = 0;
foreach($process in Get-Process |
    Where-Object { $_.Handles -gt 500 })
{
    $handleSum += $process.Handles
}
$handleSum
```

break 和 continue 语句可作为结束循环语句的标记。

while 语句

```
:loop_label while(condition)
{
    statement block
}
```

当 PowerShell 执行一个 while 语句的时候，它首先根据给定的 condition 对表达式进行评估。如果表达式评估结果为 \$true，则 PowerShell 执行给定的表达式语句。只要 condition 评估的结果为 \$true，PowerShell 会继续执行表达式语句，例如：

```
$command = "";
while($command -notmatch "quit")
{
    $command = Read-Host "Enter your command"
}
```

break 和 continue 语句可作为结束循环语句的标记。

do ... while 语句

句

```
:loop_label do
{
    statement block
} while(condition)
```

或：

```
:loop_label do
{
    statement block
} until(condition)
```

当 PowerShell 执行一个 do...while 或 do...until 语句的时候，它首先执行给定的语句。在 do...while 声明语句中，只要 condition 评估的结果为 \$true，PowerShell 会继续执行循环语句；在 do...until 语句中，只要 condition 计算为 \$false，PowerShell 会继续执行循环语句，例如：

```
$validResponses = "Yes", "No"
$response = ""
do
{
    $response = read-host "Yes or No?"
} while($validResponses -notcontains $response)
"Got it."

$response = ""
do
{
    $response = read-host "Yes or No?"
} until($validResponses -contains $response)
"Got it."
```

break 和 continue 语句可作为结束循环语句的标记。

流程控制语句

PowerShell 支持两个语句以帮助你控制循环内部的流程：break 和 continue。

break

break 语句终止当前循环的执行，然后 PowerShell 跳到当前的循环语句的结尾处继续执行，就像循环语句正常执行完毕一样。如果你指定了一个具有 break 语句的标签，例如，break :outer_loop，PowerShell 终止会该循环的执行。

例如：

```
:outer for($counter = 0; $counter -lt 5; $counter++)
{
    for($counter2 = 0; $counter2 -lt 5; $counter2++)
    {
        if($counter2 -eq 2)
        {
            break :outer
        }

        Write-Host "Processing item $counter,$counter2"
    }
}
```

输出如下：

```
Processing item 0,0  
Processing item 0,1  
Processing item 1,0  
Processing item 1,1  
Processing item 2,0  
Processing item 2,1  
Processing item 3,0  
Processing item 3,1  
Processing item 4,0  
Processing item 4,1
```

Continue

`continue` 语句跳过执行当前语句块的其余部分。

然后 PowerShell 继续下一个当前执行的循环语句的迭代，就像循环语句正常执行完毕一样。如果你指定了一个具有 `continue` 语句的标签，例如，`continue :outer`，PowerShell 会继续执行该循环的下一个迭代。

例如：

```
:outer for($counter = 0; $counter -lt 5; $counter++)
{
    for($counter2 = 0; $counter2 -lt 5; $counter2++)
    {
        if($counter2 -eq 2) :下取果昔
        {
            continue :outer
        }
        Write-Host "Processing item $counter,$counter2"
    }
}
```

输出如下：

```
Processing item 0,0
Processing item 0,1
Processing item 0,3
Processing item 0,4
Processing item 1,0
Processing item 1,1
Processing item 1,3
Processing item 1,4
Processing item 2,0
Processing item 2,1
Processing item 2,3
Processing item 2,4
Processing item 3,0
Processing item 3,1
Processing item 3,3
Processing item 3,4
Processing item 4,0
Processing item 4,1
Processing item 4,3
Processing item 4,4
```

使用 .NET 框架

微软的功能强大的 .NET 框架，使得 PowerShell 既可以完成系统管理任务，同时又可以用来开发应用程序。

在 PowerShell 中使用 .NET 框架的主要的方式是：调用方法或访问属性。

静态方法

要调用类的静态方法，可以输入：

```
[ClassName]::MethodName(parameter list)
```

例如：

```
PS >[System.Diagnostics.Process]::GetProcessById(0)
```

返回进程 ID 是 0 的进程，结果如下：

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

实例方法

要调用对象实例的方法，可以键入：

```
$objectReference.MethodName(parameter list)
```

例如：

```
PS >$process = [System.Diagnostics.Process]::GetProcessById(0)  
PS >$process.Refresh()
```

这个例子保存进程 ID 是 0 的进程到变量 \$process 中，然后调用对象 \$process 的 Refresh() 方法来刷新进程。

静态属性

要访问一个类的静态的方法，可以键入：

```
[ClassName]::PropertyName
```

或者：

```
[ClassName]::PropertyName = value
```

例如，[System.DateTime] 类提供了一个 Now 的静态属性，可以返回当前时间。

```
PS >[System.DateTime]::Now  
Sunday, July 16, 2006 2:07:20 PM
```

比较少见的，一些类允许你设置静态属性的值。

实例的属性

要访问一个对象实例的属性，可以键入：

```
$objectReference.PropertyName
```

或者：

```
$objectReference.PropertyName = value
```

例如：

```
PS >$today = [System.DateTime]::Now  
PS >$today.DayOfWeek  
Sunday
```

这个例子把当前的日期保存到 \$today 变量中，然后调用 \$today 对象的 DayOfWeek 方法。

了解类型

学习类和类型的两个主要途径是 Get-Member cmdlet 和 .NET 框架子文档。

The Get-Member Cmdlet

要了解一个给定的类支持哪些方法和属性，可以将它重定向到 Get-Member cmdlet，如表 A-11 所示。

表 A-11：使用 Get-Member cmdlet

操作	结果
[typename] Get-Member -Static	返回类的所有静态的方法和属性
\$objectReference Get-Member -Static	按类型返回 \$objectReference 中所有静态的方法和属性
\$objectReference Get-Member	按类型返回 \$objectReference 中所有需要实例的方法和属性。如果 \$objectReference 代表项的集合，那么返回该集合实例和他包含的类型的属性。若要查看实例和一个集合本身的属性，使用 Get-member 的 -InputObject 参数： Get-Member -InputObject \$objectReference
[typename] Get-Member	表示此类型所有实例的方法和属性，是一个 System.RuntimeType 对象

.NET 框架模板

有关 .NET 框架中的类的另一个信息源是文档本身，可通过 <http://www.microsoft.com/china/msdn> 在线搜索文档。

文档典型的格式是这样的：首先是对该类的简单介绍，然后是该类支持的属性和方法的超链接。

注意：要快速搜索类的成员的文档，可以在搜索的时候加入“member”关键字，如下：

classname members

一个类的成员文档列出了它的构造函数、方法、属性和更多内容。使用 S 图标来表示静态的方法和属性。单击成员的名称来获得该成员的更多信息，包括该成员生成对象的类型。

类型快捷方式

当你在输入一个类的名称的时候，PowerShell 允许你使用一个快捷方式，表 A-12 列出了一些比较常用的快捷方式。

表 A-12：PowerShell 类

类型的快捷名	类的全名
[Adsi]	[System.DirectoryServices.DirectoryEntry]
[Hashtable]	[System.Collections.Hashtable]
[PSObject]	[System.Management.Automation.PSObject]
[Ref]	[System.Management.Automation.PSReference]
[Regex]	[System.Text.RegularExpressions.Regex]
[ScriptBlock]	[System.Management.Automation.ScriptBlock]
[Switch]	[System.Management.Automation.SwitchParameter]
[Wmi]	[System.Management.ManagementObject]
[WmiClass]	[System.Management.ManagementClass]
[WmiSearcher]	[System.Management.ManagementObjectSearcher]
[Xml]	[System.Xml.XmlDocument]
[TypeName]	[System.TypeName]

创建类型的实例

尽管一个类的静态方法和属性可以生成对象，有些时候你仍可能需要显式地创建它们，例如：

```
$webClient = New-Object Net.WebClient  
$webClient.DownloadString("http://search.msn.com")
```

若要加载一个程序集，可以使用 System.Reflection.Assembly 类提供的方法。如下：

```
PS >[Reflection.Assembly]::LoadWithPartialName("System.Web")  
GAC Version Location  
--- ---  
True v2.0.50727 C:\WINDOWS\assembly\GAC_32\(\...)\System.Web.dll
```

```
PS >[Web.HttpUtility]::UrlEncode("http://search.msn.com")
http%3a%2f%2fsearch.msn.com
```

警告：方法 LoadWithPartialName 是不适合在要与他人共享的脚本中使用的。因为它加载当前系统中最新的版本的程序集，可能与你开发时使用的程序集的版本不同。要想以最安全的方式加载一个程序集，要在方法 [Reflection.Assembly]::Load() 中使用它的完整的名称。

与 COM 对象进行交互

PowerShell 允许你访问 COM 对象的方法和属性，就像在.NET 框架中与对象交互一样。要与 COM 对象进行交互，可以使用 New-Object 对象，把 Com 对象的 ProgId 传递给 -ComObject 参数（通常缩写为 -Com），如下：

```
PS >$shell = New-Object -Com Shell.Application
PS >$shell.Windows() | Select-Object LocationName, LocationUrl
```

更多有关对系统管理员最有用的 COM 对象的信息，请参阅附录 G。

扩展类型

PowerShell 提供了两种方法，可以允许你向任意的类添加自己的方法和属性，包括 Add-Member cmdlet 和自定义类型扩展文件。

Add-Member cmdlet

Add-Member cmdlet 允许你动态地添加方法、属性和更多的对象。它支持的扩展类型如表 A-13 所示。

表 A-13：Add-Member cmdlet 支持的可以选择的成员类型

成员类型	含义
AliasProperty	该属性为另一个属性定义的别名
	<pre>PS >\$testObject = [PsObject] "Test" PS >\$testObject Add-Member "AliasProperty" Count Length PS >\$testObject.Count 4</pre>
CodeProperty	由 System.Reflection.MethodInfo 定义的属性。 此方法必须是公共的、静态、返回结果 (nonvoid)，并且接受一个类型是 PsObject 的参数。

表 A-13: Add-Member cmdlet 支持的可以选择的成员类型 (续)

成员类型	含义
NoteProperty	由你提供初始值定义的一个属性。
	<pre>PS >\$testObject = [PsObject] "Test" PS >\$testObject Add-Member NoteProperty Reversed test PS >\$testObject.Reversed tseT</pre>
ScriptProperty	由你提供的脚本块定义的一个属性，在脚本块中，\$this 是对当前实例的引用。
	<pre>PS >\$testObject = [PsObject] ("Hi" * 100) PS >\$testObject Add-Member ScriptProperty IsLong { >> \$this.Length -gt 100 >> } >> \$testObject.IsLong >> True</pre>
PropertySet	定义成一组属性的快捷方式的属性。在如 select-Object 这样的 cmdlet 中使用：
	<pre>PS >\$testObject = [PsObject] [DateTime]::Now PS >\$collection = New-Object >> Collections.ObjectModel.Collection`1[System.String] >> \$collection.Add("Month") >> \$collection.Add("Year") >> \$testObject Add-Member PropertySet MonthYear \$collection >> \$testObject select MonthYear >> Year Month ----- 2007 6</pre>
CodeMethod	由 System.Reflection.MethodInfo 定义的一种方法。
	此方法必须是公共的、静态的，并且接受一个类型是 PsObject 的参数。
ScriptMethod	由你提供的脚本块定义的一种方法，在这个脚本块中，\$this 是对当前实例的引用，\$args 是对输入参数的引用。
	<pre>PS >\$testObject = [PsObject] "Hello" PS >\$testObject Add-Member ScriptMethod IsLong { >> \$this.Length -gt \$args[0] >> } >> \$testObject.IsLong(3) >> \$testObject.IsLong(100) >> True False</pre>

自定义类型扩展文件

Add-Member cmdlet 允许你自定义单个对象的属性，PowerShell 还支持通过配置文件来自定义一个给定类型的所有对象。例如，可能要将 Reverse() 方法添加到所有的字符串或将一个 HelpUrl 属性（基于 MSDN Url 别名）添加到所有类型。

PowerShell 将多个类型扩展添加到了文件 *types.ps1xml* 中，该文件位于 PowerShell 的安装目录中。这个文件可以作为扩展文件的一个示例，但你最好不要直接修改这个文件。你可以创建一个新的文件，并使用 Update-TypeData cmdlet 来加载自定义的内容。

下面的命令从与你的配置文件相同的目录加载 *Types.custom.ps1xml*：

```
$typesFile = Join-Path (Split-Path $profile) "Types.Custom.ps1xml"
Update-TypeData -PrependPath $typesFile
```

有关自定义类型扩展文件的更多信息，请参阅 3.12 节。

编写脚本，功能复用

当你准备要打包、重用你的命令的时候，最好把它们放到脚本和函数中。脚本是一个文本文件，文件中包含一系列的 PowerShell 命令。函数也包括一系列的 PowerShell 命令，但通常用在脚本中，来将其划分成较小的、更容易理解的片段。

编写脚本

若要编写一个脚本，可以在文本编辑器中编写你的 PowerShell 命令，并保存成扩展名为 *.ps1* 文件。

运行脚本

有两种方法可以执行一个脚本：调用或 dot-sourcing。

调用

调用一个脚本可以运行其内部的命令。除非显式使用 GLOBAL 范围关键字进行定义，否则，一旦该脚本退出，在脚本中定义的变量和函数也将不再保留。

你可以通过使用 invoke 或调用运算符 (&) 来调用一个脚本，使用脚本的名称作为参数：

```
& "C:\Script Directory\Run-Commands.ps1" Parameters
```

你可以使用绝对路径或相对路径来调用脚本。如果脚本在当前路径下，你必须明确地指出，如下：

```
.\Run-Commands.ps1 Parameters
```

如果路径中没有包含空格，你可以省略这两个引号和调用符号。

Dot-sourcing

Dot-sourcing 可以运行一个脚本内的命令。与调用脚本不同的是，在脚本中定义的变量和函数在脚本退出后可以保留。

你可以通过使用点运算符来调用一个脚本，并提供脚本名称作为参数，如下：

```
. "C:\Script Directory\Run-Commands.ps1" Parameters
```

你可以使用绝对路径或相对路径来调用脚本。如果该脚本位于当前目录中，你必须明确地指出，如下：

```
.\Run-Commands.ps1 Parameters
```

如果路径中没有空格，你可以忽略引号。

注意：默认情况下 PowerShell 中称为执行策略的一项安全功能将防止运行脚本。如果你要启用脚本，则必须更改这个设置。要了解可用的不同的执行策略，可以键入 Get-Help about_signing。一旦选择了一个执行策略，就可以使用 Set-ExecutionPolicy cmdlet 对其进行配置，如下：

```
Set-ExecutionPolicy RemoteSigned
```

为脚本提供输入

PowerShell 提供了几个选项用于处理脚本的输入。

参数数组

要按位置访问命令行的参数，可以使用 PowerShell 特殊变量 \$args 中的参数数组，如下：

```
$firstArgument = $args[0]
$secondArgument = $args[1]
$argumentCount = $args.Count
```

正式参数

```
param([TypeName] $variableName = Default, ...)
```

正式参数可以使你从 PowerShell 的一致的命令行分析引擎中受益。

PowerShell 公开你的参数名（例如，\$variableName），这与它公开 cmdlet 中的参数采用相同的方式。输入参数的时候，用户只需要键入足够的参数名称来消除它和其余参数之间的歧义即可。如果用户未指定参数名，PowerShell 将尝试按输入的位置来分配参数。如果你指定了参数的类型，PowerShell 会确保用户的输入是该类型。如果你指定了默认值，如果用户不为该参数提供输入，那么 PowerShell 将使用默认值。

注意：要强制使用一个参数，要为它定义一个默认值来抛出一个错误，如下：

```
param($mandatory = $(throw "This parameter is required."))
```

管道输入

要访问通过管道传递给你的脚本的数据，可以使用 PowerShell 内置于 \$input 变量中的输入枚举，如下：

```
foreach($element in $input)
{
    "Input was: $element"
}
```

通过输入管道获得的 \$input 变量是一个 .NET 枚举数。枚举数非常有效地支持流方案，但不允许你像访问数组一样访问其中的任意元素。如果你想再次处理其中的元素，一旦到达末尾，必须调用 \$input 枚举数的 Reset() 方法。

如果你需要以一种非结构化 ways 访问该管道中的输入，可以使用下面的命令将输入枚举数转换为一个数组：

```
$inputArray = @($input)
```

在脚本中的 Cmdlet 关键字

当管道输入成为你的脚本中的核心内容的时候，你可能需要用语句块来标记开始 (begin)、进行中 (process) 和结束 (end)，如下：

```
param(...)
begin
{
    ...
}
```

```
    }  
process  
{  
    ...  
}  
end  
{  
    ...  
}
```

PowerShell会在脚本加载的时候执行begin语句中的内容，对管道中的每一项执行process语句；当所有的管道输入都处理完毕后，执行end语句。在process语句块中，变量\$_代表的是当前的管道对象。

当你编写一个包括这些关键字的脚本的时候，你脚本中的所有命令必须包含在语句块中。

\$ MyInvocation 自动变量

在变量\$MyInvocation中包含着脚本运行时的一些上下文信息，包括命令的详细信息，定义它的脚本以及其他信息。

从脚本中检索输出

PowerShell提供了三个主要方法来从脚本中检索输出。

管道输出

语法：any command

脚本的返回值或输出的数据是脚本生成的，但是脚本并不捕获这些数据，如果脚本中包含下面的命令：

```
"Text Output"  
5*5
```

那么该脚本将输出赋值给一个有两个值的数组，这两个值分别是Text Output和25。

返回语句

语法：return value

语句

```
return $false
```

是管道输出的简写方式：

```
$false  
return
```

退出语句

语法：exit *errorlevel*

退出语句从当前脚本或PowerShell实例中返回一个错误代码。如果在脚本中任意的位置调用exit语句（包括嵌入的、在函数里的或某个脚本块中的），则退出该脚本。如果在脚本外调用exit语句，则退出PowerShell。exit语句将自动变量\$LastExitCode设置为*errorLevel*；反过来说，如果*errorLevel*不为零，则设置\$?自动变量为\$false。

注意：参见附录C来获得这方面的更多信息。

函数

语法：

```
function SCOPE:name(parameters)  
{  
    statement block  
}
```

或者：

```
filter SCOPE:name(parameters)  
{  
    statement block  
}
```

函数可以让你把功能相近的命令打包到一个单元里，你可以通过函数名来使用函数。

在函数中，有效的作用域名称包括global（全局）（若要创建可用于整个shell程序的函数）、script（脚本）（若要创建只对当前脚本可用的函数）、local（本地）（若要创建仅供当前范围和子范围的函数）和private（专用）（若要创建只对当前作用域可用的函数）。默认的作用域是local（本地）作用域，遵循与默认变量作用域相同的规则。

函数的语句块的内容遵循与脚本的内容相同的规则。函数支持\$args数组、正式的参数、\$input枚举数、cmdlet关键字、管道输出和等效返回语义。

注意：一个常见的错误是像调用一个方法那样去调用一个函数，如下：

```
$result = GetMyResults($item1, $item2)
```

PowerShell 会像处理脚本或其他命令那样处理函数，因此，正确的写法应该是：

```
$result = GetMyResults $item1 $item2
```

该命令将传递给一个数组到 GetMyResults 函数，其中包含项 \$item1 和 \$item2。

参数的声明，作为参数语句的替代选择，遵循与正式的参数列表相同的语法，但不需要 param 关键字。

一个筛选器是一个简单的函数，其中的语句被当作一个 process 语句块来处理。

注意：你脚本中的命令只能访问那些已经定义了的函数。这通常会导致较大的脚本的开始都是各种帮助函数的声明，这使得脚本比较难以理解。通常可以采用下面的方式使得脚本的内容变得更清晰：

```
function Main
{
    ...
    HelperFunction
    ...
}

function HelperFunction
{
    ...
}

>Main
```

为了使用脚本中的函数，你可以通过调用方式或 dot-source 方式。

直接从文件中调用 .NET API 或自定义命令。在脚本上运行命令时，将从文件中读取并执行。

脚本块

```
$objectReference =
{
    statement block
}
```

PowerShell 支持脚本块，就像使用未命名的函数和脚本一样。与脚本和函数相似，脚本块语句的内容遵循与函数或脚本的内容相同的规则。脚本块支持 \$args 数组、正式的参数、\$input 枚举数、cmdlet 关键字、管道输出和等效返回语义。

与脚本和函数一样，你可以通过调用方式或 dot-source 方式来使用脚本块。因为脚本块没有名称，所以你可以直接调用它或调用包含脚本块的变量。

管理错误

PowerShell 支持两个类型的错误：非终止和终止错误。它将这两种类型的错误收集在自动变量 \$errors 列表中。

大多数错误是非终止错误，它们不会终止执行当前的 cmdlet、脚本、函数或管道。当一个命令输出错误（通过 PowerShell 的 error-output 功能）的时候，PowerShell 将错误写入称为“错误输出流”的流中。你可以使用 write-error cmdlet（或在编写一个 cmdlet 时使用 WriteError() API）来输出一个非终止错误。

\$ ErrorActionPreference 自动变量允许你控制 PowerShell 处理非终止错误的方式。它支持下面的值，如表 A-14 中所示。

表 A-14: ErrorActionPreference 自动变量的可选值

值	含义
SilentlyContinue	不显示错误
Stop	将非终止错误视为终止错误。
Continue	显示错误，但继续执行当前 cmdlet、脚本、函数或管道。这是默认的设置。
Inquire	显示一个提示，询问应该如何处理此错误。

大多数 cmdlet 允许你通过将上面的值之一传递给它的 ErrorAction 参数来显式地配置如何处理错误。

终止错误

一个终止错误将暂停当前 cmdlet、脚本、函数或管道的执行。如果一个命令（如一个 cmdlet 或 .NET 方法调用）生成一个结构化异常（例如，如果你使用它们的一种方法，提供了有效范围之外的参数），PowerShell 会公开为一个终止错误。如果分析你的脚本、函数或管道的一个元素的操作失败，PowerShell 也会产生终止错误。

你还可以使用 throw 关键字在脚本中生成一个终止错误，如下：

throw message

【例】令命令脚本从命令行输入参数

注意：在你自己的脚本和 cmdlet 中，只有当操作的基本目的很难完成的时候才生成终止错误。例如，无法在远程服务器上执行命令应该视为一个非终止错误，但是无法连接到远程服务器完全应将视为是一个终止错误。

如果 PowerShell 遇到终止错误之前，定义了一个 trap 语句，那么它可以截取终止错误，如下：

```
trap [exception type]
{
    statement block
    [continue or break]
}
```

如果你指定一种异常类型，那么 trap 语句仅适用于该类型的终止错误。如果指定了 continue 关键字，当遇到终止错误之后，PowerShell 会继续处理你的脚本、函数或管道的其余部分的代码。

如果指定了 break 关键字，当遇到终止错误之后，PowerShell 会停止处理你的脚本、函数或管道的其余部分的代码。break 是默认模式，如果你既没指定 break 语句也没指定 continue 语句则应用 break 模式。

格式输出

语法：Pipeline | *Formatting Command*

当对象到达输出管道的末尾时，PowerShell 会将它们转换为文本，以使其适合人使用。PowerShell 支持几种选项以帮助你控制此格式的设置过程，如表 A-15 所示。

表 A-15：PowerShell 格式命令

格式化命令	结果
Format-Table <i>Properties</i>	格式化输入对象的属性为一个表，只包括指定的对象属性。如果你没有指定属性的列表，PowerShell 选取一默认组。 除了提供对象的属性，还可以提供高级格式的设置语句，如下： <pre>PS > Get-Process Format-Table -Auto Name,</pre>

表 A-15: PowerShell 格式命令 (续)

格式化命令	结果
<pre>@{Label="HexId"; Expression={ "{0:x}" -f \$_.Id} Width=4 Align="Right" }</pre>	该高级格式设置语句是一个哈希表。用 <code>label</code> 标签和 <code>Expression</code> 表达式（或其中任何短格式）来说明格式。 <code>Expression</code> 表达式项的值应为某个脚本块，返回当前对象（由 <code>\$_</code> 变量代表）的结果。 关于 <code>Format-Table</code> cmdlet 的更多信息，键入 <code>Get-Help Format-Table</code> 。
<code>Format-List Properties</code>	格式输入对象的属性为一个列表，包括你指定对象属性。如果你没有指定属性列表，PowerShell 会选取一默认组。 <code>Format-List</code> cmdlet 支持高级的格式设置语句供 <code>Format-Table</code> cmdlet 使用。 <code>Format-List</code> cmdlet 是你会经常使用的，用于获取一个对象的属性的详细的摘要。命令 <code>Format-List *</code> 返回所有的属性，但不包括那些默认 PowerShell 隐藏的属性。命令 <code>Format-List *-force</code> 返回所有属性。 有关 <code>Format-List</code> cmdlet 更多的信息，键入 <code>Get-help Format-List</code> 。
<code>Format-Wide Property</code>	以非常简洁的摘要视图来设置输入对象的属性格式。如果你没有指定一个属性，PowerShell 会选取默认值。 除了提供对象属性，还可能会提供高级格式的设置语句： <pre>PS >Get-Process Format-Wide -Auto @{ Expression={ "{0:x}" -f \$_.Id} }</pre> 高级格式设置语句是一个哈希表结构。表达式键的值应该是某个脚本块返回当前对象（由 <code>\$_ variable</code> ）的结果。 有关 <code>format-Wide</code> cmdlet 的更多信息，键入 <code>Get-help format-wide</code> 。

自定义格式文件

在 PowerShell 中所有格式的默认值（例如当你不指定一个格式设置命令的时候，或不

指定格式属性的时候) 是由安装目录中的 **.Format.ps1xml* 文件驱动的。类似于 3.12 节中提到的类型扩展文件。

要创建你自己自定义的格式设置, 可以使用这些文件作为参考, 但不要直接修改它们。创建一个新的文件, 可以使用 `Update-FormatData` cmdlet 来加载你的自定义格式文件。`Update-FormatData` cmdlet 适用于在当前 PowerShell 实例中所做的更改。如果你希望每次启动 PowerShell 都加载它们, 可以在你的配置文件脚本中调用 `Update-FormatData`。下面的命令从与你的配置文件相同的目录中加载 *Format.custom.ps1xml*:

```
$formatFile = Join-Path (Split-Path $profile) "Format.Custom.ps1xml"
Update-FormatData -PrependPath $typesFile
```

捕获输出

有几种方法来捕获 PowerShell 中命令的输出, 如表 A-16 所示。

表 A-16: 在 PowerShell 中捕获输出

命令	结果
<code>\$variable = Command</code>	存储 PowerShell 命令所产生的对象到 \$variable。
<code>\$variable = Command Out-String</code>	将 PowerShell 命令以可视化的表示形式存储到 \$variable。PowerShell 命令的输出被转换后更容易供你阅读。
<code>\$variable = NativeCommand</code>	将本机命令的输出结果(字符串)存储到 \$variable。PowerShell 将此存储为字符串列表, 每一行是本机命令的一个输出。
<code>Command -OutVariable variable</code>	大多数的 PowerShell 命令所产生的对象, 可以存储到 \$variable。参数 <code>-OutVariable</code> 也可以缩写为 <code>-Ov</code> 。
<code>Command > File</code>	重定向 PowerShell 输出(或标准的输出或本机命令)到文件, 如果文件已经存在则覆盖该文件。此重定向是不捕获错误的。
<code>Command >> File</code>	重定向 PowerShell 的直观表示(或本机命令的标准输出)到文件, 如果文件存在, 追加到文件中。重定向产生的错误不被捕获。
<code>Command 2> File</code>	将 PowerShell 或本机命令中的错误重定向到文件。如果文件存在, 则覆盖文件。
<code>Command 2>> File</code>	将 PowerShell 或本机命令中的错误重定向到文件。如果文件已经存在, 则追加到文件。

表 A-16：在 PowerShell 中捕获输出（续）

命令	结果
<code>Command > File 2>&1</code>	将错误和PowerShell标准的输出流或本机命令覆盖已有的文件。
<code>Command >> File 2>&1</code>	将错误和PowerShell标准的输出流或本机命令追加到文件中。

跟踪与调试

在 PowerShell 中三个用于跟踪与调试的命令包括：`Set-PsDebug cmdlet`、`Trace-Command cmdlet` 和 `verbose cmdlet` 输出。

Set-PsDebug cmdlet

`Set-PsDebug cmdlet` 可以让你在 PowerShell 中控制跟踪、单步调试和严格模式。表 A-17 列出了 `Set-PsDebug cmdlet` 的参数。

表 A-17：Set-PsDebug cmdlet 的参数

参数	描述
Trace	当运行命令的时候，设置 PowerShell 输出的跟踪详细信息的数量。 值为 1 时，PowerShell 计算它们的输出所有行。 值为 2 将所有行都输出为 PowerShell 计算其，以及有关变量分配、函数调用和脚本的信息。 值为 0 禁用跟踪。
Step	启用或禁用单步调试。当启用的时候，PowerShell 会在执行每条命令的时候提示你。
Strict	启用或禁用严格的模式。当启用的时候，如果你试图访问一个没有定义的变量，PowerShell 会抛出终止错误。
Off	关闭跟踪、调试和严格模式。

Trace-Command Cmdlet

语法：`Trace-Command Command Discovery -PsHost { gci c:\ }`

`Trace-Command cmdlet` 会公开 PowerShell 命令的诊断和支持信息。PowerShell 将诊断信息进行分类，称为跟踪源（trace sources）。

跟踪源的完整列表，可通过 Get-TraceSource cmdlet 来获取。

有关 Trace-Command cmdlet 的更多信息，请键入 Get-help Trace-Command。

详细的 cmdlet 输出

语法：Cmdlet -Verbose

PowerShell 命令可以使用 write-verbose cmdlet（脚本编写）或 WriteVerbose() API（编写一个 cmdlet）生成详细的输出。

自动变量 \$VerbosePreference 允许你控制 PowerShell 处理详细输出。它支持表 A-18 所列的值。

表 A-18

值	含义
SilentlyContinue	不显示详细的输出，这是默认的设置
Stop	将详细输出视为一个终止错误
Continue	显示详细输出，继续执行当前的 cmdlet、脚本、函数或管道
Inquire	显示提示，询问 PowerShell 应该如何处理此详细输出

大多数的 cmdlet 可以显式配置此参数。表 A-18 列出了它的详细参数。

常见的自定义点

PowerShell 提供了几种有用的途径来进行自定义和个性化设置。

控制台设置

Windows PowerShell 的用户界面提供了几个功能以使你的 shell 程序更有效。

调整您的窗口大小

在系统的菜单中（用鼠标右键单击控制台窗口的顶部左侧的 PowerShell 图标），选择属性→版式。窗口大小选项使你可以控制实际的窗口大小（有多大的窗口显示在屏幕上），而屏幕缓冲区大小选项让你可以控制虚拟窗口大小（多少内容窗口可以保存）。如果屏幕缓冲区大小大于实际的窗口大小，控制台窗口更改为包含滚动条。增加虚拟窗口高度

以使 PowerShell 从更早版本的会话中存储更多的输出。如果你从“开始”菜单启动 PowerShell，PowerShell 会采用一些默认值修改窗口大小。

使文本选定内容更容易

在系统菜单中，单击选项→快速编辑模式。快速编辑模式允许你使用鼠标有效地复制和粘贴文本到或超出你的 PowerShell 控制台。如果你从“开始”菜单启动 PowerShell，PowerShell 会启用快速编辑模式。

使用热键来有效的运行 shell 程序

Windows PowerShell 控制台支持多个热键，使控制台操作更有效，如表 A-19 所示。

表 A-19：Windows PowerShell 热键

热键	含义
Windows key + r, and then type powershell	运行 PowerShell
Up arrow	向前搜索历史命令
Down arrow	向后搜索历史命令
Page Up	显示历史命令中的第一个命令
Page Down	显示历史命令中的最后一个命令
Left arrow	在当前命令行中向左移动一位光标
Right arrow	在当前命令行中向右移动一位光标，如果已经到了行尾，会显示上一个命令的一个字符。
Home	将光标移到命令行的开始
End	将光标移到命令行的结尾
Control + left arrow	在命令行上将光标向左移动一个单词
Control+right arrow	在命令行上将光标向右移动一个单词
Alt + space,e,l	滚动屏幕缓冲区
Alt + space,e,f	搜索屏幕缓冲区
Alt + space,e,k	选择要从屏幕缓冲区中复制的文本。
Alt + space,e,p	将剪贴板的内容粘贴到 Windows PowerShell 控制台
Alt + space,c	关闭 PowerShell 控制台
Control+c	中断当前的操作
Control+break	强制关闭 Windows PowerShell 窗口

表 A-19: Windows PowerShell 热键 (续)

热键	含义
Control+home	删除从当前命令行的开始到 (但是不包括) 当前光标位置的字符
Control+end	删除从 (和包括) 当前光标位置到当前命令行的末尾的字符
F1	在命令行上将光标向右移动一个字符。如果到了行结尾, 从该位置插入最后一个命令的一个字符
F2	通过你键入的字符数复制最后一个命令
F3	完成命令行中的最后一个命令的内容, 从当前光标位置到末尾
F4	从光标位开始到你键入的字符 (但不包括) 删除字符
F5	向后扫描你的命令历史记录
F7	以交互方式从你的命令历史记录中选择一个命令。使用箭头键来滚动显示窗口。按 Enter 键以执行命令, 或使用右箭头键把文本放置在命令行上
F8	向后扫描你的命令历史记录, 只显示与你输入的文本匹配的到目前为止你已经在命令行键入的命令
F9	从你的命令历史记录中调用特定编号的命令。这些命令的编号对应于命令历史记录所选内容窗口 (F7) 显示数字
Alt+F7	清除命令的历史记录

注意: 表 A-19 中列出的热键都十分有用。当你通过某一热键程序 (比如丰富的 AutoHotkey, <http://www.autohotkey.com>) 将它们映射到较短的或更直观的键时, 它们会更加有用。

配置文件

Windows PowerShell 在启动过程中会自动运行表 A-20 中的四个脚本。如果存在的话, 每个脚本会允许你自定义你的执行环境。PowerShell 会运行这些文件中的任何命令, 就像你在命令行中手动输入这些命令一样。

表 A-20: Windows PowerShell 配置文件

配置文件的目的	配置文件的位置
自定义所有 PowerShell 的会话，包括所有用户在系统上驻留的所有会话	<code>InstallationDirectory\profile.ps1</code>
为系统中所有用户自定义 PowerShell.exe 会话	<code>InstallationDirectory\Microsoft.PowerShell_profile.ps1</code>
自定义所有的 PowerShell 会话，包括所有用户的宿主应用程序	<code>My Documents\WindowsPowerShell\profile.ps1</code>
典型的自定义的 PowerShell.exe 会话	<code>My Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1</code>

注意：PowerShell 通过定义自动变量 \$profile，使编辑你的配置文件脚本变得简单。

若要创建一个新的配置文件，可以键入：

```
New-Item -Type file -Force $profile
```

要编辑一个配置文件，可以键入：

```
Notepad $profile
```

有关编写脚本的更多信息，请参阅本章前面提到的“编写脚本，复用功能”一节。

提示

若要自定义你的提示，可以将提示函数添加到你的配置文件中。此函数返回一个字符串，例如：

```
function Prompt
{
    "PS [${env:COMPUTERNAME}] >"
```

有关自定义你的提示的更多信息，请参阅 1.3 节。

选项卡

你可以定义一个 TabExpansion 函数，当你按 Tab 键时，你就可以自定义 Windows PowerShell 处理属性、变量、参数和文件的方式。

你的 TabExpansion 函数重写了 PowerShell 定义的默认值，因此你可能希望使用它的定义作为起点：

```
Get-Content function:\TabExpansion
```

作为它的参数，此函数接收整个命令行作为输入，直到命令行的最后一个单词为止。如果该函数返回一个或多个字符串，PowerShell 在选项卡完成循环这些字符串。否则，它会使用其内置的用于选项卡 - 完成文件名、目录名、cmdlet 名称和变量名的逻辑。

附录 B

正则表达式参考

正则表达式在大多数文本分析和匹配任务中起到一个重要的作用。它们构成 -match 运算符、switch 语句、Select-String cmdlet 等的一个重要的基础。表 B-1 到 B-9 列出了经常使用的正则表达式。

表 B-1：字符类：表示字符集的模式

字符类	匹配项
.	除了换行符之外的任何字符。如果正则表达式使用了 SingleLine 选项，则匹配任何字符。 PS >"T" -match '.' True
[characters]	在方括号中的任何字符。例如：[aeiou]。 PS >"Test" -match '[Tes]' True
[^ characters]	不在方括号中的任何字符。例如：[^aeiou]。 PS >"Test" -match '[^Tes]' False
[start-end]	在字符 start 和 end 之间的任何字符。你可以在括号之间包含多个字符范围。例如，[a-eh-j]。 PS >"Test" -match '[e-t]' True
[^ start-end]	不在字符 start 和 end 之间的任何字符。你可以在括号之间包含多个字符范围。例如， PS >"Test" -match '[^e-t]' False
\p{character class}	在由 {字符类} 指定的任何采用 Unicode 编码的组或块范围中的字符。 PS >"+" -match '\p{Sm}' True

表 B-1：字符类：表示字符集的模式（续）

(参) 霍默·卡特秀恩《编程圣经》

字符类	匹配项
\P{character class}	不在由 {字符类} 指定的任何采用 Unicode 编码的组或块范围中的字符。 PS > "+" -match '\P{Sm}' False
\w	任何单词字符 PS > "a" -match '\w' True
\W	任何非单词字符 PS > "!" -match '\W' True
\s	任何空白字符 PS > "\t" -match '\s' True
\S	任何非空白字符 PS > " " -match '\S' False
\d	任何十进制数字 PS > "5" -match '\d' True
\D	任何非十进制数字 PS > "!" -match '\D' True

表 B-2：数量：强制前缀表达式上数量

数量	含义
<none>	一个匹配项 PS > "T" -match 'T' True
*	零个或多个匹配项，尽可能多地匹配 PS > "A" -match 'T*' True PS > "TTTTT" -match '^T*\$' True
+	一个或多个匹配项，尽可能多地匹配 PS > "A" -match 'T+' False PS > "TTTTT" -match '^T+\$' True

表B-2：数量：强制前缀表达式上数量（续）

数量	含义
? 零个或一个匹配，尽可能多地匹配	PS >"TTTTT" -match '^T?\$\$' False
{n} 完全匹配项	PS >"TTTTT" -match '^T{5}\$\$' True
{n, } n 或多个匹配项，尽可能多地匹配	PS >"TTTTT" -match '^T{4, }\$\$' True
{n, m} 在 n 和 m (含) 之间的匹配	PS >"TTTTT" -match '^T{4, 6}\$\$' True
? 零个或多个匹配项，尽可能少匹配	PS >"A" -match '^AT?\$\$' True
+? 一个或多个匹配项，尽可能少匹配	PS >"A" -match '^AT+?\$\$' False
?? 零个或一个匹配，尽可能少匹配	PS >"A" -match '^AT??\$\$' True
{n}? 与 n 完全匹配	PS >"TTTTT" -match '^T{5}?\$\$' True
{n, }? n 或多个匹配项，尽可能少匹配	PS >"TTTTT" -match '^T{4, }?\$\$' True
{n, m}? n 和 m 之间的匹配 (含)，尽可能少匹配	PS >"TTTTT" -match '^T{4, 6}?\$\$' True

表B-3：分组构造：允许你组合字符、模式和其他表达式

分组构造	描述
(text)	捕获匹配括号内的文本。这些捕获分别按基于左括号的顺序编号。 PS >"Hello" -match '^(.*)llo\$\$'; \$matches[1] True He

表B-3：分组构造：允许你组合字符、模式和其他表达式（续）

分组构造	描述
(?<name>)	捕获匹配括号内的文本。这些捕获分别由给出的 name 命名
	PS >"Hello" -match '^(<One>.*)1lo\$'; \$matches.One True He
(?<name1-name2>)	平衡组定义。这是一个高级正则表达式构造，允许你均匀地平衡匹配语句对
(?:)	非捕获组
	PS >"A1" -match '((A B)\d)'; \$matches True
	Name Value --- 2 A 1 A1 0 A1
	PS >"A1" -match '(:A B)\d)'; \$matches True
	Name Value --- 1 A1 0 A1
(?imnsx-imnsx:)	应用或禁用对此组在给定的选项。受支持的选项有：
	i 不区分大小写 m 多行 n 显式捕获 s singleline x 忽略空白
	PS >"Te`nst" -match '(T e.st)' False PS >"Te`nst" -match '(?sx:T e.st)' True
(?=)	零宽度正数提前查找声明。确保给定的模式与右侧匹配，而不实际执行匹配。
	PS >"555-1212" -match '(?=...-)(.*)'; \$matches[1] True 555-1212
(?!)	零宽度负数提前查找声明。确保给定的模式与右侧不匹配，而不实际执行匹配。
	PS >"friendly" -match '(?!friendly)friend' False

表 B-3：分组构造：允许你组合字符、模式和其他表达式（续）

分组构造	描述
(?<<=)	声明后面的零宽度正外观。确保给定的模式与左侧匹配，而不实际执行匹配
(?<!)	声明后面的零宽度负外观。确保给定的模式与左侧不匹配，而不实际执行匹配
(?>)	不回跟踪子表达式。只有此子表达式完全匹配才可以匹配

表 B-4：以原子零宽度声明：限制可能匹配项的位置

Assertion	Restriction
^	匹配必须出现在字符串的开始（或行的开始，如果启用了 Multiline 选项）
\$	匹配必须出现在字符串的末尾（或行的末尾，如果启用了 Multiline 选项）
\A	必须在字符串的开头进行匹配
\Z	匹配必须发生在字符串的结尾，或者在字符串尾的 \n 前
\z	必须在字符串的末尾进行匹配
\G	匹配必须出现在上一个匹配结束的位置。与 System.Text.RegularExpressions.Match.NextMatch() 方法一起使用

表 B-4：以原子零宽度声明：限制可能匹配项的位置（续）

Assertion	Restriction
\b	由非字母数字字符分隔单词中的第一个或最后一个字符匹配必须出现在一个词边界上
\B	匹配必须不发生在任何一个词边界上

表 B-5：替换模式：在正则表达式中使用替换操作

模式	替换
\$number	由组数字 <number> 匹配文本
PS >"Test" -replace "(.)st",'\$1ar' Tear	
\${name}	与组名 <name> 匹配文本
PS >"Test" -replace "(?<pre>.)st",'\${pre}ar' Tear	
\$\$	原义是 \$
PS >"Test" -replace ".","\$\$" # 在一个范围内用 \$ 替换所有 \$	
\$&	整个匹配的一个副本
PS >"Test" -replace "^.*\$",'Found: \$& Found: Test'	
\$`	输入字符串匹配之前的文本
PS >"Test" -replace "est\$",'Te\$' TTeT	
\$`	输入字符串后面匹配的文本
PS >"Test" -replace "^Tes",'Res\$' Restt	
\$+	最后捕获的一个组
PS >"Testing" -replace "(.)ing",'\$+ed' Tested	
\$_	整个输入的字符串
PS >"Testing" -replace "(.)ing",'String: \$_' String: Testing	

表B-6: 替换构造: 使你可以执行任何一个表达式或逻辑

替换构造	描述
与用垂直分隔栏分隔的任何的条款匹配	<pre>PS >"Test" -match '(B T)est' True</pre>
(?(expression)yes no)	如果表达式匹配到目前点, 匹配Yes术语。否则, 匹配No术语。No术语是可选的
	<pre>PS >"3.14" -match '(?(\\d)3.14 Pi)' True PS >"Pi" -match '(?(\\d)3.14 Pi)' True PS >"2.71" -match '(?(\\d)3.14 Pi)' False</pre>
(?(name)yes no)	如果名为 name 捕获组到目前为止具有一个捕获, 匹配yes术语。否则, 匹配no术语。No术语是可选的
	<pre>PS >"123" -match '(?<one>1)?(?(one)23 234)' True PS >"23" -match '(?<one>1)?(?(one)23 234)' False PS >"234" -match '(?<one>1)?(?(one)23 234)' True</pre>

表B-7: 返回引用构造: 引用内的一个捕获组的表达式

Backreference construct	Refers to
\number	表达式中的组数字数
	<pre>PS >" Text " -match '(.)Text\\1' True PS >" Text+" -match '(.)Text\\1' False</pre>
\k<name>	表达式中该组命名的名称
	<pre>PS >" Text " -match '(?<Symbol>.)Text\\k<Symbol>' True PS >" Text+" -match '(?<Symbol>.)Text\\k<Symbol>' False</pre>

表B-8: 其他构造: 修改一个正则表达式的其他表达式

构造	描述
(?imnsx-imnsx)	应用或禁用此表达式的其余部分的给定的选项。受支持的选项有:

- I 不区分大小写
- m 多行
- n 显式捕获

表 B-8: 其他构造: 修改一个正则表达式的其他表达式 (续)

构造	描述
s	单行
x	忽略空白
(#)	嵌入式注释。终止于第一个右括号
# [to end of line]	当正则表达式开启了或略空格选项时，允许启用注释窗体

表 B-9：字符转义符：字符表示另一个字符

转义符	匹配
<ordinary characters>	\$ ^ { [() *? \} 匹配本身以外的字符
\a	一个警报 \u0007
\b	如果在[]字符类中表示一个 Backspace \u0008。在正则表达式中, \b 表示一个单词的边界 (\w 和 \W 字符之间), 在一个[]字符类中, \b 是代表回格。在替换模式中, \b 始终表示一个 Backspace
\t	表示 Tab 键 \u0009
\r	表示回车换行键 \u000D
\v	垂直选项 \u000B
\f	A form feed \u000C
\n	新行 \u000A
\e	Escape 键 \u001B
\ddd	八进制的 ASCII 字符 (最多三个数字)。没有前导零的数字, 如果只有一个数字将被视作为 backreferences, 或者对应于一个捕获组数
\xdd	使用十六进制表示形式 (两个数字) 的 ASCII 字符
\cc	一个 ASCII 控制字符; 例如, \cC 是 control-C
\udddd	使用十六进制形式表示 (四个数字) 一个 Unicode 字符
*	当跟随一个字符, 不能被识别为一个转义字符的时候, 匹配字符。例如, * 是字符 *

附录 C

PowerShell 自动变量

PowerShell 定义并自动填充了一些变量。这些变量让你可以访问关于执行环境、PowerShell 首选项以及其他的信息。

表 C-1 提供了这些自动变量的列表和其含义。

表 C-1: Windows PowerShell 自动变量: Windows PowerShell 自动使用并设置的变量

变量	含义
\$	shell 程序接收到的最后一行的最后一个标记
\$?	最后一次操作的成功 / 故障状态
\$^	shell 程序接收到的最后一行的第一个标记
\$_	一个管道的脚本块中当前管道对象
\$args	参数的数组传递到脚本、函数或脚本块
\$confirmPreference	首选项，在请求确认之前，控制可能会影响的操作级别。 支持值包括 none、low、medium、high。值为 none 禁用确认消息
\$consoleFilename	如果使用了，表示的配置此会话的 PowerShell 控制台文件的文件名
\$currentlyExecutingCommand	在挂起的提示下，表示当前正在执行命令
\$debugPreference	首选项来控制 PowerShell 如何处理一个脚本或 cmdlet 的调试输出。支持的值包括 SilentlyContinue、Continue、Inquire、Stop
\$error	数组包含 shell 程序生成的（终止和非终止的）错误
\$errorActionPreference	首选项来控制 PowerShell 如何处理一个脚本或 cmdlet 的错误输出。支持的值包括 SilentlyContinue、Continue、Inquire 和 Stop

表C-1: Windows PowerShell 自动变量: Windows PowerShell自动使用并设置的变量(续)

变量	含义
\$errorView	首选项, 在 shell 程序中, 控制 PowerShell 如何输出错误。支持普通和分类视图 (错误的一个更简短和分类视图)
\$executionContext	表示的脚本可以访问的由 cmdlet 和提供者使用的常用的 API
\$false	变量, 表示布尔值 False
\$foreach	在 foreach 循环中枚举
\$formatEnumerationLimit	对象输出之前, 限制对象格式和输出
\$home	用户的主目录
\$host	表示脚本可以访问的当前主机和用户接口 API 和实现的详细信息
\$input	在管道的脚本块中, 当前的输入的管道
\$lastExitCode	最后一个命令的退出代码, 可以通过脚本显式设置, 调用本机可执行文件时, 会自动设置
\$logEngineHealthEvent	首选项, 告诉 PowerShell 日志引擎运行状况事件, 如错误和异常。支持的值包括 \$true 和 \$false
\$logEngineLifecycleEvent	首选项, 指出 PowerShell 日志引擎生命周期事件, (例如开始和停止。支持的值包括 \$true 和 \$false)
\$logCommandHealthEvent	首选项, 告诉 PowerShell 日志命令运行状况事件, 如错误和异常, 支持的值包括 \$true 和 \$false
\$logCommandLifecycleEvent	首选项, 指出 PowerShell 日志命令生命周期事件, 如开始和停止。支持的值包括 \$true, 和 \$false
\$logProviderHealthEvent	首选项, 告诉 PowerShell 日志提供程序运行状况事件, 如错误和异常。支持的值包括 \$true 和 \$false
\$logProviderLifecycleEvent	首选项, 指出 PowerShell 日志提供程序生命周期事件, 如开始和停止。支持的值包括 \$true 和 \$false
\$matches	最后一个成功的正则表达式匹配的结果 (通过 -match 运算符)
\$maximumAliasCount	限制多少别名可以定义
\$maximumDriveCount	限制多少个驱动器可被定义的。不包括默认的系统驱动器
\$maximumErrorCount	限制 PowerShell 在 \$error 集合中可以保留的错误
\$maximumFunctionCount	限制可以定义多少功能
\$maximumHistoryCount	限制将保留多少个历史记录项
\$maximumVariableCount	限制可以定义多少变量

表C-1: Windows PowerShell 自动变量: Windows PowerShell自动使用并设置的变量(续)

变量	含义
\$myInvocation	有关运行脚本、函数或脚本块时的上下文的信息,包括有关命令的详细的信息(<code>MyCommand</code>),定义他的脚本(<code>ScriptName</code>)
\$nestedPromptLevel	当前提示的嵌套级别。通过输入一个嵌套的提示符操作递增(例如 <code>\$host.EnterNestedPrompt()</code>)和 <code>exit</code> 语句递减
\$null	变量,表示 <code>Null</code> 的概念
\$ofs	输出字段分隔符。当PowerShell输出列表为字符串时,放在元素之间
\$outputEncoding	向外部进程发送管道数据时,使用的字符编码
\$pid	当前PowerShell实例的进程ID
\$profile	此主机PowerShell配置文件的位置和文件名
\$progressPreference	用于控制PowerShell如何处理脚本或cmdlet输出进度的首选项支持的值包括: <code>SilentlyContinue</code> 、 <code>Continue</code> 、 <code>Inquire</code> 和 <code>Stop</code>
\$psHome	PowerShell的安装位置
\$pwd	当前工作目录
\$shellId	此主机的shell程序标识符
\$stackTrace	最后一个错误详细的栈跟踪信息
\$this	对 <code>ScriptMethods</code> 和 <code>ScriptProperties</code> 中的当前对象的引用。
\$transcript	由 <code>start-Transcript</code> cmdlet 使用的文件名
\$true	变量,表示布尔值 <code>True</code>
\$verboseHelpErrors	首选项,告诉PowerShell在分析格式错误的帮助文件时输出详细的错误信息。支持的值包括: <code>\$true</code> 和 <code>\$false</code>
\$verbosePreference	首选项,用于控制PowerShell如何处理详细的脚本或cmdlet输出。支持的值有: <code>SilentlyContinue</code> 、 <code>Continue</code> 、 <code>Inquire</code> 和 <code>Stop</code>
\$warningPreference	首选项,用于控制PowerShell应如何处理脚本或cmdlet的警告输出。支持的值有: <code>SilentlyContinue</code> 、 <code>Continue</code> 、 <code>Inquire</code> 和 <code>Stop</code>
\$whatifPreference	首选项,控制PowerShell应如何处理脚本或cmdlet调用确认请求。支持的值有: <code>SilentlyContinue</code> 、 <code>Continue</code> 、 <code>Inquire</code> 和 <code>Stop</code>

标准 PowerShell 动词

Cmdlets 和脚本应使用 *Verb-Noun* 语法命名。例如 Get-ChildItem。大多数命令都是如此，但也有极少数例外的，cmdlet 应使用标准 PowerShell 动词。应避免使用与标准相同的同义词或概念。这使管理员能够快速地理解使用一组新名词的 cmdlet。

动词应能表达现在时，名词应该是单数形式。表 D-1 到 D-6 列出了不同类别的标准 PowerShell 动词。

表 D-1：标准 Windows PowerShell 常用的动词

动词	含义	同义词
Add	将资源添加到一个容器，或将元素附加到另一个元素	Append, Attach, Concatenate, Insert
Clear	从容器中移除所有元素	Flush, Erase, Release, Unmark, Unset, Nullify
Copy	将资源复制到另一个名称或容器	Duplicate, Clone, Replicate
Get	检索数据	Read, Open, Cat, Type, Dir, Obtain, Dump, Acquire, Examine, Find, Search
Hide	使显示不可见	Suppress
Join	加入一个资源	Combine, Unite, Connect, Associate
Lock	锁定一个资源	Restrict, Bar
Move	移动一个资源	Transfer, Name, Migrate
New	创建一个资源	Create, Generate, Build, Make, Allocate
Push	将项放到一个堆栈的顶部	Put, Add, Copy
Pop	移除一个堆栈的顶部的项	Remove, Paste
Remove	从一个容器中移除一个资源	Delete, Kill

表 D-1：标准 Windows PowerShell 常用的动词（续）

动词	含义	同义词
Rename	为资源提供一个新的名称	Ren, Swap
Search	在集合中查找一个资源（或有关该资源的摘要信息）；（没有实际检索该资源，但提供的信息可以在检索时使用）	Find, Get, Grep, Select
Select	从一个较大的数据集创建数据的子集	Pick, Grep, Filter
Set	设置数据	Write, Assign, Configure
Show	检索，格式化，并显示信息	Display, Report
Split	将数据分开到较小的元素	Divide, Chop, Parse
Unlock	解锁资源	Free, Unrestrict
Use	应用或将资源与一个上下文关联	With, Having

表 D-2：标准 Windows PowerShell 通信动词

动词	含义	同义词
Connect	源连接到目标	Join, Telnet
Disconnect	断开源与目标的连接	Break, Logoff
Read	获取来自非连接的源的信息	Prompt, Get
Receive	获取连接的源中的信息	Read, Accept, Peek
Send	将信息写入到一个已连接的目标	Put, Broadcast, Mail
Write	将信息写入到一个非连接的目标	Puts, Print

表 D-3：标准 Windows PowerShell 数据动词

动词	含义	同义词
Backup	备份数据	Save, Burn
Checkpoint	创建数据当前状态或其配置的快照	Diff, StartTransaction
Compare	将一个资源与另一个资源比较	Diff, Bc
Convert	当 cmdlet 支持双向转换或转换很多数据类型，从一种表示形式更改为另一种	Change, Resize, Resample
ConvertFrom	从一个主输入到几个受支持的输出转换	Export, Output, Out
ConvertTo	从一个主输出到几个受支持的输入转换	Import, Input, In

表 D-3: 标准 Windows PowerShell 数据动词(续)

动词	含义	同义词
Dismount	从命名空间中的位置分离一个名称实体	Dismount, Unlink
Export	将主要输入的资源存储到一个后备存储区或交换格式	Extract, Backup
Import	从后备存储区或交换格式中创建一个主输出资源	Load, Read
Initialize	准备资源使用，并将其初始化到默认状态	Setup, Renew, Rebuild
Limit	对一个资源应用约束	Quota, Enforce
Merge	从多个数据集创建一个数据实例	Combine, Join
Mount	将一个已命名的实体附加到一个命名空间中的某一位置	Attach, Link
Out	将数据发送到一个终端位置	Print, Format, Send
Publish	使一个资源对其他人可见或了解	Deploy, Release, Install
Restore	将资源还原到一组已预定义条件或设置一个检查点	Repair, Return, Fix
Unpublish	从公共可见性中移除一个资源	Uninstall, Revert
Update	更新或刷新资源	Refresh, Renew, Index

表 D-4: 标准 Windows PowerShell 诊断动词

动词	含义	同义词
Debug	检查资源、诊断操作问题	Attach, Diagnose
Measure	标识由操作消耗的资源或检索有关资源的统计信息	Calculate, Determine, Analyze
Ping	确定资源是否活动和响应。在大多数情况下，这应该是被替换成该谓词，test	Connect, Debug
Resolve	映射简写表示形式到一个更完整的形式	Expand, Determine
Test	验证有效性或资源的一致性	Diagnose, Verify, Analyze
Trace	跟踪资源的活动	Inspect, Dig

表 D-5: 标准 Windows PowerShell 生命周期动词

动词	含义	同义词
Disable	配置一个项不可用	Halt, Hide
Enable	配置项可用	Allow, Permit
Install	放置一个资源到指定的位置中, 还可以选择将他初始化	Setup, Configure
Invoke	调用或启动不能停止的活动	Run, Call, Perform
Restart	停止操作, 并再次启动它	Recycle, Hup
Resume	在已被挂起后, 开始一个操作	Continue
Start	开始活动	Launch, Initiate
Stop	中止活动	Halt, End, Discontinue
Suspend	暂停操作, 但不是终止它	Pause, Sleep, Break
Uninstall	从指定位置移除一个资源	Remove, Clear, Clean
Wait	暂停, 直到预期的事件在发生	Sleep, Pause, Join

表 D-6: 标准 Windows PowerShell 安全动词

动词	含义	同义词
Block	限制对资源的访问	Prevent, Limit, Deny
Grant	授予对资源的访问	Allow, Enable
Revoke	移除对资源的访问	Remove, Disable
Unblock	移除对资源的访问的限制	Clear, Allow

附录 E

选定的.NET 类和它们的使用

表 E-1 到 E-16 提供了到 .NET Framework 中的类型的指针，这可以补充 PowerShell 提供的该功能。有关详细的描述文档，可以搜索 <http://msdn.microsoft.com>。

表 E-1: Windows PowerShell

类	类的详细描述	描述
System.Management.Automation.PSObject	表示一个 PowerShell 对象，可向其添加注释、属性和更多	

表 E-2: 实用工具

类	描述
System.DateTime	表示一个即时时间，通常表示为一个日期和一天的时间
System.Guid	表示全局唯一标识符 (GUID)
System.Math	提供了用于三角、对数，和其他常见的数学函数的常量和静态方法
System.Random	表示随机的生成器，生成一系列符合某些统计数字随机性的需求
System.Convert	将一个基本数据类型转换为另一种基本数据类型
System.Environment	提供有关信息，表示操作，当前环境和平台
System.Console	表示控制台应用程序的标准输入、输出和错误流
System.Text.RegularExpressions.Regex	表示正则表达式

表 E-2: 实用工具 (续)

类	描述
System.Diagnostics.Debug	提供了一组方法和属性, 可以帮助调试你的代码
System.Diagnostics.EventLog	提供了与 Windows 事件日志的交互
System.Diagnostics.Process	提供对本地和远程进程的访问, 使你启动和停止本地系统进程
System.Diagnostics.Stopwatch	提供了一组方法和属性, 可用于准确地测量运行时间
System.Media.SoundPlayer	控件从 .wav 文件中播放声音

表 E-3: 集合和对象实用程序

类	描述
System.Array	提供方法用于创建、操作、搜索和排序数组, 从而在公共语言运行库中充当所有数组的基类
System.Enum	为枚举提供基类
System.String	表示一系列 Unicode 字符的文本
System.Text.StringBuilder	表示可变字符字符串
System.Collections.Specialized.OrderedDictionary	表示可通过键或索引访问的键 / 值对的集合
System.Collections.ArrayList	实现 IList 接口, 使用大小是根据需要动态地增加的数组

表 E-4: .NET Framework

类	描述
System.AppDomain	表示应用程序域, 即应用程序执行的隔离的环境
System.Reflection.Assembly	定义程序集, 是一个可重复使用, 可转换和自描述的公共语言运行库应用程序的构造块
System.Type	表示类型声明: 类类型、接口类型、数组类型、值类型、枚举类型、键入参数、泛型类型定义、打开或关闭构造泛型类型
System.Threading.Thread	创建并控制线程, 设置其优先级, 并获取它的状态

表 E-4: .NET Framework (续)

类	描述
System.Runtime.InteropServices.Marshal	提供了一个方法的集合用于分配非托管的内存，复制非托管的内存块和转换托管到非托管的类型，以及与非托管代码进行交互时使用的其他杂项方法
Microsoft.CSharp.CSharpCodeProvider	提供对 C# 代码生成器的实例的访问和代码编译器

表 E-5: 注册表

类	描述
Microsoft.Win32.Registry	提供在 Windows 注册表中表示根键的 RegistryKey 对象和静态方法来访问键 / 值
Microsoft.Win32.RegistryKey	表示 Windows 注册表中的键级别节点

表 E-6: 输入和输出

类	描述
System.IO.Stream	提供的字节序列的一般视图
System.IO.BinaryReader	作为二进制值读取基元数据类型
System.IO.BinaryWriter	以二进制形式将基元类型写入到一个流
System.IO.BufferedStream	在另一个流上添加一个缓冲层来读取和写入操作
System.IO.Directory	公开用于创建、移动和枚举目录和子目录的静态方法
System.IO.FileInfo	提供实例方法，用于创建、复制、删除、移动、打开文件和在创建 FileStream 对象时的帮助
System.IO DirectoryInfo	公开用于创建、移动和枚举目录和子目录的实例方法
System.IO.File	提供静态方法，用于创建、复制、删除、移动和打开文件，在创建 FileStream 对象时提供辅助功能
System.IO.MemoryStream	创建一个流，将内容存储到内存中
System.IO.Path	在包含文件或目录路径信息的 String 实例上执行操作。这些操作是以跨平台的方式执行

表 E-6: 输入和输出 (续)

类	描述
System.IO.TextReader	表示可读取连续字符的读取器
System.IO.StreamReader	实现一个 TextReader, 从一个特定编码的字节流中读取字符
System.IO.TextWriter	表示可以写入连续字符的编写器
System.IO.StreamWriter	实现一个 TextWriter, 用于以一种特定的编码写入字符到流中
System.IO.StringReader	实现 TextReader, 用于从字符串中读取内容
System.IO.StringWriter	实现 TextWriter, 用于写入信息到一个字符串
System.IO.Compression.DeflateStream	提供方法和属性, 使用 Deflate 算法来压缩和解压缩流
System.IO.Compression.GZipStream	提供方法和属性, 使用 GZip 算法来压缩和解压缩流
System.IO.FileSystemWatcher	侦听文件系统更改通知, 当一个目录中的目录或文件更改时引发事件

表 E-7: 安全

类	描述
System.Security.Principal.WindowsIdentity	表示一个 Windows 用户
System.Security.Principal.WindowsPrincipal	允许代码检查一个 Windows 用户的 Windows 组成员身份
System.Security.Principal.WellKnownSidType	定义了一组常用的安全标识符 (SID)
System.Security.Principal.WindowsBuiltInRole	指定与 IsInRole 一起使用的公共角色
System.Security.SecureString	表示应保持机密的文本。正在使用时, 对文本进行加密, 当不再需要时, 从计算机的内存中删除
System.Security.Cryptography.TripleDESCryptoServiceProvider	定义包装对象来访问加密服务三重 DES 算法的提供程序 (CSP) 版本
System.Security.Cryptography.PasswordDeriveBytes	从使用一个扩展 PBKDF 1 算法的密码派生密钥

表 E-7: 安全 (续)

类	描述
System.Security.Cryptography.SHA1	计算输入数据的 SHA 1 哈希值
System.Security.AccessControl.FileSystemSecurity	表示文件或目录的访问控制和审核安全
System.Security.AccessControl.RegistrySecurity	表示对一个注册表键的 Windows 访问控制安全性

表 E-8: 用户界面

类	描述
System.Windows.Forms.Form	表示一个窗口或对话框, 构成应用程序的用户界面
System.Windows.Forms.FlowLayoutPanel	表示动态布局其内容的一个面板

表 E-9: 图像处理

类	描述
System.Drawing.Image	一个类, 为位图和图元文件类提供功能
System.Drawing.Bitmap	封装 GDI 位图, 由图形图像和他的属性的像素数据组成的。一个位图是一个对象用于处理由像素数据定义的图像

表 E-10: 网络

类	描述
System.Uri	提供了一个对象, 表示统一资源标识符 (URI)。可以方便地访问 URI 的各部分
System.Net.NetworkCredential	为基于密码的身份验证提供凭据包括(如基本、摘要、Kerberos 身份验证和NTLM)
System.Net.Dns	提供了简单的域名解析功能
System.Net.FtpWebRequest	实现一个文件传输协议 (FTP) 客户端
System.Net.HttpWebRequest	提供的特定于 HTTP 实现的 WebRequest 类
System.Net.WebClient	提供常用的方法来发送和接收由 URI 标识的一个资源中的数据
System.Net.Sockets.TcpClient	提供了 TCP 网络服务的客户端连接
System.Net.Mail.MailAddress	表示某个电子邮件发件人或收件人的地址

表 E-10: 网络 (续)

类	描述
System.Net.Mail.MailMessage	表示可以使用 SmtpClient 类发送的电子邮件
System.Net.Mail.SmtpClient	允许应用程序通过使用简单邮件传输协议 (SMTP) 发送电子邮件
System.IO.Ports.SerialPort	表示串行端口资源
System.Web.HttpUtility	在处理 Web 请求的时候，提供编码和解码的方法

表 E-11: XML

类	描述
System.Xml.XmlTextWriter	提供了一种快速、非缓存、正向唯一的方法生成流或包含 XML 数据文件的编写器符合 W3C 可扩展标记语言(XML) 1.0 和 XML 中的命名空间建议
System.Xml.XmlDocument	表示一个 XML 文档

表 E-12: Windows Management Instrumentation (WMI)

类	描述
System.Management.ManagementObject	表示 WMI 实例
System.Management.ManagementClass	表示一个管理类。管理类是一个 WMI 类，如 Win32_LogicalDisk，他可以表示磁盘驱动器，Win32_Process，他表示进程（如 Notepad.exe 的一个实例。此类的成员使你能够使用特定的 WMI 类路径访问 WMI 数据。有关的详细信息，请参阅 http://www.microsoft.com/china/msdn/library 上 Windows Management Instrumentation 文档中“Win32 类”
System.Management.ManagementObjectSearcher	基于一个指定的查询检索 WMI 管理对象的集合。此类是要检索管理信息常用的人口点之一。例如，它可用于枚举系统中所有磁盘驱动器、网络适配器、进程、和其他管理对象，或查询的所有在线的网络连接，被暂停的服务等。实例化后，此类的实例将 ObjectQuery 或其派生的 WMI 查询作为输

表 E-12: Windows Management Instrumentation (WMI) (续)

类	描述
System.Management. ManagementObjectSearcher	入,还可以选择一个表示 WMI 命名空间管理作用域,来执行查询。他也可以在枚举选项中选择其他高级的选项。调用此对象上的 Get方法时,ManagementObjectSearcher 在指定的作用域执行给定的查询,并返回匹配ManagementObjectCollection中的查询条件的管理对象集合
System.Management. ManagementDateTimeConverter	提供方法,将 DMTF 日期时间和时间间隔转换为符合 CLR 的 DateTime 和 TimeSpan 的格式,反之亦然
System.Management. ManagementEventWatcher	基于指定的事件查询,预订临时事件通知

表 E-13: 活动目录

类	描述
System.DirectoryServices. DirectorySearcher	对活动目录执行查询
System.DirectoryServices. DirectoryEntry	在活动目录层次结构中, DirectoryEntry 类封装一个节点或对象

表 E-14: 数据库

类	描述
System.Data.DataSet	表示在内存中缓存的数据
System.Data.DataTable	表示在内存中的数据的一个表
System.Data.SqlClient.SqlCommand	表示针对 SQL Server 数据库来执行的 Transact-SQL 语句或存储过程
System.Data.SqlClient.SqlConnection	表示到 SQL Server 数据库的打开的连接
System.Data.SqlClient.SqlDataAdapter	表示一组数据命令和数据库连接,用于填充 DataSet 和更新 SQL 数据库
System.Data.Odbc.OdbcCommand	表示针对某一 odbc 数据源执行的一个 SQL 语句或存储过程
System.Data.Odbc.OdbcConnection	表示到 odbc 数据源的打开的连接
System.Data.Odbc.OdbcDataAdapter	表示一组数据命令和数据库连接,用于填充 DataSet 和更新 odbc 数据源

表 E-15: 消息队列

卷之三

表 E-16：事务处理

附录 F

WMI 参考

Windows 中的 Windows Management Instrumentation (WMI) 设施向管理员提供了上千个包含感兴趣的信息的类。表 F-1 列出了 WMI 覆盖的类别和子类别，它可以使你对 WMI 类的作用域有一个大概的了解。表 F-2 提供了最有用的 WMI 类的子集。有关一个类别的更多信息，可以搜索 <http://www.microsoft.com/china/msdn> 中的 WMI 文档。

表 F-1：WMI 类类别和子类别

类别	子类
Computer System Hardware	冷却设备、输入的设备、大容量存储、主板、控制器和端口、网络设备、电源、打印、电话、视频和监视器
Operating System	COM、桌面、驱动程序、文件系统、作业对象、内存和页面文件、多媒体音频/视频、网络、操作系统事件、操作系统设置、进程、注册表、计划程序作业、安全、服务、共享、开始菜单、存储、用户、Windows NT 事件日志、Windows 产品激活
WMI Service Management	WMI 配置、WMI 管理
General	安装应用程序、性能计数器、安全描述符

表 F-2：选定的 WMI 类

类	描述
Win32_BaseBoard	表示一个 baseboard，也称为母板或系统板
Win32_BIOS	表示计算机系统的基本输入 / 输出的服务(BIOS)的属性
Win32_BootConfiguration	表示一个 Windows 系统的启动配置

表 F-2：选定的 WMI 类（续）

类	描述
Win32_CDROMDrive	代表 Windows 计算机系统上一个 CD-ROM 驱动器。注意该驱动器的名称不对应于分配给该设备的逻辑驱动器号
Win32_ComputerSystem	表示在 Windows 环境中的计算机系统
Win32_Processor	在运行 Windows 操作系统的计算机上，表示可以解释指令序列的一个设备。在多处理器的计算机上，Win32_Processor 类的一个实例包括每个处理器
Win32_ComputerSystemProduct	表示一个产品。这包括这台计算机系统上使用的软件和硬件
CIM_DataFile	表示已命名的数据集合或可执行代码。目前，提供程序将返回固定和映射逻辑磁盘上的文件。将来，只有本地固定磁盘上文件的实例返回
Win32_DCOMApplication	代表 DCOM 应用程序的属性
Win32/Desktop	代表用户桌面的常用特性。此类的属性可以由用户修改来自定义桌面
Win32/DesktopMonitor	表示连接到计算机系统监视器或显示设备的类型
Win32_DeviceMemoryAddress	表示在 Windows 系统上的一个设备内存地址
Win32_DiskDrive	在运行 Windows 操作系统的计算机上代表一个物理磁盘驱动器。Windows 物理磁盘驱动器的任何接口是此类的子类 (ormember)。该磁盘的功能通过此对象对应于该驱动器的逻辑和管理特性。在某些情况下，这可能不会反映设备的实际物理特性。基于另一个逻辑设备上任何对象将不是此类的成员
Win32_DiskQuota	在 NTFS 文件系统卷上跟踪磁盘空间使用的情况。一个系统管理员 (SA) 可以配置 Windows，以避免用户超出磁盘空间的使用，和在用户超过指定的磁盘空间限制时记录一个事件。
Win32_DMACHannel	一个 SA 也可以在用户超过指定的磁盘空间警告级别时记录一个事件。此类是 Windows XP 中新增的
	表示一个 Windows 系统上的一个直接内存访问 (DMA) 信道。DMA 提供了方法将数据从一种设备移动到内存 (或相反)，而不用微处理器的帮助

表 F-2: 选定的 WMI 类 (续)

类	描述
Win32_DiskController	帮助。在系统板使用 DMA 控制器来处理固定的数据通道。通常，DMA 通道由一个或多个设备共享。每个通道可由一个（且只有一个）设备使用
Win32_Environment	表示 Windows 系统环境或系统环境设置。查询此类返回的环境变量可以在下面位置中找到： HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Sessionmanager\Environment 以及： HKEY_USERS\<user sid>\Environment
Win32_Directory	表示 Windows 系统上的目录项。目录是一种类型文件，该文件逻辑分组数据文件并提供了已分组的文件的路径信息。Win32_Directory 没有包含网络驱动器的目录
Win32_Group	表示有关一个组账户的数据。一个组账户允许更改一组用户的访问权限。示例：Administrators
Win32_IDEController	管理集成的设备电路 (IDE) 控制器设备的功能
Win32_IRQResource	代表在 Windows 系统上的中断请求线 (IRQ) 号。一个中断请求是一个设备或时间关键的事件的程序发送到 CPU 的一个信号。IRQ 可以是基于硬件或基于软件的
Win32_ScheduledJob	表示使用 AT 命令创建的一个作业。Win32_ScheduledJob 类不表示从控制面板中按计划任务向导创建的一个作业。你不能在计划任务 UI 中更改一个 WMI 创建的任务。Windows 2000 和 Windows NT 4.0：你可以使用计划任务 UI 修改你最初由 WMI 创建的任务。但是，尽管此任务是成功的修改了，你不再可以在使用 WMI 来访问此任务
Win32_TaskSchedule	计划服务安排每个作业存储（计划程序可以在重新启动后启动一个作业），并在指定的时间执行。如果在指定的作业时间计算机不是活动的或计划服务未运行，计划服务会在下一天指定的时间运行指定的作业
Win32_TaskTime	作业根据 (UTC) 进行计划，这就意味着一个作业可在任何时区使用

表 F-2：选定的 WMI 类（续）

类	描述
Win32_ScheduledJob	Win32_ScheduledJob 类枚举对象时，返回本地时间与 UTC 的偏移量，创建新的作业时将转换为本地时间。
Win32_LoadOrderGroup	例如，指定在星期一下午 10:30 在波士顿 (PST) 计算机上运行一个作业将在本地 (EST) 星期二上午 1:30 运行。
Win32_LogicalDisk	请注意，客户端必须考虑到在本地计算机操作中是否使用夏时制时间，如果是，还要减去从 UTC 偏移量 60 分钟的偏差
Win32_LogonSession	表示一组定义了执行依存关系的系统服务。因为服务是相互依赖，必须按 Load Order 组定义的顺序加载。这些相关的服务要求先行服务正常工作。此类中的数据保存在该注册表项中：
Win32_CacheMemory	System\CurrentControlSet\Control\GroupOrderList
Win32_LogicalMemoryConfiguration	在 Windows 系统中，表示解析为一个实际本地存储设备的数据源
Win32_PhysicalMemoryArray	描述在登录会话或与一个登录到 Windows NT 或 Windows 2000 的用户关联的会话
WIN32_NetworkClient	代表在计算机系统上的内部和外部缓存内存
	表示在 Windows 系统上内存的布局和可用性。从 Windows Vista 开始，此类已不再可用。
	Windows XP 和 Windows Server 2003：不再支持此类。而是使用 Win32_OperatingSystem 类。
	Windows 2000：此类是可用和受支持的
	表示有关计算机系统物理内存的详细信息。这包括内存设备的数目，可用内存的容量和内存类型，例如，系统或视频内存
	表示 Windows 系统上的网络客户端。网络上任何与系统具有客户端关系的计算机系统是此类的一个子类（或成员）（例如，运行 Windows 2000 Workstation 或 Windows 98 的计算机属于 Windows 2000 域）

表 F-2: 选定的 WMI 类 (续)

类	描述
Win32_NetworkLoginProfile	表示 Windows 系统上一个特定用户的网络登录信息。这包括但不限于密码状态、访问特权、磁盘配额和登录目录路径
Win32_NetworkProtocol	表示 Win 32 计算机系统上的一种协议和其网络特征
Win32_NetworkConnection	表示 Windows 环境中的活动的网络连接
Win32_NetworkAdapter	表示一个运行 Windows 操作系统的计算机上的网络适配器
Win32_NetworkAdapterConfiguration	表示一个网络适配器的属性和行为。这类包含额外的属性和方法，支持独立于网络适配器 TCP/IP 和 Internetworking Packet Exchange (IPX) 协议的管理
Win32_NTDomain	表示一个 Windows NT 域
Win32_NTLogEvent	用于从 Windows NT 事件日志中转换实例。一个应用程序必须具有 SeSecurityPrivilege 权限以从安全事件日志接收事件；否则，会返回“访问被拒绝”到应用程序
Win32_NTEventLogFile	代表一个逻辑文件或 Windows NT 事件的目录。该文件也称为事件日志
Win32_OnBoardDevice	表示内置在主板（系统板）上常用适配器设备
Win32_OperatingSystem	代表运行 Windows 操作系统的计算机上一个操作系统。可以在 Windows 上的进行安装的任何操作系统是一种子类或这个类的成员。Win32_OperatingSystem 是一个 singleton 类。要获取单个实例，使用 @。
	Windows Server 2003、Windows XP、Windows 2000 和 Windows NT 4.0：如果计算机安装了多个操作系统，此类只返回当前活动的操作系统实例
Win32_PageFileUsage	表示在 Win 32 系统用于处理虚拟内存文件交换的文件。从此类实例化对象中包含的信息获得指定页面文件的运行时状态

表 F-2：选定的 WMI 类（续）

类	描述
Win32_PageFileSetting	代表页面文件的设置。此类实例化对象中包含当系统启动时创建文件用到的页面文件参数的信息。此类中的属性可以进行修改并推迟到启动。这些设置是与通过关联类 Win32_PageFileUsage 表示的页面文件的运行时状态是不同的
Win32_DiskPartition	代表 Windows 系统上物理磁盘的一个分区的容量和管理功能。示例：disk #0, Partition #1
Win32_PortResource	代表 Windows 计算机系统上的 I/O 端口。
Win32_PortConnector	表示物理连接端口，如 DB-25 针插头，Centronics 或 PS/2
Win32_Printer	表示连接到运行 Microsoft Windows 操作系统的计算机的一个设备，可以输出一个打印的图像或文本到纸张或其他媒体
Win32_PrinterConfiguration	代表打印机设备的配置。包括以下功能，如分辨率、颜色、字体和方向
Win32_PrintJob	表示由一个 Windows 应用程序生成一个打印作业。运行 Windows 操作系统的计算机上的应用程序的打印命令生成的打印任务是一子类或这个类别的成员
Win32_Process	表示操作系统上的一个进程
Win32_Product	表示由 Windows 安装程序安装的产品。一种产品通常关联到一个安装包。有关安装特定的操作系统的支持或要求的信息，可以访问 http://msdn.microsoft.com ，并搜索“Operating System Availability of WMI Components”
Win32_QuickFixEngineering	表示系统范围的快速修复（QFE）或已应用于当前的操作系统的更新
Win32_QuotaSetting	包含卷的磁盘配额的设置信息
Win32_OSRecoveryConfiguration	表示当操作系统失败时，将从内存中收集信息的类型。这包括启动故障和系统崩溃
Win32_Registry	表示 Windows 计算机系统上的系统注册表
Win32_SCSIController	表示 Windows 系统上的 SCSI 控制器
Win32_PerfRawData_PerfNet_Server	提供监督 WINS 服务器服务通信的性能计数器中的原始数据

表 F-2：选定的 WMI 类（续）

类	描述
Win32_Service	表示运行在 Microsoft Windows 操作系统上的一个服务。一个服务应用程序应该符合服务控制管理器 (SCM) 的接口规则，并可以由用户通过服务控制面板实用程序启动或由一个应用程序使用 Windows API 中服务功能启动。没有用户登录时也可以启动服务
Win32_Share	表示 Windows 系统上的一个共享的资源。这可能是一个磁盘、驱动器、打印机、处理之间的通信或其他共享设备
Win32_SoftwareElement	表示一个软件元素，一个软件功能（一种产品，其中可能包含一个或多个元素的一个不同的子集）的一部分。每个软件元素是在一个 Win32_SoftwareElement 实例中定义，并在 Win32_SoftwareFeatureSoftwareElements 关联类中定义一项功能和 Win32_SoftwareFeature 实例之间的关联。有关安装特定的操作系统的支持或要求的信息，可以访问 http://msdn.microsoft.com ，并搜索“Operating System Availability of WMI Components”
Win32_SoftwareFeature	代表一种产品不同的子集，由一个或多个软件元素组成。每个软件元素是在一个 Win32_SoftwareElement 实例中定义的，一项功能和 Win32_SoftwareFeature 实例之间的关联是在 Win32_SoftwareFeatureSoftwareElements 关联类中定义的。有关安装特定的操作系统的支持或要求的信息，可以访问 http://msdn.microsoft.com ，并搜索“Operating System Availability of WMI Components”
WIN32_SoundDevice	代表 Windows 计算机系统上的声音设备的属性
Win32_StartupCommand	表示计算机系统中一个用户登录时自动运行的命令
Win32_SystemAccount	表示系统账户。系统账户由操作系统和 Windows NT 下运行的服务使用。在 Windows NT 内，有很多服务和进程需要登录内部，例如，Windows NT 安装过程中的的进程。系统账户是为此目的设计的

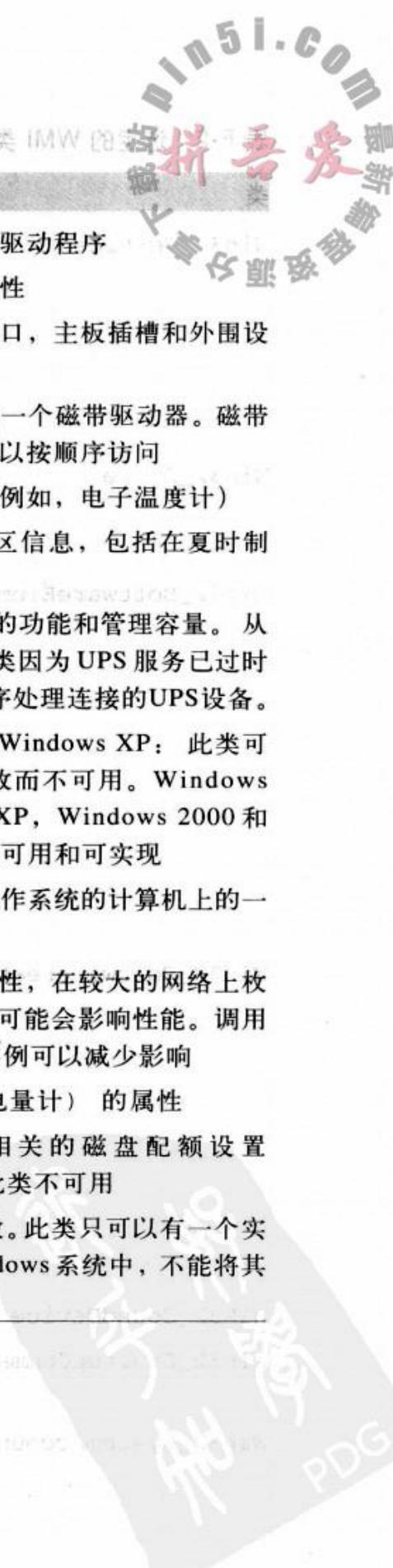


表 F-2：选定的 WMI 类（续）

类	描述
Win32_SystemDriver	表示一项基本服务的系统驱动程序
Win32_SystemEnclosure	代表与物理系统相关的属性
Win32_SystemSlot	表示物理连接点，包括端口，主板插槽和外围设备和专有的连接点
Win32_TapeDrive	代表 Windows 计算机上的一个磁带驱动器。磁带驱动器的主要特点是只可以按顺序访问
Win32_TemperatureProbe	代表温度感应器的属性（例如，电子温度计）
Win32_TimeZone	表示 Windows 系统的时区信息，包括在夏时制时，需更改时间
Win32_UninterruptiblePowerSupply	表示不间断电源（UPS）的功能和管理容量。从 Windows Vista 开始，此类因为 UPS 服务已过时而不在使用。此服务按顺序处理连接的 UPS 设备。 Windows Server 2003 和 Windows XP：此类可用，但因为 UPS 服务失败而不可用。Windows Server 2003, Windows XP, Windows 2000 和 Windows NT 4.0：此类是可用和可实现
Win32_UserAccount	包含有关运行 Windows 操作系统的计算机上的一个用户账户的信息。
Win32_VoltageProbe	因为名称和域都是关键属性，在较大的网络上枚举 Win32_UserAccount 可能会影响性能。调用 GetObject 或查询特定实例可以减少影响 代表电压感应器（电子电量计）的属性
Win32_VolumeQuotaSetting	与一个特定的磁盘卷相关的磁盘配额设置 Windows 2000 /NT 下：此类不可用
Win32_WMISetting	包含 WMI 服务的操作参数。此类只可以有一个实例，始终存在于每个 Windows 系统中，不能将其删除。不能创建其他实例

选定的 COM 对象 和它们的使用

作为可扩展的和管理的接口，许多应用程序都会通过 COM 对象公开有用的功能。虽然 PowerShell 直接处理其中的许多任务，但多个 COM 对象仍提供了重要的价值。表 G-1 列出了一些对系统管理员最有用的 COM 对象。

表 G-1：COM 标识符和说明

标识符	描述
Access.Application	允许自动化 Microsoft Access 和与之的交互
Agent.Control	允许对 Microsoft 代理三维动画字符的控制
AutoItX3.Control	(非默认) 通过 AutoIt 管理工具，提供对 Windows 自动化的访问
CEnroll.CEnroll	提供对证书注册服务的访问
CertificateAuthority.Request	提供对一个证书颁发机构的一个请求的访问
COMAdmin.COMAdminCatalog	提供了访问和管理 Windows COM 编录
Excel.Application	允许自动化 Microsoft Excel 和与之的交互
Excel.Sheet	允许与 Microsoft Excel 工作表之间的交互
HNetCfg.FwMgr	提供对 Windows 防火墙的管理功能的访问
HNetCfg.HNetShare	提供对 Windows 共享连接的管理功能的访问
HTMLFile	允许和一个新的 Internet Explorer 文档交互
InfoPath.Application	允许自动化 Microsoft InfoPath 和与之的交互
InternetExplorer.Application	允许自动化 Microsoft Internet Explorer 和与之的交互
IXSSO.Query	允许与 Microsoft Index Server 的交互
IXSSO.Util	提供对实用程序以及用于 IXSSO.Query 对象的访问

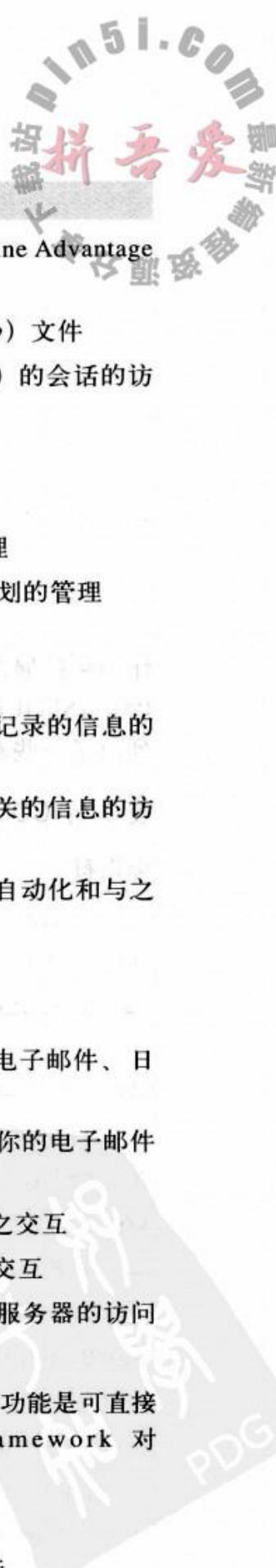


表 G-1: COM 标识符和说明 (续)

标识符	描述
LegitCheckControl.LegitCheck	提供对当前计算机上有关 Windows Genuine Advantage 状态的信息的访问
MakeCab.MakeCab	提供功能来创建和管理 CAB 文件 (.cab) 文件
MAPI.Session	提供对 MAPI (消息应用程序编程接口) 的会话的访问, 如文件夹、邮件和通讯簿
Messenger.MessengerApp	允许 Messenger 的自动化和与之的交互
Microsoft.FeedsManager	允许与 Microsoft RSS 源平台进行交互
Microsoft.ISAdm	提供了对 Microsoft Index Server 的管理
Microsoft.Update.AutoUpdate	提供了对 Microsoft Update 自动更新计划的管理
Microsoft.Update.Installer	允许从 Microsoft Update 安装更新
Microsoft.Update.Searcher	从 Microsoft Update 更新提供搜索功能
Microsoft.Update.Session	提供对有关 Microsoft Update 本地历史记录的信息的访问
Microsoft.Update.SystemInfo	提供对当前系统与 Microsoft Update 有关的信息的访问
MMC20.Application	允许 Microsoft 管理控制台 (MMC) 的自动化和与之进行交互
MSScriptControl.ScriptControl	允许评估和控制 WSH 脚本
Msxml2.XSLTemplate	允许 XSL 转换处理
Outlook.Application	允许和你的 Microsoft Outlook 应用, 如电子邮件、日历、联系人等进行交互和自动化应用
OutlookExpress.MessageList	允许通过 Microsoft Outlook Express 和你的电子邮件交互及自动化
PowerPoint.Application	允许自动化 Microsoft PowerPoint 和与之交互
Publisher.Application	允许自动化 Microsoft Publisher 和与之交互
RDS.DataSpace	提供对远程 DataSpace 业务对象的代理服务器的访问
SAPI.SpVoice	提供对 Microsoft Speech API 的访问
Scripting.FileSystemObject	提供对计算机的文件系统的访问。大多数功能是可直接通过 PowerShell, 或通过 .NET Framework 对 PowerShell 的支持
Scripting.Signer	提供了对 WSH 文件上数字签名的管理
Scriptlet.TypeLib	允许动态的创建脚本类型库 (.tlb) 文件

表 G-1: COM 标识符和说明 (续)

标识符	描述
ScriptPW.Password	允许伪装输入的纯文本密码。如果可能，你应避免使用这个。应该考虑优先使用 <code>read-host cmdlet-AsSecureString</code> 参数
SharePoint.OpenDocuments	允许与 Microsoft SharePoint Services 之间的交互
Shell.Application	提供对 Windows Explorer Shell 应用程序，如管理窗口、文件和文件夹和当前会话的方面的访问
Shell.LocalMachine	提供对有关当前计算机与 Windows shell 程序相关的信息的访问
Shell.User	提供对当前用户的 Windows 会话和配置文件的方面的访问
SQLDMO.SQLServer	提供对 Microsoft SQL Server 的管理功能的访问
Vim.Application	(非默认) 允许和 VIM 编辑器的交互与自动化
WIA.CommonDialog	通过 Windows 图像采集的访问设施对图像捕获
WMPlayer.OCX	允许和 Windows Media Player 的交互与自动化
Word.Application	允许和 Microsoft Word 的交互与自动化
Word.Document	允许与 Microsoft Word 文档进行交互
WScript.Network	提供对联网的 Windows 环境方面的访问，如打印机、网络驱动器，以及计算机和域信息
WScript.Shell	提供对 Windows Shell，例如应用程序、快捷方式、环境变量、注册表和操作环境的方面的访问
WSHController	在远程计算机上允许 WSH 脚本的执行

附录 H

.NET 字符串格式

字符串格式化语法

格式字符串是一个字符串，它包含格式项。每个格式项采用的形式如下：

{index[,alignment][:formatString]}

<index>表示在格式符之后，对象数组中的从零开始的索引的项。

<alignment>是可选的，表示项的对齐方式。正数表示项要右对齐；负数表示项要左对齐。

<formatString>是可选的，使用该类型的特定的格式字符串语法来设置格式（如表 H-1 和 H-2 中的布局）。

标准的数字格式字符串

表 H-1 列出这些标准的数字格式字符串。所有格式说明符后面可能都会跟一个介于 0 和 99 之间的数字用来控制格式设置的精度。

表 H-1：标准的数字格式字符串

格式说明符	名称	描述	例子
C or c	货币	货币数额	PS >"{0:C}" -f 1.23 \$1.23
D or d	十进制	一个十进制量（对于整数类型）。精度说明符控制结果的最小位数	PS >"{0:D4}" -f 2 0002

表 H-1：标准的数字格式字符串（续）

格式说明符	名称	描述	例子
E or e	科学	科学型（指数）表示法。精度说明符控制位小数点之后的数	PS >"{0:E3}" -f [Math]::Pi 3.142E+000
F or f	固定的点	固定的点表示法。精度说明符控制小数点后位的个数	PS >"{0:F3}" -f [Math]::Pi 3.142
G or g	常规	数的最简洁的表示形式（介于定点和科学之间）。精度说明符控制有效的位数	PS >"{0:G3}" -f [Math]::Pi 3.14 PS >"{0:G3}" -f 1mb 1.05E+06
N or n	数字	人可读形式的数，其中包括数字组之间的分隔符。精度说明符控制小数点之后的位数	PS >"{0:N4}" -f 1mb 1,048,576.0000
P or p	百分比	表示为百分比的数（通常介于 0 和 1）。精度说明符控制小数点之后的位数	PS >"{0:P4}" -f 0.67 67.0000 %
R or r	往返过程	只有 Single 和 Double 类型支持此格式。往返过程说明符保证转换为字符串的数值再次被分析为相同的数值	PS >"{0:R}" -f (1mb/2.0) 524288 PS >"{0:R}" -f (1mb/9.0) 116508.4444444444
X or x	十六进制	数转换为十六进制数字字符串。大小写说明符控制生成的十六进制数字的大小写。精度说明符控制生成的字符串中的位数最小数	PS >"{0:X4}" -f 1324 052C

自定义数字格式字符串

你可以使用表 H-2列出的自定义的数字字符串，来设置标准格式字符串不支持的数字格式。

表 H-2：自定义数字格式字符串

格式说明符	名称	描述	例子
0	零占位符	一个数字字符串指定了精度和宽度。零不匹配原始数中的数字位，输出为零	PS >"{0:00.0}" -f 4.12341234 04.1

表 H-2：自定义数字格式字符串（续）

(参) 定语言友孙客数的数

格式说明符	名称	描述	例子
#	数字占位符	一个数字字符串指定了精度和宽度。#不匹配输入数中的数字，不输出	PS >"{0:##.##}" -f 4.12341234
.	小数点	确定小数点分隔符的位置	PS >"{0:##.##}" -f 4.12341234
,	千位分隔符	在设置字符串中，如果放置在零或数字占位符之间，小数点之前的时候，添加数字组之间的分隔符	PS >"{0:#,##.##}" -f 1234.121234
,	编号，(或)	在设置字符串中，如果放置在格式文本(或隐式)小数点之前时，将输入除以1000。你可能会不止一次应用此格式说明符	PS >"{0:#,,,.000}" -f 1048576
,	缩放比例	格式文本(或隐式)小数点之后时，将输入除以1000。你可能会不止一次应用此格式说明符	PS >"{0:#,.##}" -f 1.049
%	百分比	将输入乘以100，并插入百分号，如格式说明符中所示	PS >"{0:%##.000}" -f .68
E0	科学表示法	以科学记数法显示输入。零的数遵循E定义指数字段的最小长度	PS >"{0:##.##E000}" -f 2.71828
E+0			27.2E-001
E-0			27.2E-001
e0			27.2E-001
e+0			27.2E+000
e-0			27.2E-001
'text'	文本字符串	插入按原义提供的文本到输出，而不会影响格式	PS >"{0:#.00'##'}" -f 2.71828
"text"			2.72##
;	节分隔符	允许条件格式。如果你的格式说明符没有包含NEG节分隔符，那么格式设置语句适用于所有输入。如果你的格式说明符包含一个分隔符(两个节)，那么第一节适用于正数和零。第二个节适用于负数。如果你的格式说明符包含两个分隔符(创建三个部分)，那么节应用于正数、负数和零	PS >"{0:POS;NEG;ZERO}" -f -14

表 H-2：自定义数字格式字符串（续）

操作	说明	例子
Other 字符	插入按原义提供的文本到 输出，而不会影响格式	PS >"{0:\$## Please}" -f 14 \$14 Please

左奇棋曰 ТЭИ.

书中为得到这个小面出版于《礼仪学通纂》(1745)的全部内容。参见注释16。

实验室动物实验动物学教材 · 1-1

附录 I

.NET 日期格式

DateTime 格式字符串将 DateTime 对象转换为表 I-1 中列出的几个标准格式中的一种。

表 I-1：标准的 DateTime 格式字符串

格式说明符	名称	描述	例子
d	短日期	区域性的短日期格式	PS >"{0:d}" -f [DateTime] "01/23/4567" 1/23/4567
D	长日期	区域性的长日期格式	PS >"{0:D}" -f [DateTime] "01/23/4567" Friday, January 23, 4567
f	完整的日期 / 短时间	组合的长日期和短时间格式模式	PS >"{0:f}" -f [DateTime] "01/23/4567" Friday, January 23, 4567 12:00 AM
F	完整的日期 / 长时间	组合的长日期和长时间格式模式	PS >"{0:F}" -f [DateTime] "01/23/4567" Friday, January 23, 4567 12:00:00 AM
g	常规日期 / 短时间	合并的短日期和短时间格式模式	PS >"{0:g}" -f [DateTime] "01/23/4567" 1/23/4567 12:00 AM
G	常规日期 / 长时间	合并的短日期和长时间格式模式	PS >"{0:G}" -f [DateTime] "01/23/4567" 1/23/4567 12:00:00 AM
M 或 m	月 / 天	区域性的月 / 天格式	PS >"{0:M}" -f [DateTime] "01/23/4567" January 23
o	转换日期 / 时间	将日期格式化成原始的 DateTime	PS >"{0:o}" -f [DateTime] "01/23/4567" 4567-01-23T00:00:00.0000000
R 或 r	RFC1123	标准的 RFC1123 格式	PS >"{0:R}" -f [DateTime] "01/23/4567" Fri, 23 Jan 4567 00:00:00 GMT
s	可排序的	可排序的格式，符合 ISO 8601 标准，提供适于排序的输出	PS >"{0:s}" -f [DateTime] "01/23/4567" 4567-01-23T00:00:00

表 I-1：标准的 DateTime 格式字符串（续）

格式说明符	名称	描述	例子
t	短时间	区域性的短时间格式	PS >"{0:t}" -f [DateTime] "01/23/4567" 12:00 AM
T	长日期	区域性的长时间格式	PS >"{0:T}" -f [DateTime] "01/23/4567" 12:00:00 AM
u	通用的可排序的格式	应用于该 UTC 区域性的通用的日期格式	PS >"{0:u}" -f [DateTime] "01/23/4567" 4567-01-23 00:00:00Z
U	通用的格式	区域性的完整的日期格式应用于与输入等效的 UTC	PS >"{0:U}" -f [DateTime] "01/23/4567" Friday, January 23, 4567 8:00:00 AM
Y 或 y	年 / 月	区域性的年 / 月 格式	PS >"{0:Y}" -f [DateTime] "01/23/4567" January, 4567

自定义 DateTime 格式字符串

你可以使用表 I-2 列出的自定义的 DateTime 格式字符串，以标准格式字符串中不支持的方式来格式日期。

注意：除非和其他格式说明符一起使用，否则单字符格式说明符一般被解释为一种标准 DateTime 格式字符串。

表 I-2：自定义 DateTime 格式字符串

格式说明符	描述	例子
d	月的日期用数字 1 到 31 之间的数字表示，表示一位的日期前没有前导零	PS >"{0:d dd ddd dddd}" -f [DateTime] "01/02/4567" 2 02 Fri Friday
dd	月的日期用数字 1 到 31 之间的数字表示，表示一位的日期前有前导零	PS >"{0:d dd ddd dddd}" -f [DateTime] "01/02/4567" 2 02 Fri Friday
ddd	一周中的天采用缩写的名称	PS >"{0:d dd ddd dddd}" -f [DateTime] "01/02/4567" 2 02 Fri Friday

表 I-2：自定义 DateTime 格式字符串（续）

格式说明符	描述	例子
ddd	一周中的天采用全名	PS > "{0:d dd ddd dddd}" -f [DateTime] "01/02/4567" 2 02 Fri Friday
f	秒的部分保留一位数字（毫秒）	PS > "{0:f ff fff ffff}" -f [DateTime] "01/02/4567" 0 00 000 0000
ff	秒的部分保留两位数字（毫秒）	PS > "{0:f ff fff ffff}" -f [DateTime] "01/02/4567" 0 00 000 0000
fff	秒的部分保留三位数字（毫秒）	PS > "{0:f ff fff ffff}" -f [DateTime] "01/02/4567" 0 00 000 0000
ffff	秒的部分保留四位数字（毫秒）	PS > "{0:f ff fff ffff}" -f [DateTime] "01/02/4567" 0 00 000 0000
fffff	秒的部分保留五位数字（毫秒）	PS > "{0:fffff ffffff fffffff}" -f [DateTime] "01/02/4567" 00000 000000 0000000
ffffff	秒的部分保留六位数字（毫秒）	PS > "{0:fffff ffffff fffffff}" -f [DateTime] "01/02/4567" 00000 000000 0000000
fffffff	秒的部分保留七位数字（毫秒）	PS > "{0:fffff ffffff fffffff}" -f [DateTime] "01/02/4567" 00000 000000 0000000
F	秒的部分保留一位数字（毫秒），如果为零，则显示空	PS > "{0:F FF FFF FFFF}" -f [DateTime] ::Now 6 66 669 6696
		PS > "{0: F FF FFF FFFF }" -f [DateTime] "01/02/4567"
FF	秒的部分保留二位数字（毫秒），如果为零，则显示空	PS > "{0:F FF FFF FFFF}" -f [DateTime] ::Now 6 66 669 6696
		PS > "{0: F FF FFF FFFF }" -f [DateTime] "01/02/4567"

表 I-2：自定义 DateTime 格式字符串（续）

格式说明符	描述	例子
FFF	秒的部分保留三位数字 (毫秒), 如果为零, 则显示空	PS >"{0:F FF FFF FFFF}" -f [DateTime] ::Now 6 66 669 6696
FFFF	秒的部分保留四位数字 (毫秒), 如果为零, 则显示空	PS >"{0: F FF FFF FFFF }" -f [DateTime] "01/02/4567"
FFFFFF	秒的部分保留五位数字 (毫秒), 如果为零, 则显示空	PS >"{0:F FF FFF FFFF}" -f [DateTime] ::Now 6 66 669 6696
FFFFFFF	秒的部分保留六位数字 (毫秒), 如果为零, 则显示空	PS >"{0: FFFFF FFFFFF FFFFFFFF }" -f [DateTime] "01/02/4567"
FFFFFFFF	秒的部分保留七位数字 (毫秒), 如果为零, 则显示空	PS >"{0:FFFFF FFFFFF FFFFFFF}" -f [DateTime] ::Now 1071 107106 1071068
FFFFFFFFF	秒的部分保留七位数字 (毫秒), 如果为零, 则显示空	PS >"{0: FFFFF FFFFFF FFFFFFF }" -f [DateTime] "01/02/4567"
%g or gg	纪元 (即, A.D.)	PS >"{0:gg}" -f [DateTime] "01/02/4567" A.D.
%h	小时, 数字介于 1 和 12 之间。 单个数字不包含一个前导零	PS >"{0:%h}" -f [DateTime] "01/02/4567 4:00pm" 4

表 I-2：自定义 DateTime 格式字符串（续）

(类) System.DateTimeFormatInfo 义

格式说明符	描述	例子
hh	小时，为 01 和 12 之间的数字。 单个数字包括一个前导零。 注意：这将被解释为标准的 DateTime 格式化字符串，除非 使用其他格式说明符。	PS > "{0:hh}" -f [DateTime] "01/02/4567 4:00pm" 04
%H	小时，为 0 到 23 之间的数字。 单个数字不包含一个前导零。	PS > "{0:%H}" -f [DateTime] "01/02/4567 4:00pm" 16
HH	小时，为 00 到 23 之间的数字。 单个数字包括一个前导零。	PS > "{0:%H}" -f [DateTime] "01/02/4567 4:00am" 04
K	DateTime.Kind 说明符对应于 输入日期的类型（比如，本地、 Utc，或未指定）	PS > "{0: K}" -f [DateTime]::Now.ToUniversalTime() Z
M	分钟，在数字 0 和 59 之间。单 个数字不包含一个前导零。	PS > "{0: m}" -f [DateTime]::Now 7
mm	分钟，为 00 和 59 之间数字。 单个数字包括一个前导零。	PS > "{0:mm}" -f [DateTime]::Now 08
M	月份，介于 1 和 12 之间的数字。 单个数字不包含一个前导零。	PS > "{0:M MM MMM MMMM}" -f [DateTime] "01/02/4567" 1 01 Jan January
MM	月份，为 01 和 12 之间数字。 单个数字包括一个前导零	PS > "{0:M MM MMM MMMM}" -f [DateTime] "01/02/4567" 1 01 Jan January
MMM	缩写的月份名称	PS > "{0:M MM MMM MMMM}" -f [DateTime] "01/02/4567" 1 01 Jan January
MMMM	完整月份名称	PS > "{0:M MM MMM MMMM}" -f [DateTime] "01/02/4567" 1 01 Jan January
s	秒数，在 0 和 59 之间的数字。 单个数字不包含一个前导零	PS > "{0:s ss t tt}" -f [DateTime]::Now 3 03 A AM
ss	秒数，为 00 和 59 之间数字。 单个数字包括一个前导零	PS > "{0:s ss t tt}" -f [DateTime]::Now 3 03 A AM

表 I-2：自定义 DateTime 格式字符串（续）

(起)串转字符串 emiTetsG

格式说明符	描述	例子
T	a.m./p.m 指示符的第一个字符	PS > "{0:s ss t tt}" -f [DateTime]::Now 3 03 A AM
tt	a.m./p.m 指示符	PS > "{0:s ss t tt}" -f [DateTime]::Now 3 03 A AM
Y	年份, (最多) 2 位数	PS > "{0:y yy YYY YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567
yyy	年份, (最多) 3 位数	PS > "{0:y yy YYY YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567
yyyy	年份, (最多) 4 位数	PS > "{0:y yy YYY YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567
YYYY	年份, (最多) 5 位数	PS > "{0:y yy YYY YYYY YYYYY}" -f [DateTime] "01/02/4567" 67 67 4567 4567 04567
Z	从 GMT 时区偏移量。没有包含一个前导零	PS > "{0:z zz zzz}" -f [DateTime]::Now -7 -07 -07:00
zz	从 GMT 时区偏移量。包含一个前导零	PS > "{0:z zz zzz}" -f [DateTime]::Now -7 -07 -07:00
zzz	从 GMT 时区偏移量, 以小时和分钟数	PS > "{0:z zz zzz}" -f [DateTime]::Now -7 -07 -07:00
:	时间分隔符	PS > "{0:y/m/d h:m:s}" -f [DateTime] "01/02/4567 4:00pm" 67/0/2 4:0:0
/	日期分隔符	PS > "{0:y/m/d h:m:s}" -f [DateTime] "01/02/4567 4:00pm" 67/0/2 4:0:0
"text"	插入按原义提供的文本到输出, 而不会影响格式	PS > "{0:'Day: 'ddd}" -f [DateTime]::Now Day: Monday
%c	允许自定义格式的单字符说明符的语法。%符号不会添加到输出	PS > "{0:%h}" -f [DateTime] "01/02/4567 4:00pm"

表 1-2：自定义 DateTime 格式字符串（续）